Concurrency: Multi-core Programming & Data Processing

Linked Lists: Locking, Lock-Free, and Beyond

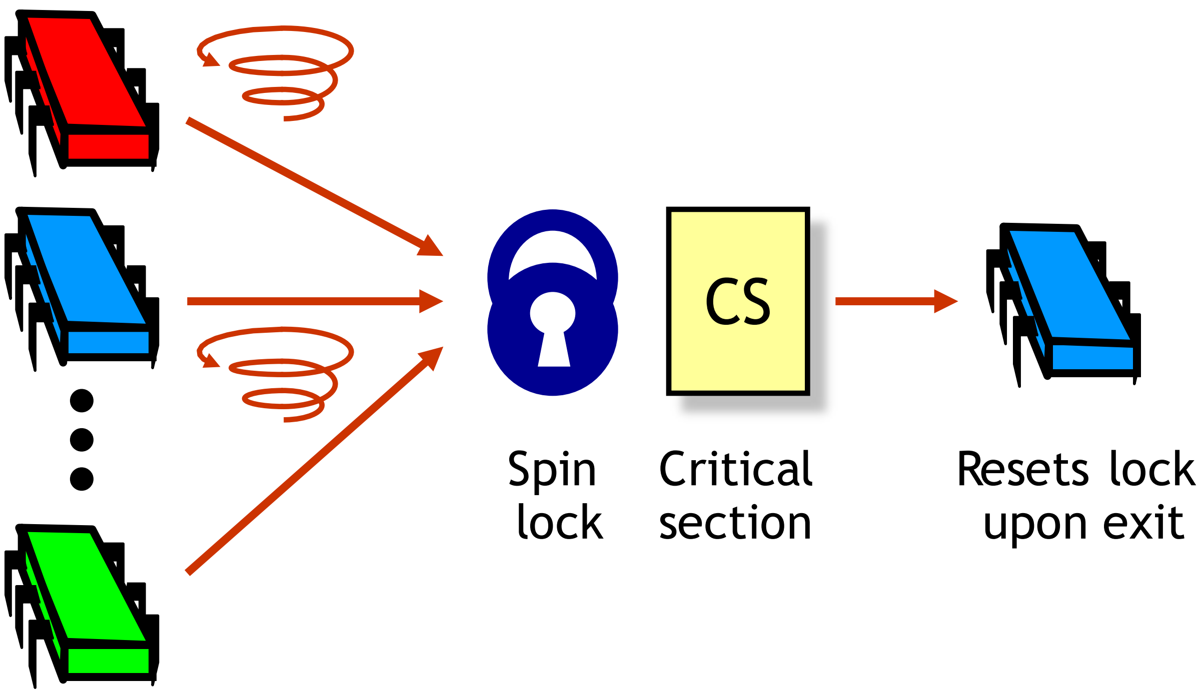
Prof. P. Felber
Pascal.Felber@unine.ch
http://iiun.unine.ch/

Based on slides by ***Maurice Herlihy*** *and* ***Nir Shavit***

Institut d'informatique
Université de Neuchâtel

Since you link the nodes in a linear fashion. Intuitively there is only limited thing you can do concurrently

Last Lecture: Spin-Locks



Today: Concurrent Objects

- Adding threads…

- Should not lower throughput
  - Contention effects
  - Mostly fixed by queue locks

- Should increase throughput
  - Not possible if inherently sequential
  - Surprising things are parallelizable

Only possible if app is not inherently sequential

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

e.g. central db is bottleneck. we want this to be highly parallel

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

Try to optimistically go through structure. Only when you find what you're looking for, you lock. You need to do some checks on what you have locked. Only makes sense when that check is less expensive than locking.

3. Lazy Synchronization

Probably the most efficient

- 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

The most parallel. Scales well. No locks. No mutex. Nothing to slow you down.

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

Linked List

Insert/Search/Remove

Implement a set using a list. Set with no duplicated and sorted.

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

Set Interface

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

List-Based Sets

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

Add object to set

Remove object from set

Is object in set?

List Node

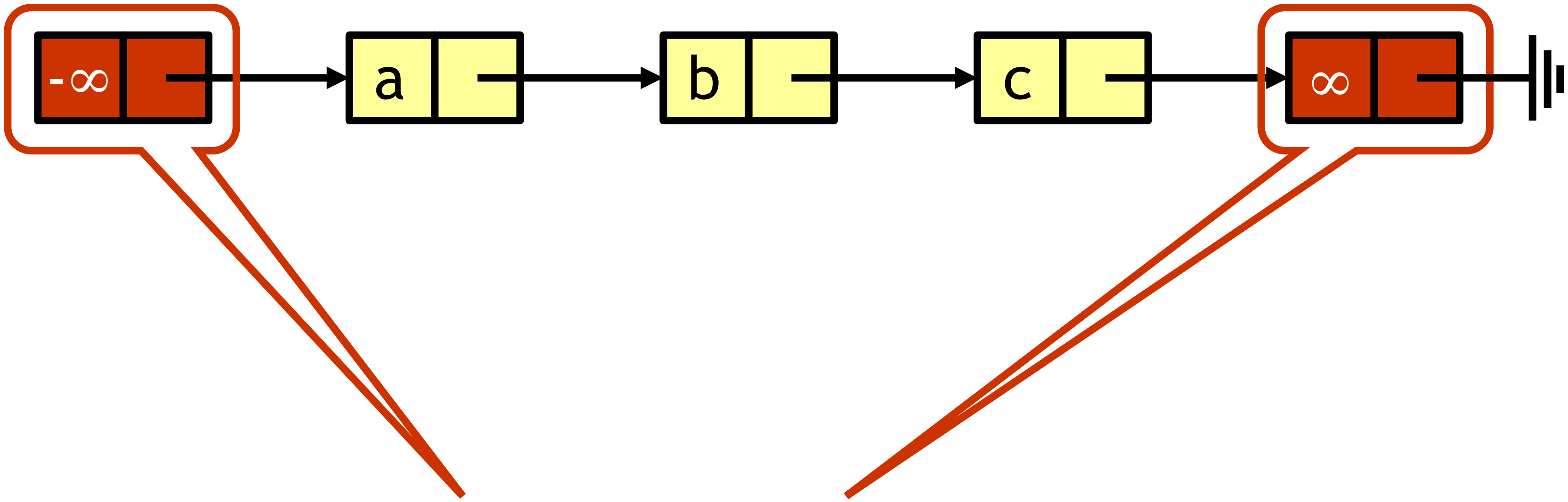
```
public class Node {
  Object object;
  int key;
  Node next;
}
```

Object of interest

Usually hash code

Reference to next node

The List-Based Set



Ordered + sentinel nodes
(min & max possible keys)

they have a special key

Reasoning about Concurrent Sets

- 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

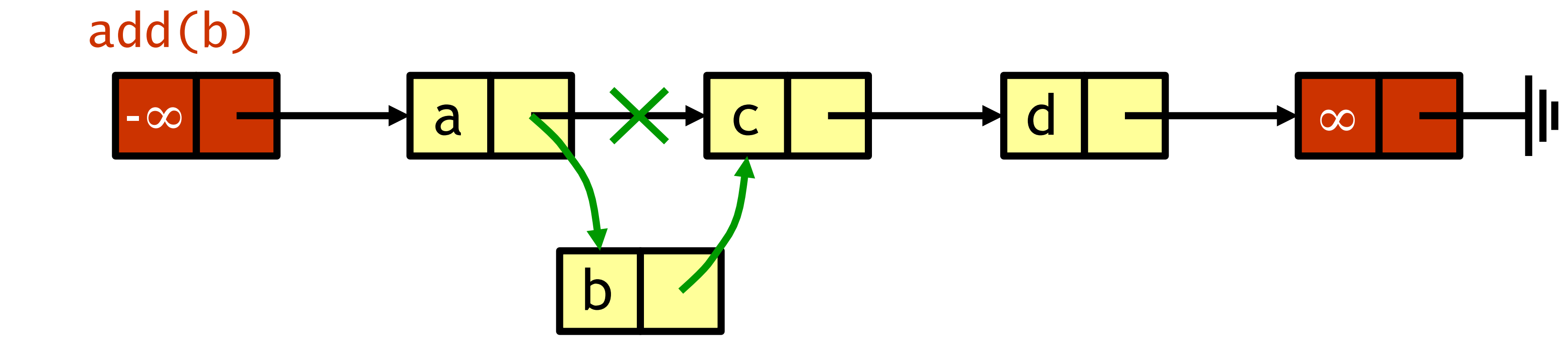
Blame Game

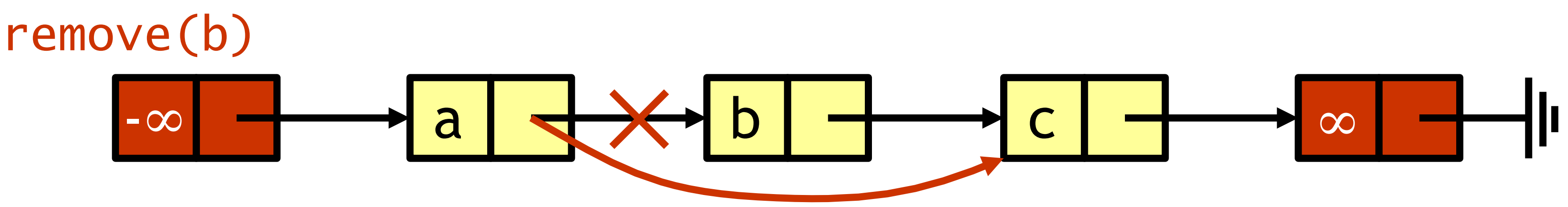
- Suppose
  - `add()` leaves behind 2 copies of `x`
  - `remove()` removes only 1
- Which one is incorrect?
  - If invariant says *no duplicates*
    - `add()` is incorrect
  - Otherwise
    - `remove()` is incorrect

Set Invariant (Partly)

- Sentinel nodes
  - Tail reachable from head
- Sorted according to hash key
- No duplicates

