Coarse-Grained Synchronization

- Ogni metodo opera mediante un lock sull'oggetto
  - Code di attesa per operare sull'oggetto
  - Nozione di correttezza (linearizability)
  - Modello astratto basato sulla nozione di storia
  - Tecniche statiche (+ model checking)
- E' fatta?

1

Coarse-Grained Synchronization

- Operare con thread
  - Non aumenta necessariamente l'efficienza complessiva di un sistema
  - Attese attive, overhead di gestione, run-time complicato,
- Multiprocessori?
  - Alcune app sono inerentemente parallele … map&reduce come usato da Google

2

Fine-Grained Synchronization

- Non utilizziamo un lock globale …
- Strutturiamo l'oggetto in un insieme di lock
  - Independently-synchronized components
- Metodi sono in conflitto quando cercano di accedere
  - Medesima componente…
  - contemoraneamente

Optimistic Synchronization

- Si cerca la componente richiesta senza usare lock…
- Una volta trovata …
  - OK: e' fatta
  - Oops: riprova
- Valutazione
  - Semplice
  - Errori di programmazione sono costosi

Interface: Set

- Metodi
  - add(x)
  - remove(x)
  - contains(x)

List-Based Sets

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

List Node

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

List Node

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

List Node

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

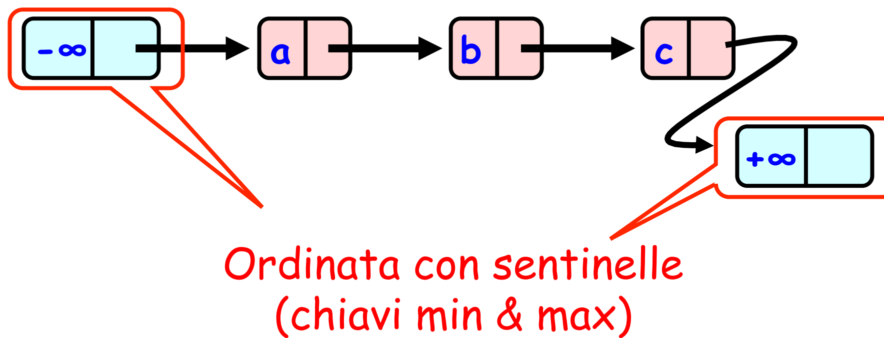
Usually hash code

List Node

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

Reference to next node

List-Based Set

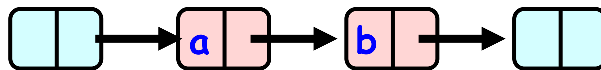


Invariante …

- Preservato da
  - `add()`
  - `remove()`
  - `contains()`
- Solita induzione sulla struttura

Funzione di astrazione

- Rep:



- Astrattamente:
  - {a, b}

Rep Invariant

- Rep invariant
  - Definisce quali sono le rappresentazioni legali "valide"
  - Preservato dai metodi
  - Dipende dai metodi

Rep Invariant

- Sentinelle
- Ordinamento
- Nessun duplicato

Abstraction Map

- S(head) =
  - { x | esiste a tale che
    - a raggoungibile da head e
    - a.item = x
  - }

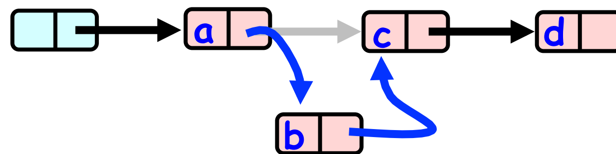
List Based Set

Add(b)

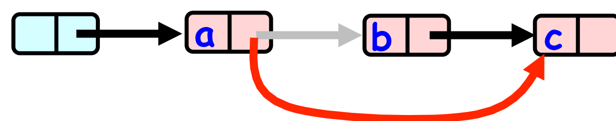
Remove(b)

List Based Set

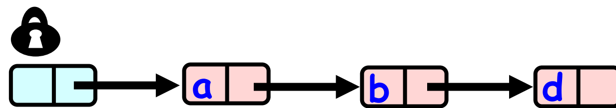
Add(b)



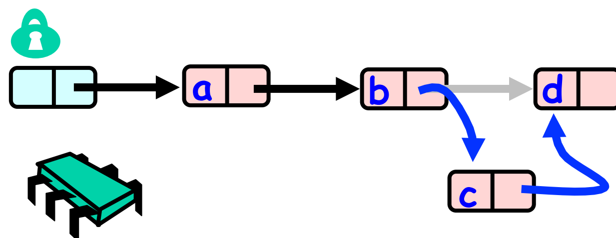
Remove(b)



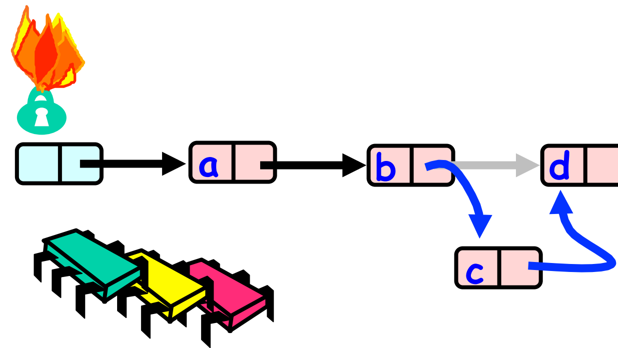
Coarse-Grained Locking



Coarse-Grained Locking



Coarse-Grained Locking



Semplice e costoso

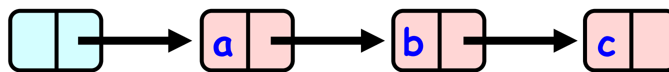
Coarse-Grained Locking

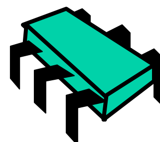
- Metodi sincronizzati (a la Java)
  - "Solo un metodo ha l'accesso …"
- Corretto
- Thread multipli
  - Code di attesa
  - Overhead

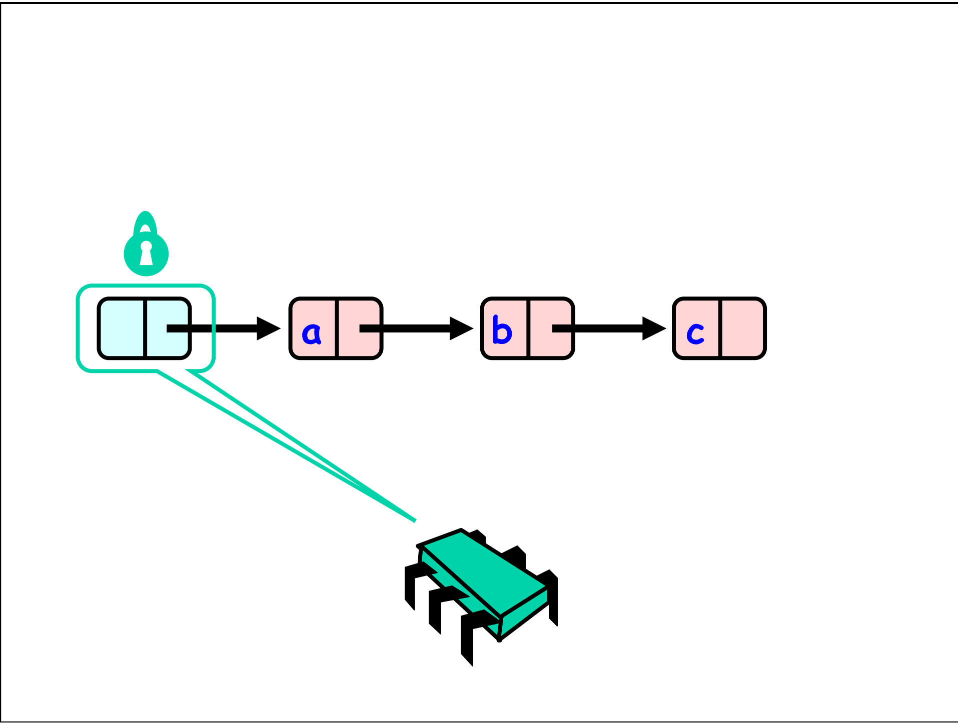
Fine-grained Locking

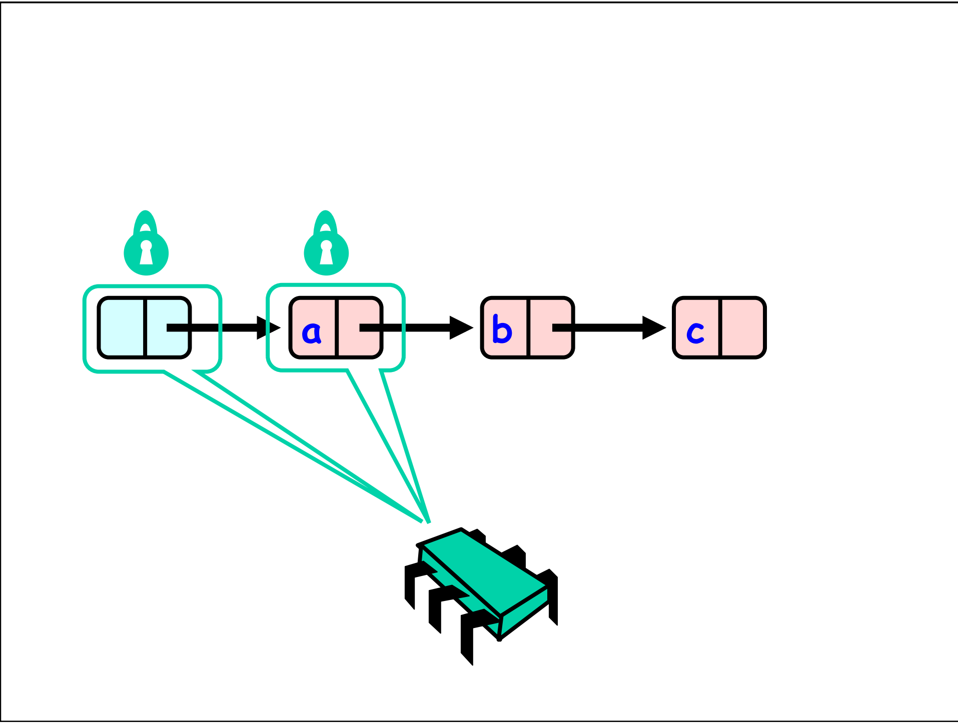
- Pensare prima di programmare
- Suddividere l'oggetto in parti
  - Ogni parte dell'oggetto ha un suo lock
  - Metodi che operano su parti disgiunte possono operare senza problema di sincronizzazione

