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

BROWN

Maurice Herlihy

CS176

Fall 2005

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 paralellizable

Coarse-Grained Synchronization

- Each method locks the object
  - Avoid contention using queue locks
  - Easy to reason about
    - In simple cases
  - Standard Java model
    - Synchronized blocks and methods
- So, are we done?

Coarse-Grained Synchronization

- Sequential bottleneck
  - All threads "stand in line"
- Adding more threads
  - Does not improve throughput
  - Struggle to keep it from getting worse
- So why even use a multiprocessor?
  - Well, some apps inherently parallel …

This Lecture

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

First:
Fine-Grained Synchronization

- Instead of using a single lock ..
- Split object into
  - Independently-synchronized components
- Methods conflict when they access
  - The same component …
  - At the same time

Second: Optimistic Synchronization

- Object = linked set of components
- Search without locking …
- If you find it, lock and check …
  - OK, we are done
  - Oops, try again
- Evaluation
  - cheaper than locking
  - mistakes are expensive

Third: Lazy Synchronization

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

Fourth: Lock-Free Synchronization

- Don't use locks at all
  - Use compareAndSet() & relatives …
- Advantages
  - Robust against asynchrony
- Disadvantages
  - Complex
  - Sometimes high overhead

Linked List

- Illustrate these patterns …
- Using a linked-list class
  - Common application
  - Building block for other apps

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-Based Sets

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

Add object to set

List-Based Sets

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

Remove object from set

List-Based Sets

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

Is object in set?