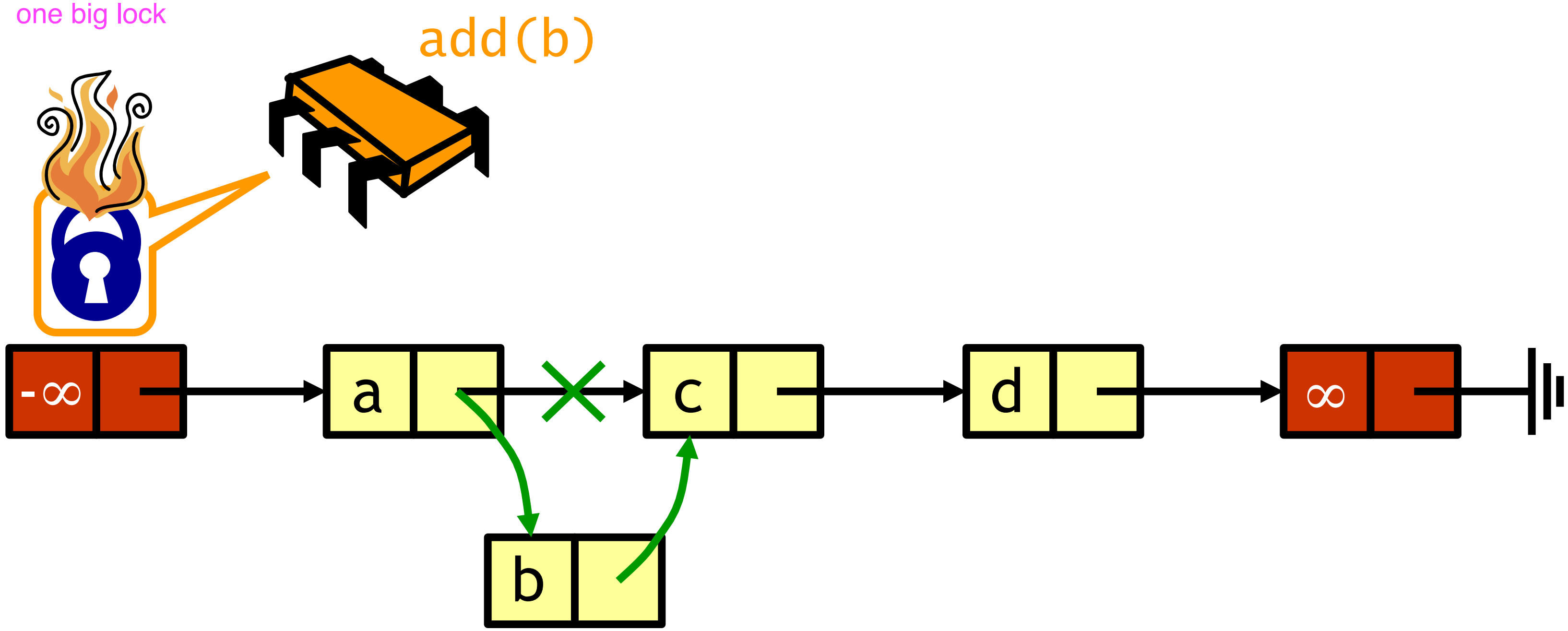
Sequential List Based Set





Coarse-Grained Locking

We protect the whole list with one big lock



Simple but hotspot + bottleneck

No parallelism

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

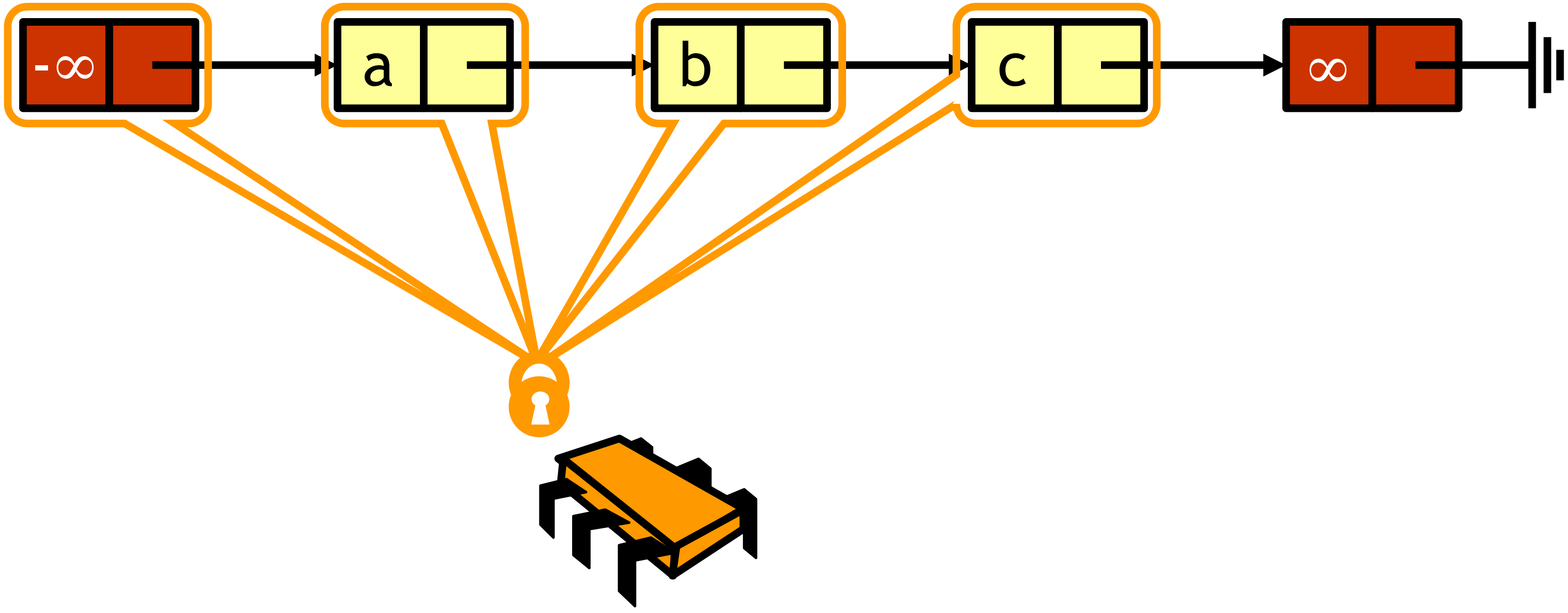
- Requires careful thought
  - "Do not meddle in the affairs of wizards, for they are subtle and quick to anger"
- Split object into pieces
  - Each piece has own lock
  - Methods that work on disjoint pieces need not exclude each other

For instance, lock individual nodes. It's not very easy to do. You need to hold multiple locks in the right order. Deadlock: Each thread has one lock and is waiting for other threads to release locks.

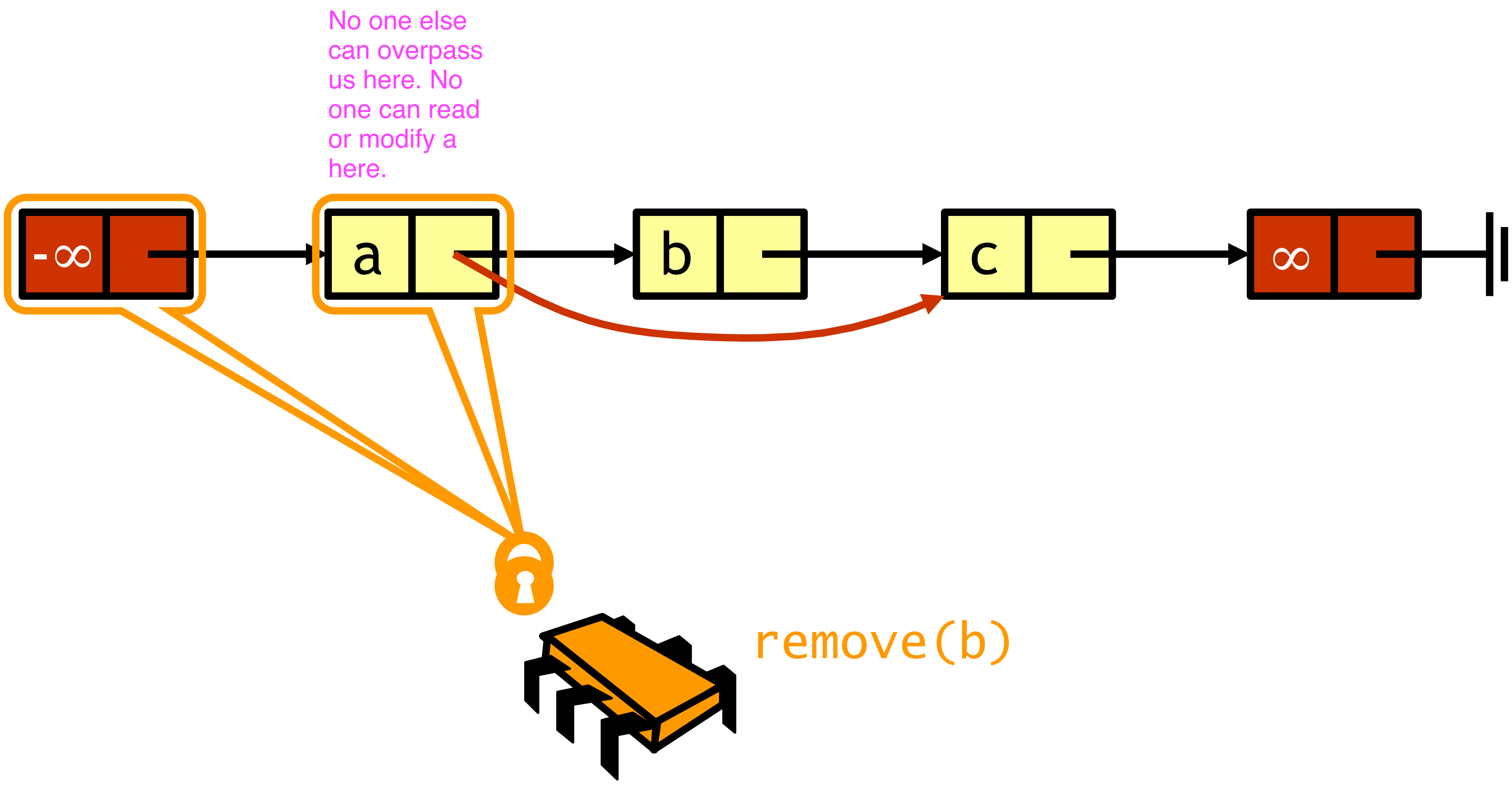
Hand-over-Hand Locking

You need to make sure that you always keep at least one lock.

