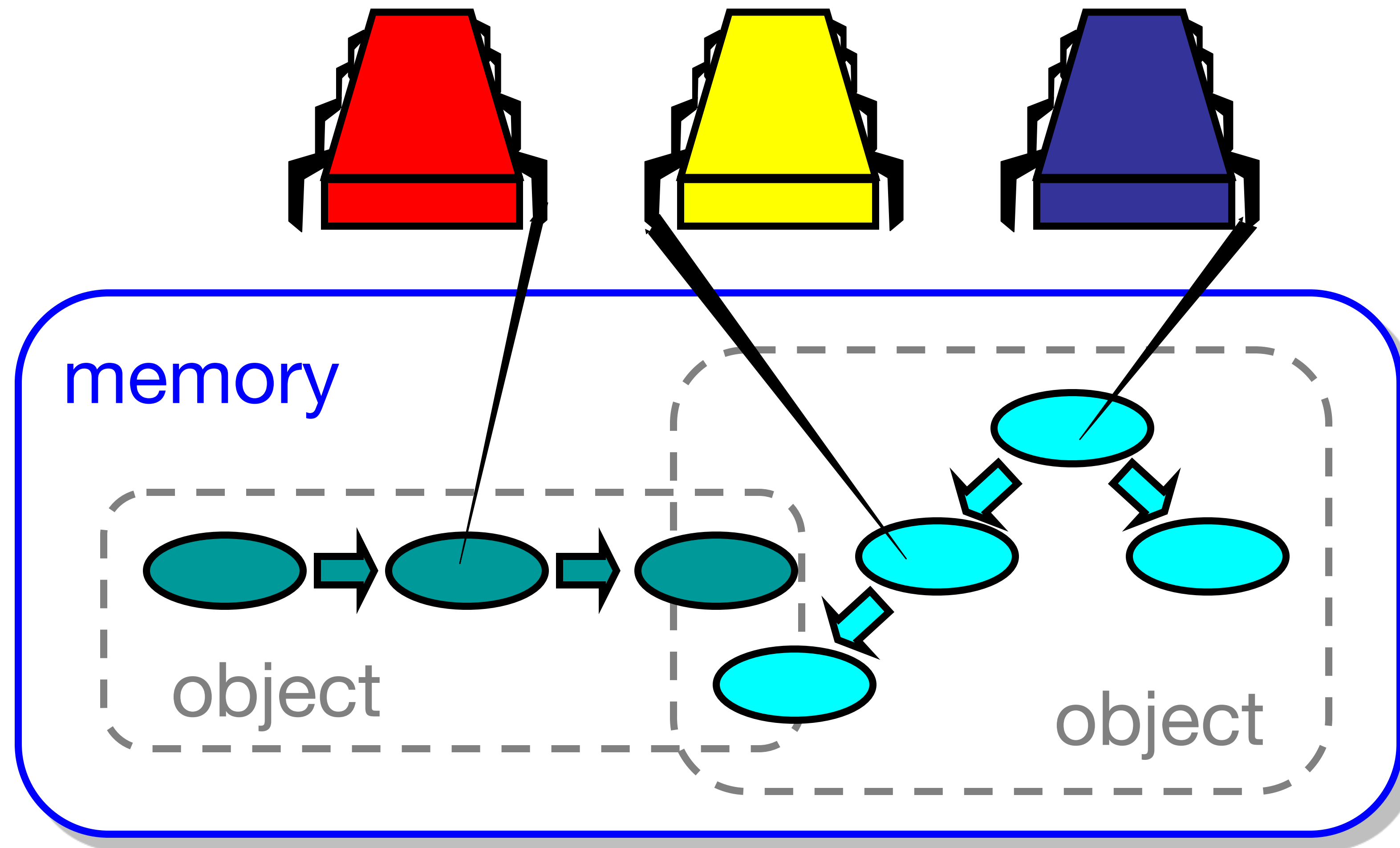# 03 Concurrent Objects

CS 6868: Concurrent Programming

KC Sivaramakrishnan

Spring 2026, IIT Madras

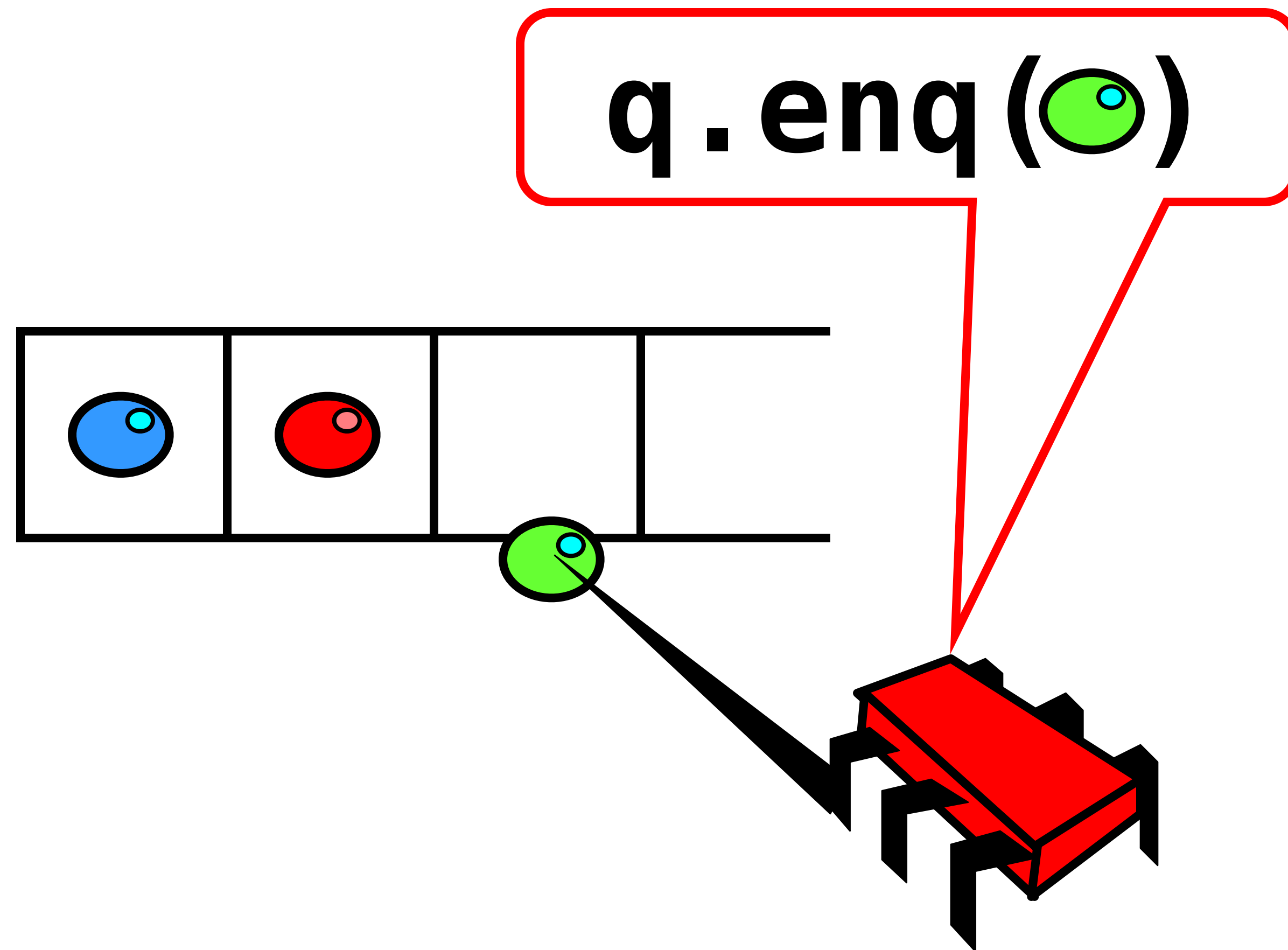# Concurrent Computation

Objectivism

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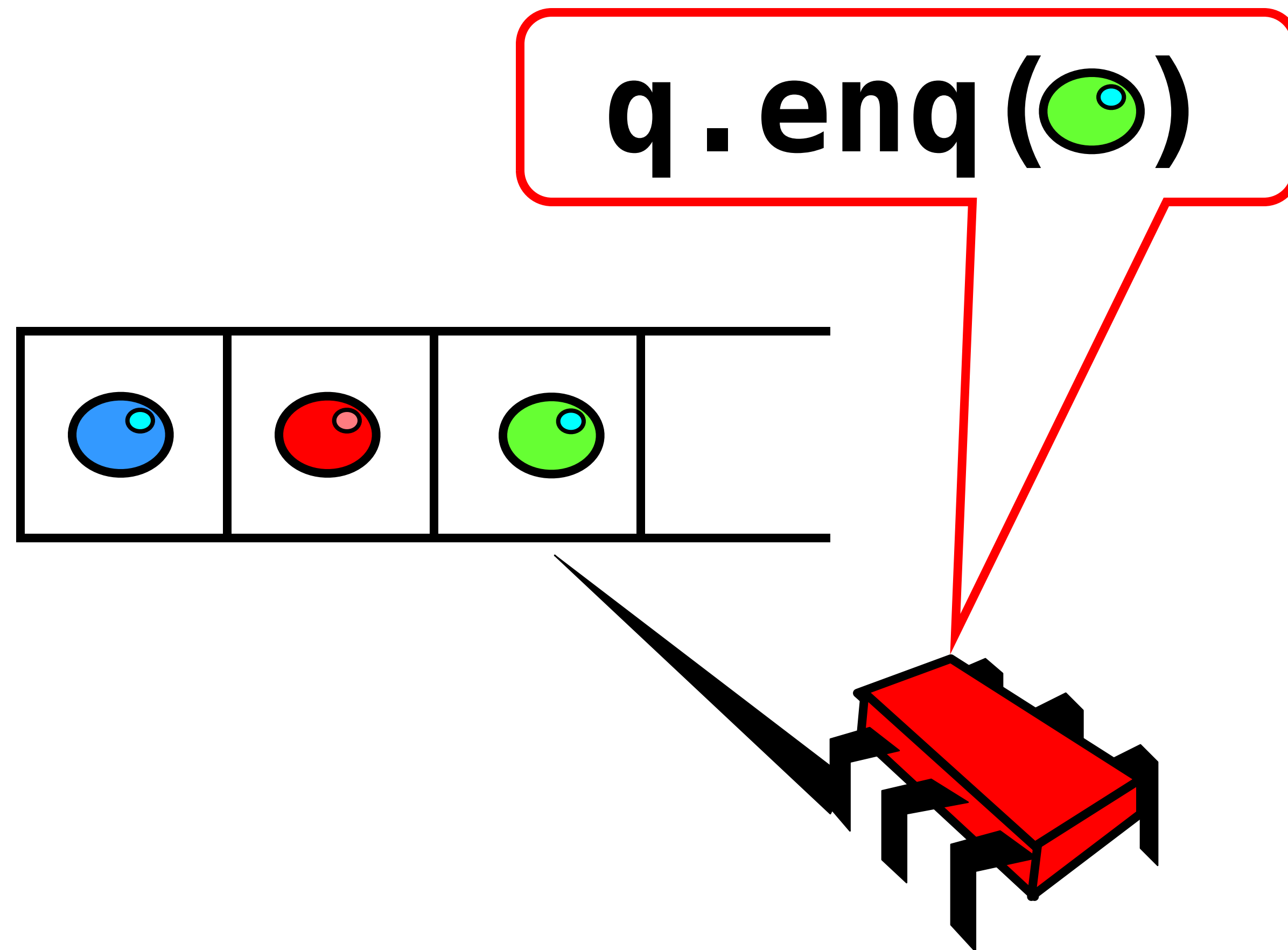 - How do we *tell if we're right*?

Objectivism

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

Concurrent Queues
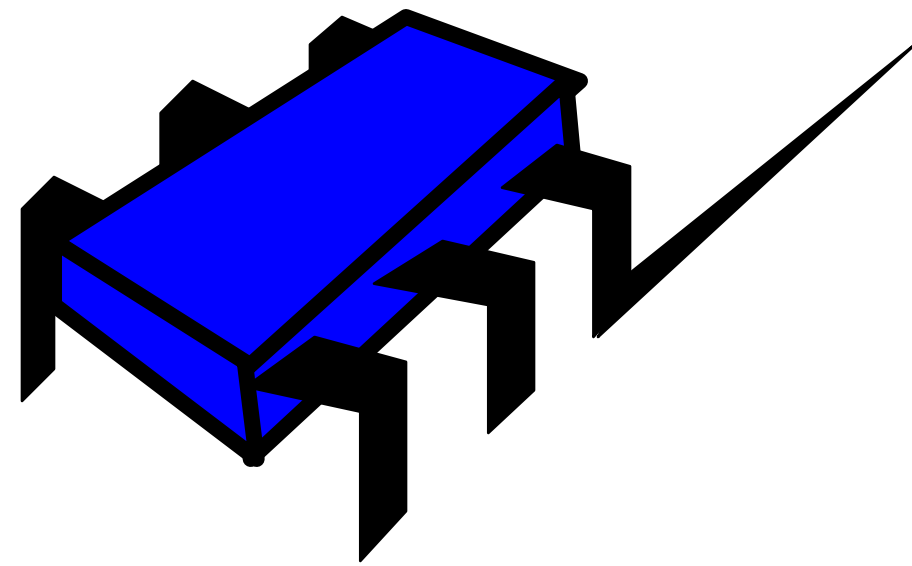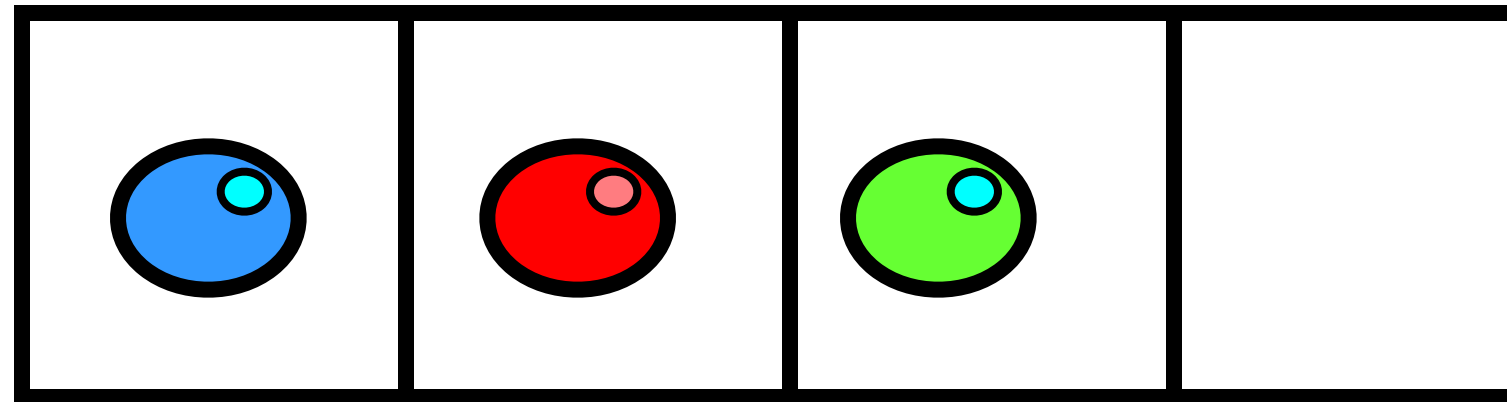
FIFO Queue — Enqeue method

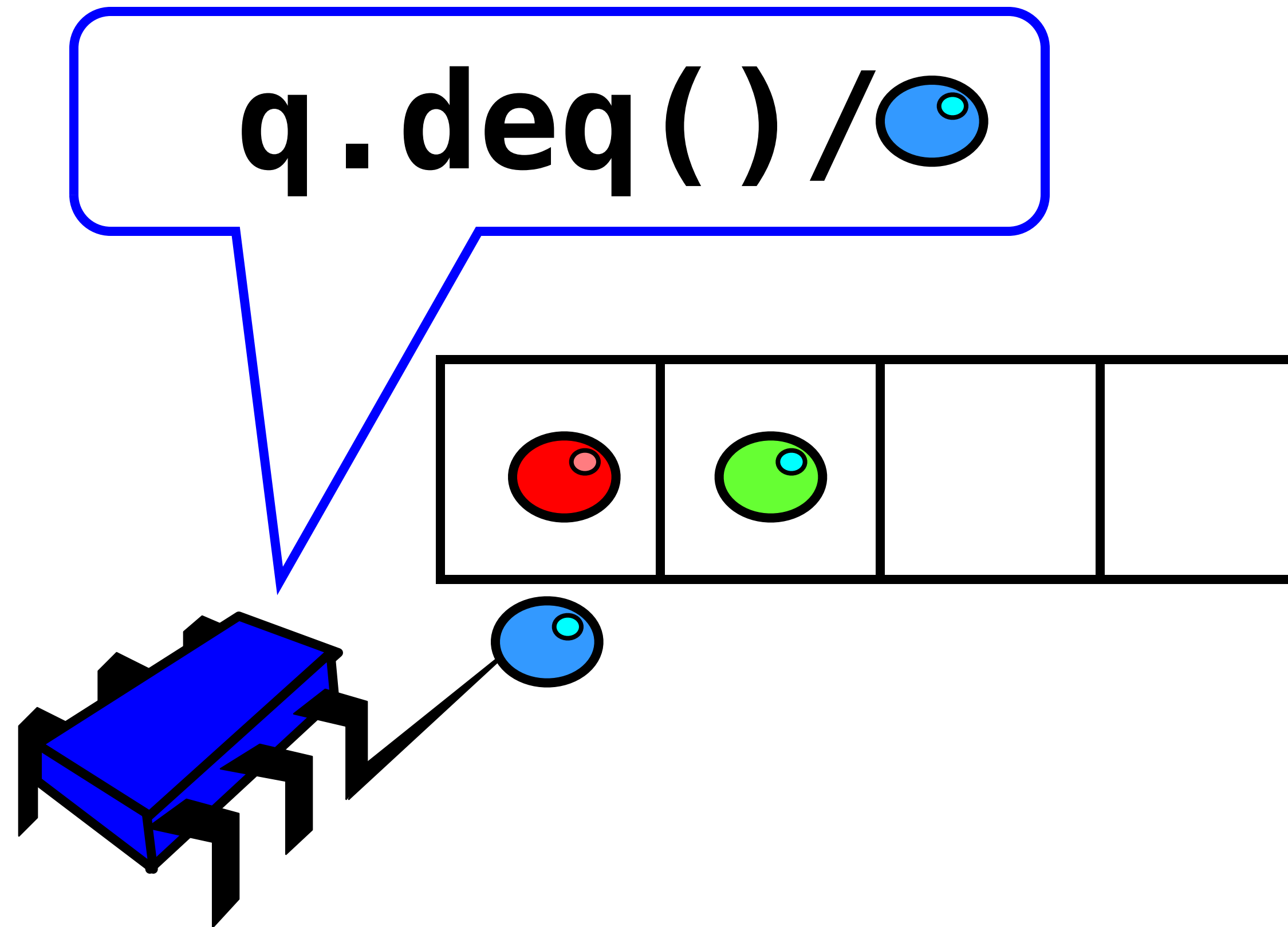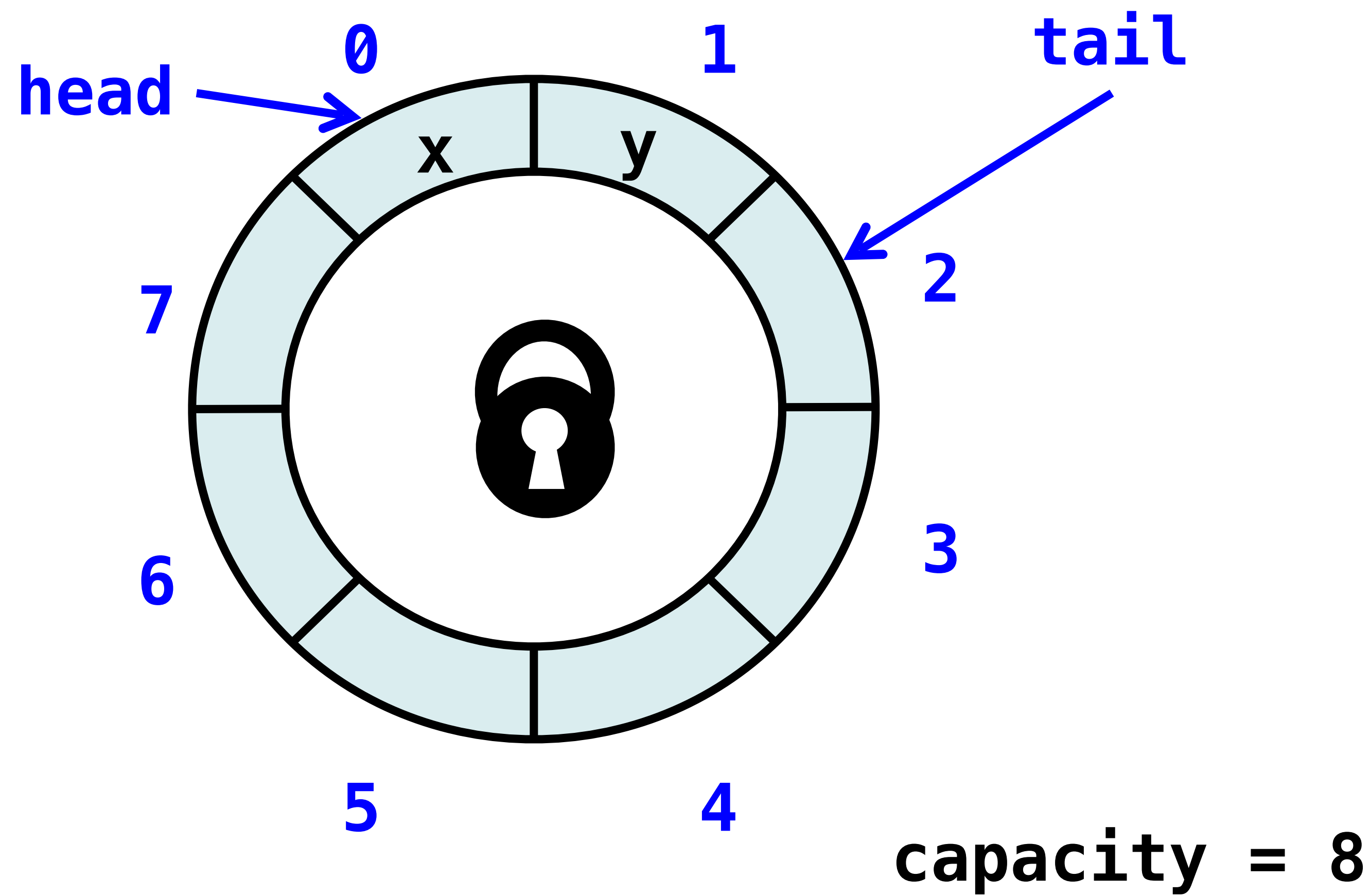# FIFO Queue — Enqeue method

FIFO Queue — Dequeue method

FIFO Queue — Dequeue method

Lock-based Queue

capacity = 8

Lock-based Queue

head → 0   1   tail

x   y

2

7

3

6

4

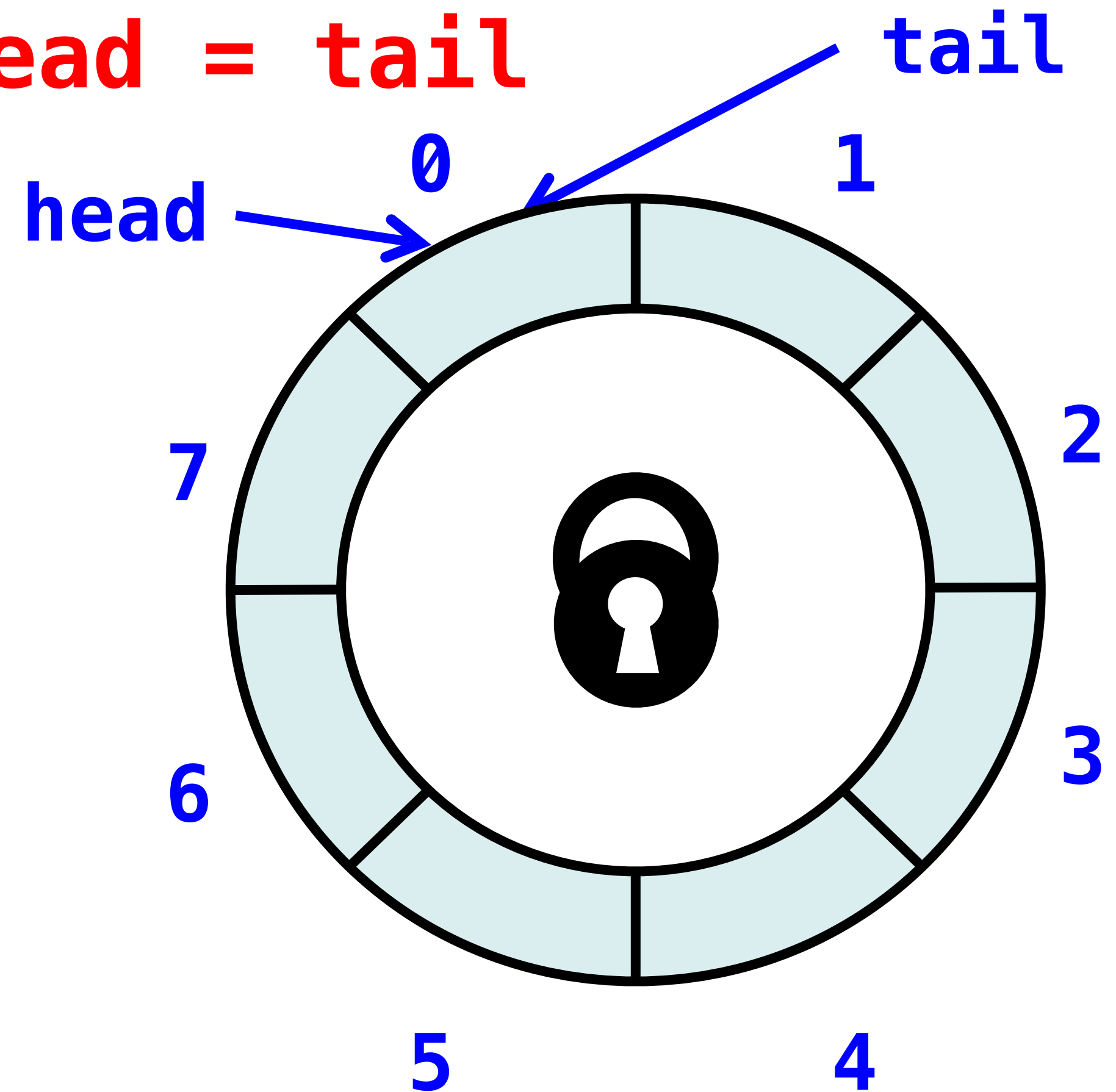Fields protected by single shared lock

5
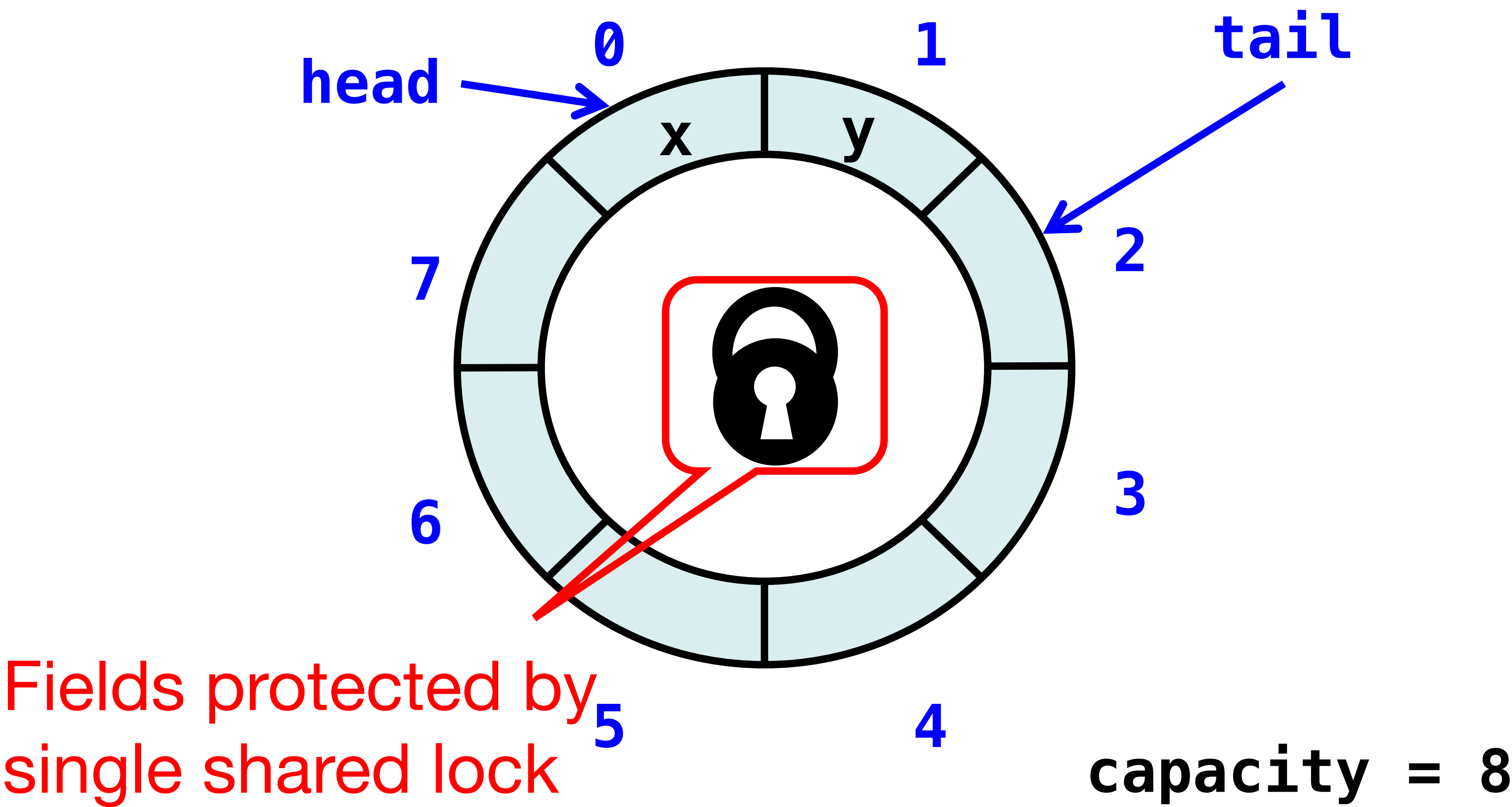
capacity = 8

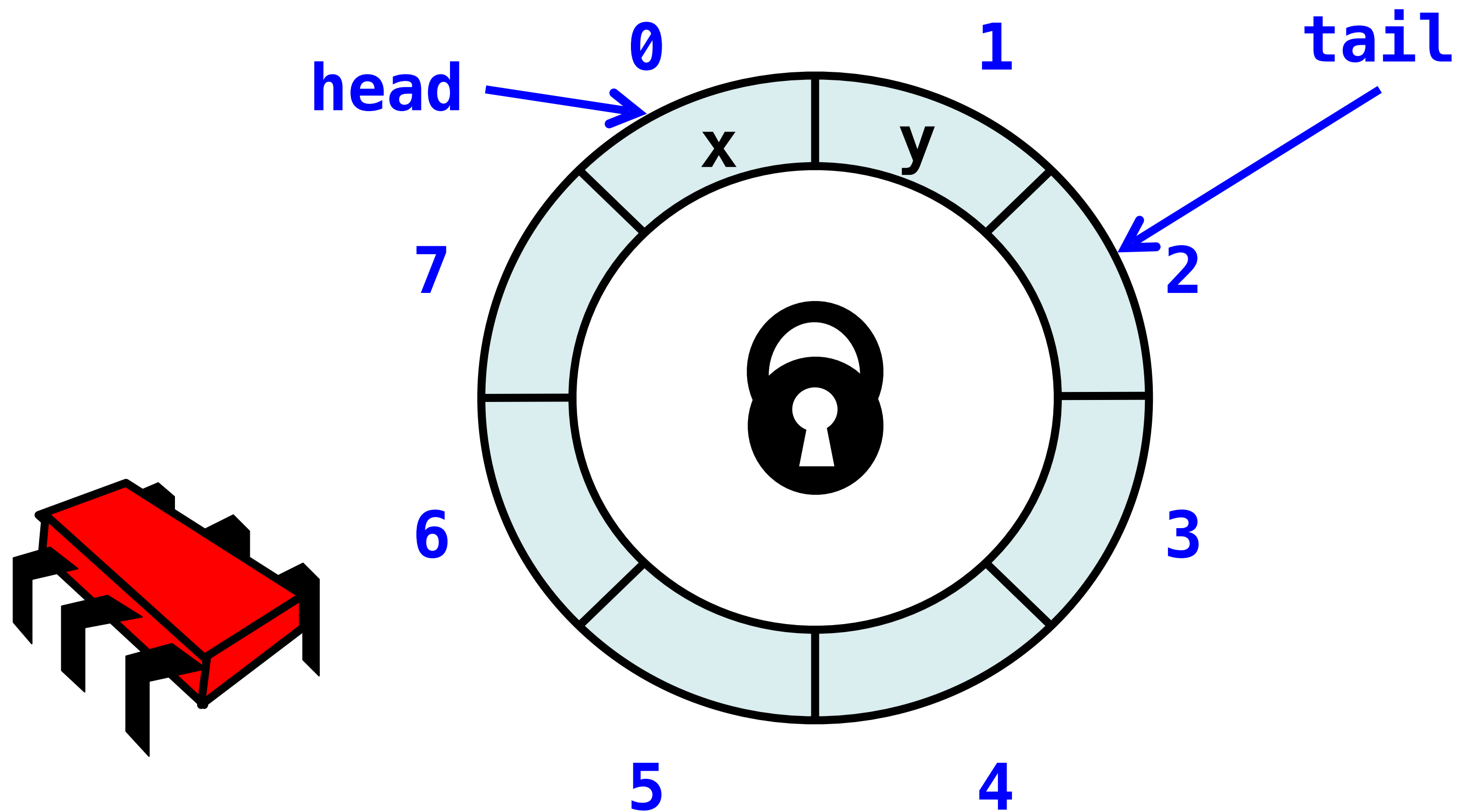# Lock-based Queue

```
exception Full
exception Empty

type 'a t = {
  items : 'a option array;
  capacity : int;
  mutable head : int;
  mutable tail : int;
  lock : Mutex.t;
}

let create capacity =
  {
    items = Array.make capacity None;
    capacity;
    head = 0;
    tail = 0;
    lock = Mutex.create ();
  }
```
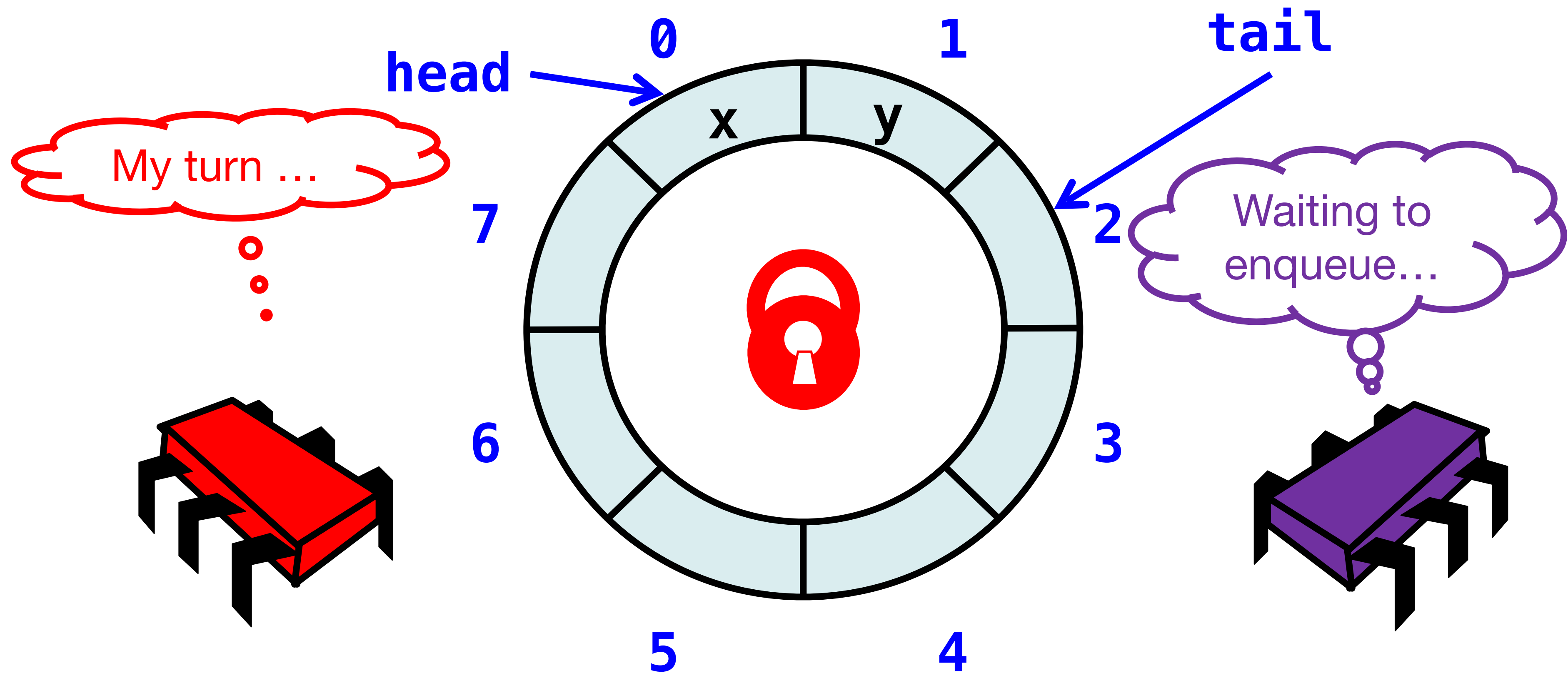
Initially: **head = tail**

Lock-based Queue — deq() operation
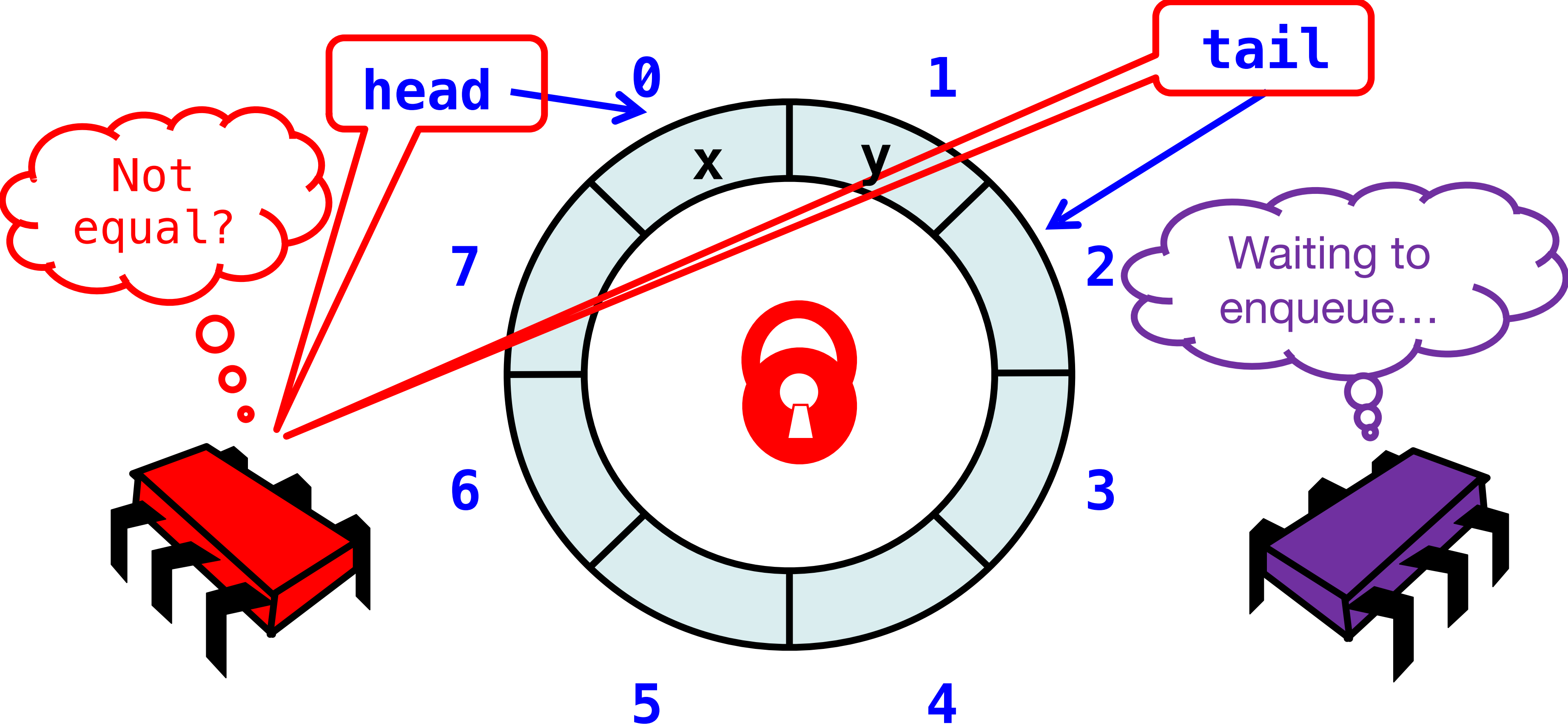
Acquire Lock

Acquire Lock

```
let deq q =
  Mutex.lock q.lock;
  try
     if q.tail = q.head then
       raise Empty;
     match q.items.(q.head mod q.capacity) with
     | None -> assert false
     | Some x ->
         q.head <- q.head + 1;
         Mutex.unlock q.lock;
         x
  with e ->
    Mutex.unlock q.lock;
    raise e
```

Acquire lock at
method start

Check if non empty

Check if non empty

```ocaml
let deq q =
  Mutex.lock q.lock;
  try
    if q.tail = q.head then
      raise Empty;
    match q.items.(q.head mod q.capacity) with
    | None -> assert false
    | Some x ->
        q.head <- q.head + 1;
        Mutex.unlock q.lock;
        x
  with e ->
    Mutex.unlock q.lock;
    raise e
```

