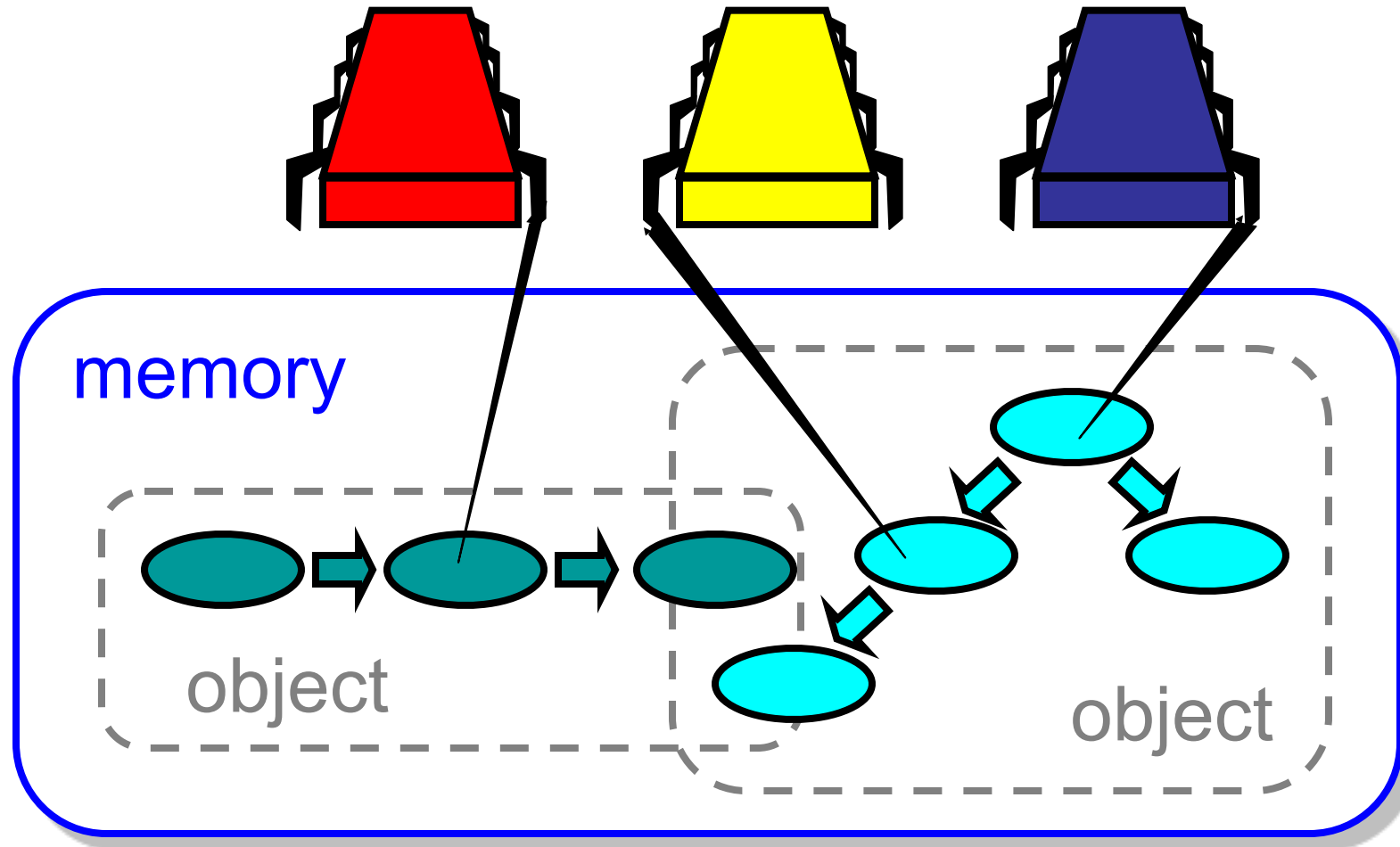


# YSC3248: Parallel, Concurrent and Distributed Programming

Concurrent Objects

# Concurrent Computation



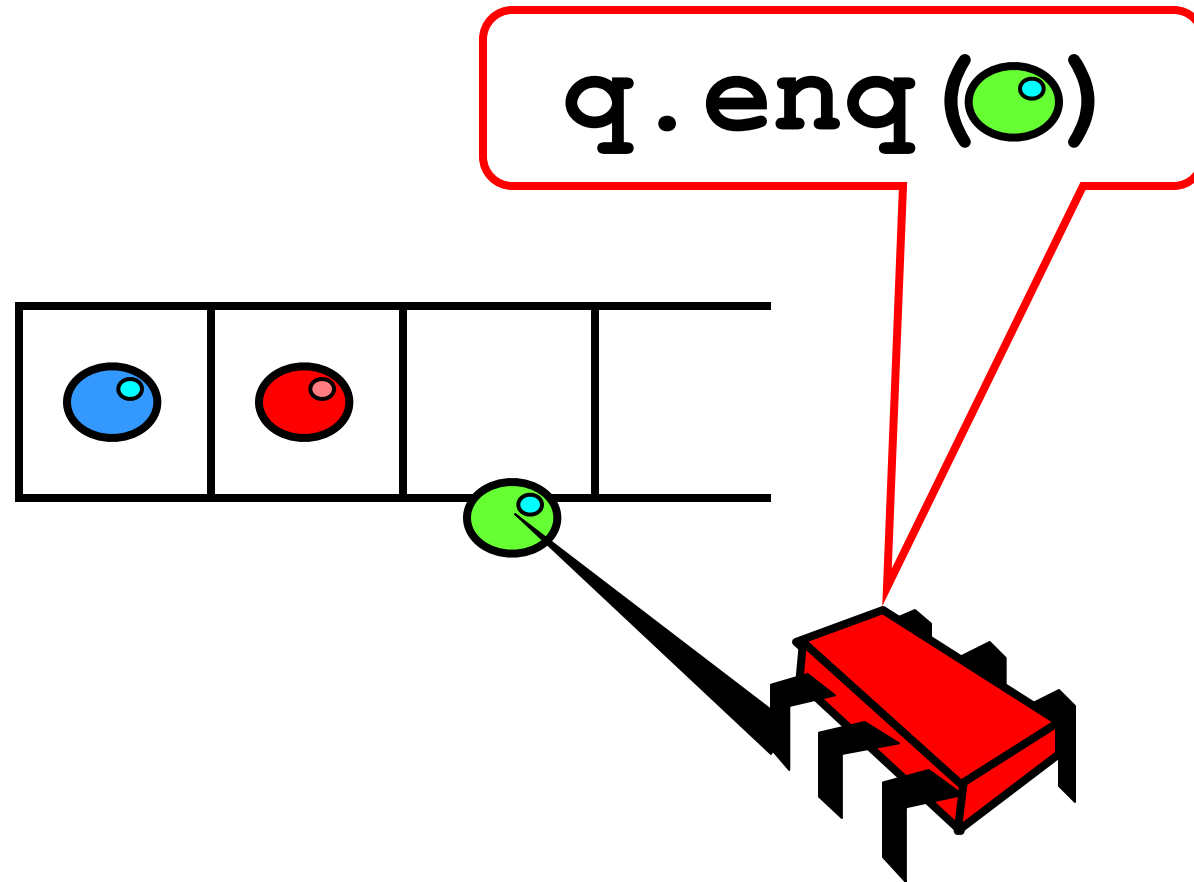
# Objectivism

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

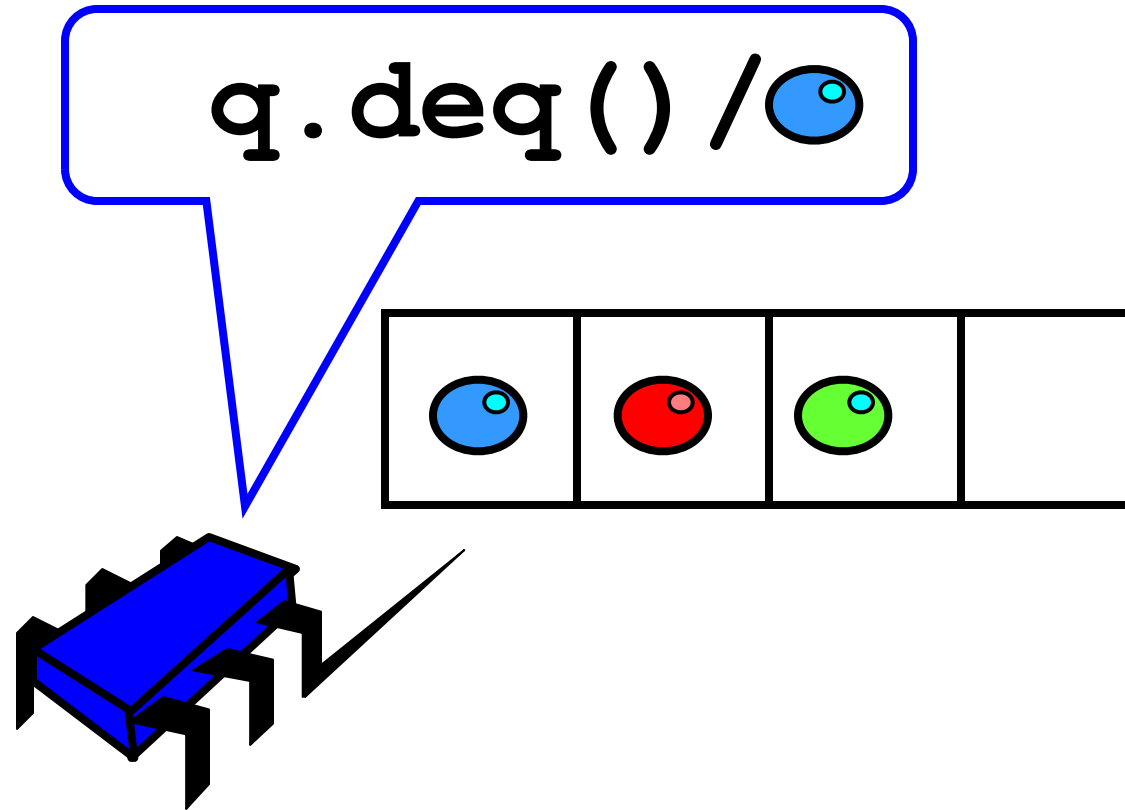
# Objectivism

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

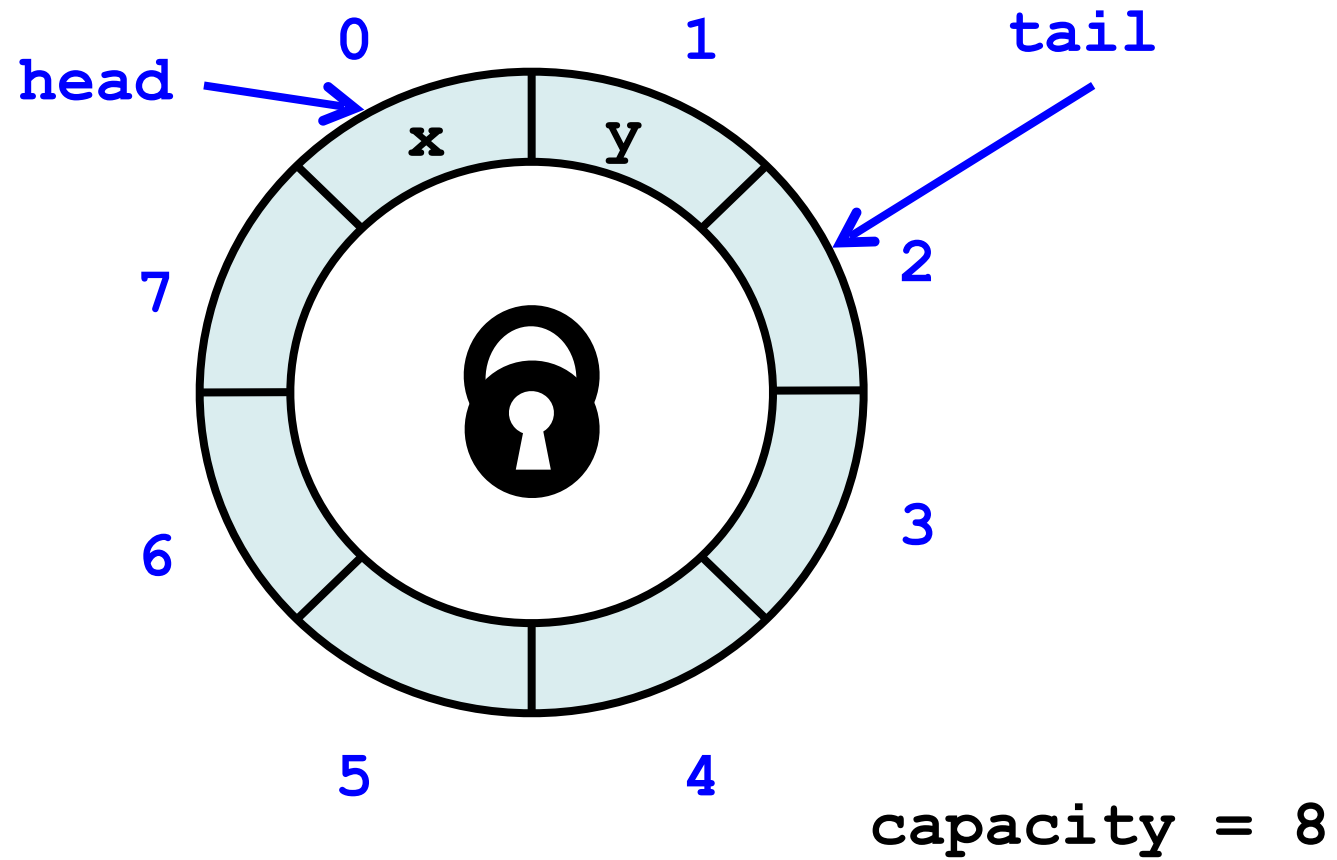
# FIFO Queue: Enqueue Method



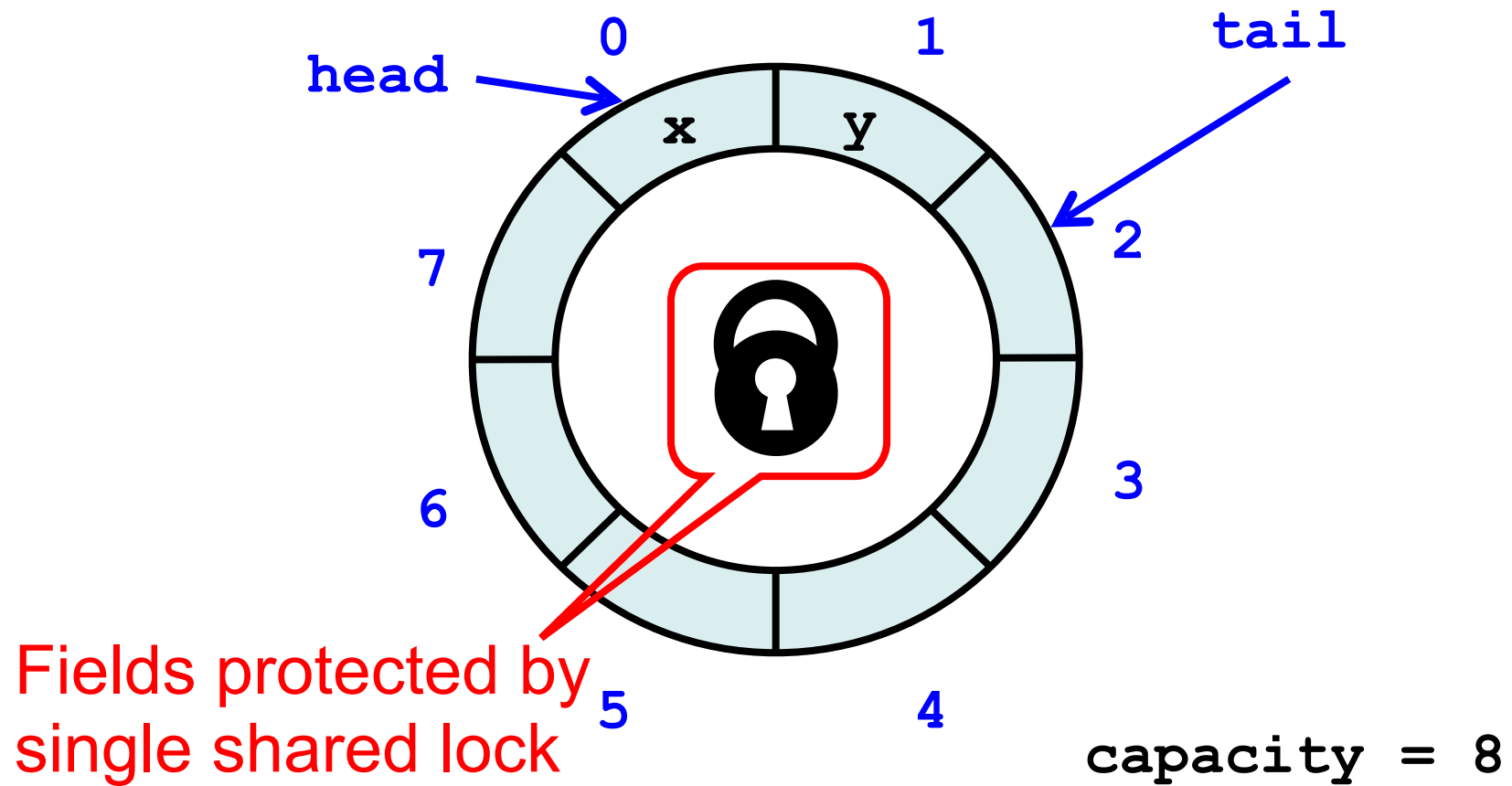
# FIFO Queue: Dequeue Method



# Lock-Based Queue



# Lock-Based Queue



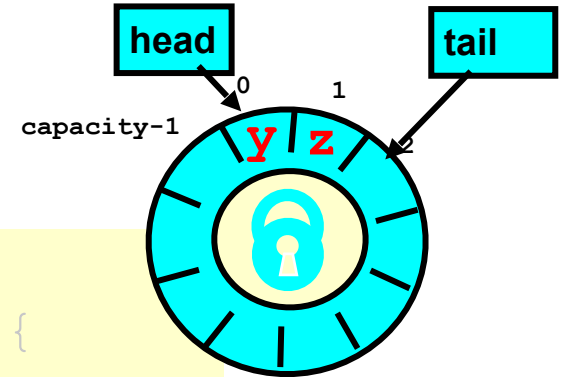


# A Lock-Based Queue

```
class LockBasedQueue[T: ClassTag]  
  (val capacity: Int) extends MyQueue[T] {  
  
  private var head, tail: Int = 0  
  private val items = new Array[T](capacity)  
  private val myLock = new ReentrantLock()
```

# A Lock-Based Queue

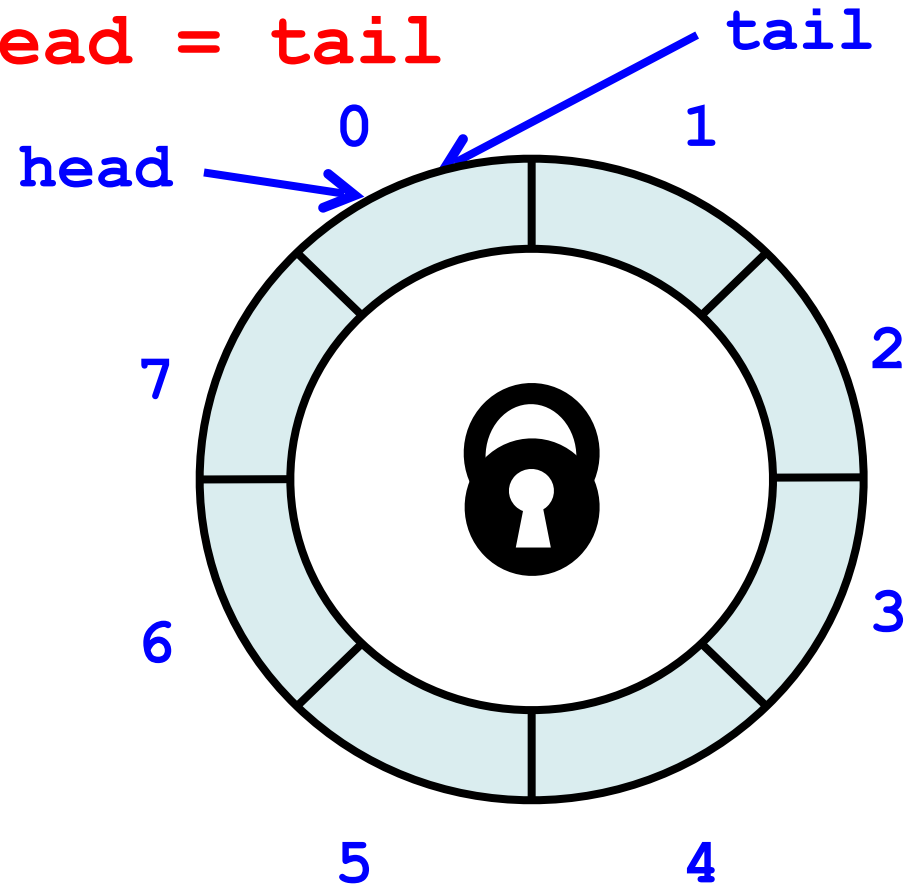
```
class LockBasedQueue[T: ClassTag]  
  (val capacity: Int) extends MyQueue[T] {  
  
    private var head, tail: Int = 0  
    private val items = new Array[T](capacity)  
    private val myLock = new ReentrantLock()
```



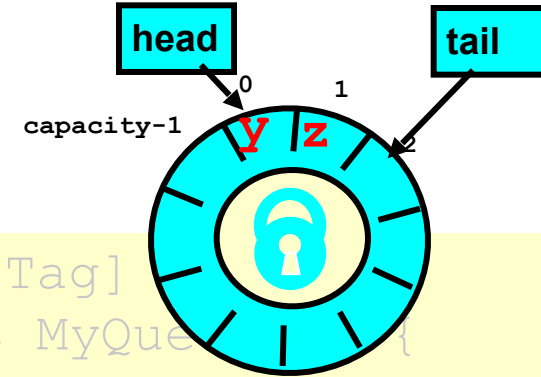
Fields protected by  
single shared lock

# Lock-Based Queue

Initially: **head = tail**



# Lock-Based Queue

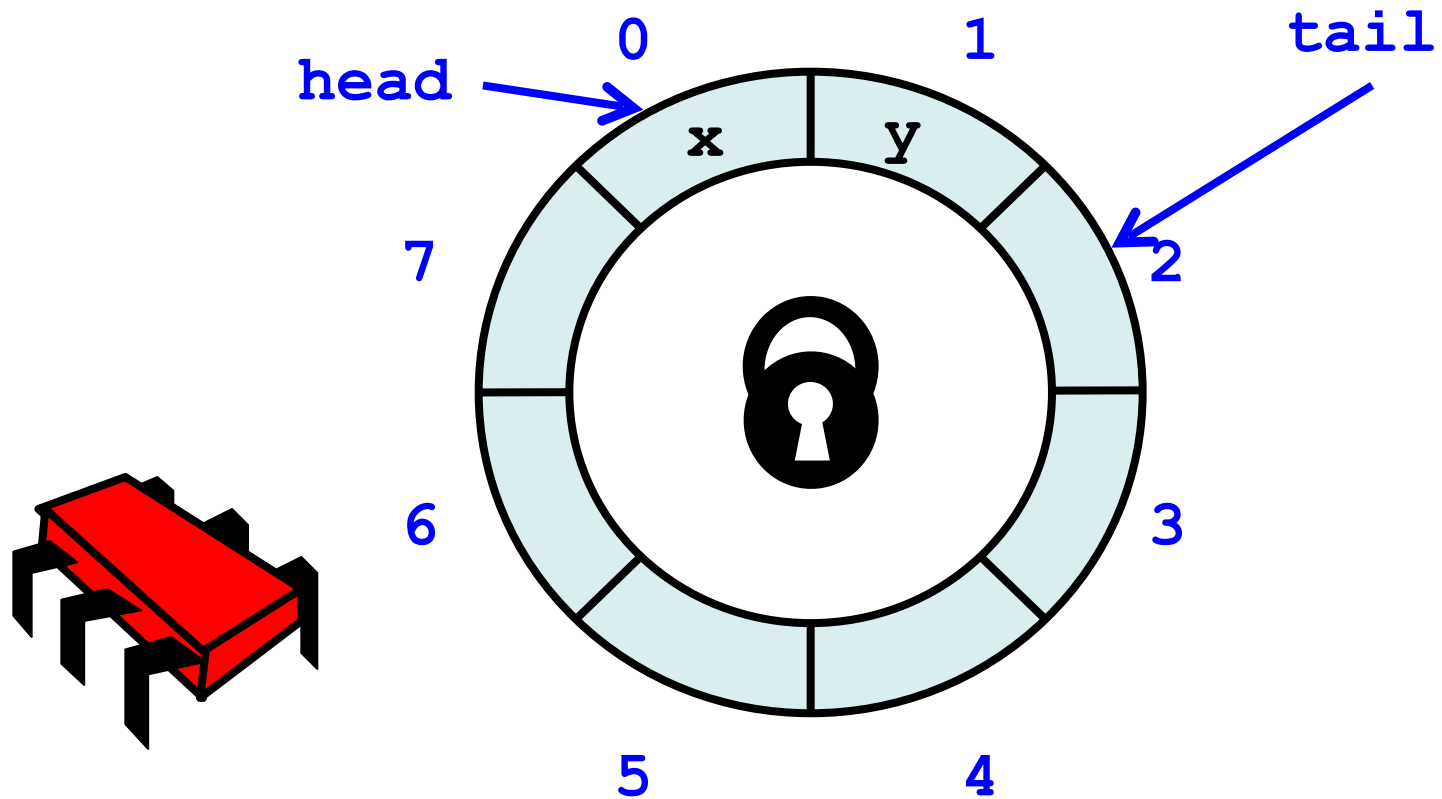


```
class LockBasedQueue[T: ClassTag]  
  (val capacity: Int) extends MyQueue {
```

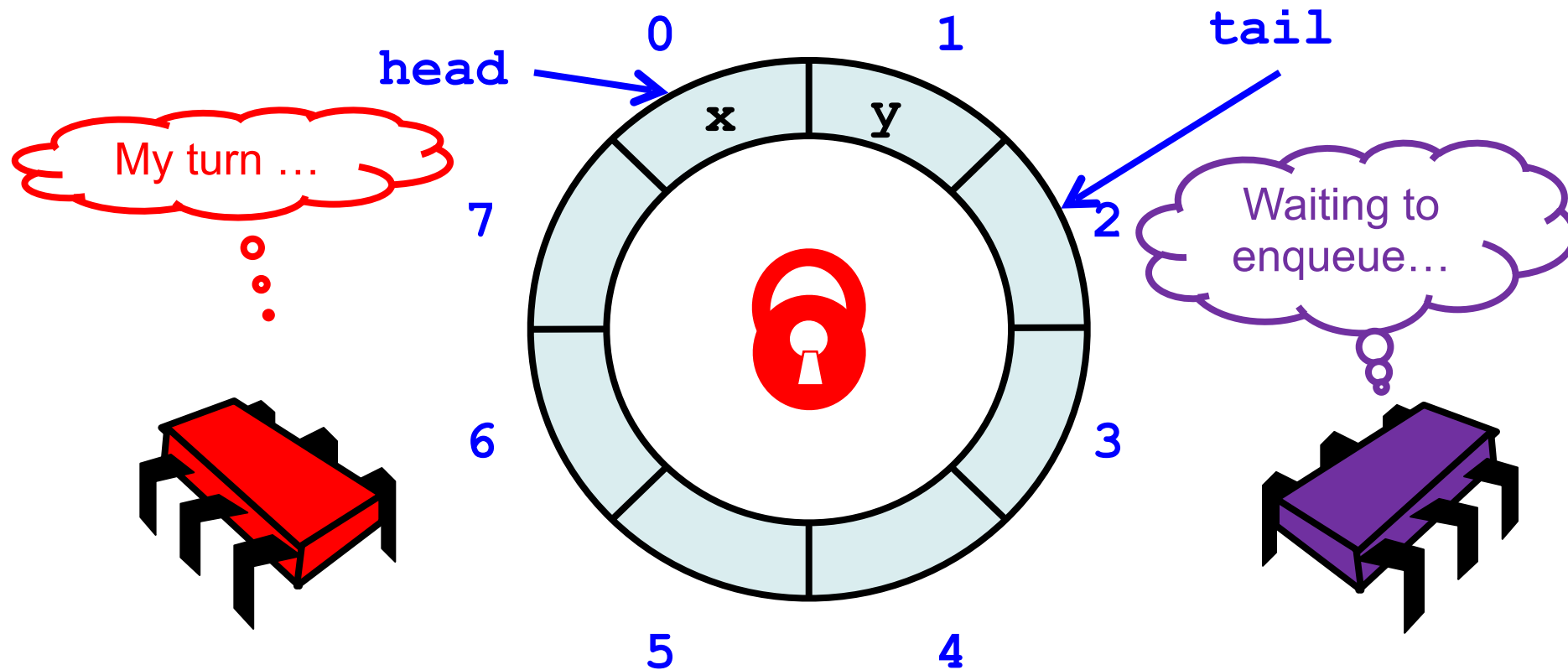
```
  private var head, tail: Int = 0  
  private val items = new Array[T](capacity)  
  private val myLock = new ReentrantLock()
```

Initially head = tail

# Lock-Based `deq()`

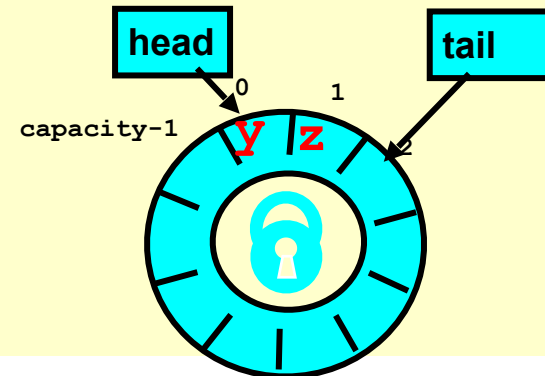


# Acquire Lock



# Implementation: `deq()`

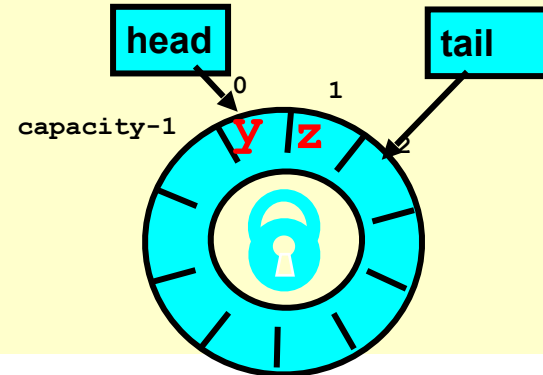
```
def deq() : T = {  
  myLock.lock()  
  try {  
    if (tail == head) {  
      throw EmptyException  
    }  
    val x = items(head % items.length)  
    head = head + 1  
    x  
  } finally {  
    myLock.unlock()  
  }  
}
```



# Implementation: `deq()`

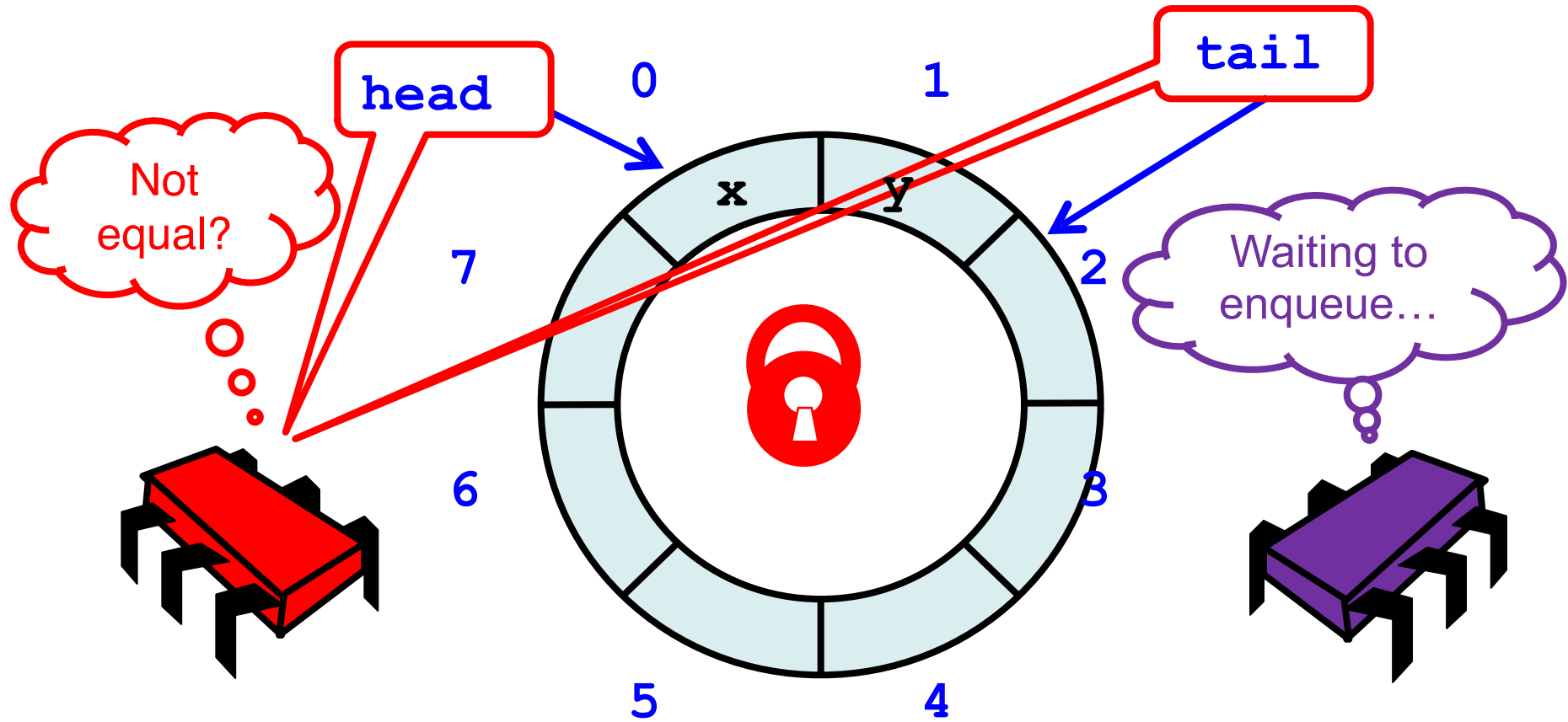
```
def deq() : T = {  
  myLock.lock()  
  try {  
    if (tail == head) {  
      throw EmptyException  
    }  
    val x = items(head % items.length)  
    head = head + 1  
    x  
  } finally {  
    myLock.unlock()  
  }  
}
```

Acquire lock at  
method start





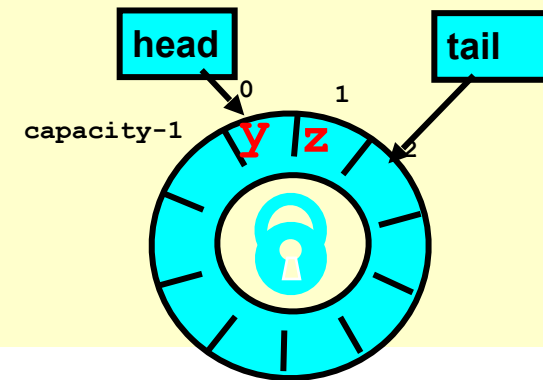
# Check if Non-Empty



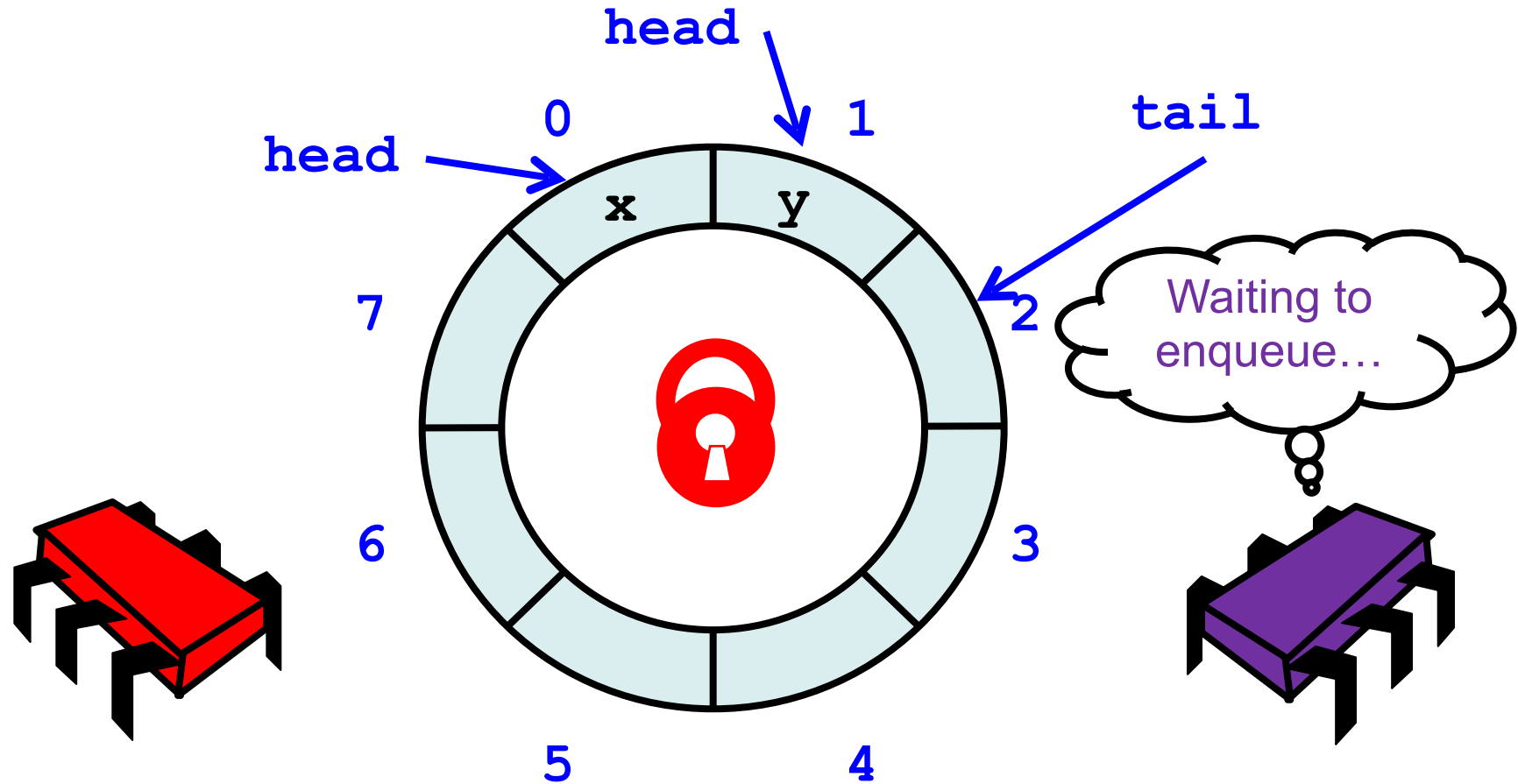
# Implementation: `deq()`

```
def deq() : T = {  
  myLock.lock()  
  try {  
    if (tail == head) {  
      throw EmptyException  
    }  
    val x = items(head % items.length)  
    head = head + 1  
    x  
  } finally {  
    myLock.unlock()  
  }  
}
```

If queue empty  
throw exception



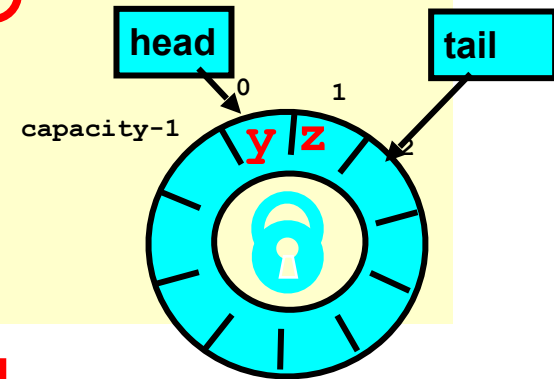
# Modify the Queue



# Implementation: `deq()`

```
def deq() : T = {  
  myLock.lock()  
  try {  
    if (tail == head) {  
      throw EmptyException  
    }  
  
    val x = items(head % items.length)  
    head = head + 1  
    x  
  } finally {  
    myLock.unlock()  
  }  
}
```

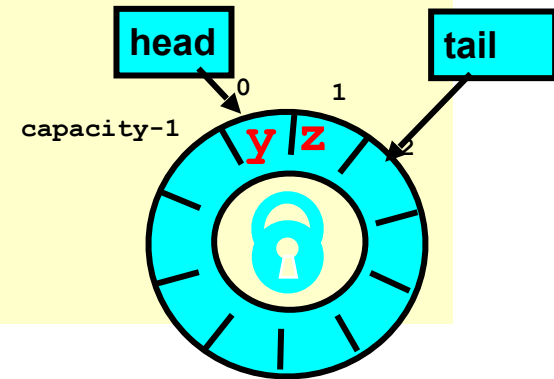
Queue not empty?  
Remove item and update head



