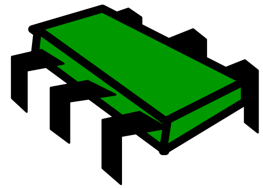
FOUNDATIONS OF CONCURRENT AND DISTRIBUTED SYSTEMS

– LINKED LISTS: LOCKING, LOCK-FREE, AND BEYOND –

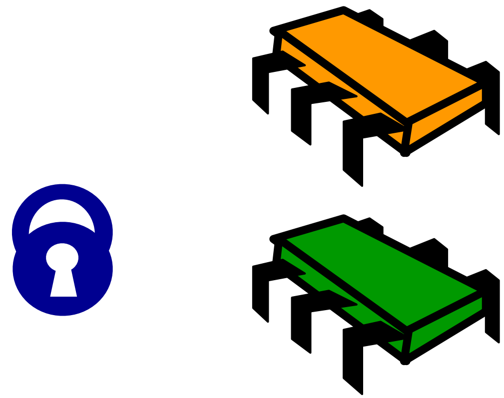
Prof. Christof Fetzer
TU Dresden

Today: Concurrent Objects

- Adding threads…
- Should **not lower throughput**
  - **Contention effects**
    - Mostly fixed by **queue locks**
  - **Overhead to acquire the lock:**
    - we look at **futex** a later lecture and briefly discuss queue locks.
- Should **increase throughput**
  - Not possible if inherently sequential
  - Surprising things are parallelizable

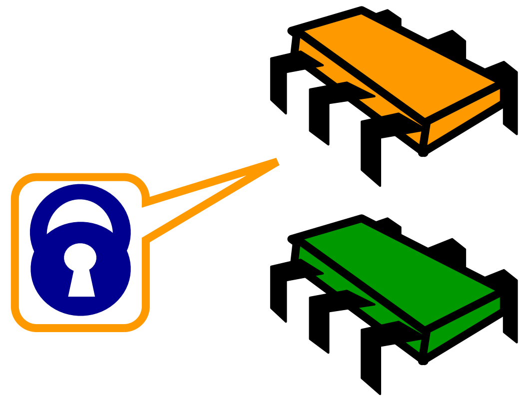
Today: Concurrent Objects



- Adding threads…
- Should **not lower throughput**
  - **Contention effects**
    - Mostly fixed by **queue locks**
  - **Overhead to acquire the lock:**
    - we look at **futex** a later lecture and briefly discuss queue locks.
- Should **increase throughput**
  - Not possible if inherently sequential
  - Surprising things are parallelizable

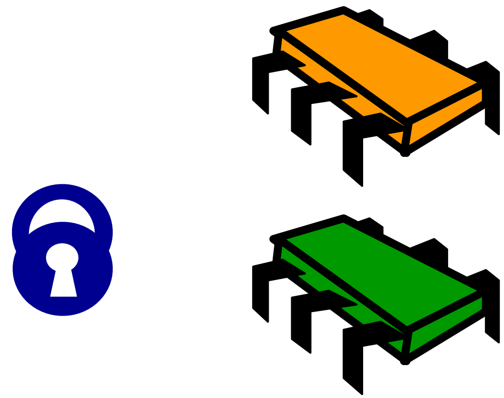
Today: Concurrent Objects

- Adding threads…
- Should **not lower throughput**
  - **Contention effects**
    - Mostly fixed by **queue locks**
  - **Overhead to acquire the lock:**
    - we look at **futex** a later lecture and briefly discuss queue locks.
- Should **increase throughput**
  - Not possible if inherently sequential
  - Surprising things are parallelizable

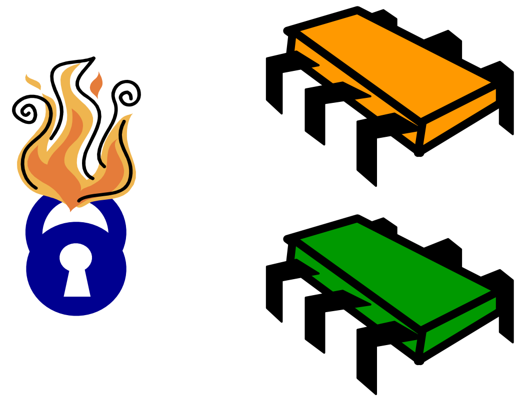


Today: Concurrent Objects

- Adding threads…
- Should **not lower throughput**
  - **Contention effects**
    - Mostly fixed by **queue locks**
  - **Overhead to acquire the lock:**
    - we look at **futex** a later lecture and briefly discuss queue locks.
- Should **increase throughput**
  - Not possible if inherently sequential
  - Surprising things are parallelizable

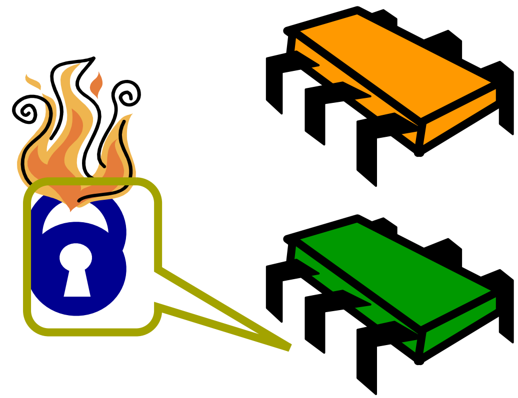


Today: Concurrent Objects



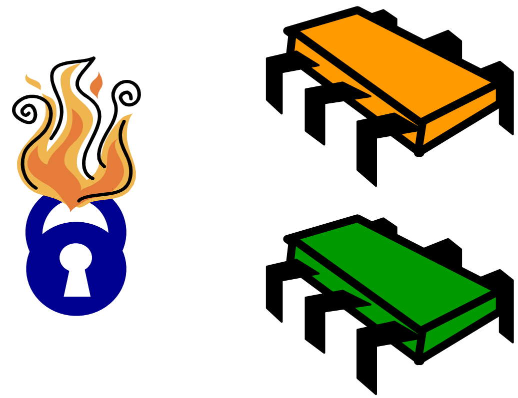
- Adding threads…
- Should **not lower throughput**
  - **Contention effects**
    - Mostly fixed by **queue locks**
  - **Overhead to acquire the lock:**
    - we look at **futex** a later lecture and briefly discuss queue locks.
- Should **increase throughput**
  - Not possible if inherently sequential
  - Surprising things are parallelizable

Today: Concurrent Objects



- Adding threads…
- Should **not lower throughput**
  - **Contention effects**
    - Mostly fixed by **queue locks**
  - **Overhead to acquire the lock:**
    - we look at **futex** a later lecture and briefly discuss queue locks.
- Should **increase throughput**
  - Not possible if inherently sequential
  - Surprising things are parallelizable

Today: Concurrent Objects



- Adding threads…
- Should **not lower throughput**
  - **Contention effects**
    - Mostly fixed by **queue locks**
  - **Overhead to acquire the lock:**
    - we look at **futex** a later lecture and briefly discuss queue locks.
- Should **increase throughput**
  - Not possible if inherently sequential
  - Surprising things are parallelizable

Coarse-Grained Synchronization

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

Coarse-Grained Synchronization

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

This Lecture

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

1. Fine-Grained Synchronization

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

2. Optimistic Synchronization

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

3. Lazy Synchronization

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

4. Lock-Free Synchronization

- Do not use locks at all
  - Use compareAndSet() and relatives…
- Advantages
  - Robust against asynchrony
- Disadvantages
  - Complex
  - Sometimes high overhead

Wait-Free Implementations

Definition: An object implementation is wait-free if every thread completes a method in a finite number of steps

No mutual exclusion

- Thread could halt in critical section

Lock-Free Implementations

Definition: An object implementation is lock-free if in an infinite execution infinitely often some method call finishes (obviously, in a finite number of steps)

No difference between lock-free and wait-free for finite executions

Example: Set Properties

- Collection of objects
- No duplicates
- Methods
  - add() a new object
  - remove() an object
  - Test if set contains() object

Set Interface

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

Set Interface

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

Add object to set

Set Interface

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

Remove object from set

Set Interface

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

Is object in set?

Linked List

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

List Node

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

List Node

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

Object of interest

List Node

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

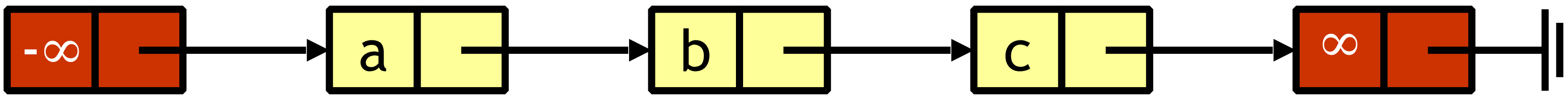
Usually hash code

List Node

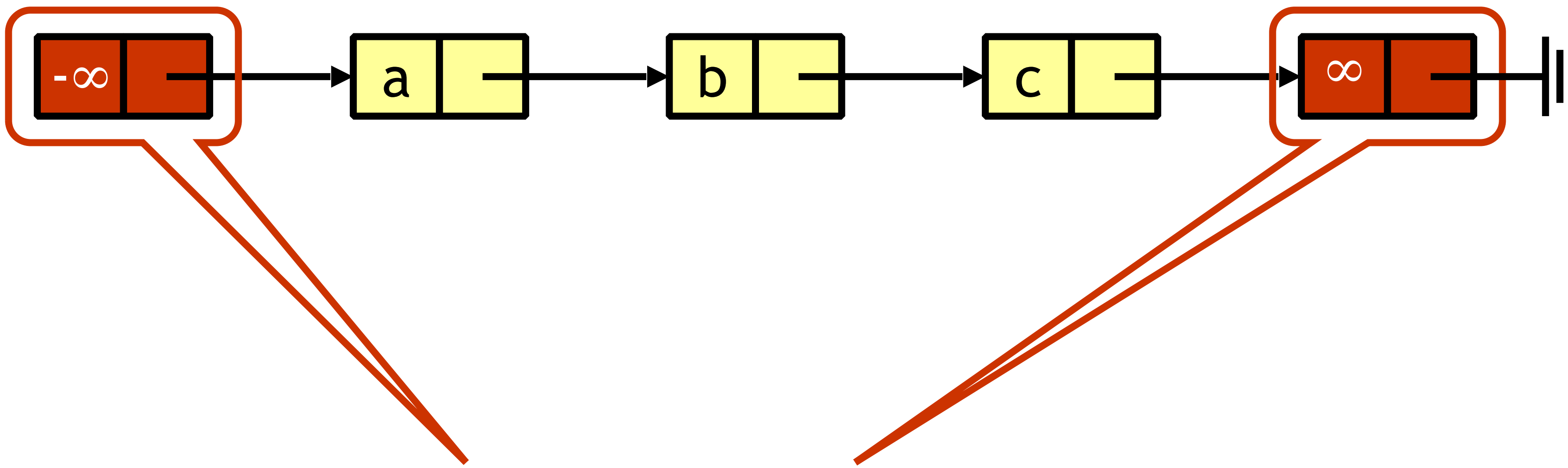
```
public class Node {
  Object object;
  int key;
  Node next;
}
```

Reference to next node

The List-Based Set



The List-Based Set



Ordered + sentinel nodes
(min & max possible keys)

Reasoning about Concurrent Data Structures

- Identify invariants
  - Properties that always holds
- True when object is created
- Truth preserved by each method
  - add(), remove(), contains()
  - Each step of each method
- Most steps are trivial
  - Usually one step tricky
  - Often linearization point

Interference

- Proof that invariants are preserved works if methods considered are the only modifiers
- Language encapsulation helps
  - List nodes not visible outside class
- Freedom from interference needed even for removed nodes
  - Some algorithms traverse removed nodes
  - Careful with malloc() and free()!
  - Garbage-collection helps here

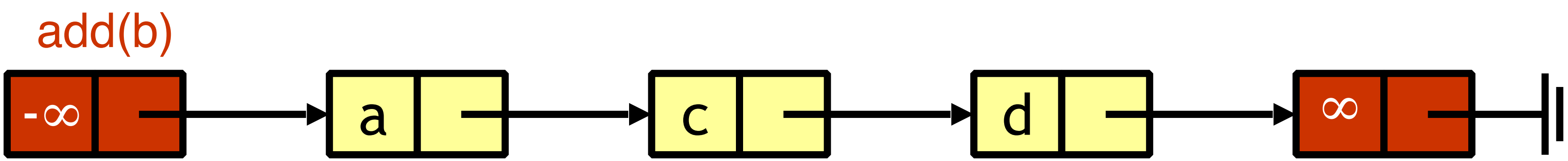
Blame Game

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

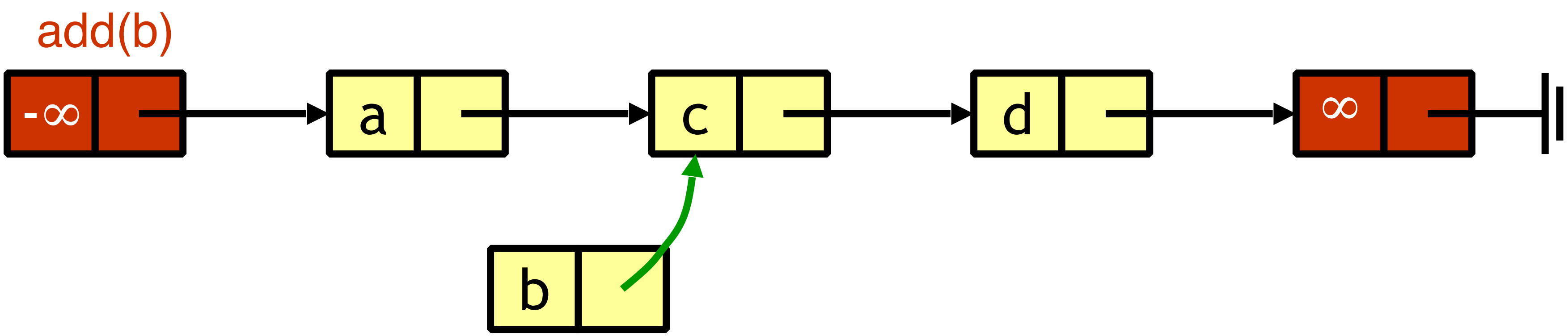
Set Invariant (Partly)

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

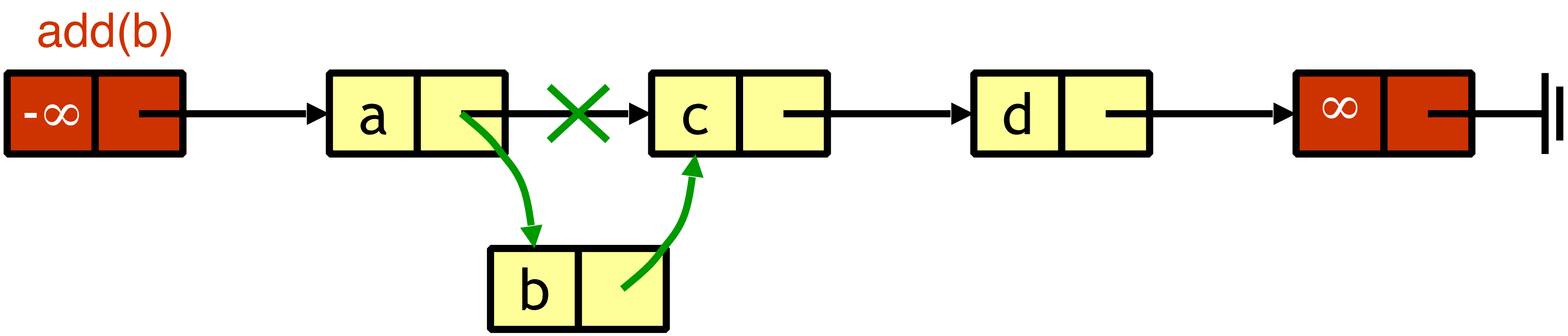
Sequential List Based Set



Sequential List Based Set

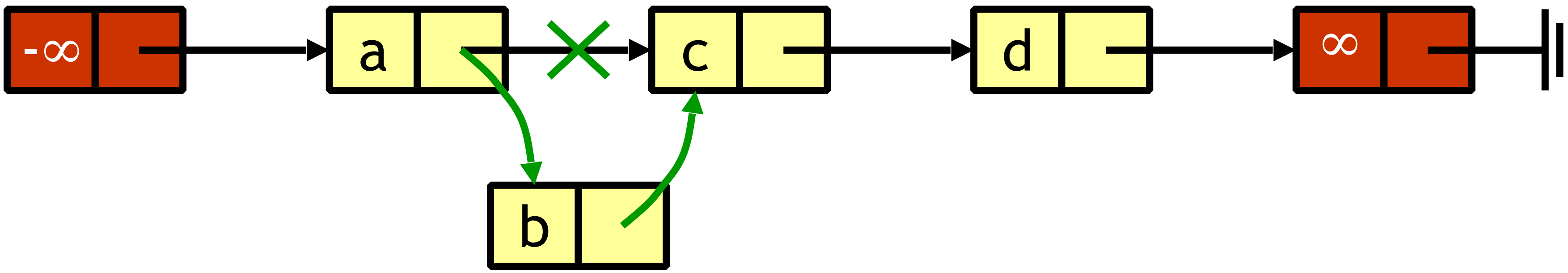


Sequential List Based Set

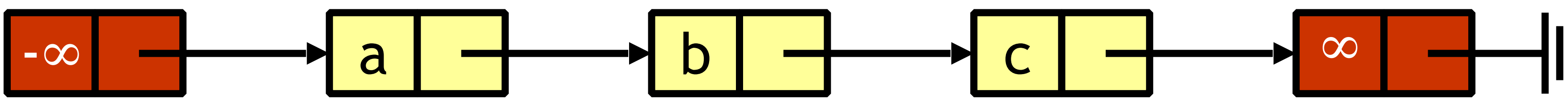


Sequential List Based Set

add(b)

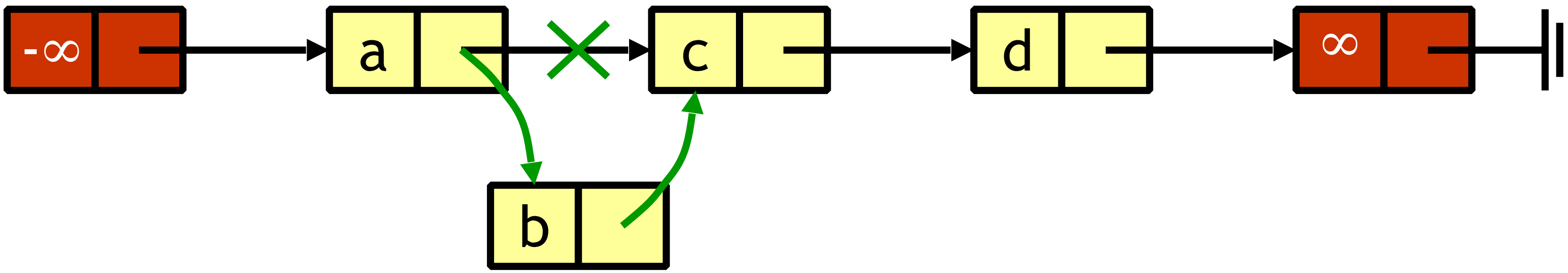


remove(b)

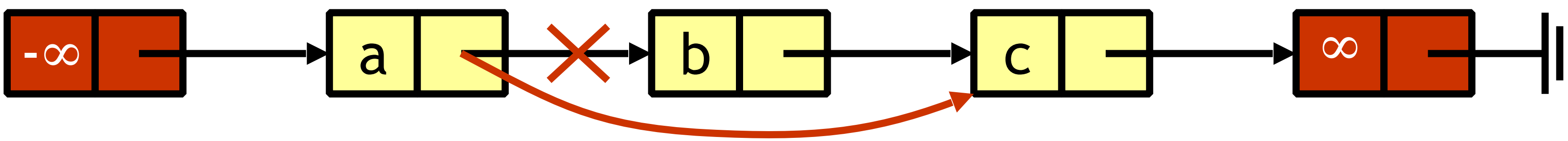


Sequential List Based Set

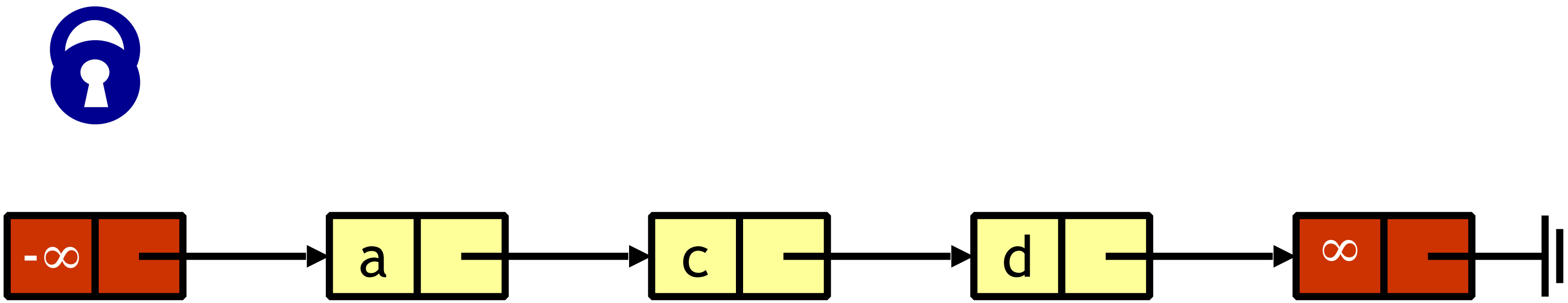
add(b)



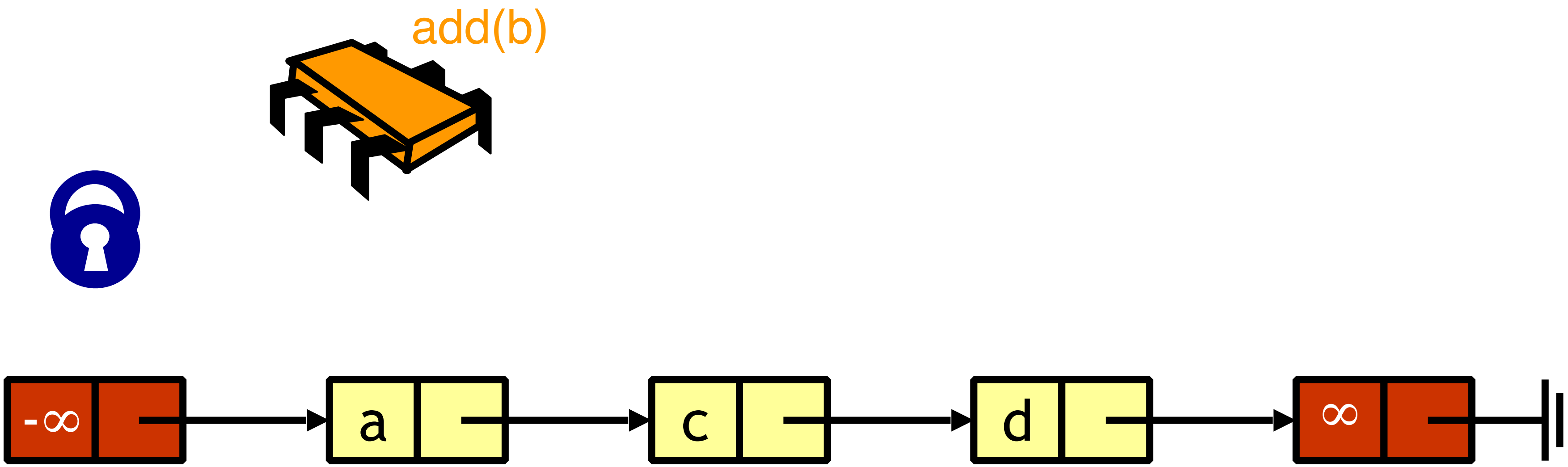
remove(b)



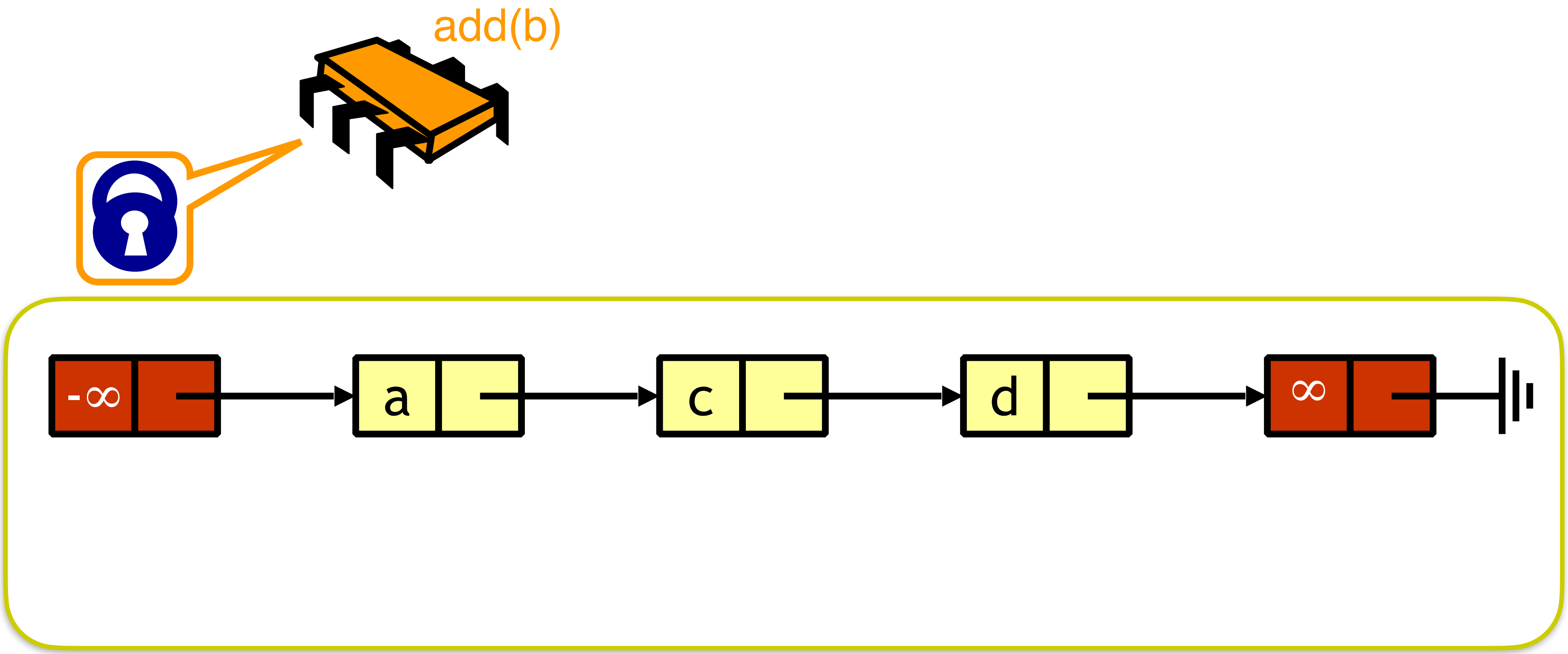
Coarse-Grained Locking



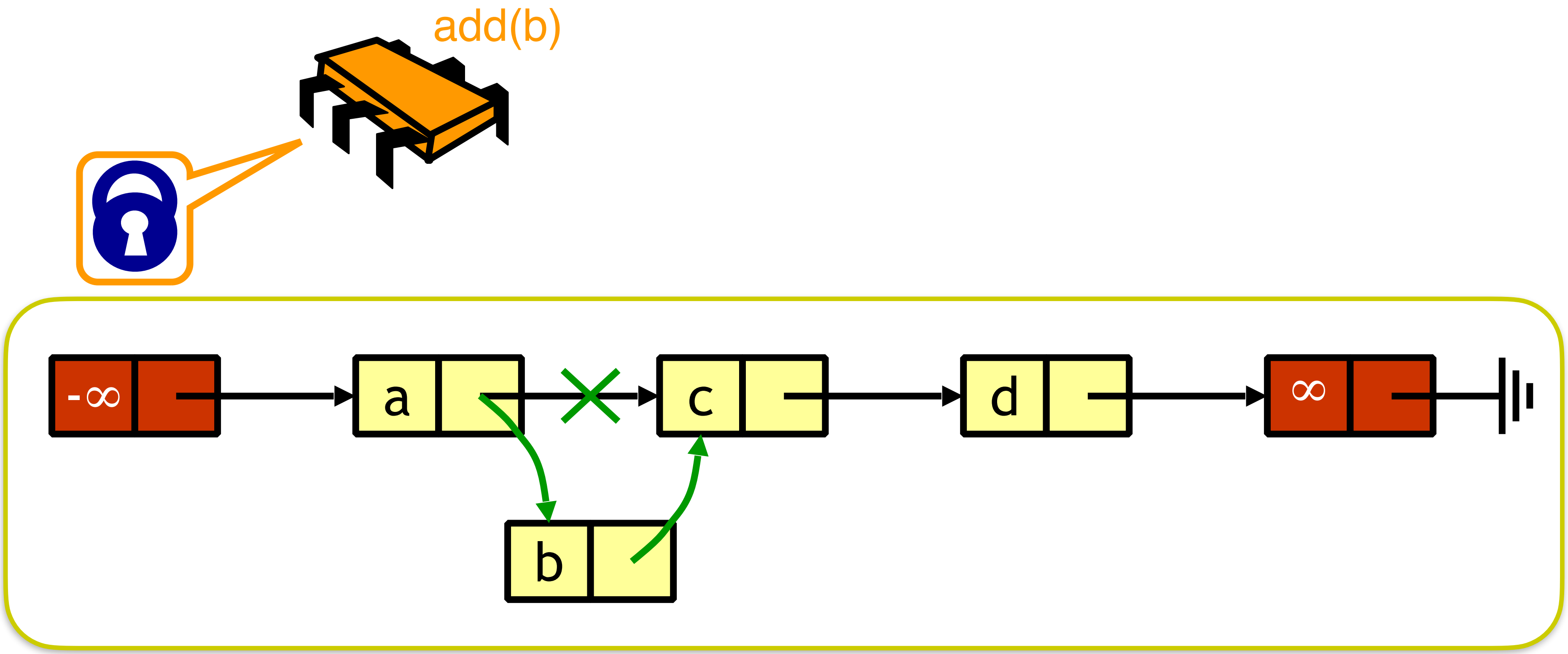
Coarse-Grained Locking



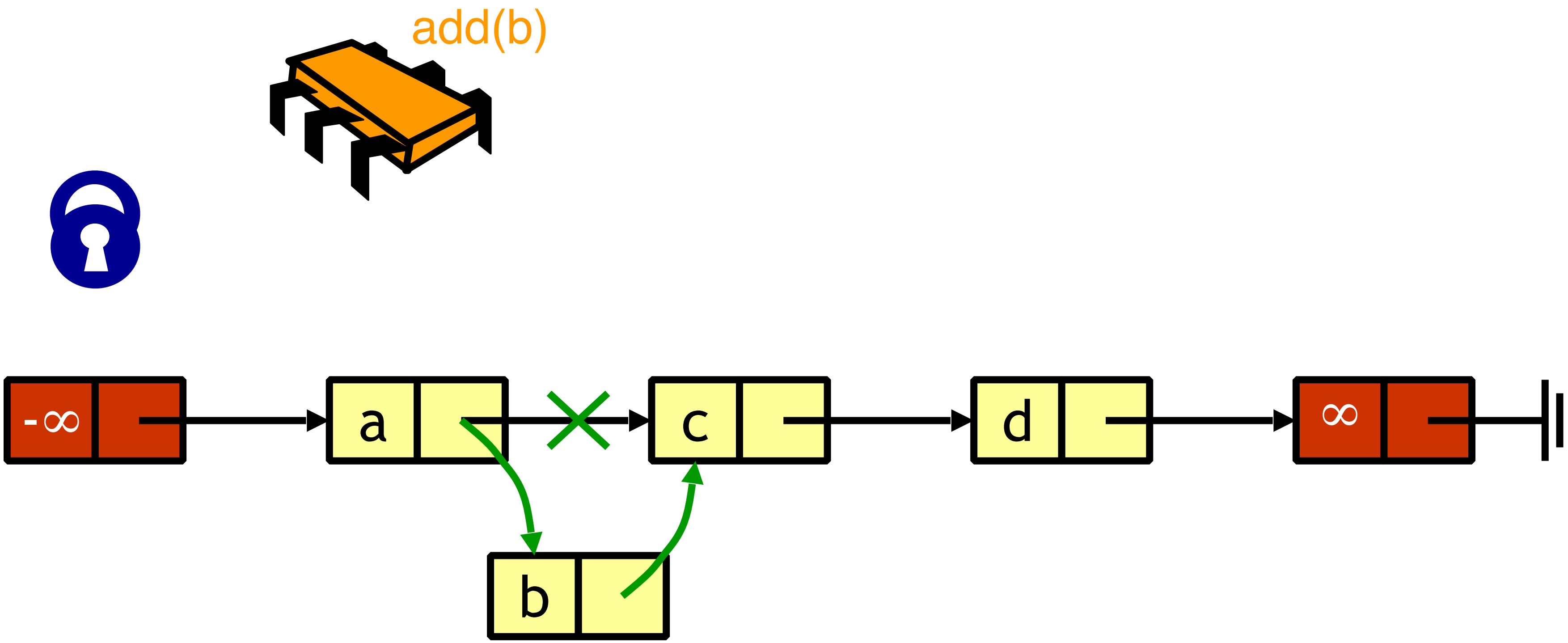
Coarse-Grained Locking



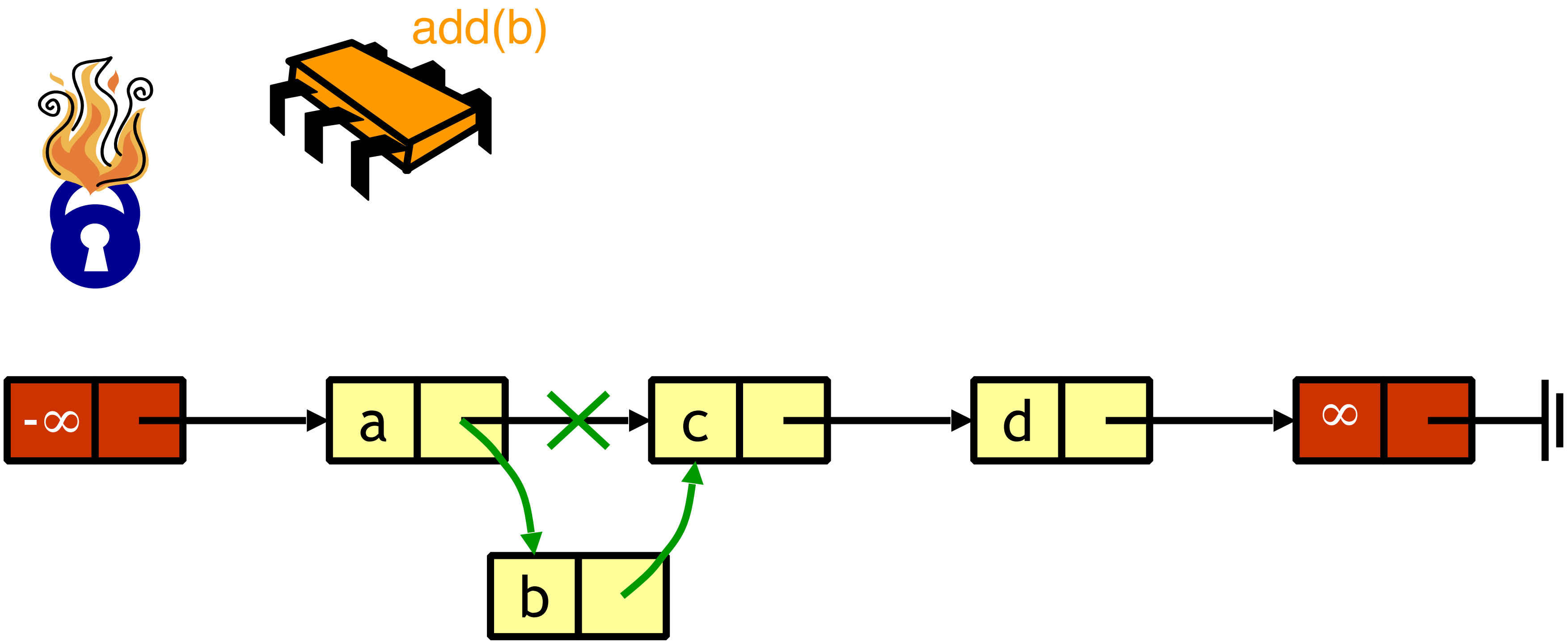
Coarse-Grained Locking



Coarse-Grained Locking



Coarse-Grained Locking



Simple but hotspot + bottleneck

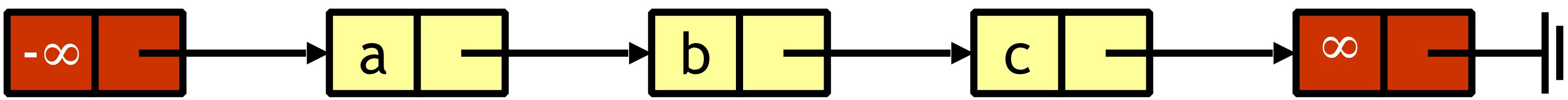
1. Coarse-Grained Locking

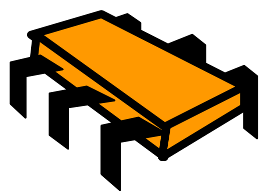
- Easy, same as synchronized methods
  - "One lock to rule them all…"
- Simple, clearly correct
  - Deserves respect!
- Works poorly with contention
  - Queue locks help
  - But bottleneck still an issue

2. Fine-grained Locking

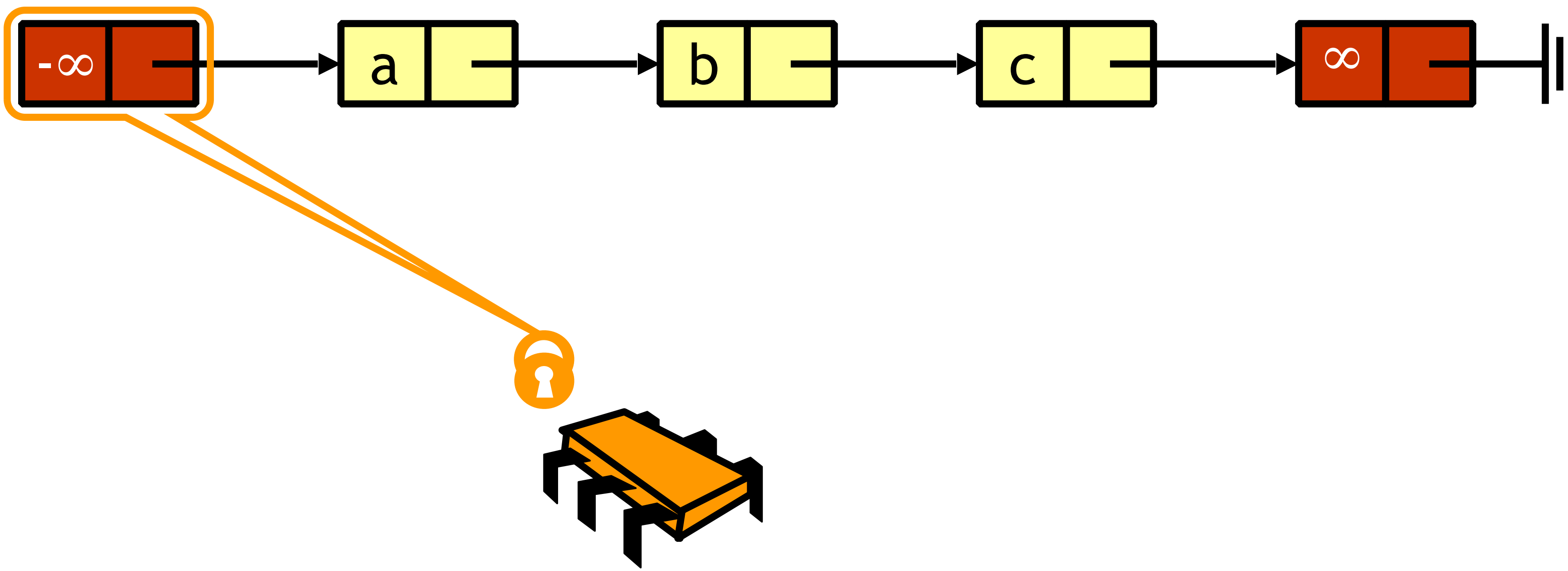
- Requires careful thought
  - "Do not meddle in the affairs of wizards, for they are subtle and quick to anger"
- Split object into pieces
  - Each piece has own lock
  - Methods that work on disjoint pieces need not exclude each other

