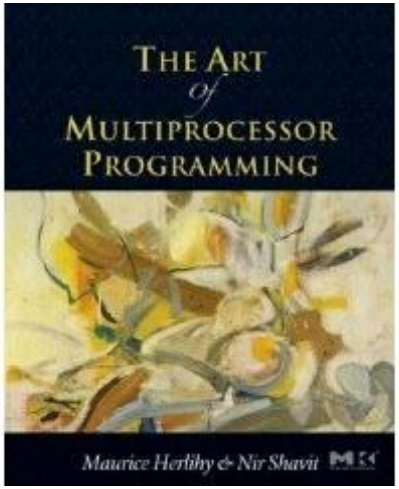
Linked Lists: Locking, Lock-Free, and Beyond …



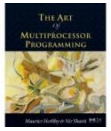
Companion slides for

The Art of Multiprocessor Programming

by Maurice Herlihy & Nir Shavit

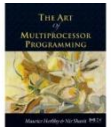
Linked List

- Illustrate these patterns …
- Using a list-based Set
  - Common application
  - Building block for other apps



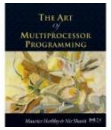
Set Interface

- Unordered collection of items



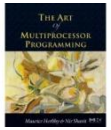
Set Interface

- Unordered collection of items
- No duplicates



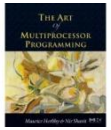
Set Interface

- Unordered collection of items
- No duplicates
- Methods
  - **`add(x)`** put **`x`** in set
  - **`remove(x)`** take **`x`** out of set
  - **`contains(x)`** tests if **`x`** in set



List-Based Sets

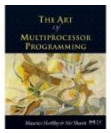
```
public interface Set<T> {
 public boolean add(T x);
 public boolean remove(T x);
 public boolean contains(T x);
}
```



List-Based Sets

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

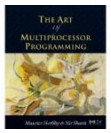
Add item to set



List-Based Sets

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

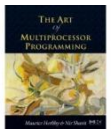
Remove item from set



List-Based Sets

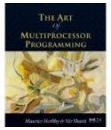
```
public interface Set<T> {
 public boolean add(T x);
 public boolean remove(T x);
 public boolean contains(T x);
}
```

Is item in set?



List Node

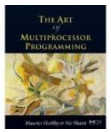
```
public class Node {
 public T item;
 public int key;
 public volatile Node next;
}
```



List Node

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

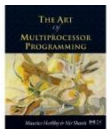
item of interest



List Node

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

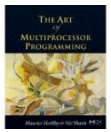
Usually hash code



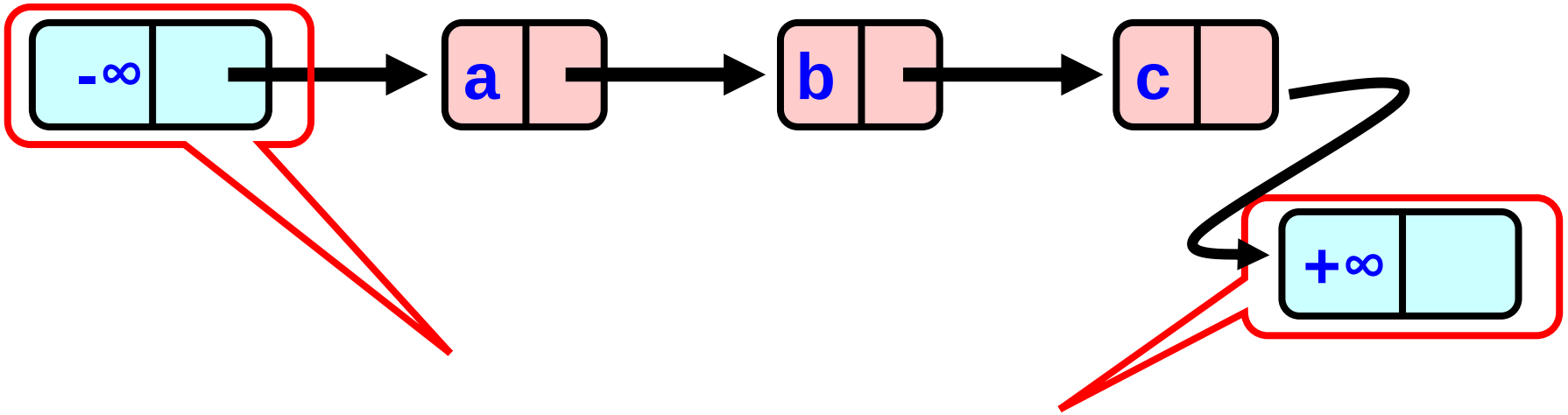
List Node

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

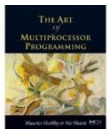
Reference to next node



The List-Based Set

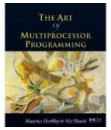


Sorted with Sentinel nodes
(min & max possible keys)



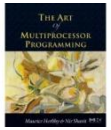
Reasoning about Concurrent Objects

- Invariant
  - Property that always holds



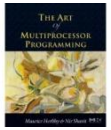
Reasoning about Concurrent Objects

- Invariant
  - Property that always holds
- Established because
  - True when object is **created**
  - Truth **preserved** by each method
    - Each **step** of each method



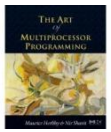
Specifically …

- Invariants preserved by
  - **`add()`**
  - **`remove()`**
  - **`contains()`**



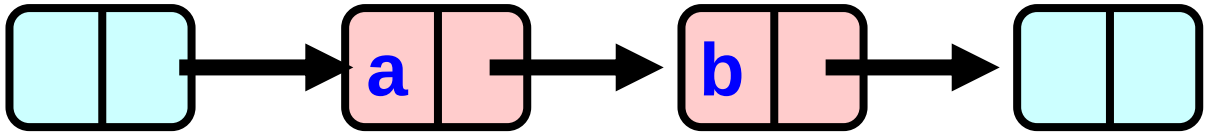
Specifically …

- Invariants preserved by
  - **`add()`**
  - **`remove()`**
  - **`contains()`**
- Most steps are trivial
  - Usually one step tricky
  - Often linearization point



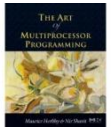
Abstract Data Types

- Concrete representation:



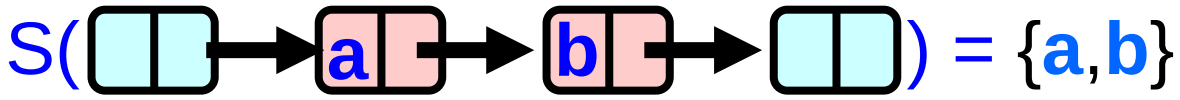
- Abstract Type:

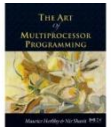
  {**a**, **b**}



Abstract Data Types

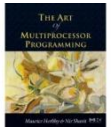
- Meaning of rep given by abstraction map





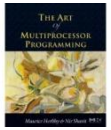
Blame Game

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



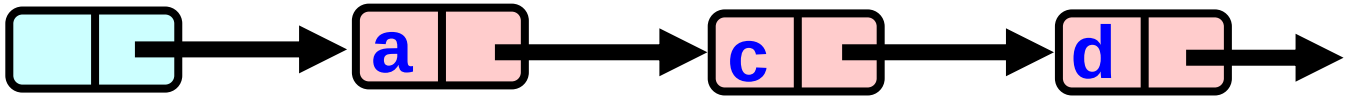
Rep Invariant (partly)

- Sentinel nodes
  - tail reachable from head
- Sorted
- No duplicates

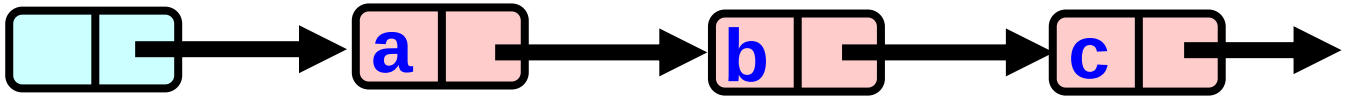


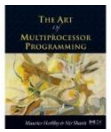
Sequential List Based Set

add()



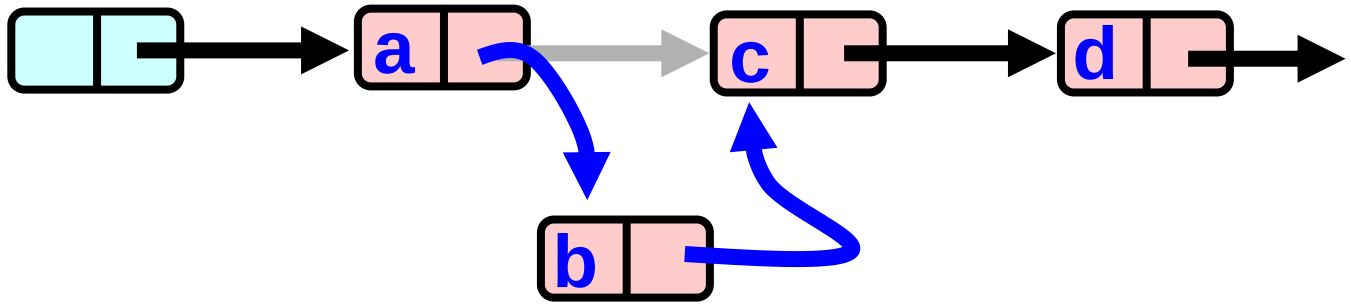
remove()



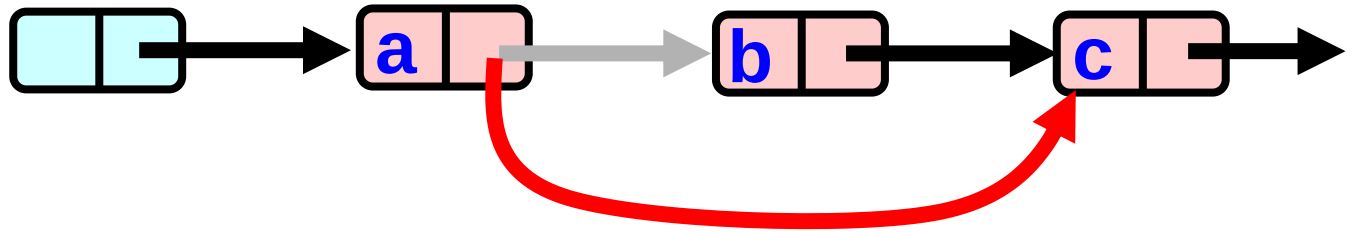


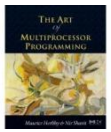
Sequential List Based Set

`add()`

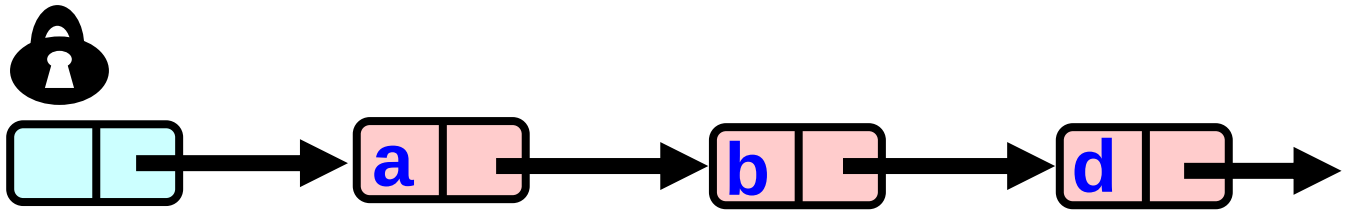


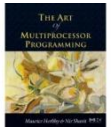
`remove()`



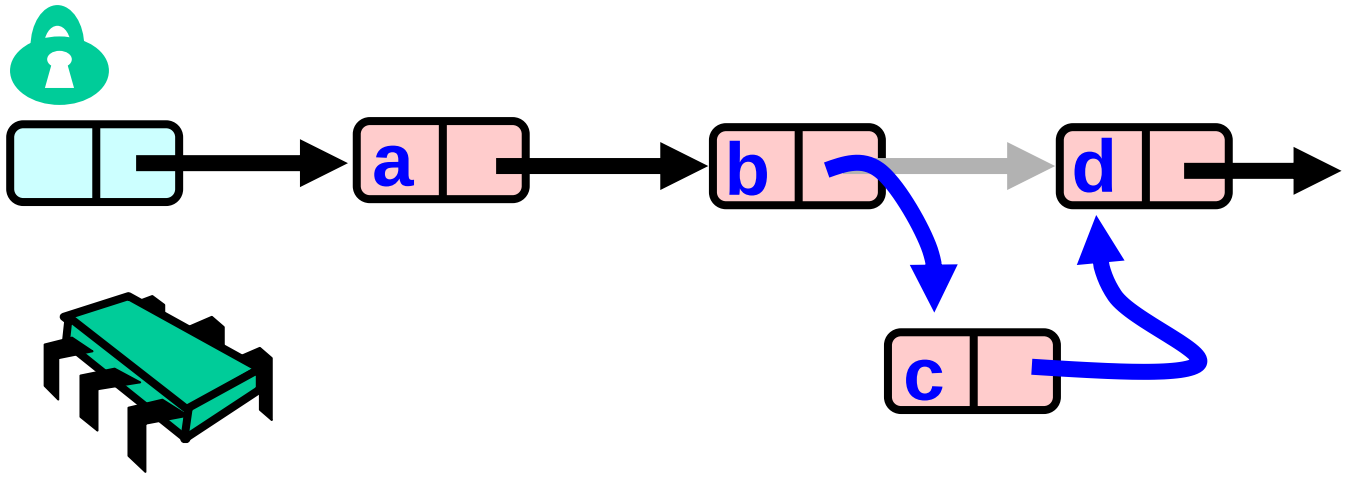


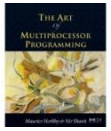
Coarse-Grained Locking



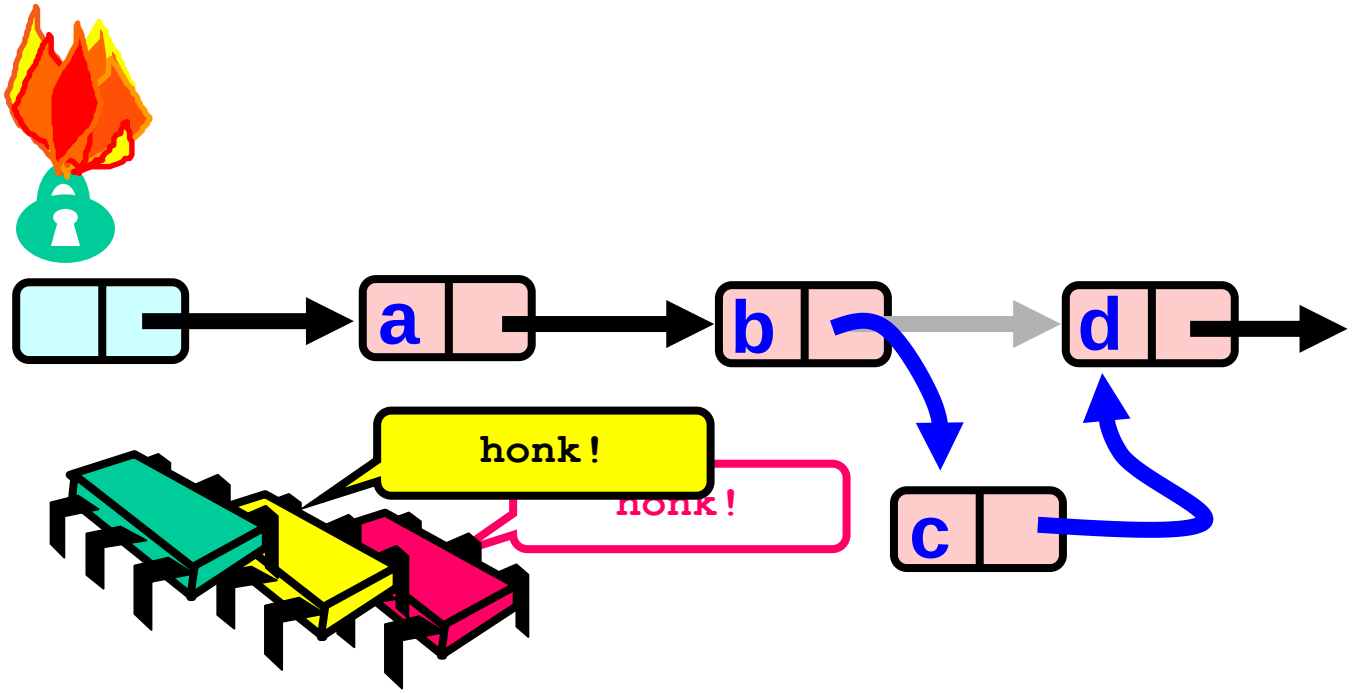


Coarse-Grained Locking

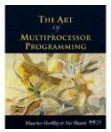




Coarse-Grained Locking

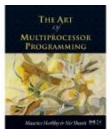


Simple but hotspot + bottleneck



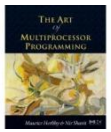
Coarse-Grained Locking

- Easy, same as synchronized methods
  - "One lock to rule them all …"
- Simple, clearly correct
  - Deserves respect!
- Works poorly with contention
  - Queue locks help
  - But bottleneck still an issue

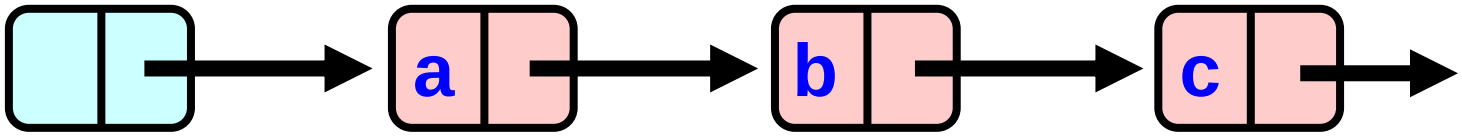


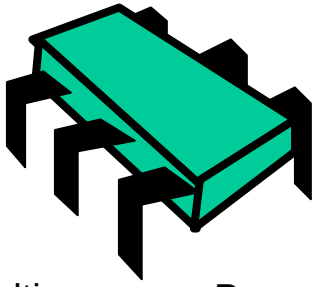
Fine-grained Locking

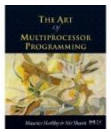
- Requires **careful** thought
  - "Do not meddle in the affairs of wizards, for they are subtle and quick to anger"
- Split object into pieces
  - Each piece has own lock
  - Methods that work on disjoint pieces need not exclude each other