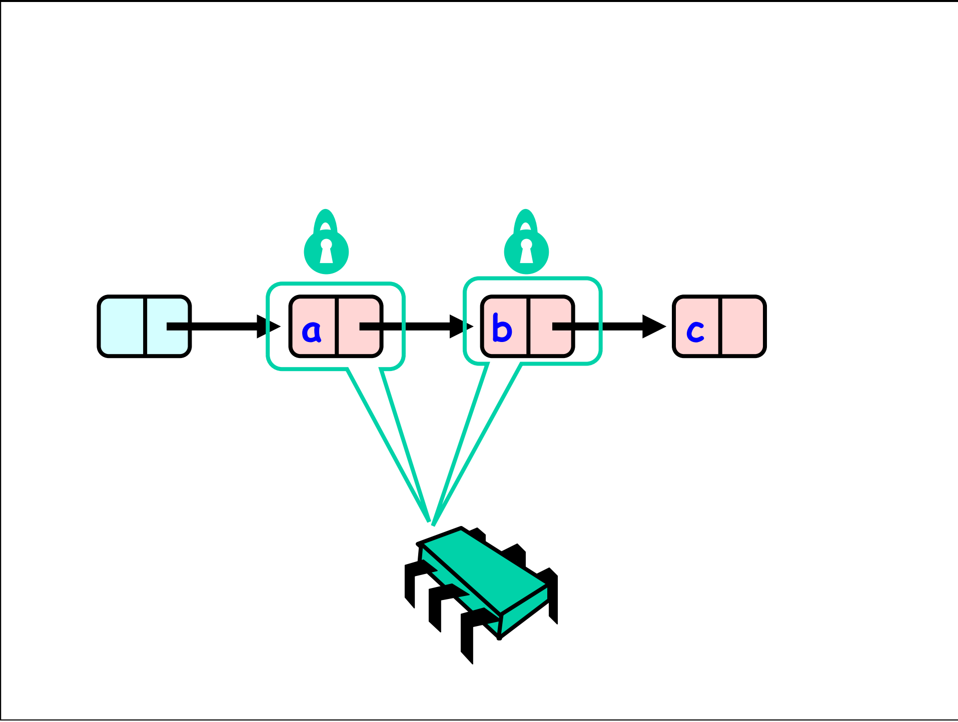
23

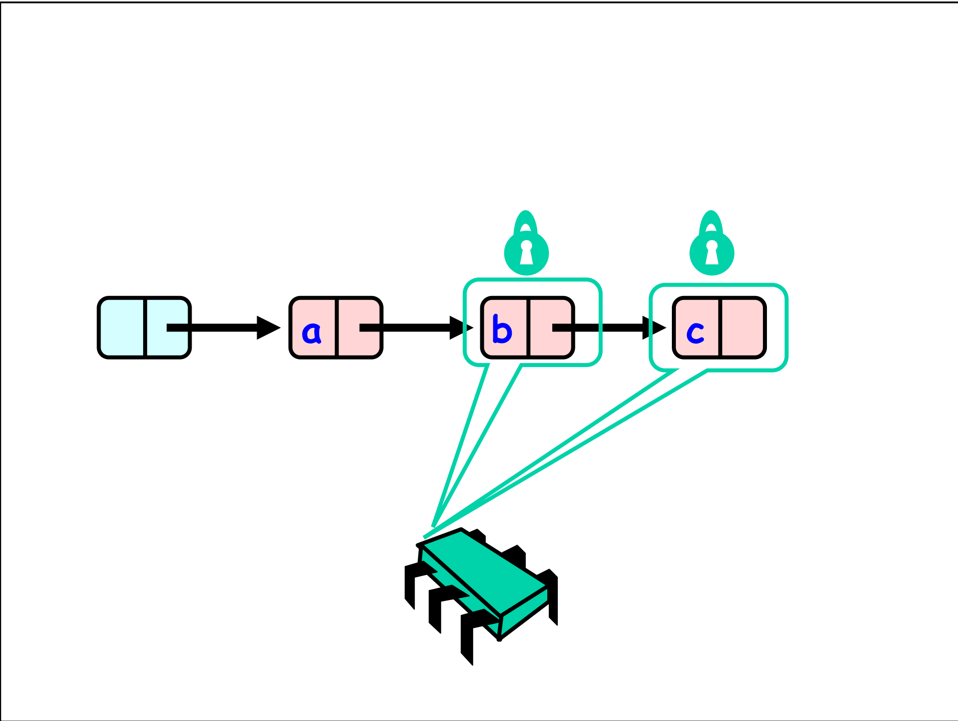




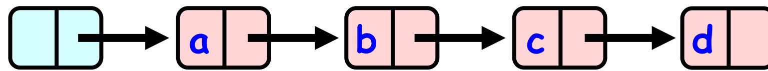

a
b
c

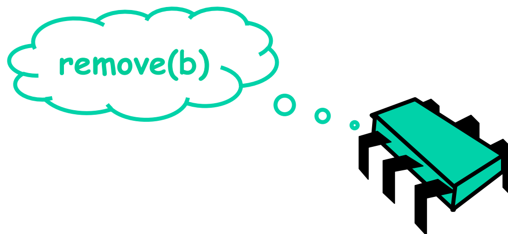

a
b
c


a
b
c

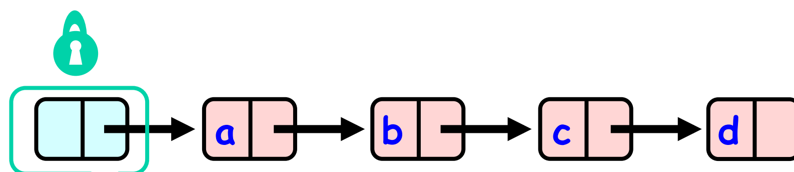

a
b
c

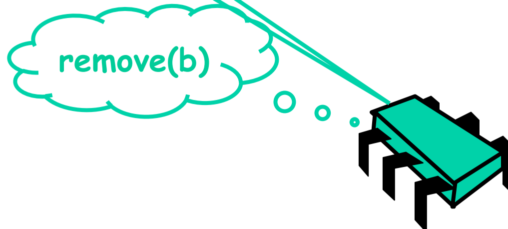
Remove



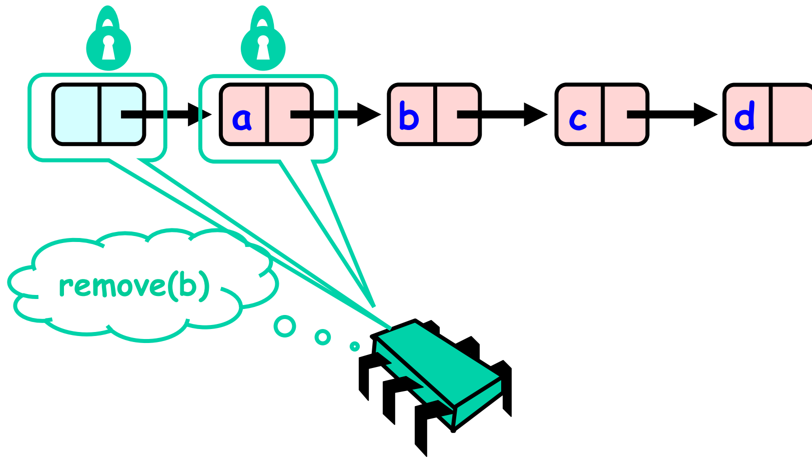


Remove

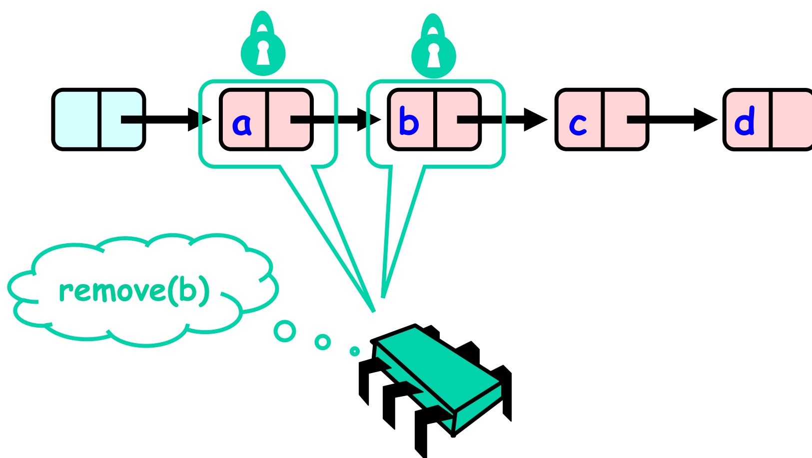




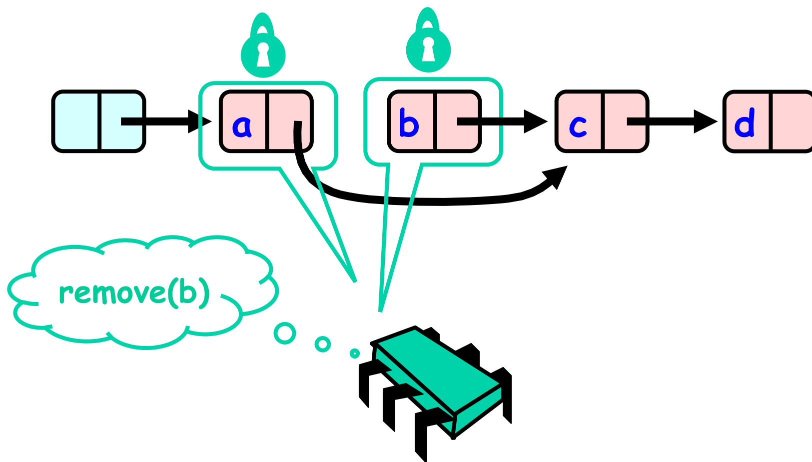
Remove



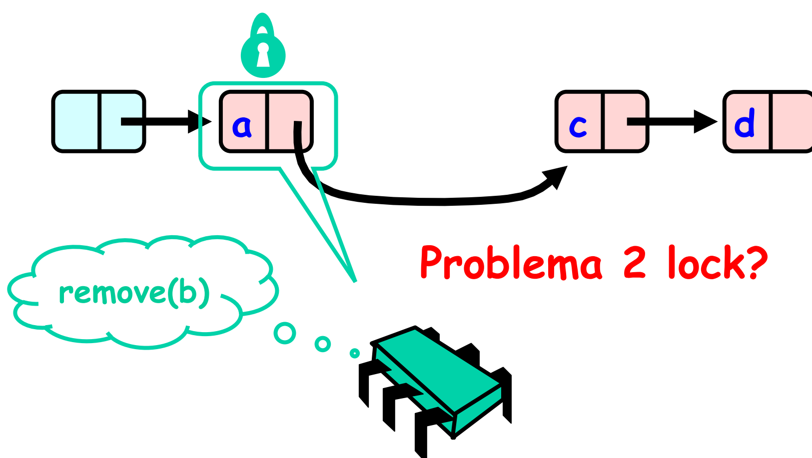
Remove



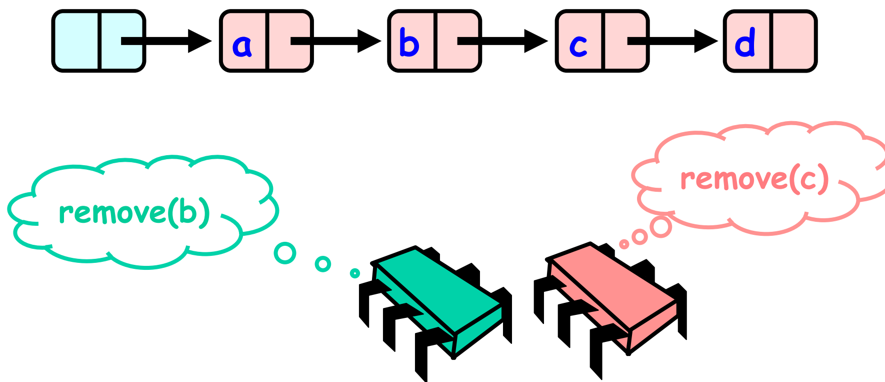
Remove



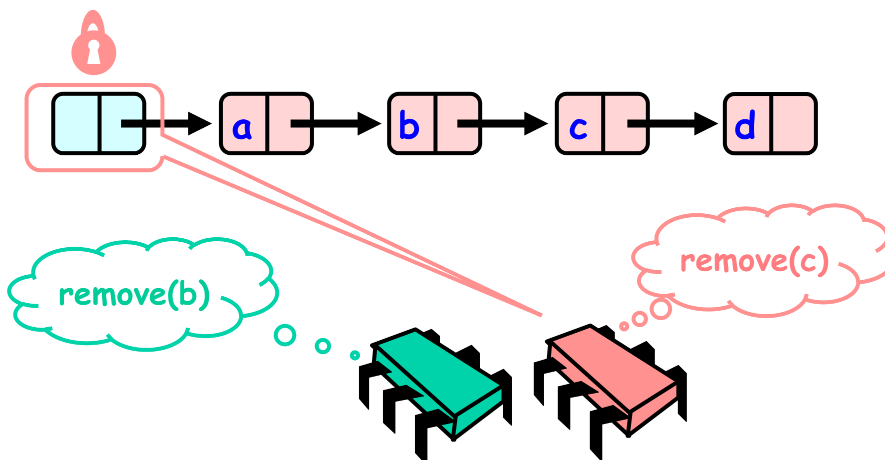
Remove



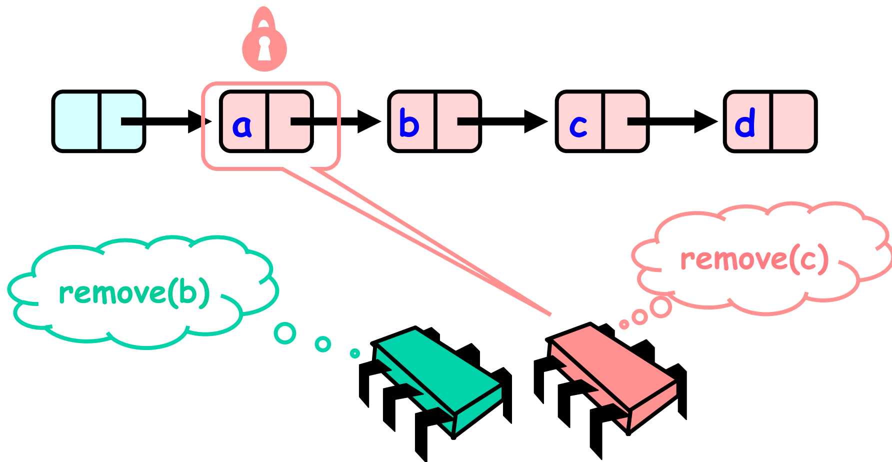
Concurrent Remove



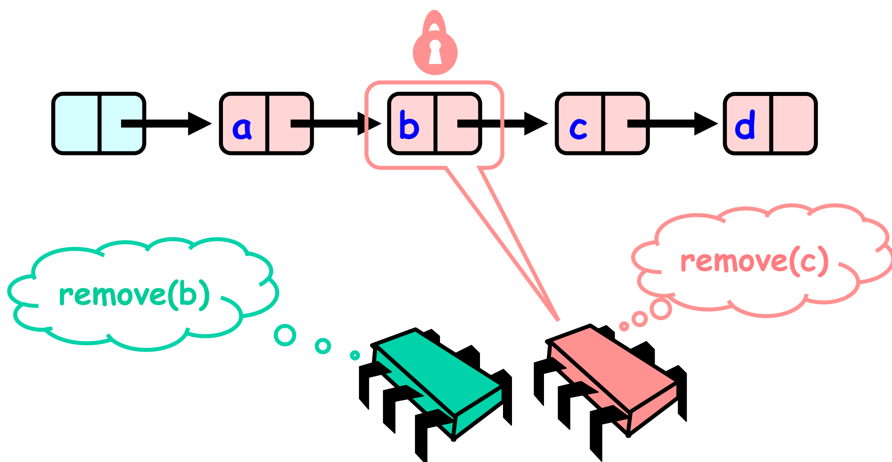
Concurrent Remove



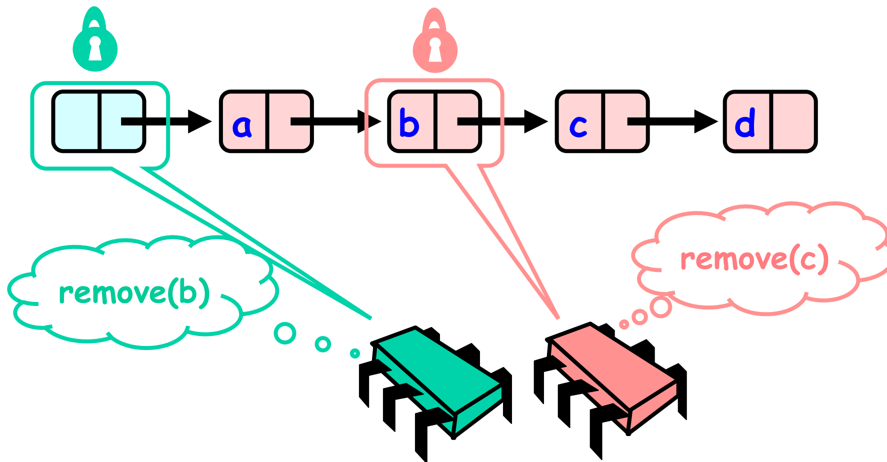
Concurrent Remove



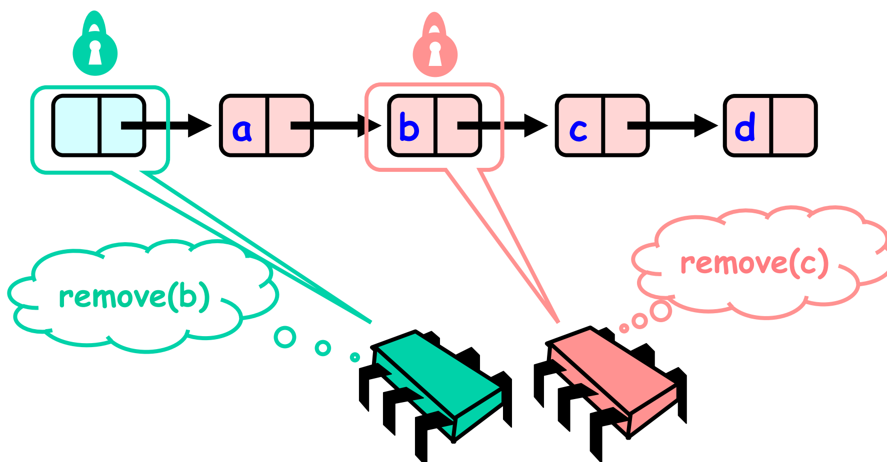
Concurrent Remove



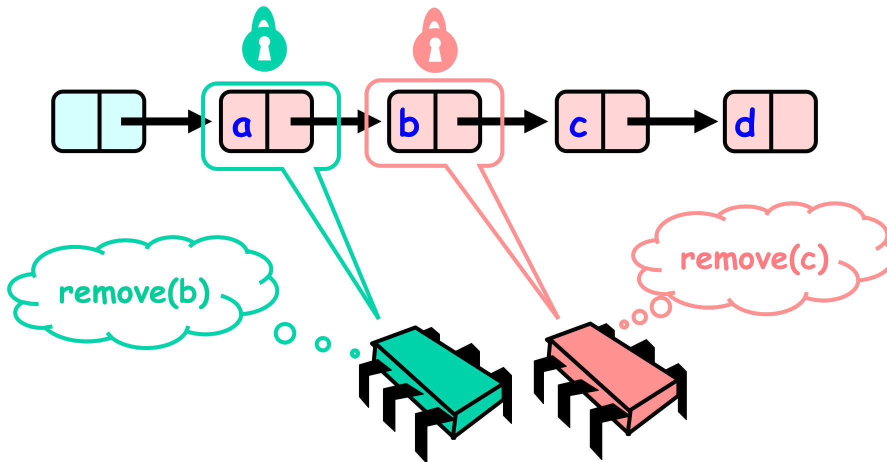
Concurrent Remove



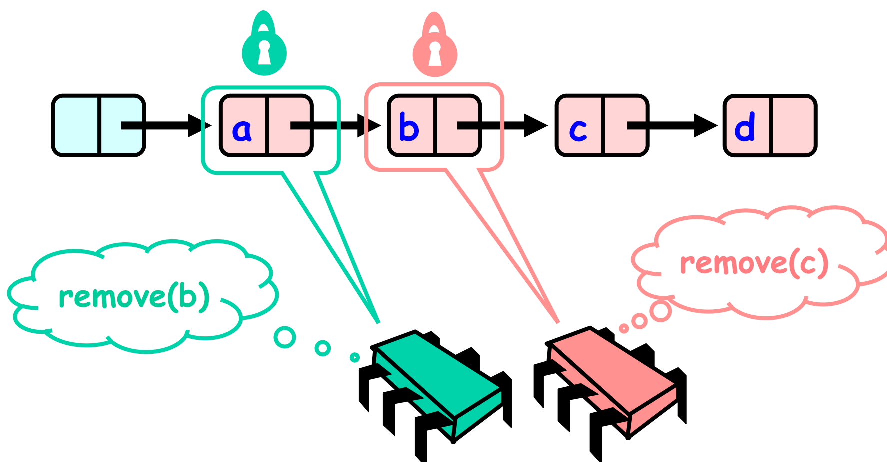
Concurrent Remove



