LOGO

Concurrency For Multicores

By Mohit Kumar

Contents

Amdahl's Law

Given a job, that is executed on n processors.

Let $p \in [0, 1]$ be the fraction of the job that can be parallelized (over n processors).

Let sequential execution of the job take 1 time unit.

Then parallel execution of the job takes $(1 - p) + \frac{p}{n}$ time units.

So the speed-up is

$$\frac{1}{(1 - p) + \frac{p}{n}}$$

Amdahl's Law

$n = 10$

$p = 0.6$ gives speed-up of $\frac{1}{0.4+\frac{0.6}{10}} = 2.2$

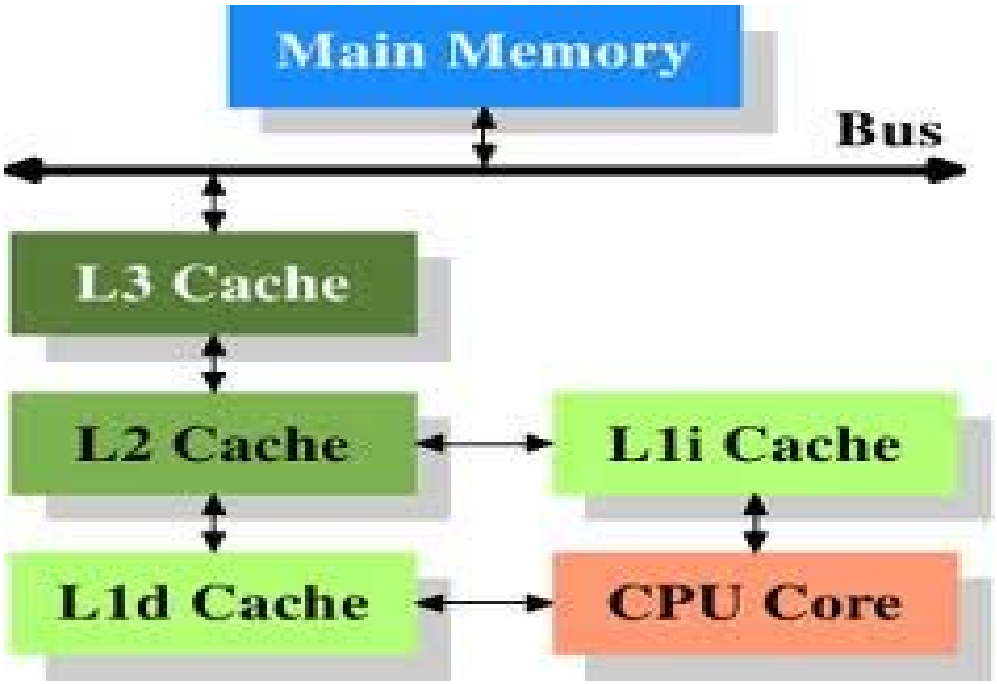
$p = 0.9$ gives speed-up of $\frac{1}{0.1+\frac{0.9}{10}} = 5.3$

$p = 0.99$ gives speed-up of $\frac{1}{0.01+\frac{0.99}{10}} = 9.2$

Conclusion: To make efficient use of multiprocessors, it is important to minimize sequential parts, and reduce idle time in which threads wait.

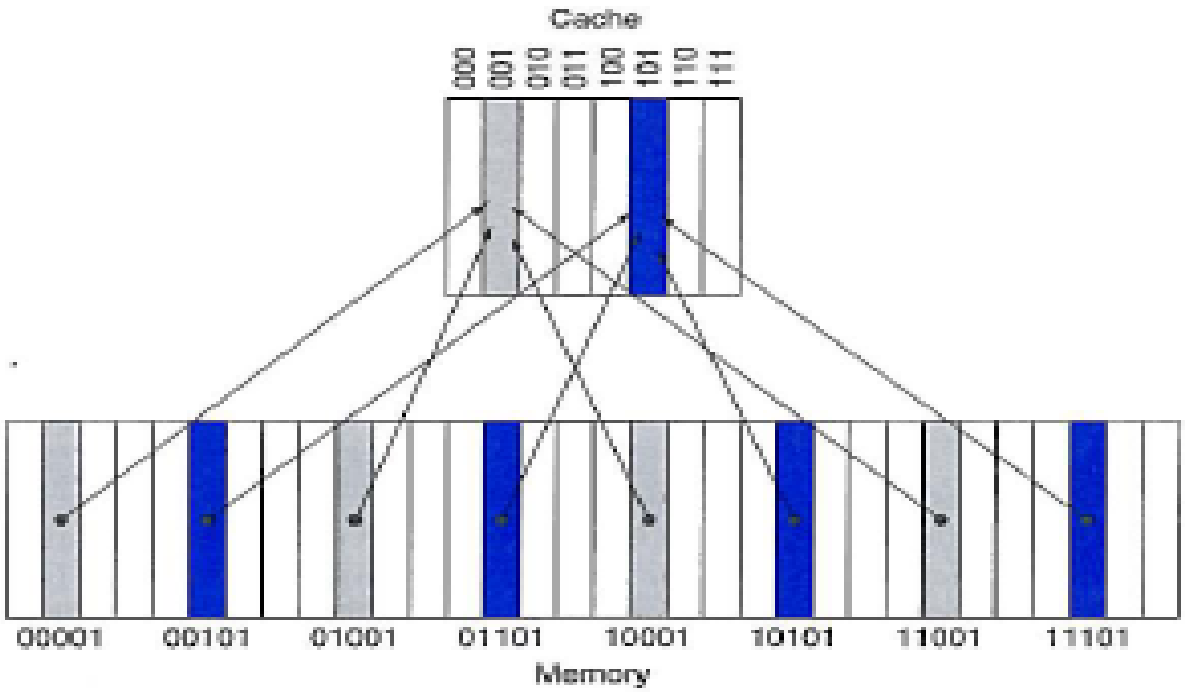
Crash Course in Modern hardware

- **Cache(Direct mapped)**



Crash Course in Modern hardware

- **Cache(Direct mapped)**



Crash Course in Modern hardware

❖Cache(Direct mapped)

Index	V	Tag	Data
000	N		
001	N		
010	N		
011	N		
100	N		
101	N		
110	N		
111	N		

a. The initial state of the cache after power-on

Index	V	Tag	Data
000	N		
001	N		
010	Y	11_{two}	Memory (11010_{two})
011	N		
100	N		
101	N		
110	Y	10_{two}	Memory (10110_{two})
111	N		

c. After handling a miss of address (11010_{two})

Index	V	Tag	Data
000	Y	10_{two}	Memory (10000_{two})
001	N		
010	Y	11_{two}	Memory (11010_{two})
011	Y	00_{two}	Memory (00011_{two})
100	N		
101	N		
110	Y	10_{two}	Memory (10110_{two})
111	N		

e. After handling a miss of address (00011_{two})

Index	V	Tag	Data
000	N		
001	N		
010	N		
011	N		
100	N		
101	N		
110	Y	10_{two}	Memory (10110_{two})
111	N		

b. After handling a miss of address (10110_{two})

Index	V	Tag	Data
000	Y	10_{two}	Memory (10000_{two})
001	N		
010	Y	11_{two}	Memory (11010_{two})
011	N		
100	N		
101	N		
110	Y	10_{two}	Memory (10110_{two})
111	N		

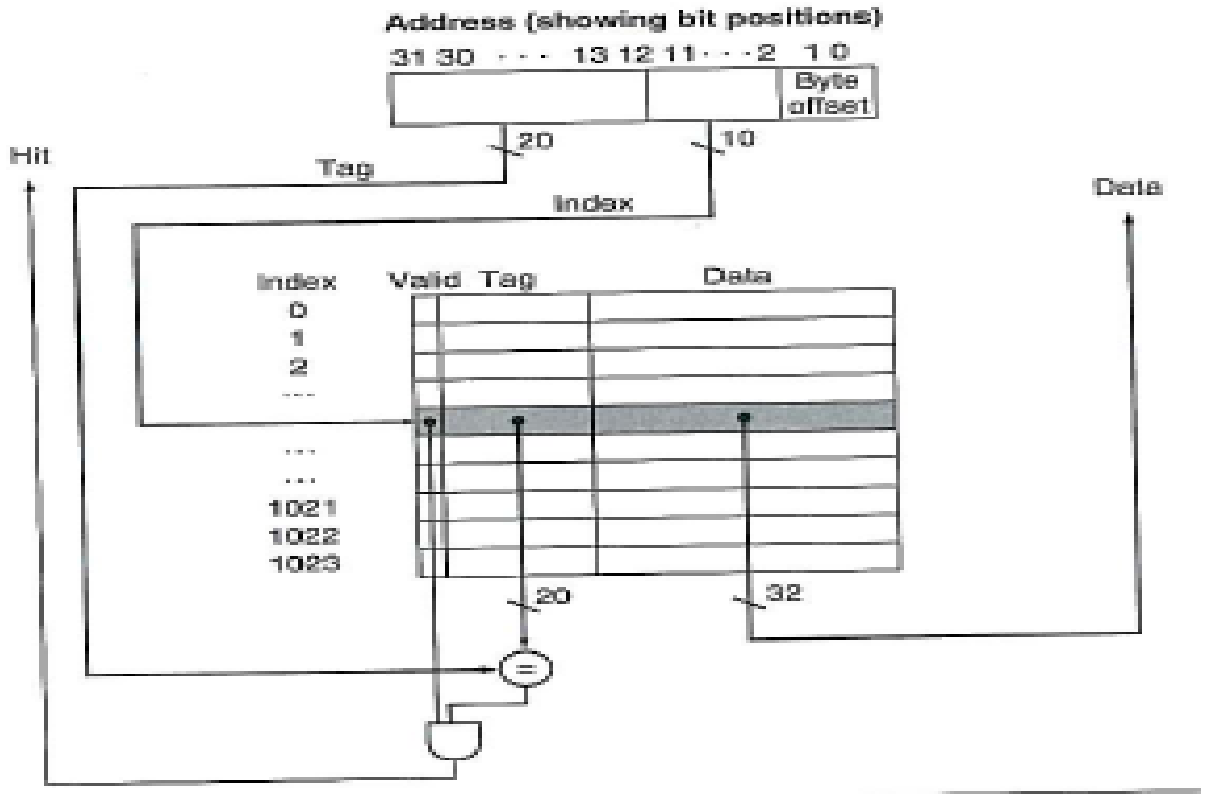
d. After handling a miss of address (10000_{two})

Index	V	Tag	Data
000	Y	10_{two}	Memory (10000_{two})
001	N		
010	Y	10_{two}	Memory (10010_{two})
011	Y	00_{two}	Memory (00011_{two})
100	N		
101	N		
110	Y	10_{two}	Memory (10110_{two})
111	N		

f. After handling a miss of address (10010_{two})

Crash Course in Modern hardware

❖Cache(Direct mapped)



Crash Course in Modern hardware

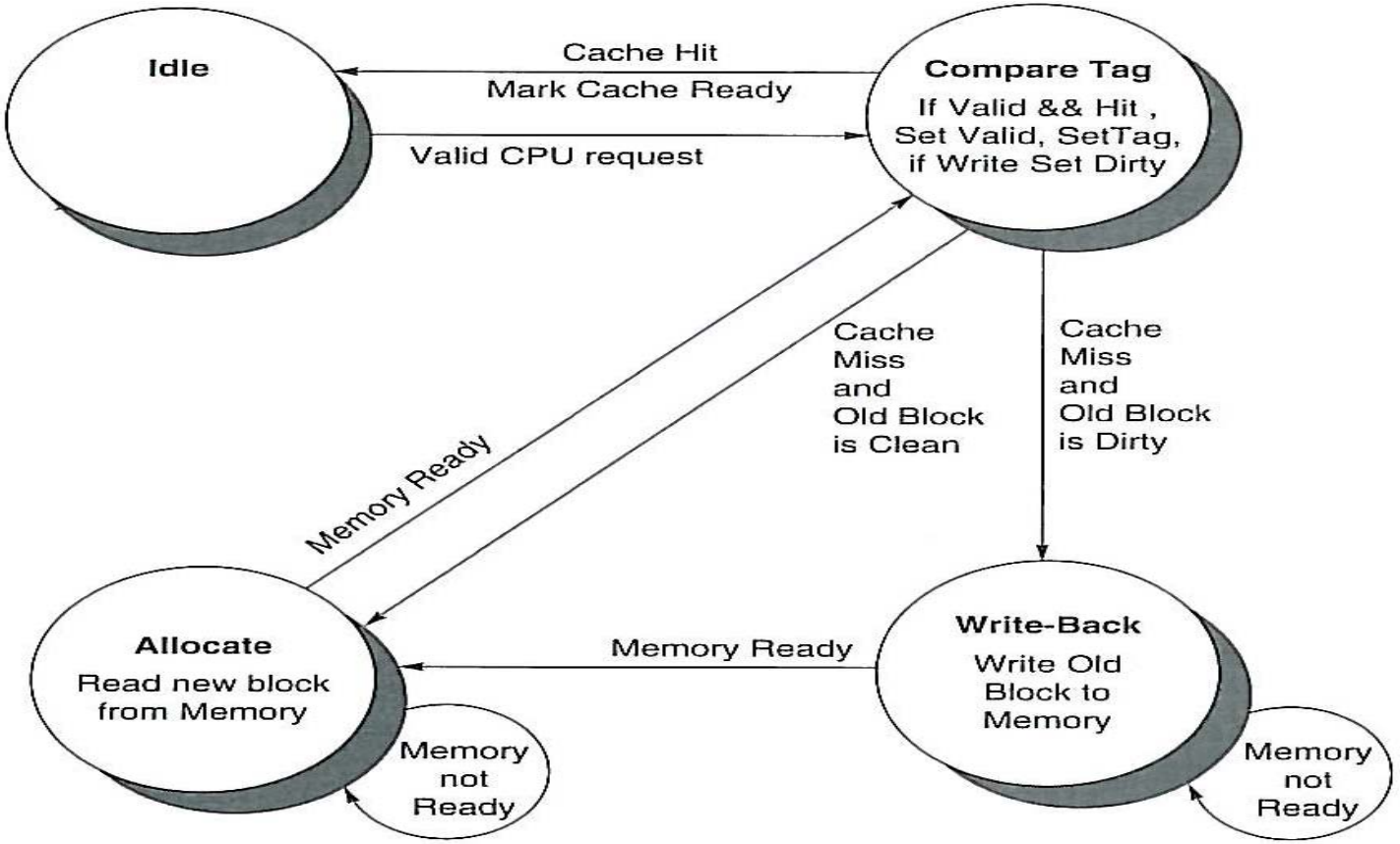
- **Cache**
  - Read
    - Send original PC value to the memory
    - Instruct main memory to perform read and wait for the memory to complete access
    - Write the cache entry(data,tag,valid)
    - Restart instruction execution.

Crash Course in Modern hardware

- **Cache**
  - Writes
    - Handling writes are more complicated
    - Many schemes
      - Write Through
      - Write Back

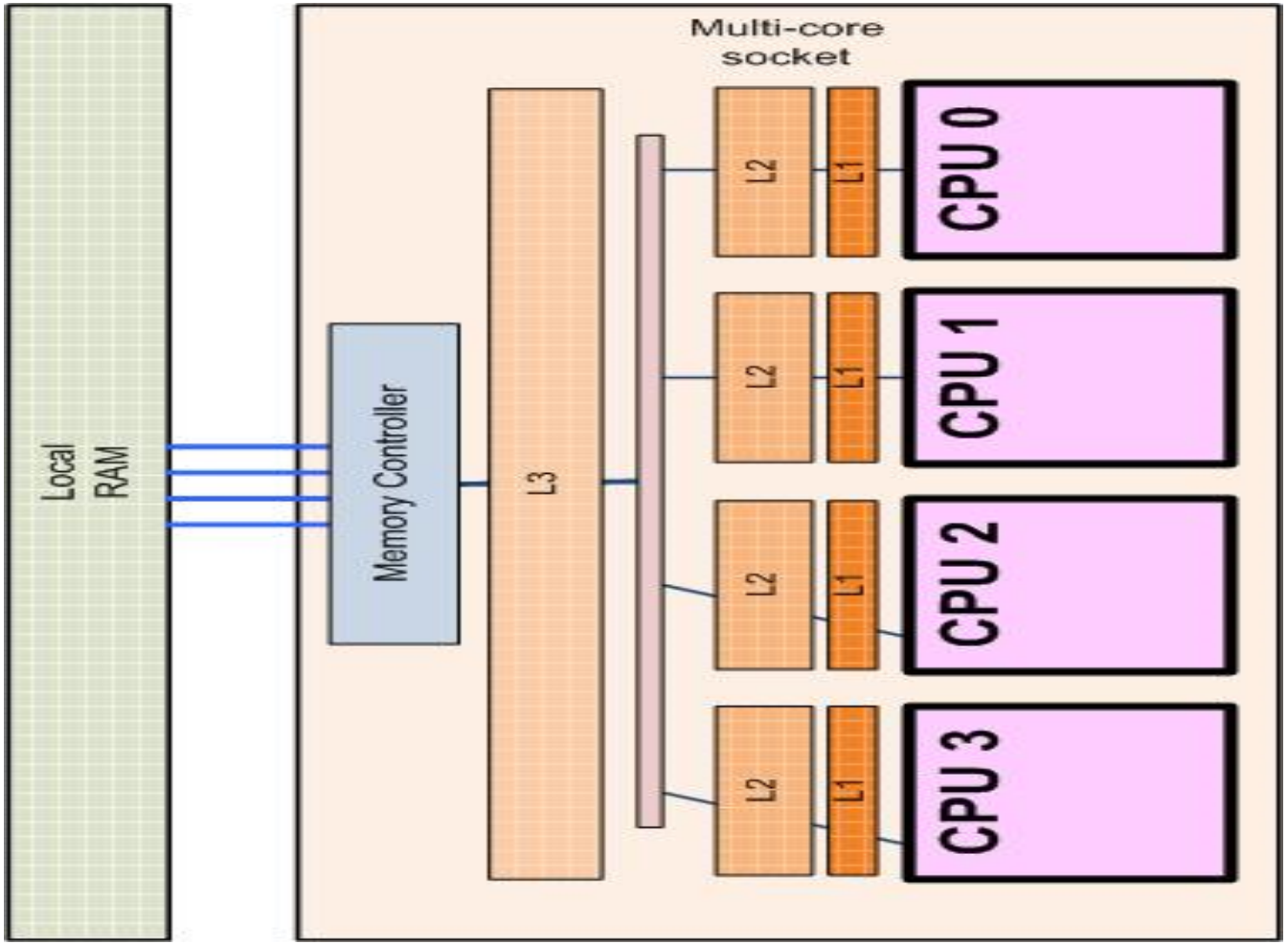
Crash Course in Modern hardware

❖Cache Controller



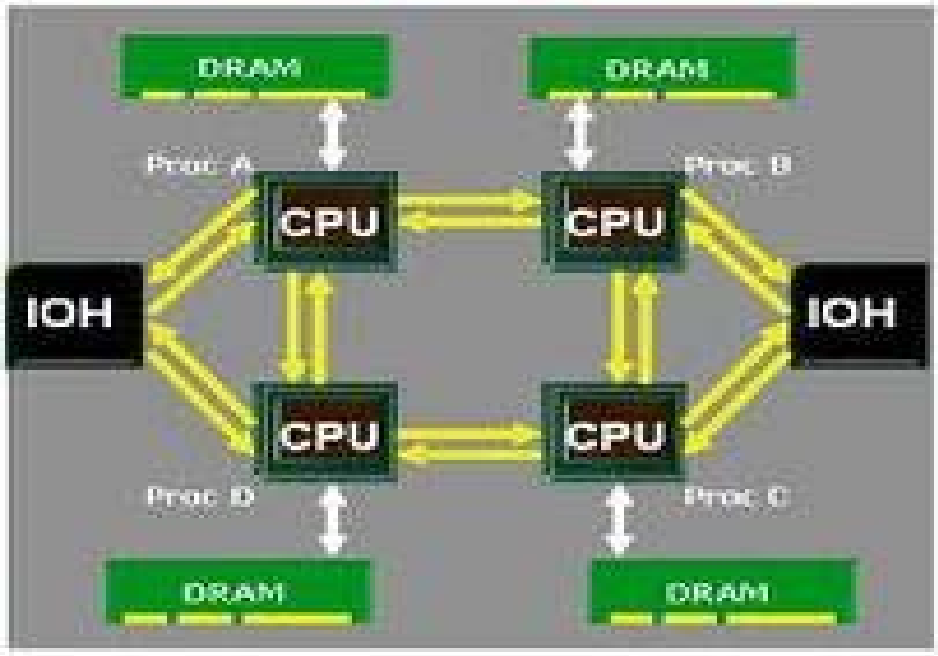
Crash Course in Modern hardware

Multicore UMA



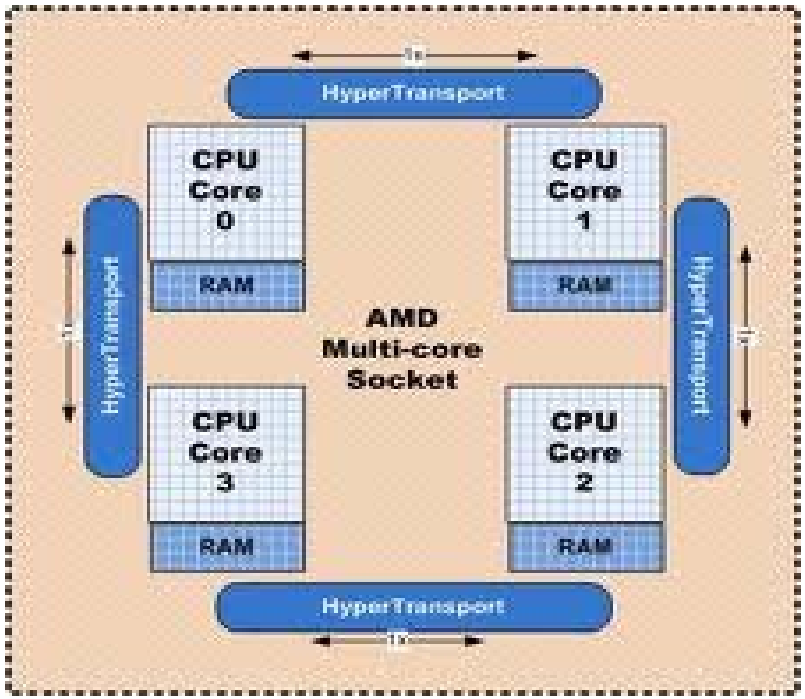
Crash Course in Modern hardware

Multicore NUMA



Crash Course in Modern hardware

Multicore NUMA



Crash Course in Modern hardware

Bus Snooping

- on a write, all caches check to see if they have a copy and then act, either invalidating or updating their copy to the new value.

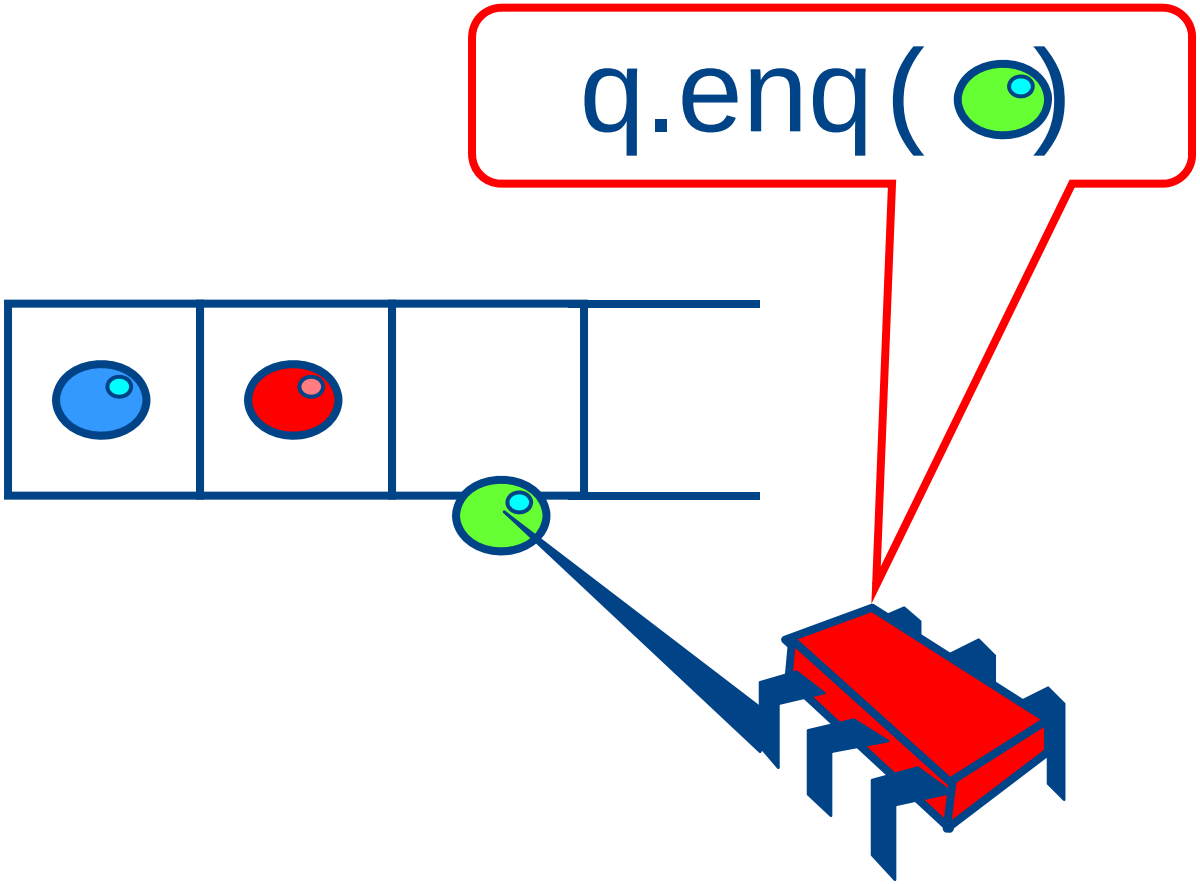
Concurrent Objects

- **Concurrent Reasoning**
  - Sequential Consistency
  - Linearizability
  - Quiescent Consistency
  - Compositionality

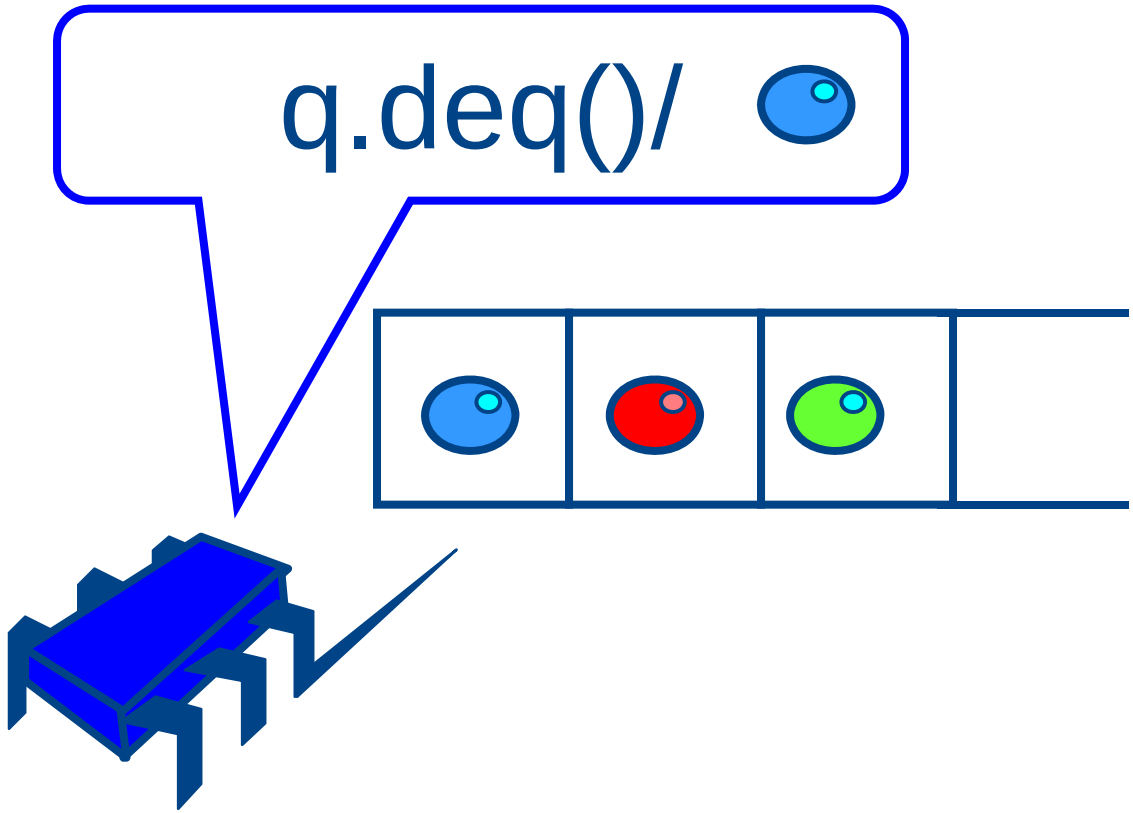
Objectivism

- **What is a concurrent object?**
  - How do we **describe** one?
  - How do we **implement** one?
  - How do we **tell if we're right**?

FIFO Queue: Enqueue Method



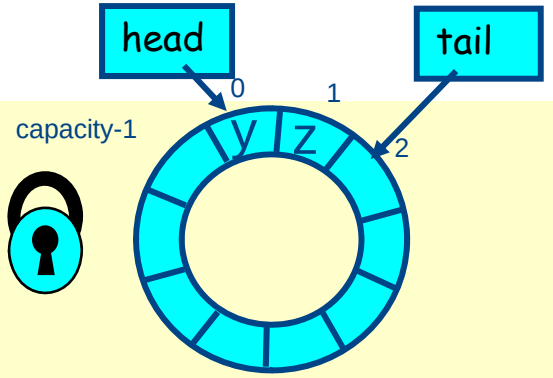
FIFO Queue: Dequeue Method



A Lock-Based Queue

```
class LockBasedQueue<T> {
  int head, tail;
  T[] items;
  Lock lock;
  public LockBasedQueue(int capacity) {
    head = 0; tail = 0;
    lock = new ReentrantLock();
    items = (T[]) new Object[capacity];
}
```

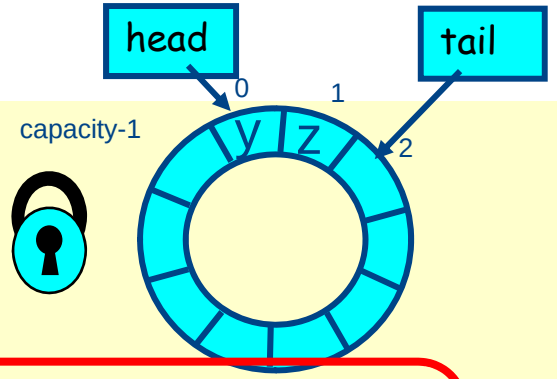
A Lock-Based Queue




```
class LockBasedQueue<T> {
  int head, tail;
  T[] items;
  Lock lock;
  public LockBasedQueue(int capacity) {
    head = 0; tail = 0;
    lock = new ReentrantLock();
    items = (T[]) new Object[capacity];
}
```

Queue fields protected by single shared lock

A Lock-Based Queue

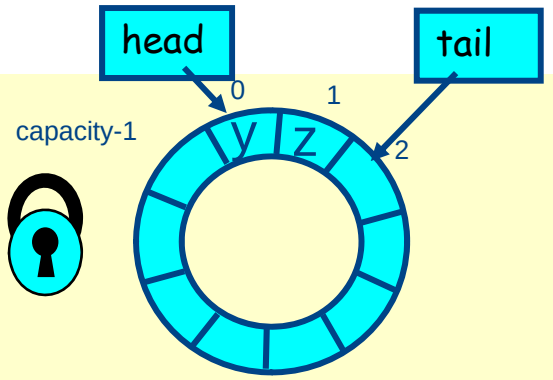



```
class LockBasedQueue<T> {
  int head, tail;
  T[] items;
  Lock lock;
  public LockBasedQueue(int capacity) {
    head = 0; tail = 0;
    lock = new ReentrantLock();
    items = (T[]) new Object[capacity];
}
```

Initially head = tail

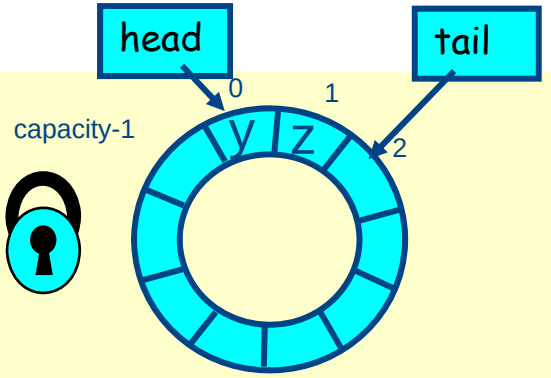
Implementation: Deq

```
public T deq() throws EmptyException {
  lock.lock();
  try {
    if (tail == head)
       throw new EmptyException();
    T x = items[head % items.length];
    head++;
    return x;
  } finally {
    lock.unlock();
  }
}
```



Implementation: Deq

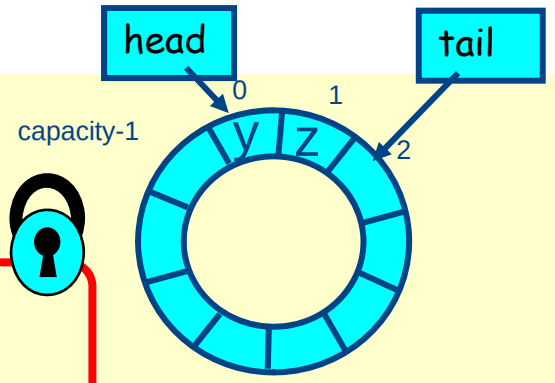
```
public T deq() throws EmptyException {
  lock.lock();
  try {
    if (tail == head)
      throw new EmptyException();
    T x = items[head % items.length];
    head++;
    return x;
  } finally {
    lock.unlock();
  }
}
```



Method calls mutually exclusive

Implementation: Deq

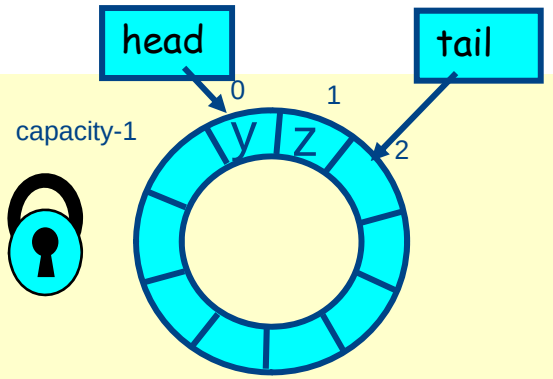
```
public T deq() throws EmptyException {
  lock.lock();
  try {
    if (tail == head)
      throw new EmptyException();
    T x = items[head % items.length];
    head++;
    return x;
  } finally {
    lock.unlock();
  }
}
```



If queue empty
throw exception

Implementation: Deq

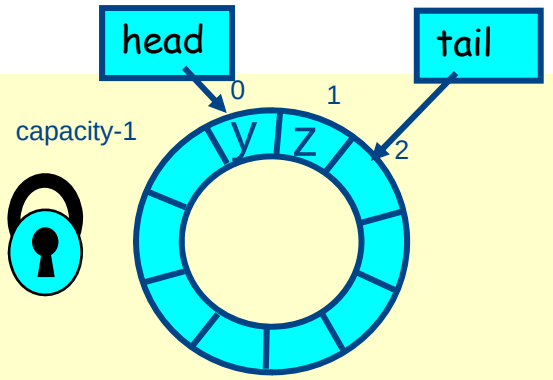
```
public T deq() throws EmptyException {
  lock.lock();
  try {
    if (tail == head)
      throw new EmptyException();
    T x = items[head % items.length];
    head++;
    return x;
  } finally {
    lock.unlock();
  }
}
```



Queue not empty: remove item and update head

Implementation: Deq

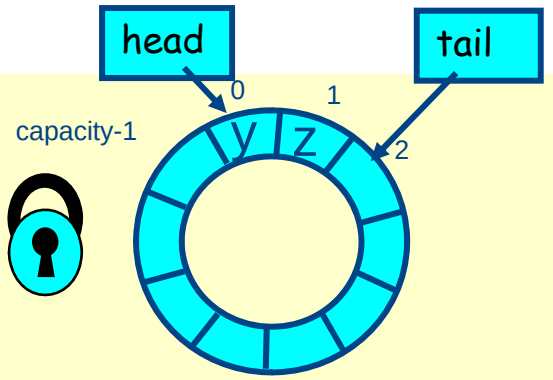
```
public T deq() throws EmptyException {
  lock.lock();
  try {
    if (tail == head)
      throw new EmptyException();
    T x = items[head % items.length];
    head++;
    return x;
  } finally {
    lock.unlock();
  }
}
```



Return result

Implementation: Deq

```
public T deq() throws EmptyException {
  lock.lock();
  try {
    if (tail == head)
       throw new EmptyException();
    T x = items[head % items.length];
    head++;
    return x;
  } finally {
    lock.unlock();
  }
}
```



Release lock no matter what!

Implementation: Deq

```
public T deq() throws EmptyException {
  lock.lock();
  try {
    if (tail == head)
       throw new EmptyException();
    T x = items[head % items.length];
    head++;
    return x;
  } finally {
    lock.unlock();
  }
}
```

Should be correct because modifications are mutually exclusive…

Now consider the following implementation

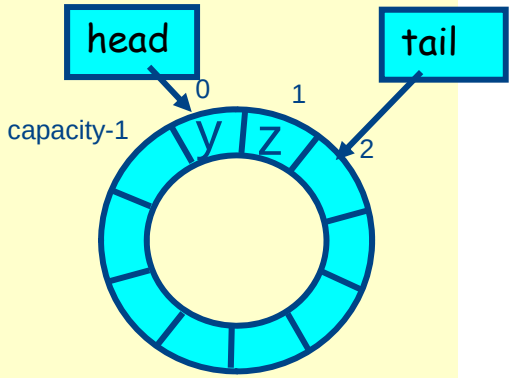
- The same thing without mutual exclusion
- For simplicity, only **two** threads
  - One thread enq only
  - The other deq only

Wait-free 2-Thread Queue

```
public class WaitFreeQueue {

  int head = 0, tail = 0;
  items = (T[]) new Object[capacity];

  public void enq(Item x) {
    while (tail-head == capacity); // busy-wait
    items[tail % capacity] = x; tail++;
  }
  public Item deq() {
     while (tail == head);     // busy-wait
     Item item = items[head % capacity]; head++;
     return item;
}}
```

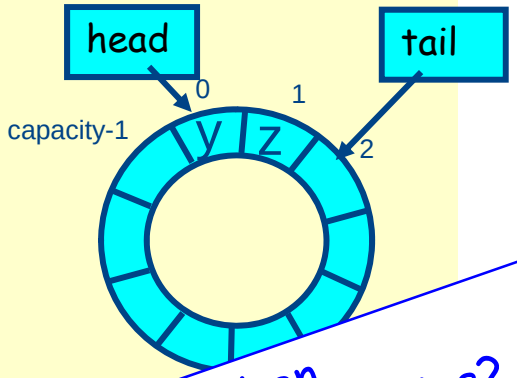
Wait-free 2-Thread Queue

```
public class LockFreeQueue {

  int head = 0, tail = 0;
  items = (T[]) new Object[capacity];

  public void enq(Item x) {
    while (tail-head == capacity); // busy-wait
    items[tail % capacity] = x; tail++;
  }
  public Item deq() {
     while (tail == head);     // busy-wait
     Item item = items[head % capacity]; head++;
     return item;
}}
```



Lock-free 2-Thread Queue

```
public class LockFreeQueue {

  int head = 0, tail = 0;
  items = (T[])new Object[capacity];

  public void enq(Item x) {
    while (tail-head == capacity); // busy-wait
    items[tail % capacity] = x; tail++;
  }
  public Item deq() {
    while (tail == head);     // busy-wait
    Item item = items[head % capacity];
    return item;
}}
```



How do we define "correct" when modifications are not mutually exclusive?

Queue is updated without a lock!

Defining concurrent queue implementations

- Need a way to **specify** a concurrent queue object
- Need a way to prove that an algorithm **implements** the object's specification
- Lets talk about object specifications …

Correctness and Progress

- **In a concurrent setting, we need to specify both the safety and the liveness properties of an object**
- **Need a way to define**
  - when an implementation is correct
  - the conditions under which it guarantees progress

Lets begin with correctness

Sequential Objects

- **Each object has a *state***
  - Usually given by a set of ***fields***
  - Queue example: sequence of items
- **Each object has a set of *methods***
  - Only way to manipulate state
  - Queue example: **enq** and **deq** methods

Sequential Specifications

- **If (precondition)**
  - the object is in such-and-such a state
  - before you call the method,
- **Then (postcondition)**
  - the method will return a particular value
  - or throw a particular exception.
- **and (postcondition, con't)**
  - the object will be in some other state
  - when the method returns,

Pre and PostConditions for Dequeue

- **Precondition:**
  - Queue is non-empty
- **Postcondition:**
  - Returns first item in queue
- **Postcondition:**
  - Removes first item in queue

Pre and PostConditions for Dequeue

- **Precondition:**
  - Queue is empty
- **Postcondition:**
  - Throws Empty exception
- **Postcondition:**
  - Queue state unchanged

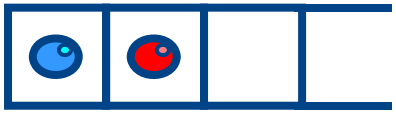
Why Sequential Specifications Totally Rock

- **Interactions among methods captured by side-effects on object state**
  - State meaningful between method calls
- **Documentation size linear in number of methods**
  - Each method described in isolation
- **Can add new methods**
  - Without changing descriptions of old methods

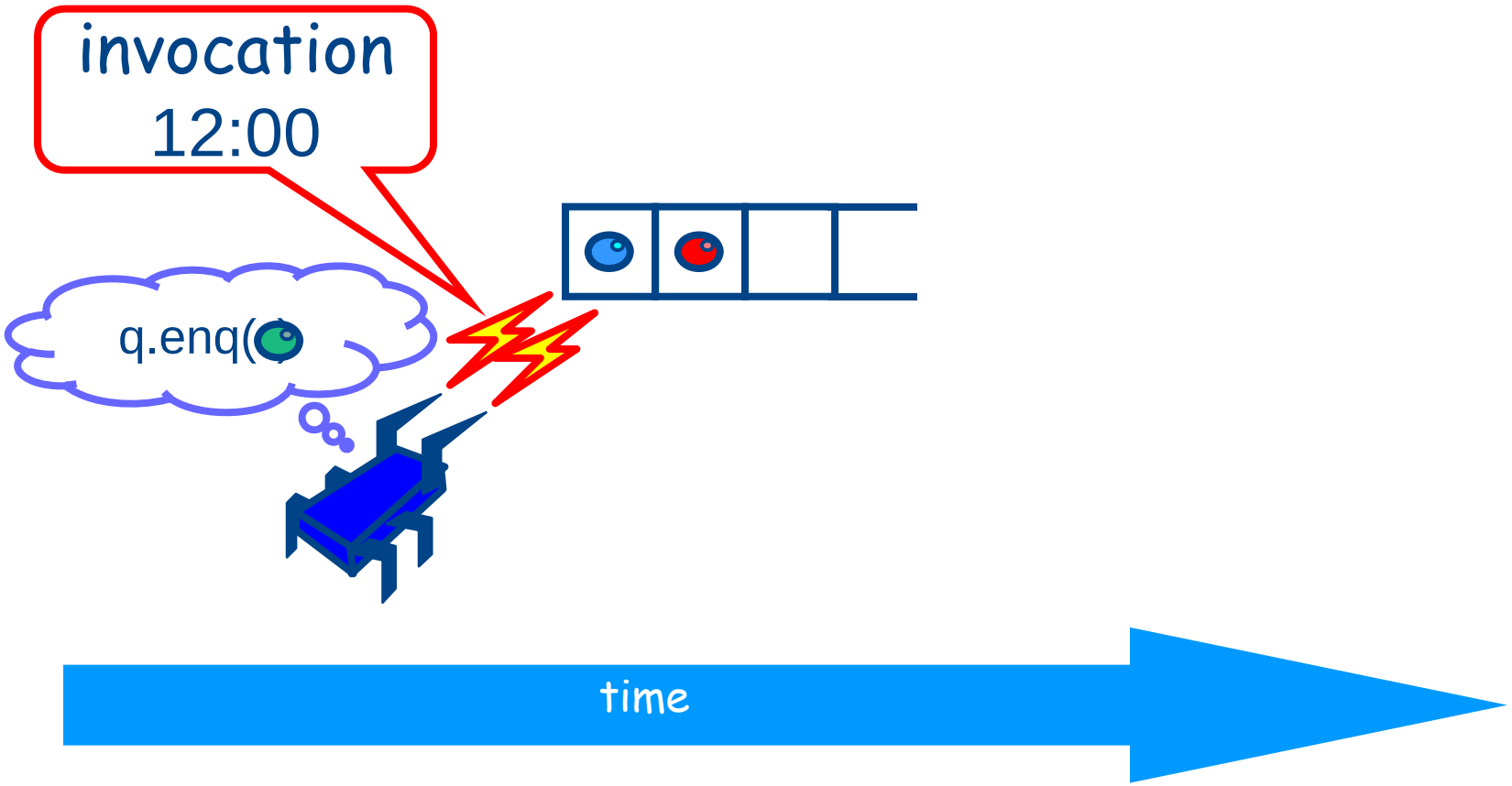
What About Concurrent Specifications ?

- Methods?
- Documentation?
- Adding new methods?

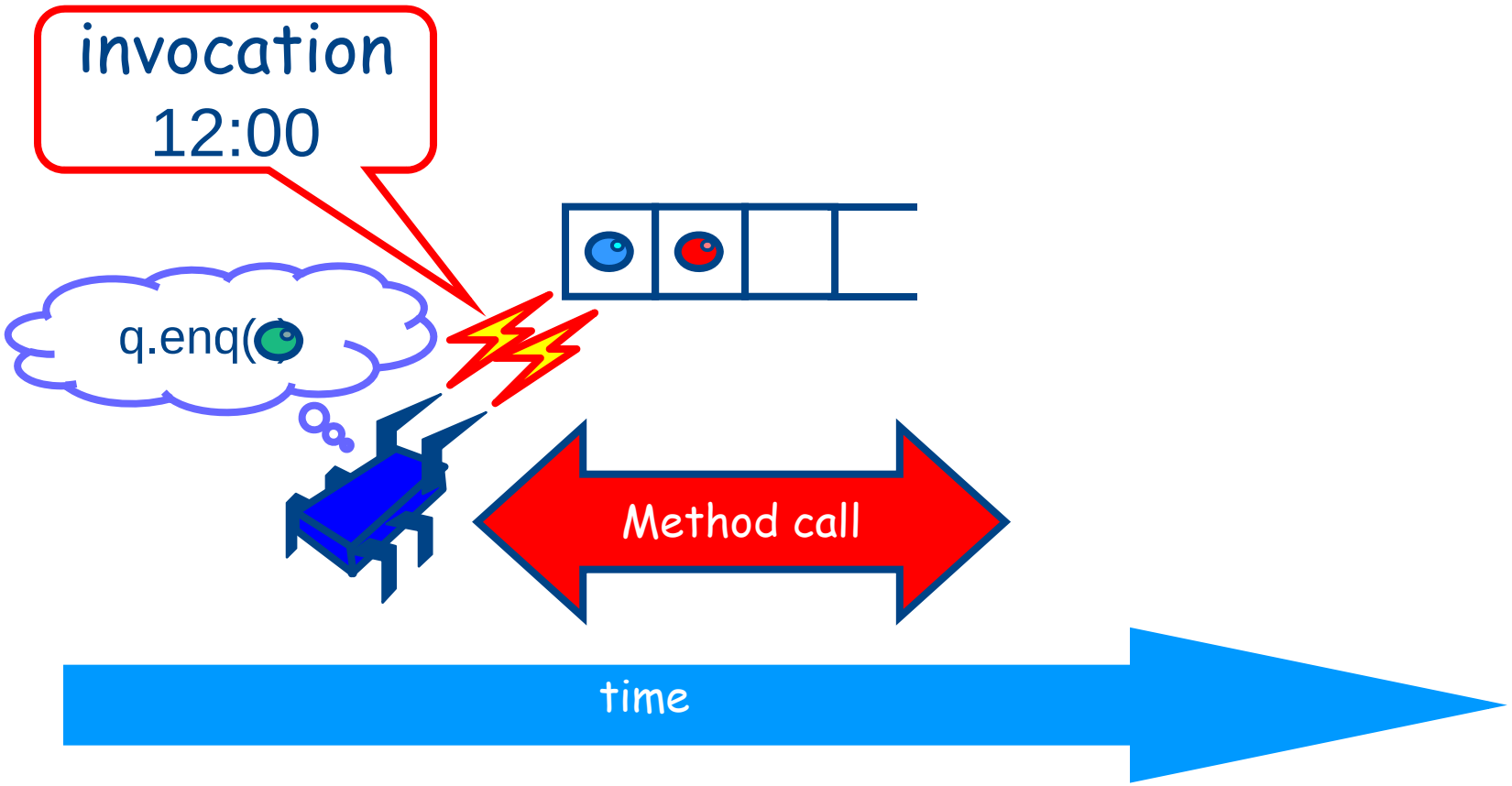
Methods Take Time



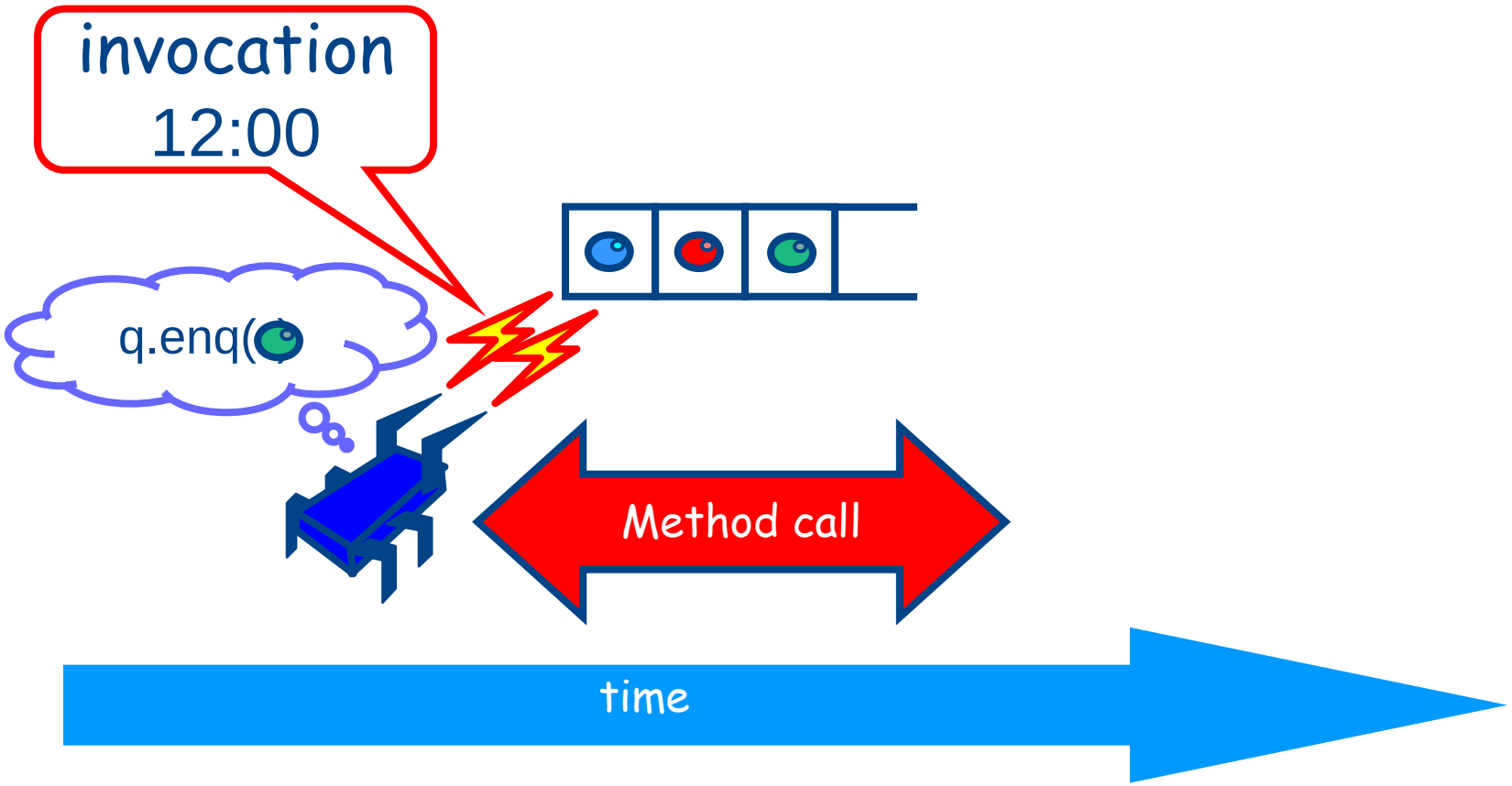
Methods Take Time



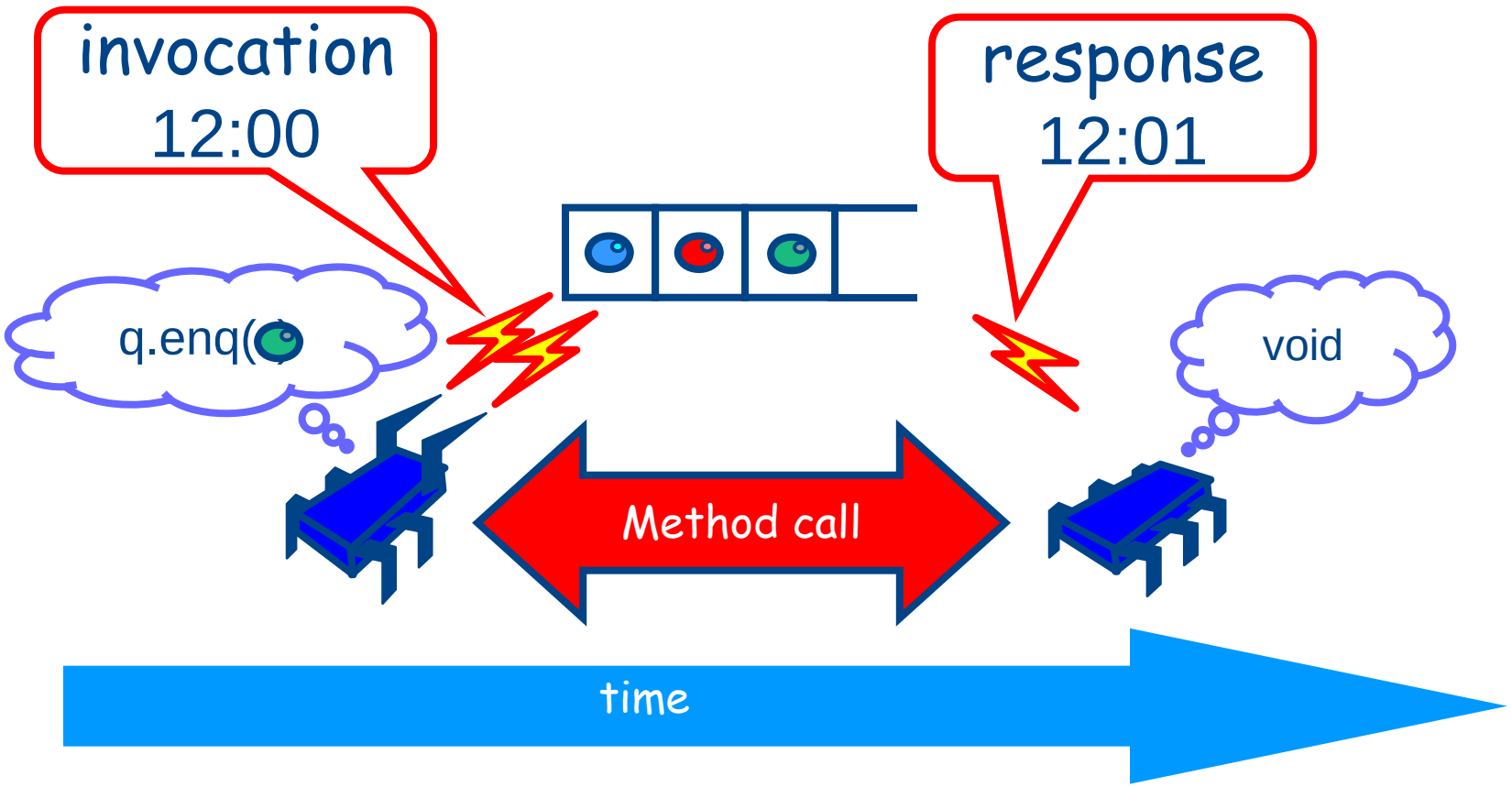
Methods Take Time



Methods Take Time



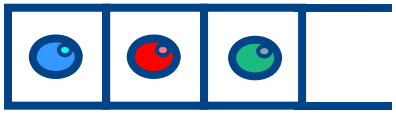
Methods Take Time



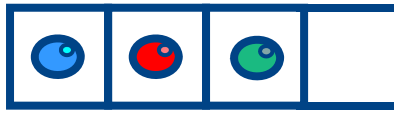
Sequential vs Concurrent

- **Sequential**
  - Methods take time? Who knew?
- **Concurrent**
  - Method call is not an event
  - Method call is an interval.