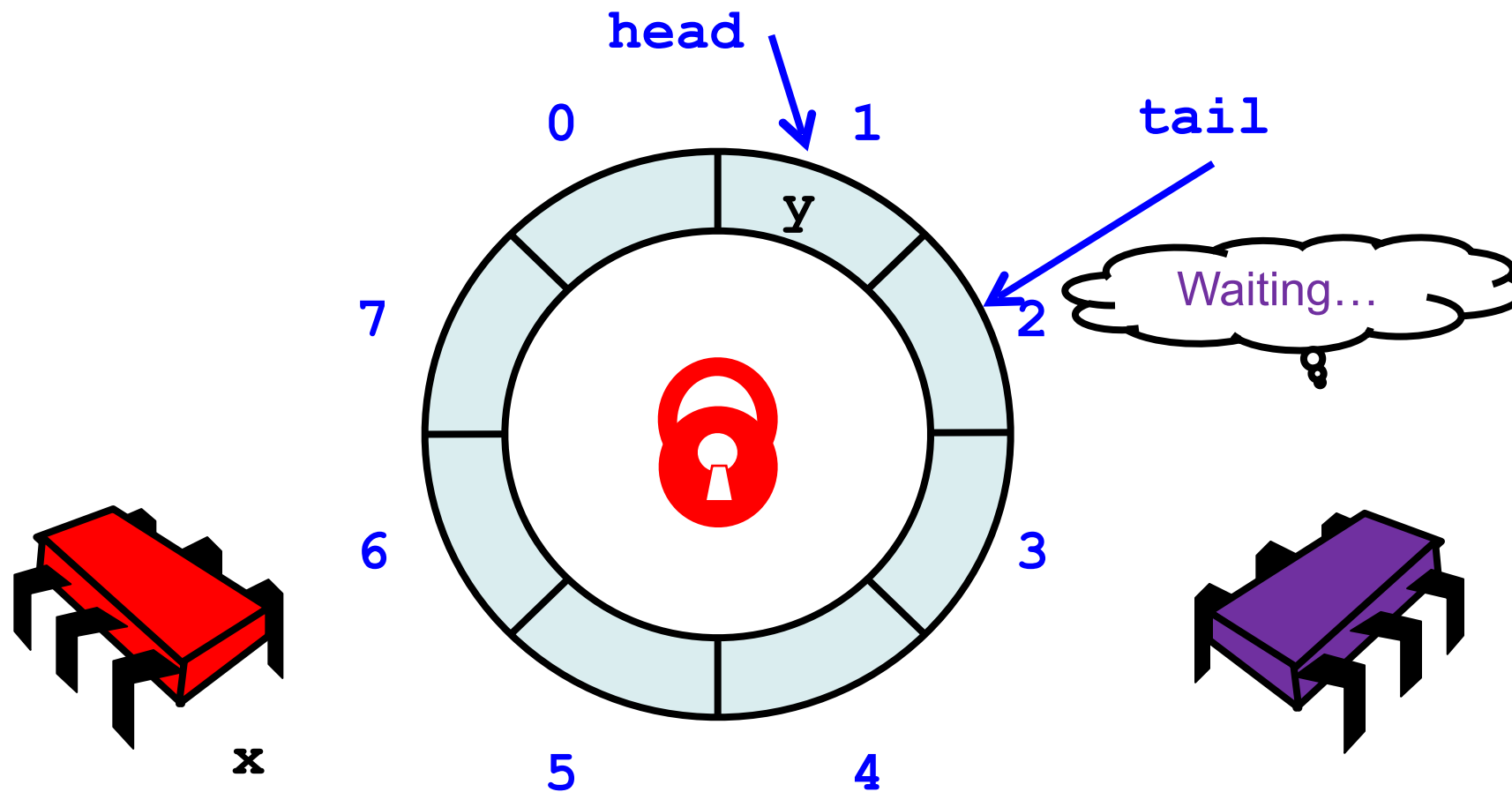
# Implementation: `deq()`

```
def deq() : T = {  
  myLock.lock()  
  try {  
    if (tail == head) {  
      throw EmptyException  
    }  
    val x = items(head % items.length)  
    head = head + 1  
    x  
  } finally {  
    myLock.unlock()  
  }  
}
```

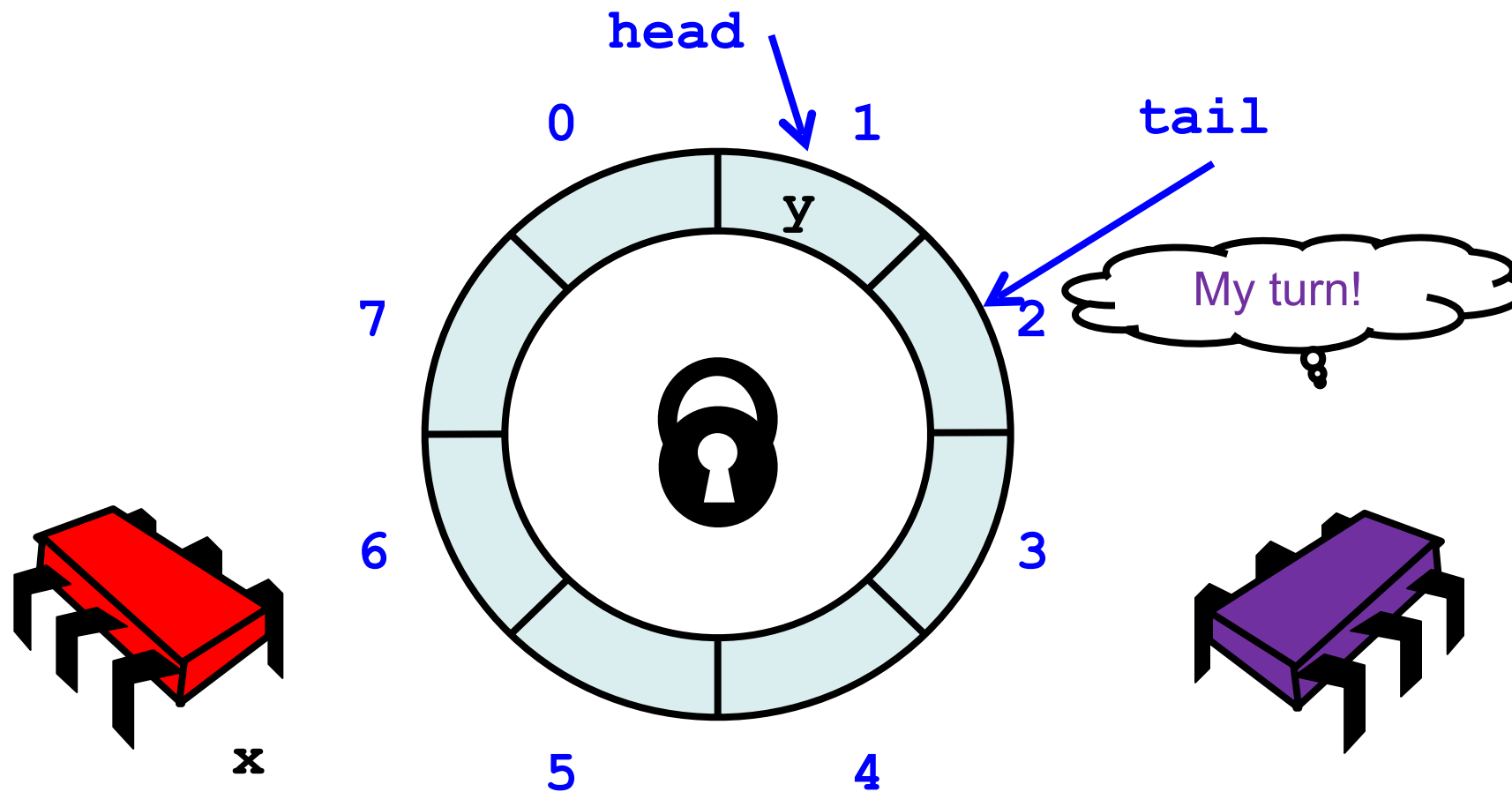
Return result



# Release the Lock



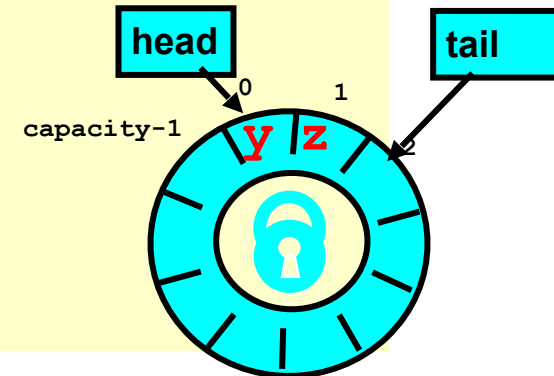
# Release the Lock



# Implementation: `deq()`

```
def deq() : T = {  
  myLock.lock()  
  try {  
    if (tail == head) {  
      throw EmptyException  
    }  
    val x = items(head % items.length)  
    head = head + 1  
    x  
  }  
  finally {  
    myLock.unlock()  
  }  
}
```

Release lock no  
matter what!





# Implementation: `deq()`

```
def deq() : T = {  
  myLock.lock()  
  try {  
    if (tail == head) {  
      throw EmptyException  
    }  
    val x = items(head % items.length)  
    head = head + 1  
    x  
  } finally {  
    myLock.unlock()  
  }  
}
```

Should be correct because  
modifications are mutually exclusive...

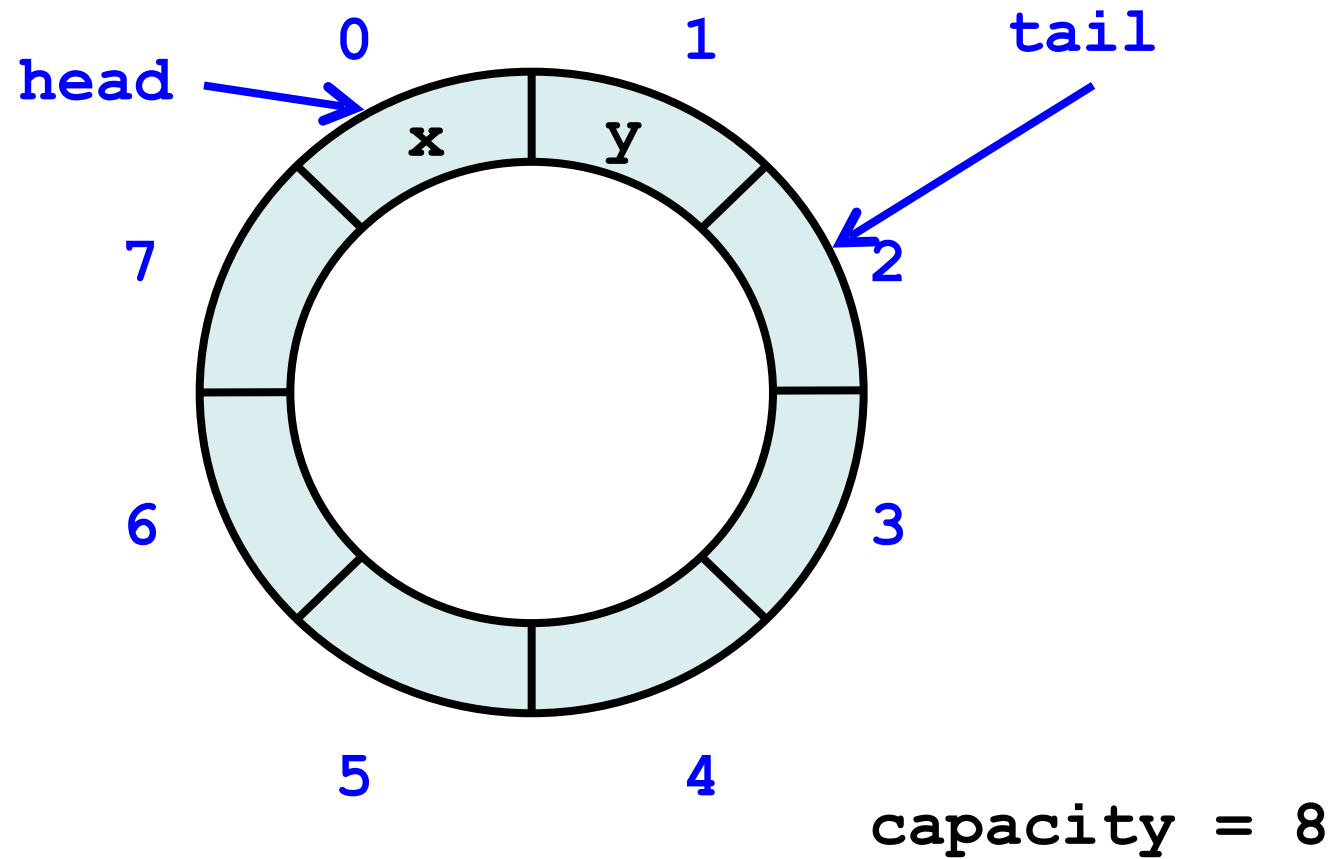
# Demo:

## Testing SWSR Queue

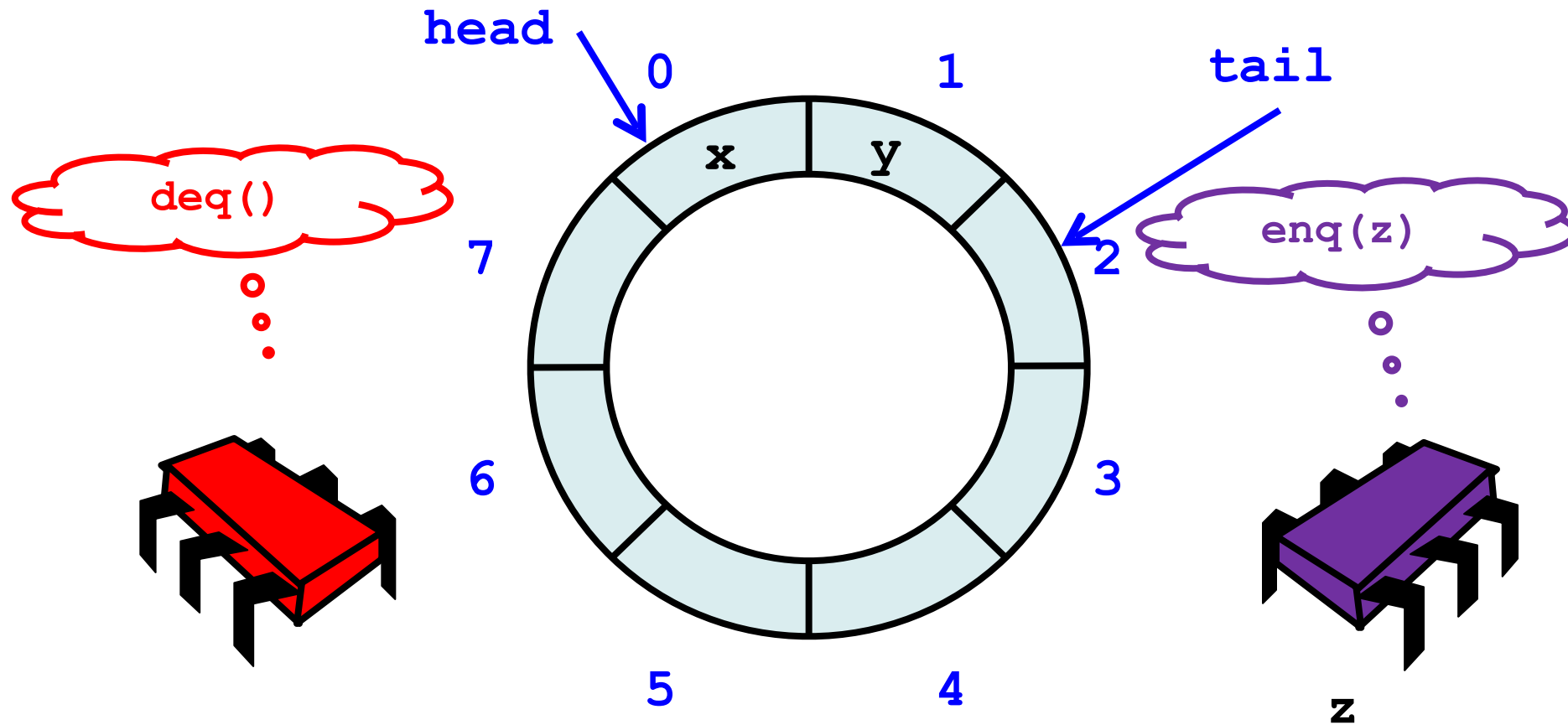
# Now consider the following implementation

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

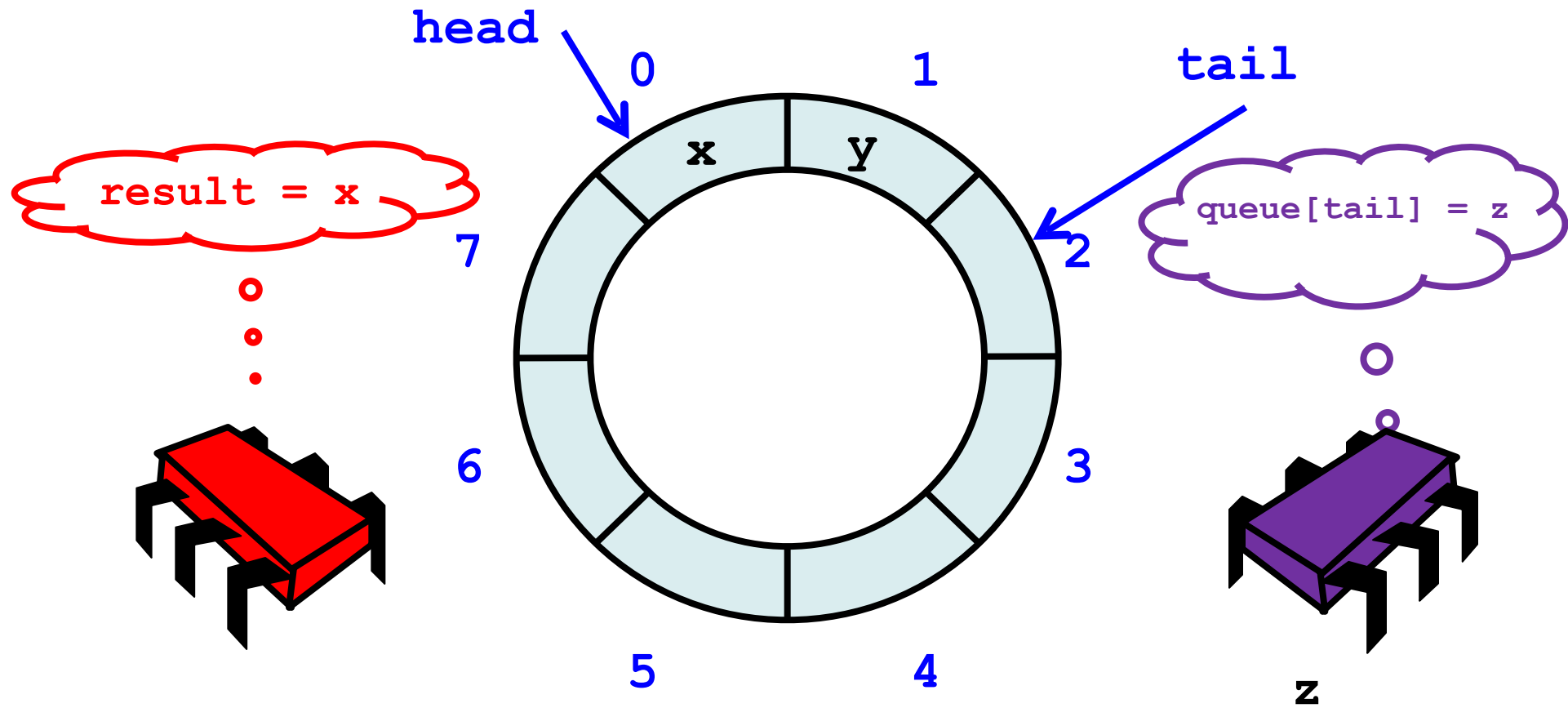
# Wait-free 2-Thread Queue



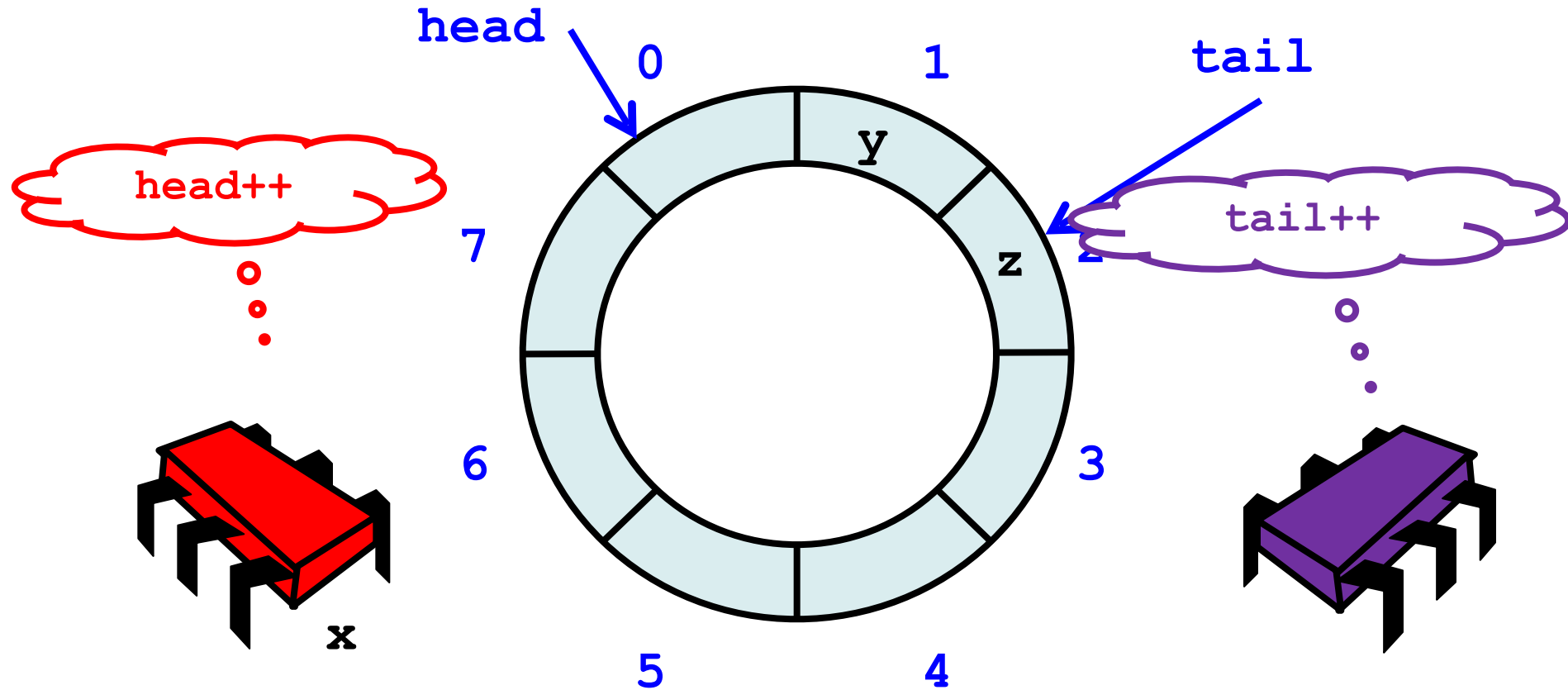
# Wait-free 2-Thread Queue



# Wait-free 2-Thread Queue

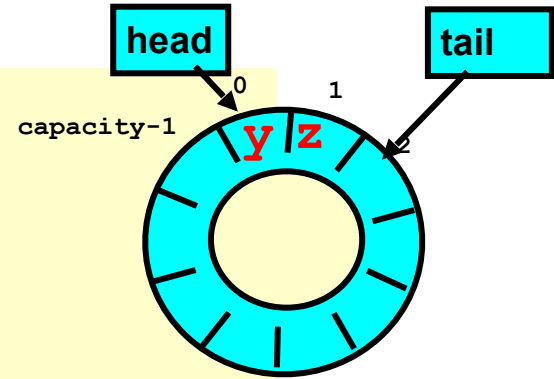


# Wait-free 2-Thread Queue



# Wait-free 2-Thread Queue

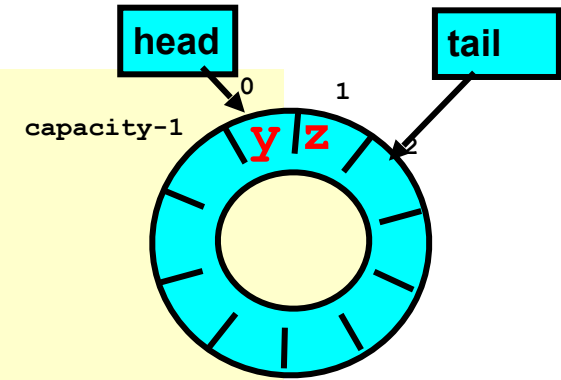
```
class LockFreeQueue[T: ClassTag](val capacity: Int) {  
  
  @volatile  
  private var head, tail: Int = 0  
  private val items = new Array[T](capacity)  
  
  def enq(x: T): Unit = {  
    if (tail - head == items.length) throw FullException  
    items(tail % items.length) = x  
    tail = tail + 1  
  }  
  
  def deq(): T = {  
    if (tail == head) throw EmptyException  
    val x = items(head % items.length)  
    head = head + 1  
    x  
  }  
}
```





# Wait-free 2-Thread Queue

```
class LockFreeQueue[T: ClassTag](val capacity: Int) {  
  
  @volatile  
  private var head, tail: Int = 0  
  private val items = new Array[T](capacity)  
  
  def enq(x: T): Unit = {  
    if (tail - head == items.length) throw FullException  
    items(tail % items.length) = x  
    tail = tail + 1  
  }  
  
  def deq(): T = {  
    if (tail == head) throw EmptyException  
    val x = items(head % items.length)  
    head = head + 1  
    x  
  }  
}
```



**No lock needed!**

# Wait-free 2-Thread Queue

```
class LockFreeQueue[T: ClassTag](val capacity: Int) {  
  
  @volatile  
  private var head, tail: Int = 0  
  private val items = new Array[T](capacity)  
  
  def enq(x: T): Unit = {  
    if (tail - head == items.length) throw FullException  
    items(tail % items.length) = x  
    tail = tail + 1  
  }  
  
  def deq(): T = {  
    if (tail == head) throw EmptyException  
    val x = items(head % items.length)  
    head = head + 1  
    x  
  }  
}
```

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

# What *is* a Concurrent Queue?

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

# Correctness and Progress

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

**Lets begin with correctness**

# Sequential Objects

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

# Sequential Specifications

- If (precondition)
  - the object is in such-and-such a state
  - before you call the method,
- Then (postcondition, result)
  - the method will return a particular value
  - or throw a particular exception,
- and (postcondition, state)
  - the object will be in some other state
  - when the method returns

# Pre and PostConditions for Dequeue

- **Precondition:**
  - Queue is non-empty
- **Postcondition (result):**
  - Returns first item in queue
- **Postcondition (state):**
  - Removes first item in queue

# Pre and PostConditions for Dequeue

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



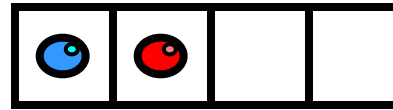
# Why Sequential Specifications *Totally Rock*

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

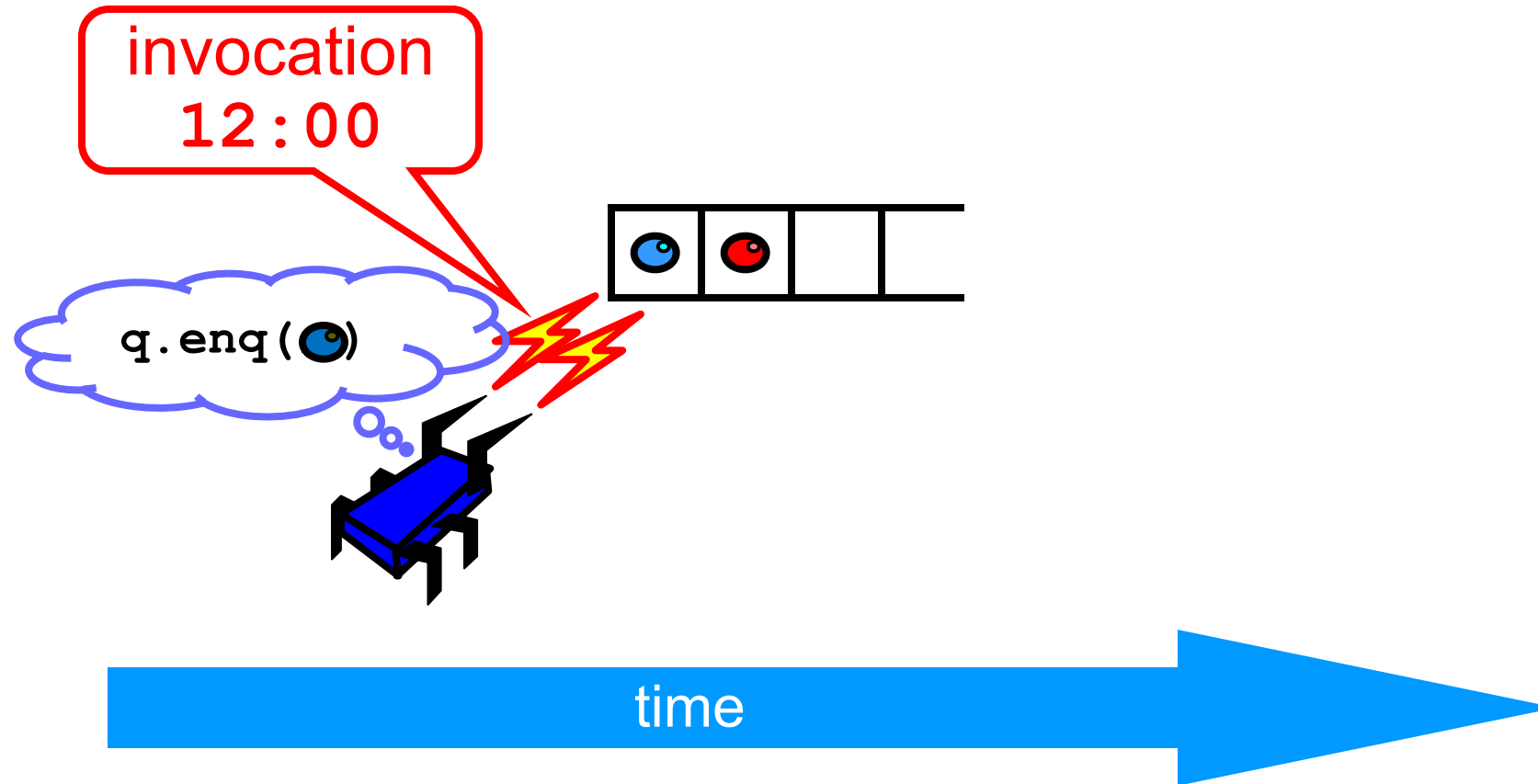
# What About Concurrent Specifications ?

- Methods?
- Documentation?
- Adding new methods?

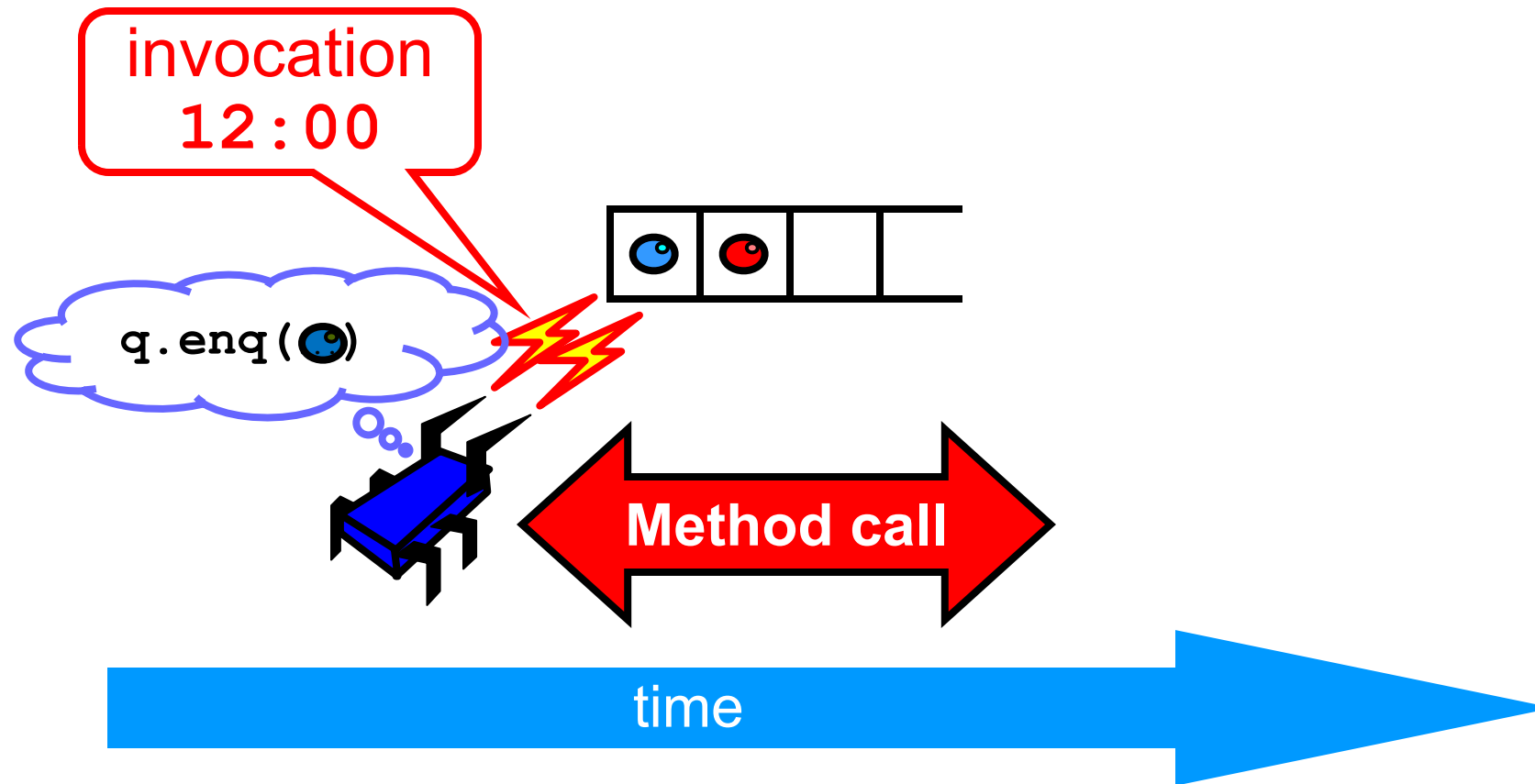
# Methods Take Time



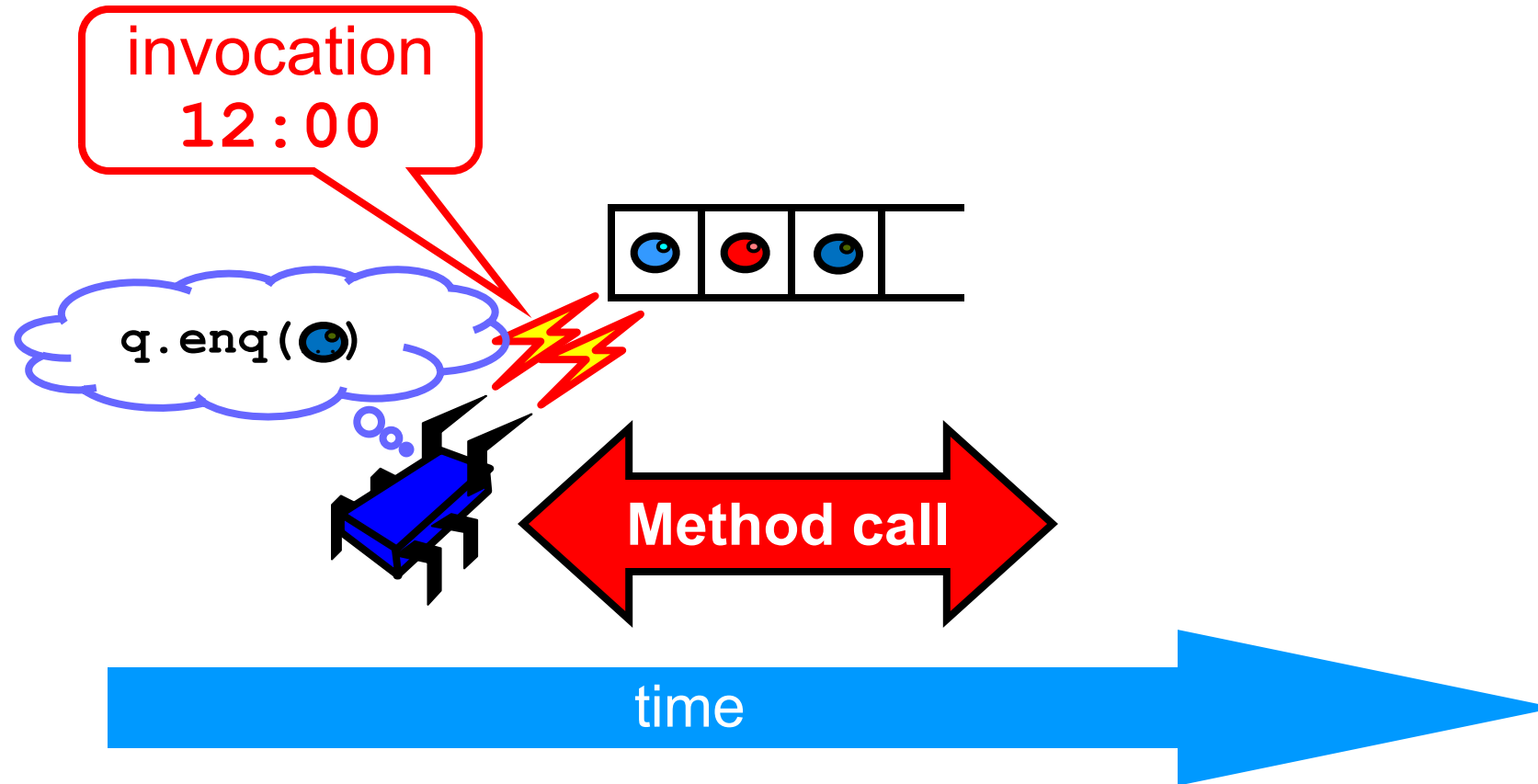
# Methods Take Time



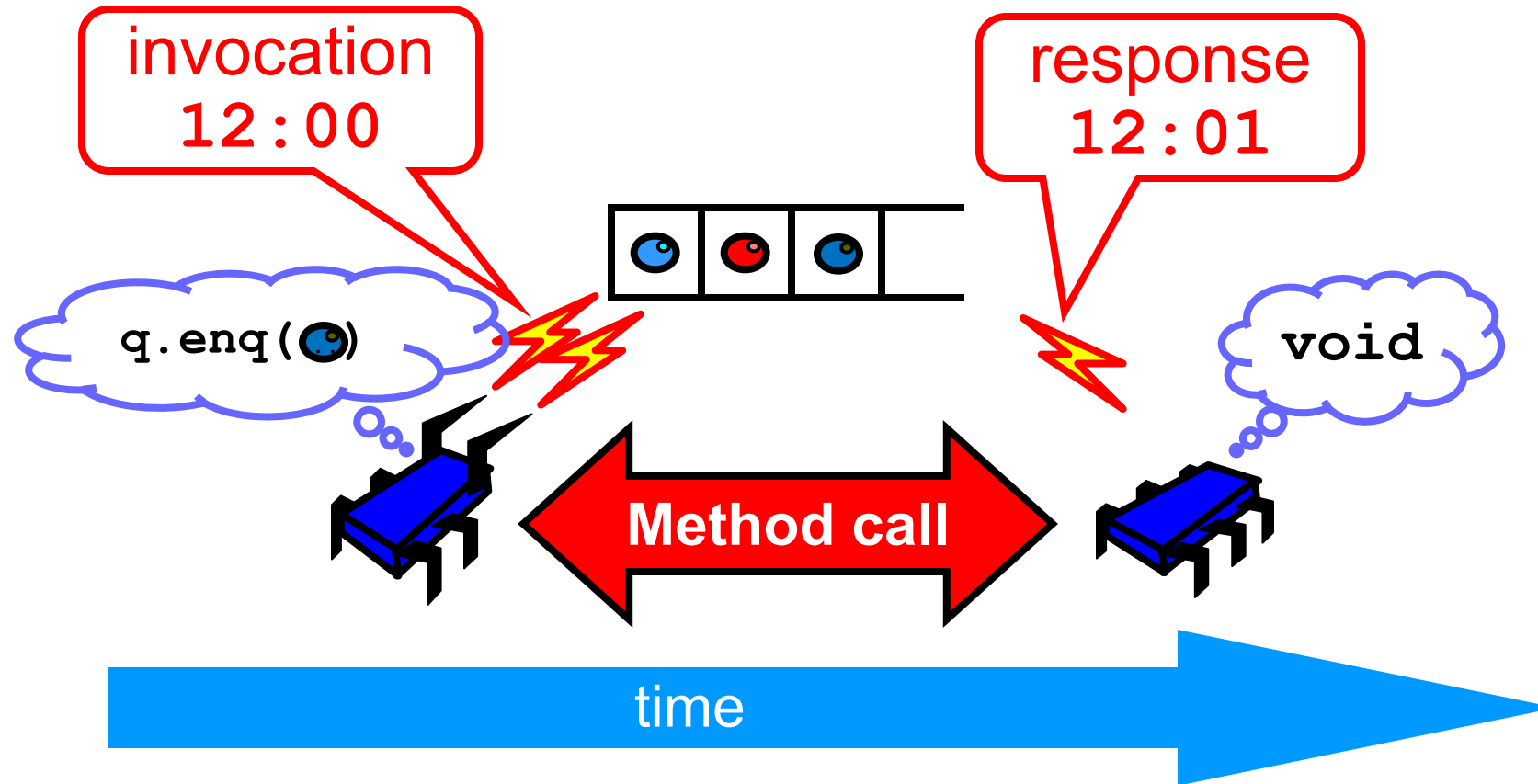
# Methods Take Time



# Methods Take Time



# Methods Take Time

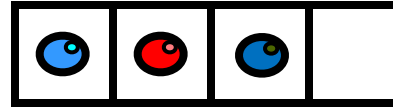


# Sequential vs Concurrent

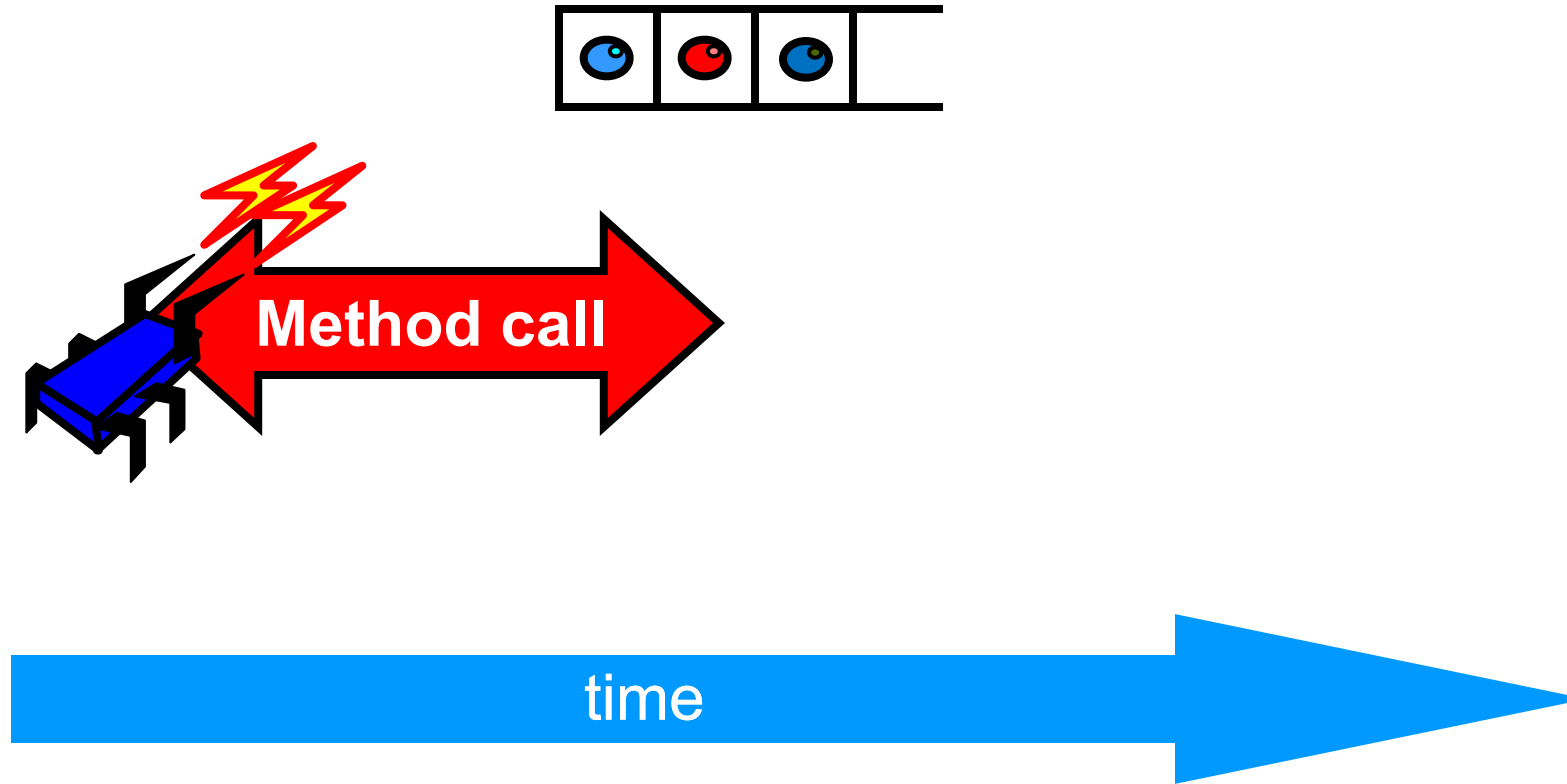
- Sequential
  - Methods take time? Who knew?
- Concurrent
  - Method call is not an event
  - Method call is a sequence of interval events.



