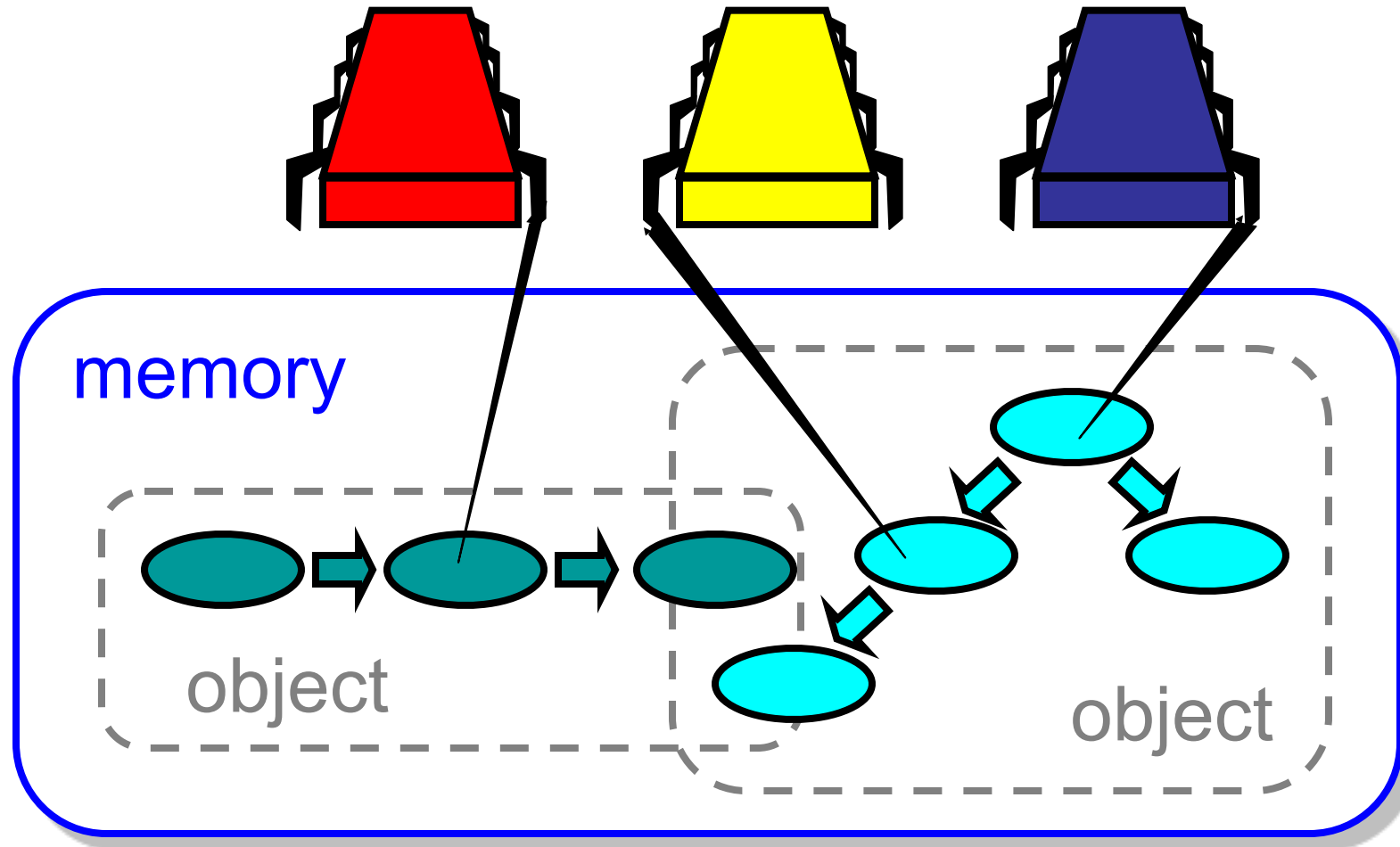


YSC3242: Parallel, Concurrent and Distributed Programming

Concurrent Objects

Concurrent Computation



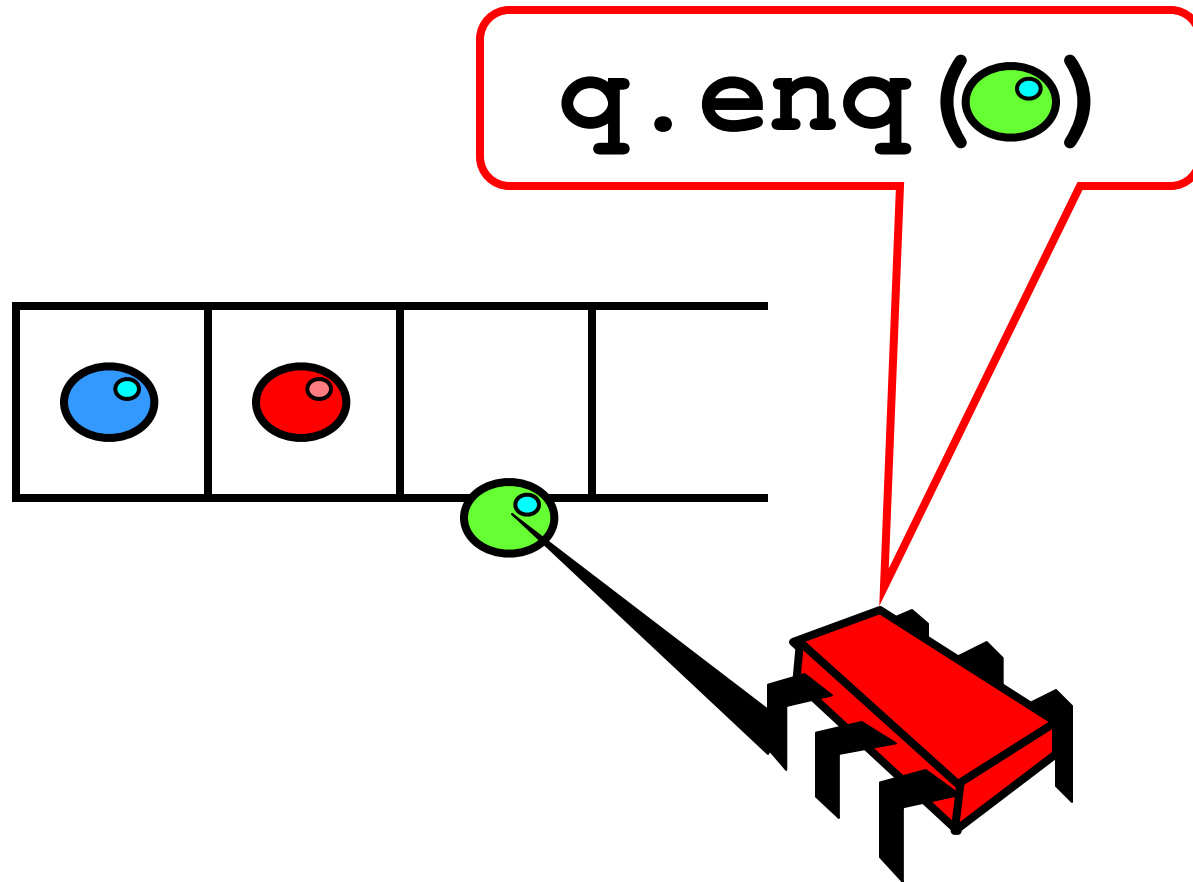
Objectivism

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

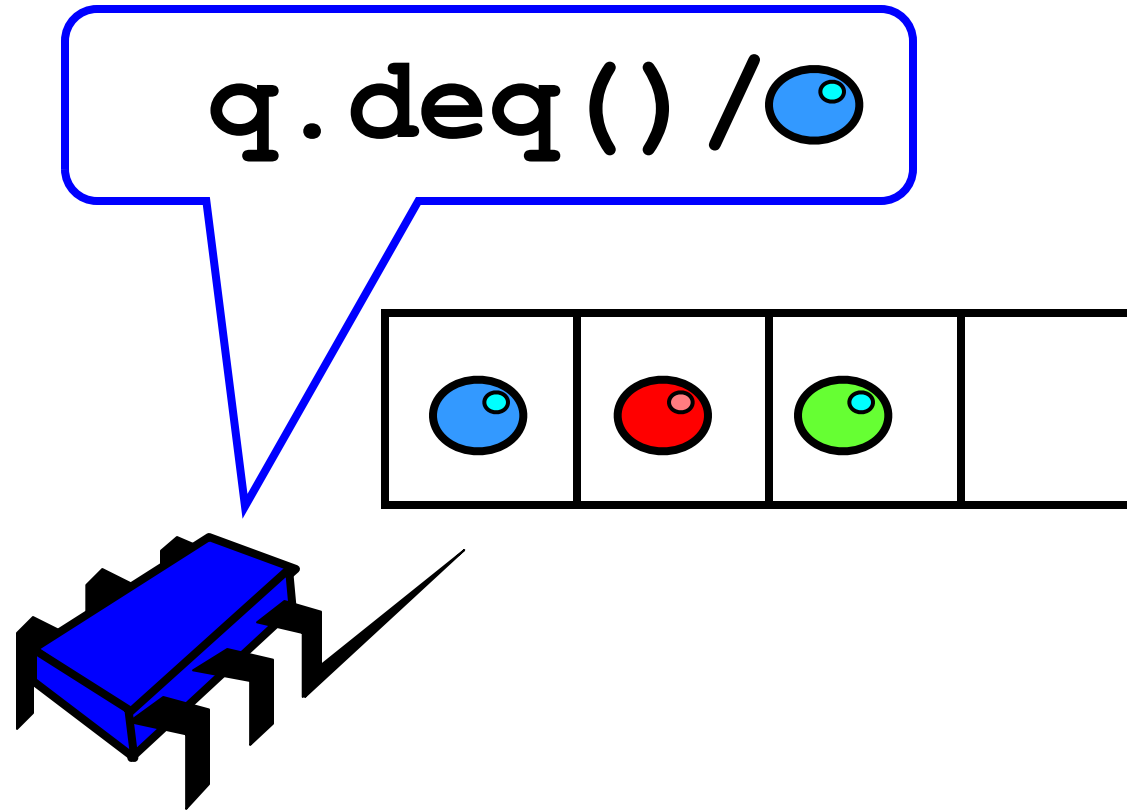
Objectivism

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

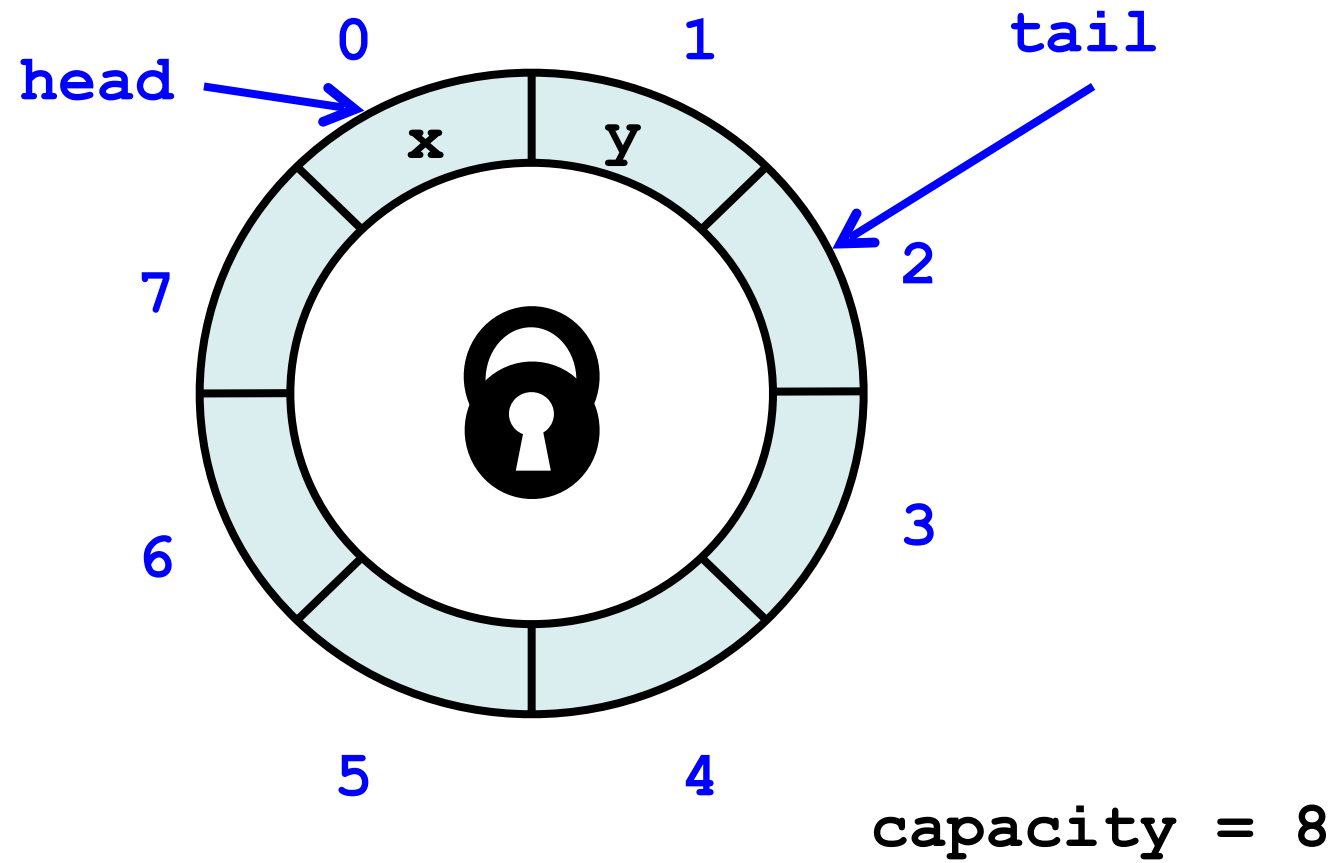
FIFO Queue: Enqueue Method



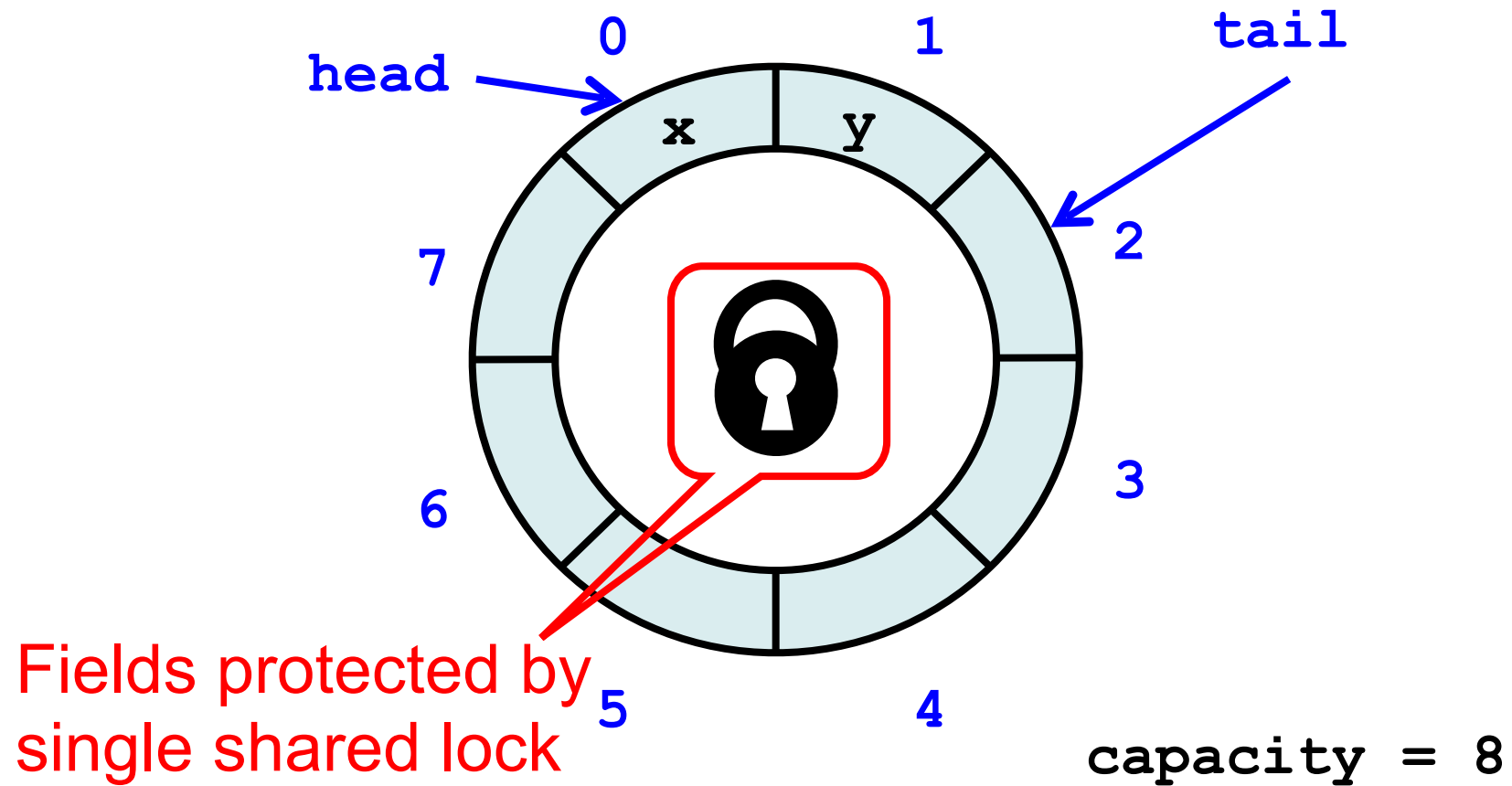
FIFO Queue: Dequeue Method



Lock-Based Queue



Lock-Based Queue

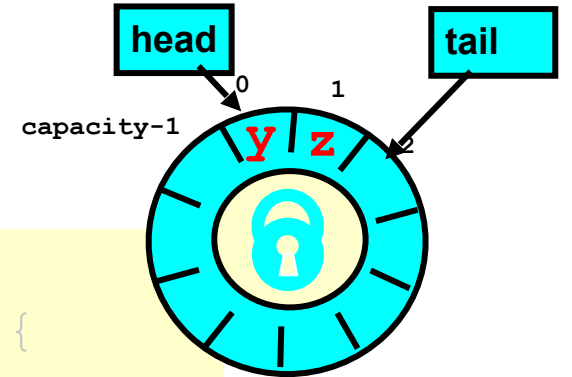


A Lock-Based Queue

```
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

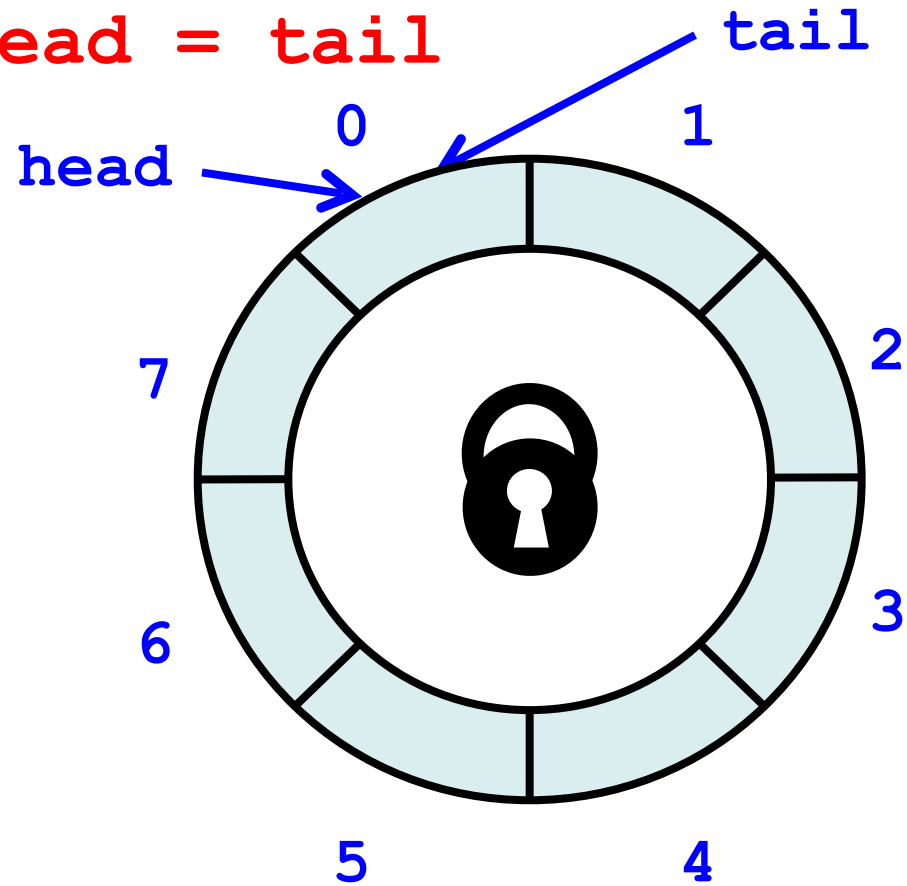
```
class LockBasedQueue[T: ClassTag]  
  (val capacity: Int) extends MyQueue[T] {  
  
    private var head, tail: Int = 0  
    private val items = new Array[T](capacity)  
    private val myLock = new ReentrantLock()
```



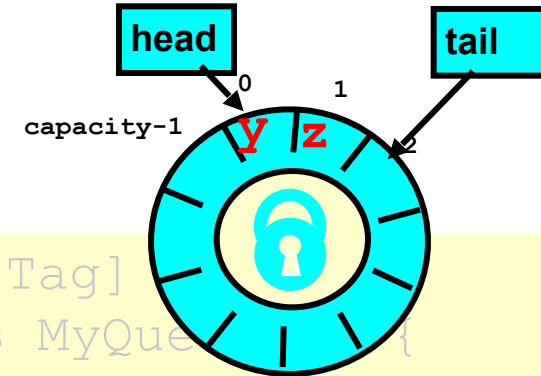
Fields protected by
single shared lock

Lock-Based Queue

Initially: **head = tail**



Lock-Based Queue

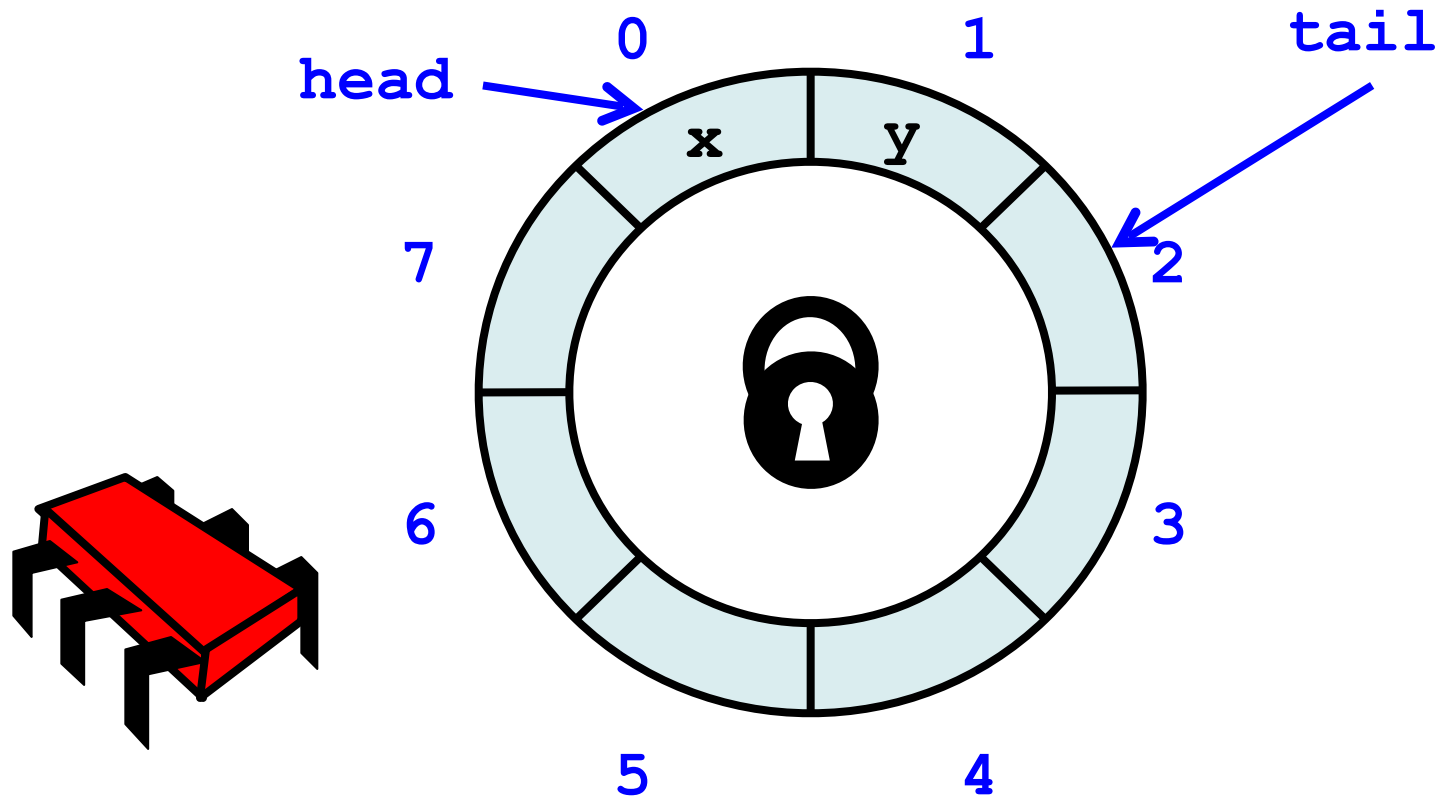


