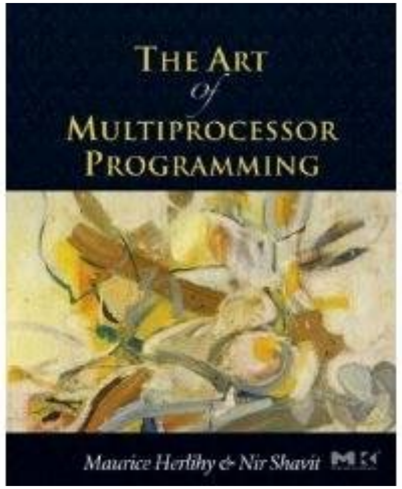
Linked Lists: Locking, Lock-Free, and Beyond …



Hyungsoo Jung

First: Fine-Grained Synchronization

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

Second:
Optimistic Synchronization

- Search without locking …
- If you find it, lock and check …
  - OK: we are done
  - Oops: start over
- Evaluation
  - Usually cheaper than locking, but
  - Mistakes are expensive

Third:
Lazy Synchronization

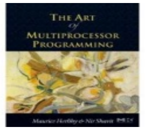
- Postpone hard work
- Removing components is tricky
  - Logical removal
    - Mark component to be deleted
  - Physical removal
    - Do what needs to be done

Fourth: Lock-Free Synchronization

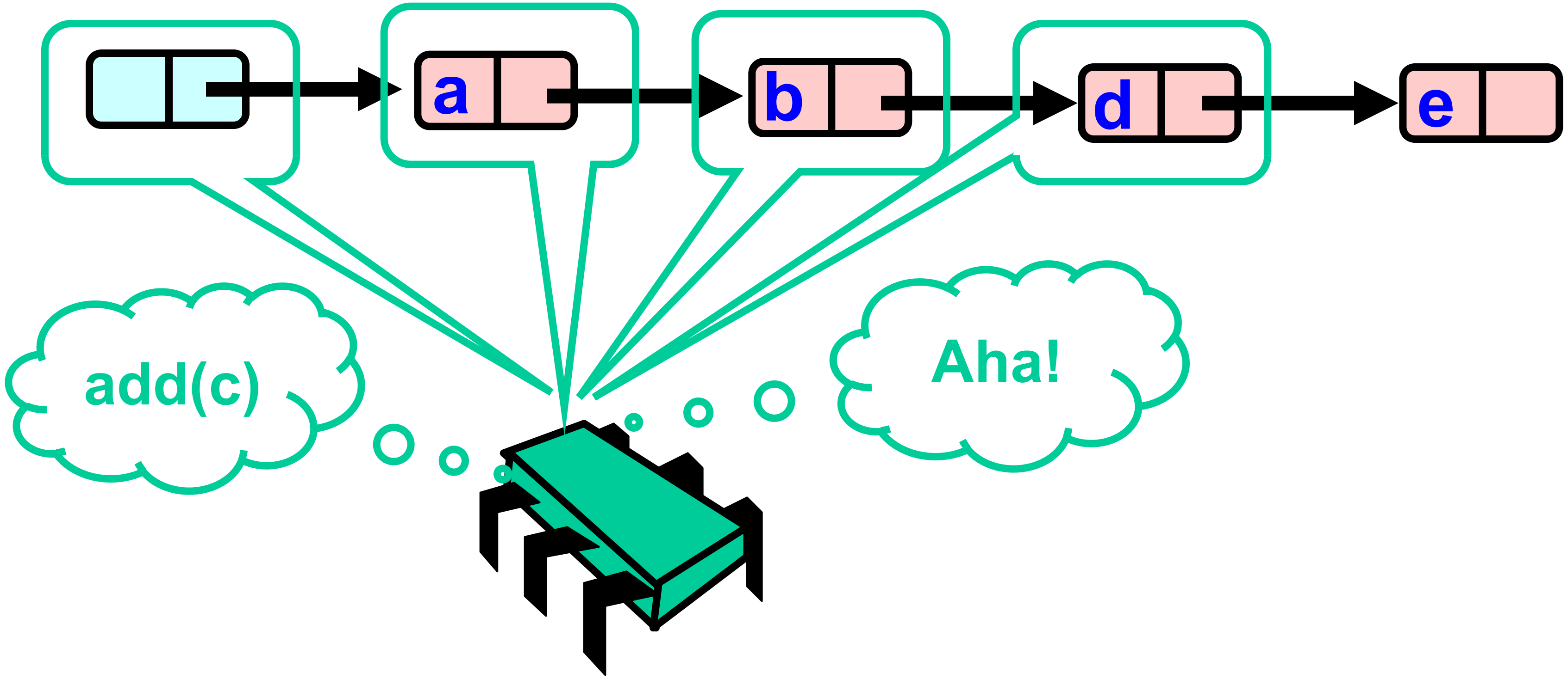
- Don’t use locks at all
  - Use compareAndSet() & relatives …
- Advantages
  - No Scheduler Assumptions/Support
- Disadvantages
  - Complex
  - Sometimes high overhead

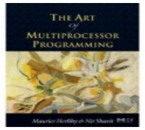
Optimistic Synchronization

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

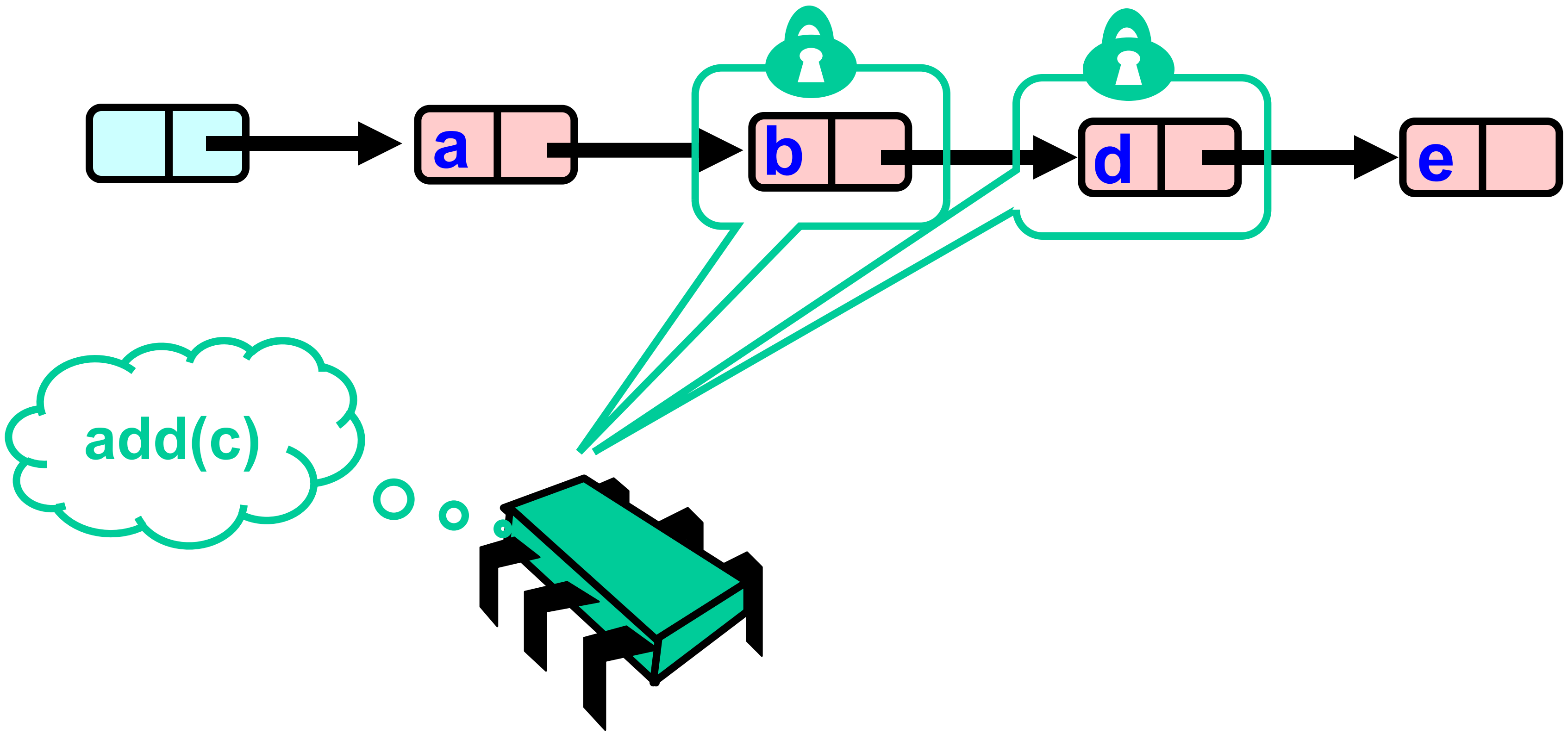


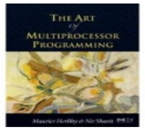
Optimistic: Traverse without Locking



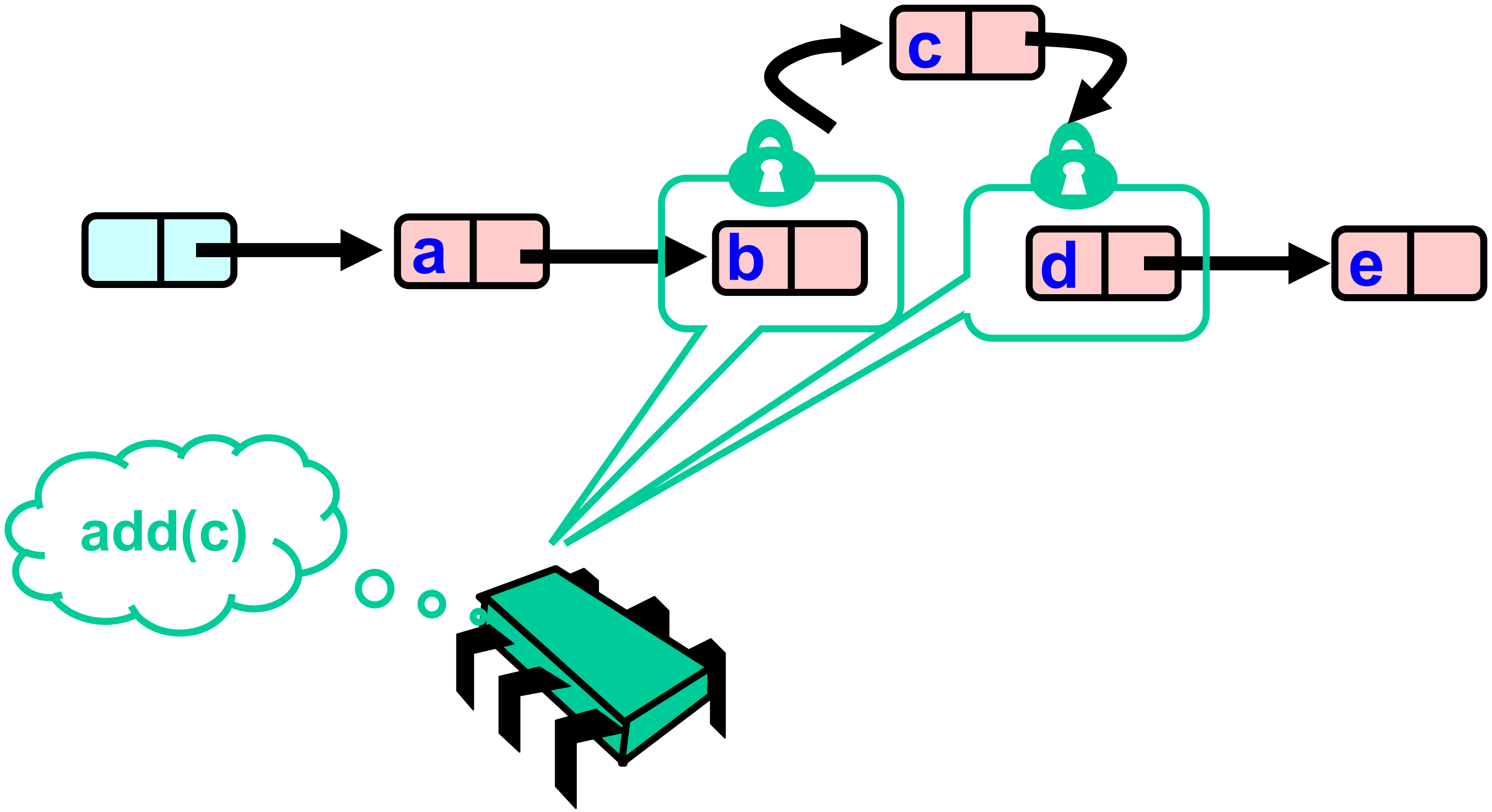


Optimistic: Lock and Load

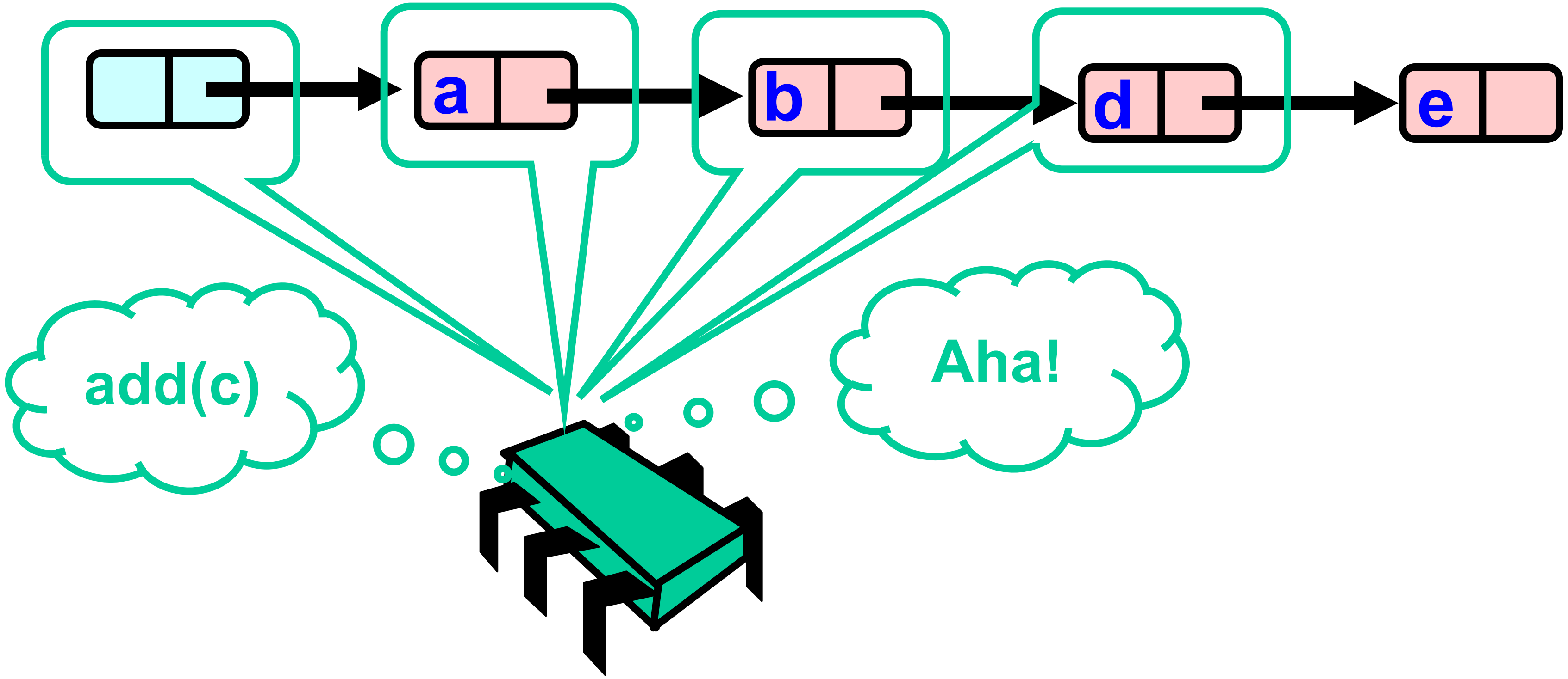




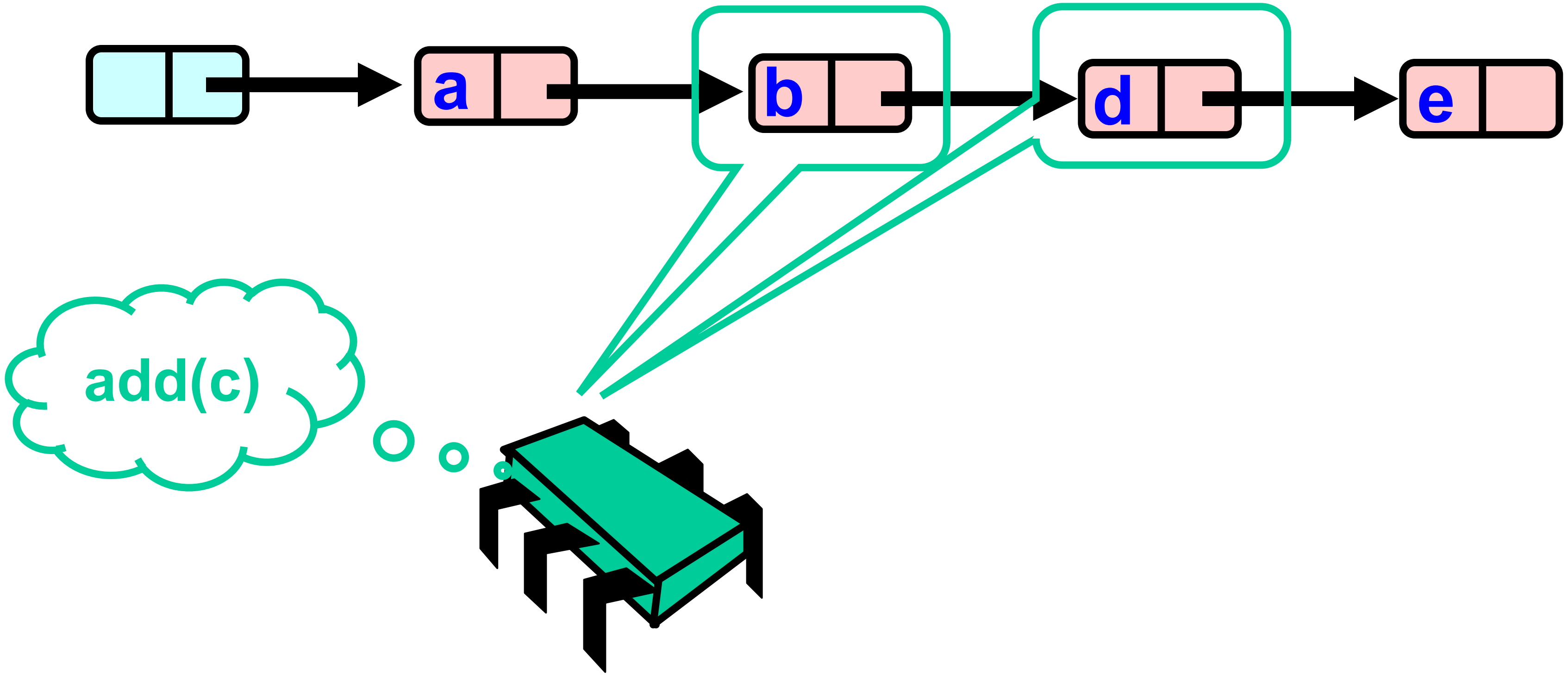
Optimistic: Lock and Load

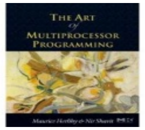


What could go wrong?