If queue empty throw exception

In case of exceptions, lock released here

Modify the queue

Modify the queue

Modify the queue

```
let deq q =
  Mutex.lock q.lock;
  try
    if q.tail = q.head then
      raise Empty;
    match q.items.(q.head mod q.capacity) with
    | None -> assert false
    | Some x ->
        q.head <- q.head + 1;
        Mutex.unlock q.lock;
        x
  with e ->
    Mutex.unlock q.lock;
    raise e
```
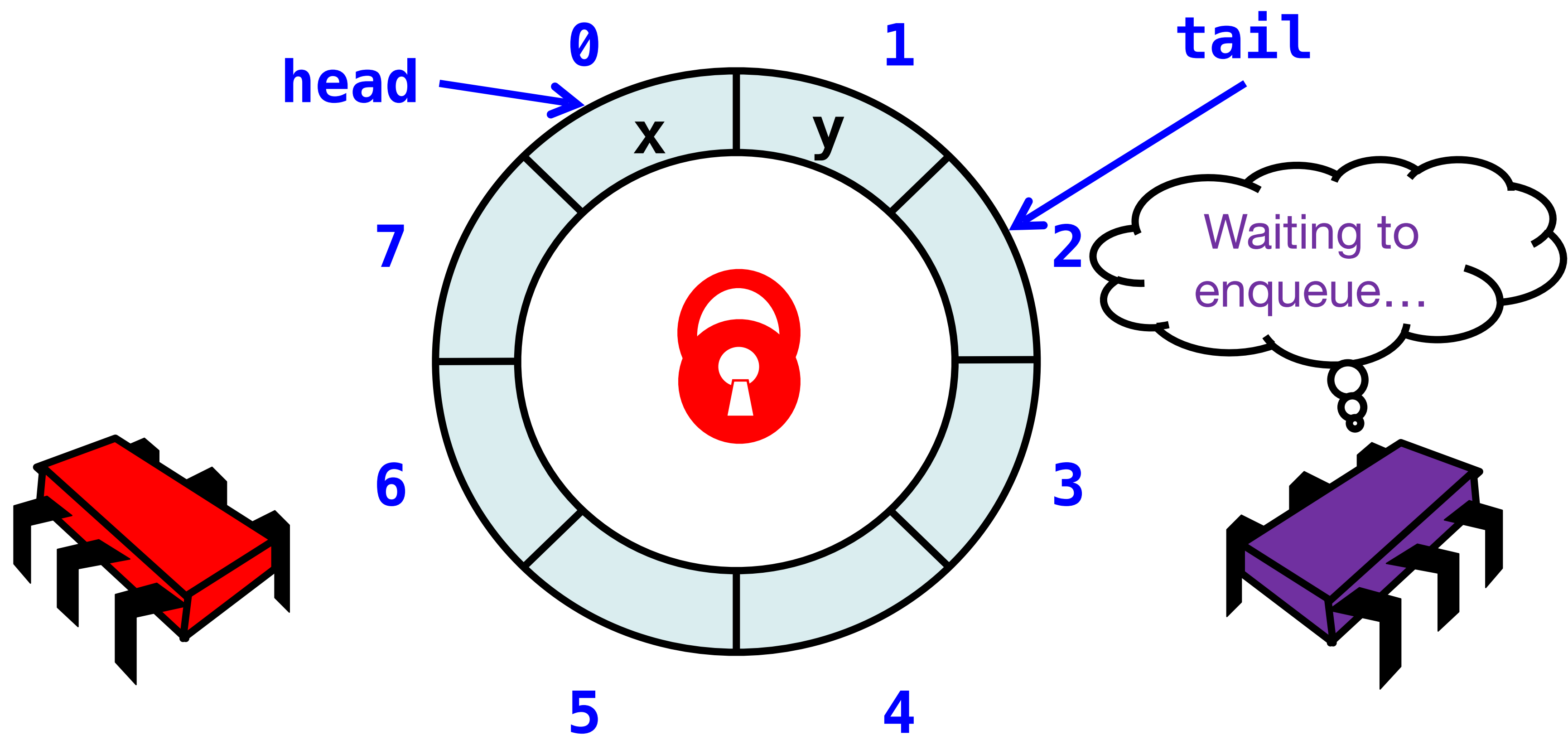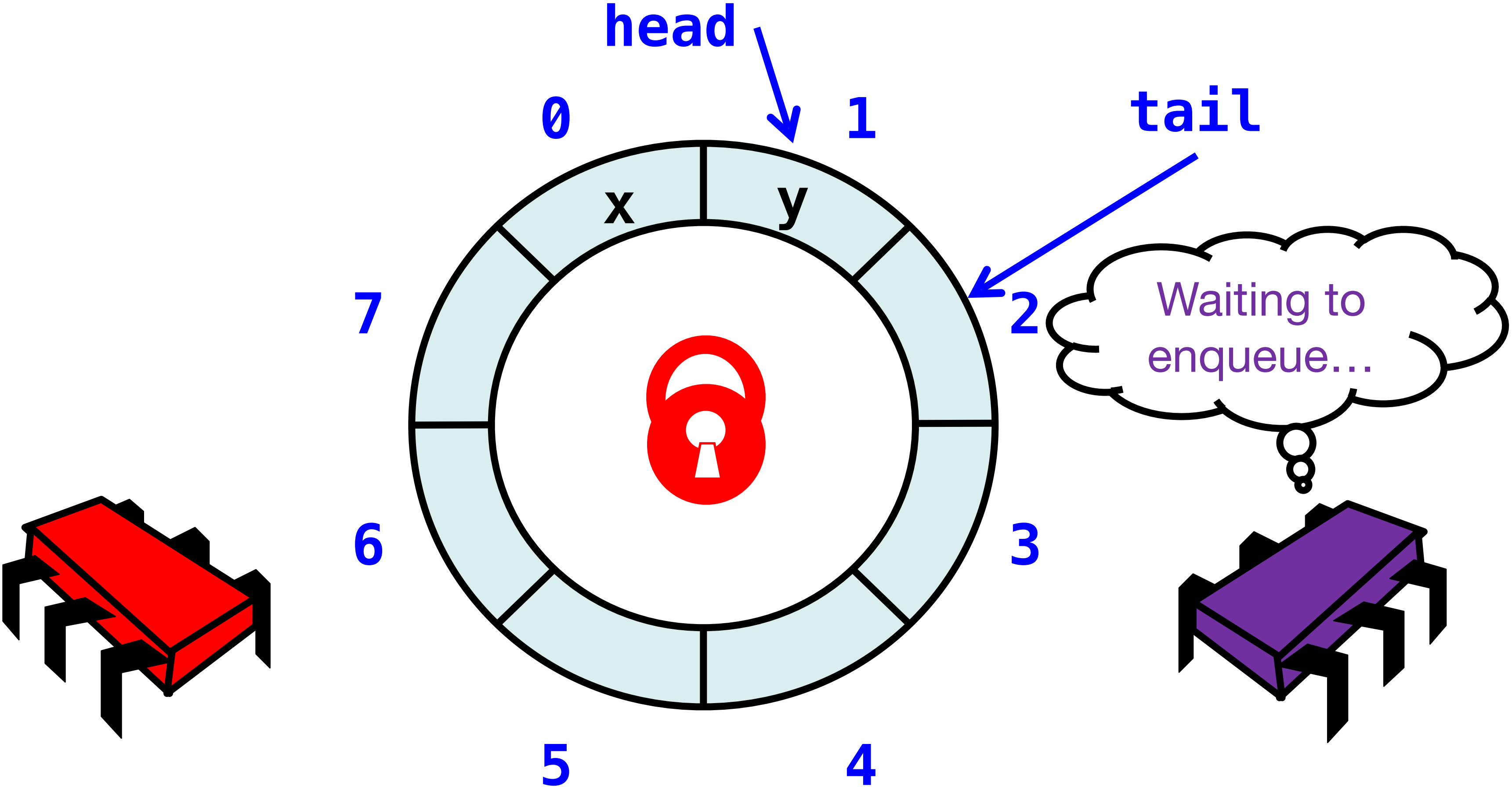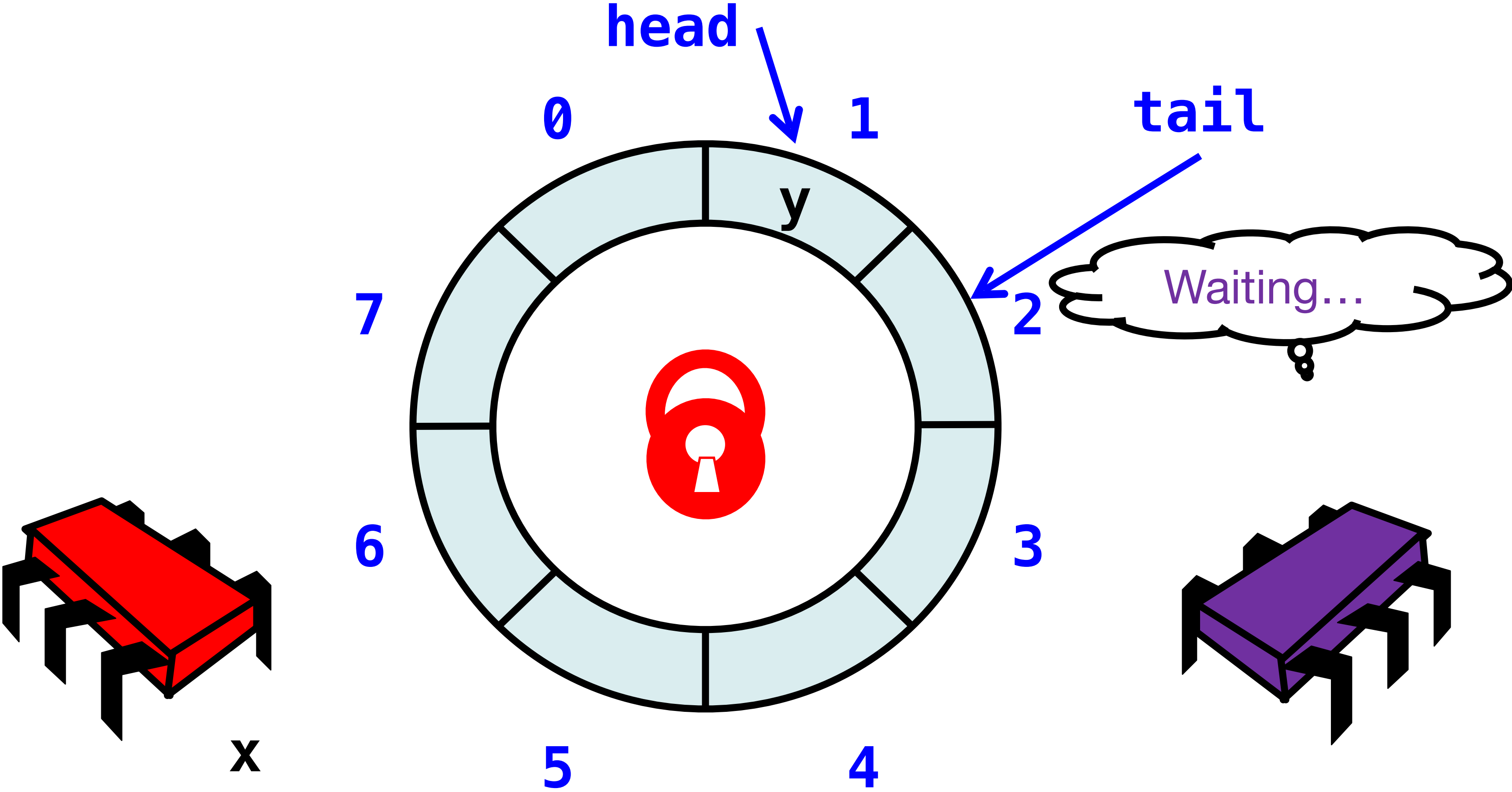
Queue not empty?
Remove item "x" and update head

Release the lock and return item

Release the lock and return item

```
let deq q =
  Mutex.lock q.lock;
  try
    if q.tail = q.head then
      raise Empty;
    match q.items.(q.head mod q.capacity) with
    | None -> assert false
    | Some x ->
        q.head <- q.head + 1;
        Mutex.unlock q.lock;
        x
  with e ->
    Mutex.unlock q.lock;
    raise e
```

Unlock and return item "x"

Implementation — deq()

```
let deq q =
  Mutex.lock q.lock;
  try
    if q.tail = q.head then
      raise Empty;
    match q.items.(q.head mod q.capacity) with
    | None -> assert false
    | Some x ->
        q.head <- q.head + 1;
        Mutex.unlock q.lock;
        x
  with e ->
    Mutex.unlock q.lock;
    raise e
```
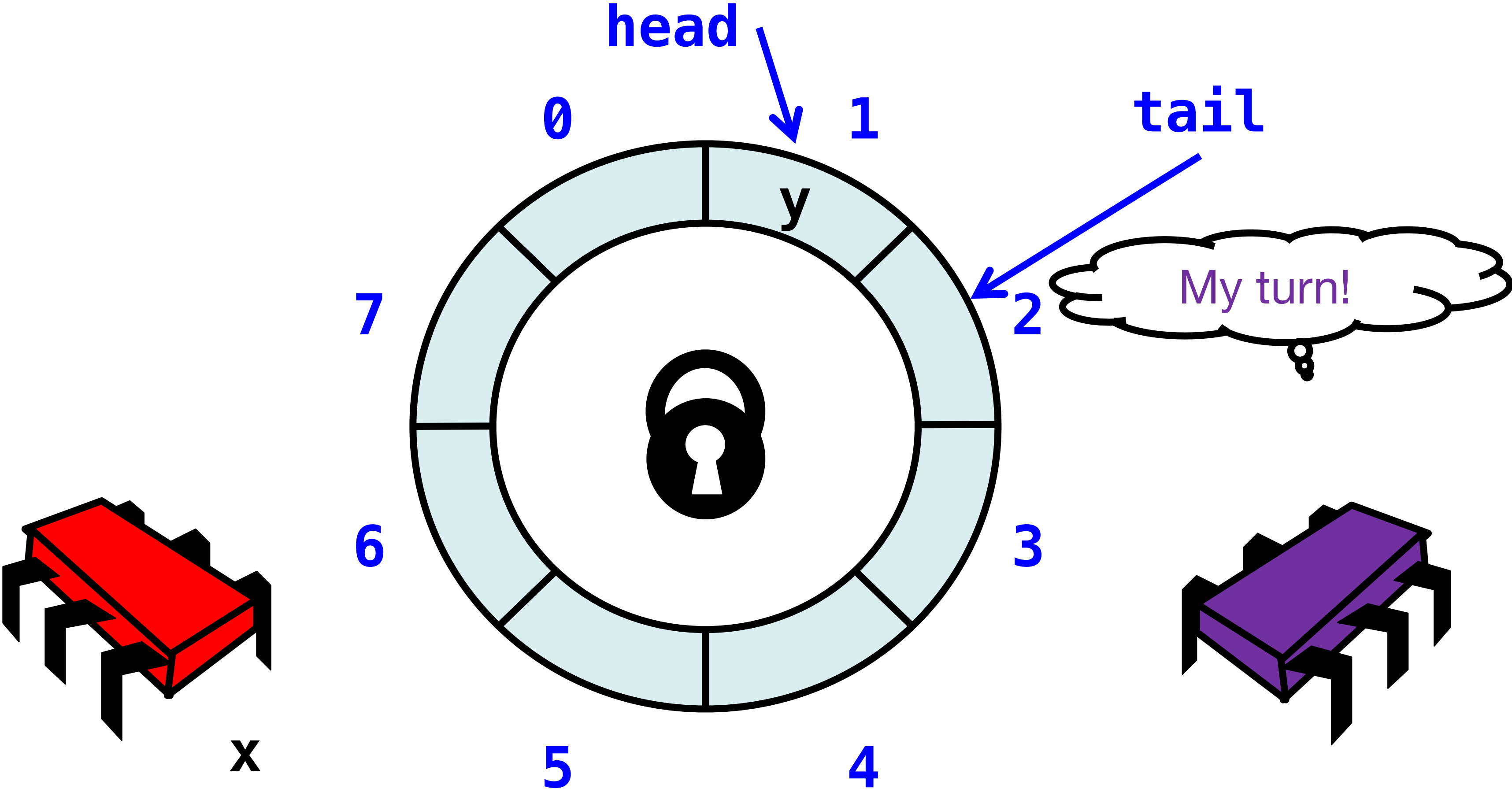
Implementation — deq()

```
let deq q =
  Mutex.lock q.lock;
  try
    if q.tail = q.head then
      raise Empty;
    match q.items.(q.head mod q.capacity) with
    | None -> assert false
    | Some x ->
        q.head <- q.head + 1;
        Mutex.unlock q.lock;
        x
  with e ->
    Mutex.unlock q.lock;
    raise e
```

Should be correct because modifications are mutually exclusive…

Implementation — deq()

```
let deq q =
  Mutex.lock q.lock;
  try
    if q.tail = q.head then
      raise Empty;
    match q.items.(q.head mod q.capacity) with
    | None -> assert false
    | Some x ->
        q.head <- q.head + 1;
        Mutex.unlock q.lock;
        x
  with e ->
    Mutex.unlock q.lock;
    raise e
```
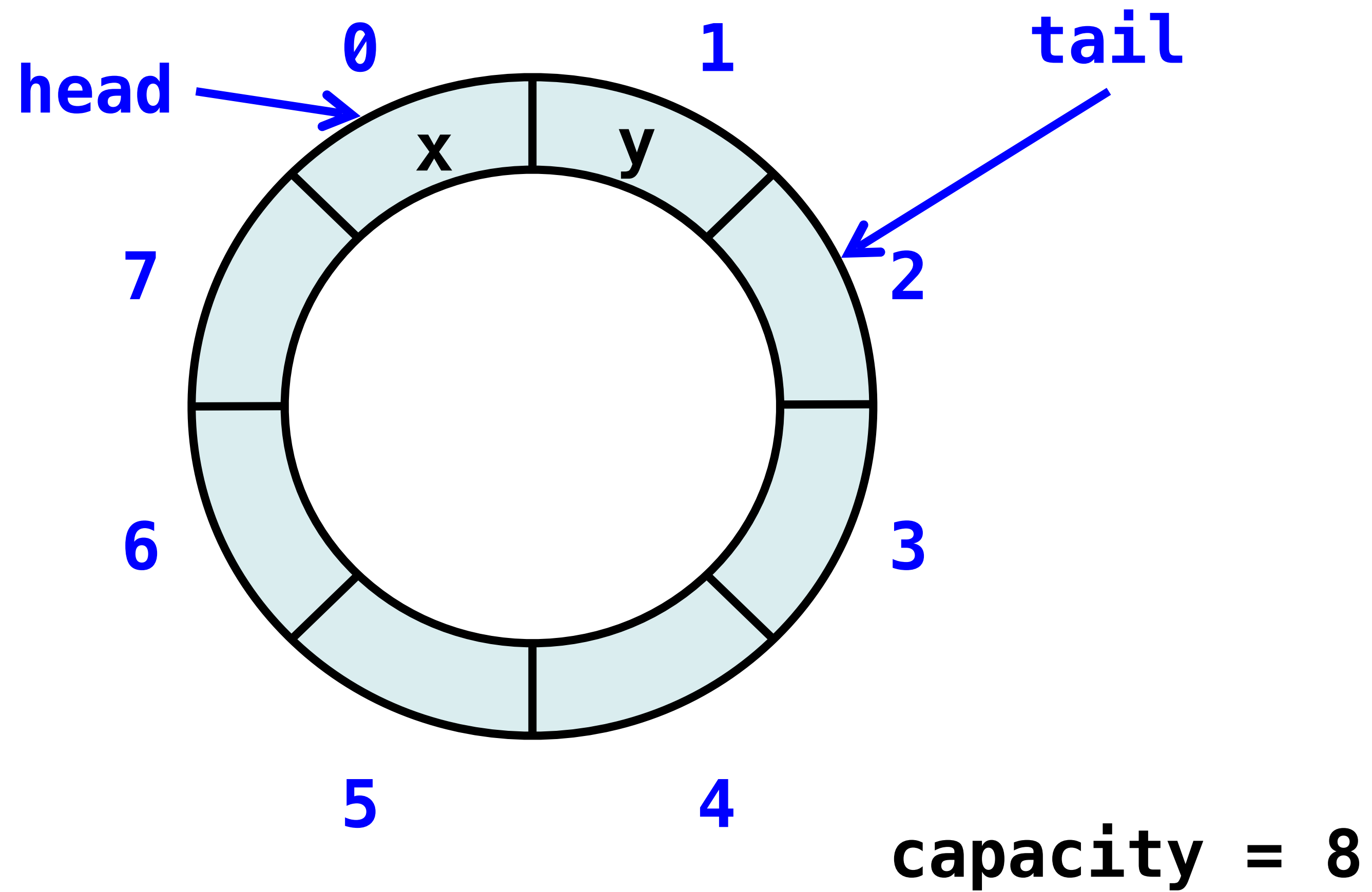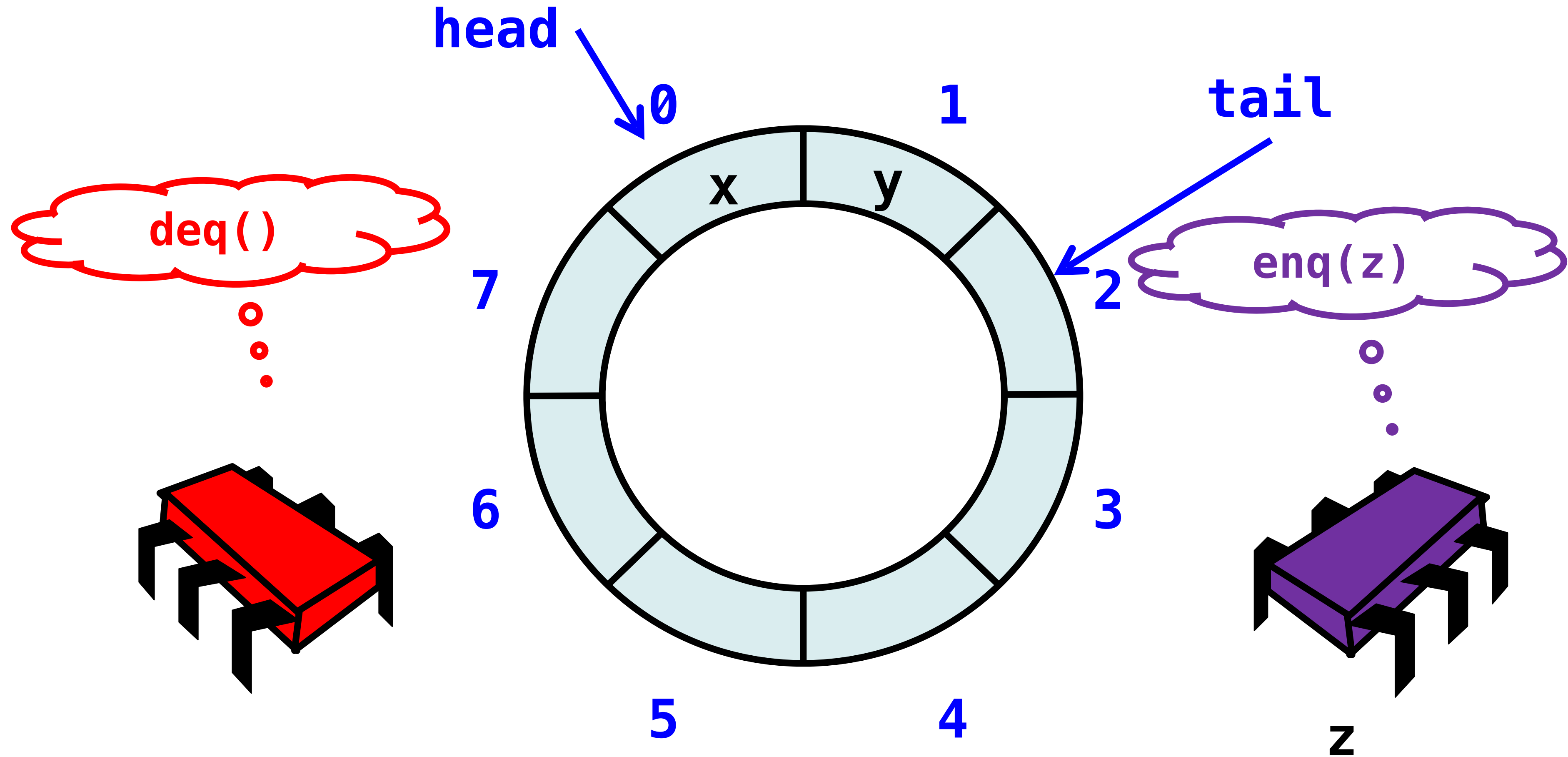
Should be correct because modifications are mutually exclusive…

Demo

Consider the following implementation

- The same thing without mutual exclusion
- For simplicity, only ***two*** threads
  - One thread enq only
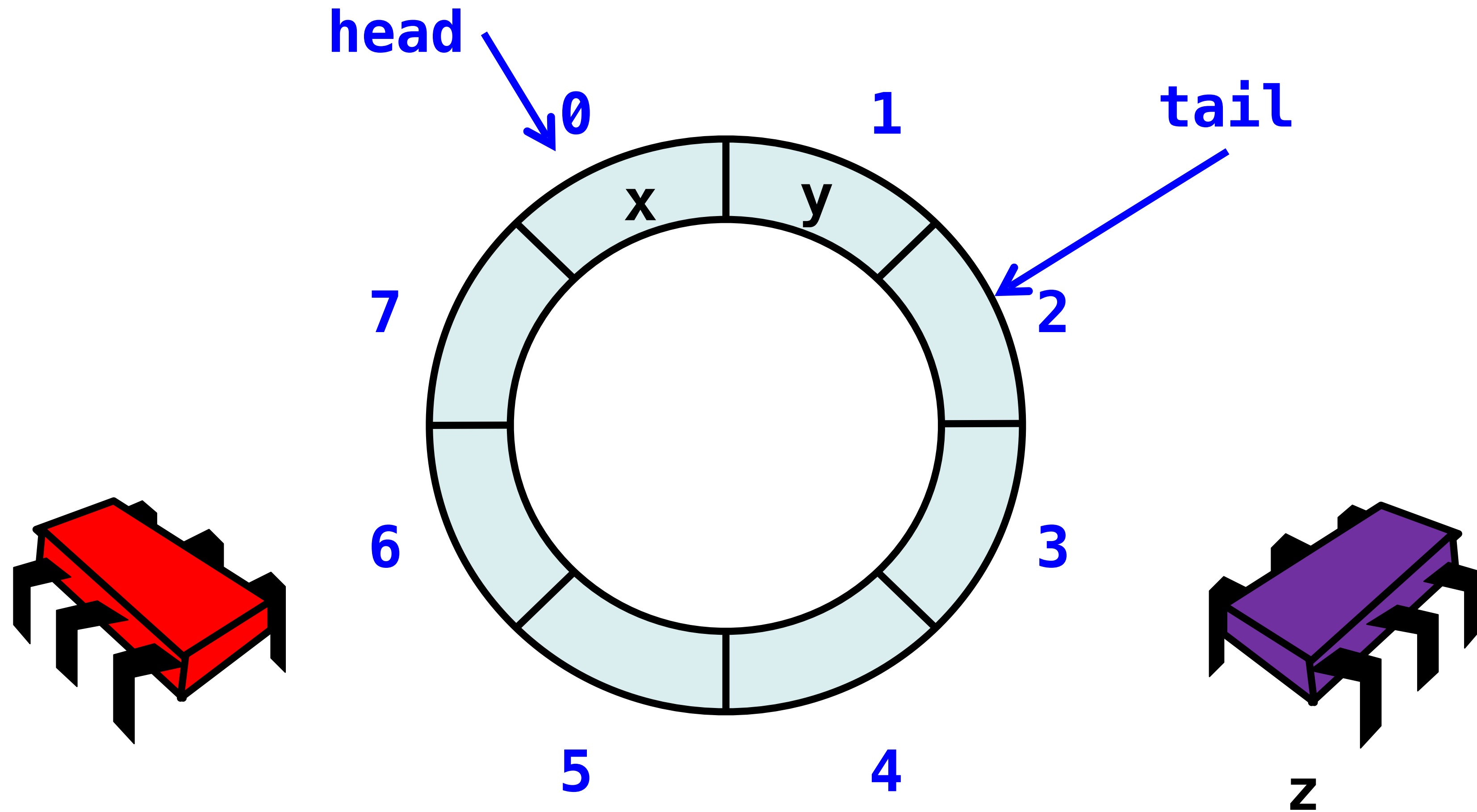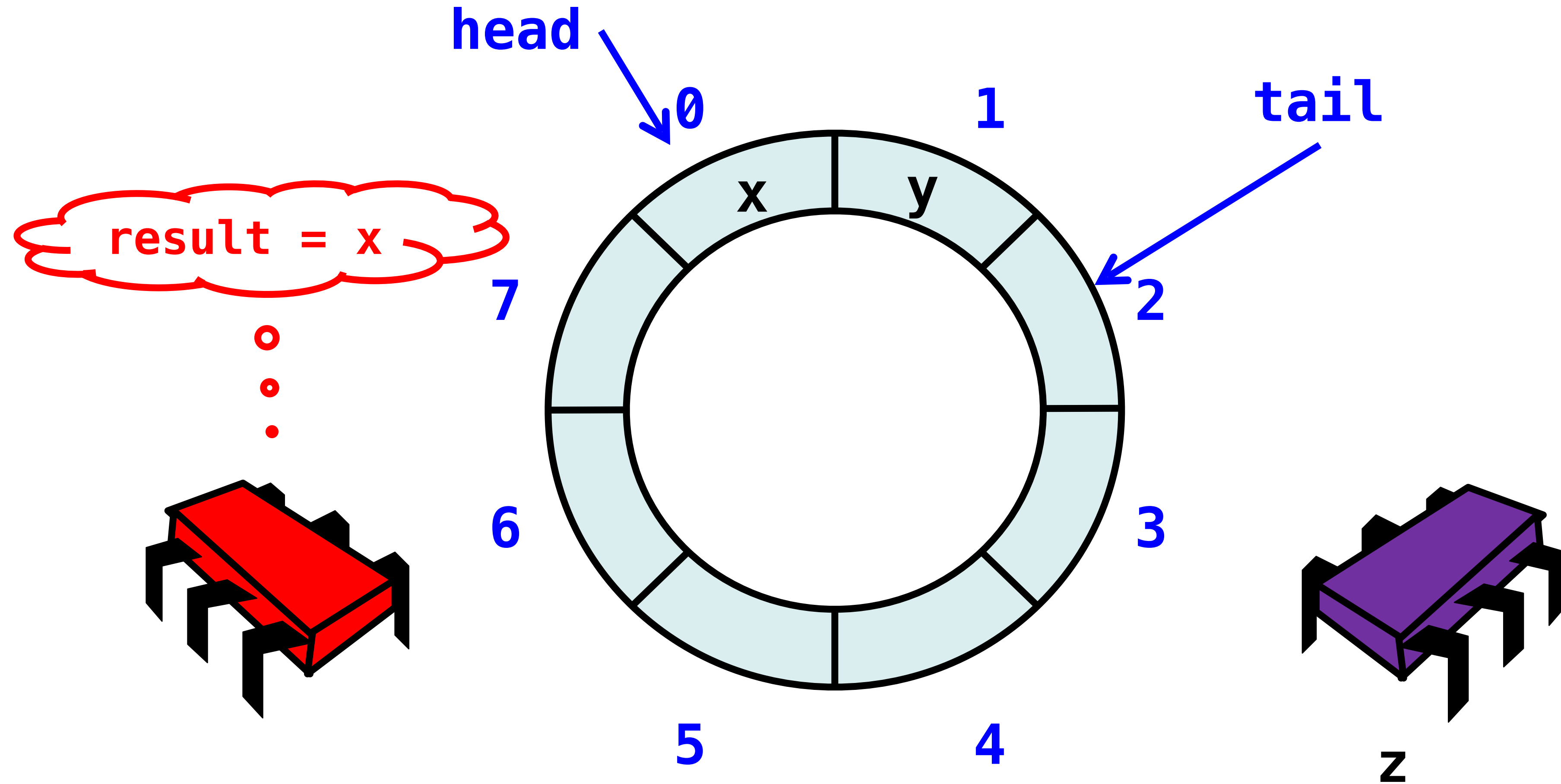 - The other deq only
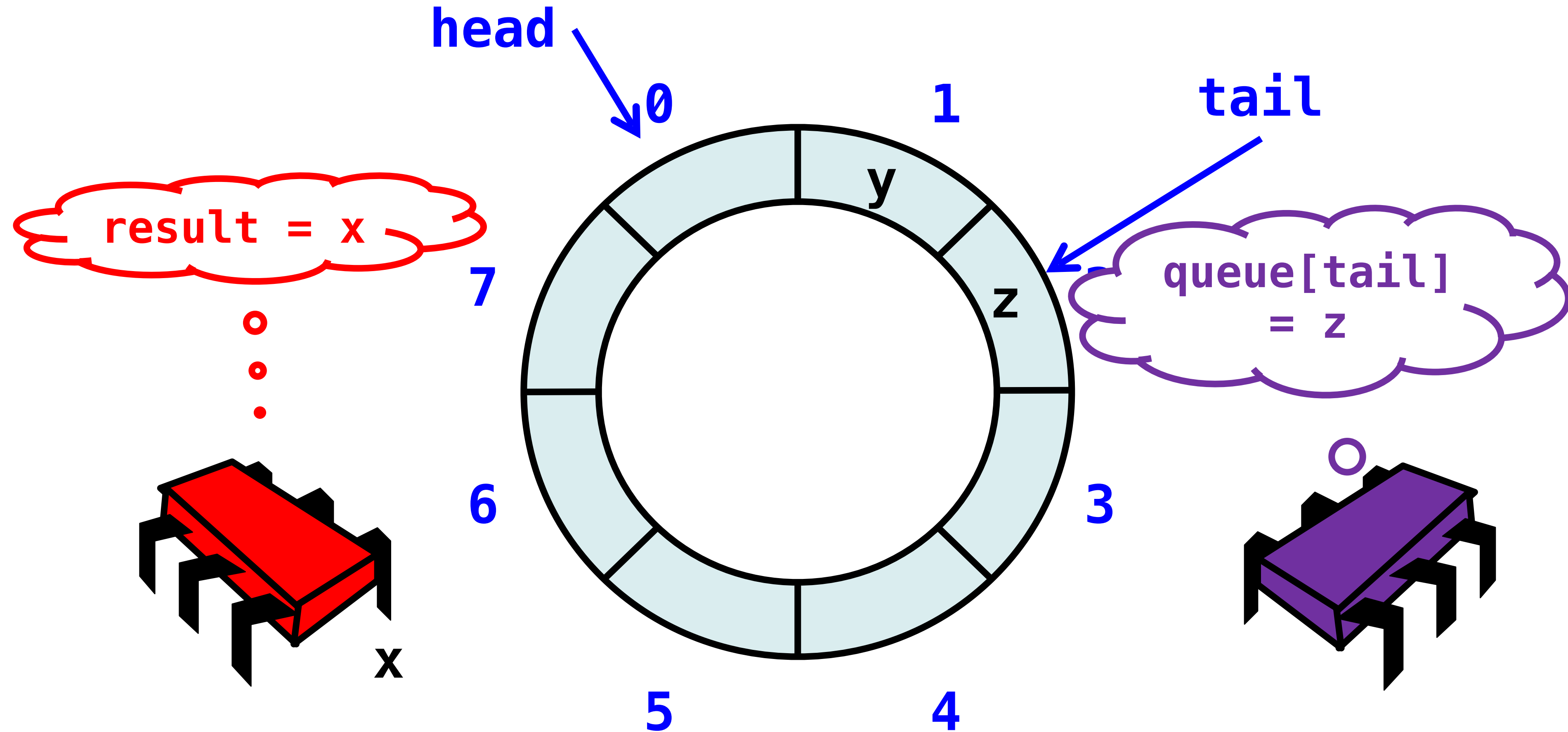
Wait-free 2-thread queue

capacity = 8

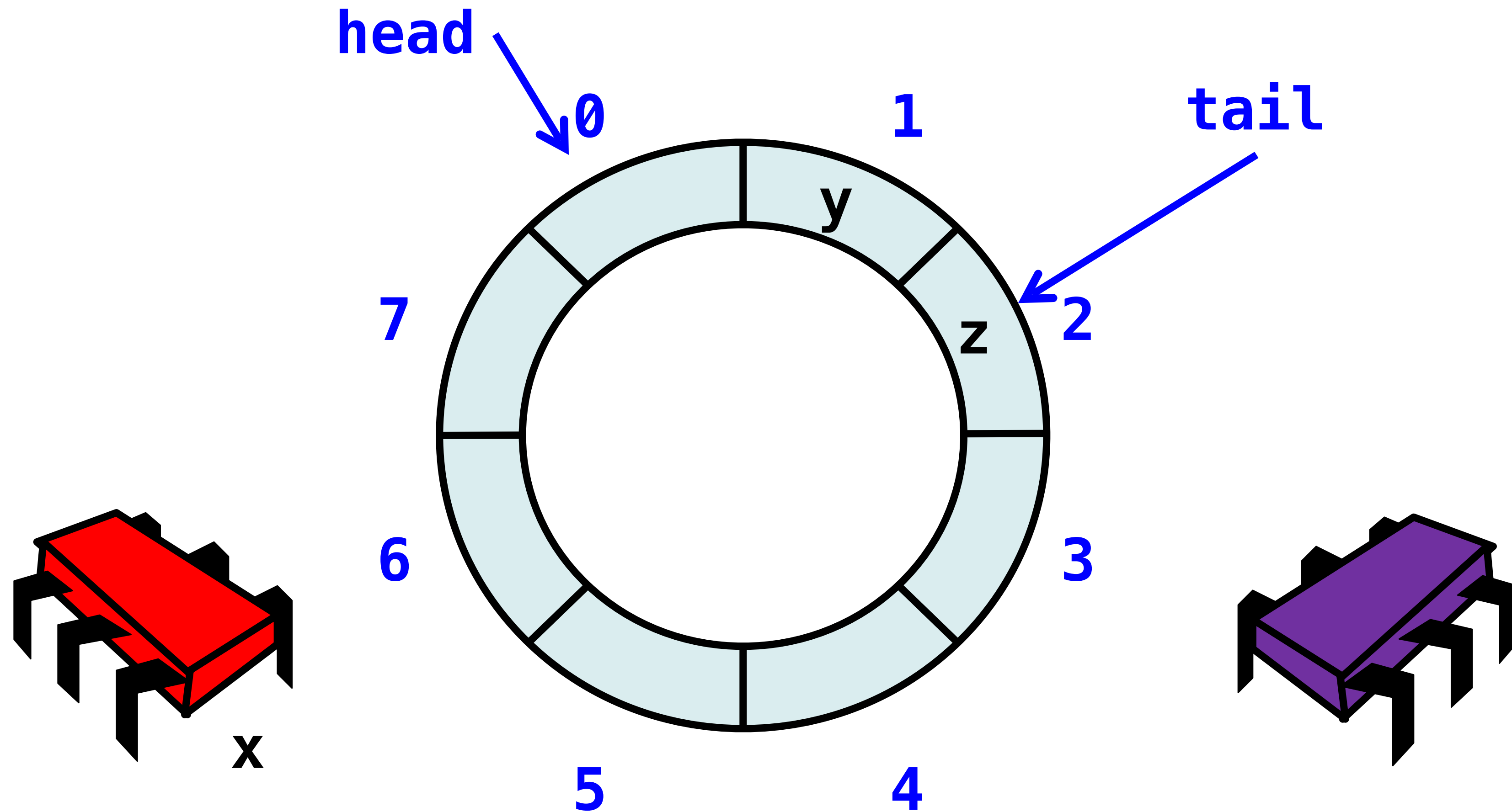# Wait-free 2-thread queue

Wait-free 2-thread queue

Wait-free 2-thread queue
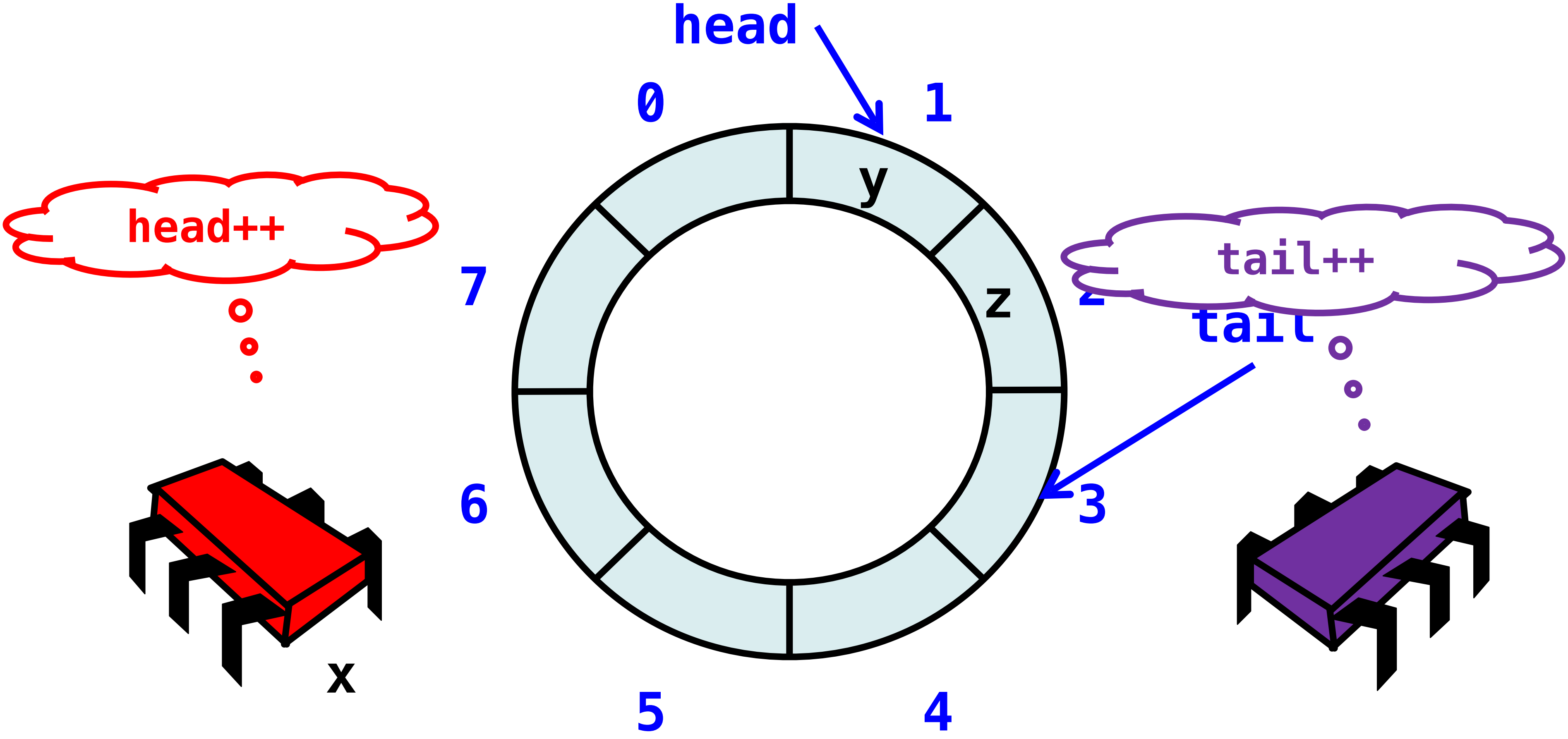
Wait-free 2-thread queue

Wait-free 2-thread queue

Wait-free 2-thread queue

Wait-free 2-thread queue

No locks needed!

```ocaml
(** Enqueue – should be called by only ONE thread *)
let enq q x =
  (* Check if queue is full *)
  if q.tail – q.head = q.capacity then
    raise Full;
  (* Write to the array *)
  q.items.(q.tail mod q.capacity) <- Some x;
  (* Advance tail *)
  q.tail <- q.tail + 1
```

```ocaml
(** Dequeue – should be called by only ONE thread *)
let deq q =
  (* Check if queue is empty *)
  if q.tail = q.head then
    raise Empty;
  (* Read from the array *)
  match q.items.(q.head mod q.capacity) with
  | None -> assert false  (* Should never happen *)
  | Some x ->
      (* Advance head *)
      q.head <- q.head + 1;
      x
```

Wait-free 2-thread queue

No locks needed!

```ocaml
(** Enqueue - should be called by only ONE thread *)
let enq q x =
  (* Check if queue is full *)
  if q.tail - q.head = q.capacity then
    raise Full;
  (* Write to the array *)
  q.items.(q.tail mod q.capacity) <- Some x;
  (* Advance tail *)
  q.tail <- q.tail + 1
```

```ocaml
(** Dequeue - should be called by only ONE thread *)
let deq q =
  (* Check if queue is empty *)
  if q.tail = q.head then
    raise Empty;
  (* Read from the array *)
  match q.items.(q.head mod q.capacity) with
  | None -> assert false  (* Should never happen *)
  | Some x ->
      (* Advance head *)
      q.head <- q.head + 1;
      x
```

Demo

Wait-free 2-thread queue

No locks needed!

```
(** Enqueue - should be called by only ONE thread *)     (** Dequeue - should be called by only ONE thread *)
let enq q x =                                             let deq q =
  (* Check if queue is full *)                              (* Check if queue is empty *)
  if q.tail - q.head = q.capacity then                      if q.tail = q.head then
    raise Full;                                               raise Empty;
  (* Write to the array *)                                  (* Read from the array *)
  q.items.(q.tail mod q.capacity) <- Some x;                match q.i          ead mod q.capacity) with
  (* Advance tail *)                                        | N            lse  (* Should never happen *)
  q.tail <- q.tail + 1
                                                                            head + 1;
```

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

Demo

Concurrency Specification

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**

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,

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.