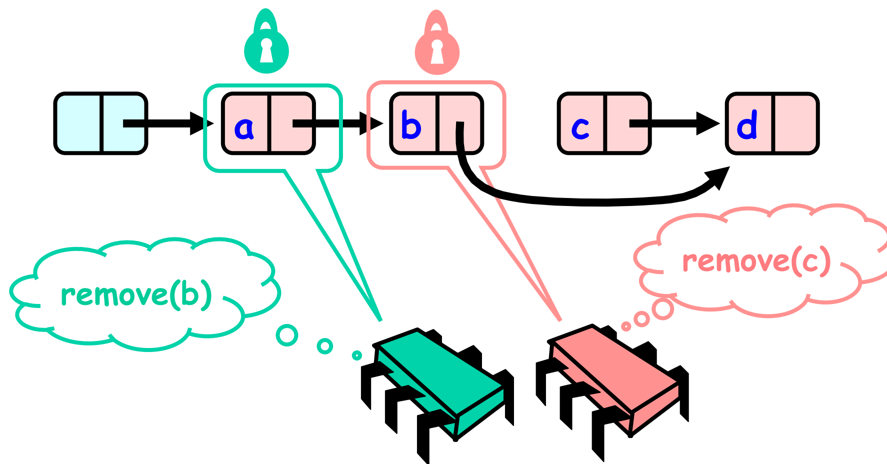
Concurrent Remove



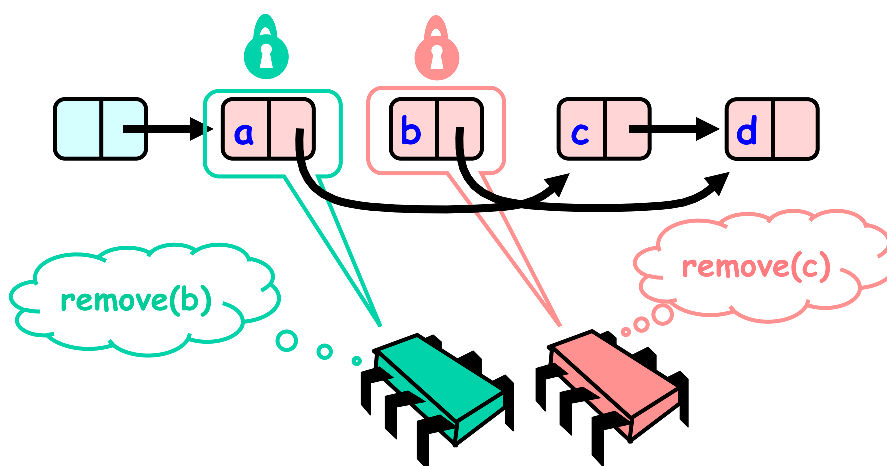
Concurrent Remove

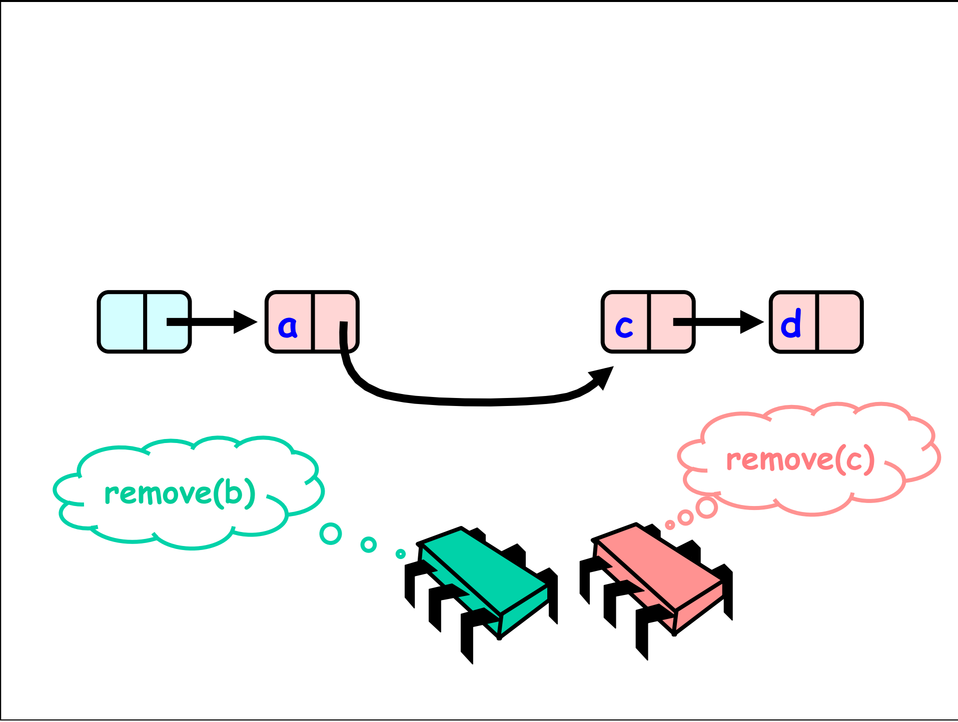


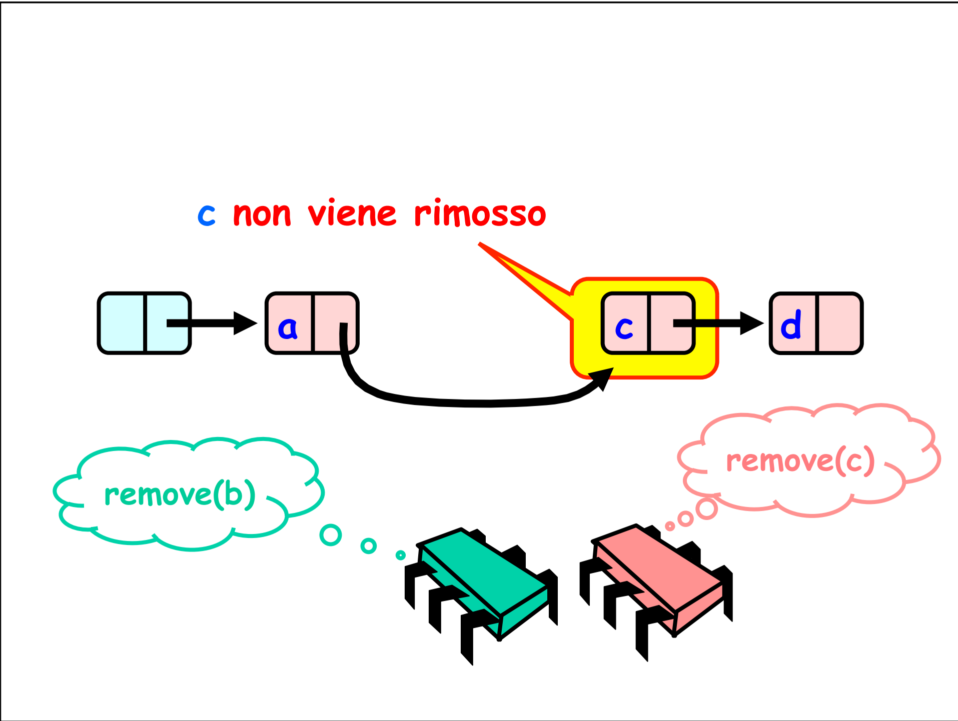
Concurrent Remove



Concurrent Remove

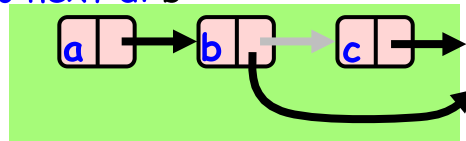


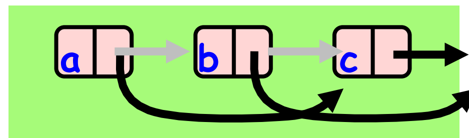

a
c
d
remove(b)
remove(c)


c non viene rimosso
a
c
d
remove(b)
remove(c)

Quale e' il problema?

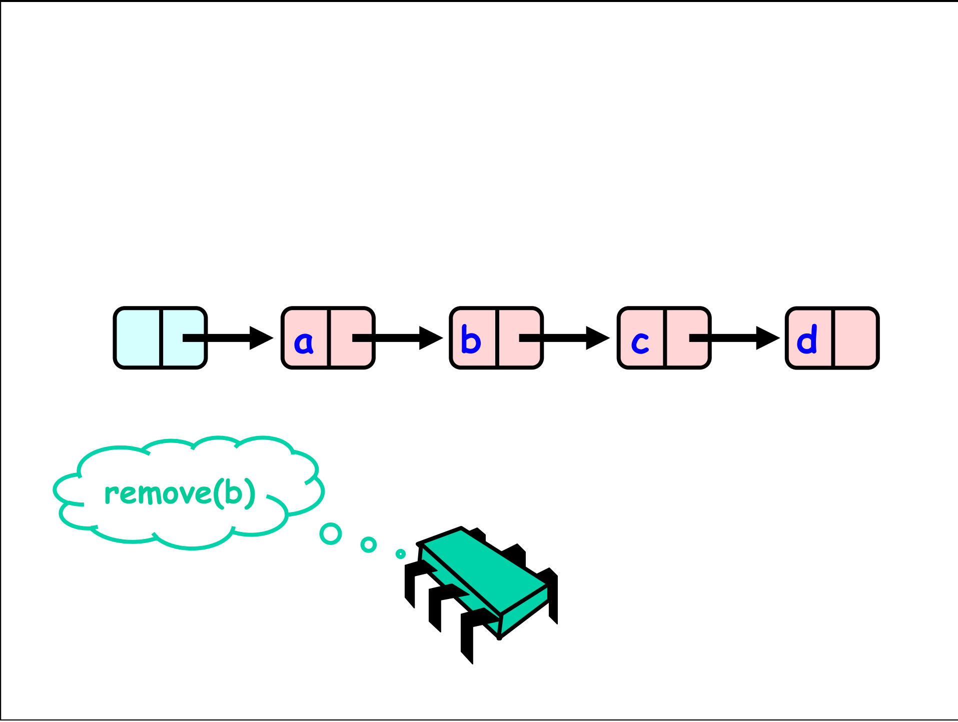
- Rimuovere c
  - Operare sul campo next di b

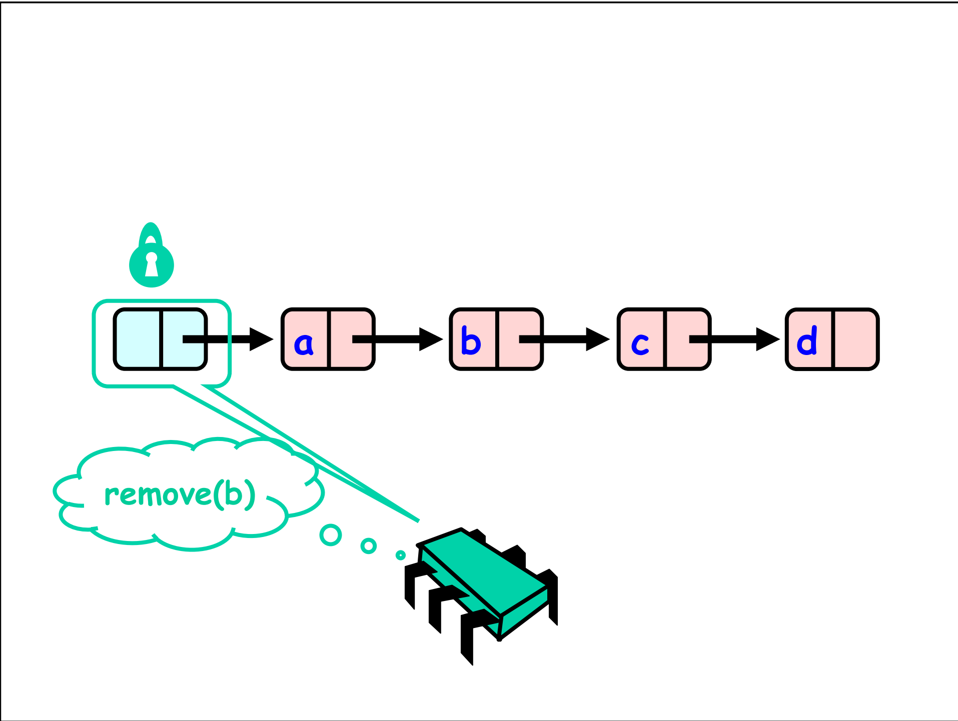


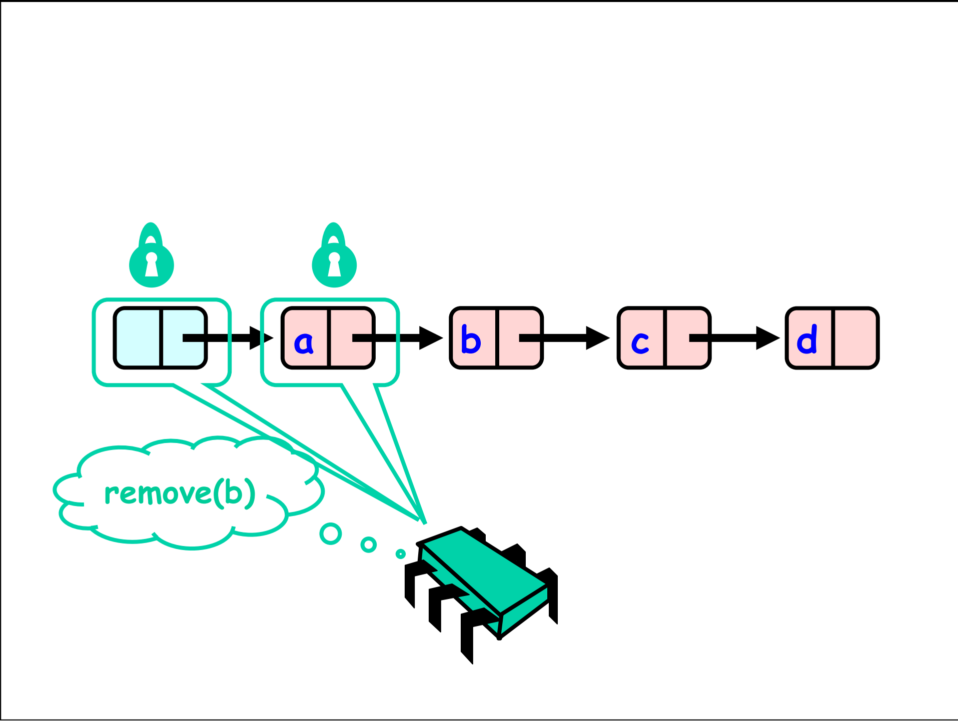


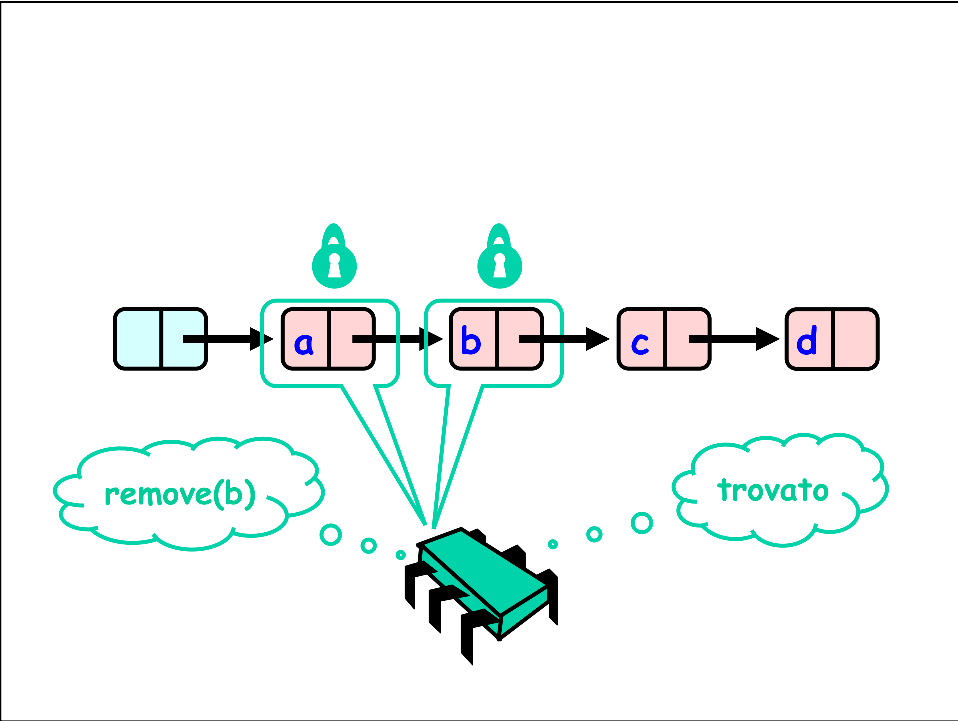
Cerchiamo di capire il problema

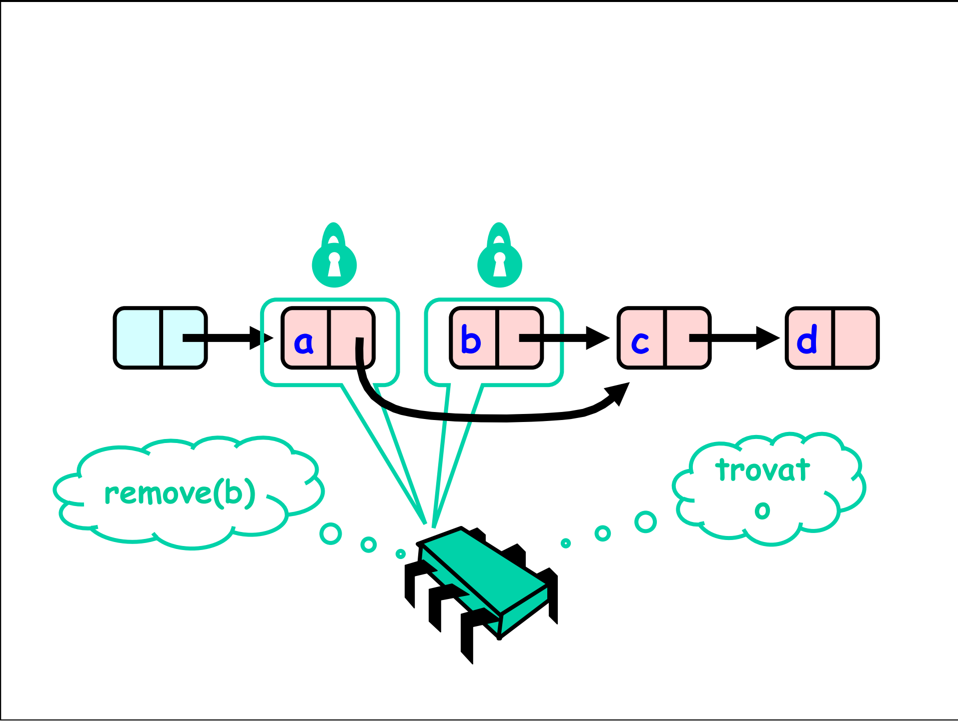
- Se un nodo e' "lock"
  - Nessuno puo cancellare il nodo *successore*
- Morale: il thread deve fare il lock
  - Sul nodo da cancellare
  - E sul predecessore

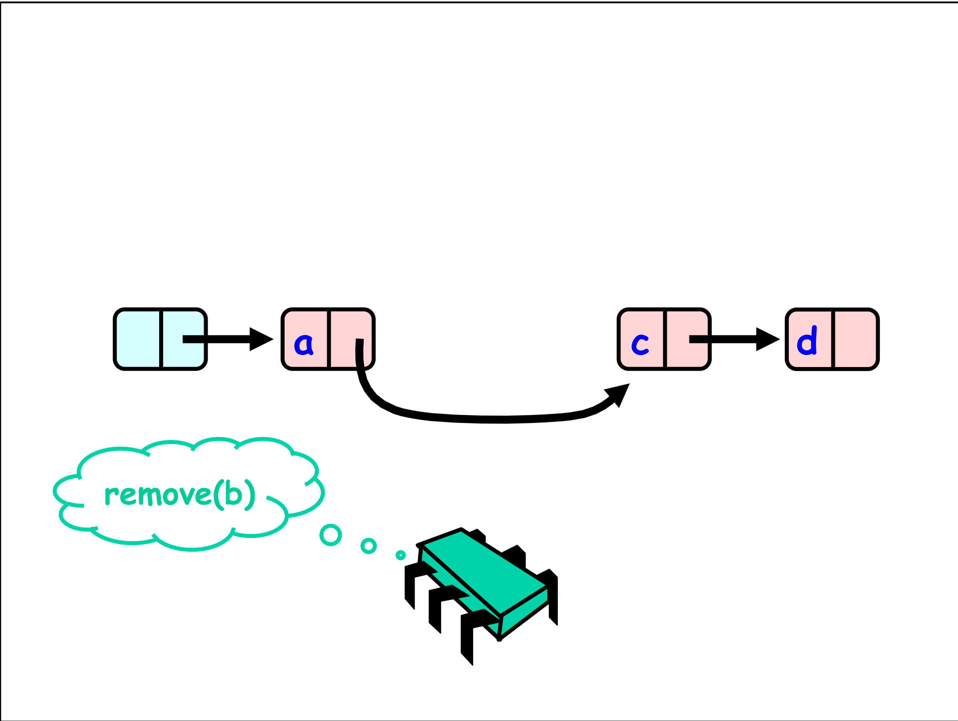

a
b
c
d
remove(b)

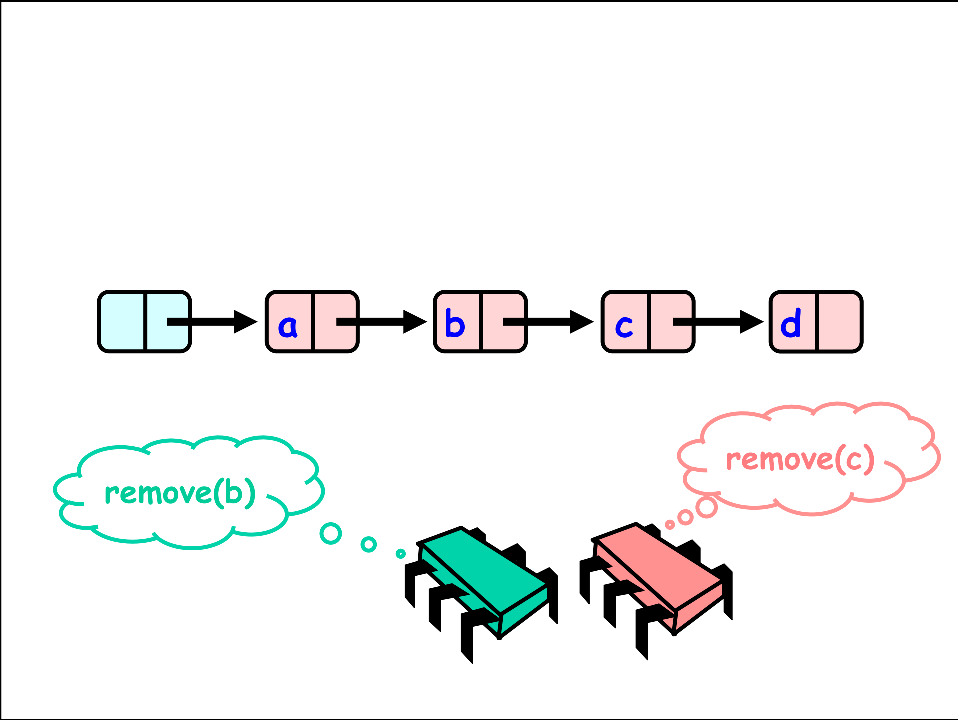

a
b
c
d
remove(b)

