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…
  - contemporaneamente

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

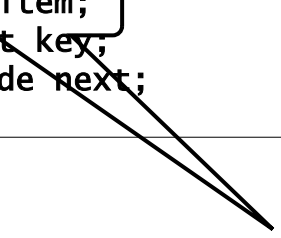
```
public interface Set<T> {
 public boolean add(T x);
 public boolean remove(T x);
 public boolean contains(T x);
}
```

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;
}
```



Contenuto informativo

List Node

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

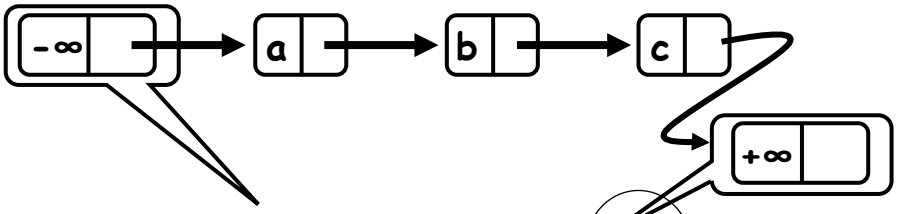
Chiave di ricerca

List Node

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

Riferimento al prossimo elemento

List-Based Set



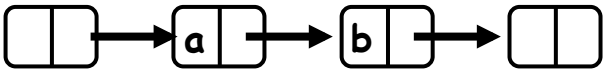
Ordinata con sentinelle
(chiavi min & max)

Invariante …

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

Funzione di astrazione

- Rep:



- Astrattamente:
  - {**a**, **b**}

Rep Invariant

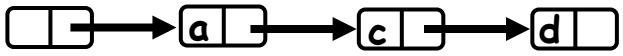
- Rep invariant
  - Definisce quali sono le rappresentazioni legali "valide"
  - Preservato dai metodi
  - Dipende dai metodi

Abstraction Function

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