Hand-over-Hand Locking

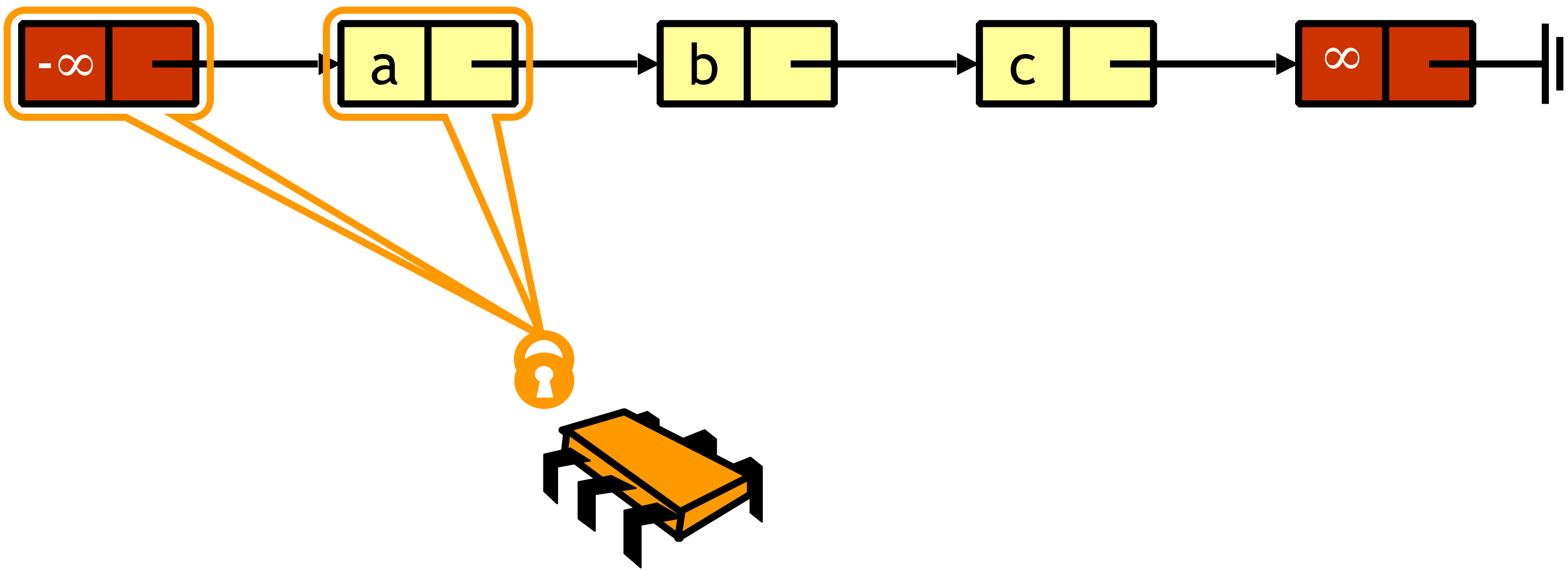




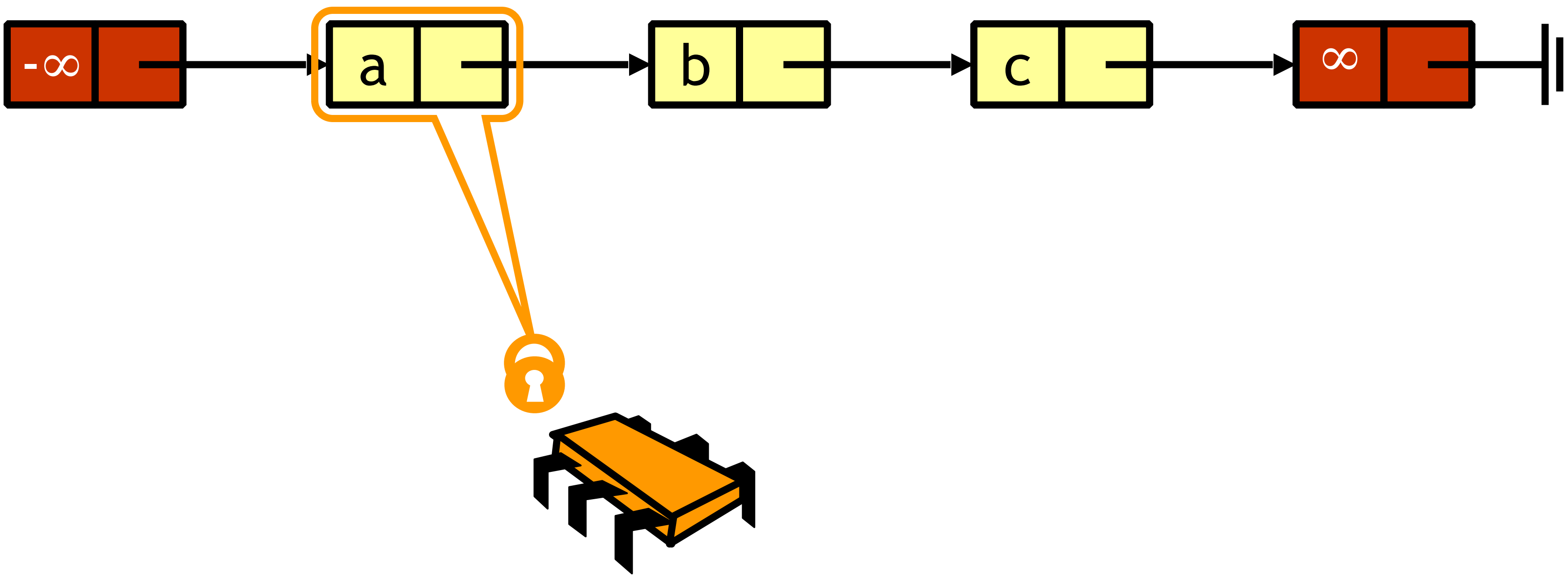
Hand-over-Hand Locking



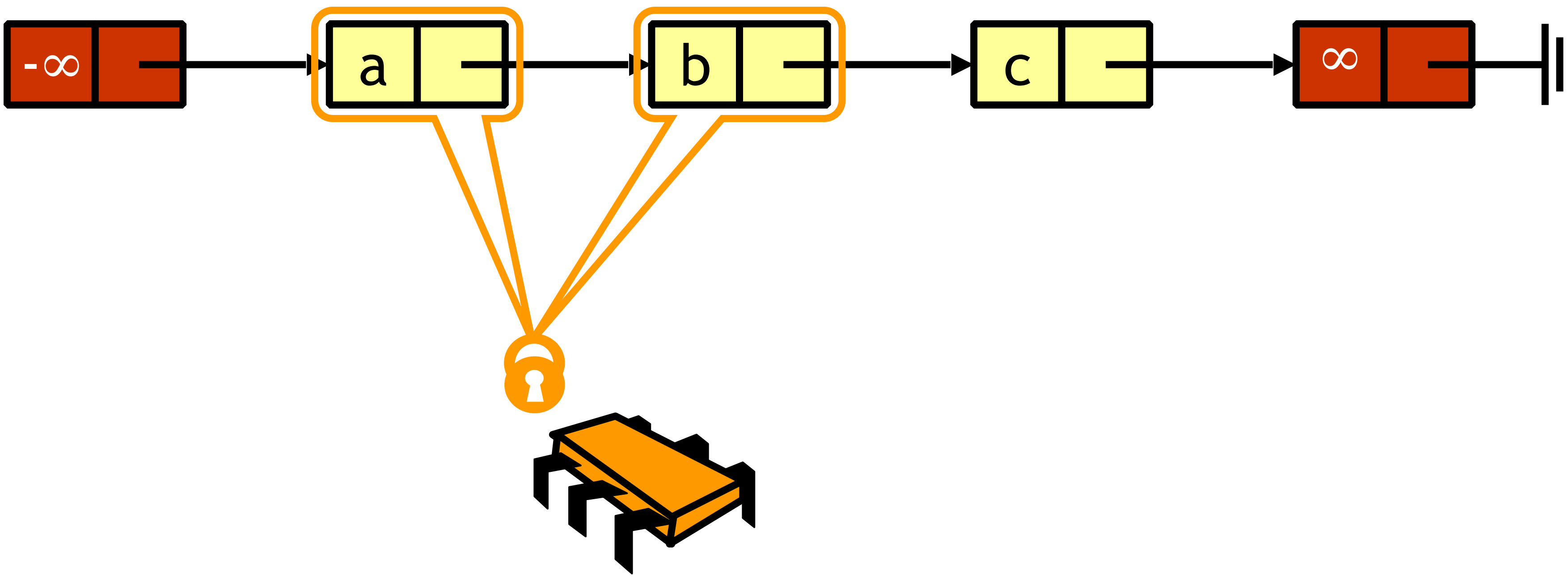
Hand-over-Hand Locking



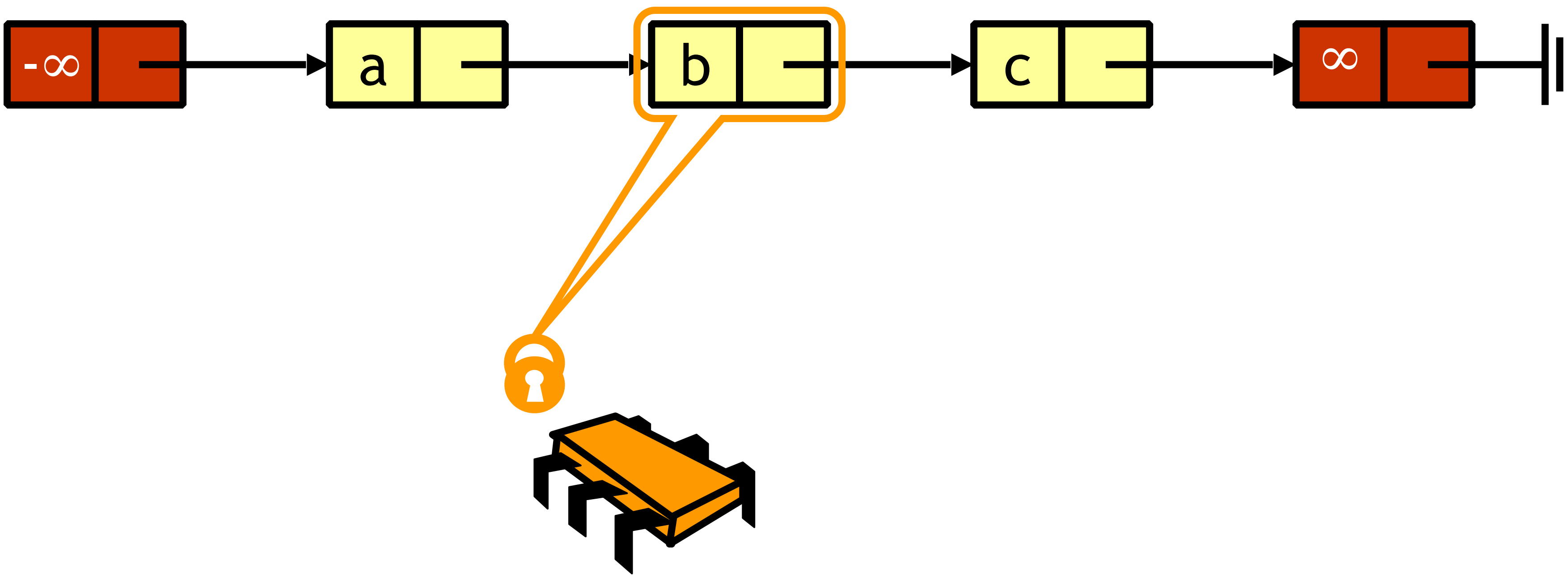
Hand-over-Hand Locking



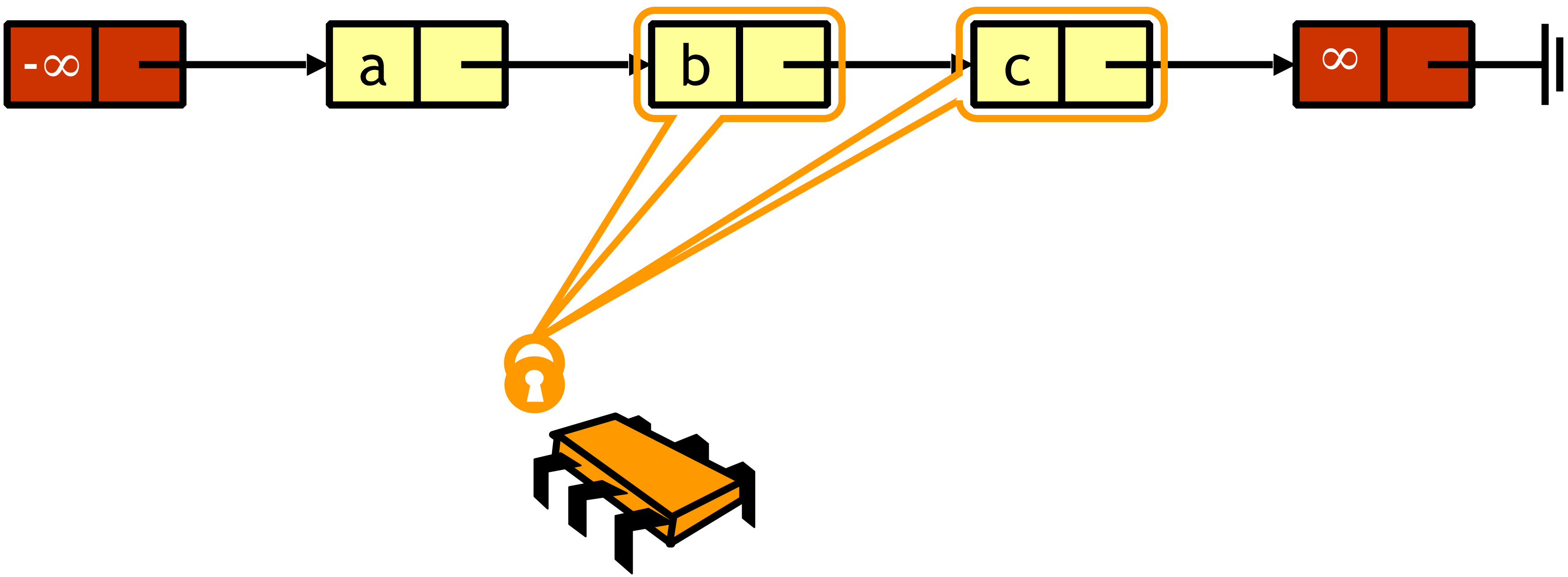
Hand-over-Hand Locking



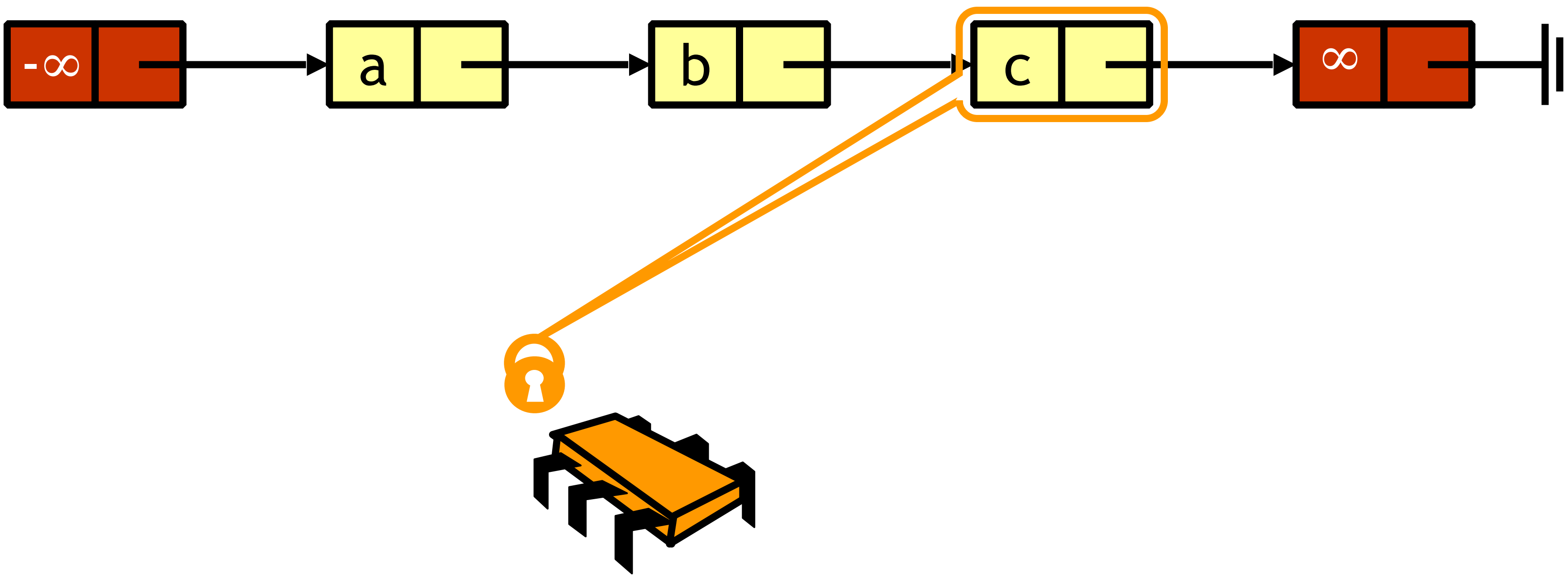
Hand-over-Hand Locking



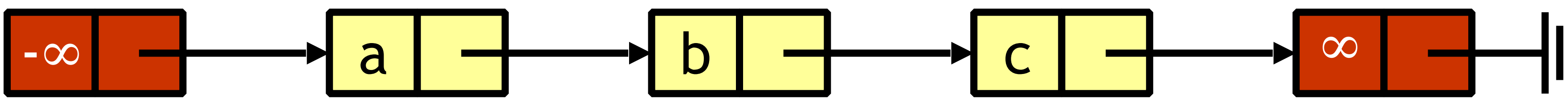
Hand-over-Hand Locking

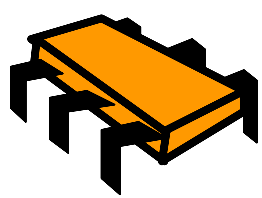


Hand-over-Hand Locking

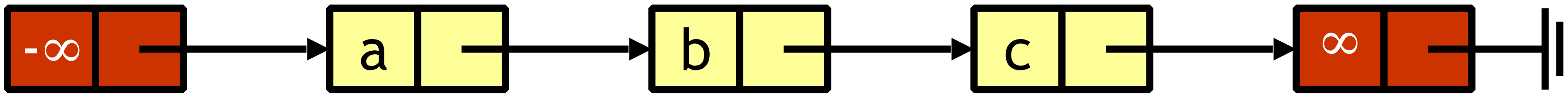


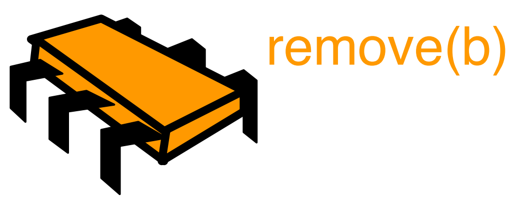
Removing a Node



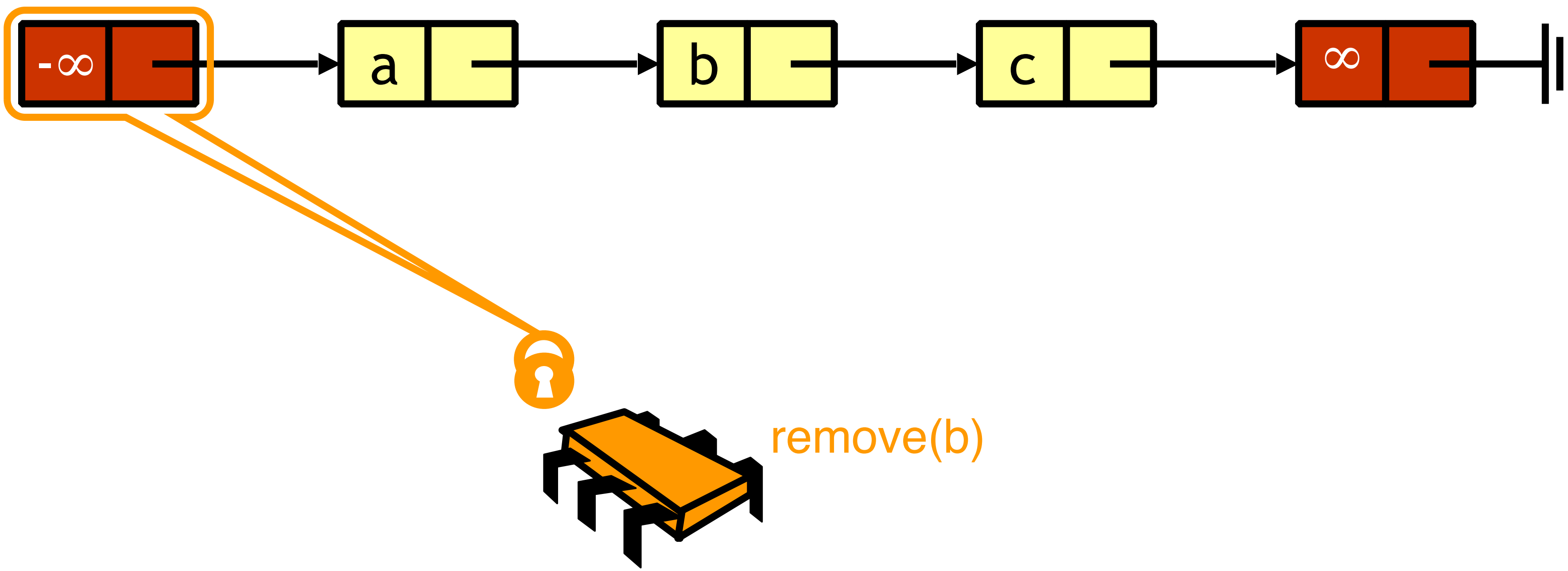


Removing a Node

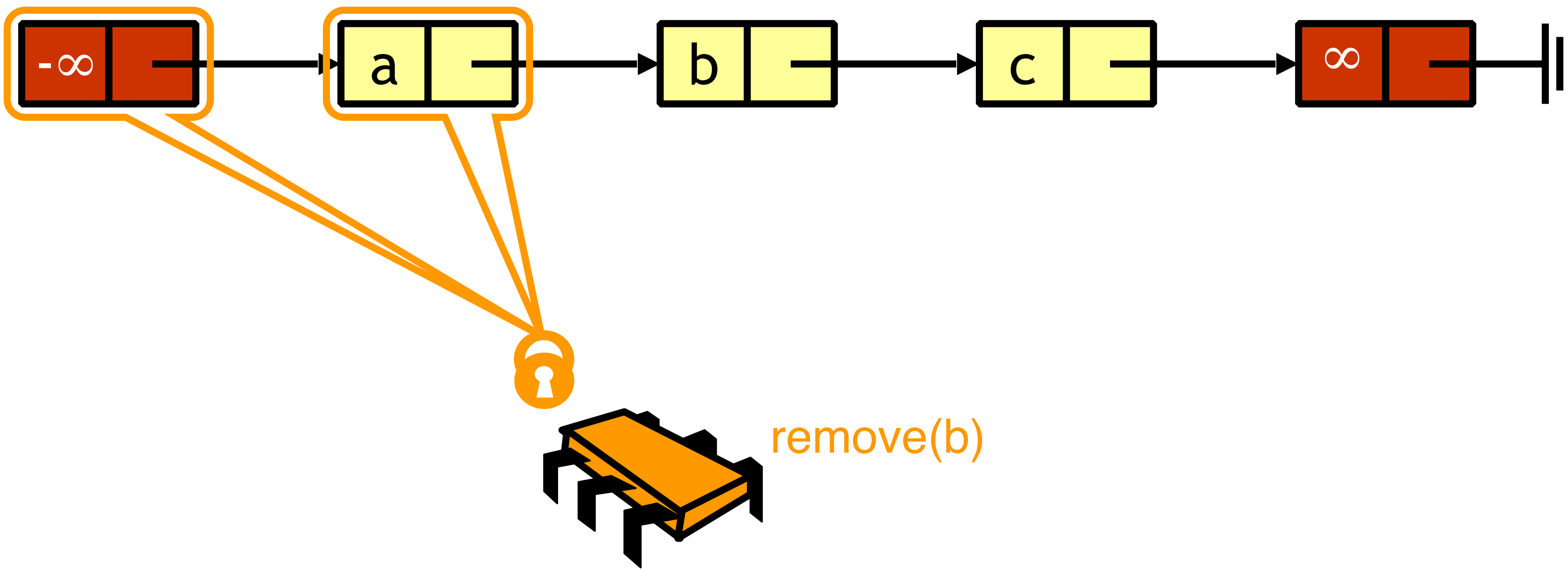




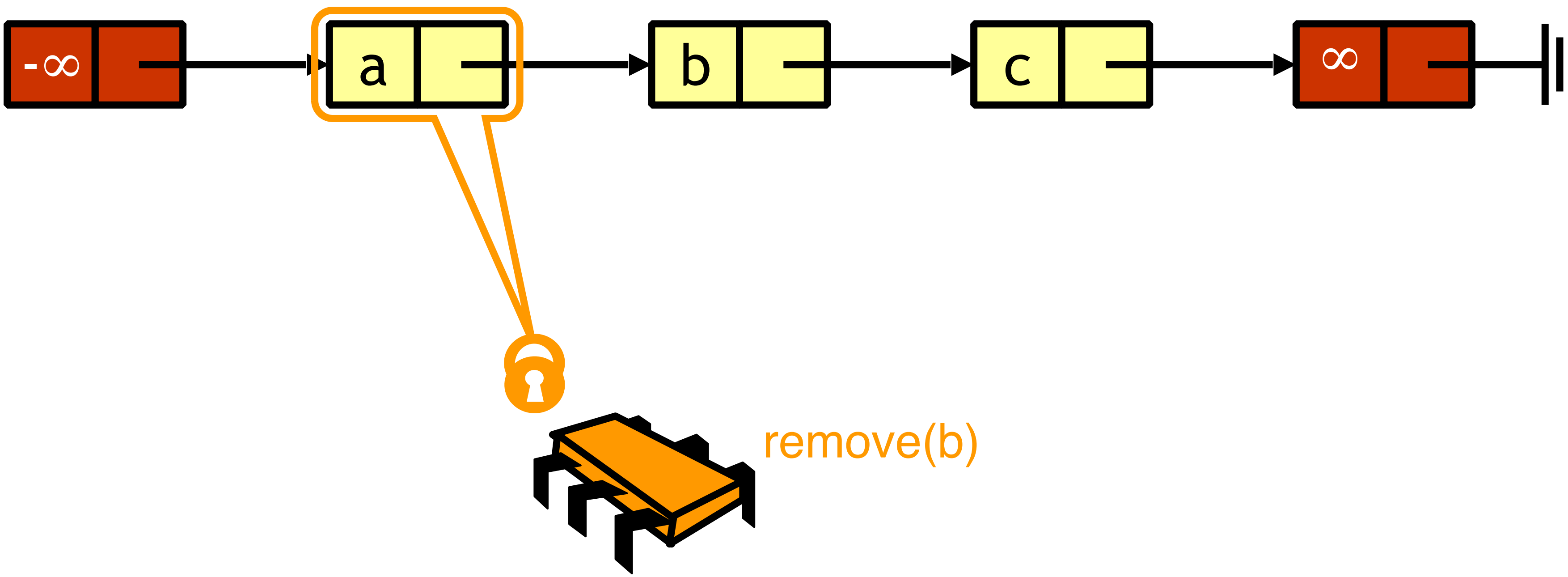
Removing a Node



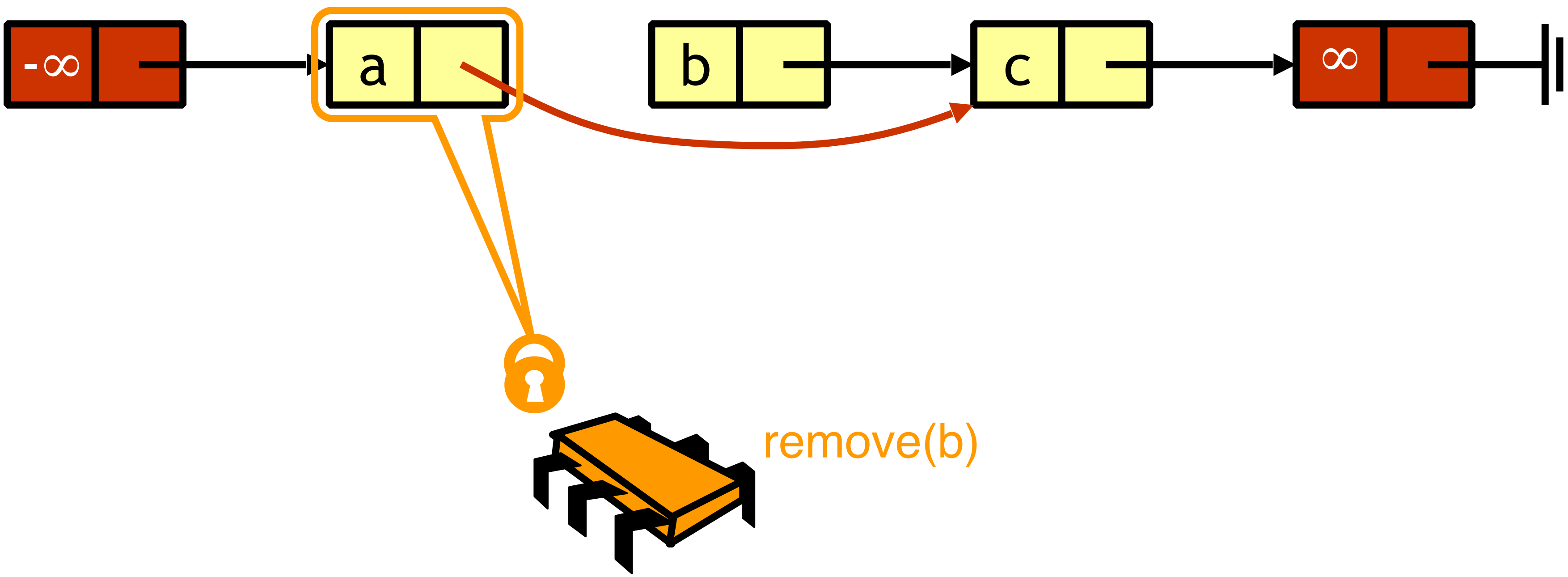
Removing a Node



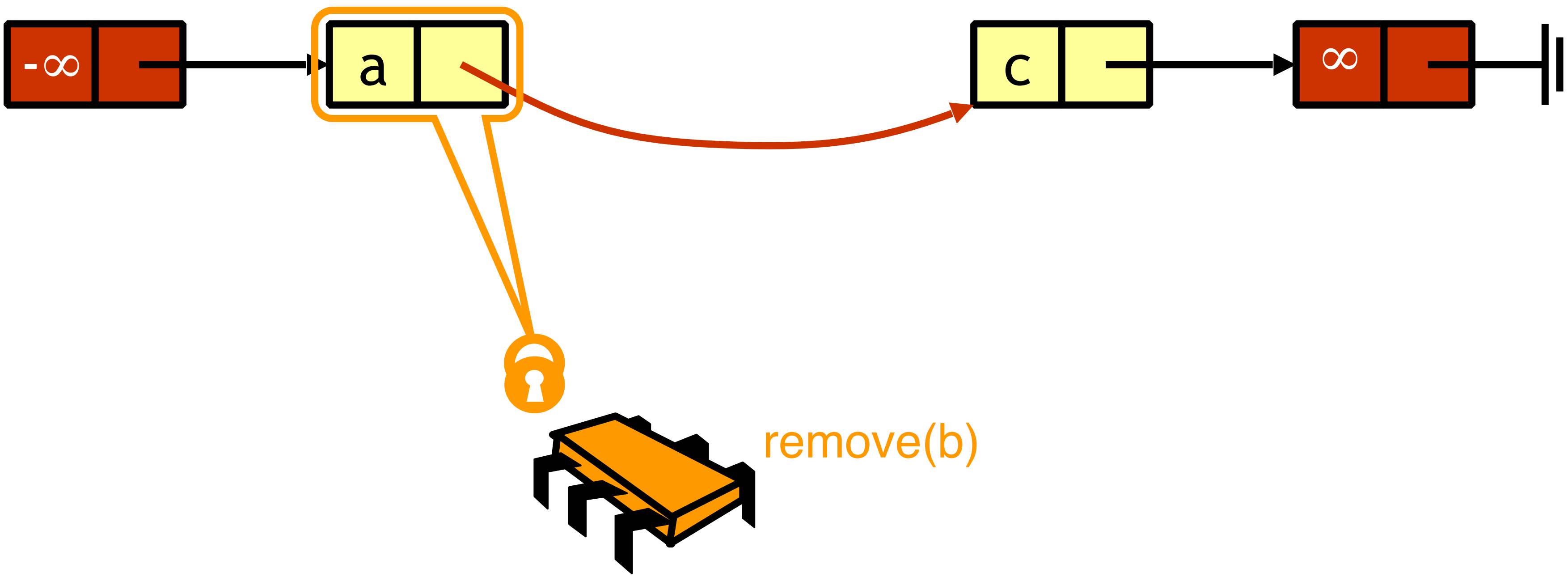
Removing a Node



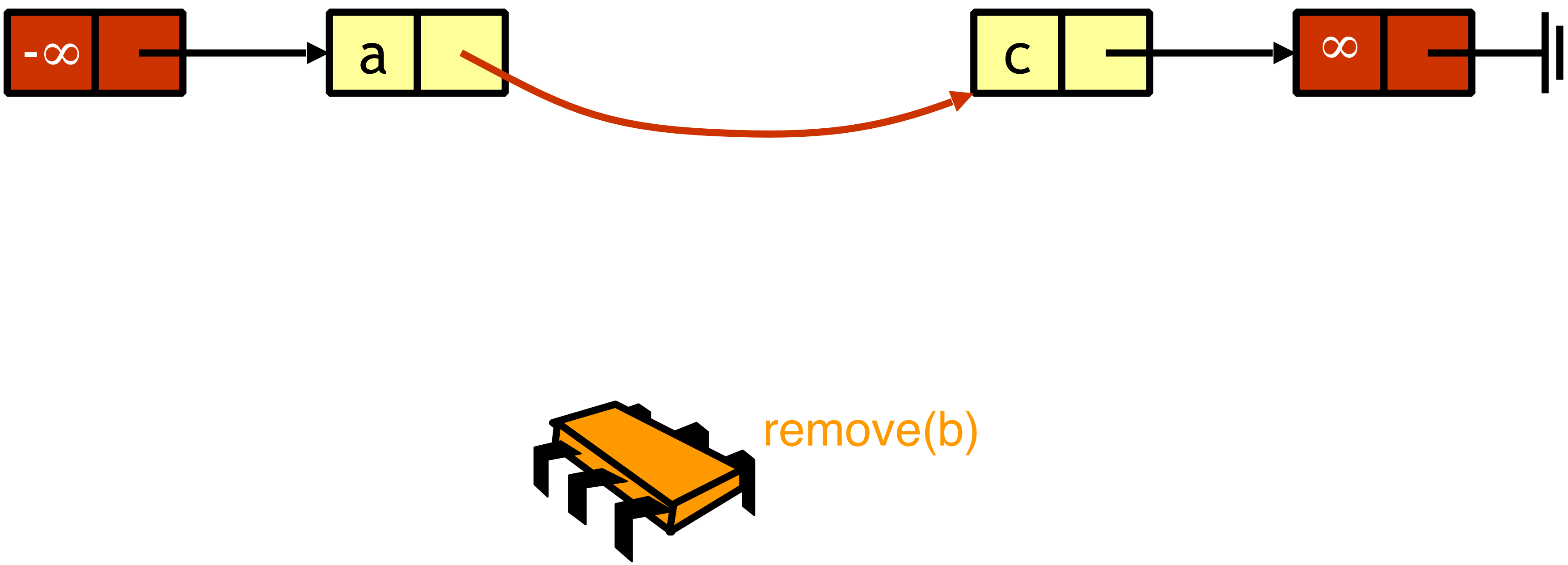
Removing a Node



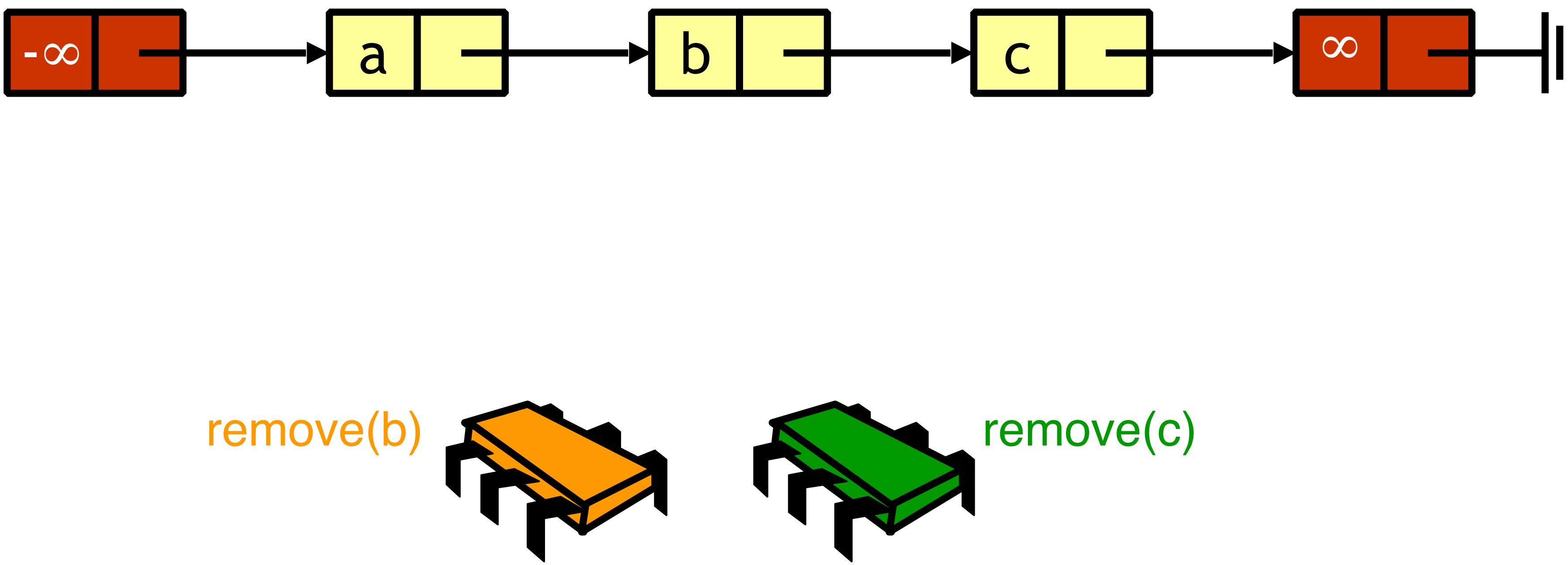
Removing a Node



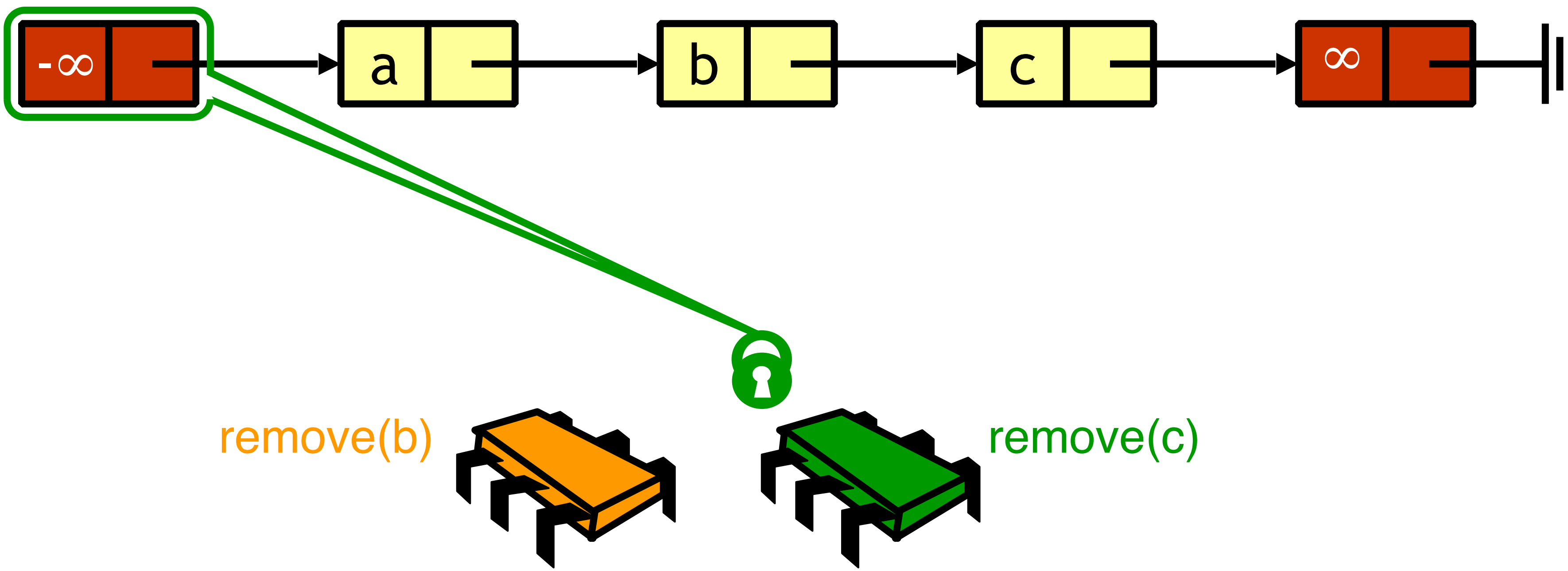
Removing a Node



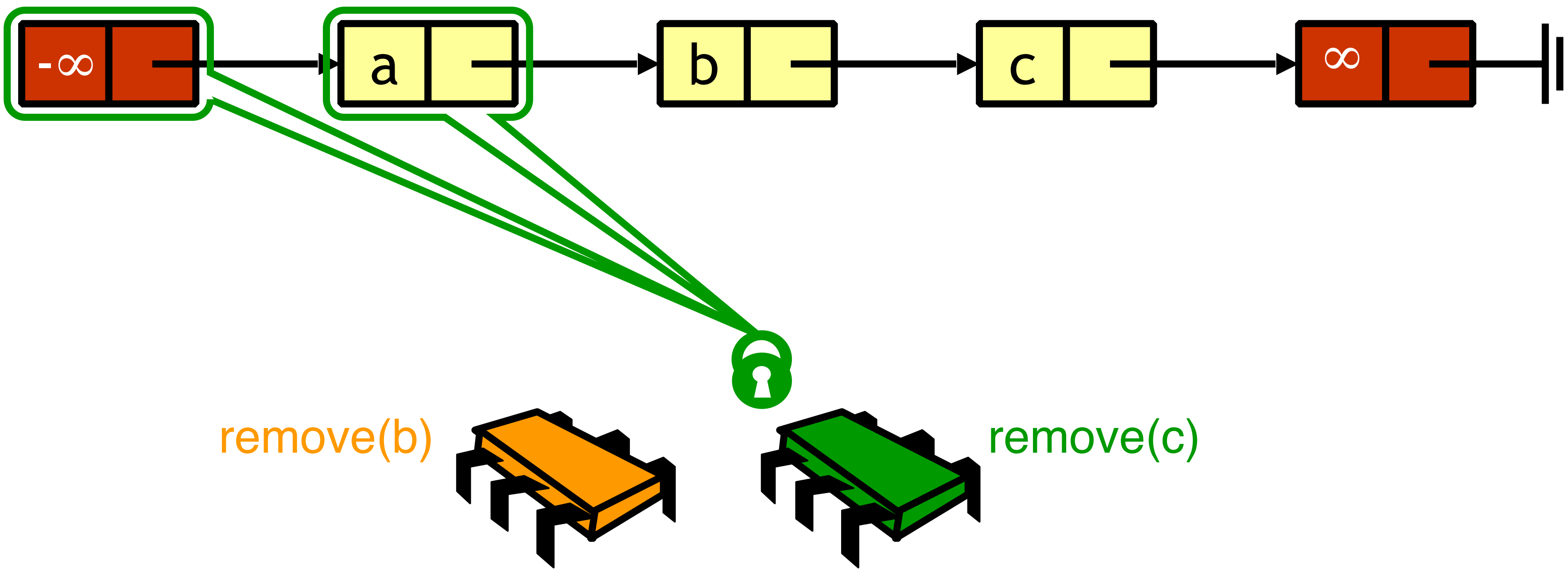
Removing a Node



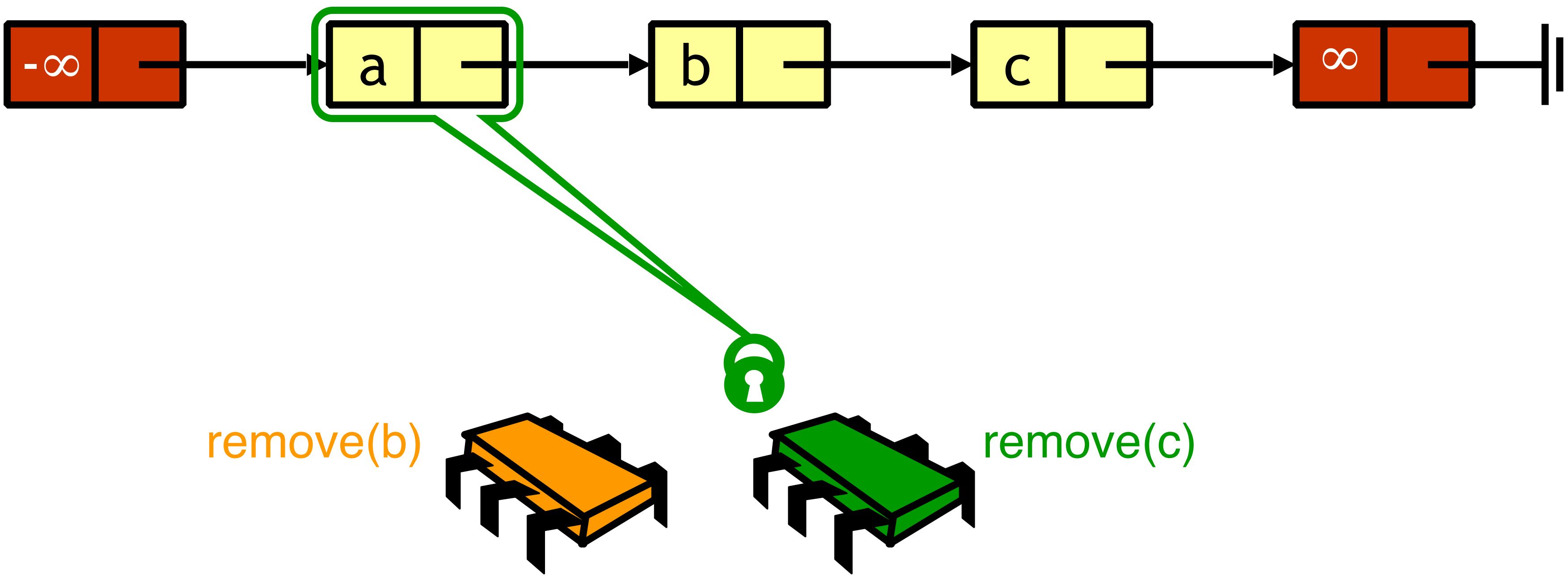
Removing a Node



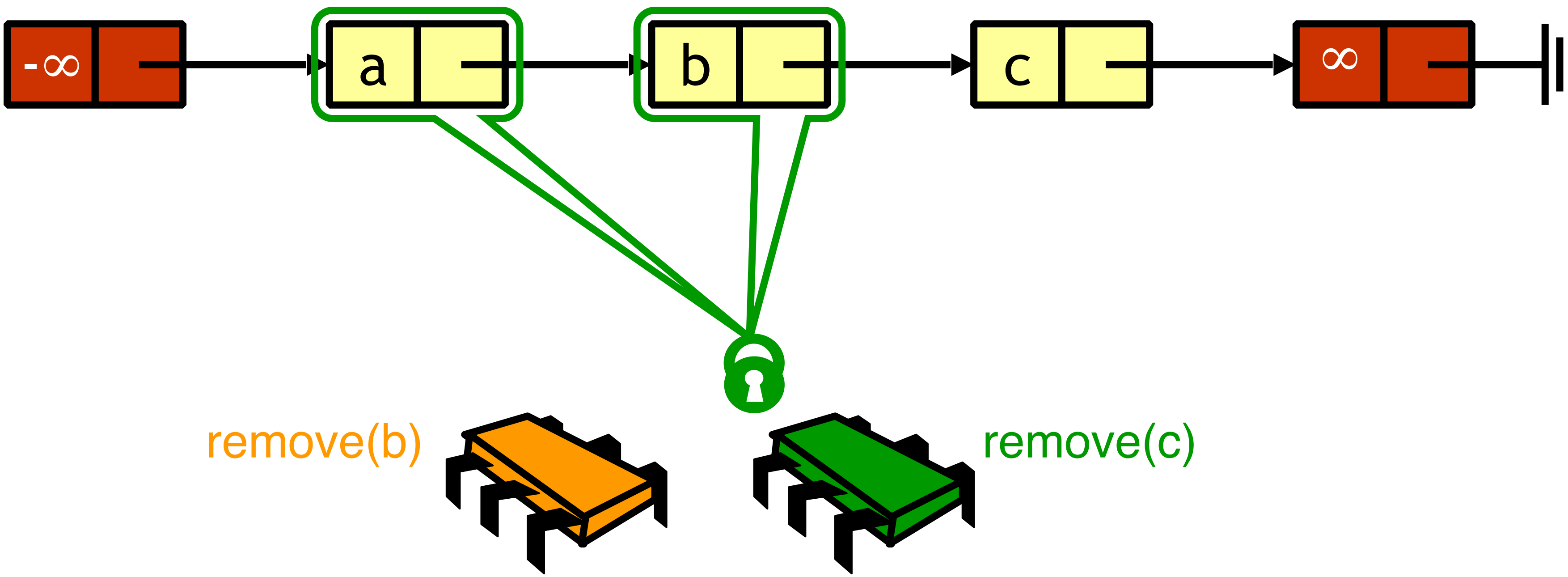
Removing a Node



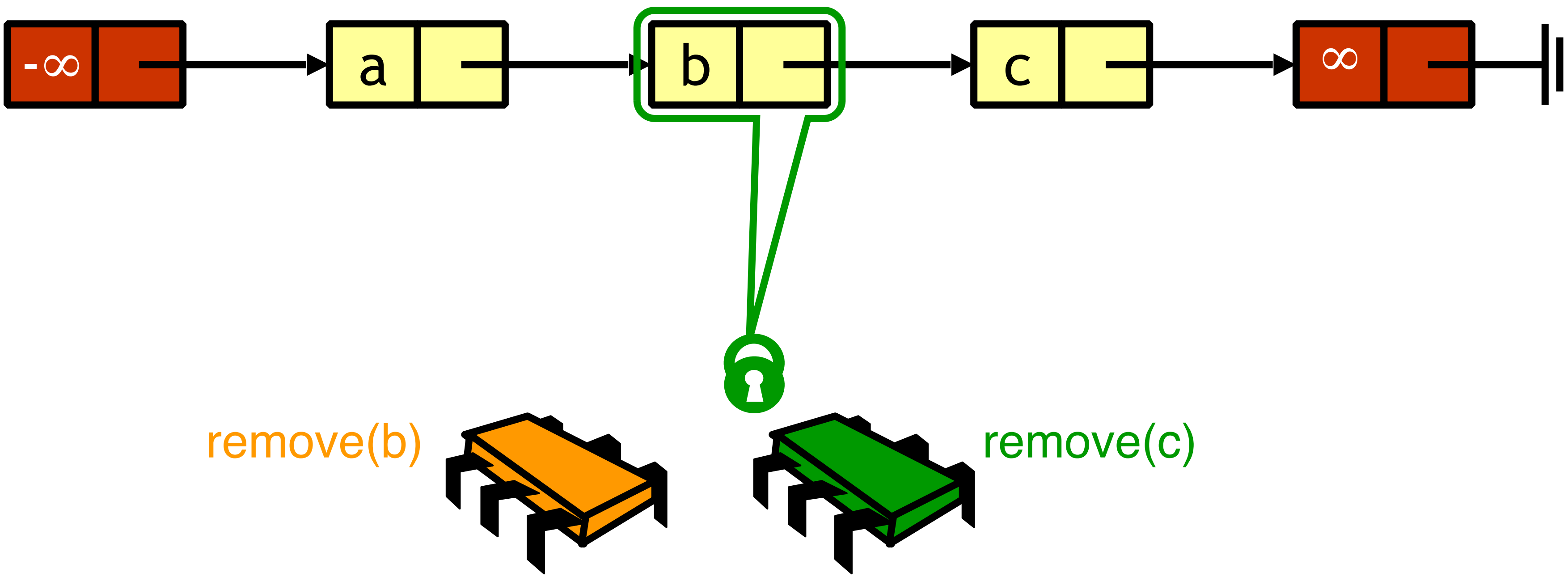
Removing a Node



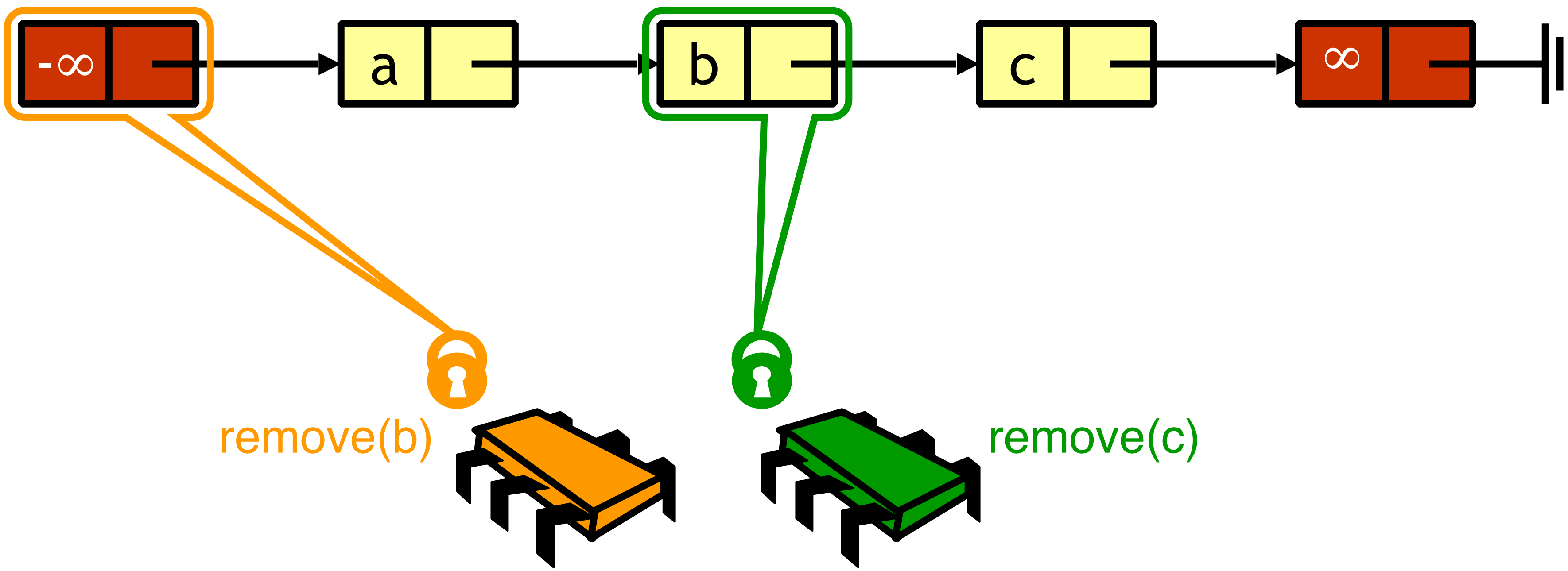
Removing a Node



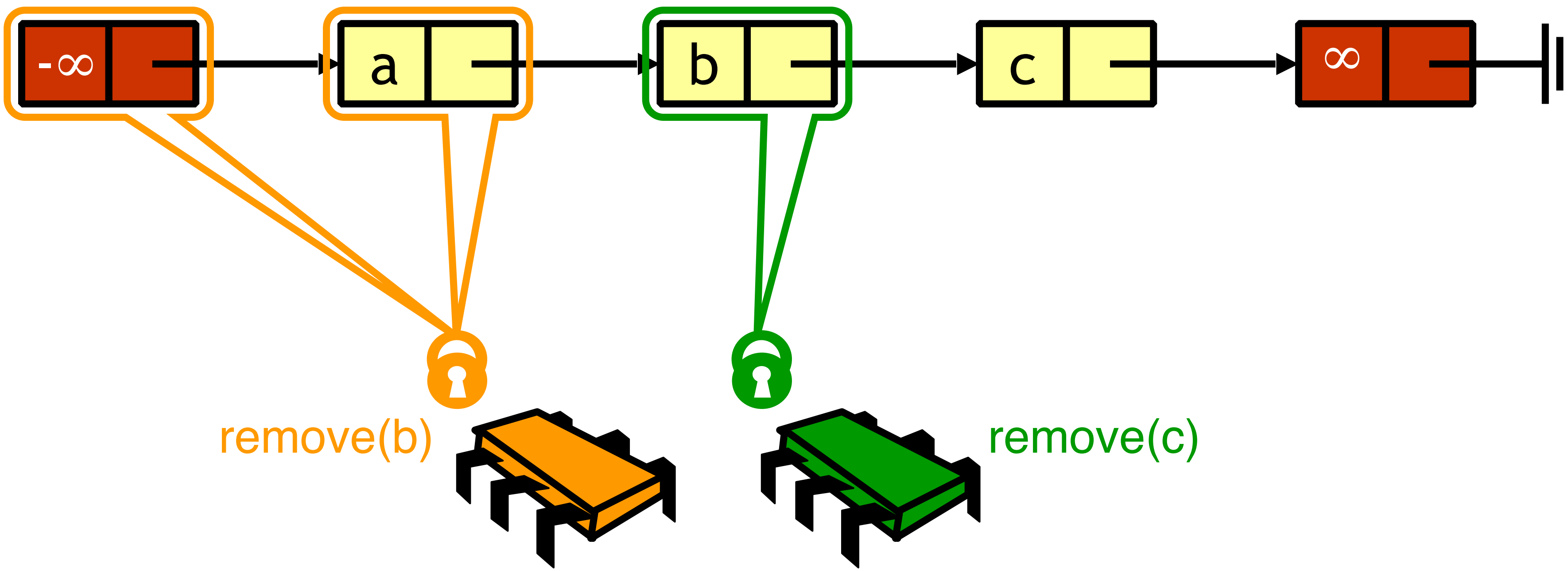
Removing a Node



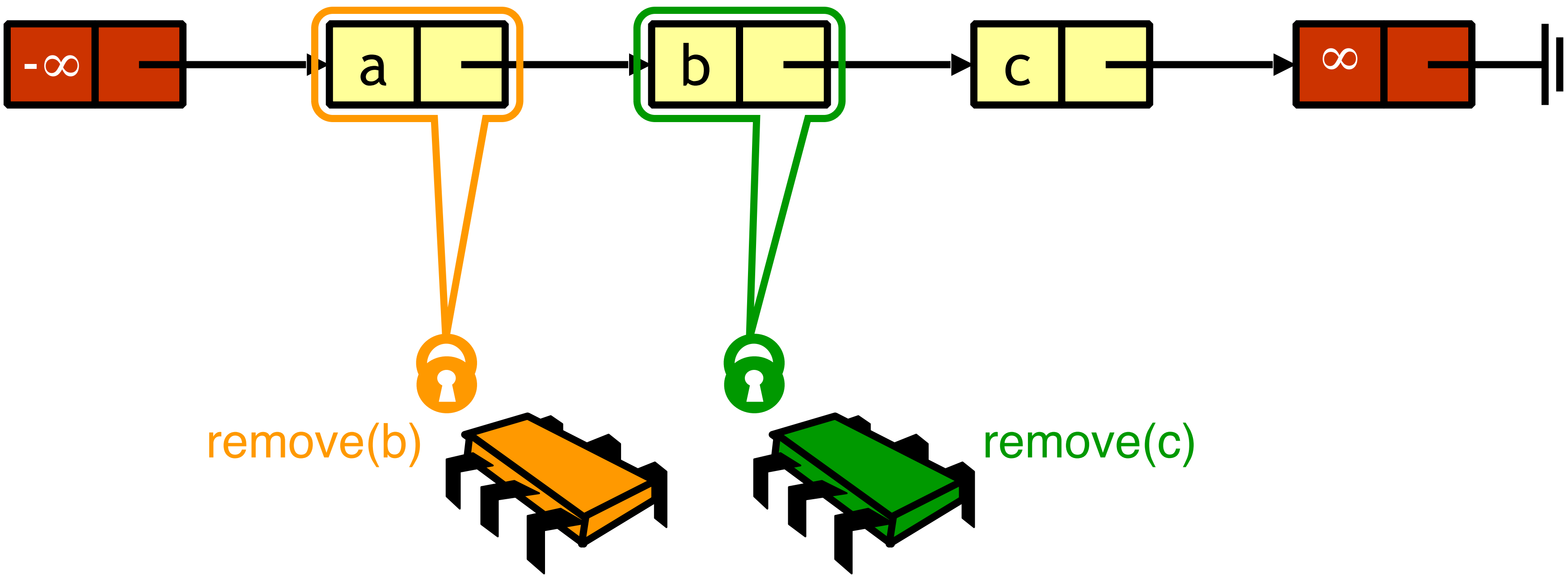
Removing a Node



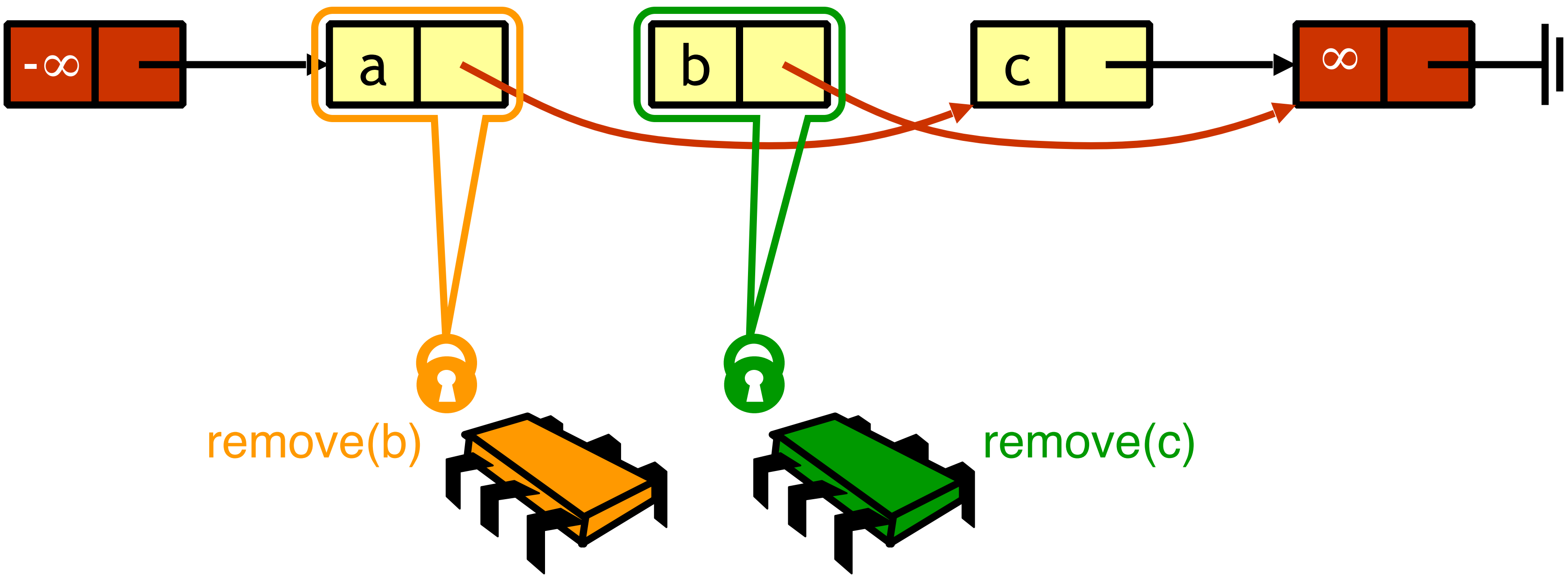
Removing a Node



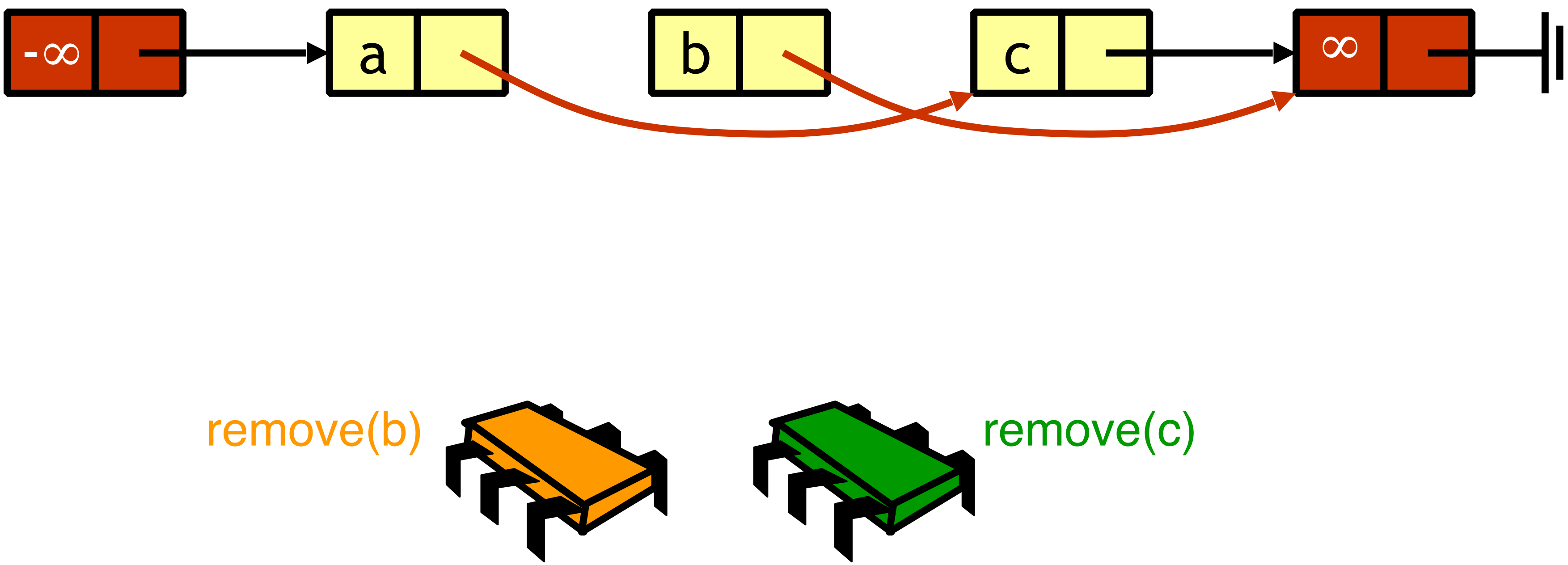
Removing a Node



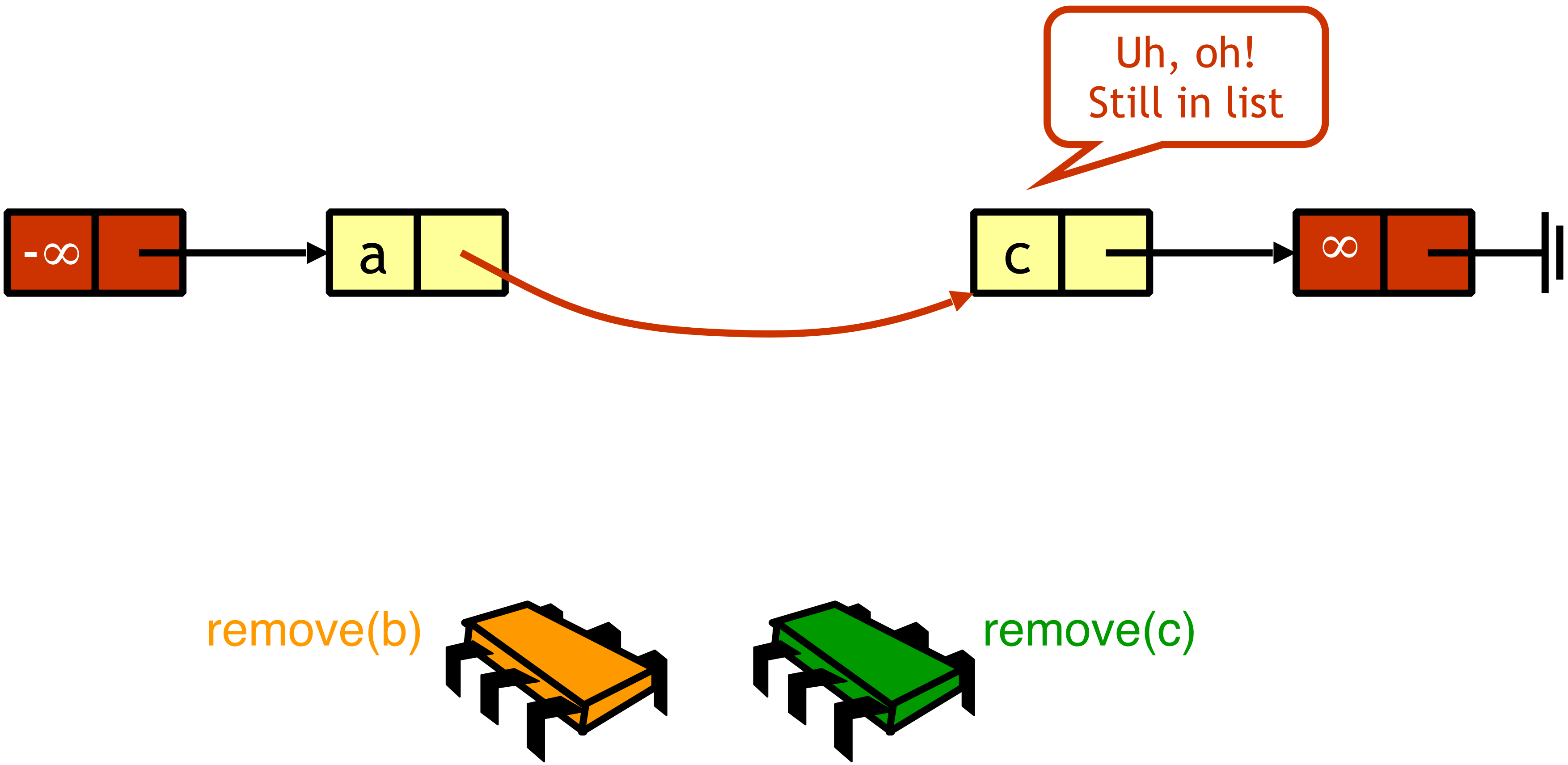
Removing a Node



Removing a Node

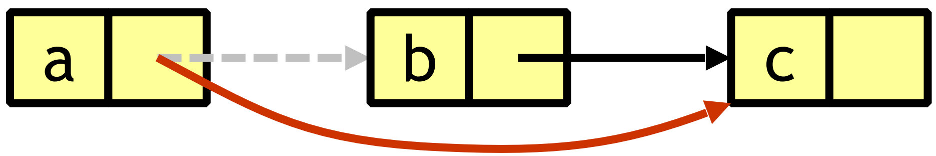


Removing a Node

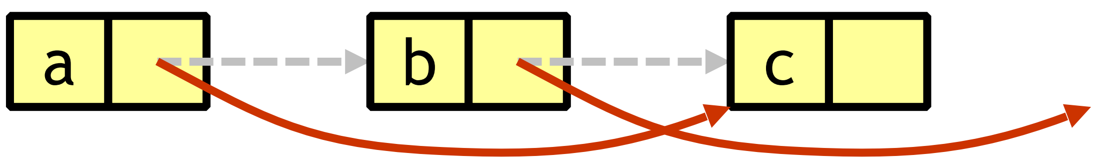


Problem

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



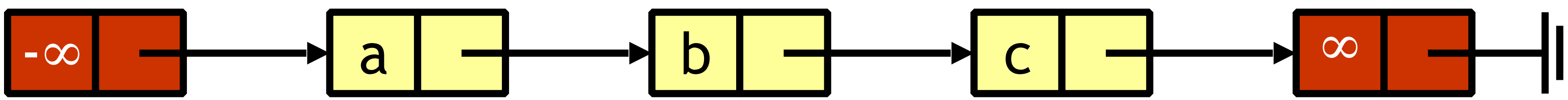
- Problem is
  - Someone could delete c concurrently

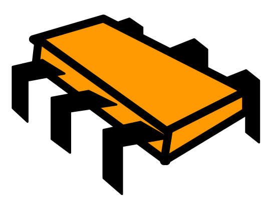


Insight

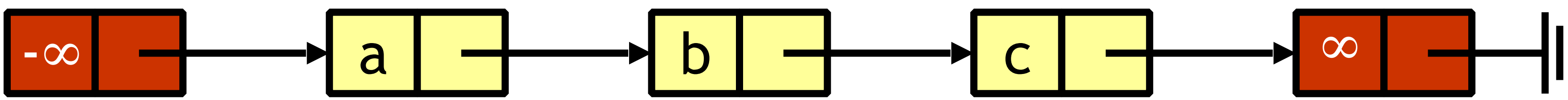
- If a node is locked
  - No one can delete node’s successor
- If a thread locks
  - The node to be deleted
  - And its predecessor
  - Then it works