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, cont*)

  - the object will be in some other state

  - when the method returns,

Pre and Post Conditions for Dequeue

- Precondition:

  – Queue is *non-empty*

- Postcondition:

  – Returns first item in queue

- Postcondition:

  – Removes first item in queue

Pre and Post Conditions for Dequeue

- Precondition:
  - Queue is *non-empty*

- Postcondition:
  - Returns first item in queue

- Postcondition:
  - Removes first item in queue

- Precondition:
  - Queue is *empty*

- Postcondition:
  - Raises `Empty` exception

- Postcondition:
  - Queue state is unchanged

Why Sequential Specifications Totally Rock

- **Interactions among *methods* captured by side-effects on object state**

  – State meaningful between method calls

- ***Documentation* size is linear in the 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

time

Methods take time

Methods take time

Methods take time

invocation
12:00

q.enq(👁)
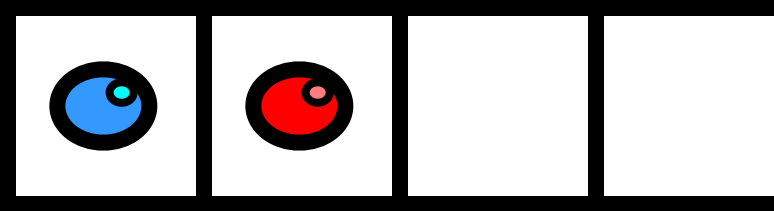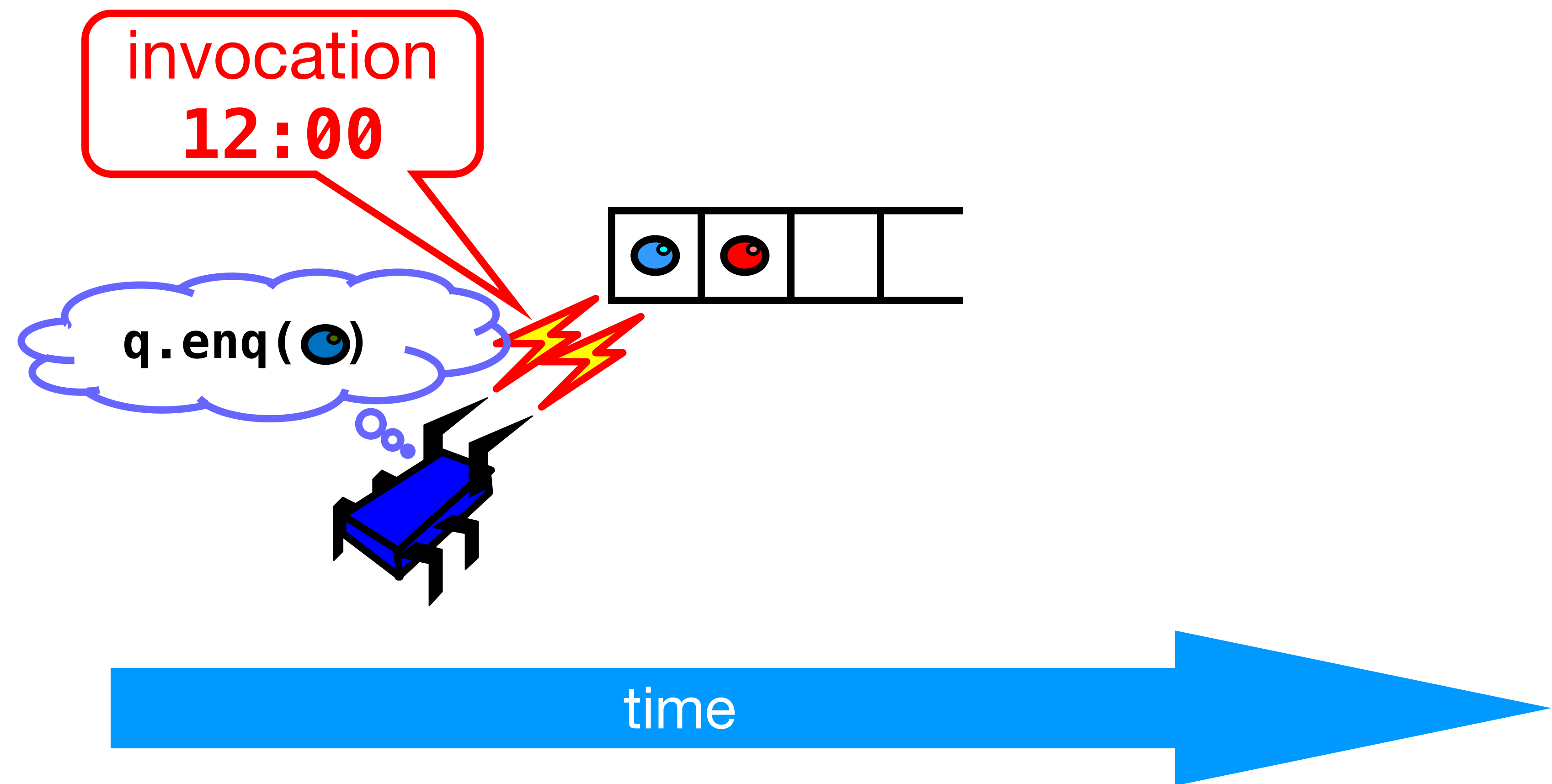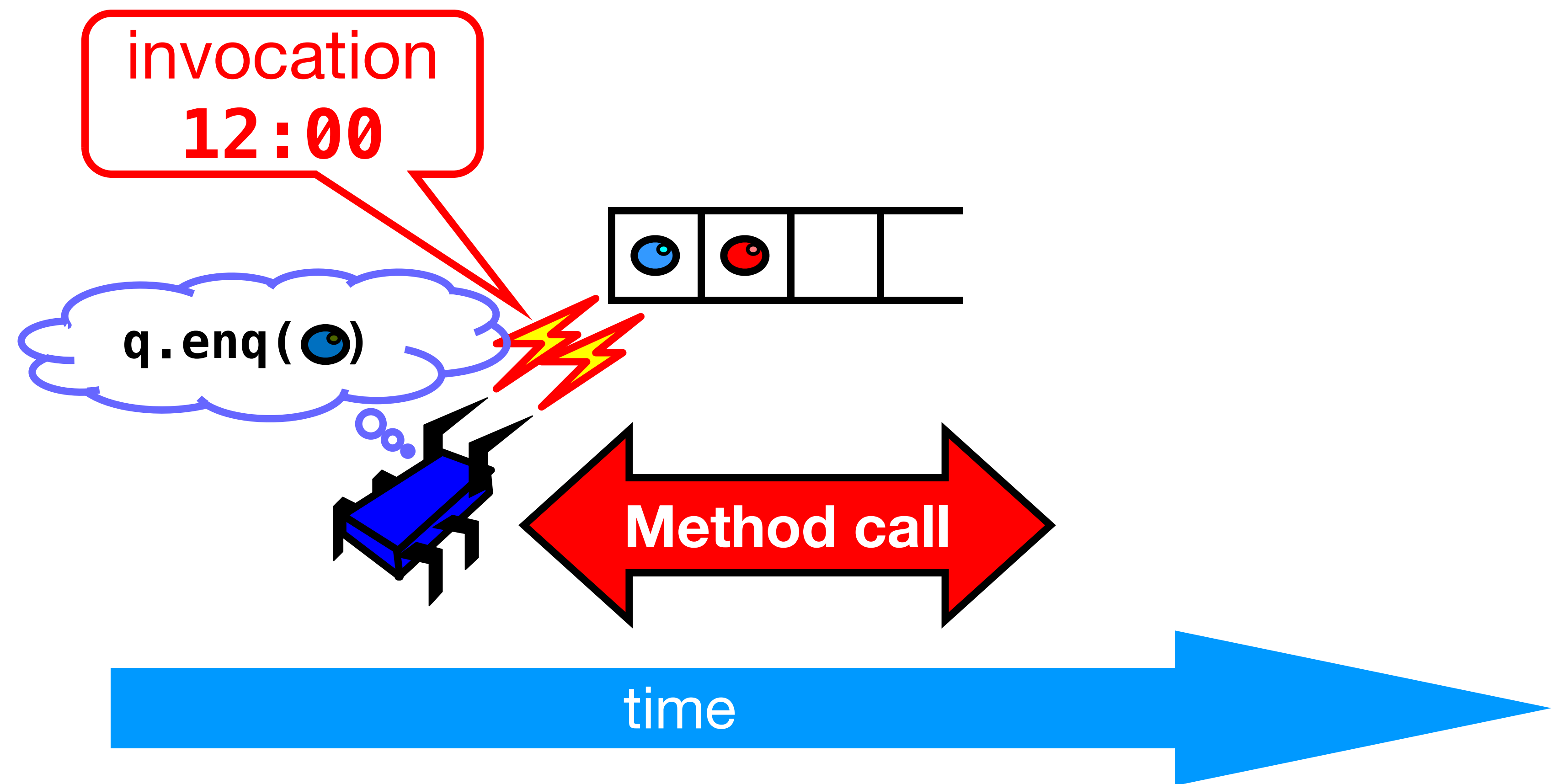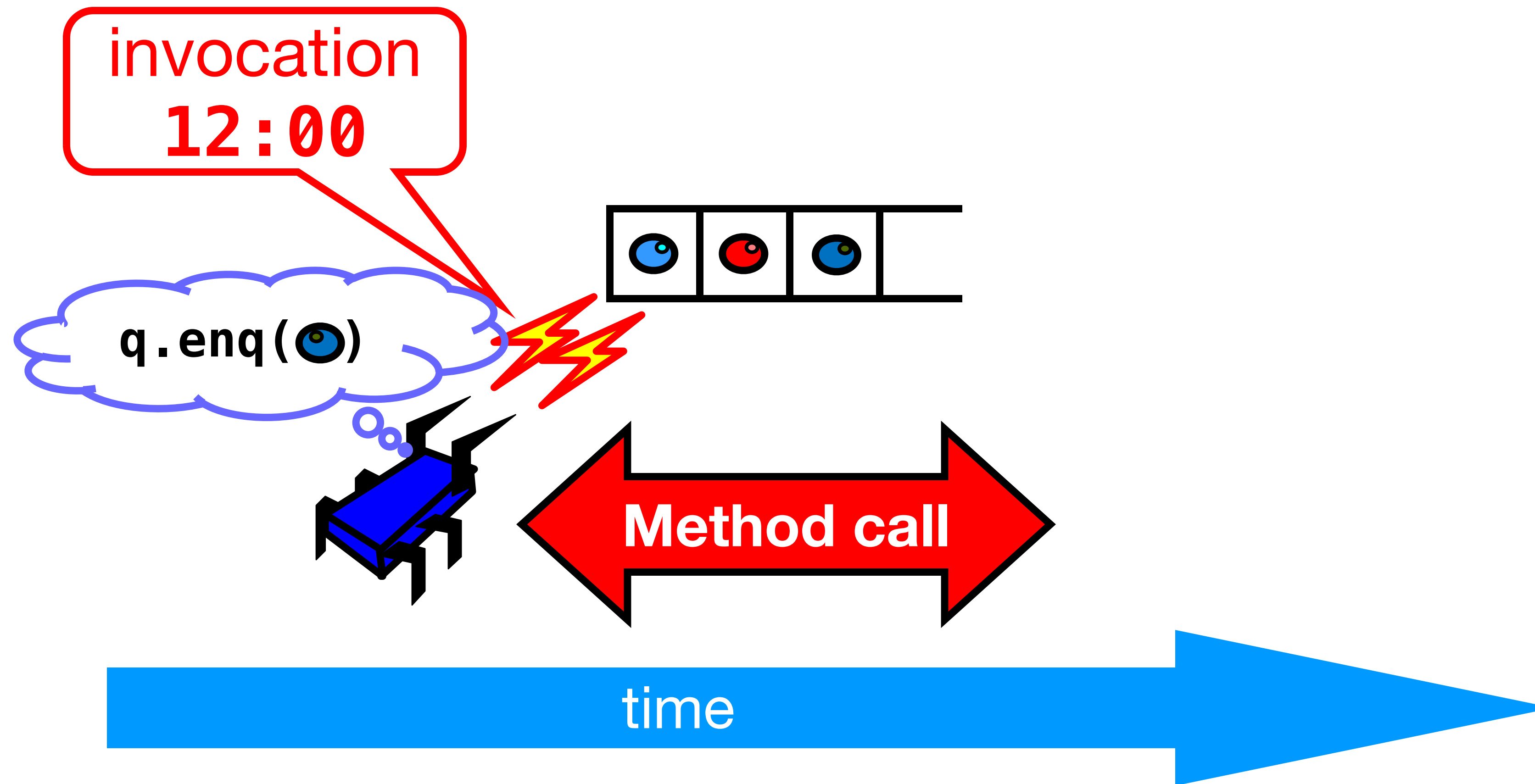
Method call

time
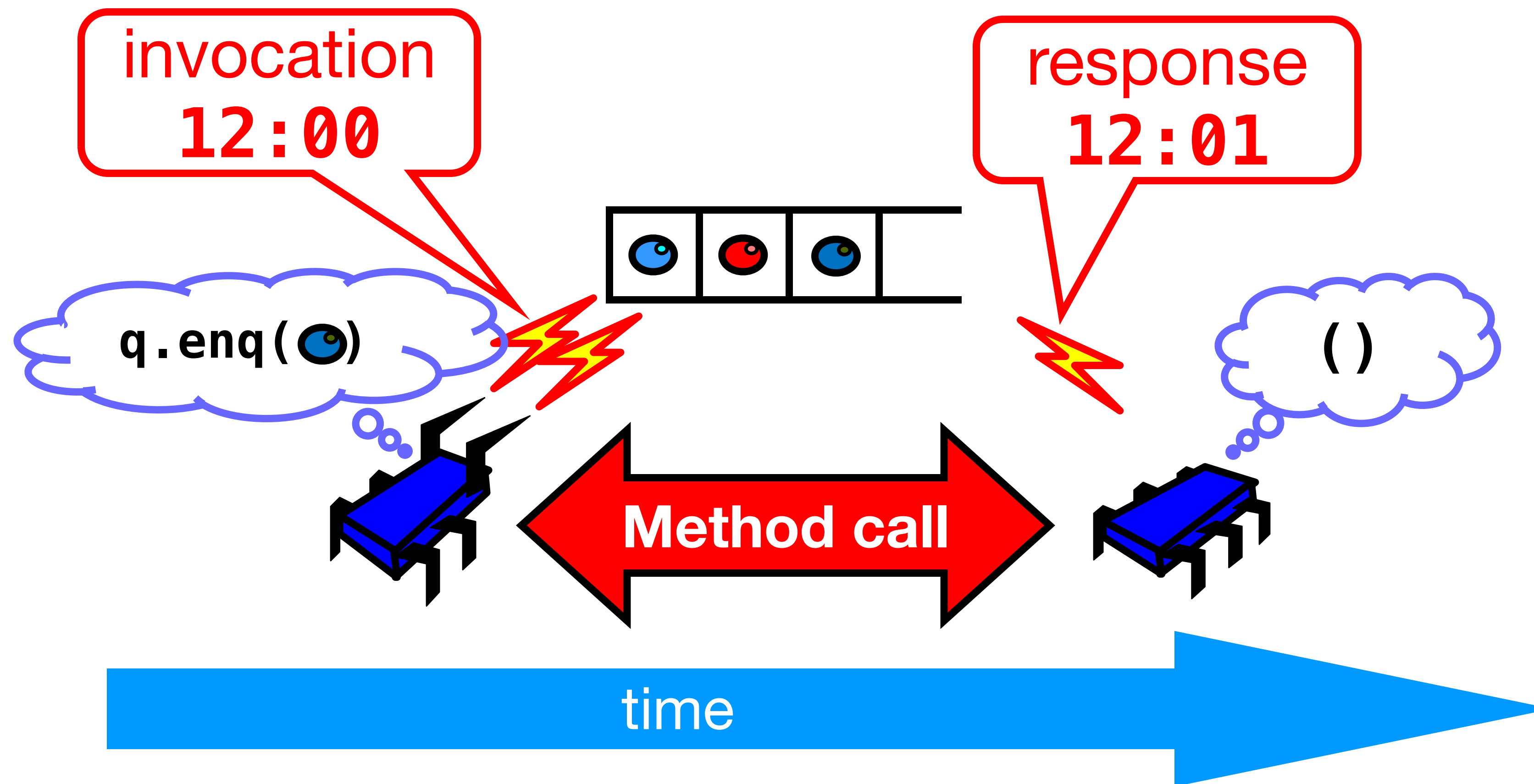
Methods take time

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
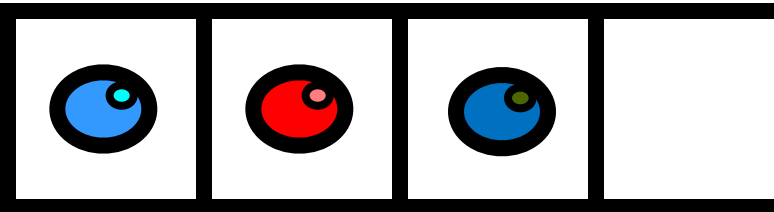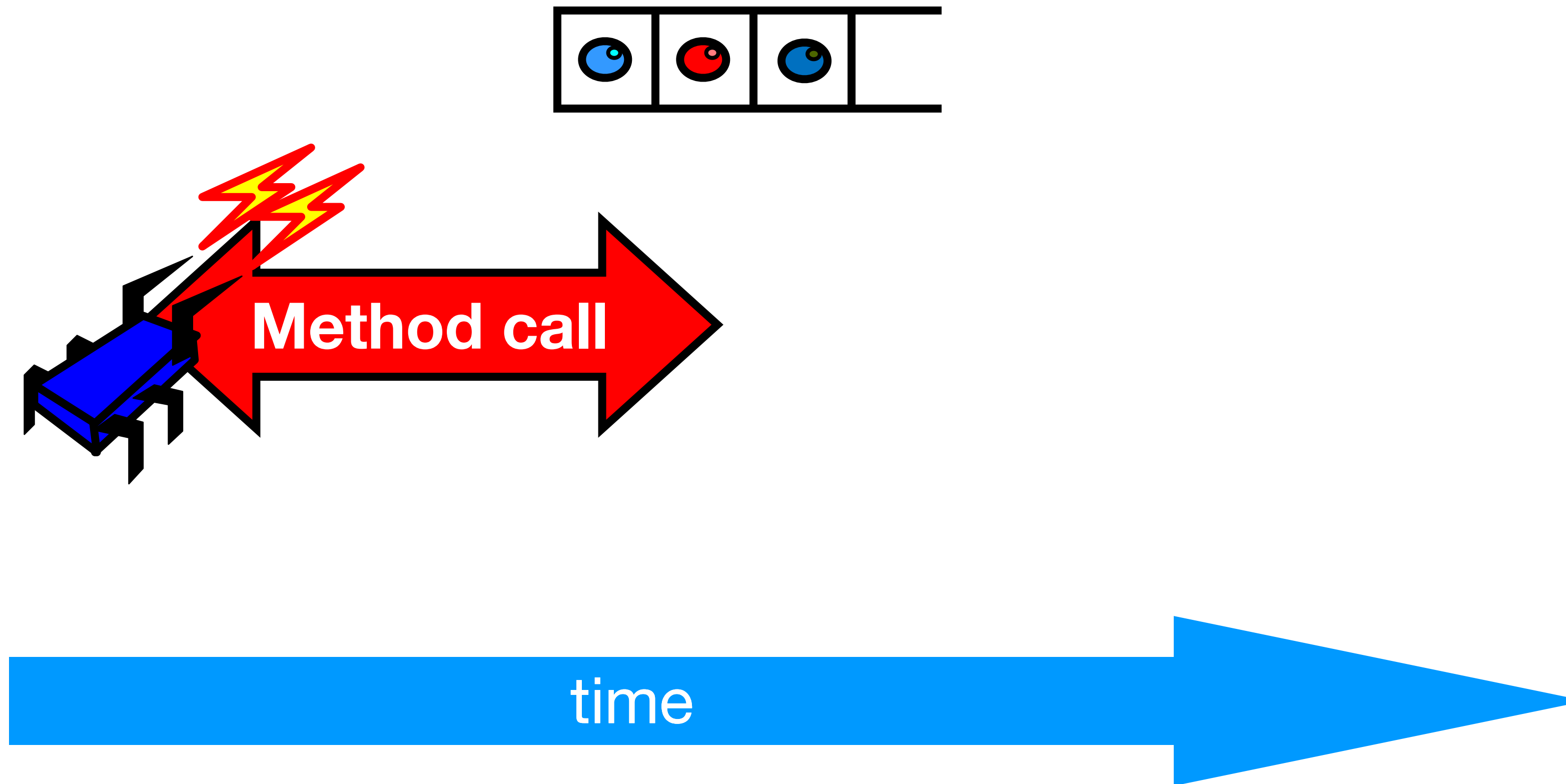
time

Concurrent Methods Take Overlapping Time

Method call

time

Concurrent Methods Take Overlapping Time

Method call

Method call

time

Concurrent Methods Take Overlapping Time
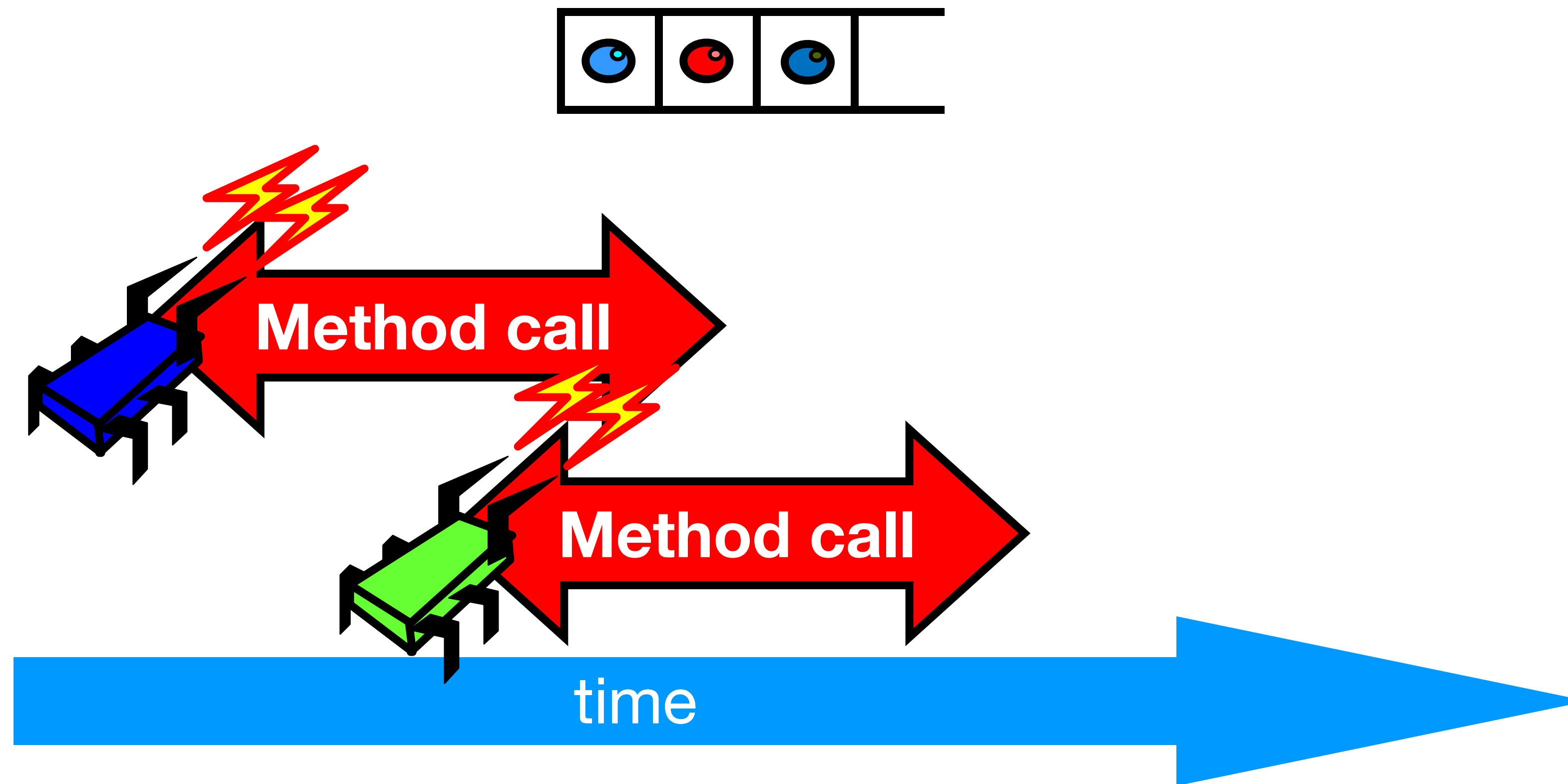
Method call

Method call

Method call

time
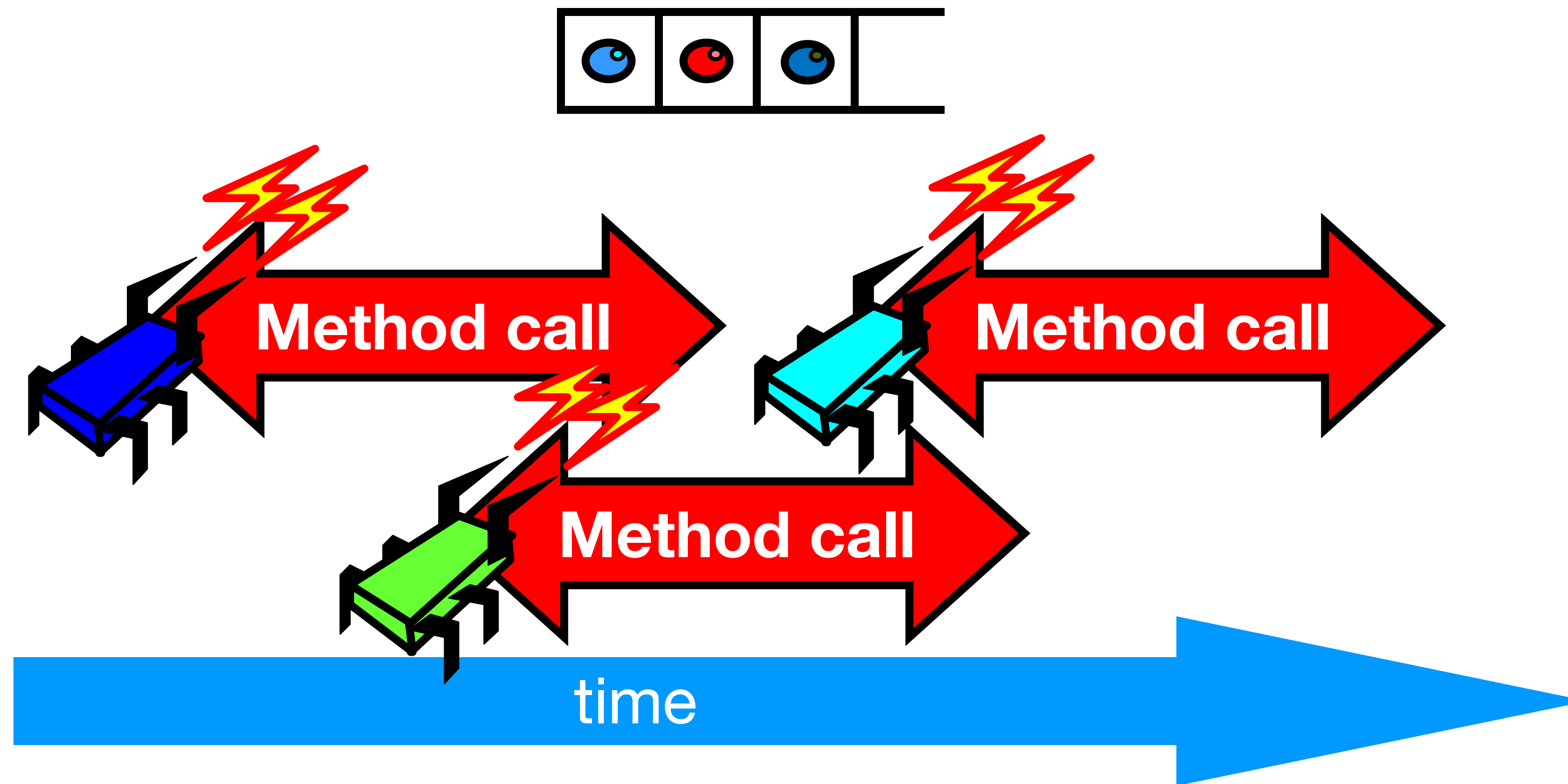
Sequential vs Concurrent

- Sequential

  - Object needs a meaningful state only **between** method calls

- Concurrent

  - Because method calls overlap, the 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

```
let deq q =
  Mutex.lock q.lock;
  try
    if q.tail = q.head then
      raise Empty;
    match q.items.(q.head mod q.capacity) with
    | None -> assert false
    | Some x ->
        q.head <- q.head + 1;
        Mutex.unlock q.lock;
        x
  with e ->
    Mutex.unlock q.lock;
    raise e
```

All queue modifications are mutually exclusive

Intuitively

Intuitively

Intuitively

Intuitively


```
              q.deq
      lock()              unlock()
◀━━━━━━━━━━━━━━━━━━━━━━▶  ◀━━▶ ◀━━▶
                          deq

              q.enq
      ◀━━▶ ◀━━▶ ◀━━▶
      lock()  enq  unlock()
```

time
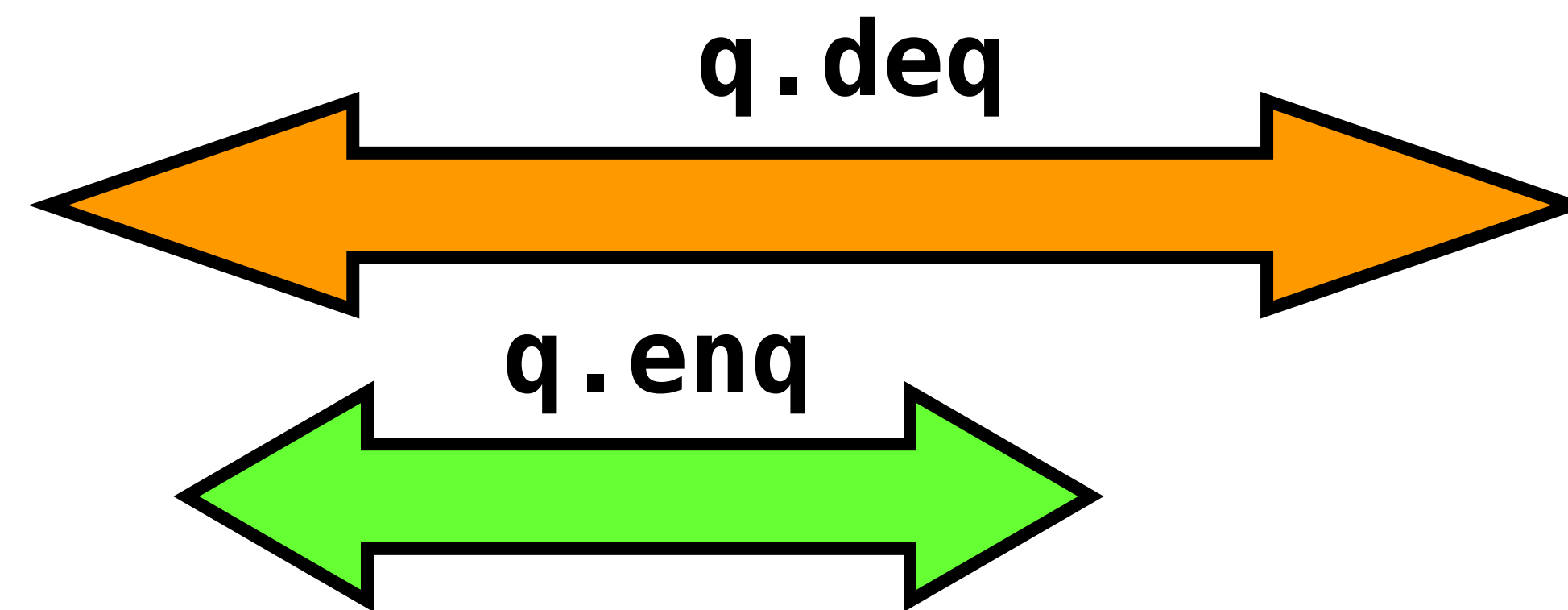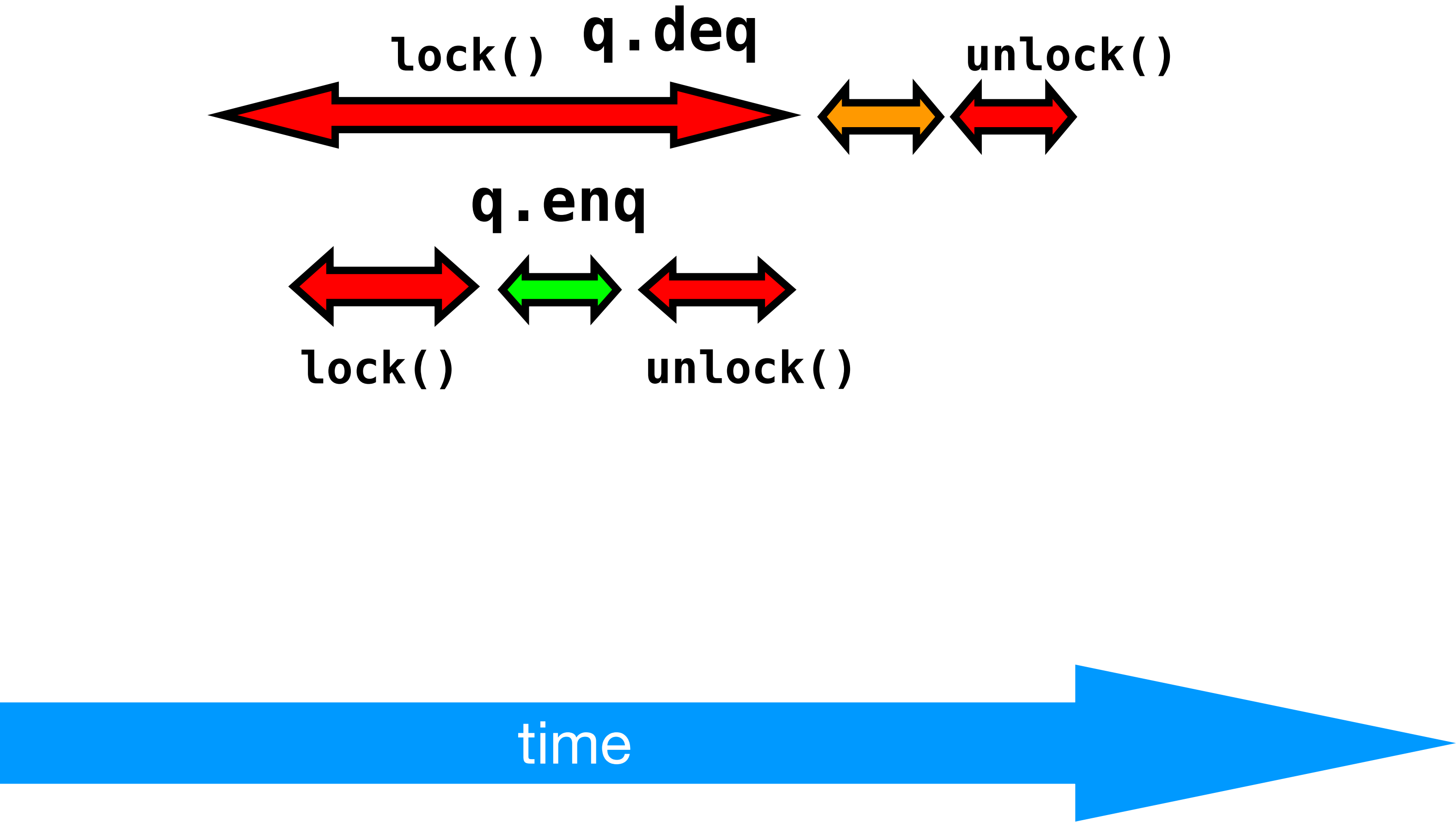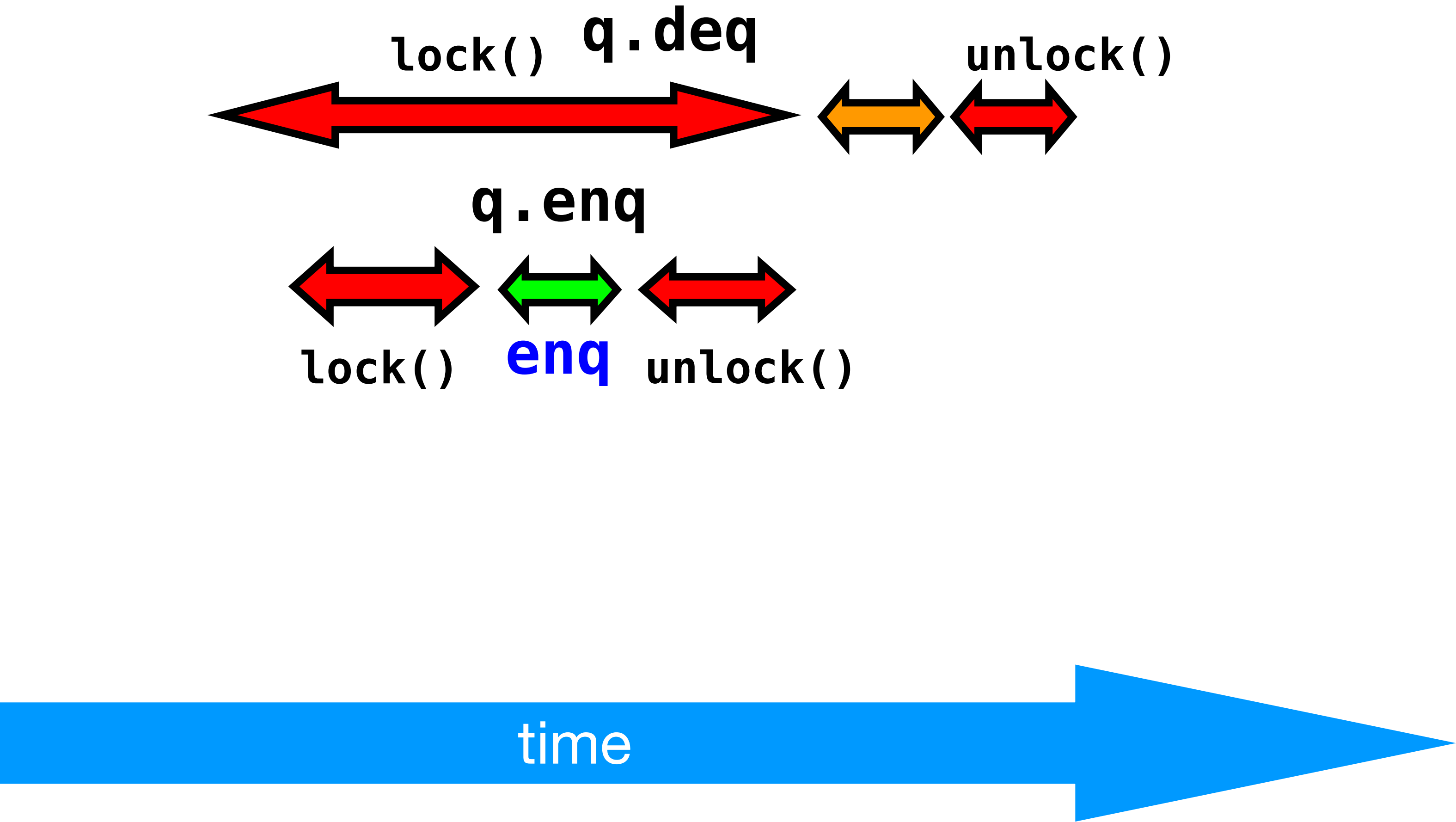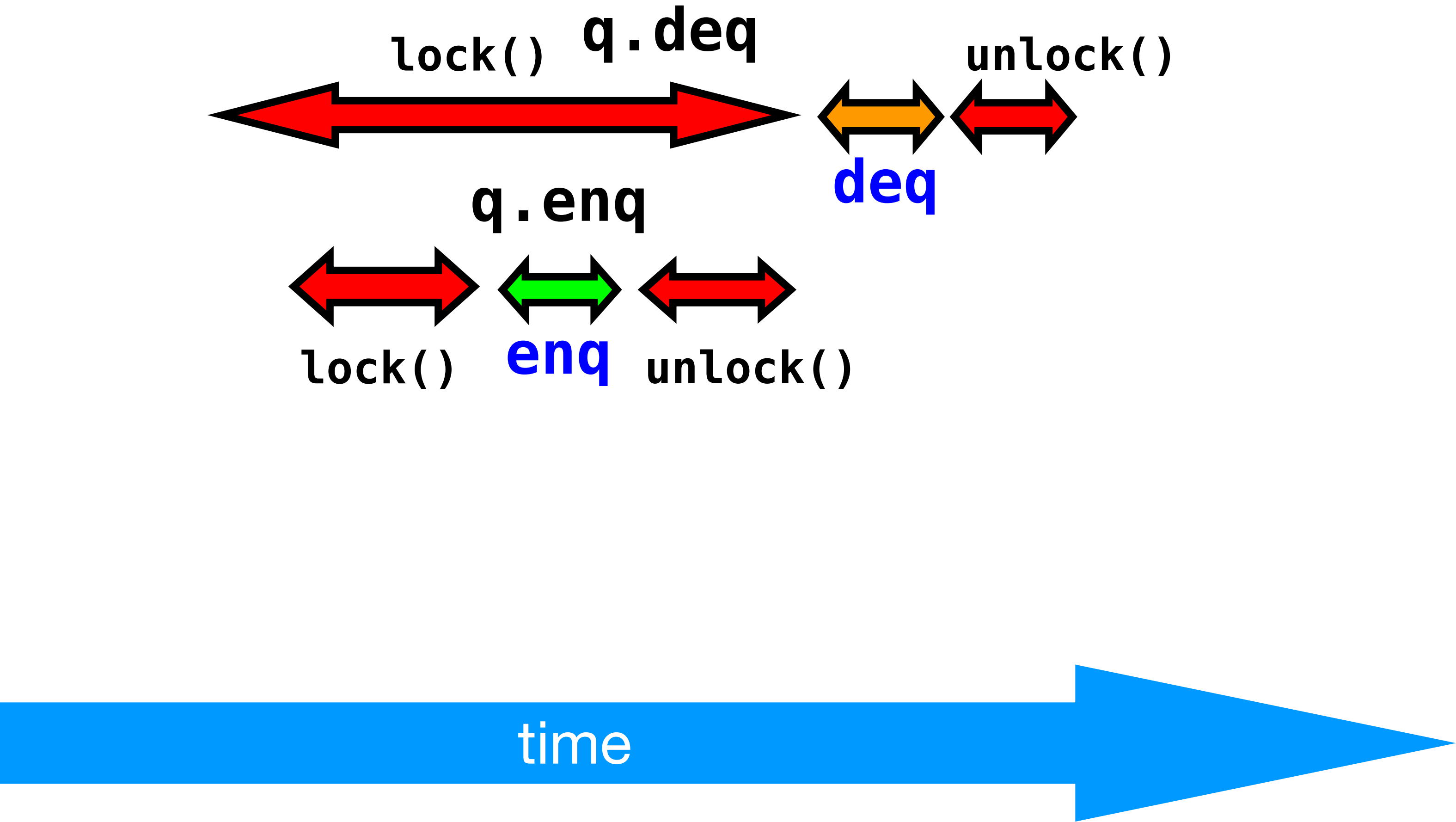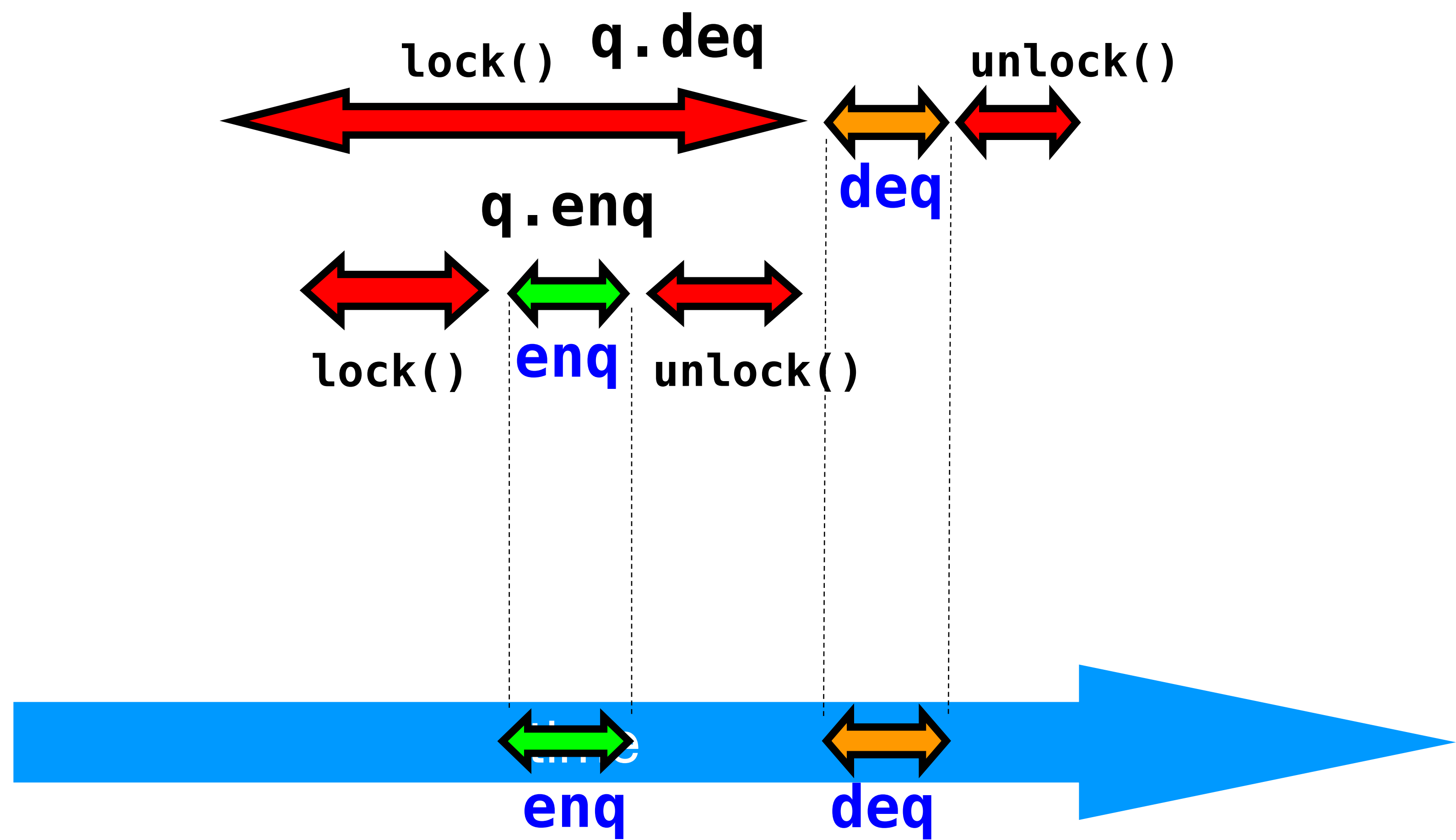
Intuitively
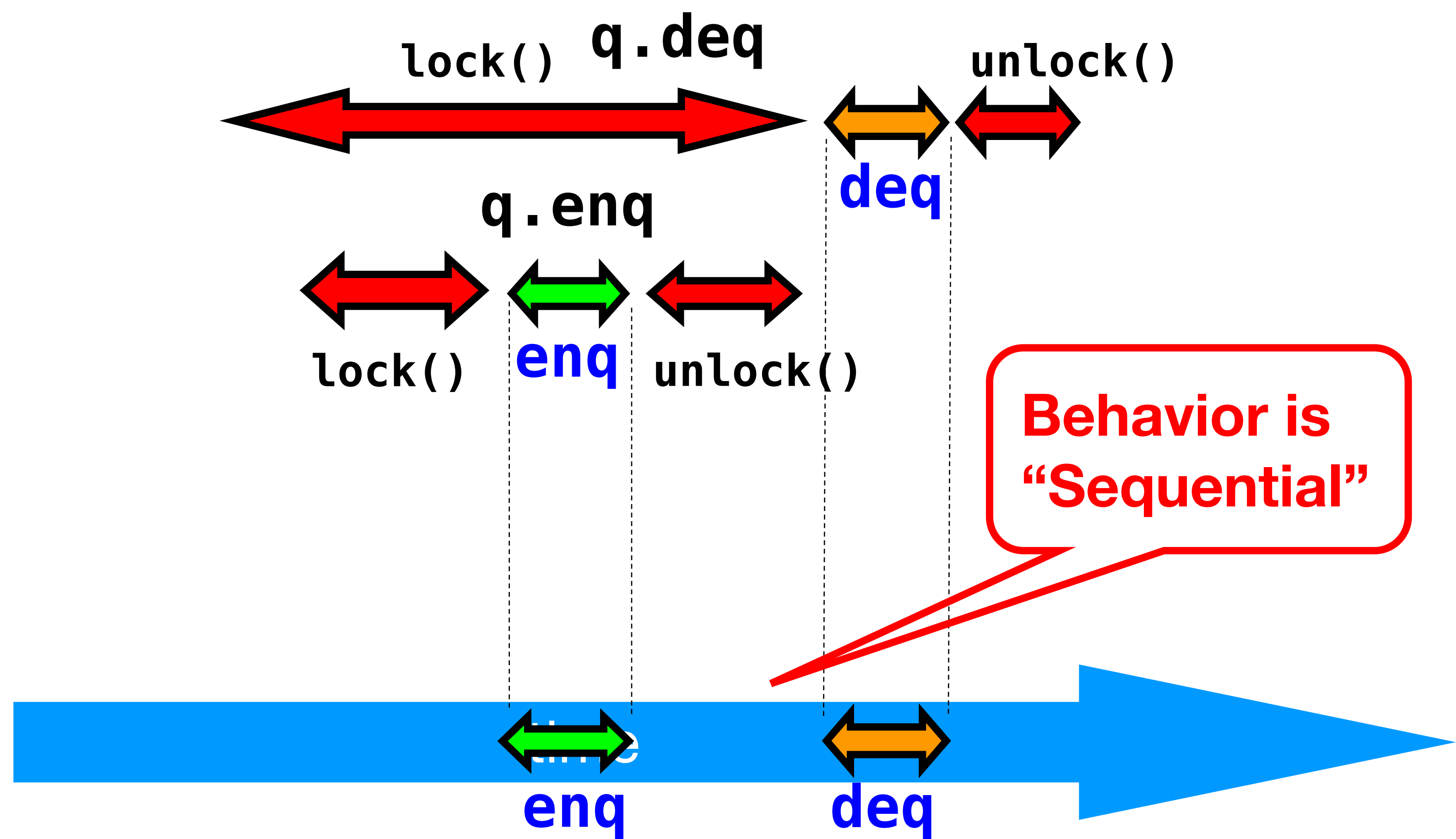
Intuitively