Felber Hand over movement

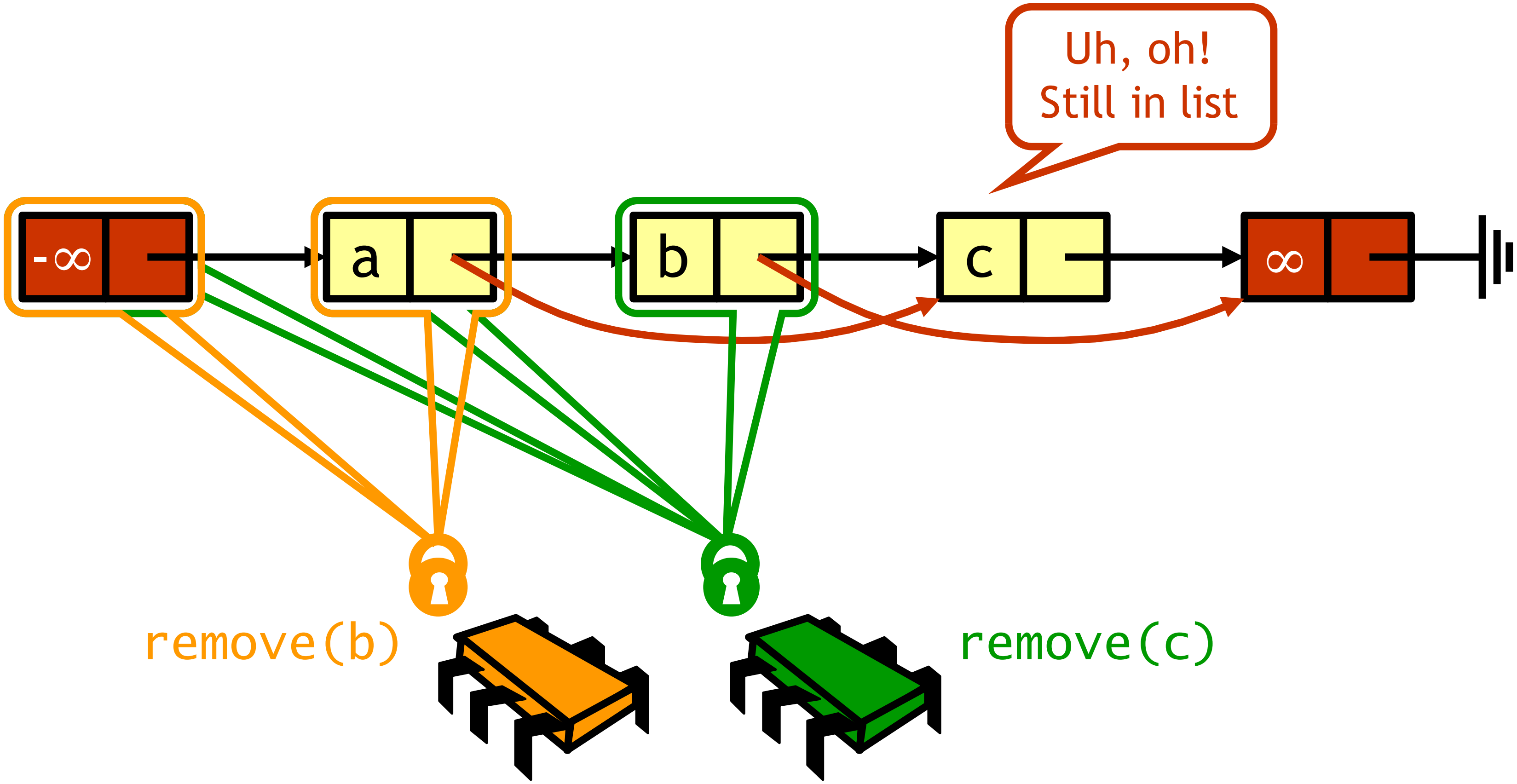


Removing a Node



Removing a Node

Two threads removing at the same time

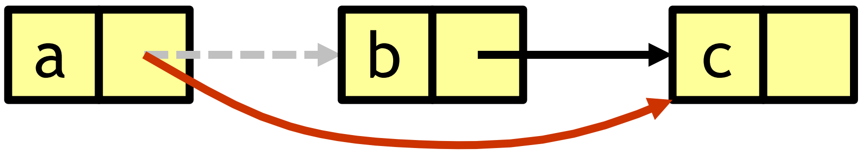


c has not been removed

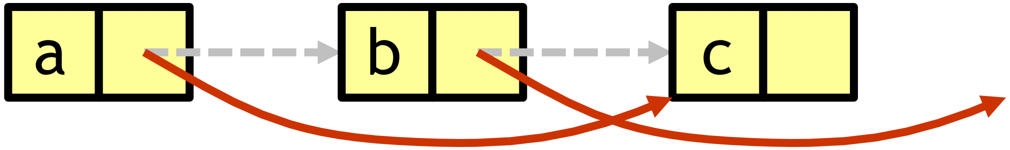
Result: start -> a -> c -> end

Problem

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



- Problem is
  - Someone could delete c concurrently



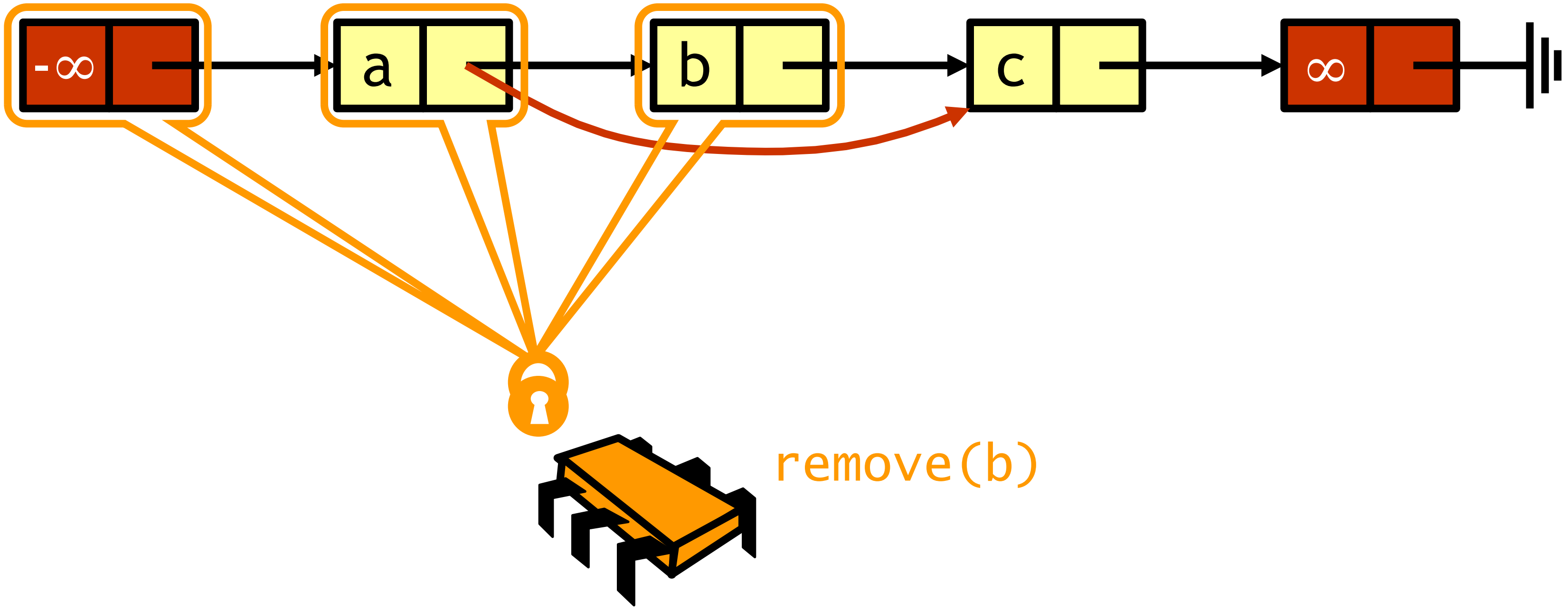
so we need to hold more than a single lock to make this modification