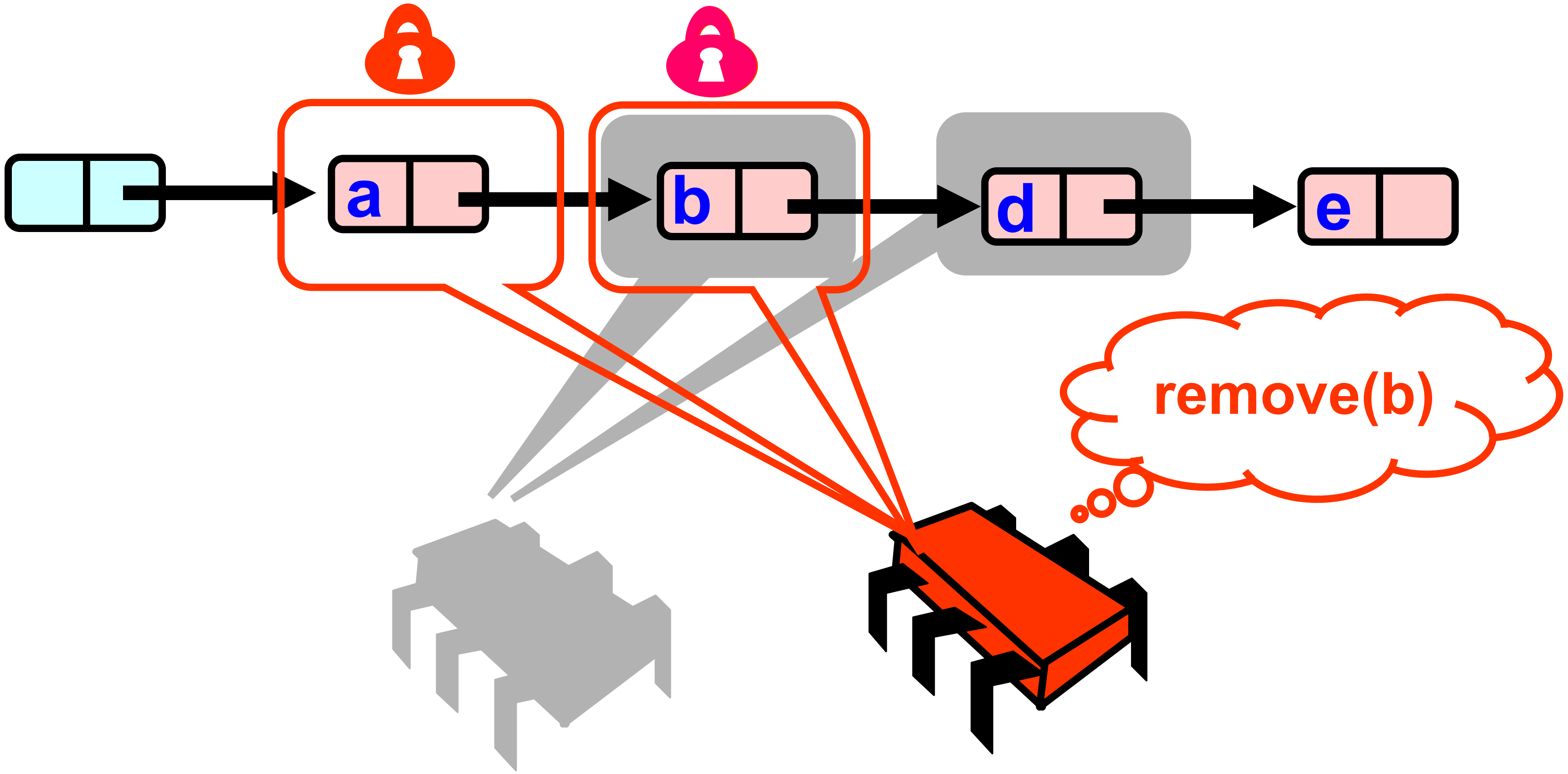


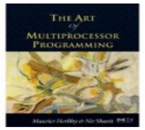
What could go wrong?



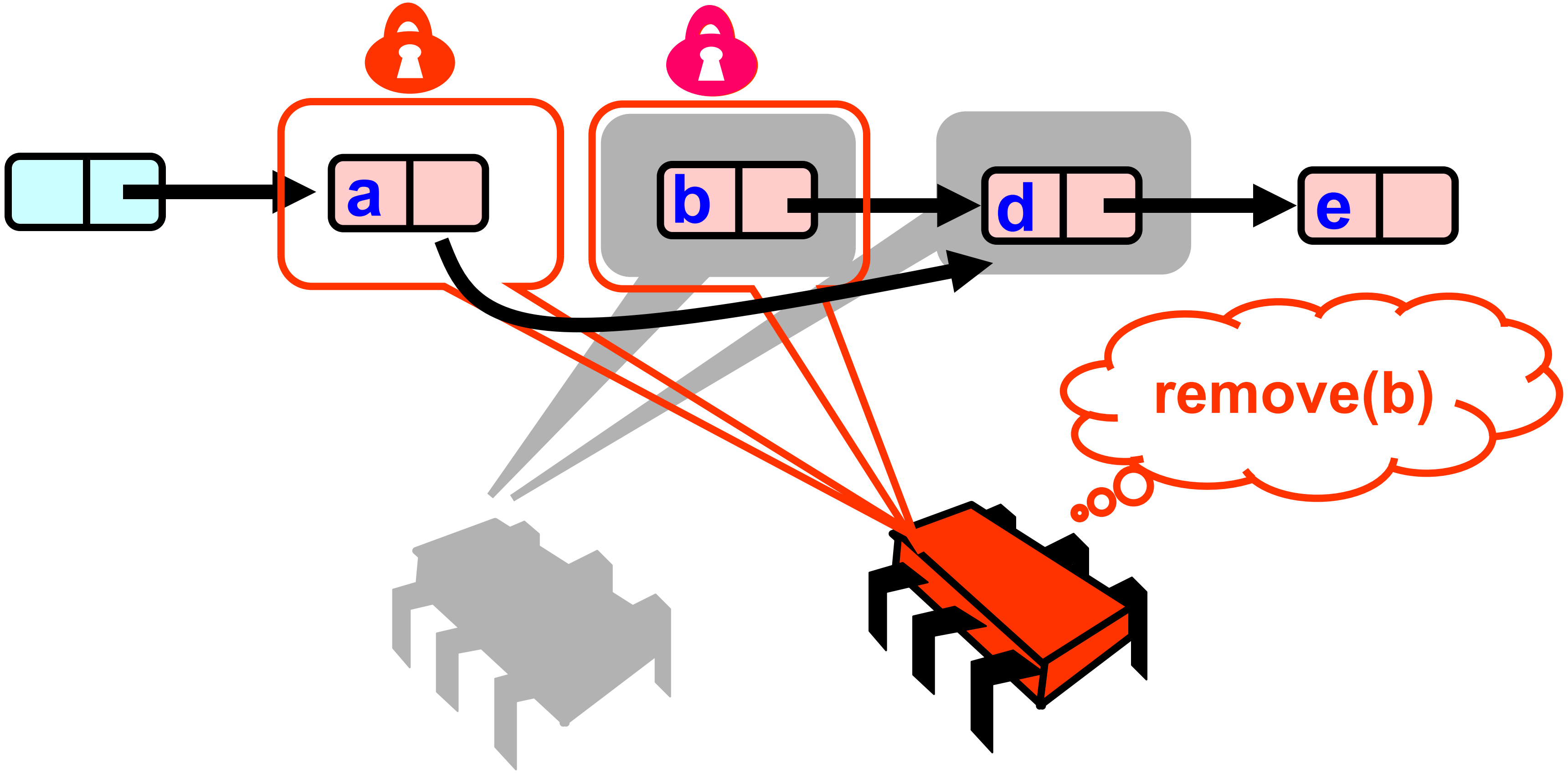


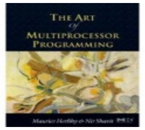
What could go wrong?



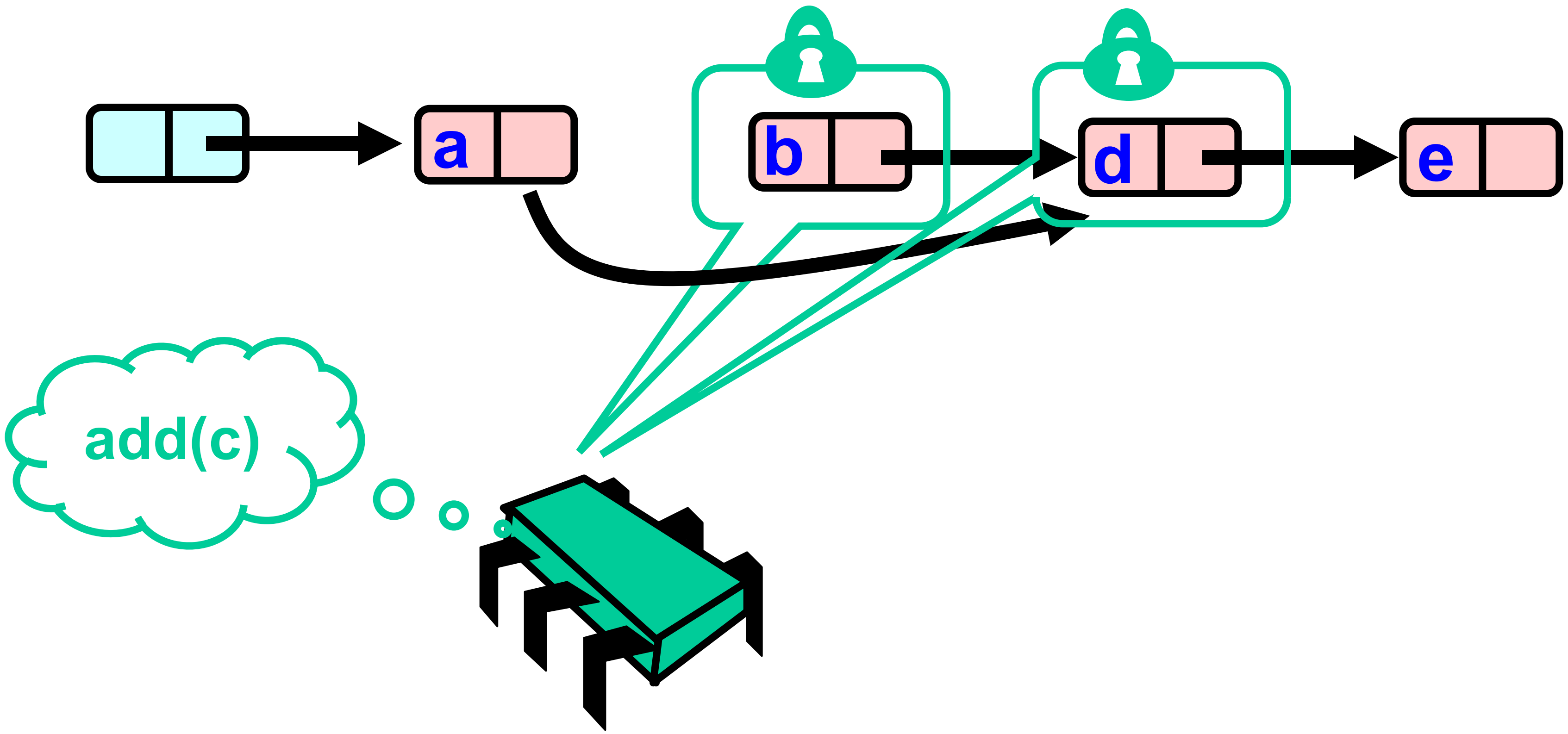


What could go wrong?

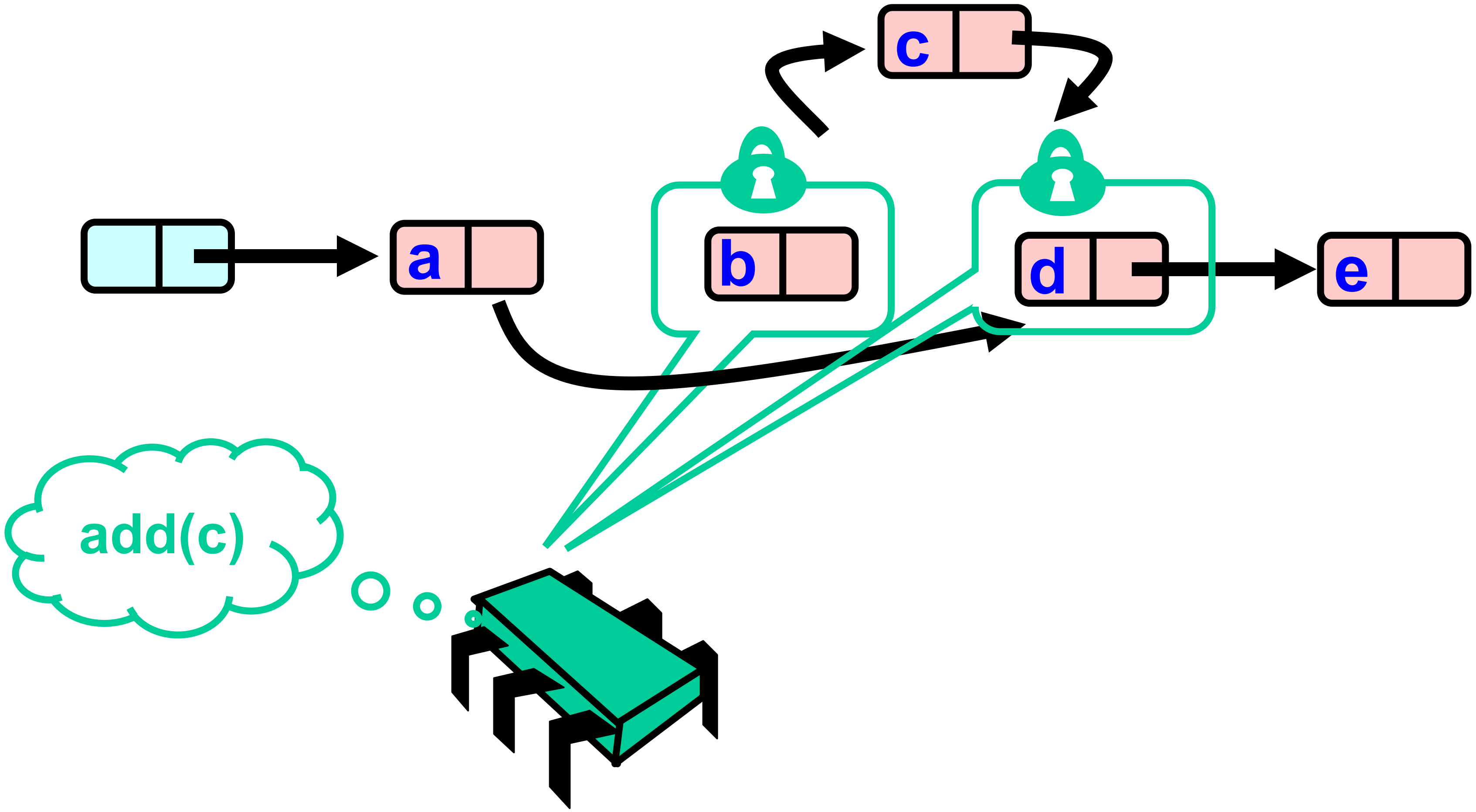


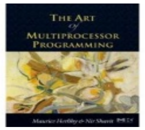


What could go wrong?