Intuitively

Lets capture the idea of describing the concurrent via the sequential

`lock()` **q.deq** `unlock()`

deq

q.enq

`lock()` **enq** `unlock()`

Behavior is "Sequential"

enq **deq**
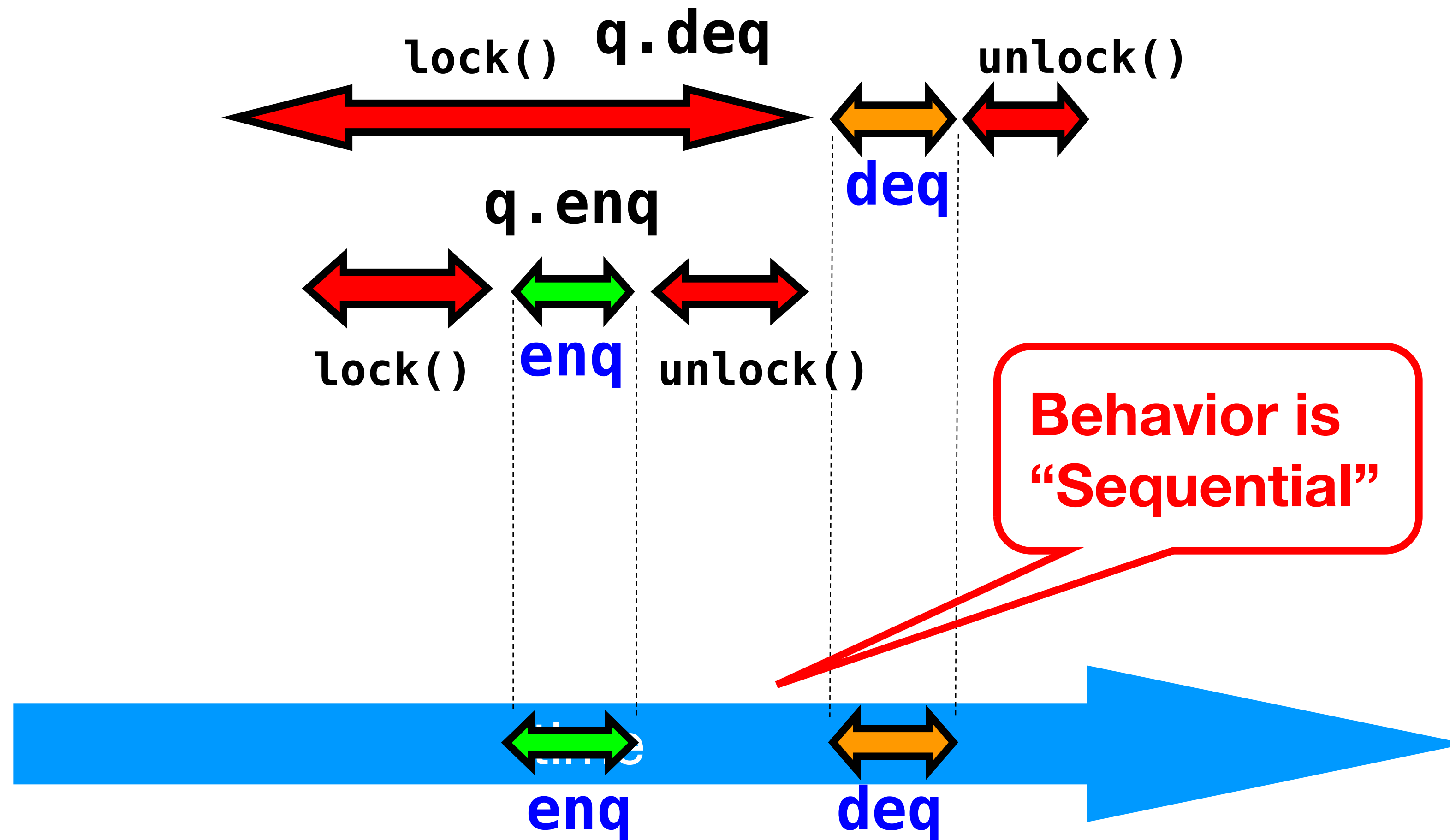
Linearizability

Linearizability

- Each method should

  - "take effect"

  - Instantaneously

  - Between *invocation* and *response* events

Linearizability

- Each method should

  - "take effect"

  - Instantaneously

  - Between *invocation* and *response* events

- Object is correct if this "sequentialised" behaviour is correct
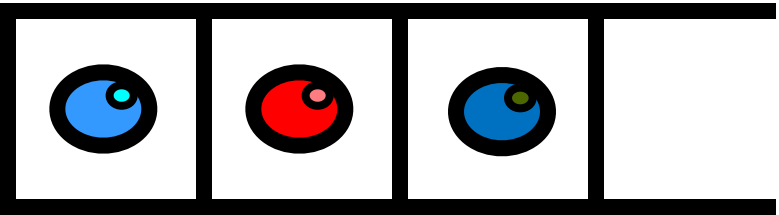
Linearizability

- Each method should

  - "take effect"

  - Instantaneously

  - Between *invocation* and *response* events

- Object is correct if this "sequentialised" behaviour is correct

- Any such concurrent object is

  - **Linearizable™**

Is it really about the object?

- Each method should

  - "take effect"

  - Instantaneously

  - Between invocation and response events

- Sounds like a property of *an execution…*

- A linearizable *object*

  - One of whose all possible executions are linearizable

Example
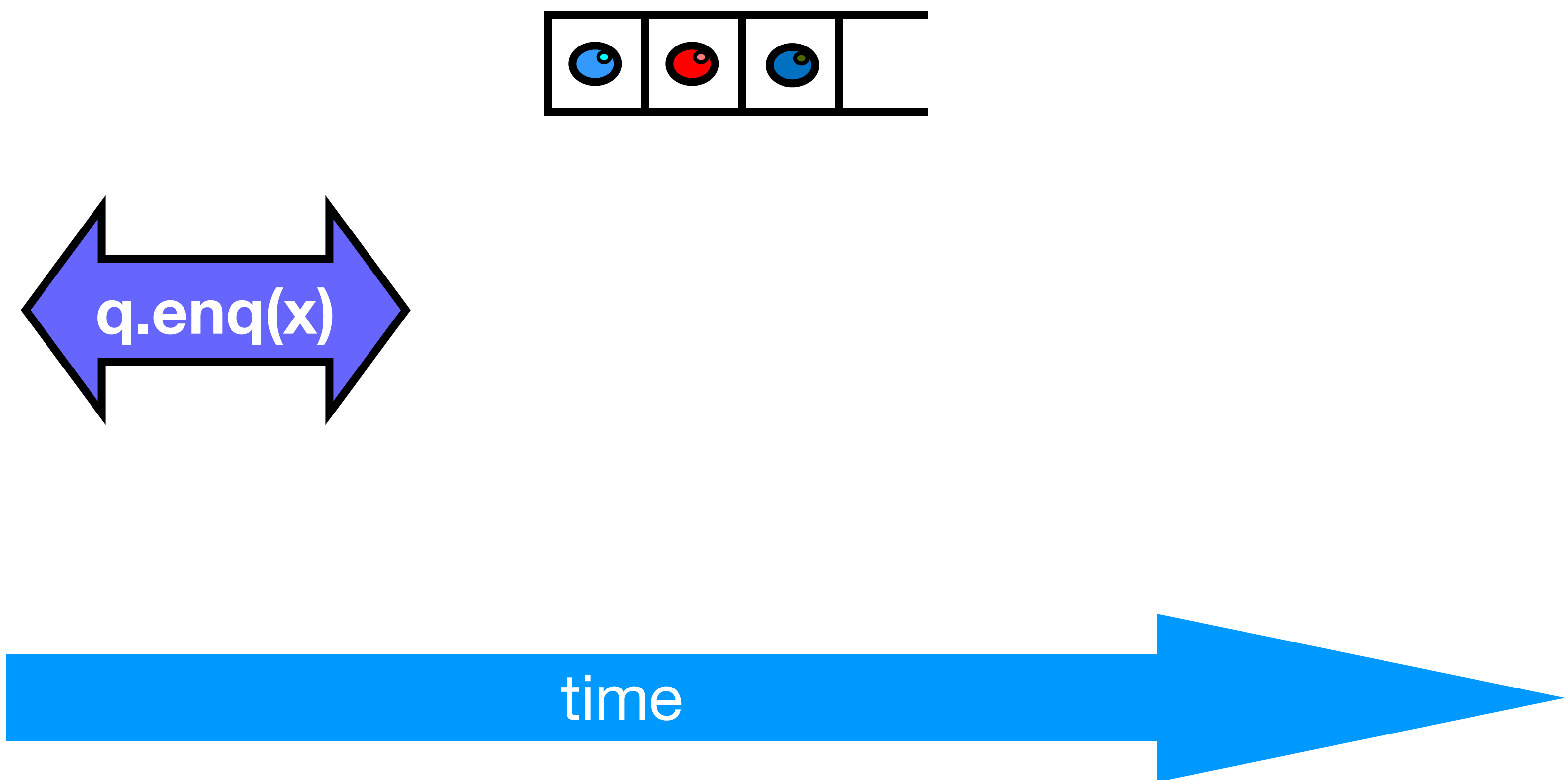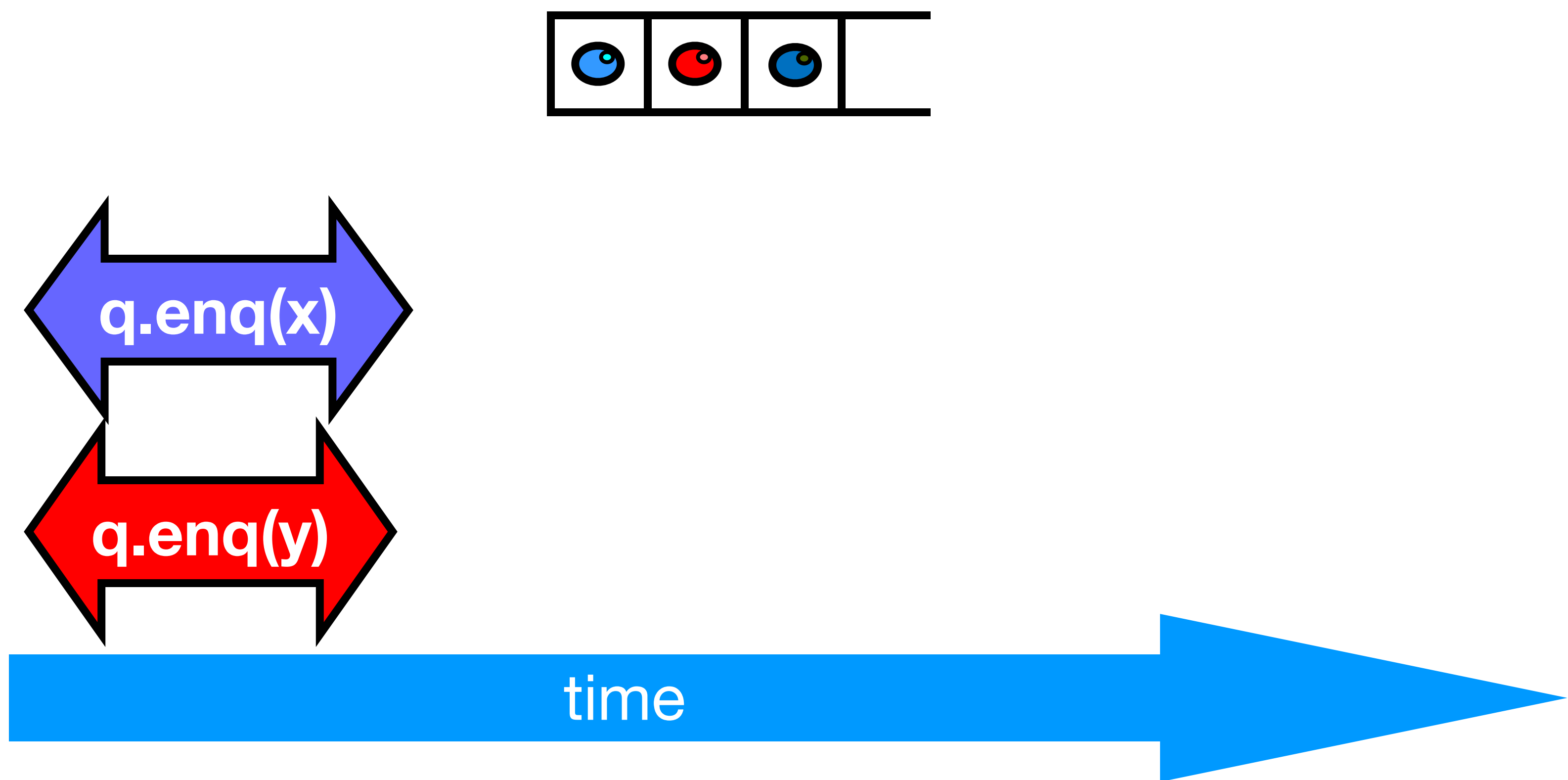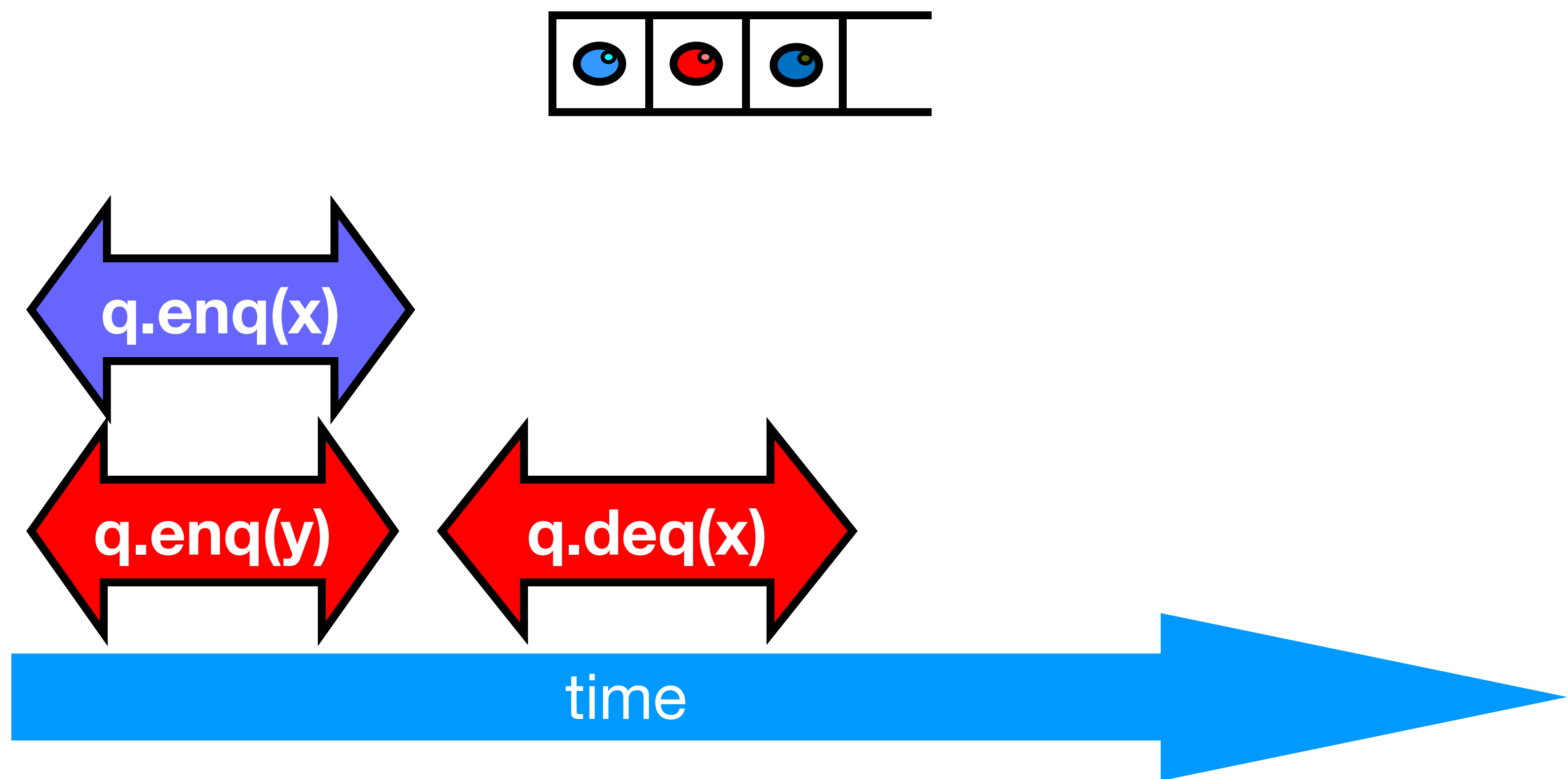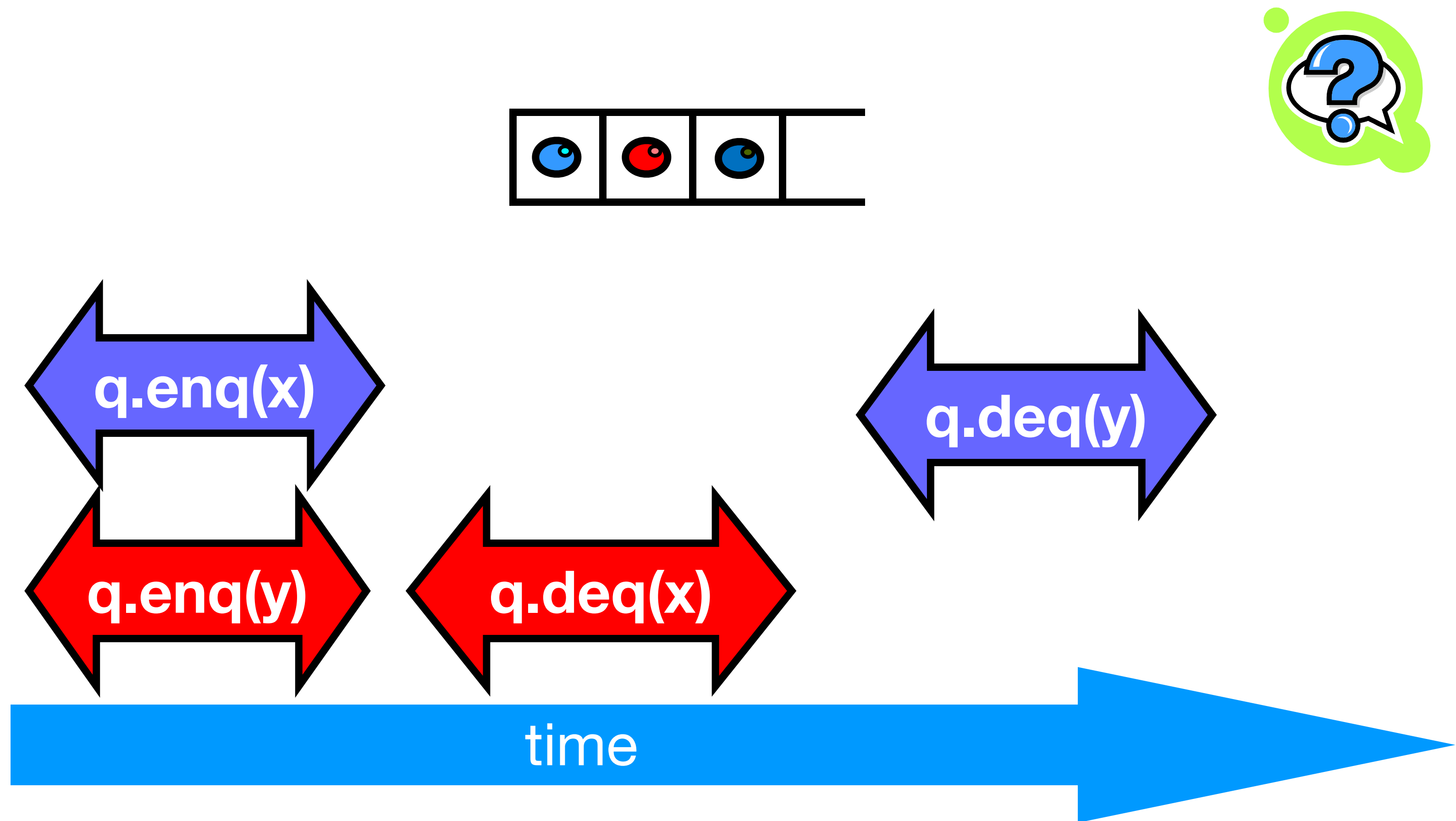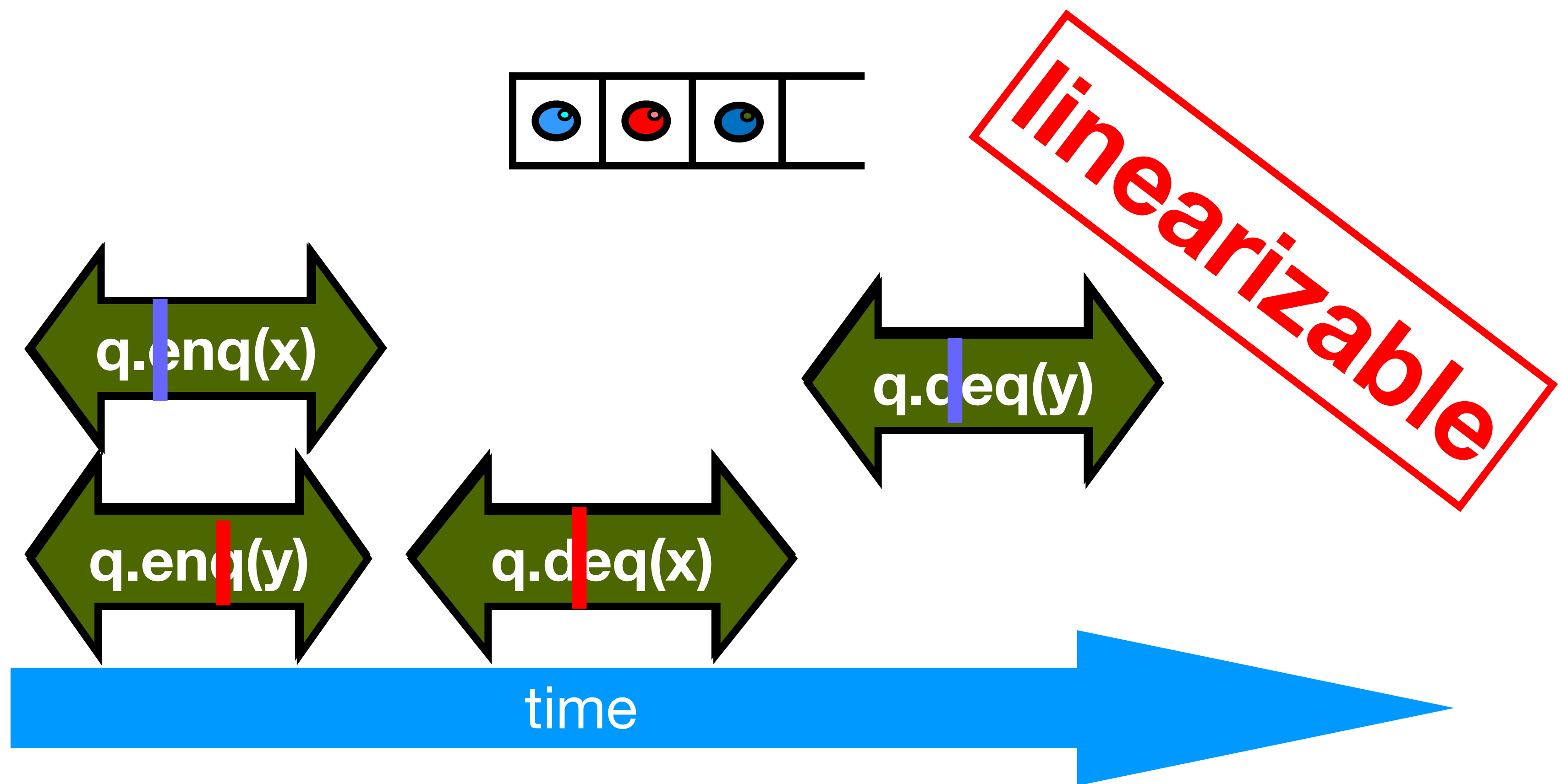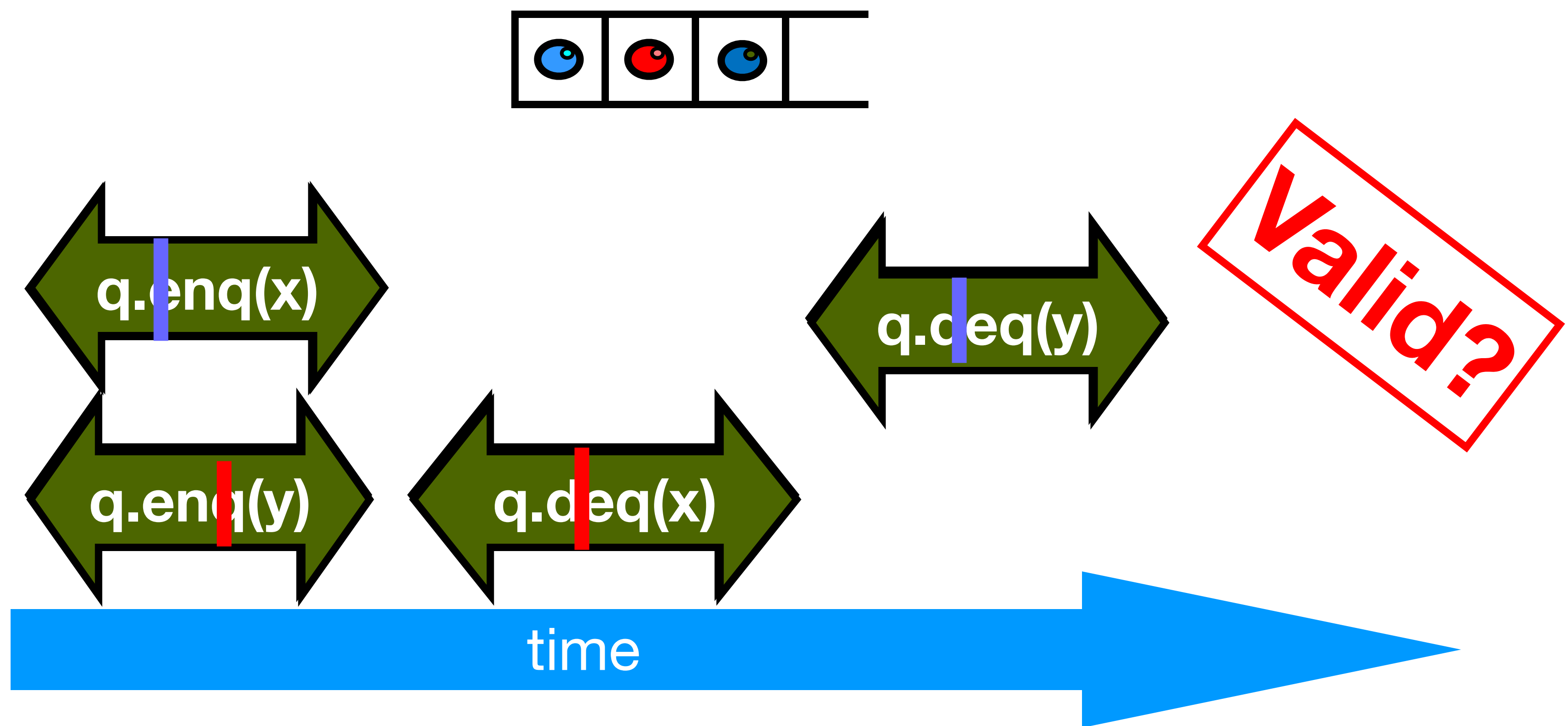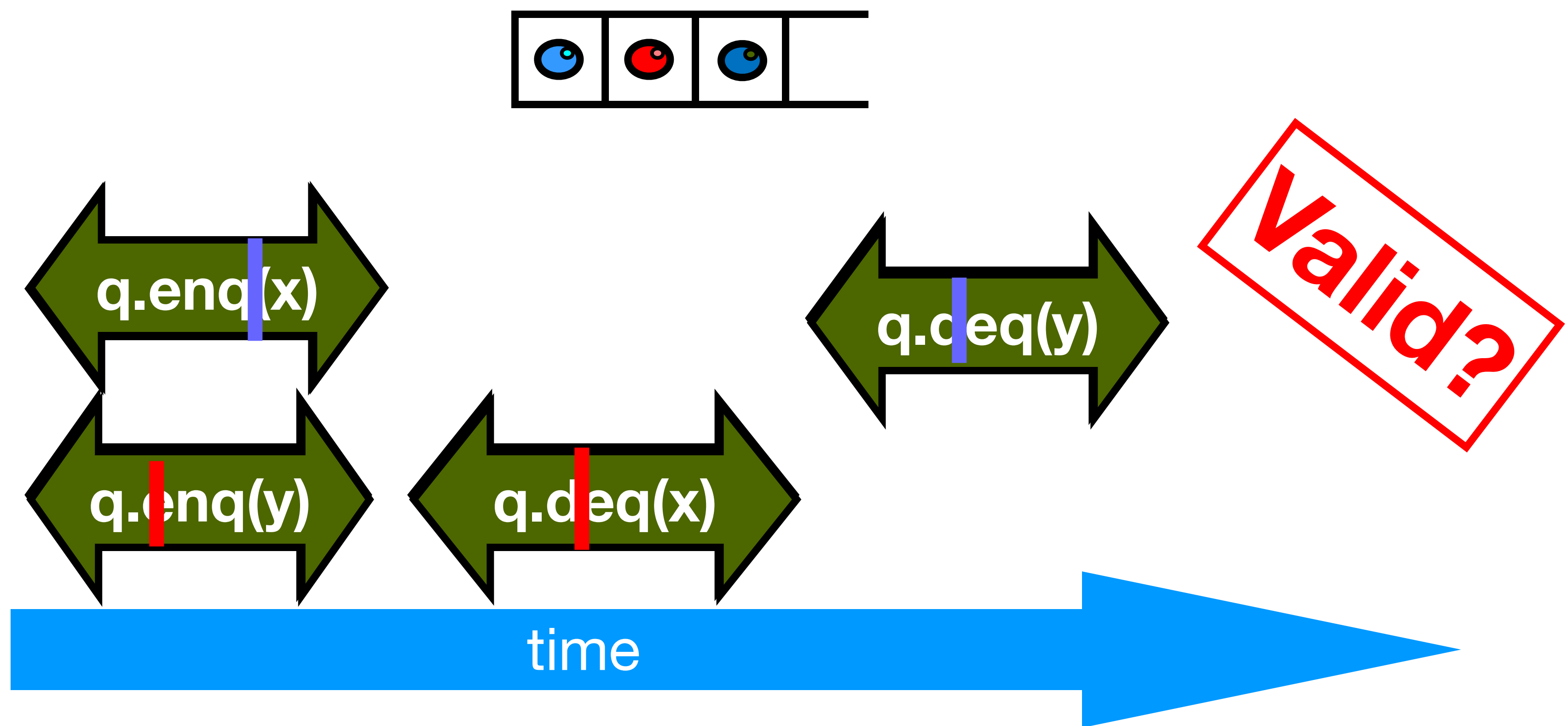
time

Example

q.enq(x)

time

Example

q.enq(x)

q.enq(y)

time

Example

Example

Example

Example

Example

Example

Example

q.enq(x)

time

Example

q.enq(x)

q.deq(y)

time

Example

Example

q.enq(x)

q.deq(y)

q.enq(y)

time

Example

q.enq(x)

q.deq(y)

q.enq(y)

time

Example

Example

Example

time

Example

q.enq(x)

time

Example

Example

Example

q.enq(x)

q.deq(x)

linearizable

time

Example

q.enq(x)

time

Example

Example

Example

q.enq(x)

q.enq(y)

q.deq(y)

q.deq(x)