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

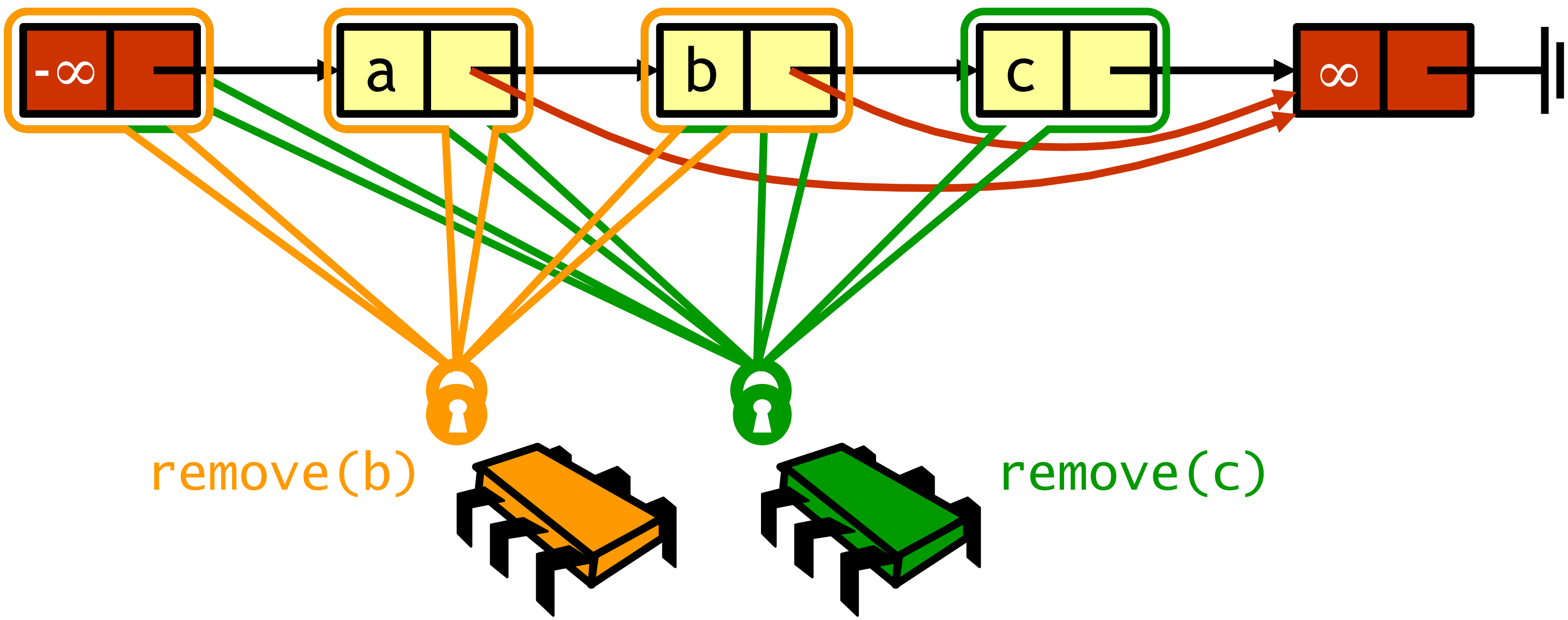
Hand-over-Hand Again

so we lock a AND b



Hand-over-Hand Again

see power point for animation



Remove

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

Key used to order node

Predecessor and current nodes

Everything else

Make sure locks released

Remove

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

Lock previous

Lock current

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;
```

Search key range (curr and pred locked)

If node found, remove it

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

Find and lock new current

Otherwise not present

Adding Nodes

- To add node b
  - Lock predecessor
  - Lock successor

lock around insertion point

- Neither can be deleted
  - (Is successor lock actually required?)

no, not necessarily. it's optional

Drawbacks

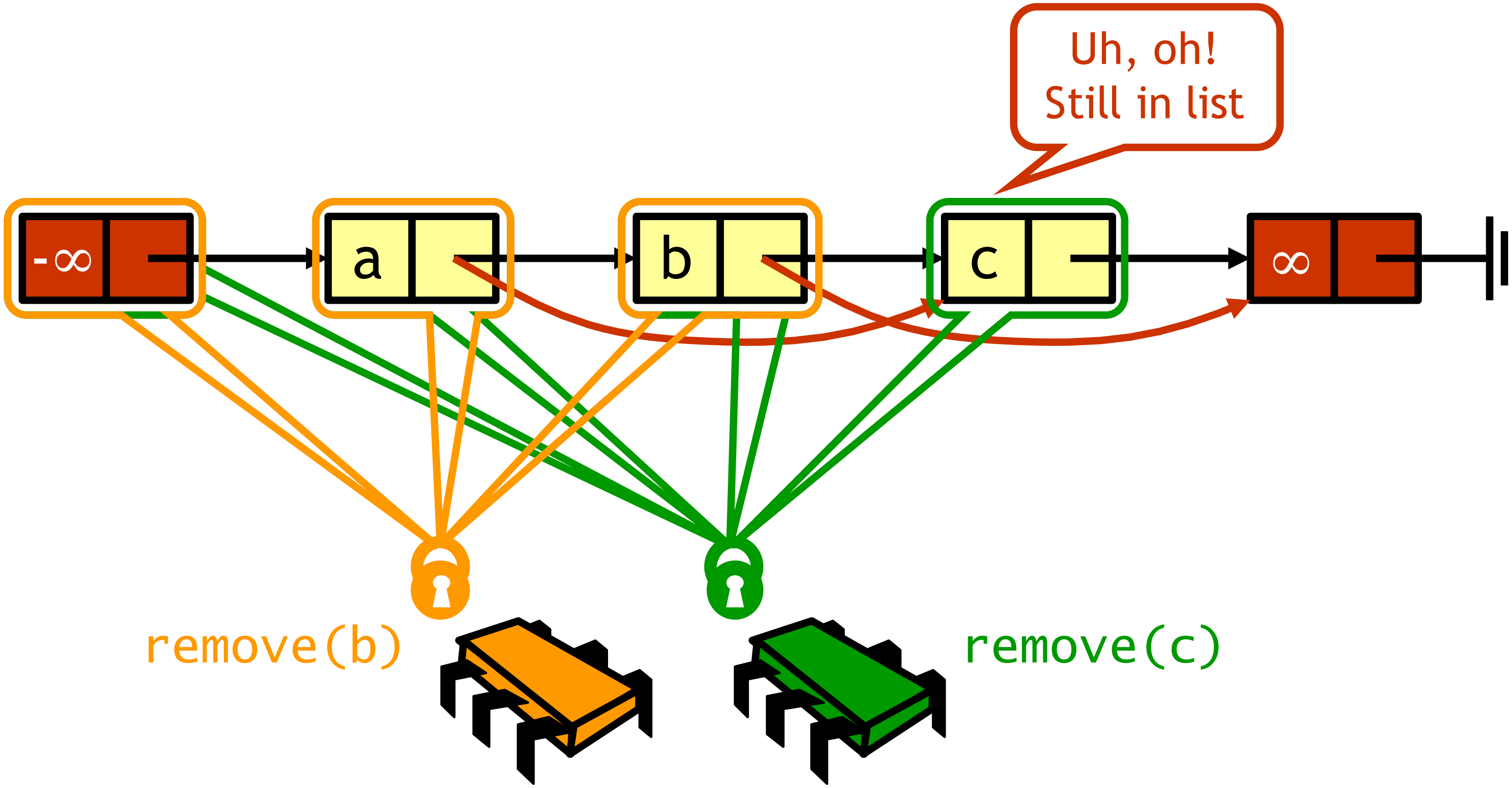
- Better than coarse-grained lock
  - Threads can traverse in parallel
- Still not ideal
  - Long chain of acquire/release
  - Threads cannot overtake one another
  - Inefficient

Optimistic Synchronization

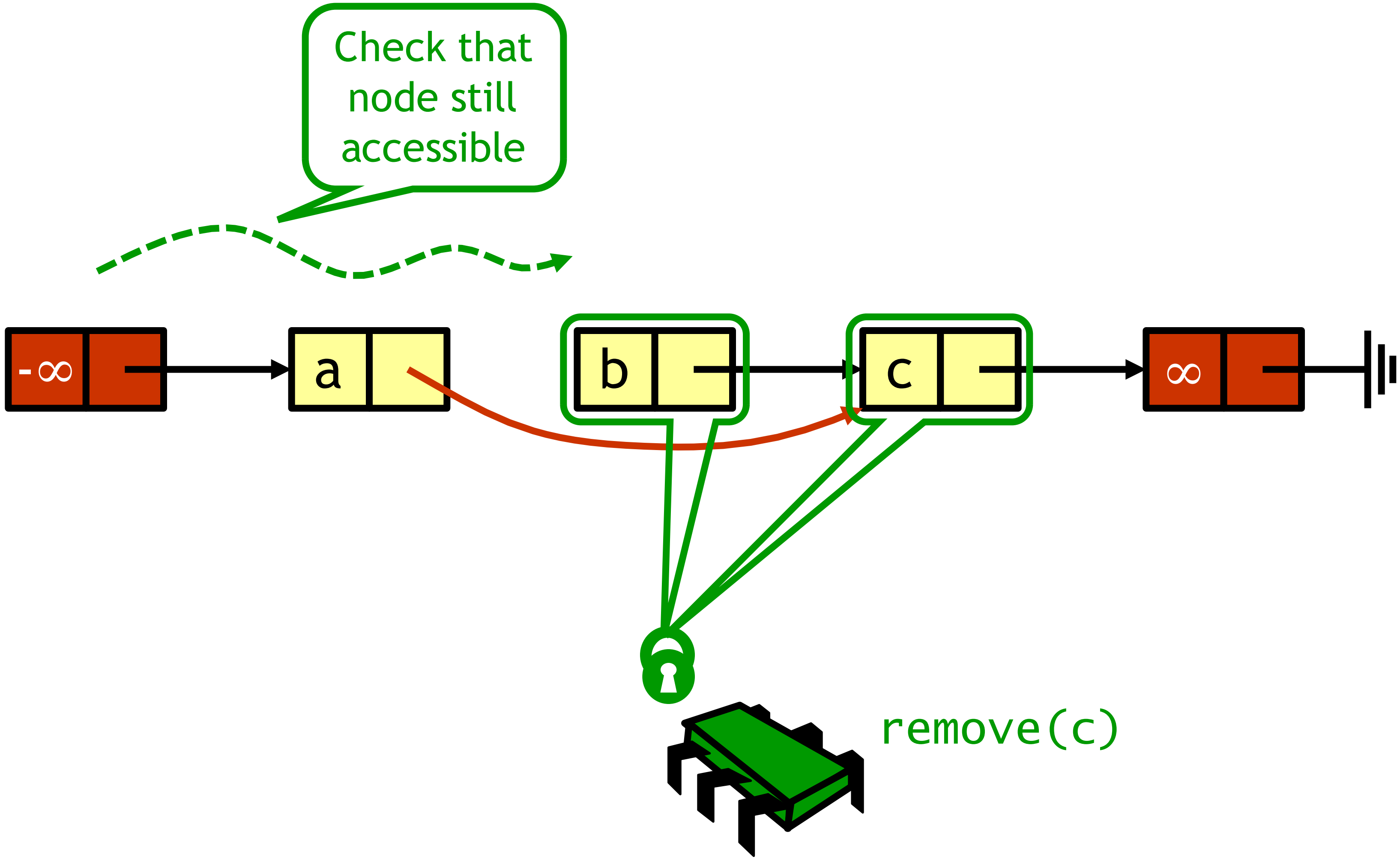
- Find nodes without locking
- Lock nodes
