Introduction to Concurrency and Multicore Programming

Slides adapted from
Art of Multicore Programming
by Herlihy and Shavit

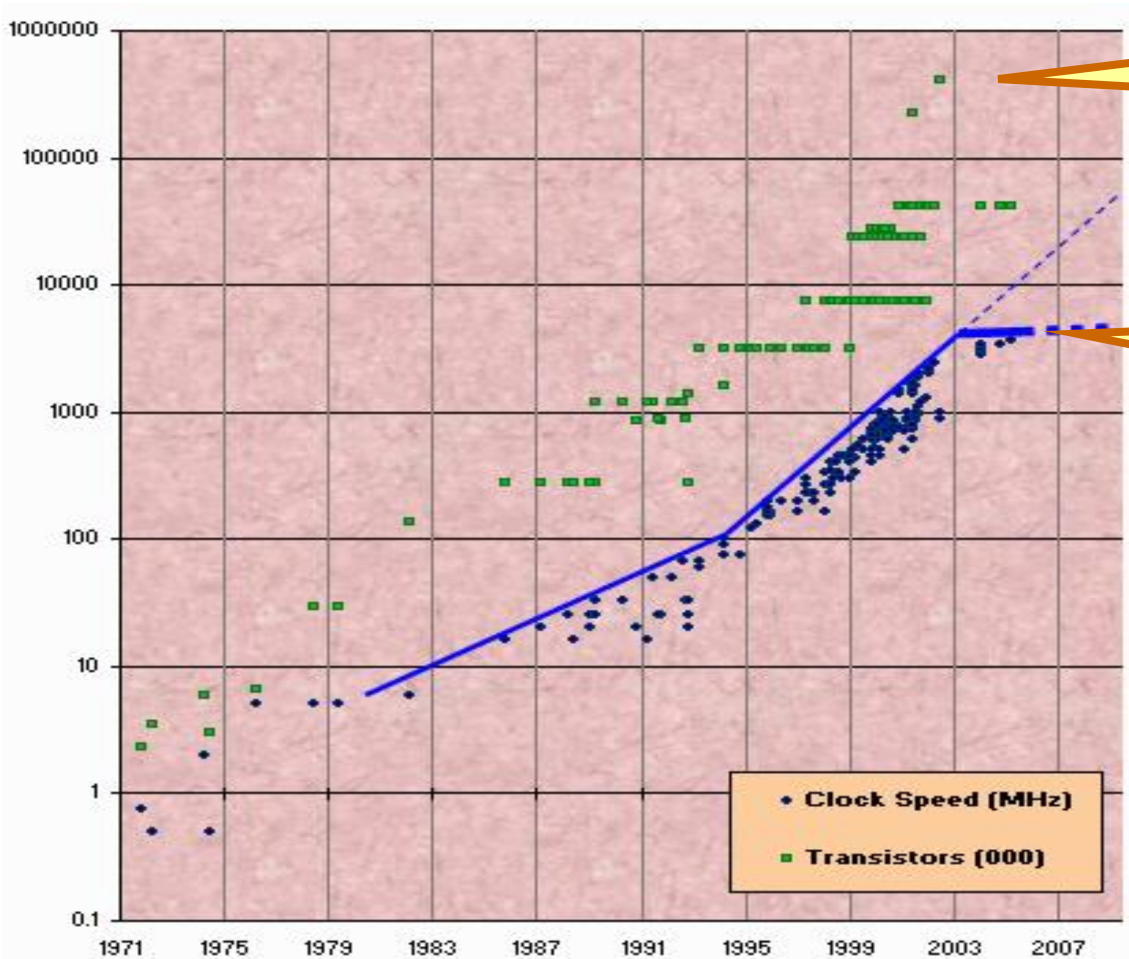
Overview

- Introduction
- Mutual Exclusion
- Linearizability
- Concurrent Data Structure
  - Linked-List Set
  - Lock-free Stack
- Summary

What is Concurrency?

A property of systems in which several processes or threads are executing at the same time.

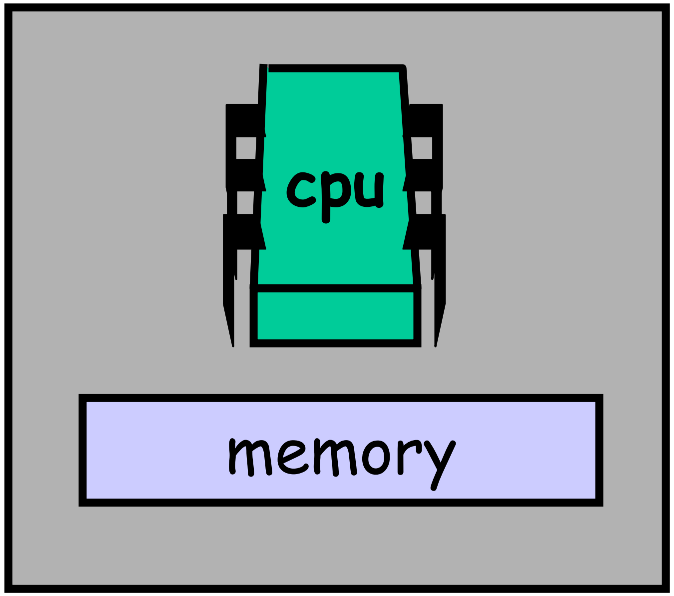
Moore's Law



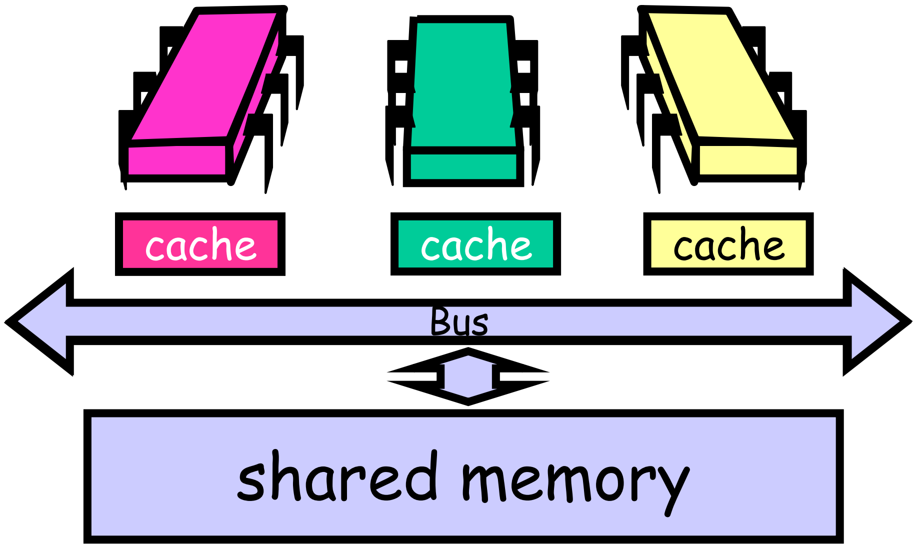
Transistor count still rising

Clock speed flattening sharply

The Uniprocessor is Vanishing!

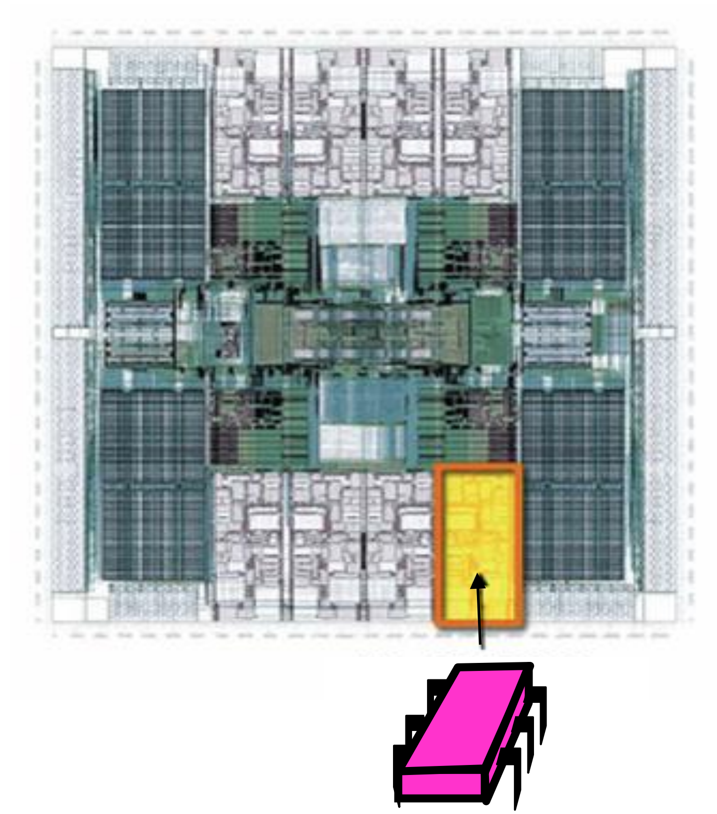


The Shared Memory Multiprocessor (SMP)



Your New Desktop: The Multicore Processor (CMP)

All on the same chip

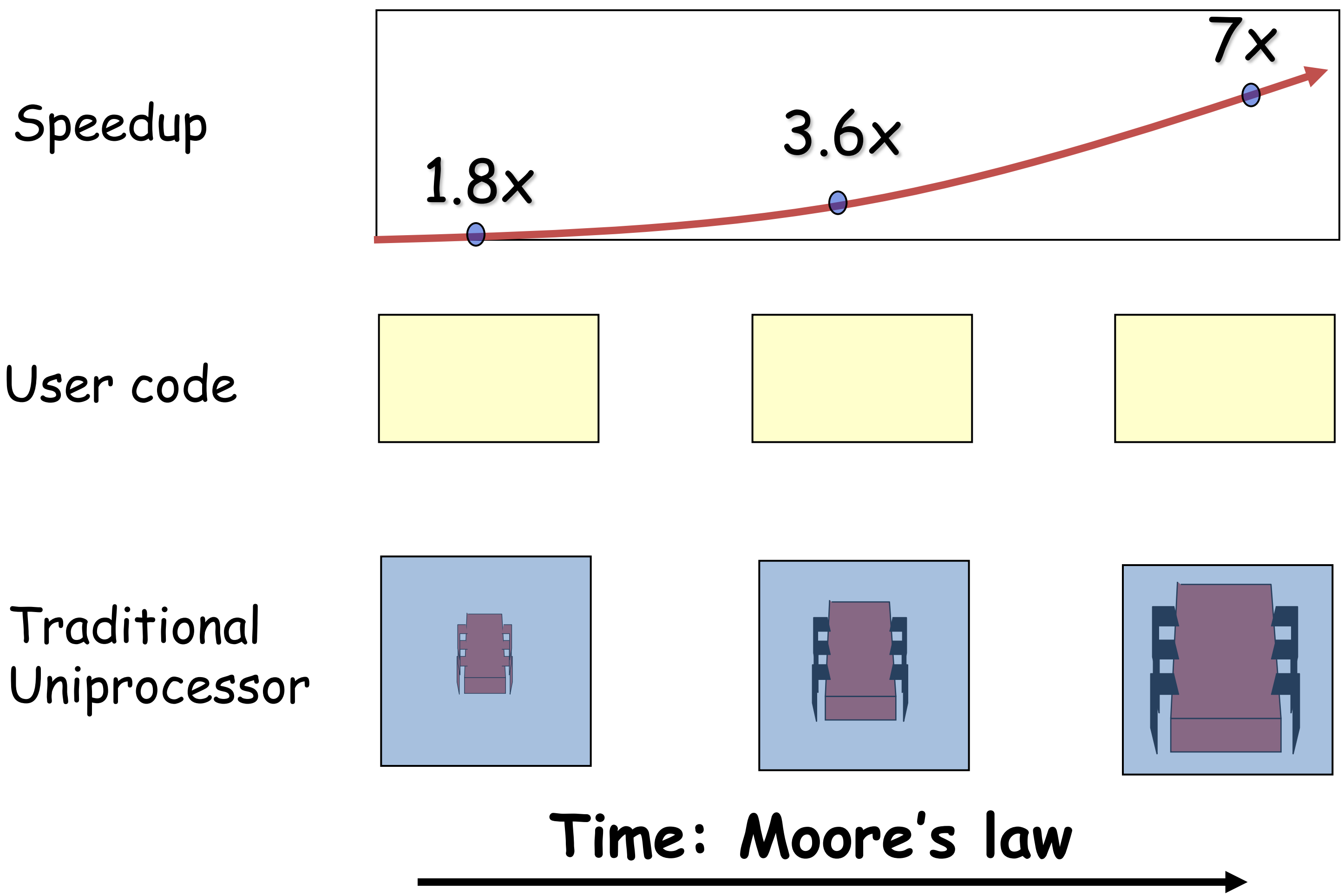


Sun T2000 Niagara

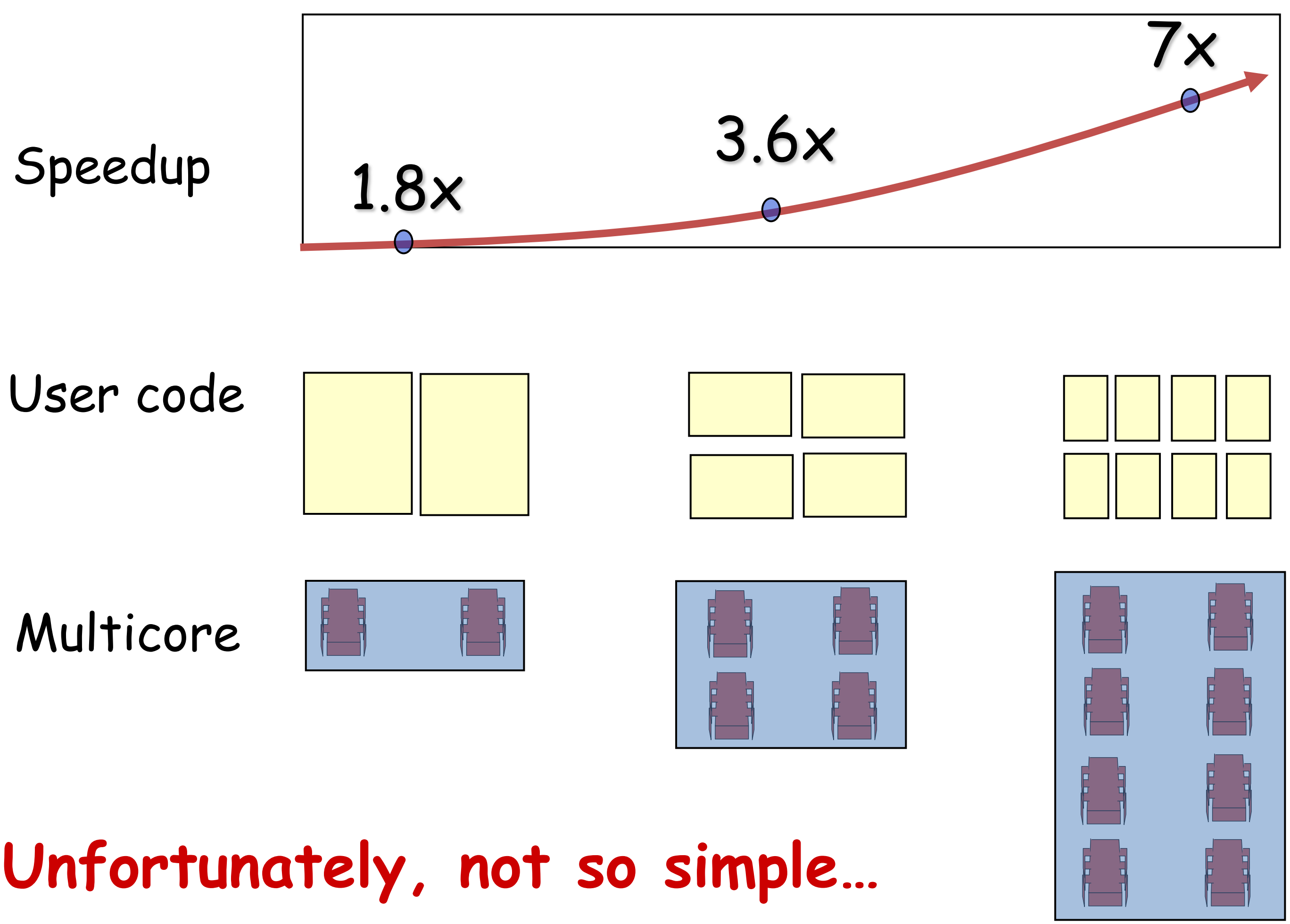
Why do we care?

- Time no longer cures software bloat
  - The "free ride" is over
- When you double your program's path length
  - You can't just wait 6 months
  - Your software must somehow exploit twice as much concurrency

Traditional Scaling Process

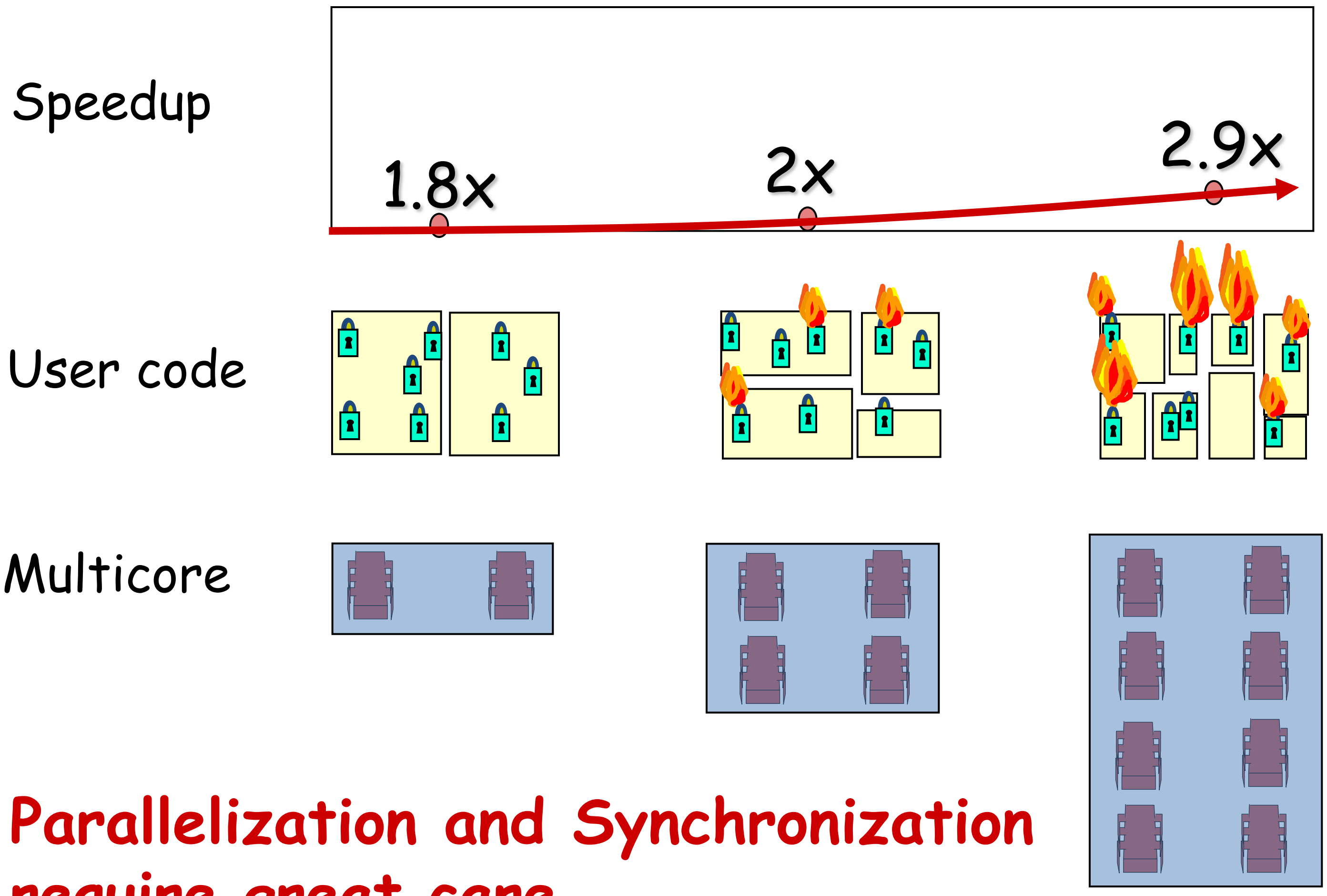


Multicore Scaling Process



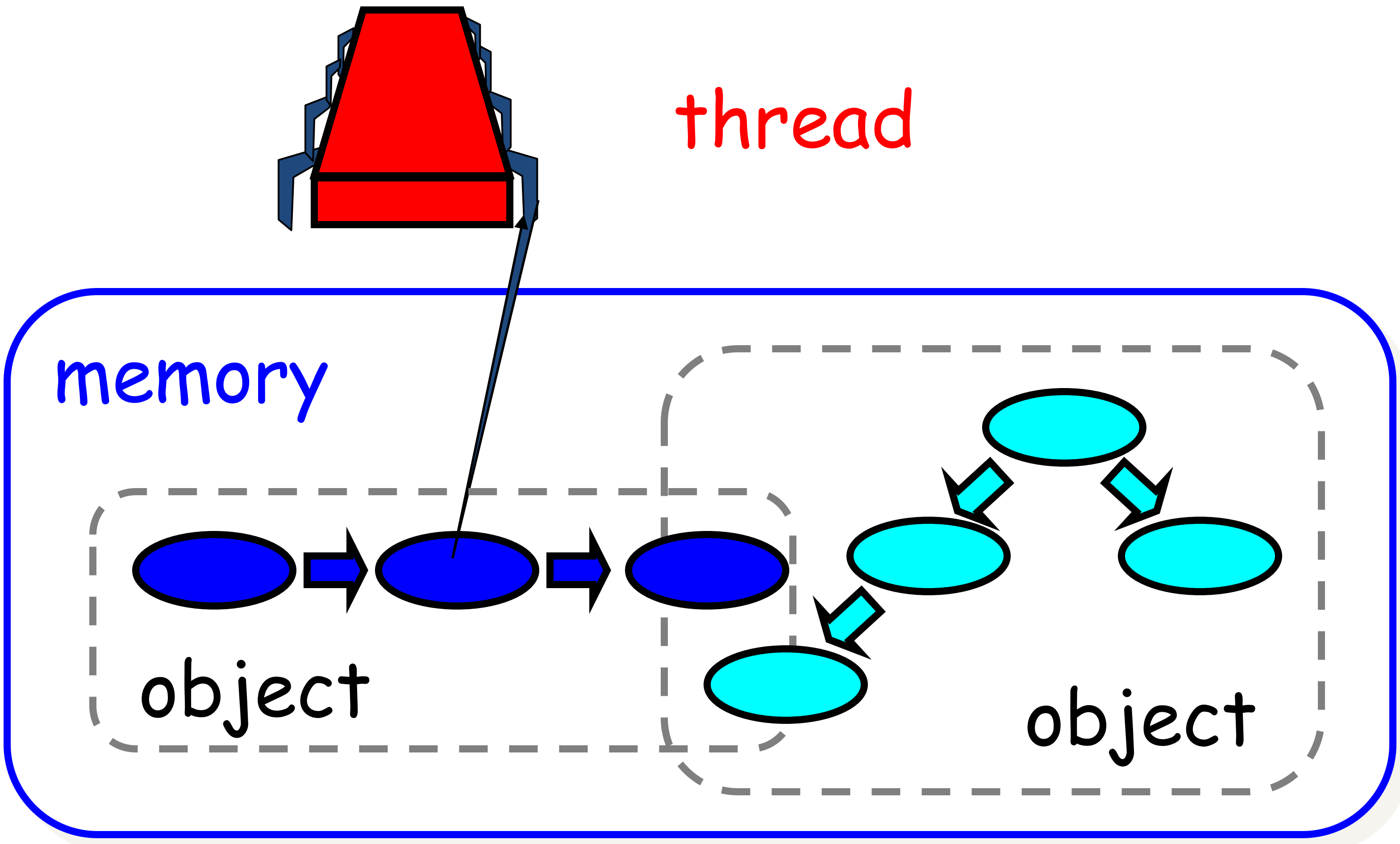
Unfortunately, not so simple…

Real-World Scaling Process

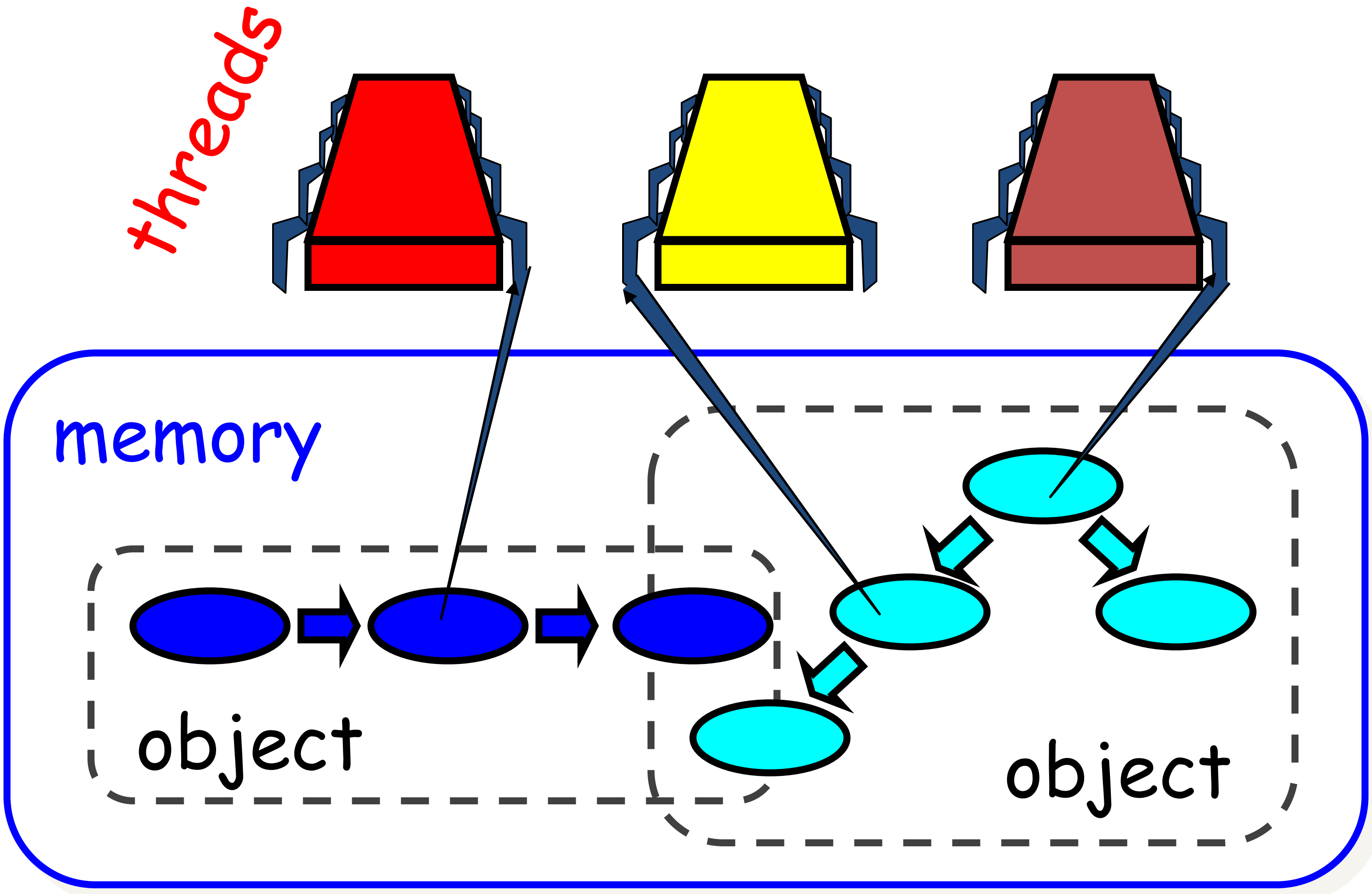


Parallelization and Synchronization require great care…

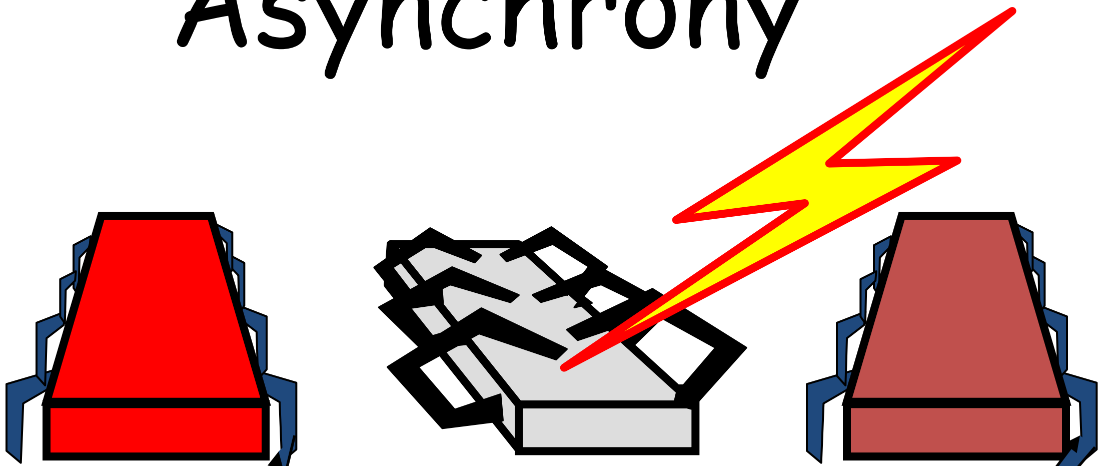
Sequential Computation



Concurrent Computation



Asynchrony



- Sudden unpredictable delays
  - Cache misses (*short*)
  - Page faults (*long*)
  - Scheduling quantum used up (*really long*)

Model Summary

- Multiple *threads*
- Single shared *memory*
- *Objects* live in memory
- Unpredictable asynchronous delays

Multithread Programming

- Java, C#, Pthreads
- Windows Thread API
- OpenMP
- Intel Parallel Studio Tool Kits

Java Thread

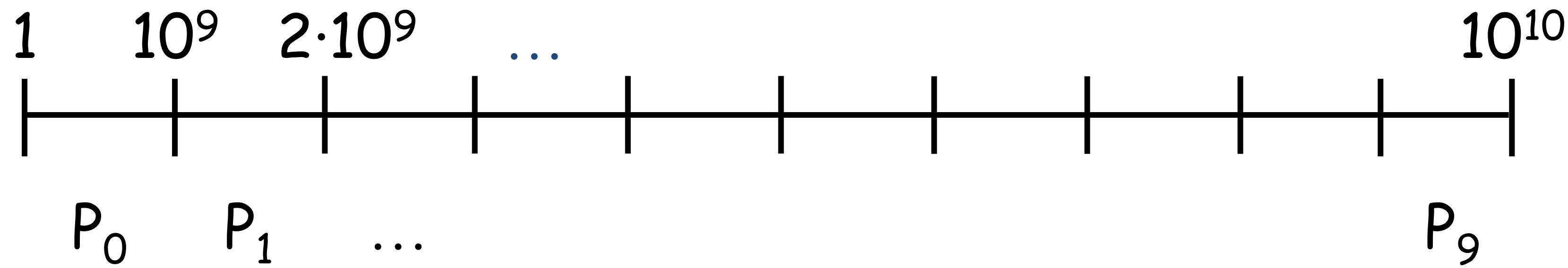
- java.lang.Thread

```
class MyThread extends Thread{
    @Override
    public void run(){
        ...
    }
}

public static void main(String args[]){
     MyThread thread = new MyThread();
     thread.start();
     try {
        thread.join();
     }
     catch (InterruptedException e) { };
}
```

Concurrency Idea

- Challenge
  - Print primes from 1 to 10^{10}
- Given
  - Ten-processor multiprocessor
  - One thread per processor
- Goal
  - Get ten-fold speedup (or close)

Load Balancing



- Split the work evenly
- Each thread tests range of 10^9

Procedure for Thread *i*

```
void primePrint {
  int i = ThreadID.get(); // IDs in {0..9}
  for (j = i*10^9+1, j<(i+1)*10^9; j++) {
    if (isPrime(j))
      print(j);
  }
}
```

Issues

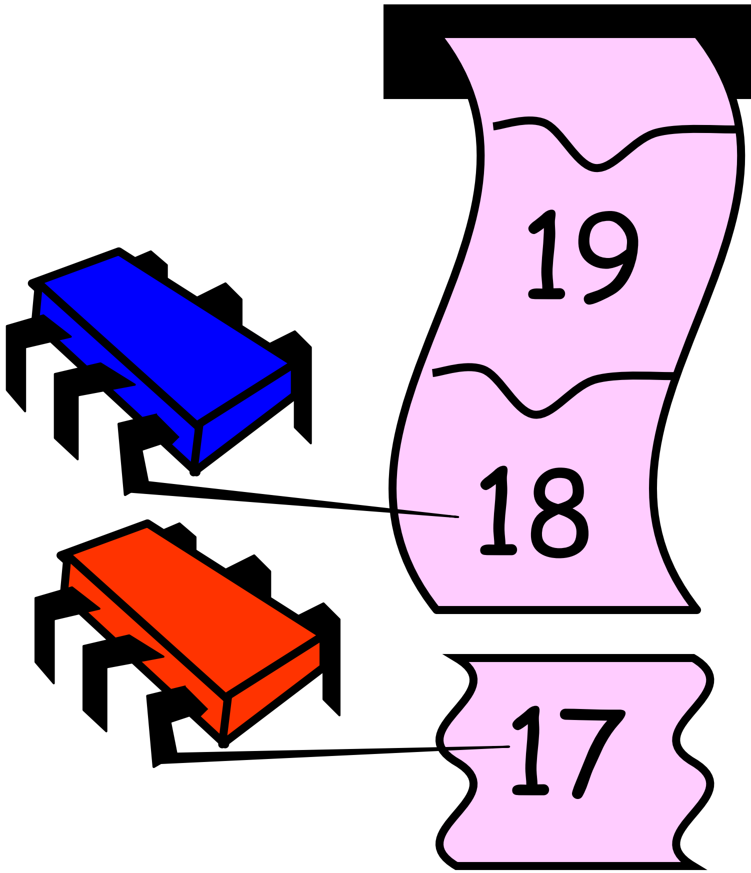
- Higher ranges have fewer primes
- Yet larger numbers harder to test
- Thread workloads
  - Uneven
  - Hard to predict

Issues

- Higher ranges have fewer primes
- Yet larger numbers harder to test
- Thread workloads
  - Uneven
  - Hard to predict
- Need *dynamic* load balancing

rejected

Shared Counter



each thread takes a number

Procedure for Thread *i*

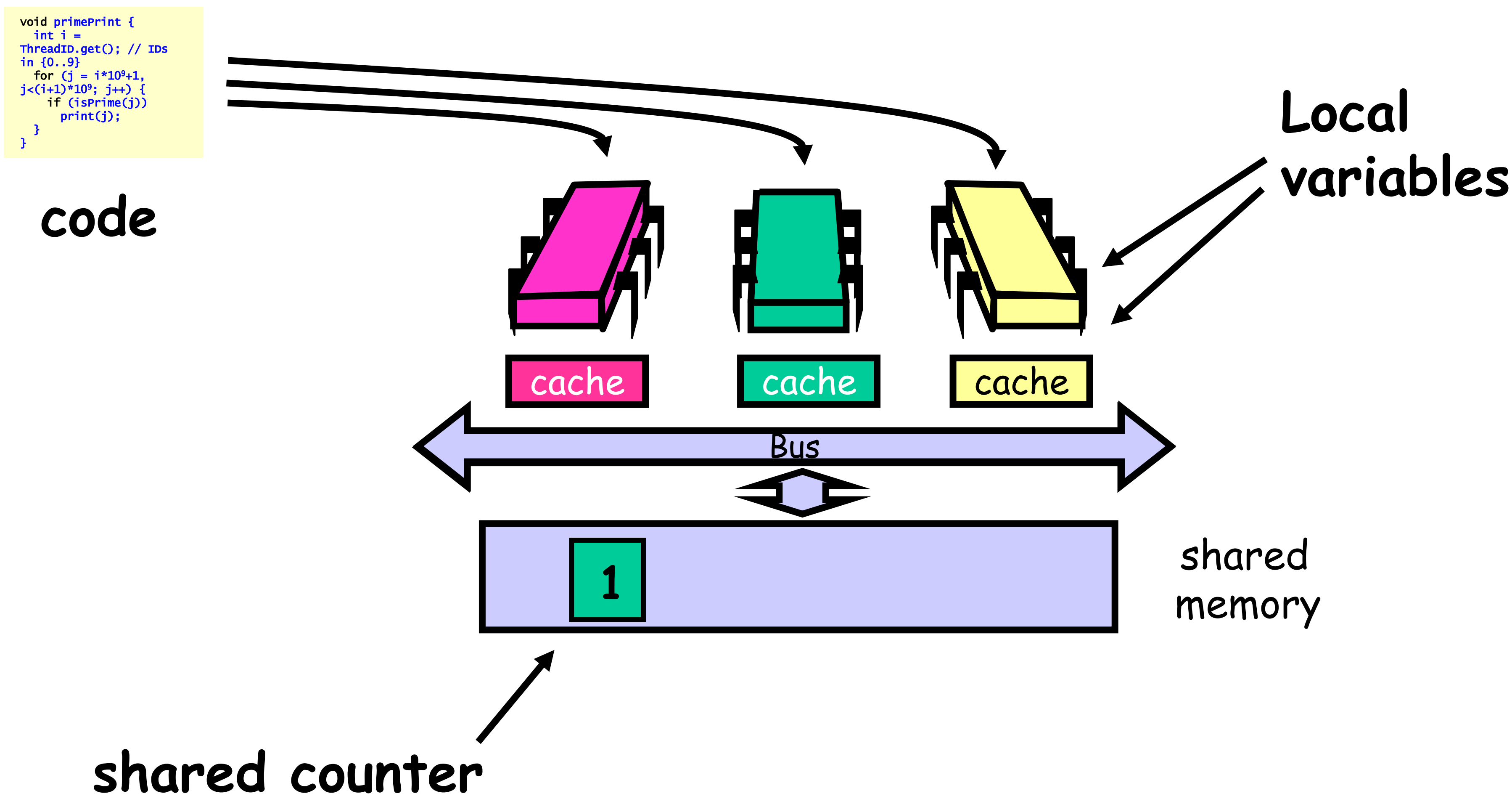
```
int counter = new Counter(1);

void primePrint {
  long j = 0;
  while (j < 10^10) {
    j = counter.getAndIncrement();
    if (isPrime(j))
      print(j);
  }
}
```

Procedure for Thread *i*

```
Counter counter = new Counter(1);

void primePrint {
  long j = 0;
  while (j < 10^10) {
    j = counter.getAndIncrement();
    if (isPrime(j))
      print(j);
  }
}
```

Shared counter object

Where Things Reside



Counter Implementation

```
public class Counter {
  private long value;

  public long getAndIncrement() {
    return value++;
  }
}
```

Counter Implementation

```
public class Counter {
  private long value;

  public long getAndIncrement() {
    return value++;
  }
}
```

OK for single thread,
not for concurrent threads

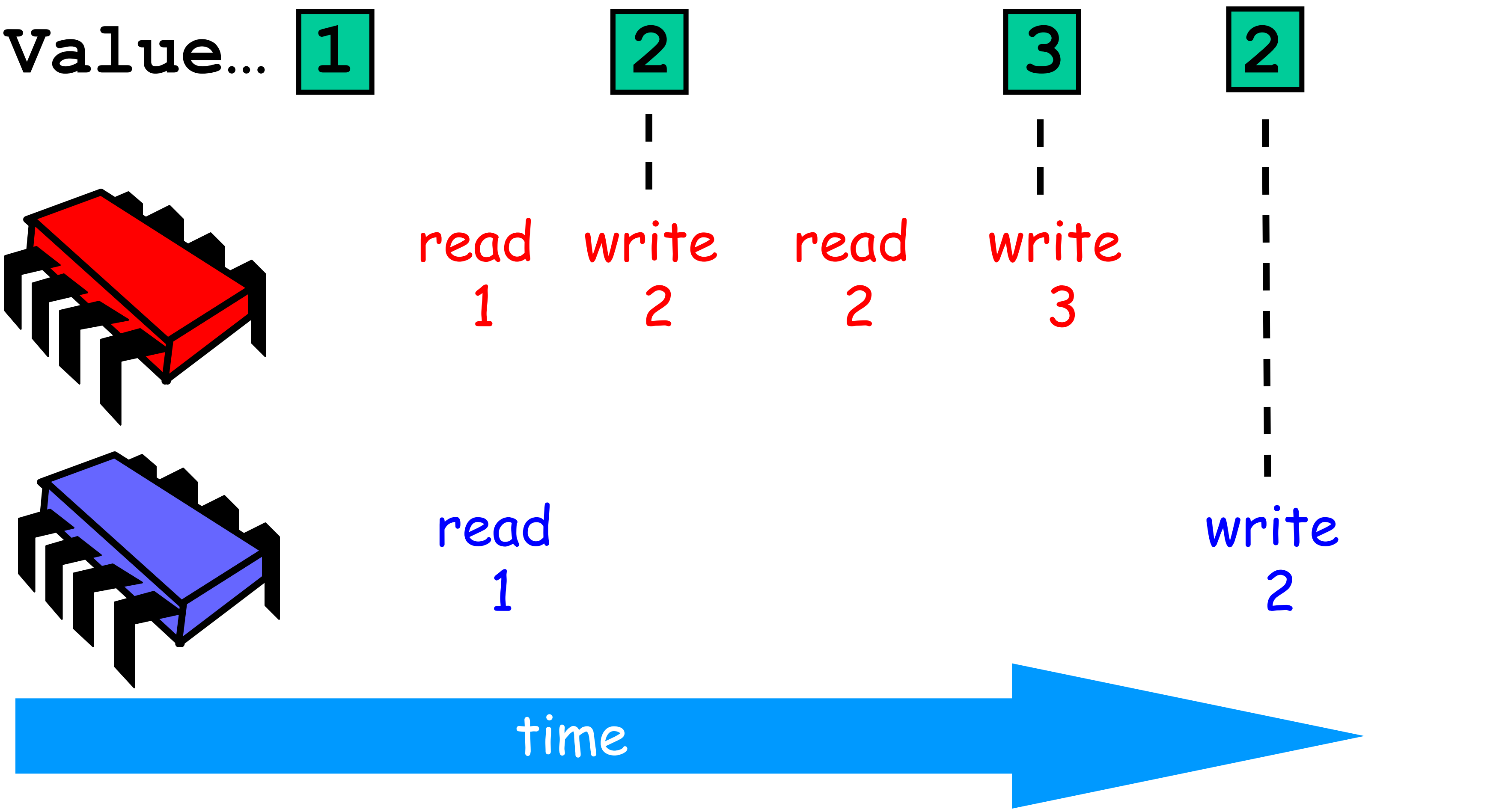
What It Means

```
public class Counter {
  private long value;

  public long getAndIncrement() {
    return value++;
  }
}
```

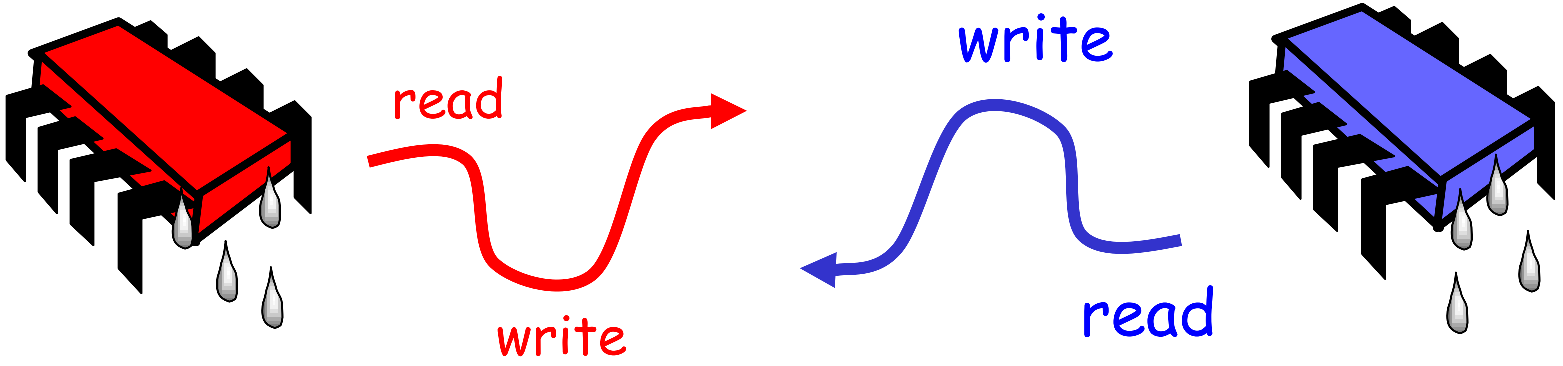
What It Means

```
public class Counter {
  private long value;

  public long getAndIncrement() {
    return value++;        temp  = value;
  }                        value = value + 1;
}                          return temp;
```

Not so good…



Is this problem inherent?



If we could only glue reads and writes…

Challenge

```
public class Counter {
  private long value;

  public long getAndIncrement() {
    temp  = value;
    value = temp + 1;
    return temp;
  }
}
```

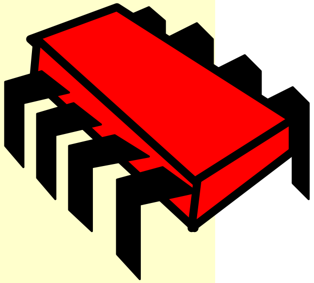
Challenge

```
public class Counter {
  private long value;

  public long getAndIncrement() {
    temp  = value;
    value = temp + 1;
    return temp;
  }
}
```

Make these steps *atomic* (indivisible)

Hardware Solution

```
public class Counter {
  private long value;

  public long getAndIncrement() {
    temp  = value;
    value = temp + 1;
    return temp;
  }
}
```



ReadModifyWrite()
instruction

An Aside: Java™

```
public class Counter {
  private long value;

  public long getAndIncrement() {
    synchronized {
      temp  = value;
      value = temp + 1;
      }
    return temp;
  }
}
```

An Aside: Java™

```
public class Counter {
  private long value;

  public long getAndIncrement() {
    synchronized {
      temp  = value;
      value = temp + 1;
      }
    return temp;
  }
}
```

Synchronized block

An Aside: Java™

```
public class Counter {
  private long value;

  public long getAndIncrement() {
    synchronized {
      temp  = value;
      value = temp + 1;
    }
    return temp;
  }
}
```

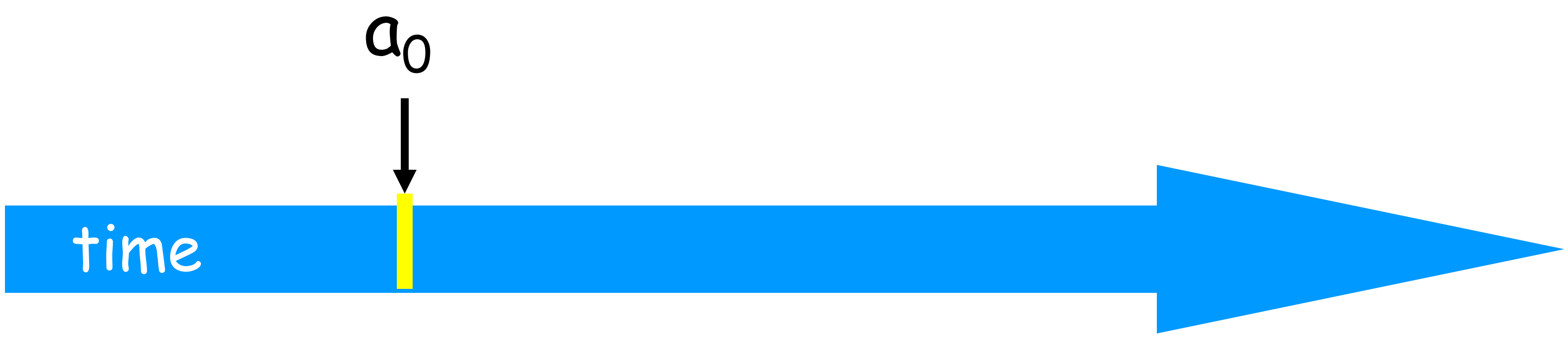
Mutual Exclusion

Mutual Exclusion

The problem of ensuring that no two processes or threads can be in their ***critical section*** at the same time.

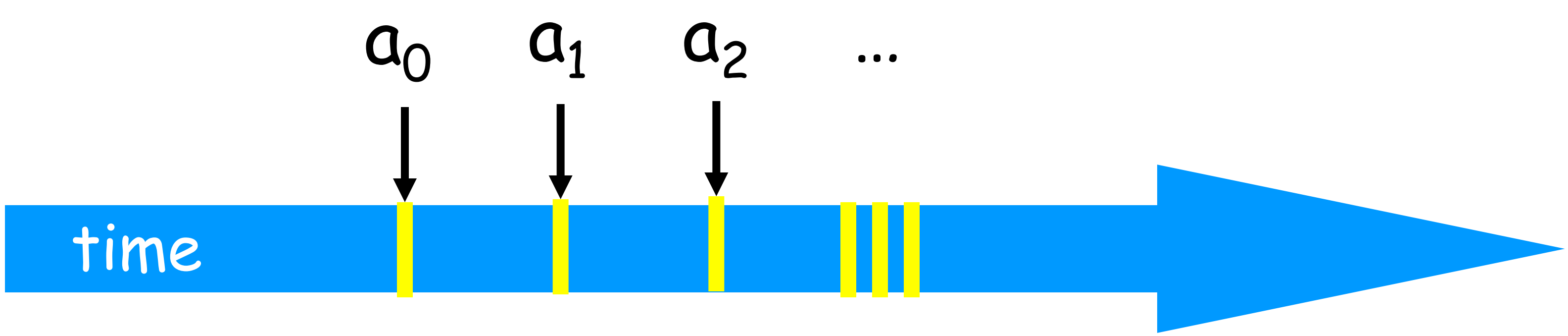
Events

- An *event* a_0 of thread *A* is
  - Instantaneous
  - No simultaneous events (break ties)



Threads

- A *thread* A is (formally) a sequence $a_0, a_1, \ldots$ of events
  - "Trace" model
  - Notation: $a_0 \rightarrow a_1$ indicates order

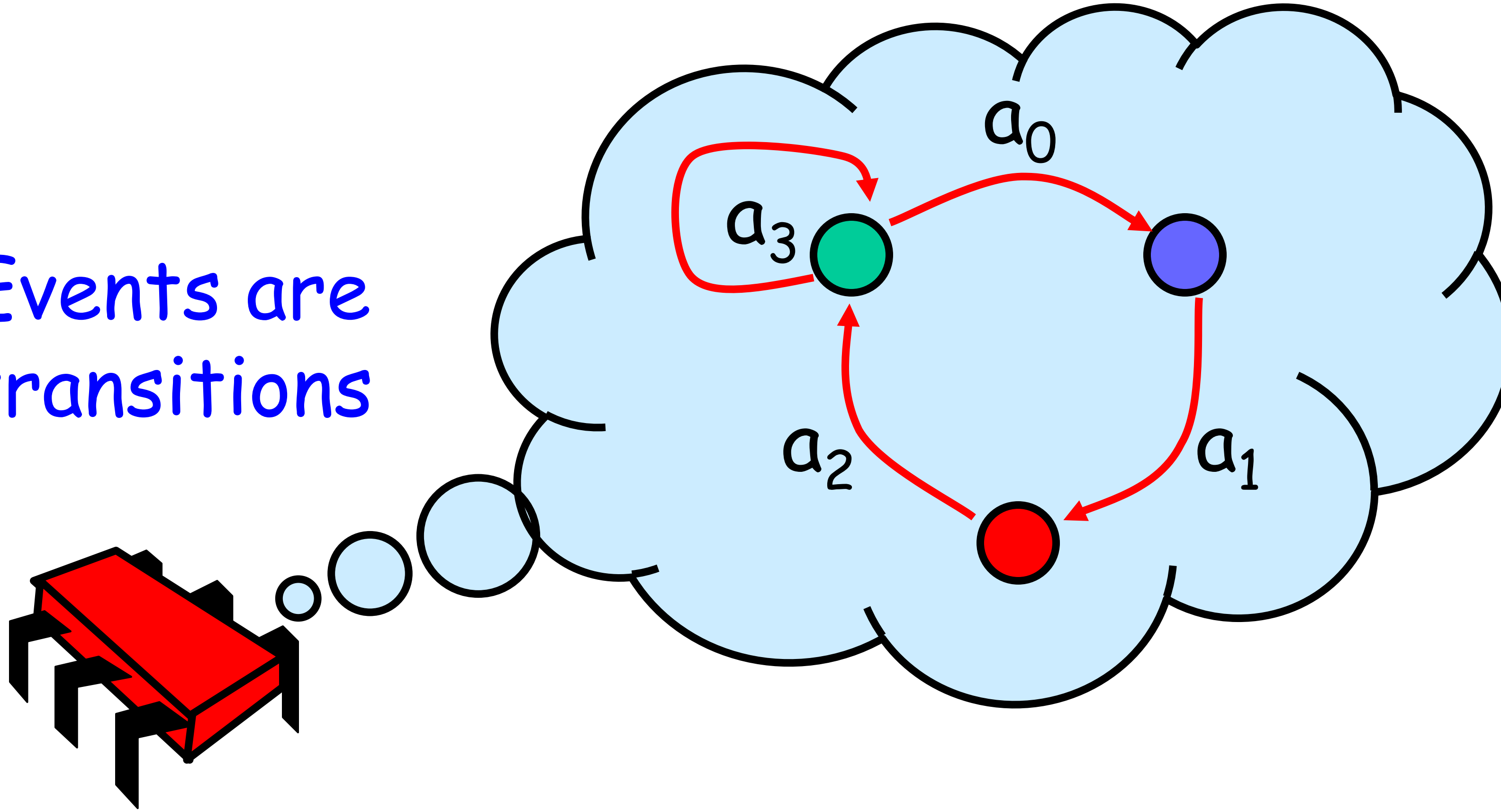


Example Thread Events

- Assign to shared variable
- Assign to local variable
- Invoke method
- Return from method
- Lots of other things …

Threads are State Machines

Events are transitions



States

- Thread State
  - Program counter
  - Local variables
- System state
  - Object fields (shared variables)
  - Union of thread states

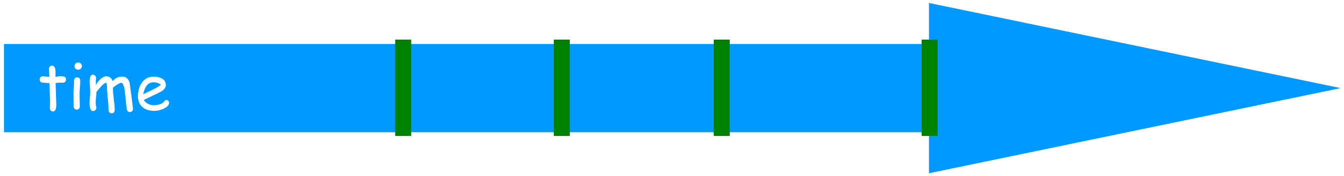
Concurrency

- Thread *A*

Concurrency

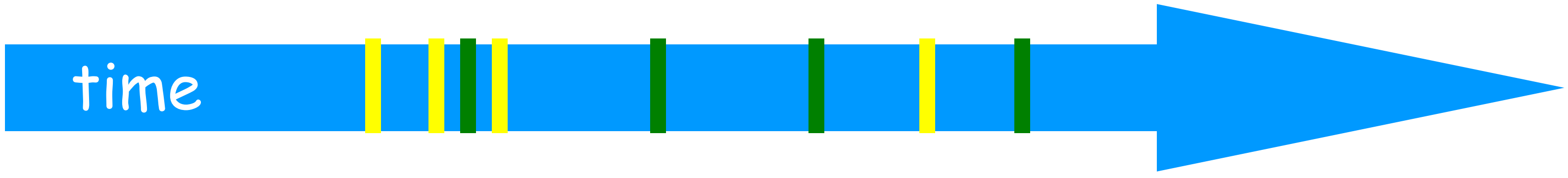
- Thread A

- Thread B



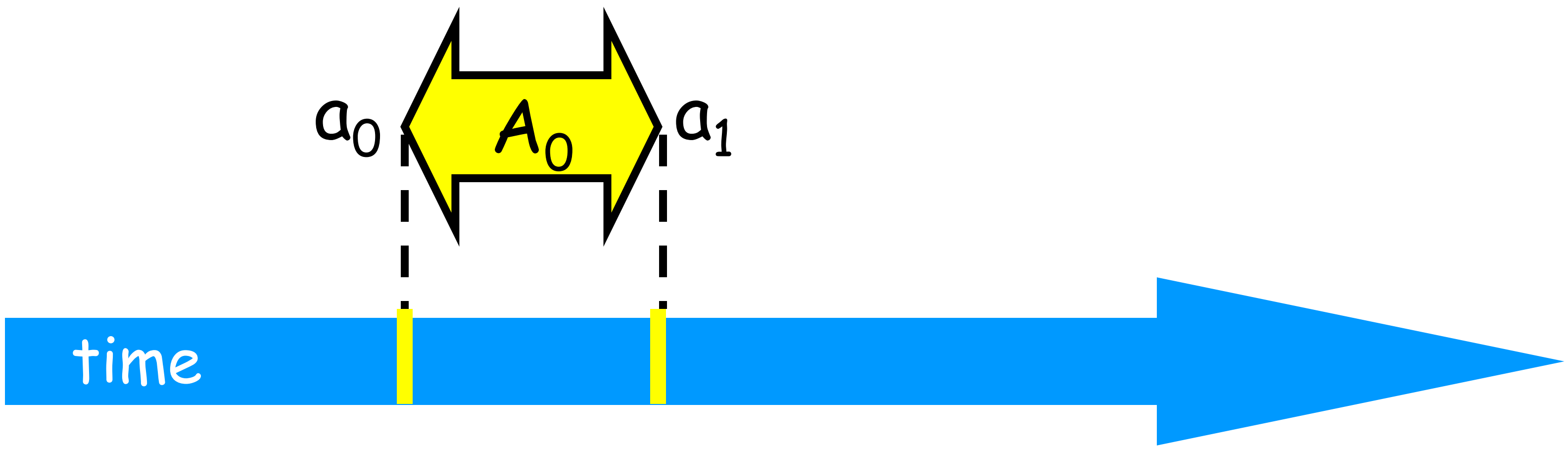
Interleavings

- Events of two or more threads
  - Interleaved
  - Not necessarily independent (why?)

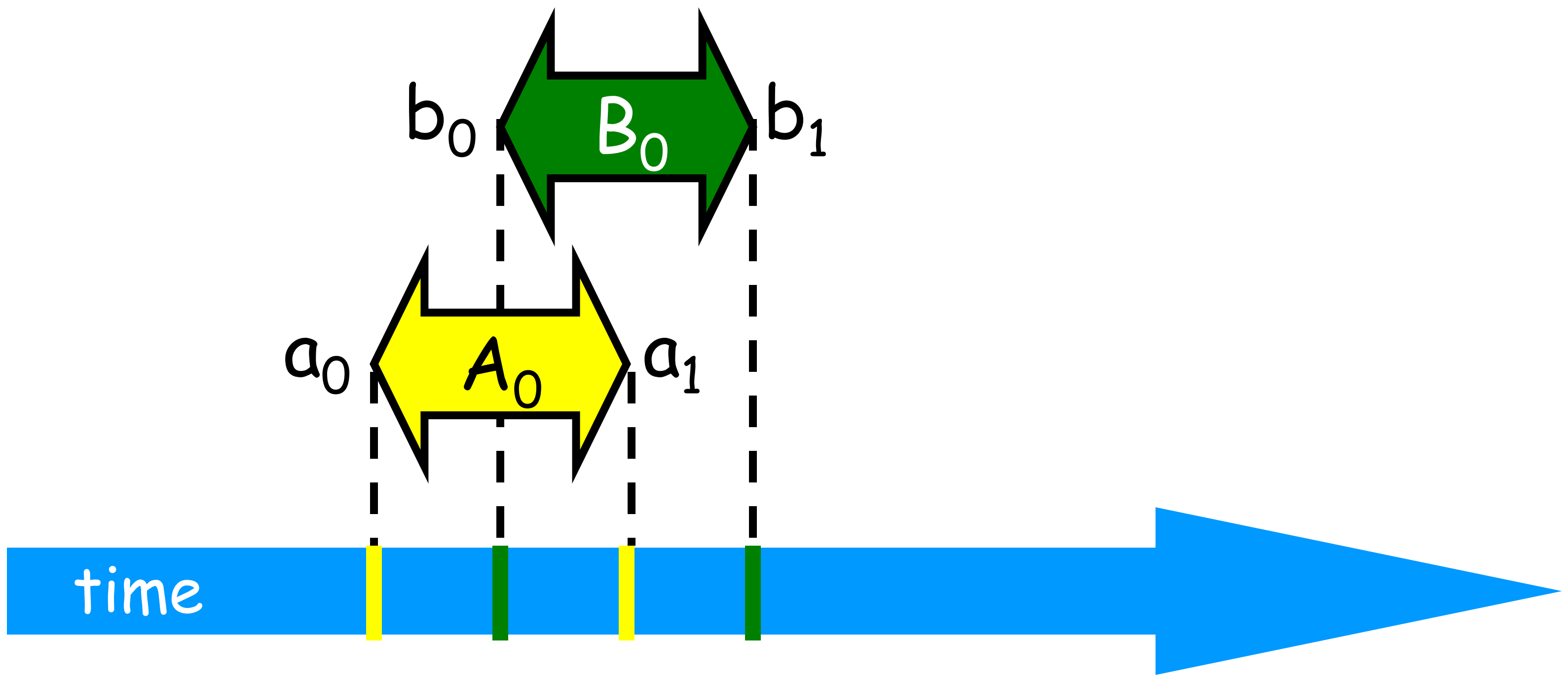


Intervals

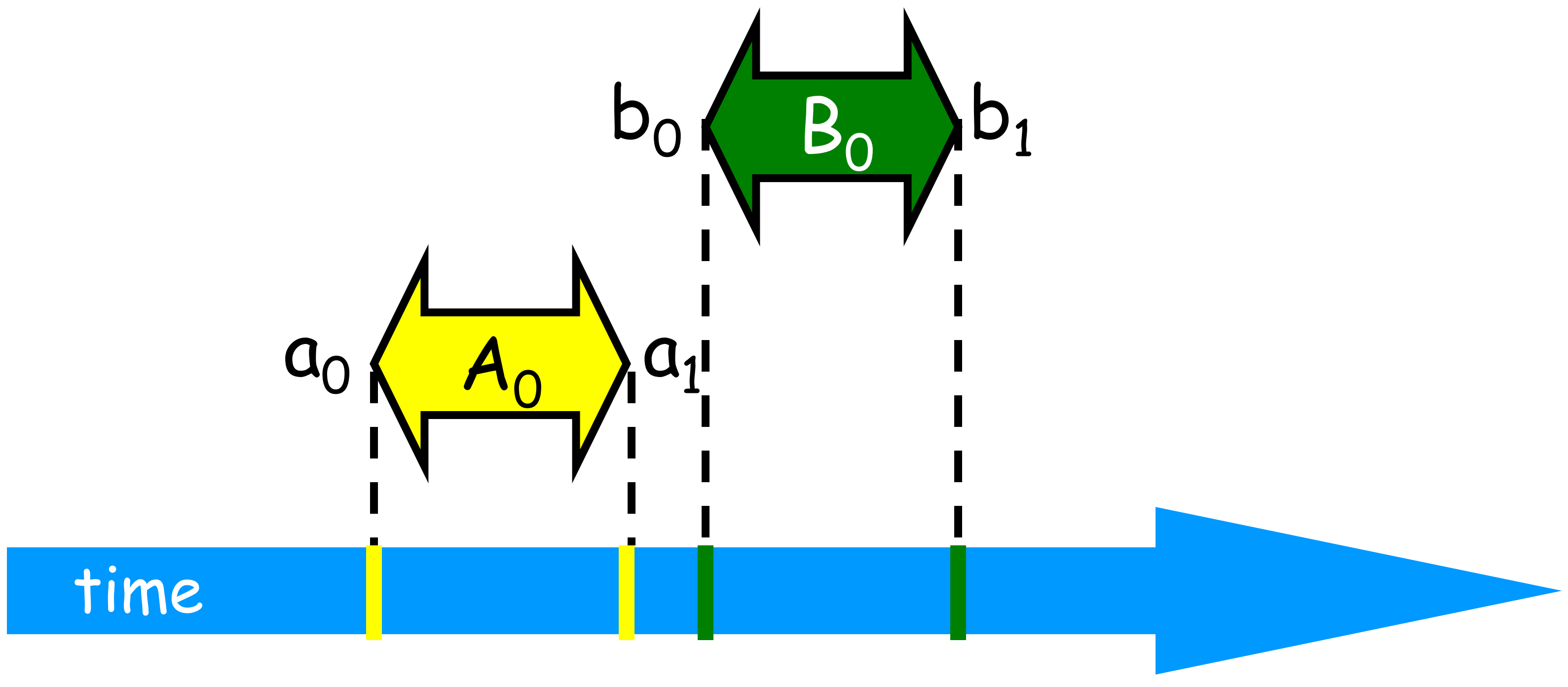
- An *interval* $A_0 = (a_0, a_1)$ is
  - Time between events a_0 and a_1



Intervals may Overlap

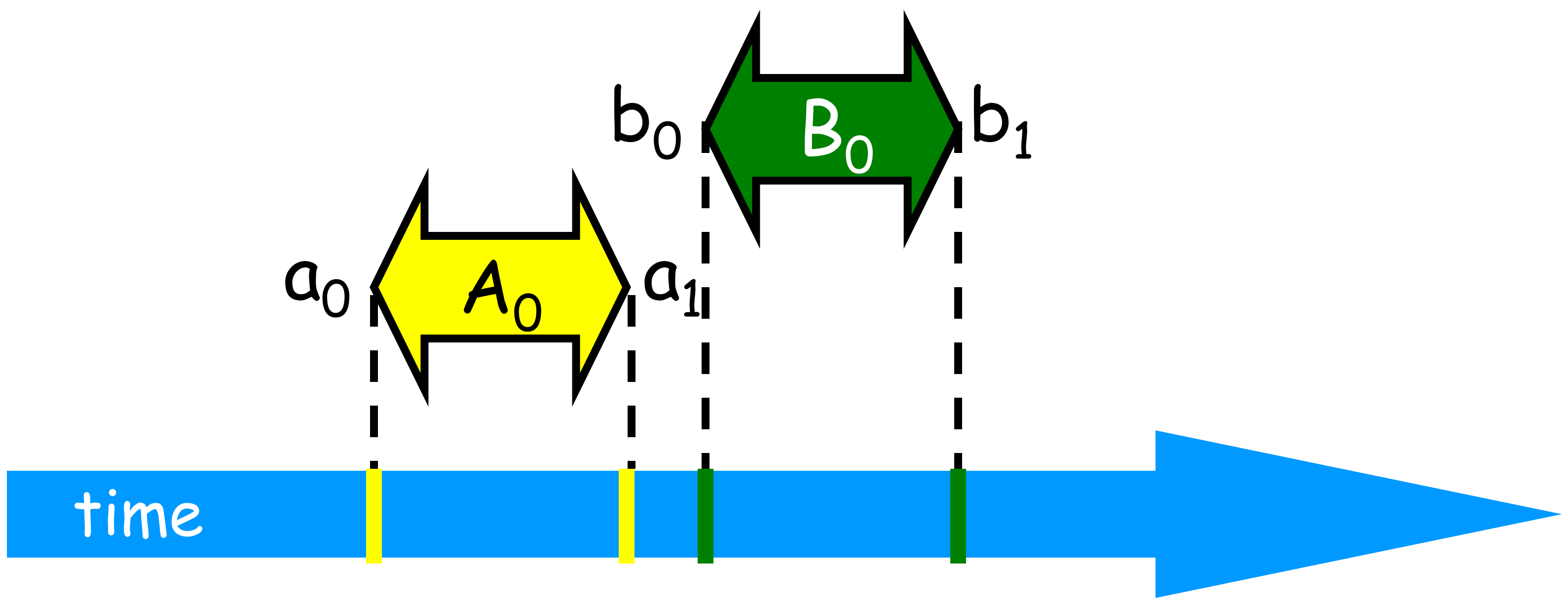


Intervals may be Disjoint

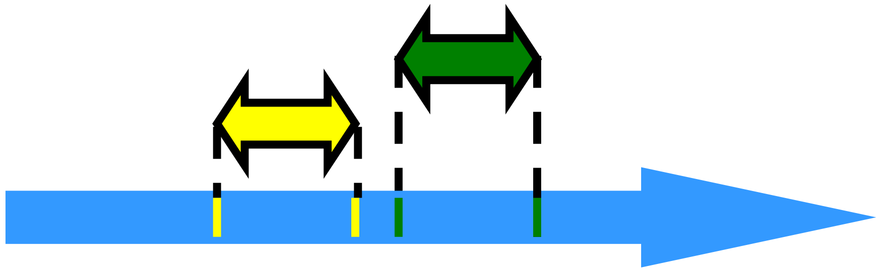


Precedence

Interval A_0 precedes interval B_0

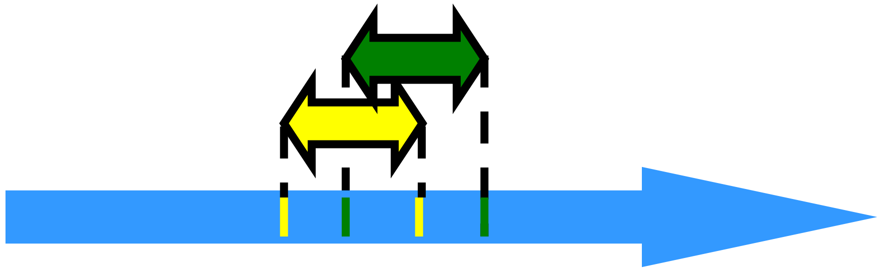


Precedence



- Notation: $A_0 \rightarrow B_0$
- Formally,
  - End event of A_0 before start event of B_0
  - Also called "happens before" or "precedes"

Precedence Ordering



- Never true that A → A
- If A → B then not true that B → A
- If A → B & B → C then A → C
- Funny thing: A → B & B → A might both be false!

Partial Orders

(you may know this already)

- Irreflexive:
  - Never true that $A \rightarrow A$
- Antisymmetric:
  - If $A \rightarrow B$ then not true that $B \rightarrow A$
- Transitive:
  - If $A \rightarrow B$ & $B \rightarrow C$ then $A \rightarrow C$

Total Orders

(you may know this already)

- Also
  - Irreflexive
  - Antisymmetric
  - Transitive
- Except that for every distinct A, B,
  - Either A → B or B → A

Implementing a Counter

```
public class Counter {
  private long value;

  public long getAndIncrement() {
    temp  = value;
    value = temp + 1;
    return temp;
  }
}
```

Make these steps *indivisible* using locks

Locks (Mutual Exclusion)

```
public interface Lock {

 public void lock();

 public void unlock();
}
```

Locks (Mutual Exclusion)

```
public interface Lock {

 public void lock();

 public void unlock();
}
```

acquire lock

Locks (Mutual Exclusion)

```
public interface Lock {

 public void lock();

 public void unlock();
}
```

acquire lock

release lock

Using Locks

```
public class Counter {
  private long value;
  private Lock lock;
  public long getAndIncrement() {
   lock.lock();
   try {
    int temp = value;
    value = value + 1;
   } finally {
     lock.unlock();
   }
   return temp;
  }}
```

Using Locks

```
public class Counter {
  private long value;
  private Lock lock;
  public long getAndIncrement() {
   lock.lock();
   try {
    int temp = value;
    value = value + 1;
   } finally {
     lock.unlock();
   }
   return temp;
  }}
```

acquire Lock

Using Locks

```
public class Counter {
  private long value;
  private Lock lock;
  public long getAndIncrement() {
   lock.lock();
   try {
    int temp = value;
    value = value + 1;
   } finally {
     lock.unlock();
   }
   return temp;
  }}
```

Release lock
(no matter what)

Using Locks

```
public class Counter {
  private long value;
  private Lock lock;
  public long getAndIncrement() {
   lock.lock();
   try {
    int temp = value;
    value = value + 1;
   } finally {
     lock.unlock();
   }
   return temp;
  }}
```

Critical section

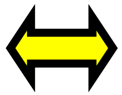
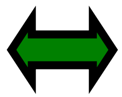
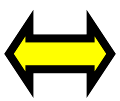
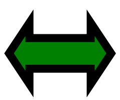
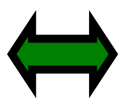
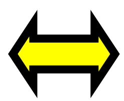
Mutual Exclusion

- Let $CS_i^k \Leftrightarrow$ be thread i's k-th critical section execution

Mutual Exclusion

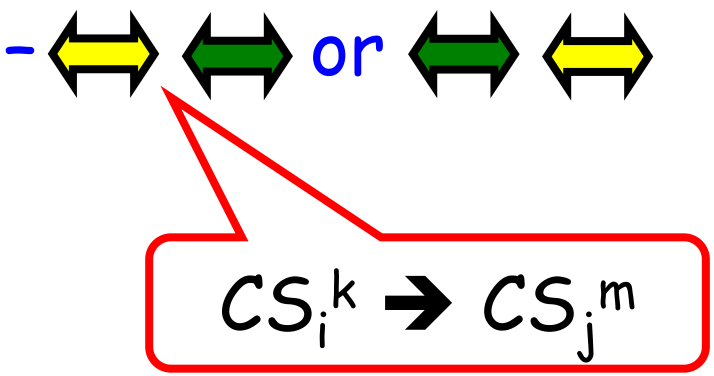
- Let $CS_i^k \Leftrightarrow$ be thread i's k-th critical section execution
- And $CS_j^m \Leftrightarrow$ be thread j's m-th critical section execution

Mutual Exclusion

- Let CS_i^k  be thread i's k-th critical section execution
- And CS_j^m  be j's m-th execution
- Then either
  -   or  

Mutual Exclusion

- Let $CS_i^k \Leftrightarrow$ be thread i's k-th critical section execution
- And $CS_j^m \Leftrightarrow$ be j's m-th execution
- Then either



Mutual Exclusion

- Let $CS_i^k \Leftrightarrow$ be thread i's k-th critical section execution
- And $CS_j^m \Leftrightarrow$ be j's m-th execution
- Then either
  - $\Leftrightarrow \Leftrightarrow$ or $\Leftrightarrow \Leftrightarrow$

$CS_i^k \rightarrow CS_j^m$

$CS_j^m \rightarrow CS_i^k$

Deadlock-Free

- If some thread calls **lock()**
  - And never returns
  - Then other threads must complete **lock()** and **unlock()** calls infinitely often
- System as a whole makes progress
  - Even if individuals starve

Starvation-Free

- If some thread calls lock()
  - It will eventually return
- Individual threads make progress

Two-Thread Conventions

```
class … implements Lock {
  …
  // thread-local index, 0 or 1
  public void lock() {
    int i = ThreadID.get();
    int j = 1 - i;
  …
  }
}
```

Two-Thread Conventions

```
class … implements Lock {
  …
  // thread-local index, 0 or 1
  public void lock() {
    int i = ThreadID.get();
    int j = 1 - i;
  …
  }
}
```

Henceforth: i is current thread, j is other thread

LockOne

```
class LockOne implements Lock {
private boolean[] flag =
                               new boolean[2];
public void lock() {
  flag[i] = true;
  while (flag[j]) {}
 }
```

LockOne

```
class LockOne implements Lock {
private boolean[] flag =
                        new boolean[2];
public void lock() {
  flag[i] = true;
  while (flag[j]) {}
 }
```

Set my flag

LockOne

```
class LockOne implements Lock {
private boolean[] flag =
                                    new boolean[2];
public void lock() {
  flag[i] = true;
  while (flag[j]) {}
  }
```

Set my flag

Wait for other flag to go false

LockOne Satisfies Mutual Exclusion

- Assume CS_A^j overlaps CS_B^k
- Consider each thread's last (j-th and k-th) read and write in the lock() method before entering
- Derive a contradiction

From the Code

- $\text{write}_A(\text{flag[A]=true}) \rightarrow \text{read}_A(\text{flag[B]==false}) \rightarrow CS_A$
- $\text{write}_B(\text{flag[B]=true}) \rightarrow \text{read}_B(\text{flag[A]==false}) \rightarrow CS_B$

```
class LockOne implements Lock {
…
public void lock() {
  flag[i] = true;
  while (flag[j]) {}
 }
```

From the Assumption

- **read_A(flag[B]==false) → write_B(flag[B]=true)**

- **read_B(flag[A]==false) → write_A(flag[B]=true)**

Combining

- Assumptions:
  - $read_A(flag[B]==false) \rightarrow write_B(flag[B]=true)$
  - $read_B(flag[A]==false) \rightarrow write_A(flag[A]=true)$
- From the code
  - $write_A(flag[A]=true) \rightarrow read_A(flag[B]==false)$
  - $write_B(flag[B]=true) \rightarrow read_B(flag[A]==false)$

Combining

- Assumptions:
  - $read_A(flag[B]==false) \rightarrow write_B(flag[B]=true)$
  - $read_B(flag[A]==false) \rightarrow write_A(flag[A]=true)$
- From the code
  - $write_A(flag[A]=true) \rightarrow read_A(flag[B]==false)$
  - $write_B(flag[B]=true) \rightarrow read_B(flag[A]==false)$

Combining

- Assumptions:
  - $\text{read}_A(\text{flag}[B]{=}{=}\text{false}) \rightarrow \text{write}_B(\text{flag}[B]{=}\text{true})$
  - $\text{read}_B(\text{flag}[A]{=}{=}\text{false}) \rightarrow \text{write}_A(\text{flag}[A]{=}\text{true})$
- From the code
  - $\text{write}_A(\text{flag}[A]{=}\text{true}) \rightarrow \text{read}_A(\text{flag}[B]{=}{=}\text{false})$
  - $\text{write}_B(\text{flag}[B]{=}\text{true}) \rightarrow \text{read}_B(\text{flag}[A]{=}{=}\text{false})$

Combining

- Assumptions:
  - $read_A(flag[B]==false) \rightarrow write_B(flag[B]=true)$
  - $read_B(flag[A]==false) \rightarrow write_A(flag[A]=true)$
- ~~From the code~~
  - $write_A(flag[A]=true) \rightarrow read_A(flag[B]==false)$
  - $write_B(flag[B]=true) \rightarrow read_B(flag[A]==false)$

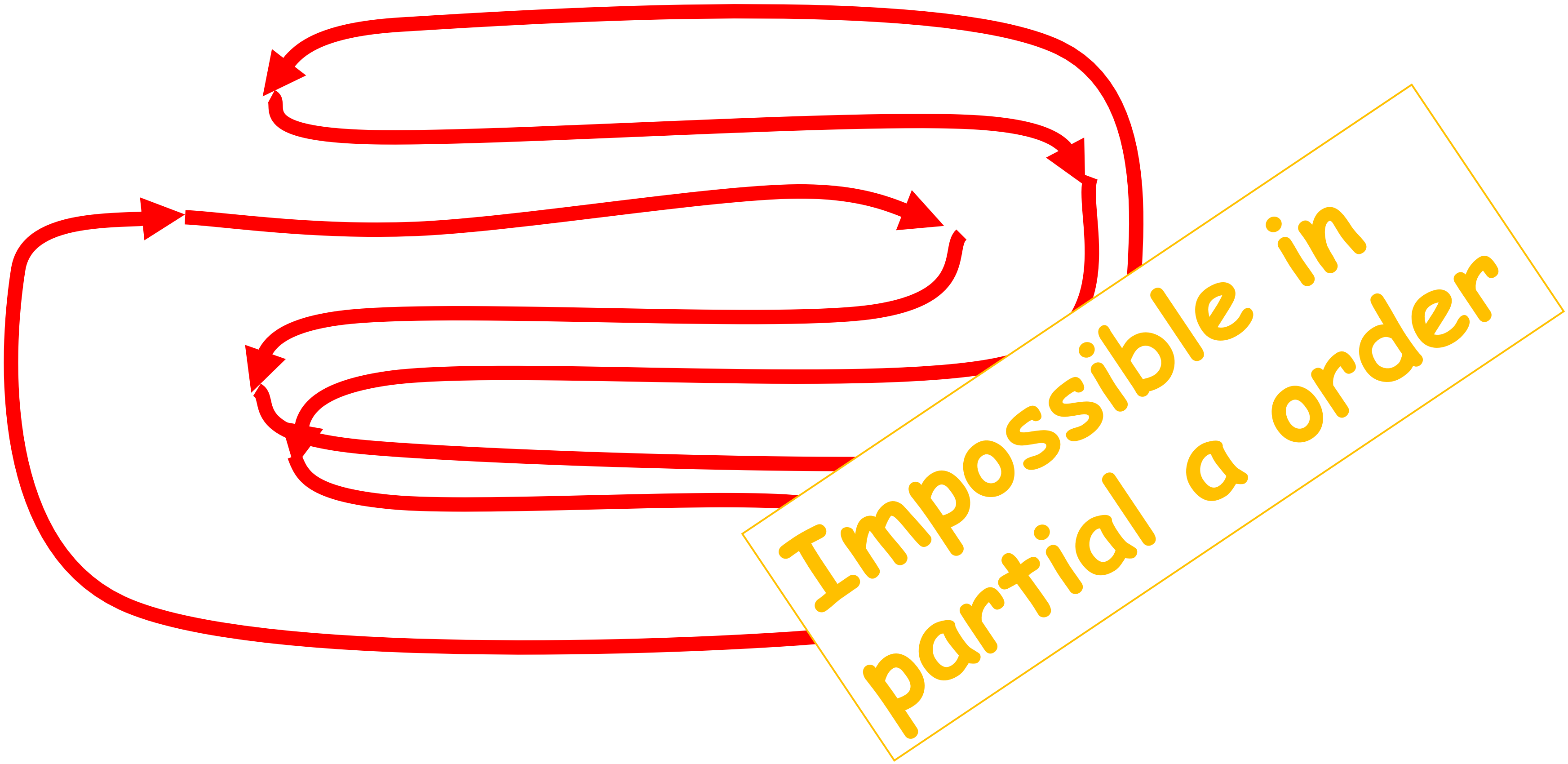
Combining

- Assumptions:
  - $read_A(flag[B]==false) \rightarrow write_B(flag[B]=true)$
  - $read_B(flag[A]==false) \rightarrow write_A(flag[A]=true)$
- From the code
  - $write_A(flag[A]=true) \rightarrow read_A(flag[B]==false)$
  - $write_B(flag[B]=true) \rightarrow read_B(flag[A]==false)$

Combining

- Assumptions:
  - read$_A$(flag[B]==false) $\rightarrow$ write$_B$(flag[B]=true)
  - read$_B$(flag[A]==false) $\rightarrow$ write$_A$(flag[A]=true)
- From the code
  - write$_A$(flag[A]=true) $\rightarrow$ read$_A$(flag[B]==false)
  - write$_B$(flag[B]=true) $\rightarrow$ read$_B$(flag[A]==false)

Cycle!