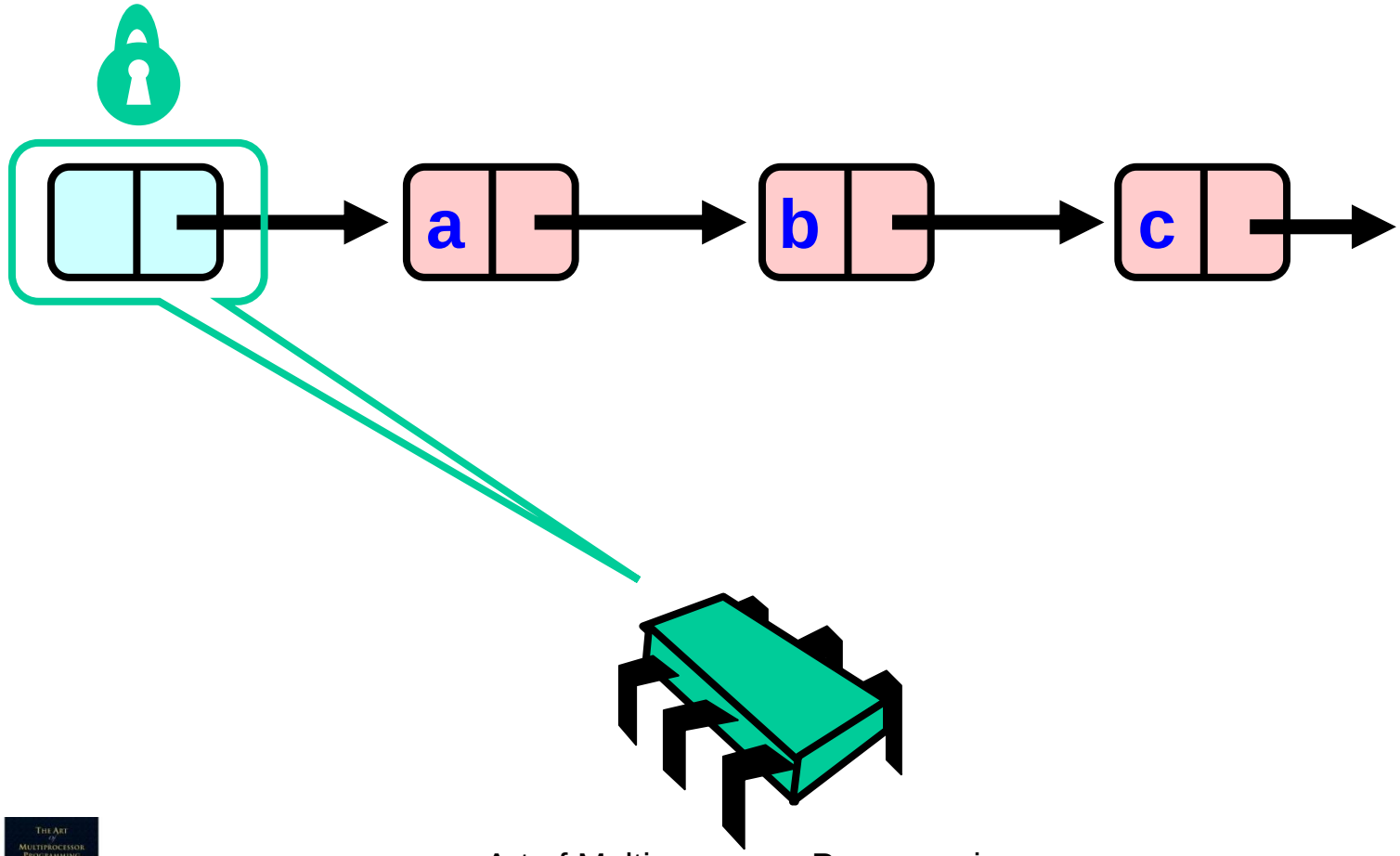
Hand-over-Hand locking

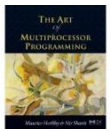




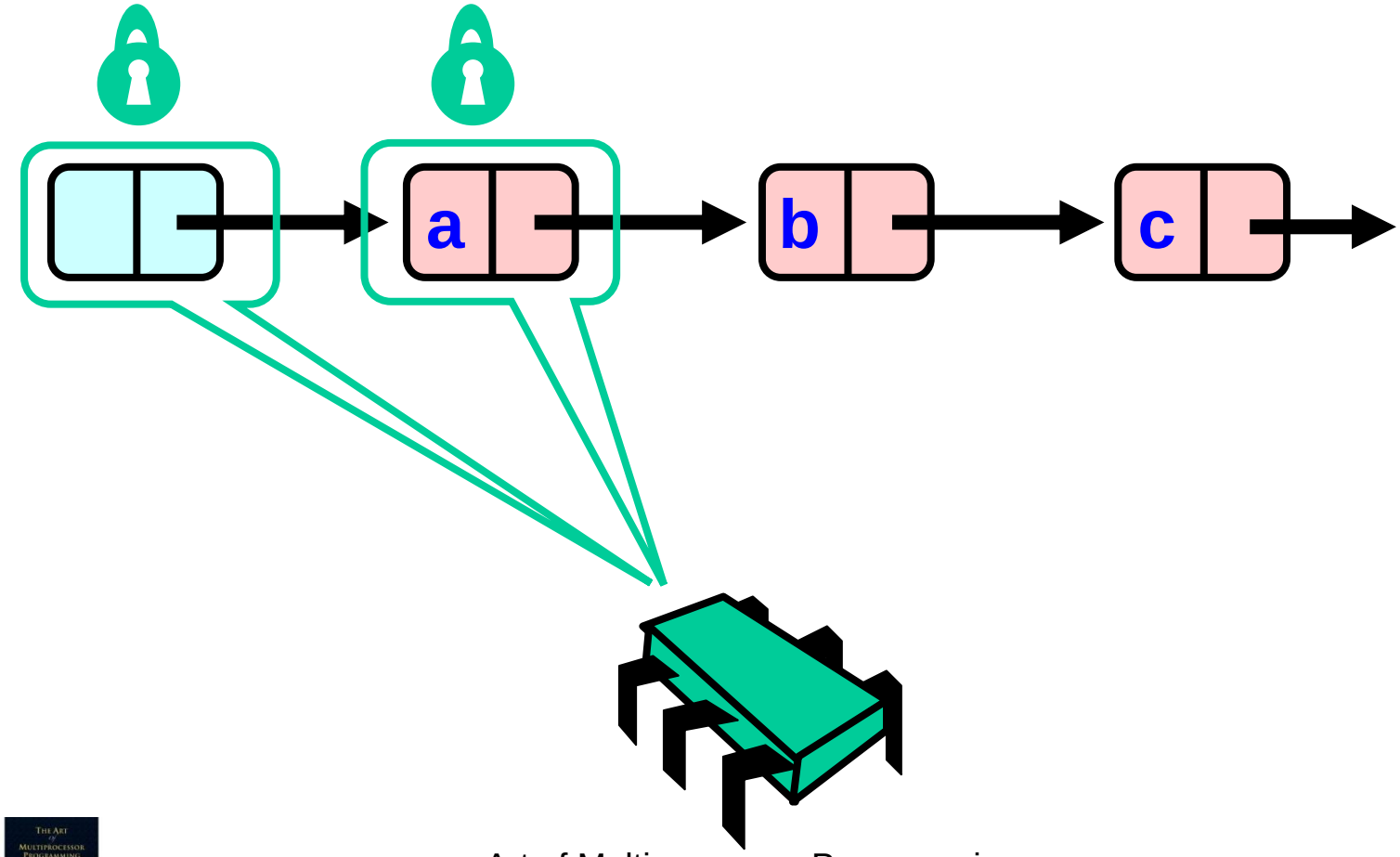


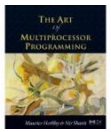
Hand-over-Hand locking



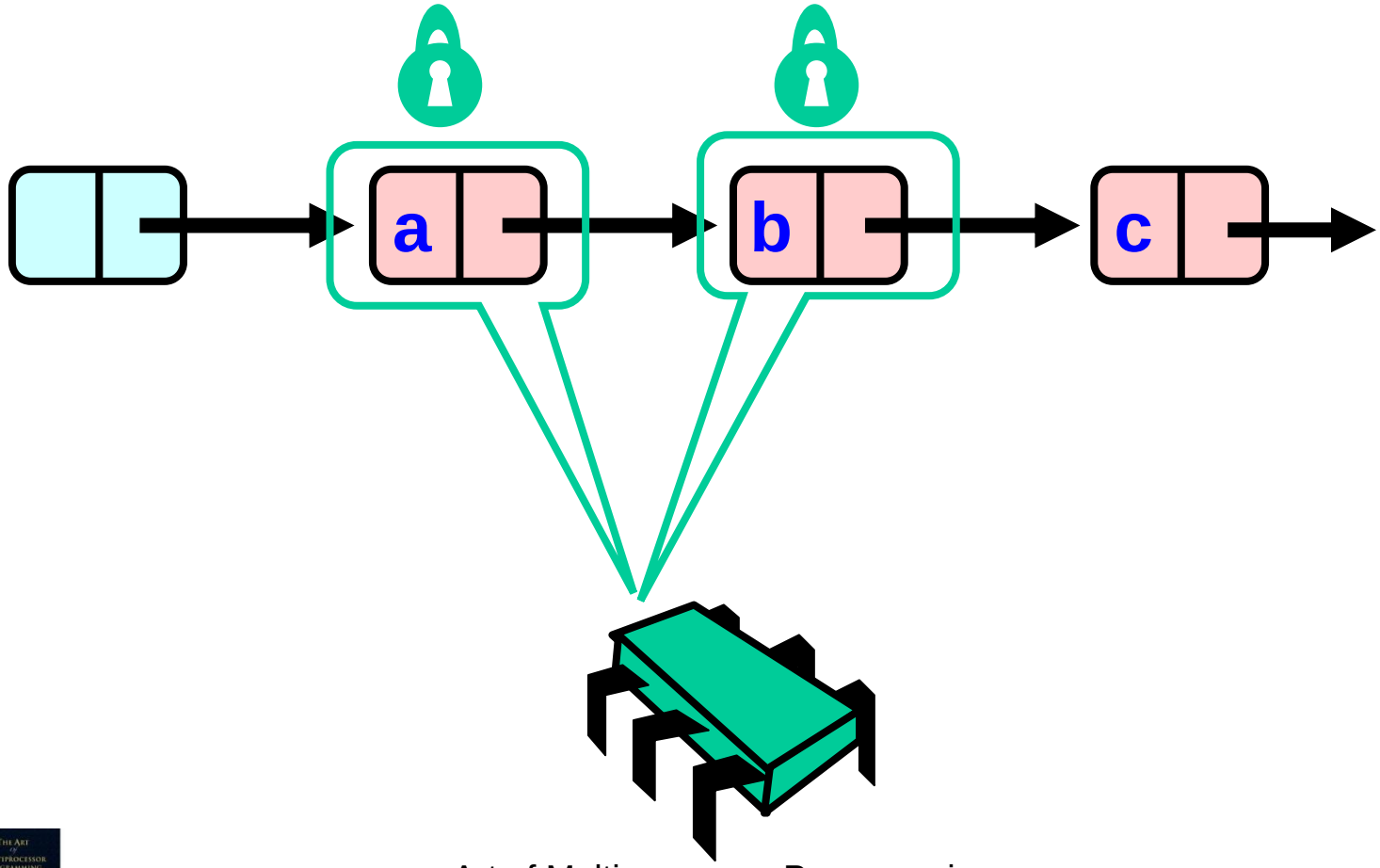


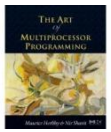
Hand-over-Hand locking



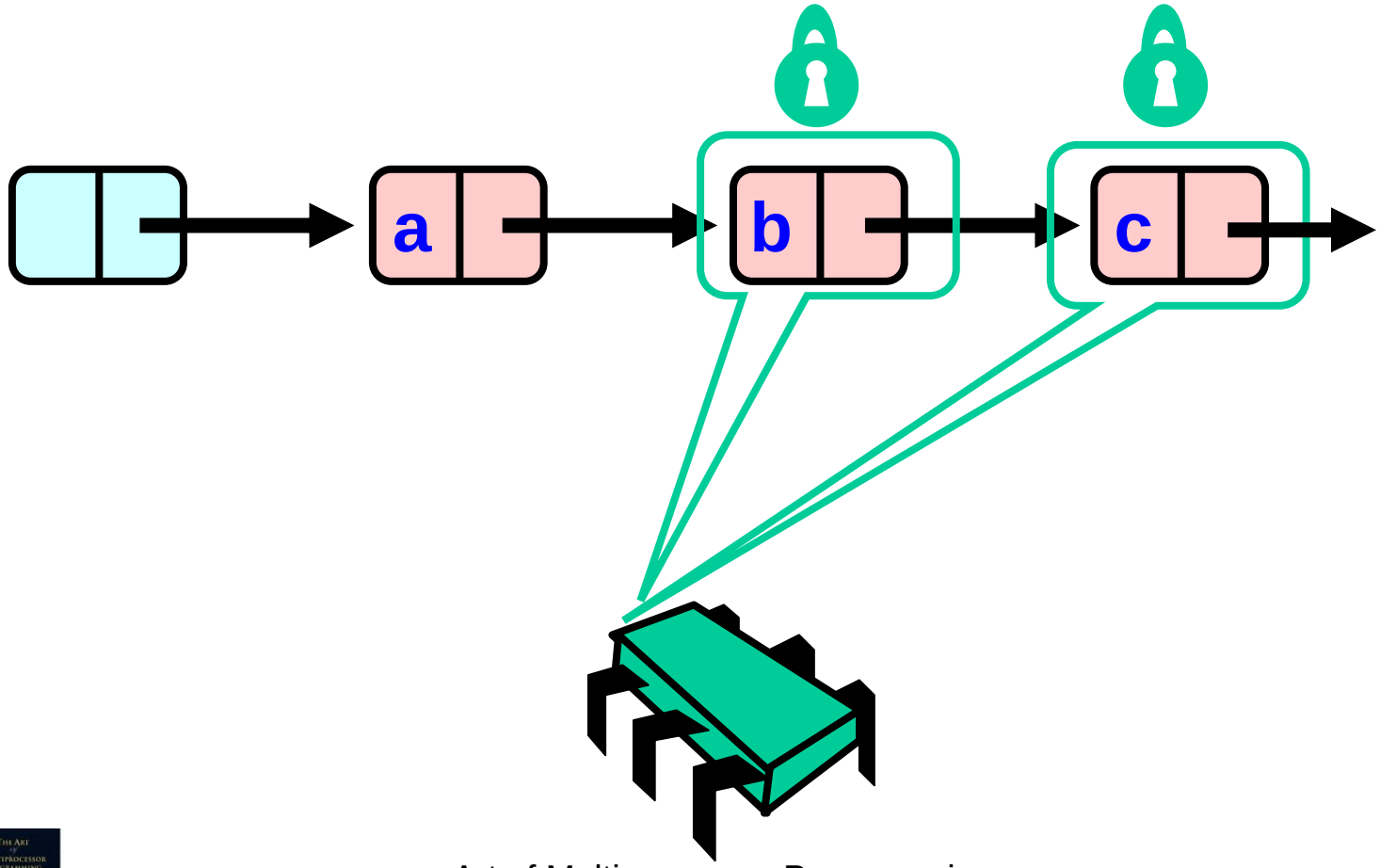


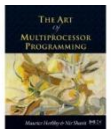
Hand-over-Hand locking



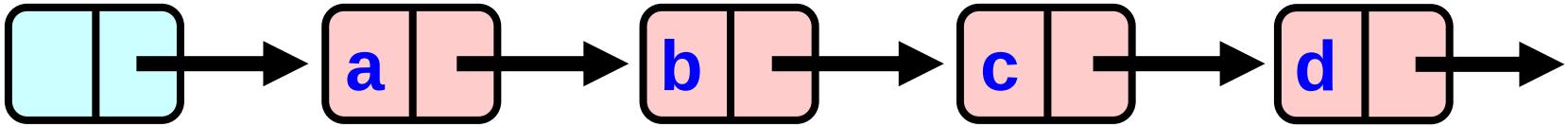


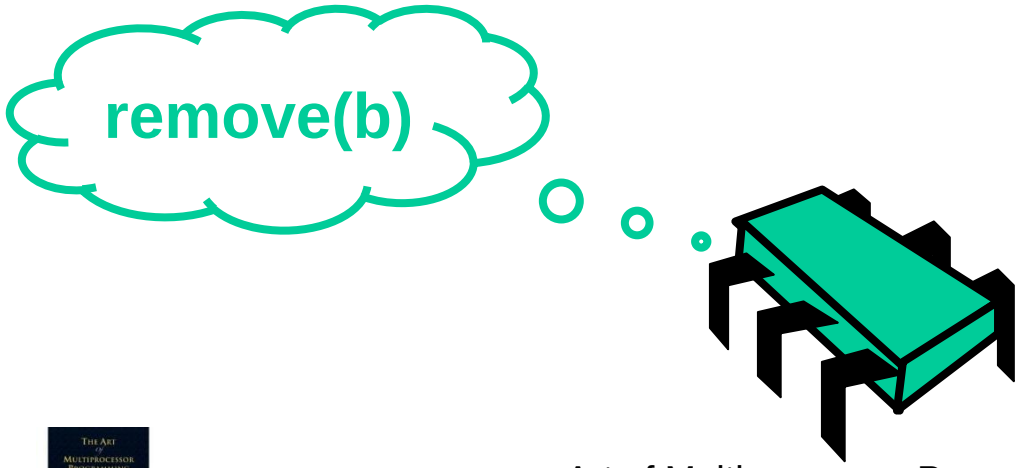
Hand-over-Hand locking

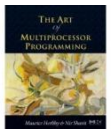




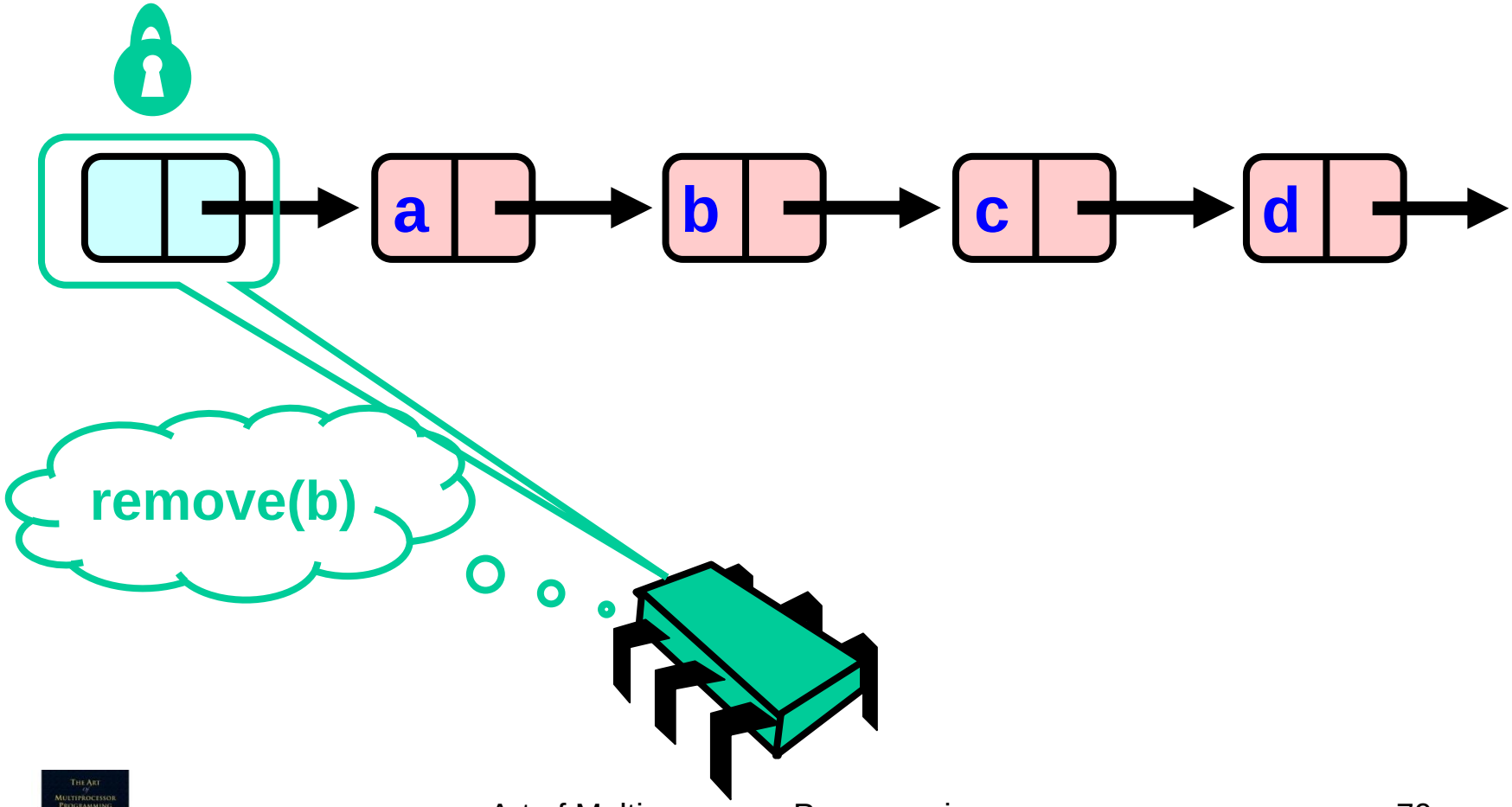
Removing a Node

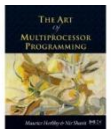




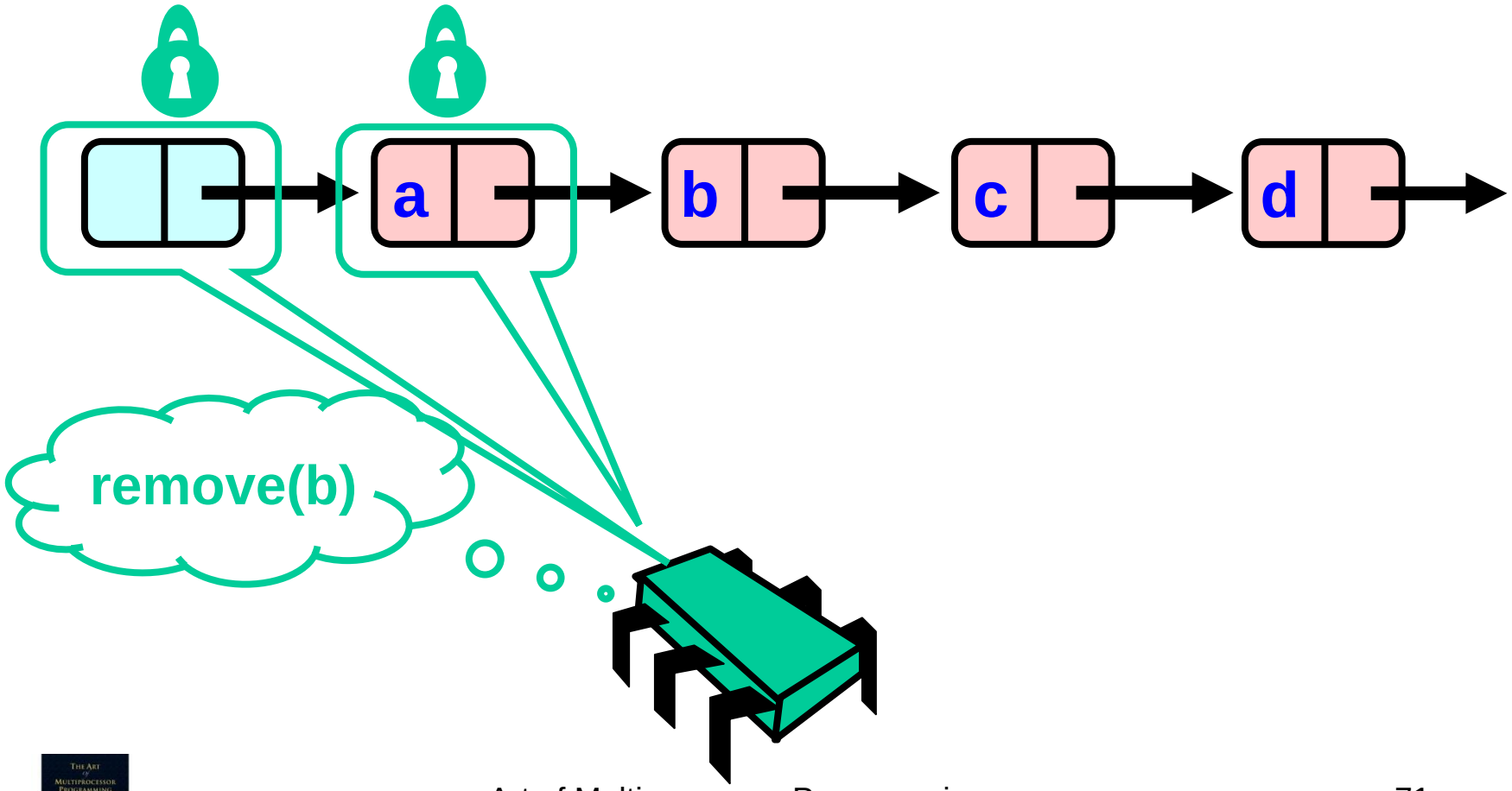


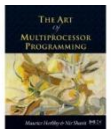
Removing a Node



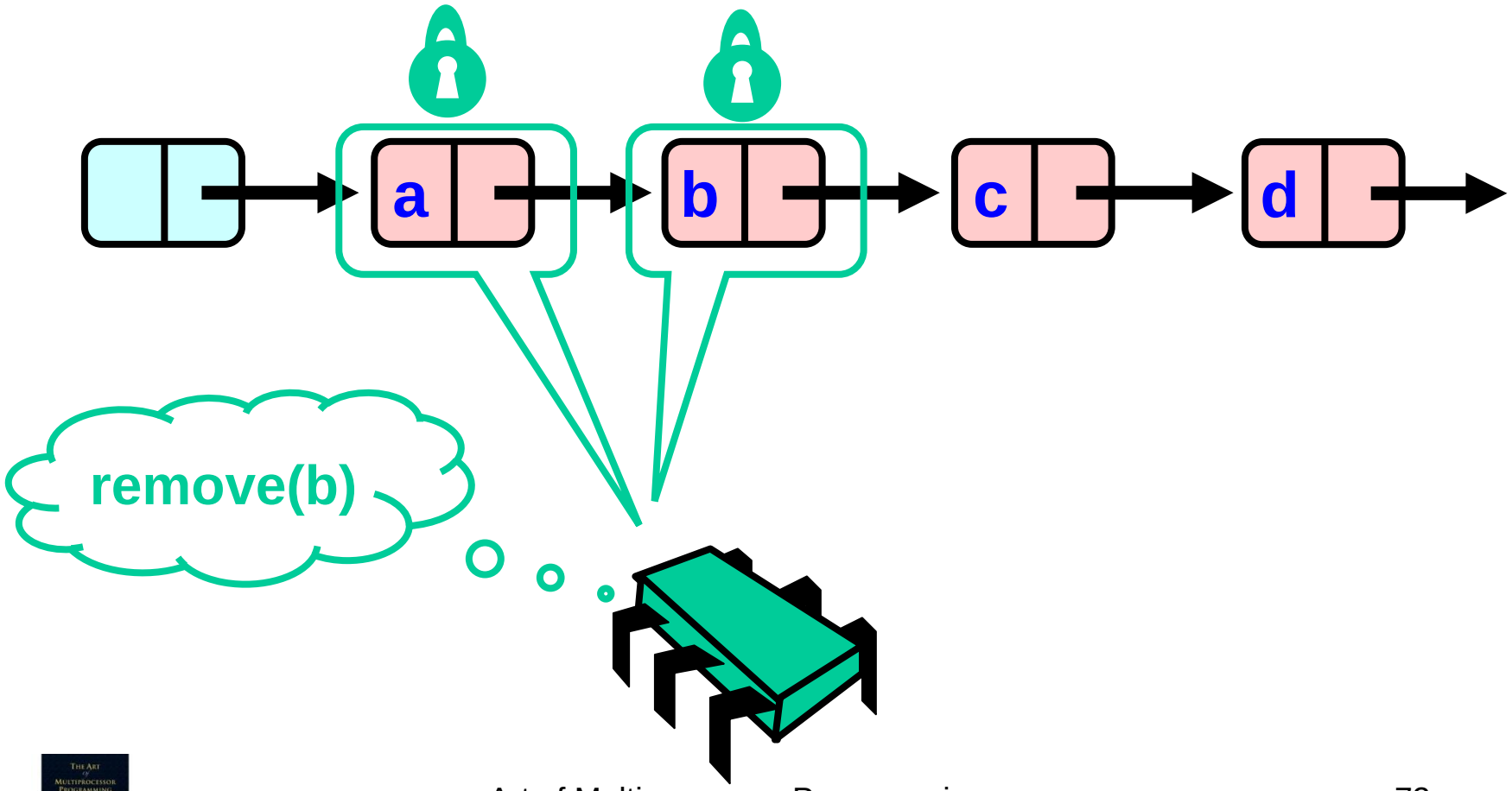


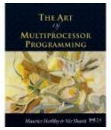
Removing a Node



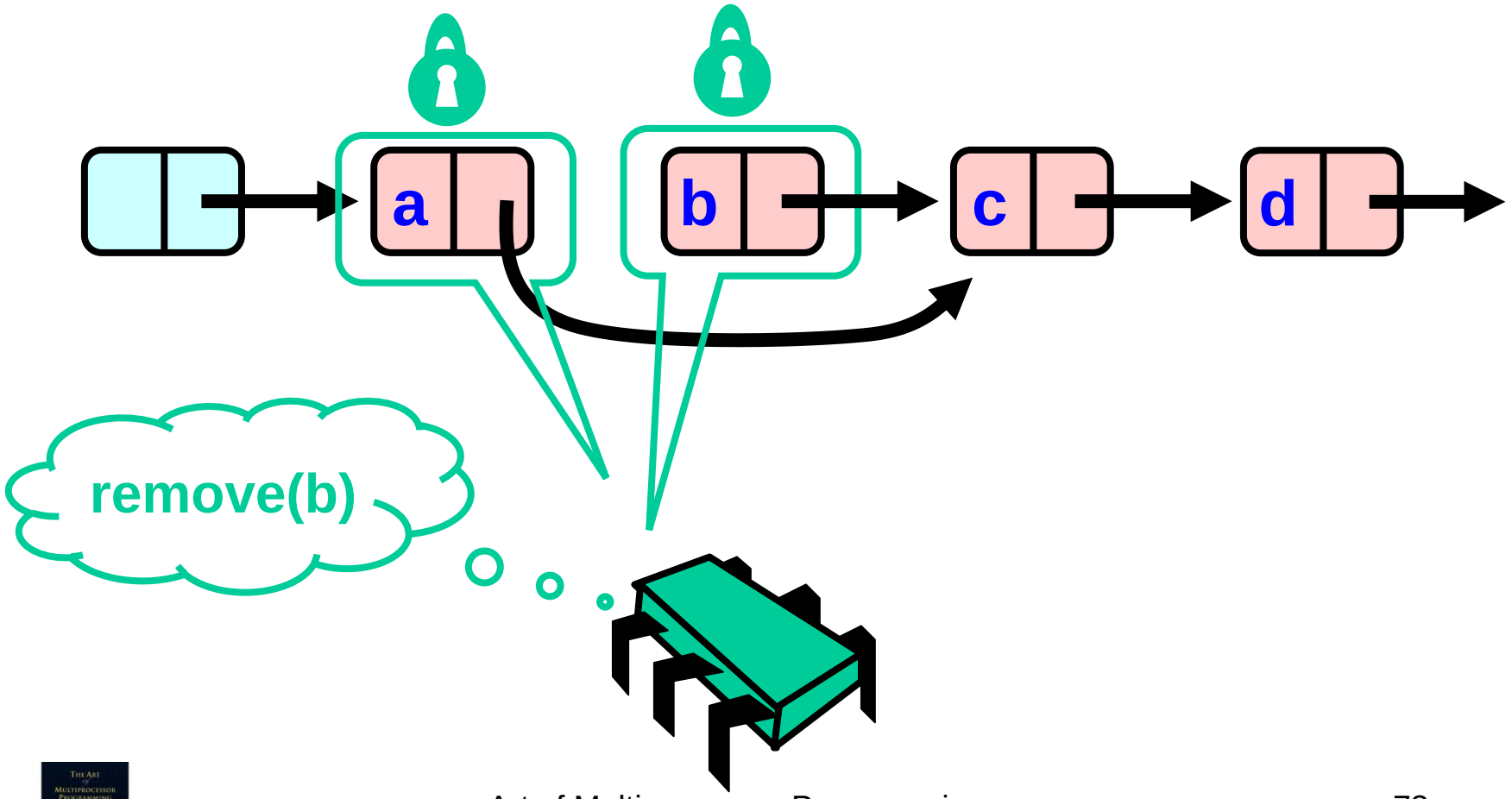


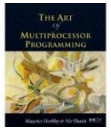
Removing a Node