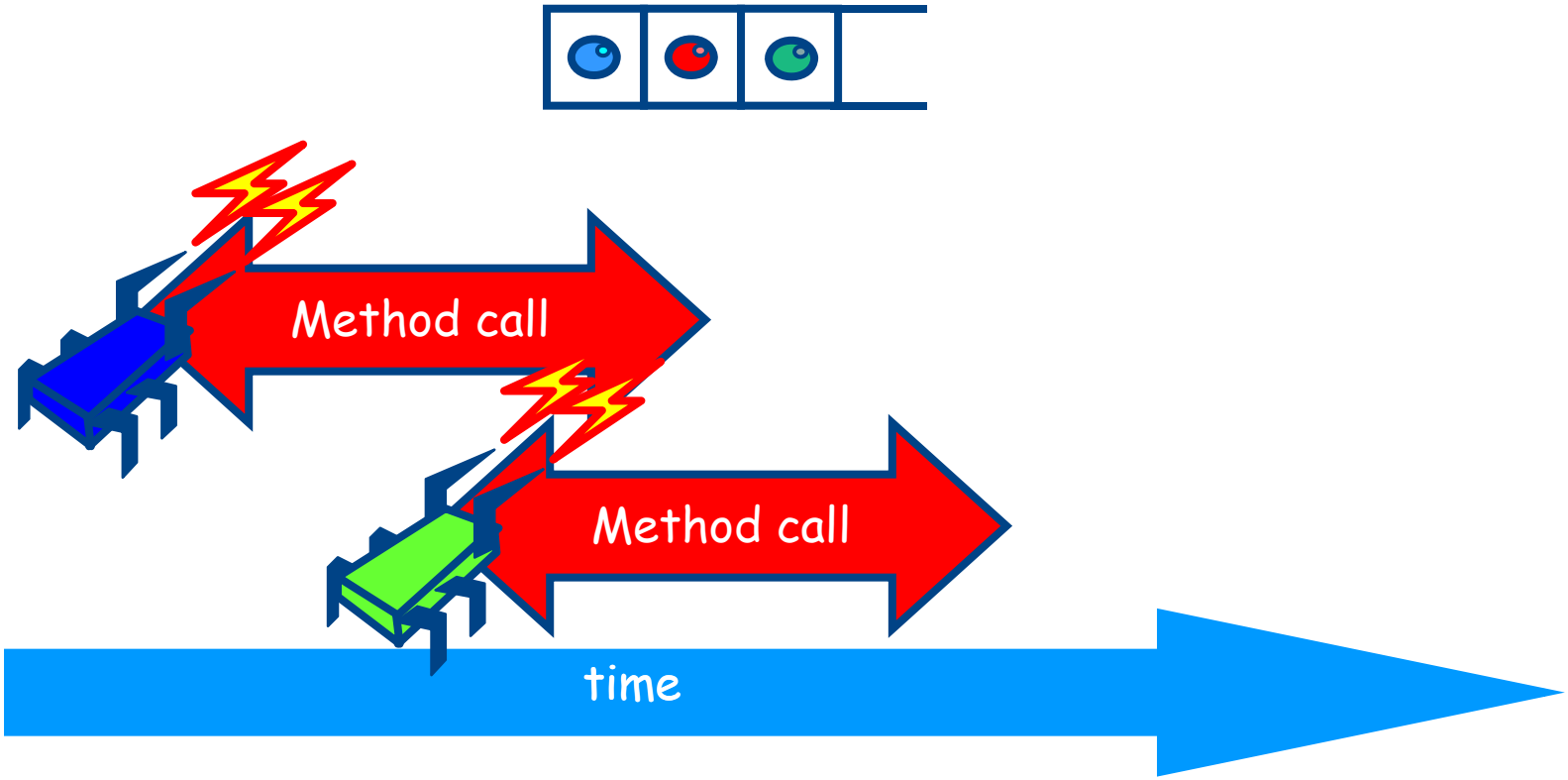
Concurrent Methods Take Overlapping Time



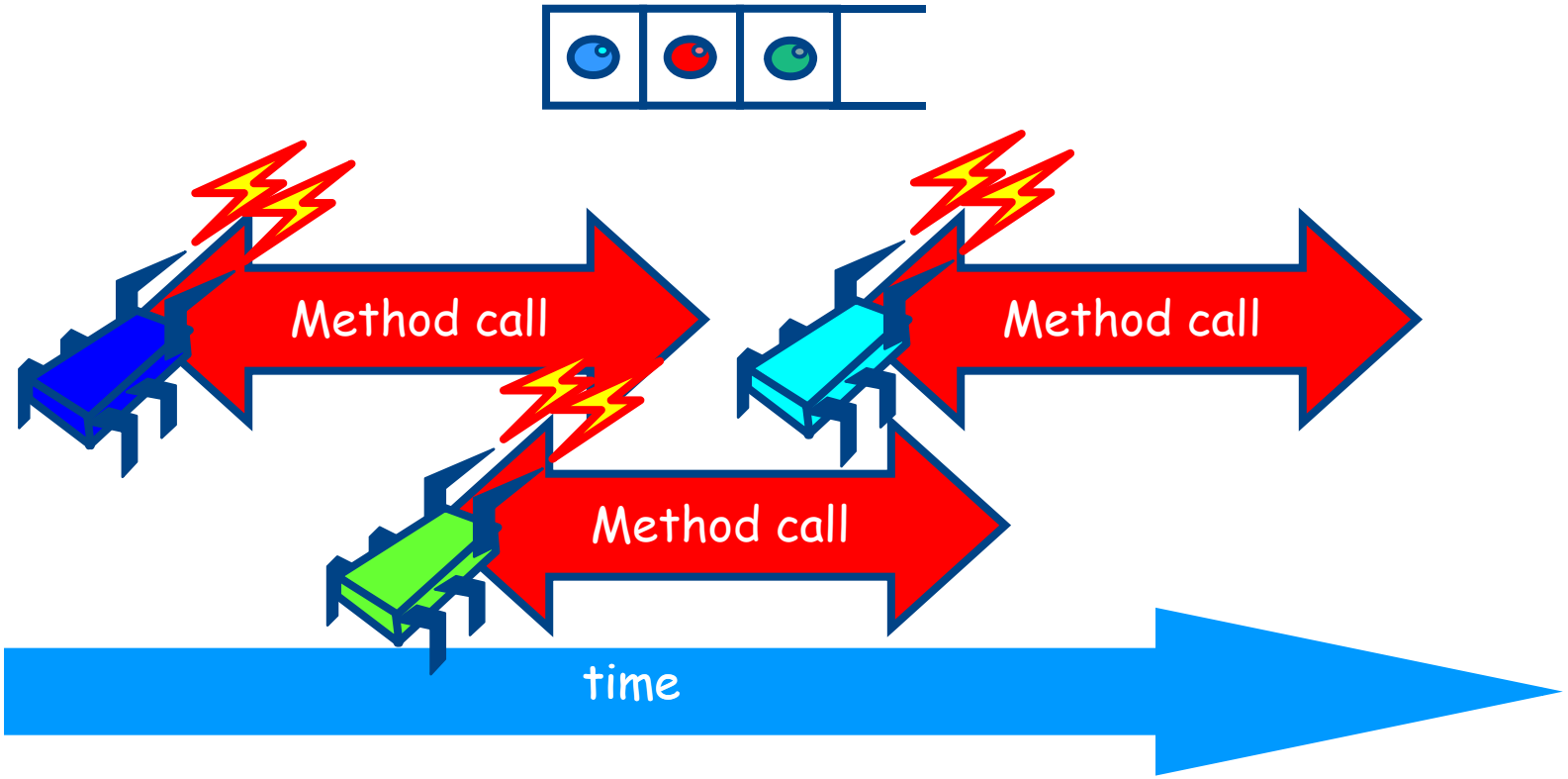
Concurrent Methods Take Overlapping Time



Concurrent Methods Take Overlapping Time



Concurrent Methods Take Overlapping Time



Sequential vs Concurrent

- **Sequential:**
  - Object needs meaningful state only between method calls
- **Concurrent**
  - Because method calls overlap, object might ***never*** be between method calls

Sequential vs Concurrent

- **Sequential:**
  - Each method described in isolation
- **Concurrent**
  - Must characterize ***all*** possible interactions with concurrent calls
    - What if two enqs overlap?
    - Two deqs? enq and deq? …

Sequential vs Concurrent

- **Sequential:**
  - Can add new methods without affecting older methods
- **Concurrent:**
  - Everything can potentially interact with everything else

Sequential vs Concurrent

- **Sequential:**
  - Can add new methods without affecting older methods
- **Concurrent:**
  - Everything can potentially interact with everything else

The Big Question

- **What does it mean for a *concurrent* object to be correct?**
  - What *is* a concurrent FIFO queue?
  - FIFO means strict temporal order
  - Concurrent means ambiguous temporal order

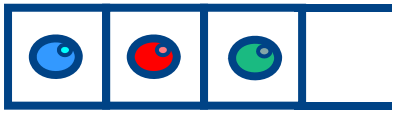
Linearizability

- **Each method should**
  - "take effect"
  - Instantaneously
  - Between invocation and response events
- **Object is correct if this "sequential" behavior is correct**
- **Any such concurrent object is**
  - **Linearizable™**

Is it really about the object?

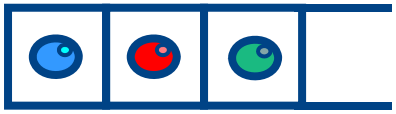
- **Each method should**
  - "take effect"
  - Instantaneously
  - Between invocation and response events
- **Sounds like a property of an execution…**
- **A linearizable object: one all of whose possible executions are linearizable**

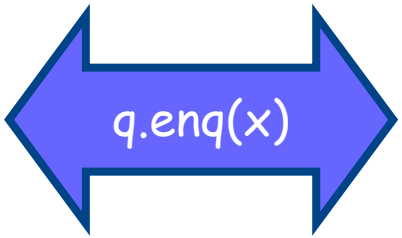
Example



(6)

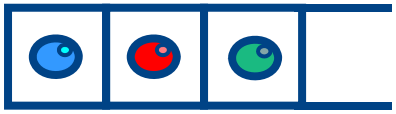
Example

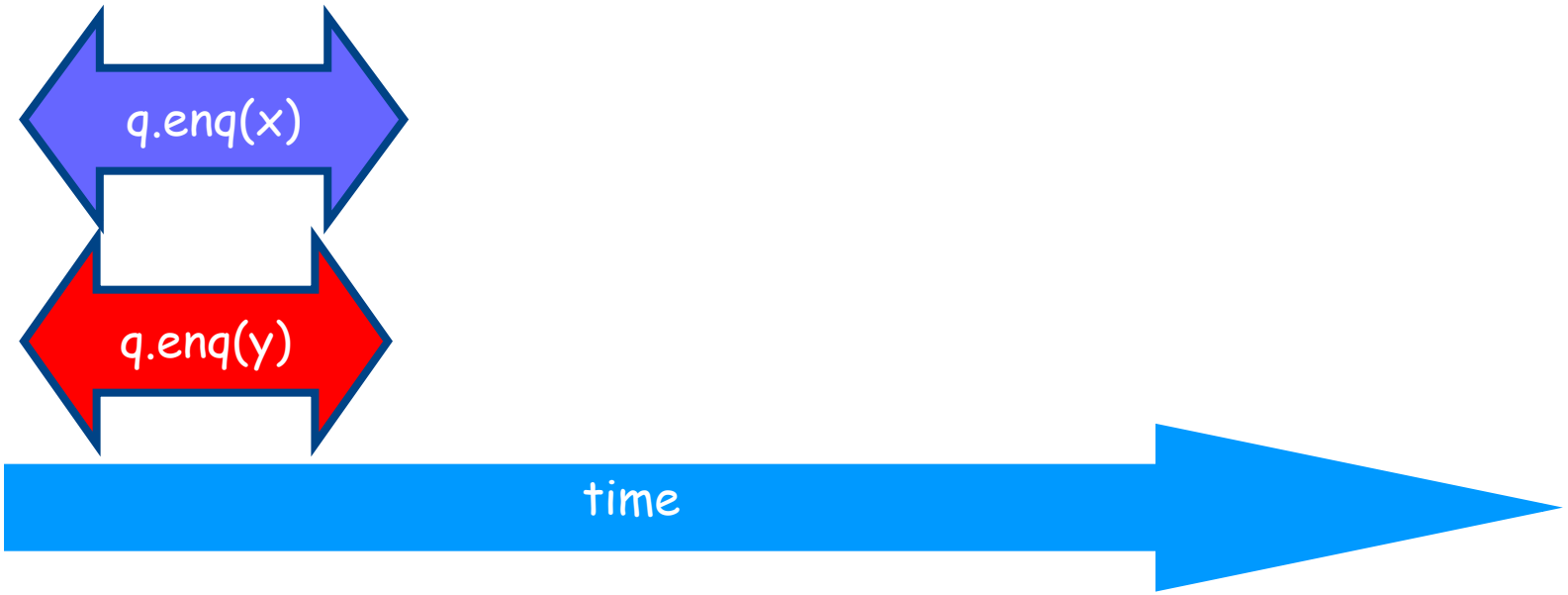




(6)

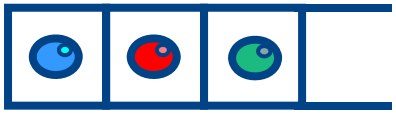
Example

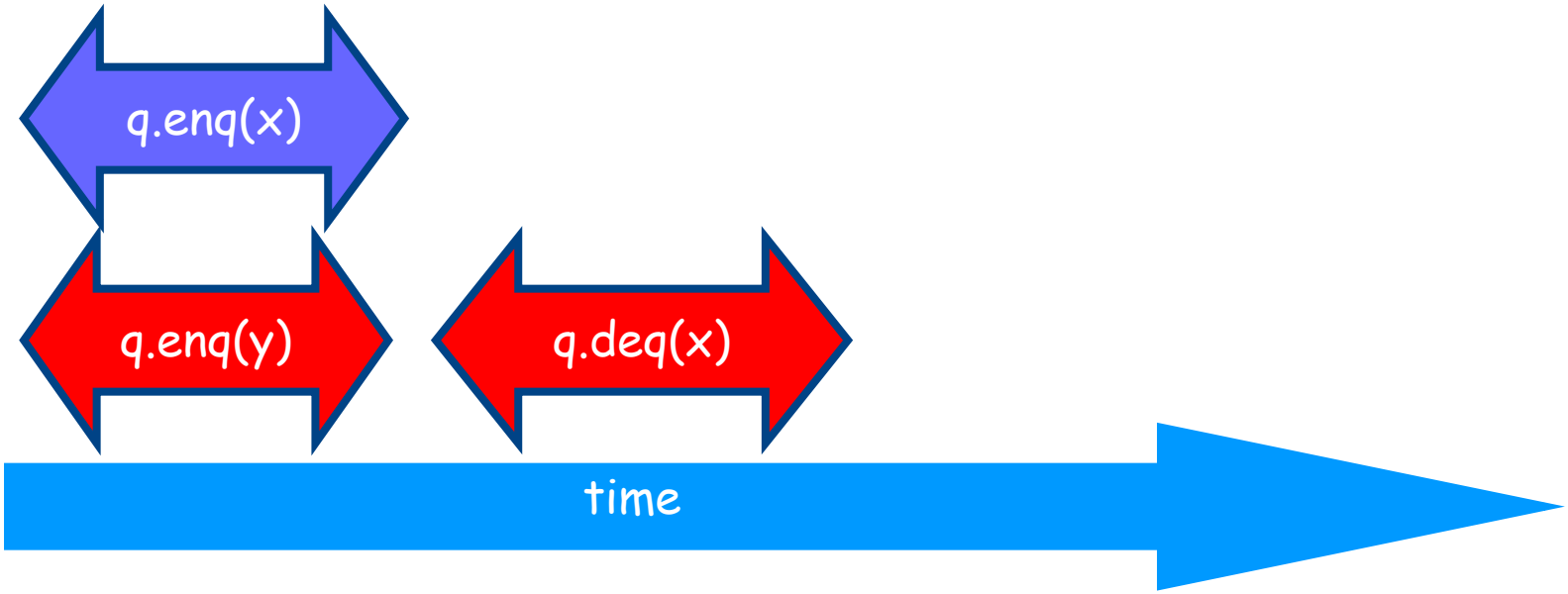




(6)

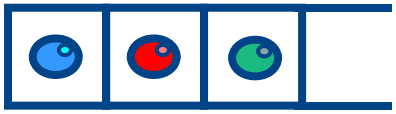
Example

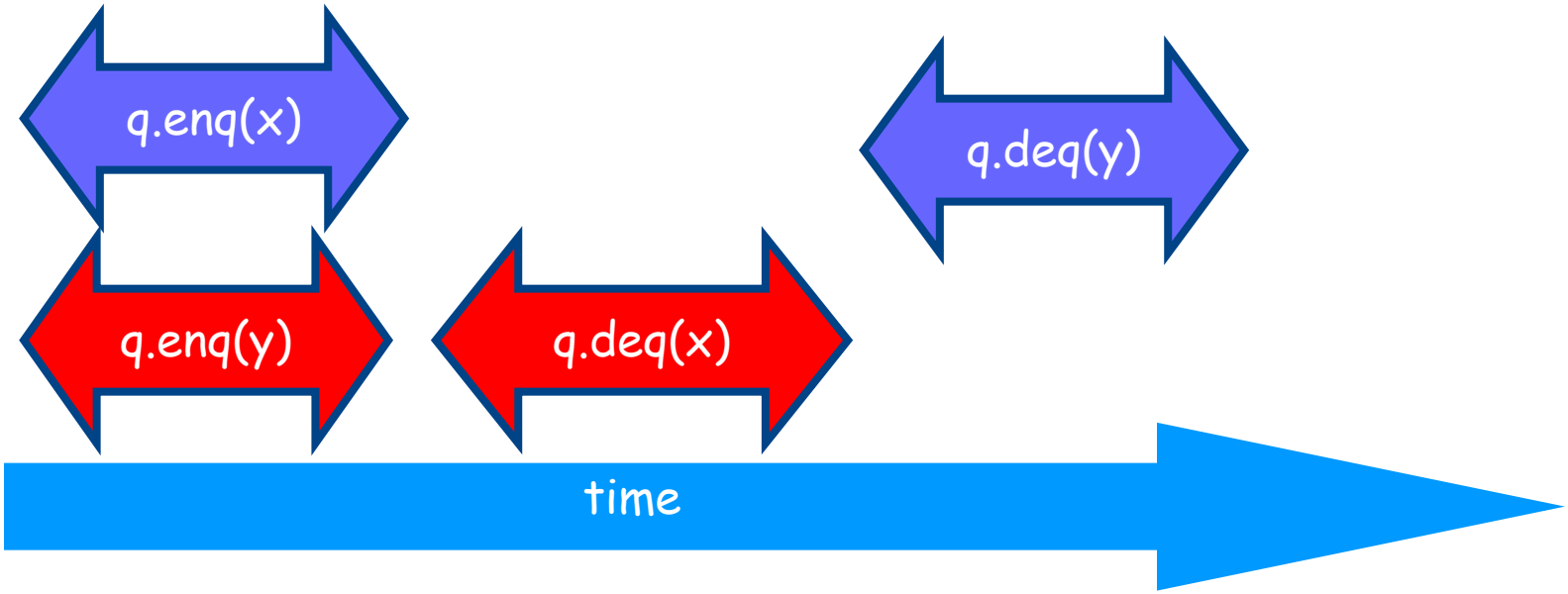




(6)

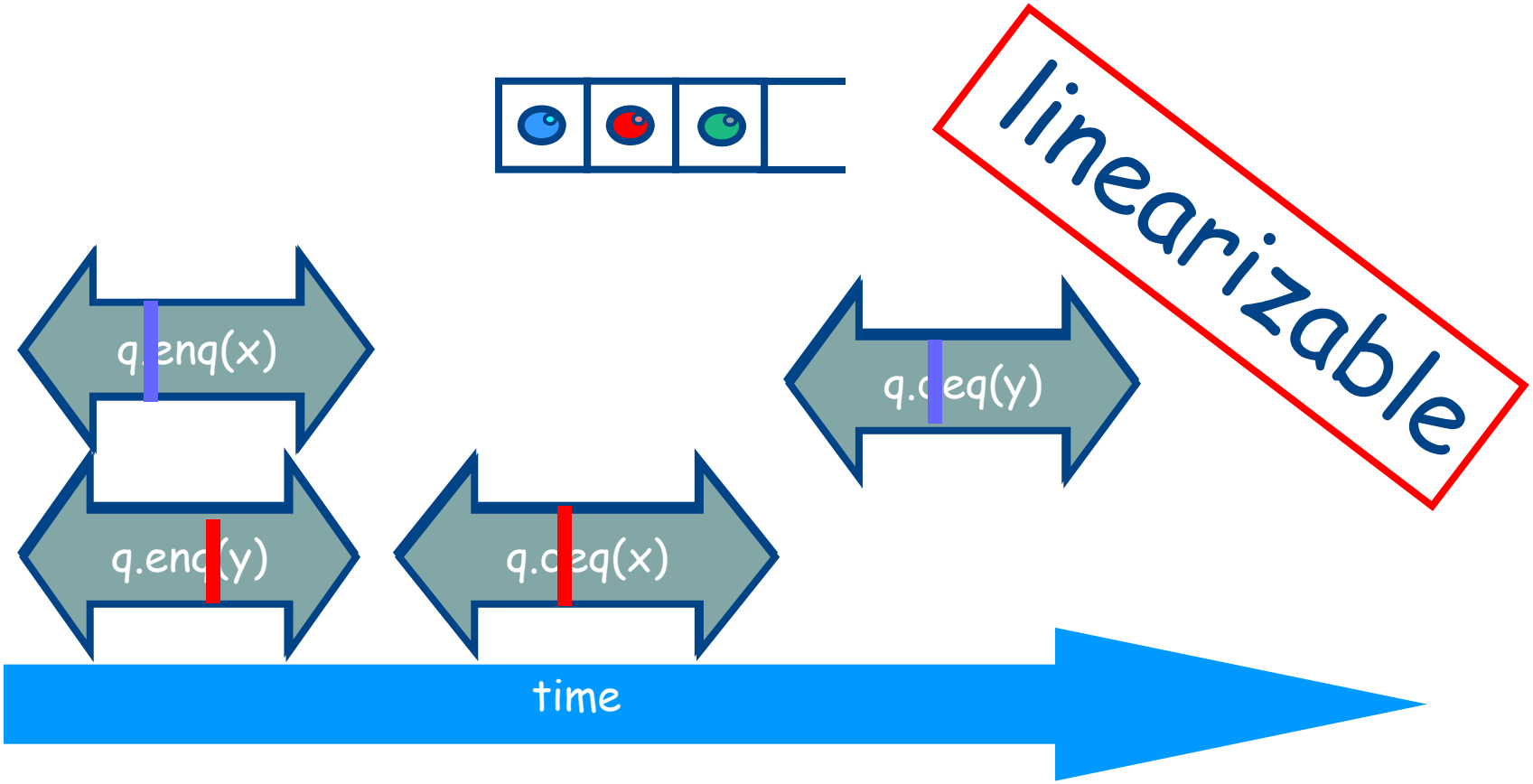
Example





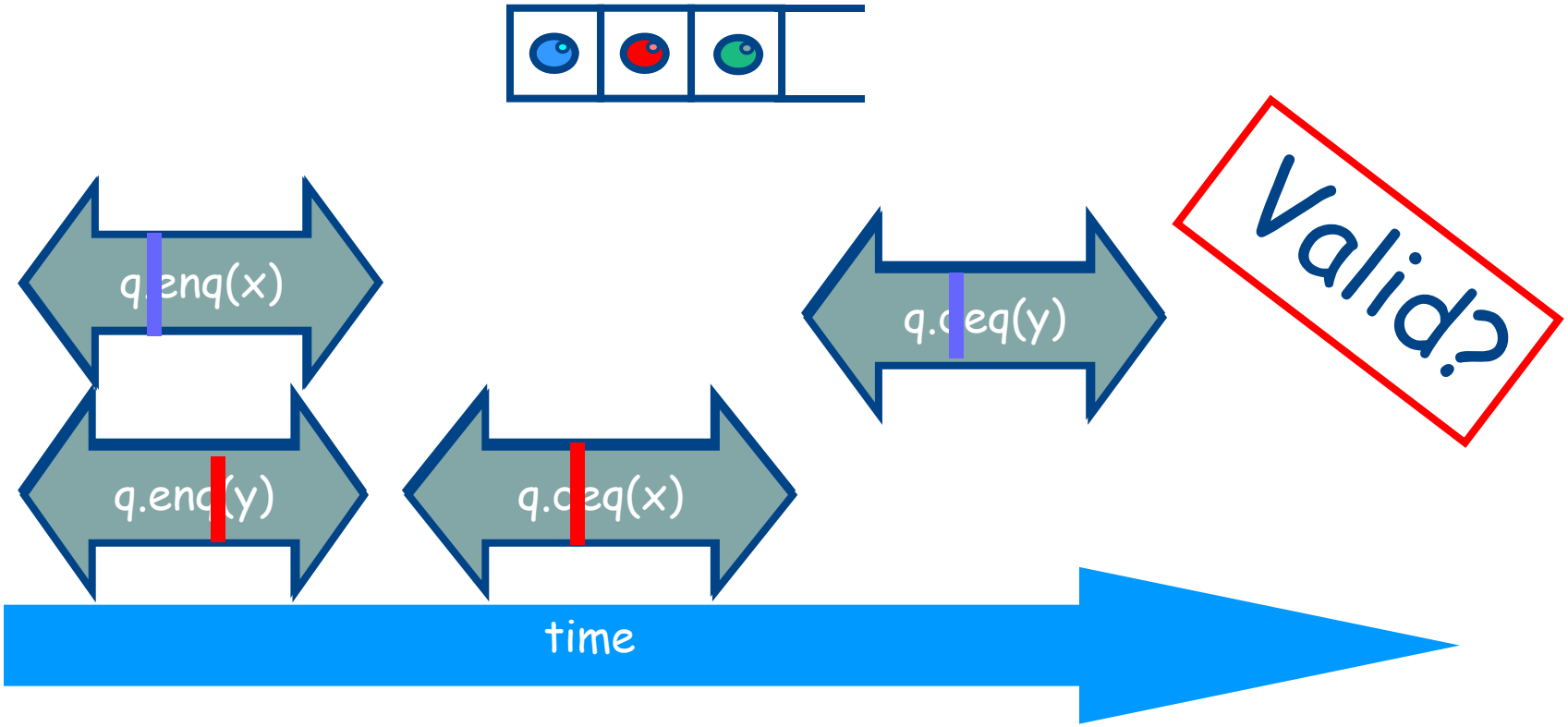
(6)

Example



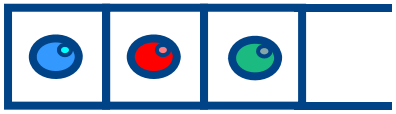
(6)

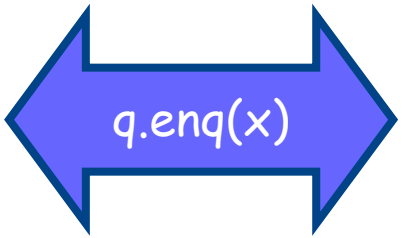
Example



(6)

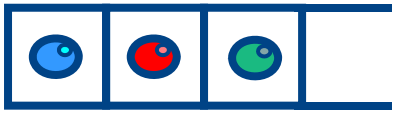
Example

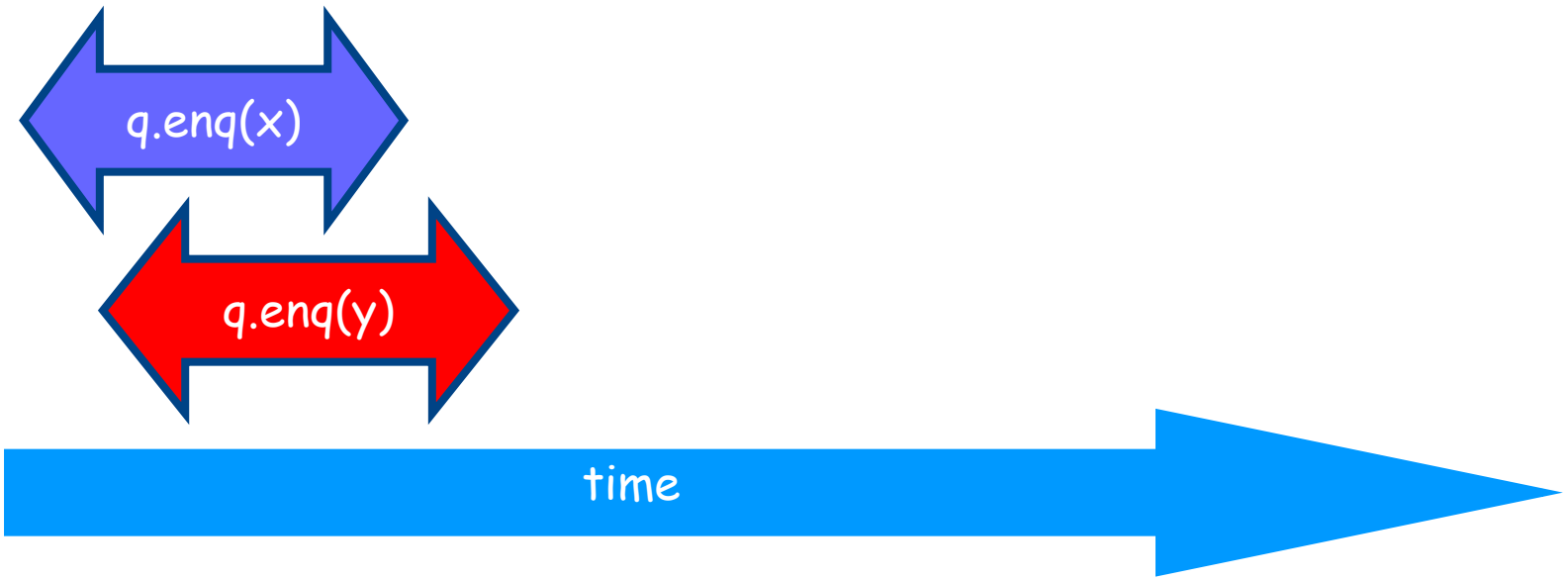




(8)

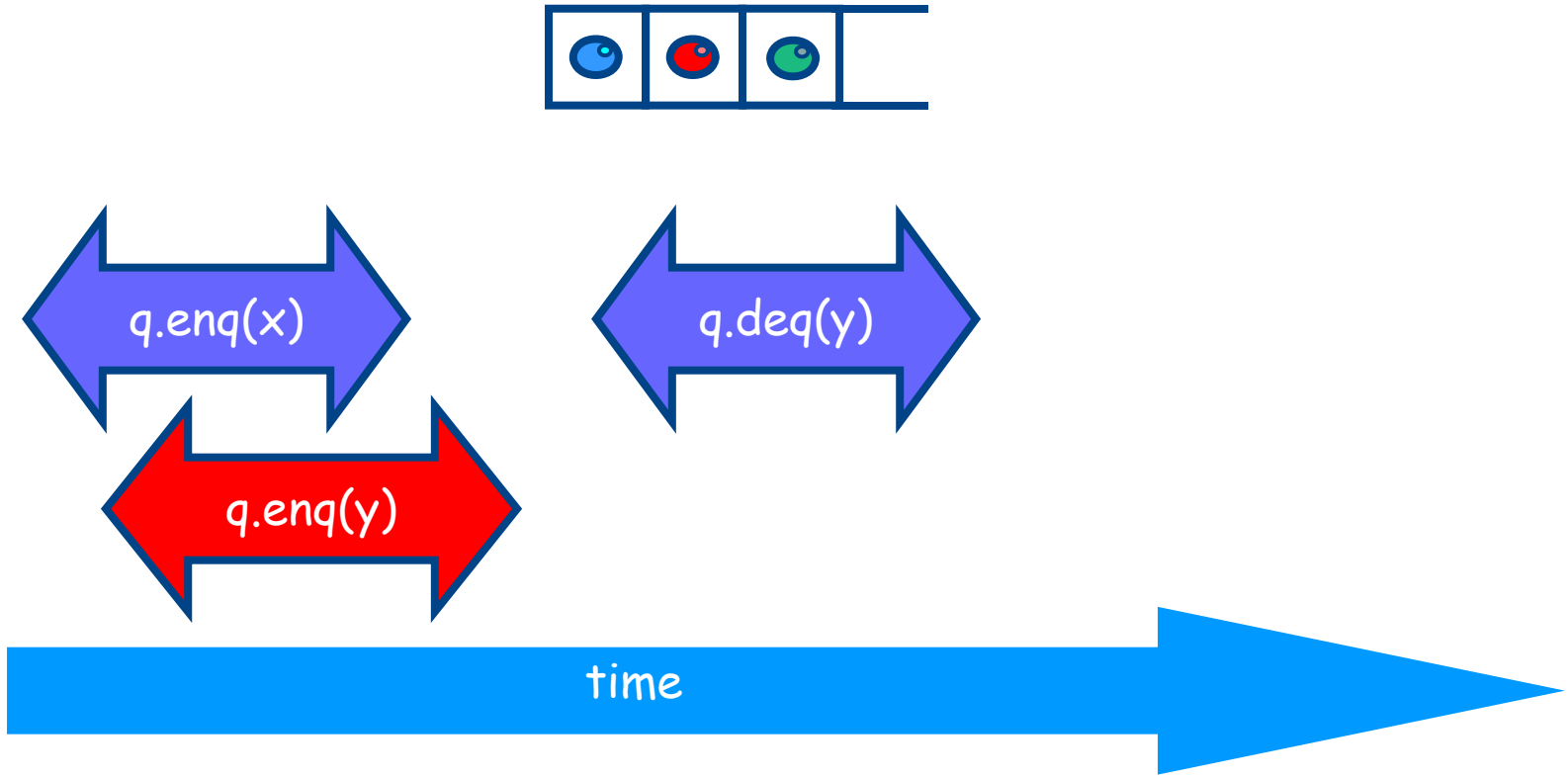
Example





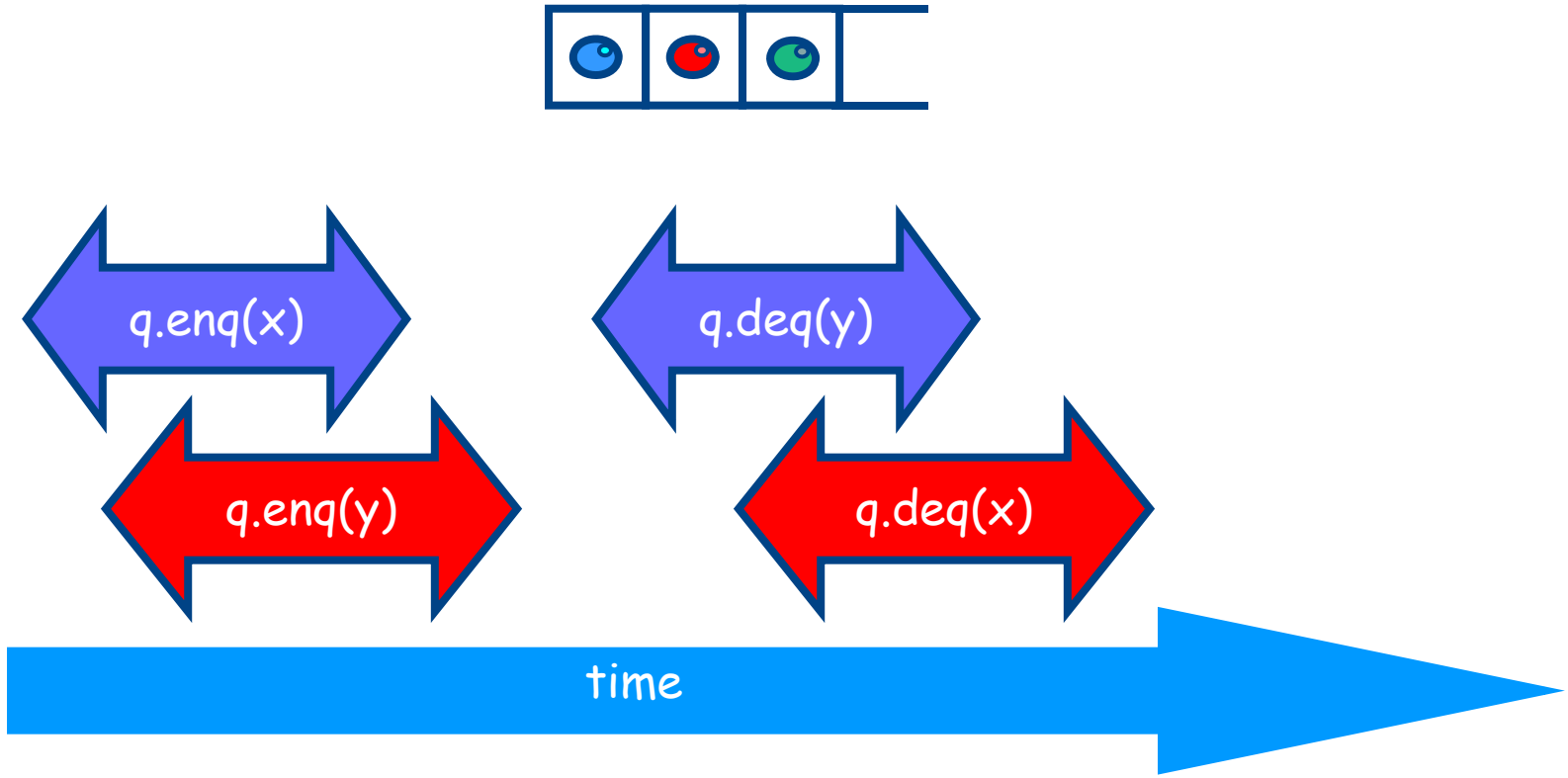
(8)

Example



(8)

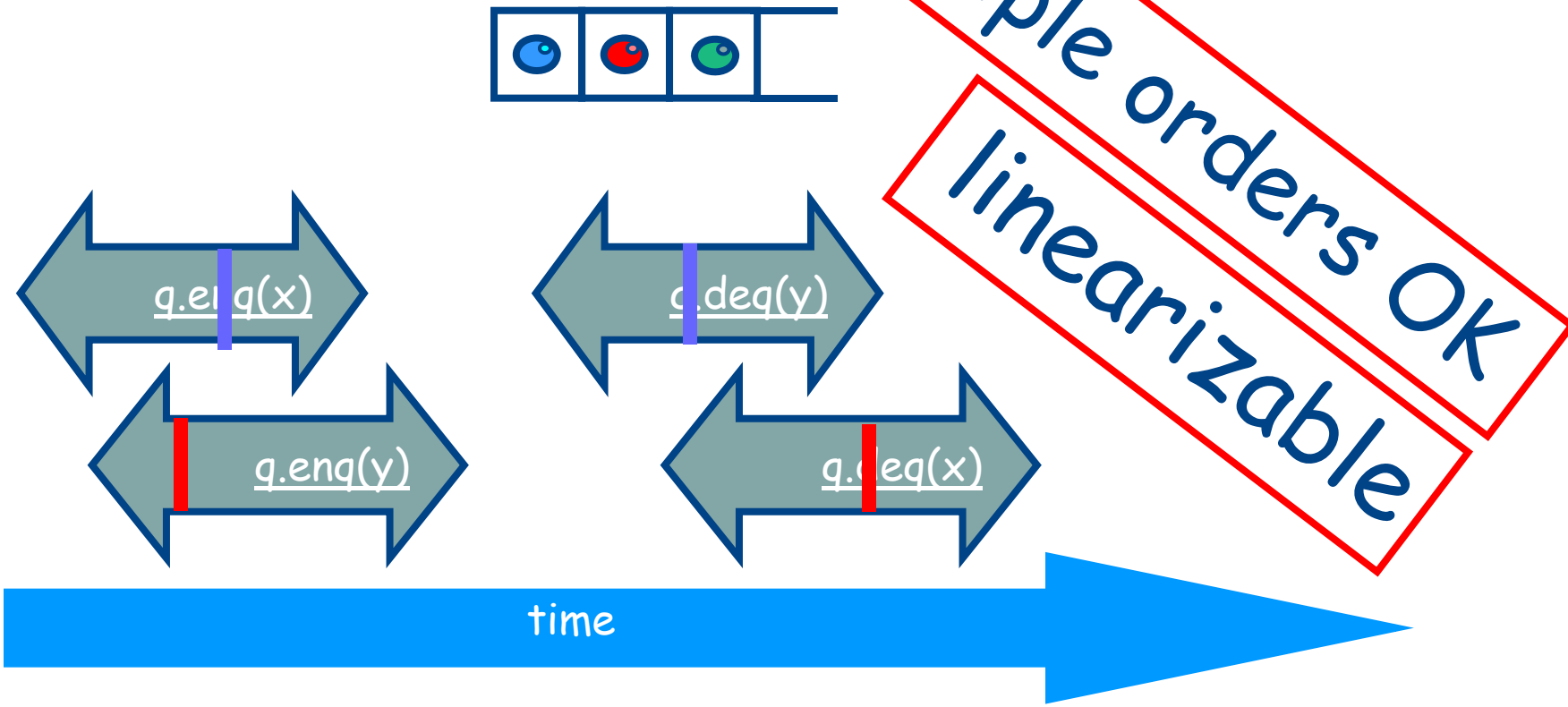
Example



(8)

Comme ci Comme ça

Example



Quiescent Consistency

Informally

- It says that any time an object becomes quiescent, then the execution so far is equivalent to some sequential execution of the completed calls.
- As an example in a priority queue the returned object is the one which had highest priority when the queue was check. But before removal started another object with higher priority was added.

Compositionality

- **Why compositionality matters?**
  - Modularity
- **What properties are compositional?**
  - Linearizability
  - Quiescent Consistency

FREEDOM

XXX-Free Hierarchy

- **Wait-Free** Algorithms (the best)
  - All threads complete in finite count of steps
  - Low priority threads cannot block high priority threads
  - No priority inversion possible
- **Lock-Free** (this work)
  - Every successful step makes Global Progress
  - But individual threads may starve
    - Hence priority inversion is possible
  - No live-lock
- **Obstruction-Free**
  - A single thread in isolation completes in finite count of steps
  - Threads may block each other
    - Hence live-lock is possible

LOGO

Linked Object Structures and Patterns

Linked Object Structures and Patterns

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

Linked Object Structures and Patterns

- **Introduce four “patterns”**
  - Fine Grained Synchronization
  - Optimistic Synchronization
  - Lazy Synchronization
  - Lock-Free Synchronization

First: 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

Second: Optimistic Synchronization

- **Search without locking …**
- **If you find it, lock and check …**
  - OK: we are done
  - Oops: start over
- **Evaluation**
  - Usually cheaper than locking
  - Mistakes are expensive

Third: Lazy Synchronization

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

Fourth: Lock-Free Synchronization

- **Don’t use locks at all**
  - Use compareAndSet() & relatives …
- **Advantages**
  - No Scheduler Assumptions/Support
- **Disadvantages**
  - Complex
  - Sometimes high overhead

Linked Object Structure

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

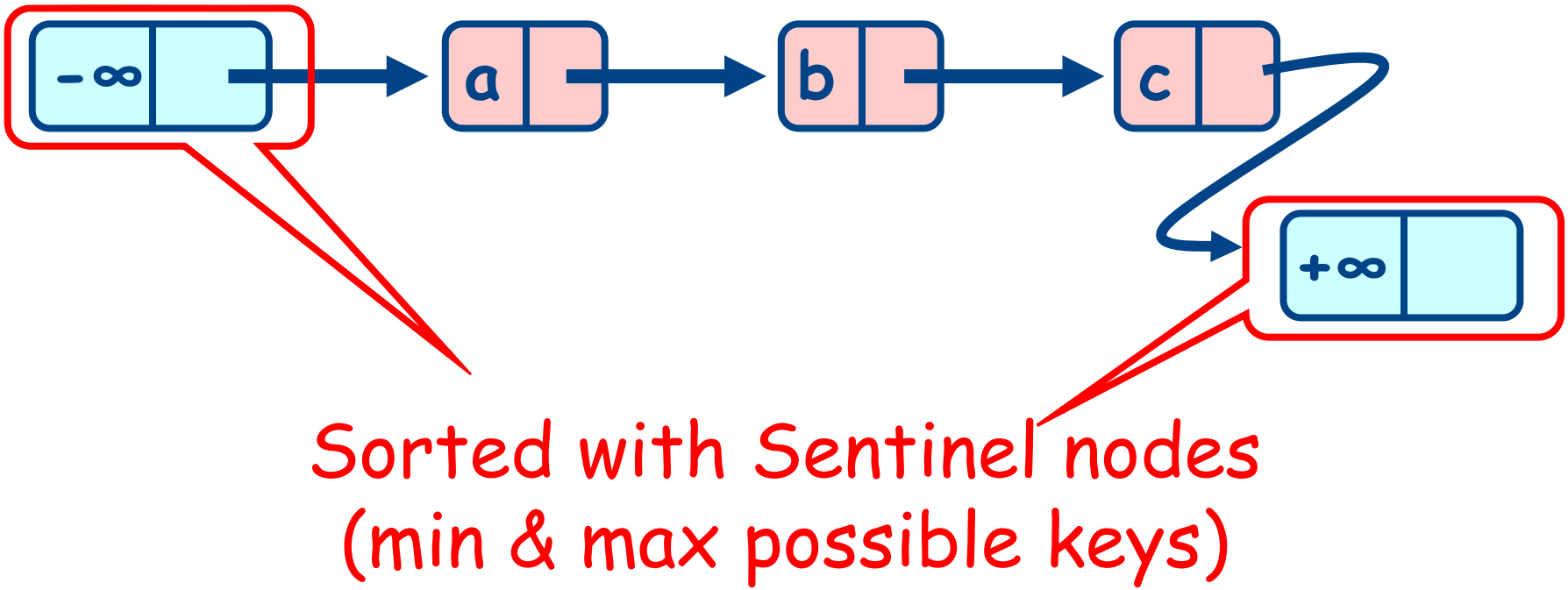
Interface

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

List Node

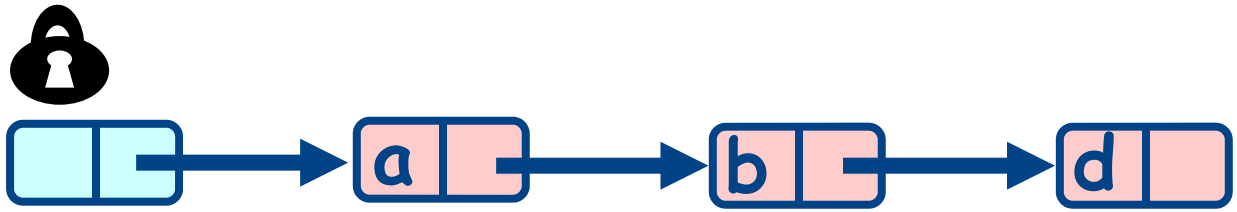
```
public class Node {
 public T item;
 public int key;
 public Node next;
}
```

The List-Based Set

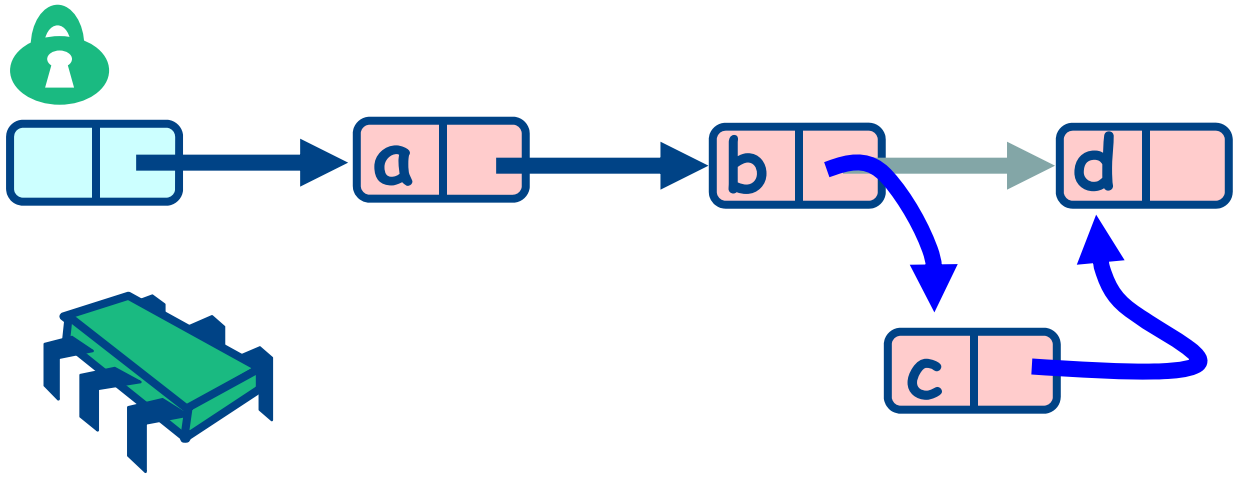


Sorted with Sentinel nodes
(min & max possible keys)

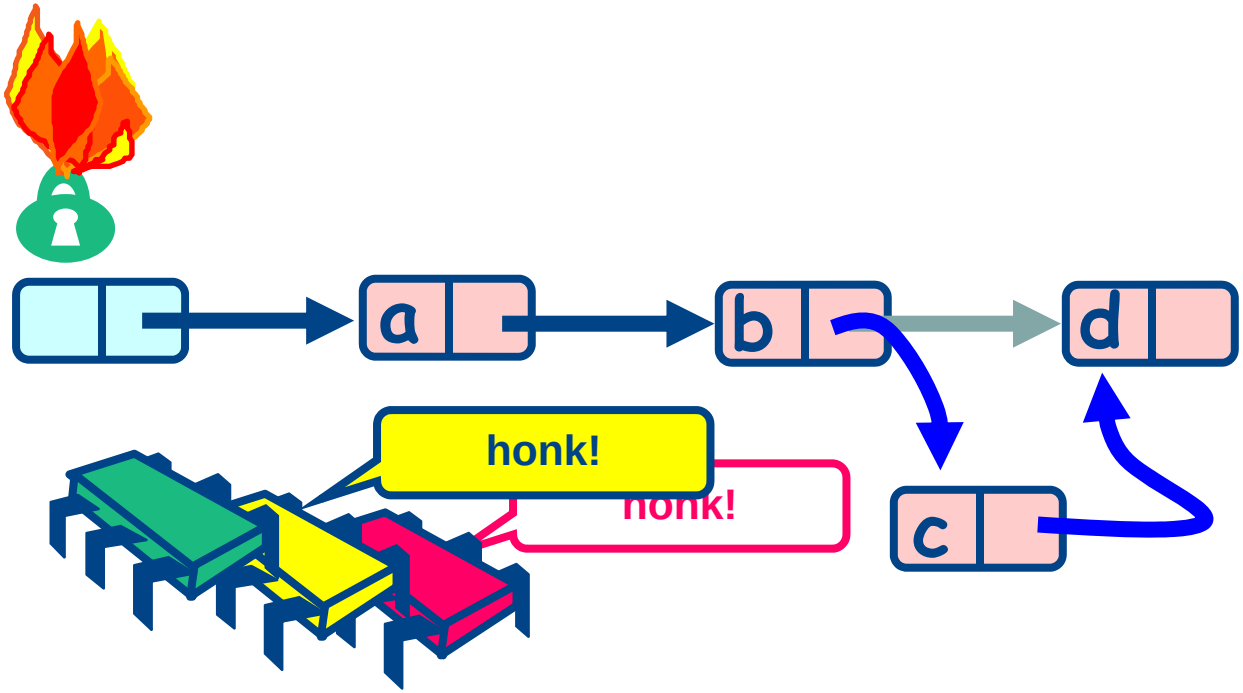
Course Grained Locking



Course Grained Locking



Course Grained Locking



Simple but hotspot + bottleneck

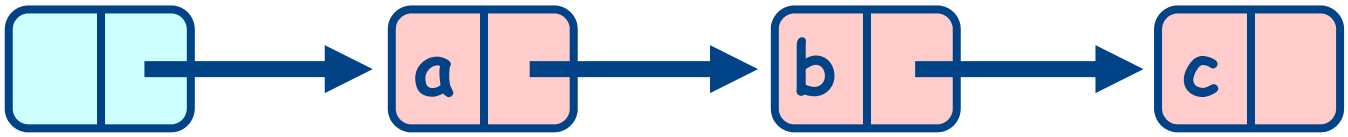
Coarse-Grained Locking

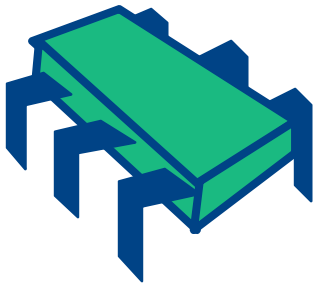
- **Easy, same as synchronized methods**
  - "One lock to rule them all …"
- **Simple, clearly correct**
  - Deserves respect!
- **Works poorly with contention**
  - Queue locks help
  - But bottleneck still an issue

Fine-grained Locking

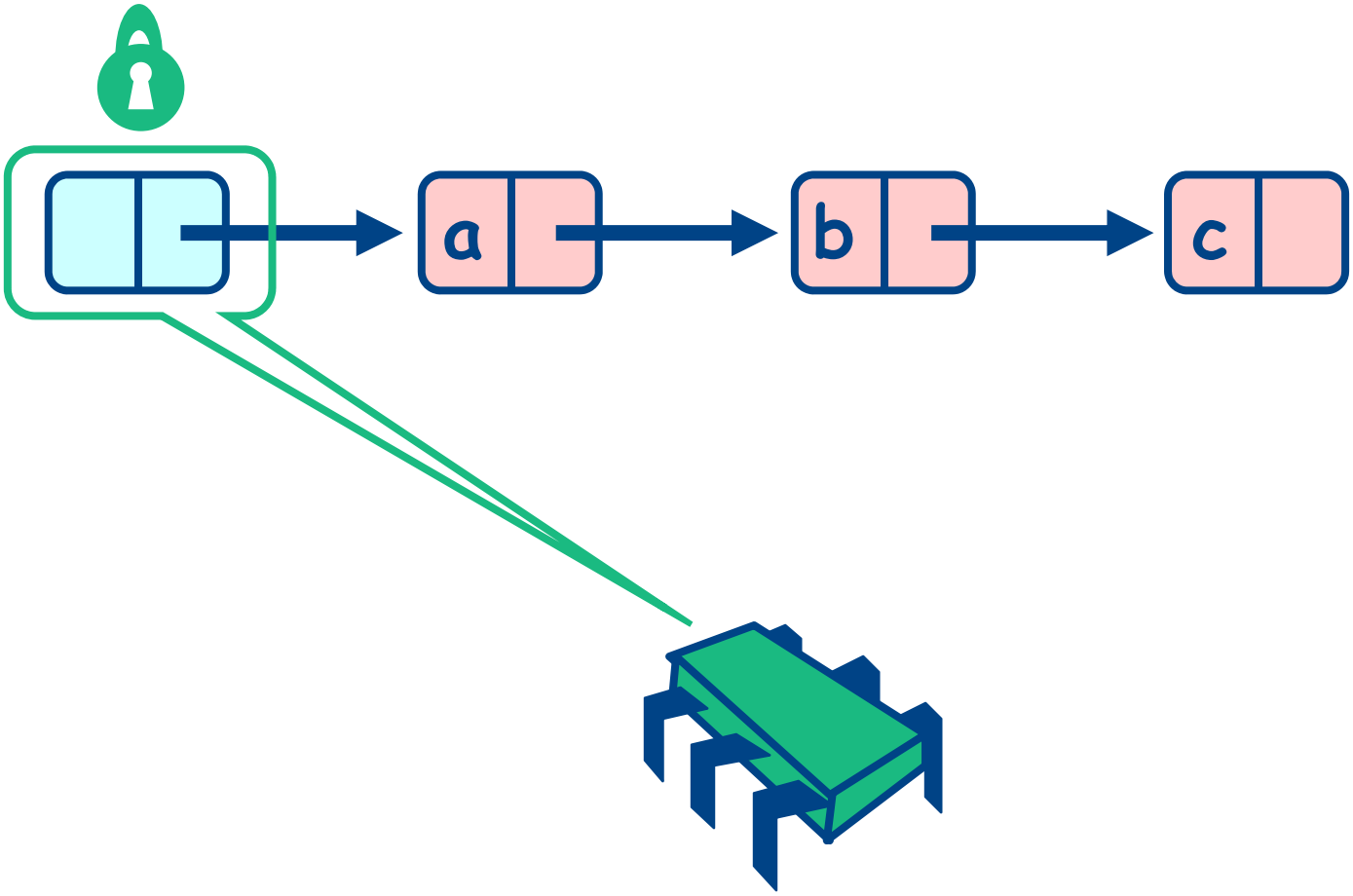
- **Requires careful thought**
  - "Do not meddle in the affairs of wizards, for they are subtle and quick to anger"
- **Split object into pieces**
  - Each piece has own lock
  - Methods that work on disjoint pieces need not exclude each other