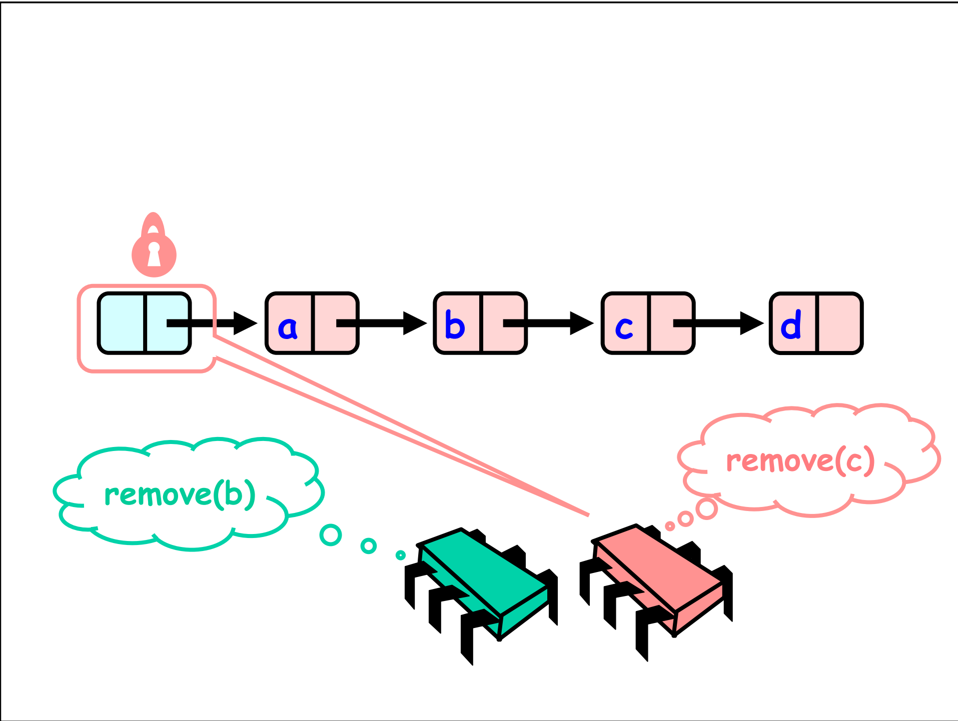

a
b
c
d
remove(b)

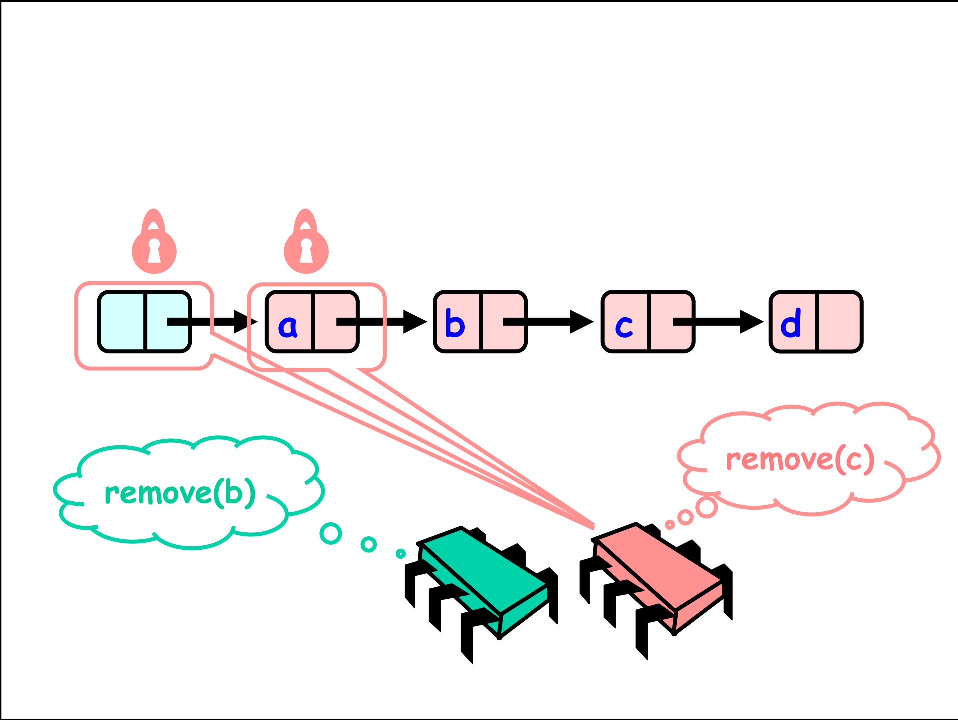

a
b
c
d
remove(b)
trovato

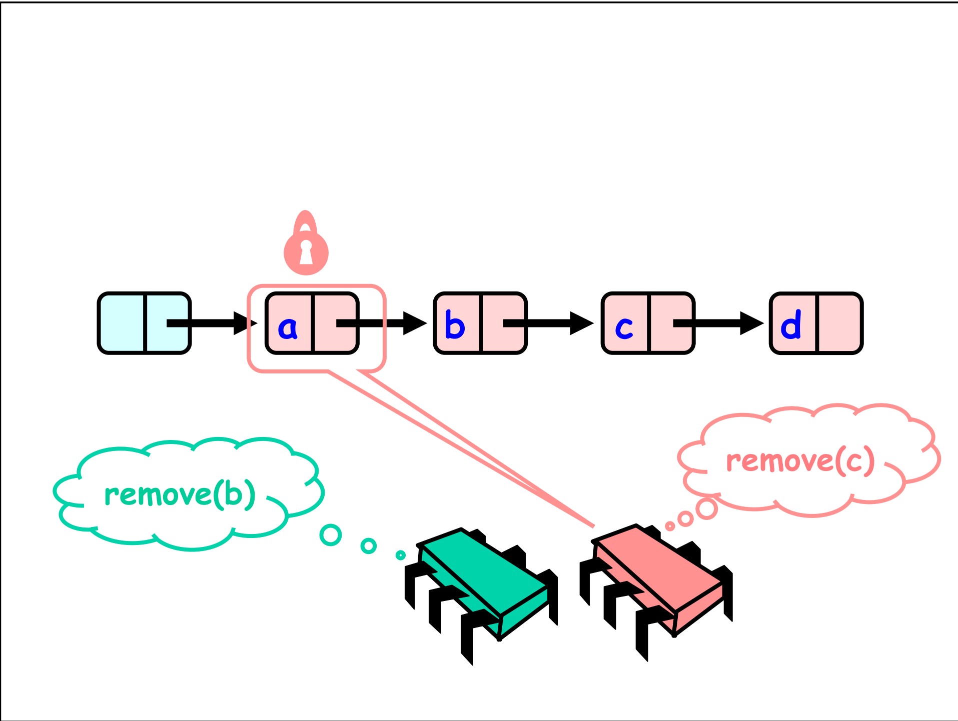

a
b
c
d
remove(b)
trovat
o

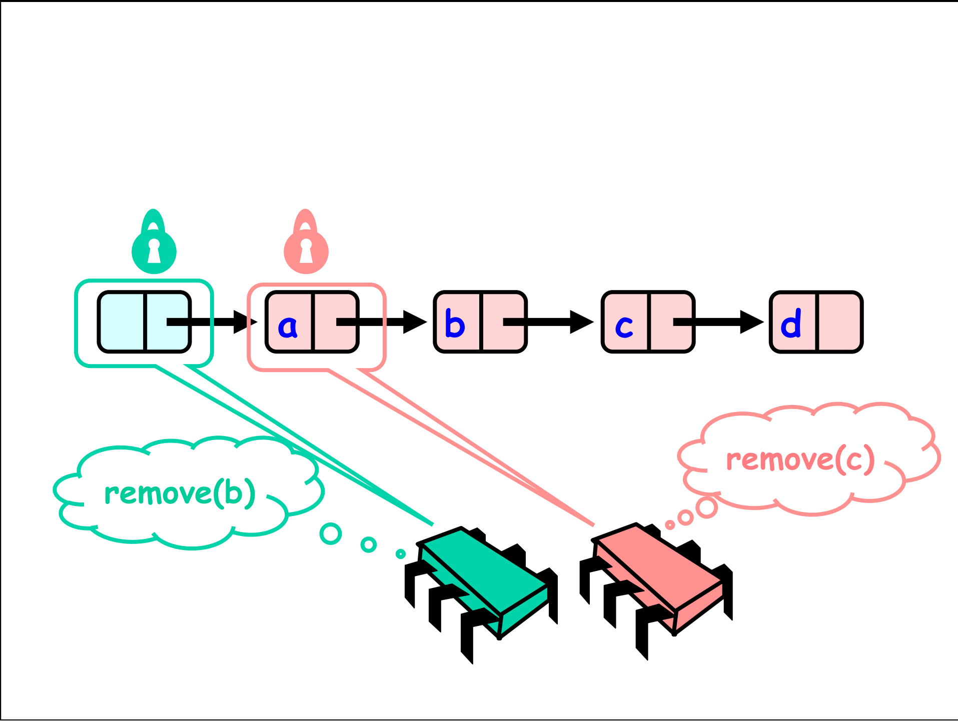

a
c
d
remove(b)

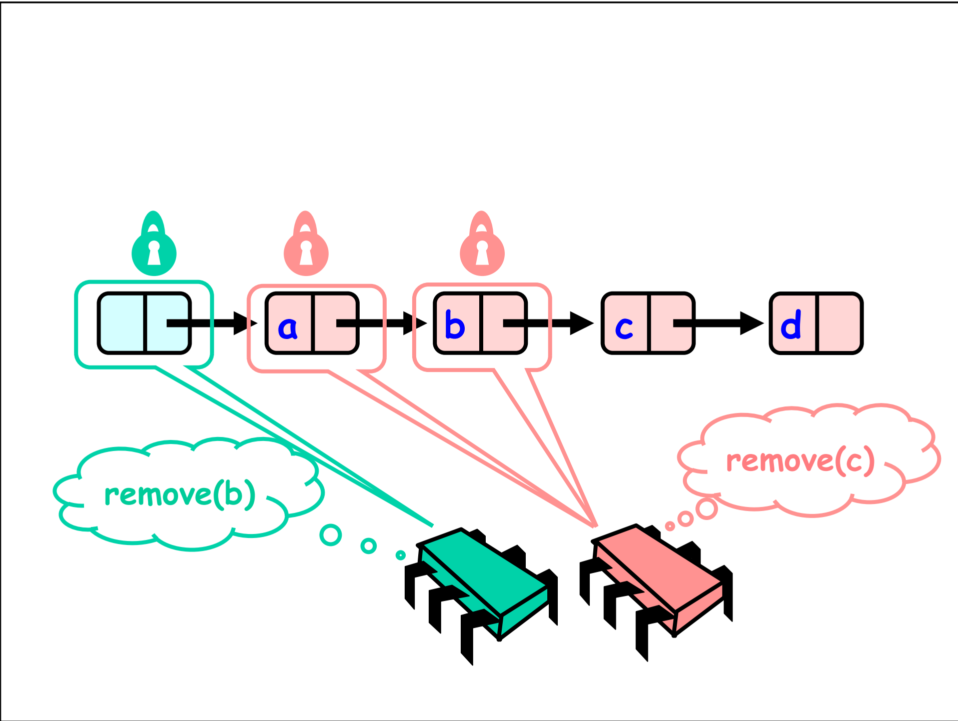

a
b
c
d
remove(b)
remove(c)

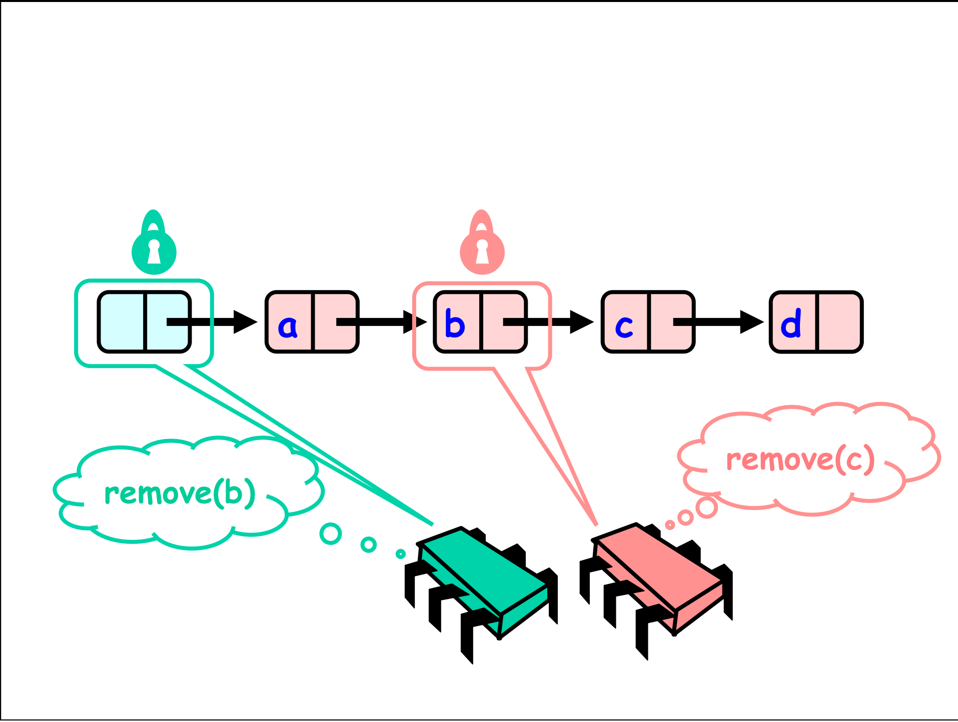

a
b
c
d
remove(b)
remove(c)

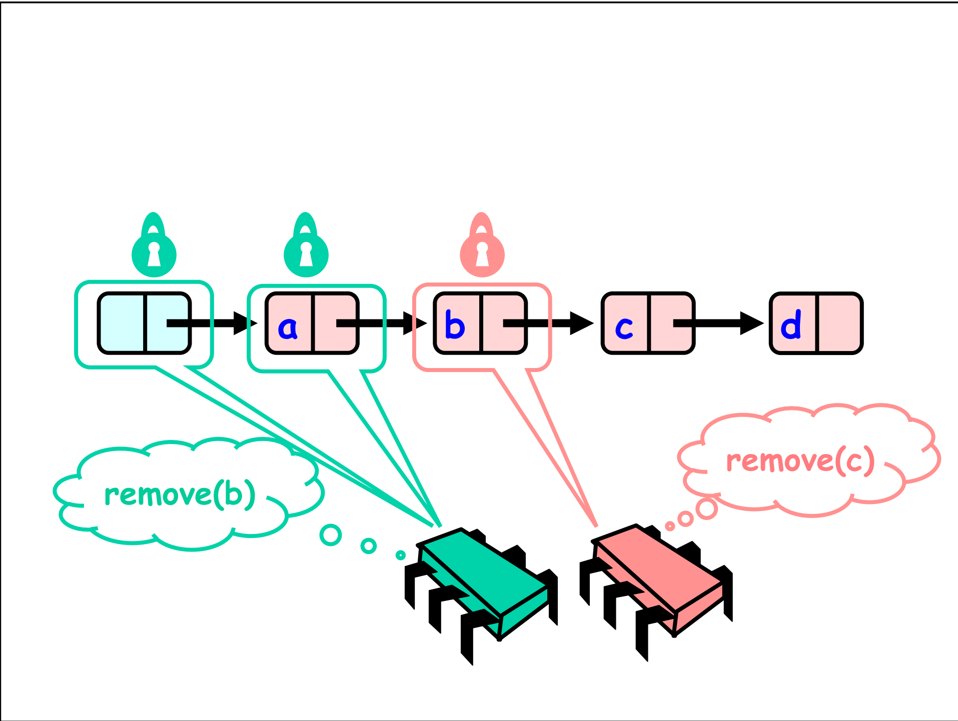

a
b
c
d
remove(b)
remove(c)

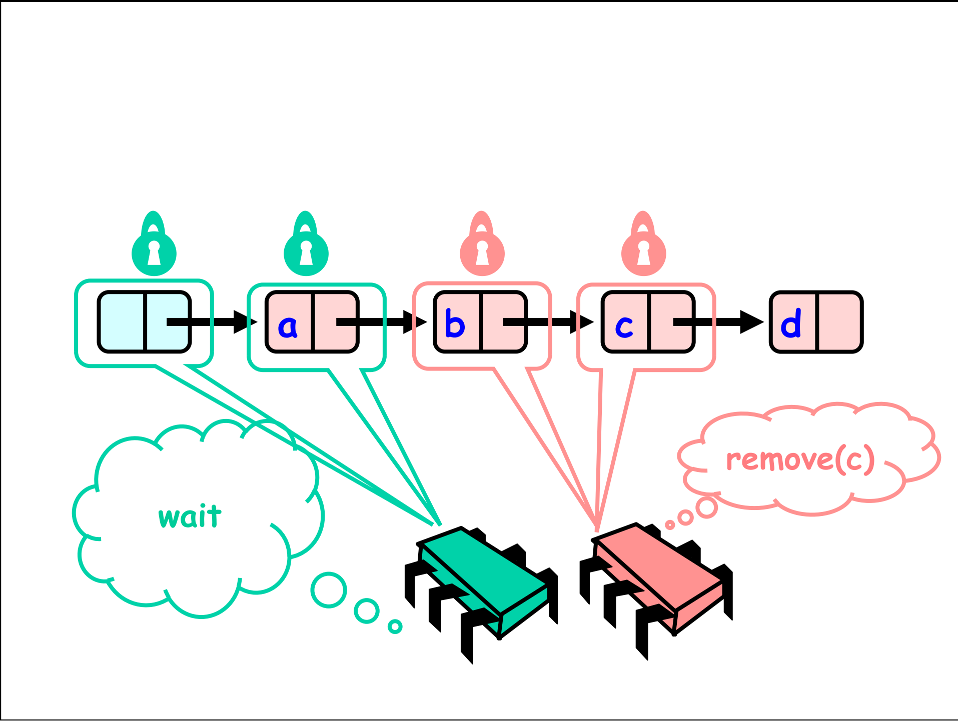

a
b
c
d
remove(b)
remove(c)