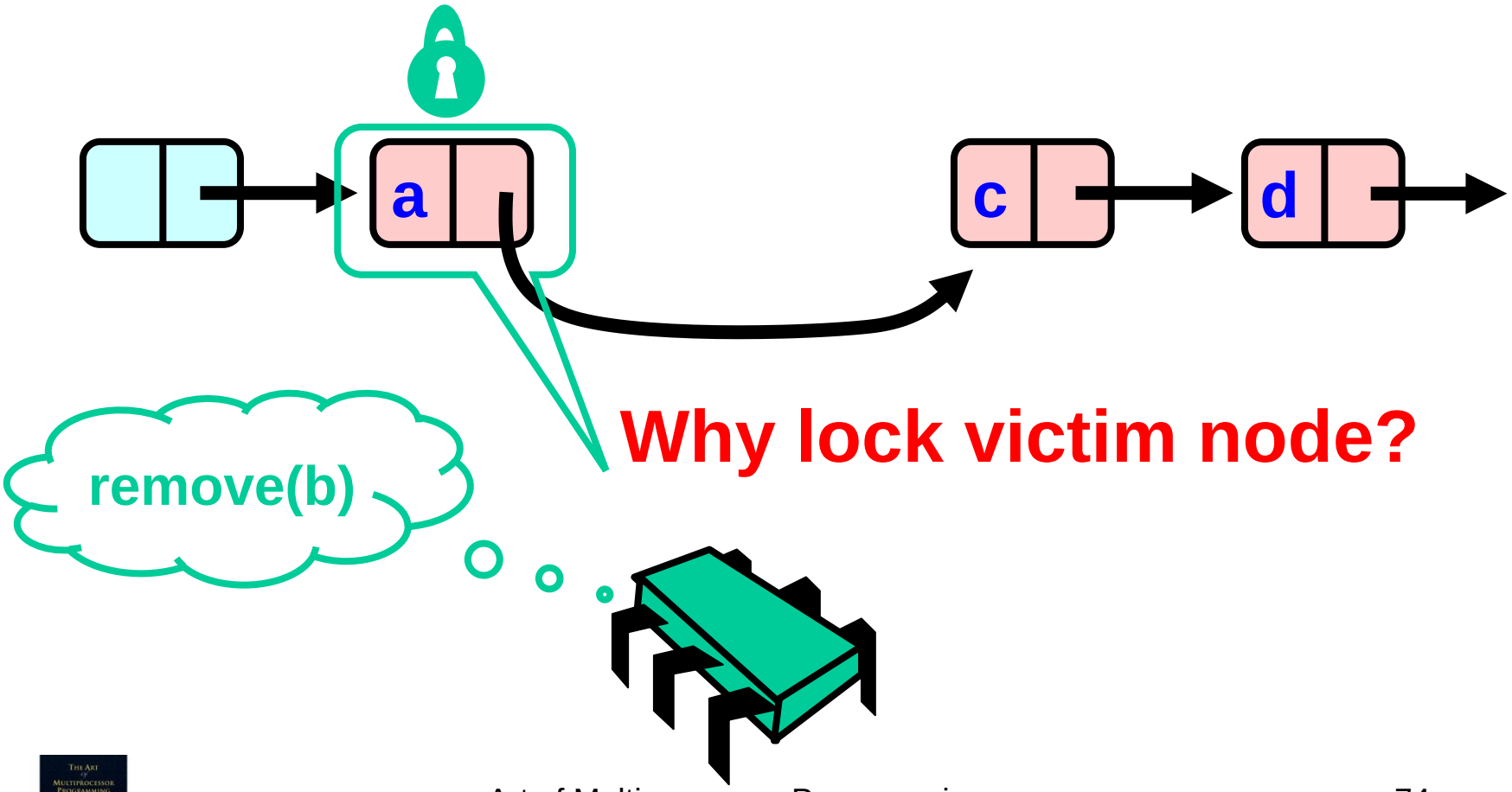


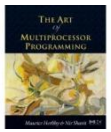
Removing a Node



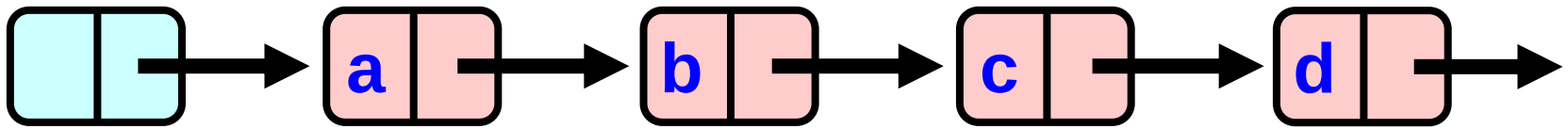


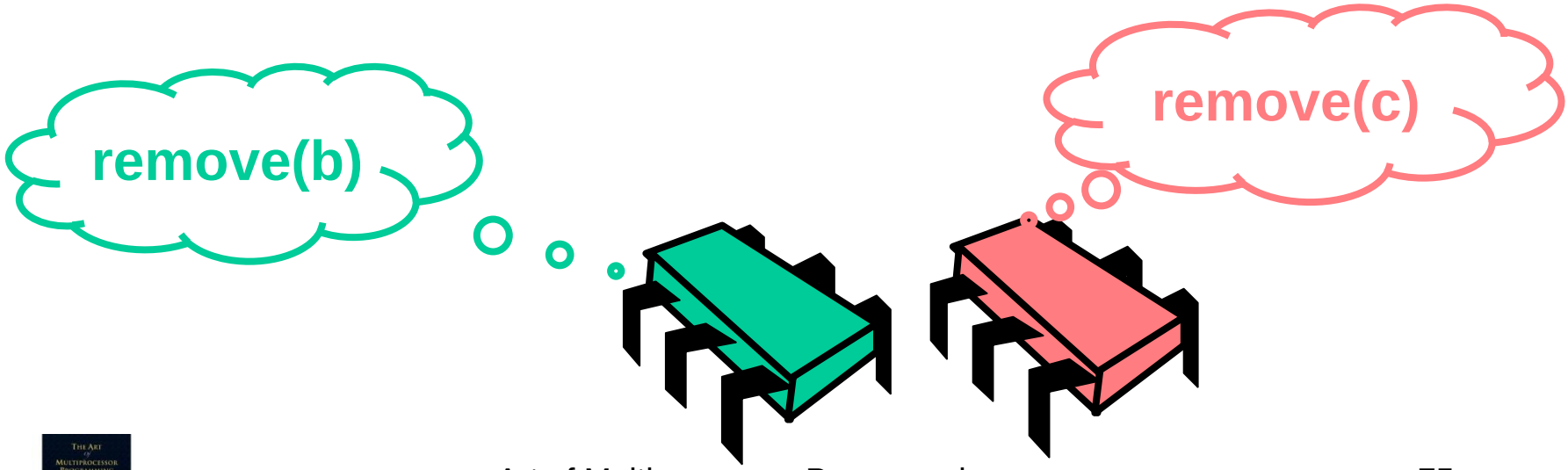
Removing a Node

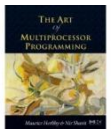




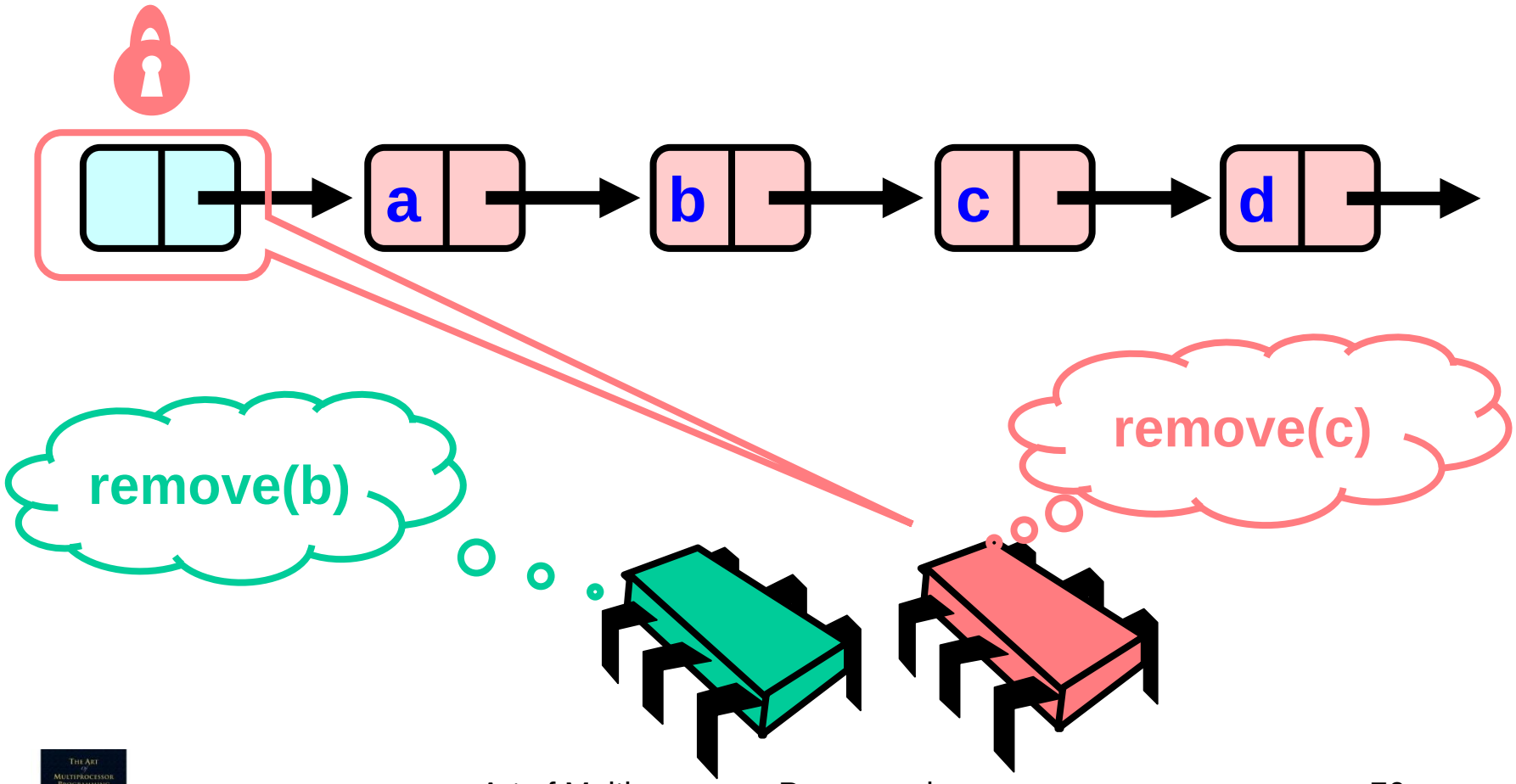
Concurrent Removes

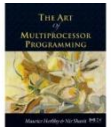




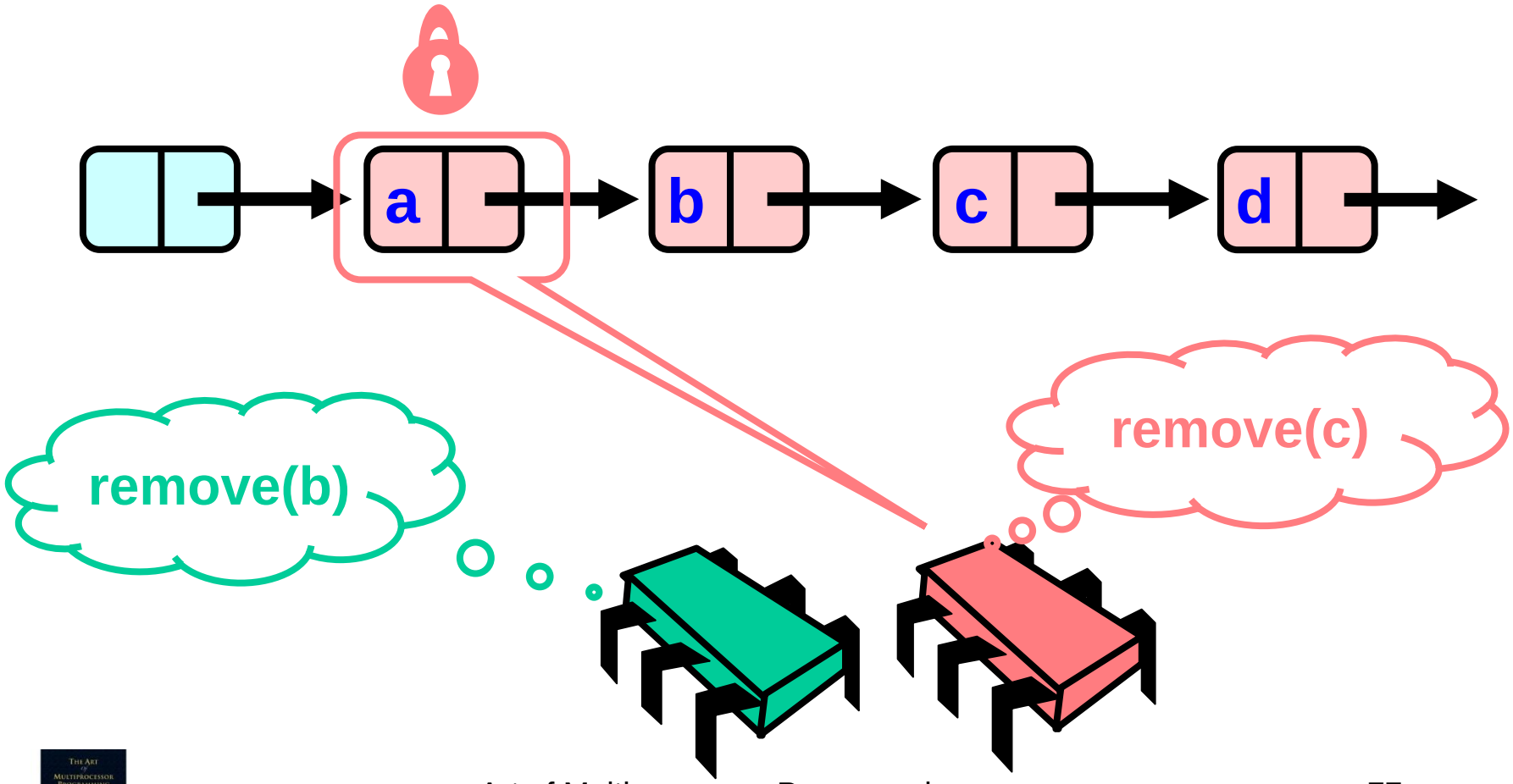


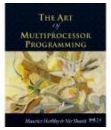
Concurrent Removes



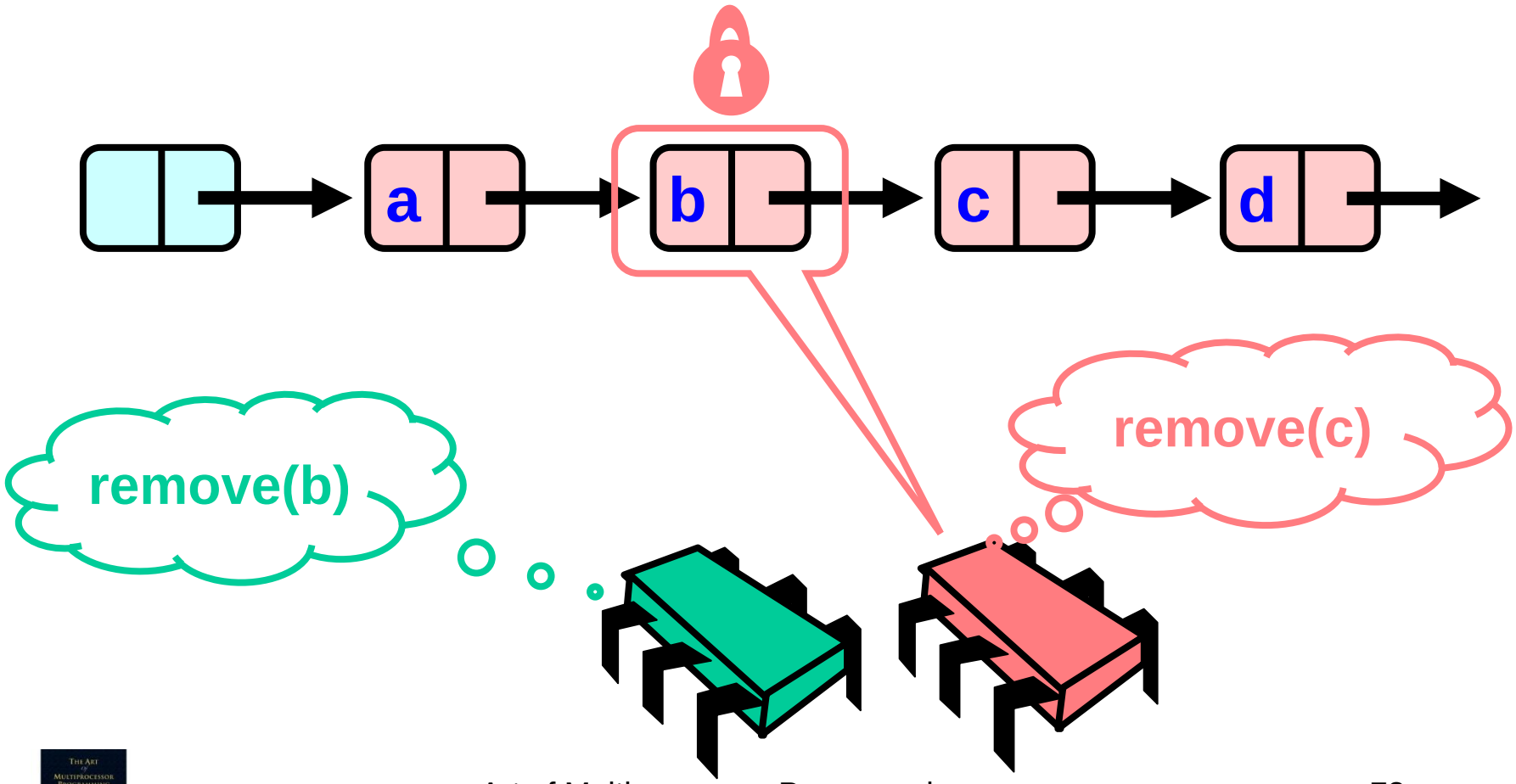


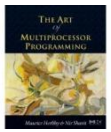
Concurrent Removes



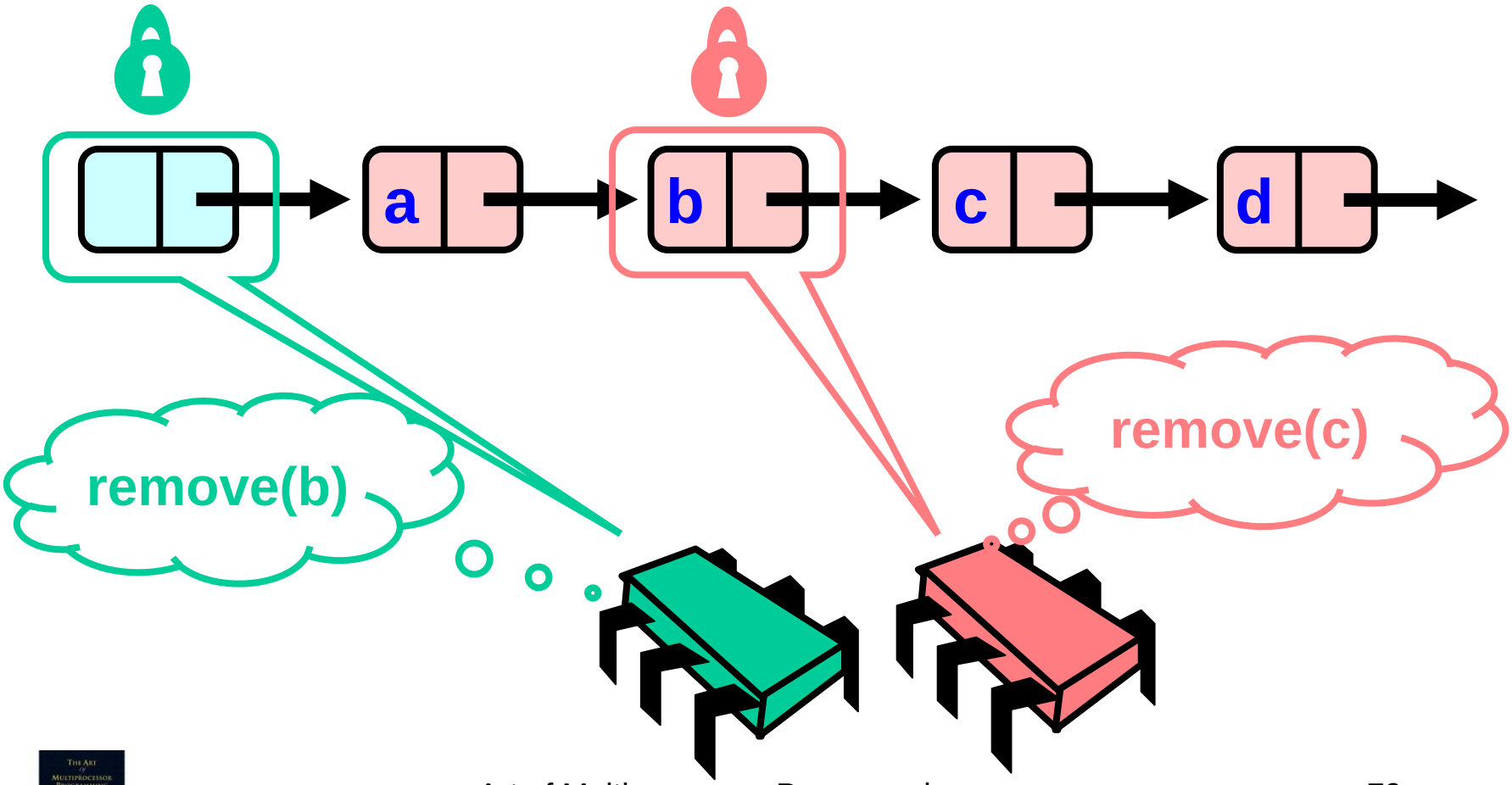


Concurrent Removes

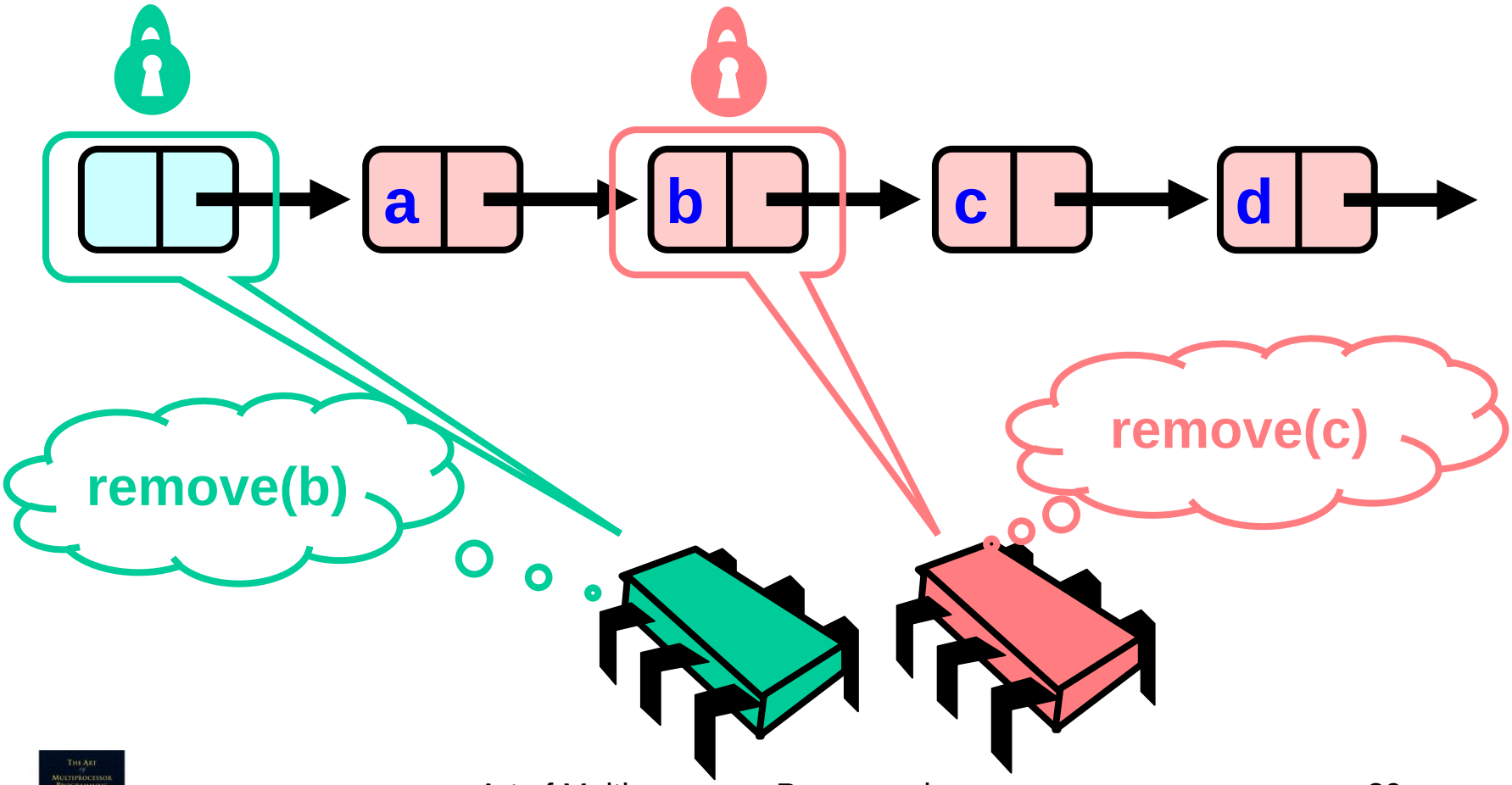


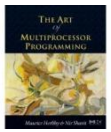


Concurrent Removes

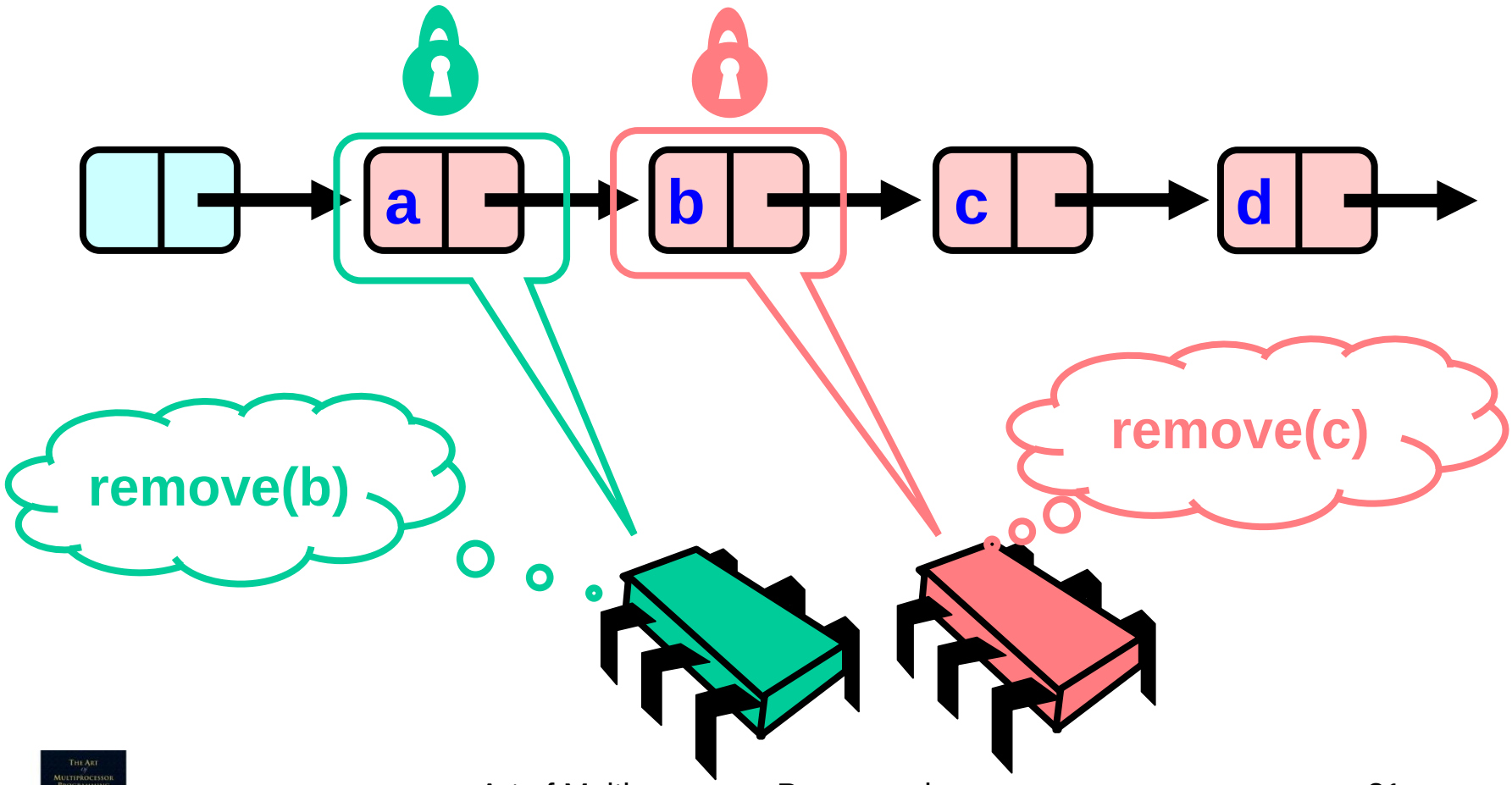


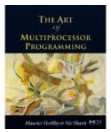
Concurrent Removes



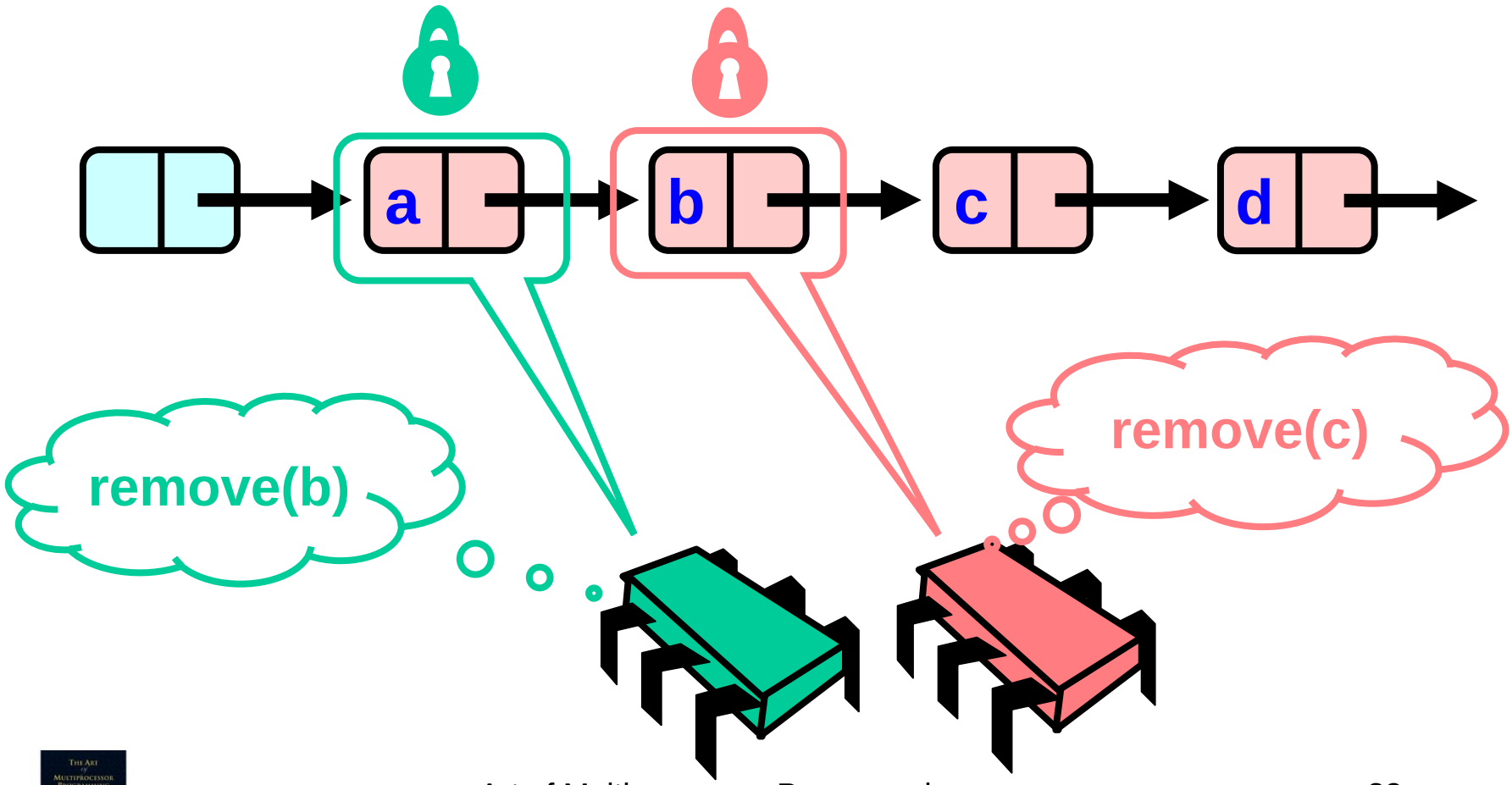


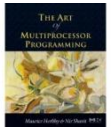
Concurrent Removes



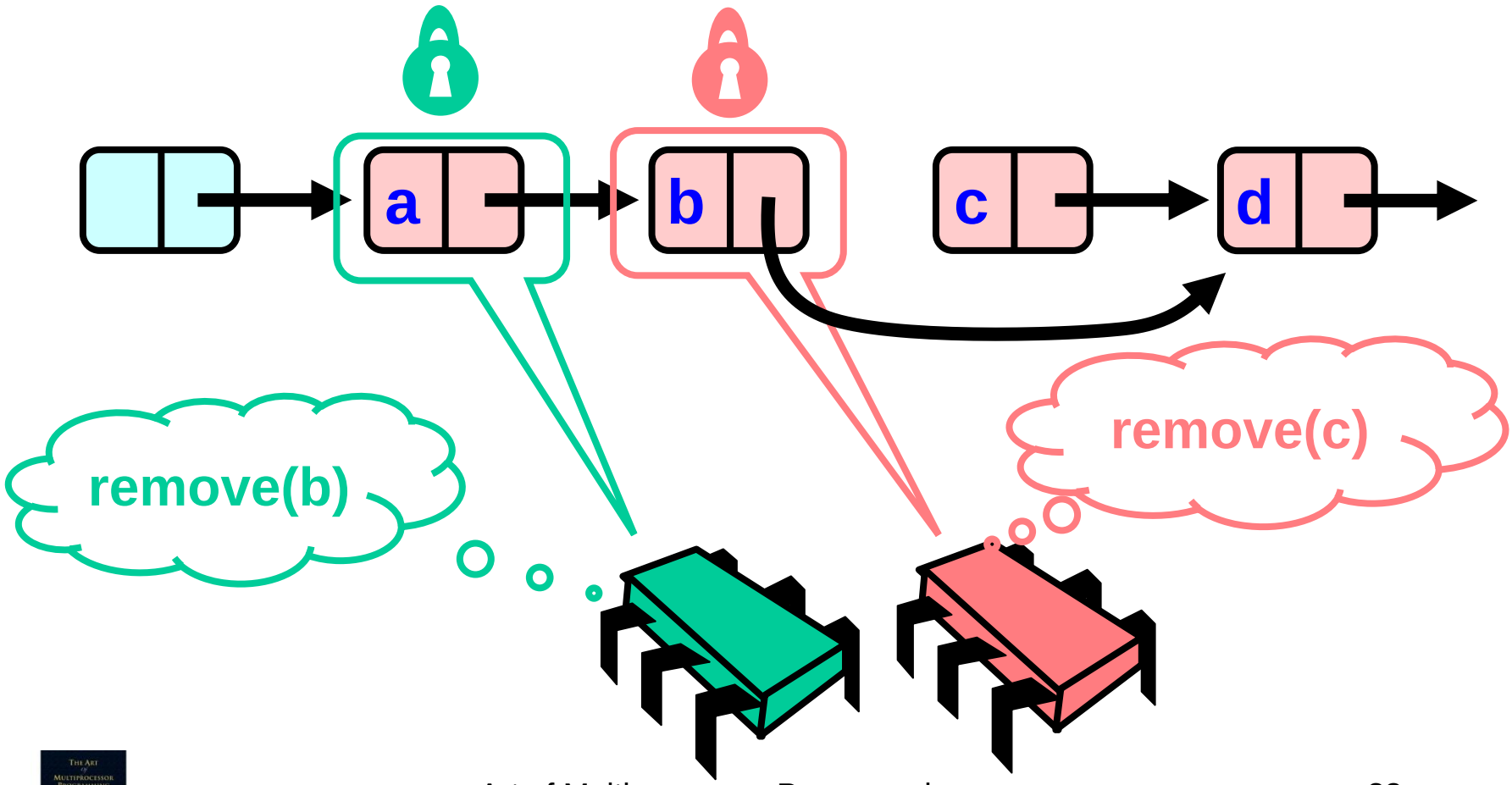


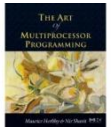