Hand-over-Hand locking

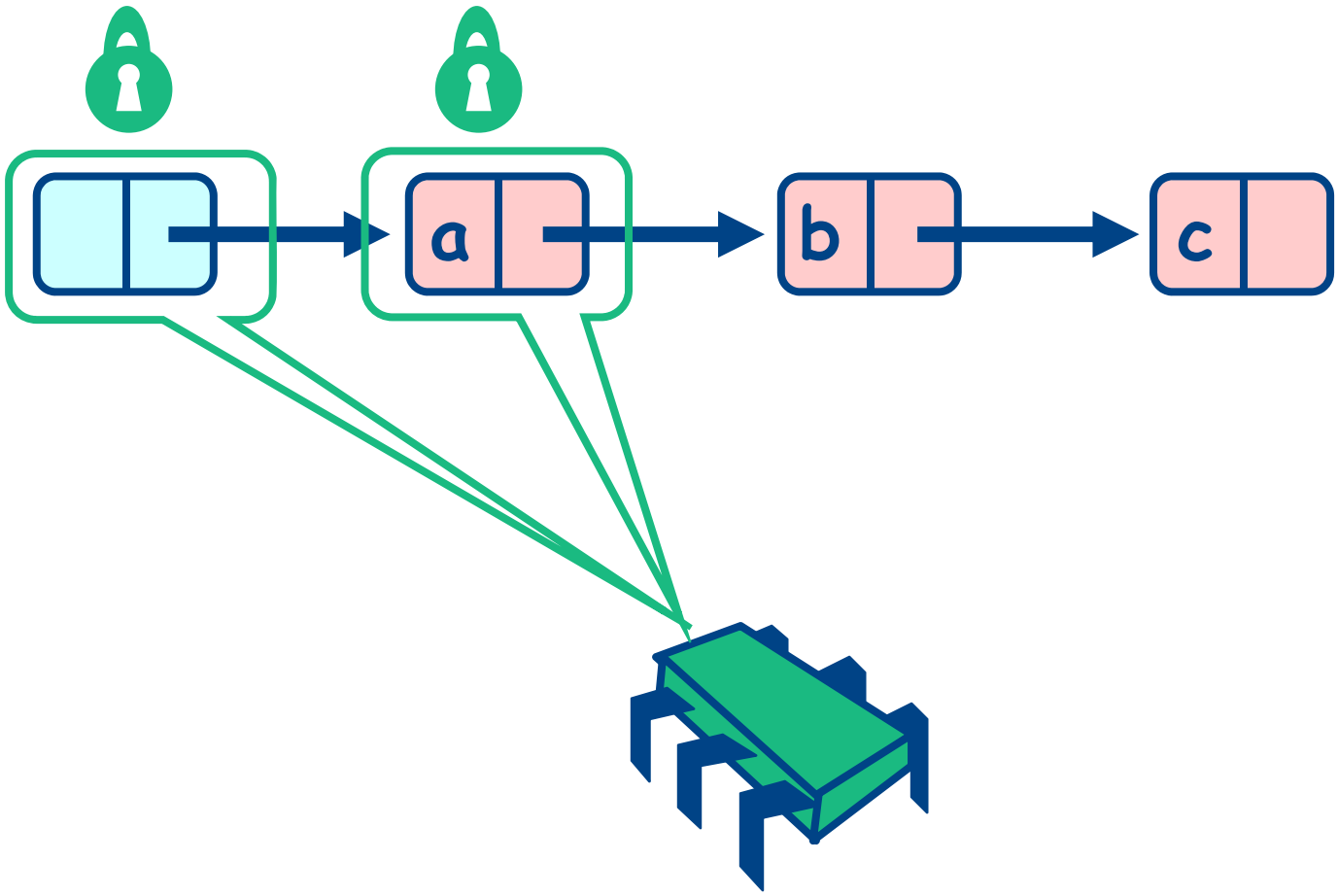




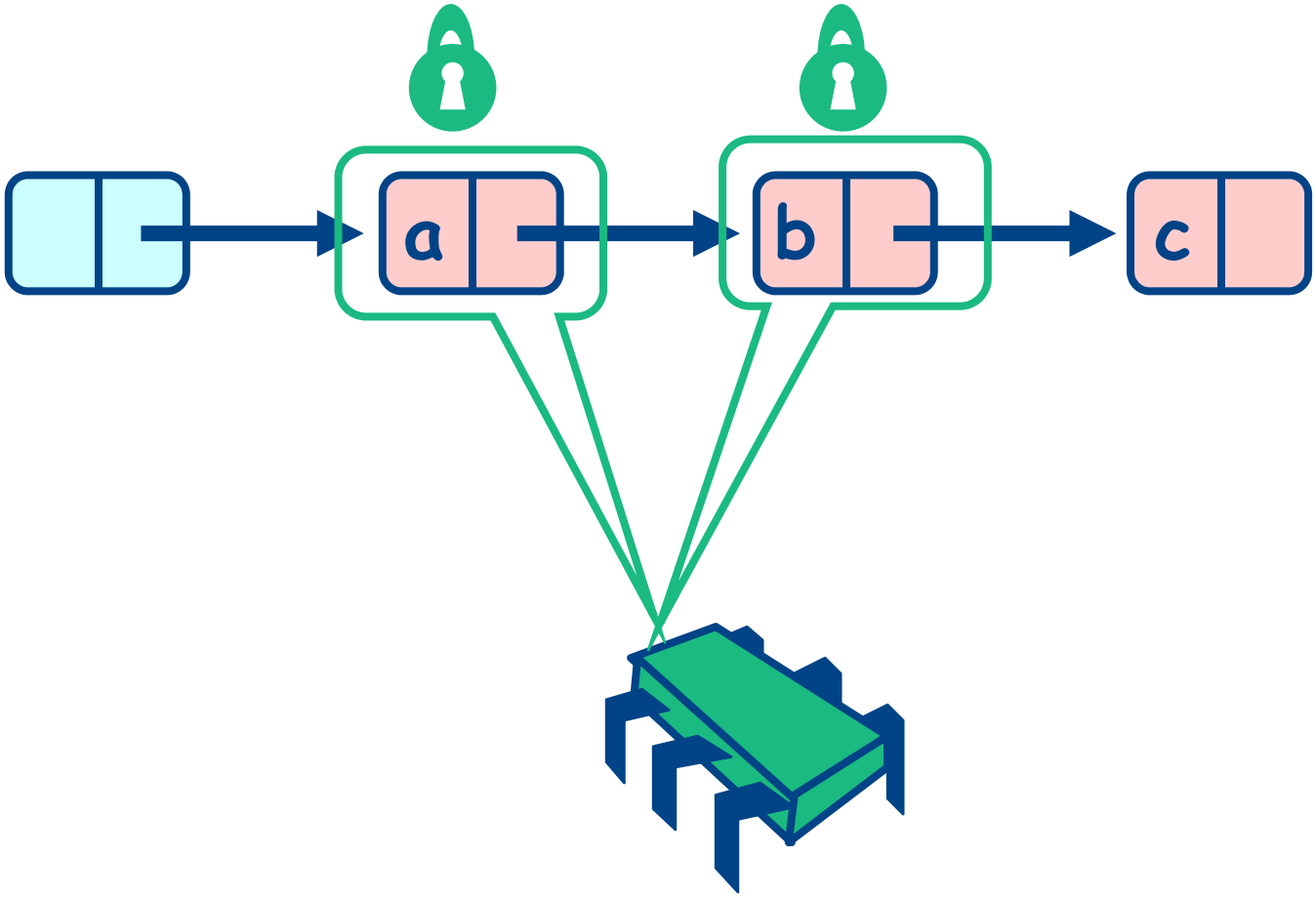
Hand-over-Hand locking



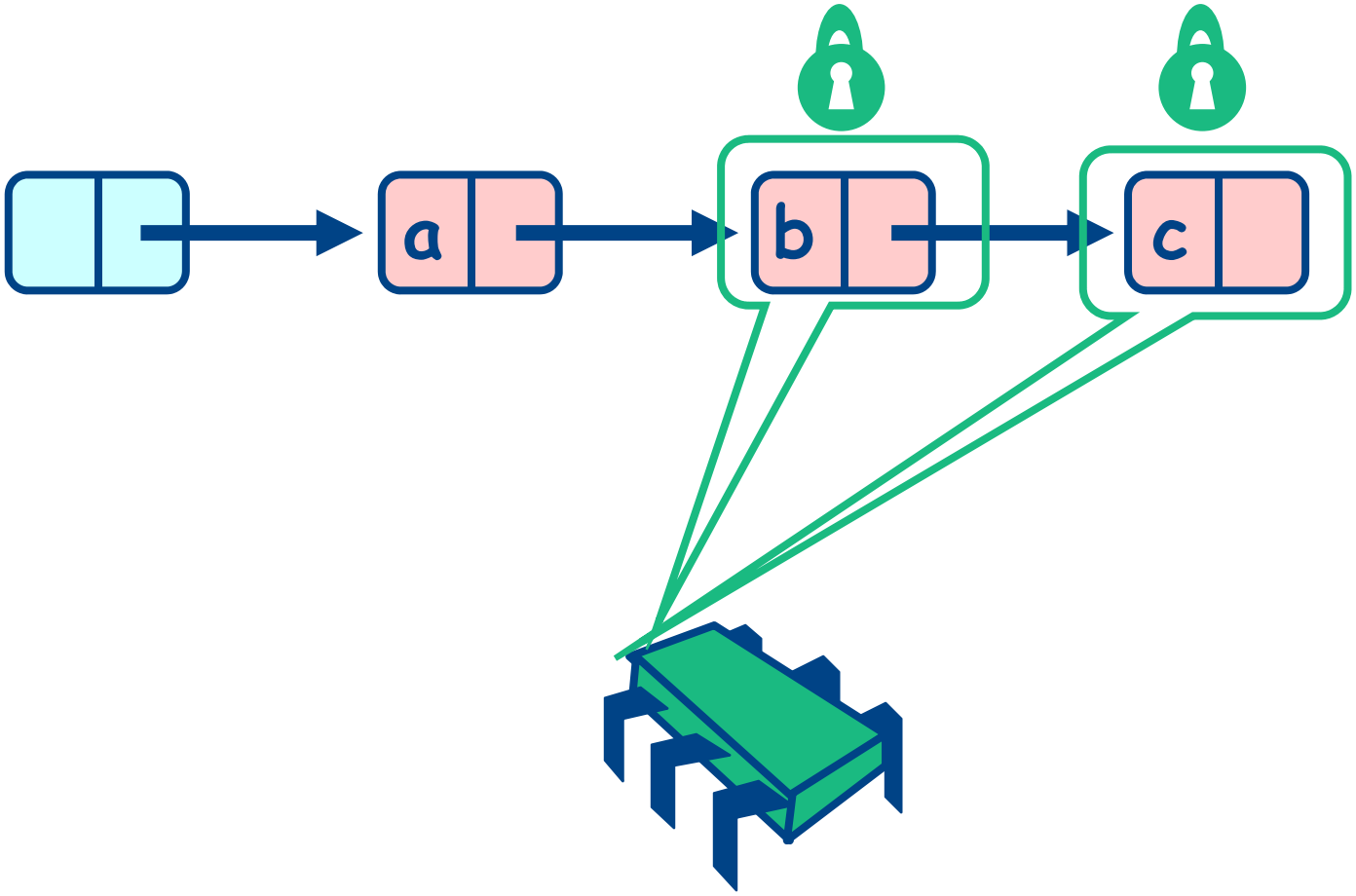
Hand-over-Hand locking



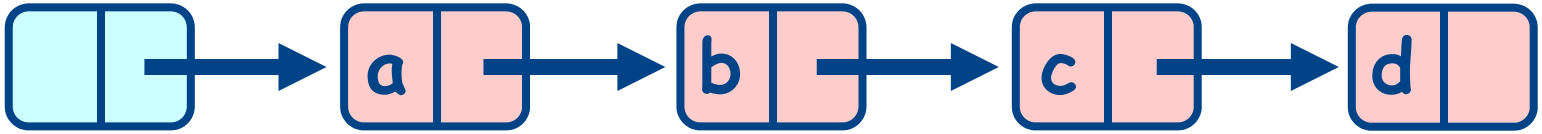
Hand-over-Hand locking



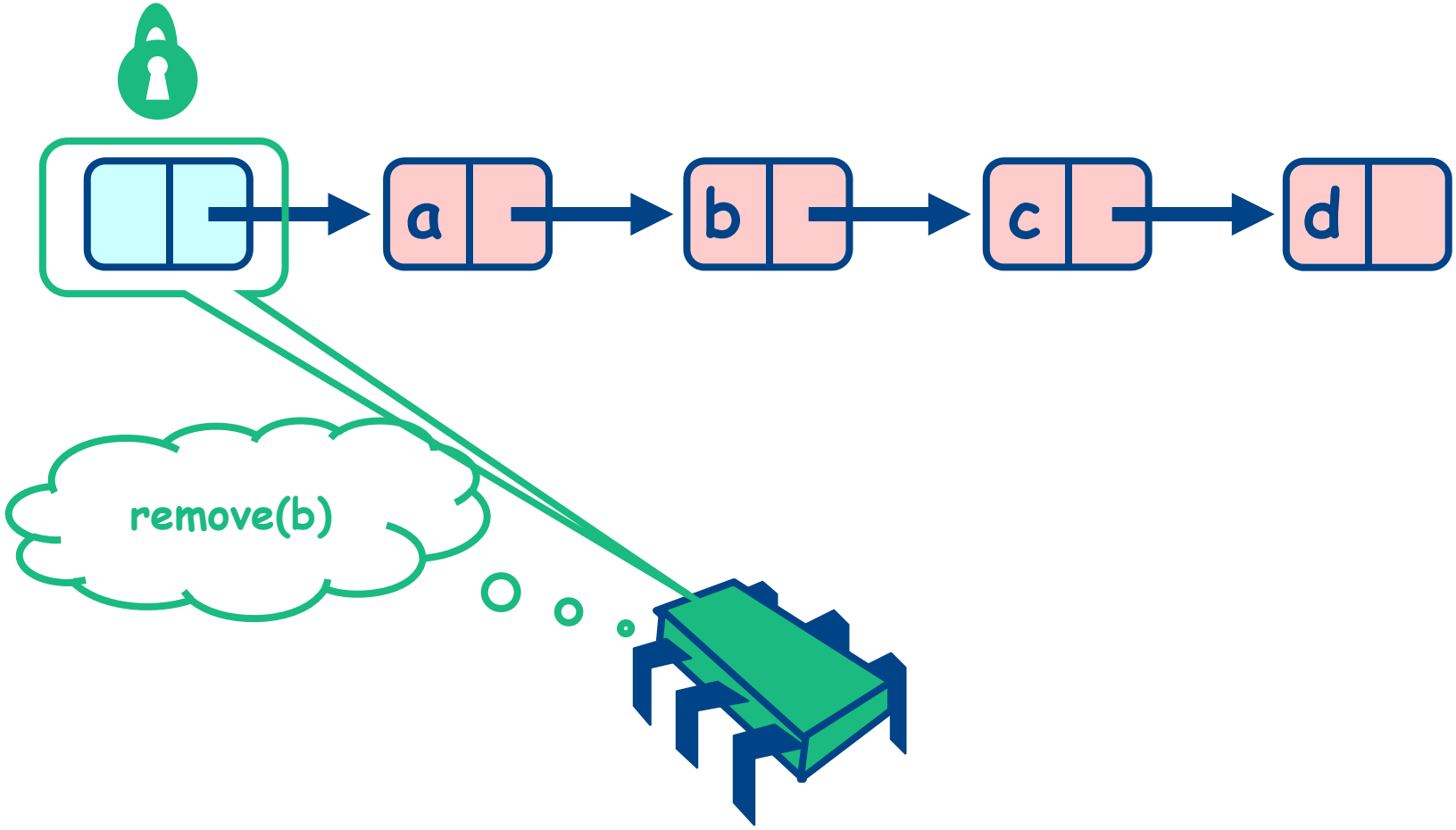
Hand-over-Hand locking



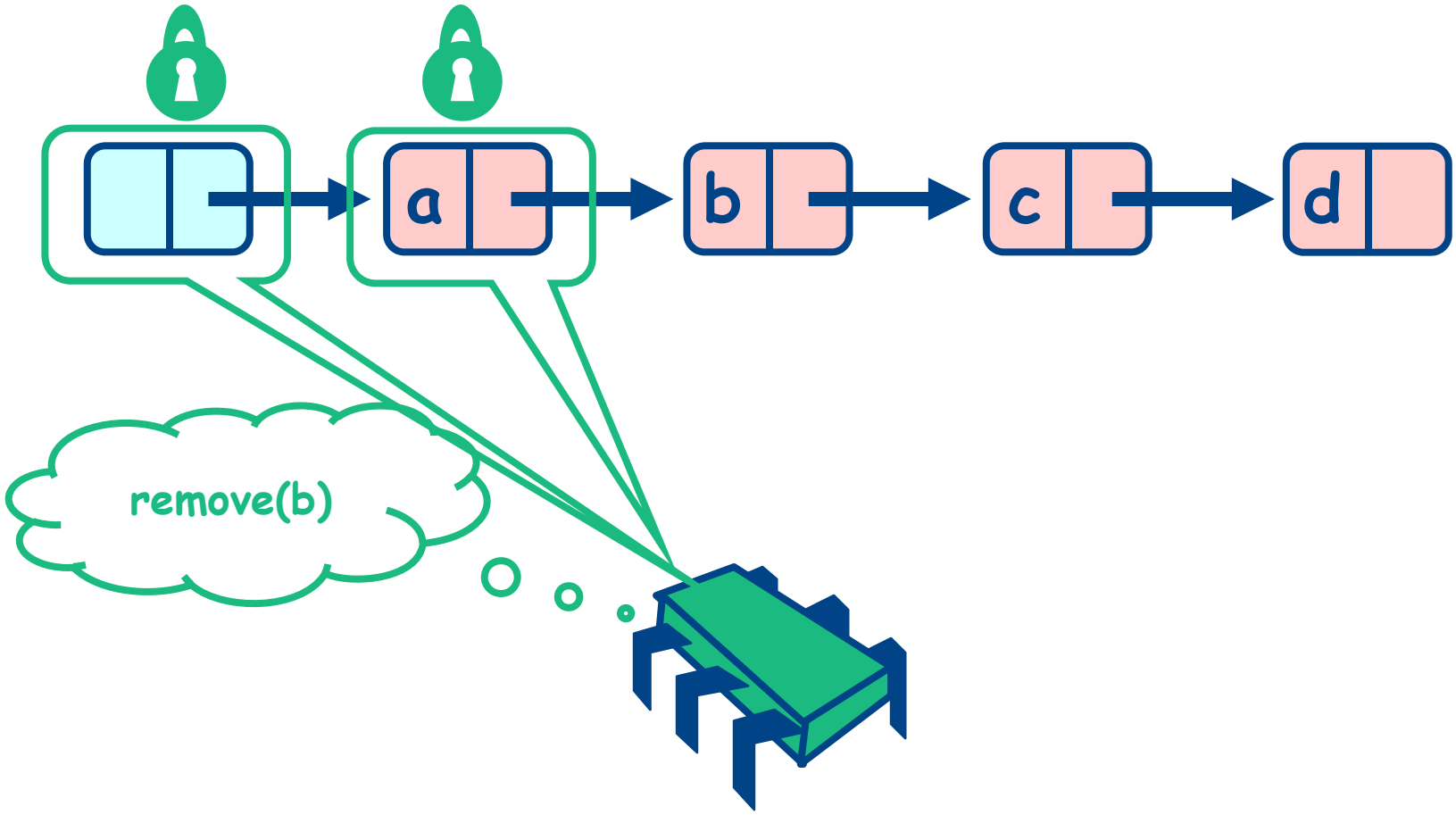
Removing a Node



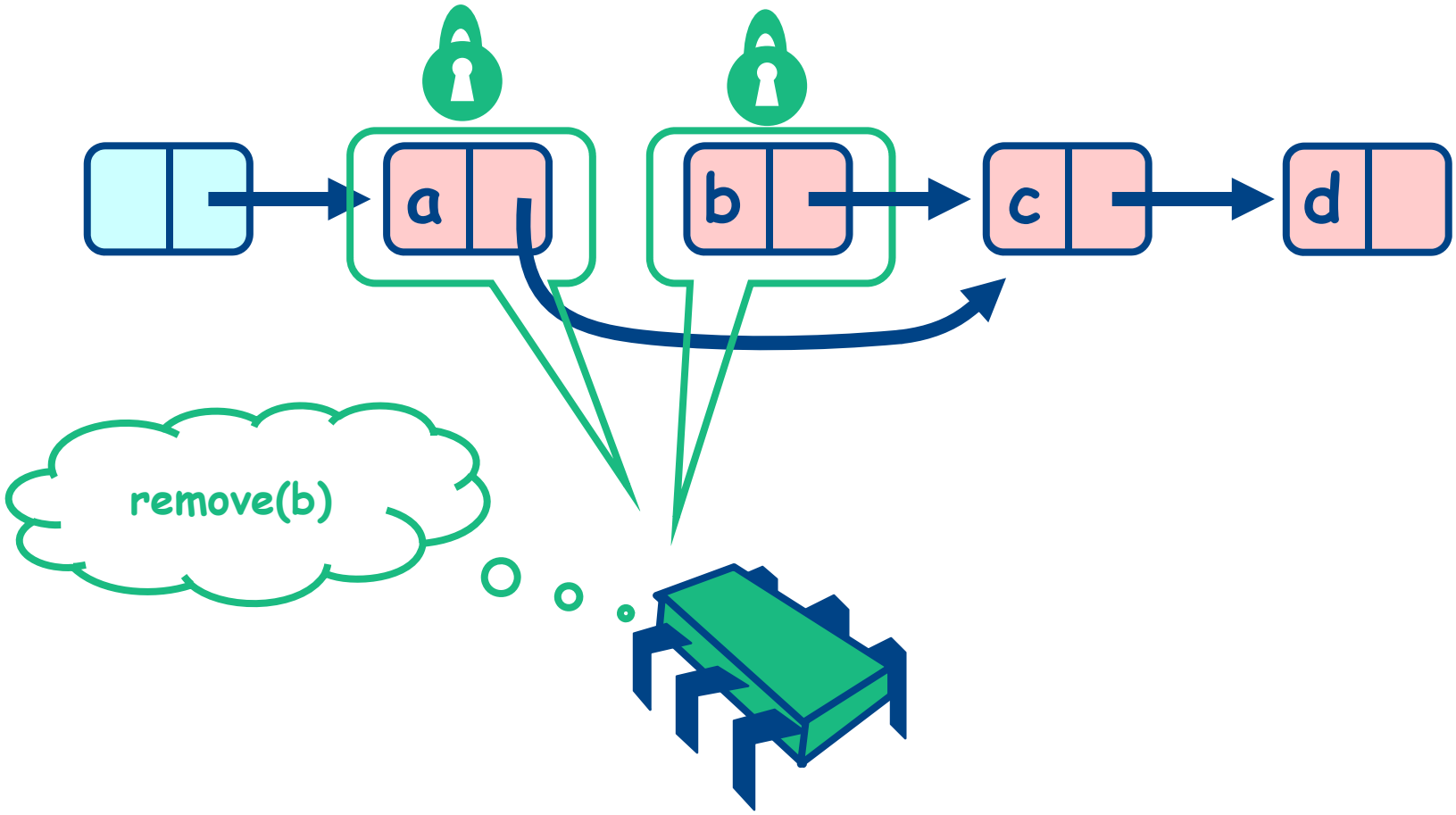
Removing a Node



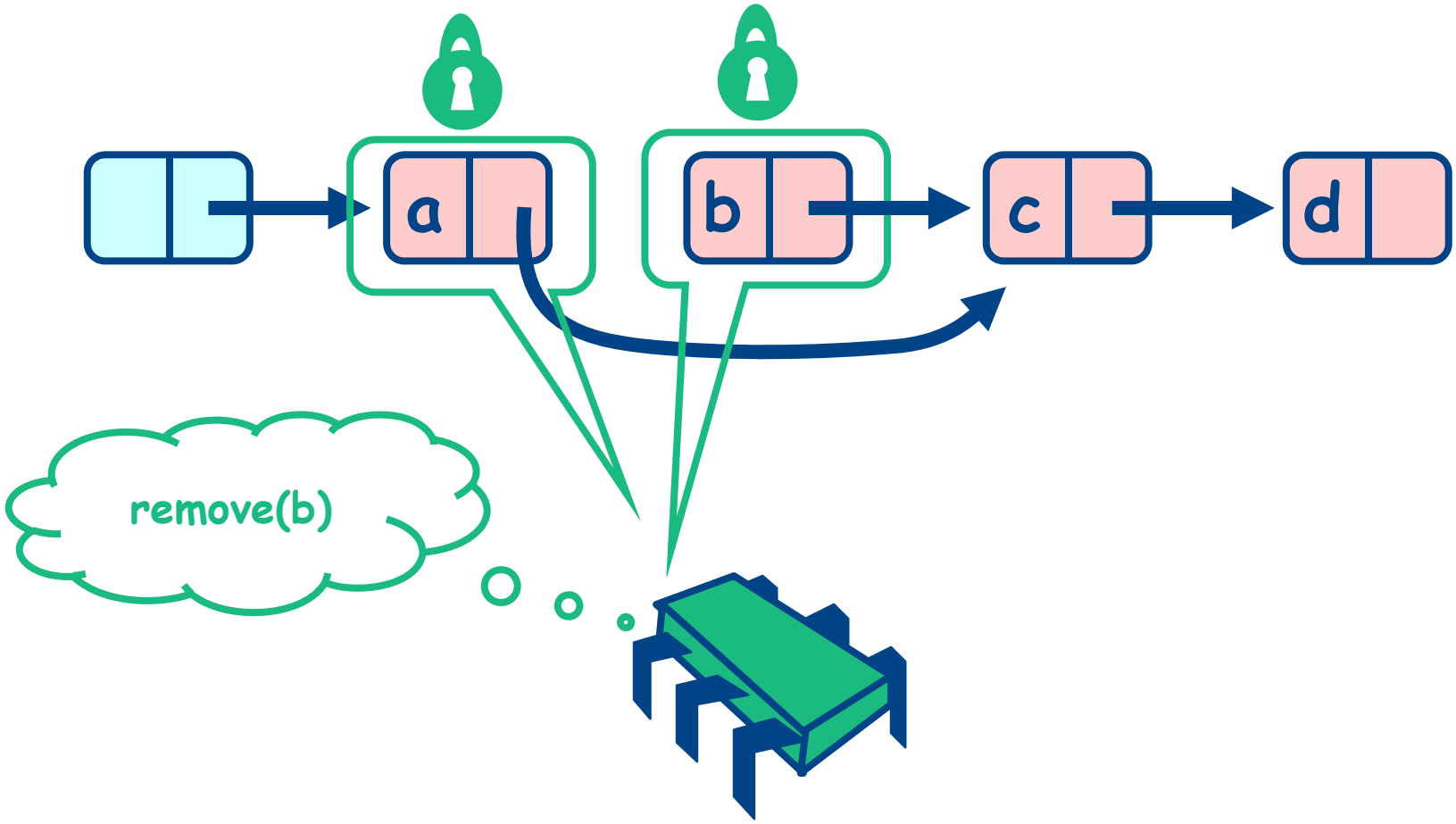
Removing a Node



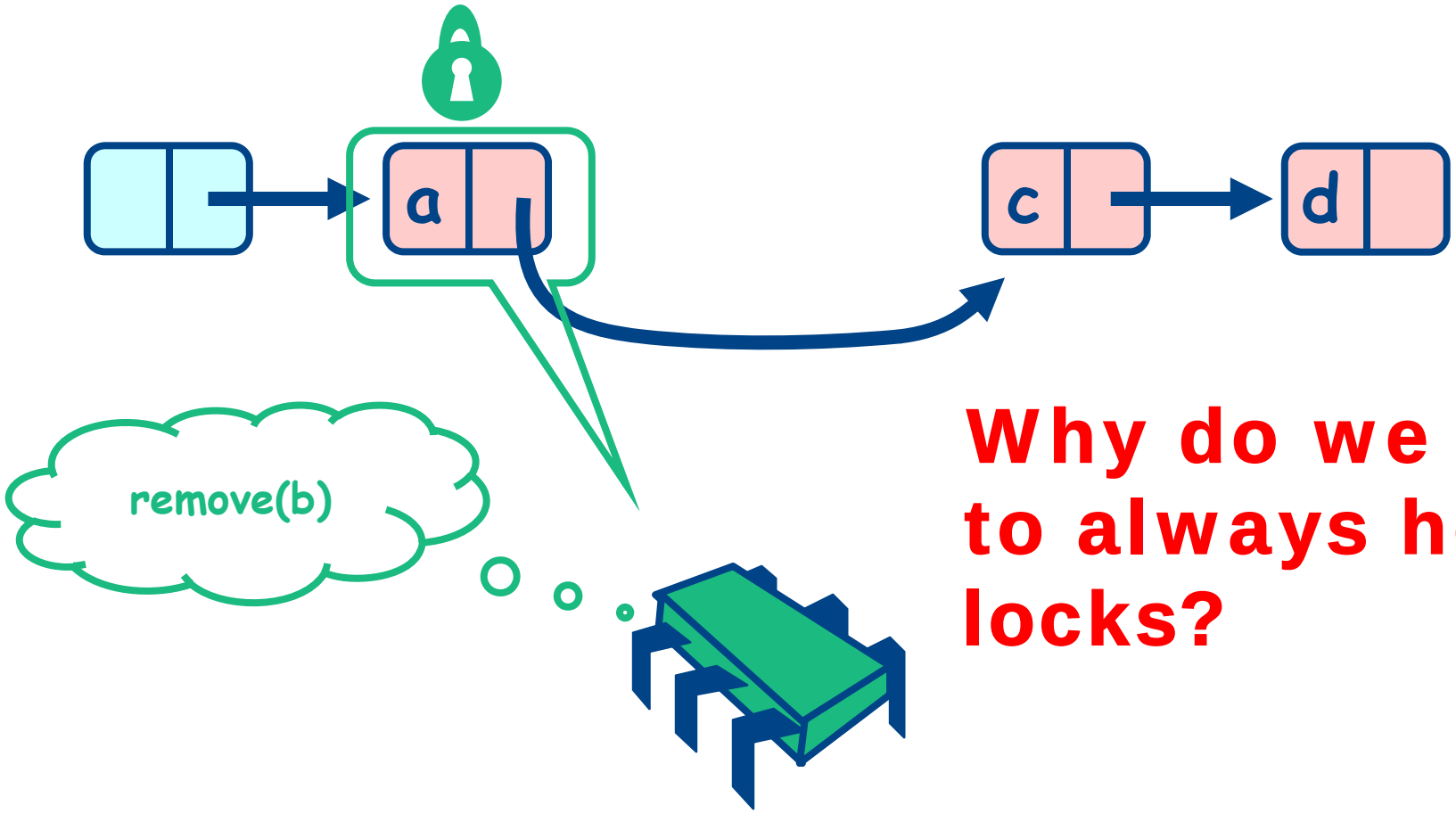
Removing a Node



Removing a Node

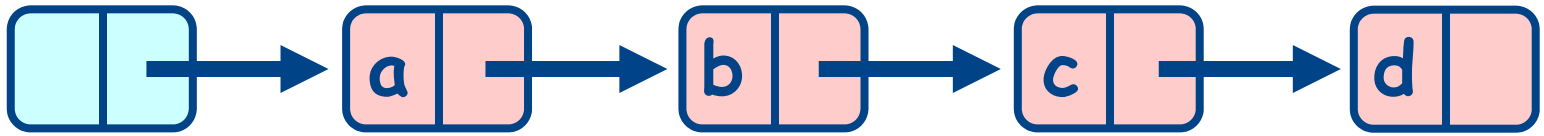


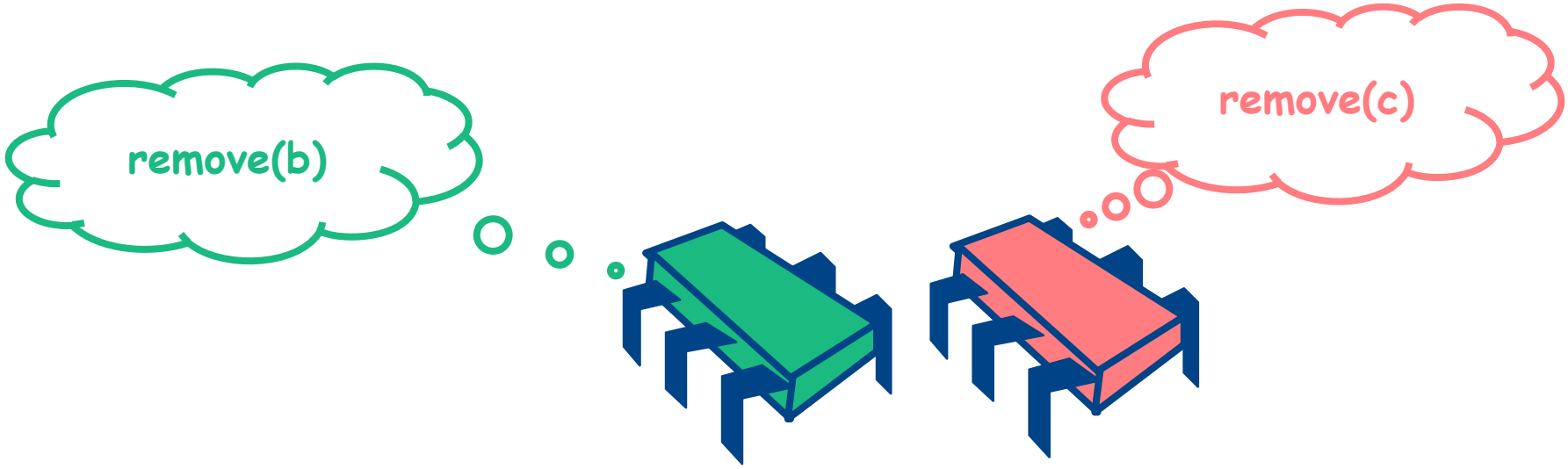
Removing a Node



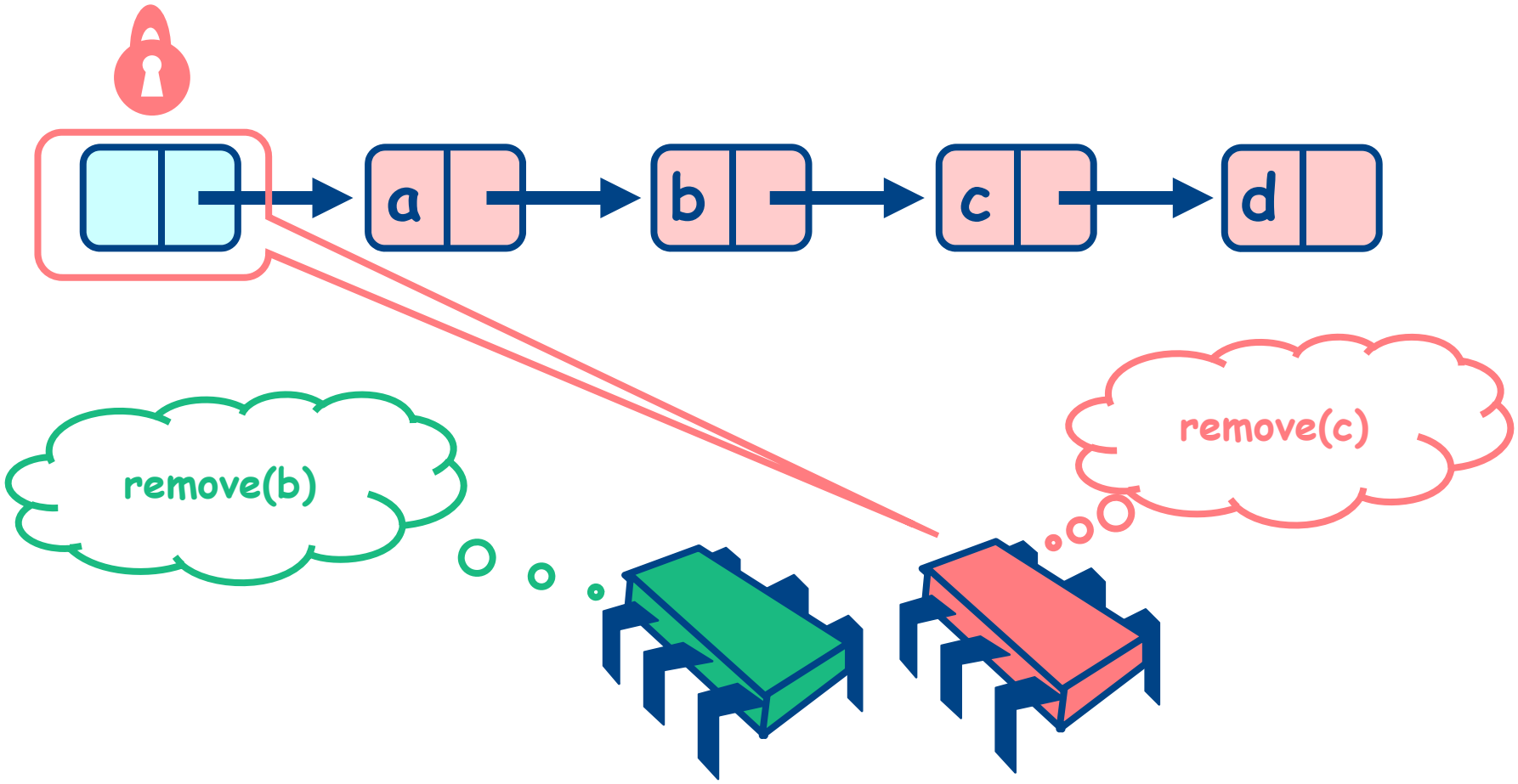
Why do we nee to always hold locks?