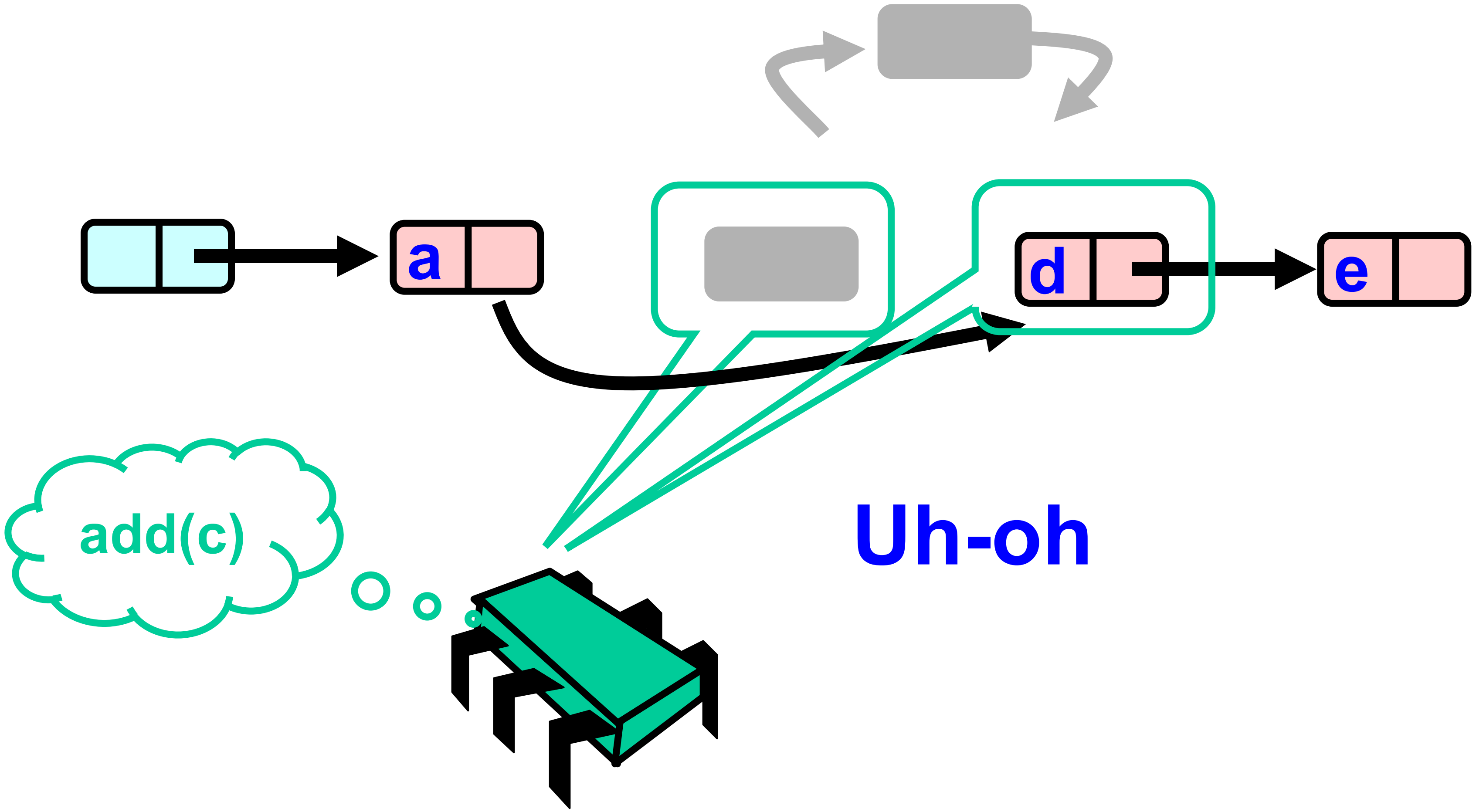


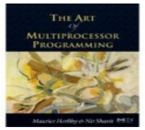
What could go wrong?



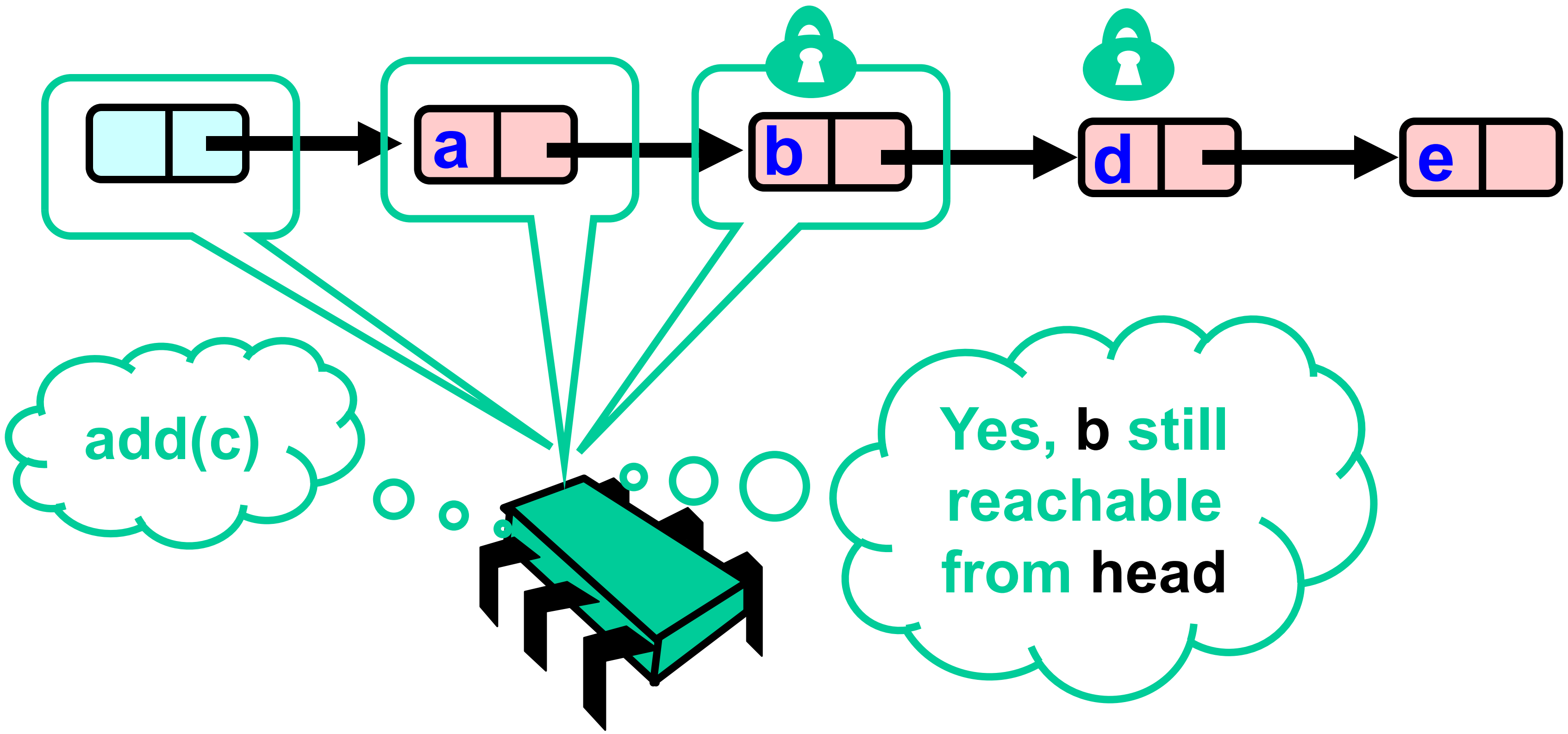


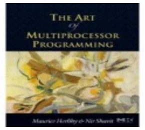
What could go wrong?



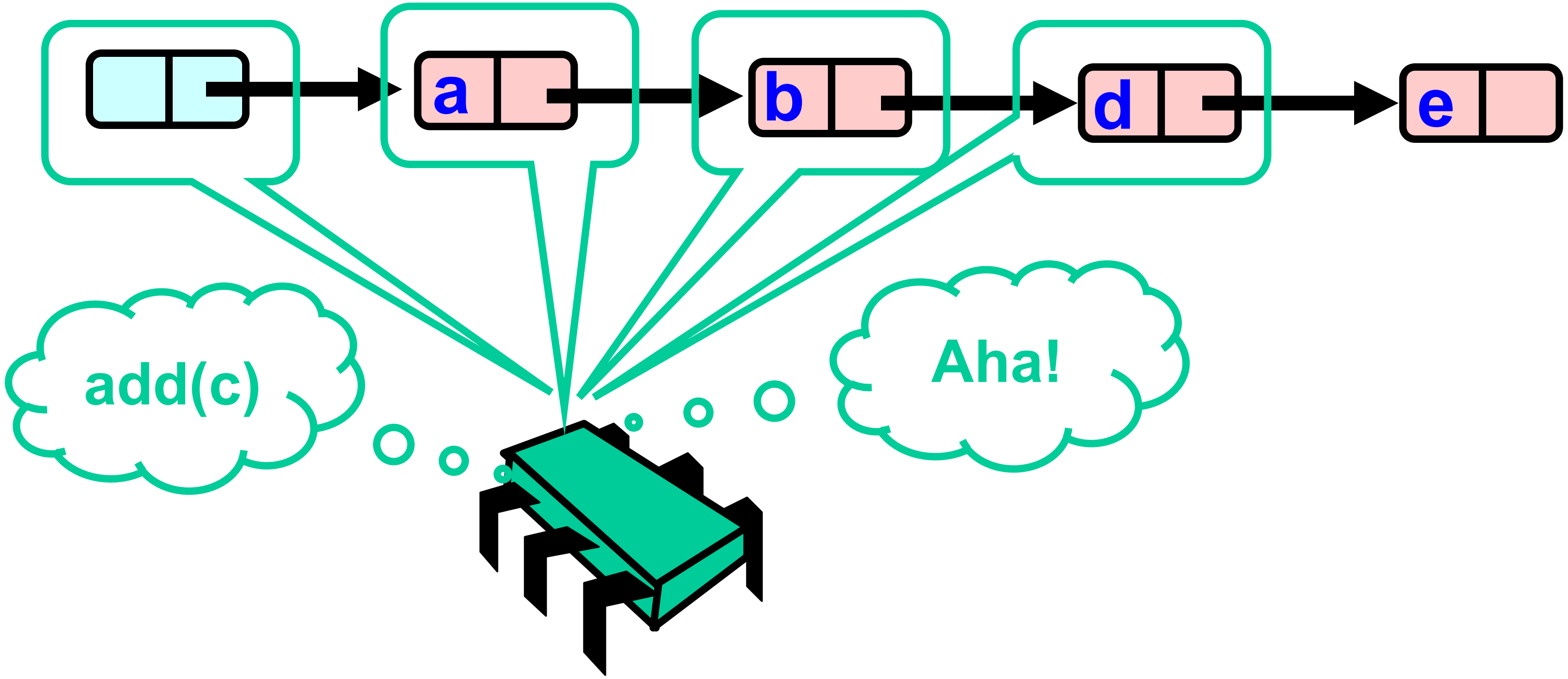


Validate – Part 1

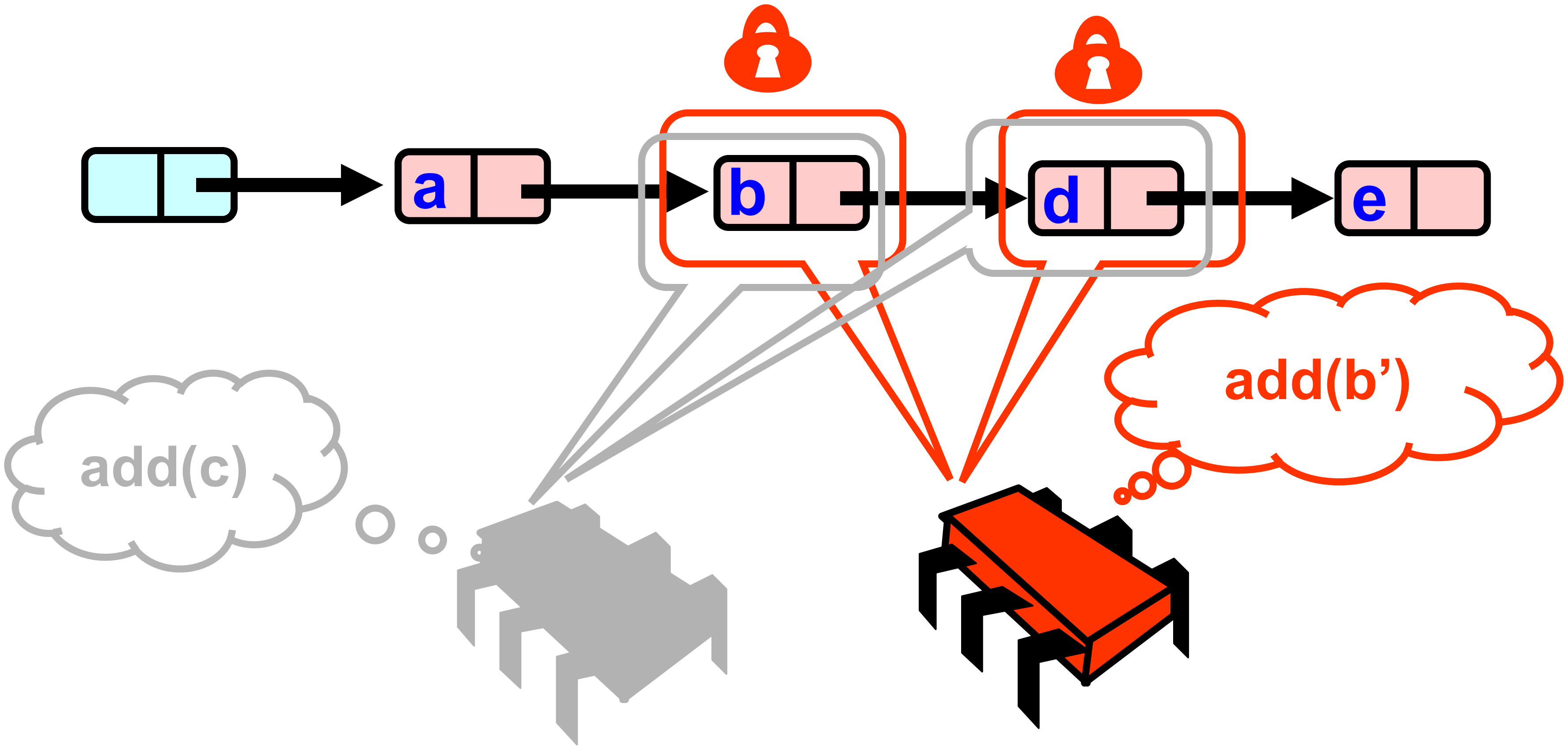


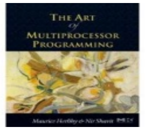


What Else Could Go Wrong?

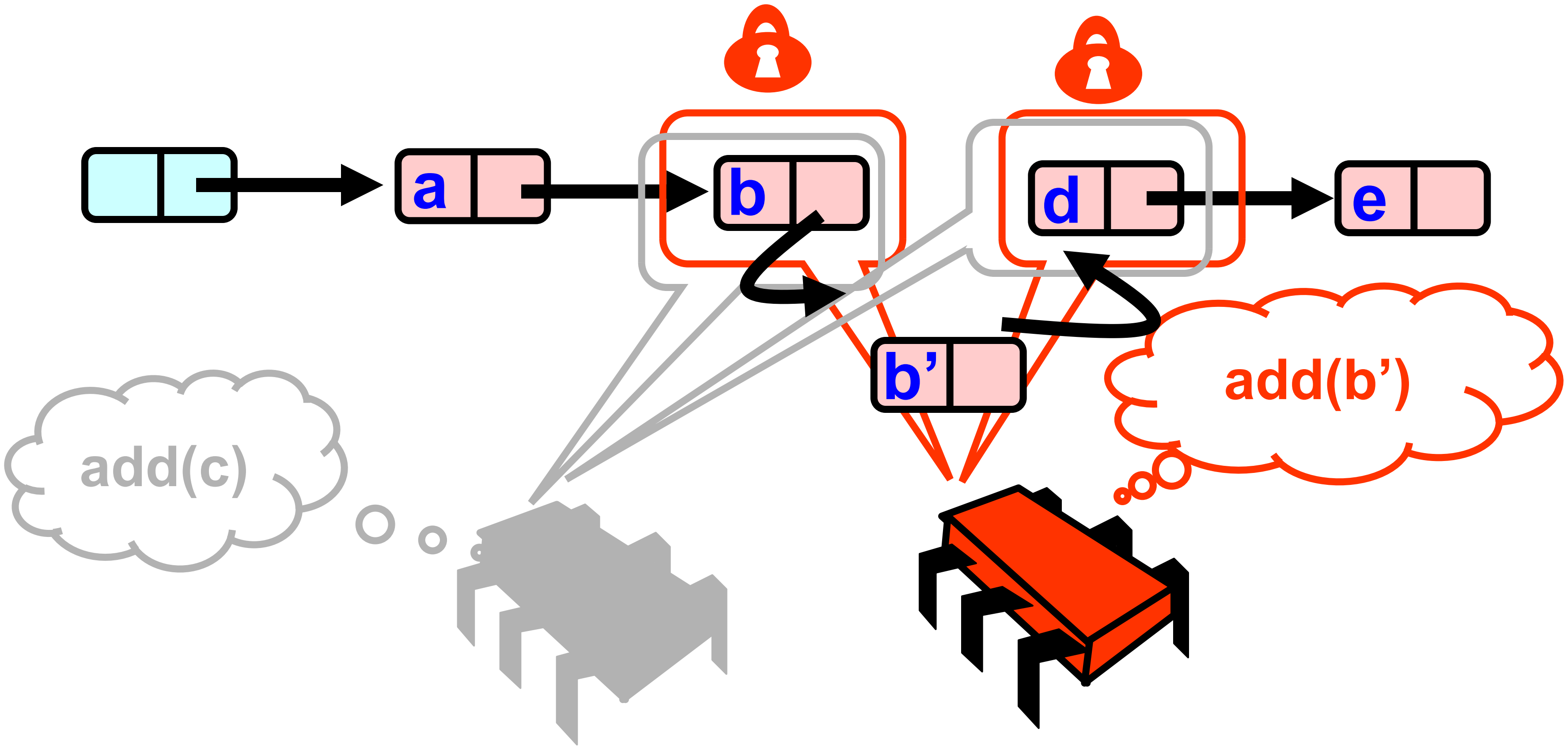


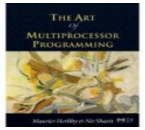
What Else Coould Go Wrong?