```
class LockBasedQueue[T: ClassTag]  
  (val capacity: Int) extends MyQueue {
```

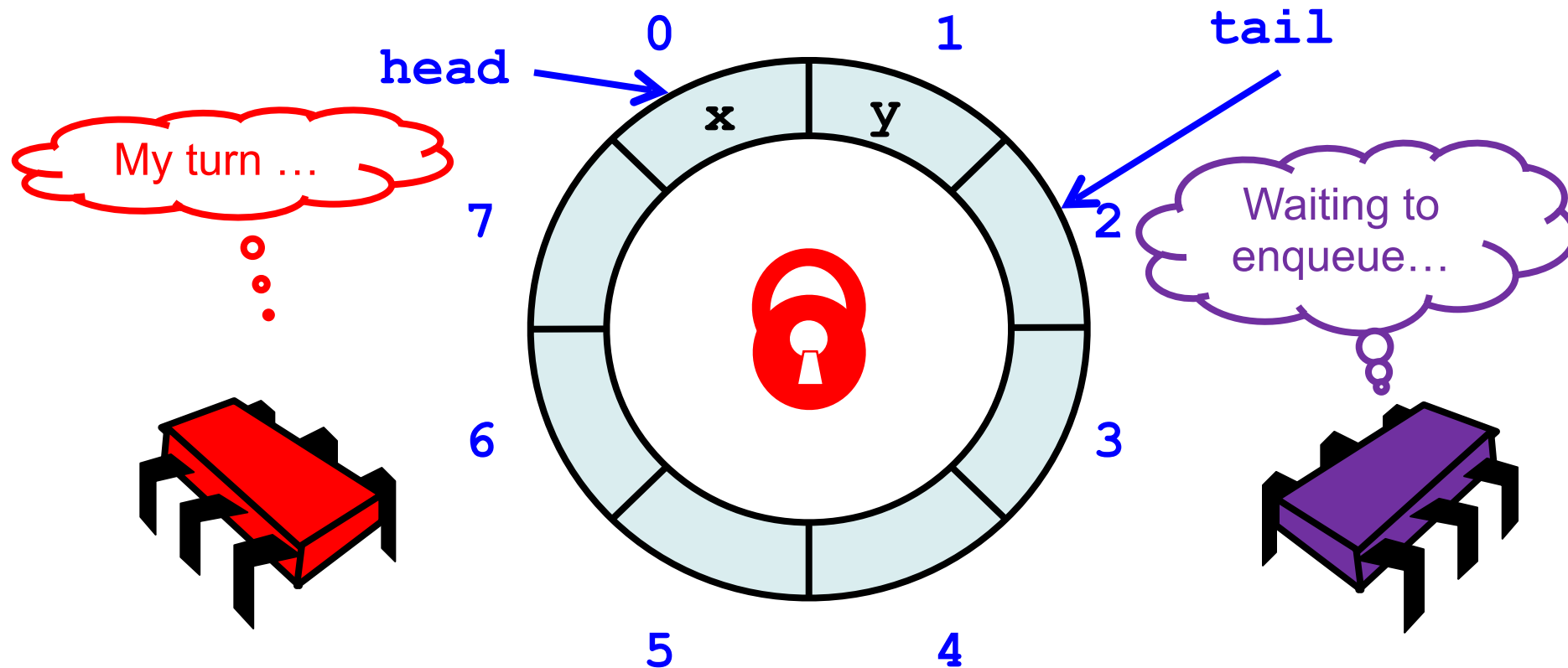
```
  private var head, tail: Int = 0  
  private val items = new Array[T](capacity)  
  private val myLock = new ReentrantLock()
```

Initially head = tail

Lock-Based `deq()`

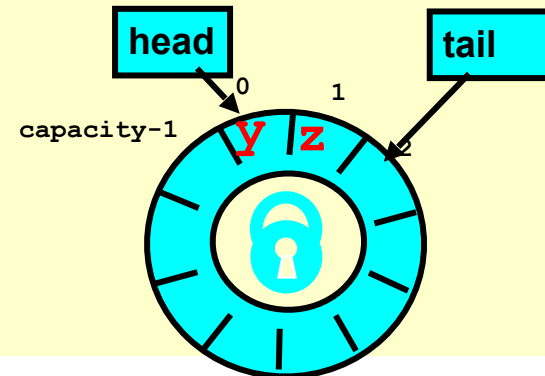


Acquire Lock



Implementation: `deq()`

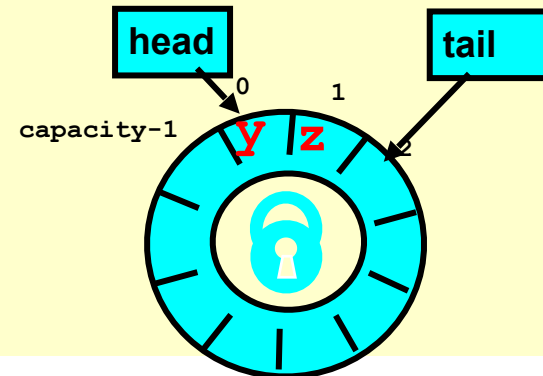
```
def deq() : T = {  
  myLock.lock()  
  try {  
    if (tail == head) {  
      throw EmptyException  
    }  
    val x = items(head % items.length)  
    head = head + 1  
    x  
  } finally {  
    myLock.unlock()  
  }  
}
```



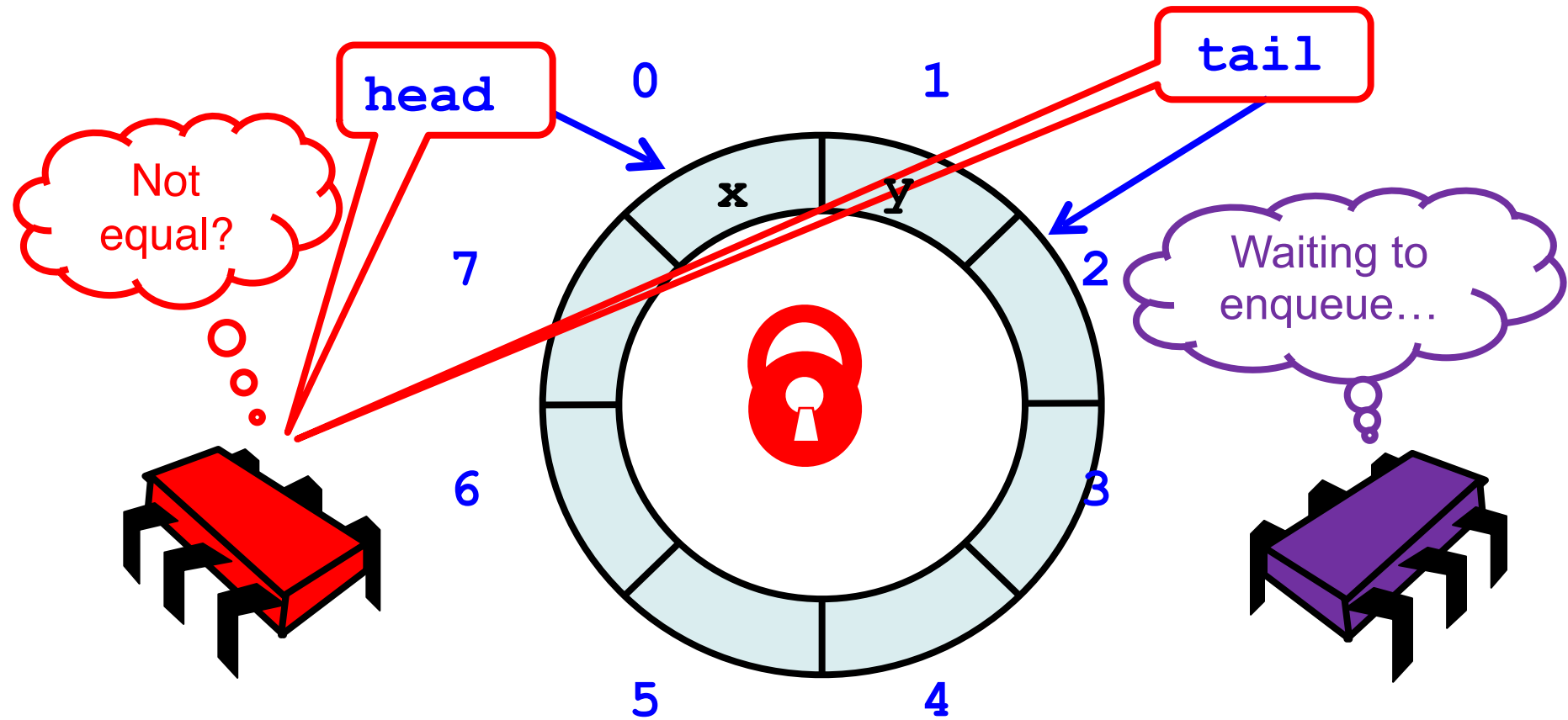
Implementation: `deq()`