Concurrent Removes

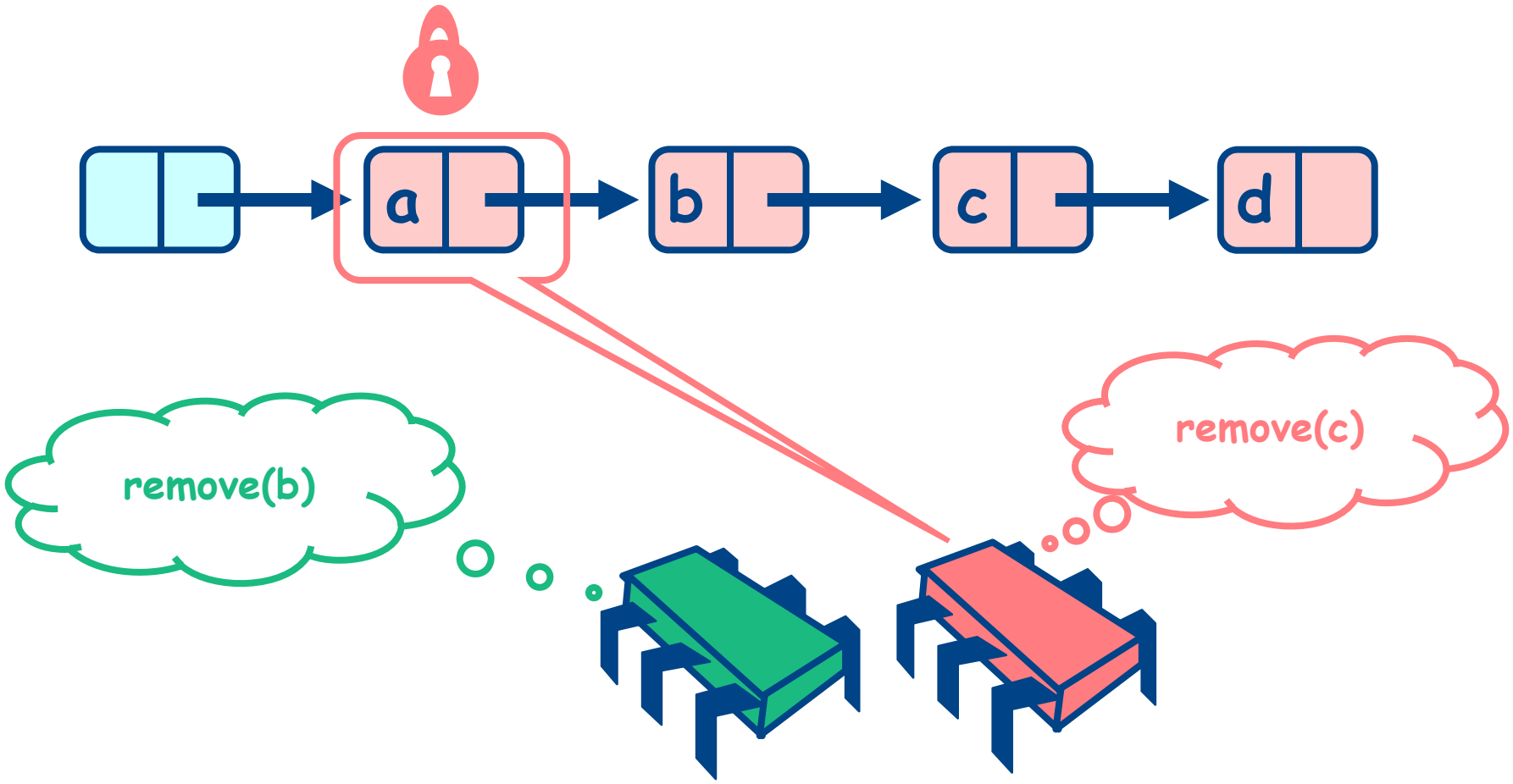




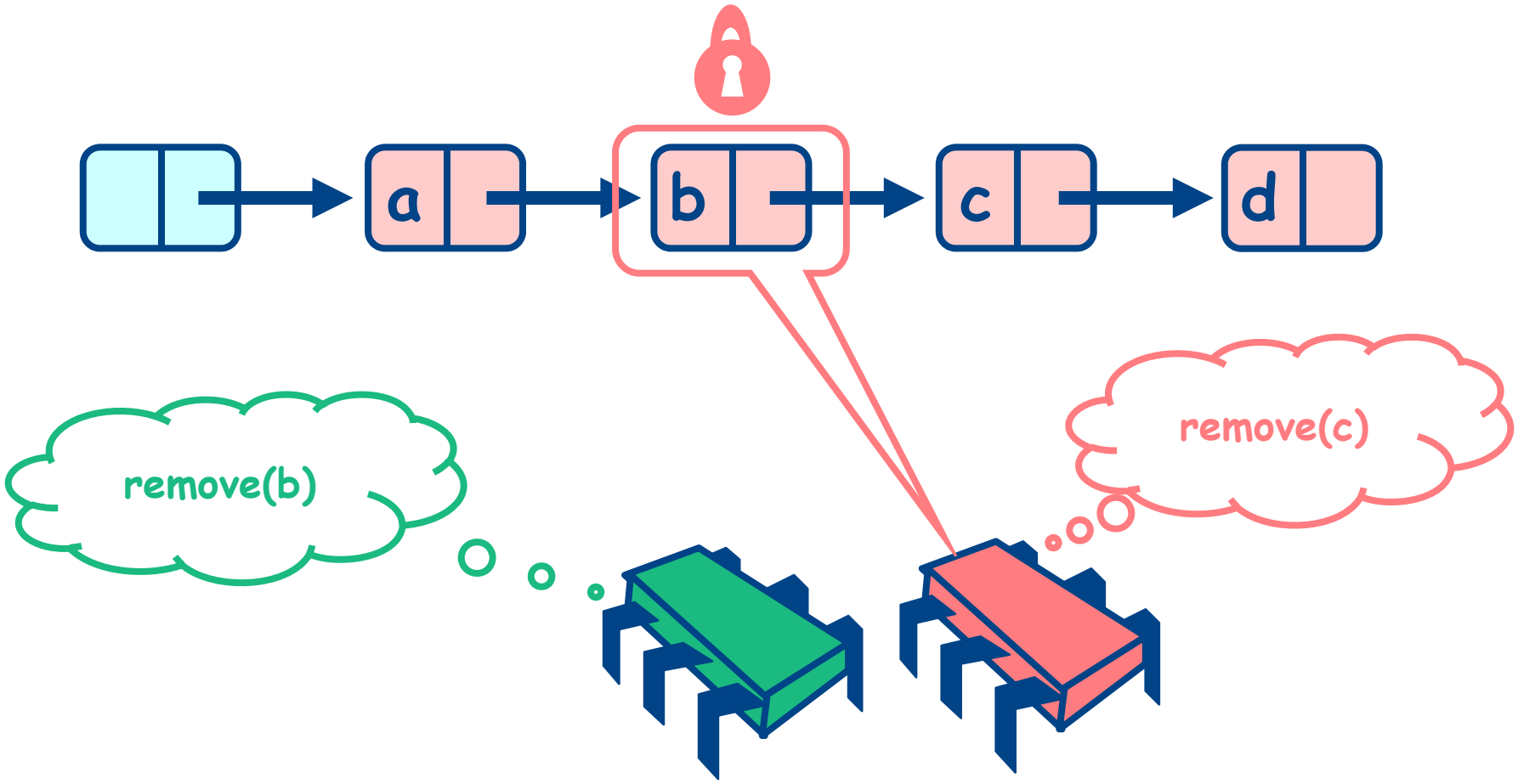
Concurrent Removes



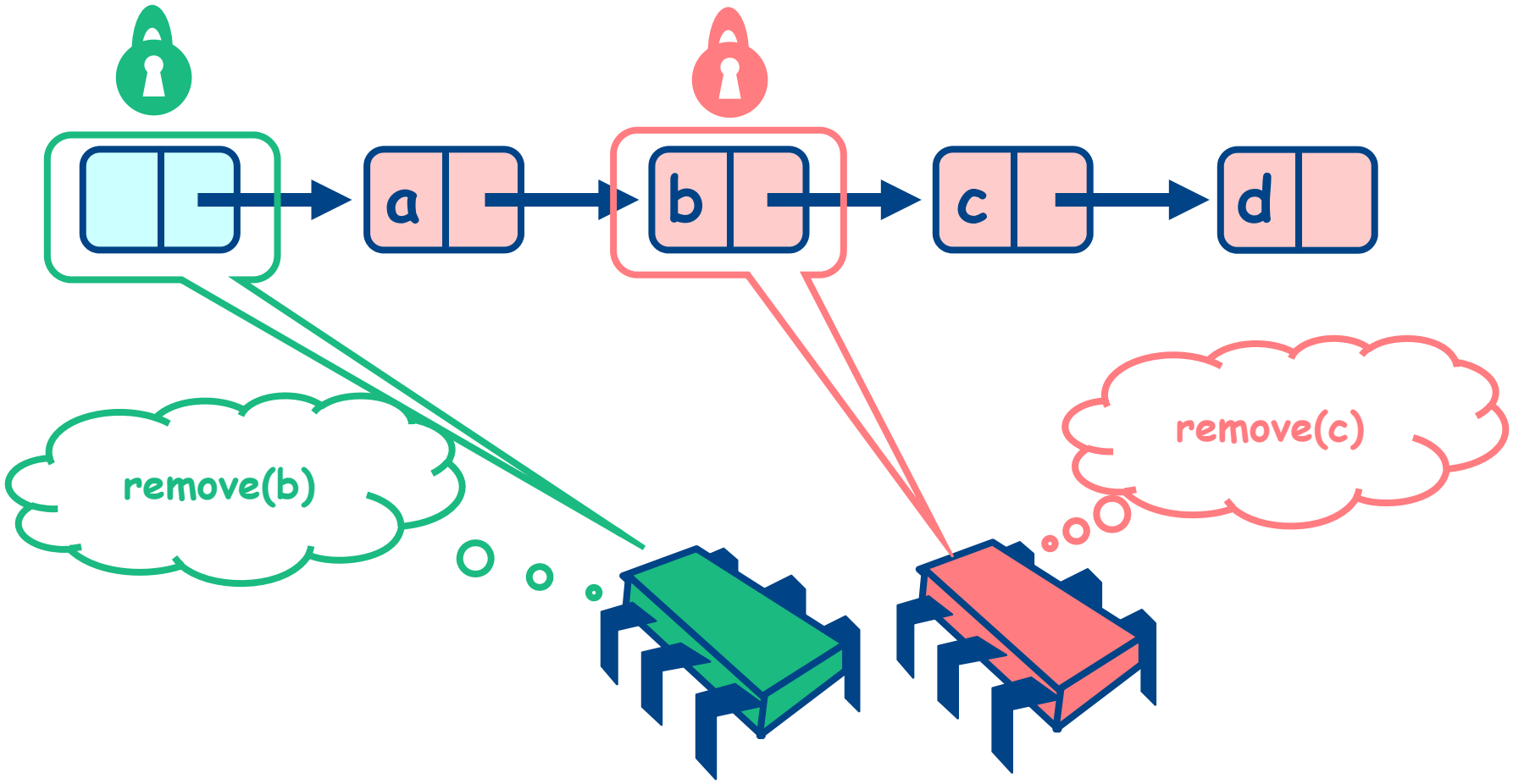
Concurrent Removes



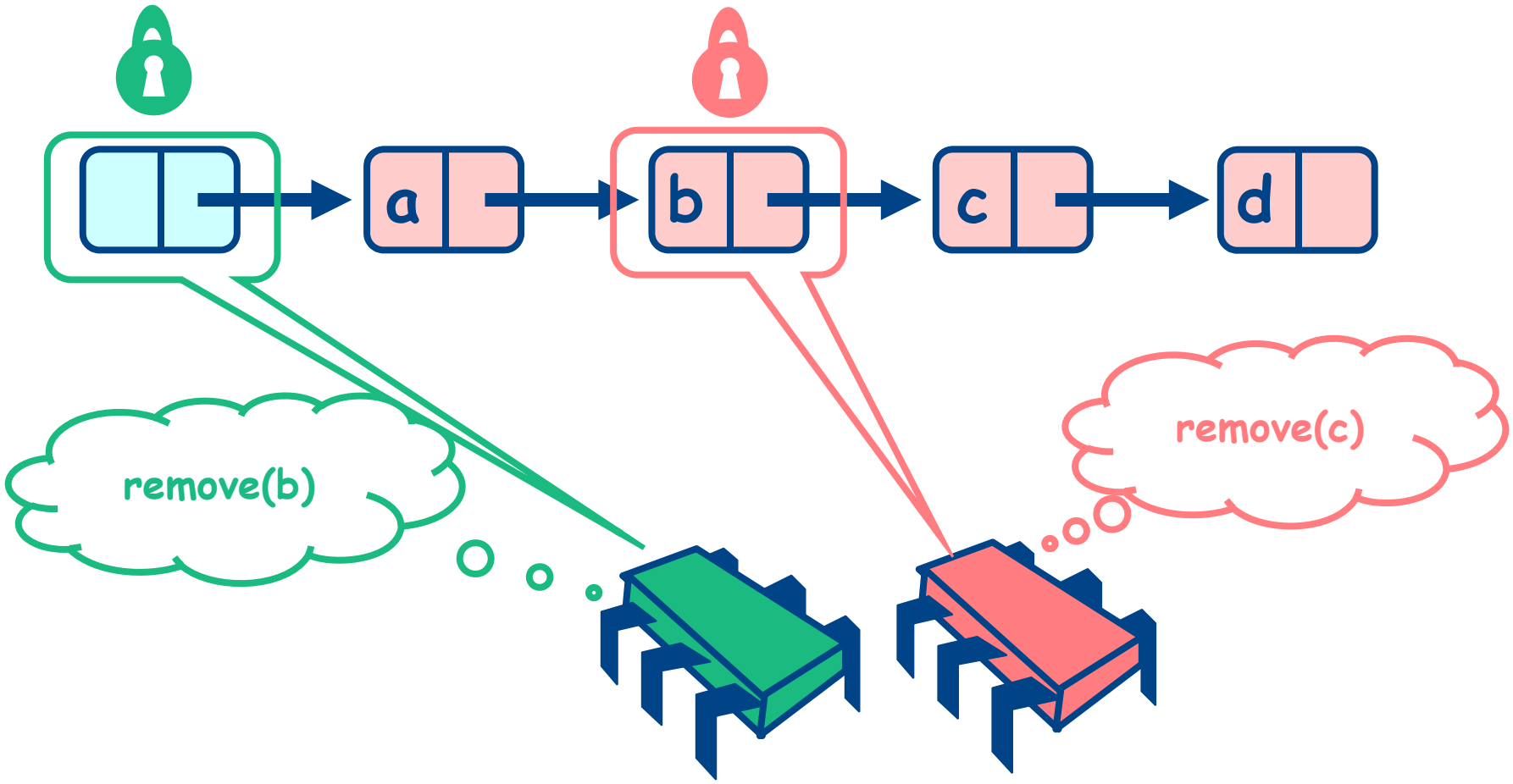
Concurrent Removes



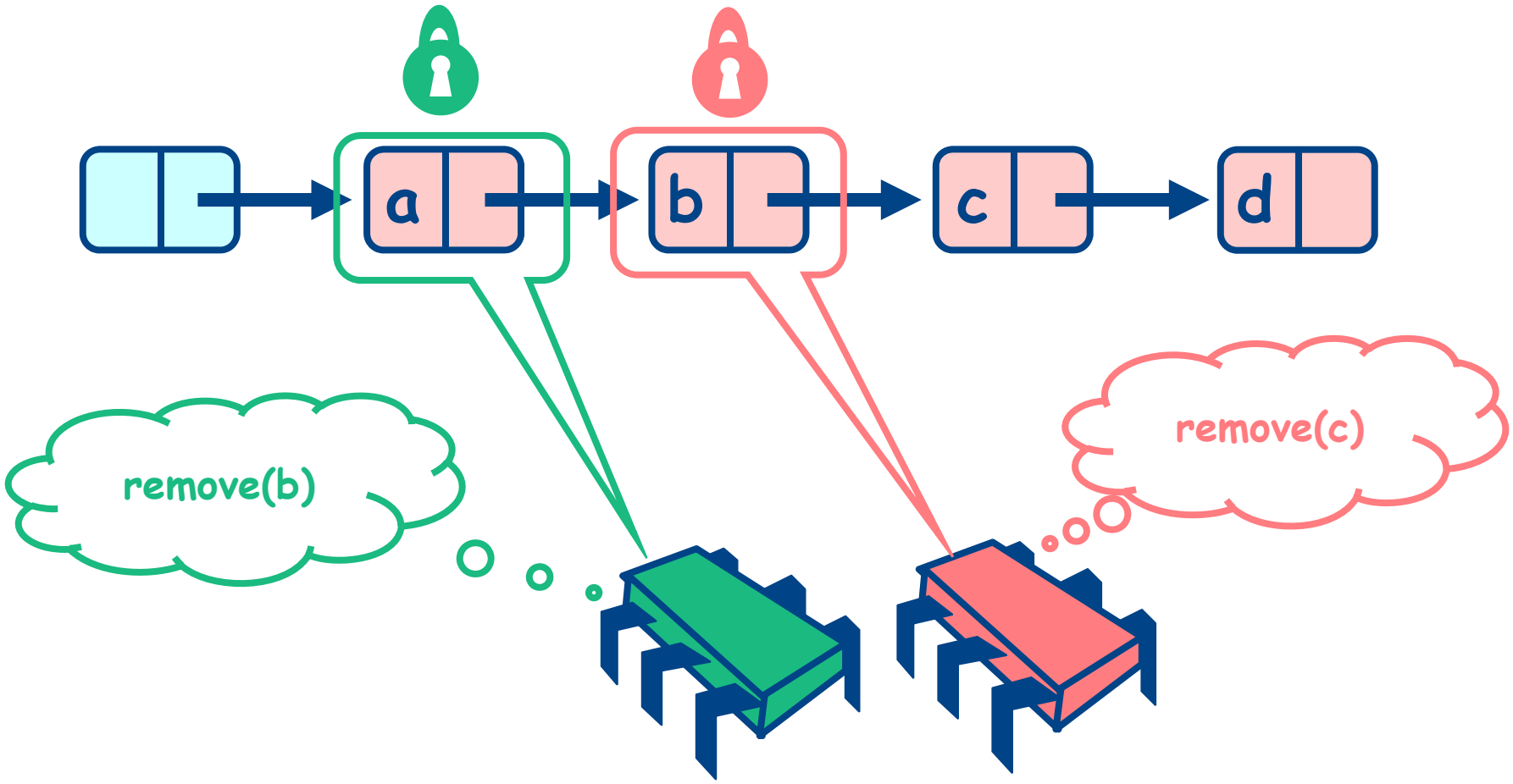
Concurrent Removes



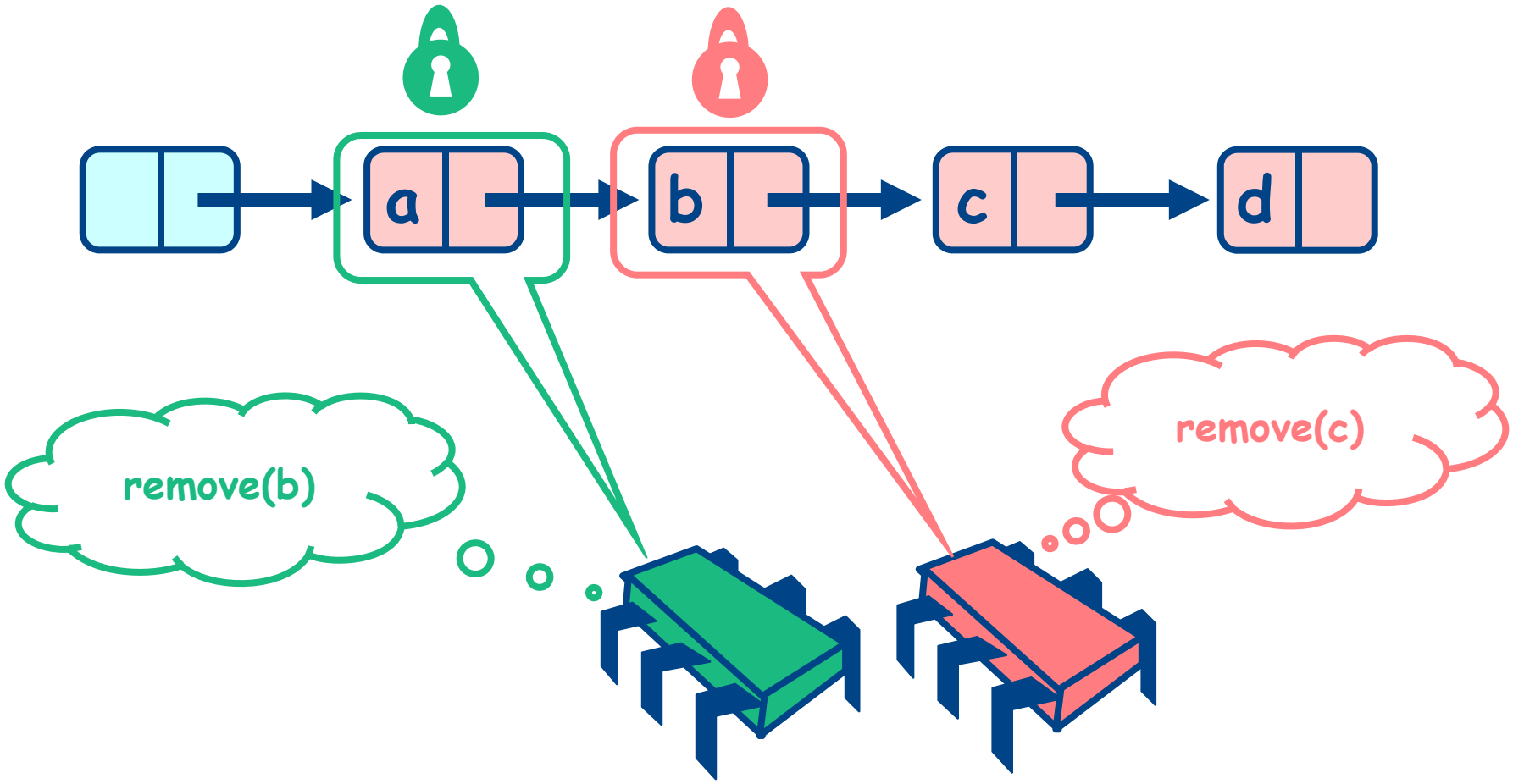
Concurrent Removes



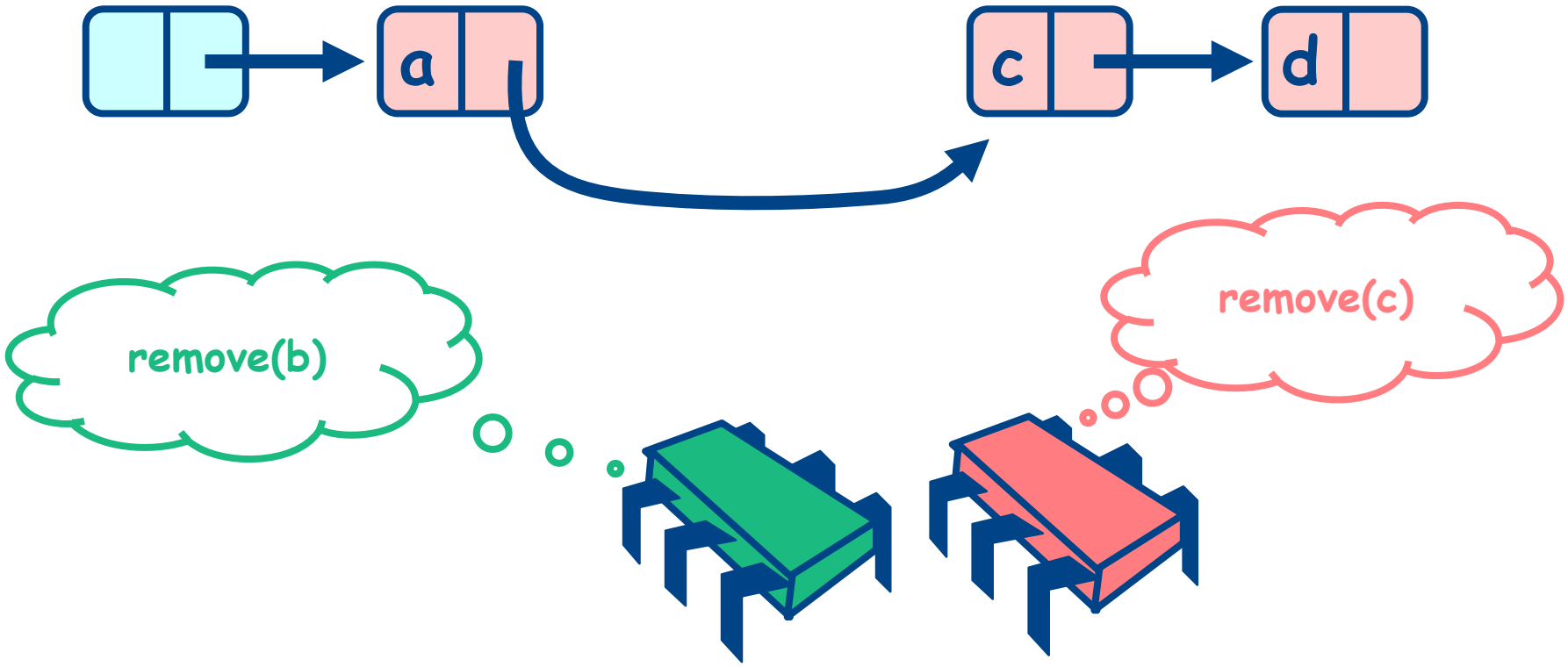
Concurrent Removes



Concurrent Removes

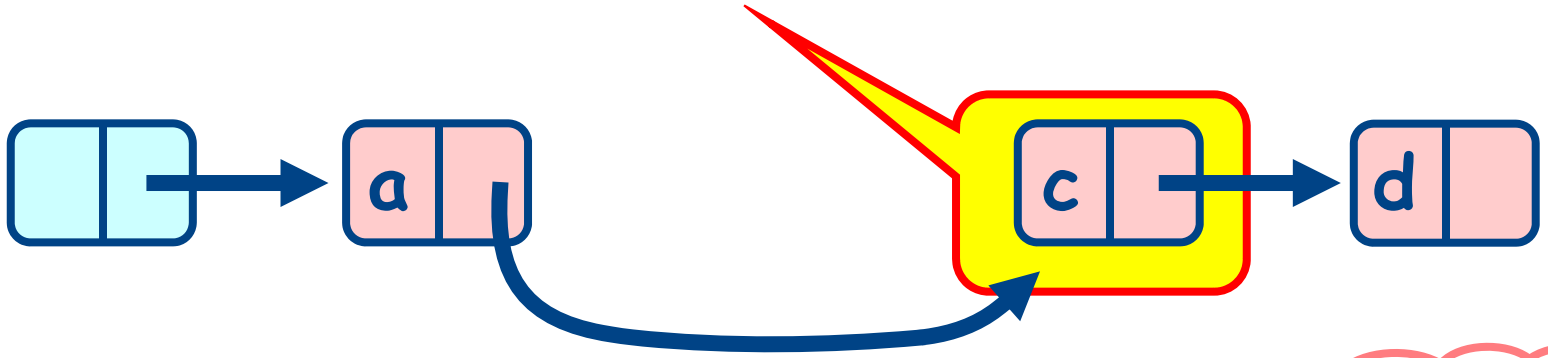


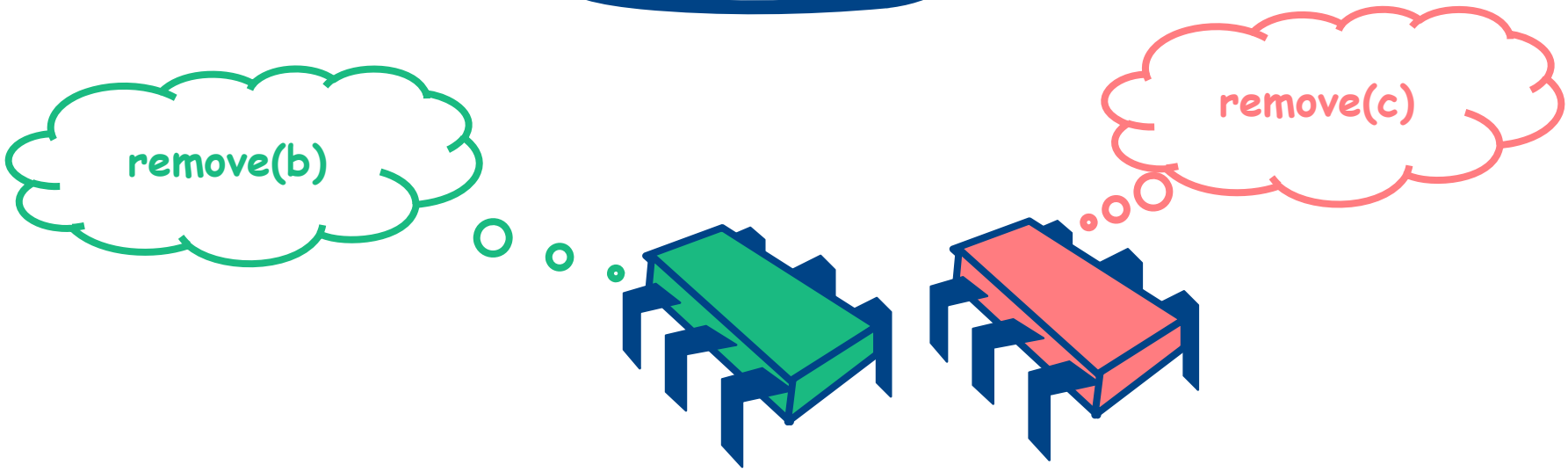
Uh, Oh



Uh, Oh

Bad news, C not removed

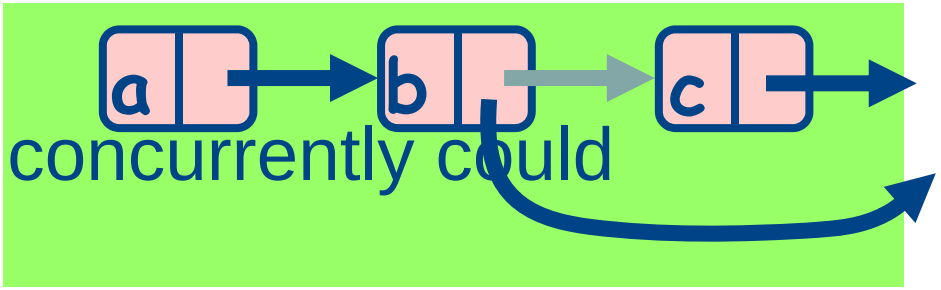


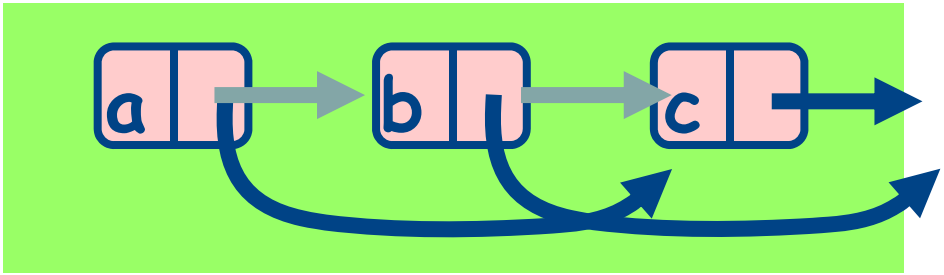


Problem

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

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

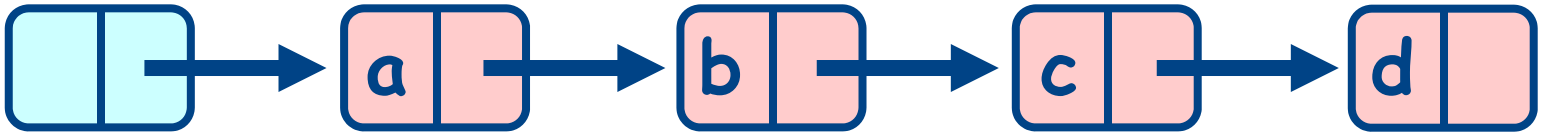




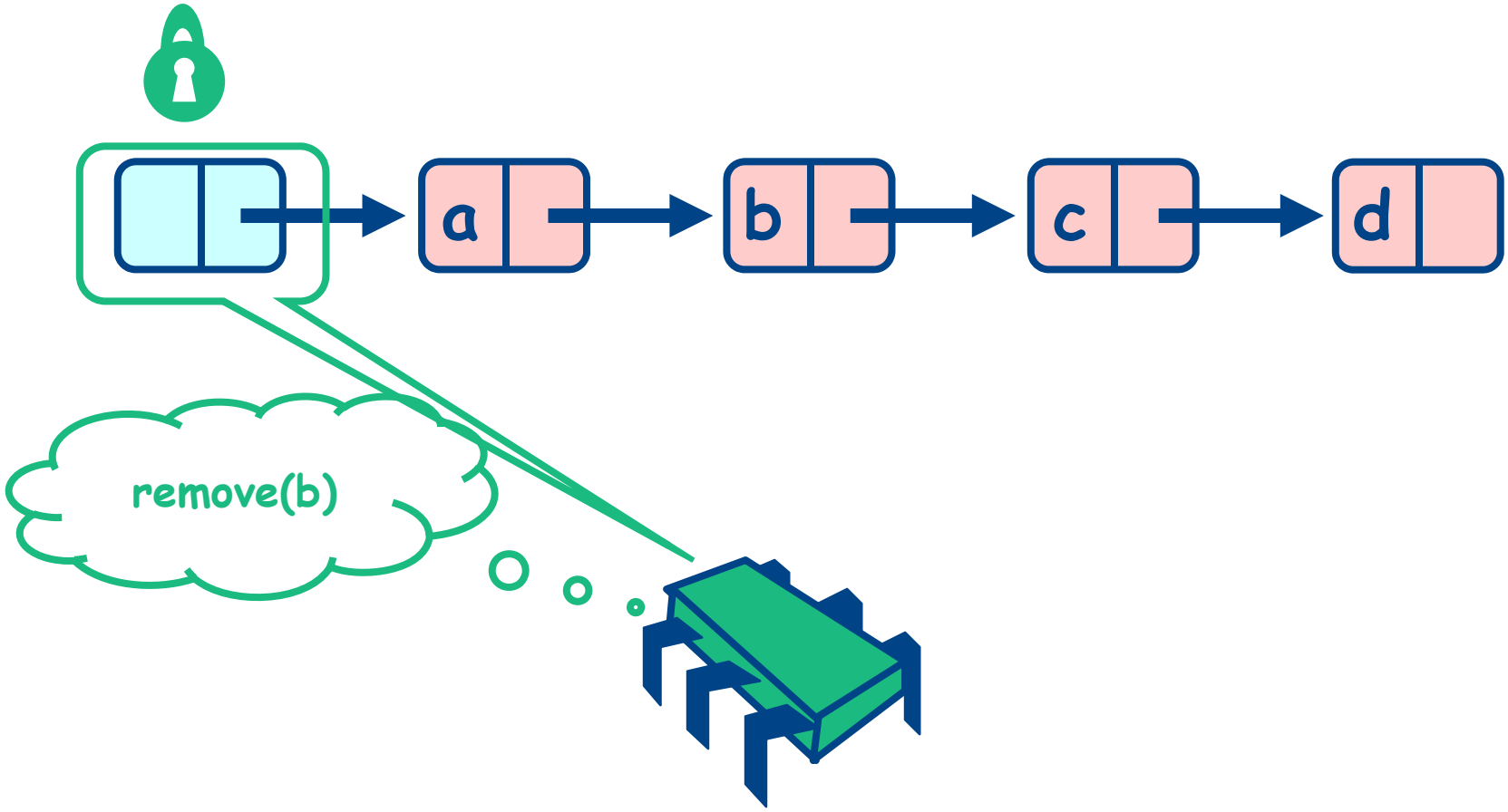
Insight

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

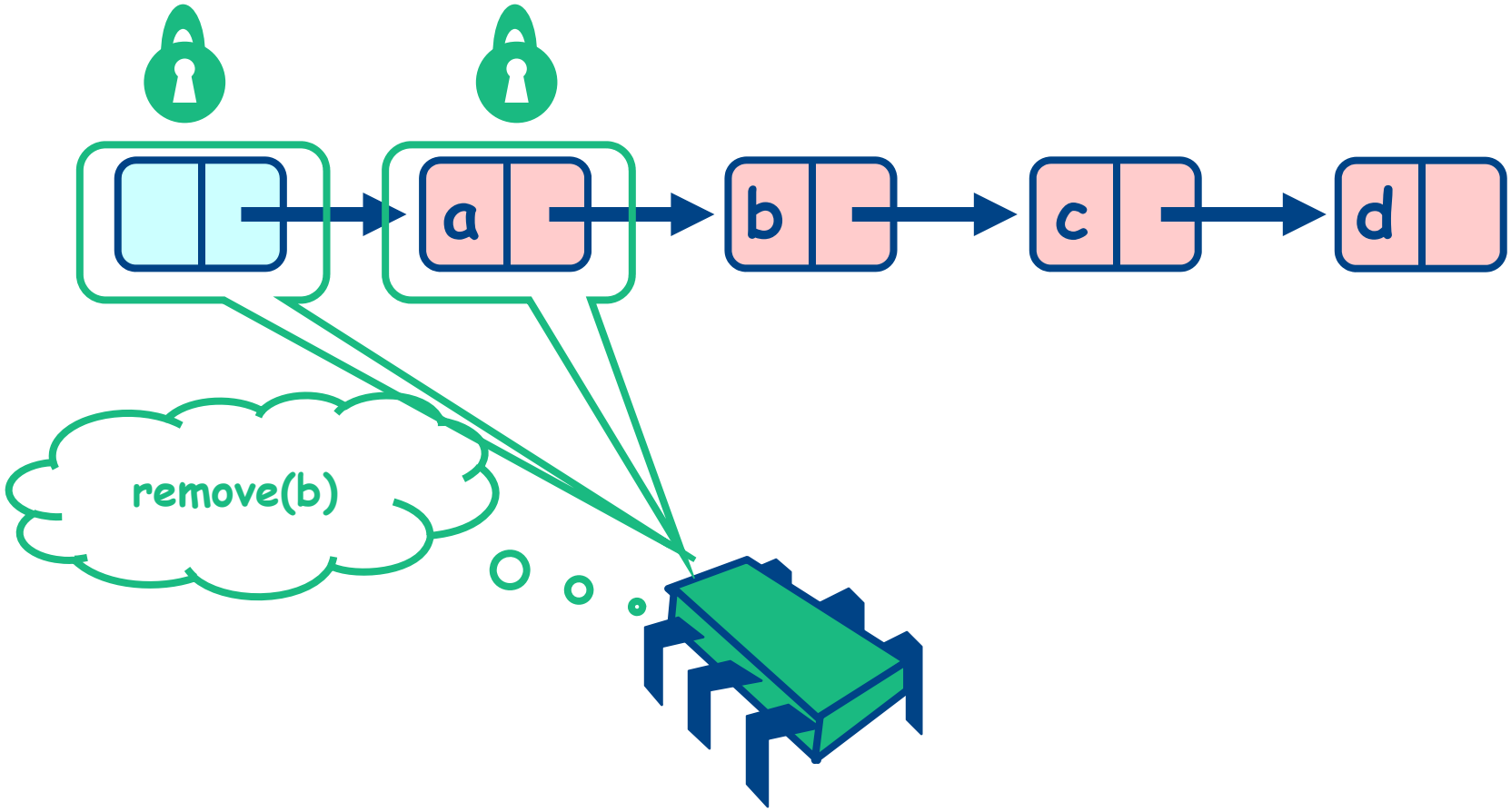
Hand-Over-Hand Again



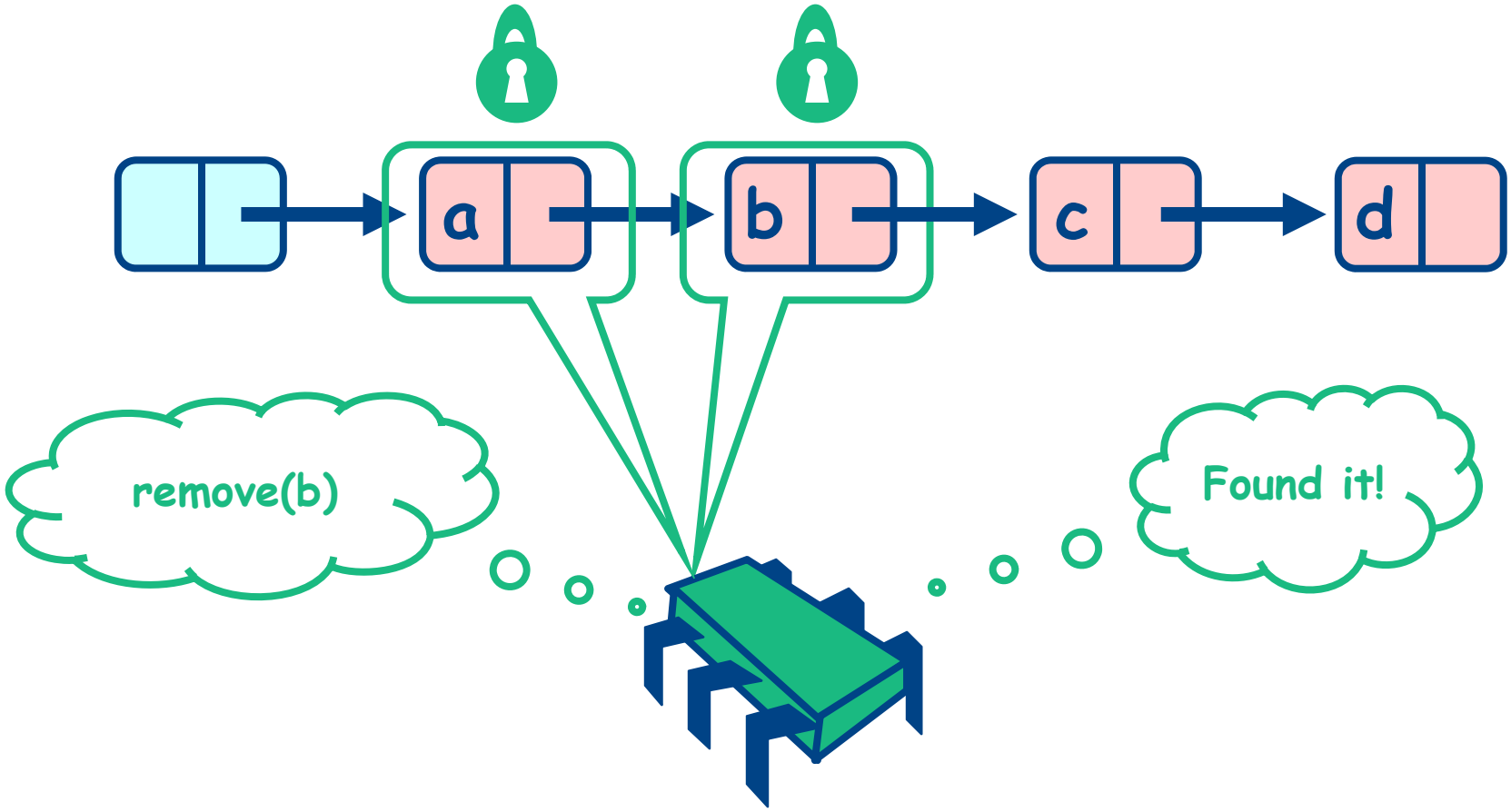
Hand-Over-Hand Again



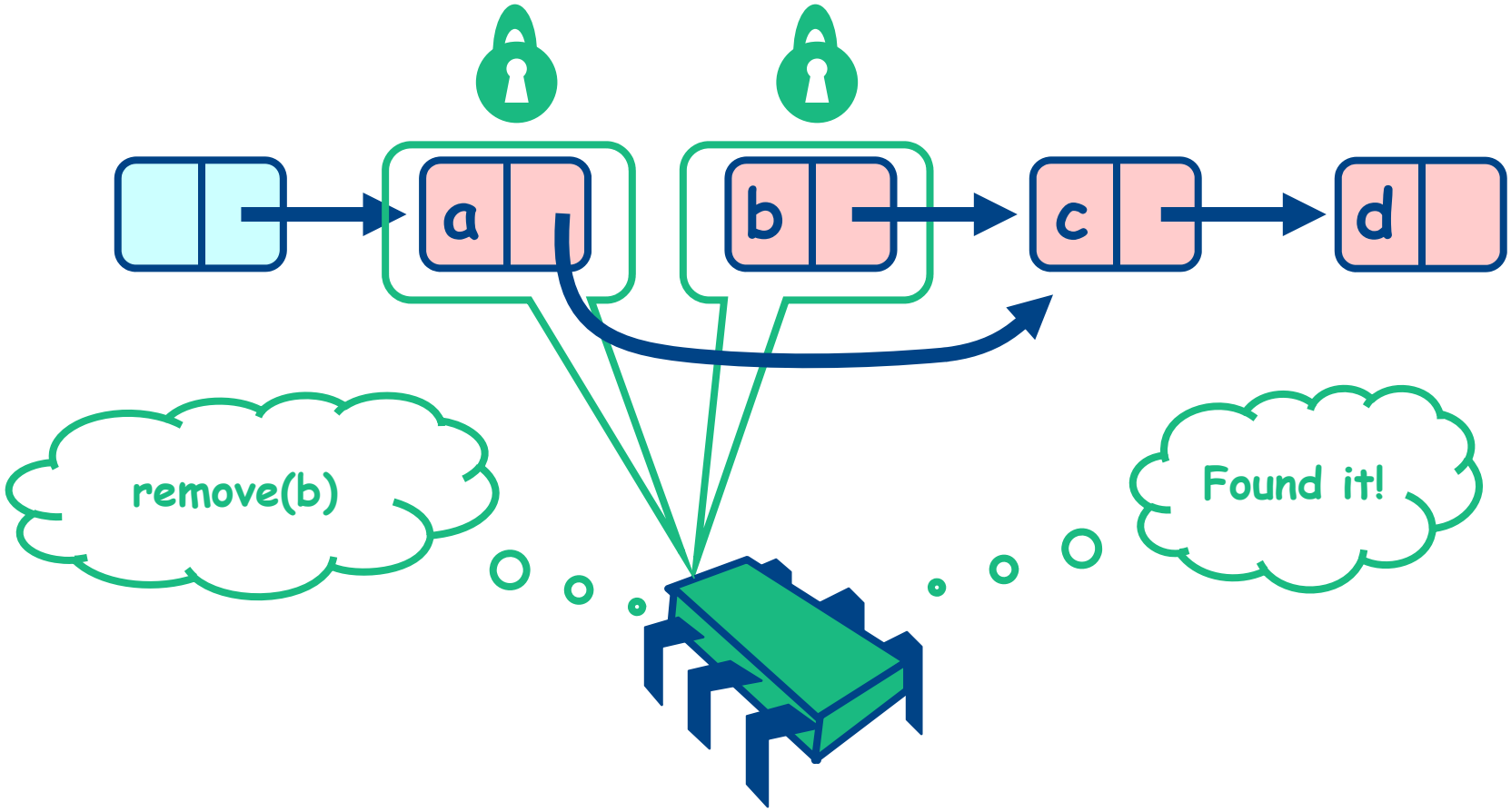
Hand-Over-Hand Again



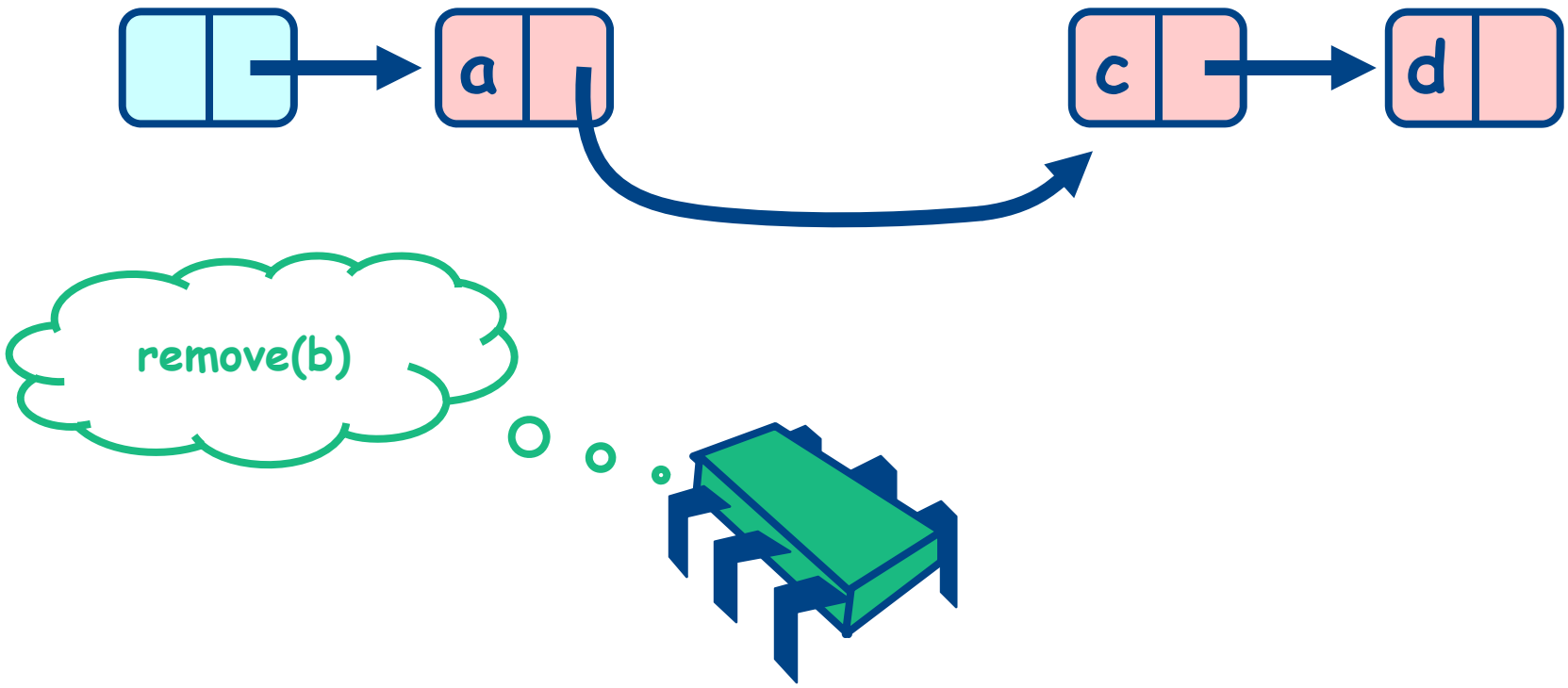
Hand-Over-Hand Again



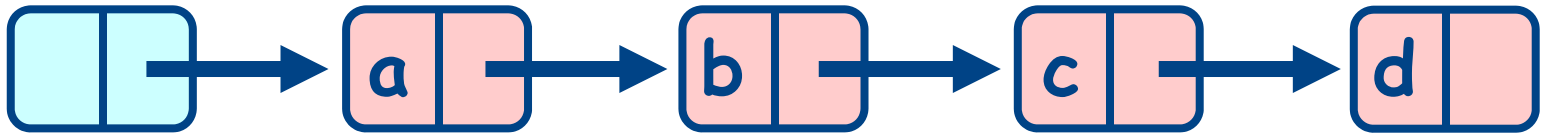
Hand-Over-Hand Again

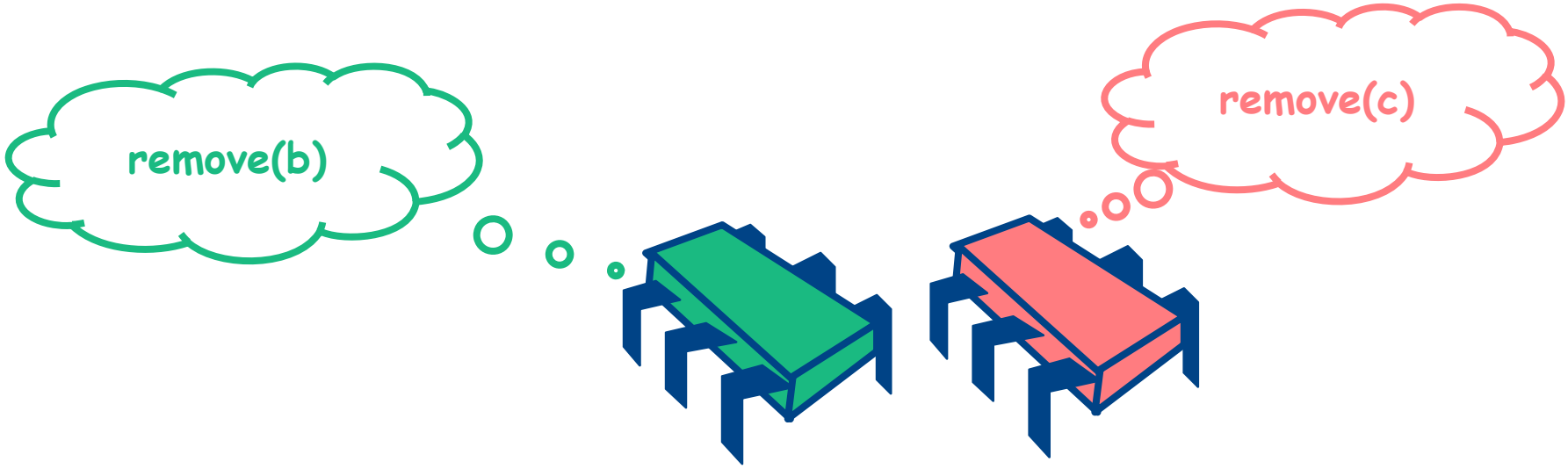


Hand-Over-Hand Again

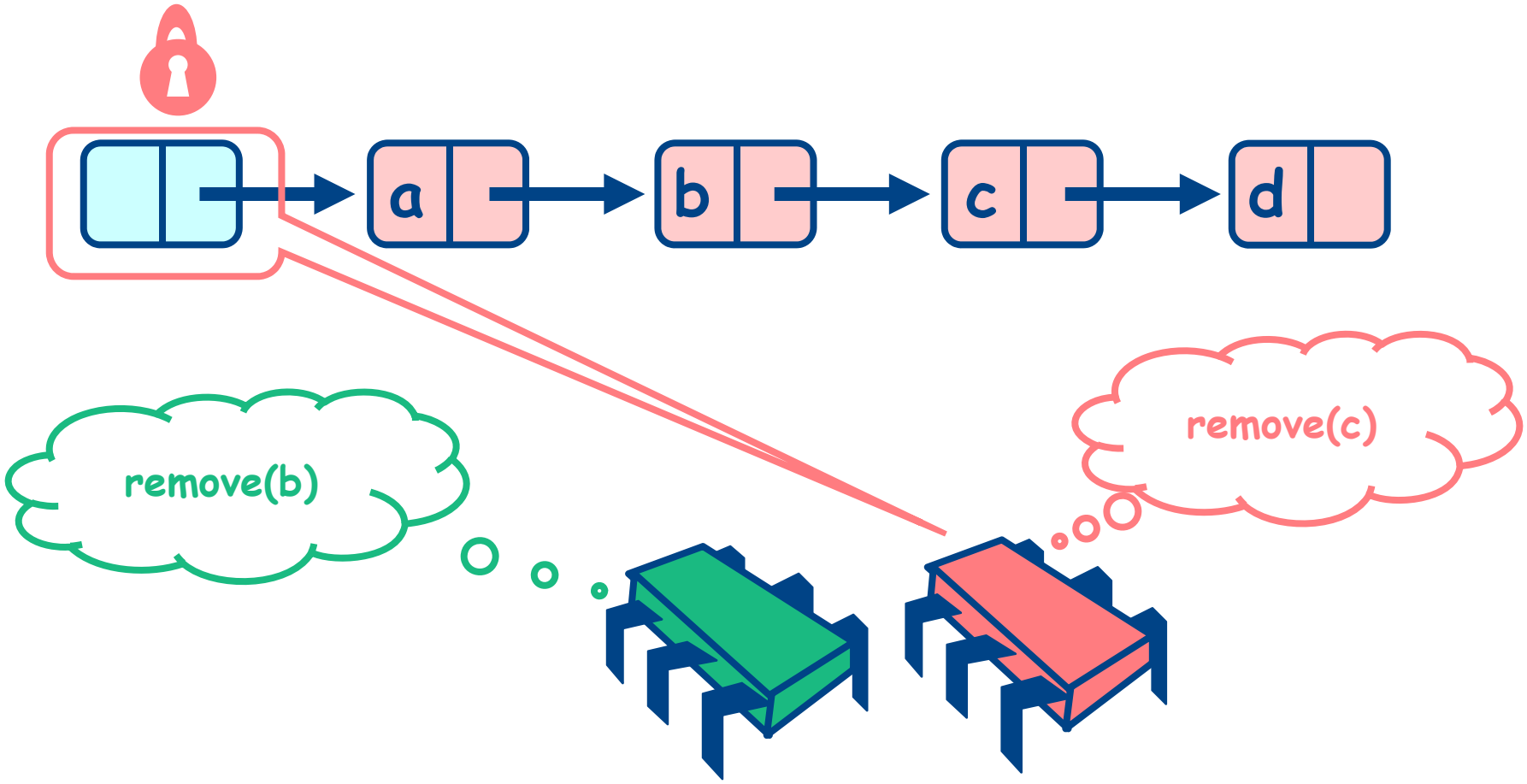


Removing a Node

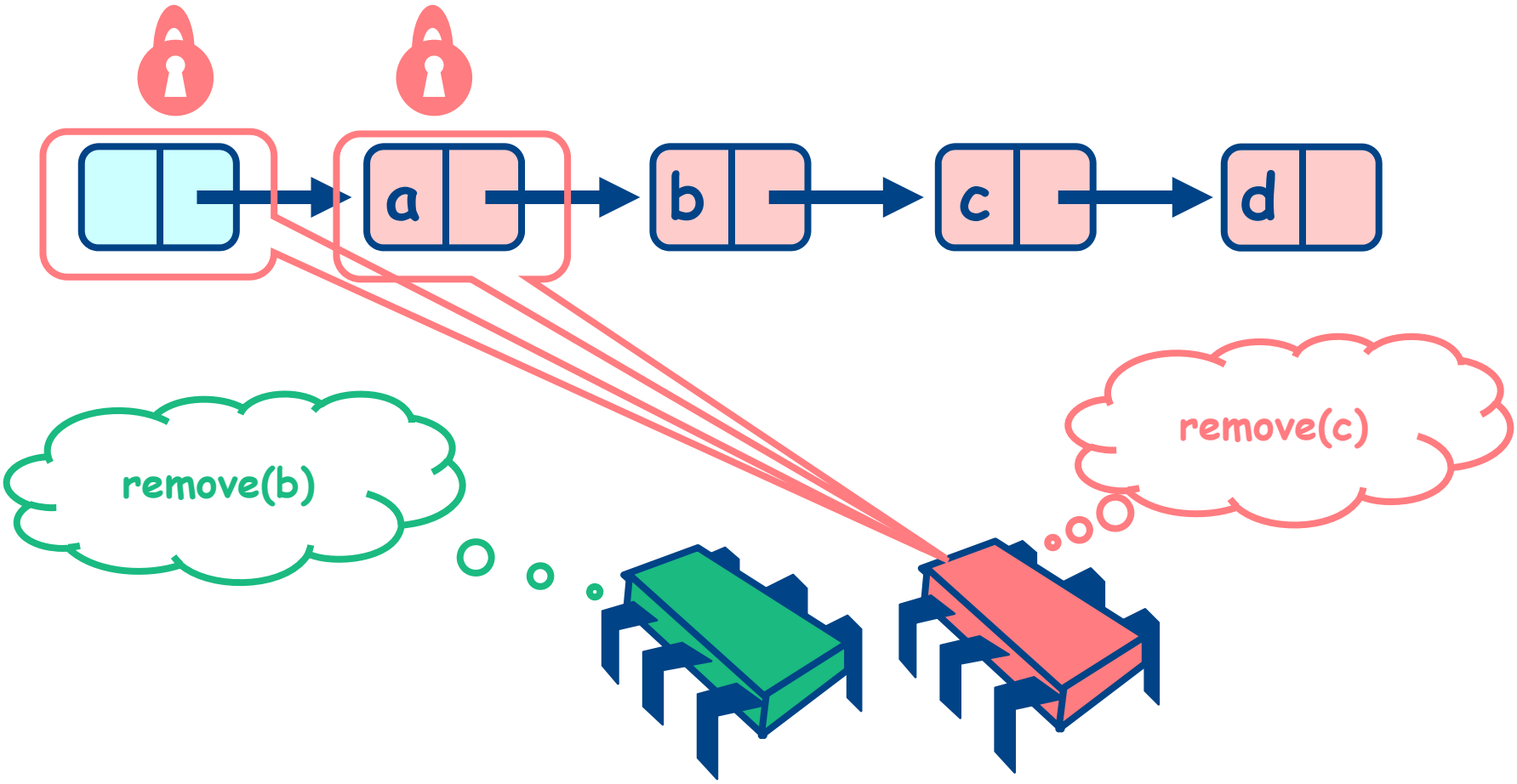




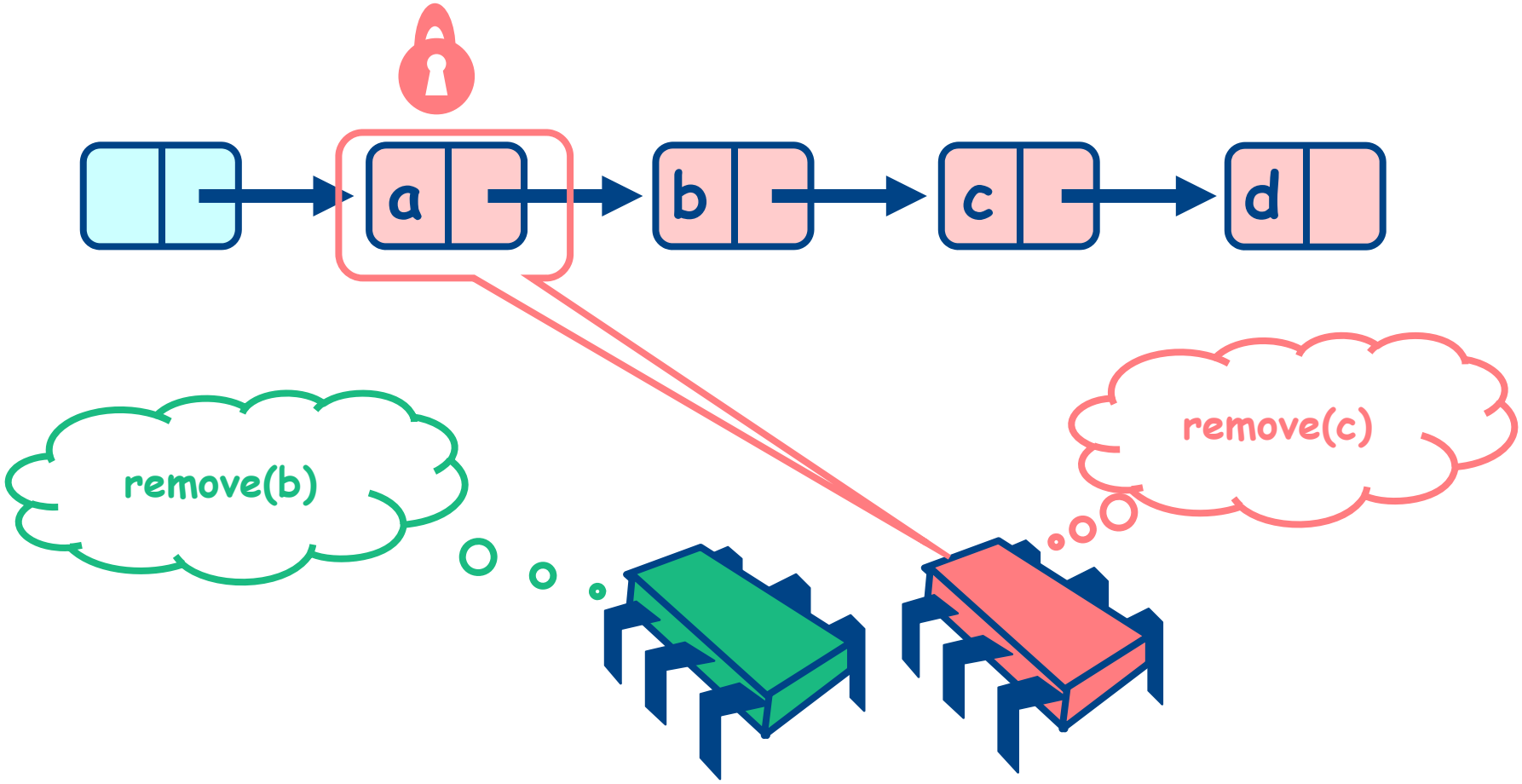
Removing a Node



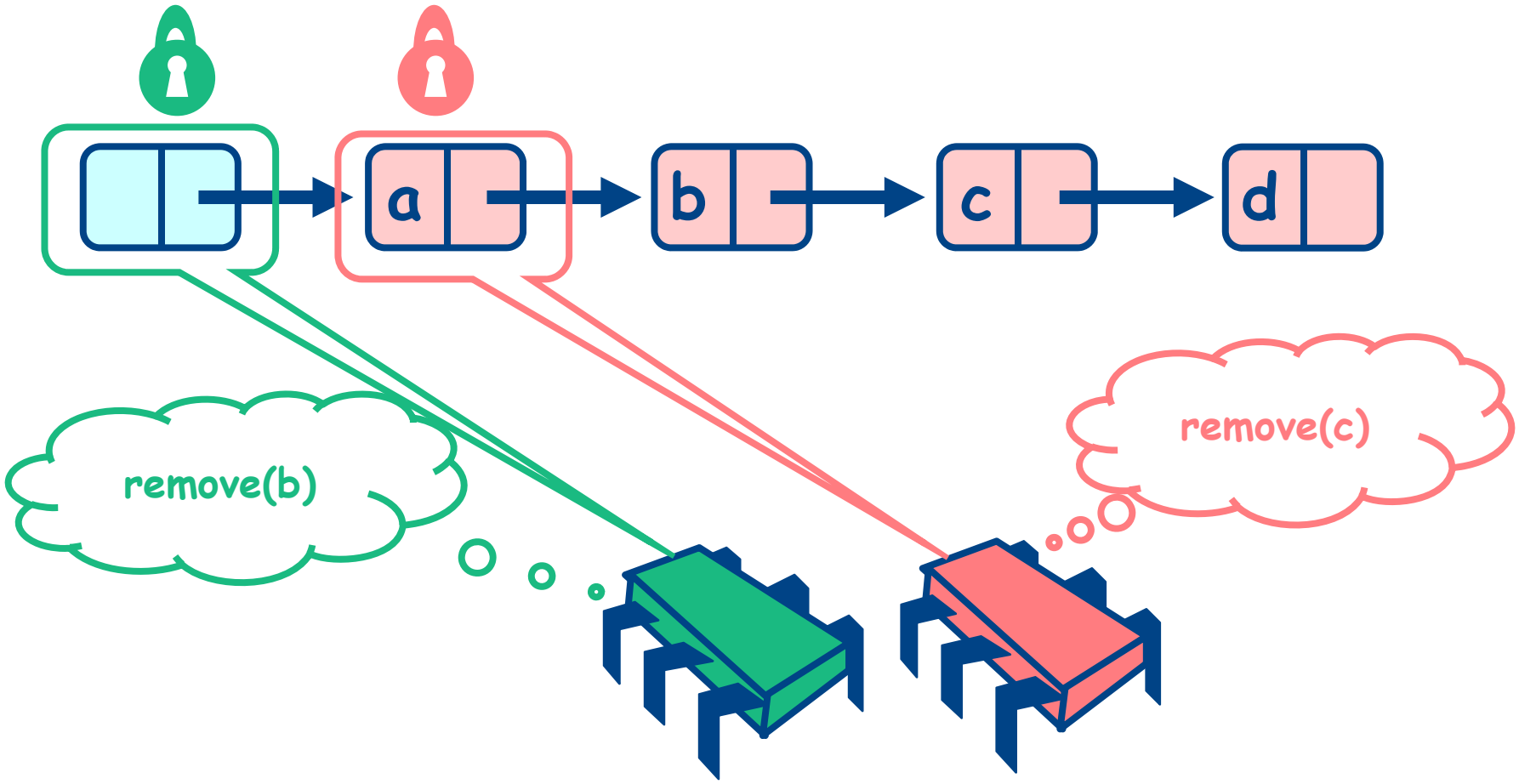
Removing a Node



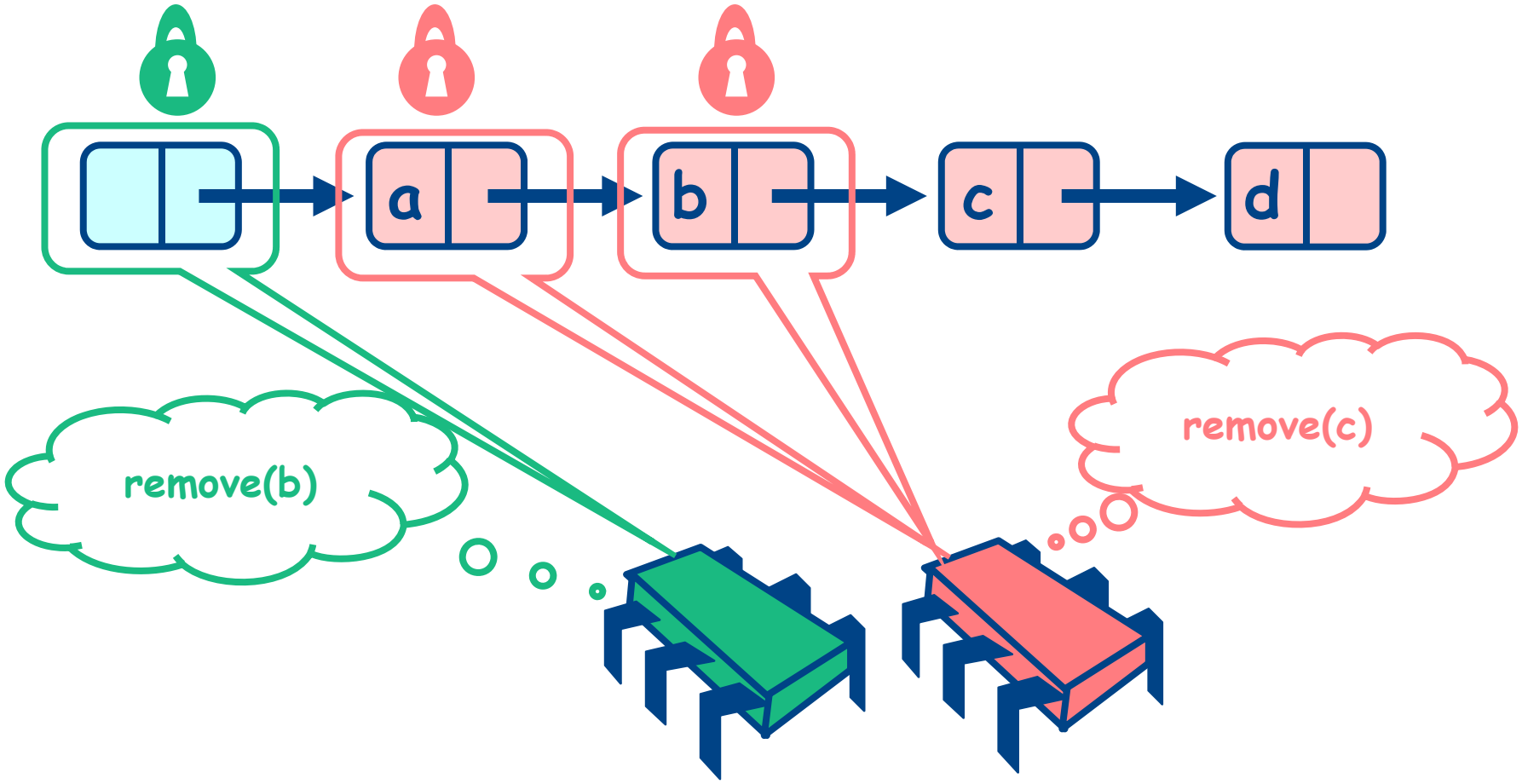
Removing a Node



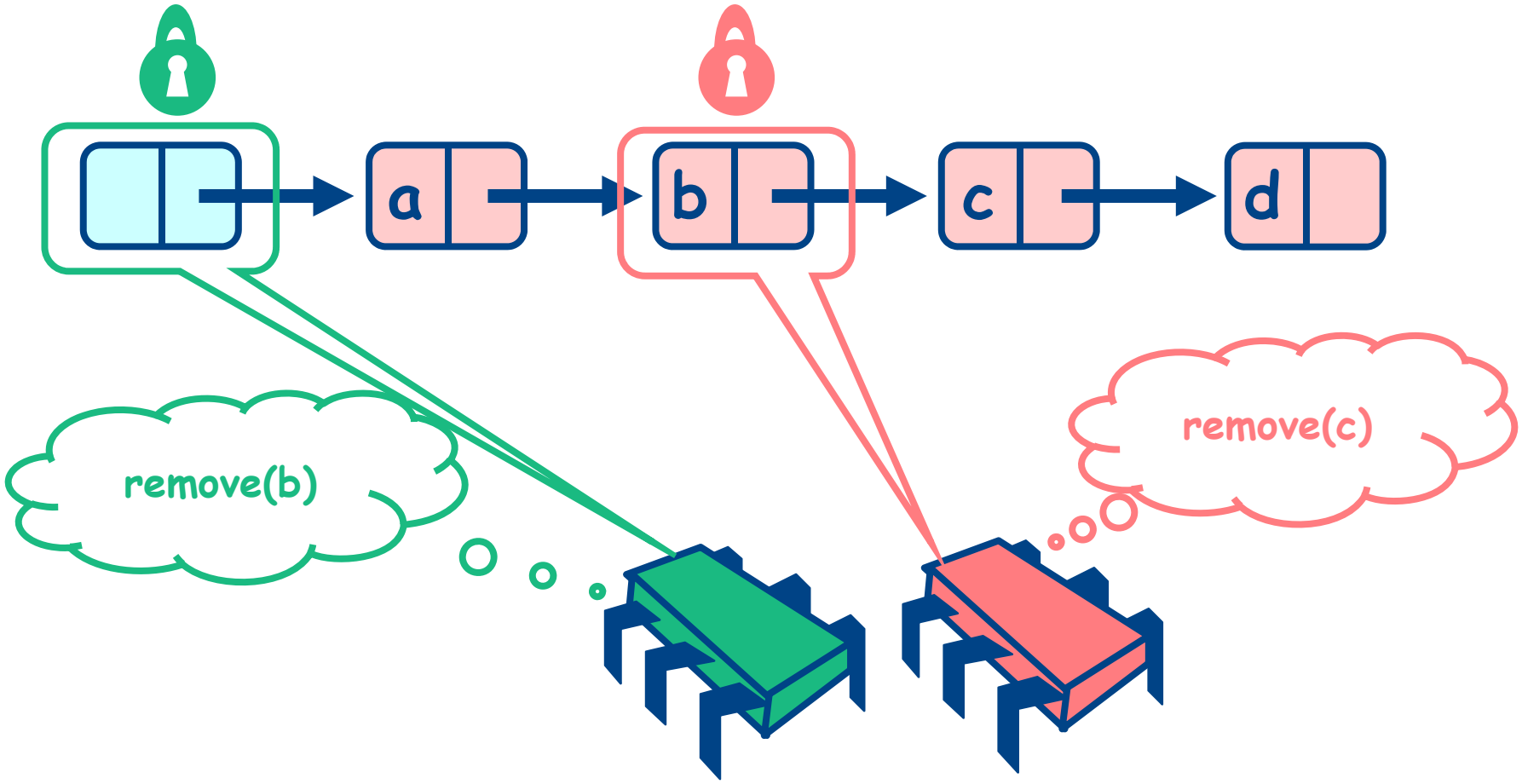
Removing a Node



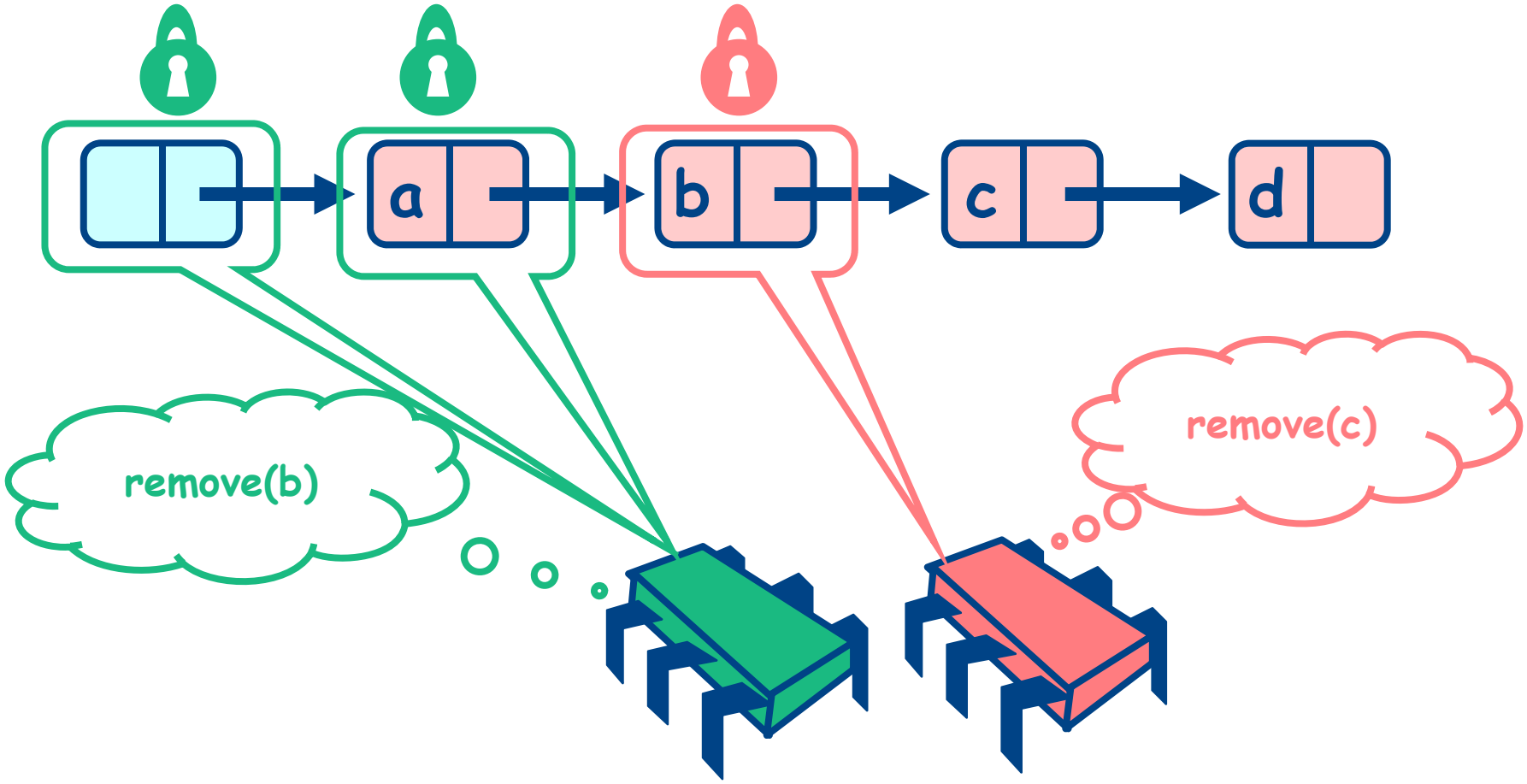
Removing a Node



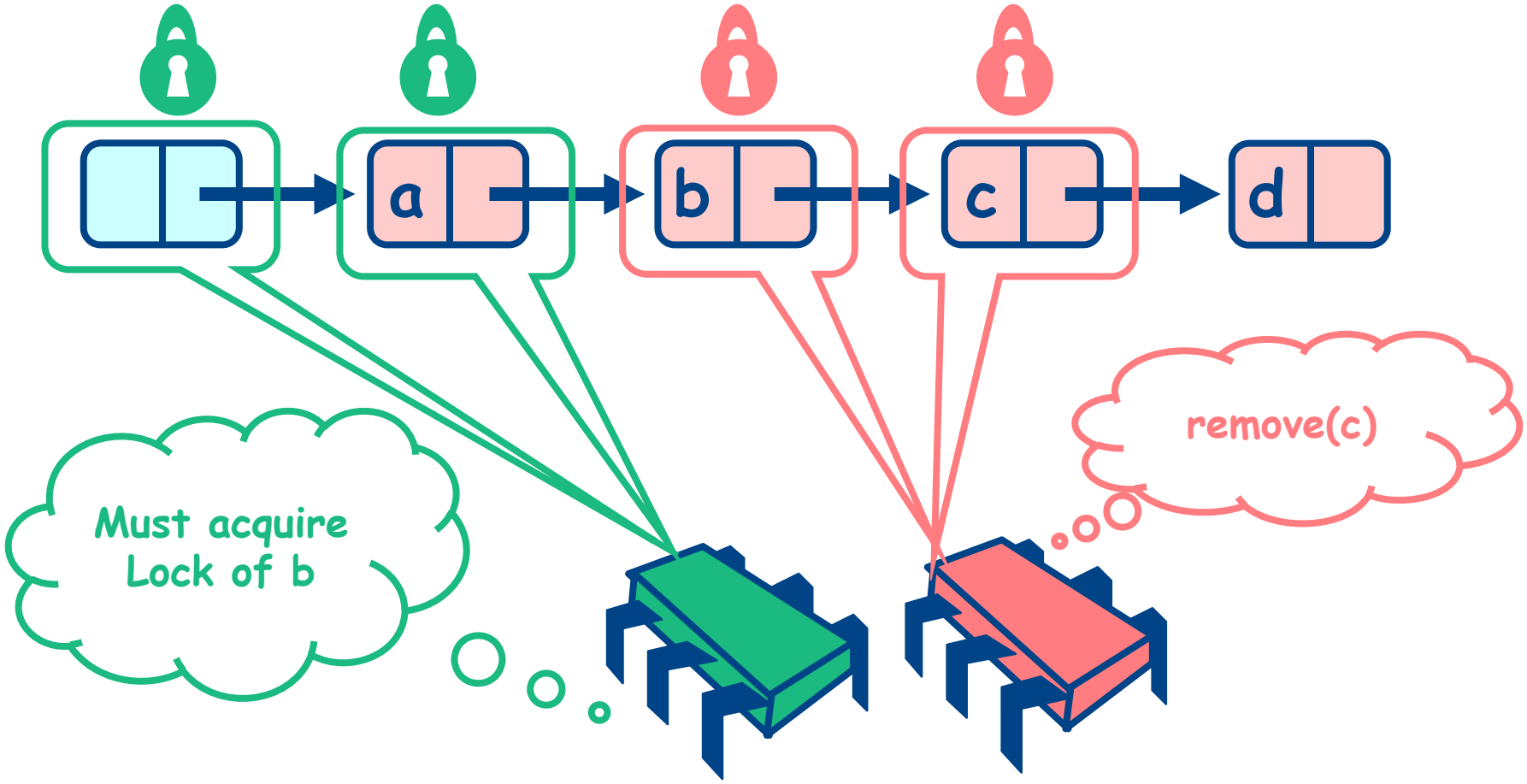
Removing a Node



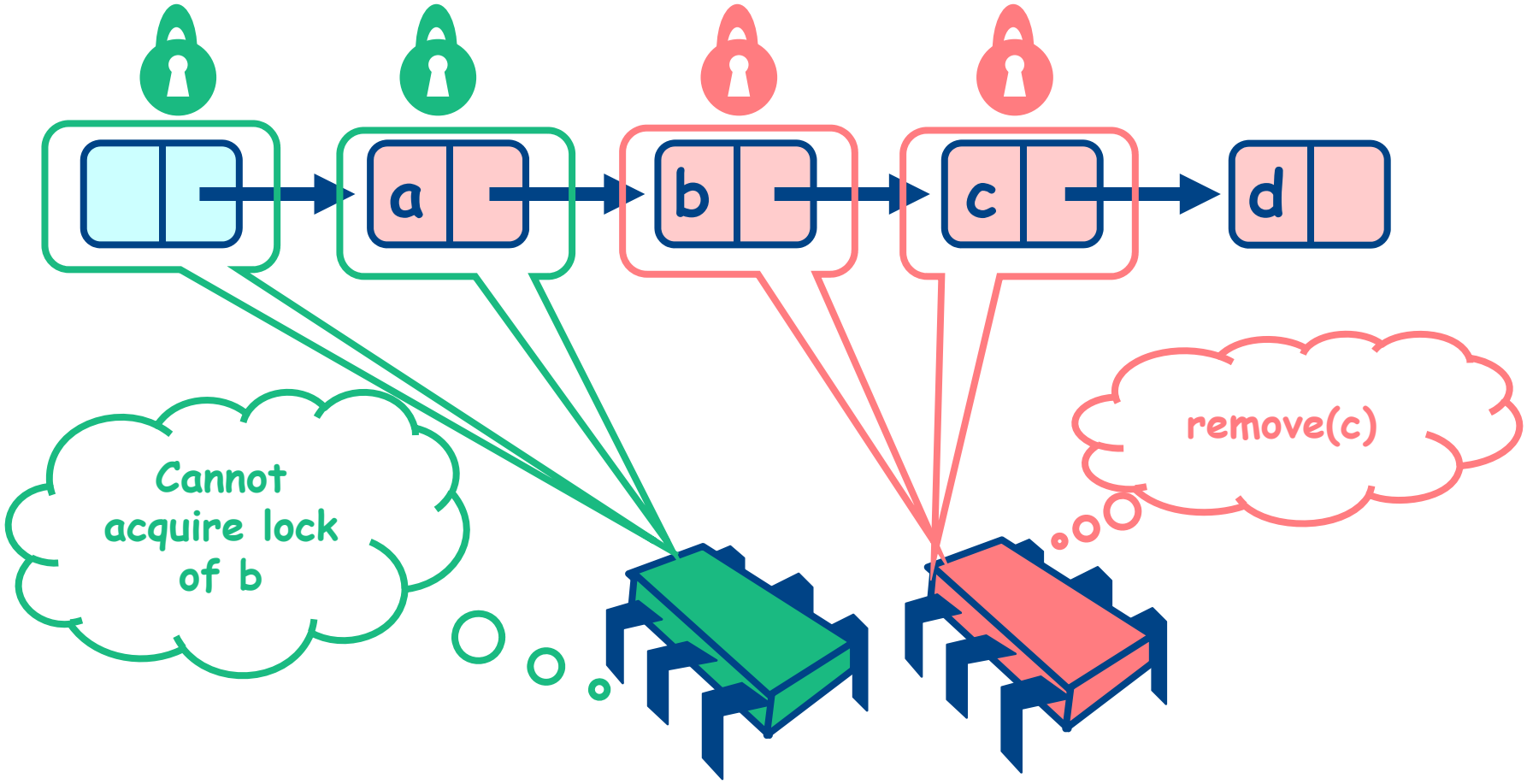
Removing a Node



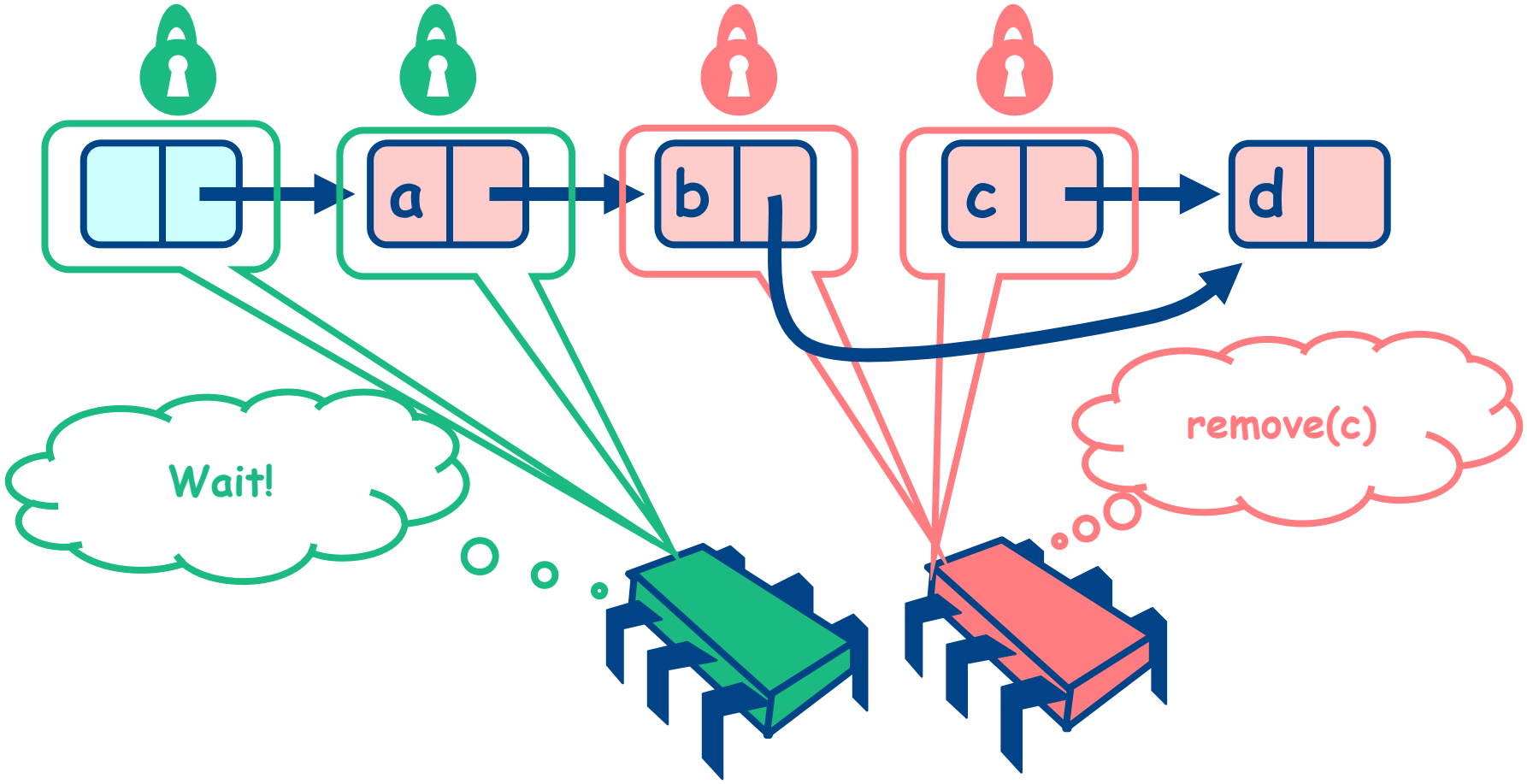
Removing a Node



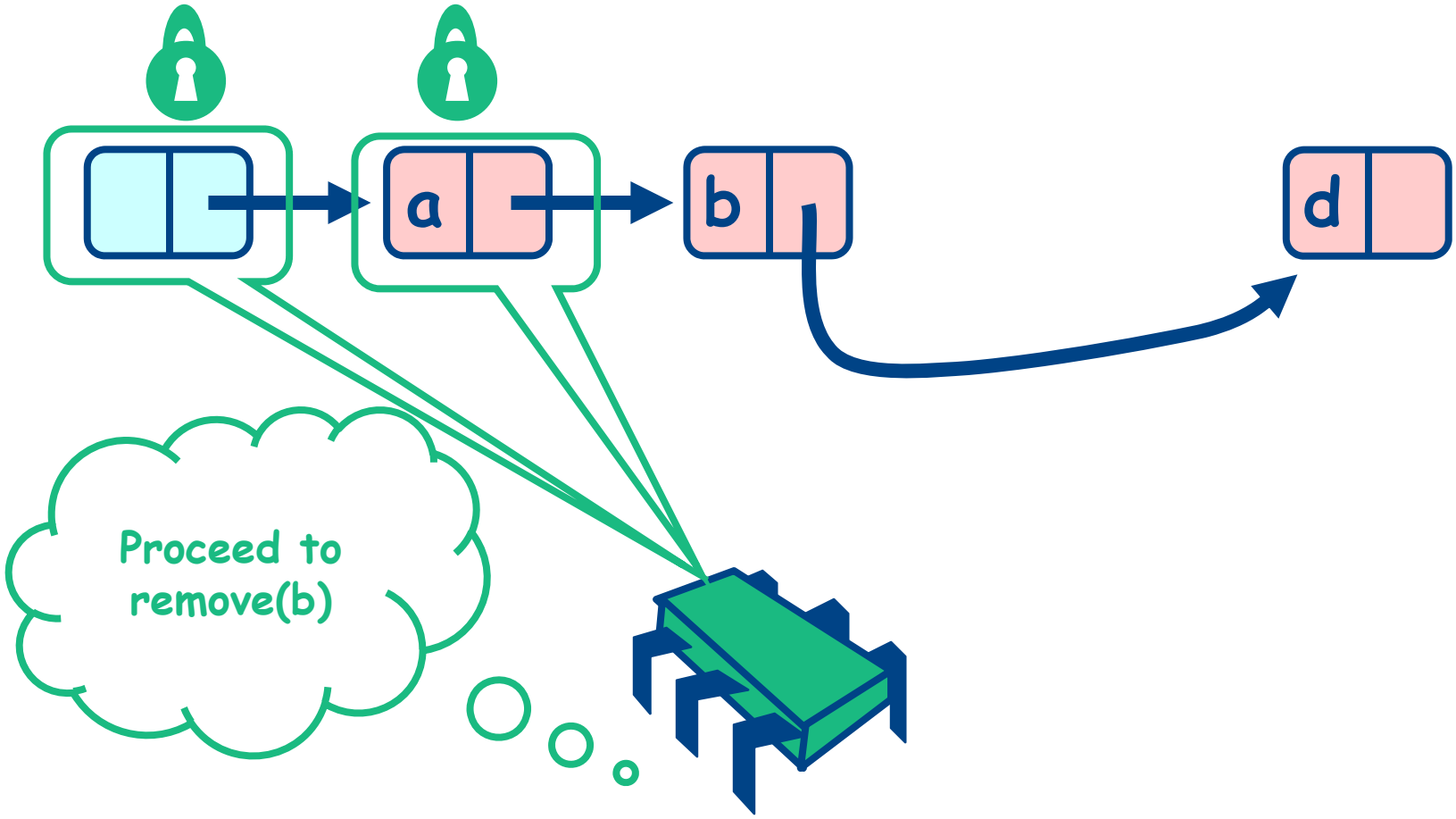
Removing a Node



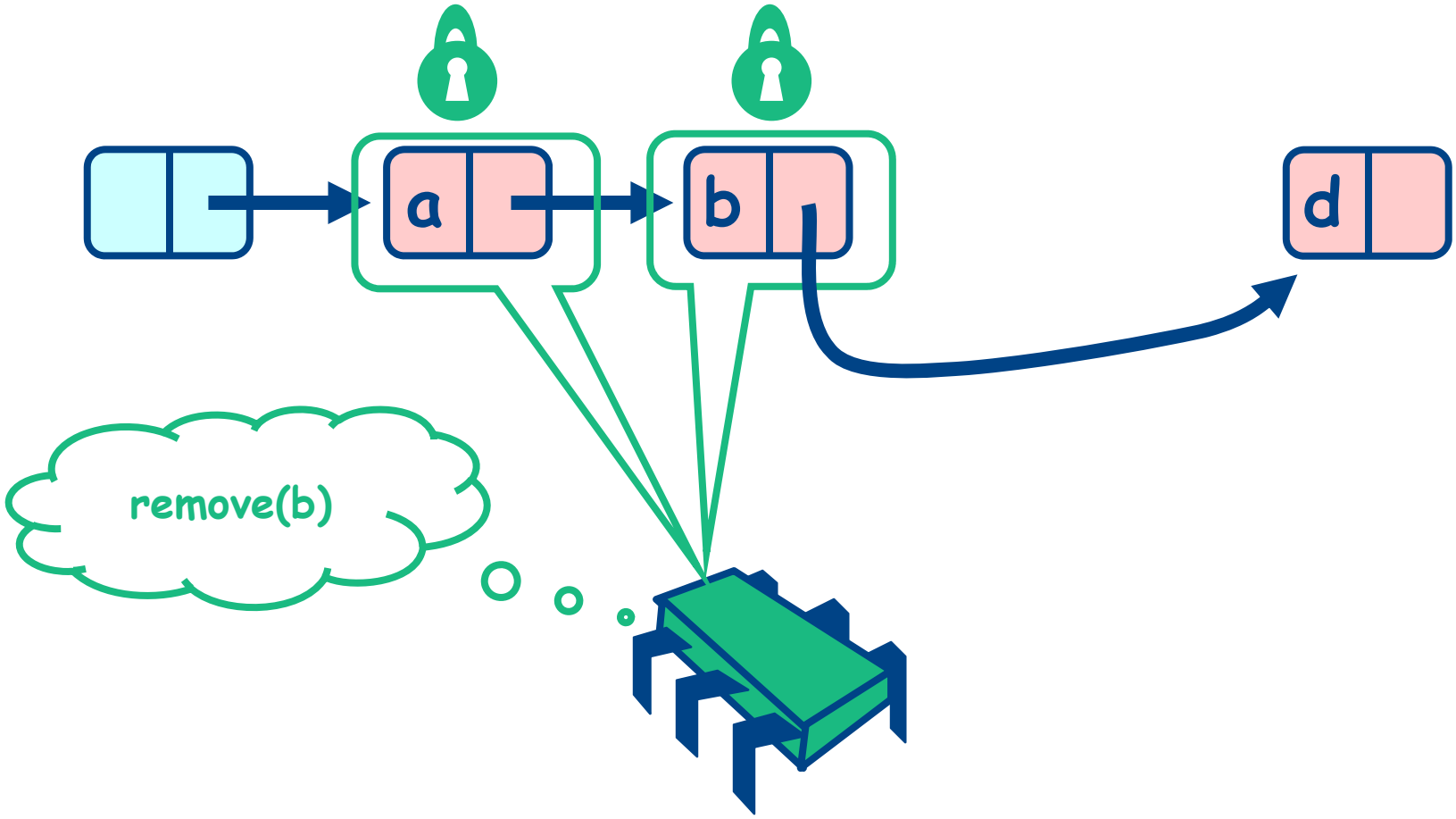
Removing a Node



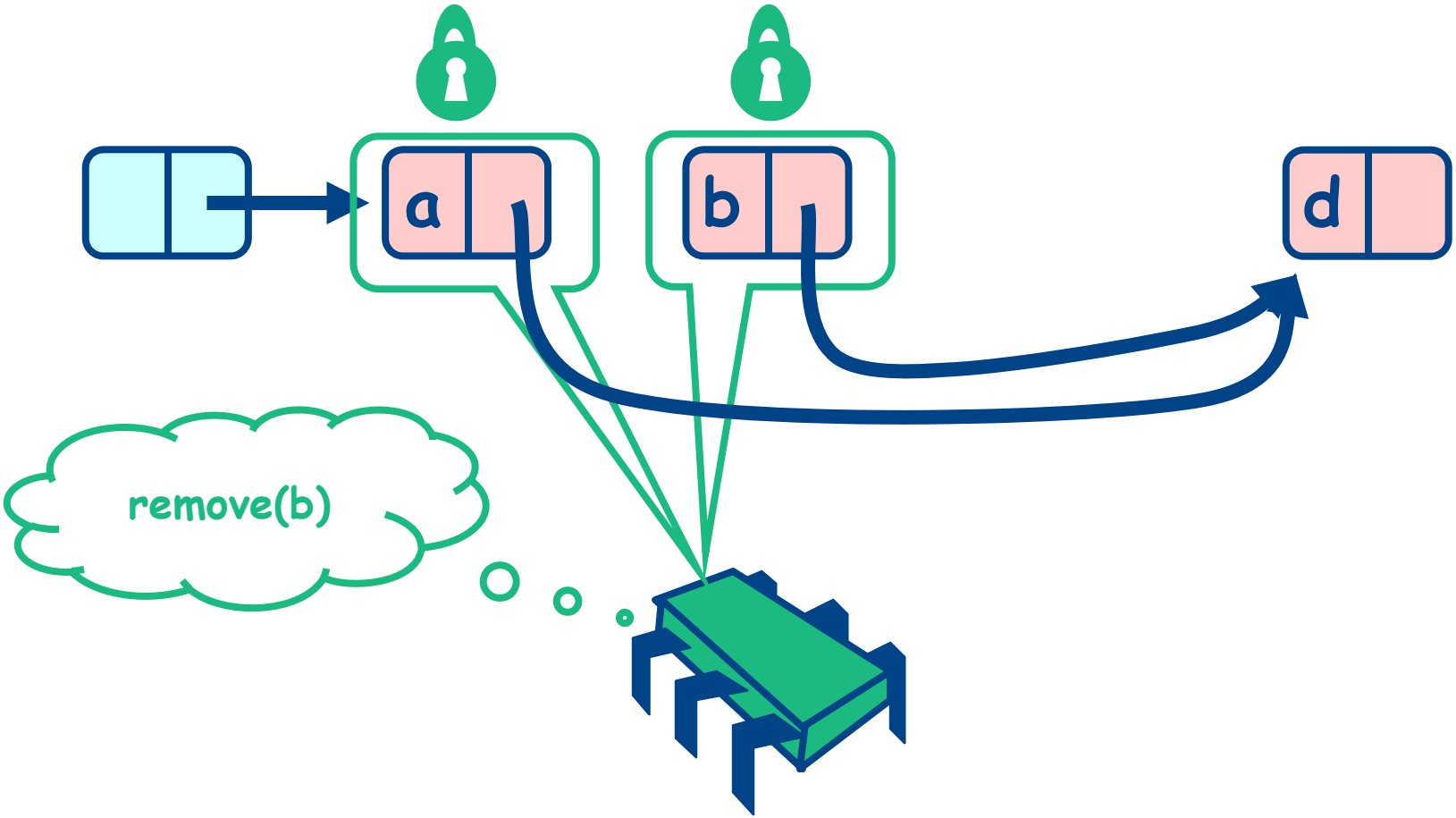
Removing a Node



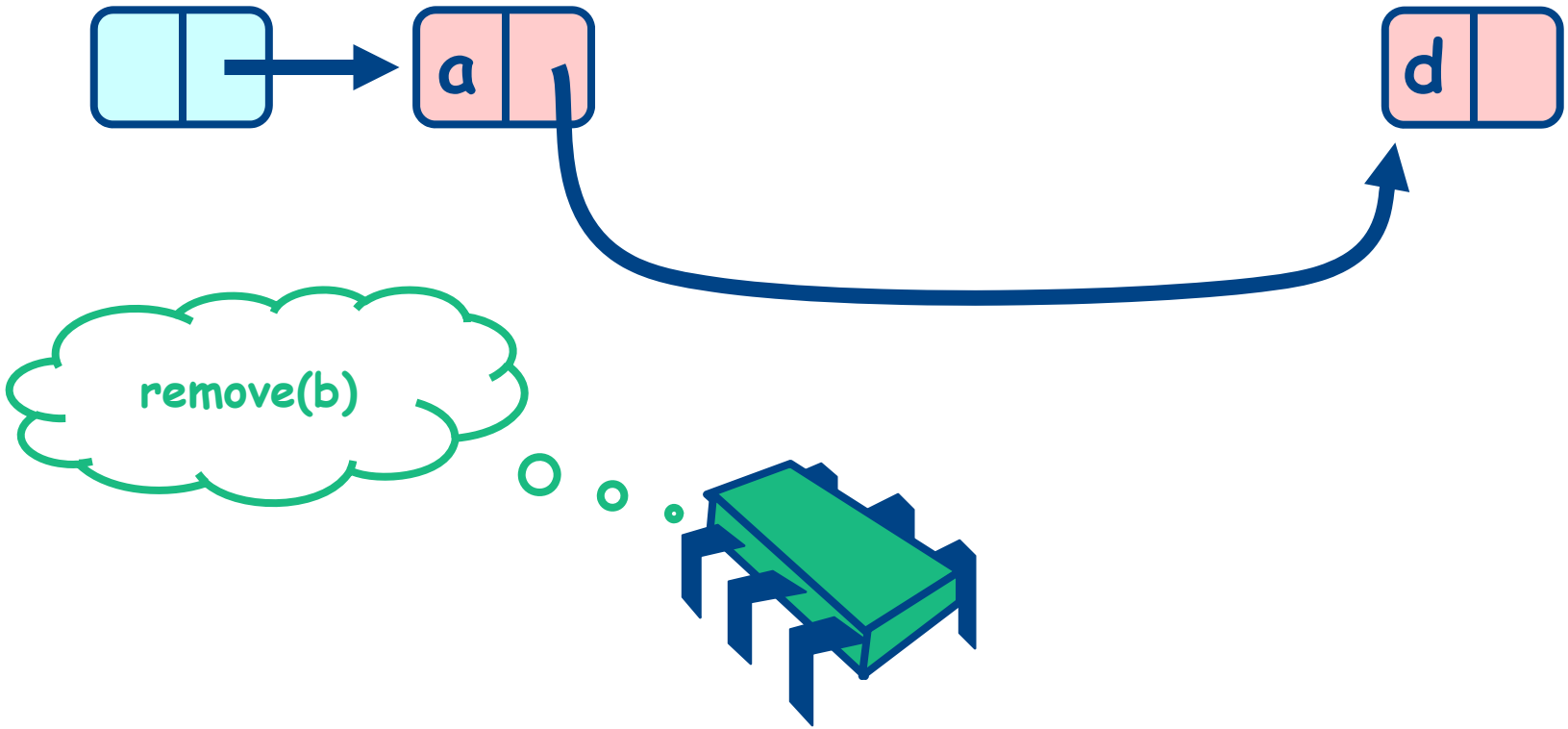
Removing a Node



Removing a Node



Removing a Node



Removing a Node

add

```
public boolean add(T item) {
   int key = item.hashCode();
   head.lock();
   Node pred = head;
   try {
      Node curr = pred.next;
      curr.lock();
      try {
         while (curr.key < key) {
            pred.unlock();
            pred = curr;
            curr = curr.next;
            curr.lock();
         }
         if (curr.key == key) {
            return false;
         }
         Node newNode = new Node(item);
         newNode.next = curr;
         pred.next = newNode;
         return true;