- Check that everything is OK

What Could Go Wrong?

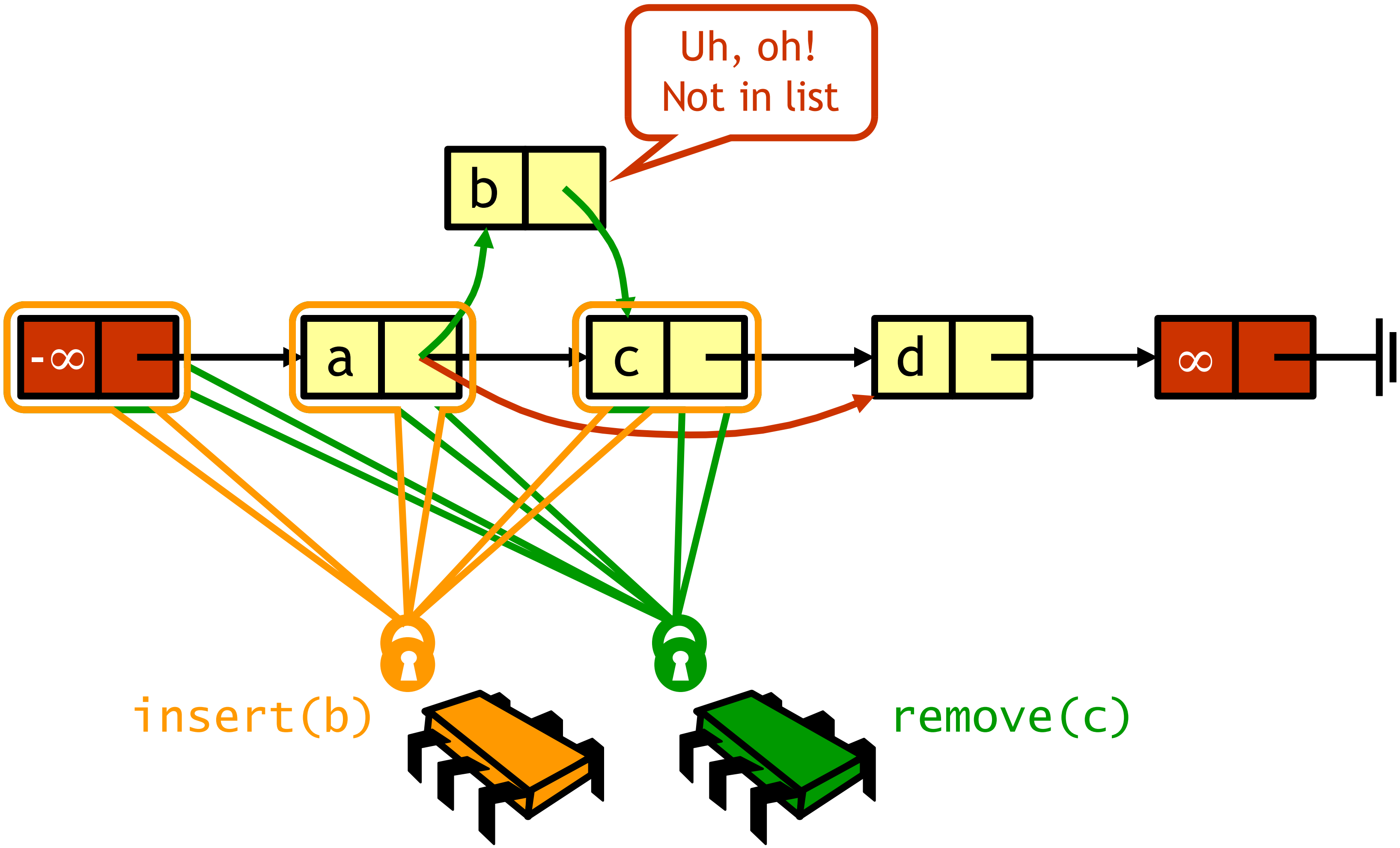
See powerpoint for animation



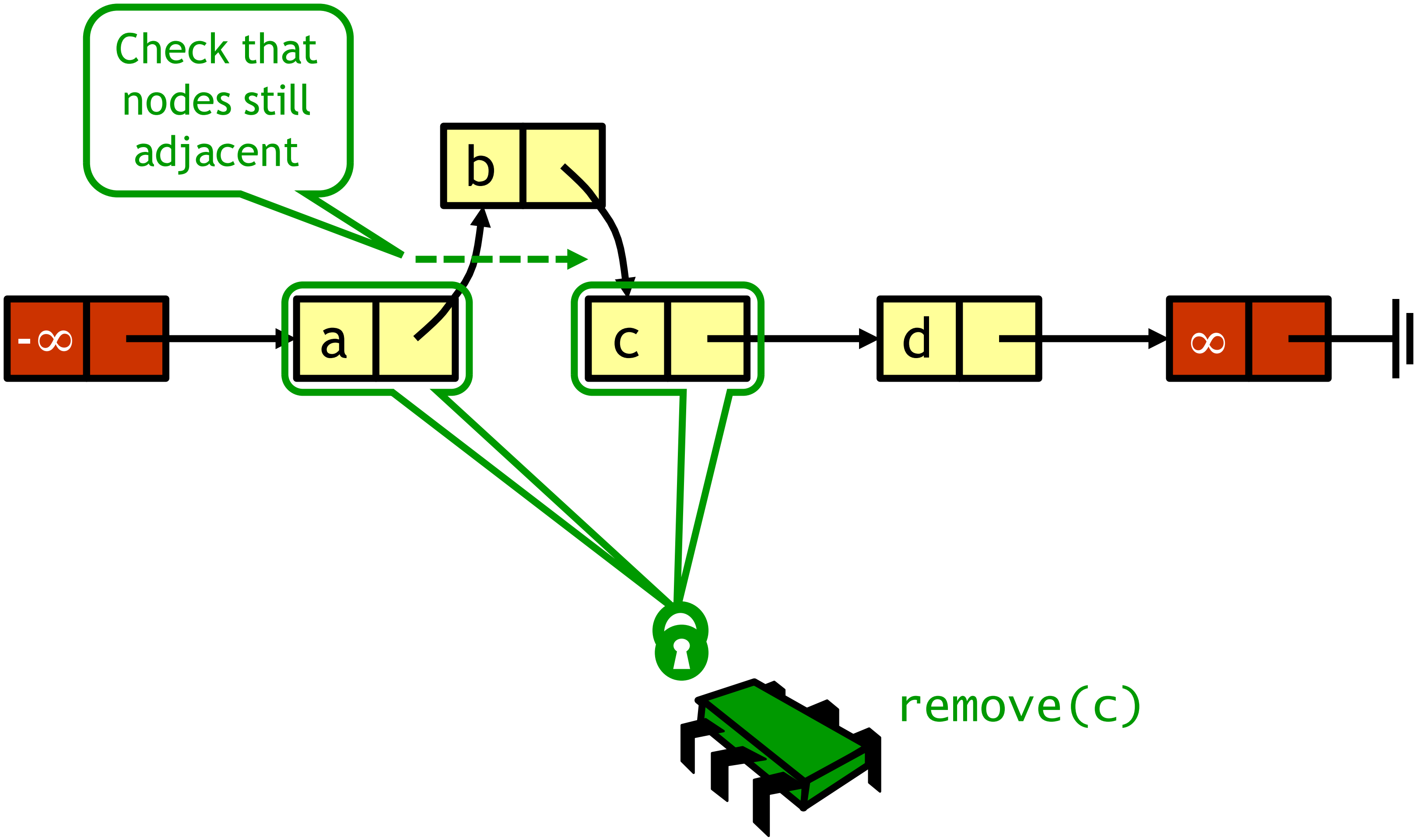
What Could Go Wrong?



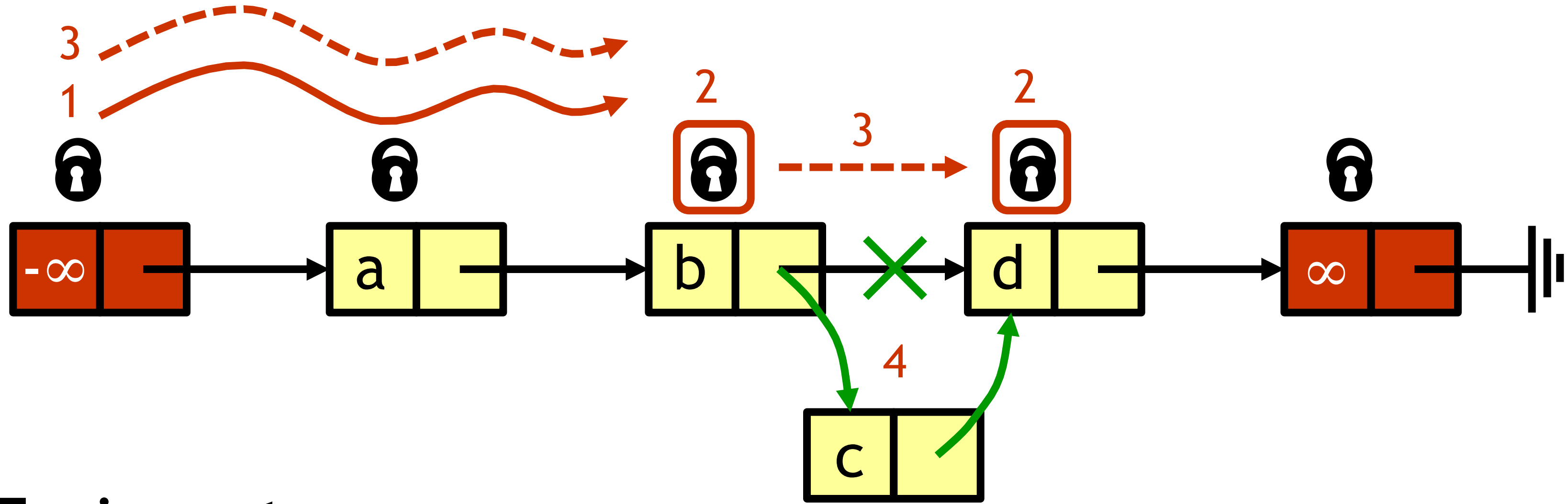
What Could Go Wrong?



What Could Go Wrong?



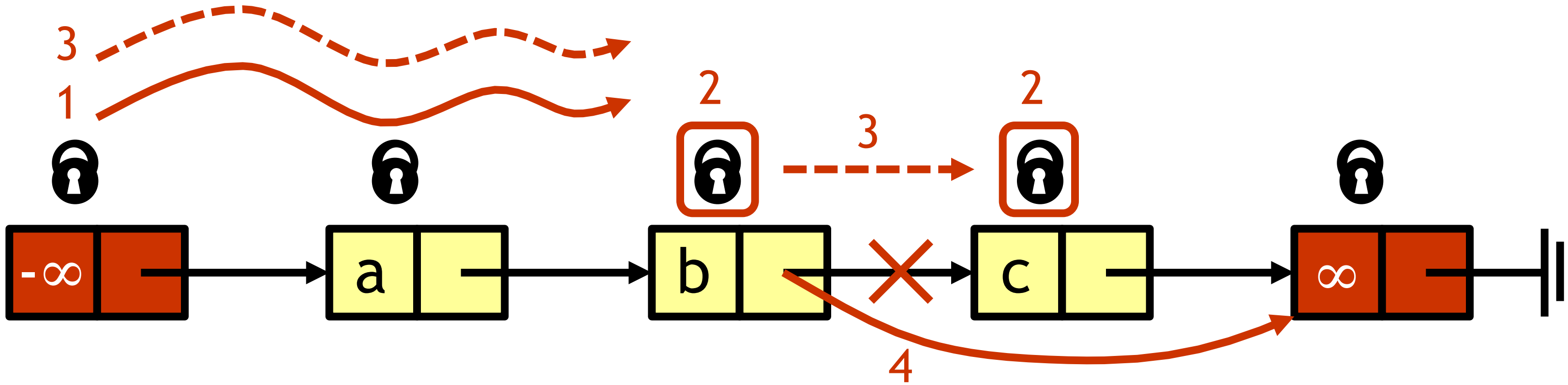
Optimistic Fine Grained



- To insert c
  1. Optimistically traverse list to find b
  2. Lock `b.curr` then lock `b.succ`
  3. Re-Traverse list to find b and verify `b.curr` precedes `b.succ`

     this means b hasn't been deleted, and successor is also there
  4. Perform insertion and release locks

Optimistic Fine Grained removal

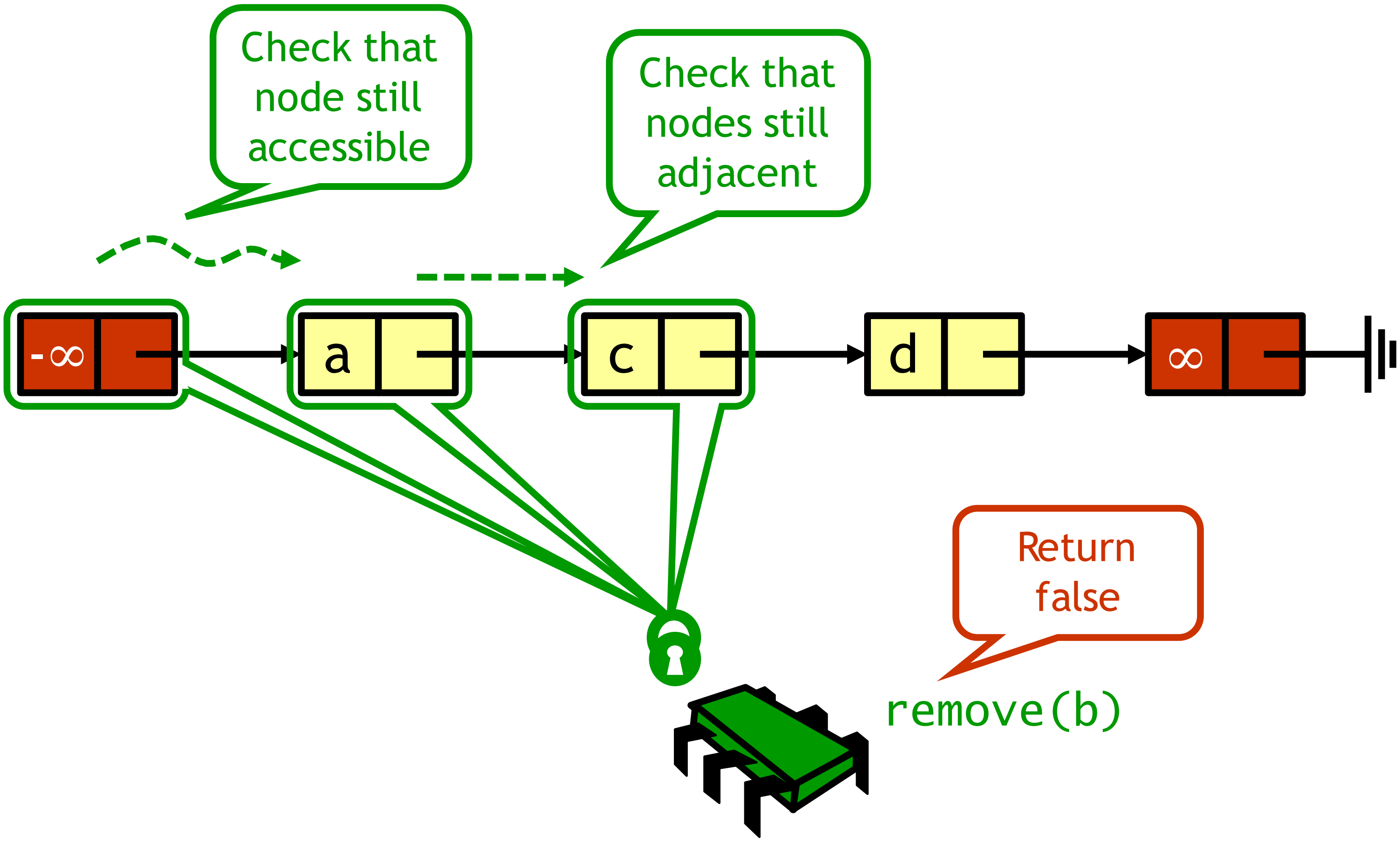


- To remove c
  1. Optimistically traverse list to find c
  2. Lock `c.pred` then lock `c.curr`
  3. Re-Traverse list to find c and verify that `c.pred` precedes `c.curr`
  4. Perform removal and release locks

Correctness

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

Removing an Absent Node



Correctness

- If
  - Nodes a and c both locked
  - Node a still accessible
  - Node c still successor to a
- Then
  - Neither will be deleted
  - No thread can add b after a
  - 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;
}
```