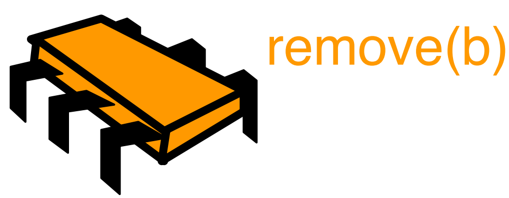
Hand-over-Hand Again



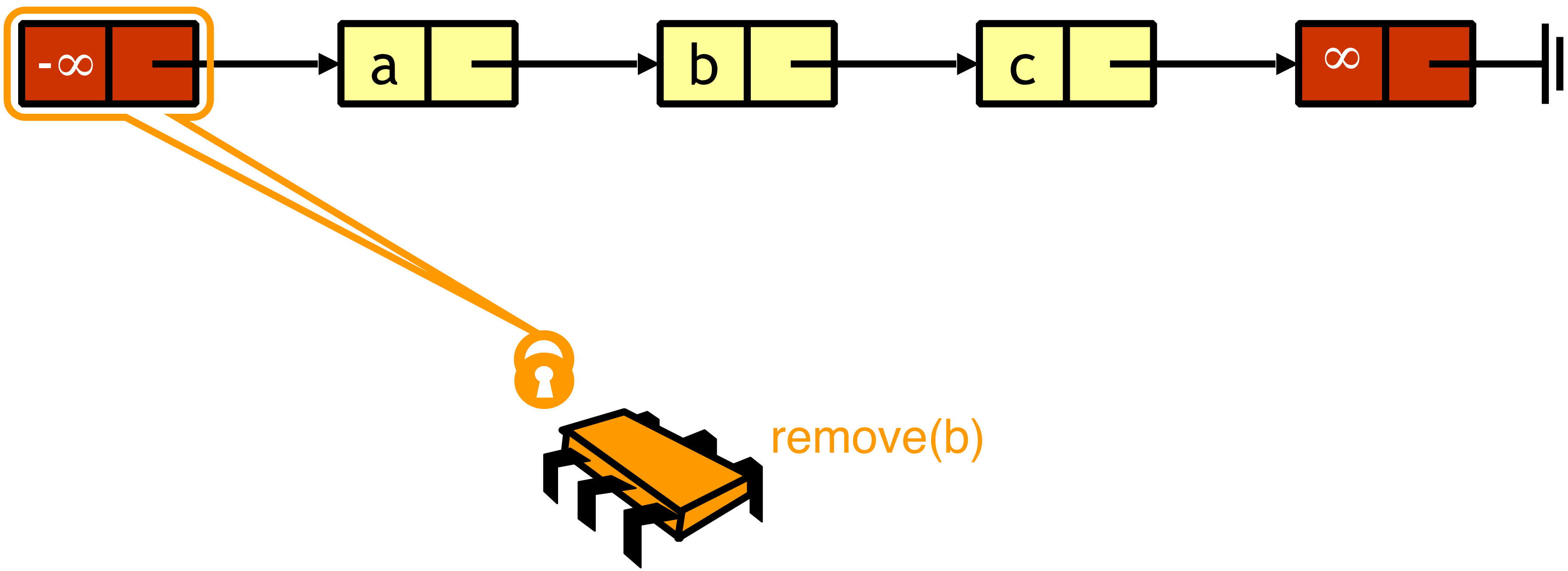


Hand-over-Hand Again

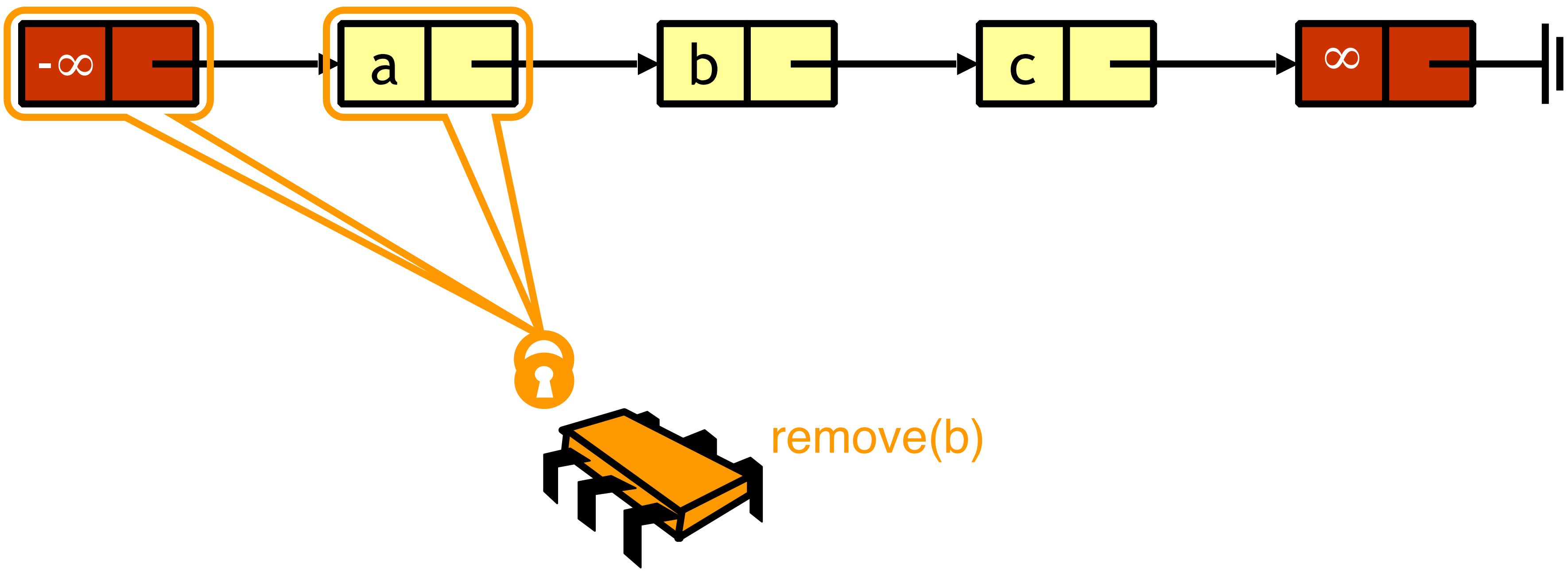




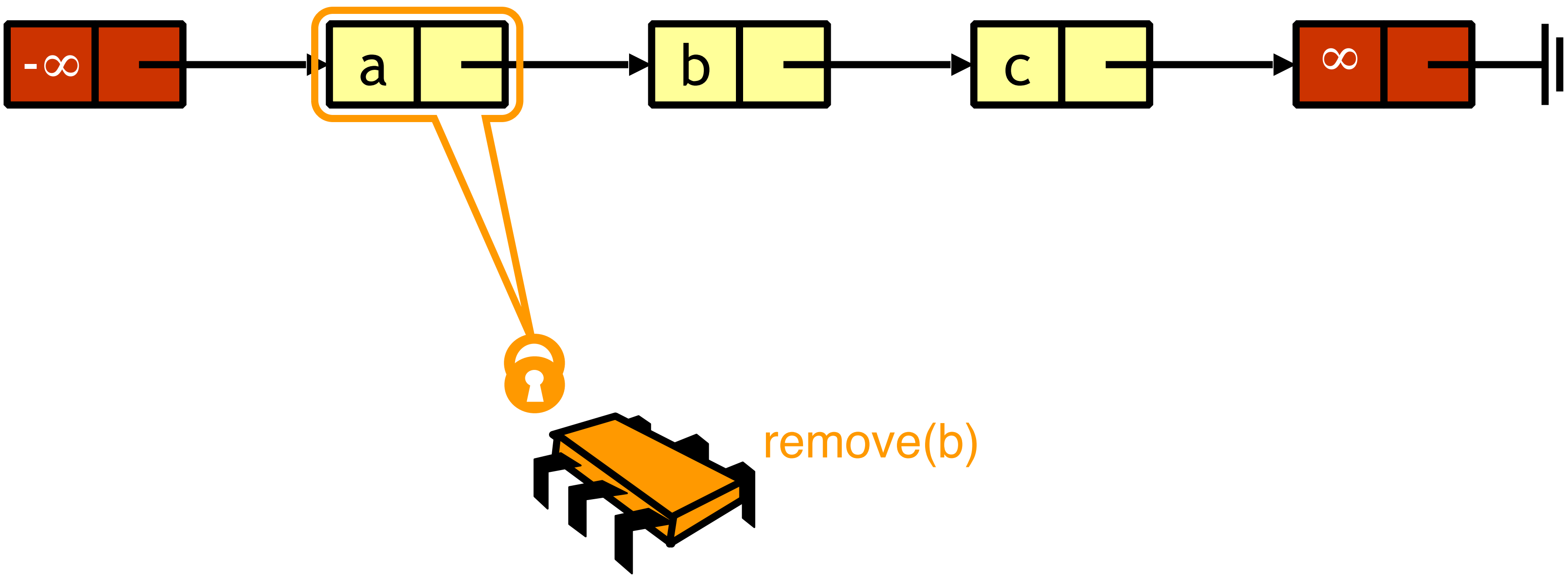
Hand-over-Hand Again



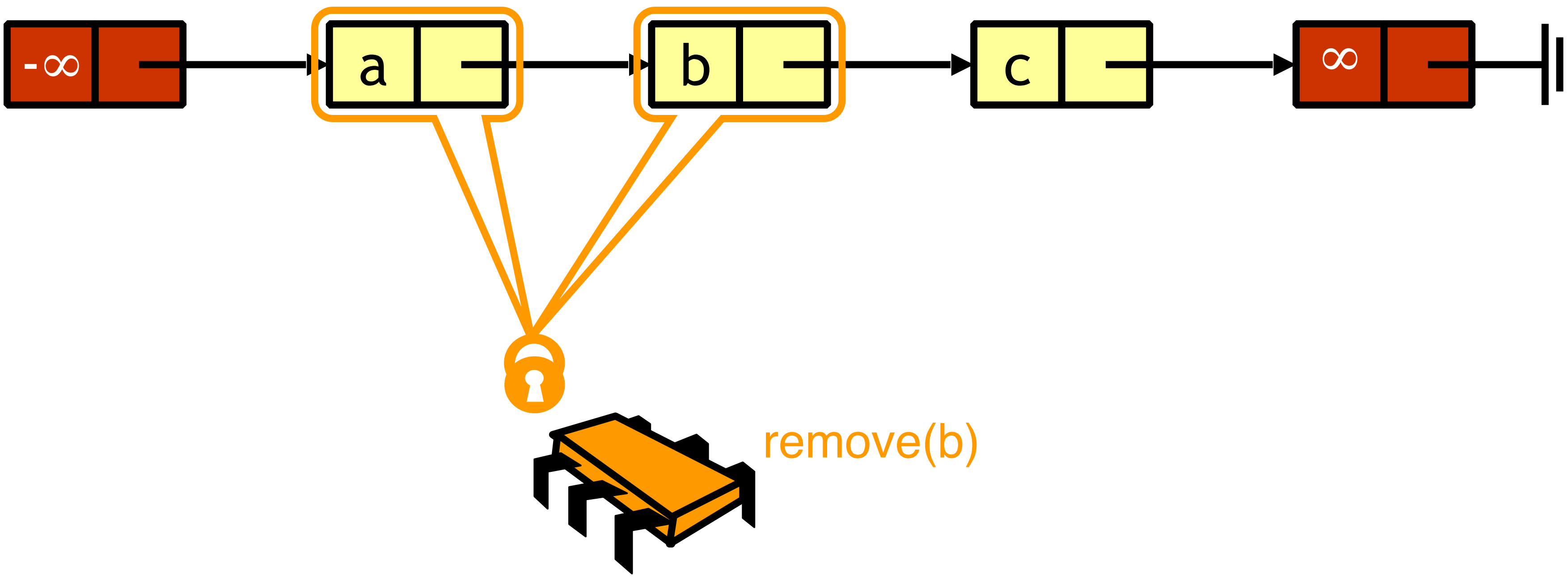
Hand-over-Hand Again



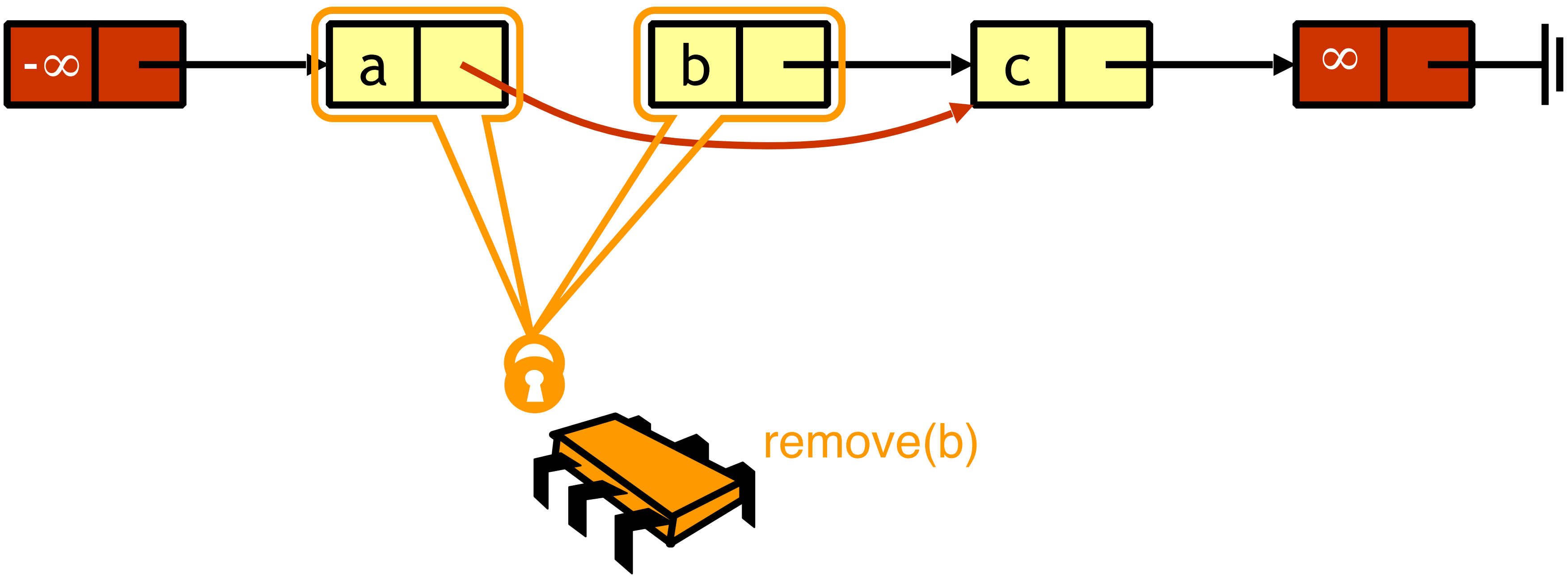
Hand-over-Hand Again



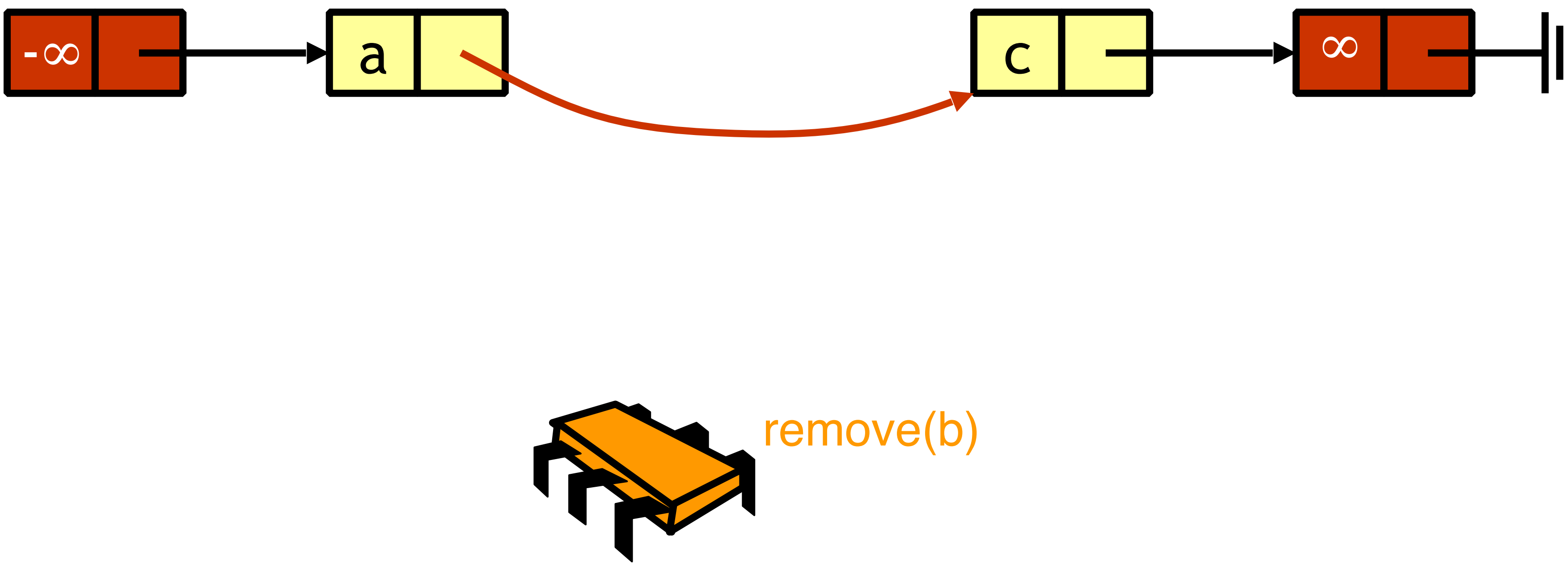
Hand-over-Hand Again



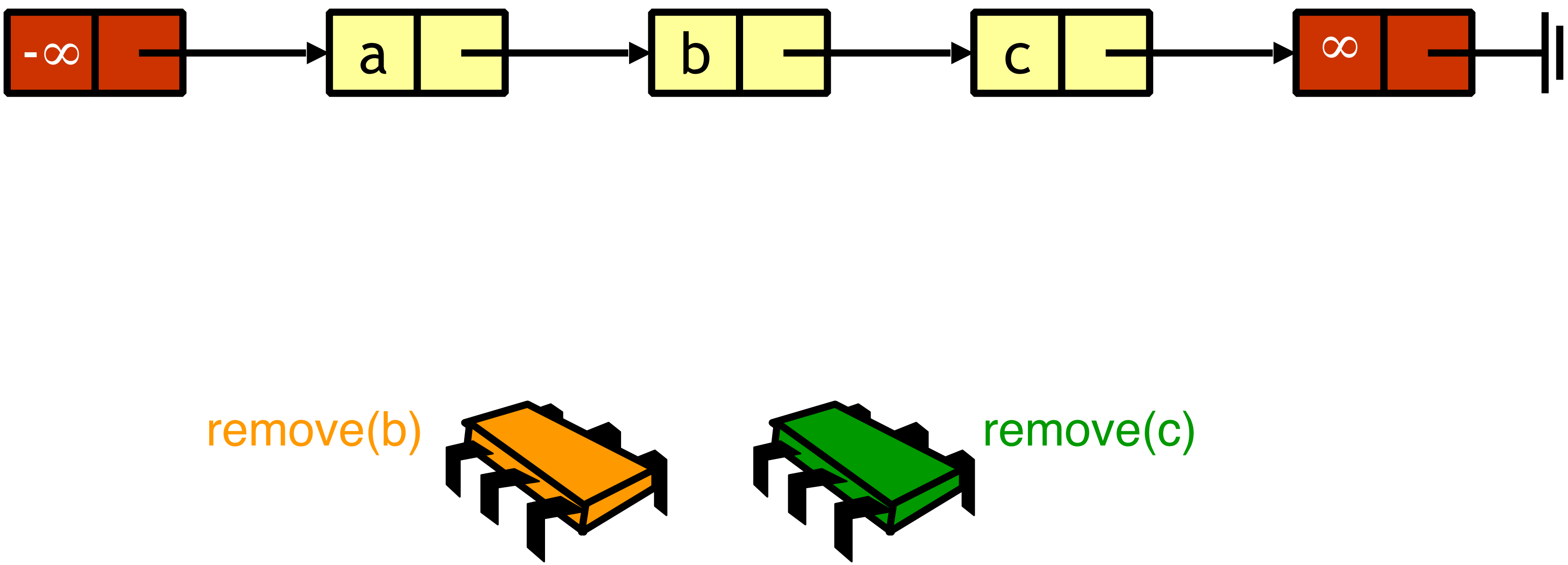
Hand-over-Hand Again



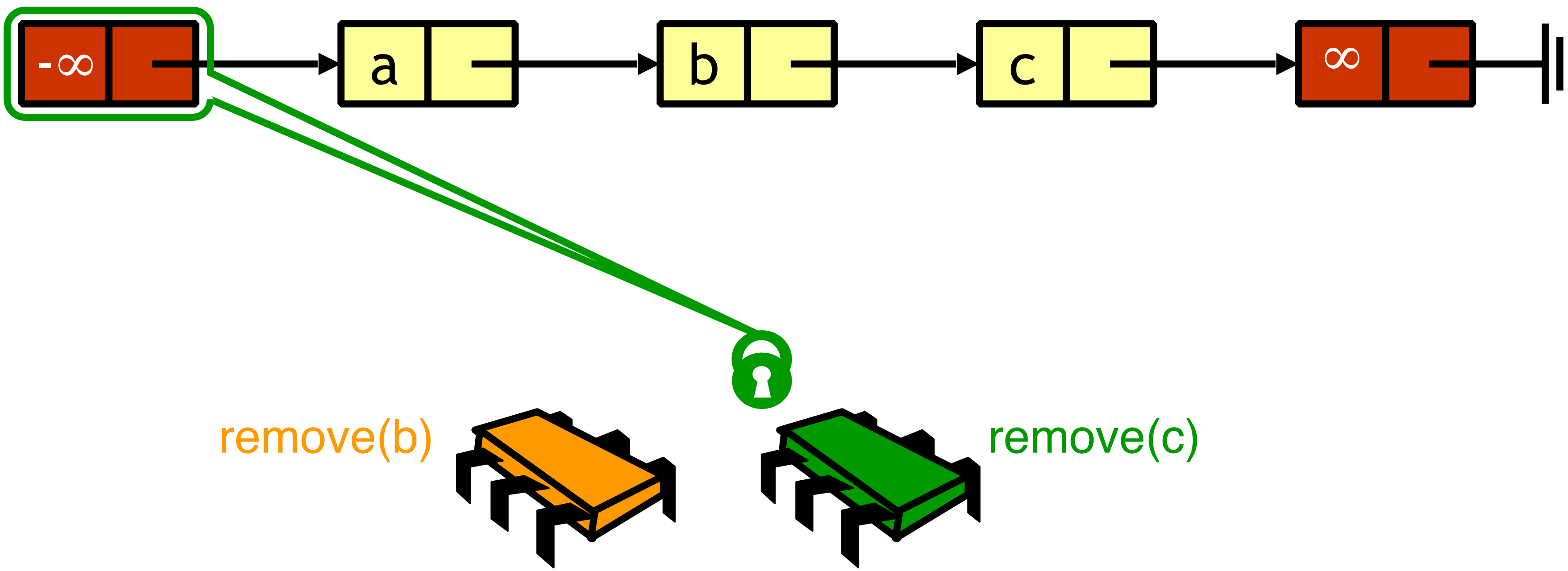
Hand-over-Hand Again



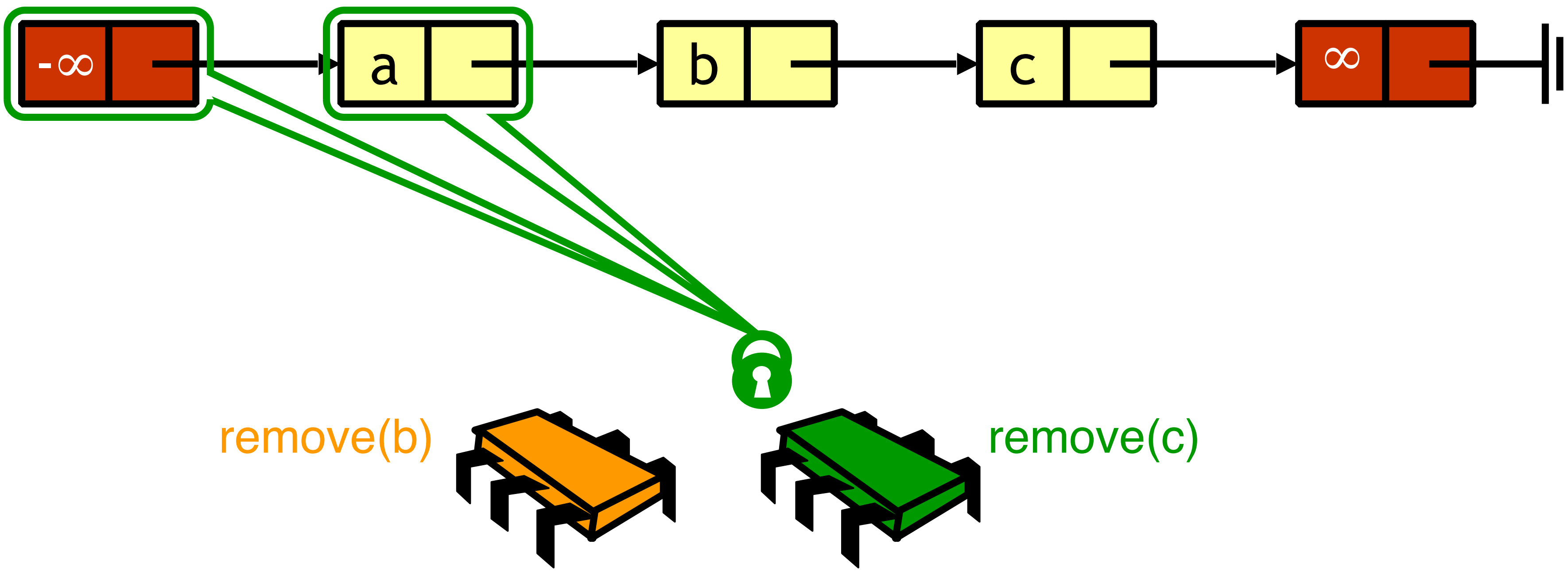
Hand-over-Hand Again



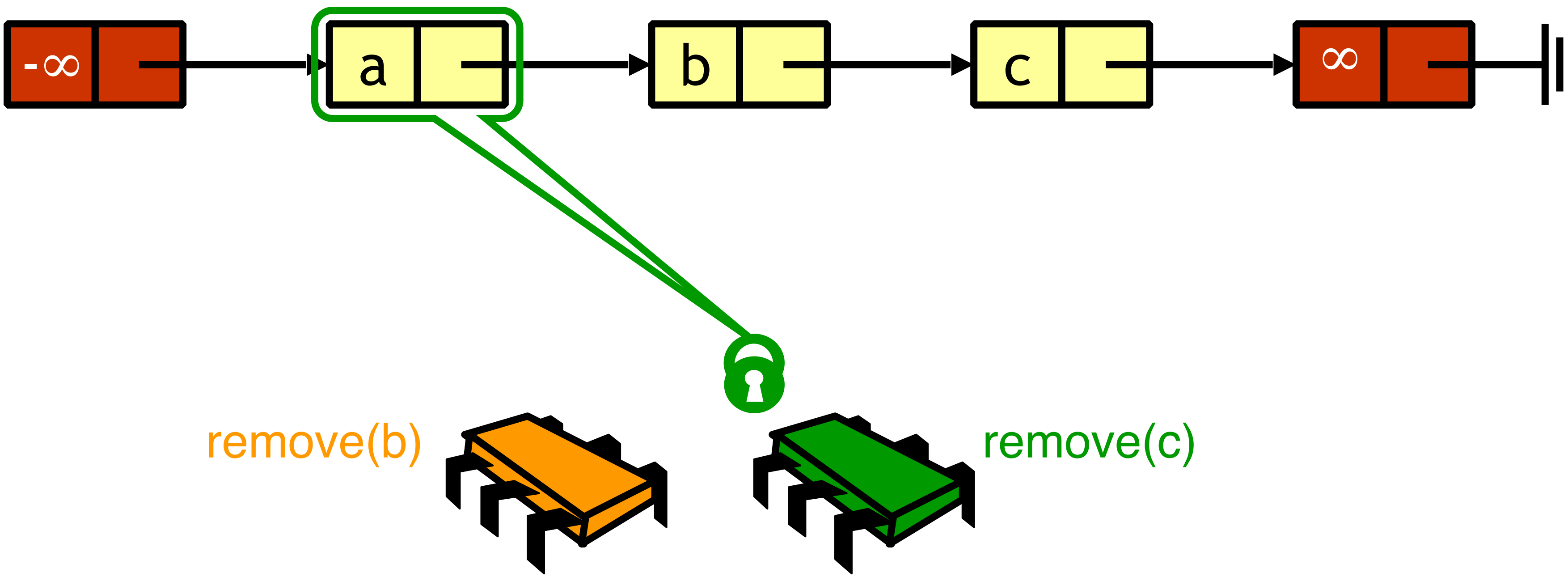
Hand-over-Hand Again



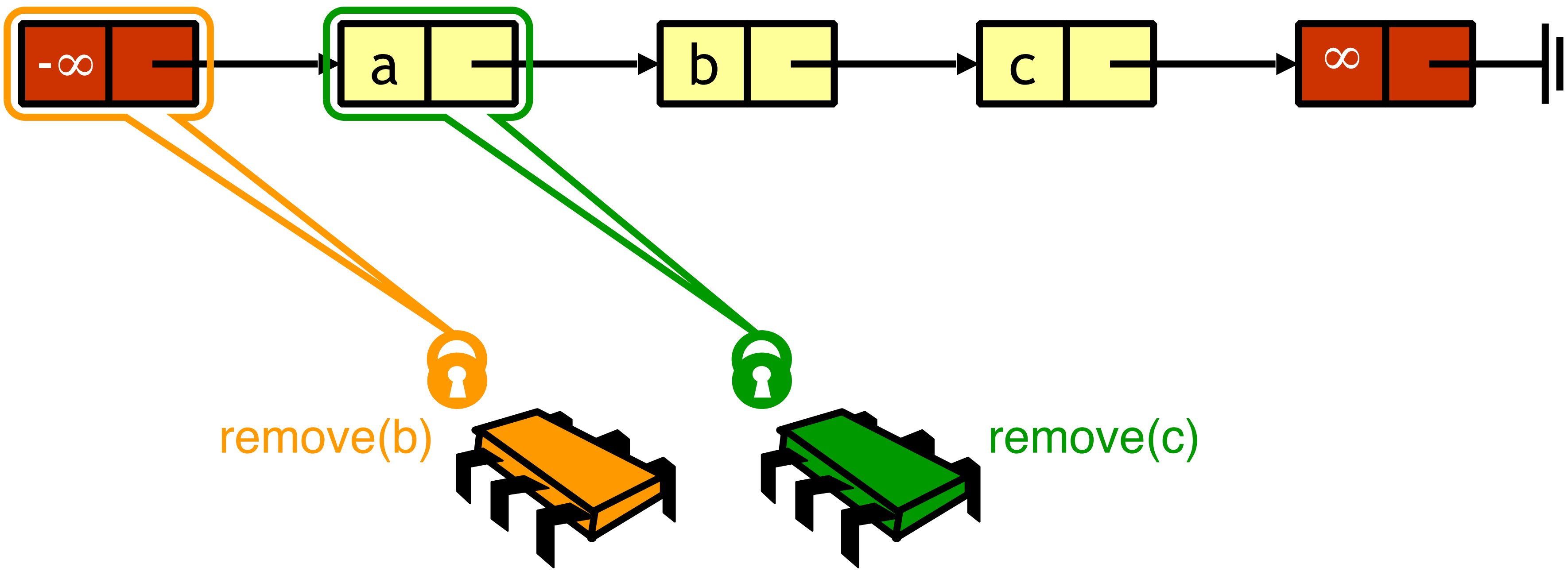
Hand-over-Hand Again



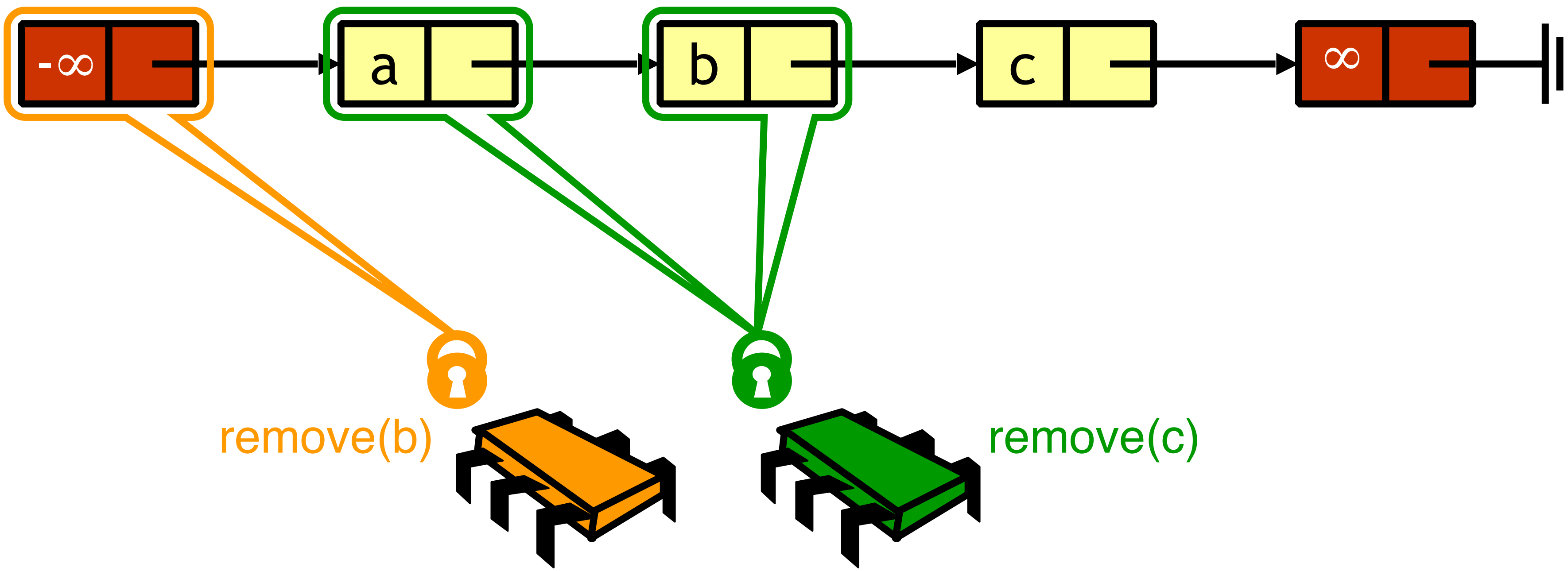
Hand-over-Hand Again



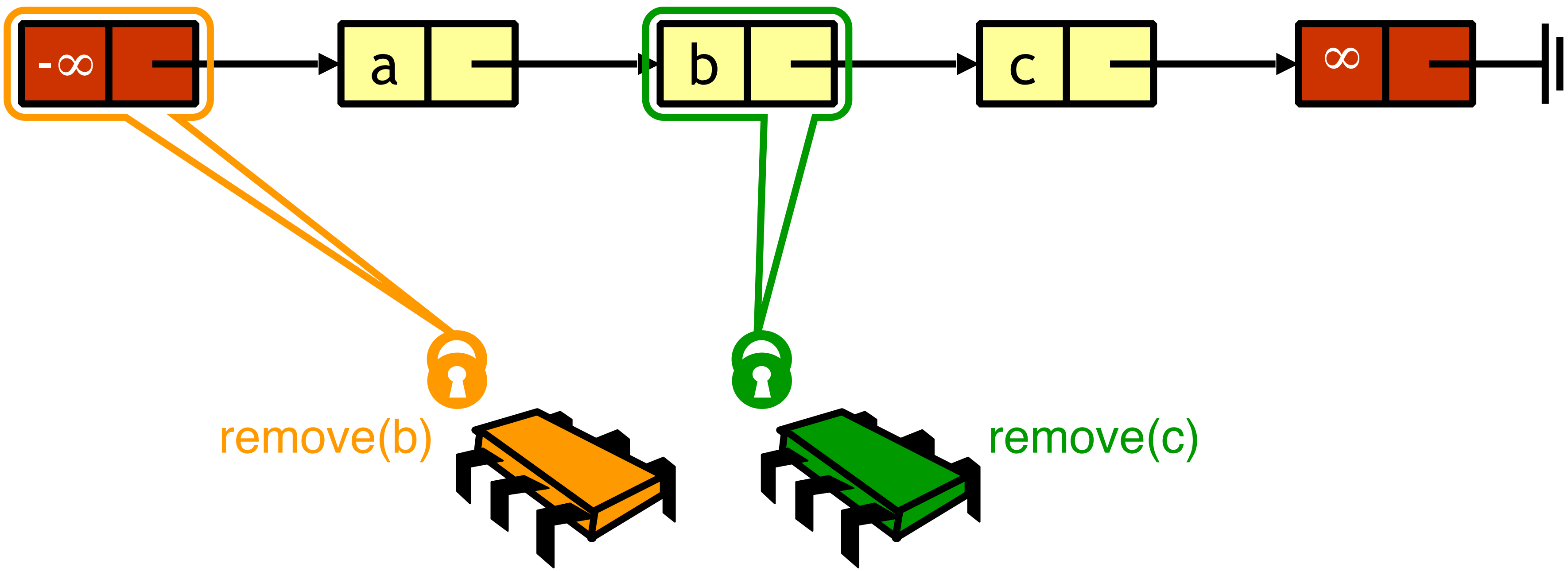
Hand-over-Hand Again



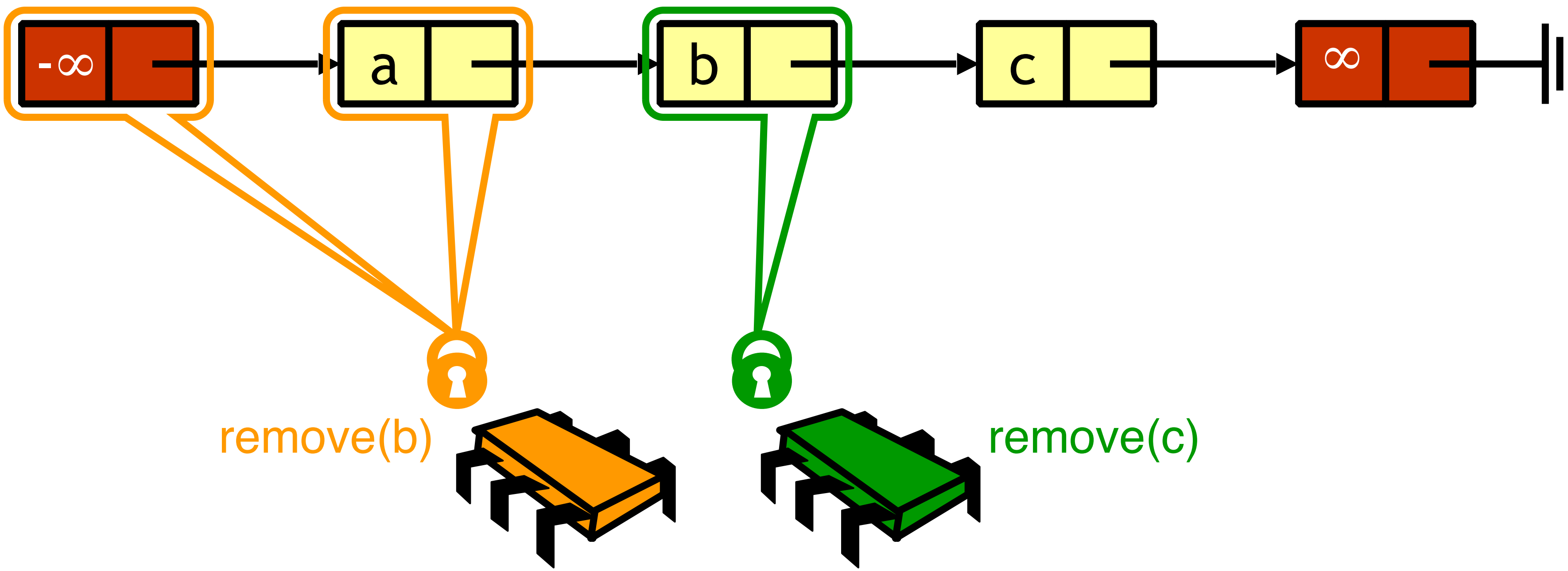
Hand-over-Hand Again



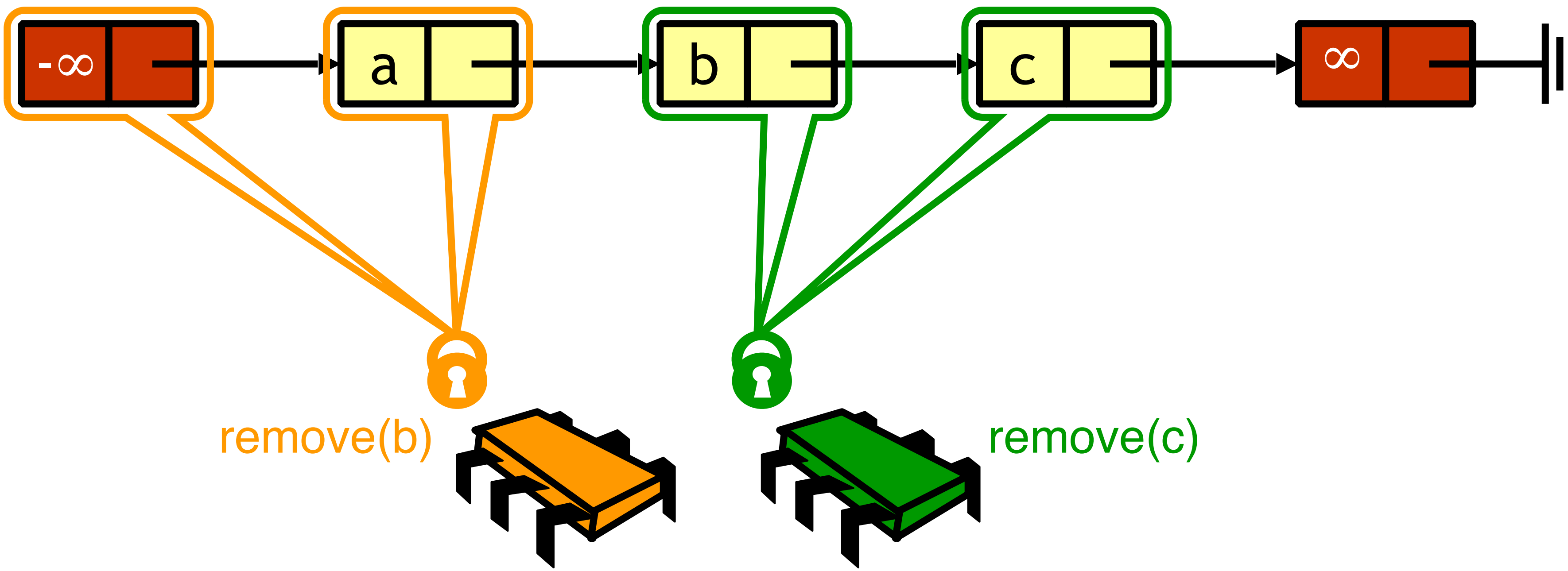
Hand-over-Hand Again



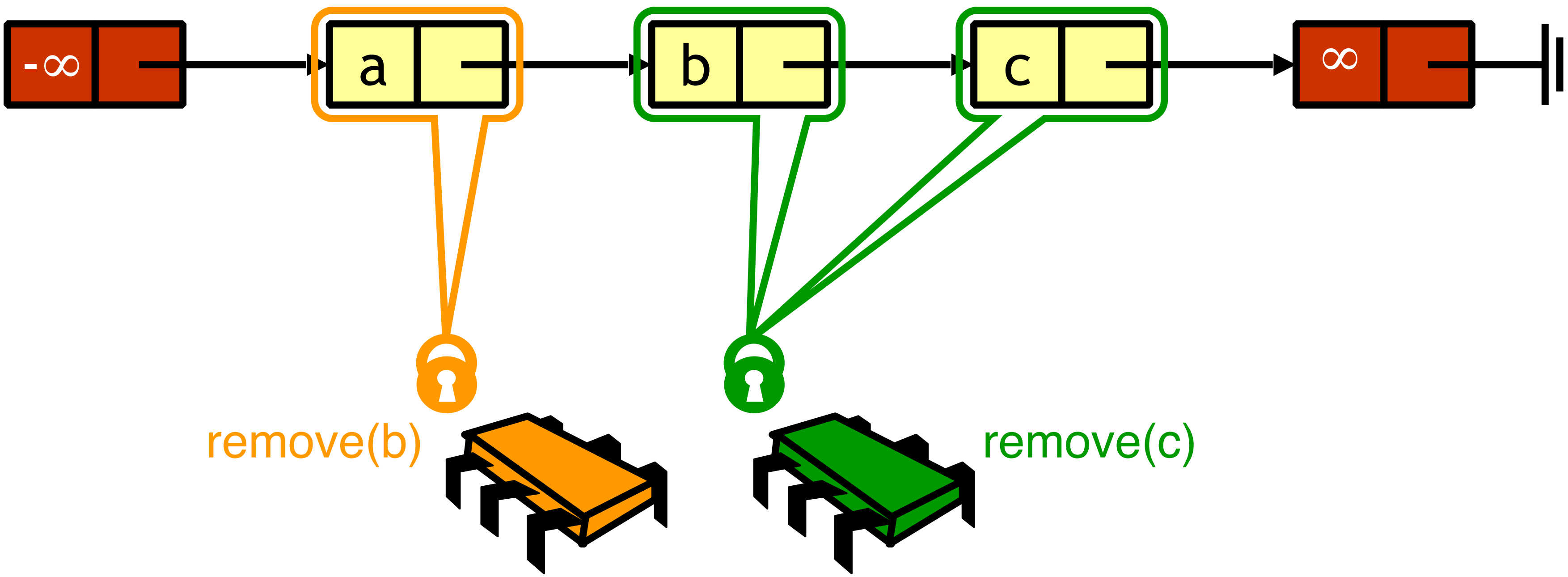
Hand-over-Hand Again



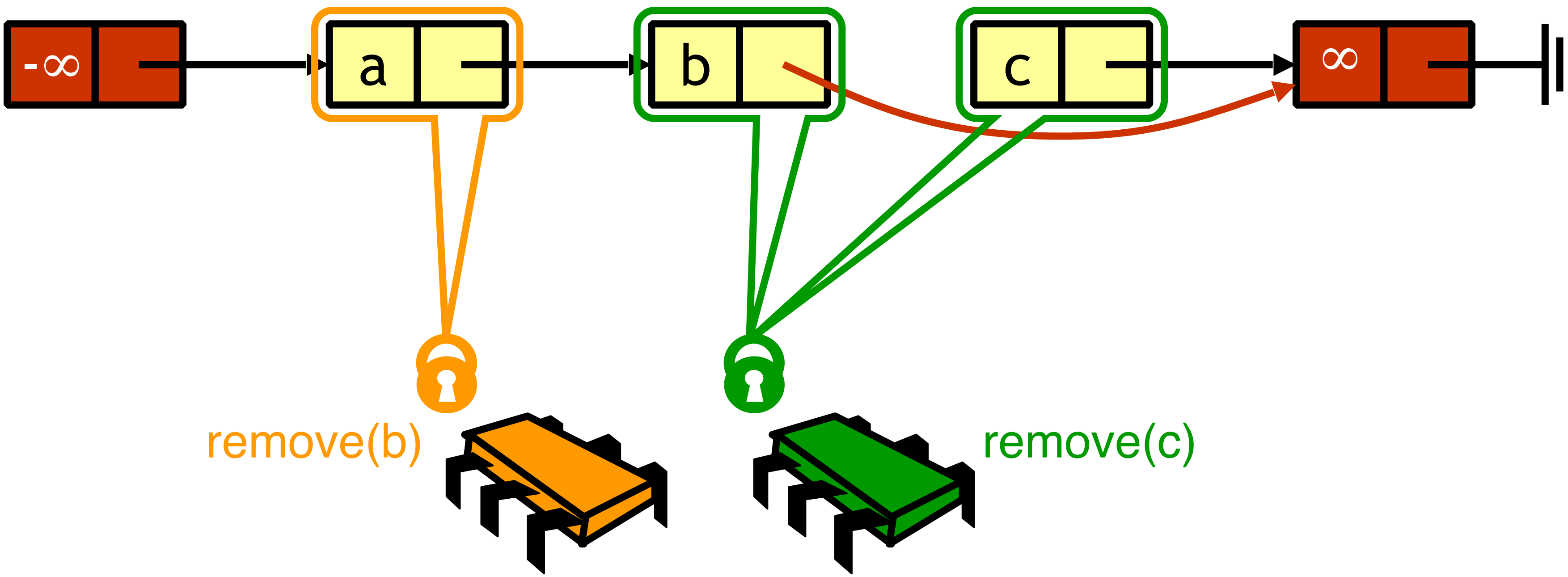
Hand-over-Hand Again



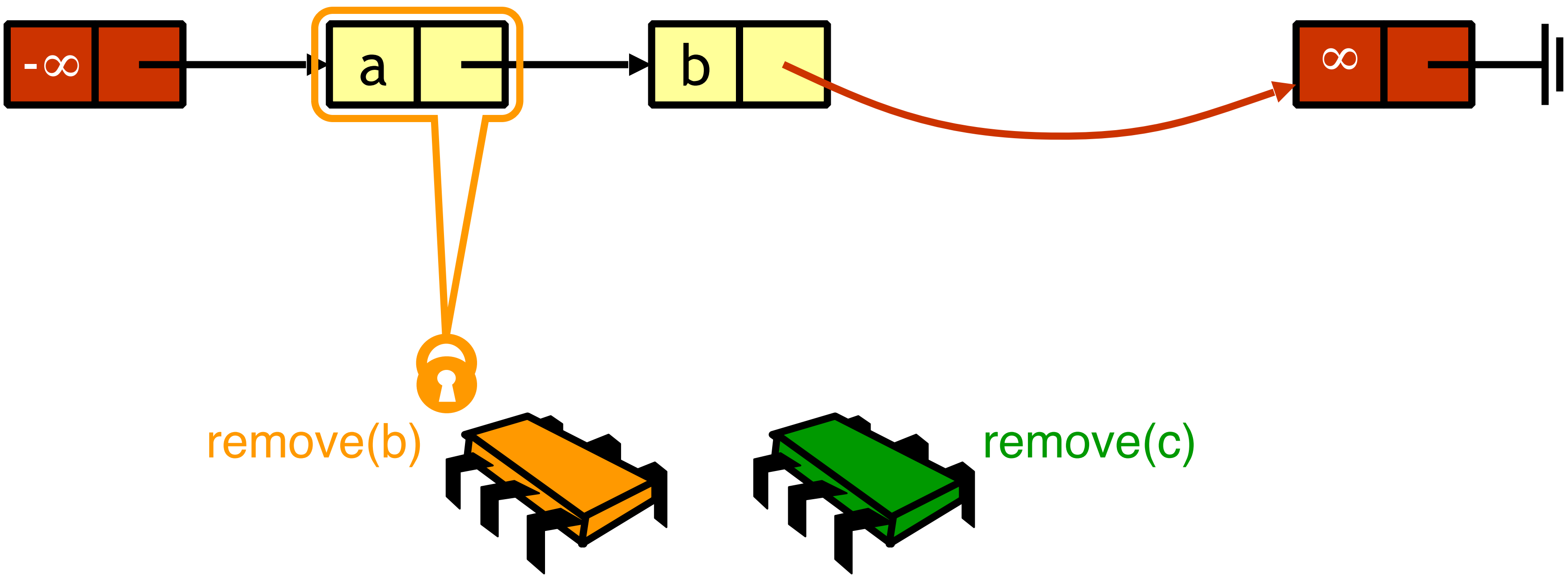
Hand-over-Hand Again



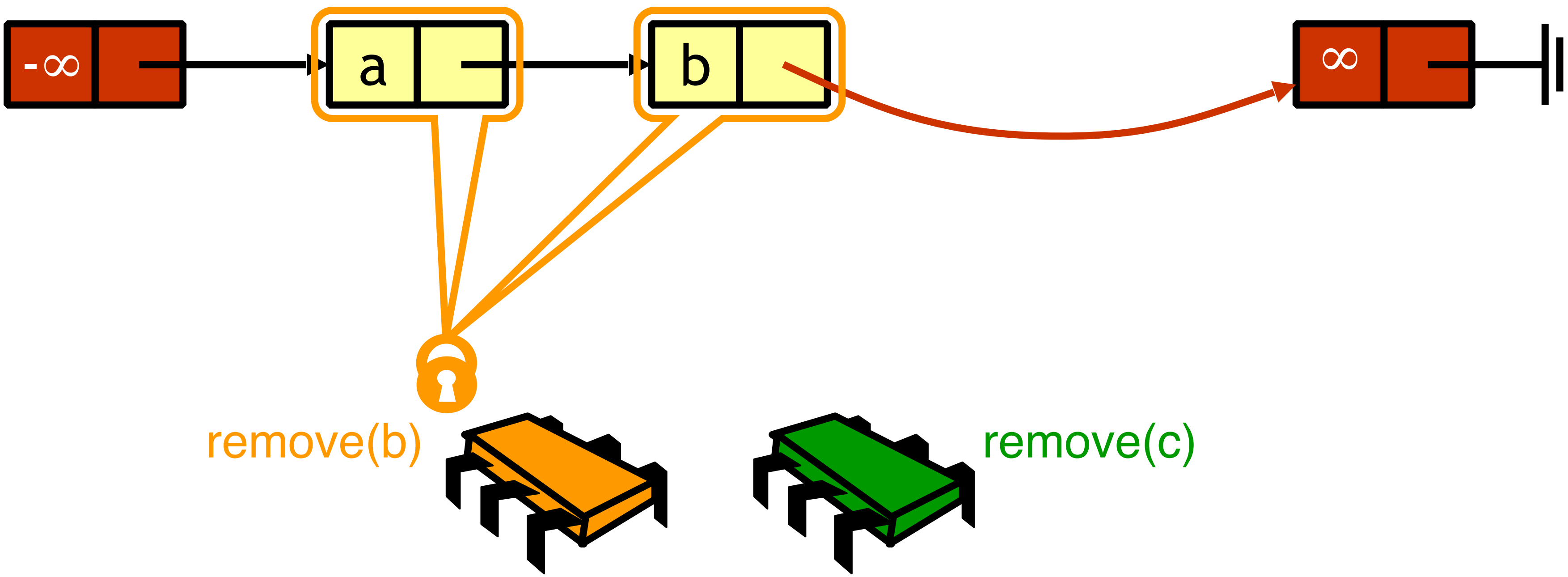
Hand-over-Hand Again



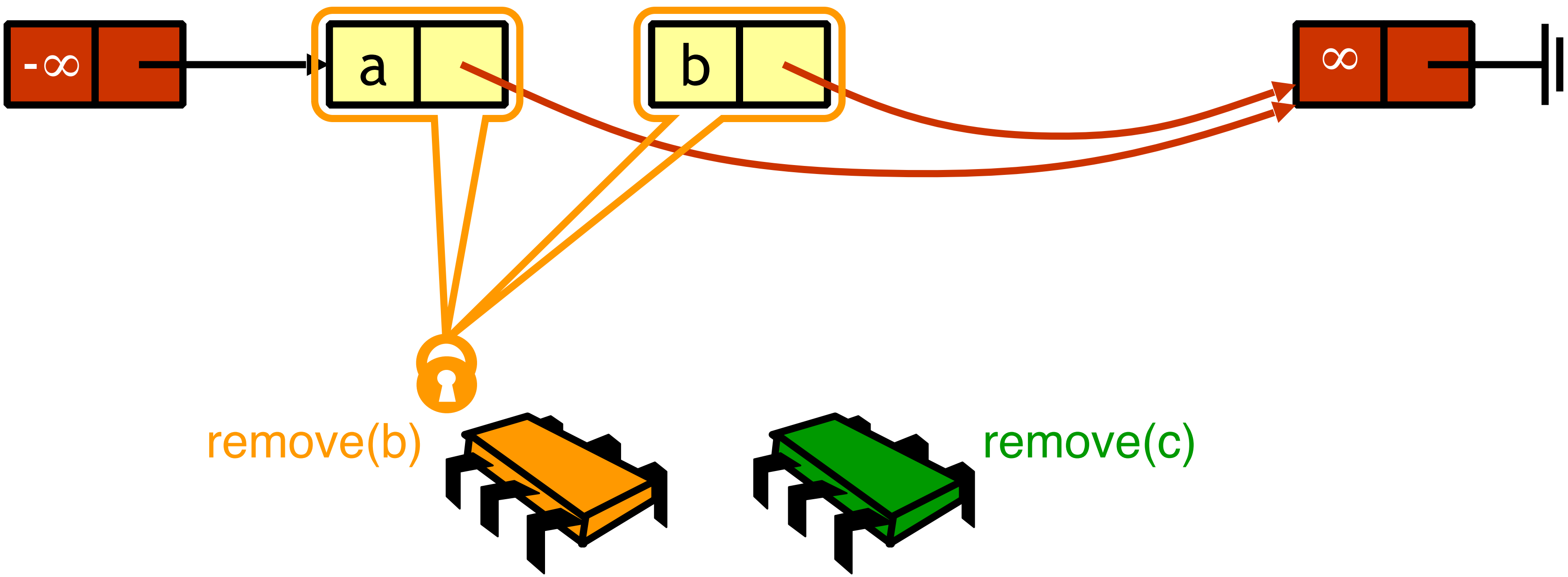
Hand-over-Hand Again



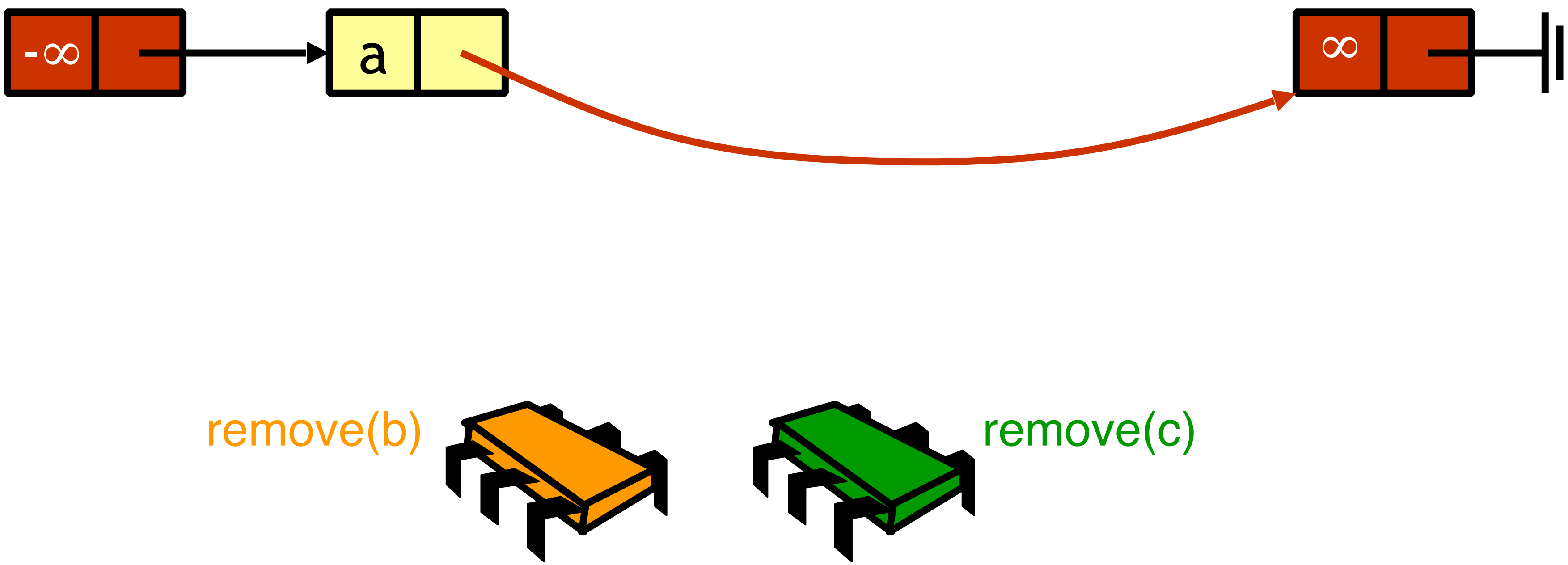
Hand-over-Hand Again



Hand-over-Hand Again



Hand-over-Hand Again



Remove

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

Remove

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

Key used to order node

Remove

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

Predecessor and current nodes

Remove

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

Make sure locks released

Remove

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

Everything else

Remove

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

Remove

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

Lock previous

Remove

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

Lock current

Remove

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

Traverse the list

Remove: Searching

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

Remove: Searching

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

Search key range (curr and pred locked)

Remove: Searching

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

If node found, remove it

Remove: Searching

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

Unlock predecessor and demote current (only one node locked!)

Remove: Searching

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

Find and lock new current

Remove: Searching

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

Otherwise not present

Adding Nodes

- To add node b
  - Lock predecessor
  - Lock successor
- Neither can be deleted

Drawbacks

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

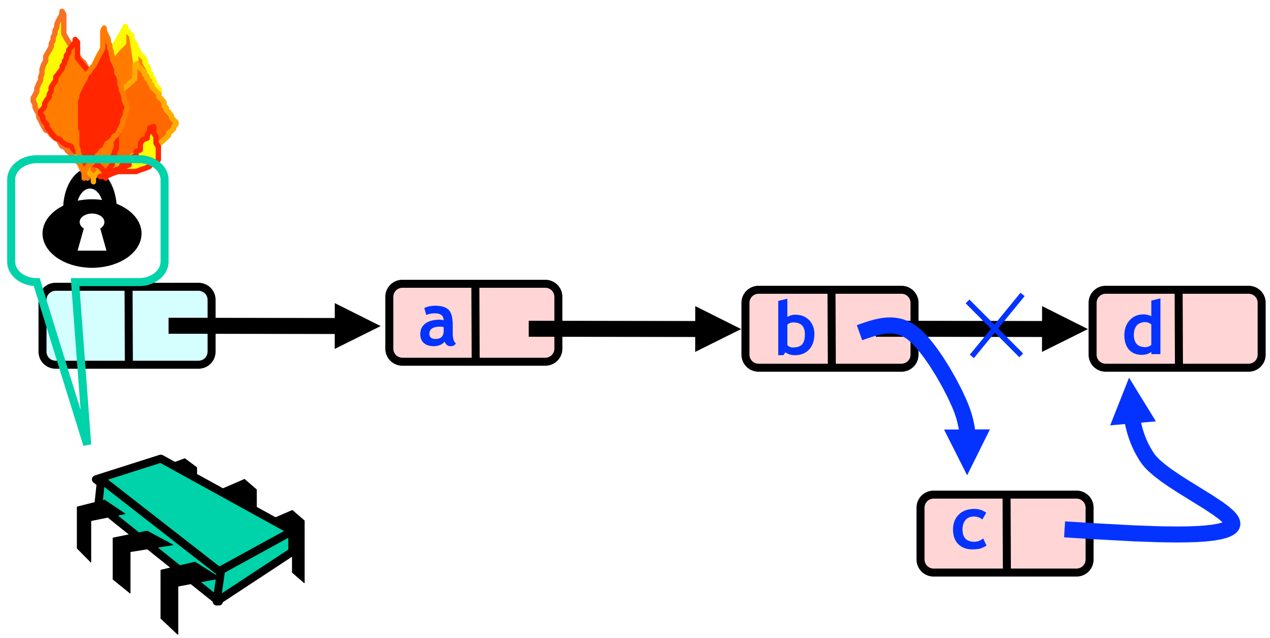
Linked List Lecture

- Five approaches to concurrent data structure design:
  - Coarse-grained locking
  - Fine-grained locking
  - Optimistic synchronization
  - Lazy synchronization
  - Lock-free synchronization

List-based Set

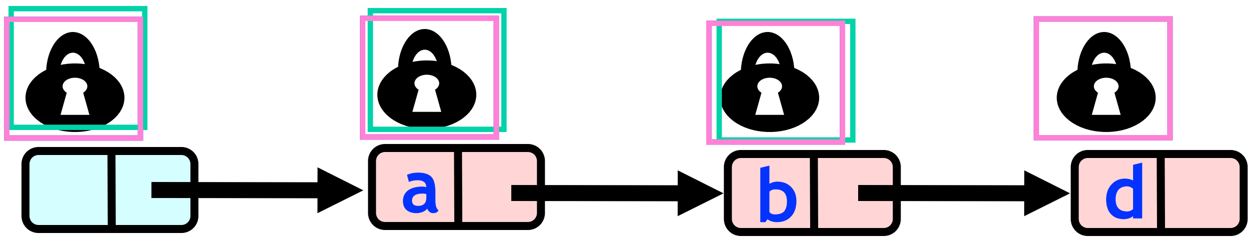
- We used an ordered list to implement a Set:
  - An unordered collection of objects
  - No duplicates
  - Methods:
    - add() a new object
    - remove() an object
    - Test if set contains() object

Course Grained Locking



Simple but hotspot + bottleneck

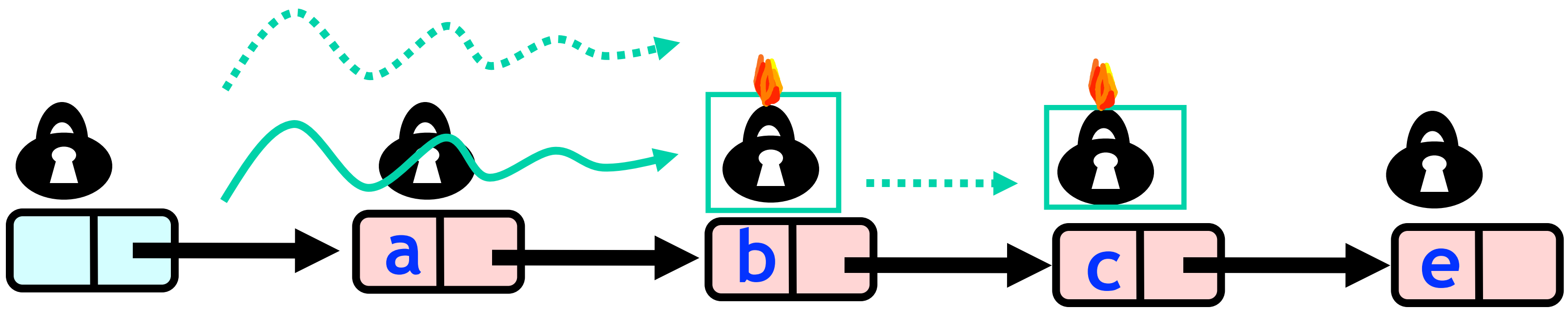
Fine Grained Locking



- Allows concurrency but everyone always delayed by front guy = bottleneck

- Lock acquisition overhead

Optimistic List

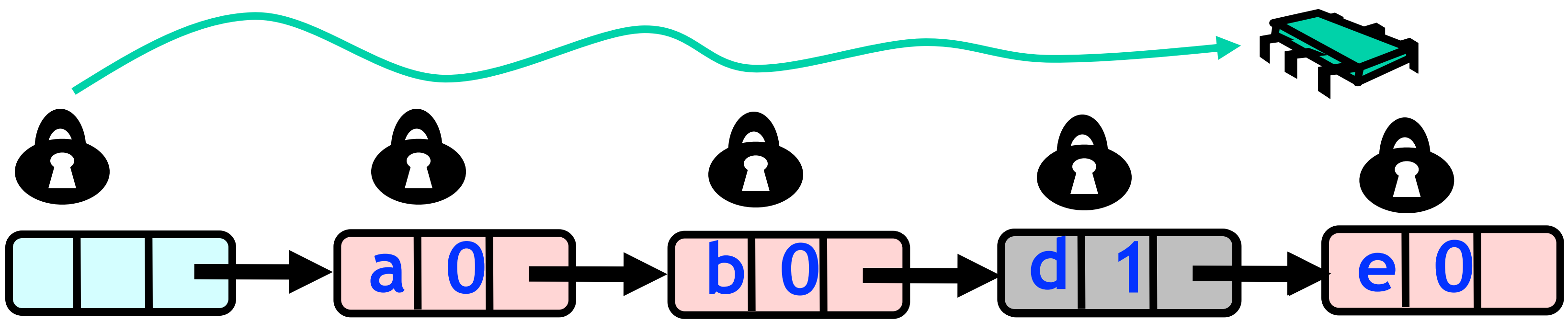


- Limited Hotspots (Only at locked Add(), Remove(), Contains() destination locations, not traversals)

- But two traversals

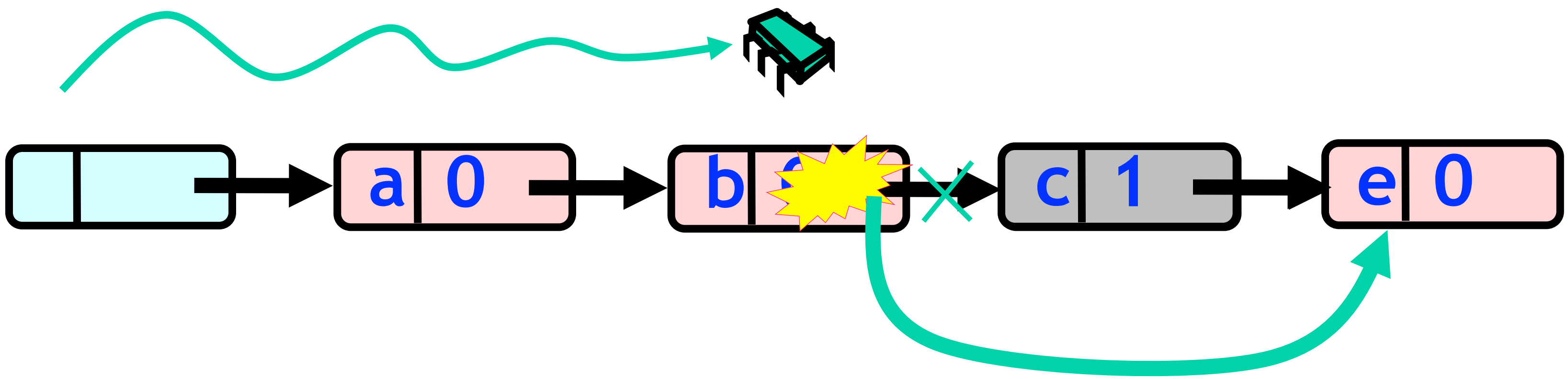
- Yet traversals are wait-free!