Deadlock Freedom

- LockOne Fails deadlock-freedom
  - Concurrent execution can deadlock

```
flag[i] = true;     flag[j] = true;
while (flag[j]){}   while (flag[i]){}
```

  - Sequential executions OK

LockTwo

```
public class LockTwo implements Lock {
 private int victim;
 public void lock() {
  victim = i;
  while (victim == i) {};
 }

 public void unlock() {}
}
```

LockTwo

```
public class LockTwo implements Lock {
 private int victim;
 public void lock() {
  victim = i;
  while (victim == i) {};
 }

 public void unlock() {}
}
```

Let other go first

LockTwo

```
public class LockTwo implements Lock {
 private int victim;
 public void lock() {
 victim = i;
  while (victim == i) {};
 }

 public void unlock() {}
}
```

Wait for permission

LockTwo

```
public class Lock2 implements Lock {
 private int victim;
 public void lock() {
  victim = i;
  while (victim == i) {};
 }

 public void unlock() {}
}
```

Nothing to do

LockTwo Claims

- Satisfies mutual exclusion
  - If thread `i` in CS
  - Then `victim == j`
  - Cannot be both 0 and 1
- Not deadlock free
  - Sequential execution deadlocks
  - Concurrent execution does not

```
public void LockTwo() {
  victim = i;
  while (victim == i) {};
 }
```

Peterson's Algorithm

```
public void lock() {
 flag[i] = true;
 victim  = i;
 while (flag[j] && victim == i) {};
}
public void unlock() {
 flag[i] = false;
}
```

Peterson's Algorithm

Announce I'm interested

```
public void lock() {
 flag[i] = true;
 victim  = i;
 while (flag[j] && victim == i) {};
}
public void unlock() {
 flag[i] = false;
}
```

Peterson's Algorithm

```
public void lock() {
 flag[i] = true;
 victim  = i;
 while (flag[j] && victim == i) {};
}
public void unlock() {
 flag[i] = false;
}
```

Announce I'm interested

Defer to other

Peterson's Algorithm

```
public void lock() {
 flag[i] = true;
 victim  = i;
 while (flag[j] && victim == i) {};
}
public void unlock() {
 flag[i] = false;
}
```

Announce I'm interested

Defer to other

Wait while other interested & I'm the victim

Peterson's Algorithm

```
public void lock() {
 flag[i] = true;
 victim  = i;
 while (flag[j] && victim == i) {};
 }
public void unlock() {
 flag[i] = false;
}
```

Announce I'm interested

Defer to other

Wait while other interested & I'm the victim

No longer interested

Mutual Exclusion

```
public void lock() {
  flag[i] = true;
  victim  = i;
  while (flag[j] && victim == i) {};
```

- If thread 0 in critical section,
  - flag[0]=true,
  - victim = 1

- If thread **1** in critical section,
  - **flag[1]=true**,
  - **victim = 0**

Cannot both be true

Deadlock Free

```
public void lock() {
  …
  while (flag[j] && victim == i) {};
```

- Thread blocked
  - only at **`while`** loop
  - only if it is the victim
- One or the other must not be the victim

Starvation Free

- Thread `i` blocked only if `j` repeatedly re-enters so that

  `flag[j] == true` and `victim == i`
- When `j` re-enters
  - it sets `victim` to `j`.
  - So `i` gets in

```
public void lock() {
  flag[i] = true;
  victim    = i;
  while (flag[j] && victim == i) {};
}

public void unlock() {
  flag[i] = false;
}
```

Other Lock Algorithms

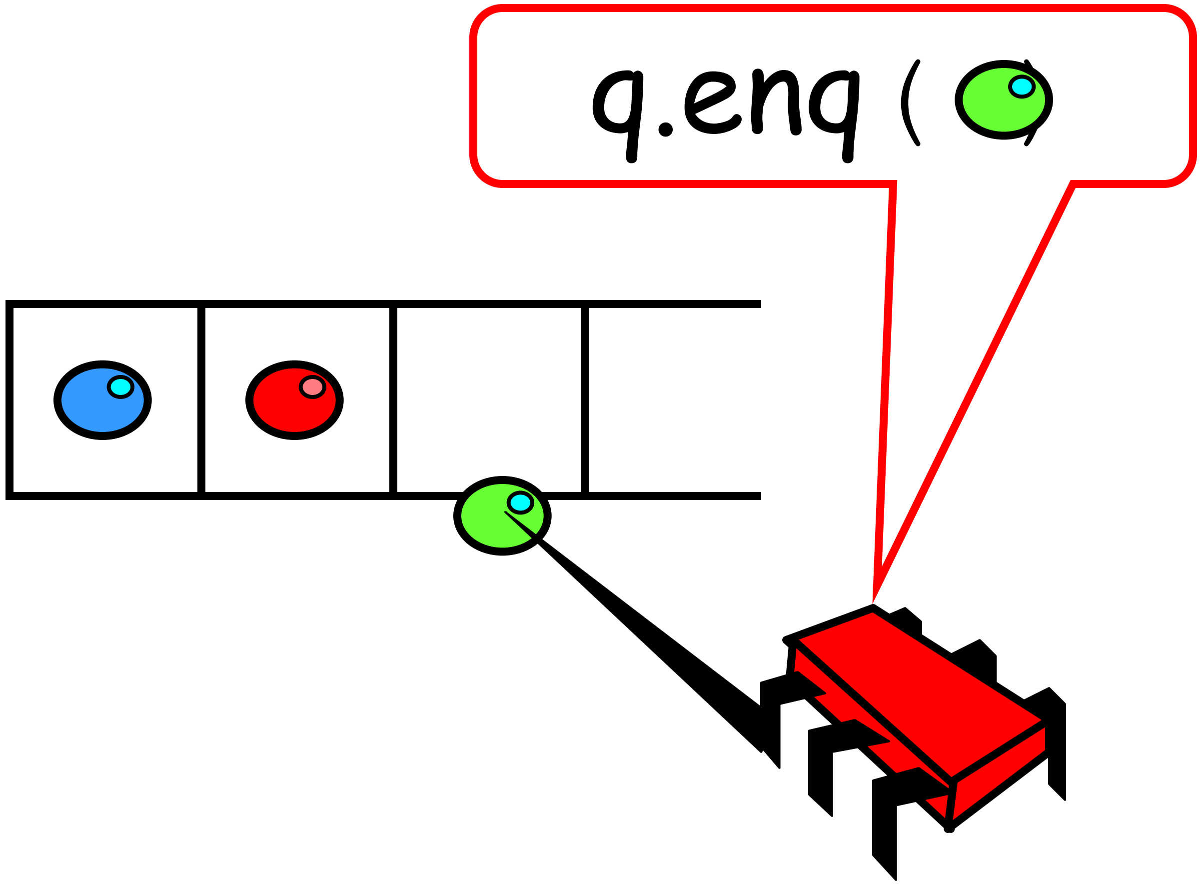
- The Filter Algorithm for *n* Threads
- Bakery Algorithm

Theorem: At least N MRSW (multi-reader/single-writer) registers are needed to solve deadlock-free mutual exclusion.

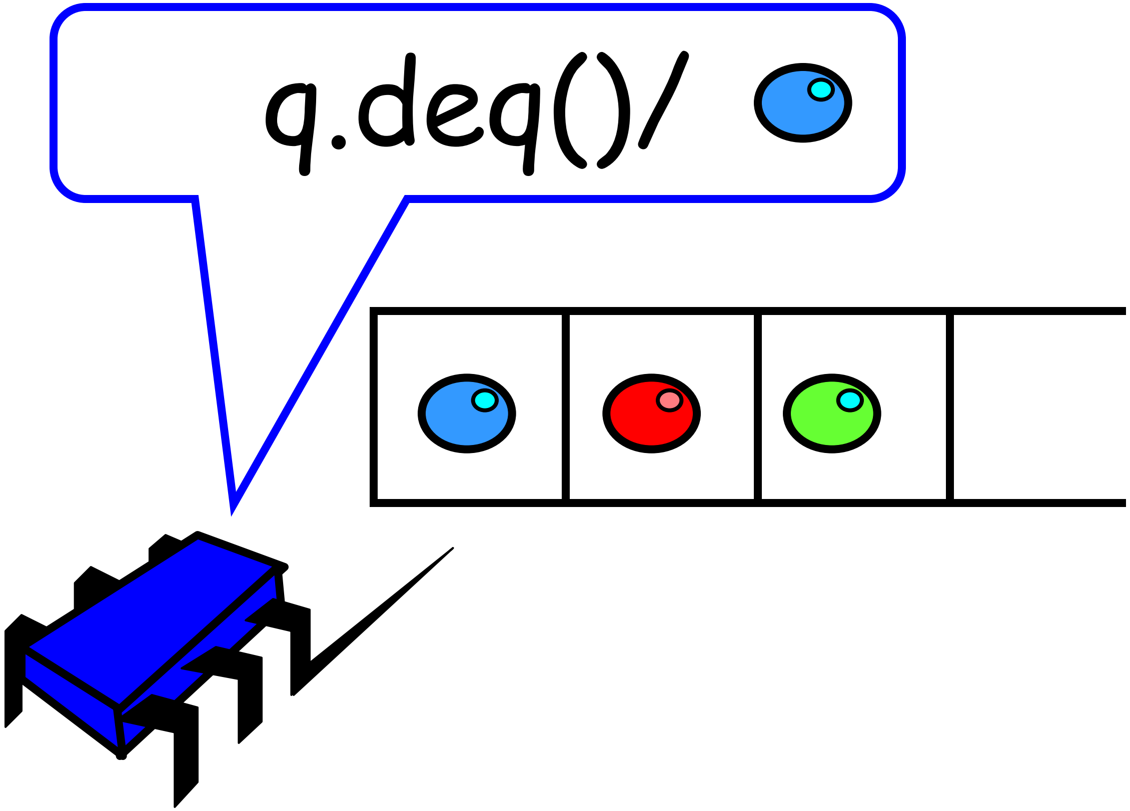
N registers like Flag[]…

Inefficient and Impractical

FIFO Queue: Enqueue Method



FIFO Queue: Dequeue Method

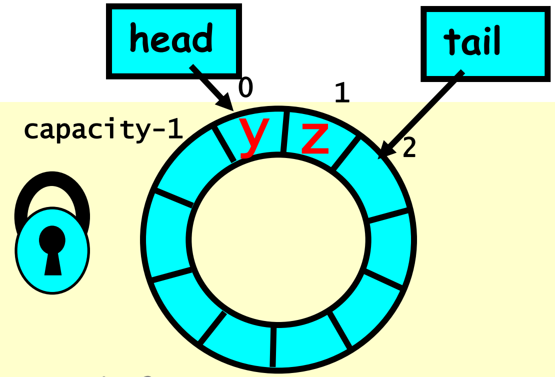


A Lock-Based Queue

```
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];
}
```

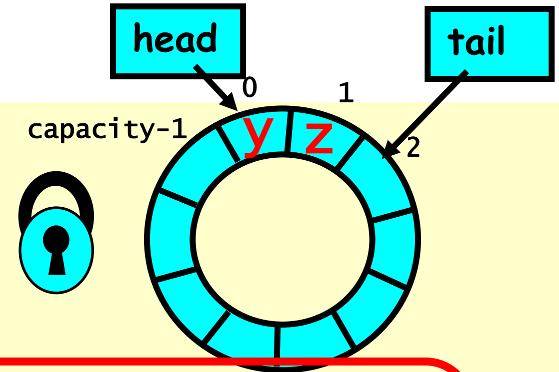
A Lock-Based Queue

```
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

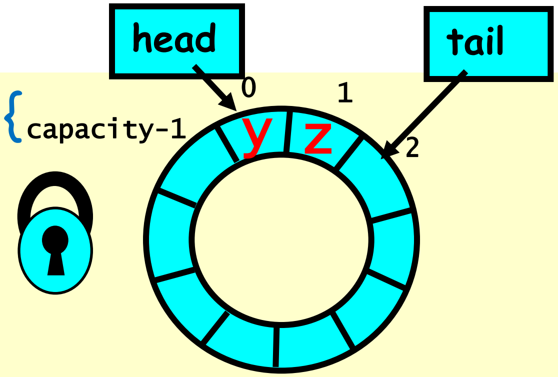



```
class LockBasedQueue<T> {
  int head, tail;
  T[] items;
  Lock lock;
  public LockBasedQueue(int capacity) {
    head = 0; tail = 0;
    lock = new ReentrantLock();
    items = (T[]) new Object[capacity];
}
```

Initially head = tail

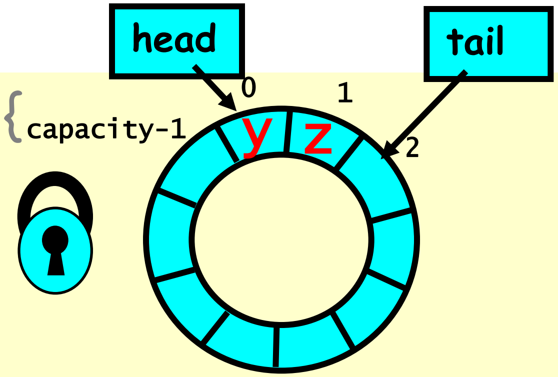
Implementation: Deq

```
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

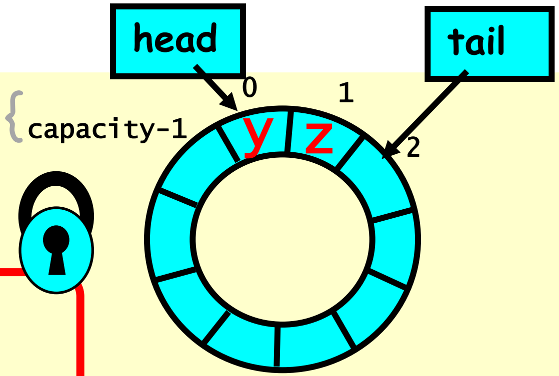
```
public T deq() throws EmptyException {
  lock.lock();
  try {
    if (tail == head)
      throw new EmptyException();
    T x = items[head % items.length];
    head++;
    return x;
  } finally {
    lock.unlock();
  }
}
```



Method calls mutually exclusive

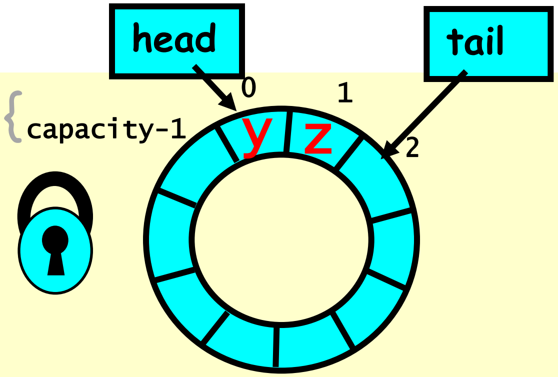
Implementation: Deq

```
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

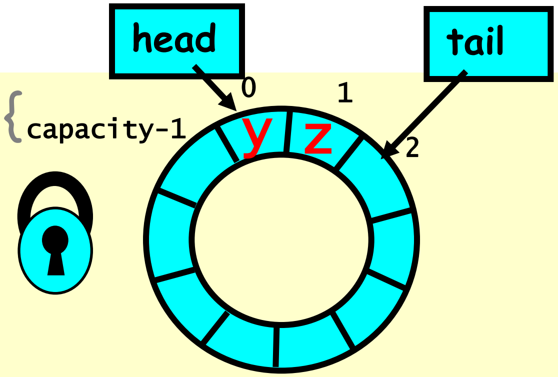



```
public T deq() throws EmptyException {
  lock.lock();
  try {
    if (tail == head)
       throw new EmptyException();
    T x = items[head % items.length];
    head++;
    return x;
  } finally {
    lock.unlock();
  }
}
```

Queue not empty: remove item and update head

Implementation: Deq

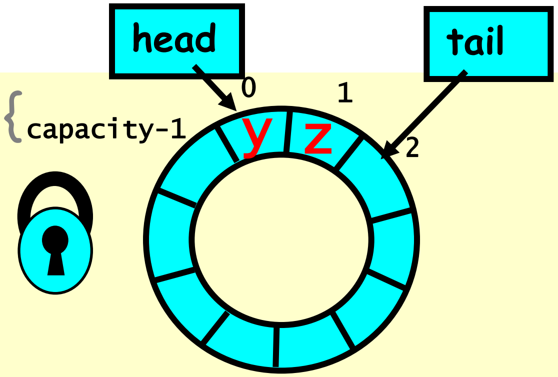
```
public T deq() throws EmptyException {
  lock.lock();
  try {
    if (tail == head)
       throw new EmptyException();
    T x = items[head % items.length];
    head++;
    return x;
  } finally {
    lock.unlock();
  }
}
```



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

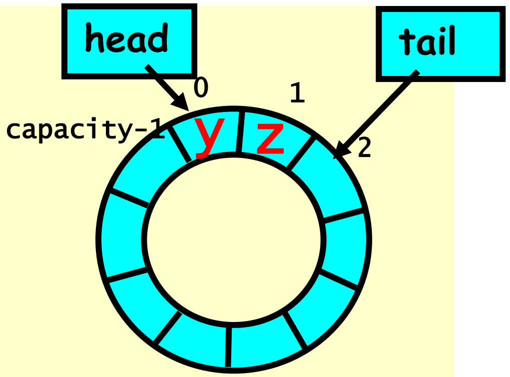
- The same thing without mutual exclusion
- For simplicity, only two threads
  - One thread enq only
  - The other deq only

Wait-free 2-Thread Queue

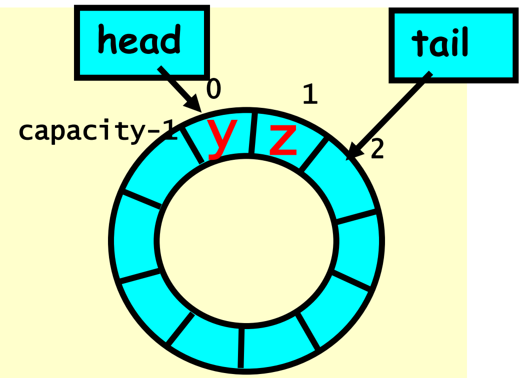
```
public class WaitFreeQueue {

  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-wa
     Item item = items[head %
     return item;
}}
```

Queue is up ... a lock!

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

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 …

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

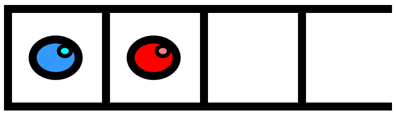
Pre and PostConditions for Dequeue

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

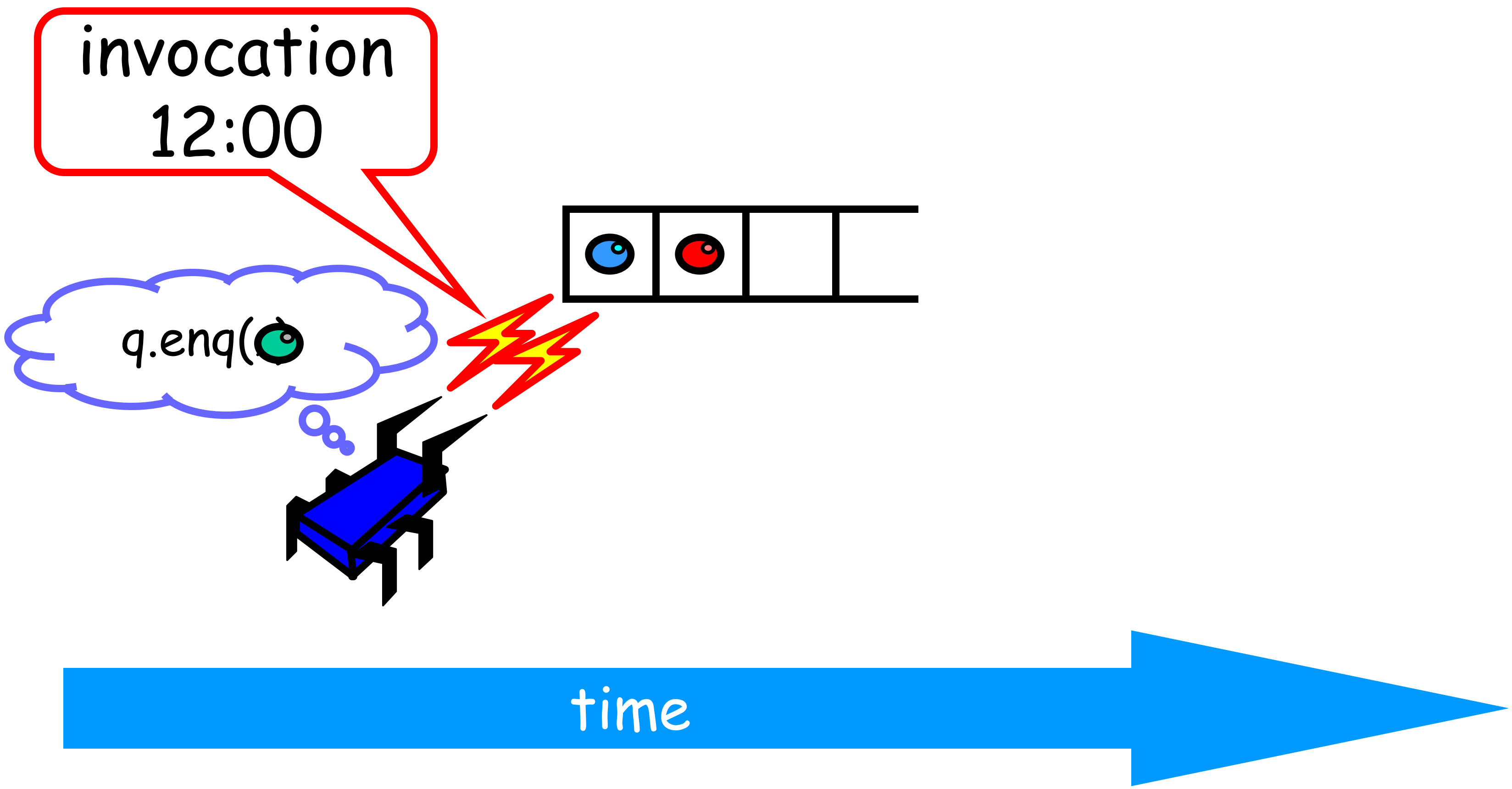
What About Concurrent Specifications ?

- Methods?
- Documentation?
- Adding new methods?

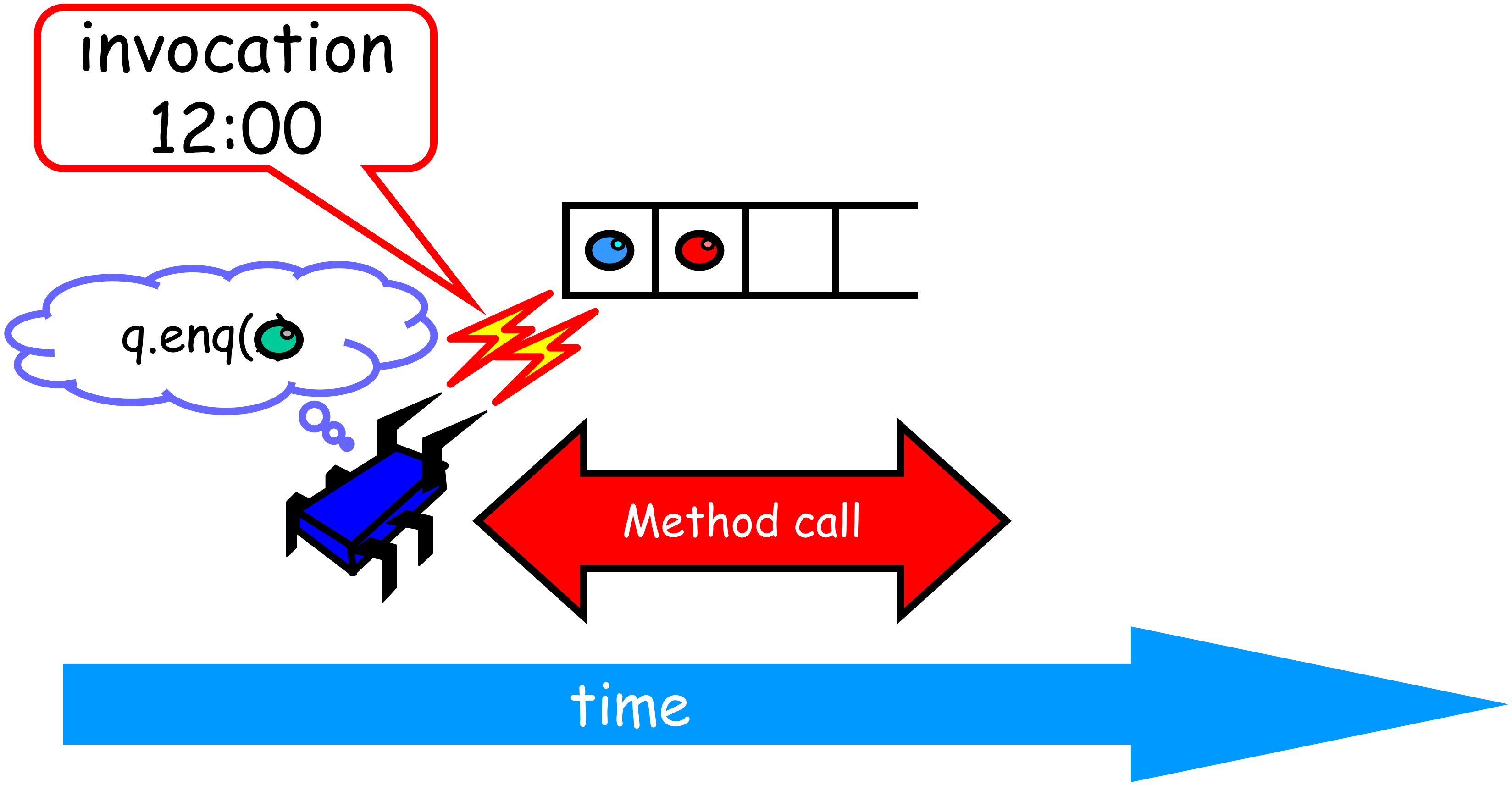
Methods Take Time



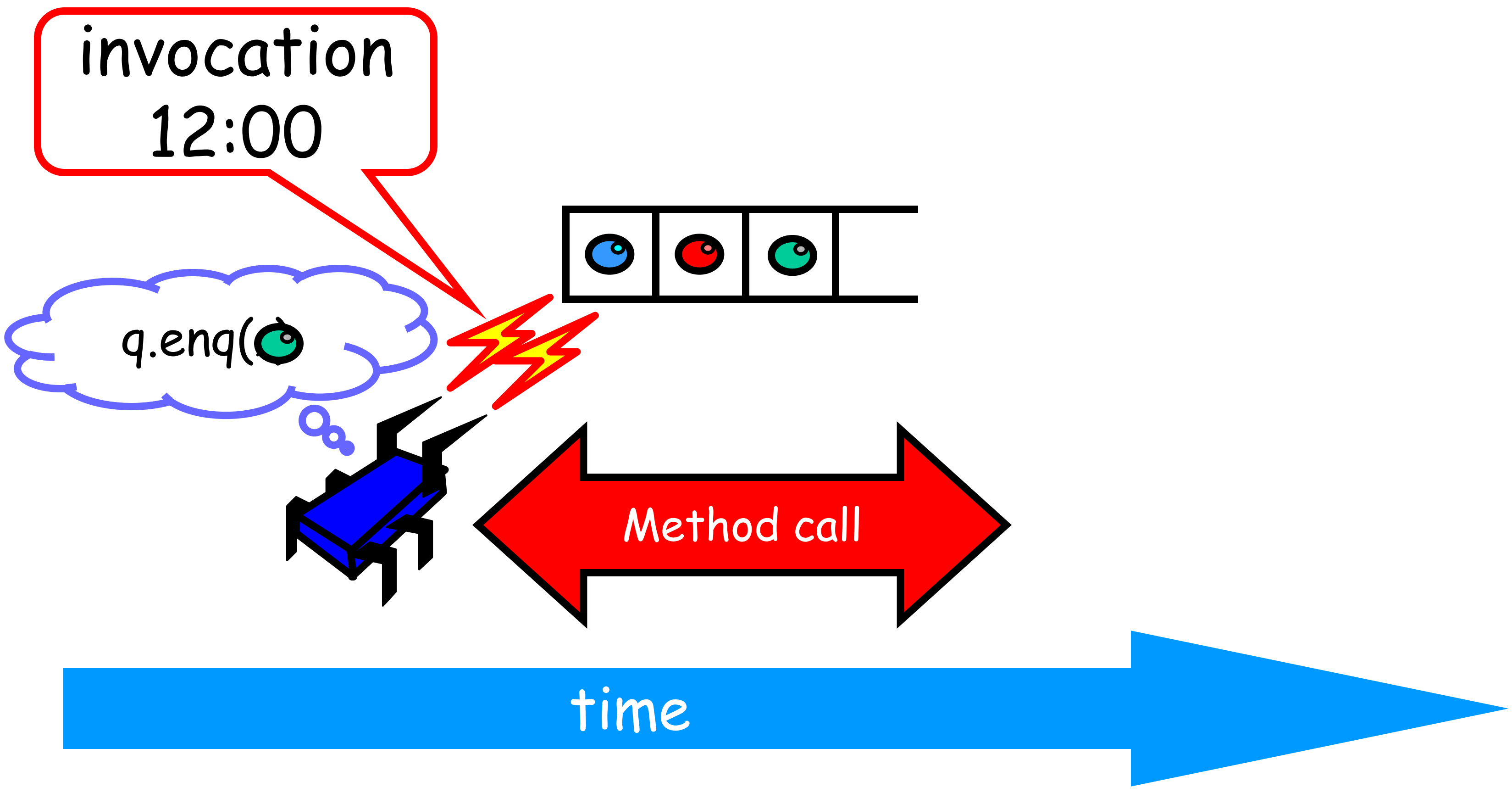
Methods Take Time



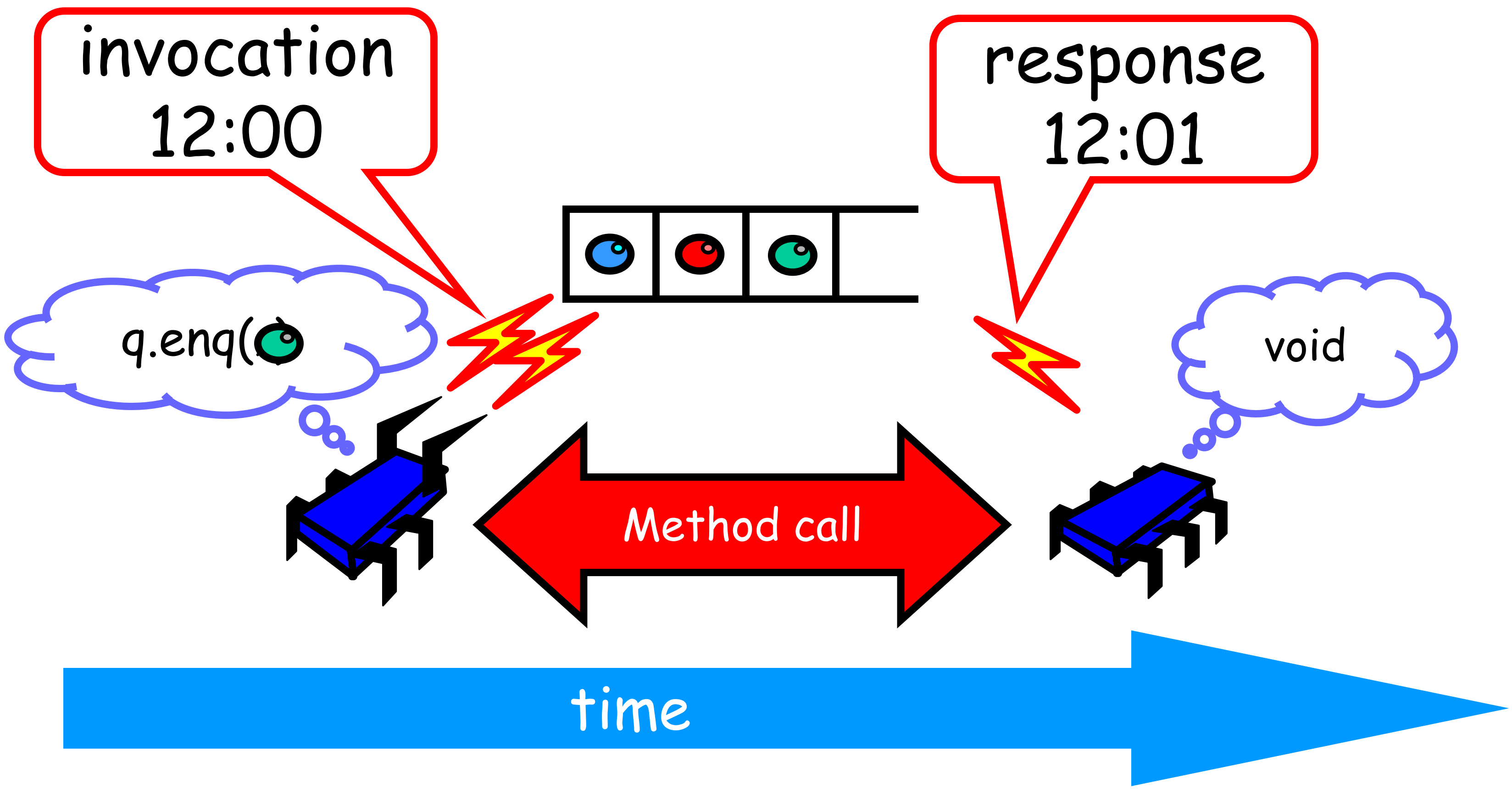
Methods Take Time



Methods Take Time



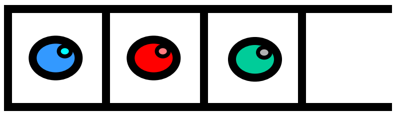
Methods Take Time



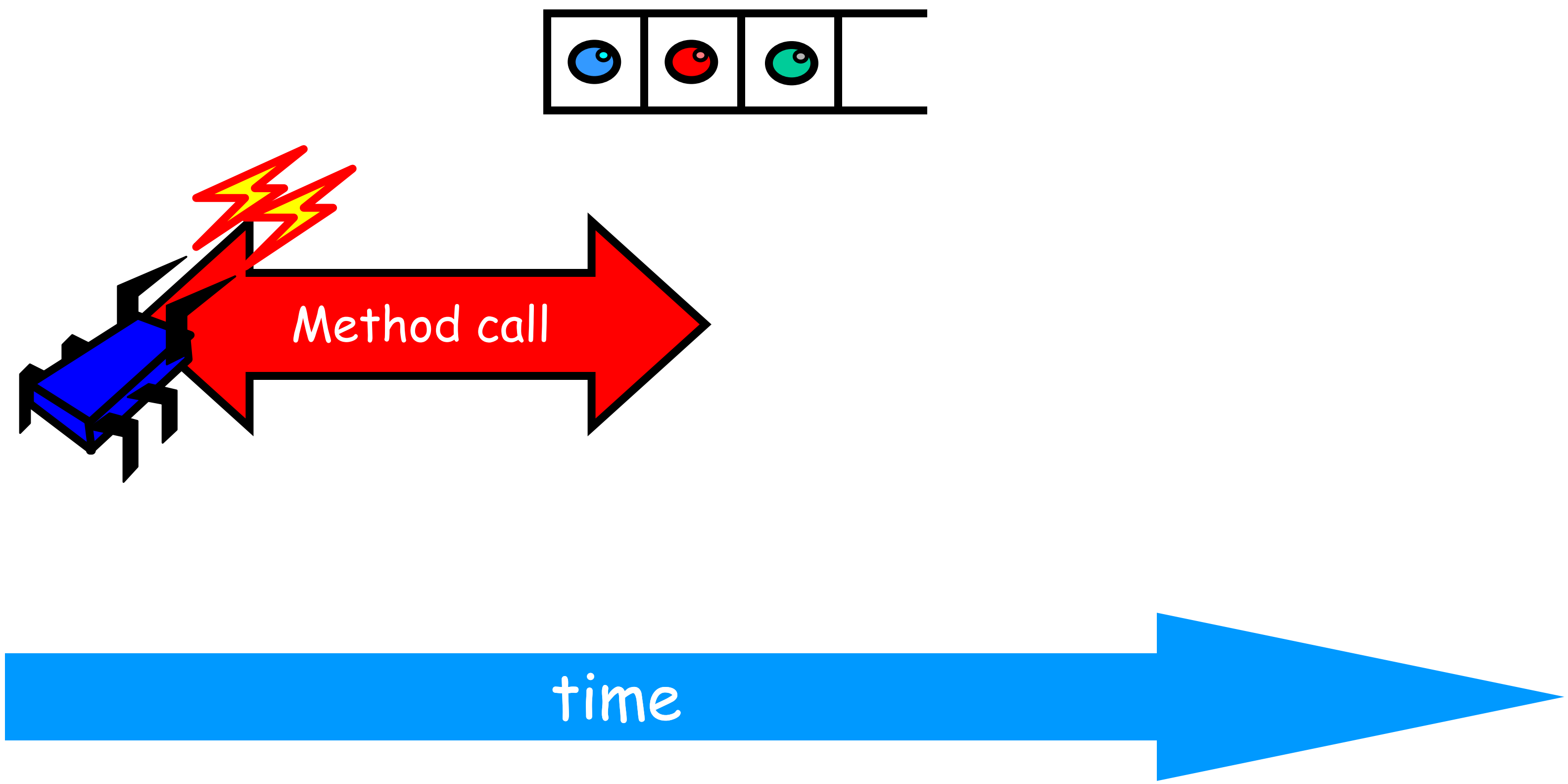
Sequential vs Concurrent

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

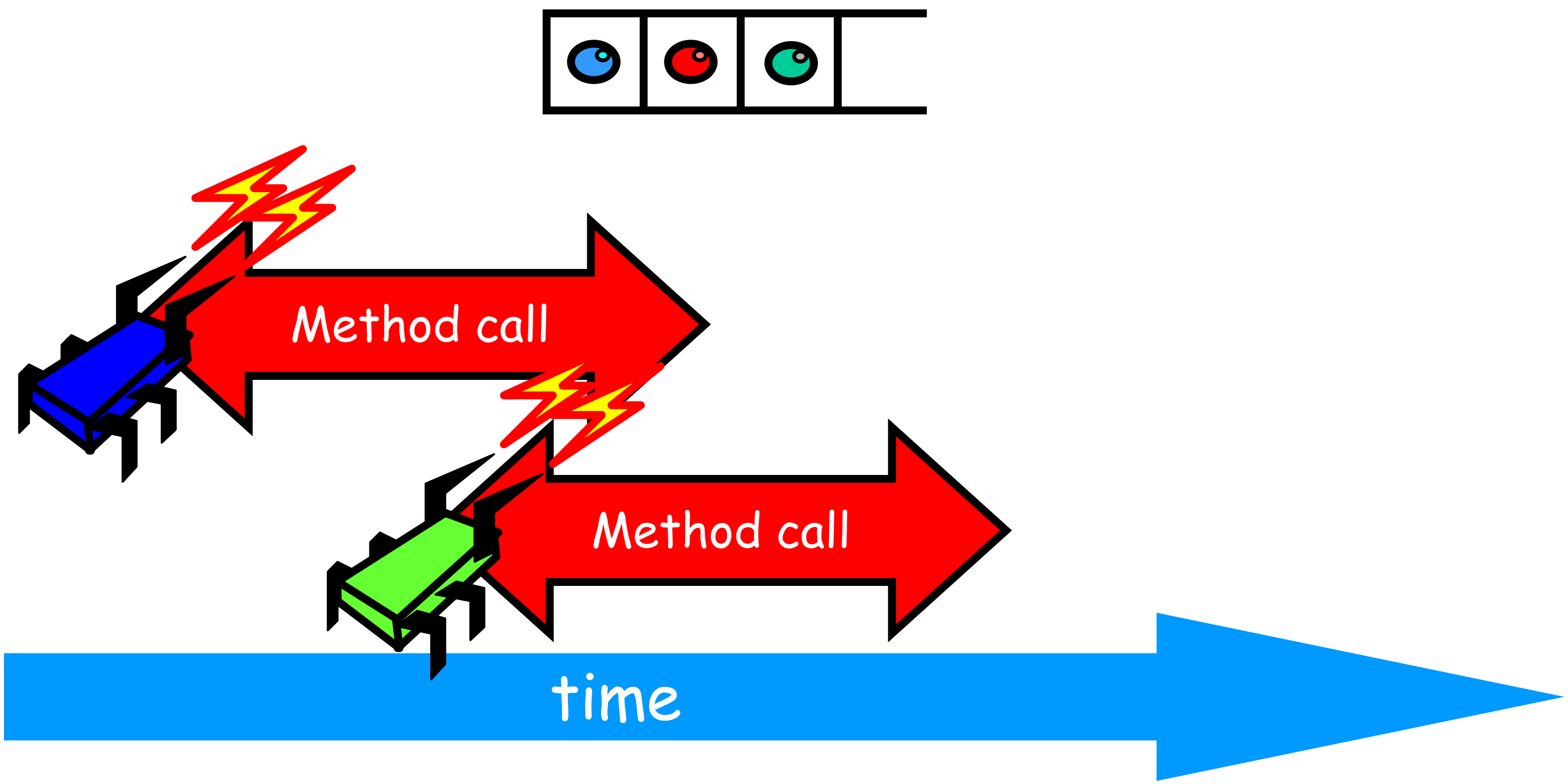
Concurrent Methods Take Overlapping Time



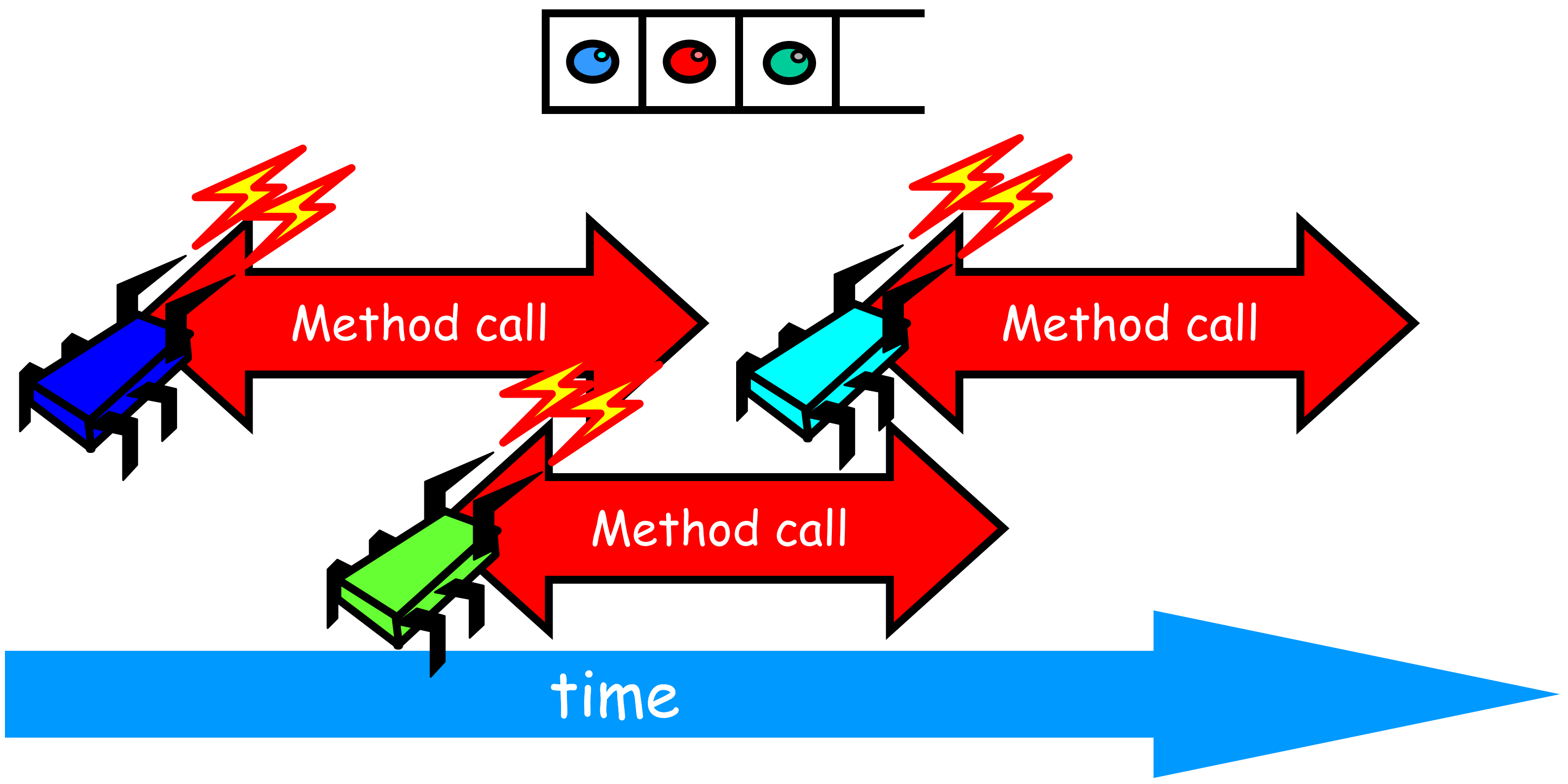
Concurrent Methods Take Overlapping Time



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

Panic!

Intuitively…

```
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();
  }
}
```

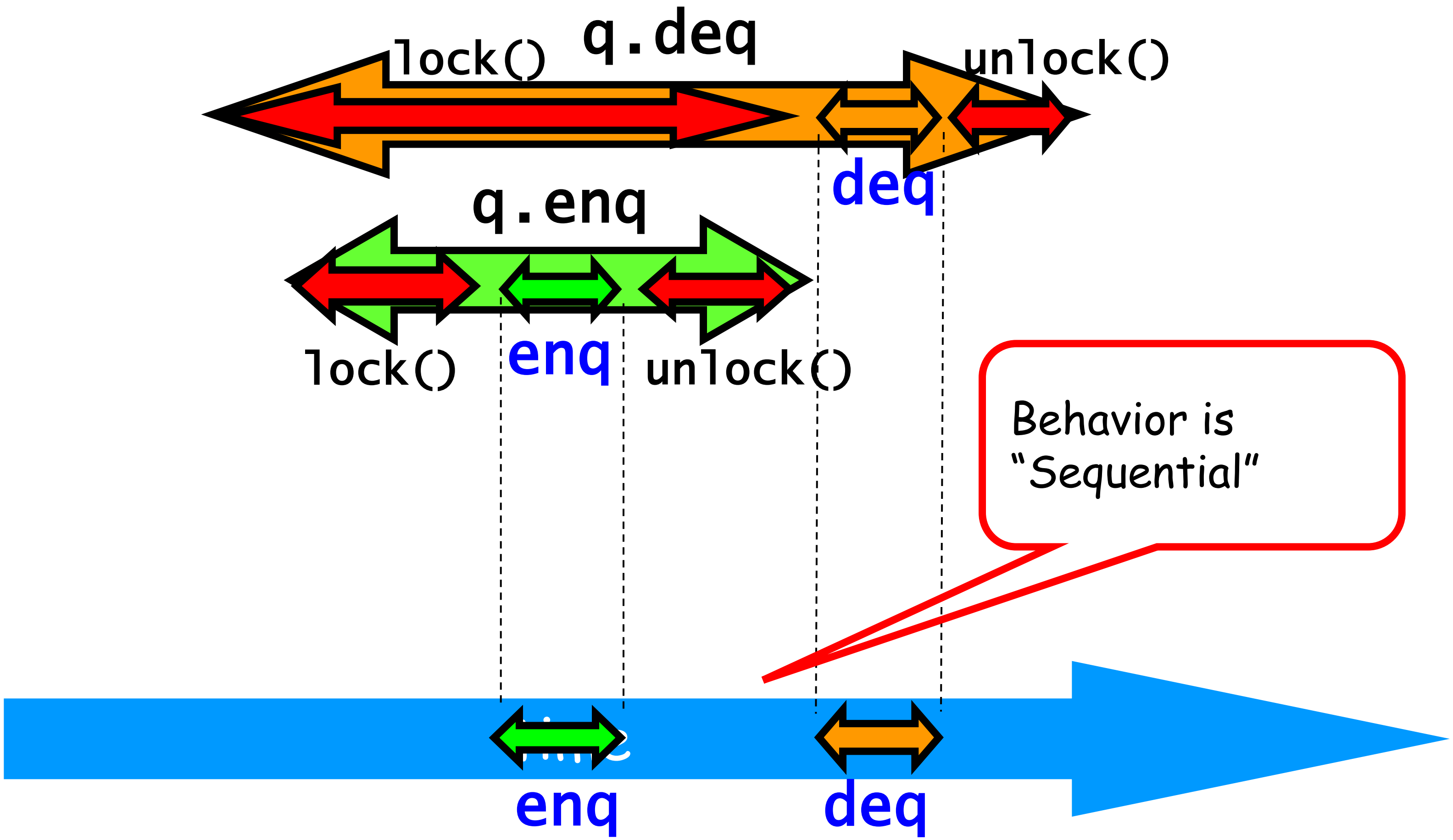
Intuitively…