time
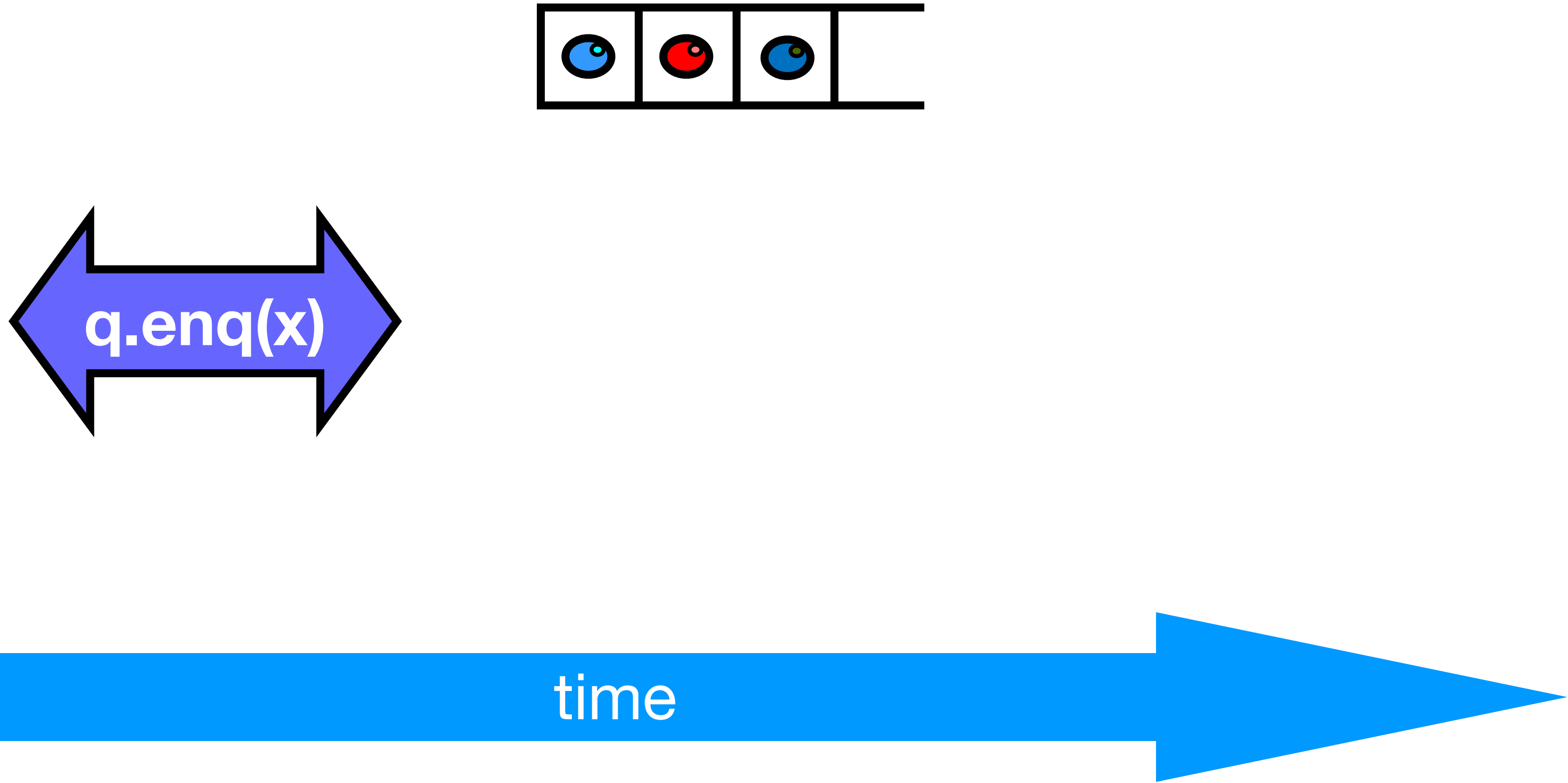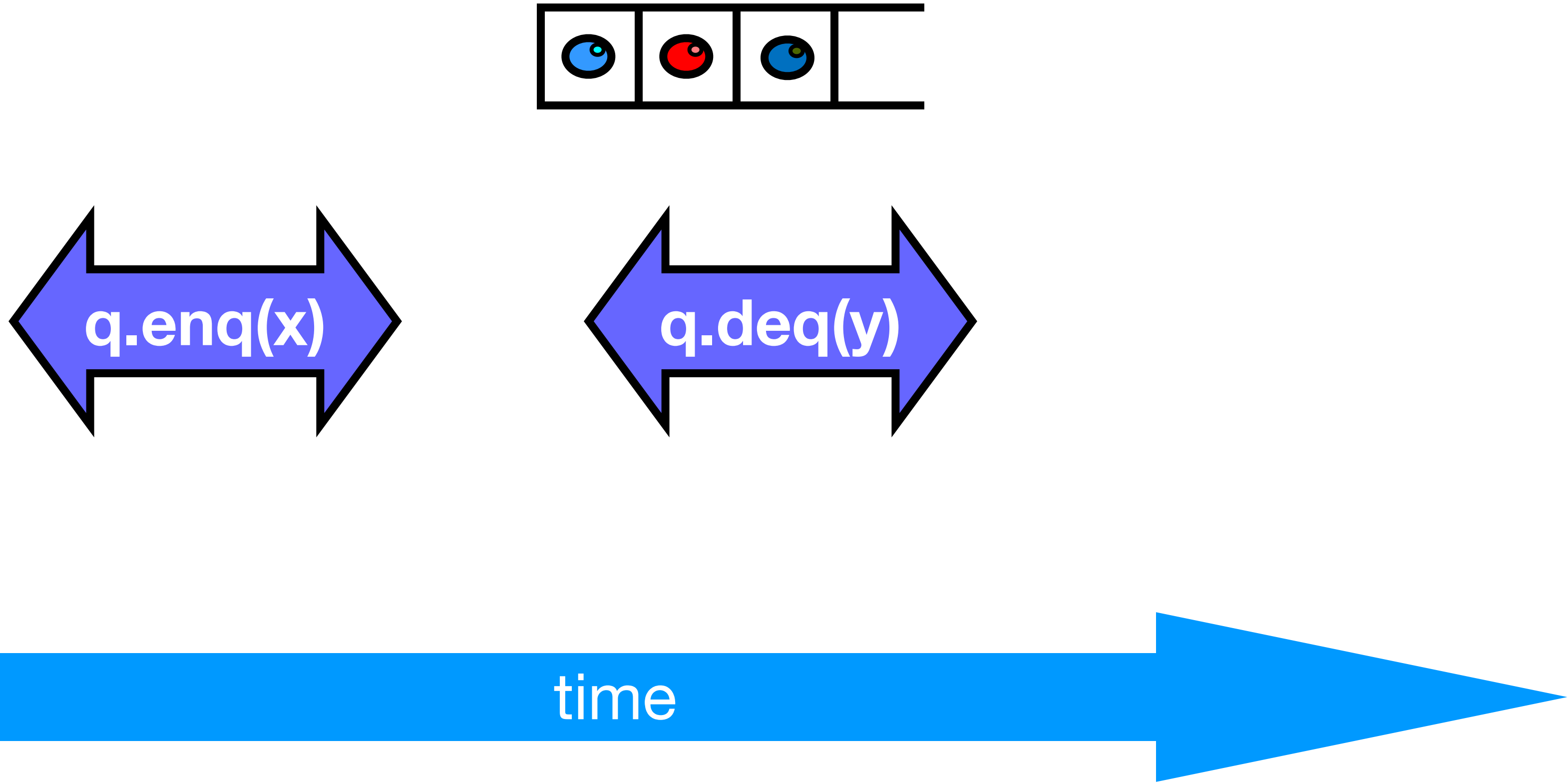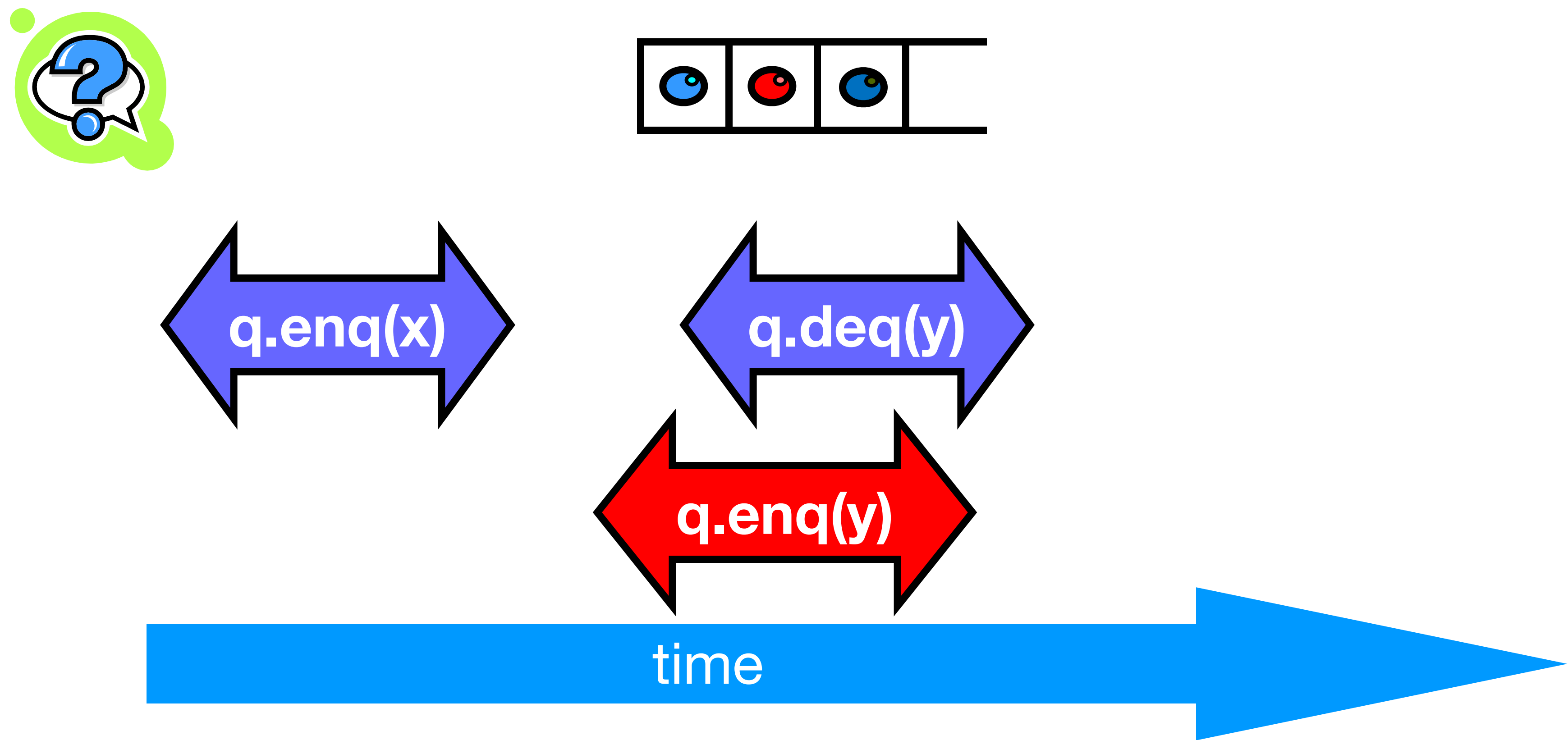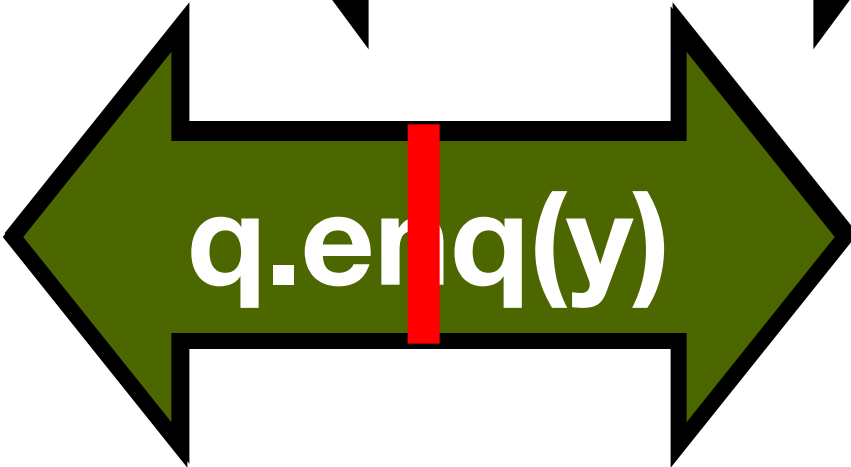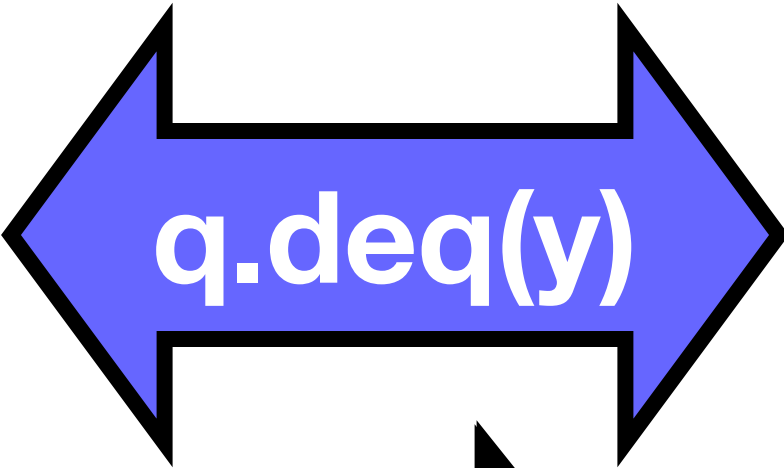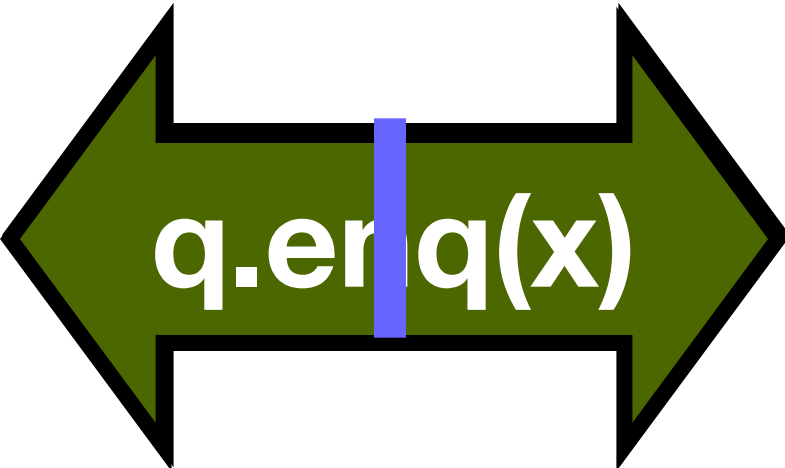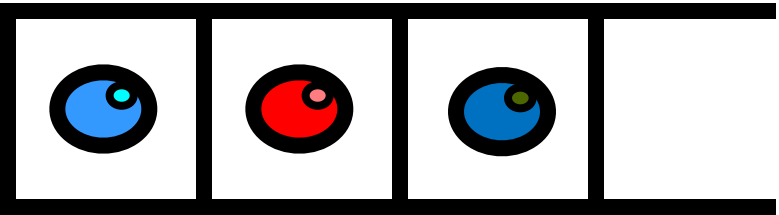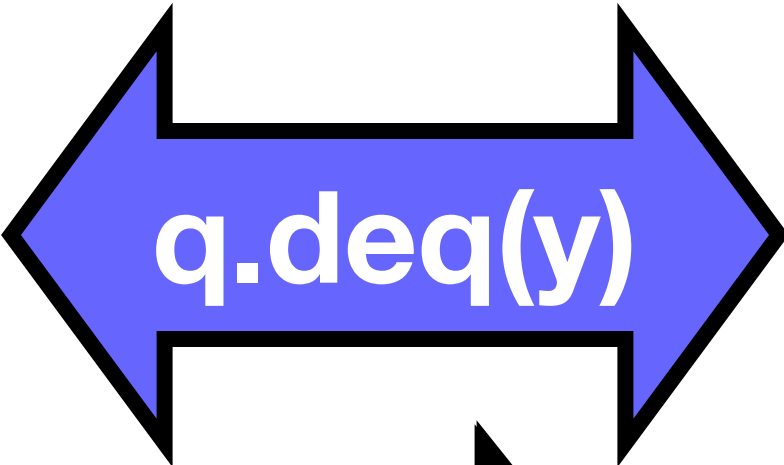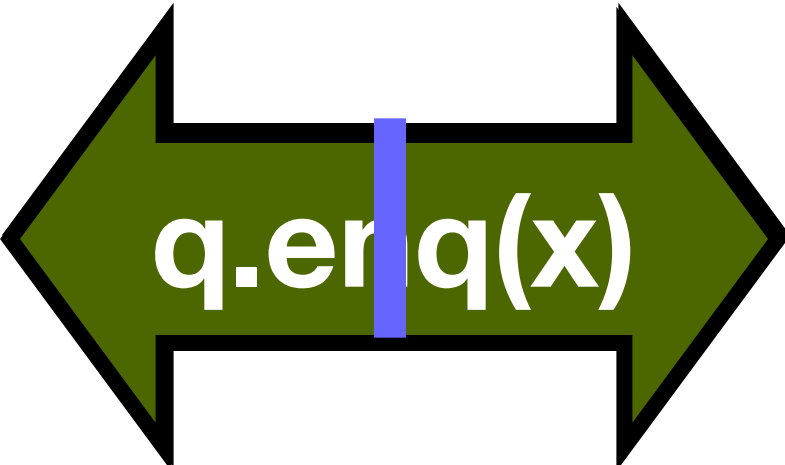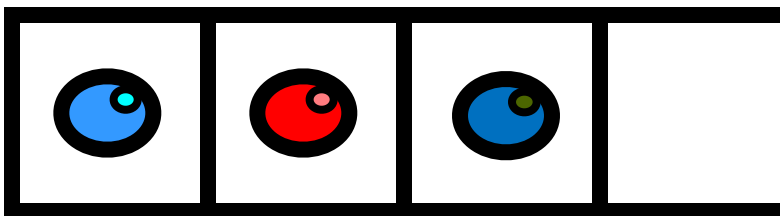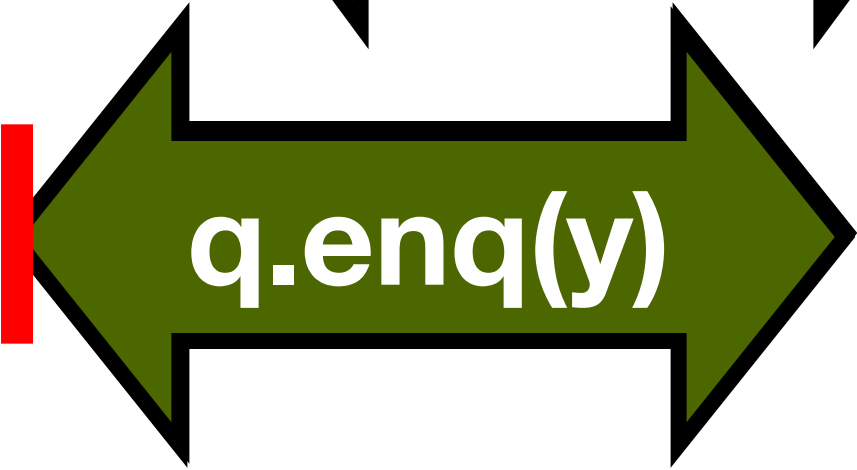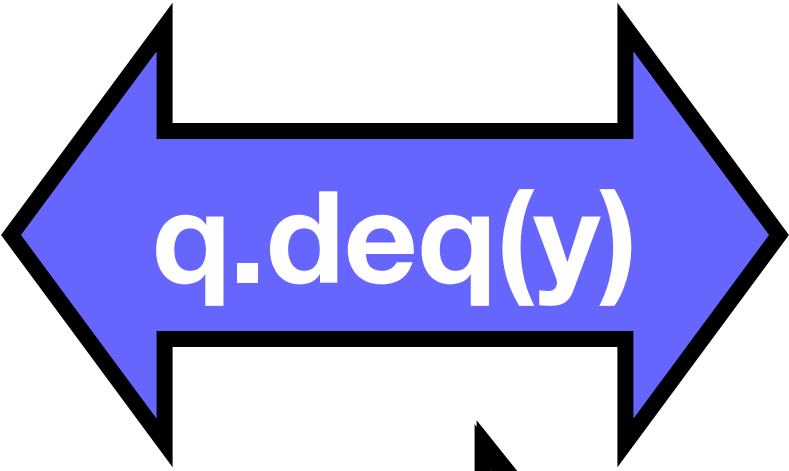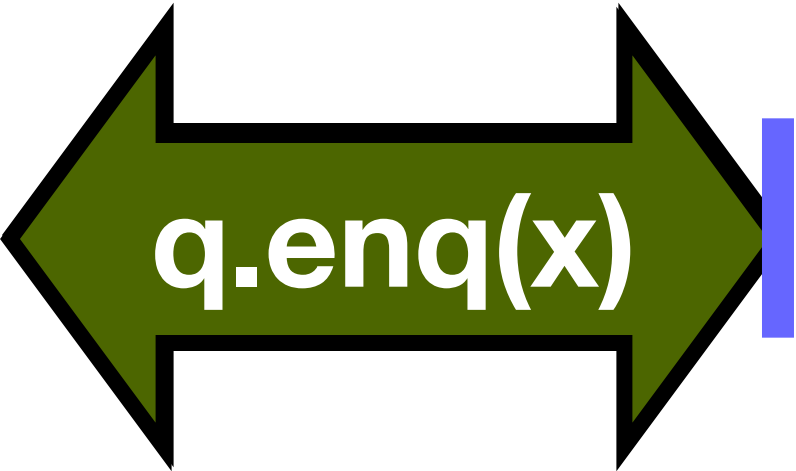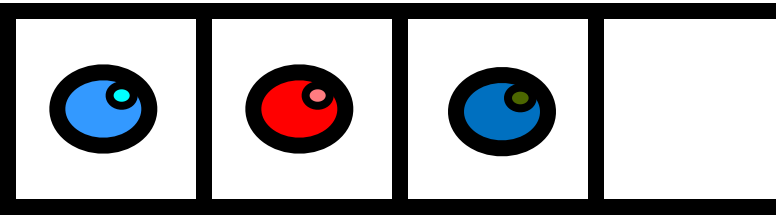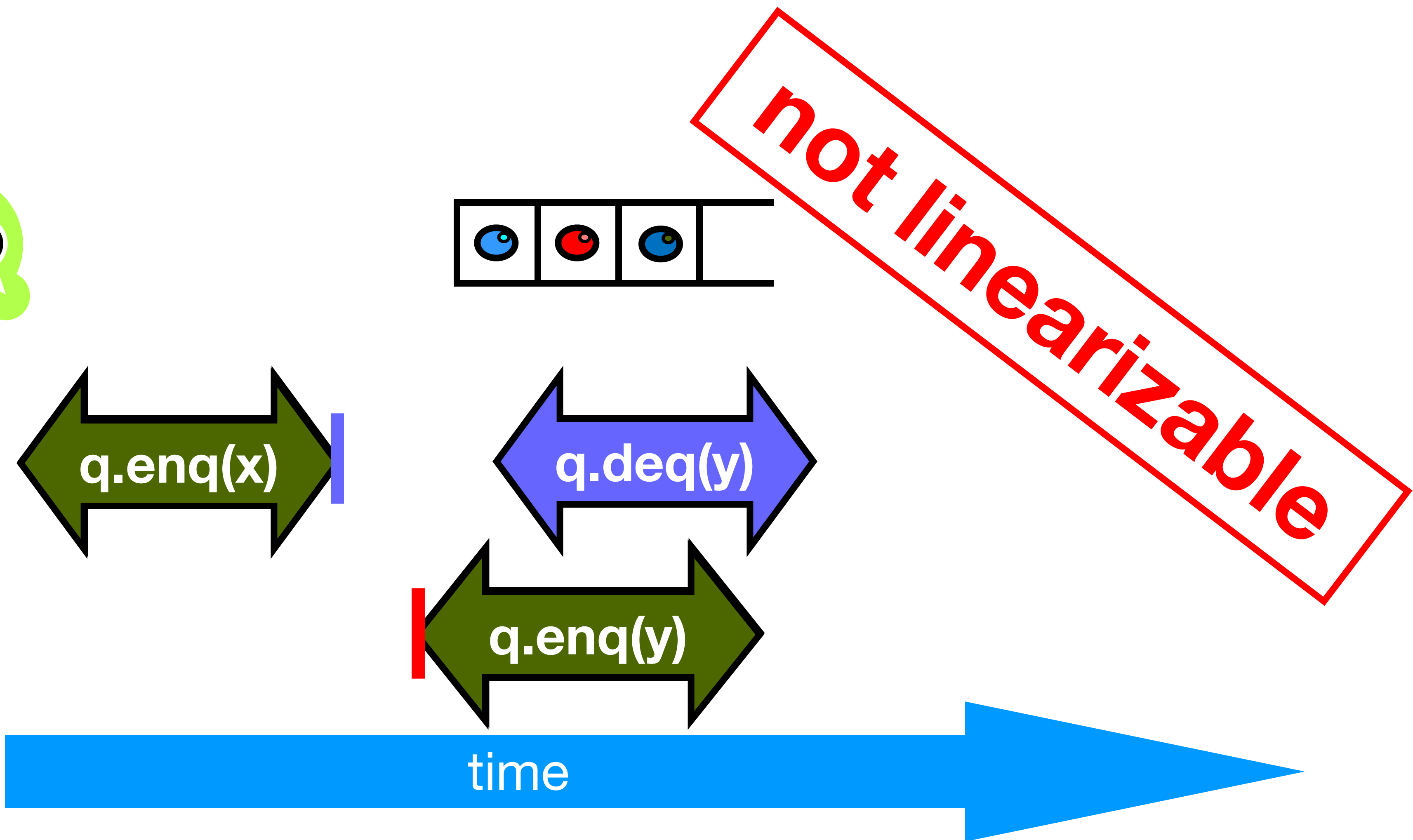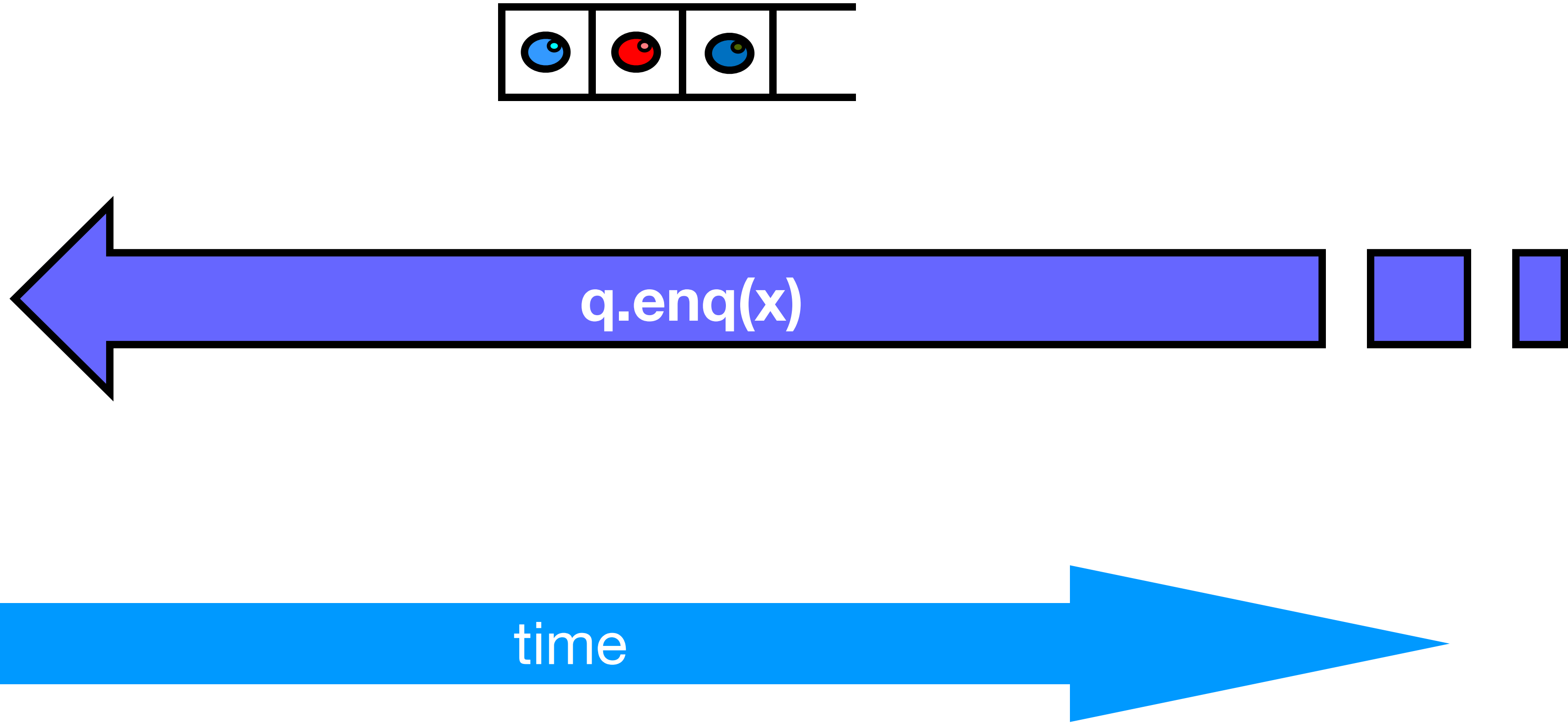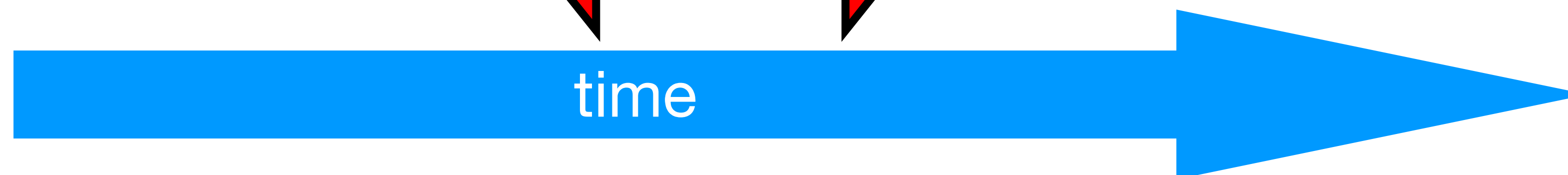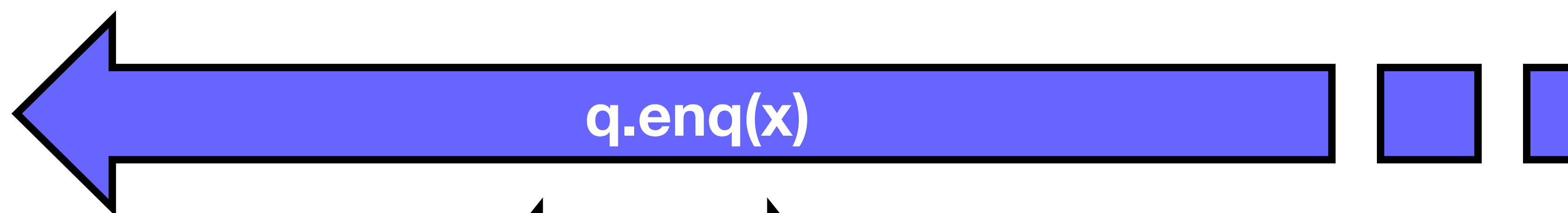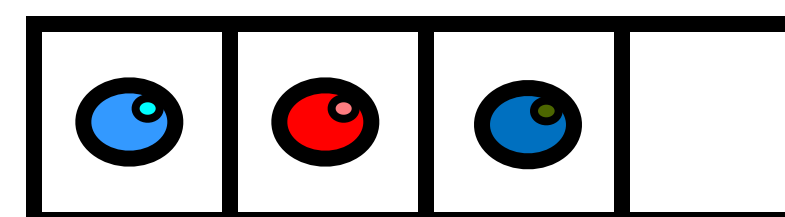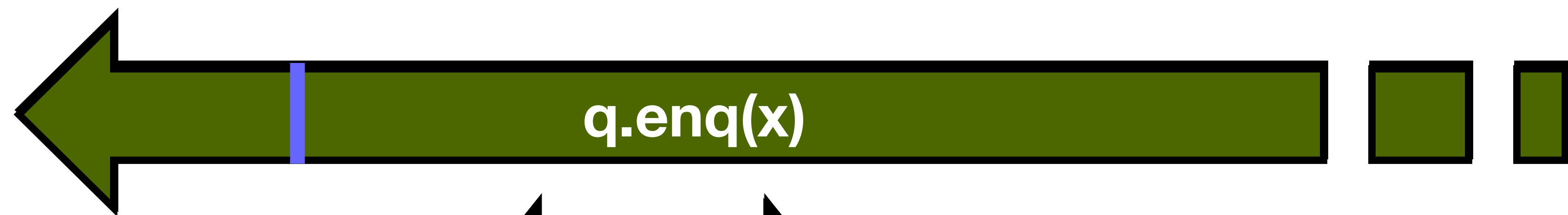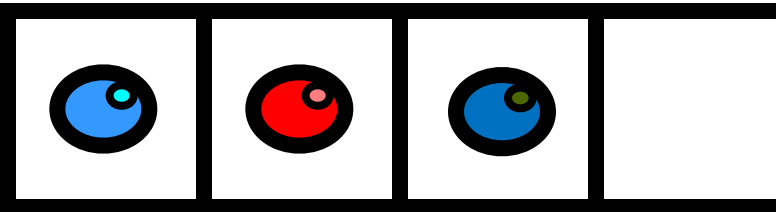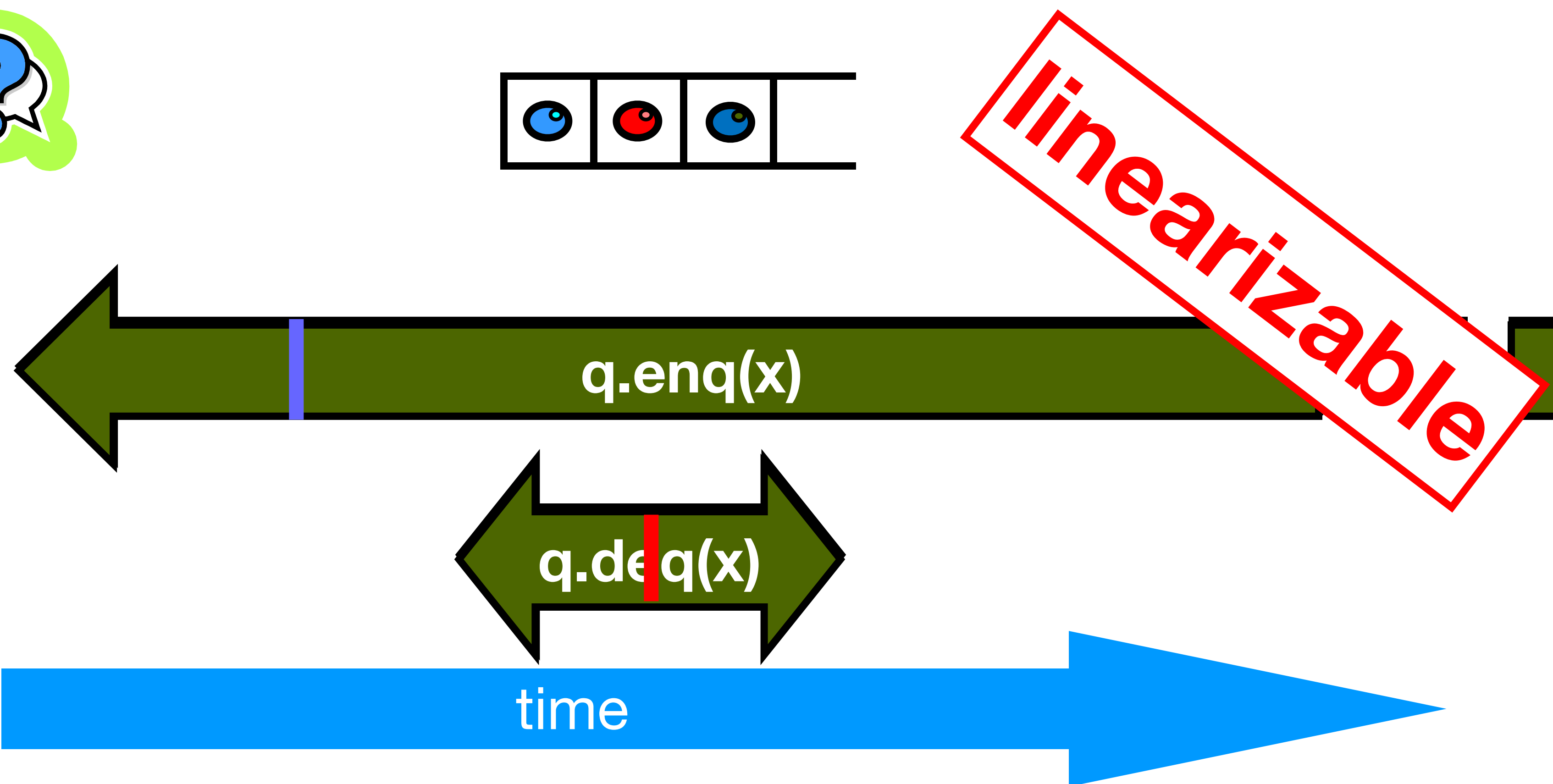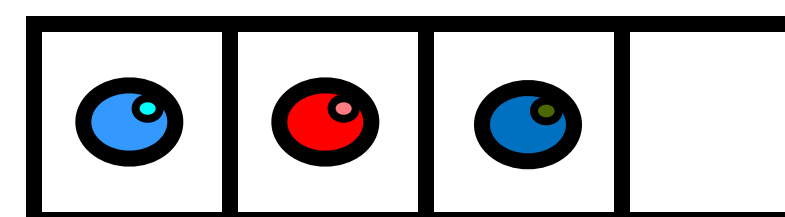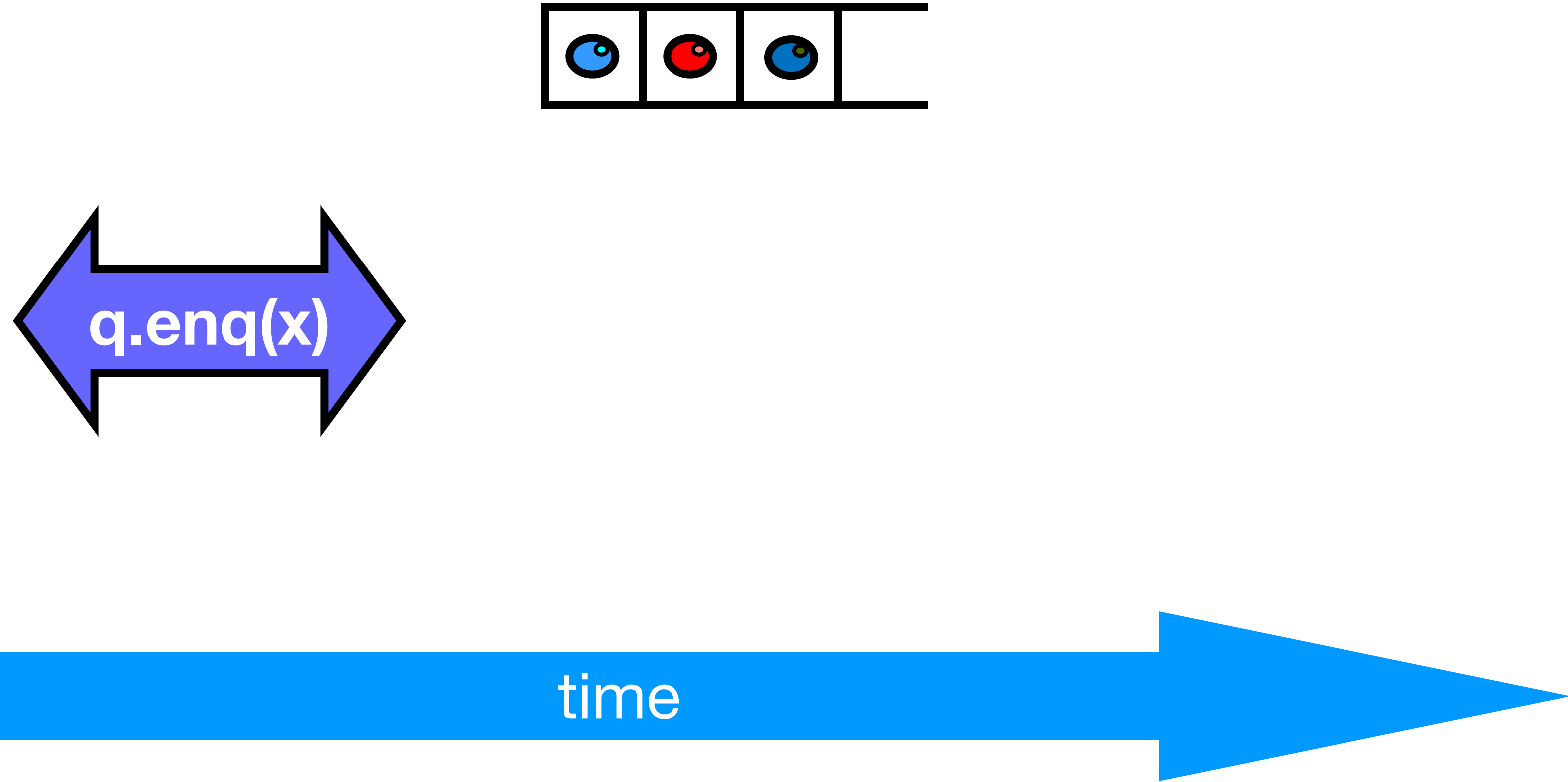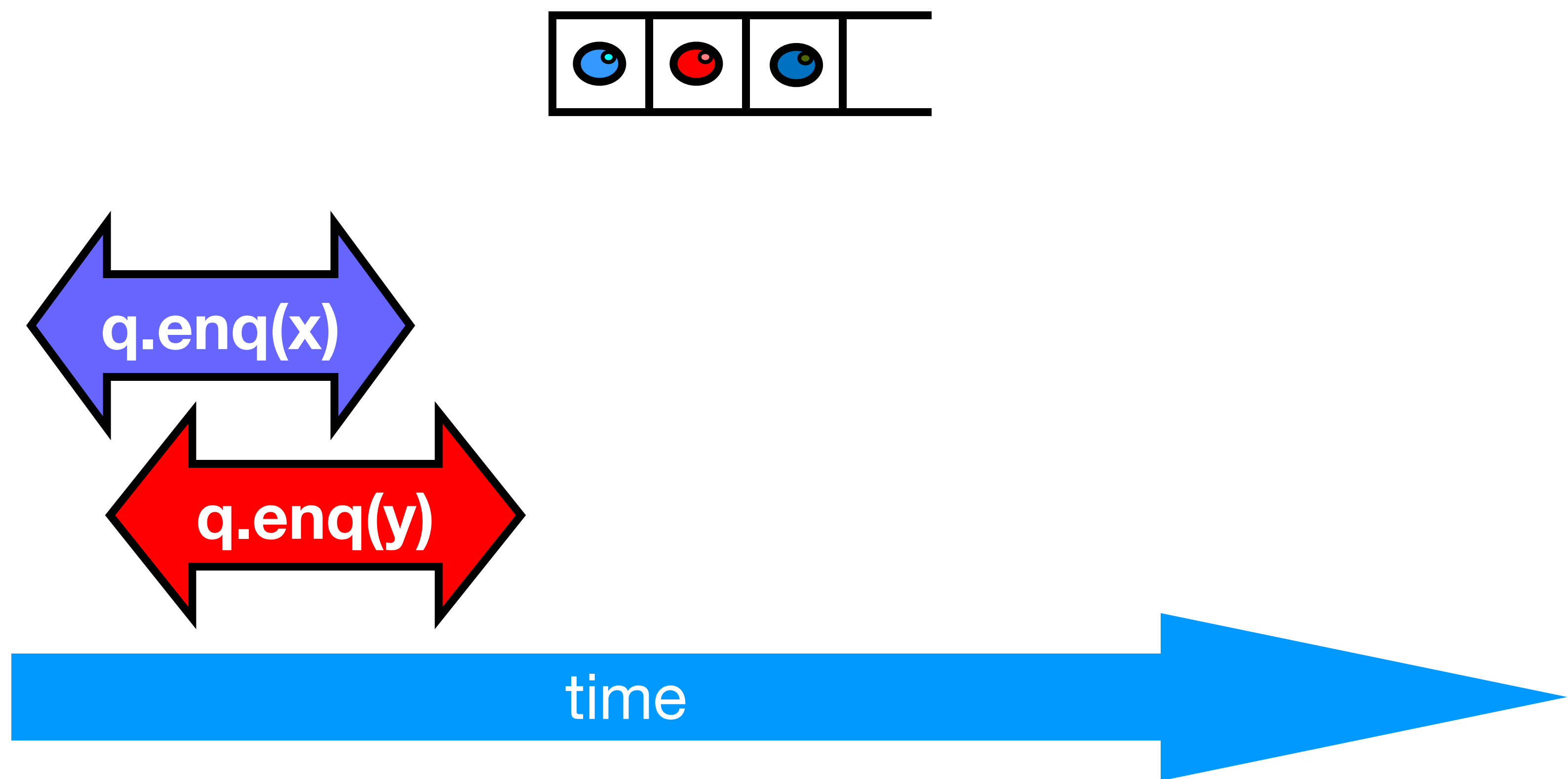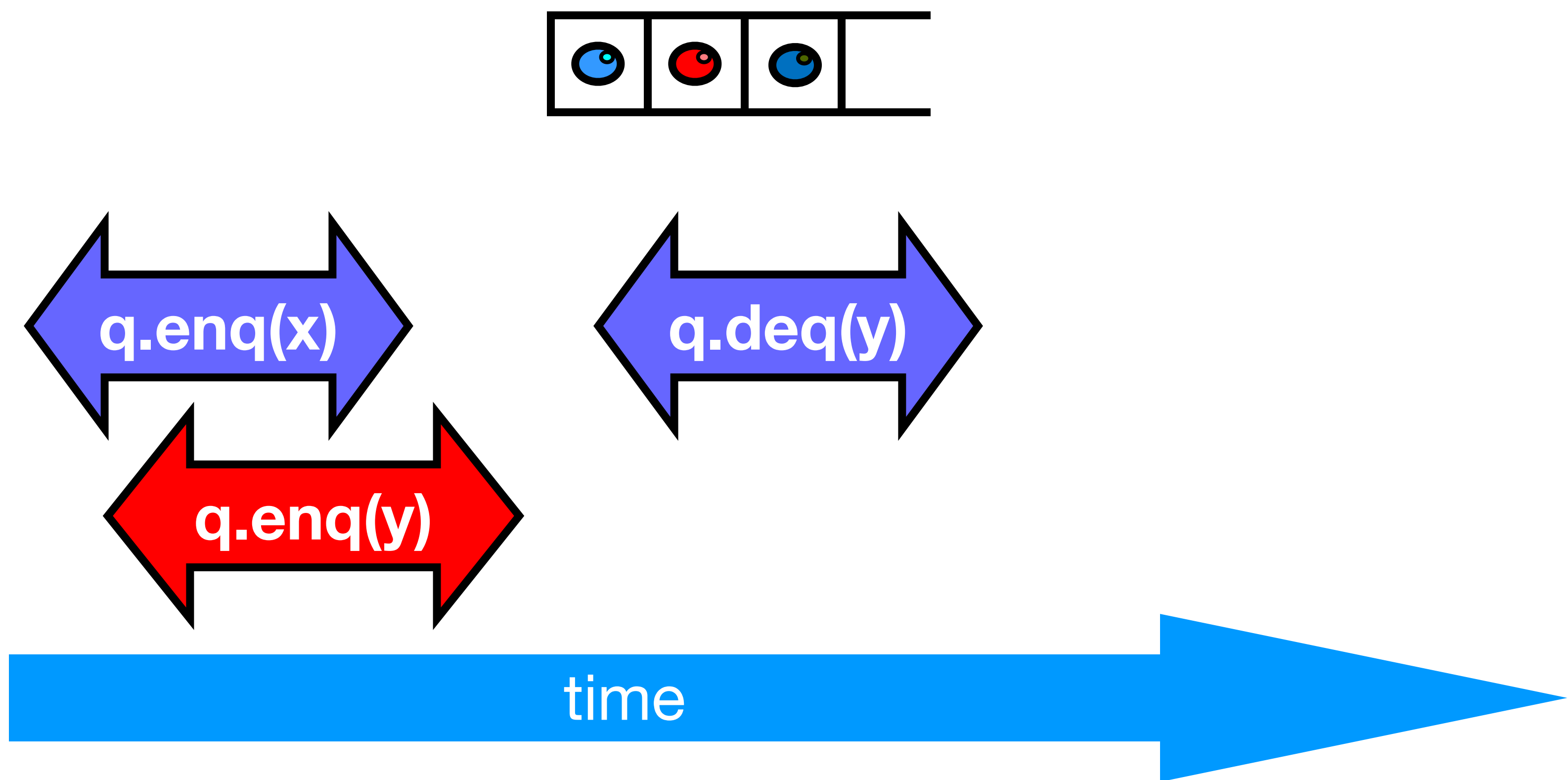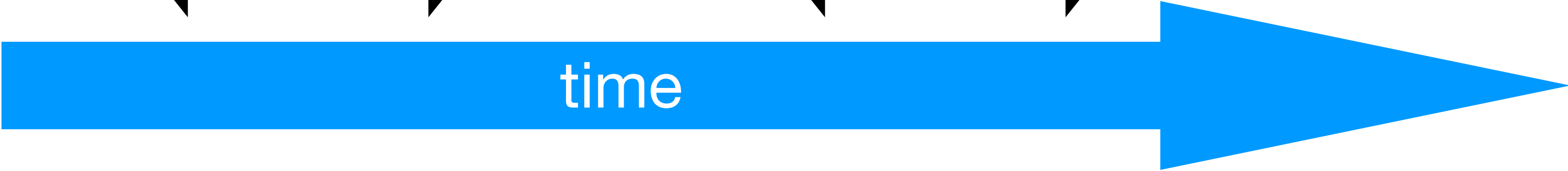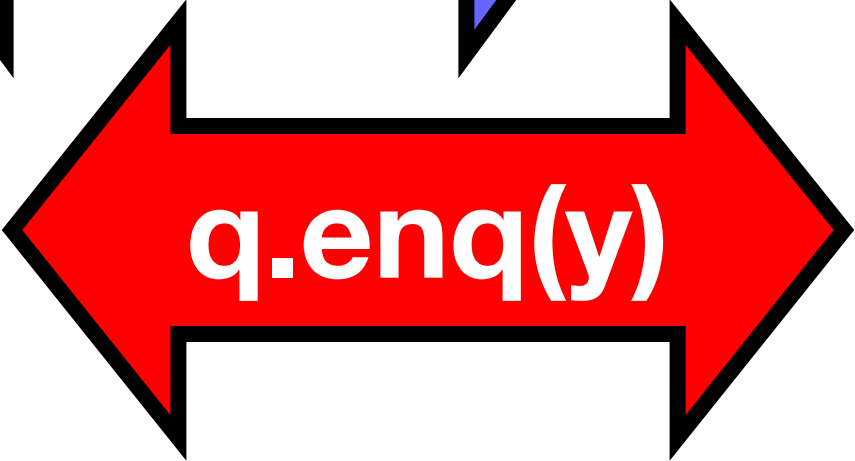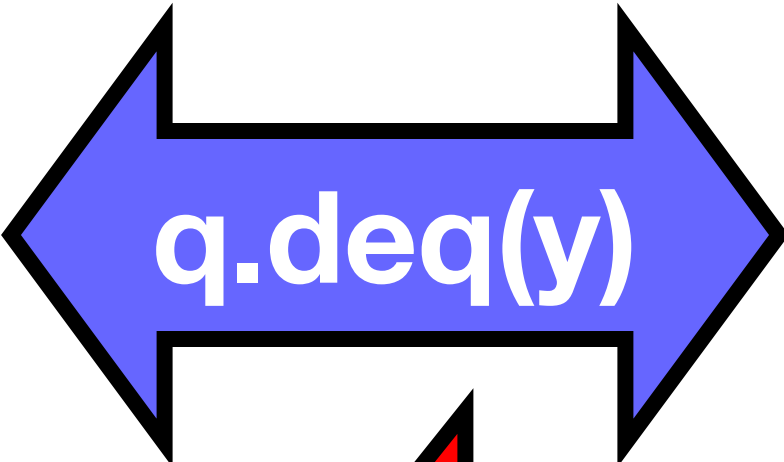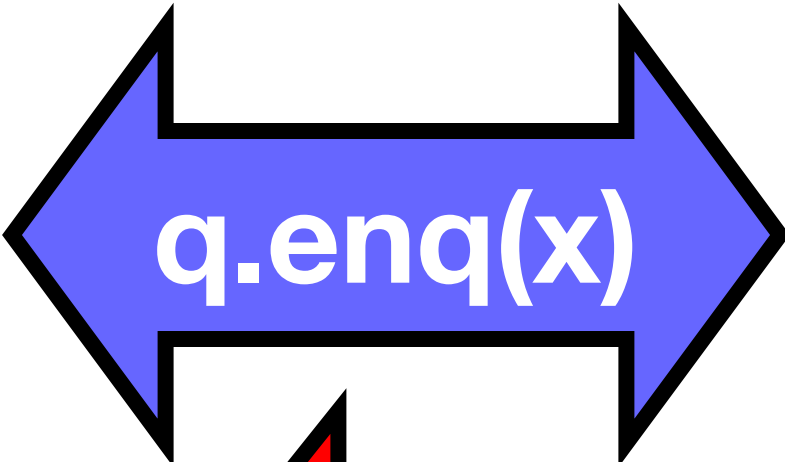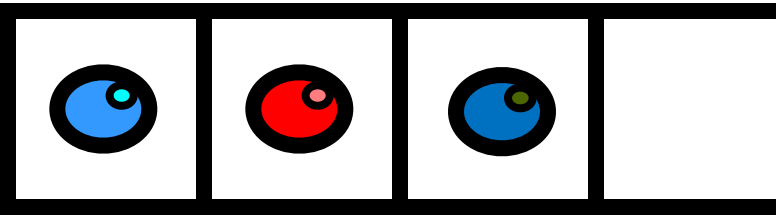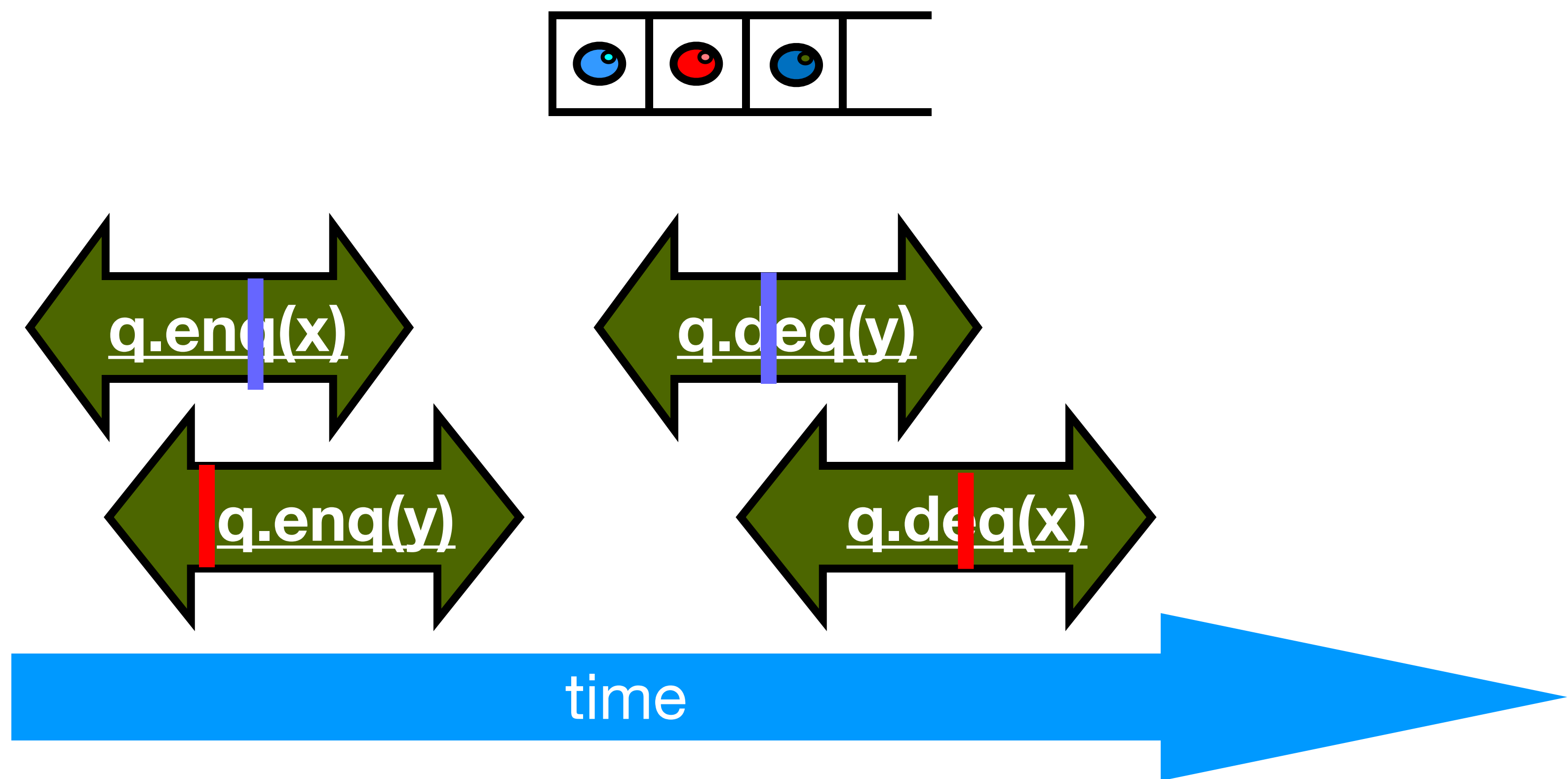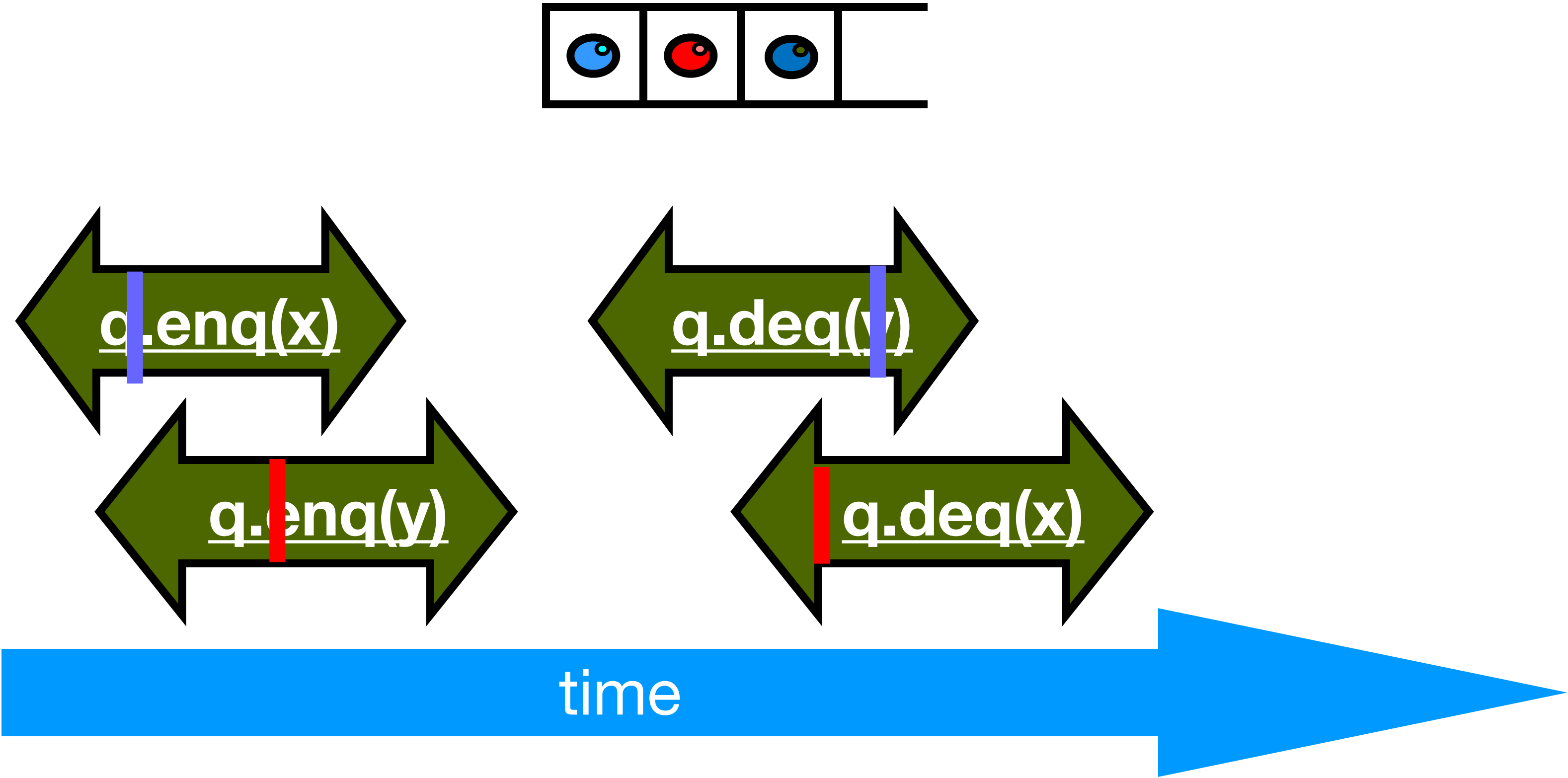
Example
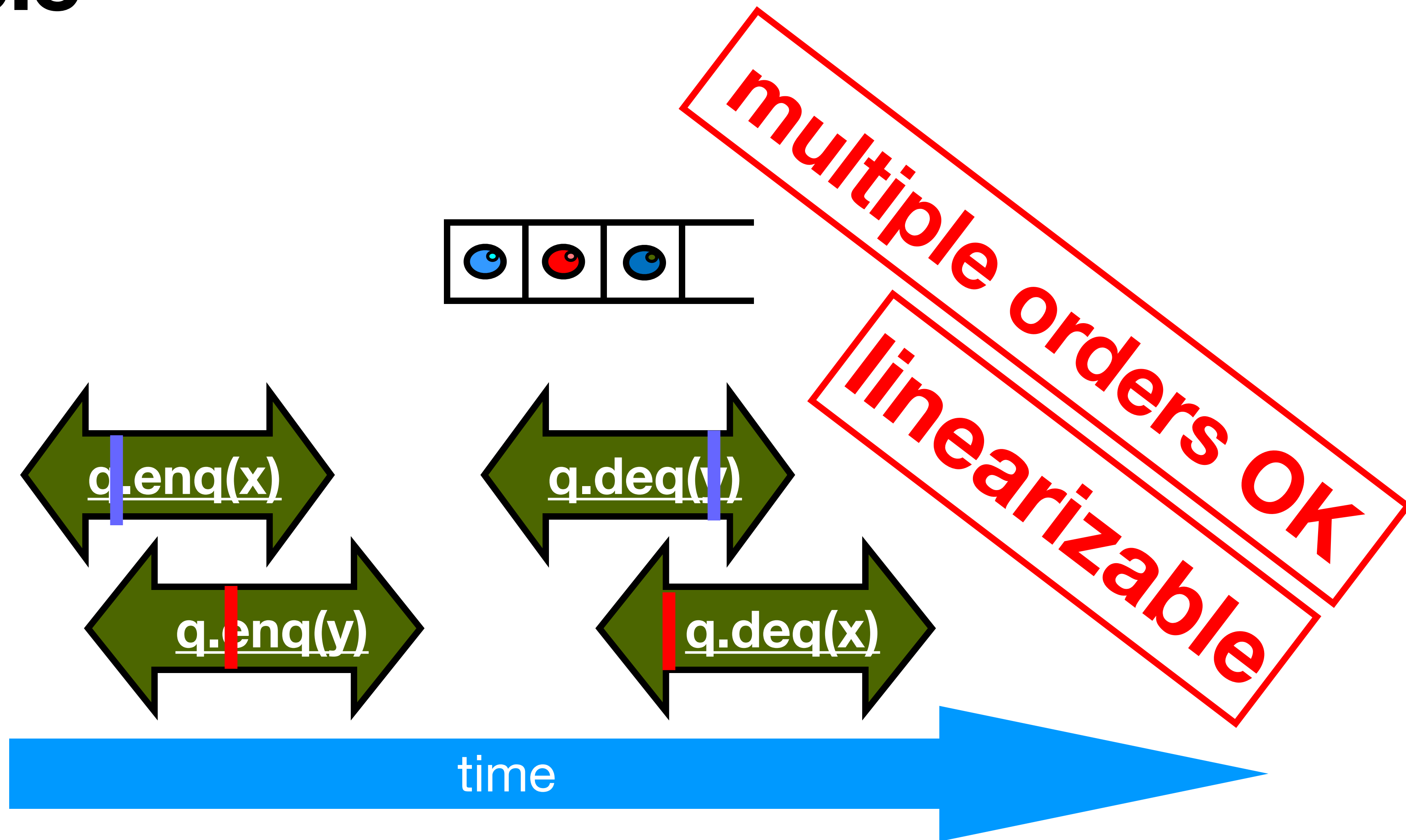
Example