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;
}
```

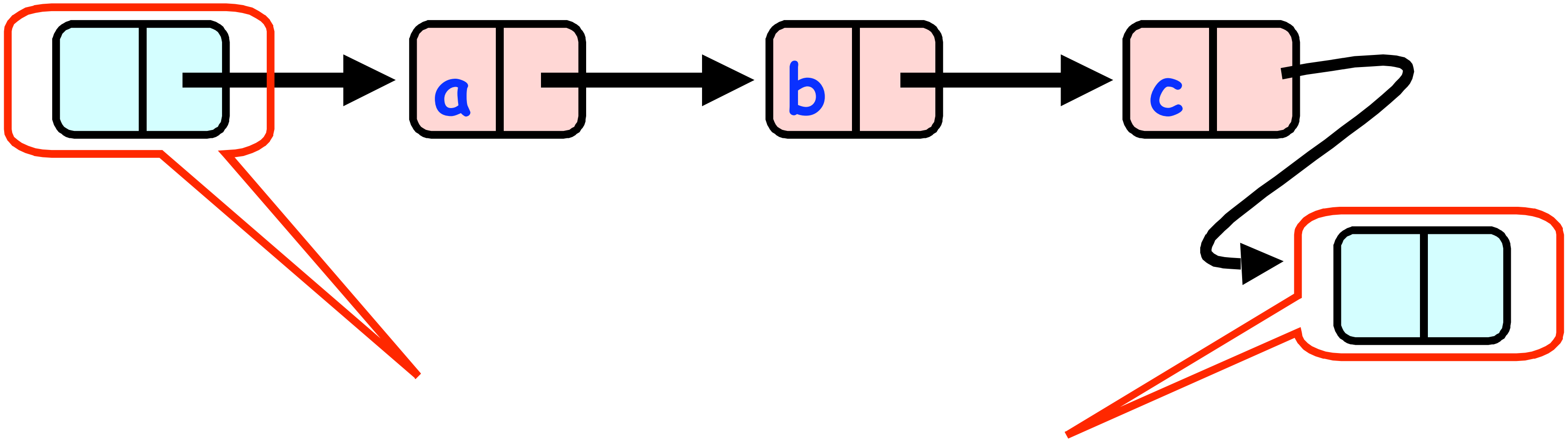
Sorting makes it
easy to detect absence

List Entry

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

Reference to next entry

List-Based Set



Sentinel nodes

(min & max possible keys)

Reasoning about Concurrent Objects

- Invariant
  - Property that always holds
- Established by
  - 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

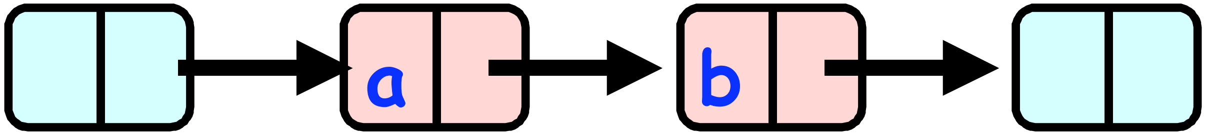
- Proof that invariants preserved works only if
  - methods considered
  - are the only modifiers
- Language encapsulation helps
  - List entries not visible outside class

Interference

- Freedom from interference neeeded even for removed entries
  - Some algorithms traverse removed entries
  - Careful with **malloc()** & **free()!**
- Garbage-collection helps here

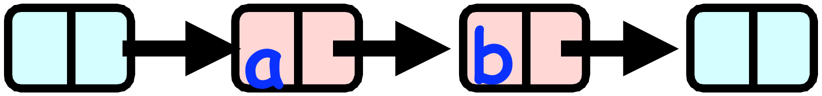
Abstract Data Types

- Concrete representation



- Abstract Type
  - {a, b}

Abstract Data Types

- Meaning of rep given by abstraction map
  - S() = {a,b}

Rep Invariant

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

Blame Game

- Rep invariant is a contract
- Suppose
  - **add()** leaves behind 2 copies of x
  - **remove()** removes only 1
- Which one is incorrect?

Blame Game

- Suppose
  - **add()** leaves behind 2 copies of x
  - **remove()** removes only 1
- Which one is incorrect?
  - If rep invariant says no duplicates
    - **add()** is incorrect
  - Otherwise
    - **remove()** is incorrect

Shorthand

- a → b means a.next = b
- a ⇒ b means b reachable from a
  - a ⇒ a
  - If a ⇒ b and b → c then a ⇒ c

Rep Invariant (partly)

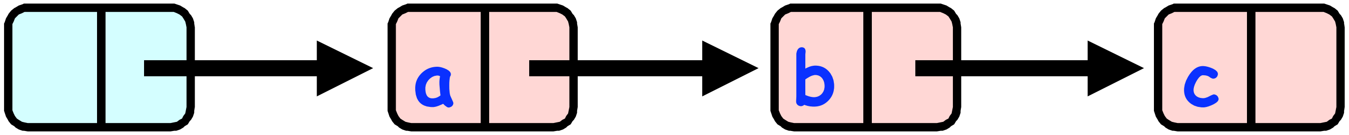
- Sentinel nodes
  - head ⇒ tail
- Sorted, no duplicates
  - If a → b then a.key < b.key

Abstraction Map

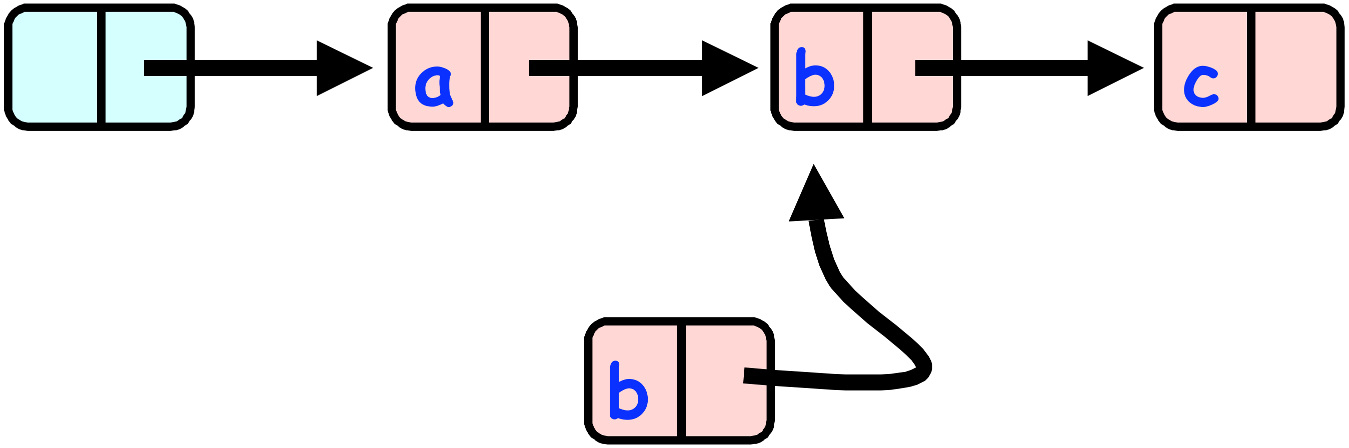