List Based Set

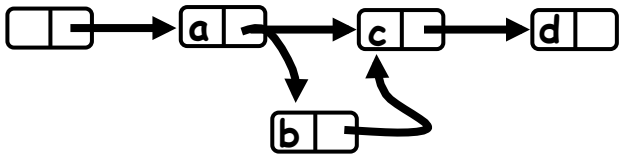
Add(b)



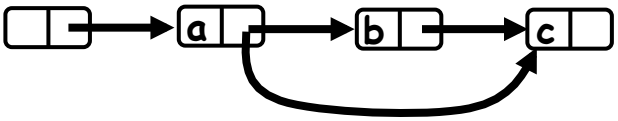
Remove(b)

List Based Set

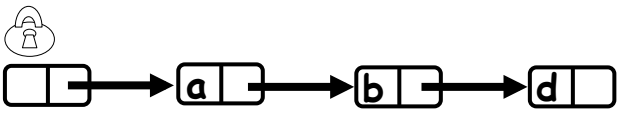
Add(b)



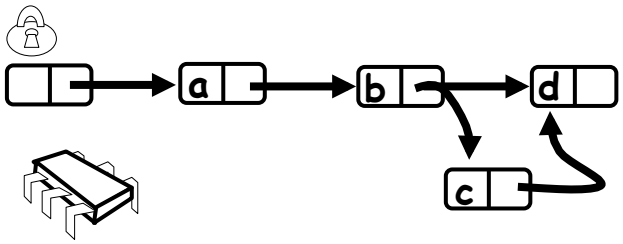
Remove(b)



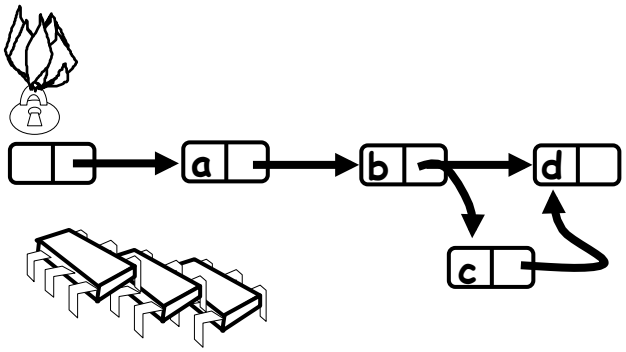
Coarse-Grained Locking



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

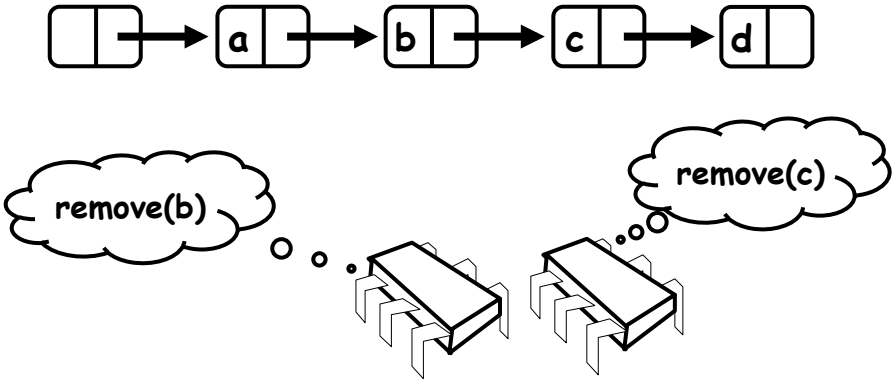
21

Fine-grained Locking

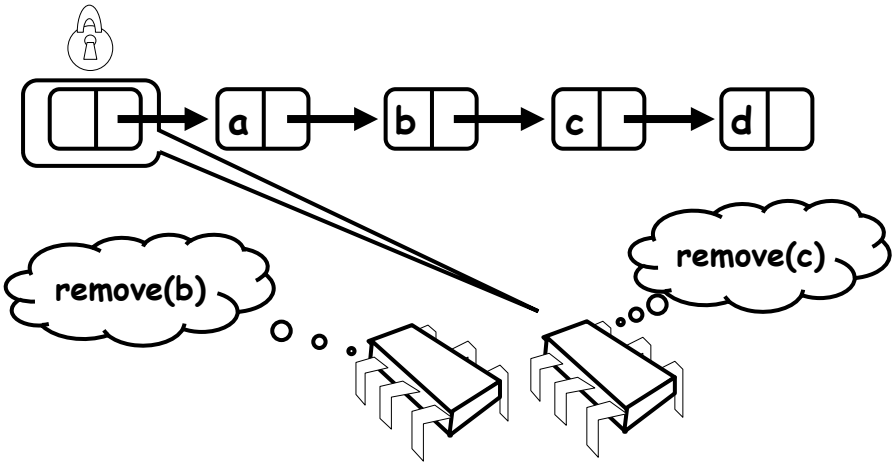
- Pensare prima di programmare a testa bassa
- Suddividere l'oggetto in parti
  - Ogni parte dell'oggetto ha un suo lock
  - Metodi che operano su parti disgiunte possono operare senza problema di sincronizzazione

22

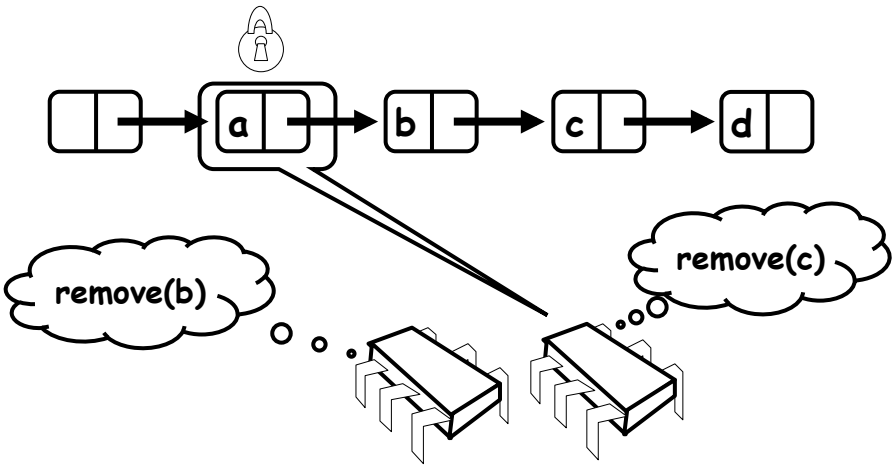
Concurrent Remove



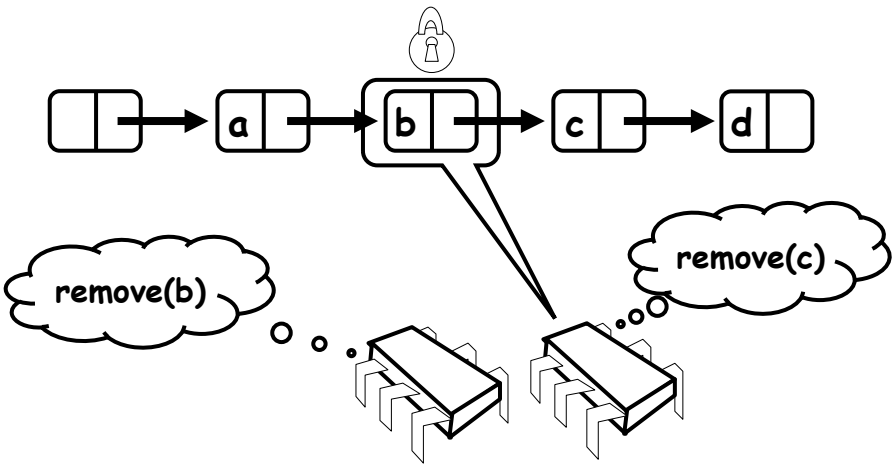
Concurrent Remove



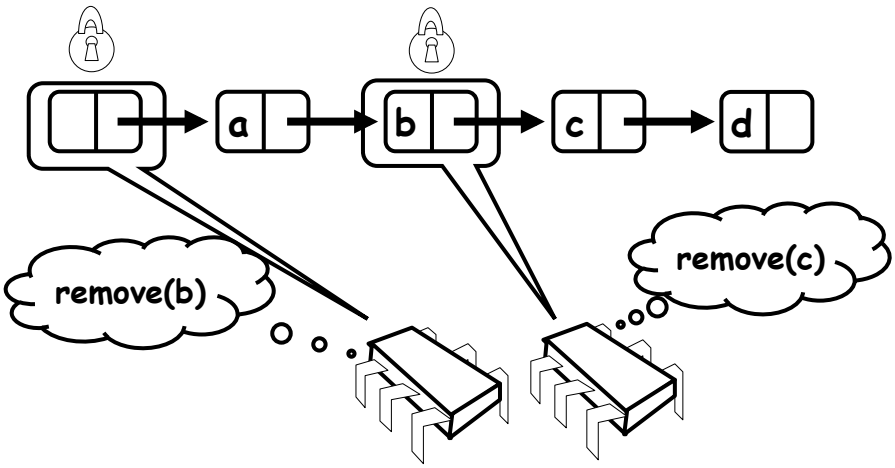
Concurrent Remove



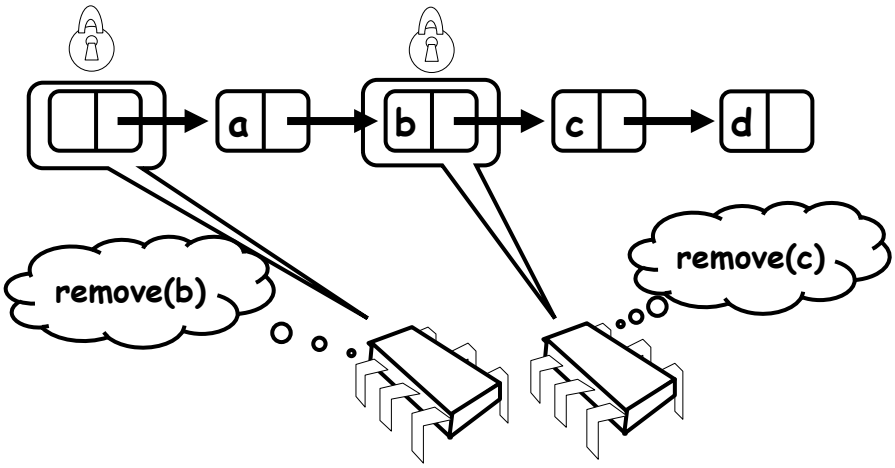
Concurrent Remove



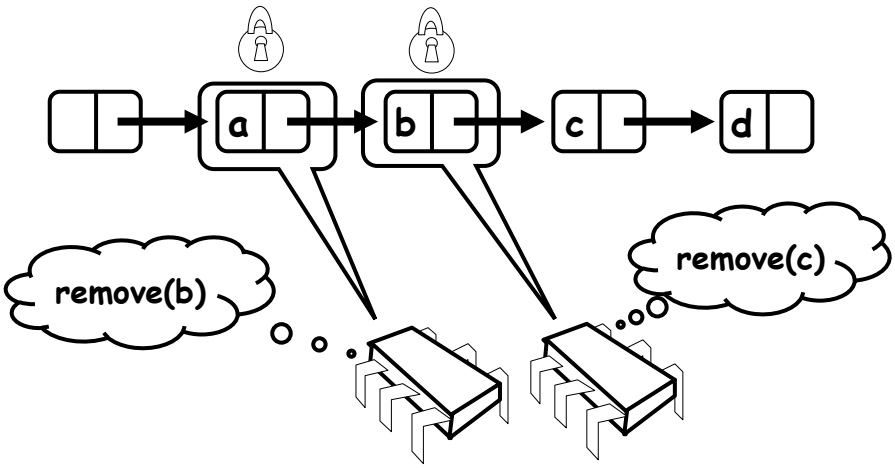
Concurrent Remove



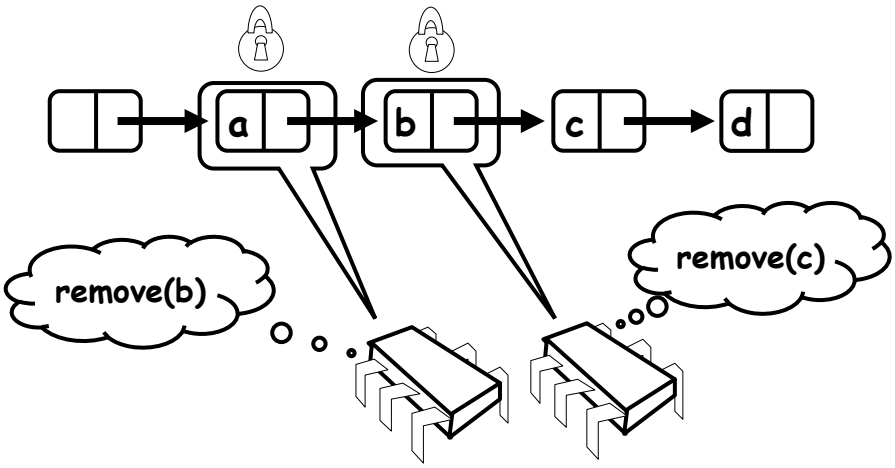
Concurrent Remove



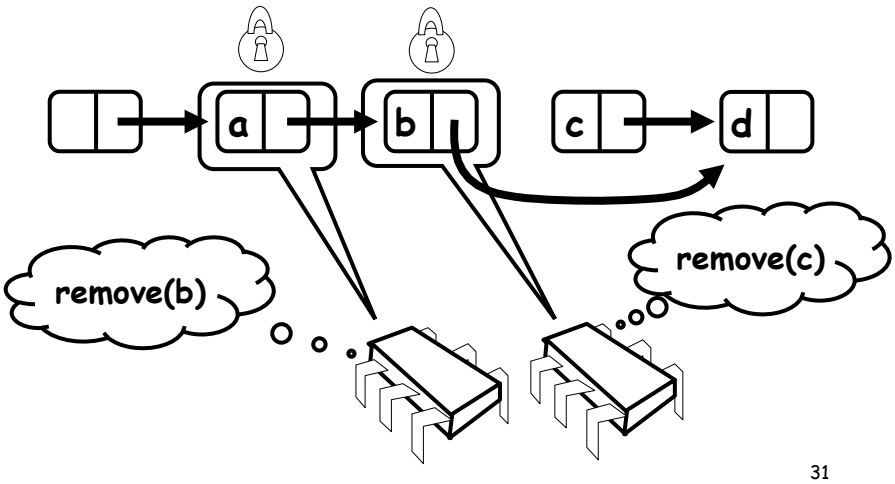