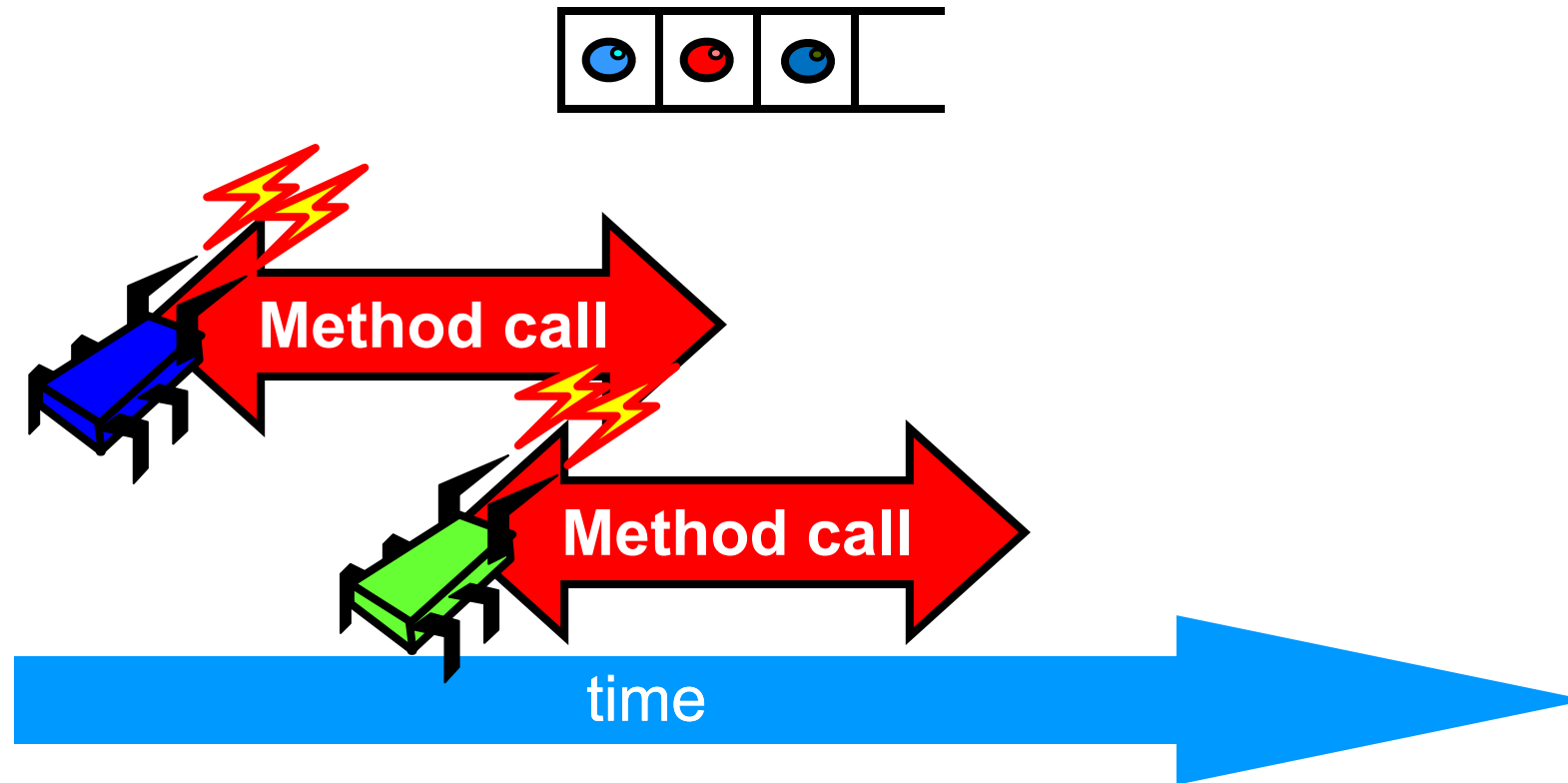
# Concurrent Methods Take **Overlapping** Time



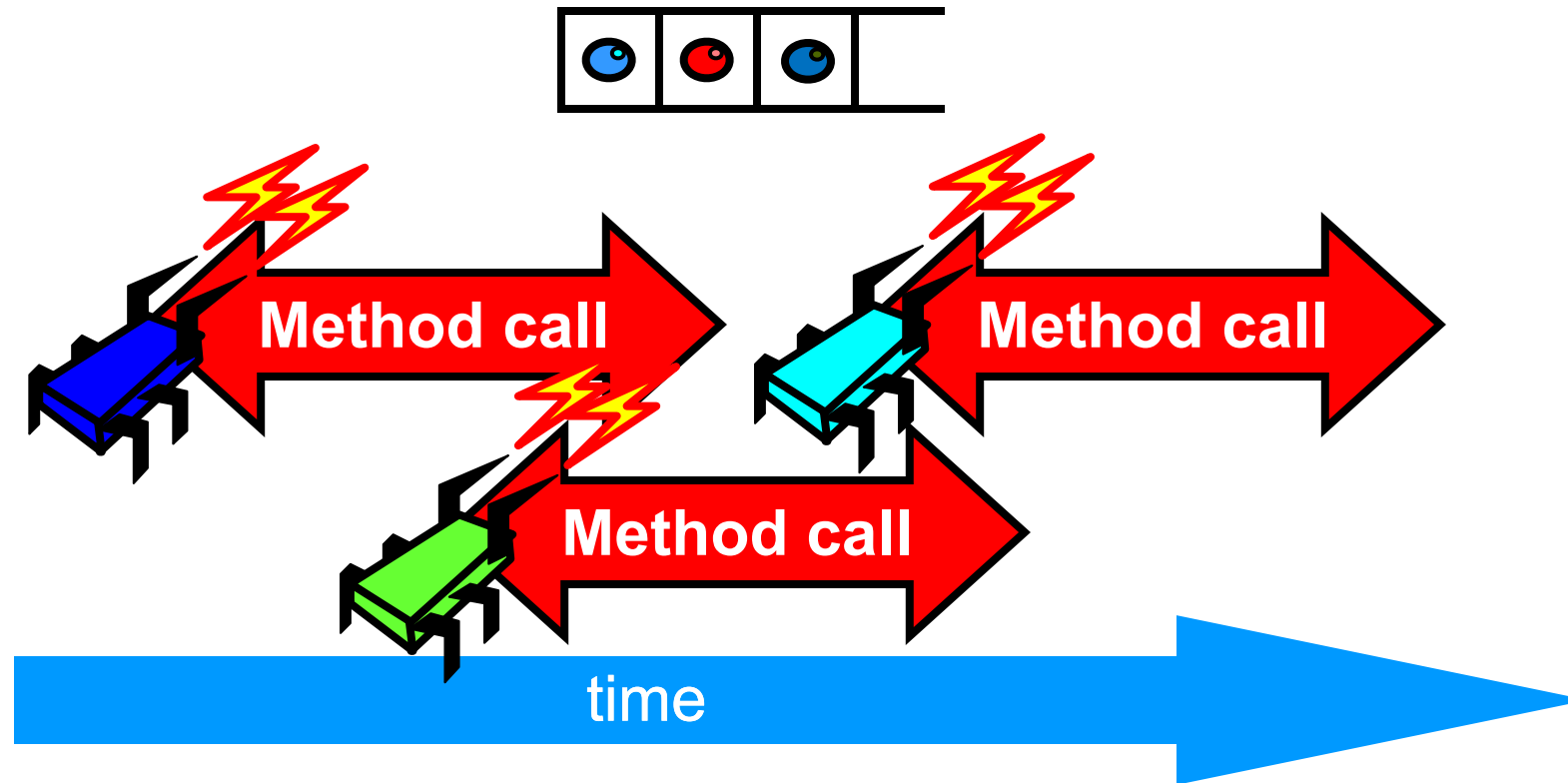
# Concurrent Methods Take **Overlapping** Time



# 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 `enq()` calls overlap?
    - Two `deq()` calls? `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!**



# The Big Question

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

# Intuitively...

```
def deq() : T = {  
  myLock.lock()  
  try {  
    if (tail == head) {  
      throw EmptyException  
    }  
    val x = items(head % items.length)  
    head = head + 1  
    x  
  } finally {  
    myLock.unlock()  
  }  
}
```

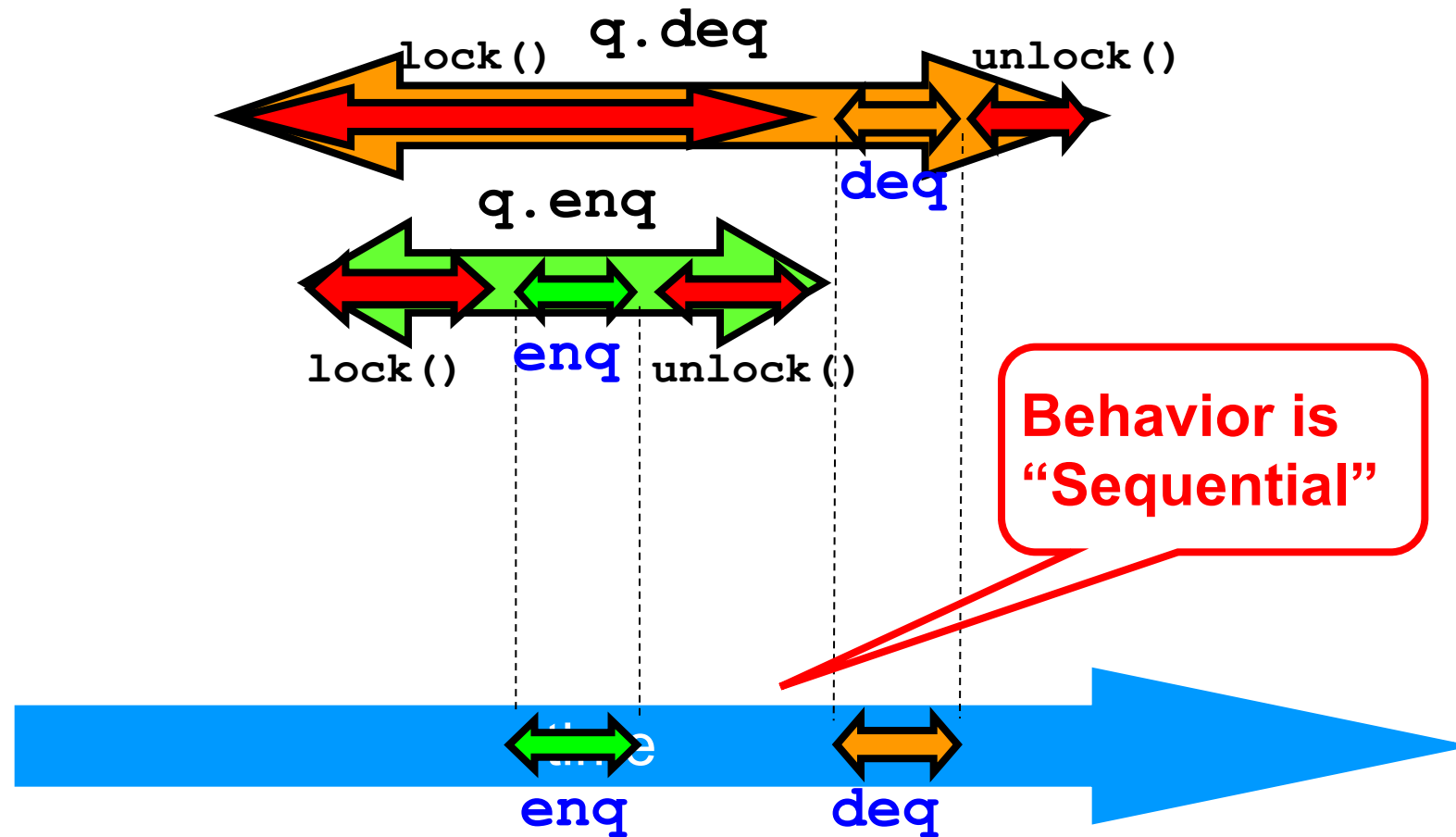
# Intuitively...

```
def dequeue() : T = {  
    myLock.lock()  
    try {  
        if (tail == head) {  
            throw EmptyException  
        }  
        val x = items(head % items.length)  
        head = head + 1  
        x  
    } finally {  
        myLock.unlock()  
    }  
}
```

All queue modifications  
are mutually exclusive

## Intuitively

Lets capture the idea of describing the concurrent via the sequential



# Linearizability

- Each method should
  - “take effect”
  - Instantaneously
  - Between invocation and response events
- Object is correct if this “sequential” behavior is correct
- Any such concurrent object is
  - **Linearizable**<sup>TM</sup>

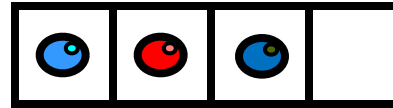
# Is it really about the object?

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

# Proving execution linearizable

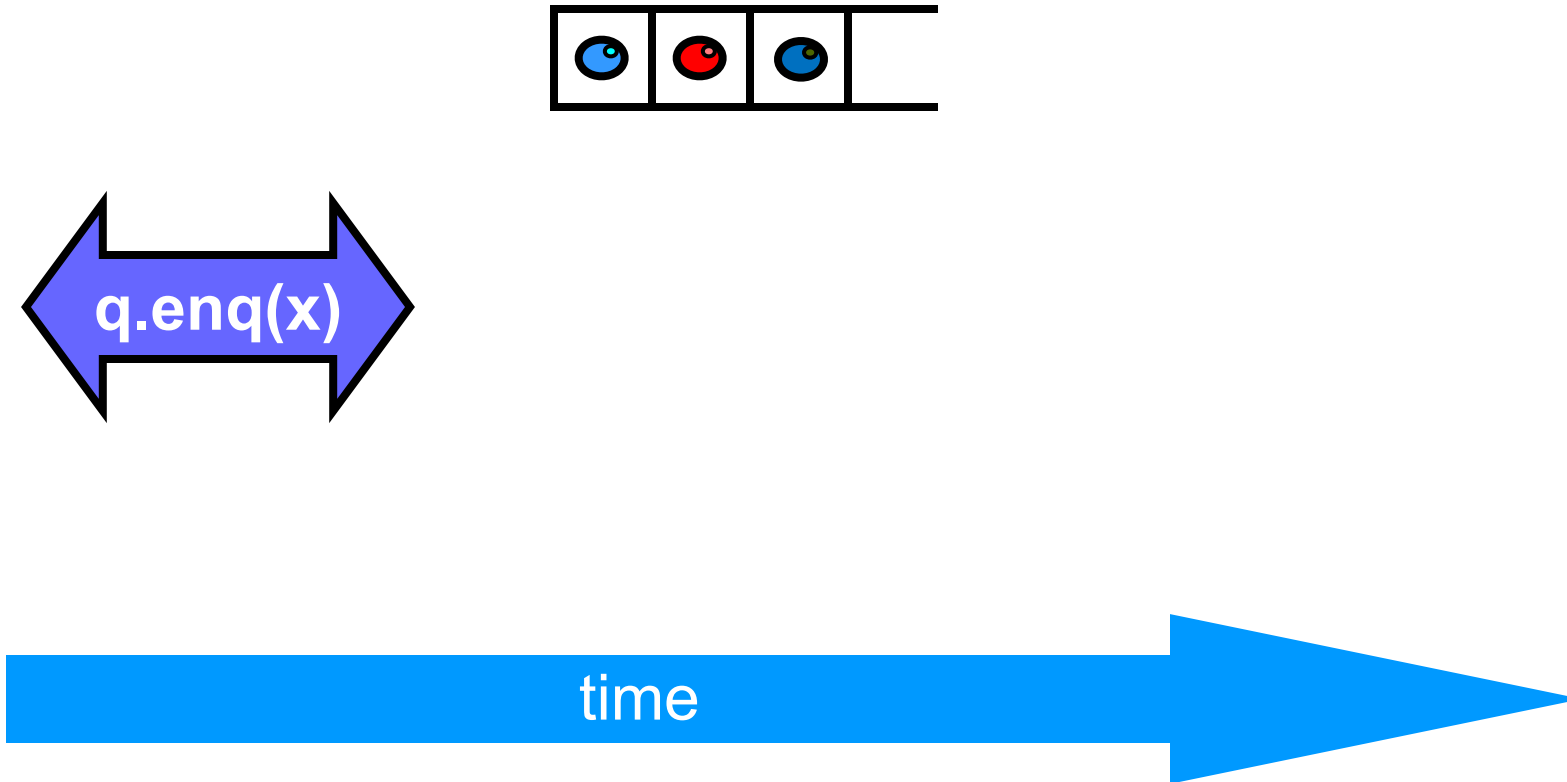
- Identify “linearization points”
  - Between invocation and response events
  - Correspond to the effect of the call
  - “Justify” the whole execution
- Multiple ways to identify linearization points exist
- If none found, execution is *non-linearizable*

# Example

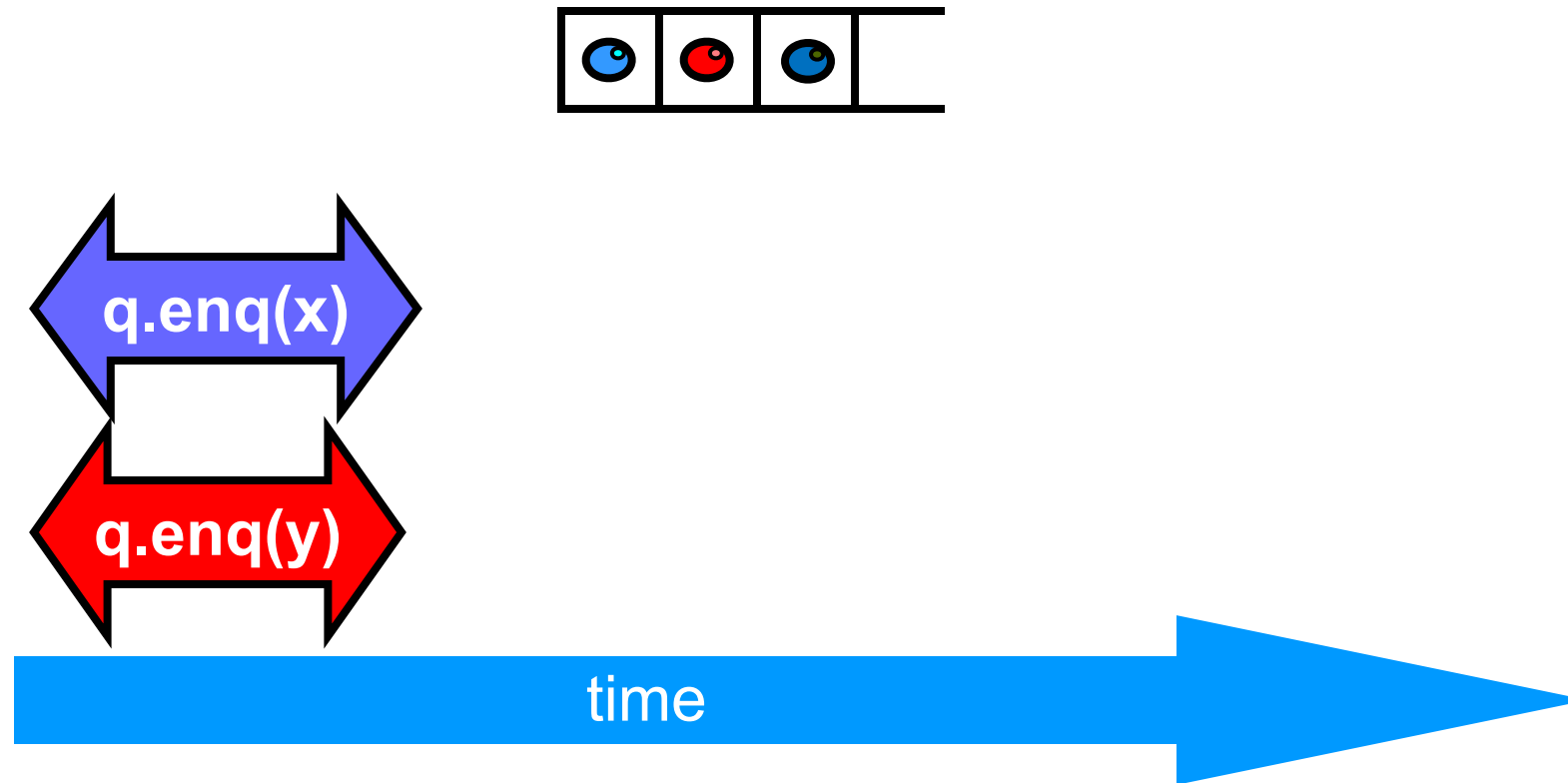




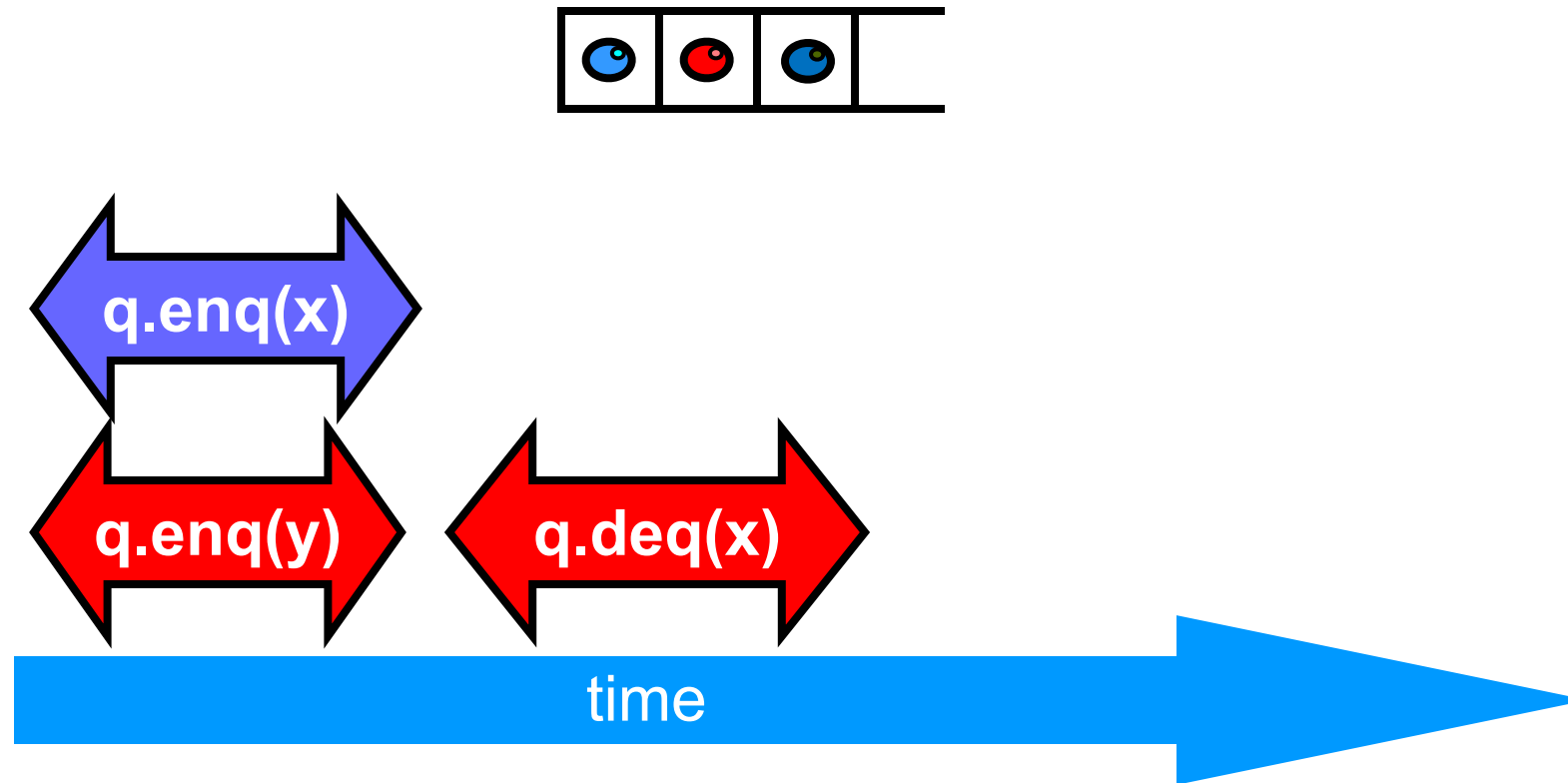
# Example



# Example

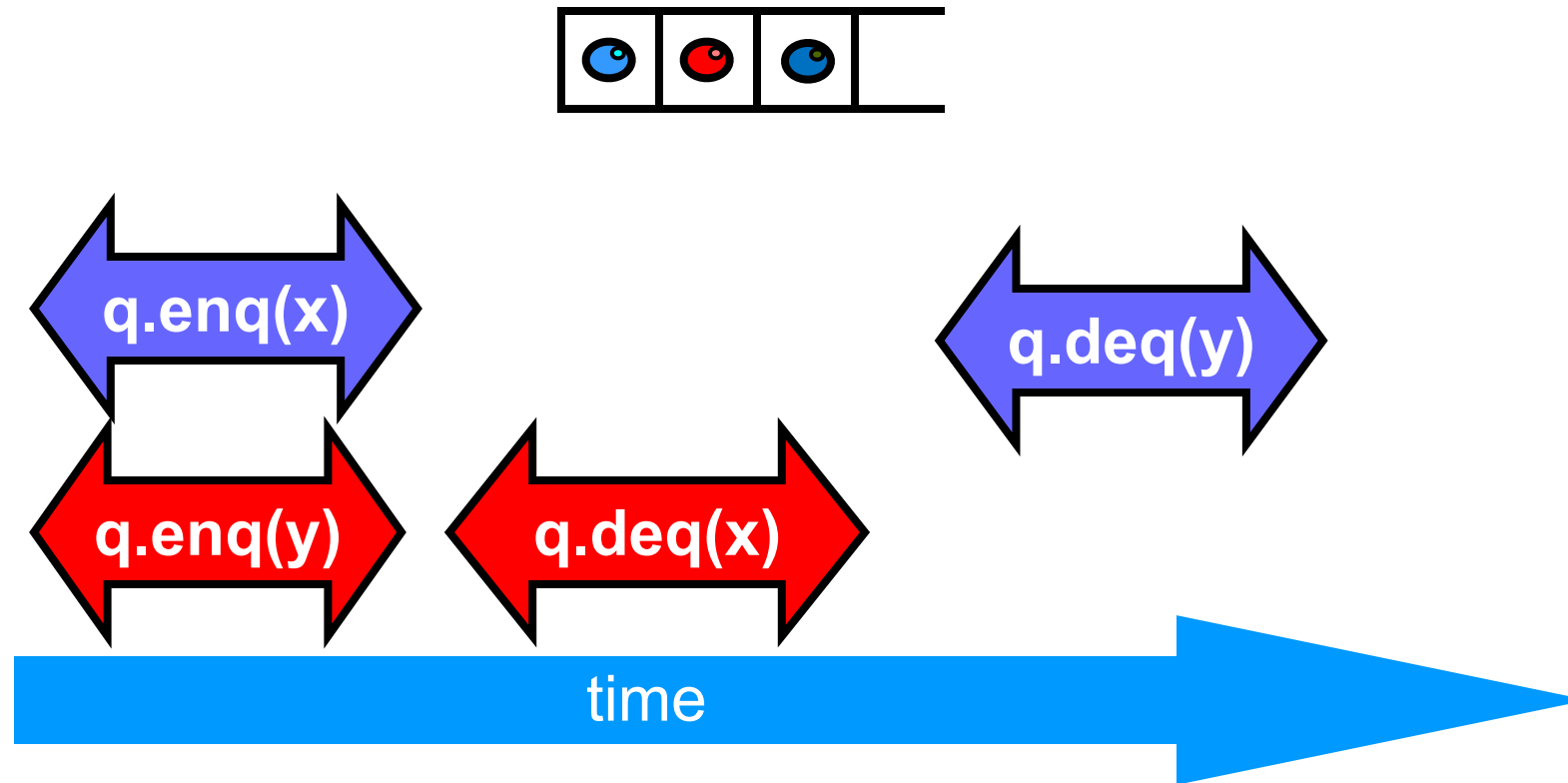


# Example

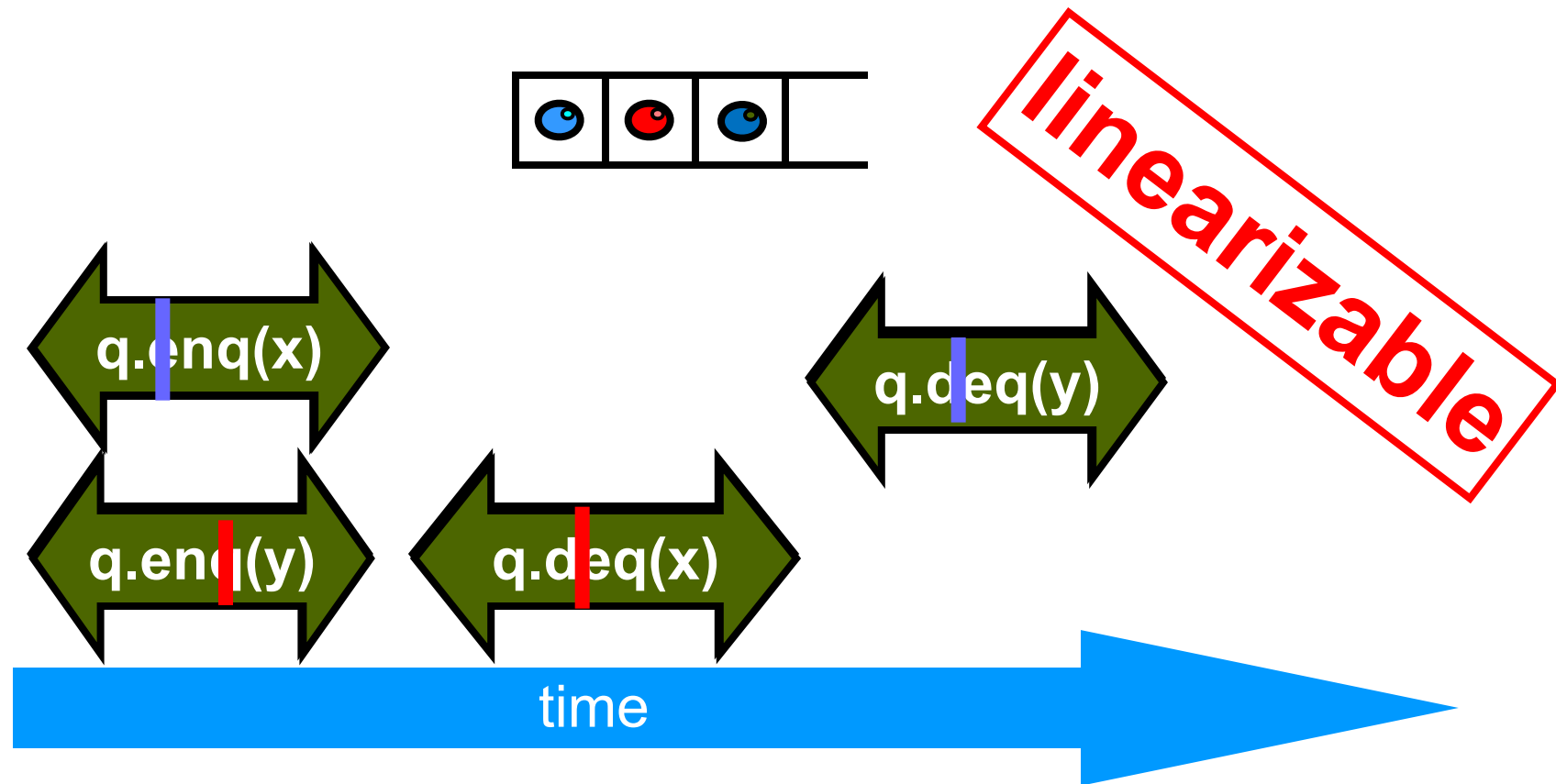




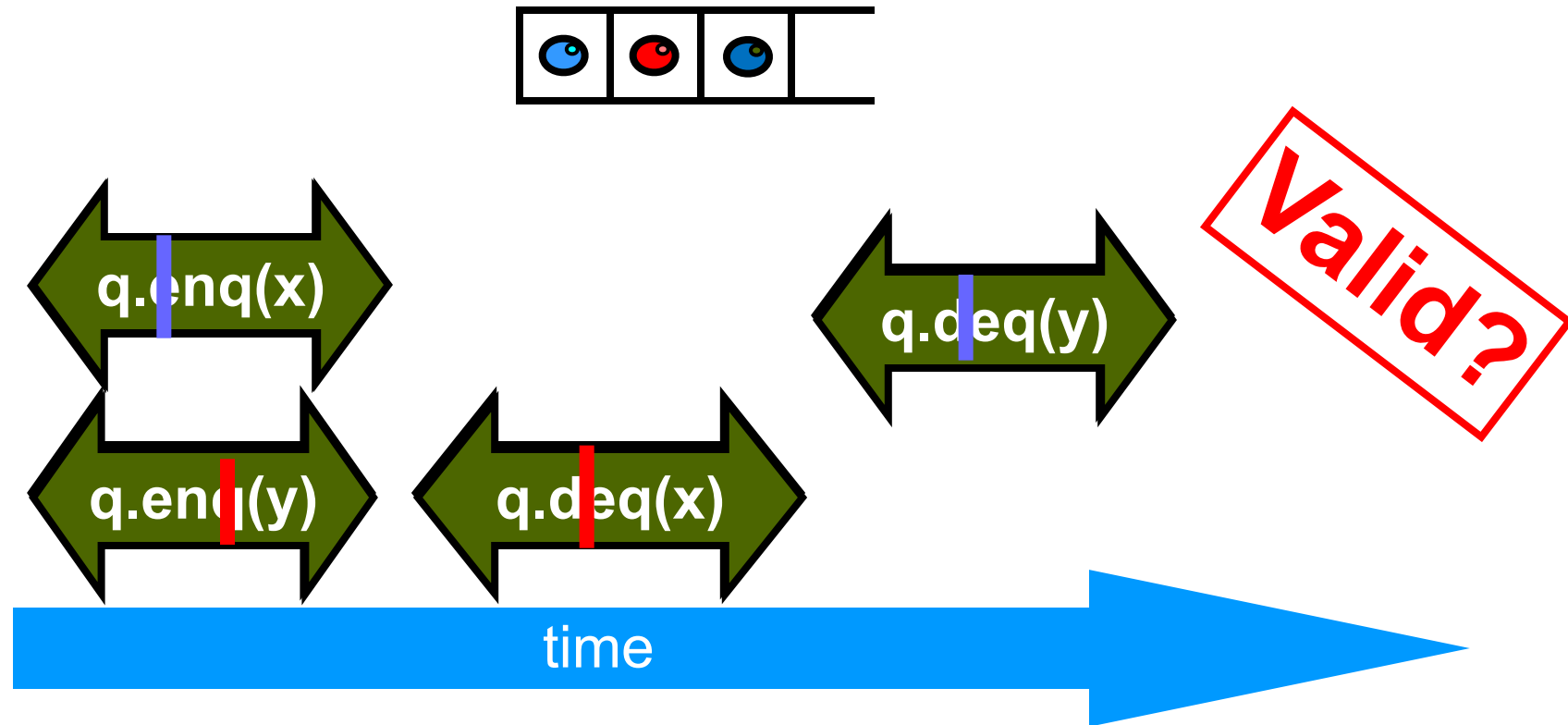
# Example



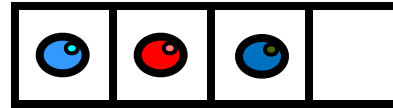
# Example



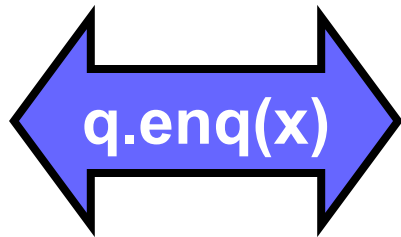
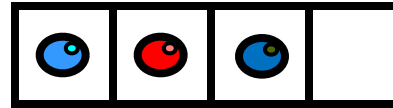
# Example