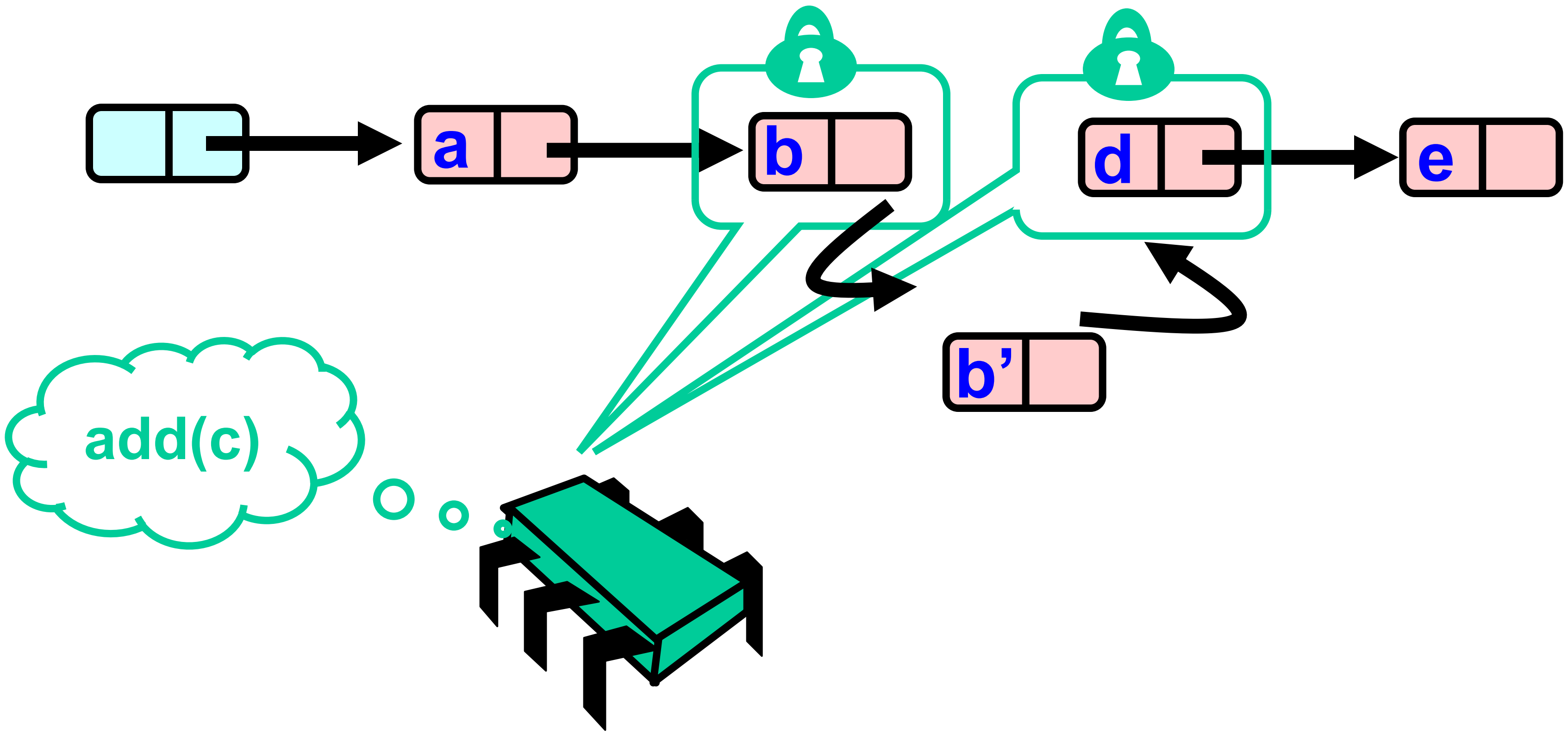


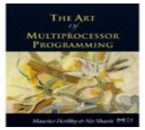
What Else Coould Go Wrong?



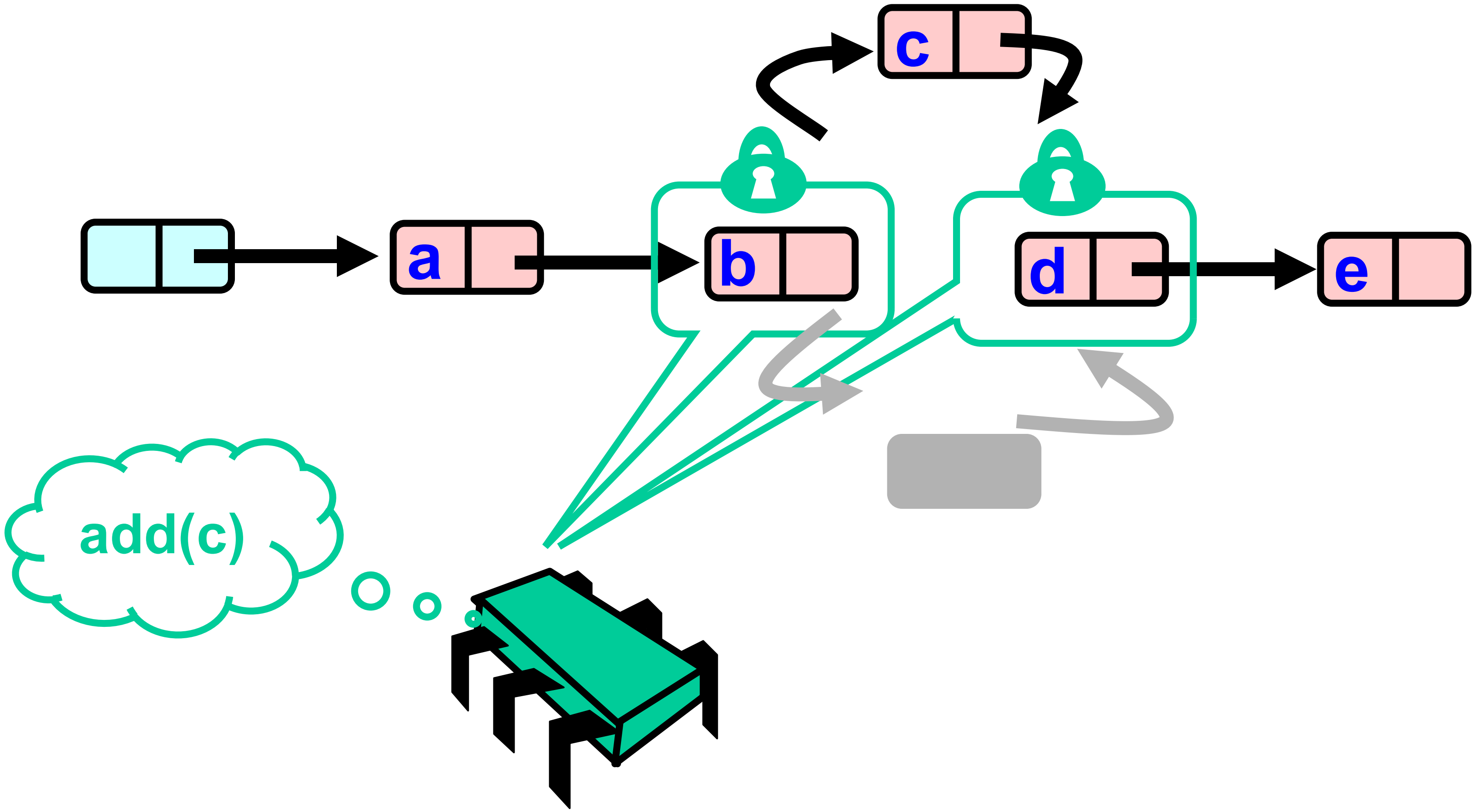


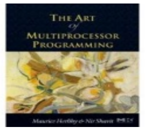
What Else Could Go Wrong?



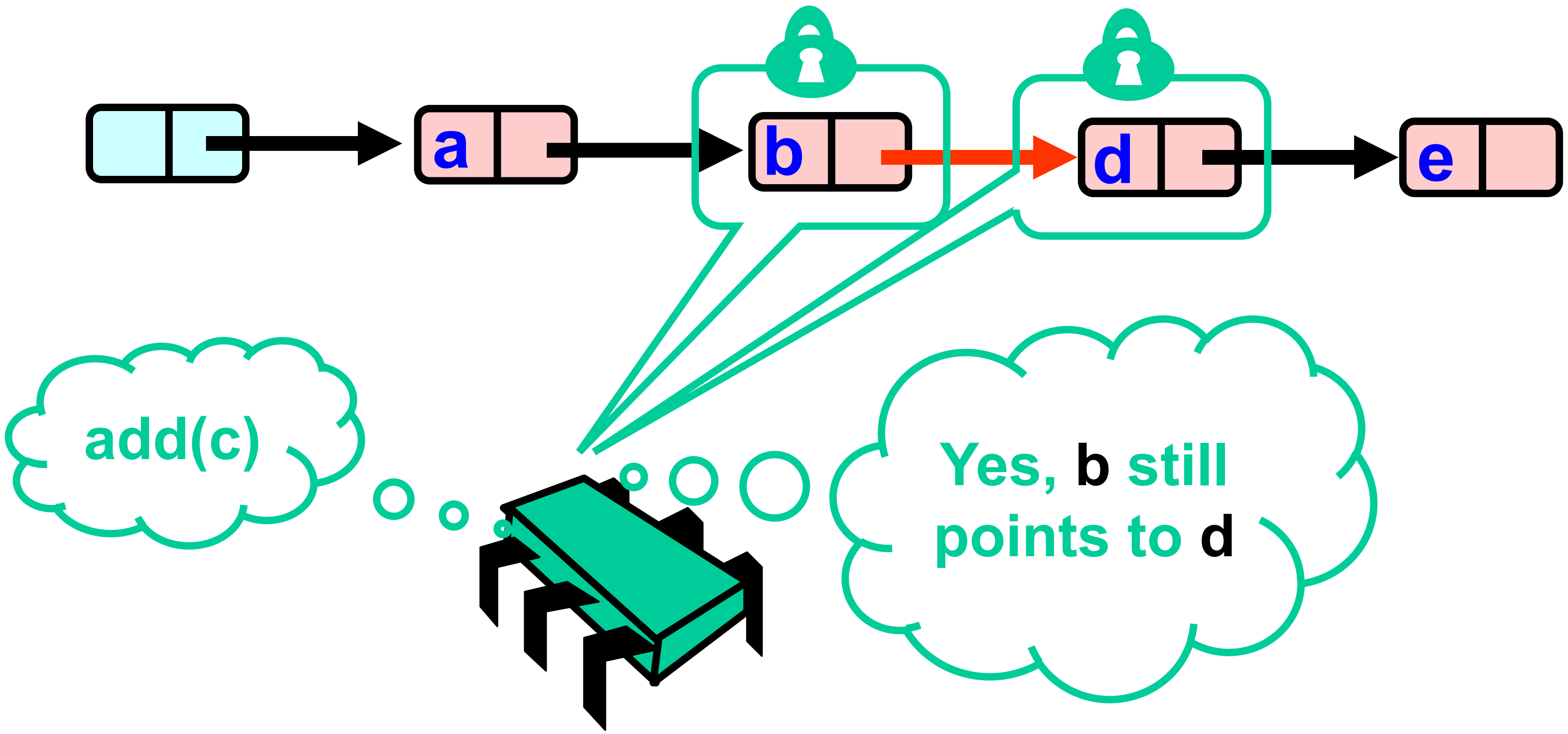


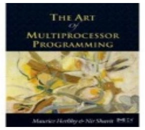
What Else Could Go Wrong?



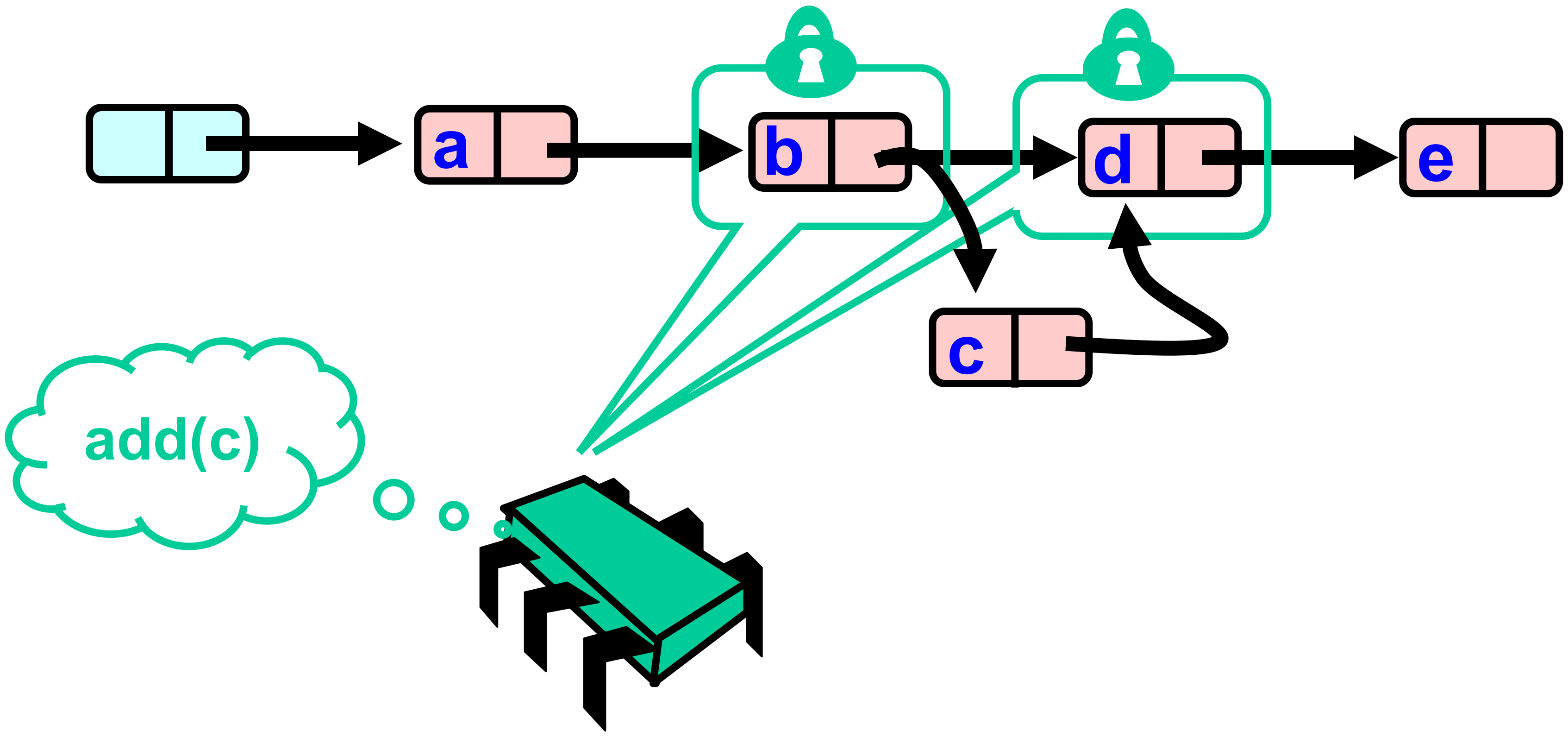


Validate Part 2 (while holding locks)



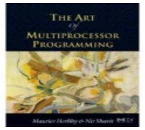


Optimistic: Linearization Point



Same Abstraction Map

- S(head) =
  - { x | there exists a such that
    - a reachable from head and
    - a.item = x
  - }