Concurrent Remove



Concurrent Remove

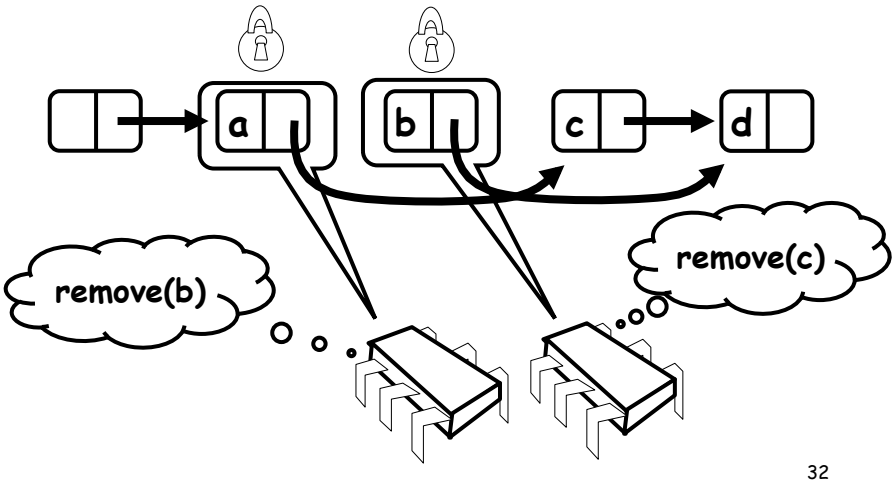


Concurrent Remove

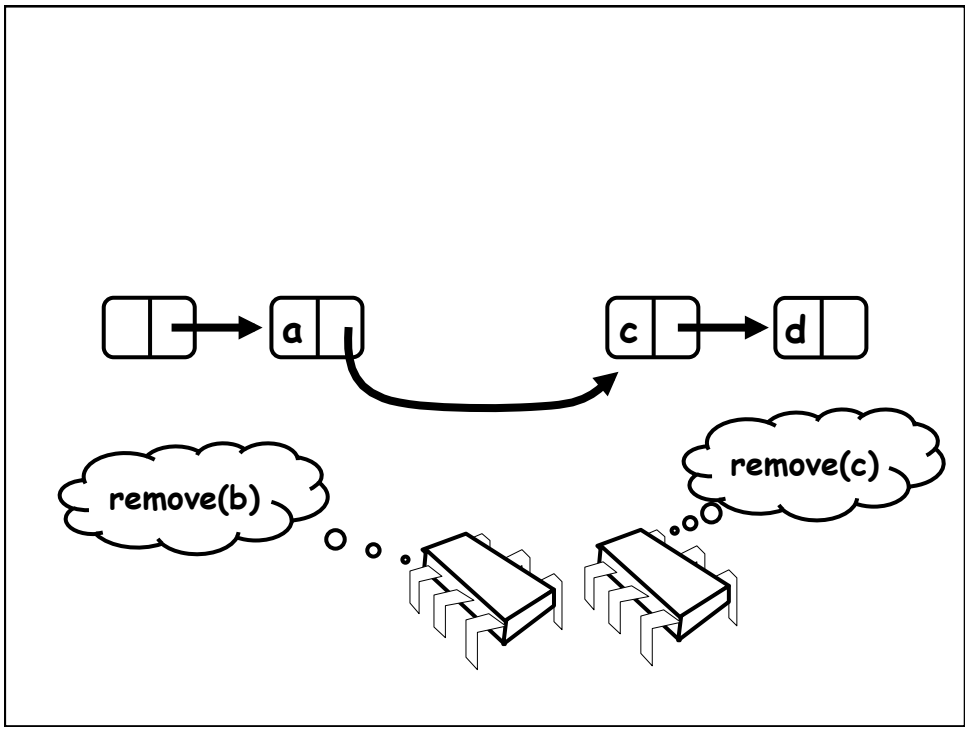


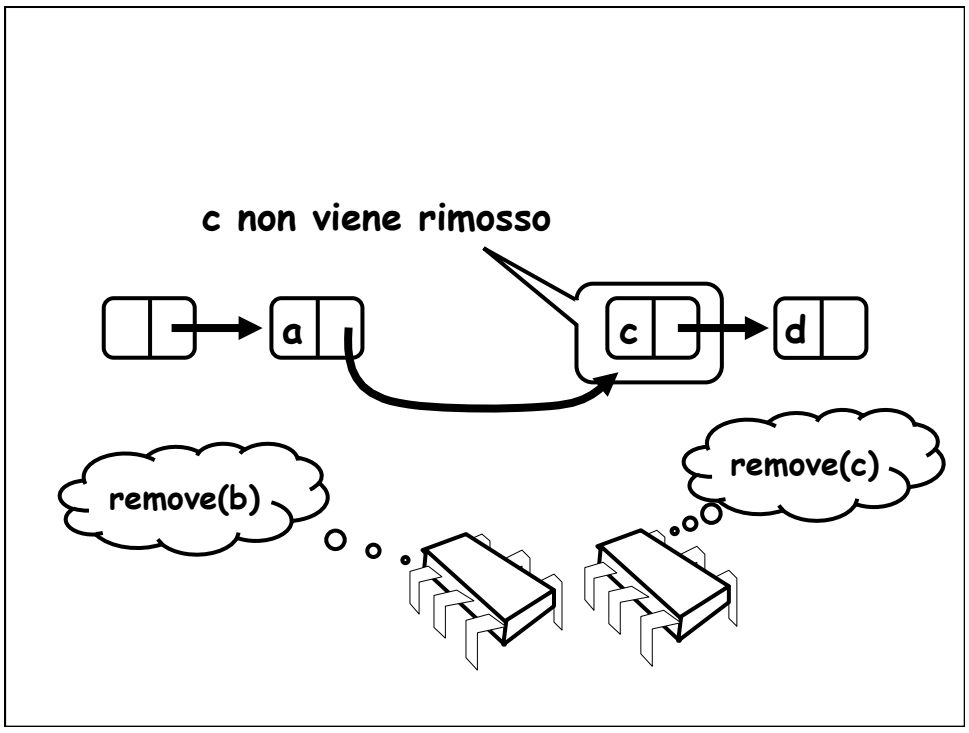
31

Concurrent Remove



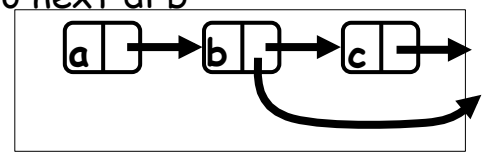
32

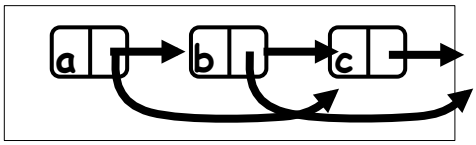

a
c
d
remove(b)
remove(c)


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

Quale e' il problema?

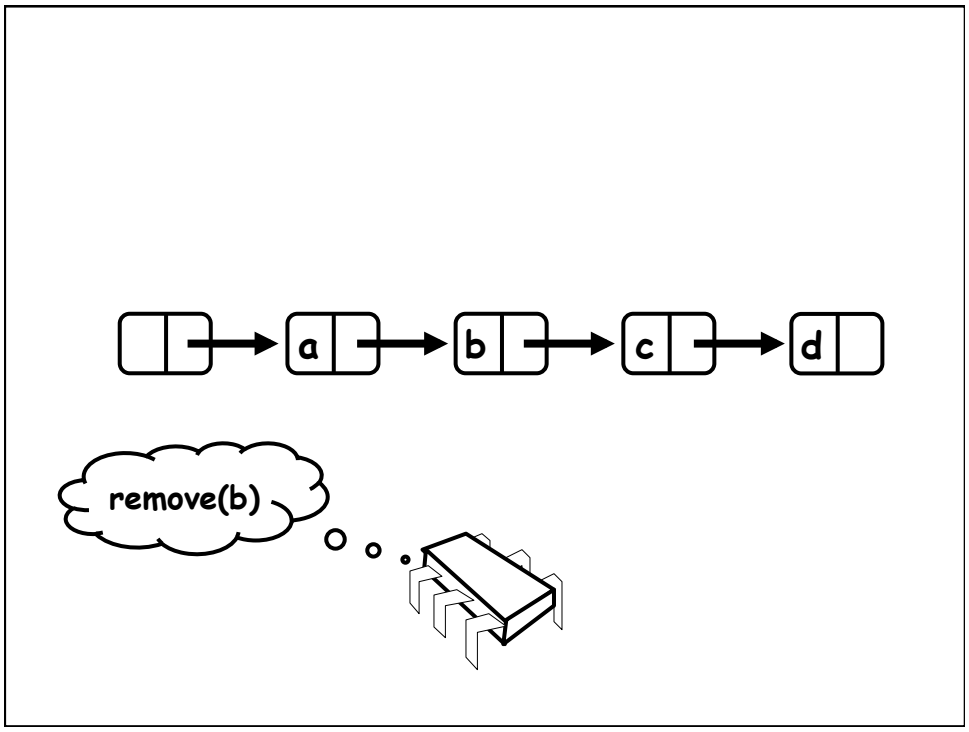
- Rimuovere c
  - Operare sul campo next di b

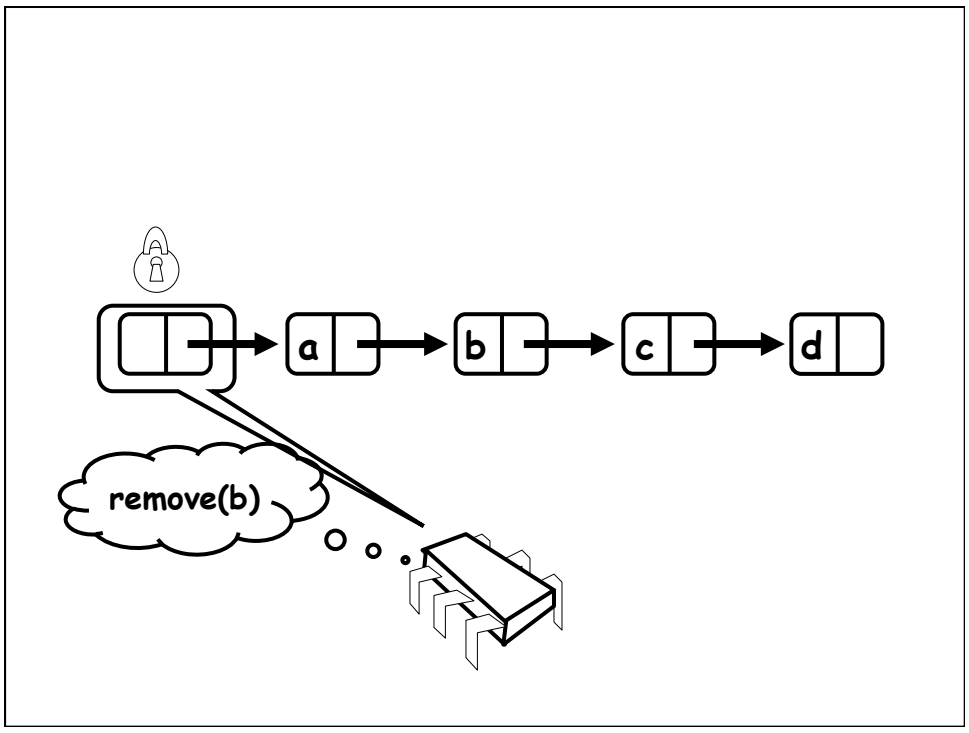


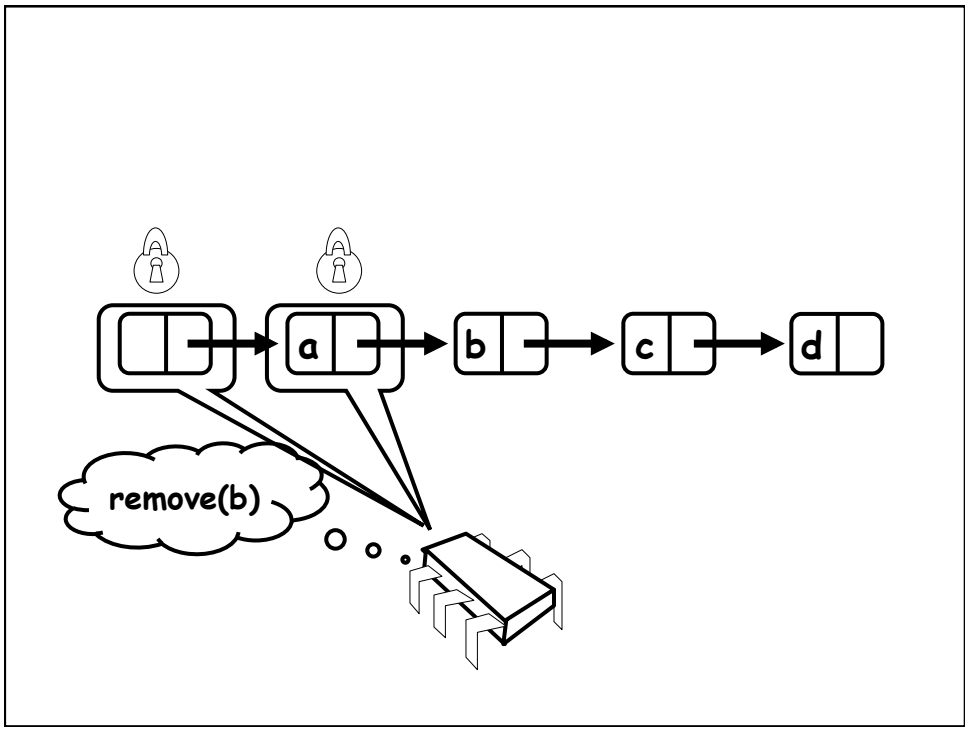


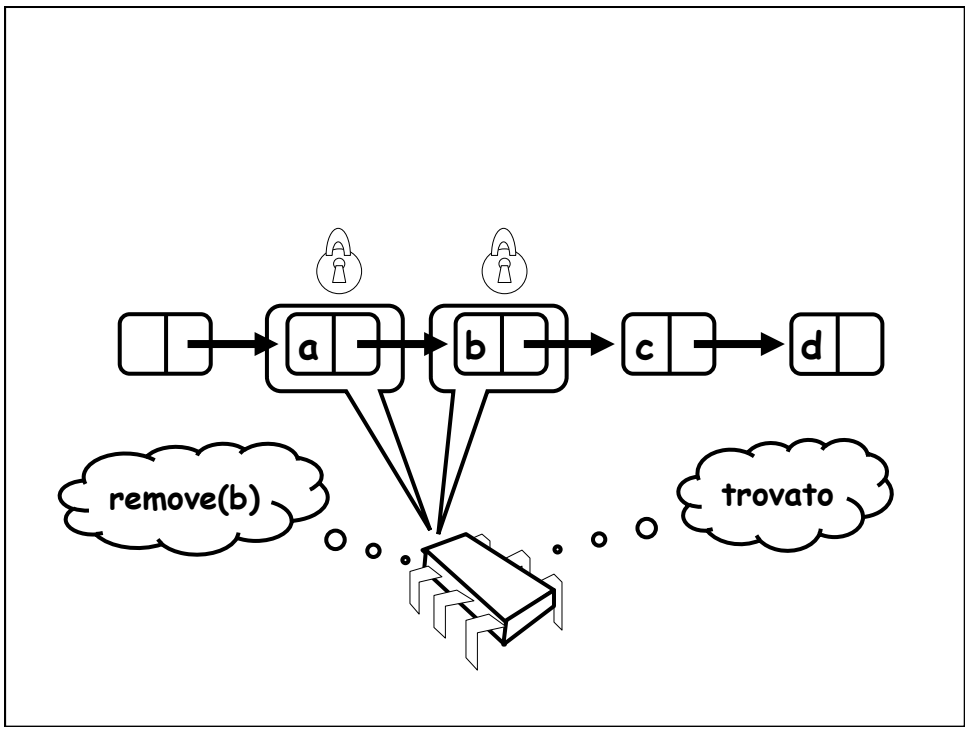
Cerchiamo di capire il problema

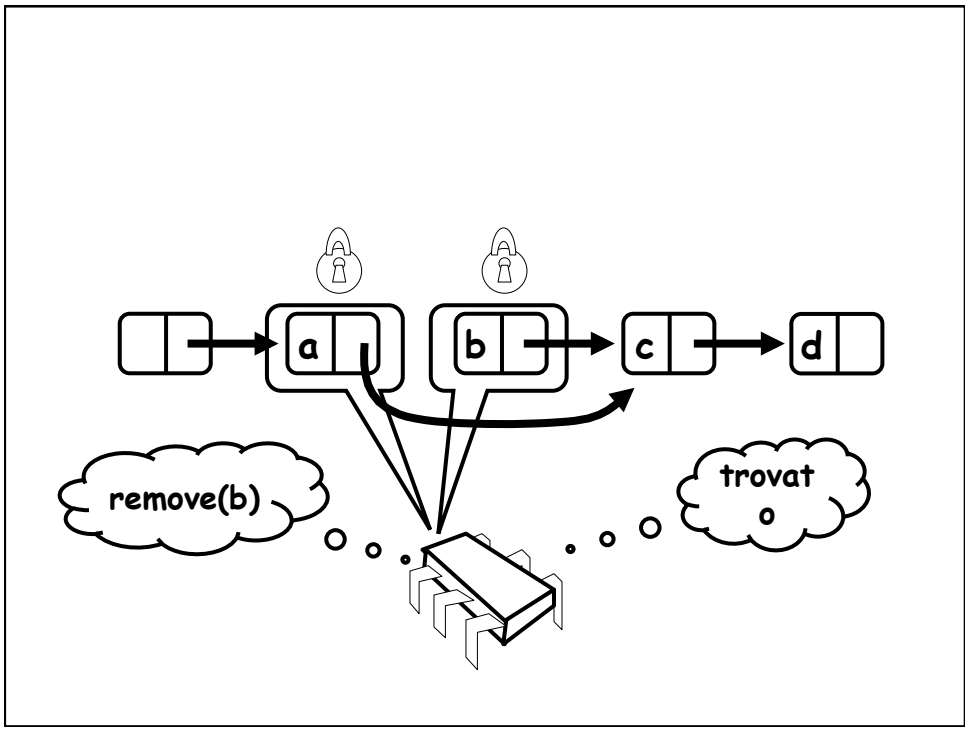
- Se un nodo e' "lock"
  - Nessuno puo cancellare il nodo *successore*
- Morale: il thread deve fare il lock
  - Sul nodo da cancellare
  - E sul predecessore

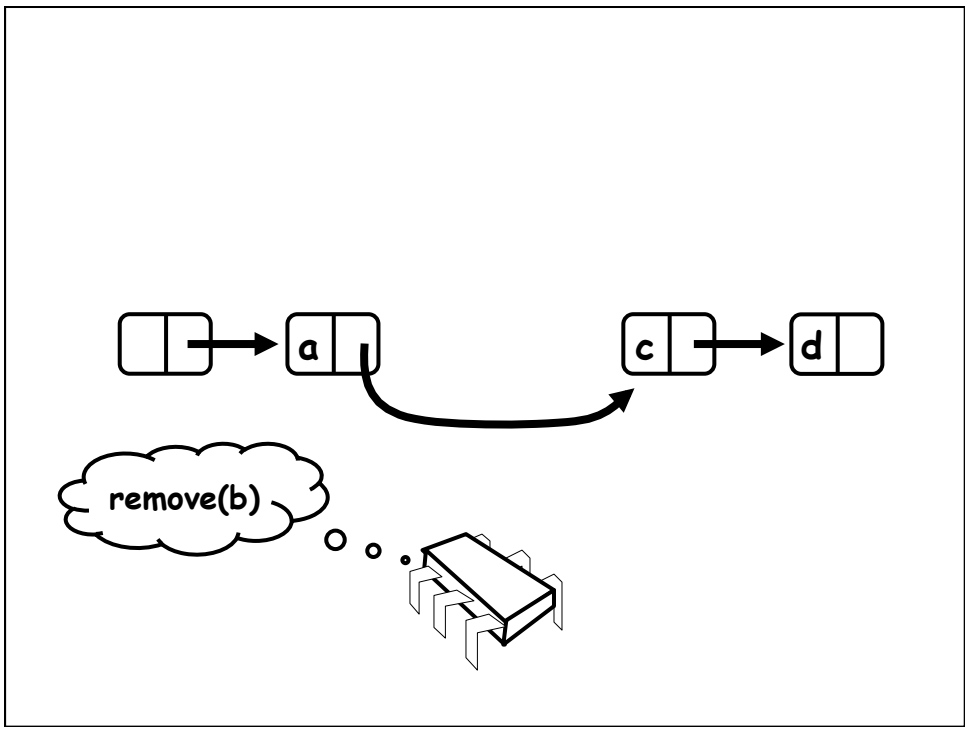

a
b
c
d
remove(b)

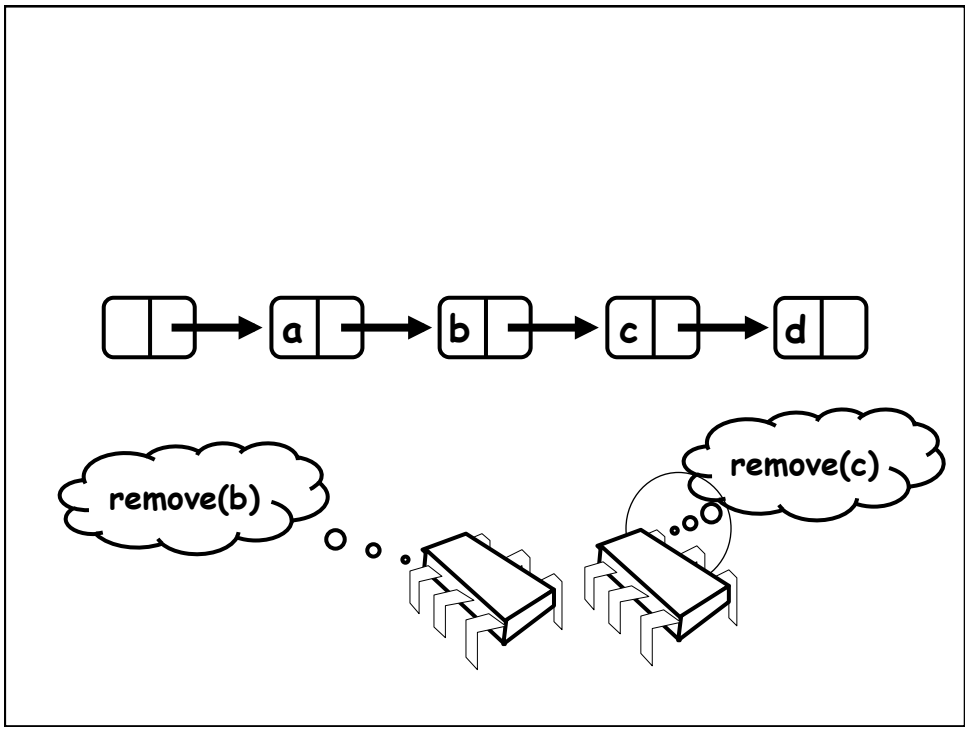

a
b
c
d
remove(b)

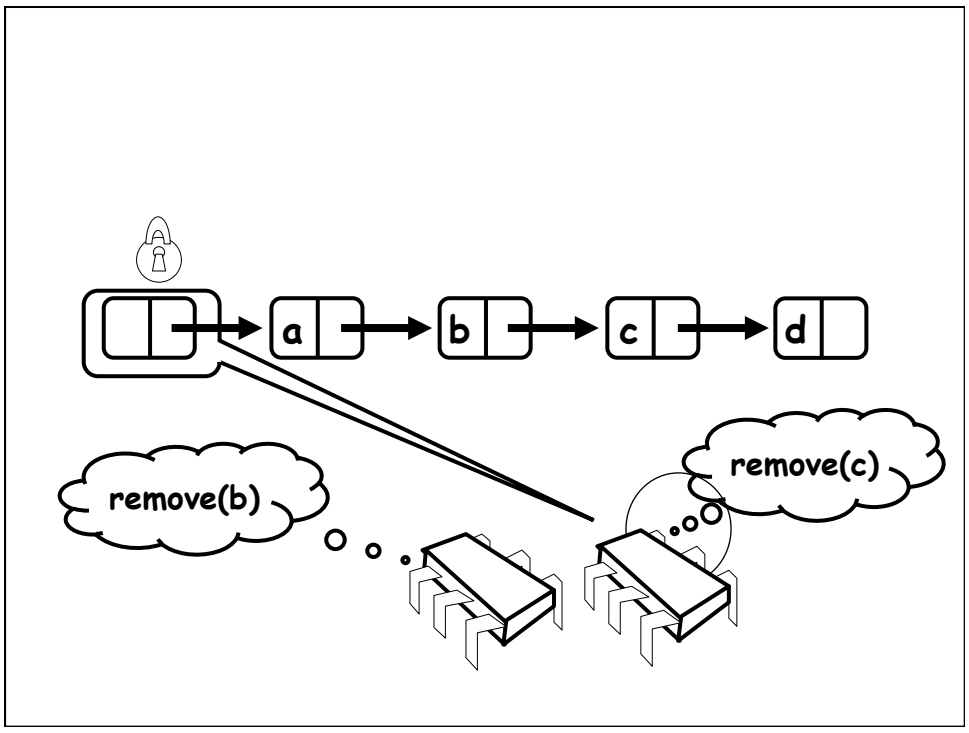

a
b
c
d
remove(b)

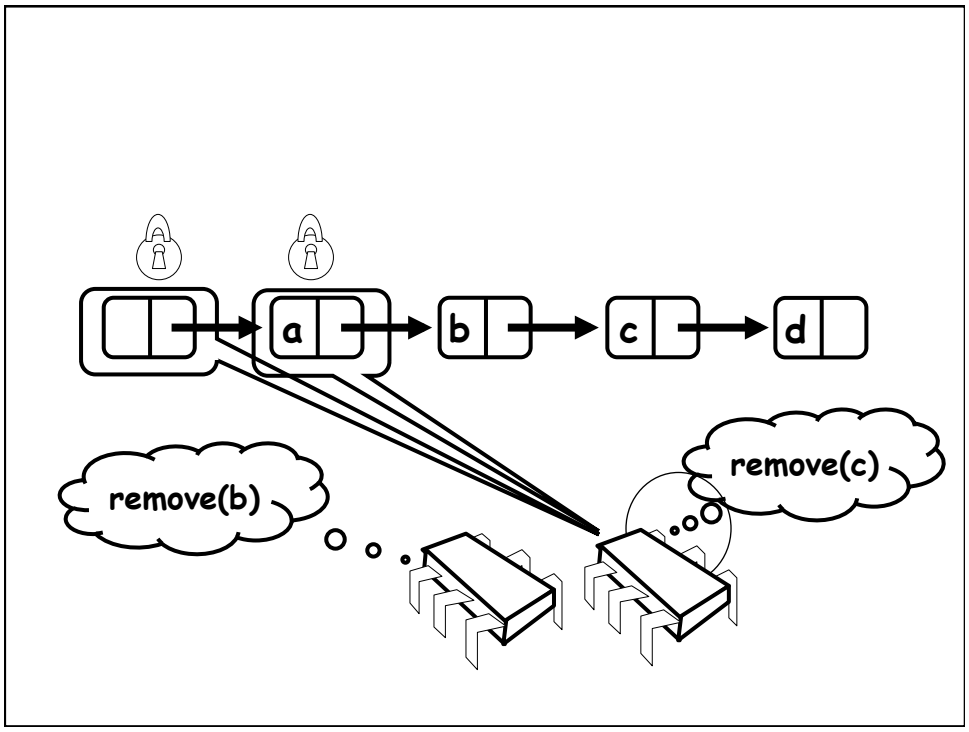

a
b
c