# Example

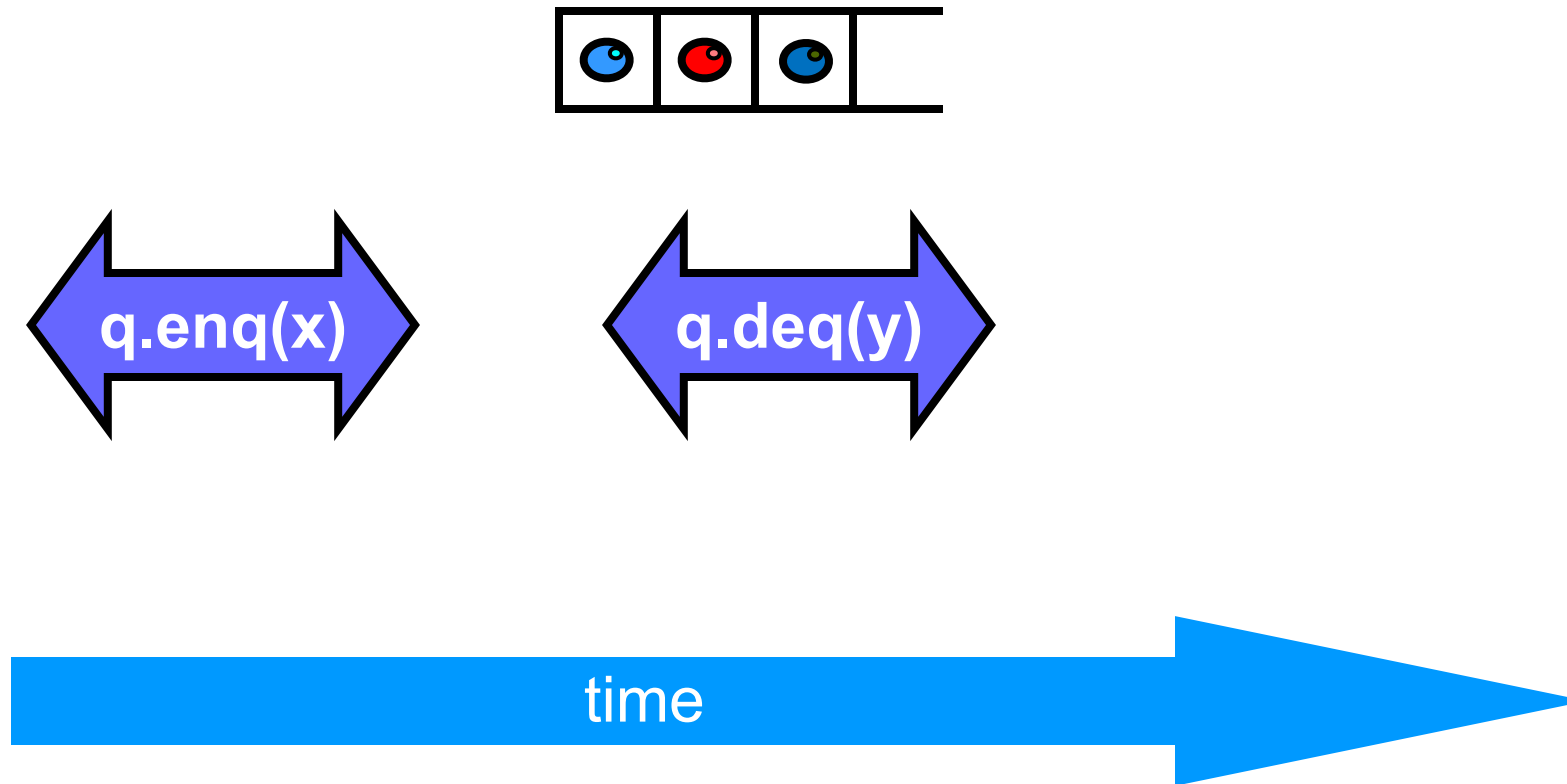


# Example



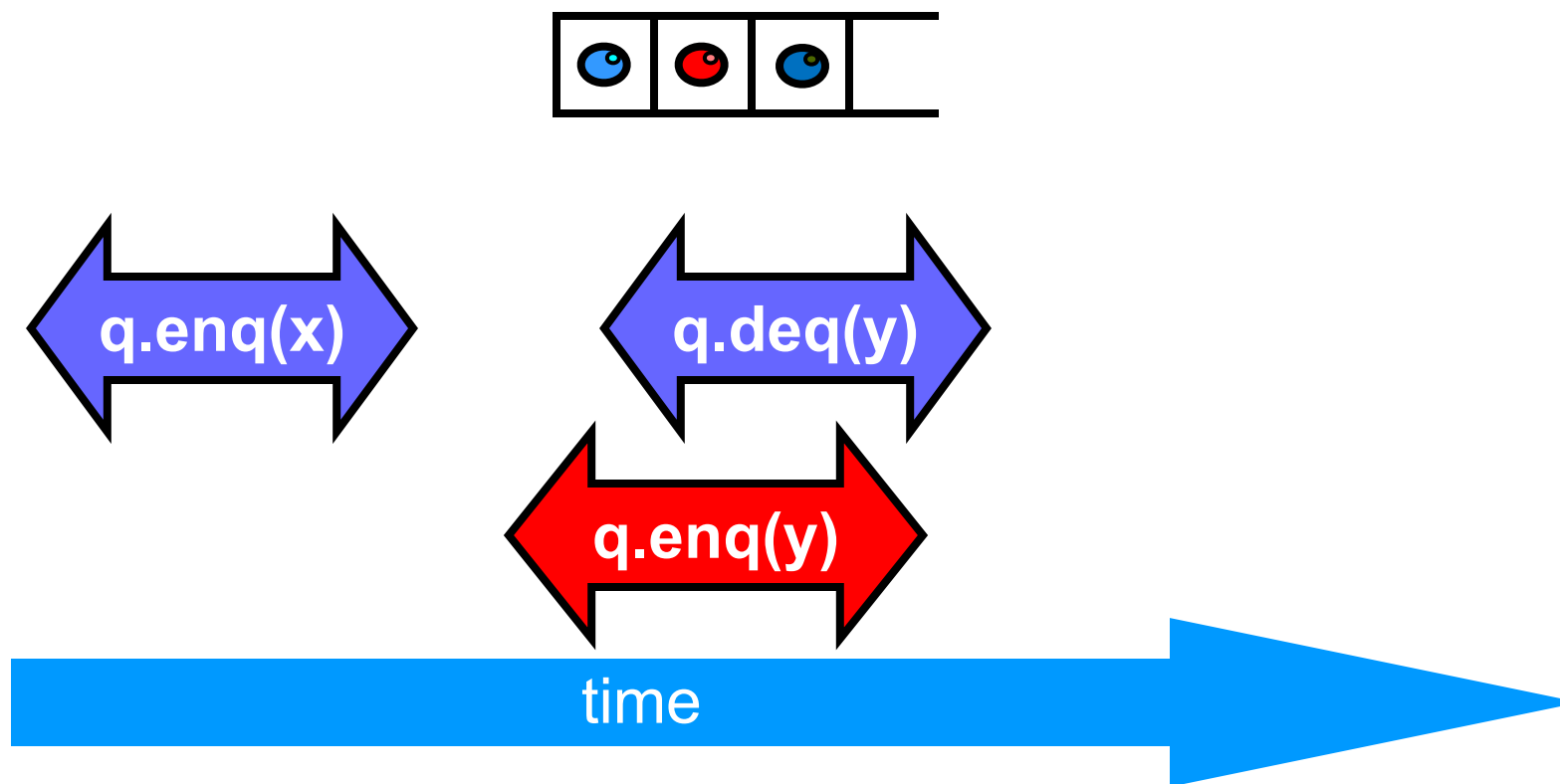


# Example



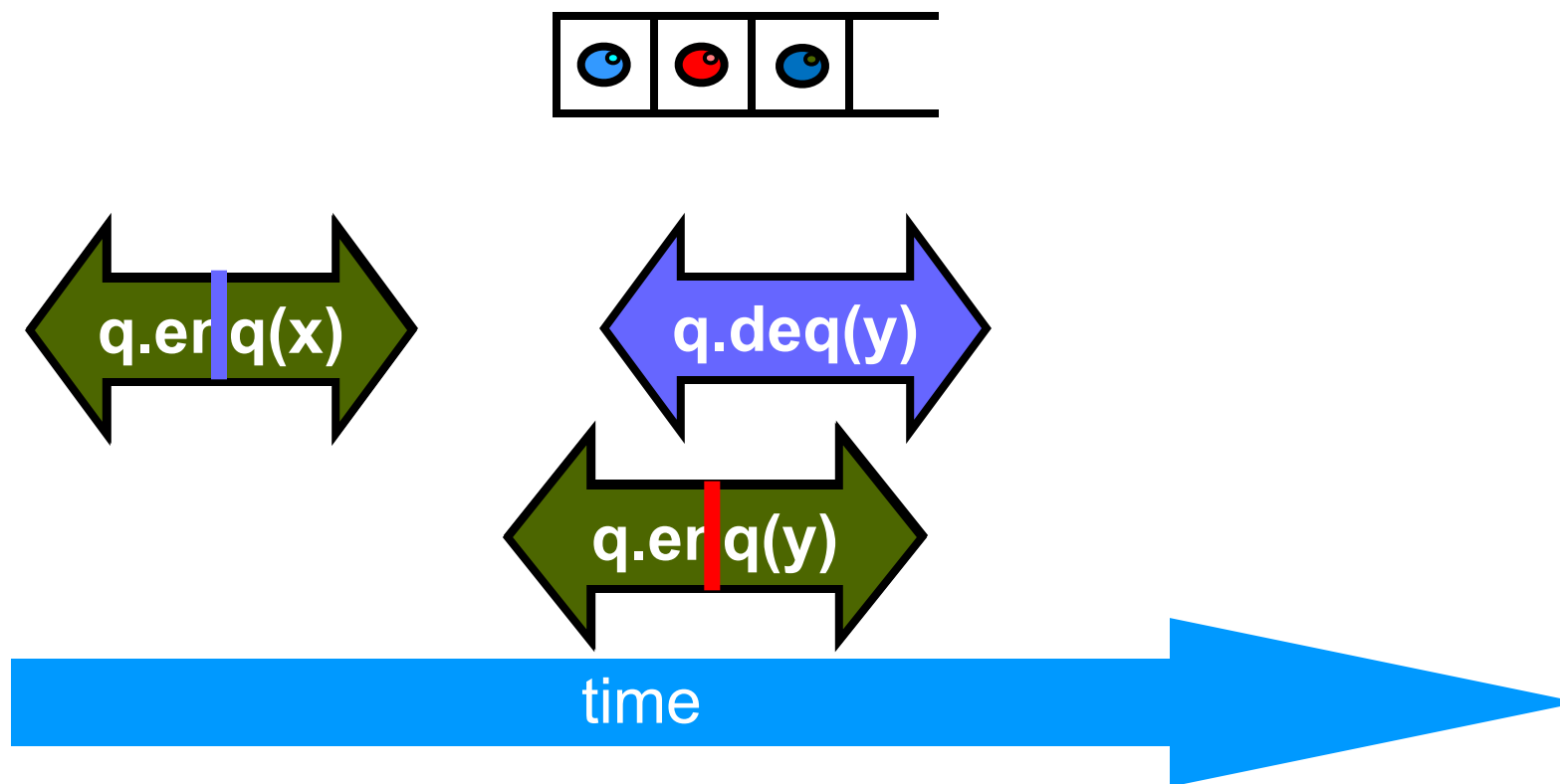


# Example



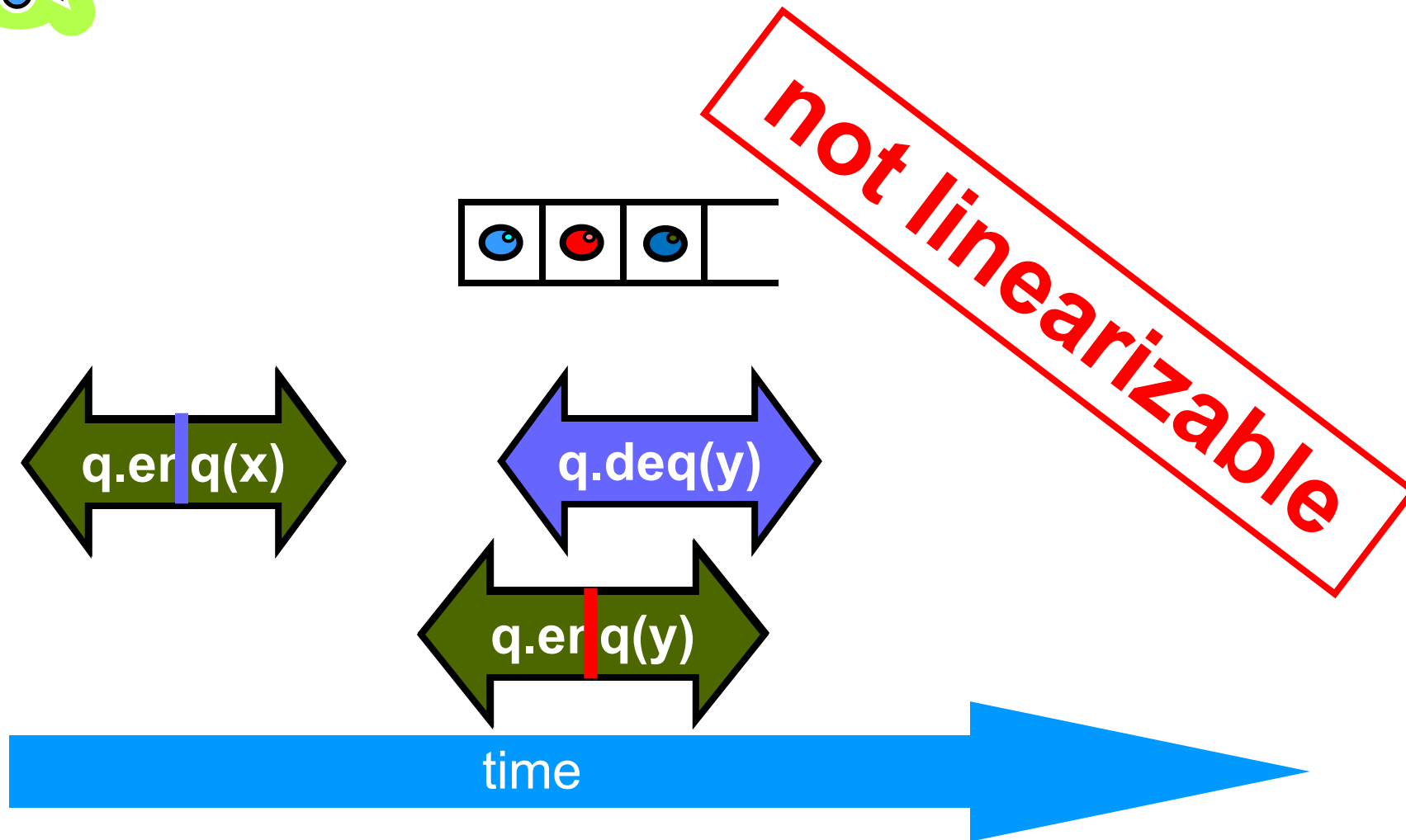


# Example

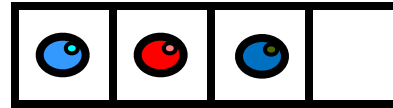




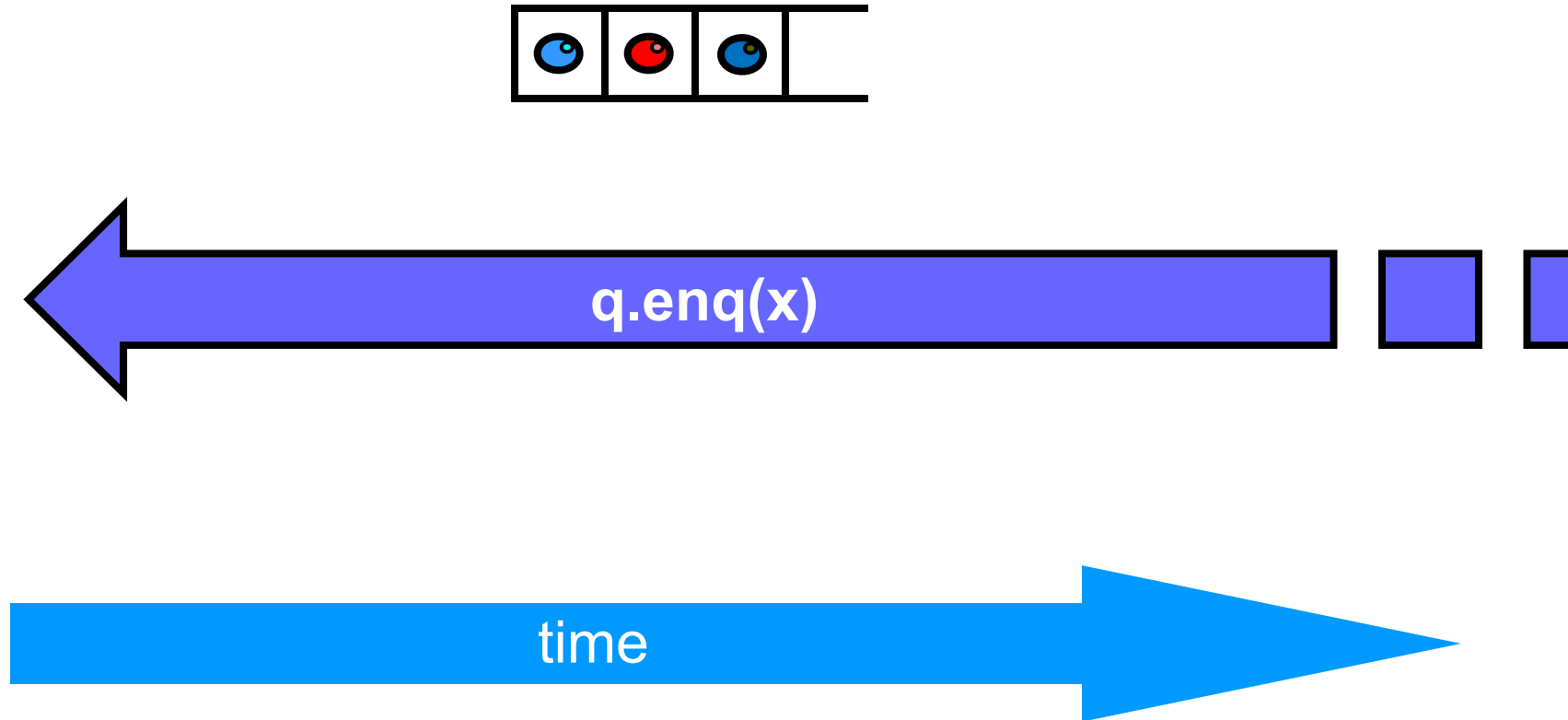
# Example



# Example

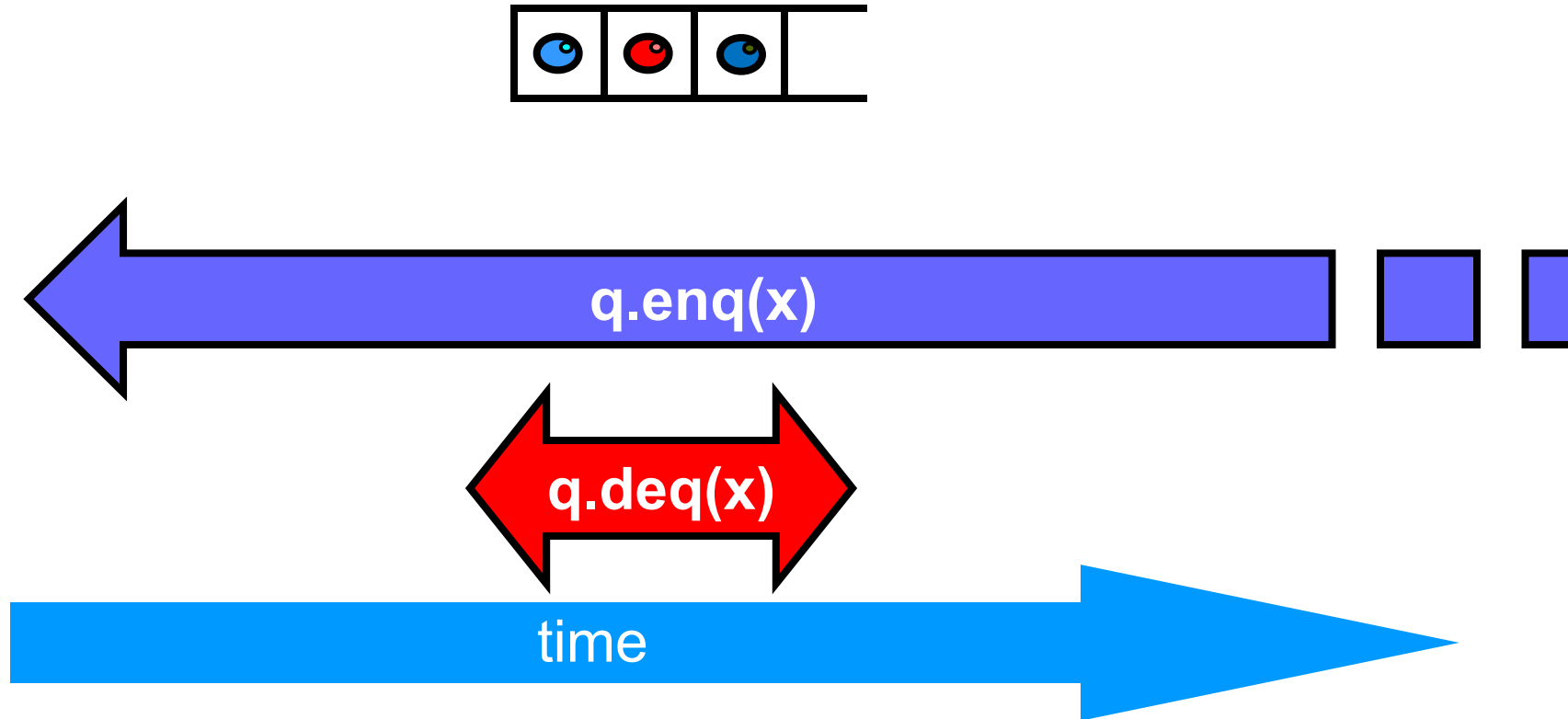


# Example



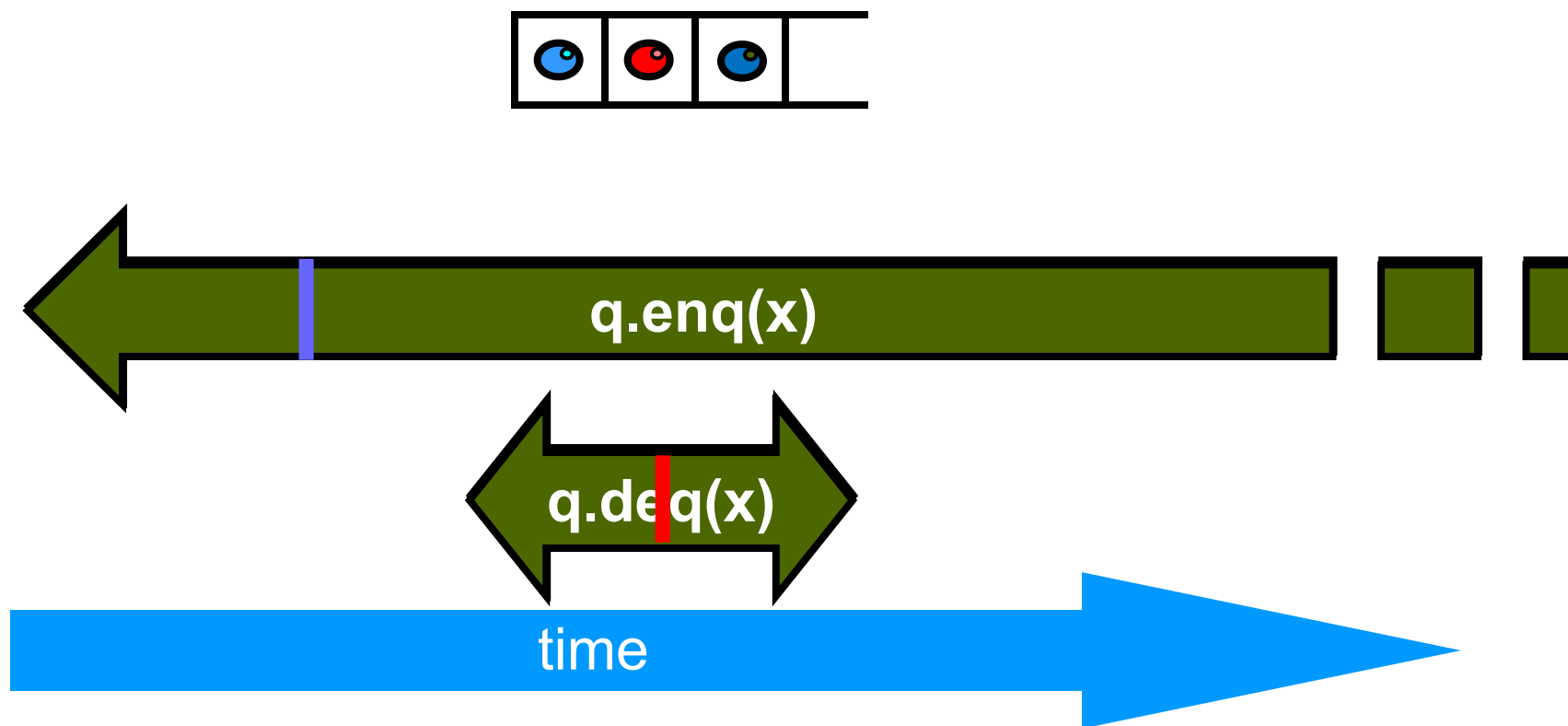


# Example





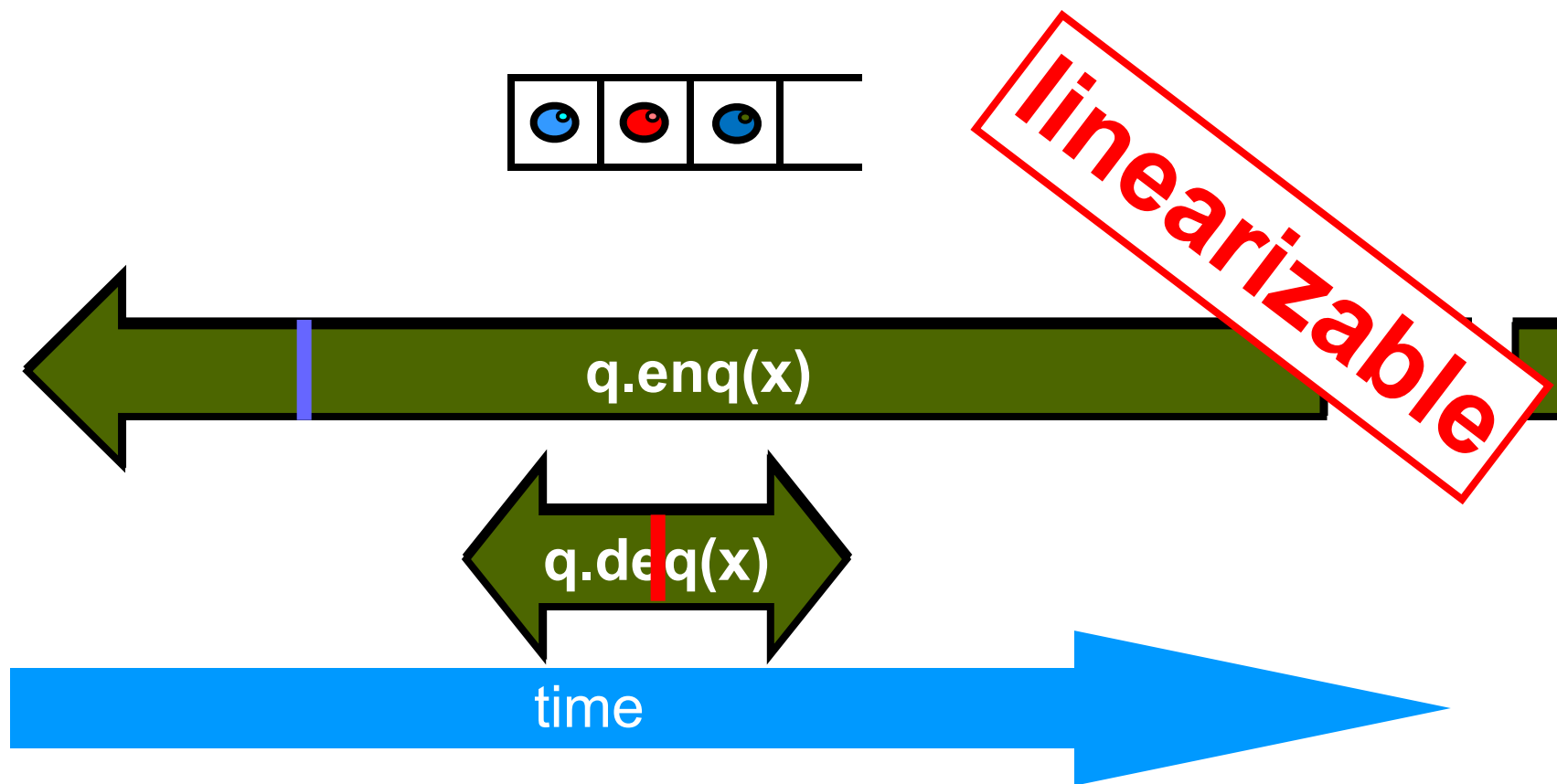
# Example



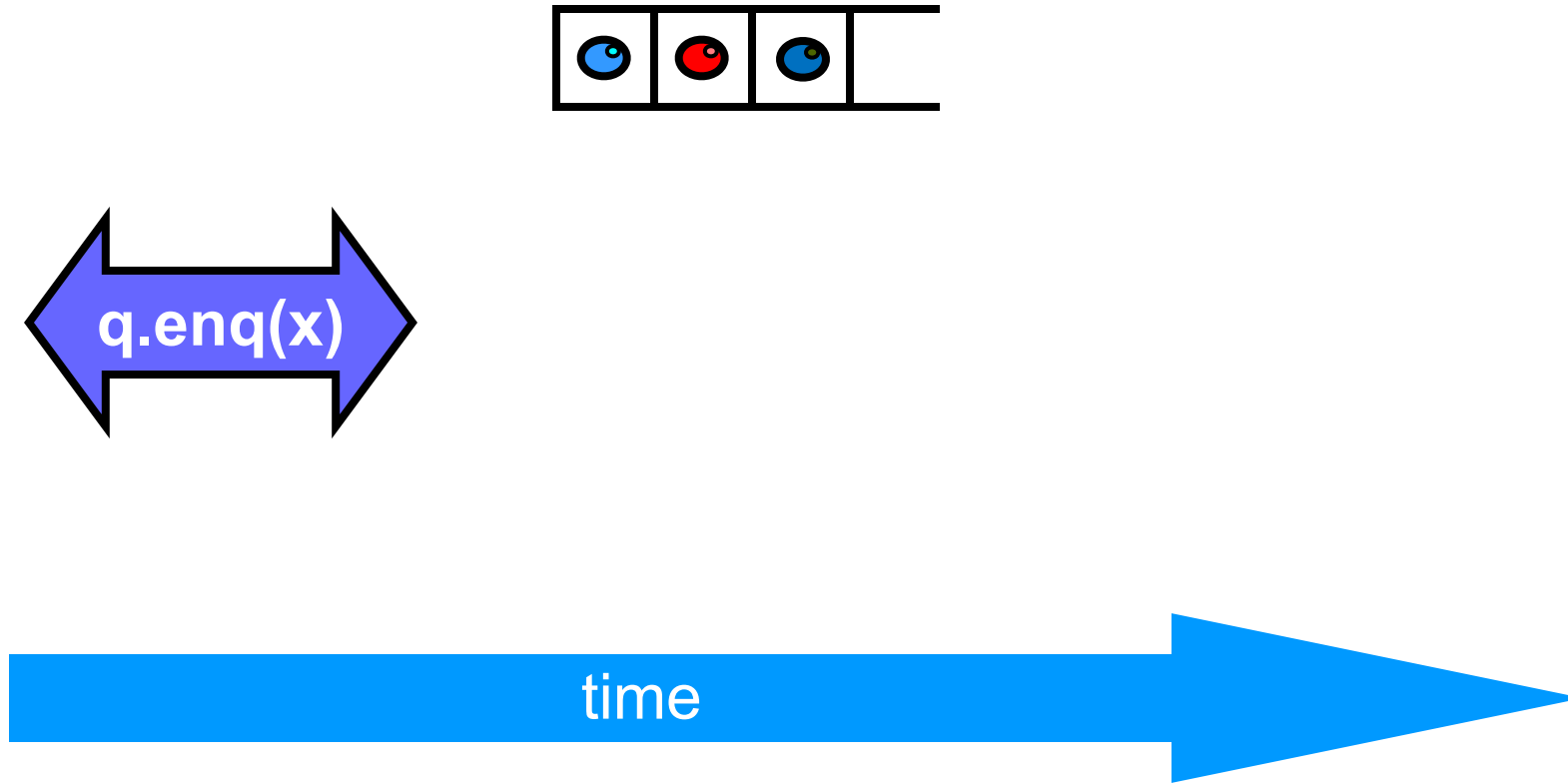




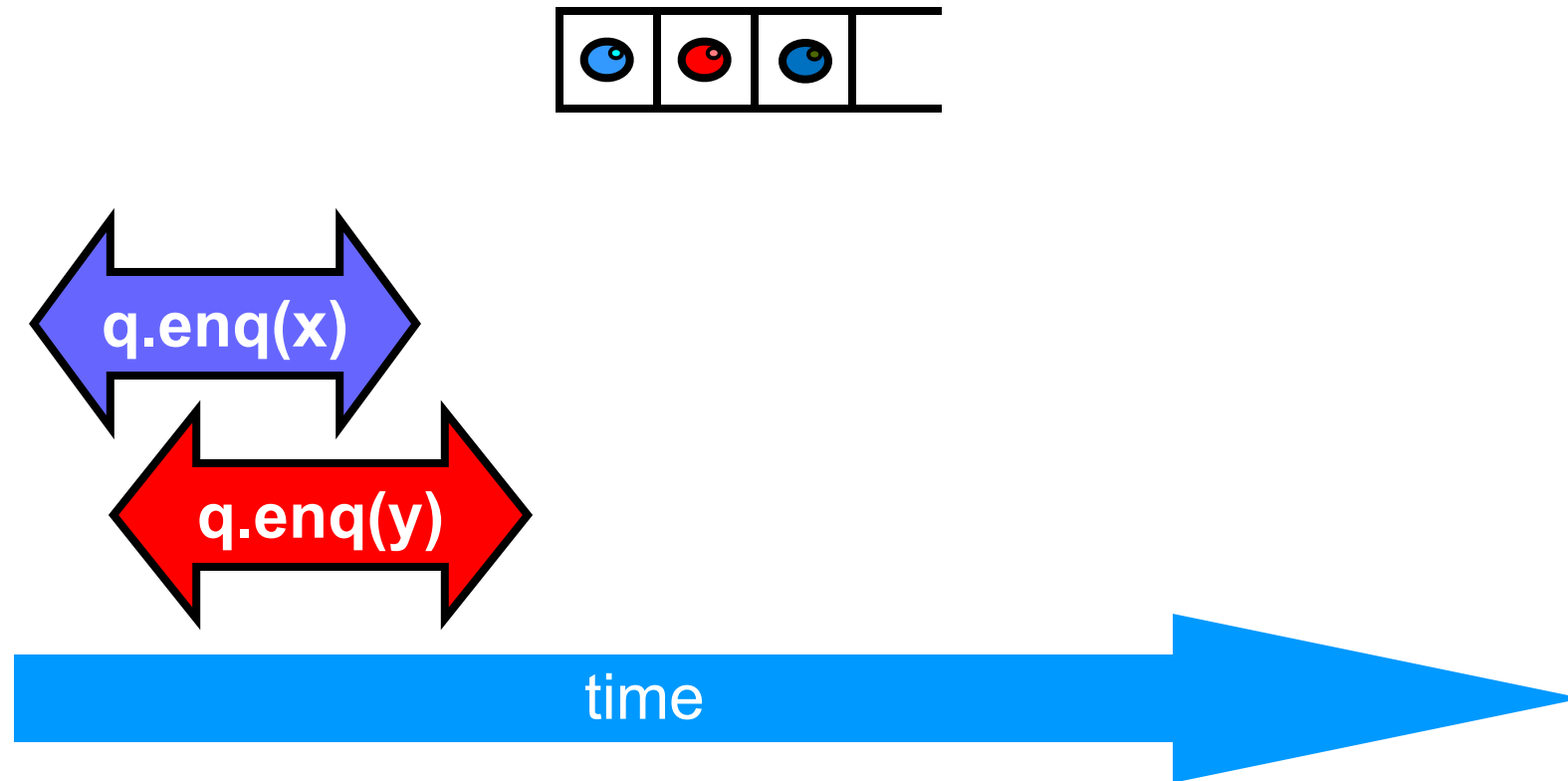
# Example



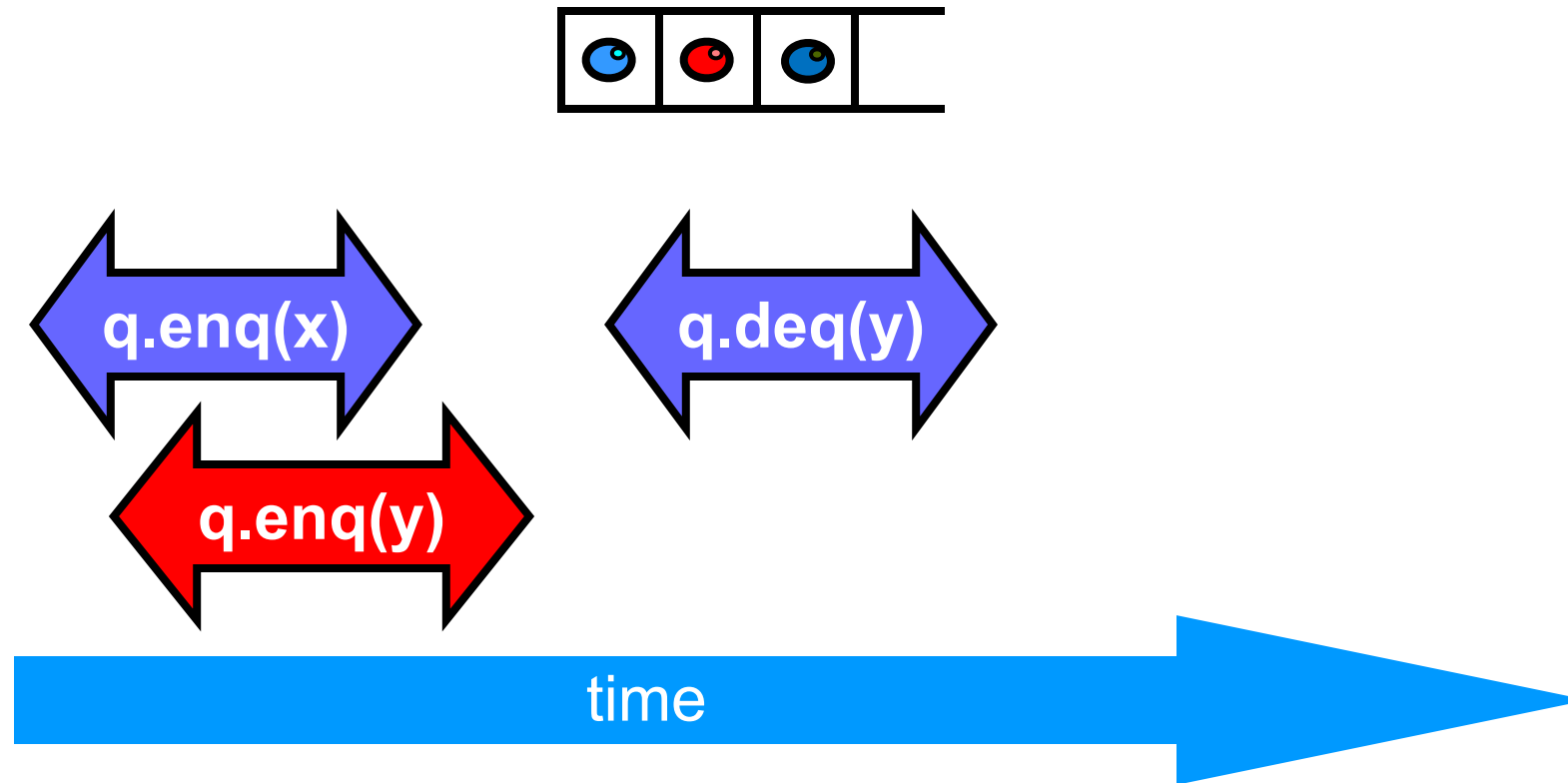
# Example



# Example

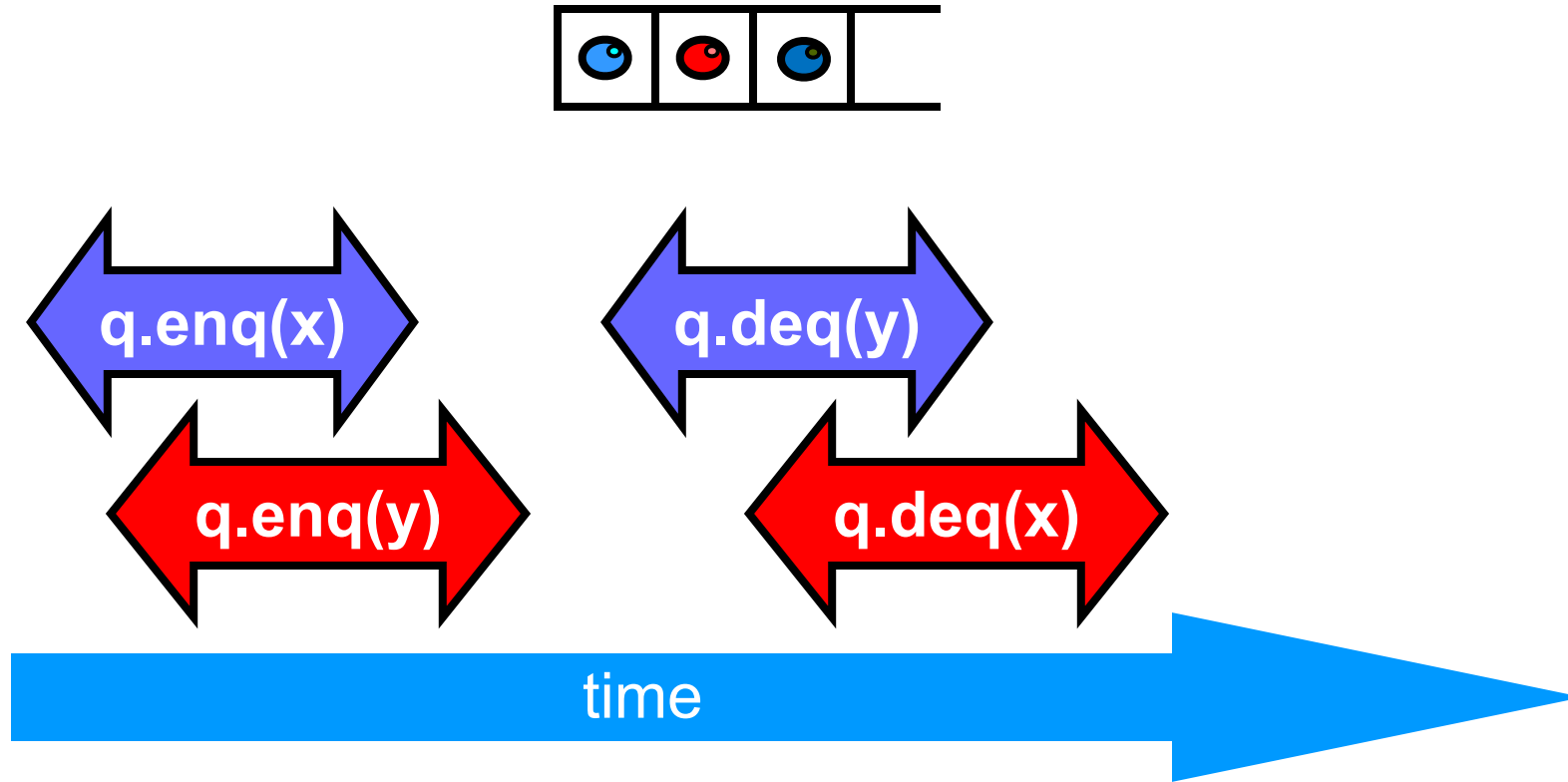


# Example



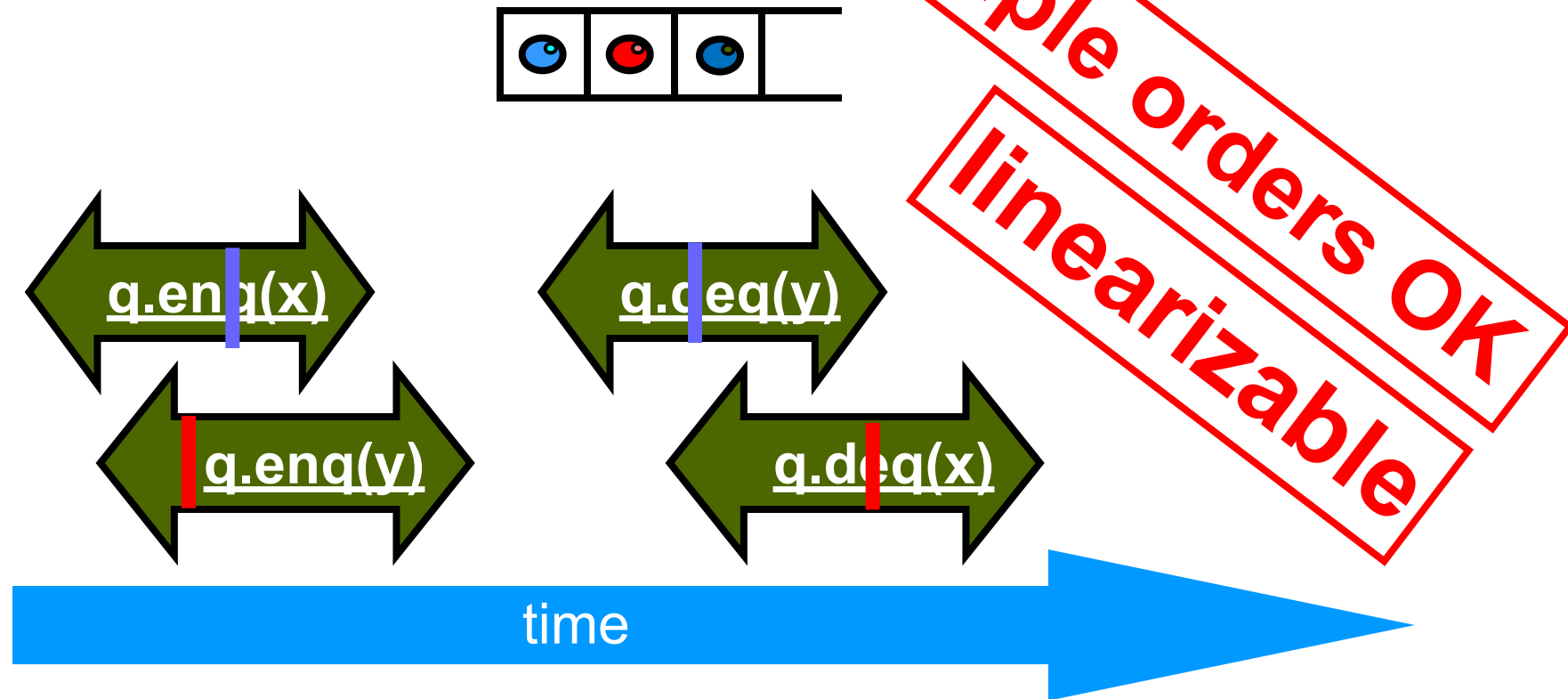


# Example

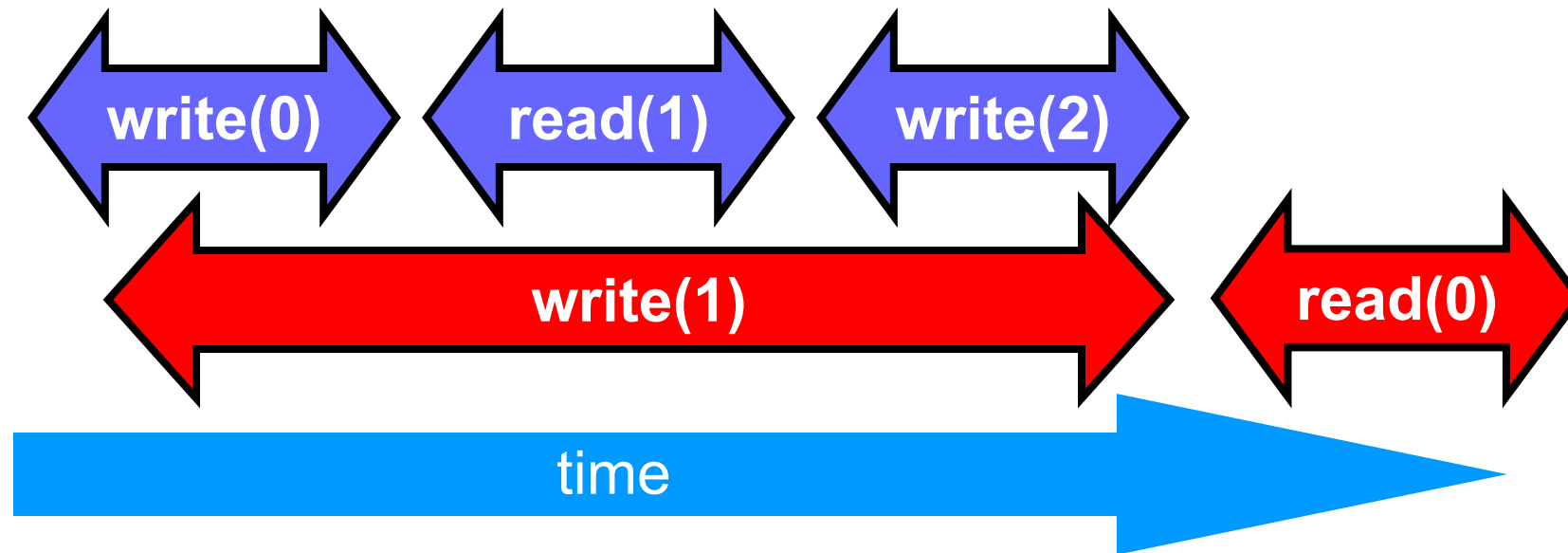


Comme ci  
Comme ça

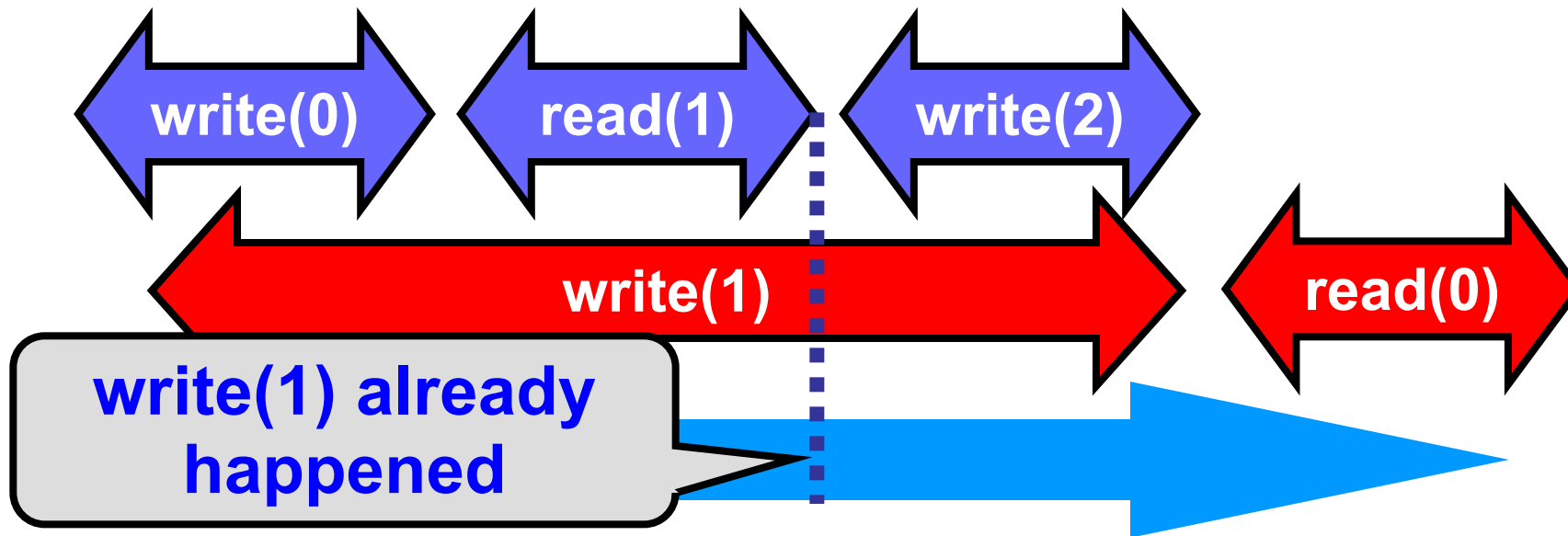
Example



# Read/Write Register Example

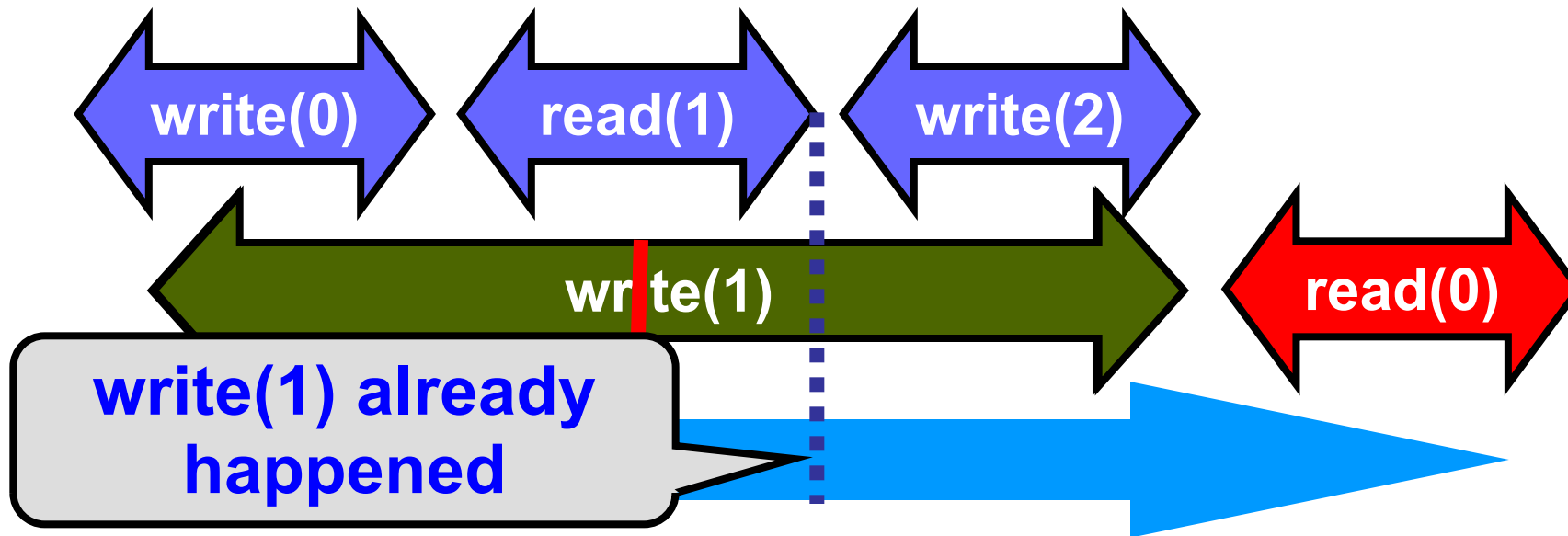


# Read/Write Register Example

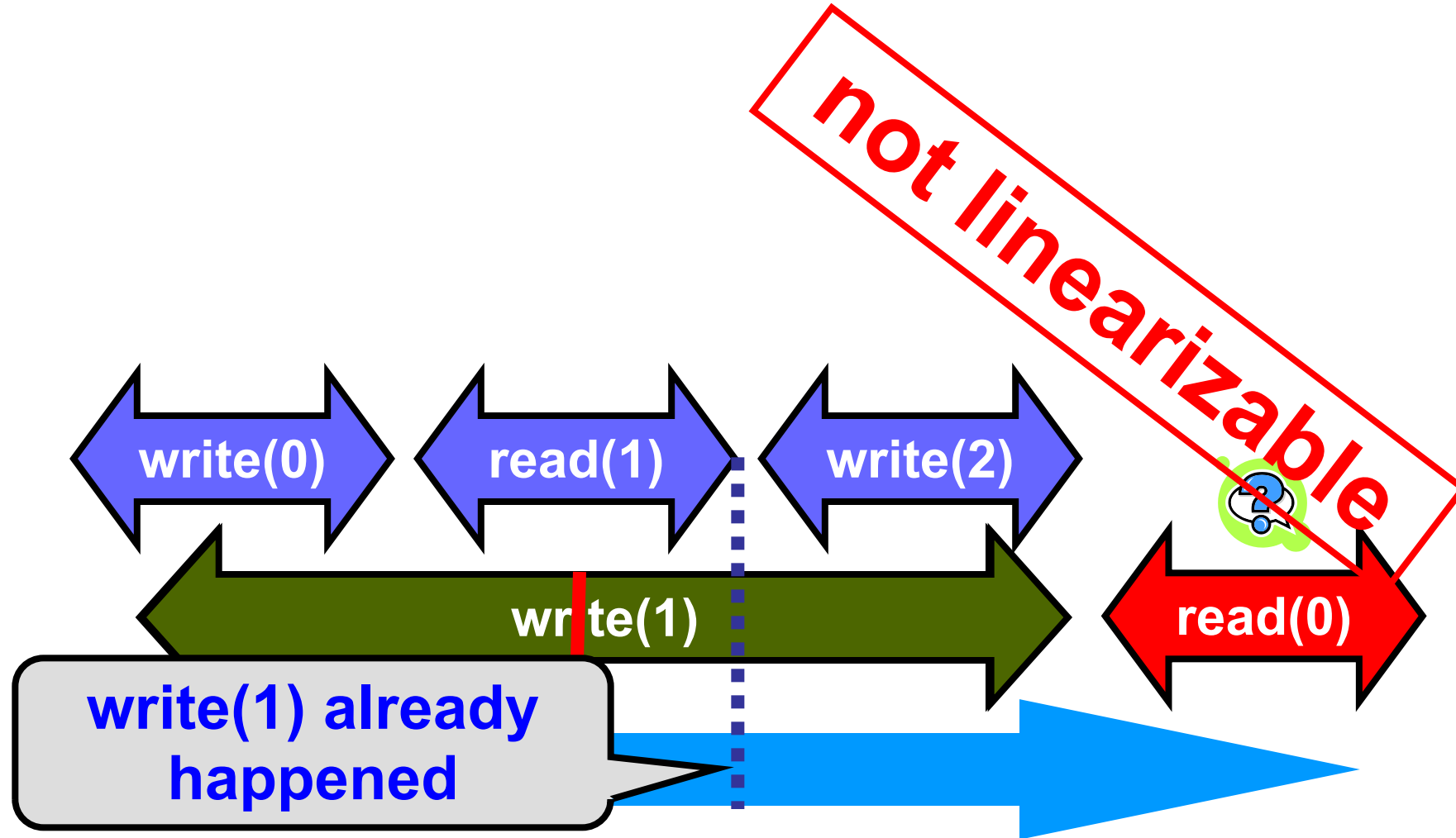




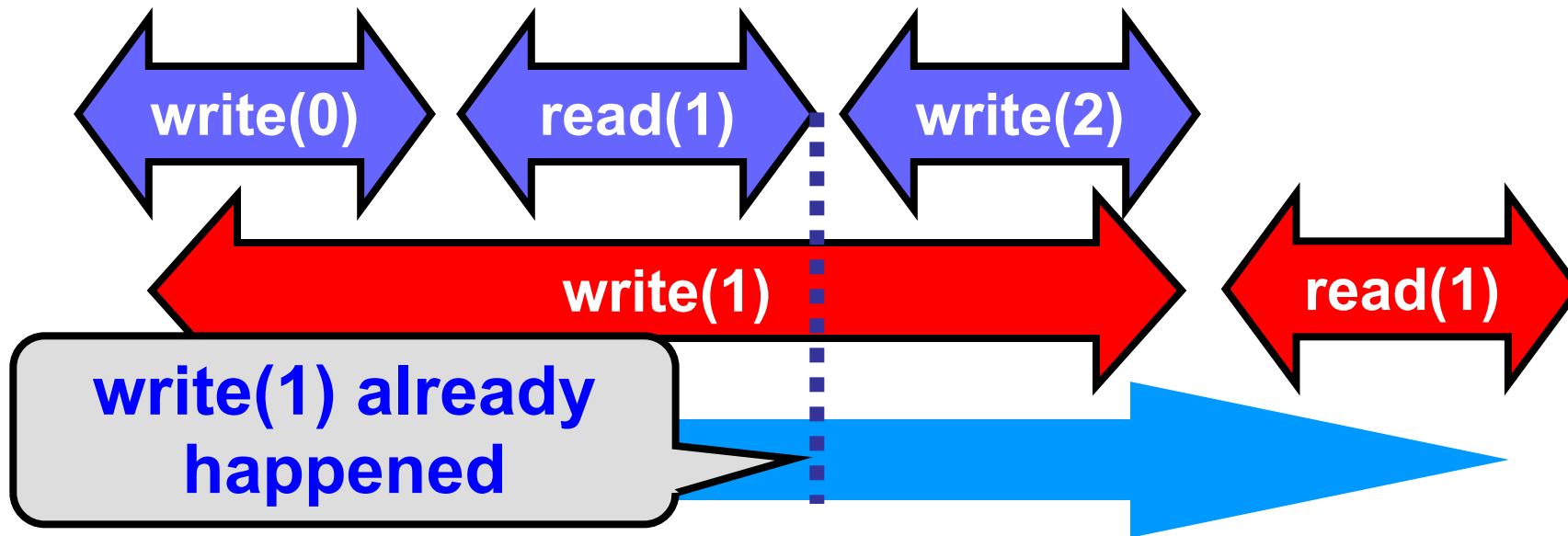
# Read/Write Register Example



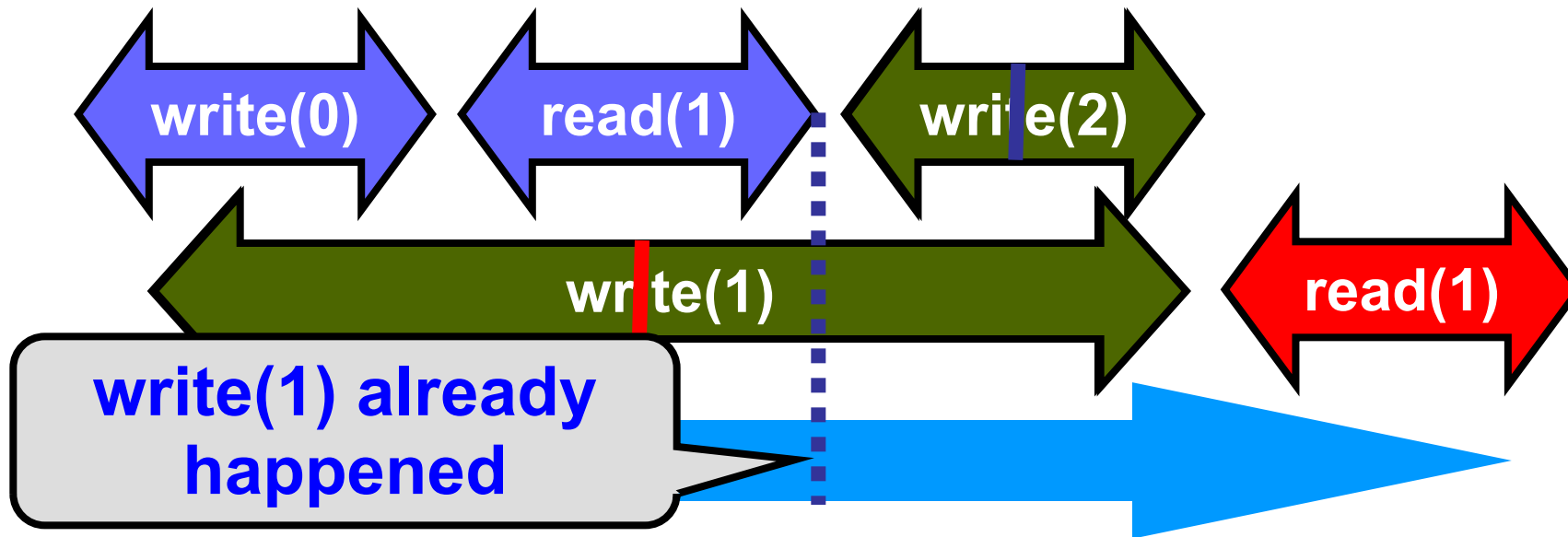
# Read/Write Register Example



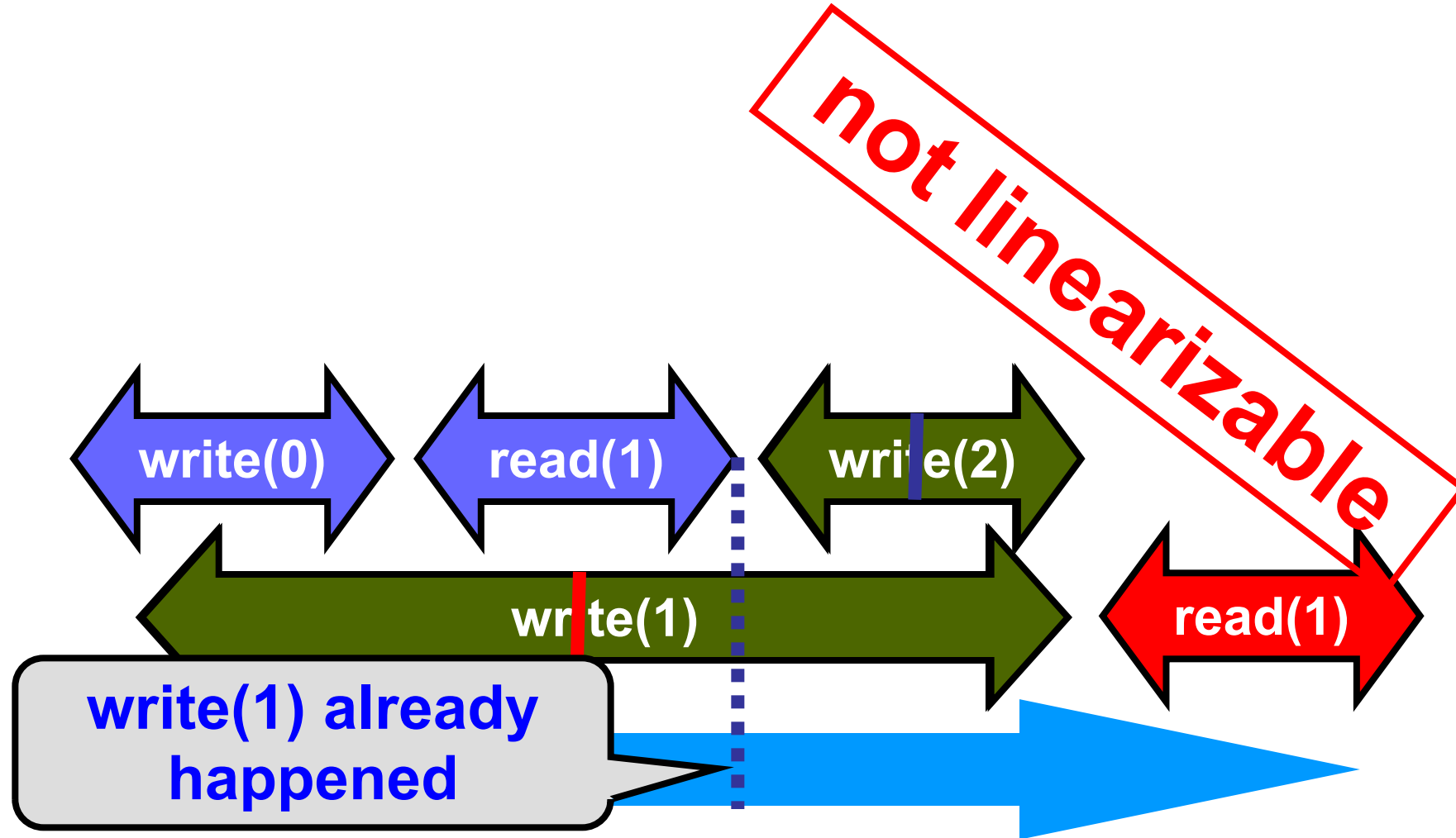