Predecessor & current nodes

Start at the beginning

Search range of keys

Predecessor reachable?

Current node next?

Otherwise move on

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;
    }
    …
```

Search key

Retry on synchronization conflict

Examine predecessor and current nodes

Search by key

Stop if we find object

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();
} …
```

Lock both nodes

Check for synchronization conflicts

noone has inserted/removed nodes

Object found, remove node

Object not found

Always unlock

Summary: Optimistic List

- Wait-free traversal   no thread blocks any other
  - 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   was good when traversing was cheaper than locks
  - 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

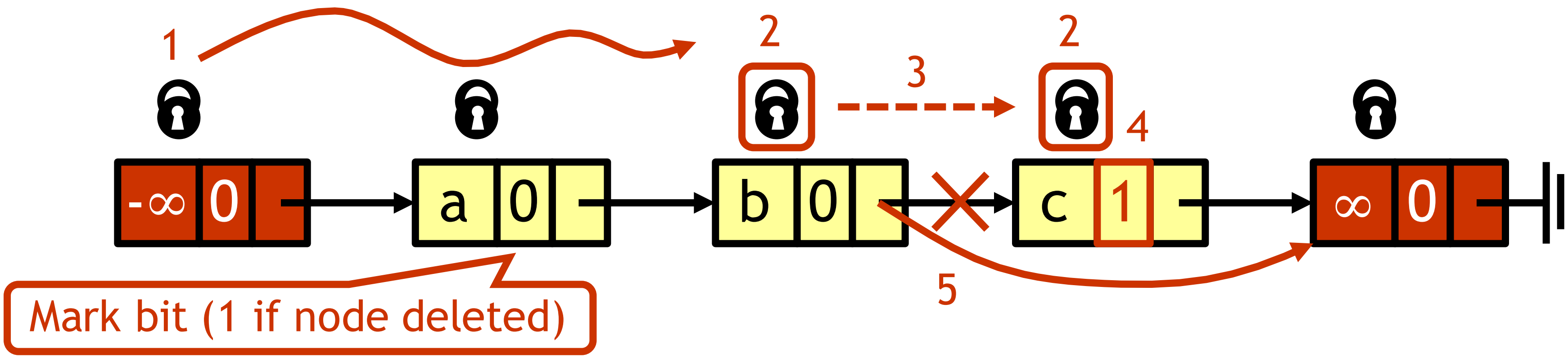
Lazy List

- 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

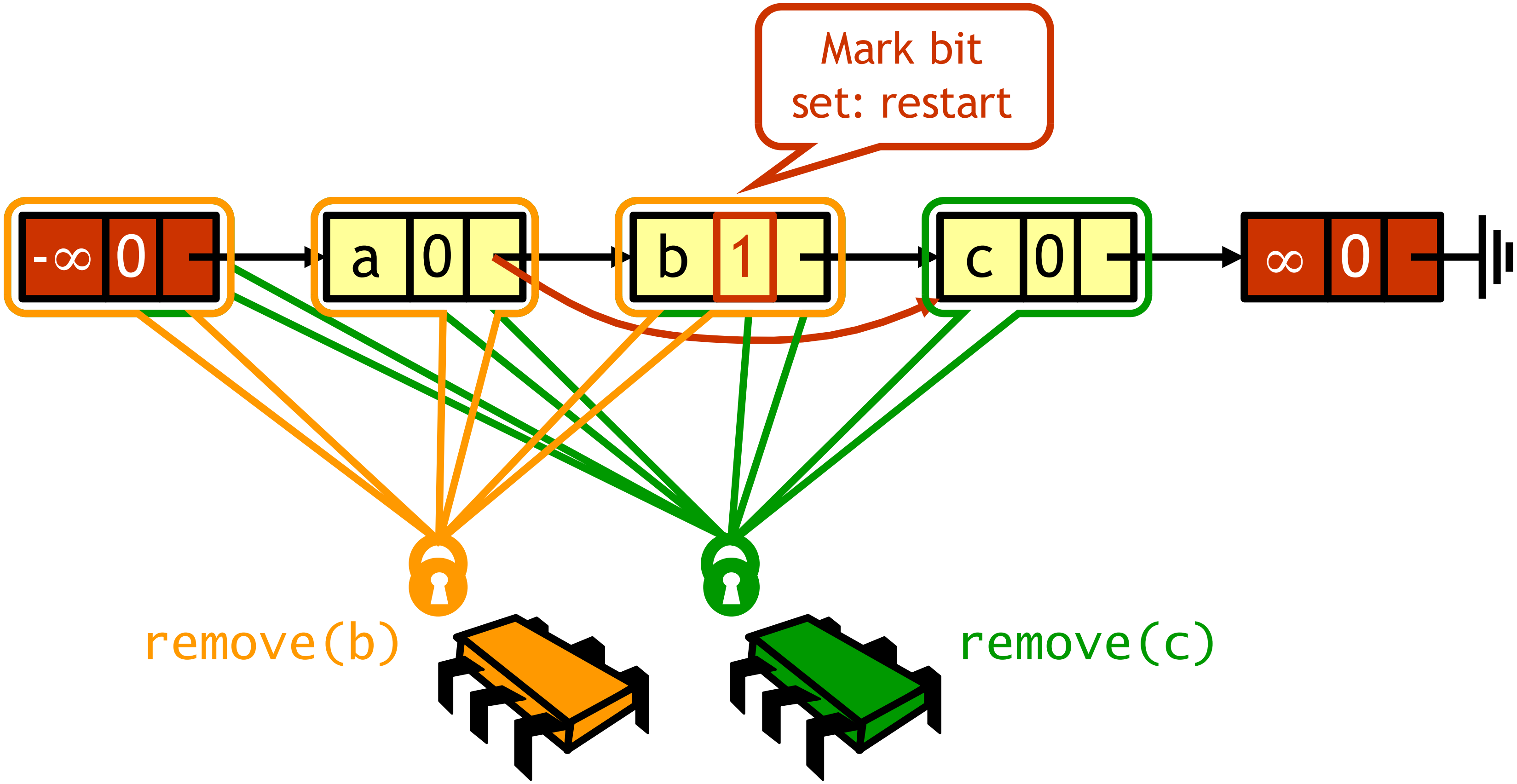


- To remove c
  1. Optimistically traverse list to find c
  2. Lock `c.pred` then lock `c.curr`
  3. Verify marks and that `c.pred` precedes `c.curr`
  4. Set mark bit (logical removal)
  5. Perform physical removal and release locks

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 for updates
- 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?



Validation

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

Predecessor not logically removed

Current not logically removed

Predecessor still points to current

Remove

```
try {
  pred.lock(); curr.lock();
  if (validate(pred, curr) {        Validate as before
    if (curr.object == object) {    Object found
      curr.marked = true;           Logical removal
      pred.next = curr.next;        Physical removal
      return true;
    } else return false;
  }
} finally {
  pred.unlock(); curr.unlock();
} …
```

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

Traverse without locking (nodes may have been removed)

Present and undeleted?