d
remove(b)
trovato

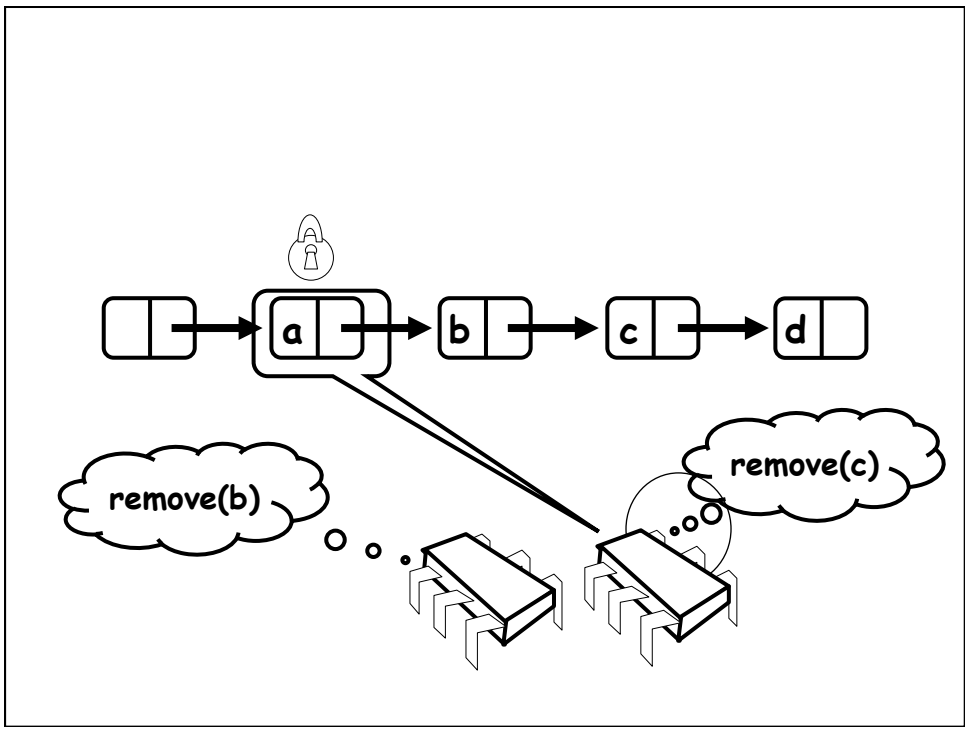

a
b
c
d
remove(b)
trovat
o

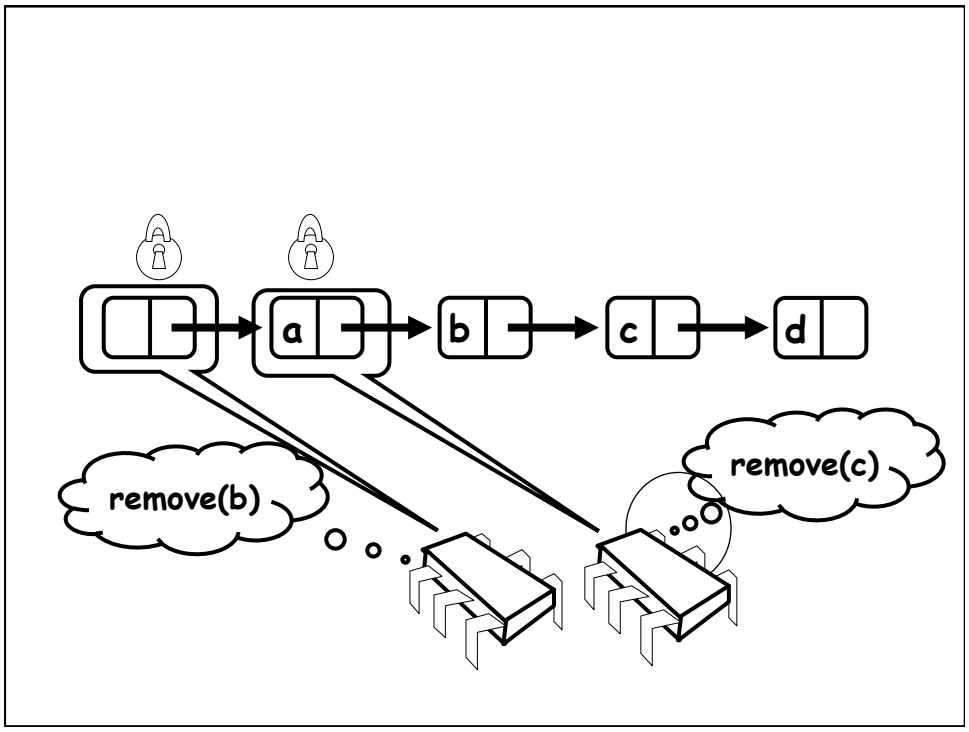

a
c
d
remove(b)

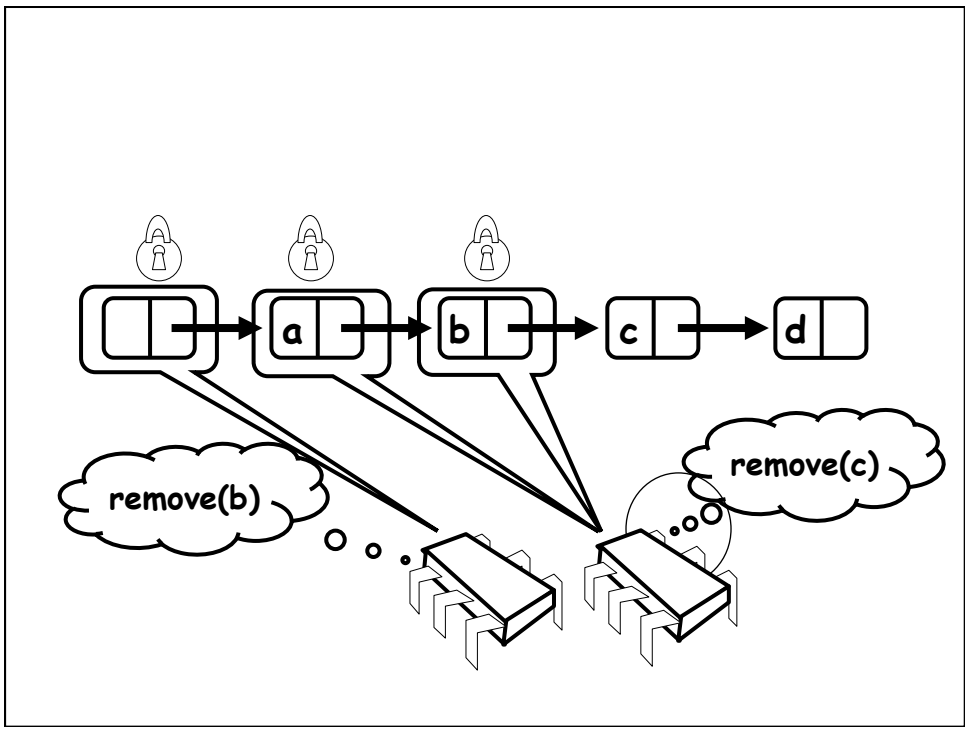

a
b
c
d
remove(b)
remove(c)

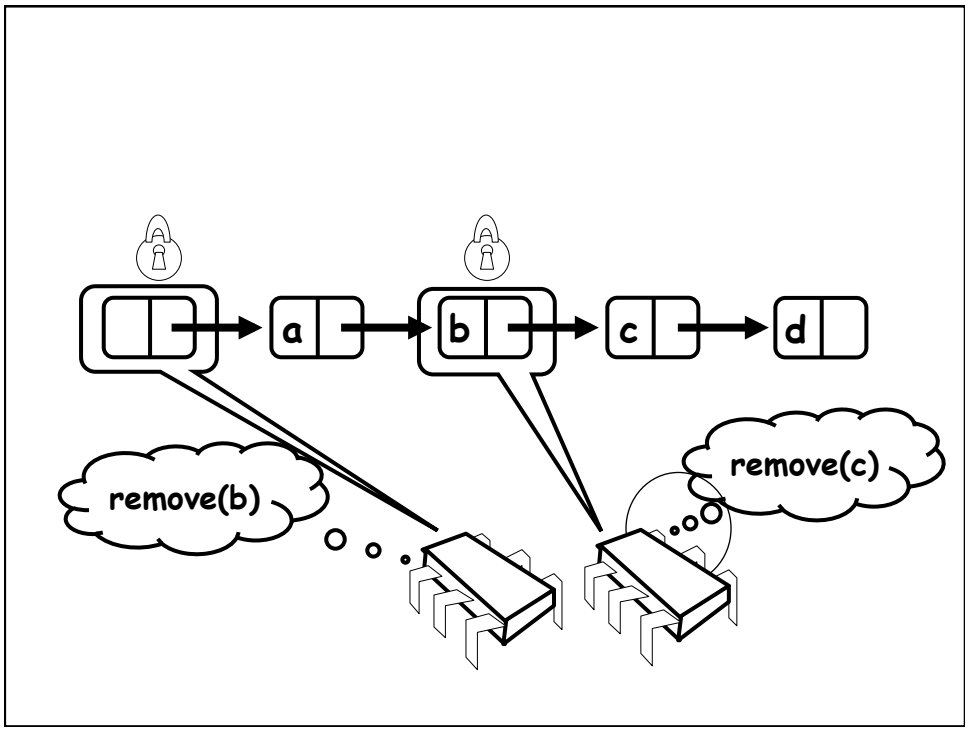

a
b
c
d
remove(b)
remove(c)

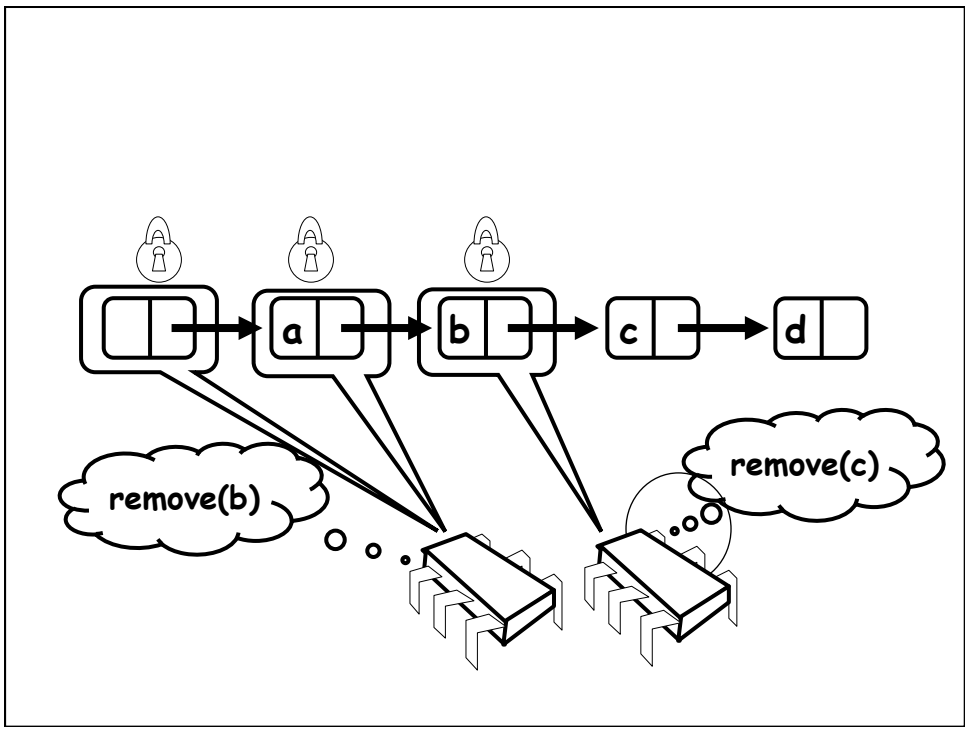

a
b
c
d
remove(b)
remove(c)

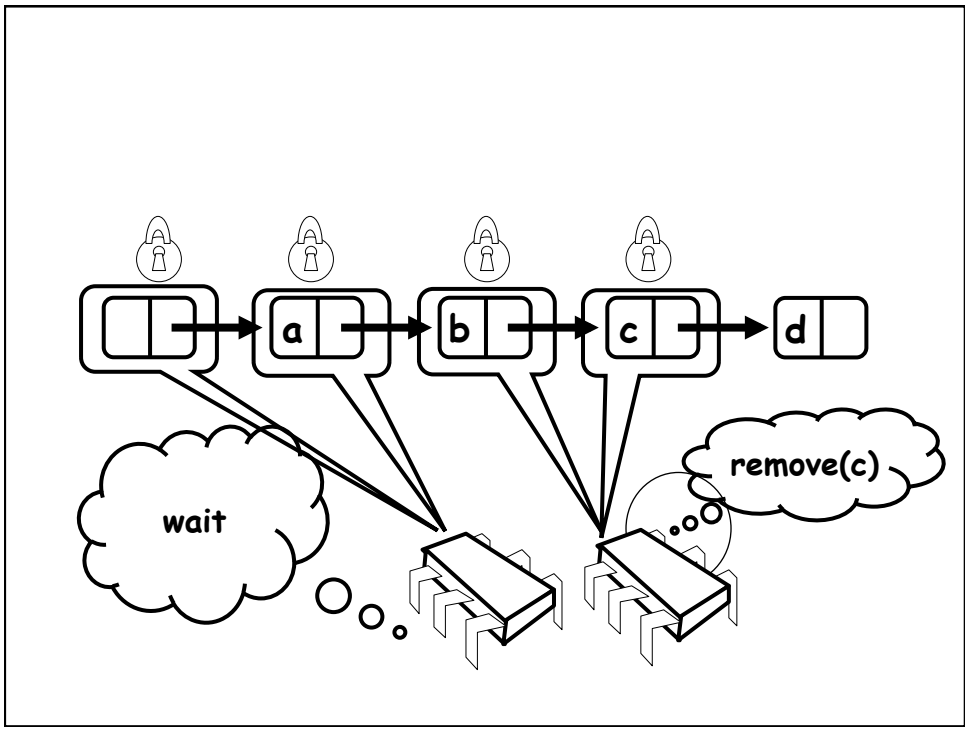

a
b
c
d
remove(b)
remove(c)

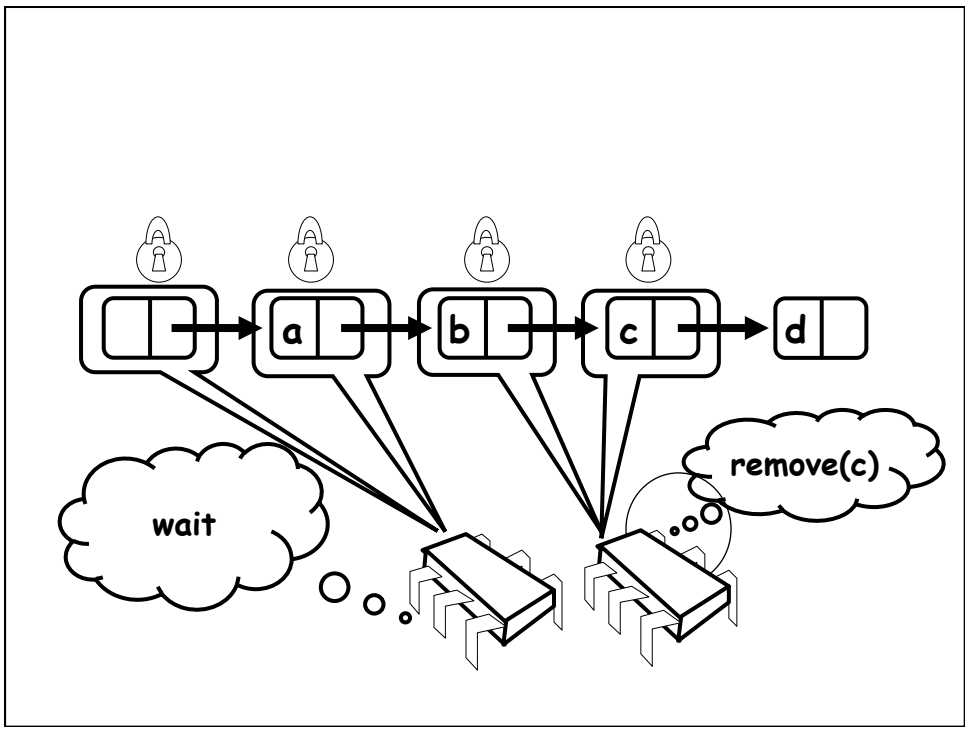

a
b
c
d
remove(b)
remove(c)

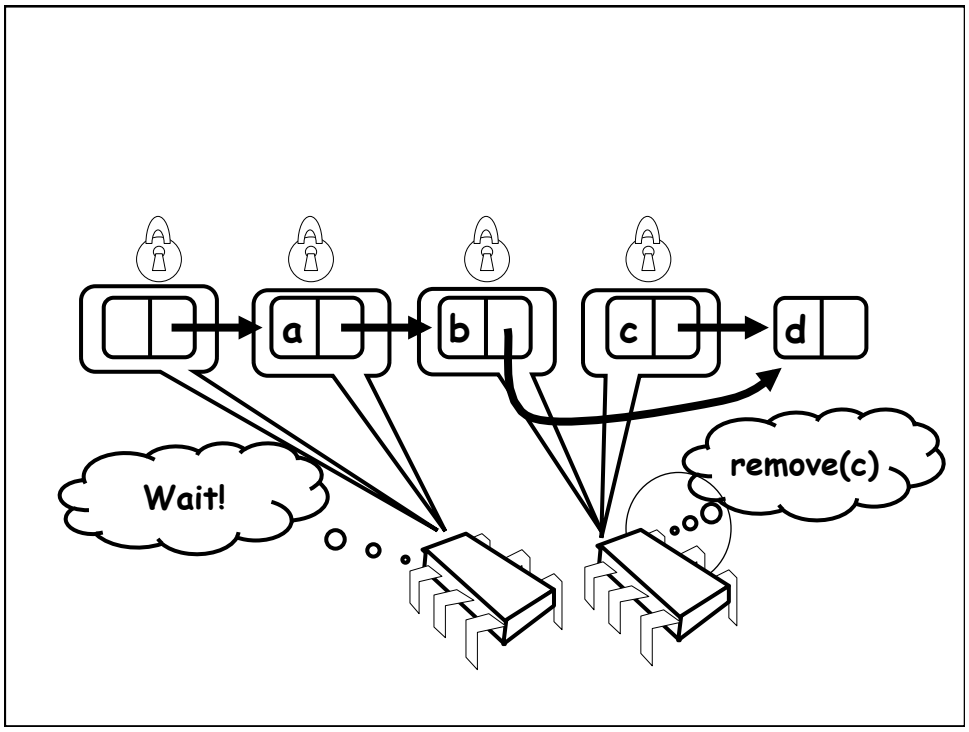

a
b
c
d
remove(b)
remove(c)

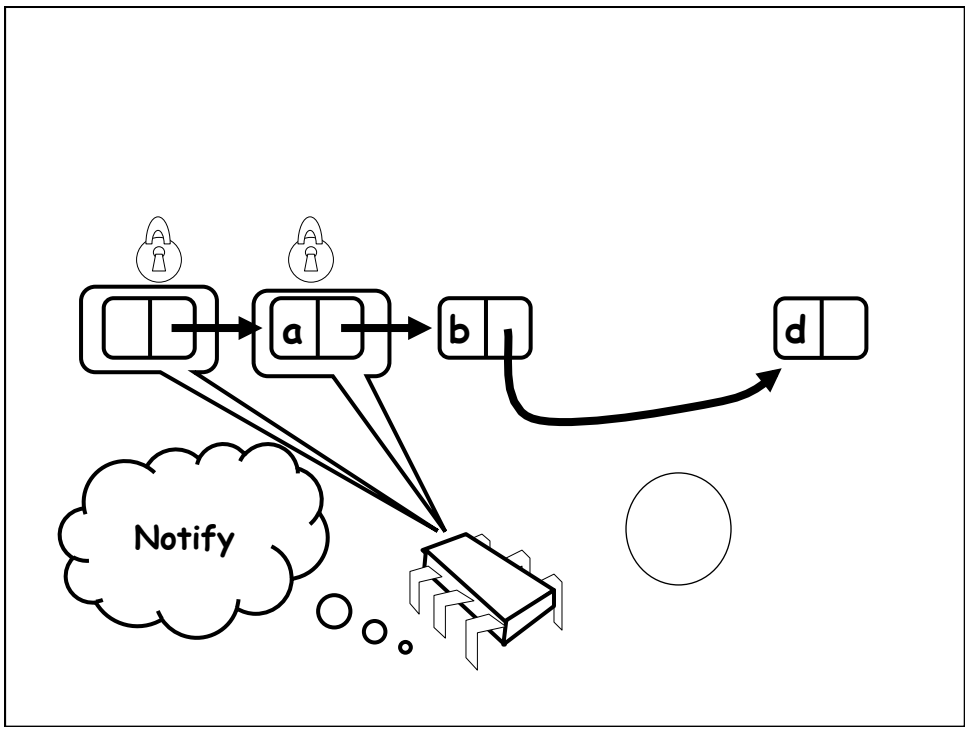

a
b
c
d
remove(b)
remove(c)

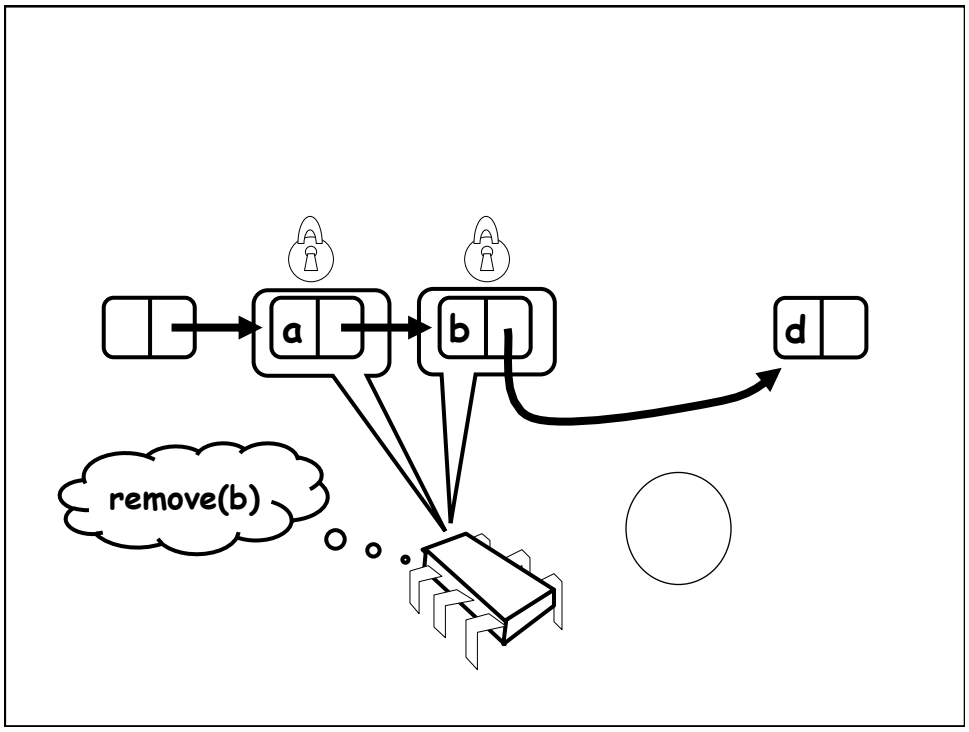

a
b
c
d
remove(b)
remove(c)

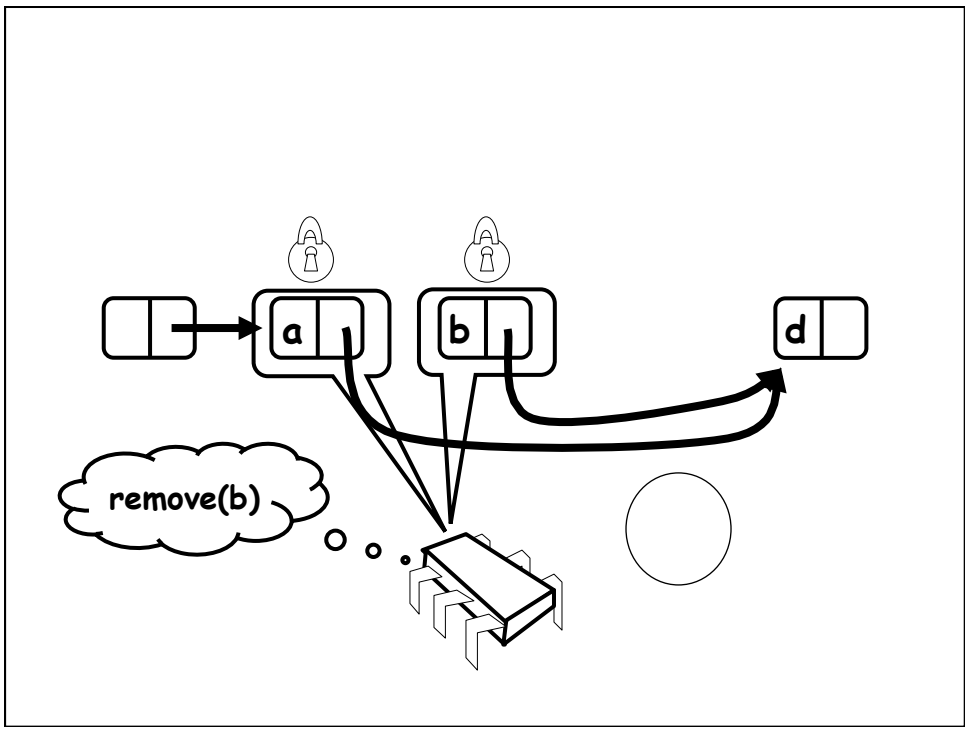

a
b
c
d
wait
remove(c)

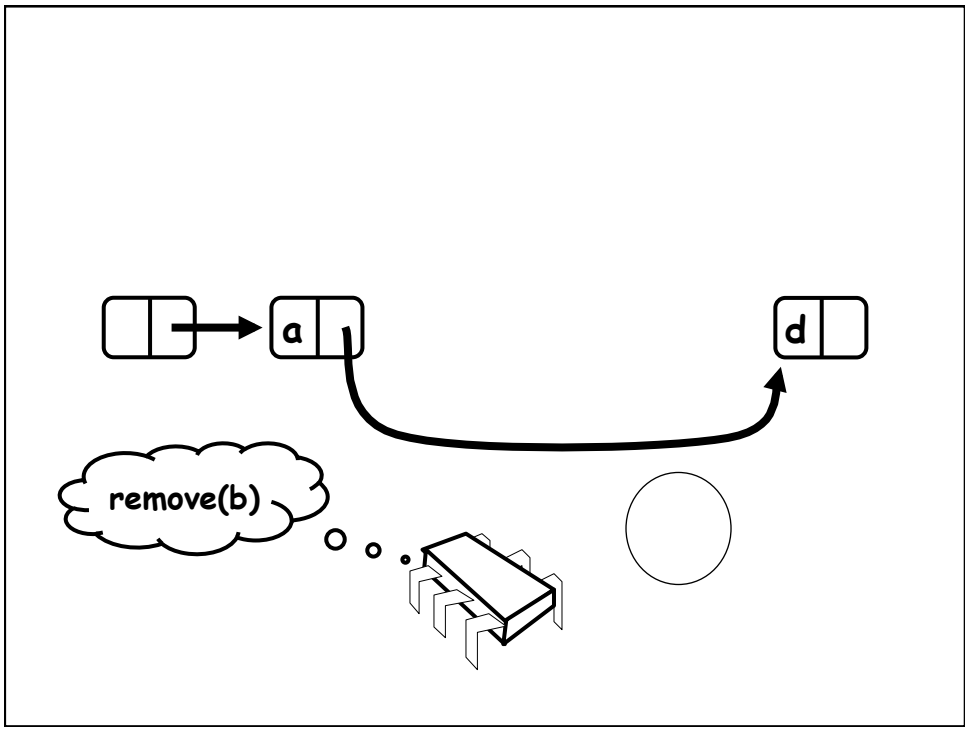

a
b
c
d
wait
remove(c)

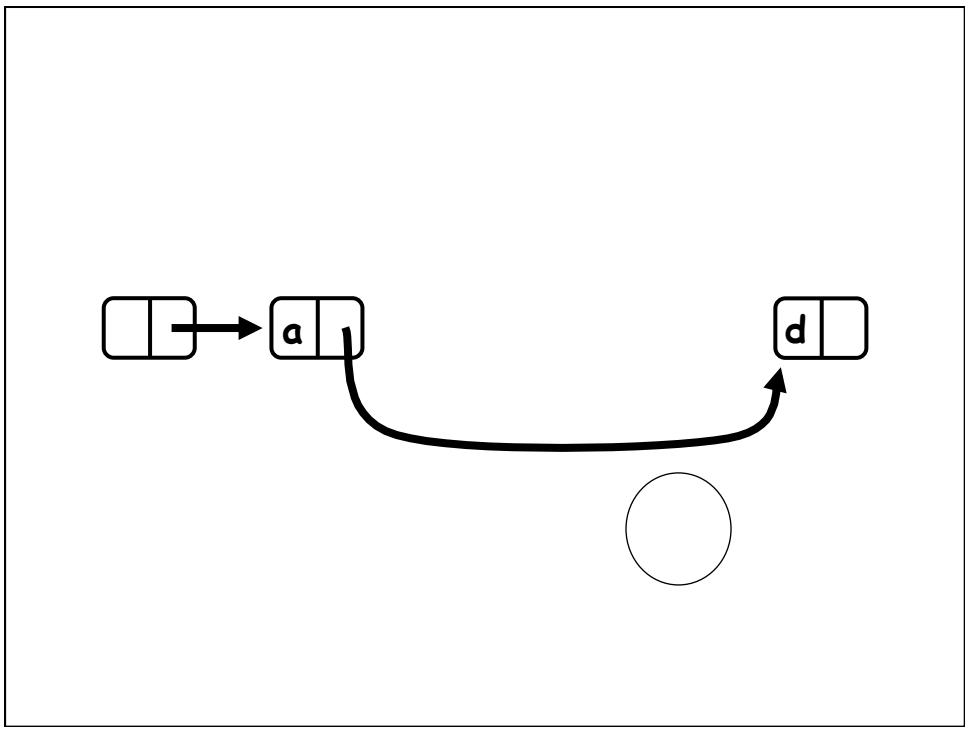

a
b
c
d
Wait!
remove(c)


a
b
d
Notify


a
b
d
remove(b)


a
b
d
remove(b)


a
d
remove(b)

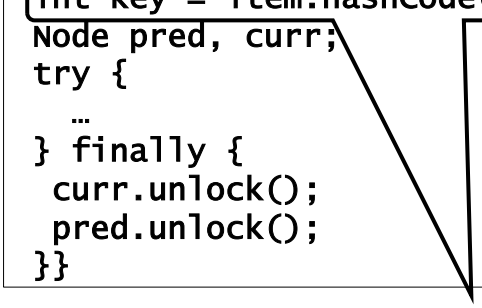

a
d

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();
 }}
```