Concurrent Removes



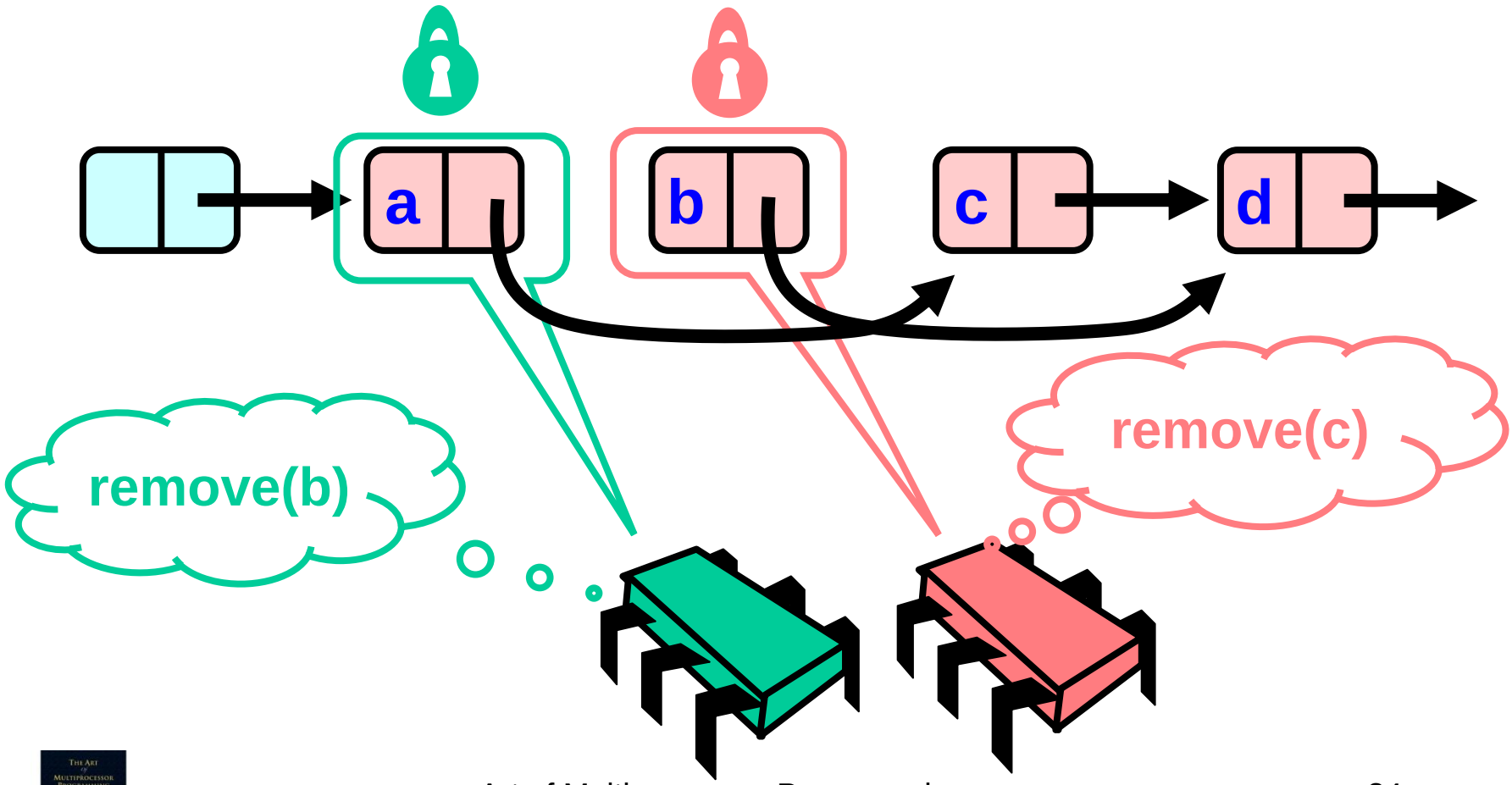


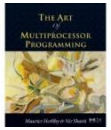
Concurrent Removes



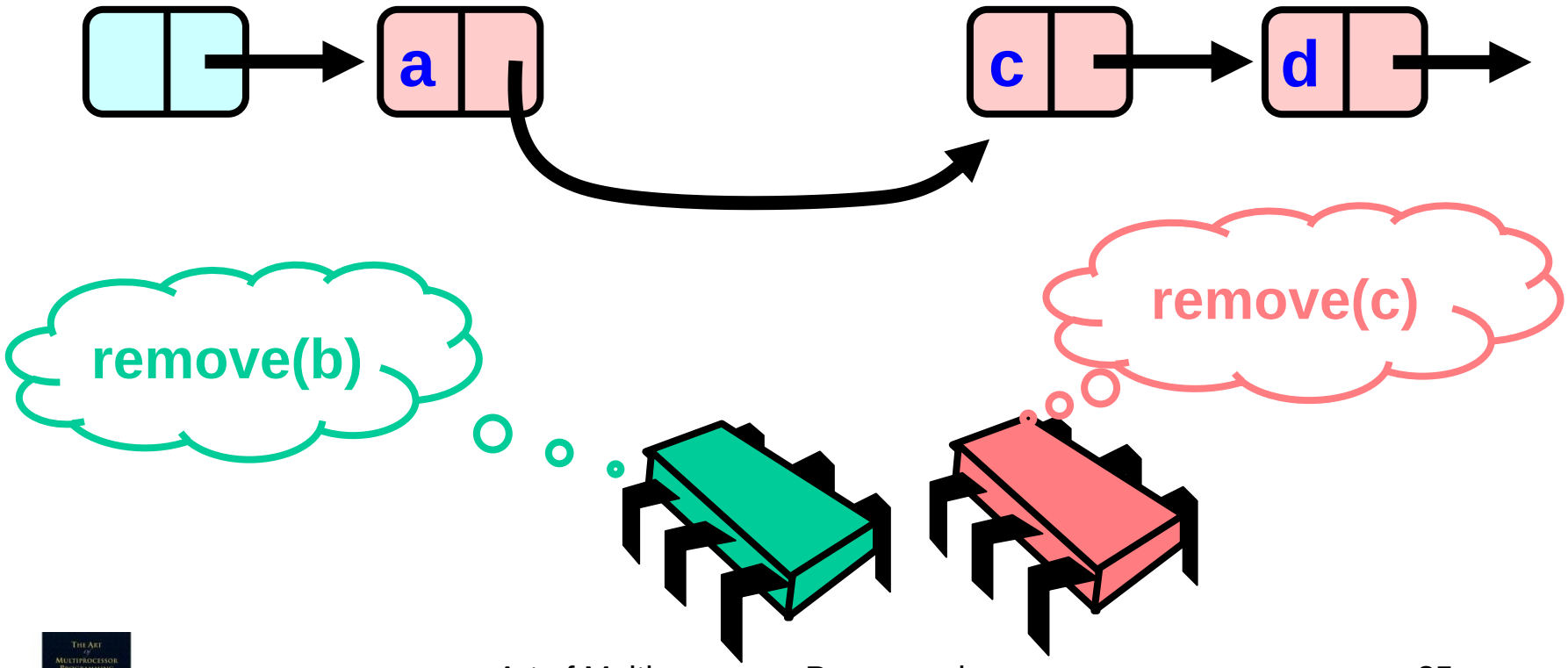


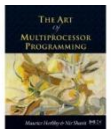
Concurrent Removes





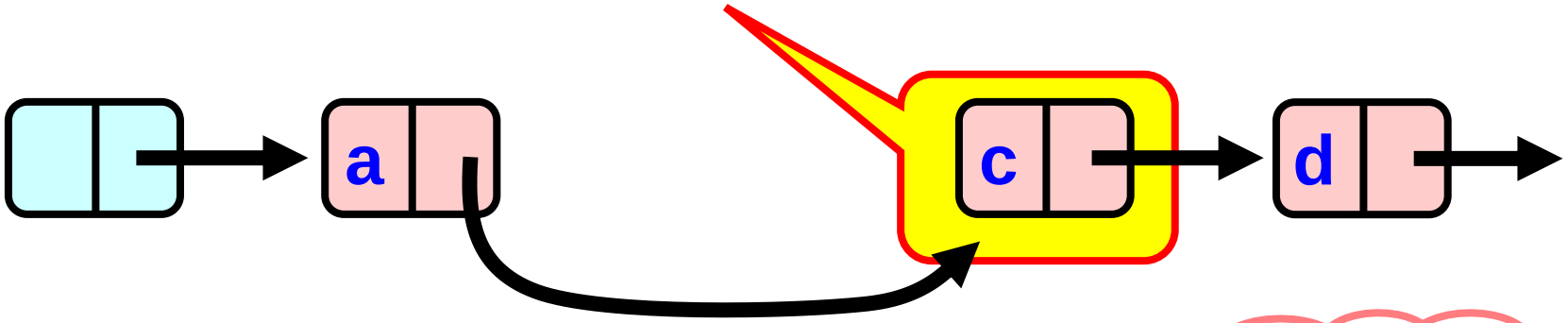
Uh, Oh

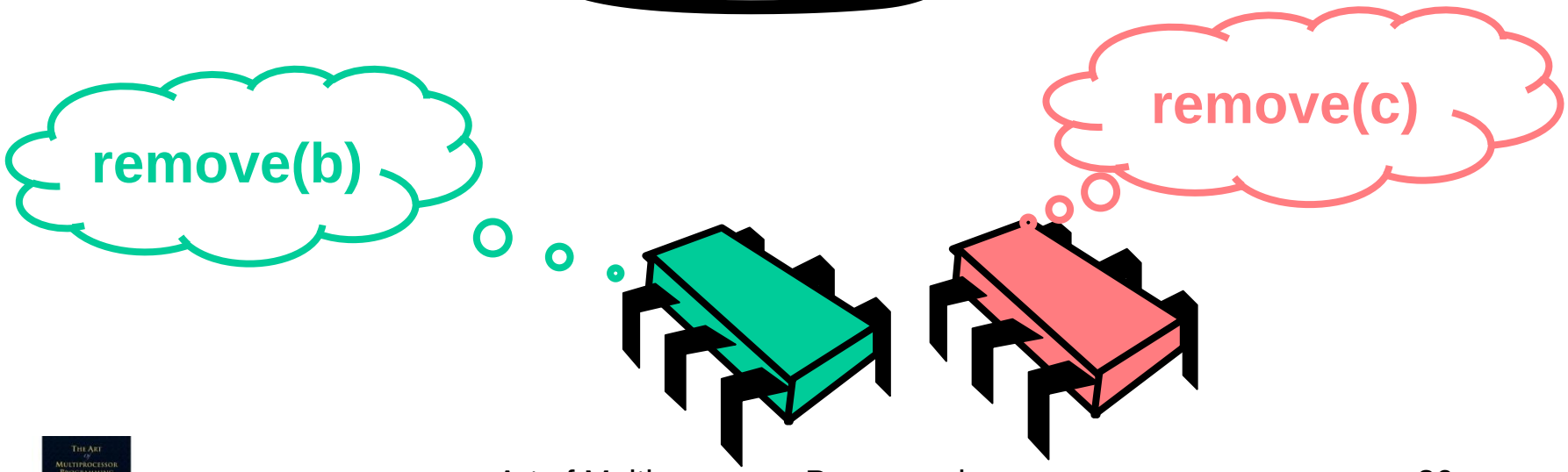


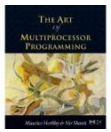


Uh, Oh

Bad news, c not removed

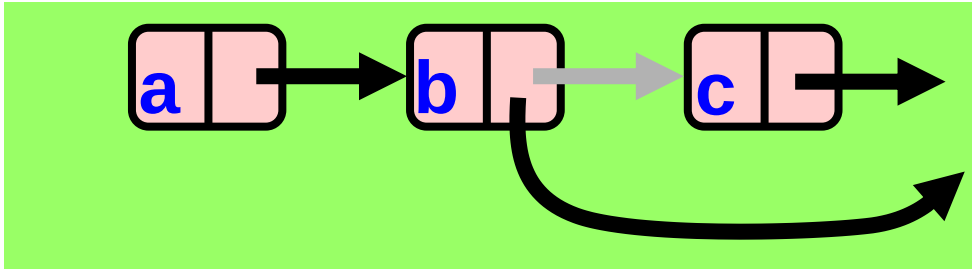




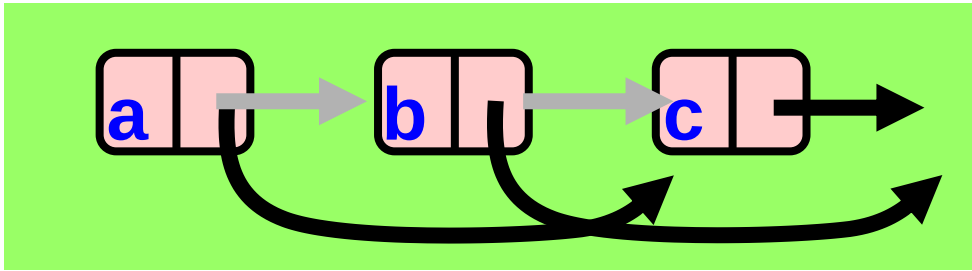


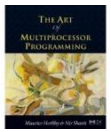
Problem

- To delete node c
  - Swing node b's next field to d



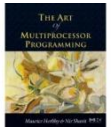
- Problem is,
  - Someone deleting b concurrently could direct a pointer to c



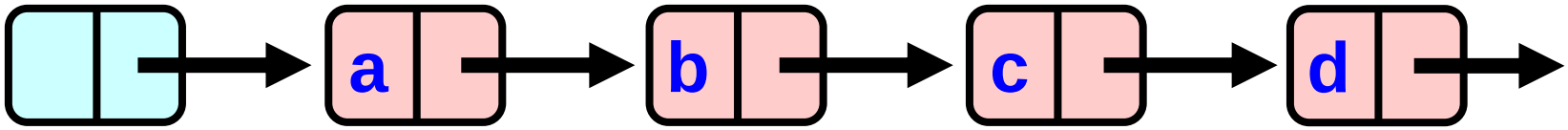


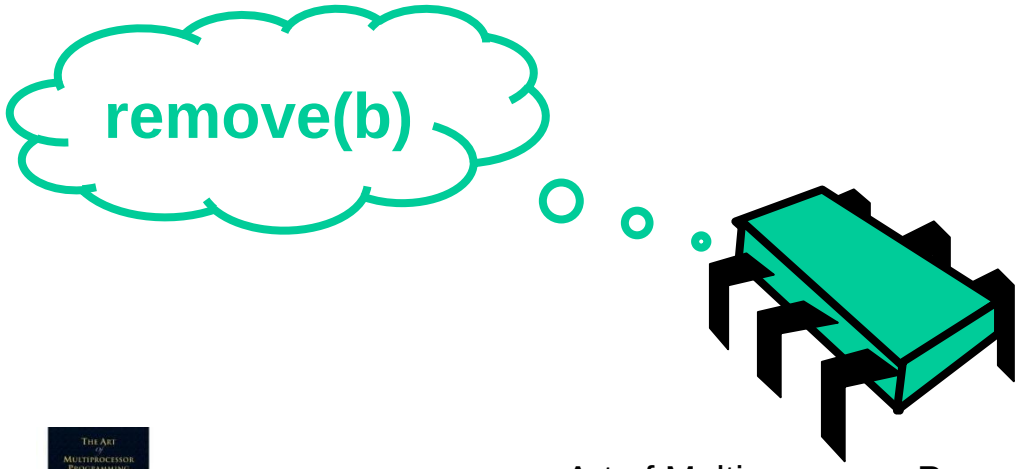
Insight

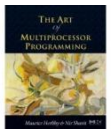
- If a node is locked
  - No one can delete node's *successor*
- If a thread locks
  - Node to be deleted
  - And its predecessor
  - Then it works