```
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



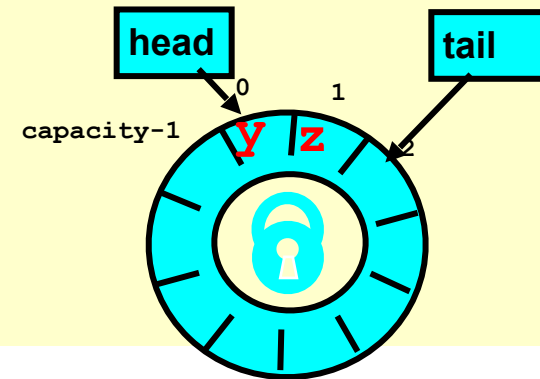
Check if Non-Empty



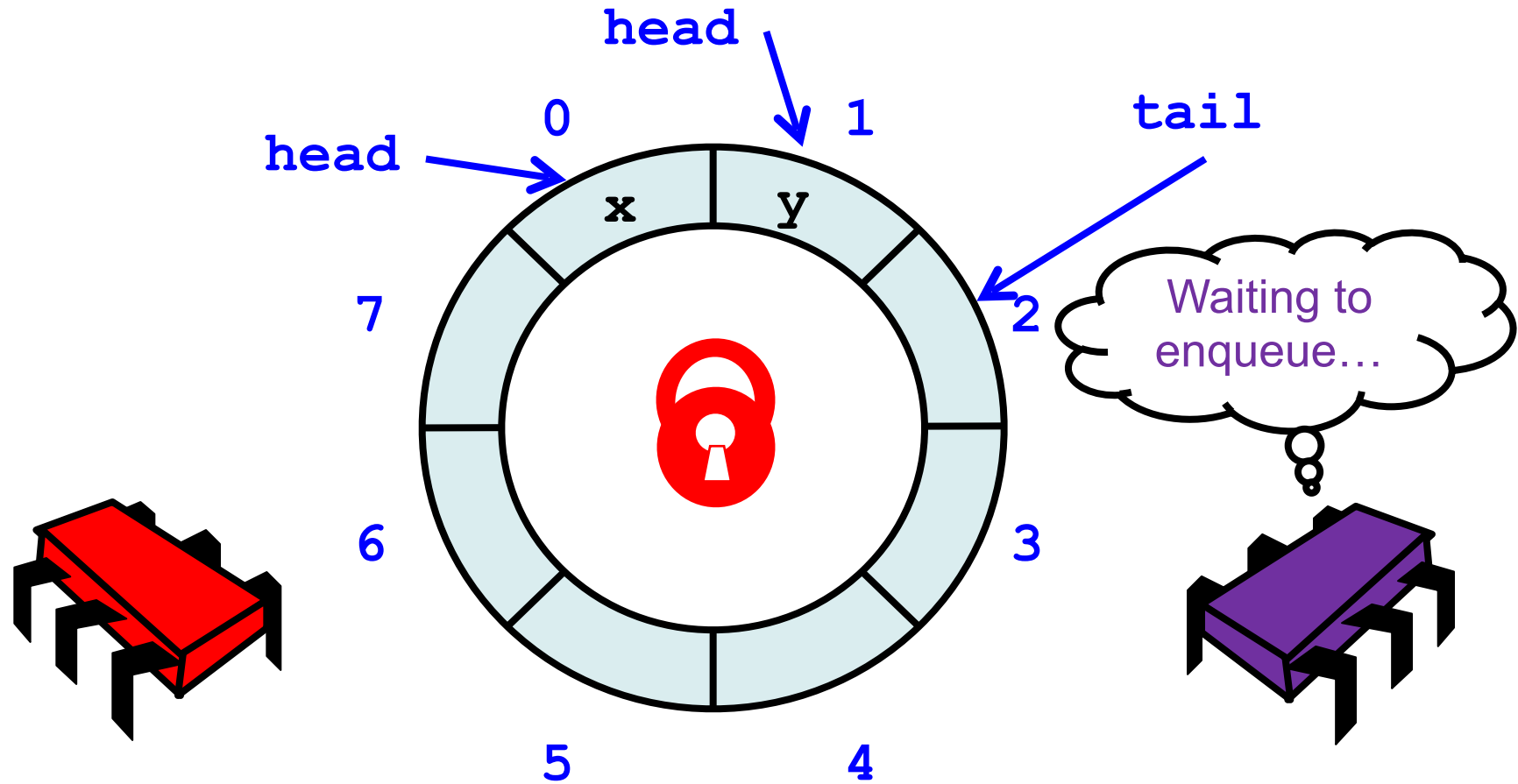
Implementation: `deq()`

```
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

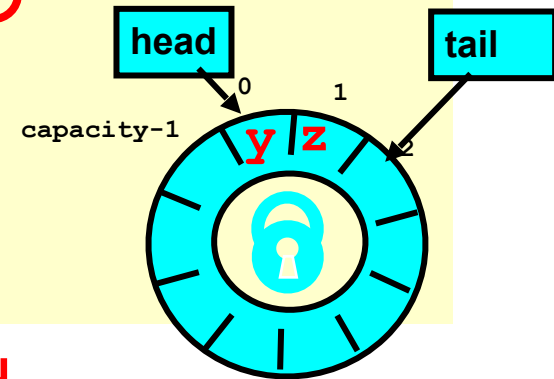


Implementation: `deq()`

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

val x = items(head % items.length)
head = head + 1

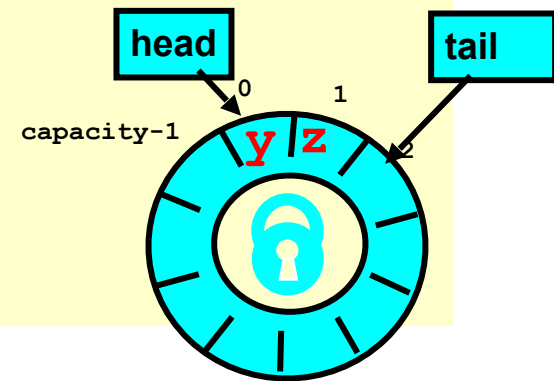
Queue not empty?
Remove item and update head



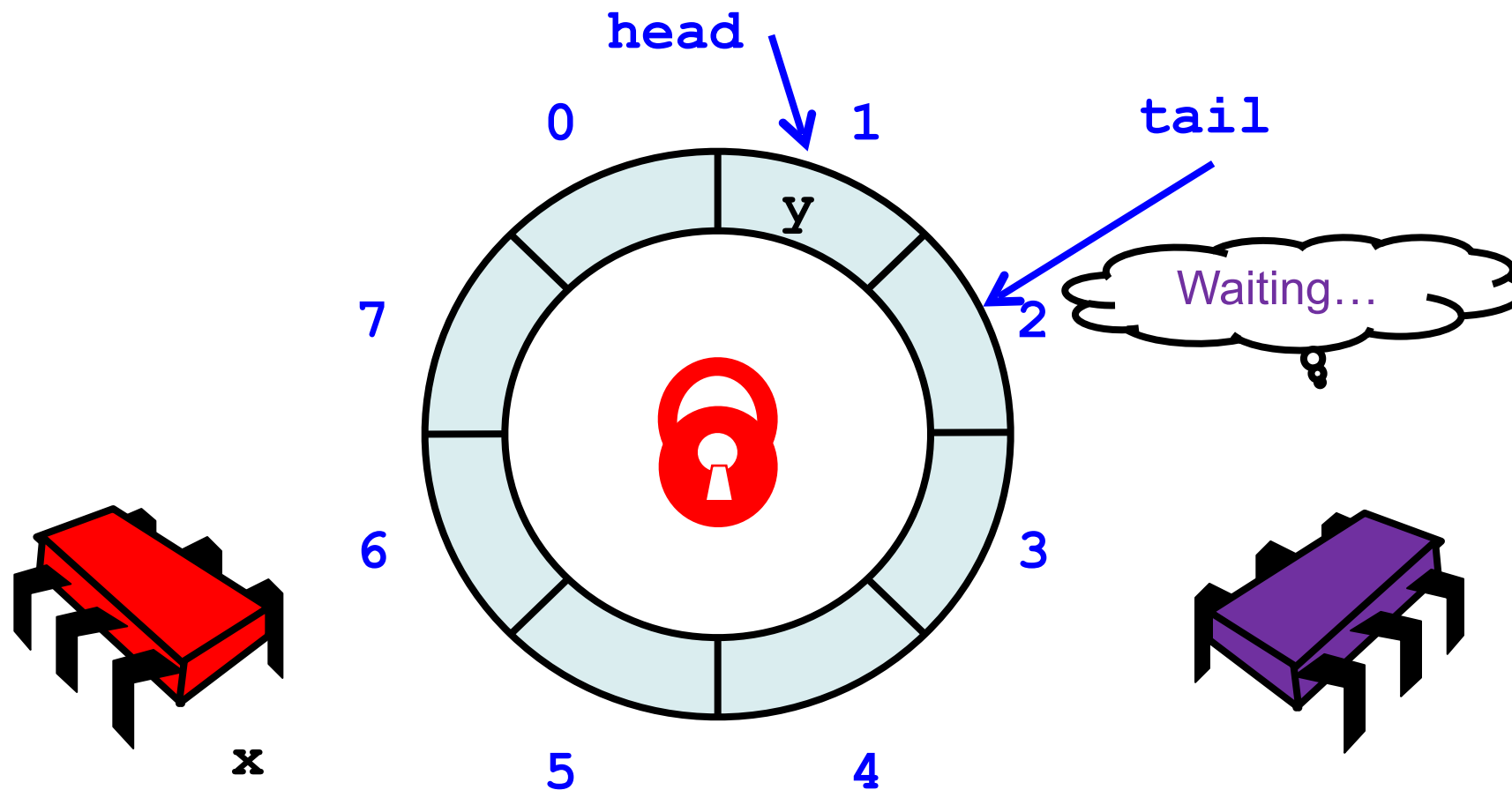
Implementation: `deq()`

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

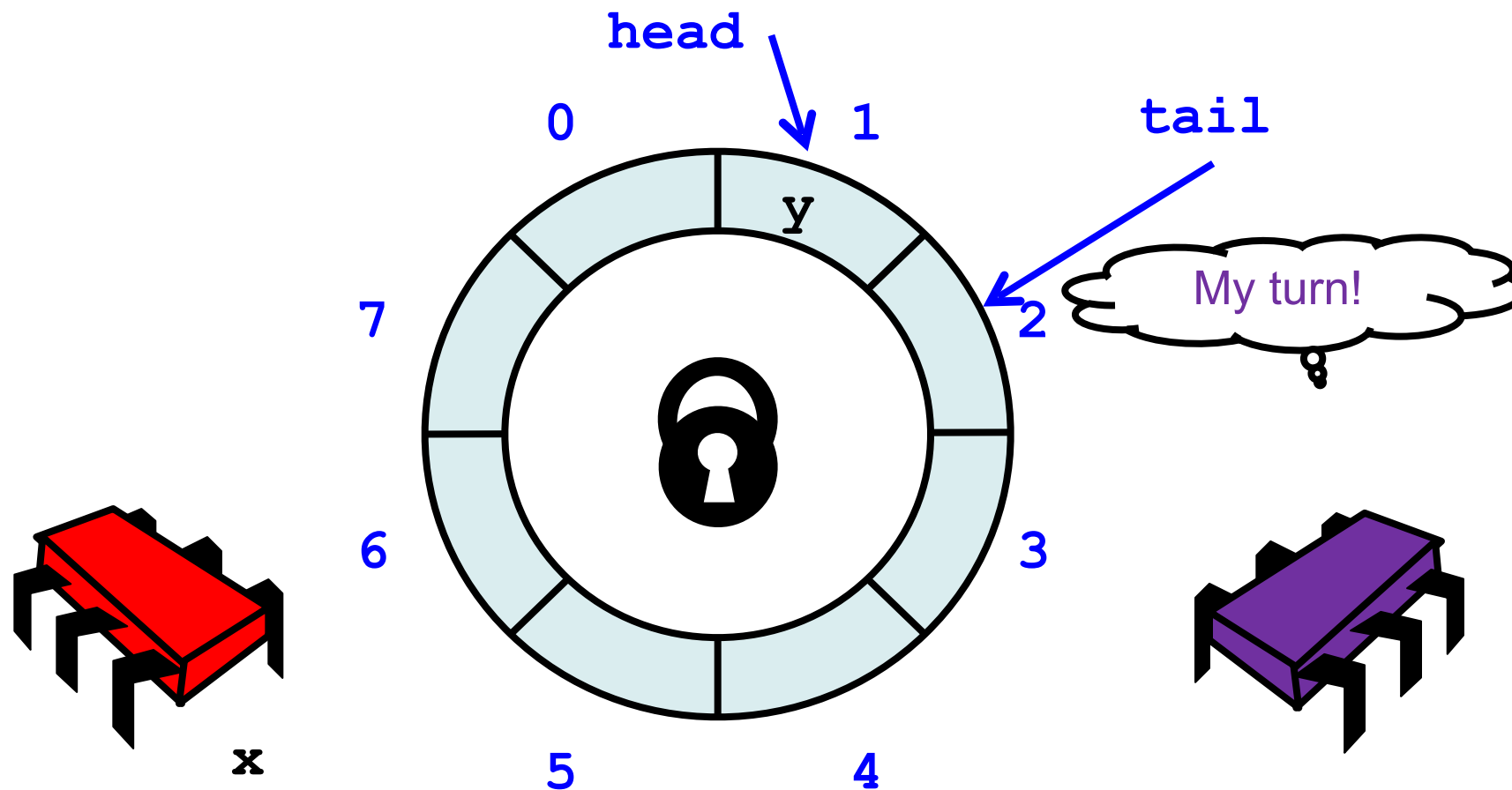
Return result



Release the Lock

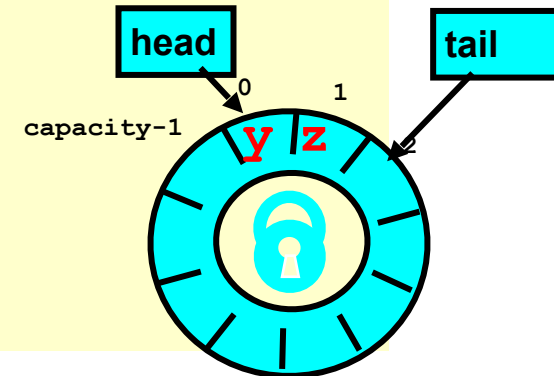


Release the Lock



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()`

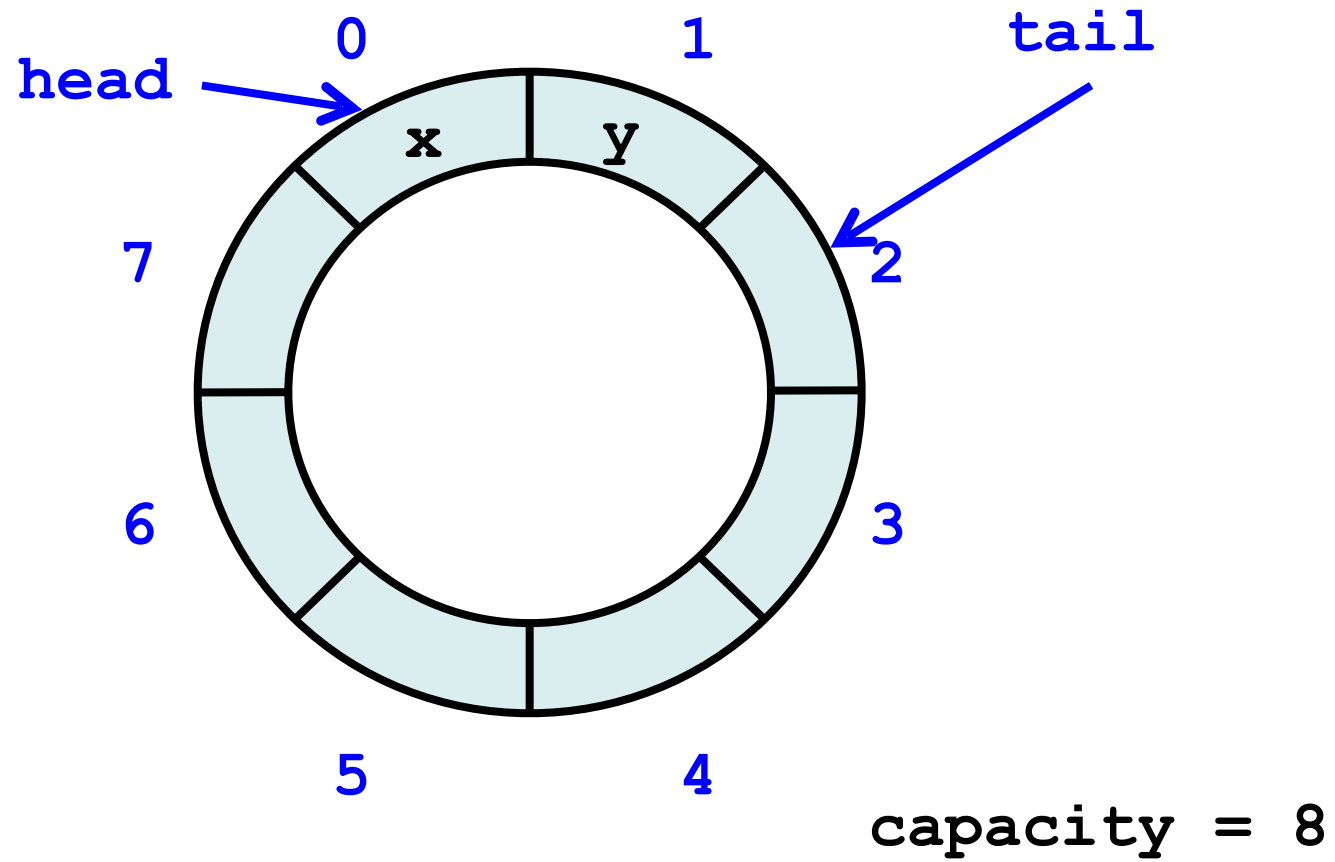
```
def deq() : T = {  
  myLock.lock()  
  try {  
    if (tail == head) {  
      throw EmptyException  
    }  
    val x = items(head % items.length)  
    head = head + 1  
    x  
  } finally {  
    myLock.unlock()  
  }  
}
```

Should be correct because
modifications are mutually exclusive...

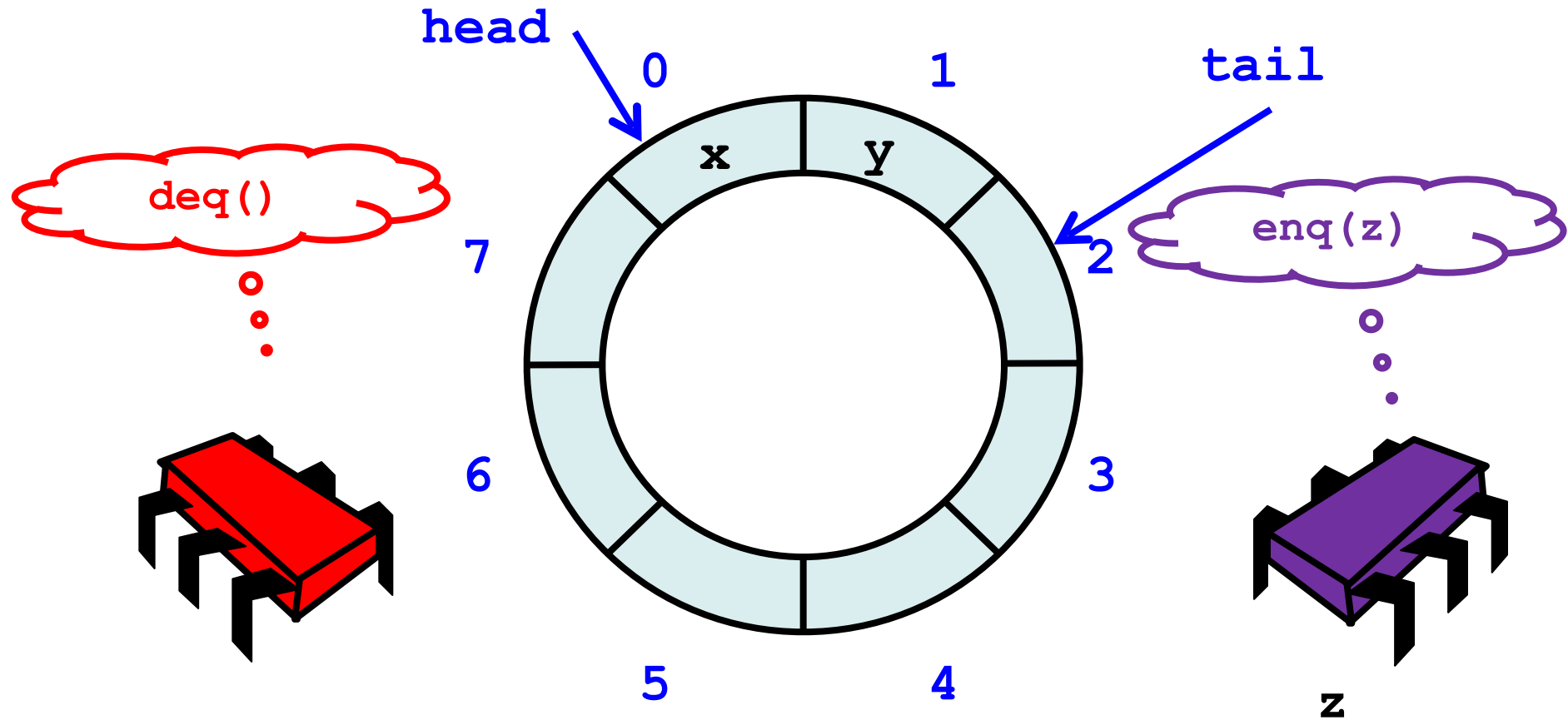
Now consider the following implementation

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

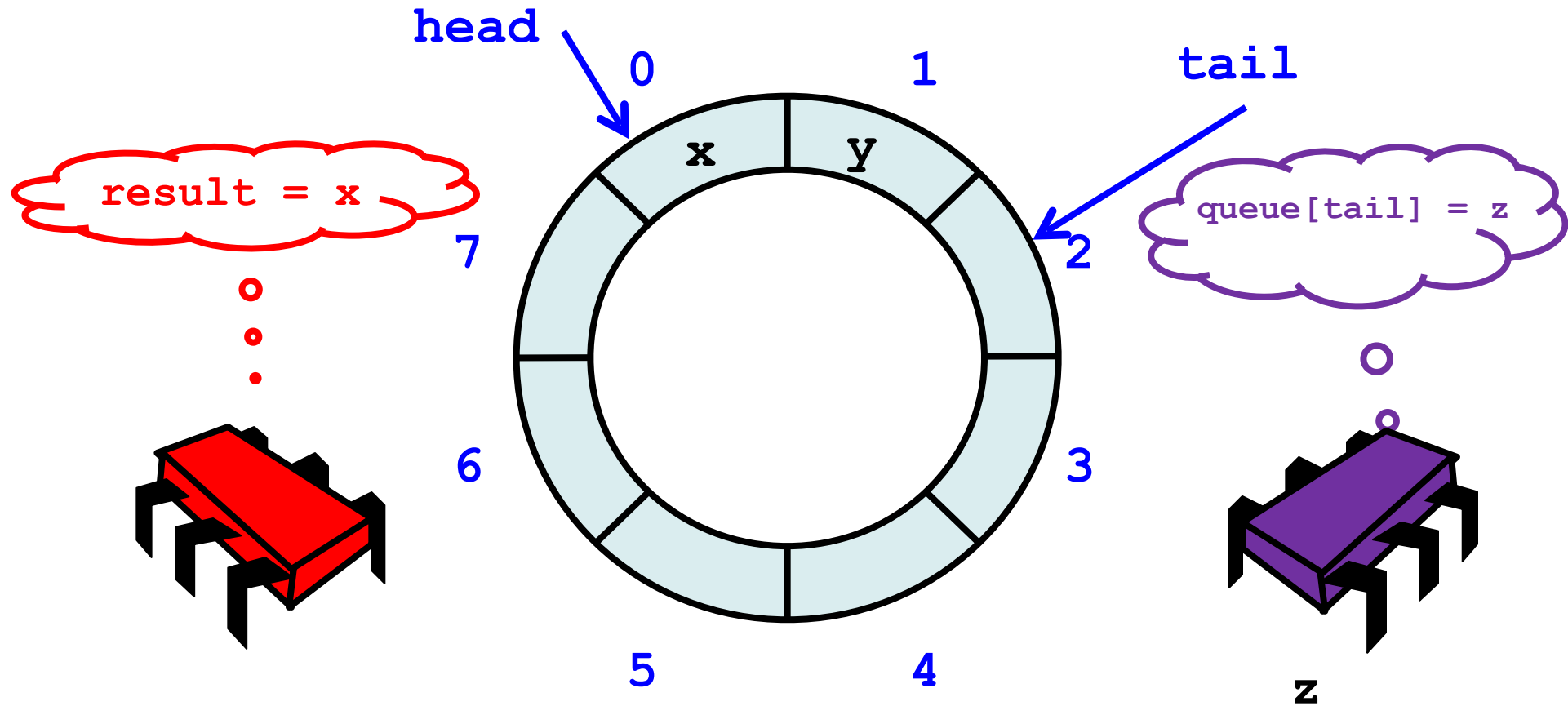
Wait-free 2-Thread Queue



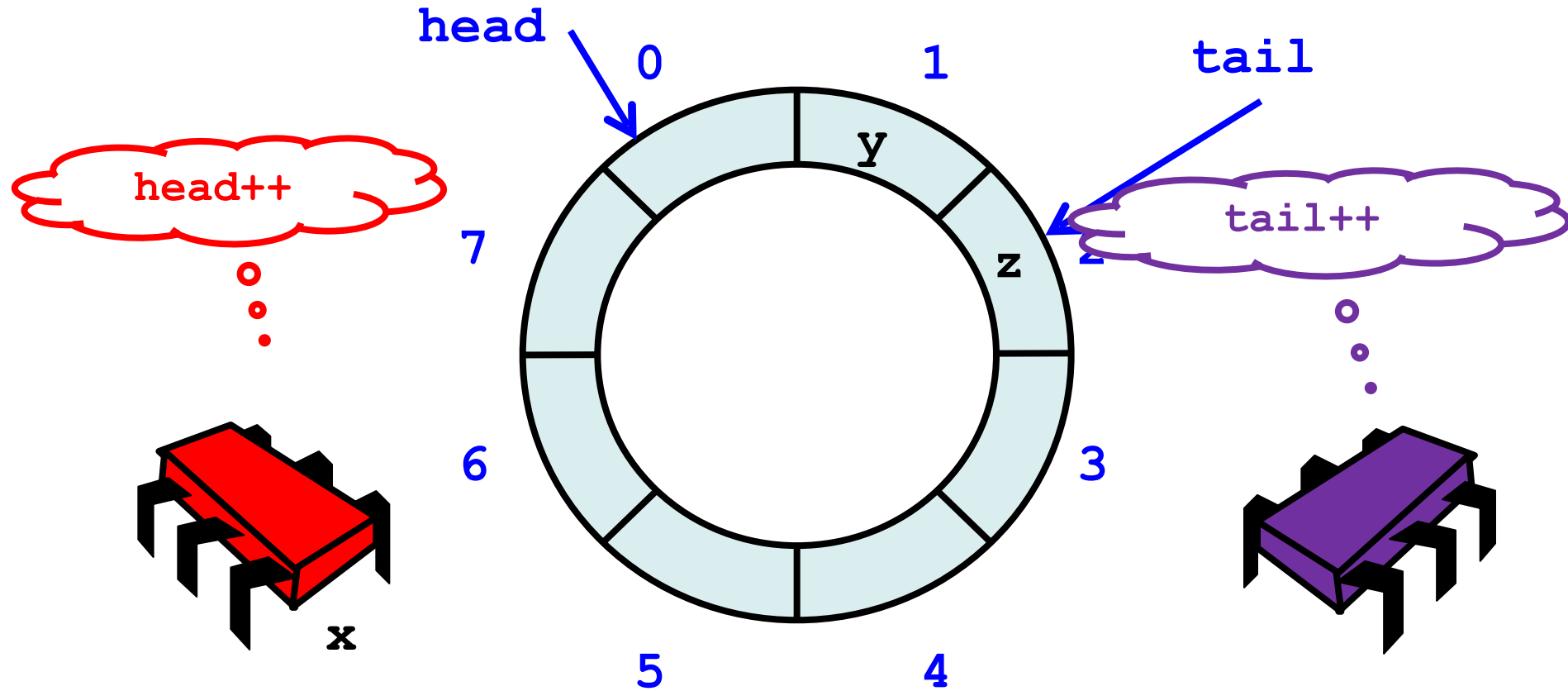
Wait-free 2-Thread Queue



Wait-free 2-Thread Queue

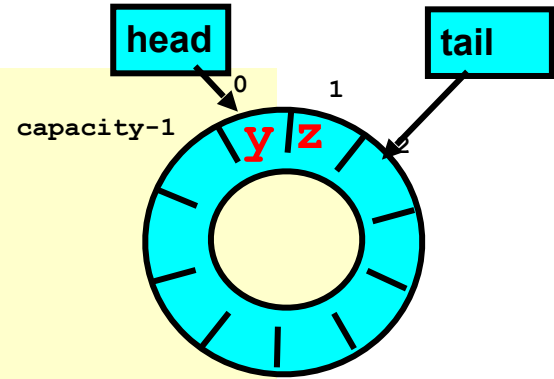


Wait-free 2-Thread Queue



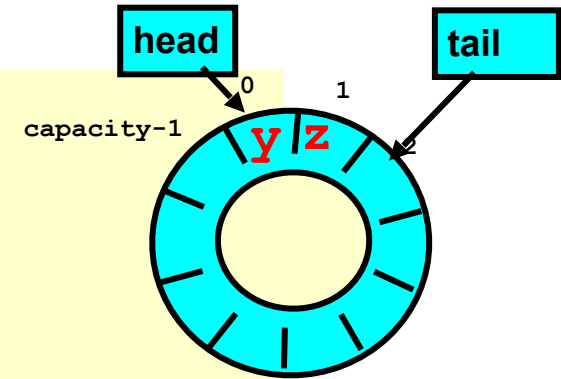
Wait-free 2-Thread Queue

```
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)
 - the method will return a particular value
 - or throw a particular exception,
- and (postcondition, con't)
 - the object will be in some other state
 - when the method returns

Pre and PostConditions for Dequeue

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

Pre and PostConditions for Dequeue

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

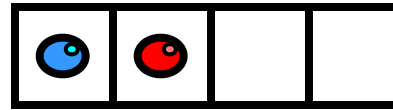
Why Sequential Specifications *Totally Rock*

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

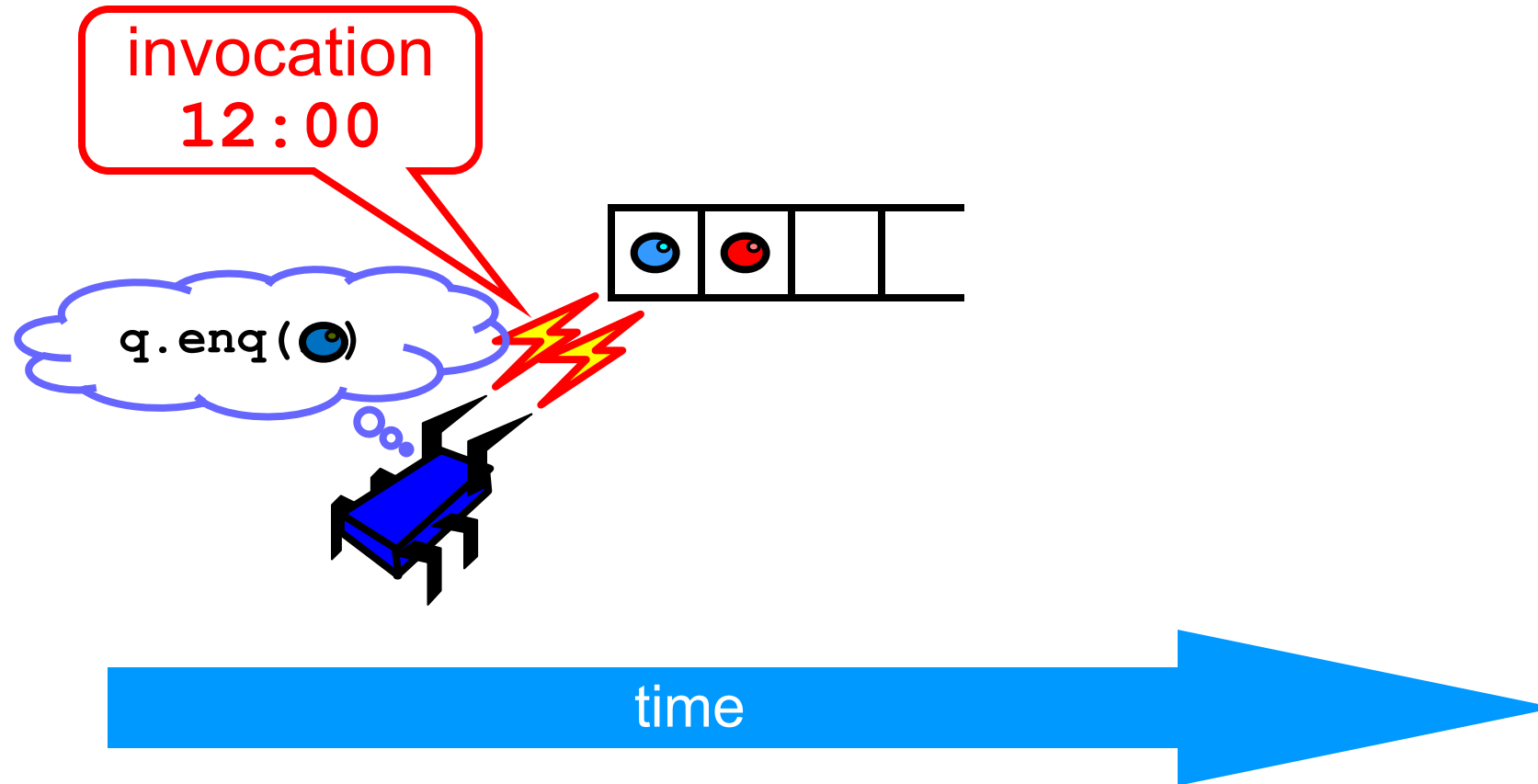
What About Concurrent Specifications ?

- Methods?
- Documentation?
- Adding new methods?