Chivi ordinate per la ricerca

Remove method

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

Predecessor e current

Remove method

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

Rilasciare Il lock!!

Remove method

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

ops

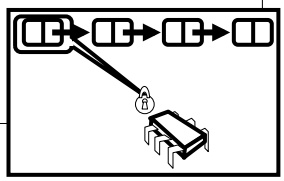
Remove method

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

Remove method

lock pred == head

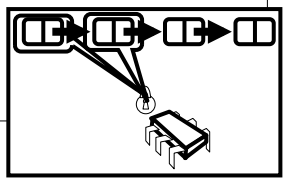
```
try {
 pred = this.head;
 pred.lock();
 curr = pred.next;
 curr.lock();
 …
} finally { … }
```



Remove method

Lock current

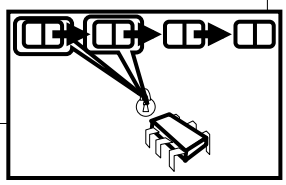
```
try {
 pred = this.head;
 pred.lock();
 curr = pred.next;
 curr.lock();
 …
} finally { … }
```



Remove method

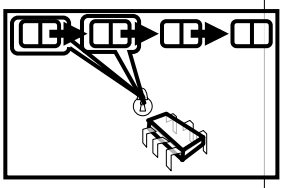
```
try {
 pred = this.head;
 pred.lock();
 curr = pred.next;
 curr.lock();
 …
} finally { … }
```

Muoversi sulla lista



Remove: searching

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



Funziona?

- Per eliminare il nodo e
  - Effettuare il lock su e
  - Deve esssre presente il lock sul predecessore del nodo e
- Se questo avviene puo' essere eliminato

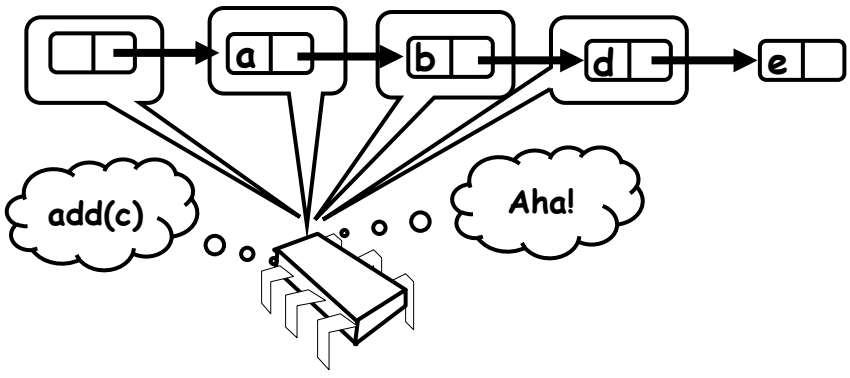
Inserzione

- Add e
  - lock predecessor
  - lock successor

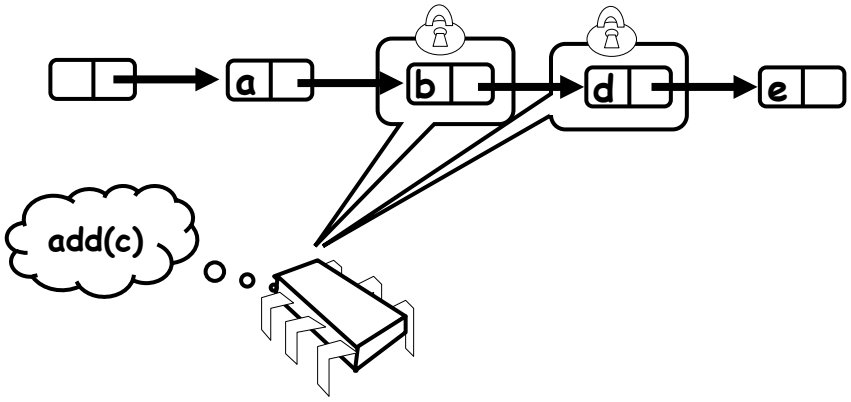
Tutto bene?

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

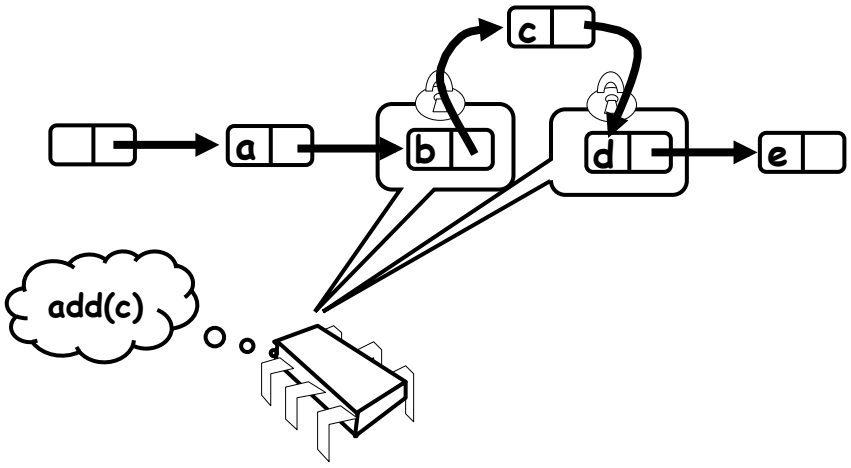
Optimistic: scorrere senza fare Lock



Optimistic

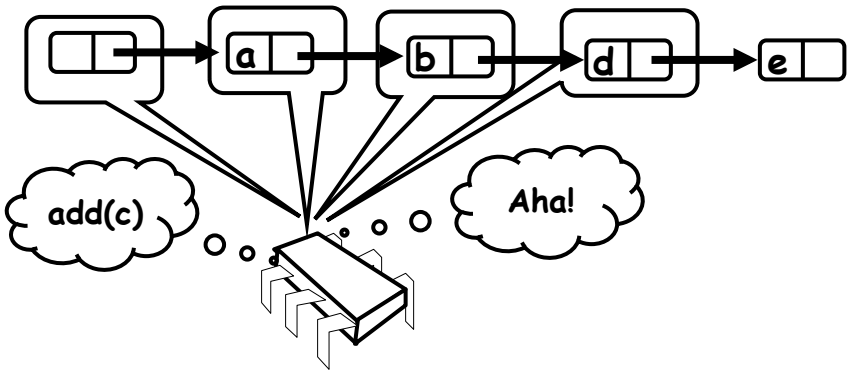


Optimistic

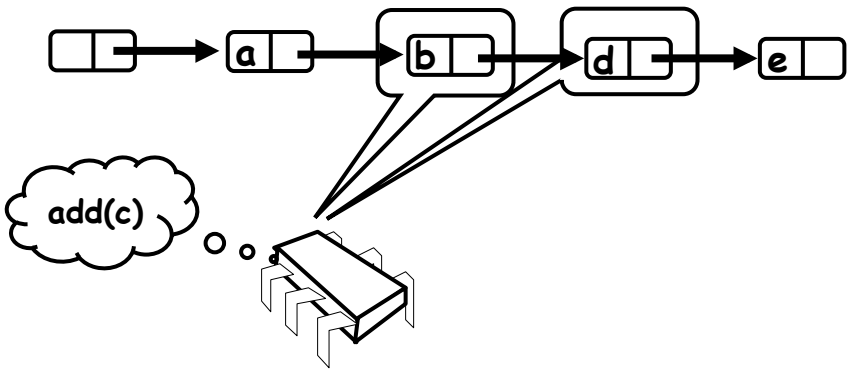


74

Cosa potrebbe andare storto?

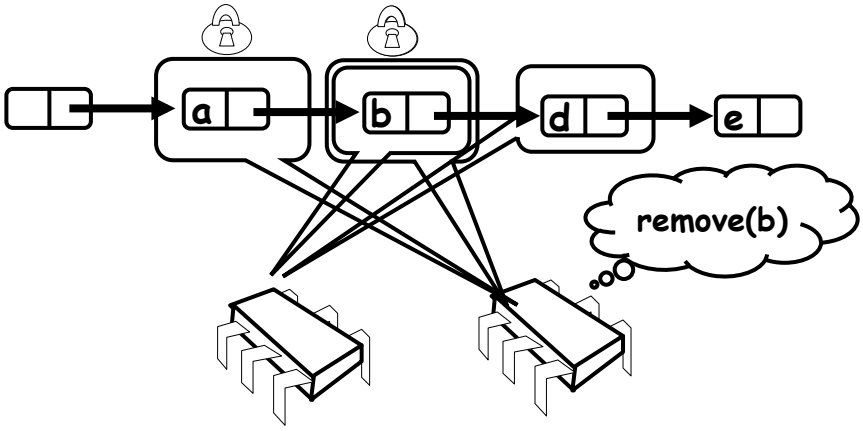


Cosa potrebbe andare storto?



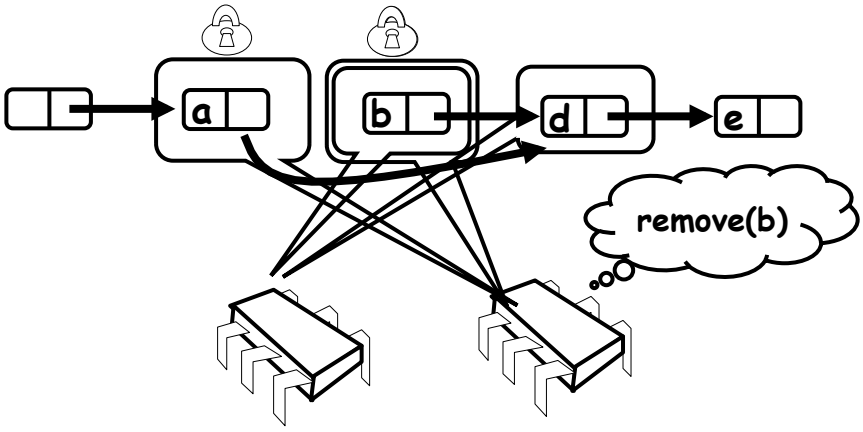
76

Cosa potrebbe andare storto?