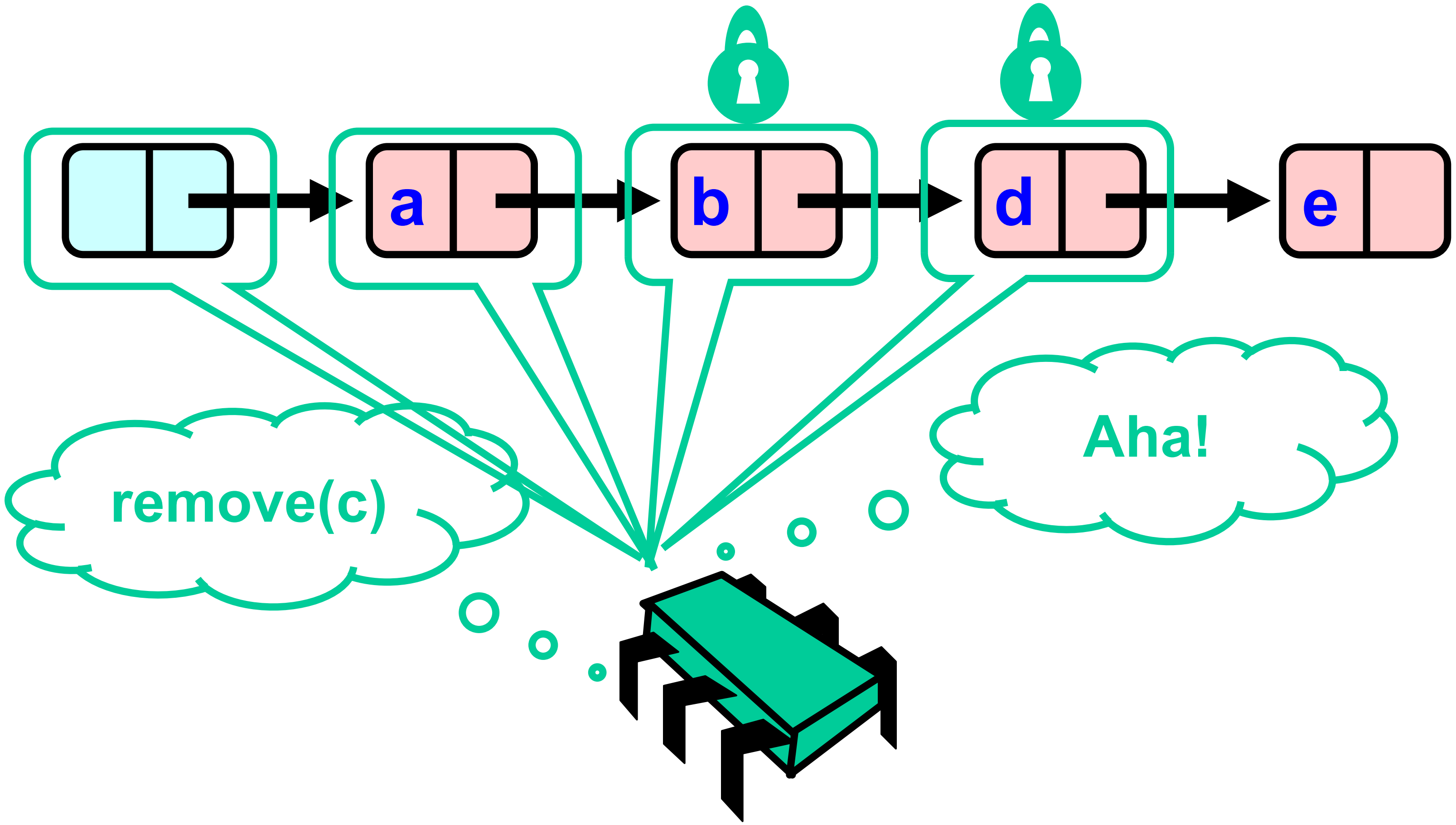
Invariants

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

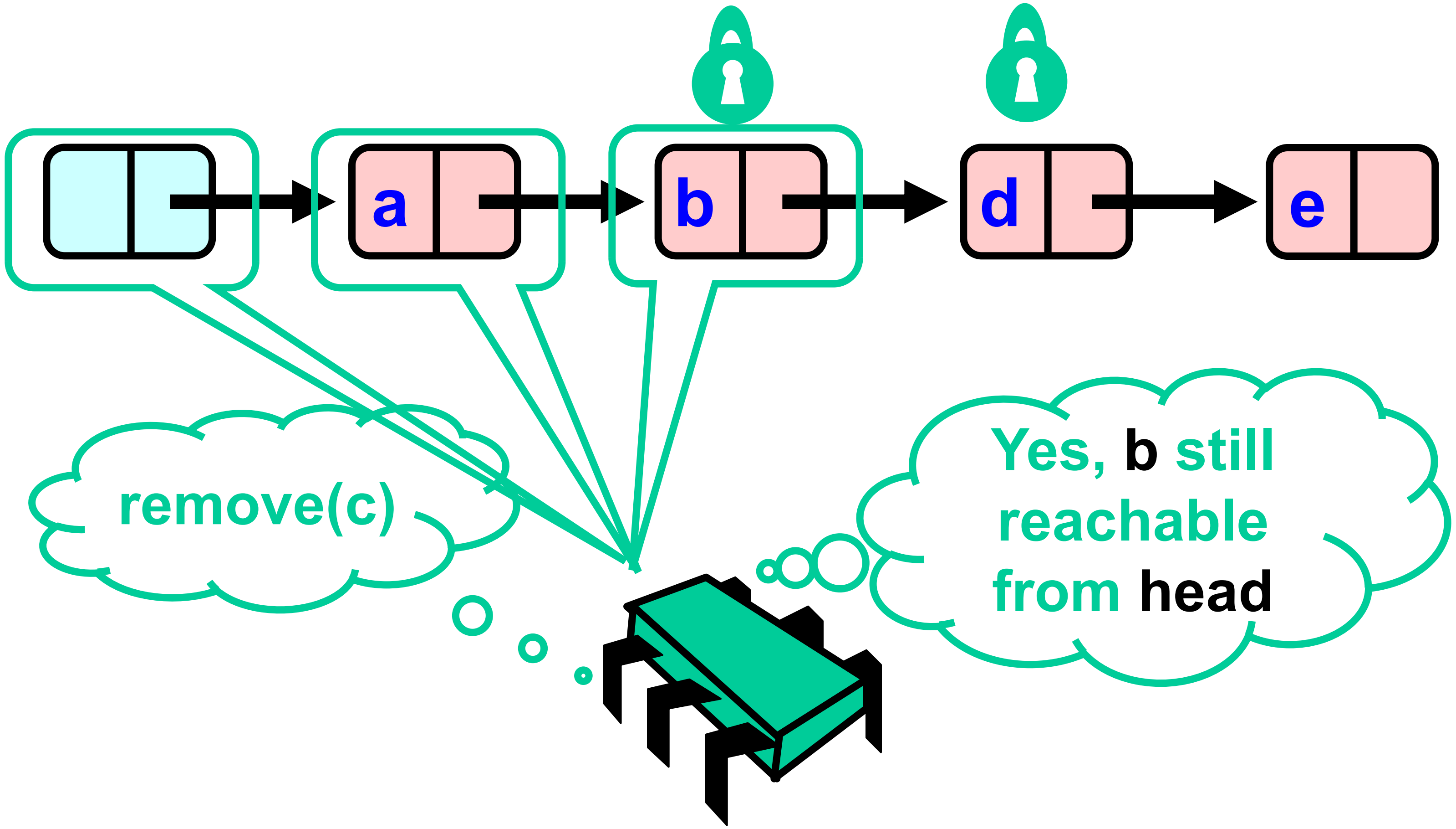
Correctness

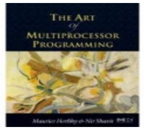
- If
  - Nodes b and c both locked
  - Node b still accessible
  - Node c still successor to b
- Then
  - Neither will be deleted
  - OK to delete and return true

Unsuccessful Remove

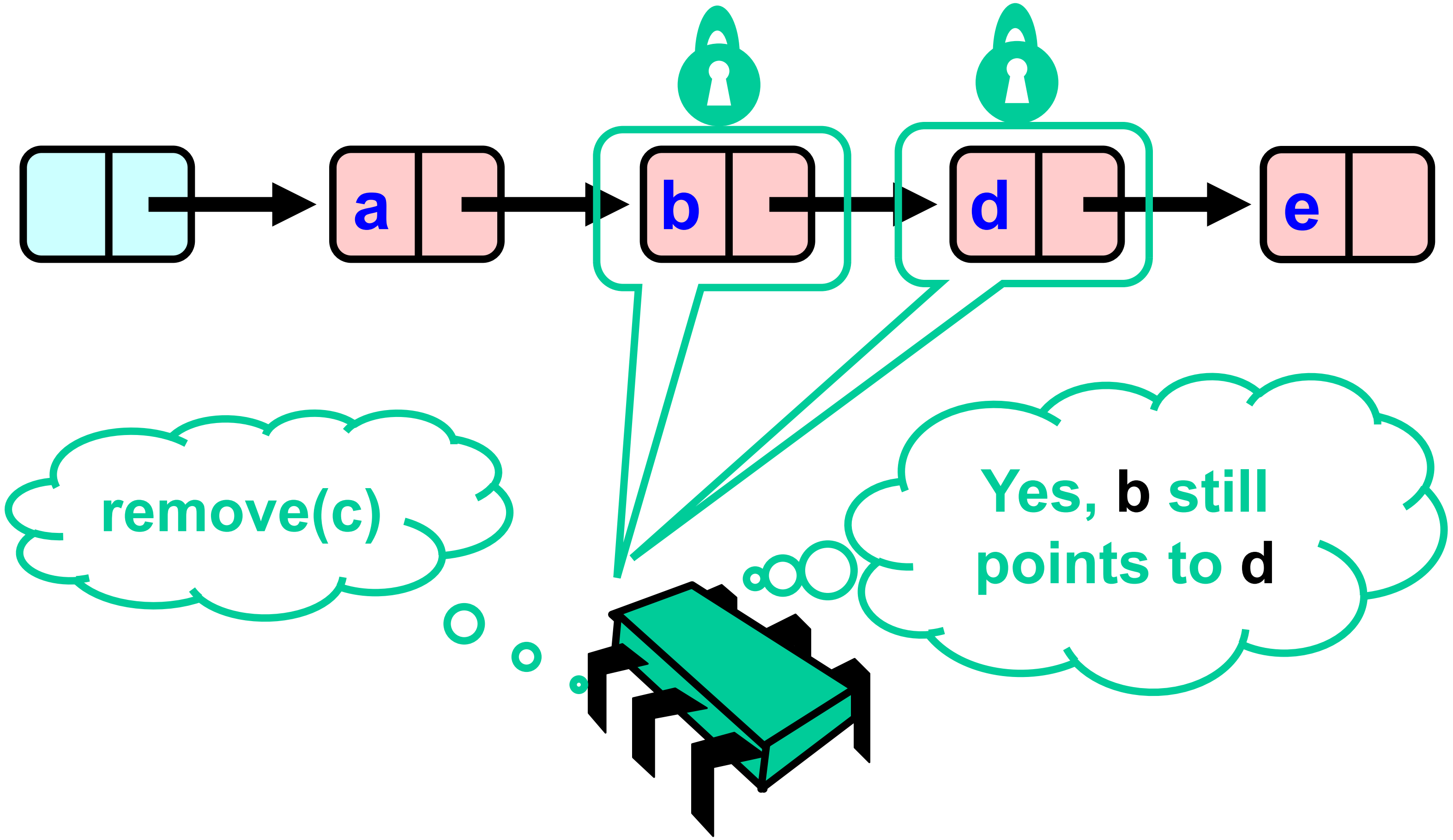


Validate (1)

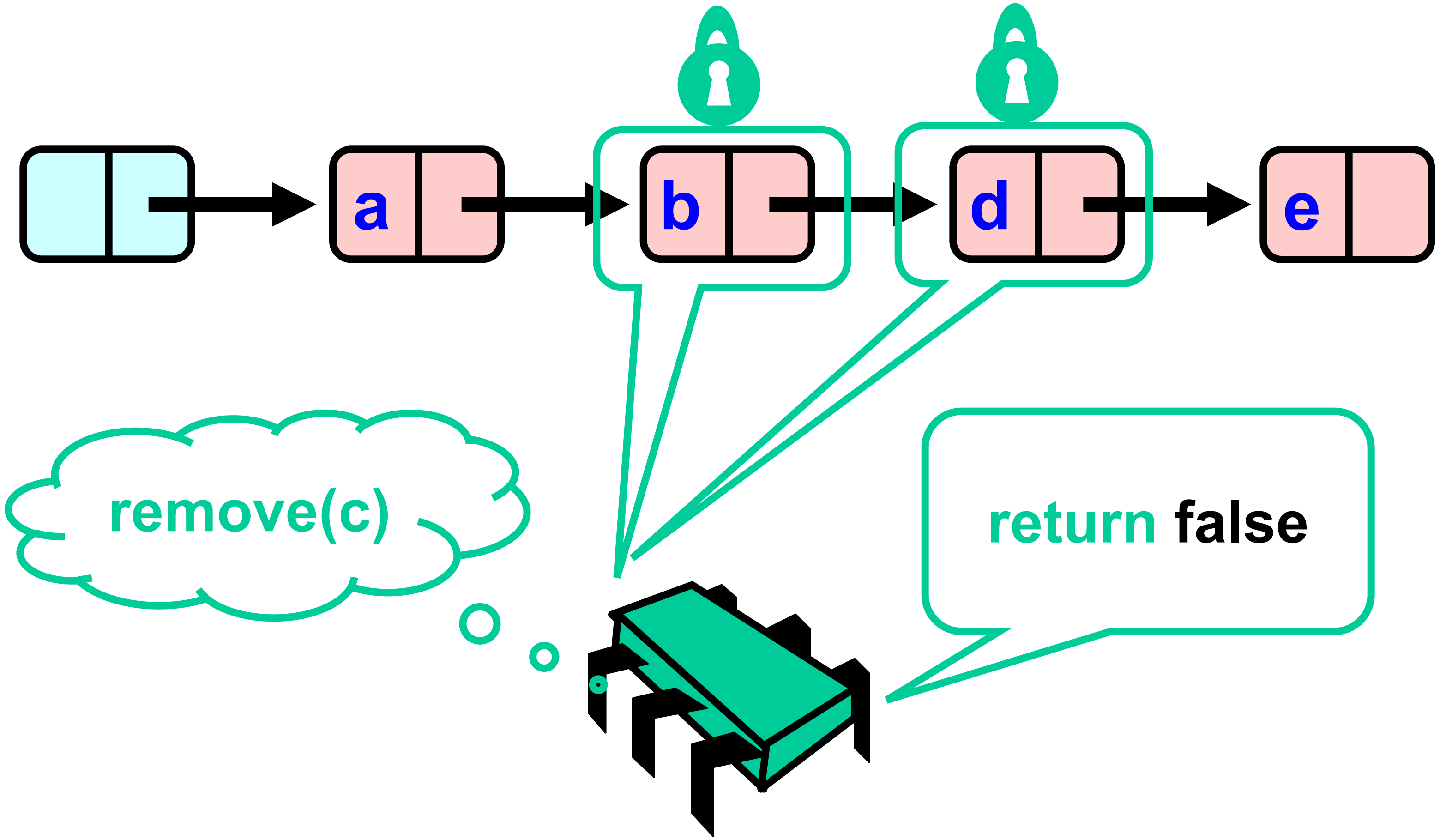


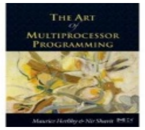


Validate (2)



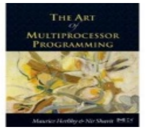
OK Computer





Correctness

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



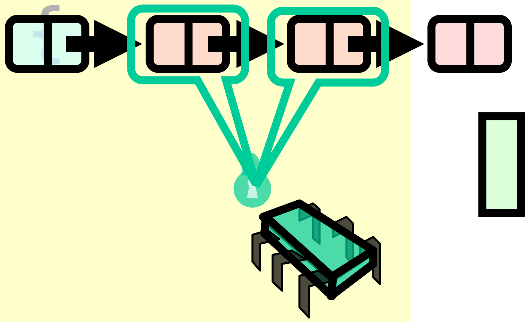
Validation

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

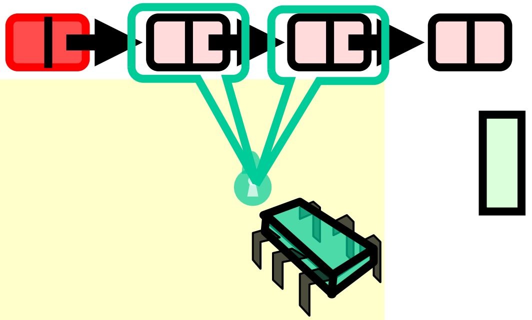
Validation

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

Predecessor & current nodes



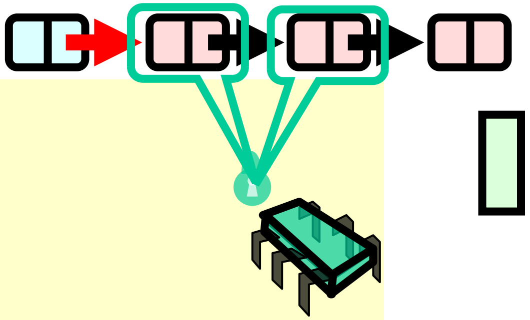
Validation



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

Begin at the beginning

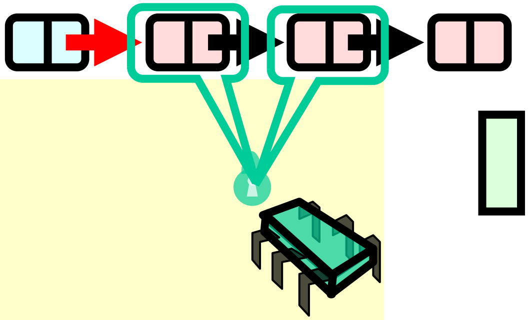
Validation



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

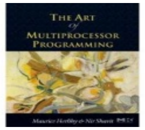
Search range of keys

Validation

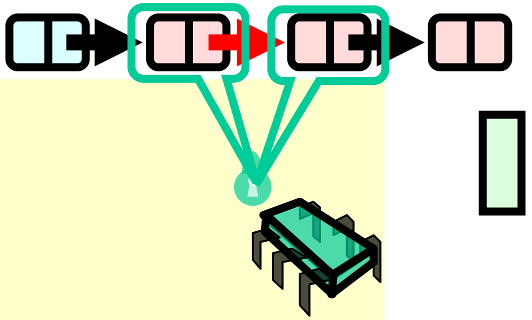


```
private boolean
 validate(Node pred,
          Node curr) {
 Node node = head;
 while (node.key <= pred.key) {
  if (node == pred)
   return pred.next == curr;
  node = node.next;
 }
 return false;
}
```

Predecessor reachable



Validation



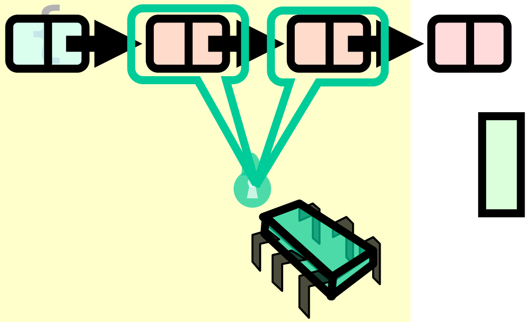
```
private boolean
 validate(Node pred,
          Node curr) {
 Node node = head;
 while (node.key <= pred.key) {
  if (node == pred)
   return pred.next == curr;
  node = node.next;
 }
 return false;
}
```

Is current node next?