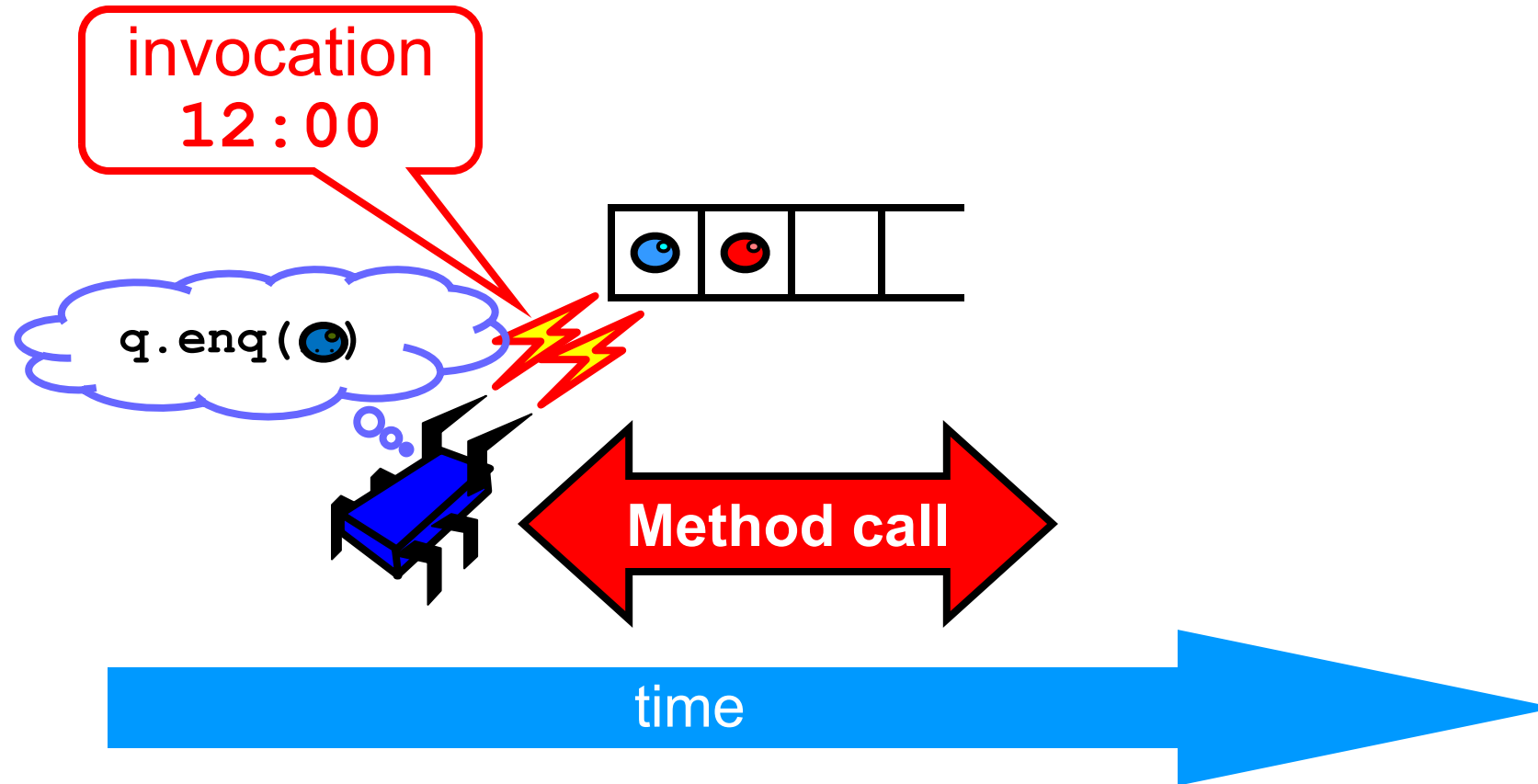
Methods Take Time



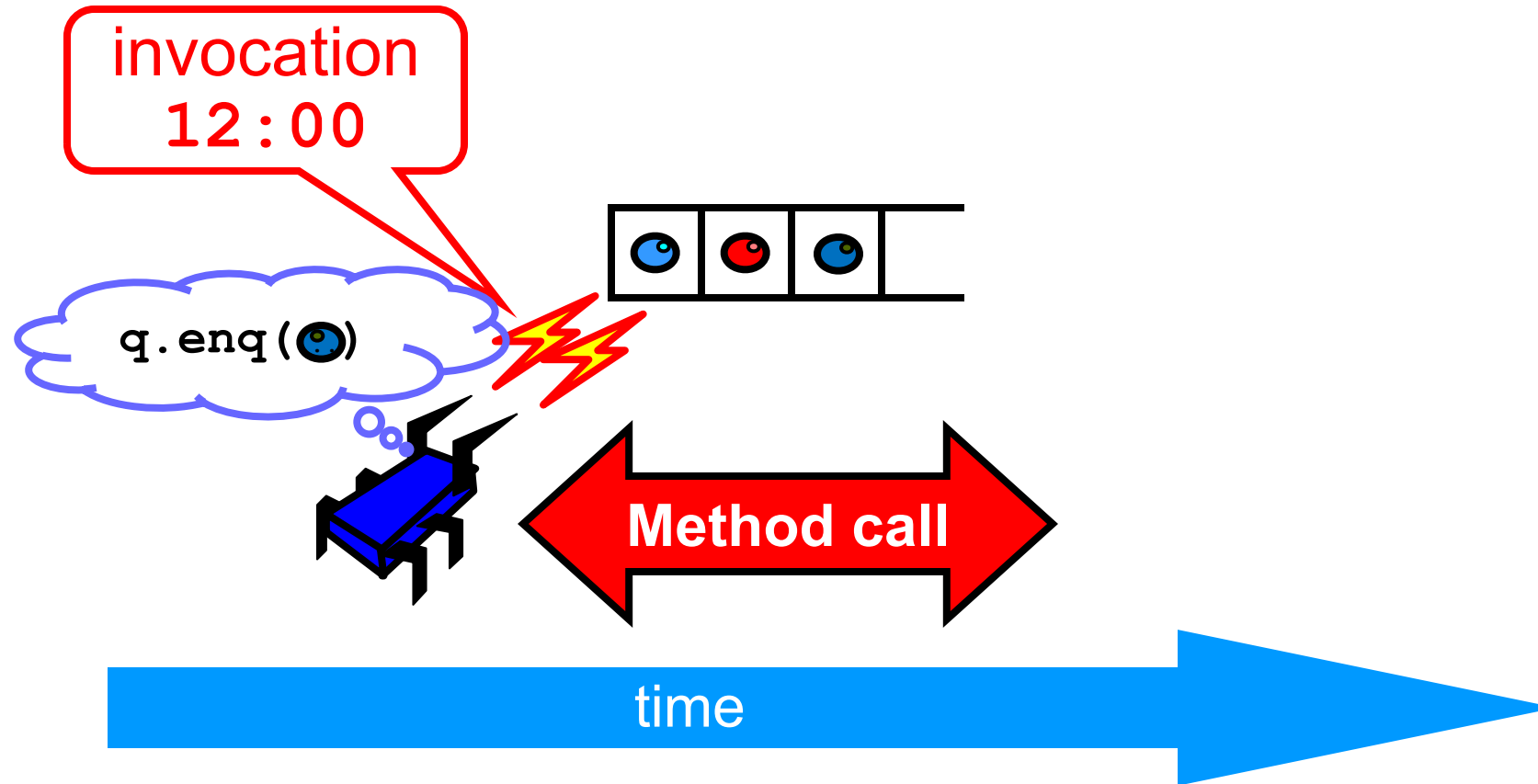
Methods Take Time



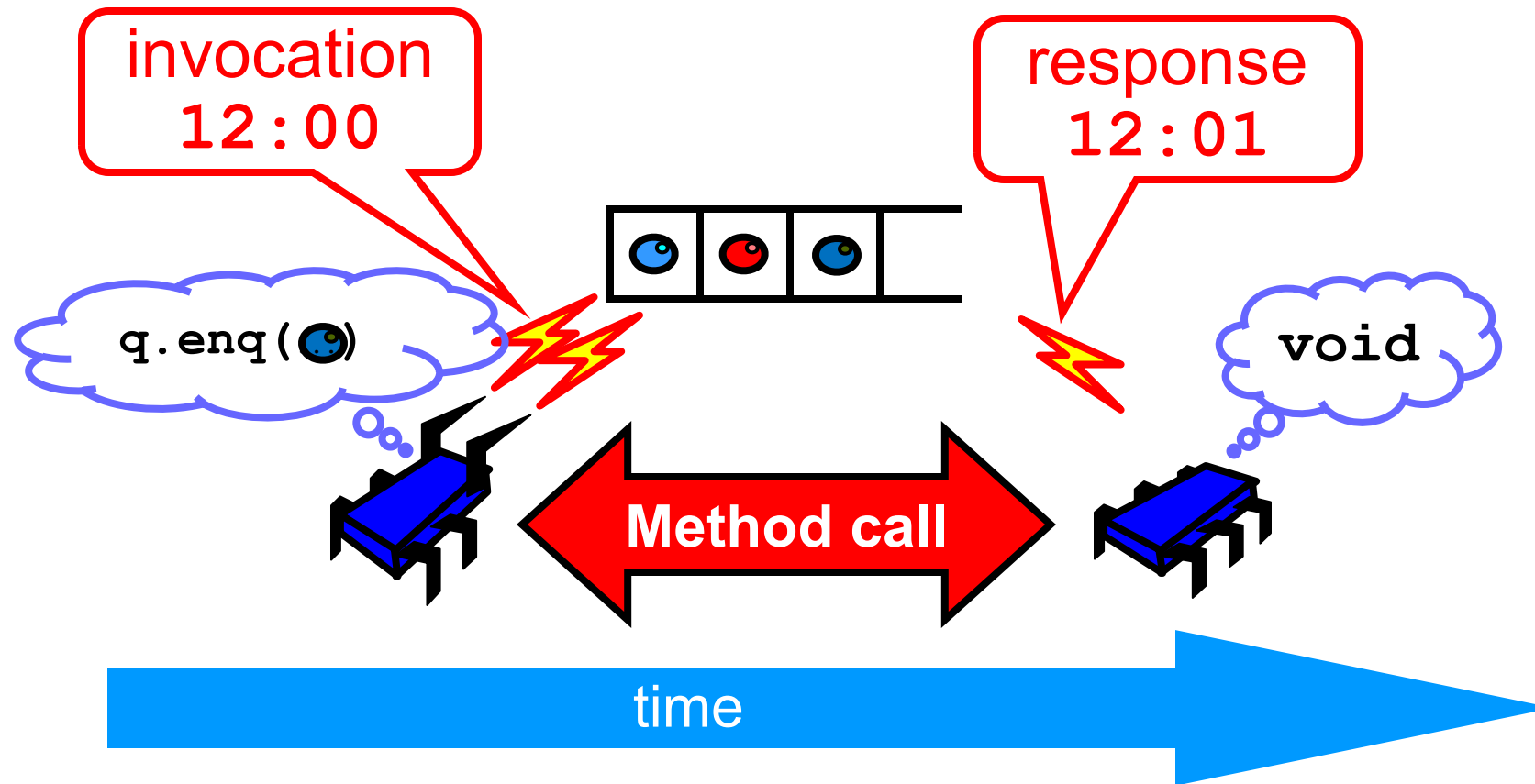
Methods Take Time



Methods Take Time



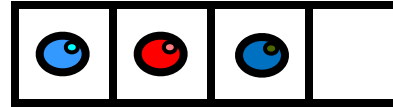
Methods Take Time



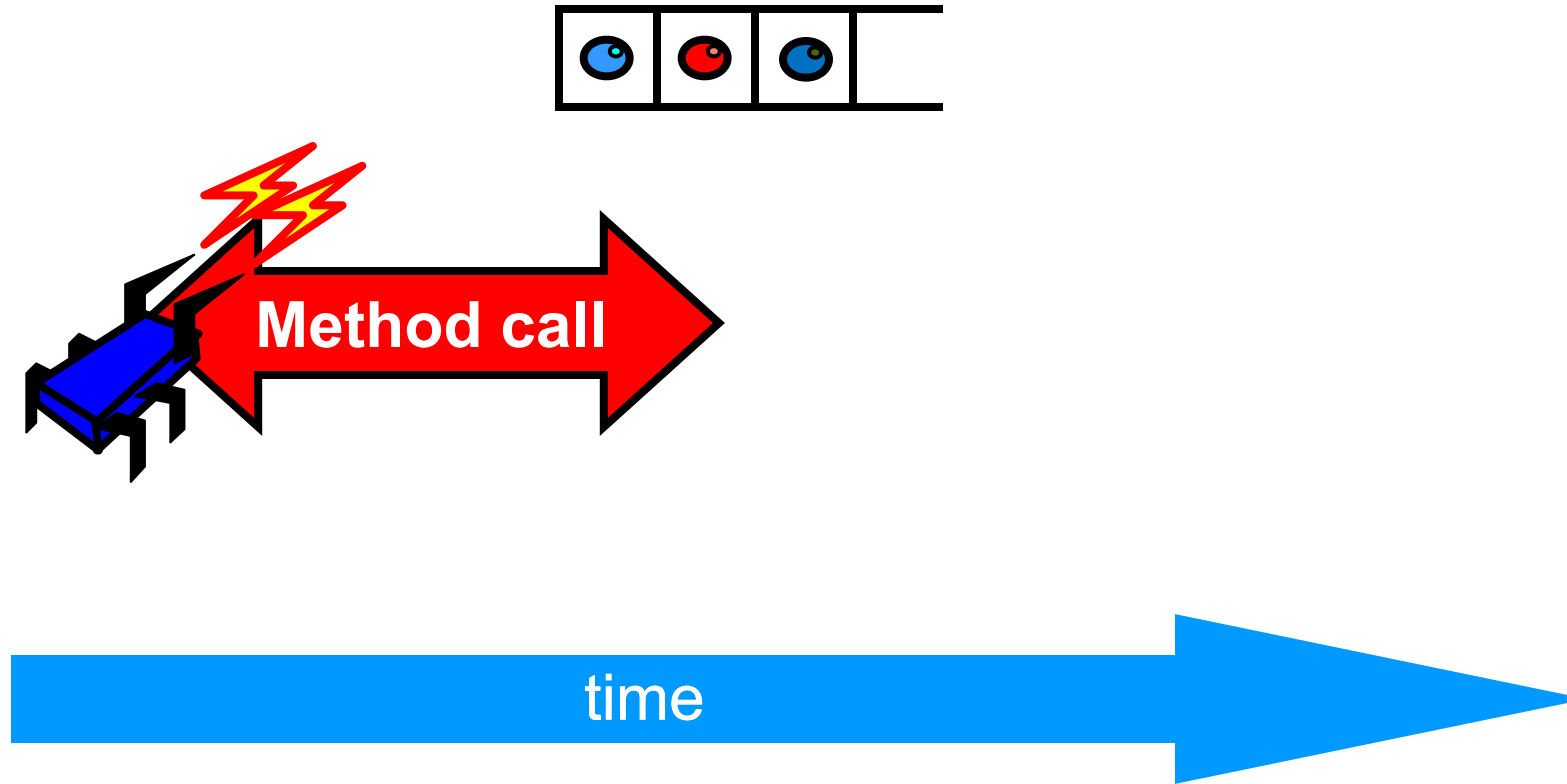
Sequential vs Concurrent

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

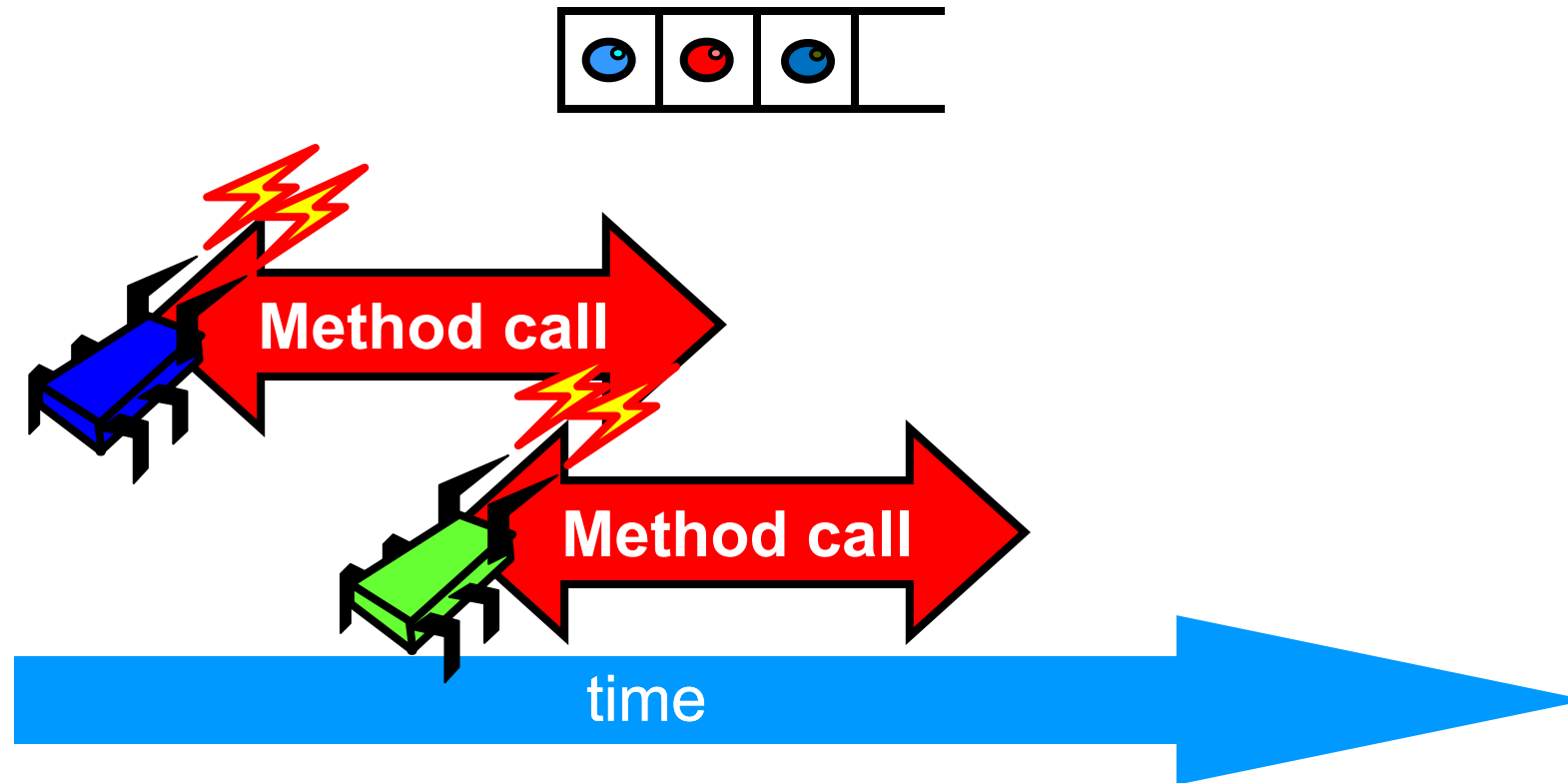
Concurrent Methods Take **Overlapping** Time



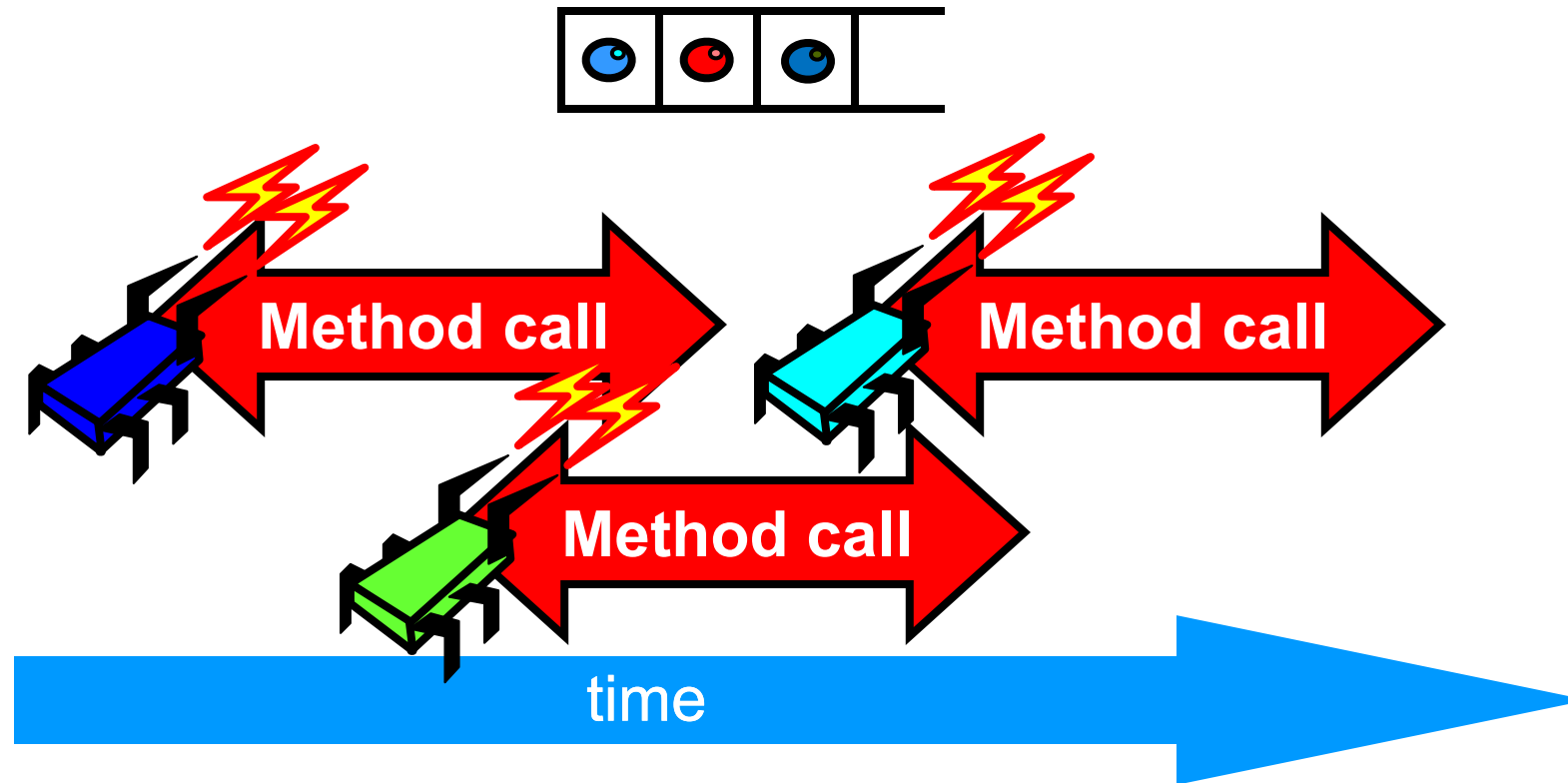
Concurrent Methods Take **Overlapping** Time



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()  
  }  
}
```

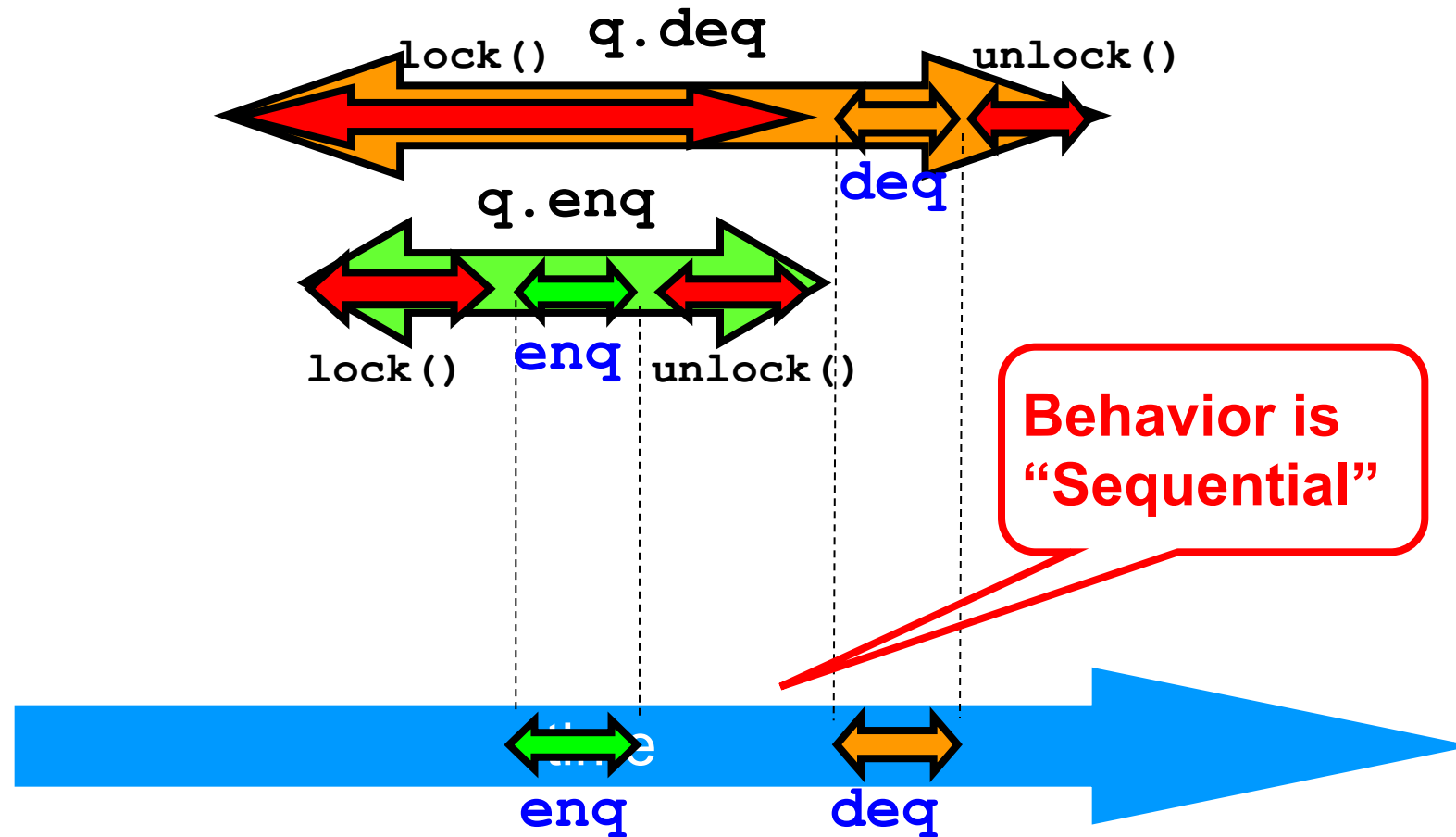
Intuitively...