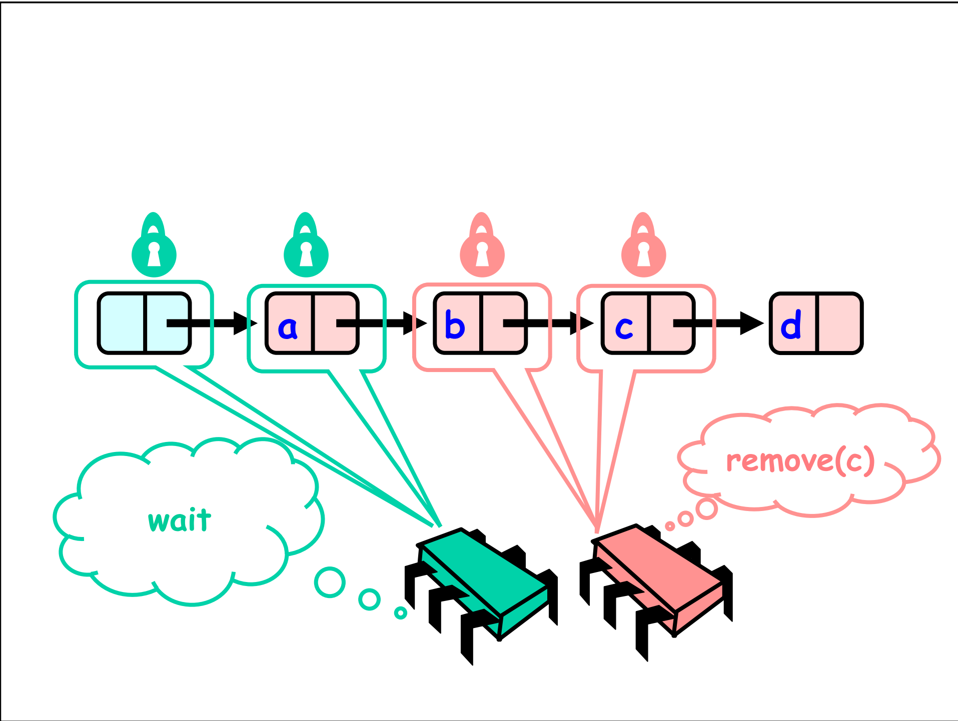

a
b
c
d
remove(b)
remove(c)

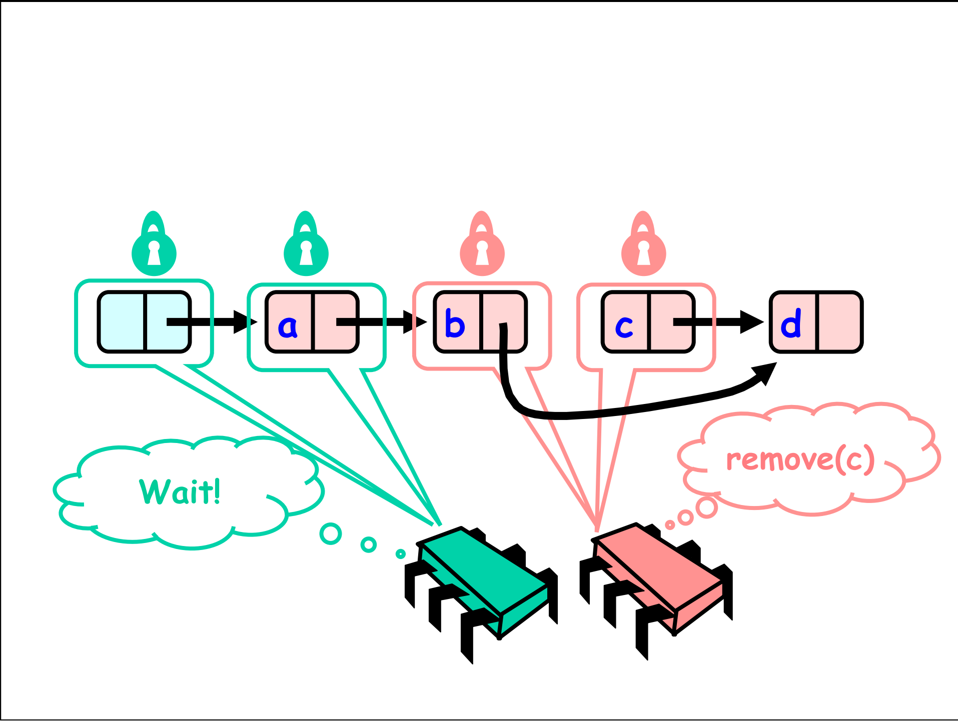

a
b
c
d
remove(b)
remove(c)

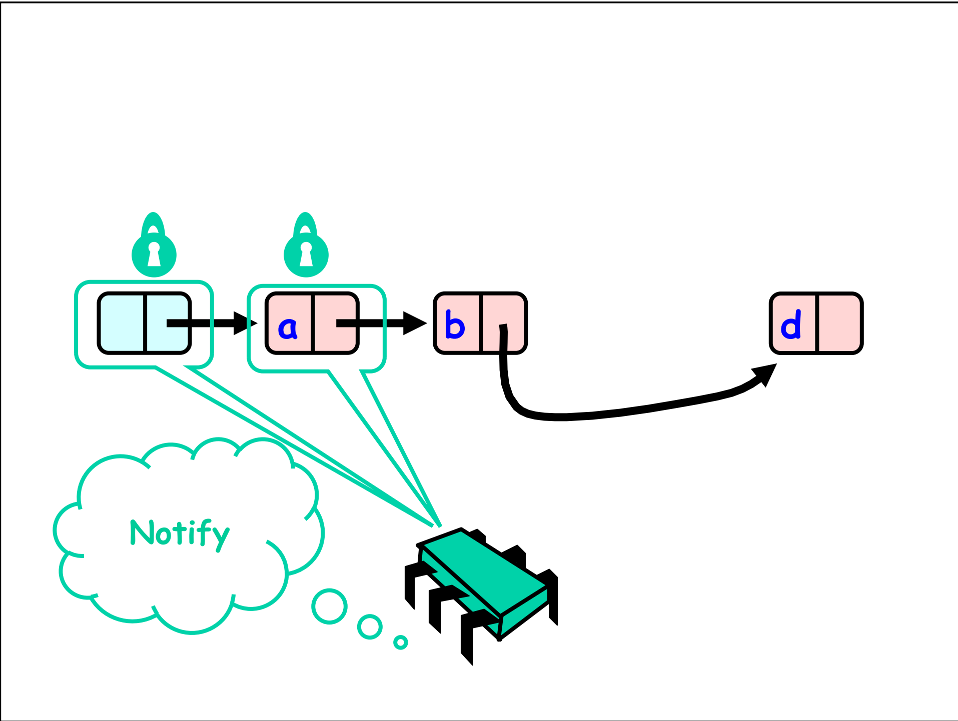

a
b
c
d
remove(b)
remove(c)

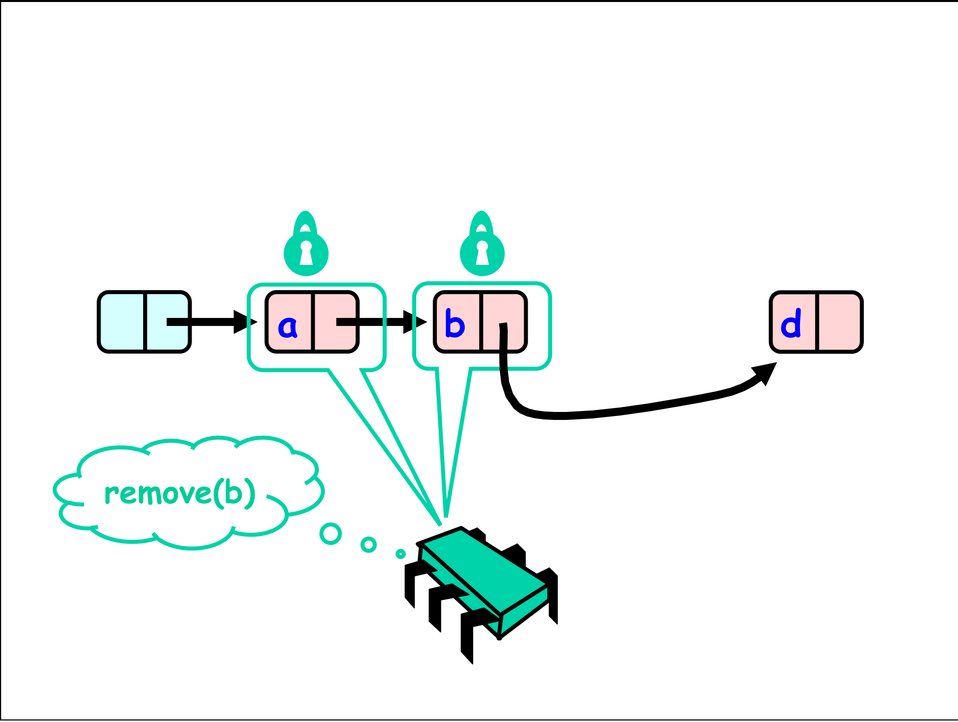

a
b
c
d
remove(b)
remove(c)

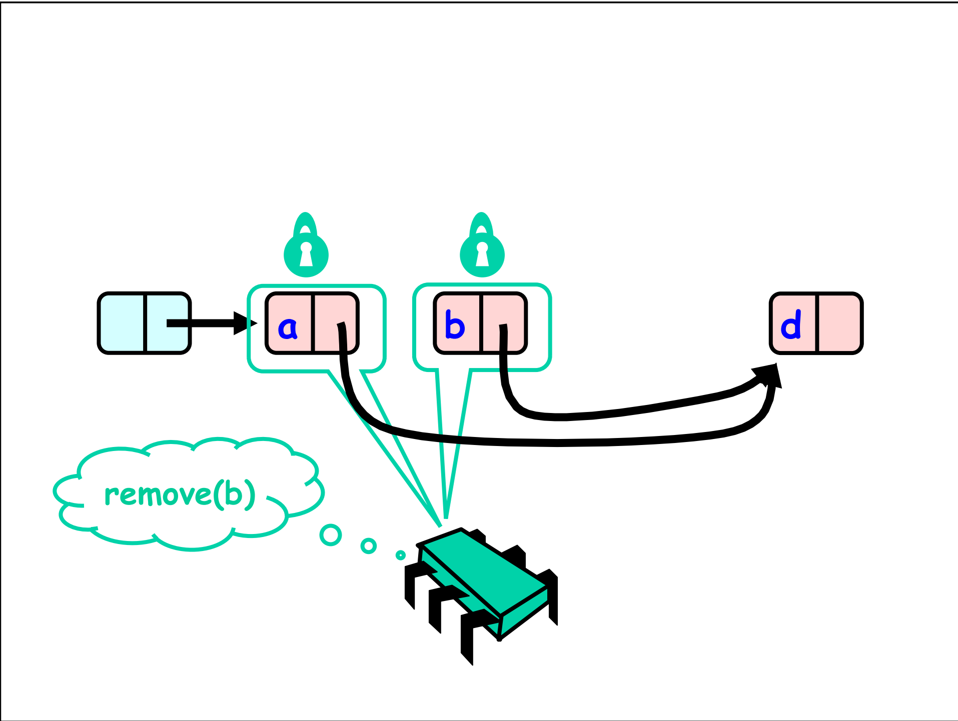

a
b
c
d
remove(c)
wait

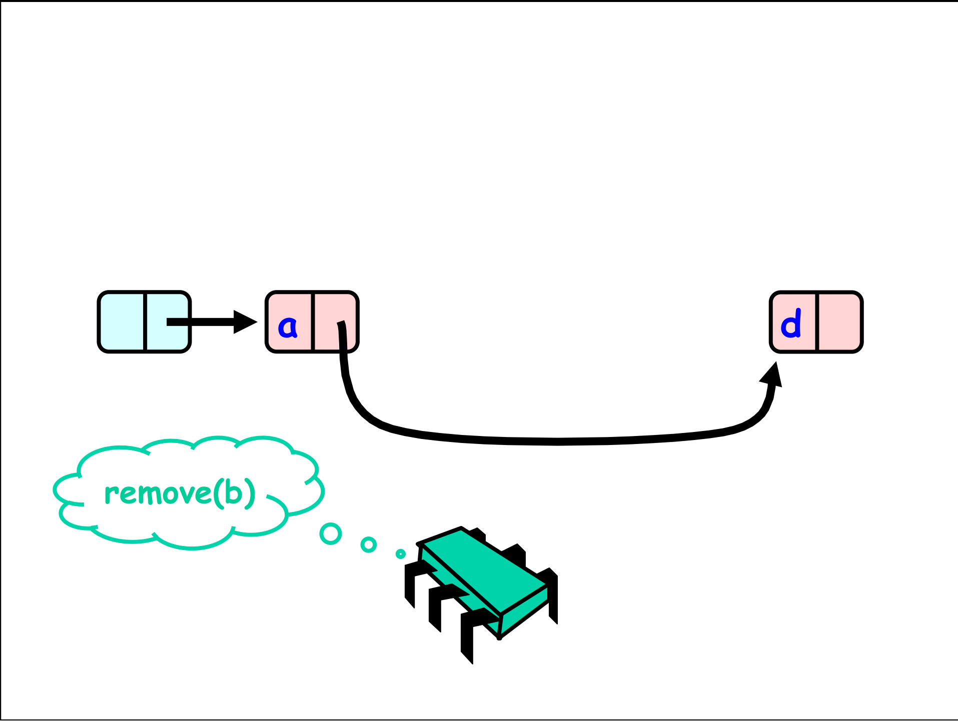

a
b
c
d
remove(c)
wait

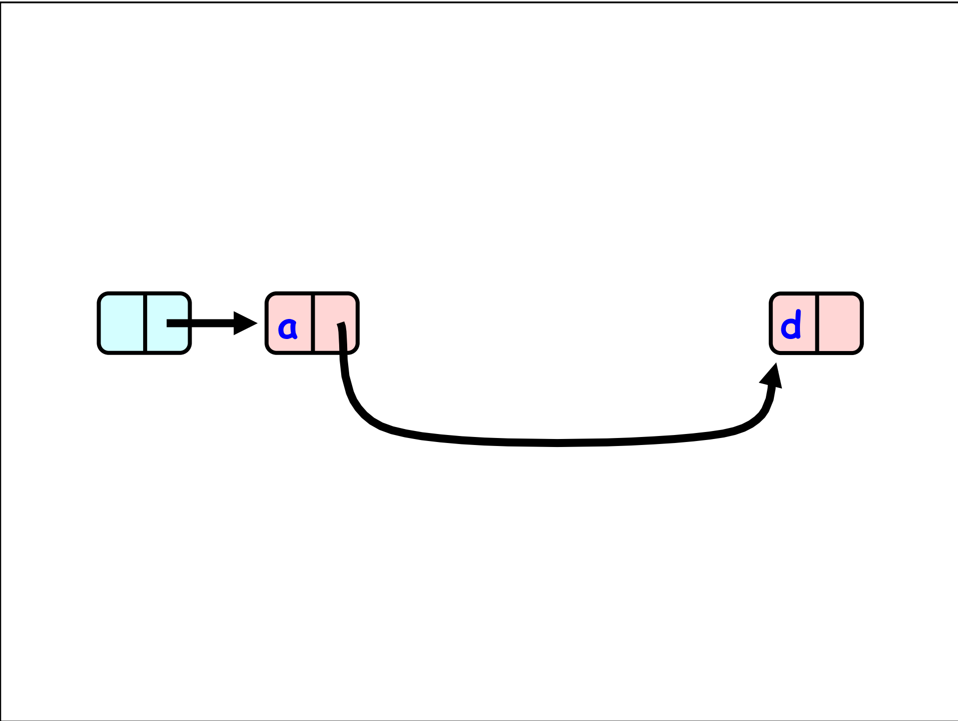

a
b
c
d
remove(c)
Wait!


a
b
d
Notify


a
b
d
remove(b)


a
b
d
remove(b)


a
d
remove(b)


a
d

Remove method

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

Remove method

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

Key used to order node

Remove method

```
public boolean remove(Item item) {
 int key = item.hashCode();
 Node pred, curr;
 try {
   …
 } finally {
  currNode.unlock();
  predNode.unlock();
 }}
```

Predecessor and current nodes

Remove method

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

Make sure locks released

Remove method

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

Everything else

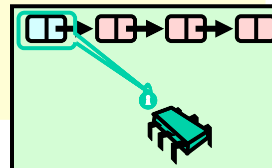
Remove method

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

Remove method

lock pred == head

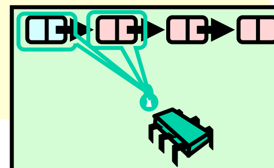
```
try {
  pred = this.head;
  pred.lock();
  curr = pred.next;
  curr.lock();
  …
} finally { … }
```



Remove method

Lock current

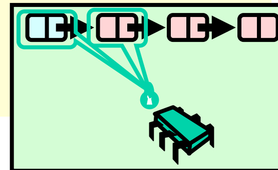
```
try {
  pred = this.head;
  pred.lock();
  curr = pred.next;
  curr.lock();
  …
} finally { … }
```



Remove method

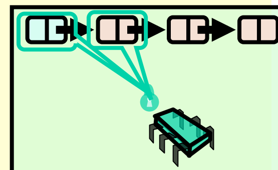
```
try {
 pred = this.head;
 pred.lock();
 curr = pred.next;
 curr.lock();
 …
} finally { … }
```

Traversing list



Remove: searching

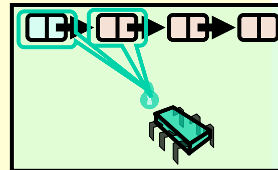
```
while (curr.key <= key) {
  if (item == curr.item) {
   pred.next = curr.next;
   return true;
  }
  pred.unlock();
  pred = curr;
  curr = curr.next;
  curr.lock();
 }
 return false;
```



Remove: searching

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

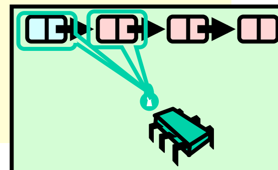
Search key range



Remove: searching

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

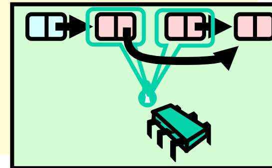
At start of each loop: curr and pred locked



Remove: searching

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

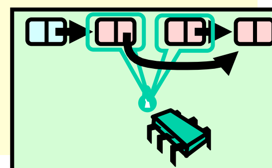
If item found, remove node



Remove: searching

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

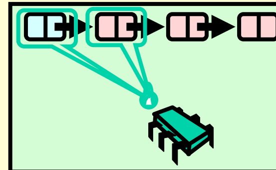
If node found, remove it



Remove: searching

Unlock predecessor

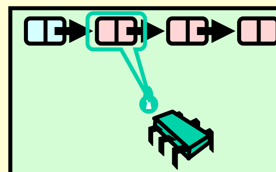
```
while (curr.key <= key) {
  if (item == curr.item) {
   pred.next = curr.next;
   return true;
  }
  pred.unlock();
  pred = curr;
  curr = curr.next;
  curr.lock();
 }
 return false;
```



Remove: searching

Only one node locked!