Lazy List



Lazy Add() and Remove() + Wait-free Contains()

Lock-free List

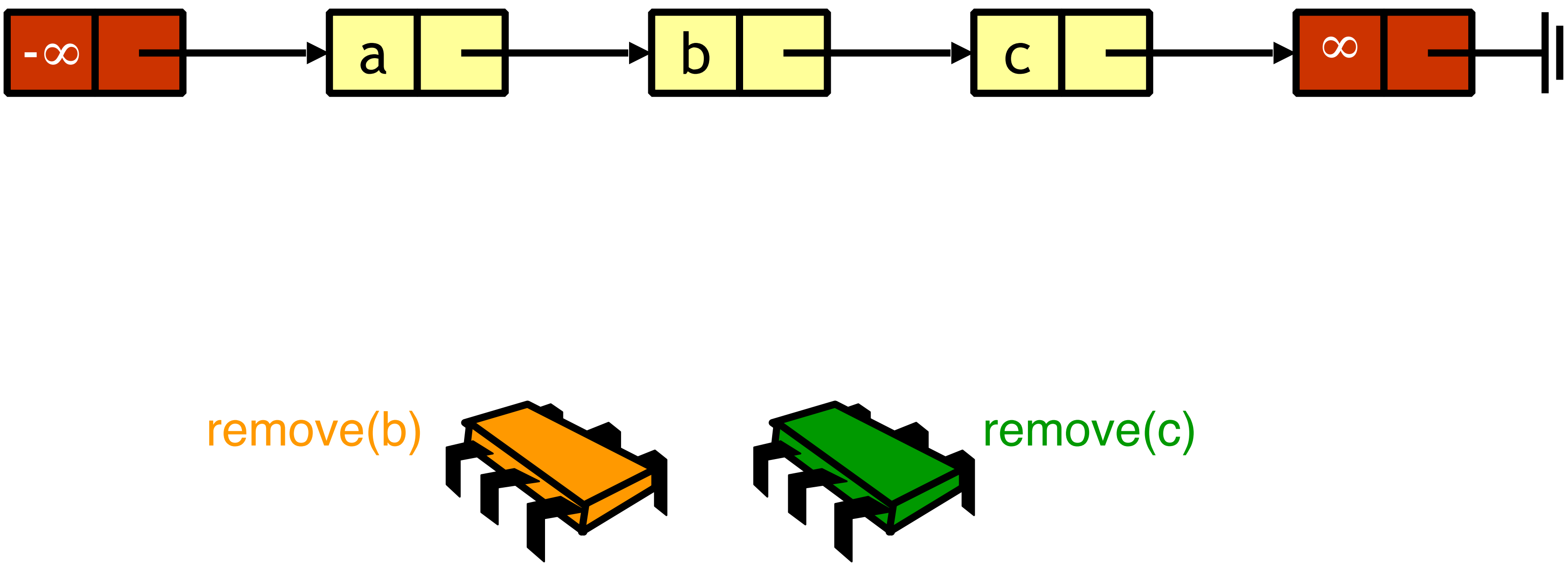


- Add() and Remove() physically remove marked nodes
- Wait-free contains() traverses both marked and removed nodes

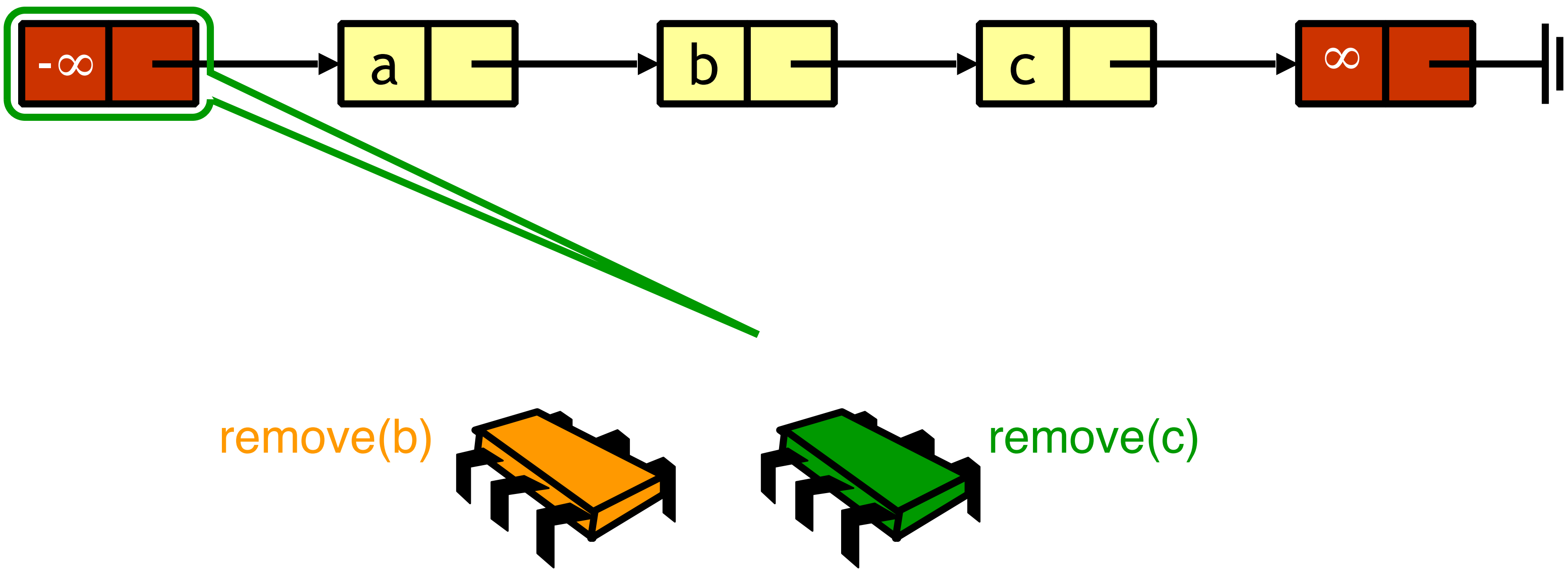
3. Optimistic Synchronization

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

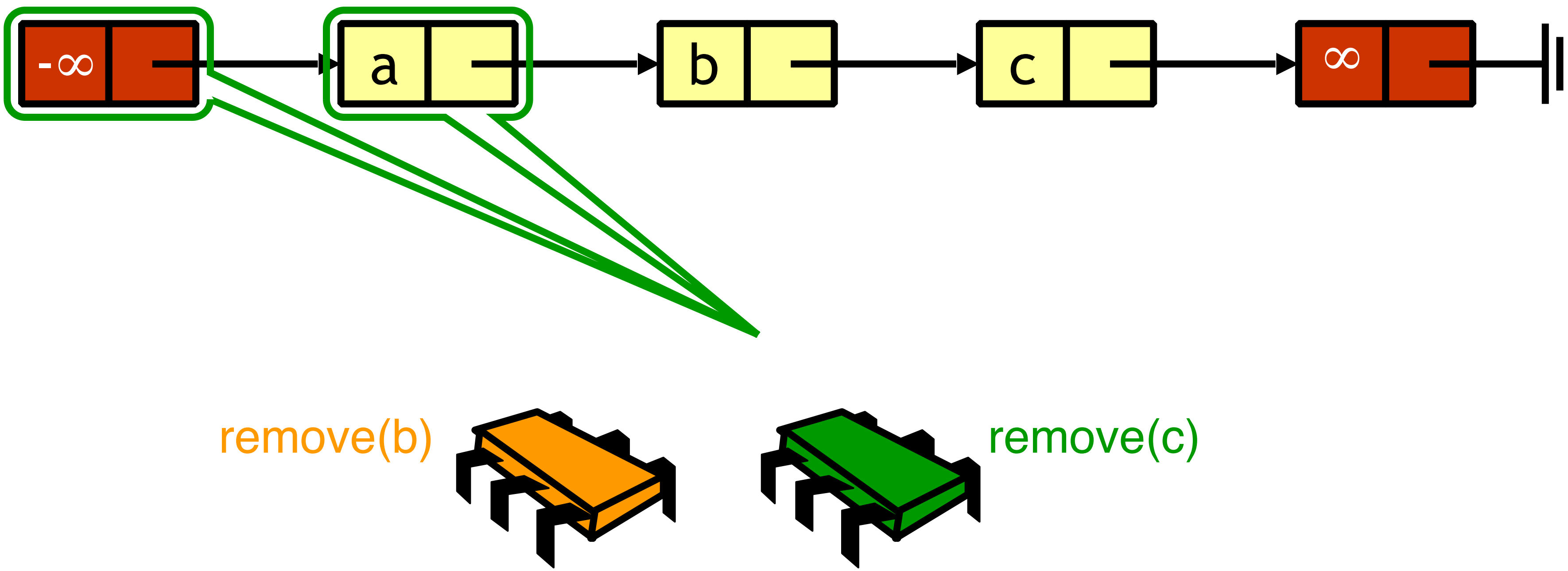
What Could Go Wrong?



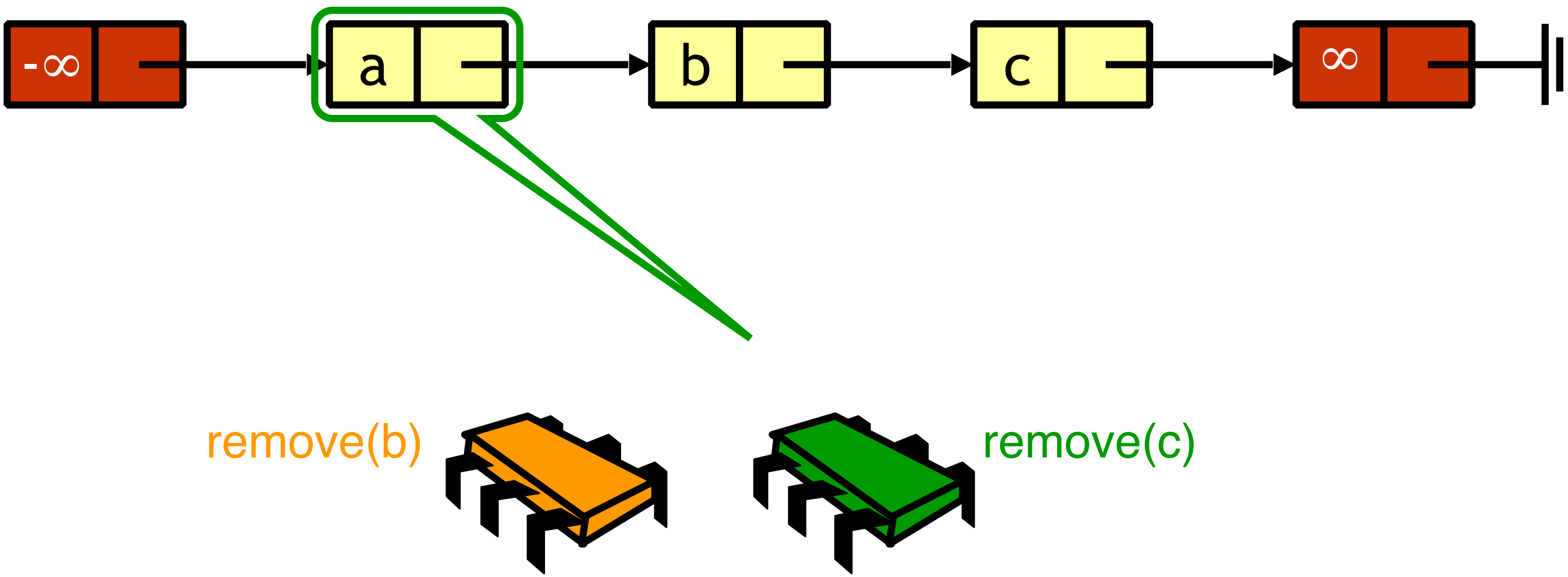
What Could Go Wrong?



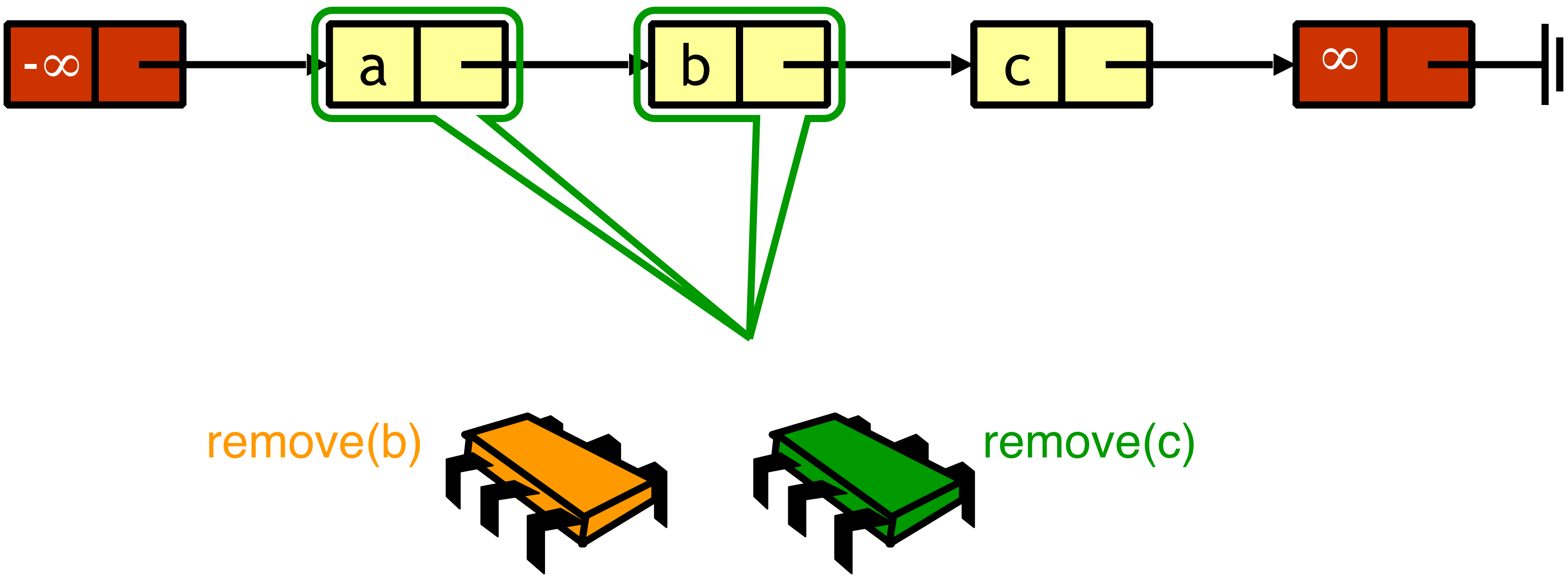
What Could Go Wrong?



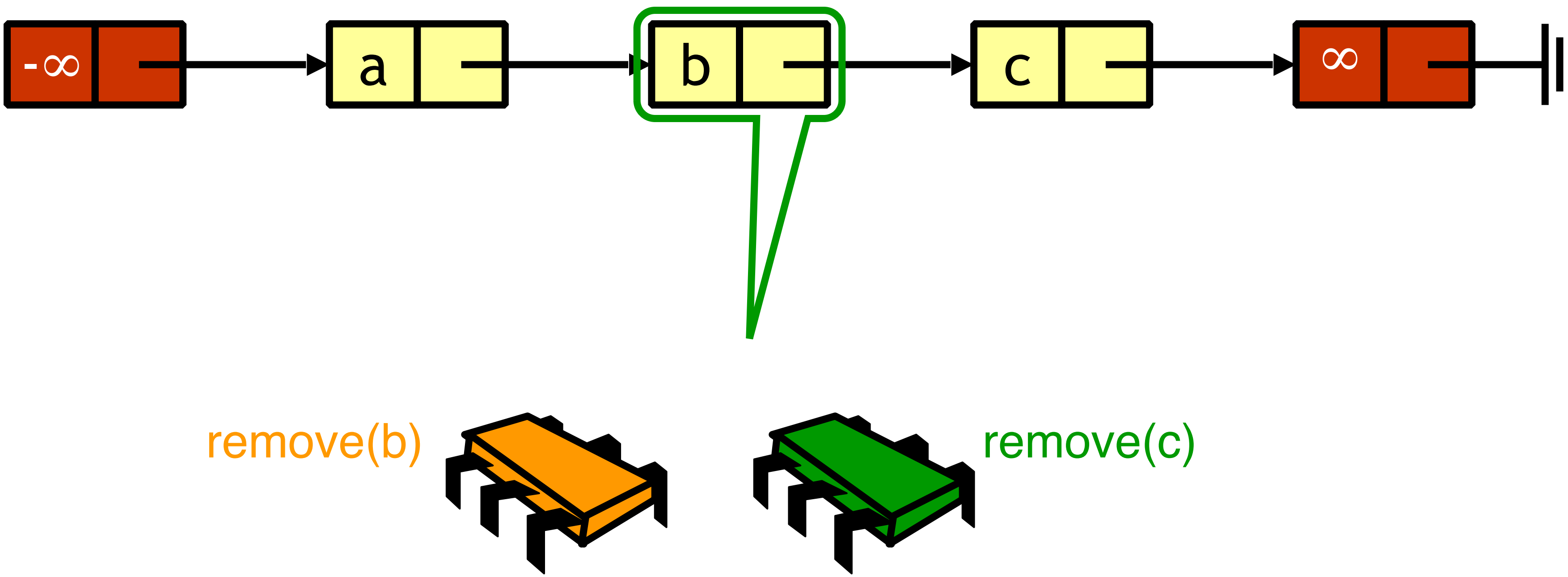
What Could Go Wrong?



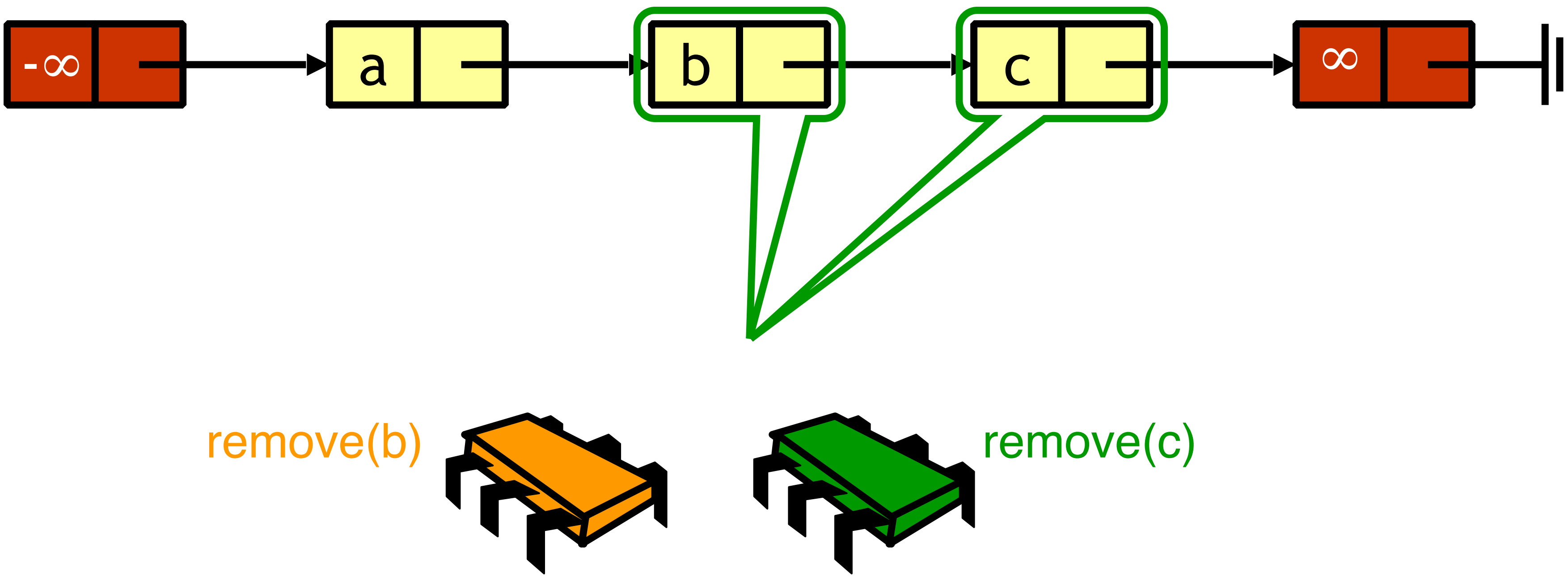
What Could Go Wrong?



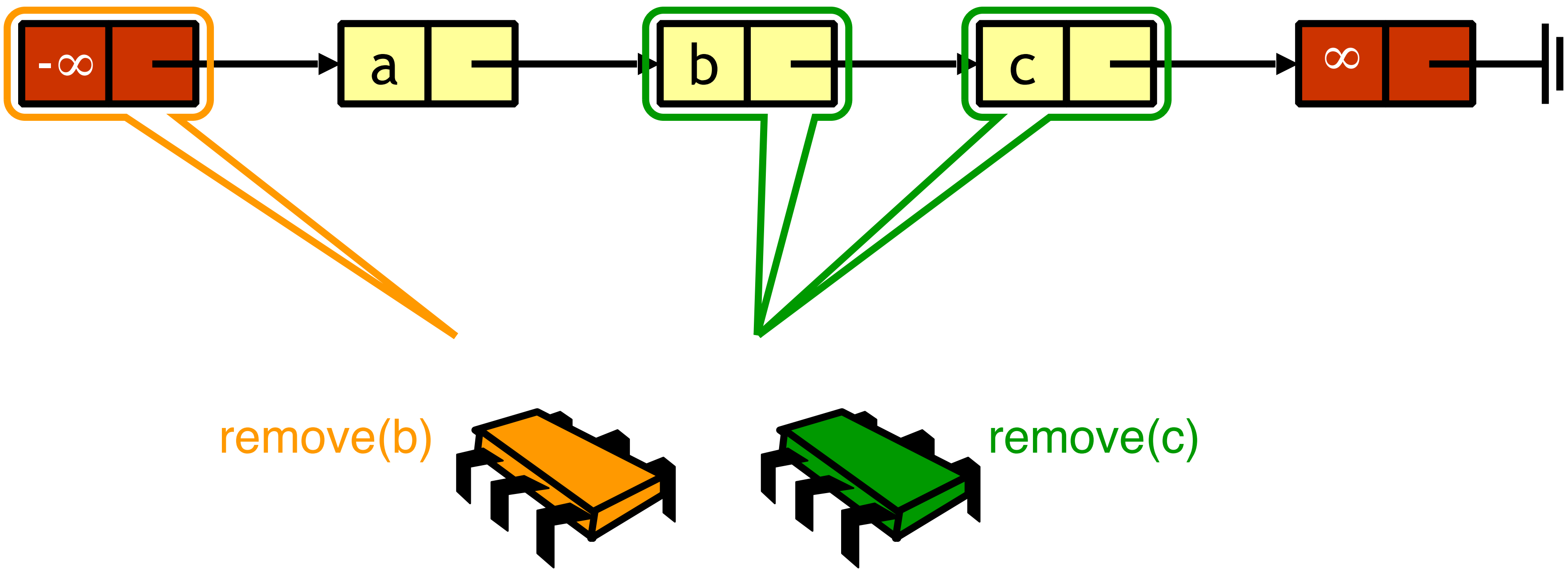
What Could Go Wrong?



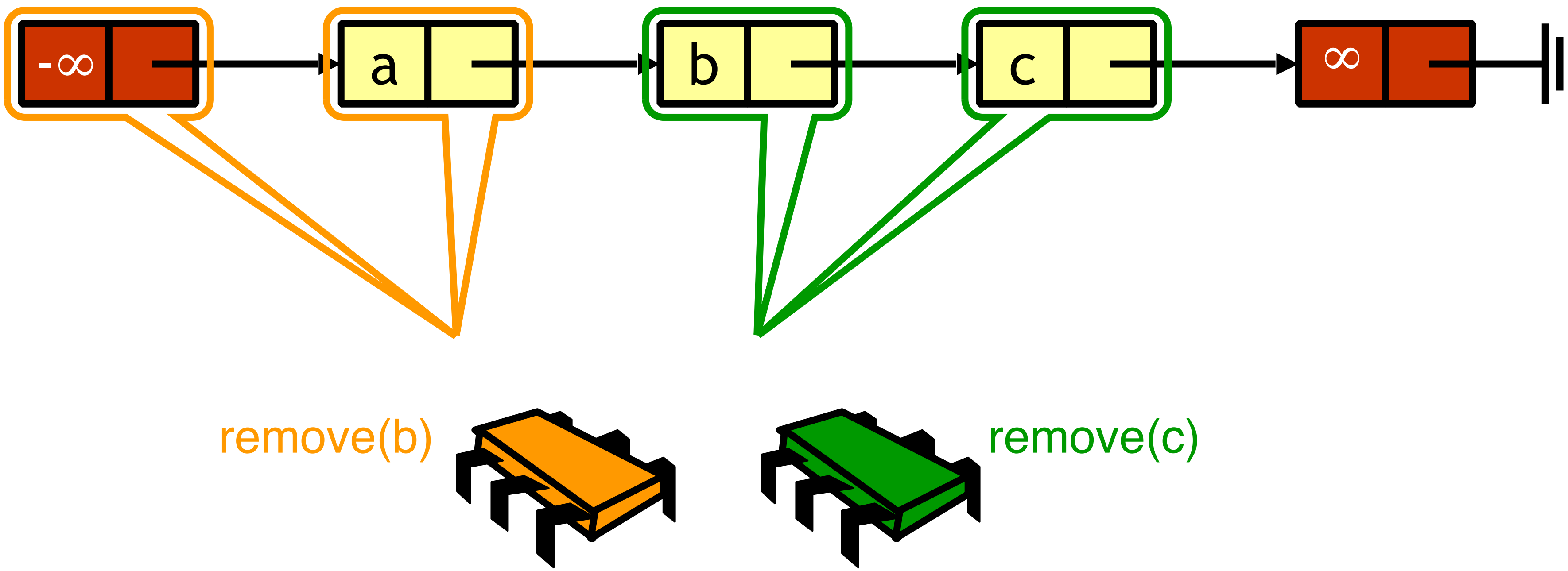
What Could Go Wrong?



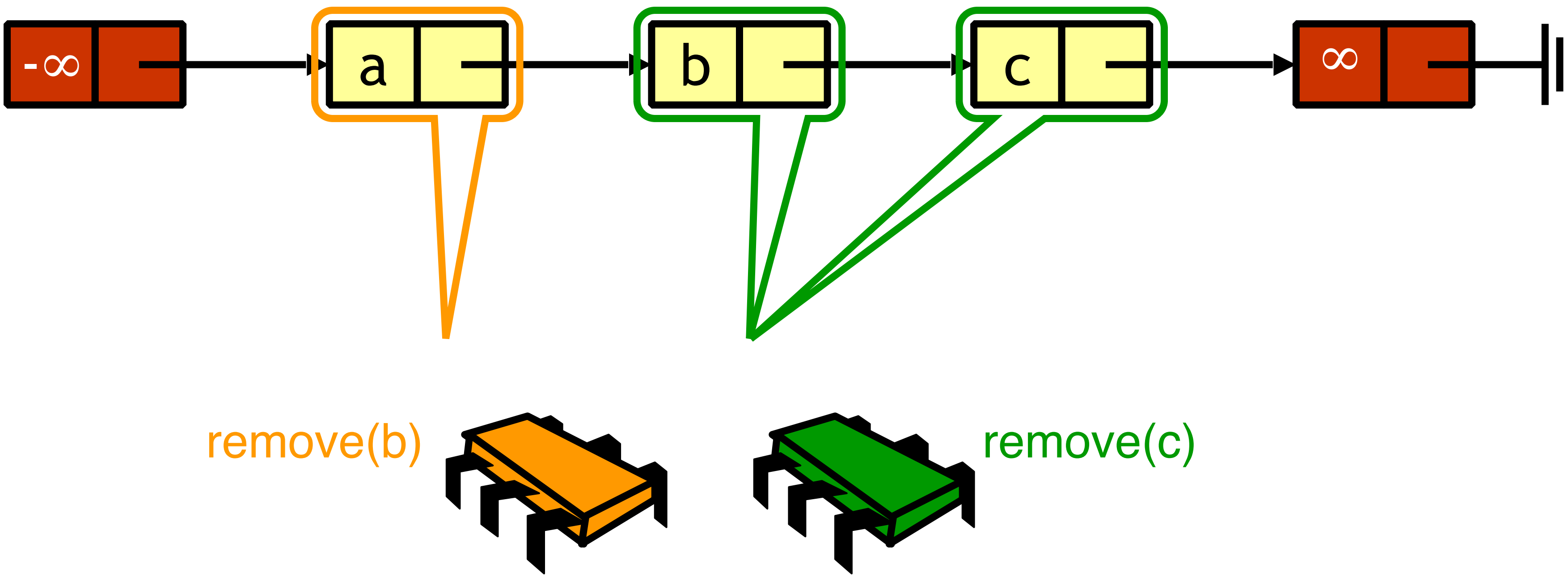
What Could Go Wrong?



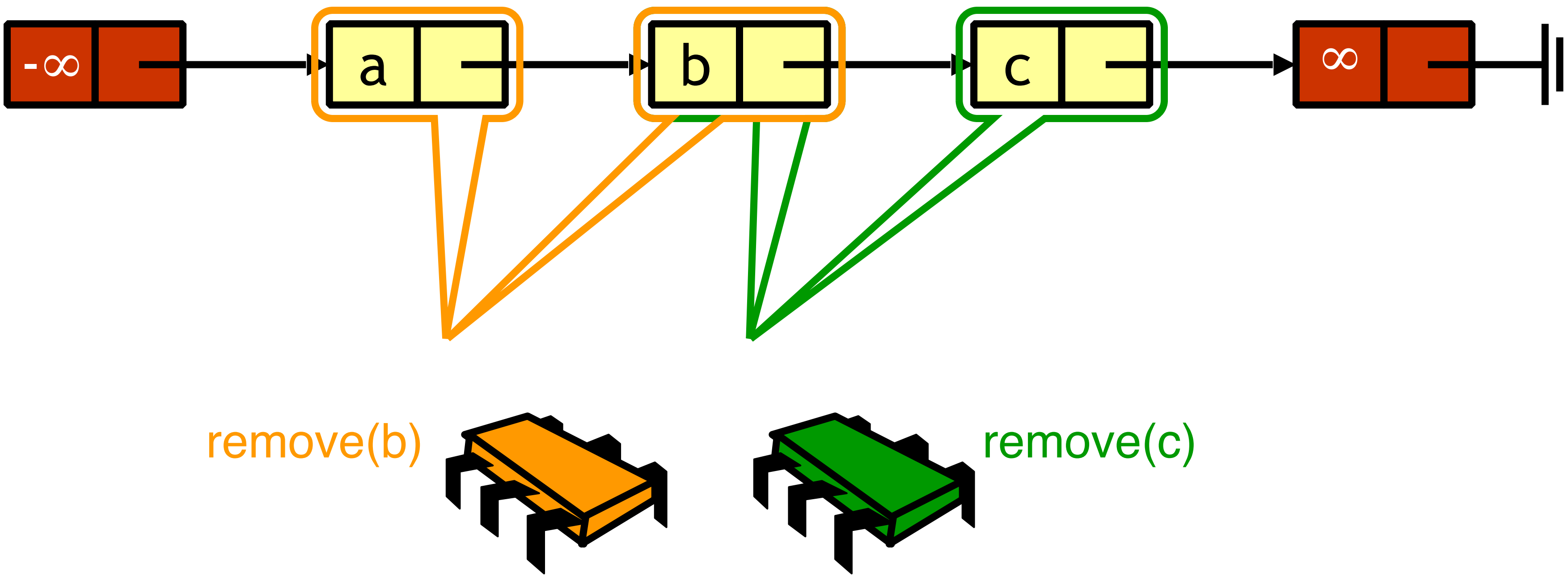
What Could Go Wrong?



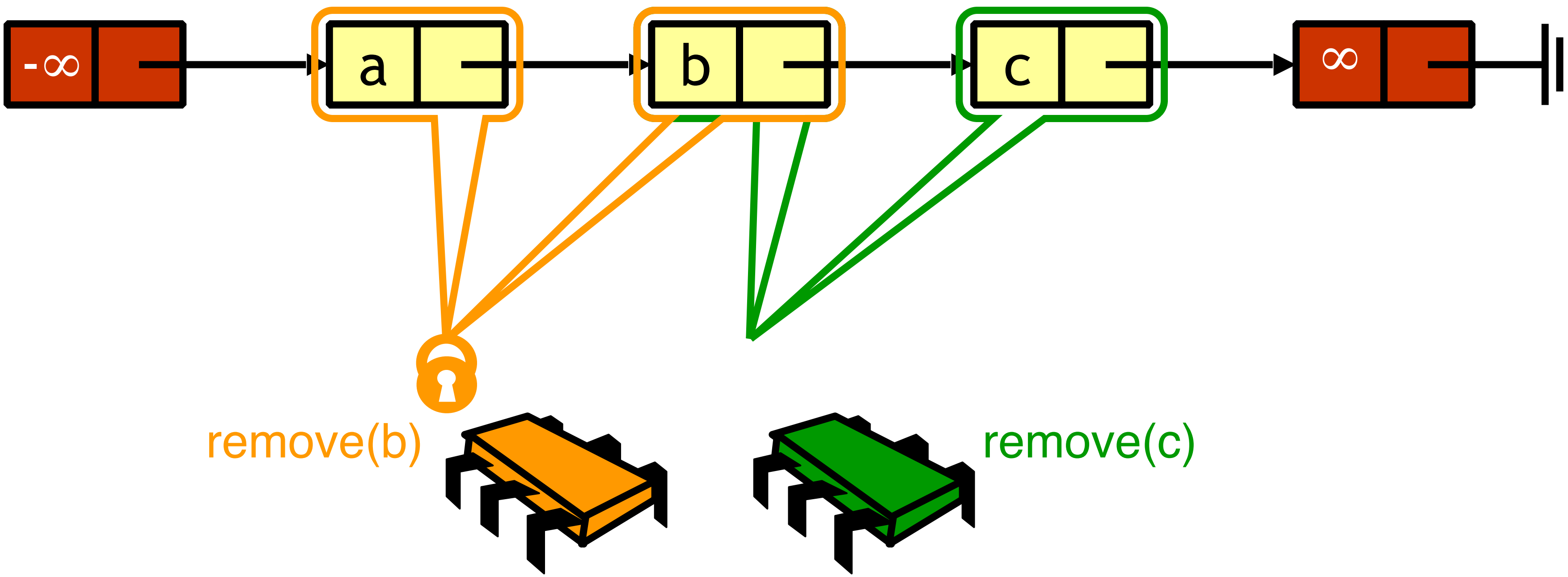
What Could Go Wrong?



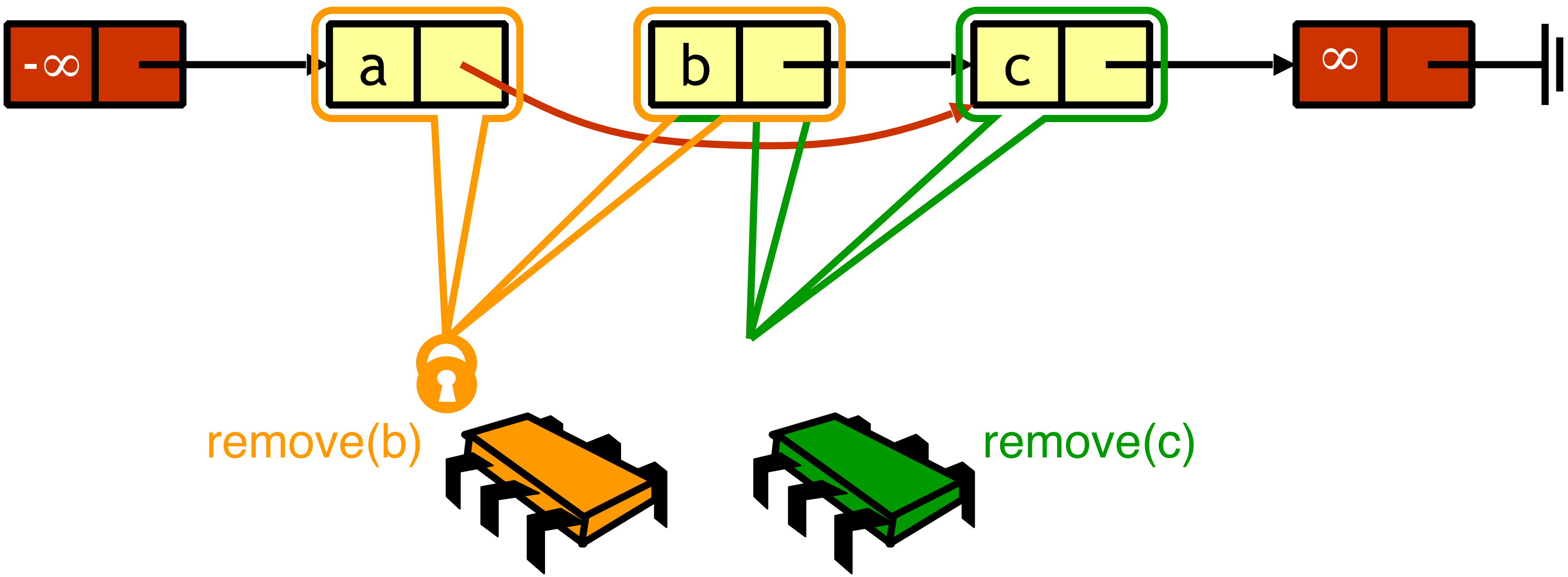
What Could Go Wrong?



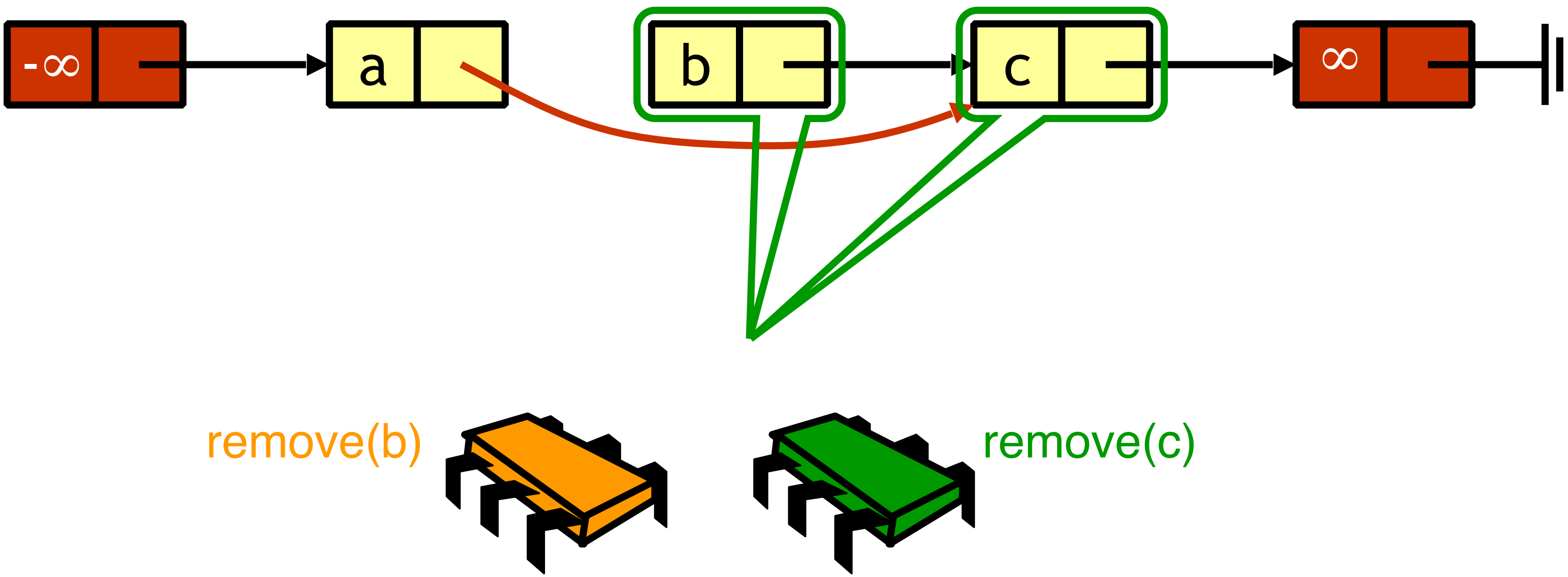
What Could Go Wrong?



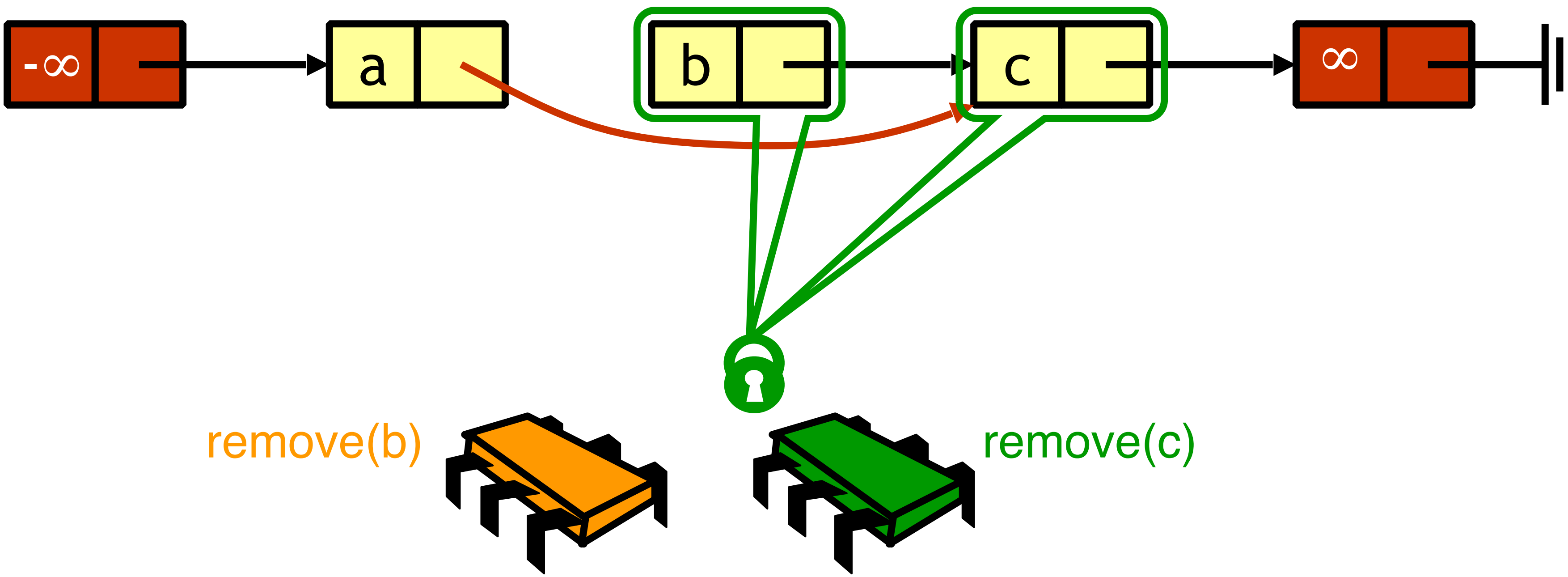
What Could Go Wrong?



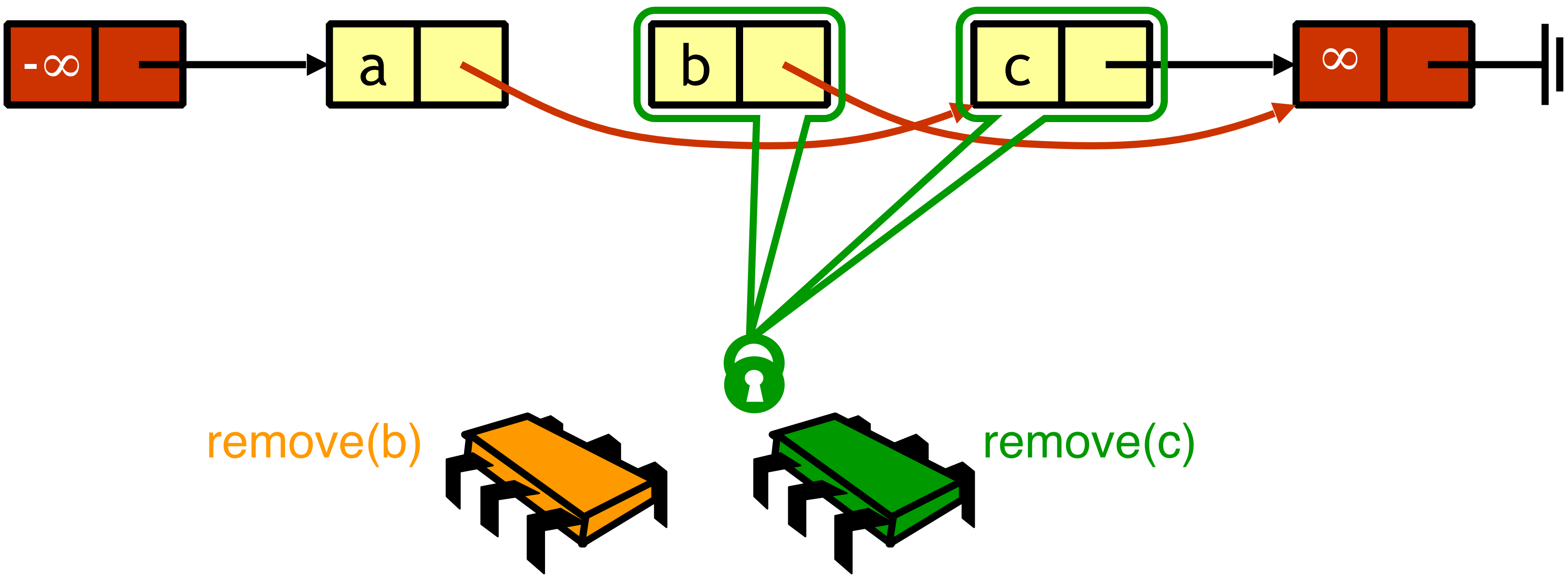
What Could Go Wrong?



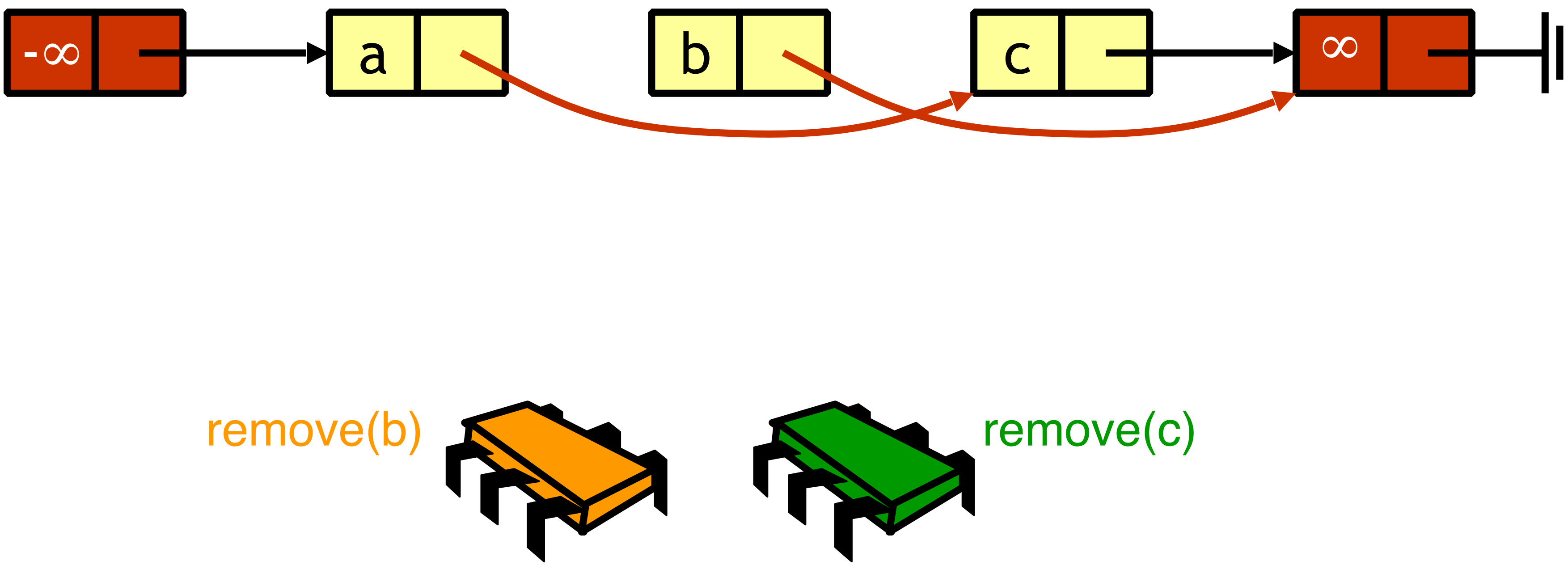
What Could Go Wrong?



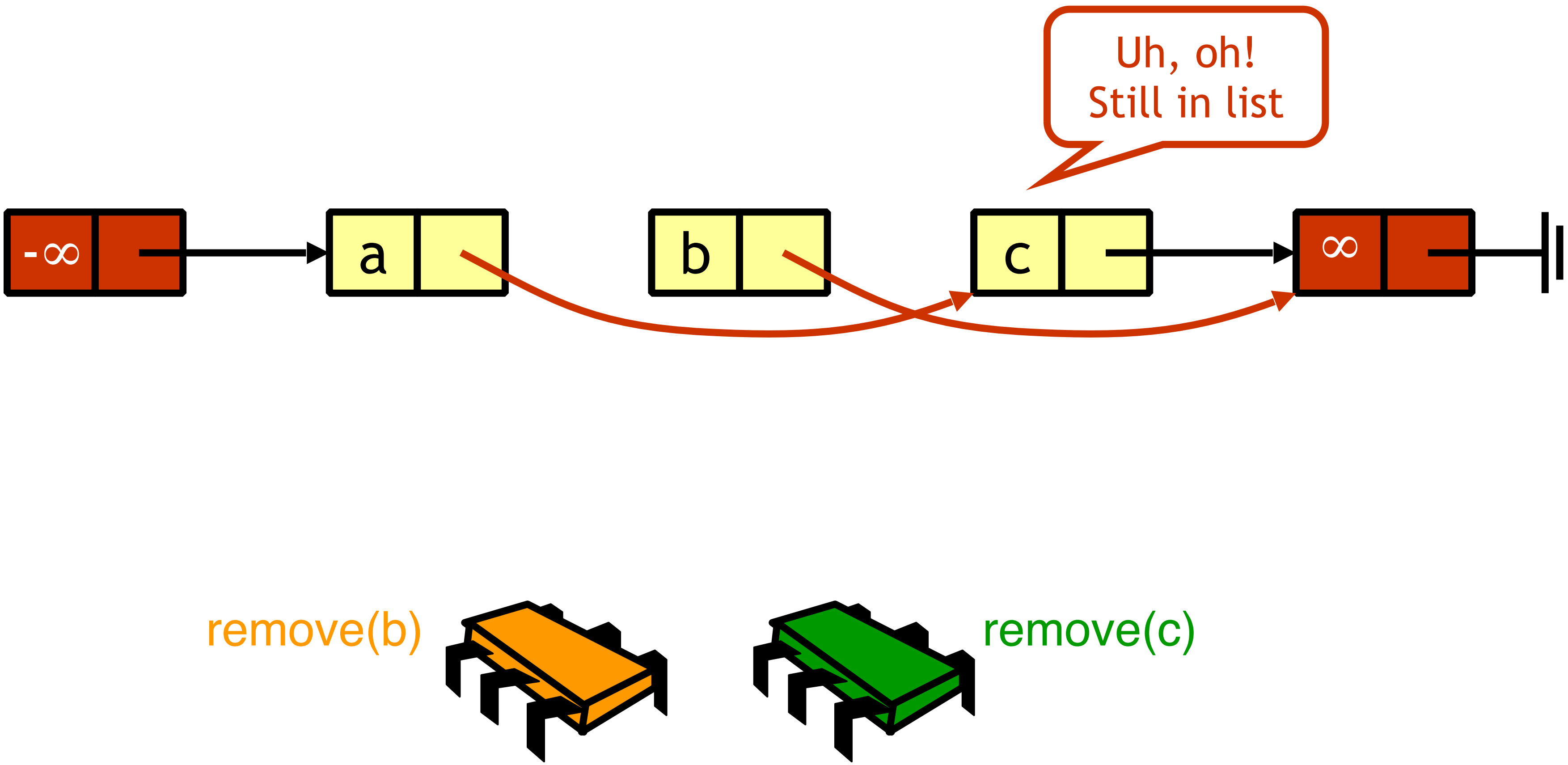
What Could Go Wrong?



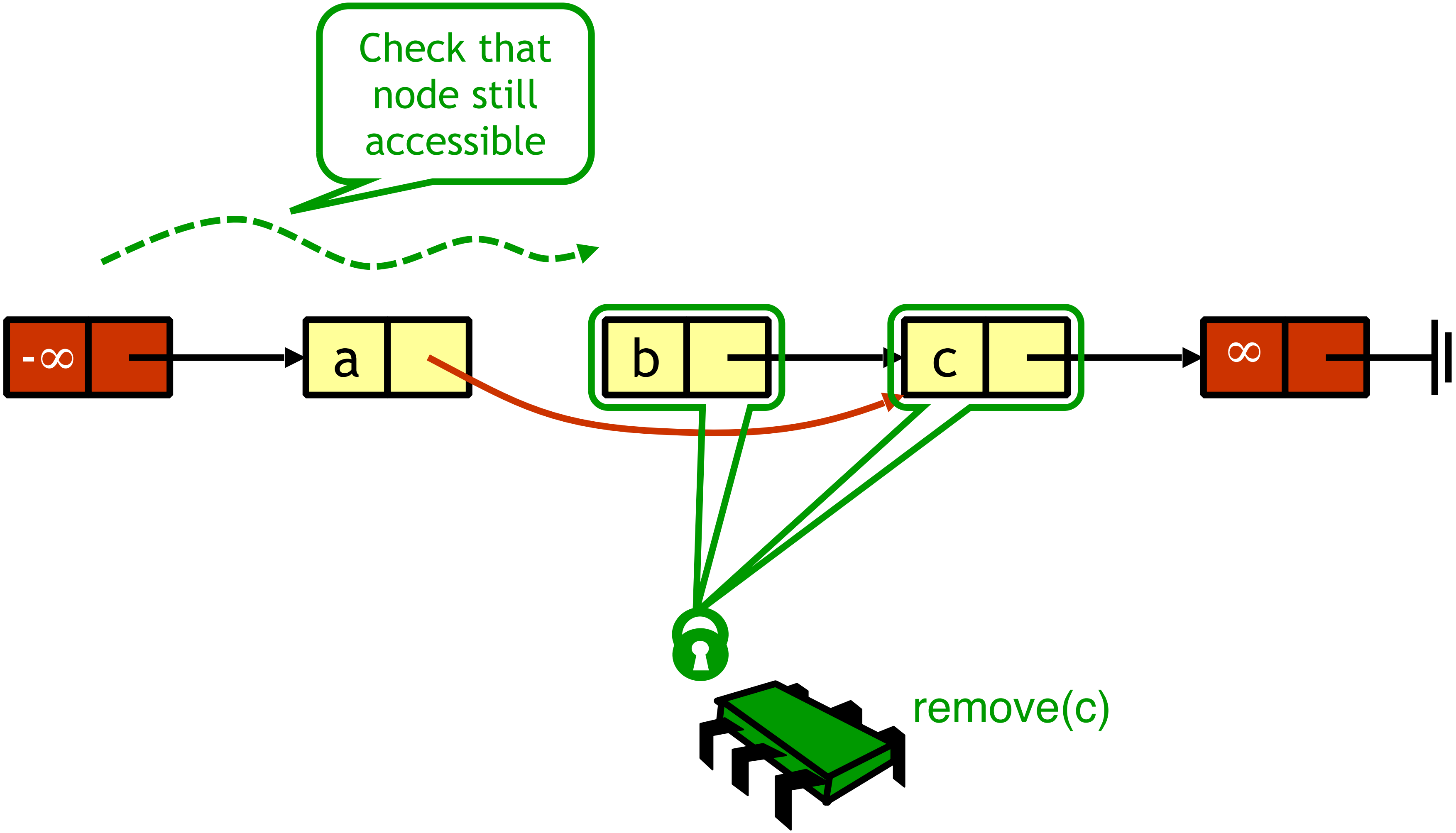
What Could Go Wrong?



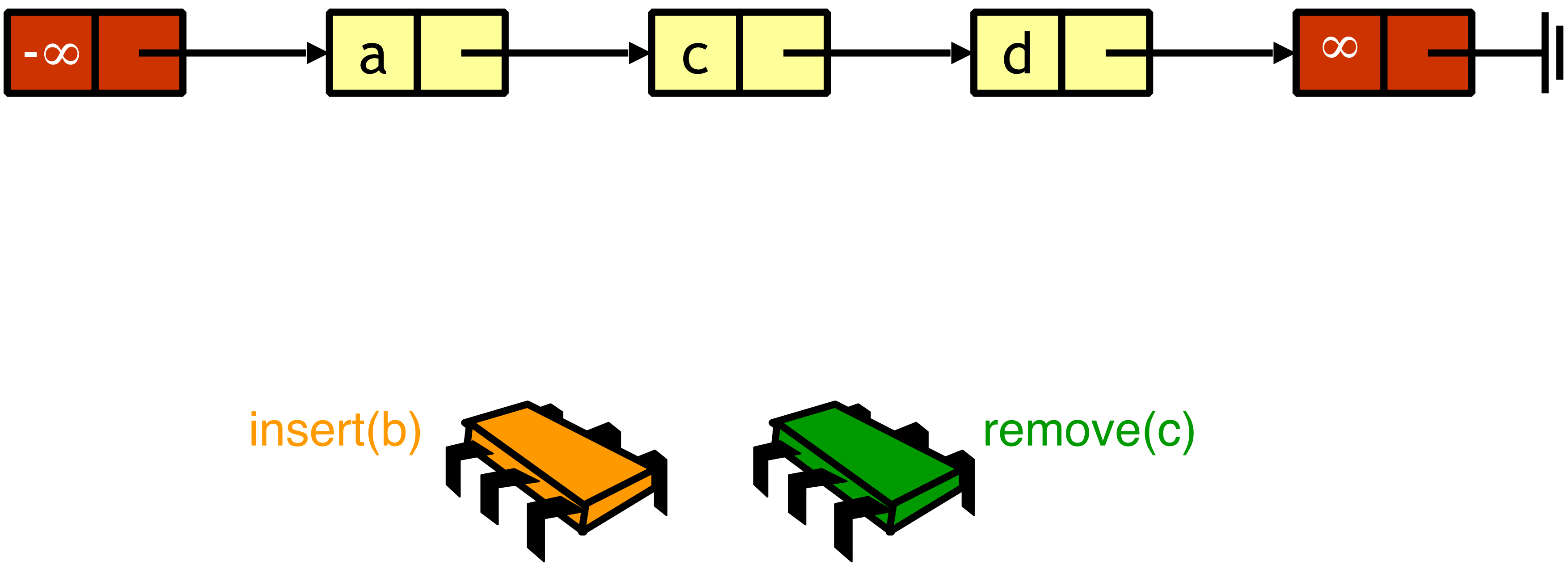
What Could Go Wrong?



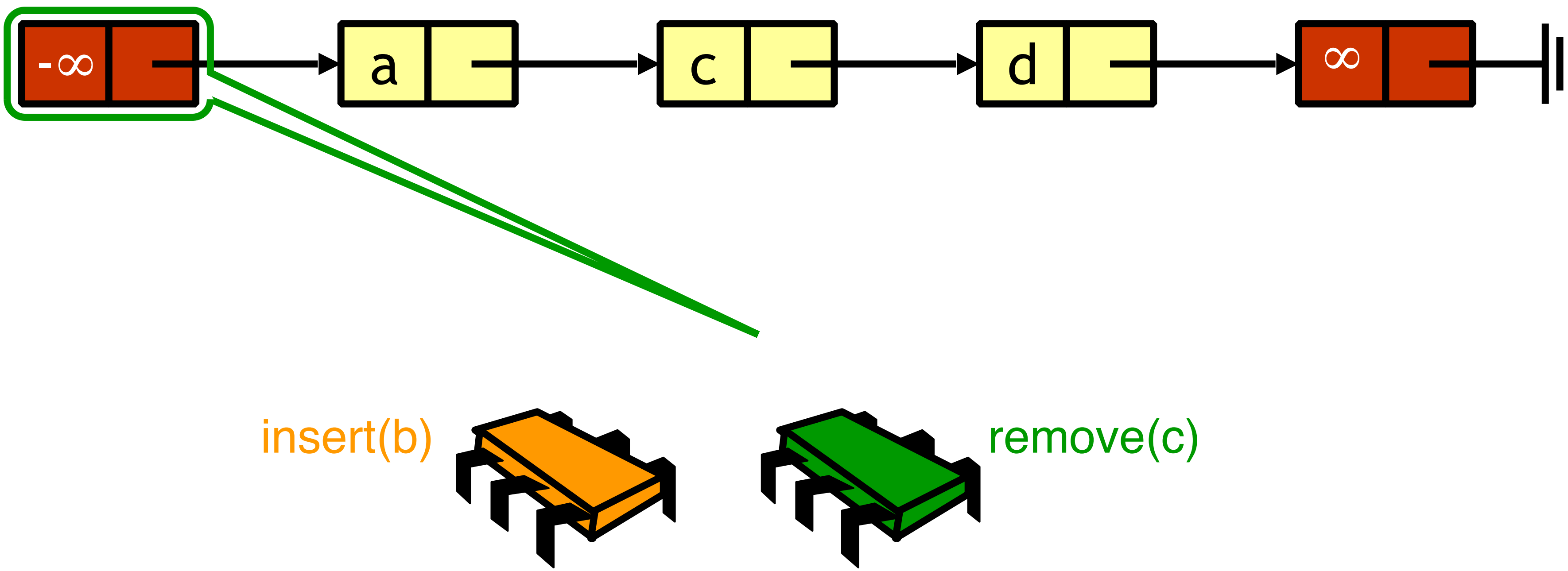
What Could Go Wrong?



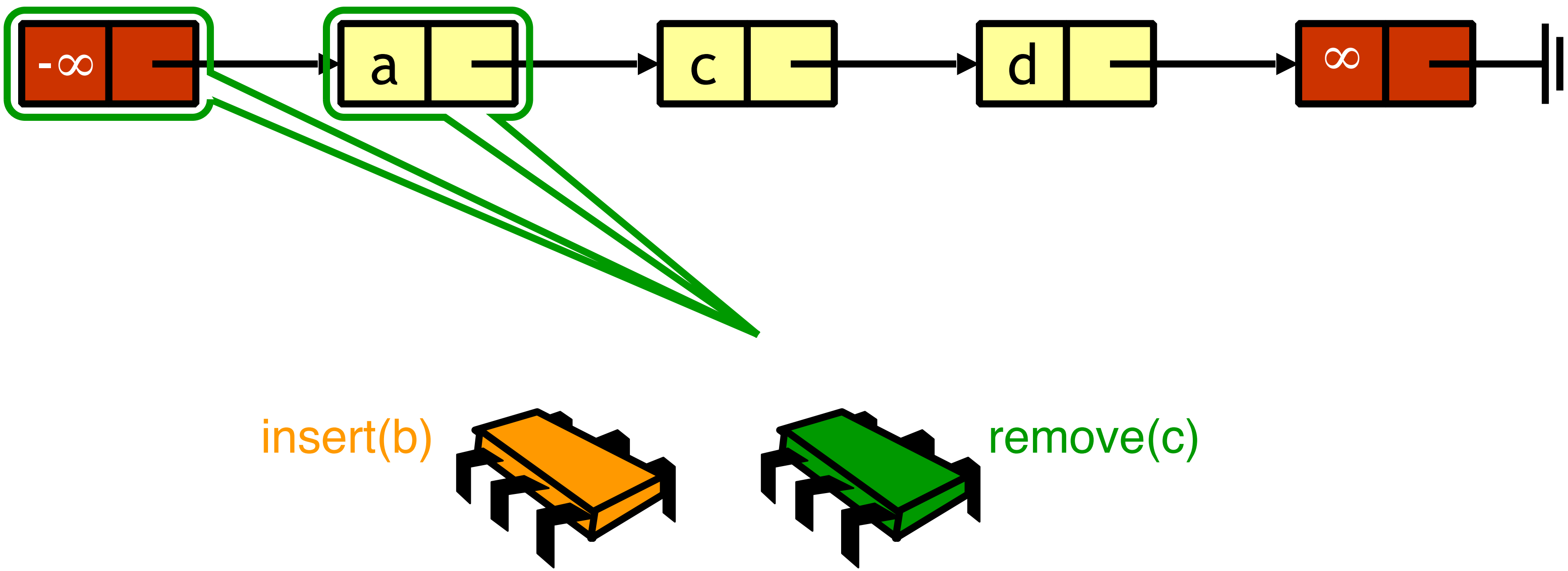
What Could Go Wrong?