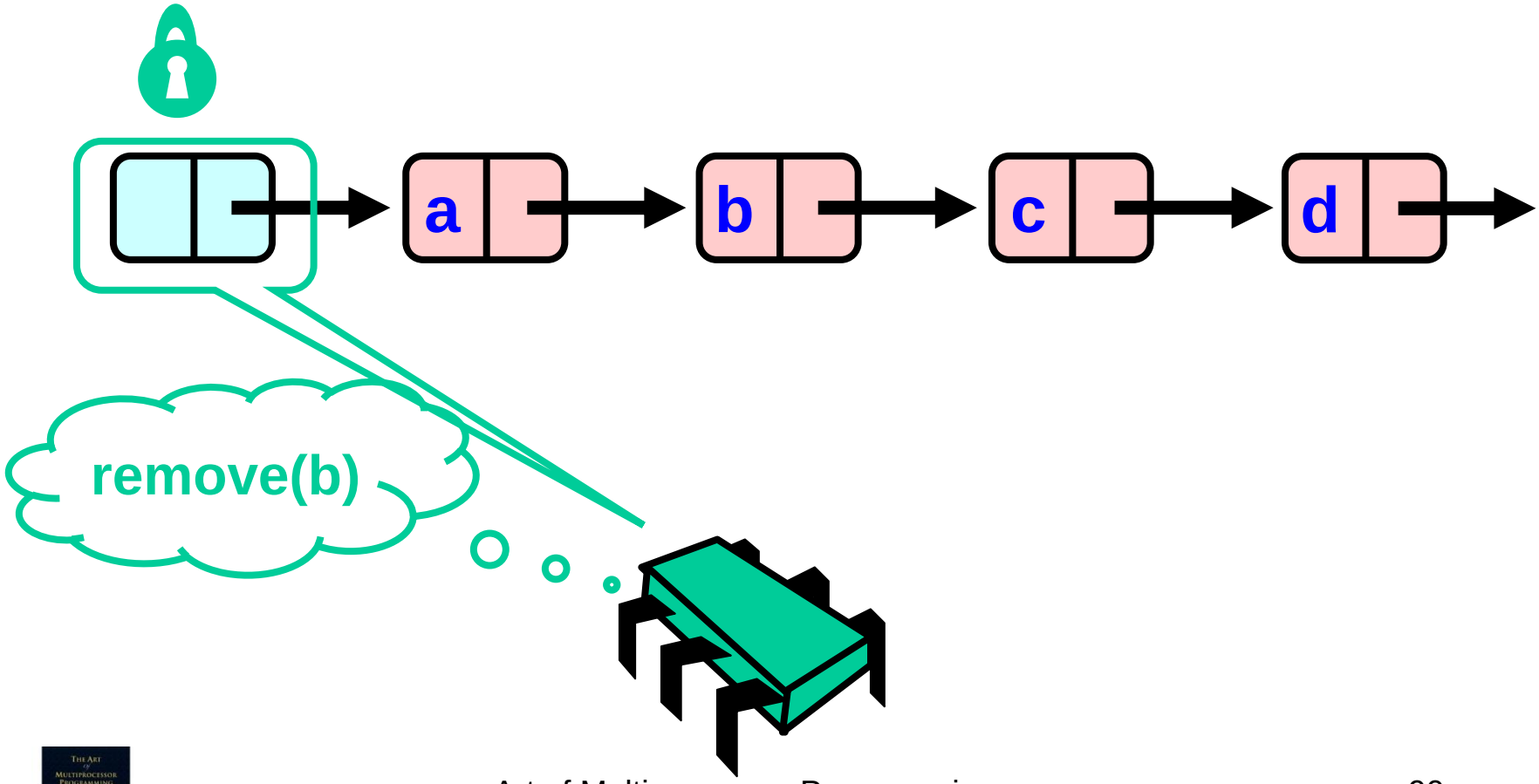
Hand-Over-Hand Again

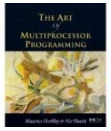




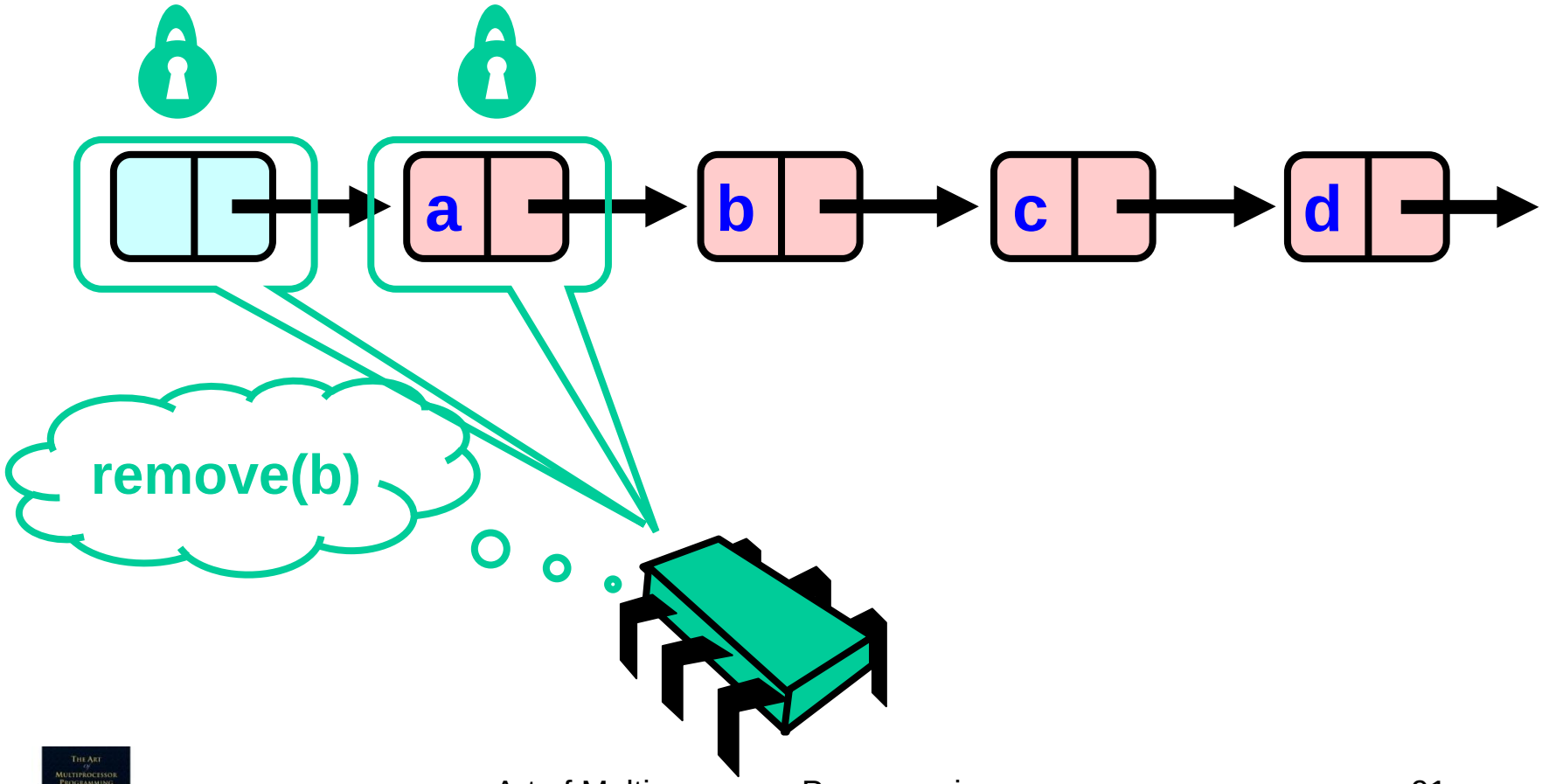


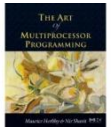
Hand-Over-Hand Again



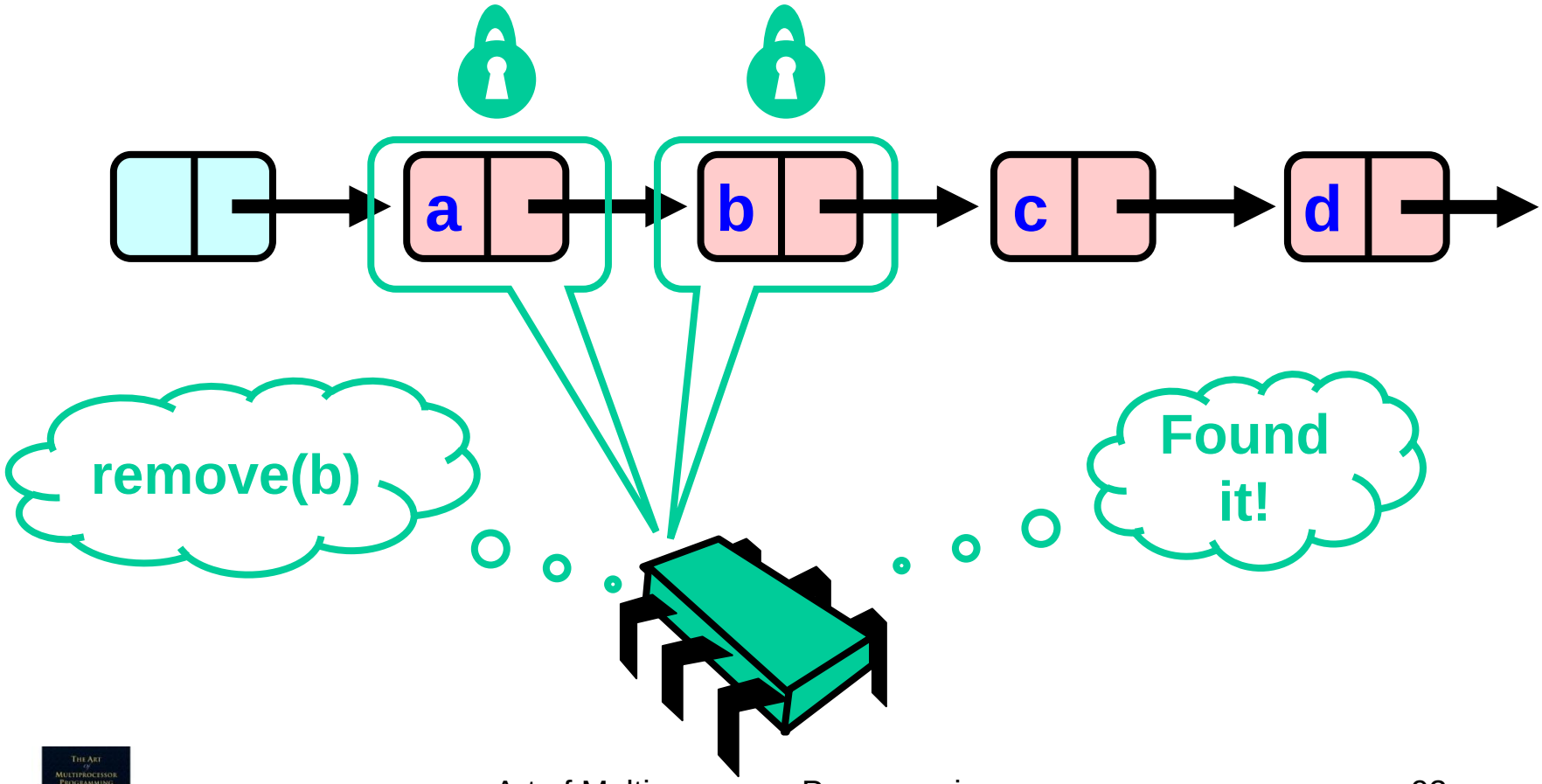


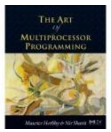
Hand-Over-Hand Again



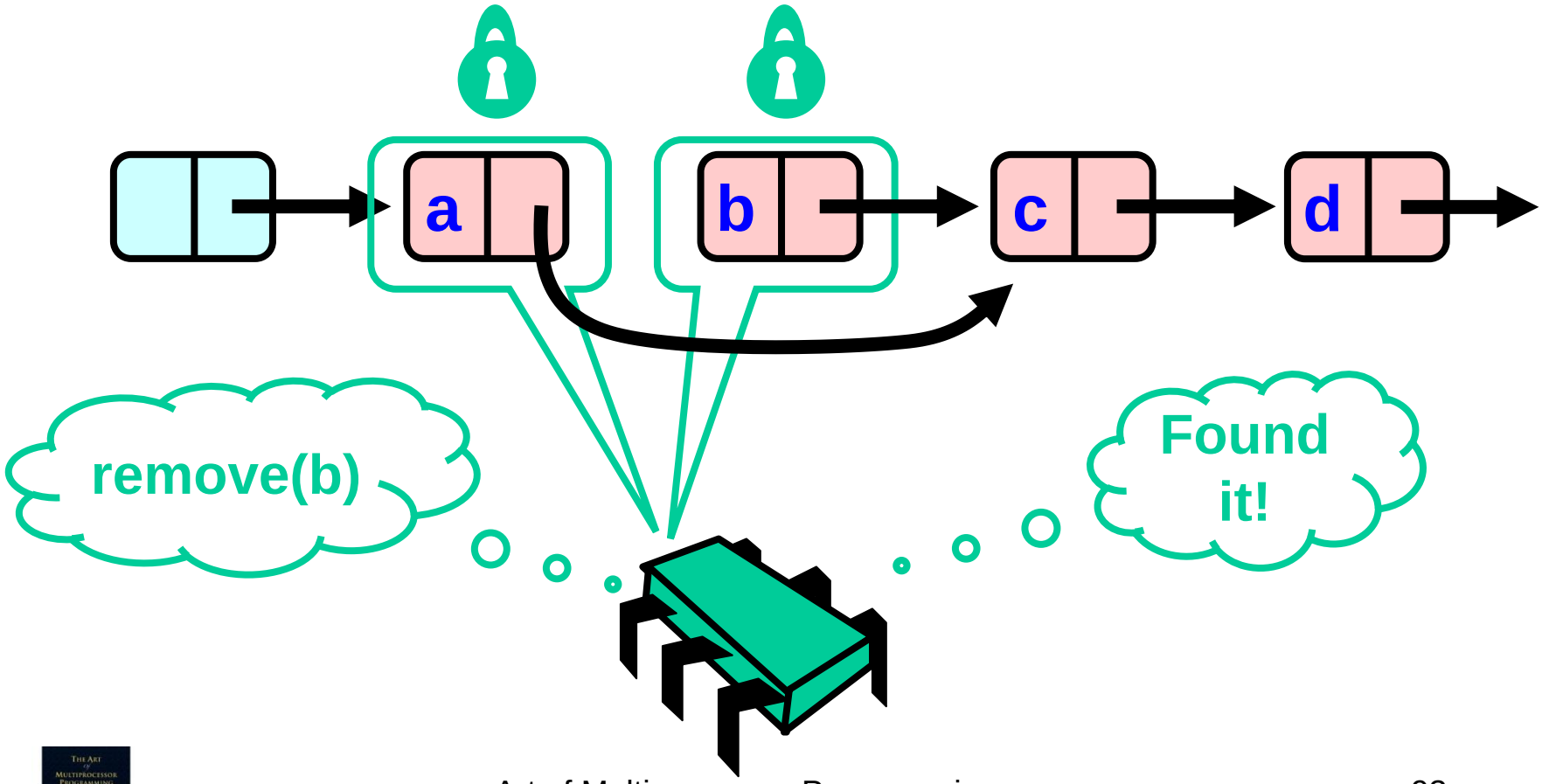


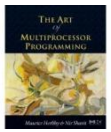
Hand-Over-Hand Again



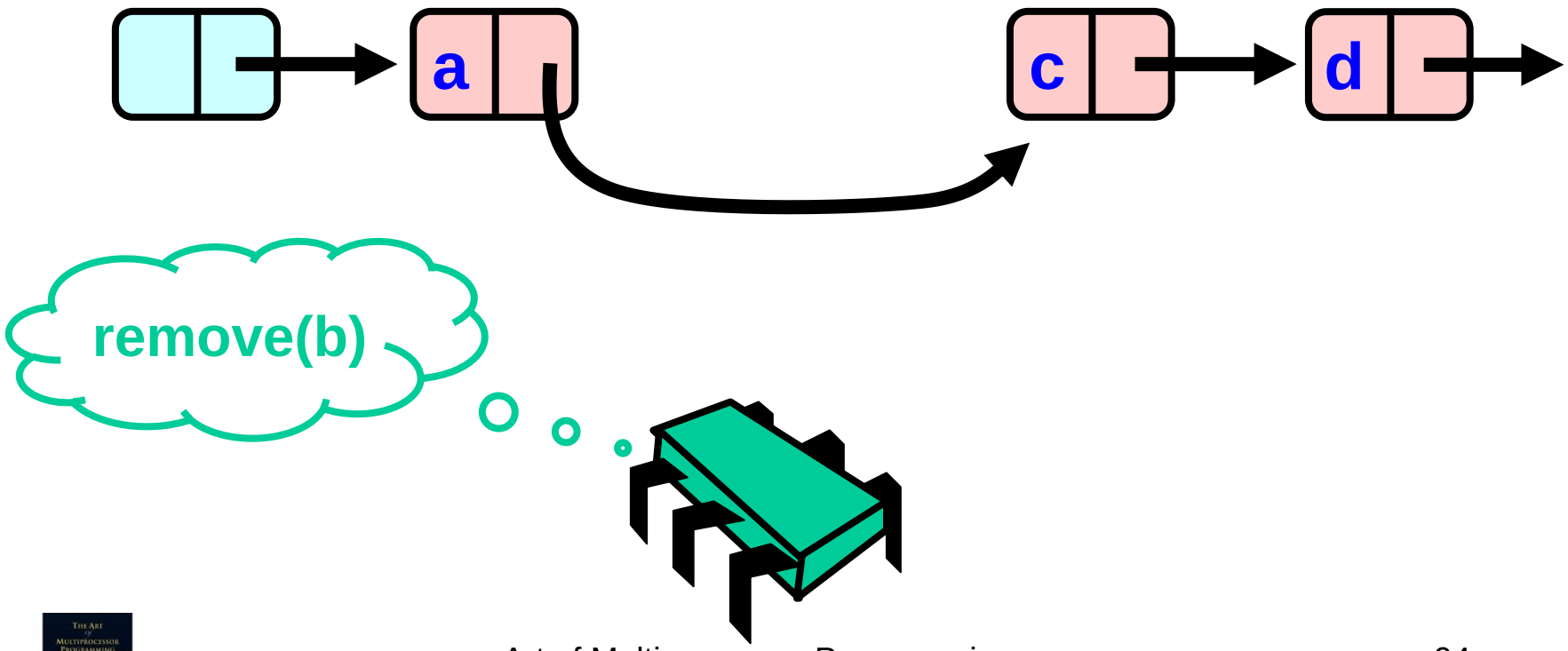


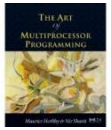
Hand-Over-Hand Again



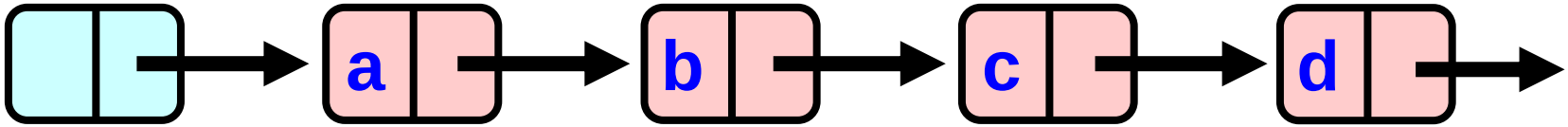


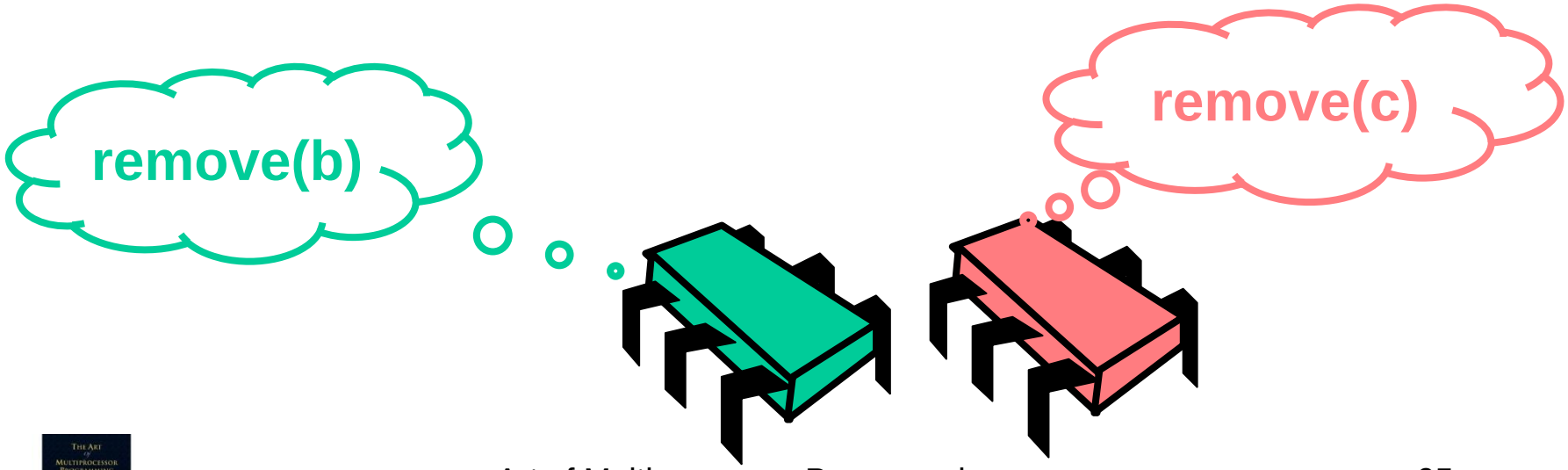
Hand-Over-Hand Again

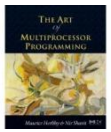




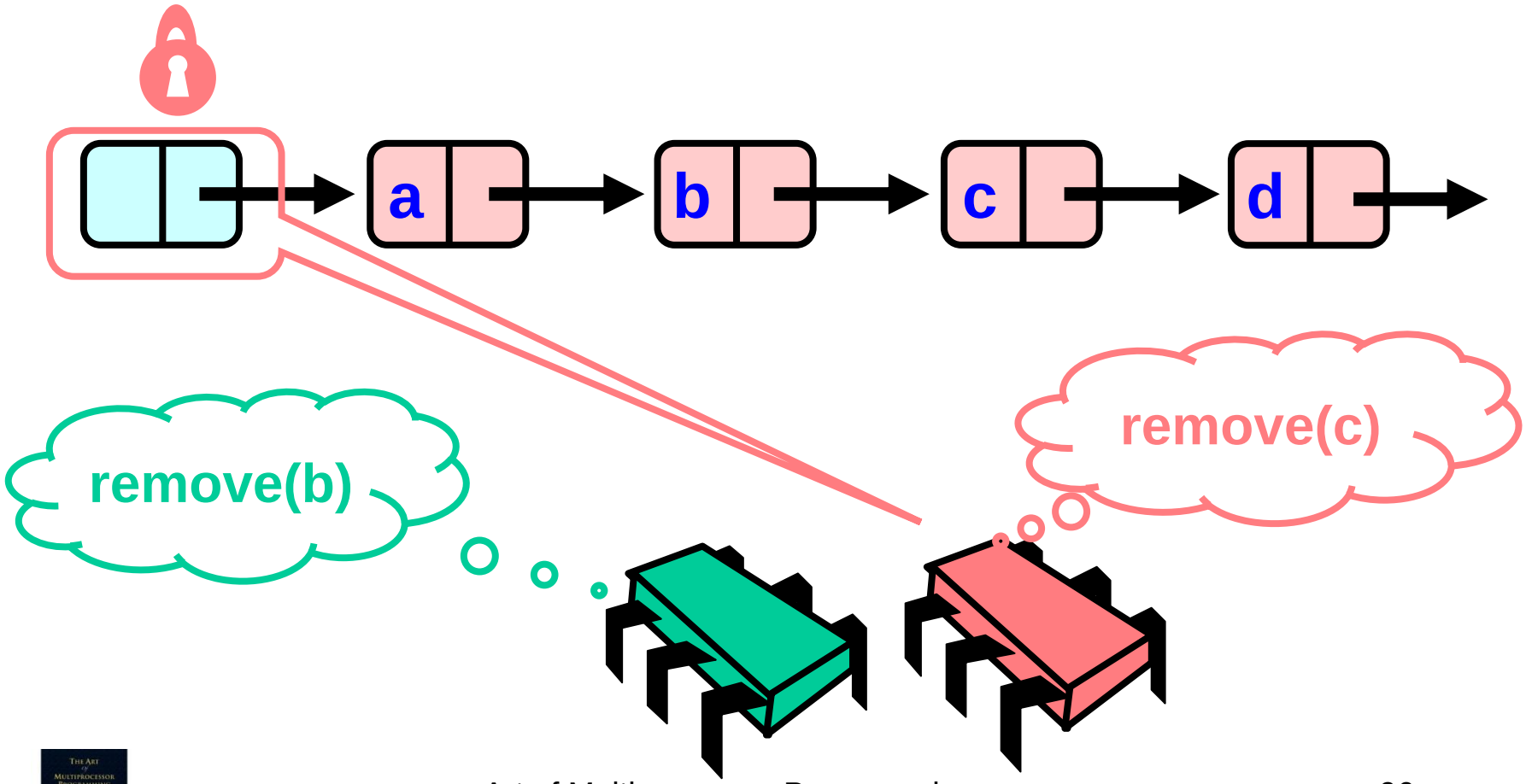
Removing a Node

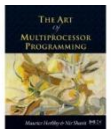




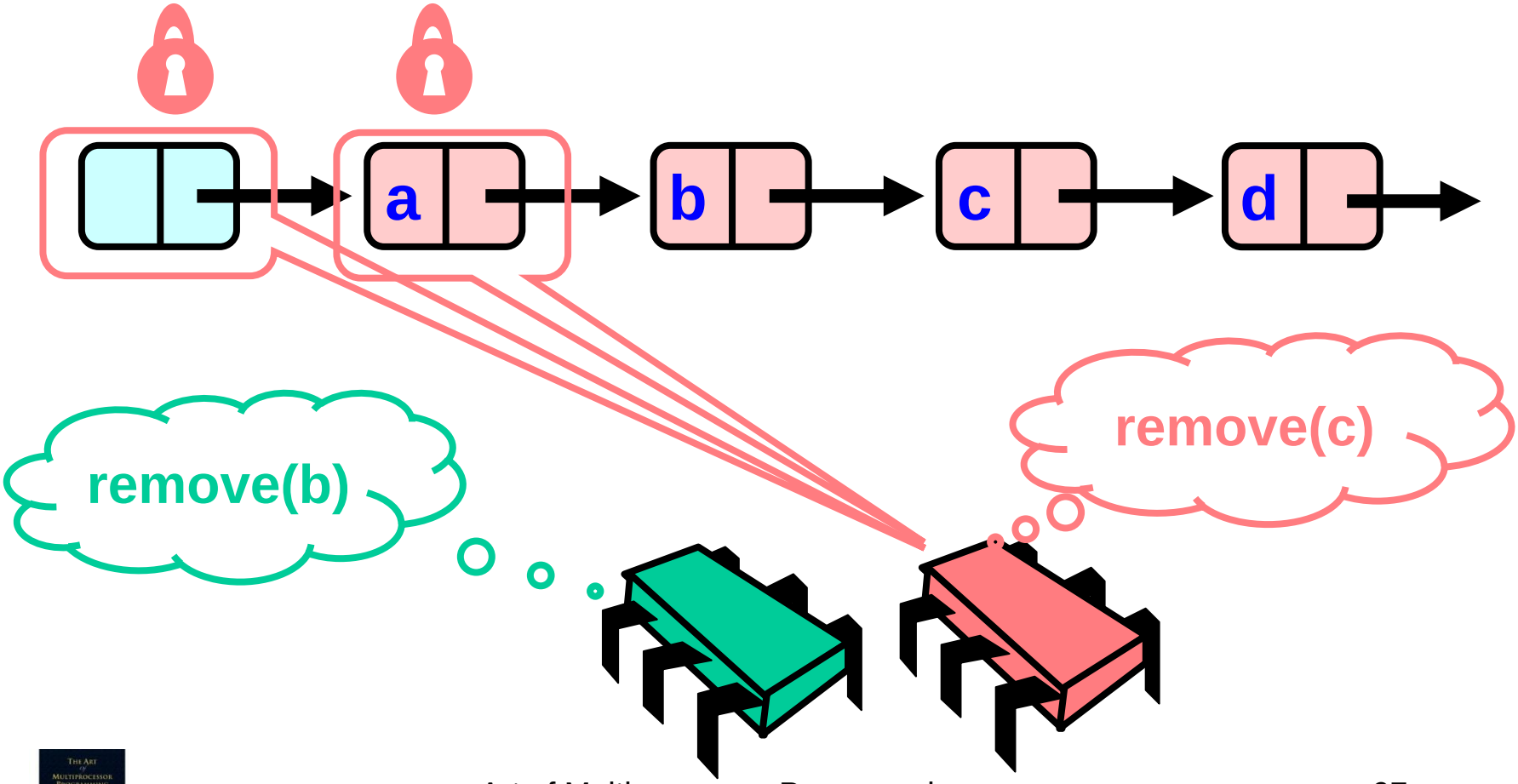


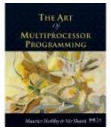
Removing a Node



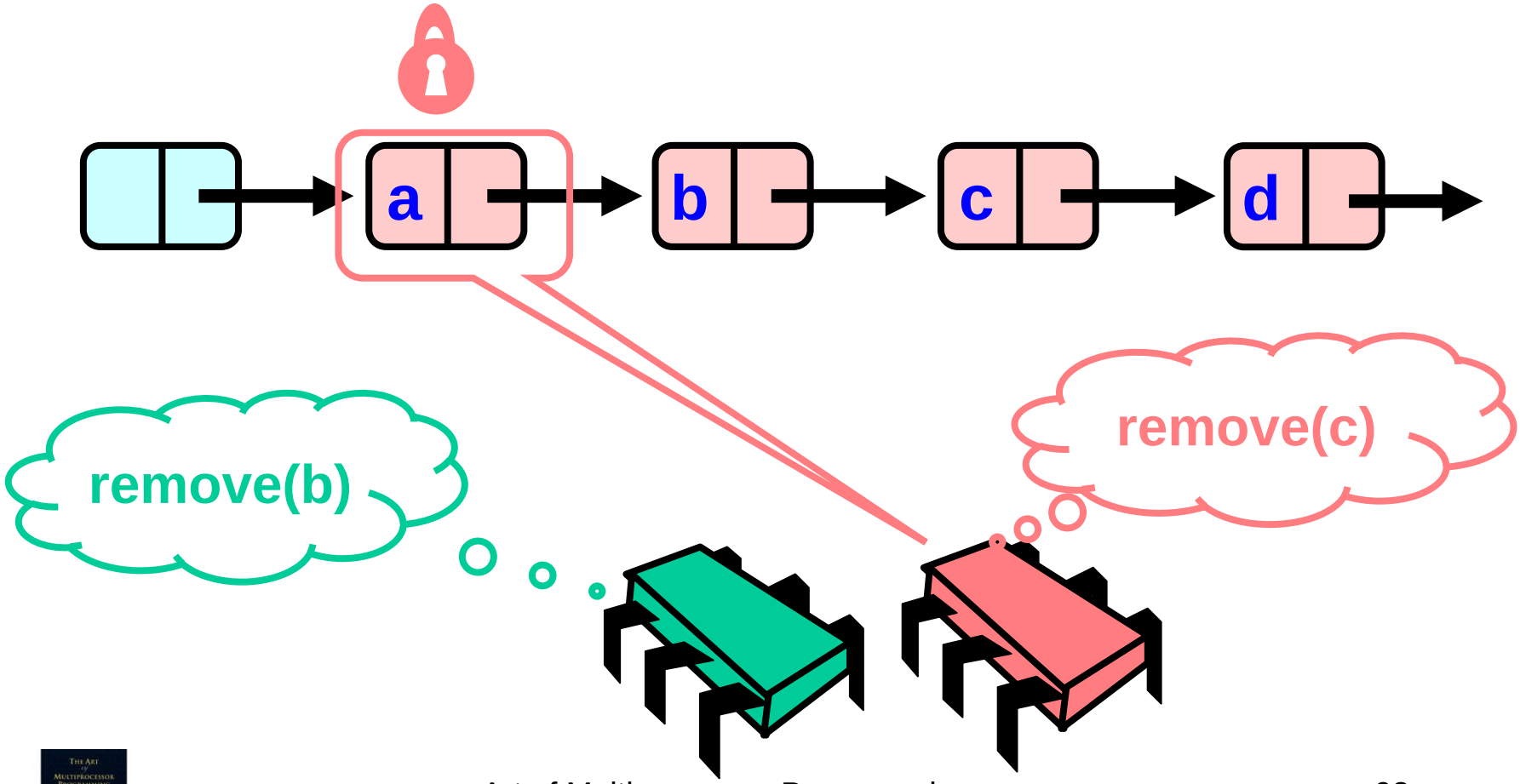


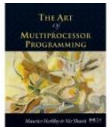
Removing a Node



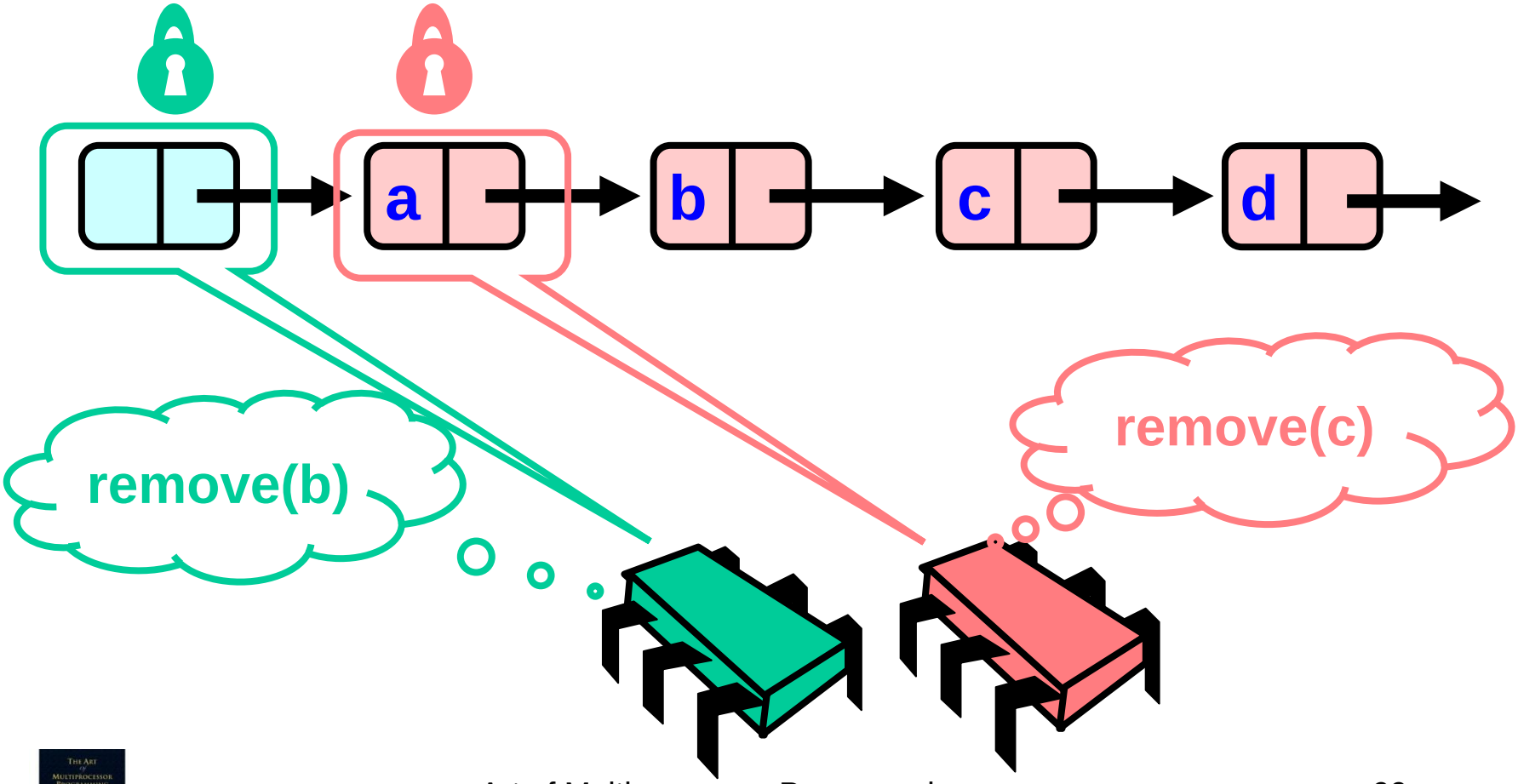


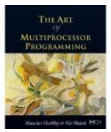
Removing a Node



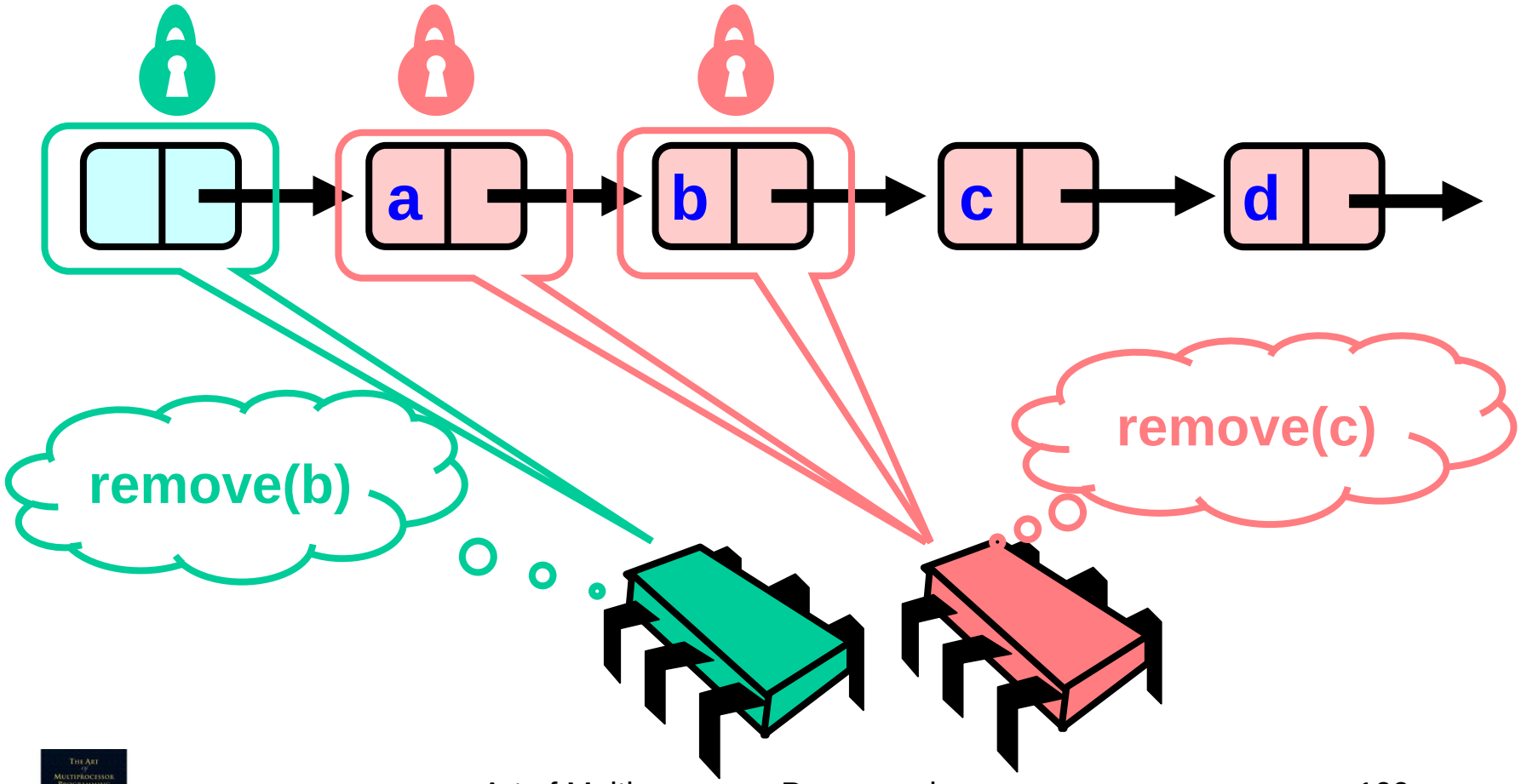


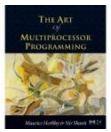
Removing a Node



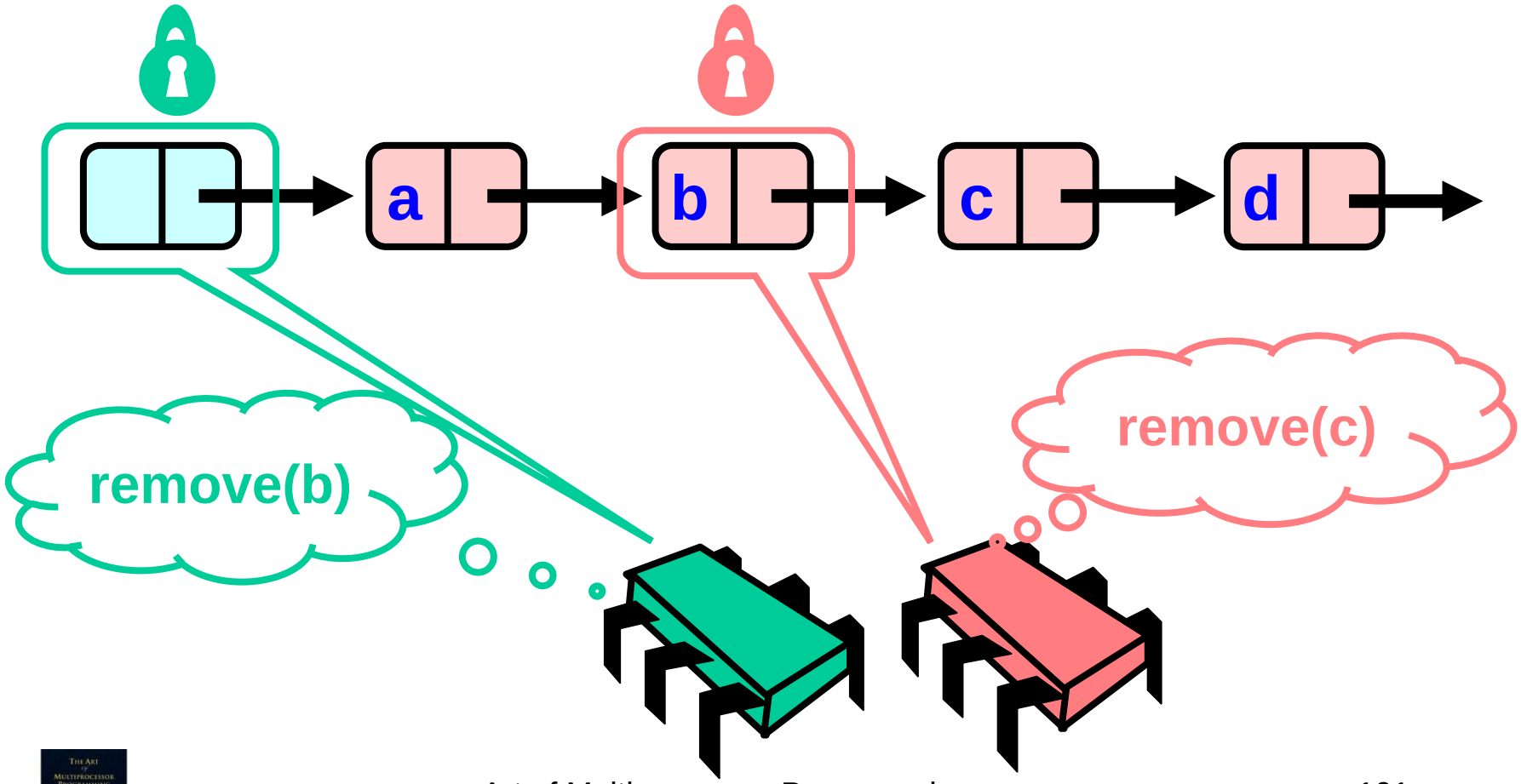


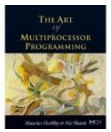
Removing a Node



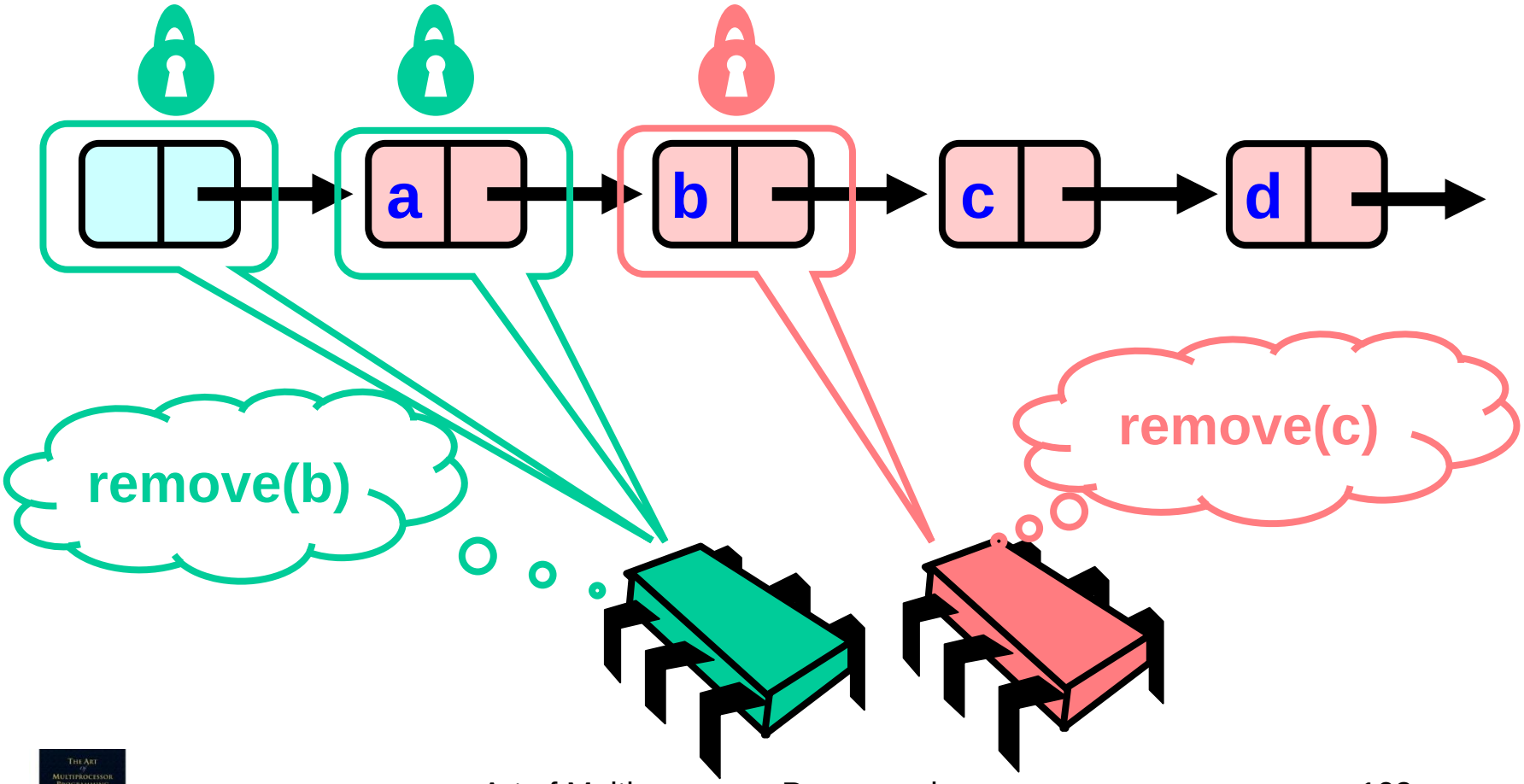


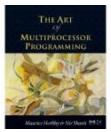
Removing a Node