      } finally {
         curr.unlock();
      }
   } finally {
      pred.unlock();
   }
}
```

remove

```
public boolean remove(T item) {
  Node pred = null, curr = null;
  int key = item.hashCode();
  head.lock();
  try {
    pred = head;
    curr = pred.next;
    curr.lock();
    try {
      while (curr.key < key) {
        pred.unlock();
        pred = curr;
        curr = curr.next;
        curr.lock();
      }
      if (curr.key == key) {
        pred.next = curr.next;
        return true;
      }
      return false;
    } finally {
      curr.unlock();
    }
  } finally {
    pred.unlock();
  }
}
```

Contains and node

```
public boolean contains(T item) {
  Node last = null, pred = null, curr = null;
  int key = item.hashCode();
  head.lock();
  try {
    pred = head;
    curr = pred.next;
    curr.lock();
    try {
      while (curr.key < key) {
        pred.unlock();
        pred = curr;
        curr = curr.next;
        curr.lock();
      }
      return (curr.key == key);
    } finally {
      curr.unlock();
    }
  } finally {
    pred.unlock();
  }
}
```

```
private class Node {
  T item;
  int key;
  Node next;
  Lock lock;
  Node(T item) {
    this.item = item;
    this.key = item.hashCode();
    this.lock = new ReentrantLock();
  }
  Node(int key) {
    this.item = null;
    this.key = key;
    this.lock = new ReentrantLock();
  }
  void lock() {lock.lock();}
  void unlock() {lock.unlock();}
}
```

Adding Nodes

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

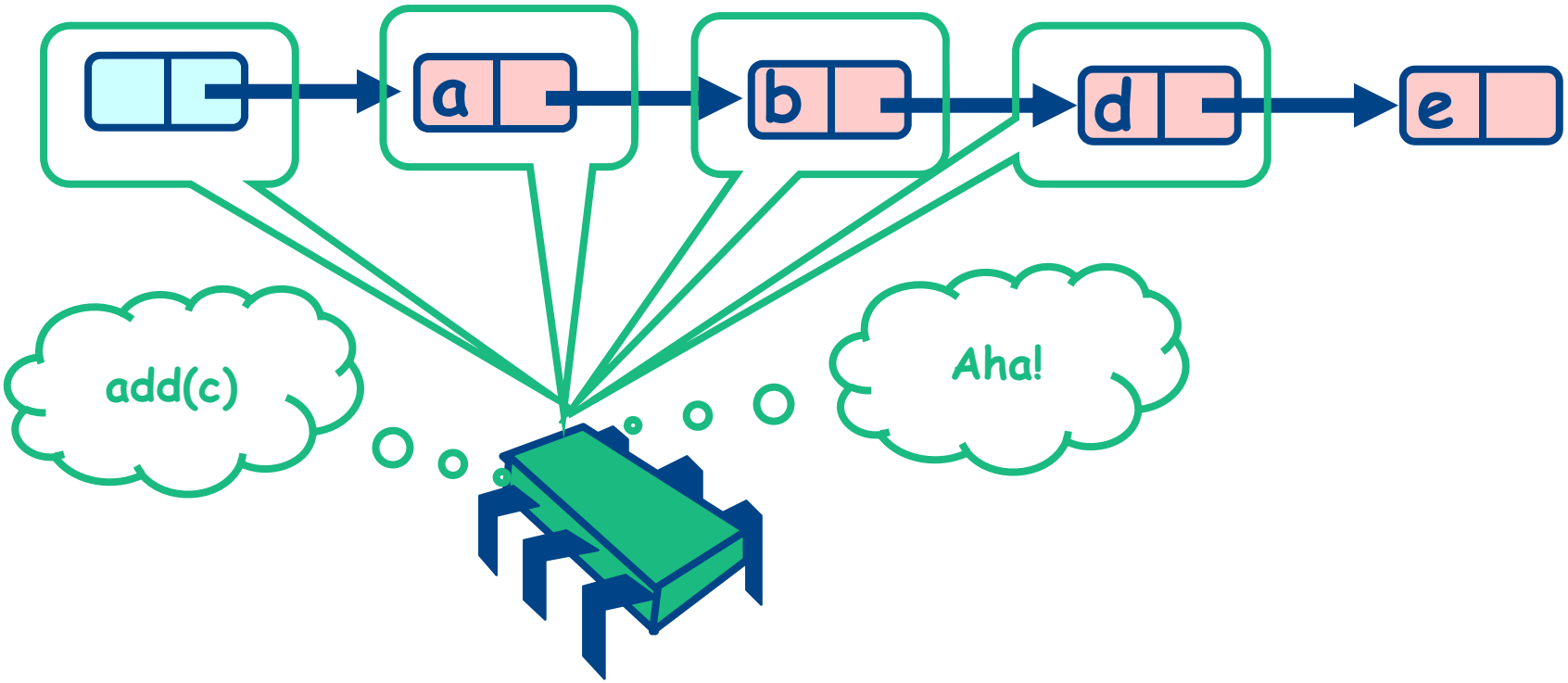
Drawbacks

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

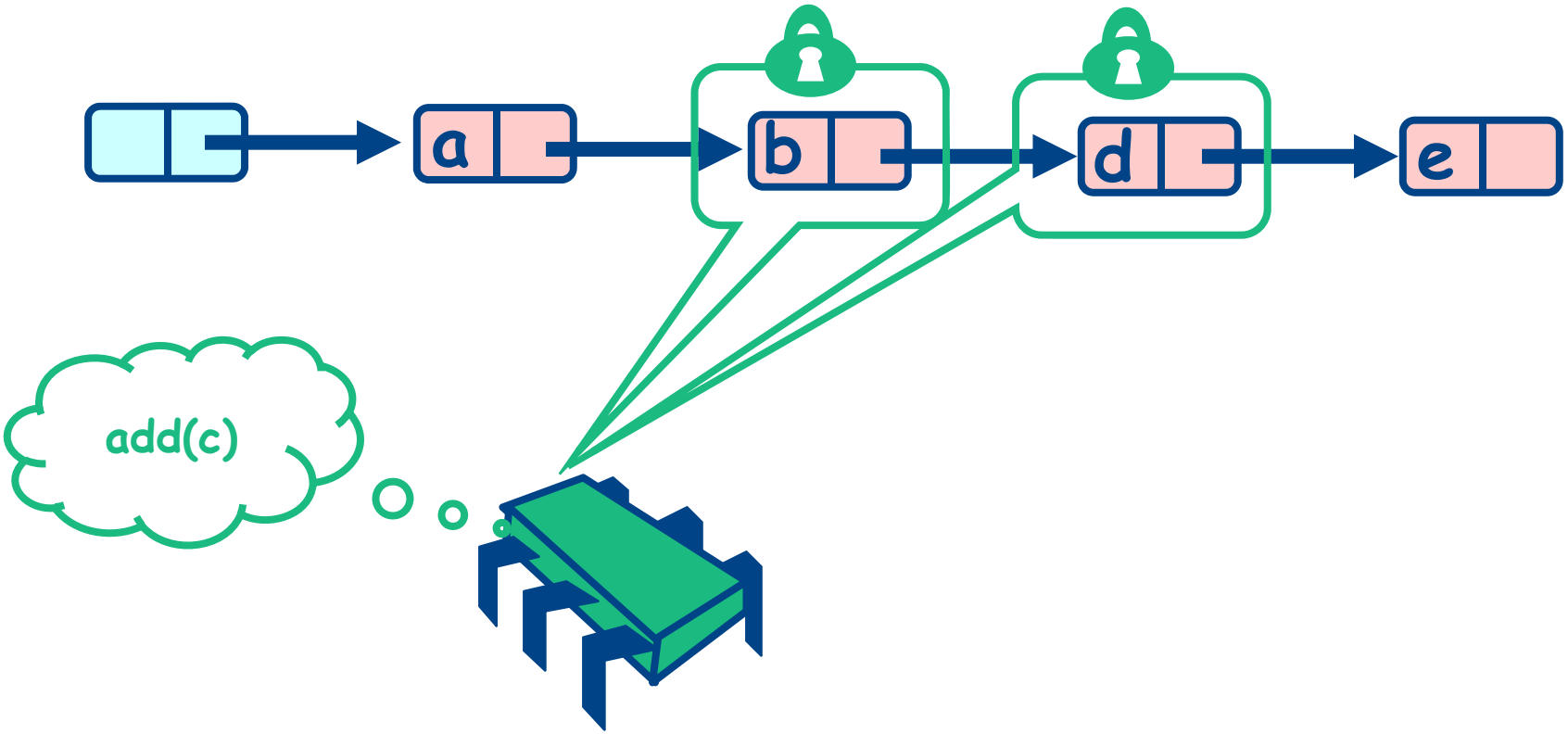
Optimistic Synchronization

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

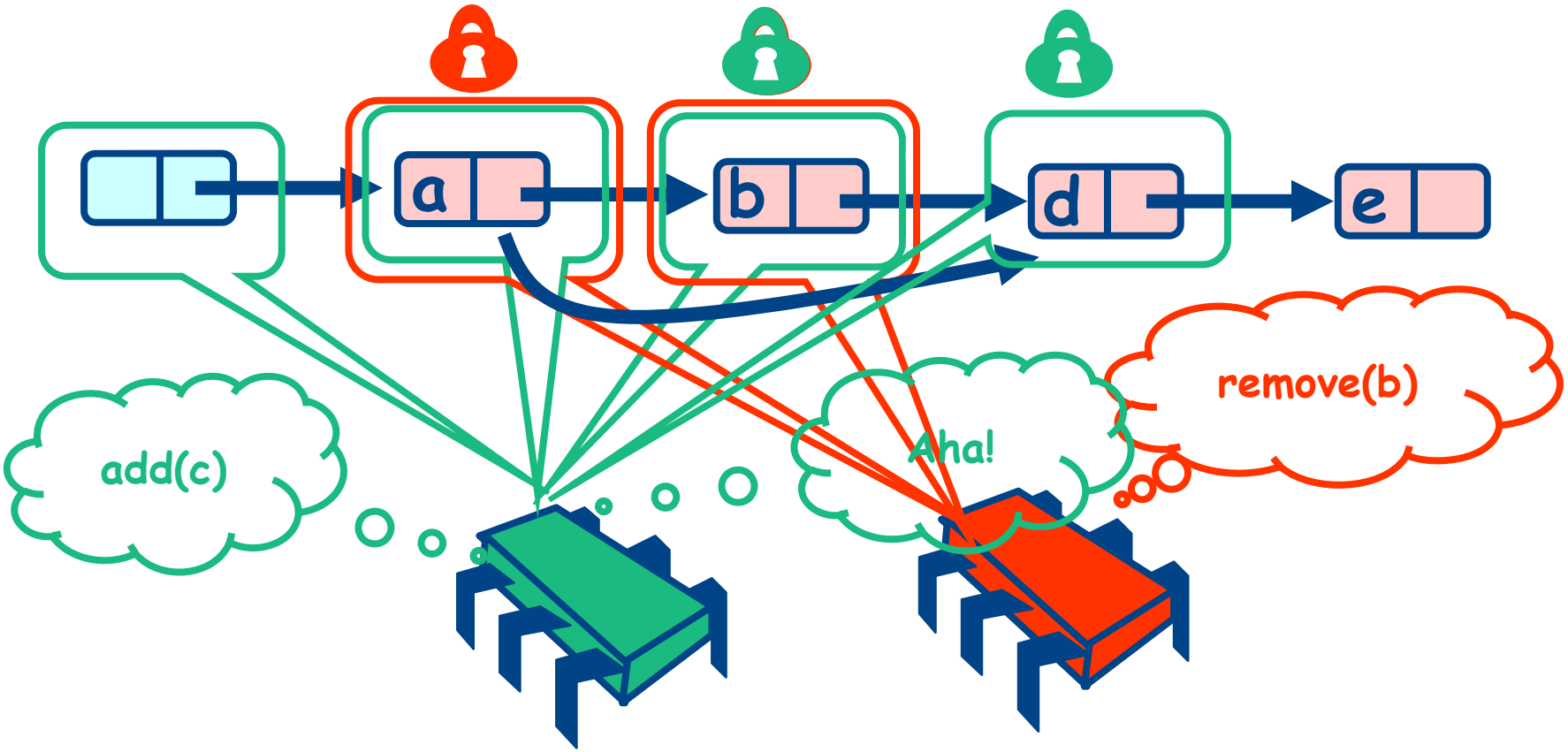
Optimistic: Traverse without



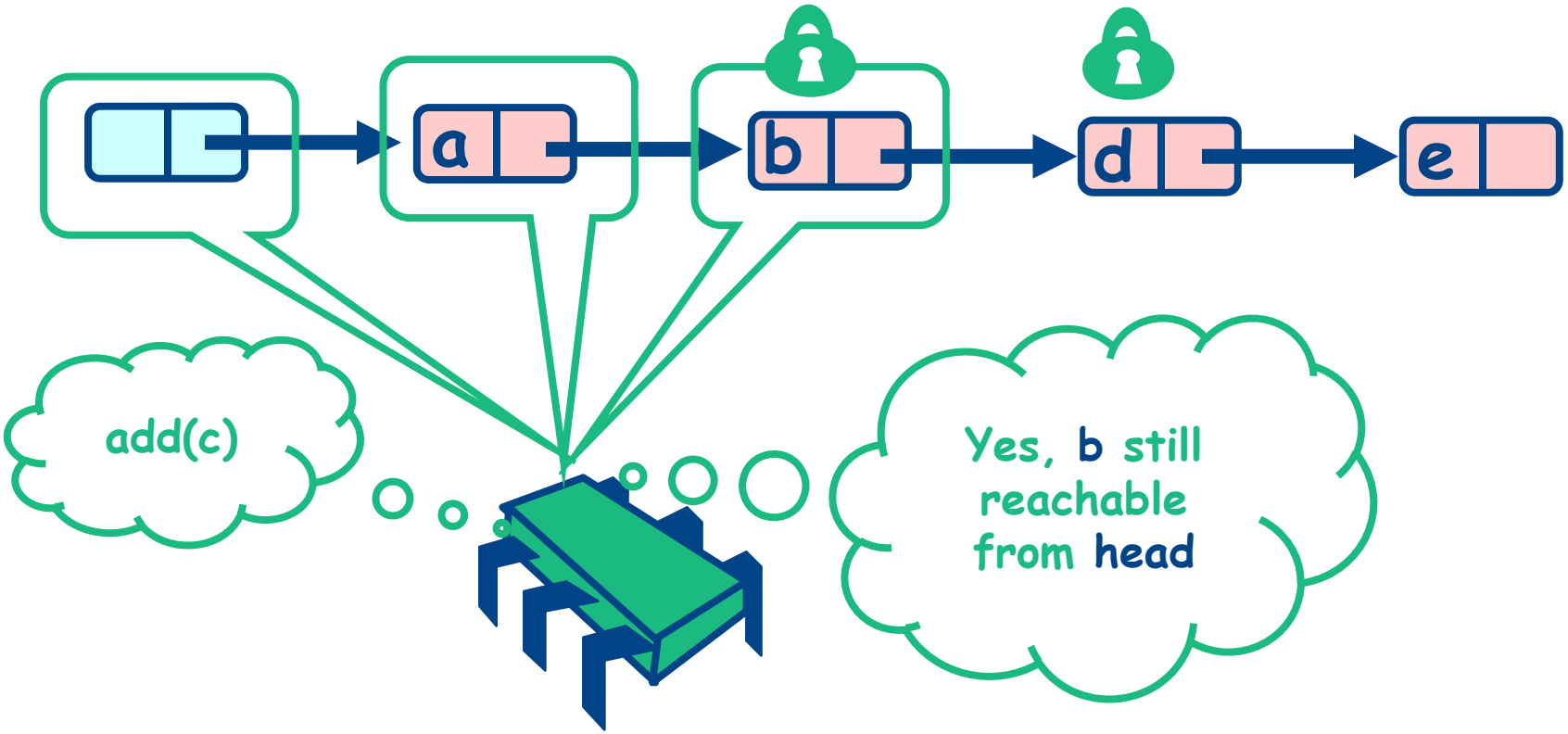
Optimistic: Lock and Load



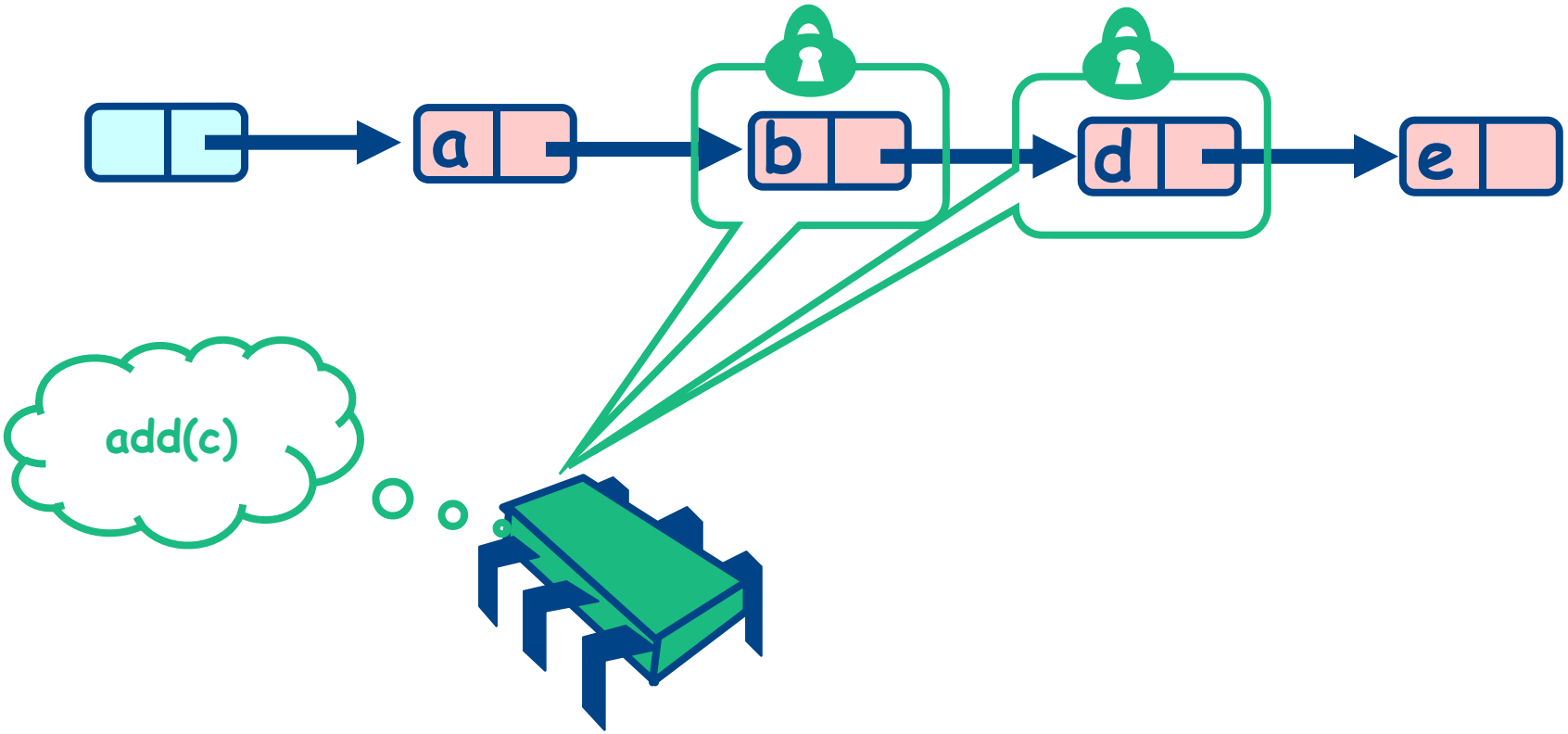
What could go wrong?



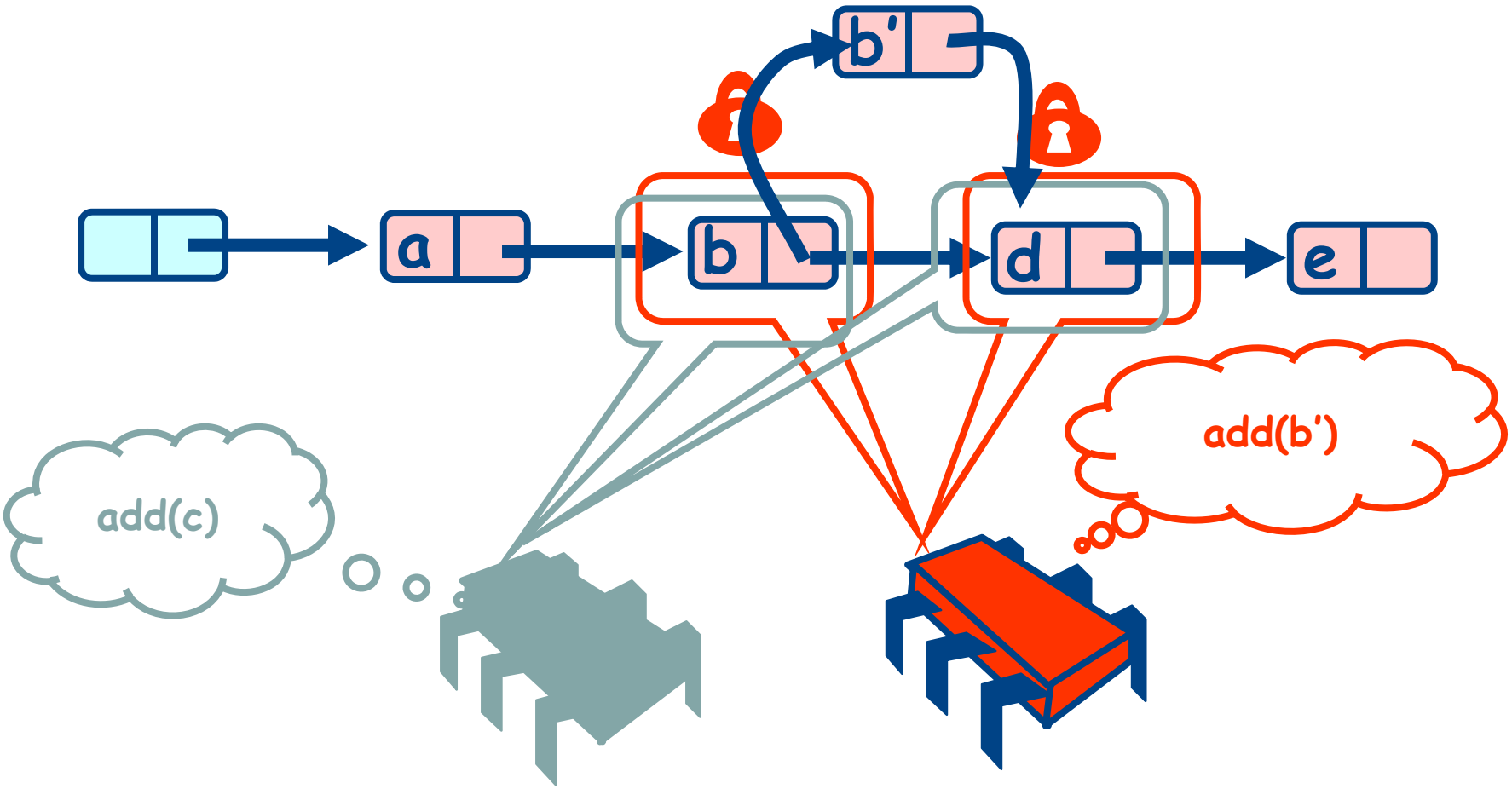
Validate – Part 1 (while holding locks)



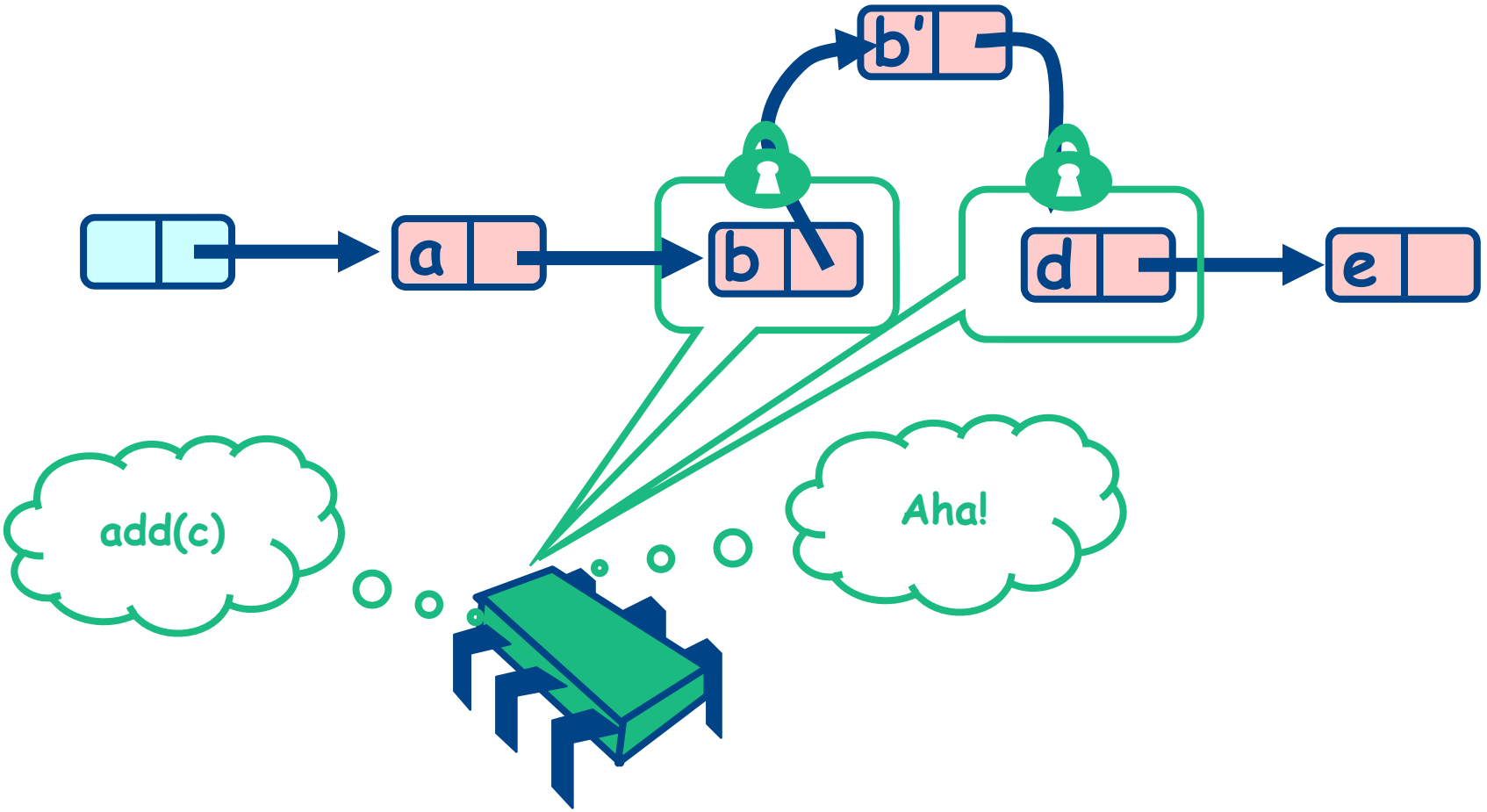
What Else Can Go Wrong?



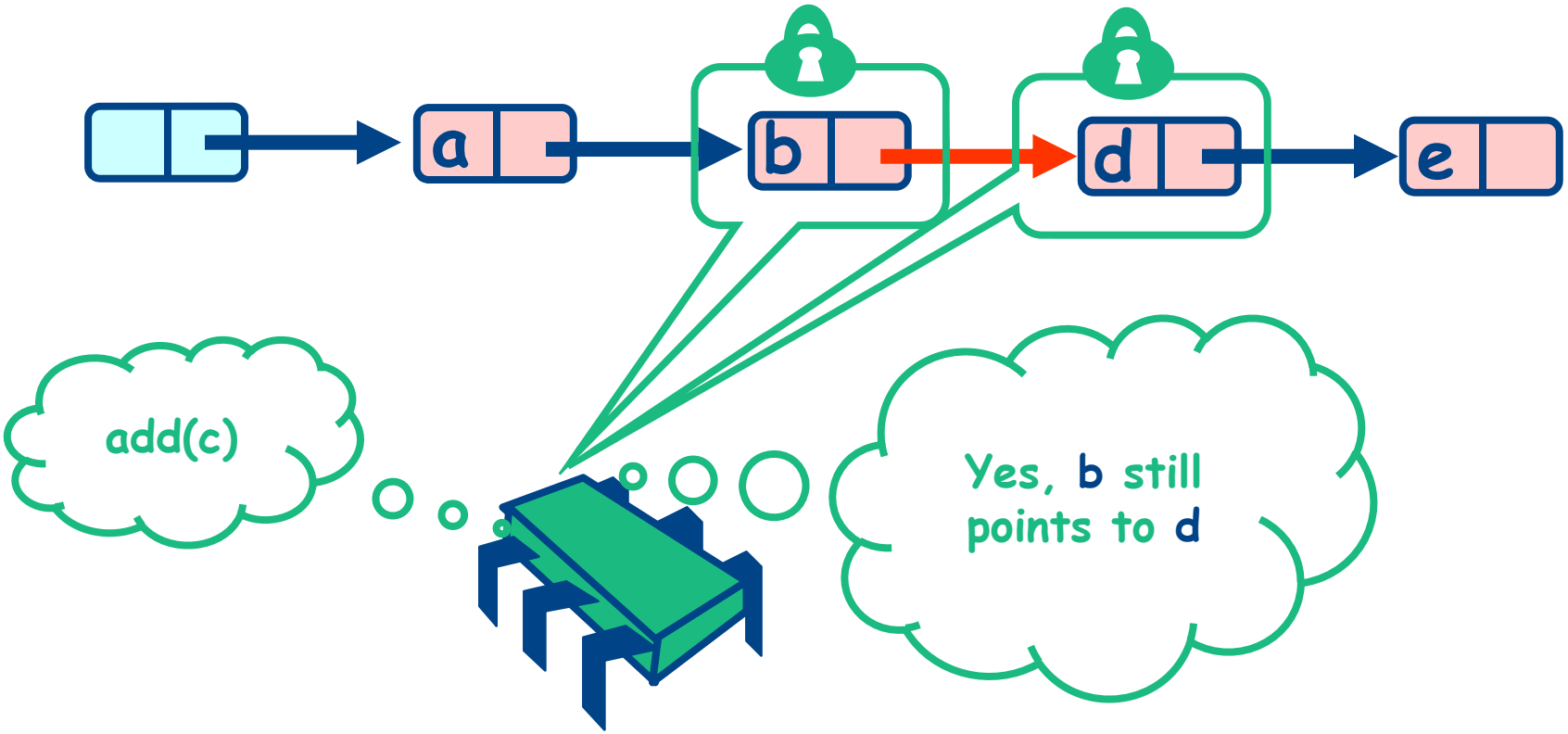
What Else Can Go Wrong?



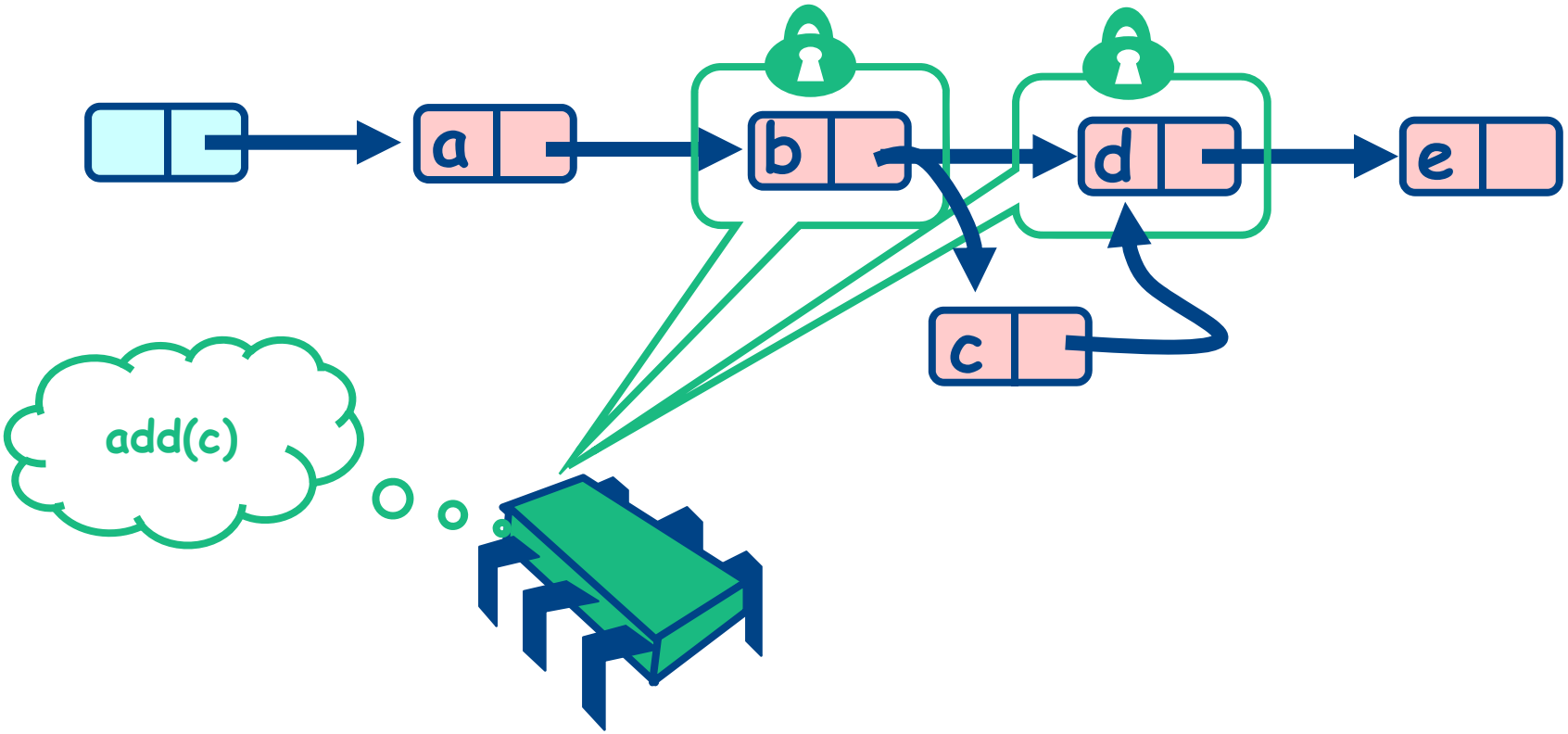
What Else Can Go Wrong?



Validate Part 2 (while holding locks)



Optimistic: Linearization



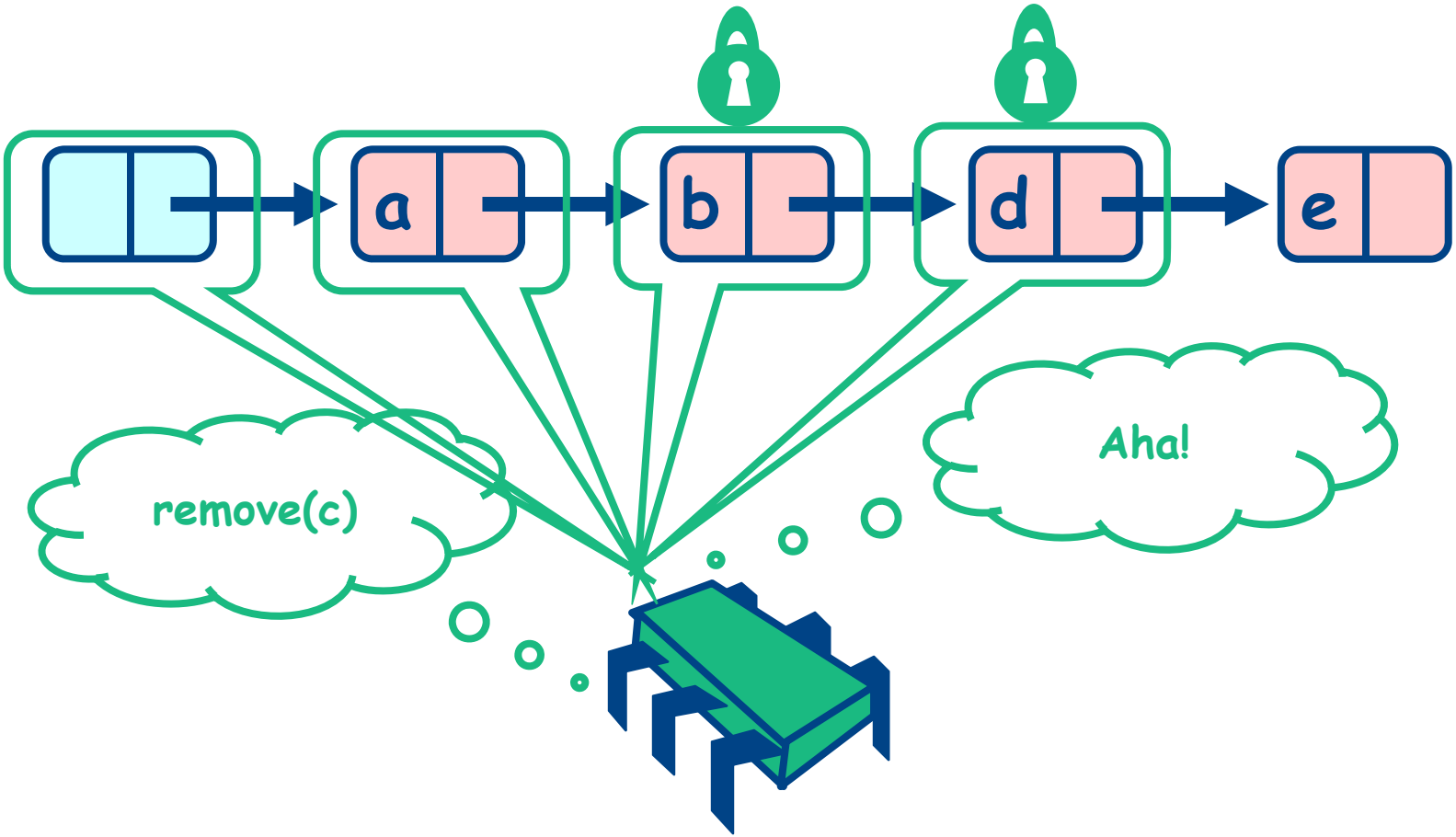
Invariants

- **Careful: we may traverse deleted nodes**
- **But we establish properties by**
  - Validation
  - After we lock target nodes

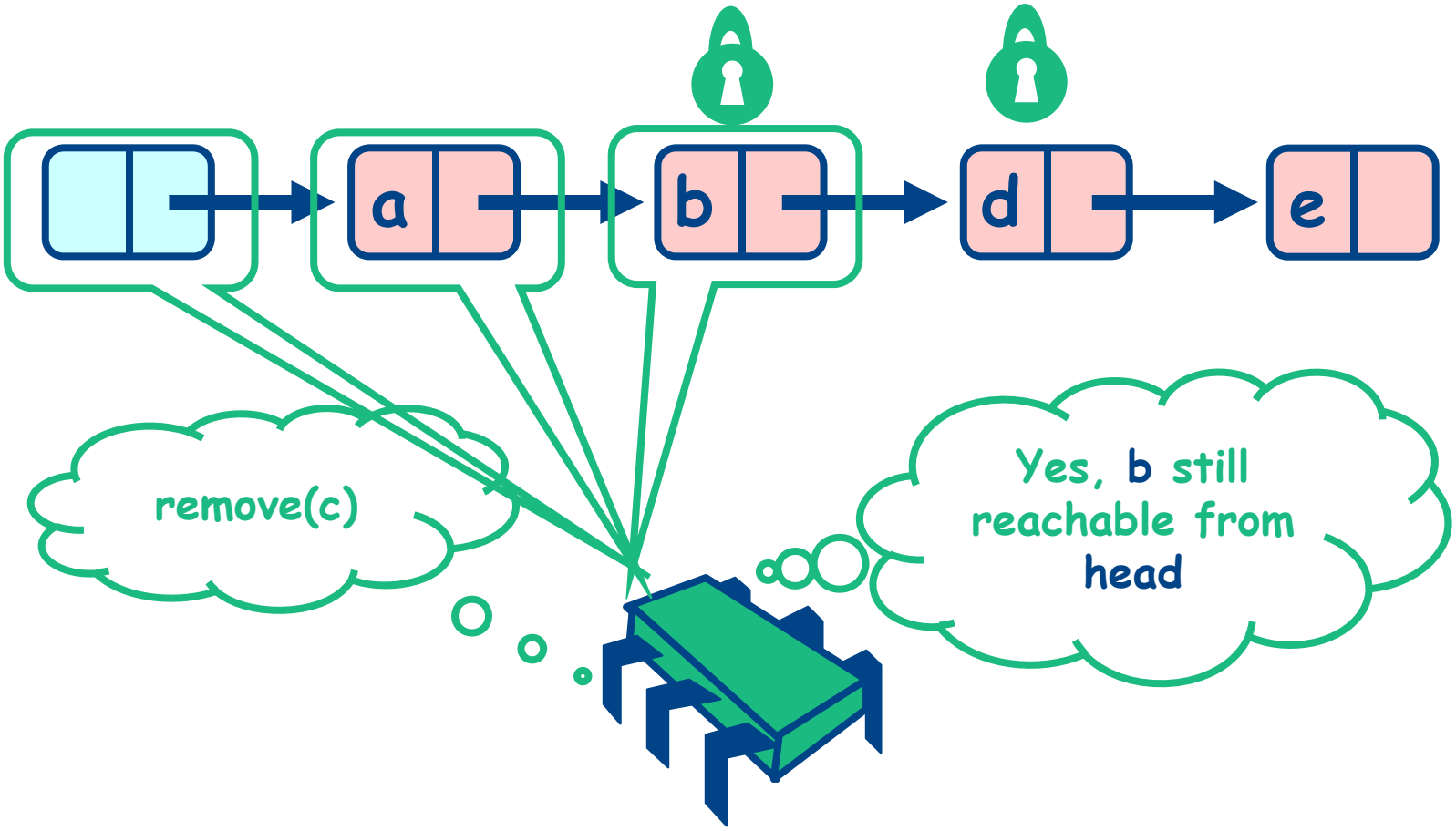
Correctness

- **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

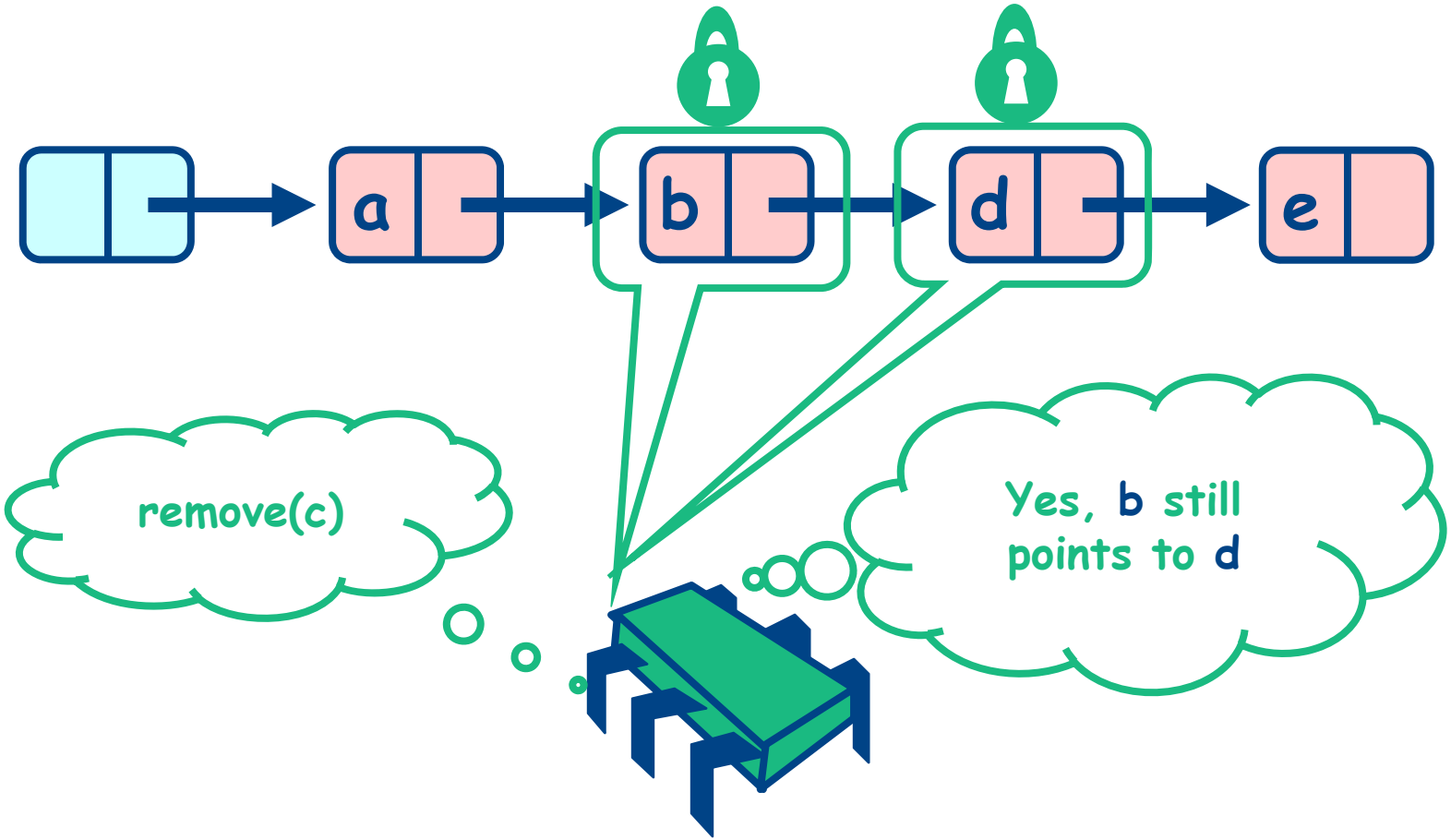
Unsuccessful Remove



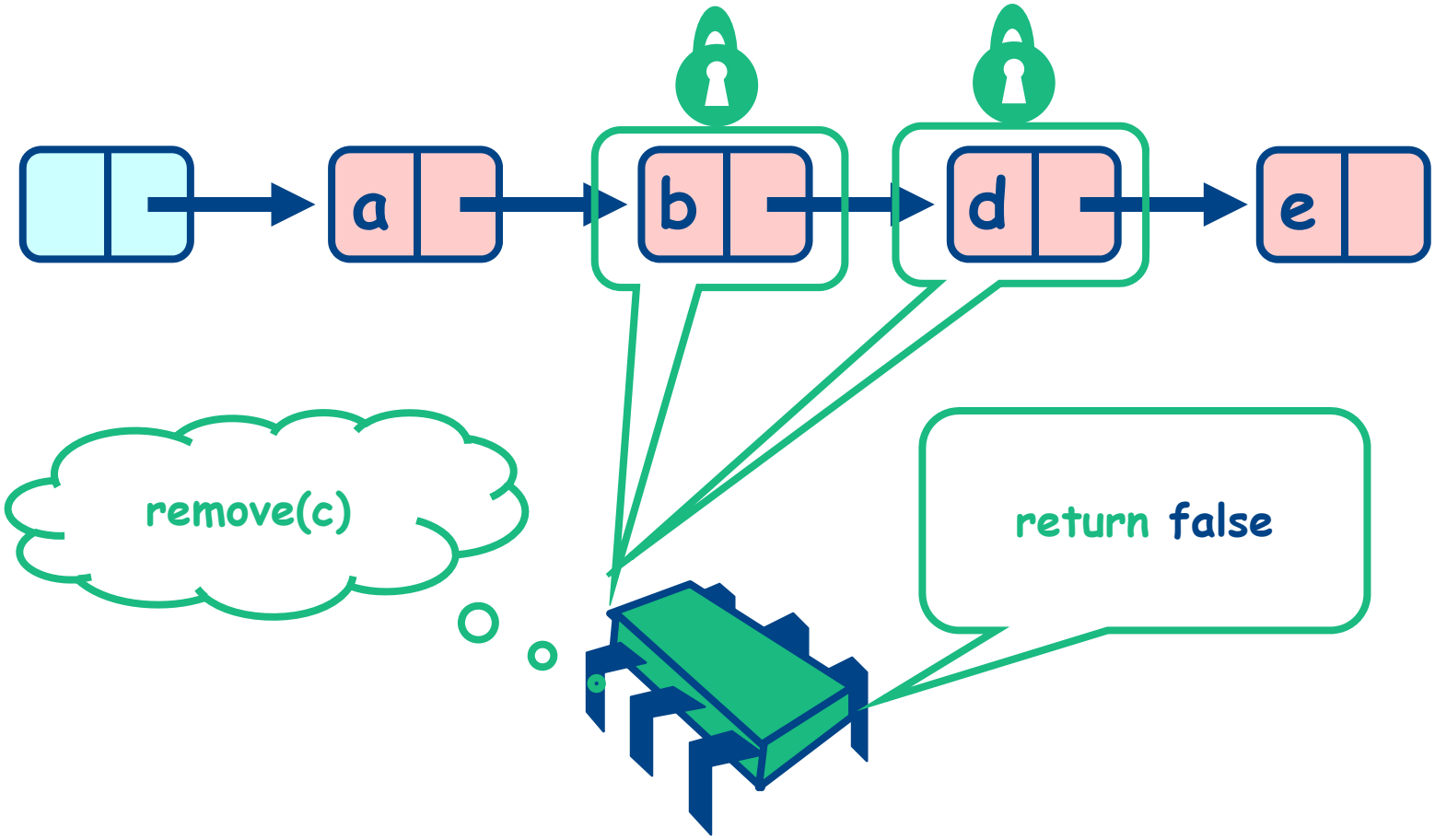
Validate (1)