```
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();
  }
}
```

All modifications of queue are done mutually exclusive

Intuitively

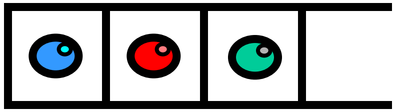
Lets capture the idea of describing the concurrent via the sequential



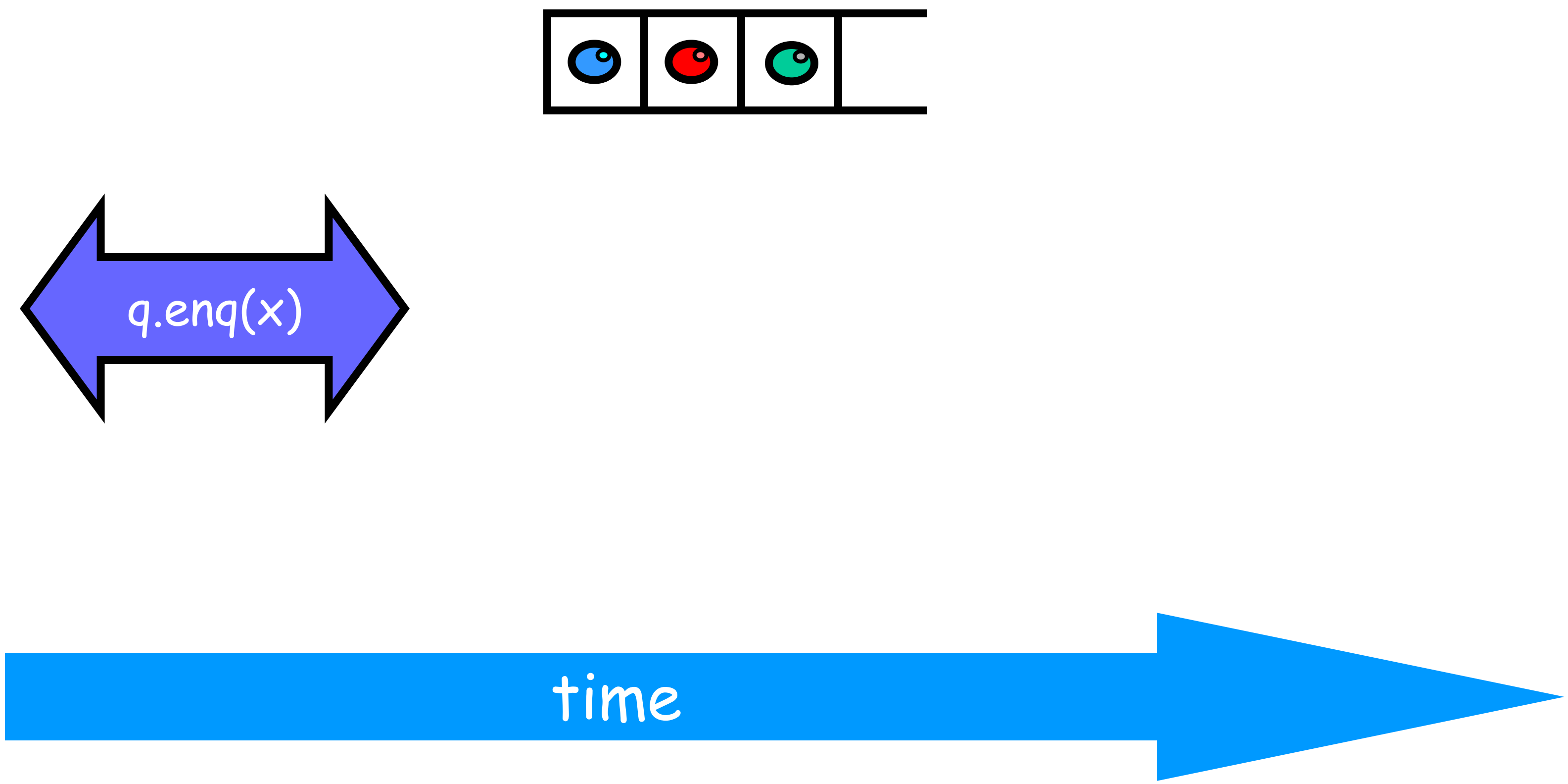
Is it really about the object?

- Each method should
  - “take effect”
  - Instantaneously
  - Between invocation and response events
- Object is correct if this “sequential” behavior is correct
- A linearizable object: one all of whose possible executions are linearizable

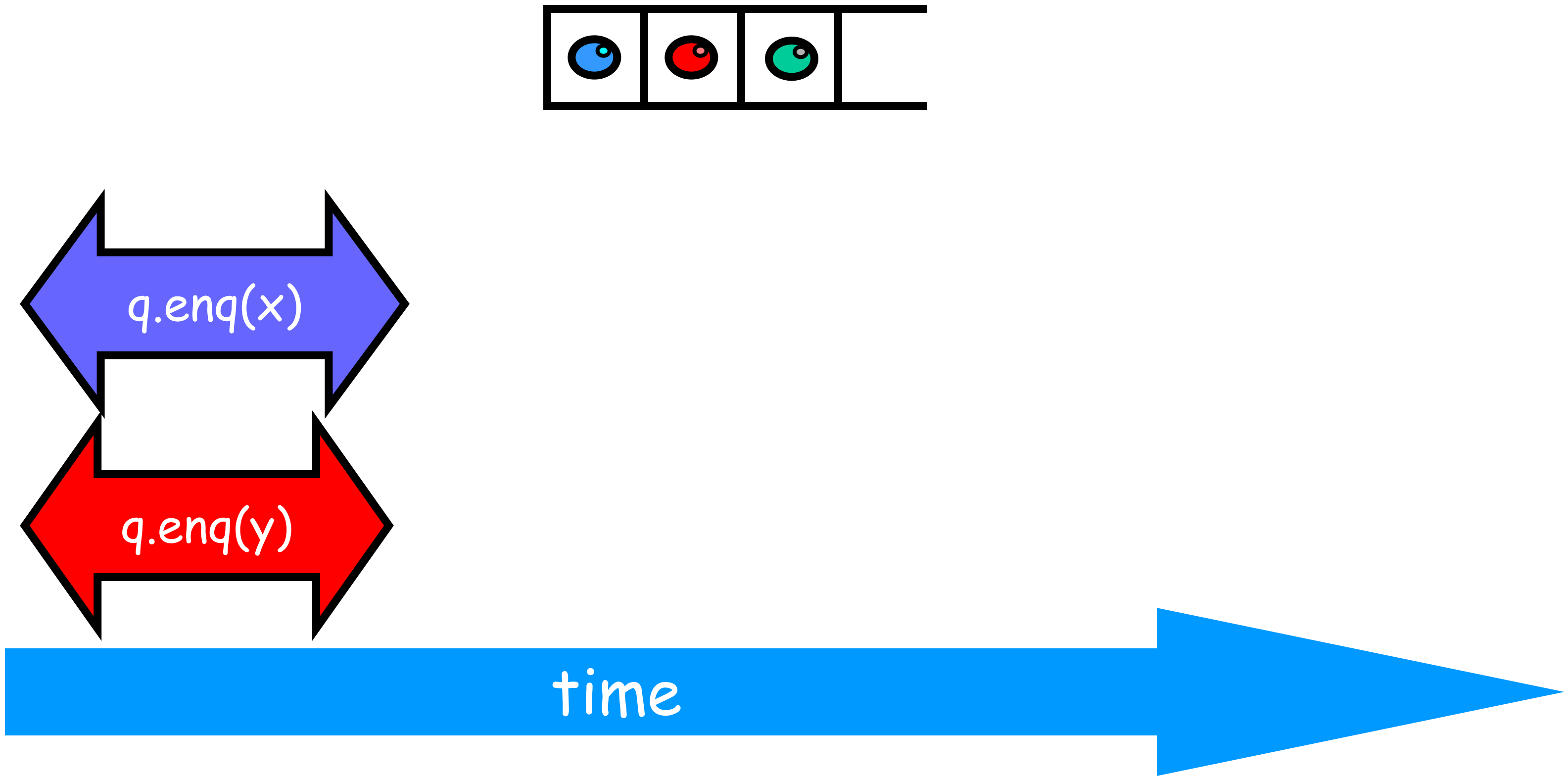
Example



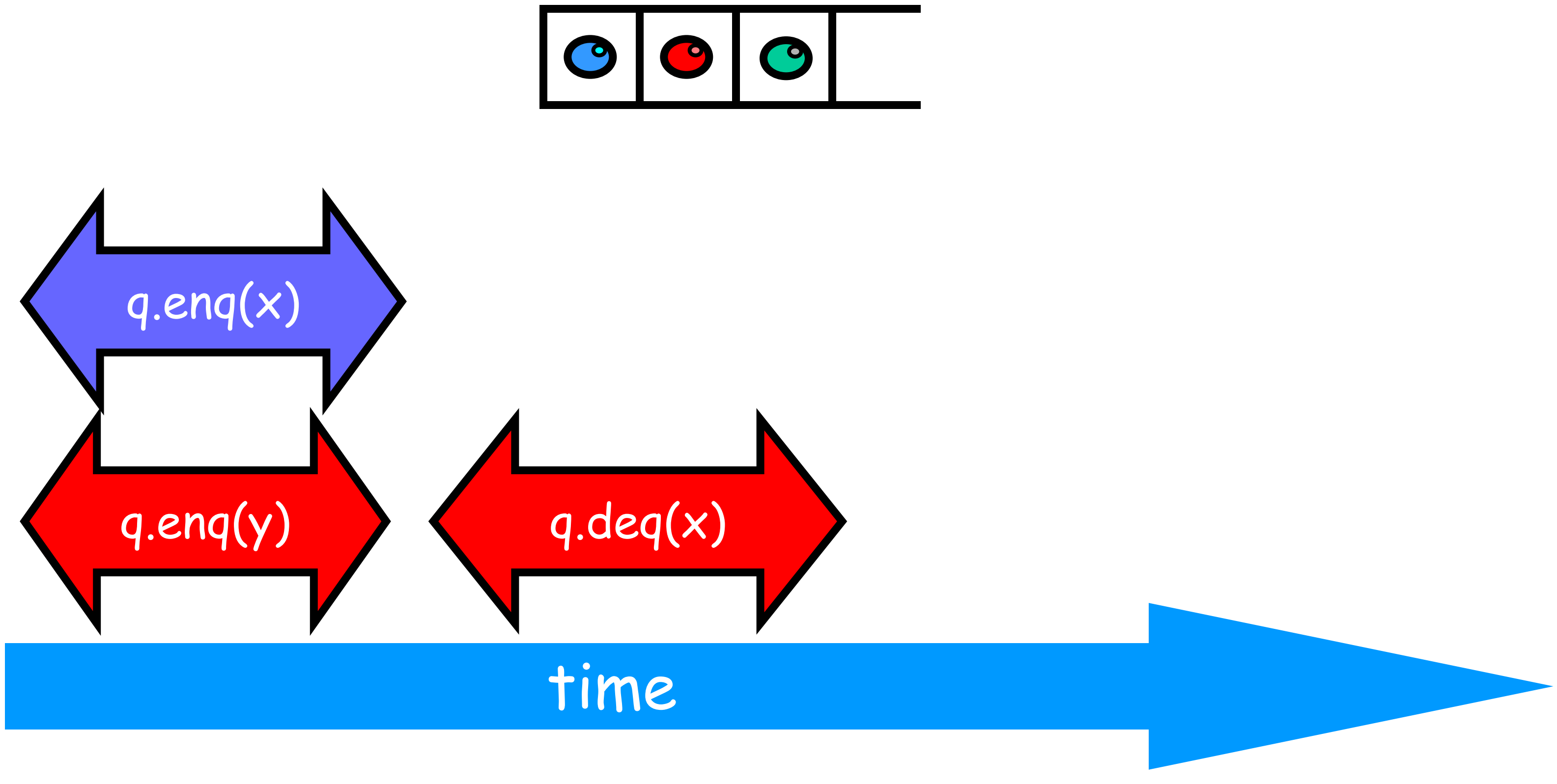
Example



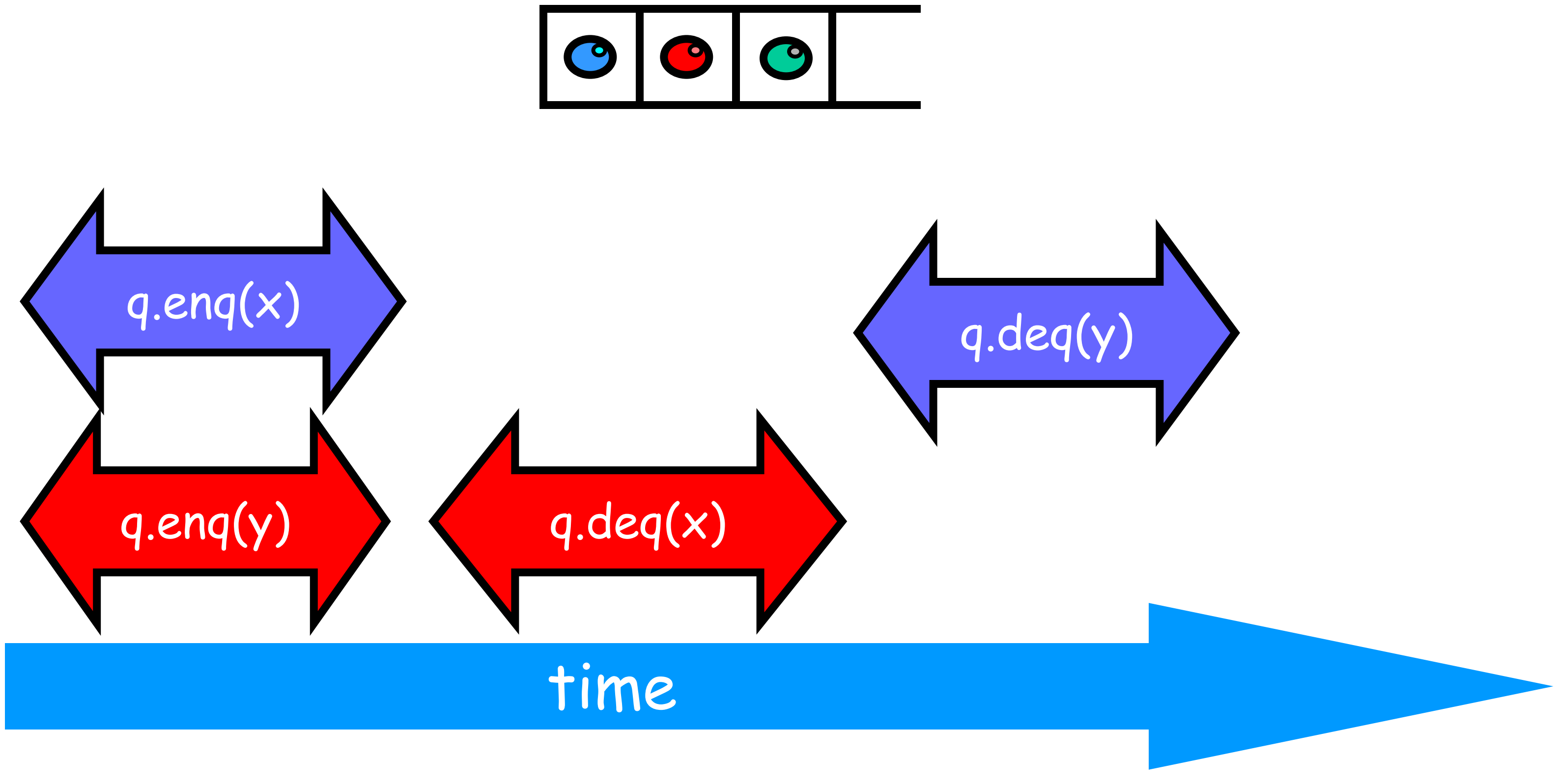
Example



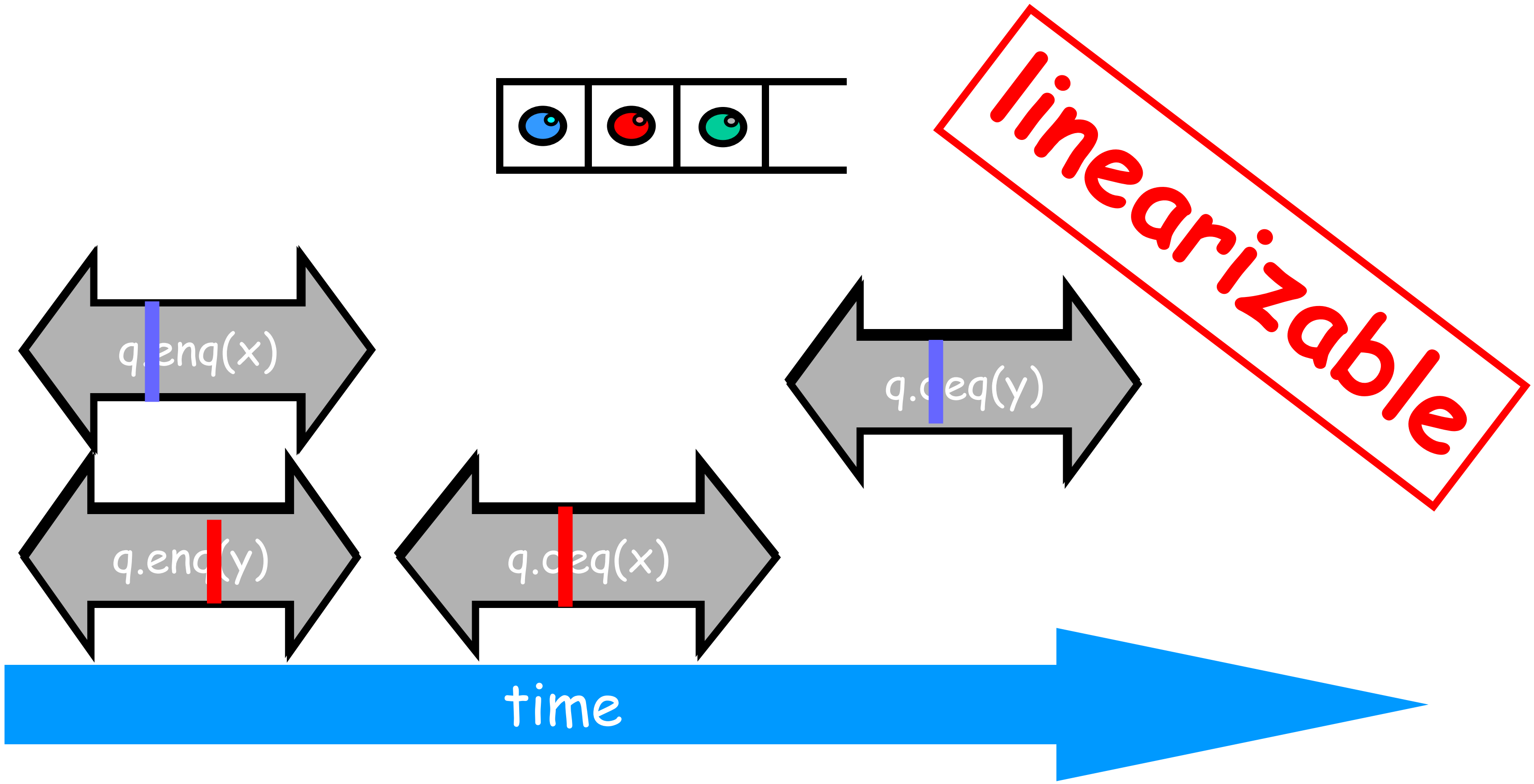
Example



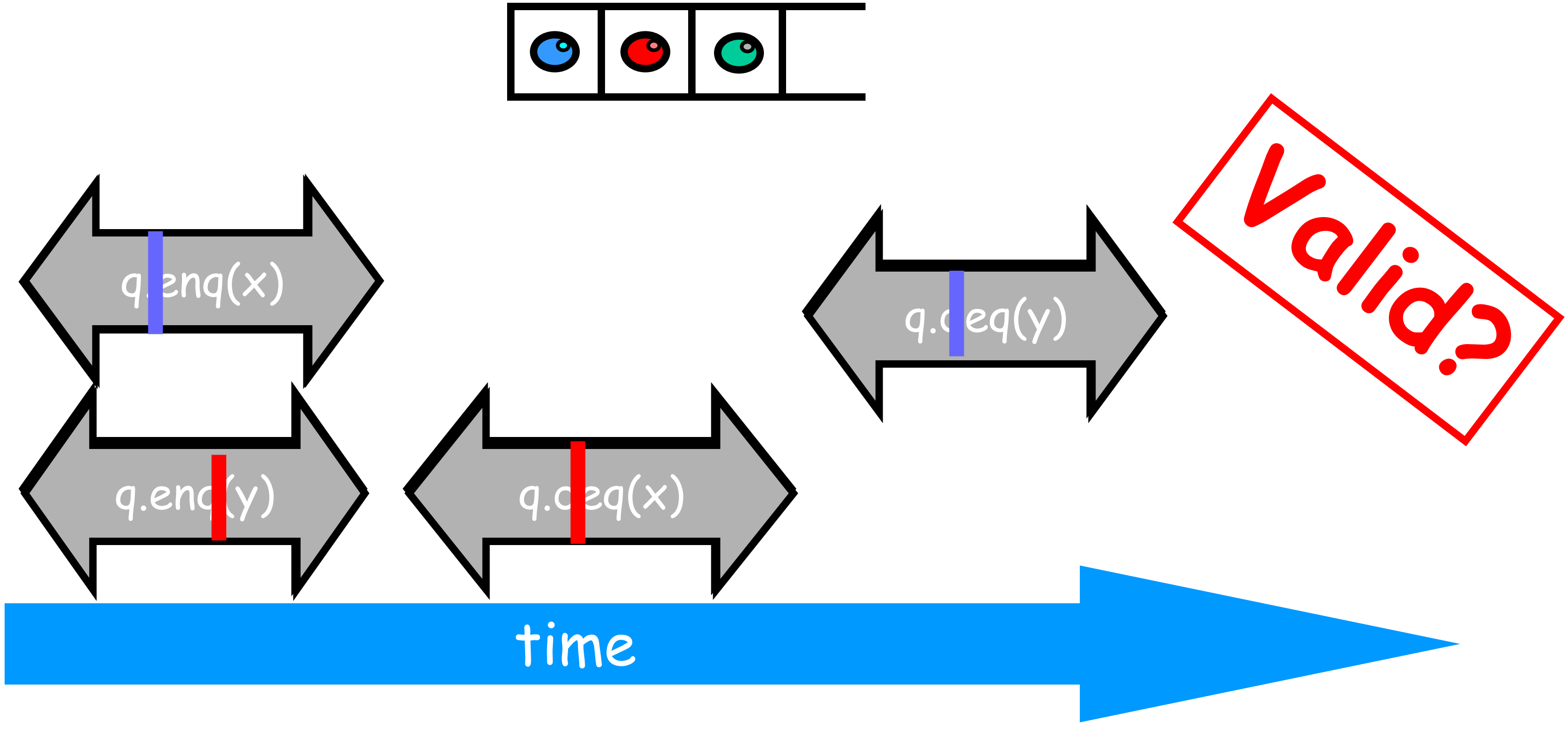
Example



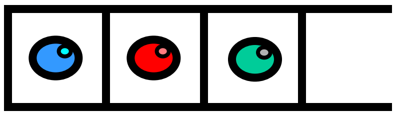
Example



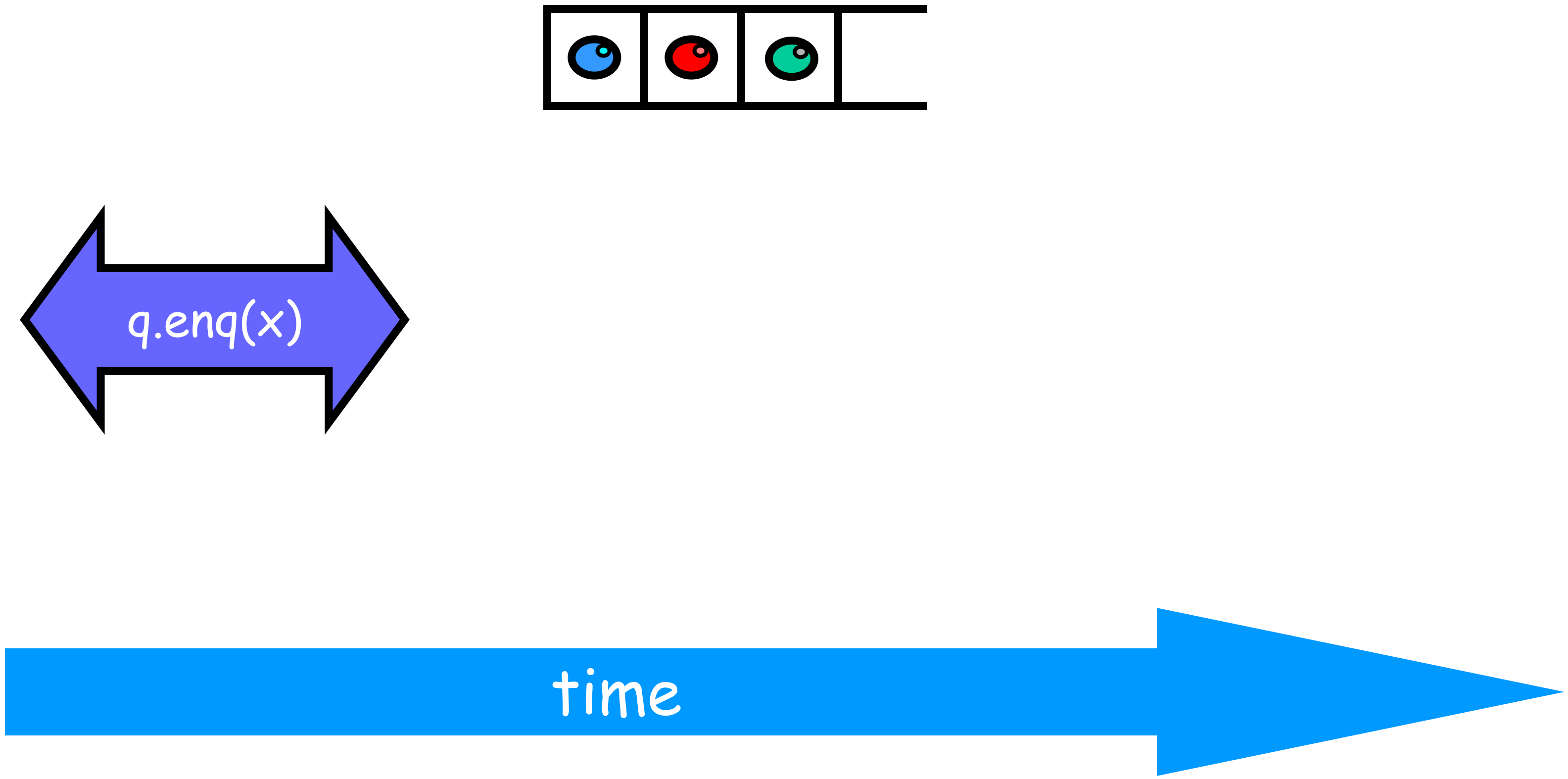
Example



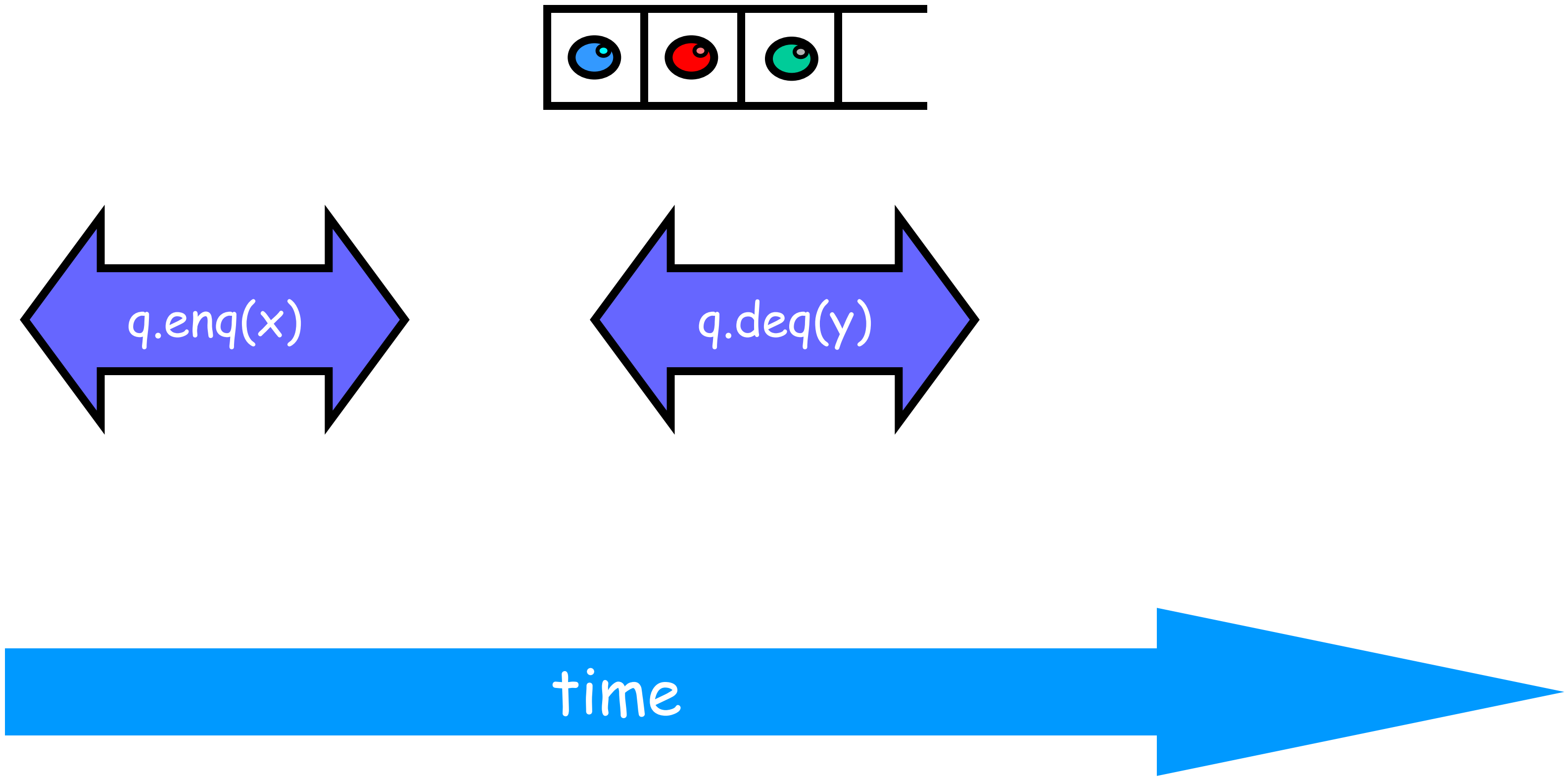
Example



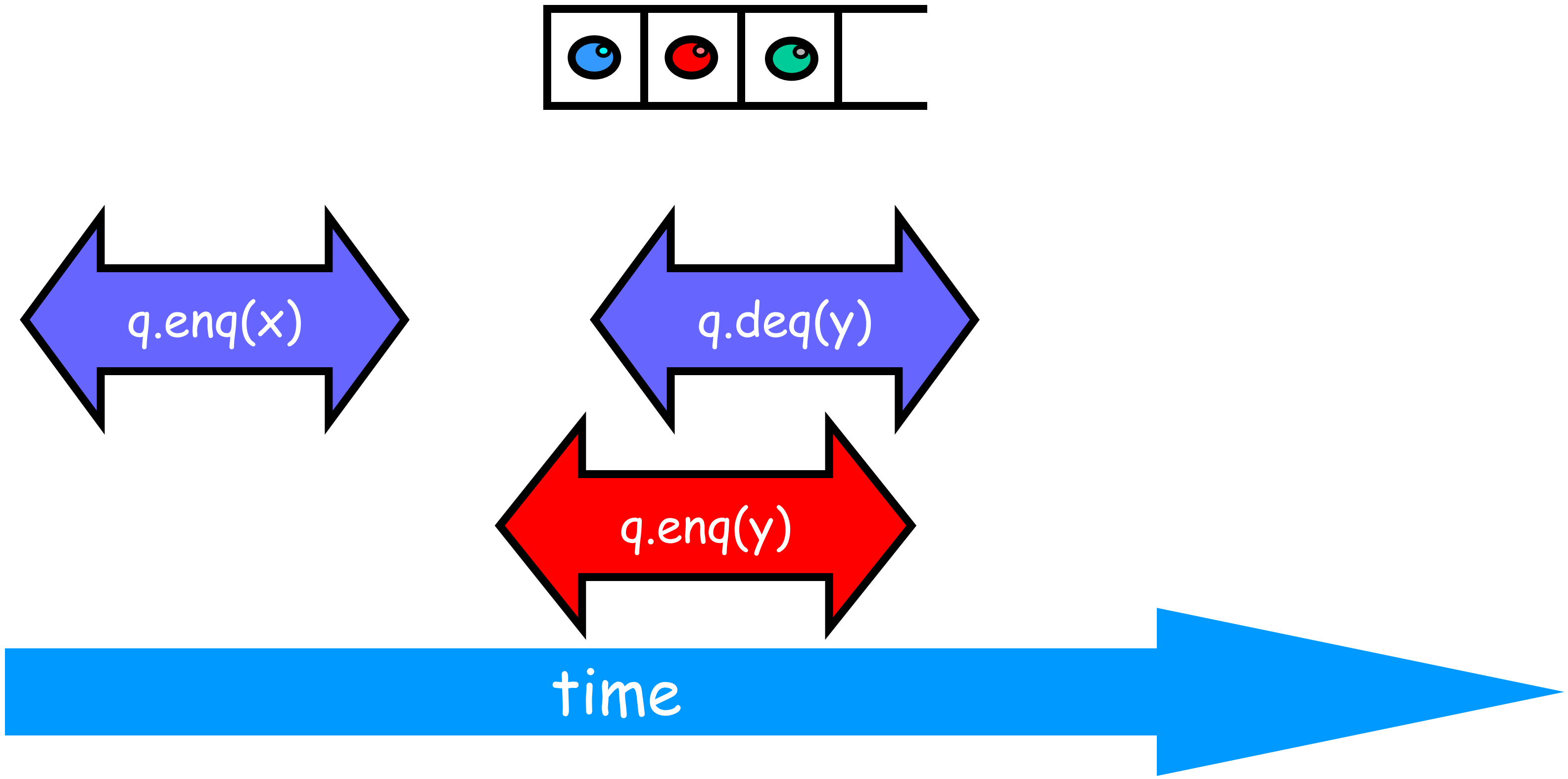
Example



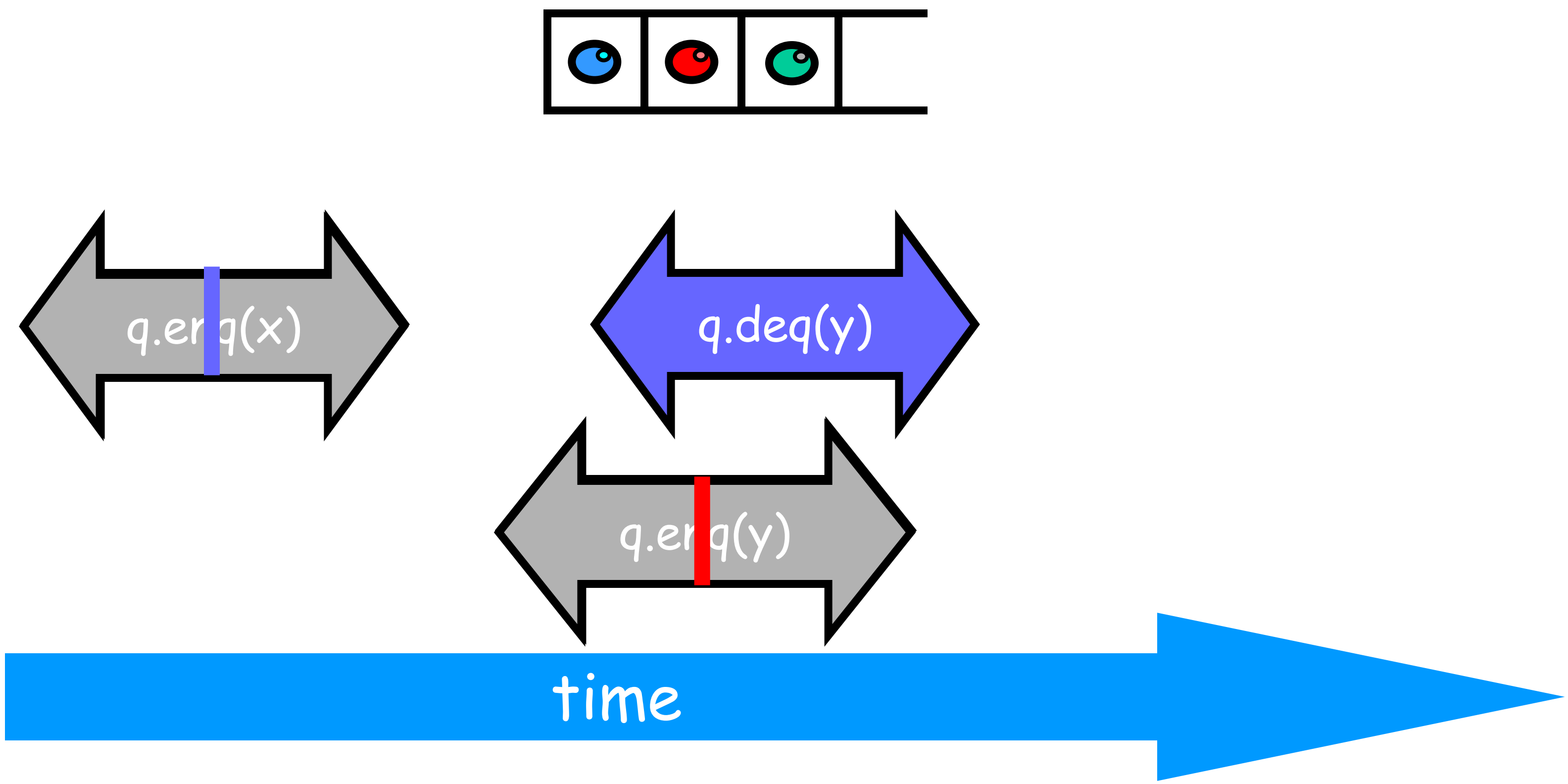
Example



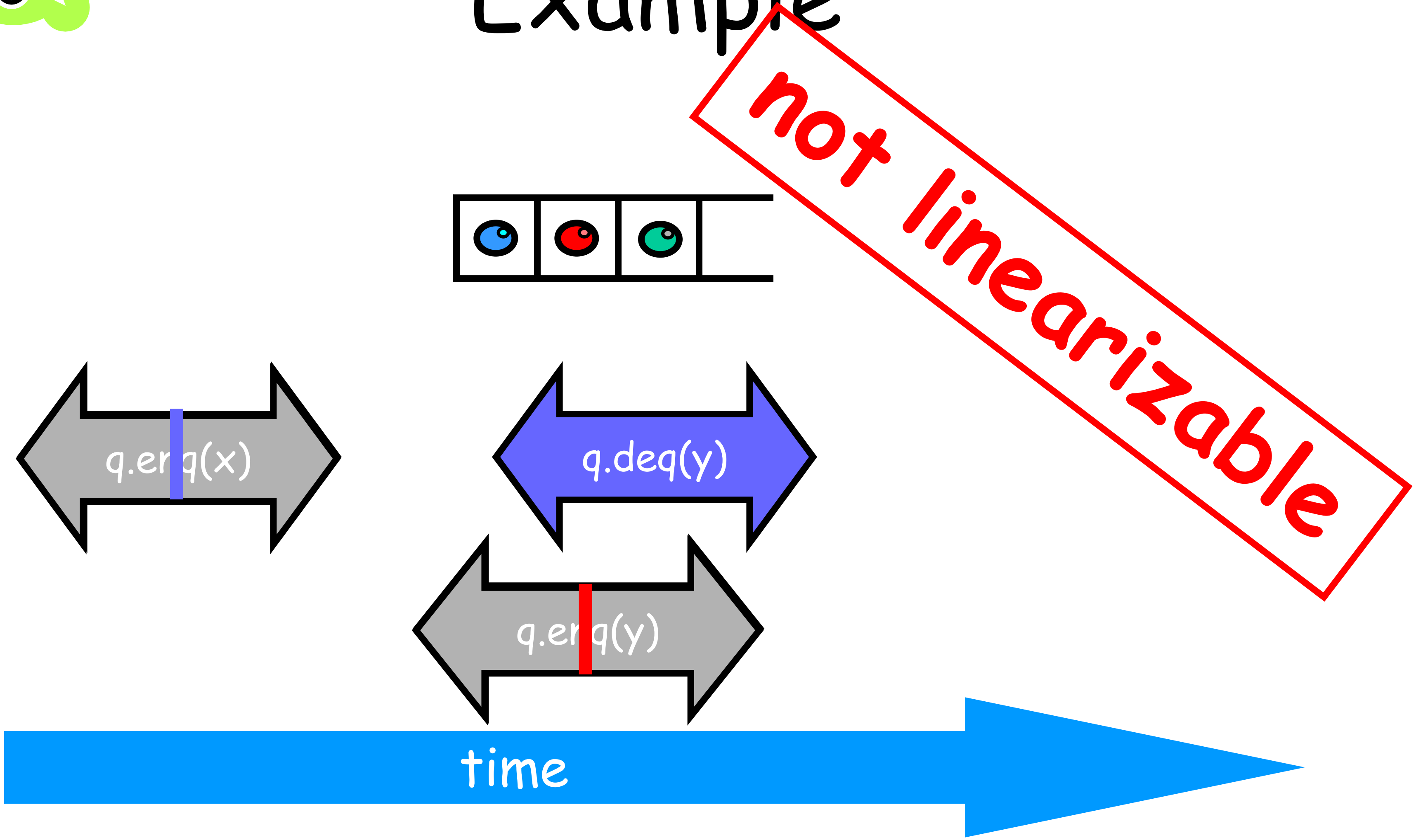
Example



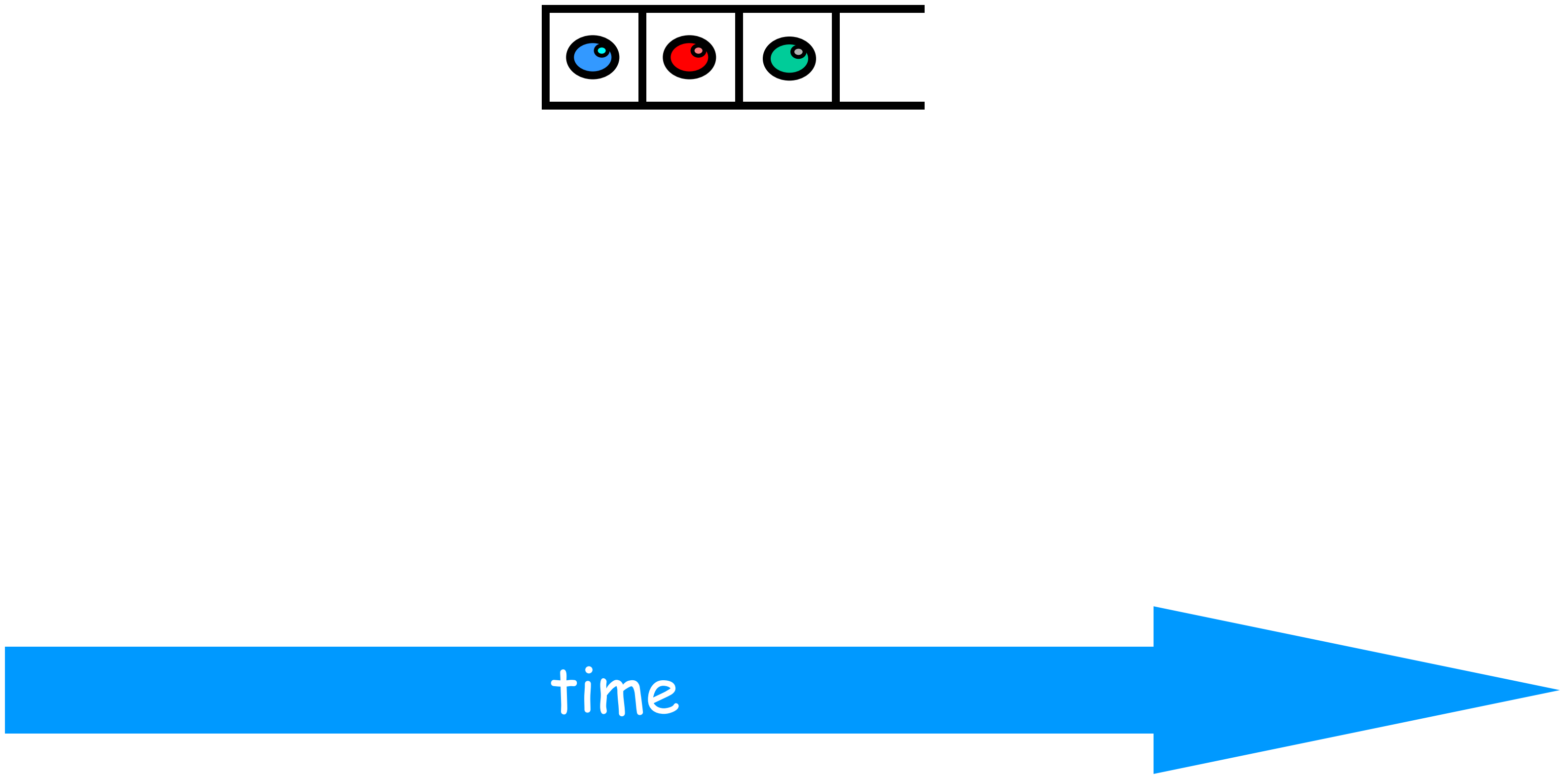
Example



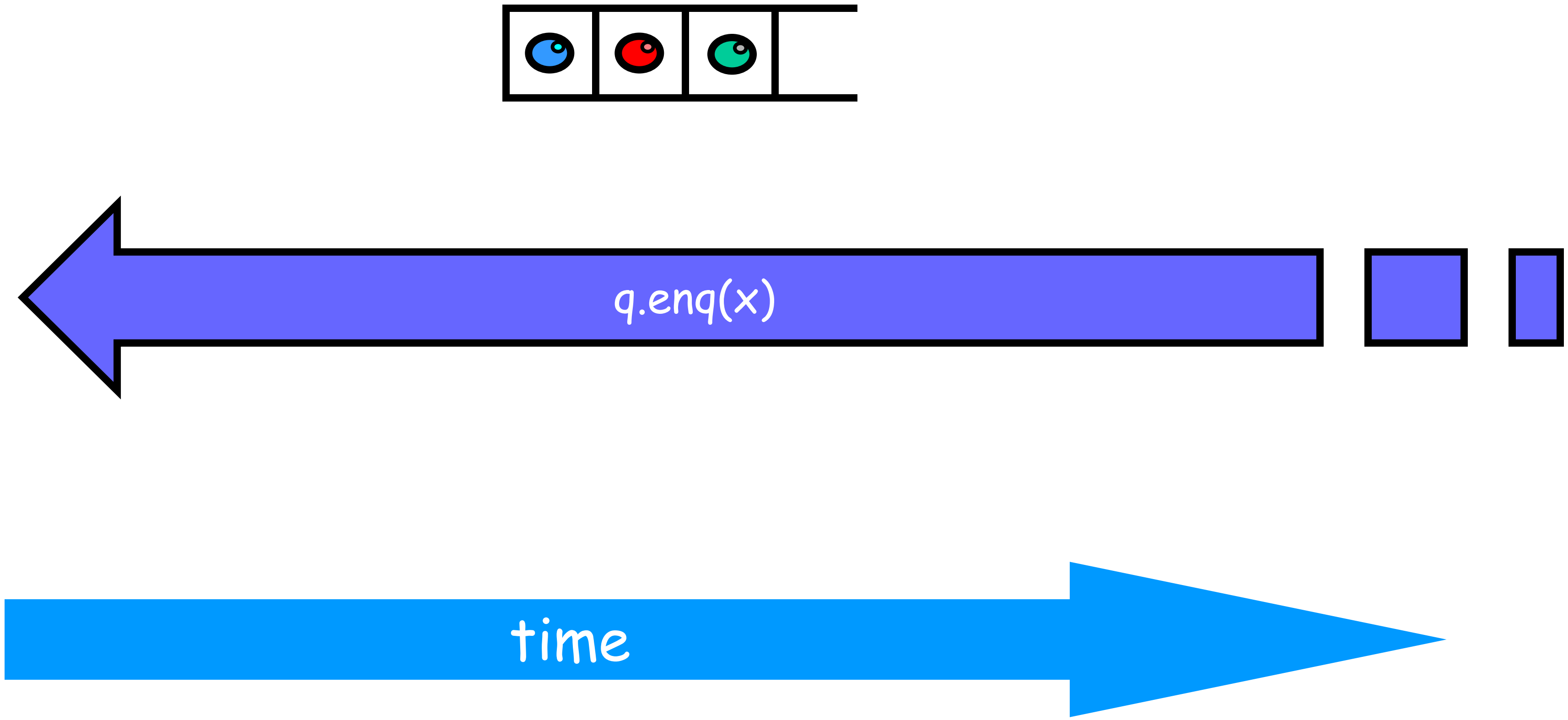
Example



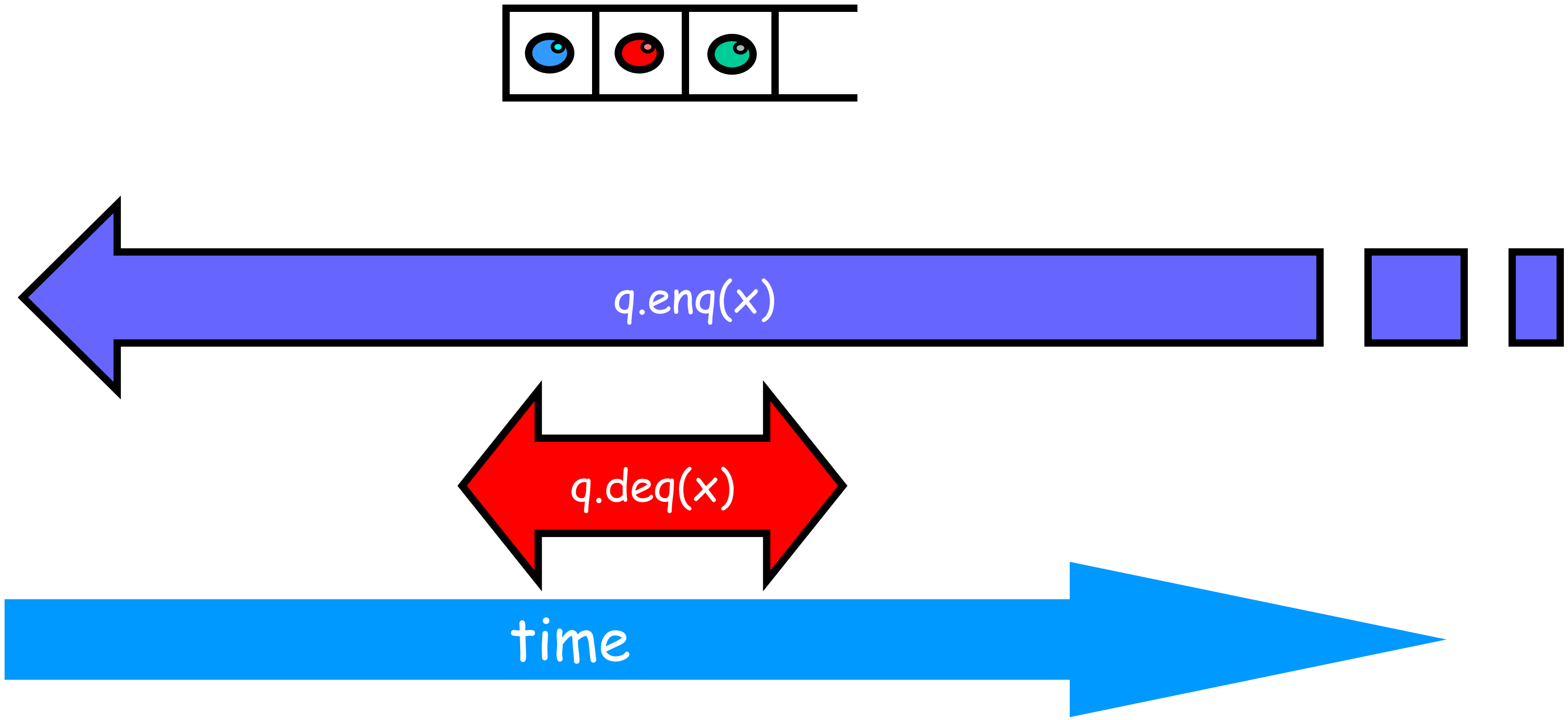
Example



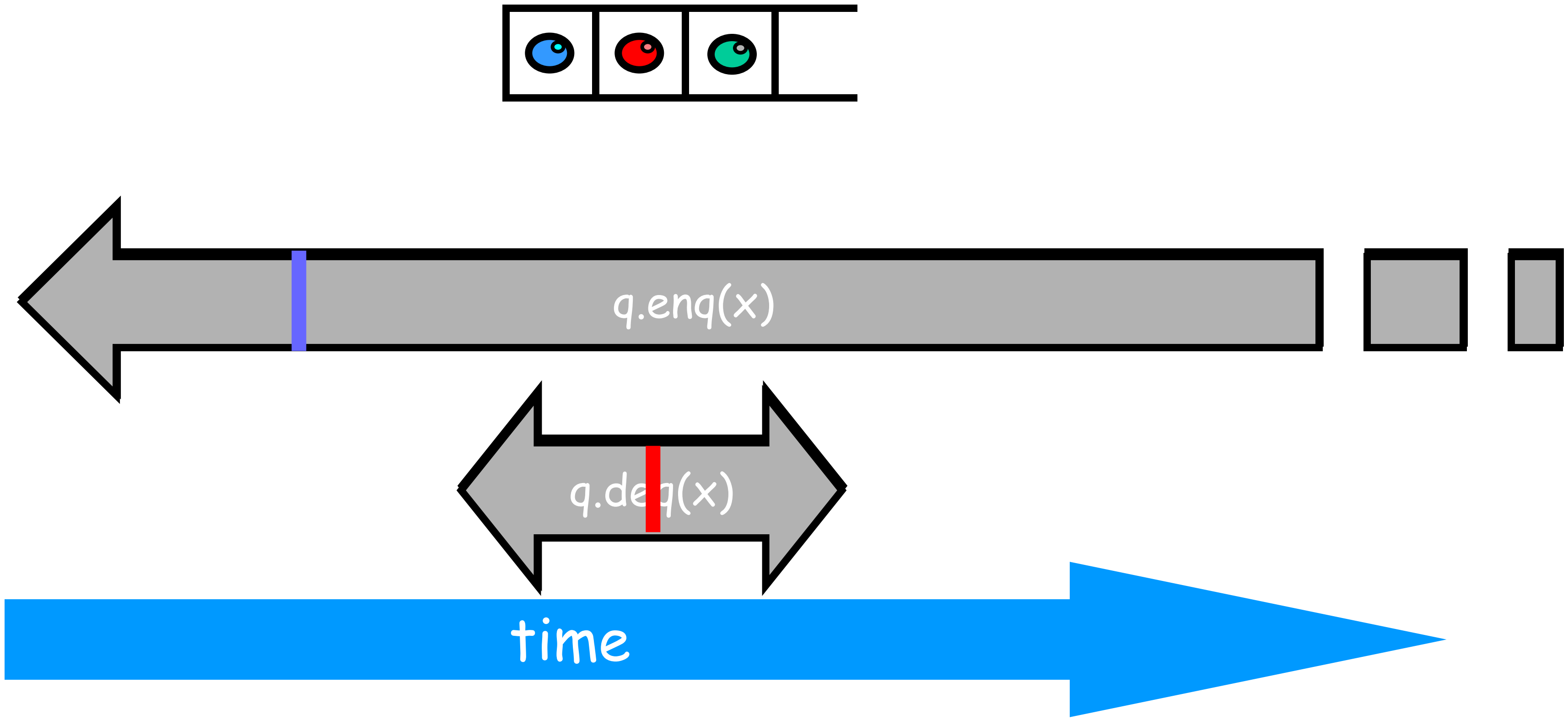
Example



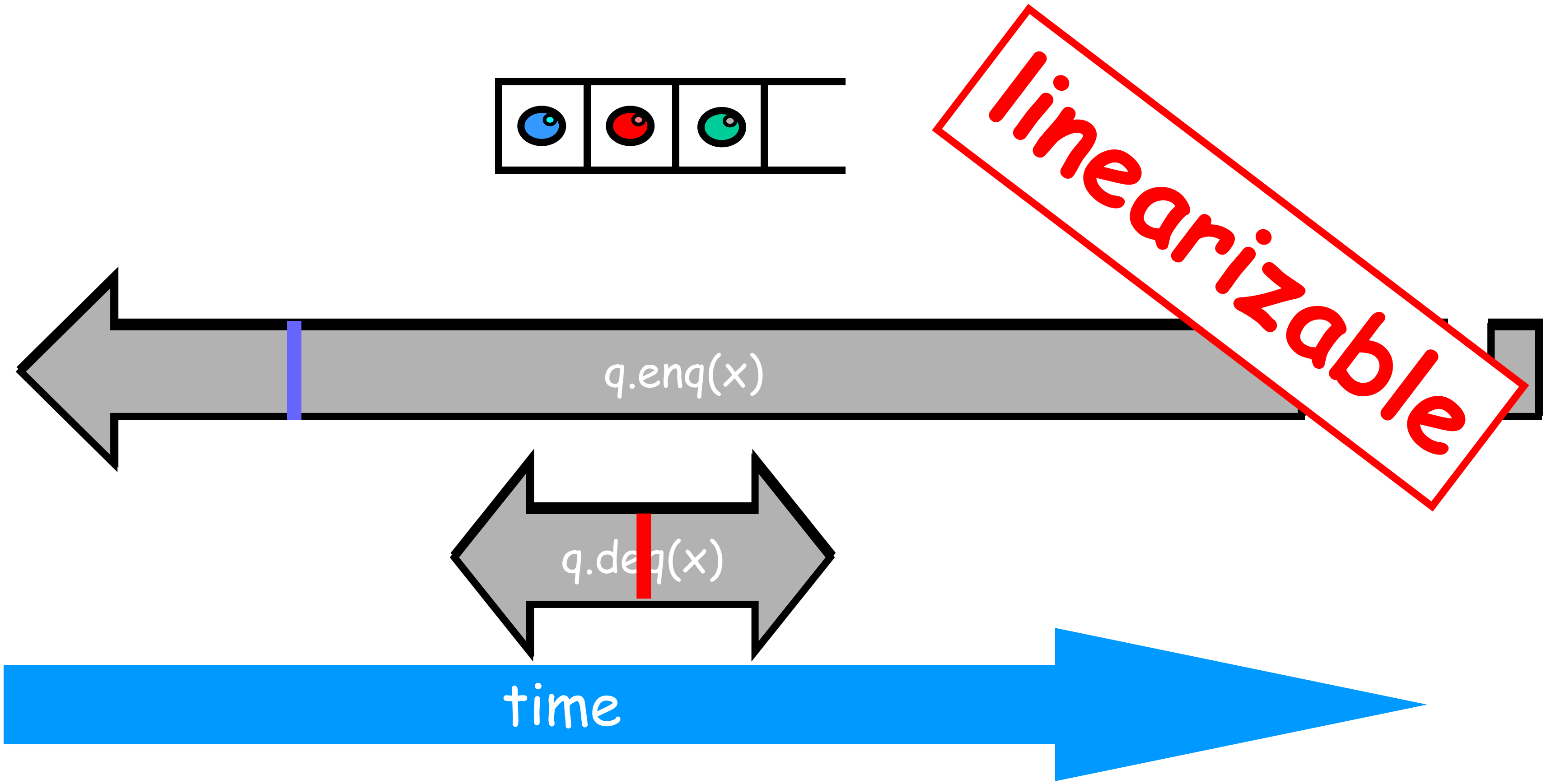
Example



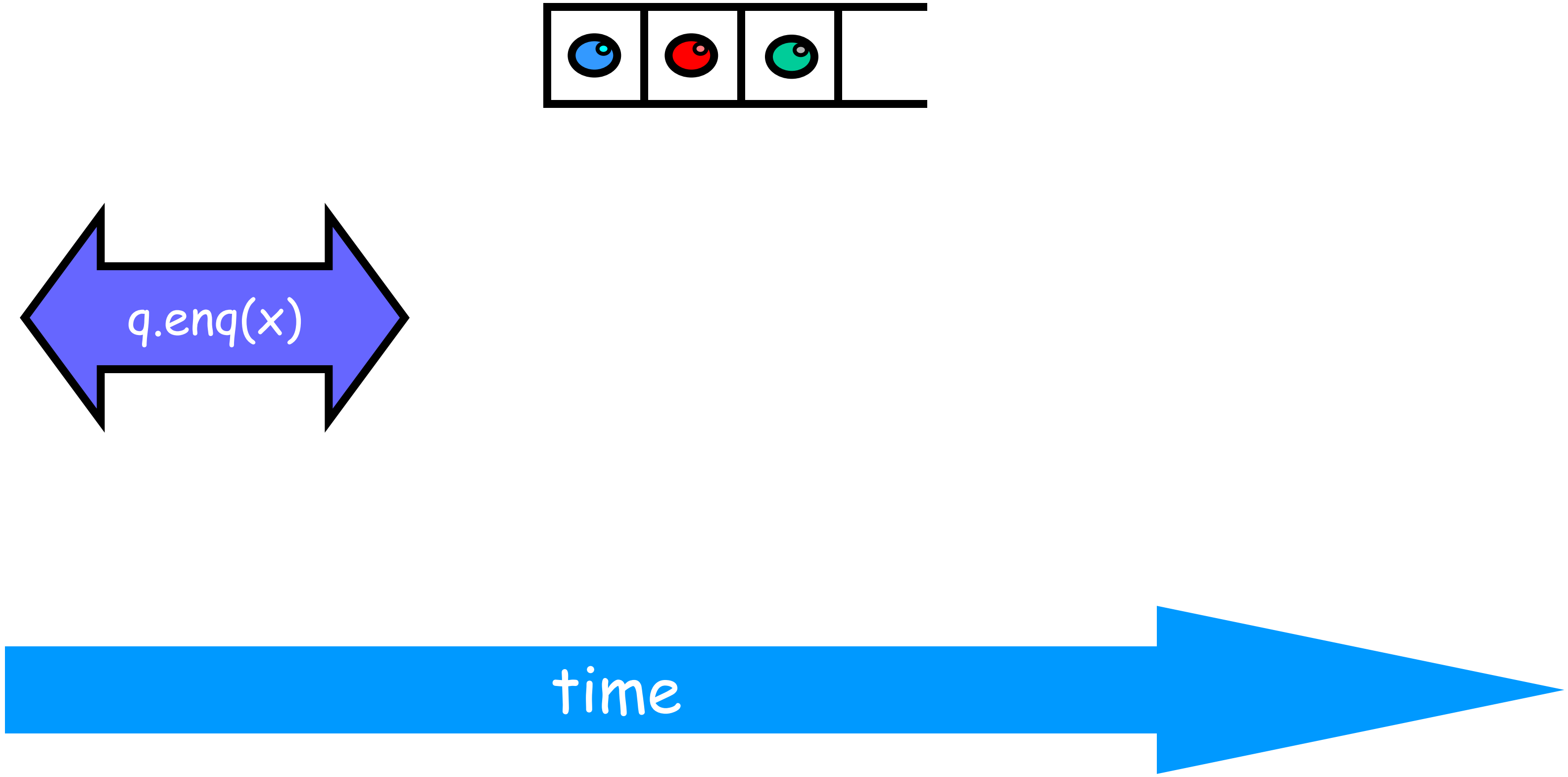
Example



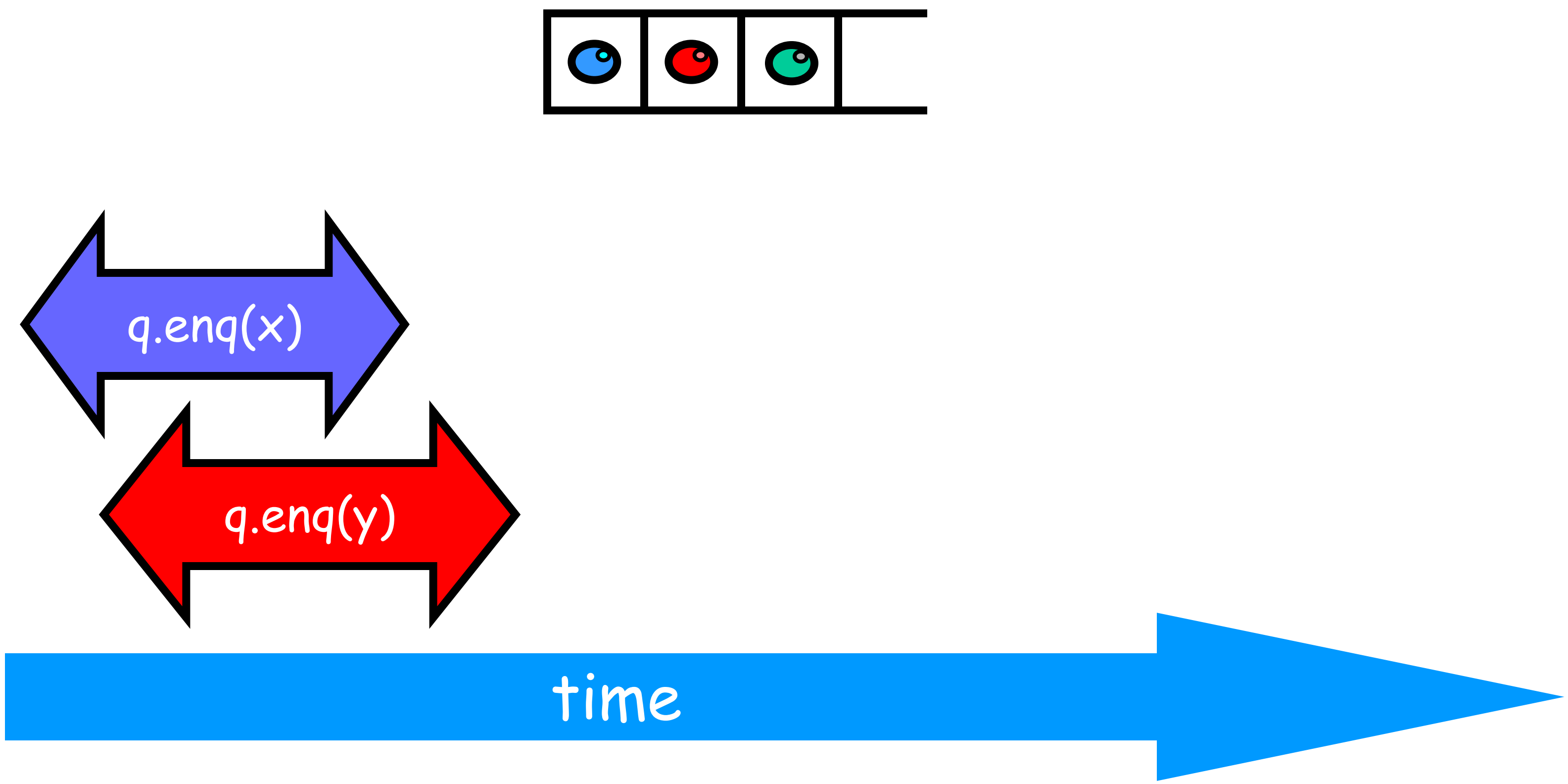
Example



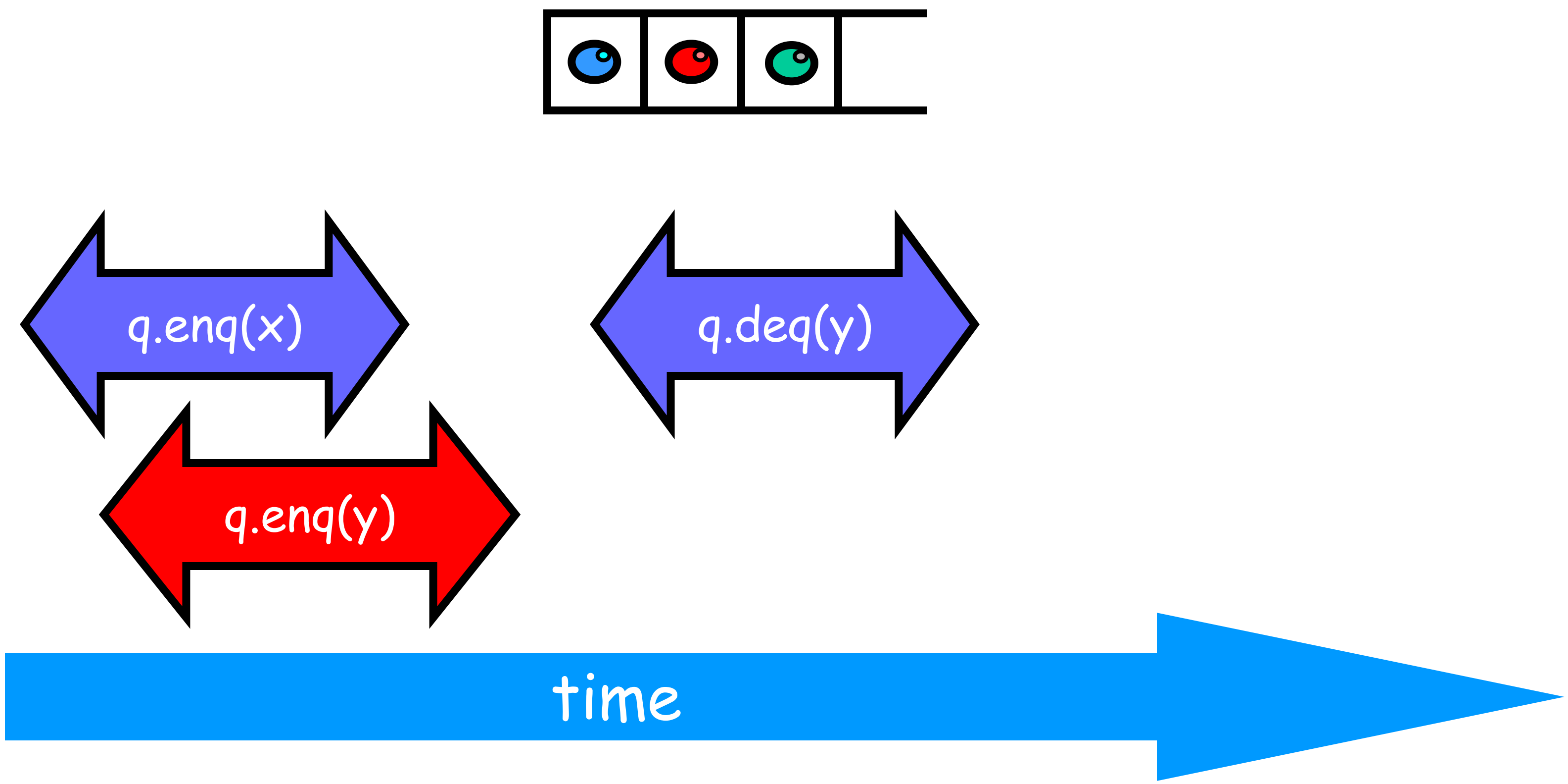
Example



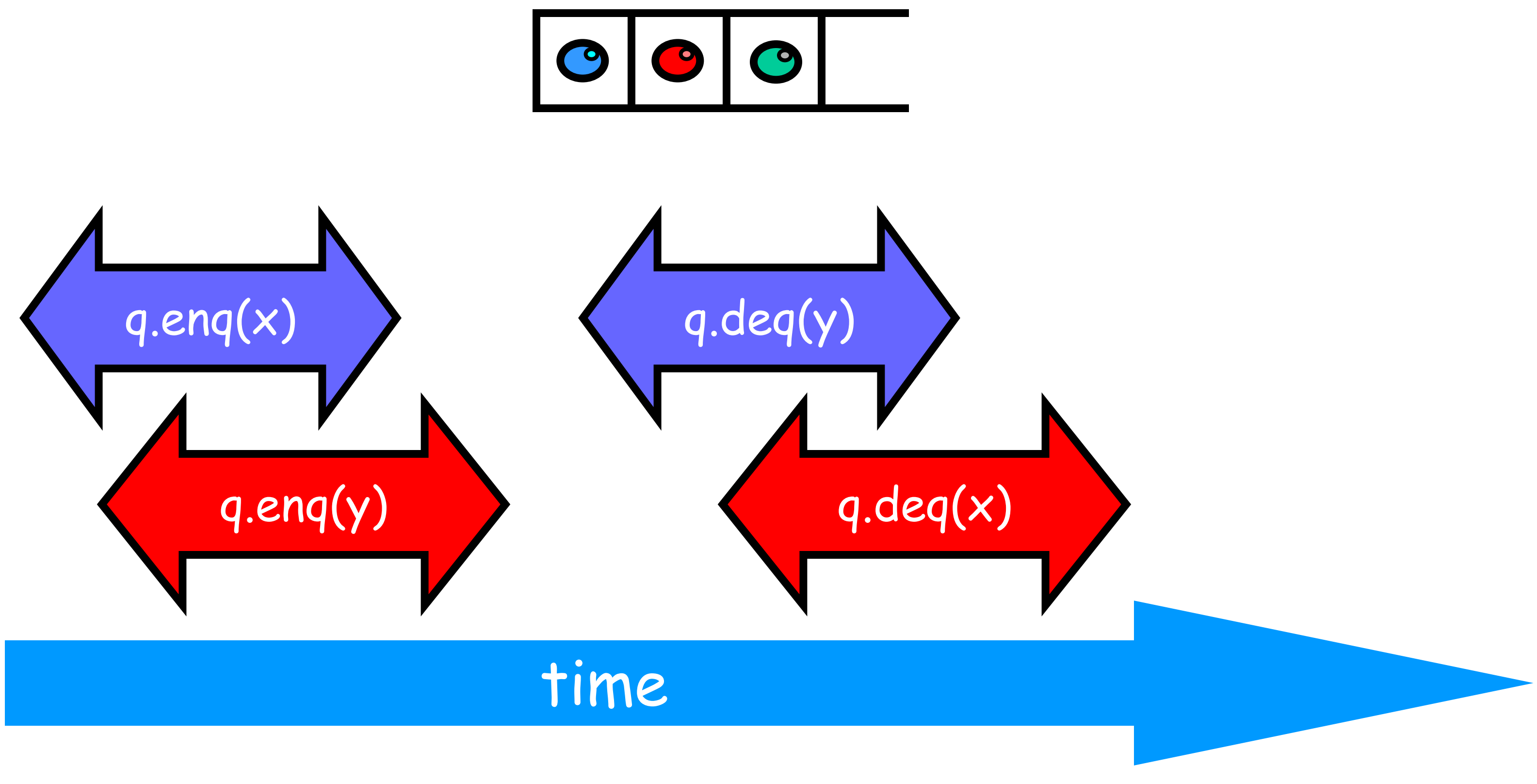
Example



Example

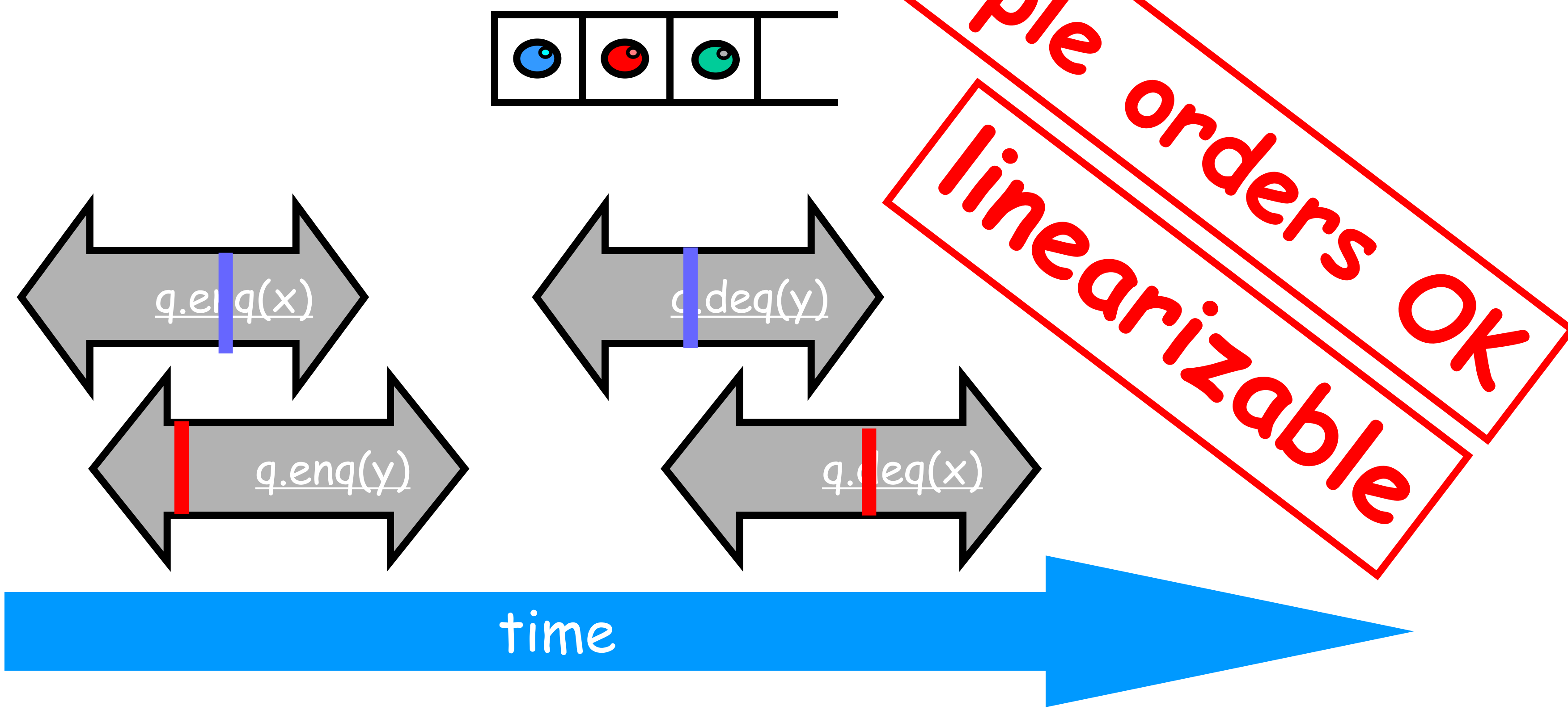


Example

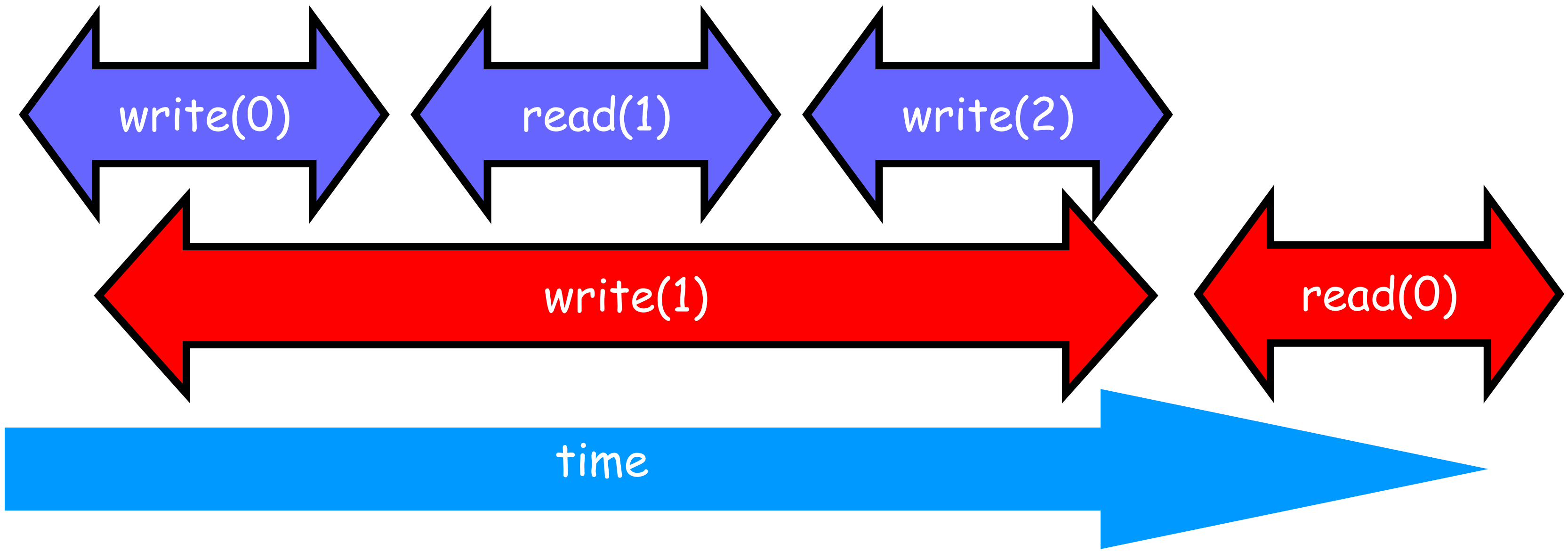


Example

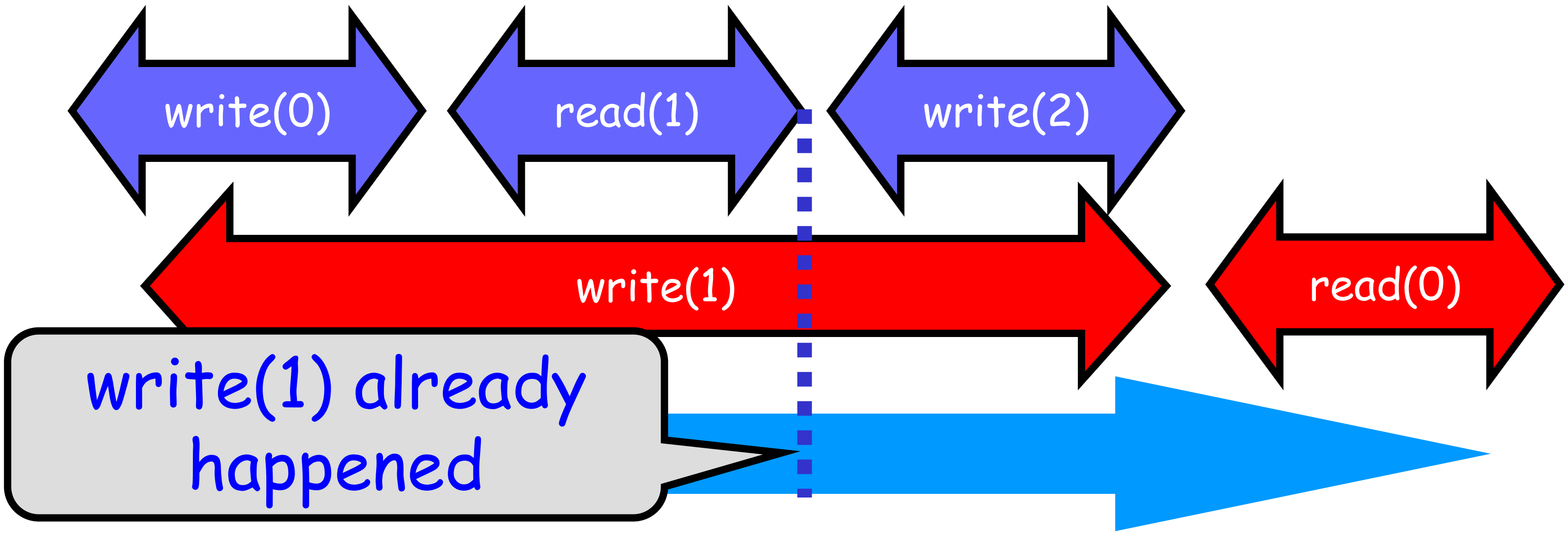
Comme ci
Comme ça



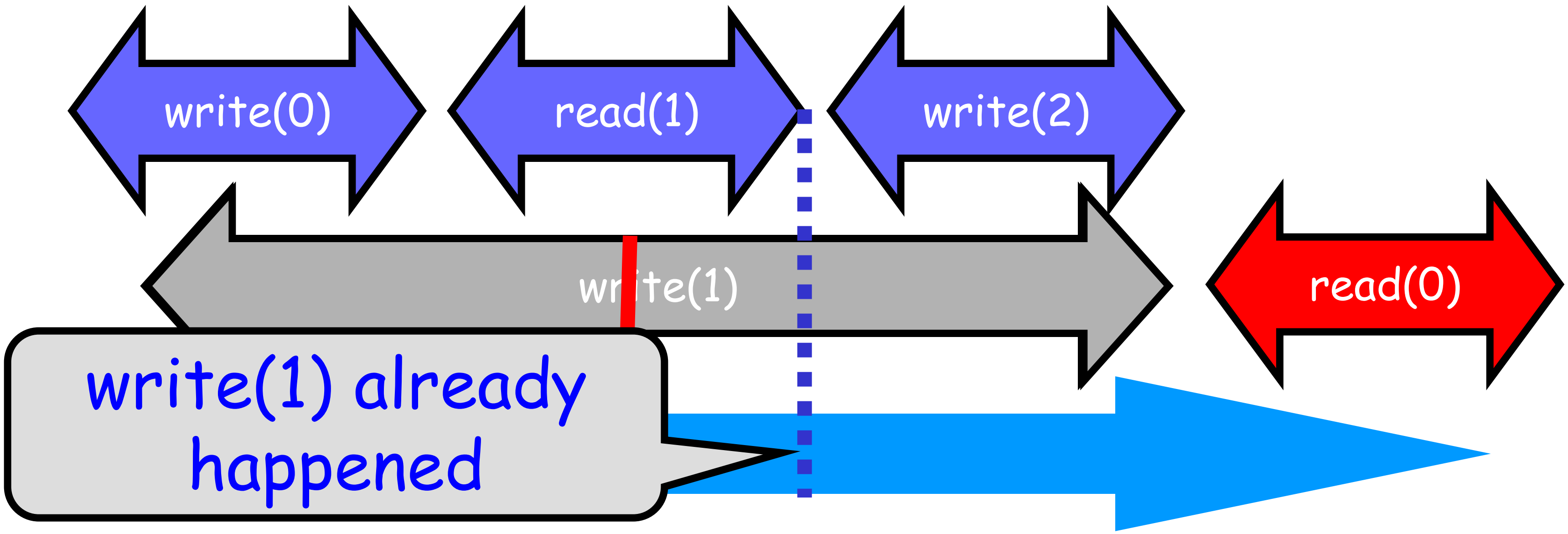
Read/Write Register Example



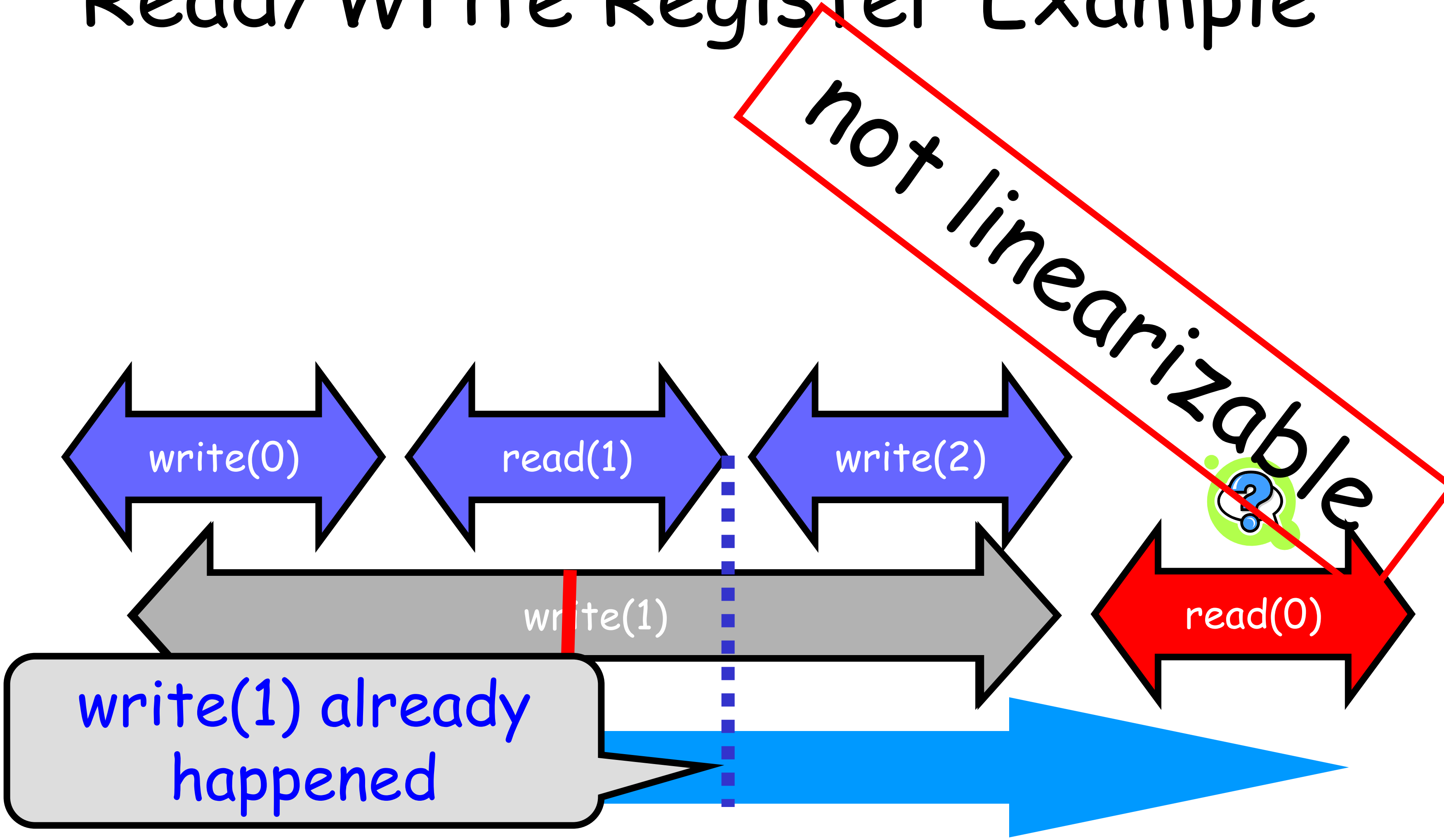
Read/Write Register Example



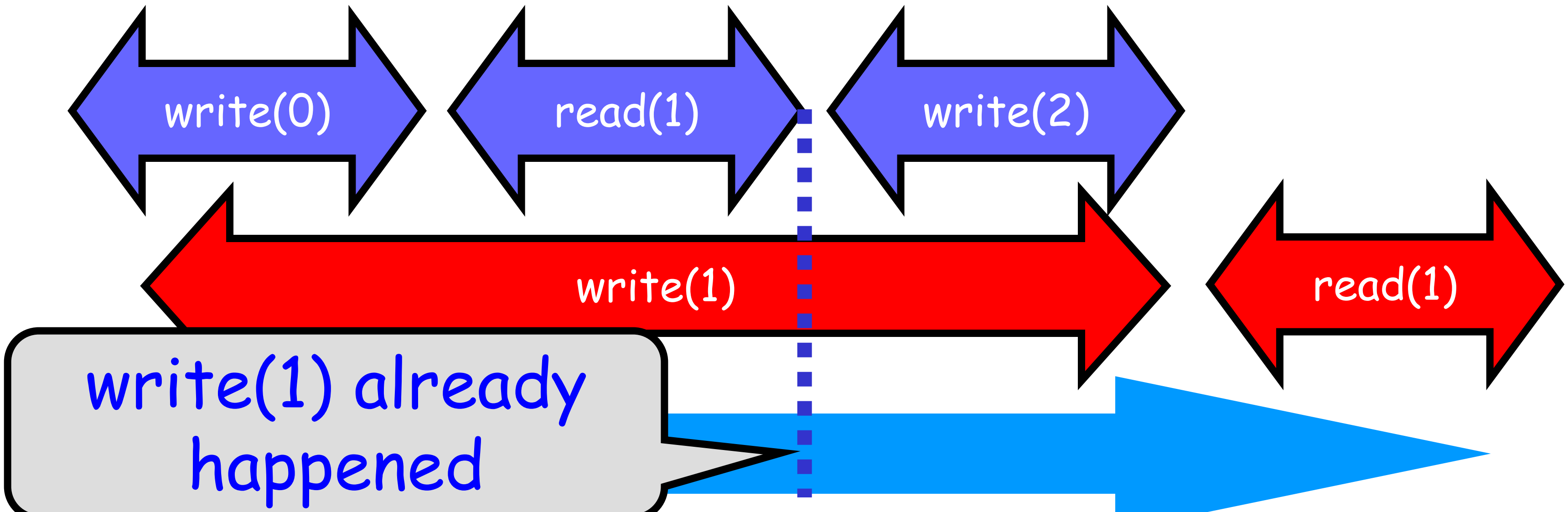
Read/Write Register Example



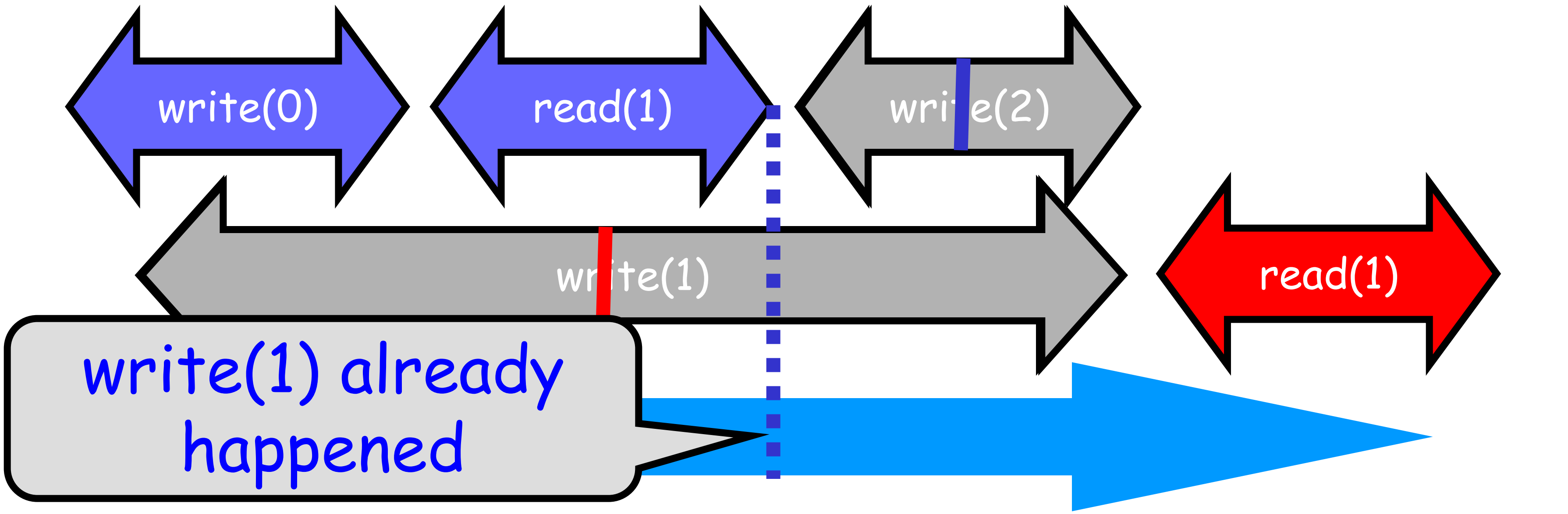
Read/Write Register Example



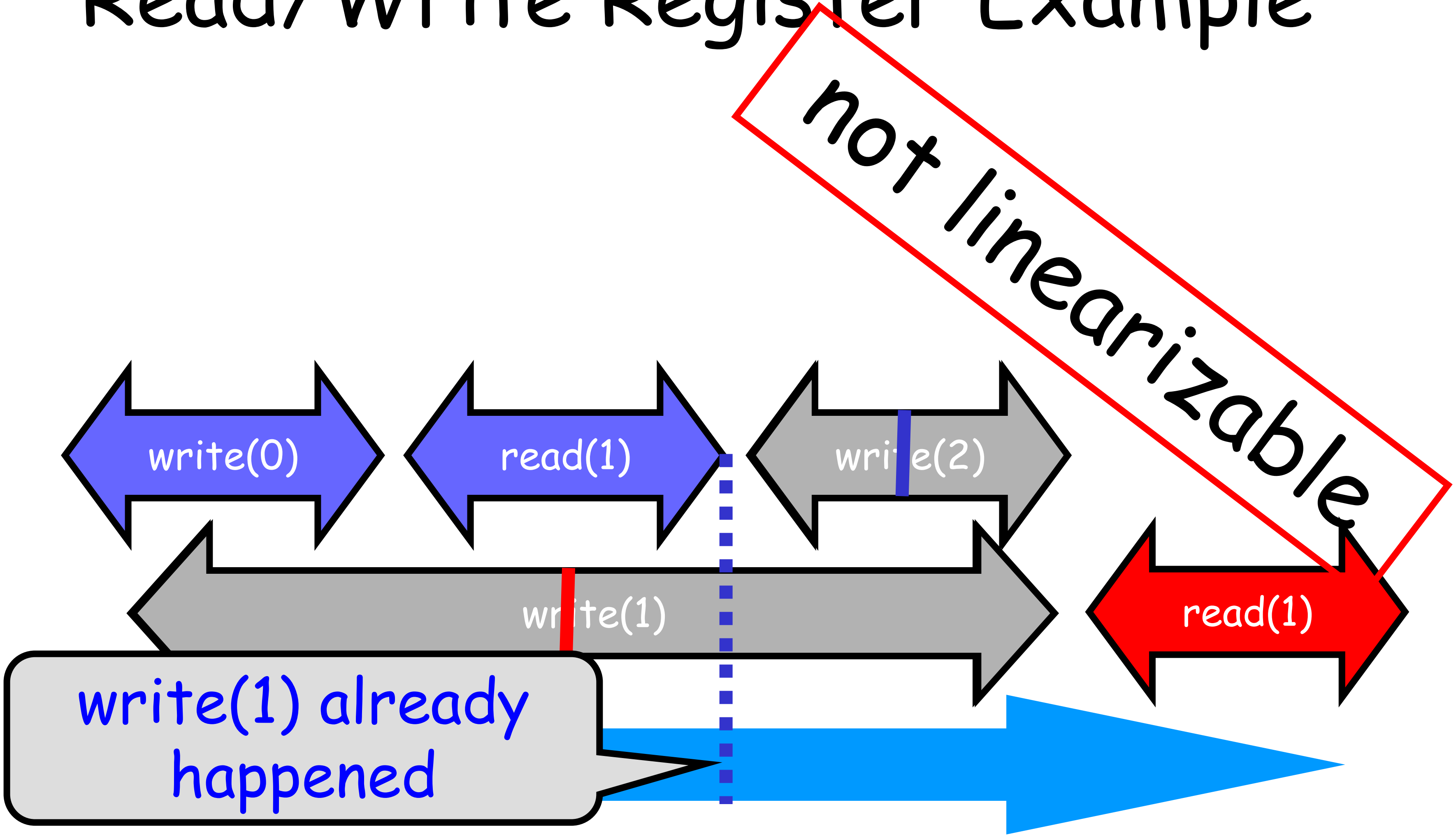
Read/Write Register Example



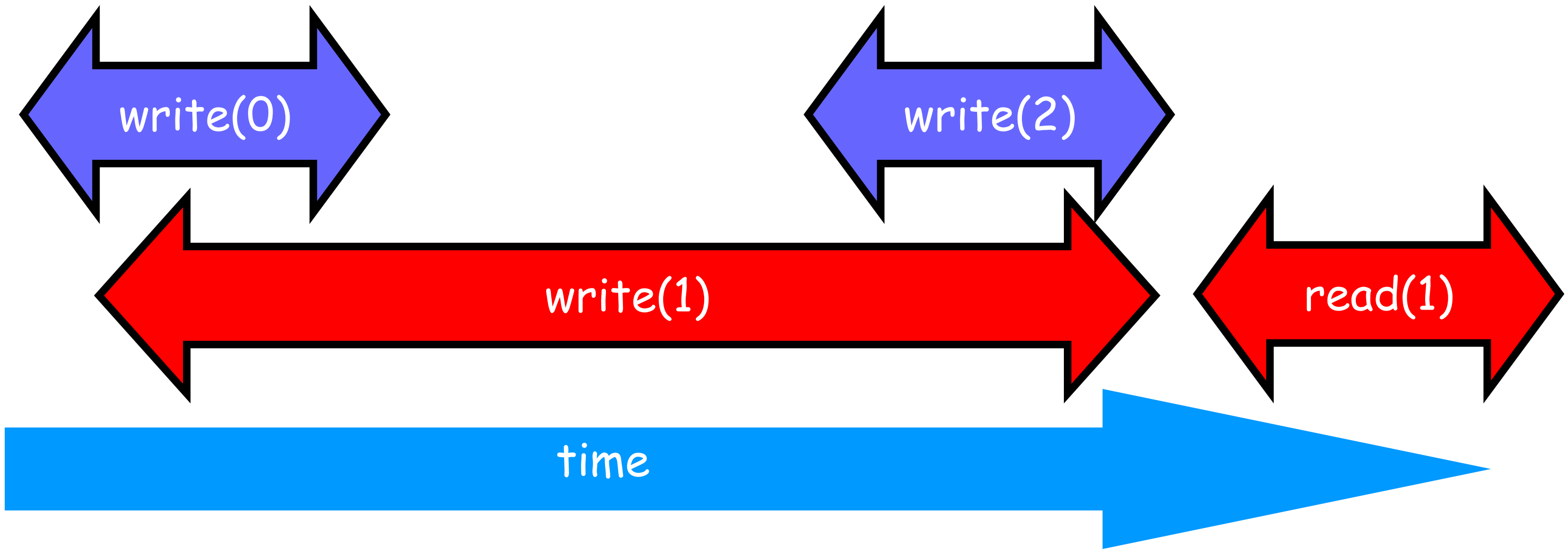
Read/Write Register Example



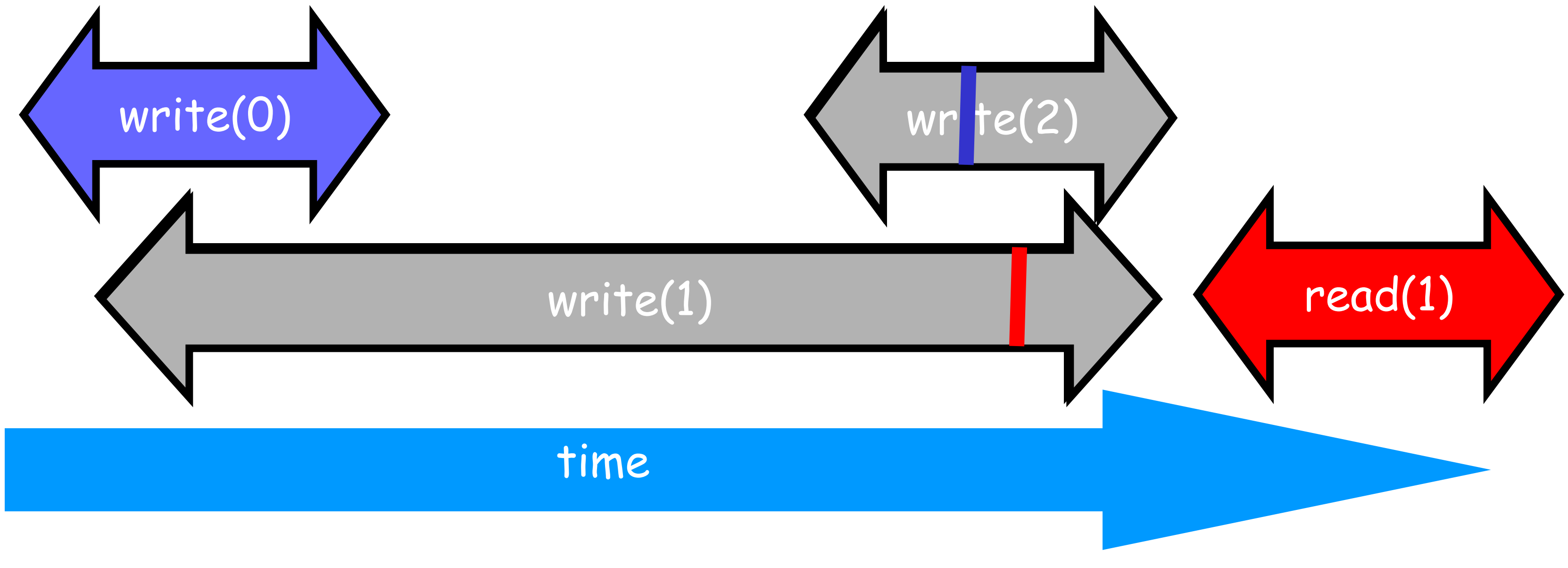
Read/Write Register Example



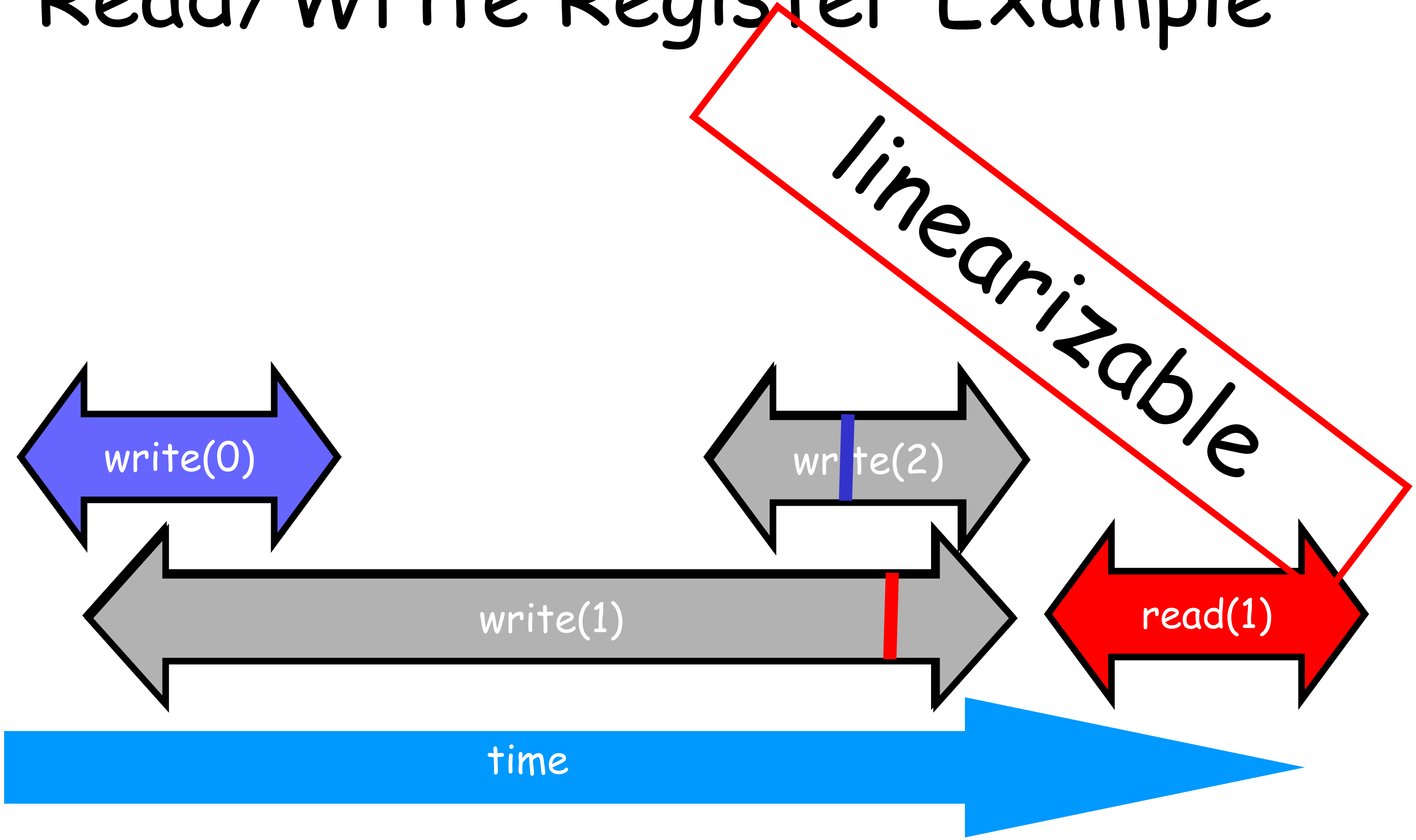
Read/Write Register Example



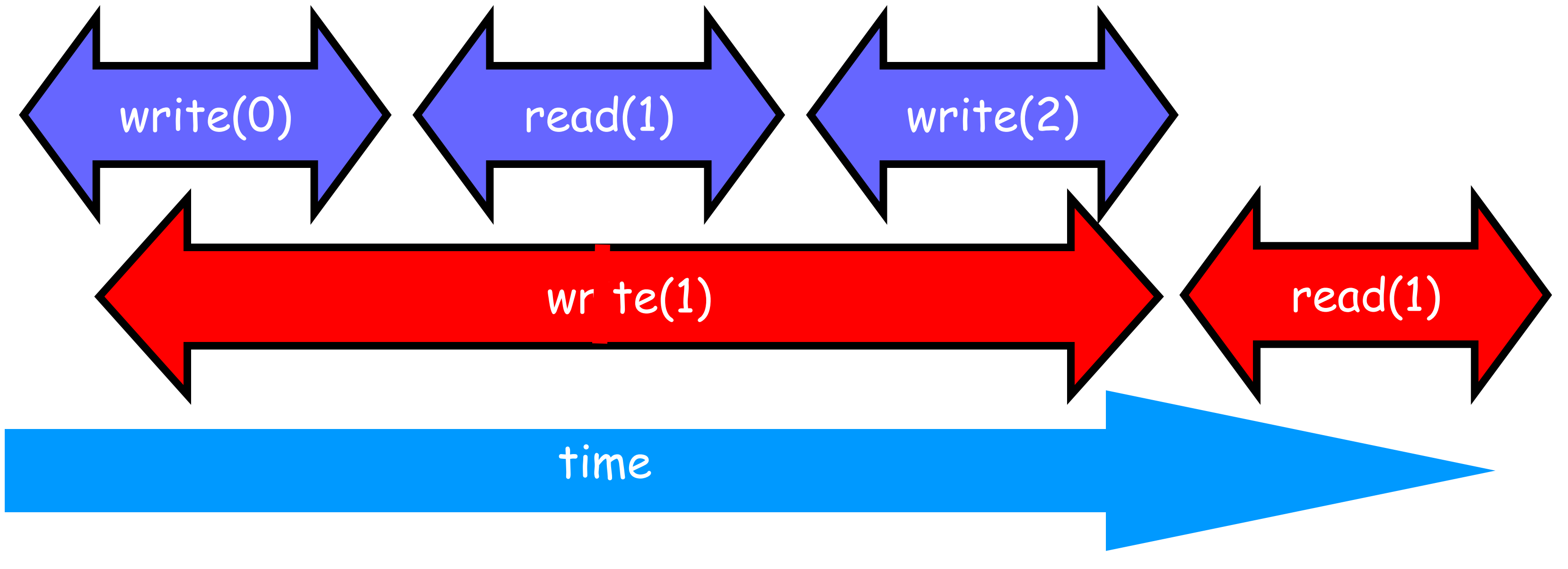
Read/Write Register Example



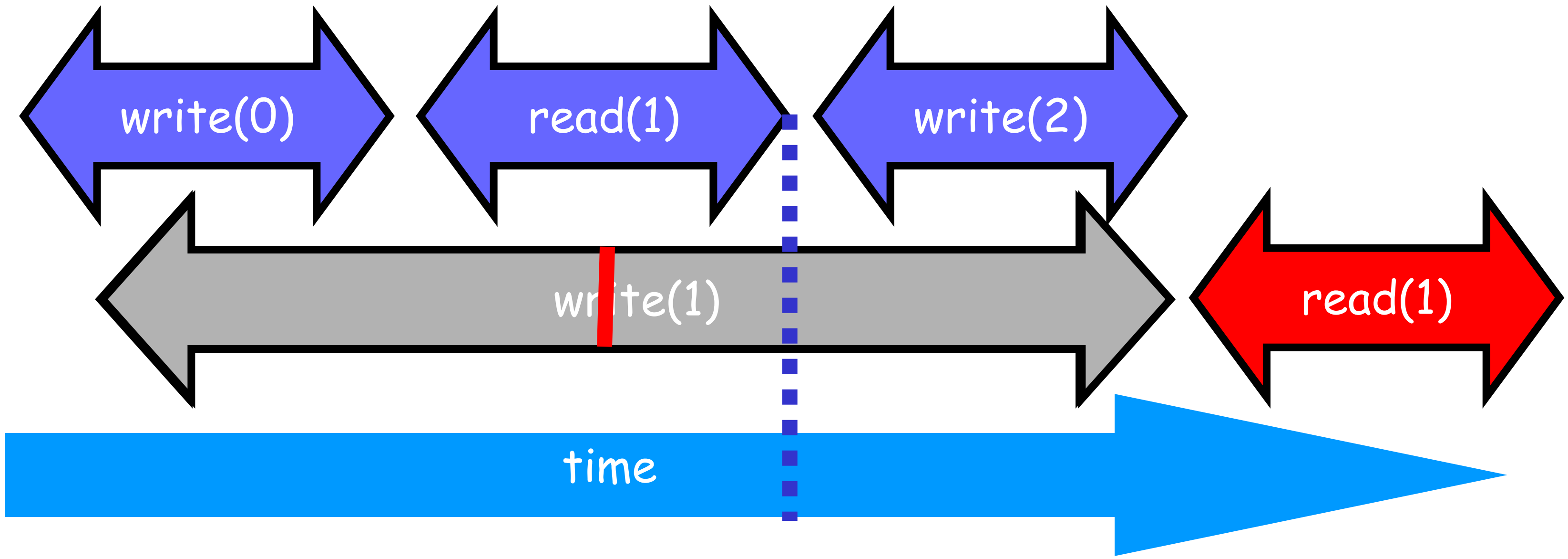
Read/Write Register Example



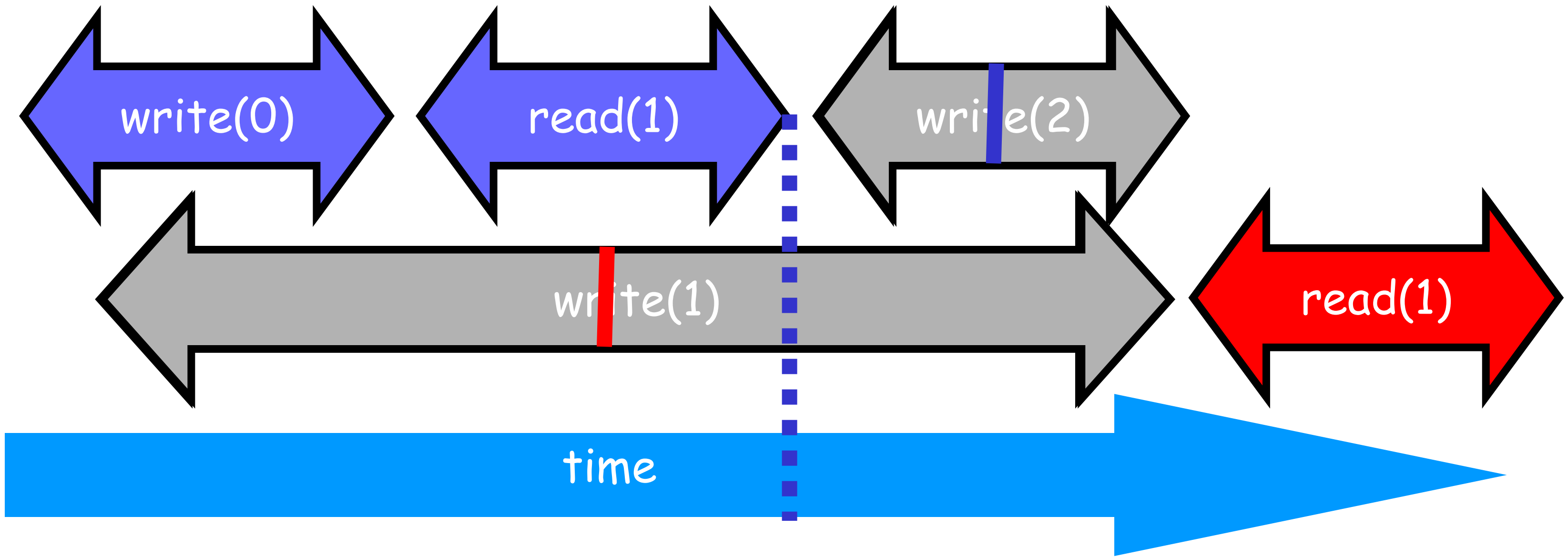
Read/Write Register Example



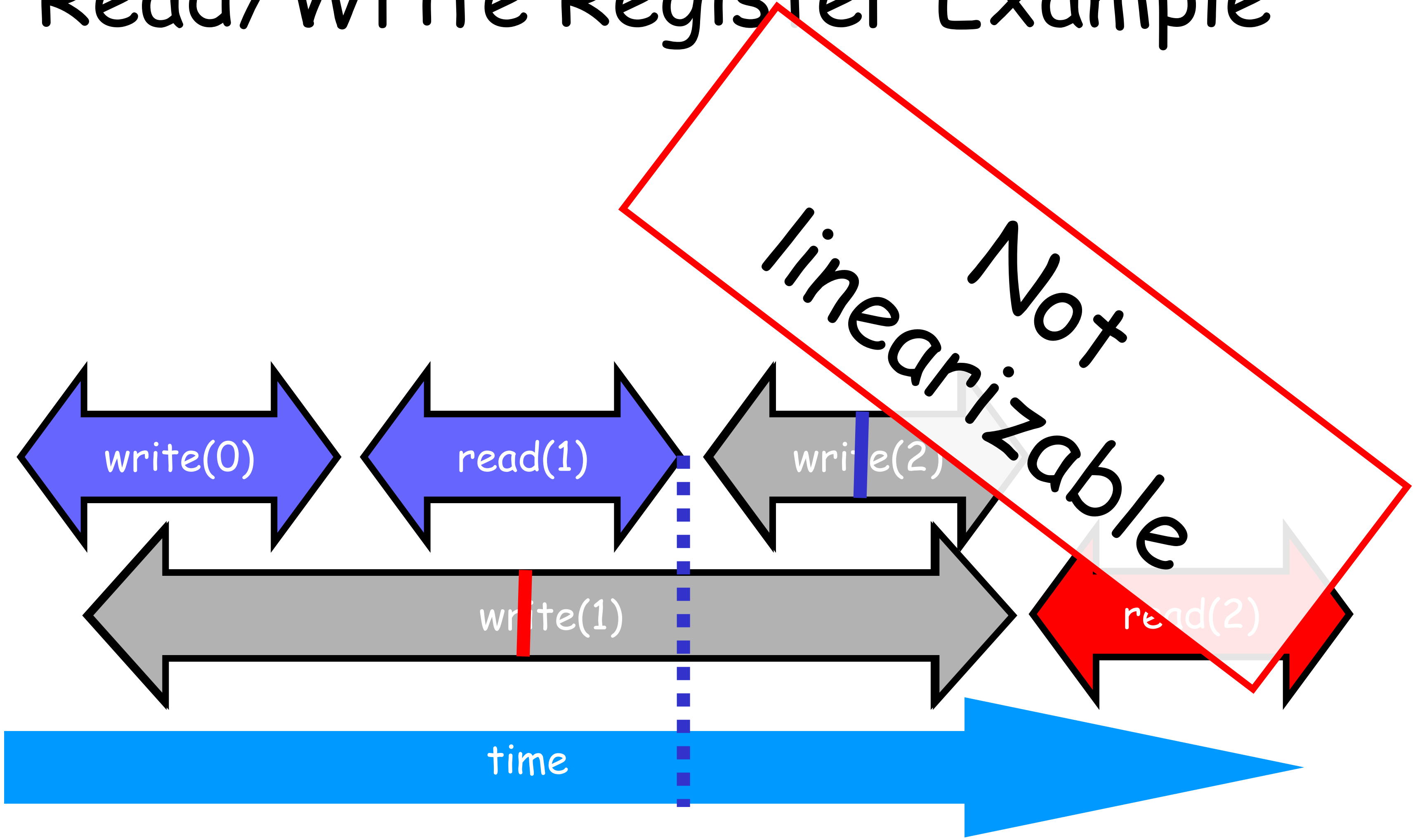
Read/Write Register Example



Read/Write Register Example



Read/Write Register Example



Talking About Executions

- Why?
  - Can't we specify the linearization point of each operation without describing an execution?
- Not Always
  - In some cases, linearization point depends on the execution

Formal Model of Executions

- Define precisely what we mean
  - Ambiguity is bad when intuition is weak
- Allow reasoning
  - Formal
  - But mostly informal

Split Method Calls into Two Events

- Invocation
  - method name & args
  - `q.enq(x)`
- Response
  - result or exception
  - `q.enq(x)` returns `void`
  - `q.deq()` returns `x`
  - `q.deq()` throws `empty`

Invocation Notation

A q.enq(x)

Invocation Notation

A q.enq(x)

thread

Invocation Notation

A q.enq(x)

thread

method

Invocation Notation

A q.enq(x)

thread

method

object

Invocation Notation

A q.enq(x)

thread

method

object

arguments

Response Notation

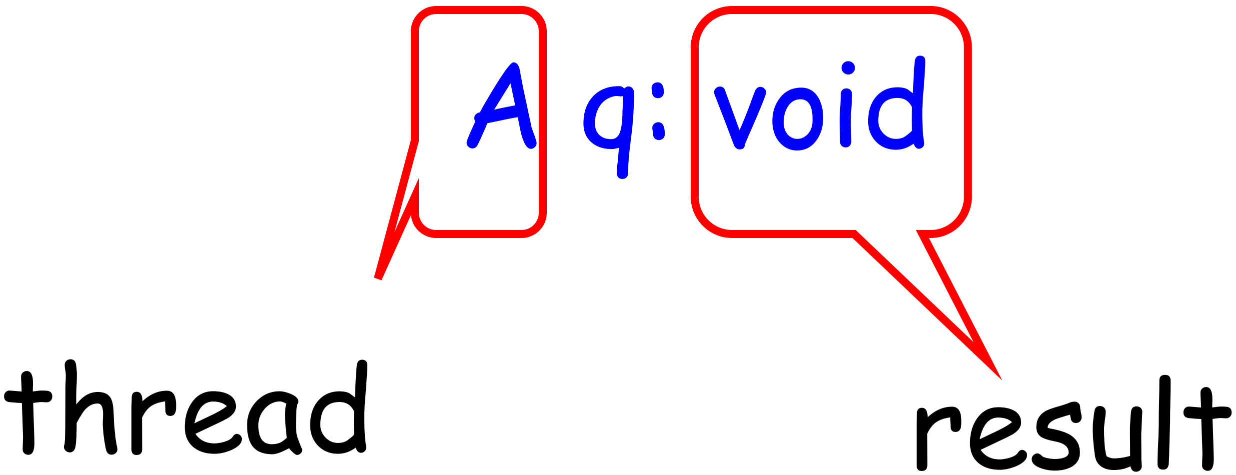
A q: void

Response Notation

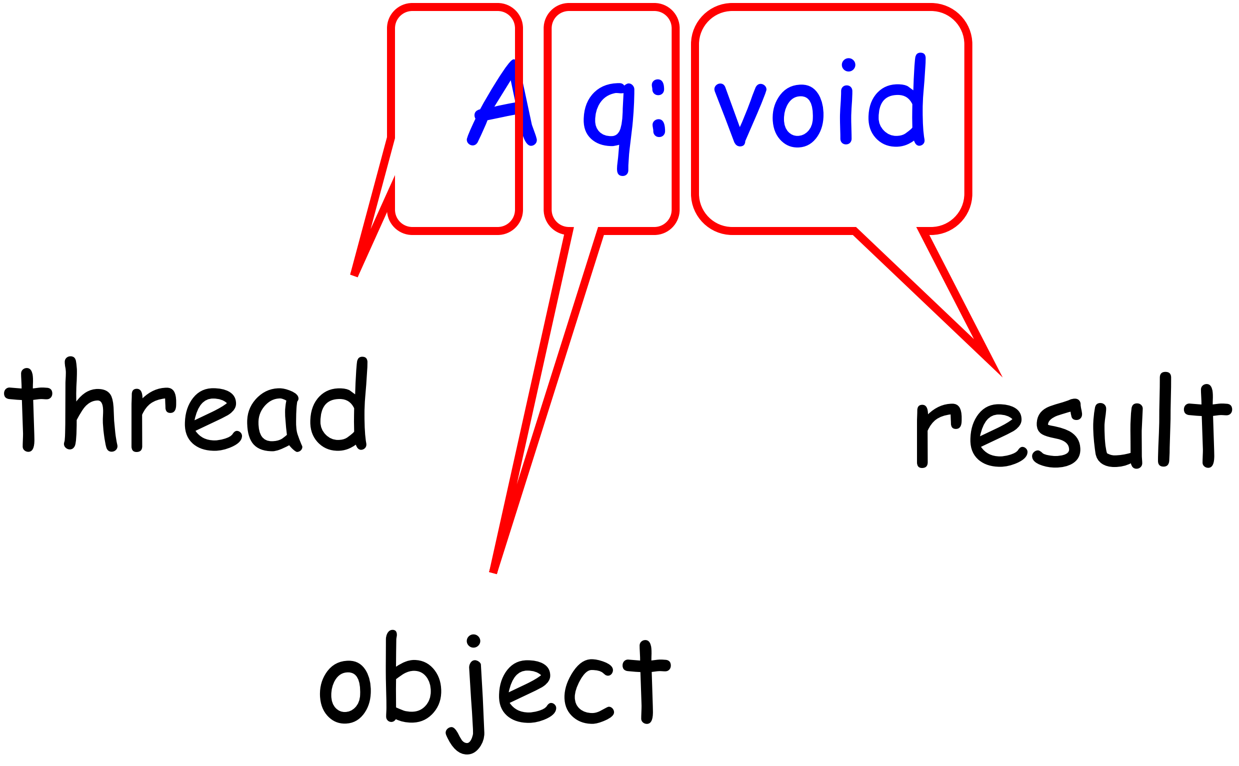
A q: void

thread

Response Notation



Response Notation



History - Describing an Execution

H = A q.enq(3)
A q:void
A q.enq(5)
B p.enq(4)
B p:void
B q.deq()
B q:3

Sequence of invocations and responses

Definition

- Invocation & response *match* if

Thread names agree

Object names agree

A q.enq(3)

A q:void

Method call

Object Projections

$$H = \begin{array}{l} A\ q.enq(3) \\ A\ q{:}void \\ B\ p.enq(4) \\ B\ p{:}void \\ B\ q.deq() \\ B\ q{:}3 \end{array}$$

Object Projections

A q.enq(3)
A q:void

H|q =

B q.deq()
B q:3

Thread Projections

H = A q.enq(3)
A q:void
B p.enq(4)
B p:void
B q.deq()
B q:3

Thread Projections

$$H|B = \begin{array}{l} B\ p.enq(4) \\ B\ p{:}void \\ B\ q.deq() \\ B\ q{:}3 \end{array}$$

Complete Subhistory

H = A q.enq(3)
A q:void
A q.enq(5)
B p.enq(4)
B p:void
B q.deq()
B q:3

An invocation is *pending* if it has no matching respnse

Complete Subhistory

H =

A q.enq(3)
A q:void
A q.enq(5)
B p.enq(4)
B p:void
B q.deq()
B q:3

May or may not have taken effect

Complete Subhistory

H =

A q.enq(3)
A q:void
A q.enq(5)
B p.enq(4)
B p:void
B q.deq()
B q:3

discard pending invocations

Complete Subhistory

A q.enq(3)
A q:void

Complete(H) = B p.enq(4)
B p:void
B q.deq()
B q:3

Sequential Histories

A q.enq(3)
A q:void
B p.enq(4)
B p:void
B q.deq()
B q:3
A q:enq(5)

Sequential Histories

A q.enq(3)
A q:void
B p.enq(4)
B p:void
B q.deq()
B q:3
A q:enq(5)

match

Sequential Histories

A q.enq(3)
A q:void
B p.enq(4)
B p:void
B q.deq()
B q:3
A q:enq(5)

match

match

Sequential Histories

A q.enq(3)
A q:void
B p.enq(4)
B p:void
B q.deq()
B q:3
A q:enq(5)

match

match

match

Sequential Histories

A q.enq(3)
A q:void

B p.enq(4)
B p:void

B q.deq()
B q:3

A q:enq(5)

match

match

match

Final pending invocation OK

Sequential Histories

A q.enq(3)
A q:void
B p.enq(4)
B p:void
B q.deq()
B q:3
A q:enq(5)

match

match

match

Final pending invocation OK

Method calls of different threads do not interleave

Well-Formed Histories

H=

A q.enq(3)
B p.enq(4)
B p:void
B q.deq()
A q:void
B q:3

Well-Formed Histories

Per-thread projections sequential

H=
A q.enq(3)
B p.enq(4)
B p:void
B q.deq()
A q:void
B q:3

H|B=
B p.enq(4)
B p:void
B q.deq()
B q:3

Well-Formed Histories

Per-thread projections sequential

H=
A q.enq(3)
B p.enq(4)
B p:void
B q.deq()
A q:void
B q:3

H|B=
B p.enq(4)
B p:void
B q.deq()
B q:3

H|A=
A q.enq(3)
A q:void

Equivalent Histories

Threads see the same thing in both $\left\{ \begin{array}{l} H|A = G|A \\ H|B = G|B \end{array} \right.$

H=

A q.enq(3)
B p.enq(4)
B p:void
B q.deq()
A q:void
B q:3

G=

A q.enq(3)
A q:void
B p.enq(4)
B p:void
B q.deq()
B q:3

Sequential Specifications

- A sequential specification is some way of telling whether a
  - Single-thread, single-object history
  - Is legal
- For example:
  - Pre and post-conditions
  - But plenty of other techniques exist …

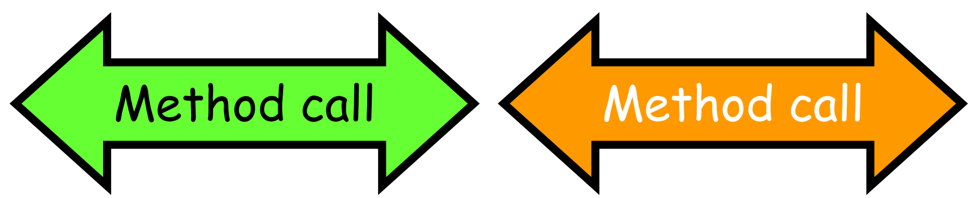
Legal Histories

- A sequential (multi-object) history H is legal if
  - For every object **x**
  - **H|x** is in the sequential spec for **x**

Precedence

A q.enq(3)
B p.enq(4)
B p.void
A q:void
B q.deq()
B q:3

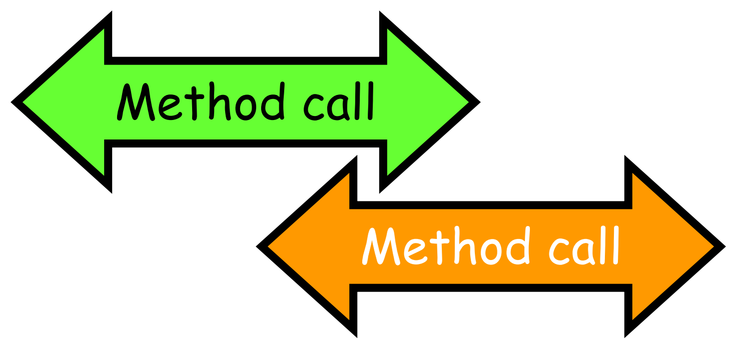
A method call precedes another if response event precedes invocation event



Non-Precedence

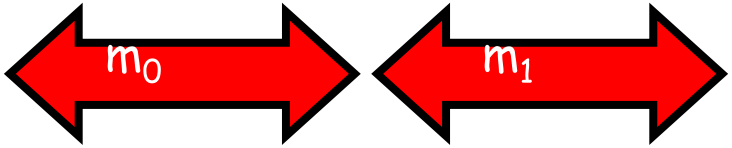
A q.enq(3)
B p.enq(4)
B p.void
B q.deq()
A q:void
B q:3

Some method calls overlap one another



Notation

- Given
  - History **H**
  - method executions $\mathbf{m_0}$ and $\mathbf{m_1}$ in **H**
- We say $m_0 \rightarrow_H m_1$, if
  - $\mathbf{m_0}$ precedes $\mathbf{m_1}$



- Relation $\mathbf{m_0} \rightarrow_H \mathbf{m_1}$ is a
  - Partial order
  - Total order if **H** is sequential

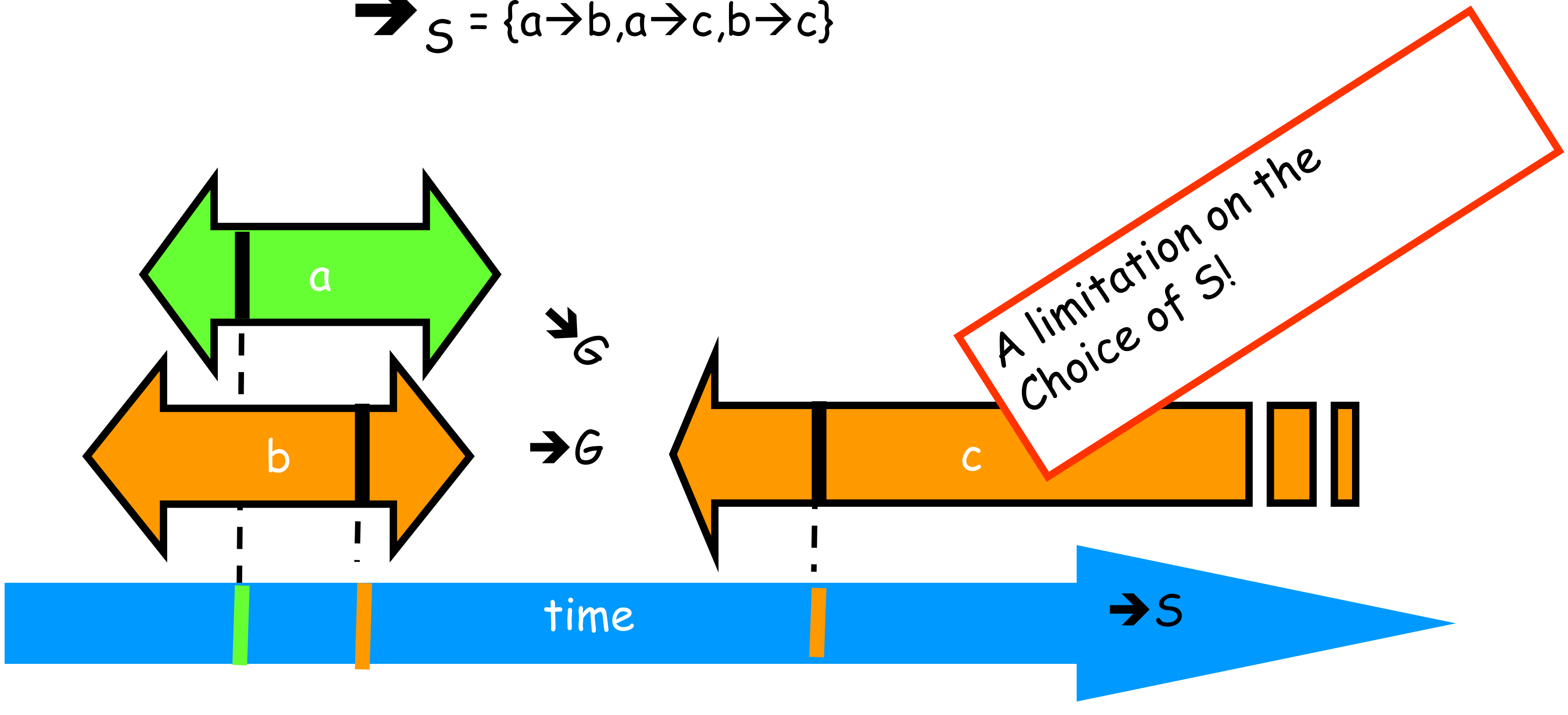
Linearizability

- History H is ***linearizable*** if it can be extended to **G** by
  - Appending zero or more responses to pending invocations
  - Discarding other pending invocations
- So that **G** is equivalent to
  - Legal sequential history **S**
  - where $\rightarrow_G \subset \rightarrow_S$

What is $\rightarrow_G \subset \rightarrow_S$

$\rightarrow_G = \{a \rightarrow c, b \rightarrow c\}$

$\rightarrow_S = \{a \rightarrow b, a \rightarrow c, b \rightarrow c\}$

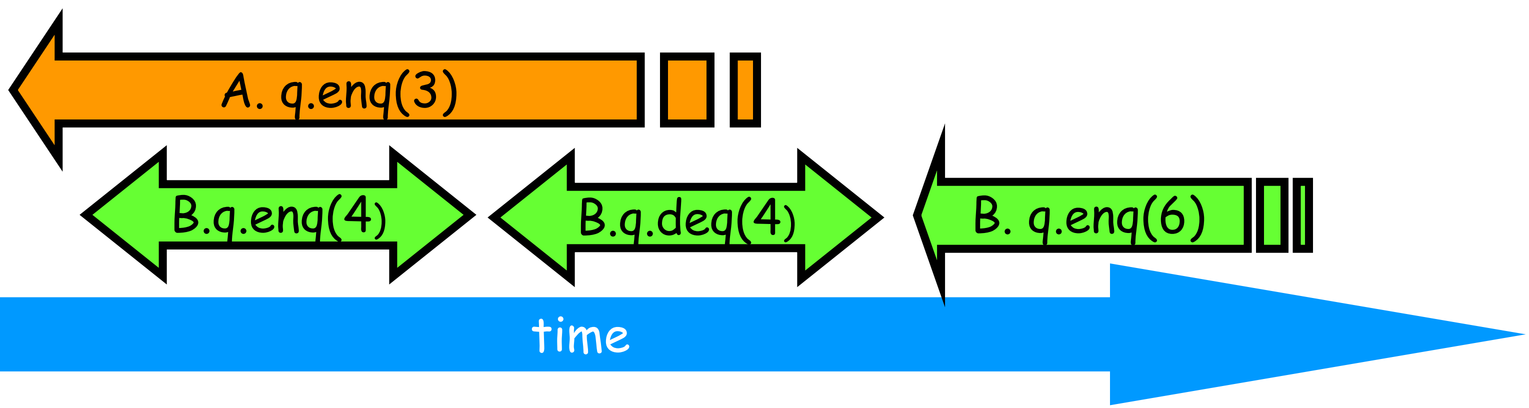


Remarks

- Some pending invocations
  - Took effect, so keep them
  - Discard the rest
- Condition $\rightarrow_{G} \subset \rightarrow_{S}$
  - Means that **S** respects "real-time order" of **G**

Example

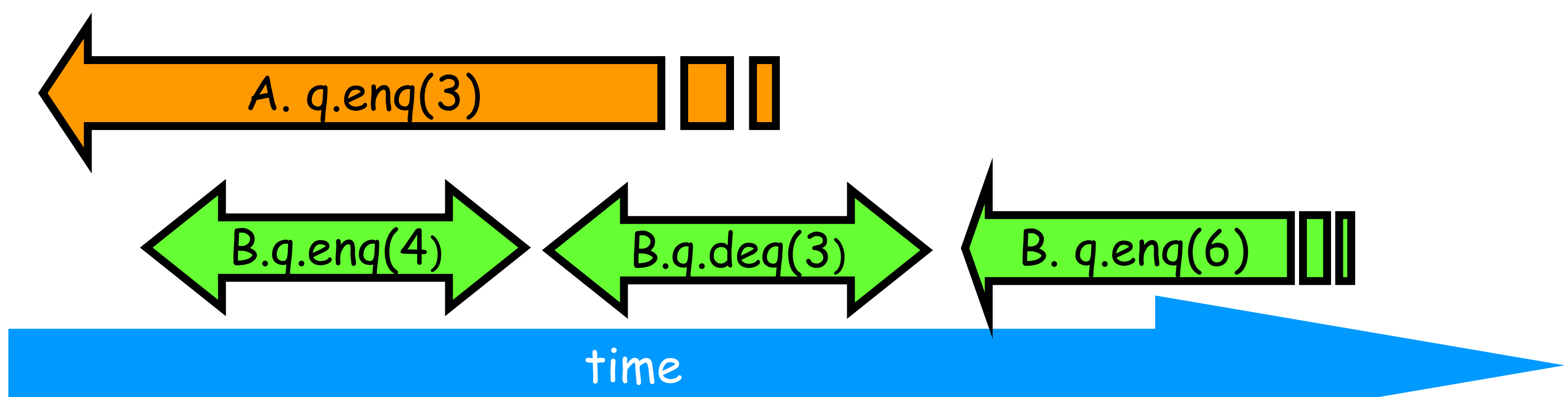
A q.enq(3)
B q.enq(4)
B q:void
B q.deq()
B q:4
B q:enq(6)



Example

A q.enq(3)
B q.enq(4)
B q:void
B q.deq()
B q:4
B q:enq(6)

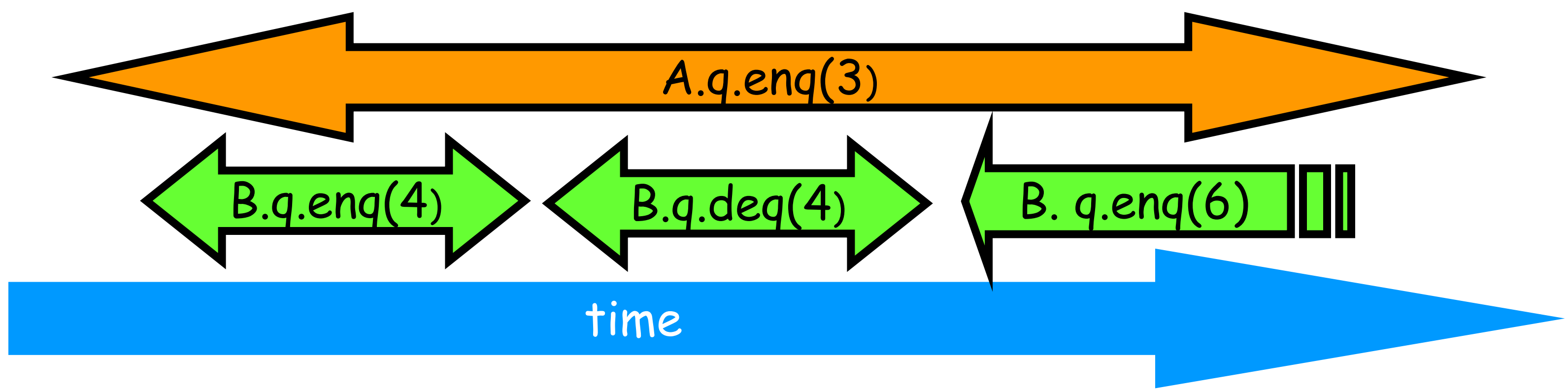
Complete this pending invocation



Example

A q.enq(3)
B q.enq(4)
B q:void
B q.deq()
B q:4
B q:enq(6)
A q:void

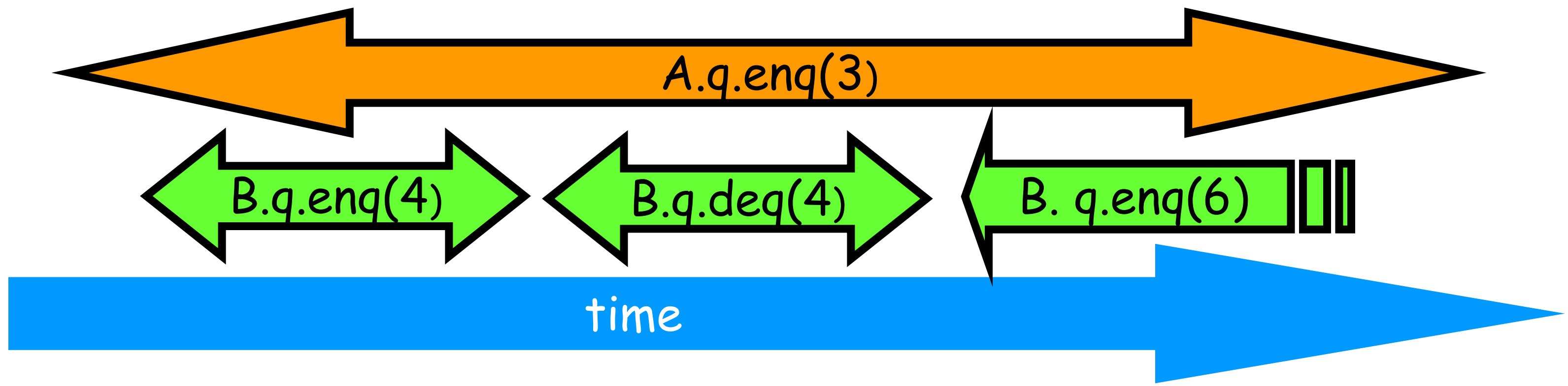
Complete this pending invocation



Example

A q.enq(3)
B q.enq(4)
B q:void
B q.deq()
B q:4
B q:enq(6)
A q:void

discard this one

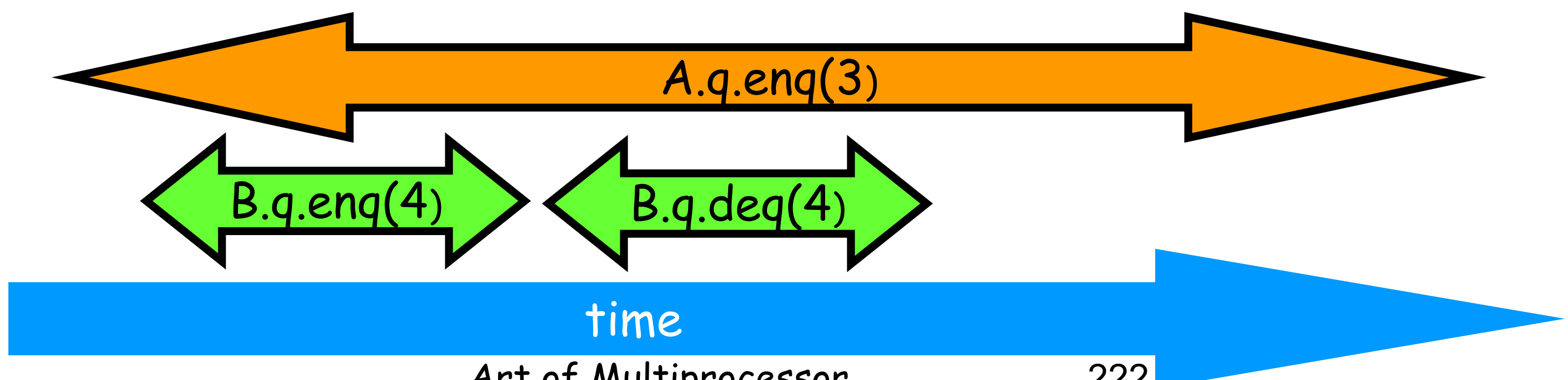


Example

A q.enq(3)
B q.enq(4)
B q:void
B q.deq()
B q:4

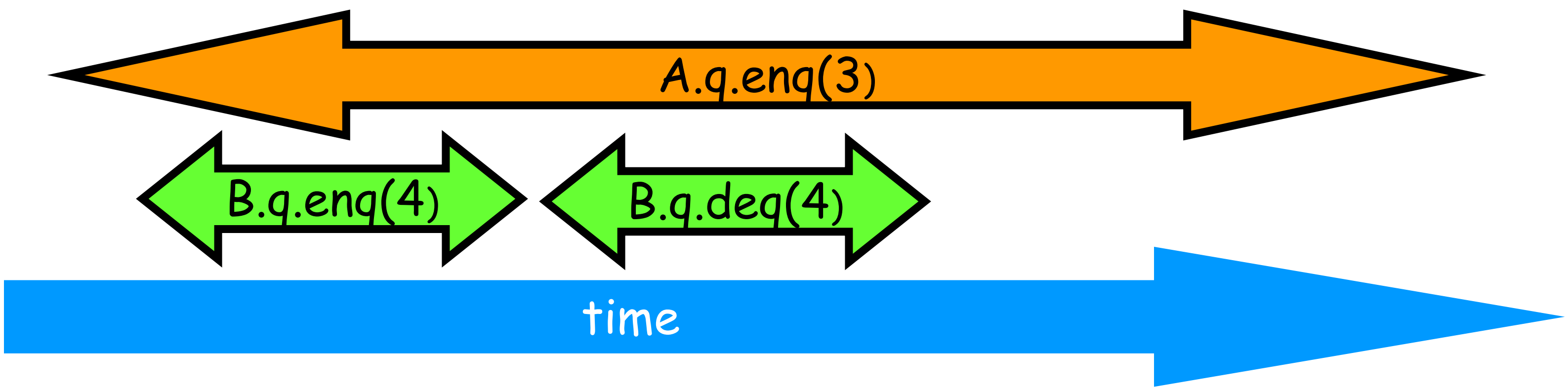
discard this one

A q:void



Example

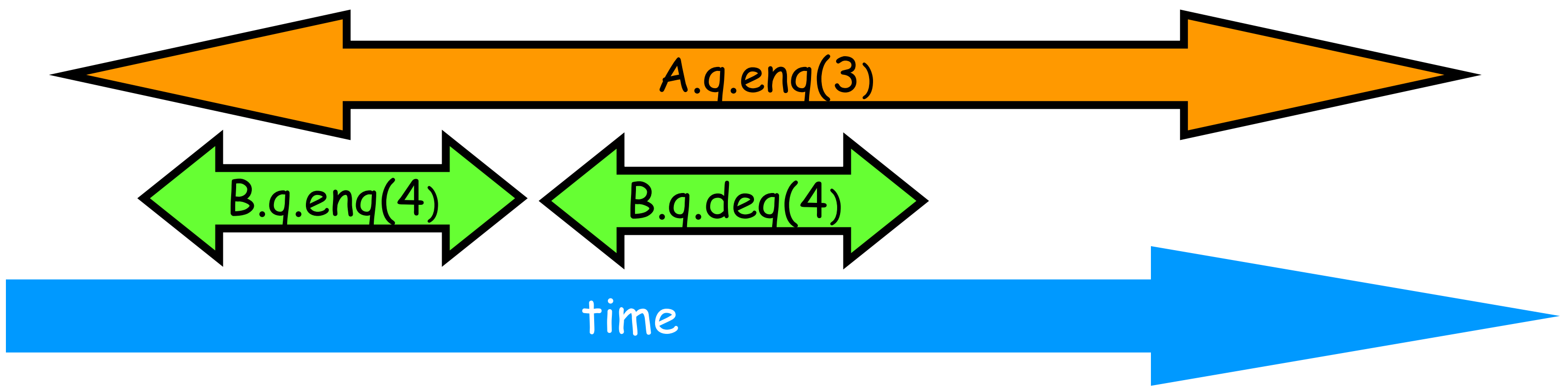
A q.enq(3)
B q.enq(4)
B q:void
B q.deq()
B q:4
A q:void



Example

A q.enq(3)
B q.enq(4)
B q:void
B q.deq()
B q:4
A q:void

B q.enq(4)
B q:void
A q.enq(3)
A q:void
B q.deq()
B q:4

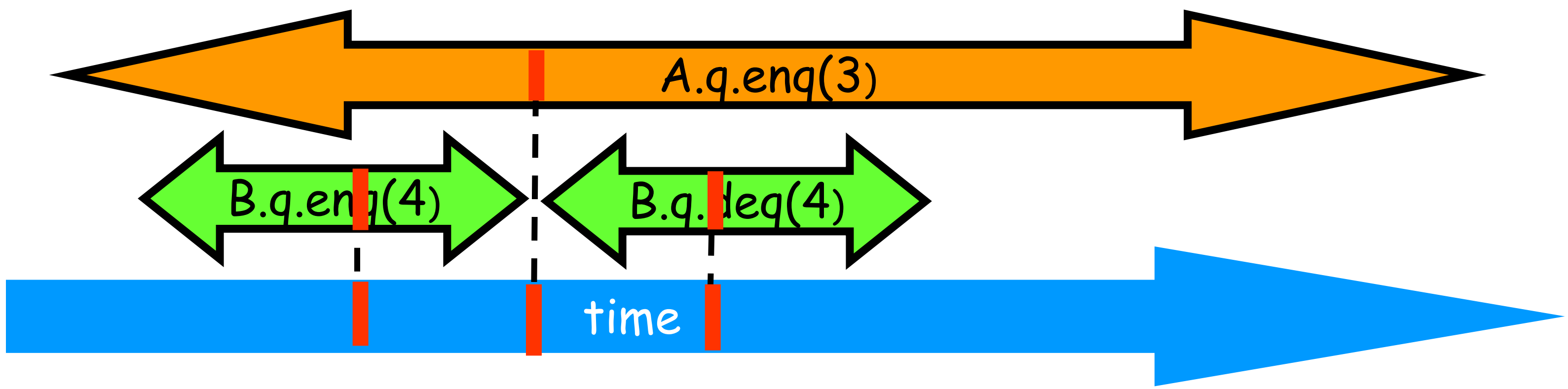


Example

A q.enq(3)
B q.enq(4)
B q:void
B q.deq()
B q:4
A q:void

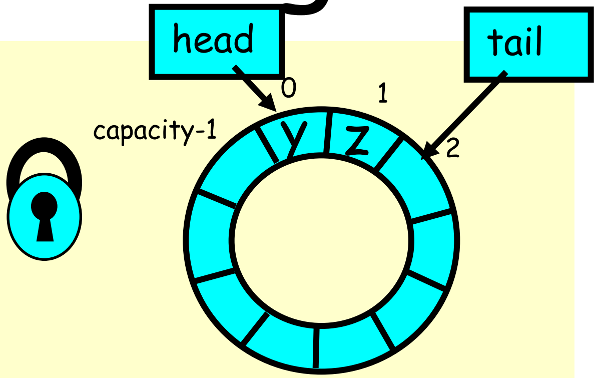
Equivalent sequential history

B q.enq(4)
B q:void
A q.enq(3)
A q:void
B q.deq()
B q:4



Reasoning About Linearizability: Locking