- S(head) =
  - { x | there exists a such that
    - head ⇒ a and
    - a.object = x
  - }

Adding an Entry

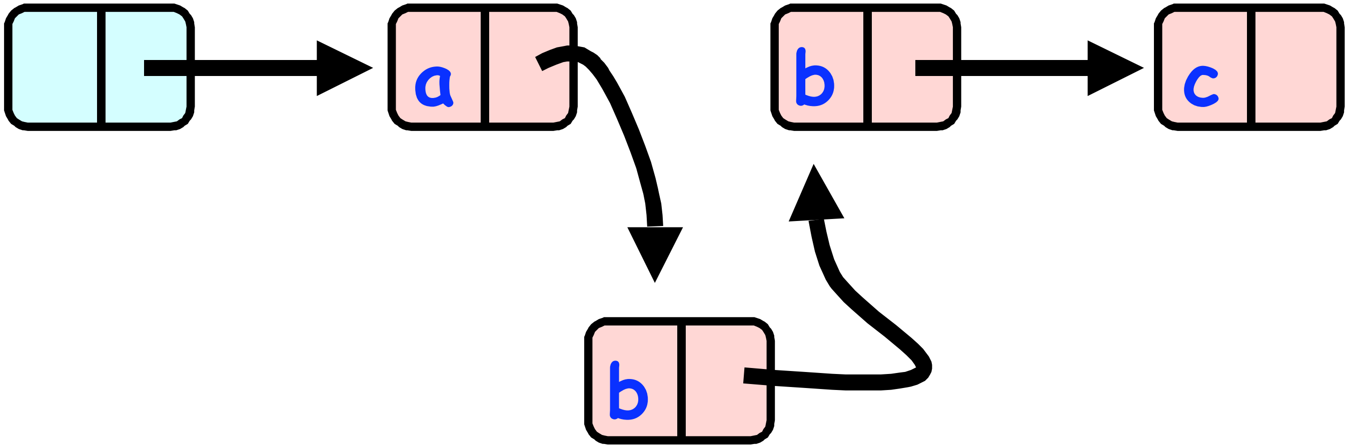
Adding an Entry



Adding an Entry

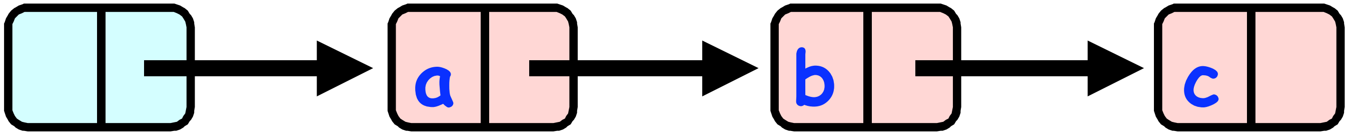


Adding an Entry

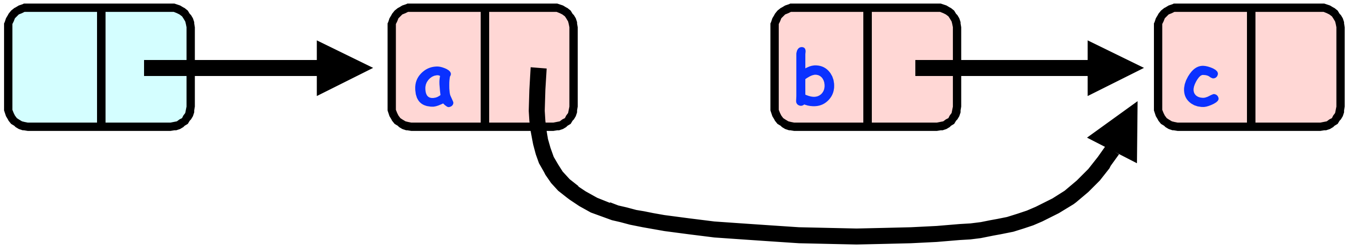


Removing an Entry

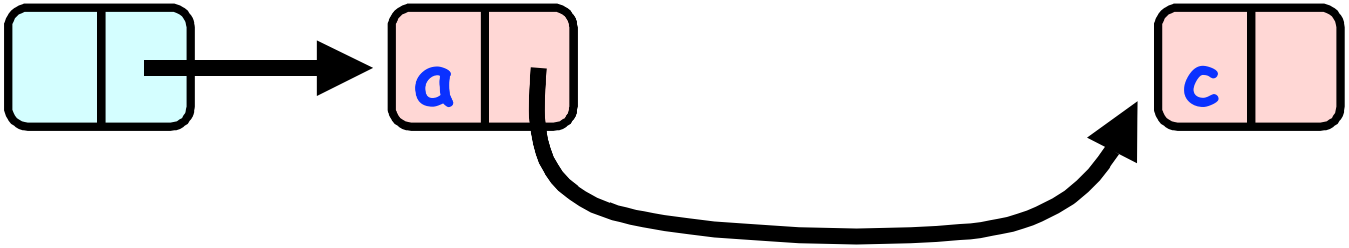
Removing an Entry



Removing an Entry



Removing an Entry



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

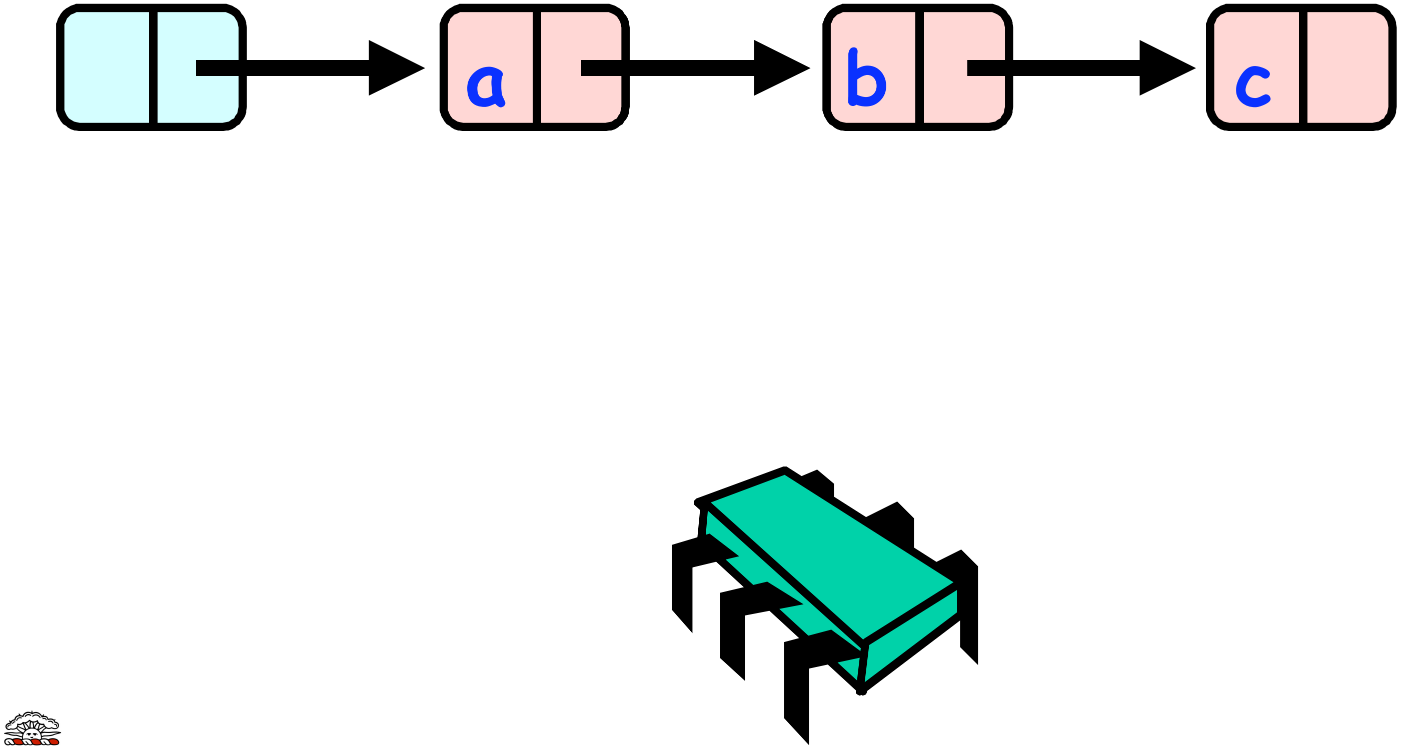
Optimistic Synchronization

- 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

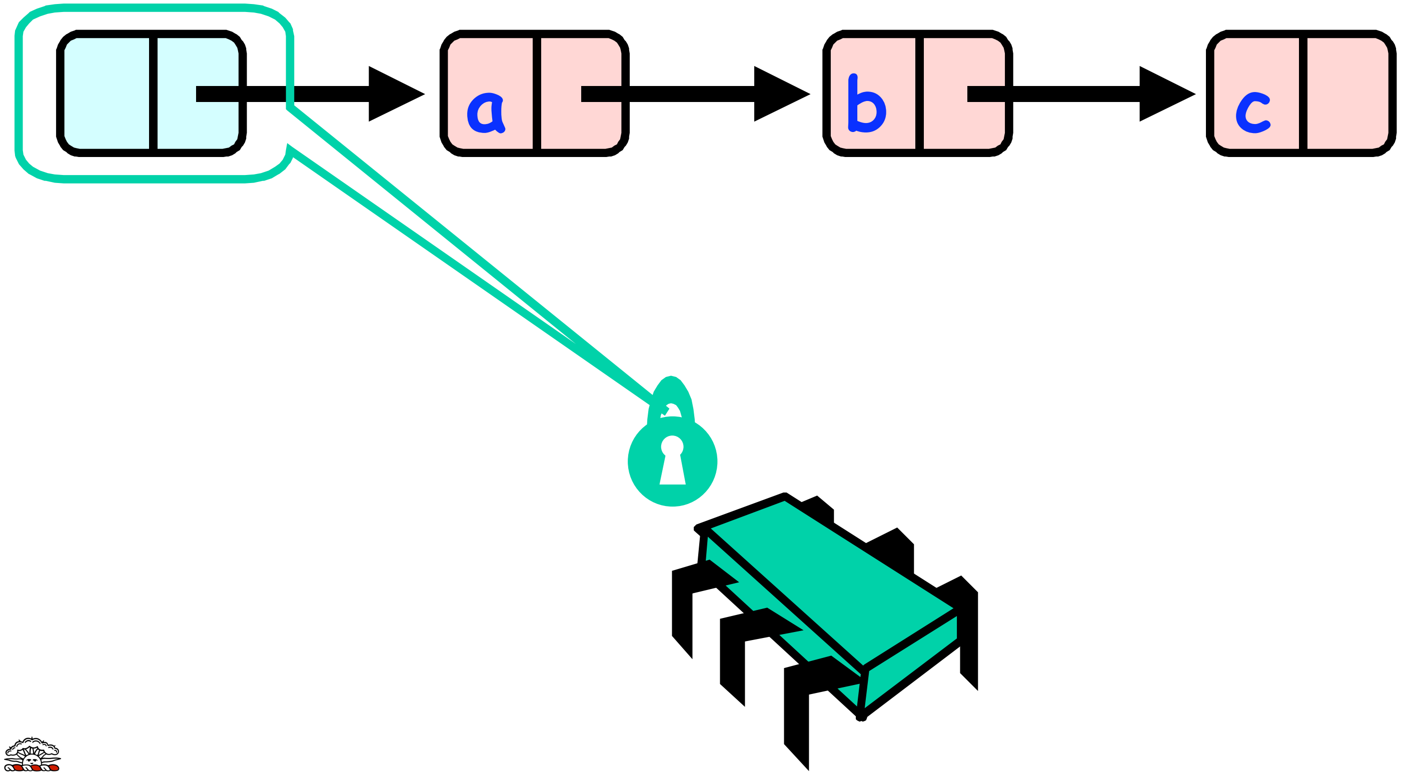
Lock-Free Synchronization

- 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()

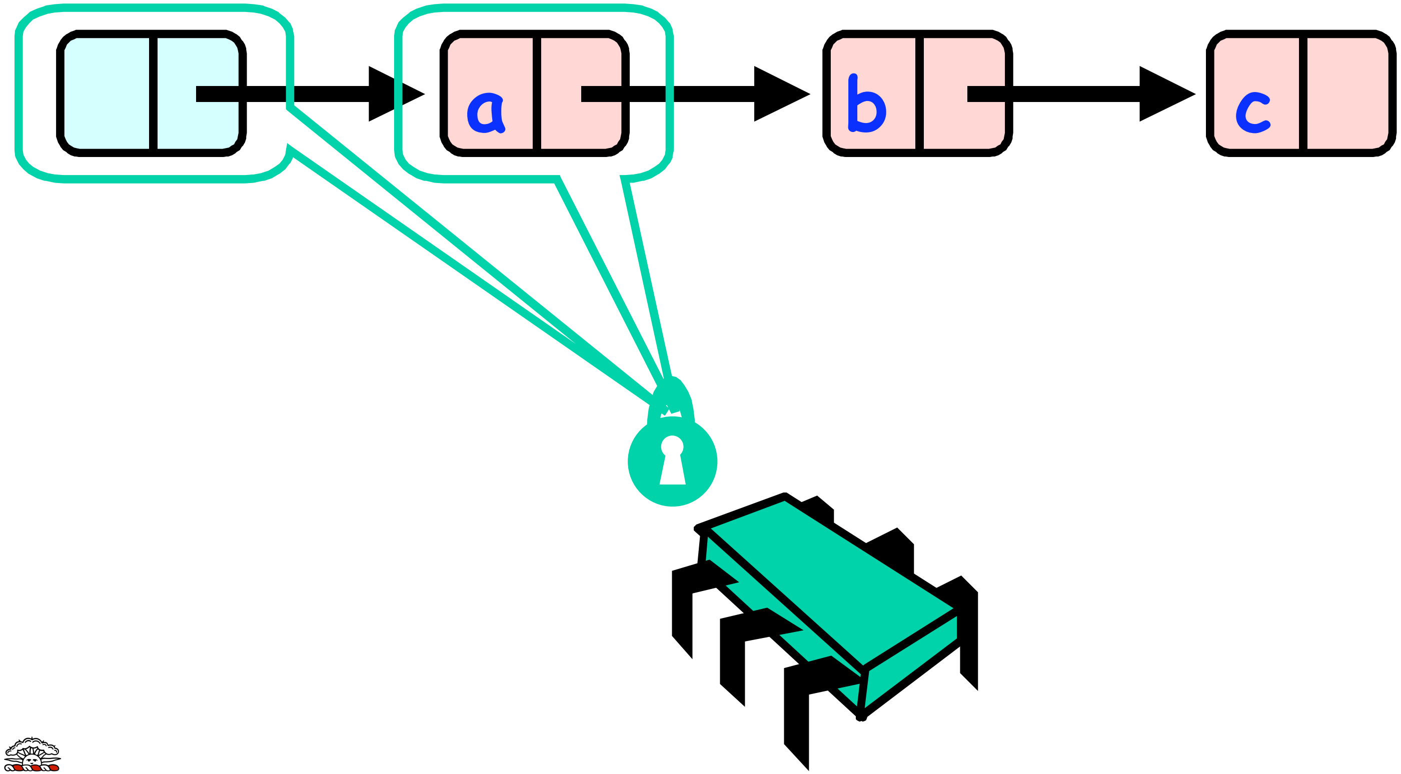
Hand-over-Hand locking



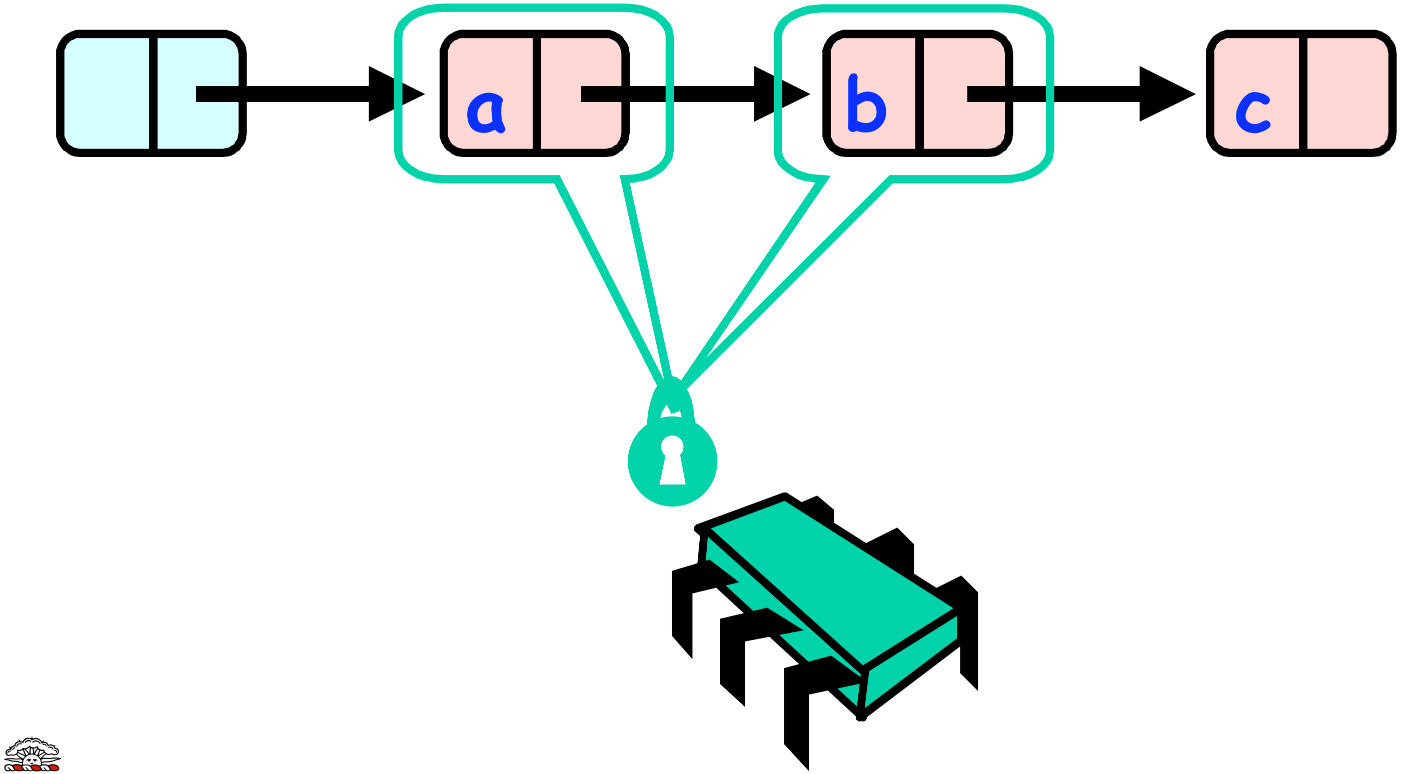
Hand-over-Hand locking



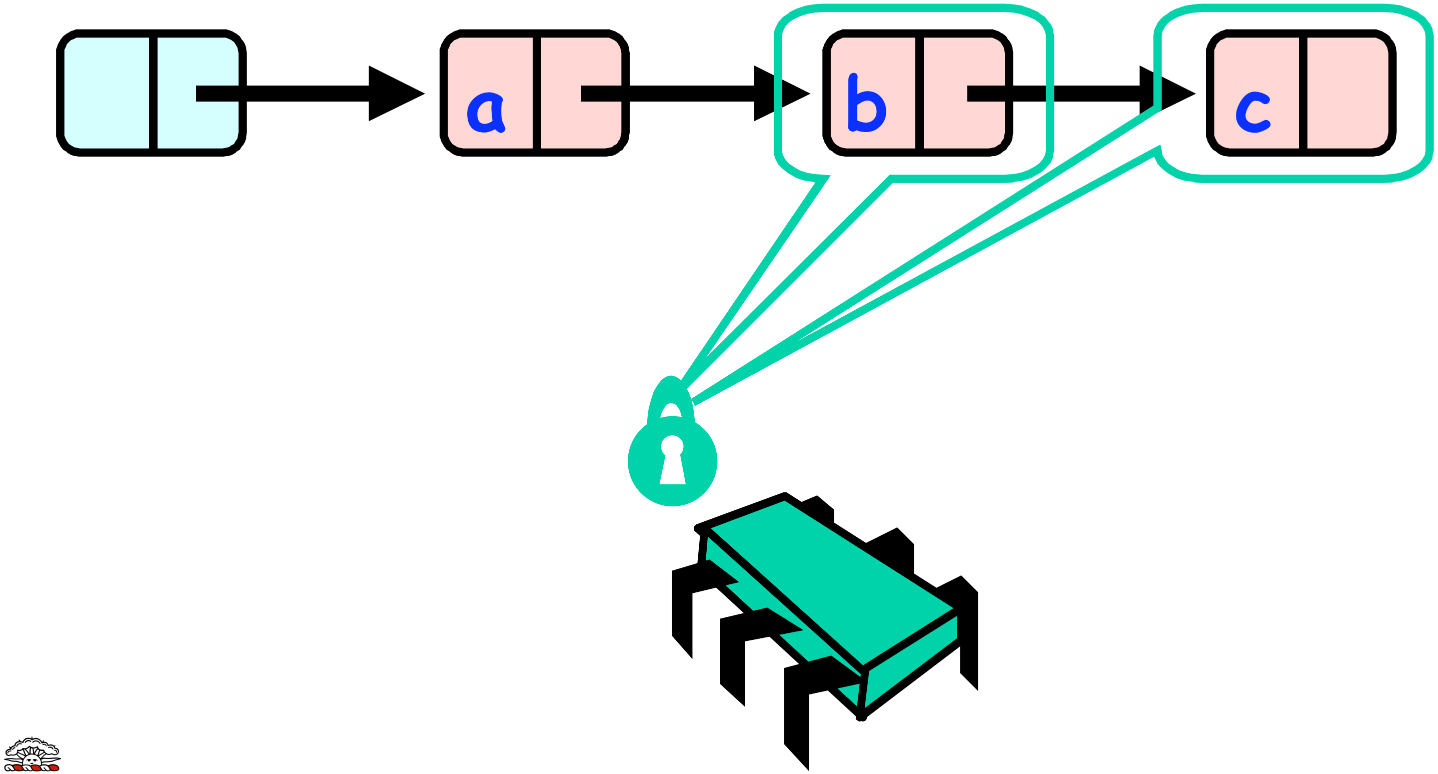
Hand-over-Hand locking



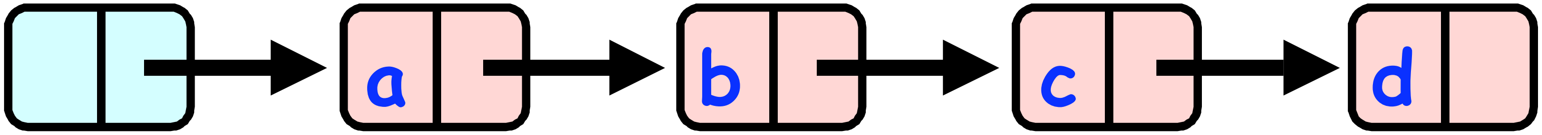
Hand-over-Hand locking



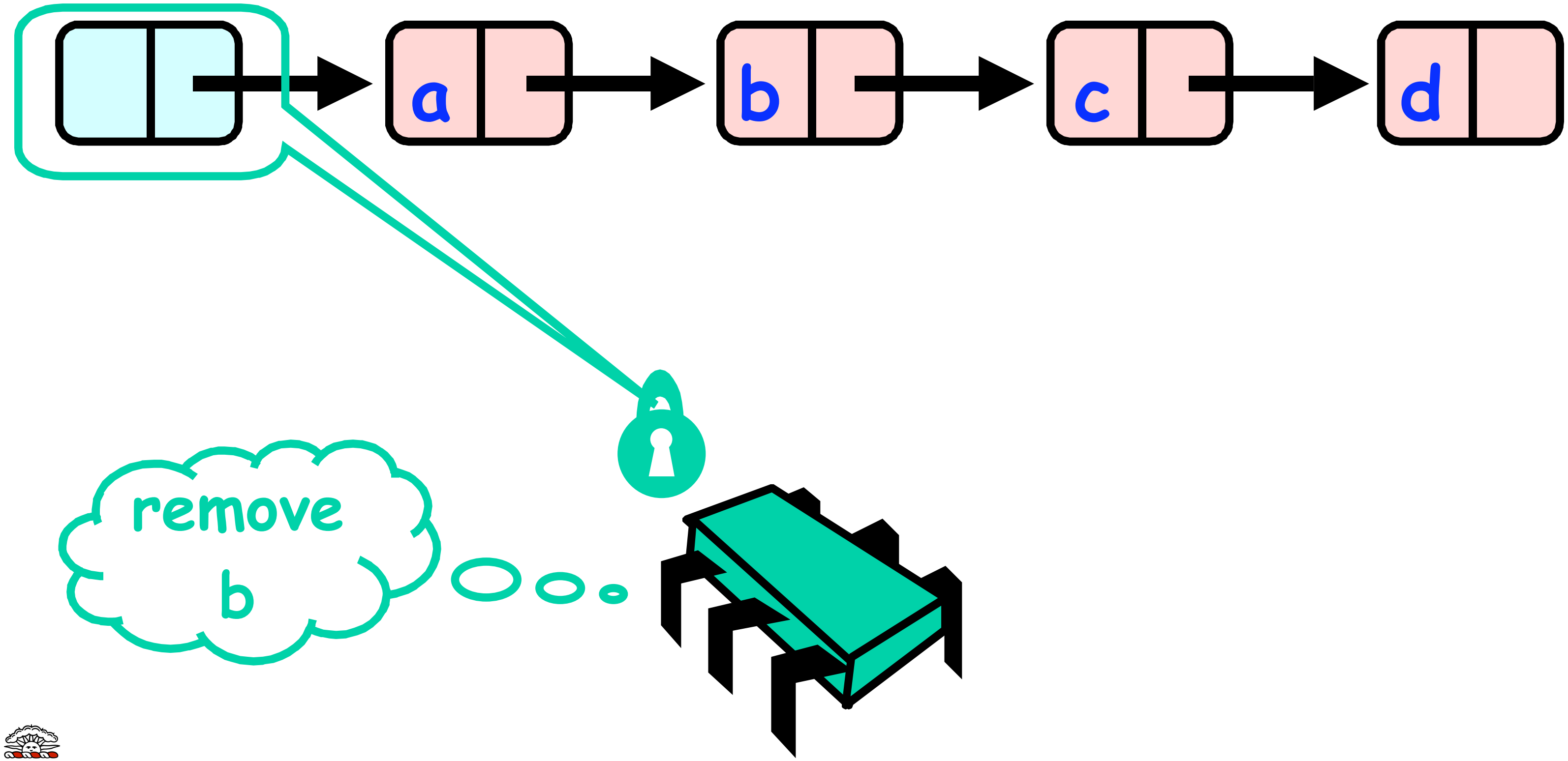
Hand-over-Hand locking



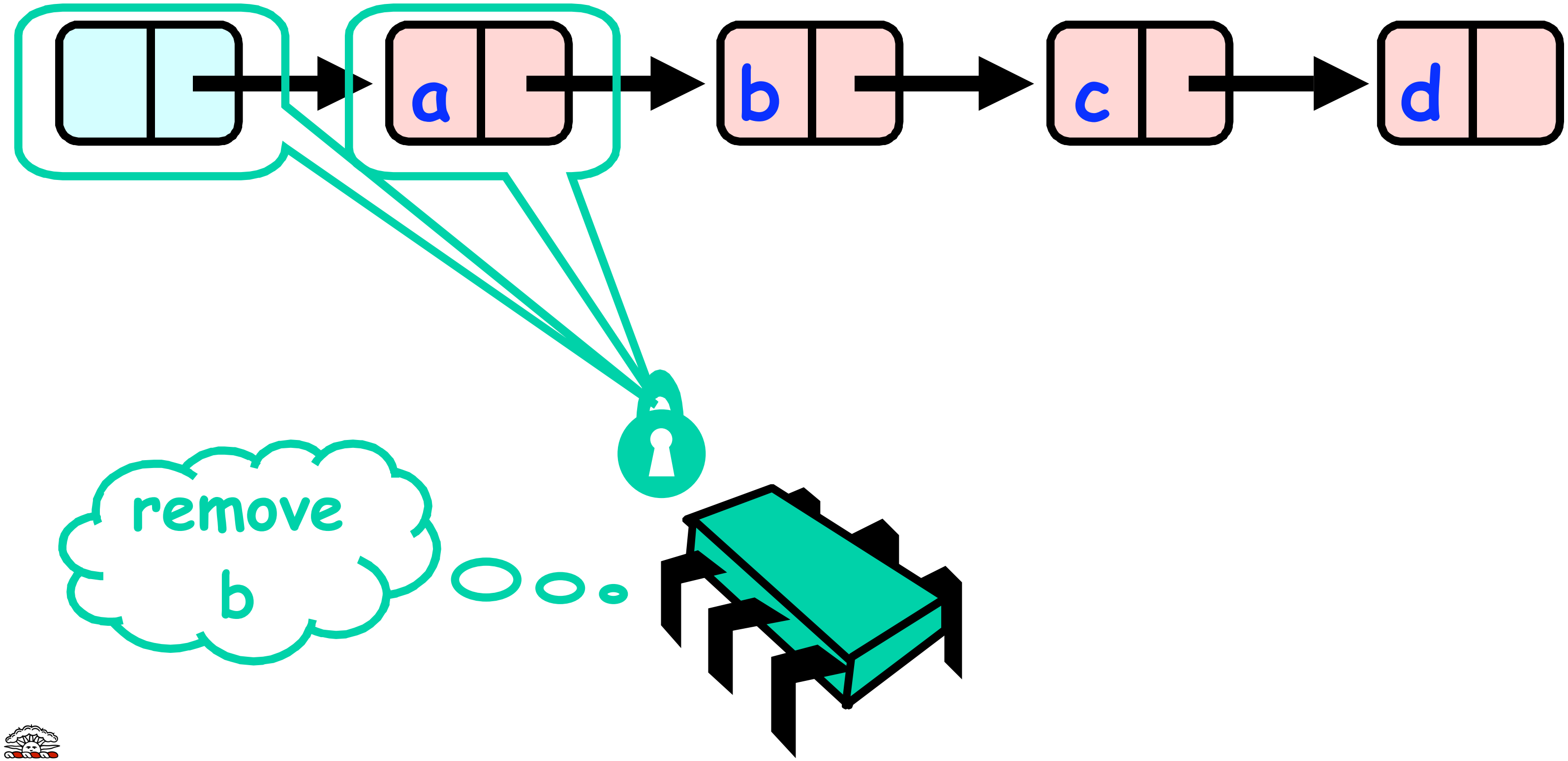
Removing an Entry



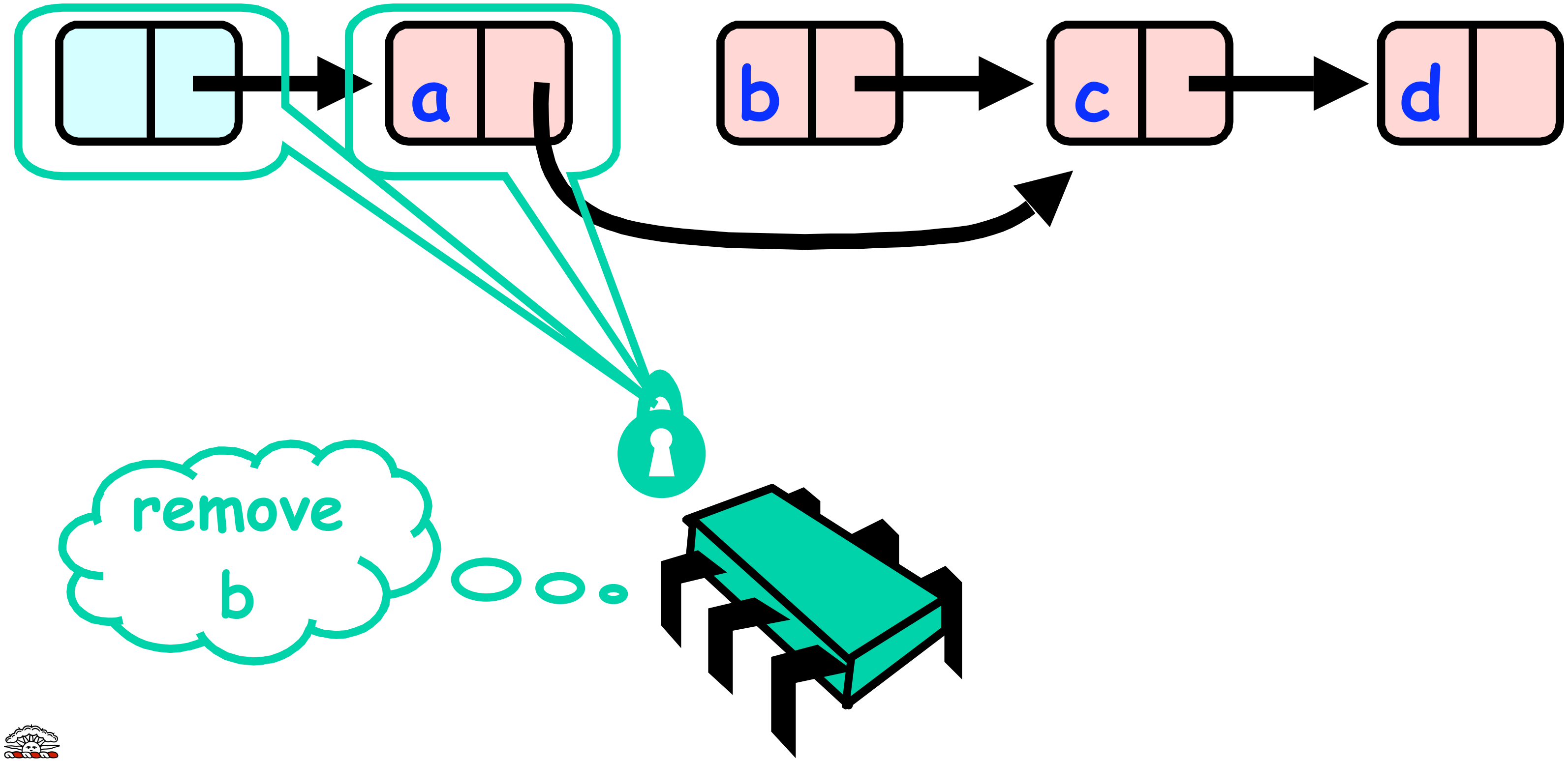
Removing an Entry



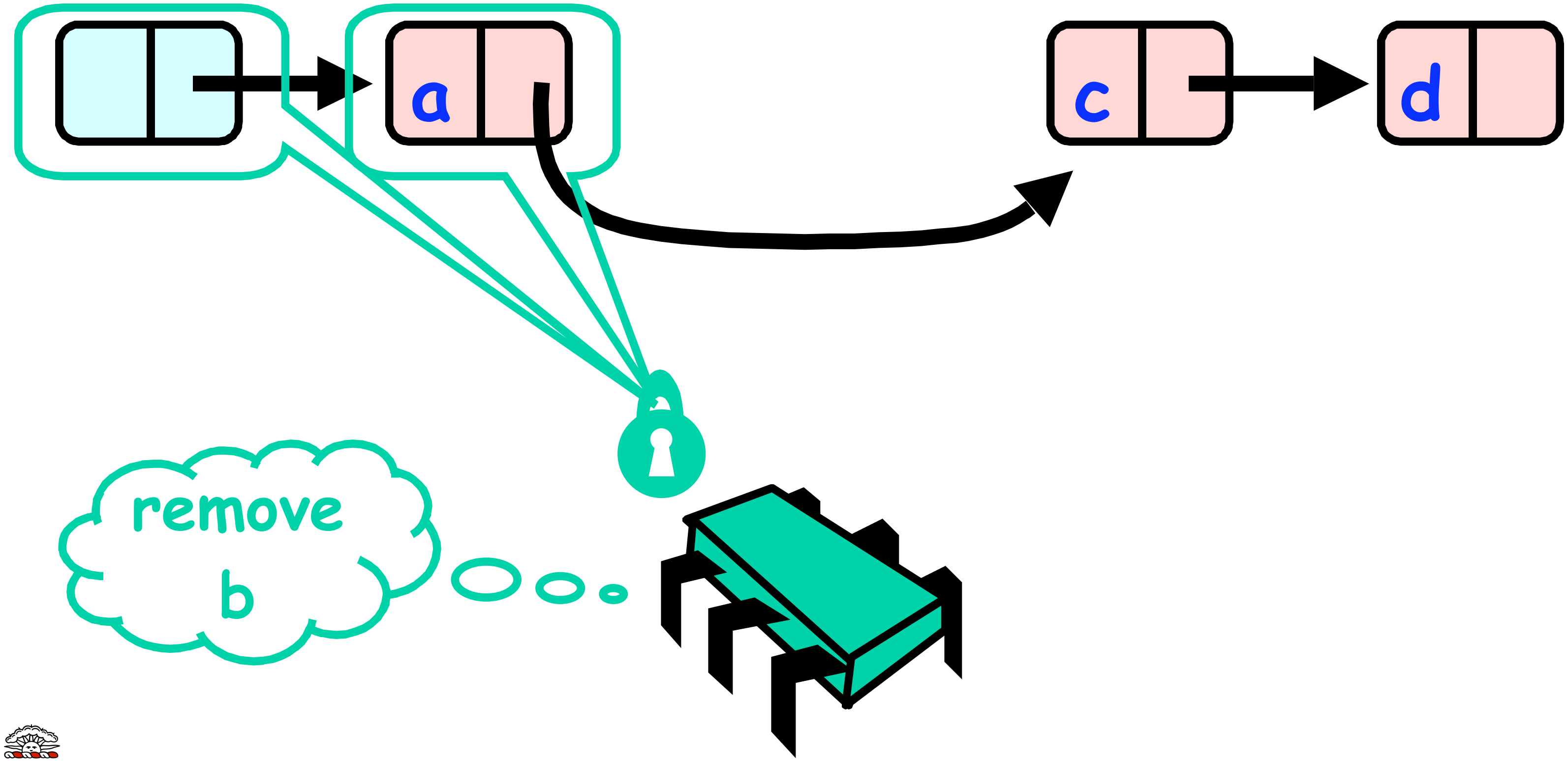
Removing an Entry



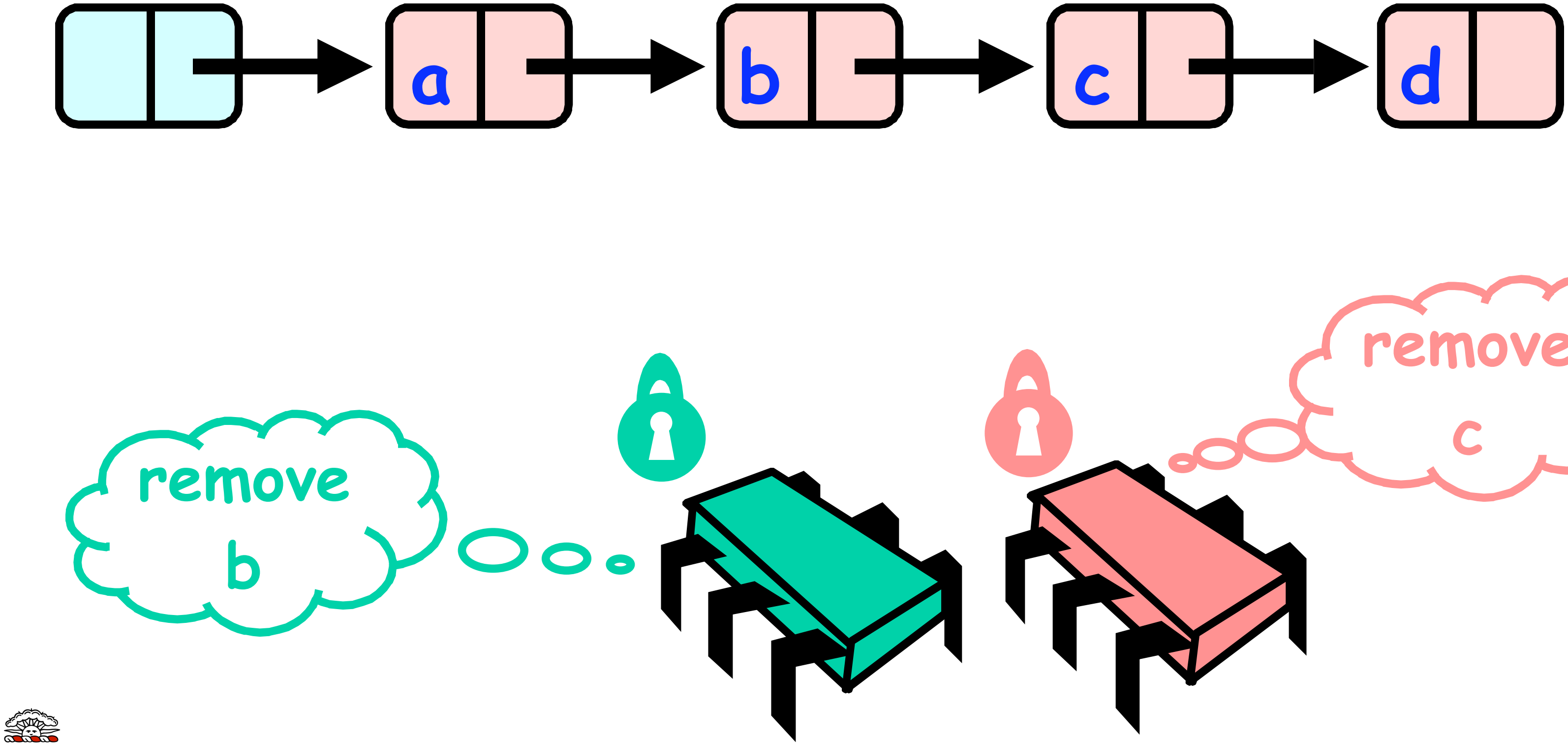
Removing an Entry



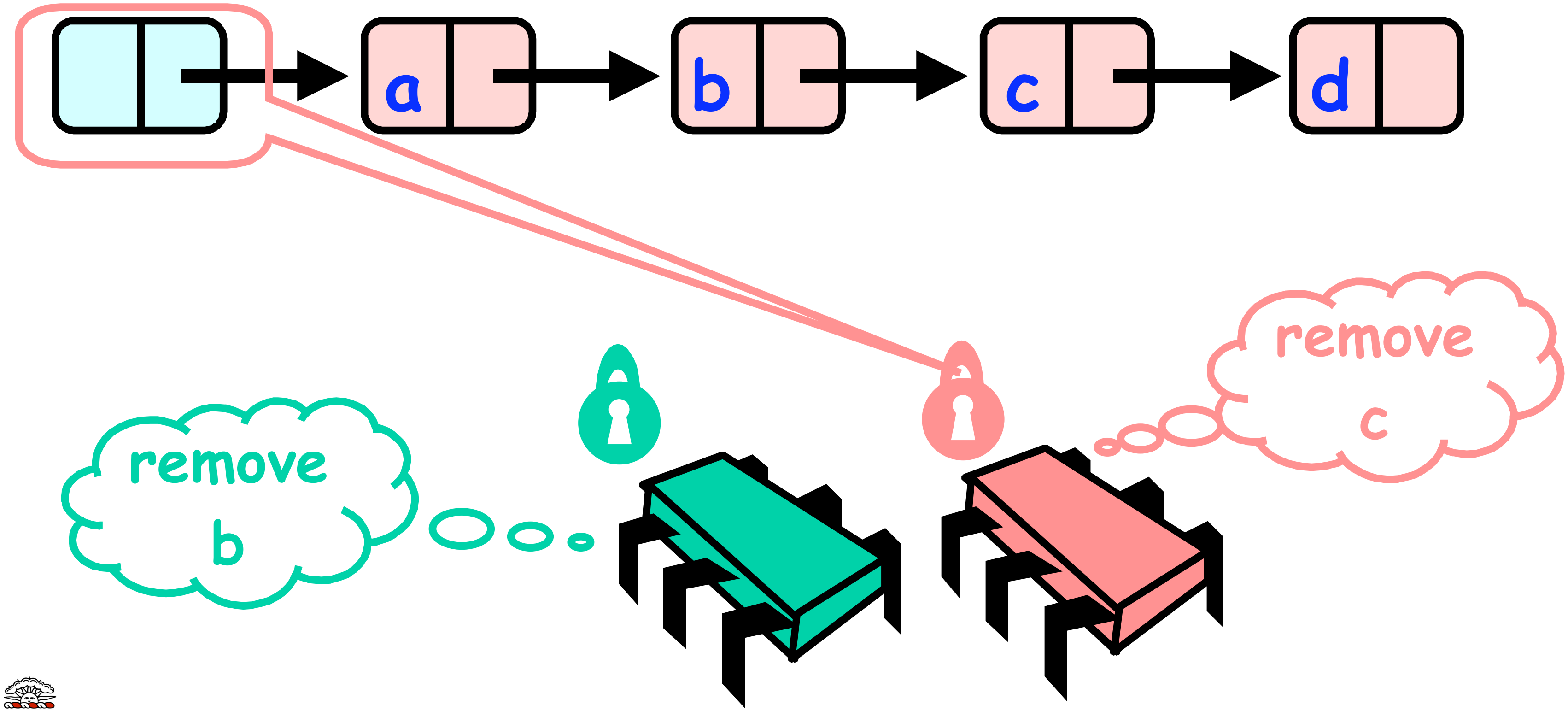
Removing an Entry



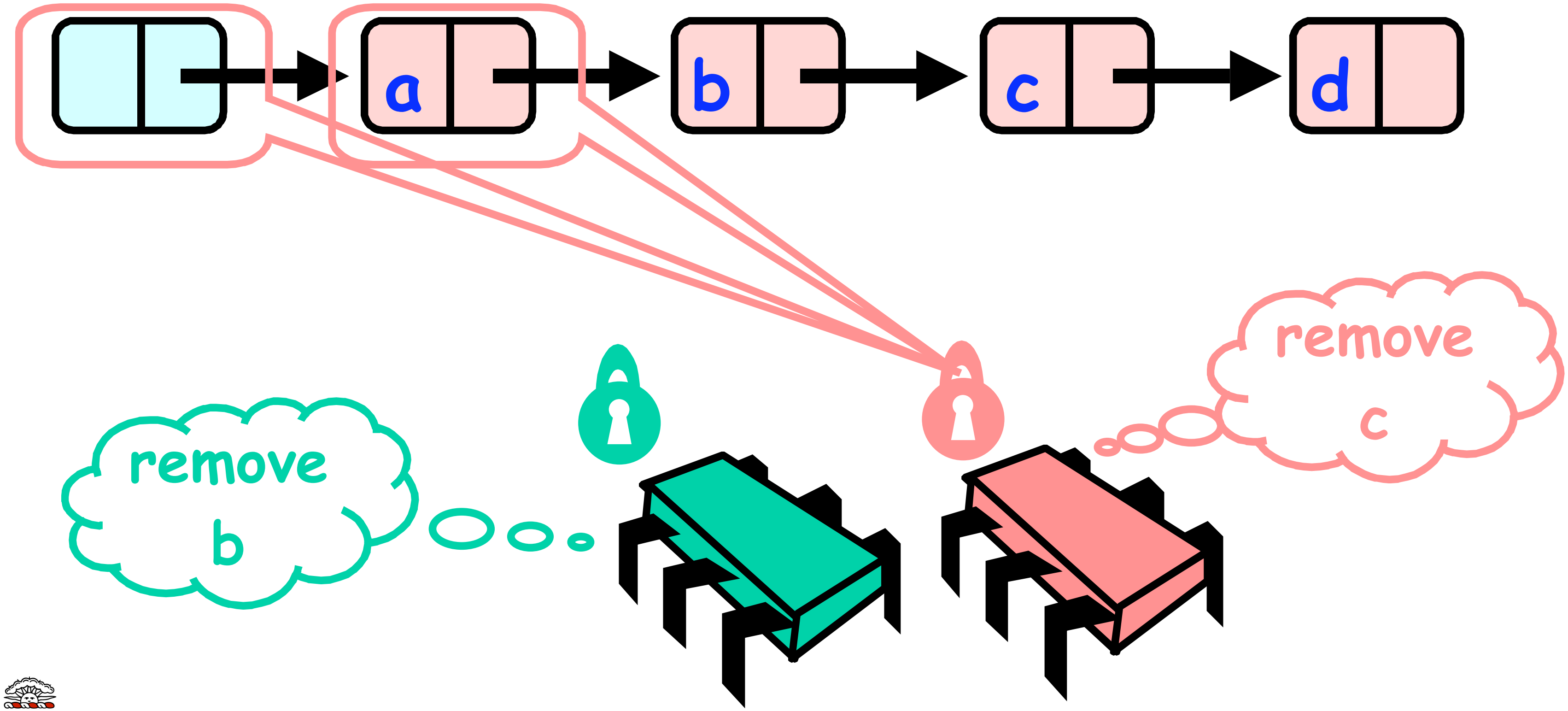
Removing an Entry



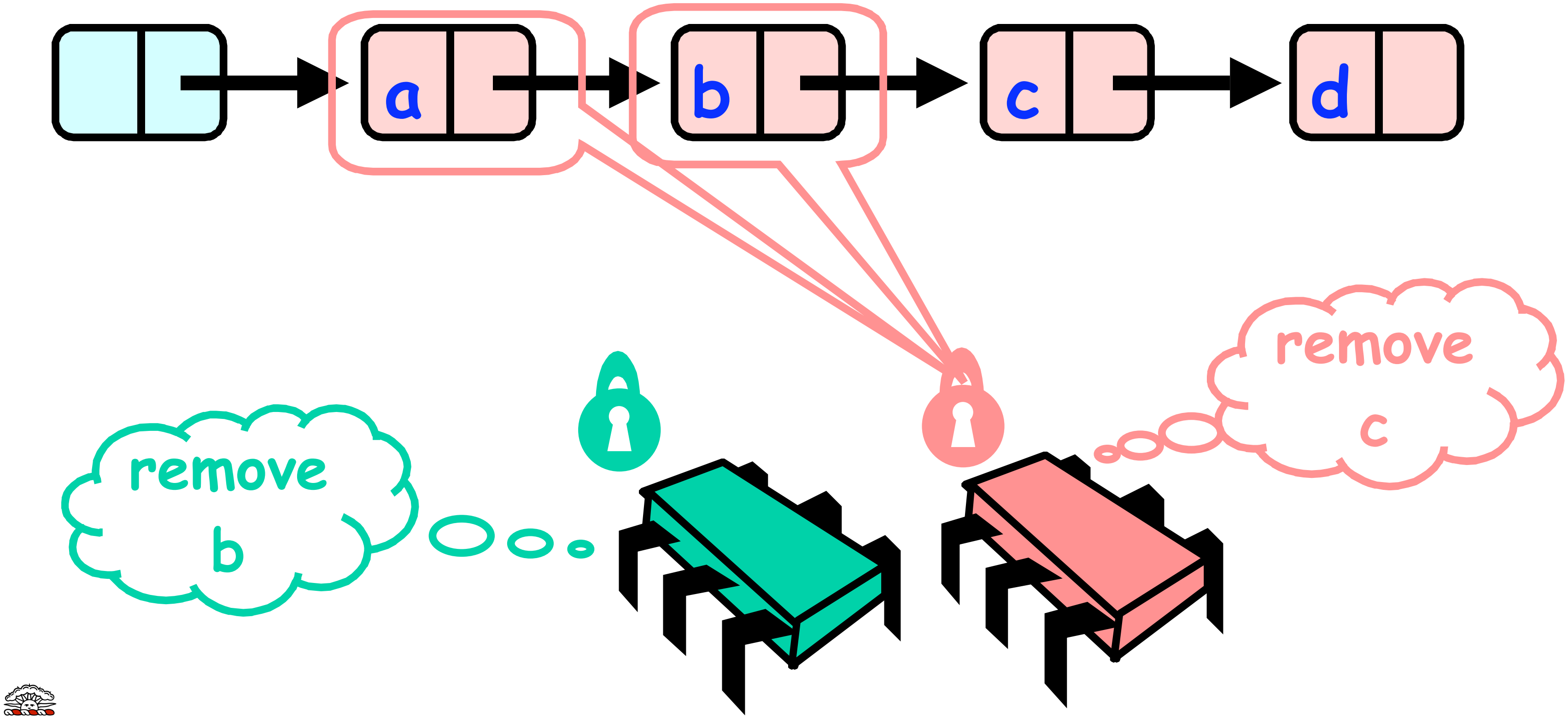
Removing an Entry



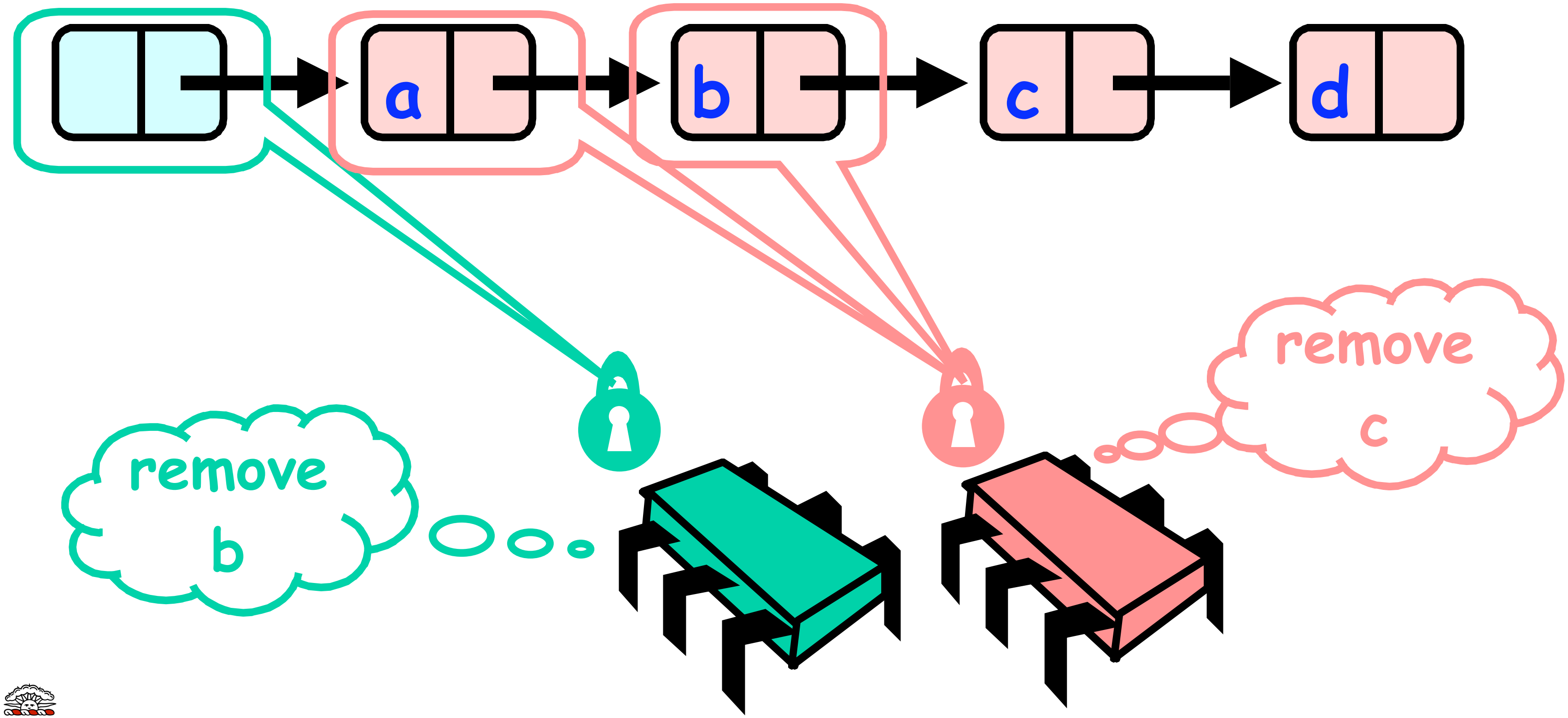
Removing an Entry



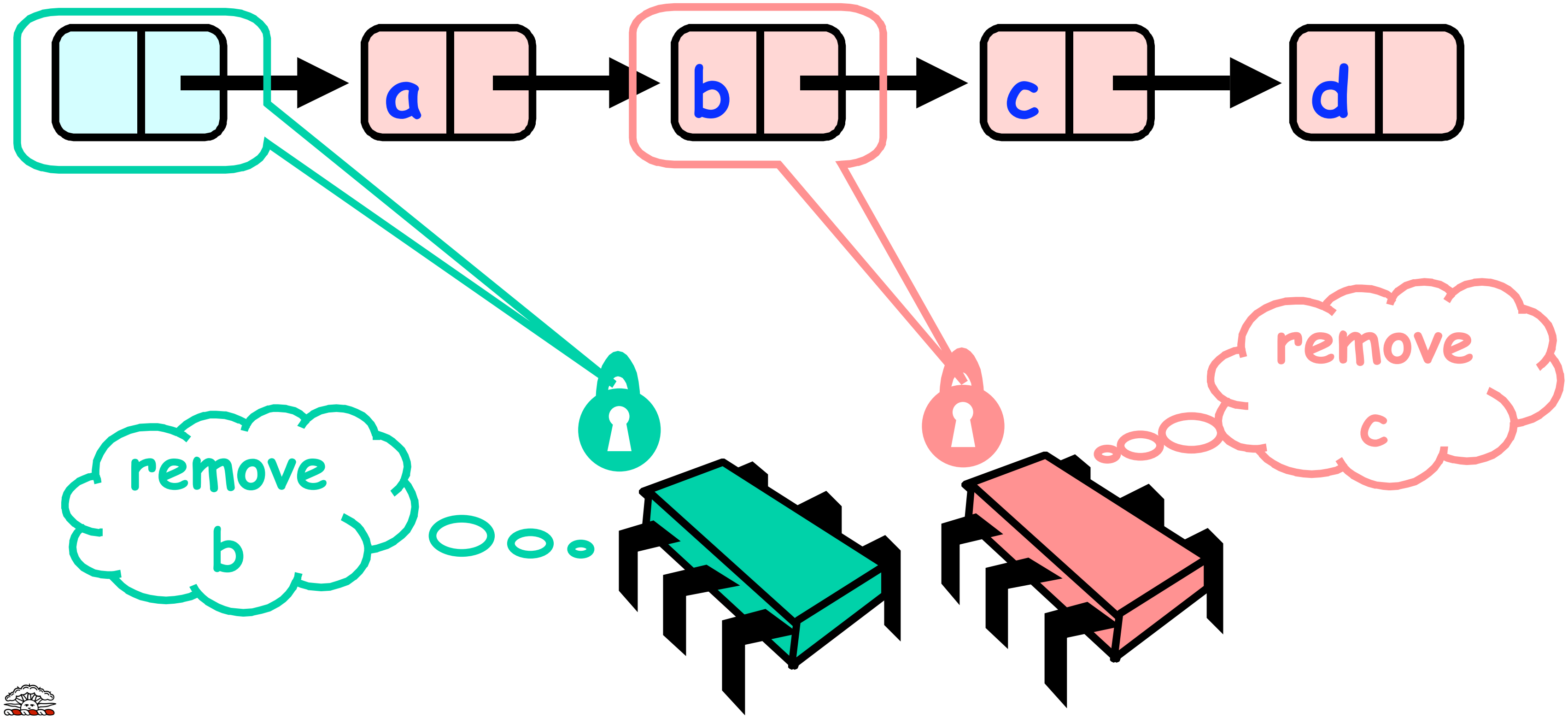
Removing an Entry



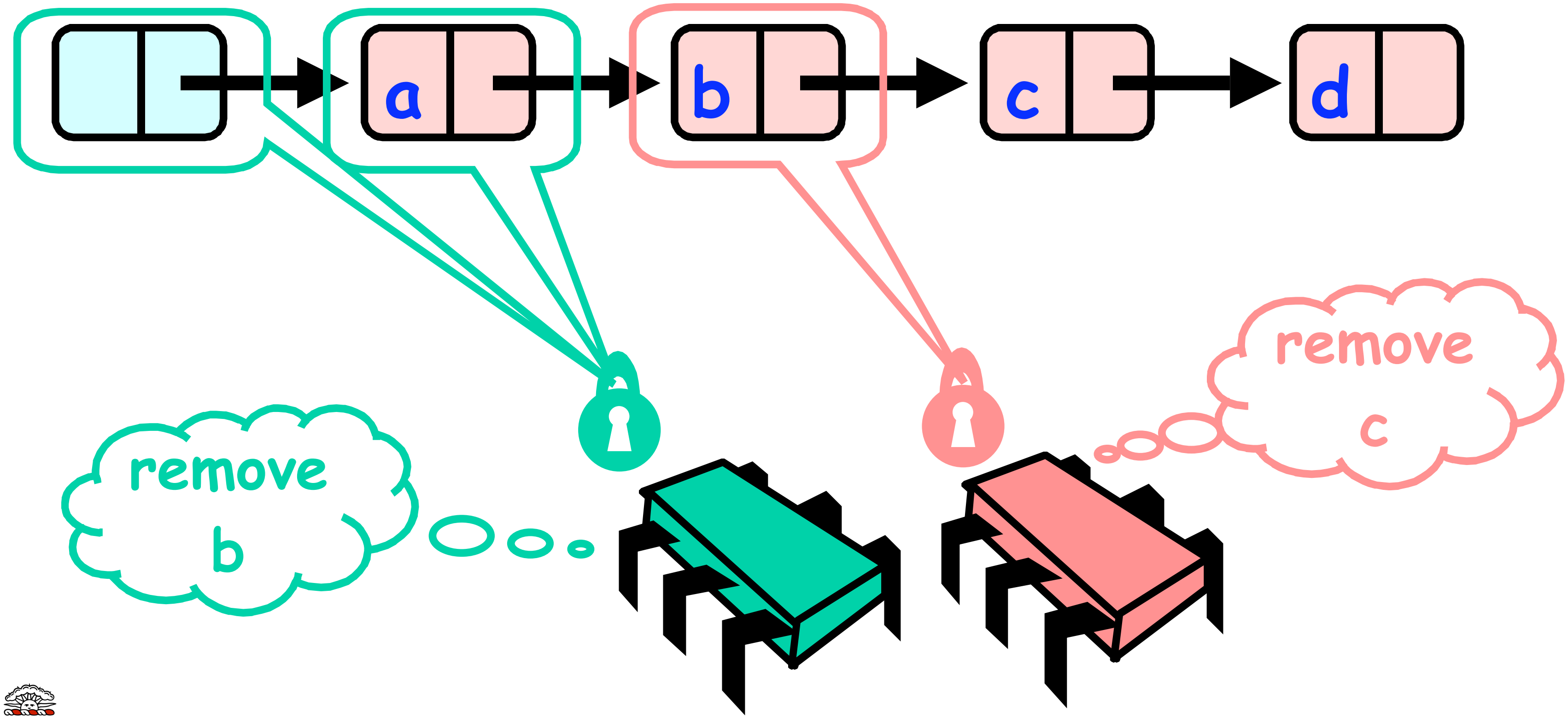
Removing an Entry



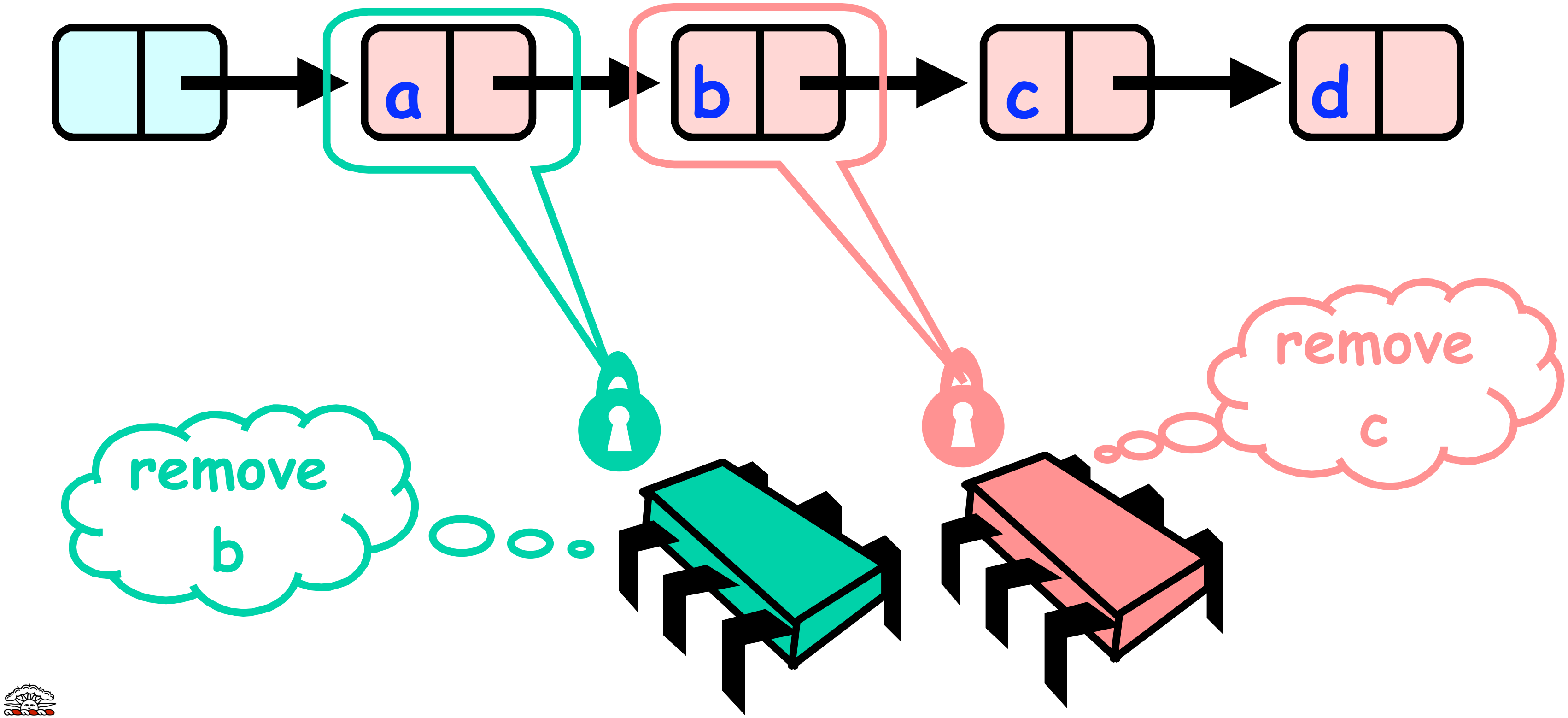
Removing an Entry



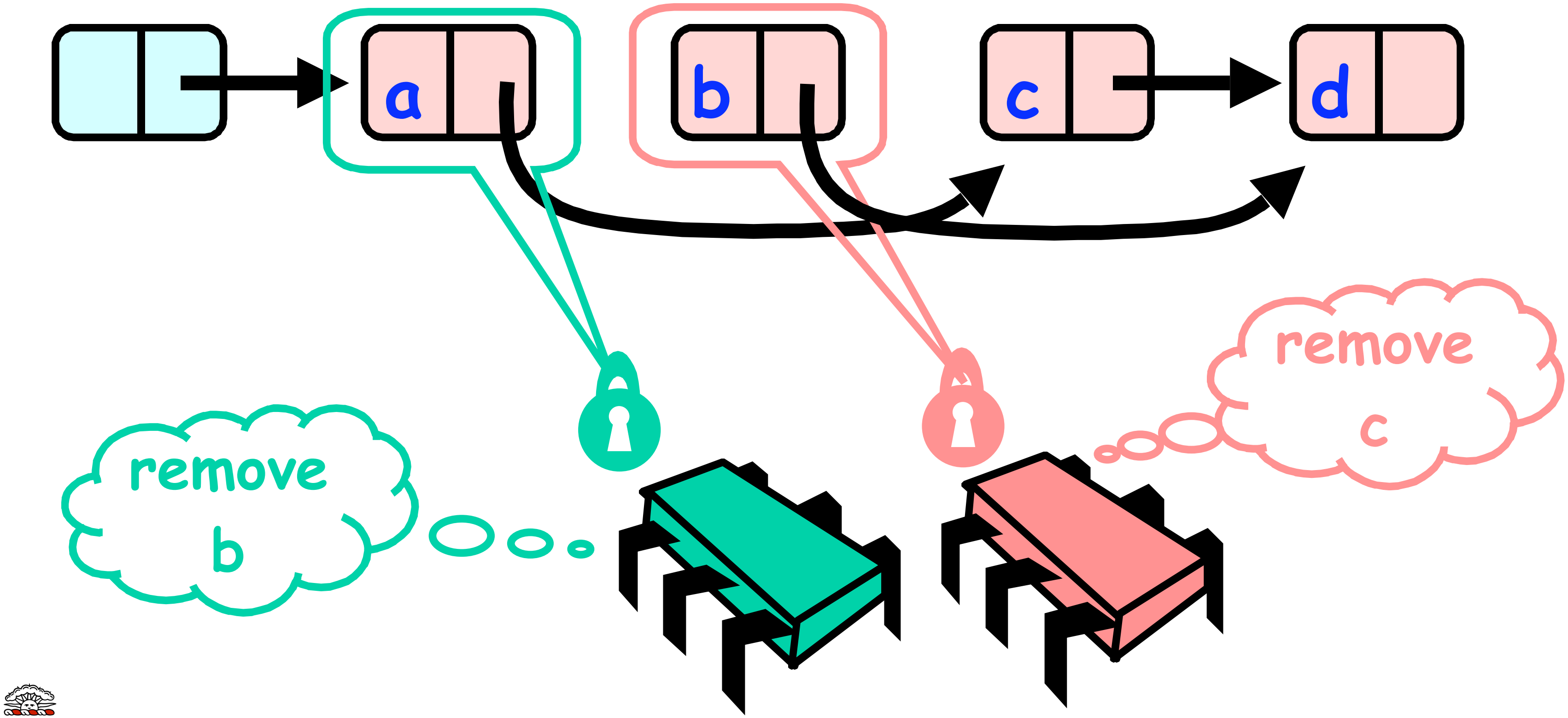
Removing an Entry



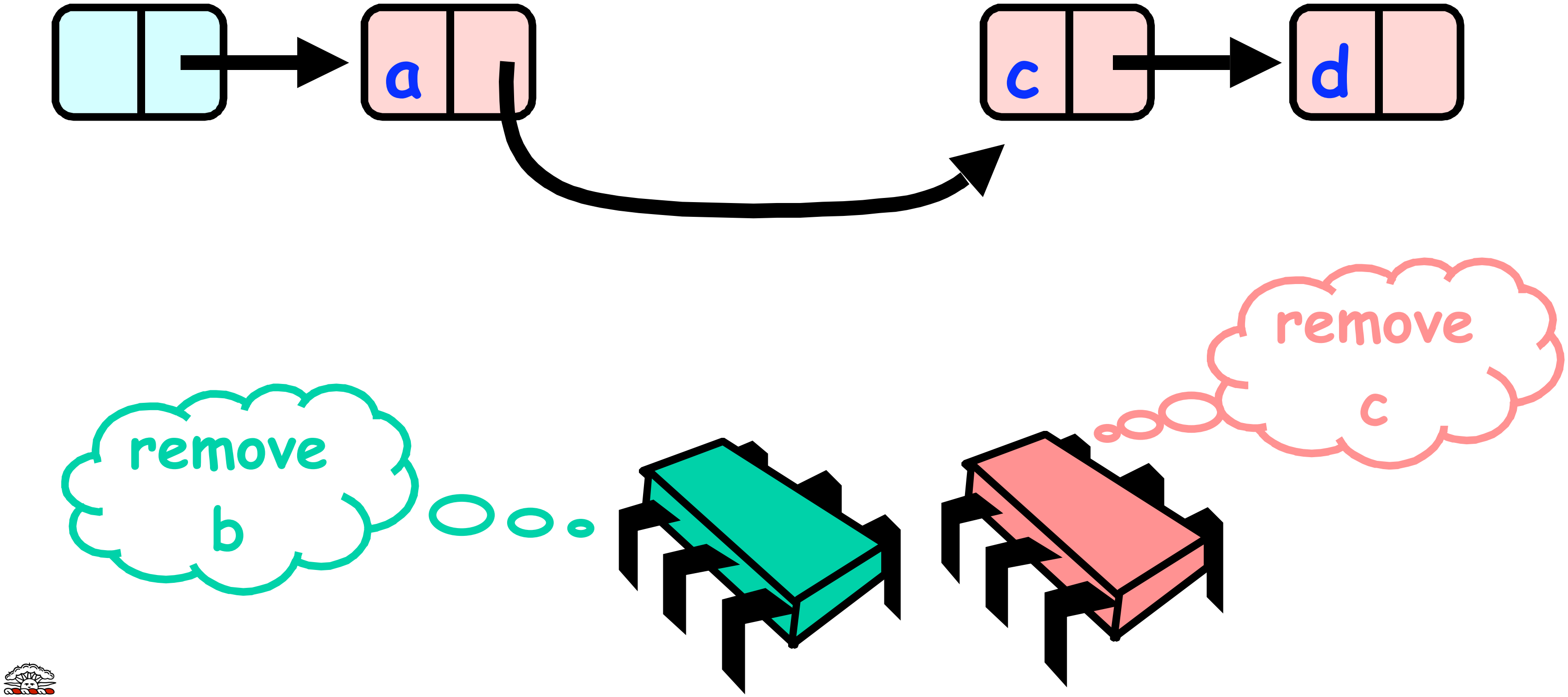