```
def deg() : 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



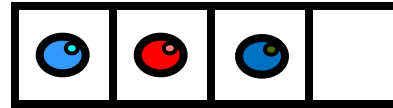
Linearizability

- 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**TM

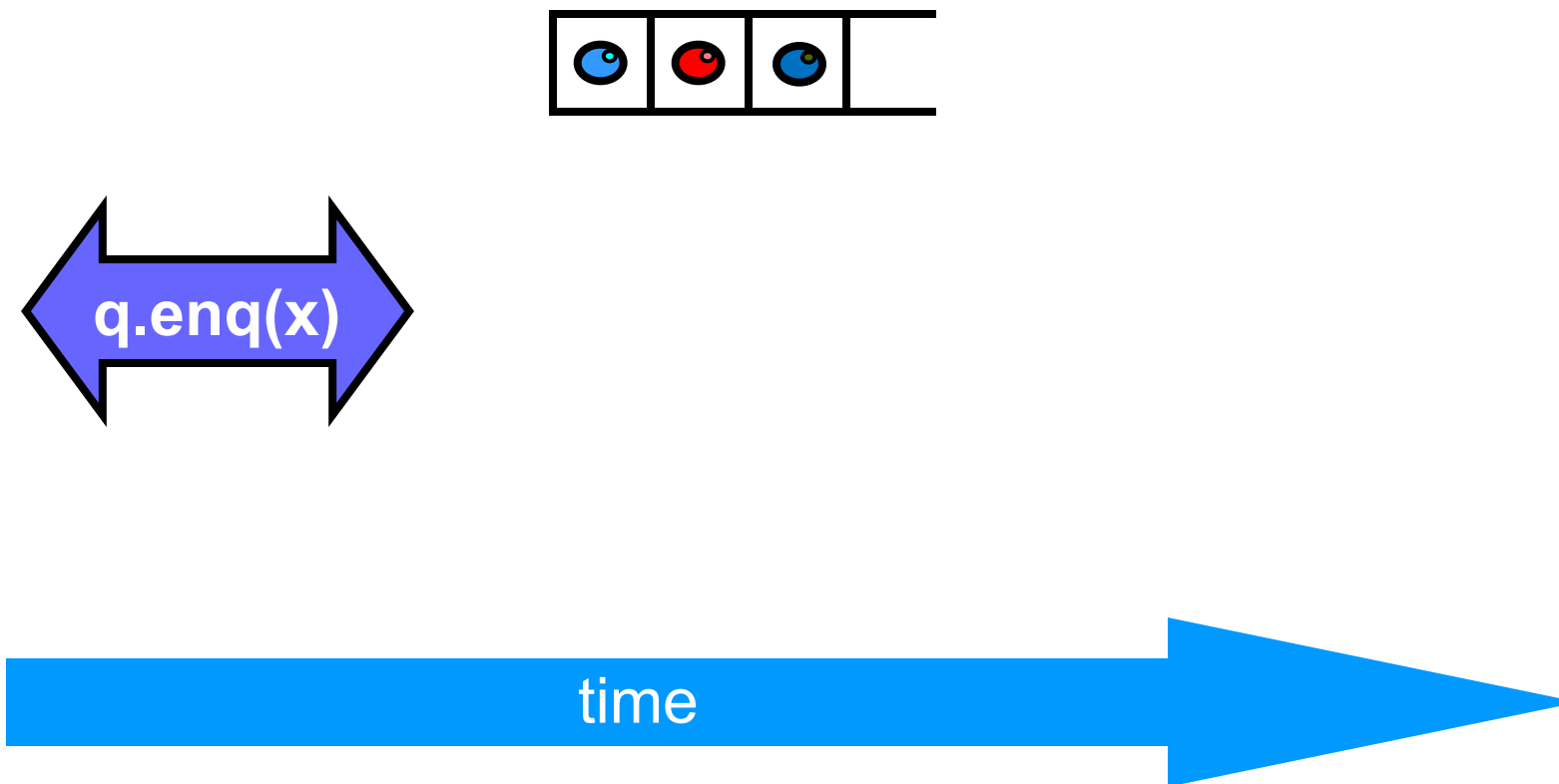
Is it really about the object?

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

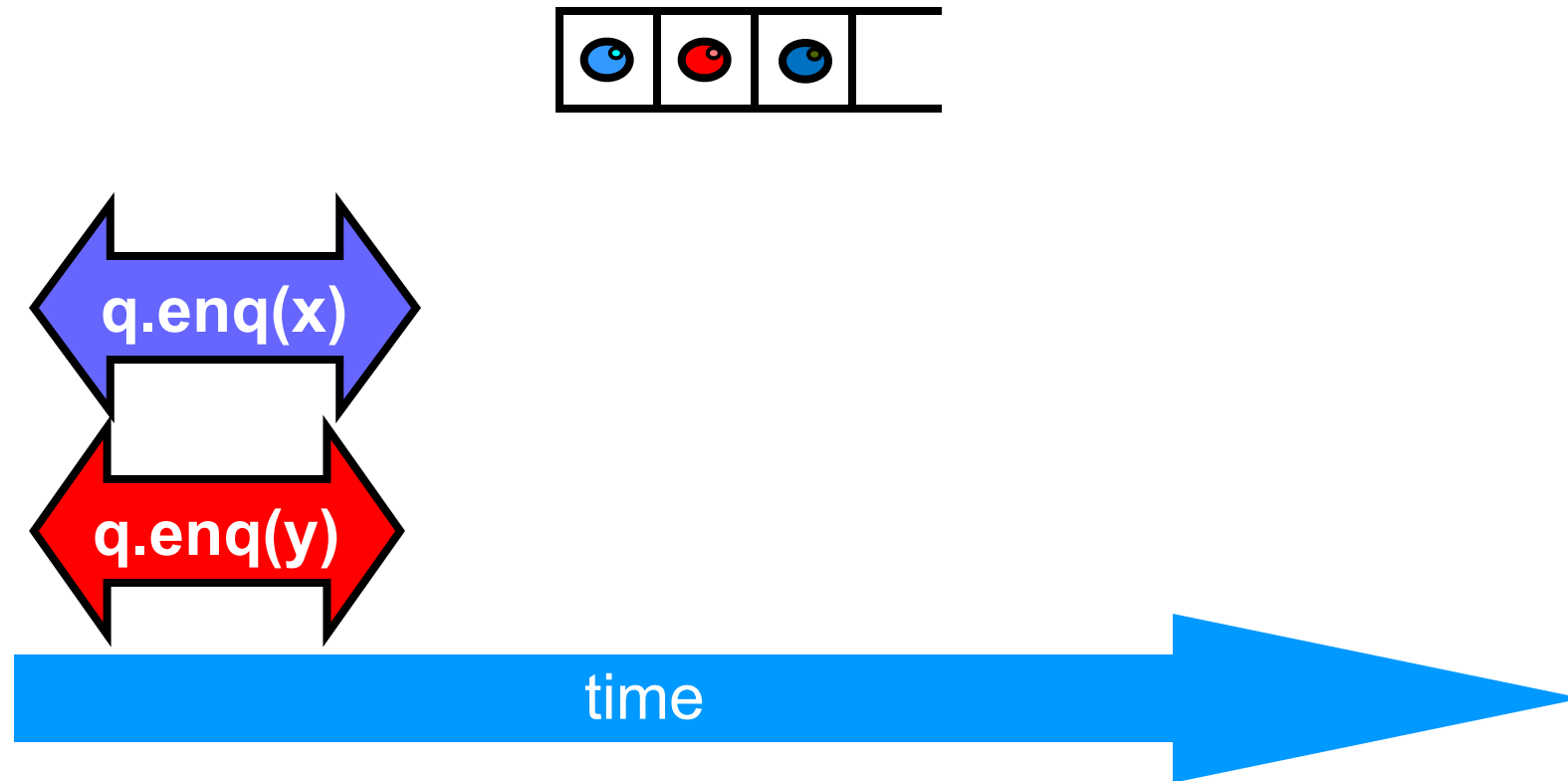
Example



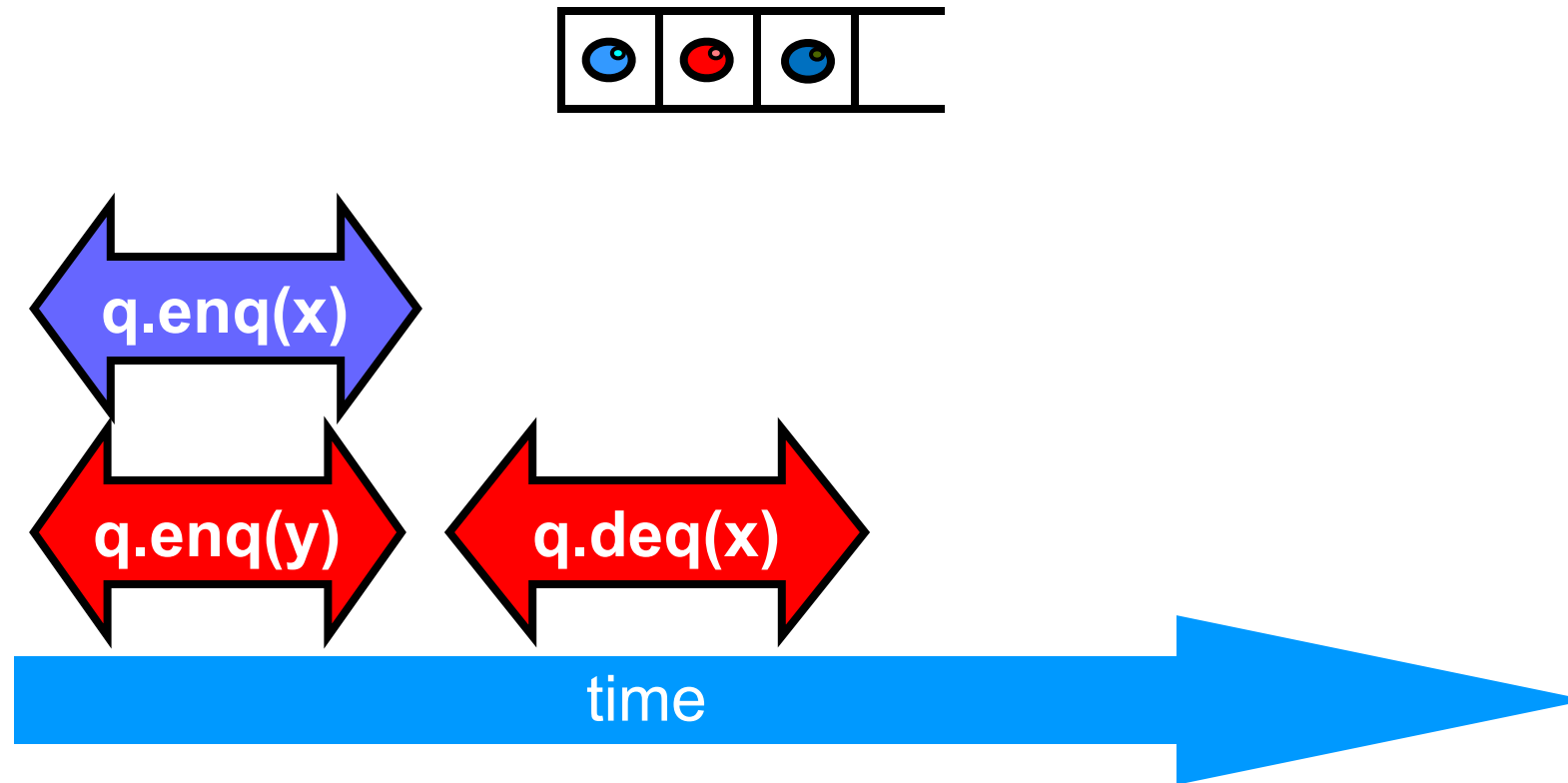
Example



Example

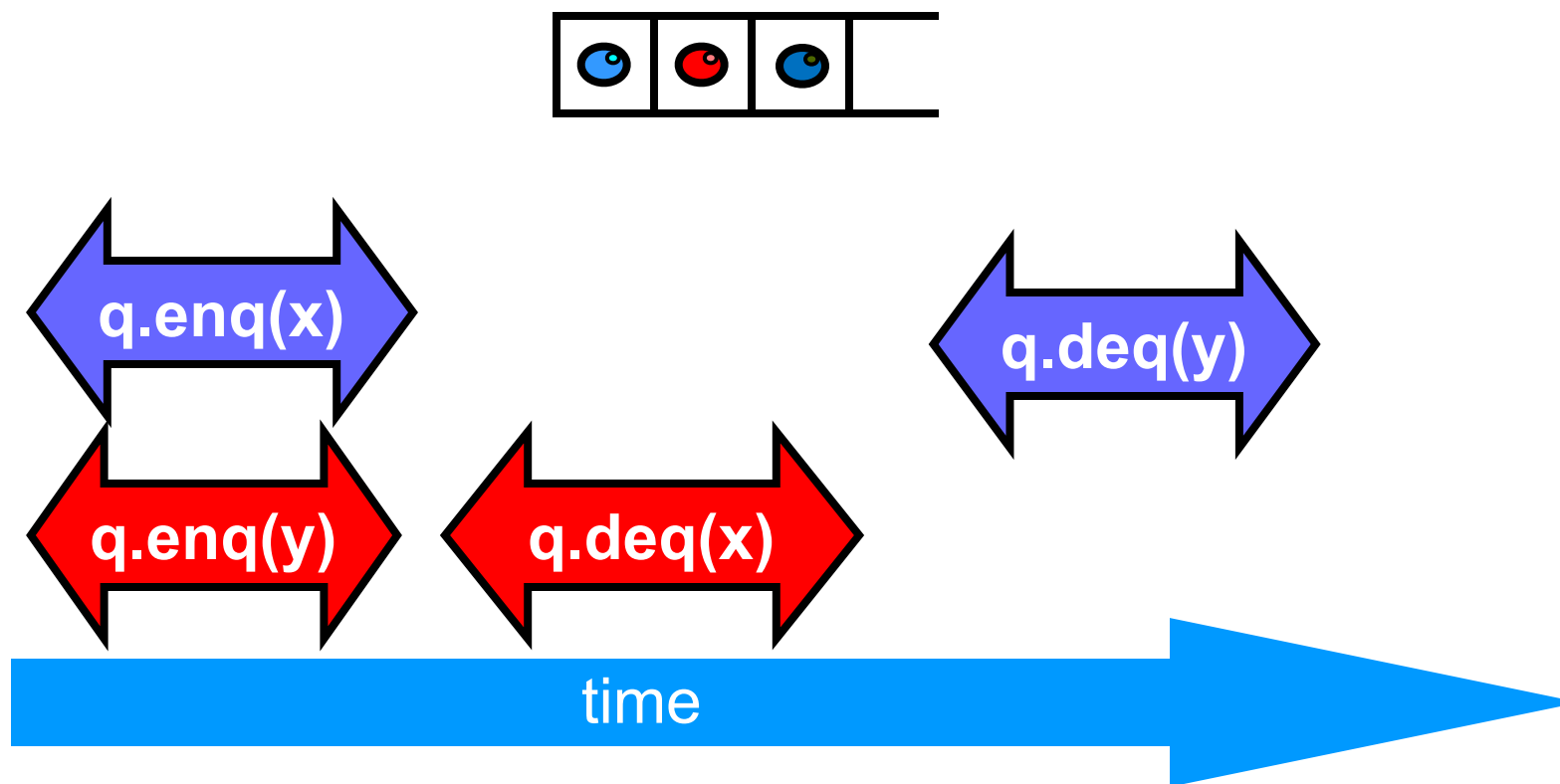


Example

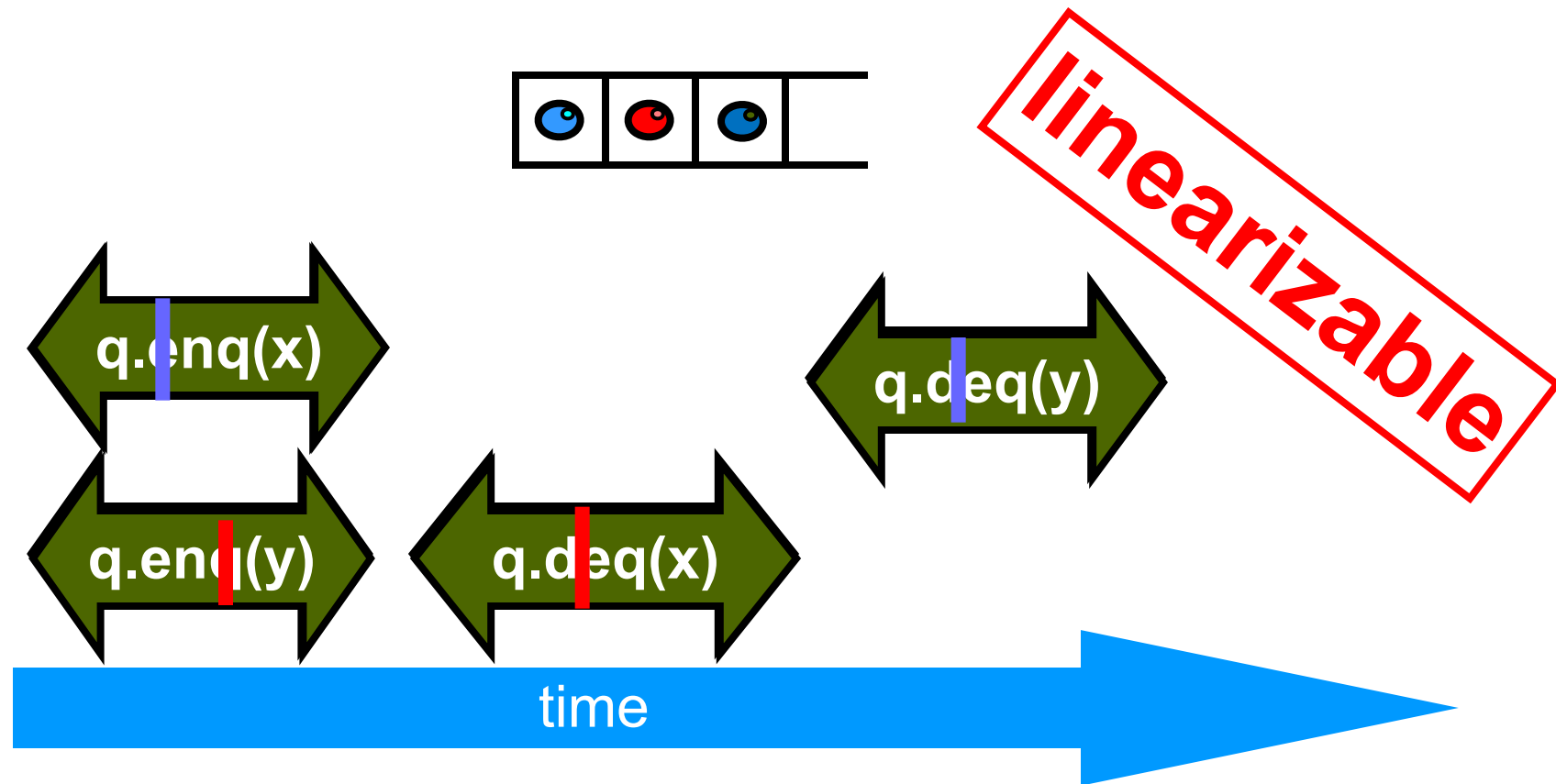




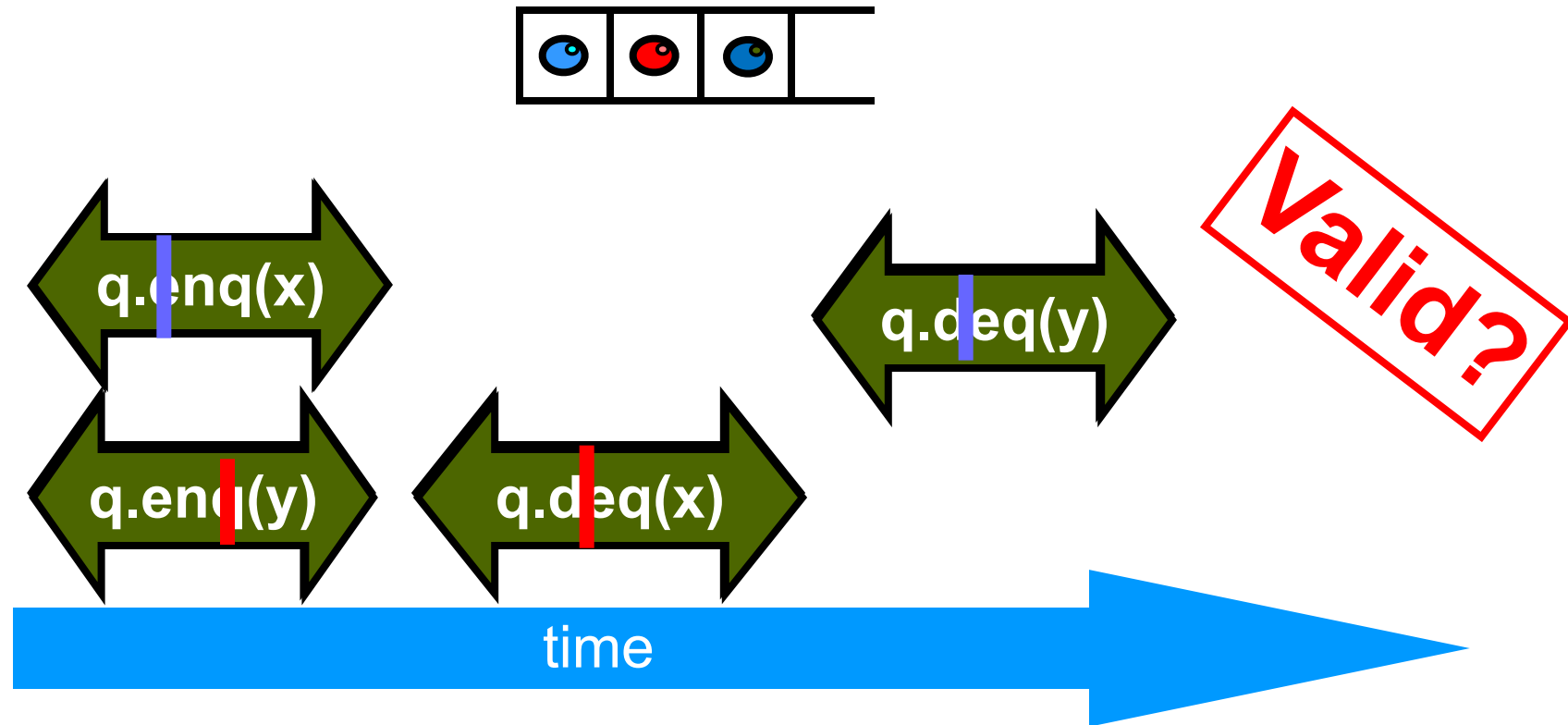
Example



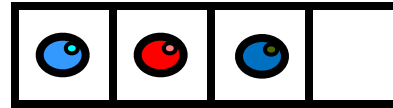
Example



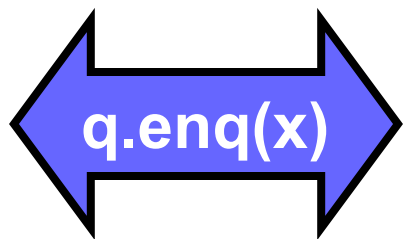
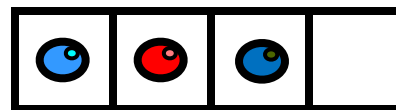
Example