Validate (2)



OK Computer



Validate and node(lock free traversal)

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

```
private class Entry {
   volatile T item;
   volatile int key;
   volatile Entry next;
   Lock lock;
   Entry(T item) {
     this.item = item;
     this.key = item.hashCode();
     lock = new ReentrantLock();
   }
   Entry(int key) {
     this.key = key;
     lock = new ReentrantLock();
   }
   void lock() {lock.lock();}
   void unlock() {lock.unlock();}
}
```

add

```
public boolean add(T item) {
  int key = item.hashCode();
  while (true) {
    Entry pred = this.head;
    Entry curr = pred.next;
    while (curr.key <= key) {
      pred = curr; curr = curr.next;
    }
    pred.lock(); curr.lock();
    try {
      if (validate(pred, curr)) {
        if (curr.key == key) { // present
          return false;
        } else {                    // not present
          Entry entry = new Entry(item);
          entry.next = curr;
          pred.next = entry;
          return true;
        }
      }
    } finally {                     // always unlock
      pred.unlock(); curr.unlock();
    }
  }
}
```

remove

```
public boolean remove(T item) {
  int key = item.hashCode();
  while (true) {
    Entry pred = this.head;
    Entry curr = pred.next;
    while (curr.key < key) {
      pred = curr; curr = curr.next;
    }
    pred.lock(); curr.lock();
    try {
      if (validate(pred, curr)) {
        if (curr.key == key) { // present in list
          pred.next = curr.next;
          return true;
        } else {               // not present in list
          return false;
        }
      }
    } finally {                // always unlock
      pred.unlock(); curr.unlock();
    }
  }
}
```

contains

```
public boolean contains(T item) {
  int key = item.hashCode();
  while (true) {
    Entry pred = this.head; // sentinel node;
    Entry curr = pred.next;
    while (curr.key < key) {
      pred = curr; curr = curr.next;
    }
    try {
      pred.lock(); curr.lock();
      if (validate(pred, curr)) {
        return (curr.key == key);
      }
    } finally {                    // always unlock
      pred.unlock(); curr.unlock();
    }
  }
}
```

Correctness

- **If**
  - Nodes b and d both locked
  - Node b still accessible
  - Node d still successor to b
- **Then**
  - Neither will be deleted
  - No thread can add c after b
  - OK to return false

Optimistic List

- **Limited hot-spots**
  - Targets of add(), remove(), contains()
  - No contention on traversals
- **Moreover**
  - Traversals are wait-free
  - Food for thought …

So Far, So Good

- **Much less lock acquisition / release**
  - Performance
  - Concurrency
- **Problems**
  - Need to traverse list twice
  - contains() method acquires locks

Evaluation

- **Optimistic is effective if**
  - cost of scanning twice without locks
    is less than
  - cost of scanning once with locks
- **Drawback**
  - contains() acquires locks
  - 90% of calls in many apps

Lazy List

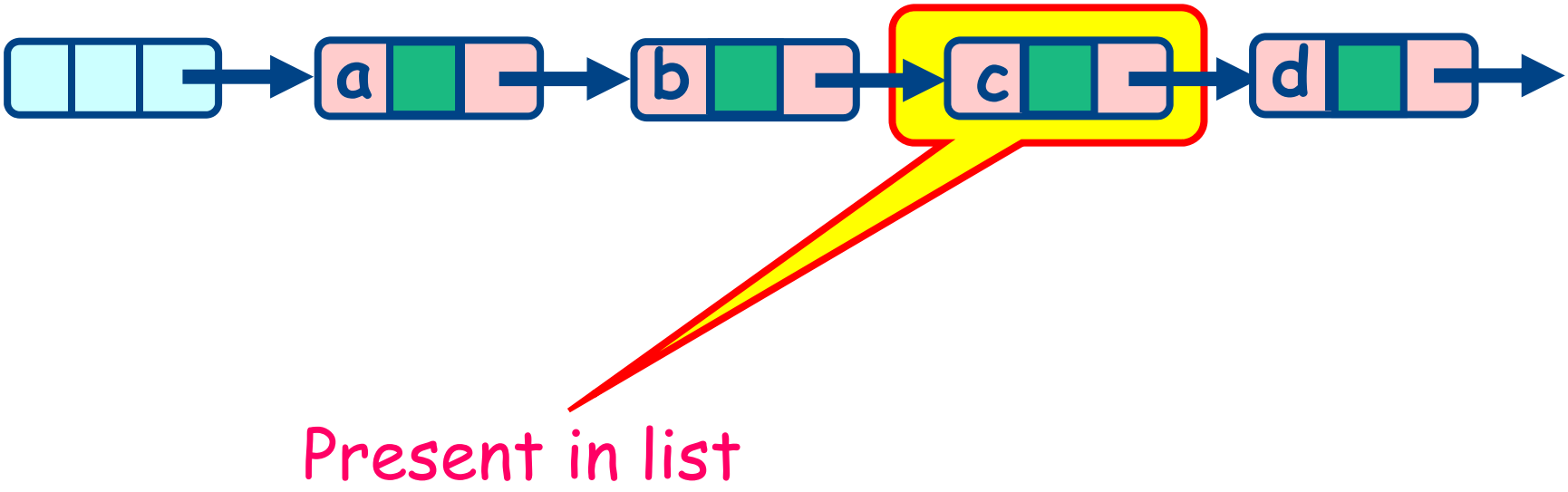
- **Like optimistic, except**
  - Scan once
  - `contains(x)` never locks …
- **Key insight**
  - Removing nodes causes trouble
  - Do it “lazily”

Lazy List

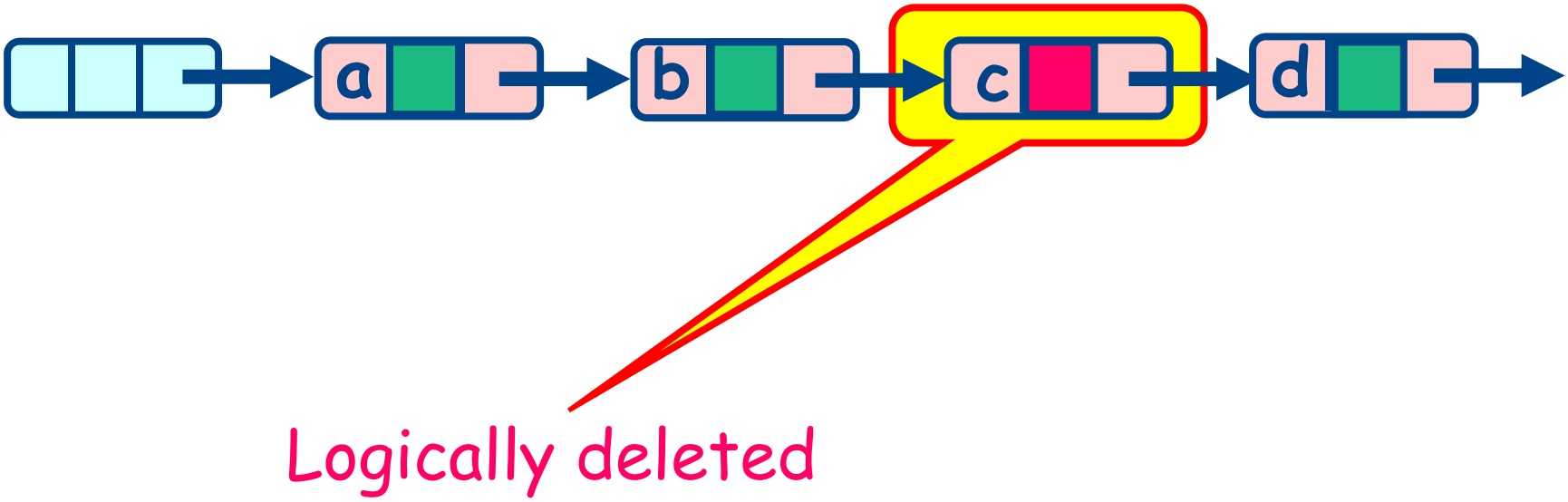
- `remove()`
  - Scans list (as before)
  - Locks predecessor & current (as before)
- **Logical delete**
  - Marks current node as removed (new!)
- **Physical delete**
  - Redirects predecessor's next (as before)

Lazy Removal

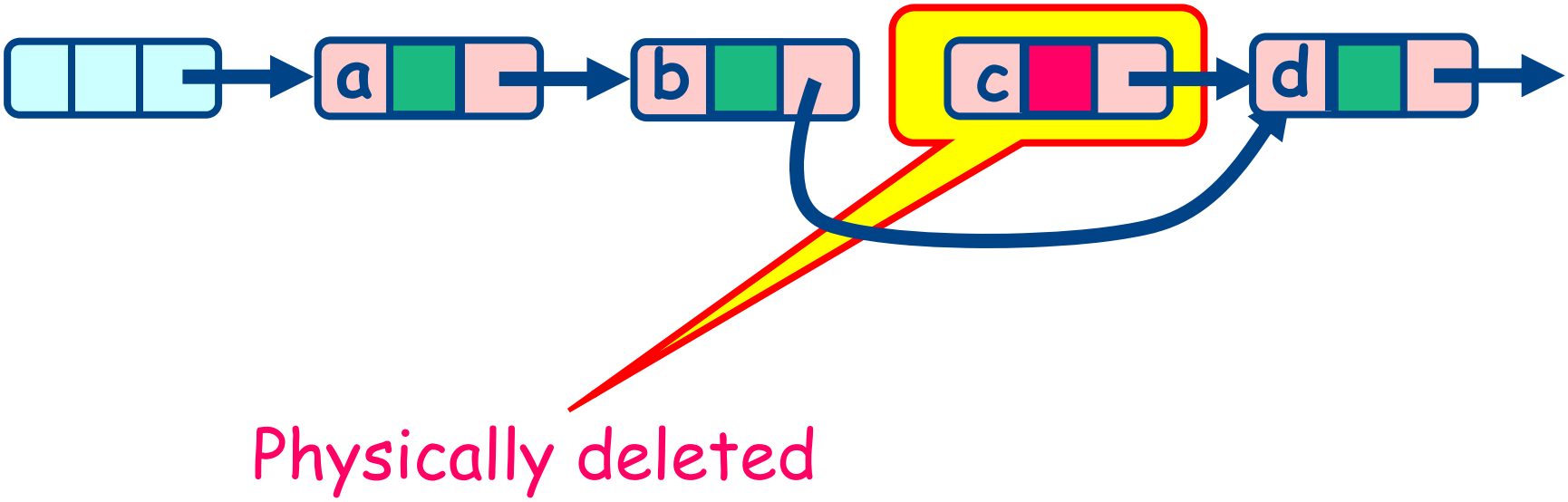
Lazy Removal



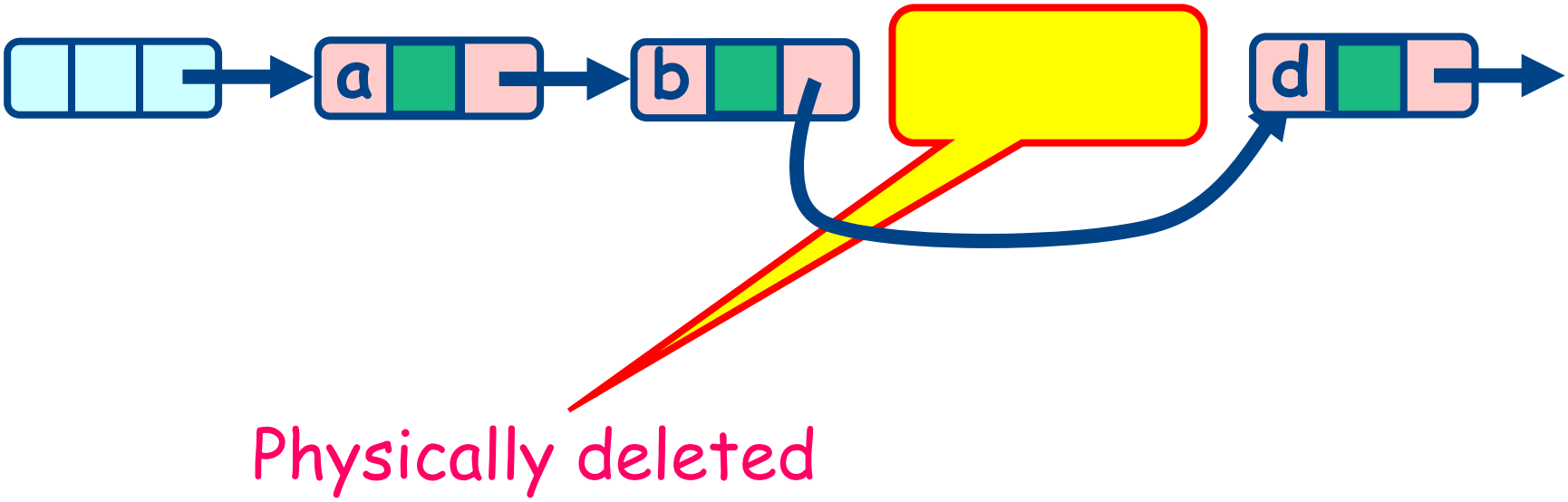
Lazy Removal



Lazy Removal



Lazy Removal



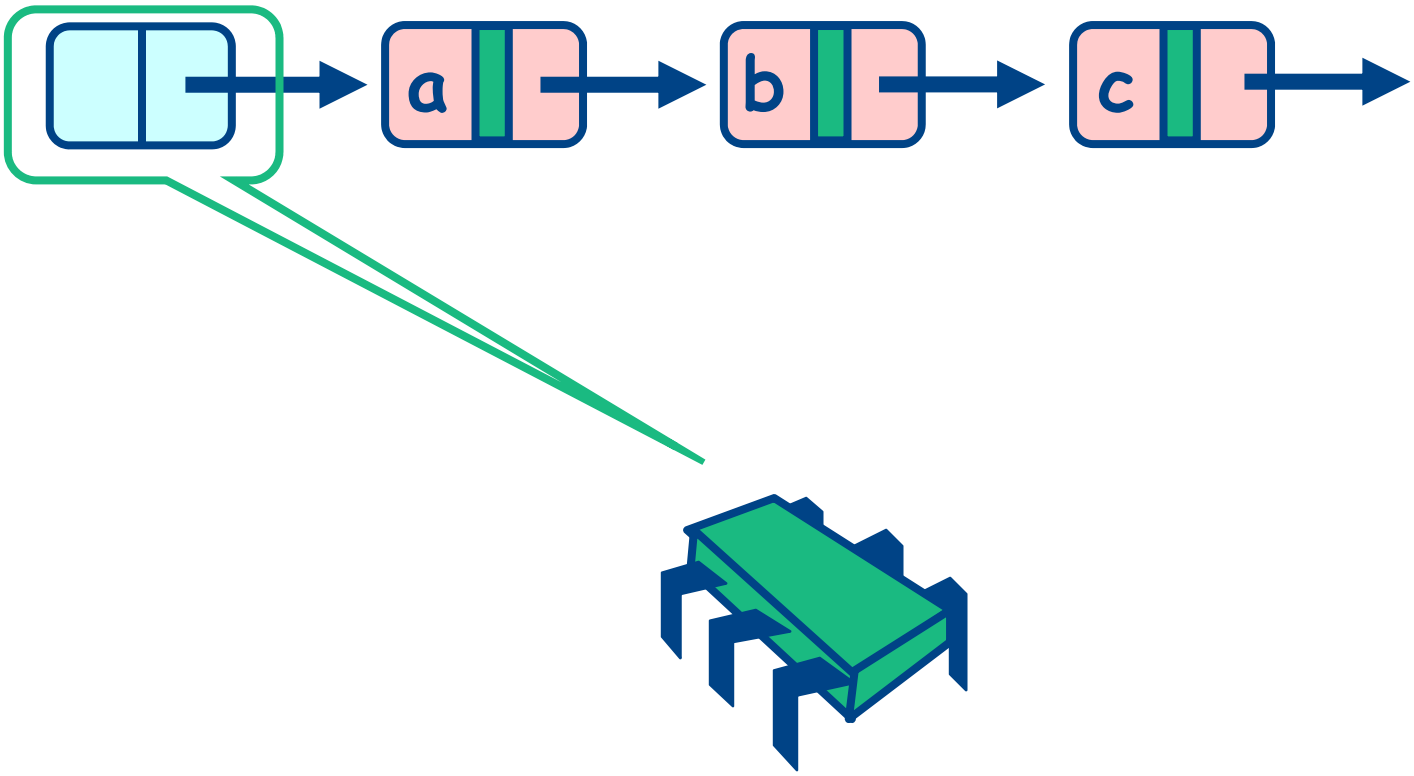
Lazy List

- **All Methods**
  - Scan through locked and marked nodes
  - Removing a node doesn't slow down other method calls …
- **Must still lock pred and curr nodes.**

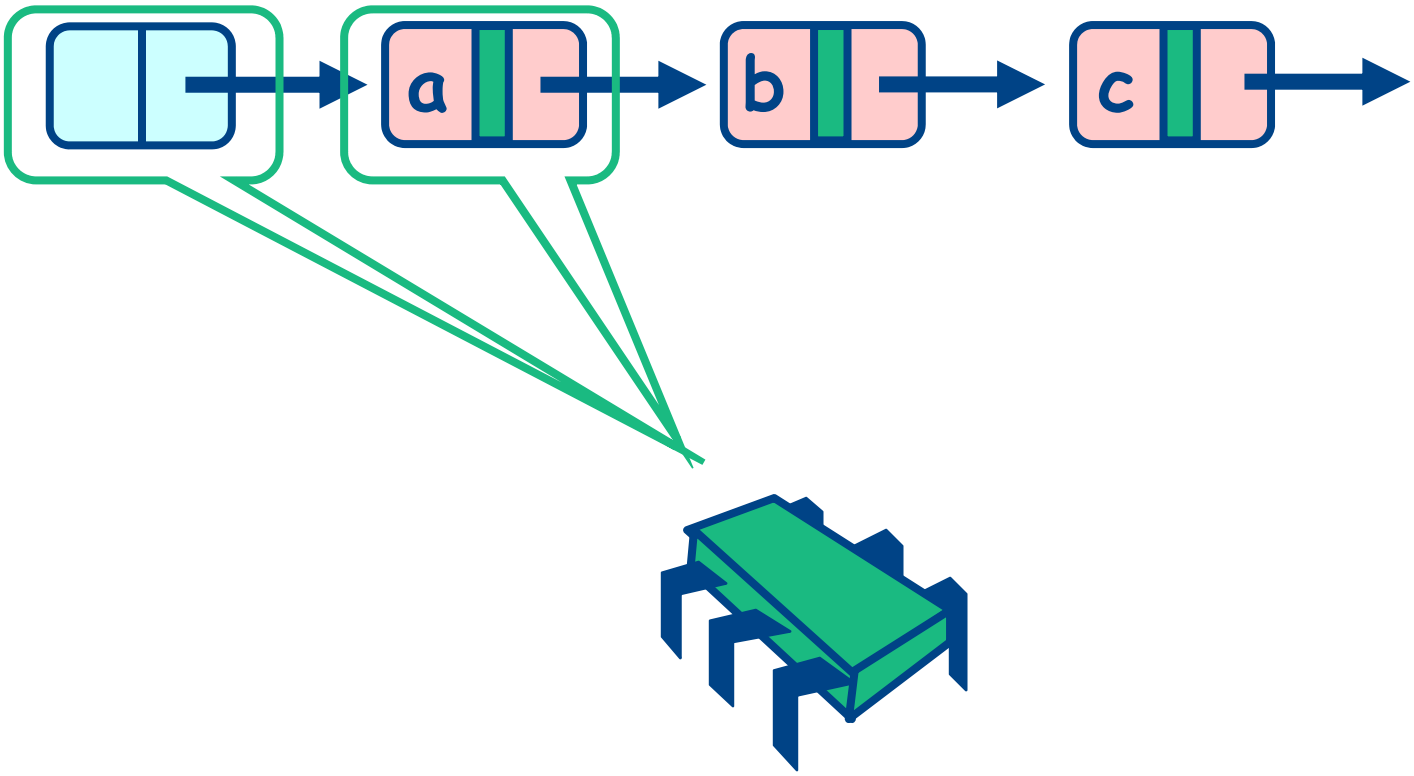
Validation

- No need to rescan list!
- Check that **pred** is not marked
- Check that **curr** is not marked
- Check that **pred** points to **curr**

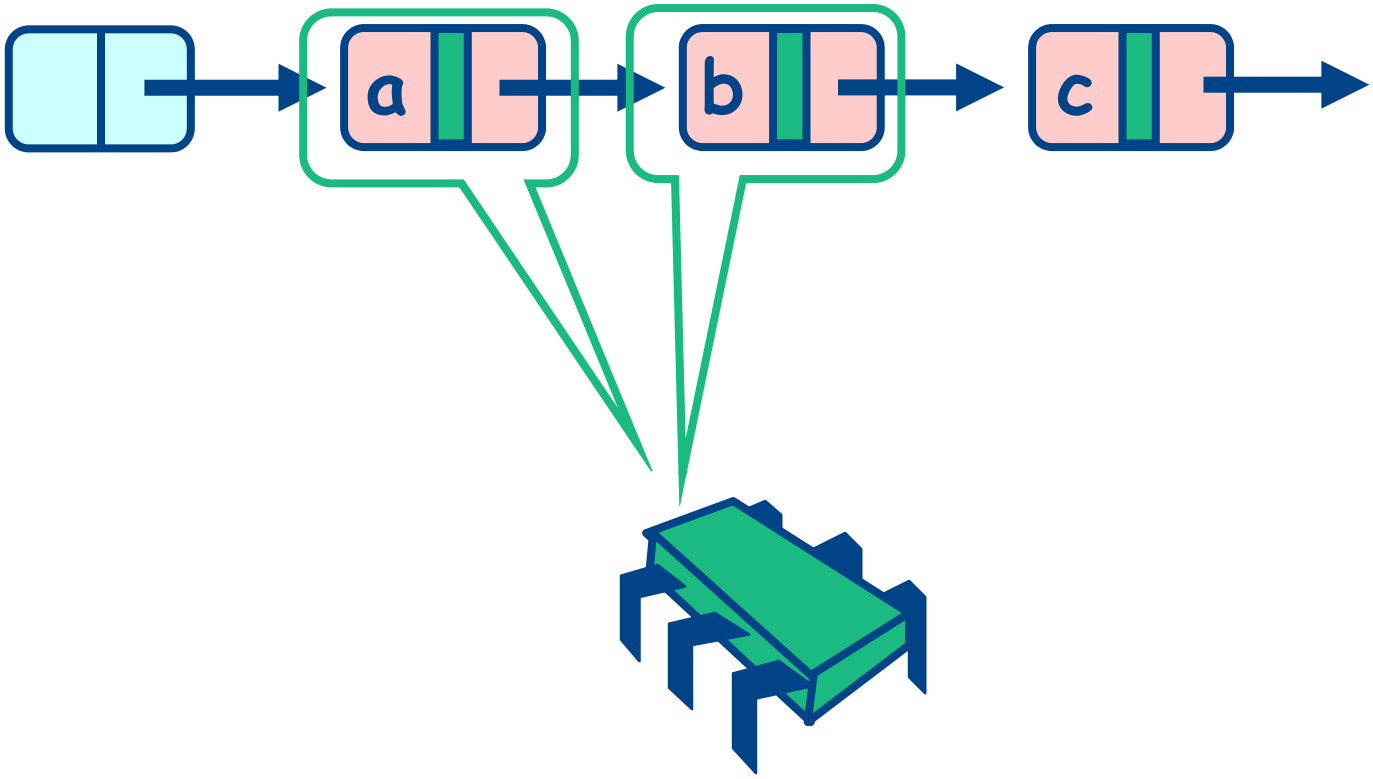
Business as Usual



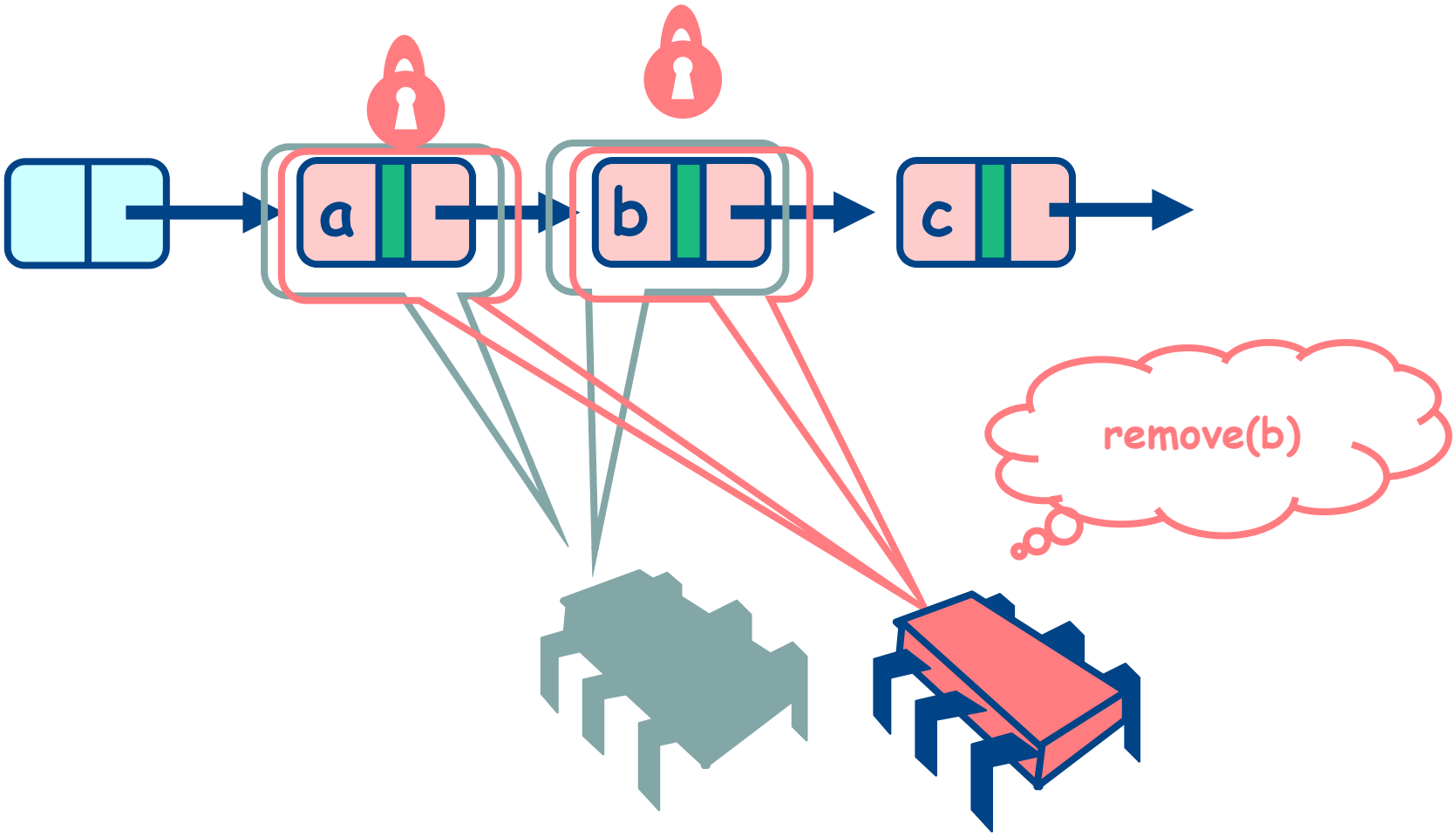
Business as Usual



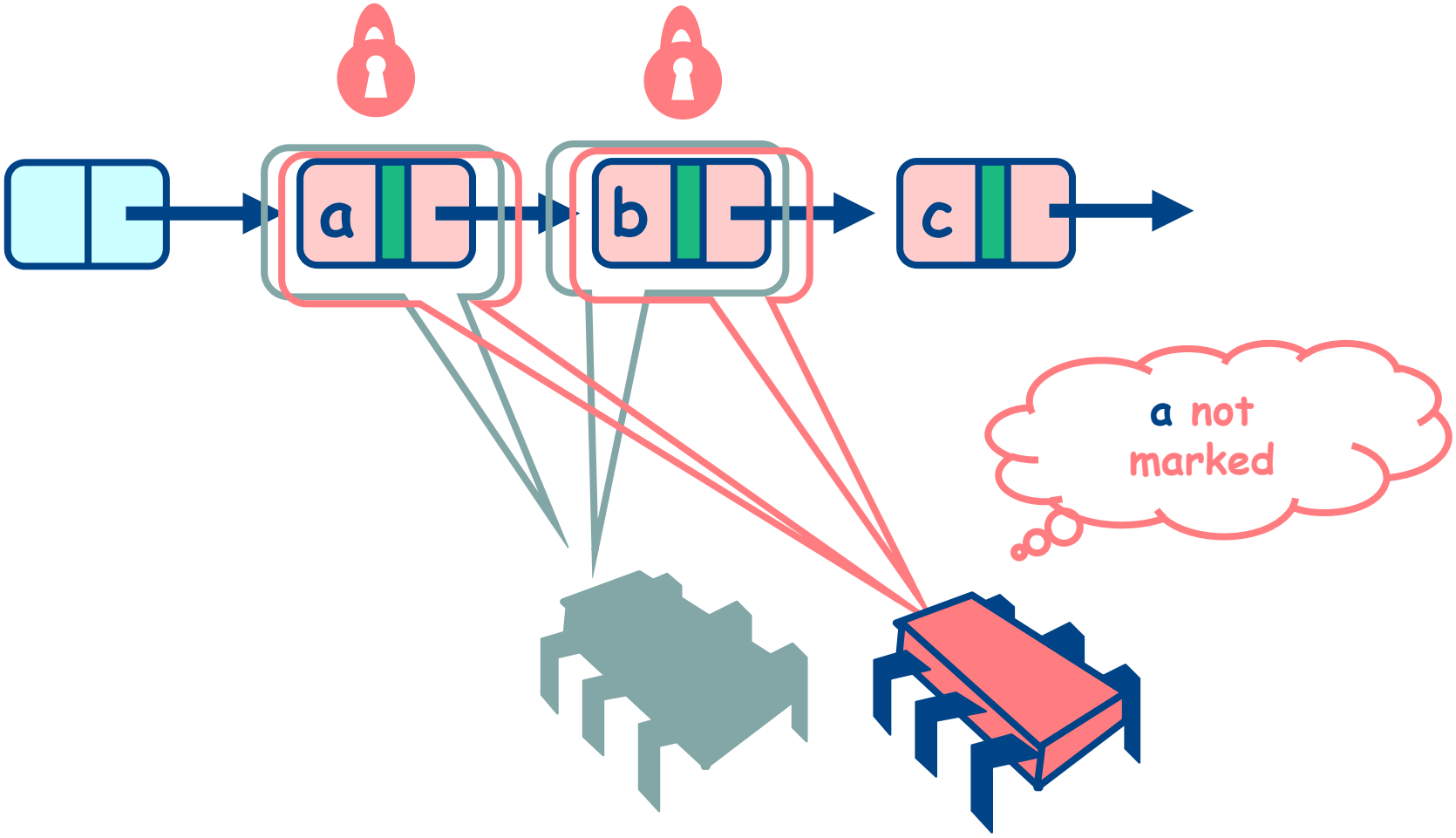
Business as Usual



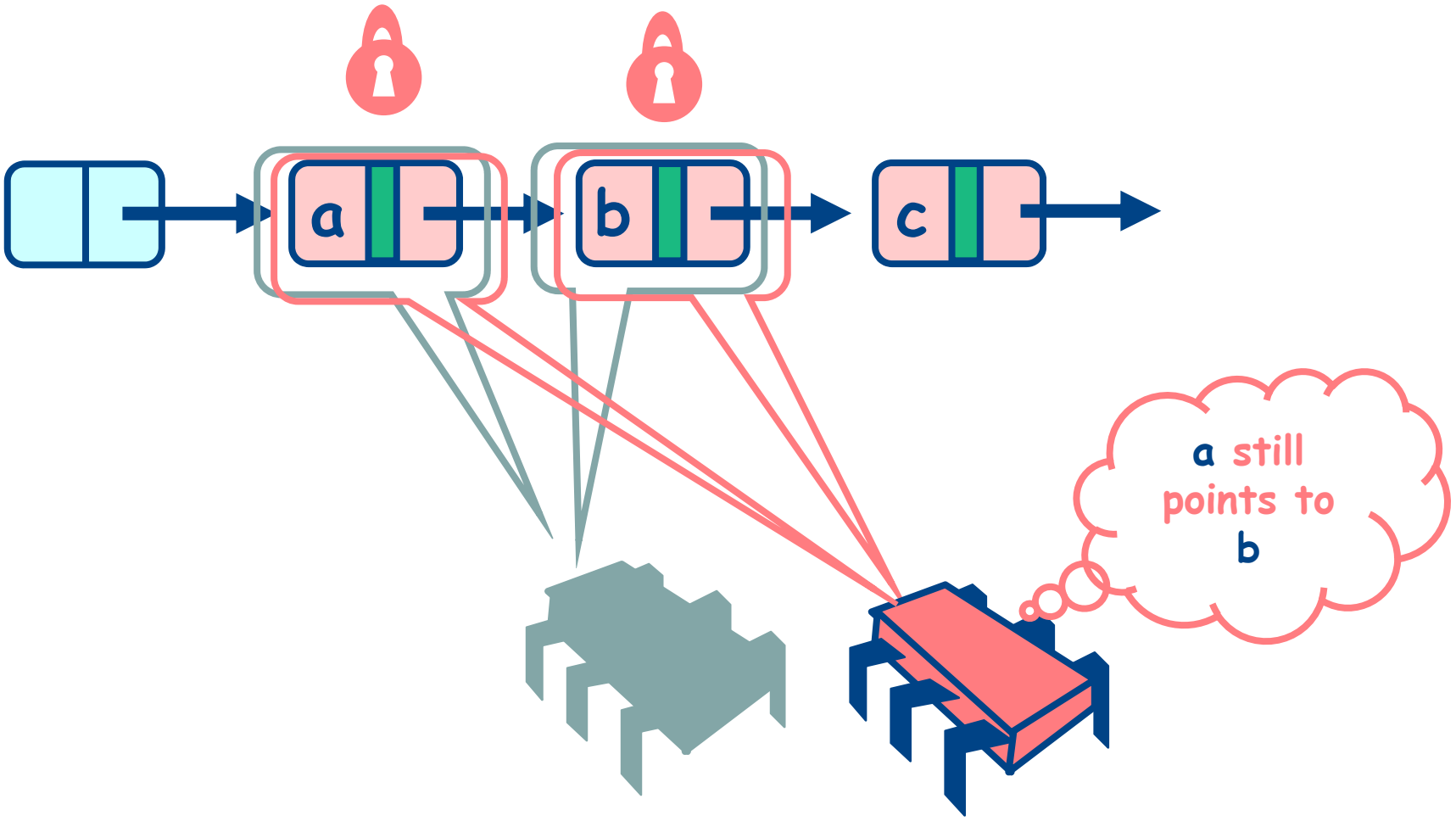
Business as Usual



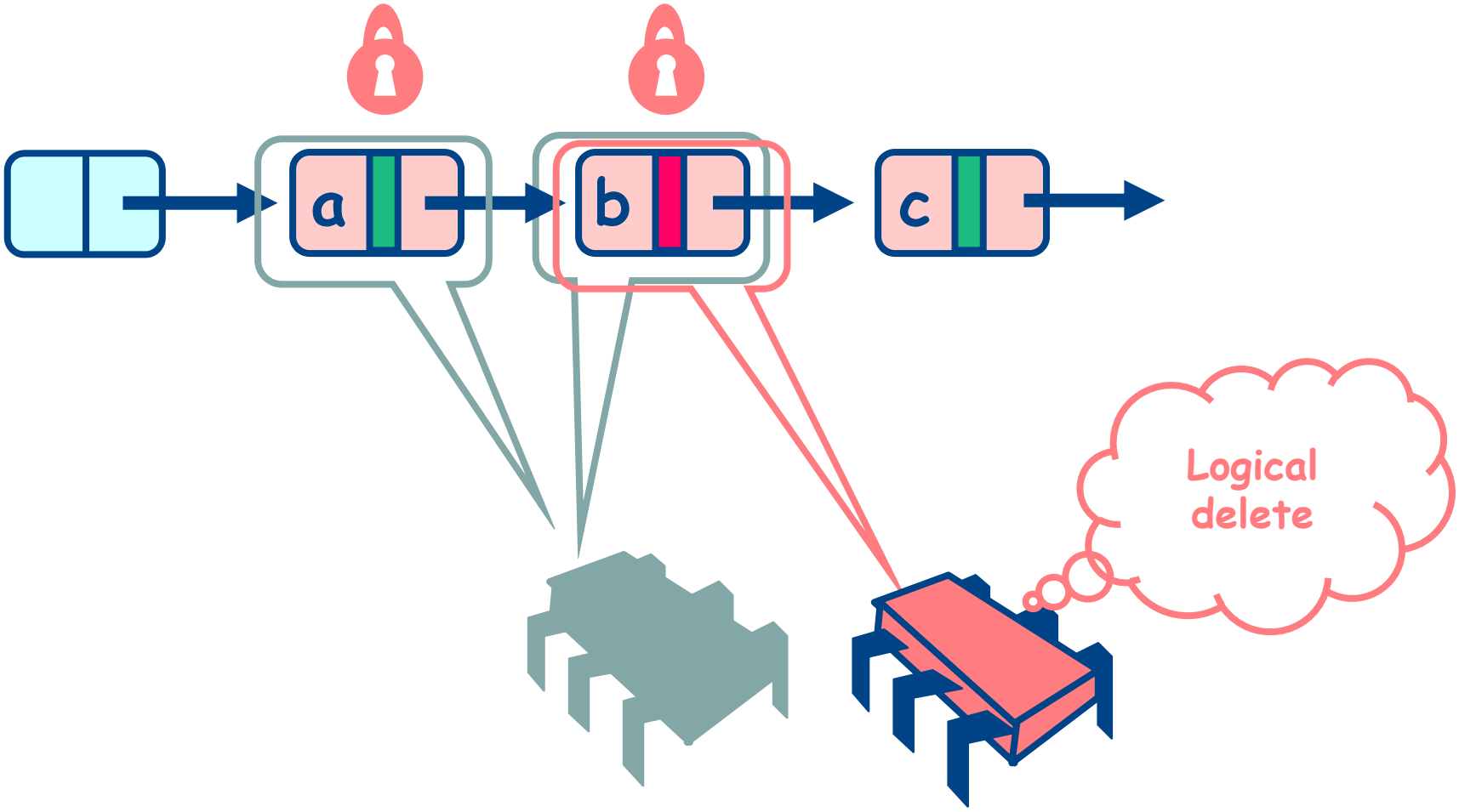
Business as Usual



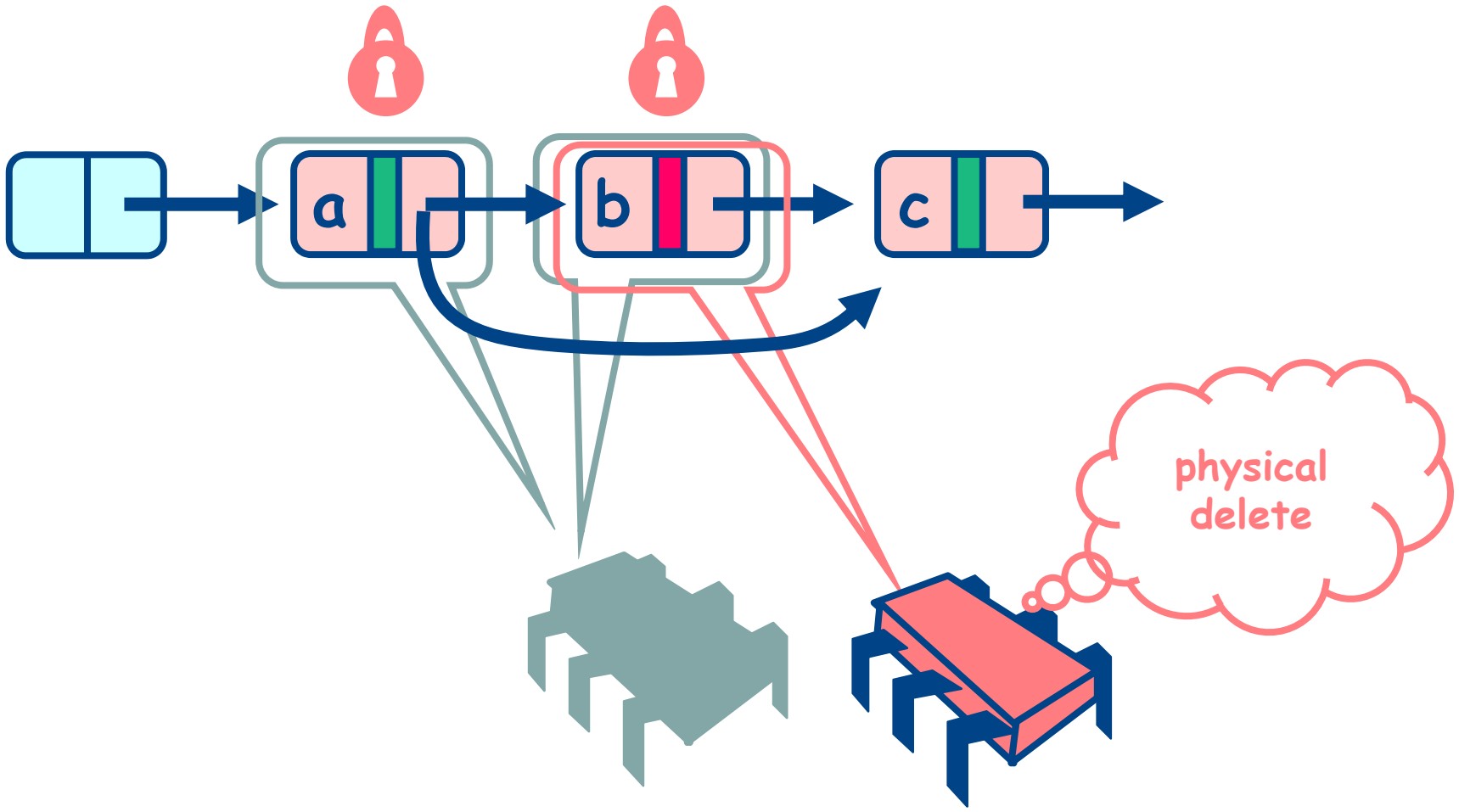
Business as Usual



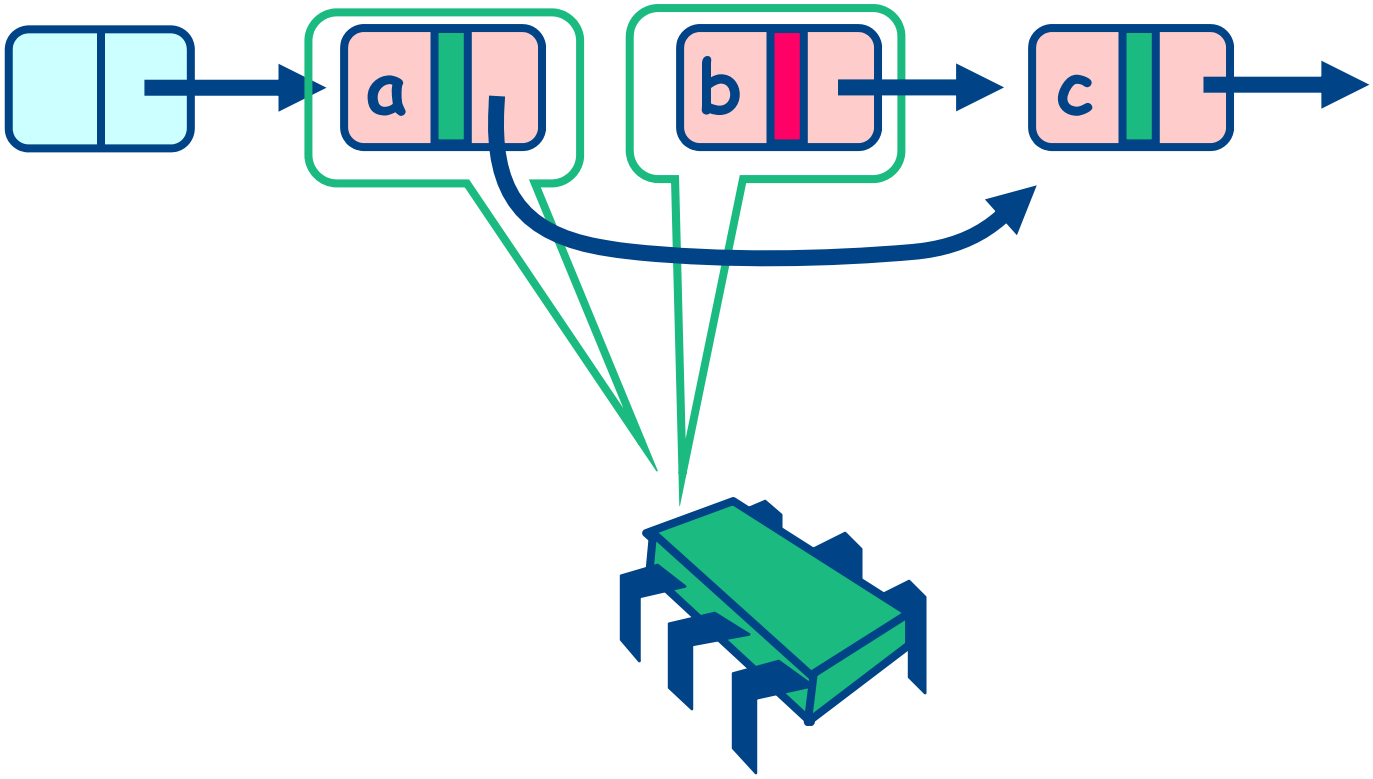
Business as Usual



Business as Usual



Business as Usual



Invariant

- If not marked then item in the set
- and reachable from **head**
- and if not yet traversed it is reachable from **pred**

Validate and node

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

```
private class Node {
   volatile T item;
   volatile int key;
   volatile Node next;
   volatile boolean marked;
   Lock lock;
   Node(T item) {        // usual constructor
      this.item = item;
      this.key = item.hashCode();
      this.next = null;
      this.marked = false;
      this.lock = new ReentrantLock();
   }
   Node(int key) { // sentinel constructor
      this.item = null;
      this.key = key;
      this.next = null;
      this.marked = false;
      this.lock = new ReentrantLock();
   }
   void lock() {lock.lock();}
   void unlock() {lock.unlock();}
}
```

add

```
public boolean add(T item) {
  int key = item.hashCode();
  while (true) {
    Node pred = this.head;
    Node curr = head.next;
    while (curr.key < key) {
      pred = curr; curr = curr.next;
    }
    pred.lock();
    try {
      curr.lock();
      try {
        if (validate(pred, curr)) {
          if (curr.key == key) { // present
            return false;
          } else {                    // not present
            Node Node = new Node(item);
            Node.next = curr;
            pred.next = Node;
            return true;
          }
        }
      } finally { // always unlock
        curr.unlock();
      }
    } finally { // always unlock
      pred.unlock();
    }
  }
}
```

remove

```
public boolean remove(T item) {
  int key = item.hashCode();
  while (true) {
    Node pred = this.head;
    Node curr = head.next;
    while (curr.key < key) {
      pred = curr; curr = curr.next;
    }
    pred.lock();
    try {
      curr.lock();
      try {
        if (validate(pred, curr)) {
          if (curr.key != key) {     // present
            return false;
          } else {                   // absent
            curr.marked = true;      // logically remove
            pred.next = curr.next;   // physically remove
            return true;
          }
        }
      } finally {                    // always unlock curr
        curr.unlock();
      }
    } finally {                      // always unlock pred
      pred.unlock();
    }
  }
}
```

contains

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

Evaluation

- **Good:**
  - contains() doesn't lock
  - In fact, its wait-free!
  - Good because typically high % contains()
  - Uncontended calls don't re-traverse
- **Bad**
  - Contended add() and remove() 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”
    - Cache miss, page fault, descheduled …
  - Everyone else using that lock is stuck!
  - Need to trust the scheduler….

Reminder: Lock-Free Data Structures

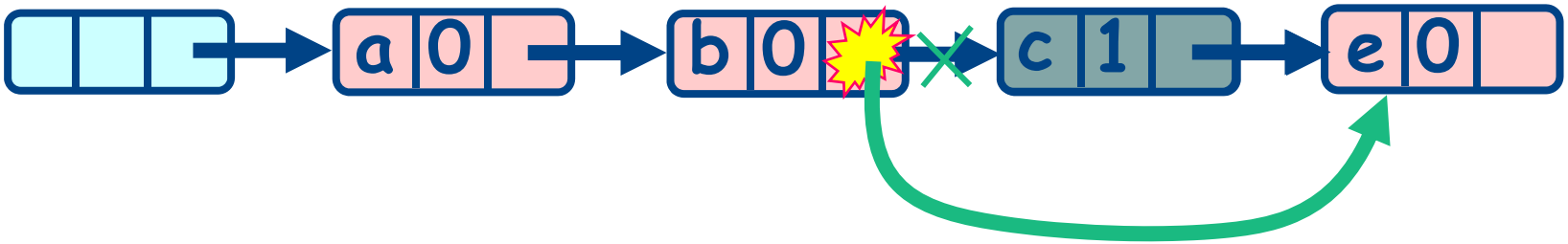
- **No matter what …**
  - Guarantees minimal progress in all execution
  - i.e. Some thread will always complete a method call
  - Even if others halt at malicious times
  - Implies that implementation can't use locks

Lock-free Lists

- Next logical step
- Eliminate locking entirely
- contains() wait-free and add() and remove() lock-free
- Use only compareAndSet()
- What could go wrong?

Remove Using CAS

Logical Removal =
Set Mark Bit



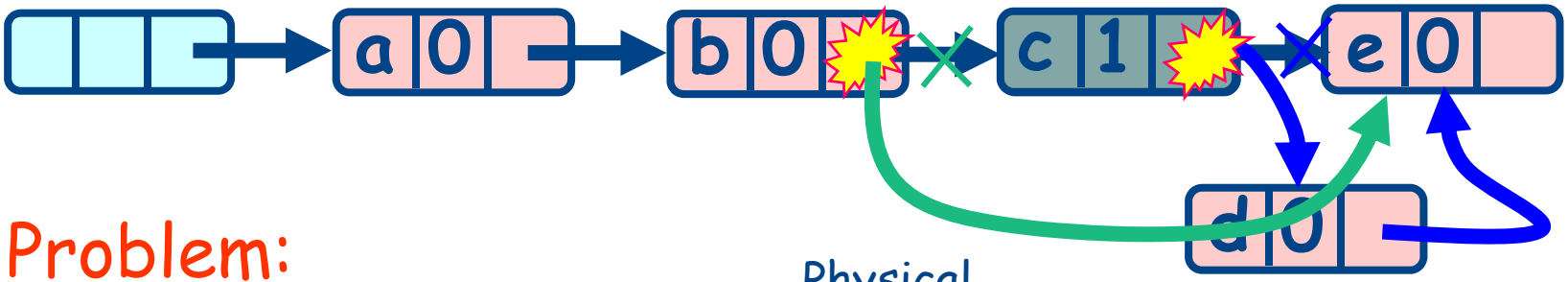
Use CAS to verify pointer is correct

Physical Removal CAS pointer

Not enough!