Removing an Entry



Removing an Entry

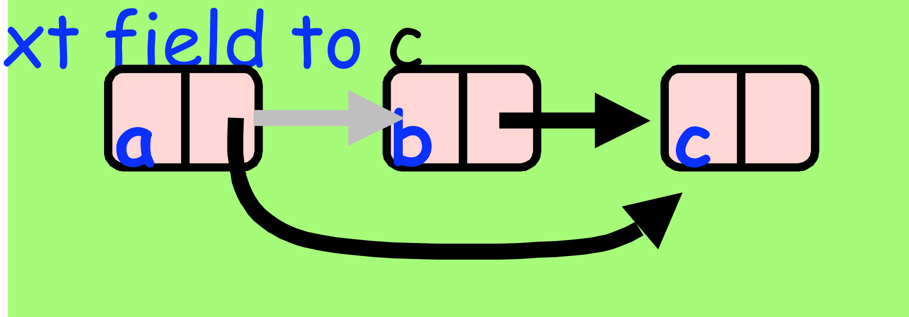


Uh, Oh

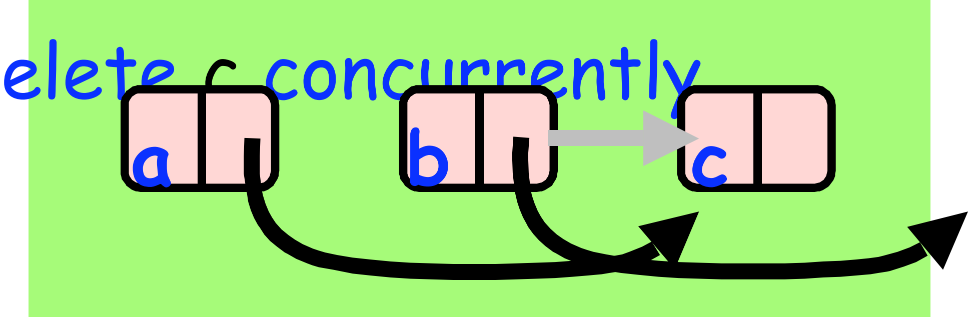


Problem

- To delete entry b
  - Swing entry a's next field to c



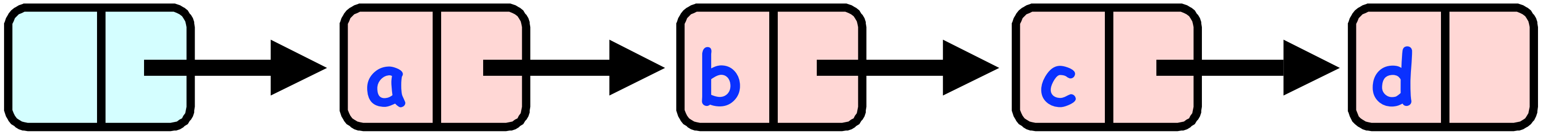
- Problem is,
  - Someone could delete c concurrently



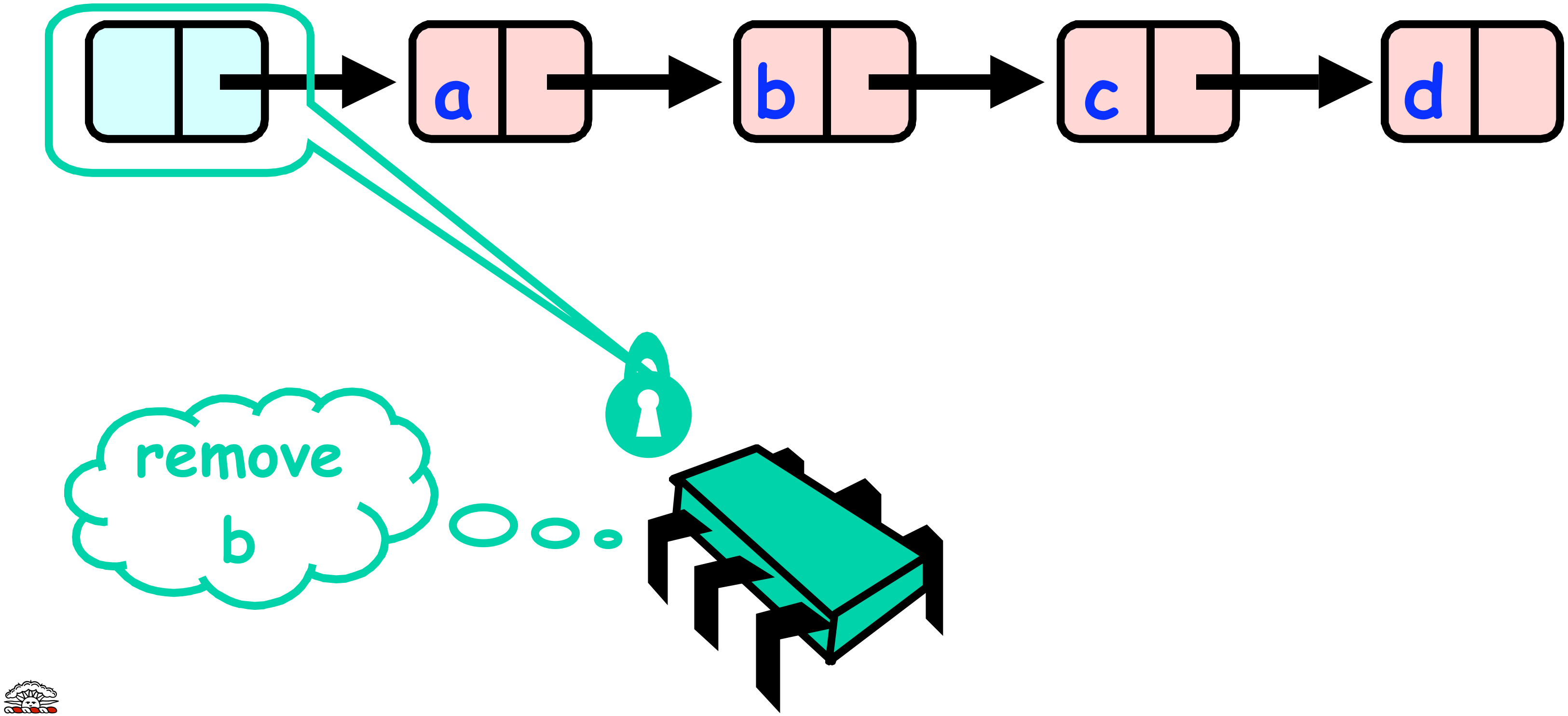
Insight

- 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

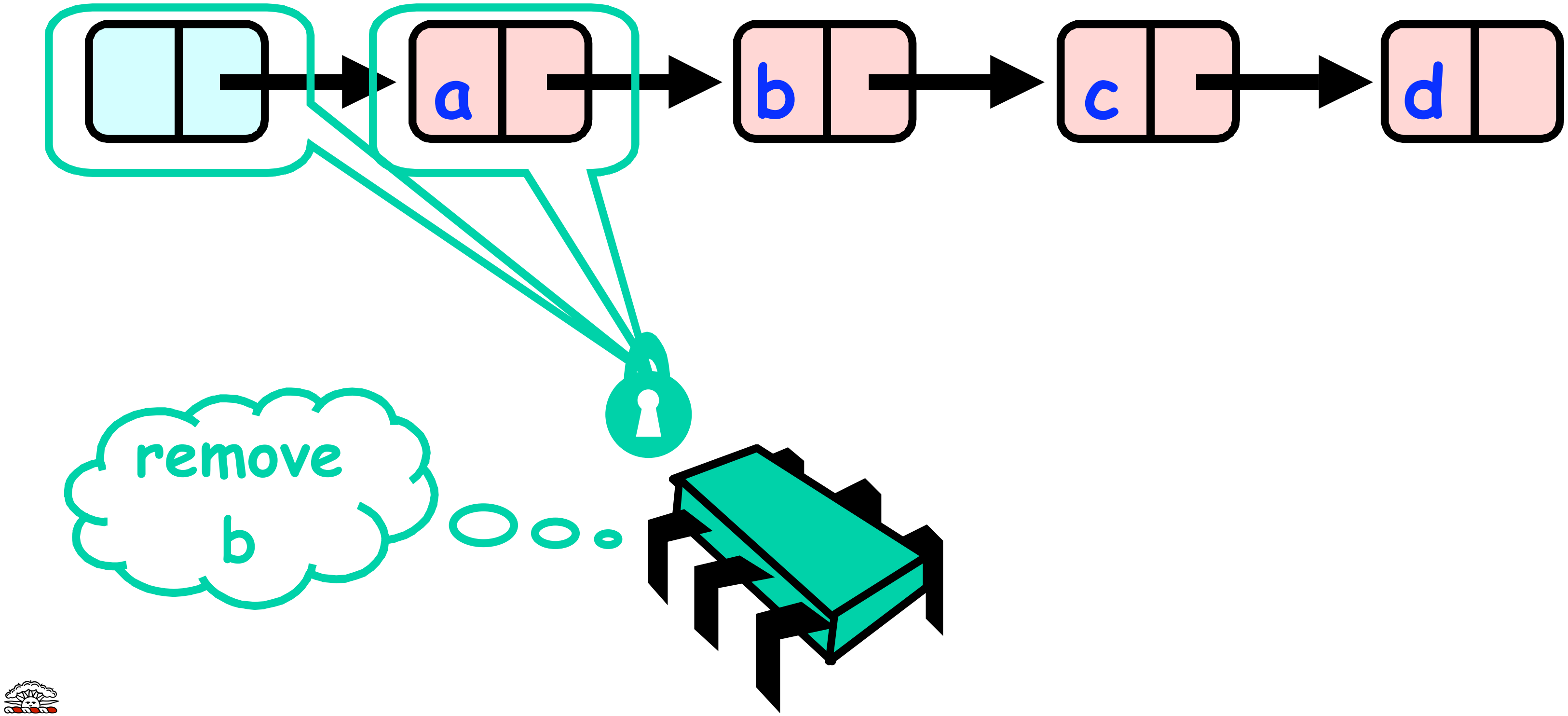
Hand-Over-Hand Again



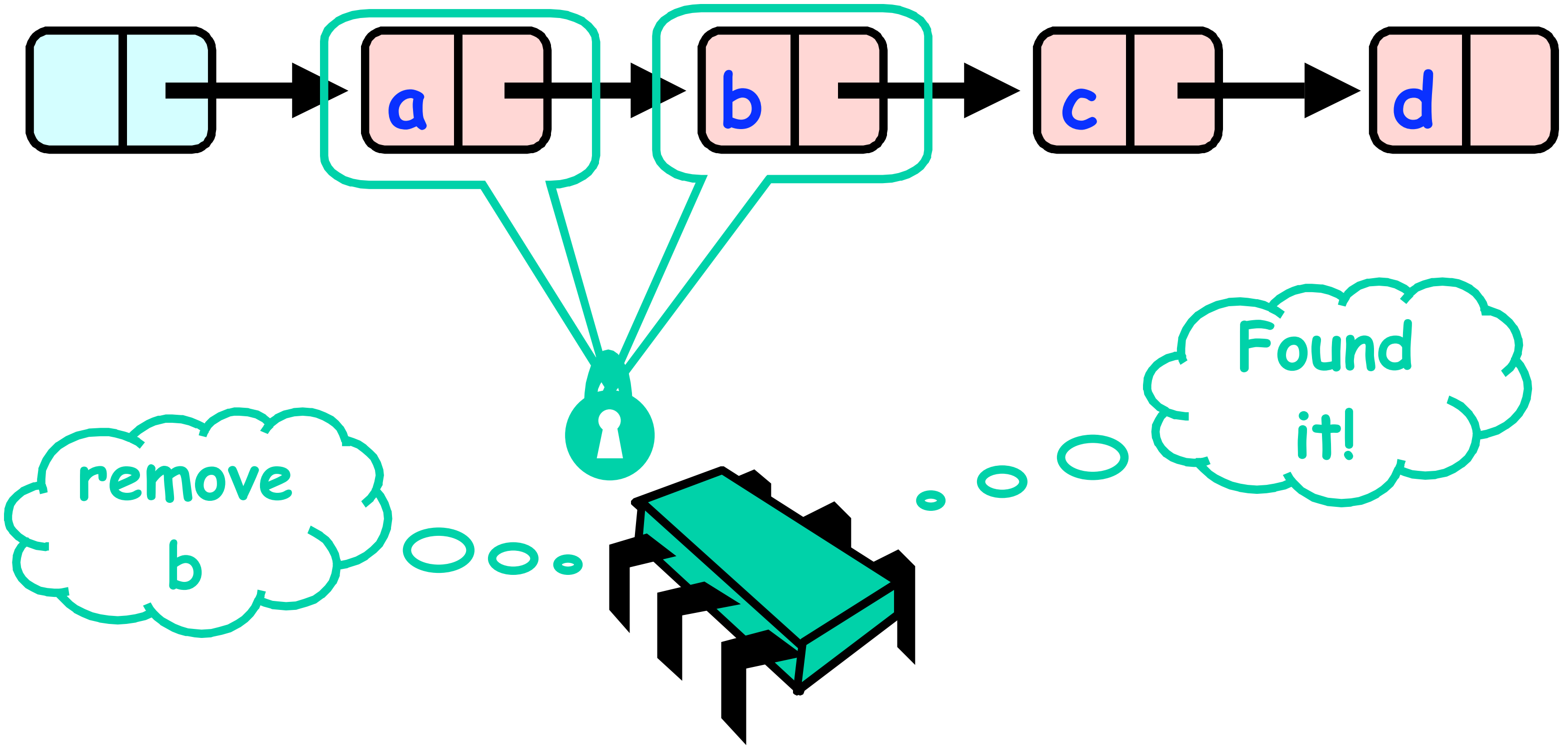
Hand-Over-Hand Again



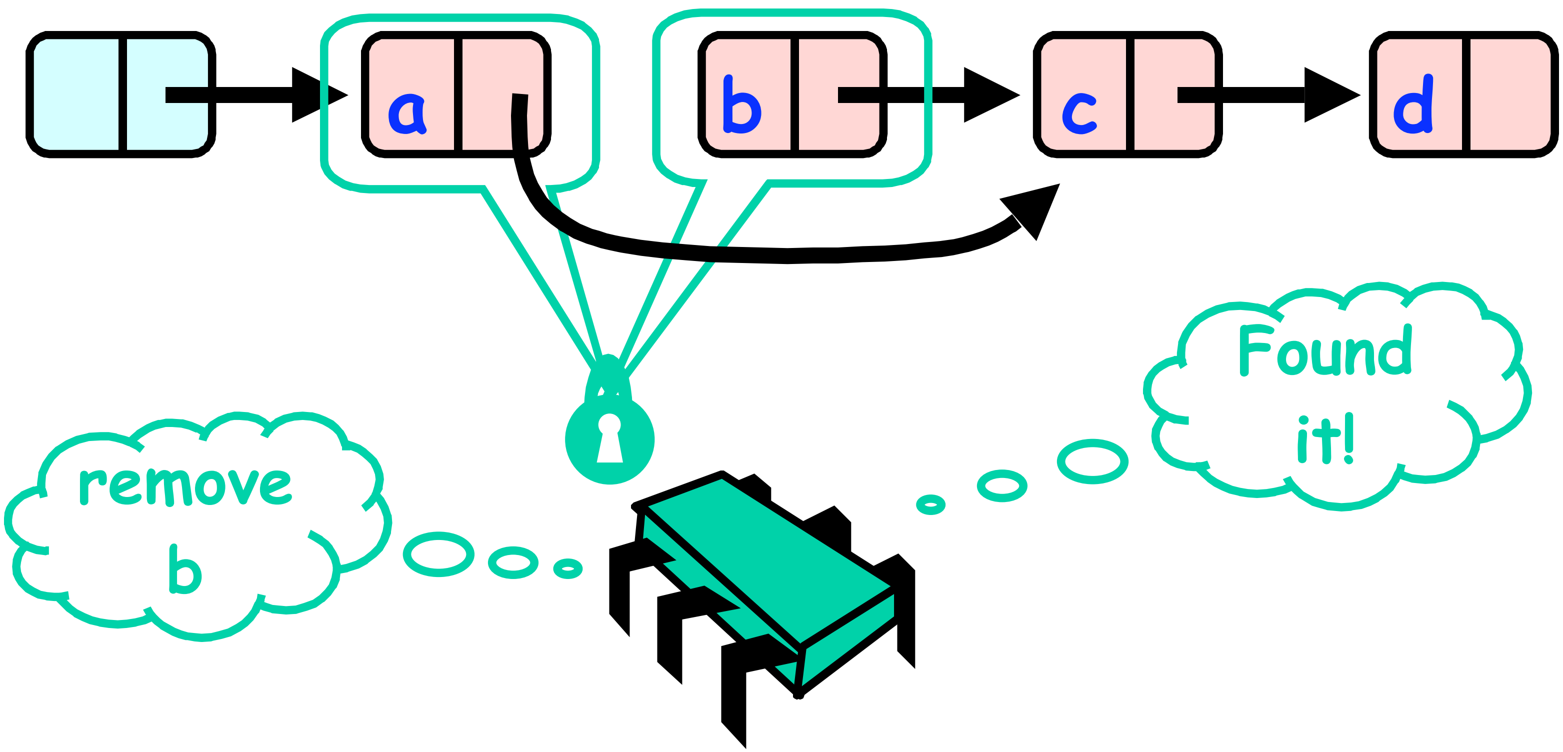
Hand-Over-Hand Again



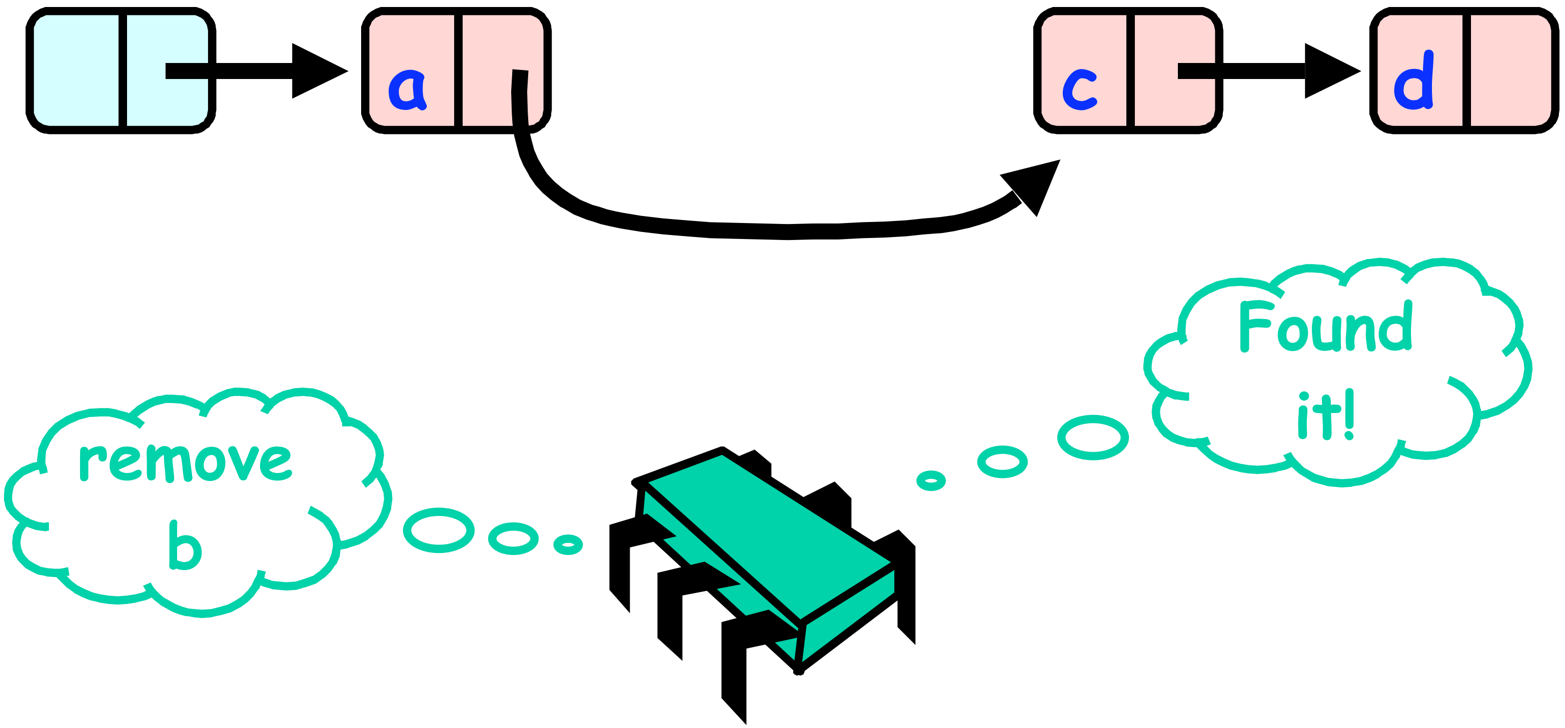
Hand-Over-Hand Again



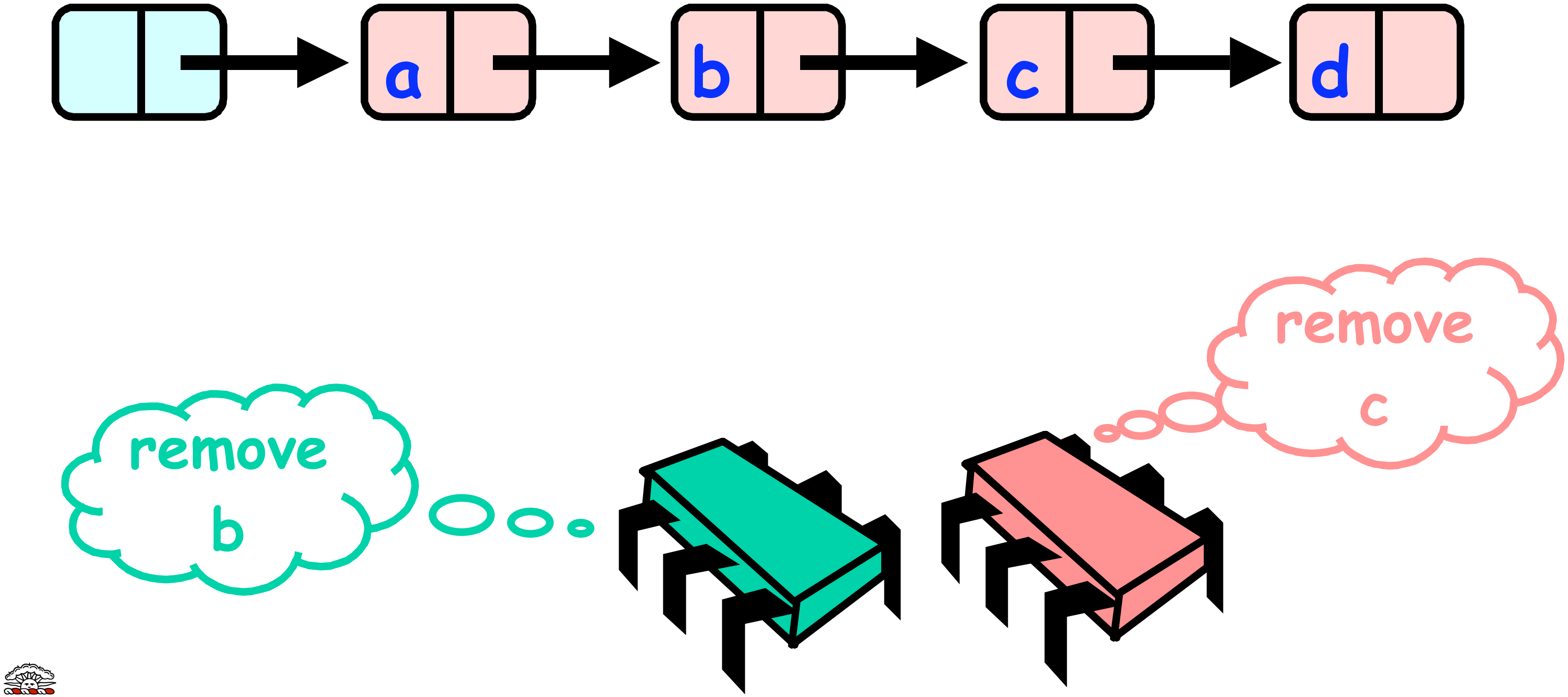
Hand-Over-Hand Again



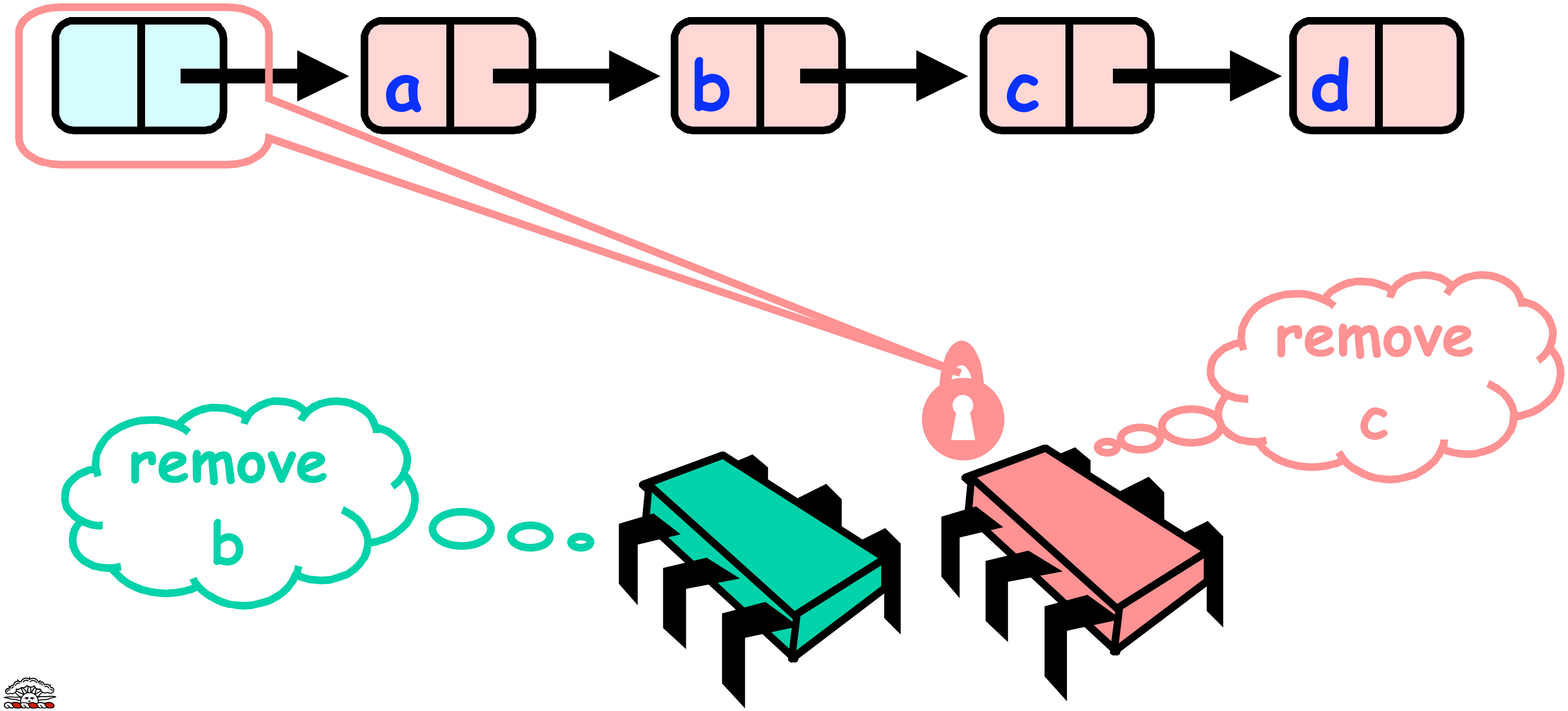
Hand-Over-Hand Again



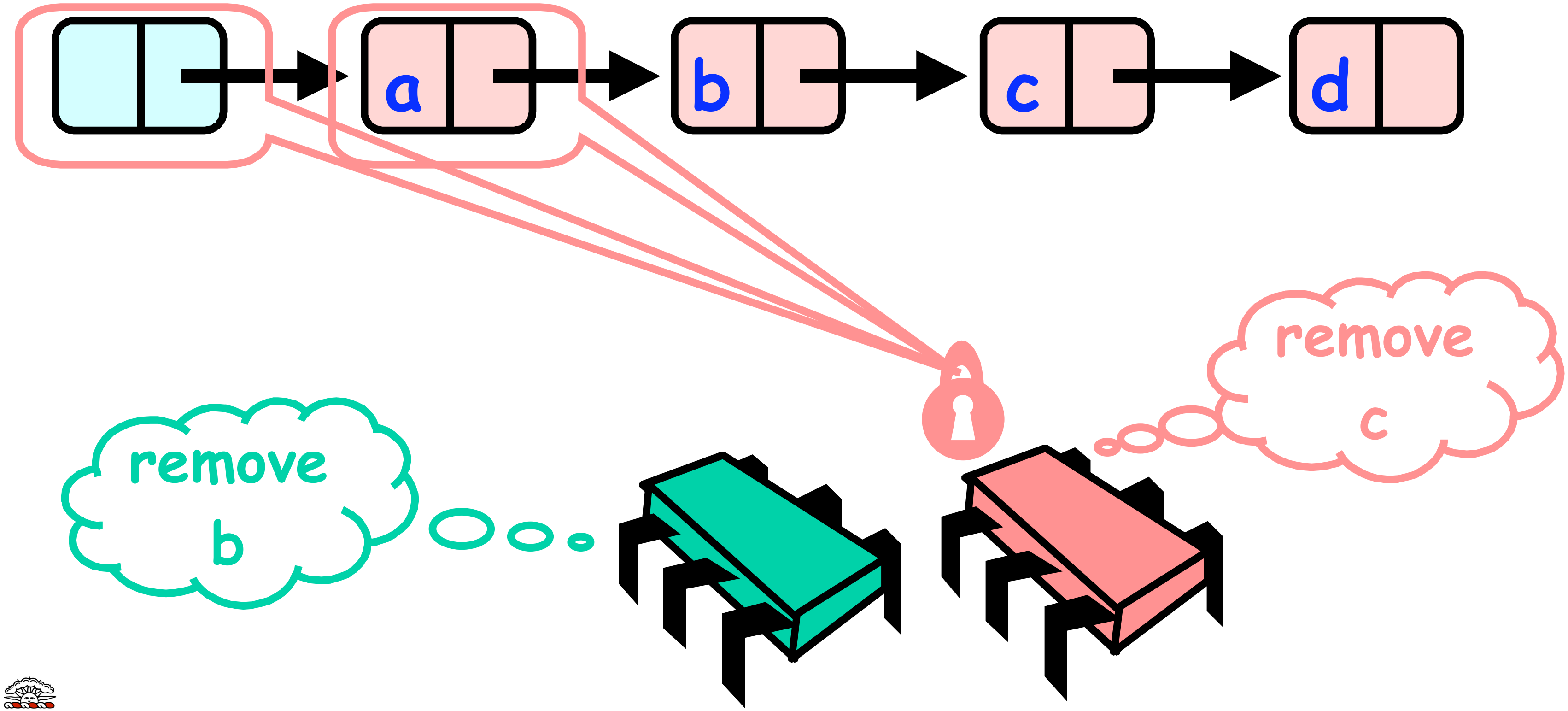
Removing an Entry



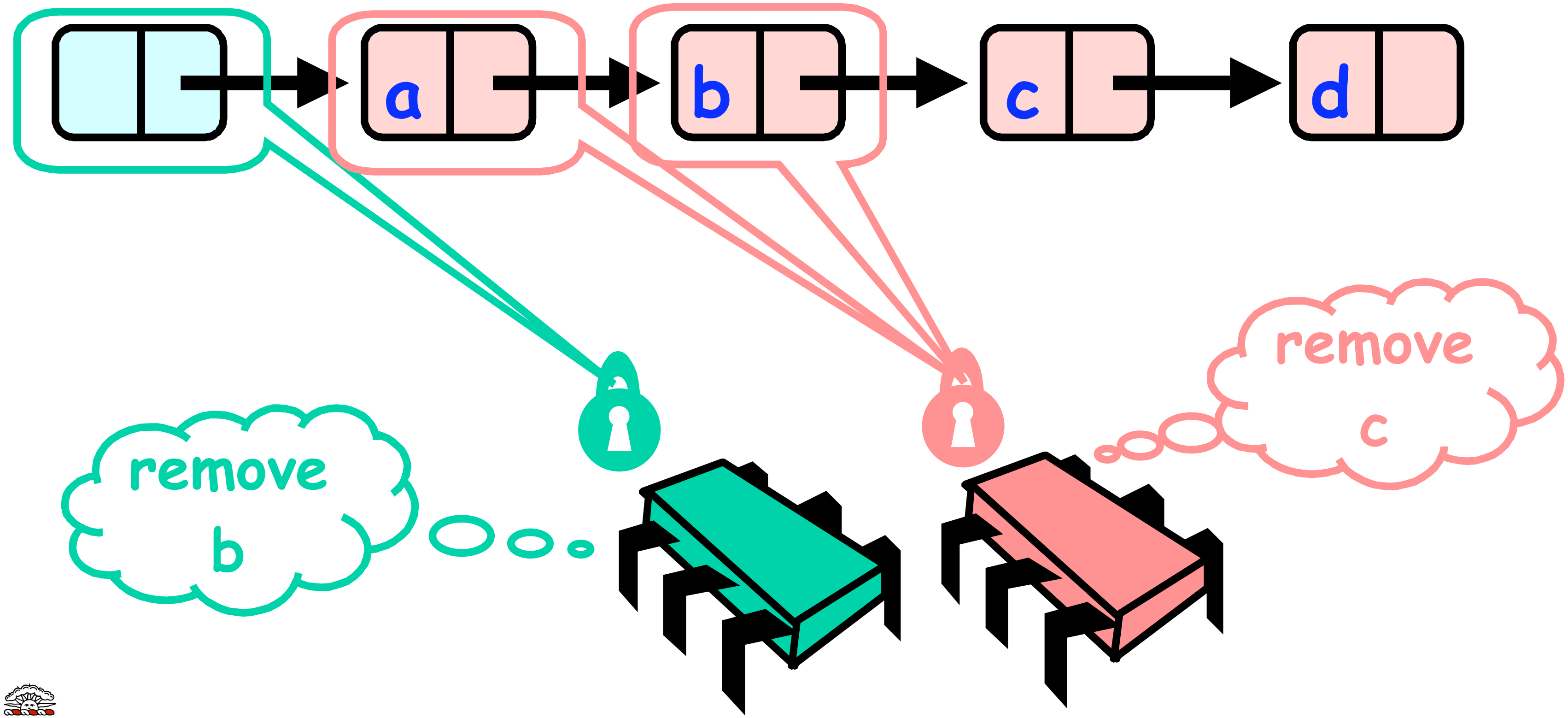
Removing an Entry



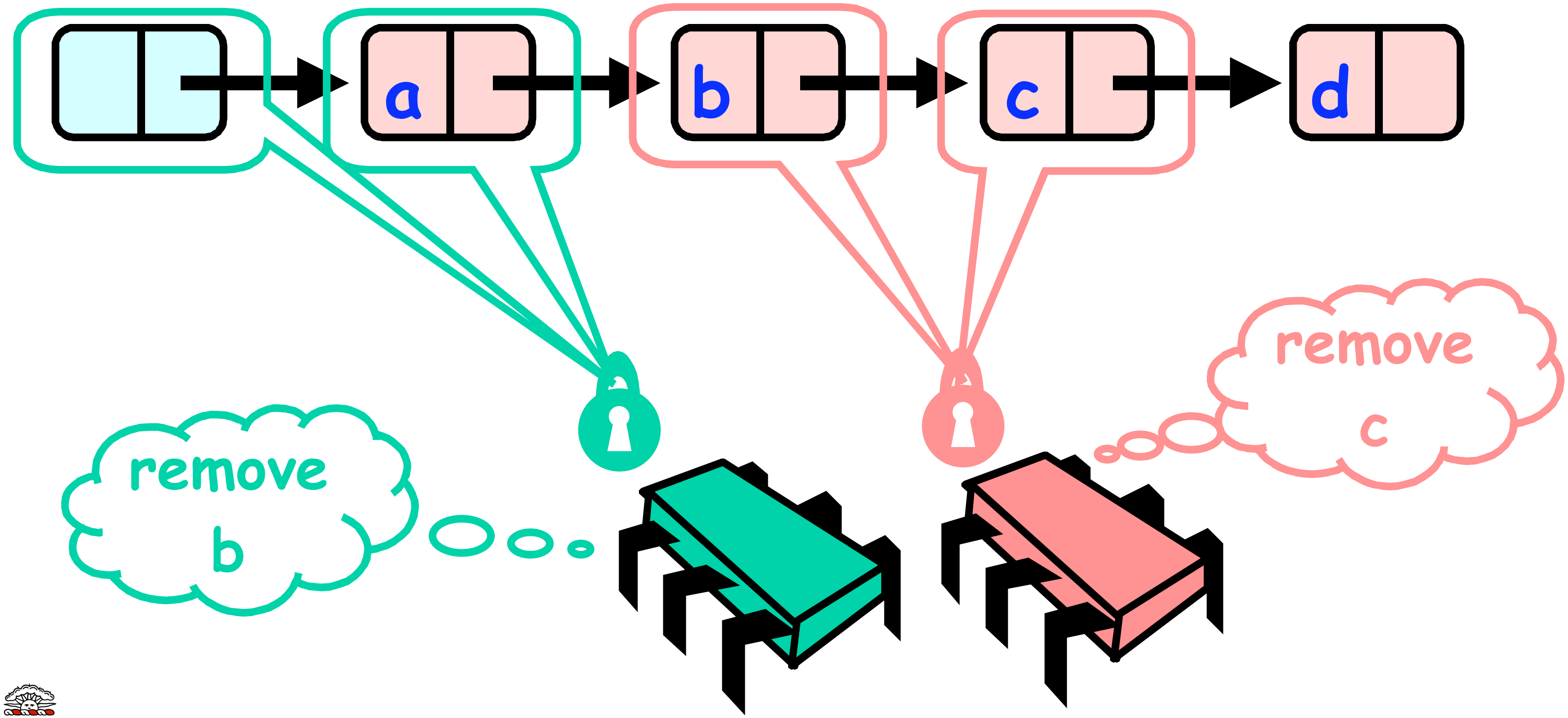
Removing an Entry



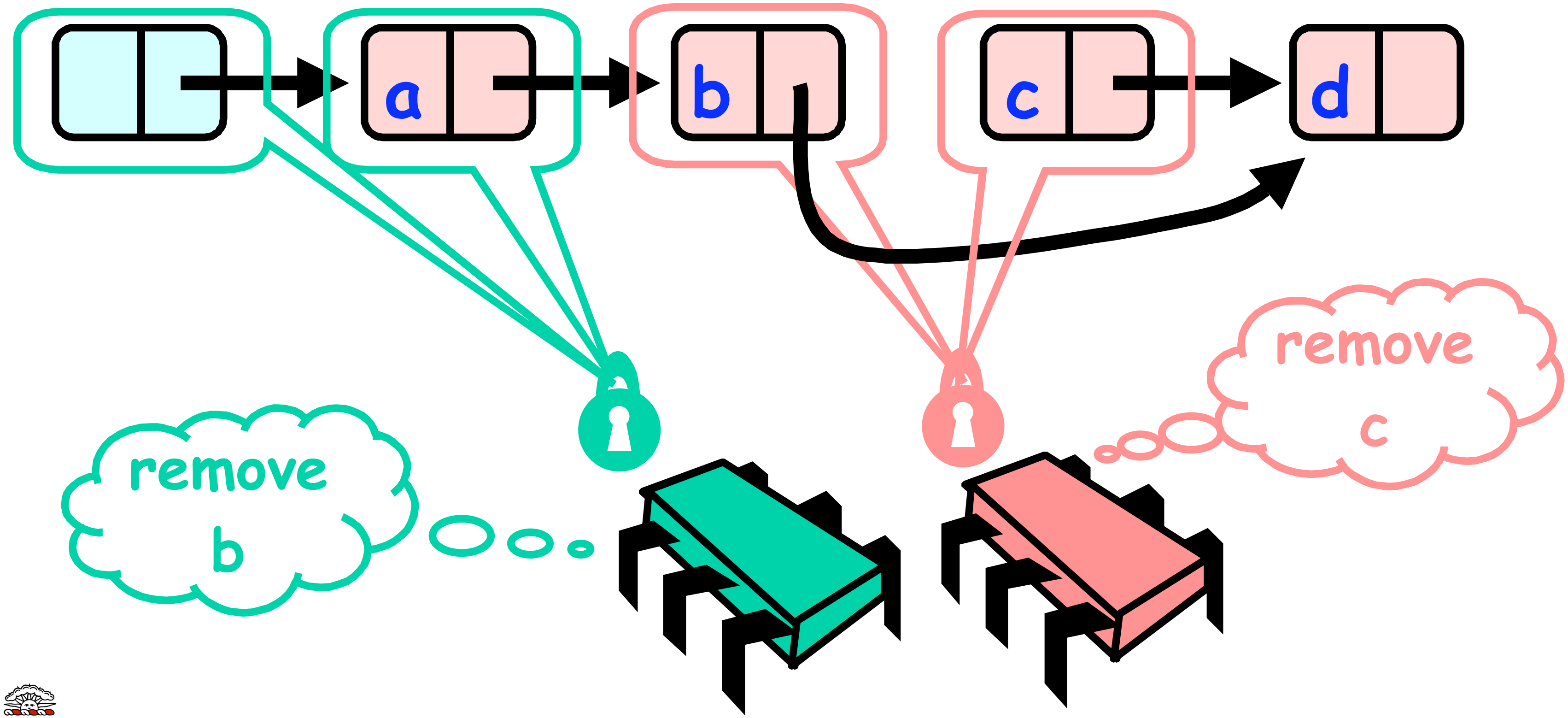
Removing an Entry



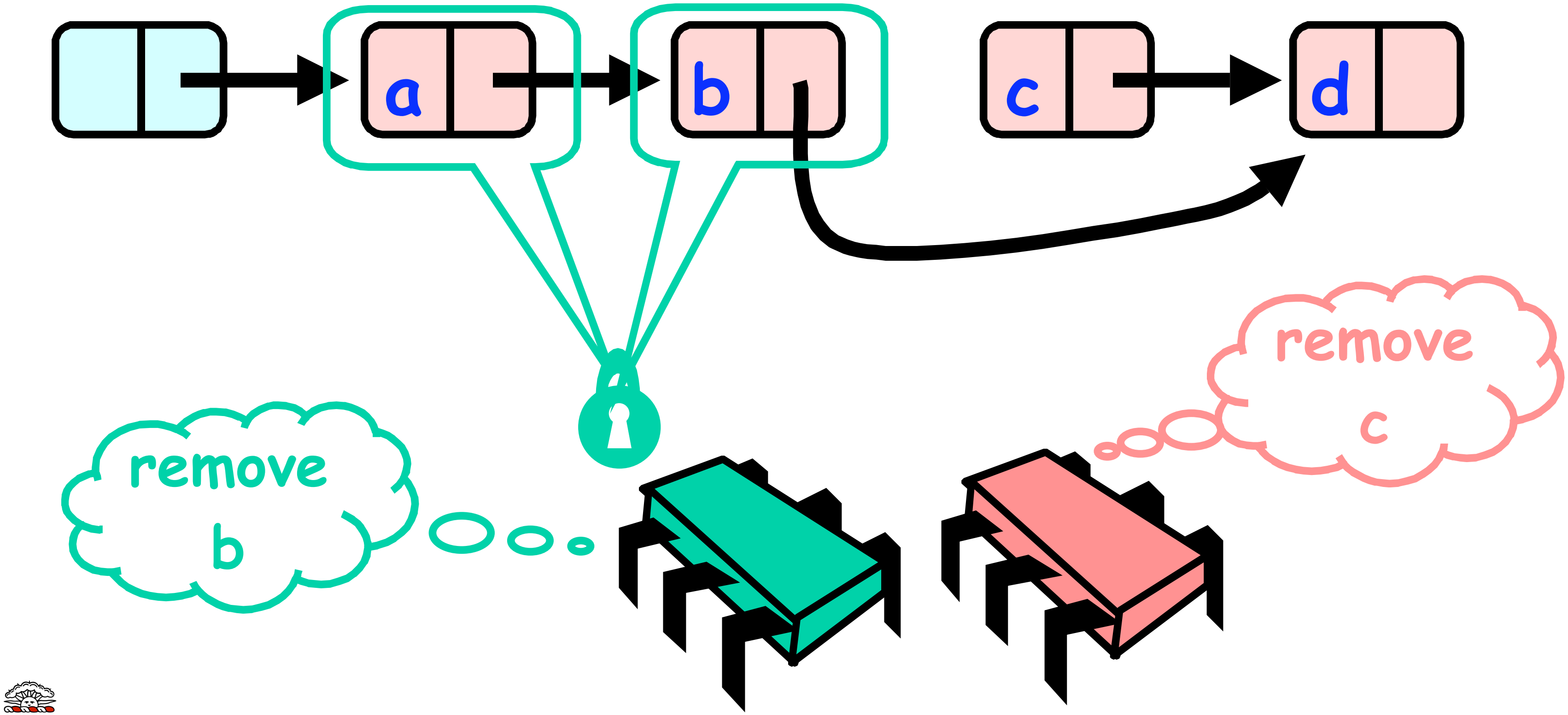
Removing an Entry



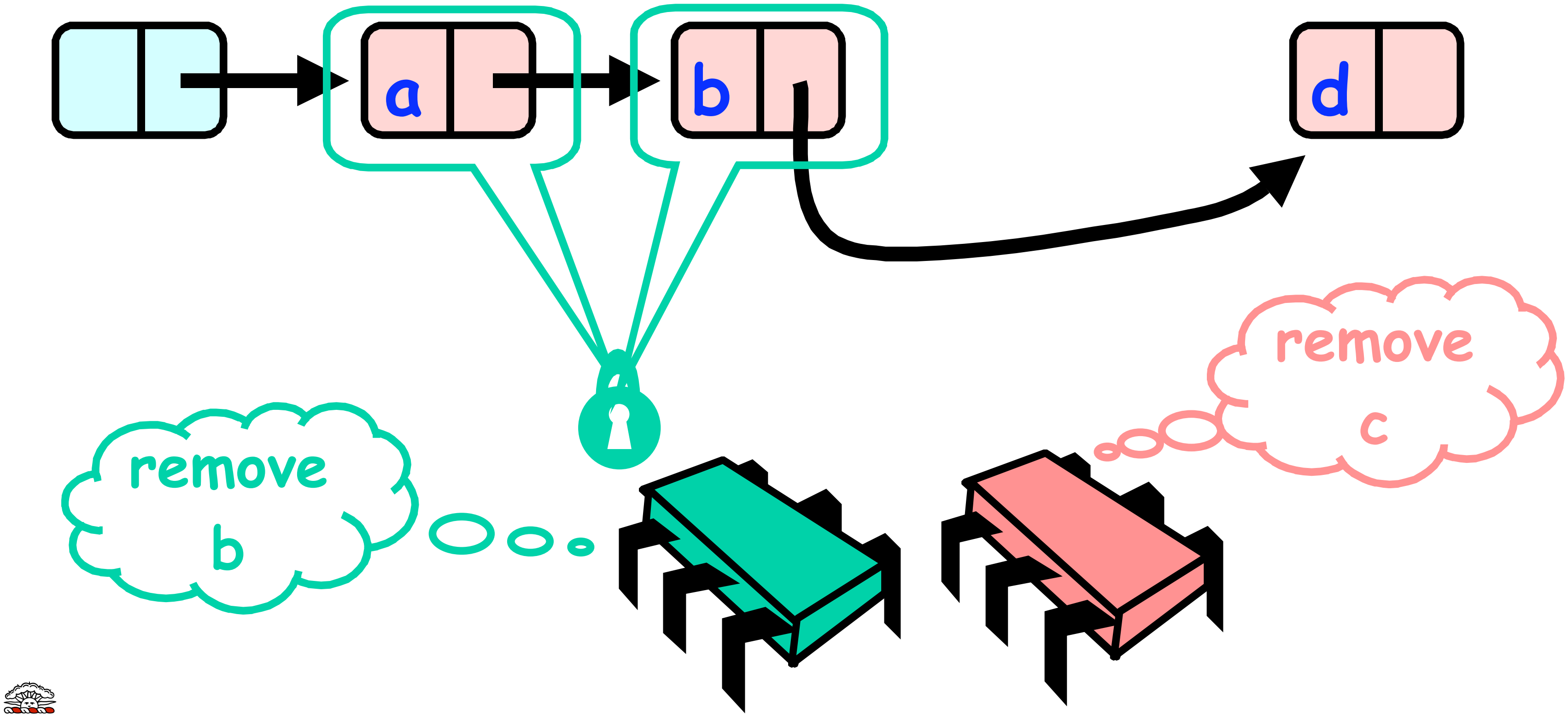
Removing an Entry



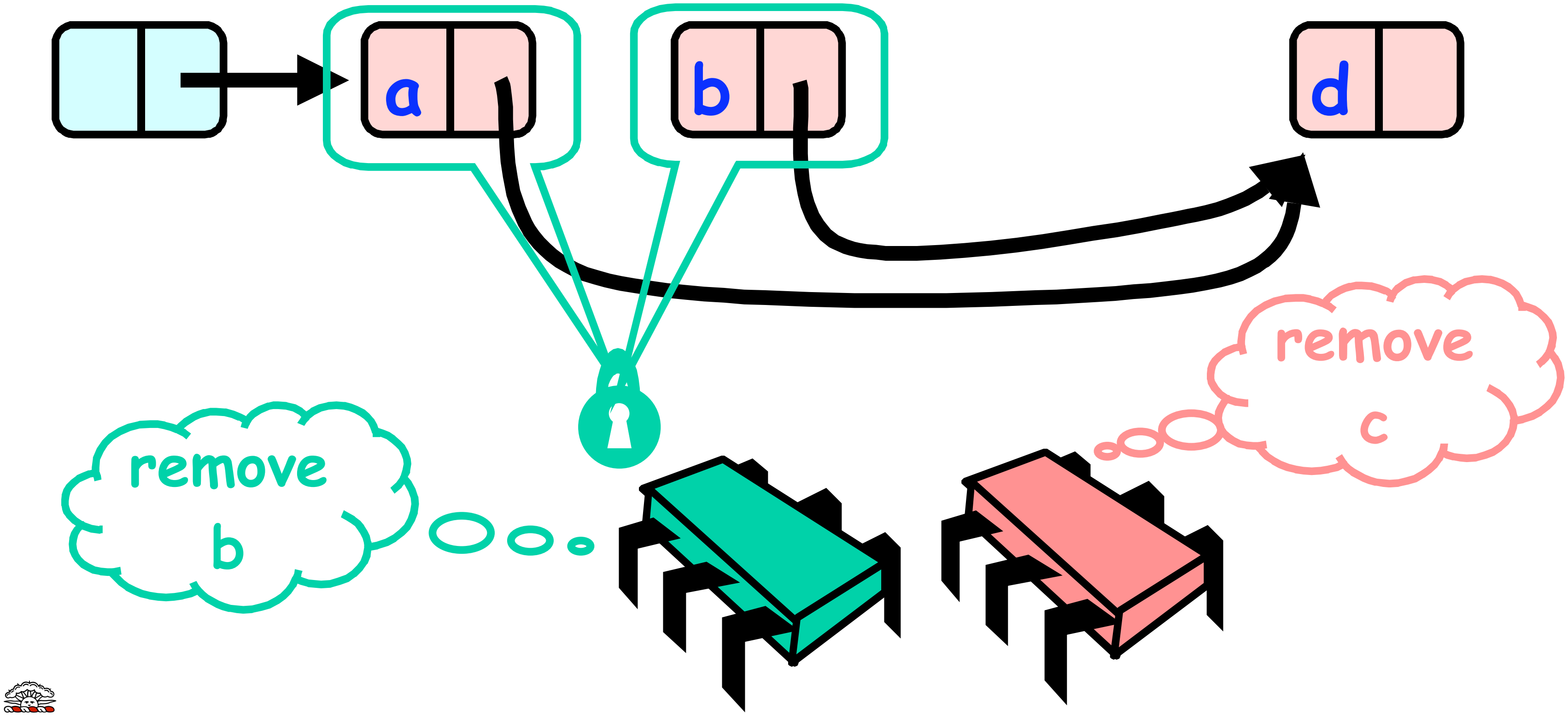
Removing an Entry



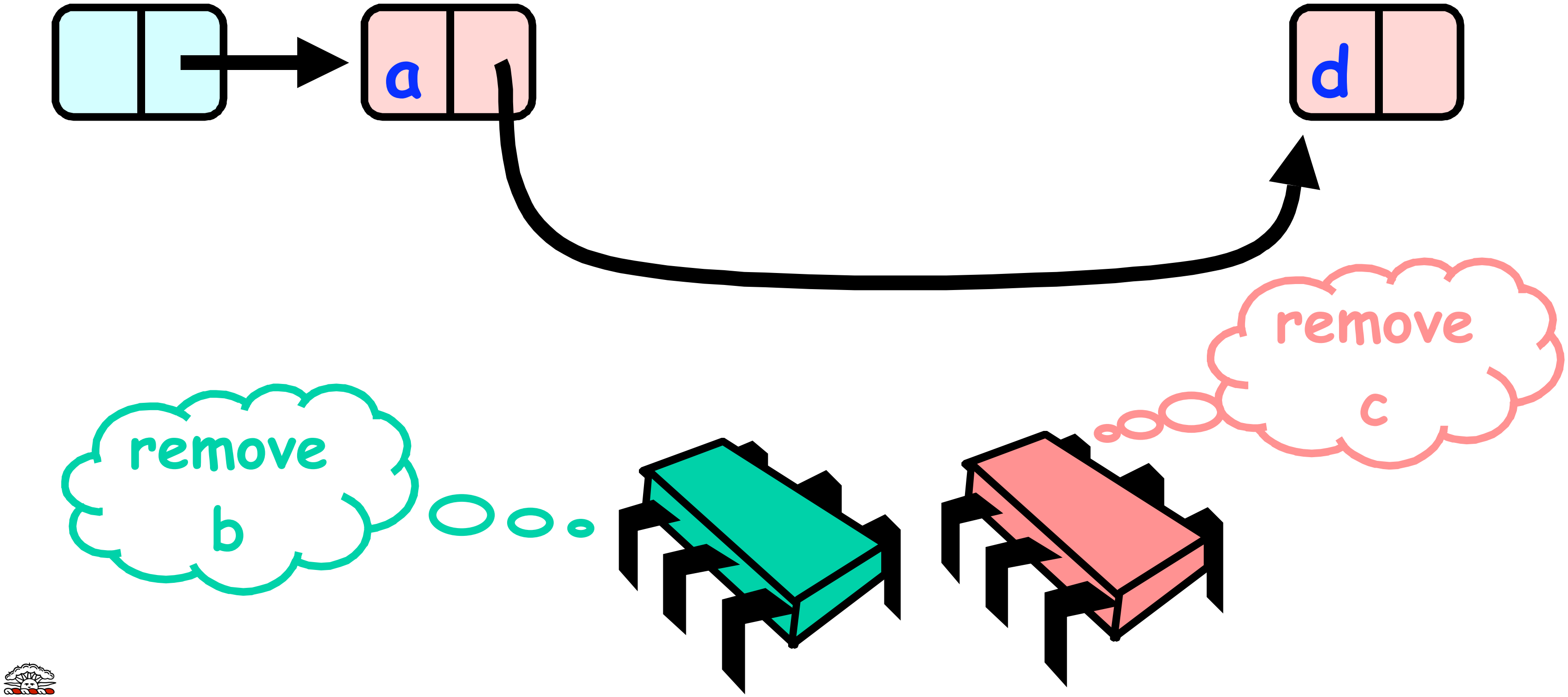
Removing an Entry



Removing an Entry



Removing an Entry



Remove method

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

Remove method

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

Key used to order entry

Remove method

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

Predecessor and current entries

Remove method

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

Make sure locks released

Remove method

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

Everything else

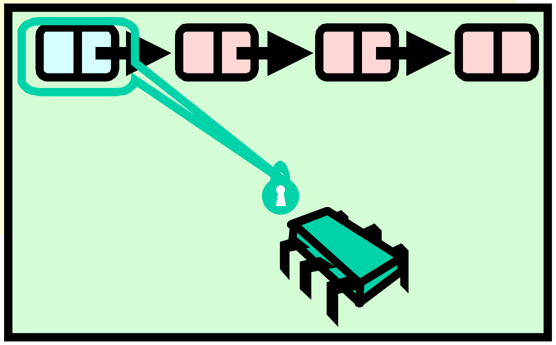
Remove method