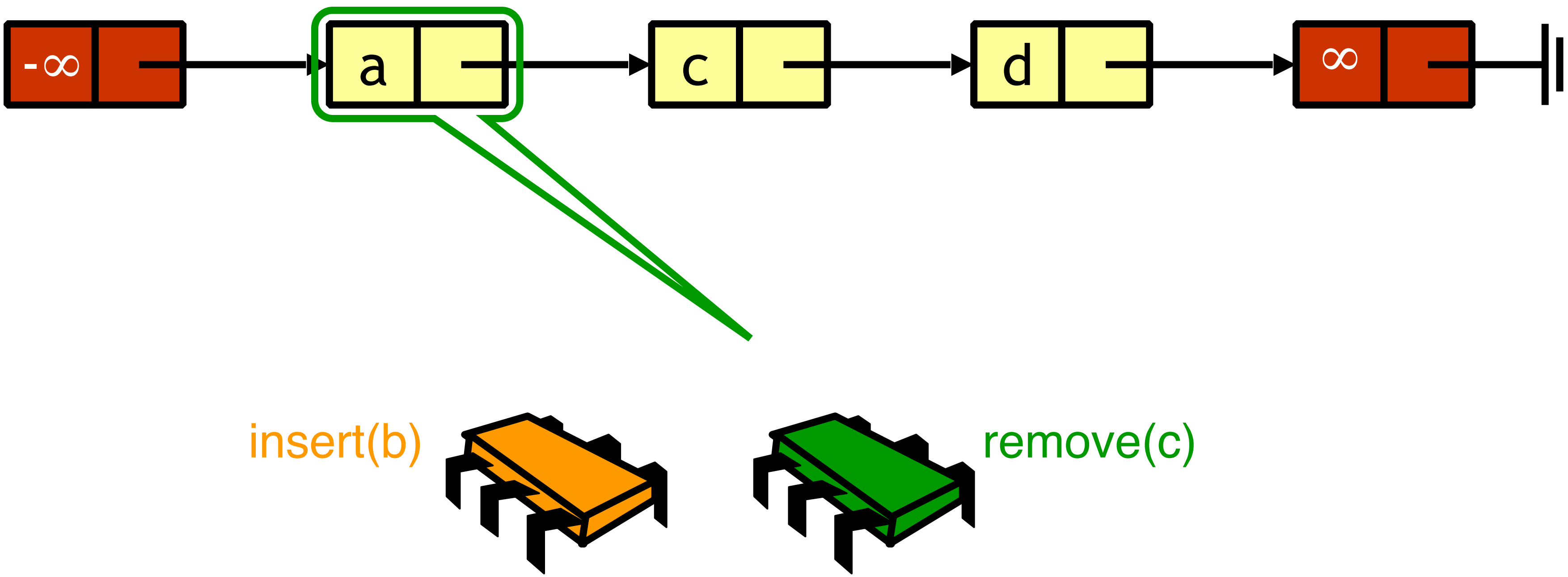
What Could Go Wrong?



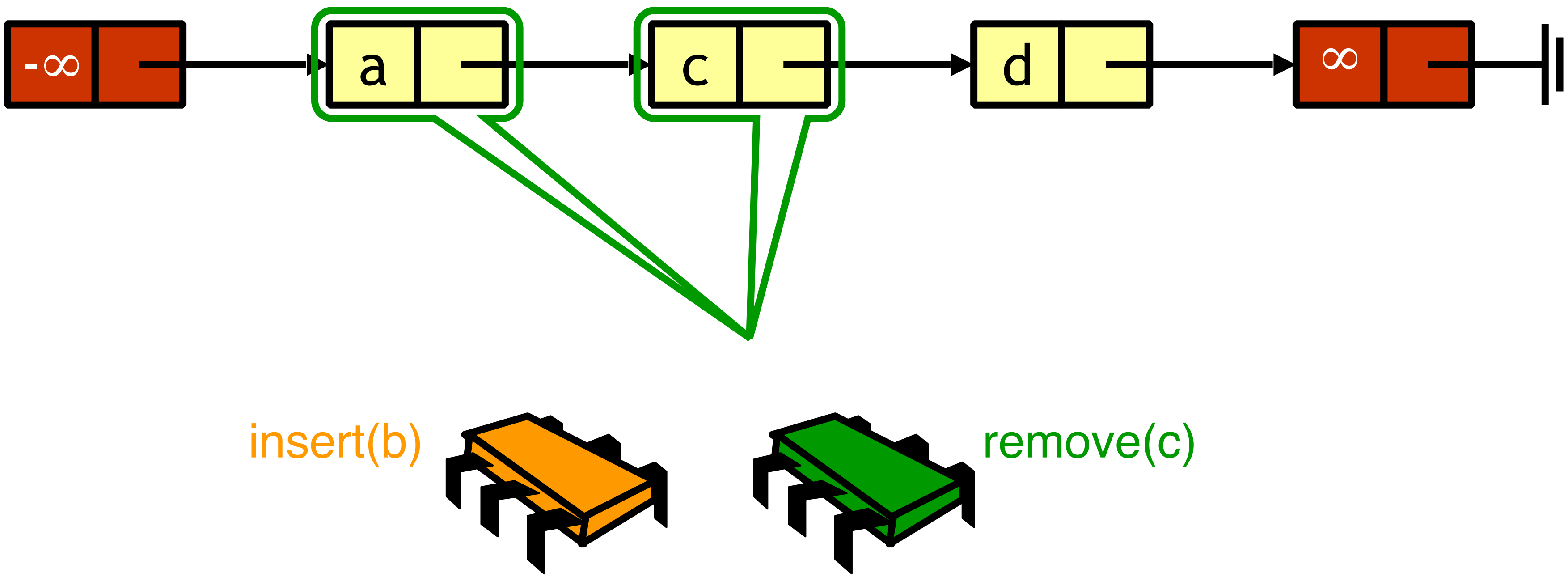
What Could Go Wrong?



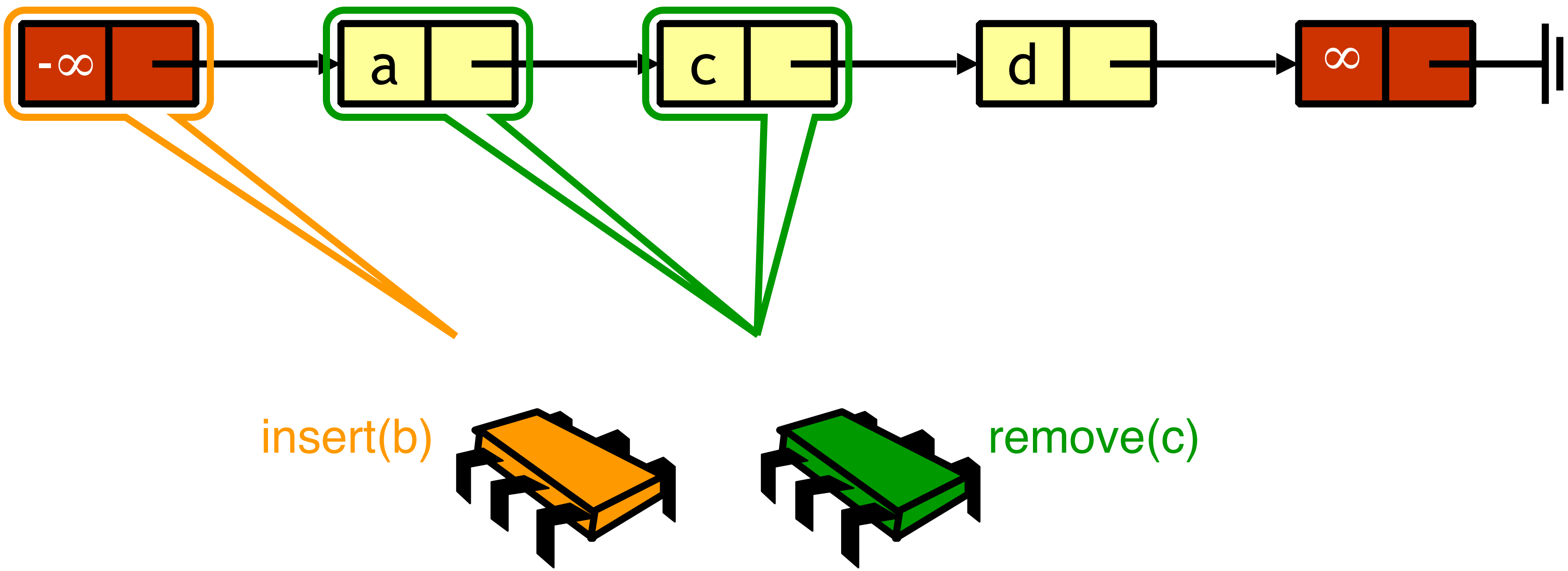
What Could Go Wrong?



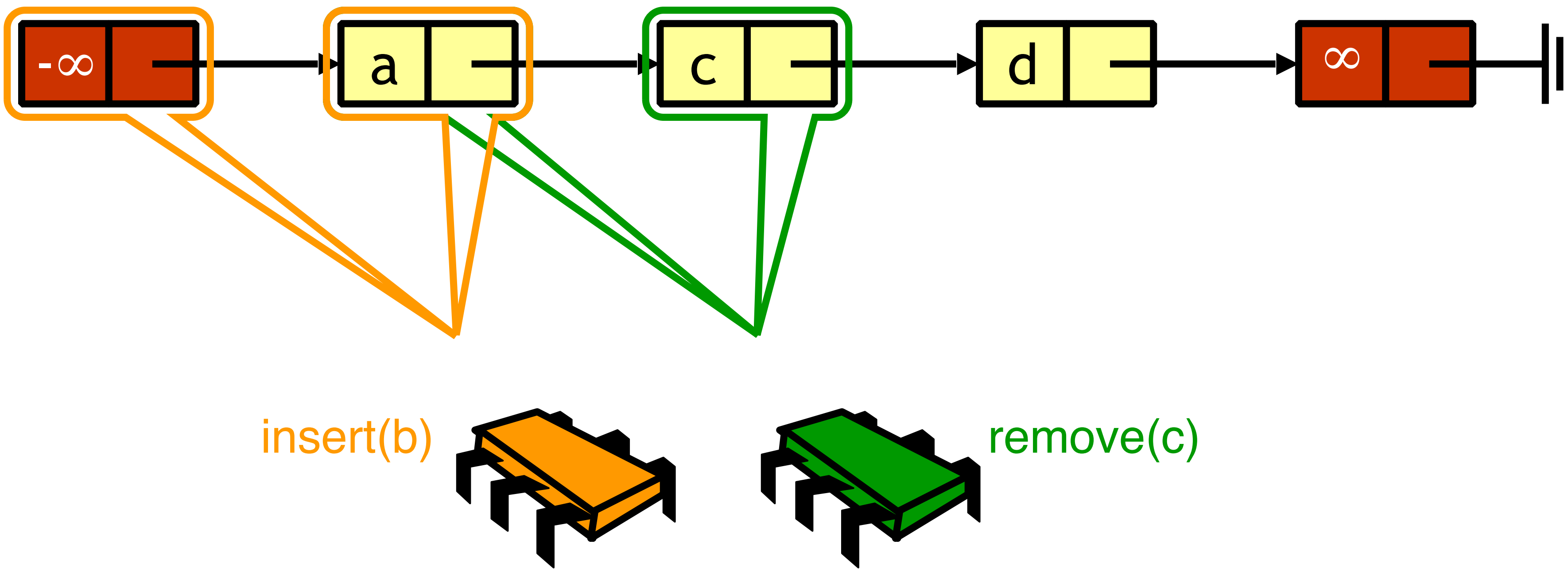
What Could Go Wrong?



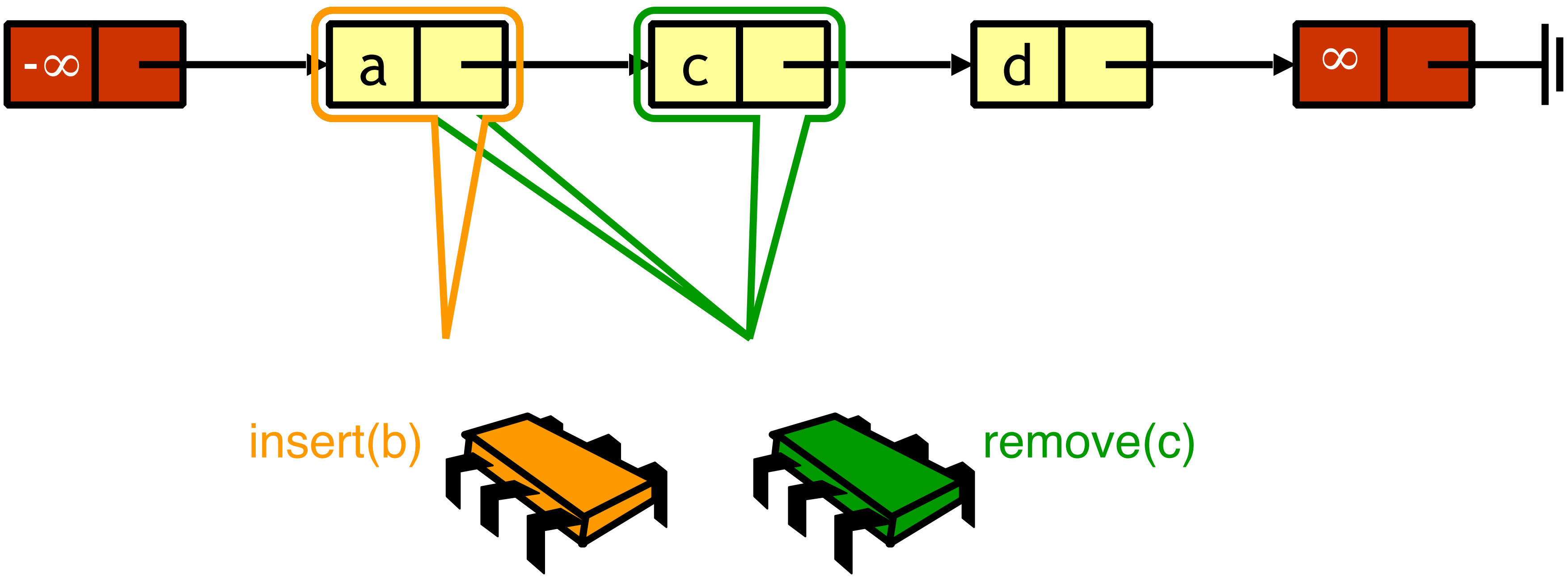
What Could Go Wrong?



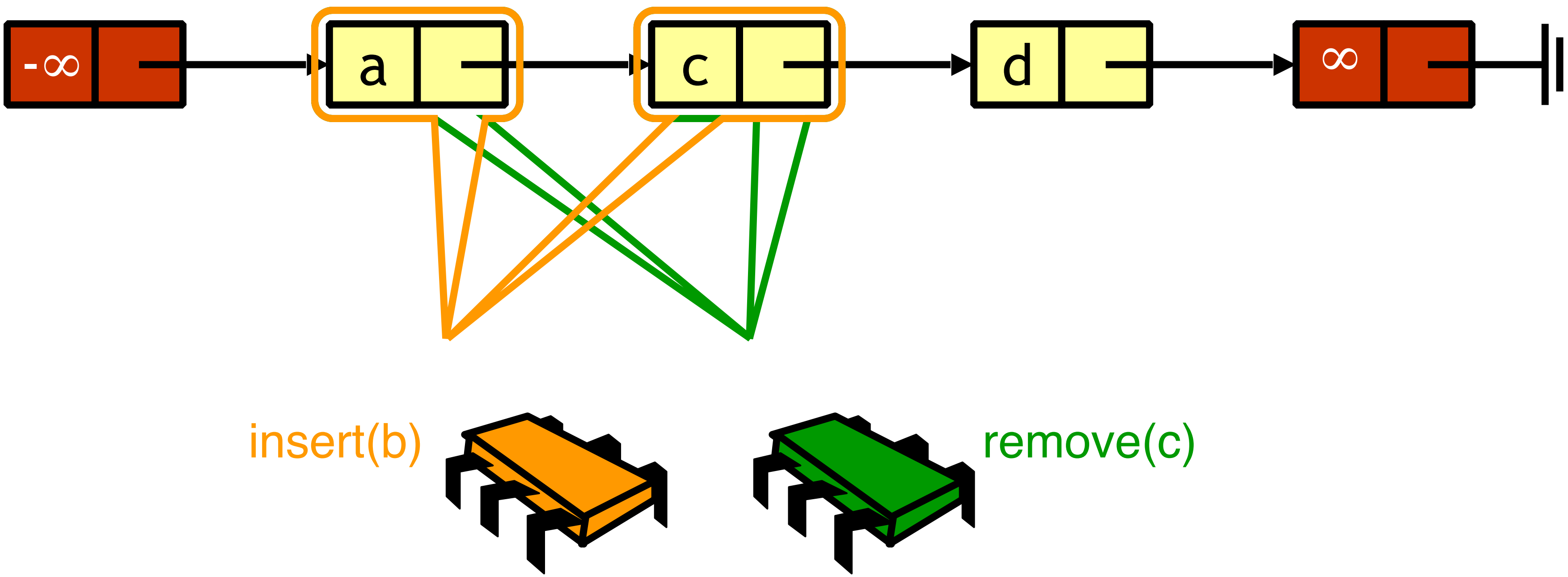
What Could Go Wrong?



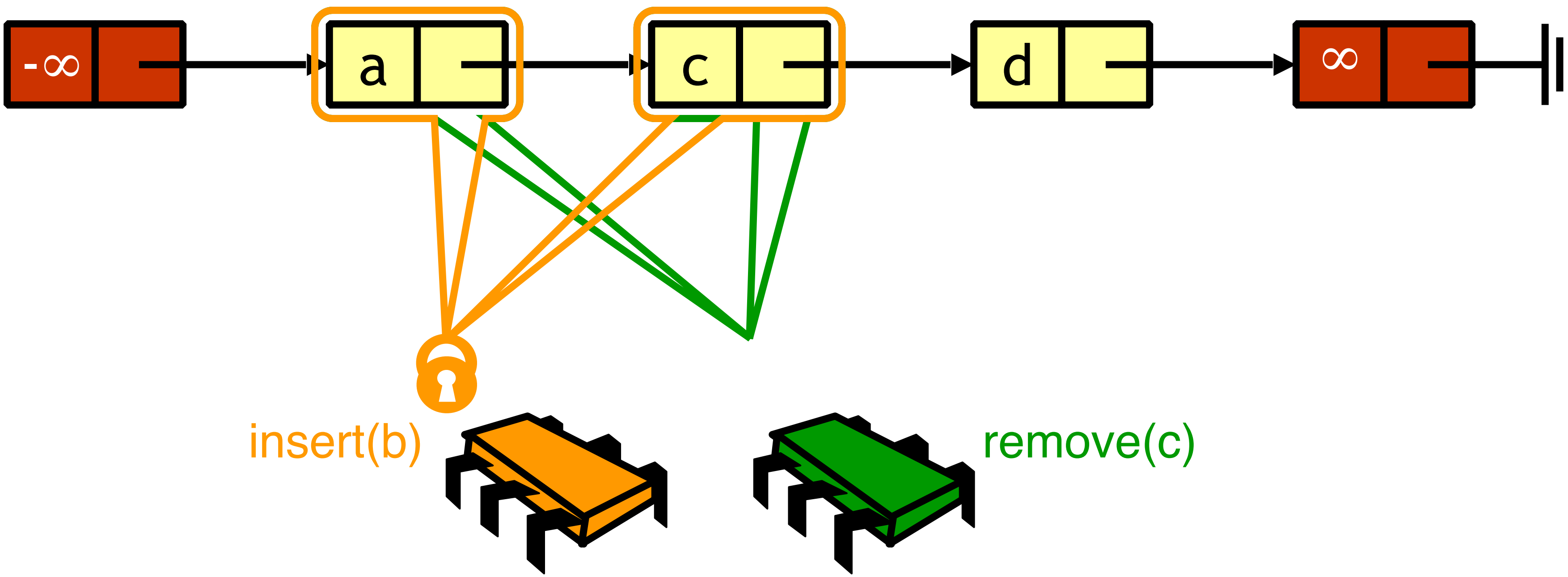
What Could Go Wrong?



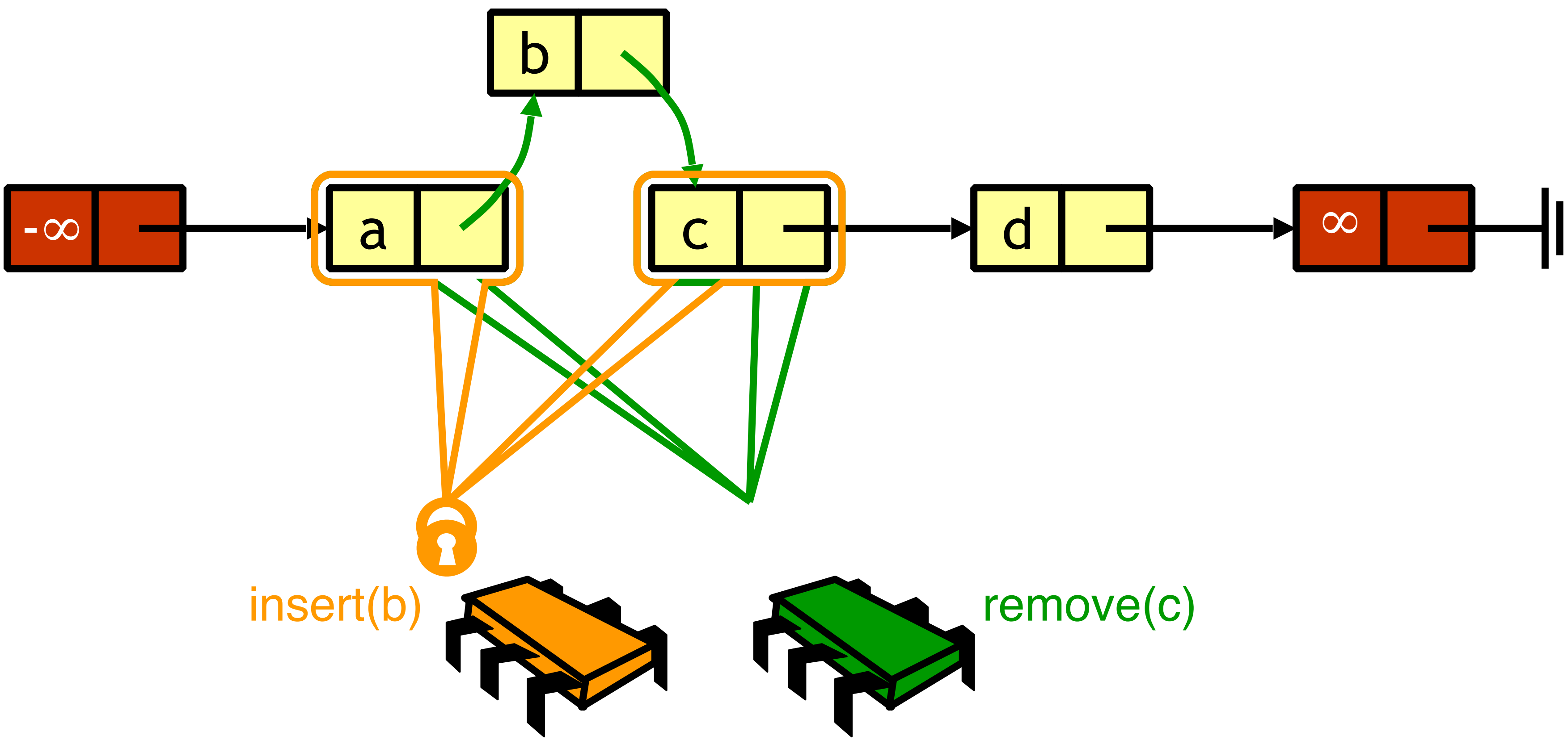
What Could Go Wrong?



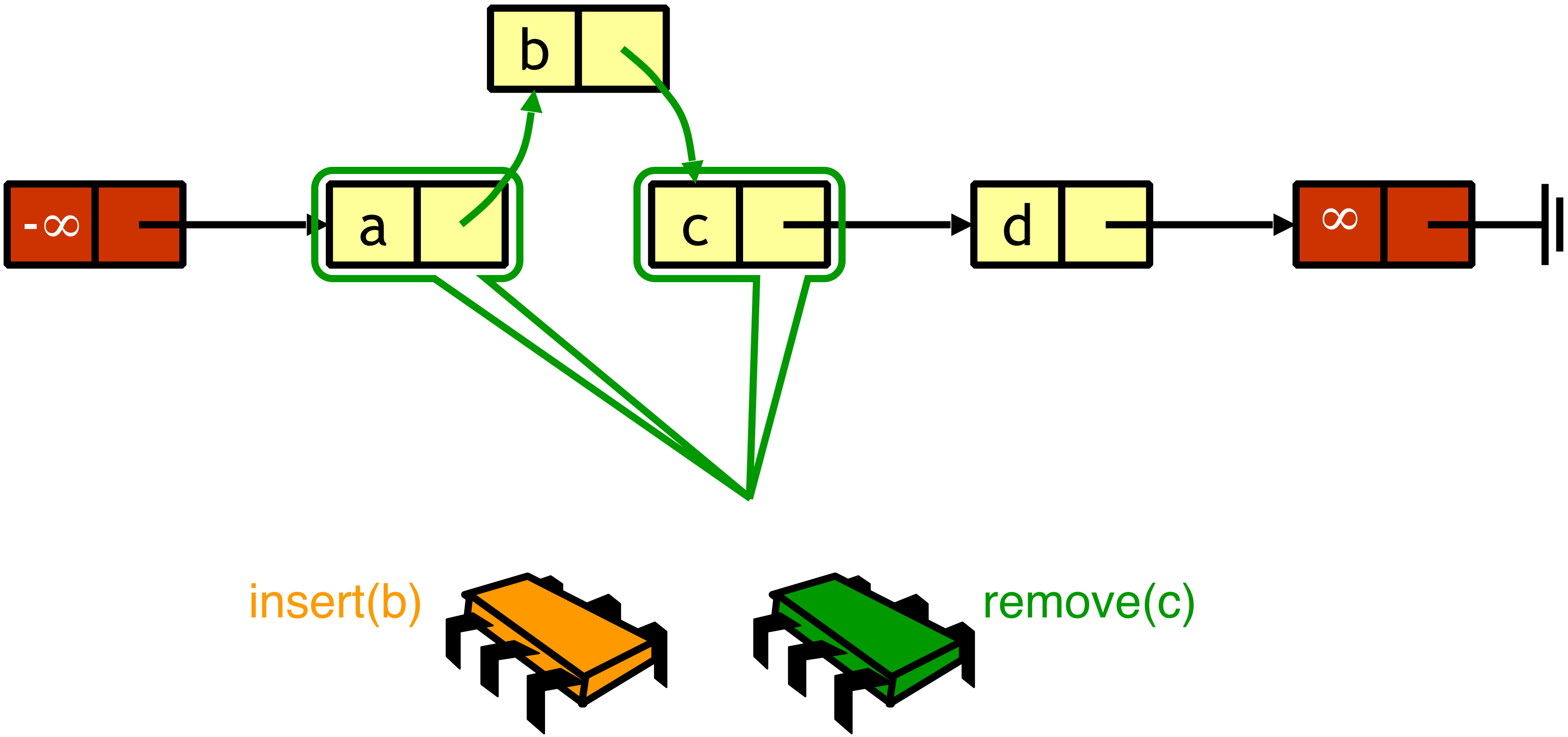
What Could Go Wrong?



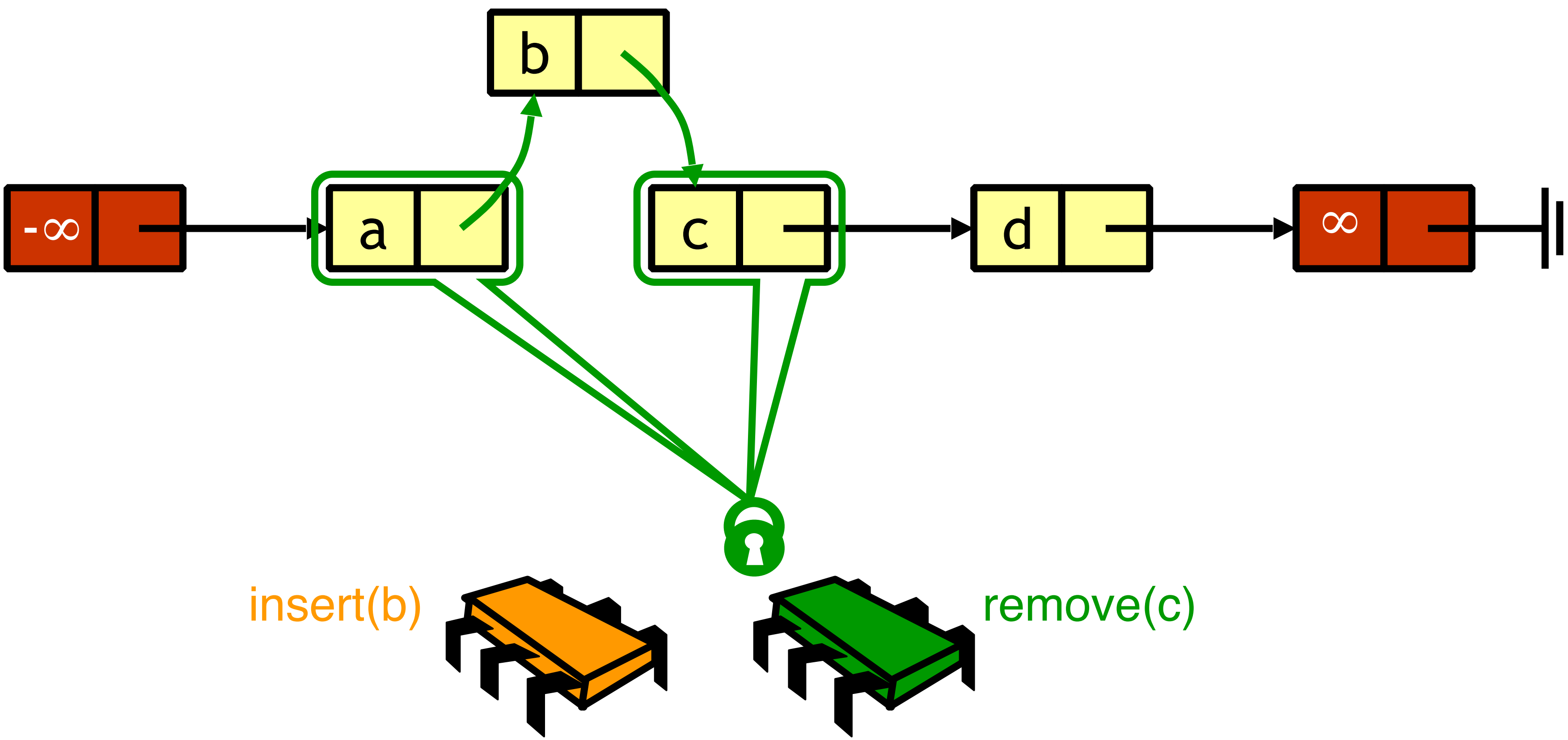
What Could Go Wrong?



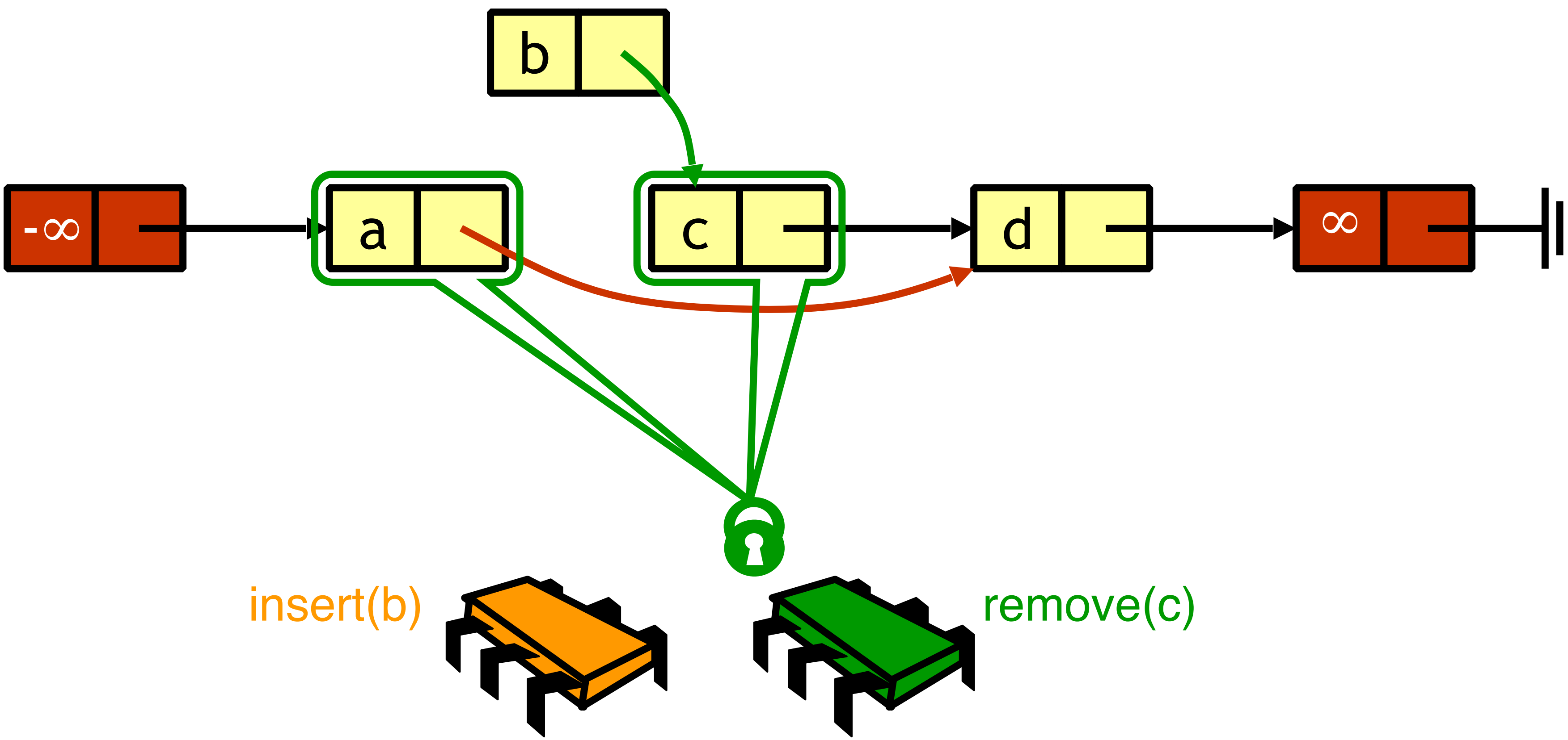
What Could Go Wrong?



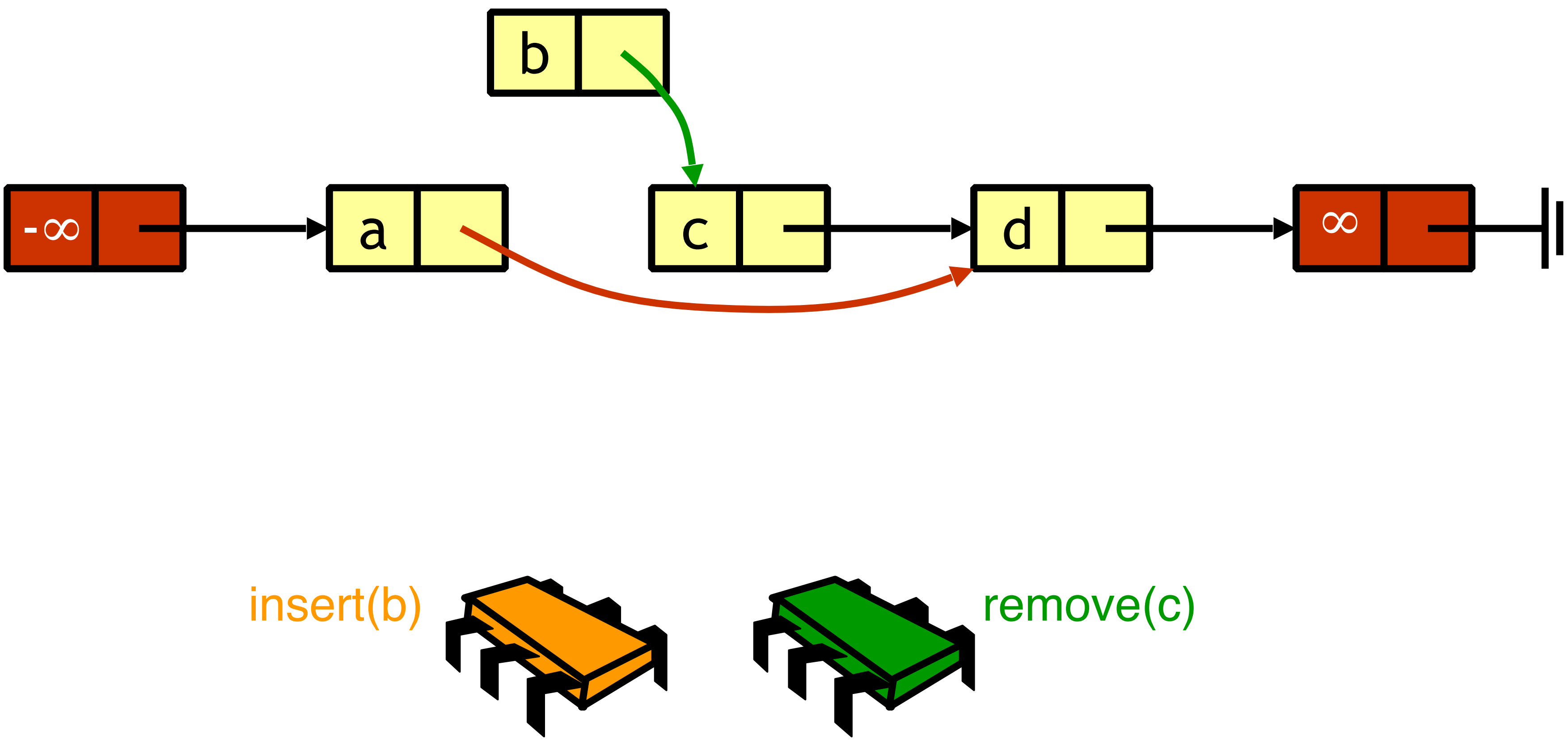
What Could Go Wrong?



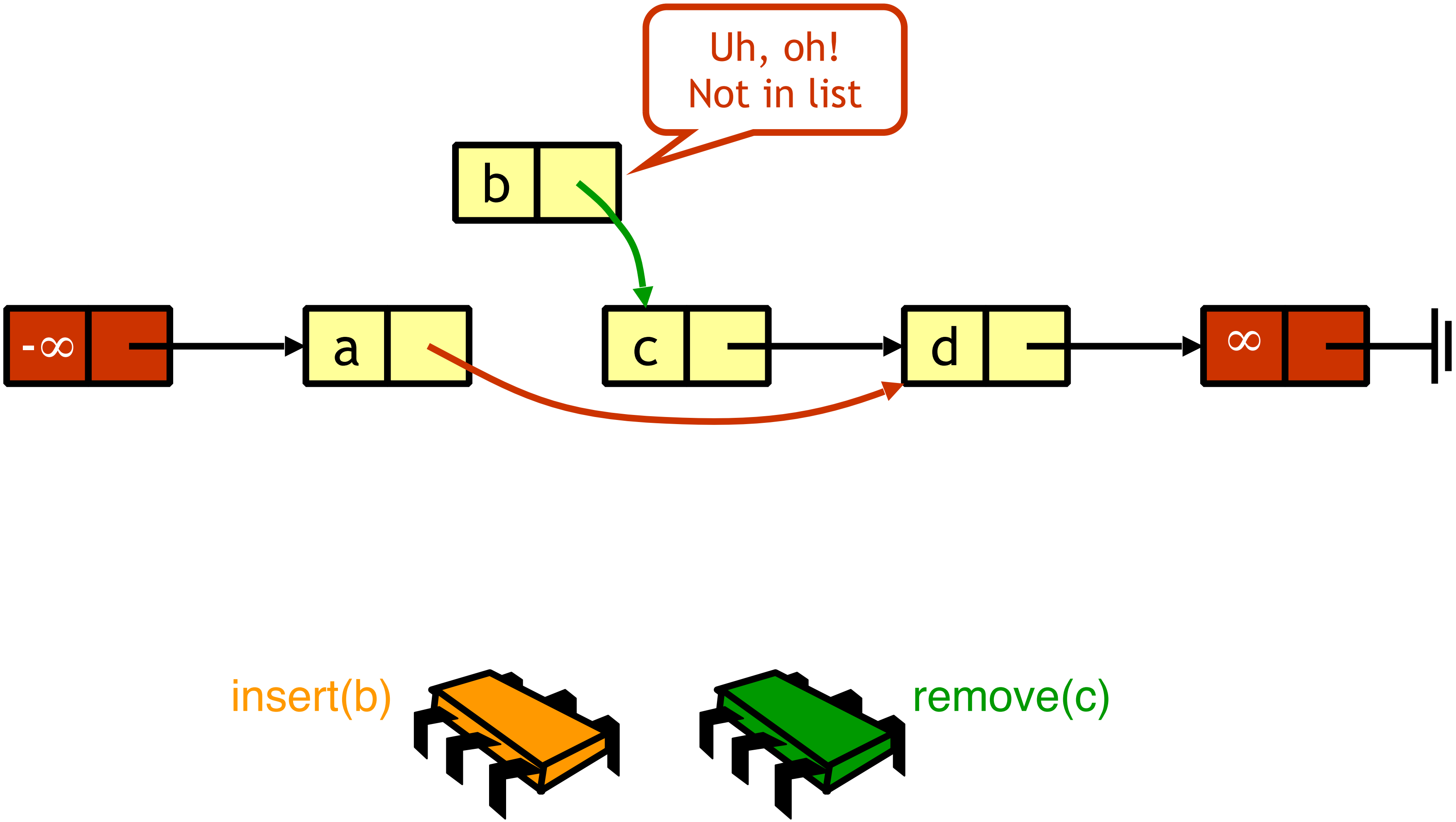
What Could Go Wrong?



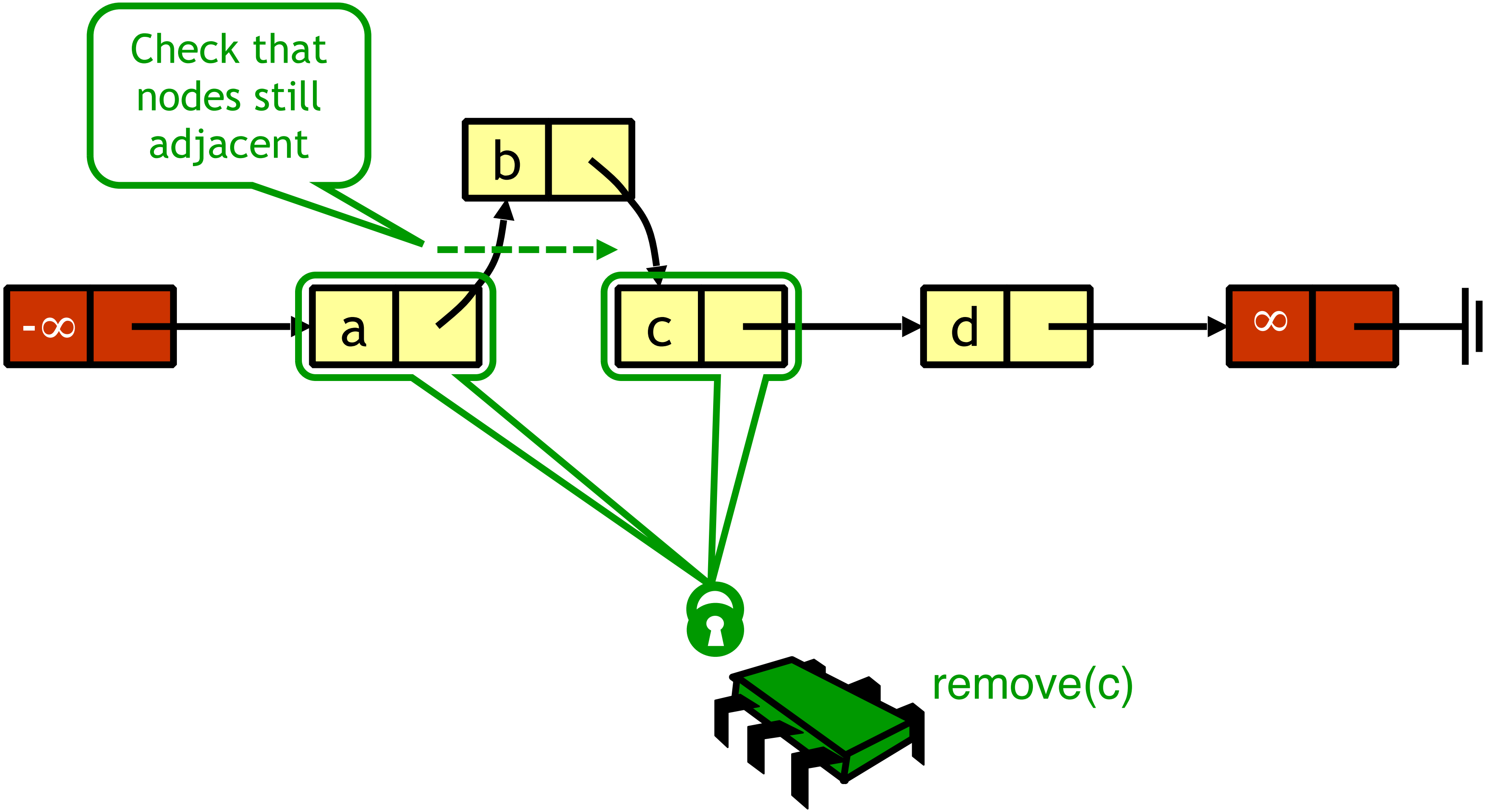
What Could Go Wrong?



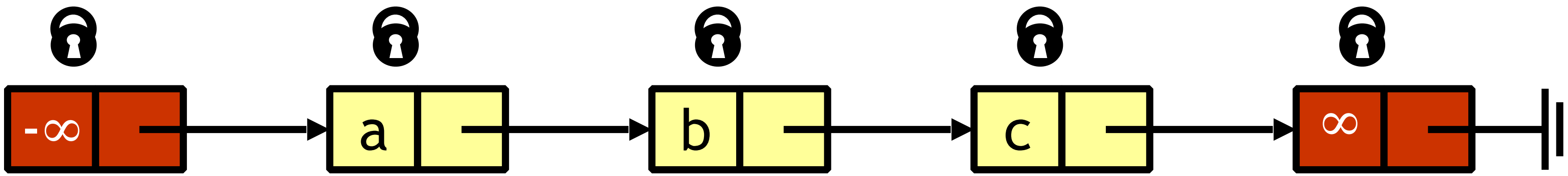
What Could Go Wrong?



What Could Go Wrong?

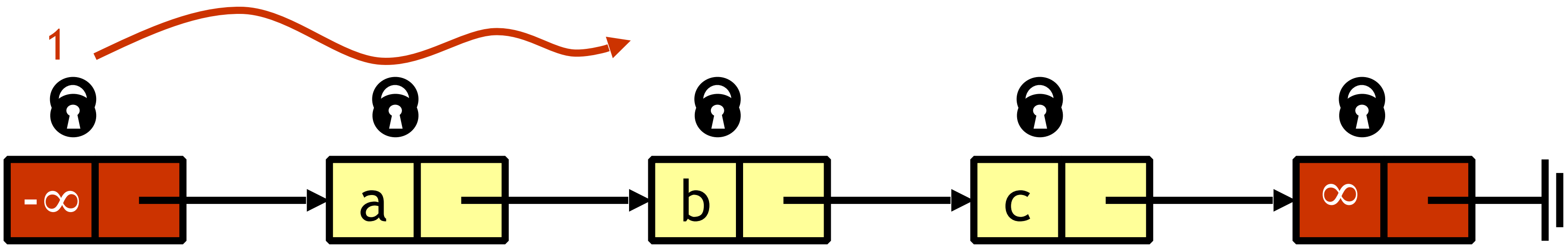


Optimistic Fine Grained



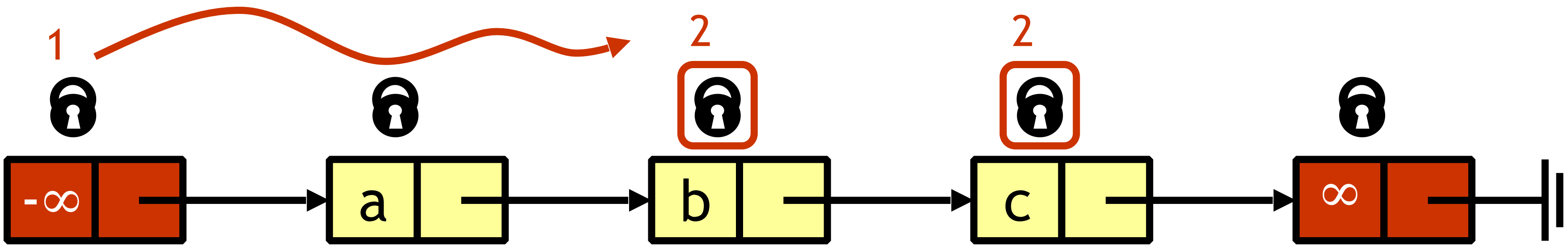
- To remove c
  - Optimistically traverse list to find c
  - Lock c.pred then lock c.curr
  - Re-Traverse list to find c and verify that c.pred precedes c.curr
  - Perform removal and release locks

Optimistic Fine Grained



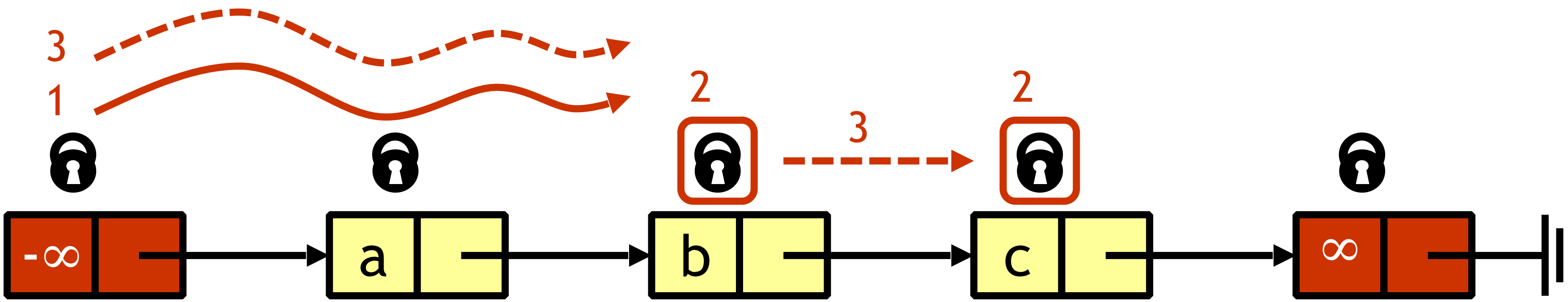
- To remove c
  - Optimistically traverse list to find c
  - Lock c.pred then lock c.curr
  - Re-Traverse list to find c and verify that c.pred precedes c.curr
  - Perform removal and release locks

Optimistic Fine Grained



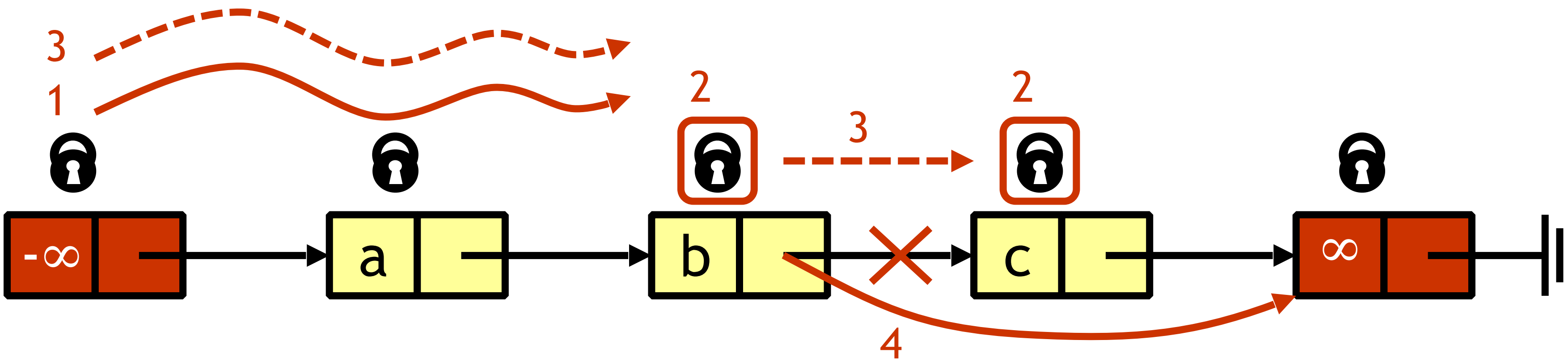
- To remove c
  - Optimistically traverse list to find c
  - Lock c.pred then lock c.curr
  - Re-Traverse list to find c and verify that c.pred precedes c.curr
  - Perform removal and release locks

Optimistic Fine Grained



- To remove c
  - Optimistically traverse list to find c
  - Lock c.pred then lock c.curr
  - Re-Traverse list to find c and verify that c.pred precedes c.curr
  - Perform removal and release locks

Optimistic Fine Grained

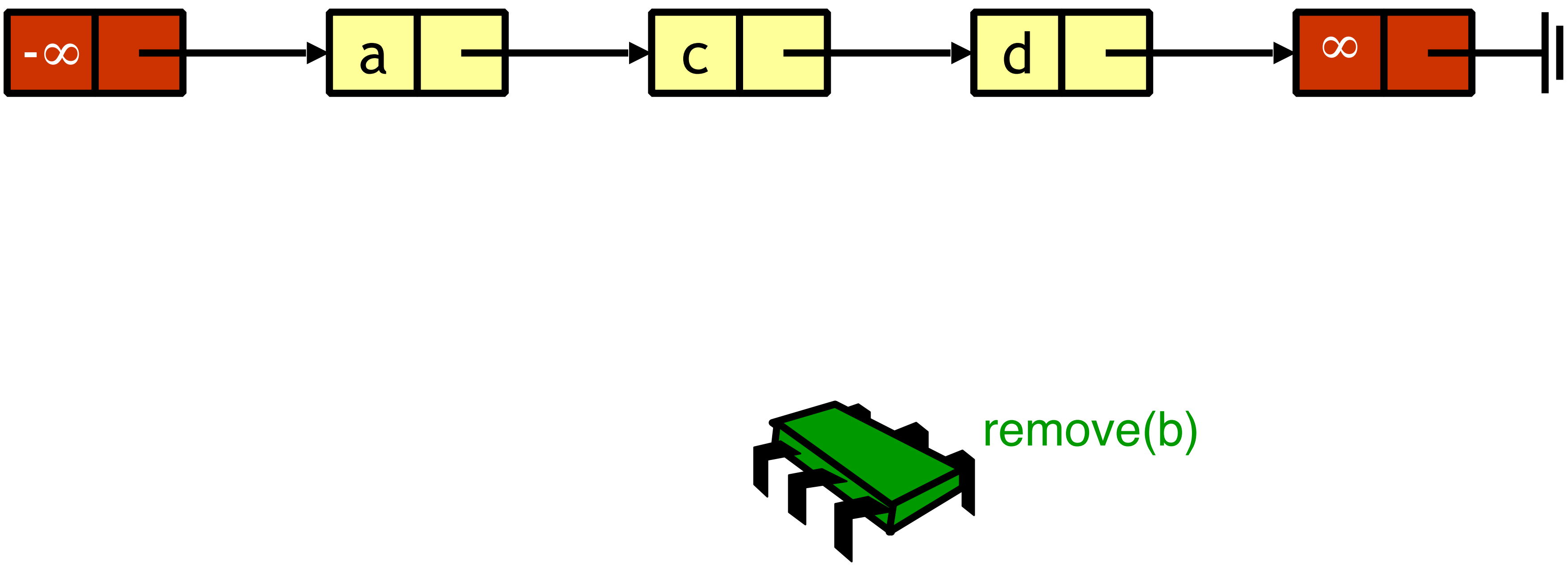


- To remove c
  - Optimistically traverse list to find c
  - Lock c.pred then lock c.curr
  - Re-Traverse list to find c and verify that c.pred precedes c.curr
  - Perform removal and release locks