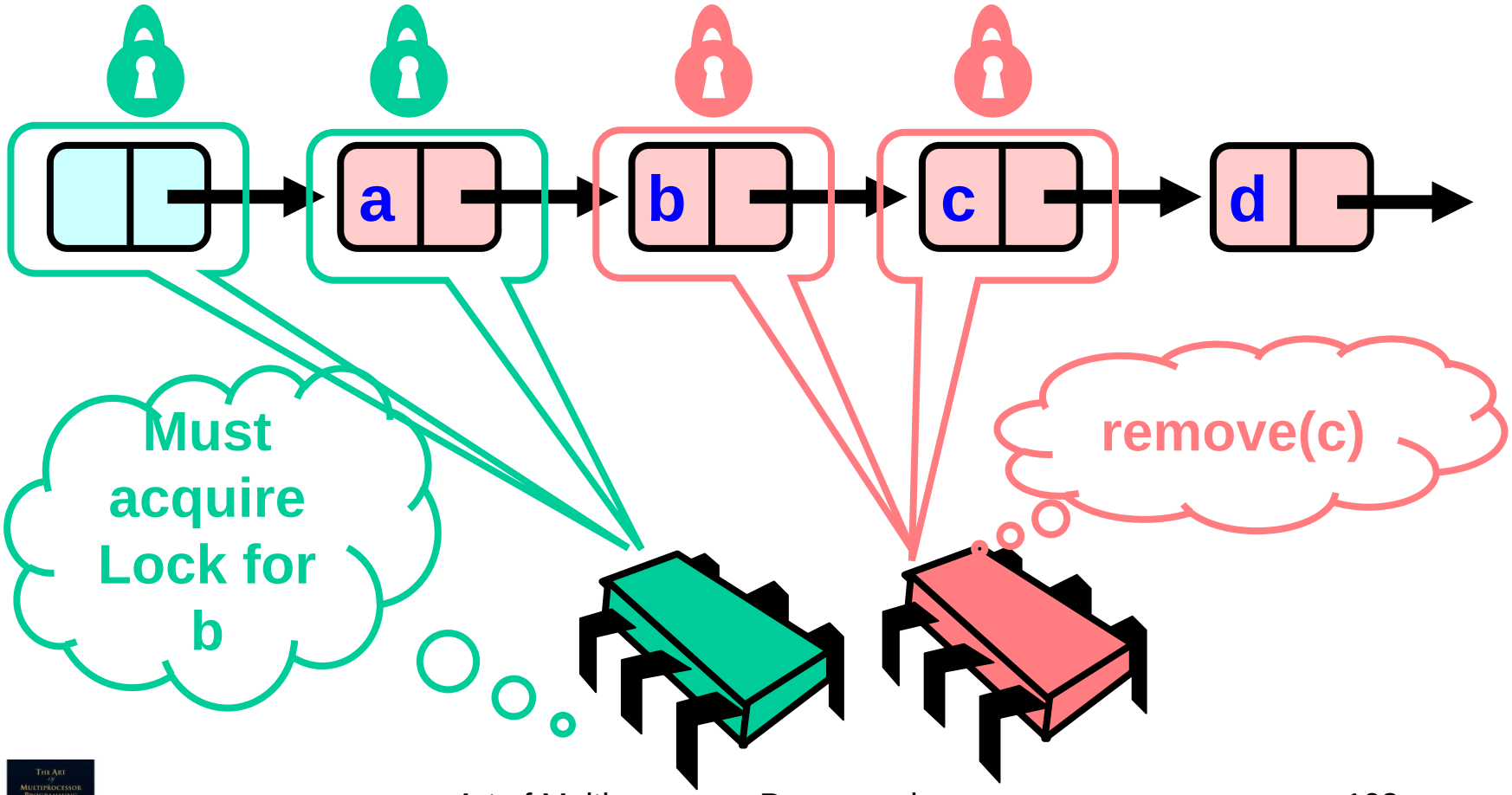


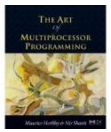
Removing a Node



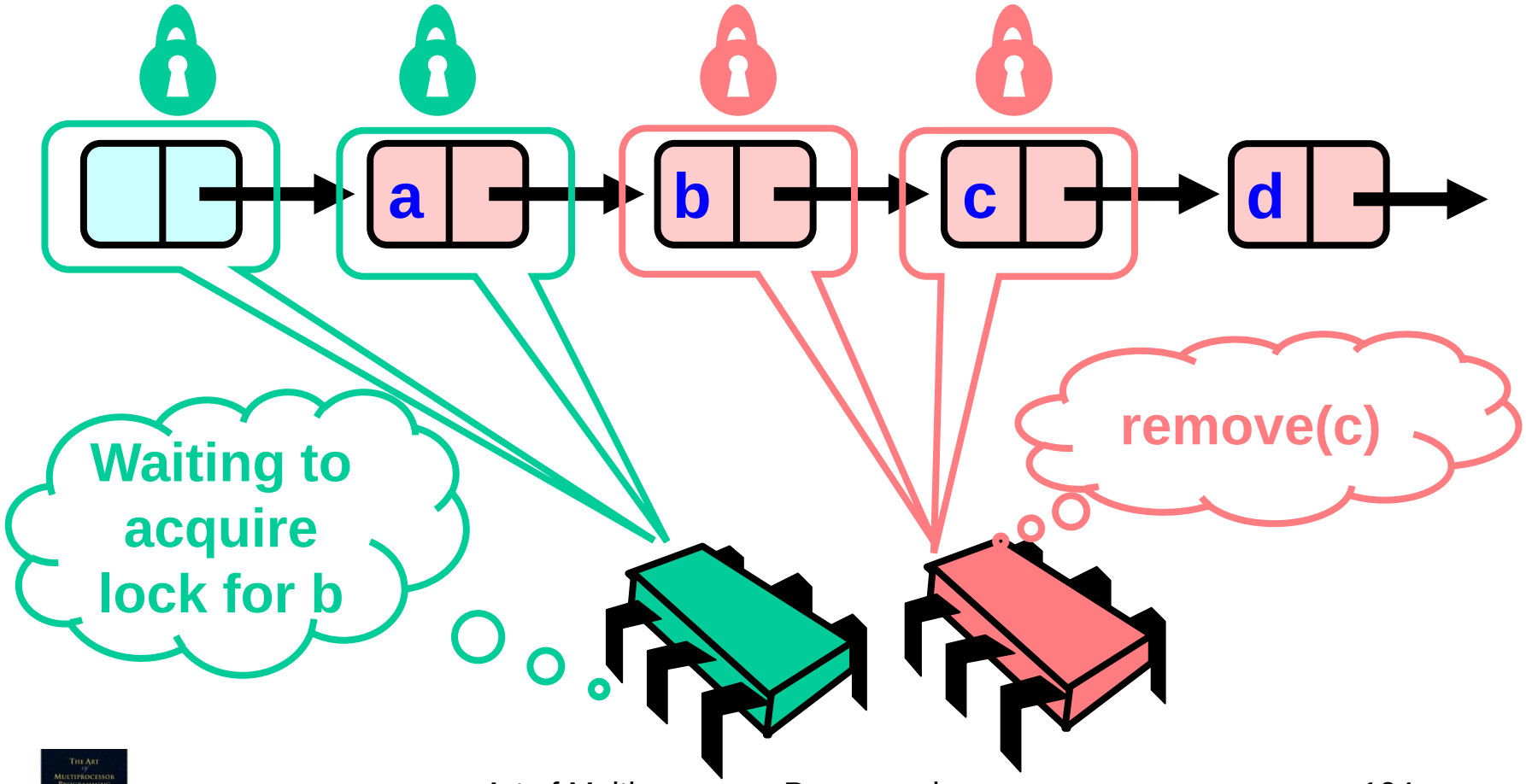


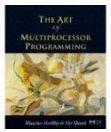
Removing a Node



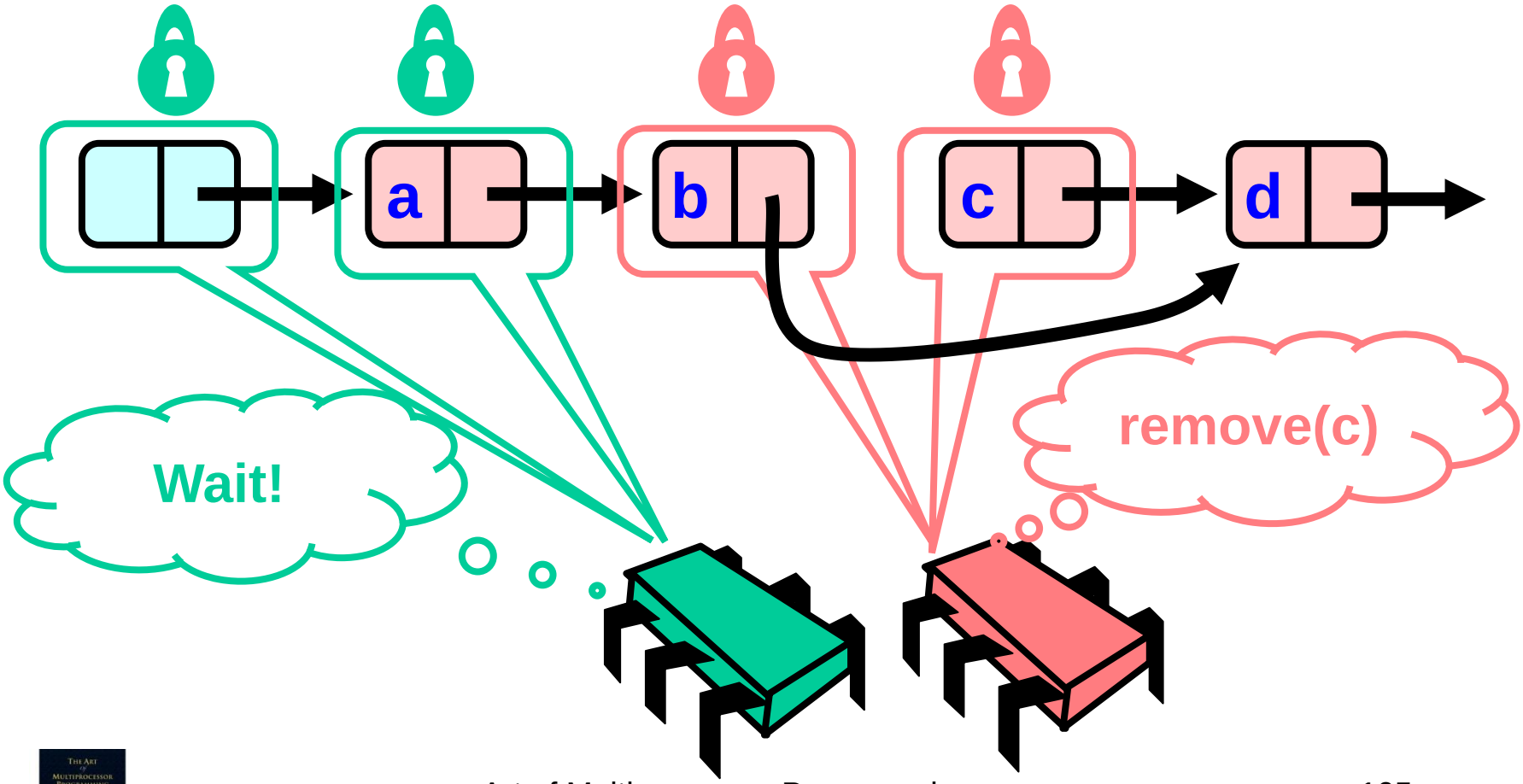


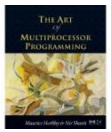
Removing a Node



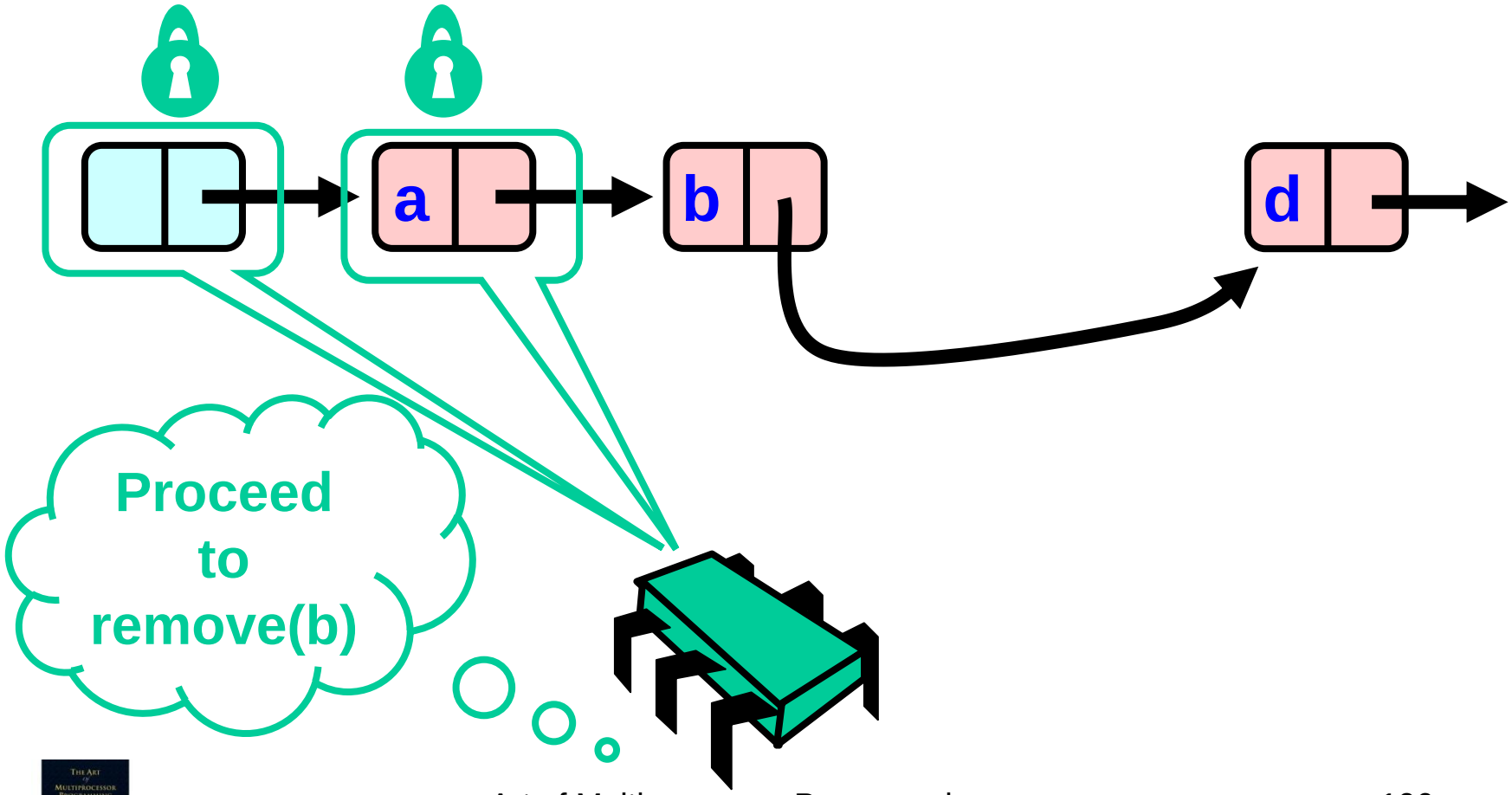


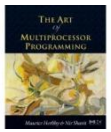
Removing a Node



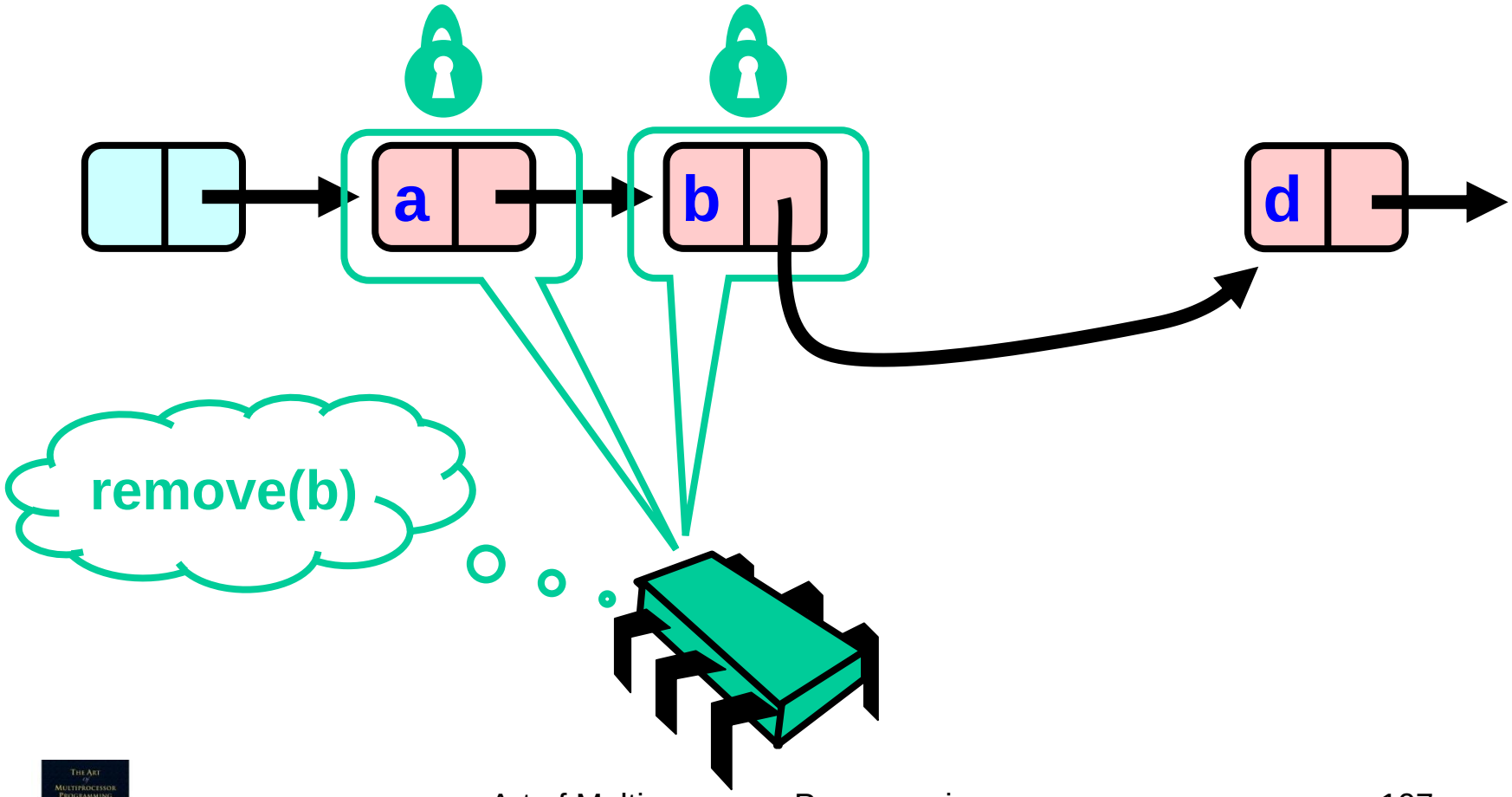


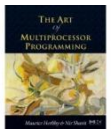
Removing a Node



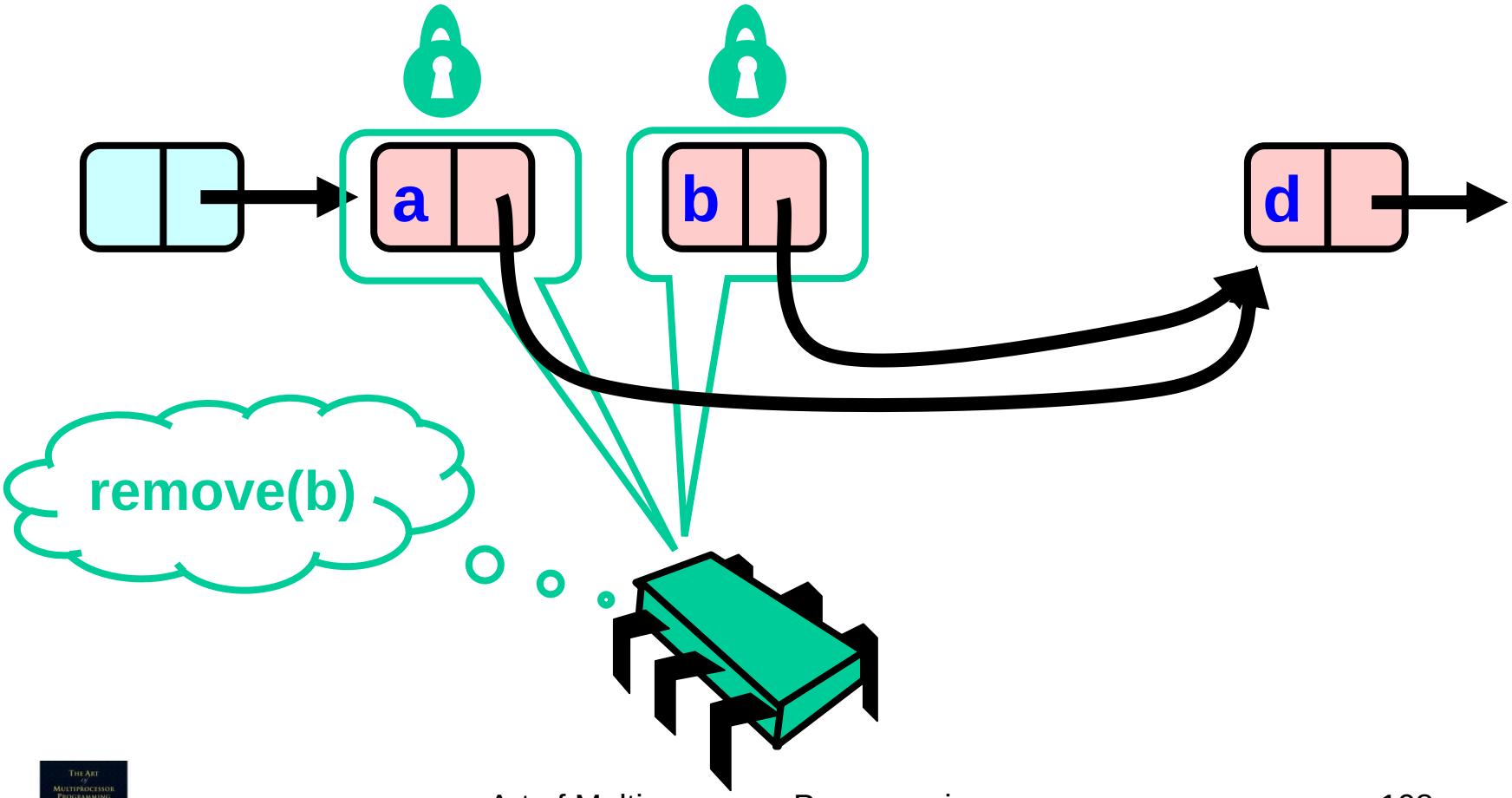


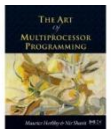
Removing a Node



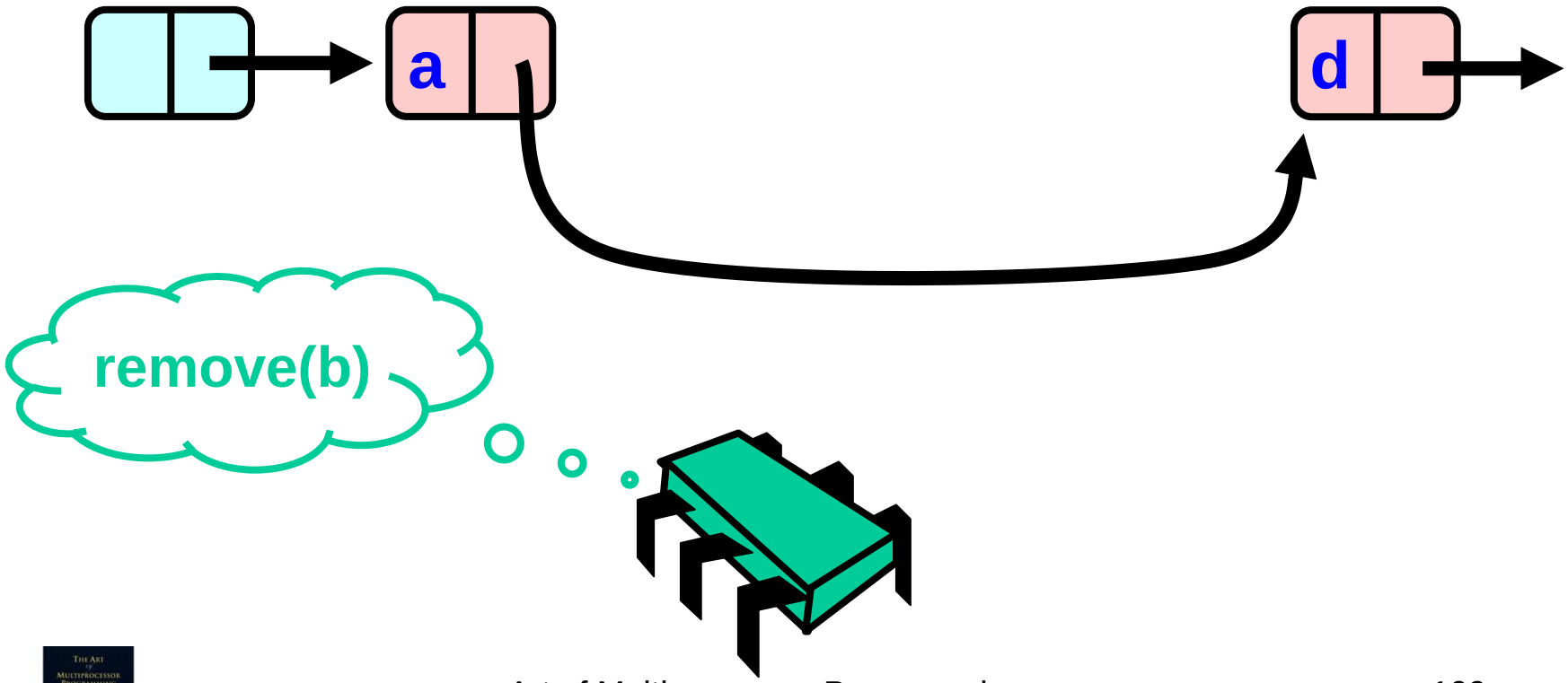


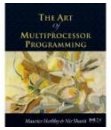
Removing a Node



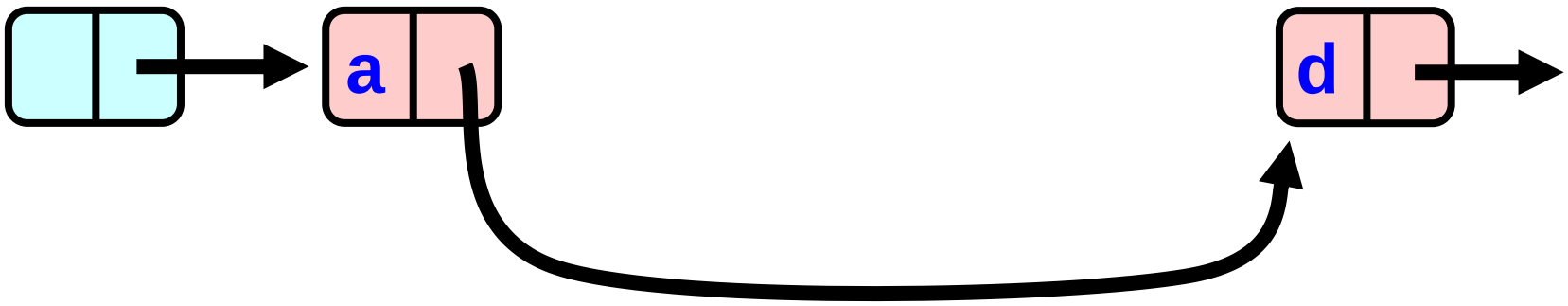


Removing a Node



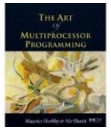


Removing a Node



Remove method

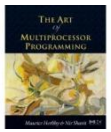
```
public boolean remove(T item) {
 int key = item.hashCode();
 Node pred, curr;
 try {
    …
 } finally {
  curr.unlock();
  pred.unlock();
 }}
```



Remove method

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

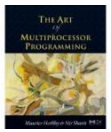
Key used to order node



Remove method

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

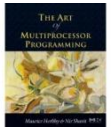
Predecessor and current nodes



Remove method

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

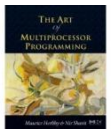