Example

Read/Write Register Example
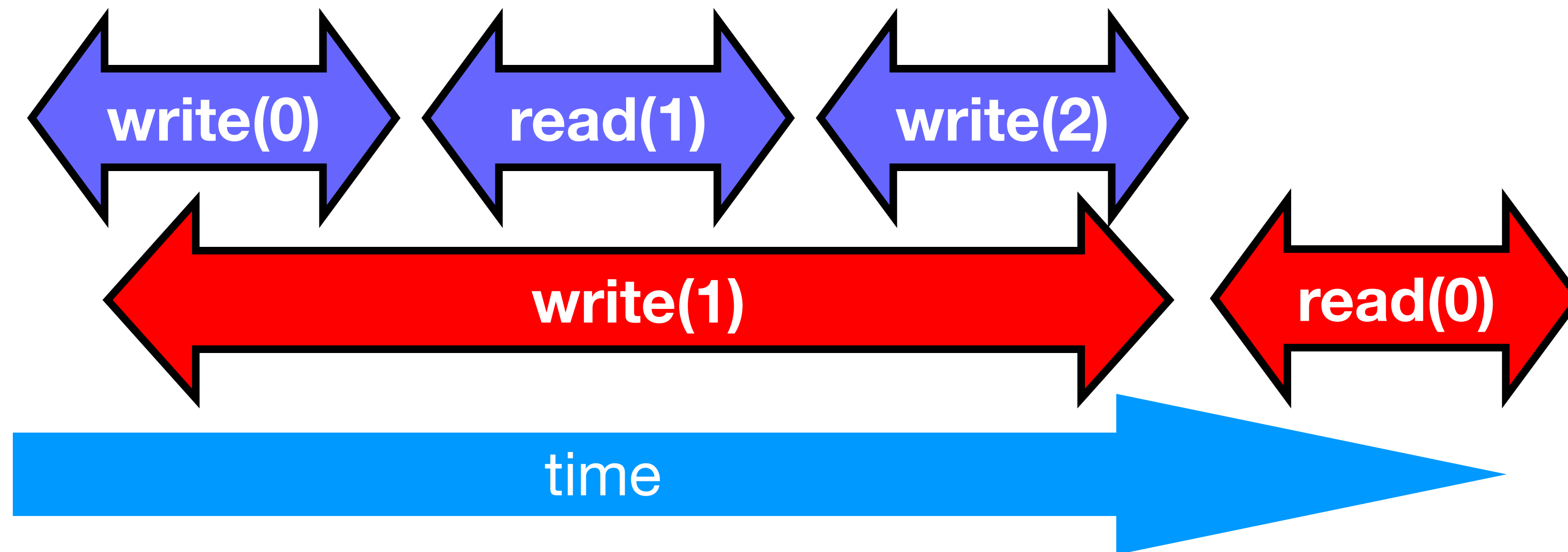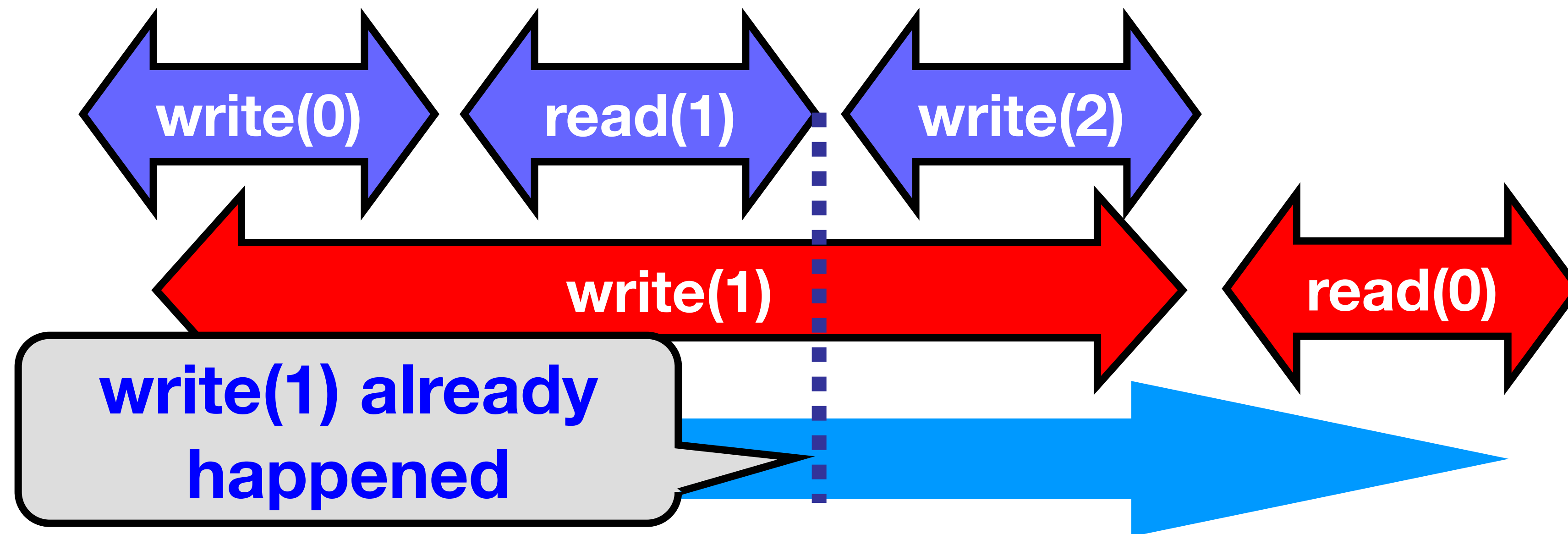
Read/Write Register Example

Read/Write Register Example
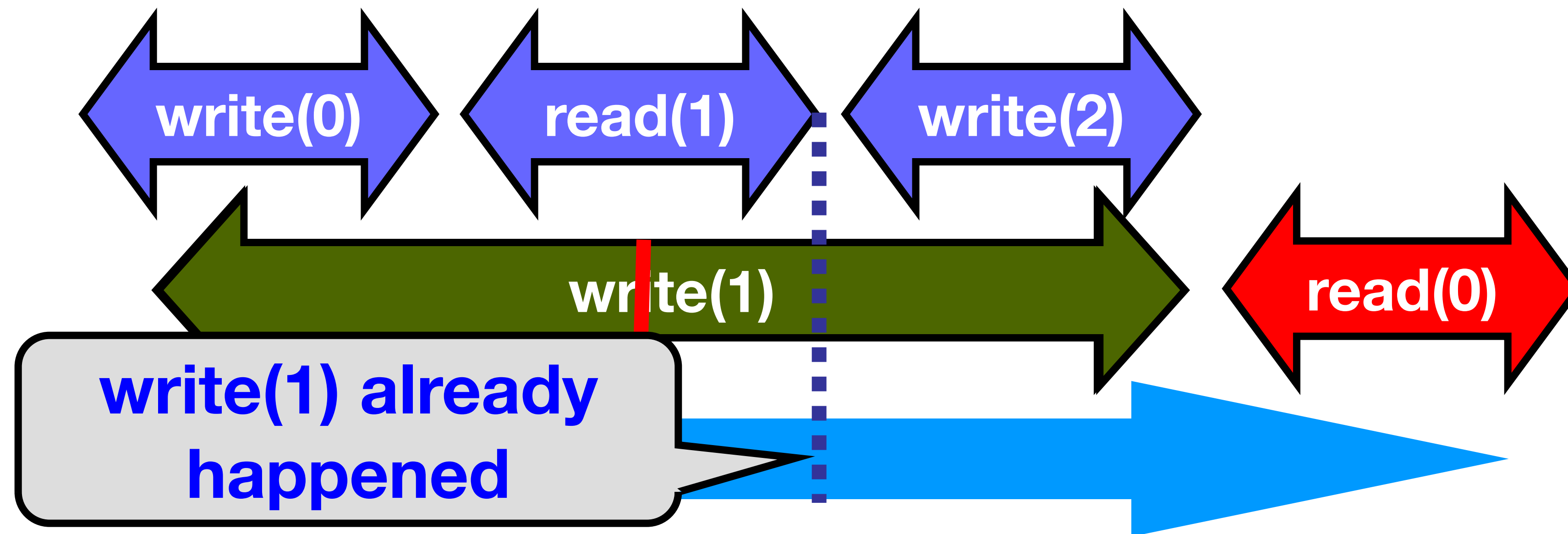
Read/Write Register Example
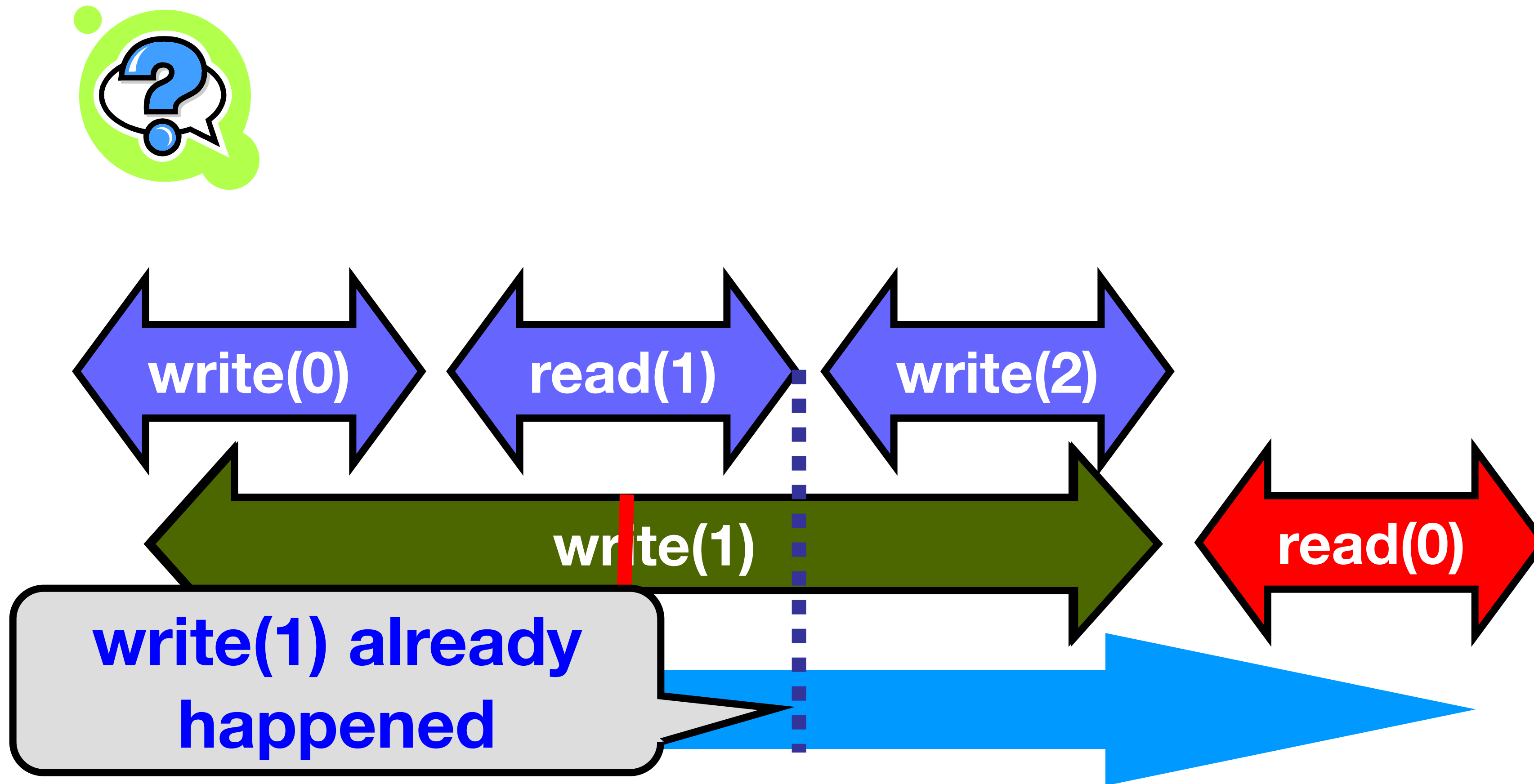
Read/Write Register Example
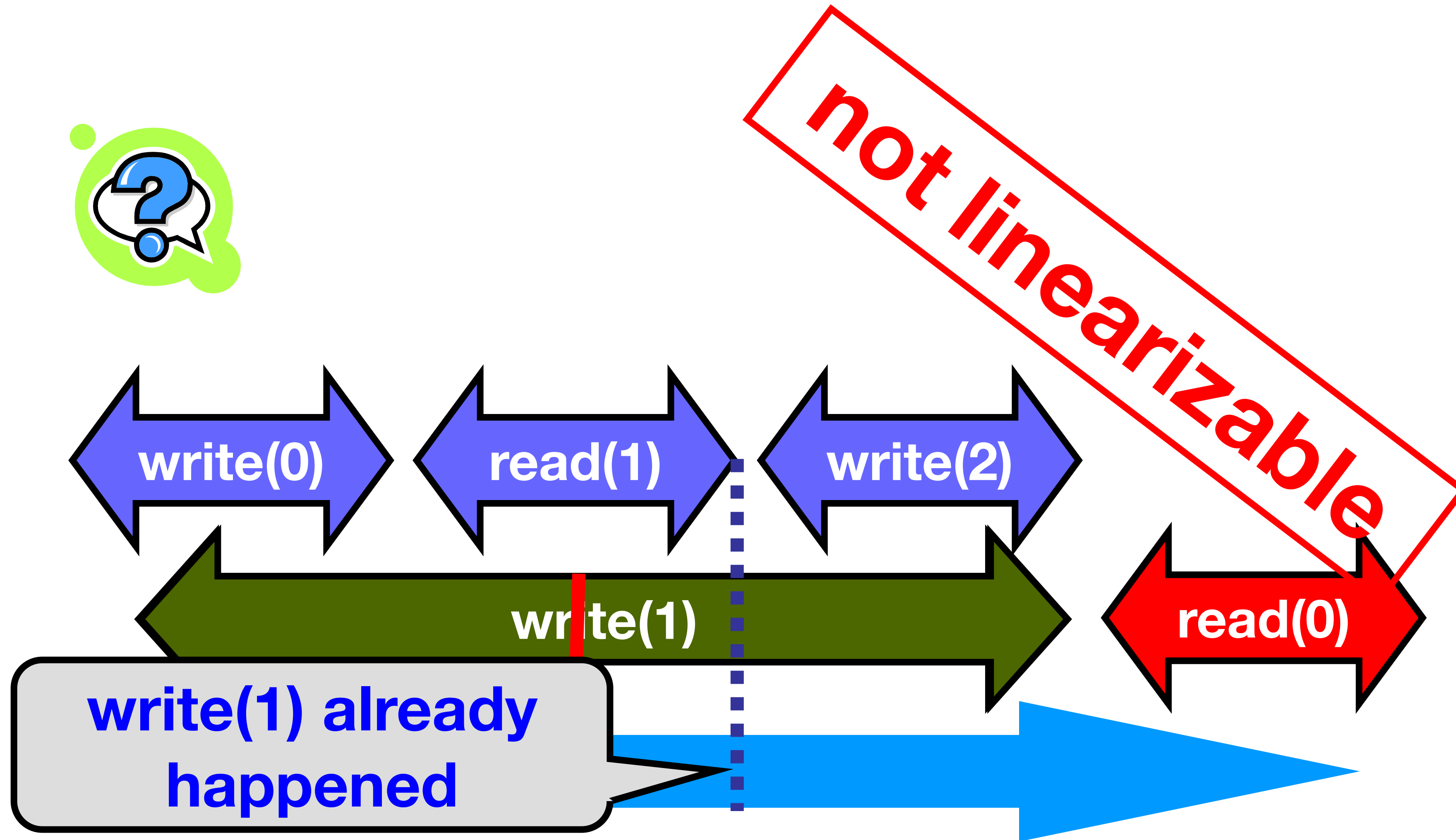
Read/Write Register Example
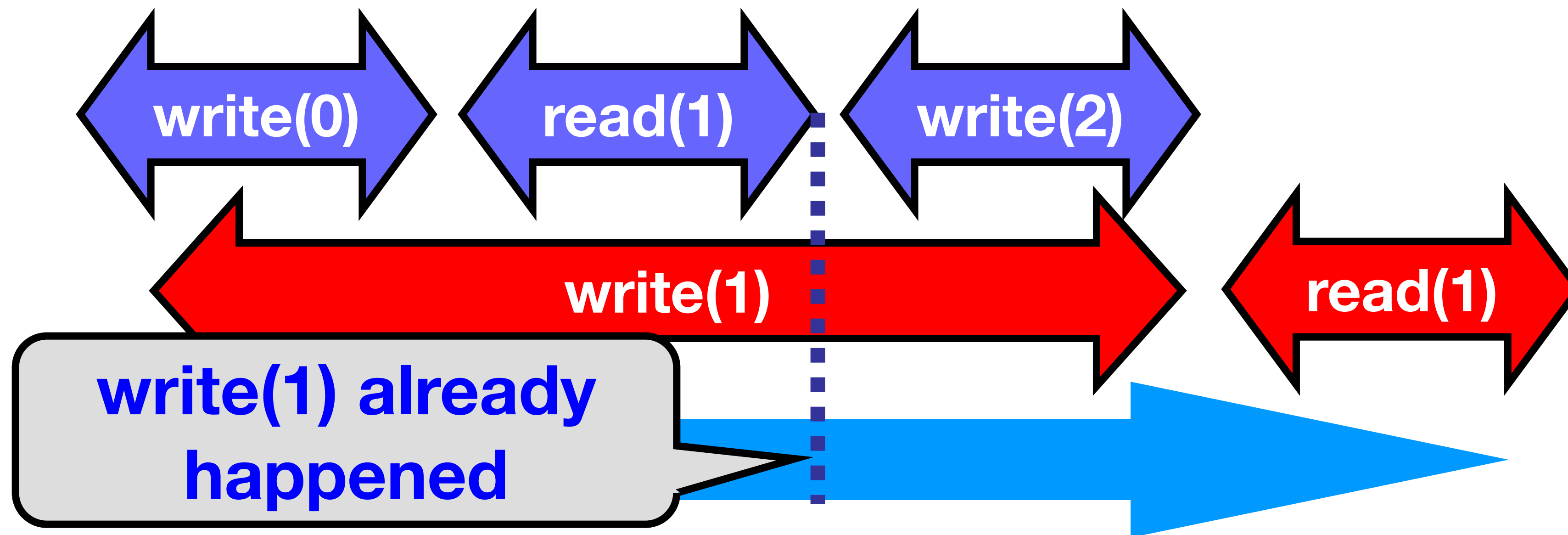
Read/Write Register Example
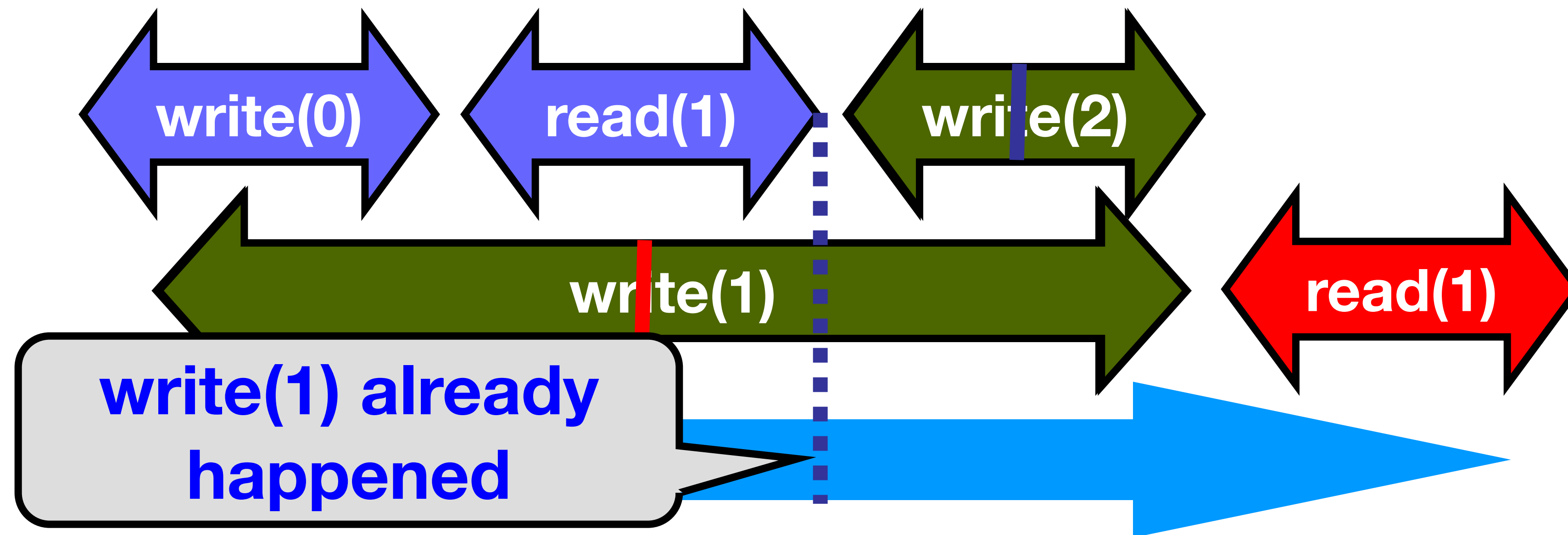
Read/Write Register Example
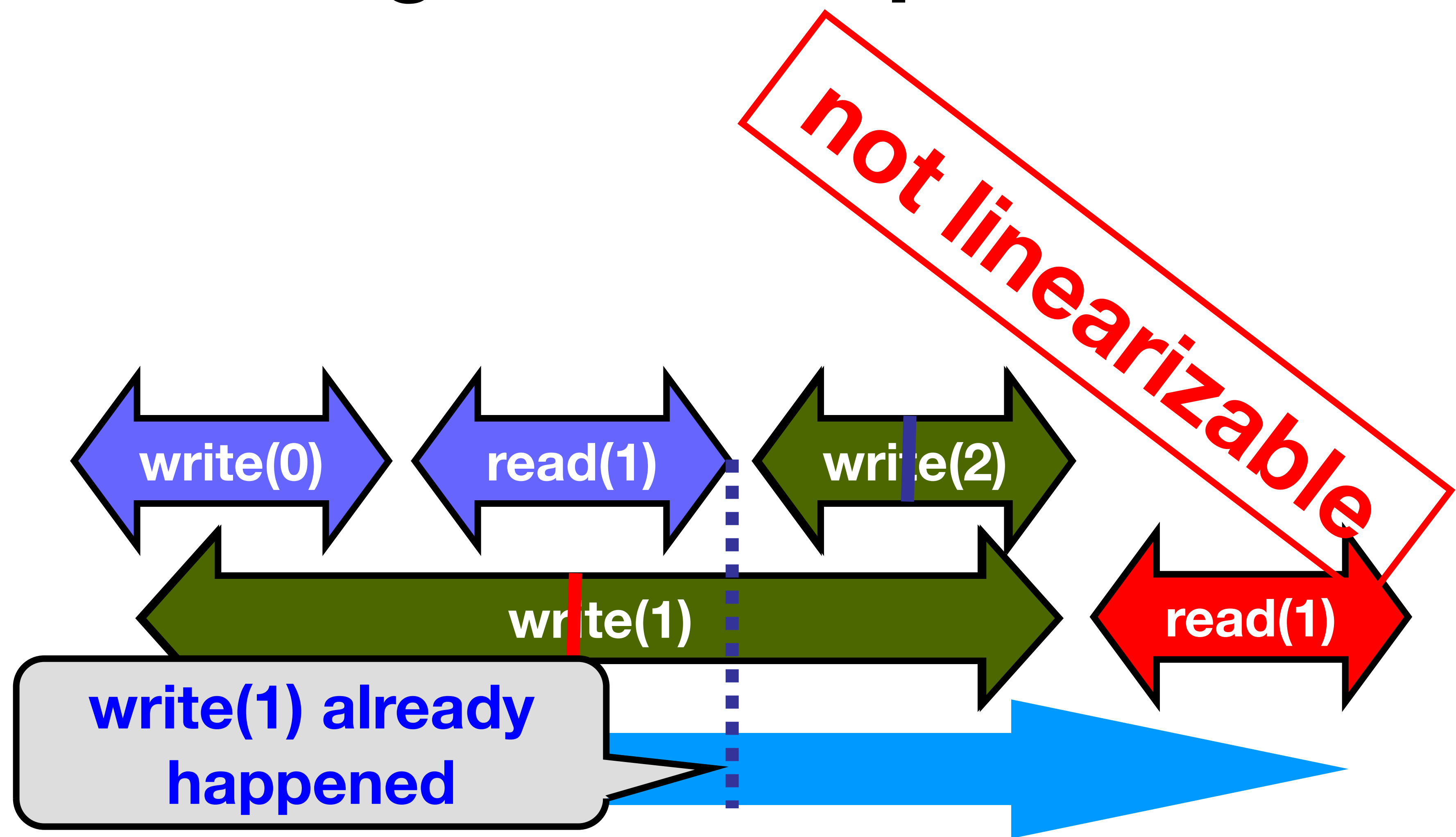
Read/Write Register Example
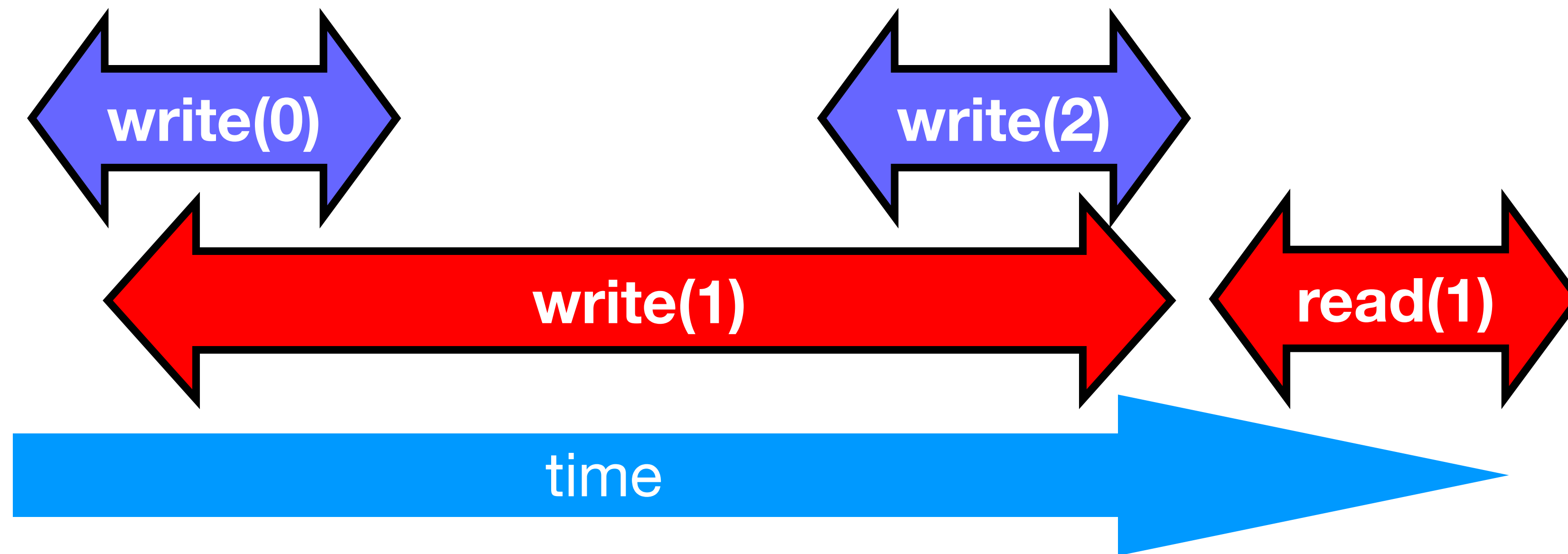
Read/Write Register Example
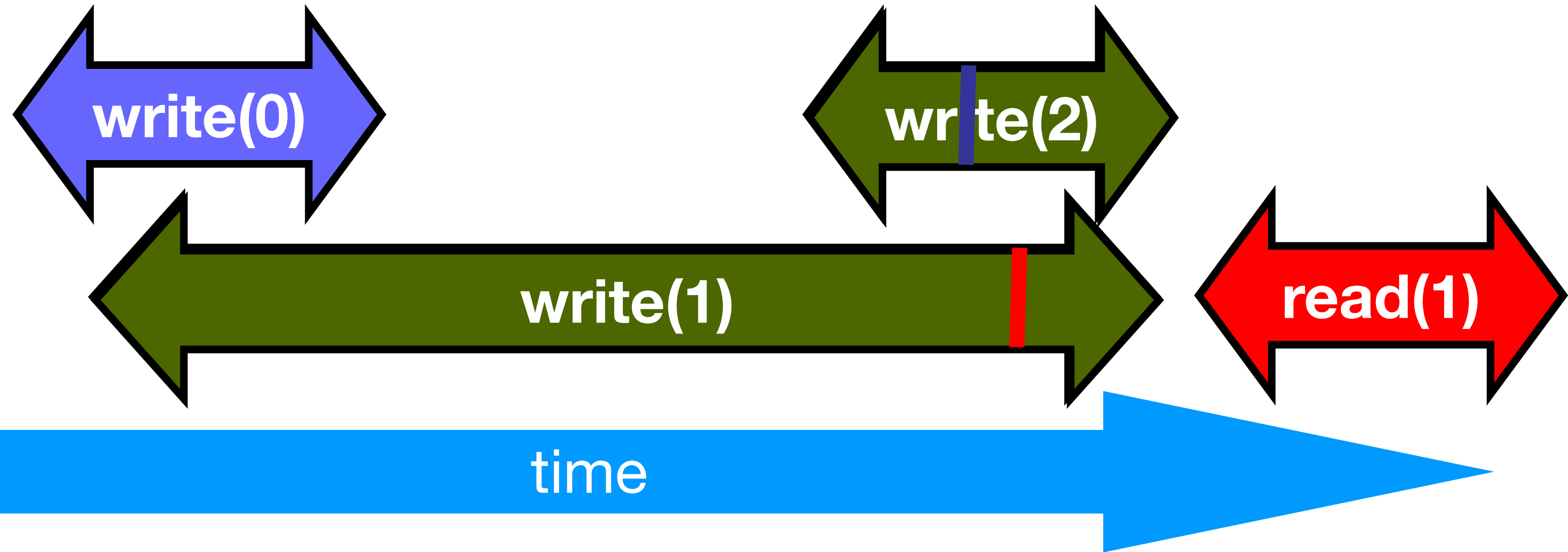
Read/Write Register Example
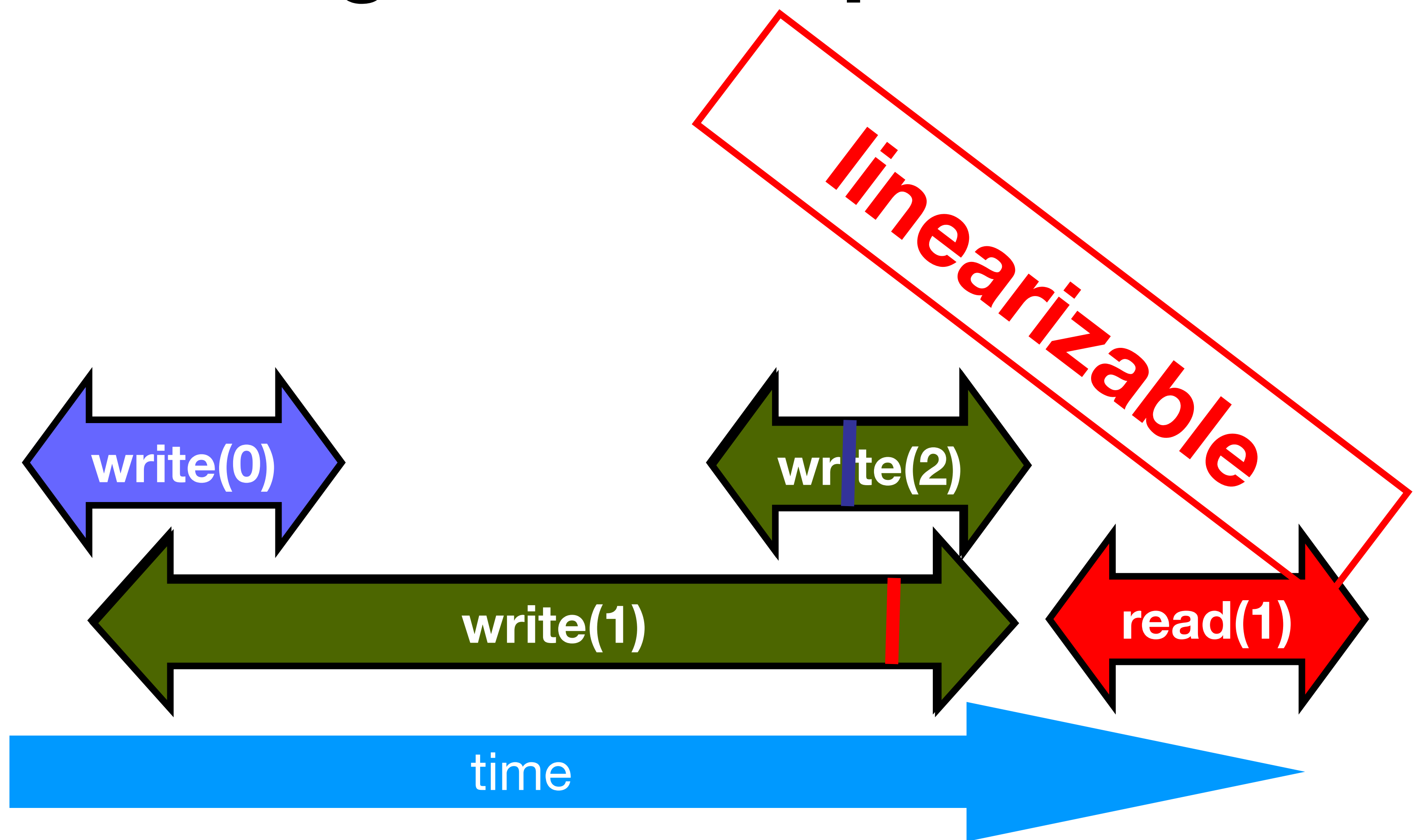
Talking About Executions

- 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

    • In the long run, actually, more important

    • Ask me why!