# Read/Write Register Example



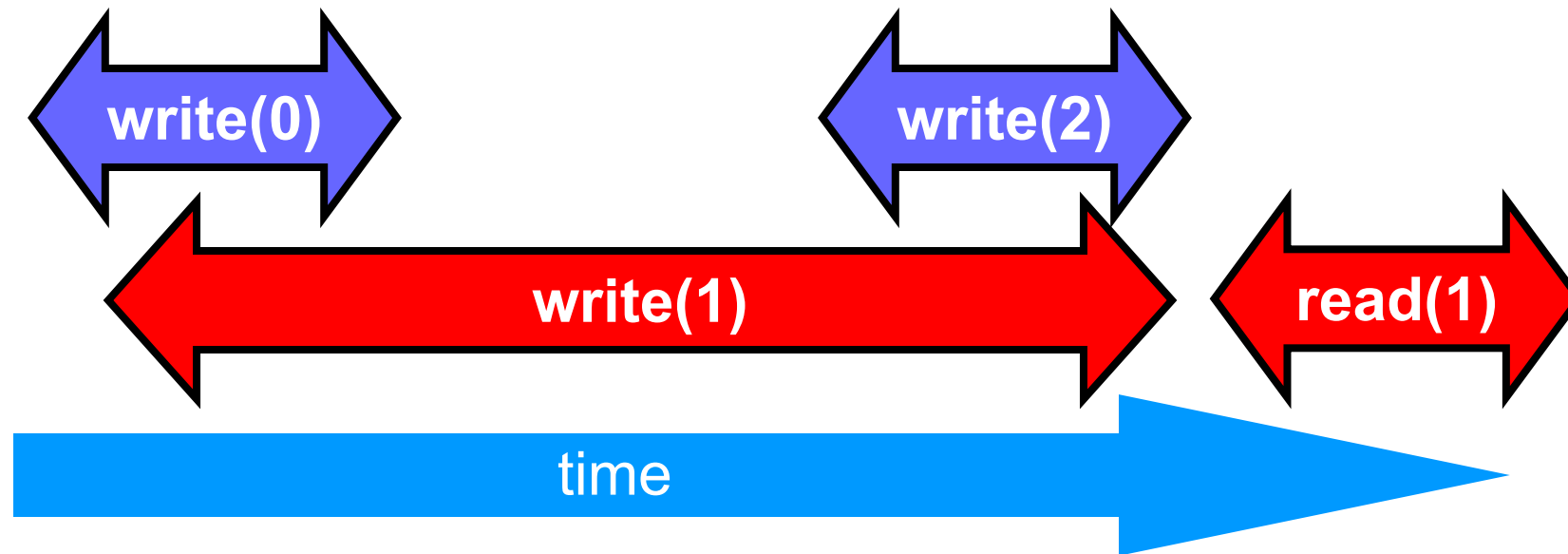
# Read/Write Register Example



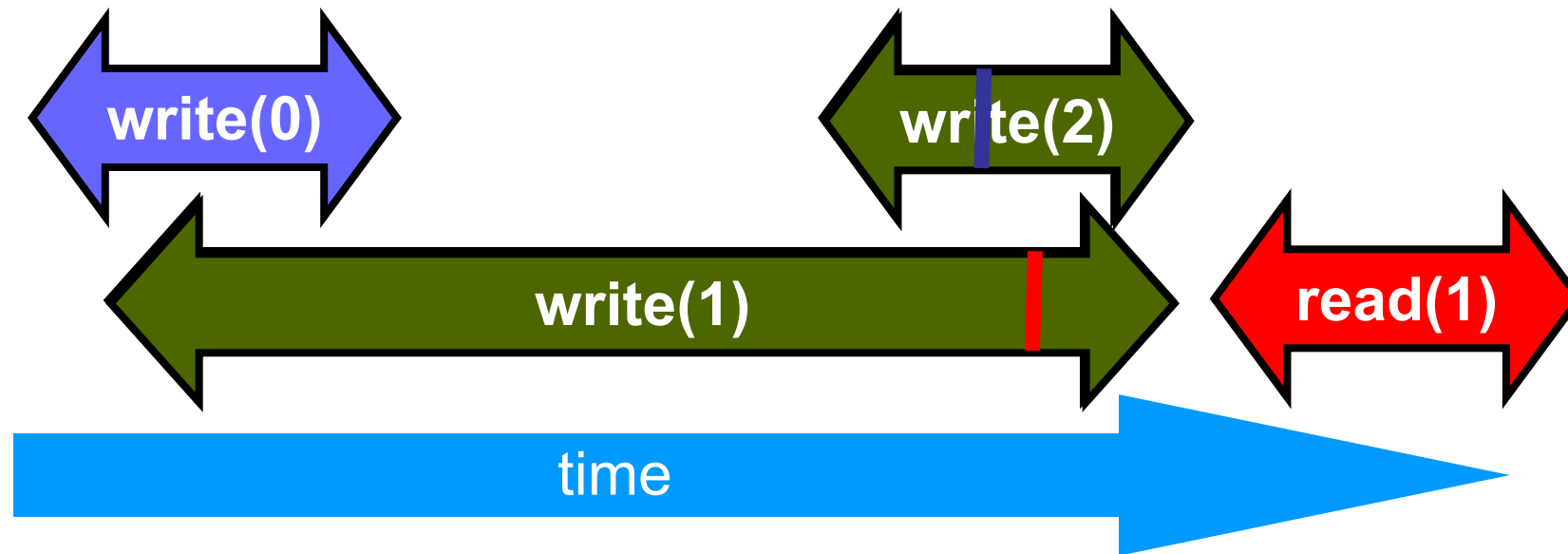
# Read/Write Register Example



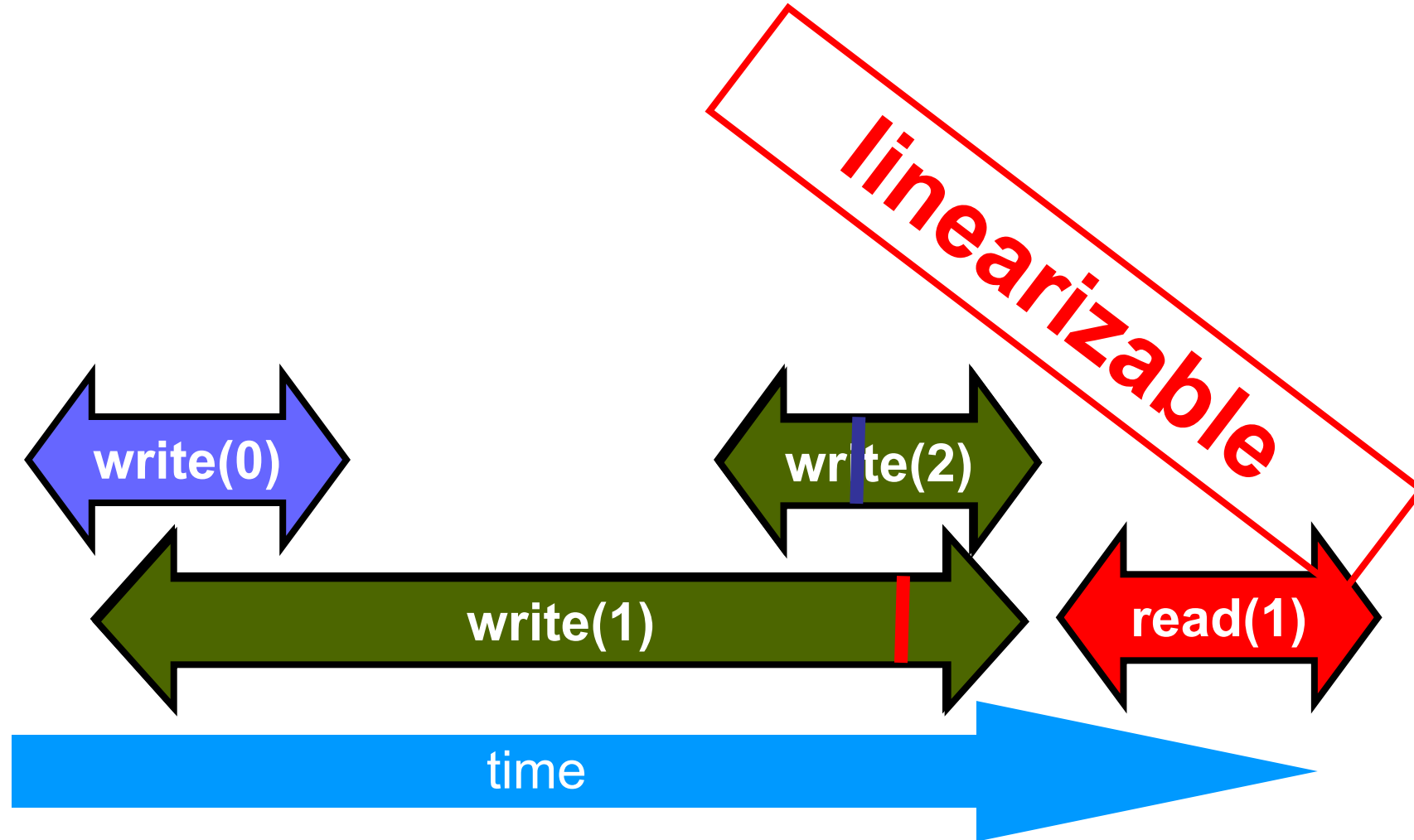
# Read/Write Register Example



# 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***

**Let's stop here for today**

**Next lecture:**  
formal model for linearizability

# Formal Model of Executions

- Define precisely what we mean
  - Ambiguity is bad when intuition is weak
- Allow reasoning
  - Formal
  - But mostly informal
    - In the long run, actually more important

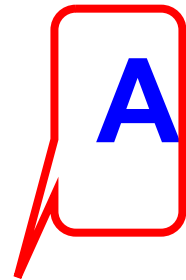
# Split Method Calls into Two Events

- 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**

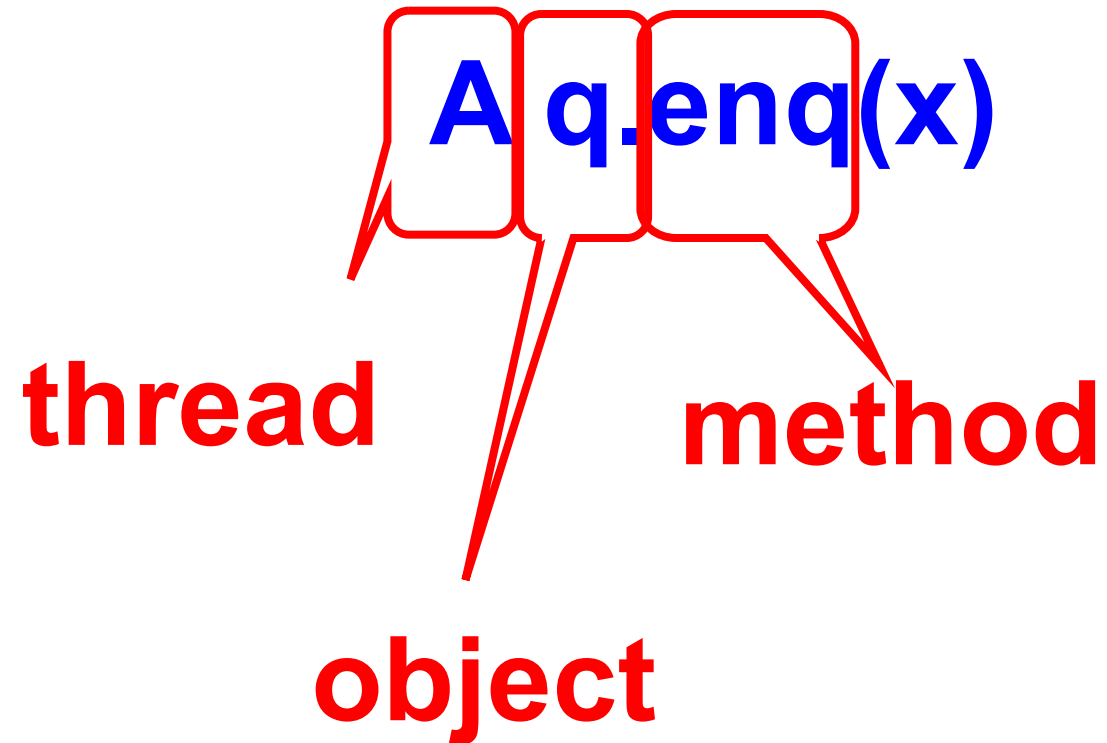
# Invocation Notation

**A** **q.enqueue(x)**

**thread** **method**

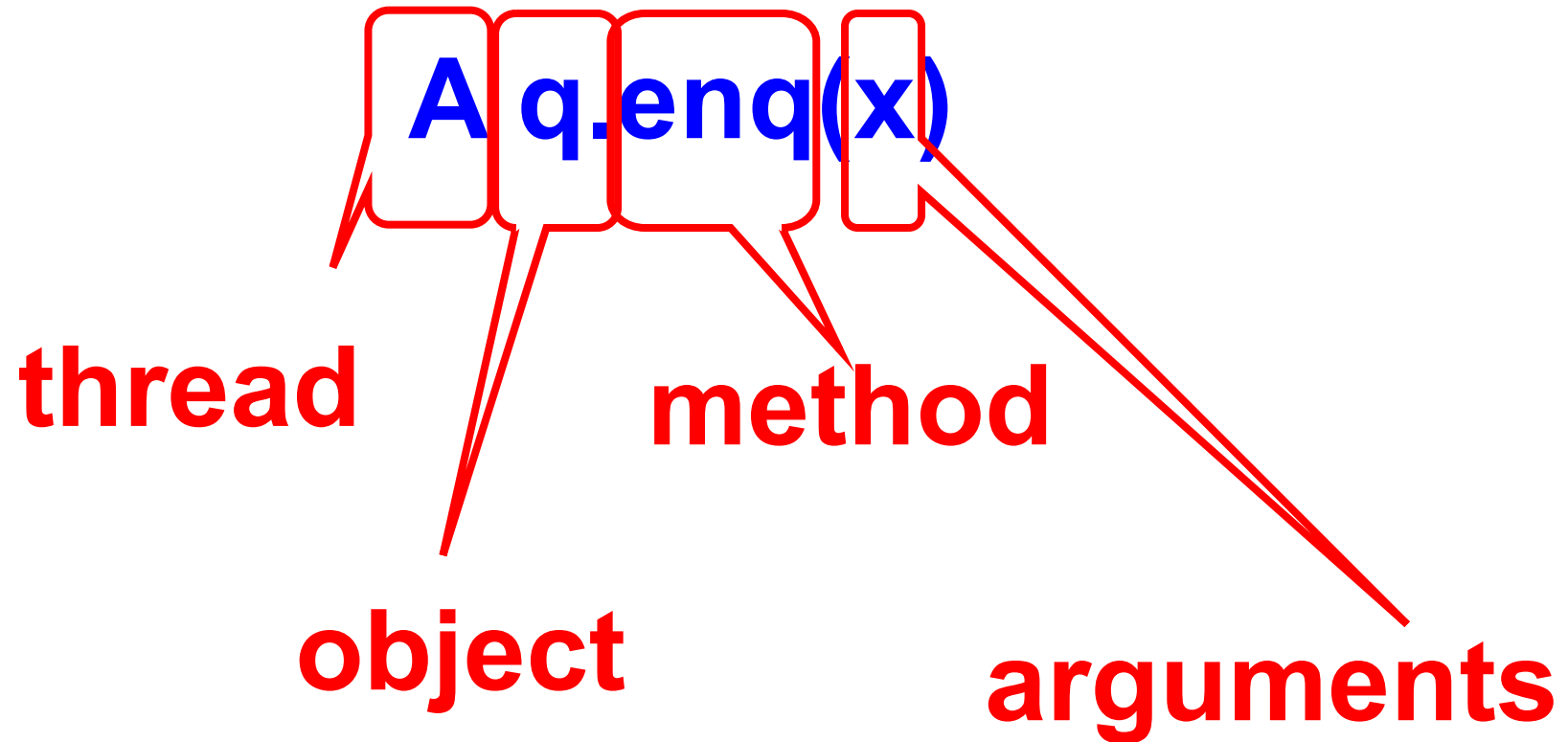
The diagram illustrates the components of the invocation notation **A.q.enqueue(x)**. The expression is written in blue. A red callout box points to the **A**, which is labeled **thread** in red text below it. Another red callout box points to the **enqueue(x)** part, which is labeled **method** in red text below it.