Make sure locks released



Remove method

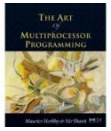
```
public boolean remove(T item) {
 int key = item.hashCode();
 Node pred, curr;
 try {
   ...
 } finally {
  curr.unlock();
  pred.unlock();
 }}
```

Everything else



Remove method

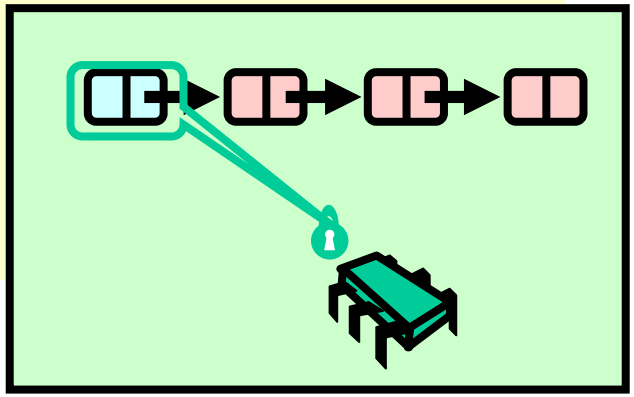
```
try {
 pred = head;
 pred.lock();
 curr = pred.next;
 curr.lock();
 …
} finally { … }
```

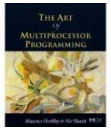


Remove method

lock pred == head

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

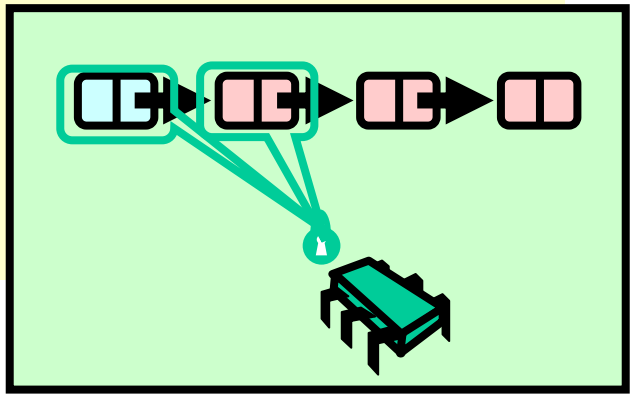


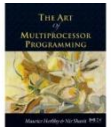


Remove method

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

Lock current

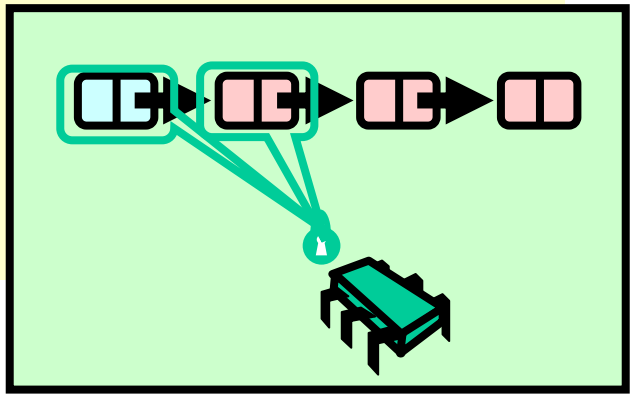




Remove method

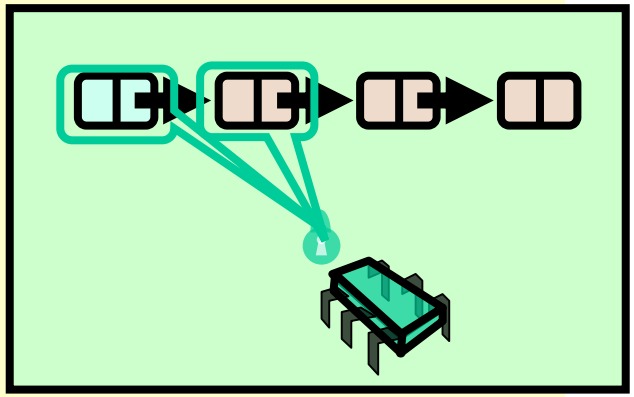
```
try {
 pred = head;
 pred.lock();
 curr = pred.next;
 curr.lock();
 …
} finally { … }
```

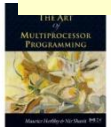
Traversing list



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;
```

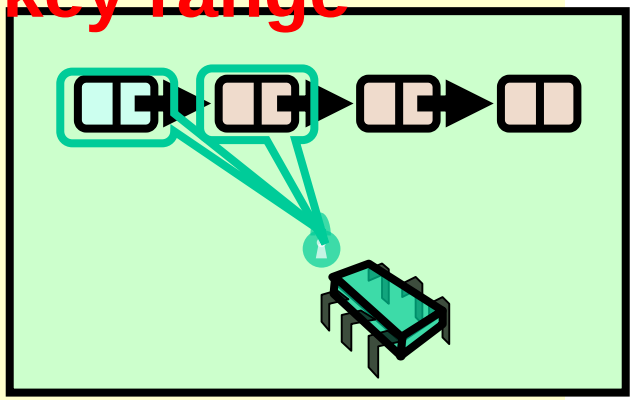


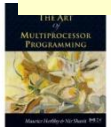


Remove: searching

```
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

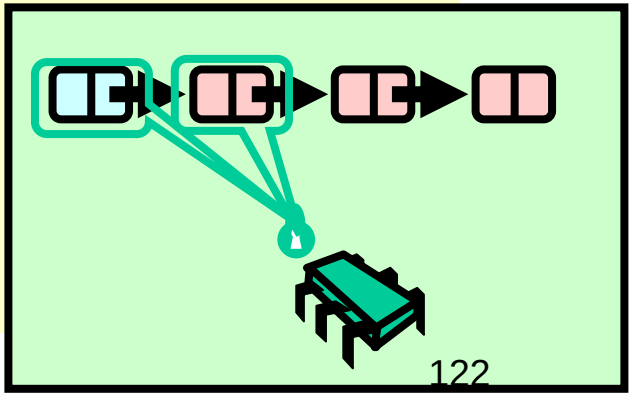


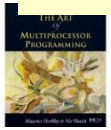


Remove: searching

```
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

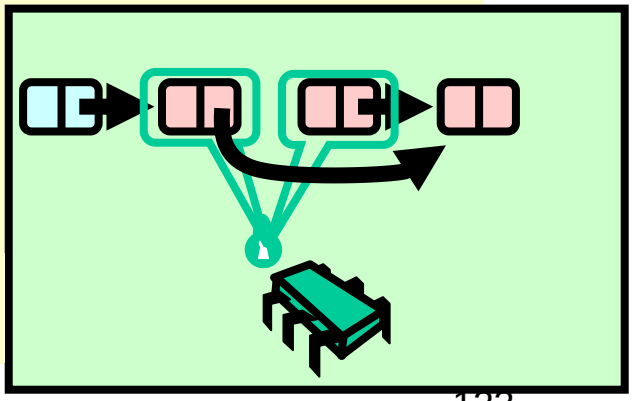


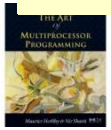