```
try {
 pred = this.head;
 pred.lock();
 curr = pred.next;
 curr.lock();
 …
} finally { … }
```

Remove method

```
try {
 pred = this.head;
 pred.lock();
 curr = pred.next;
 curr.lock();
 …
} finally { … }
```

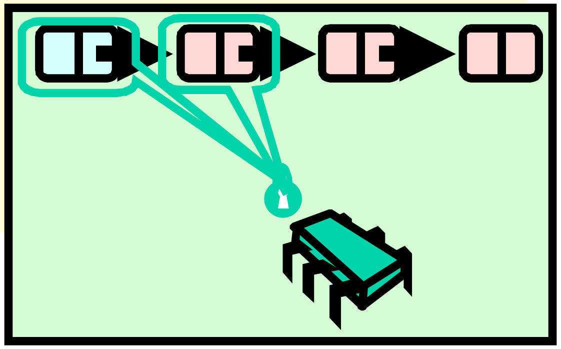
lock previous



Remove method

```
try {
 pred = this.head;
 pred.lock();
 curr = pred.next;
 curr.lock();
 …
} finally { … }
```

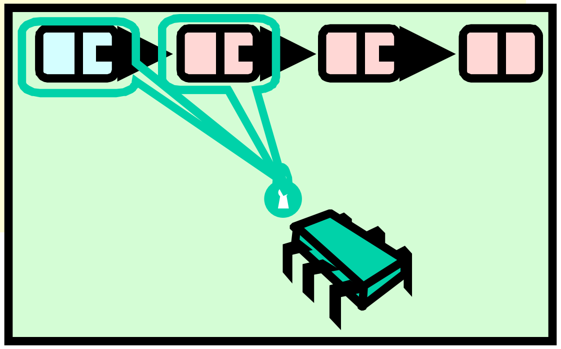
Lock current



Remove method

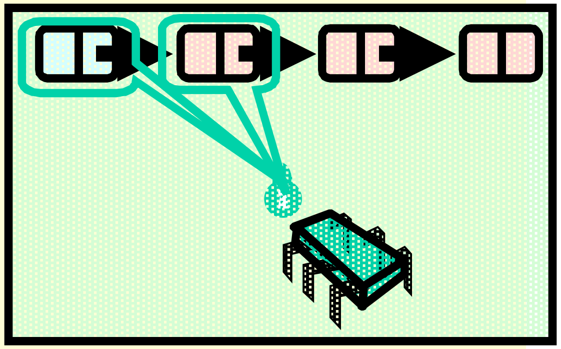
```
try {
 pred = this.head;
 pred.lock();
 curr = pred.next;
 curr.lock();
 ...
} finally { … }
```

Traversing list



Remove: searching

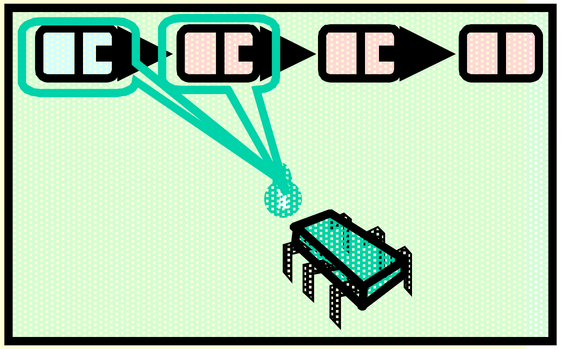
```
while (curr.key <= key) {
  if (object == curr.object) {
   pred.next = curr.next;
   return true;
  }
  pred.unlock();
  pred = curr;
  curr = curr.next;
  curr.lock();
 }
 return false;
```



Remove: searching

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

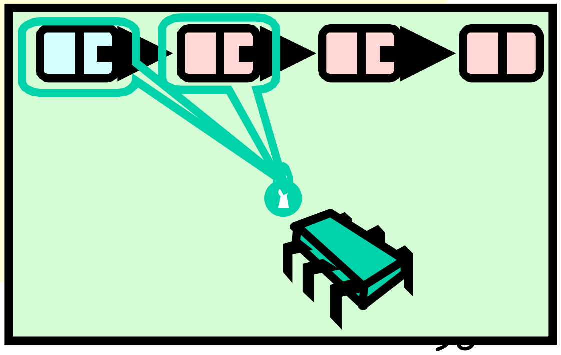
Search key range



Remove: searching

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

At start of each loop: curr and predy locked

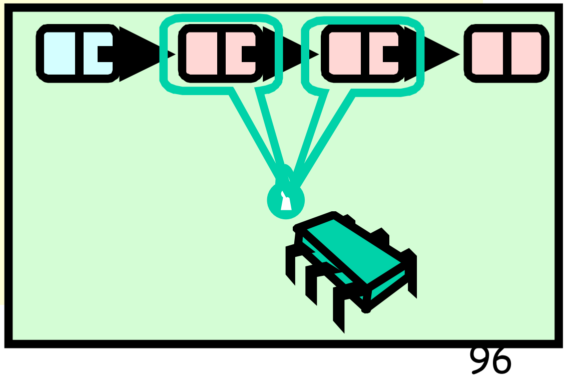


BROWN

Remove: searching

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

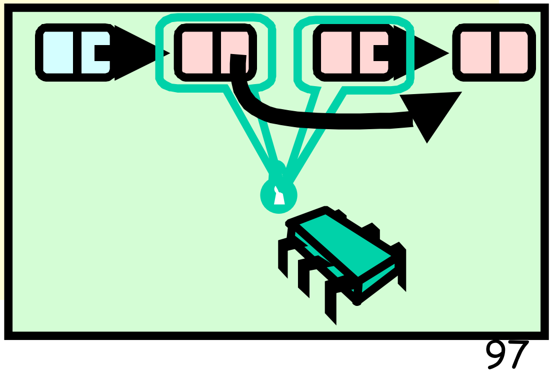
If entry found, remove it



Remove: searching

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

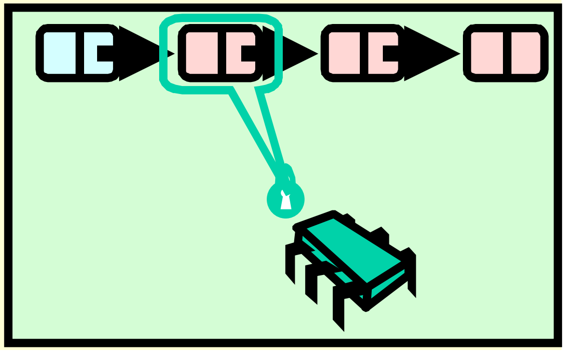
If entry found, remove it



Remove: searching

Unlock predecessor

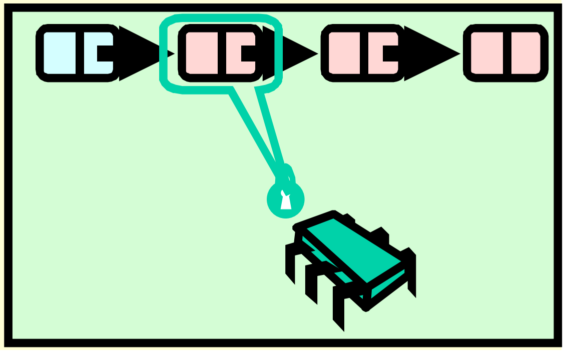
```
while (curr.key <= key) {
  if (object == curr.object) {
   pred.next = curr.next;
   return true;
  }
  pred.unlock();
  pred = curr;
  curr = curr.next;
  curr.lock();
 }
 return false;
```



Remove: searching

Only one entry locked!

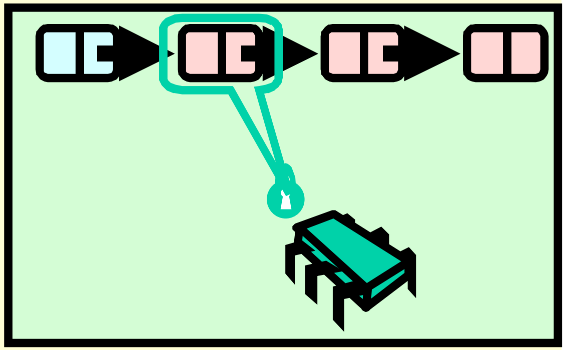
```
while (curr.key <= key) {
  if (object == curr.object) {
   pred.next = curr.next;
   return true;
  }
  pred.unlock();
  pred = curr;
  curr = curr.next;
  curr.lock();
 }
 return false;
```



Remove: searching

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

demote current



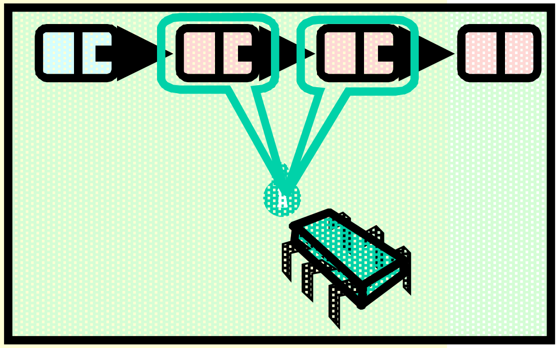
BROWN

© 2005 Herlihy & Shavit

Remove: searching

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

Find and lock new current

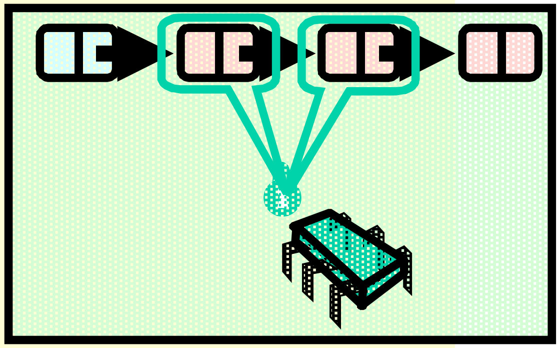


BROWN

Remove: searching

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

Lock invariant restored



Remove: searching

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

Otherwise, not present

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

First Invariant

- Different threads have different **pred** values
- If $A \neq B$, and **pred**$_A \neq$null
  - Then **pred**$_A \neq$ **pred**$_B$

1st Invariant

- If A ≠B, and $\textbf{pred}_A \neq$ null
  - Then $\textbf{pred}_A \neq \textbf{pred}_B$
- Holds initially
- Must show it is preserved

Claim

- If **pred**$_A$ ≠ null then A holds lock
  - True at start when **pred**$_A$ is **head**
  - **curr**$_A$ locked before assigned to **pred**$_A$
  - Other statements don't change **pred**$_A$

1st Invariant

- If **pred**$_A$ ≠null
  - then A holds lock
- If **pred**$_B$ ≠null
  - then B holds lock
- Must be distinct

2^{nd} Invariant

- Threads never traverse deleted entries
- If **pred**$_A$ ≠ null
  - Then **head** ⇒ **pred**$_A$ ⇒ **tail**

2^{nd} Invariant

- True initially
- *A* holds lock for **pred**$_A$ throughout traversal
- No other thread can remove it
- So **head** ⇒ **pred**$_A$ is invariant.
- Same for **pred**$_A$ ⇒ **tail**

Why remove() is linearizable

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

$\textbf{head} \Rightarrow \textbf{pred}_A \rightarrow \textbf{curr}_A$

so the object is in the set

Why remove() is linearizable

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

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

Why remove() is linearizable

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

Linearization point

Why remove() is linearizable

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

Object not present

Why remove() is linearizable

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

$\mathbf{pred}_A \rightarrow \mathbf{curr}_A$

$\mathbf{pred}_A.\mathrm{key} < \mathrm{key}$

$\mathrm{key} < \mathbf{curr}_A.\mathrm{key}$

Why remove() is linearizable