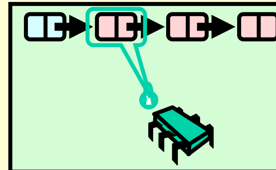
```
while (curr.key <= key) {
  if (item == curr.item) {
   pred.next = curr.next;
   return true;
  }
  pred.unlock();
  pred = curr;
  curr = curr.next;
  curr.lock();
 }
 return false;
```



Remove: searching

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

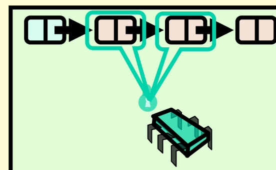
demote current



Remove: searching

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

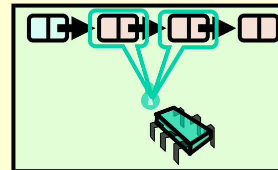
Find and lock new current



Remove: searching

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

Lock invariant restored



Remove: searching

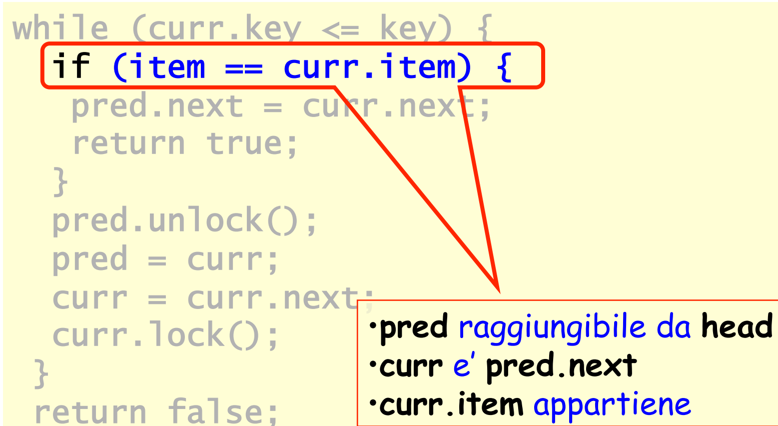
```
while (curr.key <= key) {
  if (item == curr.item) {
   pred.next = curr.next;
   return true;
  }
  pred.unlock();
  pred = curr;
  curr = curr.next;
  curr.lock();
 }
 return false;
```

Otherwise, not present

Why does this work?

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

remove() e' linearizable



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

Linearization point if item is present

Why remove() is linearizable

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

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

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

Item not present

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

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

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

Linearization point

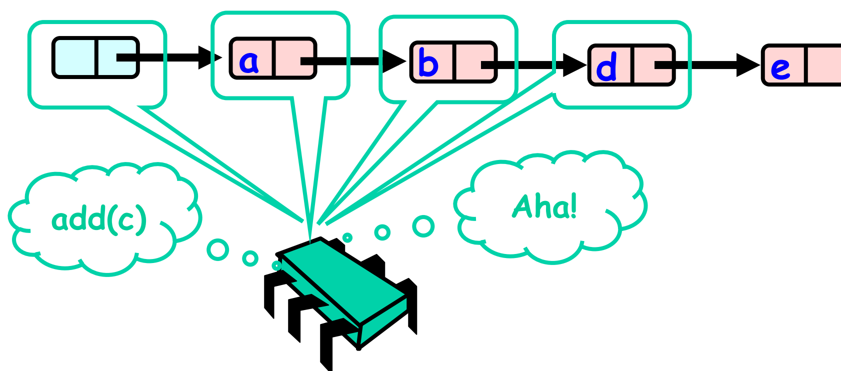
Inserzione

- Add e
  - lock predecessor
  - lock successor

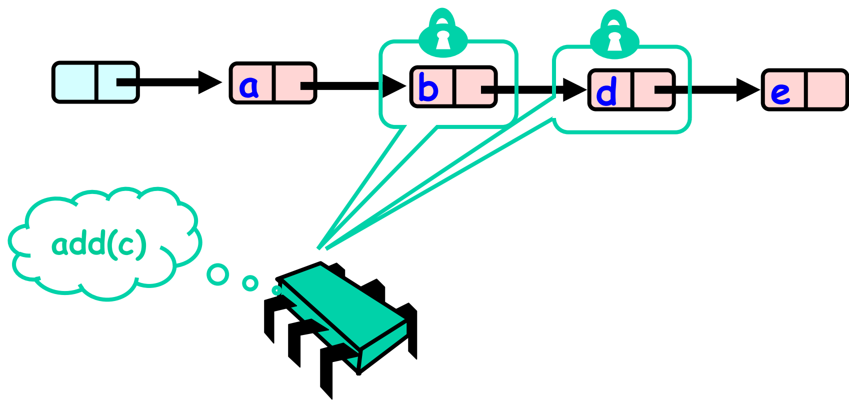
Tutto bene?

- Sicuramente meglio del lock globale
  - Thread possono scorrere in parallelo
- Comportamento ideale?
  - Catena di lock/unlock
  - Potrebbe non essere efficiente

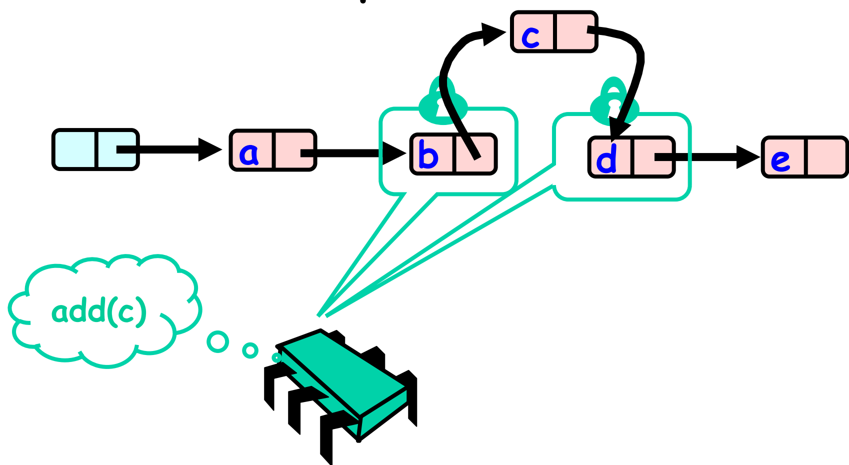
Optimistic: scorrere senza fare Lock



Optimistic

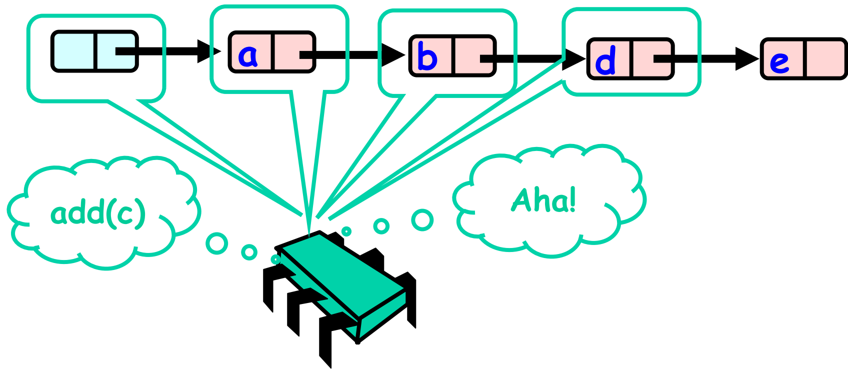


Optimistic

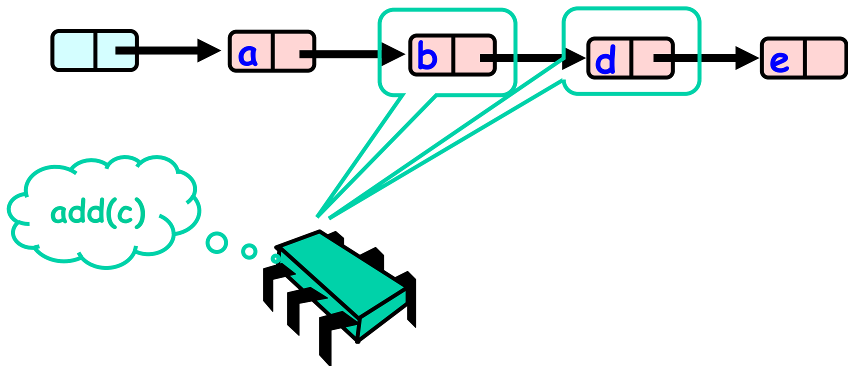


102

Cosa potrebbe andare storto?

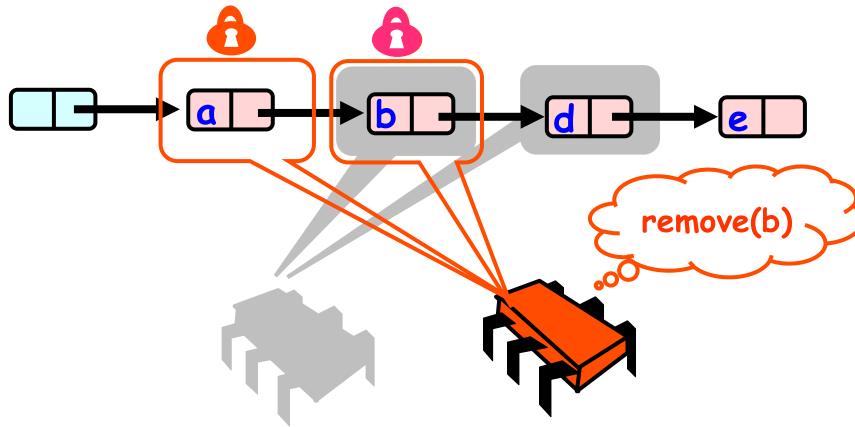


Cosa potrebbe andare storto?



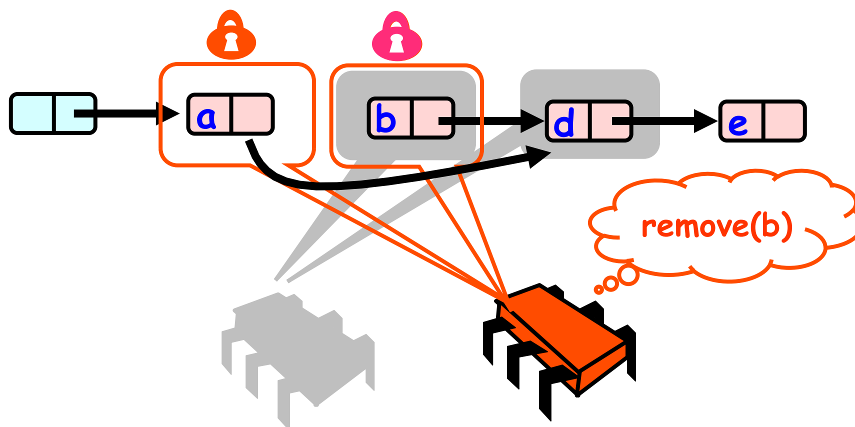
104

Cosa potrebbe andare storto?



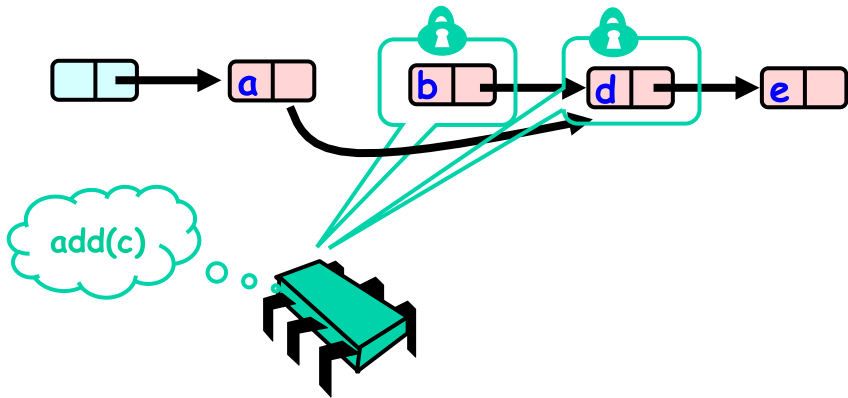
105

Cosa potrebbe andare storto?



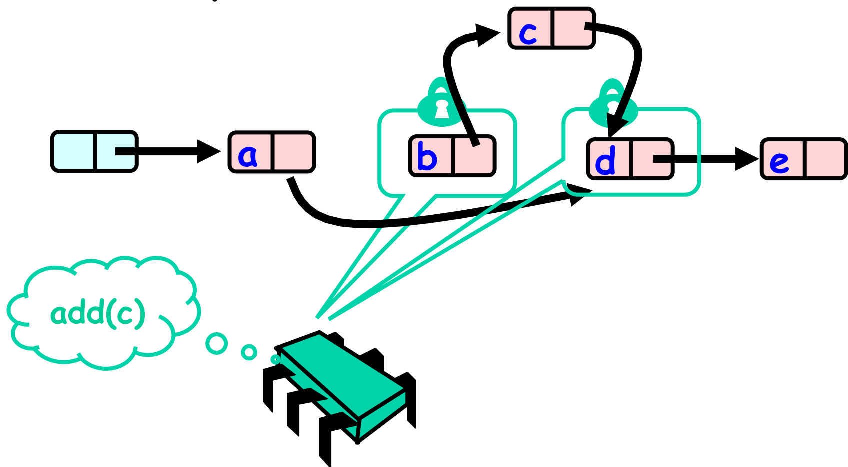
106

Cosa potrebbe andare storto?



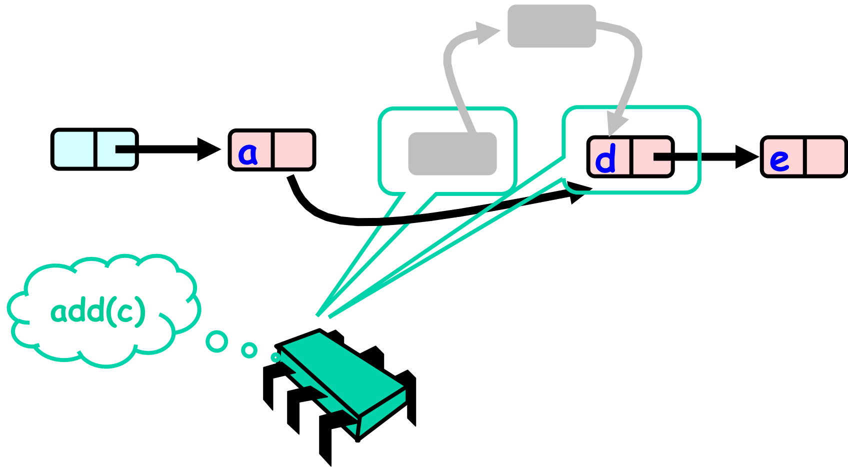
107

Cosa potrebbe andare storto?

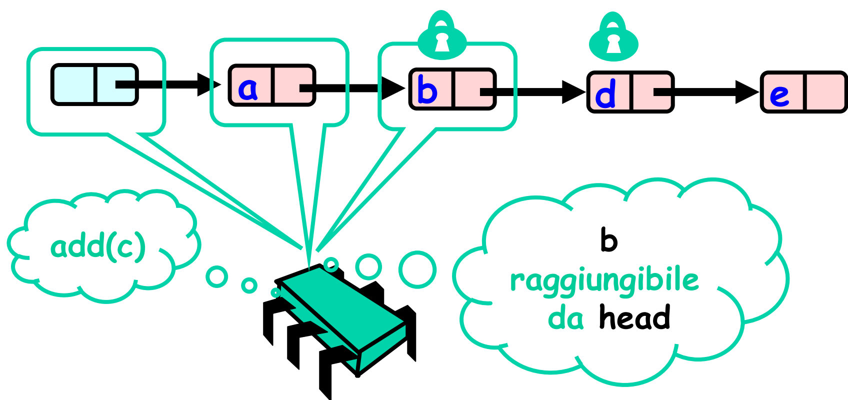


108

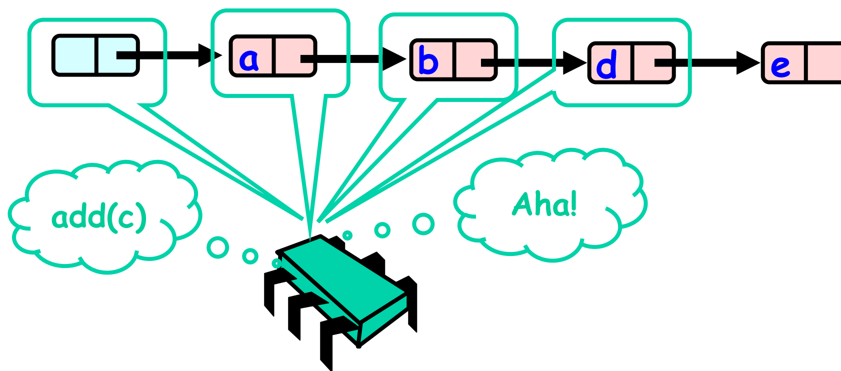
Cosa potrebbe andare storto?