# Invocation Notation





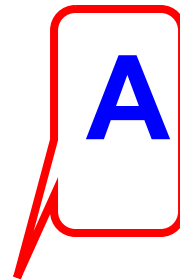
# Invocation Notation



# Response Notation

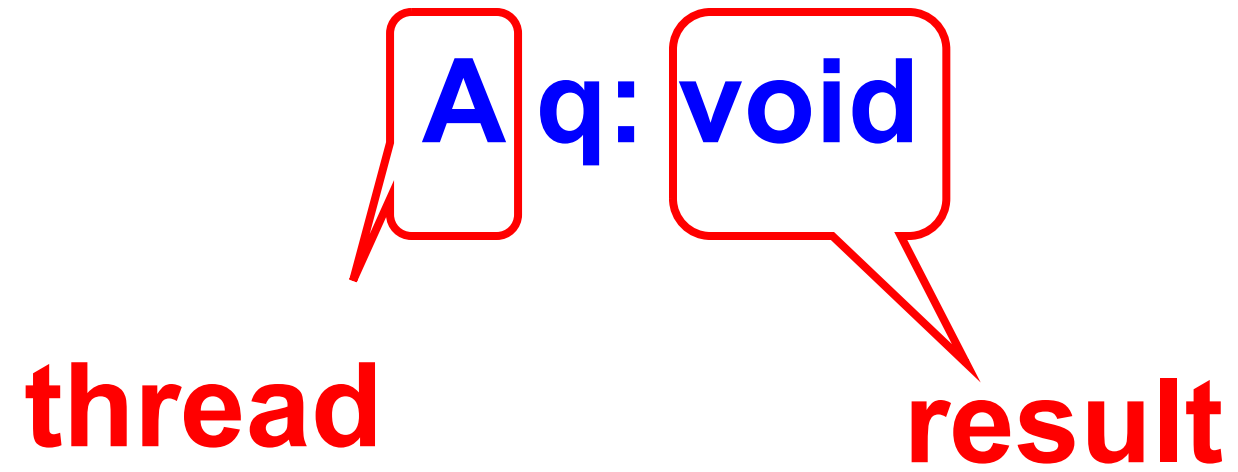
**A q: void**

# Response Notation

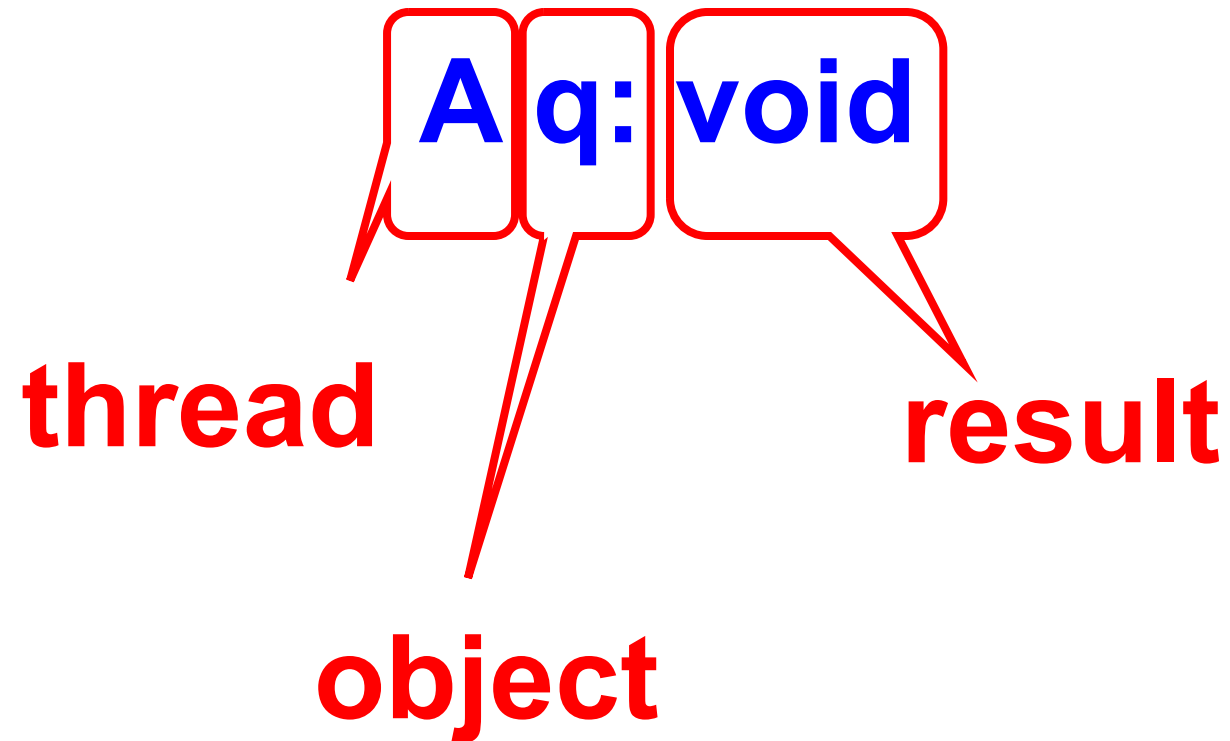
 **Aq: void**

**thread**

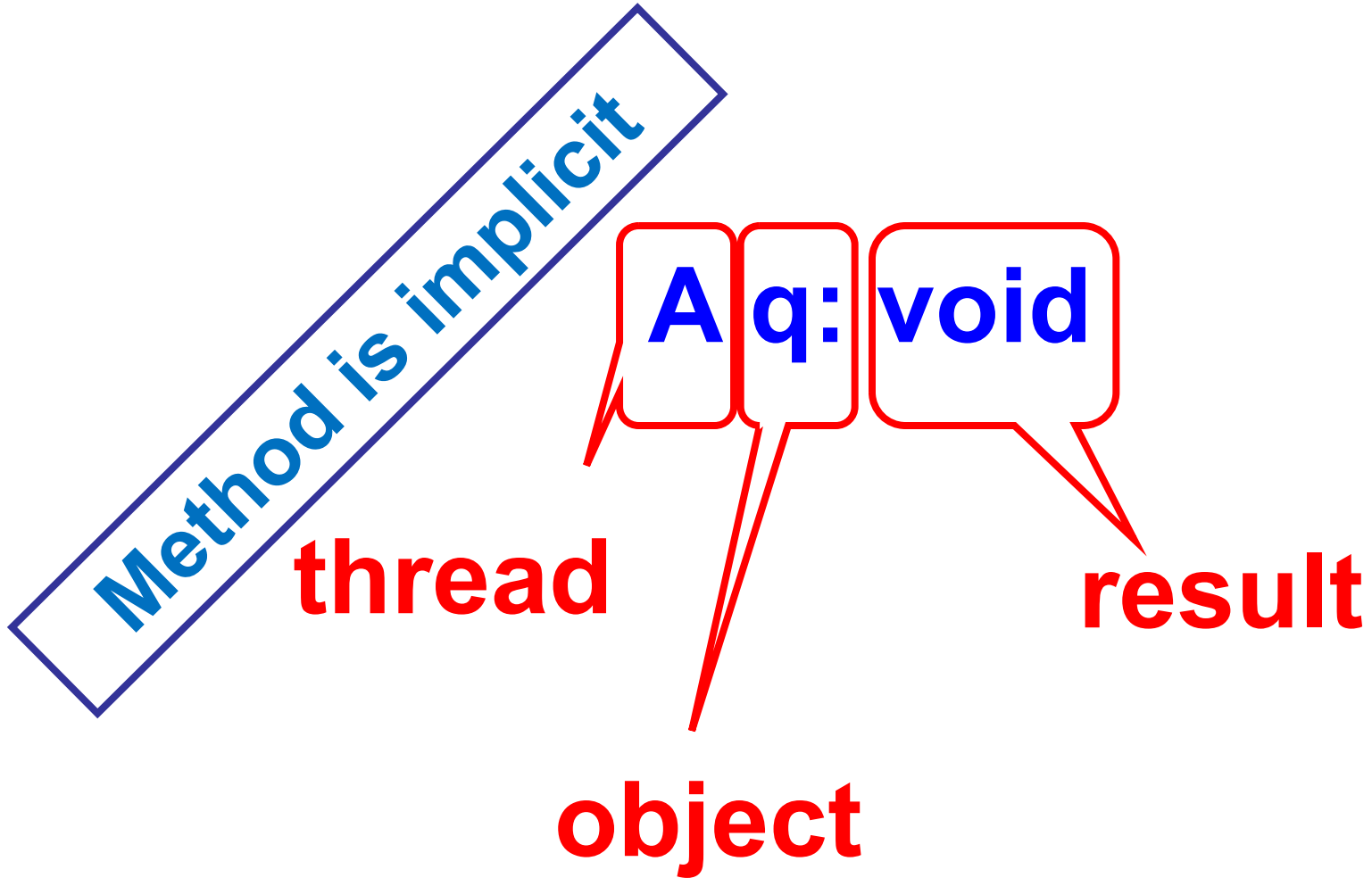
# Response Notation



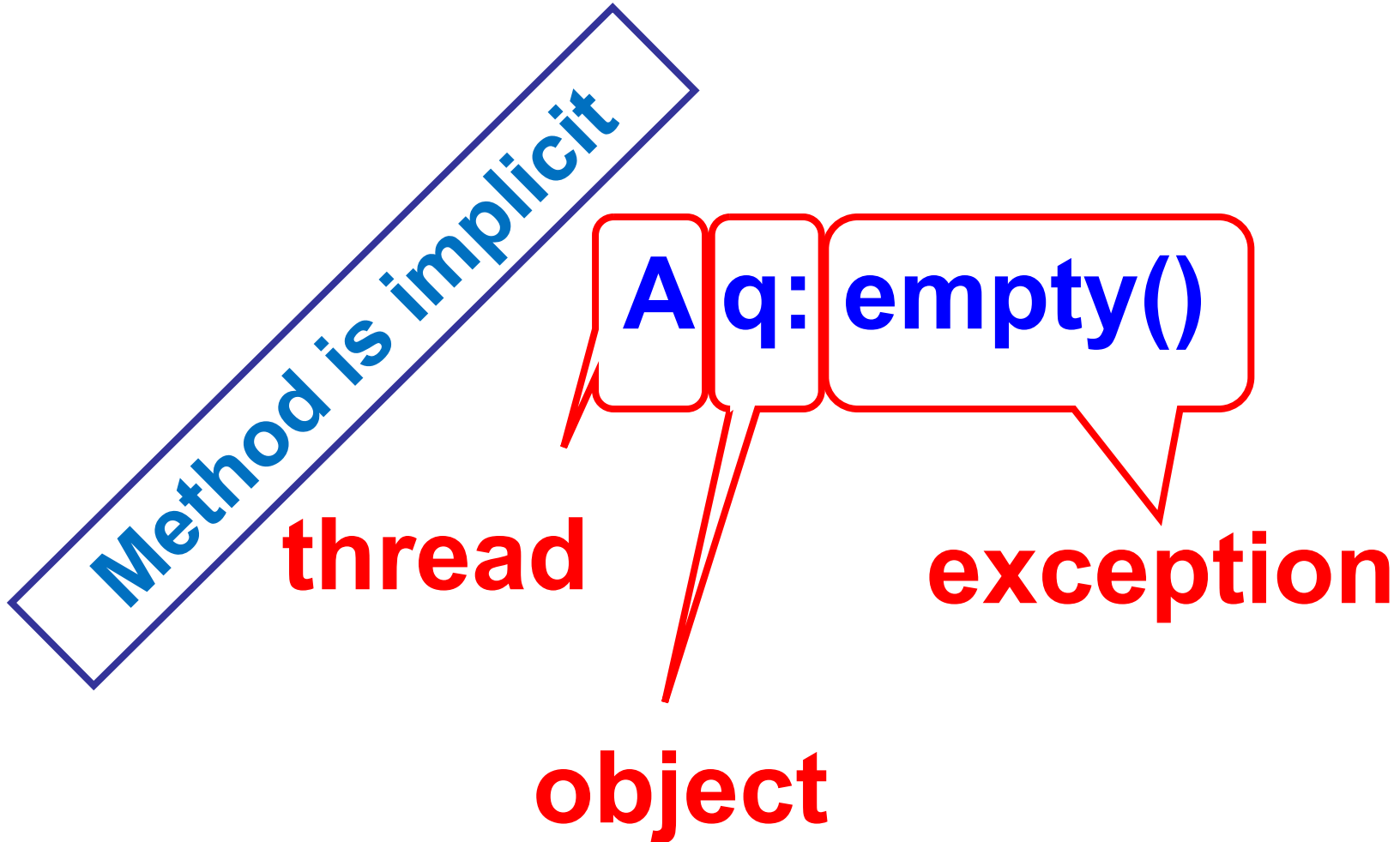
# Response Notation



# Response Notation



# Response Notation



# History - Describing an Execution

**H =**

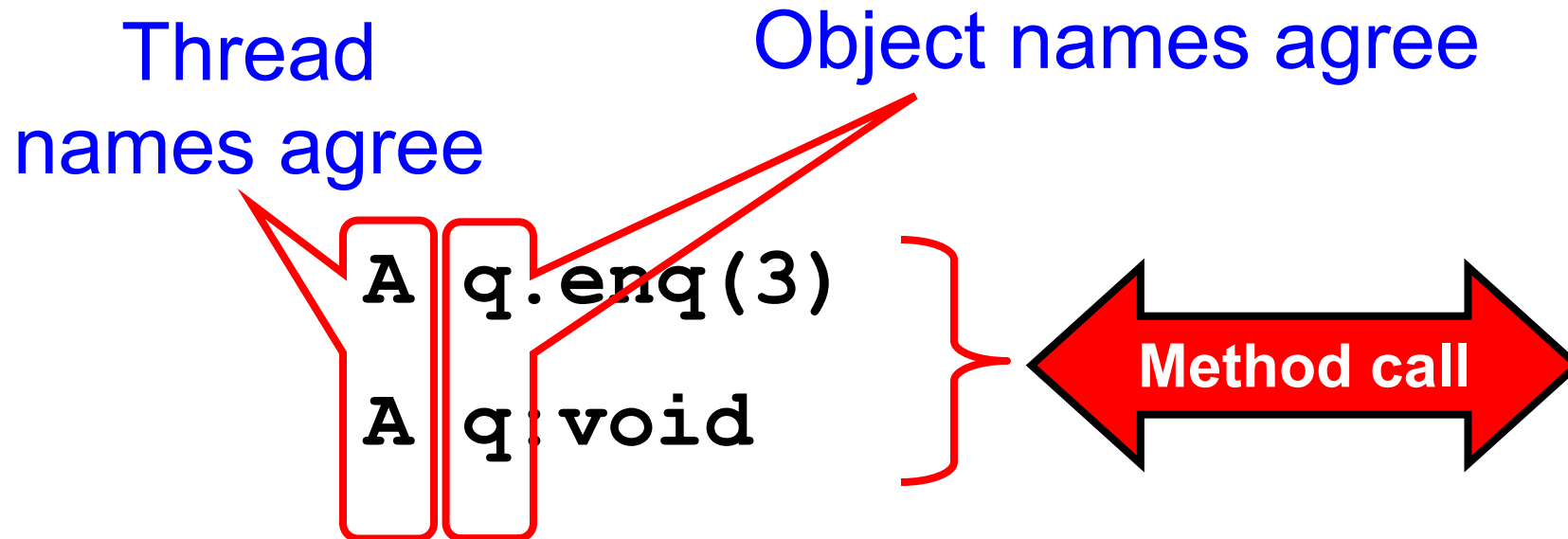
- A q.enq(3)
- A q:void
- A q.enq(5)
- B p.enq(4)
- B p:void
- B q.deq()
- B q:3

**Sequence of  
invocations and  
responses**



# Definition

- Invocation & response *match* if



# Object Projections

$H =$

- A `q.enqueue(3)`
- A `q: void`
- B `p.enqueue(4)`
- B `p: void`
- B `q.dequeue()`
- B `q: 3`

# Object Projections

A `q.enqueue(3)`

A `q:void`

$H|q =$

B `q.dequeue()`

B `q:3`

# Thread Projections

$H =$

- A `q.enqueue(3)`
- A `q: void`
- B `p.enqueue(4)`
- B `p: void`
- B `q.dequeue()`
- B `q: 3`

# Thread Projections

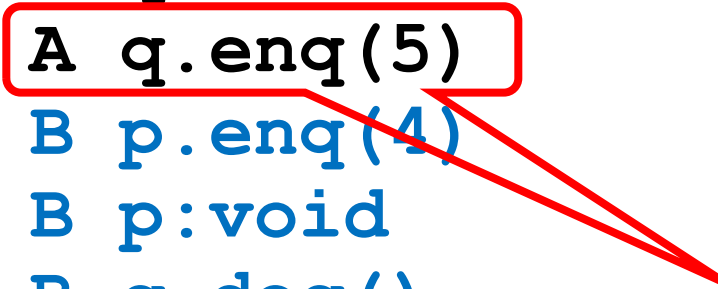
$H|B =$

- $B \text{ p.enq}(4)$
- $B \text{ p:void}$
- $B \text{ q.deq}()$
- $B \text{ q:3}$

# 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
```



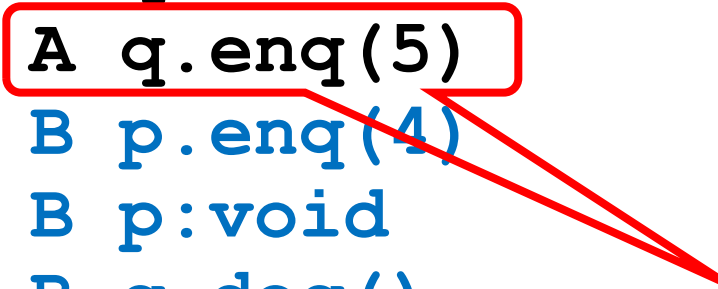
**An invocation is *pending* if it has no matching response**

# 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**

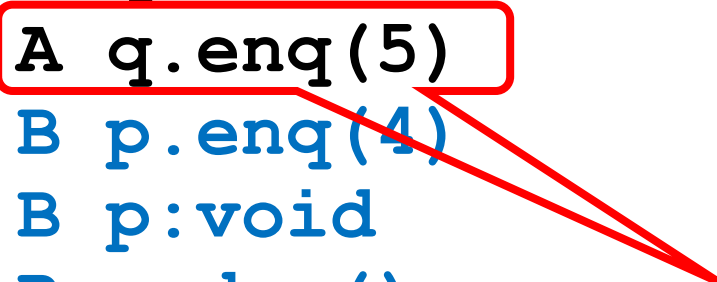


# Complete Subhistory

**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

**match**

B p.enq(4)

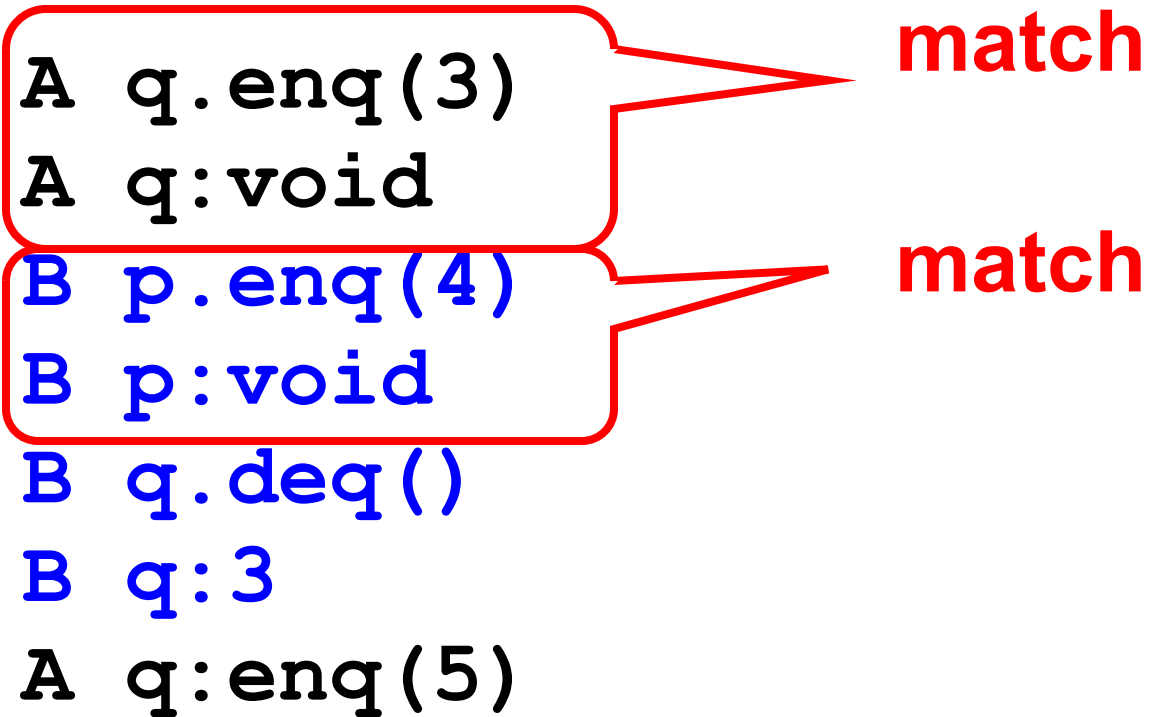
B p:void

B q.deq()

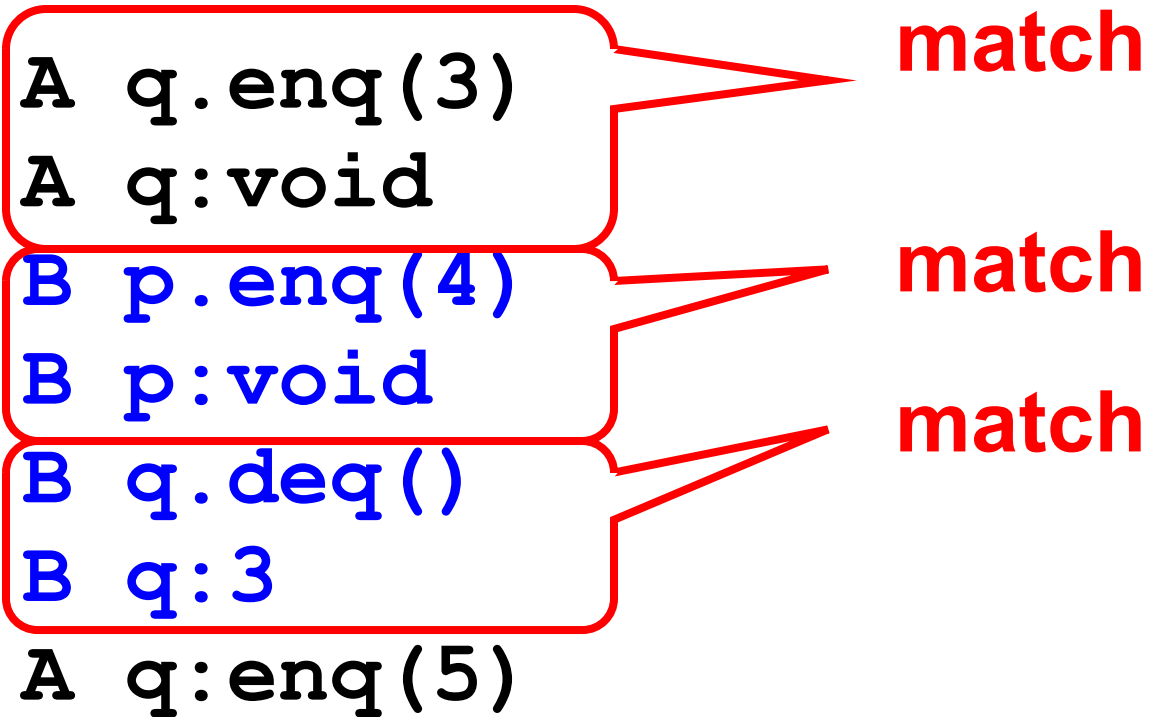
B q:3

A q:enq(5)

# Sequential Histories



# Sequential Histories



# Sequential Histories

A q.enq(3)  
A q:void

**match**

B p.enq(4)  
B p:void

**match**

B q.deq()  
B q:3

**match**

A q:enq(5)

**Final pending  
invocation OK**

# Sequential Histories

A q.enq(3)  
A q:void

**match**

B p.enq(4)  
B p:void

**match**

B q.deq()  
B q:3

**match**

A q:enq(5)

**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

$H \mid B =$

- 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

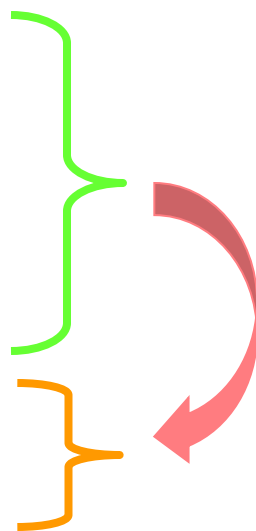
- A sequential specification is some way of telling whether a
  - Single-thread, single-object history
  - Is legal
- For example:
  - Pre and post-conditions
  - But plenty of other techniques exist ...

# Legal Histories

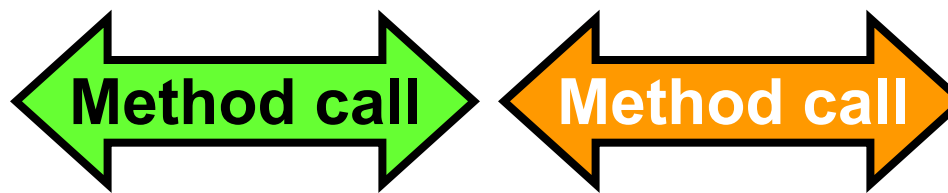
- A sequential (multi-object) history  $H$  is legal if
  - For every *object*  $x$
  - $H|x$  is in the sequential spec for  $x$
  - *Not talking about threads now!*

# Precedence

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

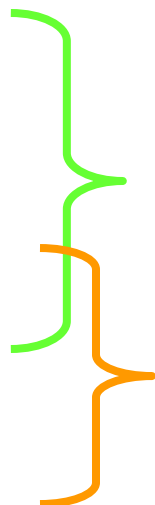


A method call **precedes**  
another if *response* event  
precedes *invocation* event

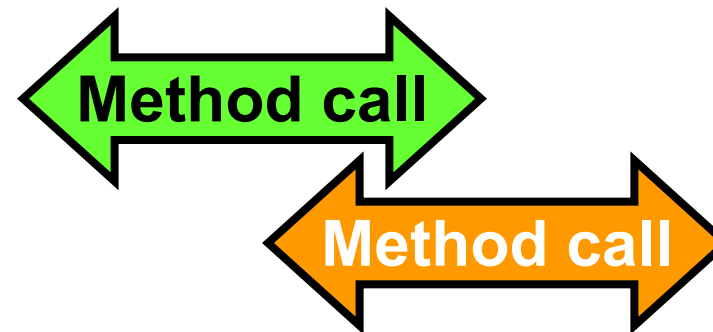


# Non-Precedence

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

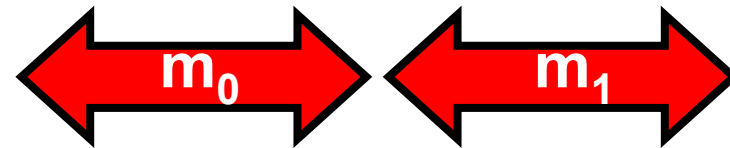
A diagram showing two overlapping method calls. A green bracket groups the first three lines: 'A q.enq(3)', 'B p.enq(4)', and 'B p.void'. An orange bracket groups the last three lines: 'B q.deq()', 'A q:void', and 'B q:3'. The two brackets overlap, indicating that the two method calls are concurrent.

**Some method calls  
overlap one another**



# Notation

- Given
  - History  $H$
  - method executions  $m_0$  and  $m_1$  in  $H$
- We say  $m_0 \rightarrow_H m_1$ , if
  - $m_0$  precedes  $m_1$
- Relation  $m_0 \rightarrow_H m_1$  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  $\rightarrow_G \subset \rightarrow_S$

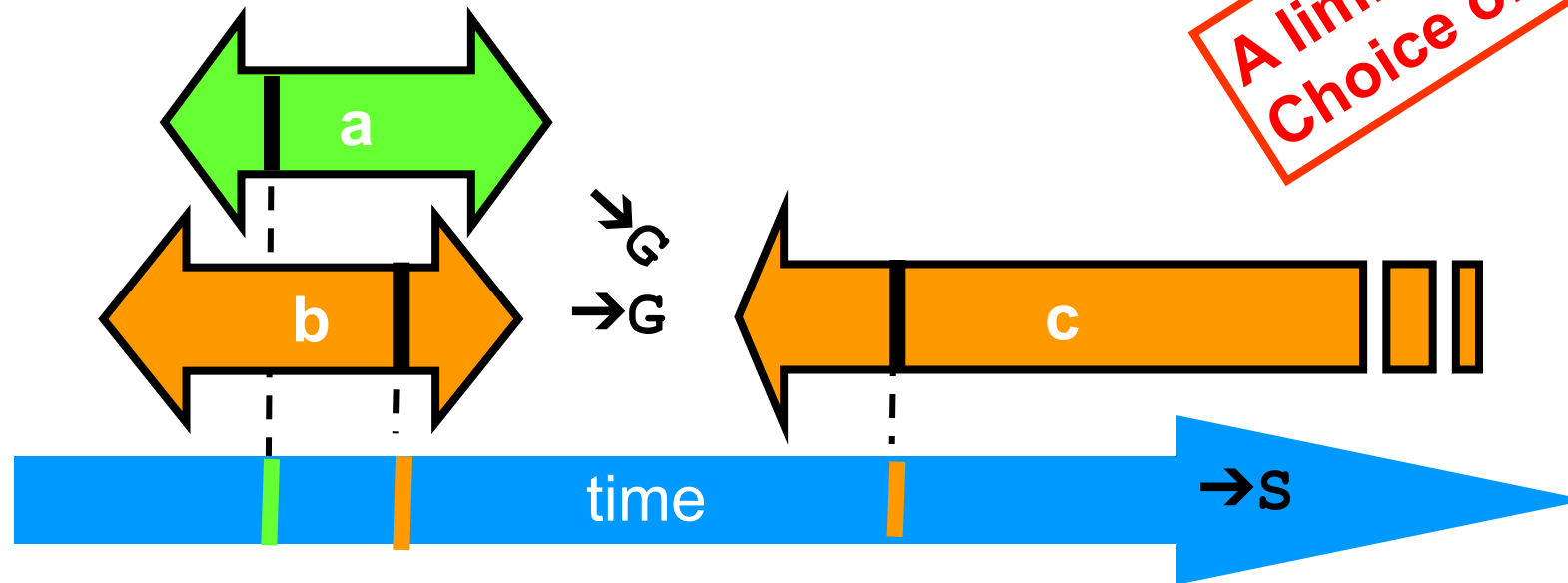
# Remarks

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

Ensuring  $\rightarrow_G \subset \rightarrow_S$

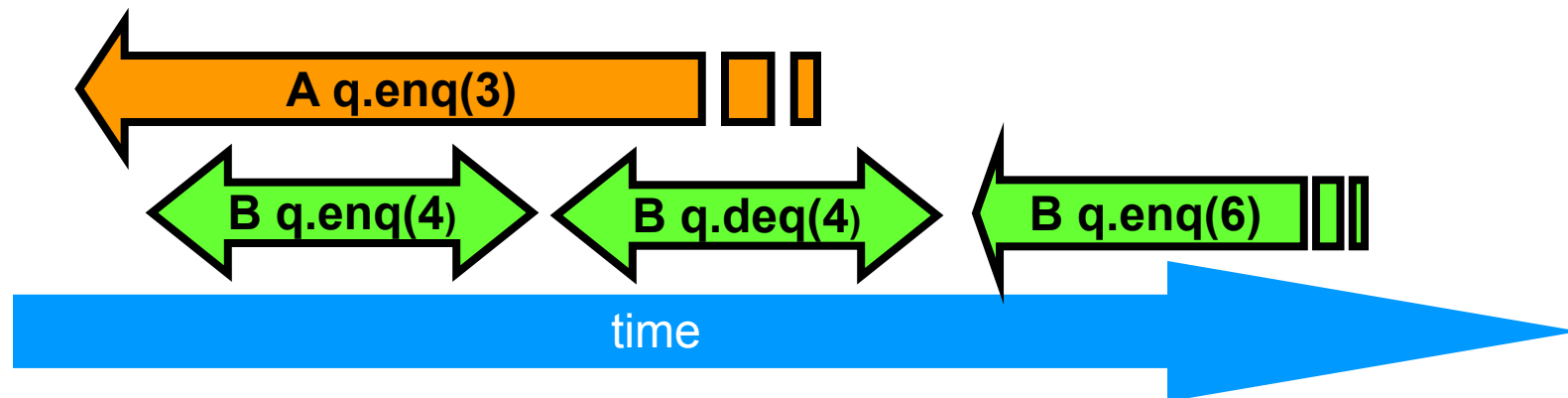
$$\rightarrow_G = \{a \rightarrow c, b \rightarrow c\}$$

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



# 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)

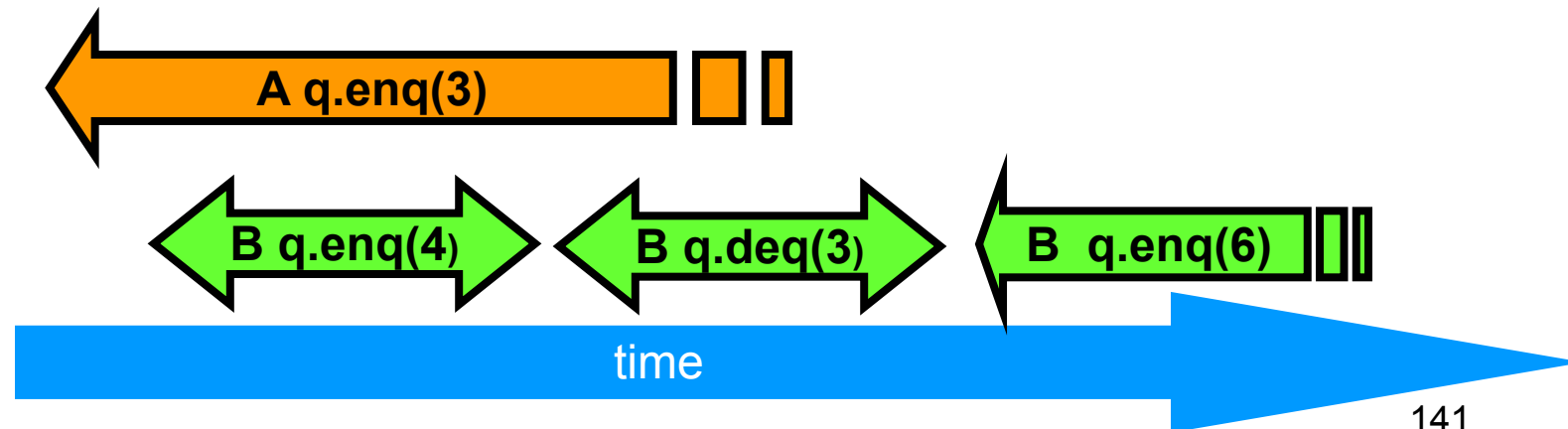
B q:void

B q.deq()

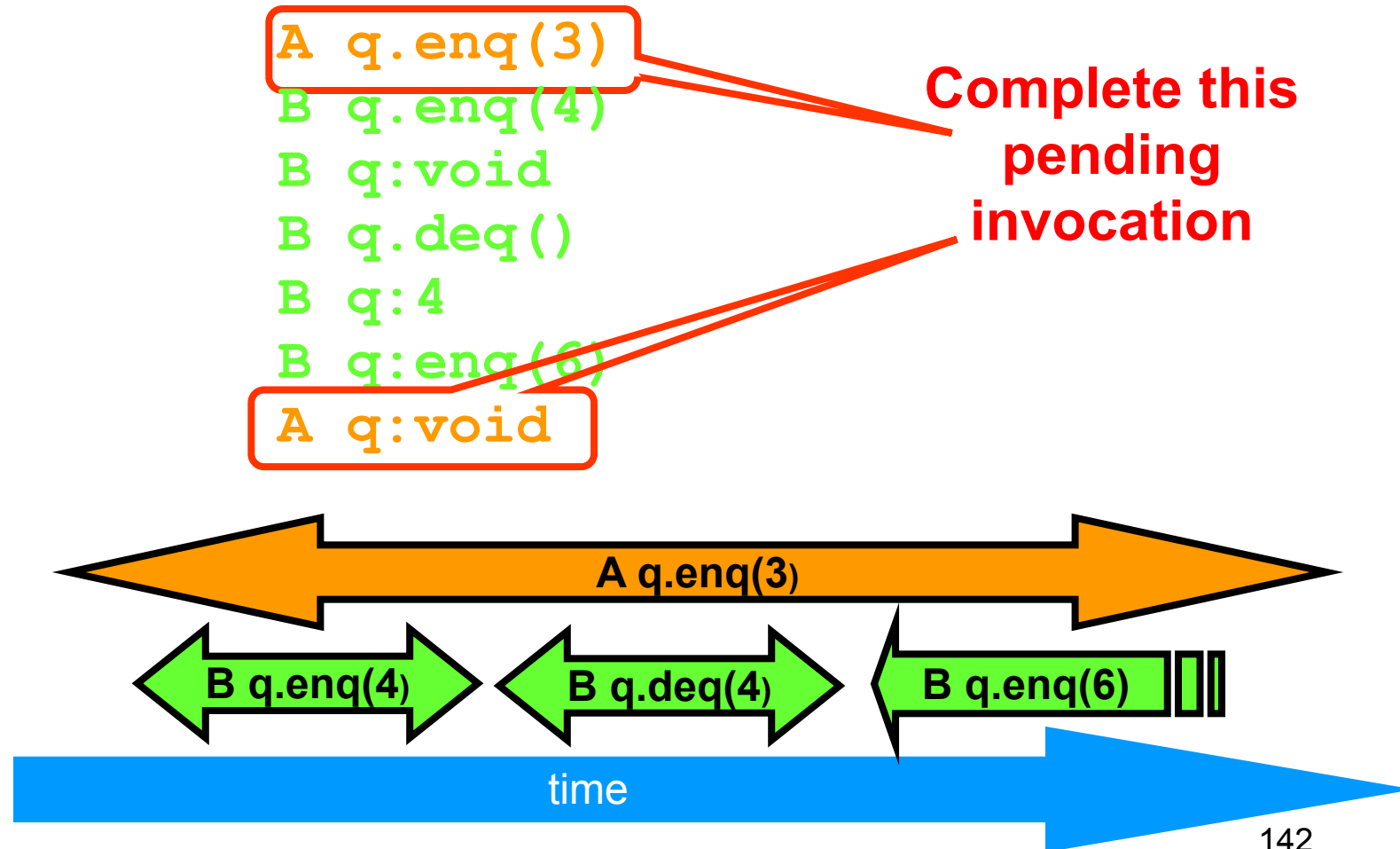
B q:4

B q:enq(6)

Complete this  
pending  
invocation



# Example



# Example

discard this one

A q.enq(3)

B q.enq(4)

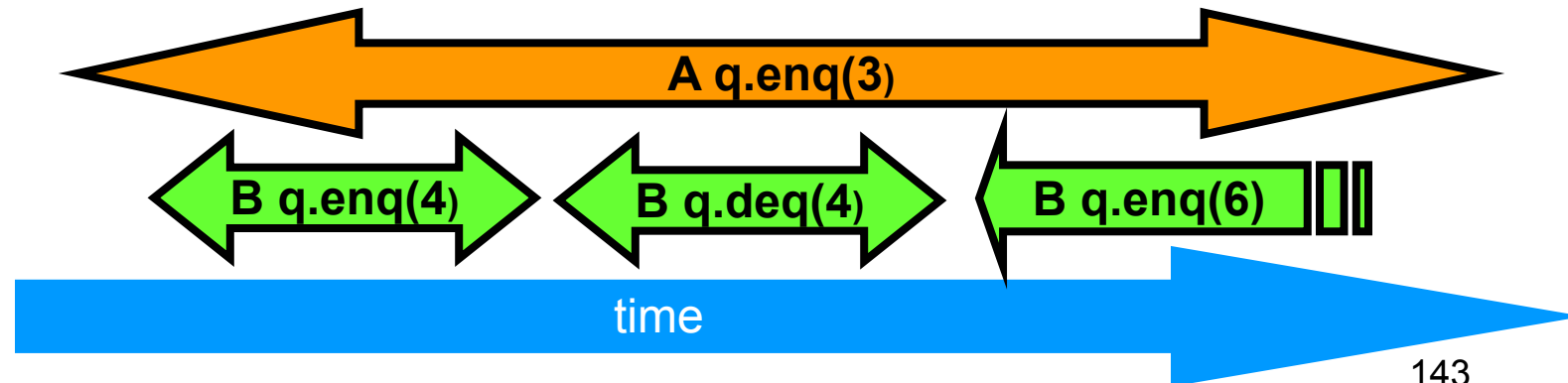
B q:void

B q.deq()

B q:4

B q:enq(6)

A q:void



# Example

A q.enq(3)

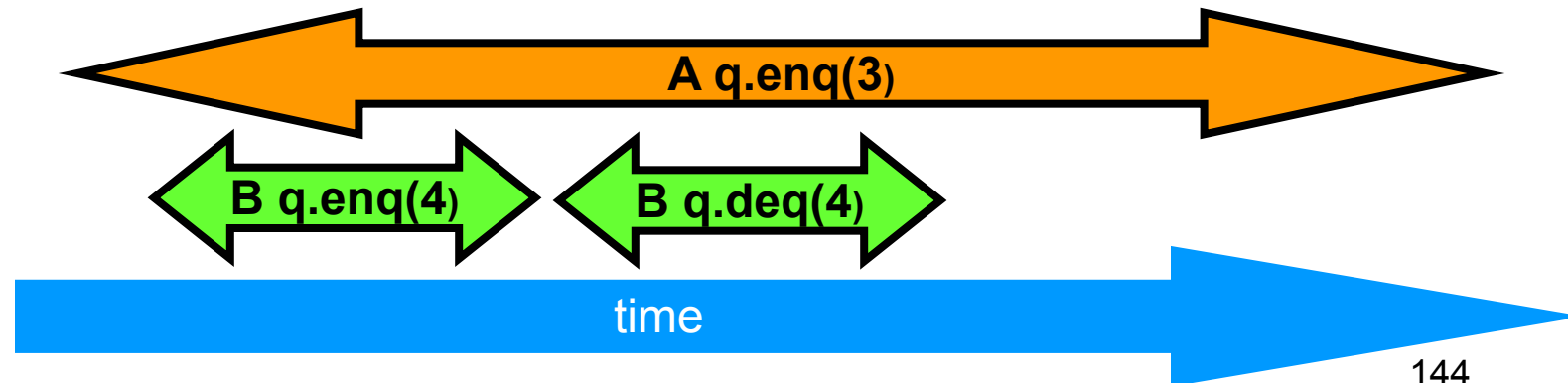
B q.enq(4)

B q:void

B q.deq()

B q:4

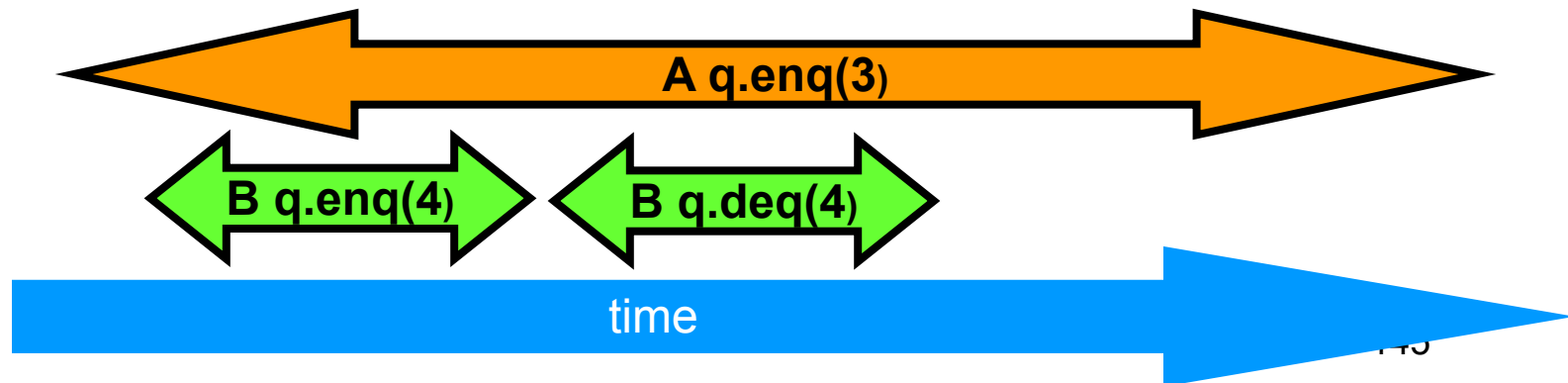
A q:void





# Example

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



# 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)

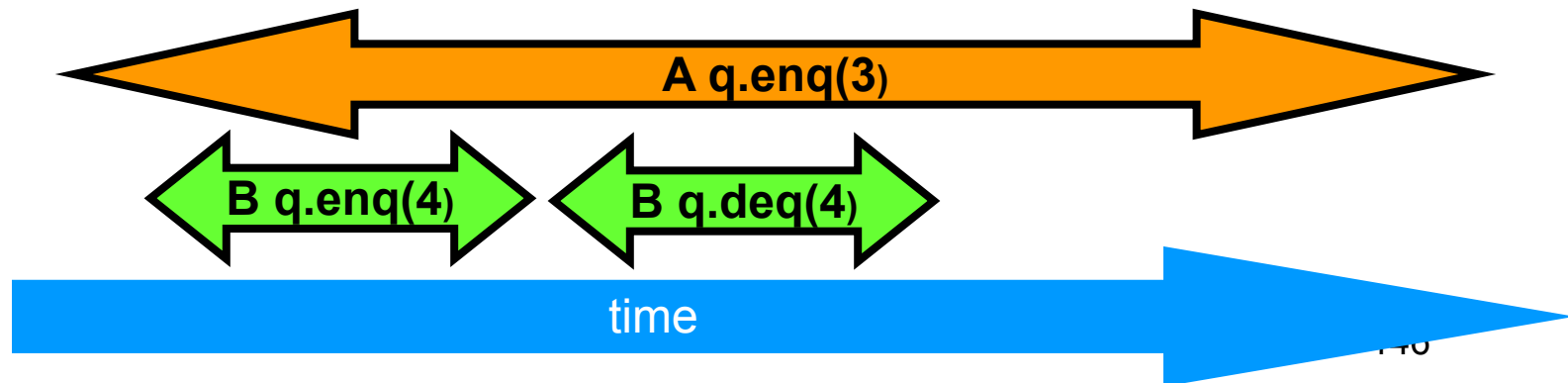
B q:void

A q.enq(3)

A q:void

B q.deq()

B q:4

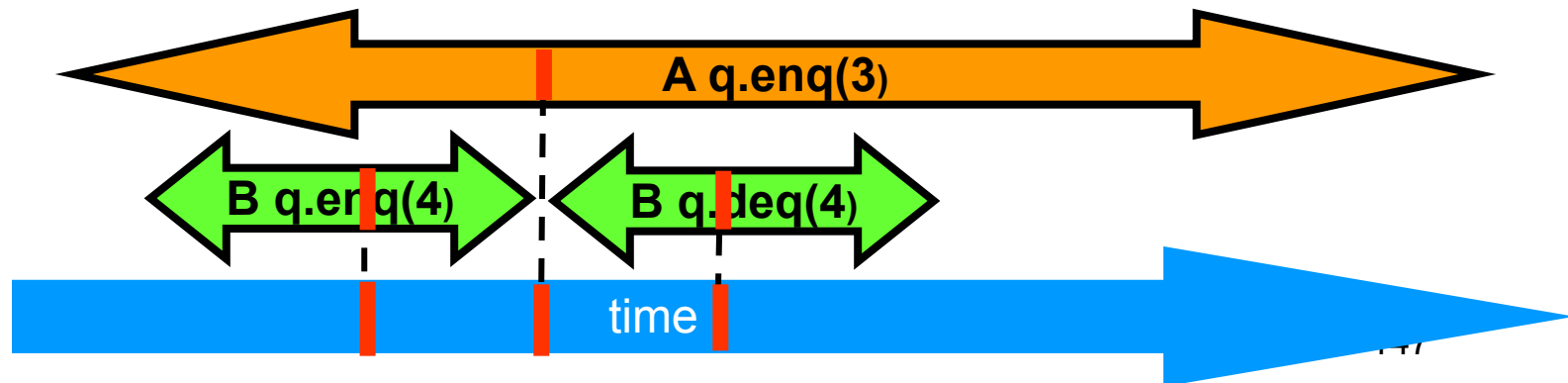


# Example

## Equivalent sequential history

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

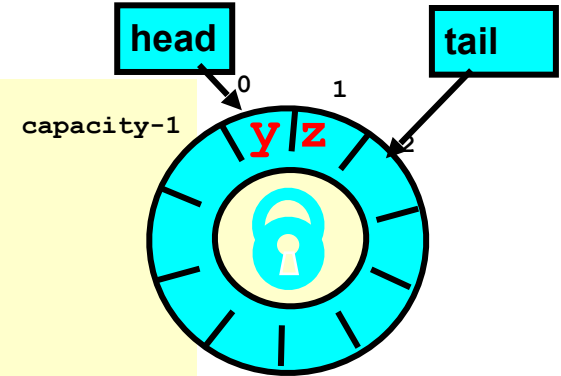


# Why Does Composability Matter?

- Modularity
- Can prove linearizability of objects in isolation
- Can compose independently-implemented objects
  - A history of two linearizable objects is linearizable

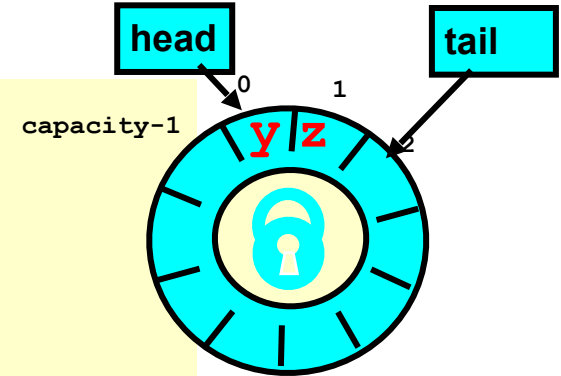
# Reasoning About Linearizability: Locking