```
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();
  }
}
```



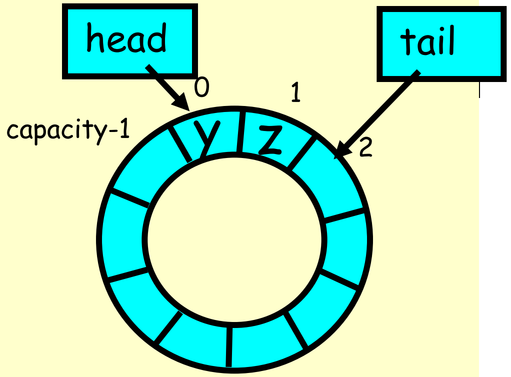
Reasoning About Linearizability: Locking

```
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();
  }
}
```

Linearization points are when locks are released

More Reasoning: Wait-free

```
public class WaitFreeQueue {

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

  public void enq(Item x) {
    if (tail-head == capacity) throw
        new FullException();
    items[tail % capacity] = x; tail++;
  }
  public Item deq() {
     if (tail == head) throw
        new EmptyException();
     Item item = items[head % capacity]; head++;
     return item;
}}
```

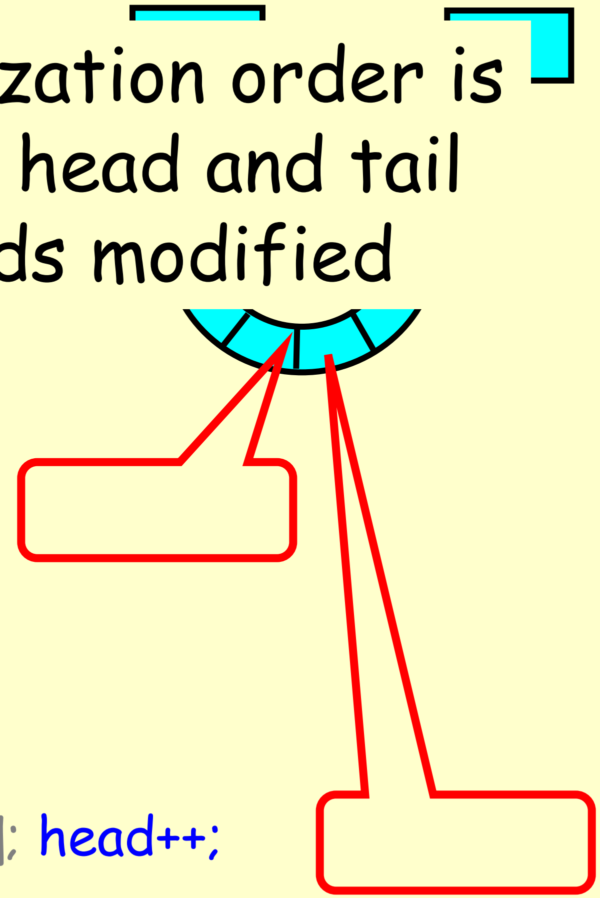


More Reasoning: Wait-free