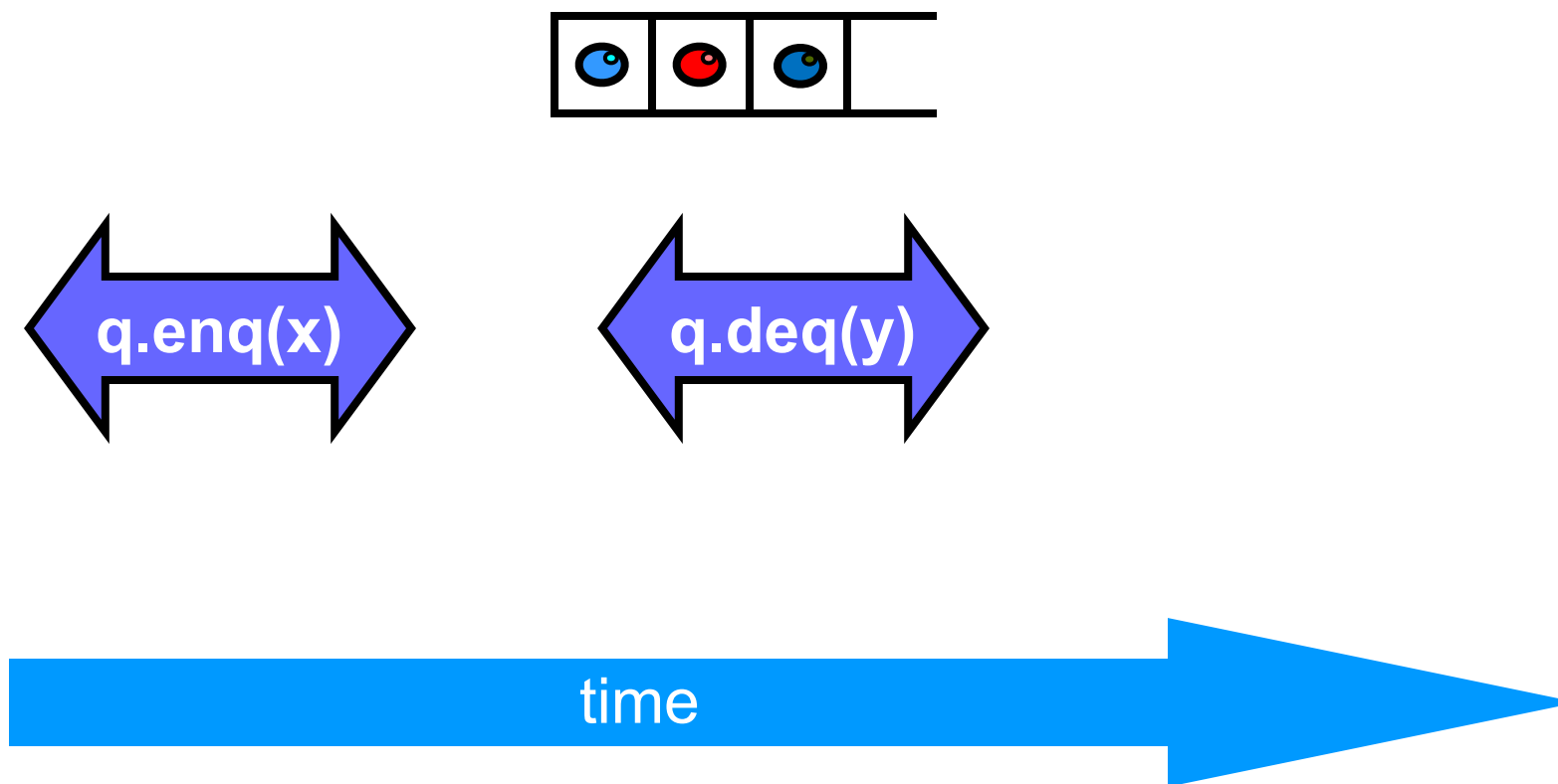
Example



Example

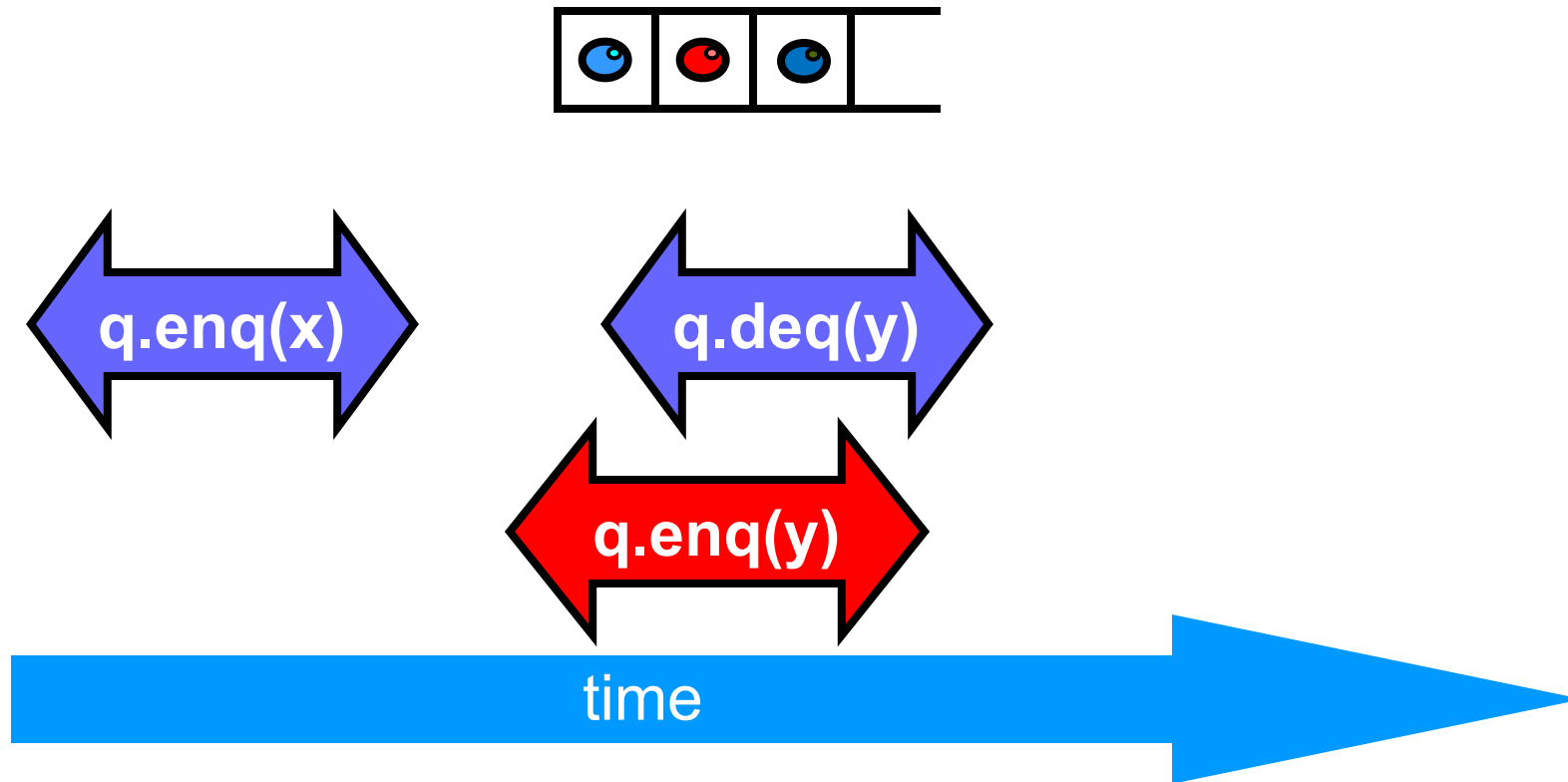


Example



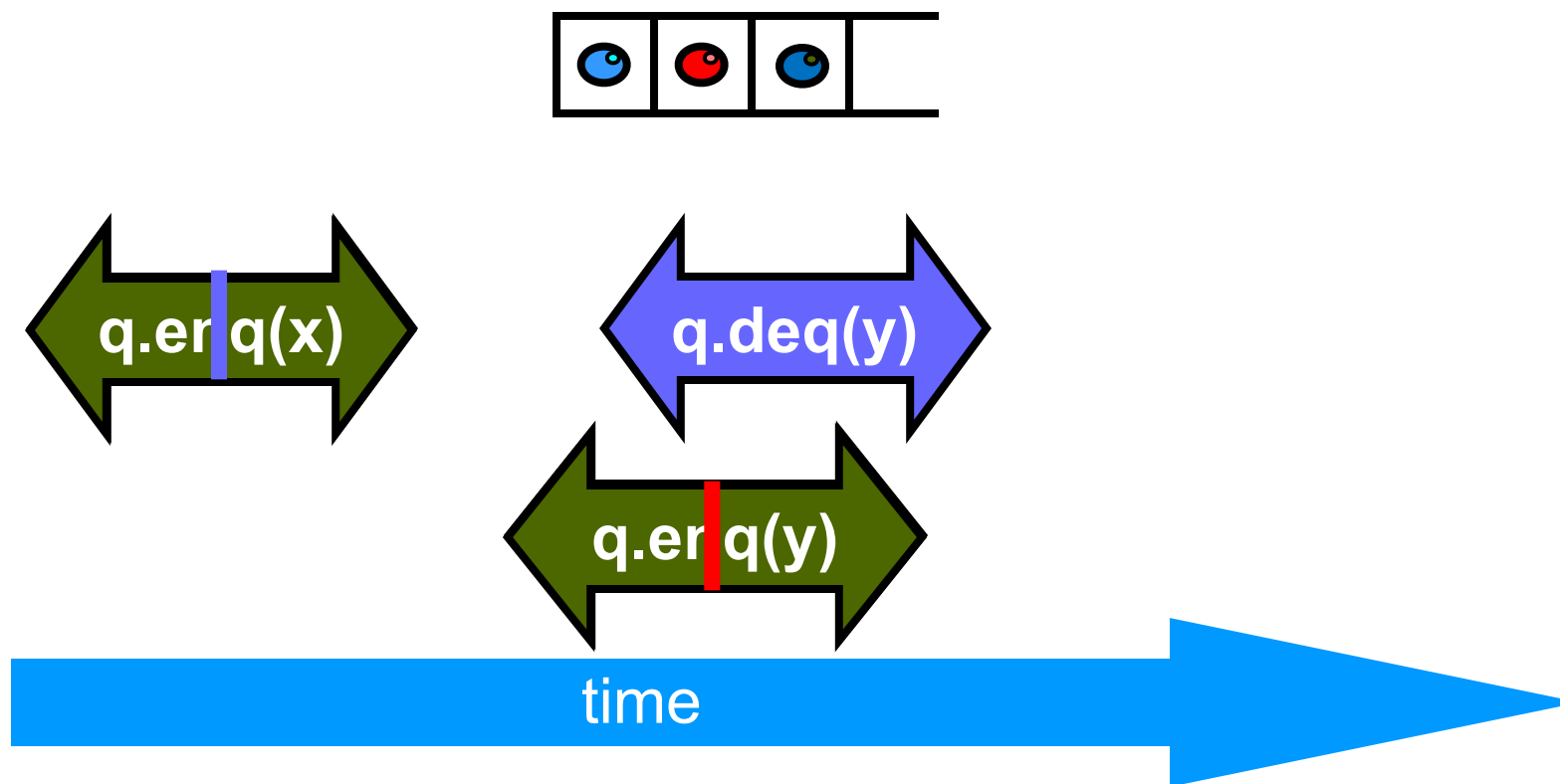


Example



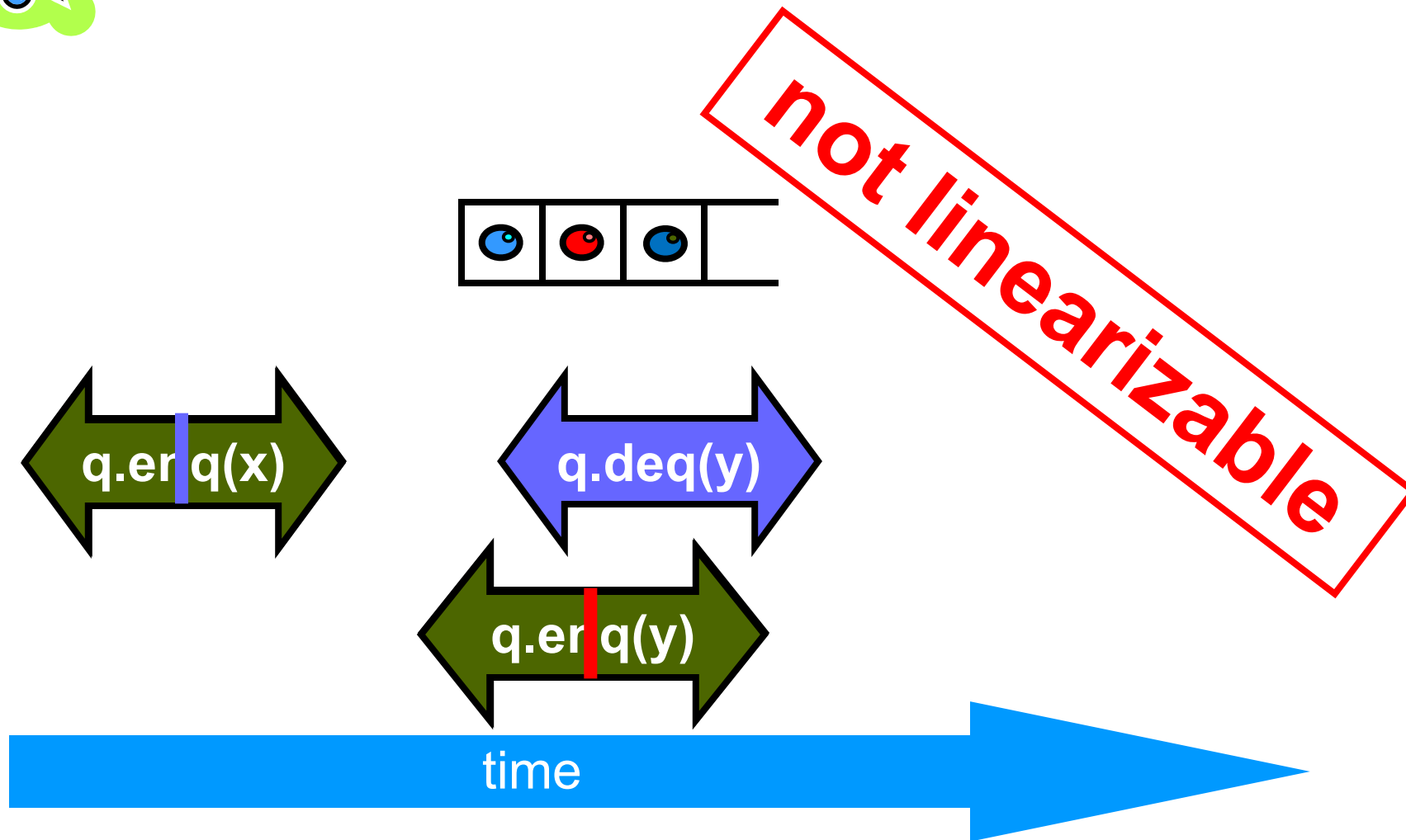


Example

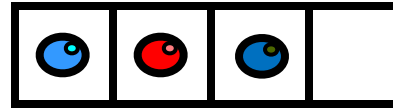




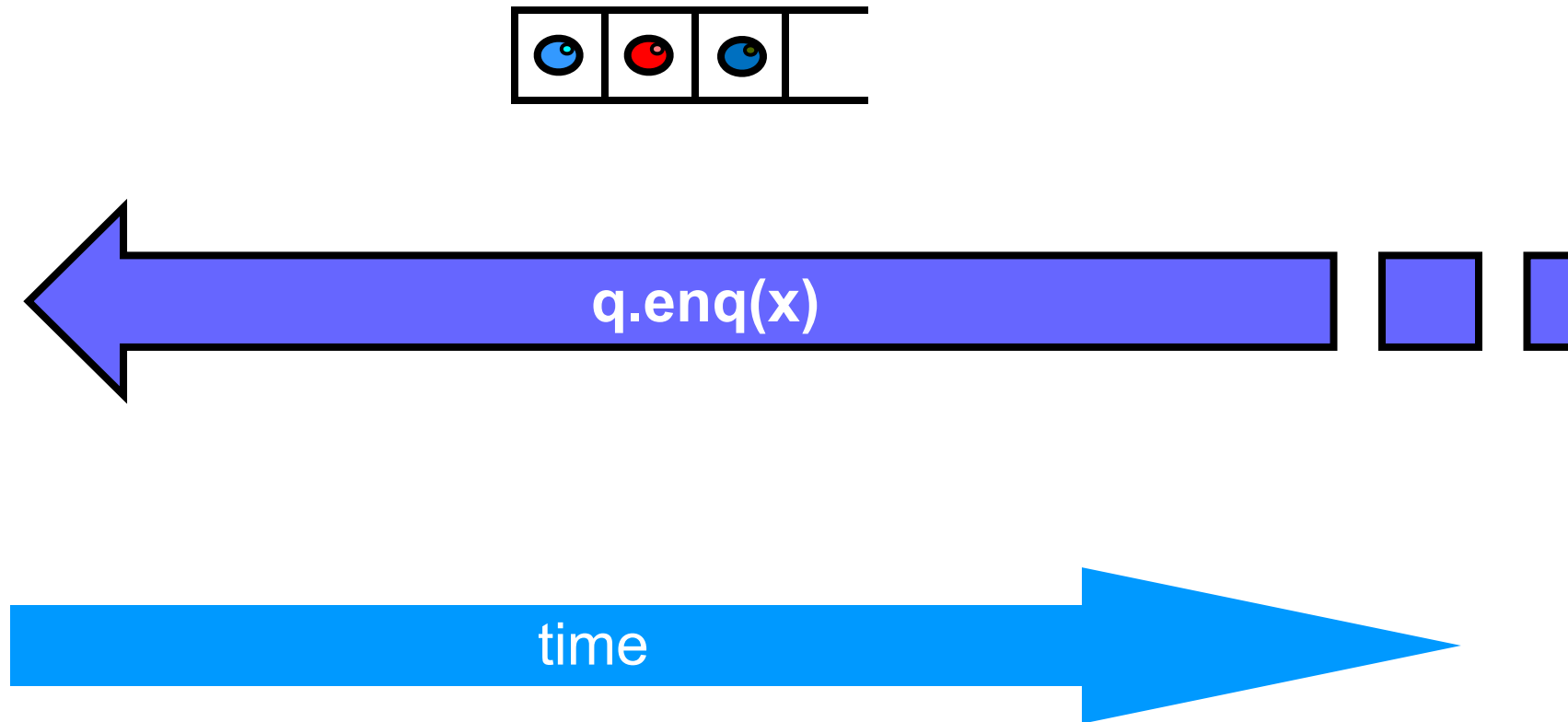
Example



Example

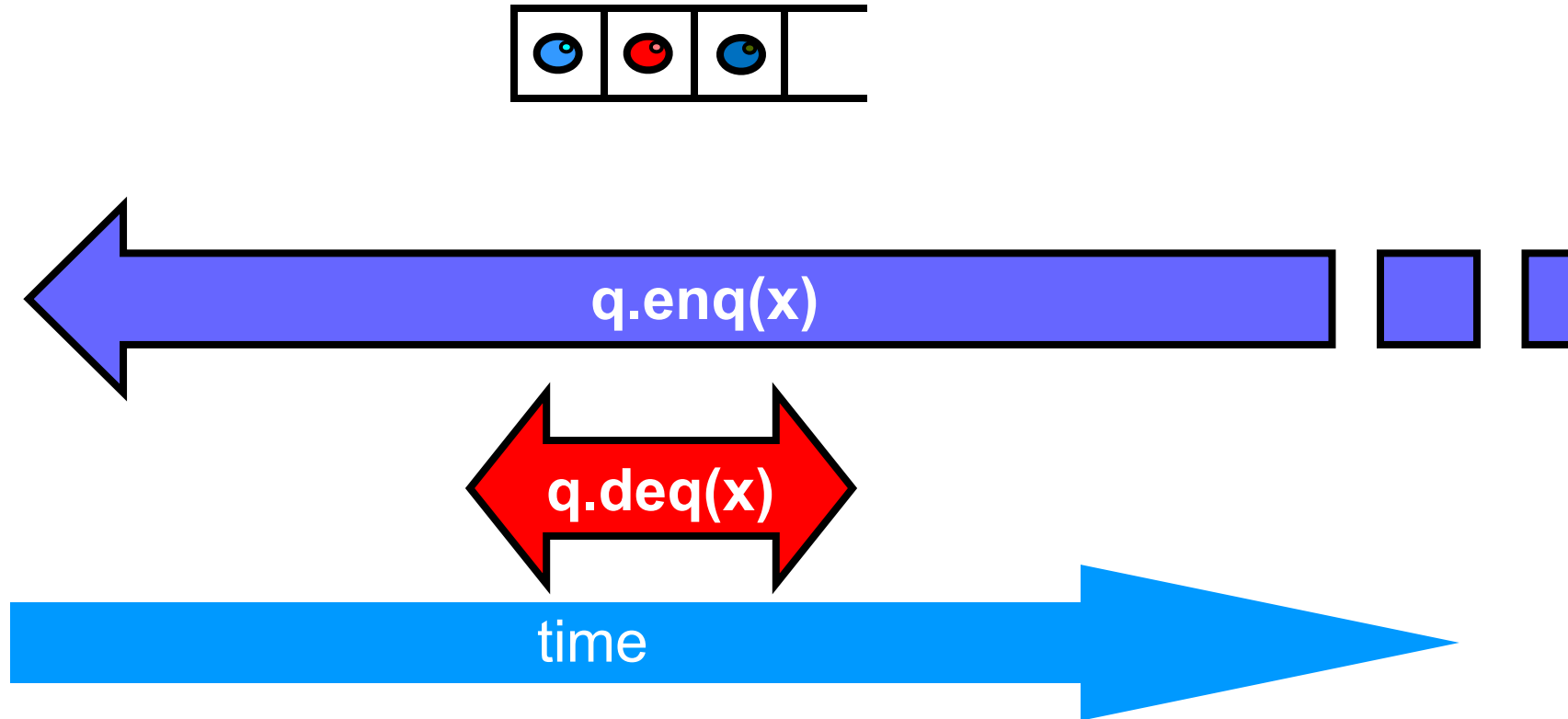


Example



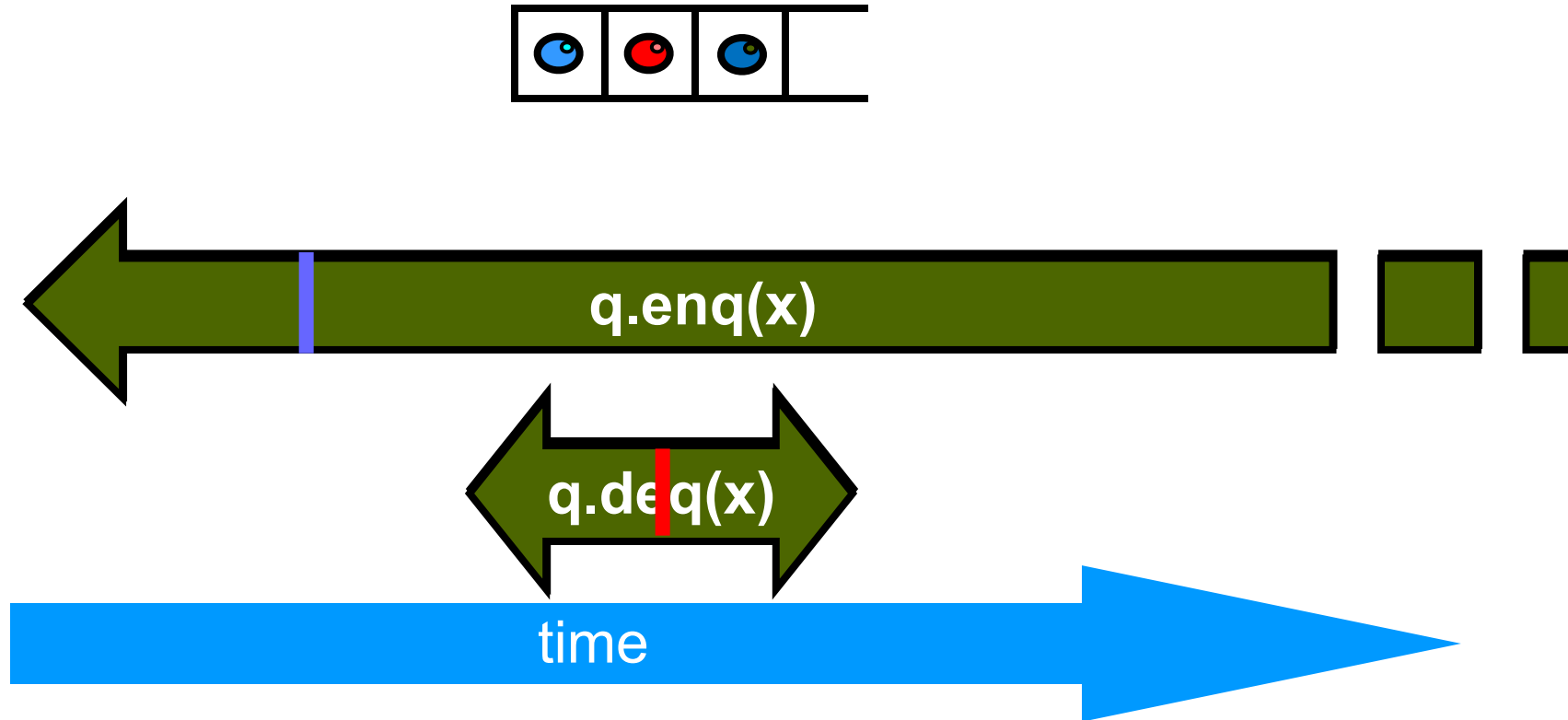


Example



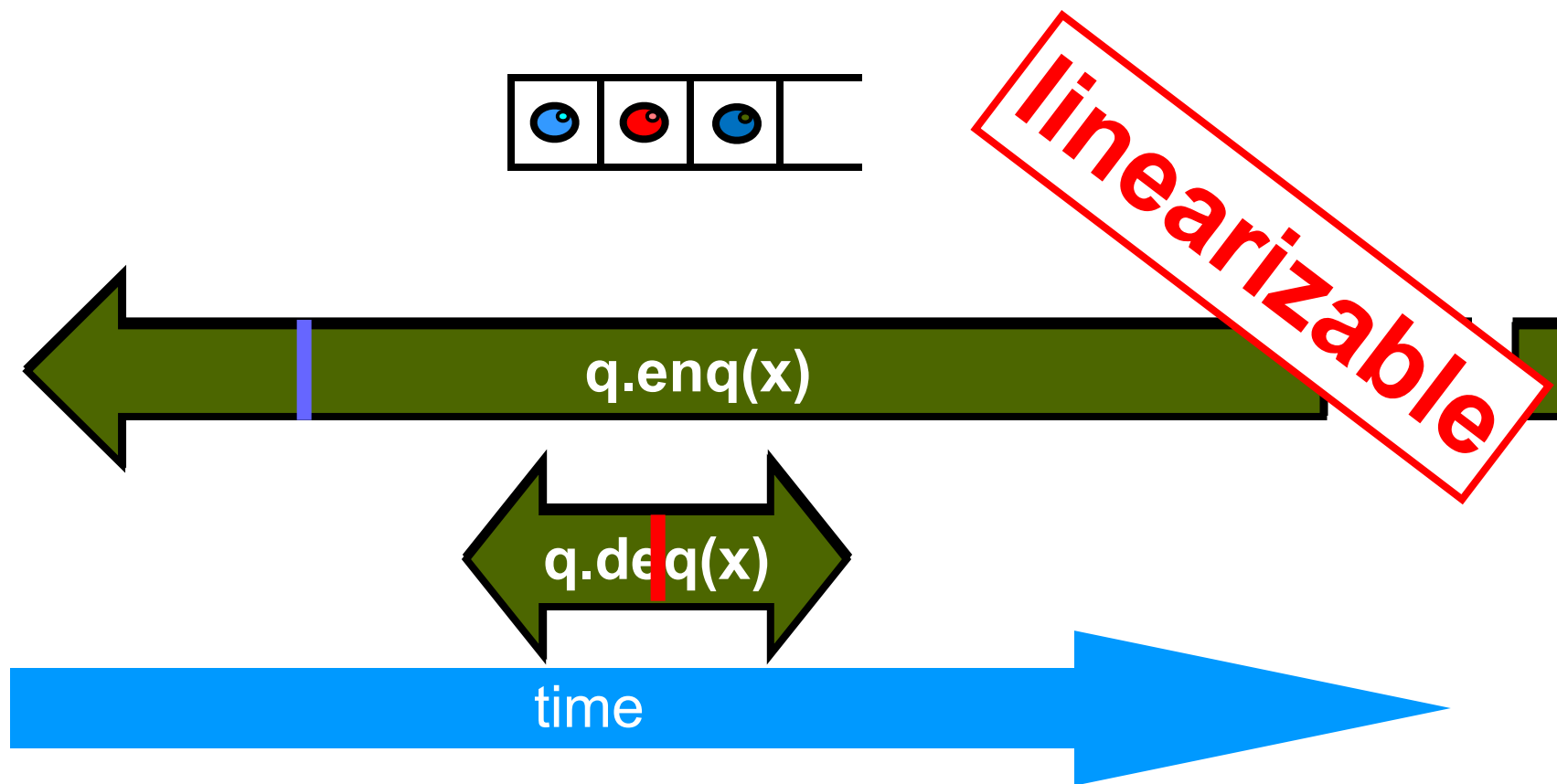


Example

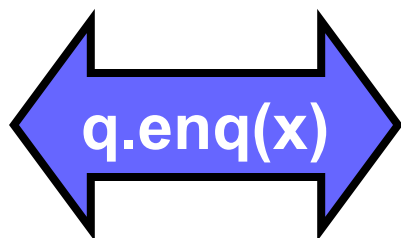
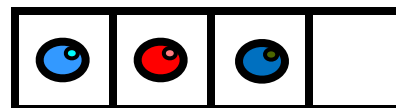