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`**

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`**

- Note that I'm following the convention of the book

  - Book uses OO

  - Code in this course uses FP

- Note that we're still reasoning using objectivism

- For the current discussion, distinction doesn't matter

  – `q.enq(x)` is read as **`enq q x`** in code

  – Returns **`void`** is read as returns **`()`**

  – Throws **`empty`** is read as raises **`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

object

method

Invocation Notation

A q.enq(x)

thread

object

method

arguments
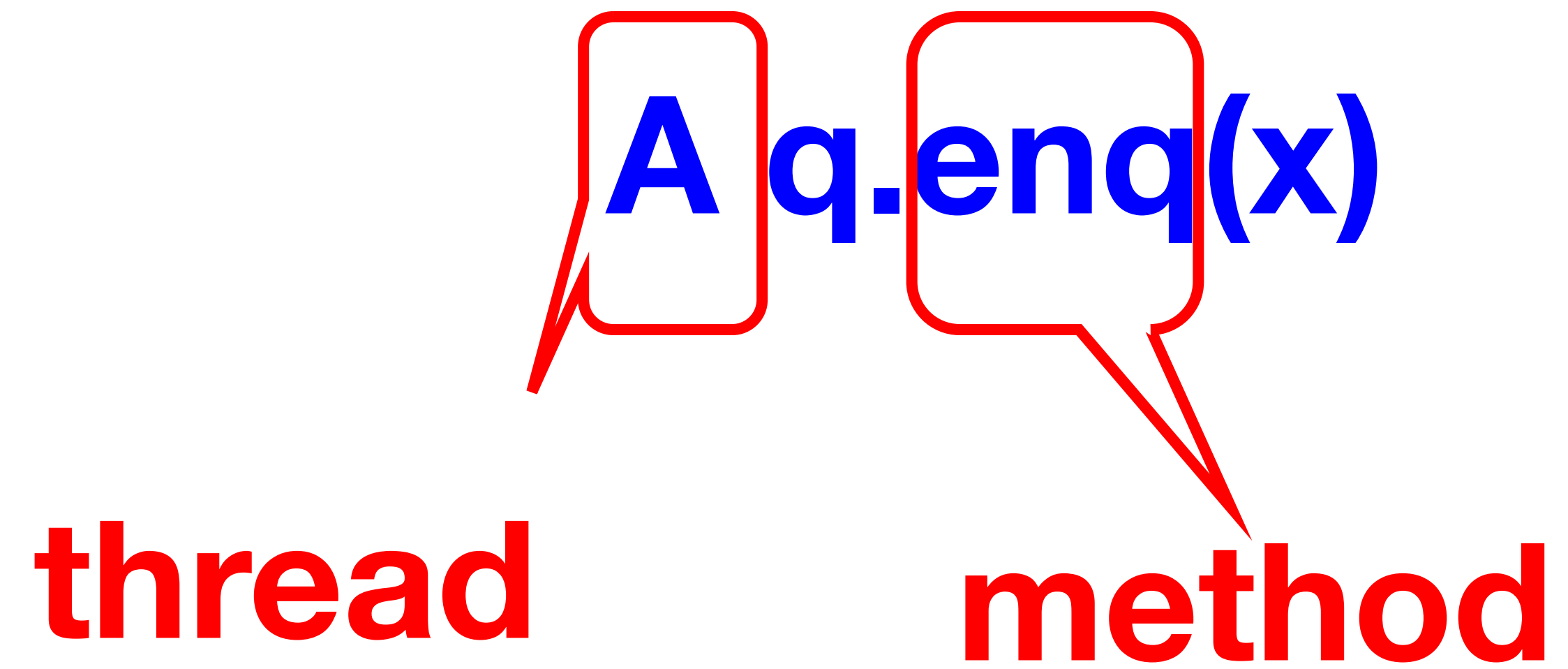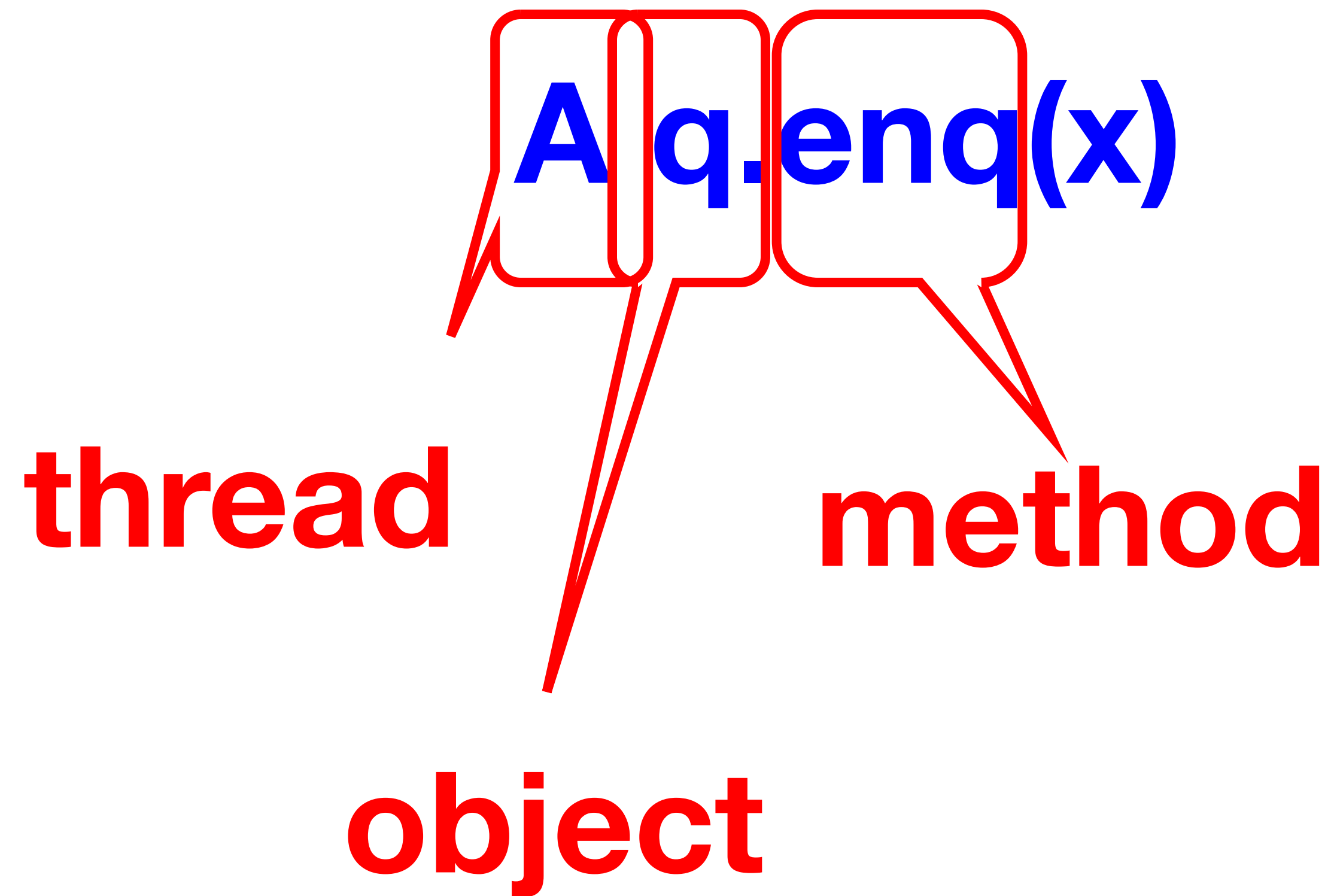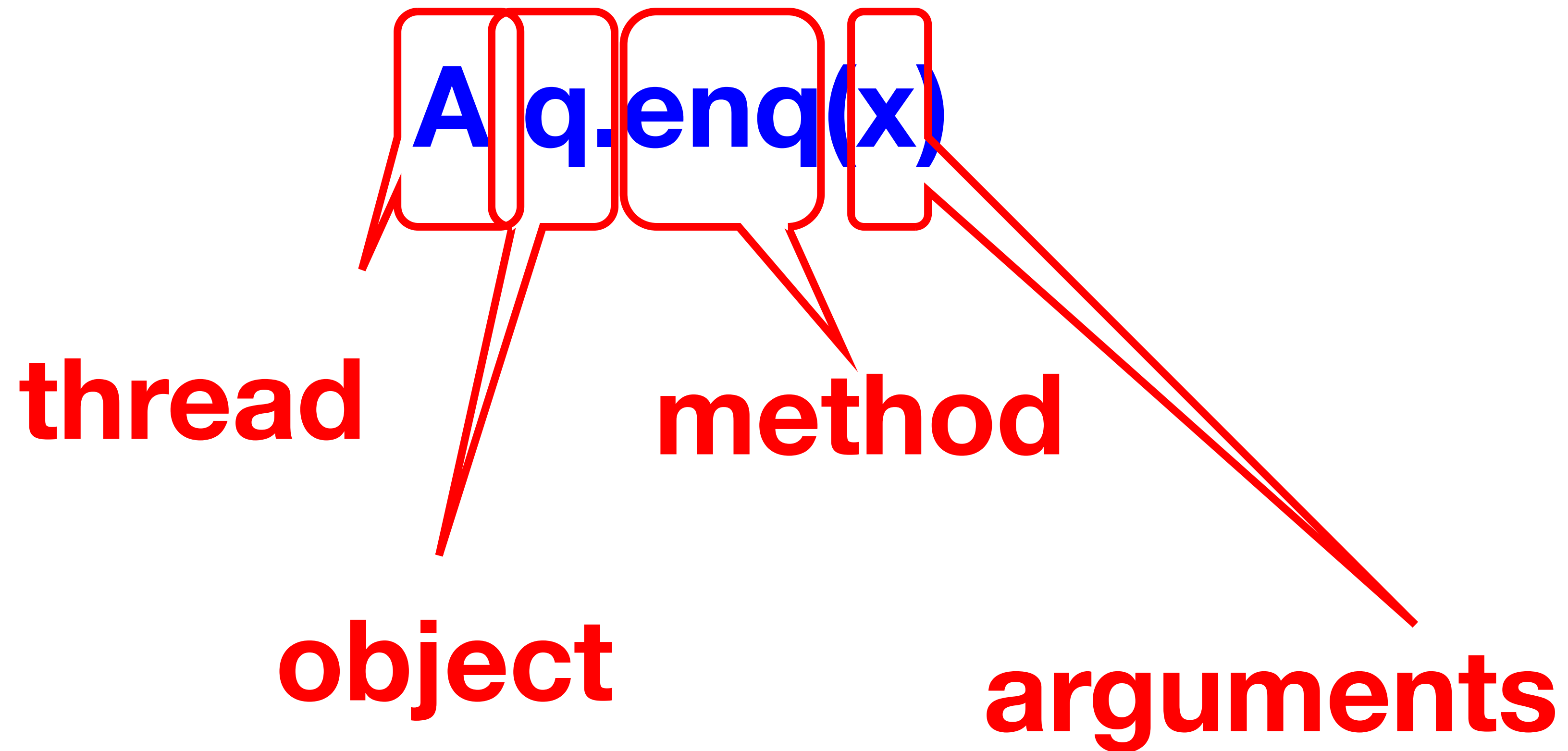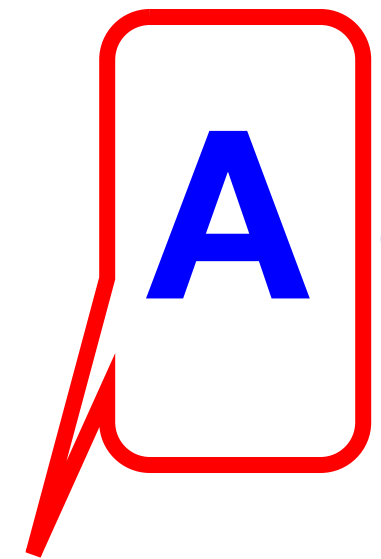
Response Notation

A q: void

Response Notation

A **q: void**

thread

Response Notation
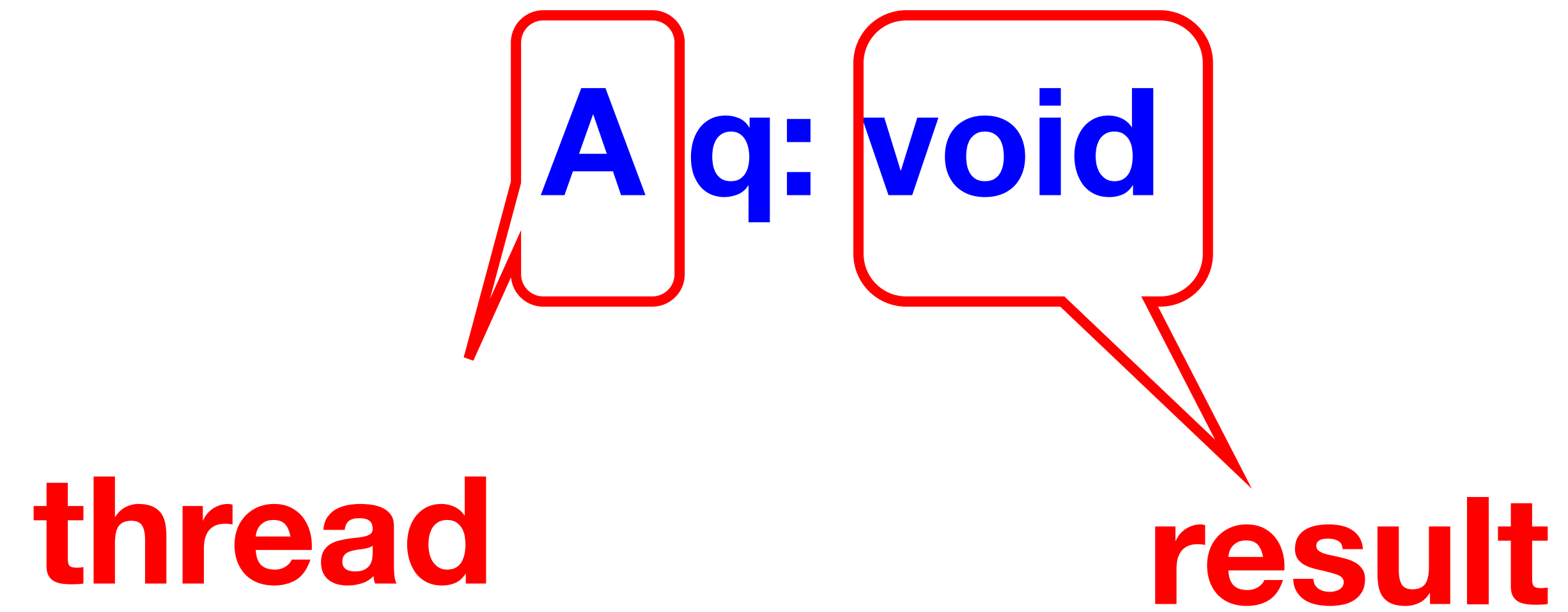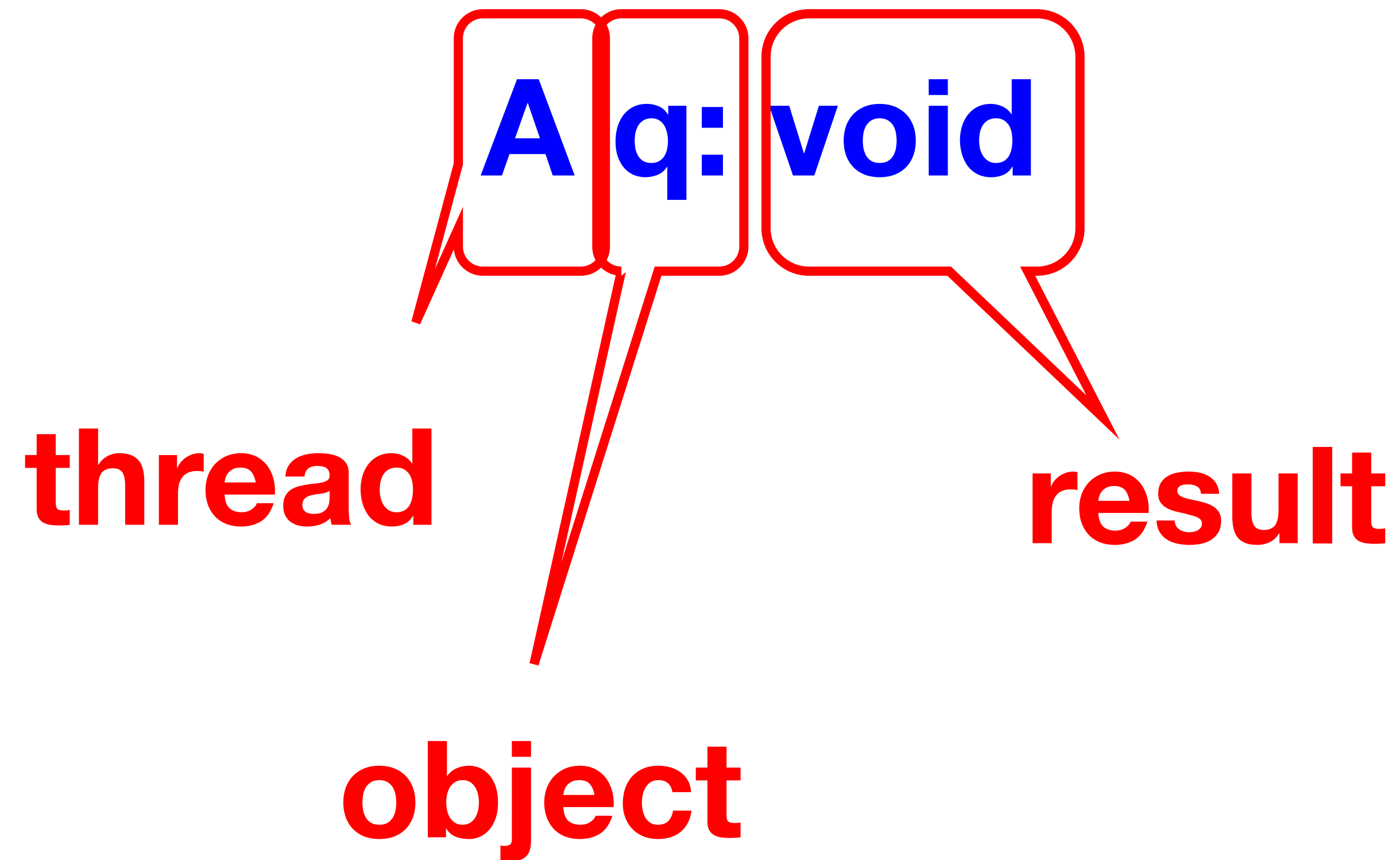
Response Notation
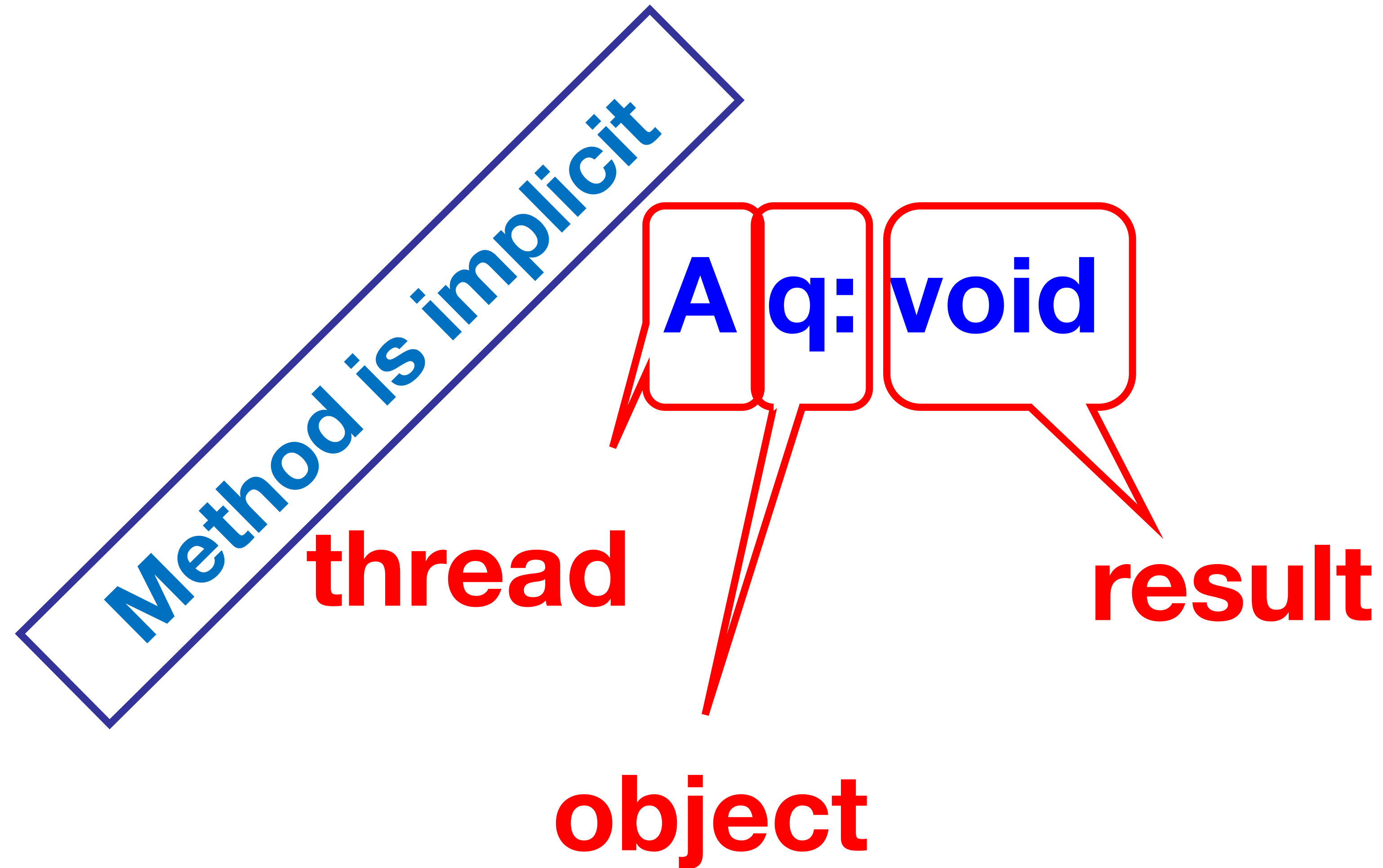
Response Notation

Response Notation
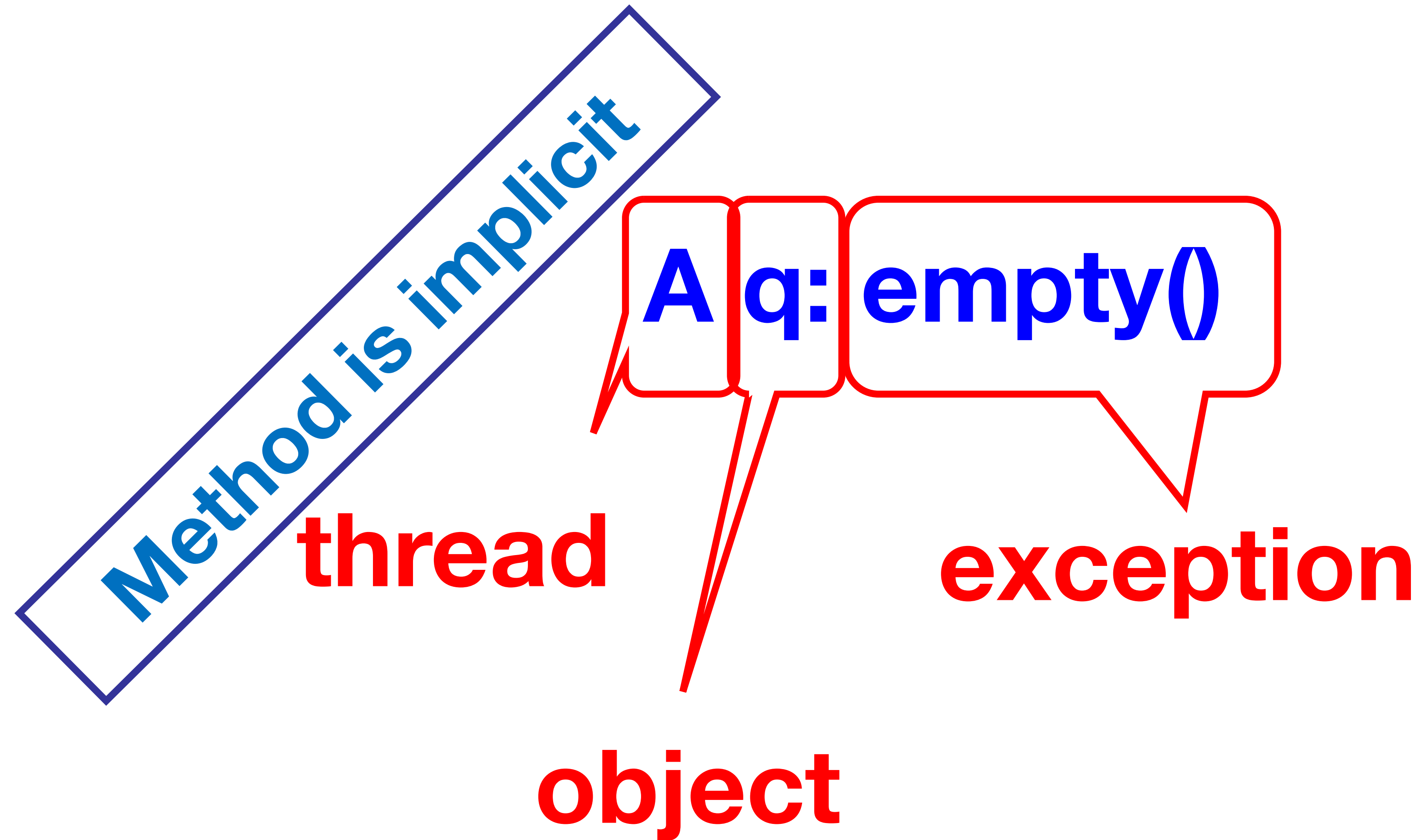
History — Describing an execution

$$H = \begin{array}{l} \text{A q.enq(3)} \\ \text{A q:void} \\ \text{A q.enq(5)} \\ \text{B p.enq(4)} \\ \text{B p:void} \\ \text{B q.deq()} \\ \text{B q:3} \end{array}$$

Sequence of invocations and responses

History — Describing an execution

- Invocation & response *match* if

Thread
names agree

Object names agree

```
A  q.enq(3)
A  q:void
```

Method call

Object Projections

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

Object Projections

```
A q.enq(3)
A q:void
```

H|q =

```
B q.deq()
B q:3
```
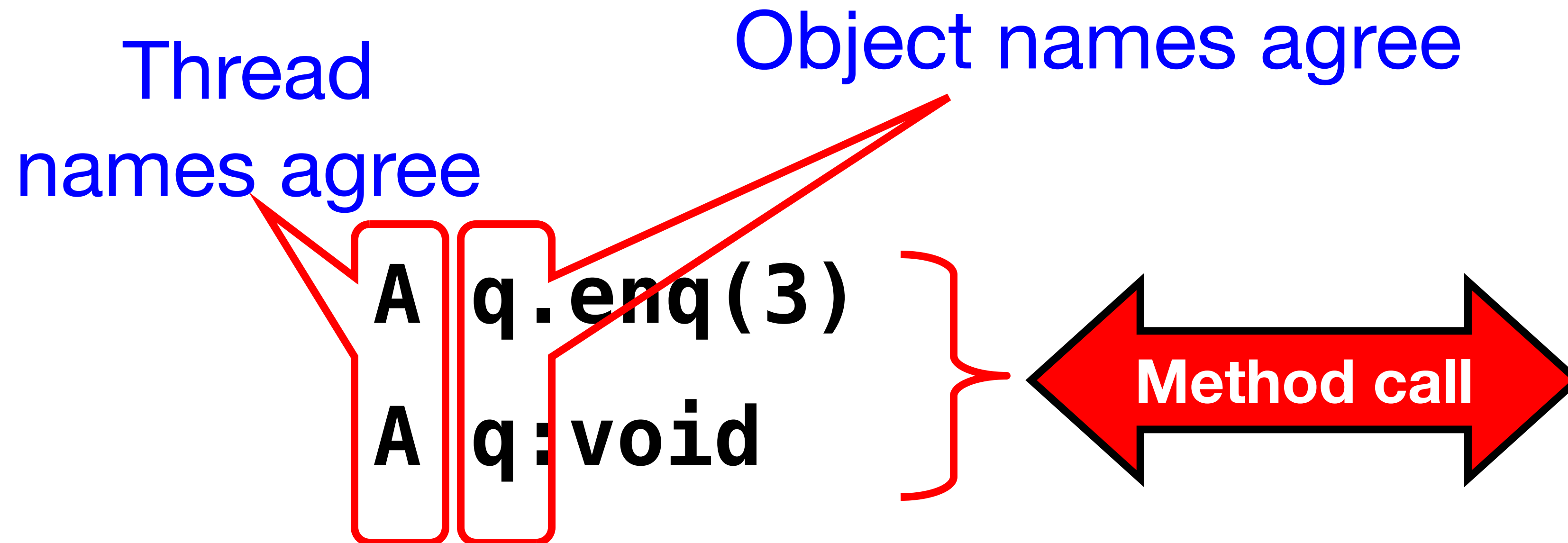
Thread Projections