```
public class W        eQueue {

  int hea
  ite                 bject[c

                q(Item x) {
           ead == capacity) throw
          w FullException();
     ems[tail % capacity] = x; tail++;
  }
  public Item deq() {
     if (tail == head) throw
         new EmptyException();
     Item item = items[head % capacity]; head++;
     return item;
}}
```

Remember that there is only one enqueuer and only one dequeuer

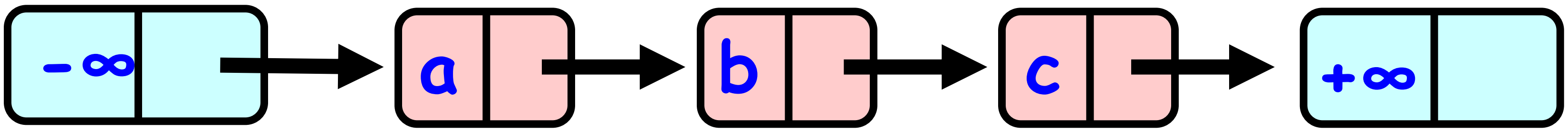
Linearization order is order head and tail fields modified



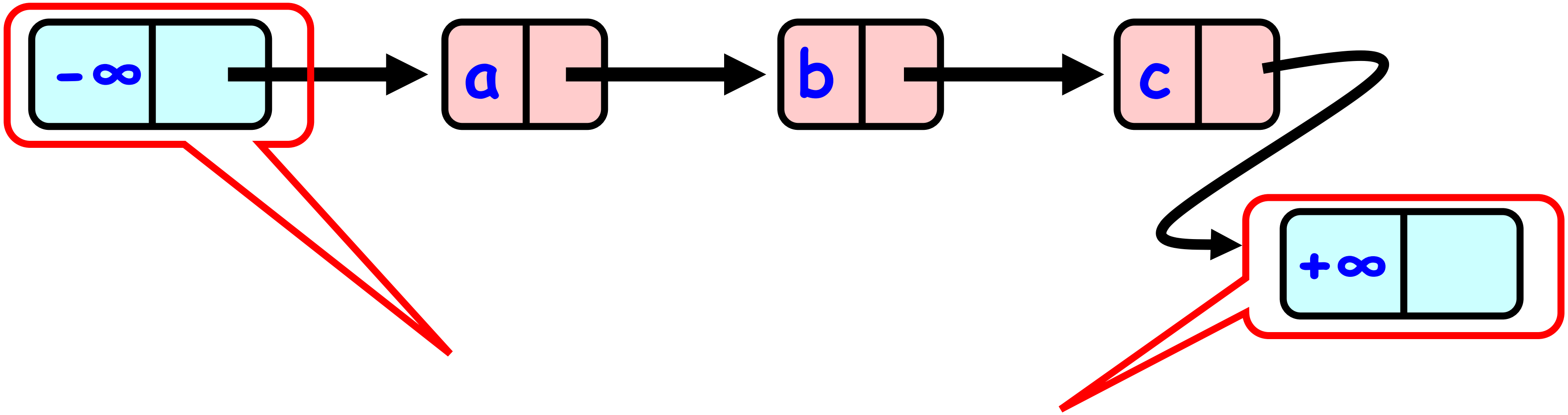
Linearizability: Summary

- Powerful specification tool for shared objects
- Allows us to capture the notion of objects being "atomic"
- Don't leave home without it

Ordered linked list implementation of a set



Defining the linked list



Sorted with Sentinel nodes
(min & max possible keys)

Defining concurrent methods properties

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

Defining concurrent methods properties

- Rep-Invariant:
  - The invariant on our concrete Representation = on the list.
  - Preserved by methods.
  - Relied on by methods.
  - Allows us to reason about each method in isolation without considering how they interact.

Defining concurrent methods properties

- Our Rep-invariant:
  - Sentinel nodes
    - tail reachable from head.
  - Sorted
  - No duplicates

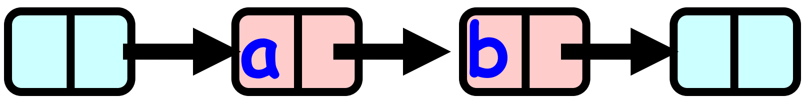
- Depends on the implementation.

Defining concurrent methods properties

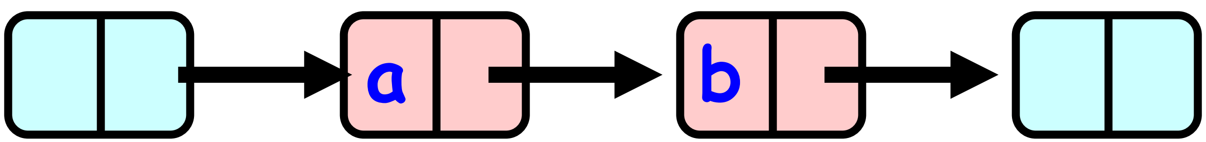
- Abstraction Map:
- S(List) =
  - { x | there exists a such that
    - a reachable from head and
    - a.item = x
  - }

- Depends on the implementation.

Abstract Data Types

- Example:
  - S(

) = {a,b}
- Concrete representation:



- Abstract Type:
  - {a, b}

Defining concurrent methods properties

- *Wait-free:* Every call to the function finishes in a finite number of steps.

Supposing the Scheduler is fair:

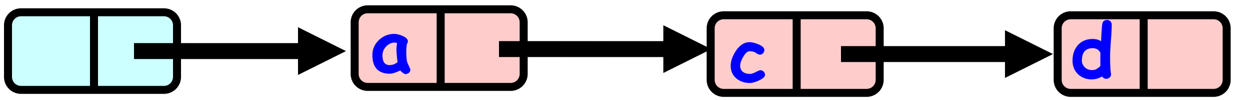
- *Starvation-free:* every thread calling the method eventually returns.

Algorithms

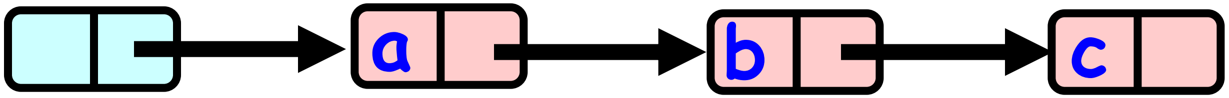
- Next: going throw each algorithm.
  - 1. Describing the algorithm.
  - 2. Explaining why every step of the algorithm is needed.
  - 3. Code review.
  - 4. Analyzing each method properties.
  - 5. Advantages / Disadvantages.
  - 6. Presenting running times for the implementation of the algorithm.
  - + Example of proving correctness for Remove(x) in FineGrained.

0.Sequential List Based Set

Add()

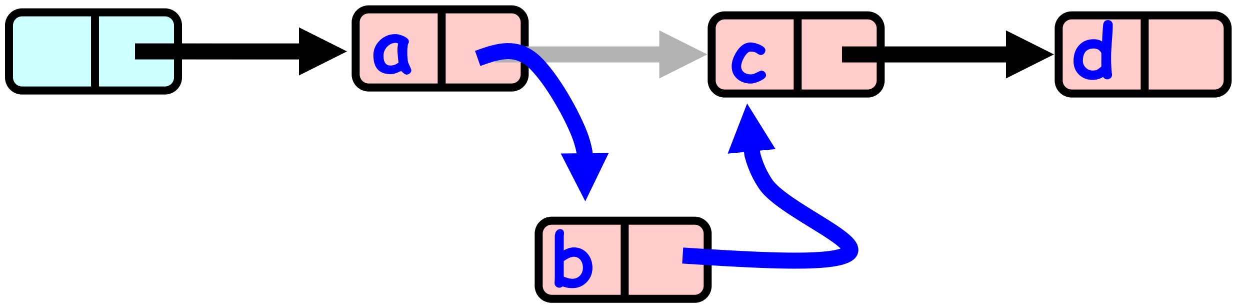


Remove()

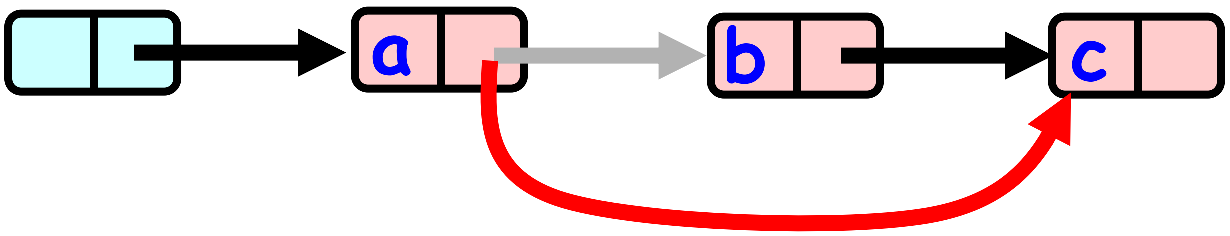


0.Sequential List Based Set

Add()



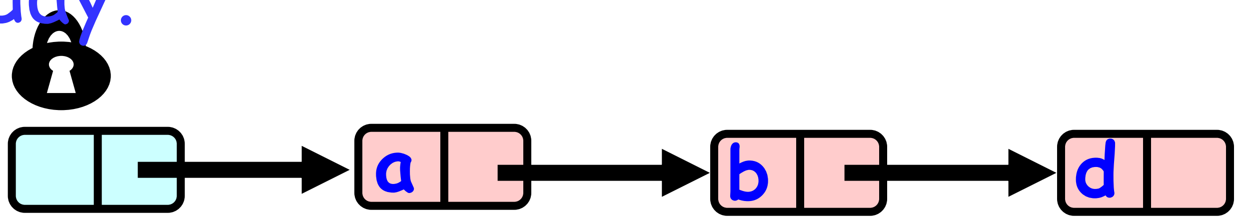
Remove()



1.Course Grained

1. Describing the algorithm:

- Most common implementation today.

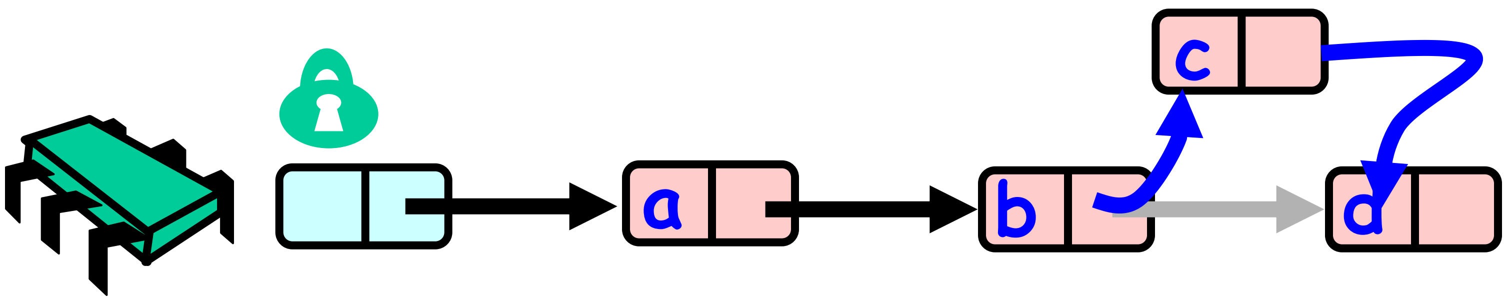


- Add(x) / Remove(x) / Contains(x):
  - Lock the entire list then perform the operation.

1.Course Grained

1. Describing the algorithm:

- Most common implementation today



- All methods perform operations on the list while holding the lock, so the execution is essentially sequential.

1.Course Grained

3. Code review:

Add:

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

Finding the place to add the item

Adding the item if it wasn't already in the list

1.Course Grained

3. Code review:

Remove:

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

Finding the item

Removing the item

1.Course Grained

3. Code review:

Contains:

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

Finding the item

Returning true if found

1.Course Grained

4. Methods properties:

- The implementation inherits its progress conditions from those of the Lock, and so assuming fair Scheduler:

  - If the Lock implementation is Starvation free

    Every thread will eventually get the lock and eventually the call to the function will return.

- So our implementation of Insert, Remove and Contains is Starvation-free

1.Course Grained

5. Advantages / Disadvantages:

Advantages:

- Simple.
- Obviously correct.

Disadvantages:

- High Contention.
- Bottleneck!

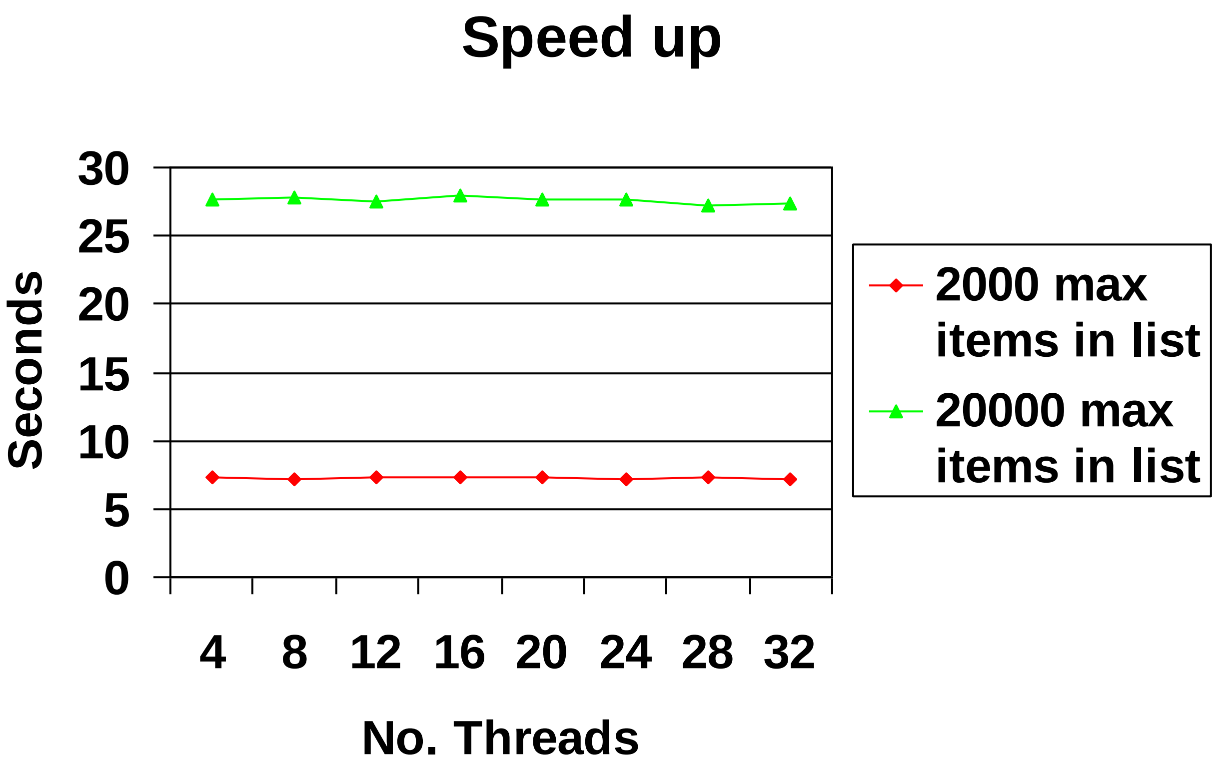
1.Course Grained

6. Running times:

- The tests were run on Aries – Supports 32 running threads. UltraSPARC T1 - Sun Fire T2000.
- Total of 200000 operations.
- 10% adds, 2% removes, 88% contains – normal work load percentages on a set.
- Each time the list was initialized with 100 elements.
- One run with a max of 20000 items in the list. Another with only 2000.

1.Course Grained

6. Running times:



2.Fine Grained

1. Describing the algorithm:

- Split object into pieces
  - Each piece has own lock.
  - Methods that work on disjoint pieces need not exclude each other.

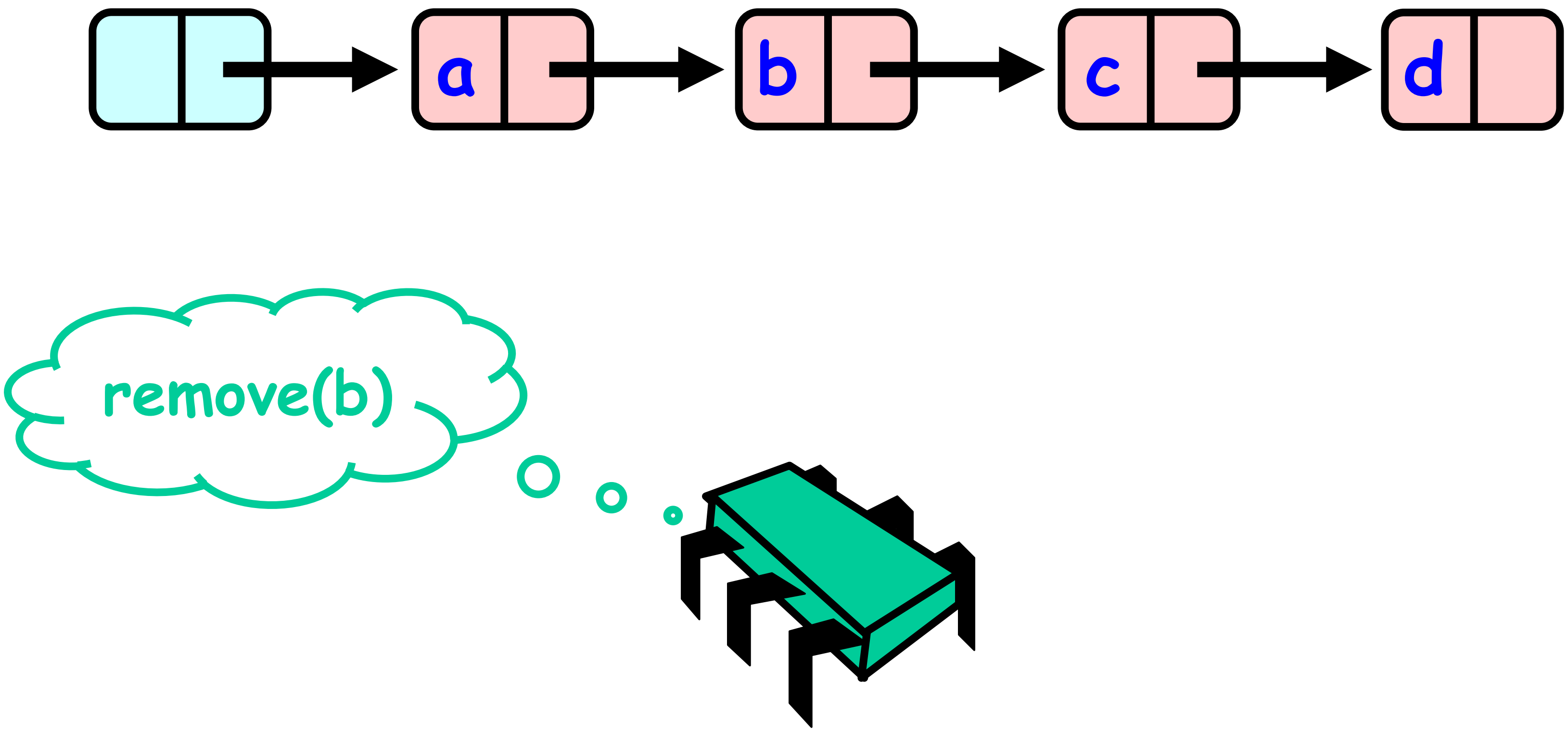
2.Fine Grained

1. Describing the algorithm:

- Add(x) / Remove(x) / Contains(x):
  - Go throw the list, lock each node and release only after the lock of the next element has been acquired.
  - Once you have reached the right point of the list perform the Add / Remove / Contains operation.

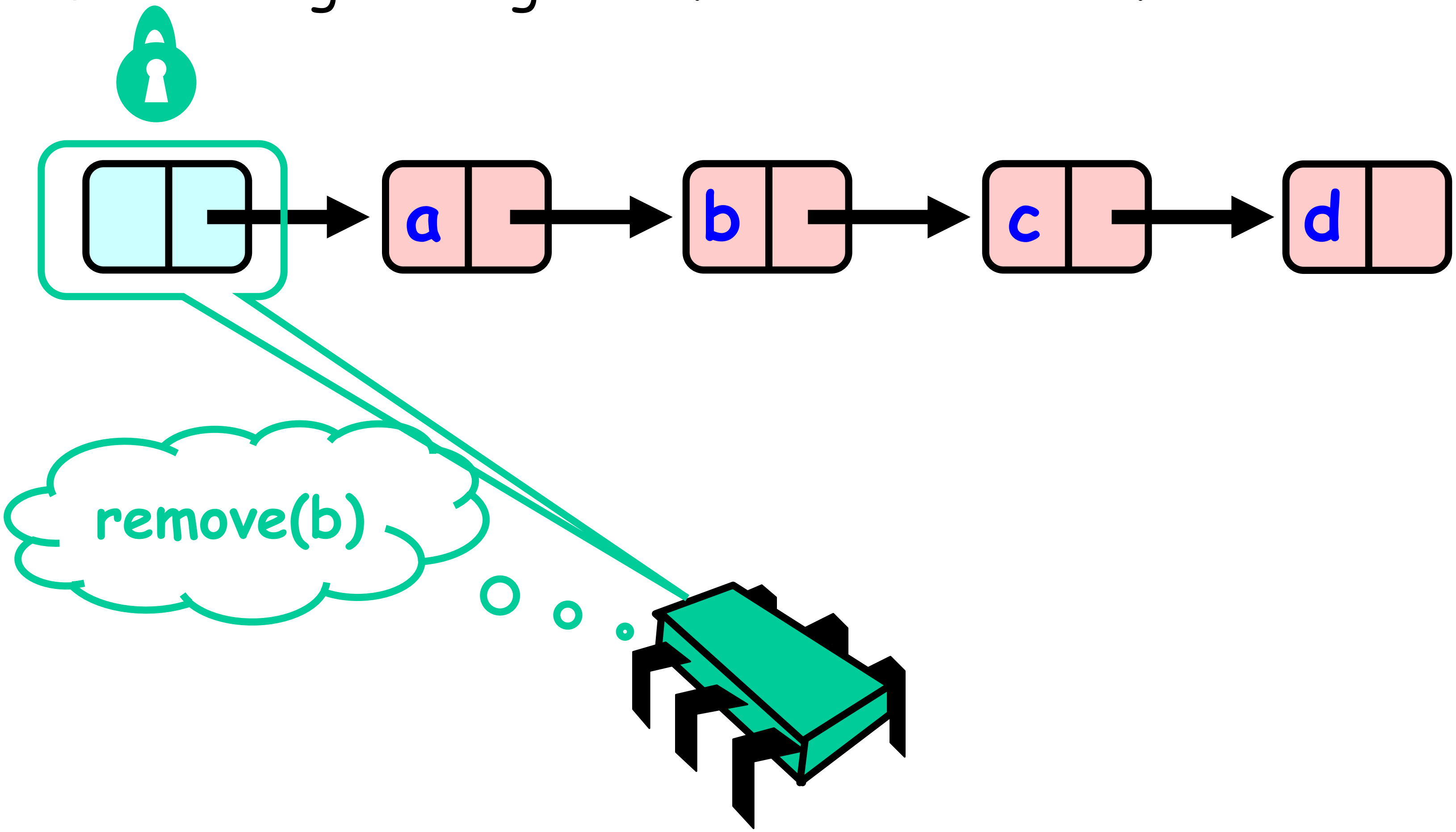
2.Fine Grained

1. Describing the algorithm: illustrated Remove.



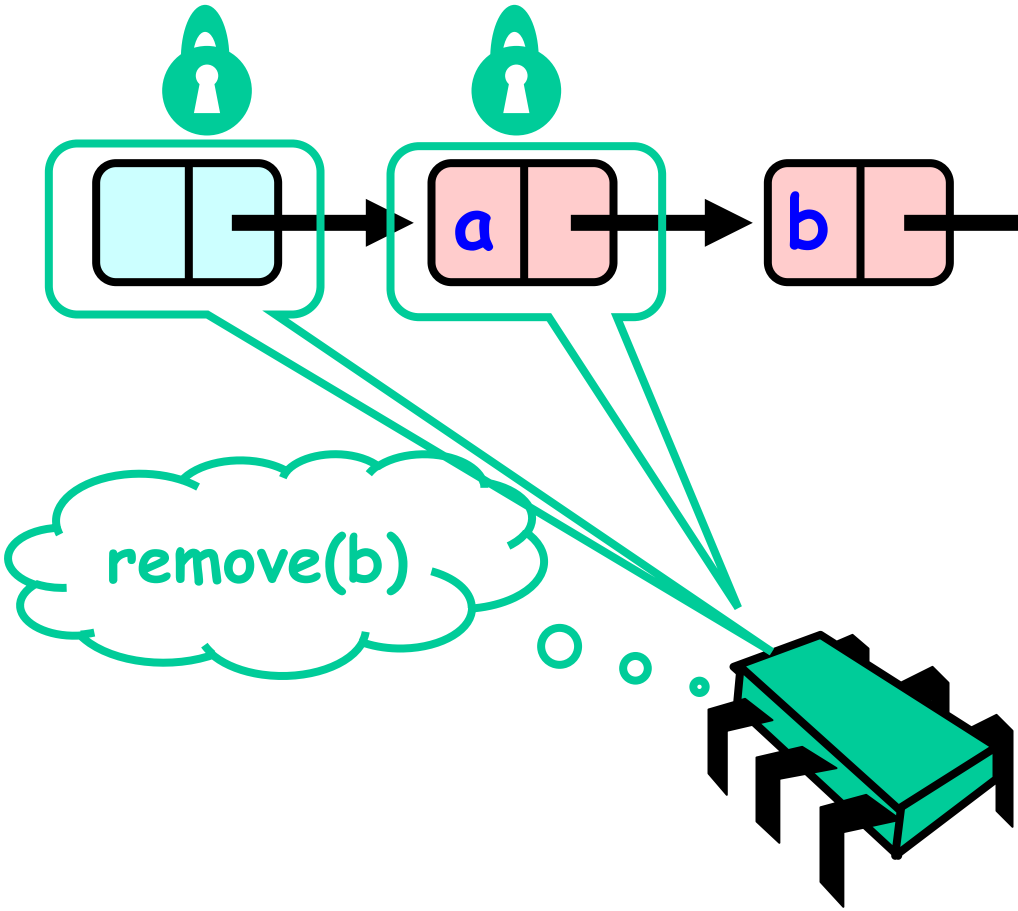
2.Fine Grained

1. Describing the algorithm: illustrated Remove.



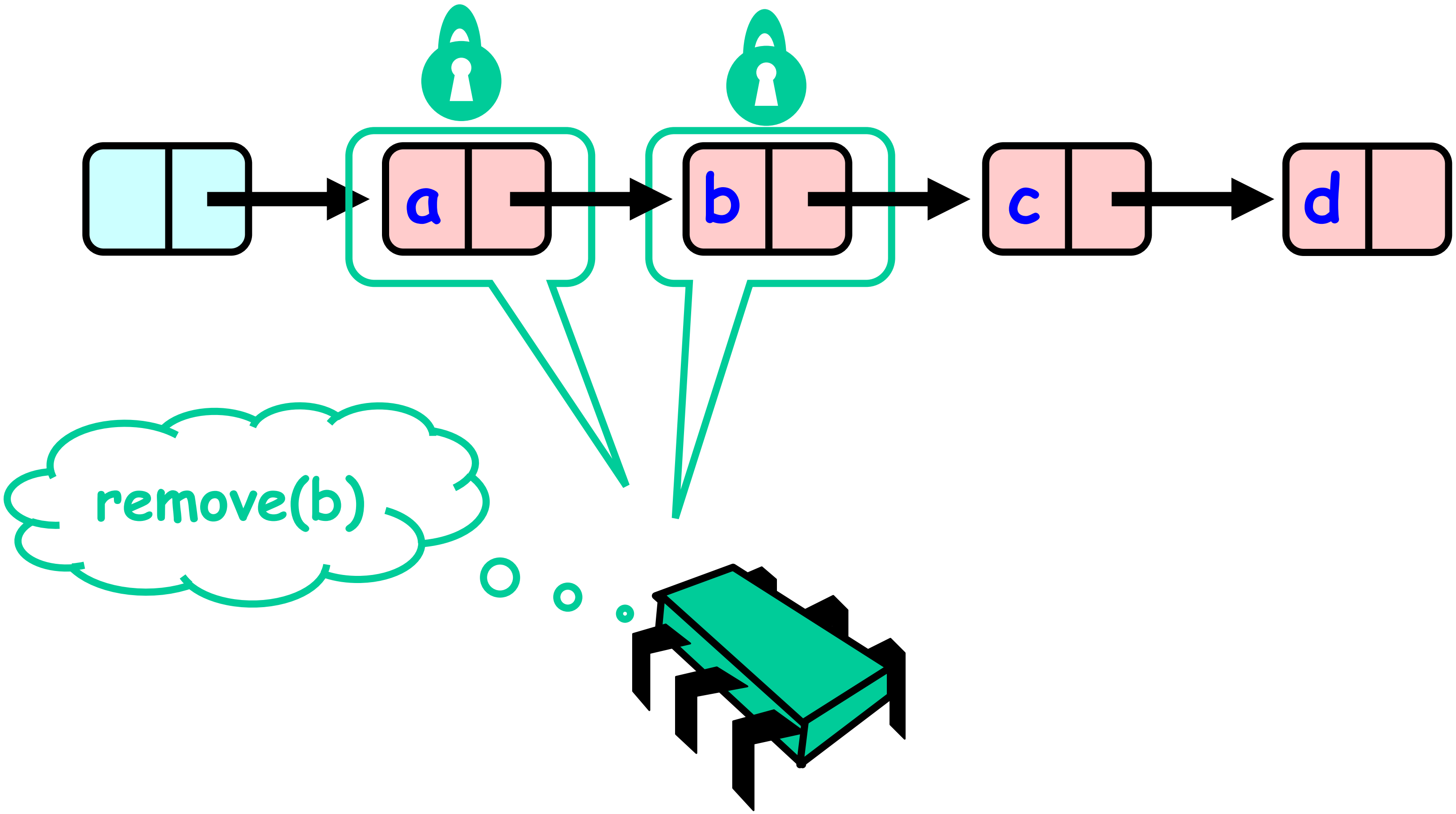
2.Fine Grained

1. Describing the algorithm: illustrated Remove.



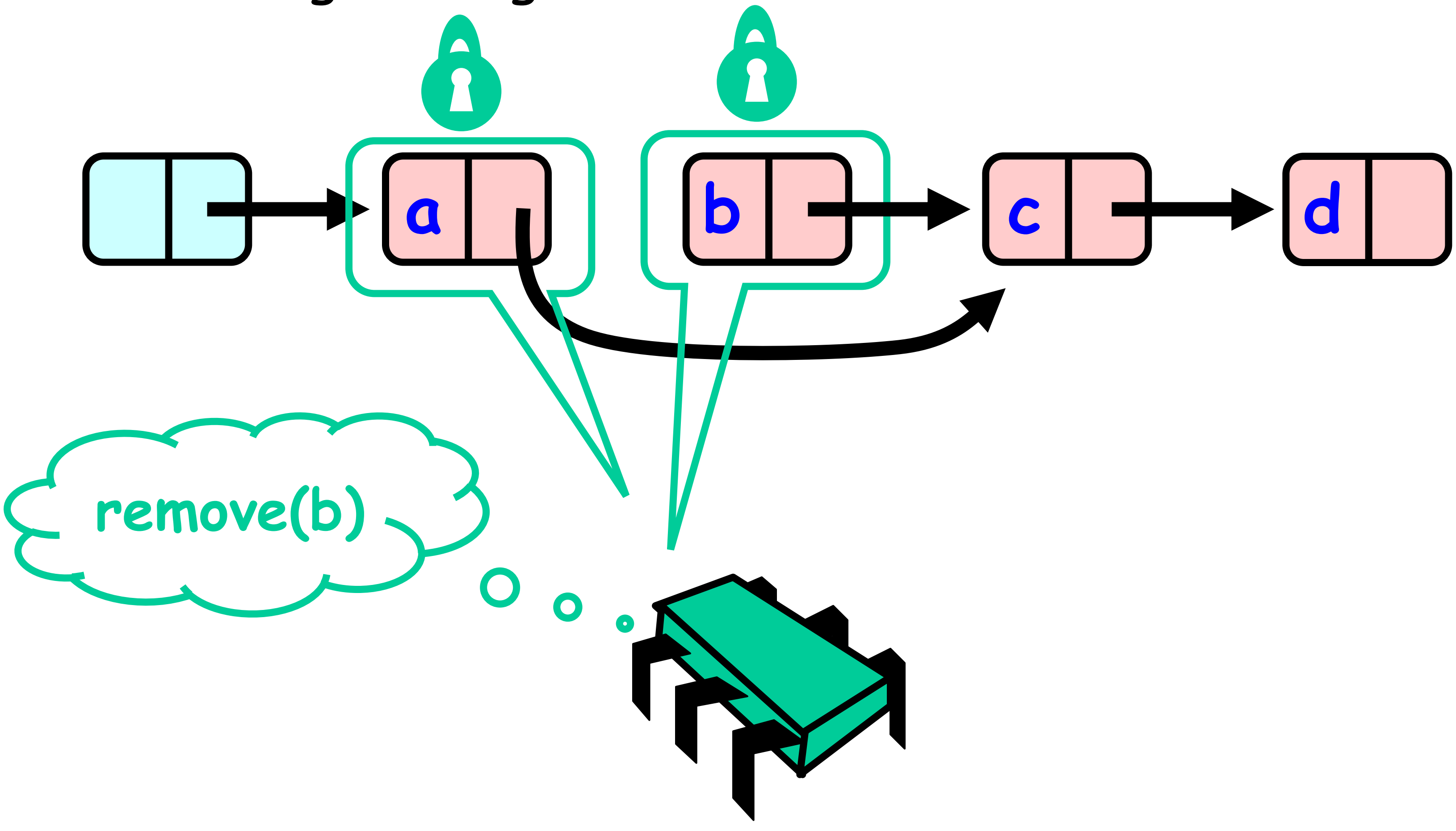
2.Fine Grained

1. Describing the algorithm: illustrated Remove.



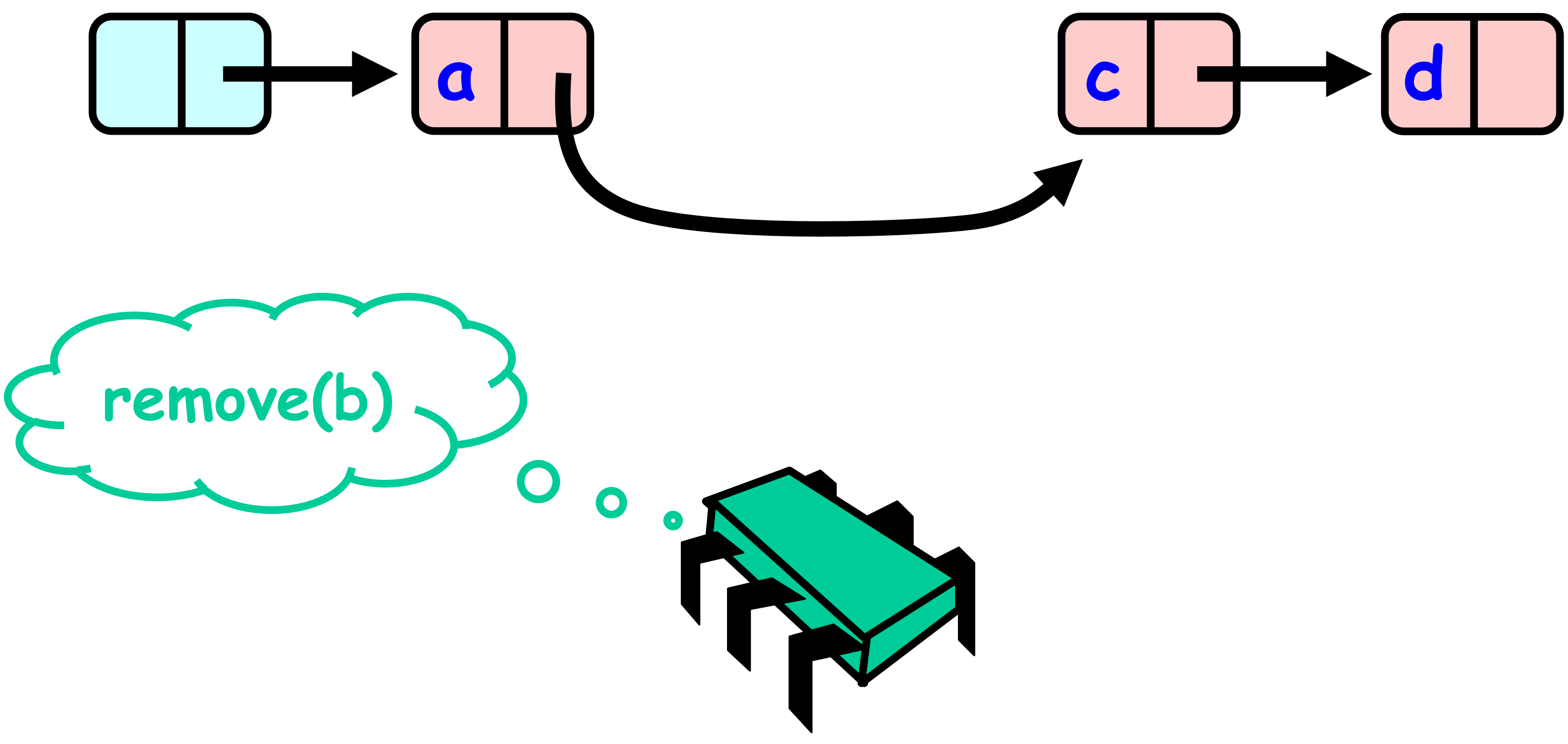
2.Fine Grained

1. Describing the algorithm: illustrated Remove.



2.Fine Grained

1. Describing the algorithm: illustrated Remove.



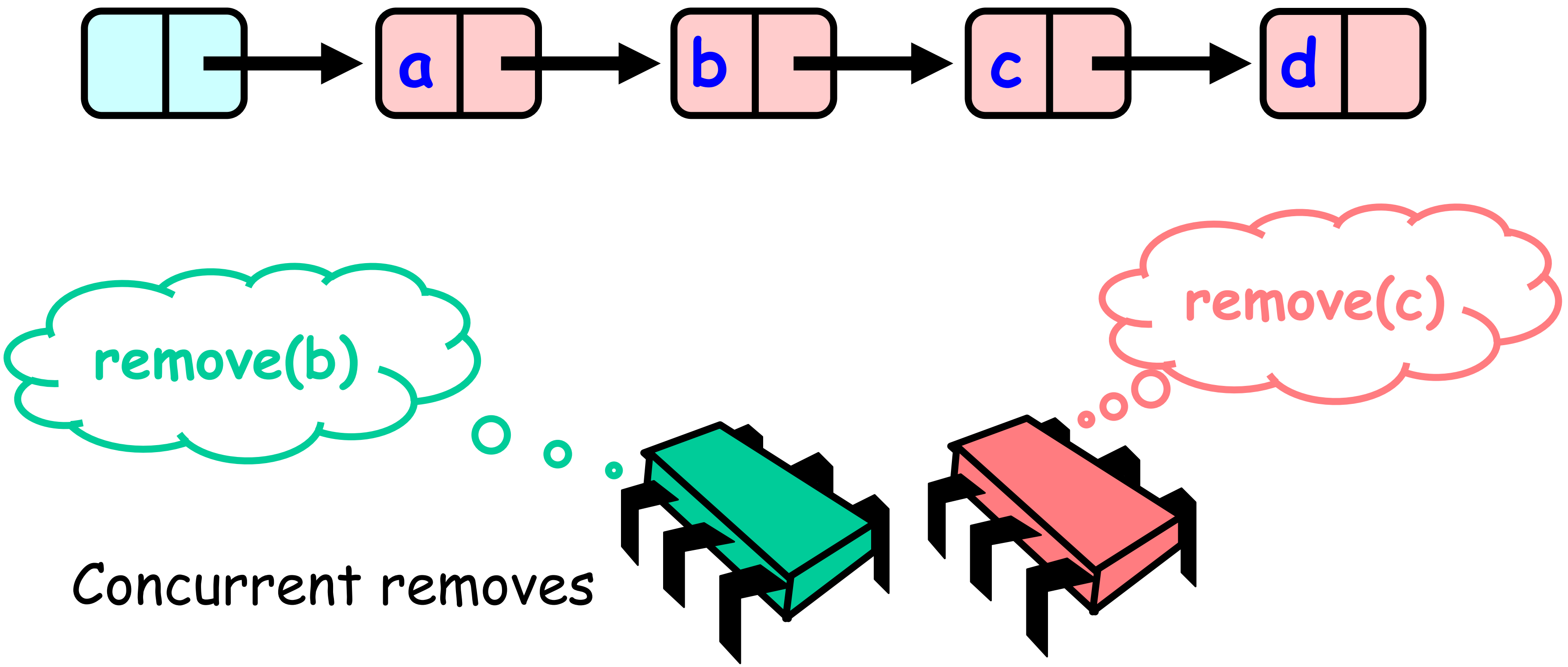
2.Fine Grained

2. Explaining why every step is needed.

Why do we need to always hold 2 locks?

2.Fine Grained

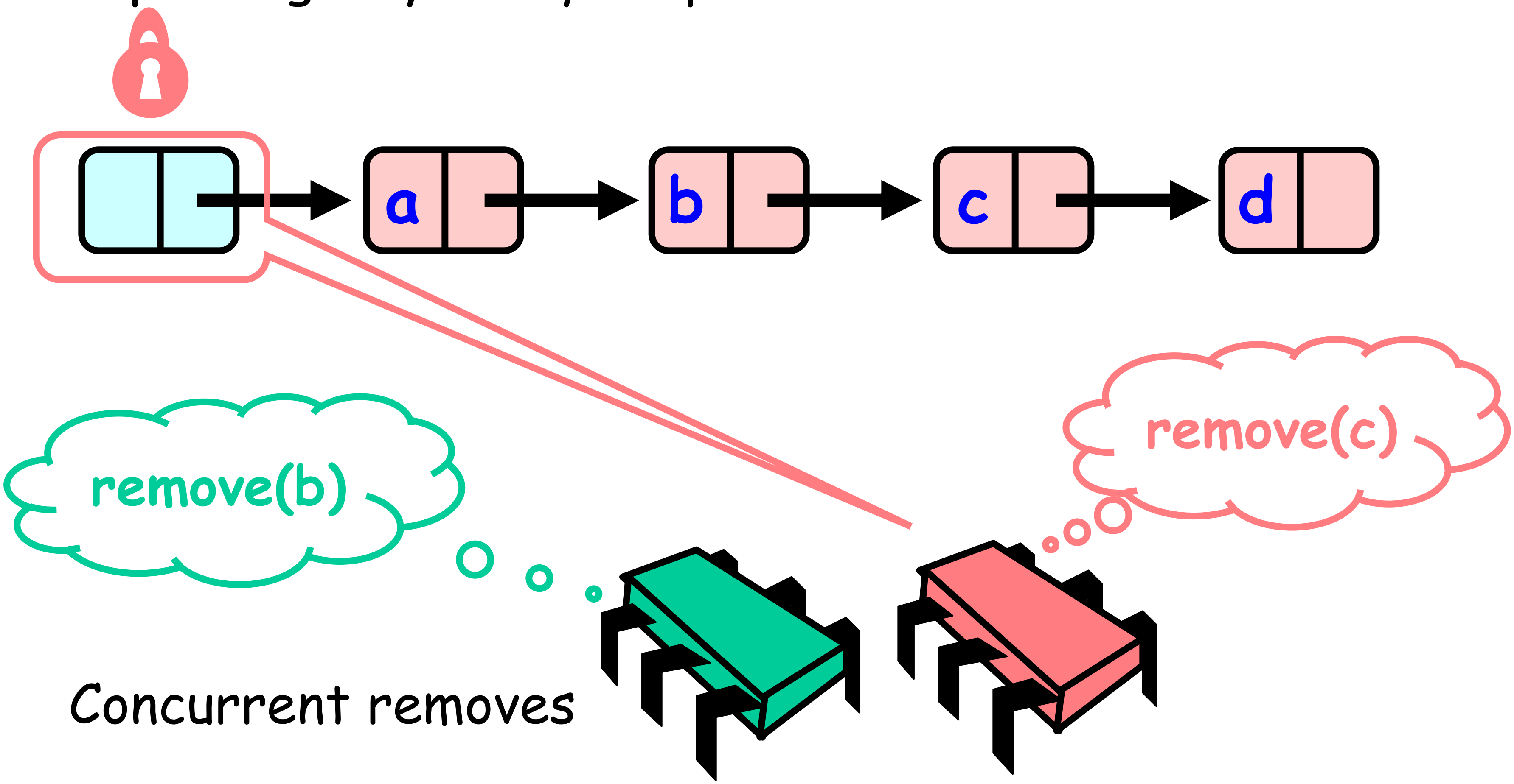
2. Explaining why every step is needed.



Concurrent removes

2.Fine Grained

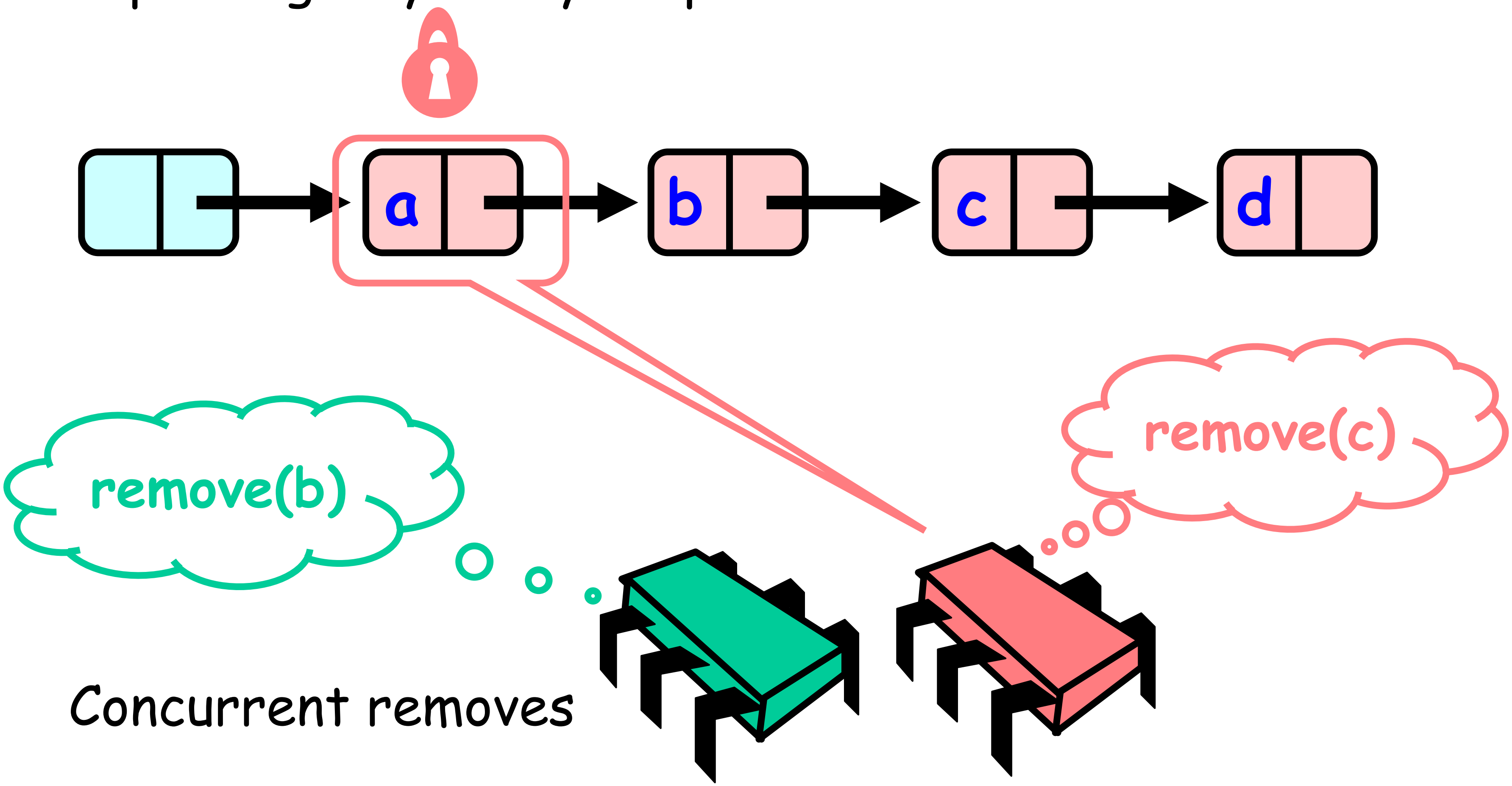
2. Explaining why every step is needed.



Concurrent removes

2.Fine Grained

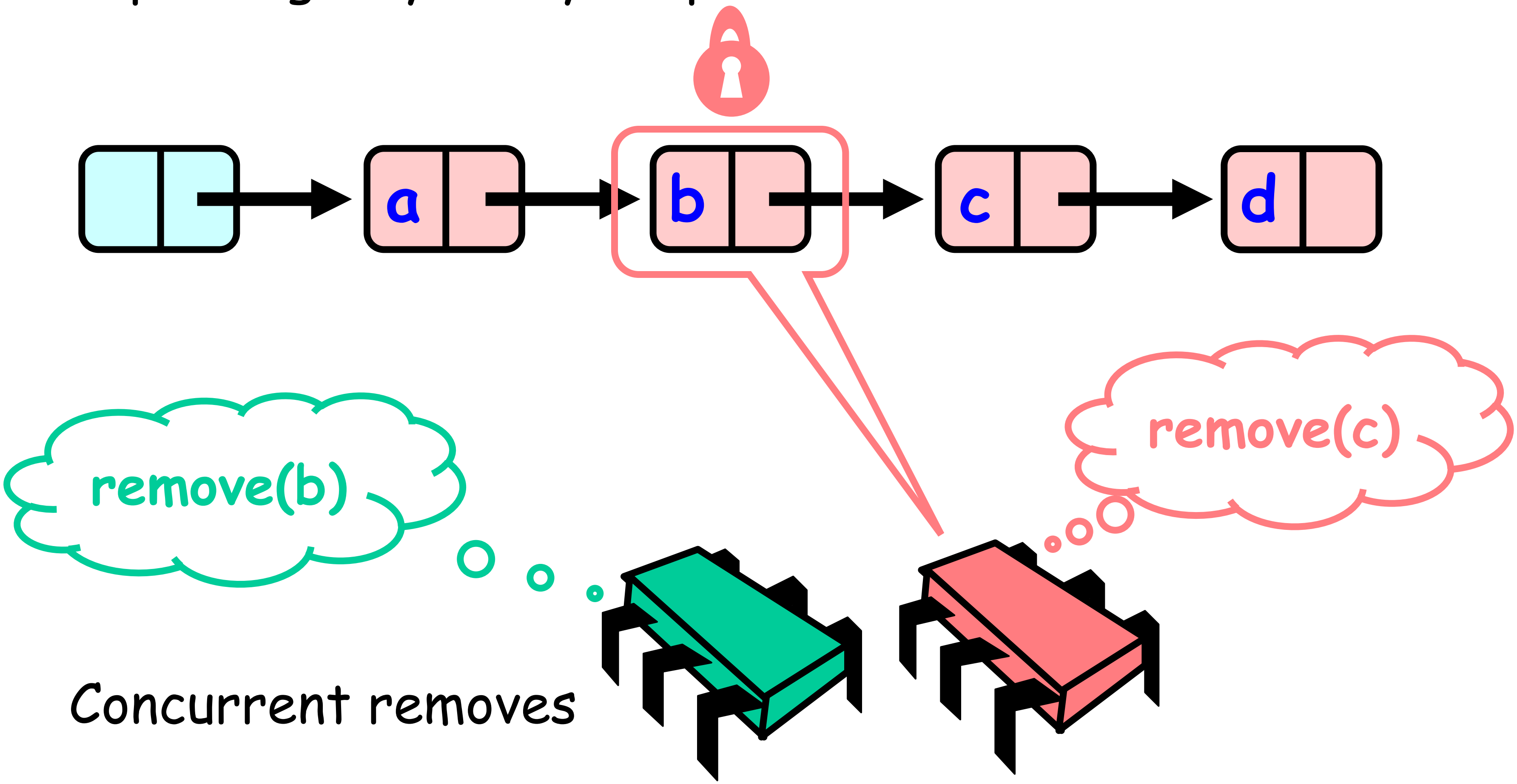
2. Explaining why every step is needed.



Concurrent removes

2.Fine Grained

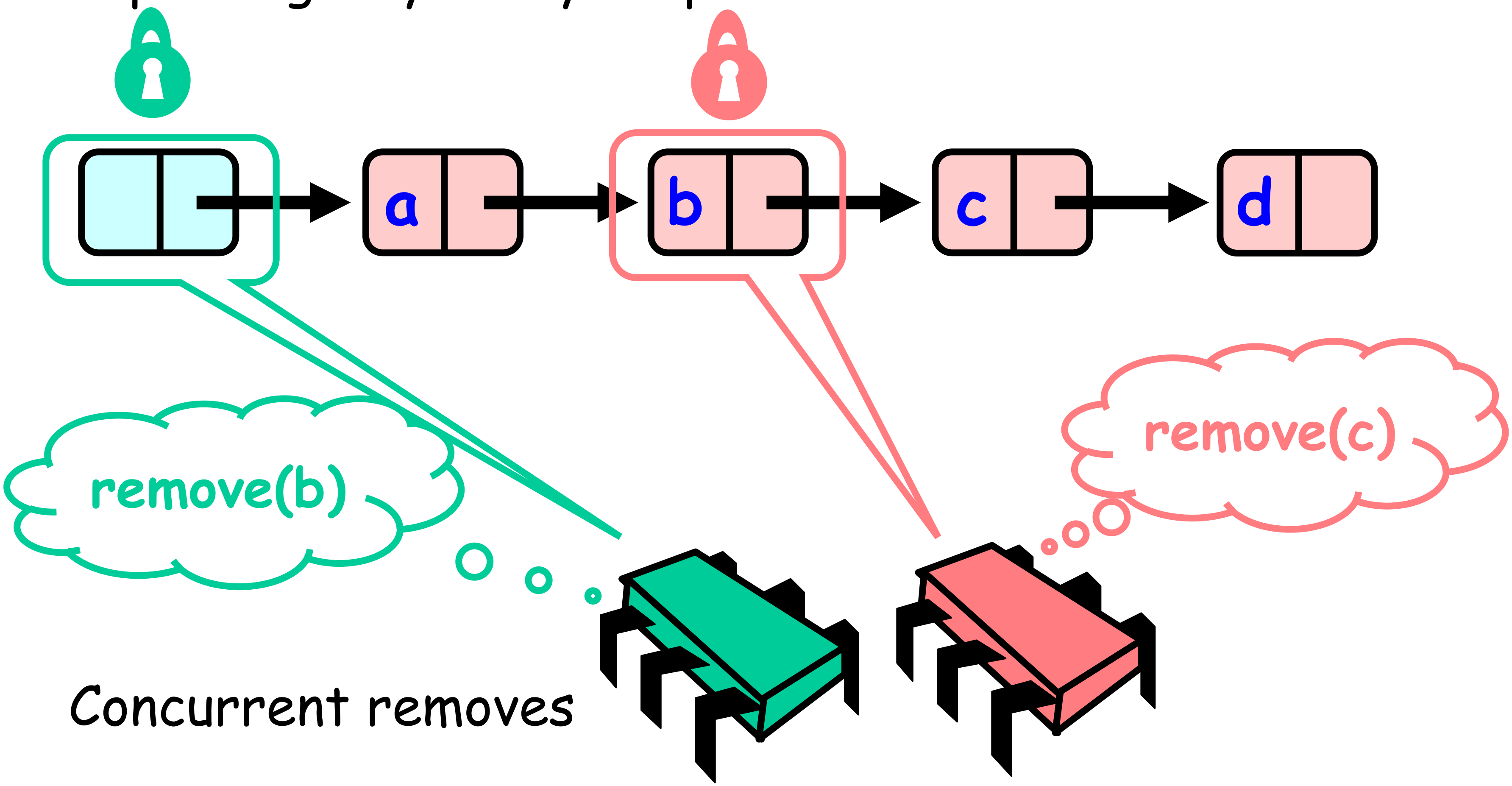
2. Explaining why every step is needed.



Concurrent removes

2.Fine Grained

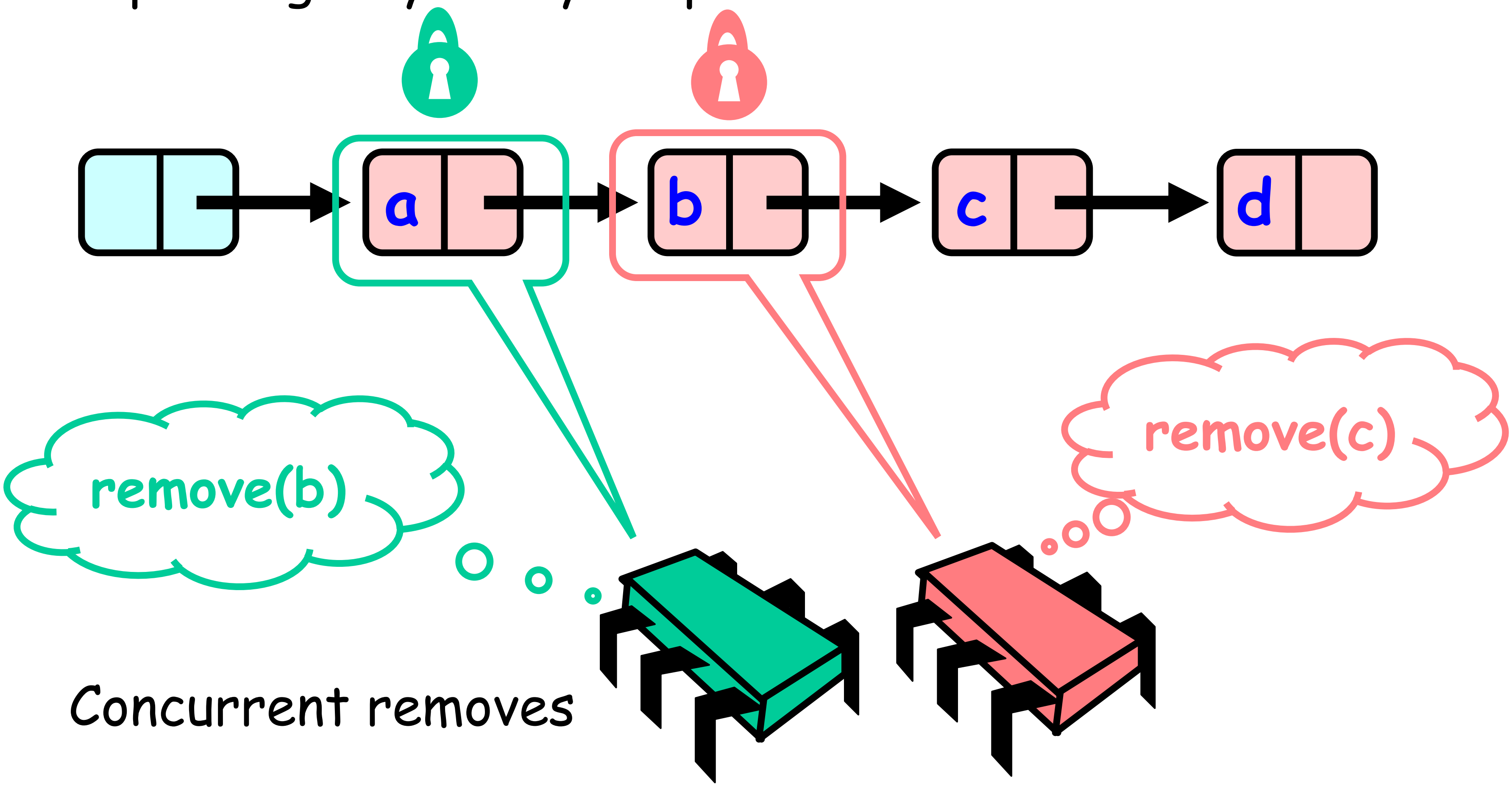
2. Explaining why every step is needed.



Concurrent removes

2.Fine Grained

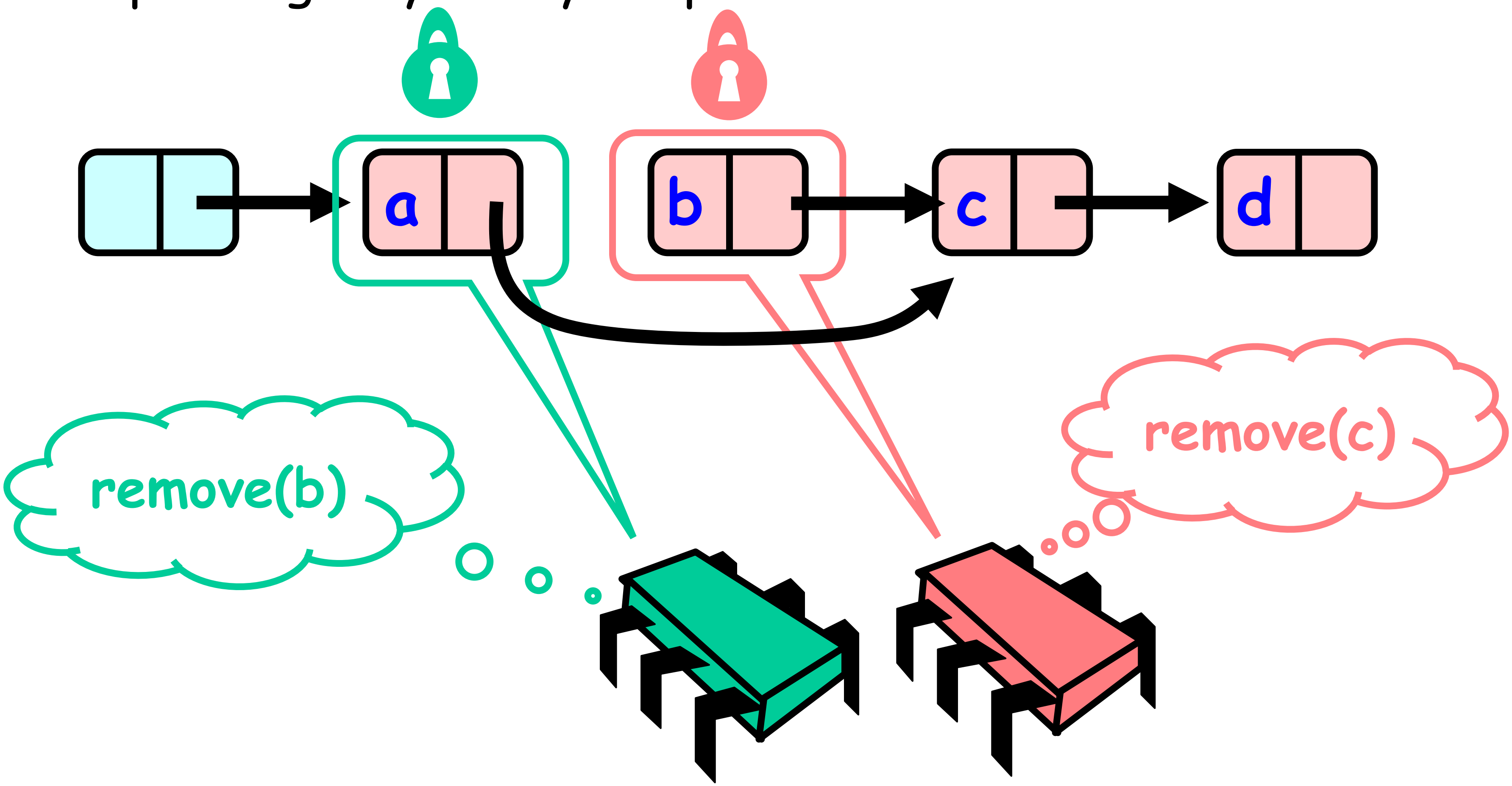
2. Explaining why every step is needed.



Concurrent removes

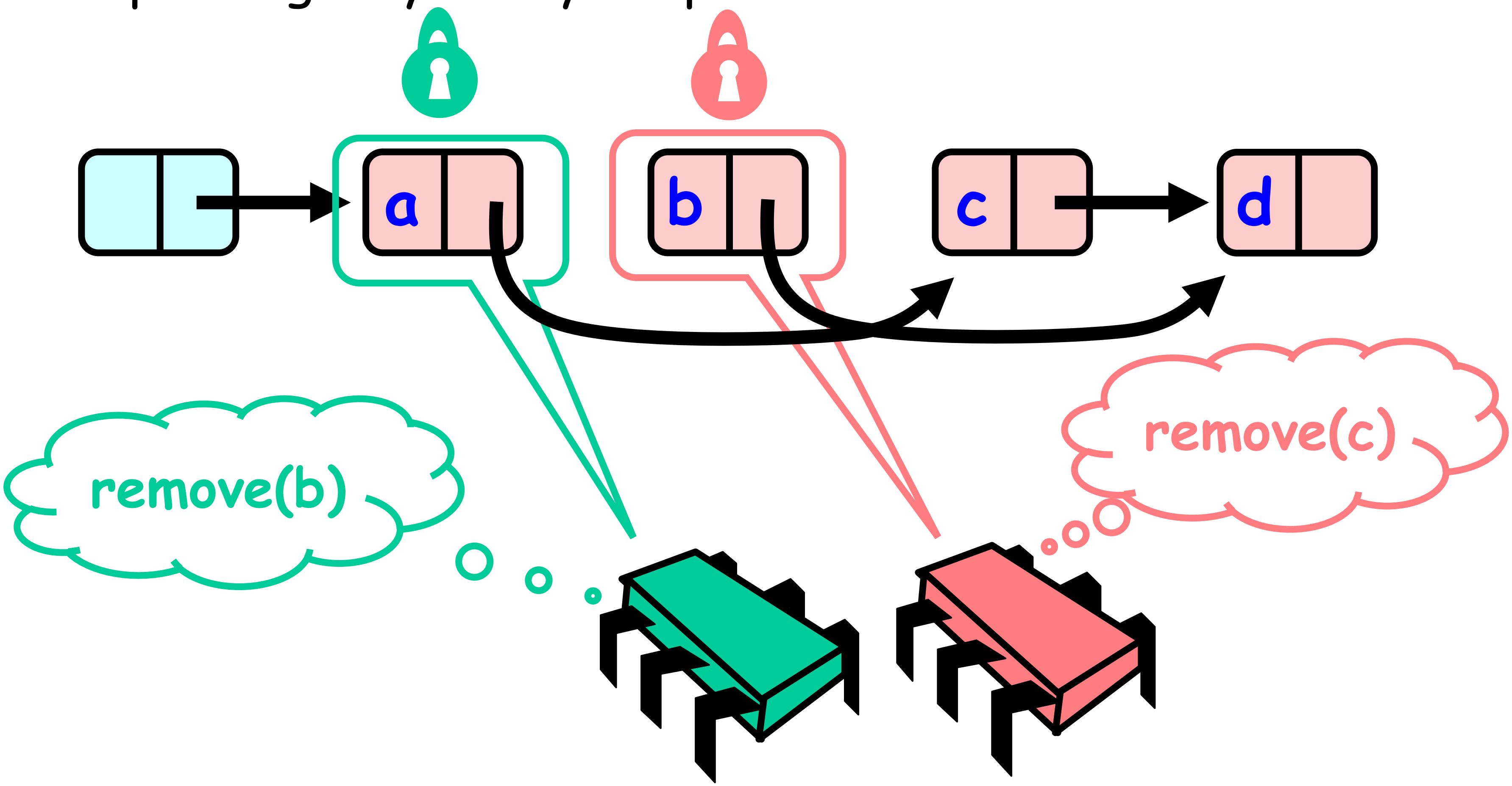
Concurrent Removes

2. Explaining why every step is needed.



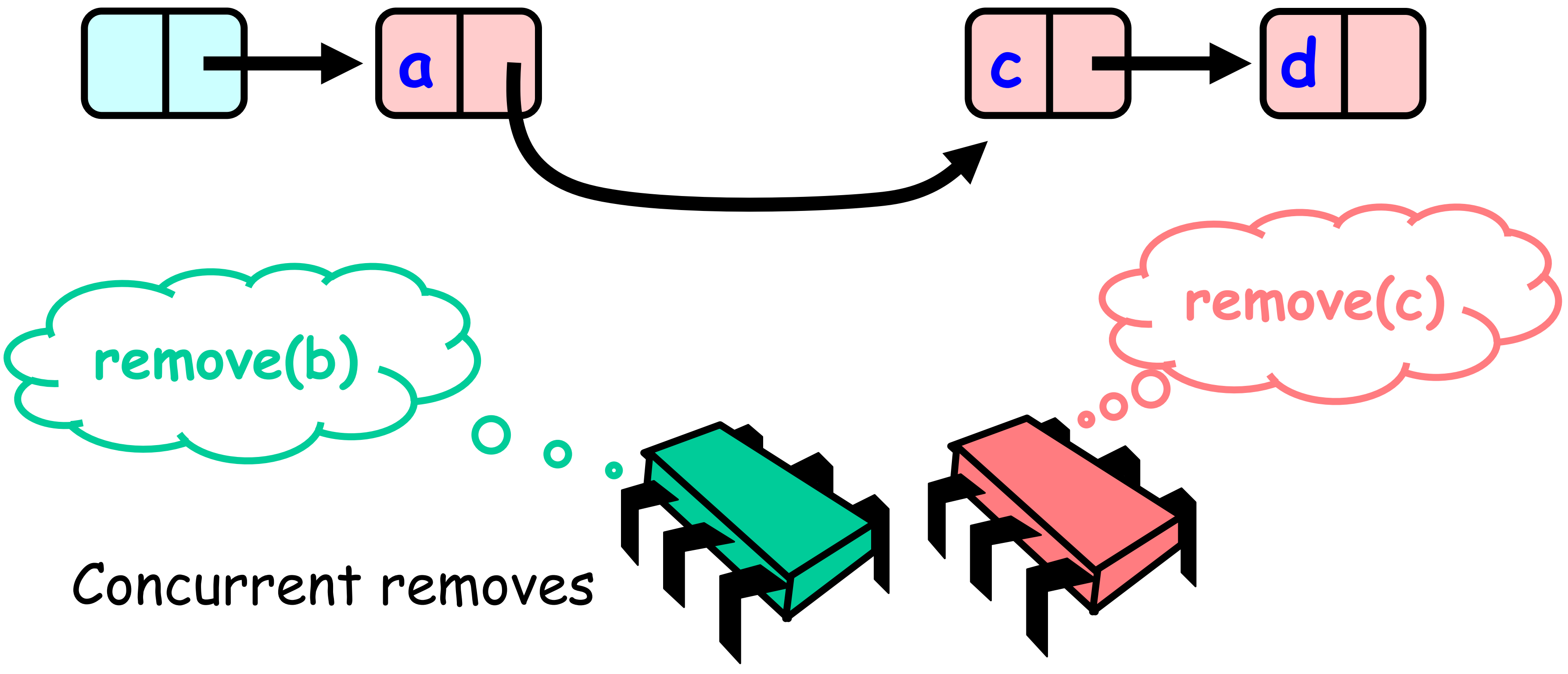
Concurrent Removes

2. Explaining why every step is needed.



2.Fine Grained

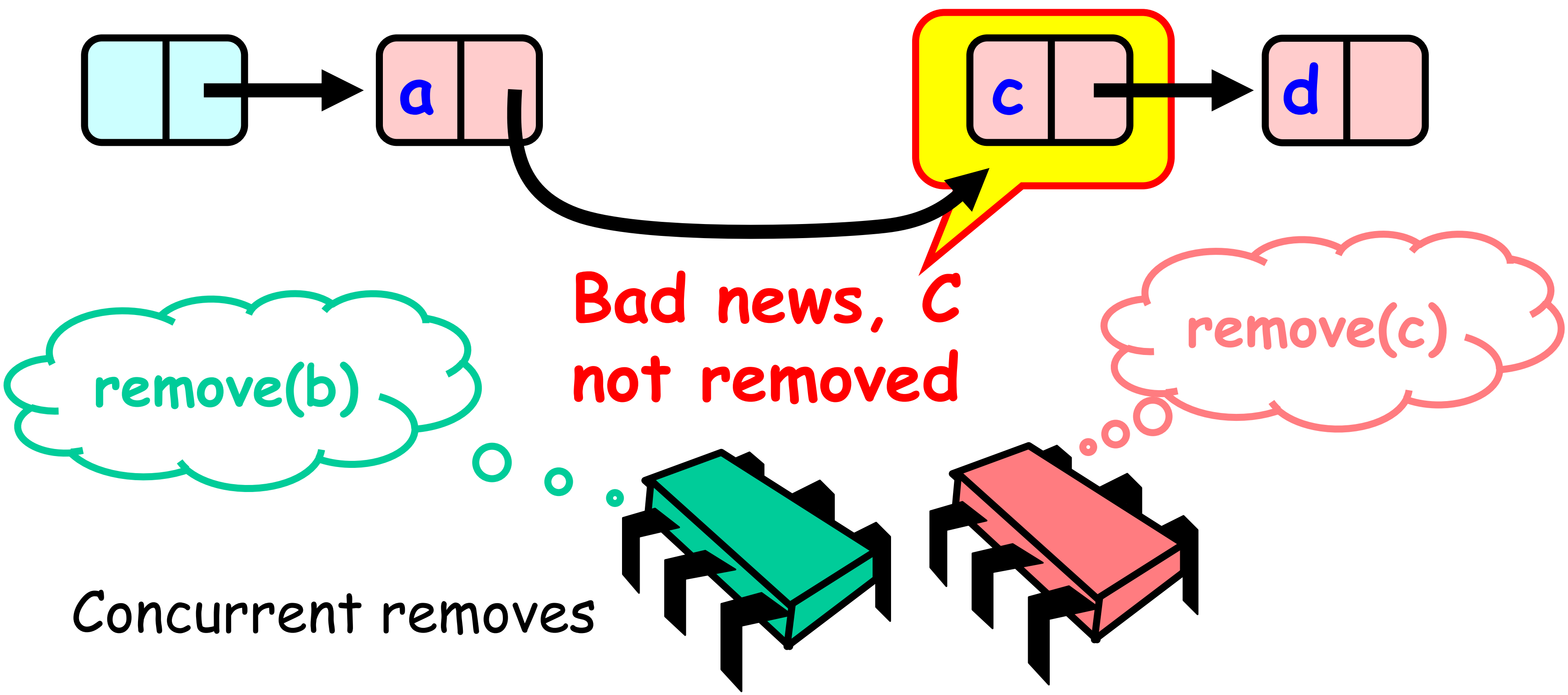
2. Explaining why every step is needed.



Concurrent removes

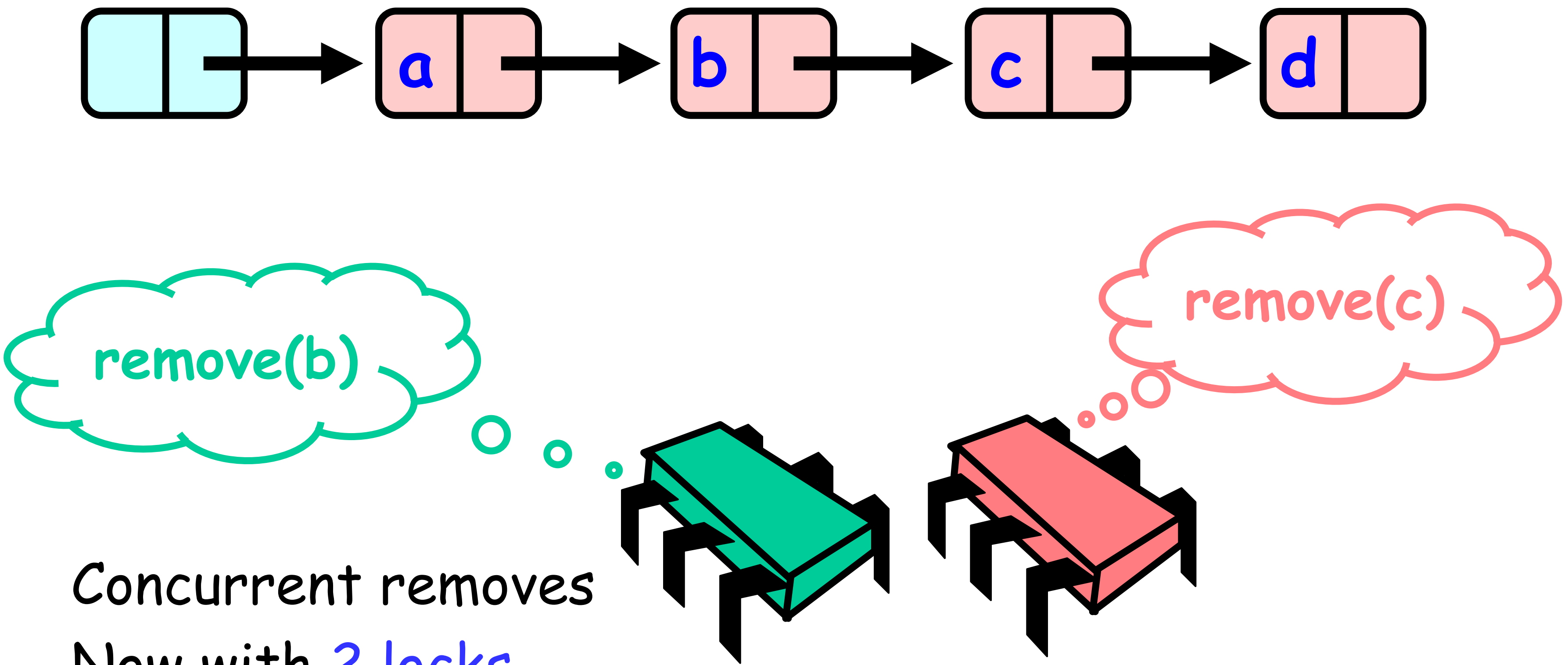
2.Fine Grained

2. Explaining why every step is needed.



2.Fine Grained

2. Explaining why every step is needed.

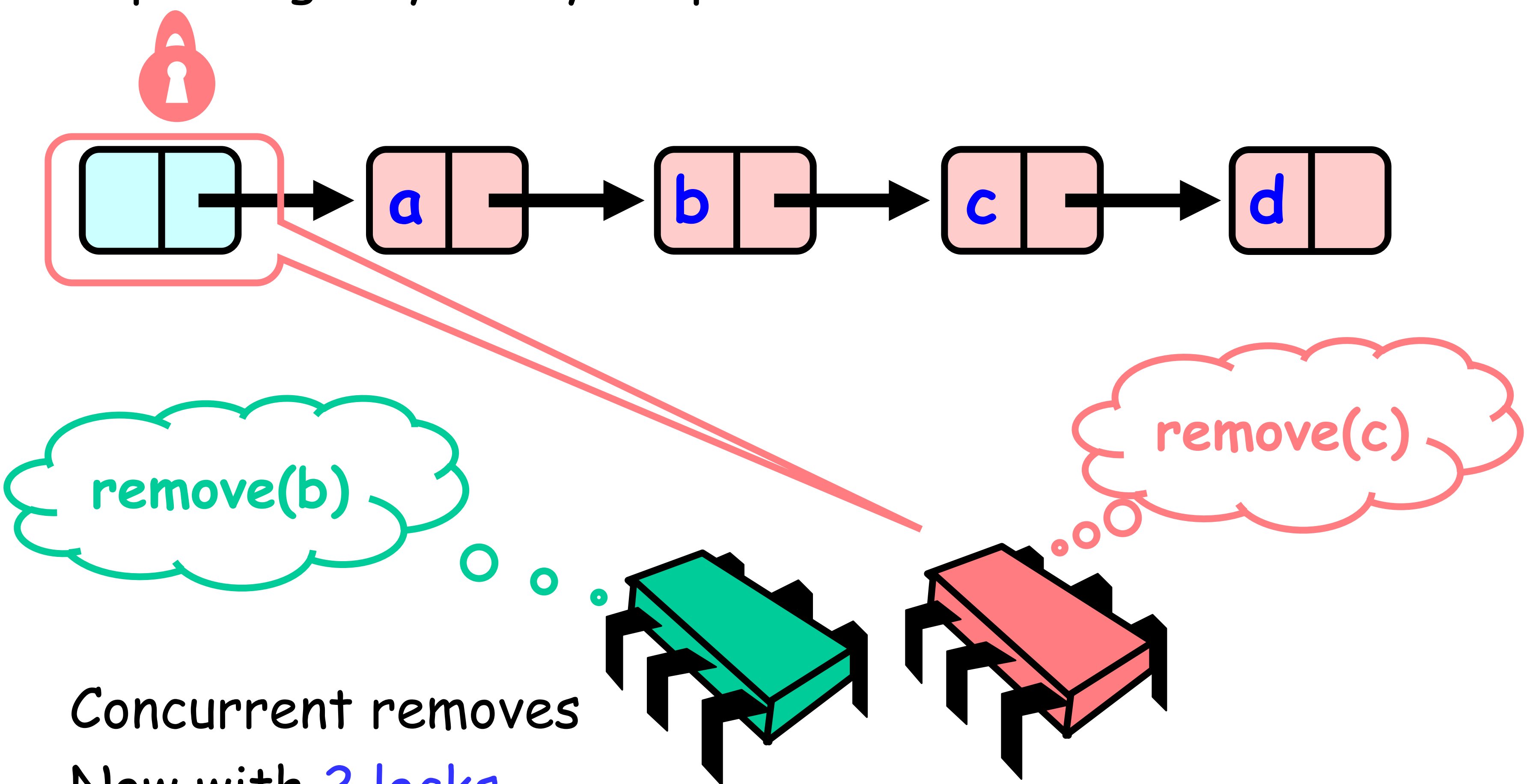


Concurrent removes

Now with 2 locks.

2.Fine Grained

2. Explaining why every step is needed.

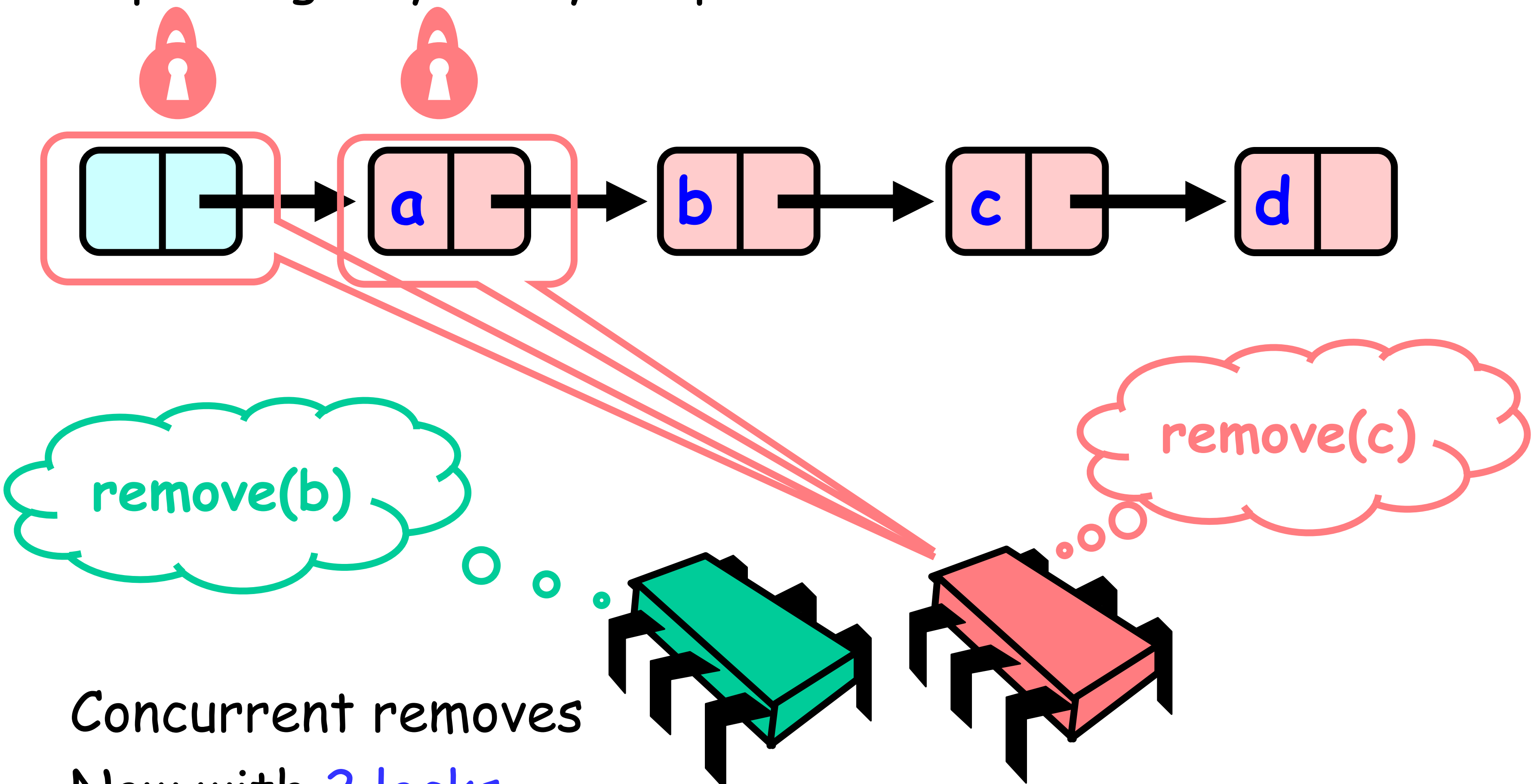


Concurrent removes

Now with 2 locks.

2.Fine Grained

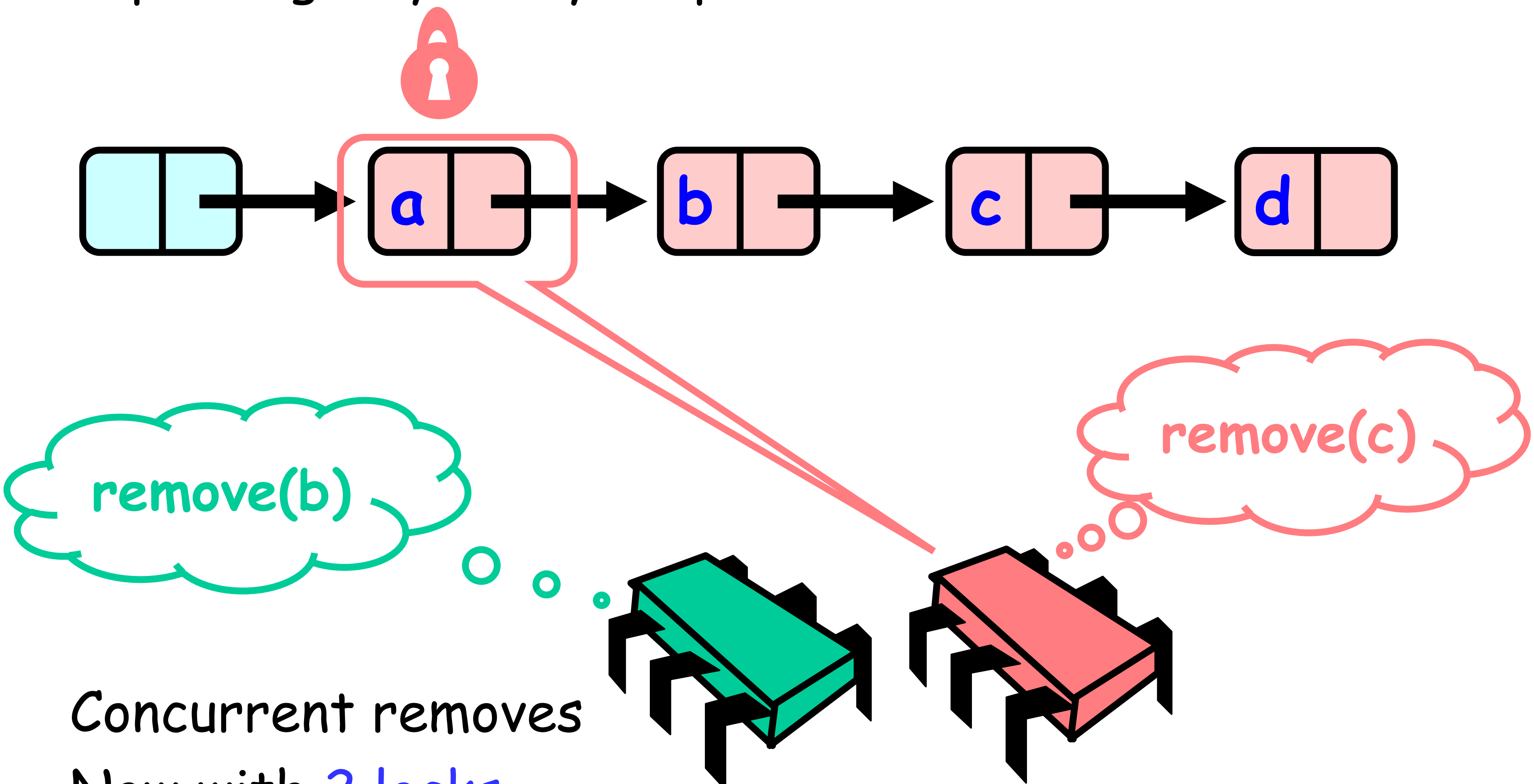
2. Explaining why every step is needed.



Concurrent removes
Now with 2 locks.

2.Fine Grained

2. Explaining why every step is needed.

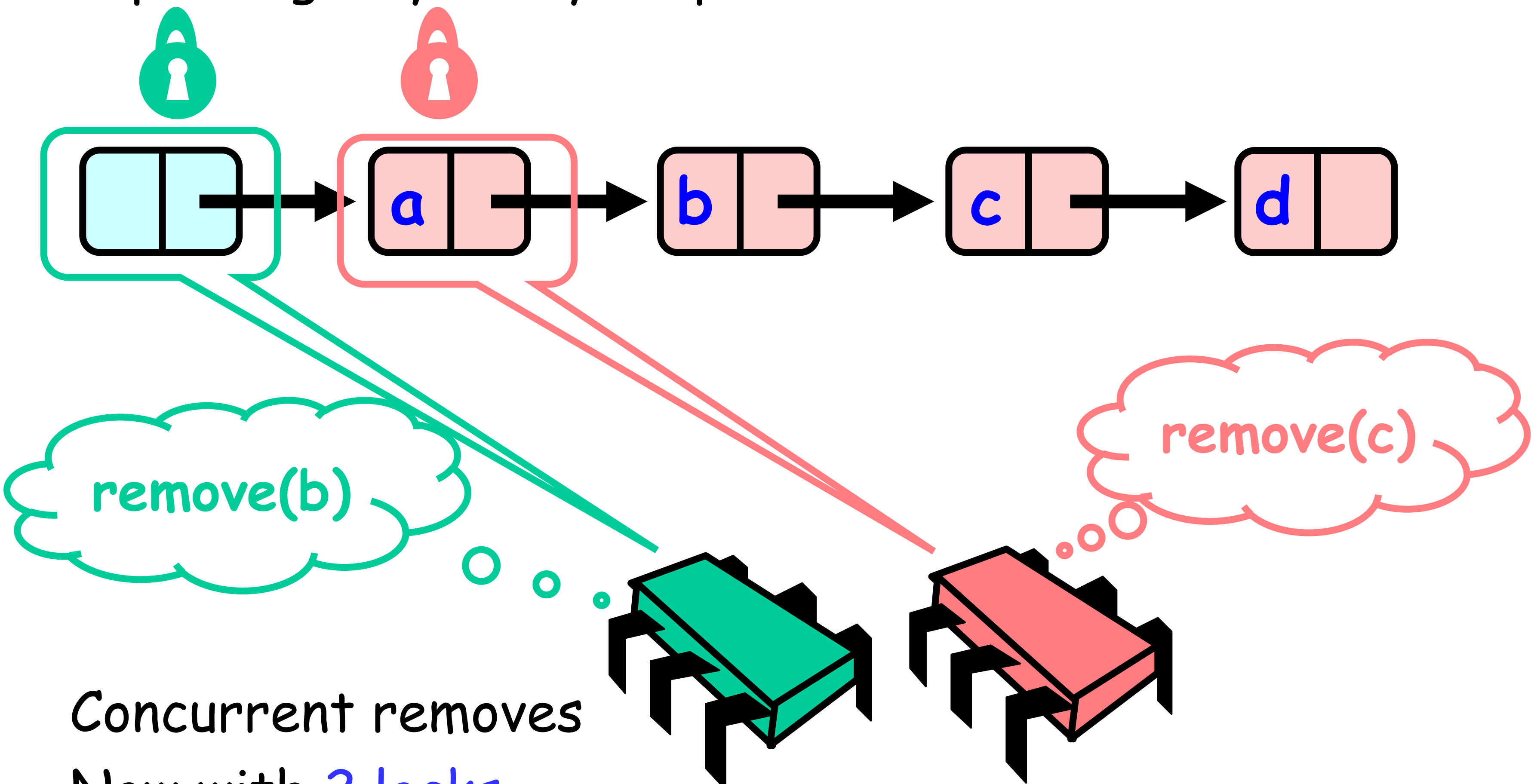


Concurrent removes

Now with 2 locks.

2.Fine Grained

2. Explaining why every step is needed.

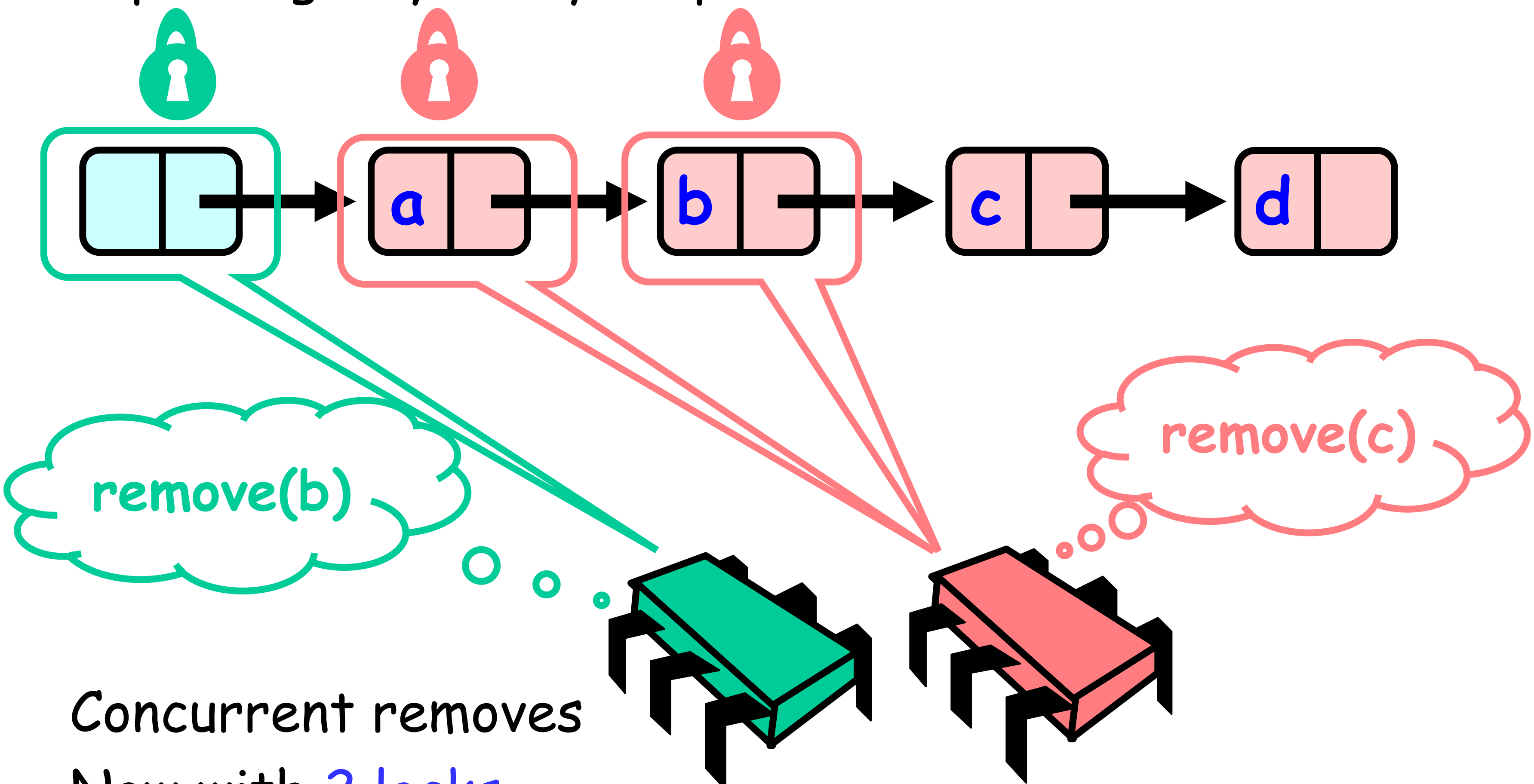


Concurrent removes

Now with 2 locks.

2.Fine Grained

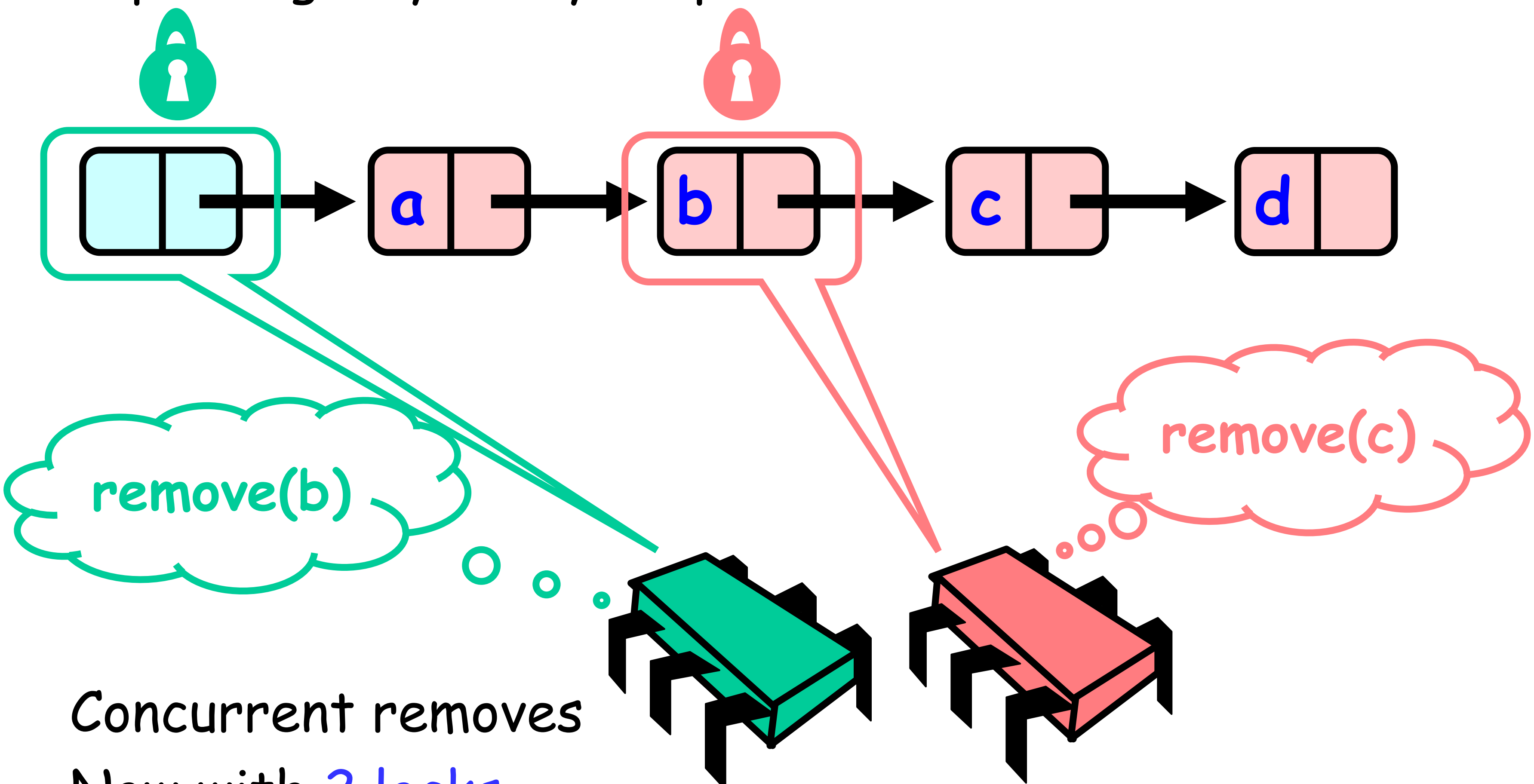
2. Explaining why every step is needed.



Concurrent removes
Now with 2 locks.

2.Fine Grained

2. Explaining why every step is needed.

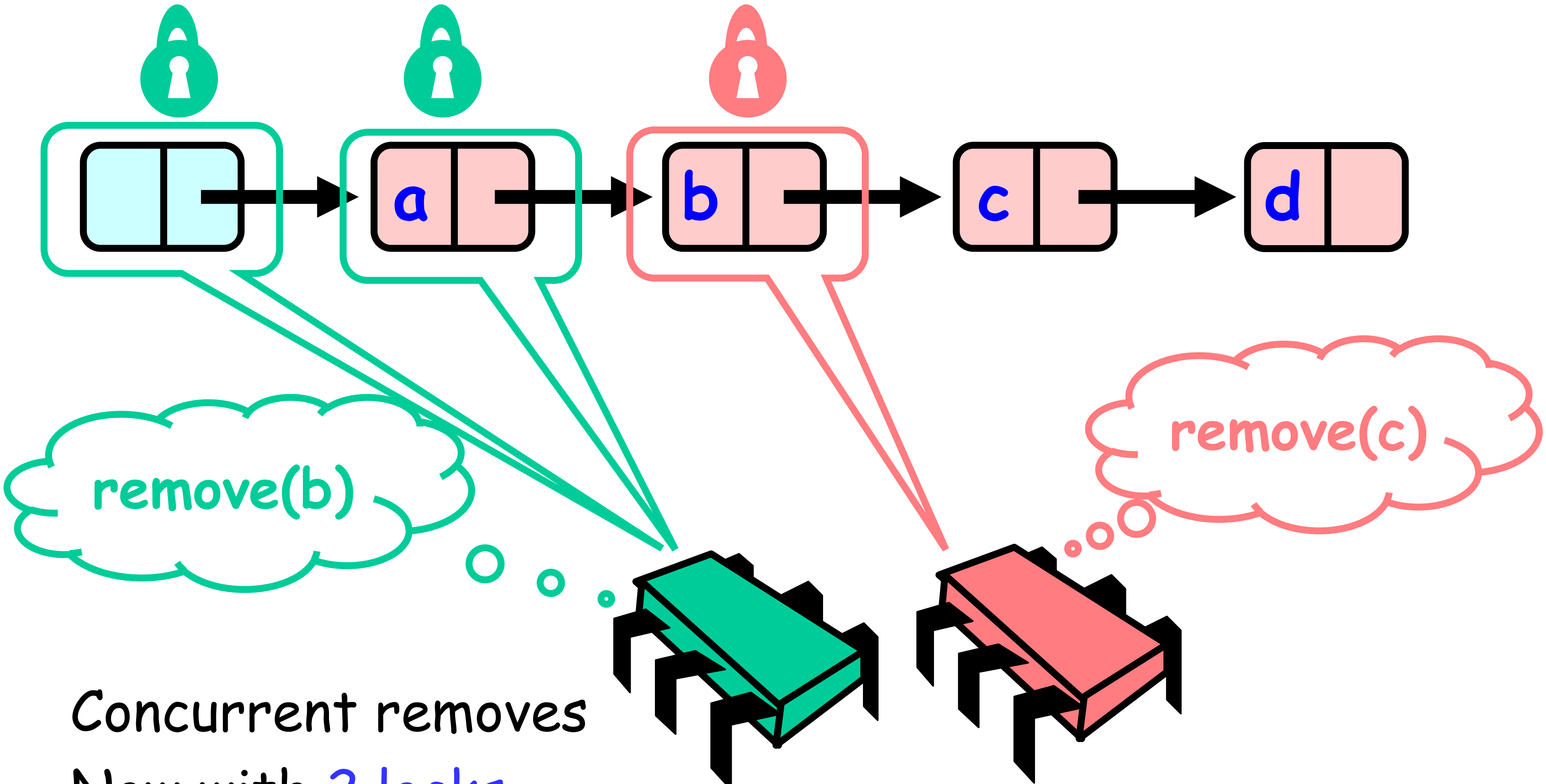


Concurrent removes

Now with 2 locks.

2.Fine Grained

2. Explaining why every step is needed.

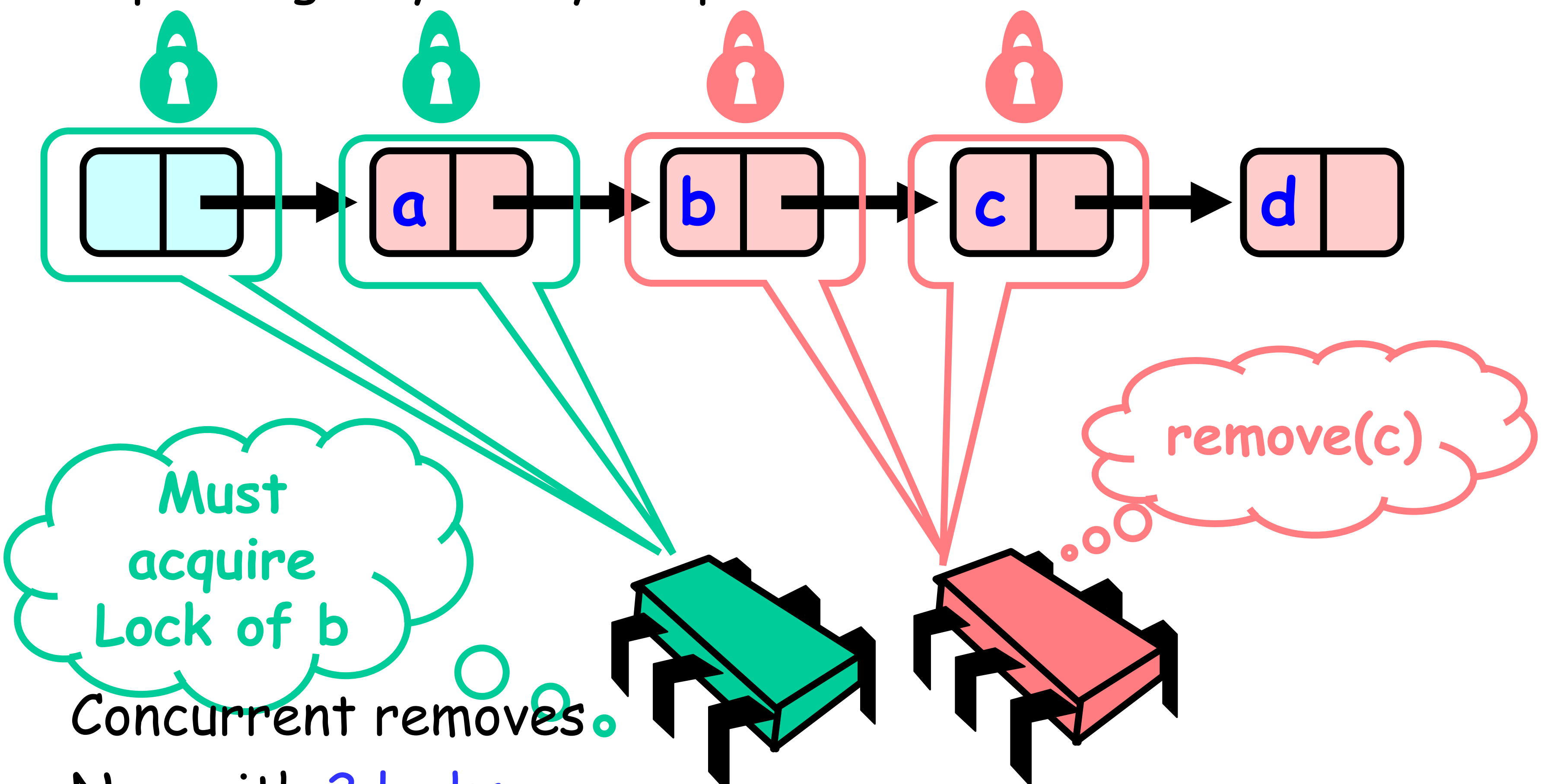


Concurrent removes

Now with 2 locks.

2.Fine Grained

2. Explaining why every step is needed.

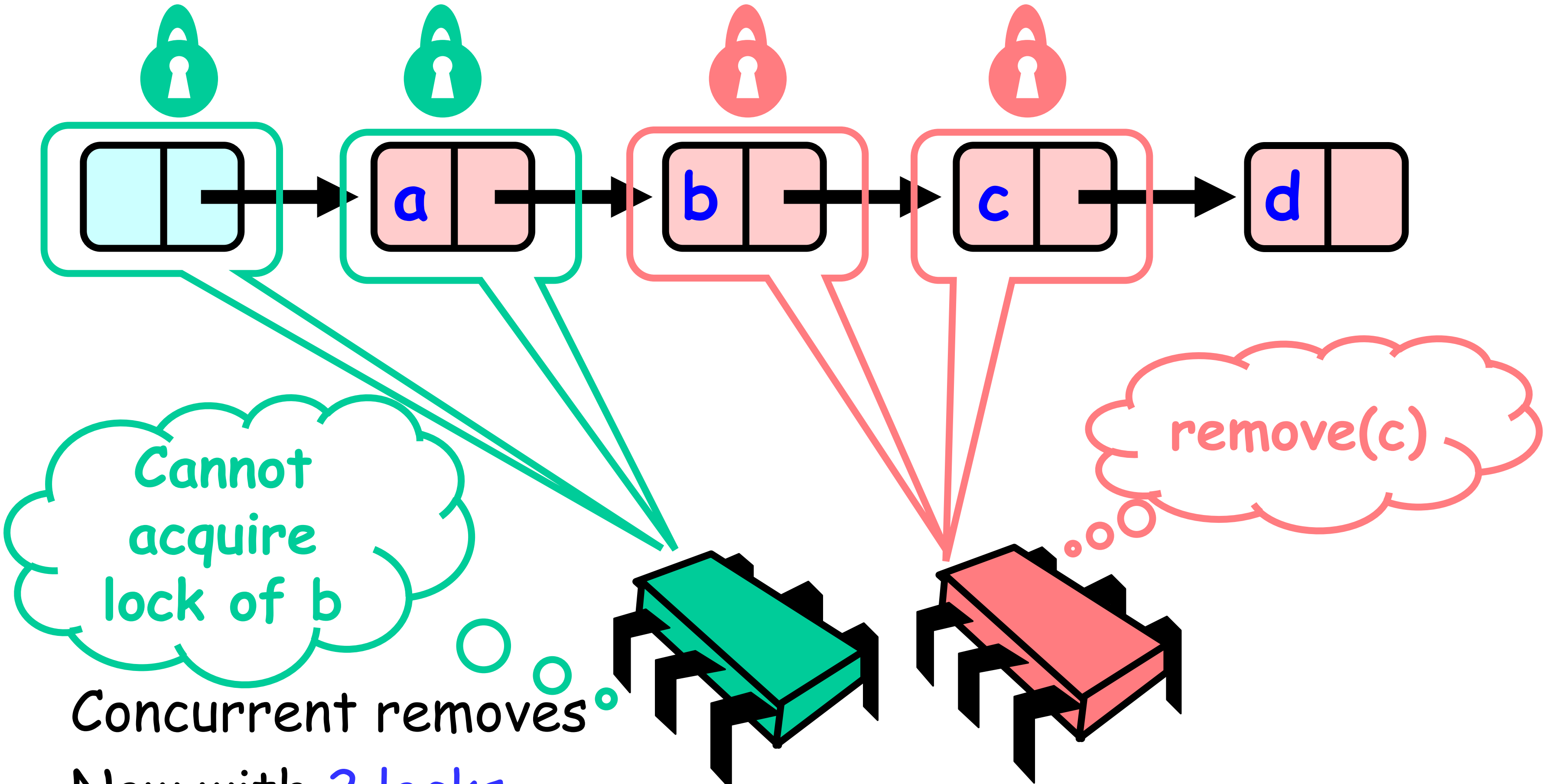


Concurrent removes

Now with 2 locks.

2.Fine Grained

2. Explaining why every step is needed.

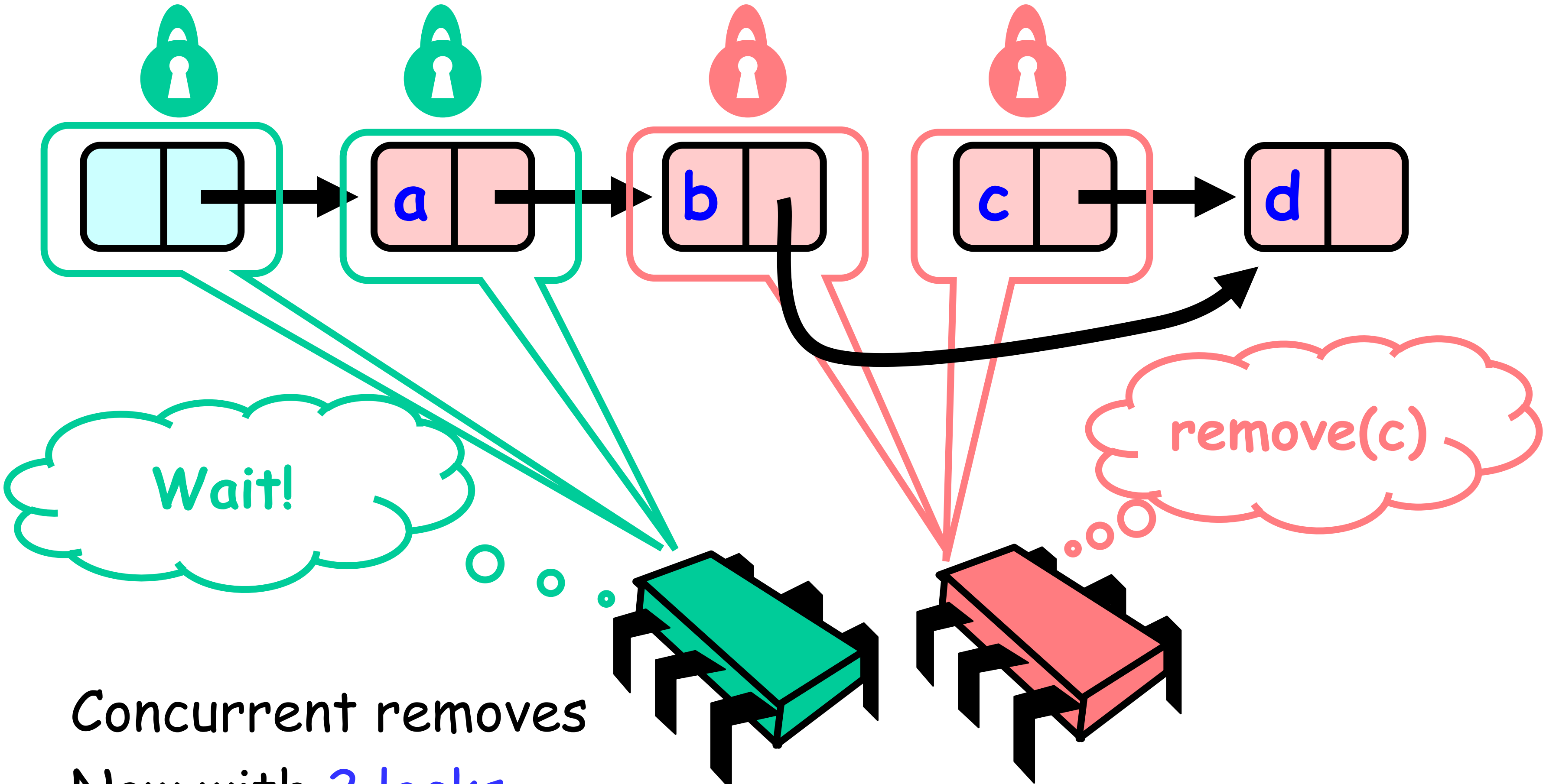


Concurrent removes

Now with 2 locks.

2.Fine Grained

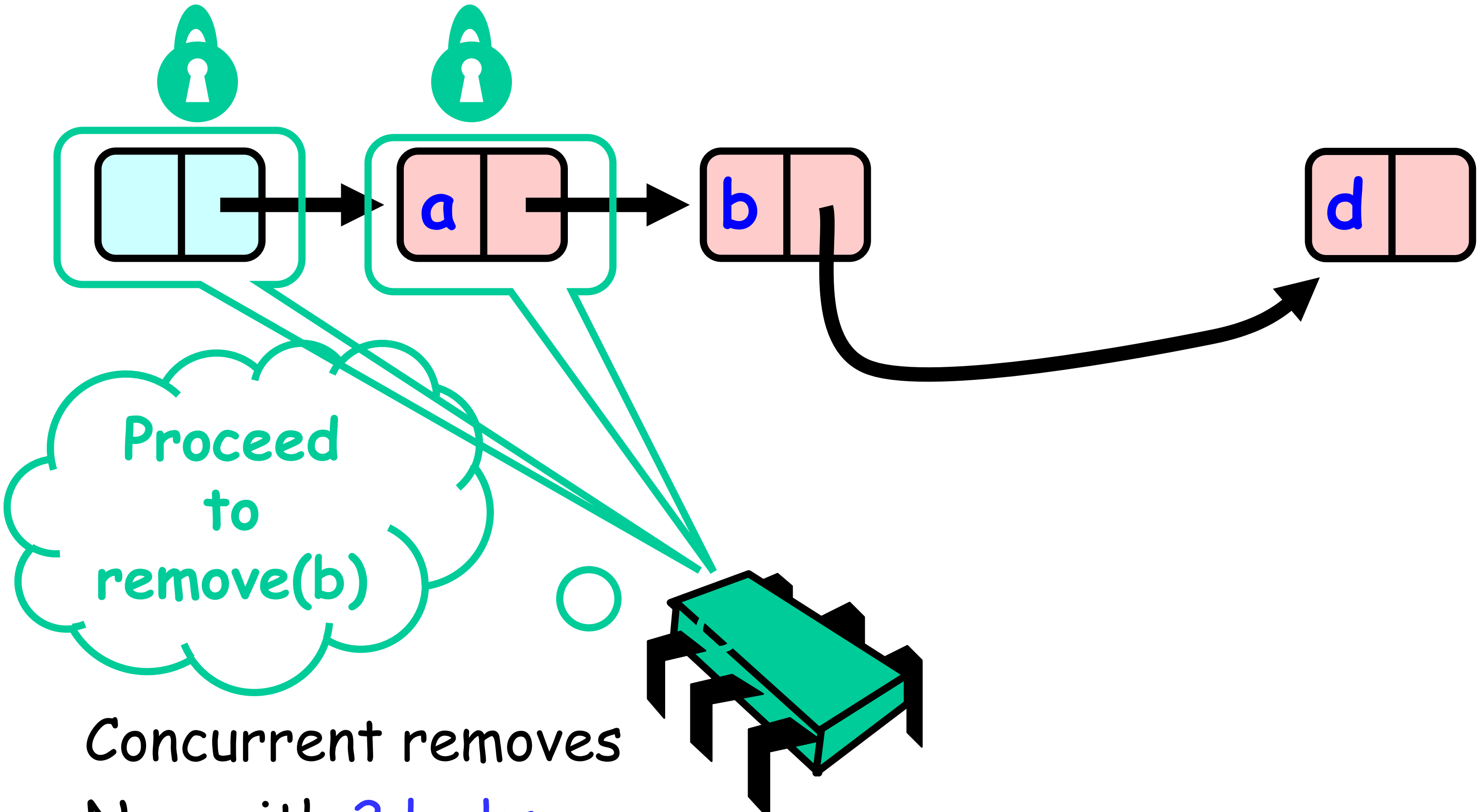
2. Explaining why every step is needed.



Concurrent removes
Now with 2 locks.

2.Fine Grained

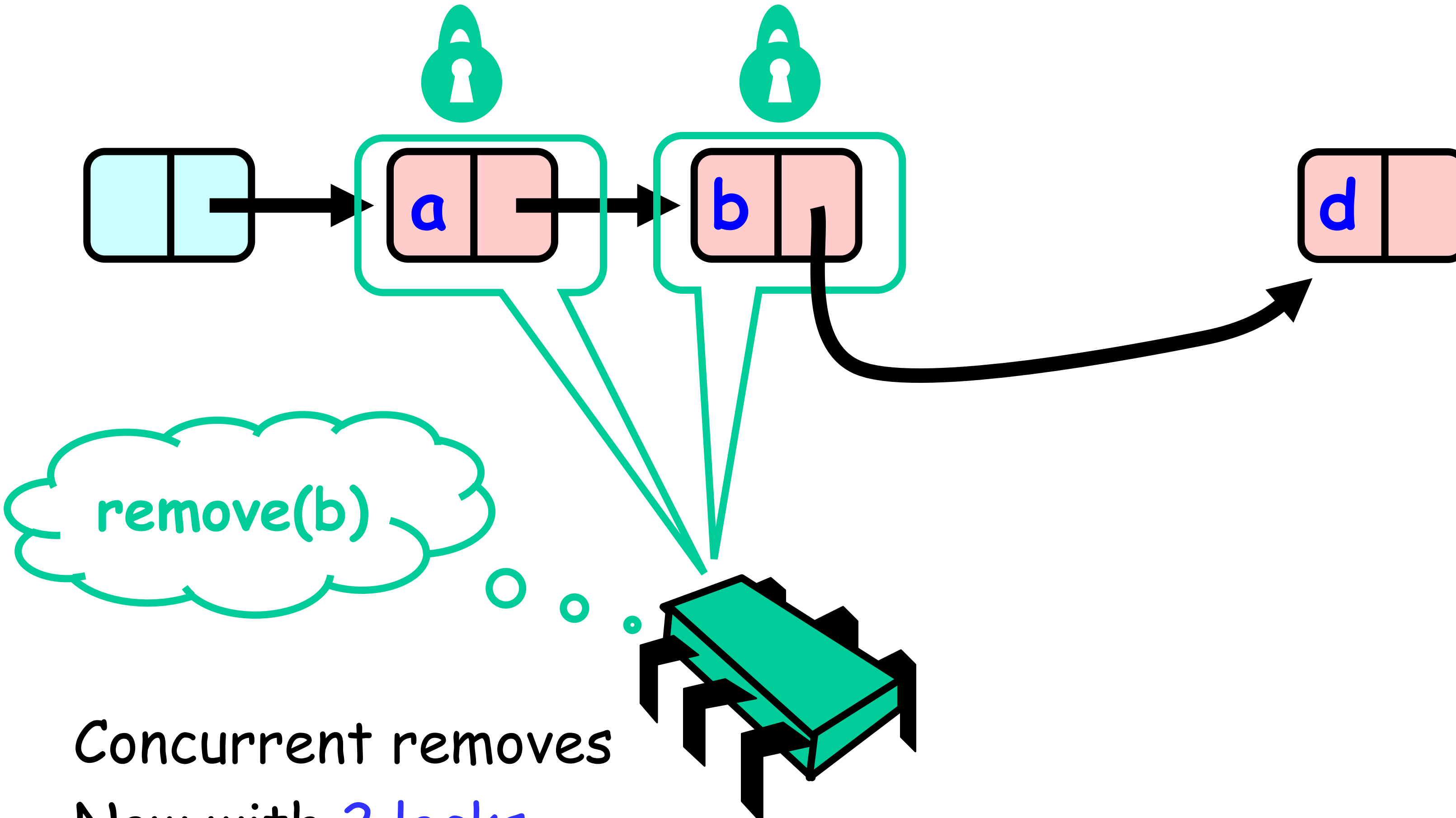
2. Explaining why every step is needed.



Concurrent removes
Now with 2 locks.

2.Fine Grained

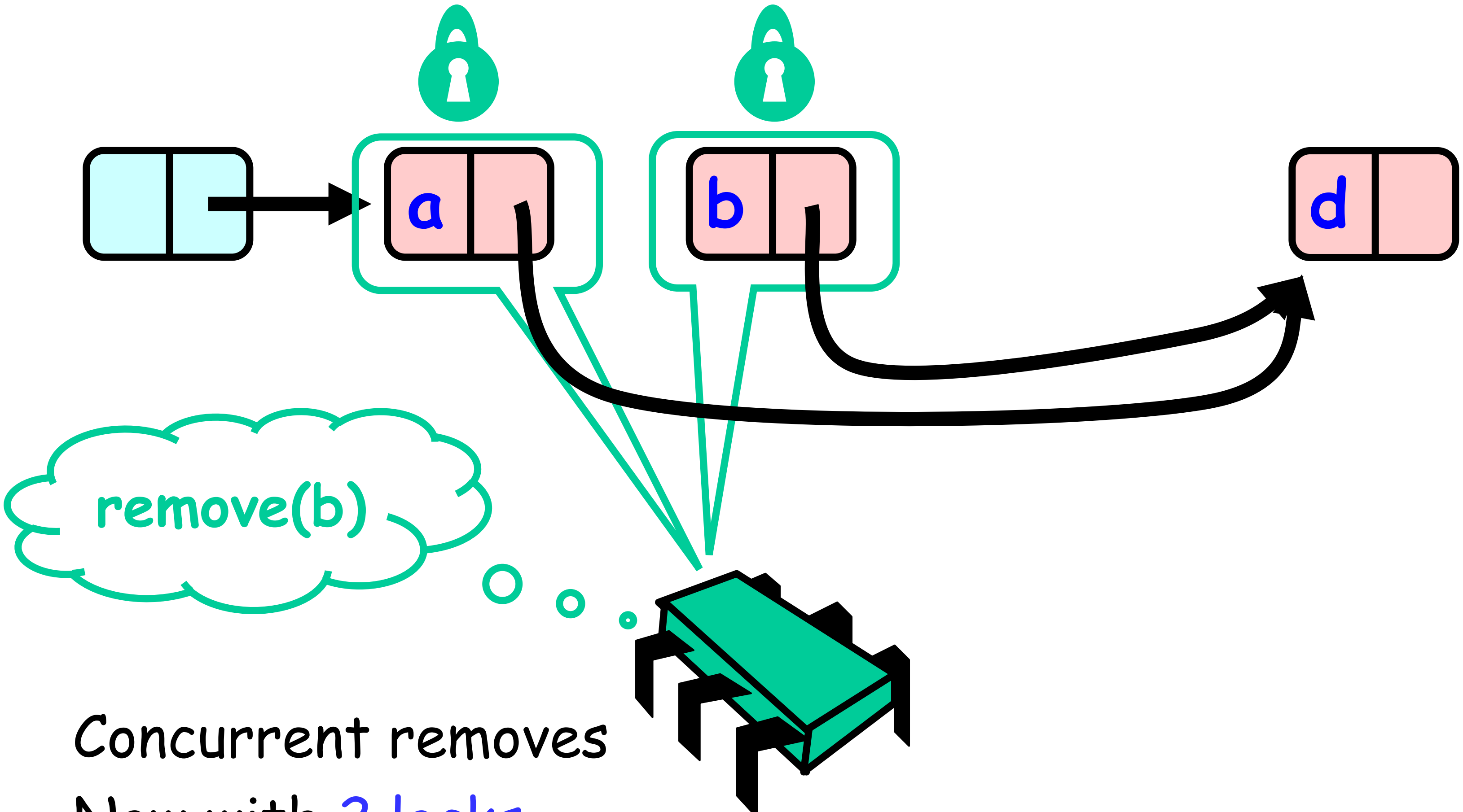
2. Explaining why every step is needed.



Concurrent removes
Now with 2 locks.

2.Fine Grained

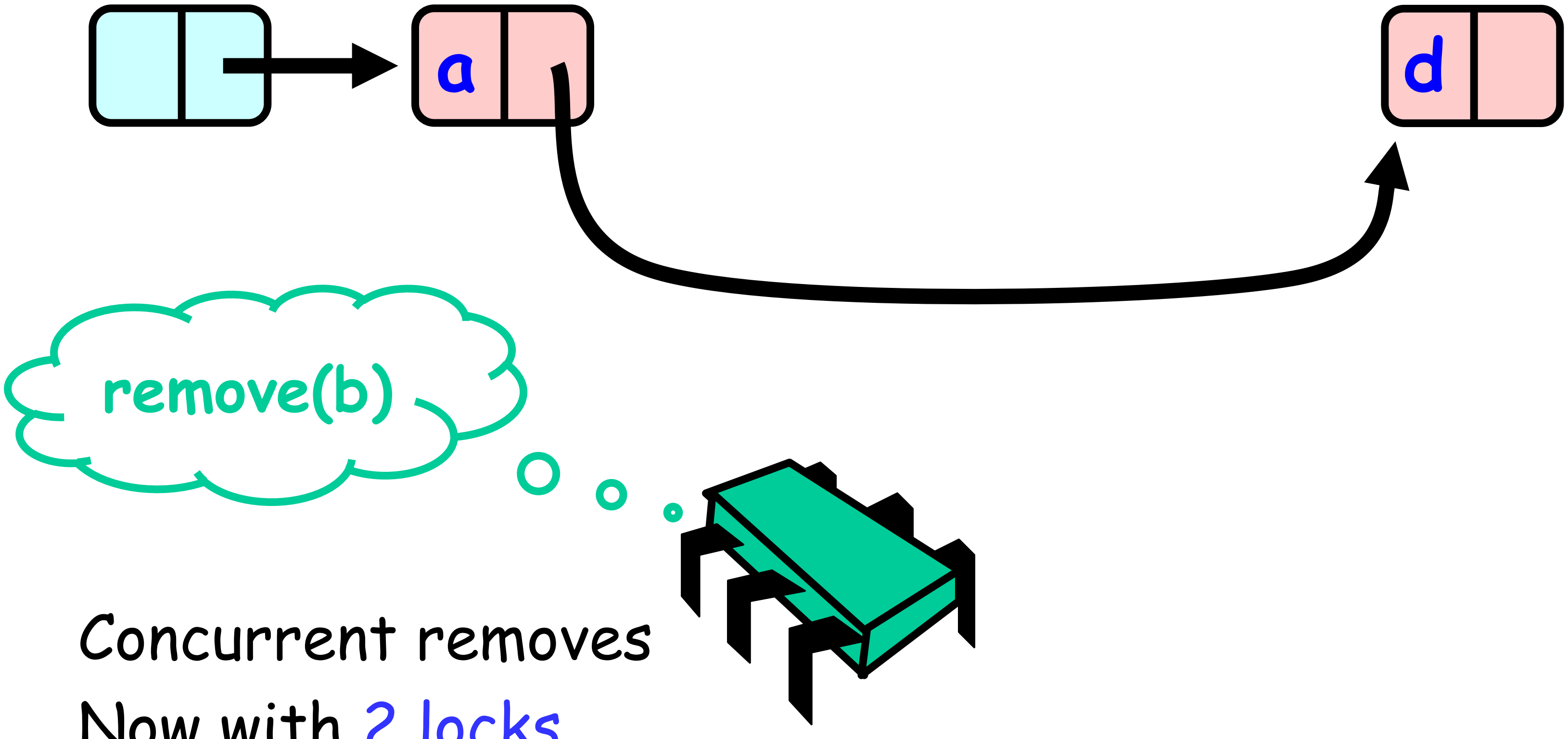
2. Explaining why every step is needed.



Concurrent removes
Now with 2 locks.

2.Fine Grained

2. Explaining why every step is needed.



Concurrent removes

Now with 2 locks.

2.Fine Grained

2. Explaining why every step is needed.

- Conclusion:

- Now that we hold 2 locks for Remove / Add / Contains. If a node is locked :
  - It can't be removed and so does the next node in the list.
  - No new node can be added before it and after it.

Remove method

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

Remove method

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

Key used to order node

Remove method

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

Predecessor and current nodes

Remove method

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

Make sure locks released

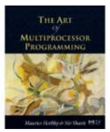
Remove method

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

Everything else

Remove method

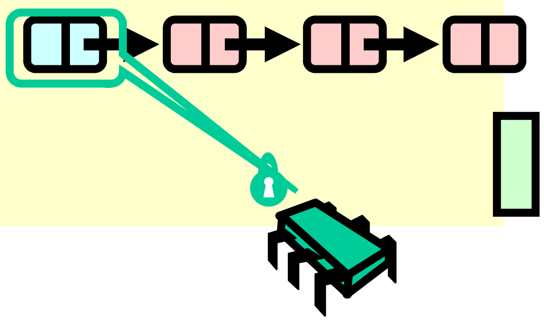
```
try {
  pred = this.head;
  pred.lock();
  curr = pred.next;
  curr.lock();
  …
} finally { … }
```

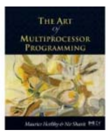


Remove method

lock pred == head

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

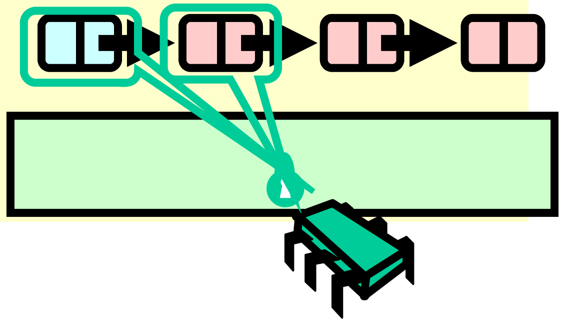




Remove method

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

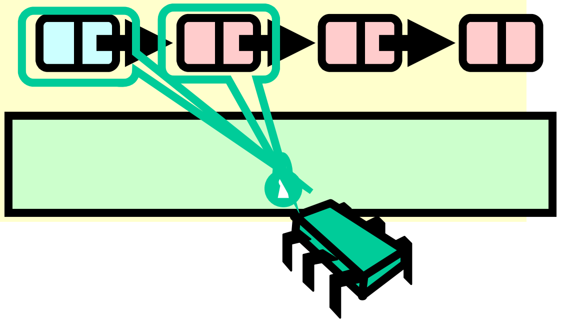
Lock current



Remove method

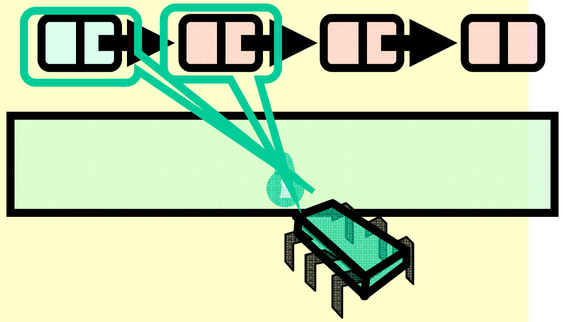
```
try {
 pred = this.head;
 pred.lock();
 curr = pred.next;
 curr.lock();
 ...
} finally { … }
```

Traversing list



Remove: searching

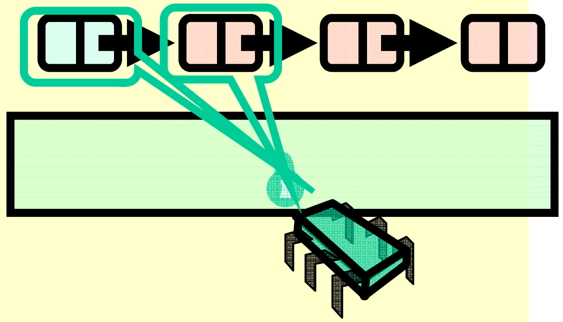
```
while (curr.key <= key) {
  if (item == curr.item) {
   pred.next = curr.next;
   return true;
  }
  pred.unlock();
  pred = curr;
  curr = curr.next;
  curr.lock();
 }
 return false;
```



Remove: searching

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

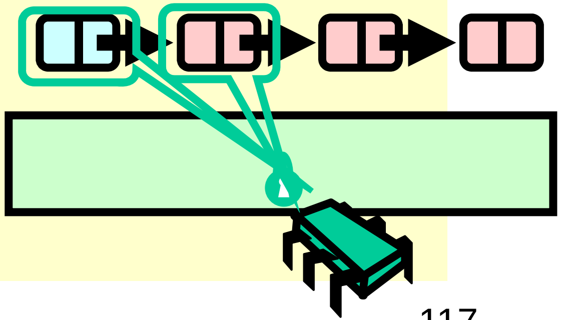
Search key range

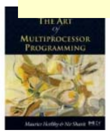


Remove: searching

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

At start of each loop: curr and pred locked

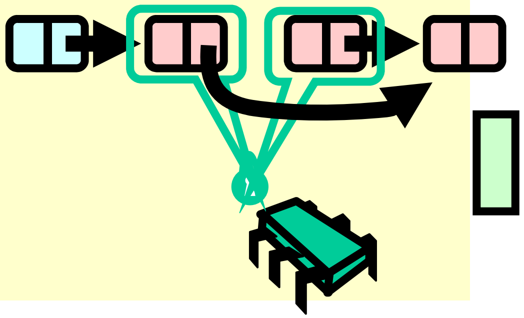




Remove: searching

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

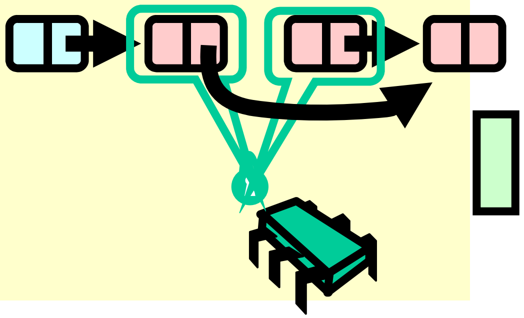
If item found, remove node

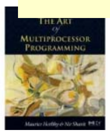


Remove: searching

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

If node found, remove it

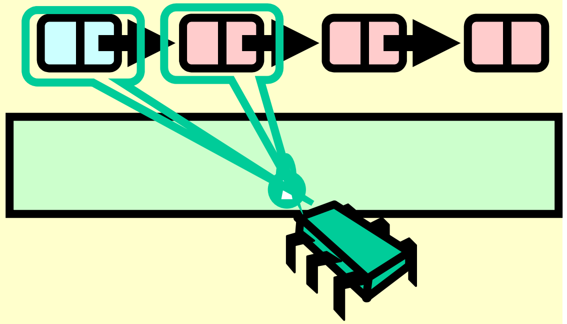




Remove: searching

Unlock predecessor

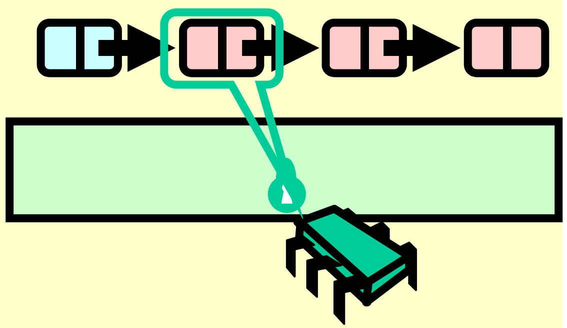
```
while (curr.key <= key) {
  if (item == curr.item) {
   pred.next = curr.next;
   return true;
  }
  pred.unlock();
  pred = curr;
  curr = curr.next;
  curr.lock();
 }
 return false;
```



Remove: searching

Only one node locked!

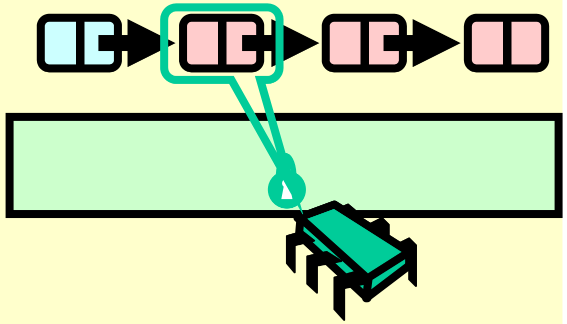
```
while (curr.key <= key) {
  if (item == curr.item) {
   pred.next = curr.next;
   return true;
  }
  pred.unlock();
  pred = curr;
  curr = curr.next;
  curr.lock();
 }
 return false;
```



Remove: searching

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

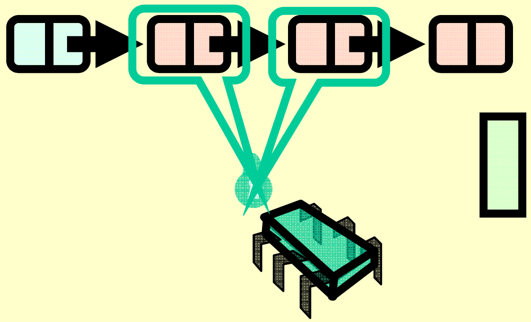
demote current



Remove: searching

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

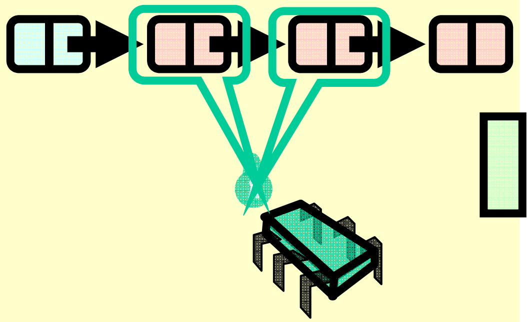
Find and lock new current



Remove: searching

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

Lock invariant restored



Remove: searching

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

Otherwise, not present

2.Fine Grained

3. Code review:

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();
        }
```