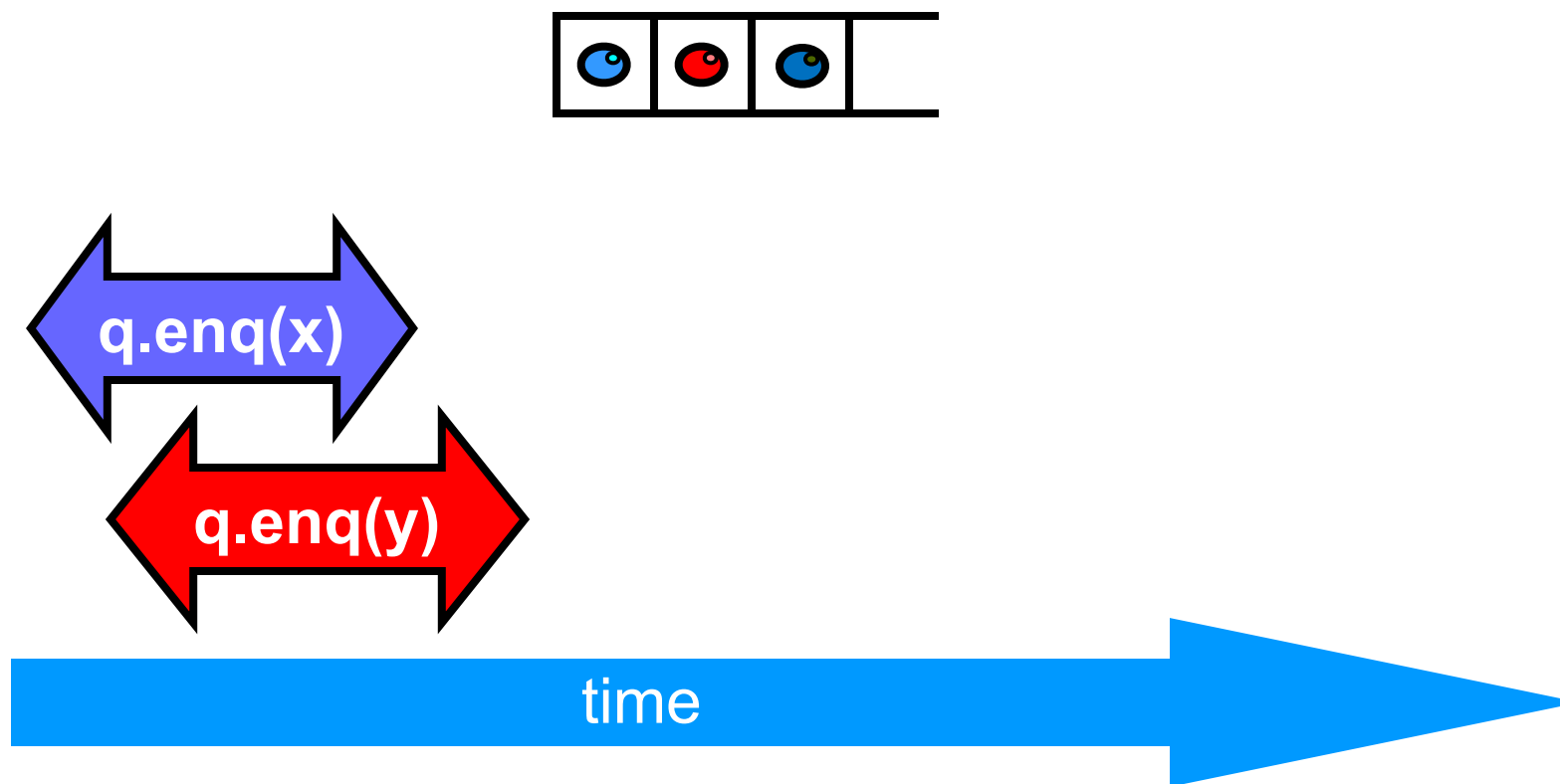
Example



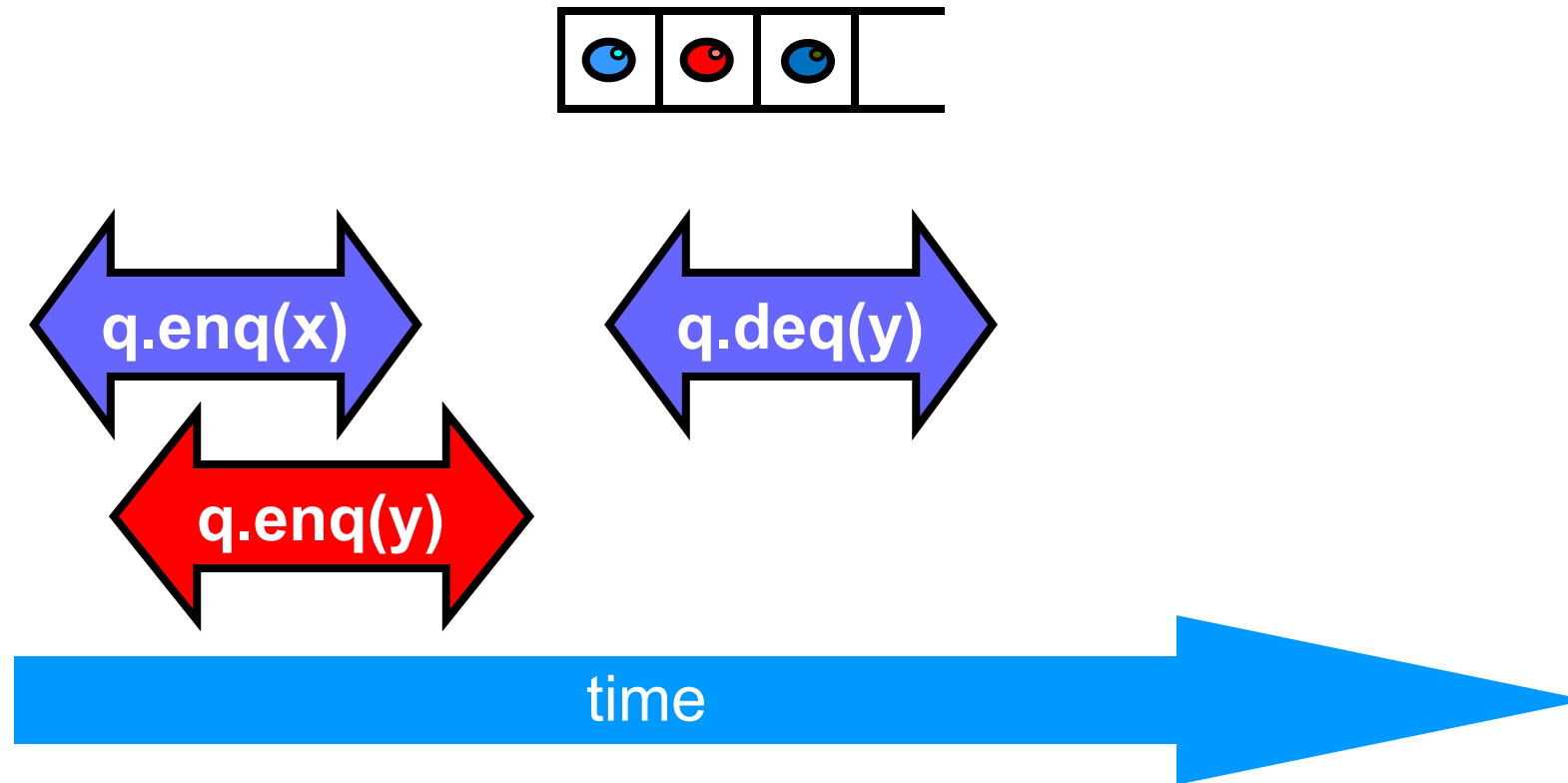
Example



Example

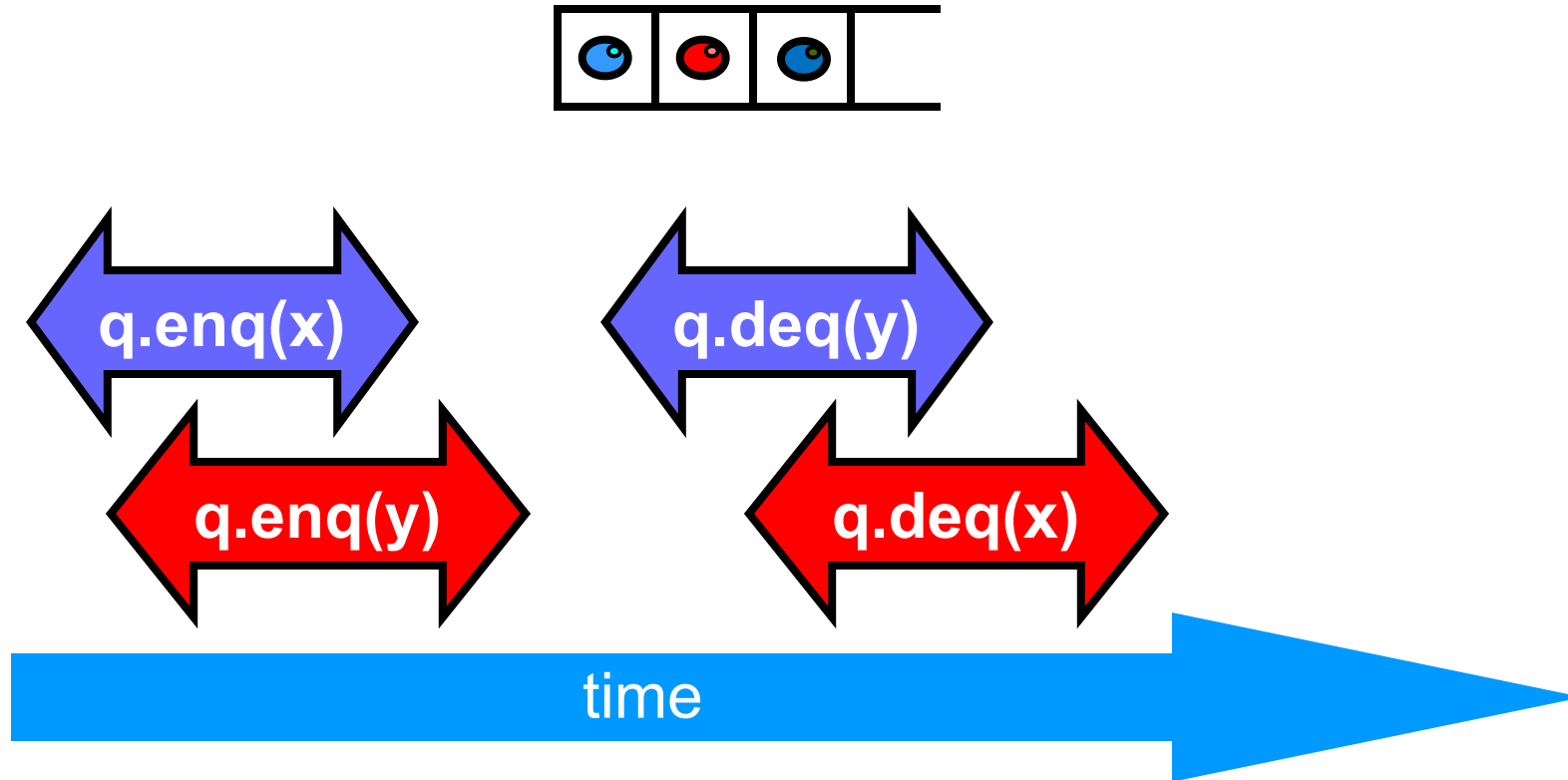


Example



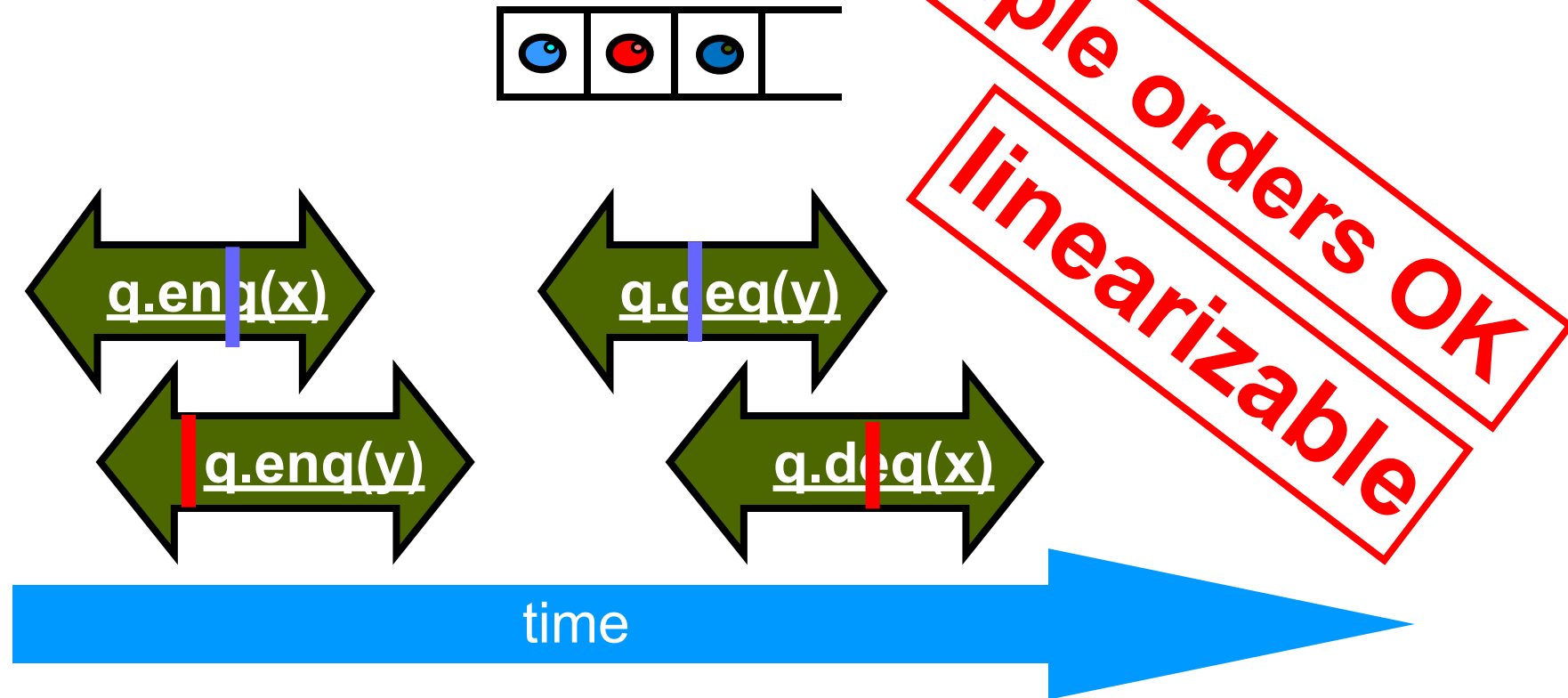


Example

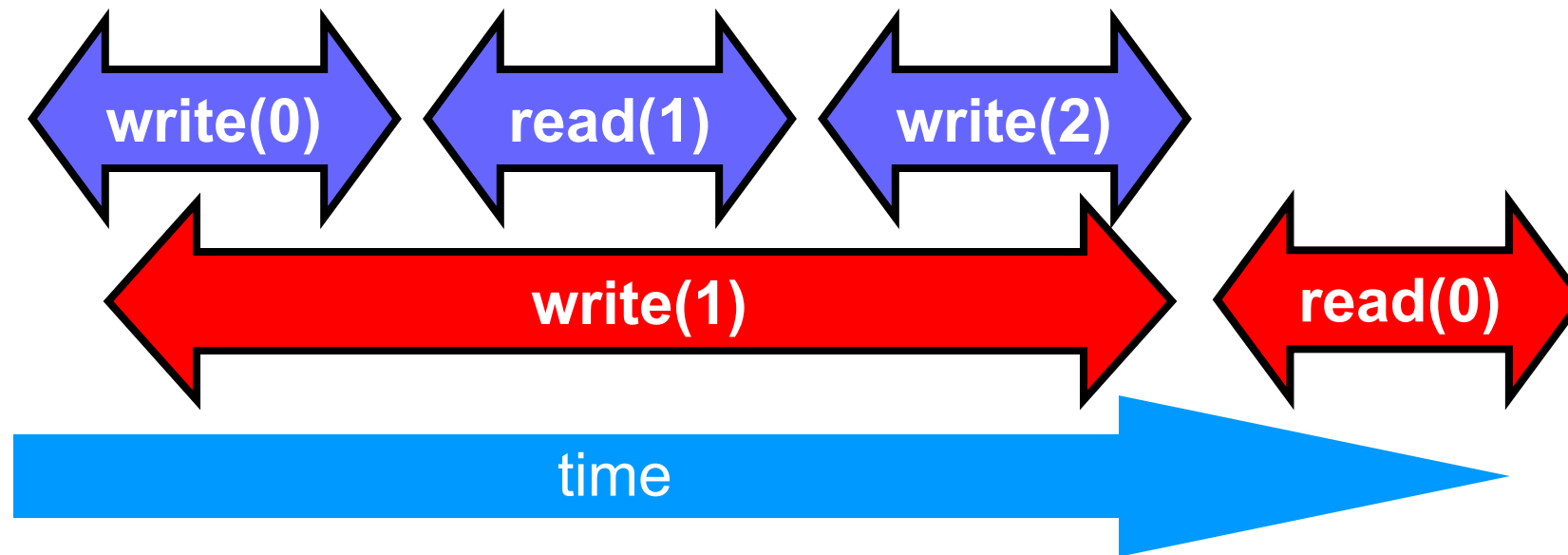


Comme ci
Comme ça

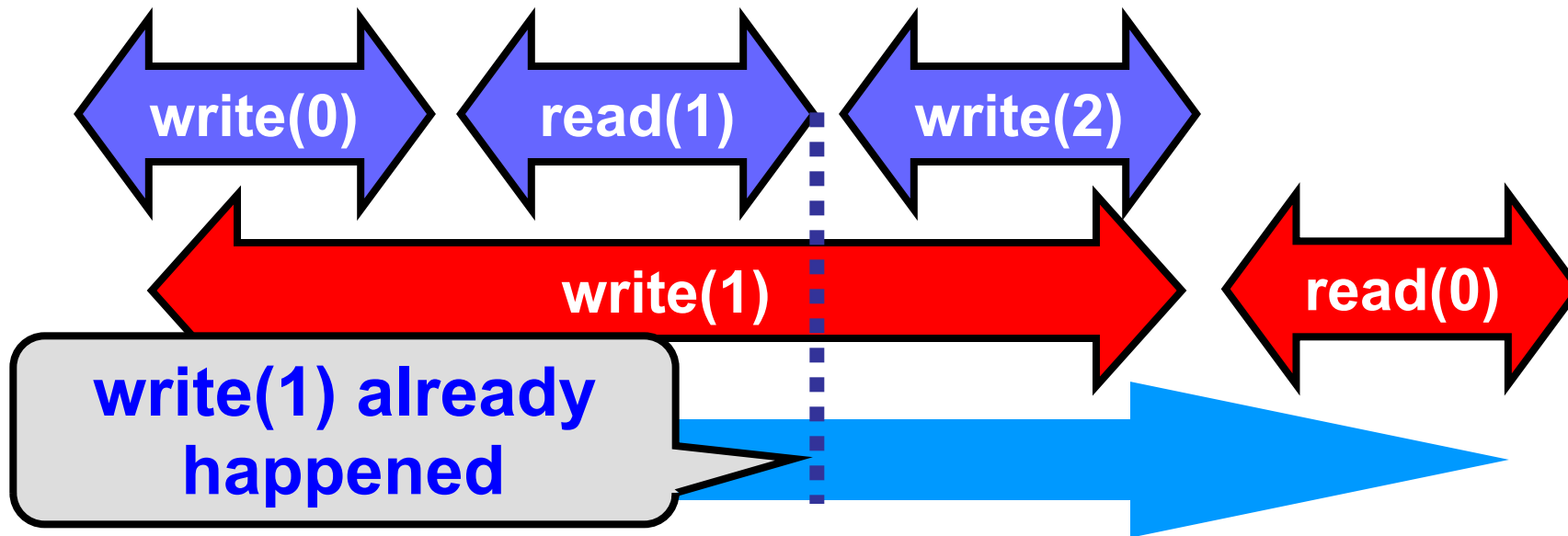
Example



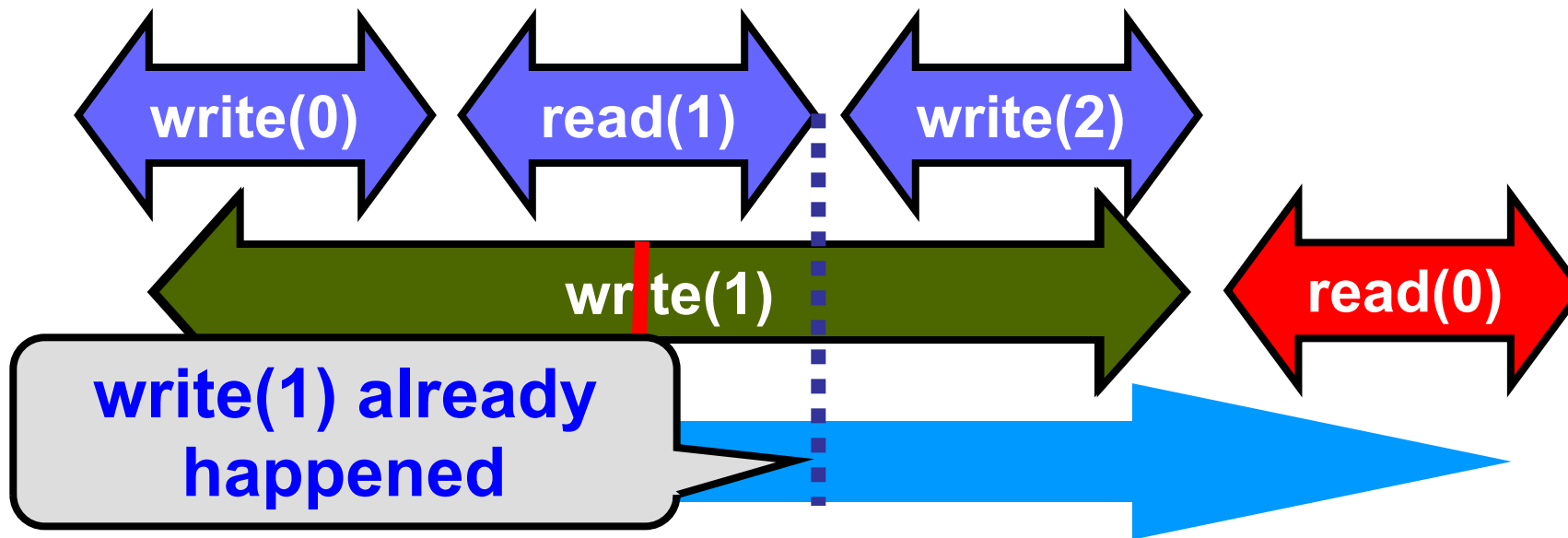
Read/Write Register Example



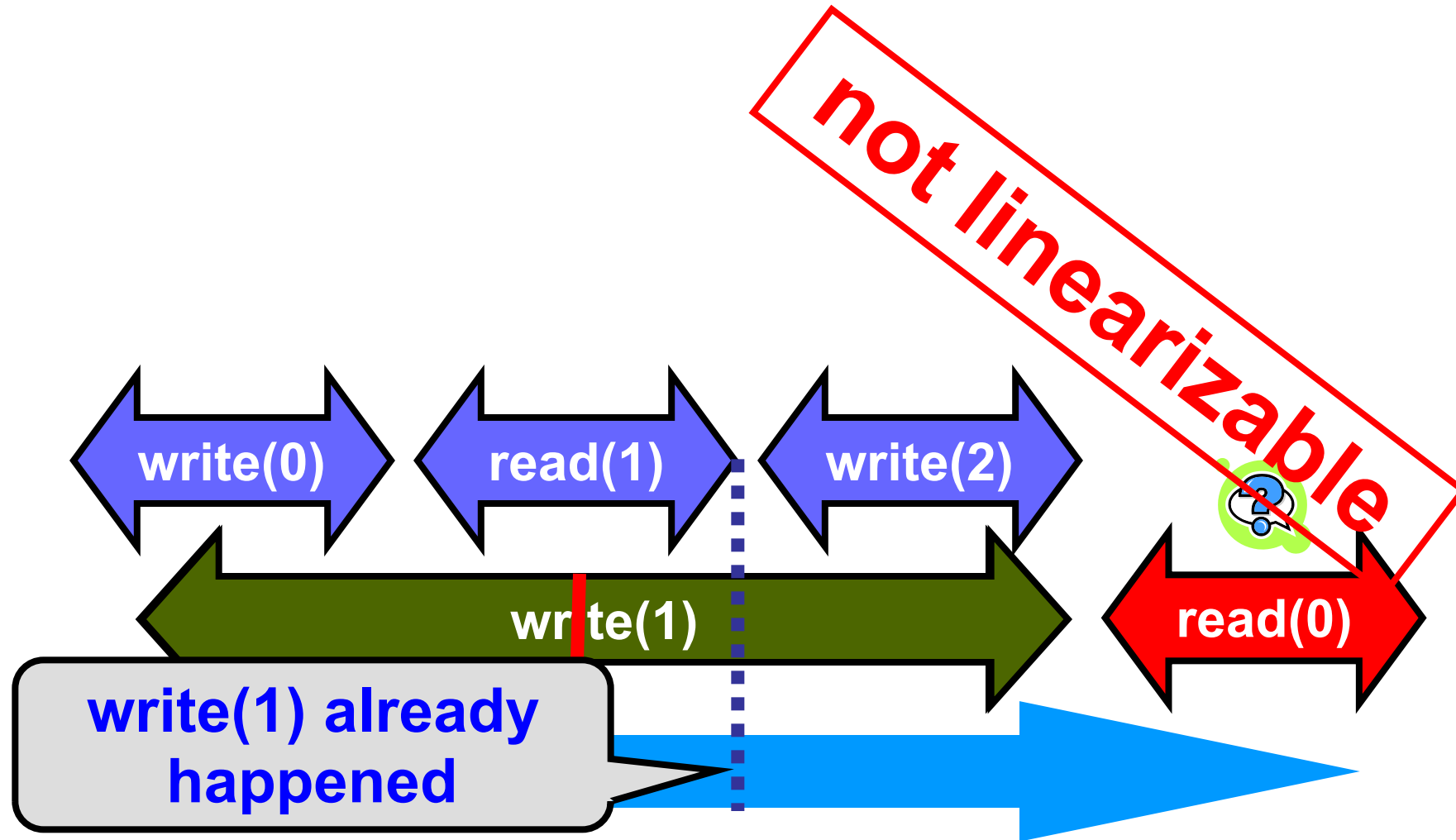
Read/Write Register Example



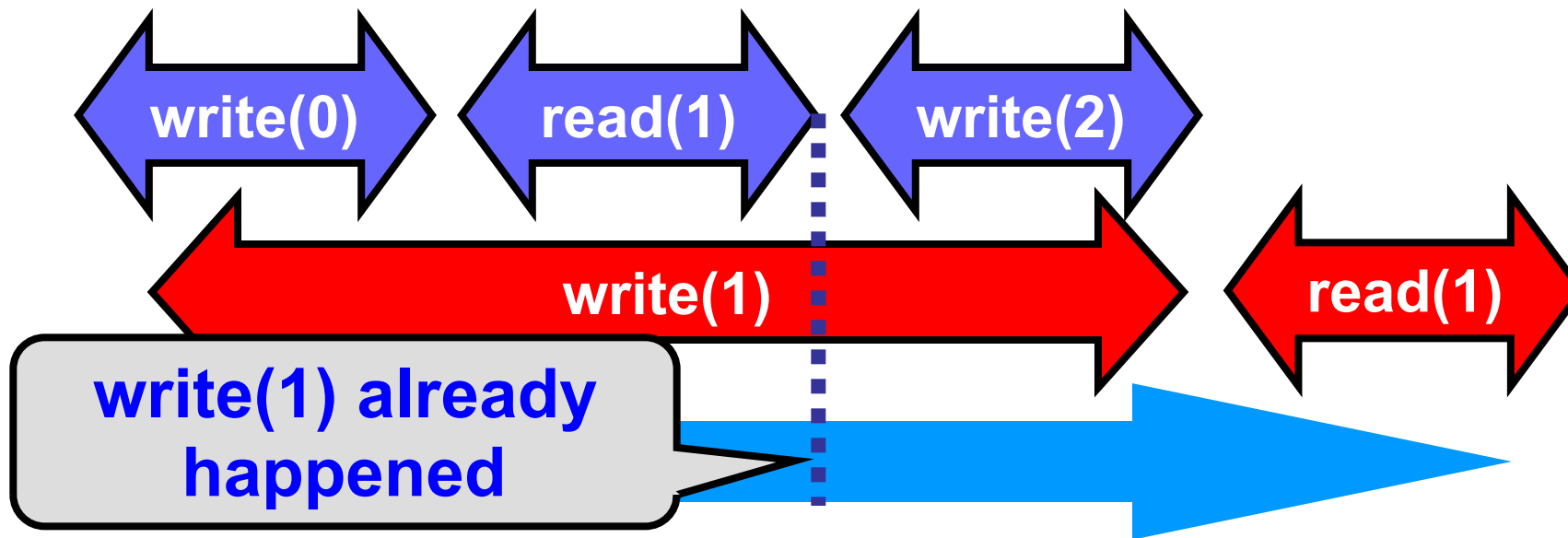
Read/Write Register Example