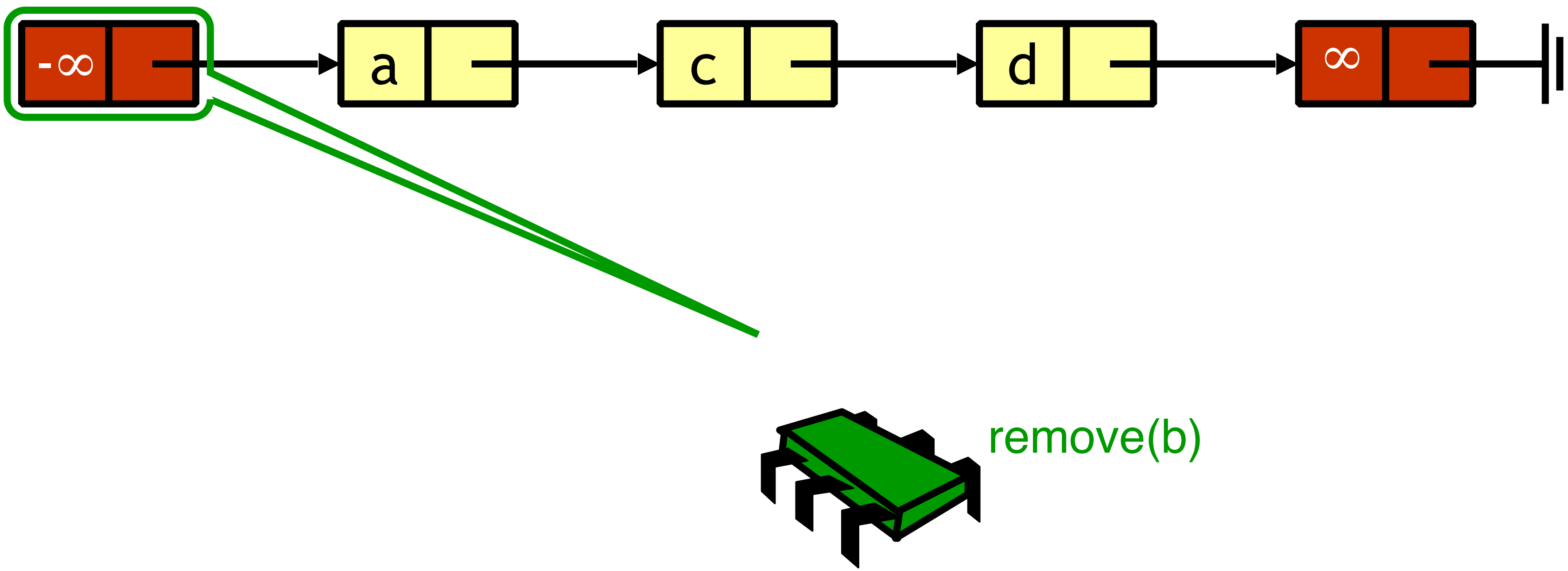
Correctness

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

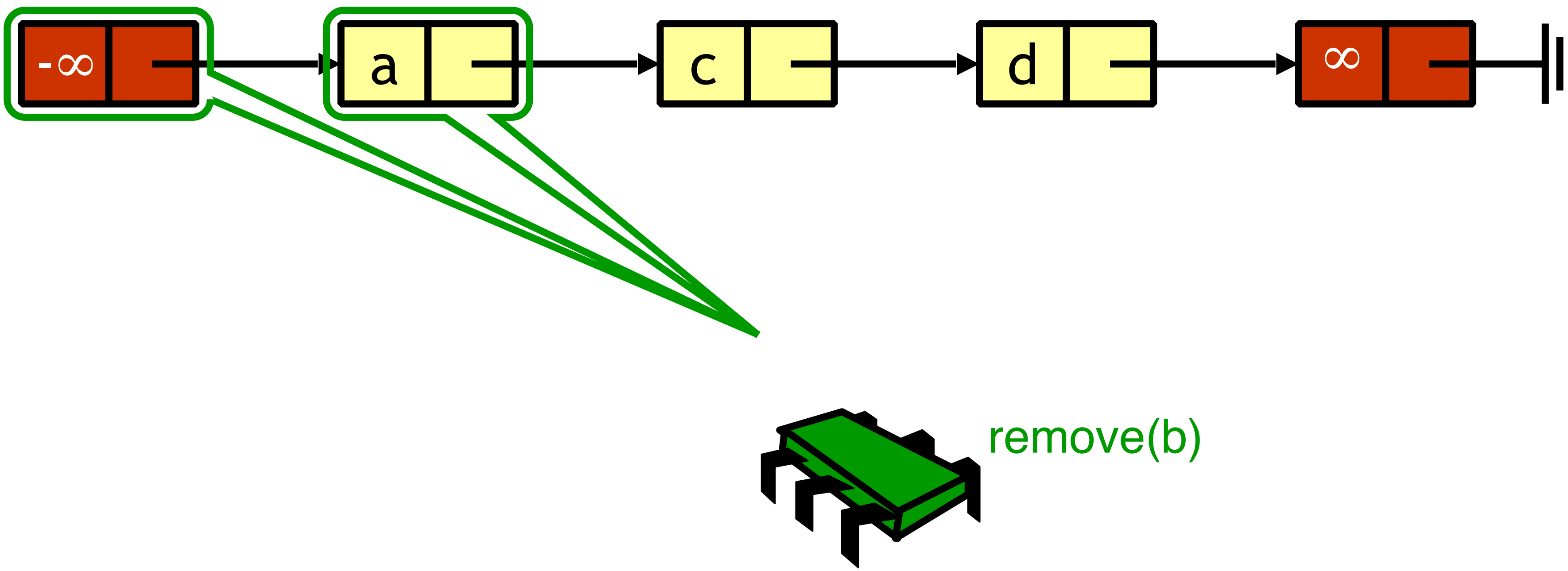
Removing an Absent Node



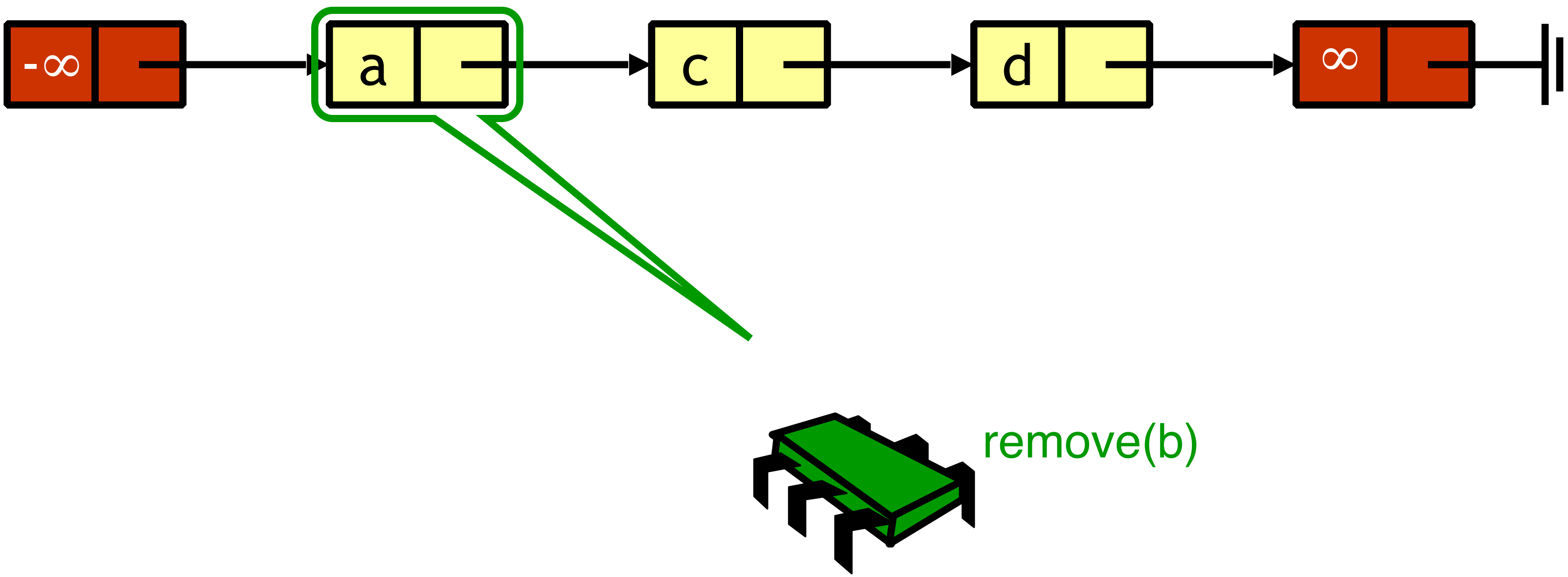
Removing an Absent Node



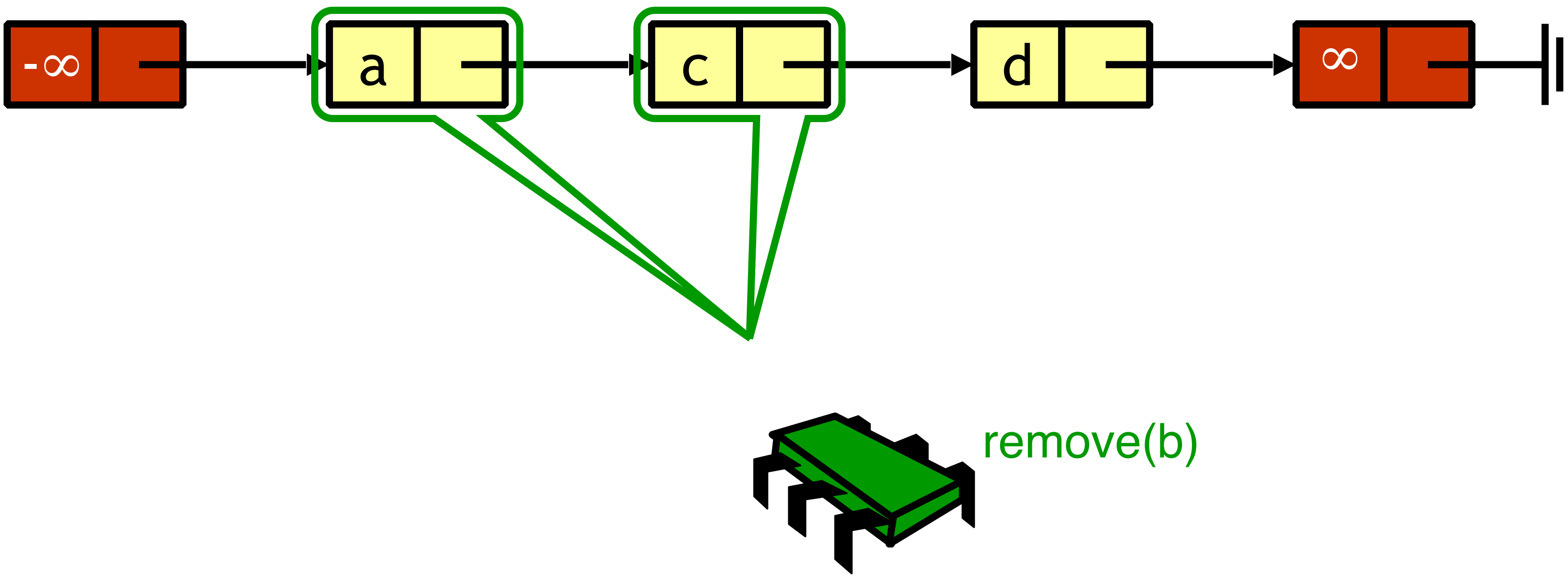
Removing an Absent Node



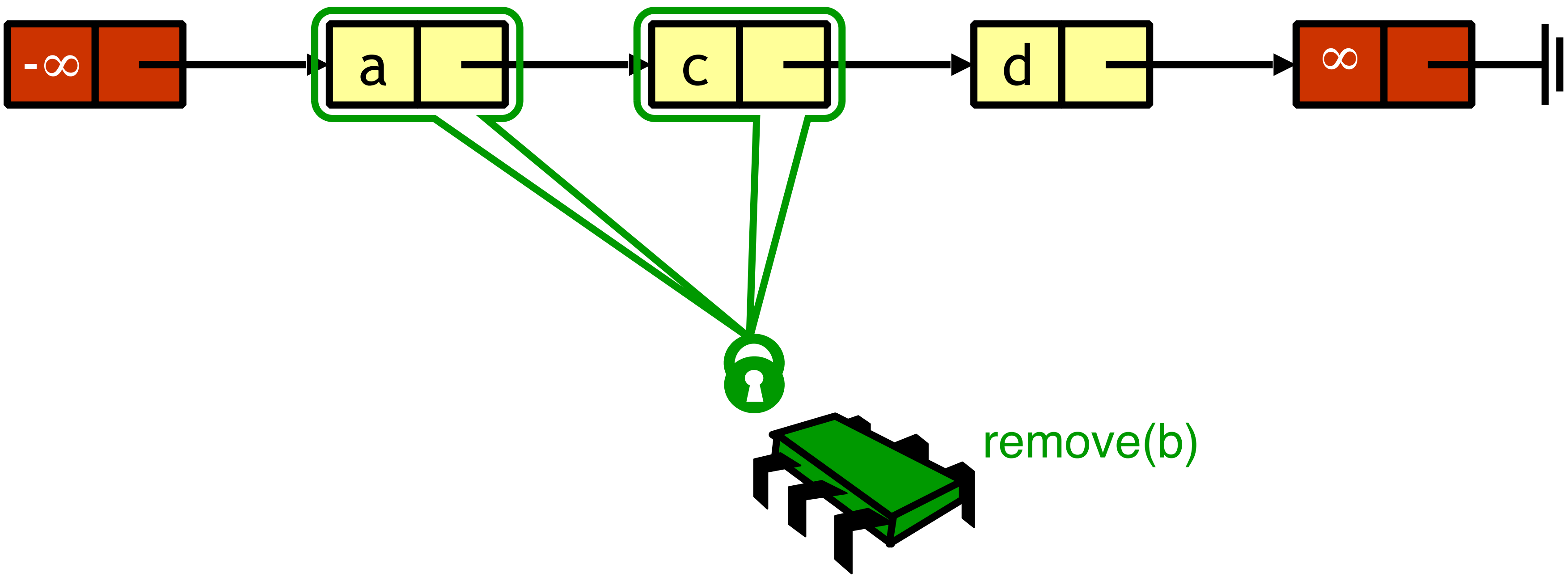
Removing an Absent Node



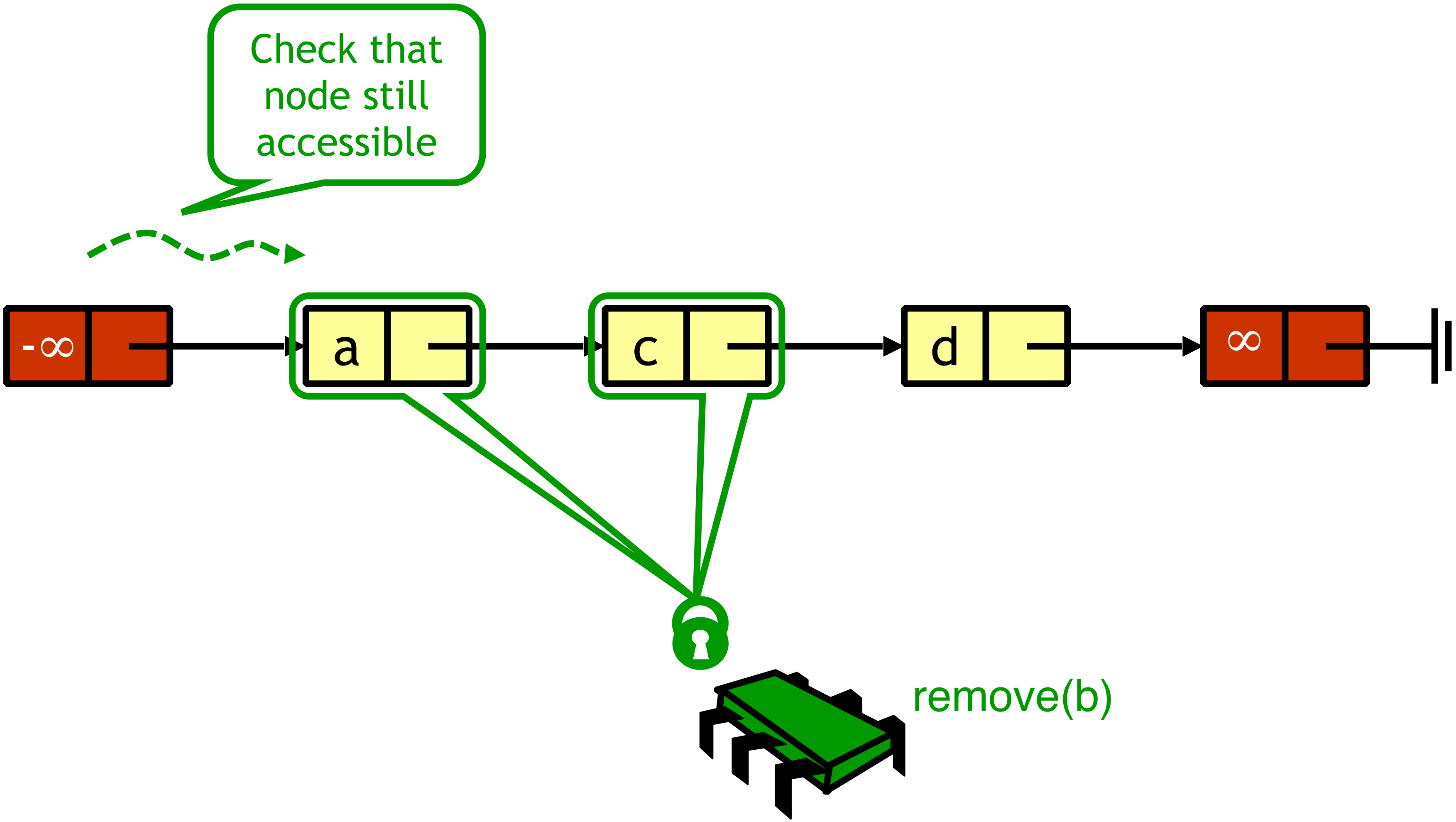
Removing an Absent Node



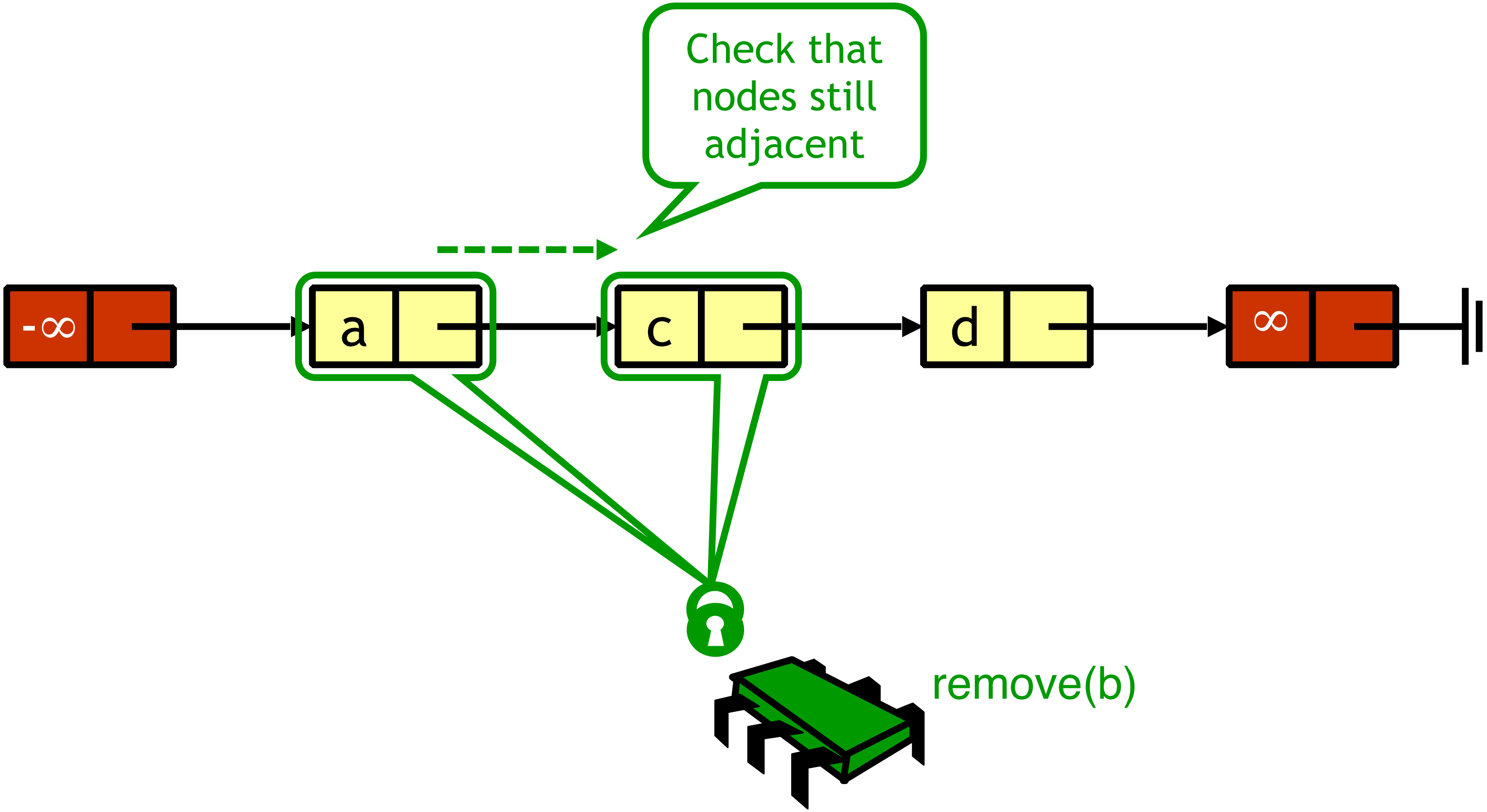
Removing an Absent Node



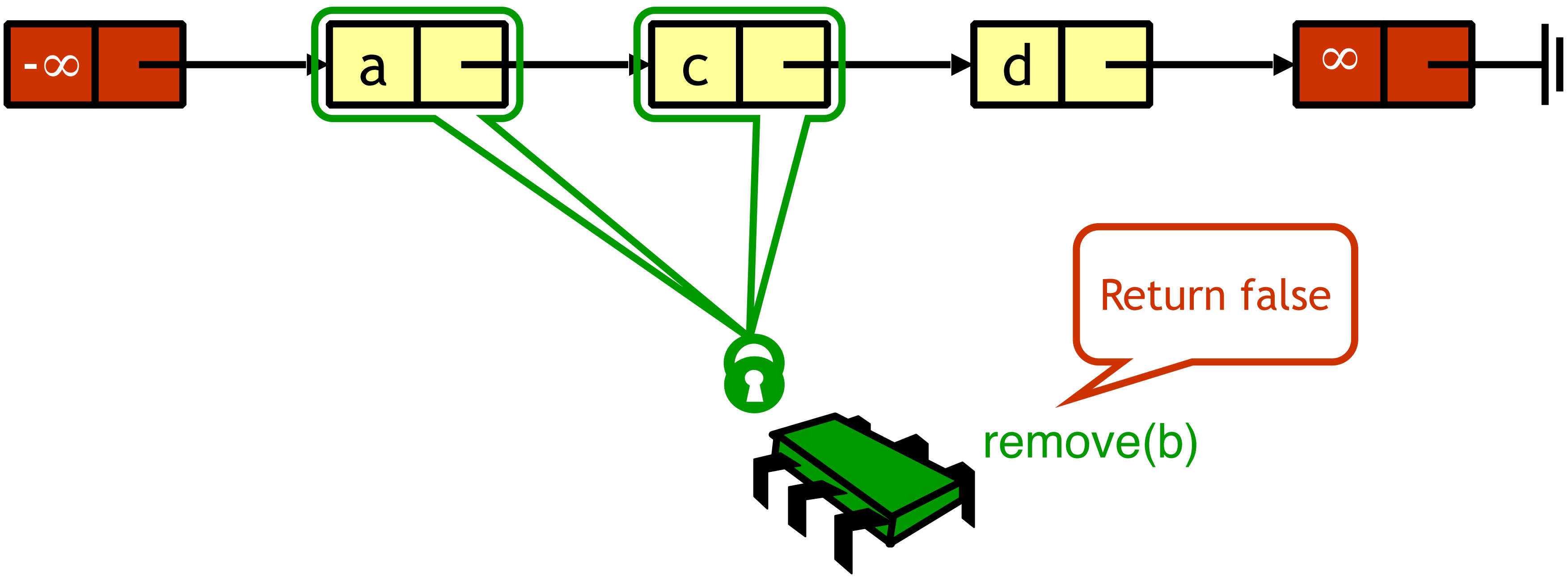
Removing an Absent Node



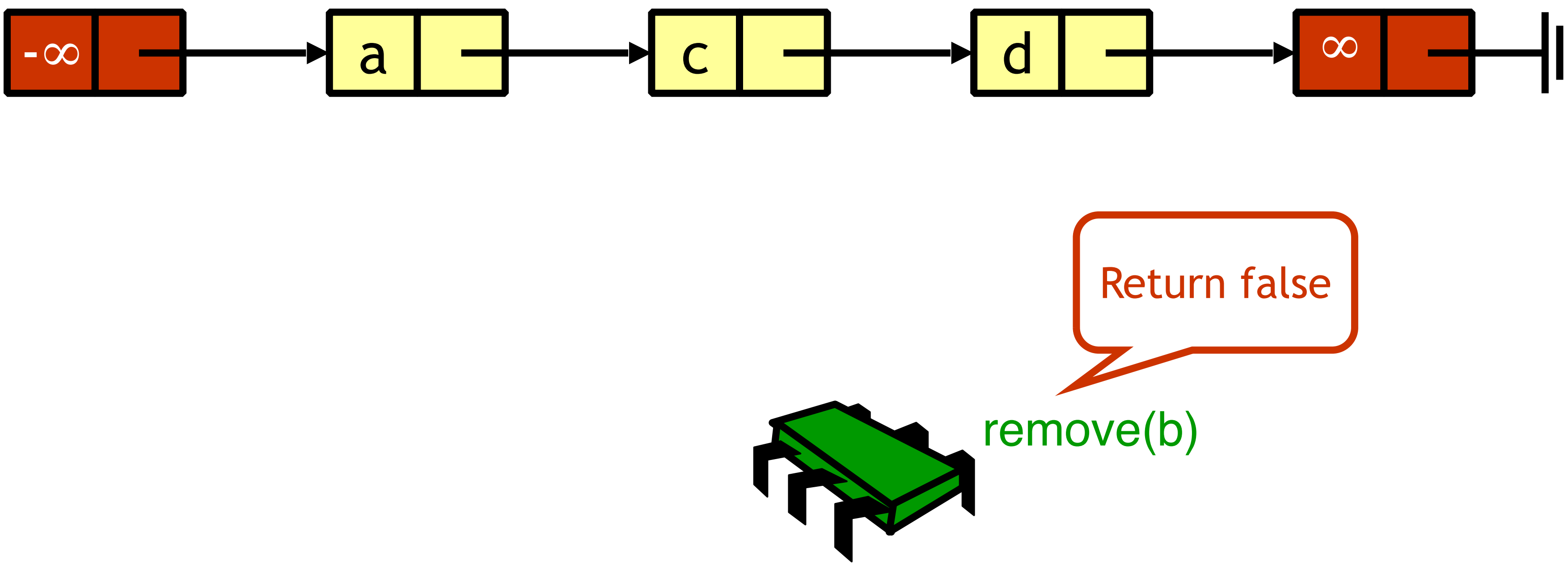
Removing an Absent Node



Removing an Absent Node



Removing an Absent Node



Correctness

- If
  - Nodes a and c both locked
  - Node a still accessible
  - Node c still successor to a
- Then
  - Neither has been deleted
  - No thread can add b after a while a is locked
  - OK to 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;
}
```

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;
}
```

Start 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;
}
```

Current node next?

Validation

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

Otherwise move on

Validation

```
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 not reachable

Remove: Searching

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

Remove: Searching

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

Search key

Remove: Searching

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

Retry on synchronization conflict

Remove: Searching

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

Examine predecessor and current nodes

Remove: Searching

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

Search by key

Remove: Searching

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

Stop if we find object

Remove: Searching

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

Move along

On Exit from Loop

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

Remove

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

Remove

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

Always unlock

Remove

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

Lock both nodes

Remove

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

Check for synchronization conflicts

Remove

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

Object found,
remove node

Remove

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

Object not found

Summary: Optimistic List

- Wait-free traversal
  - May traverse removed nodes
  - Must have non-interference (natural in languages with GC like Java)
- Limited hotspots
  - Only at locked add(), remove(), contains() destination locations, not traversals
- But two traversals
  - Yet traversals are wait-free

So Far, So Good

- Much less lock acquisition/release
  - Performance
  - Concurrency
- Problems
  - Need to traverse list twice
  - contains() acquires locks
    - Most common method call (90% in many applications)
- Optimistic works if
  - Cost of scanning twice without locks <
    cost of scanning once with locks

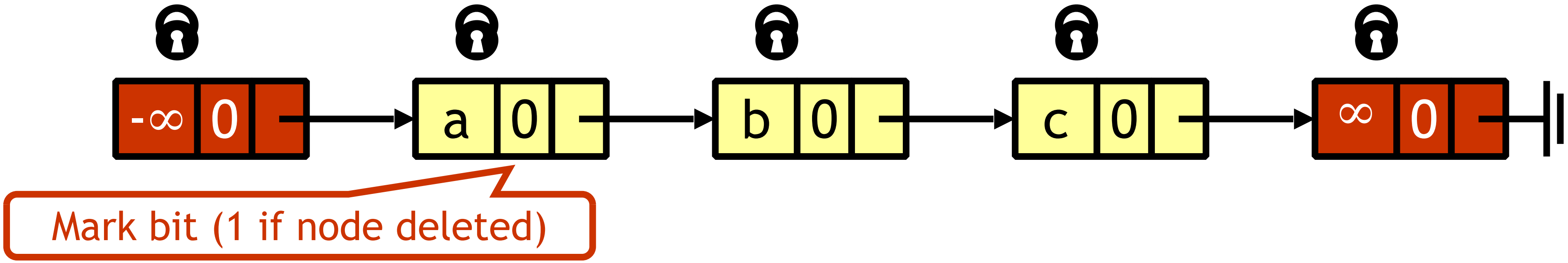
Lazy List

- Like optimistic, except
  - Scan once
  - contains() never locks
- Key insight
  - Removing nodes causes trouble
  - Do it “lazily”

Lazy List

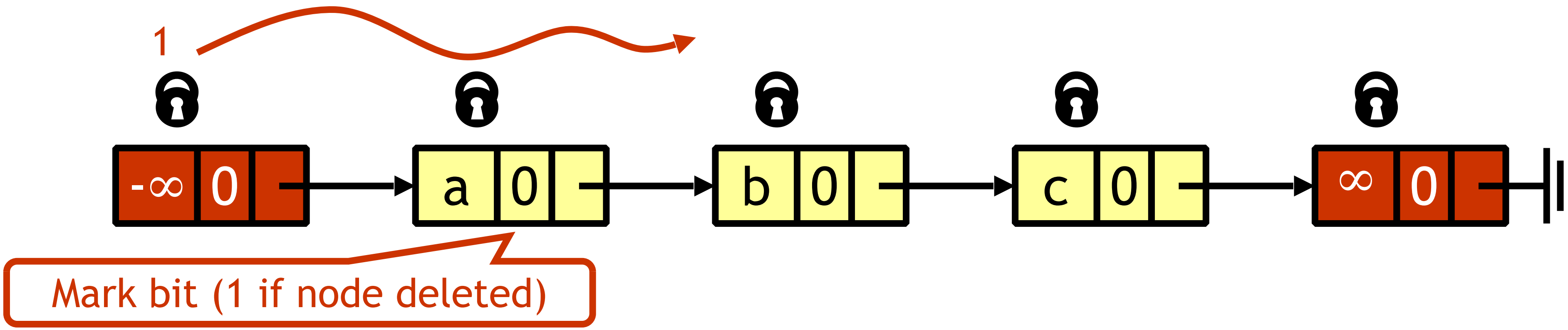
- Remove Method
  - Scans list (as before)
  - Locks predecessor & current (as before)
- Logical delete
  - Marks current node as removed (new!)
  - Use additional mark bit in node
- Physical delete
  - Redirects predecessor's next (as before)

Lazy Removal



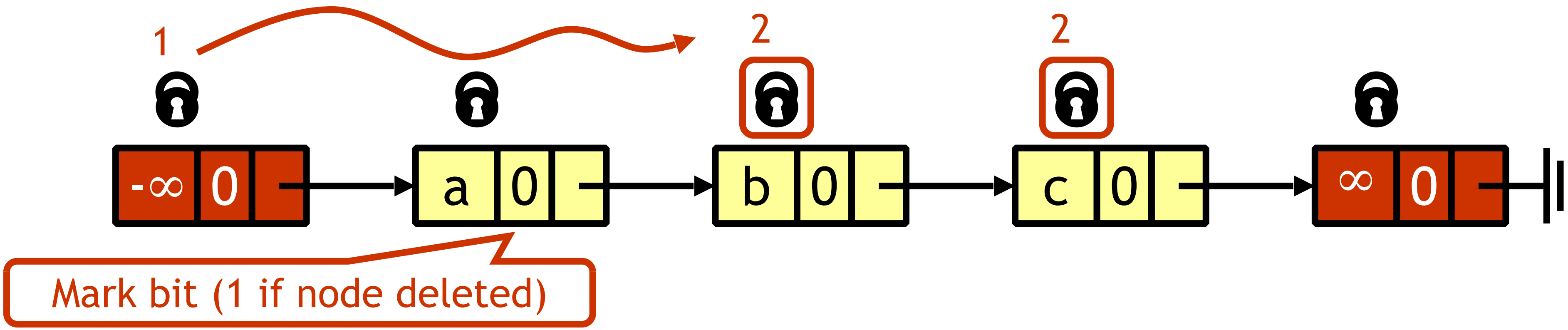
- To remove c
  - Optimistically traverse list to find c
  - Lock c.pred then lock c.curr
  - Verify marks and that c.pred precedes c.curr
  - Set mark bit (logical removal)
  - Perform physical removal and release locks

Lazy Removal



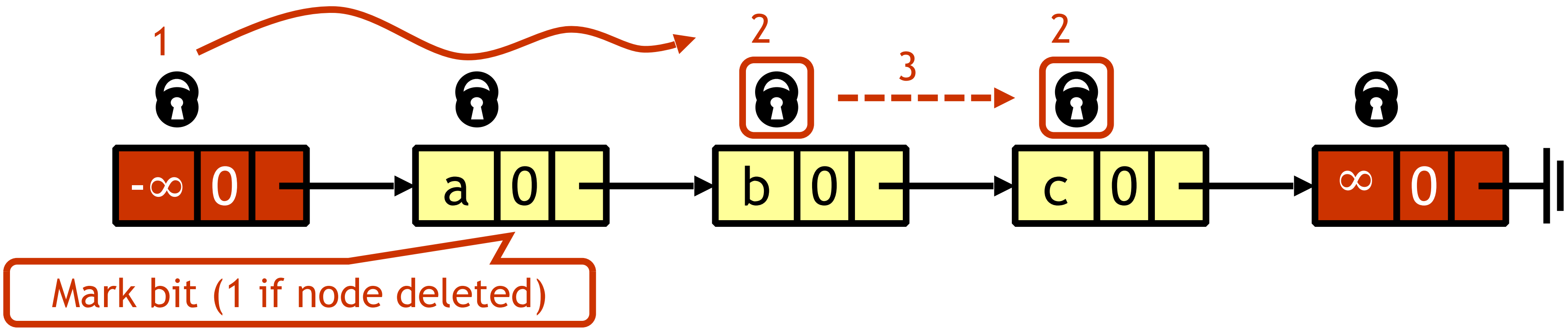
- To remove c
  - Optimistically traverse list to find c
  - Lock c.pred then lock c.curr
  - Verify marks and that c.pred precedes c.curr
  - Set mark bit (logical removal)
  - Perform physical removal and release locks

Lazy Removal



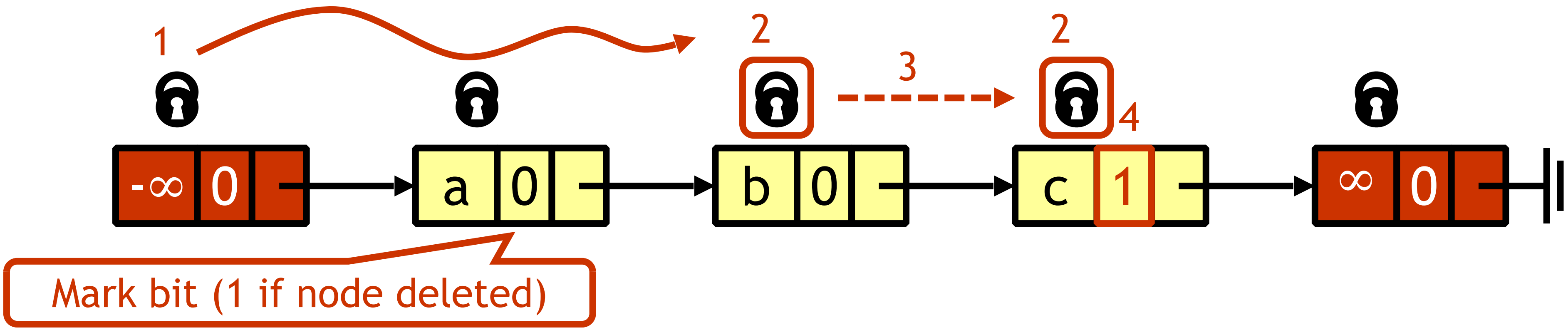
- To remove c
  - Optimistically traverse list to find c
  - Lock c.pred then lock c.curr
  - Verify marks and that c.pred precedes c.curr
  - Set mark bit (logical removal)
  - Perform physical removal and release locks

Lazy Removal



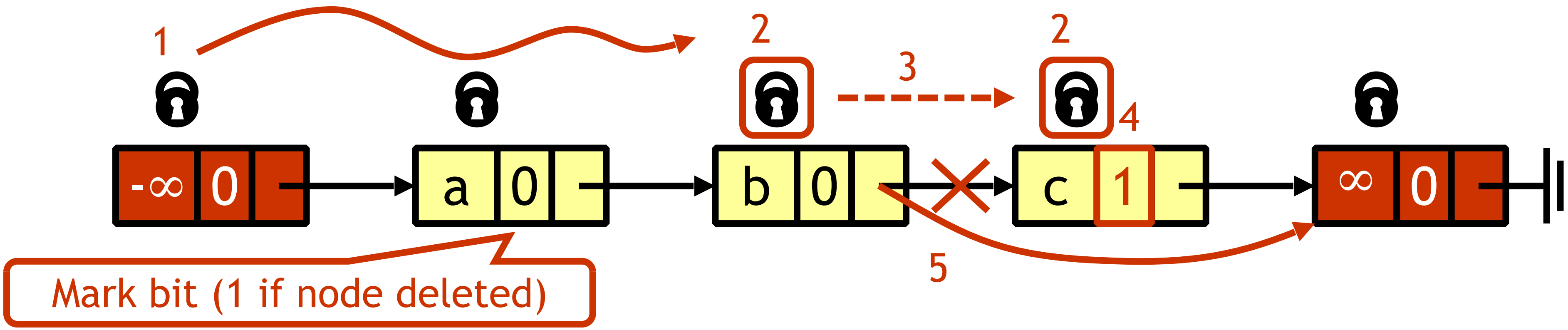
- To remove c
  - Optimistically traverse list to find c
  - Lock c.pred then lock c.curr
  - Verify marks and that c.pred precedes c.curr
  - Set mark bit (logical removal)
  - Perform physical removal and release locks

Lazy Removal



- To remove c
  - Optimistically traverse list to find c
  - Lock c.pred then lock c.curr
  - Verify marks and that c.pred precedes c.curr
  - Set mark bit (logical removal)
  - Perform physical removal and release locks

Lazy Removal

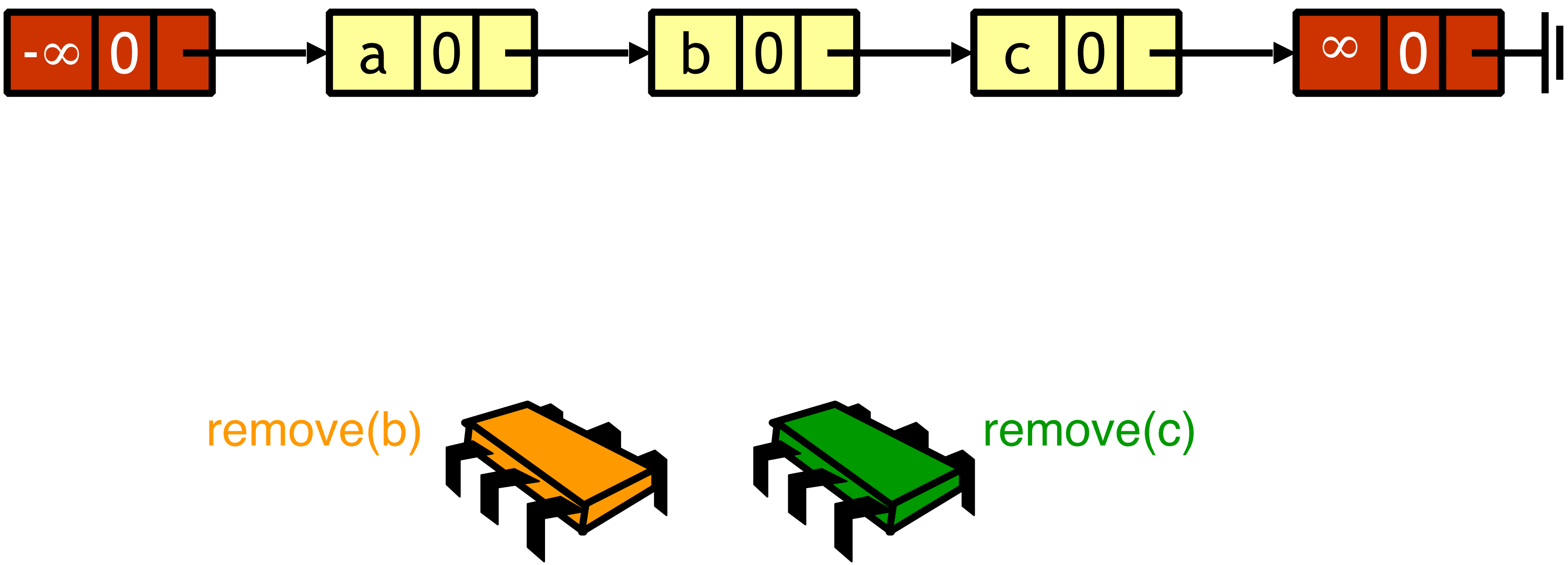


- To remove c
  - Optimistically traverse list to find c
  - Lock c.pred then lock c.curr
  - Verify marks and that c.pred precedes c.curr
  - Set mark bit (logical removal)
  - Perform physical removal and release locks

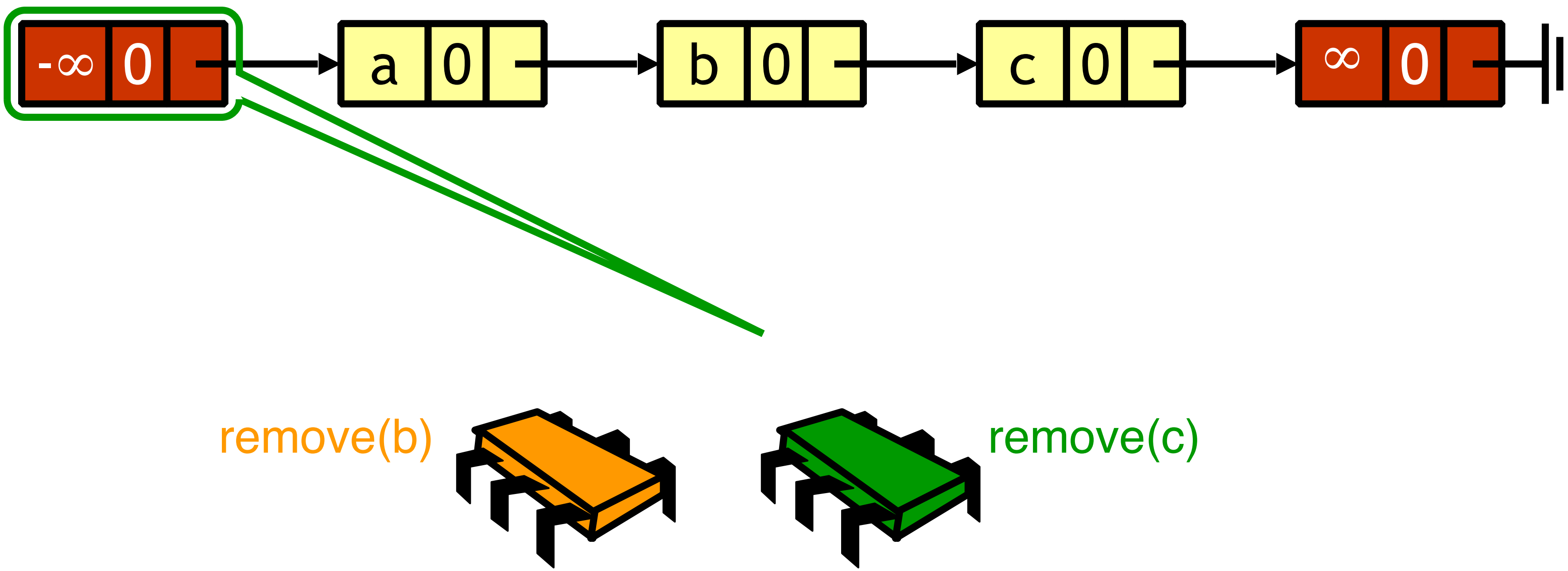
Lazy List

- All Methods
  - Scan through locked and marked nodes
  - Removing a node does not slow down other method calls…
  - Must still lock pred and curr nodes
- Validation
  - No need to rescan list!
  - Check that pred is not marked
  - Check that curr is not marked
  - Check that pred points to curr

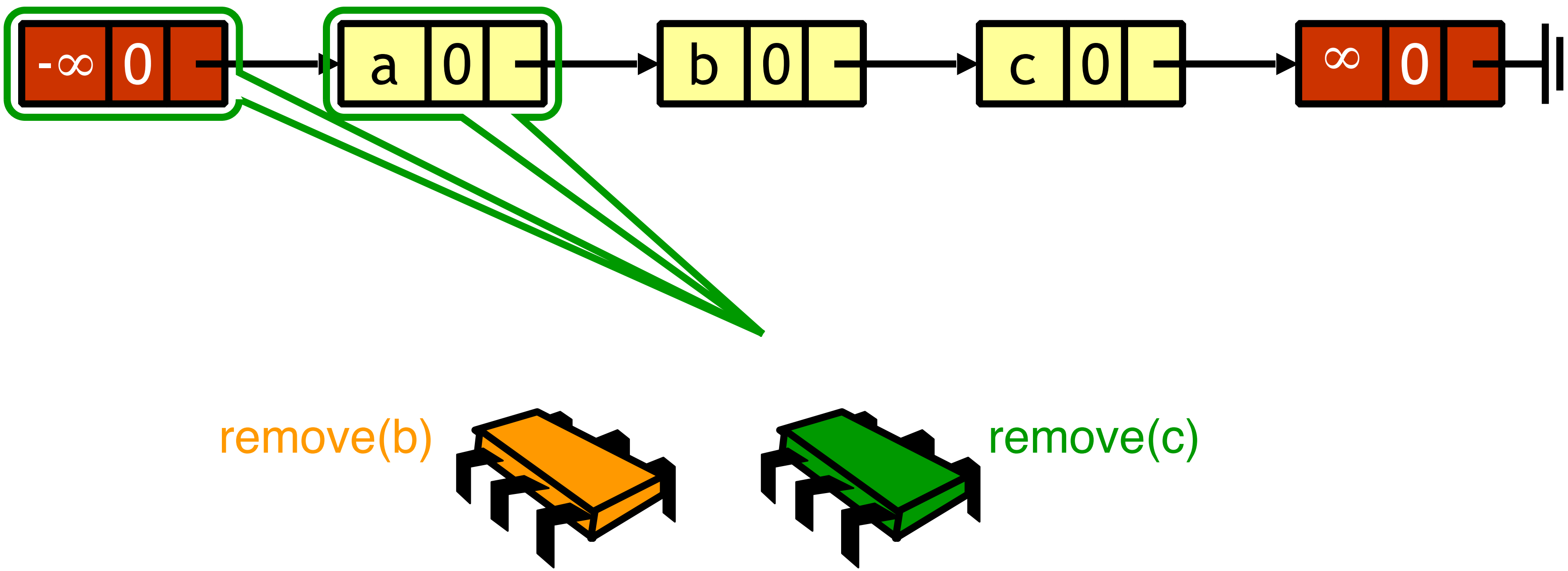
What Could Go Wrong?



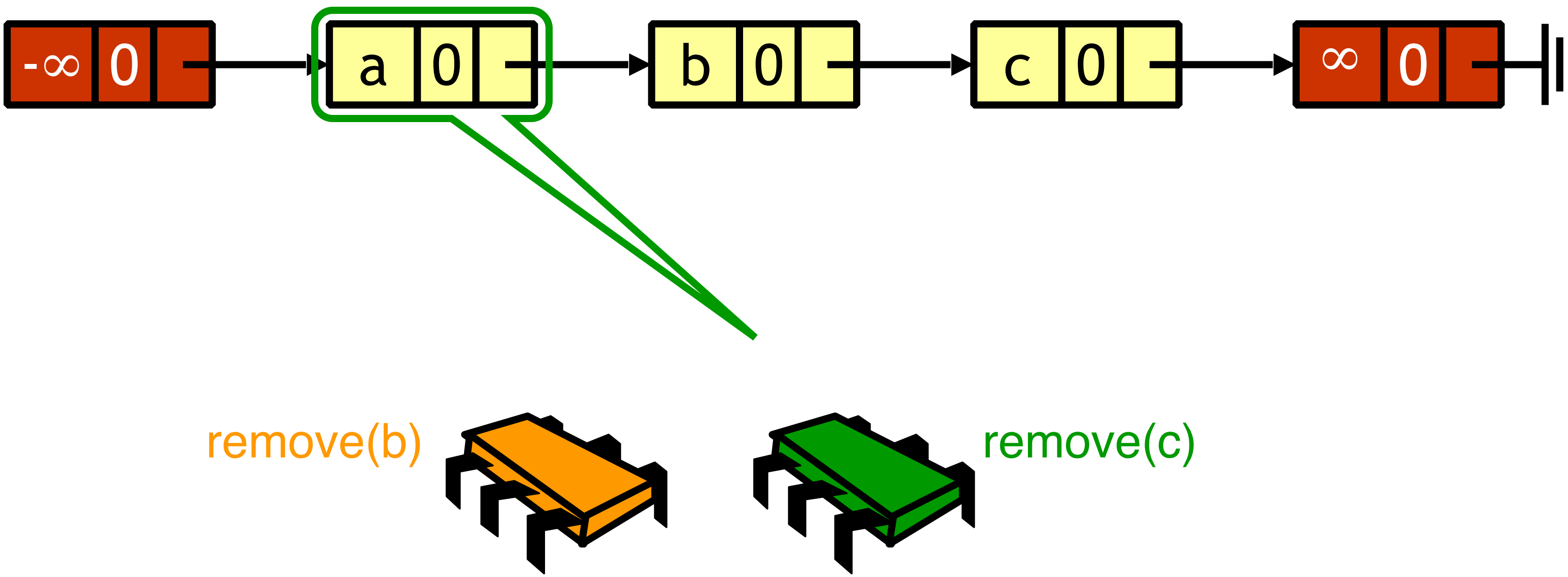
What Could Go Wrong?



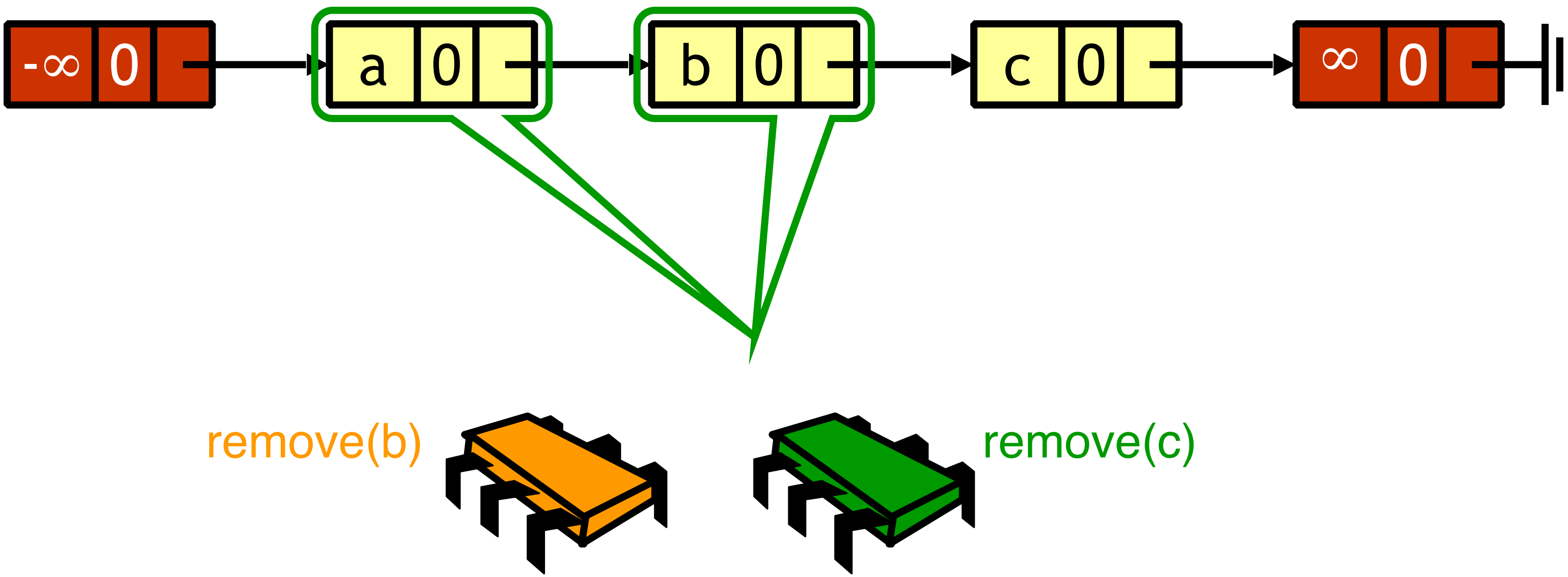
What Could Go Wrong?



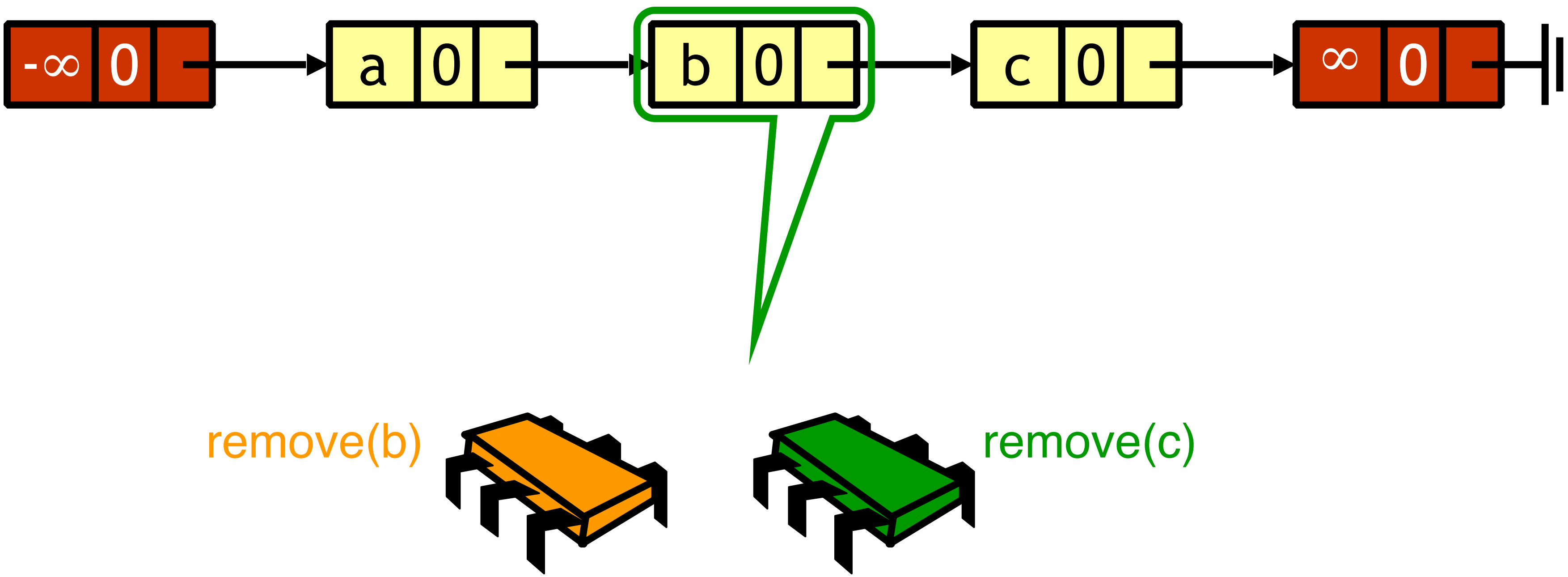
What Could Go Wrong?



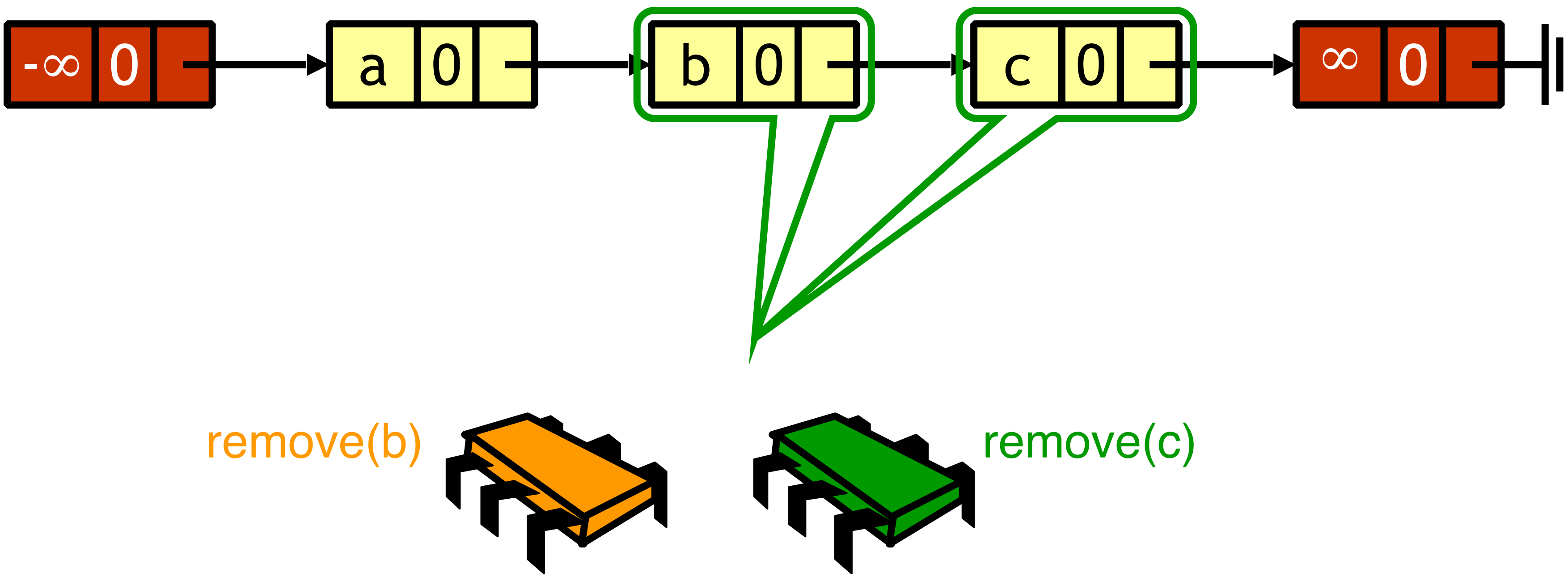
What Could Go Wrong?



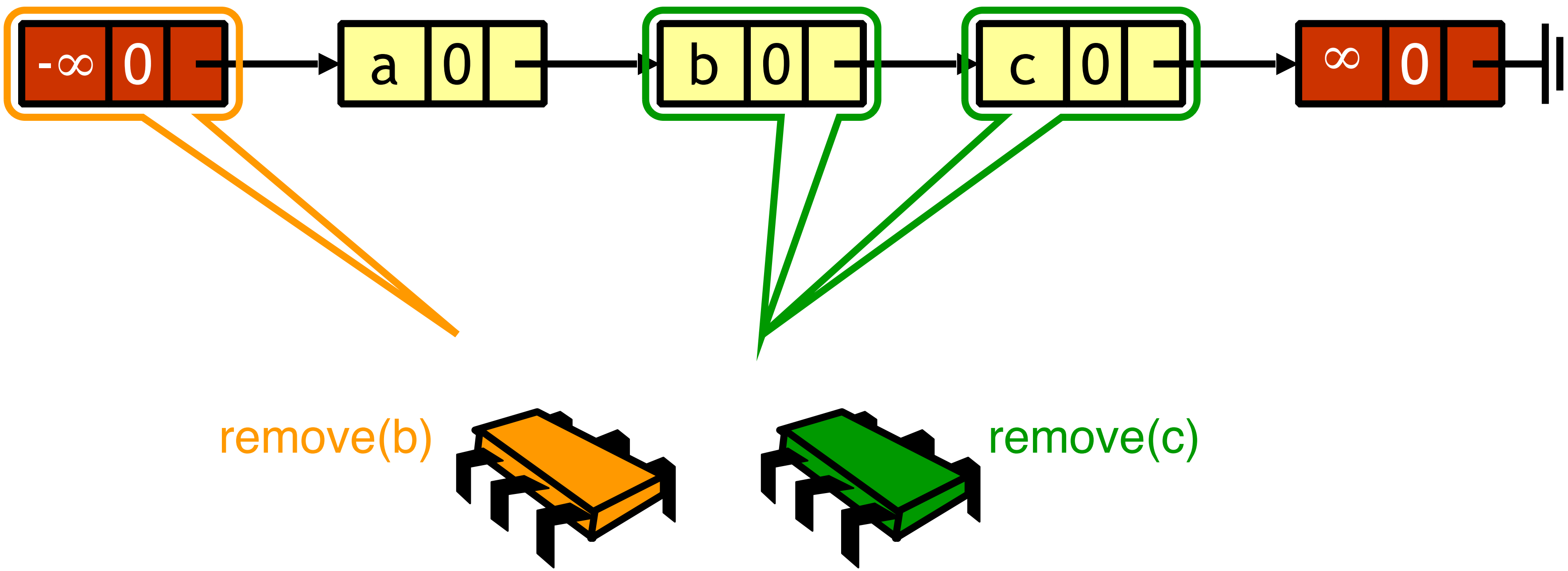
What Could Go Wrong?



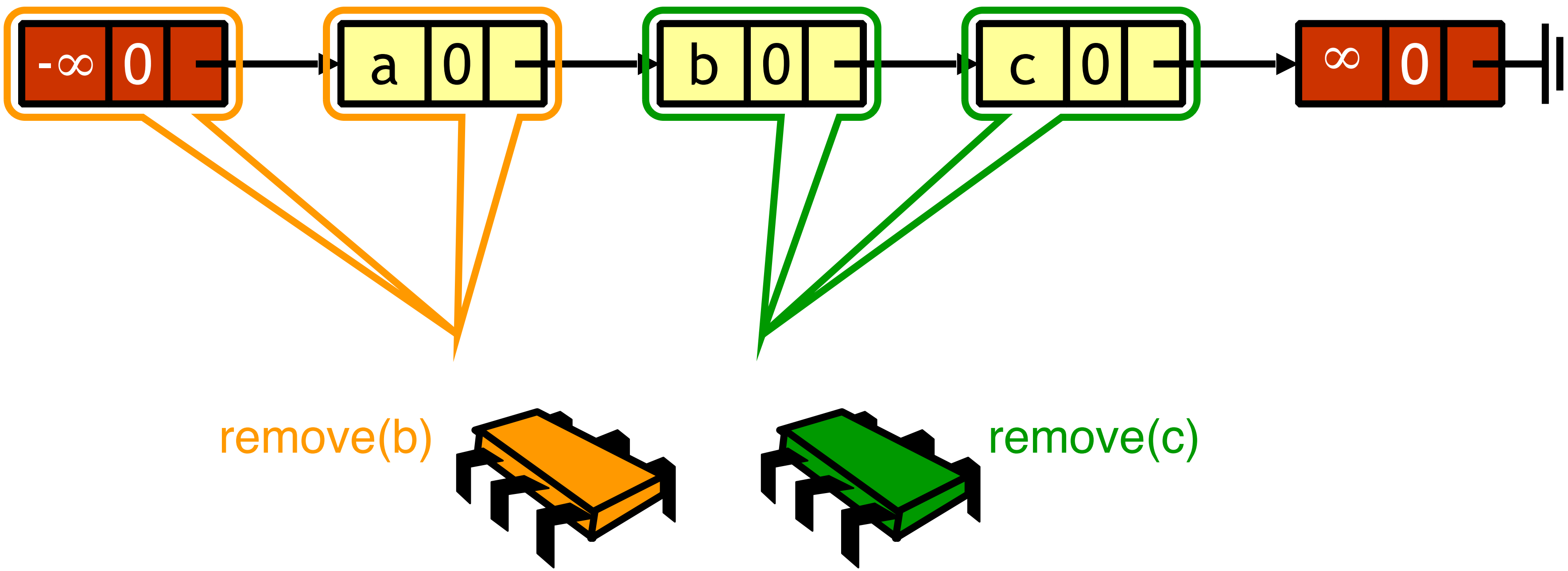
What Could Go Wrong?