Finding the place to add the item:

Continued:

```
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();
    }
  }
```

Adding the item:

2.Fine Grained

3. Code review:

Contains:

```
public boolean contains(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();
      }
```

Finding the place to add the item:

Continued:

```
return (curr.key == key);
      } finally {
        curr.unlock();
      }
    } finally {
      pred.unlock();
    }
  }
```

Return true iff found

2.Fine Grained

Proving correctness for Remove(x) function:

- So how do we prove correctness of a method in a concurrent environment?

1. Decide on a Rep-Invariant. Done!
2. Decide on an Abstraction map. Done!
3. Defining the operations:

   Remove(x): If x in the set => x won't be in the set and return true.

   If x isn't in the set => don't change the set and return false.

   Done!

2.Fine Grained

Proving correctness for Remove(x) function:

4. Proving that each function keeps the Rep-Invariant:
   1. Tail reachable from head.
   2. Sorted.
   3. No duplicates.
   1. The newly created empty list obviously keeps the Rep-invariant.
   2. Easy to see from the code that for each function if the Rep-invariant was kept before the call it will still hold after it.

   Done!

2.Fine Grained

Proving correctness for Remove(x) function:

5. Split the function to all possible run time outcomes.

In our case:

1. Successful remove. (x was in the list)
2. Failed remove. (x wasn't in the list)

Done!

6. Proving for each possibility.

We will start with a successful remove. (failed remove is not much different)

2.Fine Grained

Proving correctness for Remove(x) function:

successful remove.

6. Deciding on a linearization point for a successful remove.

   Reminder: Linearization point – a point in time that we can say the function has happened in a running execution.

We will set the Linearization point to after the second lock was acquired.

Done!

Why remove() is linearizable

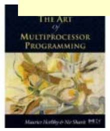
```
while (curr.key <= key) {
  if (item == curr.item) {
   pred.next = curr.next;
   return true;
  }
  pred.unlock();
  pred = curr;
  curr = curr.next;
  curr.lock();
 }
 return false;
```

- **pred** reachable from **head**
- **curr** is **pred.next**
- So **curr.item** is in the set

Why remove() is linearizable

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

Linearization point if item is present



Why remove() is linearizable

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

Node locked, so no other thread can remove it ….

Why remove() is linearizable

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

Item not present

Why remove() is linearizable

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

- **pred** reachable from **head**
- **curr** is **pred.next**
- **pred.key <** key
- key < **curr.key**

Why remove() is linearizable

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

Linearization point

2.Fine Grained

Proving correctness for Remove(x) function:

successful remove.

7. Now that the linearization point is set we need to prove that:
   7.1. Before the linearization point the set contained x.
   7.2. After it the set won't contain x.

2.Fine Grained

Proving correctness for Remove(x) function:

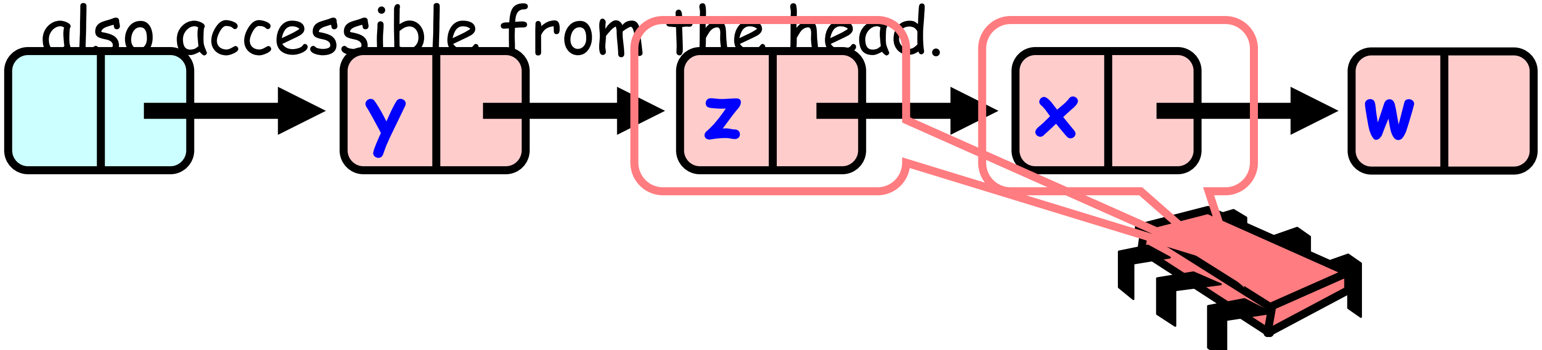
successful remove.

7.1. Before the linearization point the set contained x.

1. Since we proved the Rep-Invariant holds then pred=z is accessible from the head.

2. Since z,x are locked. No other concurrent call can remove them.

3. Since curr=x is pointed to by pred then x is also accessible from the head.

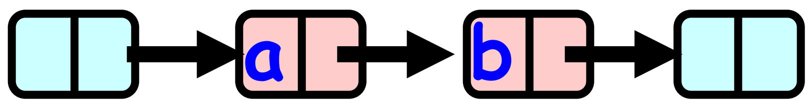


2.Fine Grained

Proving correctness for Remove(x) function:

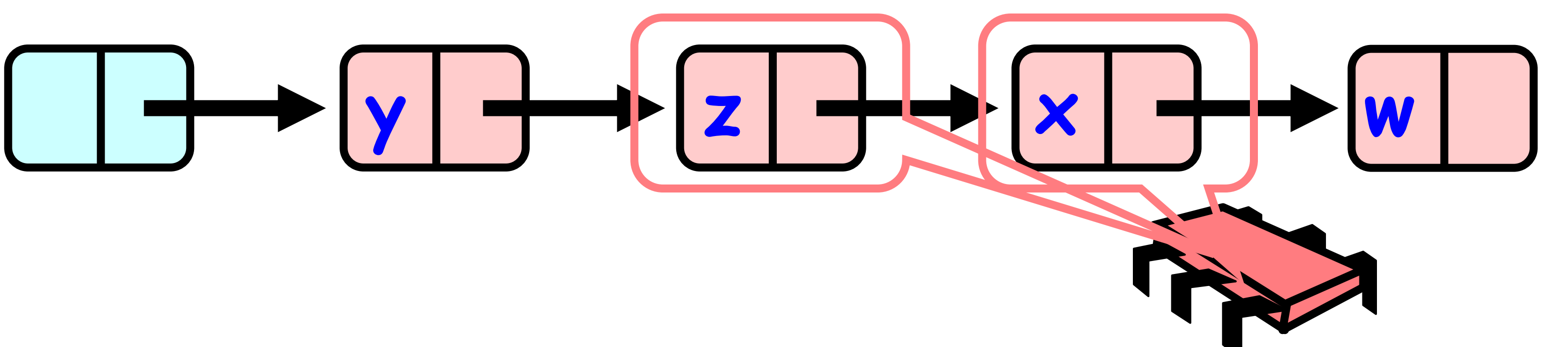
successful remove.

7.1. Before the linearization point the set contained x. Now by the Abstraction map definition:

- S() = {a,b}

since x is reachable from the head => x is in the set!

Done!



2.Fine Grained

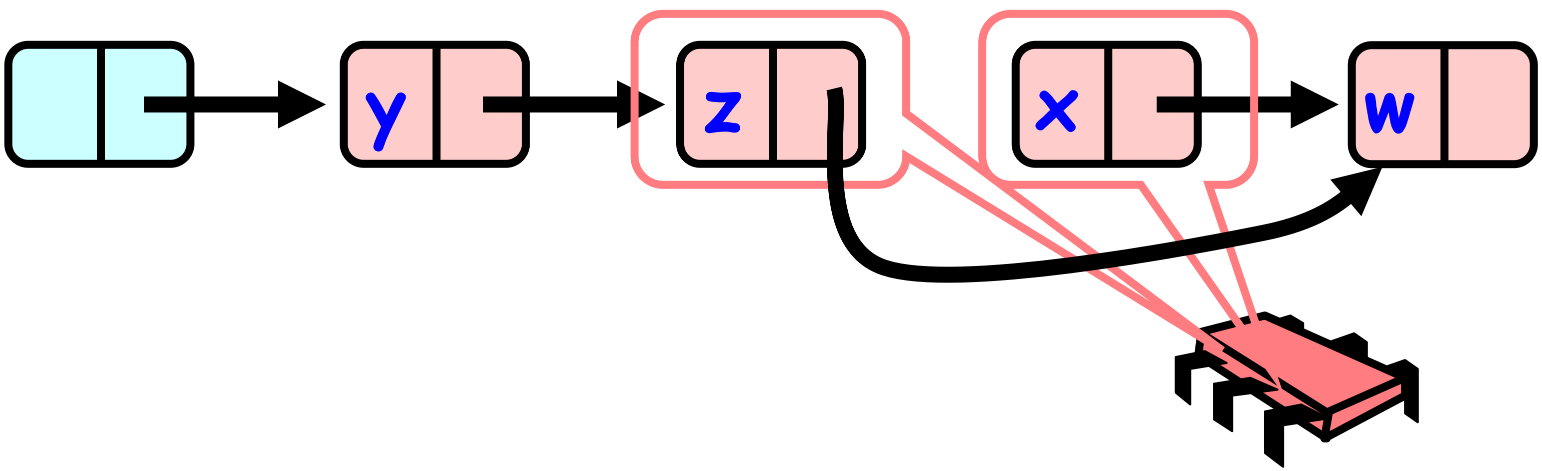
Proving correctness for Remove(x) function:

successful remove.

7.1. After it the set won't contain x.

1. after the linearization point: pred.next = curr.next;

Curr=x won't be pointed to by pred=z and so won't be accessible from head.



2.Fine Grained

Proving correctness for Remove(x) function:

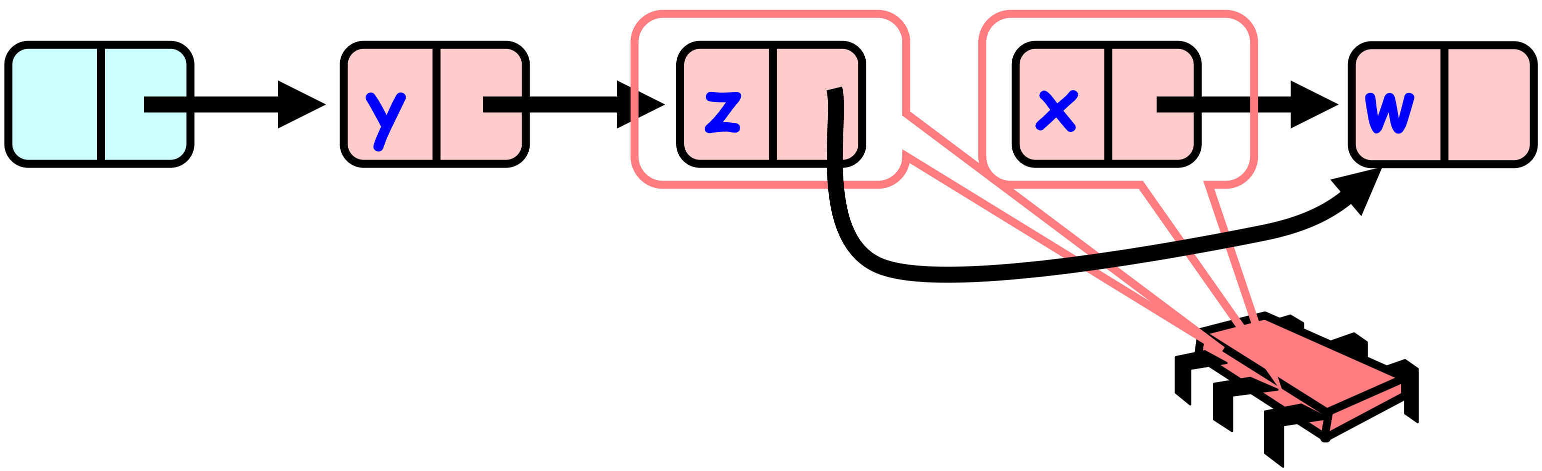
successful remove.

7.1. After it the set won't contain x.

2. Now by the Abstraction map definition:

since x is not reachable from the head => x is not in the set!

Done!



2.Fine Grained

Proving correctness for Remove(x) function:

- In conclusion:
  - For every possible run time execution for Remove(x) we found a linearization point that holds the remove function specification in the set using the Abstraction map while holding the Rep-Invariant.

Done!

2.Fine Grained

4. Methods properties:

- Assuming fair scheduler. If the Lock implementation is Starvation free:

  Every thread will eventually get the lock and since all methods move in the same direction in the list there won't be deadlock and eventually the call to the function will return.

- So our implementation of Insert, Remove and Contains is Starvation-free.

2.Fine Grained

5. Advantages / Disadvantages:

Advantages:

- Better than coarse-grained lock

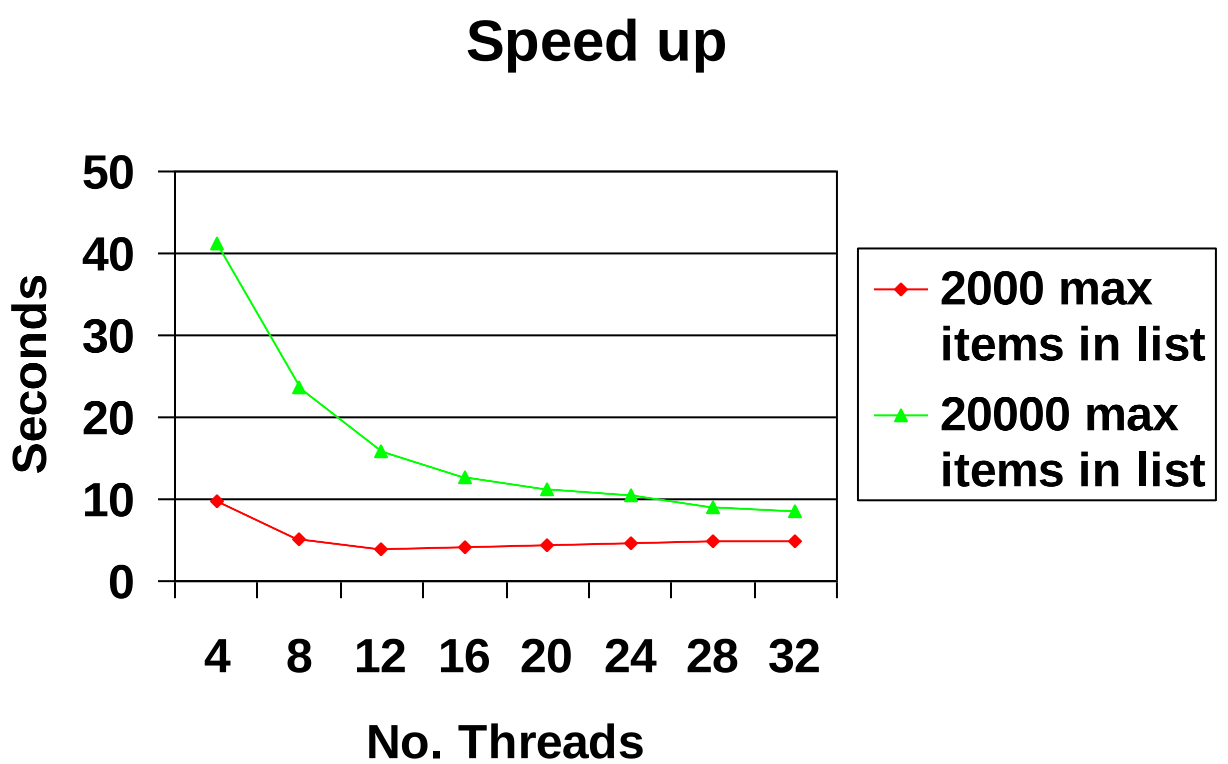
  Threads can traverse in parallel.

Disadvantages:

- Long chain of acquire/release.
- Inefficient.

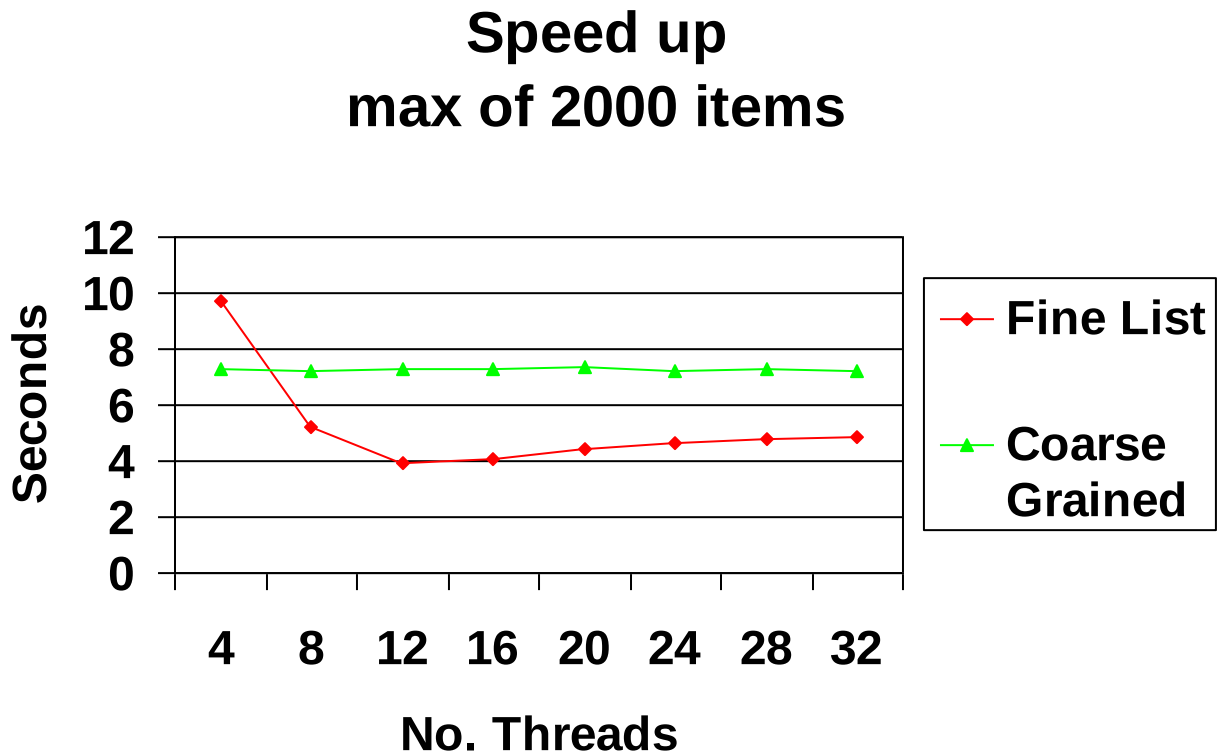
2.Fine Grained

6. Running times:



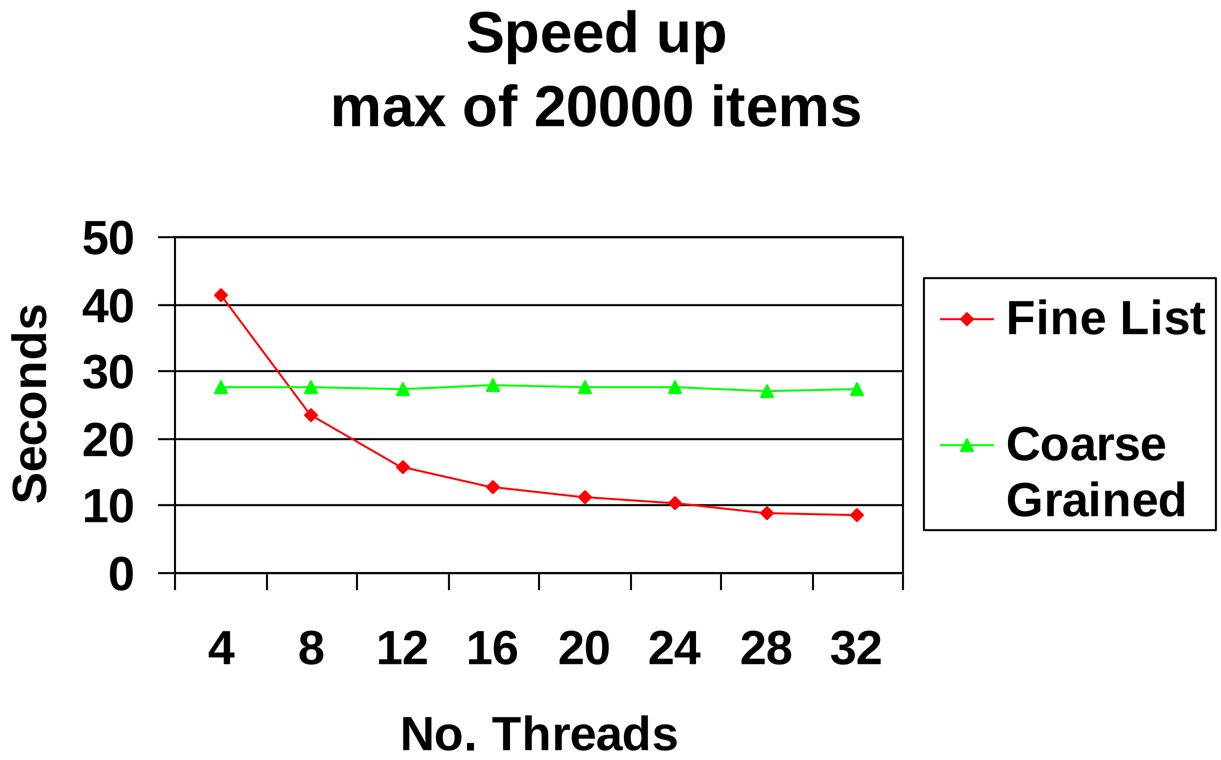
2.Fine Grained

6. Running times:



2.Fine Grained

6. Running times:



3. Optimistic

1. Describing the algorithm:

   Add(x) / Remove (x) / Contains(x):

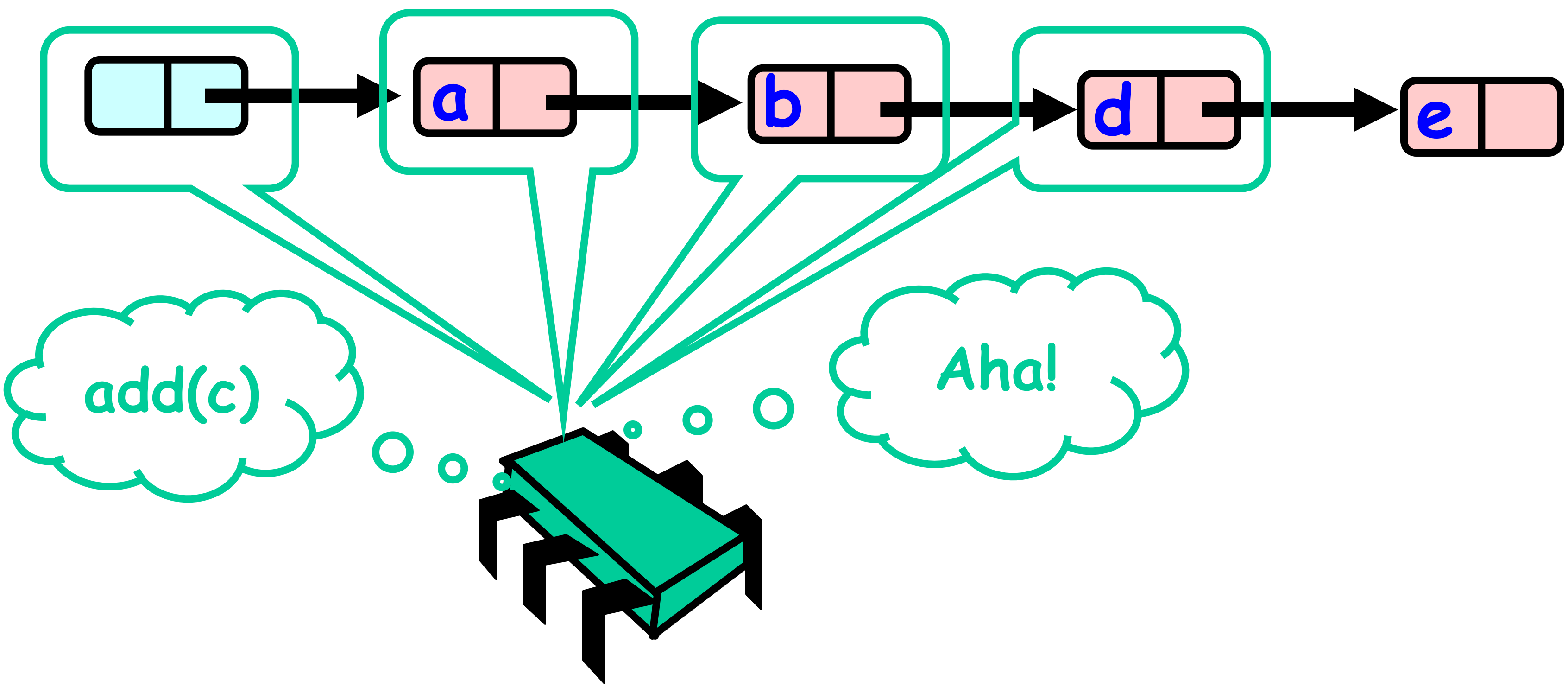
   1. Find nodes without locking
   2. Lock nodes
   3. Check that everything is OK = Validation.
      3.1 Check that pred is still reachable from head.
      3.2 Check that pred still points to curr.
   4. If validation passed => do the operation.

3. Optimistic

1. Describing the algorithm:

- Example of add(c):

Finding without locking

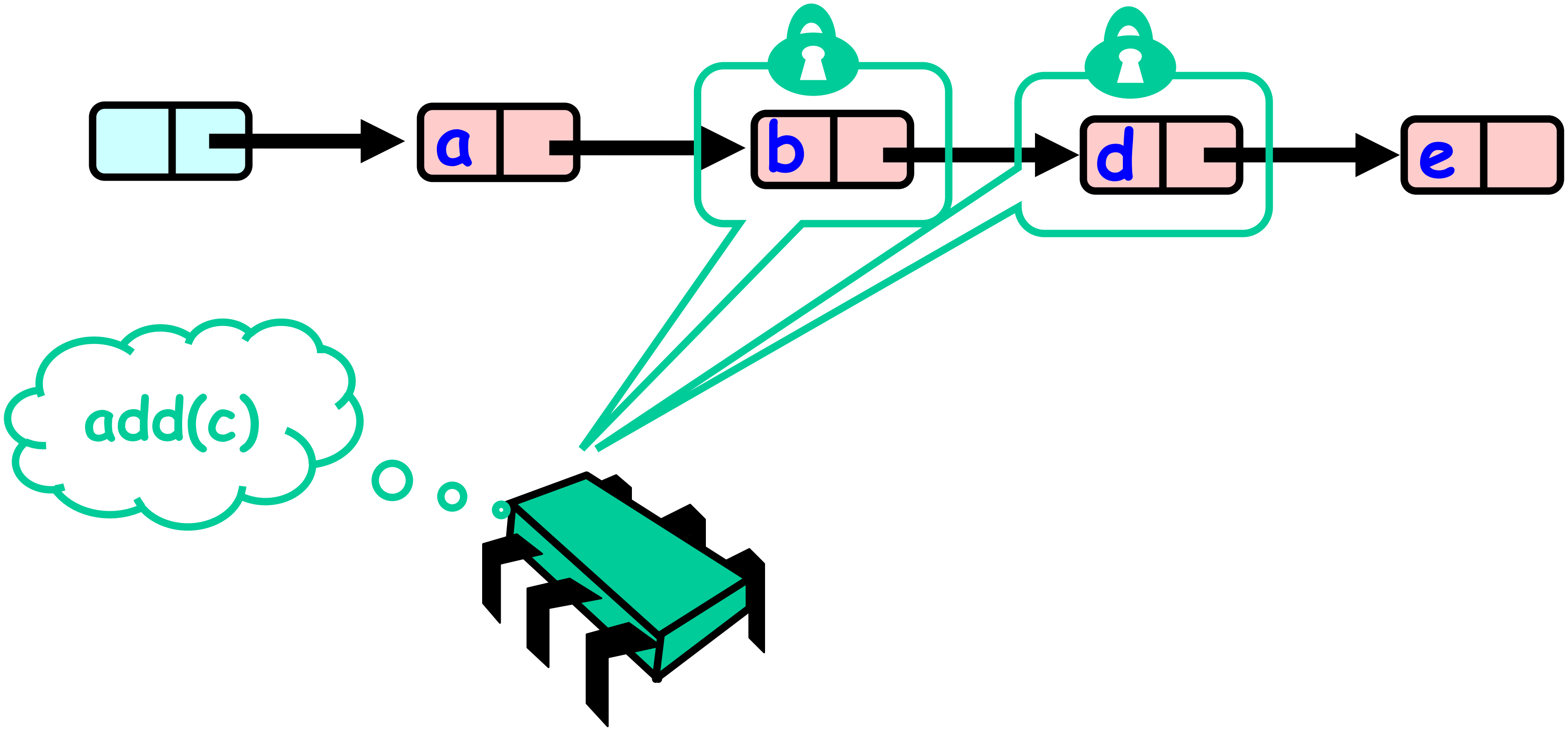


3. Optimistic

1. Describing the algorithm:

Locking

- Example of add(c):

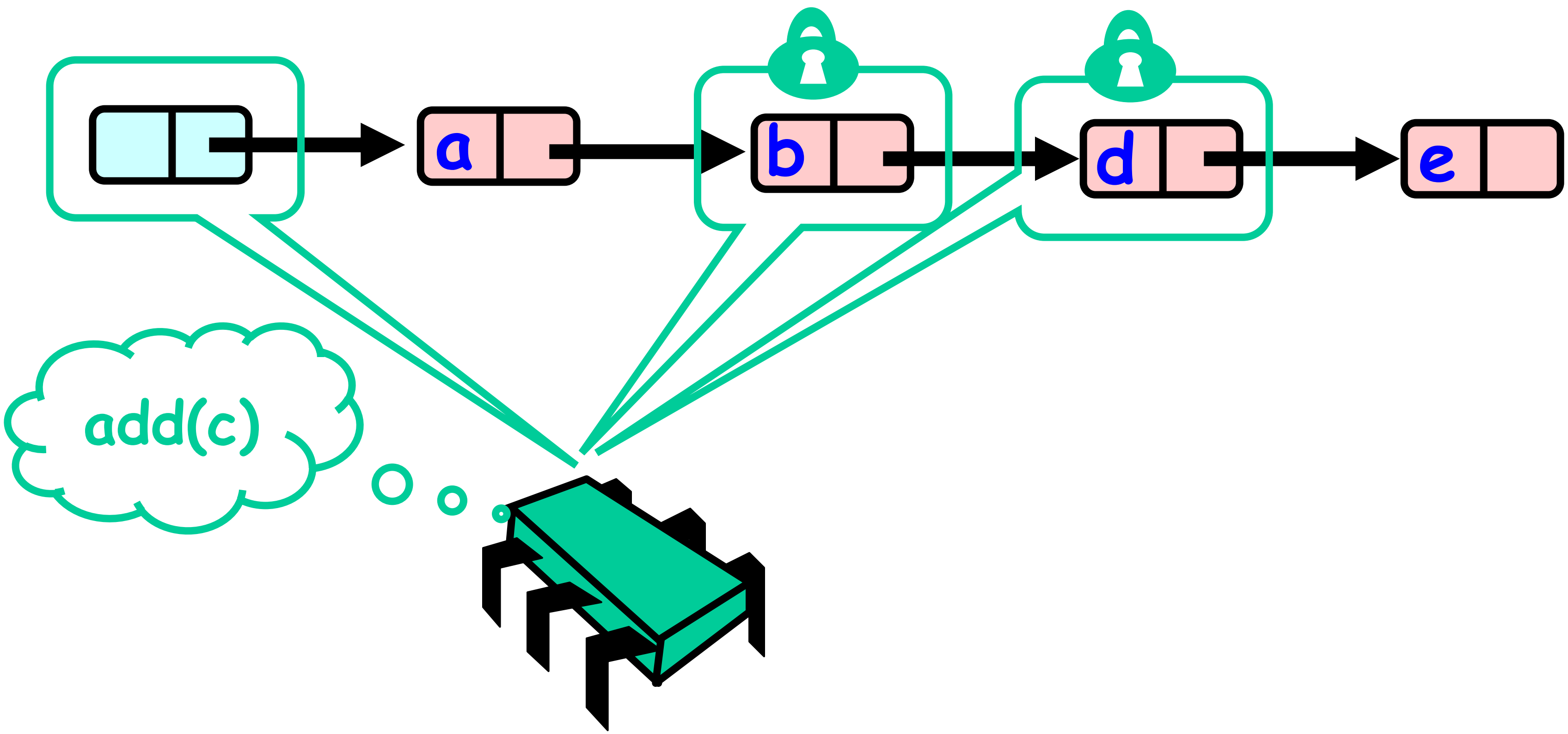


3. Optimistic

1. Describing the algorithm:

Validation 1

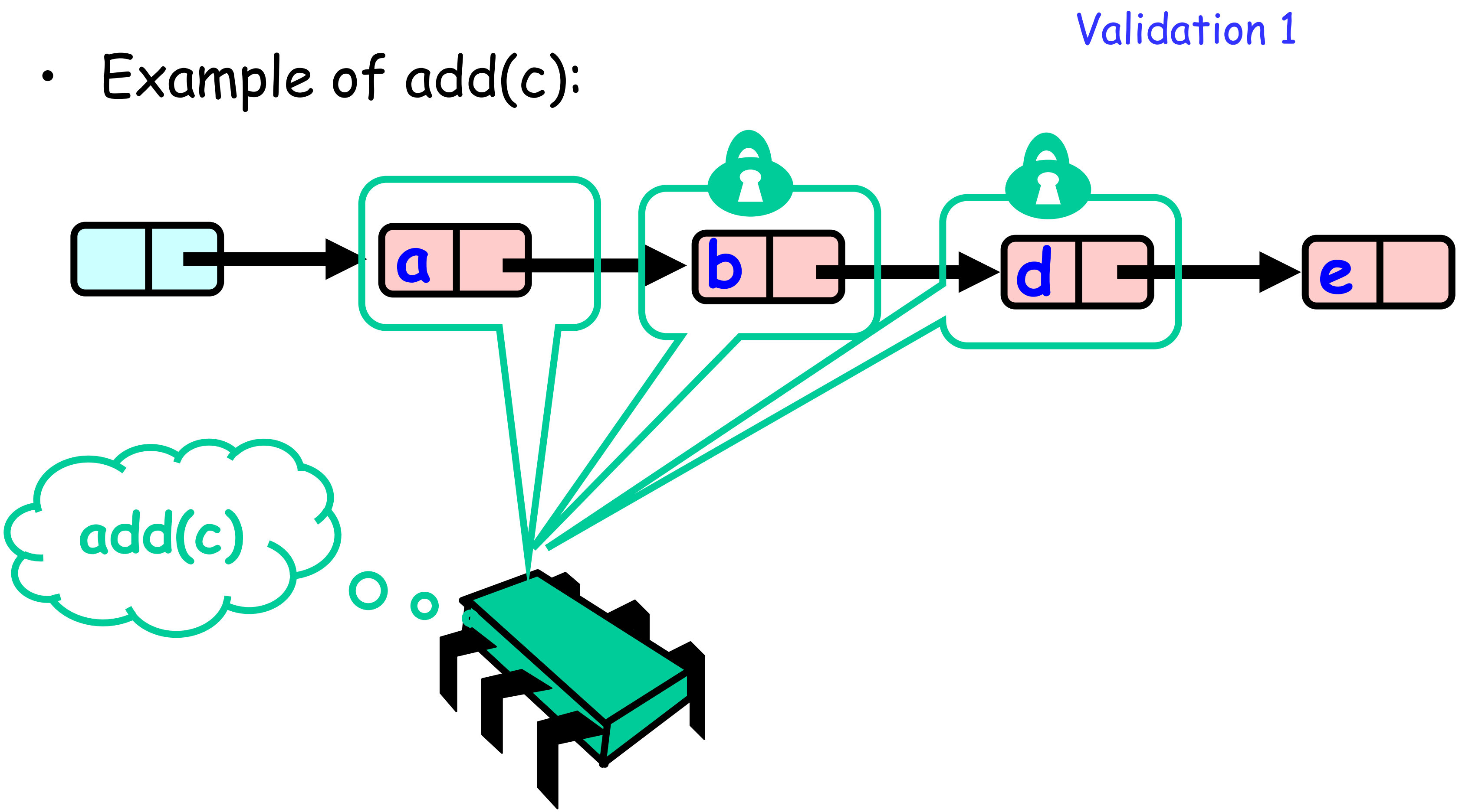
- Example of add(c):



3. Optimistic

1. Describing the algorithm:

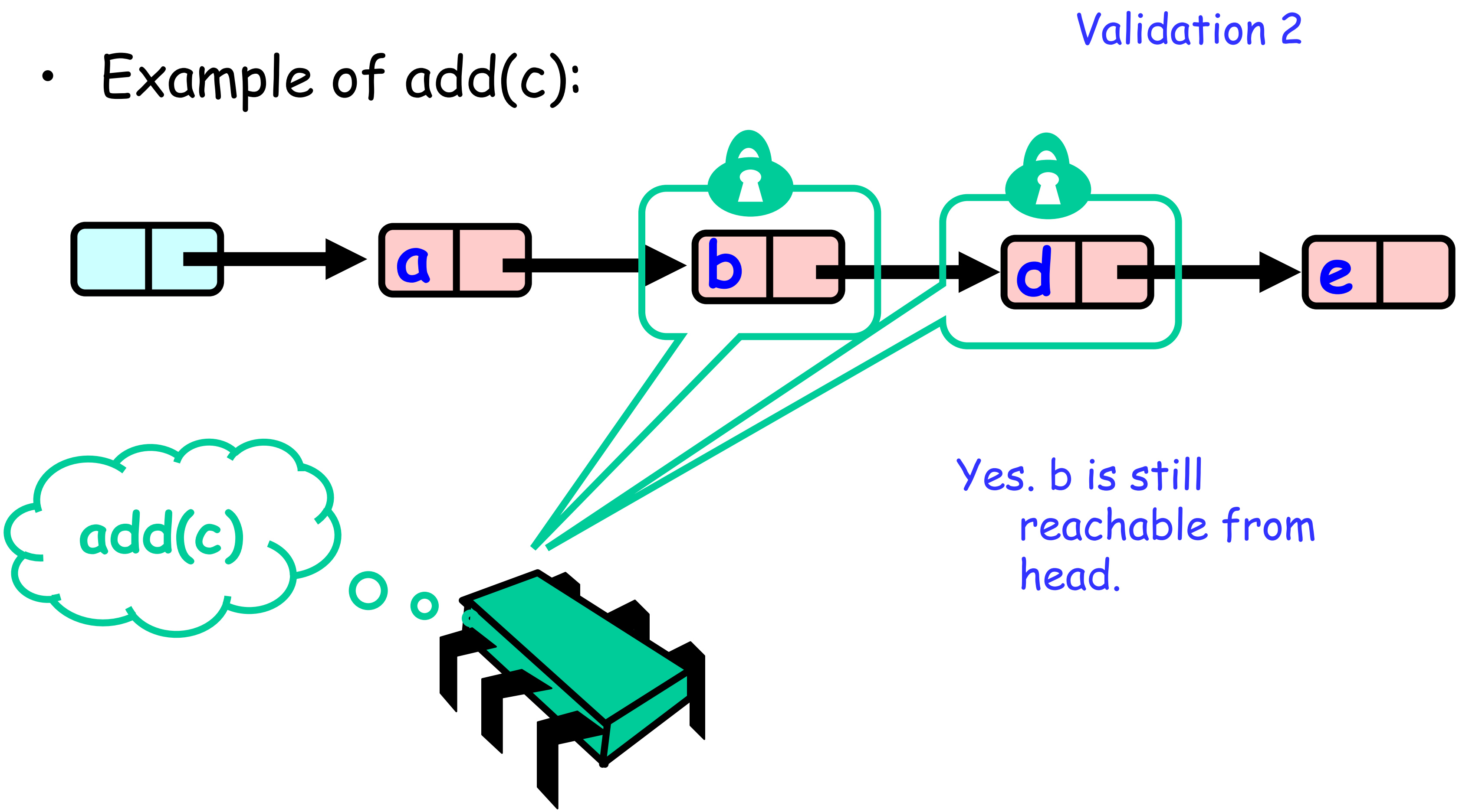
- Example of add(c):



3. Optimistic

1. Describing the algorithm:

- Example of add(c):

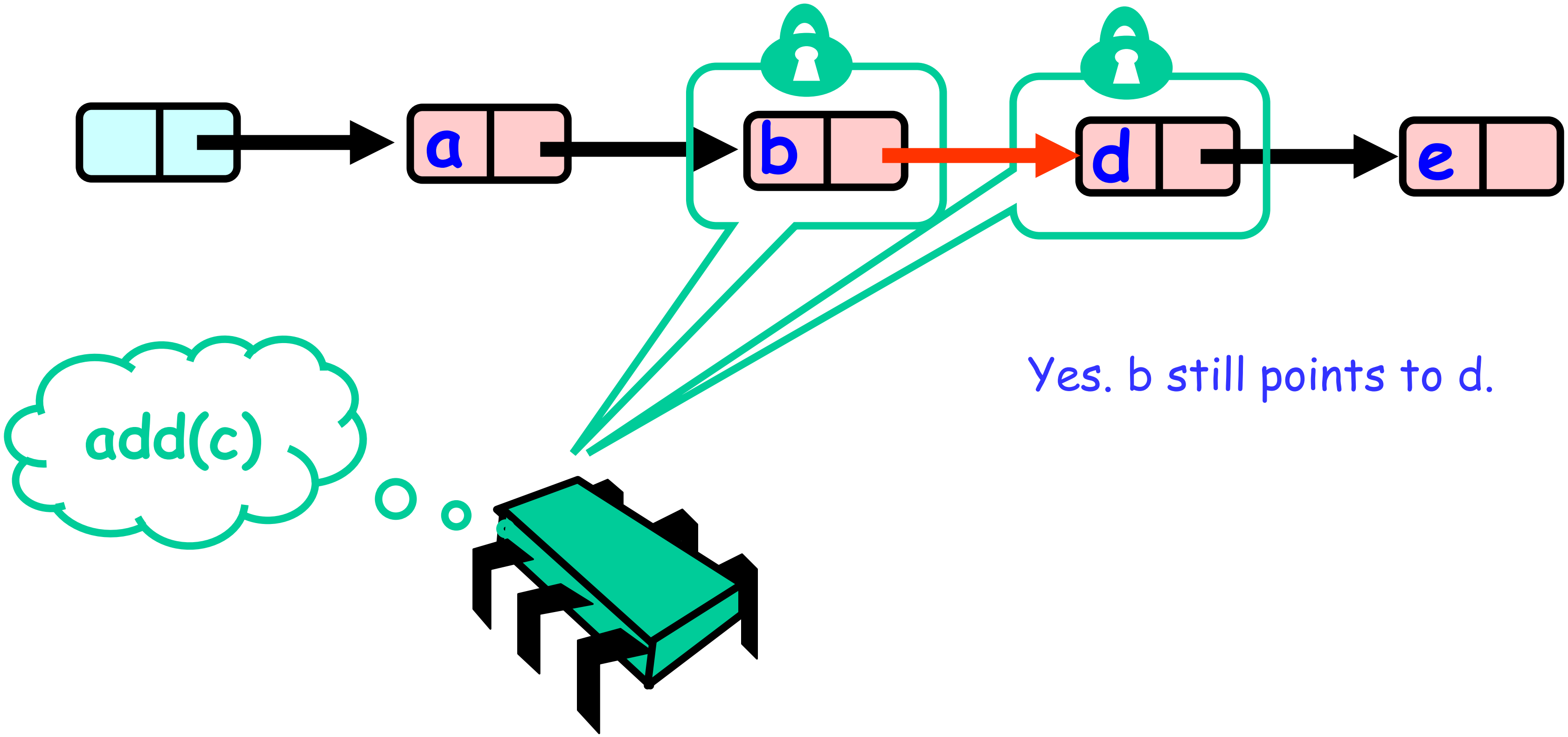


3. Optimistic

1. Describing the algorithm:

- Example of add(c):

Validation 2

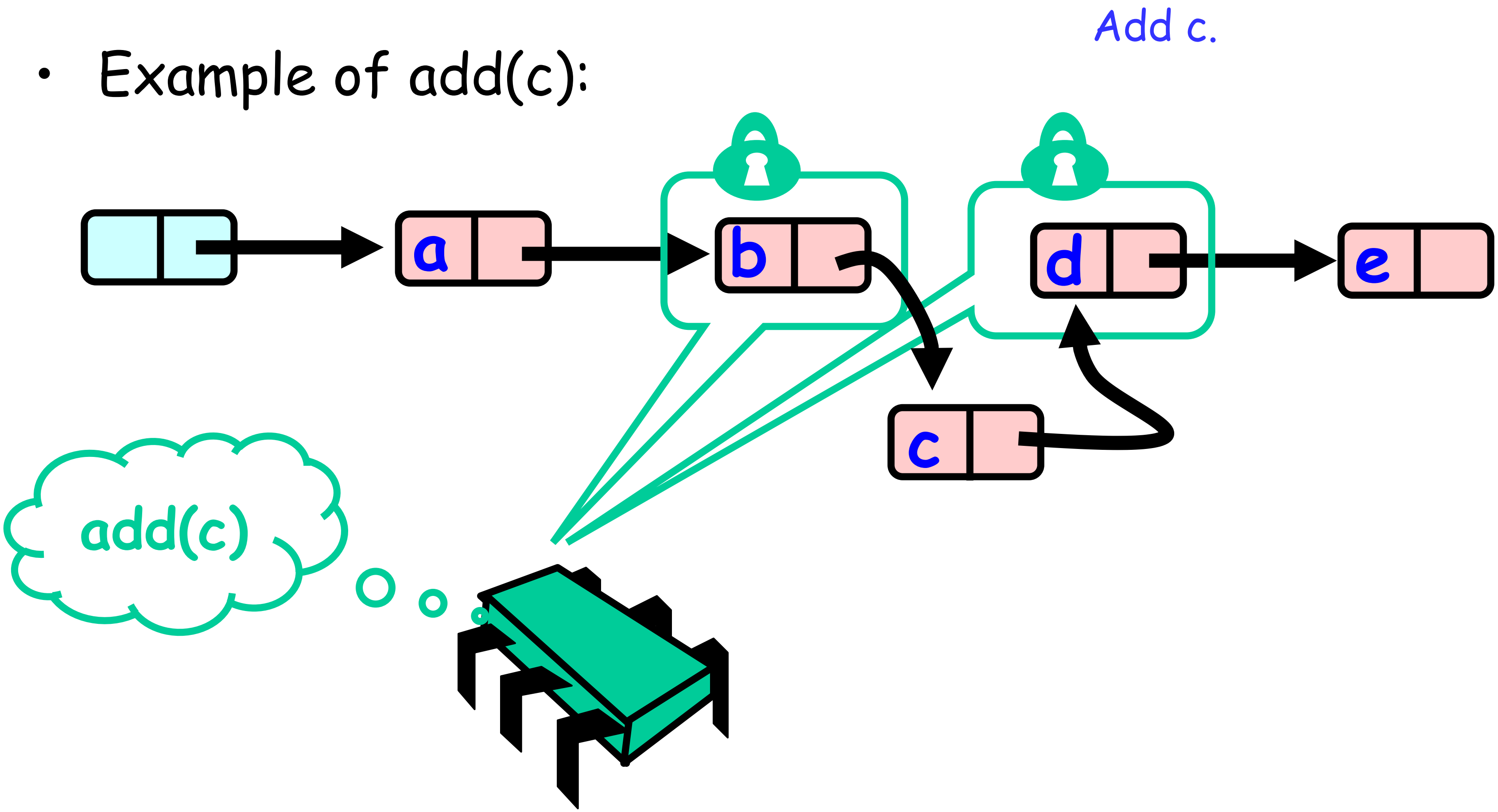


Yes. b still points to d.

3. Optimistic

1. Describing the algorithm:

- Example of add(c):



3. Optimistic

2. Explaining why every step is needed.

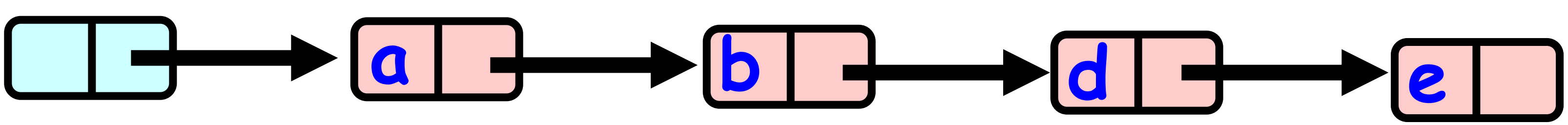
Why do we need to Validate?

3. Optimistic

2. Explaining why every step is needed.

- First: Why do we need to validate that pred is accessible from head?

- Thread A Adds(c).
- After thread A found b, before A locks. Another thread removes b.

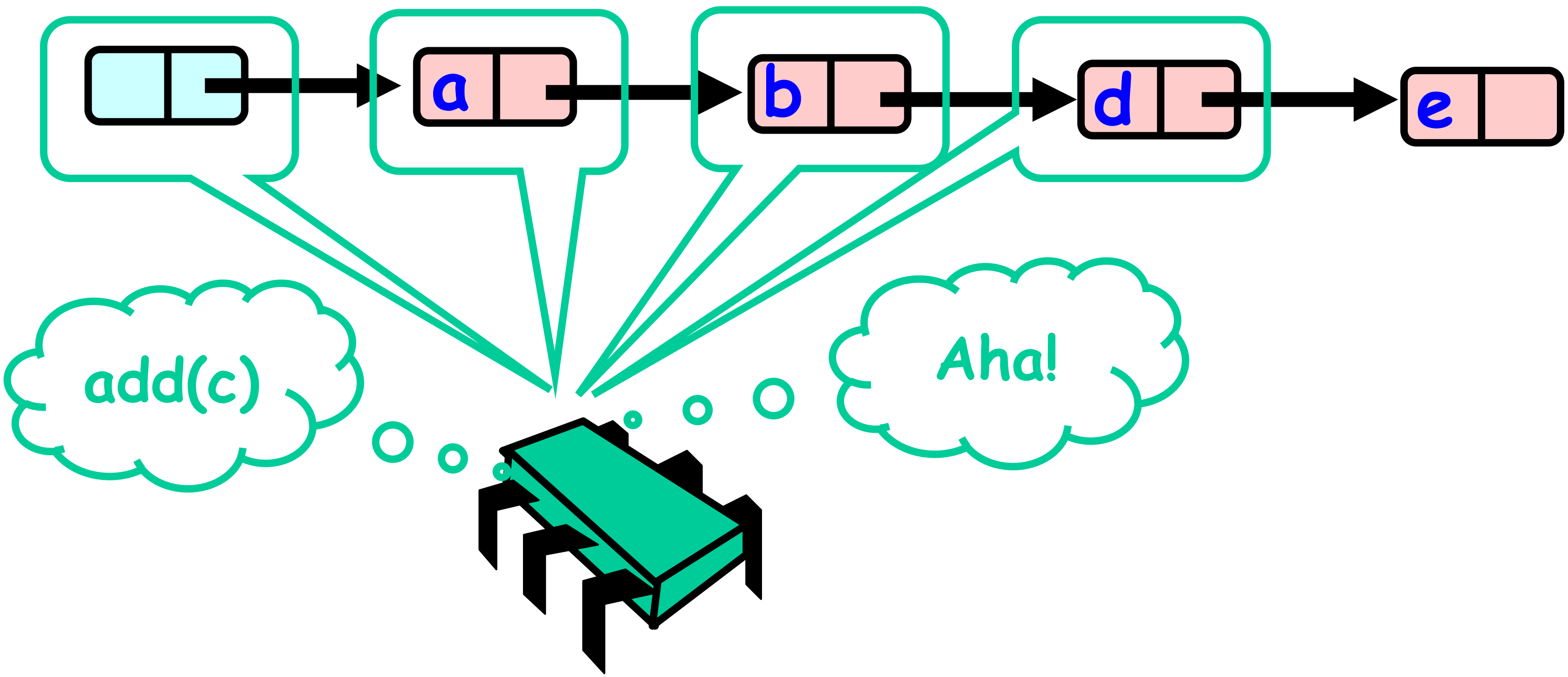


3. Optimistic

2. Explaining why every step is needed.

- Adds(c).

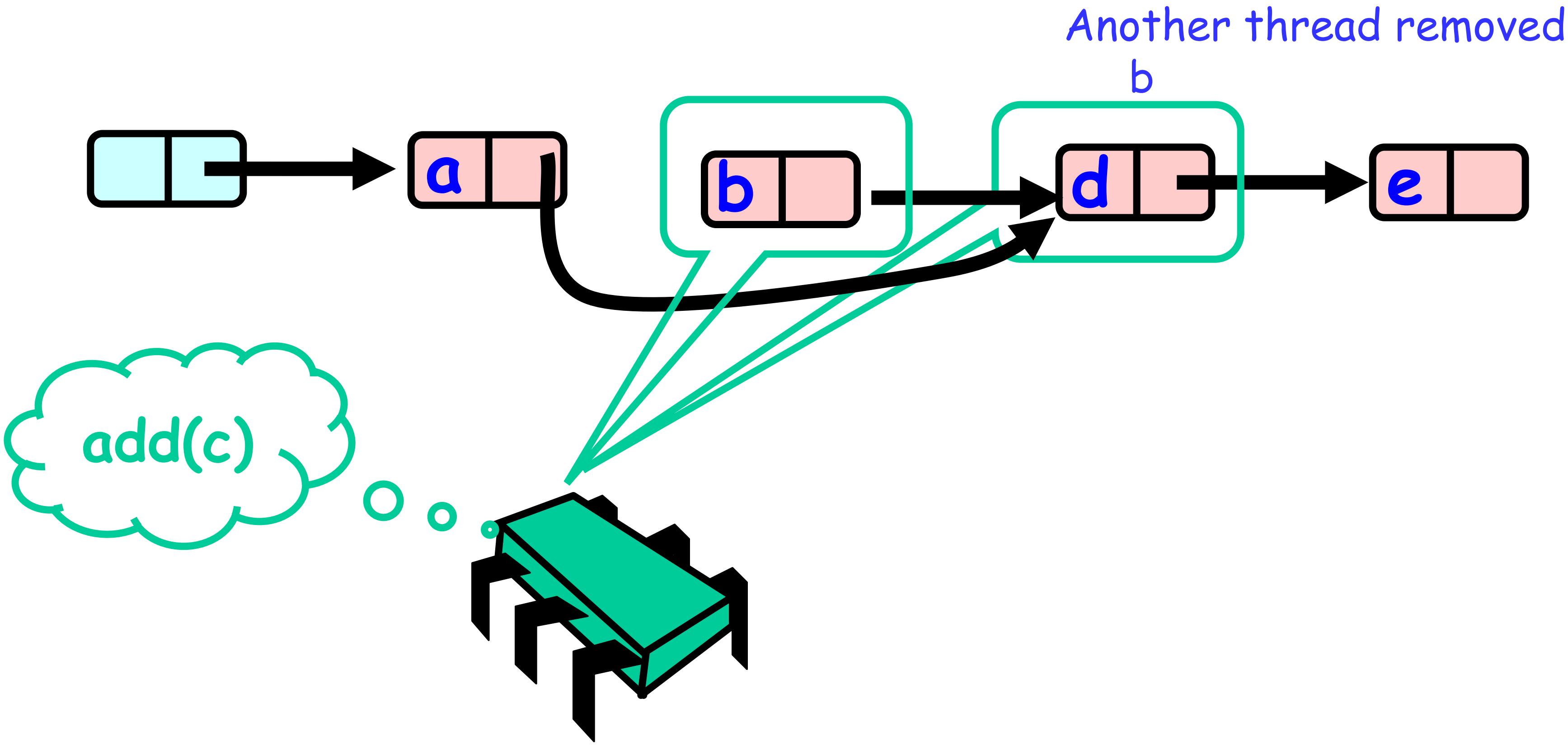
Finding without locking



3. Optimistic

2. Explaining why every step is needed.

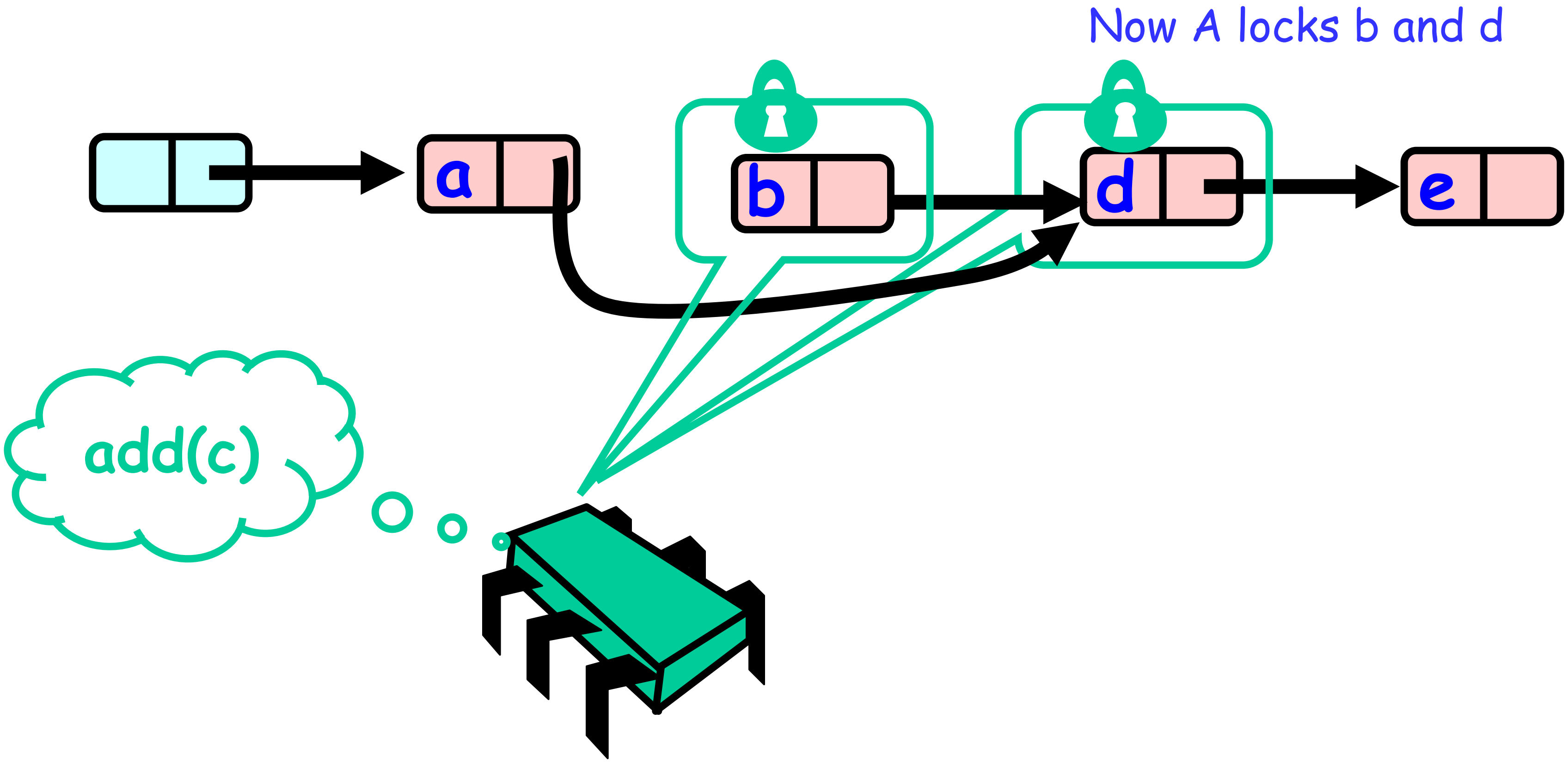
- Adds(c).



3. Optimistic

2. Explaining why every step is needed.

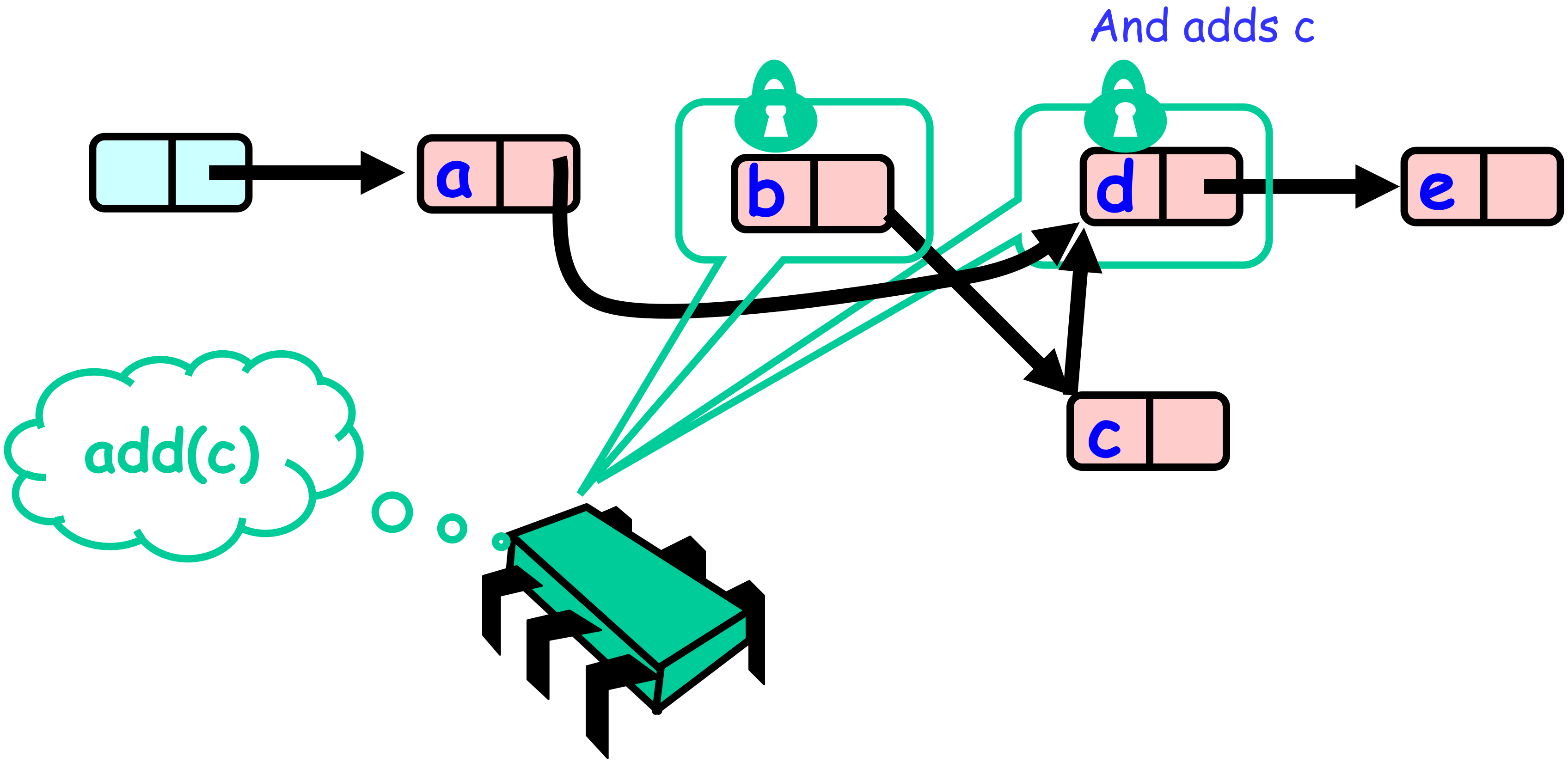
- Adds(c).



3. Optimistic

2. Explaining why every step is needed.

- Adds(c).