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

Linearization point: when curr_A set to entry with higher key

Adding Entries

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

Rep Invariant

- Easy to check that
  - Tail always reachable from head
  - Entries 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 entries without locking
- Lock entries
- Check that everything is OK

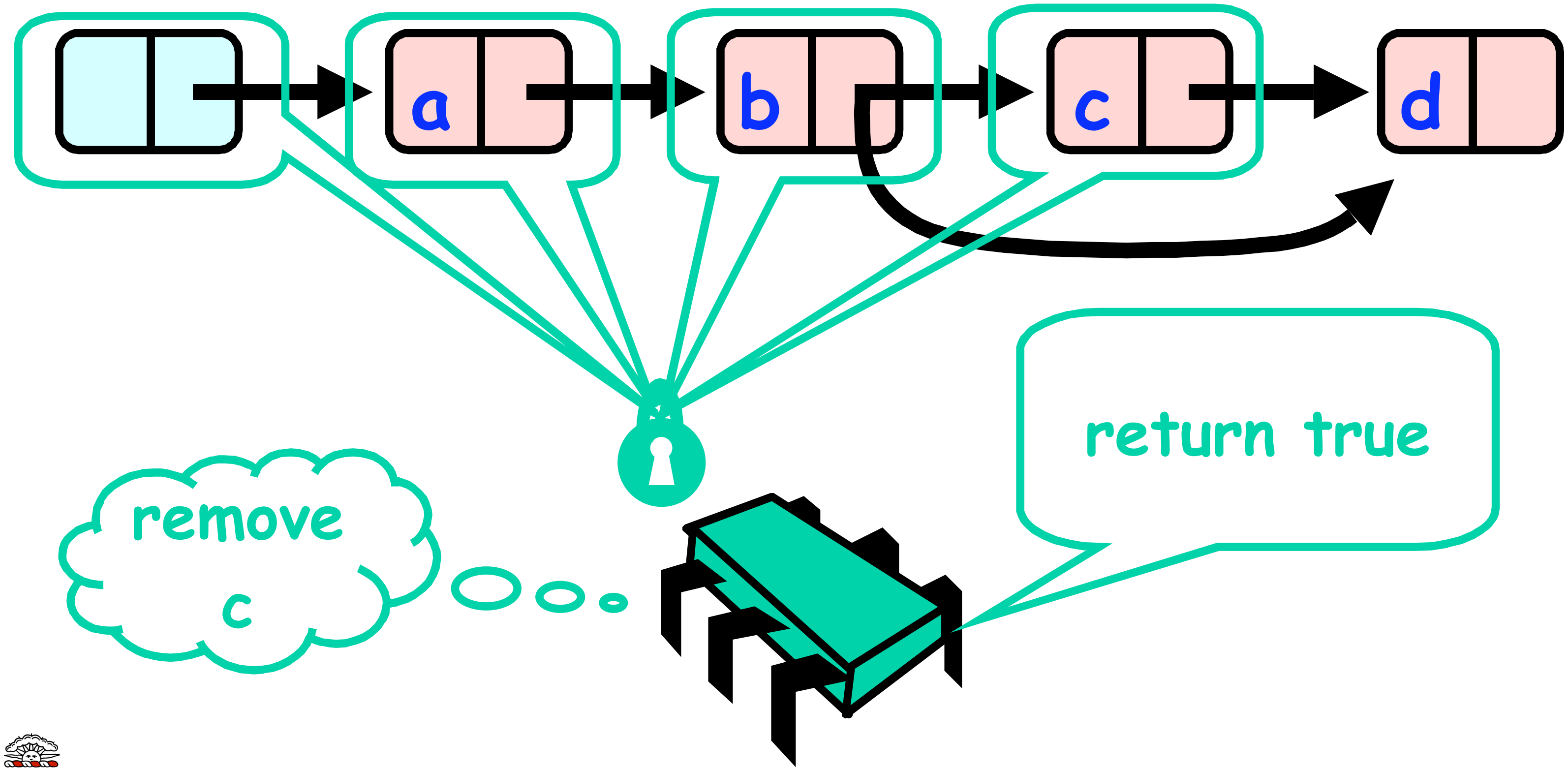
Invariants

- Invariants no longer hold
  - OK to scan deleted elements
- But we establish properties by
  - Validation
  - After we lock target entries

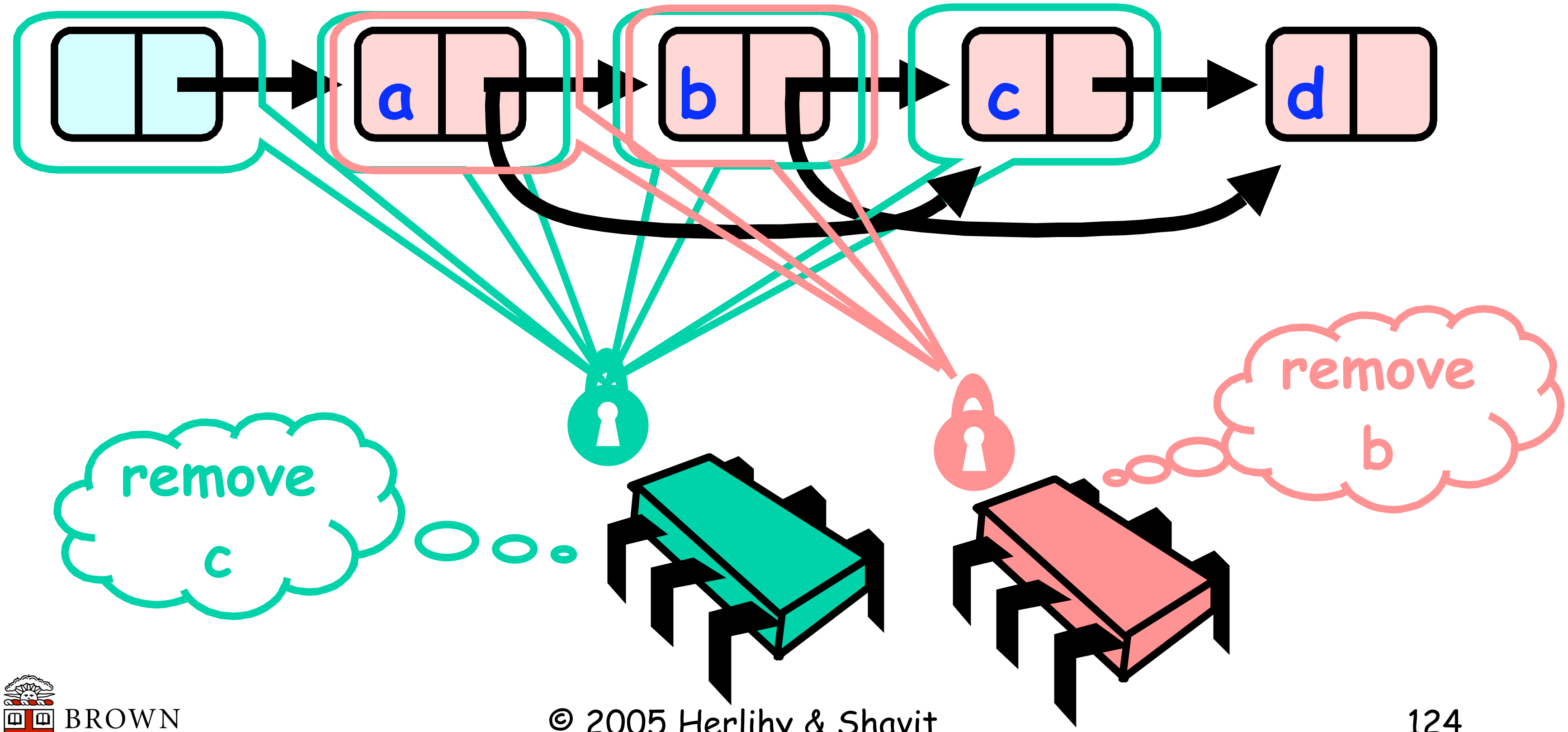
Key Property

- Fine-grained synchronization
  - **head** ⇒ **pred**$_{A}$ ⇒ **tail**
  - Is invariant
- Optimistic synchronization
  - Validation checks same property
  - After the fact
  - Must restart if validation fails

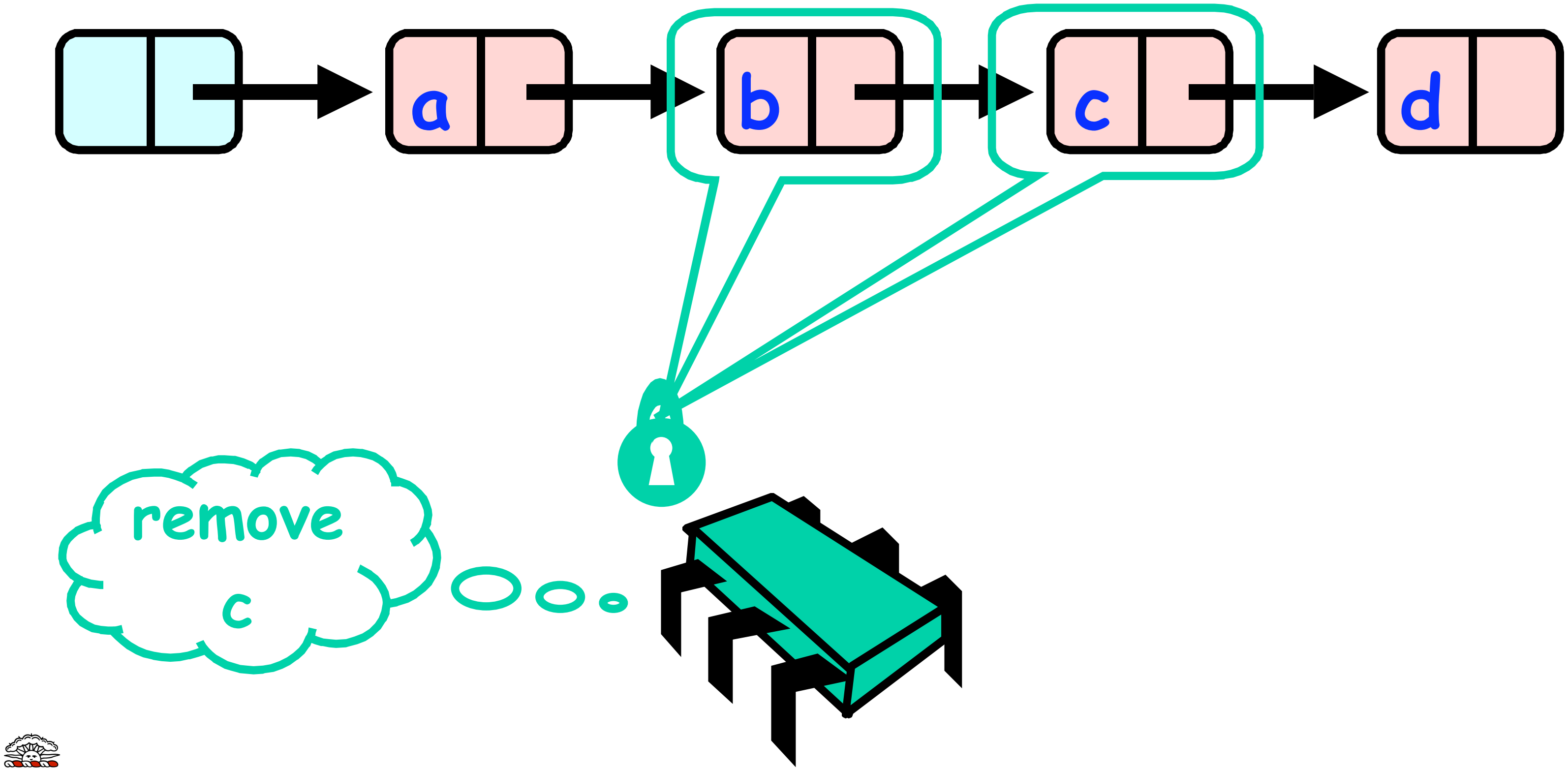
Removing an Entry



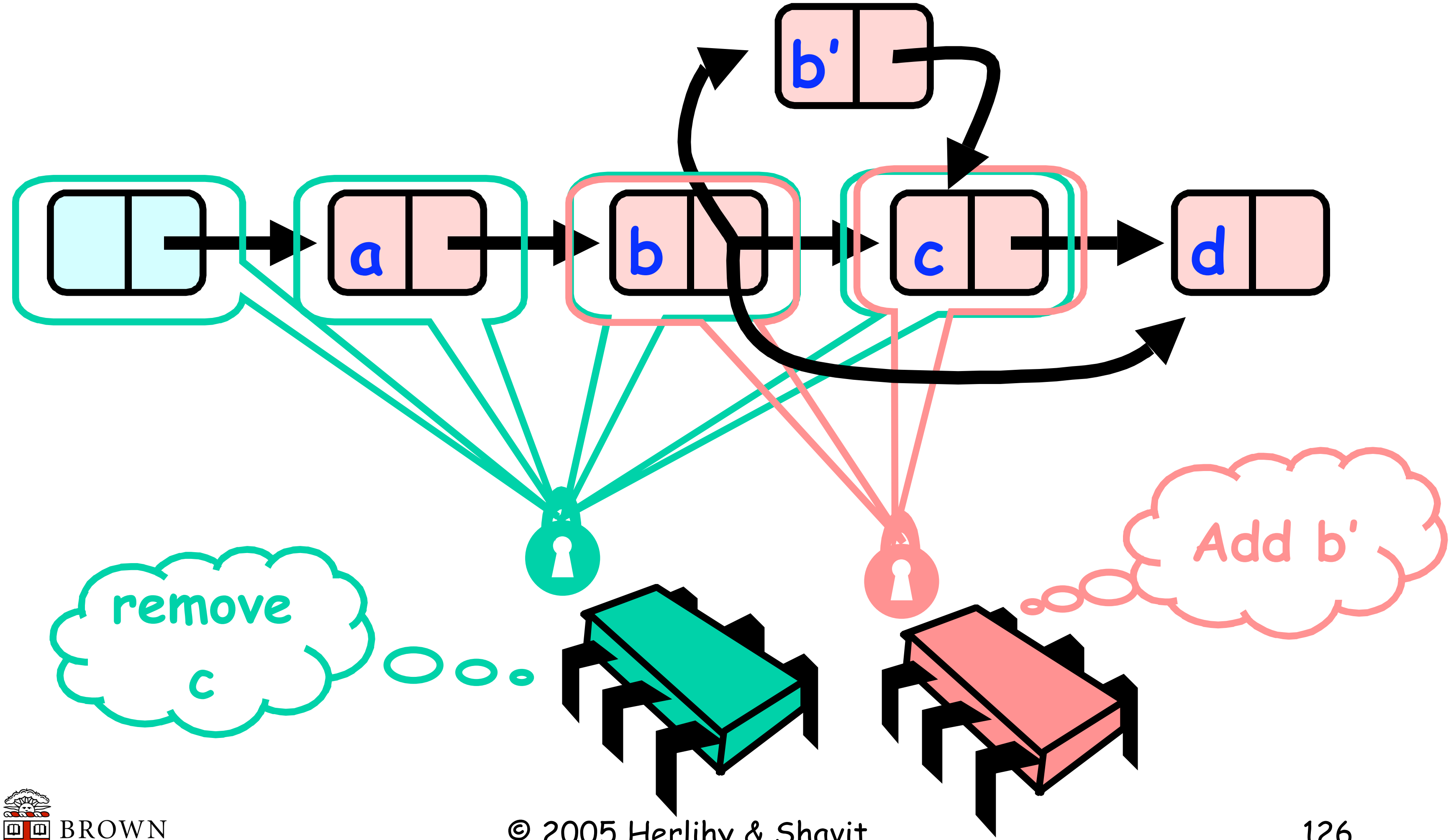
What Can Go Wrong?



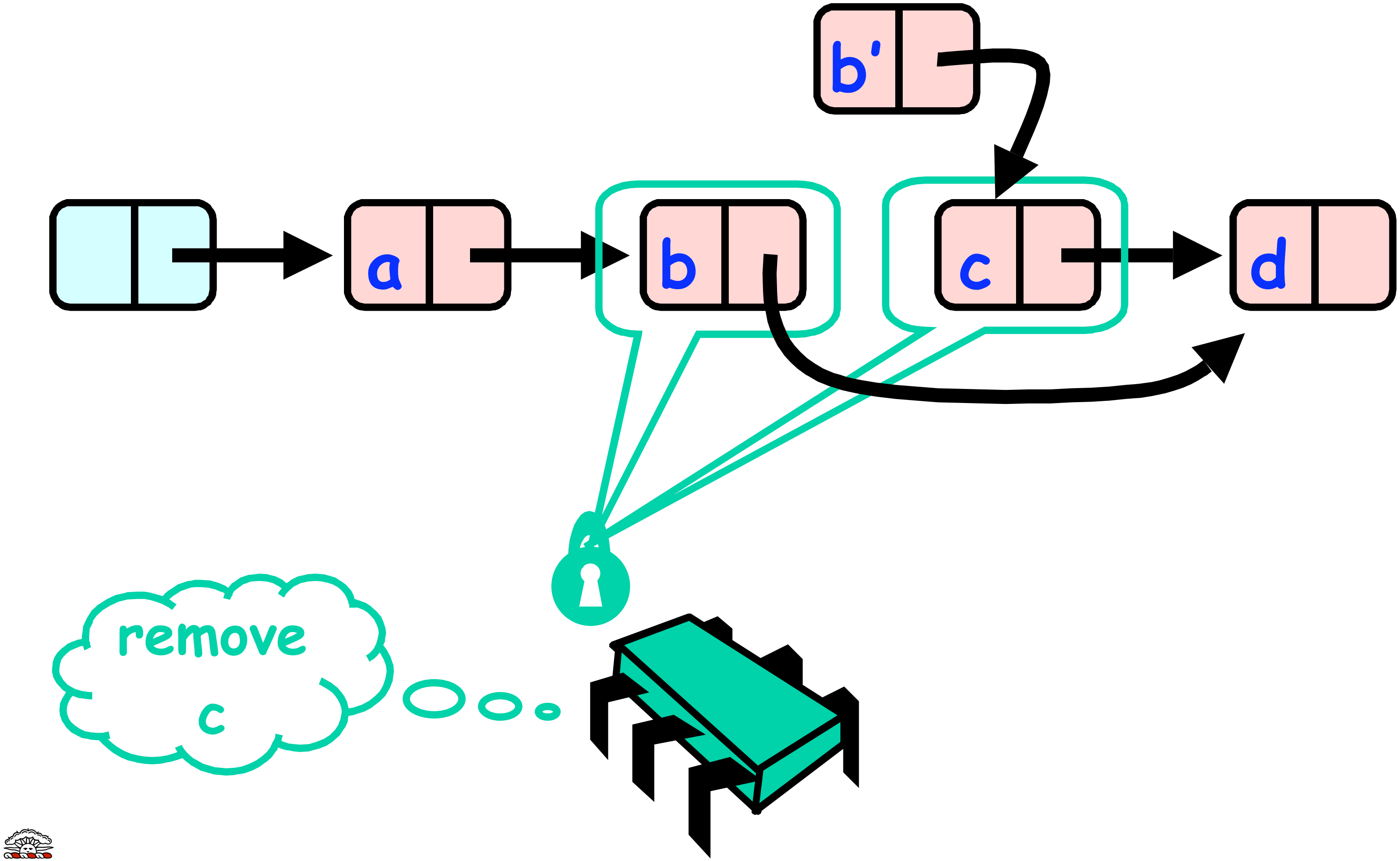
Check that Entry is Still Accessible



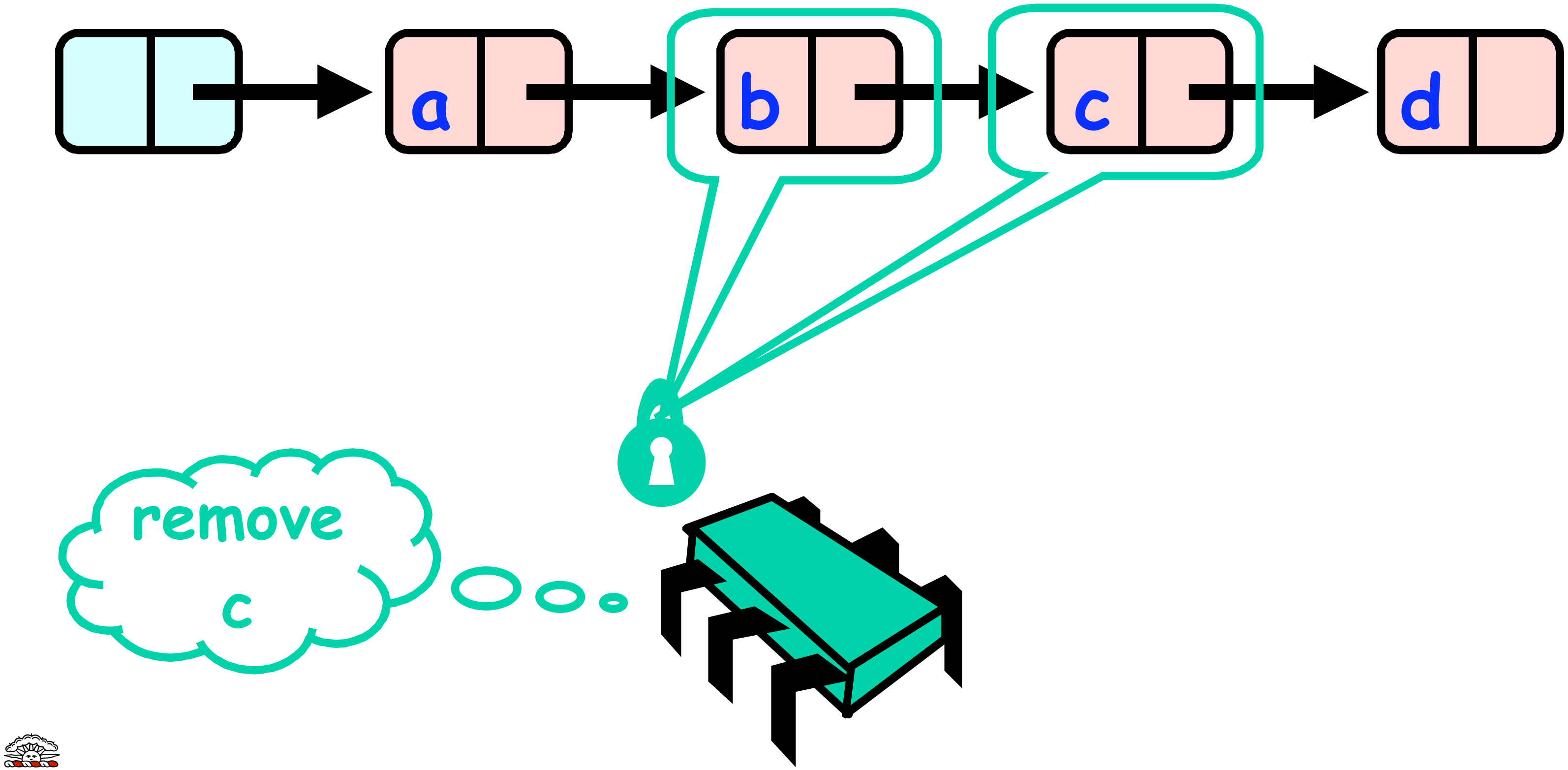
What Can Go Wrong?



What Can Go Wrong?



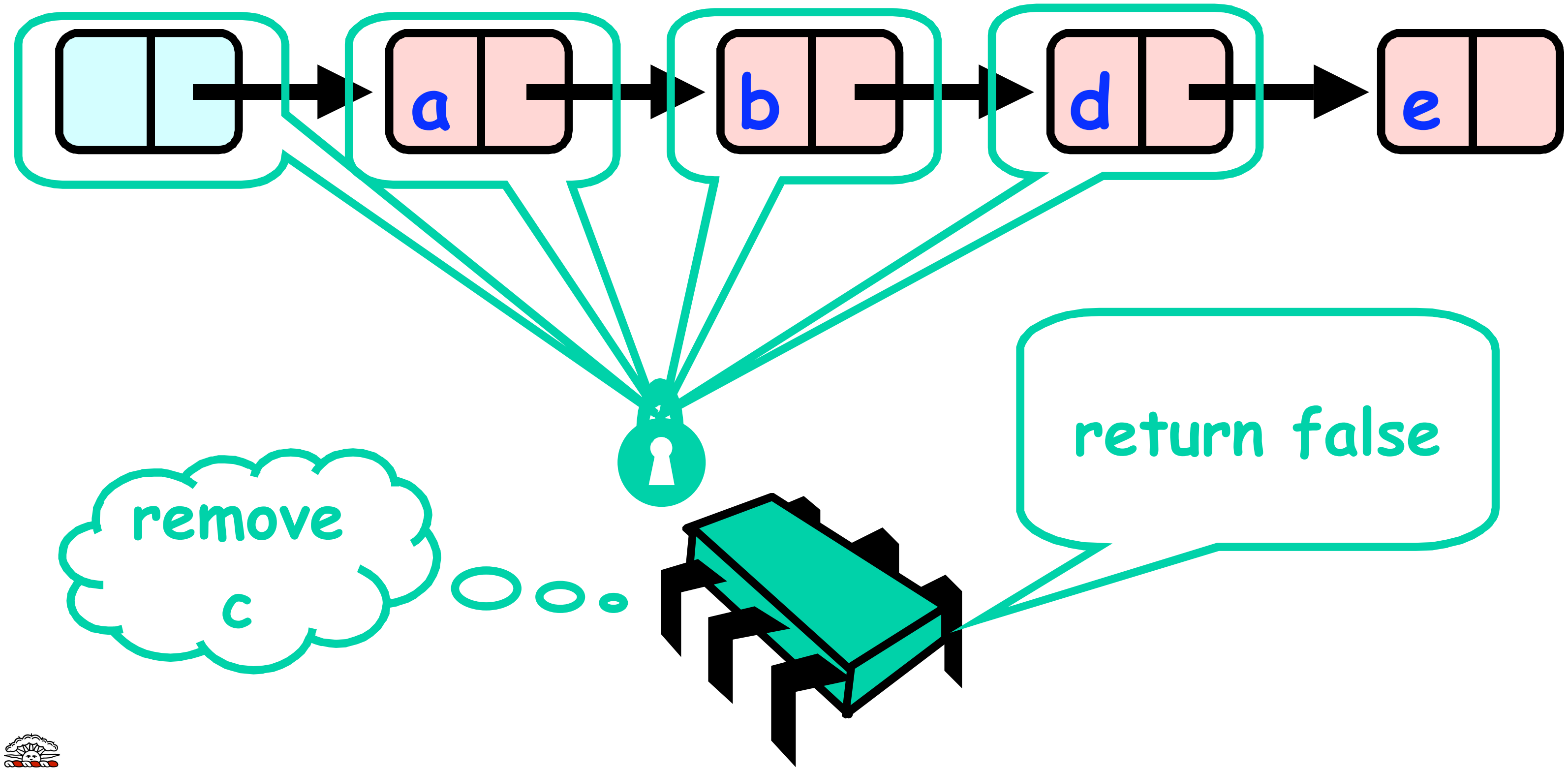
Check that Entries Still Adjacent



Correctness

- If
  - Entries b and 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
  - Entries b and 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

1st Invariant

- Different threads have different **pred** values if they're locked
- If $A \neq B$, and **pred**$_A \neq$null and locked
  - Then **pred**$_A \neq$ **pred**$_B$

2nd Invariant

- An entry will remain reachable from **pred**$_A$ as long as it is reachable from the head
- For all reachable a,
  - If **pred**$_A$ ≠ **null**, **pred**$_A$**.key < a.key**
  - Then **pred**$_A \Rightarrow$ **a**

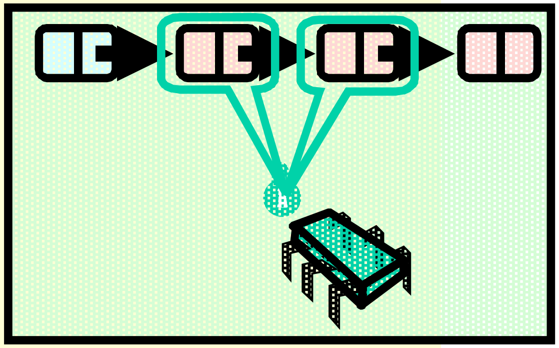
Validation

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

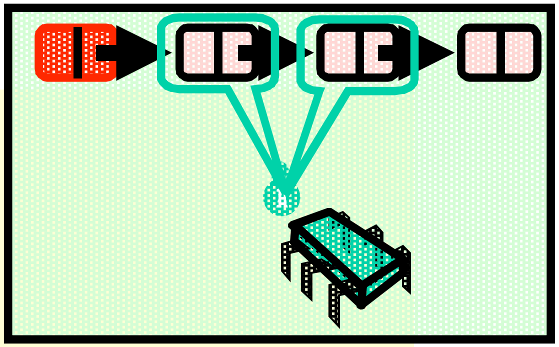
Validation

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

Predecessor & current entries



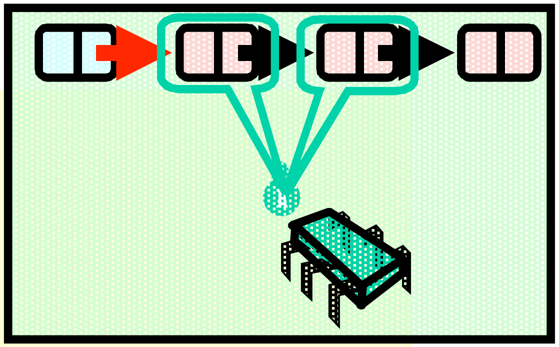
Validation



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

Start at the beginning

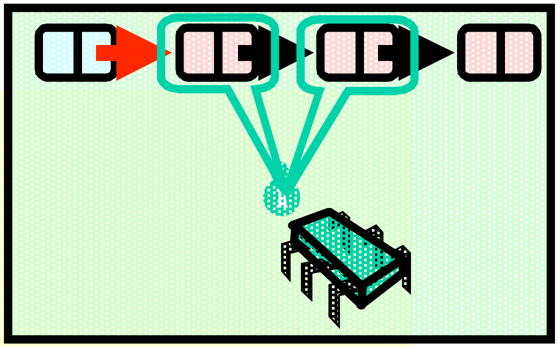
Validation



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

Search range of keys

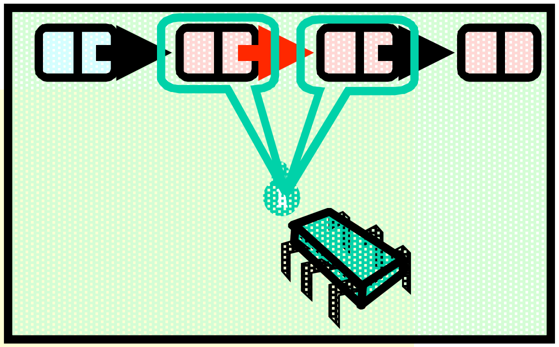
Validation



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

Predecessor reachable

Validation



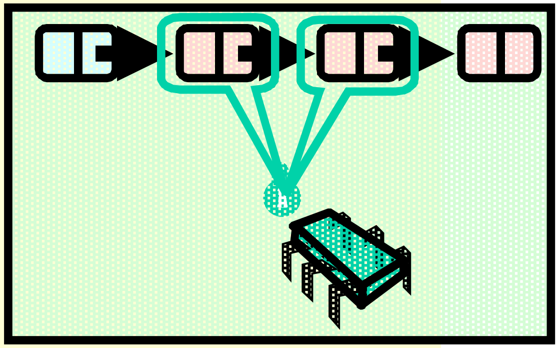
```
private boolean
 validate(Entry pred,
          Entry curry) {
 Entry entry = head;
 while (entry.key <= pred.key) {
  if (entry == pred)
   return pred.next == curr;
  entry = entry.next;
 }
 return false;
}
```

Is current entry next?

Validation

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

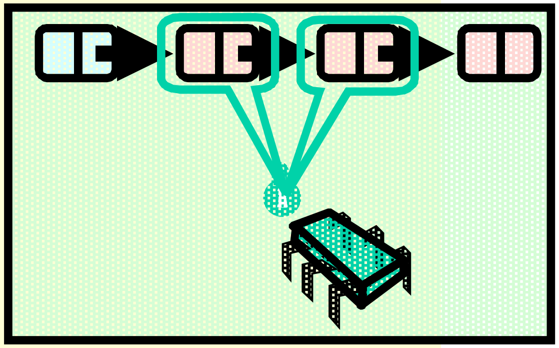
Otherwise move on



Validation

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

Predecessor not reachable



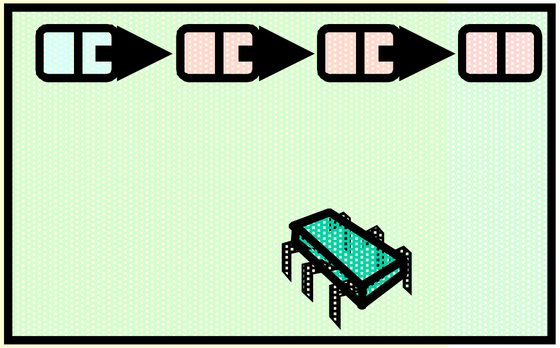
Remove: searching

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

Remove: searching

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

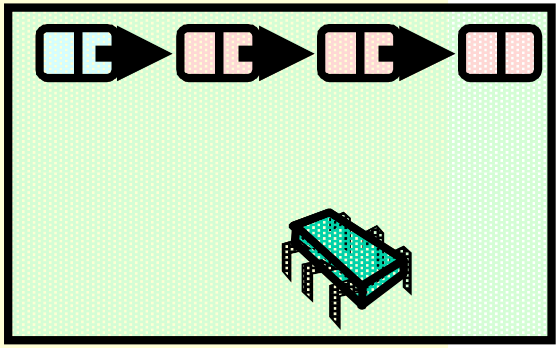
Search key



Remove: searching

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

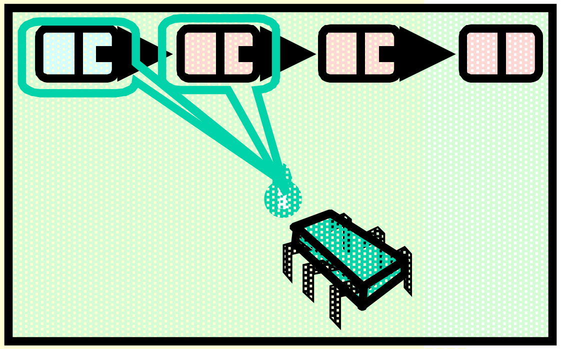
Retry on synchronization conflict



Remove: searching

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

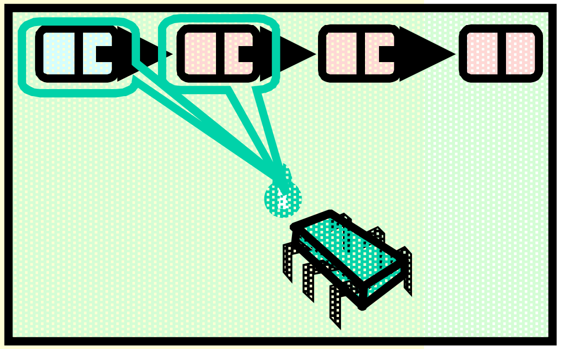
Examine predecessor and current entries



Remove: searching

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

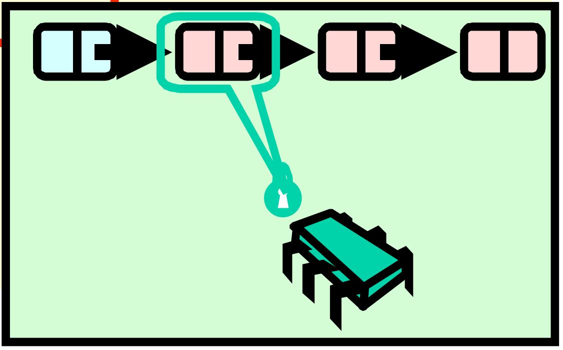
Search by key



Remove: searching

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

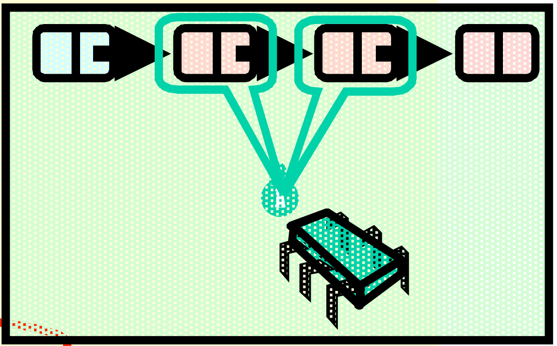
Stop if we find object



Remove: searching

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

Move along



On Exit from Loop

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

Remove Method

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

Remove Method

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

Always unlock

Remove Method

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

Lock both entries

Remove Method

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

Check for synchronization conflicts

Remove Method

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

Object found, remove entry