Problem…

Logical Removal =
Set Mark Bit



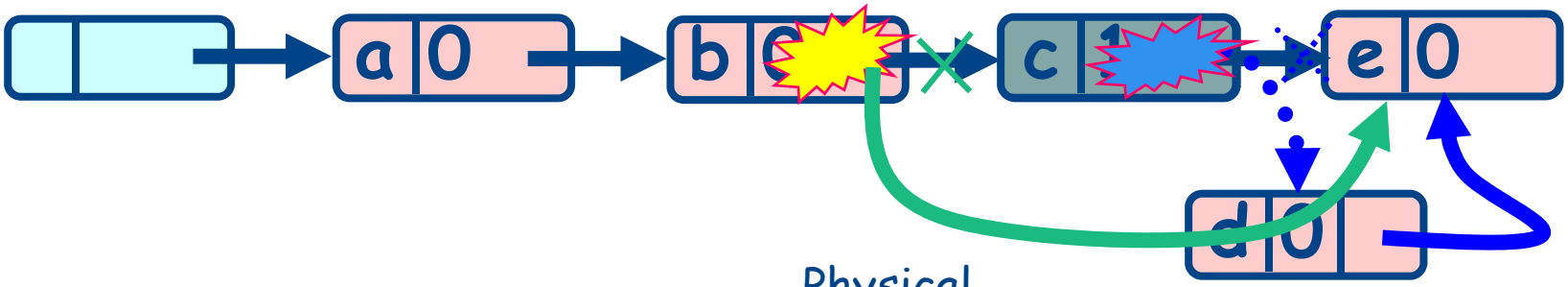
Problem:
d not added to list…
Must Prevent manipulation of removed node's pointer

Physical Removal CAS

Node added Before Physical Removal CAS

The Solution: Combine Bit

Logical Removal =
Set Mark Bit



Physical
Removal
CAS

Fail CAS: Node not added after logical Removal

Mark-Bit and Pointer are CASed together (AtomicMarkableReference)

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

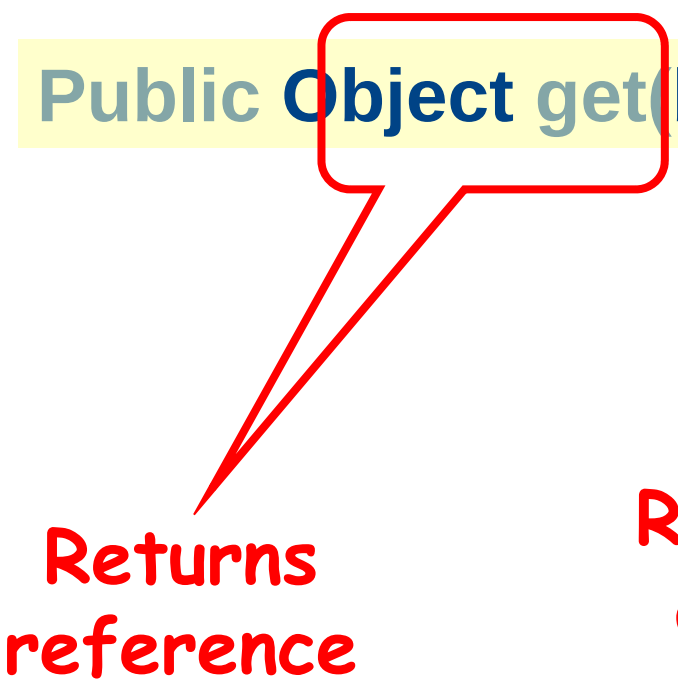
- AtomicMarkableReference class
  - Java.util.concurrent.atomic package

Extracting Reference & Mark

Public Object get(boolean[] marked);

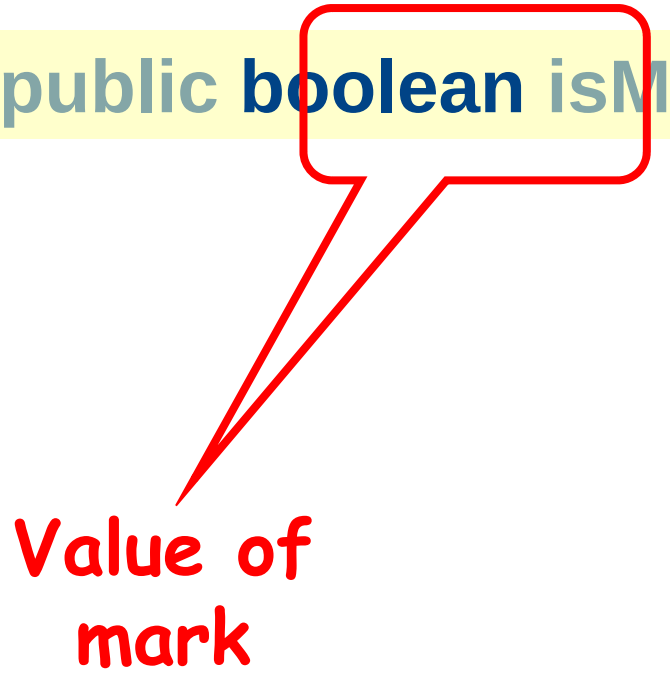
Extracting Reference & Mark

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



Extracting Reference Only

```
public boolean isMarked();
```



Changing State

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

Changing State

If this is the current reference …

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

And this is the current mark …

Changing State

…then change to this new reference …

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

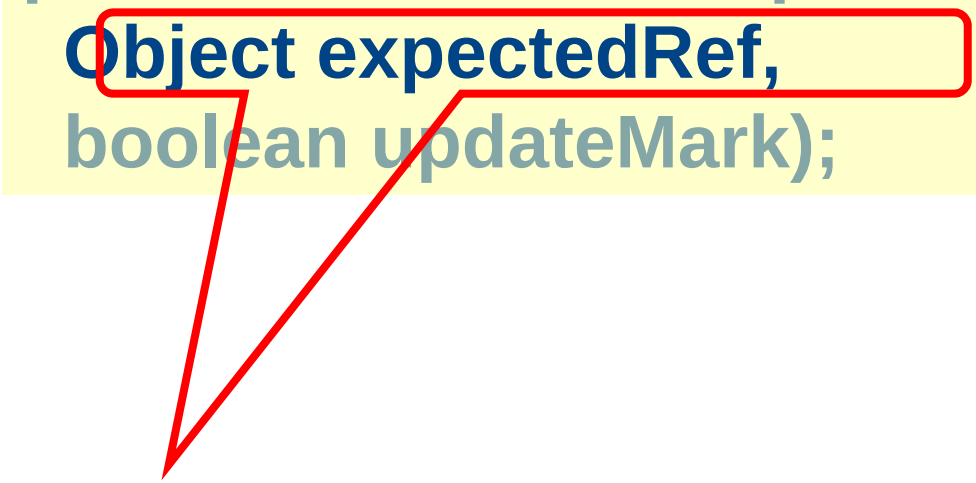
… and this new mark

Changing State

```
public boolean attemptMark(
  Object expectedRef,
  boolean updateMark);
```

Changing State

```
public boolean attemptMark(
  Object expectedRef,
  boolean updateMark);
```

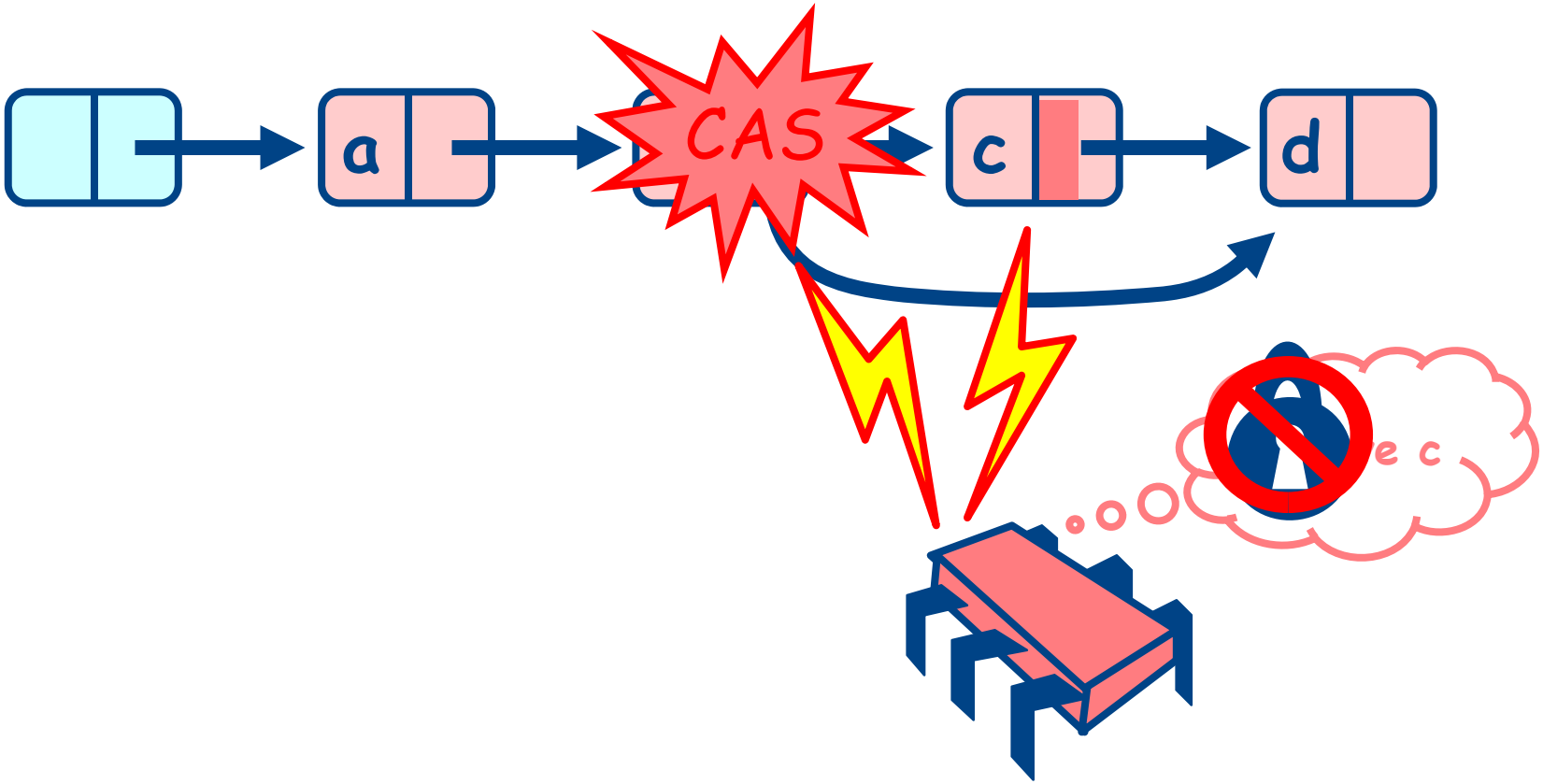


Changing State

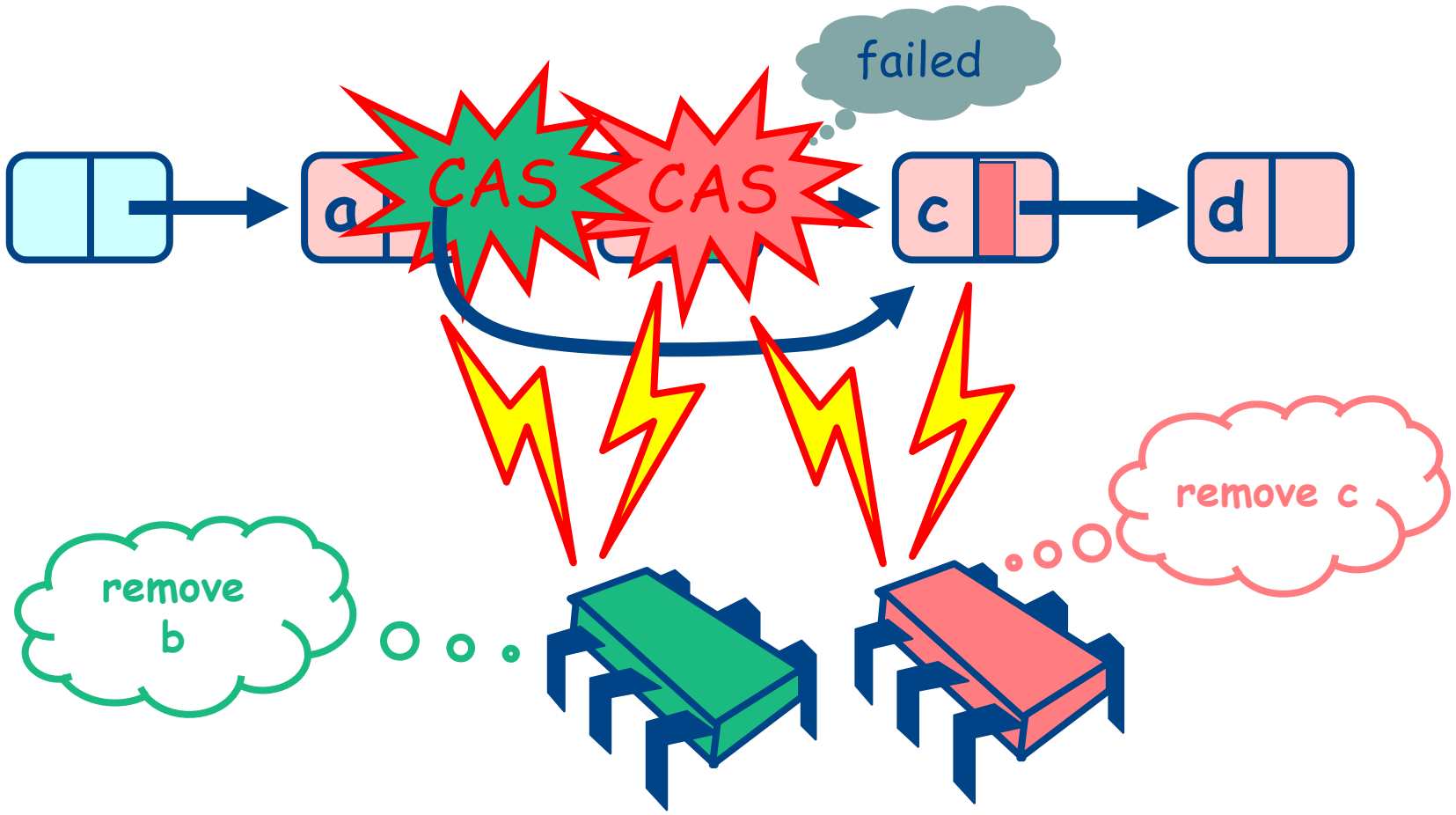
```
public boolean attemptMark(
  Object expectedRef,
  boolean updateMark);
```

.. then change to this new mark.

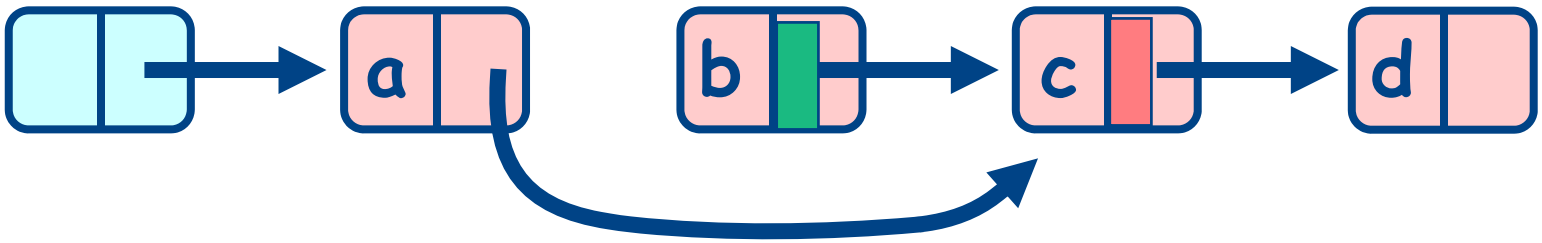
Removing a Node

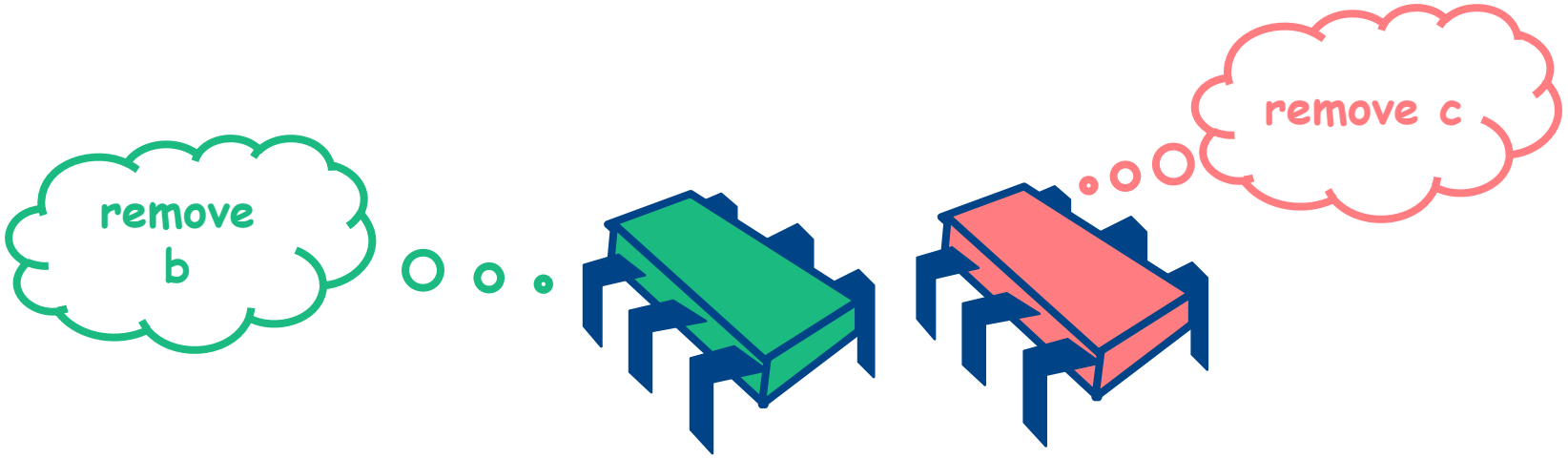


Removing a Node

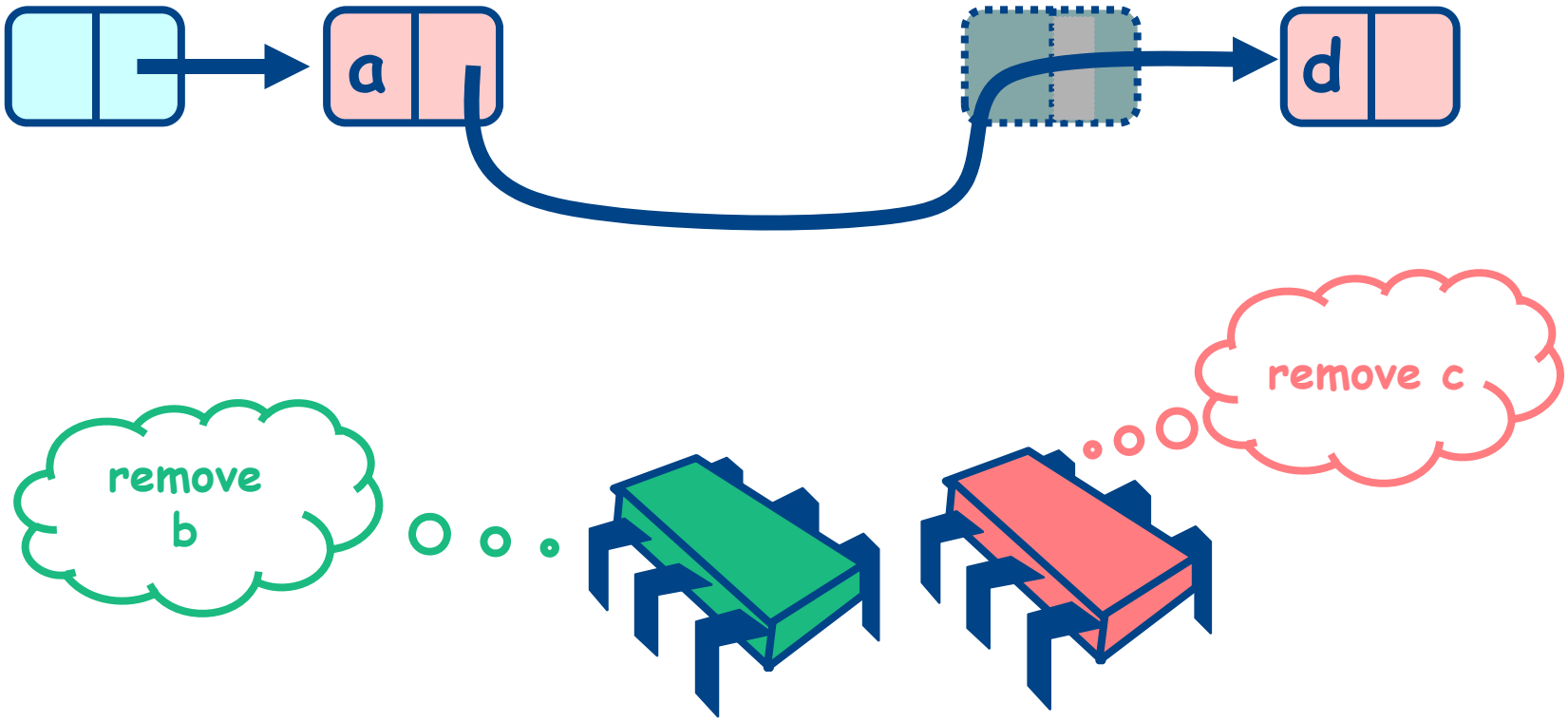


Removing a Node





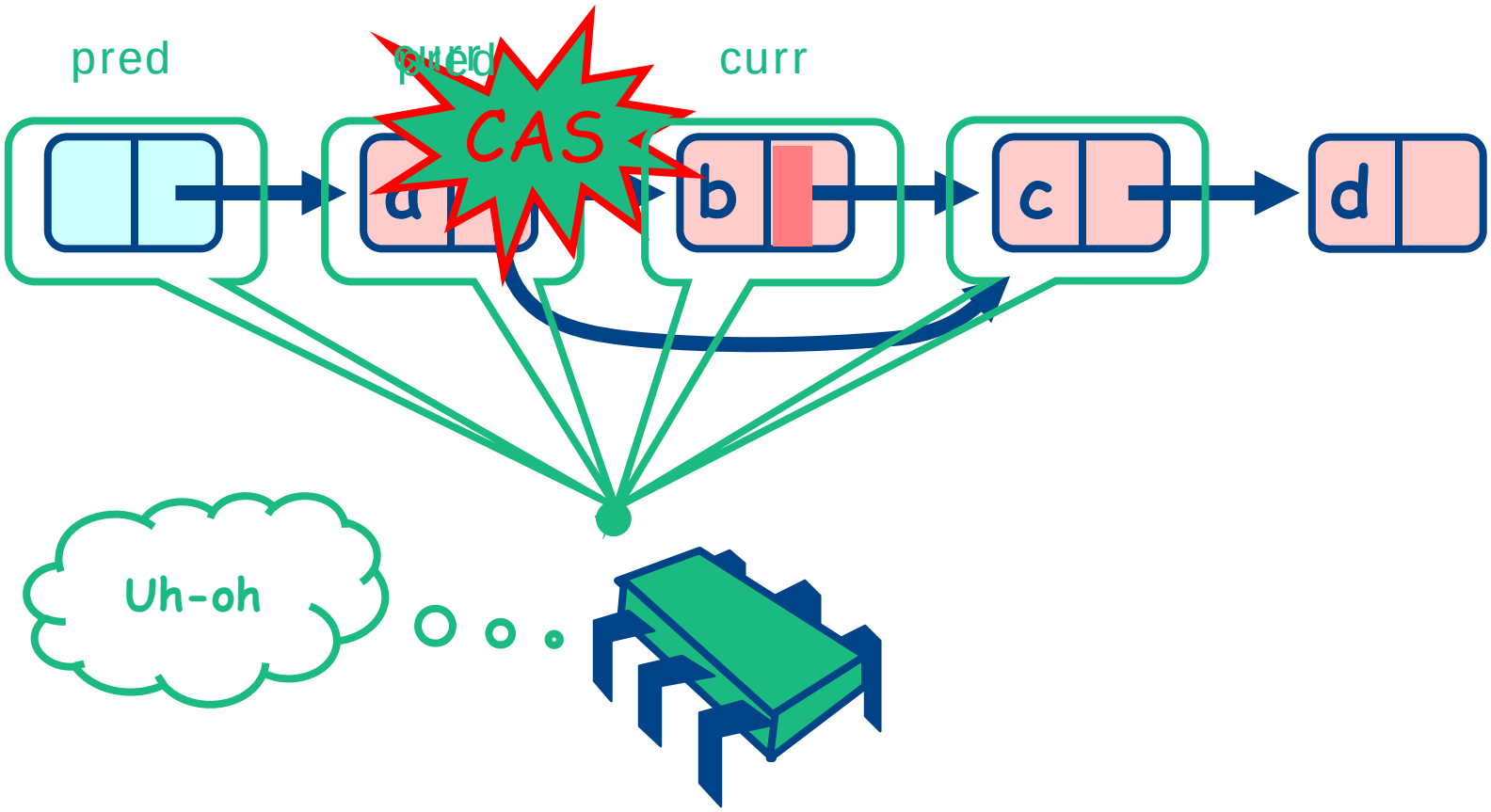
Removing a Node



Traversing the List

- **Q: what do you do when you find a "logically" deleted node in your path?**
- **A: finish the job.**
  - CAS the predecessor's next field
  - Proceed (repeat as needed)

Lock-Free Traversal (only Add and Remove)



Node and window

```
private class Node {
  T item;
  int key;
  AtomicMarkableReference<Node> next;
  Node(T item) {       // usual constructor
    this.item = item;
    this.key = item.hashCode();
    this.next = new AtomicMarkableReference<Node>(null, false);
  }
  Node(int key) { // sentinel constructor
    this.item = null;
    this.key = key;
    this.next = new AtomicMarkableReference<Node>(null, false);
  }
}

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

find

```
public Window find(Node head, int key) {
  Node pred = null, curr = null, succ = null;
  boolean[] marked = {false}; // is curr marked?
  boolean snip;
  retry: while (true) {
    pred = head;
    curr = pred.next.getReference();
    while (true) {
      succ = curr.next.get(marked);
      while (marked[0]) {           // replace curr if marked
        snip = pred.next.compareAndSet(curr, succ, false, false);
        if (!snip) continue retry;
        curr = pred.next.getReference();
        succ = curr.next.get(marked);
      }
      if (curr.key >= key)
        return new Window(pred, curr);
      pred = curr;
      curr = succ;
    }
  }
}
```

remove

```
public boolean remove(T item) {
  int key = item.hashCode();
  boolean snip;
  while (true) {
    // find predecessor and curren entries
    Window window = find(head, key);
    Node pred = window.pred, curr = window.curr;
    // is the key present?
    if (curr.key != key) {
      return false;
    } else {
      // snip out matching node
      Node succ = curr.next.getReference();
      snip = curr.next.attemptMark(succ, true);
      if (!snip)
        continue;
      pred.next.compareAndSet(curr, succ, false, false);
      return true;
    }
  }
}
```

add

```
public boolean add(T item) {
  int key = item.hashCode();
  boolean splice;
  while (true) {
    // find predecessor and curren entries
    Window window = find(head, key);
    Node pred = window.pred, curr = window.curr;
    // is the key present?
    if (curr.key == key) {
      return false;
    } else {
      // splice in new node
      Node node = new Node(item);
      node.next = new AtomicMarkableReference(curr, false);
      if (pred.next.compareAndSet(curr, node, false, false)) {
        return true;
      }
    }
  }
}
```

Iteration

```
final class Itr implements Iterator<E> {
    private Node nextNode;   // next node to return item for
    private E nextItem;      // the corresponding item
    private Node lastRet;    // last returned node, to support remove
    private Node lastPred;   // predecessor to unlink lastRet

    Itr() {
        advance(null);
    }

    public final boolean hasNext() {
        return nextNode != null;
    }

    public final E next() {
        Node p = nextNode;
        if (p == null) throw new NoSuchElementException();
        E e = nextItem;
        advance(p);
        return e;
    }

    public final void remove() {
        Node p = lastRet;
        if (p == null) throw new IllegalStateException();
        if (p.tryMatchData())
            unsplice(lastPred, p);
    }
```

Iteration

```
    private void advance(Node prev) {
        lastPred = lastRet;
        lastRet = prev;
        for (Node p = (prev == null) ? head : succ(prev);
             p != null; p = succ(p)) {
            Object item = p.item;
            if (p.isData) {
                if (item != null && item != p) {
                    nextItem = LinkedTransferQueue.this.<E>cast(item);
                    nextNode = p;
                    return;
                }
            }
            else if (item == null)
                break;
        }
        nextNode = null;
    }
}
```

Queues

- **All concurrent queues are implementation of this technique.**
- **JDK 7 version add more optimizations like**
  - Slack.
  - Lazy writes.

List

- There are no concurrent list implementations????

LOGO

Concurrent Skiplist

Set Object Interface

- **Collection of elements**
- **No duplicates**
- **Methods**
  - add() a new element
  - remove() an element
  - contains() if element is present

Many are Cold but Few are

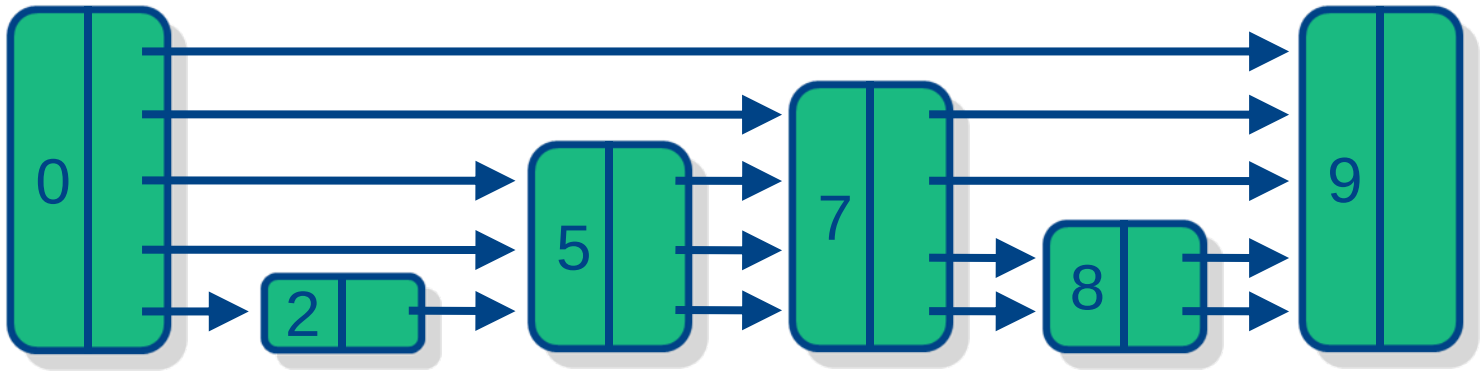
- **Typically high % of contains() calls**
- **Many fewer add() calls**
- **And even fewer remove() calls**
  - 90% contains()
  - 9% add()
  - 1% remove()
- **Folklore?**
  - Yes but probably mostly true

Concurrent Sets

- Balanced Trees?
  - Red-Black trees, AVL trees, …
- Problem: no one does this well …
- … because rebalancing after add() or remove() is a global operation

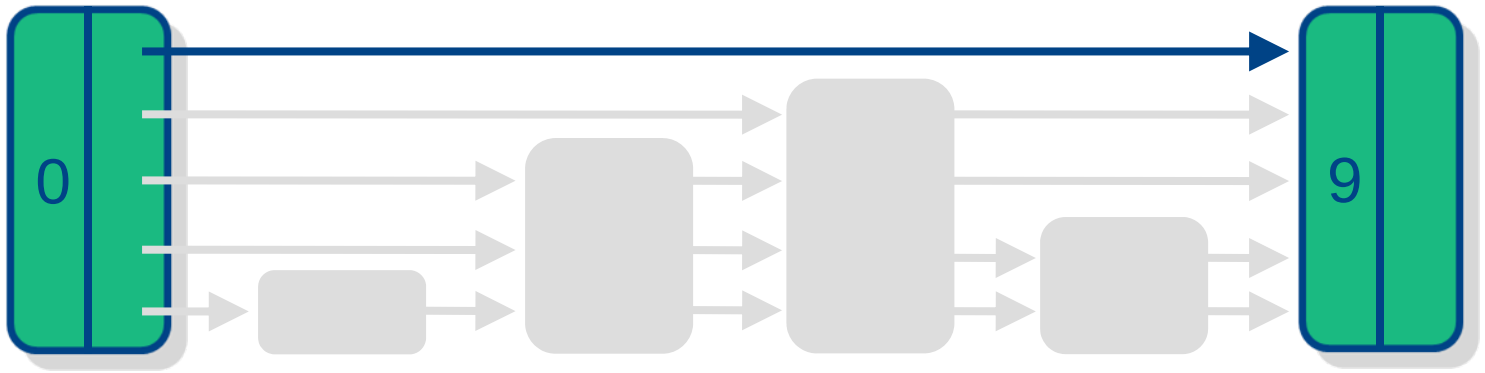
Skip Lists

- Probabilistic Data Structure
- No global rebalancing
- Logarithmic-time search



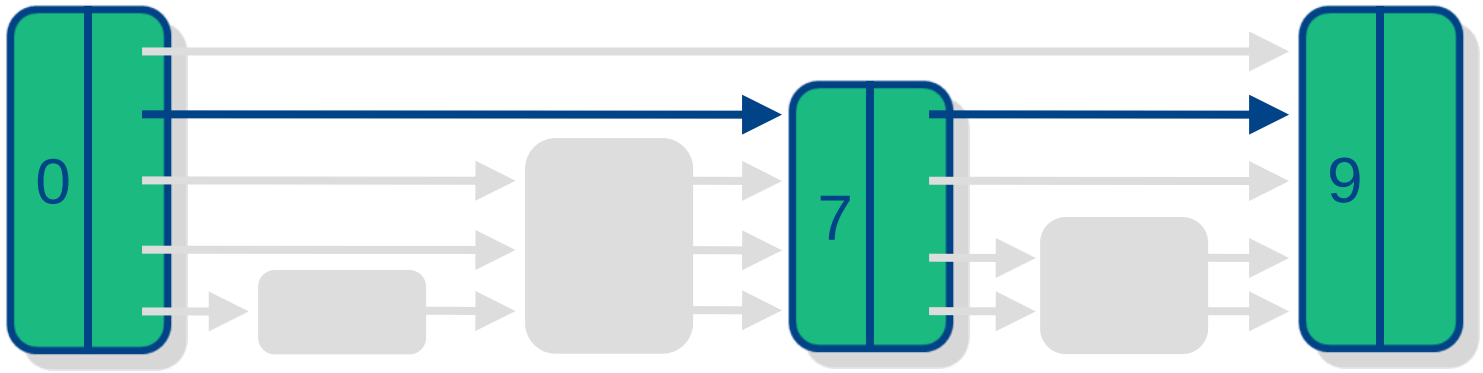
Skip List Property

- Each layer is sublist of lower-levels



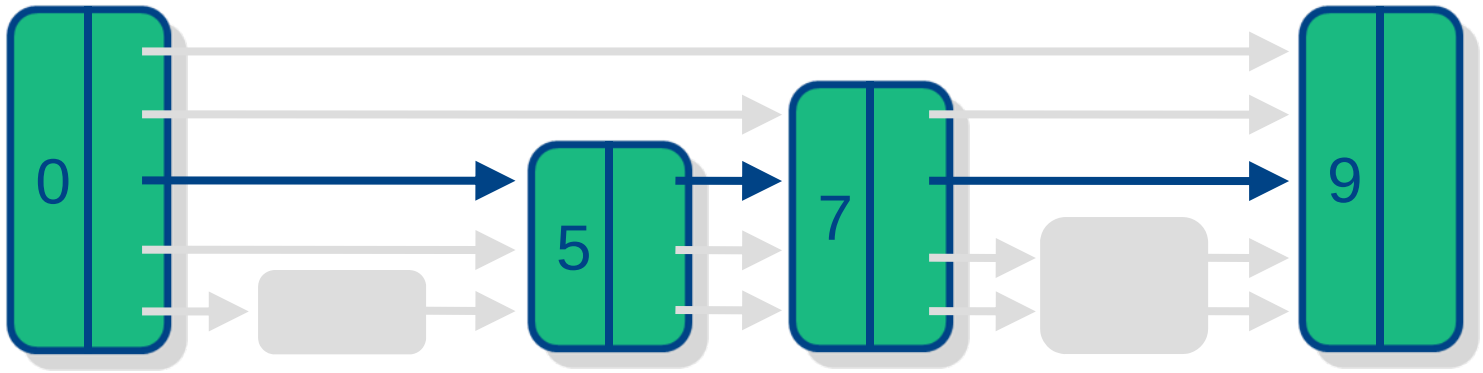
Skip List Property

- Each layer is sublist of lower-levels



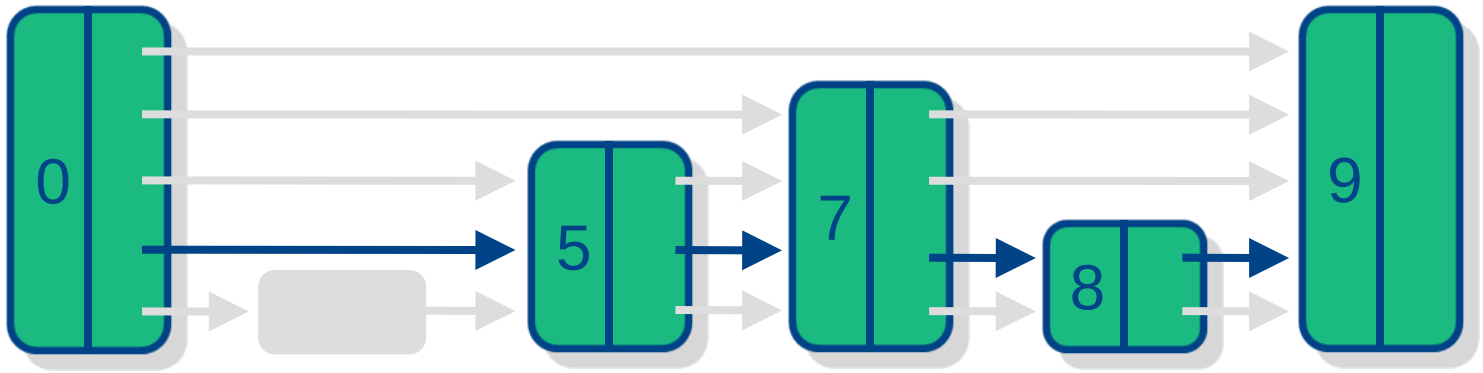
Skip List Property

- Each layer is sublist of lower-levels



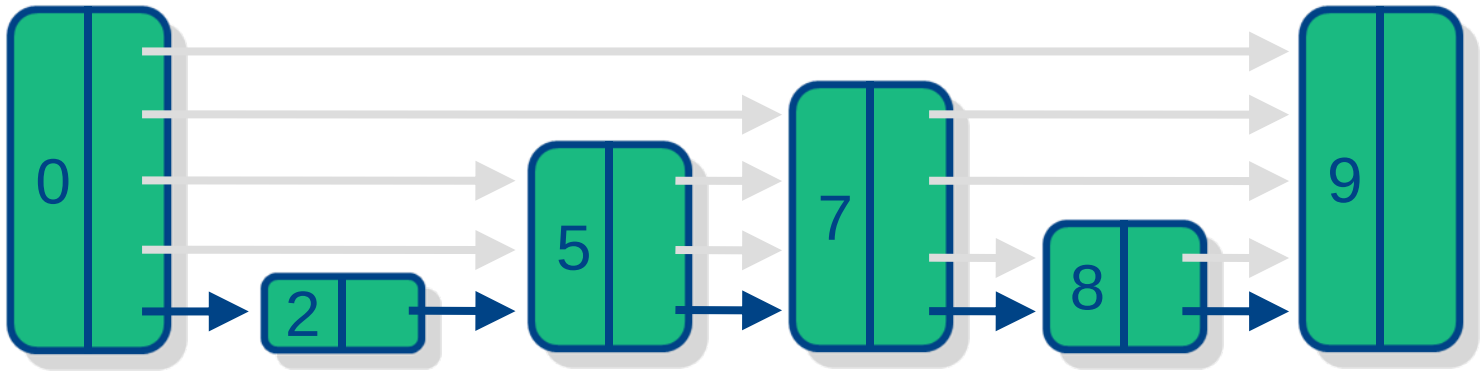
Skip List Property

- Each layer is sublist of lower-levels



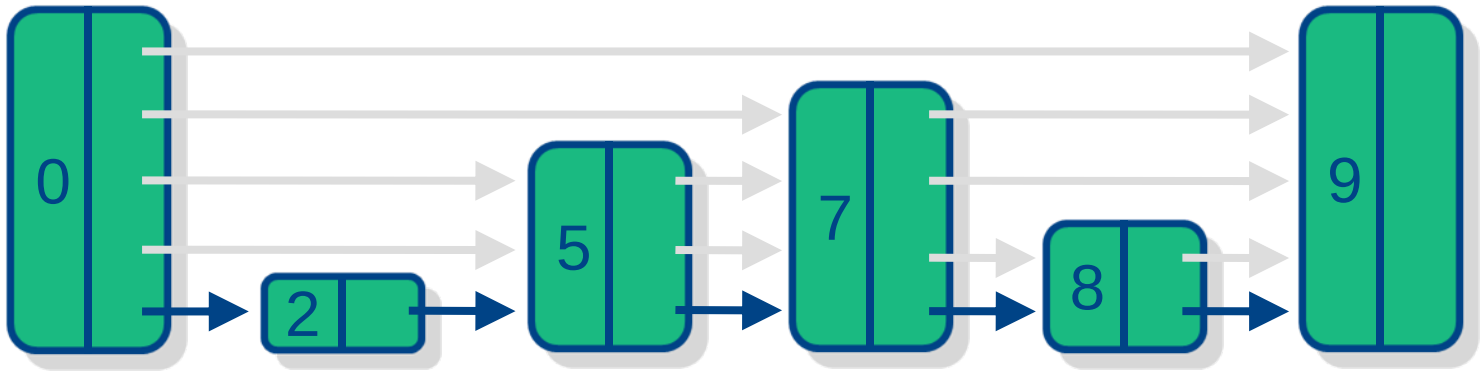
Skip List Property

- Each layer is sublist of lower-levels
- Lowest level is entire list

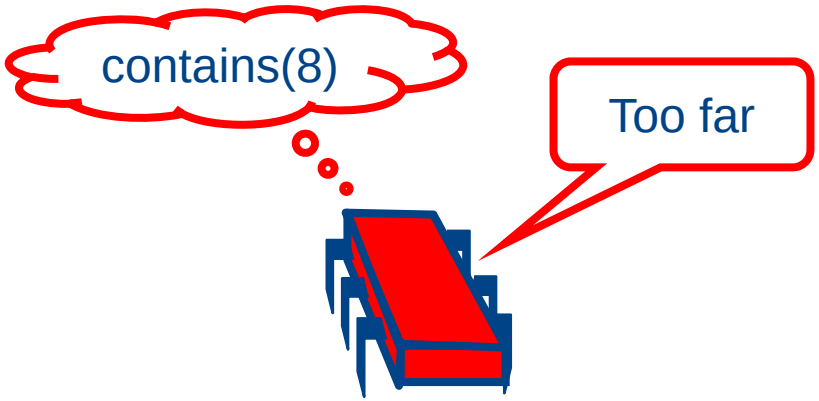


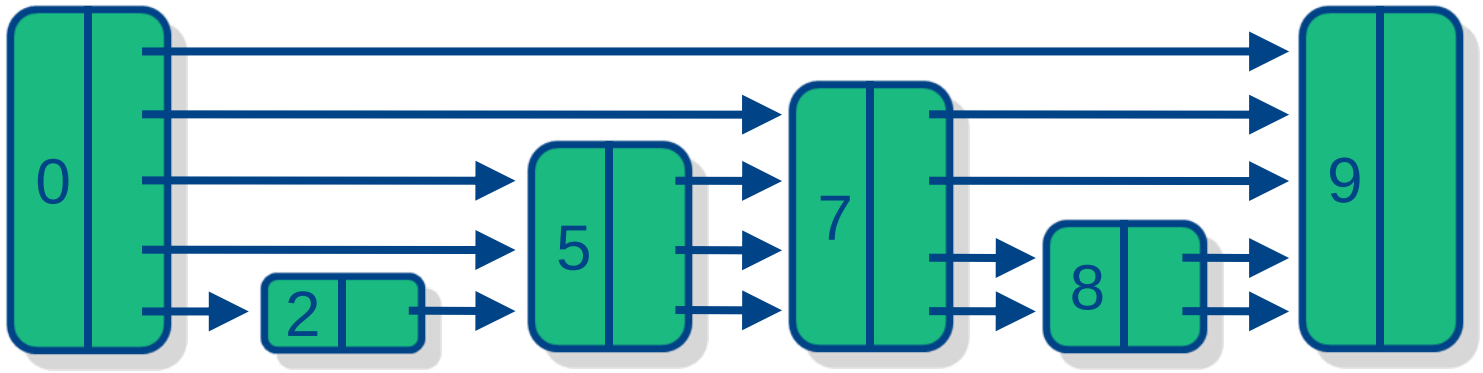
Skip List Property

- Each layer is sublist of lower-levels
- Not easy to preserve in concurrent implementations …

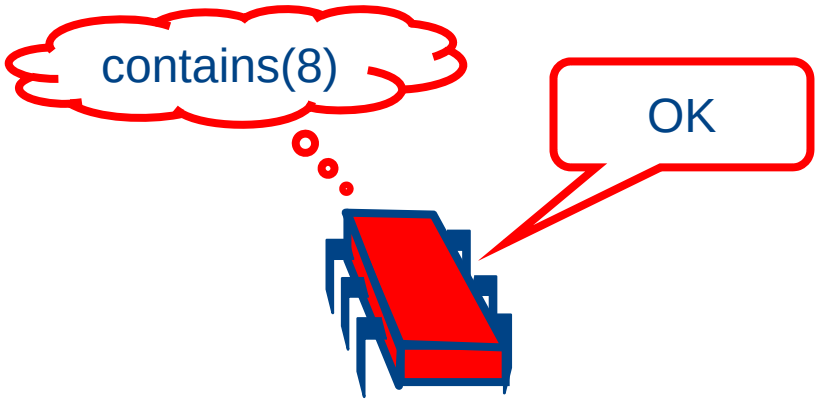


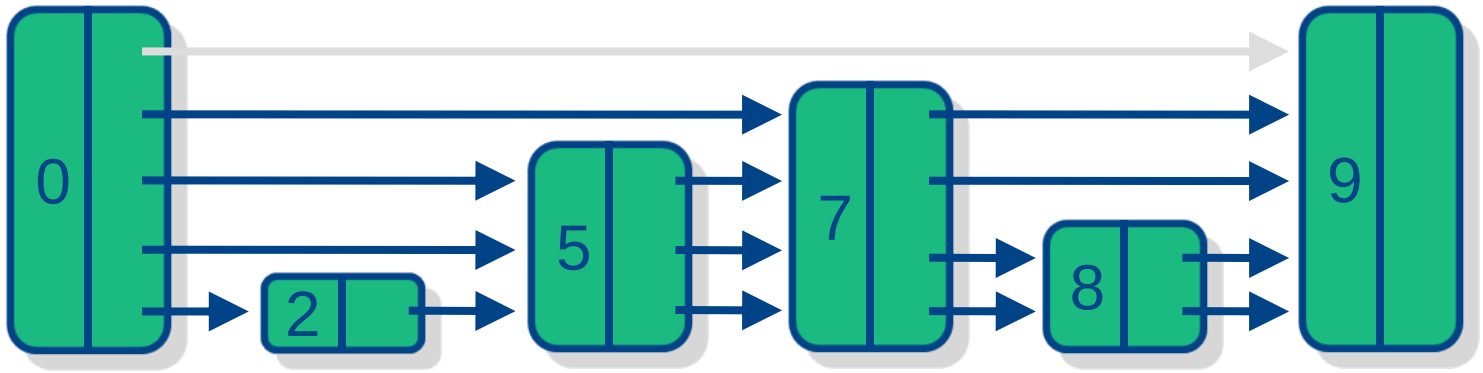
Search



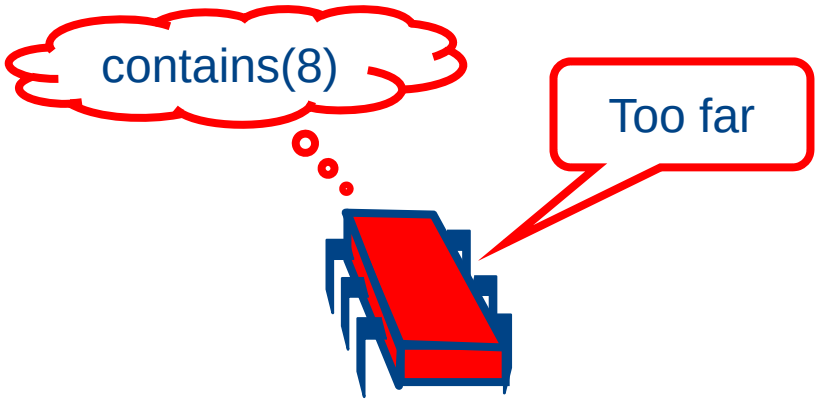


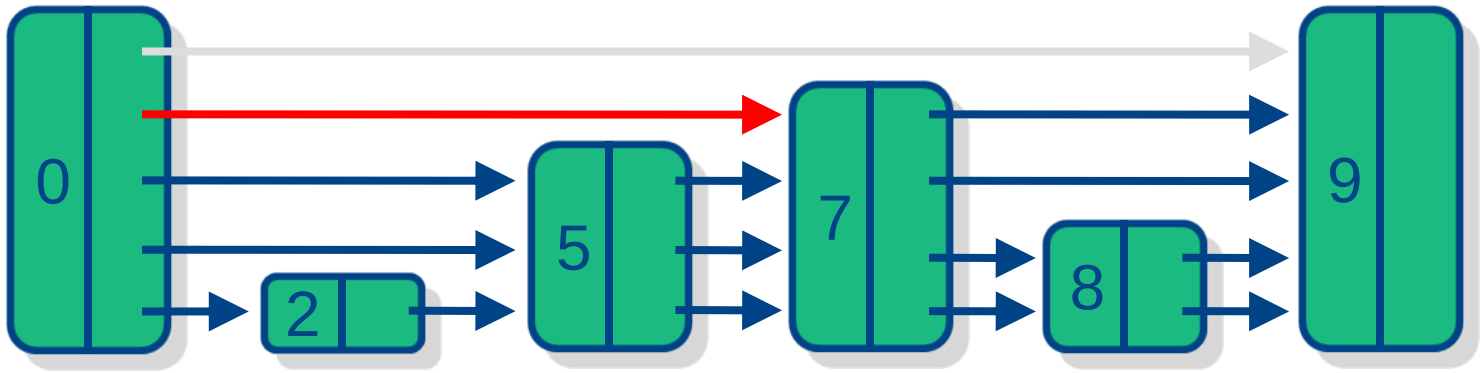
Search



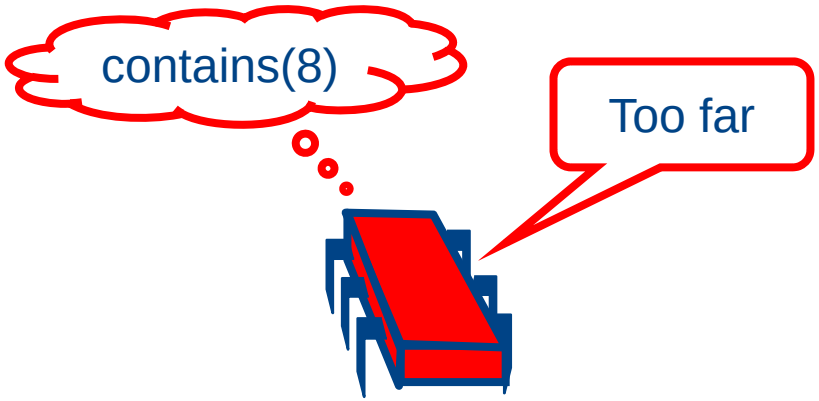


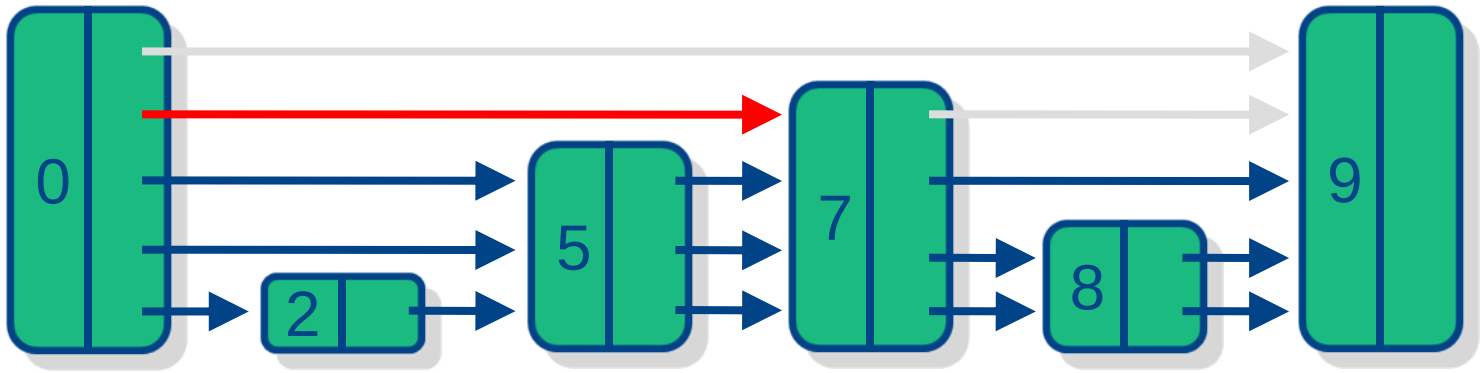
Search



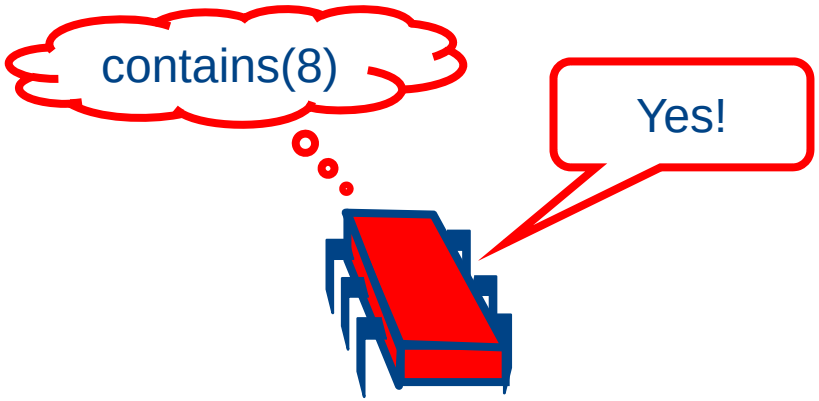


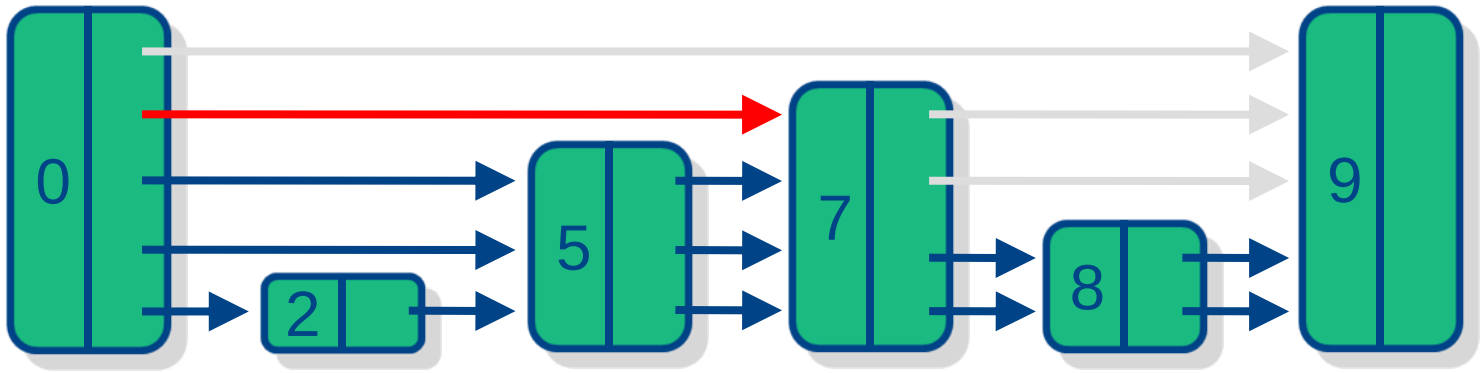
Search



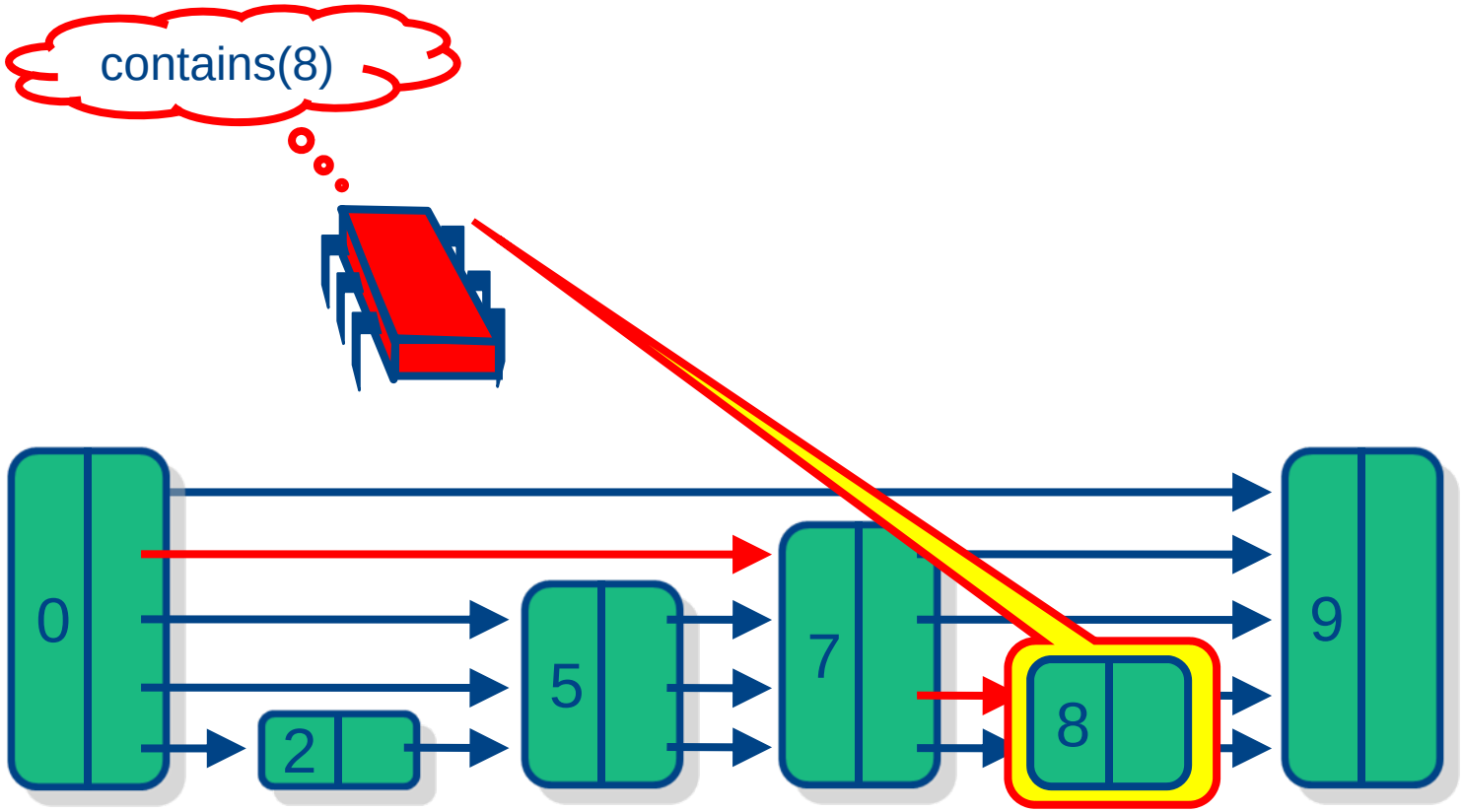


Search

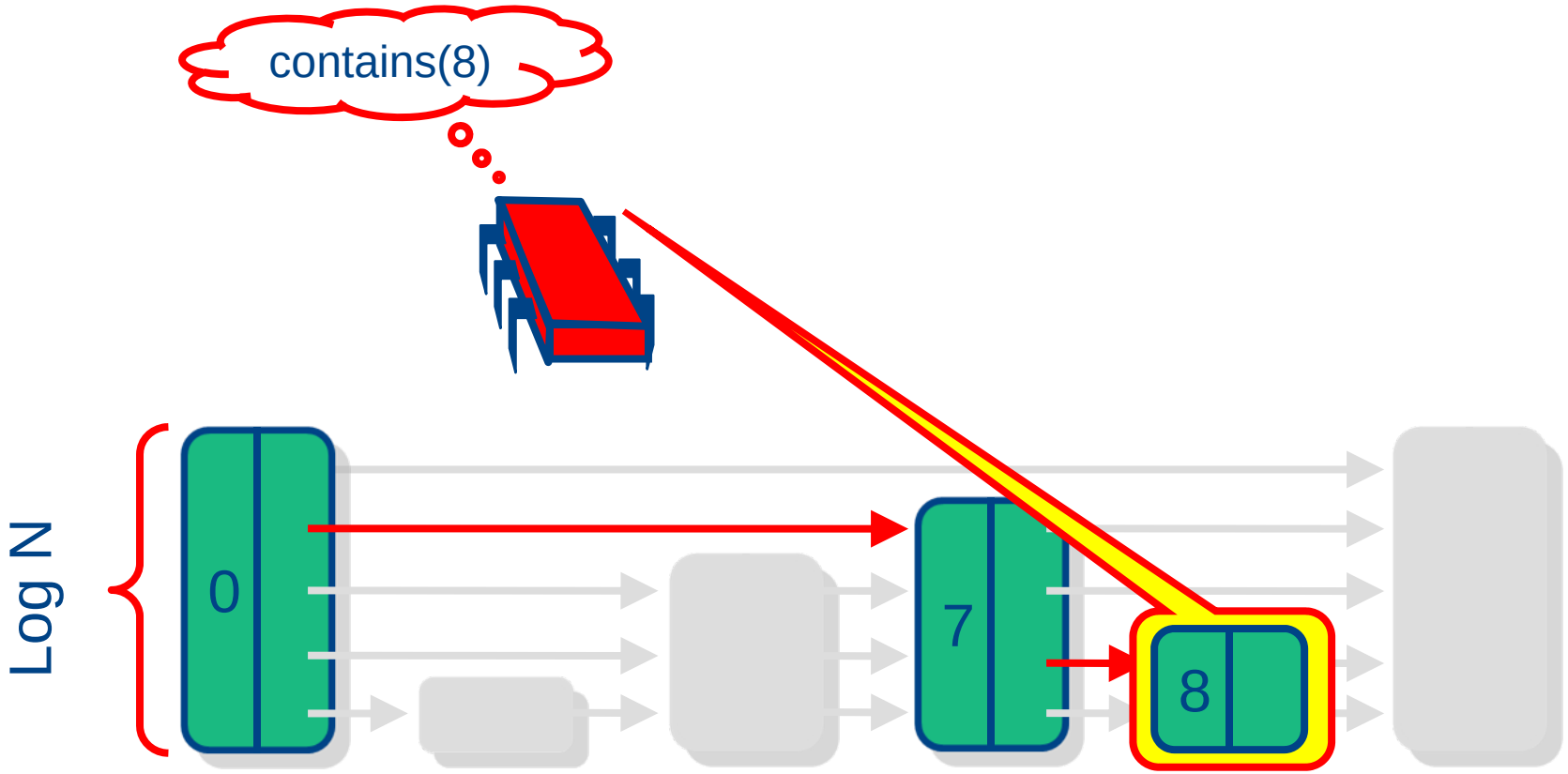




Search

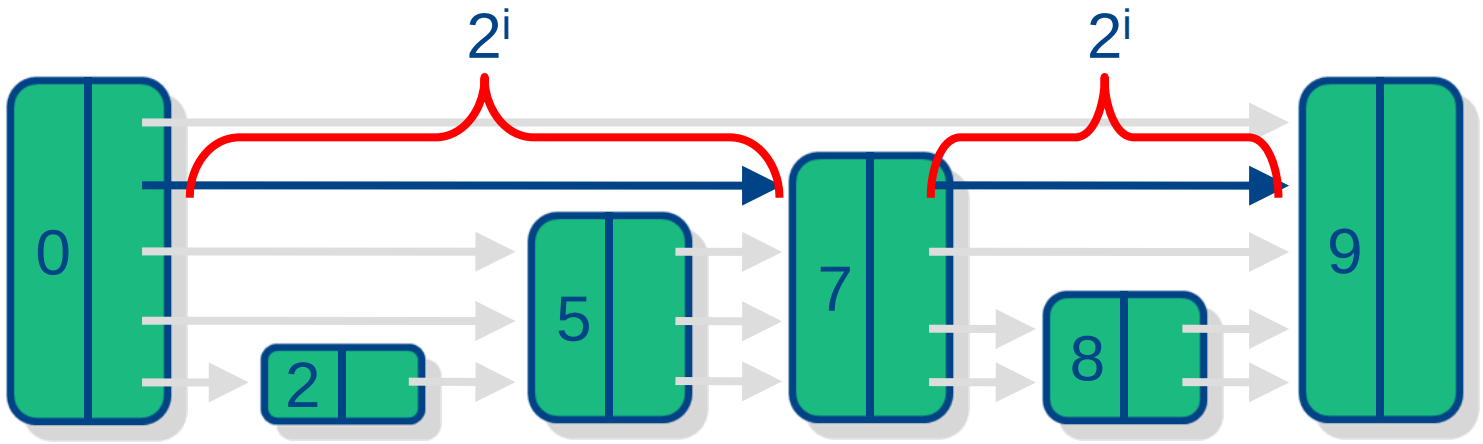


Logarithmic



Why Logarthimic

- Property: Each pointer at layer i jumps over roughly 2^i nodes
- Pick node heights randomly so property guaranteed probabilistically



Find() -- Sequential