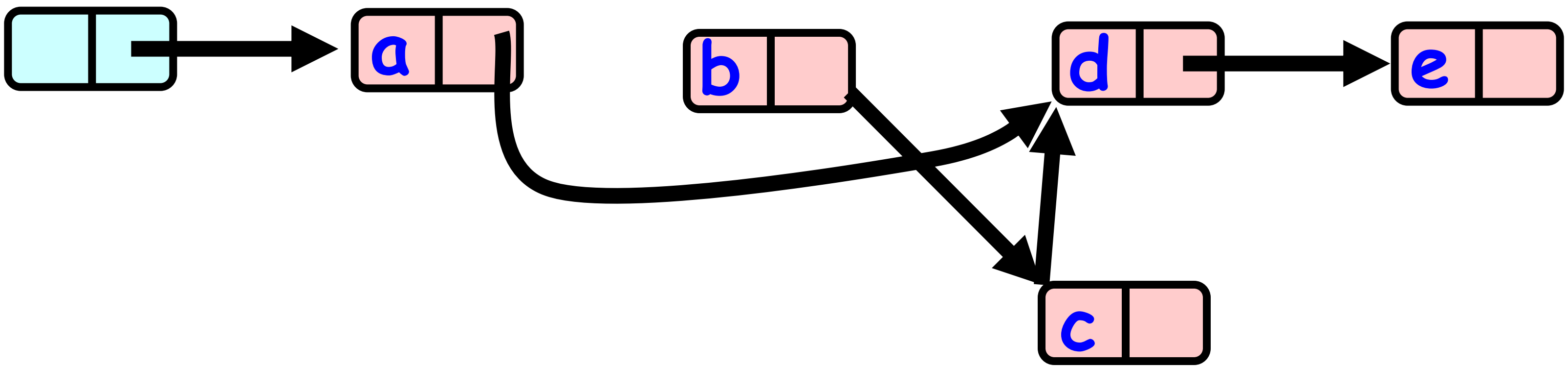


3. Optimistic

2. Explaining why every step is needed.

- Adds(c).

Now frees the locks.



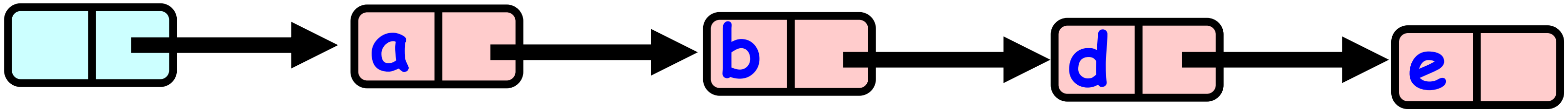
But c isn't added!

3. Optimistic

2. Explaining why every step is needed.

- Second: Why do we need to validate that pred Still points to curr?

- Thread A removes(d).
- then thread A found b, before A locks. Another thread adds(c).

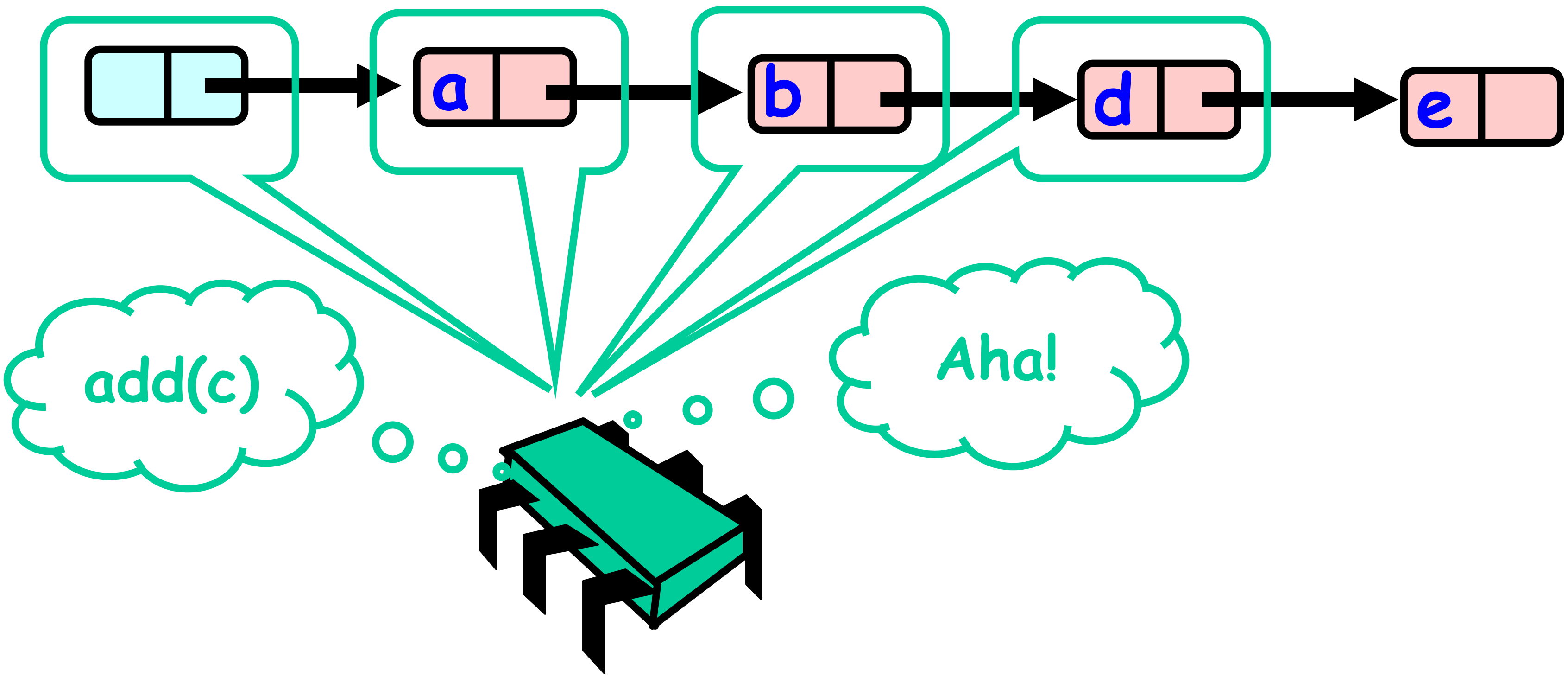


3. Optimistic

2. Explaining why every step is needed.

- Removes(d)

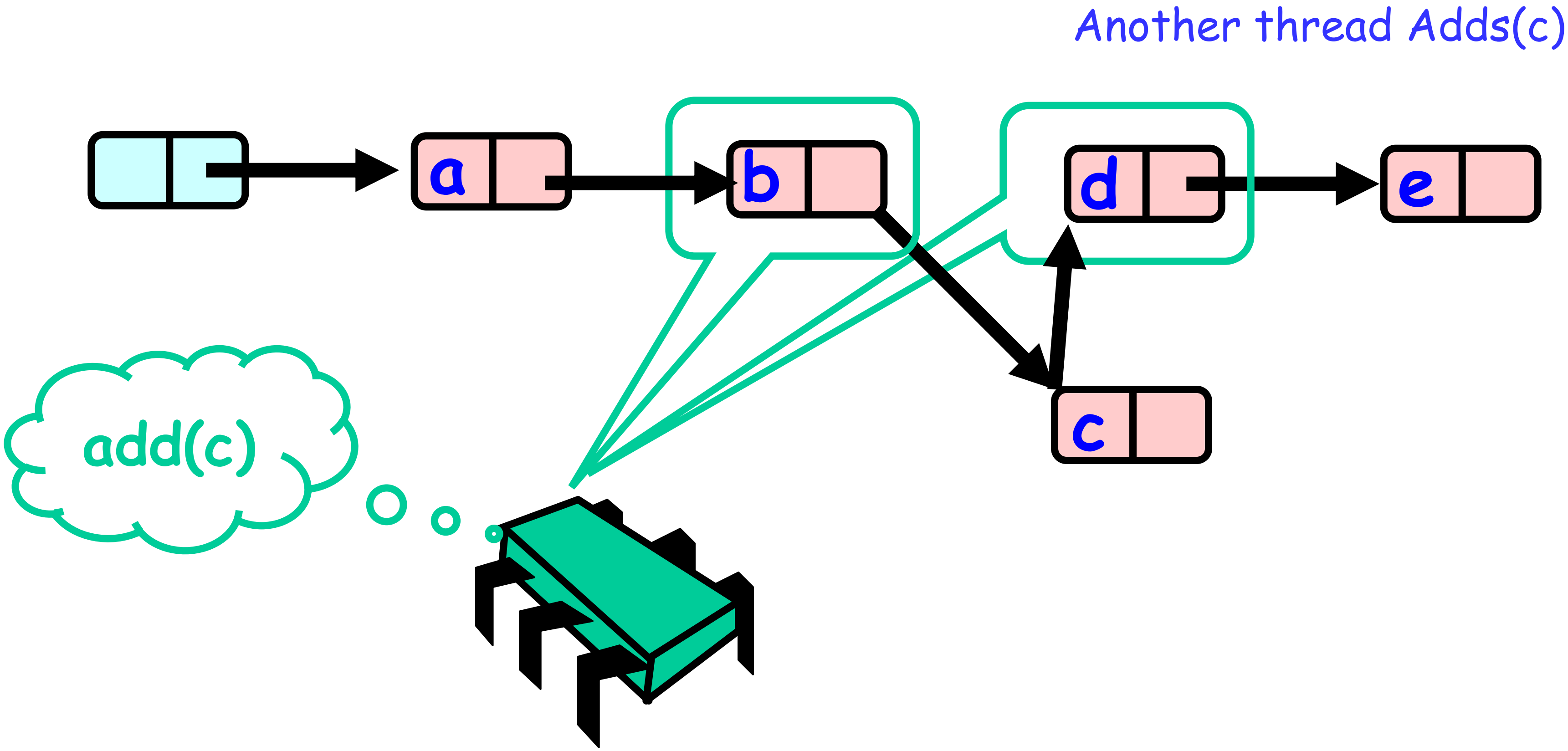
Finding without locking



3. Optimistic

2. Explaining why every step is needed.

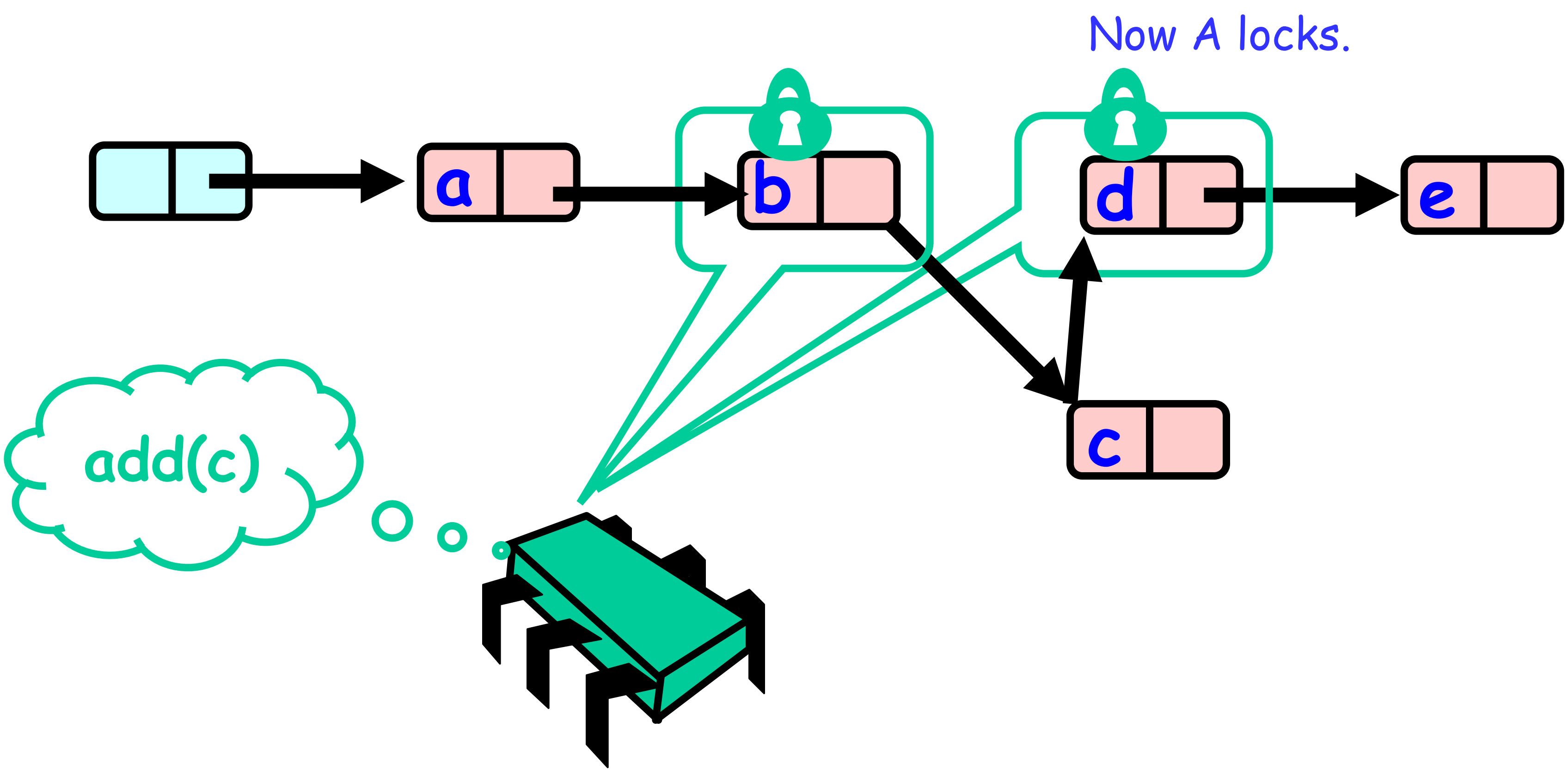
- Removes(d)



3. Optimistic

2. Explaining why every step is needed.

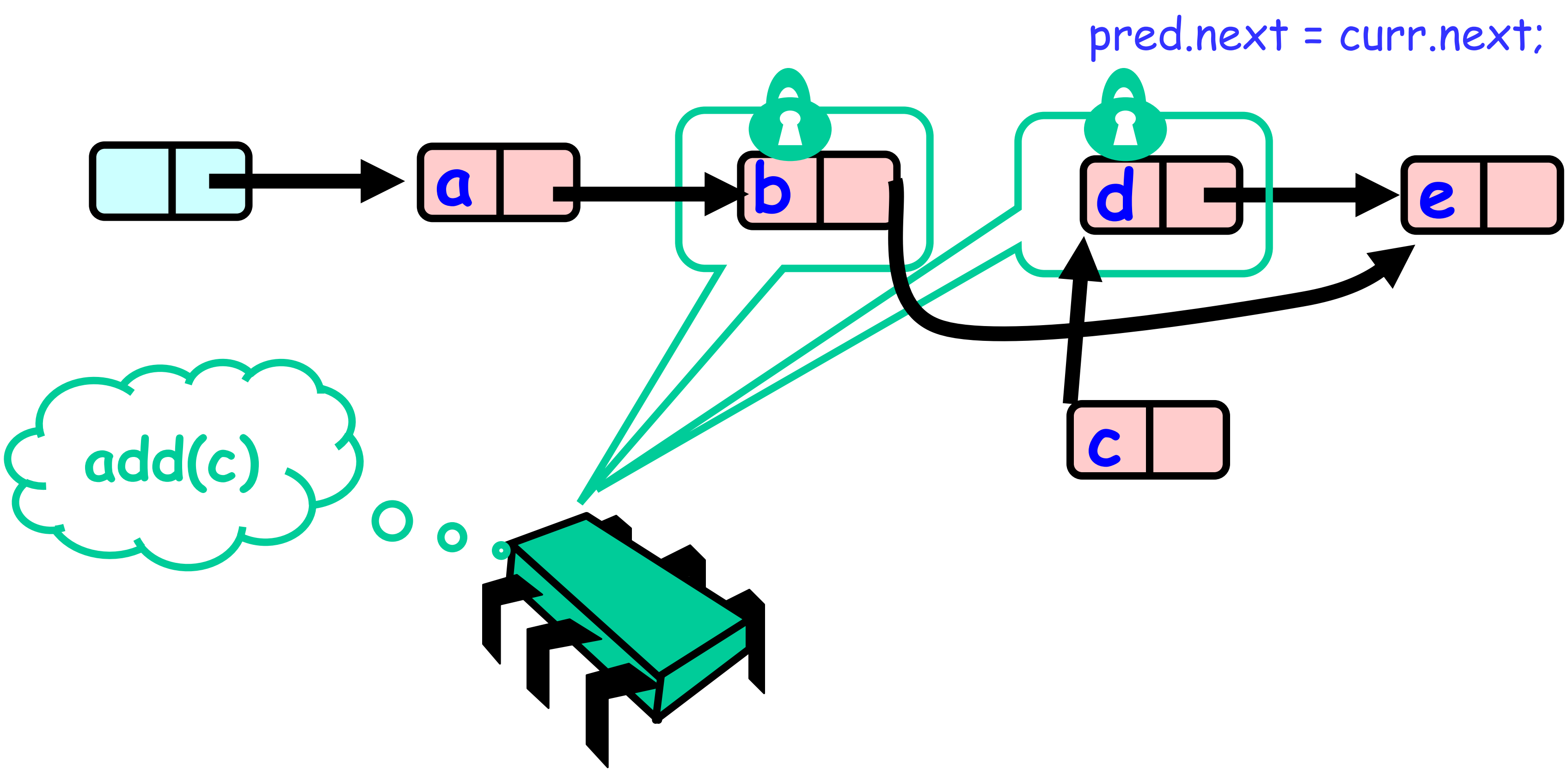
- Removes(d)



3. Optimistic

2. Explaining why every step is needed.

- Removes(d)

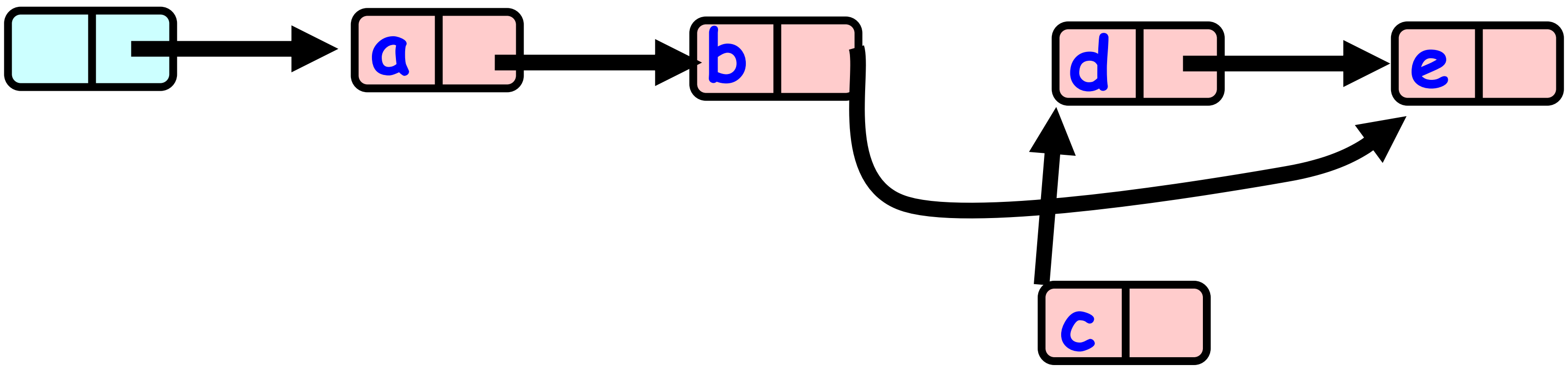


3. Optimistic

2. Explaining why every step is needed.

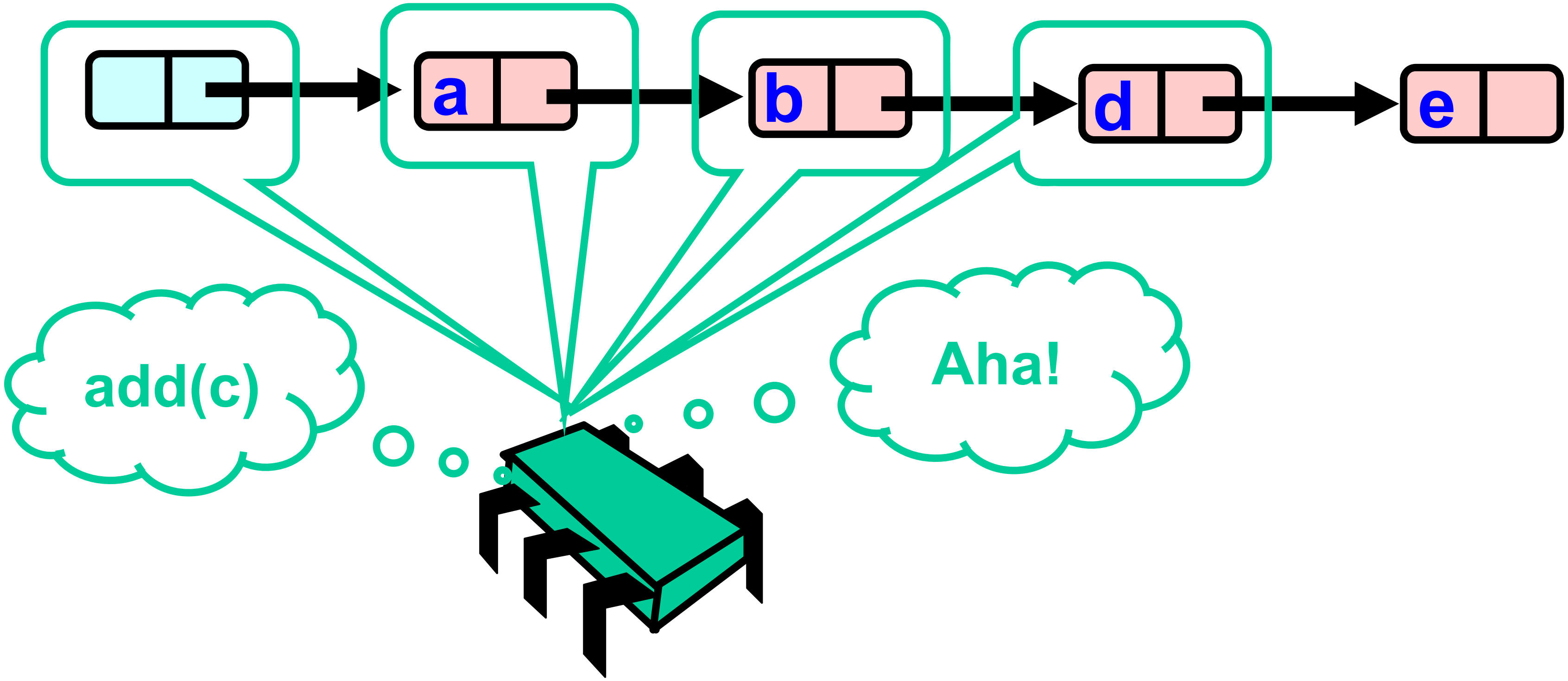
- Removes(d)

Now frees the locks.

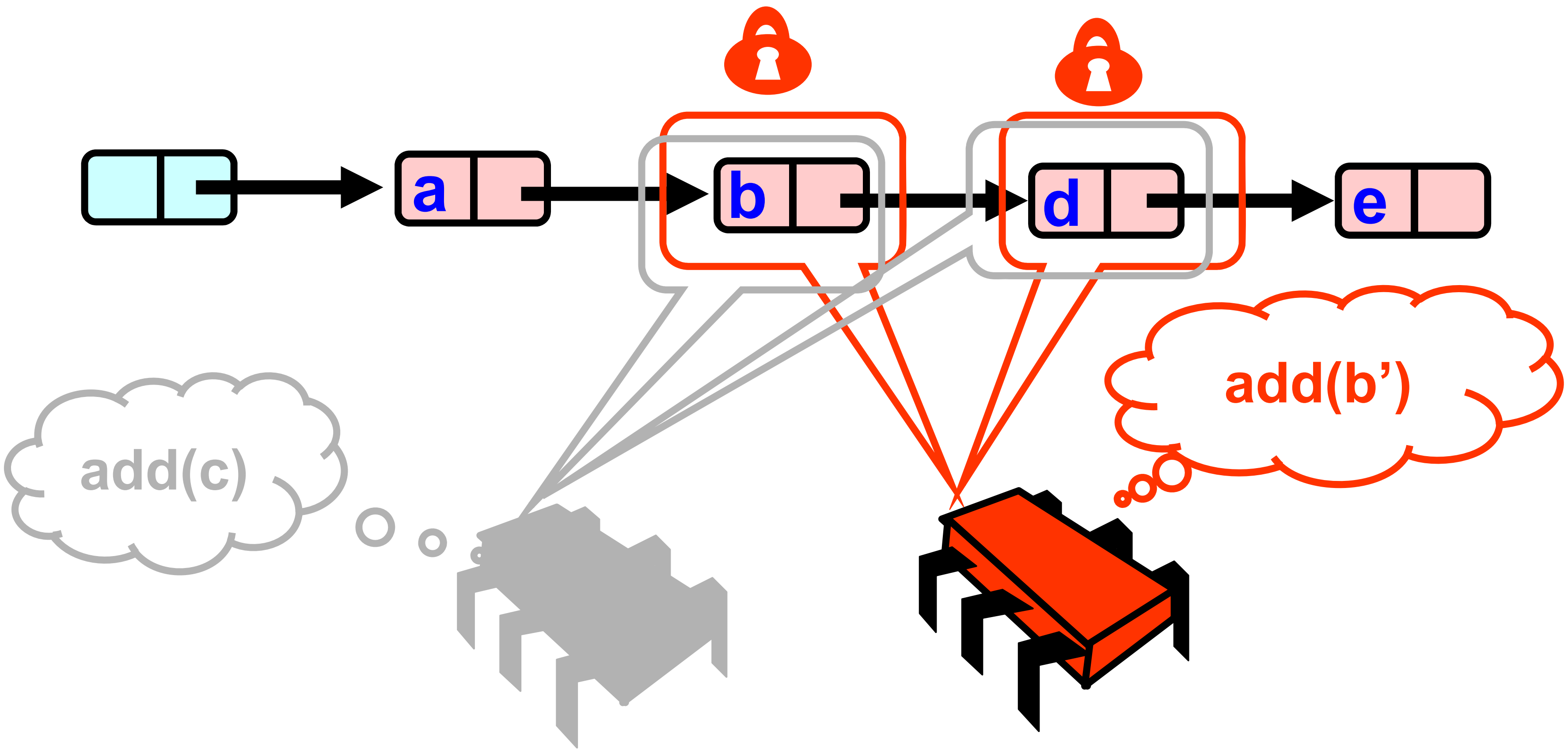


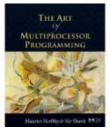
Instead c and d were deleted!

What Else Could Go Wrong?

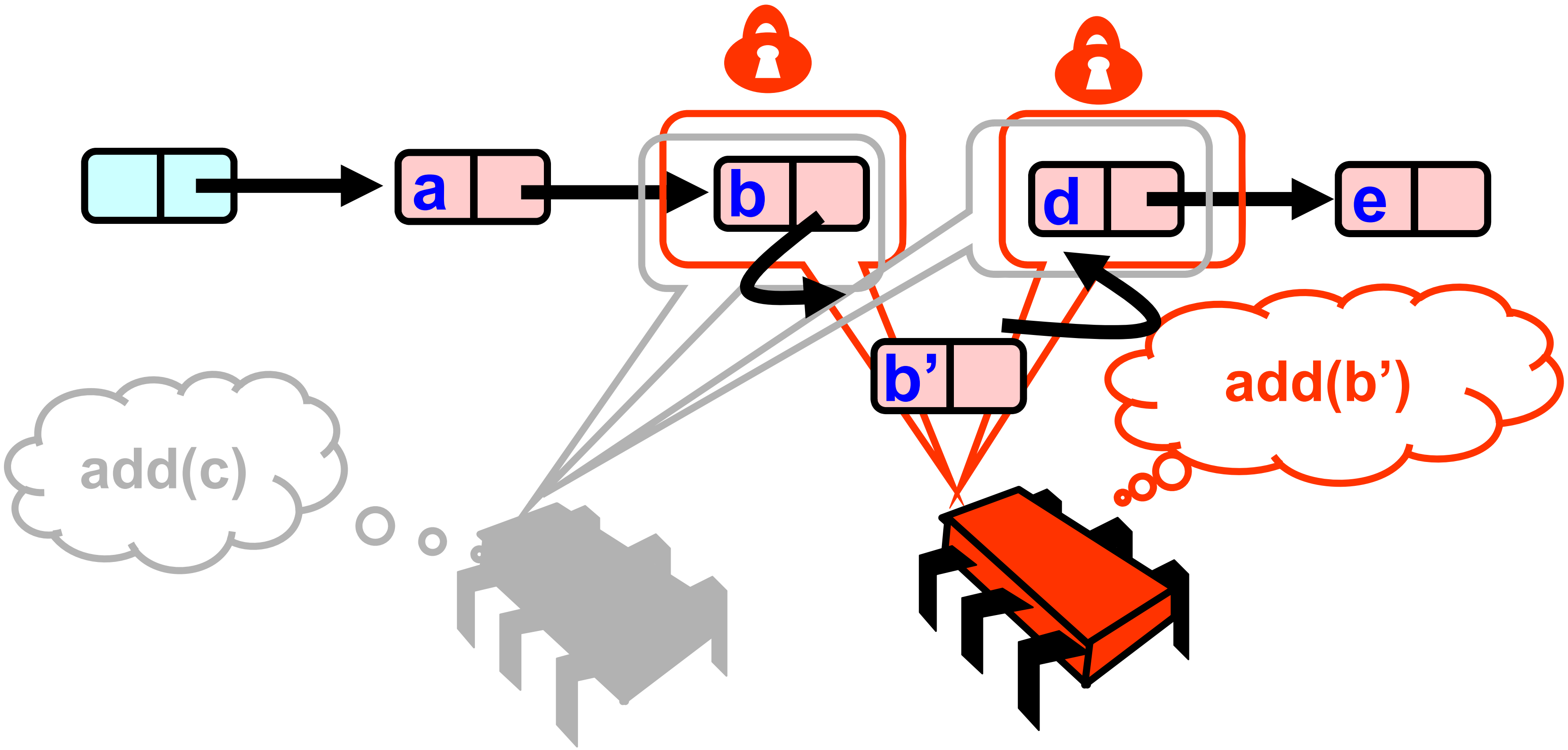


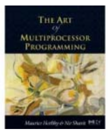
What Else Coould Go Wrong?



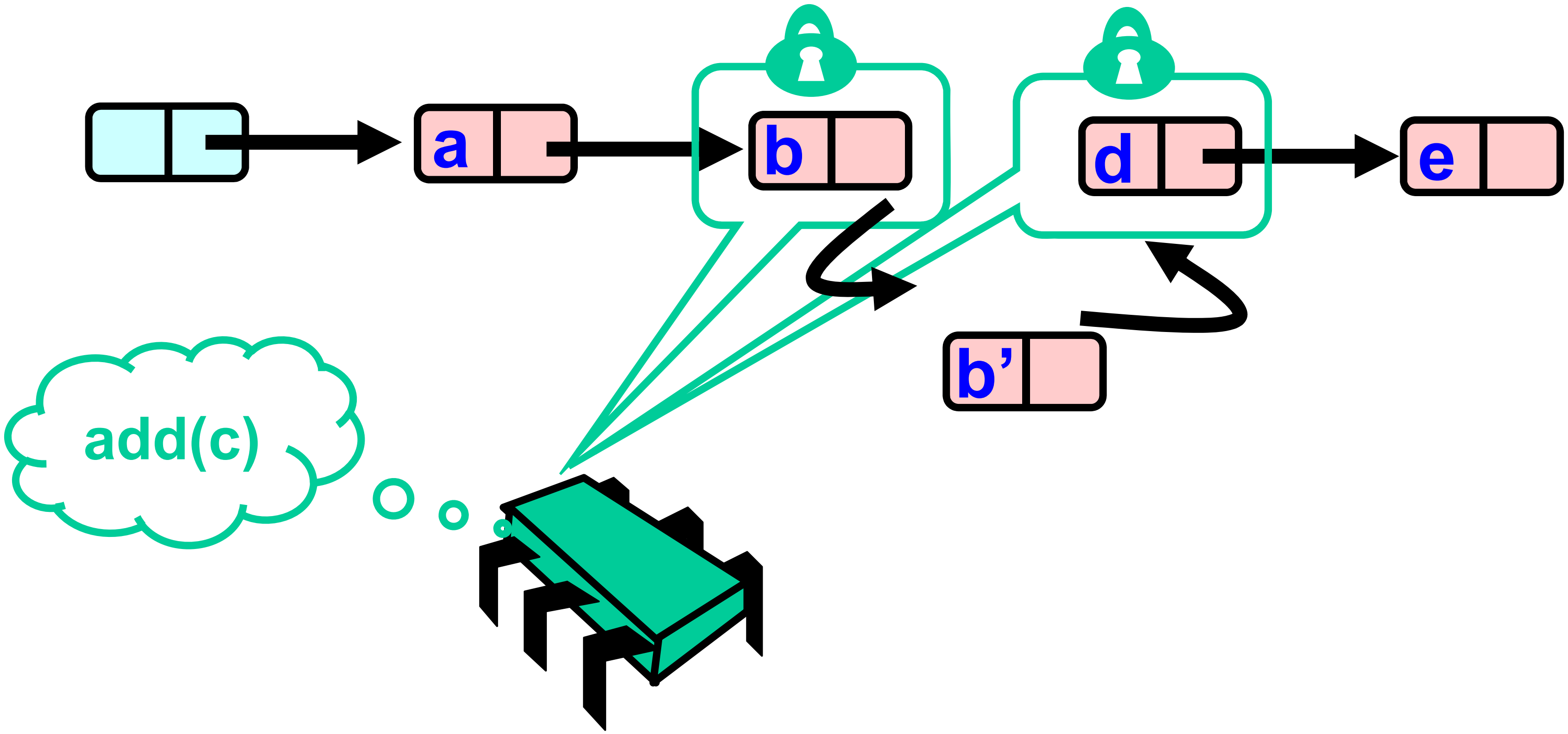


What Else Coould Go Wrong?

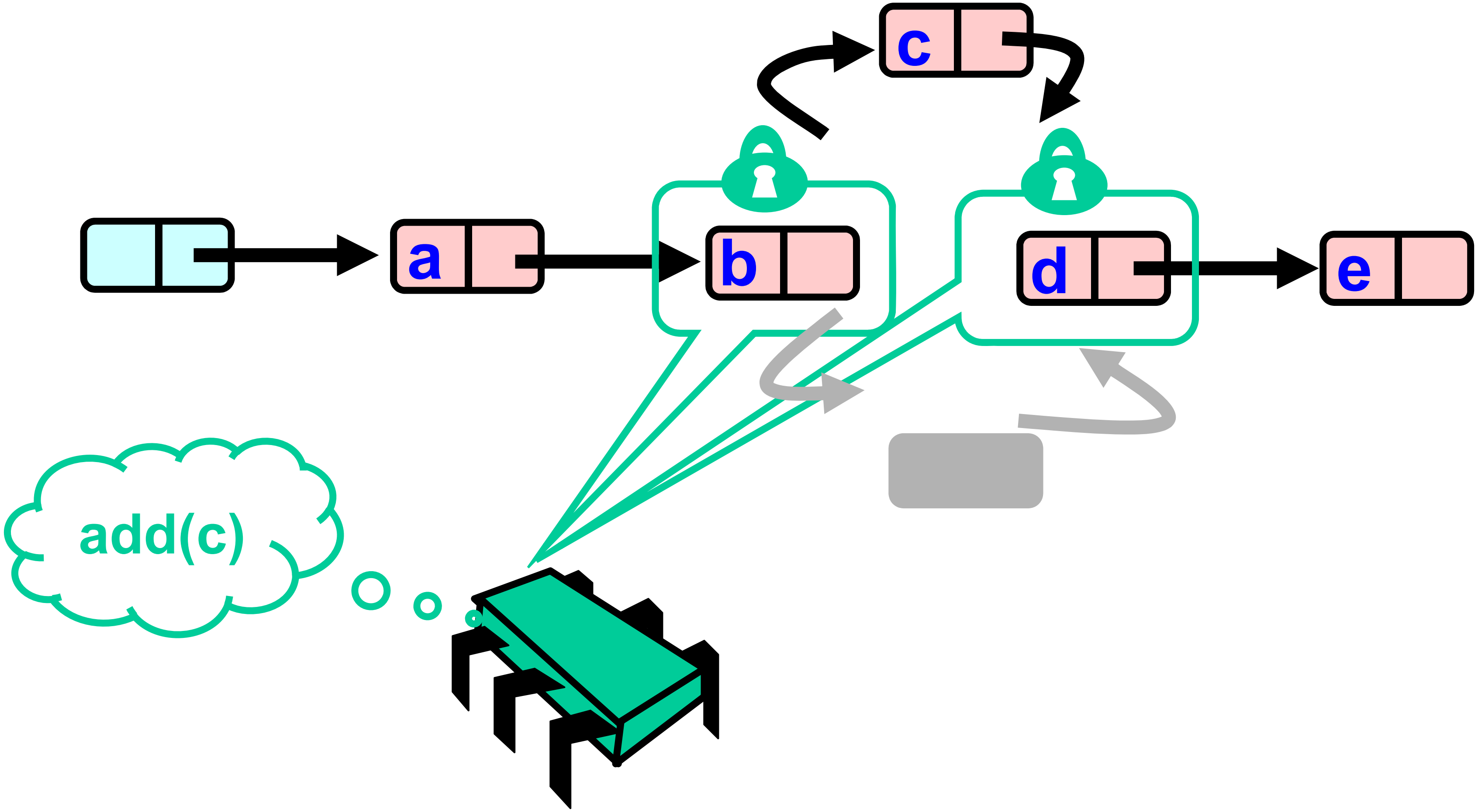


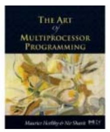


What Else Could Go Wrong?



What Else Could Go Wrong?





3. Optimistic

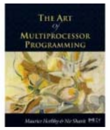
Important comment.

- Do notice that threads might traverse deleted nodes. May cause problems to our Rep-Invariant.

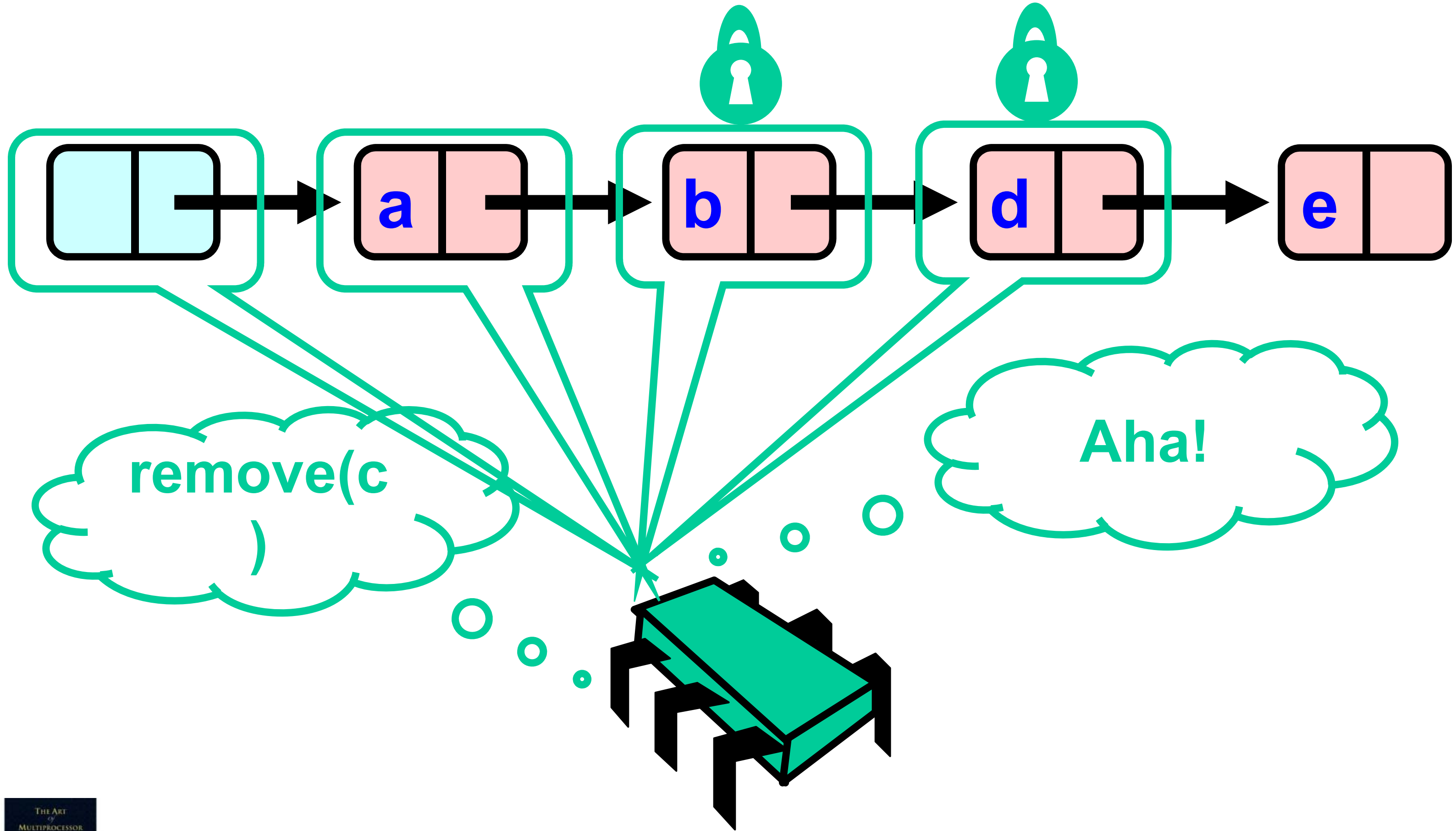
- Careful not to recycle to the lists nodes that were deleted while threads are in a middle of an operation.
- With a garbage collection language like java – ok.
- For C – you need to solve this manually.

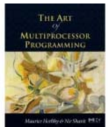
Correctness

- 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

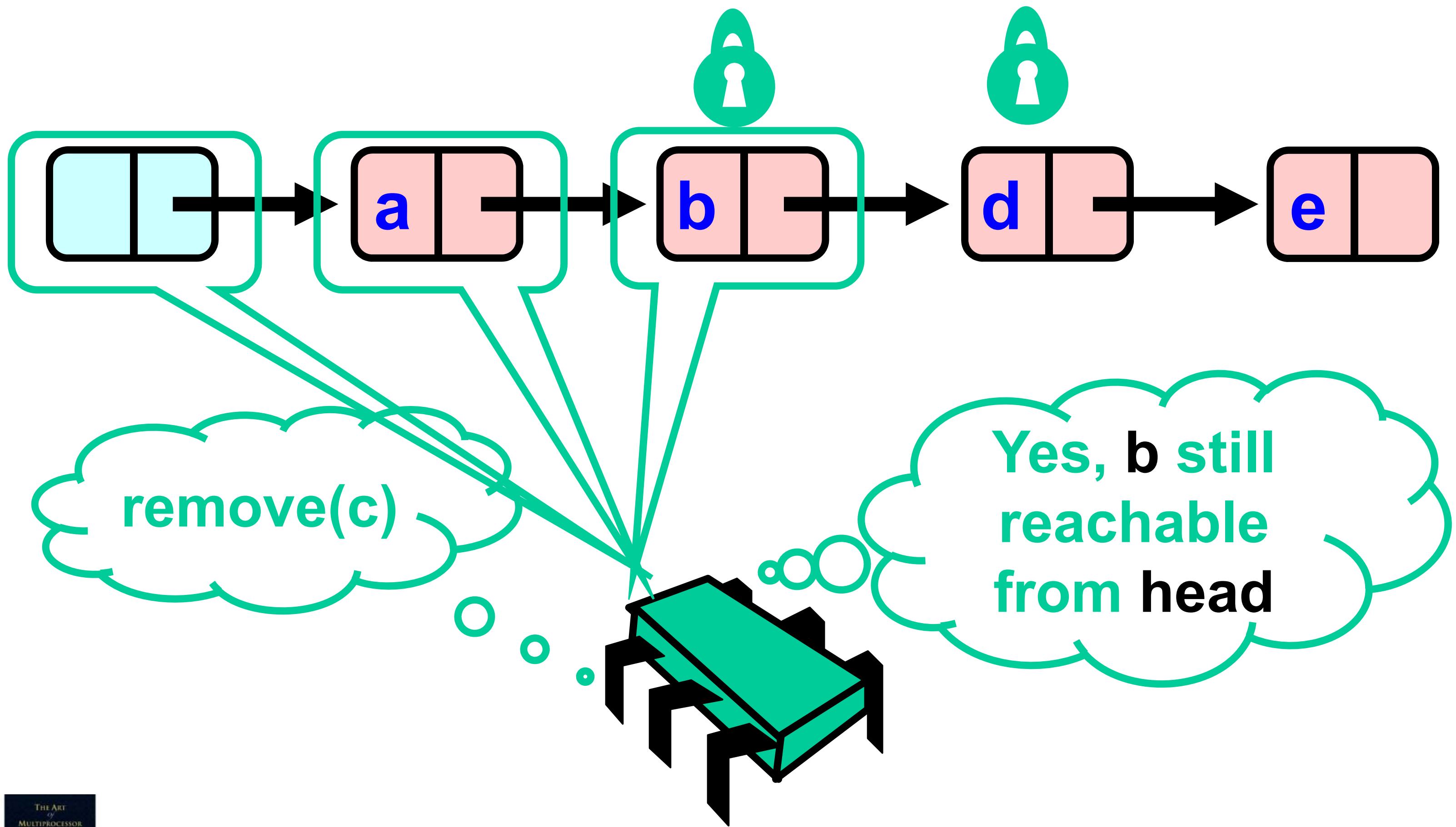


Unsuccessful Remove

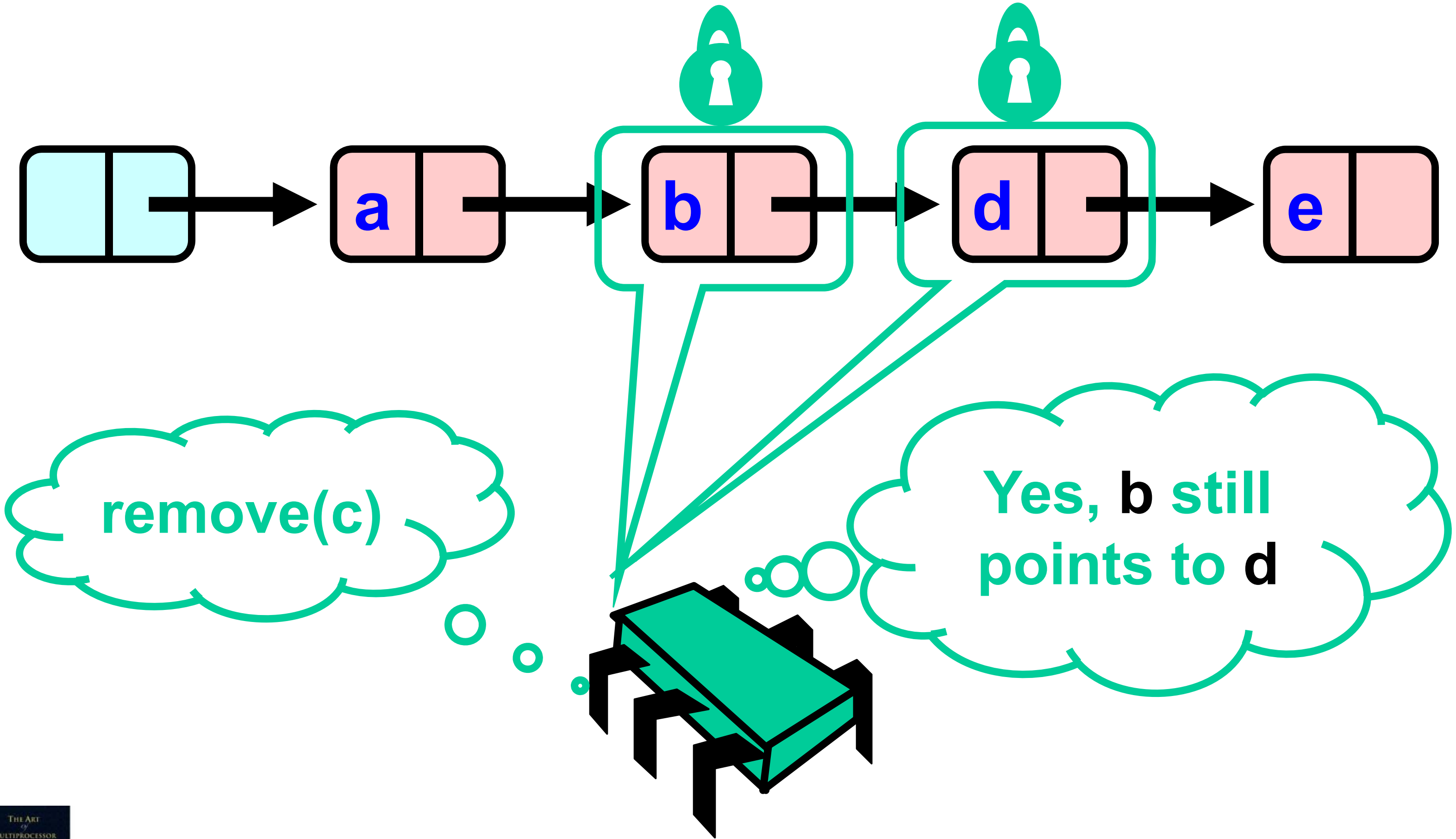


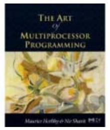


Validate (1)

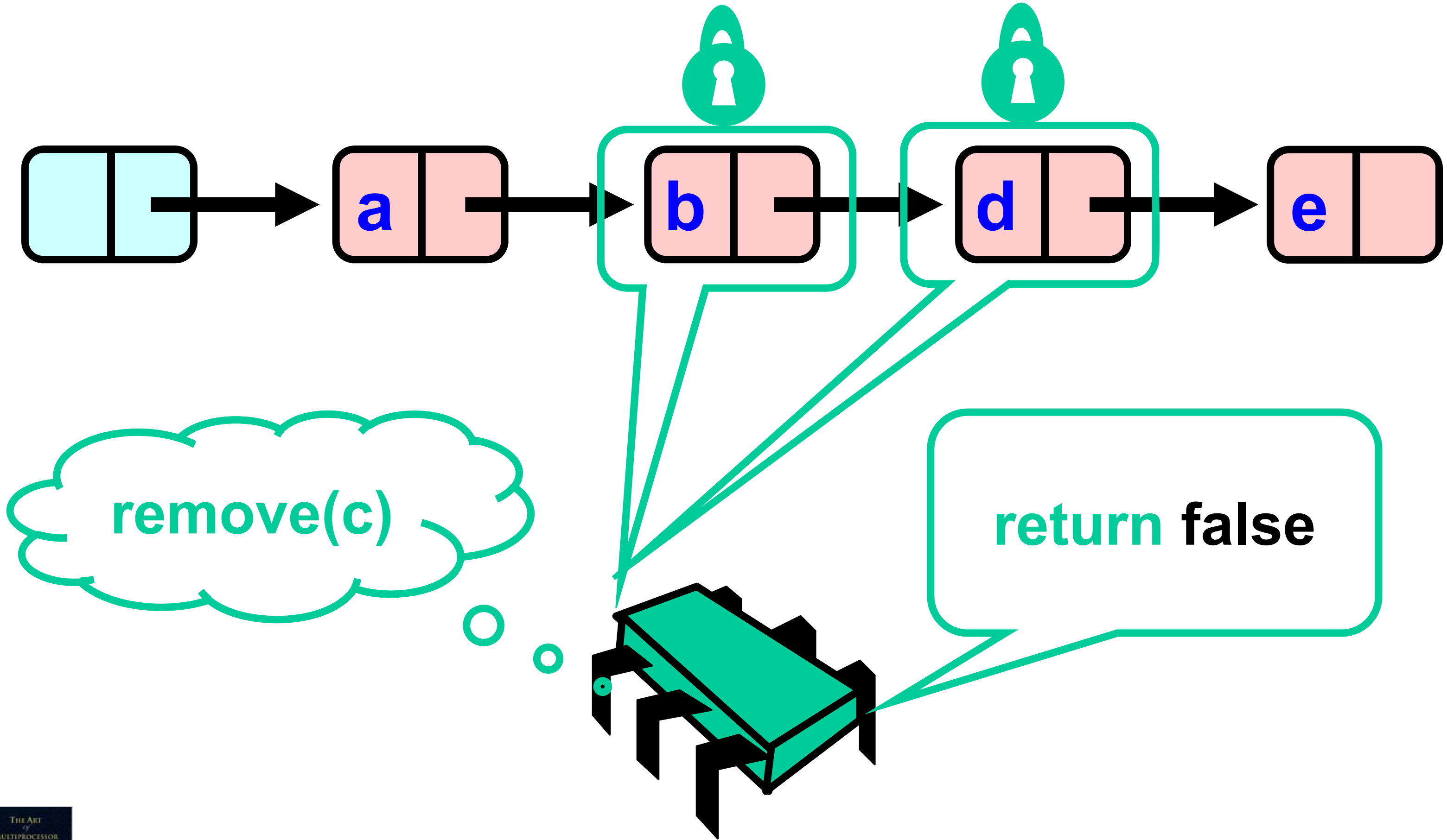


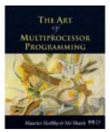
Validate (2)





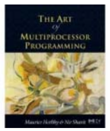
OK Computer





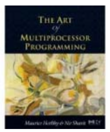
Correctness

- 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



Validation

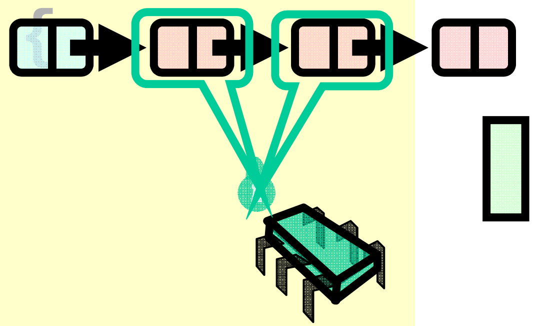
```
private boolean
 validate(Node pred,
          Node curry) {
 Node node = head;
 while (node.key <= pred.key) {
  if (node == pred)
   return pred.next == curr;
  node = node.next;
 }
 return false;
}
```



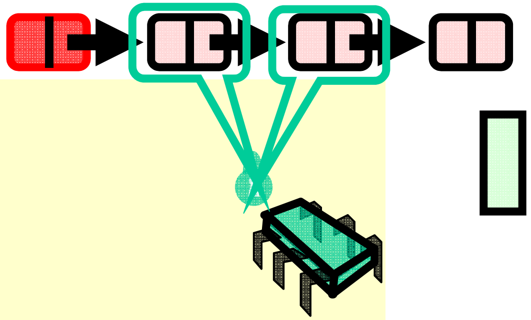
Validation

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

Predecessor & current nodes



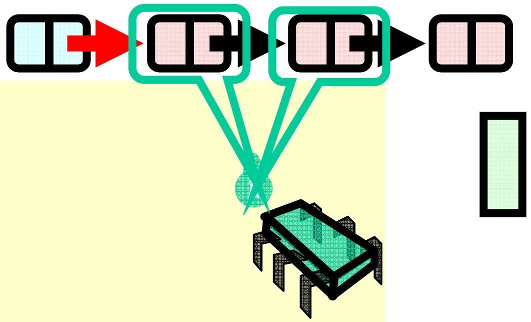
Validation



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

Begin at the beginning

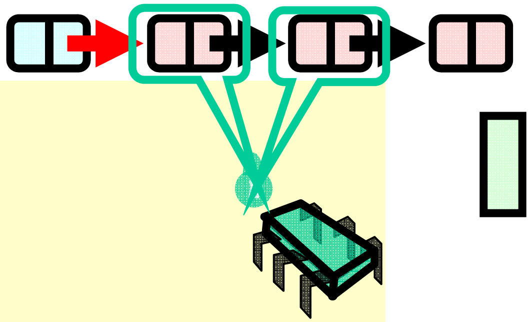
Validation



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

Search range of keys

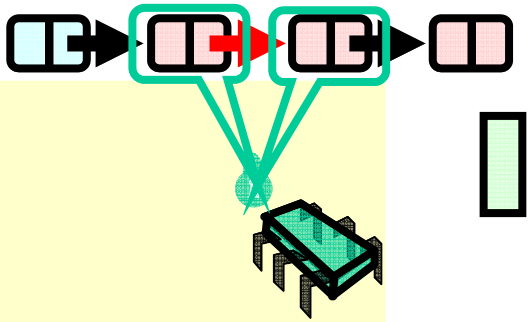
Validation



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

Predecessor reachable

Validation



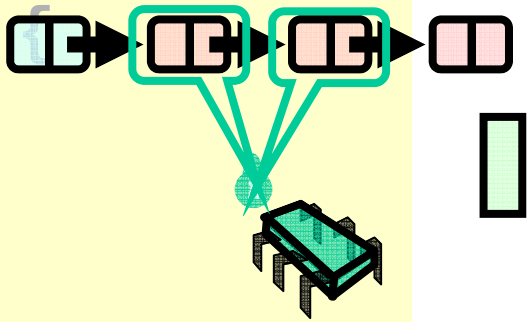
```
private boolean
 validate(Node pred,
          Node curr) {
 Node node = head;
 while (node.key <= pred.key) {
  if (node == pred)
   return pred.next == curr;
  node = node.next;
 }
 return false;
}
```

Is current node next?

Validation

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

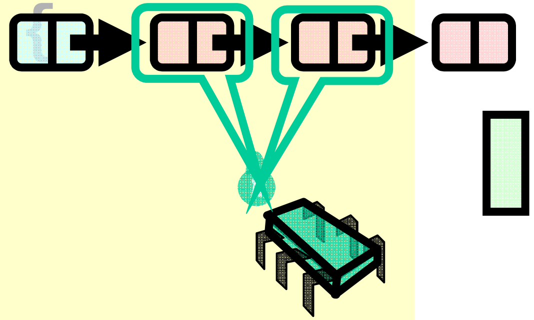
Otherwise move on



Validation

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

Predecessor not reachable



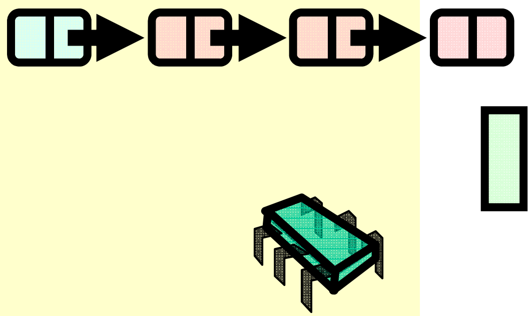
Remove: searching

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

Remove: searching

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

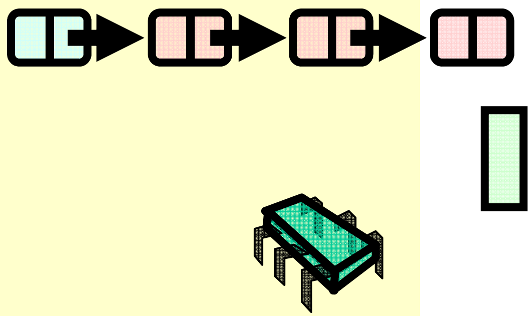
Search key

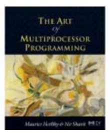


Remove: searching

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

Retry on synchronization conflict

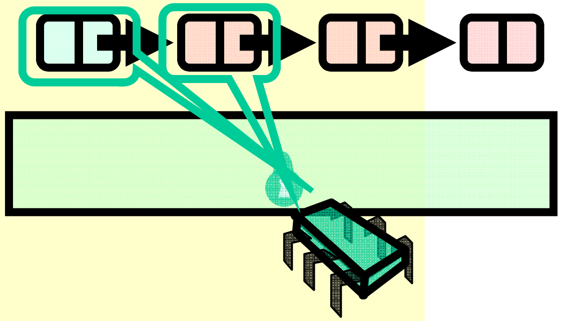




Remove: searching

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

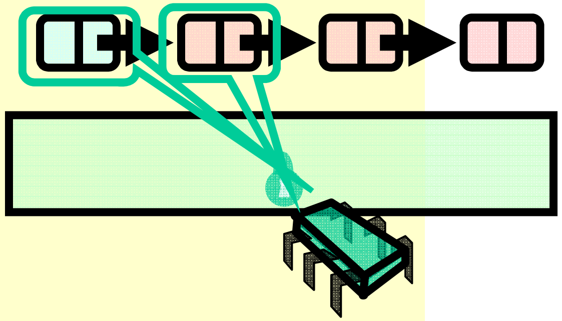
Examine predecessor and current nodes

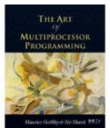


Remove: searching

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

Search by key

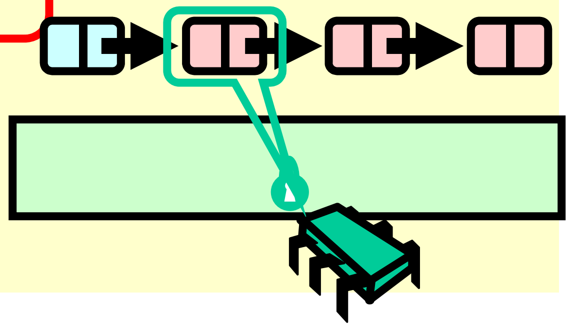




Remove: searching

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

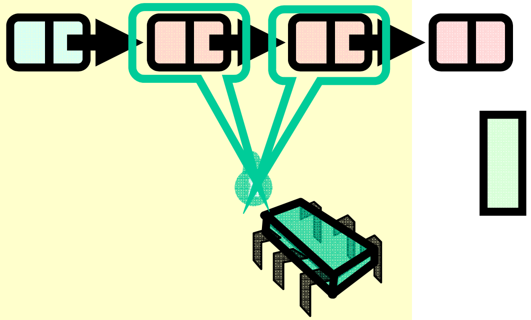
Stop if we find item

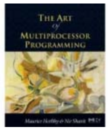


Remove: searching

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

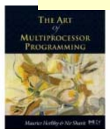
Move along





Remove Method

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



Remove Method

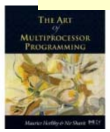
```
try {
  pred.lock(); curr.lock();
  if (validate(pred,curr) {
   if (curr.item == item) {
    pred.next = curr.next;
    return true;
   } else {
    return false;
   }}} finally {
    pred.unlock();
    curr.unlock();
   }}}
```

Always unlock

Remove Method

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

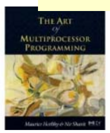
Lock both nodes



Remove Method

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

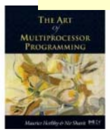
Check for synchronization conflicts



Remove Method

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

target found, remove node



Remove Method

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

target not found

3. Optimistic

3. Code review:

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();
```

Search the list from the beginning each time, until validation succeeds

Continued:

```
try {
        if (validate(pred, curr)) {
          if (curr.key == key) {
            return false;
          } else {
            Entry entry = new Entry(item);
            entry.next = curr;
            pred.next = entry;
            return true;
          }
        }
      } finally {
        pred.unlock(); curr.unlock();
      }
    }
  }
```

If validation succeeds Attempt Add

3. Optimistic

3. Code review:

Contains:

```
public boolean contains(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;
    }
    try {
      pred.lock(); curr.lock();
      if (validate(pred, curr)) {
        return (curr.key == key);
      }
    } finally {
      pred.unlock(); curr.unlock();
    }
  }
}
```

Search the list from the beginning each time, until validation succeeds

If validation succeeds Return the result

3. Optimistic

4. Methods properties:

- Assuming fair scheduler. Even if all the lock implementations are Starvation free. We will show a scenario in which the methods Remove / Add / Contains do not return.

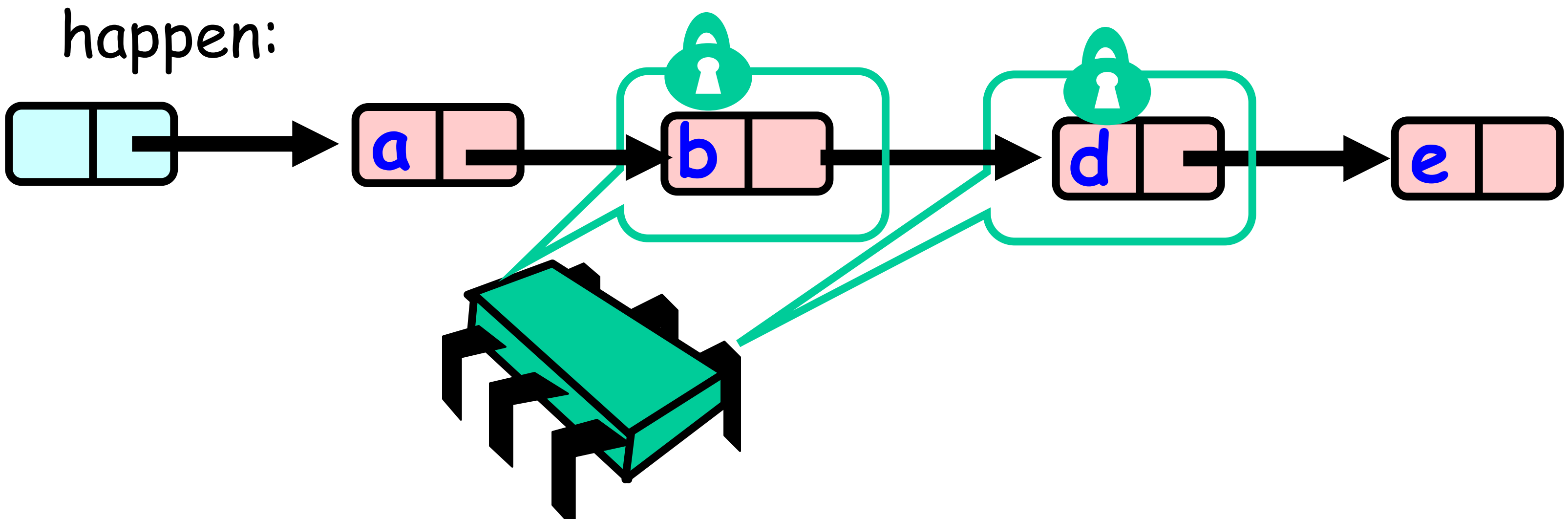
- And so our implementation won't be starvation free.

3. Optimistic

4. Methods properties:

- Assuming Thread A operation is Remove(d) / Add(c) / Contains(c).

- If the following sequence of operations will happen:



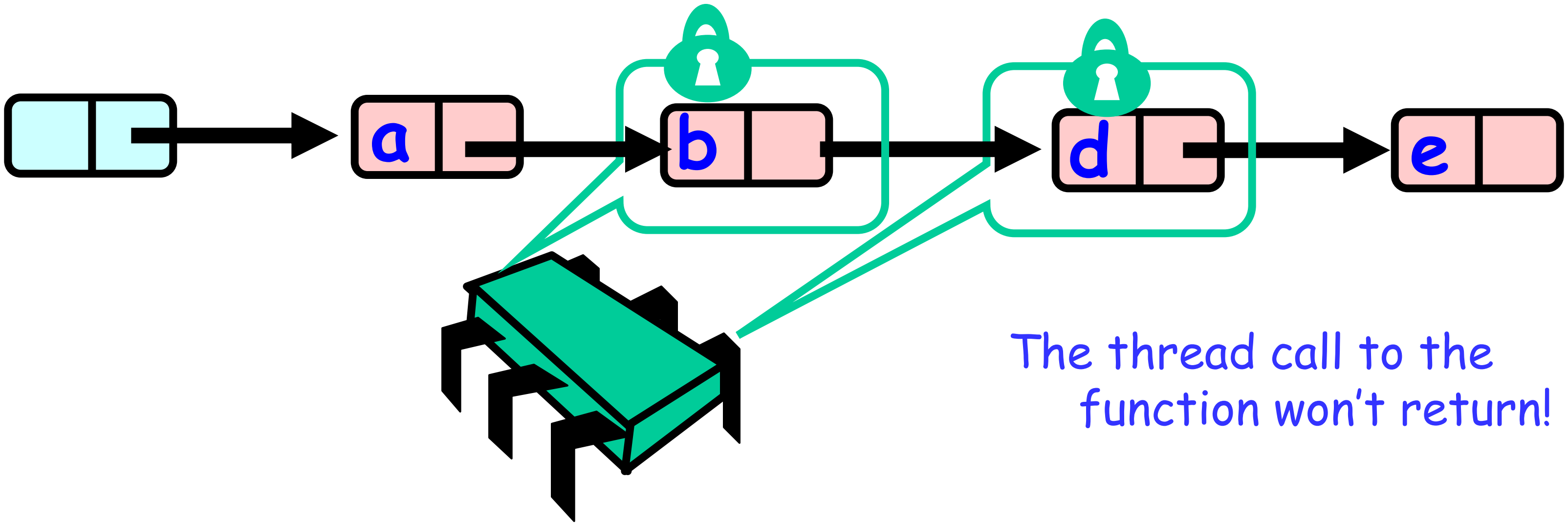
3. Optimistic

4. Methods properties:

The sequence:

- 1. Thread A will find b.
- 2. Thread B will remove b.
- 3. The validation of thread A will fail.
- 4. Thread C will add b.

   now go to 1.



The thread call to the function won't return!

3.Optimistic

5. Advantages / Disadvantages:

Advantages:

- Limited hot-spots
  - Targets of add(), remove(), contains().
  - No contention on traversals.
- Much less lock acquisition/releases.
- Better concurrency.

Disadvantages:

- Need to traverse list twice!
- Contains() method acquires locks.

3.Optimistic

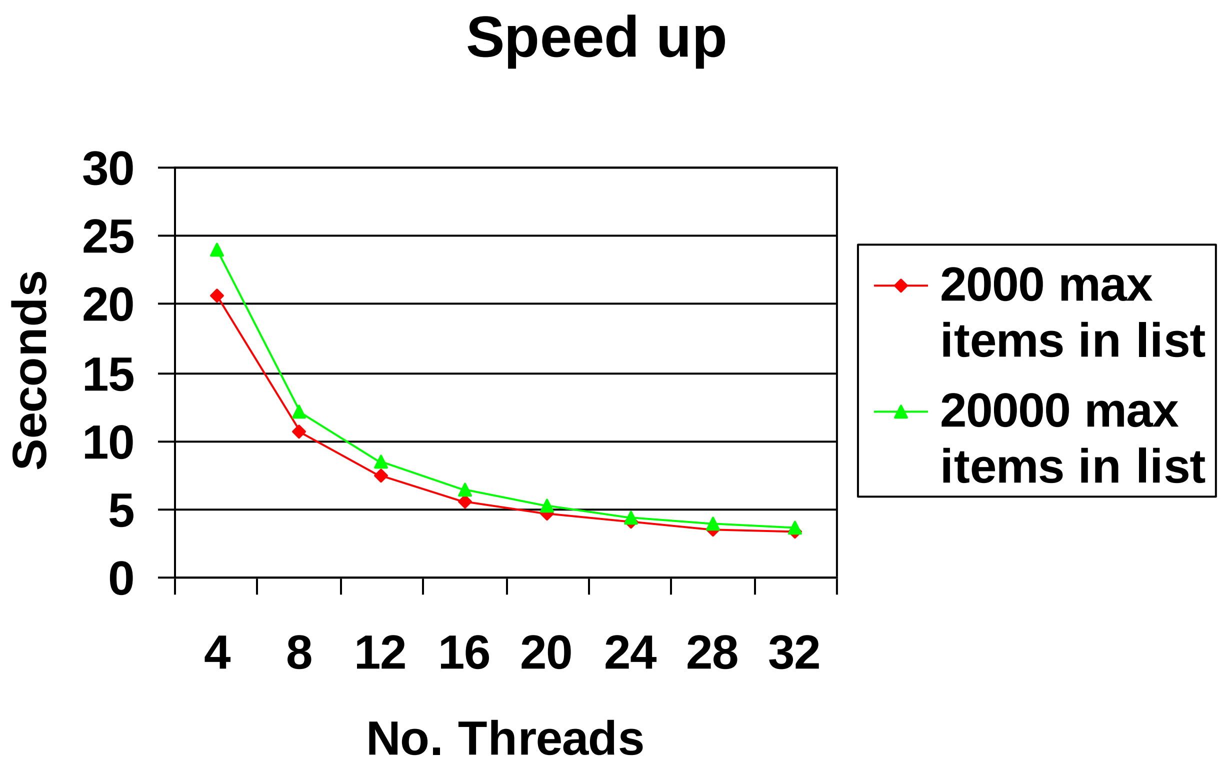
5. Advantages / Disadvantages:

- Optimistic is effective if:
  - The cost of scanning twice without locks is less than the cost of scanning once with locks

- Drawback:
  - Contains() acquires locks. Normally, about 90% of the calls are contains.

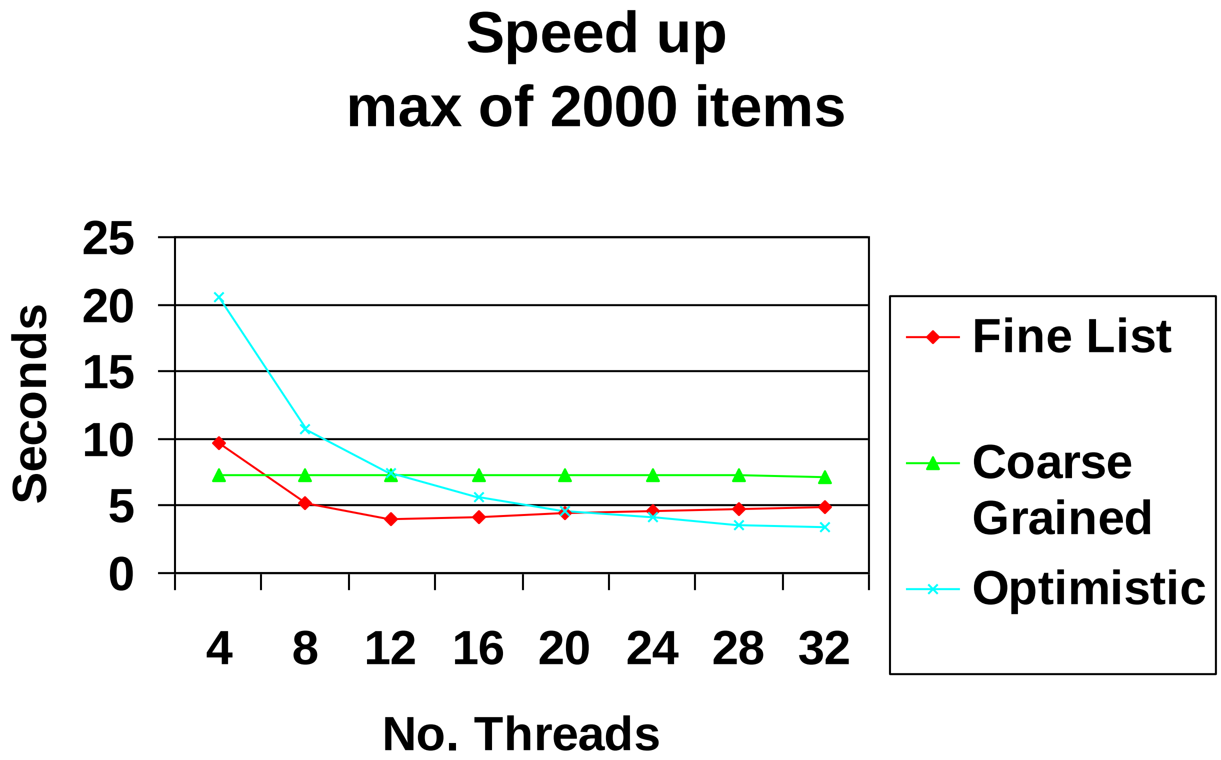
3. Optimistic

6. Running times:



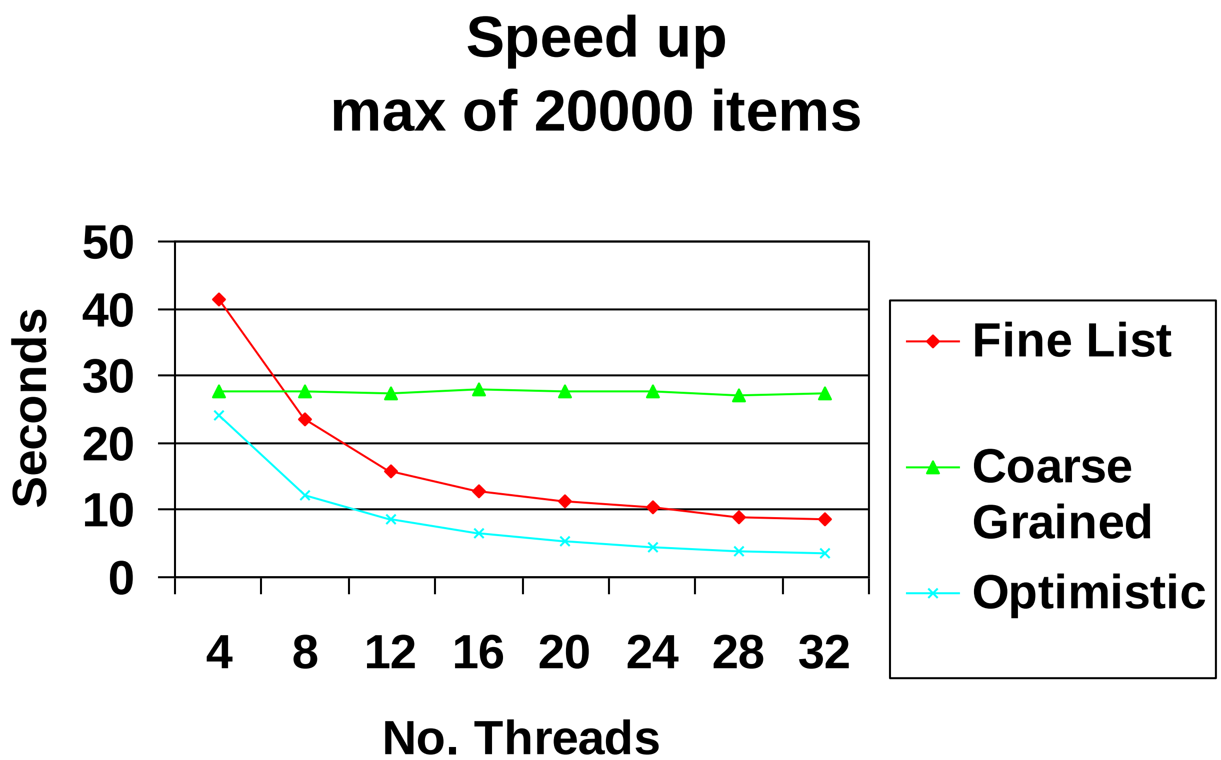
3. Optimistic

6. Running times:



3. Optimistic

6. Running times:



4. Lazy

1. Describing the algorithm:

    Validate:

    - Pred is not marked as deleted.
    - Curr is not marked as deleted.
    - Pred points to curr.

4. Lazy

1. Describing the algorithm:

Remove(x):

- Find the node to remove.
- Lock pred and curr.
- Validate. (New validation!)
- Logical delete
  - Marks current node as removed (new!).
- Physical delete
  - Redirects predecessor's next.

4. Lazy

1. Describing the algorithm:

   Add(x):

   - Find the node to remove.
   - Lock pred and curr.
   - Validate. (New validation!)
   - Physical add
     - The same as Optimistic.

4. Lazy

1. Describing the algorithm:

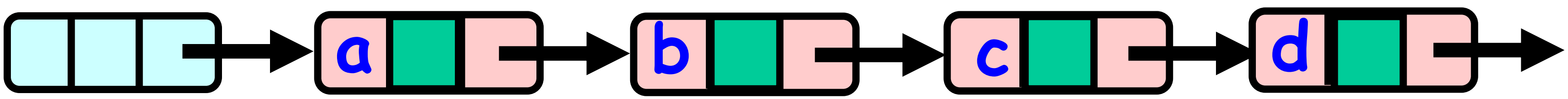
Contains(x):

- Find the node to remove without locking!
- Return true if found the node and it isn't marked as deleted.

- No locks!

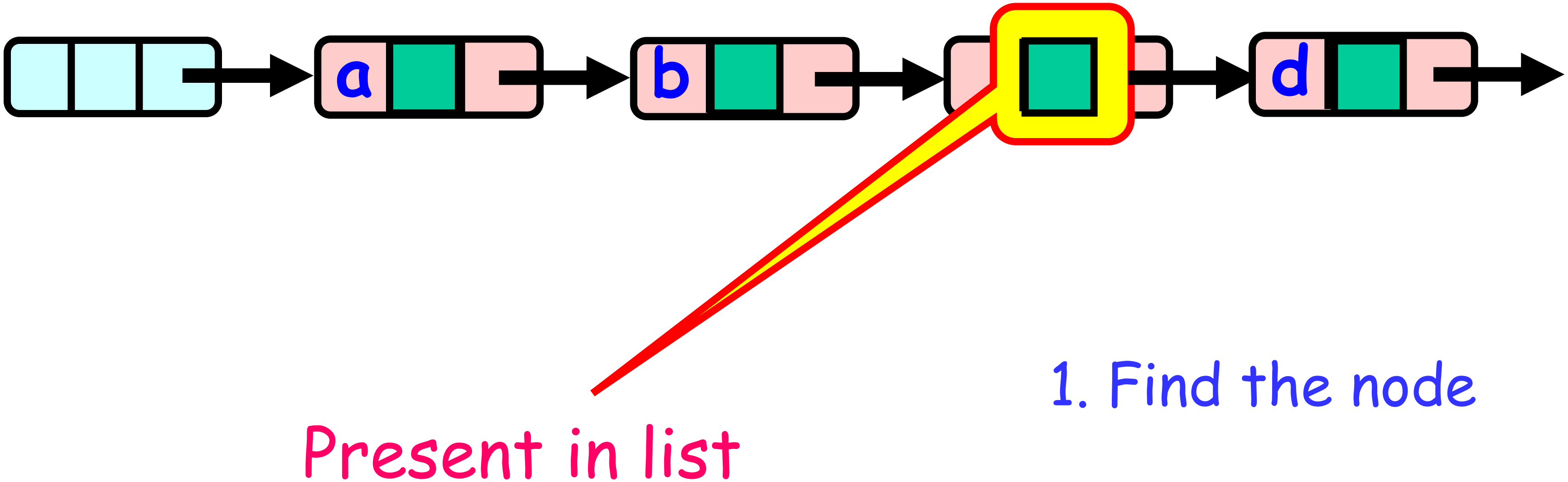
4. Lazy

1. Describing the algorithm:
   - Remove(c):



4. Lazy

1. Describing the algorithm:
    - Remove(c):

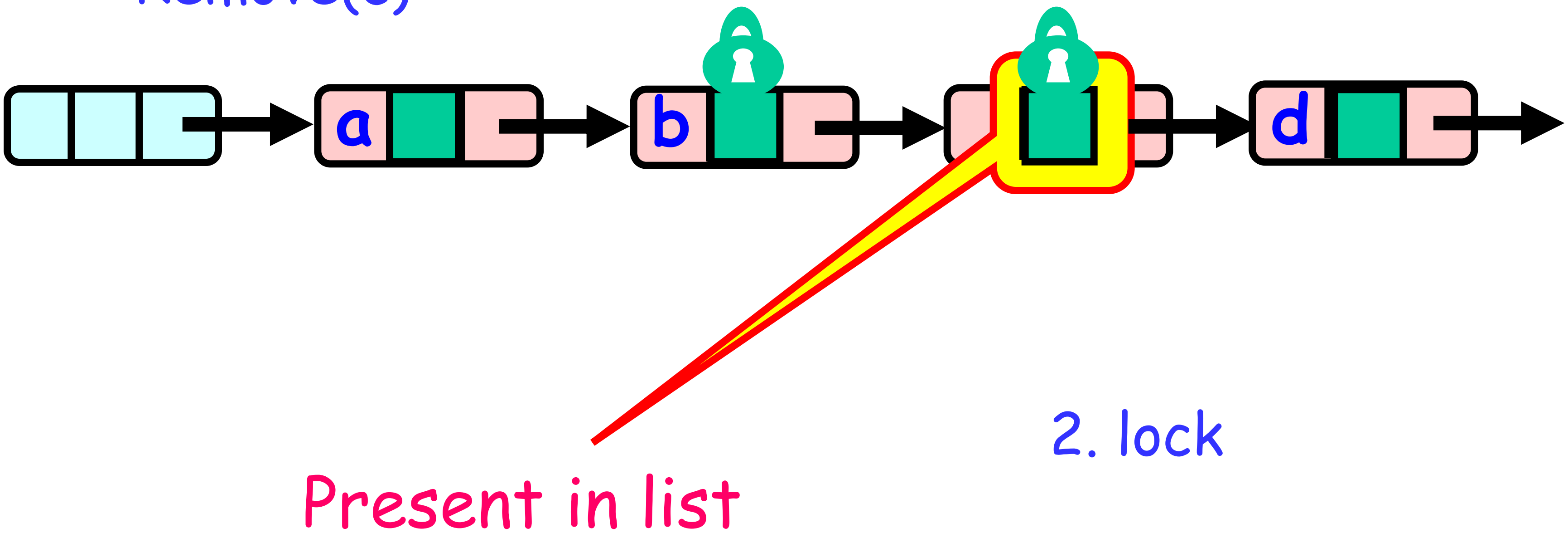


1. Find the node

4. Lazy

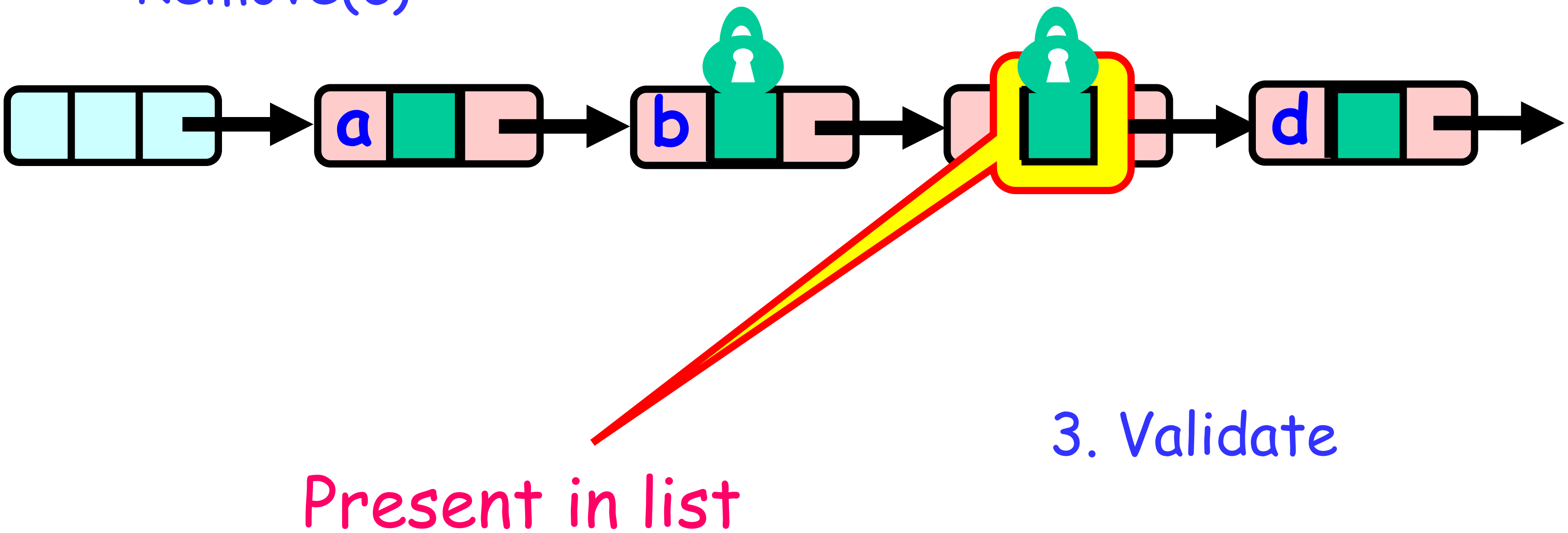
1. Describing the algorithm:

- Remove(c):



4. Lazy

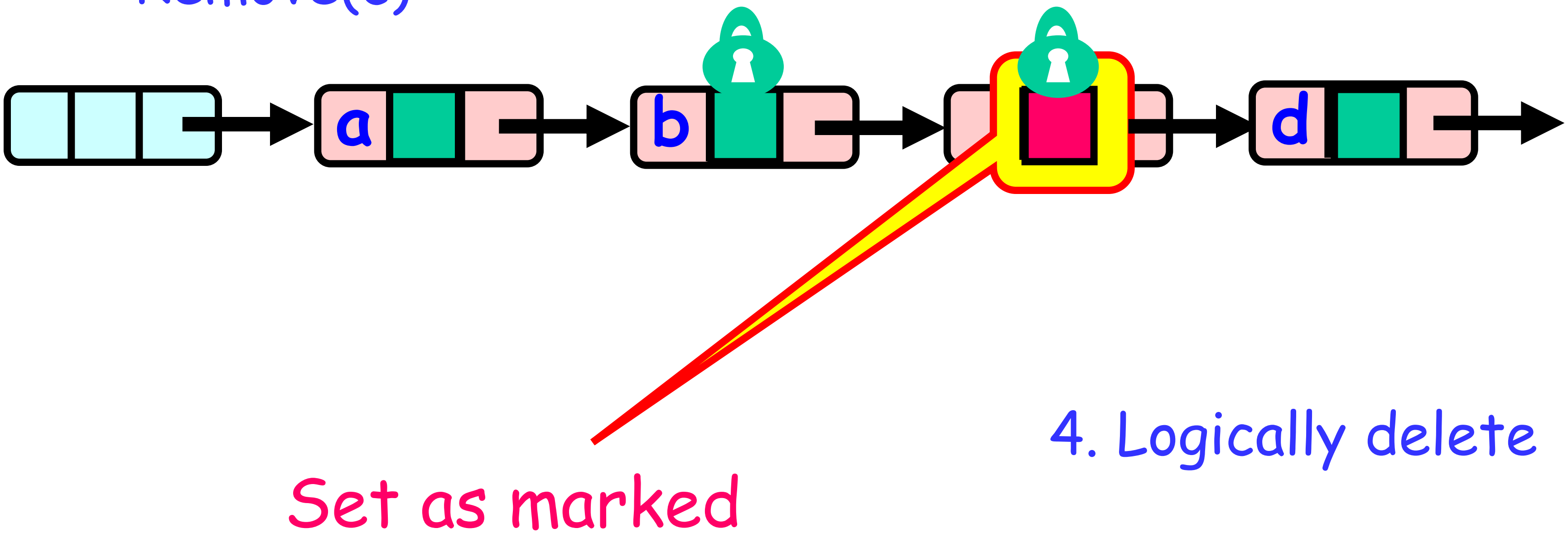
1. Describing the algorithm:
   - Remove(c):



3. Validate

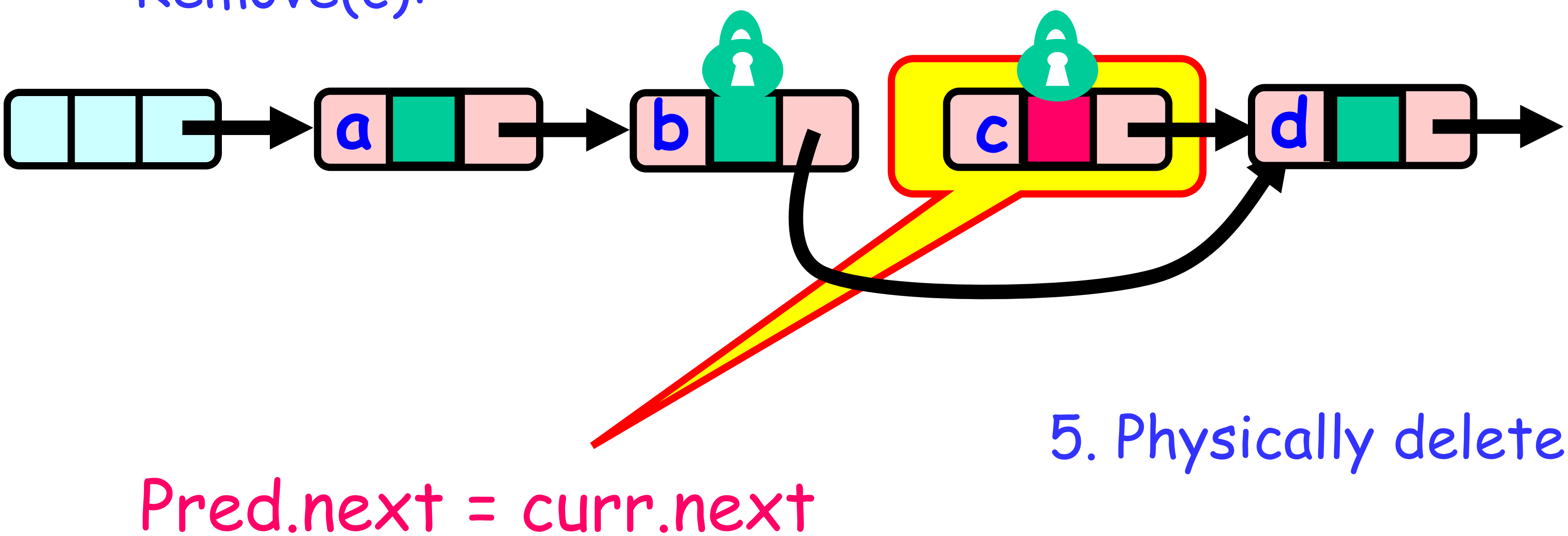
4. Lazy

1. Describing the algorithm:
   - Remove(c):



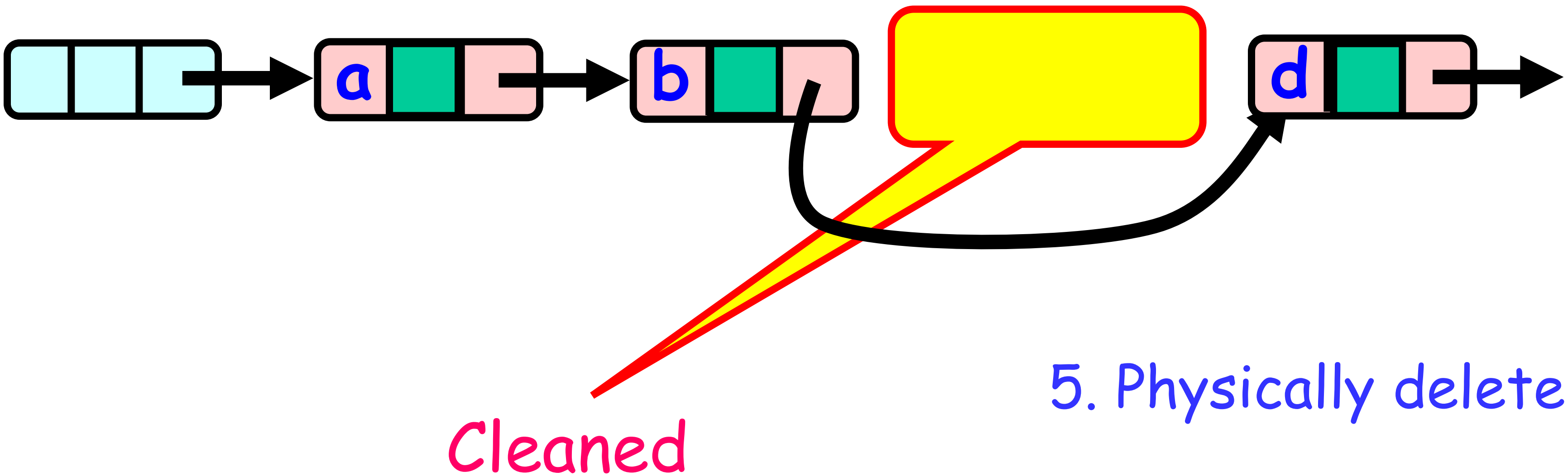
4. Lazy

1. Describing the algorithm:
   - Remove(c):



4. Lazy

1. Describing the algorithm:
    - Remove(c):



4. Lazy

1. Describing the algorithm:

Given the Lazy Synchronization algorithm.

What else should we change?

4. Lazy

1. Describing the algorithm:

- New Abstraction map!

- S(head) =
  - { x | there exists node a such that
    - a reachable from head and
    - a.item = x and
    - a is unmarked
  - }

4. Lazy

2. Explaining why every step is needed.

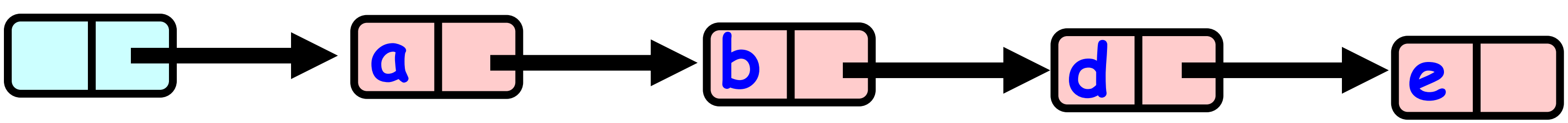
Why do we need to Validate?

4. Lazy

2. Explaining why every step is needed.

- First: Why do we need to validate that pred Still points to curr?

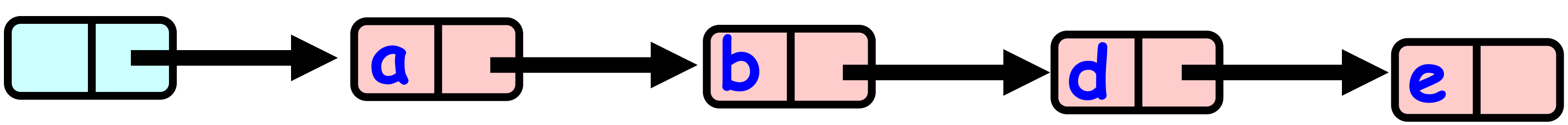
- The same as in Optimistic:
- Thread A removes(d).
- Then thread A found b, before A locks. Another thread adds(c).
  - c and d will be removed instead of just d.



4. Lazy

2. Explaining why every step is needed.

- Second: Why do we need to validate that pred and curr aren't marked logically removed?
- To make sure a thread hasn't removed them between our find and our lock.
- The same scenario we showed for validating that pred is still accessible from head holds here:
  - After thread A found b, before A locks. Another thread removes b. (our operation won't take place).



4. Lazy

3. Code review:

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();
```