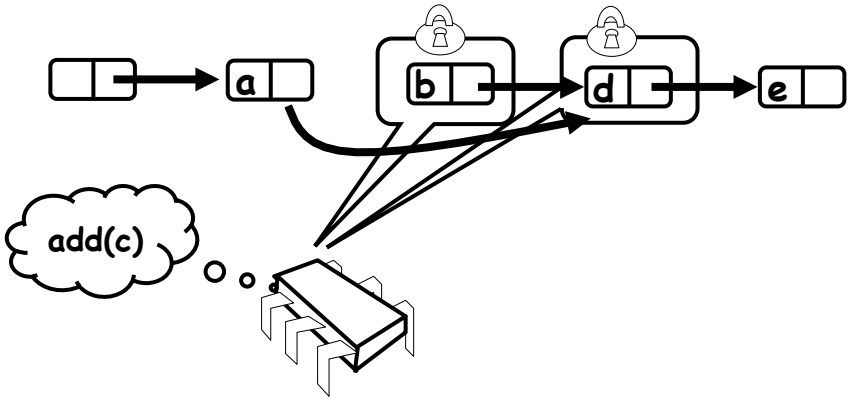
77

Cosa potrebbe andare storto?



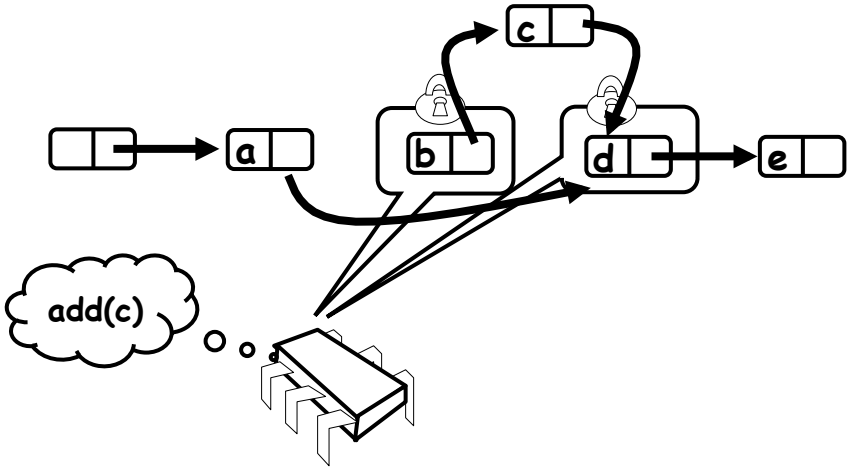
78

Cosa potrebbe andare storto?



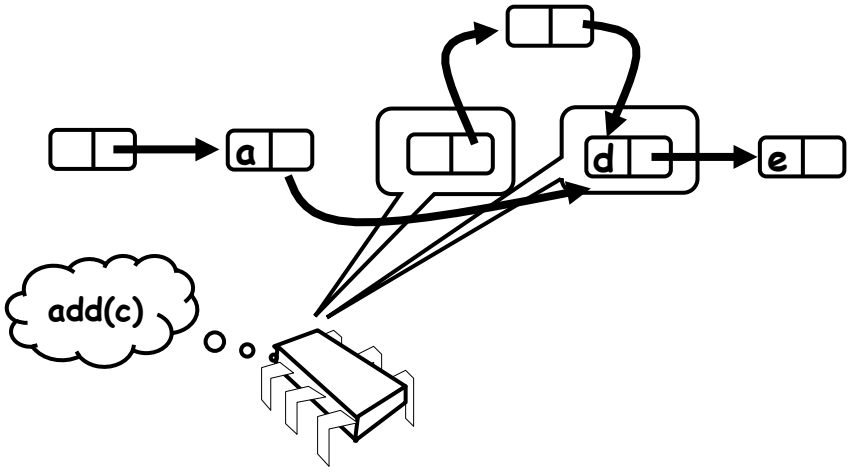
79

Cosa potrebbe andare storto?

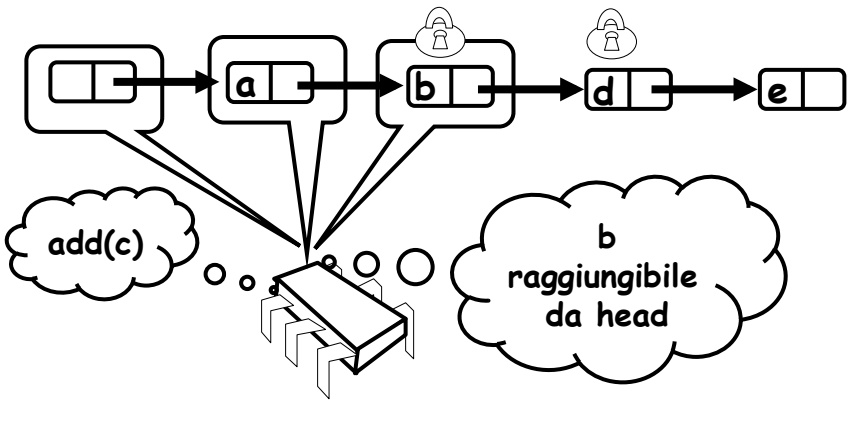


80

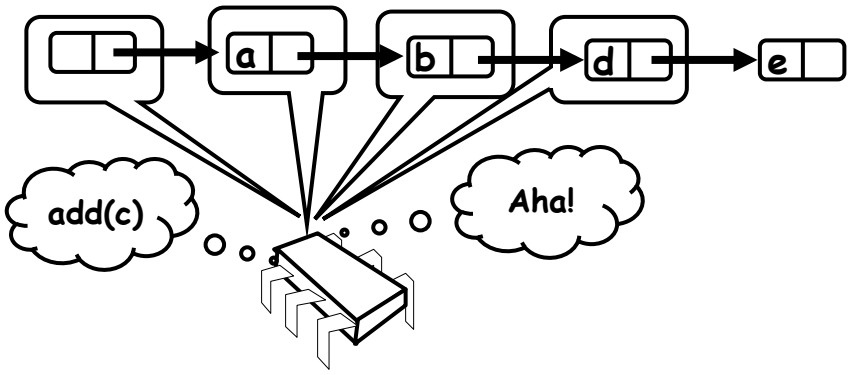
Cosa potrebbe andare storto?



81

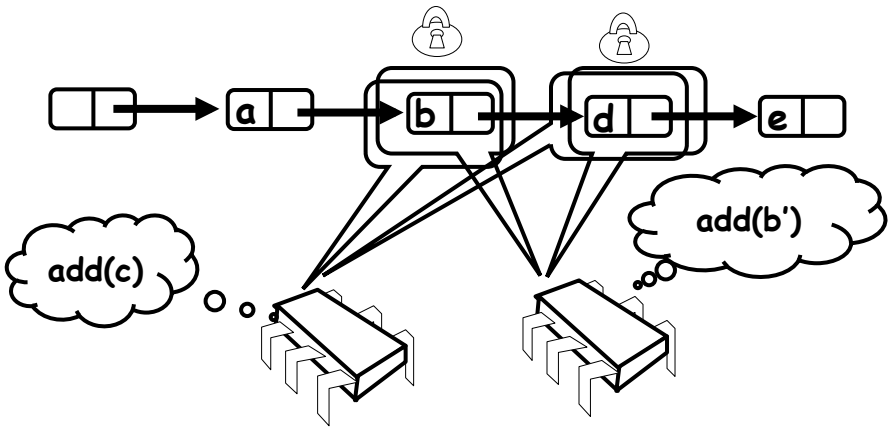


Ancora problemi

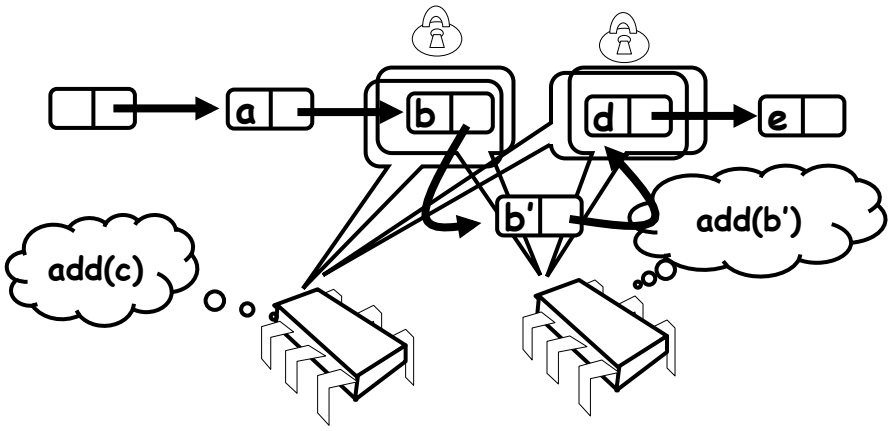


83

Ancora problemi

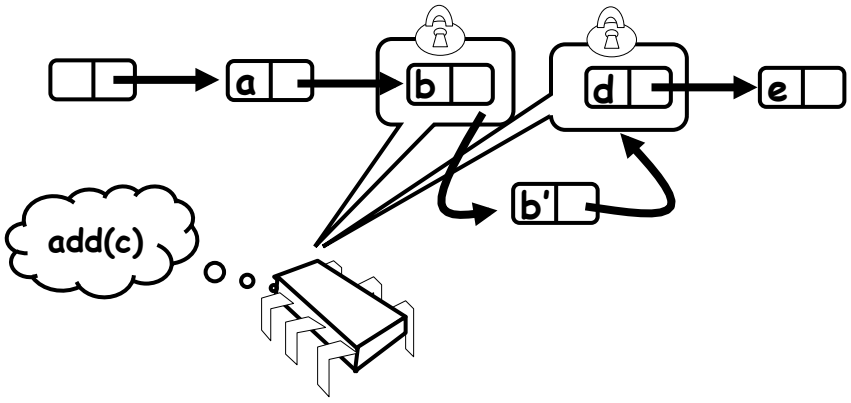


Ancora problemi



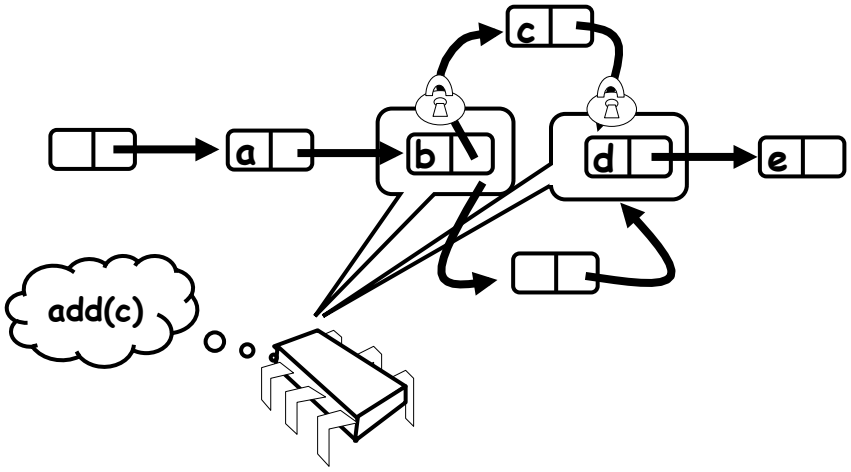
85

Ancora problemi

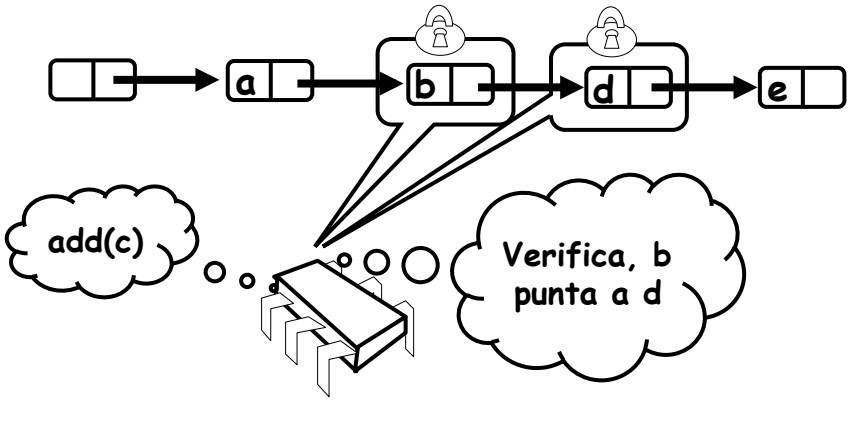


86

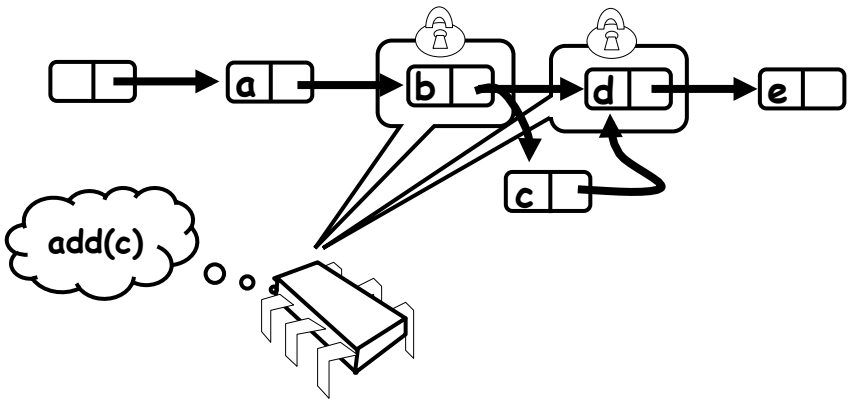
Ancora problemi



87



Punto di linearizzazione

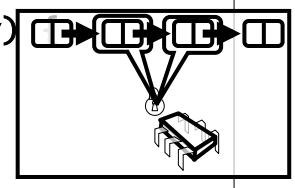


Verifica

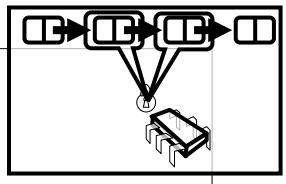
```
private boolean
 validate(Node pred,
          Node curr) {
 Node node = head;
 while (node.key <= pred.key) {
  if (node == pred)
   return pred.next == curr;
  node = node.next;
 }
 return false;
}
```

```
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;
}
```

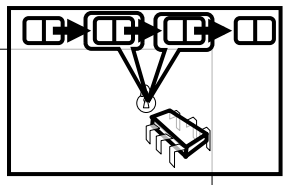
Predecessor & current



```
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;
}
```

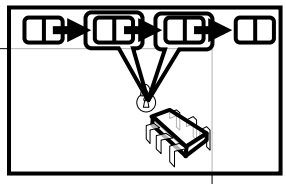


Inizializzazione



```
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;
}
```

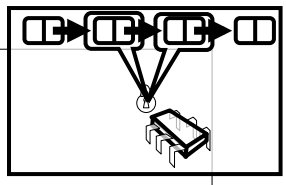
Ricerca



```
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;
}
```

Raggiungibilita'

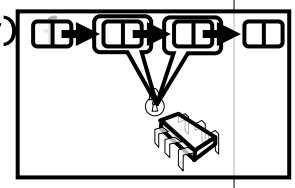
```
private boolean
 validate(Node pred,
          Node curr) {
 Node node = head;
 while (node.key <= pred.key) {
  if (node == pred)
   return pred.next == curr;
  node = node.next;
 }
 return false;
}
```



Check sulla continuazione

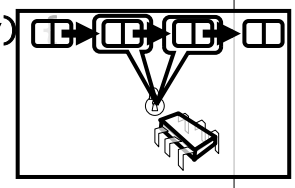
```
private boolean
 validate(Node pred,
          Node curr) {
 Node node = head;
 while (node.key <= pred.key) {
  if (node == pred)
   return pred.next == curr;
  node = node.next;
 }
 return false;
}
```

Scorrere in avanti



```
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;
}
```

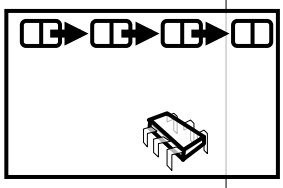
Predecessore non raggiungibile



Remove

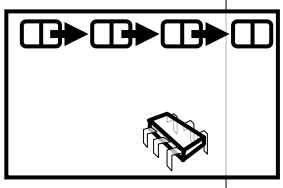
```
public boolean remove(Item item) {
 int key = item.hashCode();
 retry: while (true) {
   Node pred = this.head;
   Node curr = pred.next;
   while (curr.key <= key) {
    if (item == curr.item)
      break;
    pred = curr;
    curr = curr.next;
   } …
```

```
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;
   } …
```