Summary: Lazy List

- Wait-free traversal uses mark bit + fact that list is ordered
  - Not marked $\Rightarrow$ in the set
  - Marked or missing $\Rightarrow$ 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, put aside by scheduler…
  - 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

practically interesting

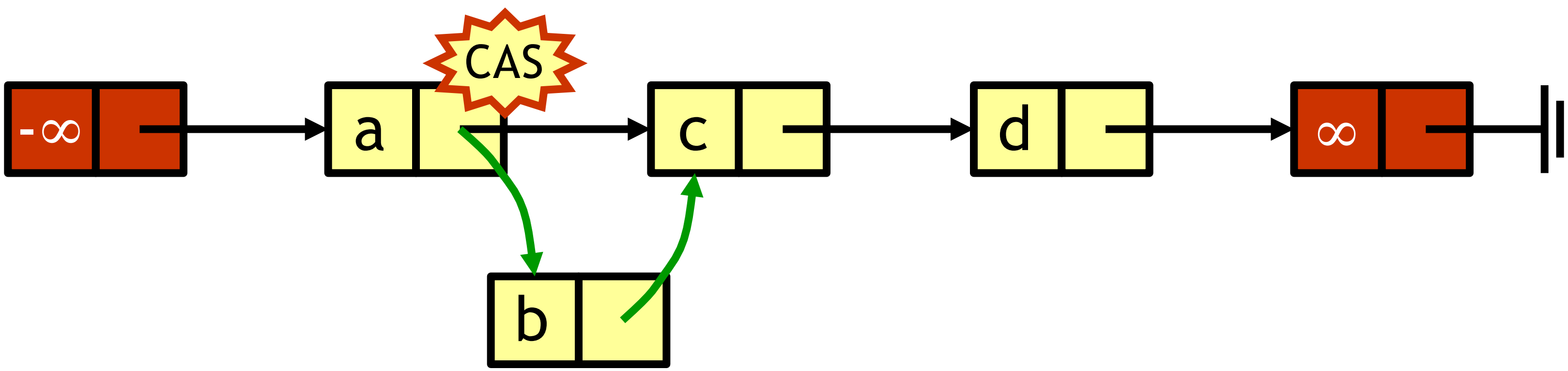
- No matter what…
  - Some thread will complete method call
  - Even if others halt at malicious times
  - Weaker than wait-free, yet
- Implies that
  - You cannot use locks
  - Um, that is why they call it lock-free

Lock-Free Lists

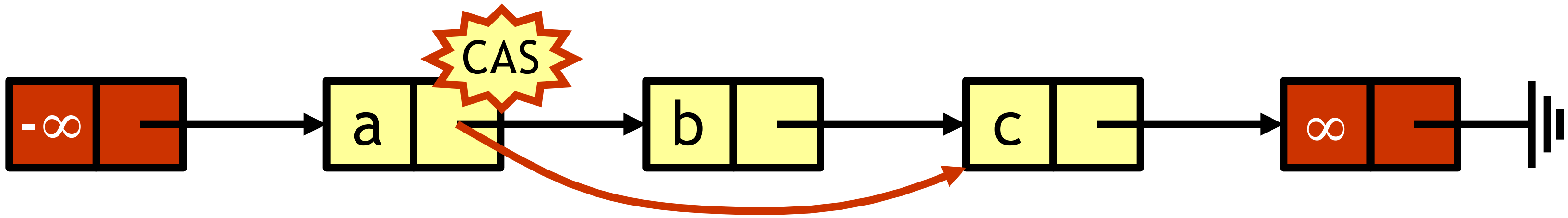
- Next logical step
- Eliminate locking entirely
- `contains()` wait-free and `add()` and `remove()` lock-free
- Use only `compareAndSet()` to atomically update links

Adding a Node

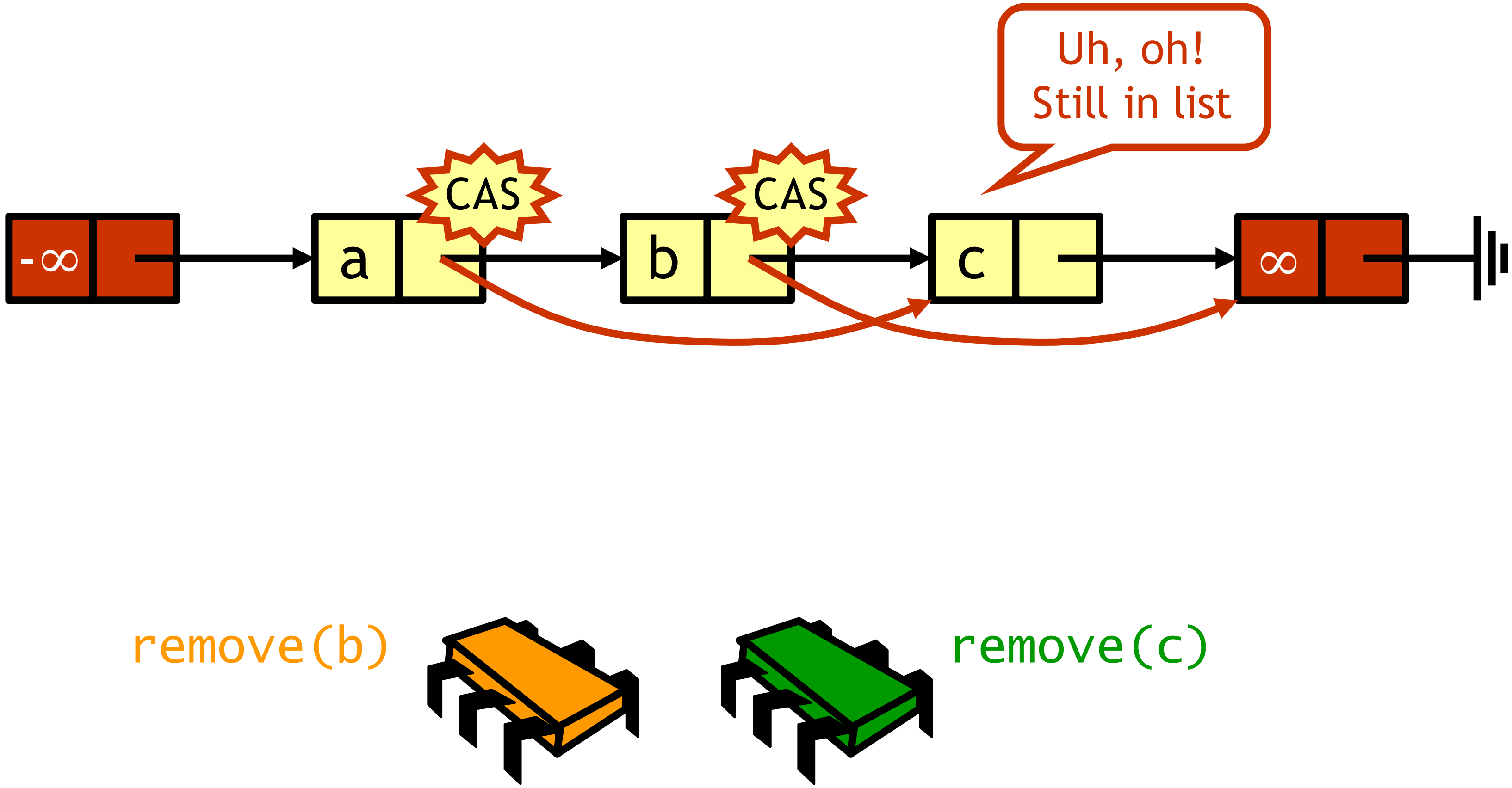
am a still pointing to c -> set atomic



Removing a Node



What Could Go Wrong?

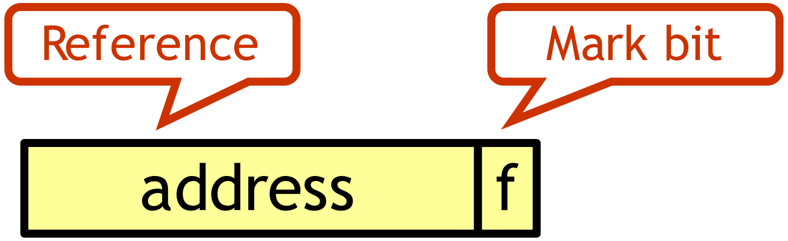


What Could Go Wrong?