109

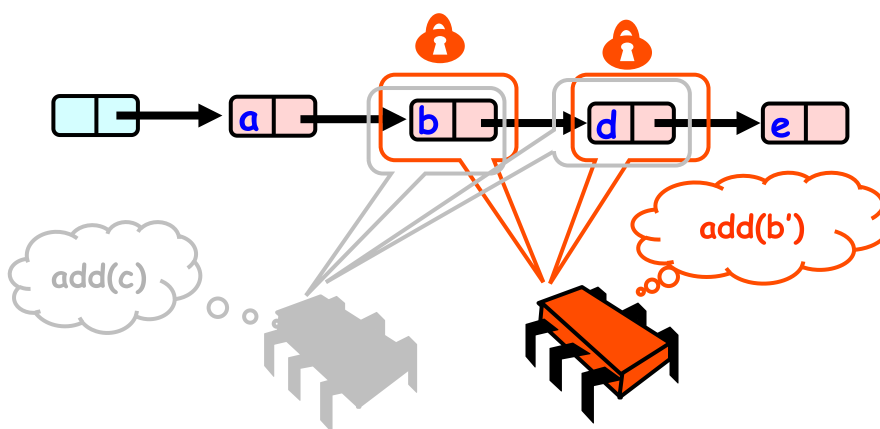


Ancora problemi

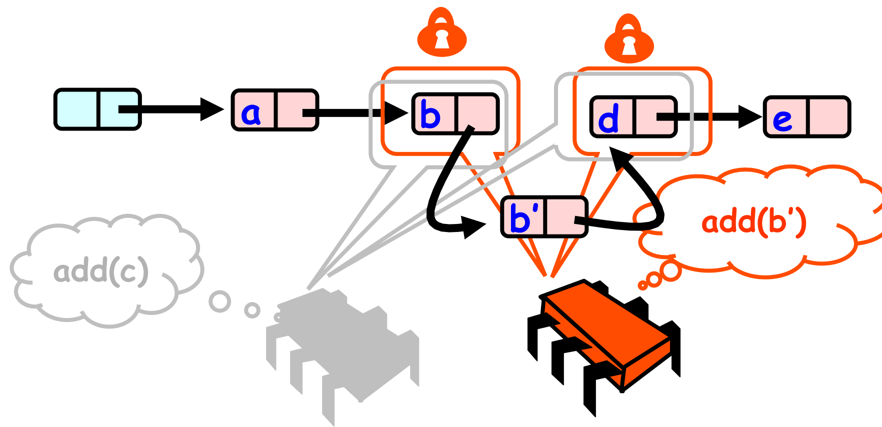


111

Ancora problemi

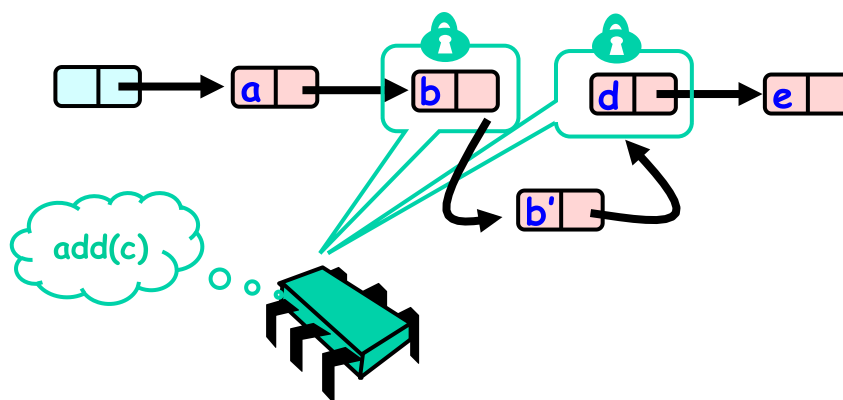


Ancora problemi



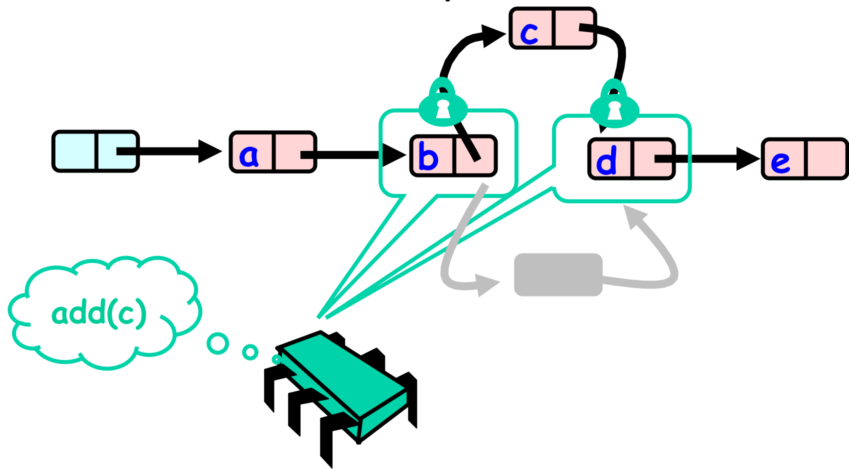
113

Ancora problemi

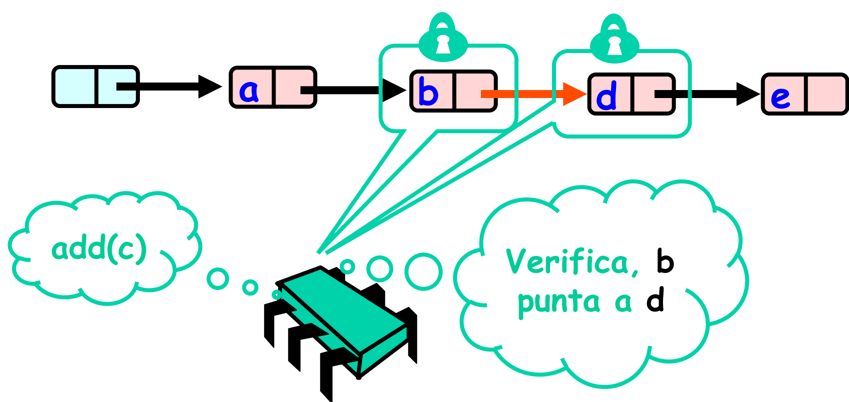


114

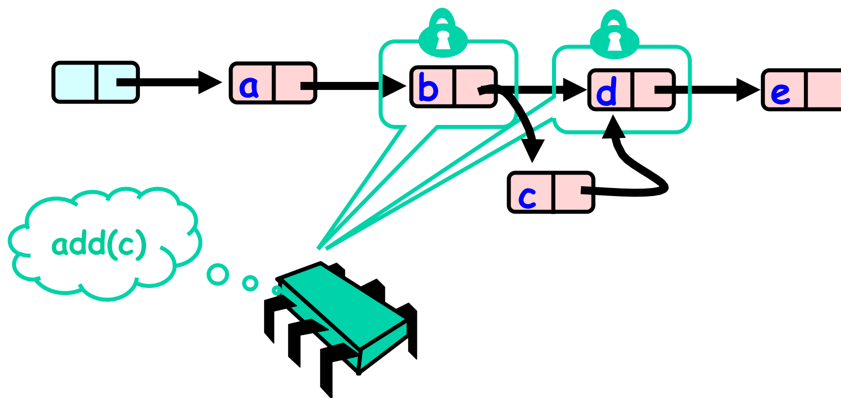
Ancora problemi



115



Optimistic: Linearization Point

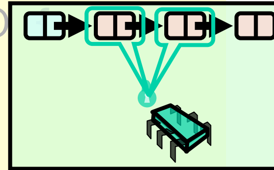


Validation

```
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;
}
```

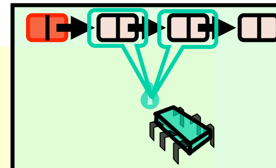
Validation

```
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

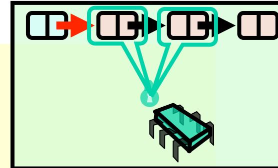
Validation



```
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

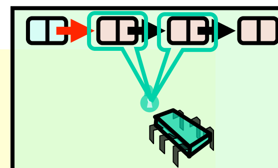
Validation



```
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

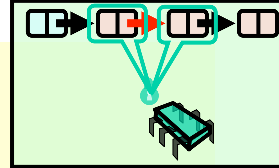
Validation



```
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



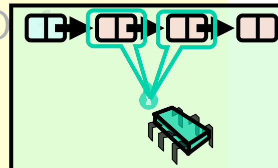
```
private boolean
 validate(Node pred,
          Node curr) {
 Node node = head;
 while (node.key <= pred.key) {
  if (node == pred)
   return pred.next == curr;
  node = node.next;
 }
 return false;
}
```

Is current node next?

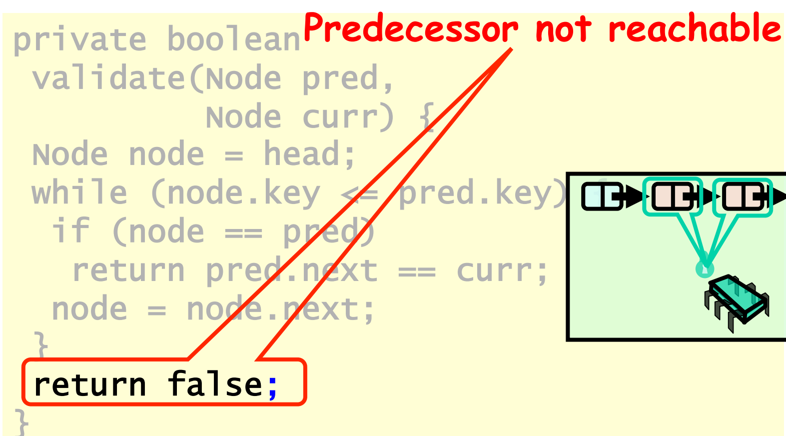
Validation

Otherwise move on

```
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;
}
```



Validation

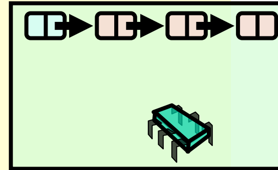


Remove: searching

```
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

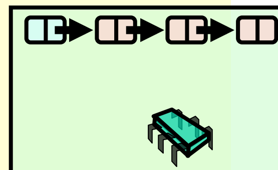
```
public boolean remove(Item item) {
 int key = item.hashCode();
 retry: while (true) {
   Node pred = this.head;
   Node curr = pred.next;
   while (curr.key <= key) {
    if (item == curr.item)
     break;
    pred = curr;
    curr = curr.next;
   } …
```



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;
   } …
```

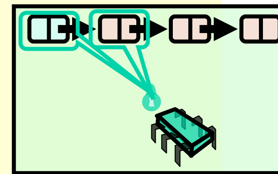


Retry on synchronization conflict

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;
   } …
```

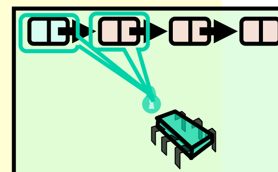
Examine predecessor and current nodes



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;
   } …
```