Search the list from the beginning each time, until validation succeeds

Continued:

```
          try {
            if (validate(pred, curr)) {
              if (curr.key == key) {
                return false;
              } else {
                Node Node = new Node(item);
                Node.next = curr;
                pred.next = Node;
                return true;
              }
            }
          } finally {
            curr.unlock();
          }
        } finally {
          pred.unlock();
        }
      }
    }
```

If validation succeeds Attempt Add

4. Lazy

3. Code review:

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 {
```

Search the list from the beginning each time, until validation succeeds

Continued:

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

Validation

Logically remove

Physically remove

4. Lazy

3. Code review:

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;
}
```

No Lock!

Check if its there
and not marked

4. Lazy

4. Methods properties:

Remove and Add:

- Assuming fair scheduler. Even if all the lock implementations are Starvation free. The same scenario we showed for optimistic holds here.
- (only here the validation will fail because the node will be marked and not because it can't be reached from head)

- And so our implementation won't be starvation free.

4. Lazy

4. Methods properties:

But… Contains:

- Contains does not lock!
- In fact it isn't dependent on other threads to work.
- And so… Contains is Wait-free.
- Do notice that other threads can't increase the list forever while the thread is in contains because we have a maximum size to the list (<tail).

4. Lazy

5. Advantages / Disadvantages:

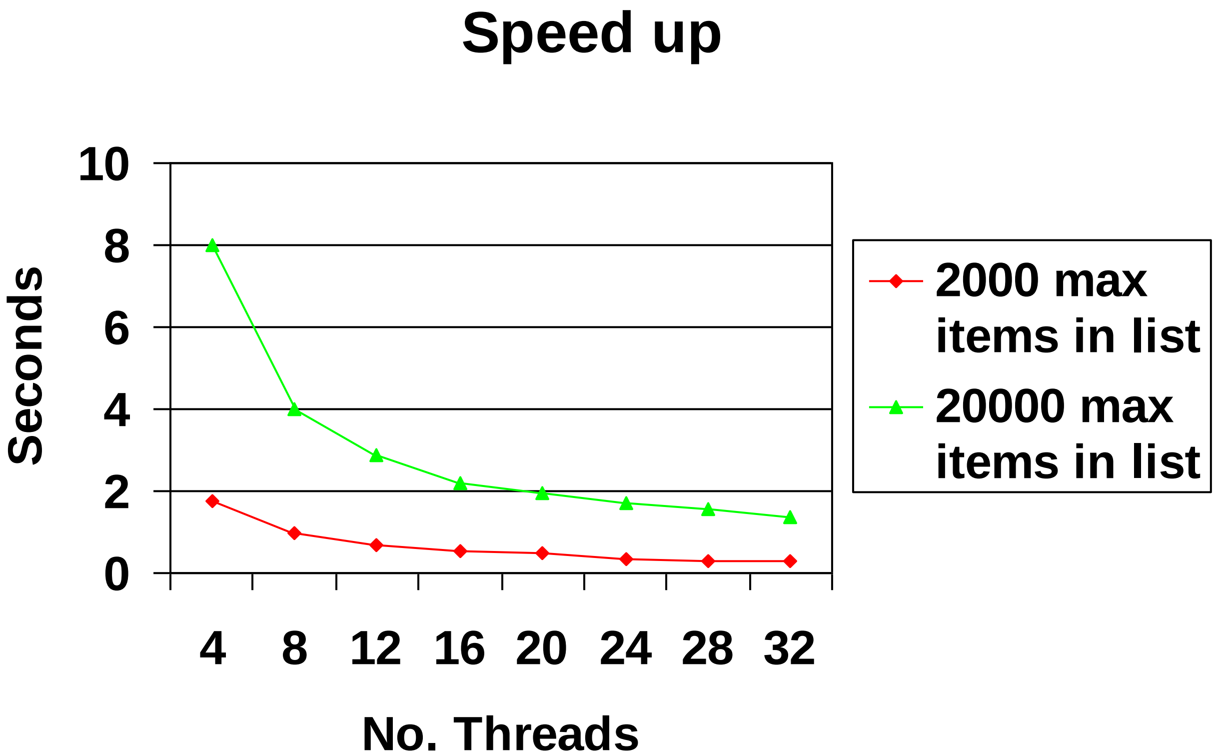
- Advantages:
  - Contains is Wait-free. Usually 90% of the calls!
  - Validation doesn't rescan the list.

- Drawbacks:
  - Failure to validate restarts the function call.
  - Add and Remove use locks.

Lock-free implementation

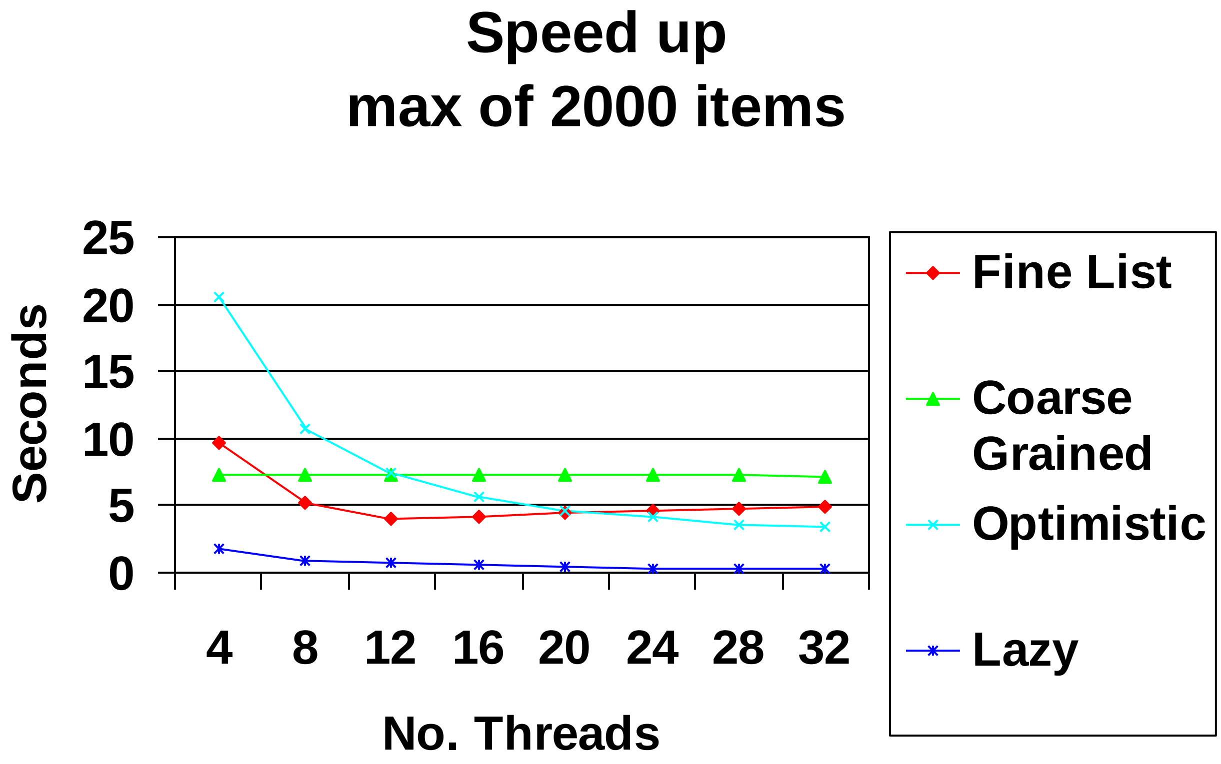
4. Lazy

6. Running times:



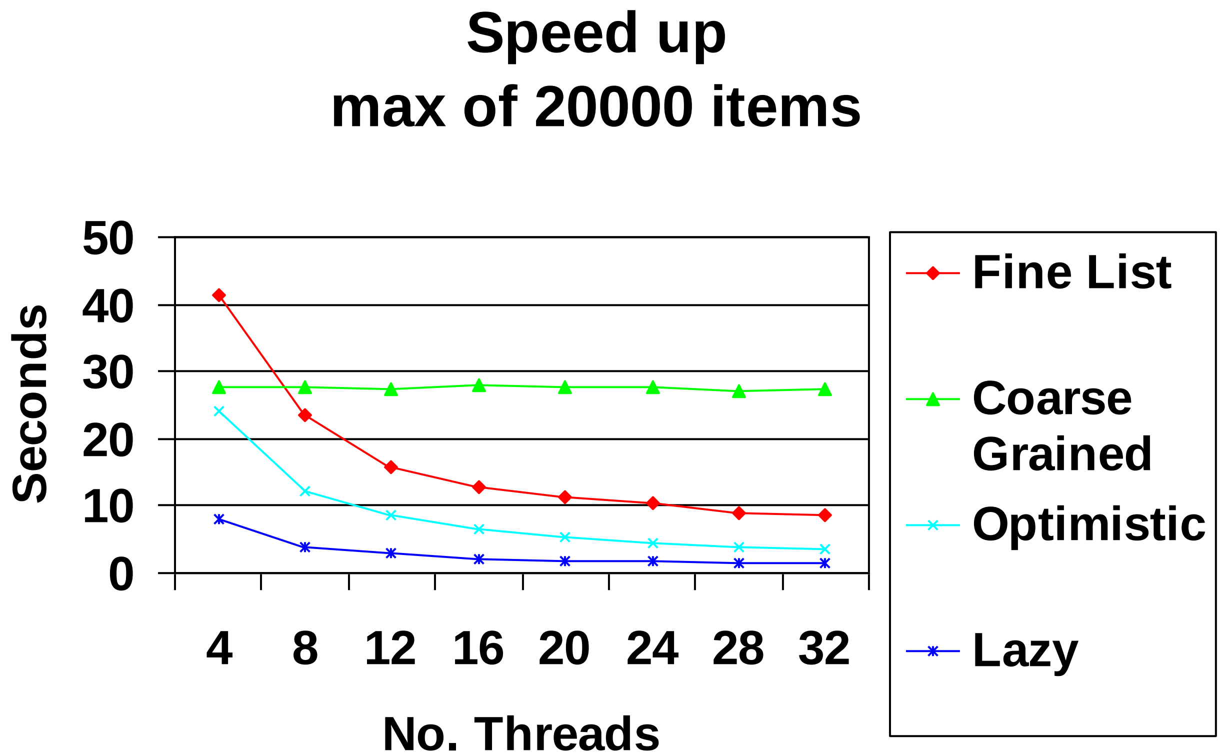
4. Lazy

6. Running times:



4. Lazy

6. Running times:



Optimistic lock-free Concurrency

CAS(&x,a,b) = if *x = a then *x = b return true else return false

Pessimistic	Optimistic

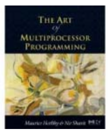
lock x;
  x++;
unlock x;
	int t;
	do {
	    t = x;
	} while (!CAS(&x, t, t+1))

Reminder: Lock-Free Data Structures

- No matter what …
  - Guarantees minimal progress in any 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

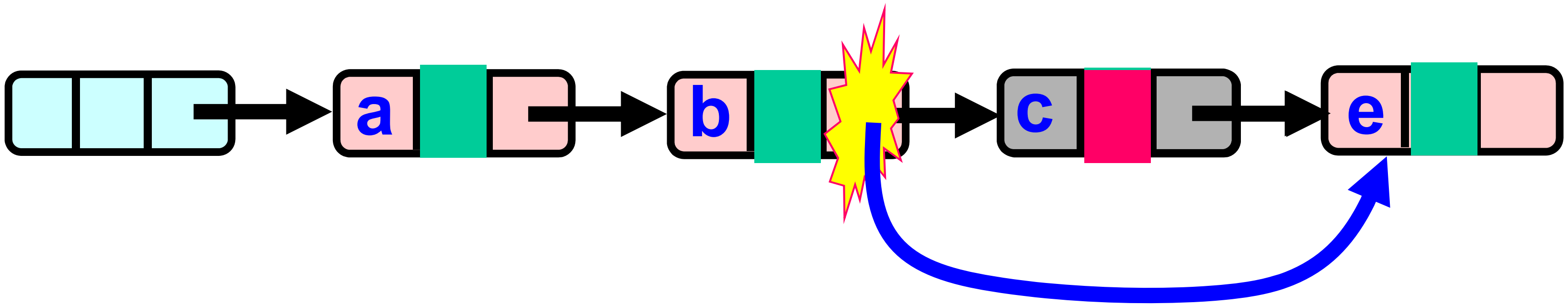
Lock-free Lists

- Next logical step
  – Wait-free contains()
  – lock-free add() and remove()
- Use only compareAndSet()
  – What could go wrong?



Lock-free Lists

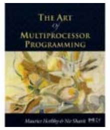
Logical Removal



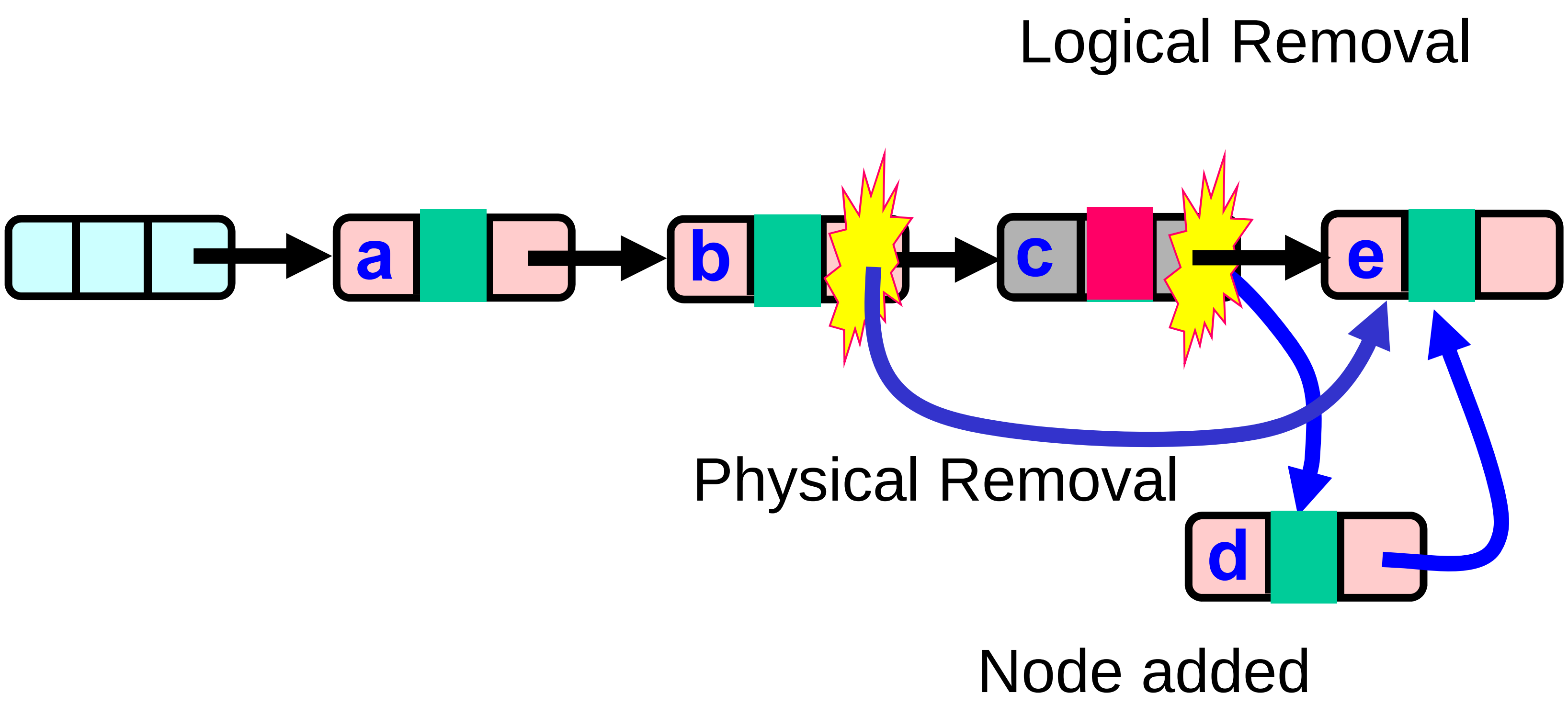
Physical Removal

Use CAS to verify pointer is correct

Not enough!

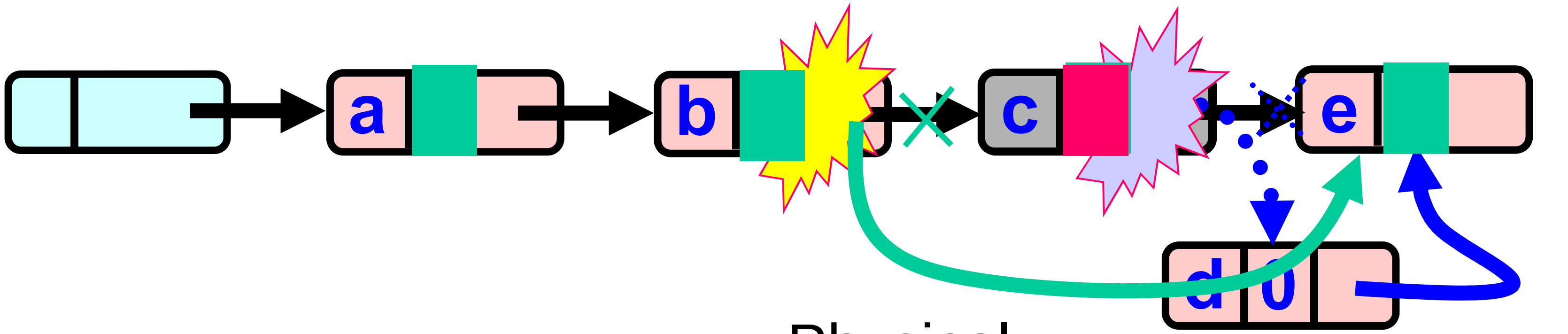


Problem…



The Solution: Combine Bit and Pointer

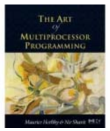
Logical Removal = Set Mark Bit



Physical Removal CAS

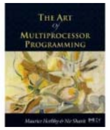
Fail CAS: Node not added after logical Removal

Mark-Bit and Pointer are CASed together (AtomicMarkableReference)



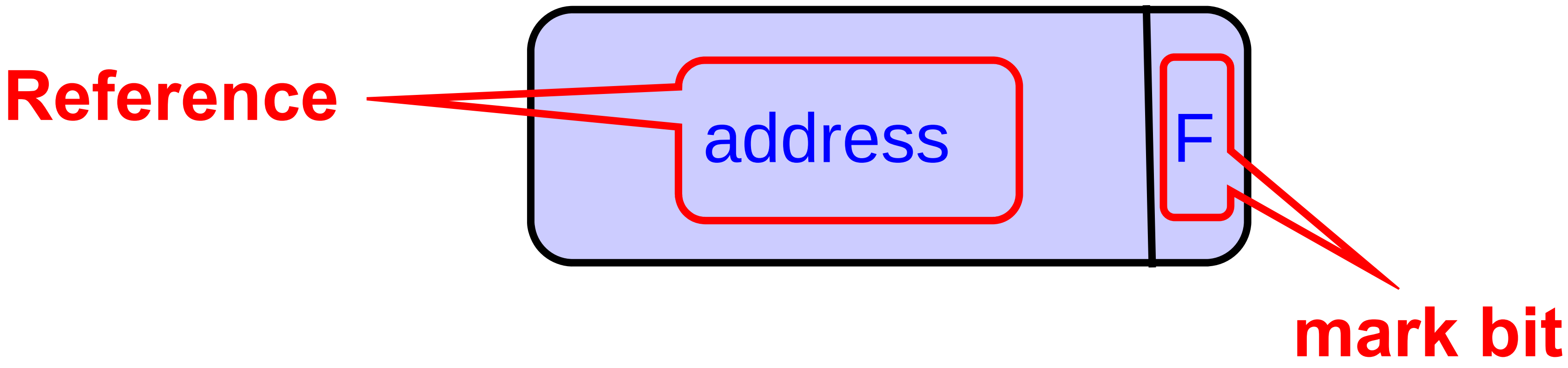
Solution

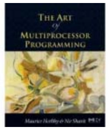
- Use AtomicMarkableReference
- Atomically
  - Swing reference and
  - Update flag
- Remove in two steps
  - Set mark bit in next field
  - Redirect predecessor's pointer



Marking a Node

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





Extracting Reference & Mark

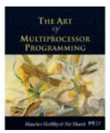
```
Public Object get(boolean[] marked);
```

Extracting Reference & Mark

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

Returns reference

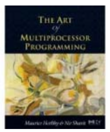
Returns mark at array index 0!



Extracting Mark Only

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

Value of mark



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

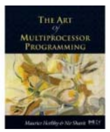
…then change to this new reference …

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

… and this new mark

Changing State

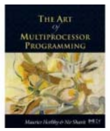
```
public boolean attemptMark(
  Object expectedRef,
  boolean updateMark);
```



Changing State

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

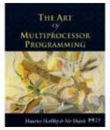
If this is the current reference …



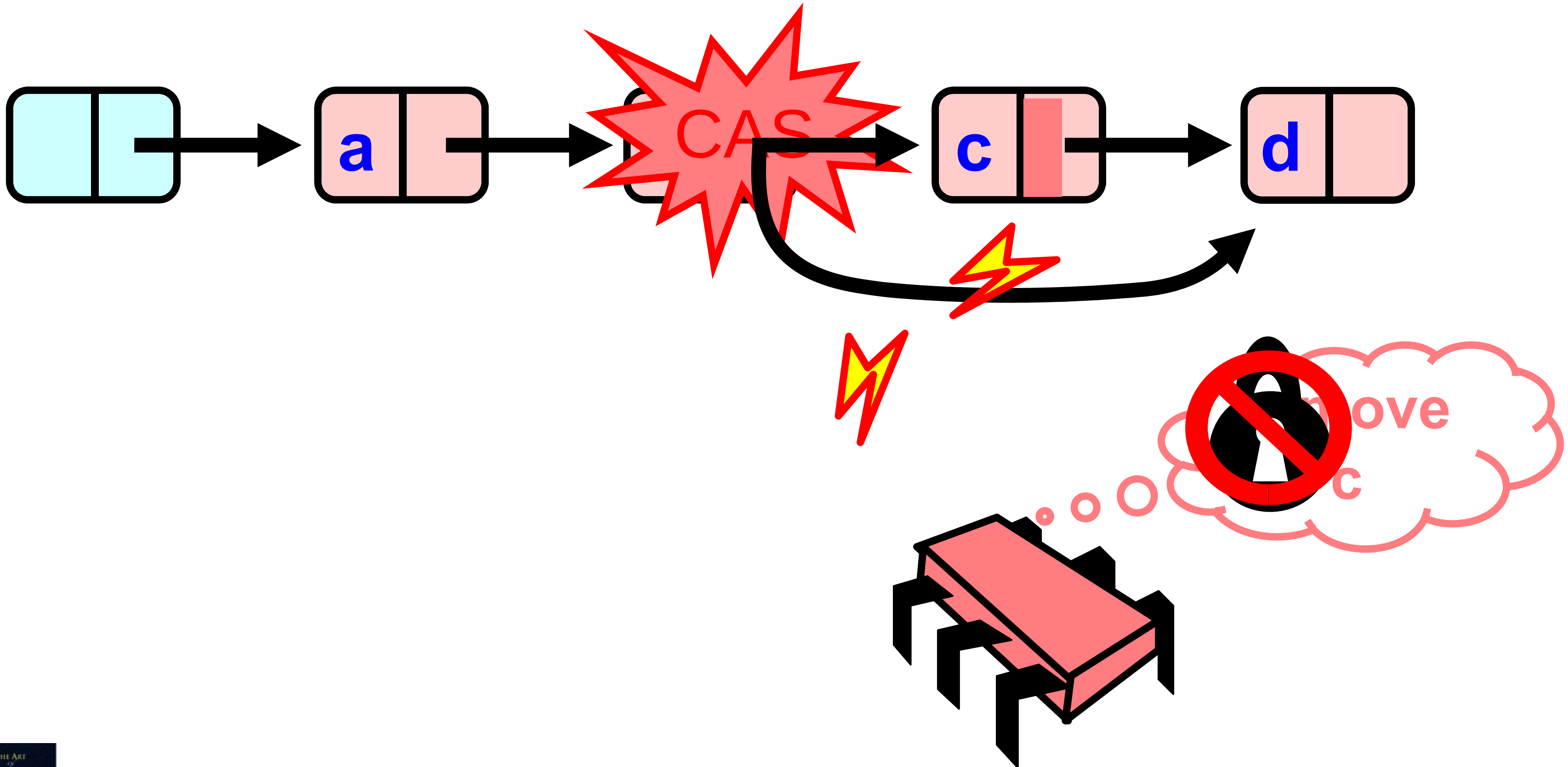
Changing State

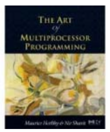
```
public boolean attemptMark(
  Object expectedRef,
  boolean updateMark);
```

.. then change to this new mark.

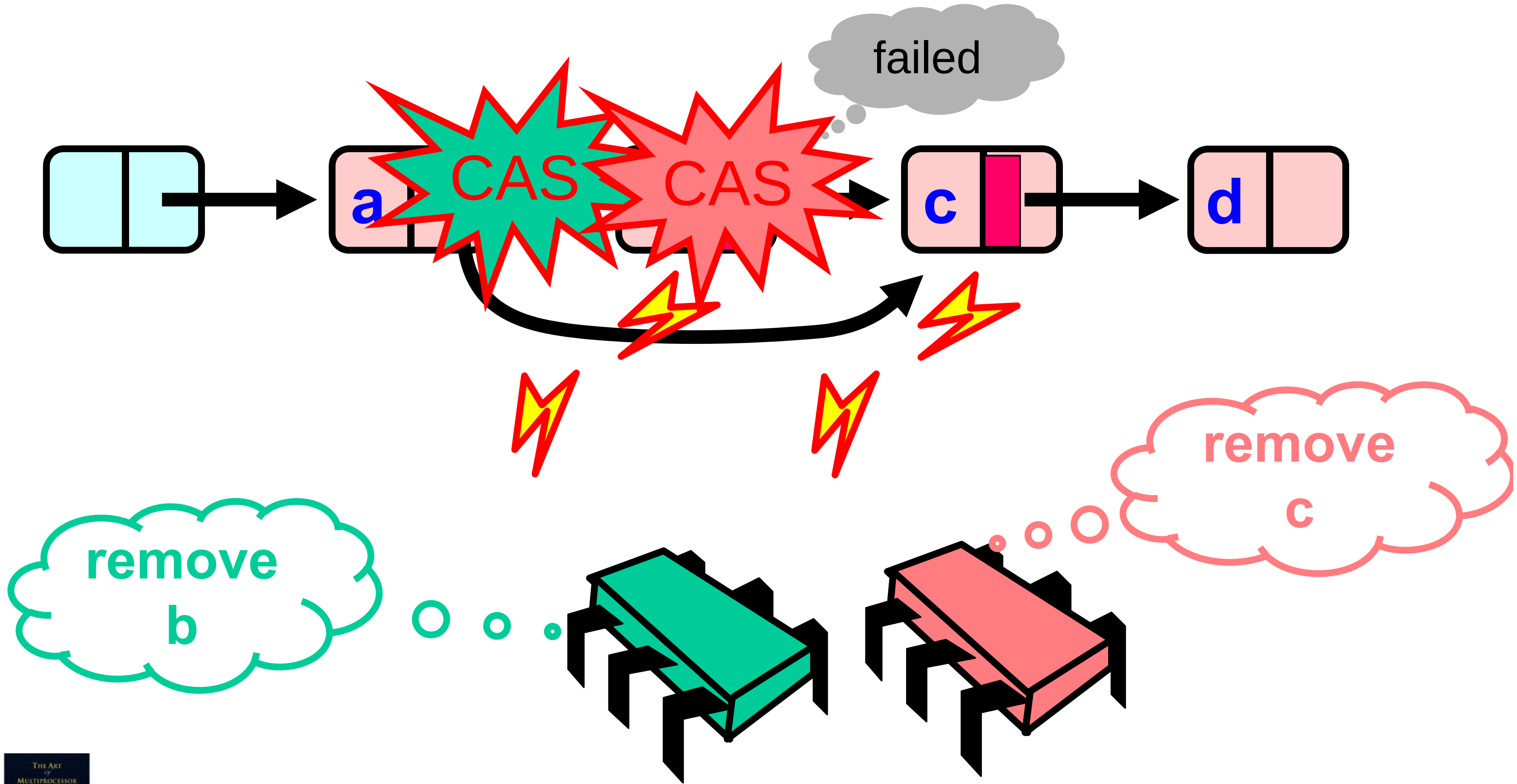


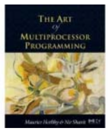
Removing a Node



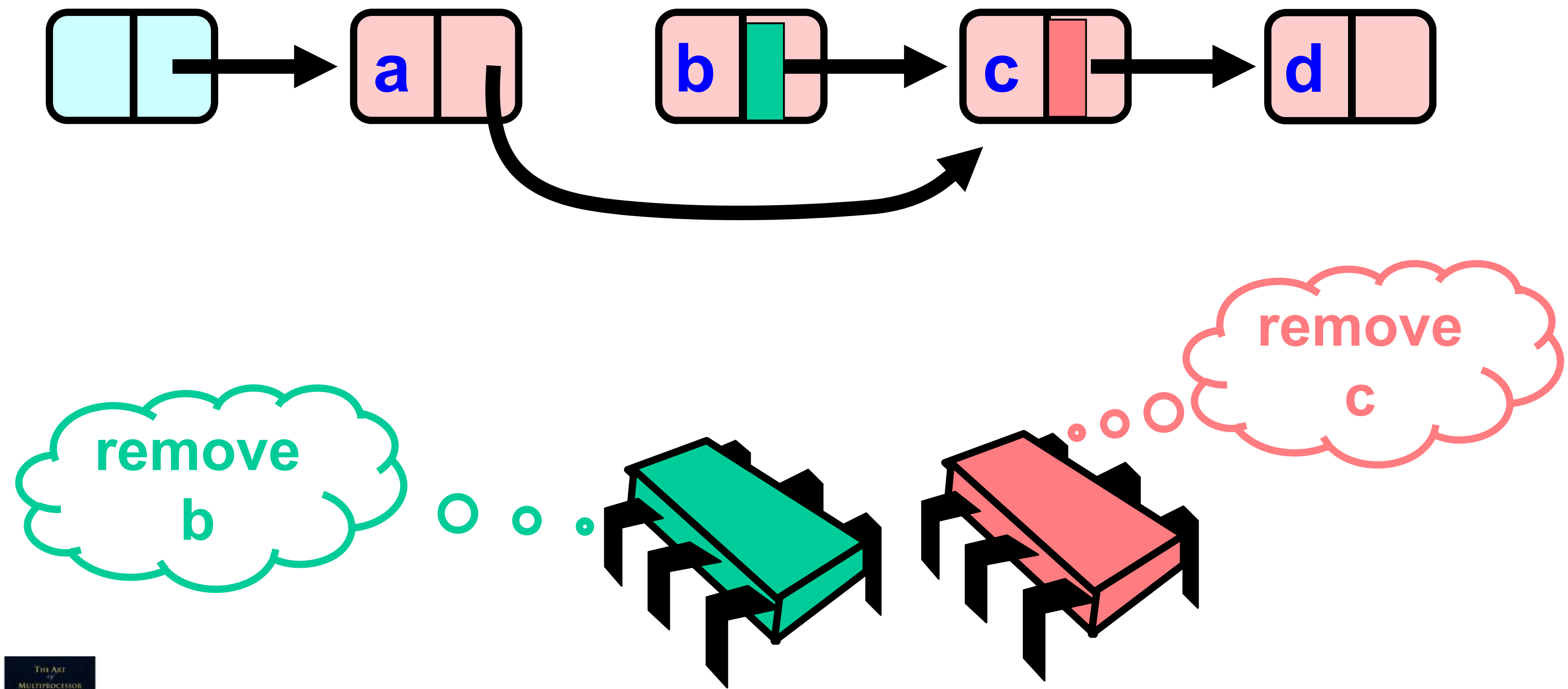


Removing a Node

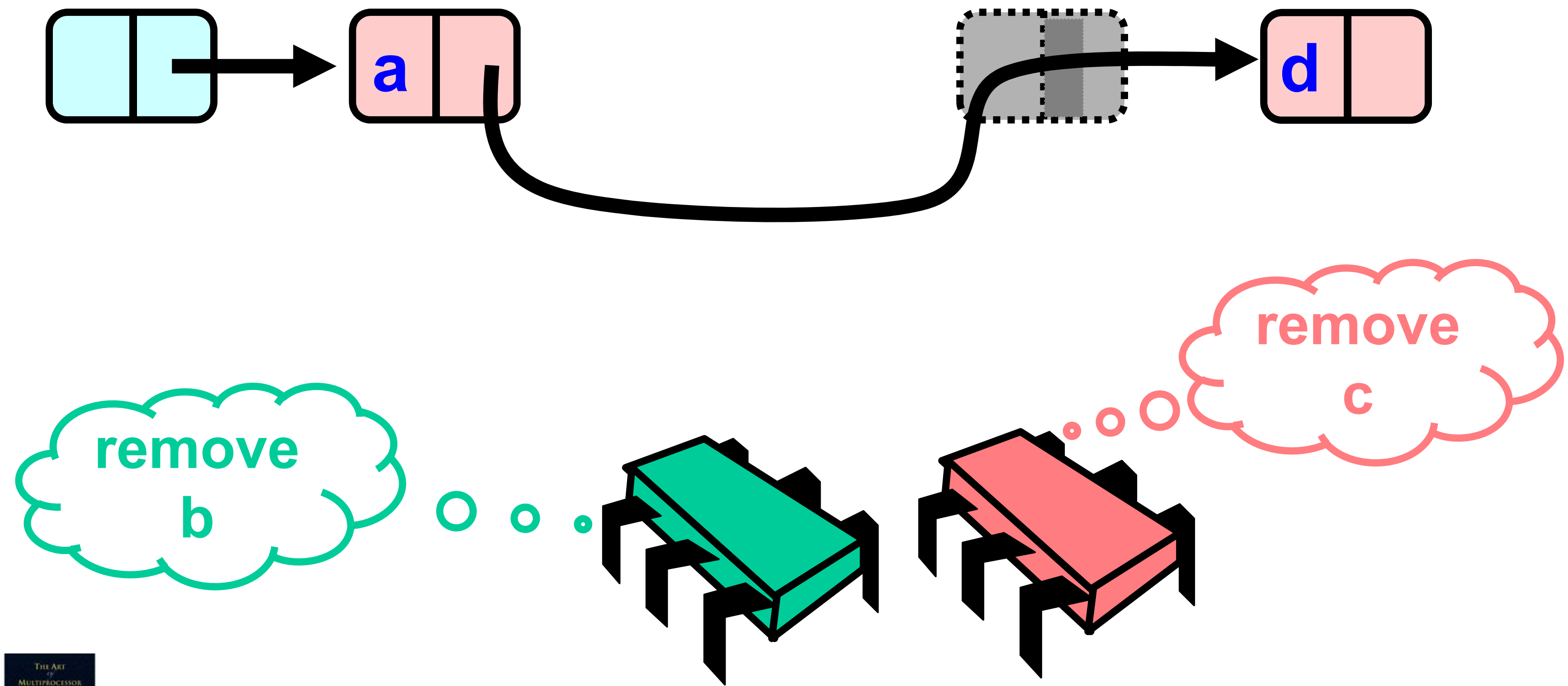


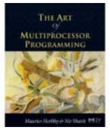


Removing a Node



Removing a Node

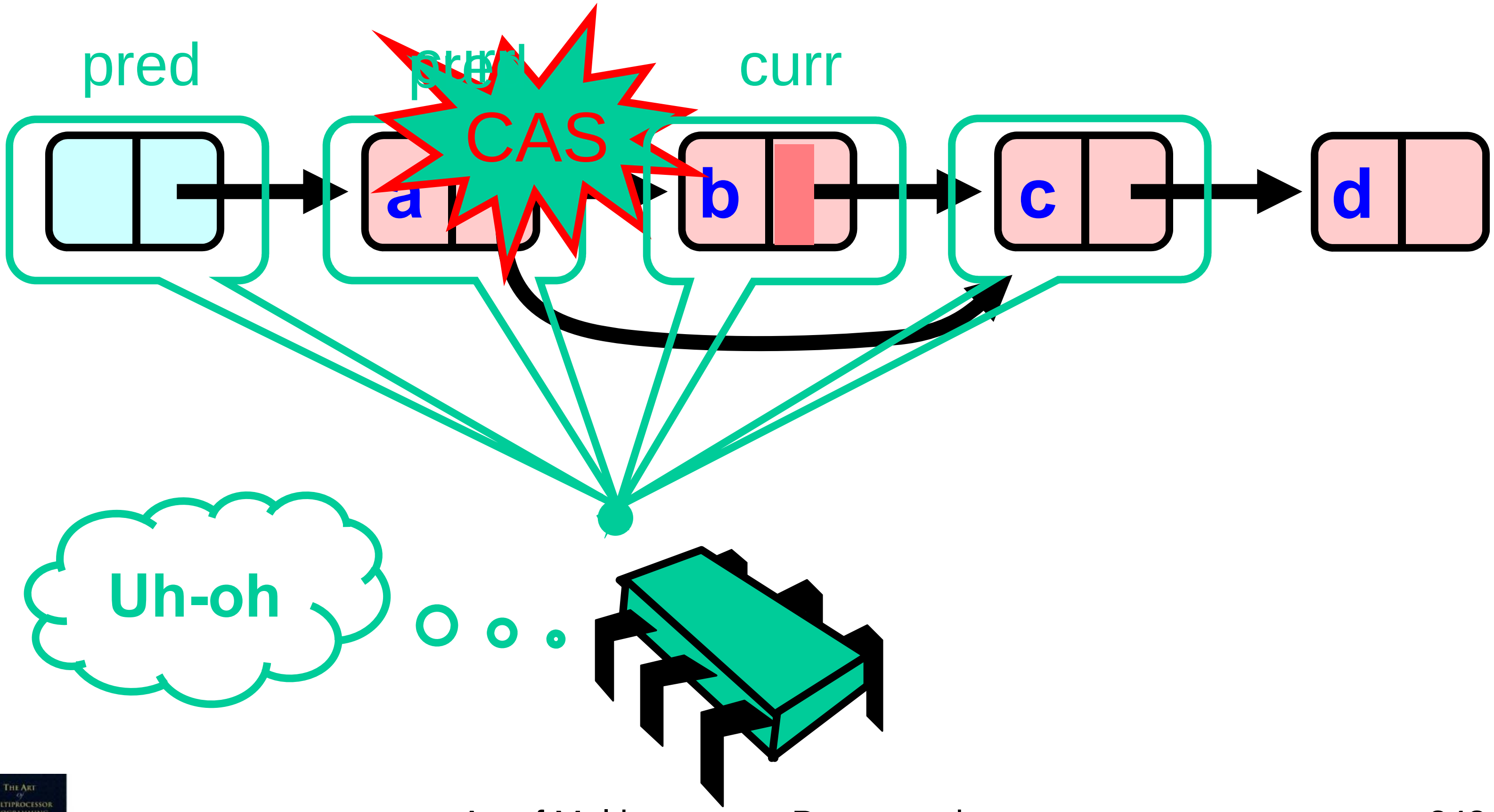


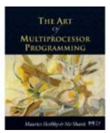


Traversing the List

- 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)





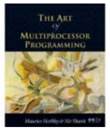
The Window Class

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

The Window Class

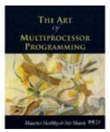
```
class Window {
 public Node pred;
 public Node curr;
 Window(Node pred, Node curr) {
   this.pred = pred; this.curr = curr;
 }
}
```

A container for pred and current values



Using the Find Method

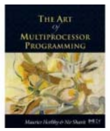
```
Window window = find(head, key);
Node pred = window.pred;
curr = window.curr;
```



Using the Find Method

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

Find returns window



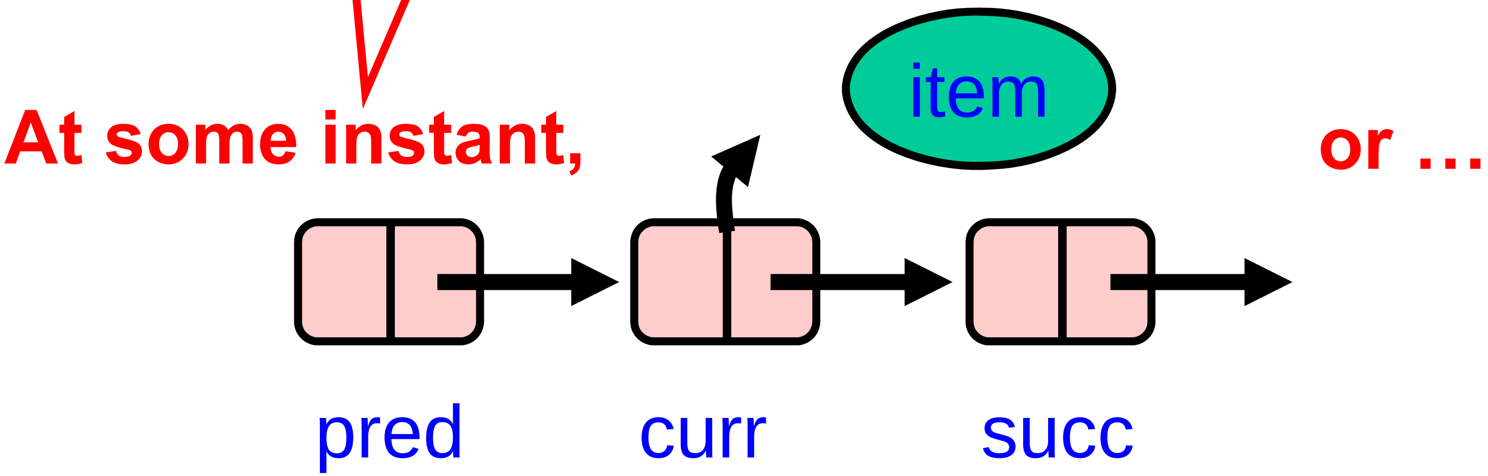
Using the Find Method

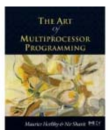
```
Window window = find(head, key);
Node pred = window.pred;
curr = window.curr;
```

Extract pred and curr

The Find Method

```
Window window = find(item);
```

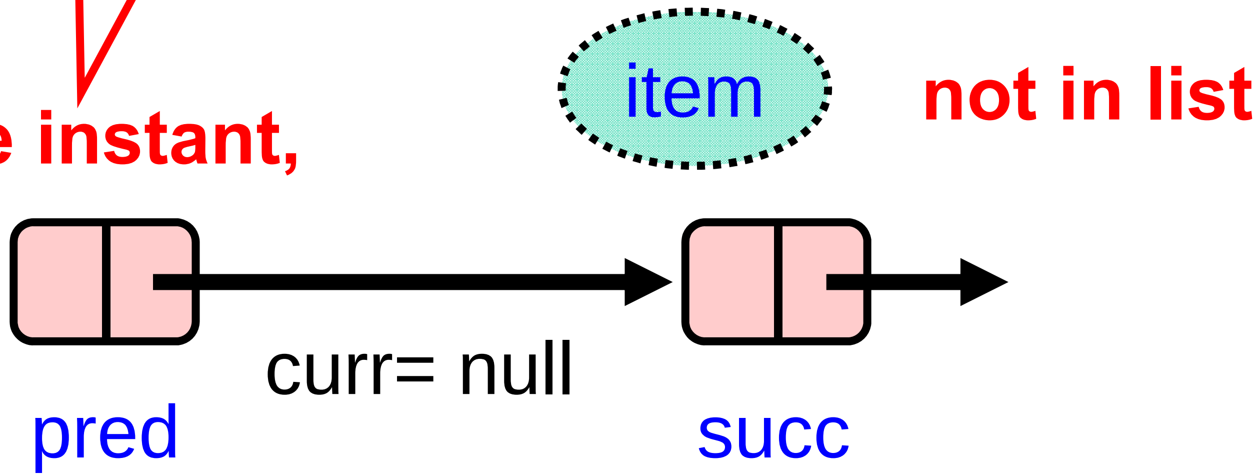


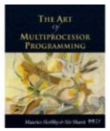


The Find Method

```
Window window = find(item);
```

At some instant,





Remove

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

Remove

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

Keep trying

Remove

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

Find neighbors

Remove

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

She’s not there …

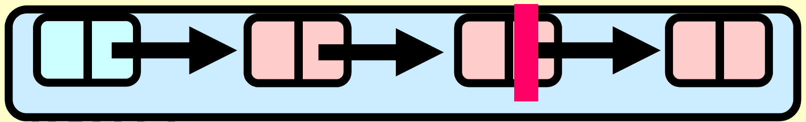
Remove

Try to mark node as deleted

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

Remove

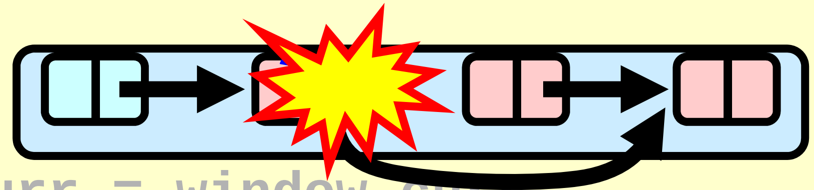
If it doesn't work, just retry, if it does, job essentially done



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

Remove

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



Try to advance reference
(if we don't succeed, someone else did or will).

Add

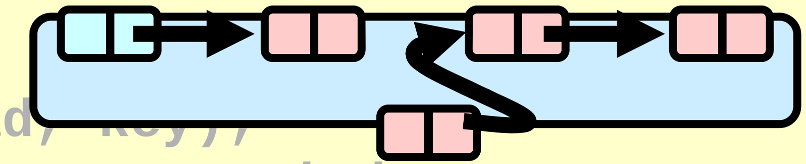
```
public boolean add(T item) {
 boolean splice;
 while (true) {
   Window window = find(head, key);
   Node pred = window.pred, curr = window.curr;
   if (curr.key == key) {
      return false;
   } else {
   Node node = new Node(item);
   node.next = new AtomicMarkableRef(curr, false);
   if (pred.next.compareAndSet(curr, node, false,
false)) {return true;}
}}}
```

Add

```
public boolean add(T item) {
 boolean splice;
 while (true) {
   Window window = find(head, key);
   Node pred = window.pred, curr = window.curr;
   if (curr.key == key) {
      return false;
   } else {
   Node node = new Node(item);
   node.next = new AtomicMarkableRef(curr, false);
   if (pred.next.compareAndSet(curr, node, false,
false)) {return true;}
}}}
```

Item already there.

Add

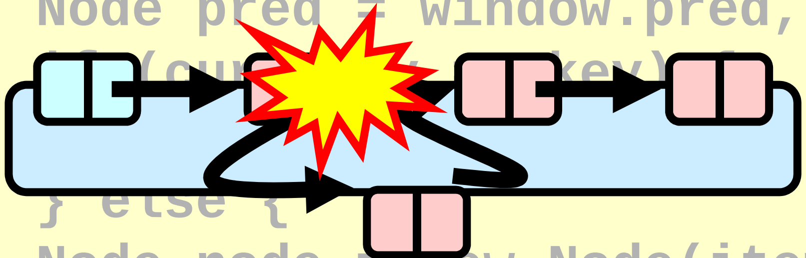


```
public boolean add(T item) {
 boolean splice;
 while (true) {
   Window window = find(head, key);
   Node pred = window.pred, curr = window.curr;
   if (curr.key == key) {
      return false;
   } else {
   Node node = new Node(item);
   node.next = new AtomicMarkableRef(curr, false);
   if (pred.next.compareAndSet(curr, node, false,
false)) {return true;}
}}}
```

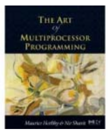
create new node

Add

```
public boolean add(T item) {
 boolean splice;
 while (true) {
   Window window = find(head, key);
   Node pred = window.pred, curr = window.curr;
   if (curr.key == key) {
   } else {
   Node node = new Node(item);
   node.next = new AtomicMarkableRef(curr, false);
   if (pred.next.compareAndSet(curr, node, false, 
false)) {return true;}
}}}
```



Install new node, else retry loop



Wait-free Contains

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

Wait-free Contains

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

Only diff is that we get and check marked

Lock-free Find

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

Lock-free Find

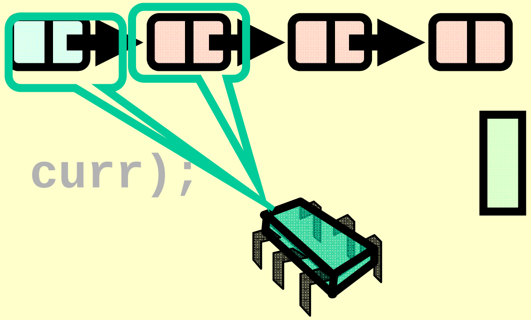
```
public Window find(Node head, int key) {
 Node pred = null, curr = null, succ = null;
 boolean[] marked = {false}; boolean snip;
 retry: while (true) {
   pred = head;
   curr = pred.next.getReference();
   while (true) {
    succ = curr.next.get(marked);
    while (marked[0]) {
    …
    }
    if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

If list changes while traversed, start over

Lock-free Find

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

Start looking from head



Lock-free Find

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

Move down the list

Lock-free Find

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

Get ref to successor and current deleted bit

Lock-free Find

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

Try to remove deleted nodes in path…code details soon

Lock-free Find

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

If curr key that is greater or equal, return pred and curr

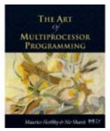
Lock-free Find

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

Otherwise advance window and loop again

Lock-free Find

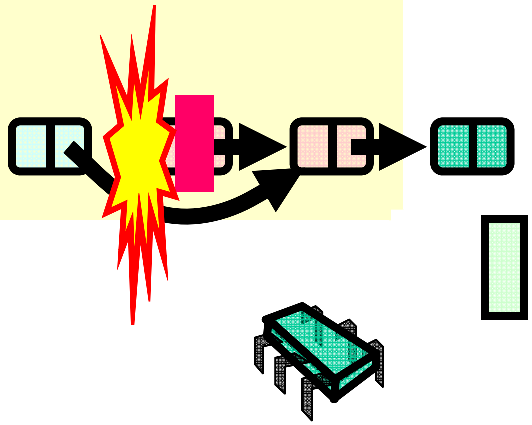
```
retry: while (true) {
   …
   while (marked[0]) {
     snip = pred.next.compareAndSet(curr,
                                     succ, false, false);
     if (!snip) continue retry;
     curr = succ;
     succ = curr.next.get(marked);
   }
…
```



Lock-free Find

Try to snip out node

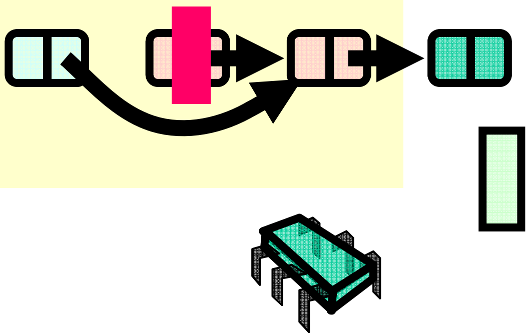
```
retry: while (true) {
   …
   while (marked[0]) {
     snip = pred.next.compareAndSet(curr,
                                     succ, false, false);
     if (!snip) continue retry;
     curr = succ;
     succ = curr.next.get(marked);
   }
…
```



Lock-free Find

if predecessor's next field changed, retry whole traversal

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



Lock-free Find

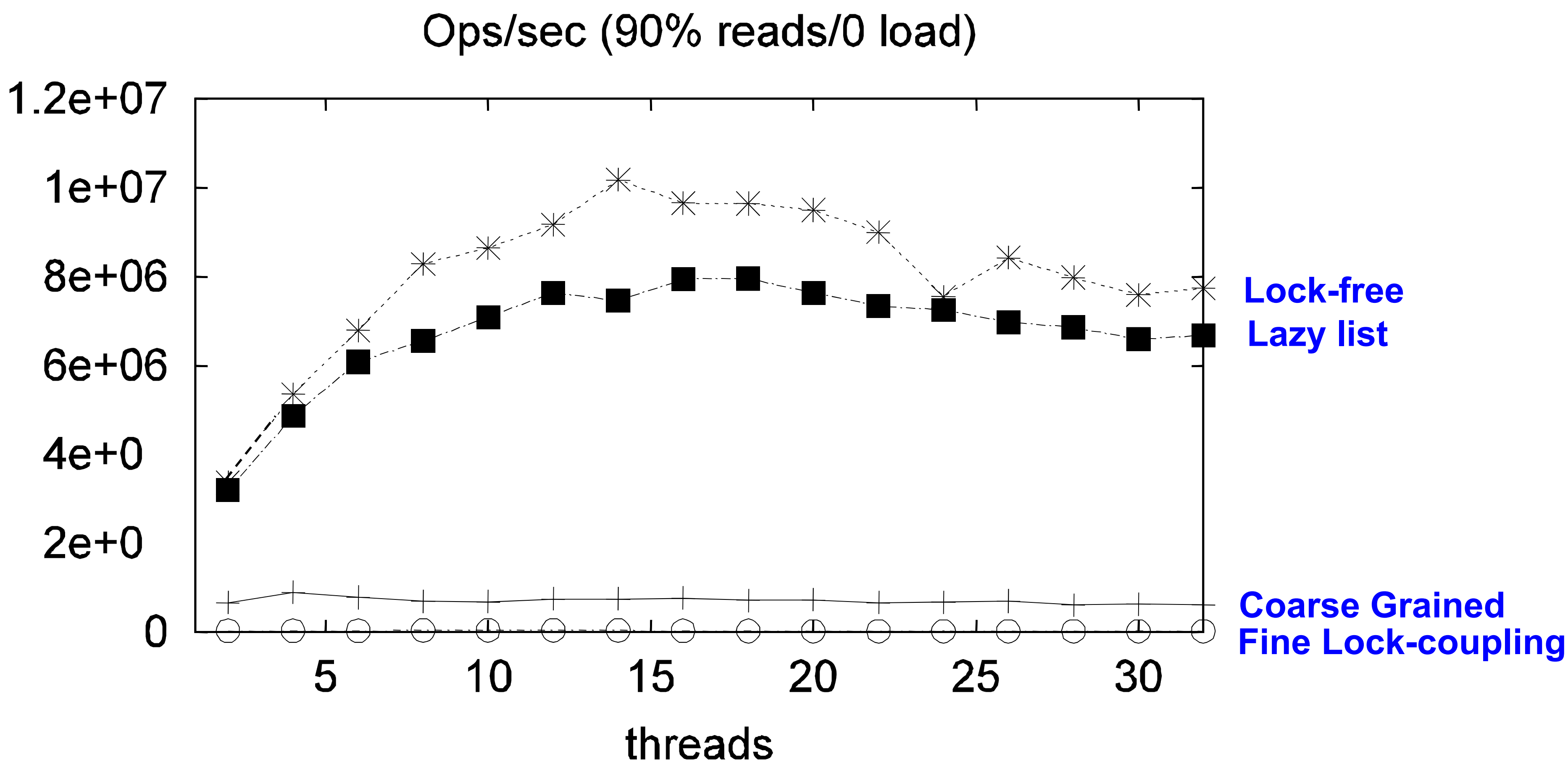
Otherwise move on to check if next node deleted

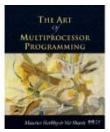
```
retry: while (true) {
   …
   while (marked[0]) {
     snip = pred.next.compareAndSet(curr,
                                     succ, false, false);
     if (!snip) continue retry;
     curr = succ;
     succ = curr.next.get(marked);
   }
…
```

Performance

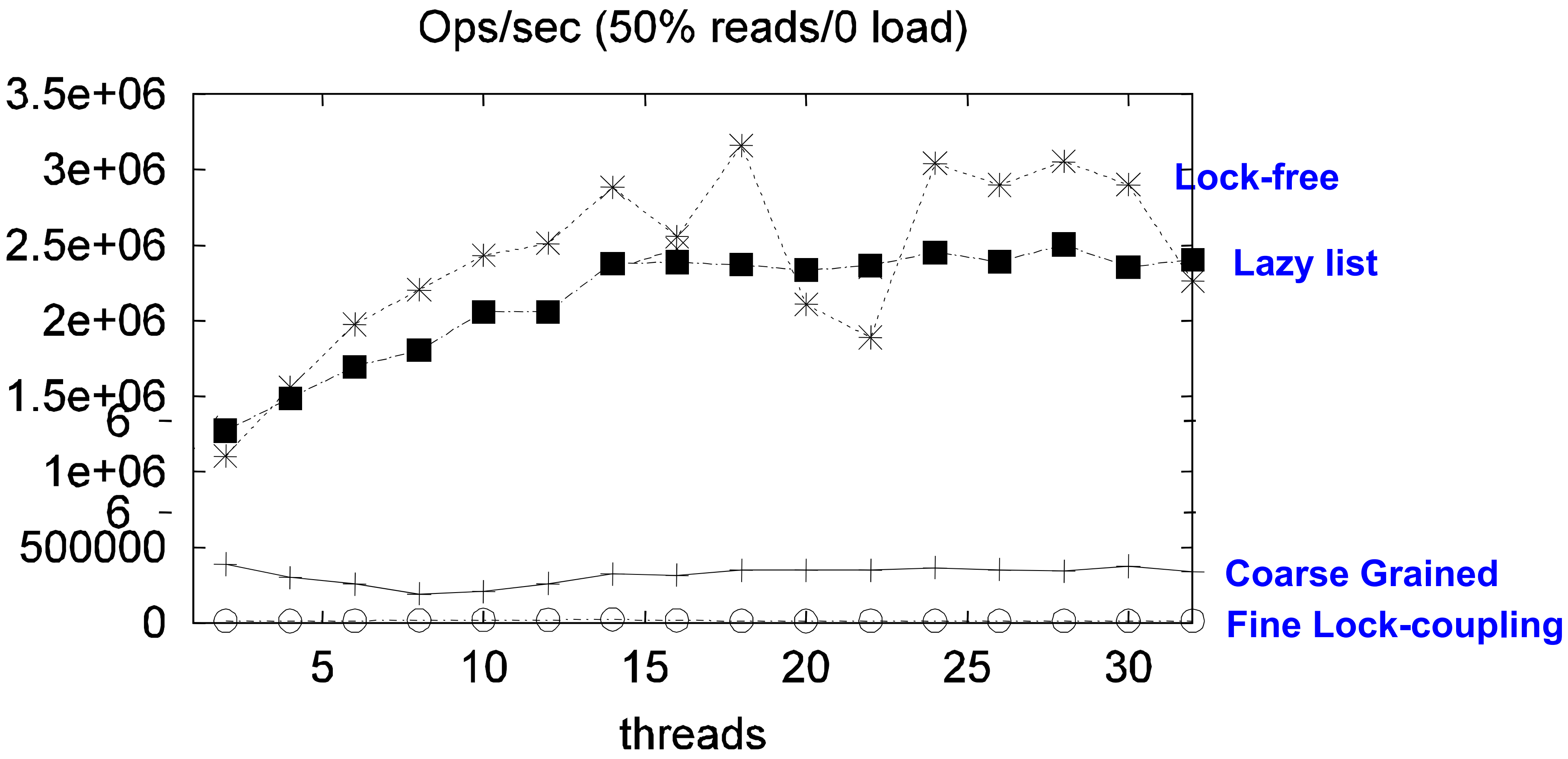
- Different list-based set implementaions
- 16-node machine
- Vary percentage of **`contains()`** calls

High Contains Ratio

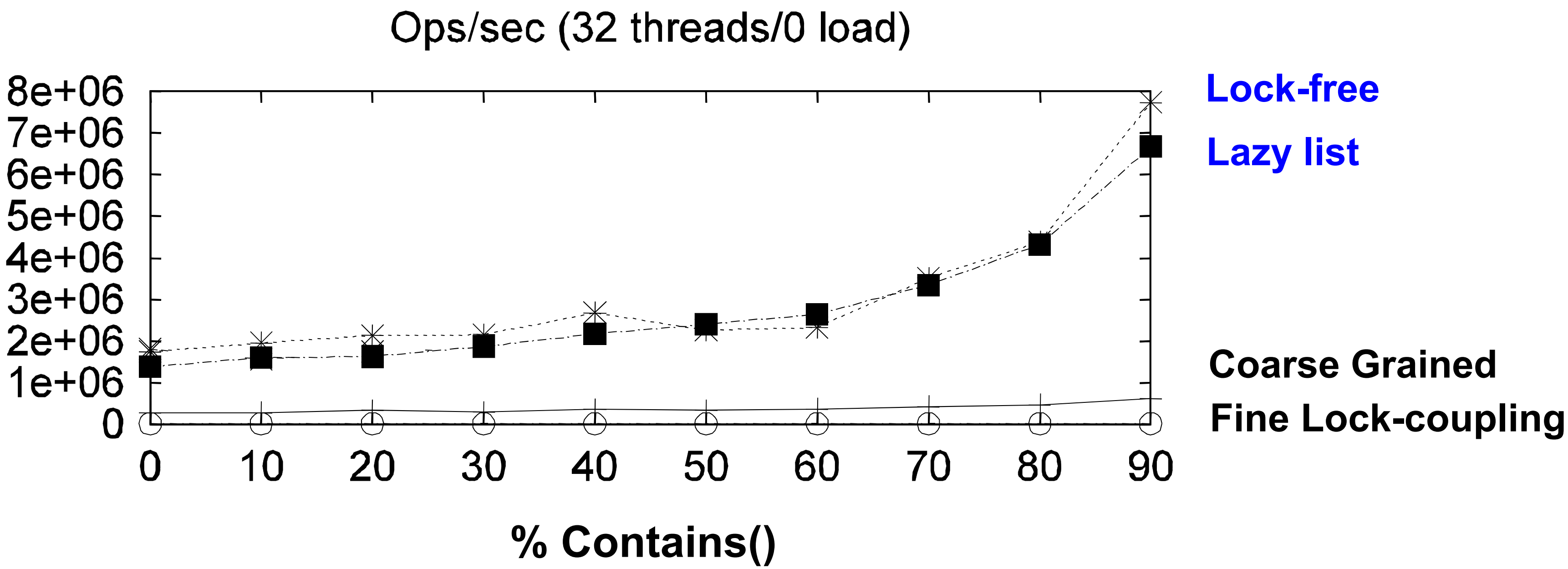


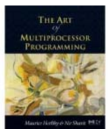


Low Contains Ratio



As Contains Ratio Increases





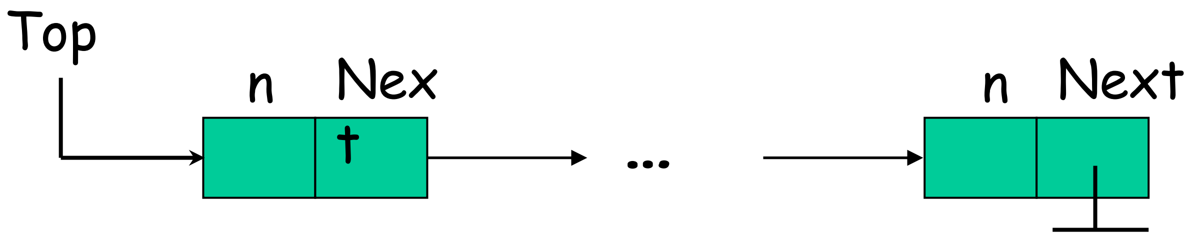
Summary

- Coarse-grained locking
- 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
- The answer: nobler to compromise
  - Example: Lazy list combines blocking **`add()`** and **`remove()`** and a wait-free **`contains()`**
  - Remember: Blocking/non-blocking is a property of a method

An Optimistic Lock-free Stack




```
pop( ){
    local  done, next, t;
    done  = false;
    while (!done) {
        t = Top;
        if (t==null) return null;
        next = t.Next;
       done = CAS(&      t, next);
    }
    re
```

```
push(x){
    local done, t;

    done = false;
    while(!done) {
        t = Top;
        x.Next = t;
       done =  CAS(&Top, t, x);
    }
```

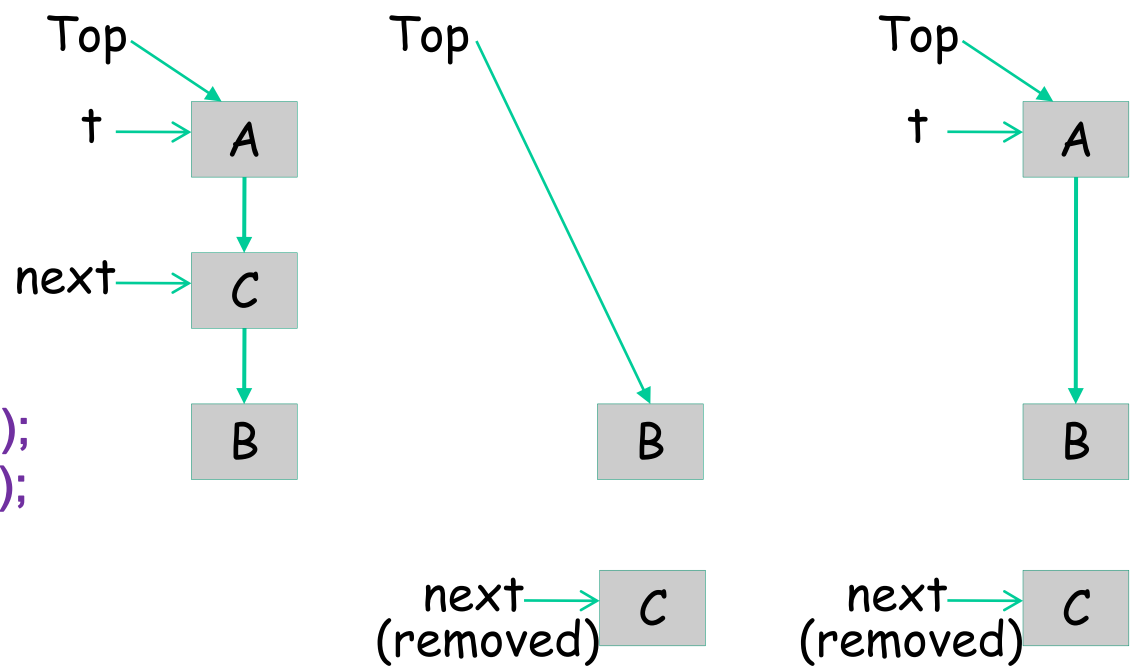
Bug#2: ABA problem leads to corrupted stacks

ABA Problem

Threads T1 and T2 are interleaved as follows:

T1:
pop()
{
    t = Top
    next = t.Next
    interrupted

T2:
    a = pop();
    c = pop();
    push(a);



    resumes
CAS(&Top,t,next)
succeeds
stack corrupted

Timeline

Summary

Our winner:	Optimistic Lock-free.
Second best:	Lazy.
Third:	Optimistic.
Fourth:	Fine-Grained.
Last:	Coarse-Grained.

?

Summary

Answer: No.

Choose your implementation carefully based on your requirements.

Summary

- Concurrent programming is hard.
- Concurrency is error-prone.
- Formal method is necessary.