- 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



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);
  …
}
```

Extract reference and mark (at index 0)

If this is the current reference…

…then change to this new reference…

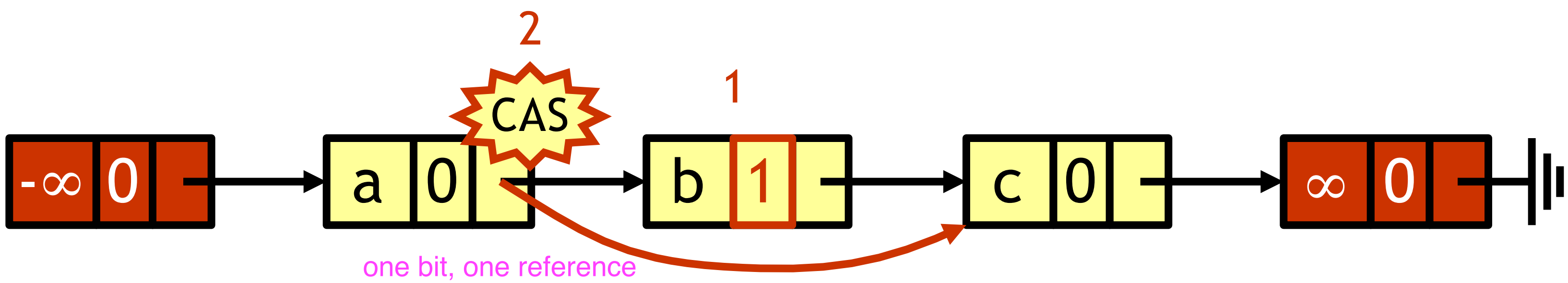
…and this is the current mark…

…and this new mark

If this is the current reference…

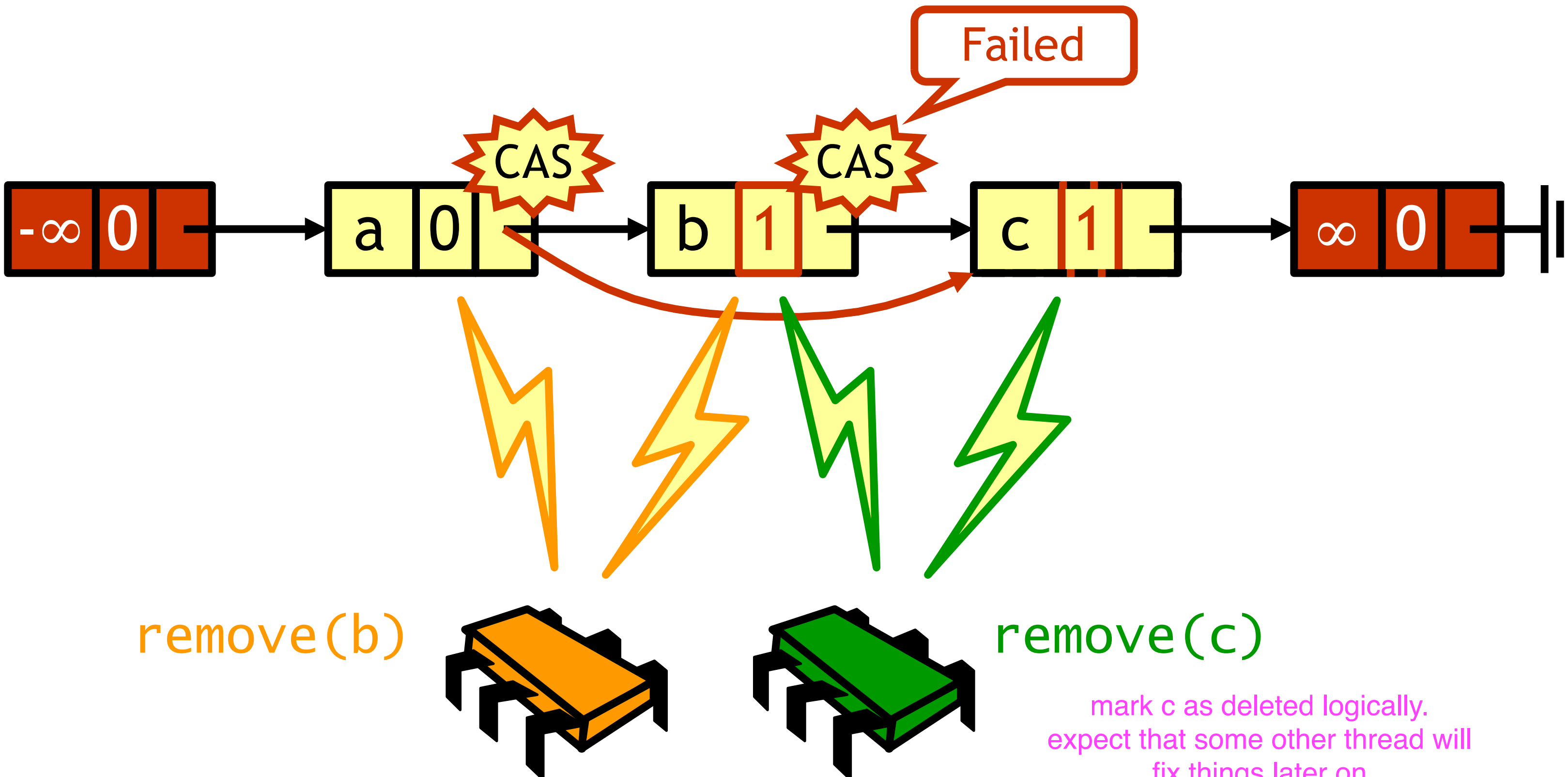
…then change to this new mark

Removing a Node



What Could Go Wrong?

mark has been set to 1. CAS expects 0

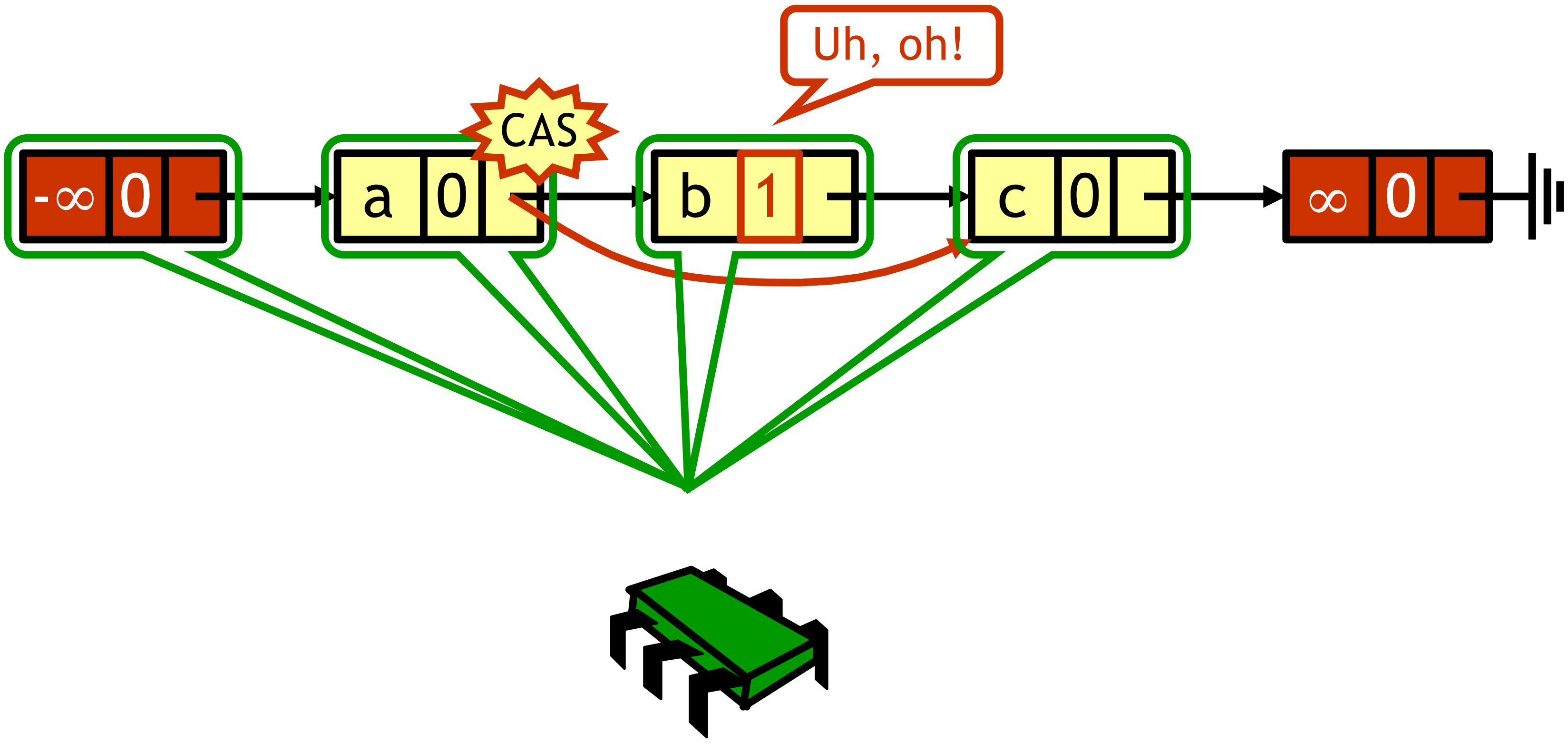


mark c as deleted logically.
expect that some other thread will
fix things later on

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



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;
…
```

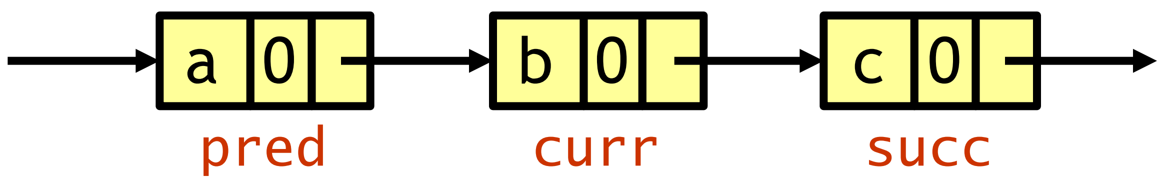
Find window

Extract pred and curr

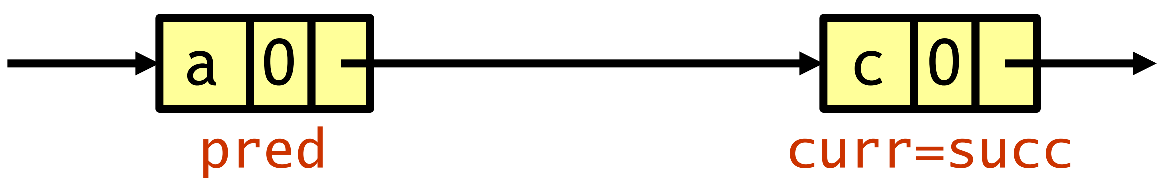
The Find Method

```
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.attemptMark(succ, true);
    if (!b) continue;
    pred.next.compareAndSet(curr, succ, false, false);
    return true;
  }
}
```

Keep trying

Find neighbors

Not there

Try to mark node as deleted

If it fails, retry, otherwise job done

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;
  }
}
```

Already there

Create new node

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]);
}
```

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;
    }
  }
}
```

Restart if list changes while traversed

Start from head

Move down the list

Get successor and mark

Try to remove deleted nodes

If found object or greater key, return pred and curr

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);
}
…
```

Try to snip out node

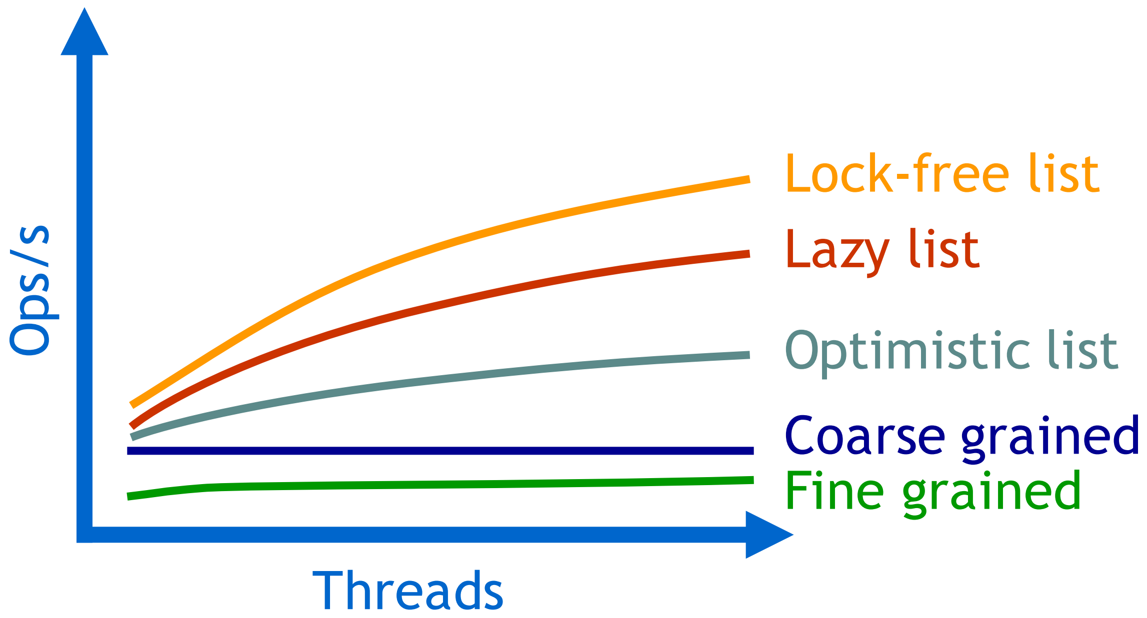
If predecessor's next field changed must retry whole traversal

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
- Wait-free `contains()` traverses both marked and removed nodes
- Physically remove marked nodes in `find()`

Performance



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