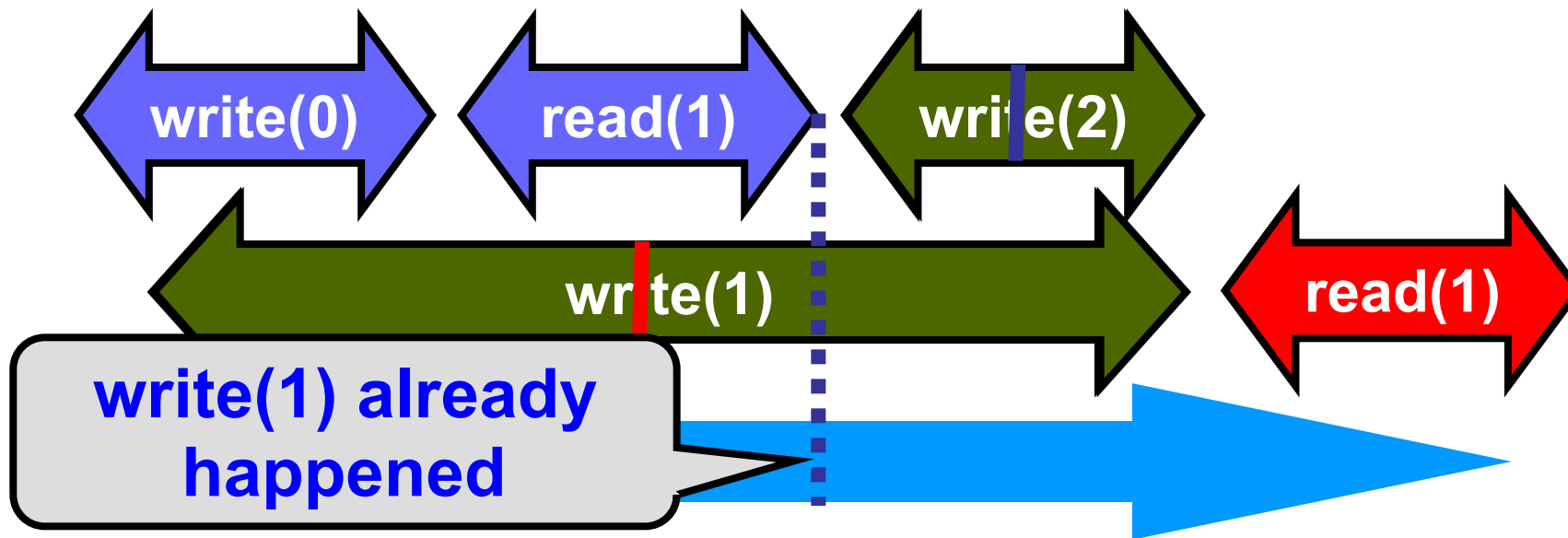
Read/Write Register Example



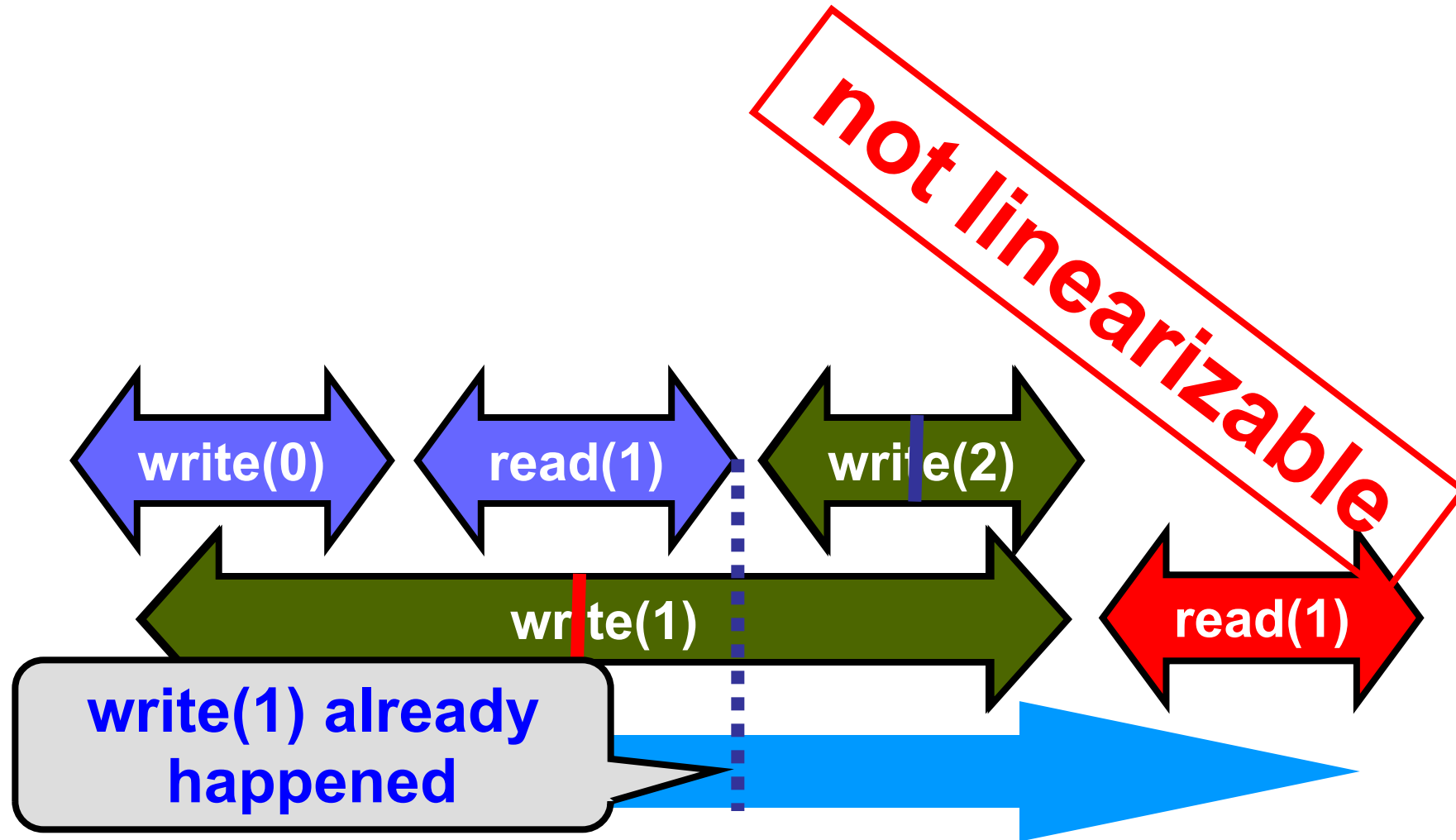
Read/Write Register Example



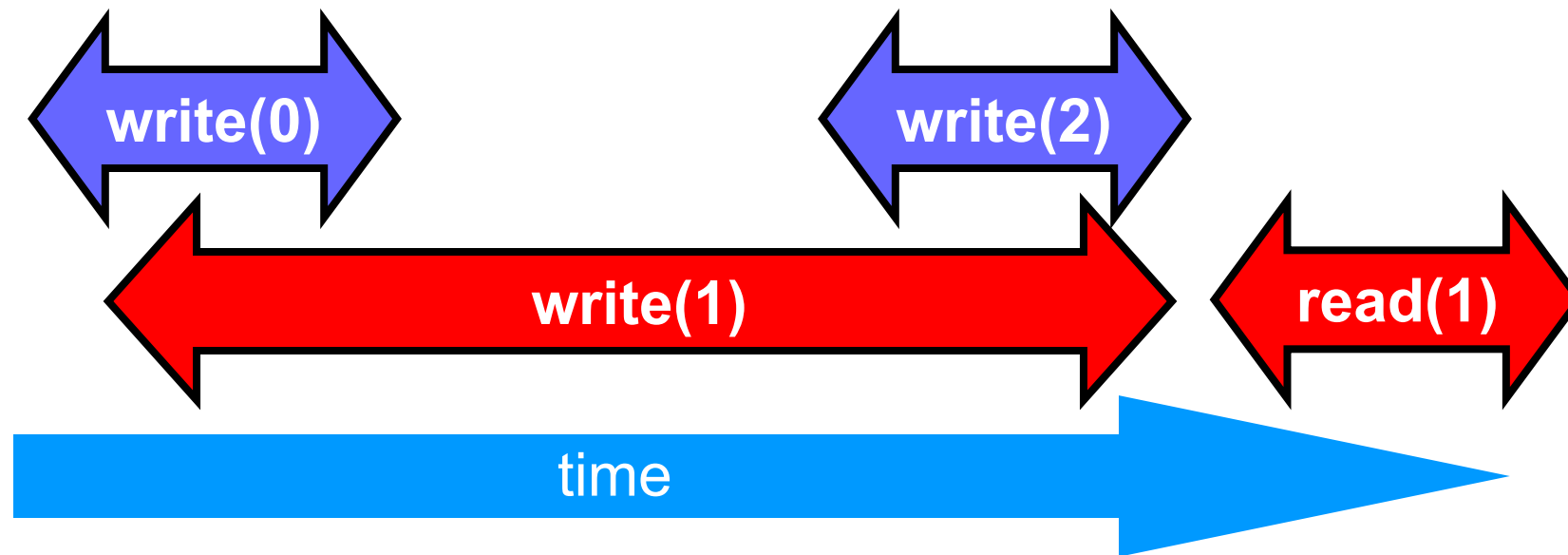
Read/Write Register Example



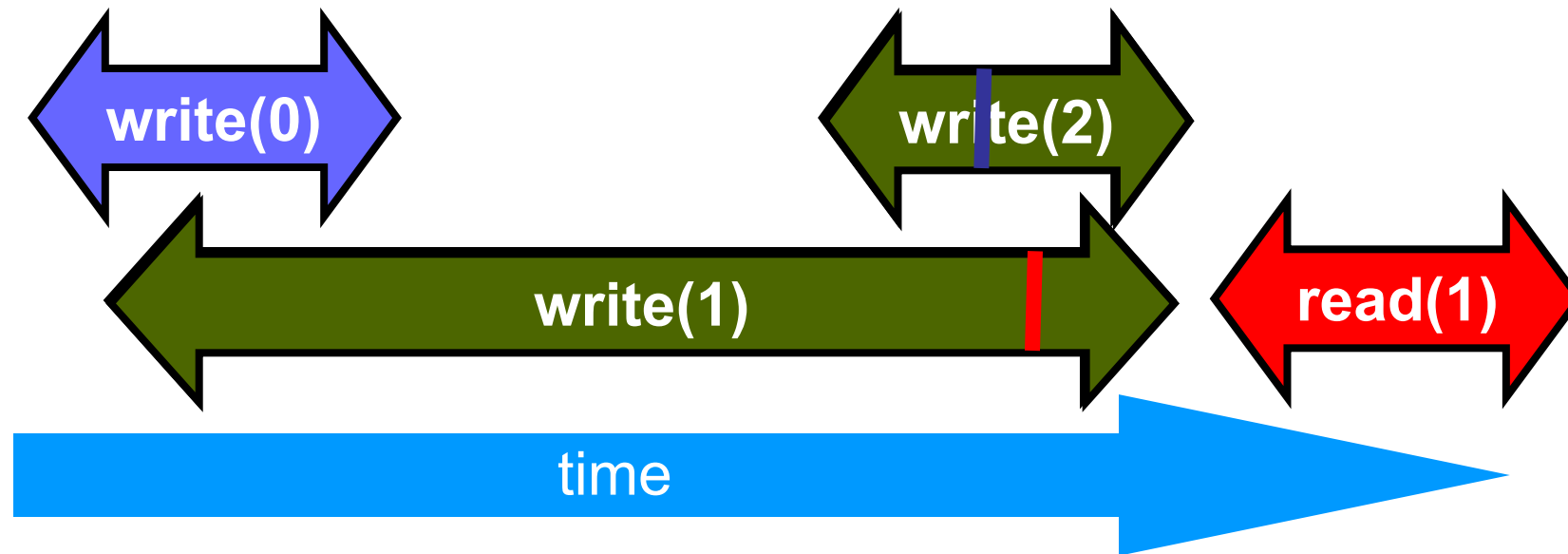
Read/Write Register Example



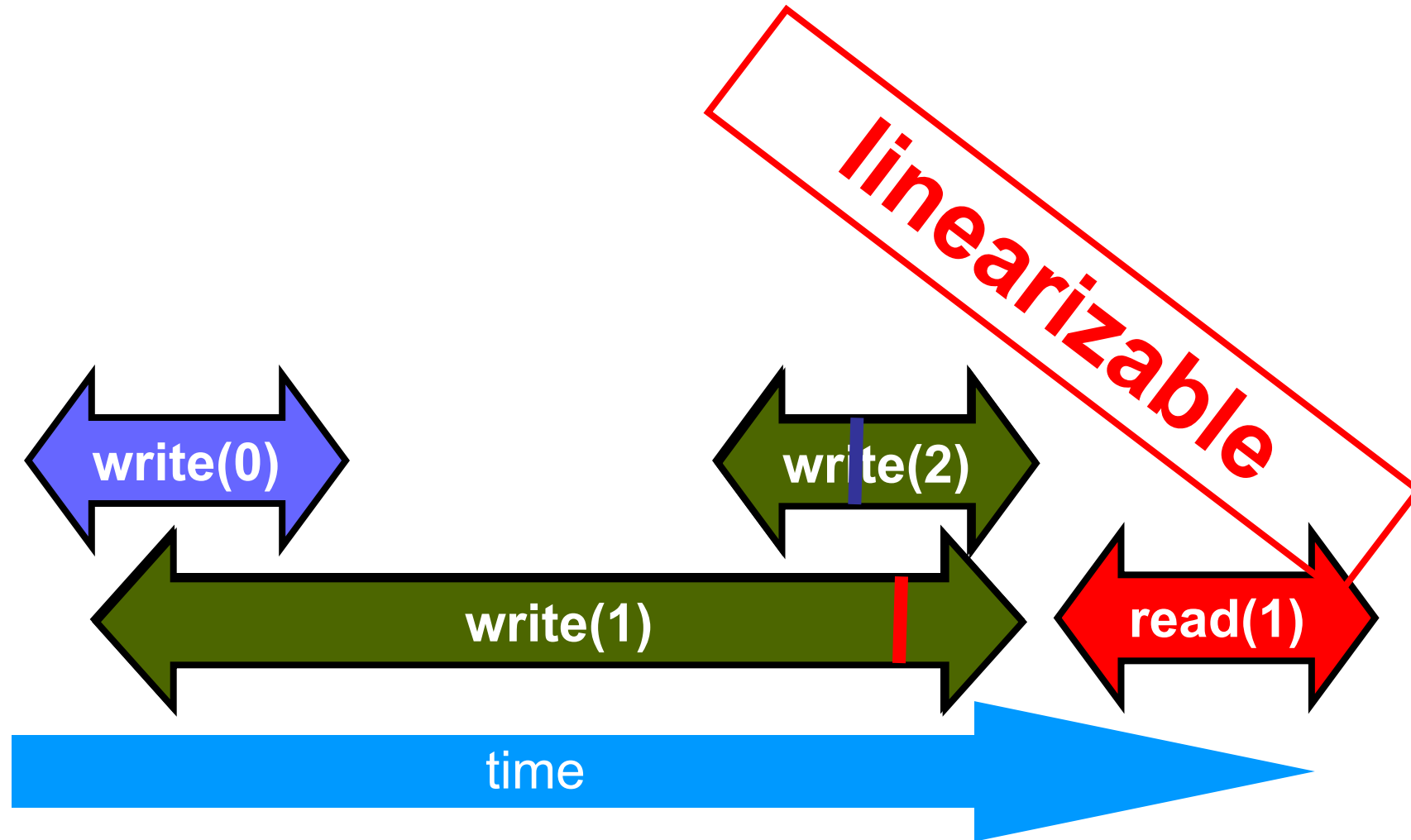
Read/Write Register Example



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



This work is licensed under a [Creative Commons Attribution-ShareAlike 2.5 License](http://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.