```
def deq() : T = {  
  myLock.lock()  
  try {  
    if (tail == head) {  
      throw EmptyException  
    }  
    val x = items(head % items.length)  
    head = head + 1  
    x  
  } finally {  
    myLock.unlock()  
  }  
}
```



# Reasoning About Linearizability: Locking

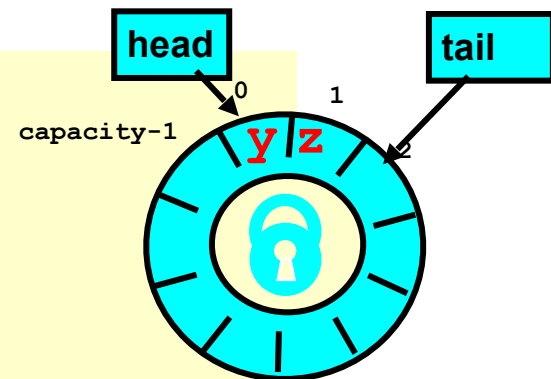
```
def deq() : T = {  
  myLock.lock()  
  try {  
    if (tail == head) {  
      throw EmptyException  
    }  
    val x = items(head % items.length)  
    head = head + 1  
    x  
  } finally {  
    myLock.unlock()  
  }  
}
```



Linearization points  
are when locks are  
released

# More Reasoning: Wait-free

```
class LockFreeQueue[T: ClassTag] (val capacity: Int) {  
  
  @volatile  
  private var head, tail: Int = 0  
  private val items = new Array[T](capacity)  
  
  def enq(x: T): Unit = {  
    if (tail - head == items.length) throw FullException  
    items(tail % items.length) = x  
    tail = tail + 1  
  }  
  
  def deq(): T = {  
    if (tail == head) throw EmptyException  
    val x = items(head % items.length)  
    head = head + 1  
    x  
  }  
}
```



# More Reasoning: Wait-free

**Remember that there  
is only one enqueuer  
and only one dequeuer**

```
class LockFreeQueue[T: ClassTag](val capacity: Int) {  
    head, tail: Int = 0  
    items = new Array[T](capacity)  
  
    def enq(x: T): Unit = {  
        if (tail - head == capacity) throw EmptyException  
        items[tail % items.length] = x  
        tail = tail + 1  
    }  
  