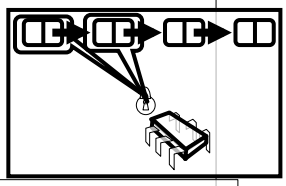
Search key

```
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;
   } …
```



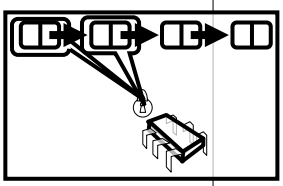
Retry: synchronization

```
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;
   } …
```



Analisi predecessor & current

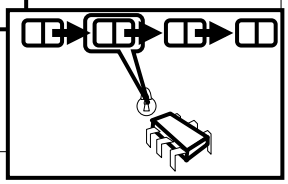
```
public boolean remove(Item item) {
 int key = item.hashCode();
 retry: while (true) {
   Node pred = this.head;
   Node curr = pred.next;
   while (curr.key <= key) {
    if (item == curr.item)
     break;
    pred = curr;
    curr = curr.next;
   } …
```



Guardia sulla chiave

```
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;
   } …
```

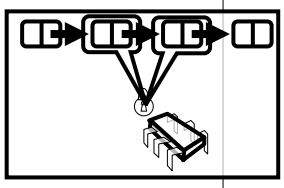
trovato



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

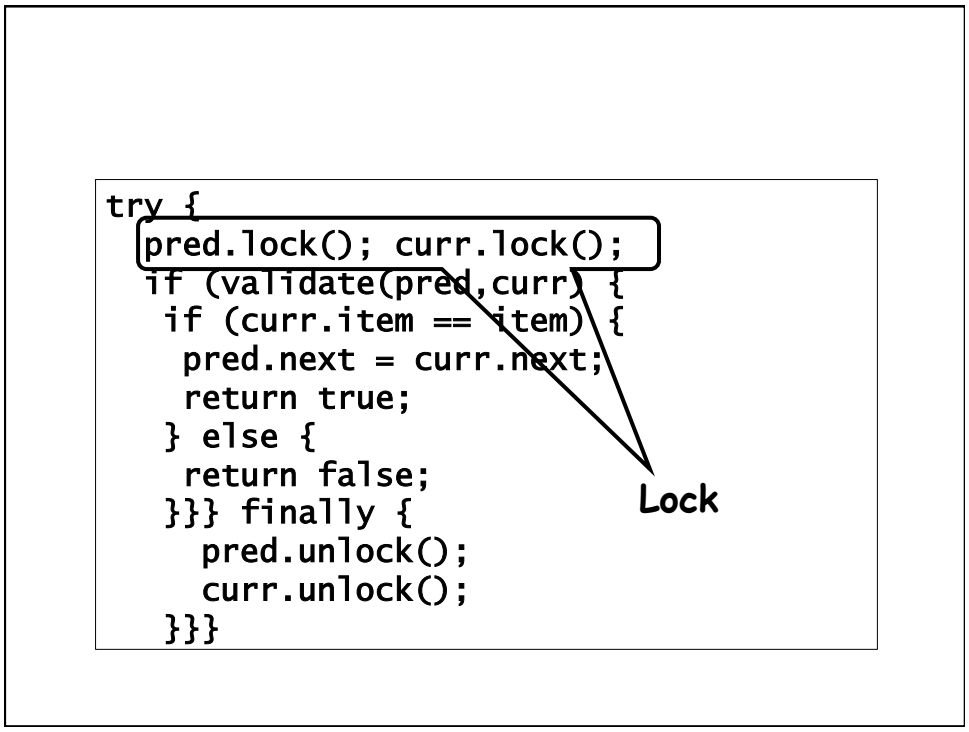


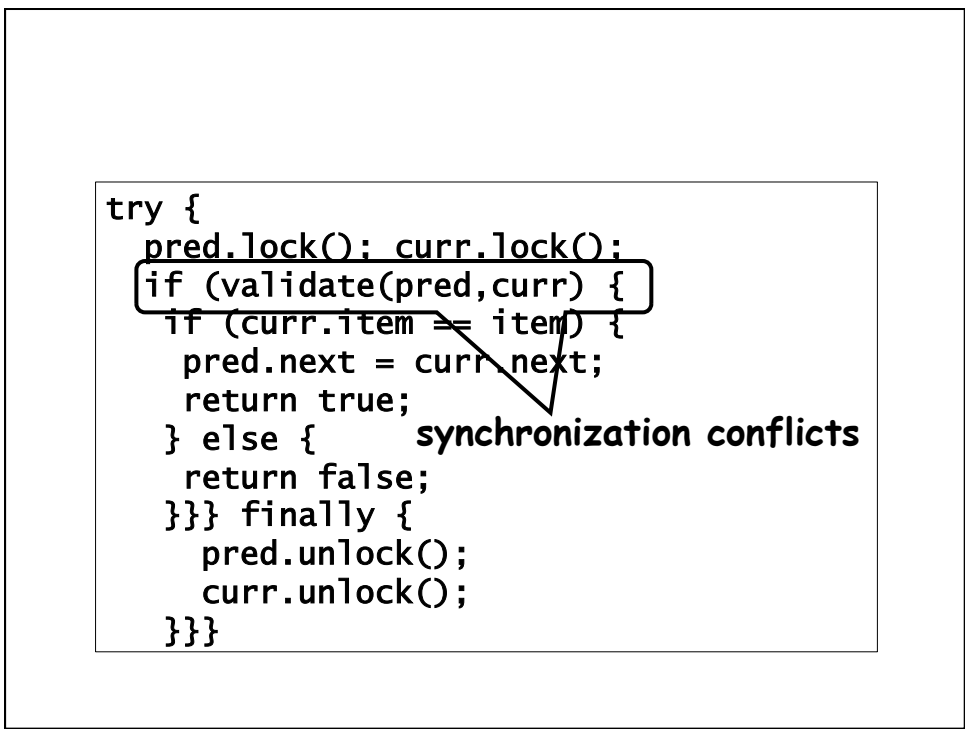
Remove Method

```
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();
   }}}
```

```
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();
   }}}
```

unlock


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


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();
}}}
synchronization conflicts

```
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();
   }}}
```

trovato, rimozione

```
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();
   }}}
```

Non trovato

Valutazione

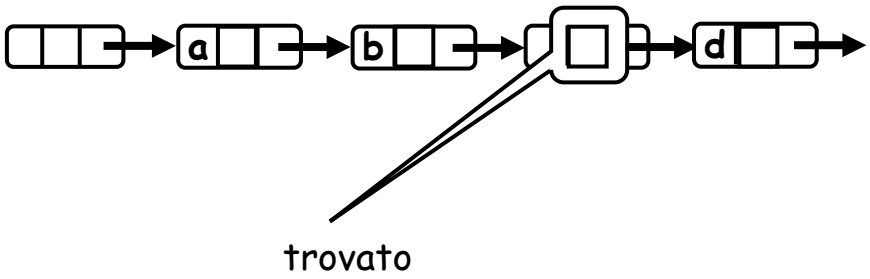
- Buona gestione dei lock
  - Performance
  - Concurrency
- Problemi
  - Scorrere la lista due volte
  - contains() comunque richiede lock

Lazy List

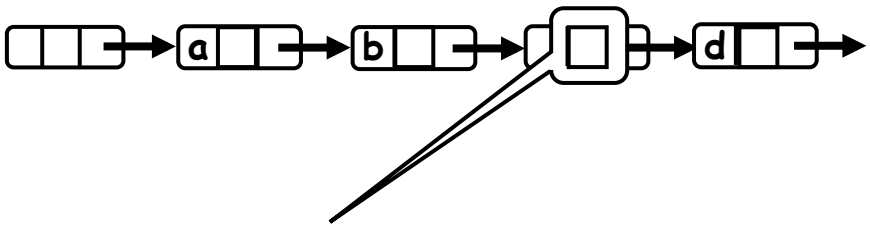
- remove()
  - Scasione della lista
  - Lock predecessor & current
- Logical delete
  - Marcare il nodo come eliminato (!!!!)
- Physical delete
  - Ridirezionare i puntatori