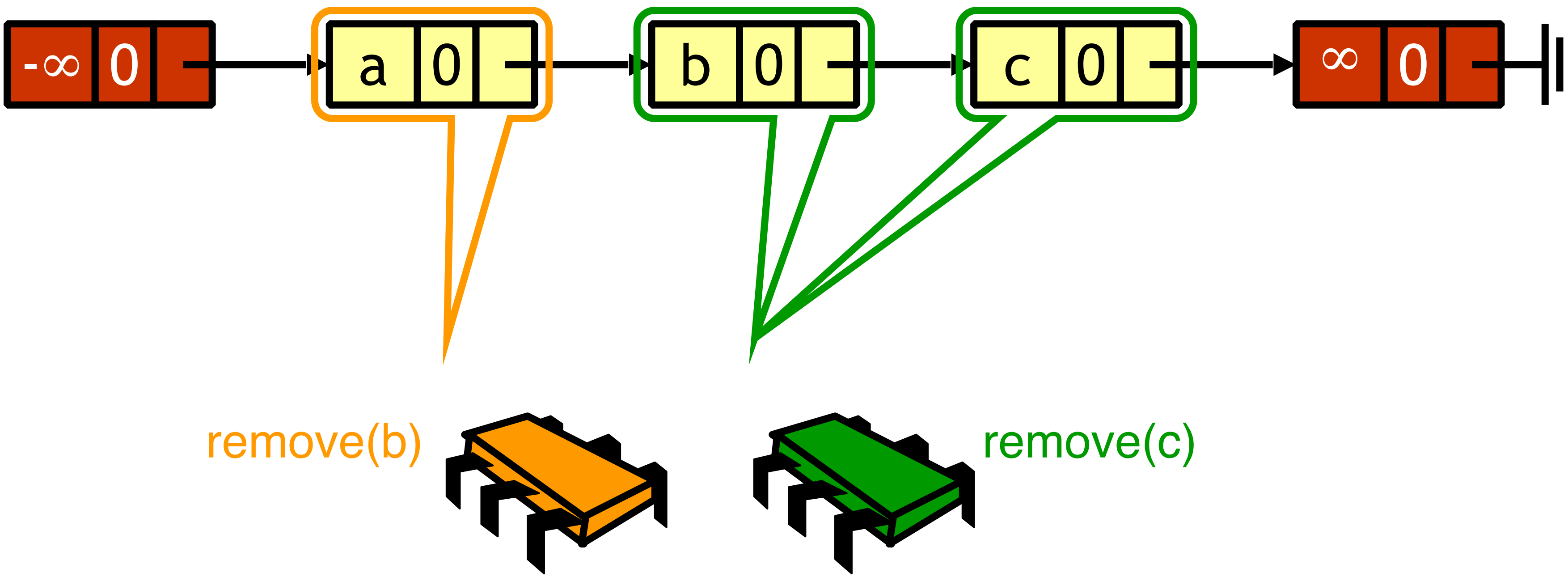
What Could Go Wrong?



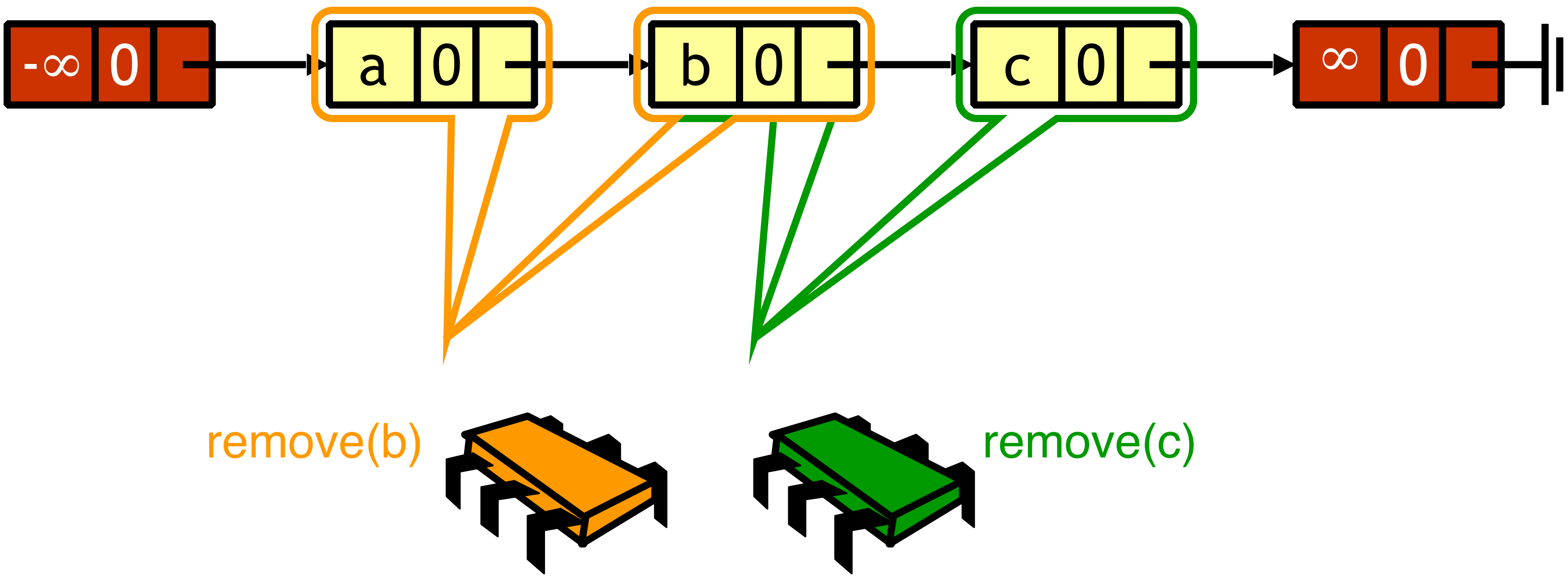
What Could Go Wrong?



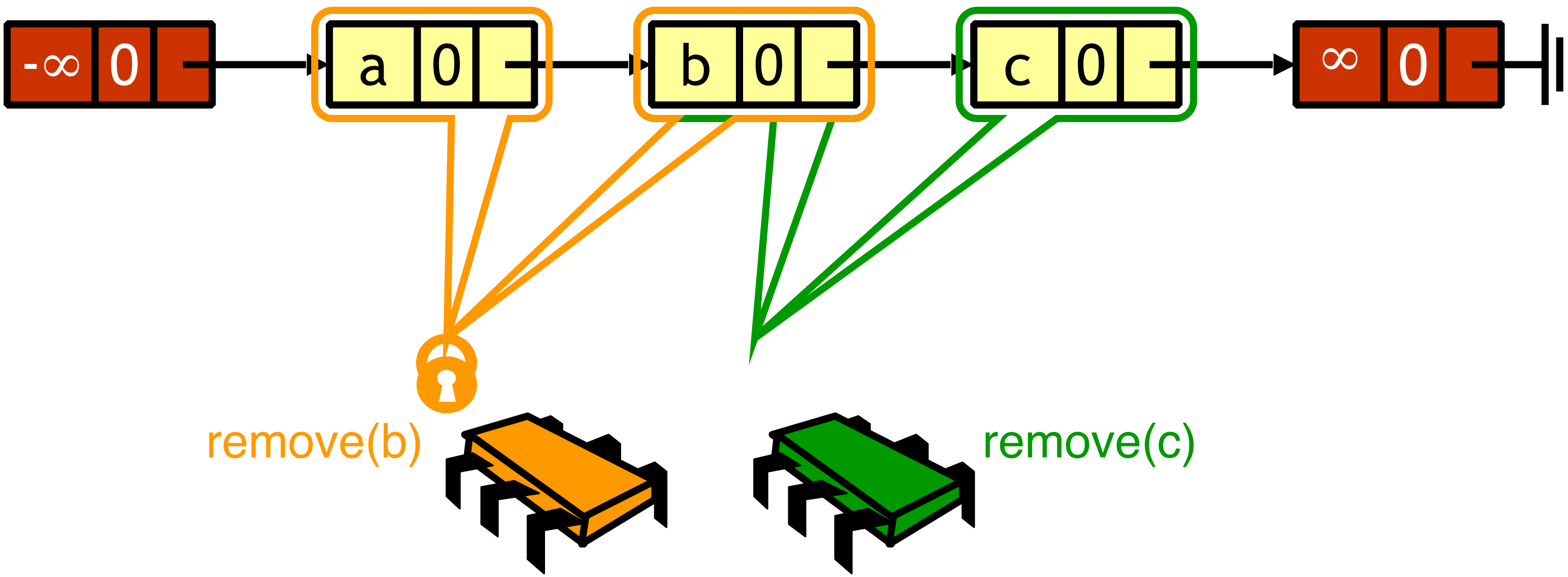
What Could Go Wrong?



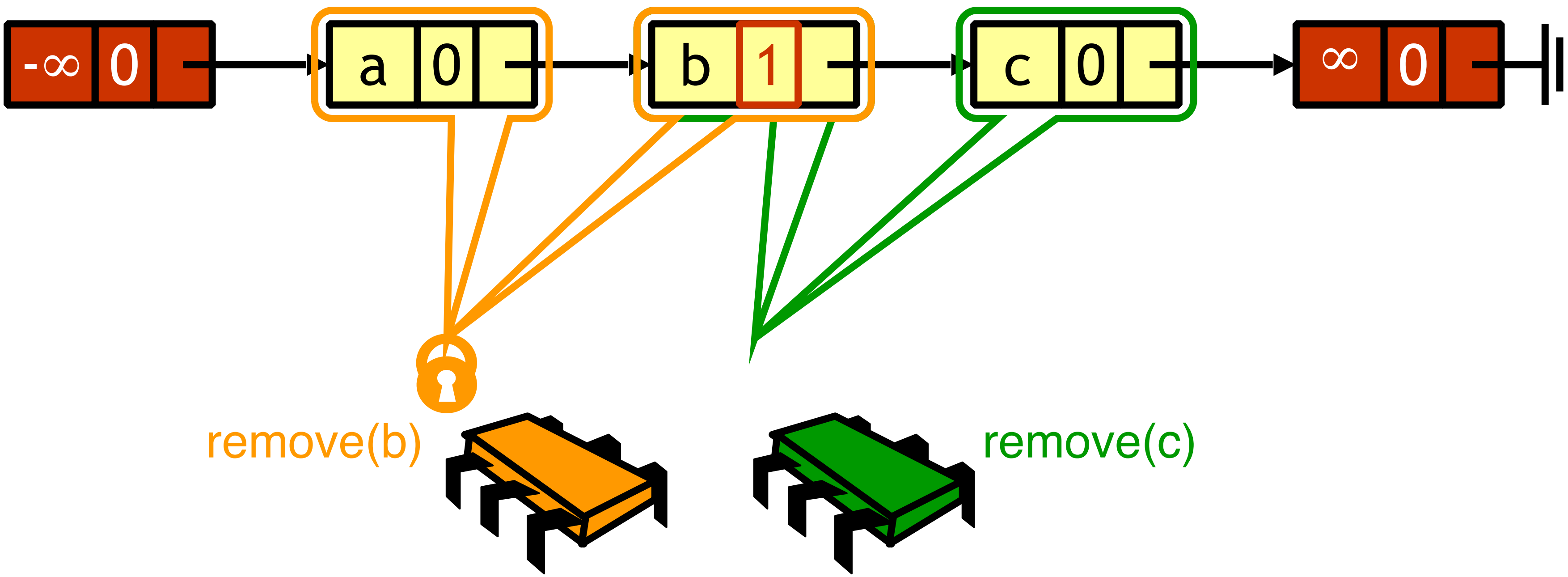
What Could Go Wrong?



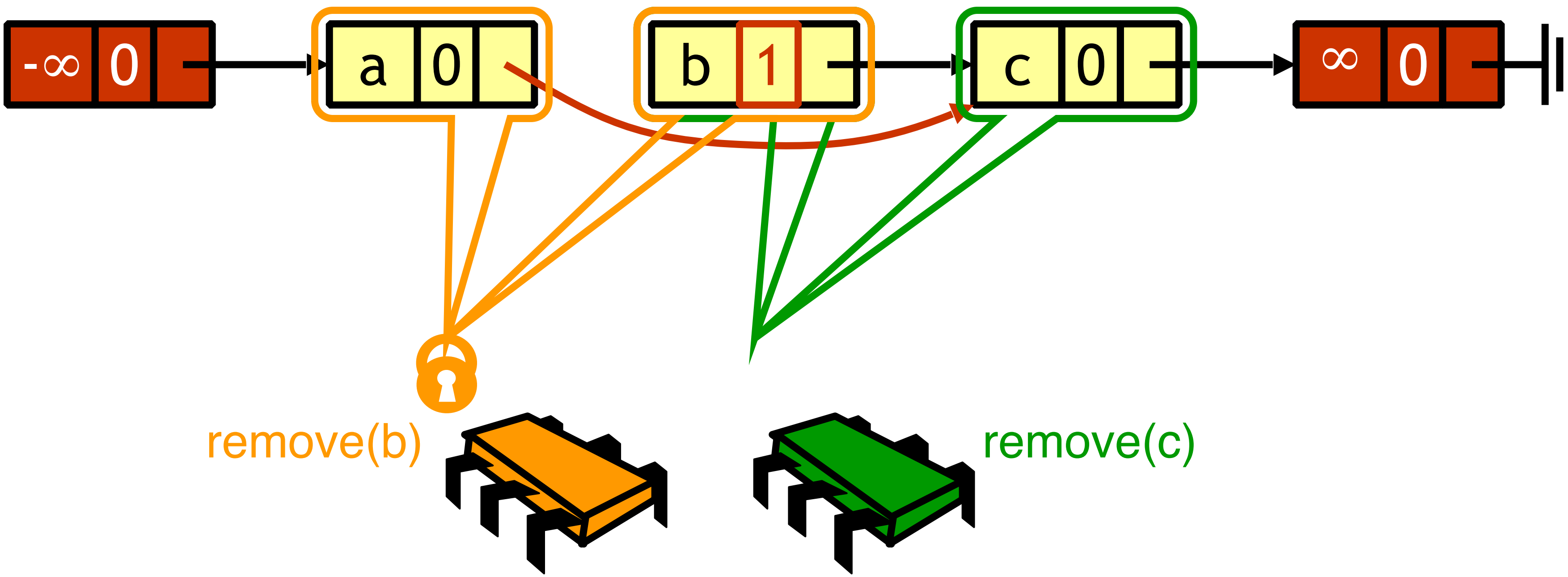
What Could Go Wrong?



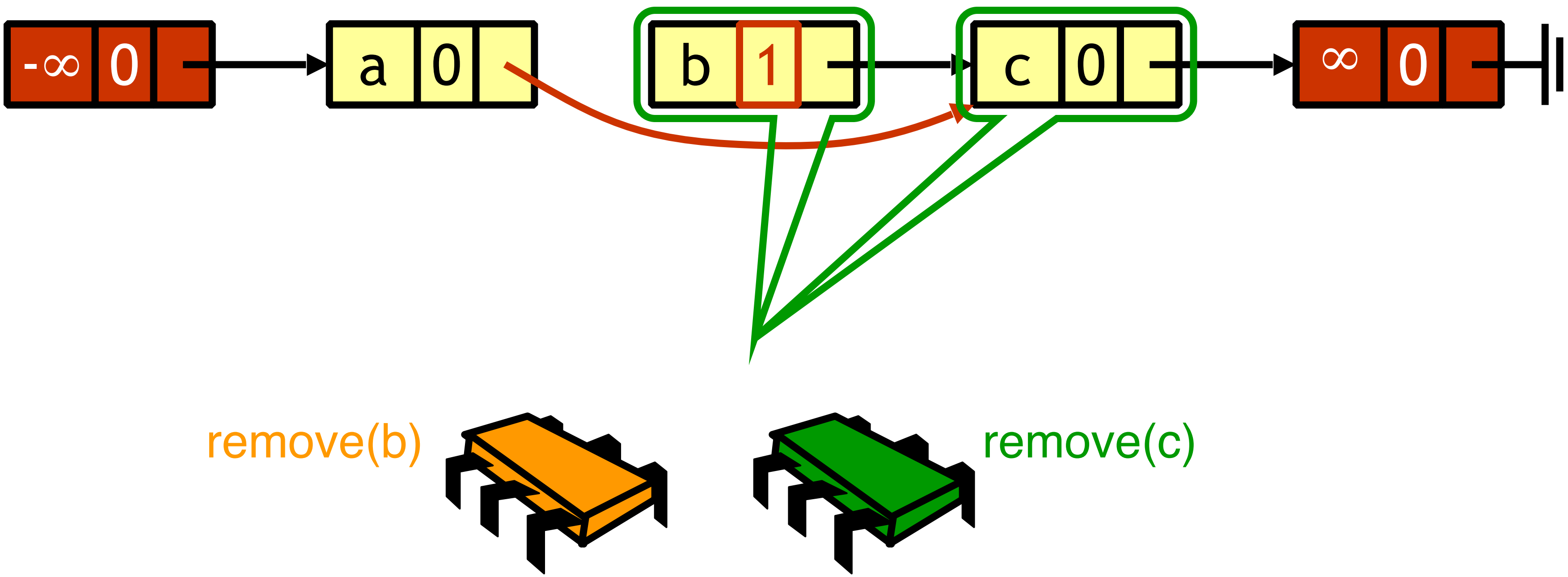
What Could Go Wrong?



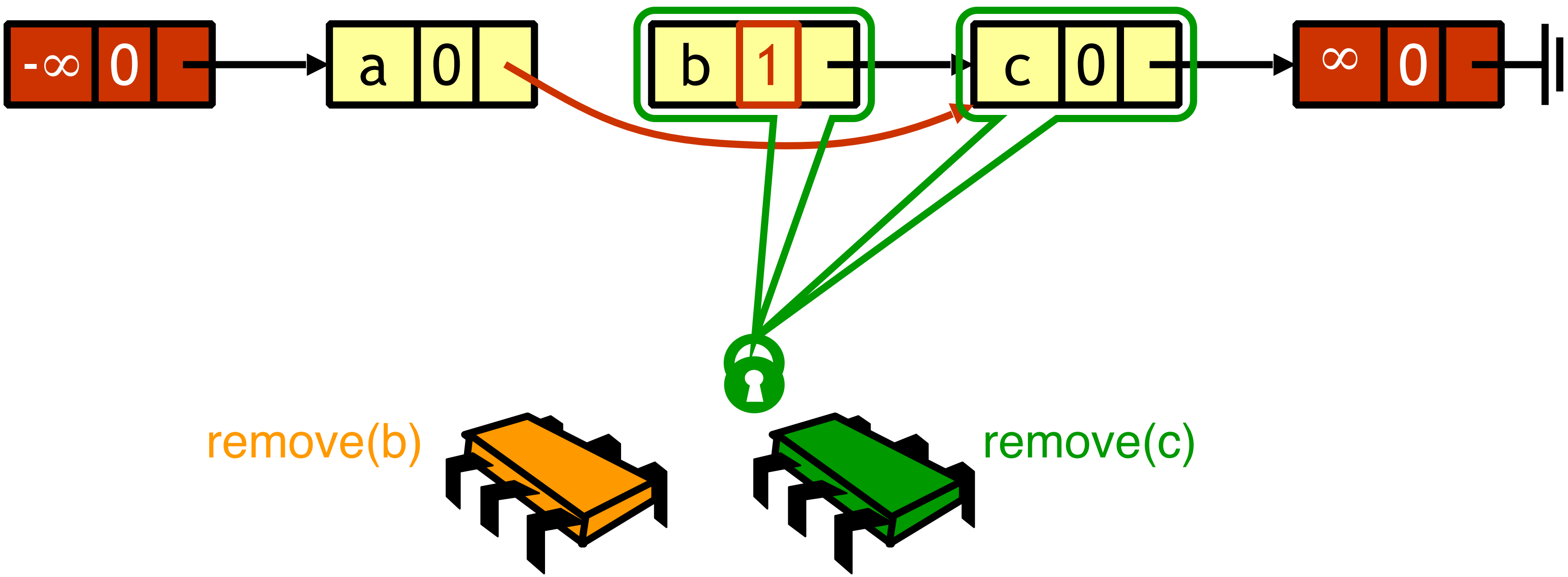
What Could Go Wrong?



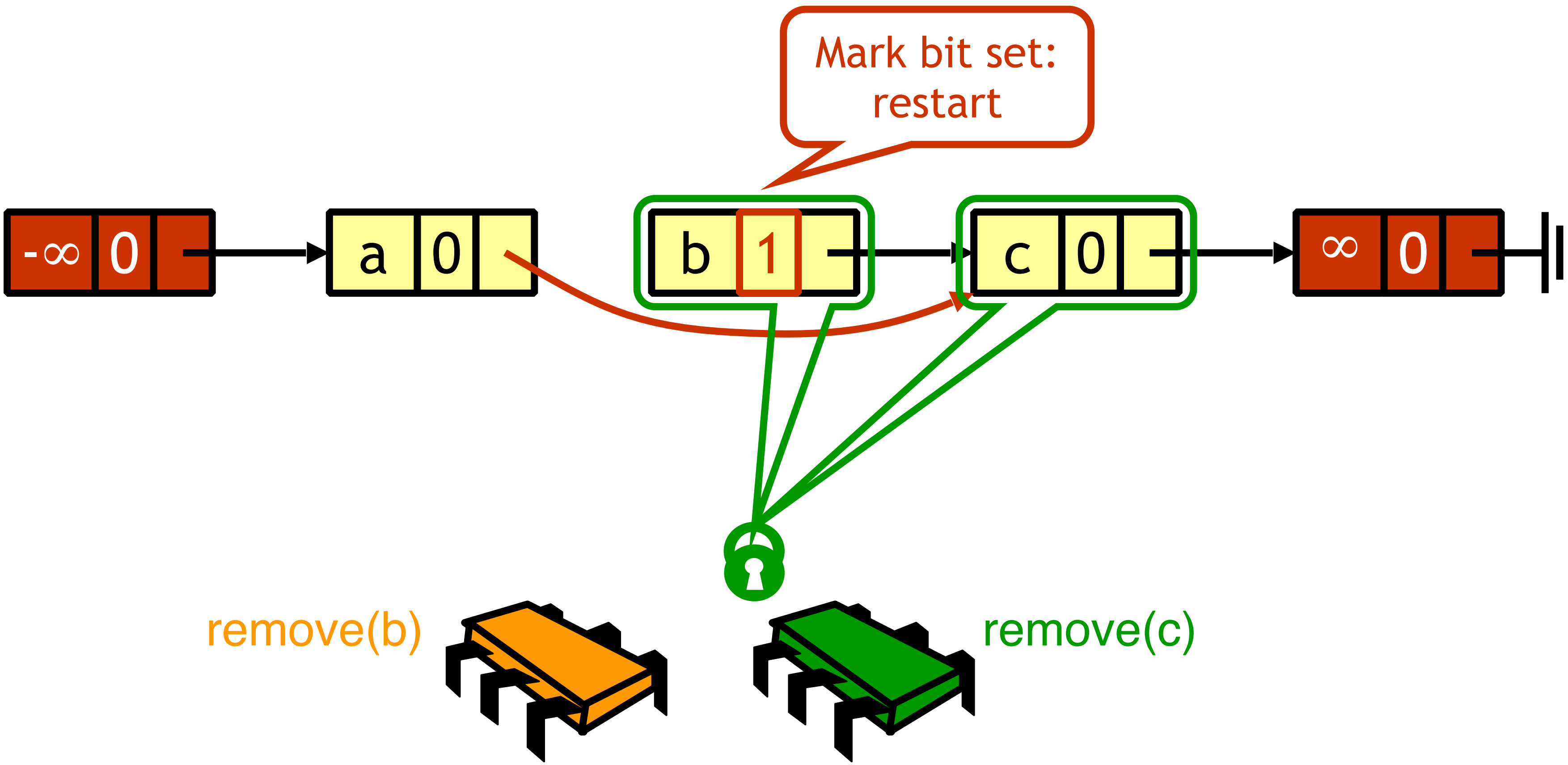
What Could Go Wrong?



What Could Go Wrong?



What Could Go Wrong?



Validation

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

Validation

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

Predecessor not logically removed

Validation

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

Current not logically removed

Validation

```
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.object == object) {
      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.object == object) {
      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.object == object) {
      curr.marked = true;
      pred.next = curr.next;
      return true;
    } else return false;
  }
} finally {
    pred.unlock(); curr.unlock();
} …
```

Object found

Remove

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

Logical removal

Remove

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

Physical removal

Contains

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

Contains

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

Start at the head

Contains

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

Traverse without locking
(nodes may have been removed)

Contains

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

Present and undeleted?

Summary: Lazy List

- Wait-free traversal uses mark bit + fact that list is ordered
  - Not marked ⇒ in the set
  - Marked or missing ⇒ not in the set
- Lazy add()
- Lazy remove()
- Wait-free contains()

Evaluation

- Good
  - contains() does not need to lock
    - In fact, it is wait-free!
    - Good because it is typically called often
  - Uncontended calls do not re-traverse
- Bad
  - Contended calls do re-traverse
  - Traffic jam if one thread delays

Traffic Jam

- Any concurrent data structure based on mutual exclusion has a weakness
- If one thread
  - Enters critical section
  - And "eats the big muffin" (stops running)
    - Cache miss, page fault, de-scheduled…
  - Everyone else using that lock is stuck!

Wait/Lock/Obstruction Freedom

Wait freedom

"All thread always makes progress"

Guarantees per-thread progress

vs.

Lock freedom

"Some thread always makes progress"

Guarantees system-wide progress

vs.

Obstruction freedom

"Any thread that runs by itself for long enough makes progress"

Lock-Free Data Structures

- No matter what…
  - Some thread will complete method call
  - Even if others halt at malicious times
  - Weaker than wait-free, yet
- Implies that
  - You cannot use locks
  - Um, that is why they call it lock-free

RMW Atomic Operations

- Read-modify-write operation combines…
  - Read from memory
  - Modify value
  - Write to memory
- … atomically
- Supported by modern processors
  - Atomic increment/decrement, test-and-set, compare-and-set, etc.
- In Java: `java.util.concurrent.atomic`

Atomic-Inc/Dec

```
public class AtomicInteger {
  int value;

  public synchronized int
  incrementAndGet() {
    value = value + 1;
    return value;
  }
  public synchronized int
  decrementAndGet() {
    return --value;
  }
}
```

Atomic-Inc/Dec

```
public class AtomicInteger {
  int value;

  public synchronized int
  incrementAndGet() {
    value = value + 1;
    return value;
  }
  public synchronized int
  decrementAndGet() {
    return --value;
  }
}
```

Package
java.util.concurrent.atomic

Atomic-Inc/Dec

```
public class AtomicInteger {
  int value;

  public synchronized int
  incrementAndGet() {
    value = value + 1;
    return value;
  }
  public synchronized int
  decrementAndGet() {
    return --value;
  }
}
```

Increment value

Atomic-Inc/Dec

```
public class AtomicInteger {
  int value;

  public synchronized int
  incrementAndGet() {
    value = value + 1;
    return value;
  }
  public synchronized int
  decrementAndGet() {
    return --value;
  }
}
```

Decrement value (pre-decrement operator is not atomic!)

Atomic-Inc/Dec

```
public class AtomicInteger {
  int value;

  public synchronized int
  incrementAndGet() {
    value = value + 1;
    return value;
  }
  public synchronized int
  decrementAndGet() {
    return --value;
  }
}
```

```
; x86
LOCK INC …
LOCK DEC …
LOCK XADD …
```

Decrement value (pre-decrement operator is not atomic!)

Get-and-Set

```
public class AtomicBoolean {
  boolean value;

  public synchronized boolean
  getAndSet(boolean newValue) {
    boolean prior = value;
    value = newValue;
    return prior;
  }
}
```

Get-and-Set

```
public class AtomicBoolean {
  boolean value;

  public synchronized boolean
  getAndSet(boolean newValue) {
    boolean prior = value;
    value = newValue;
    return prior;
  }
}
```

Set new value and return old value

Get-and-Set

```
public class AtomicBoolean {
  boolean value;

  public synchronized boolean
  getAndSet(boolean newValue) {
    boolean prior = value;
    value = newValue;
    return prior;
  }
}
```

Set new value and return old value

```
; x86
LOCK XCHG …
```

Compare-and-Set

```
public class AtomicInteger {
  int value;

  public synchronized boolean
  compareAndSet(int expValue,
              int newValue) {
    if (value == expValue) {
      value = newValue;
      return true;
    }
    return false;
  }
}
```

Compare-and-Set

```
public class AtomicInteger {
  int value;

  public synchronized boolean
  compareAndSet(int expValue,
              int newValue) {
    if (value == expValue) {
      value = newValue;
      return true;
    }
    return false;
  }
}
```

Set new value and return true if old value matches expected value, return false otherwise

Compare-and-Set

```
public class AtomicInteger {
  int value;

  public synchronized boolean
  compareAndSet(int expValue,
                int newValue) {
    if (value == expValue) {
      value = newValue;
      return true;
    }
    return false;
  }
}
```

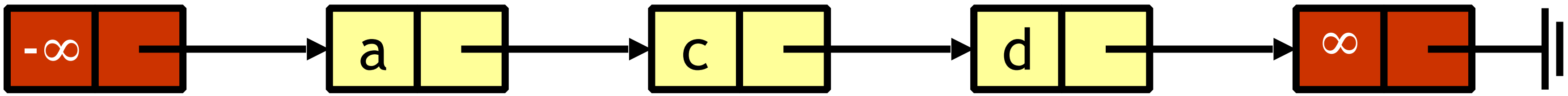
Set new value and return true if old value matches expected value, return false otherwise

```
; x86
LOCK CMPXCHG …
```

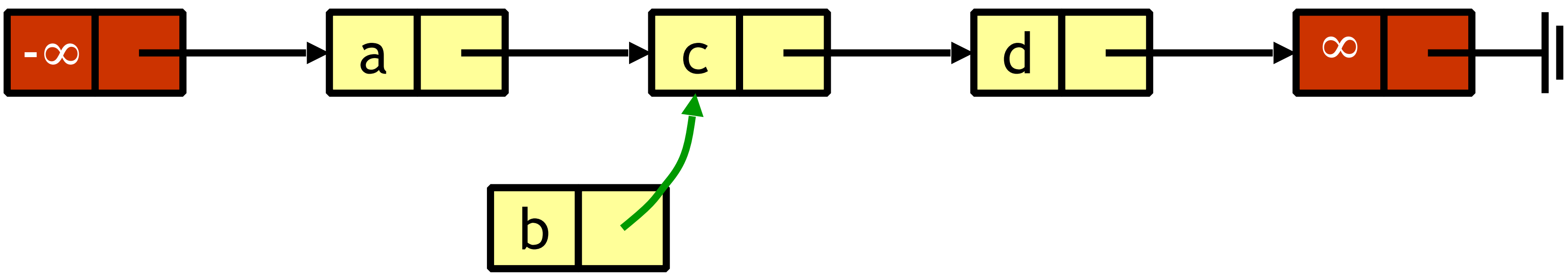
Lock-Free Lists

- Next logical step
- Eliminate locking entirely
- contains() wait-free and add() and remove() lock-free
- Use only compareAndSet() to atomically update links

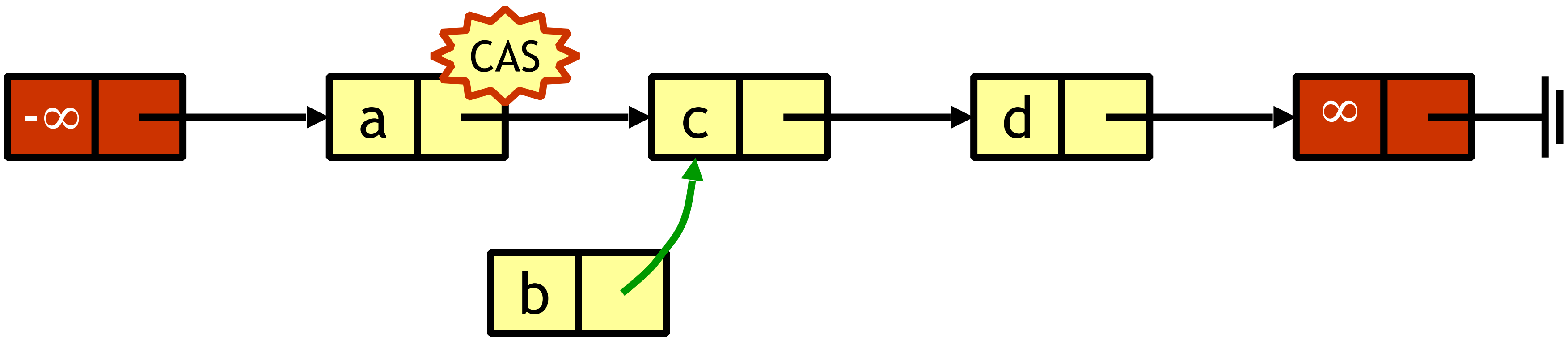
Adding a Node



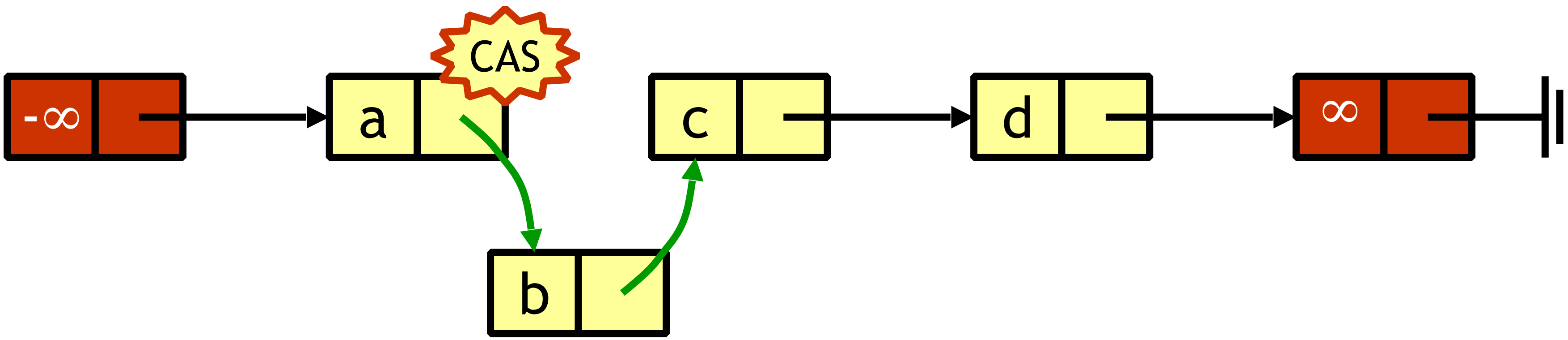
Adding a Node



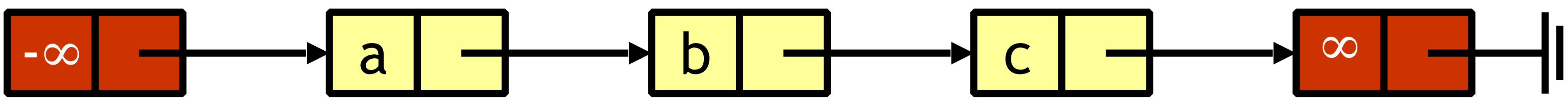
Adding a Node



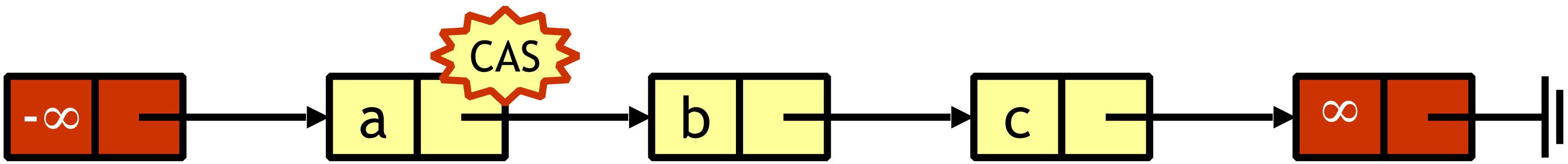
Adding a Node



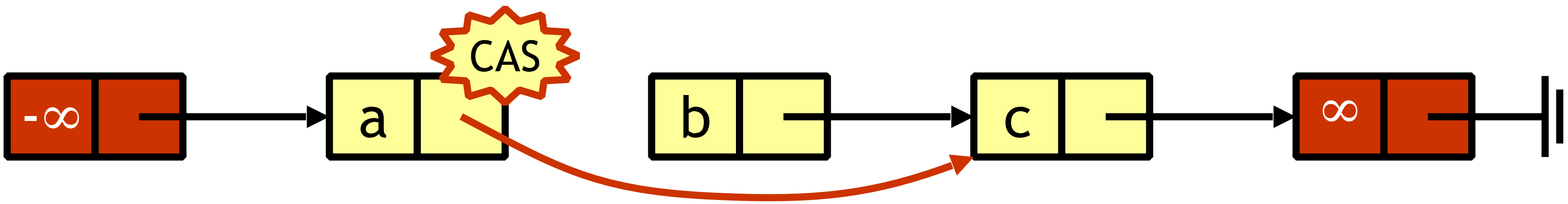
Removing a Node



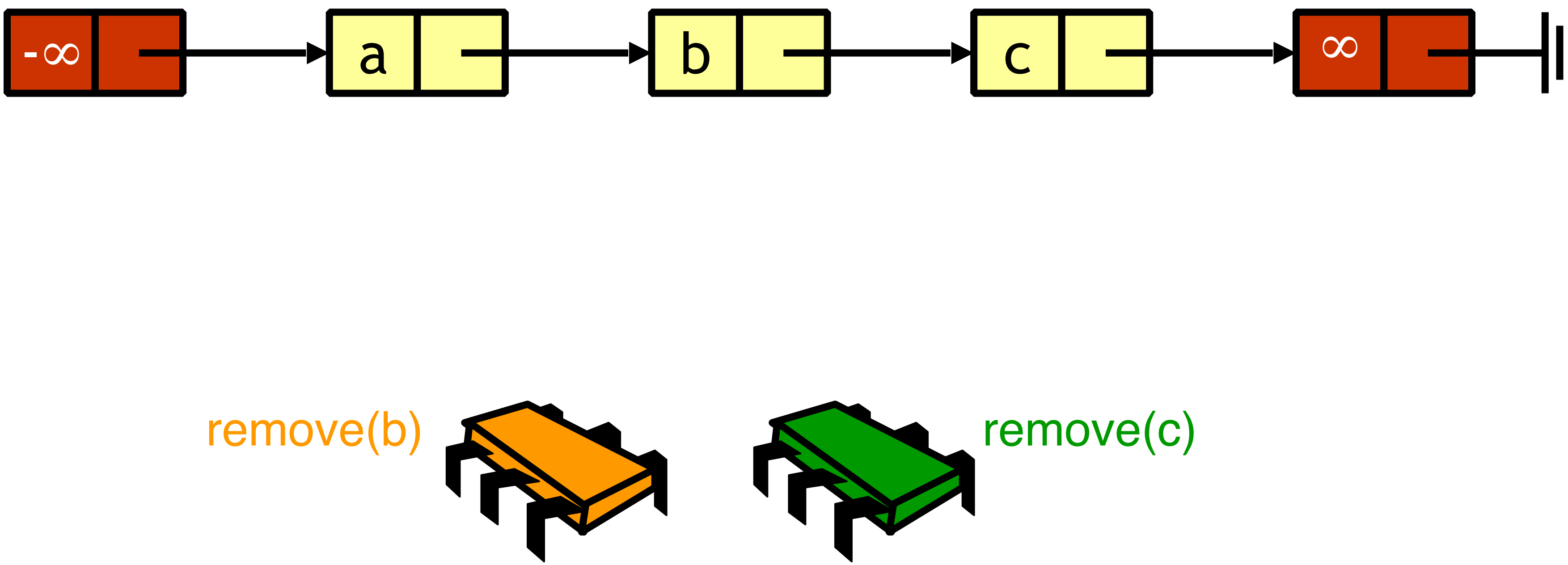
Removing a Node



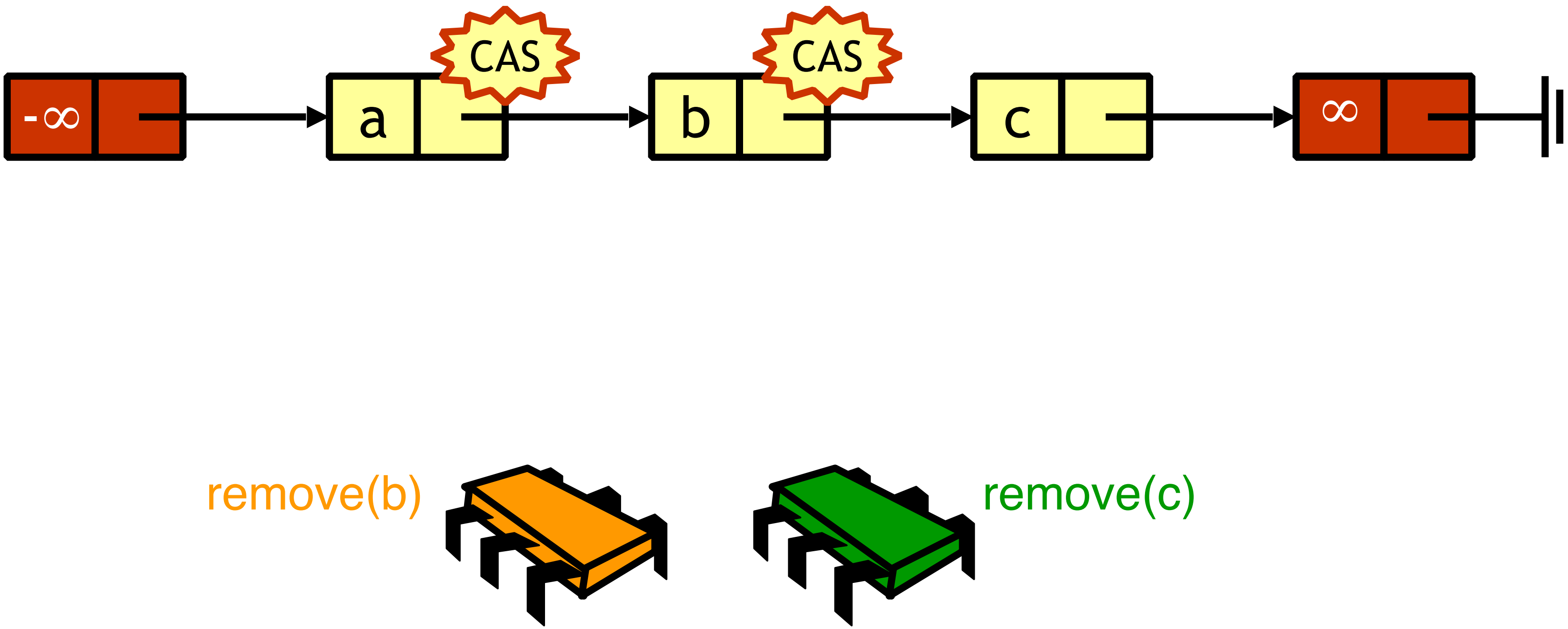
Removing a Node



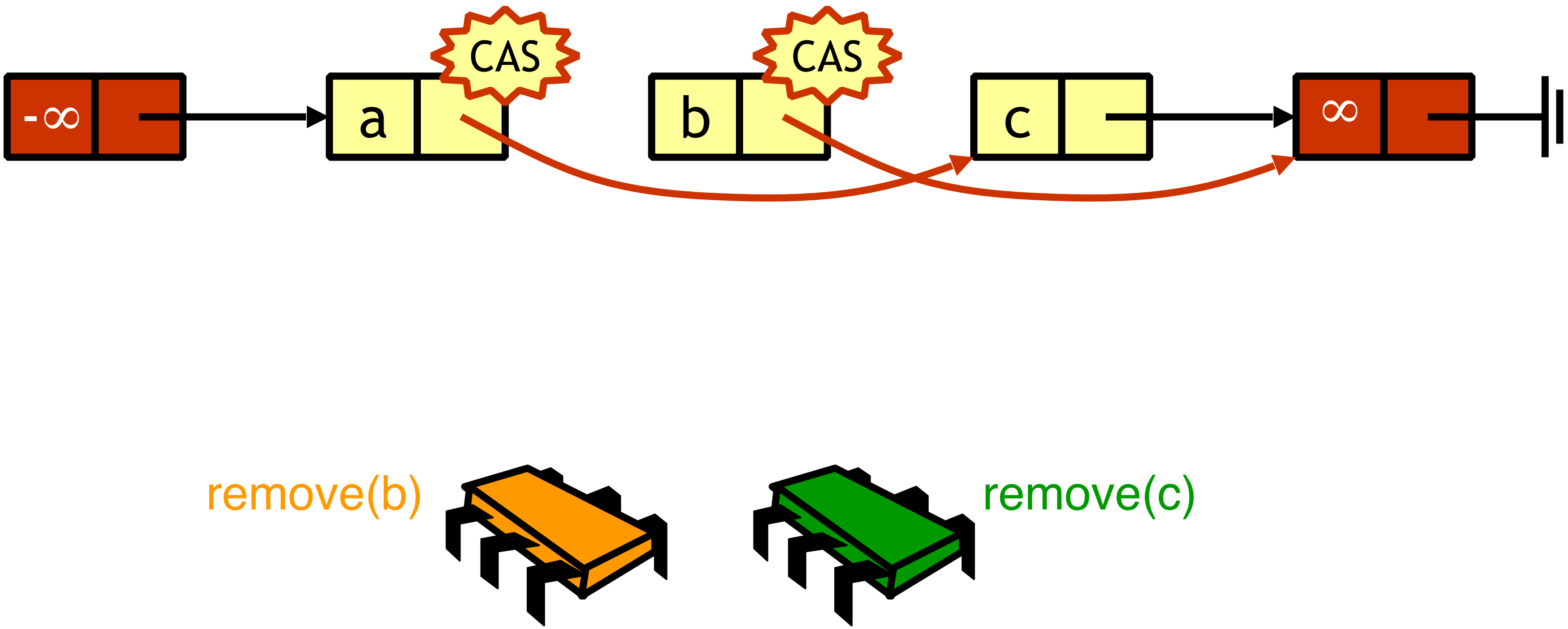
What Could Go Wrong?



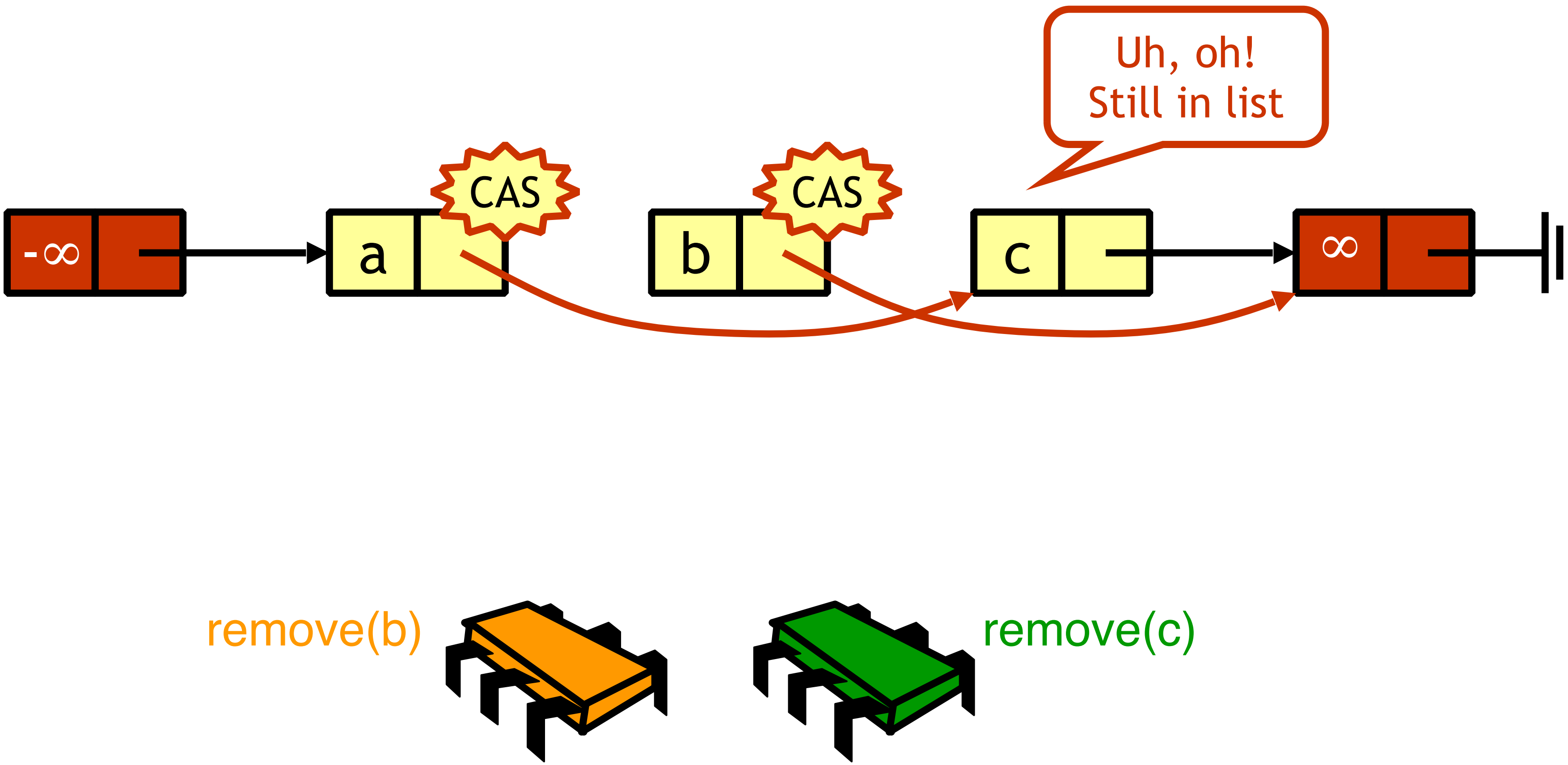
What Could Go Wrong?



What Could Go Wrong?



What Could Go Wrong?



What Could Go Wrong?

- Problem
  - Method updates node's next field after node has been removed
- Solution
  - Use AtomicMarkableReference
  - Atomically
    - Swing reference and update flag
  - Remove in two steps
    - Set mark bit in next field
    - Redirect predecessor's pointer

Marking a Node

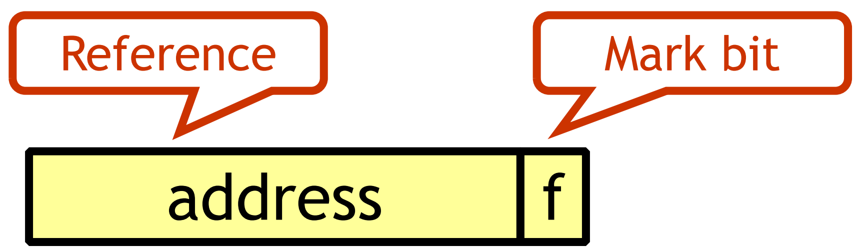
- AtomicMarkableReference class
  - In package java.util.concurrent.atomic
  - Holds a reference and a mark bit

Marking a Node

- AtomicMarkableReference class
  - In package java.util.concurrent.atomic
  - Holds a reference and a mark bit

Marking a Node

- AtomicMarkableReference class
  - In package java.util.concurrent.atomic
  - Holds a reference and a mark bit



AtomicMarkableReference

```
public class AtomicMarkableReference <T> {
  public T get(boolean[] marked);
  public boolean compareAndSet(
    T expectedRef,
    T updateRef,
    boolean expectedMark,
    boolean updateMark);
  public boolean attemptMark(
    T expectedRef,
    boolean updateMark);
  …
}
```

AtomicMarkableReference

Data type

```
public class AtomicMarkableReference <T> {
  public T get(boolean[] marked);
  public boolean compareAndSet(
    T expectedRef,
    T updateRef,
    boolean expectedMark,
    boolean updateMark);
  public boolean attemptMark(
    T expectedRef,
    boolean updateMark);
  …
}
```

AtomicMarkableReference

```
public class AtomicMarkableReference <T> {
  public T get(boolean[] marked);
  public boolean compareAndSet(
    T expectedRef,
    T updateRef,
    boolean expectedMark,
    boolean updateMark);
  public boolean attemptMark(
    T expectedRef,
    boolean updateMark);
  …
}
```

Extract reference and mark (at index 0)

AtomicMarkableReference

```
public class AtomicMarkableReference <T> {
  public T get(boolean[] marked);
  public boolean compareAndSet(
    T expectedRef,
    T updateRef,
    boolean expectedMark,
    boolean updateMark);
  public boolean attemptMark(
    T expectedRef,
    boolean updateMark);
  …
}
```

If this is the current reference…

AtomicMarkableReference

```
public class AtomicMarkableReference <T> {
  public T get(boolean[] marked);
  public boolean compareAndSet(
    T expectedRef,
    T updateRef,
    boolean expectedMark,
    boolean updateMark);
  public boolean attemptMark(
    T expectedRef,
    boolean updateMark);
  …
}
```

If this is the current reference…

…and this is the current mark…

AtomicMarkableReference

```
public class AtomicMarkableReference <T> {
  public T get(boolean[] marked);
  public boolean compareAndSet(
    T expectedRef,
    T updateRef,
    boolean expectedMark,
    boolean updateMark);
  public boolean attemptMark(
    T expectedRef,
    boolean updateMark);
  …
}
```

…then change to this new reference…

AtomicMarkableReference

```
public class AtomicMarkableReference <T> {
  public T get(boolean[] marked);
  public boolean compareAndSet(
    T expectedRef,
    T updateRef,
    boolean expectedMark,
    boolean updateMark);
  public boolean attemptMark(
    T expectedRef,
    boolean updateMark);
  …
}
```

…then change to this new reference…

…and this new mark

AtomicMarkableReference

```
public class AtomicMarkableReference <T> {
  public T get(boolean[] marked);
  public boolean compareAndSet(
    T expectedRef,
    T updateRef,
    boolean expectedMark,
    boolean updateMark);
  public boolean attemptMark(
    T expectedRef,
    boolean updateMark);
  …
}
```

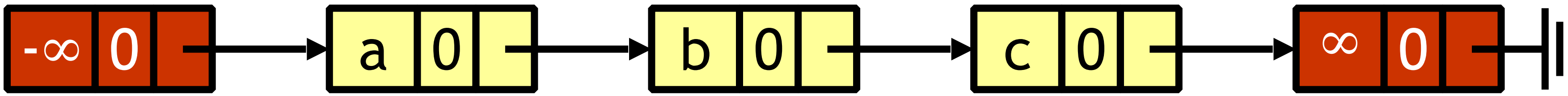
If this is the current reference…

AtomicMarkableReference

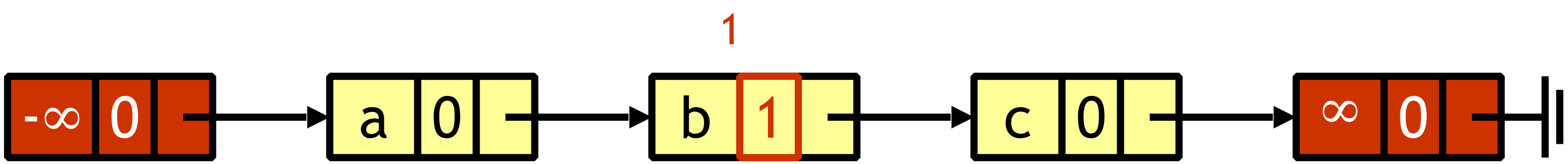
```
public class AtomicMarkableReference <T> {
  public T get(boolean[] marked);
  public boolean compareAndSet(
    T expectedRef,
    T updateRef,
    boolean expectedMark,
    boolean updateMark);
  public boolean attemptMark(
    T expectedRef,
    boolean updateMark);
  …
}
```

…then change to this new mark

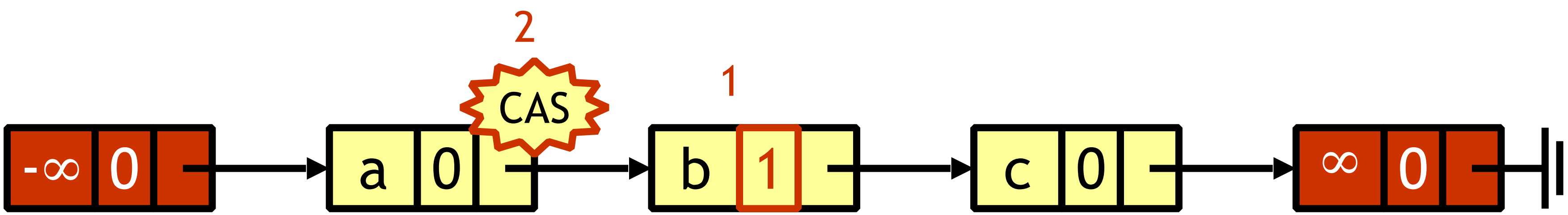
Removing a Node



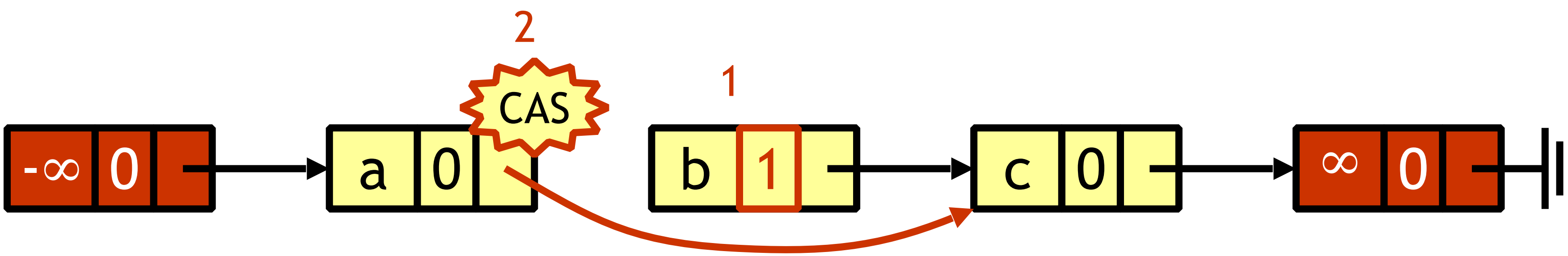
Removing a Node



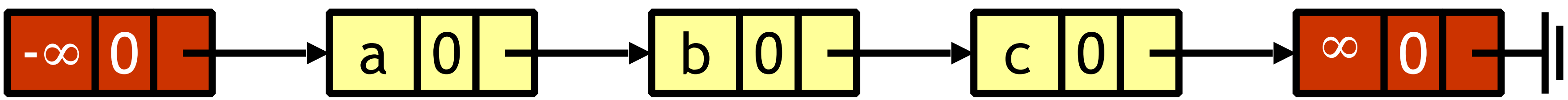
Removing a Node

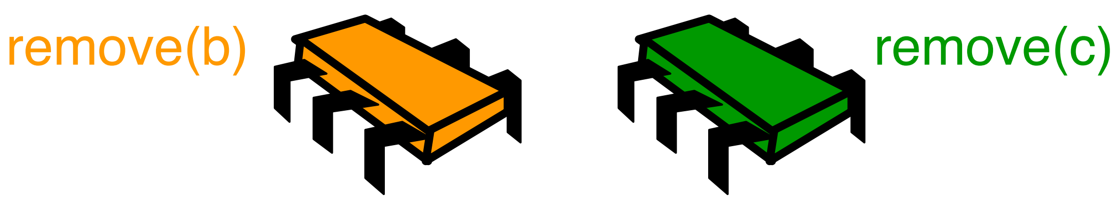


Removing a Node

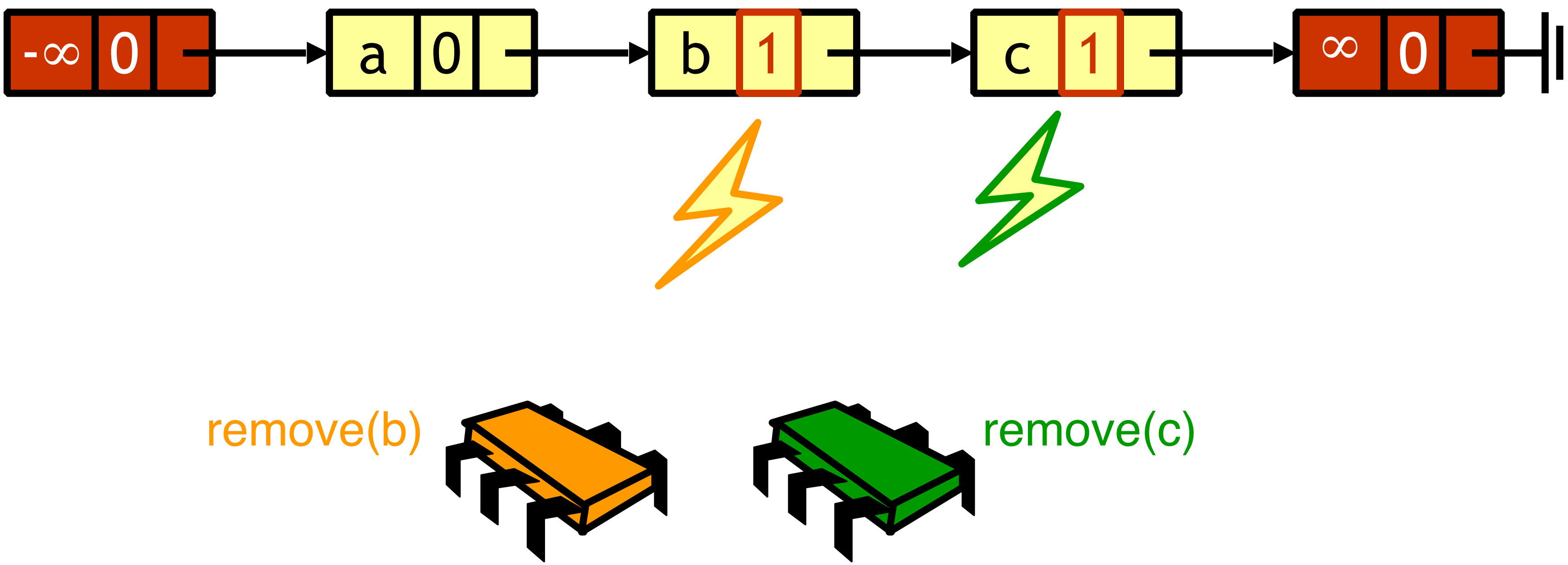


What Could Go Wrong?