```
int find(T x, Node<T>[] preds, Node<T>[] succs) {
  …
  }
```

Find() -- Sequential

```
int find(T x, Node<T>[] preds, Node<T>[] succs) {
  …
}
```

object height
(-1 if not there)

Find() -- Sequential

```
int find(T x, Node<T>[] preds, Node<T>[] succs) {
  ...
}
```

object height
(-1 if not there)

Object sought

Find() -- Sequential

```
int find(T x, Node<T>[] preds, Node<T>[] succs) {
  ...
}
```

Object height
(-1 if not there)

object sought

return predecessors

Find() -- Sequential

```
int find(T x, Node<T>[] preds, Node<T>[] succs) {
  ...
}
```

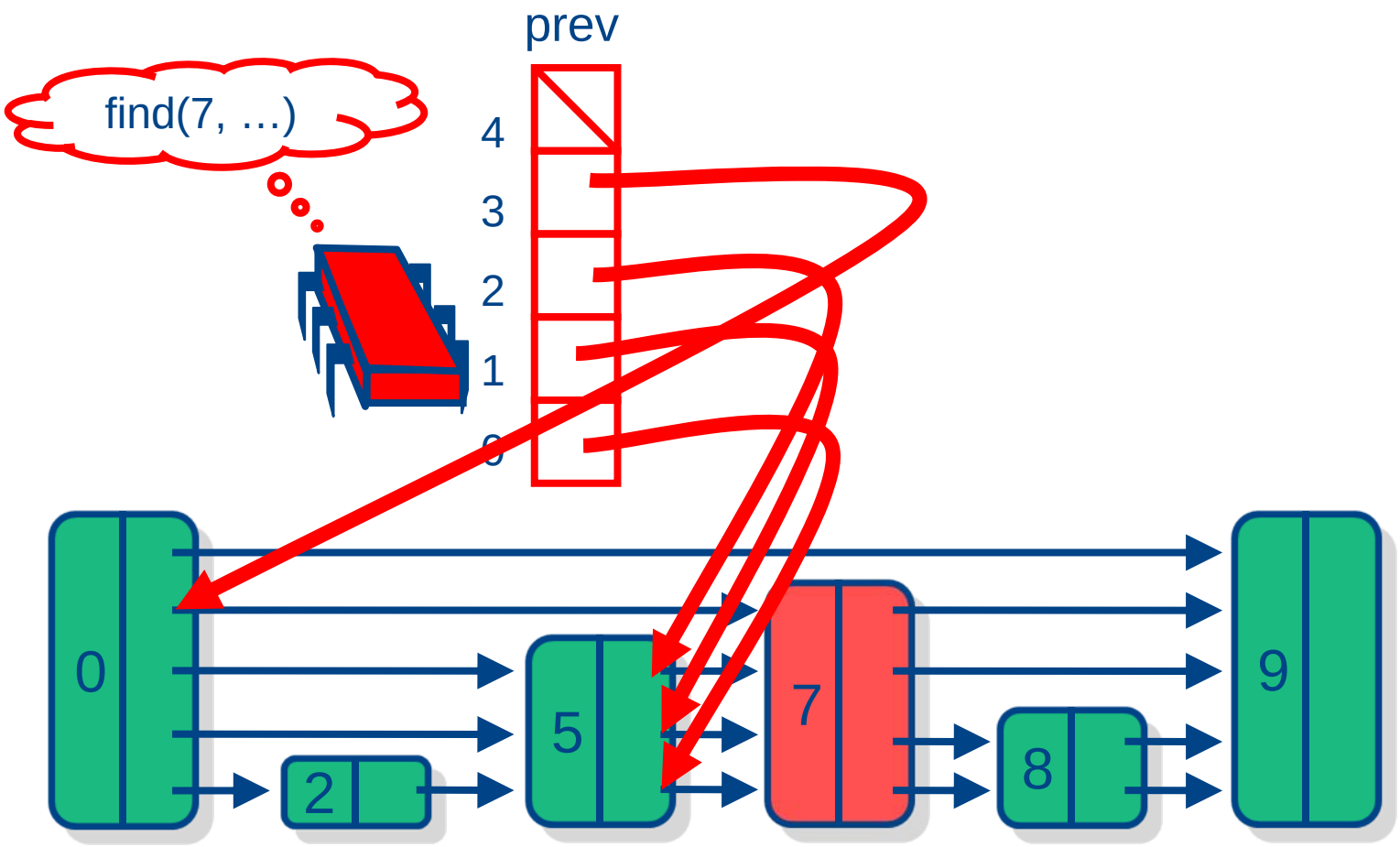
object height
(-1 if not there)

object sought

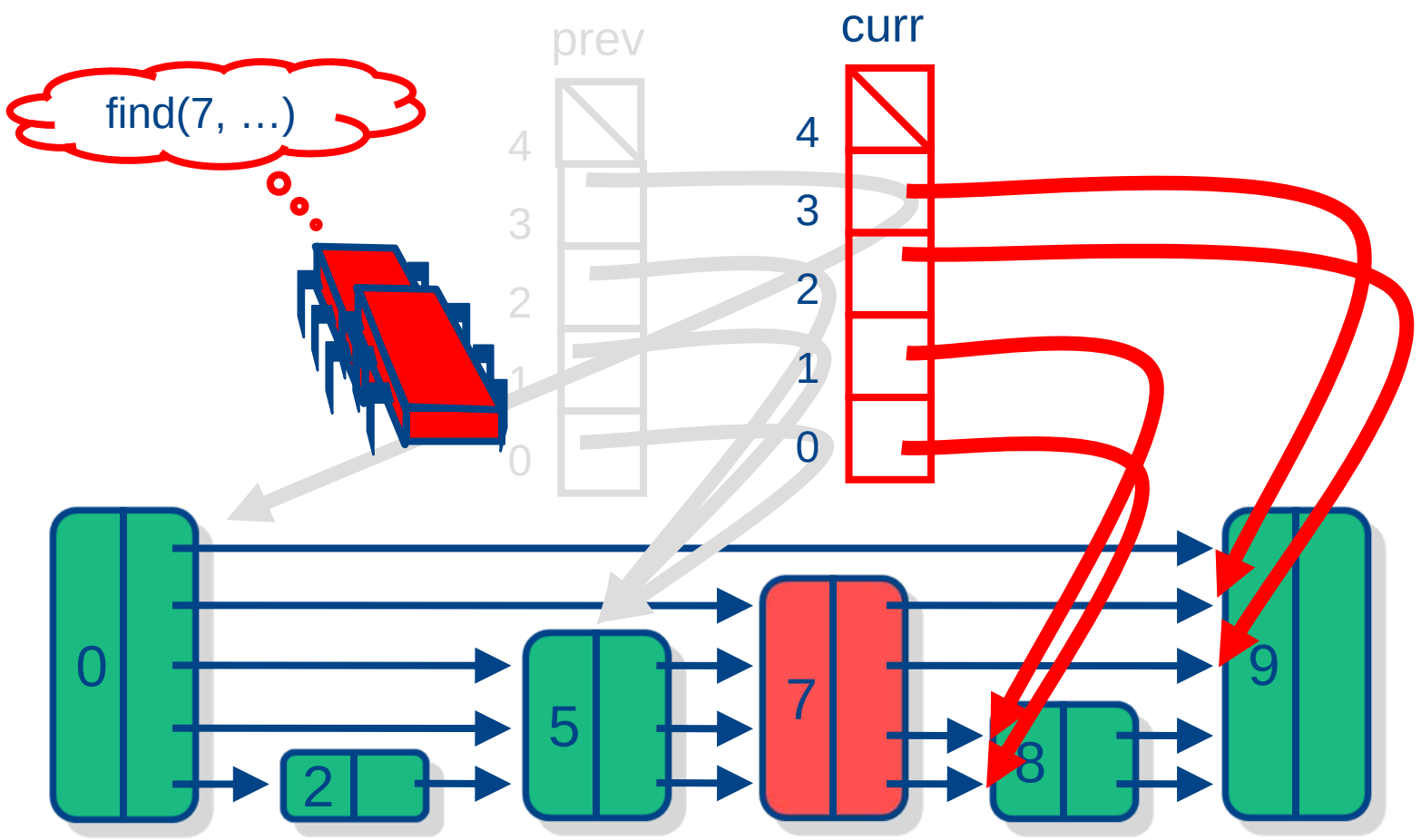
return predecessors

return successors

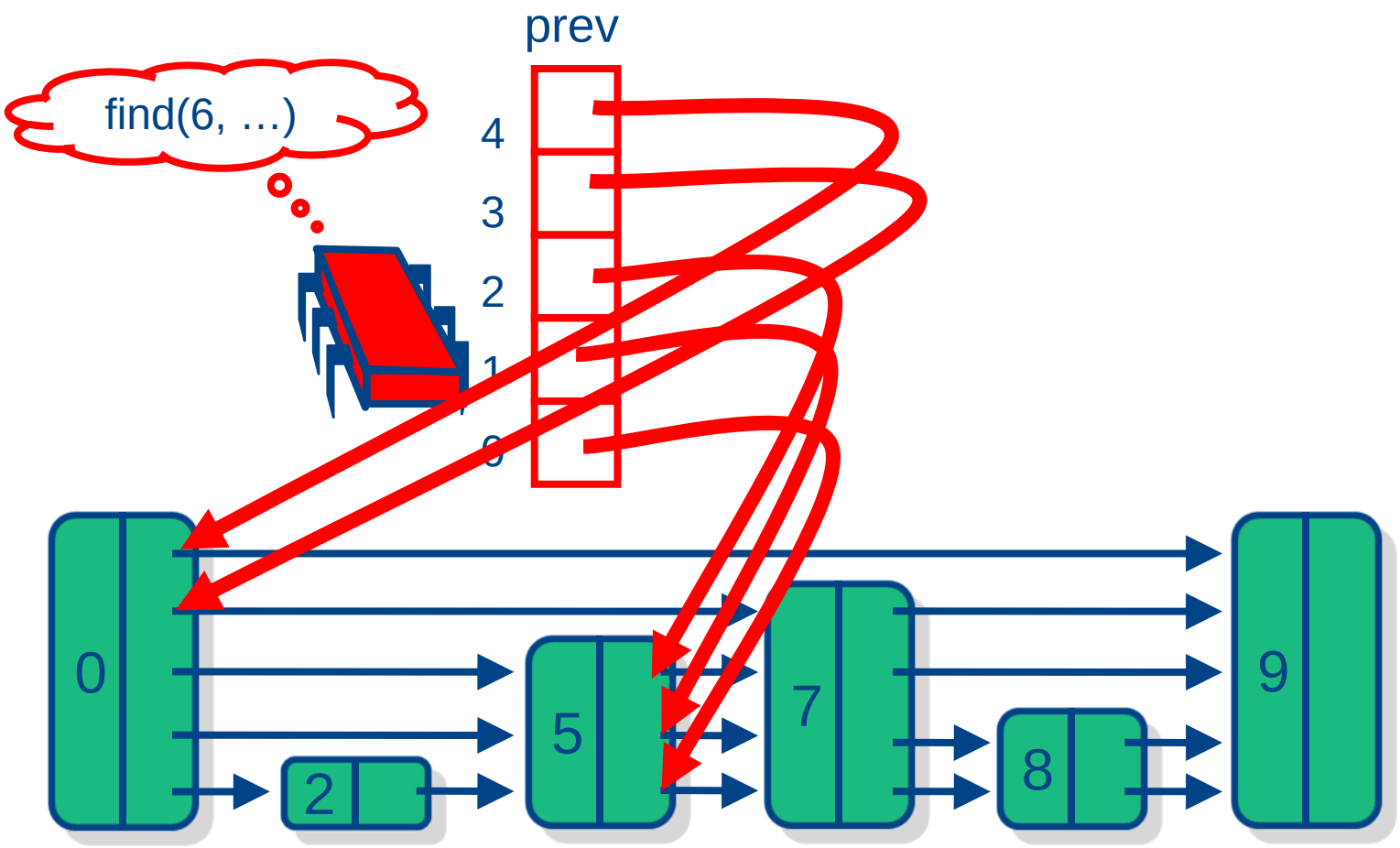
Successful Search



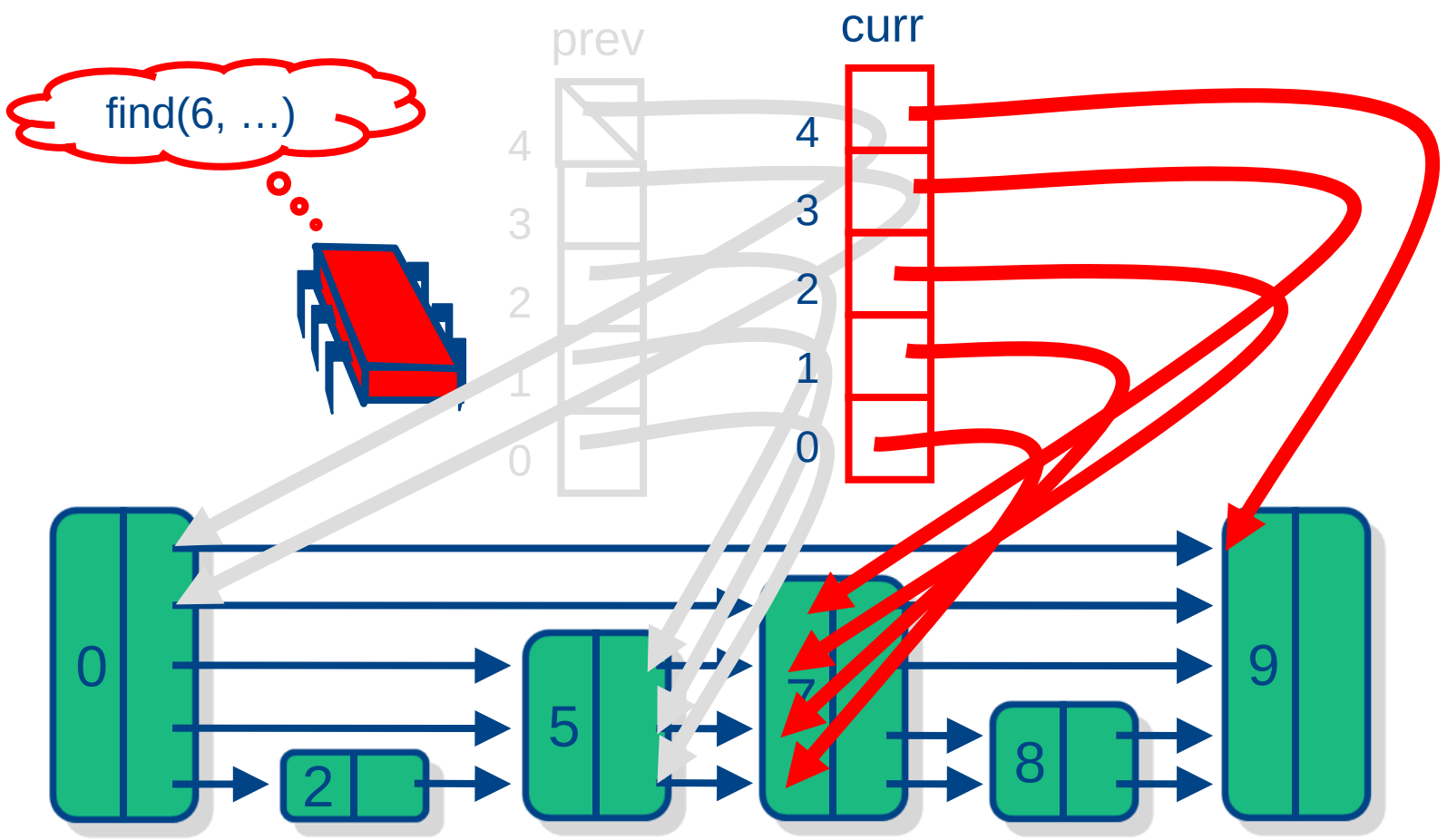
Successful Search



Unsuccessful Search



Unsuccessful Search



LockFreeSkipList

```
public final class LockFreeSkipList<T> {
  static final int MAX_LEVEL = ...;
  final Node<T> head = new Node<T>(Integer.MIN_VALUE);
  final Node<T> tail = new Node<T>(Integer.MAX_VALUE);
  public LockFreeSkipList() {
    for (int i = 0; i < head.next.length; i++) {
      head.next[i]
      = new AtomicMarkableReference<LockFreeSkipList.Node<T>>(tail, false);
    }
  }
```

LockFreeSkipList

```
public static final class Node<T> {
  final T value; final int key;
  final AtomicMarkableReference<Node<T>>[] next;
  private int topLevel;
  // constructor for sentinel nodes
  public Node(int key) {
    value = null; key = key;
    next = (AtomicMarkableReference<Node<T>>[])
     new AtomicMarkableReference[MAX_LEVEL + 1];
    for (int i = 0; i < next.length; i++) {
      next[i] = new AtomicMarkableReference<Node<T>>(null,false);
    }
    topLevel = MAX_LEVEL;
  }
  // constructor for ordinary nodes
  public Node(T x, int height) {
    value = x;
    key = x.hashCode();
    next = (AtomicMarkableReference<Node<T>>[])
     new AtomicMarkableReference[height + 1];
    for (int i = 0; i < next.length; i++) {
      next[i] = new AtomicMarkableReference<Node<T>>(null,false);
    }
    topLevel = height;
  }
}
```

LockFreeSkipList

```
boolean add(T x) {
  int topLevel = randomLevel();
  int bottomLevel = 0;
  Node<T>[] preds = (Node<T>[]) new Node[MAX_LEVEL + 1];
  Node<T>[] succs = (Node<T>[]) new Node[MAX_LEVEL + 1];
  while (true) {
    boolean found = find(x, preds, succs);
    if (found) {
      return false;
    } else {
      Node<T> newNode = new Node(x, topLevel);
      for (int level = bottomLevel; level <= topLevel; level++) {
        Node<T> succ = succs[level];
        newNode.next[level].set(succ, false);
      }
      Node<T> pred = preds[bottomLevel];
      Node<T> succ = succs[bottomLevel];
      newNode.next[bottomLevel].set(succ, false);
      if (!pred.next[bottomLevel].compareAndSet(succ, newNode,
                                                false, false)) {
        continue;
      }
      for (int level = bottomLevel+1; level <= topLevel; level++) {
        while (true) {
          pred = preds[level];
          succ = succs[level];
          if (pred.next[level].compareAndSet(succ, newNode, false, false))
            break;
          find(x, preds, succs);
        }
      }
      return true;
    }
  }
}
```

LockFreeSkipList

```
boolean remove(T x) {
  int bottomLevel = 0;
  Node<T>[] preds = (Node<T>[]) new Node[MAX_LEVEL + 1];
  Node<T>[] succs = (Node<T>[]) new Node[MAX_LEVEL + 1];
  Node<T> succ;
  while (true) {
    boolean found = find(x, preds, succs);
    if (!found) {
      return false;
    } else {
      Node<T> nodeToRemove = succs[bottomLevel];
      for (int level = nodeToRemove.topLevel;
           level >= bottomLevel+1; level--) {
        boolean[] marked = {false};
        succ = nodeToRemove.next[level].get(marked);
        while (!marked[0]) {
          nodeToRemove.next[level].attemptMark(succ, true);
        succ = nodeToRemove.next[level].get(marked);
        }
      }
      boolean[] marked = {false};
      succ = nodeToRemove.next[bottomLevel].get(marked);
      while (true) {
        boolean iMarkedIt =
          nodeToRemove.next[bottomLevel].compareAndSet(succ, succ,
                                                       false, true);
        succ = succs[bottomLevel].next[bottomLevel].get(marked);
        if (iMarkedIt) {
           find(x, preds, succs);
           return true;
        }
        else if (marked[0]) return false;
      }
    }
  }
}
```

LockFreeSkipList

```
boolean find(T x, Node<T>[] preds, Node<T>[] succs) {
  int bottomLevel = 0;
  int key = x.hashCode();
  boolean[] marked = {false};
  boolean snip;
  Node<T> pred = null, curr = null, succ = null;
  retry:
    while (true) {
      pred = head;
      for (int level = MAX_LEVEL; level >= bottomLevel; level--) {
        curr = pred.next[level].getReference();
        while (true) {
          succ = curr.next[level].get(marked);
          while (marked[0]) {
            snip = pred.next[level].compareAndSet(curr, succ,
                                                  false, false);
            if (!snip) continue retry;
            curr = pred.next[level].getReference();
            succ = curr.next[level].get(marked);
          }
          if (curr.key < key){
            pred = curr; curr = succ;
          } else {
            break;
          }
        }
        preds[level] = pred;
        succs[level] = curr;
      }
      return (curr.key == key);
    }
}
```

LOGO

Priority Queue

Priority Queue

```
public final class PrioritySkipList<T> {
  public static final class Node<T> {
    final T item;
    final int score;
    AtomicBoolean marked;
    final AtomicMarkableReference<Node<T>>[] next;
    // sentinel node constructor
    public Node(int myPriority) { ... }
    // ordinary node constructor
    public Node(T x, int myPriority) { ... }
  }
  boolean add(Node node) { ... }
  boolean remove(Node<T> node) { ... }
  public Node<T> findAndMarkMin() {
    Node<T> curr = null, succ = null;
    curr = head.next[0].getReference();
    while (curr != tail) {
      if (!curr.marked.get()) {
        if (curr.marked.compareAndSet(false, true)) {
          return curr;
        } else {
          curr = curr.next[0].getReference();
        }
      }
    }
    return null; // no unmarked nodes
  }
```

Priority Queue

```
public class SkipQueue<T> {
  PrioritySkipList<T> skiplist;
  public SkipQueue() {
    skiplist = new PrioritySkipList<T>();
  }
  public boolean add(T item, int score) {
    Node<T> node = (Node<T>)new Node(item, score);
    return skiplist.add(node);
  }
  public T removeMin() {
    Node<T> node = skiplist.findAndMarkMin();
    if (node != null) {
      skiplist.remove(node);
      return node.item;
    } else{
      return null;
    }
  }
}
```

LOGO

NonBlockingHashmap

NonBlockingHashmap

- Array of K/V Pairs
  - Keys in even slots, Values odd slots
  - CAS each word separately, but FSM spans both words
  - Value can also be 'Tombstone'
  - Key & Value both start as **null**
- Mark payload by 'boxing' values
- Copy on resize, or to flush stale keys
- Supports concurrent insert, remove, test, resize
- Linear scaling on Azul to 768 CPUs
  - More than billion reads/sec simultaneous with
  - More than 10million updates/sec
- Code up in SourceForge, high-scale-lib
  - Passes Java Compatibility Kit (JCK) for ConcurrentHashMap

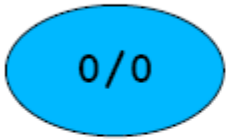
NonBlockingHashmap

"Uninteresting" Details

- Good, standard engineering – nothing special
- Closed Power-of-2 Hash Table
  - Reprobe on collision
  - Stride-1 reprobe: better cache behavior
  - (complicated argument about 2^n vs prime goes here)
- Key & Value on same cache line
- Hash memoized
  - Should be same cache line as K + V
  - But hard to do in pure Java
- No allocation on get() or put()
- Auto-Resize

NonBlockingHashmap

HashTable State Machine

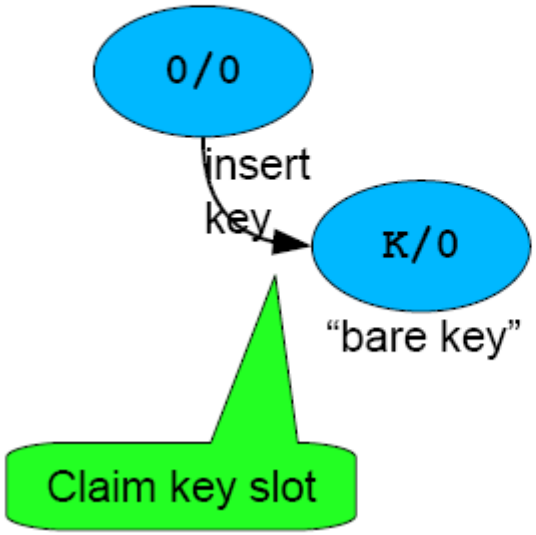


"initial"

- Inserting K/V pair
- Already probed table, missed
- Found proper empty K/V slot
- Ready to claim slot for this Key

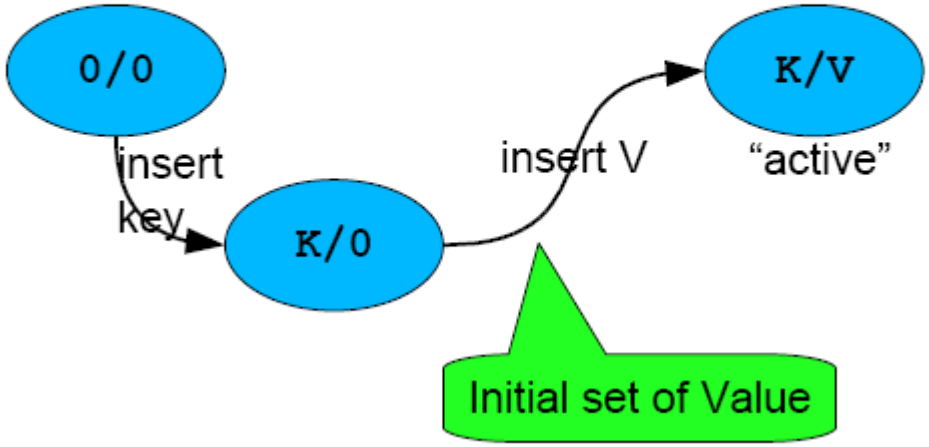
NonBlockingHashmap

HashTable State Machine



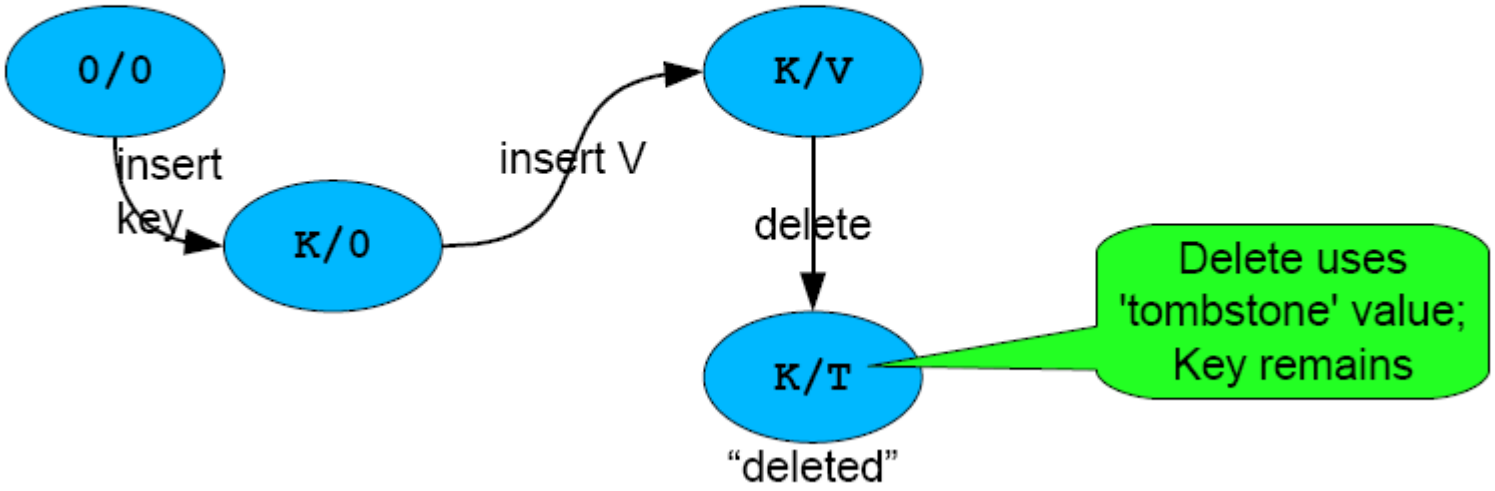
NonBlockingHashmap

HashTable State Machine



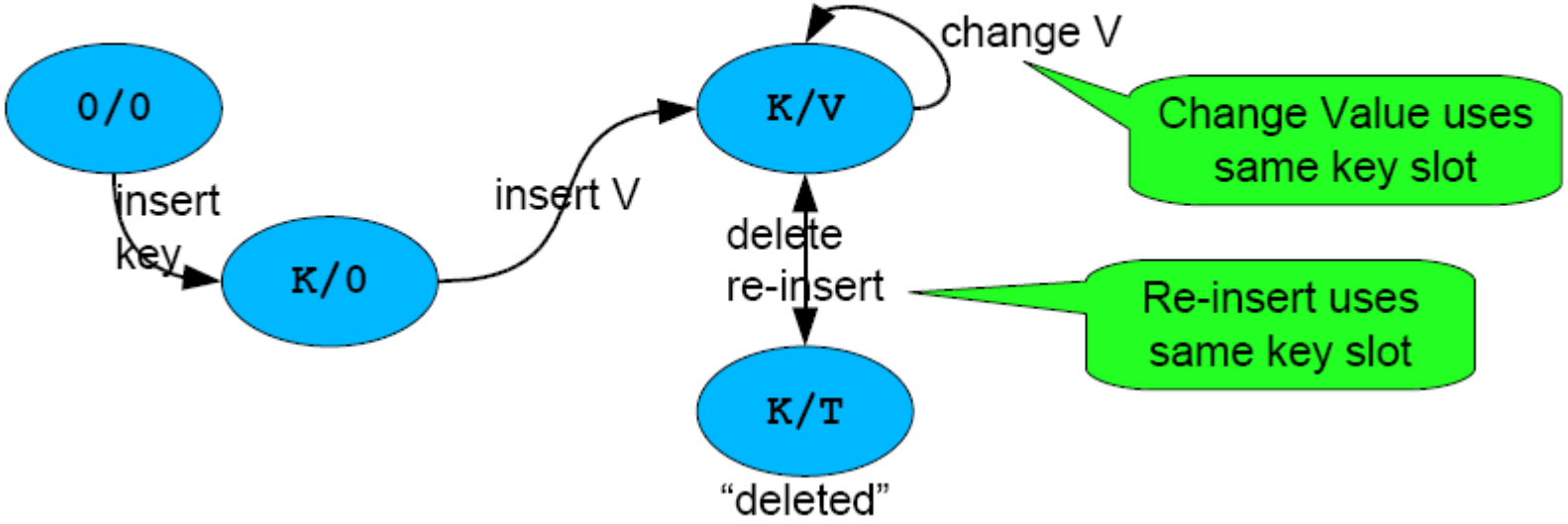
NonBlockingHashmap

HashTable State Machine



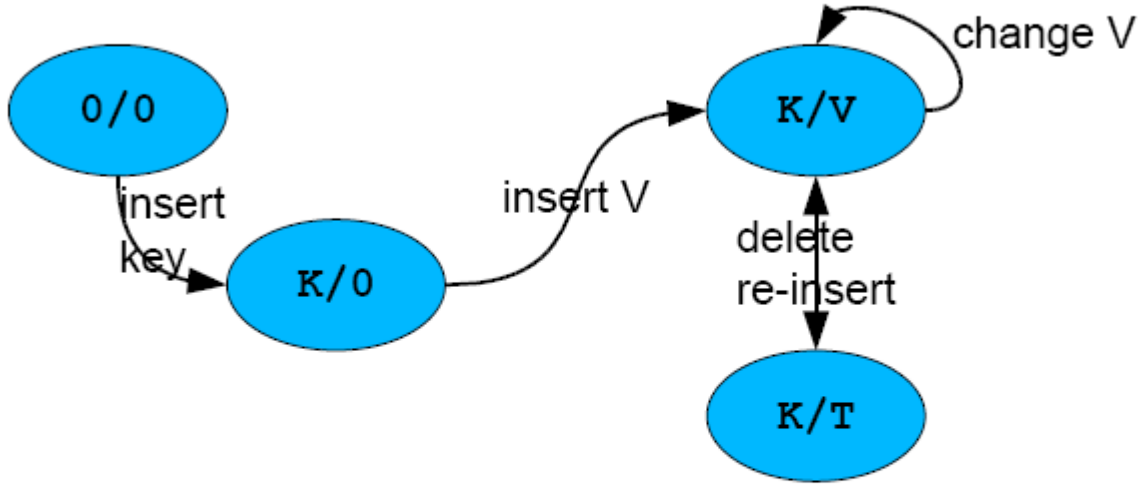
NonBlockingHashmap

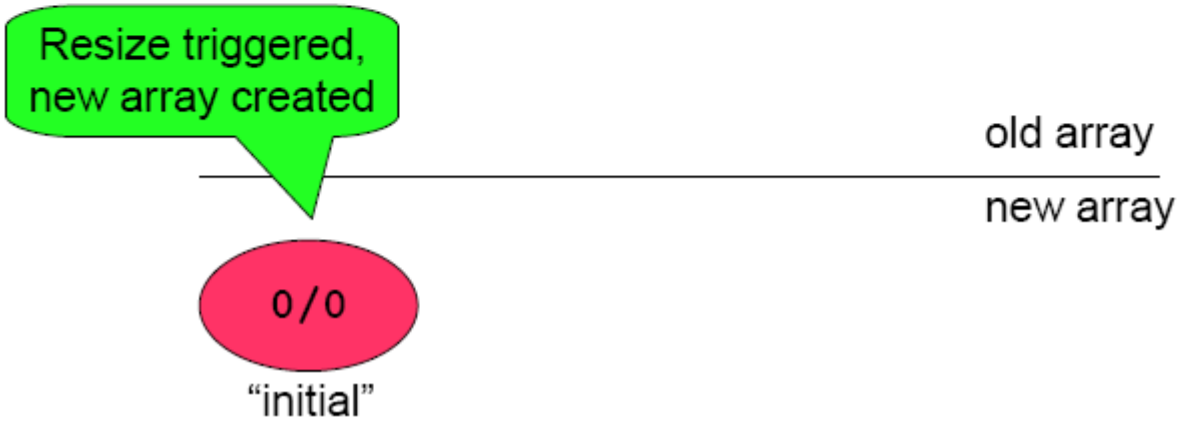
HashTable State Machine



NonBlockingHashmap

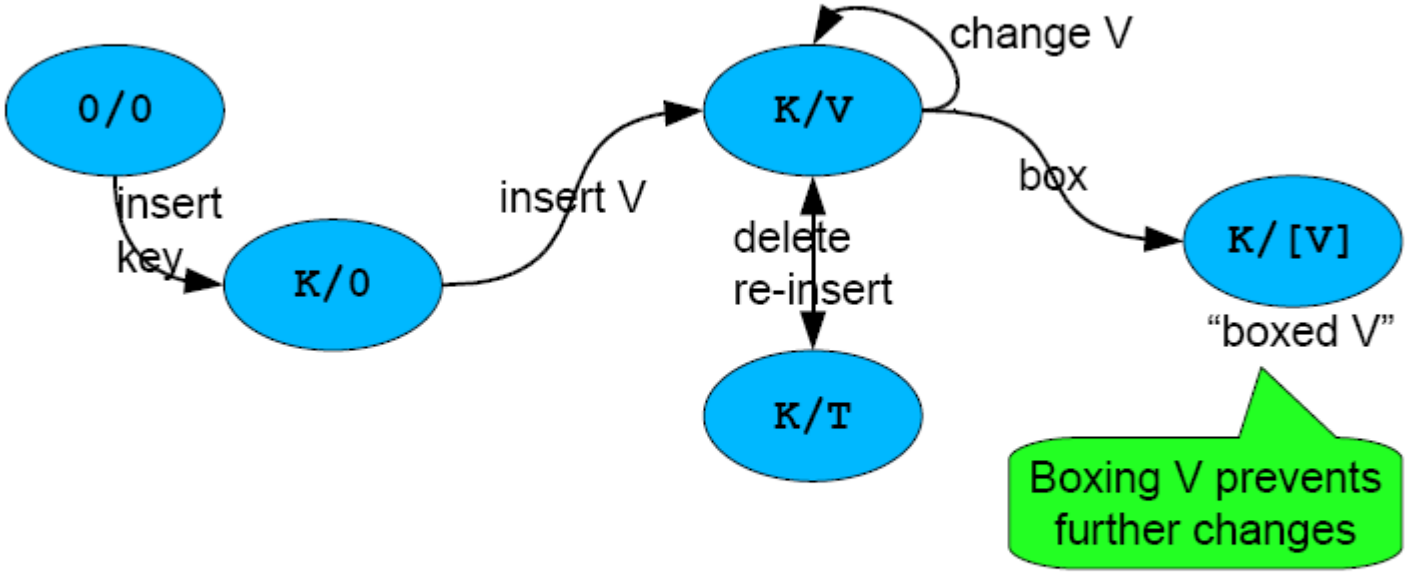
HashTable State Machine





NonBlockingHashmap

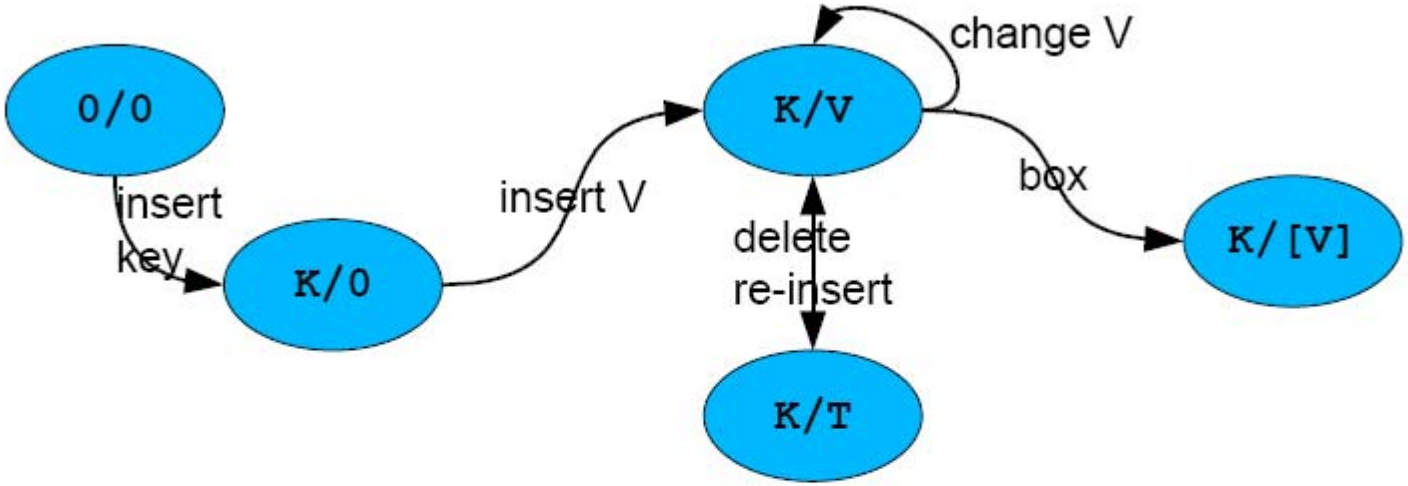
HashTable State Machine

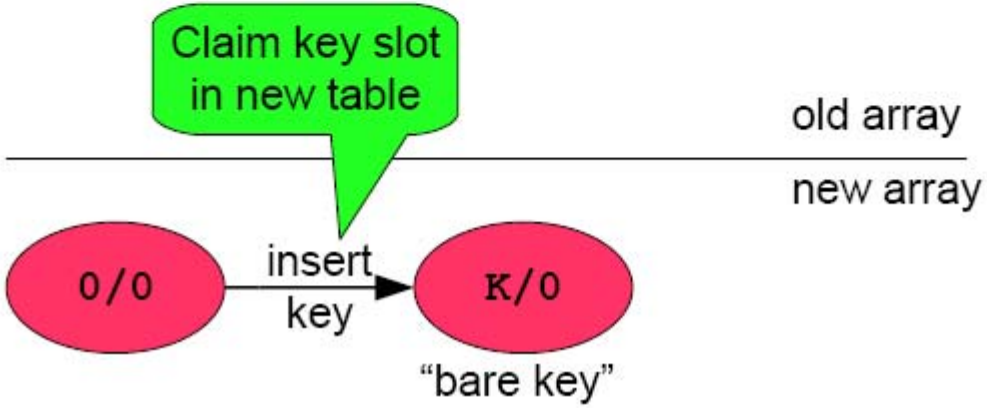


old array

new array

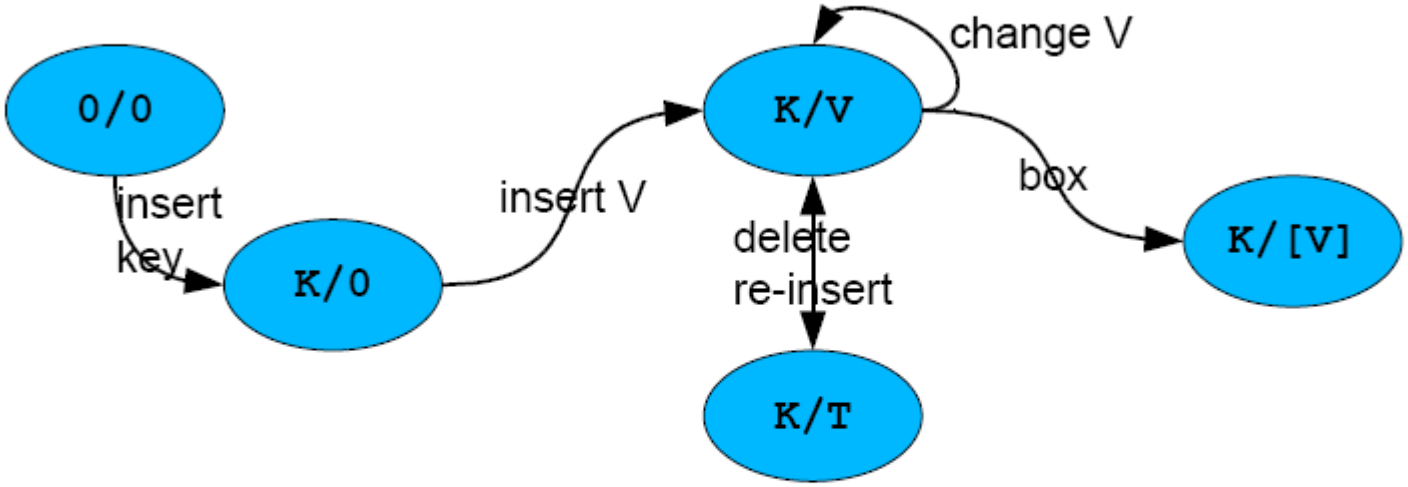
NonBlockingHashmap

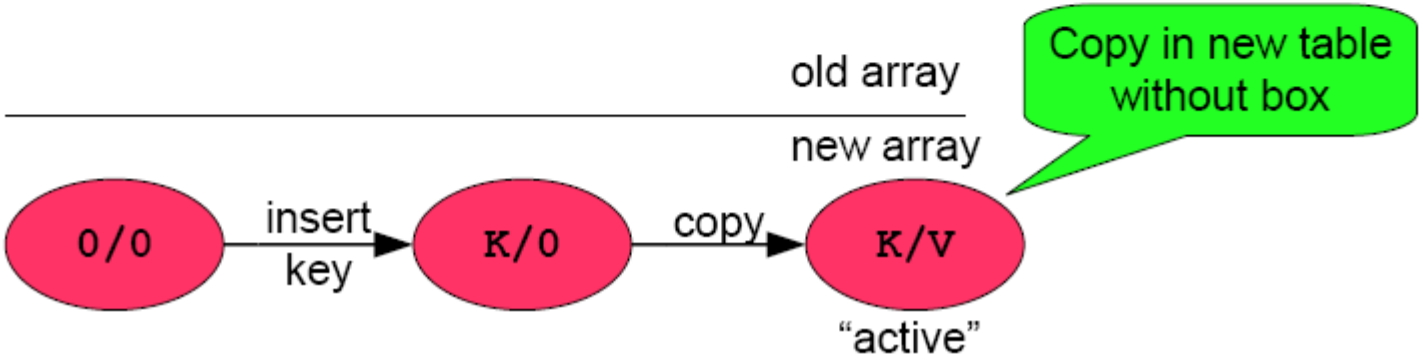




NonBlockingHashmap

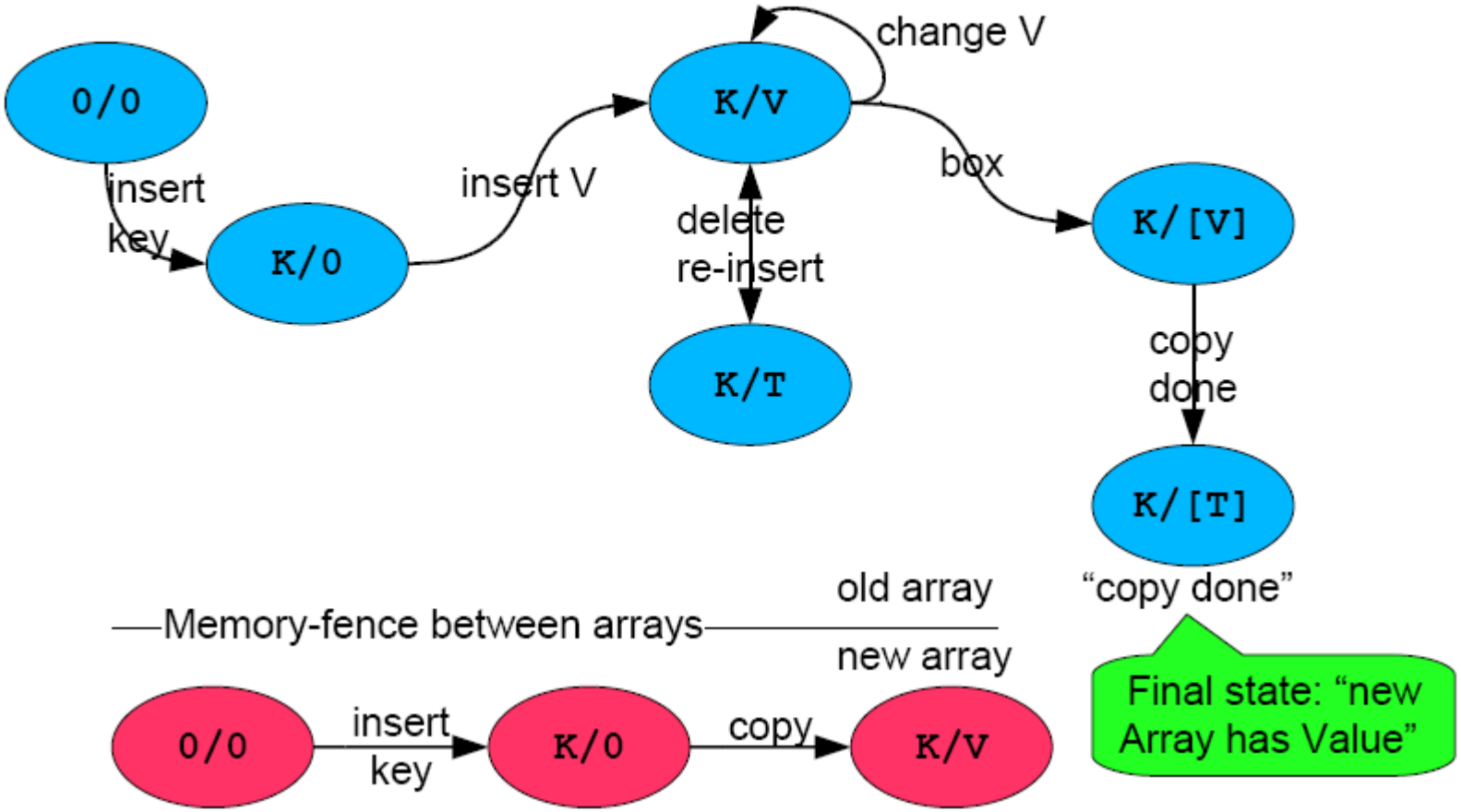
HashTable State Machine





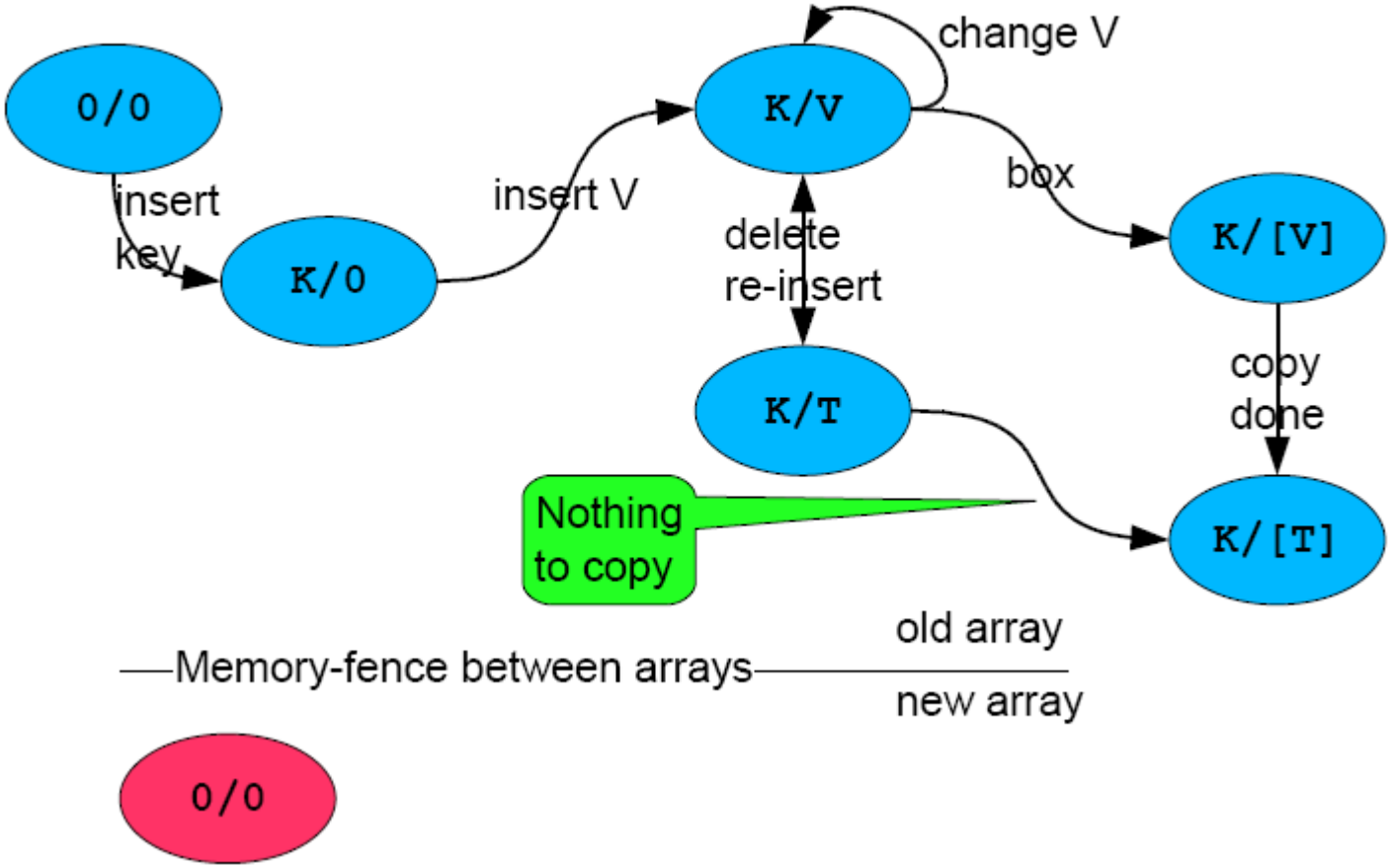
NonBlockingHashmap

HashTable State Machine



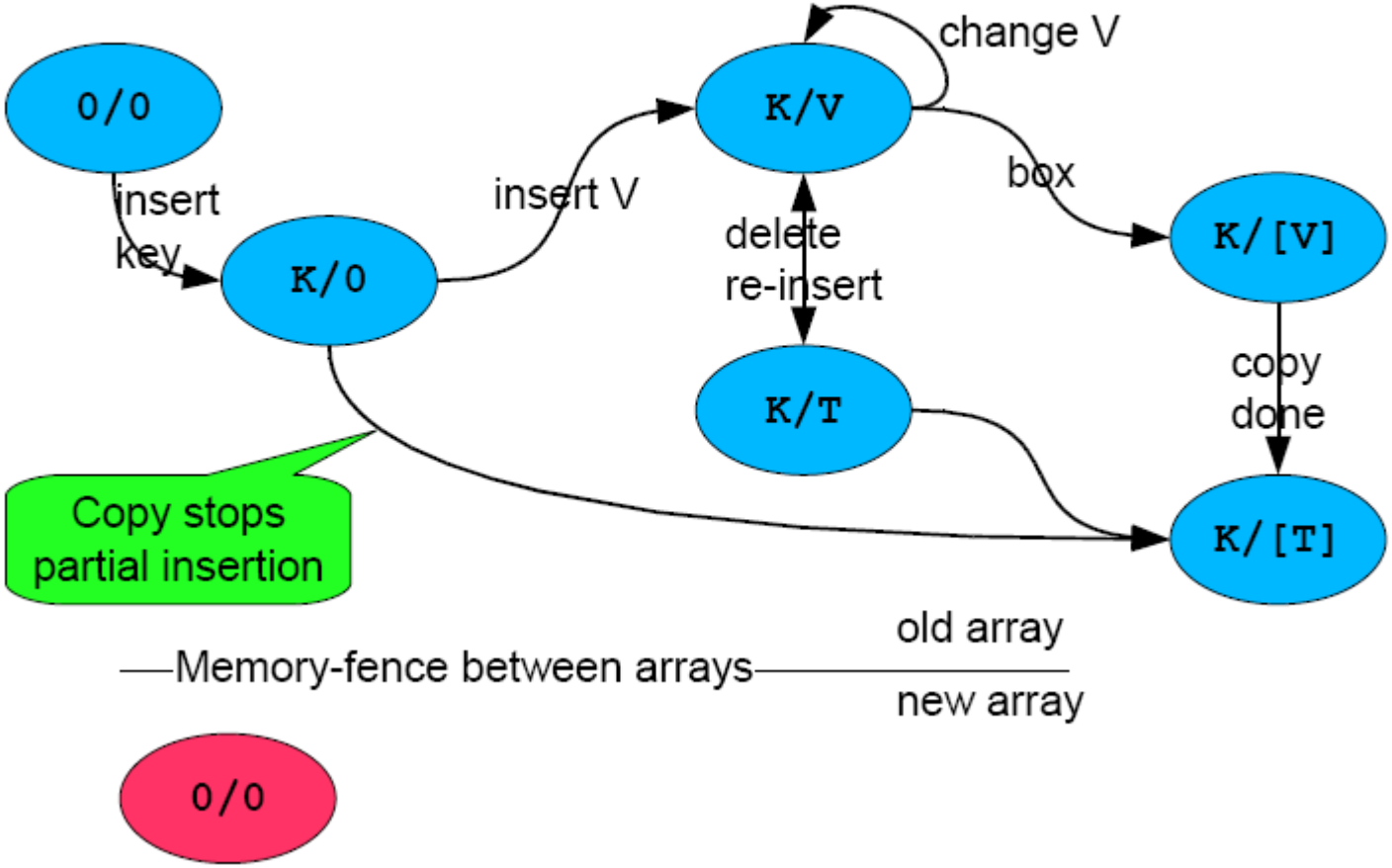
NonBlockingHashmap

HashTable State Machine



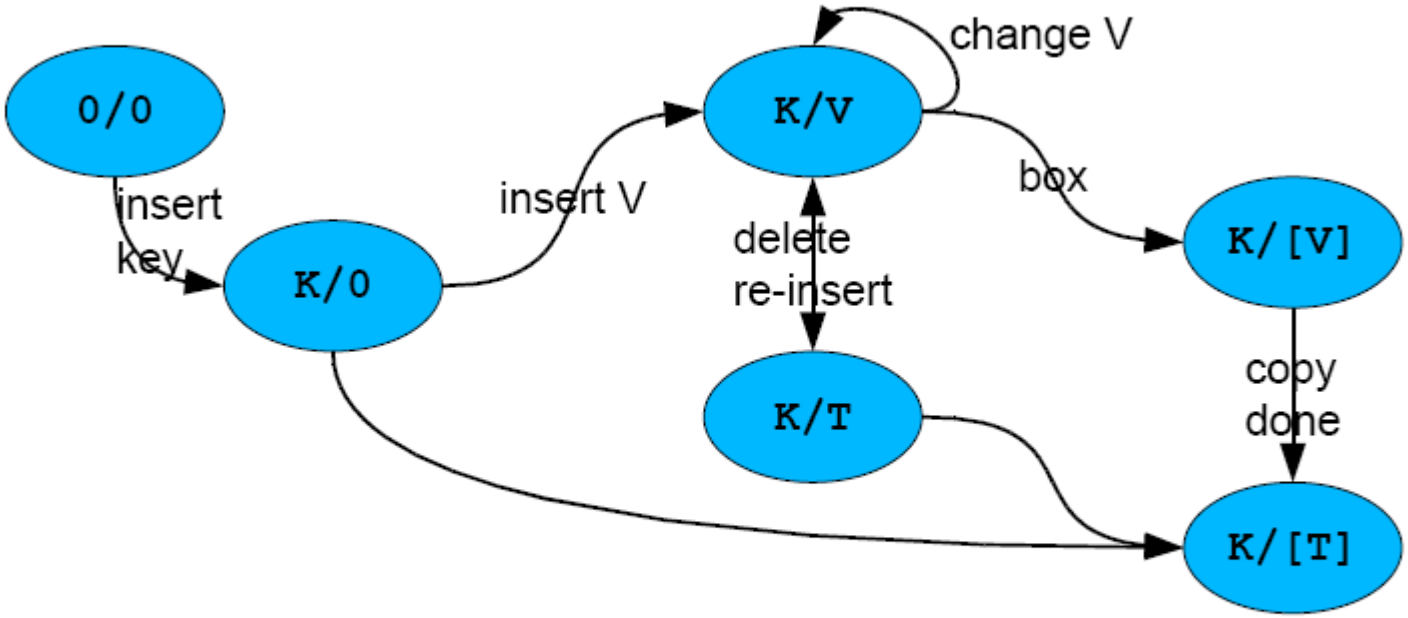
NonBlockingHashmap

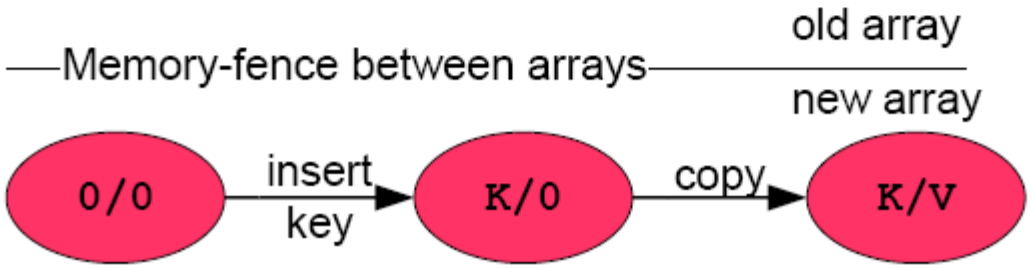
HashTable State Machine



NonBlockingHashmap

HashTable State Machine





LOGO

Thank You !

www.themegallery.com