Remove: searching

```
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

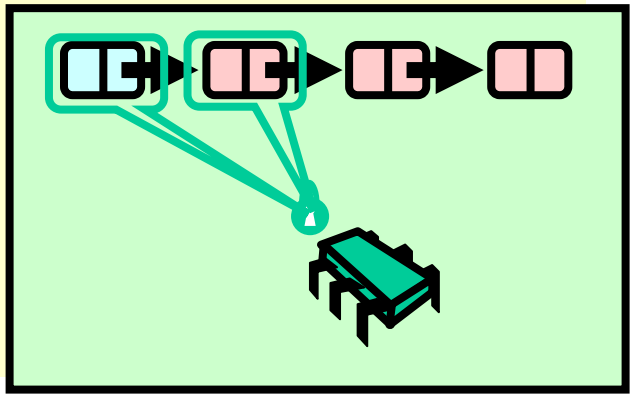


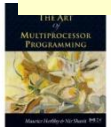


Remove: searching

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;
```

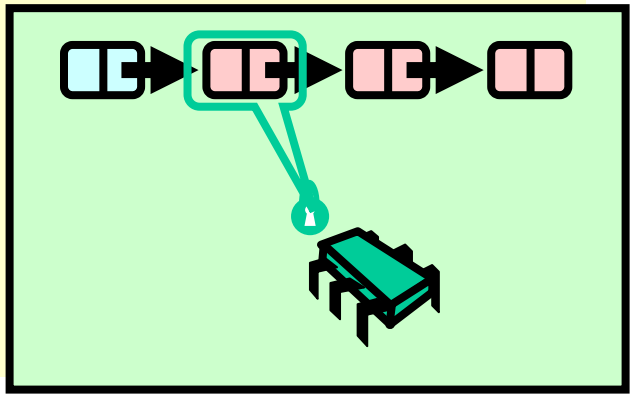


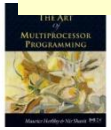


Remove: searching

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;
```

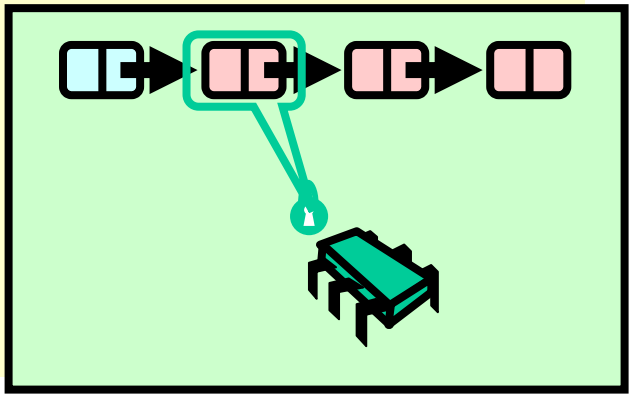


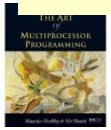


Remove: searching

```
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

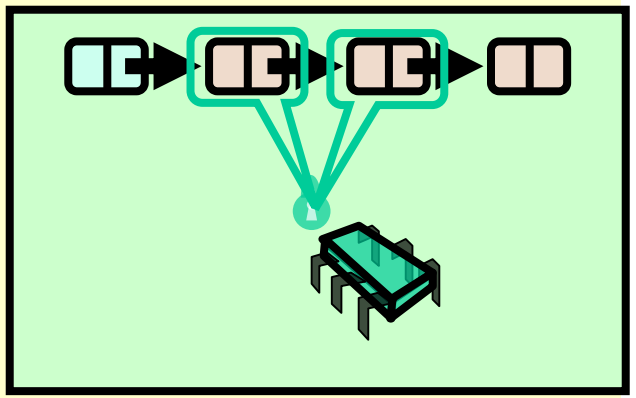




Remove: searching

```
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;
```

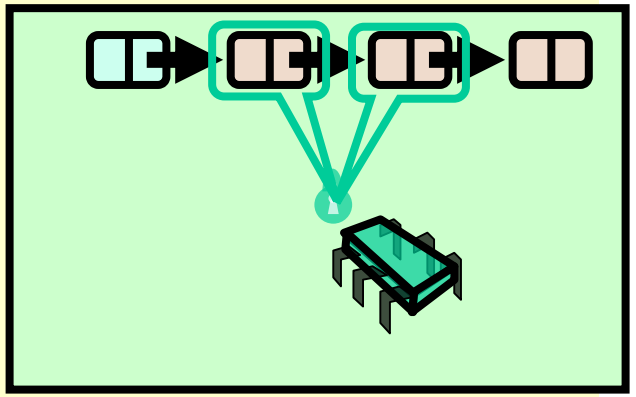
Find and lock new current



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;
```

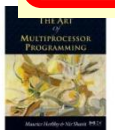
Lock invariant restored



Remove: searching

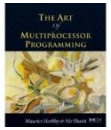
```
while (curr.key <= key) {
  if (item == curr.item) {
   pred.next = curr.next;
   return true;
  }
  pred.unlock();
  pred = curr;
  curr = curr.next;
  curr.lock();
 }
 return false;
```

Otherwise, not present



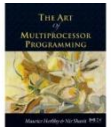
Why does this work?

- 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



Adding Nodes

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



Drawbacks

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