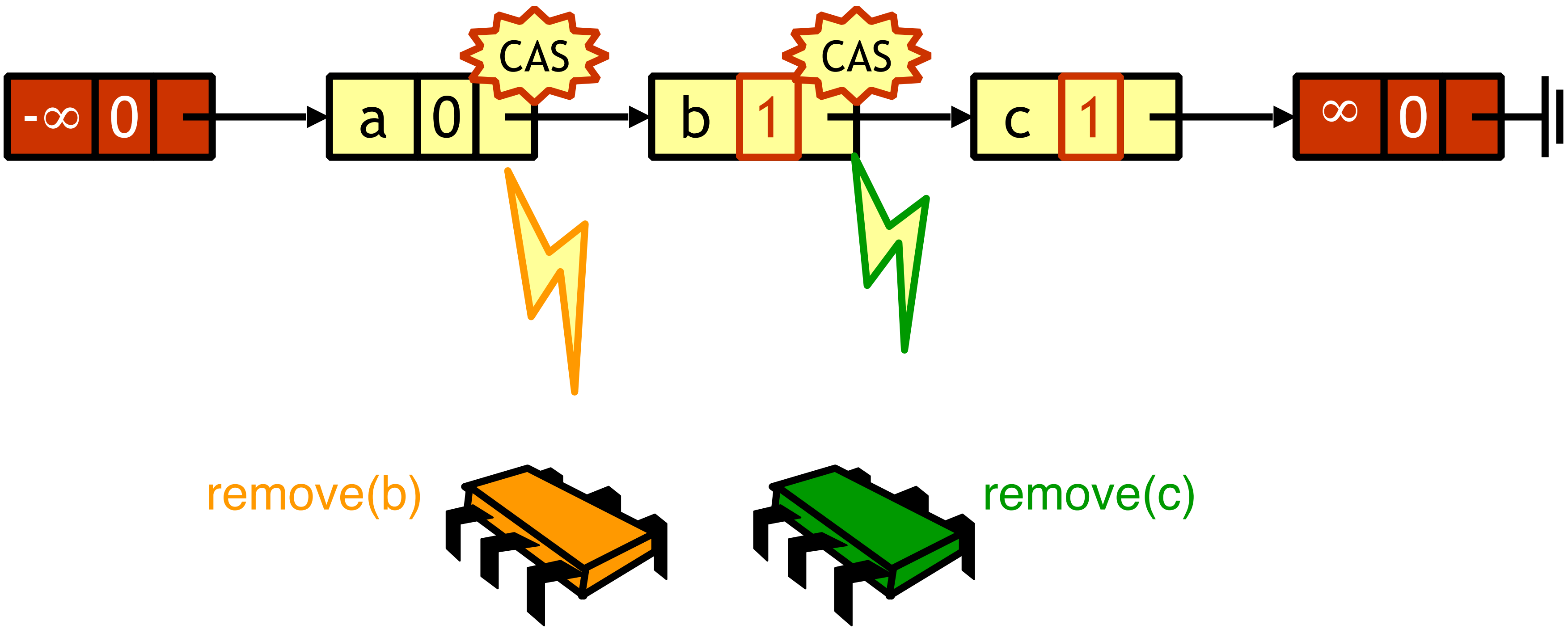




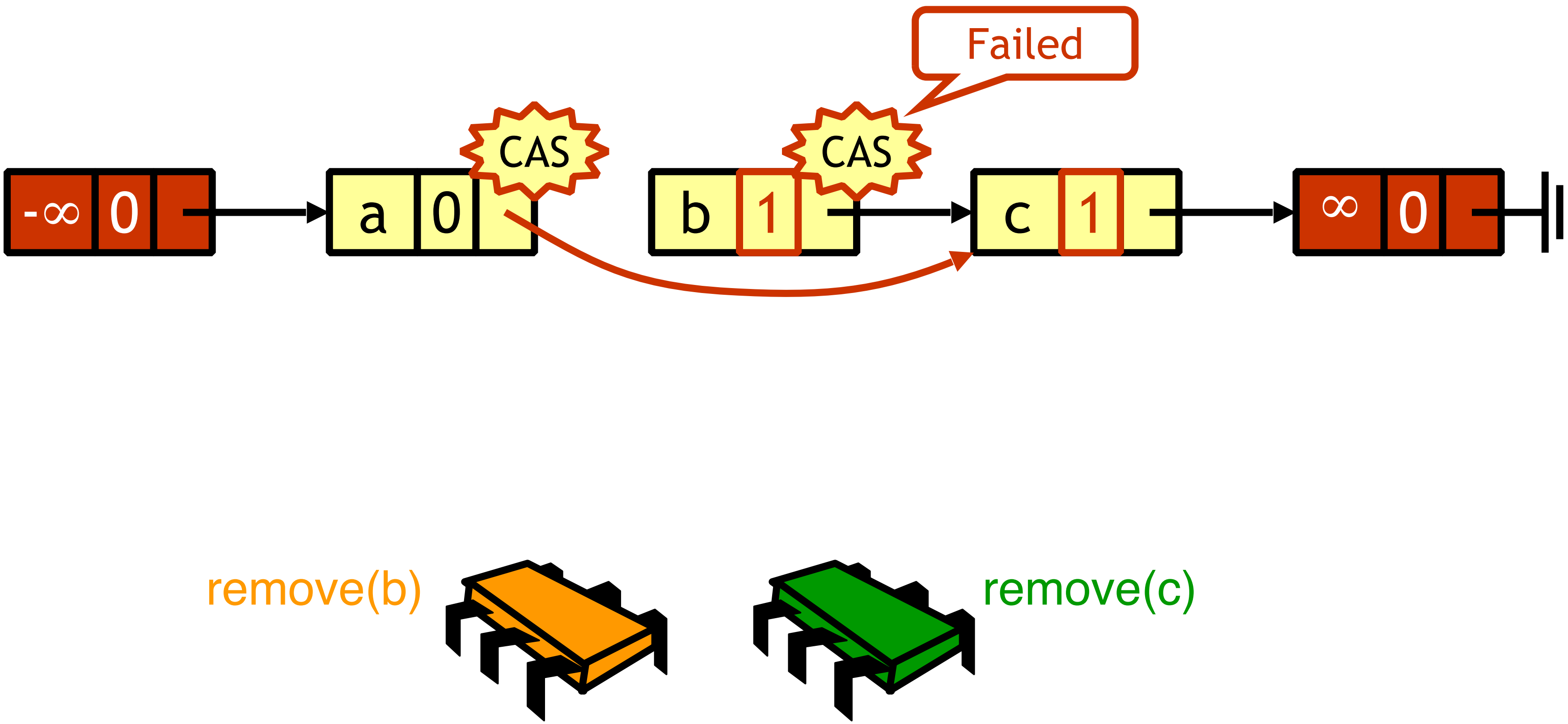
What Could Go Wrong?



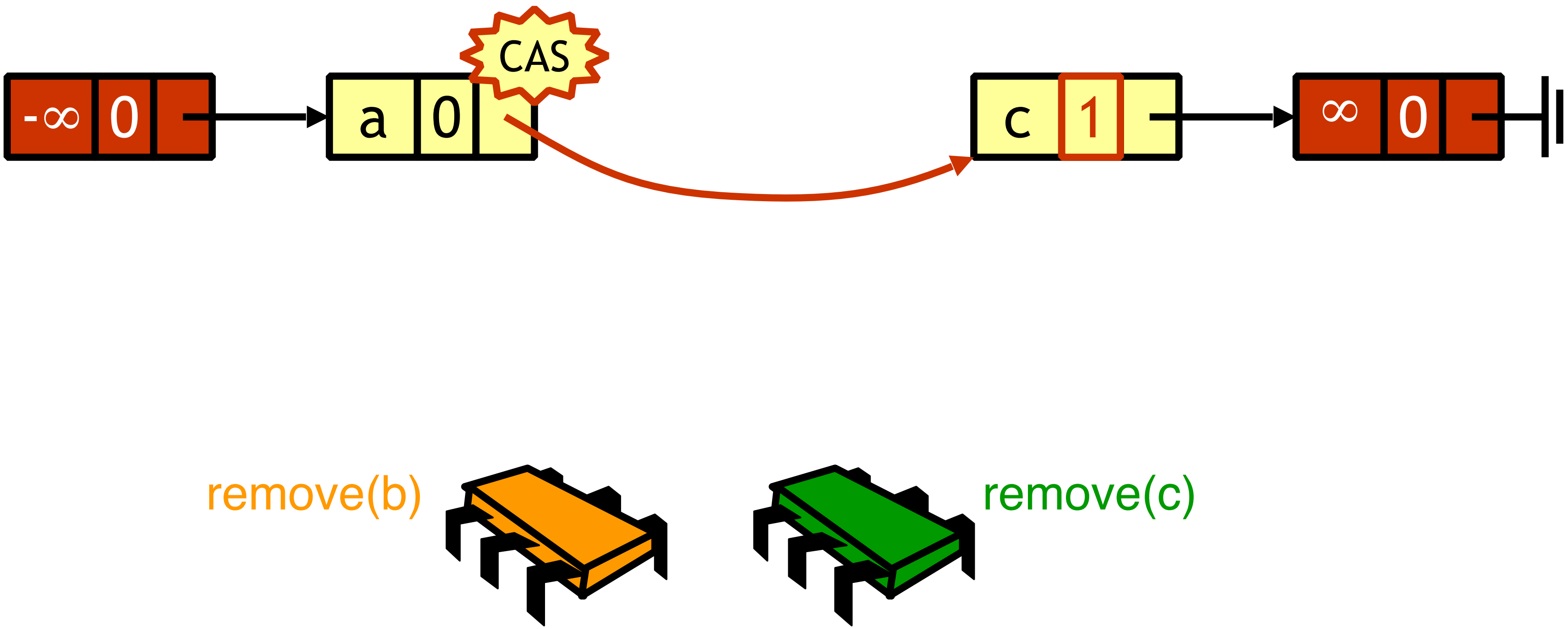
What Could Go Wrong?



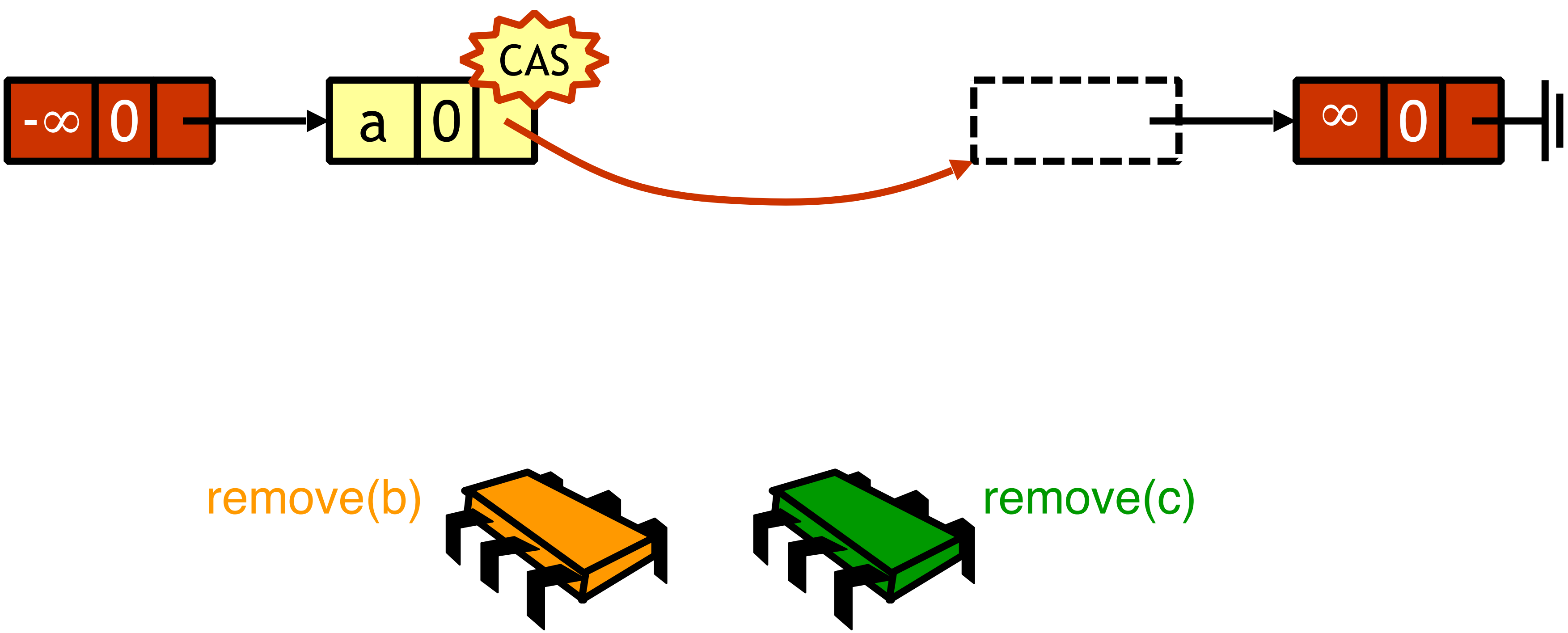
What Could Go Wrong?



What Could Go Wrong?



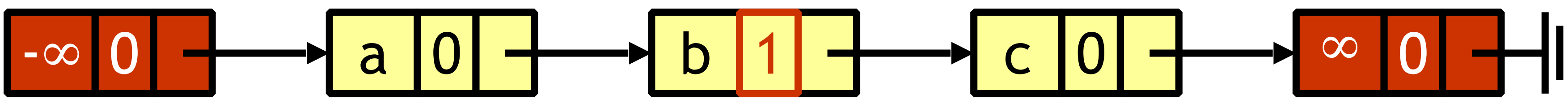
What Could Go Wrong?

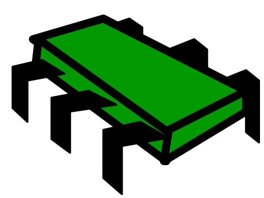


Traversing the List

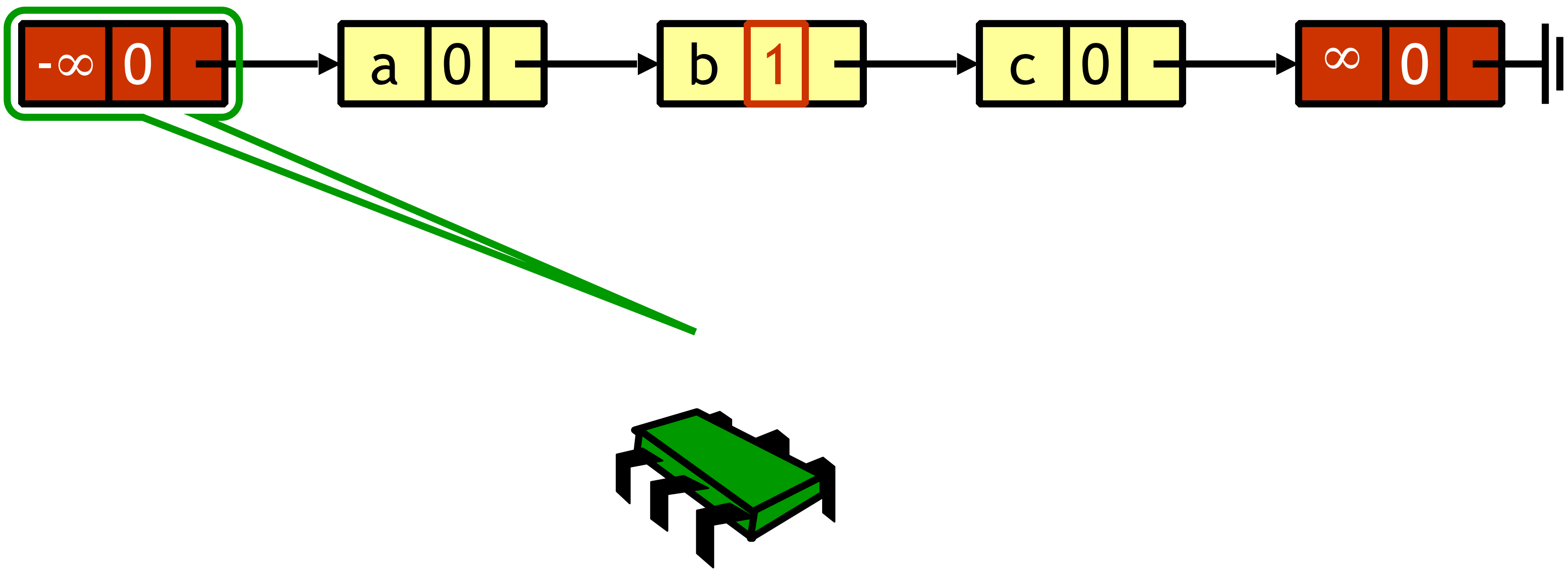
- What do you do when you find a “logically” deleted node in your path?
- Finish the job
  - CAS the predecessor’s next field
  - Proceed (repeat as needed)

Lock-Free Traversal

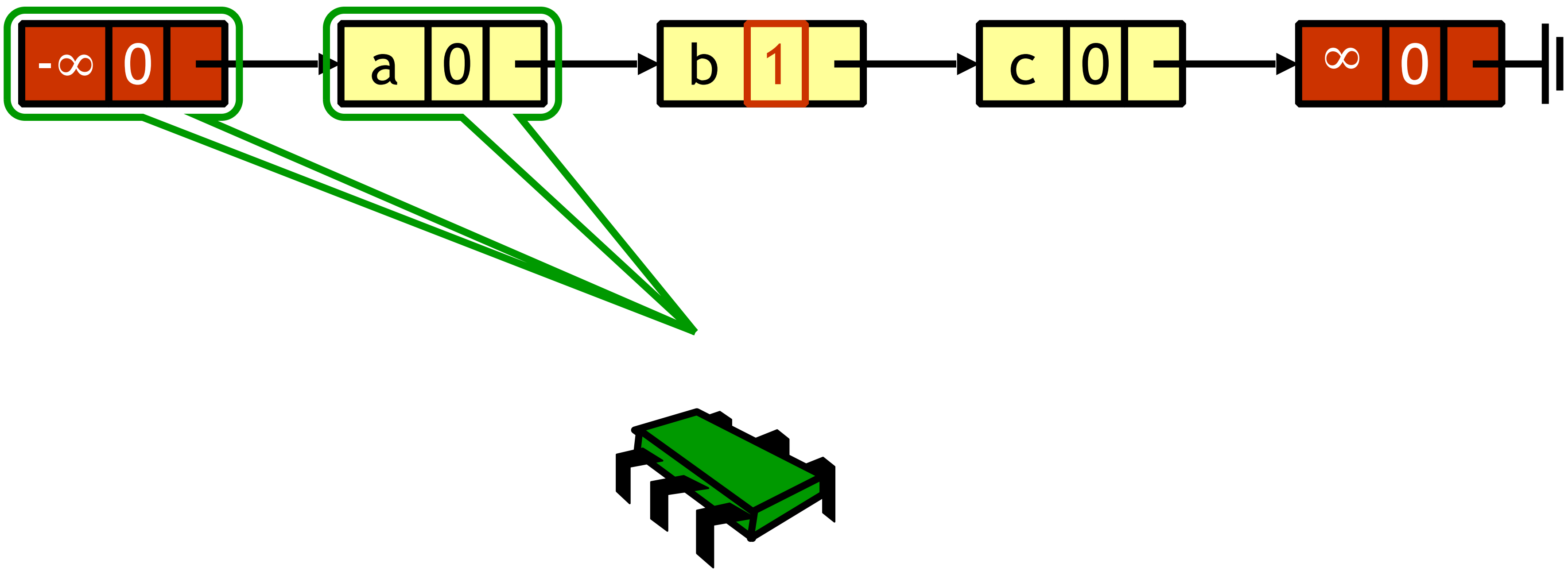




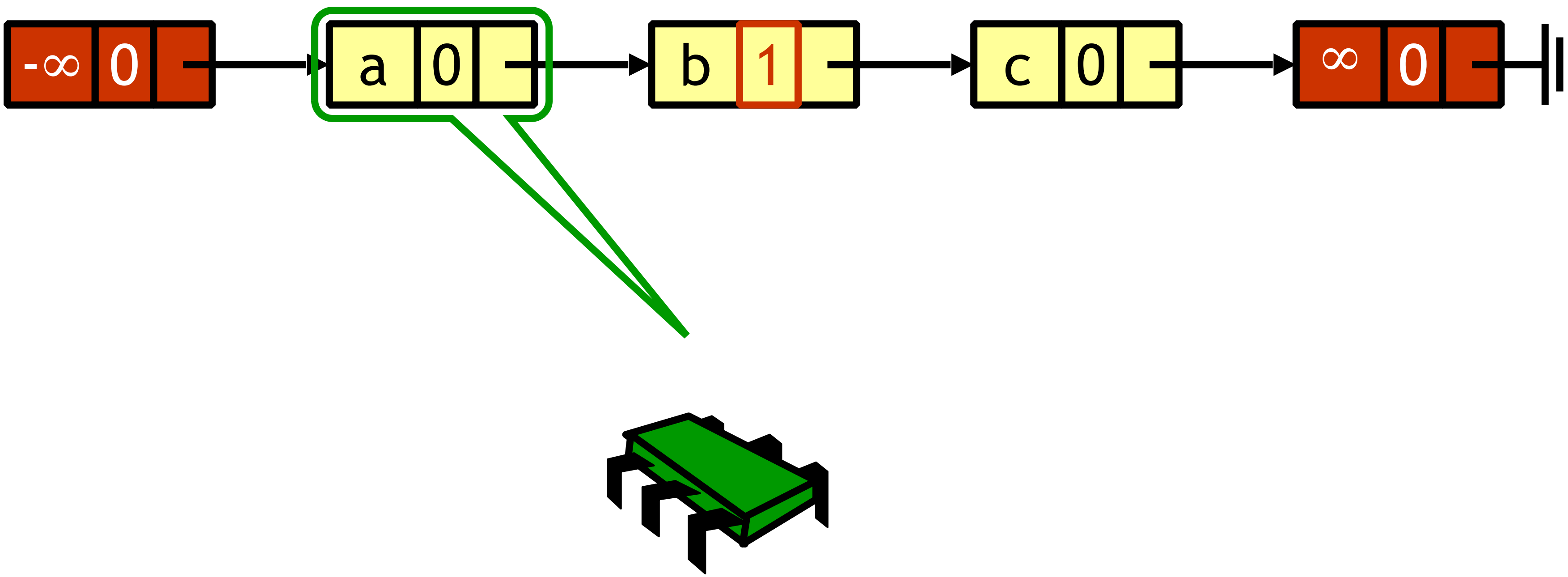
Lock-Free Traversal



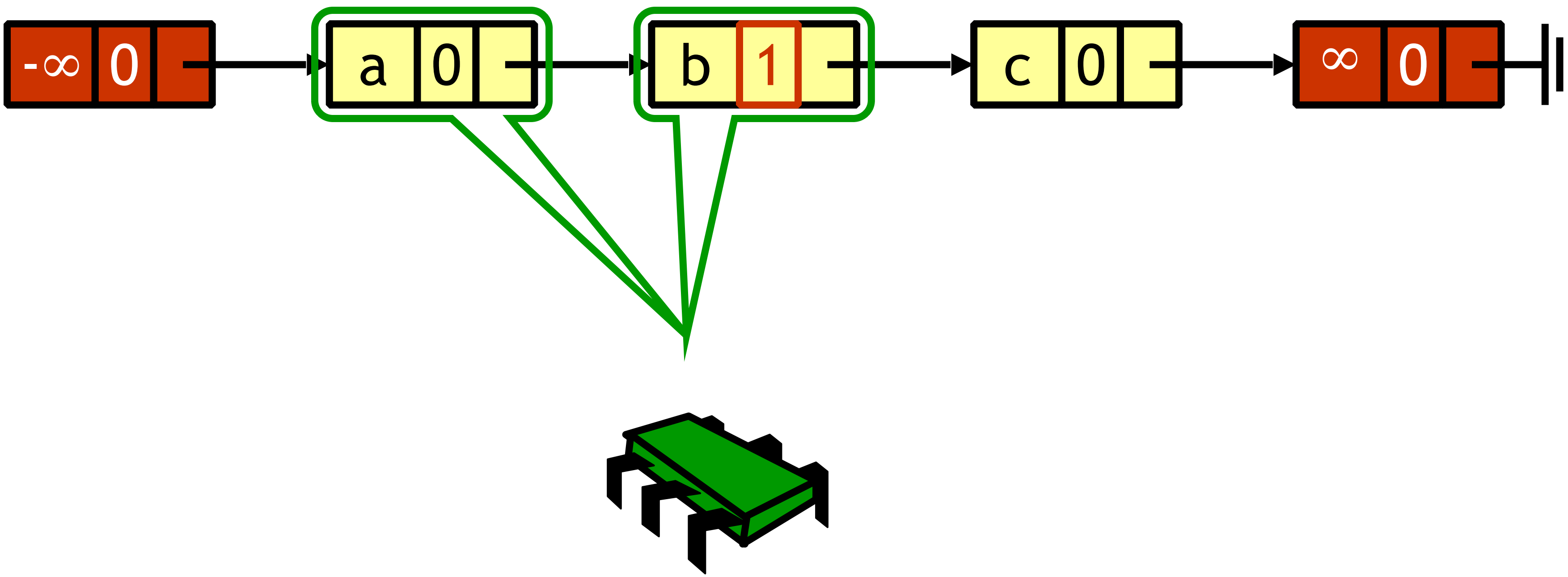
Lock-Free Traversal



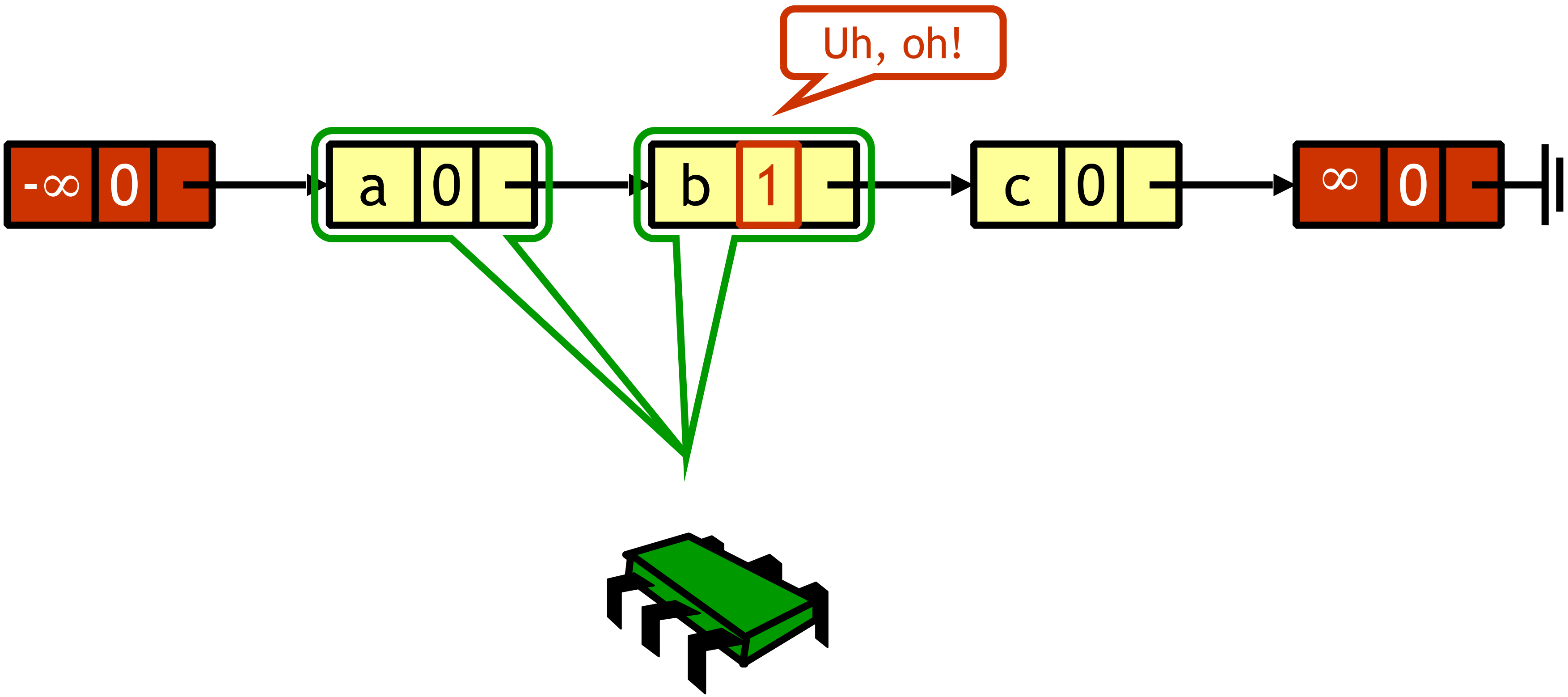
Lock-Free Traversal



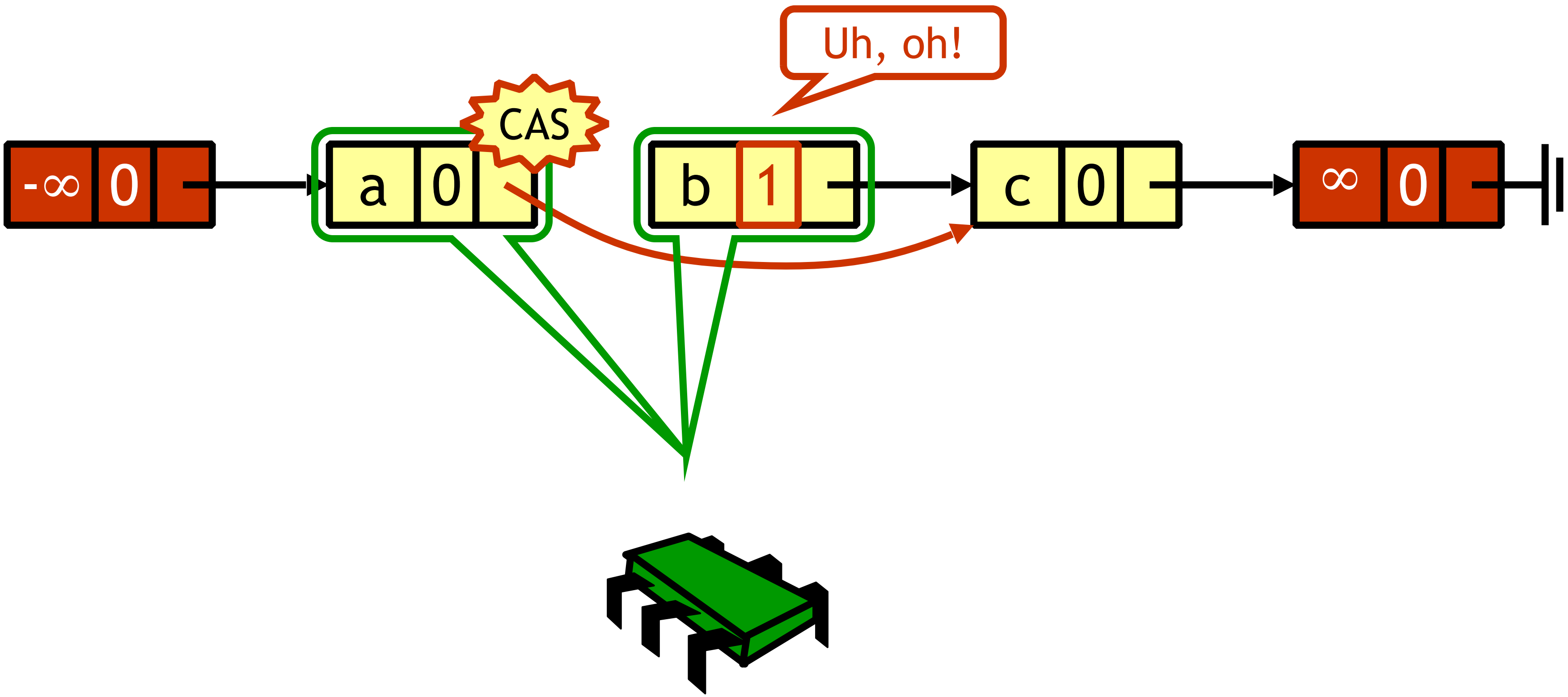
Lock-Free Traversal



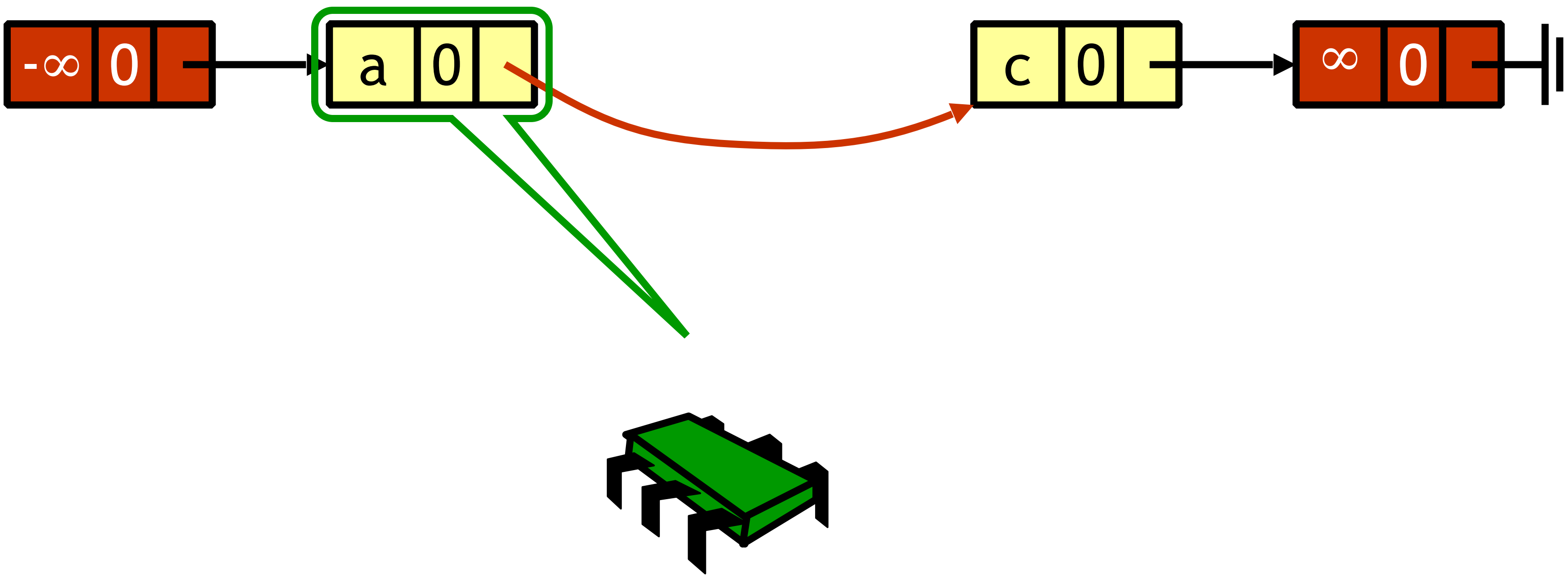
Lock-Free Traversal



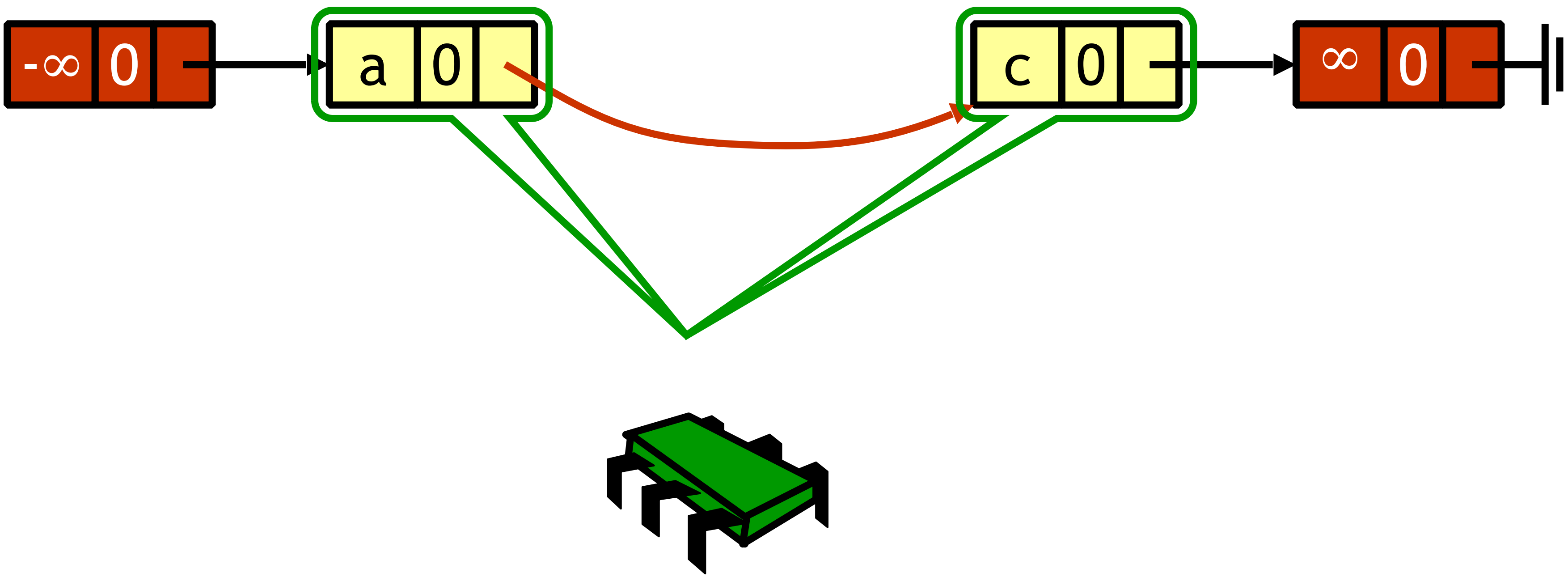
Lock-Free Traversal



Lock-Free Traversal



Lock-Free Traversal



The Window Class

```
class Window {
  Node pred;
  Node curr;
  Window(Node pred, Node curr) {
    this.pred = pred;
    this.curr = curr;
  }
}
```

The Window Class

```
class Window {
  Node pred;
  Node curr;
  Window(Node pred, Node curr) {
    this.pred = pred;
    this.curr = curr;
  }
}
```

A container for predecessor and current nodes

Using the Find Method

```
…
Window window = find(head, object);
Node pred = window.pred;
Node curr = window.curr;
…
```

Using the Find Method

```
…
Window window = find(head, object);
Node pred = window.pred;
Node curr = window.curr;
…
```

Find window

Using the Find Method

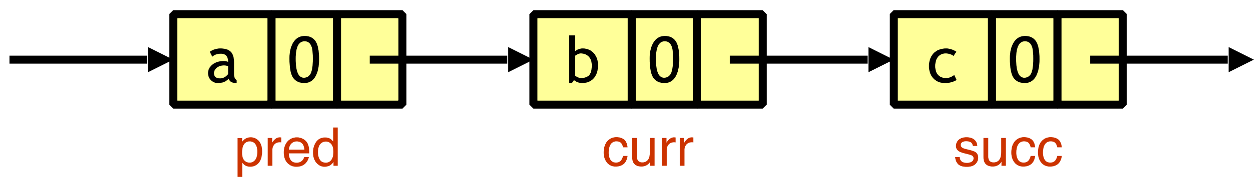
```
…
Window window = find(head, object);
Node pred = window.pred;
Node curr = window.curr;
…
```

Extract pred and curr

The Find Method

```
Window window = find(head, b);
Node pred = window.pred;
Node curr = window.curr;
```

Object in list:



Object not in list:

Remove

```
public boolean remove(T object) {
  boolean b;
  while (true) {
    Window window = find(head, object);
    Node pred = window.pred, curr = window.curr;
    if (curr.object != object)
      return false;
    Node succ = curr.next.getReference();
    b = curr.next.compareAndSet(succ, succ, false, true);
    if (!b) continue;
    pred.next.compareAndSet(curr, succ, false, false);
    return true;
  }
}
```

Remove

```
public boolean remove(T object) {
  boolean b;
  while (true) {                Keep trying
    Window window = find(head, object);
    Node pred = window.pred, curr = window.curr;
    if (curr.object != object)
      return false;
    Node succ = curr.next.getReference();
    b = curr.next.compareAndSet(succ, succ, false, true);
    if (!b) continue;
    pred.next.compareAndSet(curr, succ, false, false);
    return true;
  }
}
```

Remove

```
public boolean remove(T object) {
  boolean b;
  while (true) {
    Window window = find(head, object);
    Node pred = window.pred, curr = window.curr;
    if (curr.object != object)
      return false;
    Node succ = curr.next.getReference();
    b = curr.next.compareAndSet(succ, succ, false, true);
    if (!b) continue;
    pred.next.compareAndSet(curr, succ, false, false);
    return true;
  }
}
```

Find neighbors

Remove

```
public boolean remove(T object) {
  boolean b;
  while (true) {
    Window window = find(head, object);
    Node pred = window.pred, curr = window.curr;
    if (curr.object != object)
      return false;
    Node succ = curr.next.getReference();
    b = curr.next.compareAndSet(succ, succ, false, true);
    if (!b) continue;
    pred.next.compareAndSet(curr, succ, false, false);
    return true;
  }
}
```

Not there

Remove

```
public boolean remove(T object) {
  boolean b;
  while (true) {
    Window window = find(head, object);
    Node pred = window.pred, curr = window.curr;
    if (curr.object != object)
      return false;
    Node succ = curr.next.getReference();
    b = curr.next.compareAndSet(succ, succ, false, true);
    if (!b) continue;
    pred.next.compareAndSet(curr, succ, false, false);
    return true;
  }
}
```

Try to mark node as deleted

Remove

```
public boolean remove(T object) {
  boolean b;
  while (true) {
    Window window = find(head, object);
    Node pred = window.pred, curr = window.curr;
    if (curr.object != object)
      return false;
    Node succ = curr.next.getReference();
    b = curr.next.compareAndSet(succ, succ, false, true);
    if (!b) continue;
    pred.next.compareAndSet(curr, succ, false, false);
    return true;
  }
}
```

If it fails, retry, otherwise job done

Remove

```
public boolean remove(T object) {
  boolean b;
  while (true) {
    Window window = find(head, object);
    Node pred = window.pred, curr = window.curr;
    if (curr.object != object)
      return false;
    Node succ = curr.next.getReference();
    b = curr.next.compareAndSet(succ, succ, false, true);
    if (!b) continue;
    pred.next.compareAndSet(curr, succ, false, false);
    return true;
  }
}
```

Try to advance reference
(if it fails, someone else did or will advance it)

Add

```
public boolean add(T object) {
  while (true) {
    Window window = find(head, object);
    Node pred = window.pred, curr = window.curr;
    if (curr.object == object)
      return false;
    Node n = new Node(object);
    n.next = new AtomicMarkableReference(curr, false);
    if (pred.next.compareAndSet(curr, n, false, false))
      return true;
  }
}
```

Add

```
public boolean add(T object) {
  while (true) {
    Window window = find(head, object);
    Node pred = window.pred, curr = window.curr;
    if (curr.object == object)
      return false;
    Node n = new Node(object);
    n.next = new AtomicMarkableReference(curr, false);
    if (pred.next.compareAndSet(curr, n, false, false))
      return true;
  }
}
```

Already there

Add

```
public boolean add(T object) {
  while (true) {
    Window window = find(head, object);
    Node pred = window.pred, curr = window.curr;
    if (curr.object == object)
      return false;
    Node n = new Node(object);
    n.next = new AtomicMarkableReference(curr, false);
    if (pred.next.compareAndSet(curr, n, false, false))
      return true;
  }
}
```

Create new node

Add

```
public boolean add(T object) {
  while (true) {
    Window window = find(head, object);
    Node pred = window.pred, curr = window.curr;
    if (curr.object == object)
      return false;
    Node n = new Node(object);
    n.next = new AtomicMarkableReference(curr, false);
    if (pred.next.compareAndSet(curr, n, false, false))
      return true;
  }
}
```

Install new node, else retry loop

Wait-Free Contains

```
public boolean contains(T object) {
  boolean marked[] = new boolean[1];
  int key = object.hashCode();
  Node curr = this.head;
  while (curr.key <= key) {
    if (object == curr.object)
      break;
    curr = curr.next;
  }
  curr.next.get(marked);
  return (object == curr.object && !marked[0]);
}
```

Wait-Free Contains

```
public boolean contains(T object) {
  boolean marked[] = new boolean[1];
  int key = object.hashCode();
  Node curr = this.head;
  while (curr.key <= key) {
    if (object == curr.object)
      break;
    curr = curr.next;
  }
  curr.next.get(marked);
  return (object == curr.object && !marked[0]);
}
```

Only difference from lazy list is that we get and check mark

Lock-Free Find

```
public Window find(Node head, T object) {
  Node pred, curr, succ; int key = object.hashCode();
  boolean[] marked = { false }; boolean b;
  retry: while (true) {
    pred = head; curr = pred.next.getReference();
    while (true) {
      succ = curr.next.get(marked);
      while (marked[0]) { … }
      if ((curr.key == key && curr.object == object)
          || curr.key > key)
        return new Window(pred, curr);
      pred = curr; curr = succ;
    }
  }
}
```

Lock-Free Find

```
public Window find(Node head, T object) {
  Node pred, curr, succ; int key = object.hashCode();
  boolean[] marked = { false }; boolean b;
  retry: while (true) {
    pred = head; curr = pred.next.getReference();
    while (true) {
      succ = curr.next.get(marked);
      while (marked[0]) { … }
      if ((curr.key == key && curr.object == object)
          || curr.key > key)
        return new Window(pred, curr);
      pred = curr; curr = succ;
    }
  }
}
```

Restart if list changes while traversed

Lock-Free Find

```
public Window find(Node head, T object) {
  Node pred, curr, succ; int key = object.hashCode();
  boolean[] marked = { false }; boolean b;
  retry: while (true) {
    pred = head; curr = pred.next.getReference();
    while (true) {
      succ = curr.next.get(marked);
      while (marked[0]) { … }
      if ((curr.key == key && curr.object == object)
          || curr.key > key)
        return new Window(pred, curr);
      pred = curr; curr = succ;
    }
  }
}
```

Start from head

Lock-Free Find

```
public Window find(Node head, T object) {
  Node pred, curr, succ; int key = object.hashCode();
  boolean[] marked = { false }; boolean b;
  retry: while (true) {
    pred = head; curr = pred.next.getReference();
    while (true) {                                   Move down the list
      succ = curr.next.get(marked);
      while (marked[0]) { … }
      if ((curr.key == key && curr.object == object)
          || curr.key > key)
        return new Window(pred, curr);
      pred = curr; curr = succ;
    }
  }
}
```

Lock-Free Find

```
public Window find(Node head, T object) {
 Node pred, curr, succ; int key = object.hashCode();
 boolean[] marked = { false }; boolean b;
 retry: while (true) {
  pred = head; curr = pred.next.getReference();
  while (true) {
   succ = curr.next.get(marked);
   while (marked[0]) { … }
   if ((curr.key == key && curr.object == object)
       || curr.key > key)
    return new Window(pred, curr);
   pred = curr; curr = succ;
  }
 }
}
```

Get successor and mark

Lock-Free Find

```
public Window find(Node head, T object) {
  Node pred, curr, succ; int key = object.hashCode();
  boolean[] marked = { false }; boolean b;
  retry: while (true) {
    pred = head; curr = pred.next.getReference();
    while (true) {
      succ = curr.next.get(marked);
      while (marked[0]) { … }
      if ((curr.key == key && curr.object == object)
          || curr.key > key)
        return new Window(pred, curr);
      pred = curr; curr = succ;
    }
  }
}
```

Try to remove deleted nodes

Lock-Free Find

```
public Window find(Node head, T object) {
  Node pred, curr, succ; int key = object.hashCode();
  boolean[] marked = { false }; boolean b;
  retry: while (true) {
    pred = head; curr = pred.next.getReference();
    while (true) {
      succ = curr.next.get(marked);
      while (marked[0]) { … }
      if ((curr.key == key && curr.object == object)
          || curr.key > key)
        return new Window(pred, curr);
      pred = curr; curr = succ;
    }
  }
}
```

If found object or greater key, return pred and curr

Lock-Free Find

```
public Window find(Node head, T object) {
 Node pred, curr, succ; int key = object.hashCode();
 boolean[] marked = { false }; boolean b;
 retry: while (true) {
  pred = head; curr = pred.next.getReference();
  while (true) {
   succ = curr.next.get(marked);
   while (marked[0]) { … }
   if ((curr.key == key && curr.object == object)
       || curr.key > key)
    return new Window(pred, curr);
   pred = curr; curr = succ;
  }
 }
}
```

Otherwise advance window and loop again

Lock-Free Find

```
…
while (marked[0]) {
  b = pred.next.compareAndSet(curr, succ, false, false);
  if (!b) continue retry;
  curr = succ;
  succ = curr.next.get(marked);
}
…
```

Lock-Free Find

Try to snip out node

```
…
while (marked[0]) {
  b = pred.next.compareAndSet(curr, succ, false, false);
  if (!b) continue retry;
  curr = succ;
  succ = curr.next.get(marked);
}
…
```

Lock-Free Find

```
…
while (marked[0]) {
  b = pred.next.compareAndSet(curr, succ, false, false);
  if (!b) continue retry;
  curr = succ;
  succ = curr.next.get(marked);
}
…
```

If predecessor's next field changed must retry whole traversal

Lock-Free Find

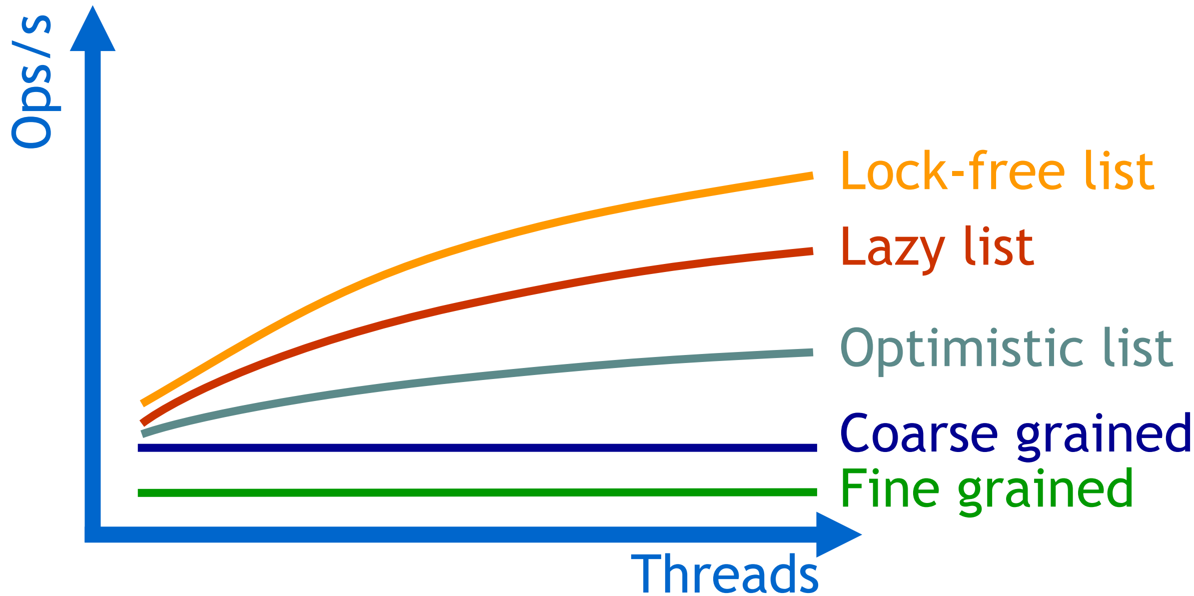
```
…
while (marked[0]) {
  b = pred.next.compareAndSet(curr, succ, false, false);
  if (!b) continue retry;
  curr = succ;
  succ = curr.next.get(marked);
}
…
```

Otherwise move on to check if next node deleted

Summary: Lock-Free List

- AtomicMarkableReference atomically updates mark and reference
    - Prevents manipulation of logically-removed next pointer
- Lock-free add() and remove()
    - Remove performs logical removal, may leave node
- Lock-free find() traverses both marked and removed nodes
    - Physically clean up (remove) marked nodes

Performance



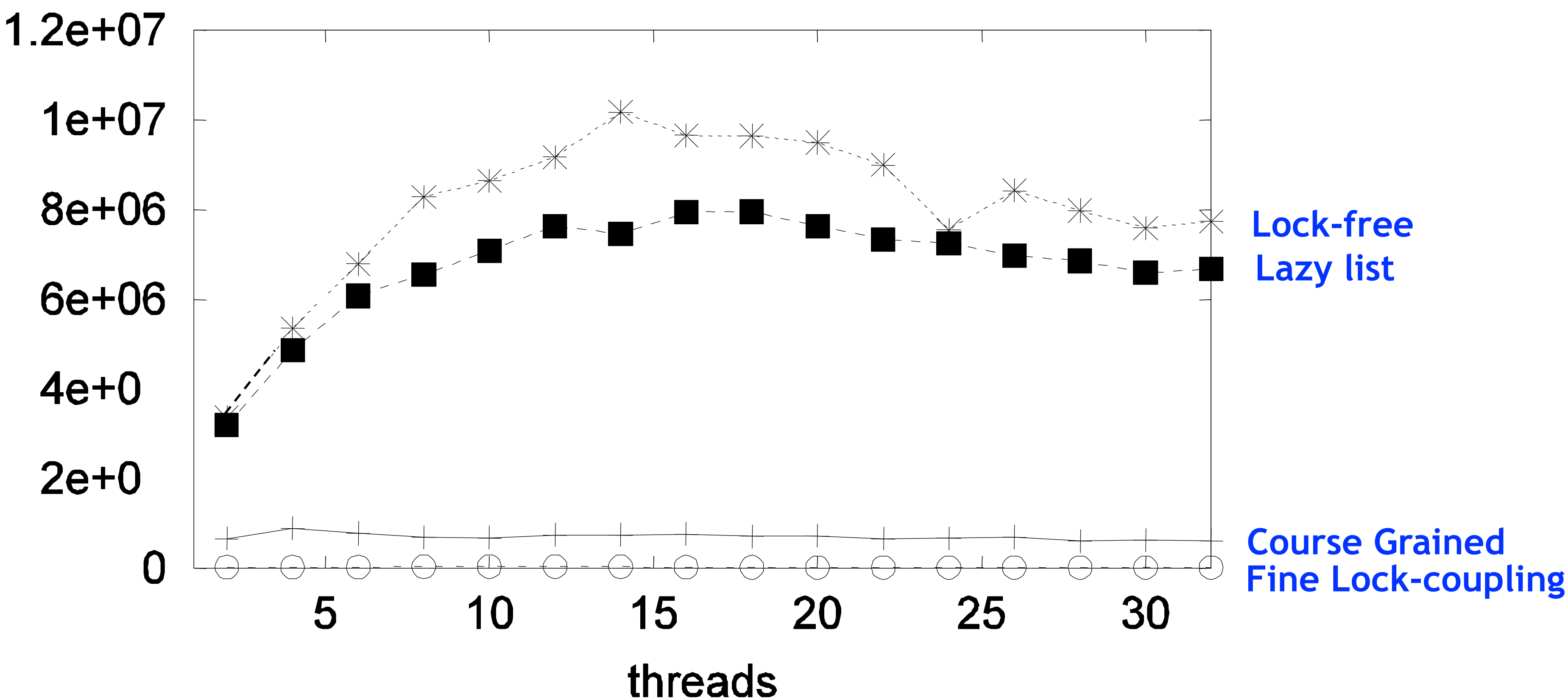
Performance

On 16 node shared memory machine

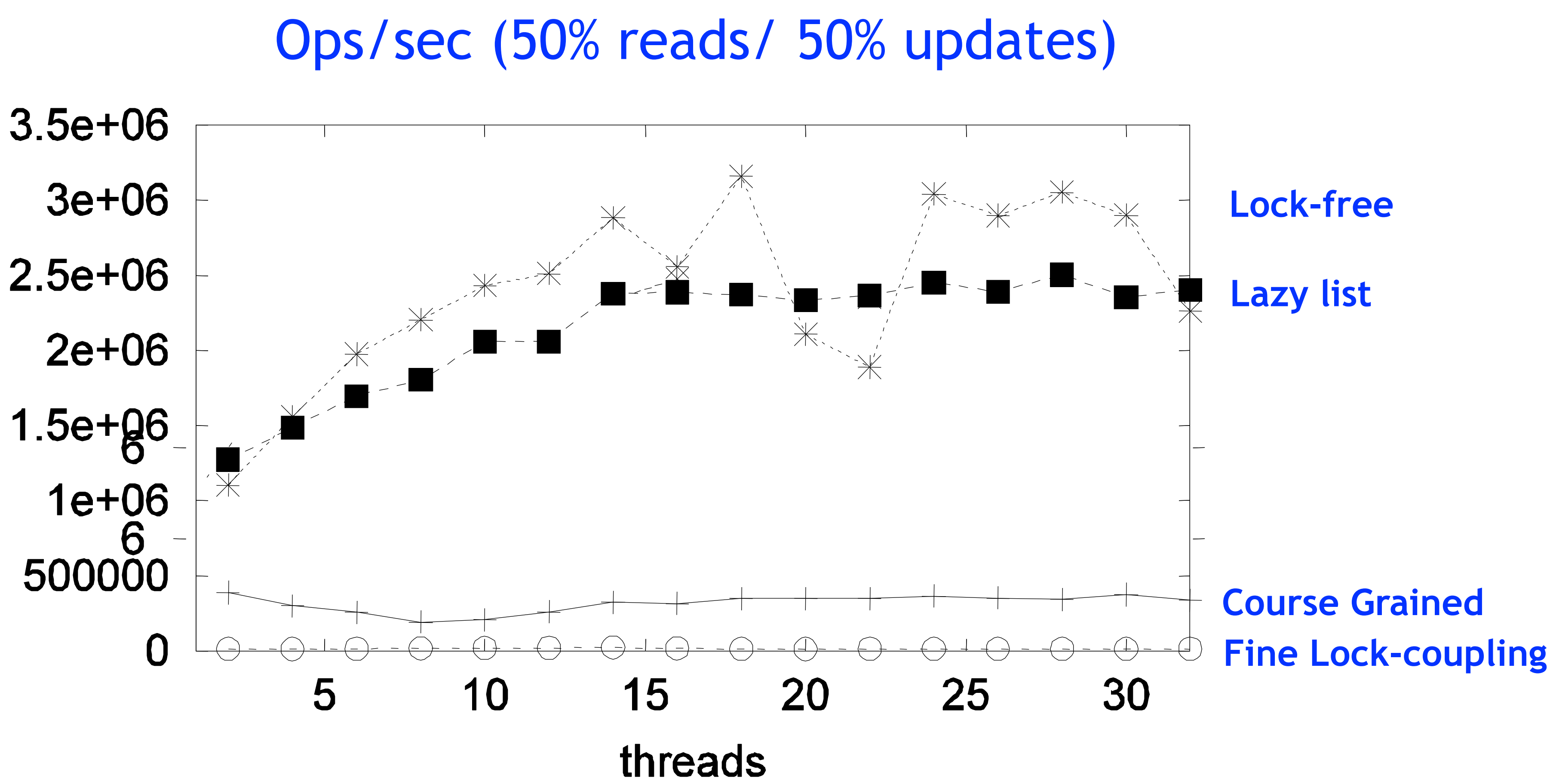
Benchmark throughput of Java List-based Set algs. Vary % of Contains() method Calls.

High Contains Ratio

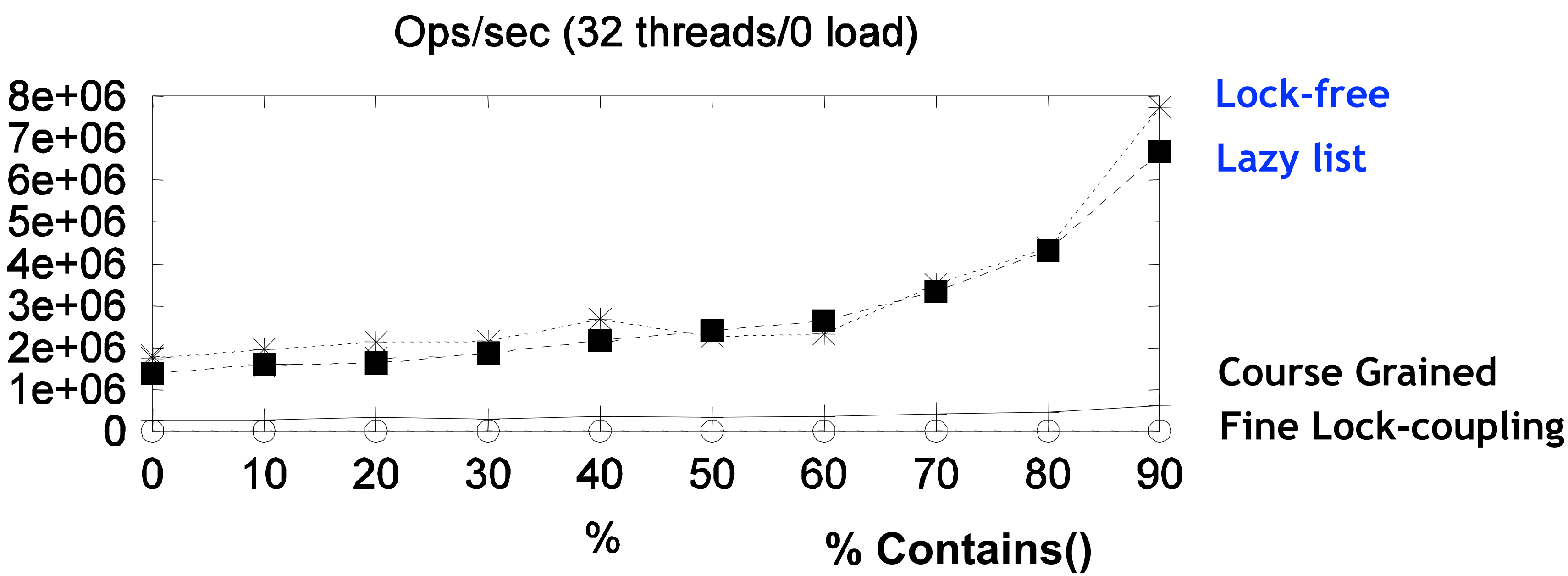
Ops/sec (90% reads/ 10% updates)



Low Contains Ratio



As Contains Ratio Increases



Summary

- Four “generic” approaches to concurrent data structure design
  - Fine-grained locking
  - Optimistic synchronization
  - Lazy synchronization
  - Lock-free synchronization

"To Lock or Not to Lock"

- Locking vs. non-blocking
  - Extremist views on both sides
- Nobler to compromise, combine locking and non-blocking
  - Example: Lazy list combines blocking add() and remove() and a wait-free contains()
  - Blocking/non-blocking is a property of a method