Validation

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

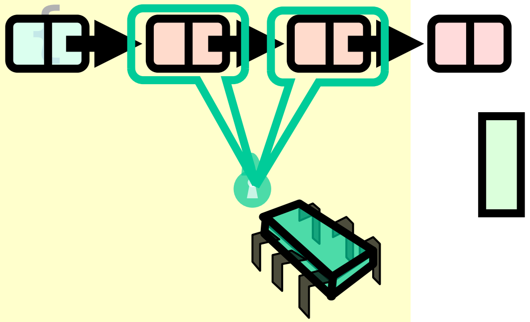
Otherwise move on



Validation

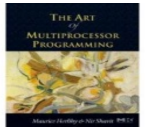
Predecessor not reachable

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



Remove: searching

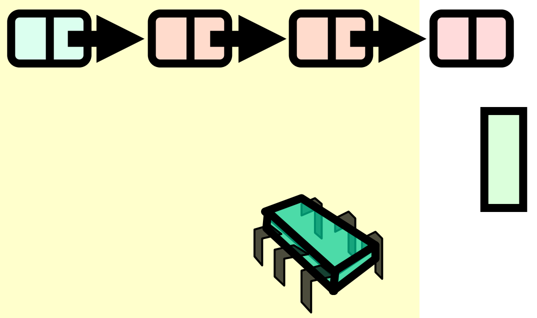
```
public boolean remove(Item item) {
 int key = item.hashCode();
 retry: while (true) {
   Node pred = this.head;
   Node curr = pred.next;
   while (curr.key <= key) {
    if (item == curr.item)
      break;
    pred = curr;
    curr = curr.next;
   } …
```

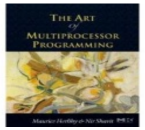


Remove: searching

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

Search key

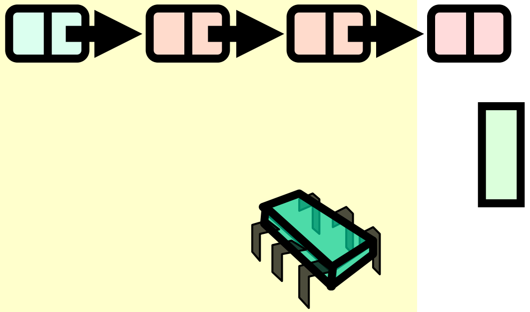


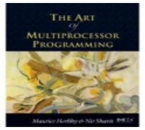


Remove: searching

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

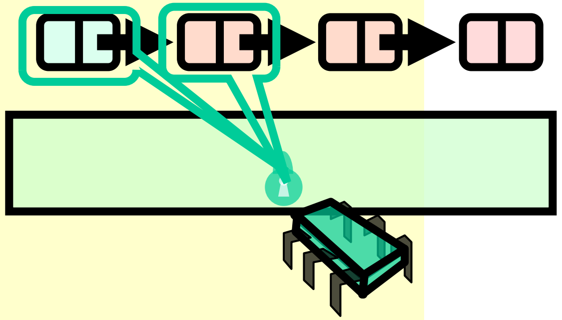
Retry on synchronization conflict





Remove: searching

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

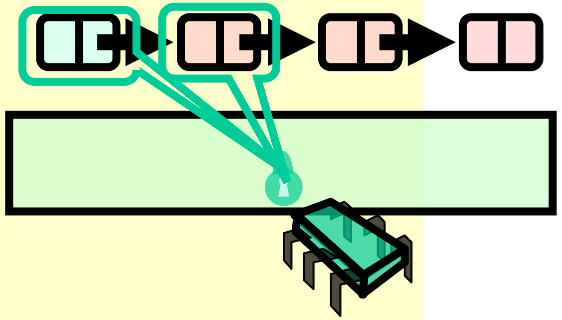


Examine predecessor and current nodes

Remove: searching

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

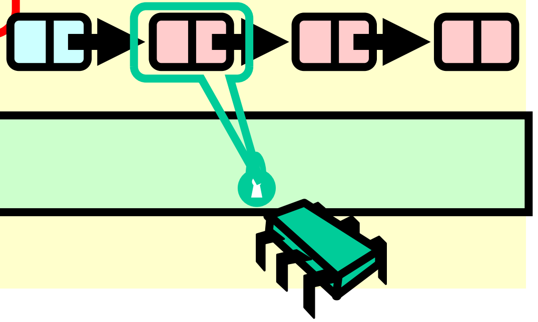
Search by key

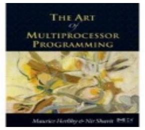


Remove: searching

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

Stop if we find item

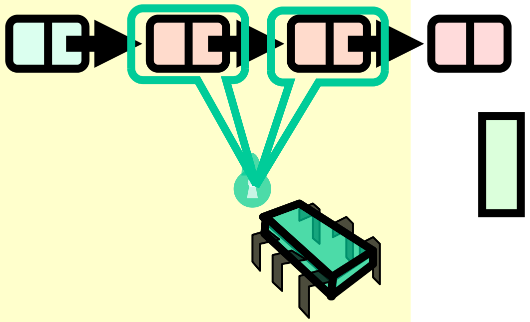




Remove: searching

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

Move along

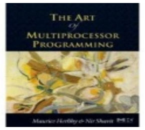


On Exit from Loop

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

Remove Method

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



Remove Method

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

Always unlock

Remove Method

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

Lock both nodes

Remove Method

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

Check for synchronization conflicts

Remove Method

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

target found, remove node

Remove Method

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

target not found

Optimistic List

- Limited hot-spots
  - Targets of add(), remove(), contains()
  - No contention on traversals
- Moreover
  - Traversals are wait-free
  - Food for thought …

So Far, So Good

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

Evaluation

- Optimistic is effective if
  - cost of scanning twice without locks

    is less than
  - cost of scanning once with locks
- Drawback
  - contains() acquires locks
  - 90% of calls in many apps

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

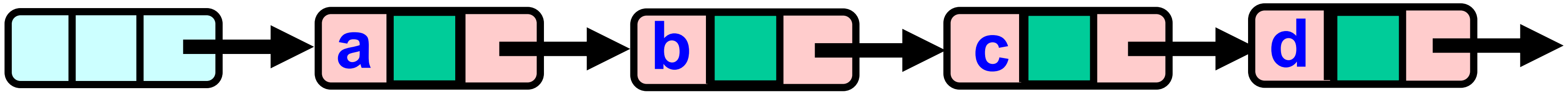
Lazy List

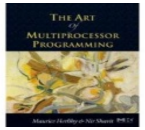
- Like optimistic, except
  - Scan once
  - `contains(x)` never locks …
- Key insight
  - Removing nodes causes trouble
  - Do it "lazily"

Lazy List

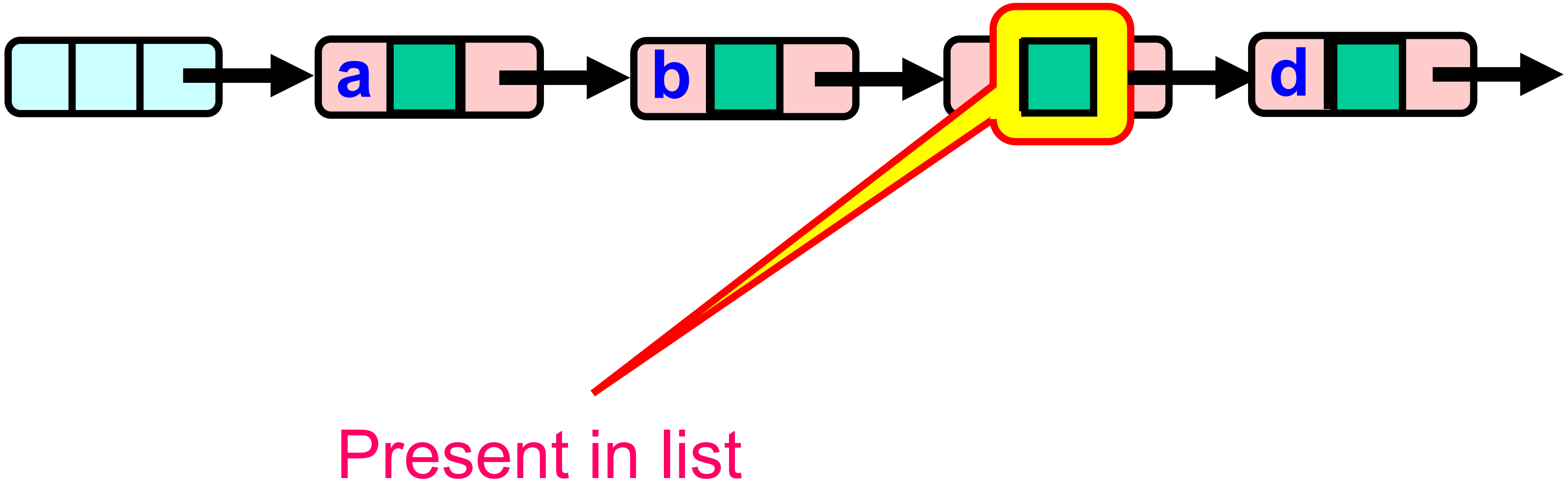
- `remove()`
  - Scans list (as before)
  - Locks predecessor & current (as before)
- Logical delete
  - Marks current node as removed (new!)
- Physical delete
  - Redirects predecessor's next (as before)

Lazy Removal

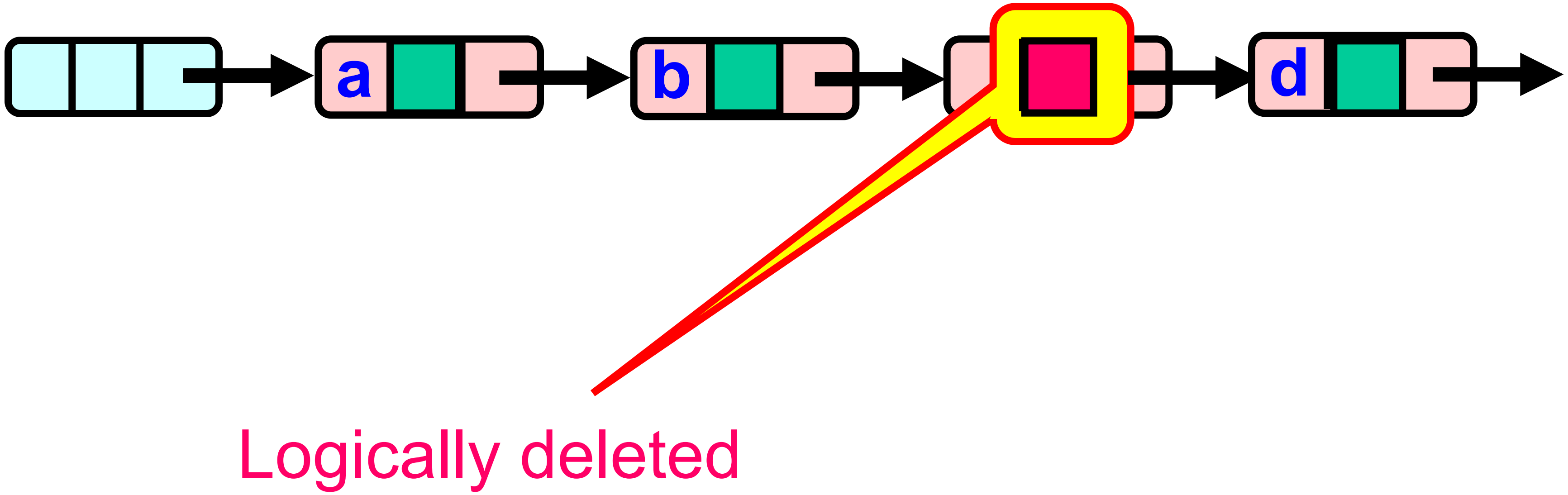




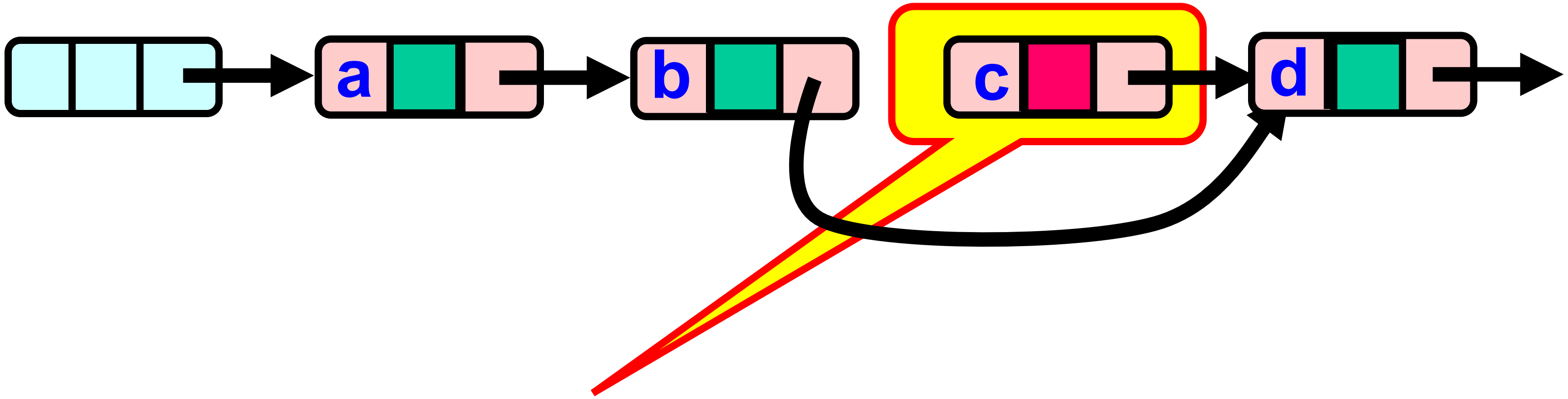
Lazy Removal



Lazy Removal

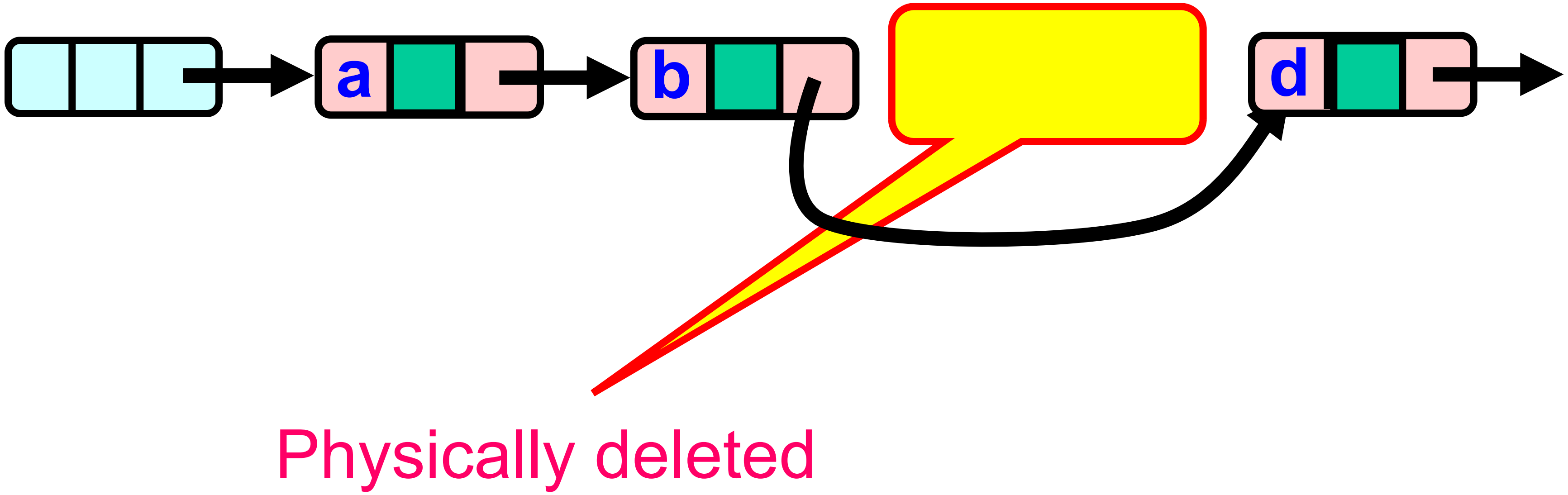


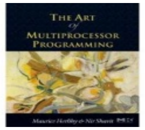
Lazy Removal



Physically deleted

Lazy Removal





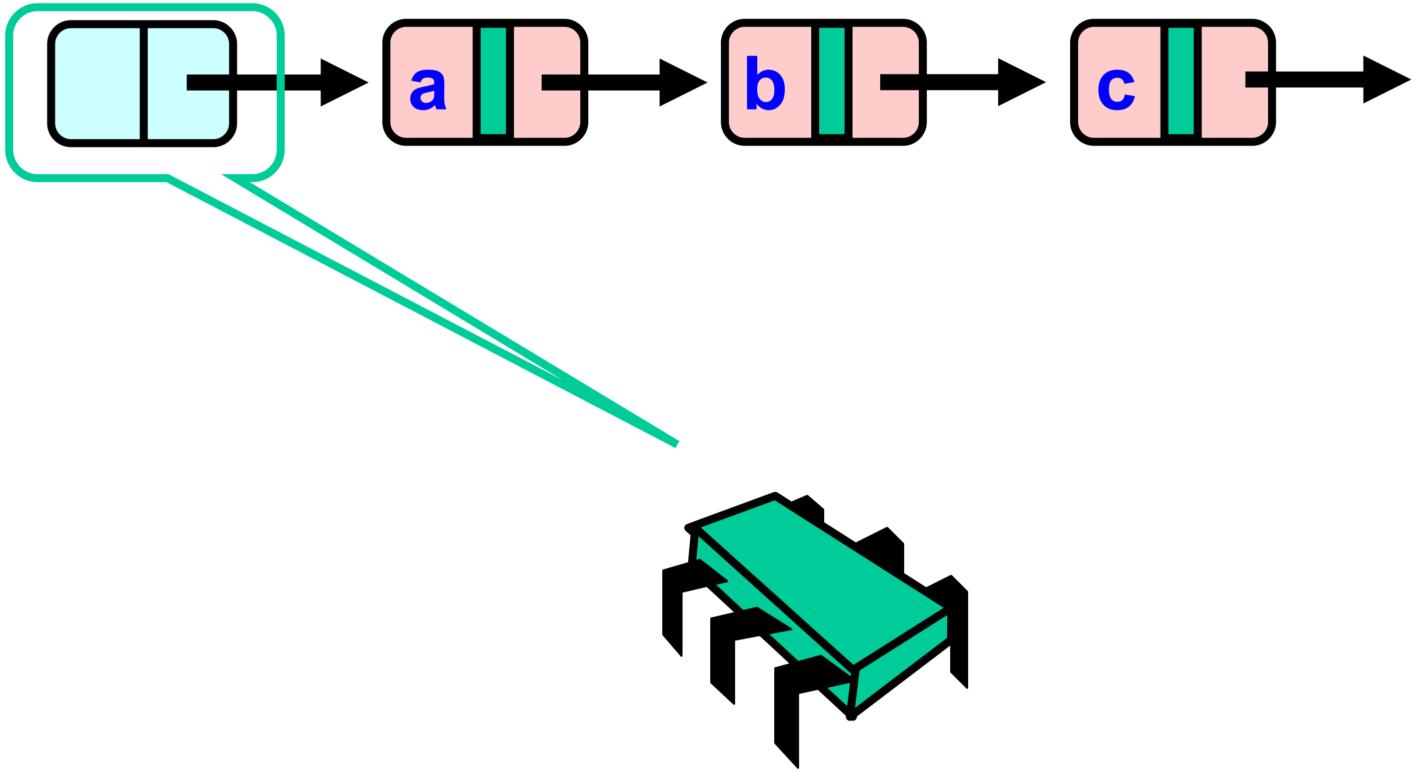
Lazy List

- All Methods
  - Scan through locked and marked nodes
  - Removing a node doesn’t slow down other method calls …
- Must still lock pred and curr nodes.