    def deq(): T = {  
        if (tail == head) throw EmptyException  
        val x = items(head % items.length)  
        head = head + 1  
        x  
    }  
}
```

**Linearization order is order head  
and tail fields modified**




# Strategy

- Identify one atomic step where method “happens”
  - Critical section
  - Machine instruction
- Doesn't always work
  - Might need to define several different steps for a given method

# Linearizability: Summary

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

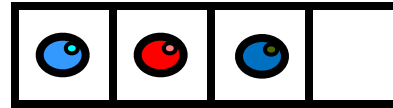
# Alternative: Sequential Consistency

- History  $H$  is **Sequentially Consistent** 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 a
    - Legal sequential history  $S$
    - ~~– Where  $\rightarrow G \subseteq \rightarrow S$~~
- Differs from linearizability
- 

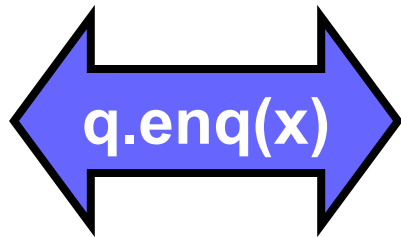
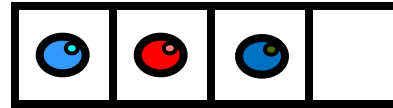
# Sequential Consistency

- No need to preserve real-time order
  - Cannot re-order operations done by the same thread
  - Can re-order non-overlapping operations done by different threads
- Often used to describe multiprocessor memory architectures

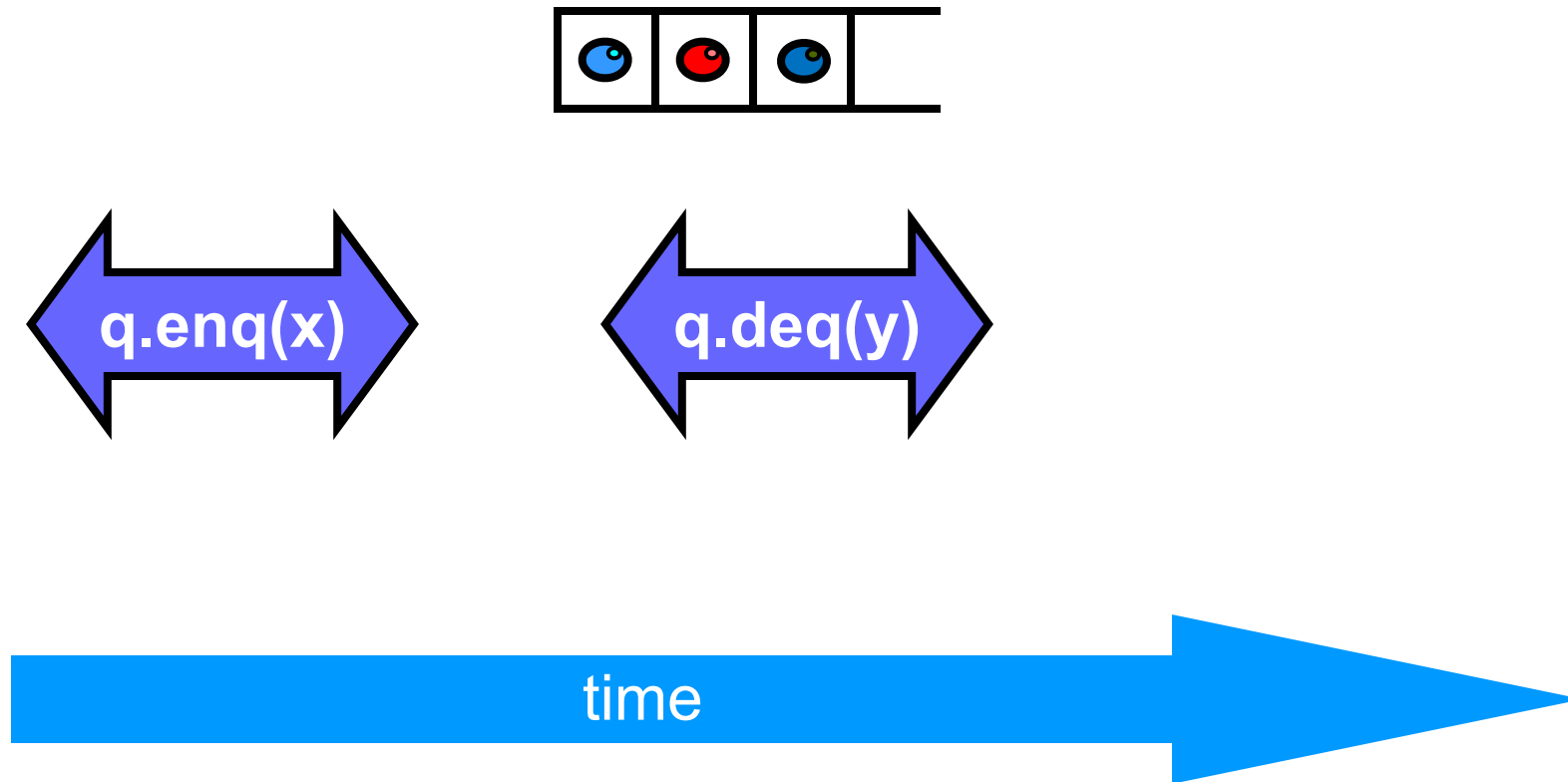
# Example



# Example

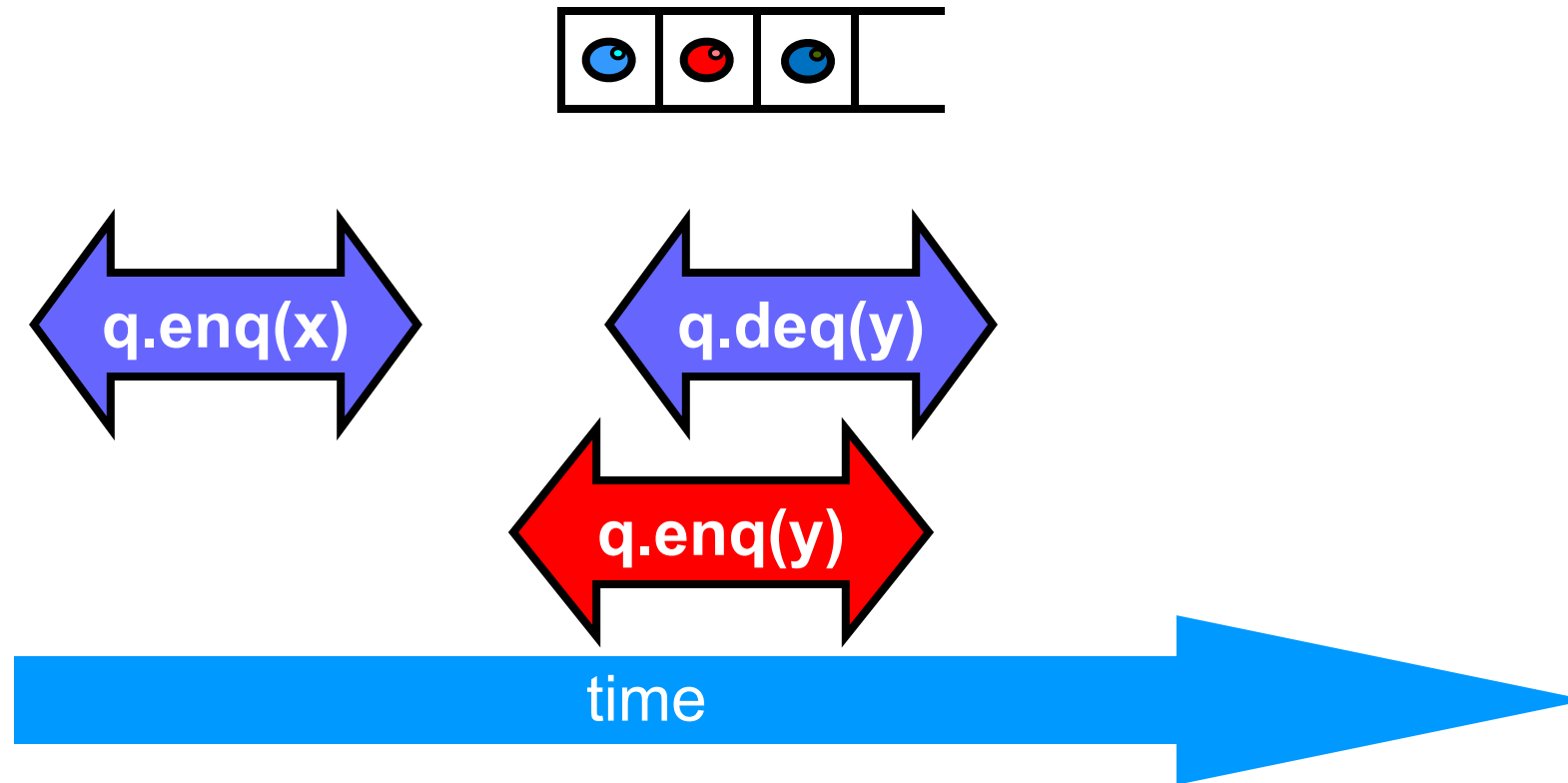


# Example





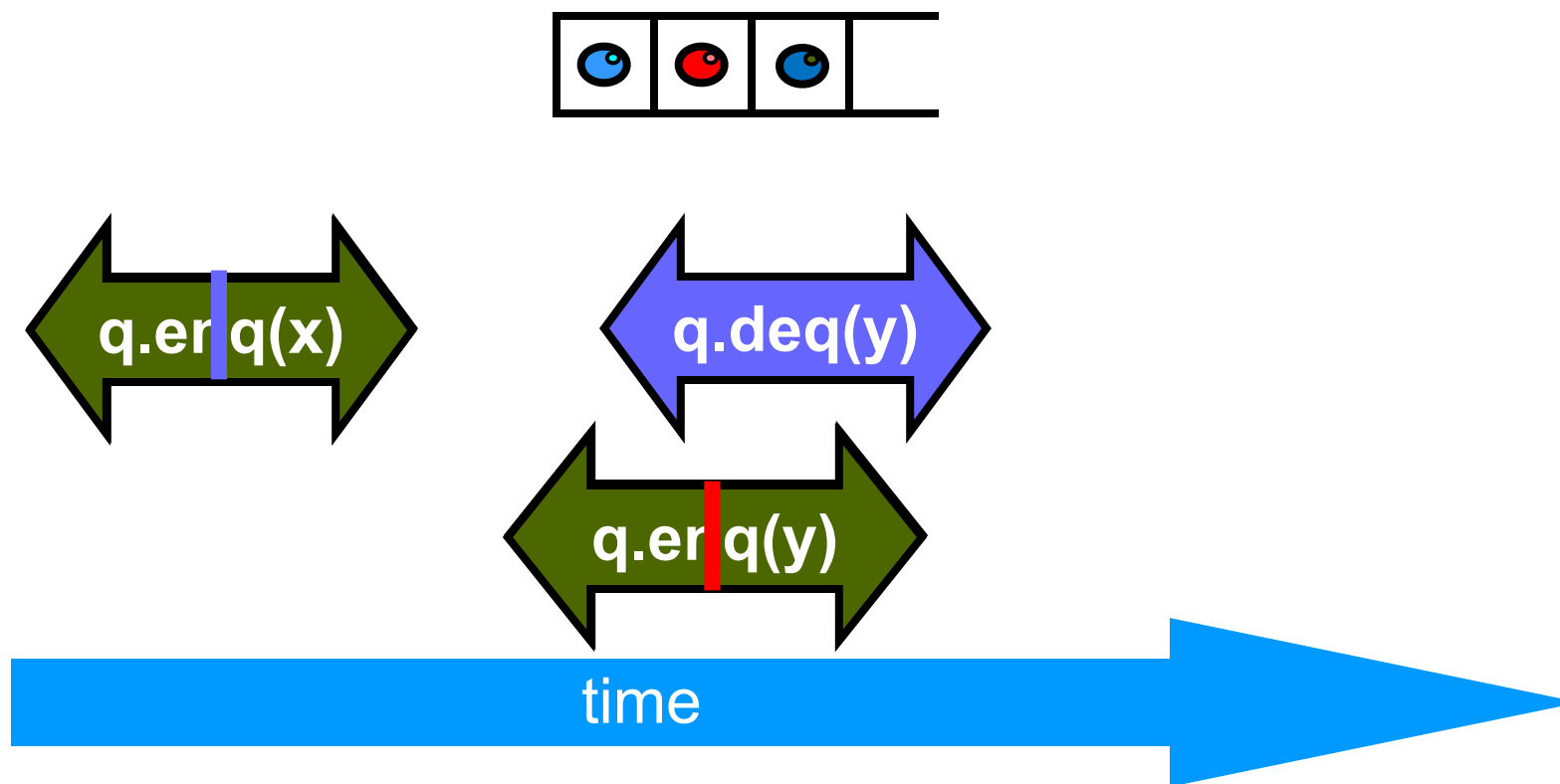
# Example





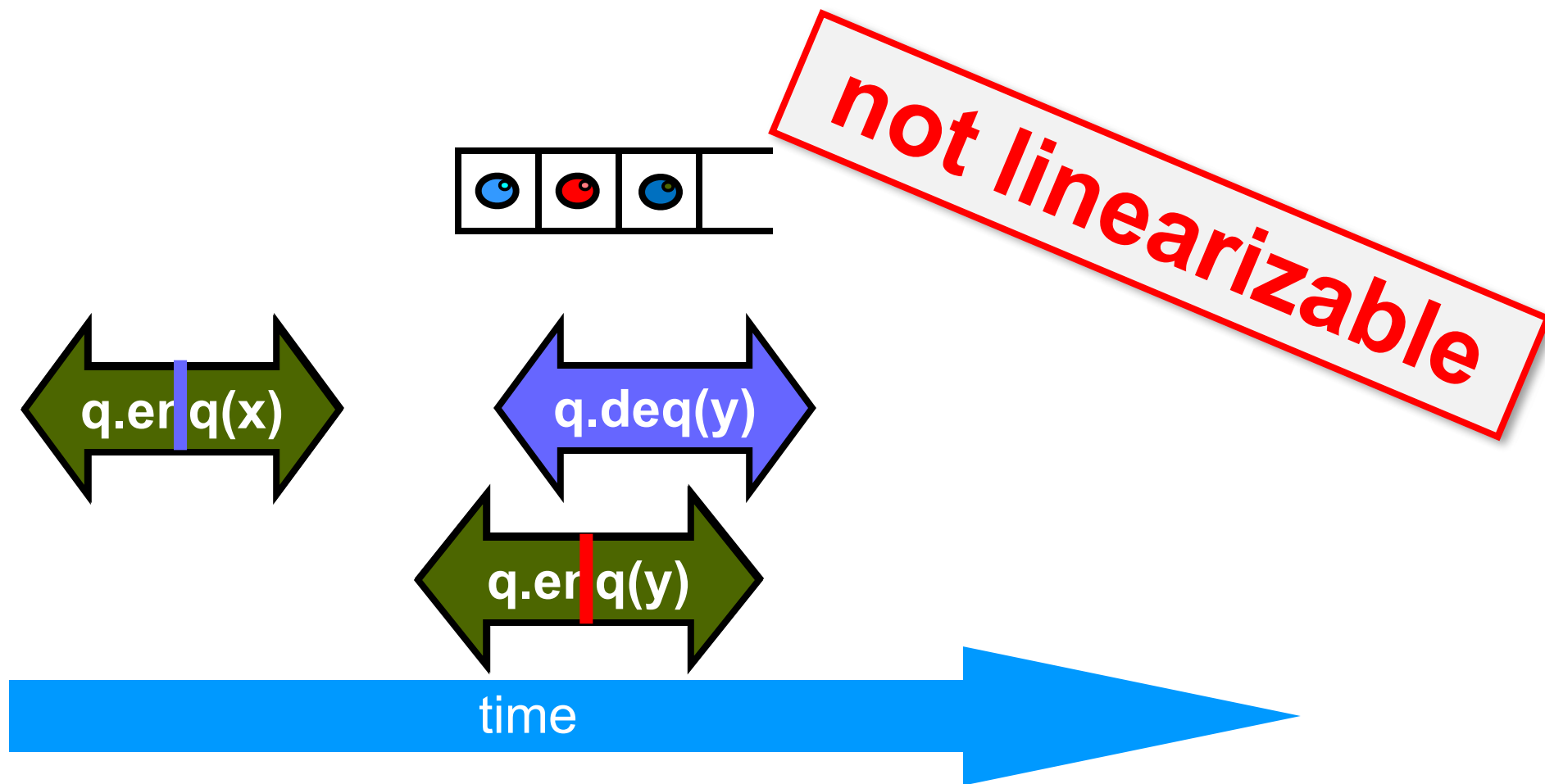


# Example





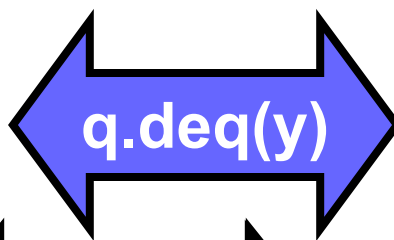
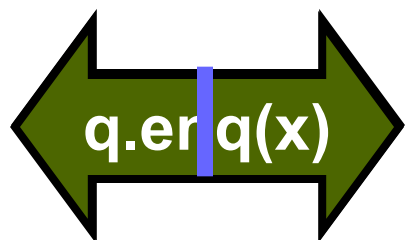
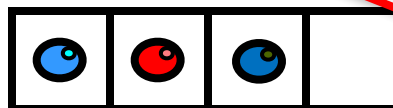
# Example





# Example

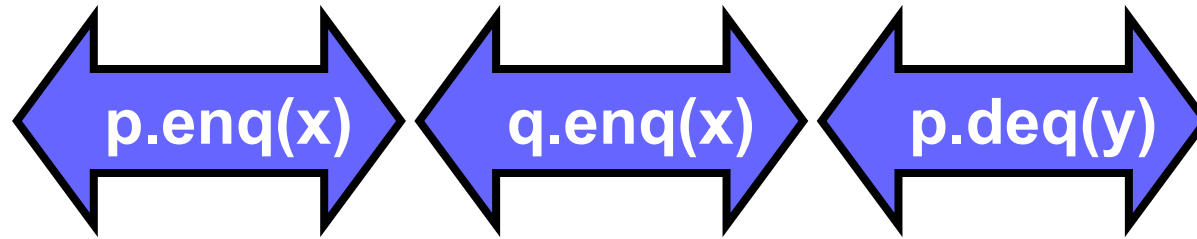
**Yet Sequentially  
Consistent**



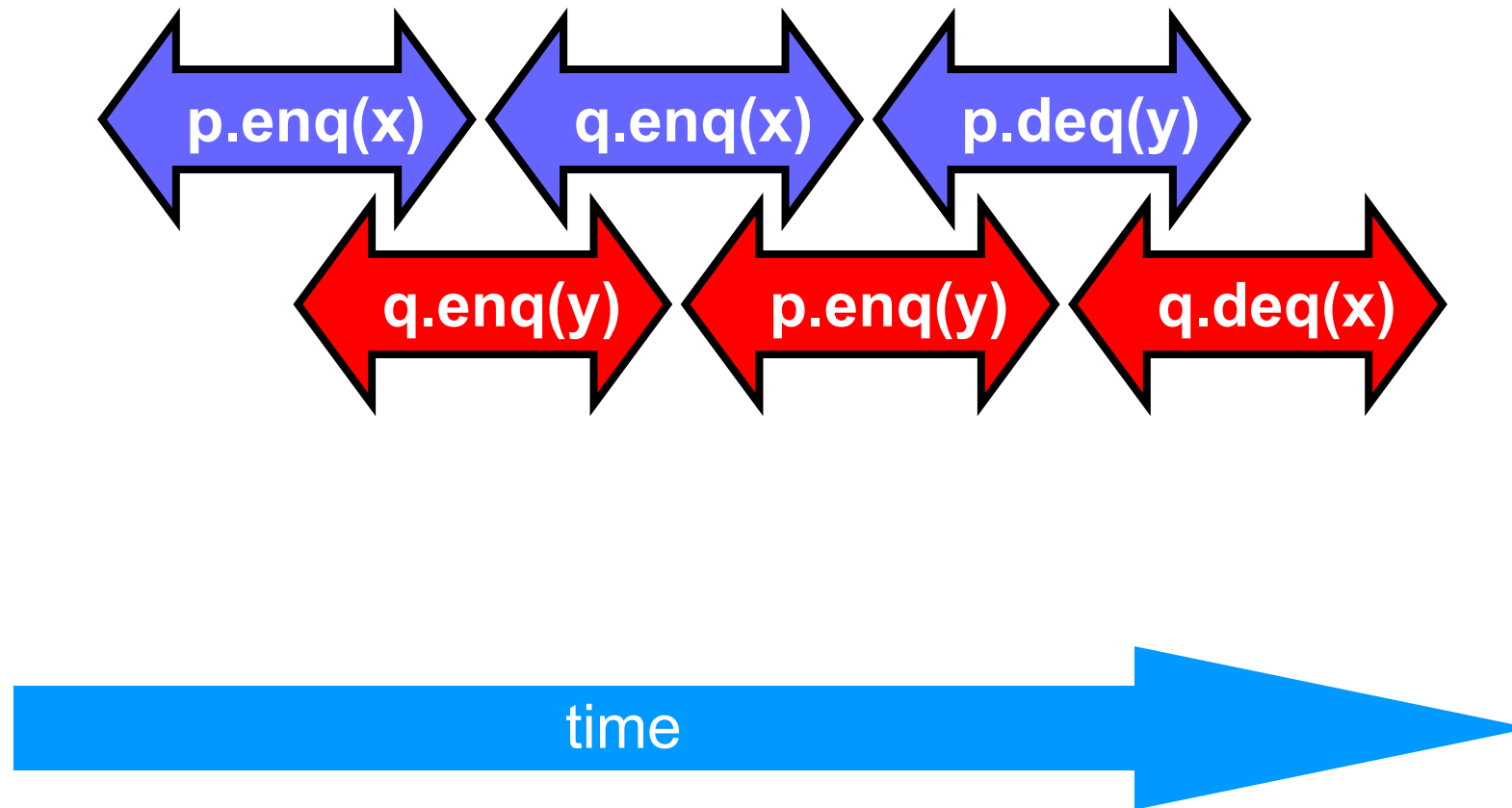
# Theorem

Sequential Consistency is not composable

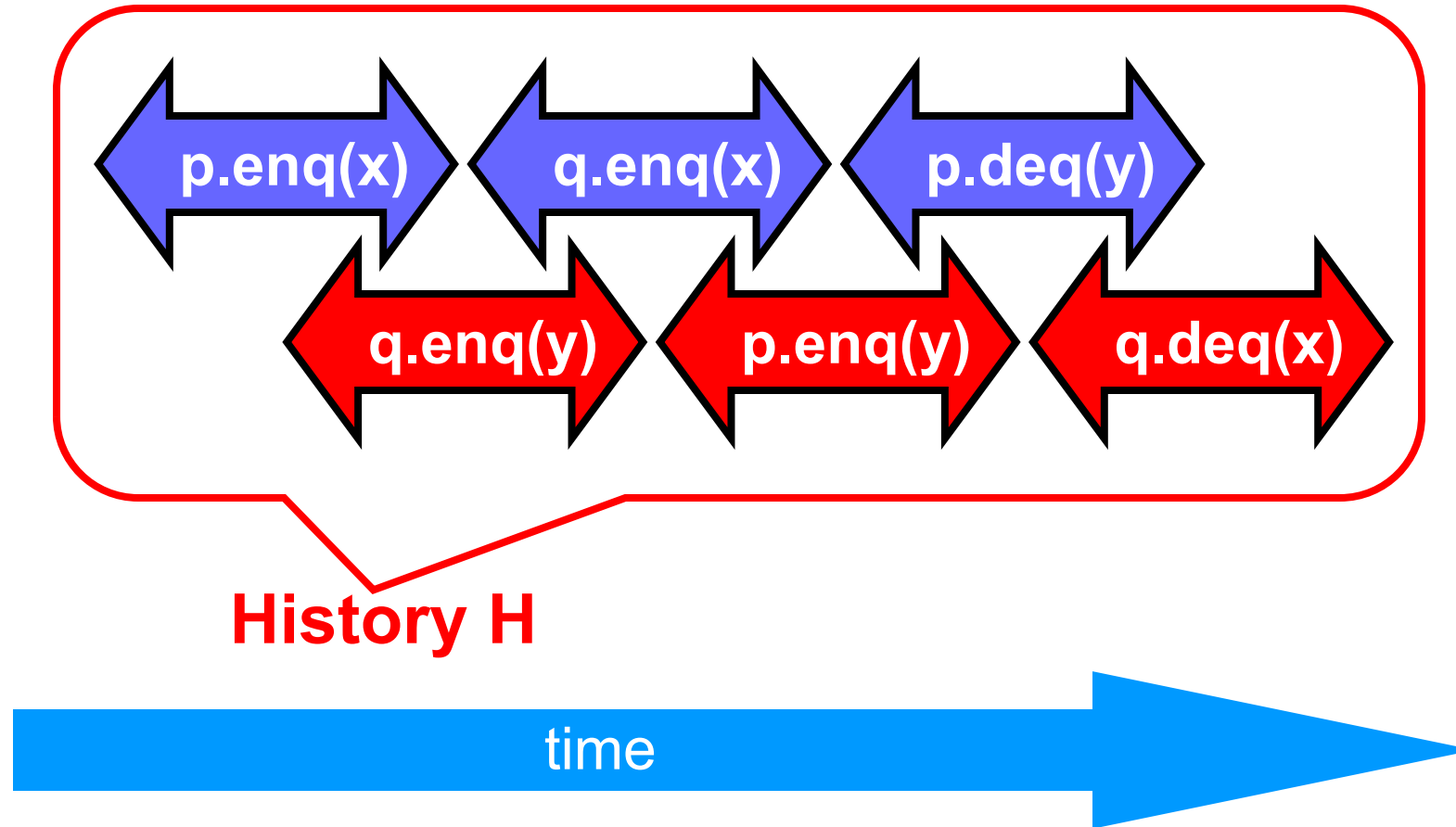
# FIFO Queue Example



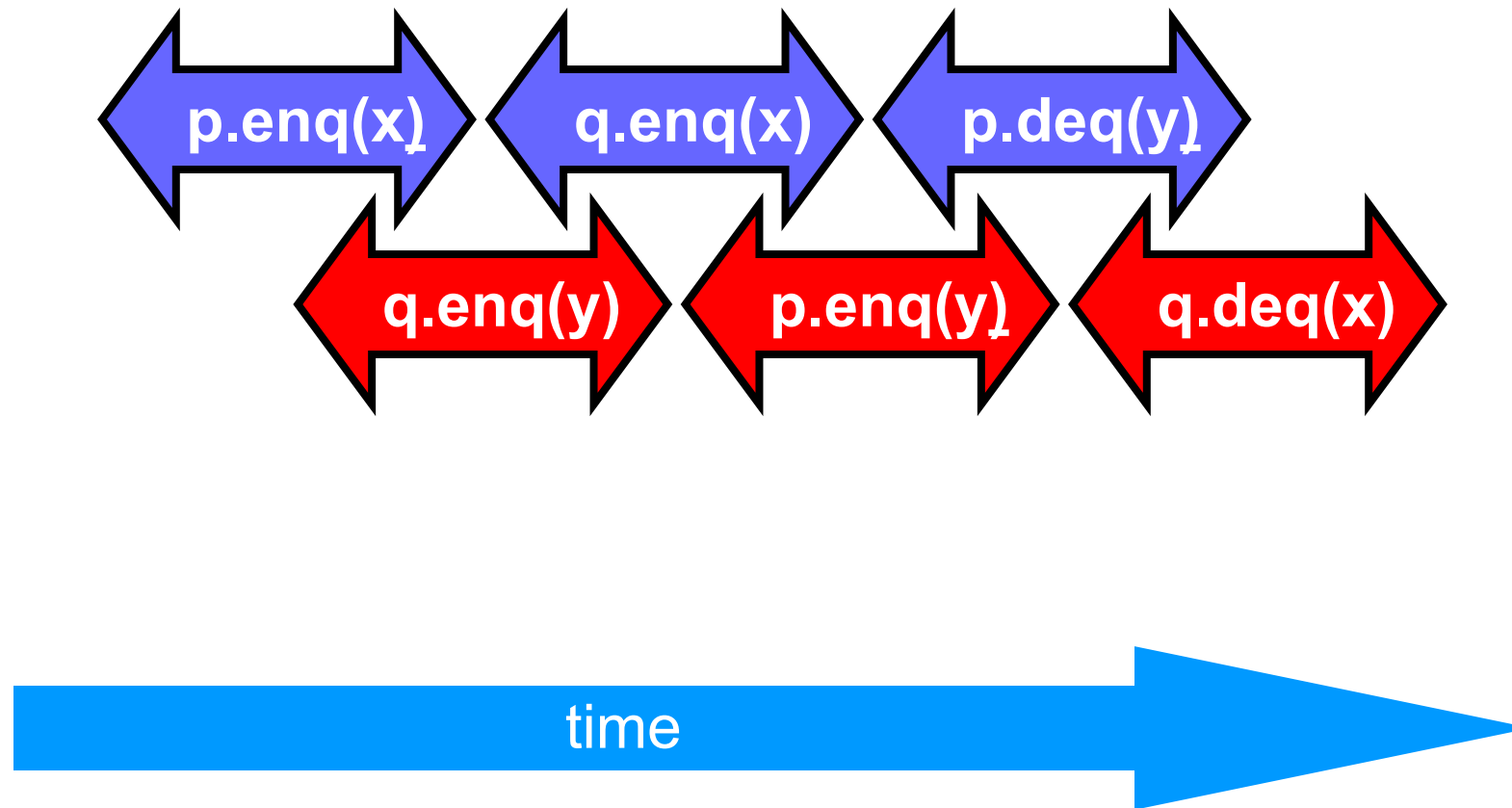
# FIFO Queue Example



# FIFO Queue Example

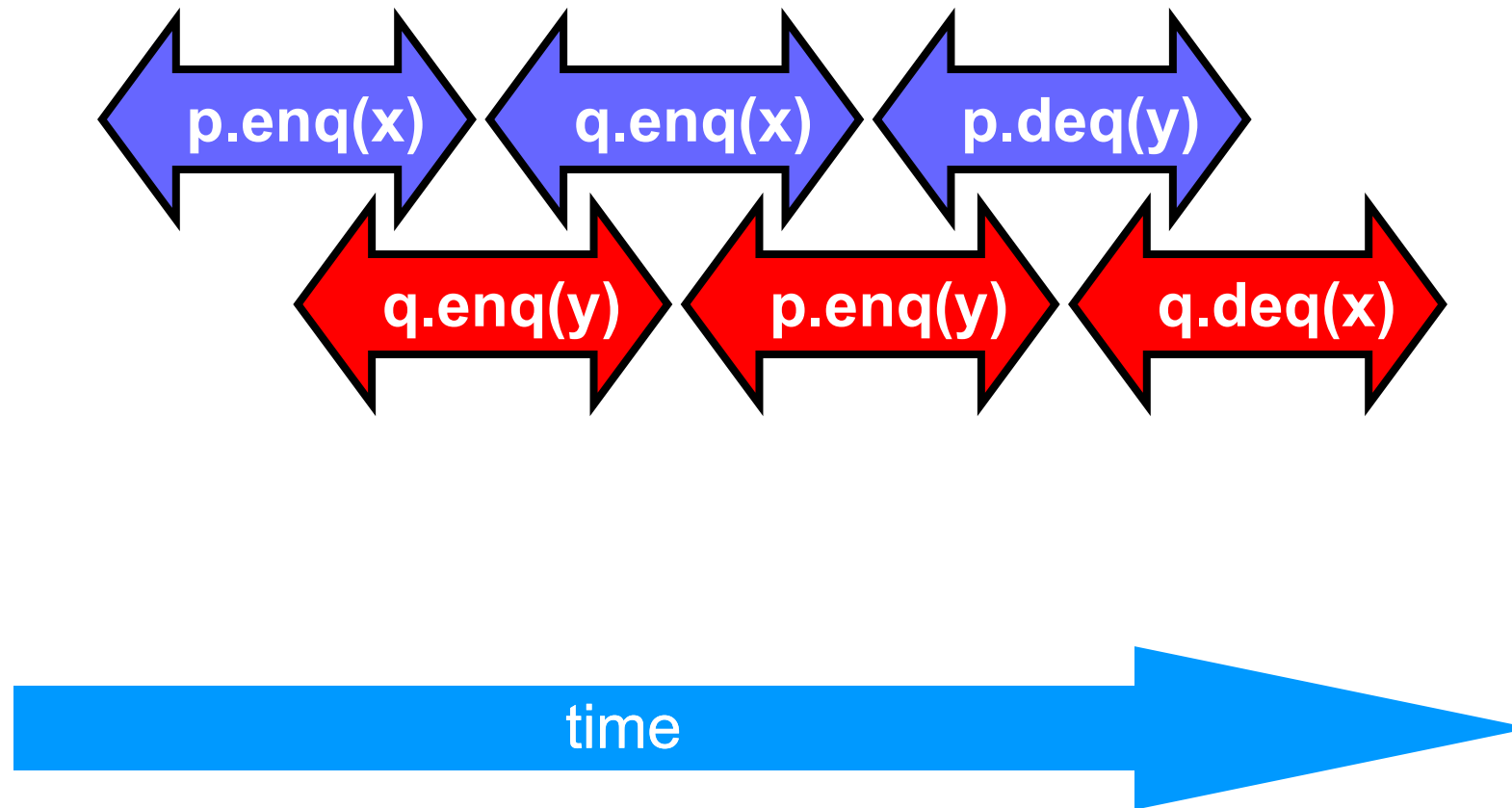


# H/p Sequentially Consistent

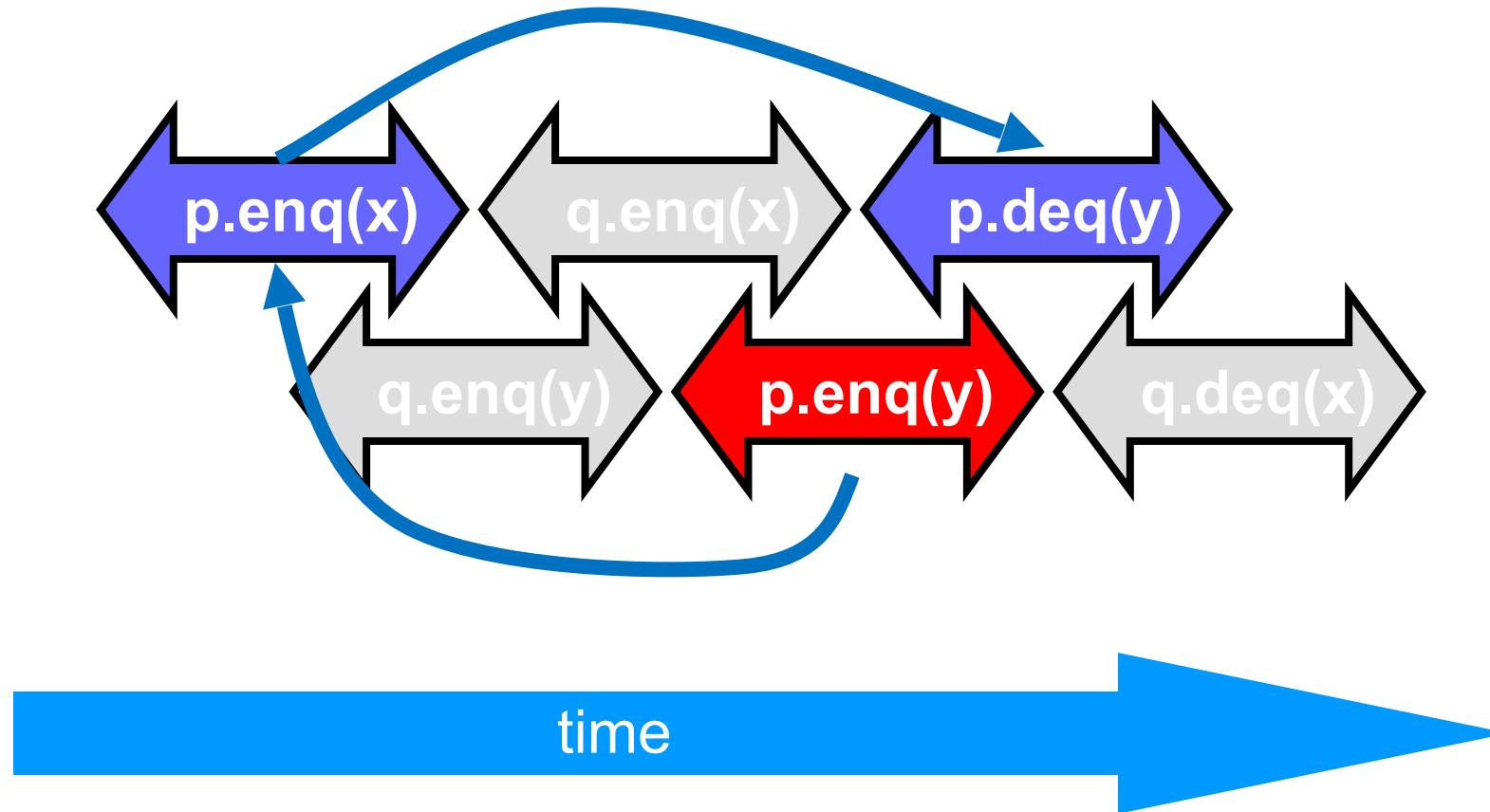




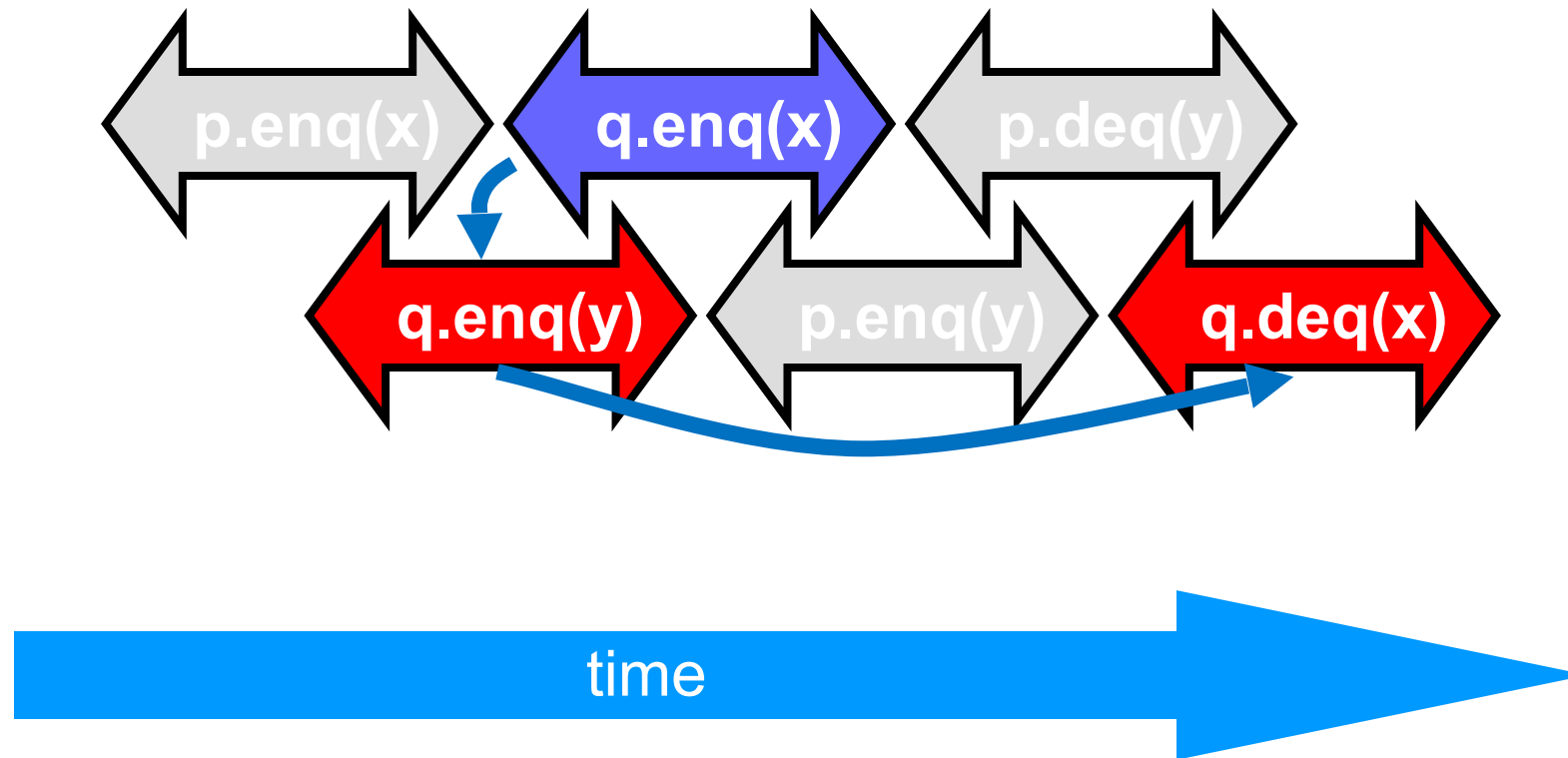
# H/q Sequentially Consistent



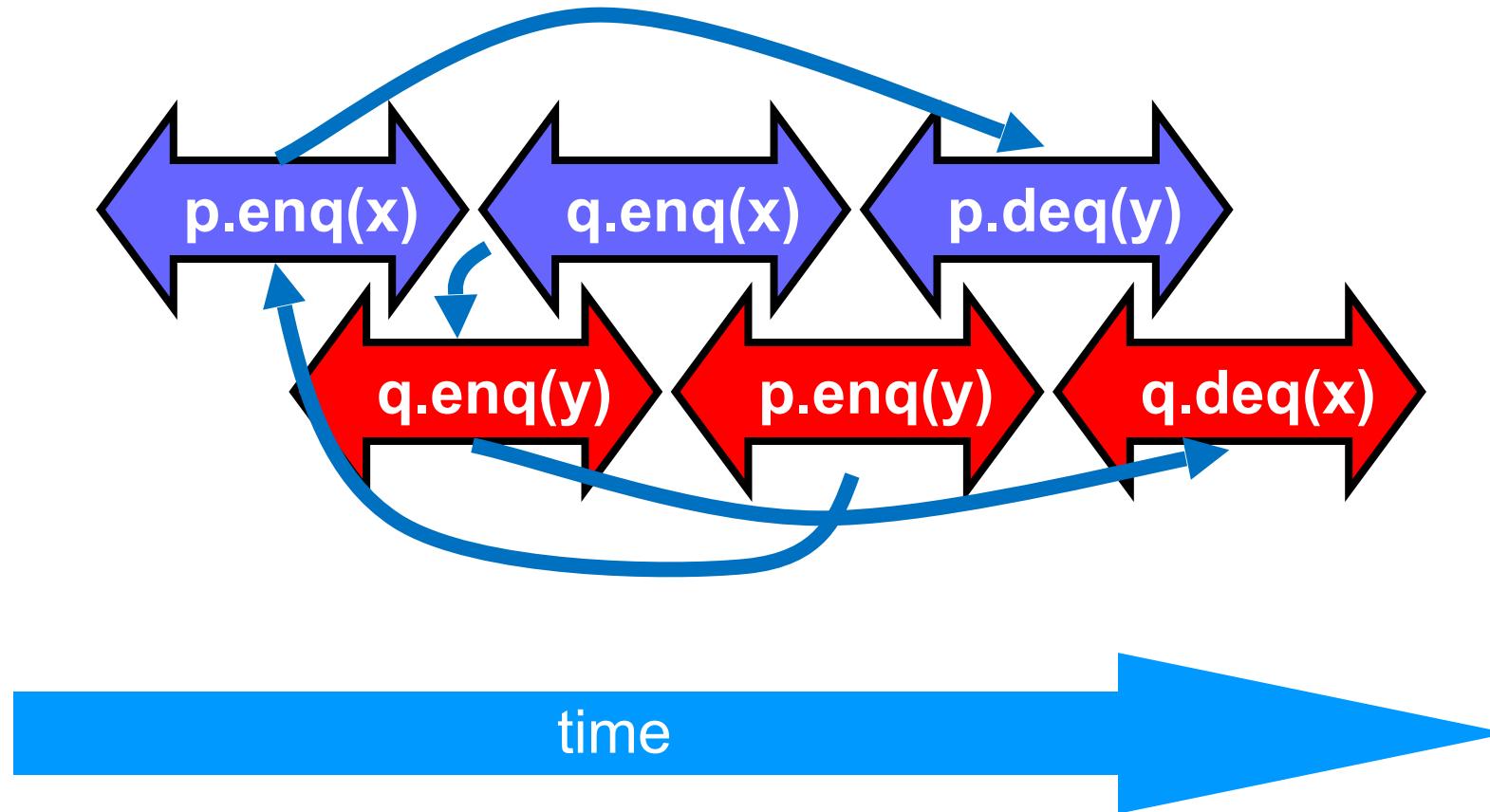
# Ordering imposed by p



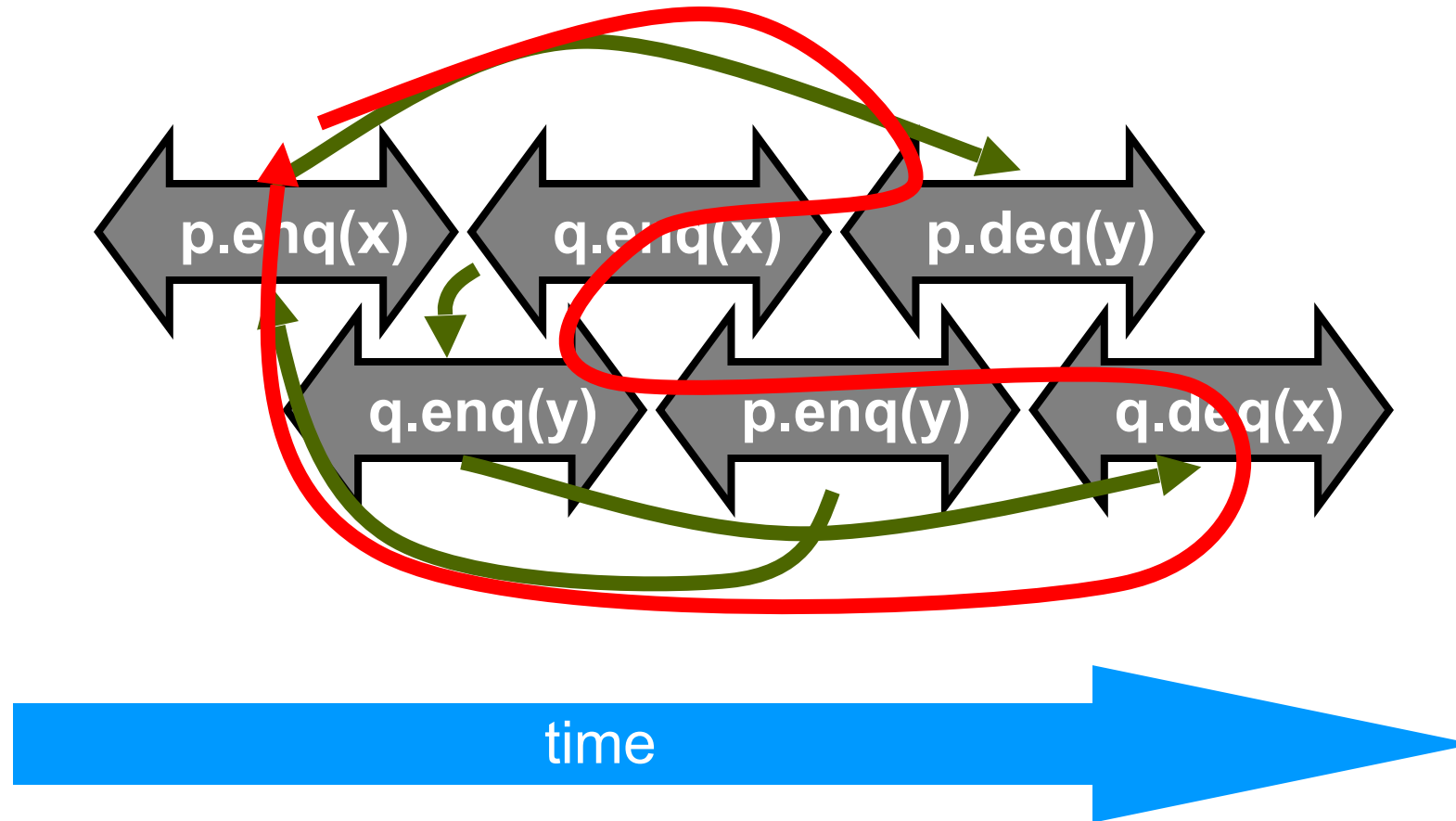
# Ordering imposed by q



# Ordering imposed by both



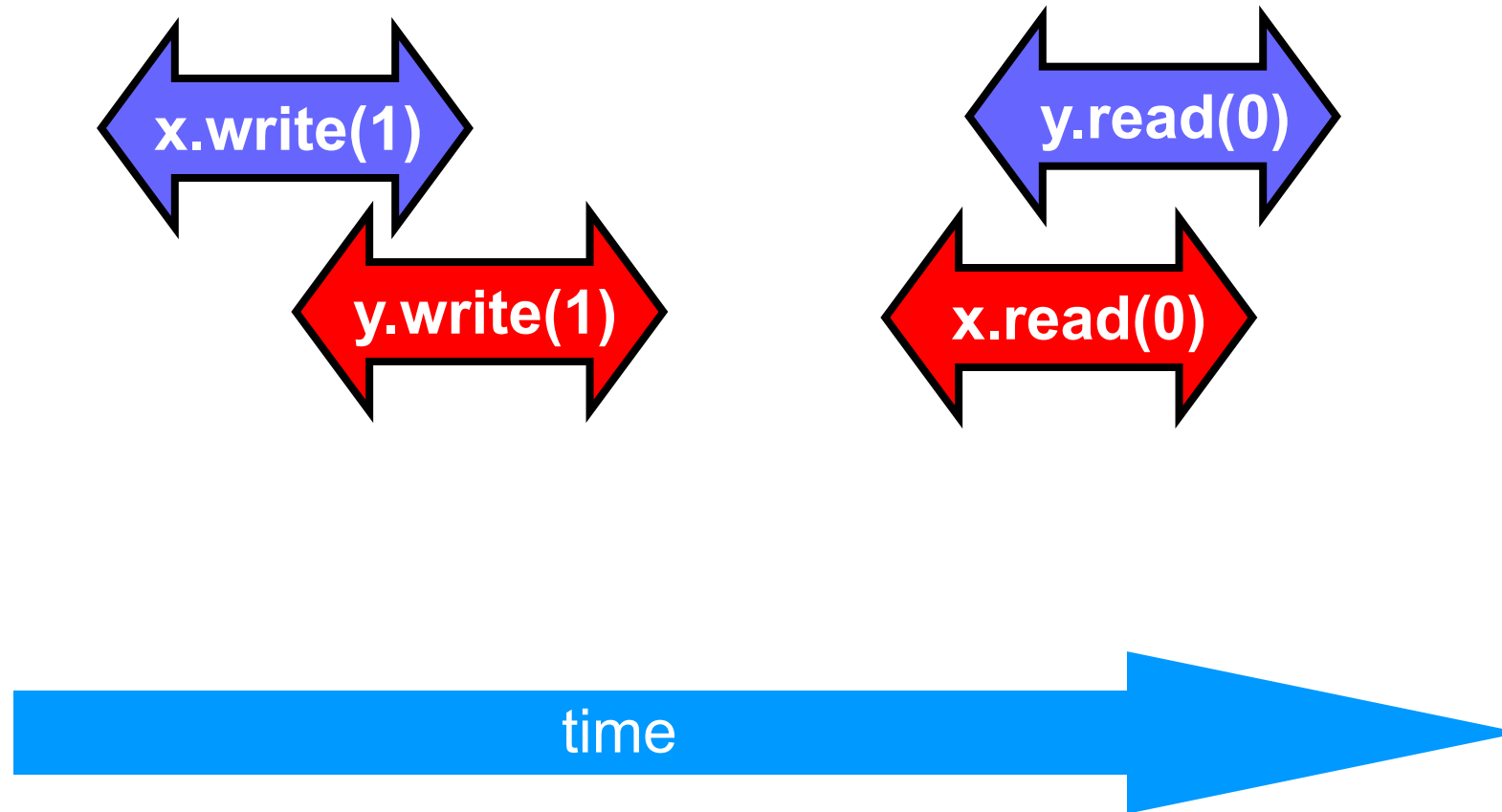
# Combining orders



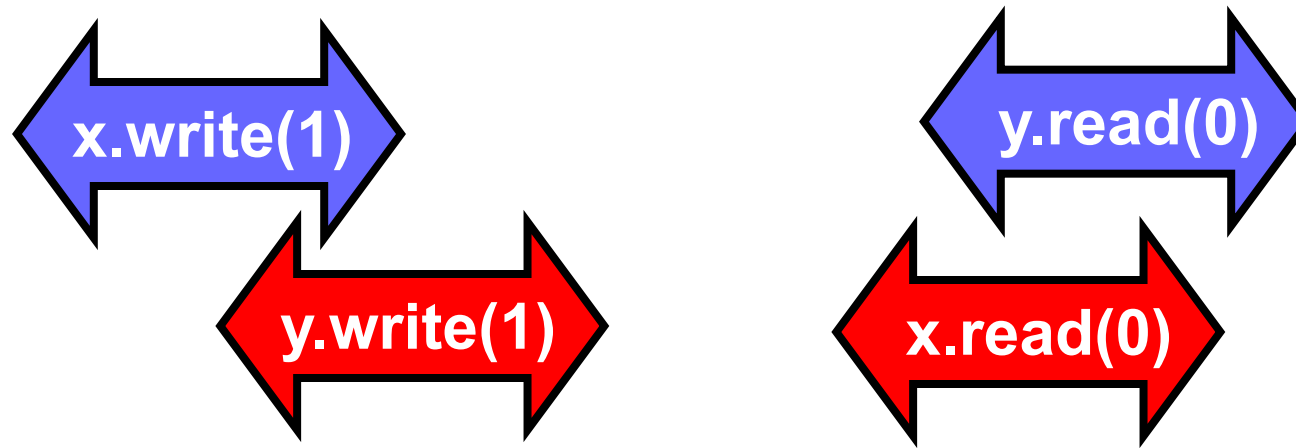
# Fact

- Most hardware architectures don't support sequential consistency
- Because they think it's too strong
- Here's another story ...

# The Flag Example



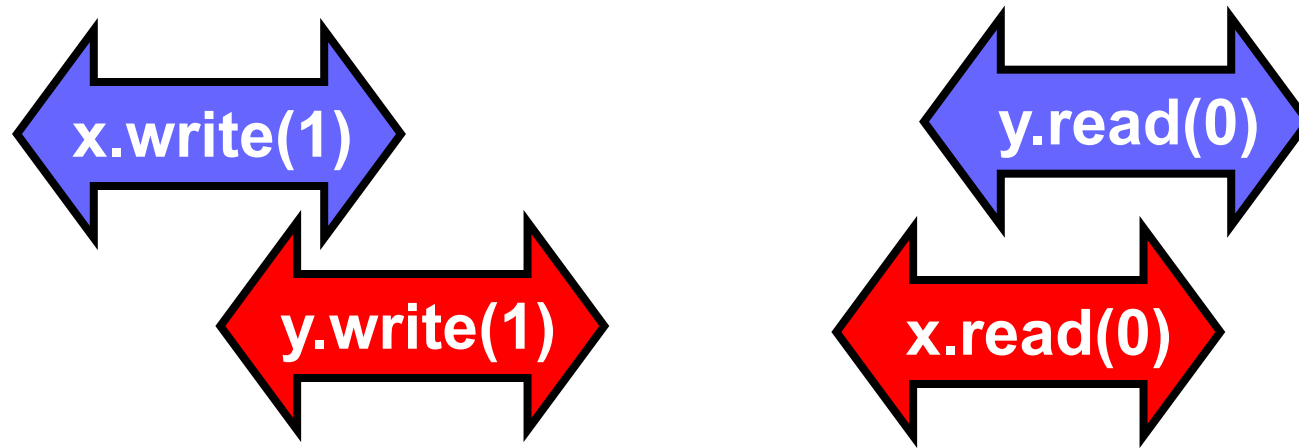
# The Flag Example



- Each thread's view is sequentially consistent
  - It went first

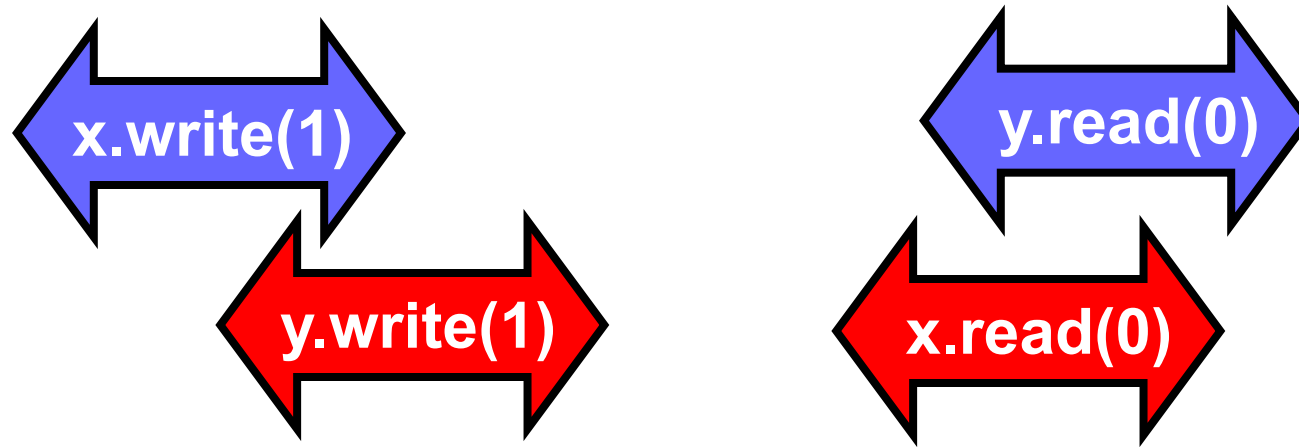


# The Flag Example



- Entire history isn't sequentially consistent
  - Can't both go first

# The Flag Example



- Is this behavior really so wrong?
  - We can argue either way ...

# Opinion: It's Wrong

- This pattern
  - Write mine, read yours
- Is exactly the flag principle
  - Beloved of Alice and Bob
  - Heart of mutual exclusion
    - Peterson
    - Bakery, etc.
- It's non-negotiable!

# Peterson's Algorithm

```
def lock(): Unit = {  
    flag(i) = true  
    victim = i  
    while (flag(1 - i) && victim == i) {}  
}  
  
def unlock(): Unit = {  
    val i = ThreadID.get  
    flag(i) = false  
}
```

# Crux of Peterson Proof

(1)  $\text{write}_B(\text{flag}[B]=\text{true}) \rightarrow$

(3)  $\text{write}_B(\text{victim}=B) \rightarrow$

(2)  $\text{write}_A(\text{victim}=A) \rightarrow \text{read}_A(\text{flag}[B])$   
 $\rightarrow \text{read}_A(\text{victim})$

# Crux of Peterson Proof

(1)  $\text{write}_B(\text{flag}[B]=\text{true}) \rightarrow$

(3)  $\text{write}_B(\text{victim}=B) \rightarrow$

(2)  $\text{write}_A(\text{victim}=A) \rightarrow \text{read}_A(\text{flag}[B])$   
 $\rightarrow \text{read}_A(\text{victim})$

Observation: proof relied on fact that if a location is stored, a later load by some thread will return this or a later stored value.

# Opinion: But It Feels So Right ...

- Many hardware architects think that sequential consistency is too strong
- Too expensive to implement in modern hardware
- OK if flag principle
  - violated by default
  - Honored by explicit request

# Hardware Consistency

**Initially,  $a = b = 0$ .**

**Processor 0**

```
mov 1, a    ;Store  
mov b, %ebx ;Load
```

**Processor 1**

```
mov 1, b    ;Store  
mov a, %eax ;Load
```

What are the final possible values of `%eax` and `%ebx` after both processors have executed?

Sequential consistency implies that no execution ends with `%eax = %ebx = 0`

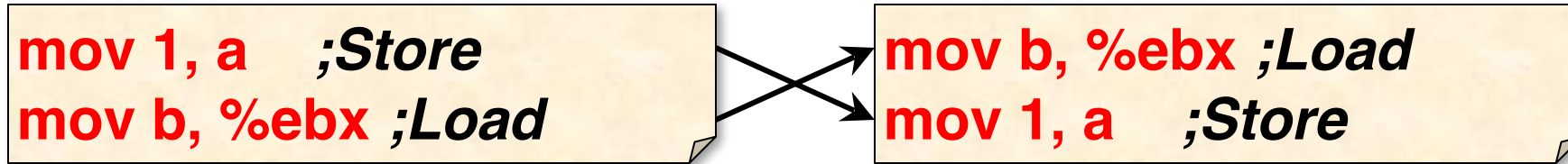


# Hardware Consistency

- No modern-day processor implements sequential consistency.
- Hardware actively reorders instructions.
- Compilers may reorder instructions, too.
- Why?
- Because most of performance is derived from a single thread's unsynchronized execution of code!

This is known as **Weak (Relaxed) Memory Semantics**

# Weak-Memory Instruction Reordering

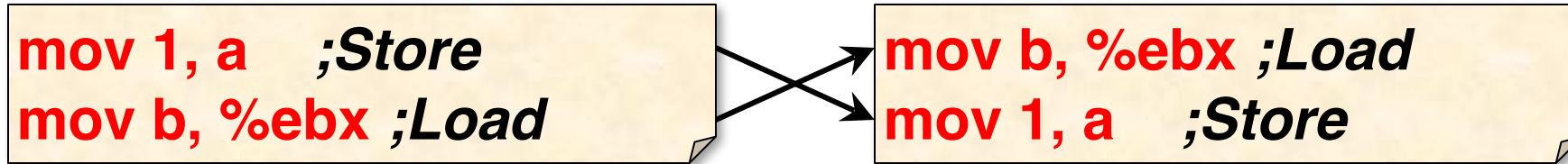


Program Order

Execution Order

- Q. Why might the hardware or compiler decide to reorder these instructions?
- A. To obtain higher performance by covering load latency — *instruction-level parallelism*.

# Weak-Memory Instruction Reordering



Program Order

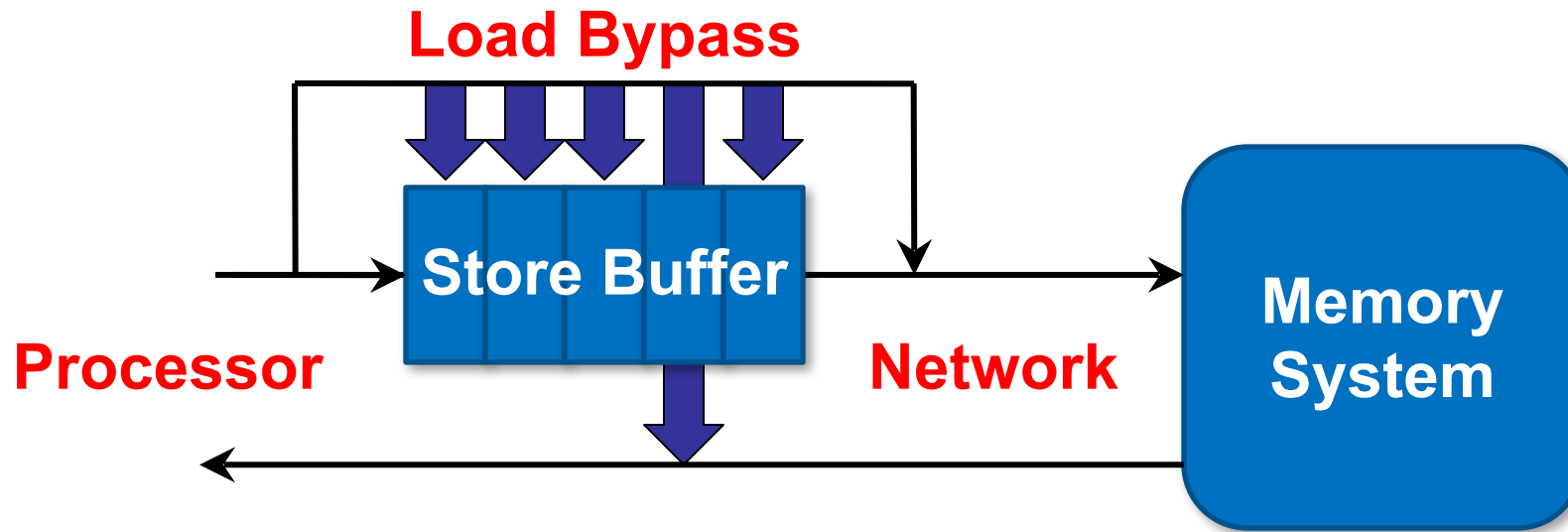
Execution Order

Q. When is it safe for the hardware or compiler to perform this reordering?

A. When  $a \neq b$ .

A'. And there's no concurrency.

# Hardware Reordering



- Processor can issue stores faster than the network can handle them  $\Rightarrow$  store buffer.
- Loads take priority, bypassing the store buffer.
- Except if a load address matches an address in the store buffer, the store buffer returns the result.

# X86 Relaxed Memory Model

## Thread's Code



~~Store1~~  
~~Store2~~  
Load1  
~~Load2~~  
~~Store3~~  
Store4  
Load3  
~~Load4~~  
~~Load5~~

1. Loads *are not* reordered with loads.
2. Stores *are not* reordered with stores.
3. Stores *are not* reordered with prior loads.
4. A load *may* be reordered with a prior store to a different location *but not* with a prior store to the same location.
5. Stores to the same location *respect a global total order*.

# X86 Relaxed Memory Model

## Thread's Code

Store1  
Store2  
Load1  
Load2  
Store3  
Store4  
Load3  
Load4  
Load5

↑  
L  
O  
A  
D  
S

1. Loads *are not* reordered with loads.
2. Stores *are not* reordered with stores.
3. **Total Store Ordering (TSO)** ...weaker than sequential consistency
4. *with a prior store to the same location.*
5. Stores to the same location *respect a global total order.*

**OK!**

# Memory Barriers (Fences)

- *A memory barrier (or memory fence) is a hardware action that enforces an ordering constraint between the instructions before and after the fence.*
- *A memory barrier can be issued explicitly as an instruction (x86: mfence)*
- *The typical cost of a memory fence is comparable to that of an L2-cache access.*

# X86 Relaxed Memory Model

## Thread's Code

Store1

Store2

Load1

Load2

Store3

Store4

**Barrier**

Load3

Load4

Load5

1. Loads
2. Store
3. Store loads
4. A load store with a prior store to the same location.
5. Stores to the same location respect a global total order.

Total Store Ordering +  
properly placed memory barriers  
= sequential consistency



# Memory Barriers

- Explicit Synchronization
- Memory barrier will
  - Flush write buffer
  - Bring caches up to date
- Compilers often do this for you
  - Entering and leaving critical sections

# Java/Scala Volatile Variables

- In Java, can ask compiler to keep a variable up-to-date by declaring it **volatile**
- In Scala, use **@volatile** annotation
- Adds a memory barrier after each store
- Inhibits reordering, removing from loops, & other “compiler optimizations”

Demo: Realistic Locks

# Summary: Real-World

- Hardware weaker than sequential consistency
- Can get sequential consistency at a price
- Linearizability better fit for high-level software

# Linearizability

- Linearizability
  - Operation takes effect instantaneously between invocation and response
  - Uses sequential specification, locality implies composability

# Summary: Correctness

- Sequential Consistency
  - Not composable
  - Harder to work with
  - Good way to think about hardware models
- We will use *linearizability* as our consistency condition in the remainder of this course unless stated otherwise

This work is licensed under a [Creative Commons Attribution-ShareAlike 2.5 License](https://creativecommons.org/licenses/by-sa/3.0/).

- **You are free:**
  - **to Share** — to copy, distribute and transmit the work
  - **to Remix** — to adapt the work
- **Under the following conditions:**
  - **Attribution.** You must attribute the work to “The Art of Multiprocessor Programming” (but not in any way that suggests that the authors endorse you or your use of the work).
  - **Share Alike.** If you alter, transform, or build upon this work, you may distribute the resulting work only under the same, similar or a compatible license.
- For any reuse or distribution, you must make clear to others the license terms of this work. The best way to do this is with a link to
  - <http://creativecommons.org/licenses/by-sa/3.0/>.
- Any of the above conditions can be waived if you get permission from the copyright holder.
- Nothing in this license impairs or restricts the author's moral rights.