Remove Method

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

Evaluation

- Optimistic works if cost of
  - scanning twice without locks <
  - Scanning once with locks
- Drawback
  - Contains() acquires locks
  - 90% of calls in many apps

Lazy List

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

Lazy List

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

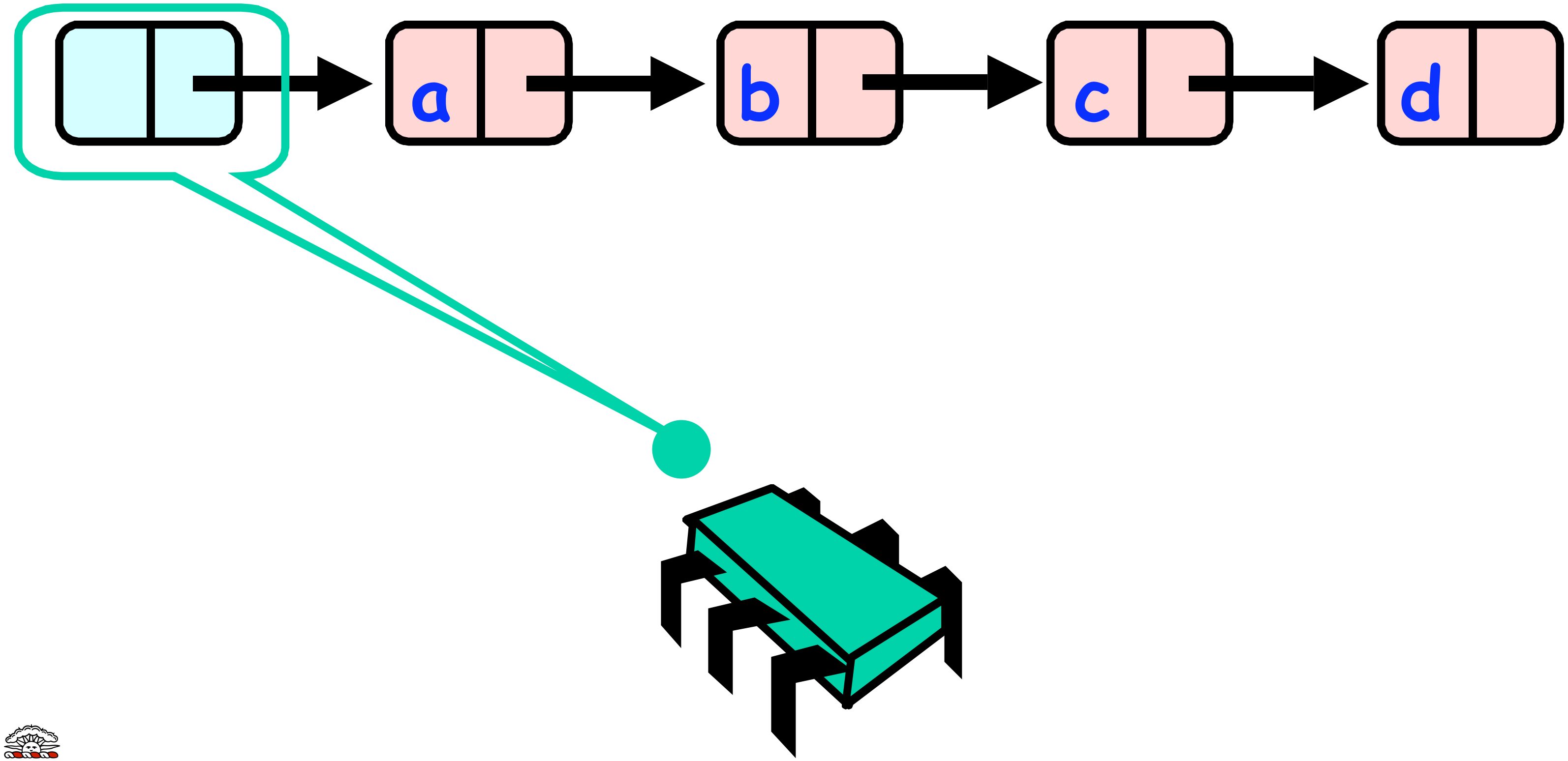
Lazy List

- All Methods
  - Scan through marked entry
  - Removing an entry doesn't slow down other method calls …
- Must still lock pred and curr entries.

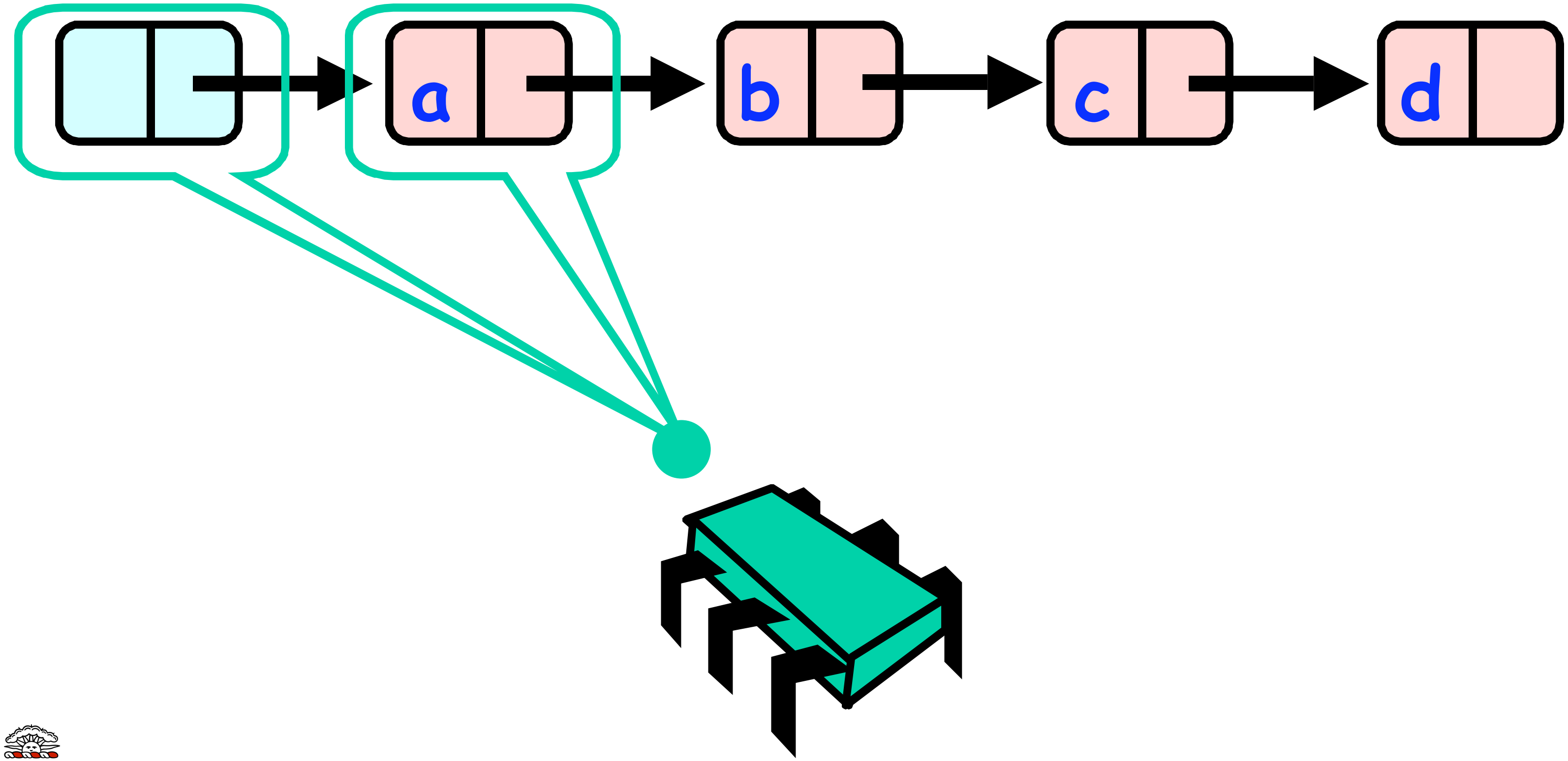
Validation

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

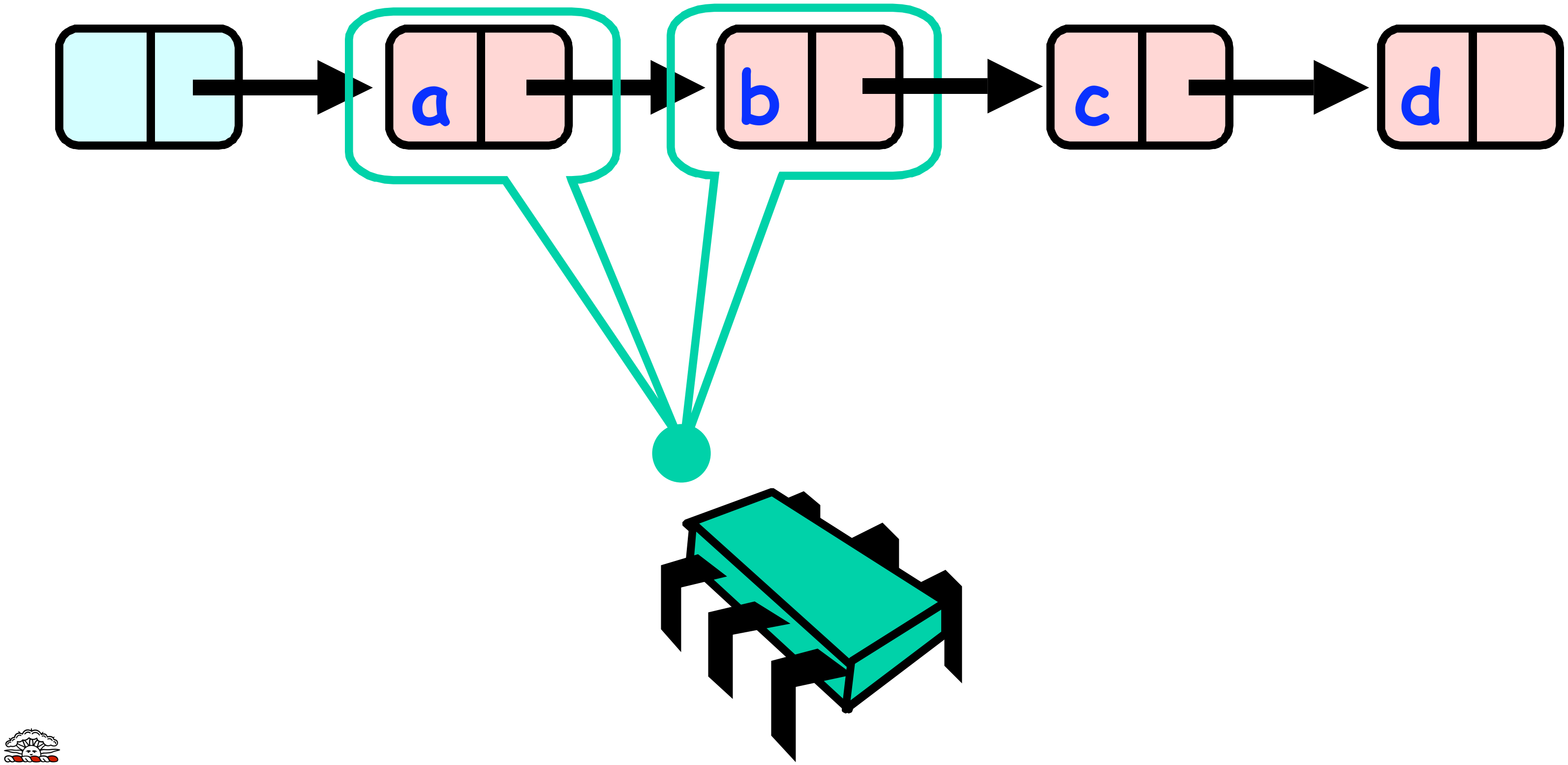
Business as Usual



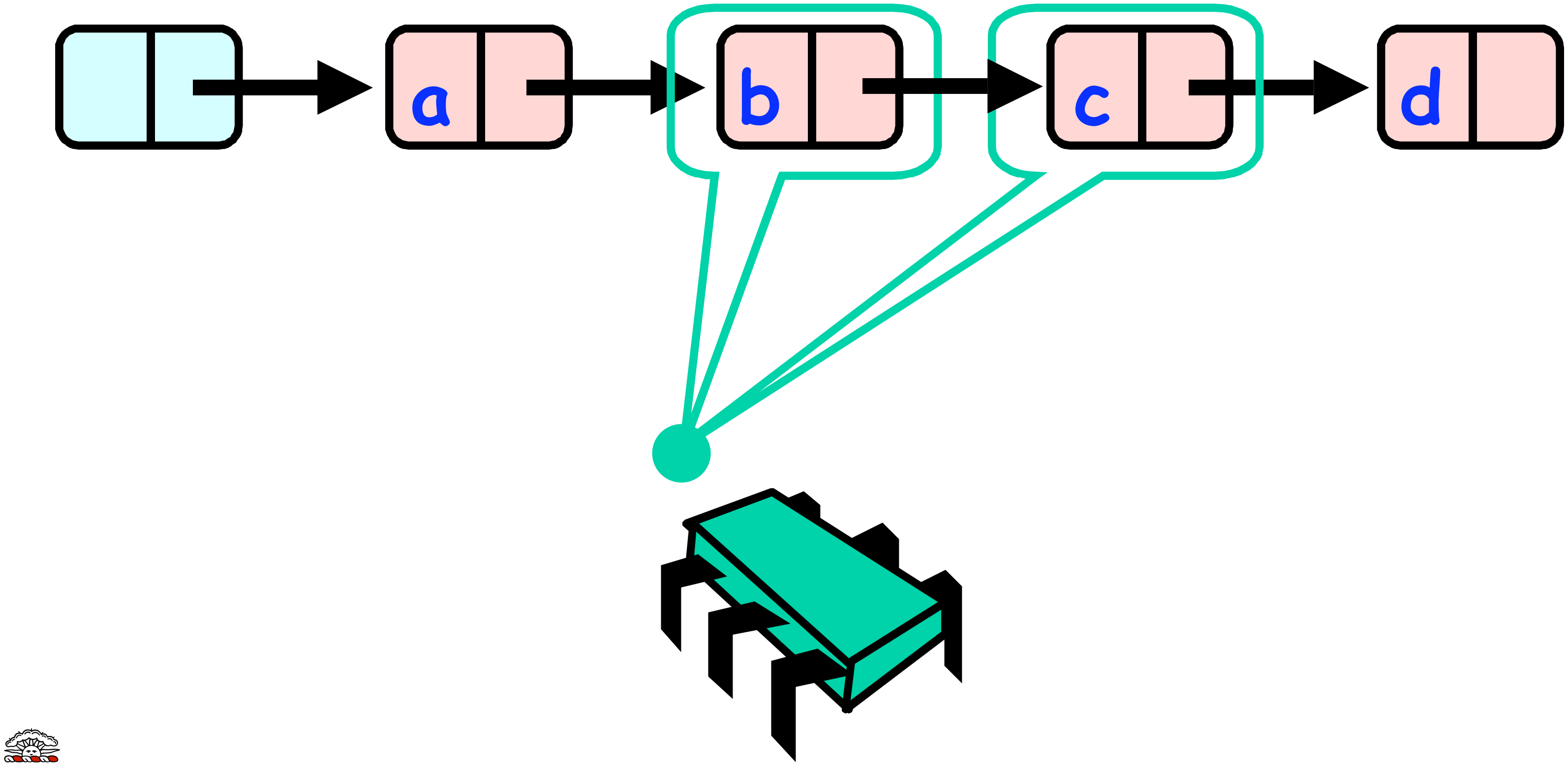
Business as Usual



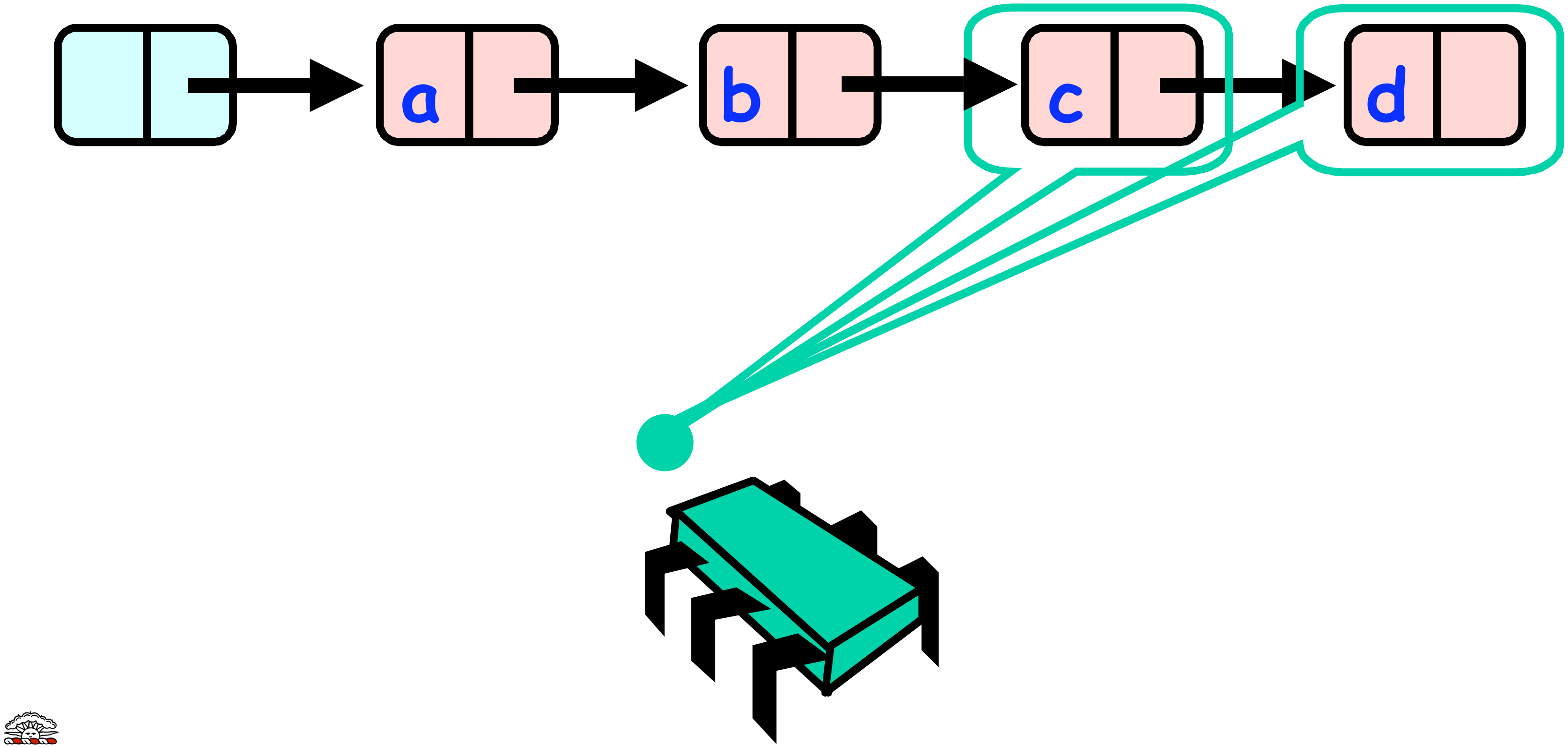
Business as Usual



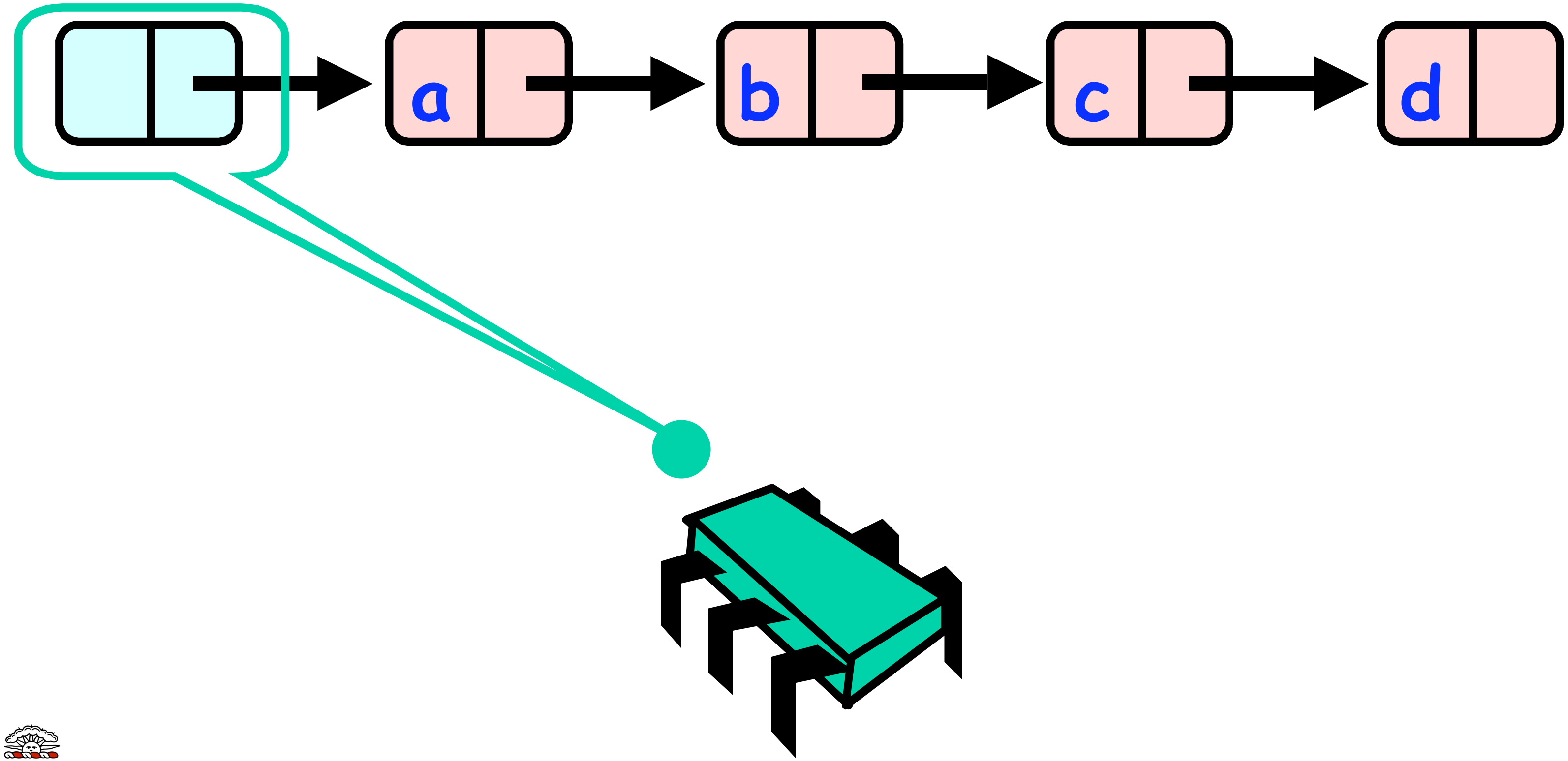
Business as Usual



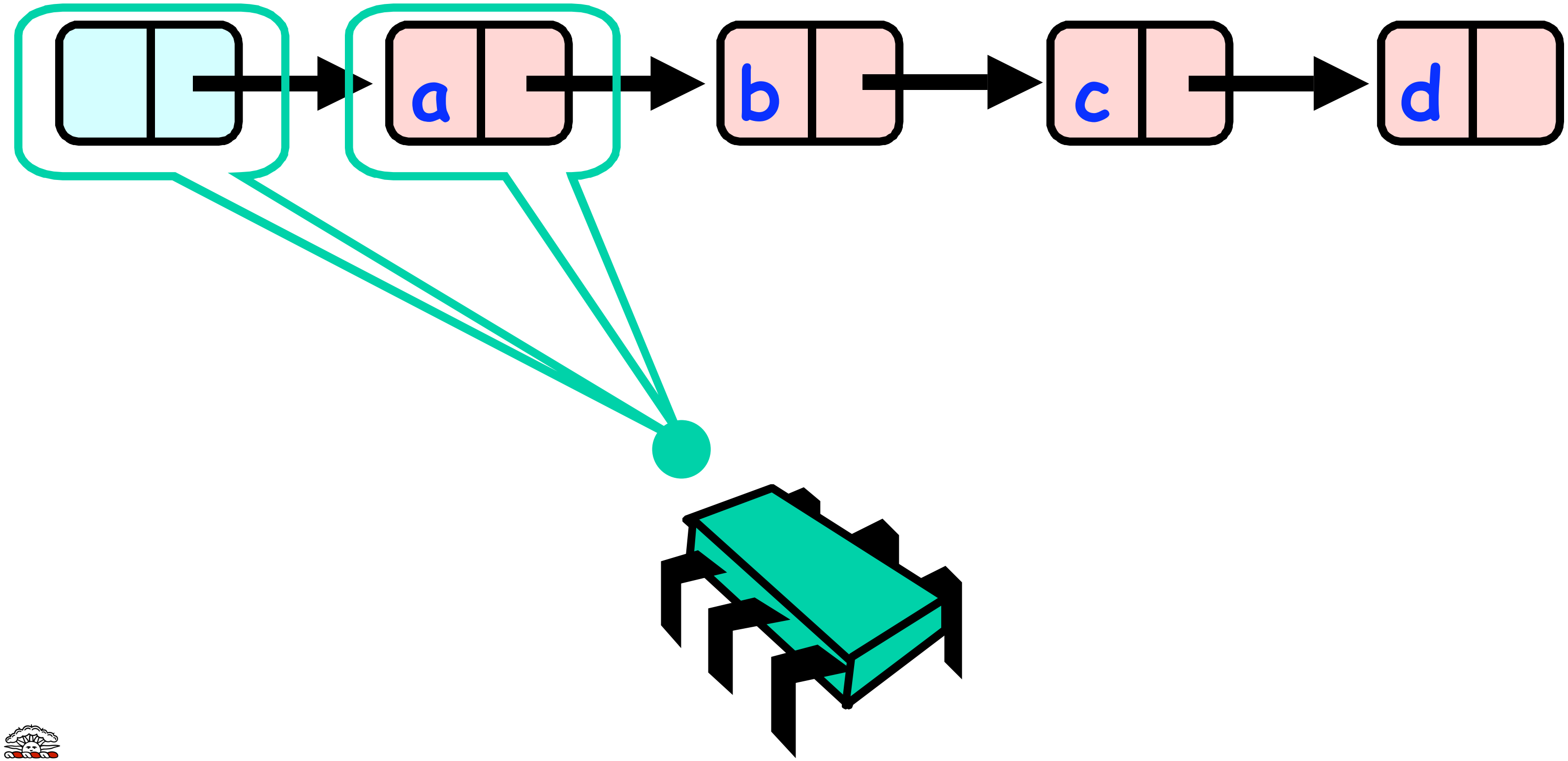
Business as Usual



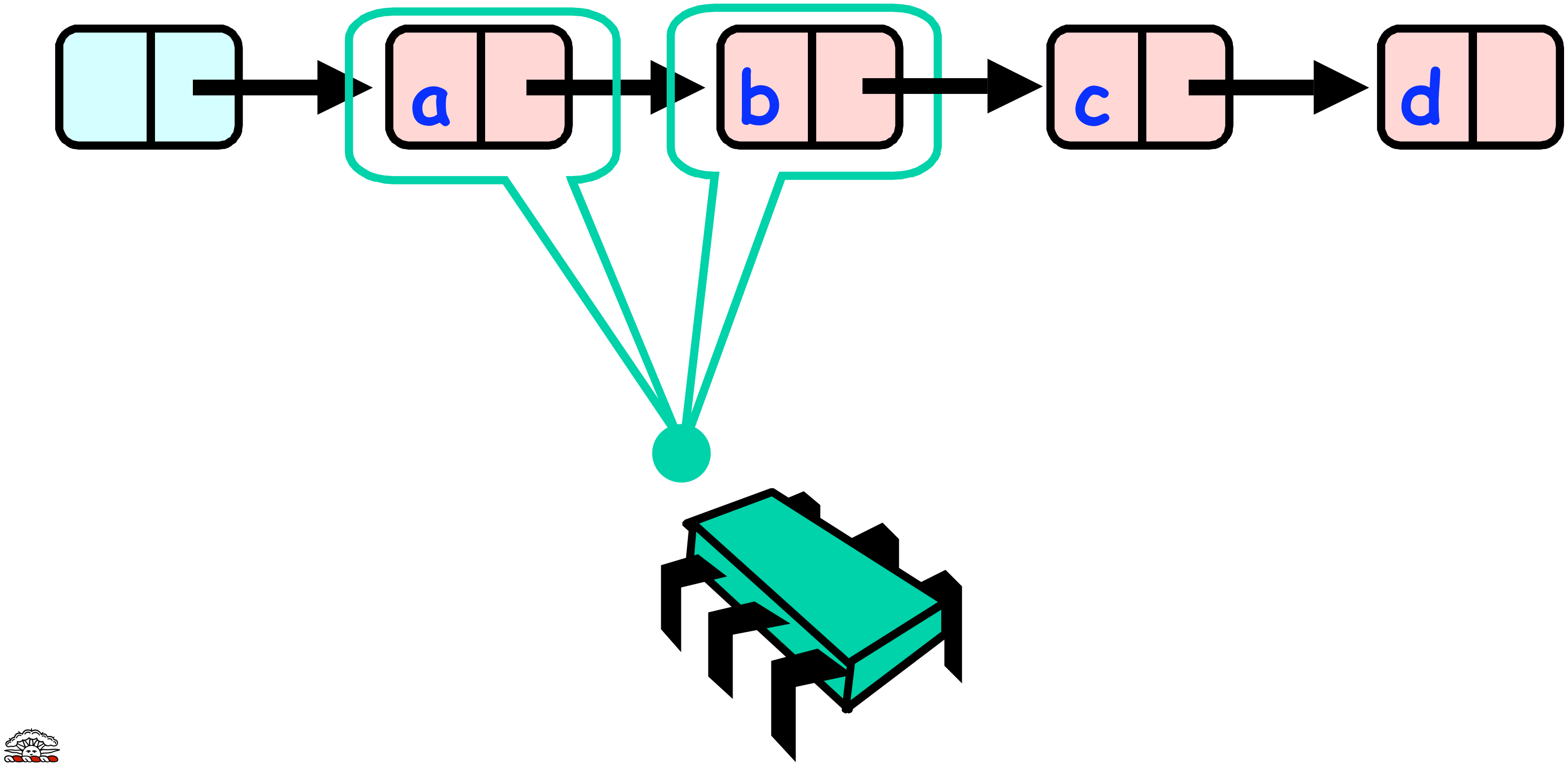
Interference



Interference

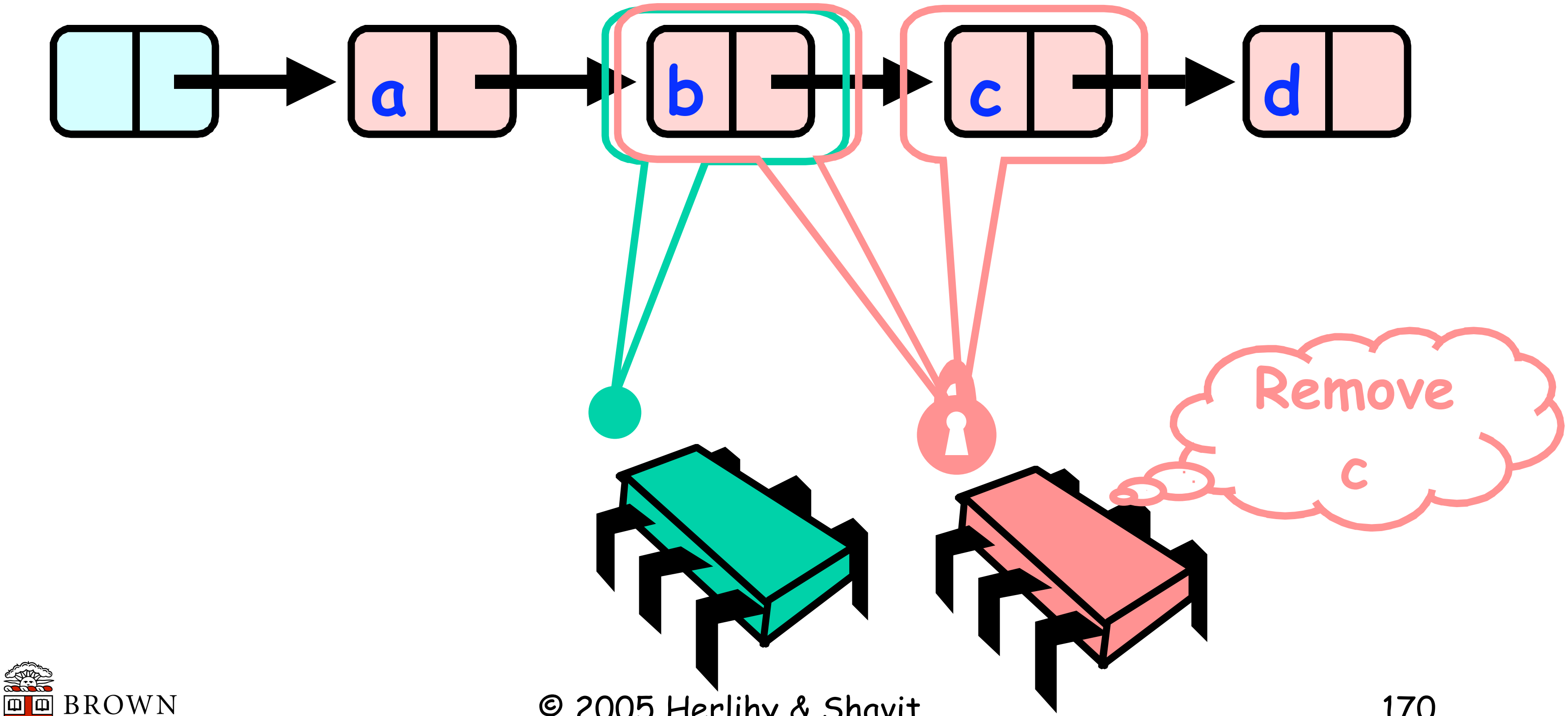


Interference

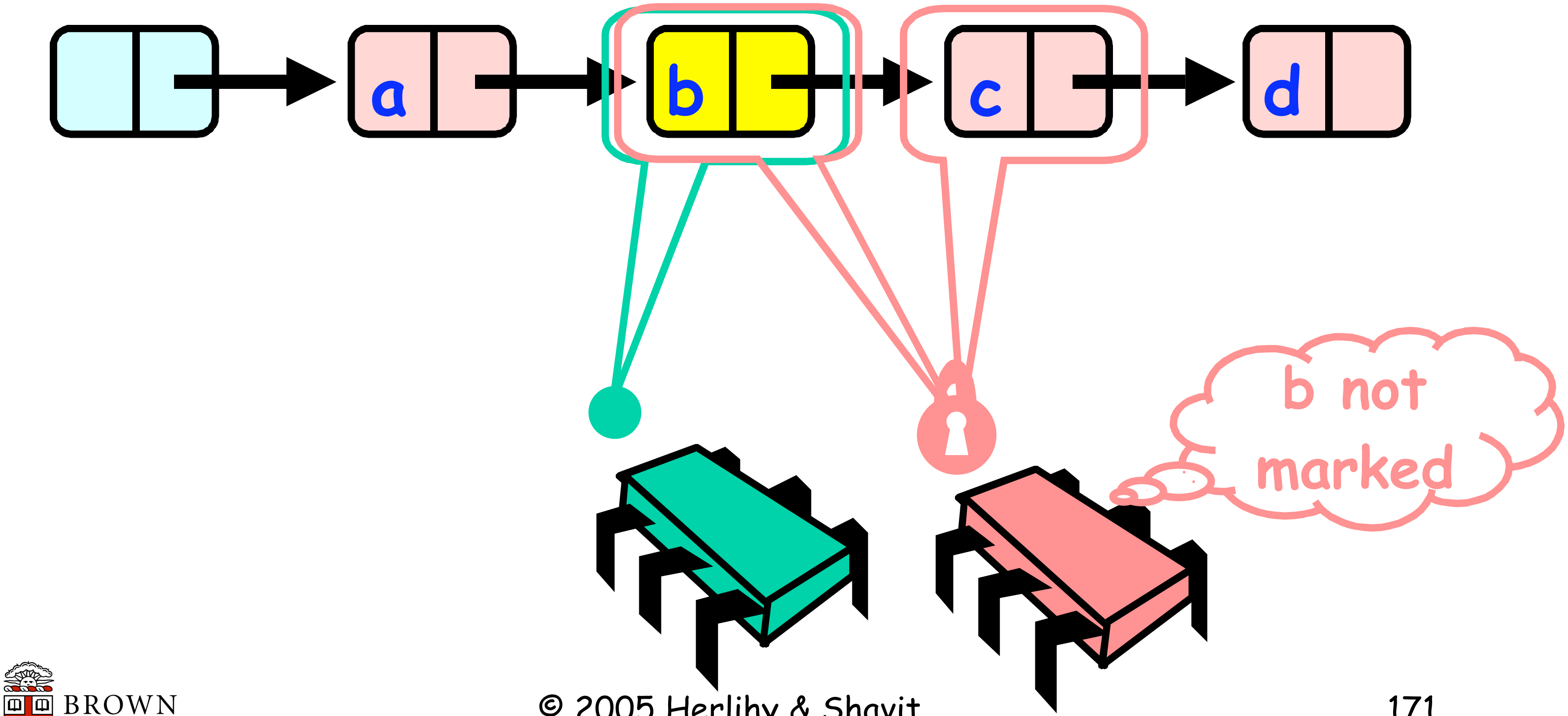


© 2005 Herlihy & Shavit

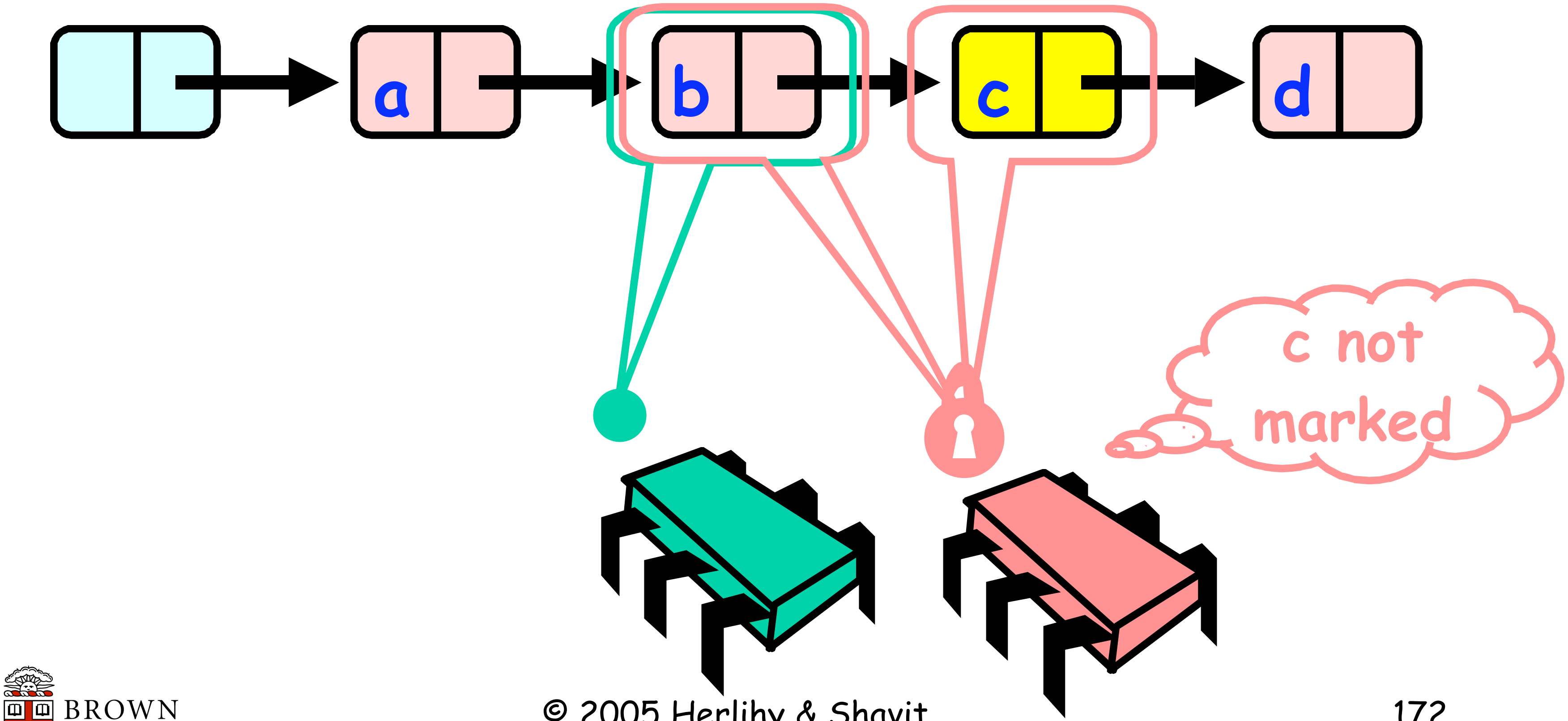
Interference



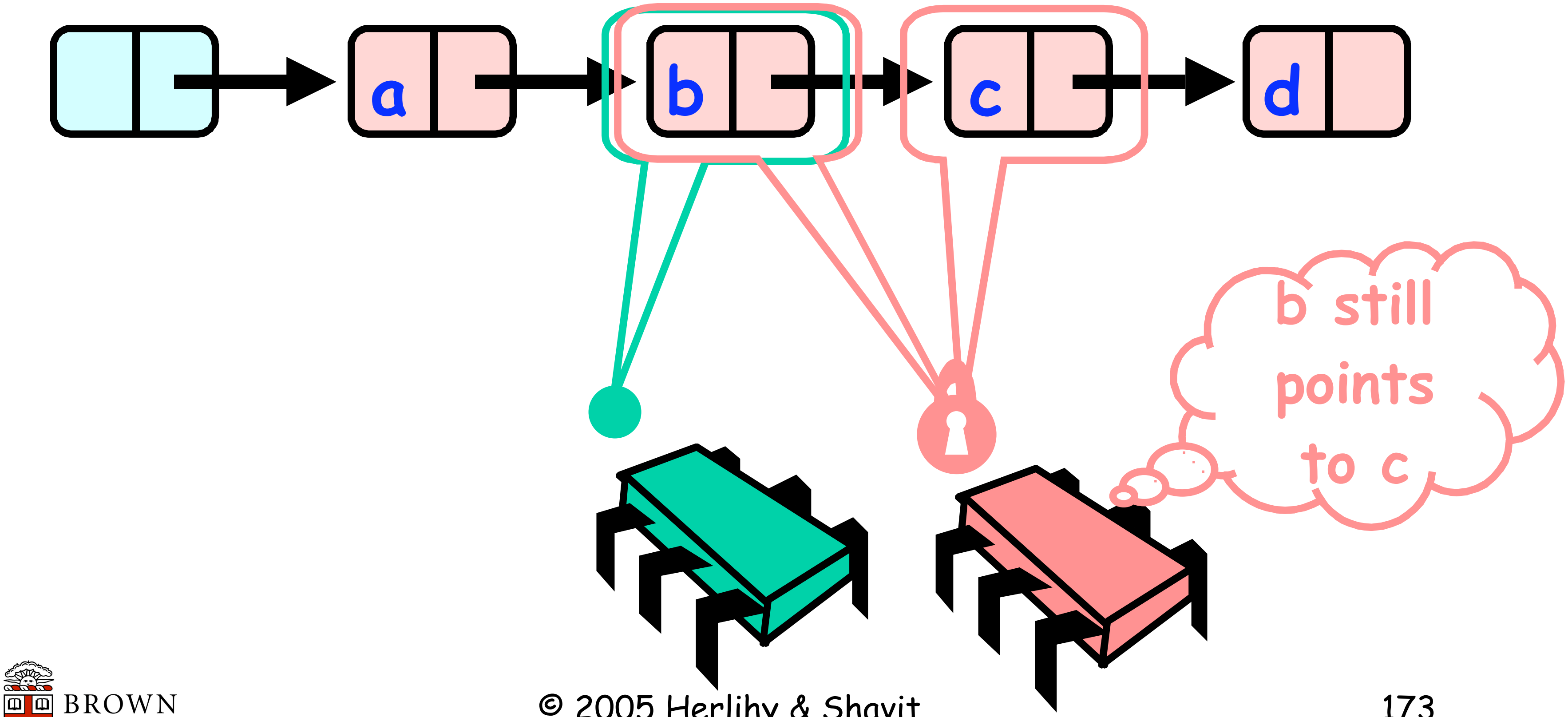
Validation



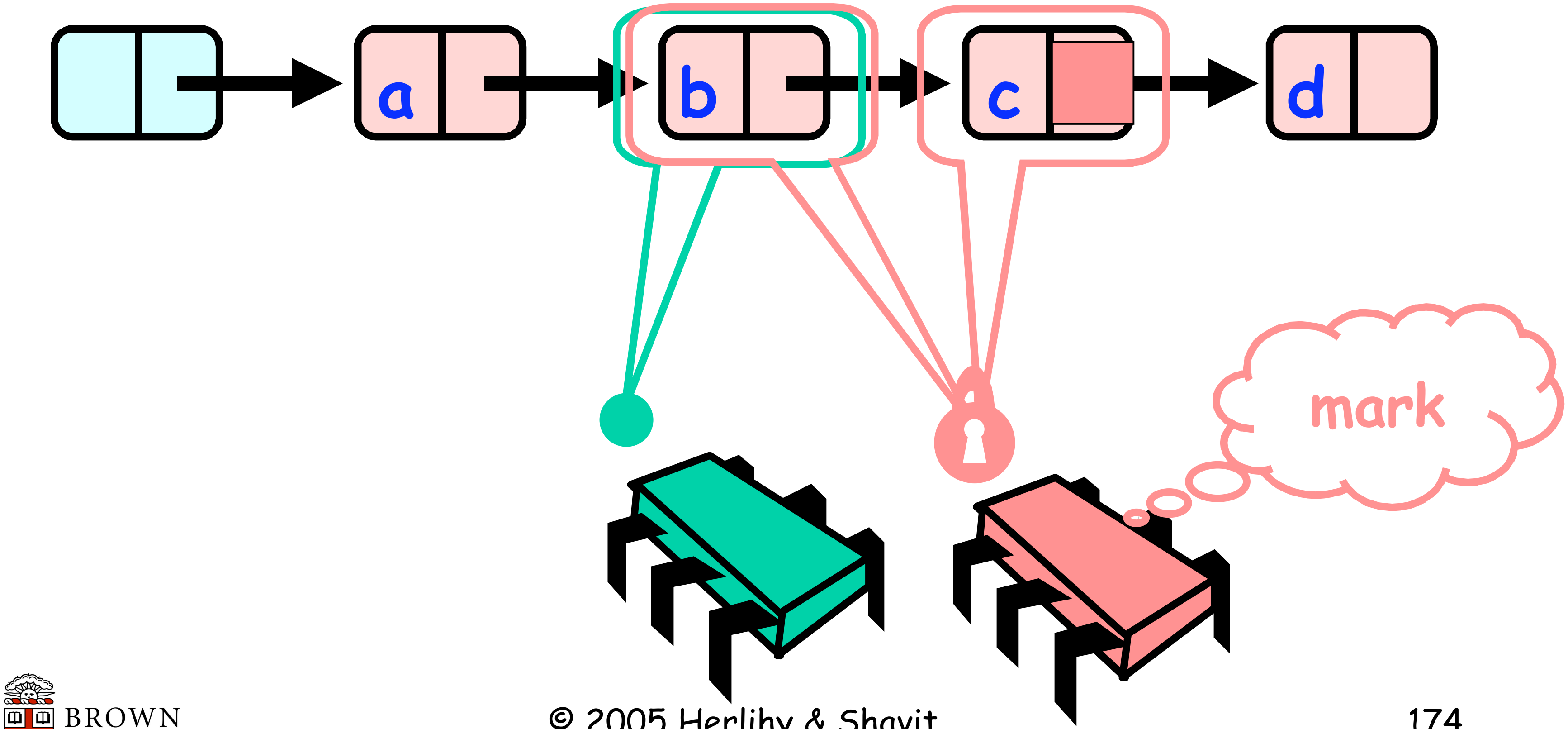
Interference



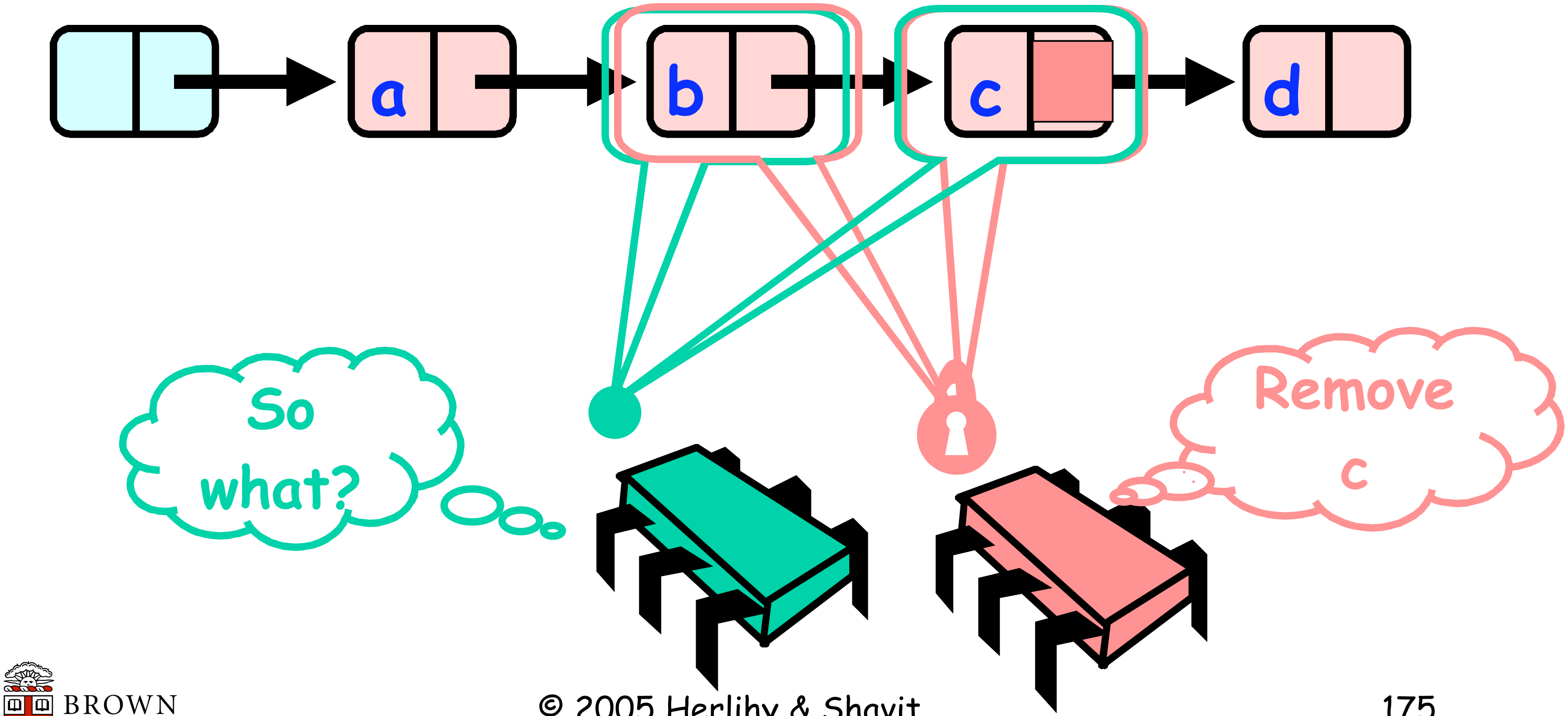
Interference



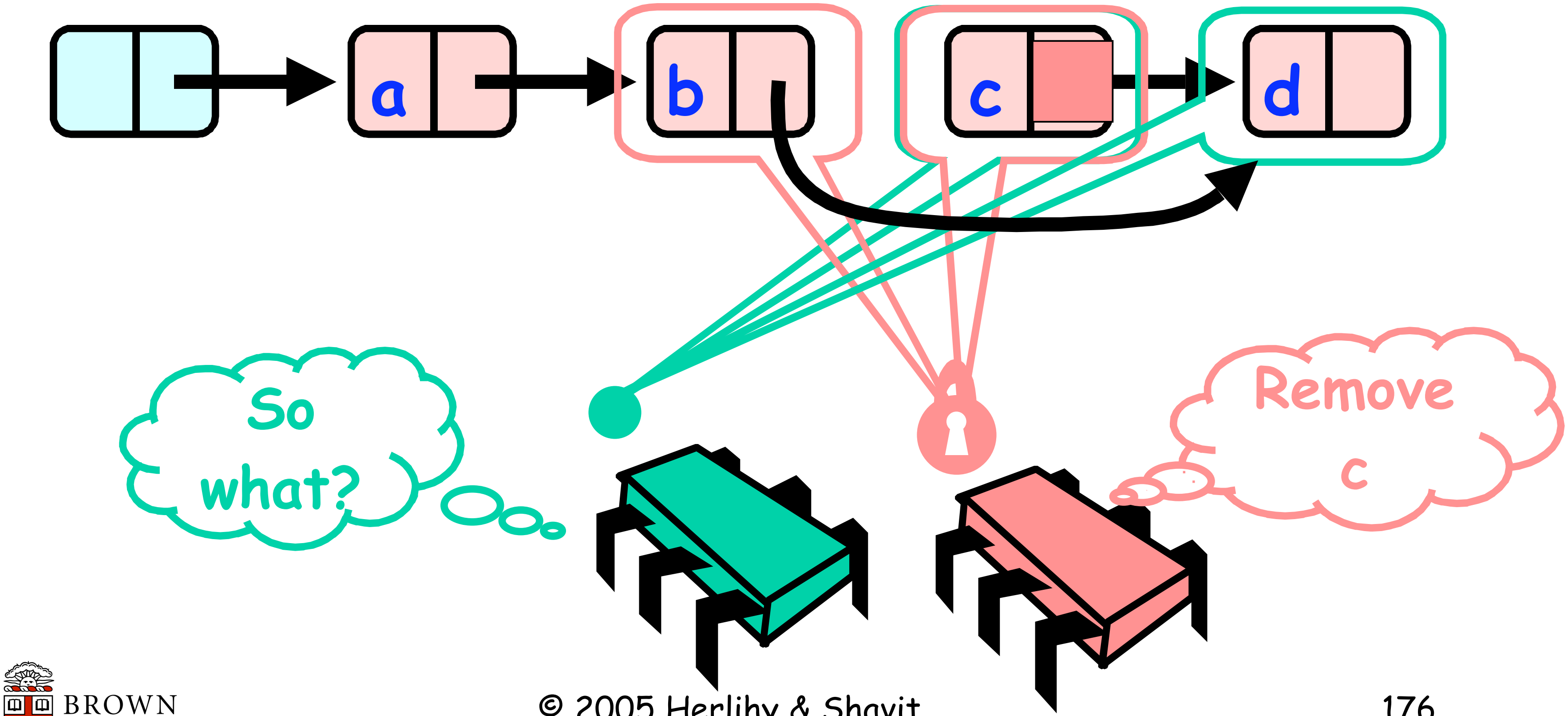
Logical Delete



Scan Through



Physical Deletion



New Abstraction Map

- S(head) =
  - { x | there exists entry a such that
    - head ⇒ a and
    - a.object = x and
    - a is unmarked
  - }

Modified Invariant

- If A's pred entry is unmarked, then it is reachable
- If **pred**$_A$ ≠ null and is not marked
  - Then **head** ⇒ **pred**$_A$ ⇒ **tail**

Invariant

- Holds initially
- Not modified by **add()** or **contains()**
- **Remove()**?
  - Marking doesn't violate invariant
    - No entry made unreachable
  - Physical remove doesn't violate
    - Entry made unreachable is already marked

Modified Invariant

- If **pred**$_A$ ≠ null and is not marked
  - Then **head** ⇒ **pred**$_A$ ⇒ **tail**
- Justifies why **contains()** doesn't need to lock
  - Unmarked reachable entry
  - Remains reachable
  - As long as it remains unmarked

Validation

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

List Validate Method

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

Predecessor not Logically removed

List Validate Method

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

Current not
Logically removed

List Validate Method

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

Predecessor still
Points to current

Remove

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

Remove

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

Validate as before

Remove

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

Key found

Remove

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

Logical remove

Remove

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

physical remove

Contains

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

Contains

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

Start at the head

Contains

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

Search key range

Contains

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

Traverse without locking
(nodes may have been removed)

Contains

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

Present and undeleted?

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

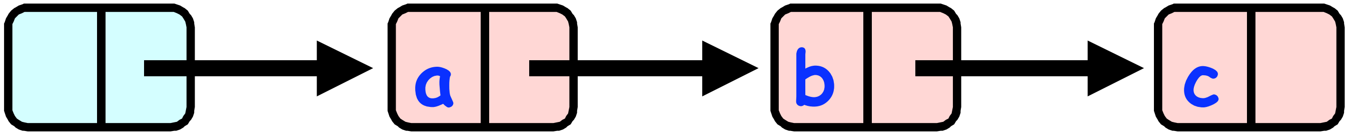
Lock-Free ≠Wait-Free

- 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 …

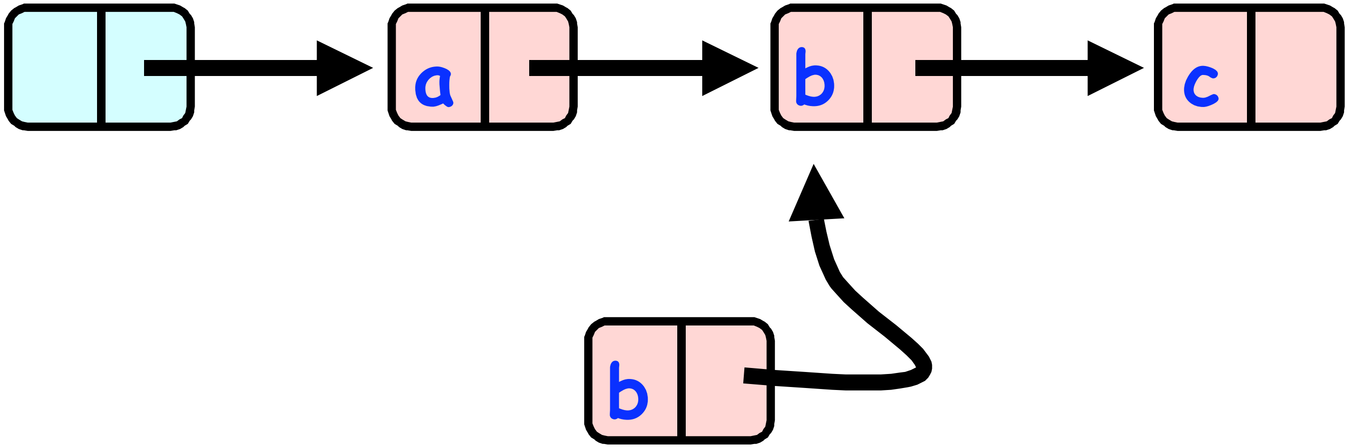
Lock-Free Lists

- Next logical step
- Eliminate locking entirely
- Use only compareAndSet()
- What could go wrong?

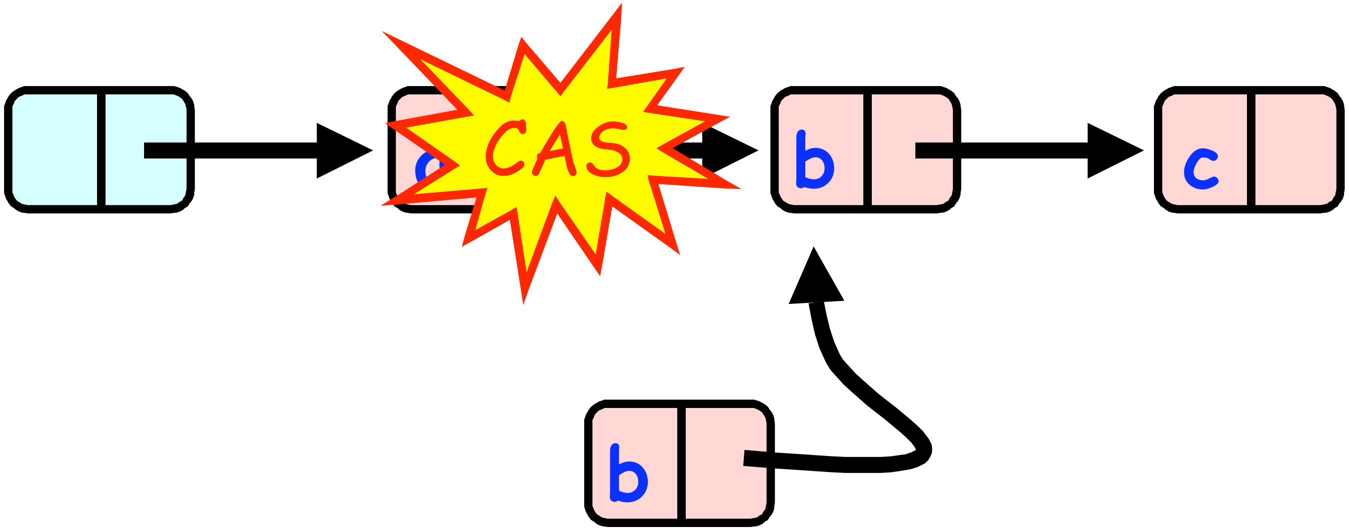
Adding an Entry



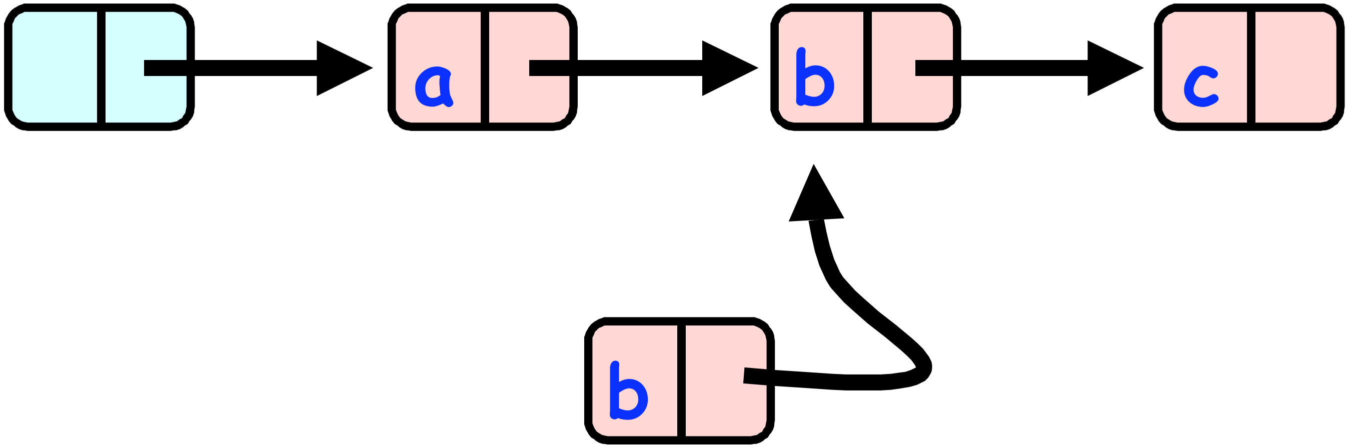
Adding an Entry



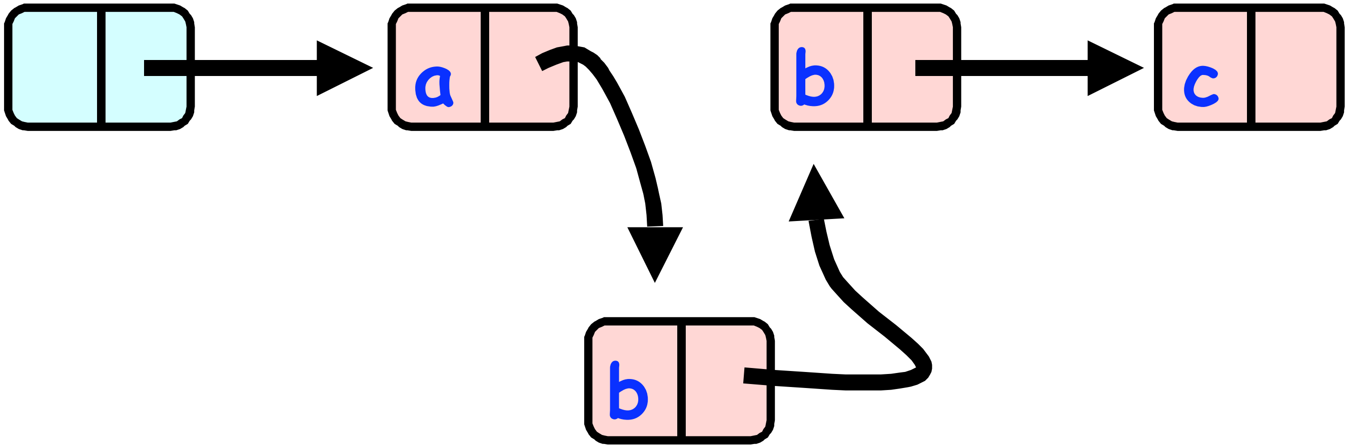
Adding an Entry



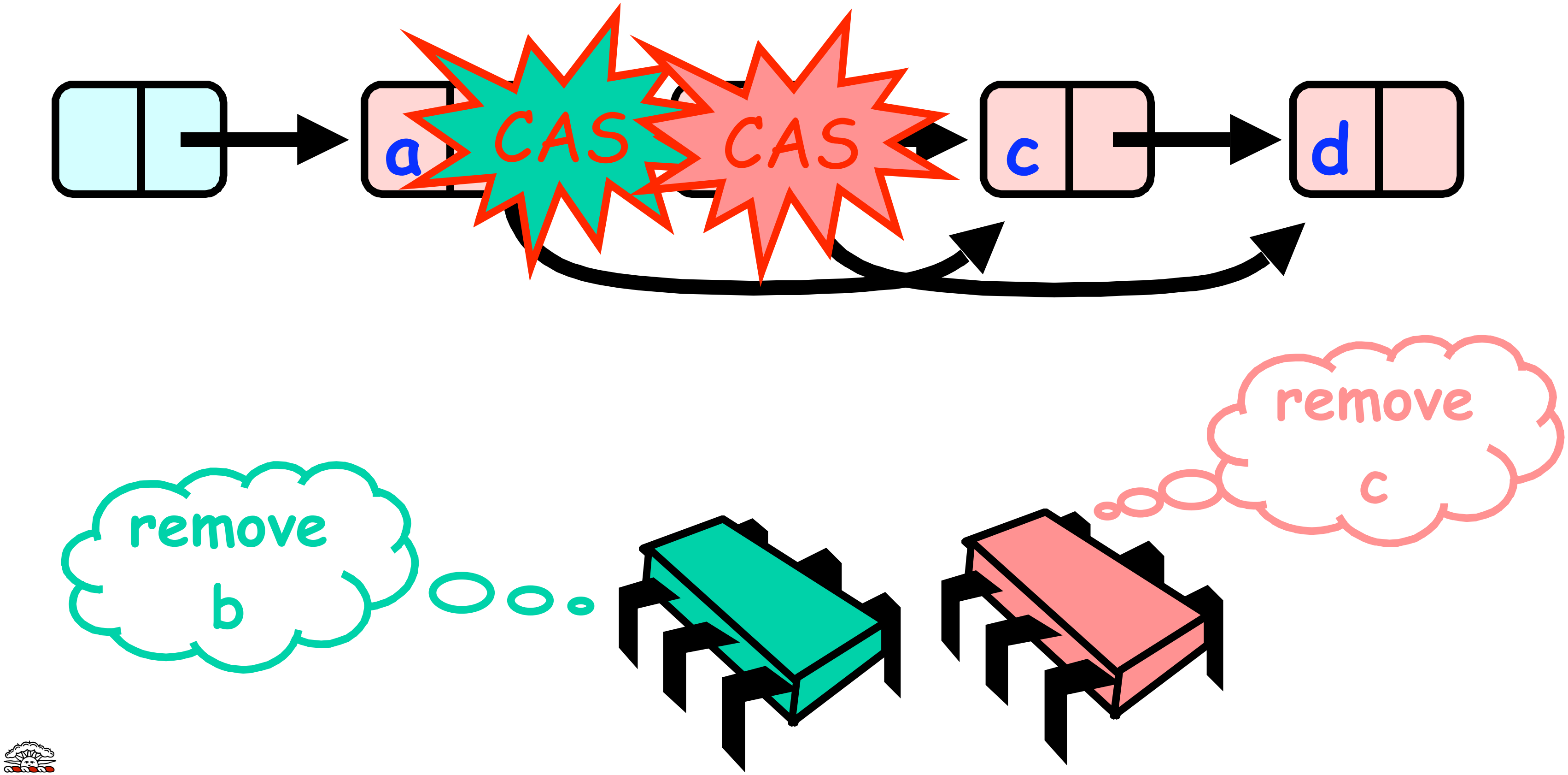
Adding an Entry



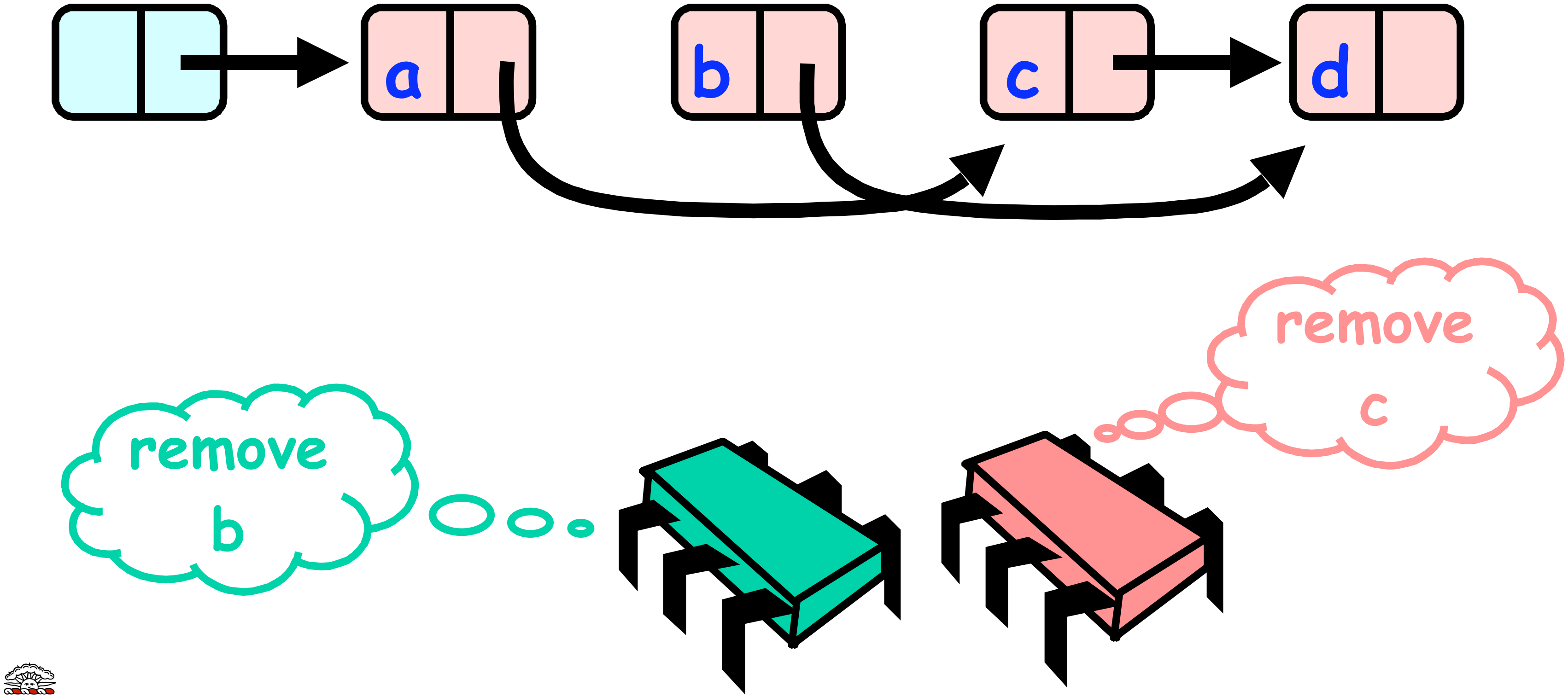
Adding an Entry



Removing an Entry



Look Familiar?



Problem

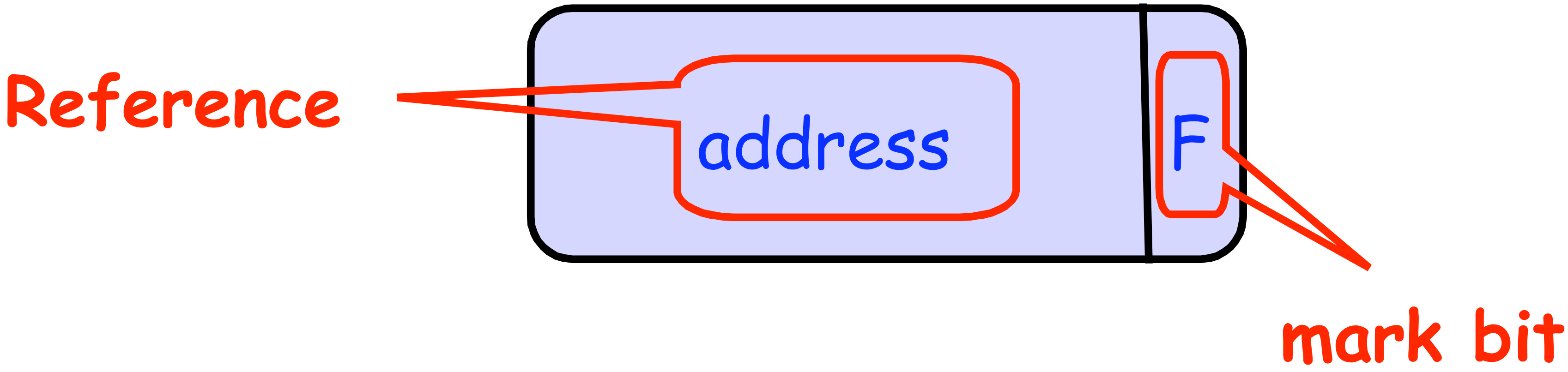
- Method updates entry's next field
- After entry has been removed

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[]);
```