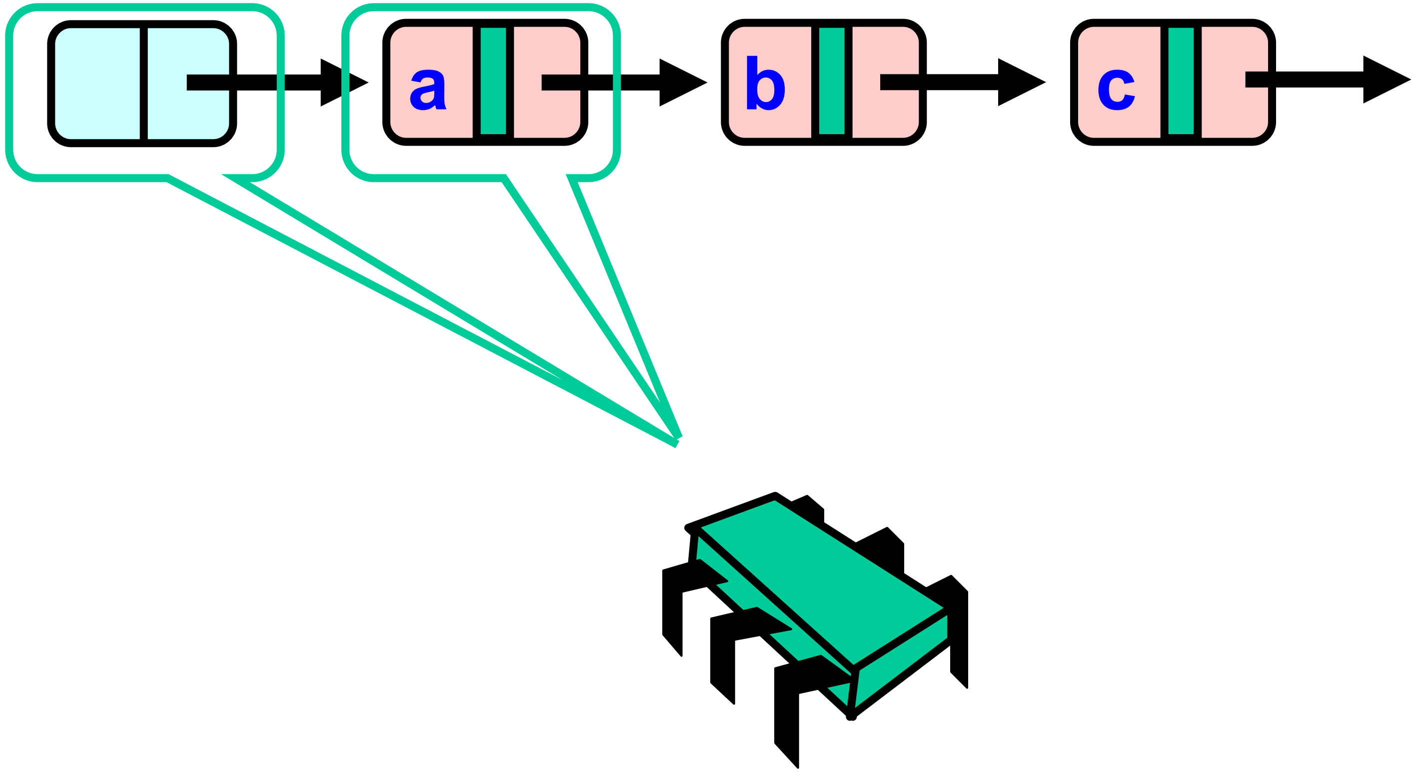
Validation

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

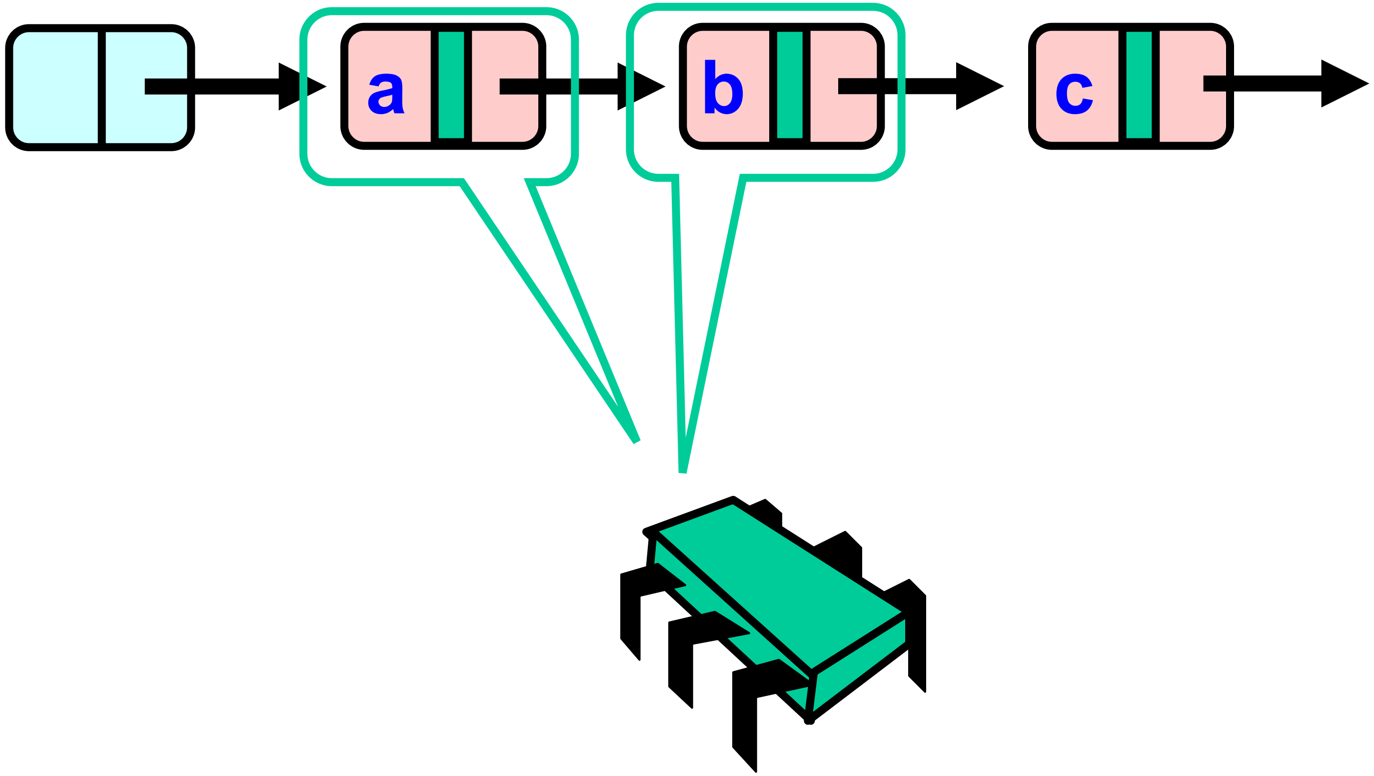
Business as Usual

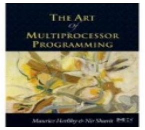


Business as Usual

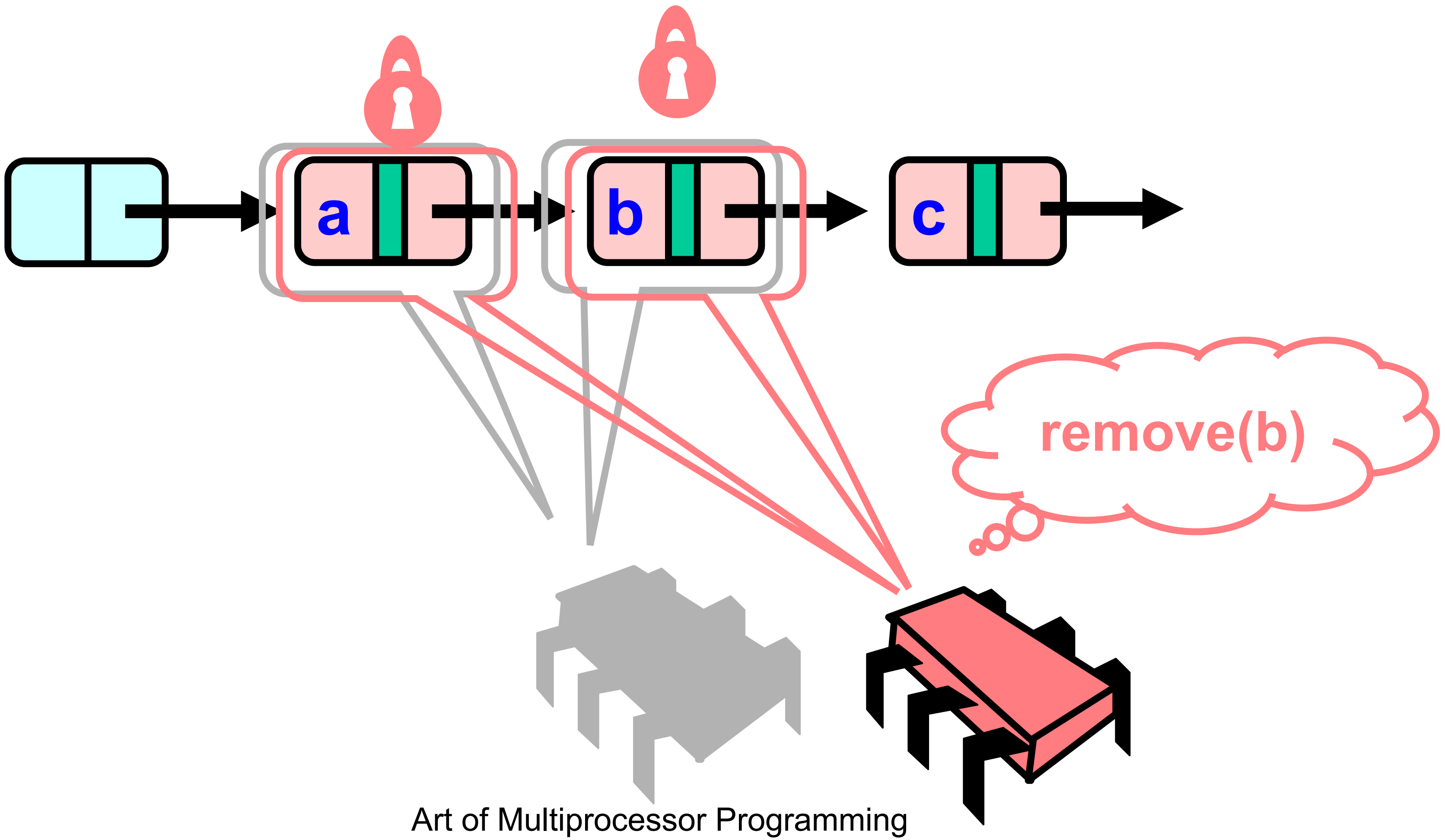


Business as Usual

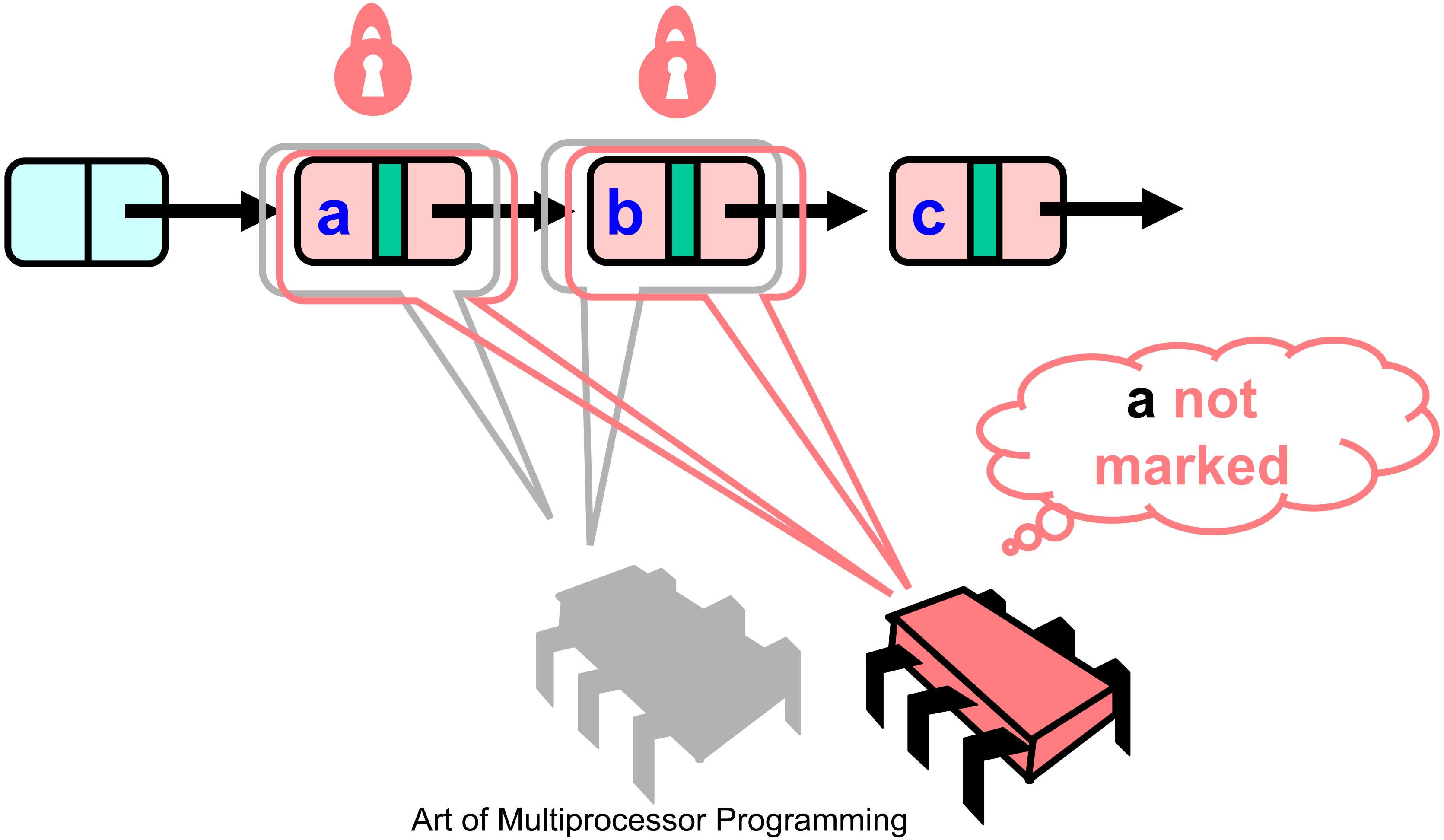




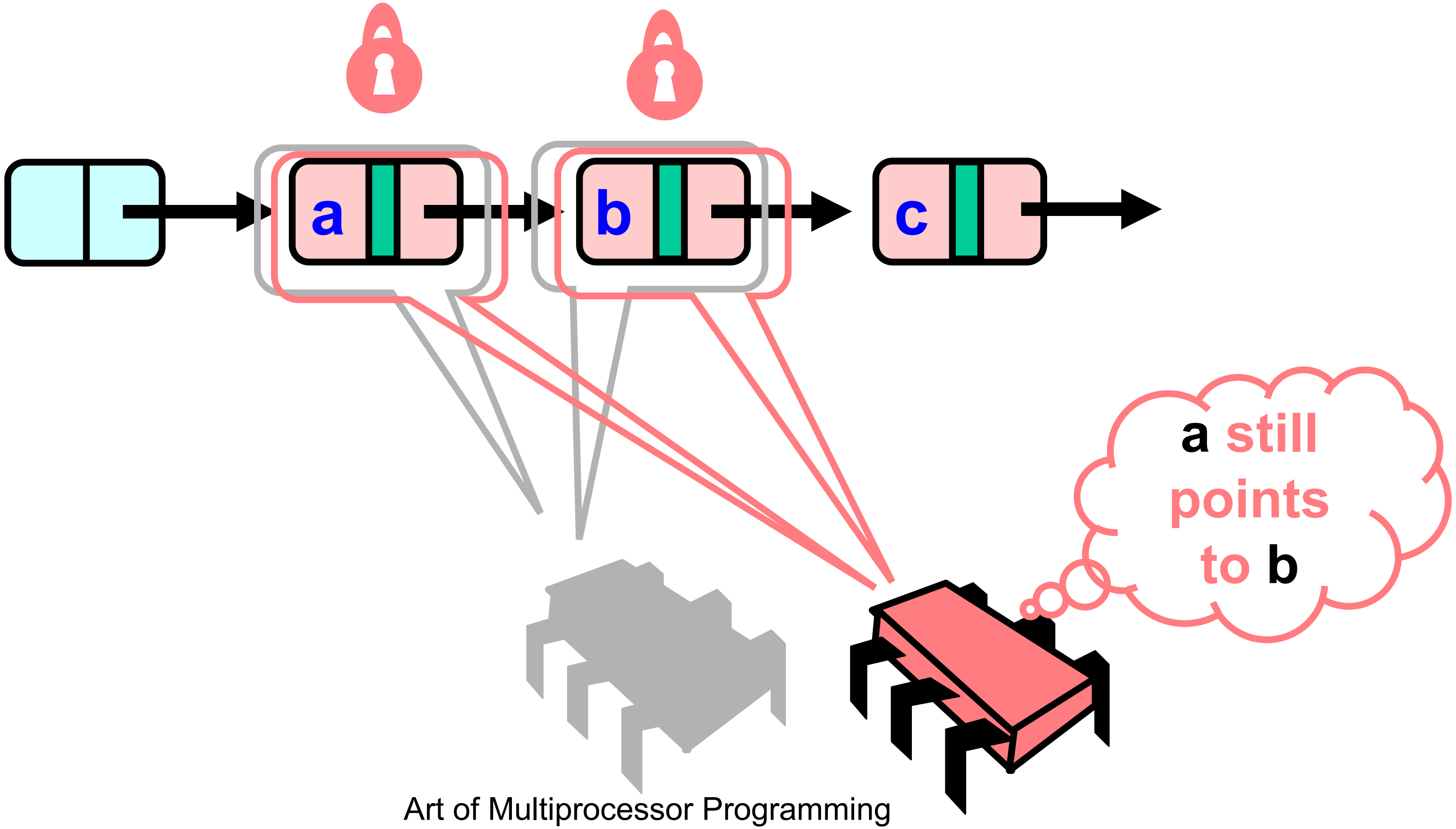
Business as Usual

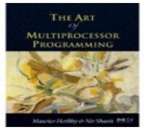


Business as Usual

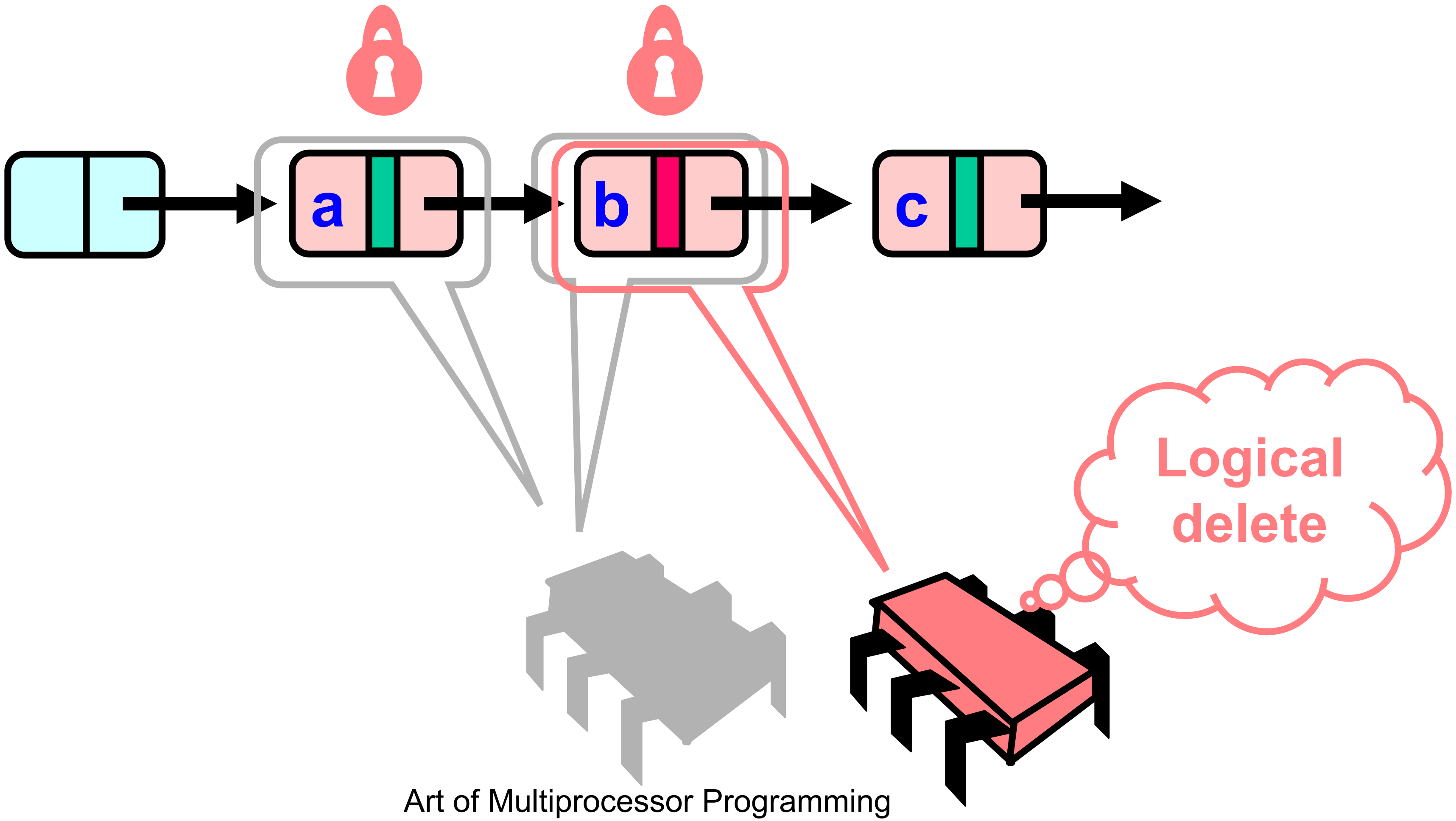


Business as Usual

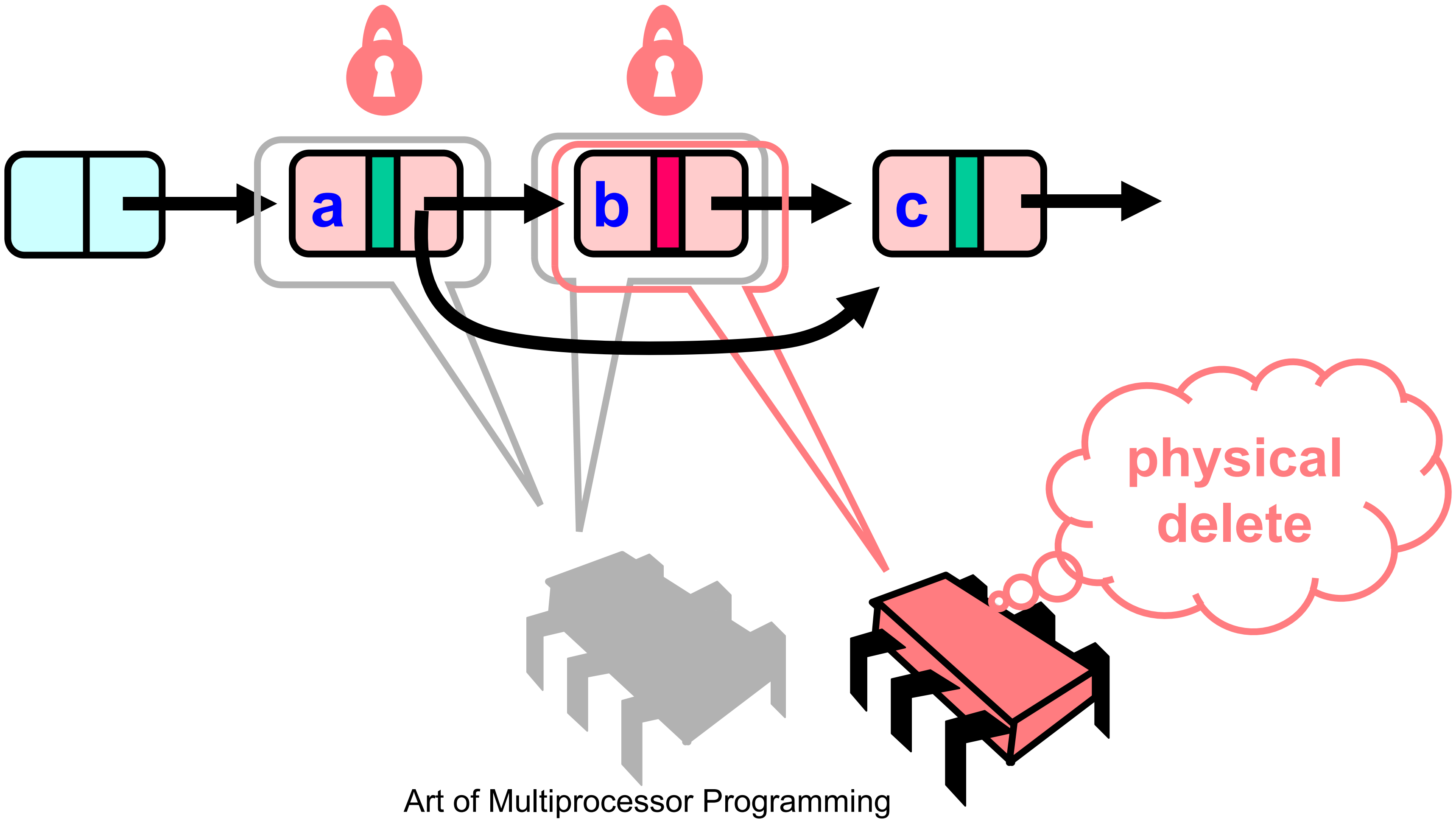




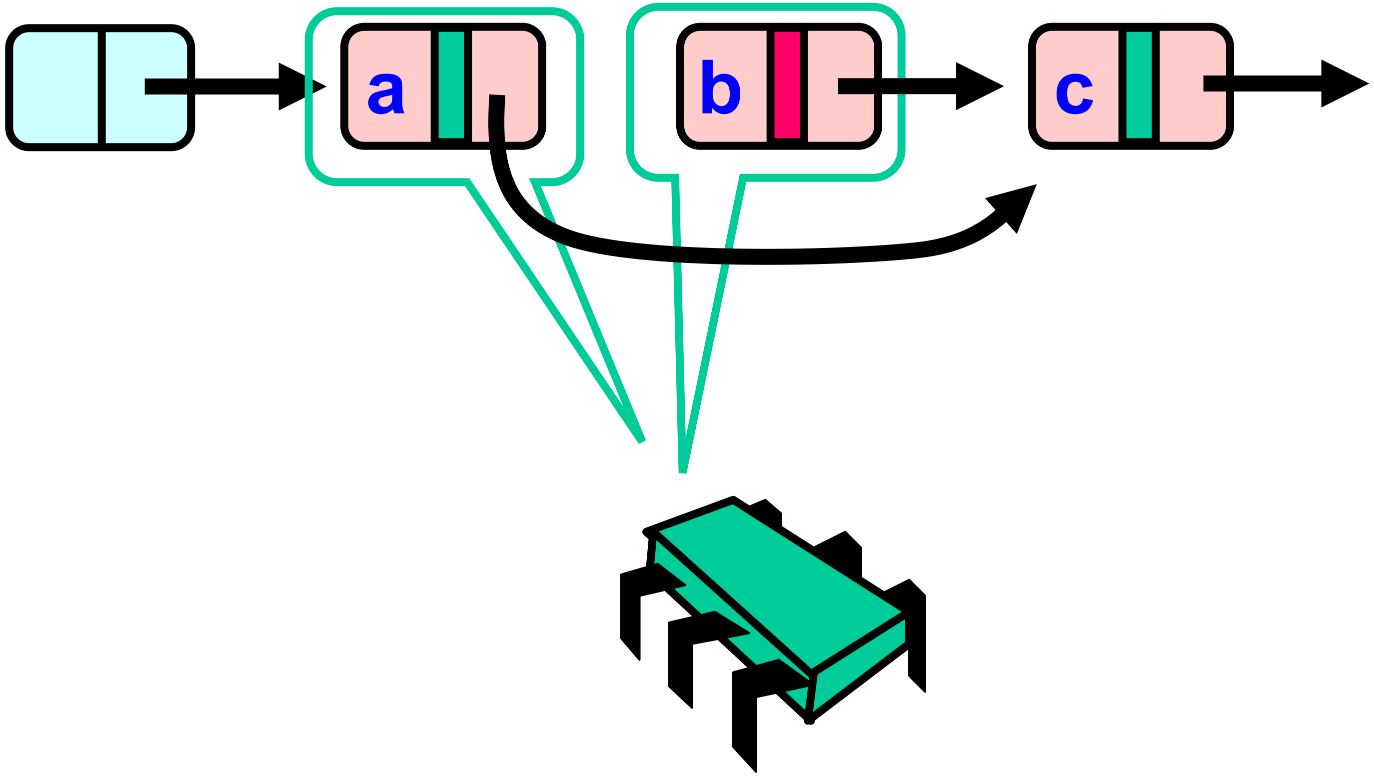
Business as Usual



Business as Usual



Business as Usual



New Abstraction Map

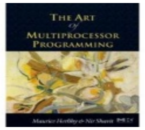
- S(head) =
  - { x | there exists node a such that
    - a reachable from head and
    - a.item = x and
    - a is unmarked
  - }

Invariant

- If not marked then item in the set
- and reachable from head
- and if not yet traversed it is reachable from pred

Validation

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



List Validate Method

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

Predecessor not Logically removed

List Validate Method

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

Current not Logically removed

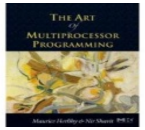
List Validate Method

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

Predecessor still Points to current

Remove

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



Remove

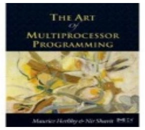
```
try {
  pred.lock(); curr.lock();
  if (validate(pred,curr) {
   if (curr.key == key) {
    curr.marked = true;
    pred.next = curr.next;
    return true;
   } else {
    return false;
   }}} finally {
     pred.unlock();
     curr.unlock();
   }}}
```

Validate as before

Remove

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

Key found



Remove

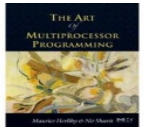
```
try {
  pred.lock(); curr.lock();
  if (validate(pred,curr) {
   if (curr.key == key) {