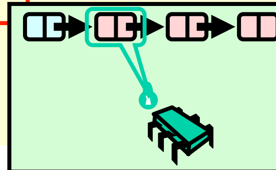
Search by key



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;
```

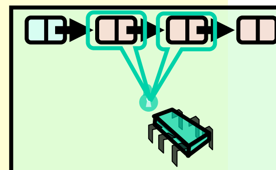
Stop if we find item



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



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

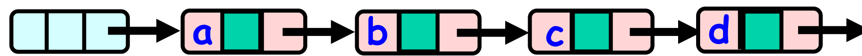
Valutazione

- Buona gestione dei lock
  - Performance
  - Concurrency
- Problemi
  - Scorrere la lista due volte
  - contains() comunque richiede lock

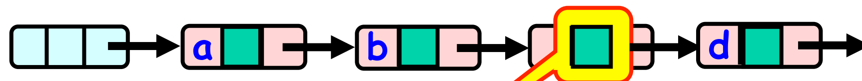
Lazy List

- **remove()**
  - Scan list
  - Lock predecessor & current
- Logical delete
  - Marcare il nodo come eliminato (!!!!)
- Physical delete
  - Ridirezionare i puntatori

Lazy Removal

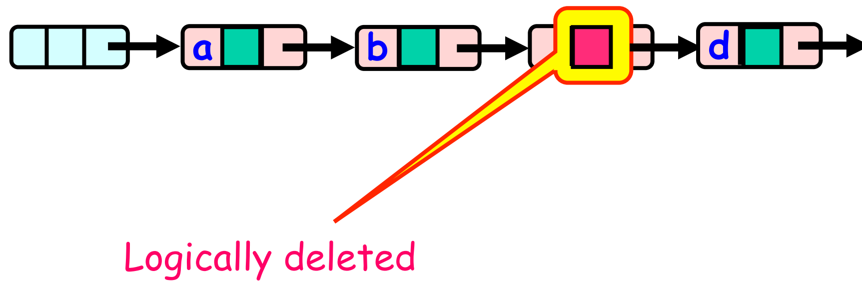


Lazy Removal



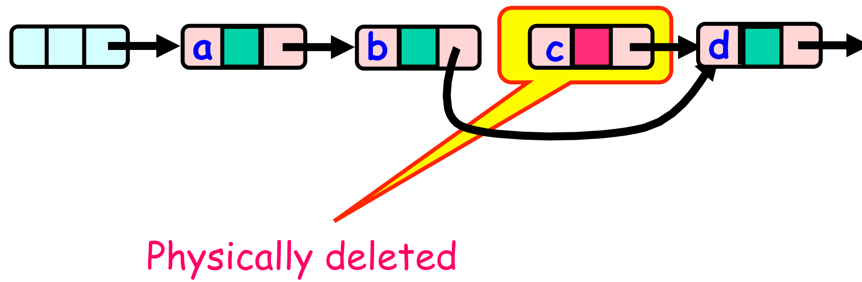
Present in list

Lazy Removal

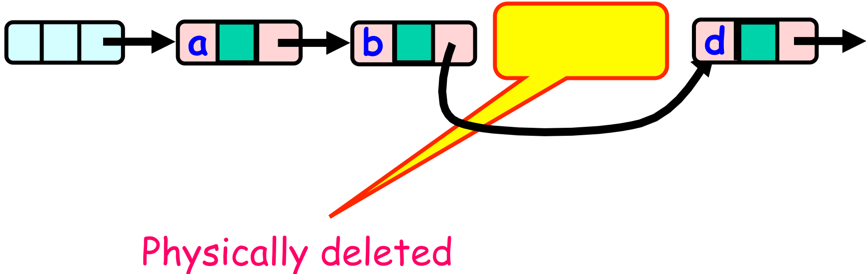


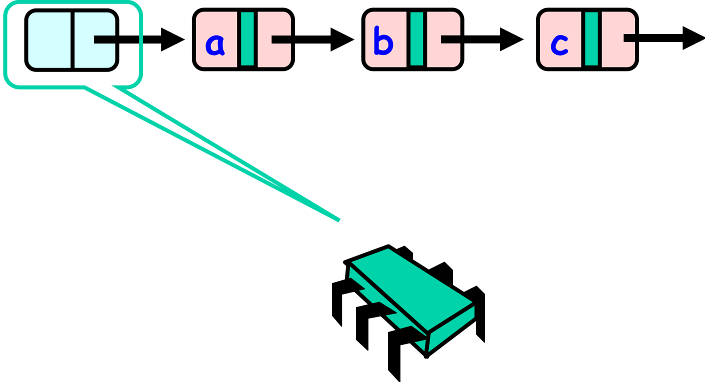
143

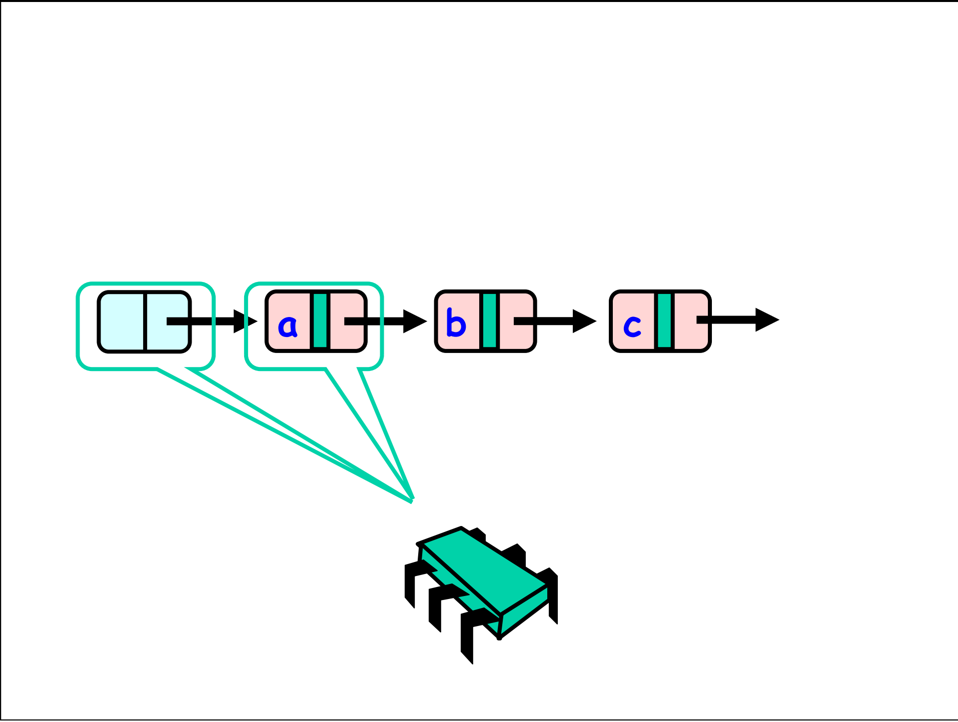
Lazy Removal



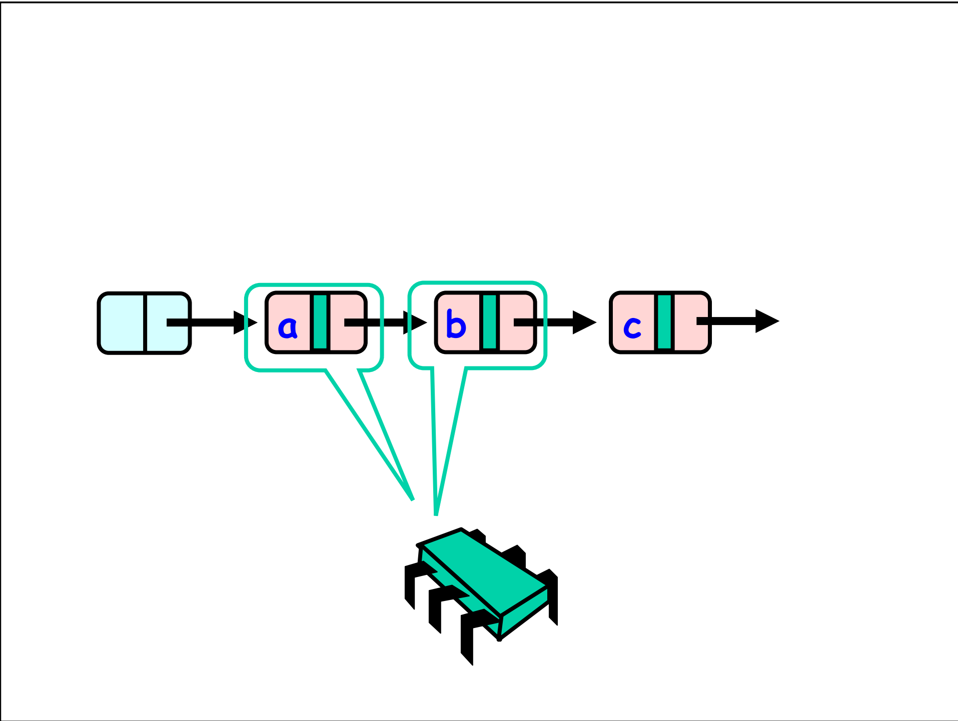
Lazy Removal



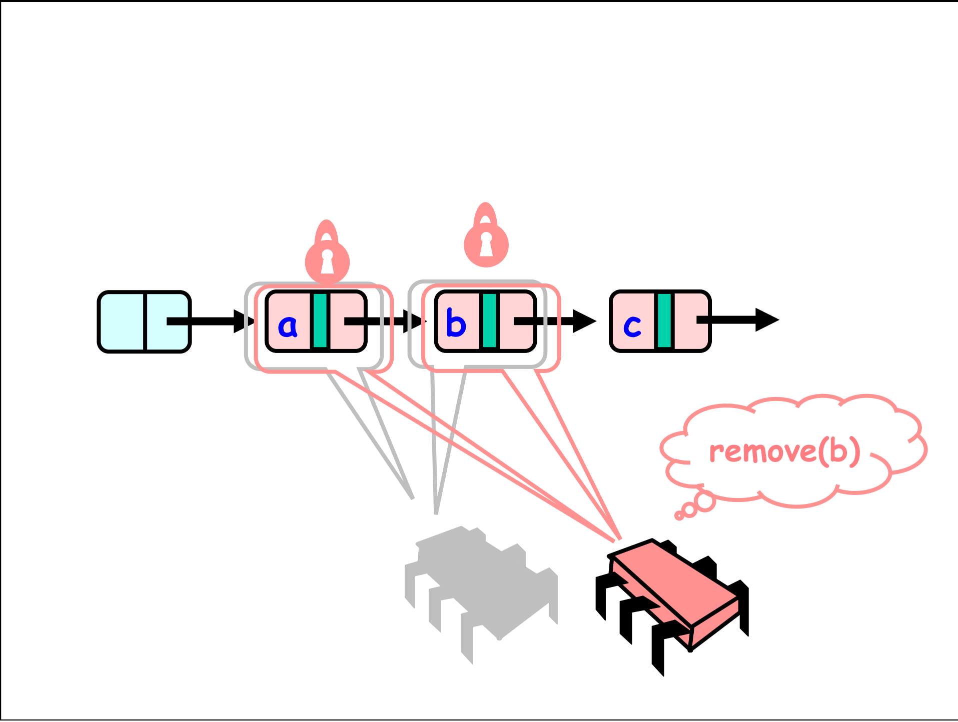


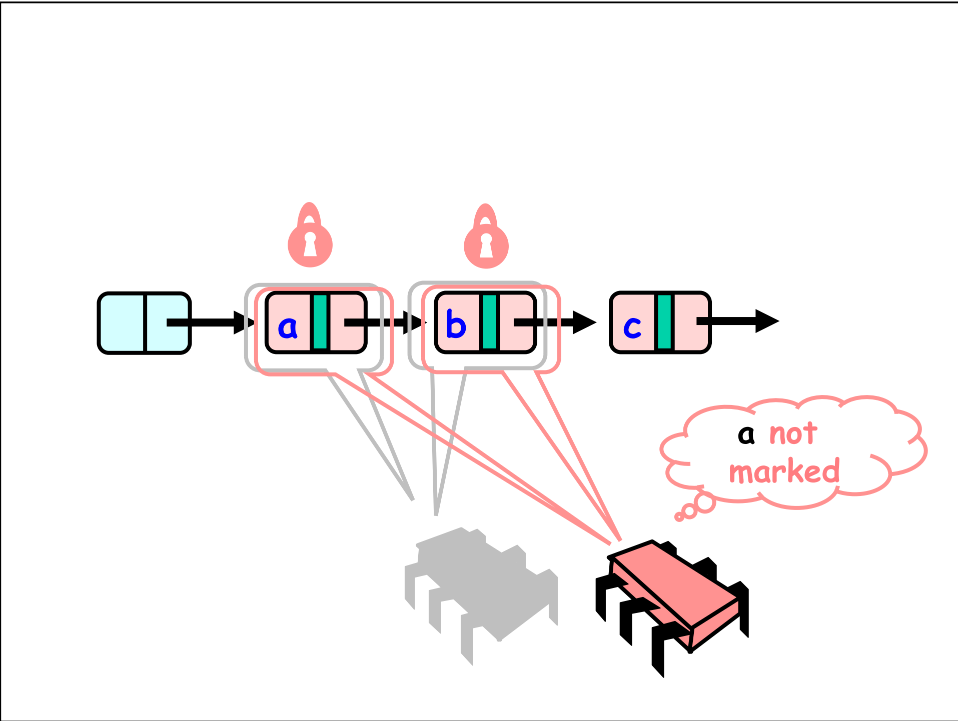


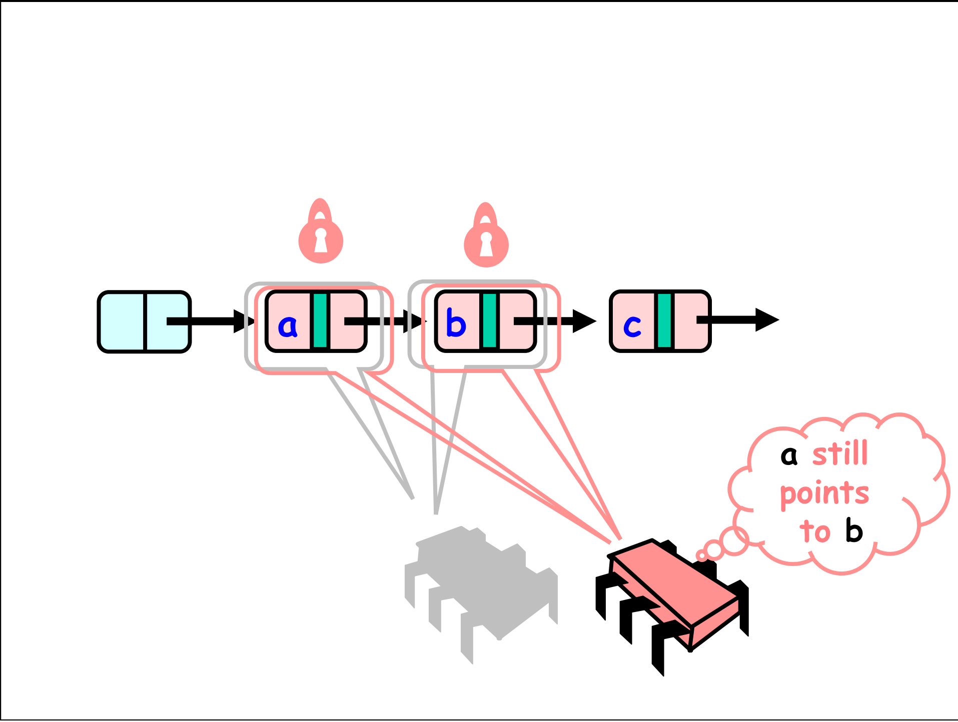
a
b
c

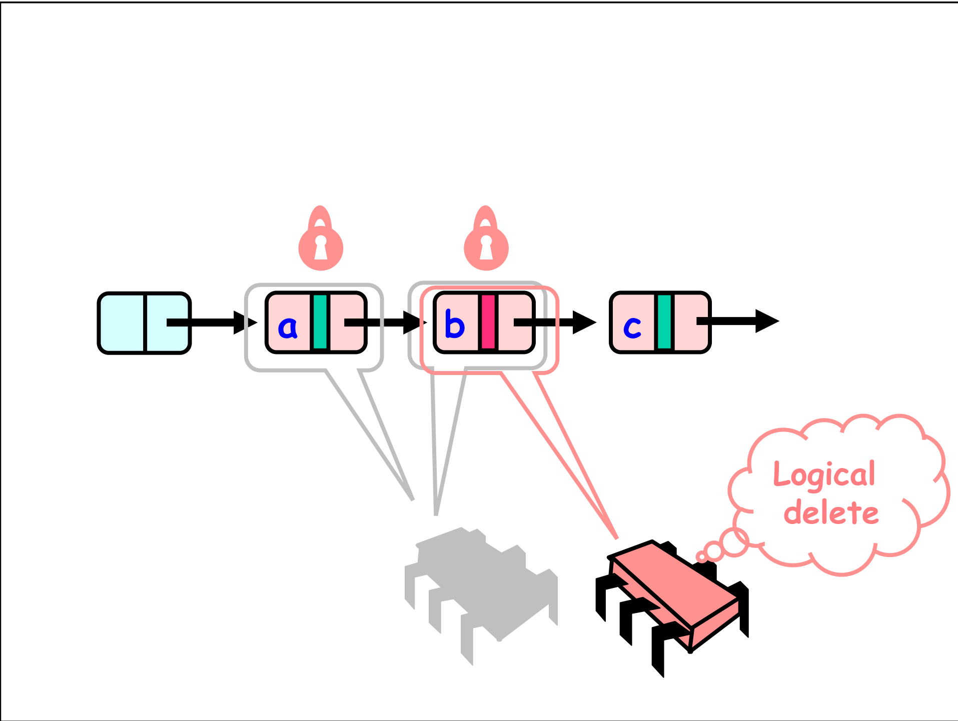


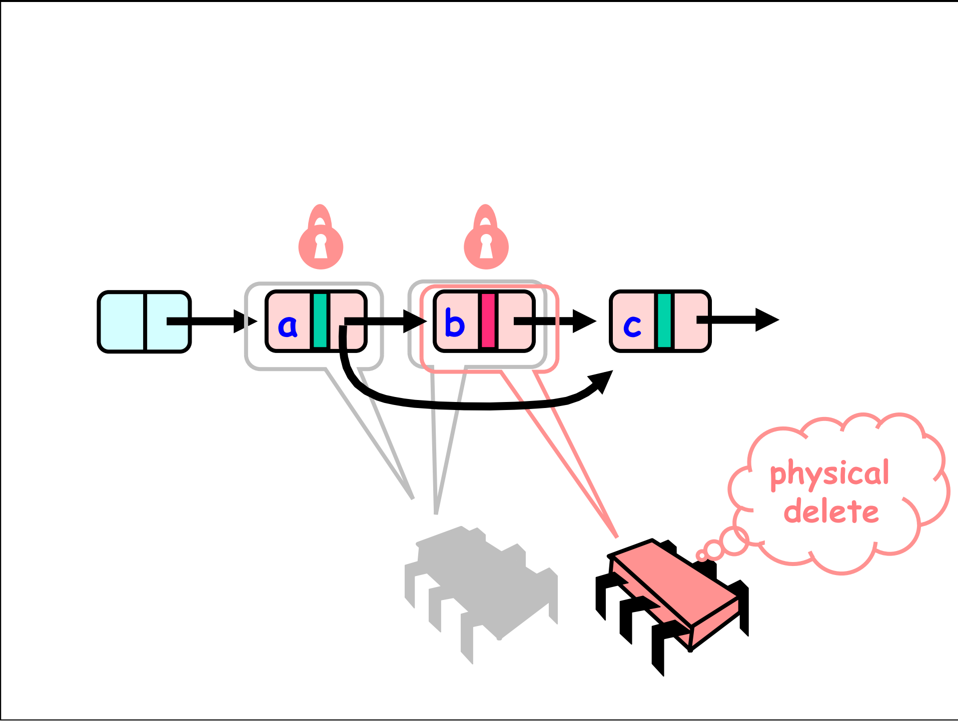
a
b
c

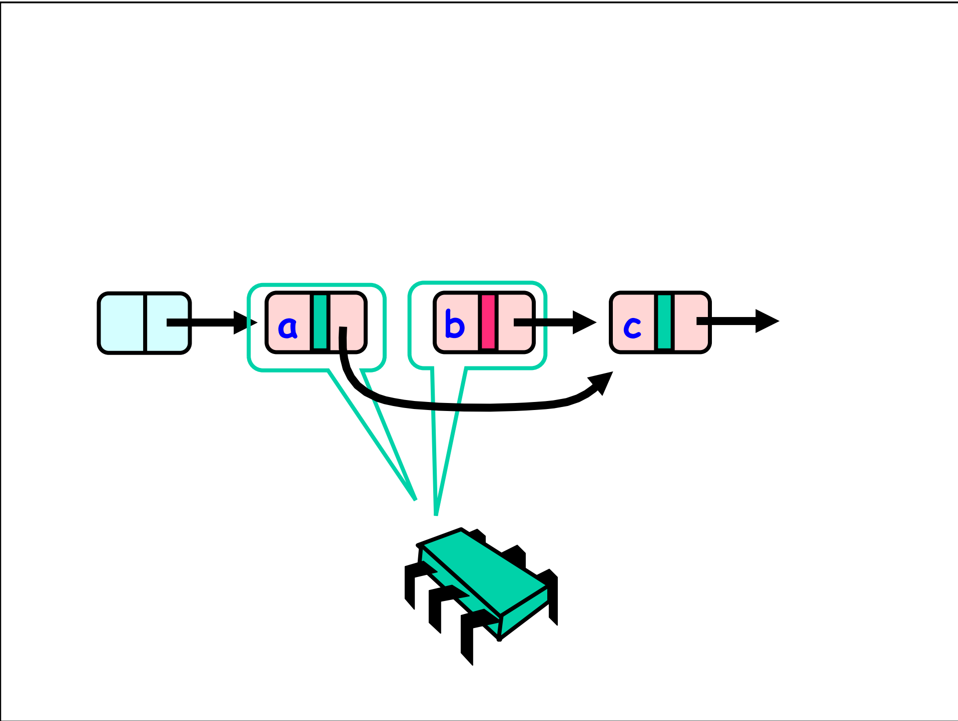

a
b
c
remove(b)


a
b
c
a not
marked


a
b
c
a still points to b


a
b
c
Logical delete


a
b
c
physical
delete


a
b
c

Abstraction Map

- S(head) =
  - { x | esiste a tale che
    - a raggiungibile da head e
    - a.item = x e
    - aunmarked
  - }

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;
}
```

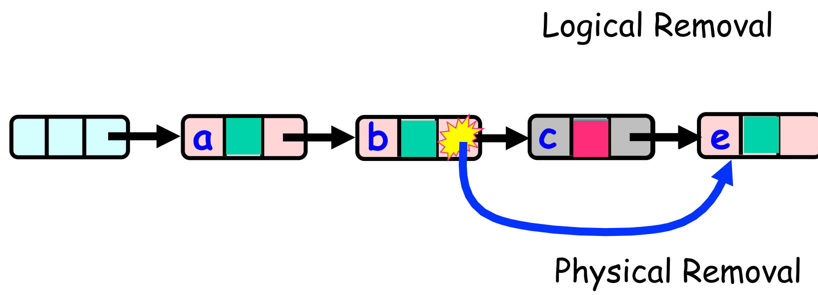
Traverse without locking
(nodes may have been removed)

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?

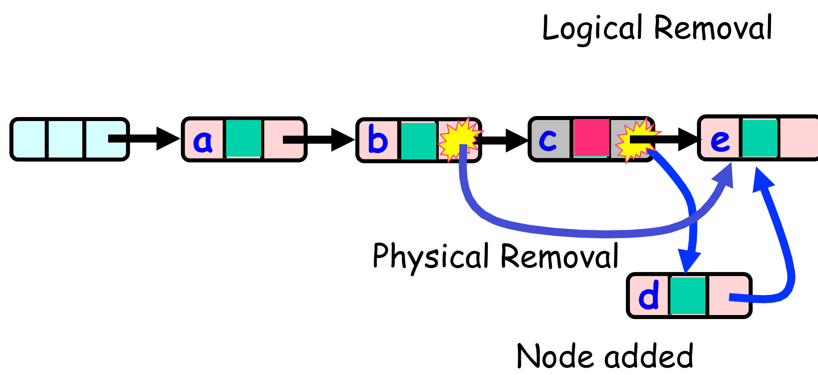
Lock-free Lists



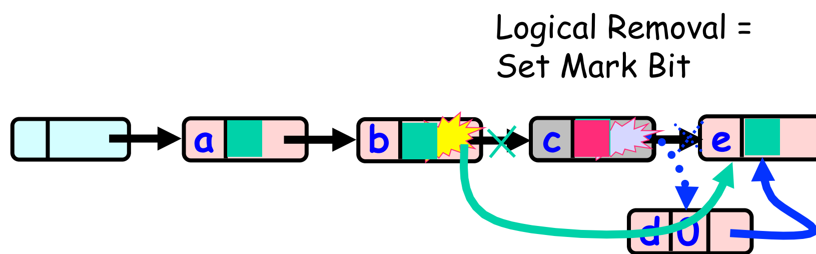
Non basta

171

Problema



Bit di marcatura e puntatori



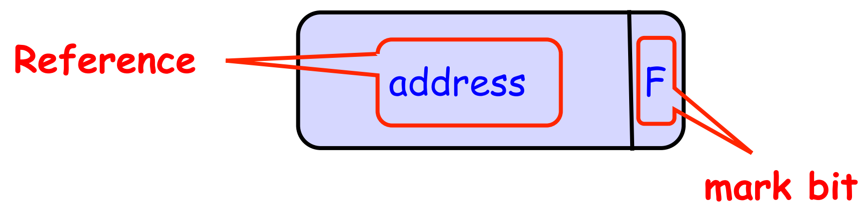
AtomicMarkableReference

AtomicMarkableReference

- Operazione atomica
  - Modificare il puntatore
  - Modificare la marcatura
- Remove (due passi)
  - Operare sul mark bit del campo next
  - Modificare il predecessor

In Java

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



Extracting Reference & Mark

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

Extracting Reference & Mark

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

Returns reference

Returns mark at array index 0!

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

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

…then change to this new reference …

… 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.

"To Lock or Not to Lock"

- Locking vs. Non-blocking: visioni estreme
- La risposta: compinare blocking e non blocking
  - Esempi: Lazy list :blocking add() eremove() con wait-free contains()