Extracting Reference & Mark

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

Returns reference

Returns mark at array index 0!

Extracting Reference Only

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

Extracting Reference Only

```
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);
```

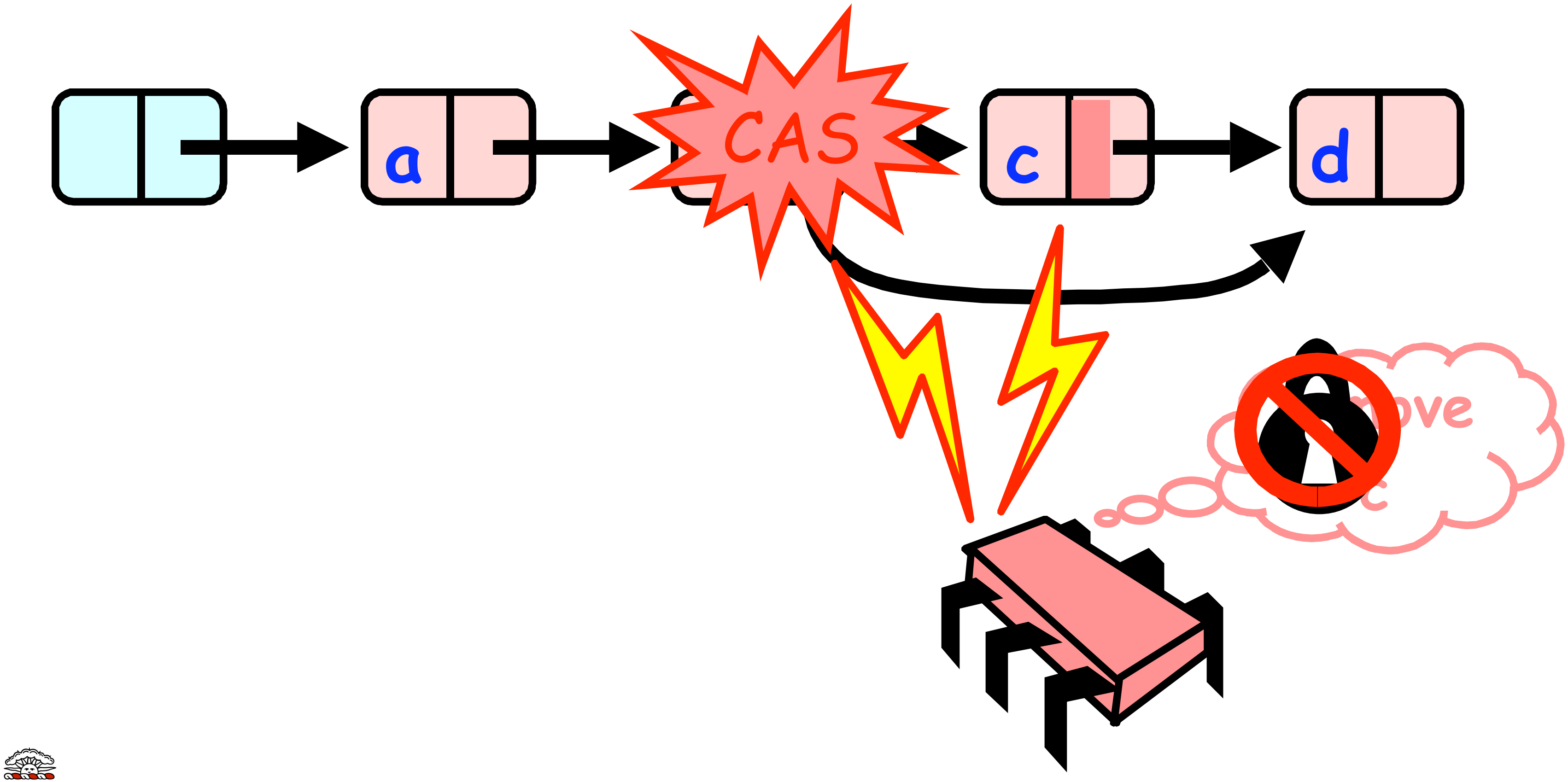
If this is the current reference …

Changing State

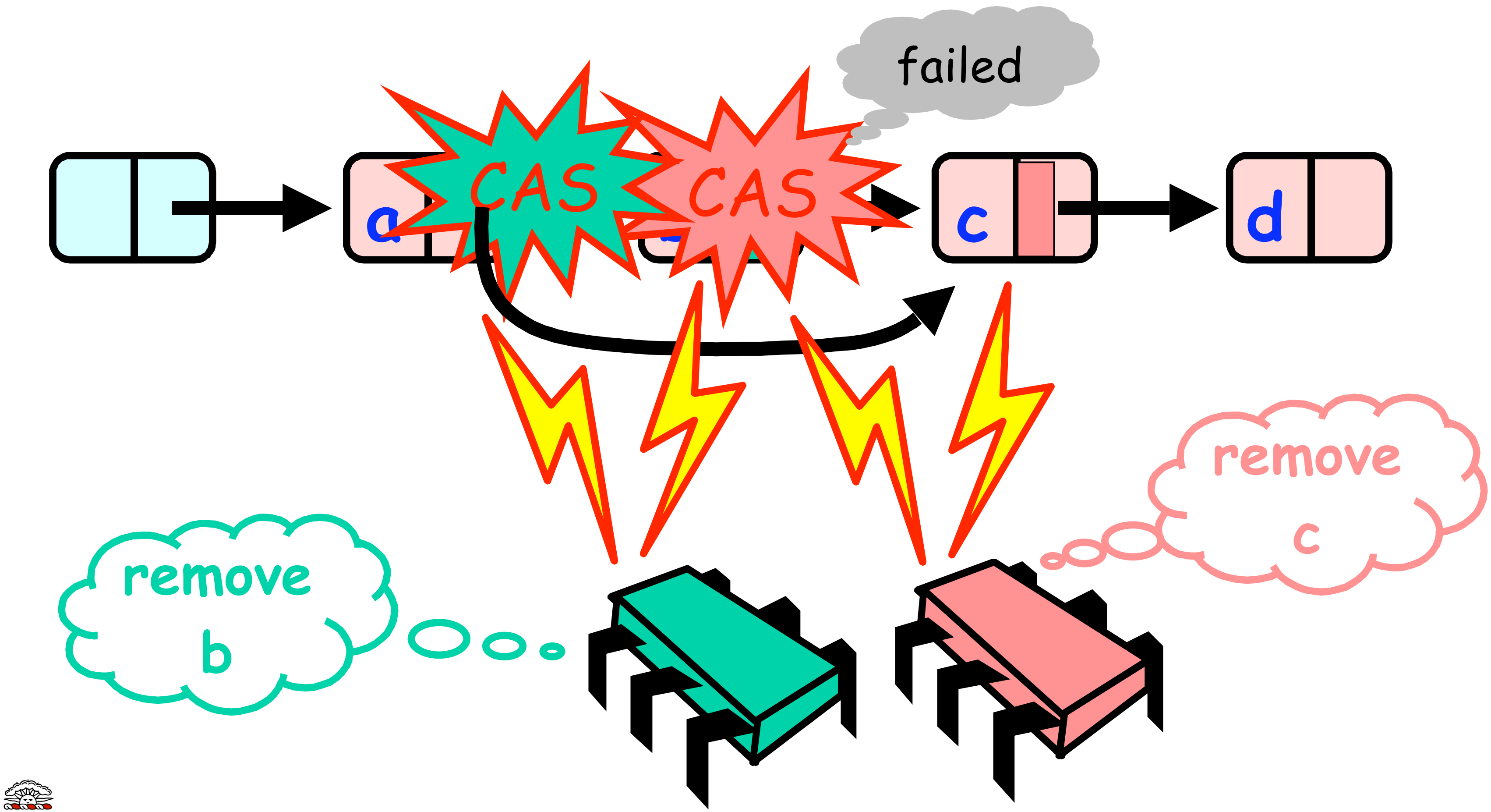
```
public boolean attemptMark(
  Object expectedRef,
  boolean updateMark);
```

.. then change to this new mark.

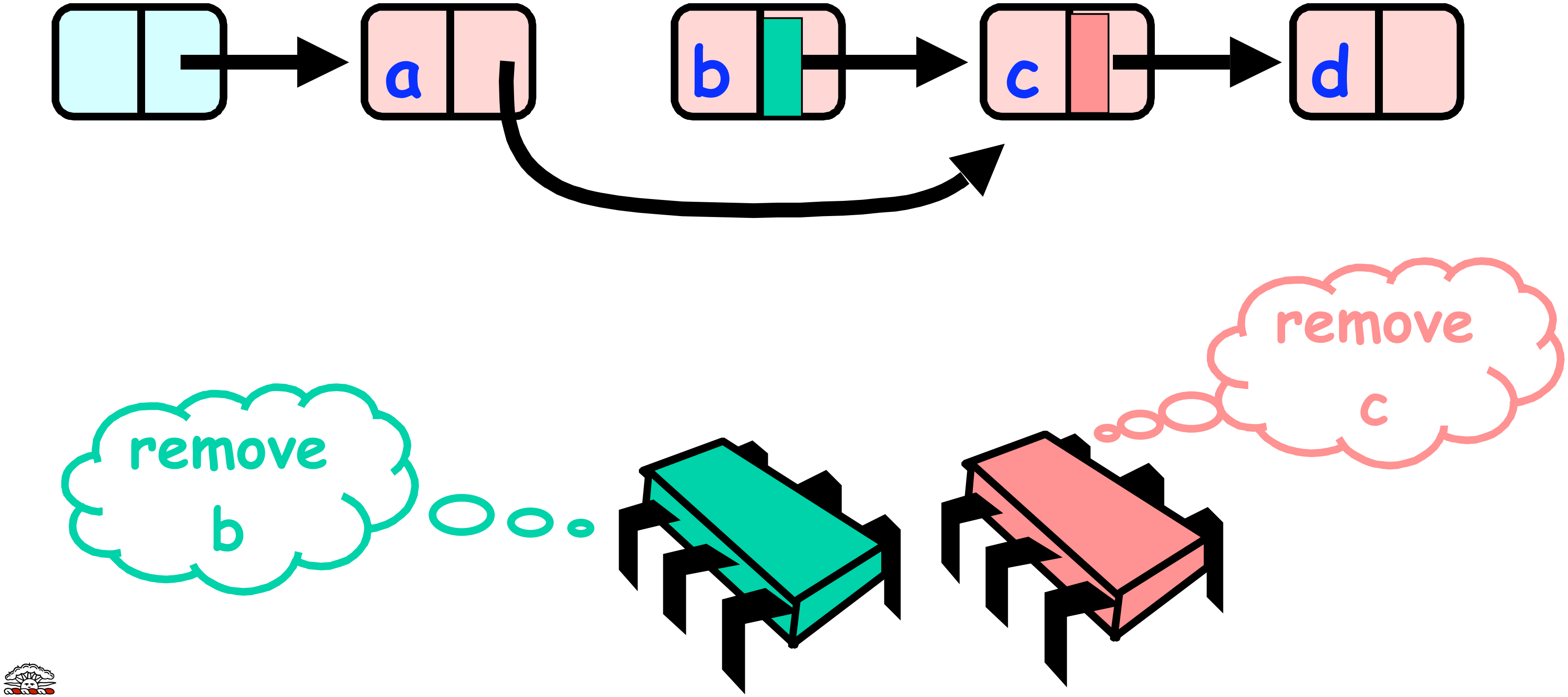
Removing an Entry



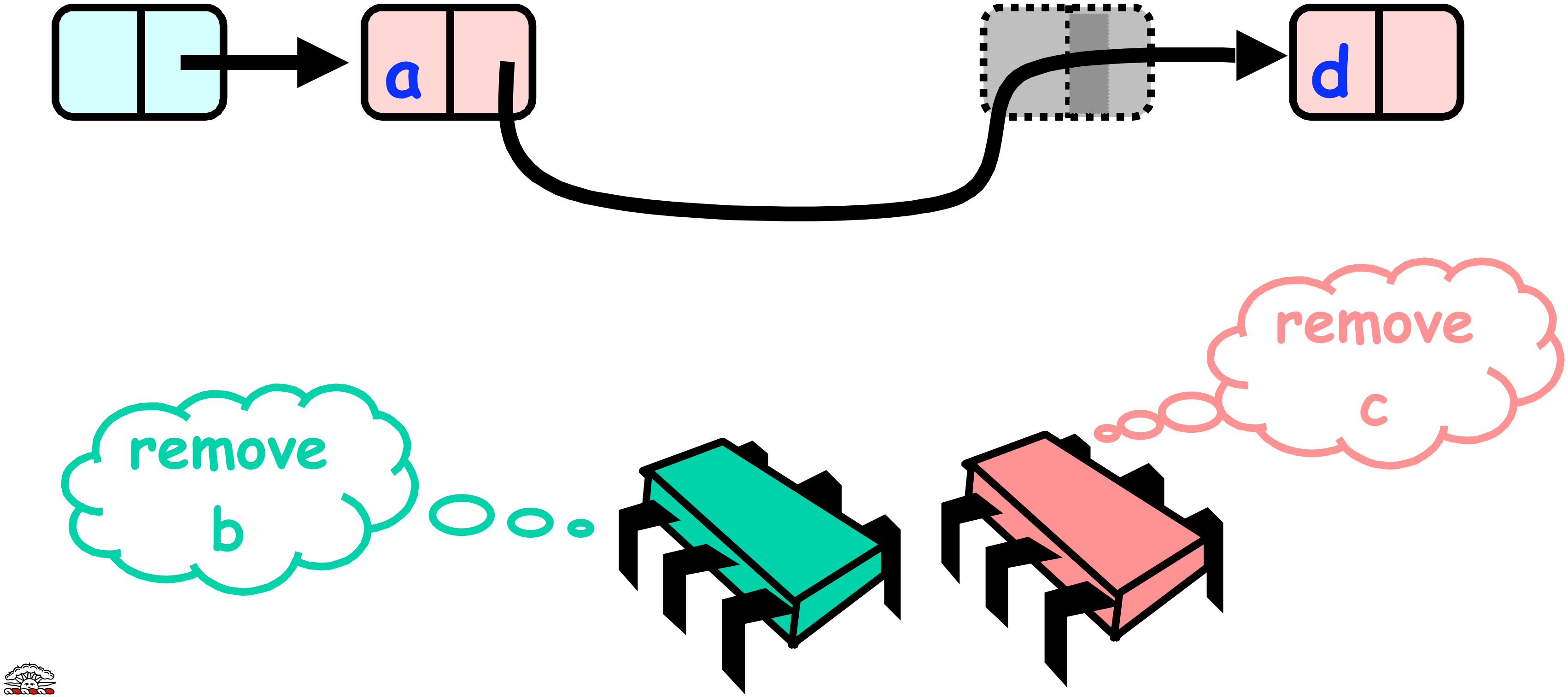
Removing an Entry



Removing an Entry



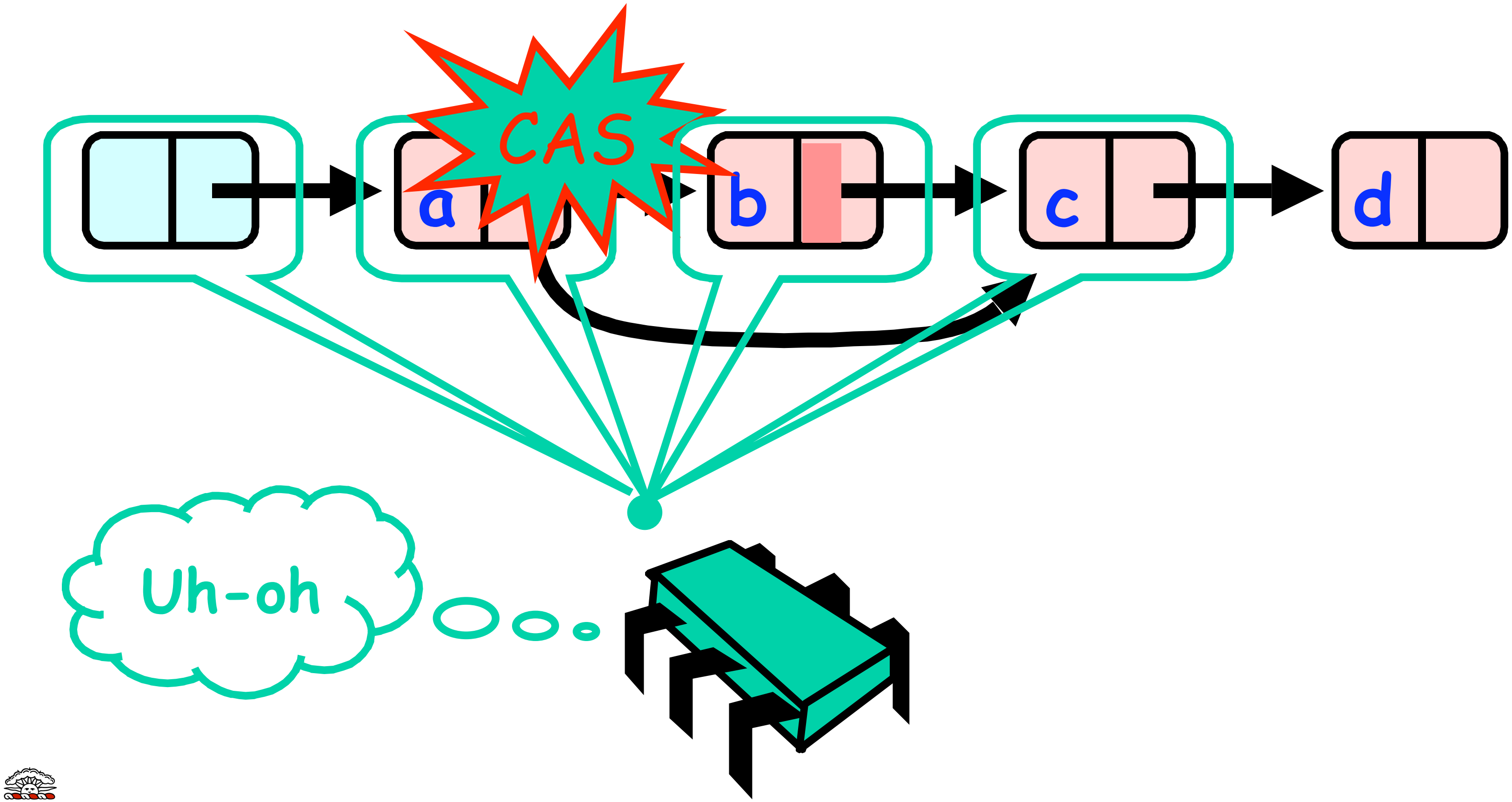
Removing an Entry



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



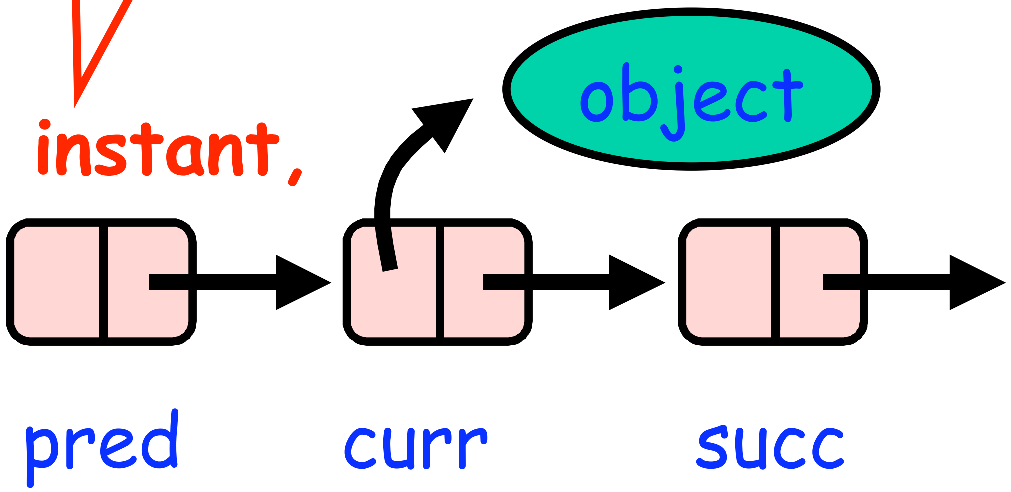
The Find Method

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

The Find Method

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

At some instant,



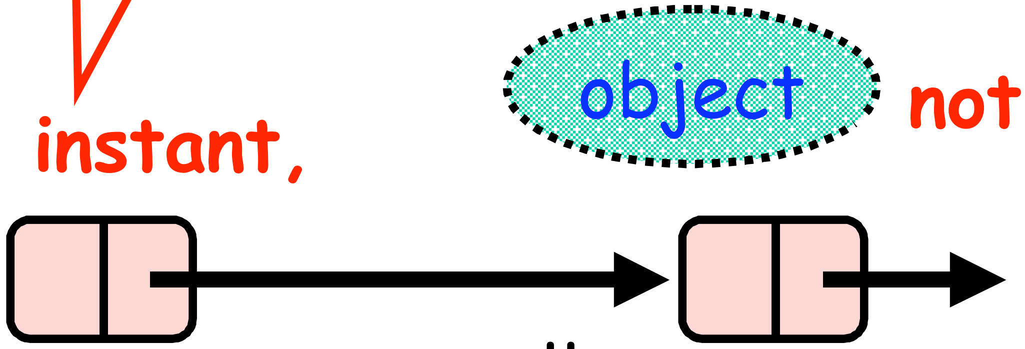
or …

The Find Method

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

At some instant,

object not in list



curr= null

pred

succ

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 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

```
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;
}}
```

Try to mark entry as deleted

Remove

If it doesn't work, just retry

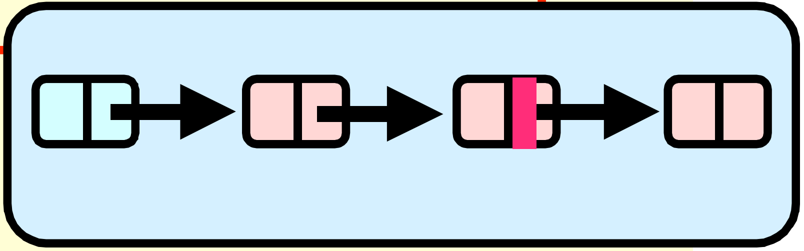
```
while (true) {
 pred,curr,succ = find(object);
 if (curr == null)
  return false;
 if (!curr.next.attemptMark(succ,
                         true))
  continue;
 pred.next.compareAndSet(curr, succ,
                        false,false);
 return true;
}}
```

BROWN

Remove

If it works, our job is (essentially) done

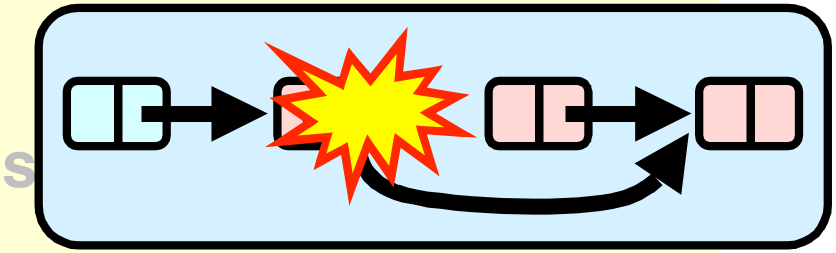
```
while (true) {
 pred,curr,succ = find(object);
 if (curr == null)
  return false;
 if (!curr.next.attemptMark(succ,
                        true))
  continue;
 pred.next.compareAndSet(
                       false,false);
 return true;
}}
```



Remove

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

```
pred,curr,succ = find(object,set);
if (curr == null)
 return false;
if (!curr.next.attemptMark(s
                        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

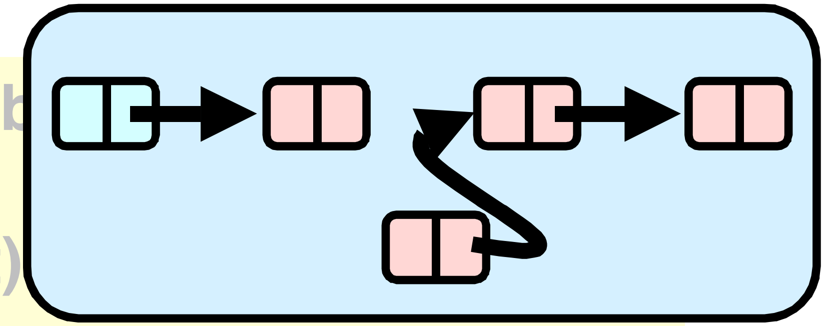
```
public boolean add(Object object) {
 while (true) {
  pred,curr,succ= find(object);
  if (curr != null)
   return false;
  Entry entry = new Entry(object);
  entry.next = new AMR(succ,false);
  if (pred.next.CAS(succ, entry,
                    false, false))
```

Object already there.

```
 }}
```

Add

```
public boolean add(Object ob
 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))
```



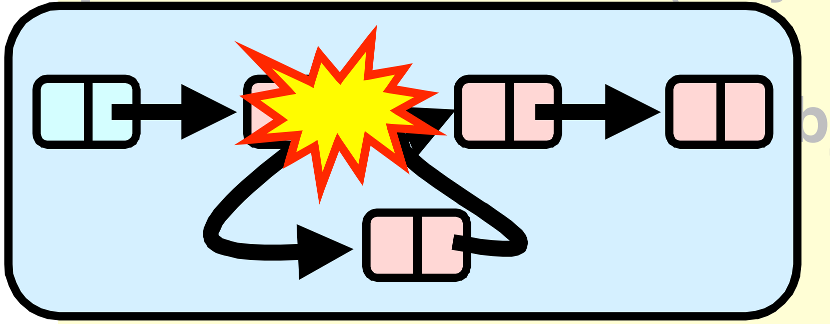
create new entry

Add

```
public boolean add(Object object) {

                                  bject);

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



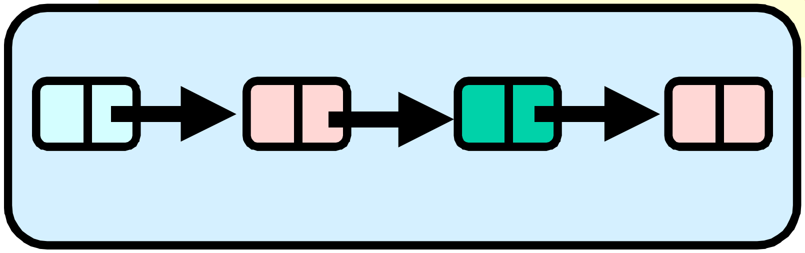
Install new entry

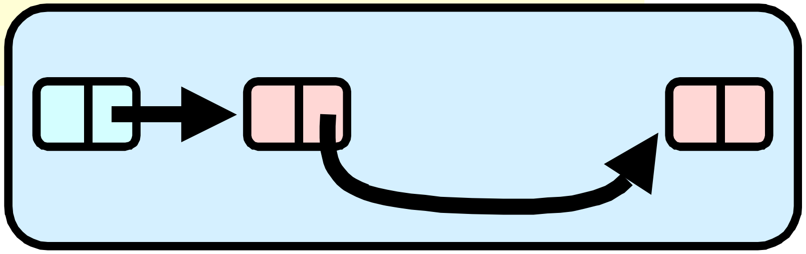
Contains

```
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

```
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) {
   …
}}}
```

The entries we seek

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

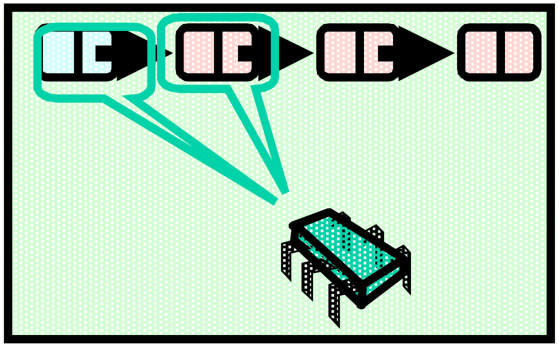
```
private Entry,Entry,Entry
  find(Object object) {
 Entry pred, curr, succ;
 boolean[] pmark = new boolean[1];
 boolean[] cmark = new boolean[1];
 int key = object.hashCode();
 tryAgain: while (true) {
   ...
}}}
```

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



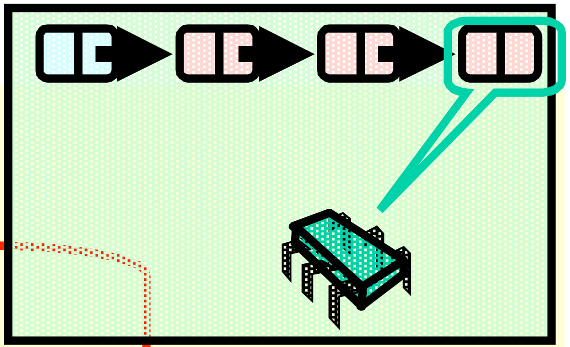
Find

```
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;
}}}
```

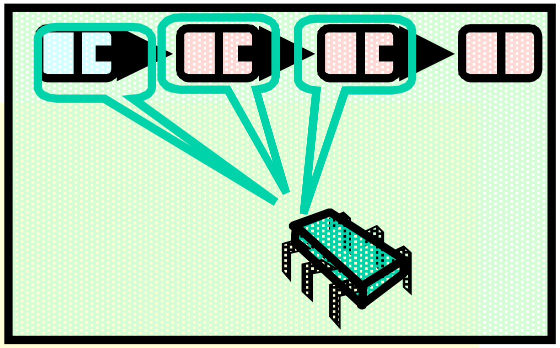
Get ref to successor and current deleted bit

Find

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

Panic if predecessor's next field changed

Find

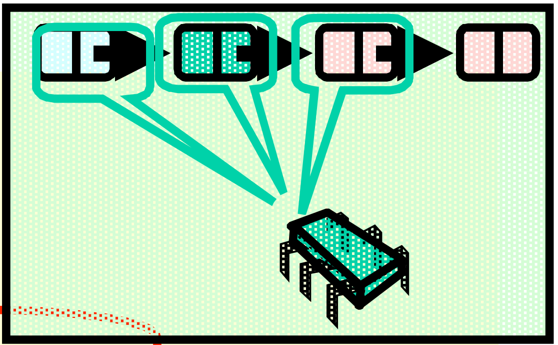


```
while (true) {
 ...
if (!cmark[0]) {
  if (curr.object == object)
   return pred, curr, succ;
  else if (ckey <= key) {
   pred = curr;
 } else
  return prev, null, curr;
} else {
 ...
}}}
```

If current node is not deleted

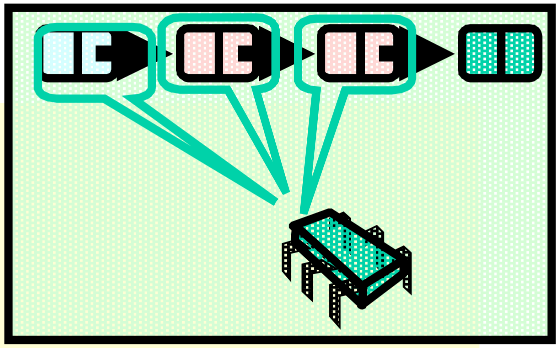
Find

```
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



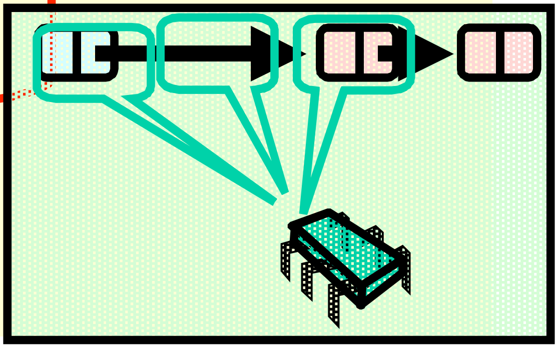
```
while (true) {
 ...
if (!cmark[0]) {
  if (curr.object == object)
   return pred, curr, succ;
  else if (ckey <= key) {
   pred = curr;
 } else
  return prev, null, curr;
} else {
 ...
}}}
```

Keep looking

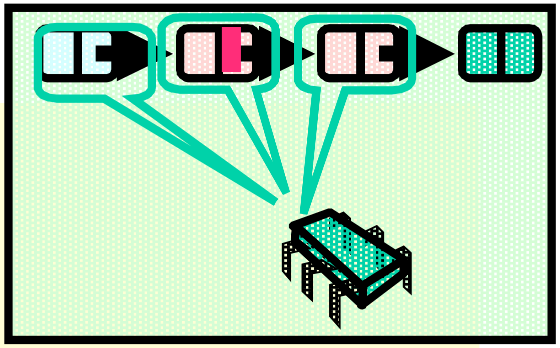
Find

```
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



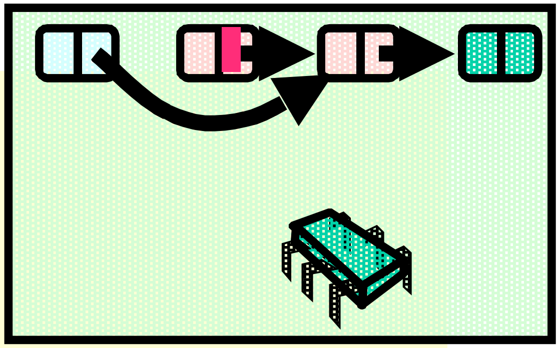
Find



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

Current entry is logically deleted

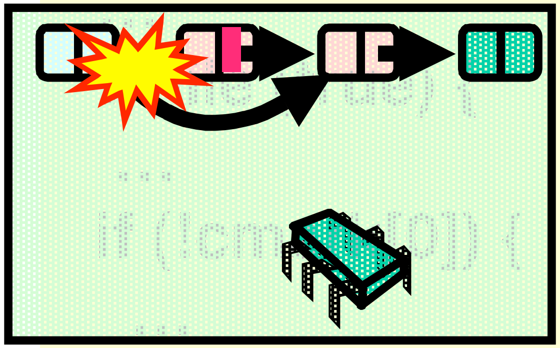
Find



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

Try to redirect predecessor's next reference

Find



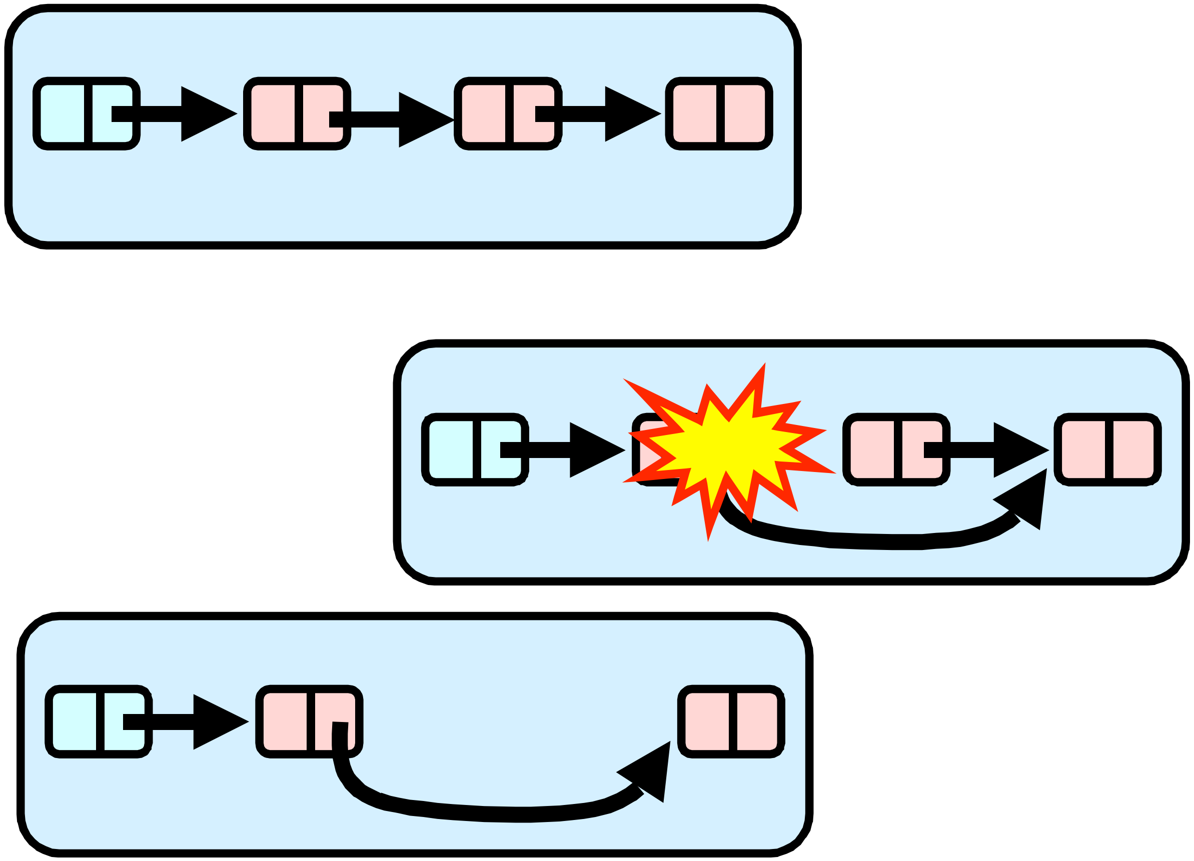
On success, keep going, on failure, start over

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

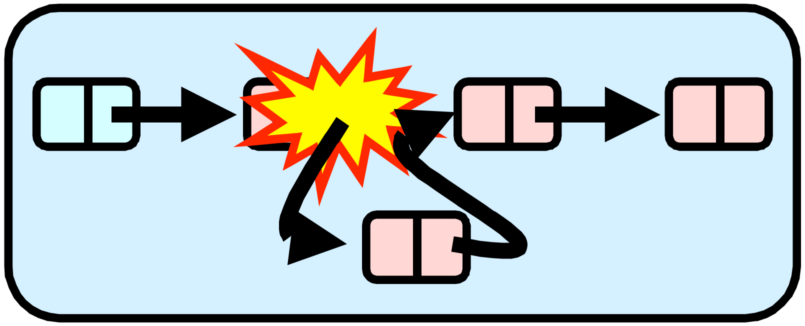
Summary

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

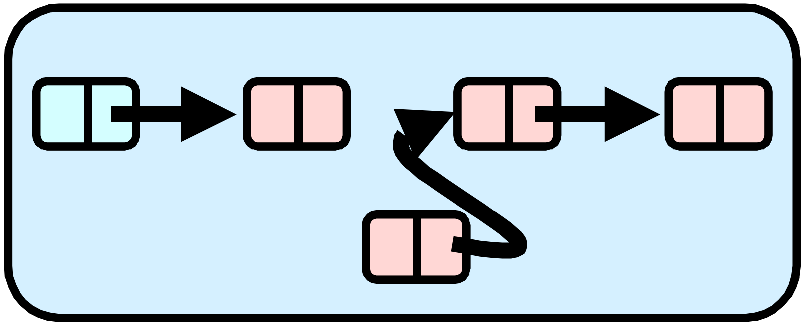
Scratch



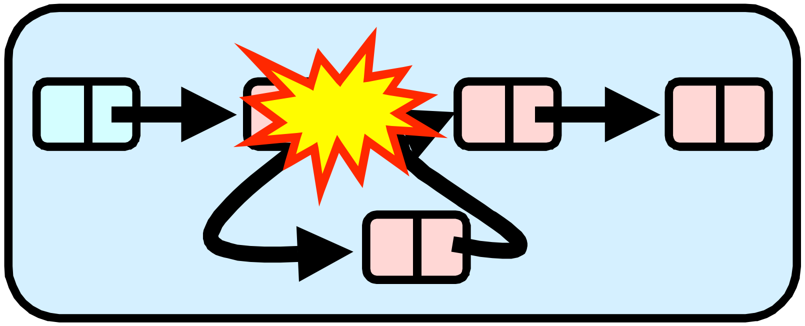
Scratch



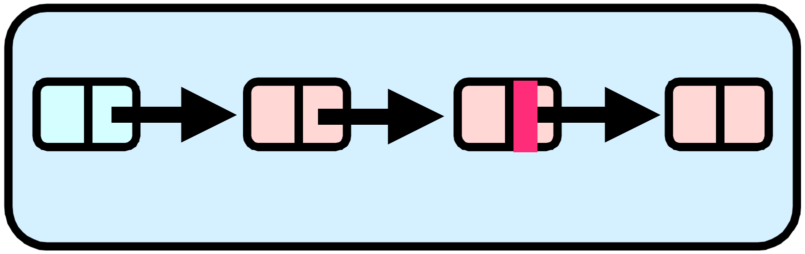
Scratch



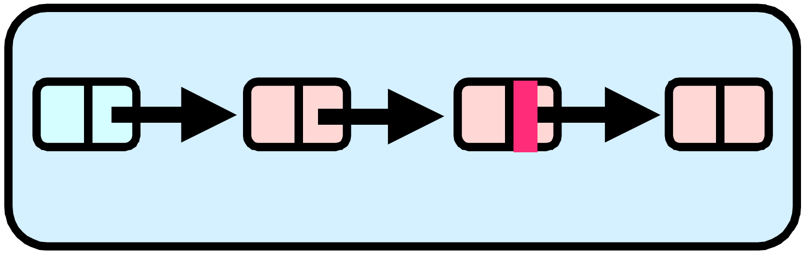
Scratch



Scratch



Scratch



Removing an Entry