Lazy Removal

Lazy Removal



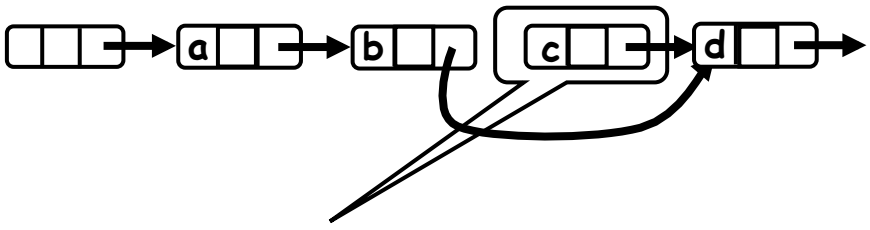
Lazy Removal



Logically deleted

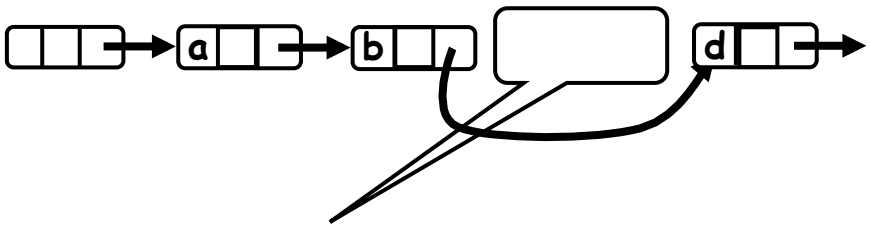
115

Lazy Removal

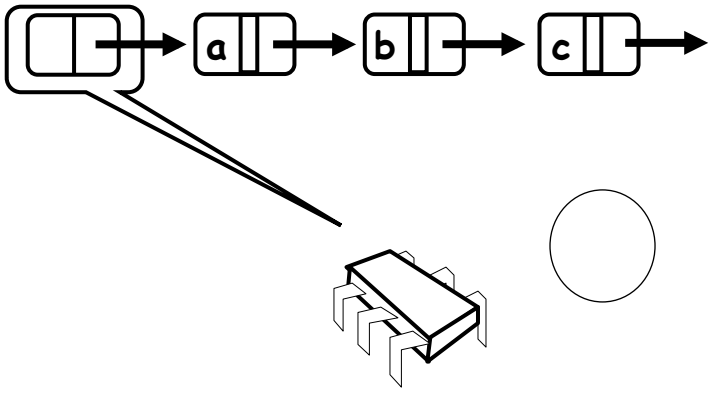


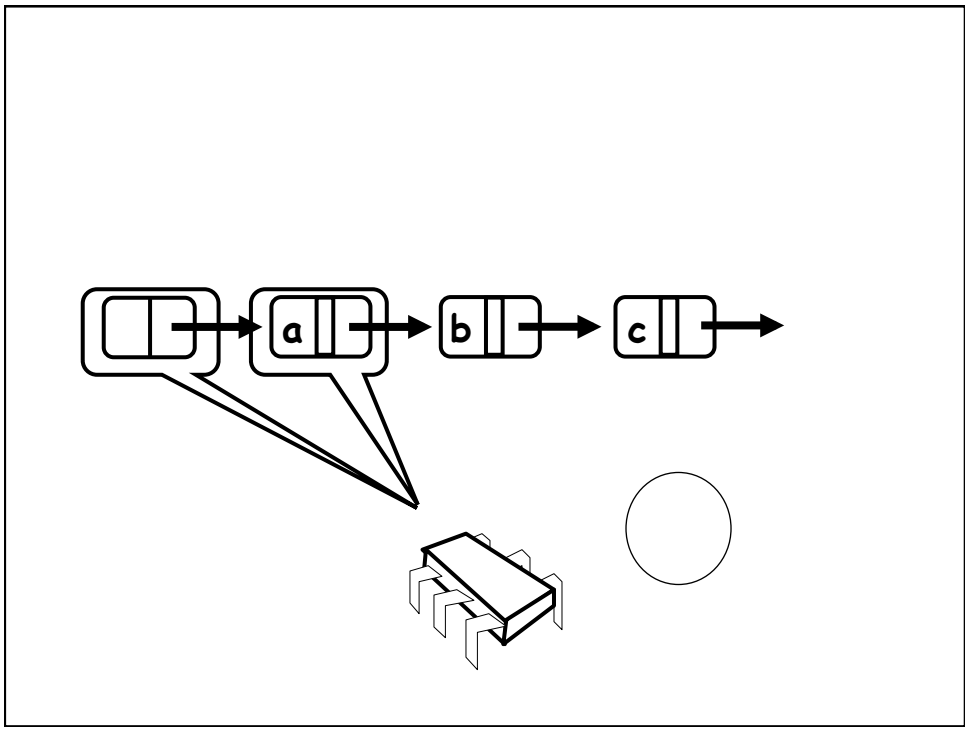
Physically deleted

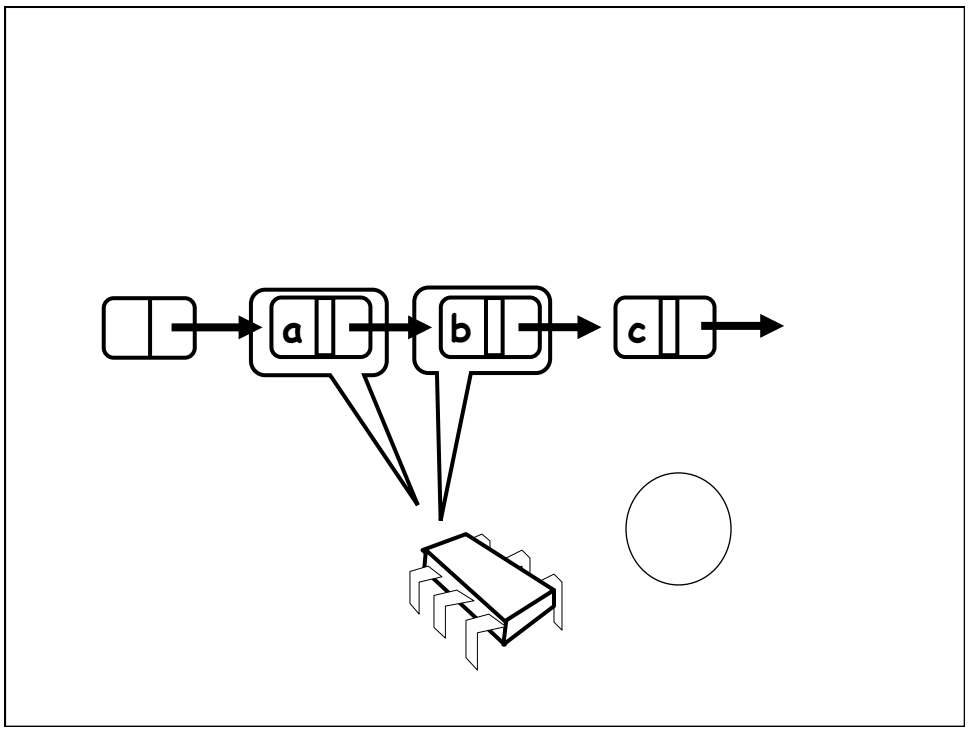
Lazy Removal

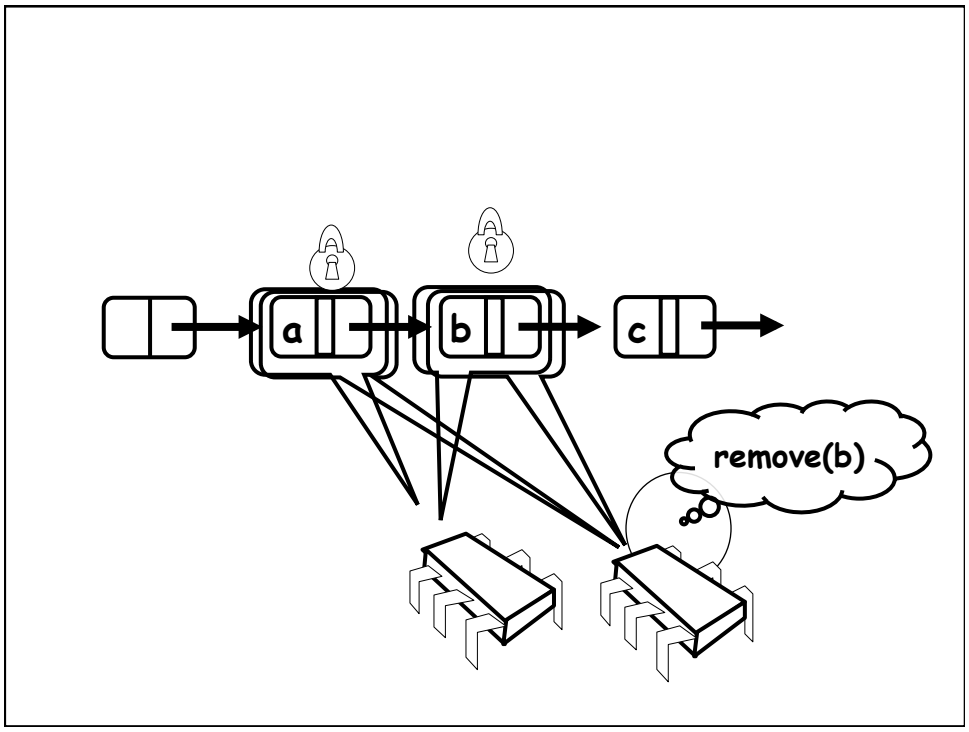


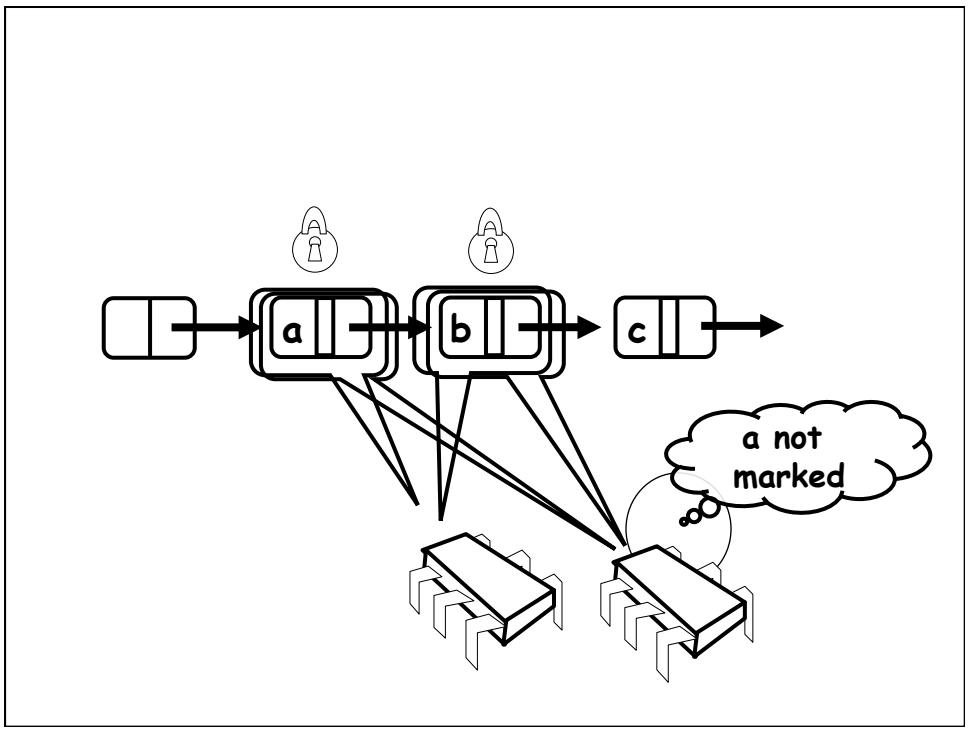
Physically deleted

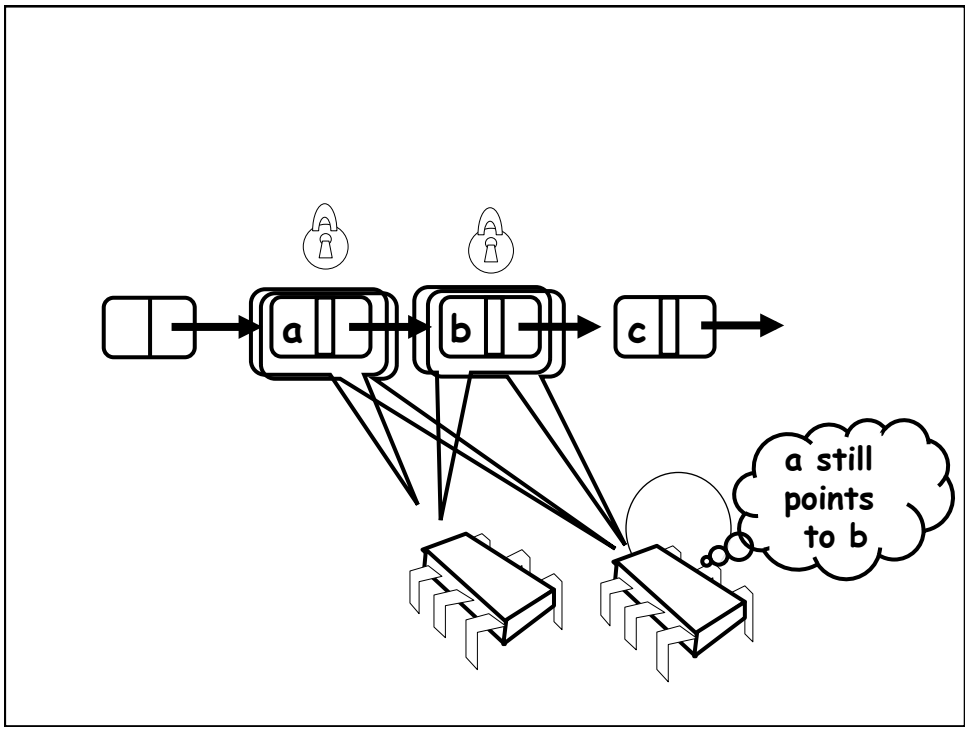


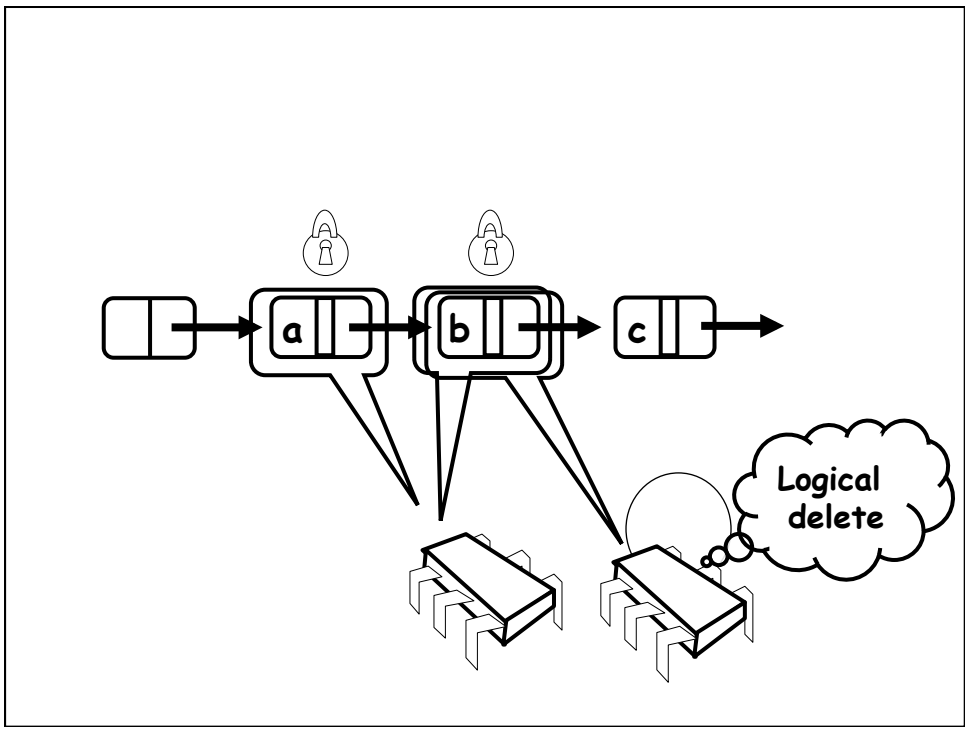

a
b
c

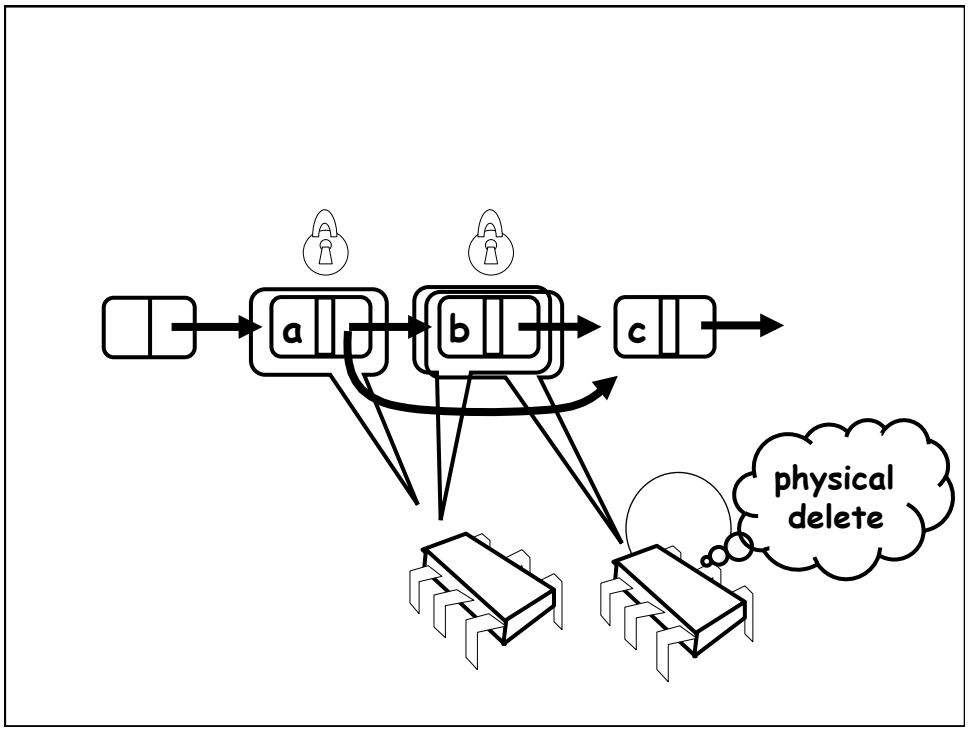

a
b
c

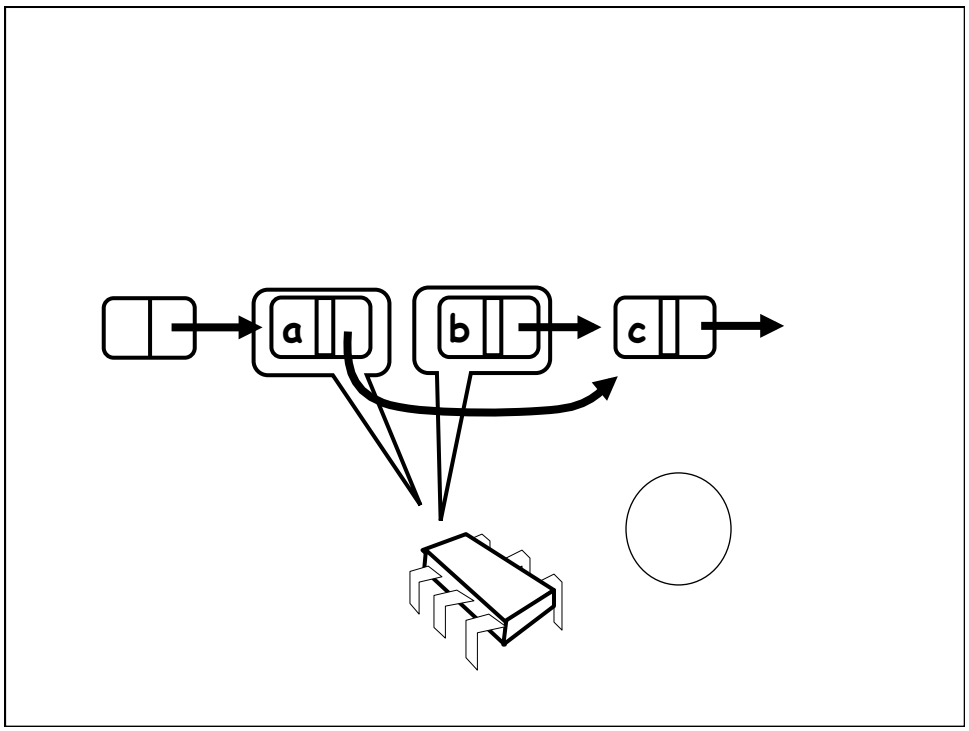

a
b
c
remove(b)


a
b
c
a not
marked


a
b
c
a still
points
to b


a
b
c
Logical
delete

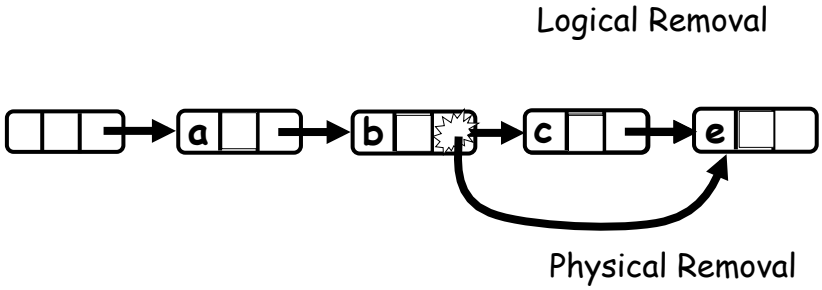

a
b
c
physical
delete


a
b
c

Abstraction Function

- alpha(head) =
  - { x | esiste a tale che
    - a raggiungibile da head e
    - a.item = x e
    - a non marcato
  - }

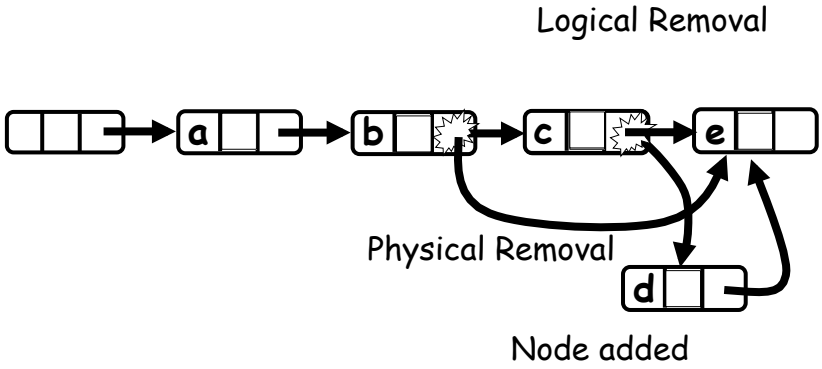
Lock-free Lists



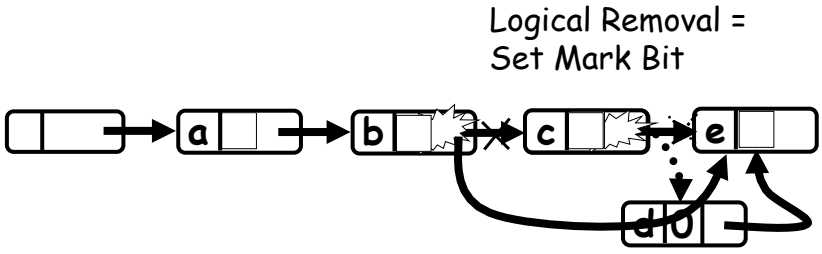
Non basta

128

Problema



Bit di marcatura e puntatori



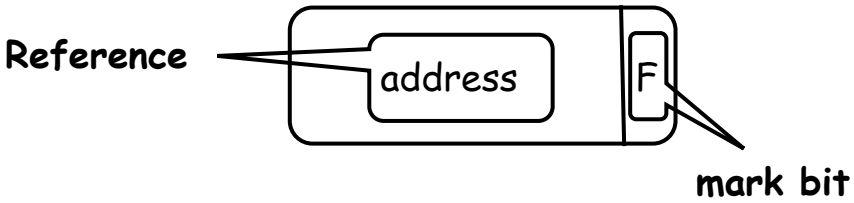
AtomicMarkableReference

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

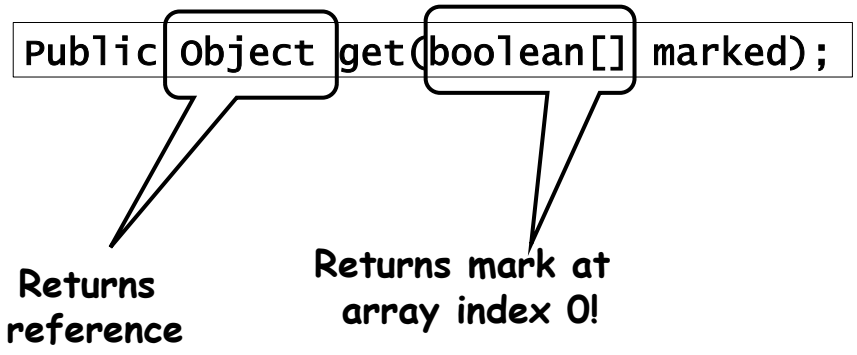


Extracting Reference & Mark

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

Extracting Reference & Mark

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



"To Lock or Not to Lock"

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