    curr.marked = true;
    pred.next = curr.next;
    return true;
   } else {
    return false;
   }}} finally {
     pred.unlock();
     curr.unlock();
   }}}
```

Logical remove

Remove

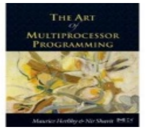
```
try {
  pred.lock(); curr.lock();
  if (validate(pred,curr) {
   if (curr.key == key) {
    curr.marked = true;
    pred.next = curr.next;
    return true;
   } else {
    return false;
   }}} finally {
     pred.unlock();
     curr.unlock();
   }}}
```

physical remove



Contains

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



Contains

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

Start at the head

Contains

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

Search key range

Contains

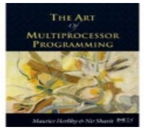
```
public boolean contains(Item item) {
  int key = item.hashCode();
  Node curr = this.head;
  while (curr.key < key) {
    curr = curr.next;
  }
  return curr.key == key && !curr.marked;
}
```

Traverse without locking
(nodes may have been removed)

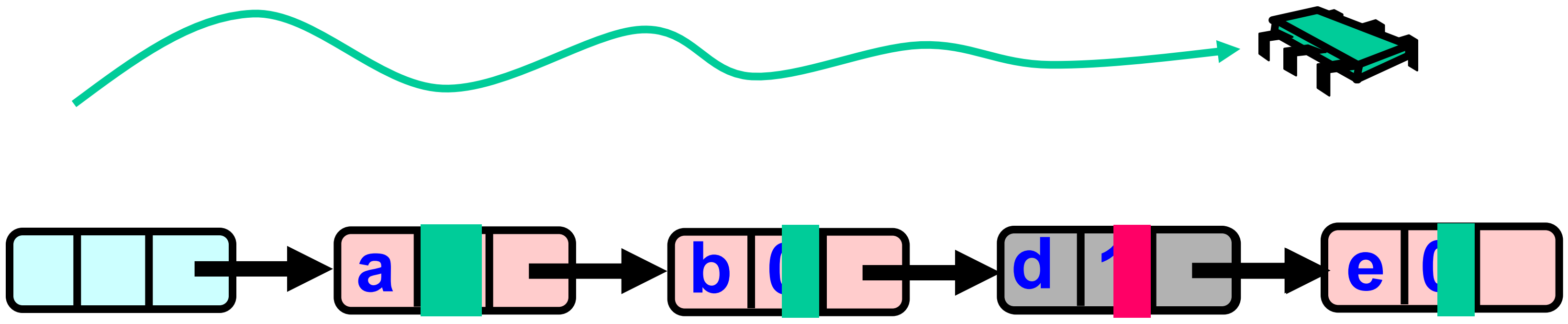
Contains

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

Present and undeleted?



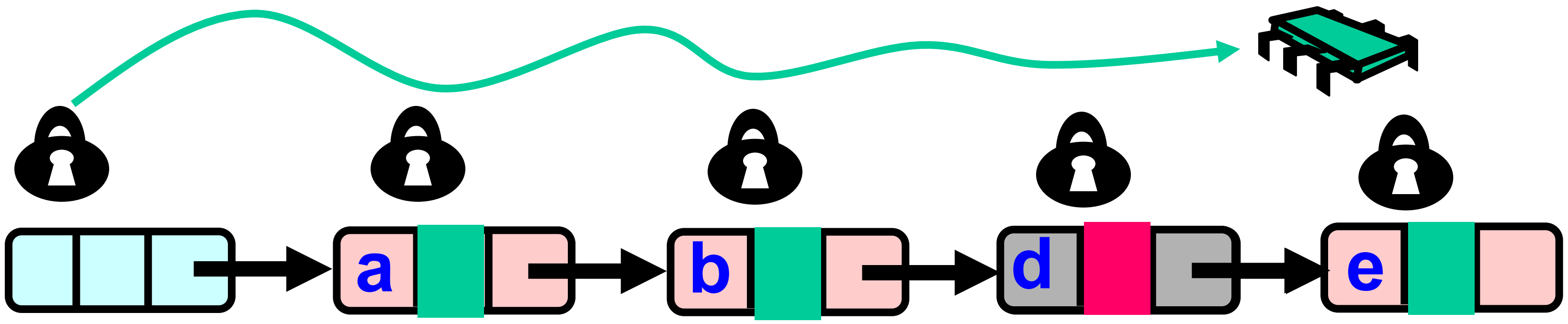
Summary: Wait-free Contains



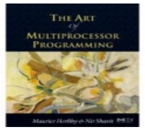
Use Mark bit + list ordering

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

Lazy List

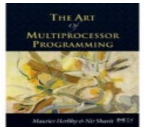


Lazy add() and remove() + Wait-free contains()



Evaluation

- Good:
  - contains() doesn't lock
  - In fact, its wait-free!
  - Good because typically high % contains()
  - Uncontended calls don't re-traverse
- Bad
  - Contended add() and remove() calls do re-traverse
  - Traffic jam if one thread delays



Traffic Jam

- Any concurrent data structure based on mutual exclusion has a weakness
- If one thread
  - Enters critical section
  - And “eats the big muffin”
    - Cache miss, page fault, descheduled …
  - Everyone else using that lock is stuck!
  - Need to trust the scheduler….