$H =$

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

Thread Projections

$H|B =$

B p.enq(4)
B p:void
B q.deq()
B 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 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
```
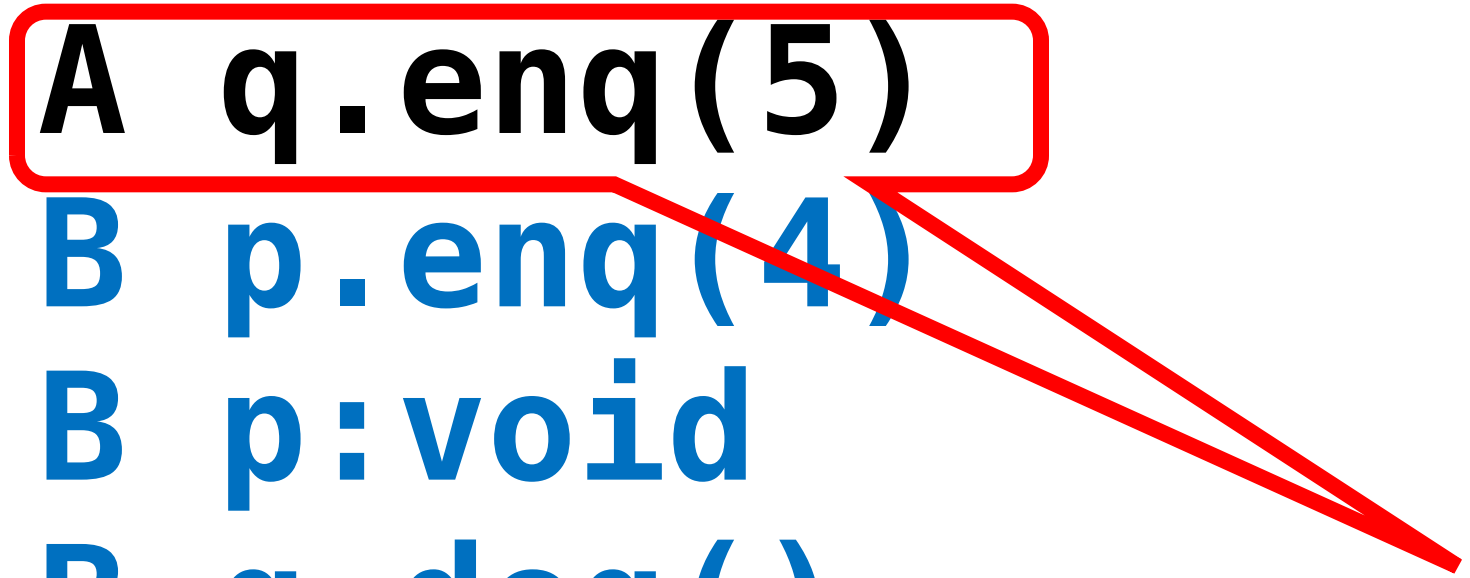
May or may not
have taken effect

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 ← **match**
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   **match**

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   **match**

B q.deq()
B q:3   **match**

A q:enq(5)

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

B p.enq(4)
B p:void

B q.deq()
B q:3

A q:enq(5)

match

match

match

Method calls of different threads do not interleave

Final pending invocation OK

Well-formed Histories

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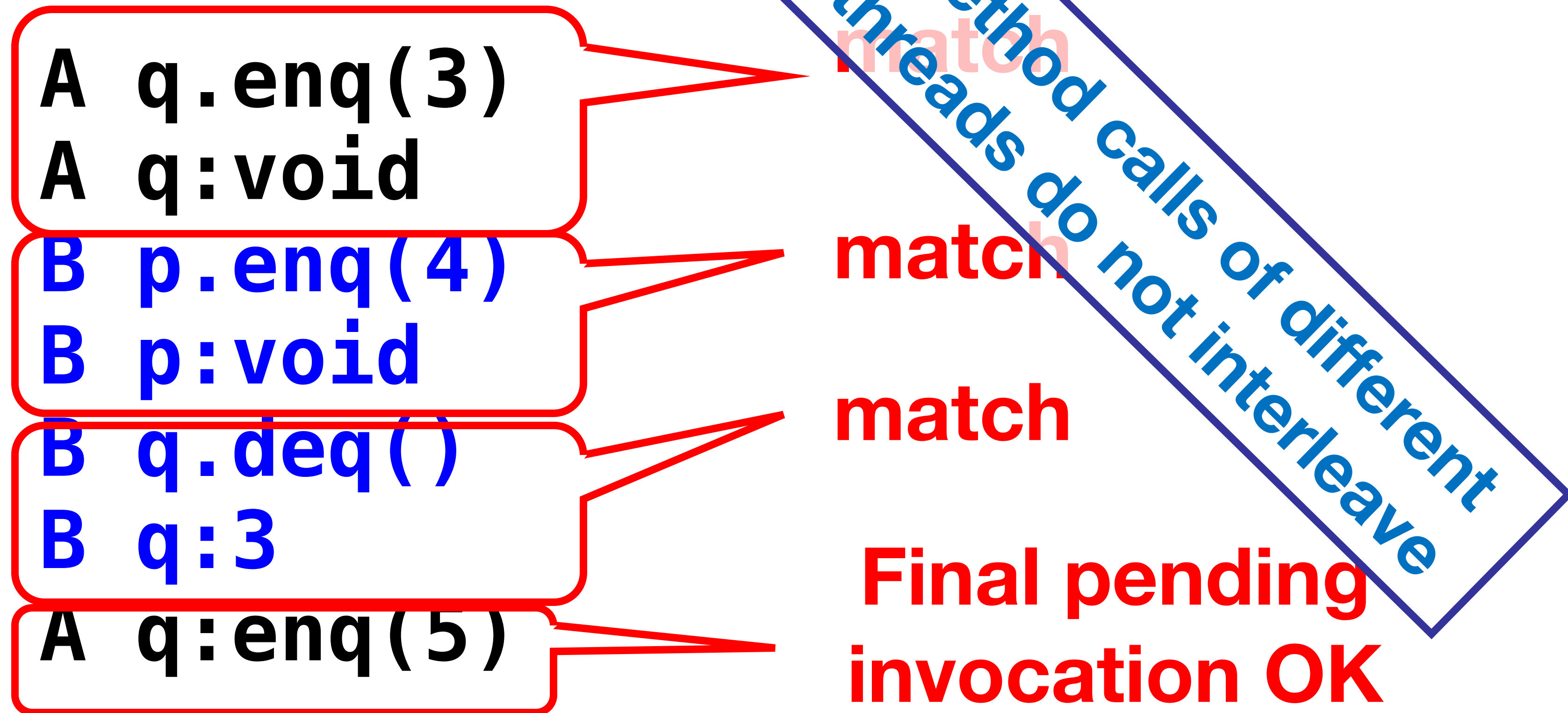
```
A q.enq(3)
B p.enq(4)
B p:void
B q.deq()
A q:void
B q:3
```

Well-formed Histories

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

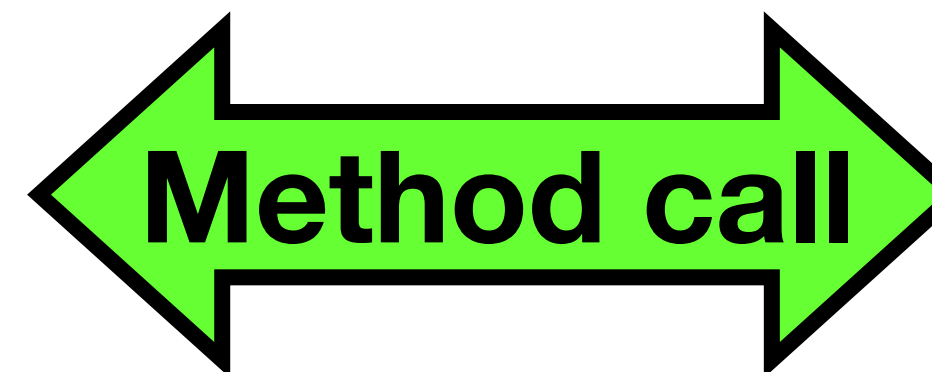 - But plenty of other techniques exist …

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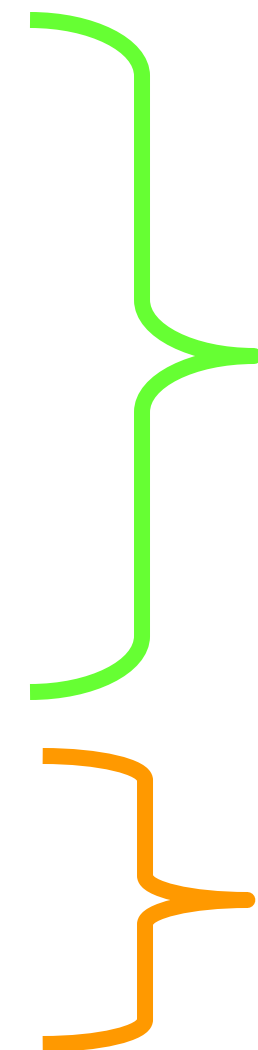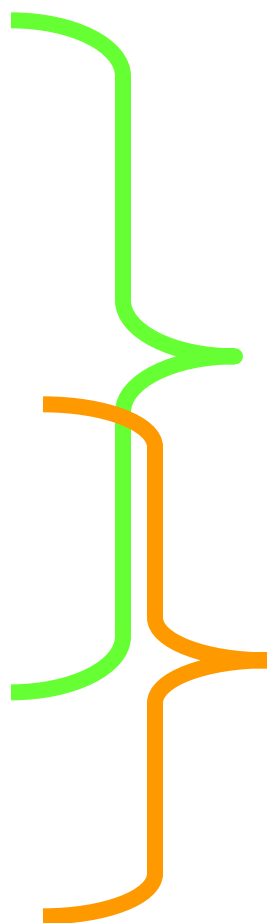
A method call precedes another if response event precedes invocation event

Method call **Method call**

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

overlap shown in red

Method call

Method call
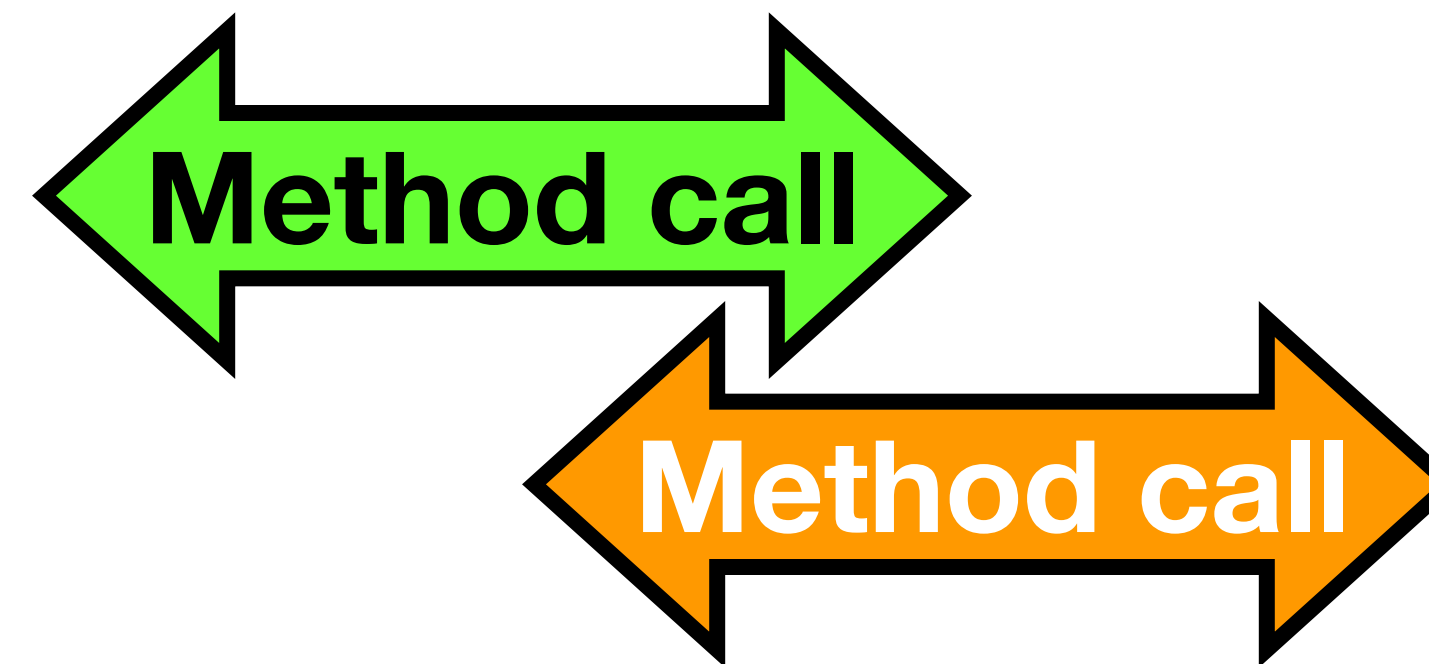
Notation

- 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
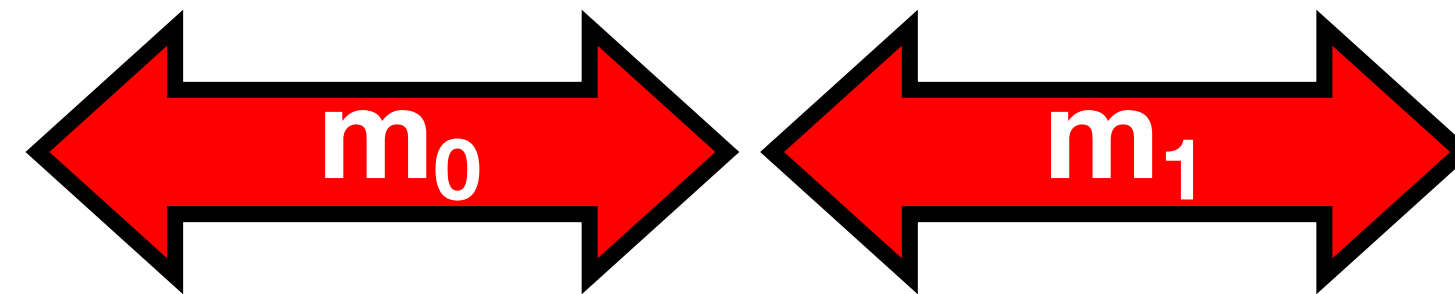
Linearizability

- 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**

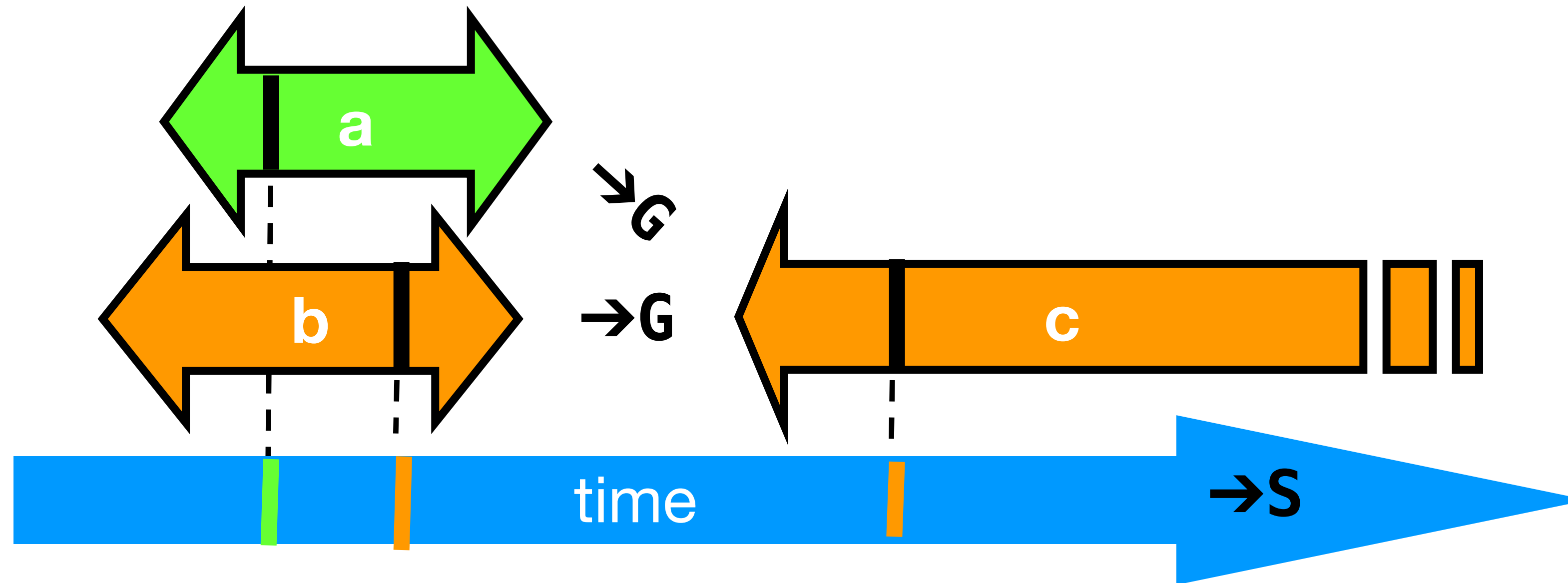 - where $\rightarrow_G \subset \rightarrow_S$

Remarks on Linearizability

- Some pending invocations

  – Took effect, so keep them

  – Discard the rest

- Condition $\rightarrow_G \subseteq \rightarrow_S$

  – Means that **S** respects "real-time order" of **G**

Ensuring →G ⊂ →S

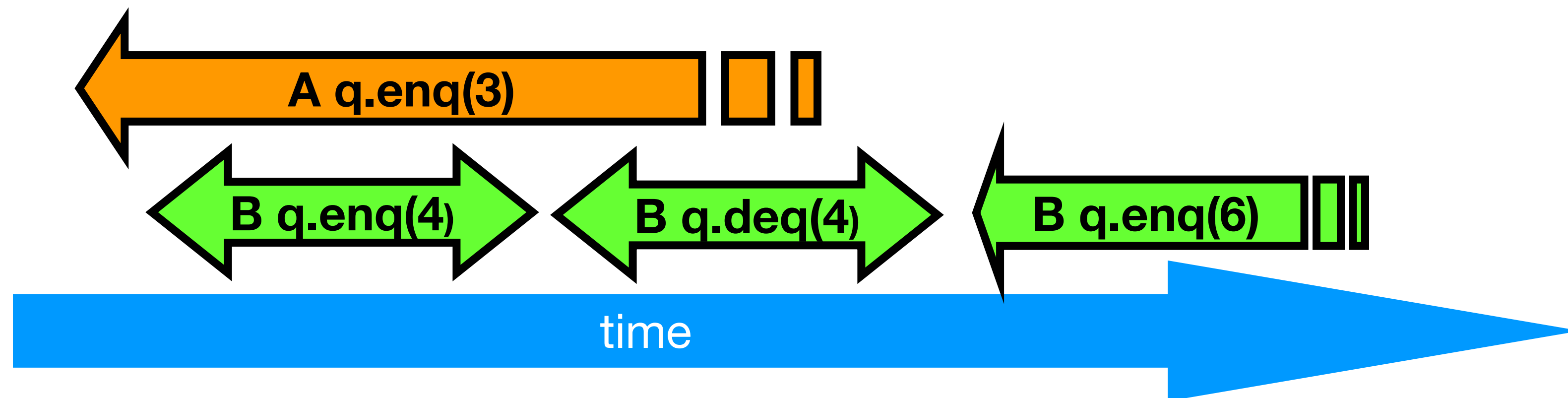$$\rightarrow_G = \{a \rightarrow c, b \rightarrow c\}$$
$$\rightarrow_S = \{a \rightarrow b, a \rightarrow c, b \rightarrow c\}$$

Ensuring →G ⊂ →S

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

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

A limitation on the Choice of S!
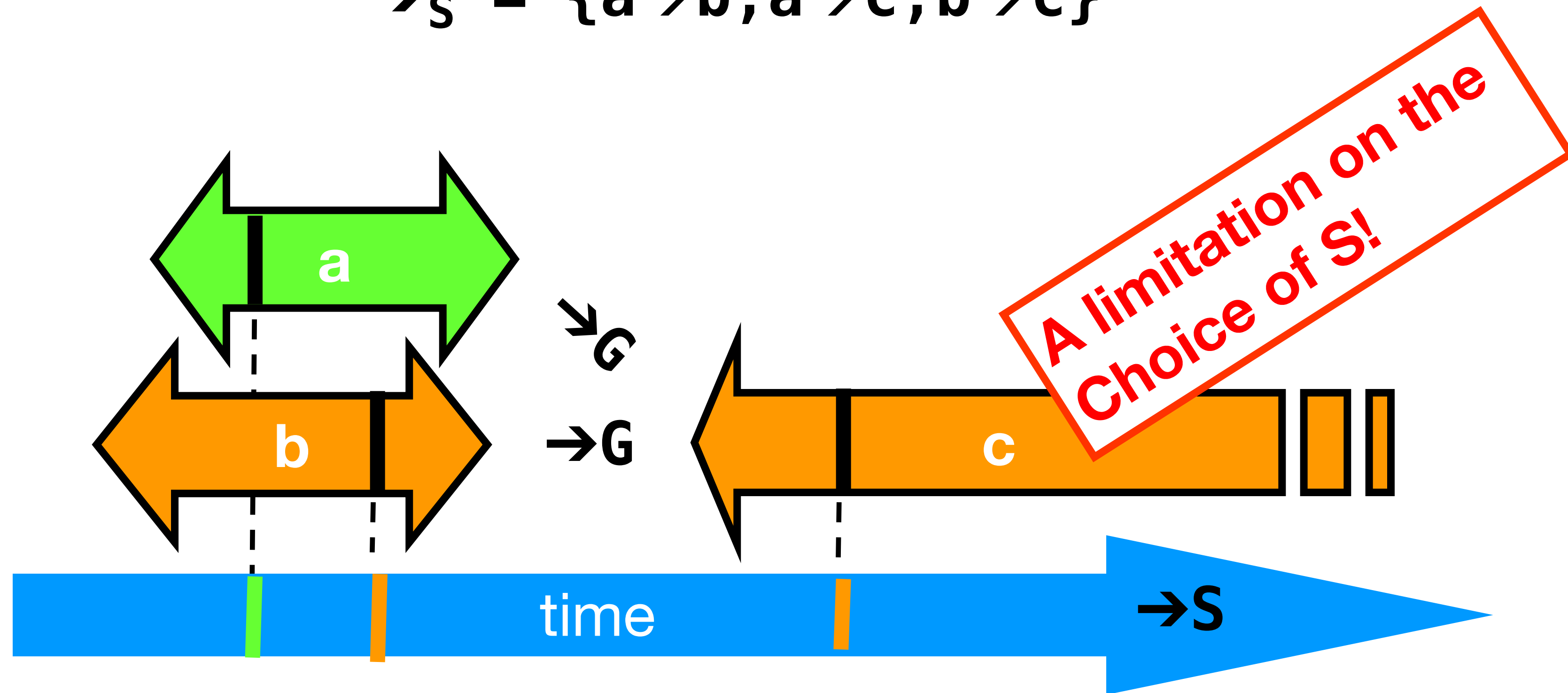
Example

```
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

A q.enq(3)

B q.enq(4)   B q.deq(3)   B q.enq(6)

time

Example

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
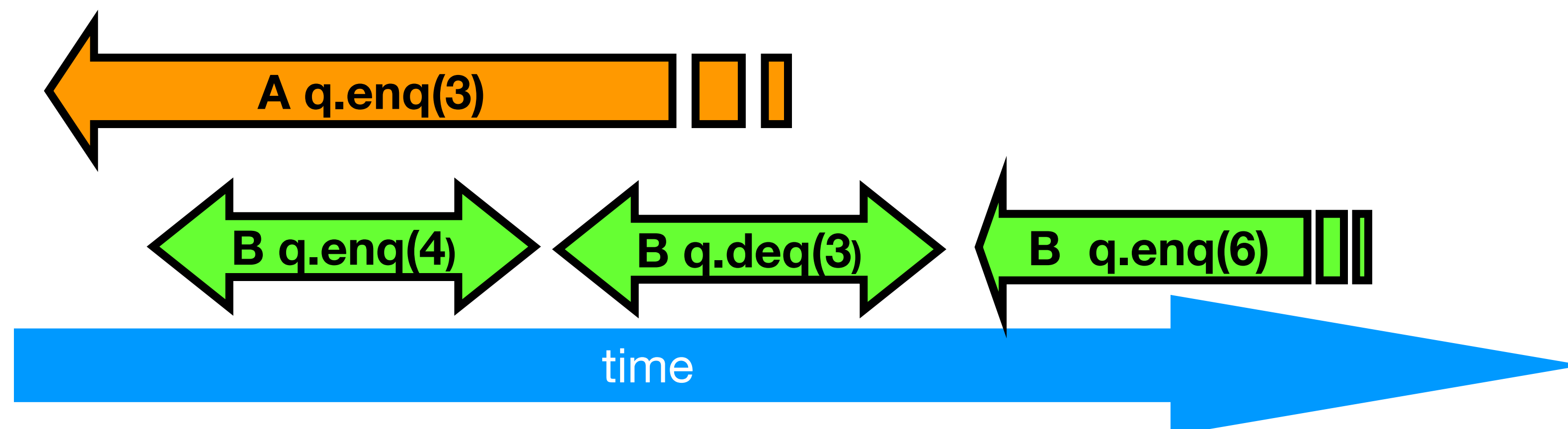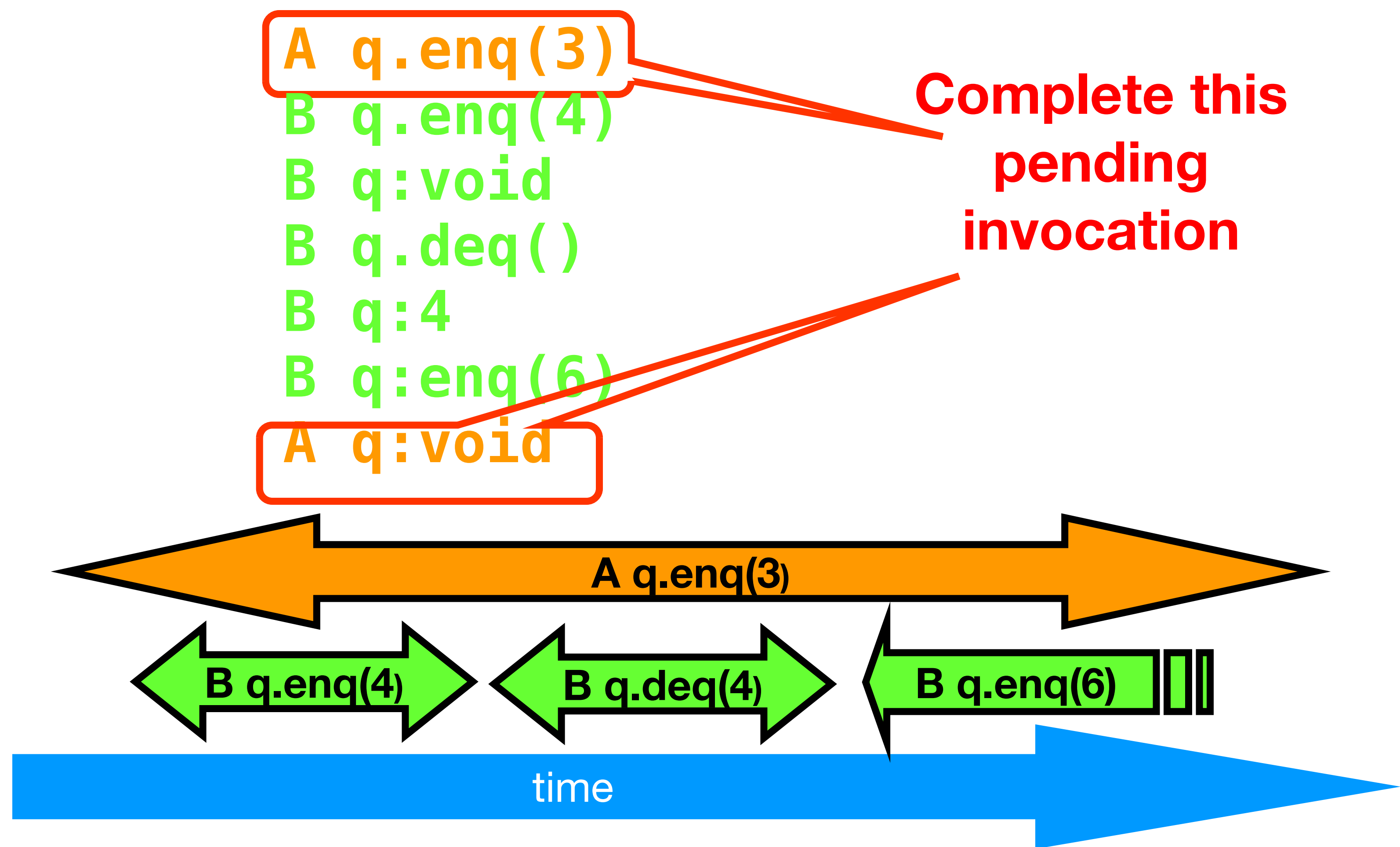
discard this one

A q.enq(3)

B q.enq(4)   B q.deq(4)   B q.enq(6)

time

Example

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

discard this one

A q:void

A q.enq(3)

B q.enq(4)   B q.deq(4)

time

Example

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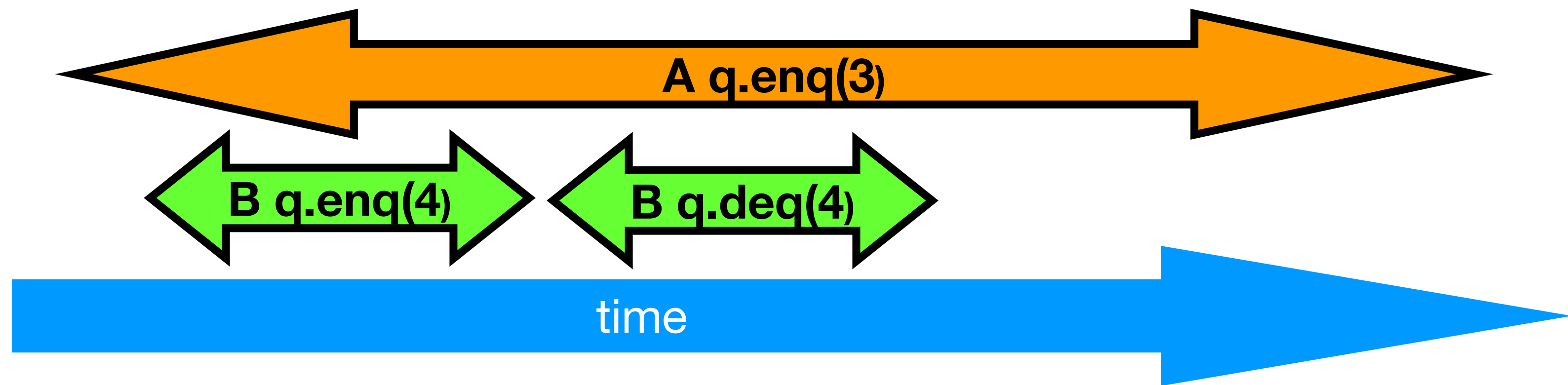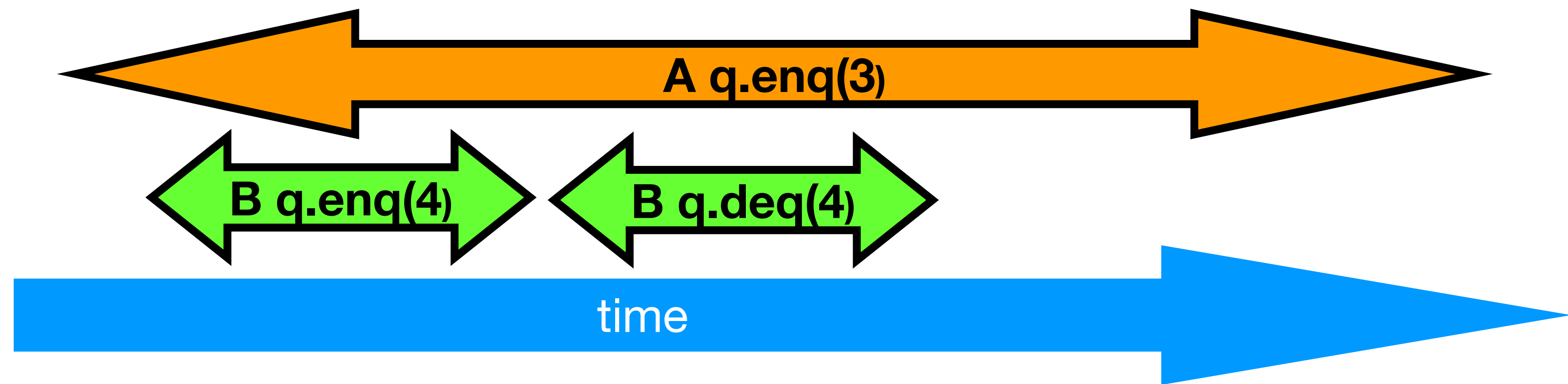
Example

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

A q.enq(3)

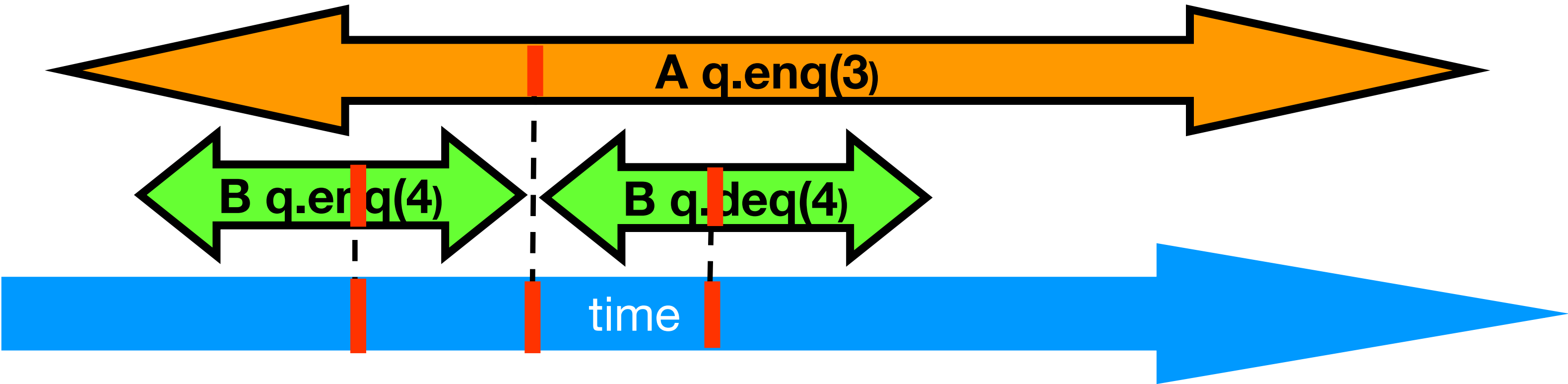B q.enq(4)   B q.deq(4)

time

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
```


A q.enq(3)

B q.enq(4)

B q.deq(4)

time

Composability Theorem

- History H is linearizable if and only if

  – For every object x

  – $H|x$ is linearizable

- We care about objects only!

  – (Materialism?)

Why does composability matter?

- Modularity

- Can prove linearizability of objects in isolation

- Can compose independently-implemented objects

Reasoning about Linearizability: Locking

```
let deq q =
  Mutex.lock q.lock;
  try
    if q.tail = q.head then
      raise Empty;
    match q.items.(q.head mod q.capacity) with
    | None -> assert false
    | Some x ->
        q.head <- q.head + 1;
        Mutex.unlock q.lock;
        x
  with e ->
    Mutex.unlock q.lock;
    raise e
```

Linearization points
are when locks are
released

More Reasoning — Wait-free

```ocaml
(** Enqueue – should be called by only ONE thread *)
let enq q x =
  (* Check if queue is full *)
  if q.tail – q.head = q.capacity then
    raise Full;
  (* Write to the array *)
  q.items.(q.tail mod q.capacity) <- Some x;
  (* Advance tail *)
  q.tail <- q.tail + 1
```

```ocaml
(** Dequeue – should be called by only ONE thread *)
let deq q =
  (* Check if queue is empty *)
  if q.tail = q.head then
    raise Empty;
  (* Read from the array *)
  match q.items.(q.head mod q.capacity) with
  | None -> assert false  (* Should never happen *)
  | Some x ->
      (* Advance head *)
      q.head <- q.head + 1;
      x
```

More Reasoning — Wait-free

```
(** Enqueue – should be called by only ONE thread *)
let enq q x =
  (* Check if queue is full *)
  if q.tail – q.head = q.capacity then
    raise Full;
  (* Write to the array *)
  q.items.(q.tail mod q.capacity) <- Some x;
  (* Advance tail *)
  q.tail <- q.tail + 1
```

```
(** Dequeue – should be called by only ONE thread *)
let deq q =
  (* Check if queue is empty *)
  if q.tail = q.head then
    raise Empty;
  (* Read from the array *)
  match q.items.(q.head mod q.capacity) with
  | None -> assert false  (* Should never happen *)
  | Some x ->
      (* Advance head *)
      q.head <- q.head + 1;
      x
```

Linearization order is order head and tail fields modified

More Reasoning — Wait-free

```
(** Enqueue – should be called by only ONE thread          ue – should be called by only ONE thread *)
let enq q x =
  (* Check if queue is full *)
  if q.tail – q.head = q.capacity then                      ck if queue is empty *)
    raise Full;                                             q.tail = q.head then
  (* Write to the array *)                                  raise Empty;
  q.items.(q.tail mod q.capacity)                          (* Read from the array *)
  (* Advance tail *)                                        match q.items.(q.head mod q.capacity) with
  q.tail <- q.tail + 1                                      | None -> assert false   (* Should never happen *)
                                                            | Some x ->
                                                                (* Advance head *)
                                                                q.head <- q.head + 1;
                                                                x
```

Remember that there is only one enqueuer and only one dequeuer

Linearization order is order head and tail fields modified

Finding linearisation points

- Identify one atomic step where the method "happens"

  – Critical section

  – Machine instruction

- Doesn't always work

  – Might need to define several different steps for a given method

  – We will see this phenomenon in future lectures

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

  - Legal sequential history **S**

  - ~~where $\rightarrow_G \subset \rightarrow_S$~~

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
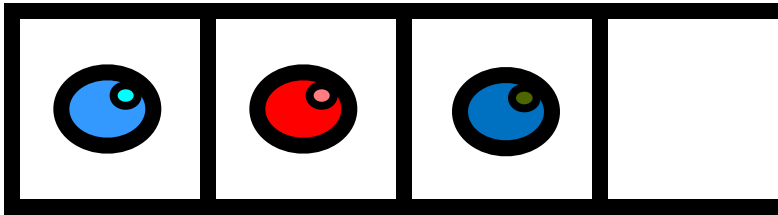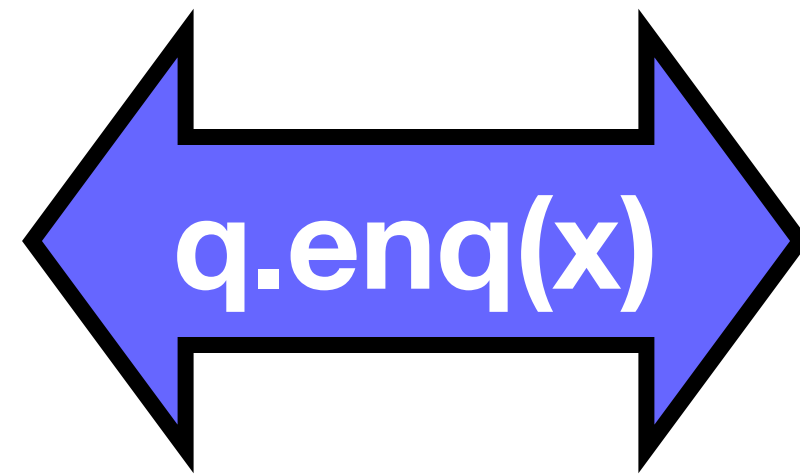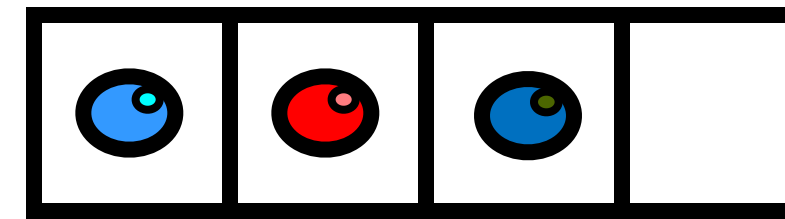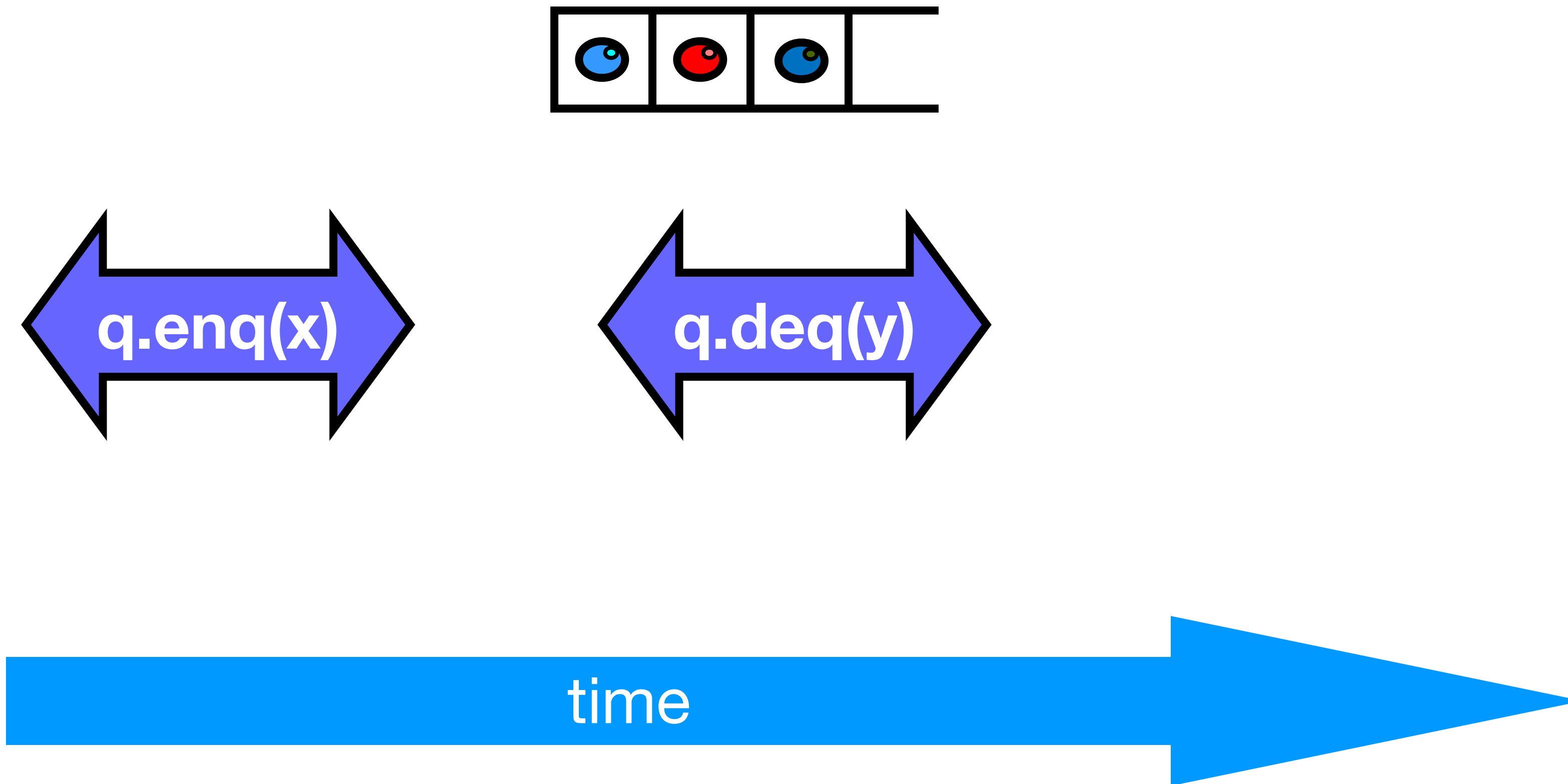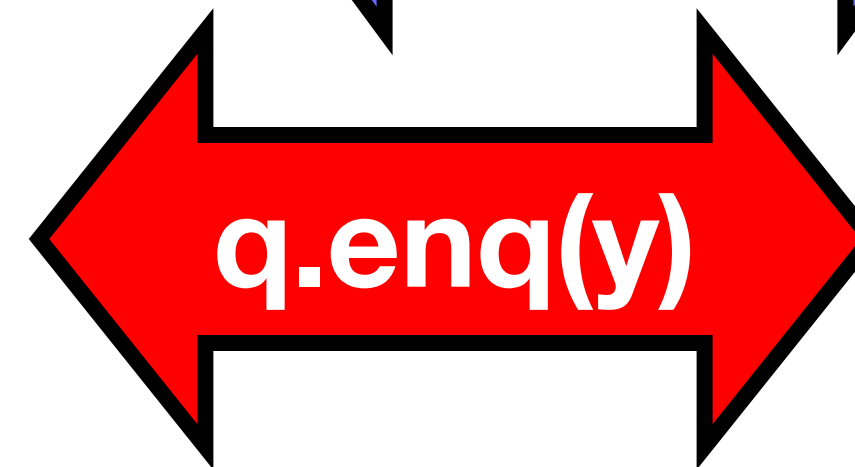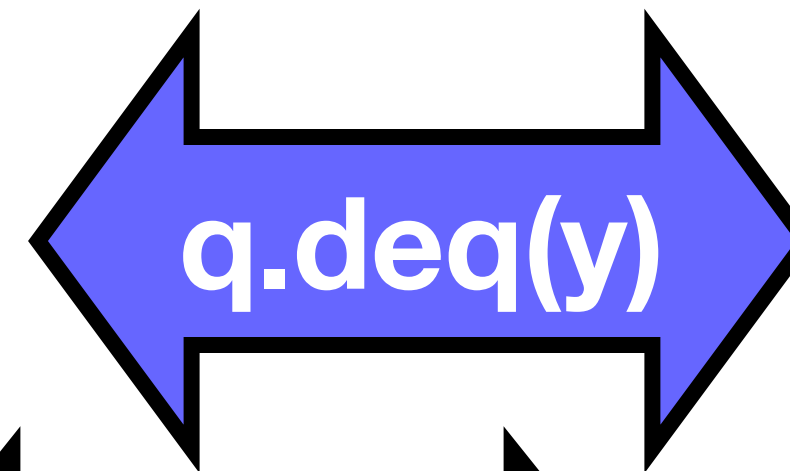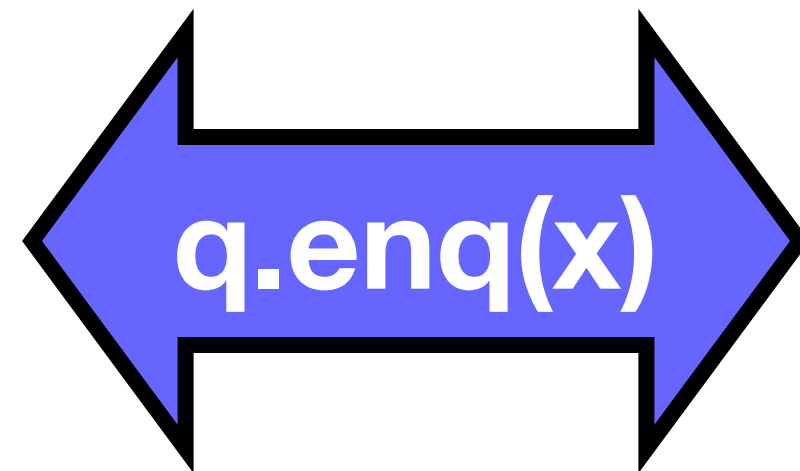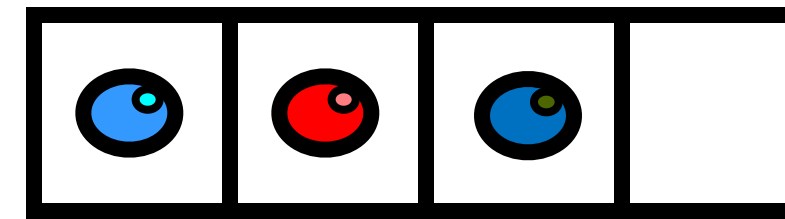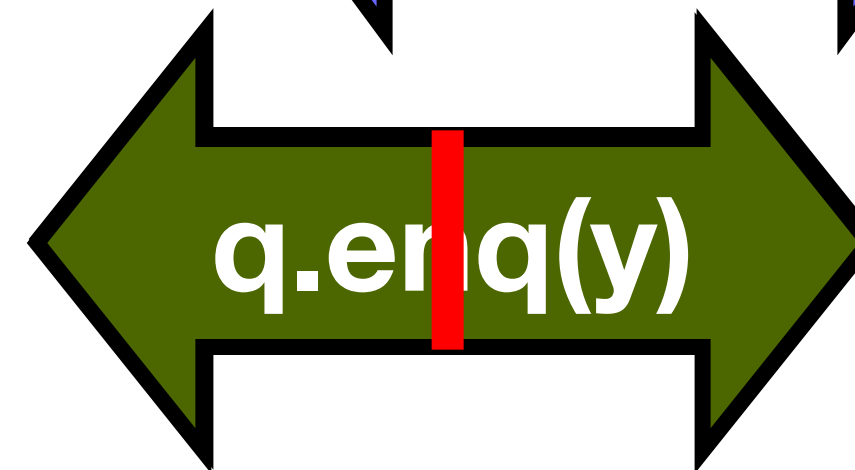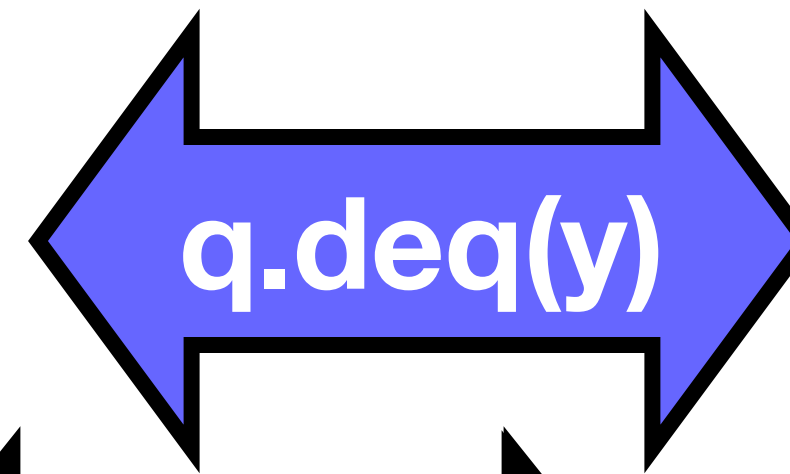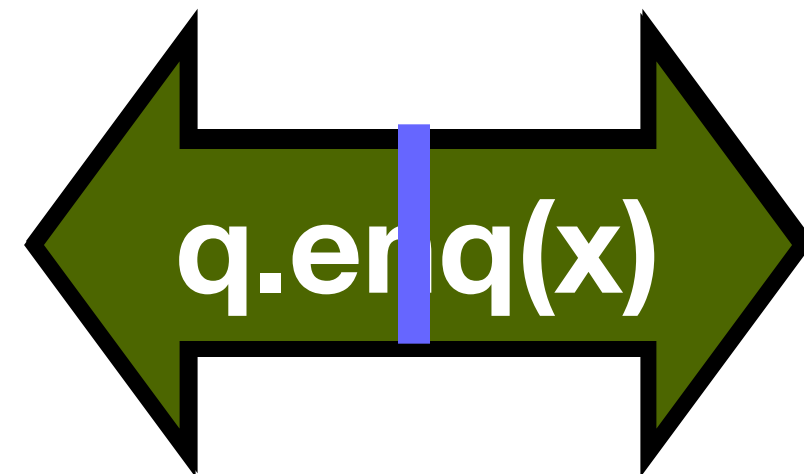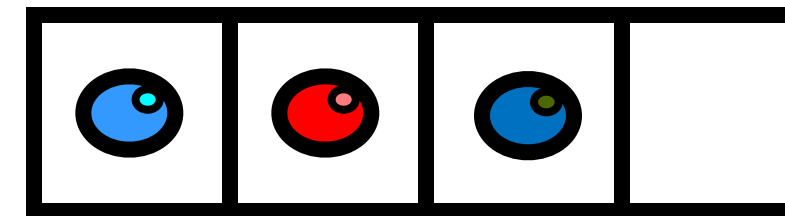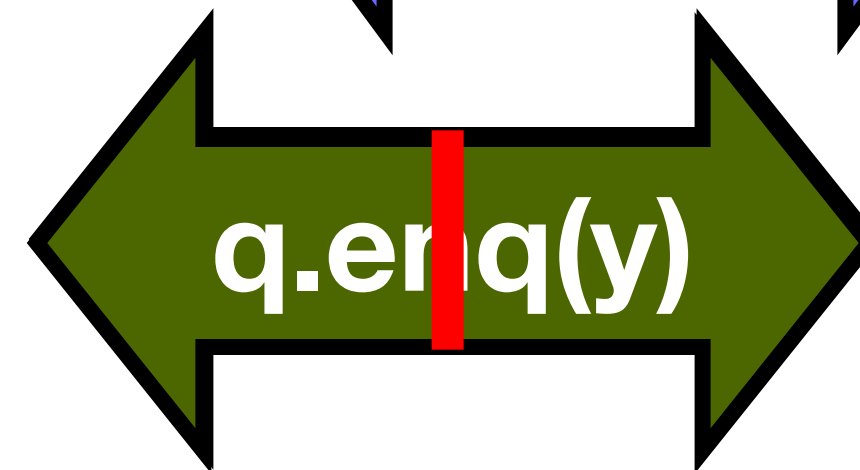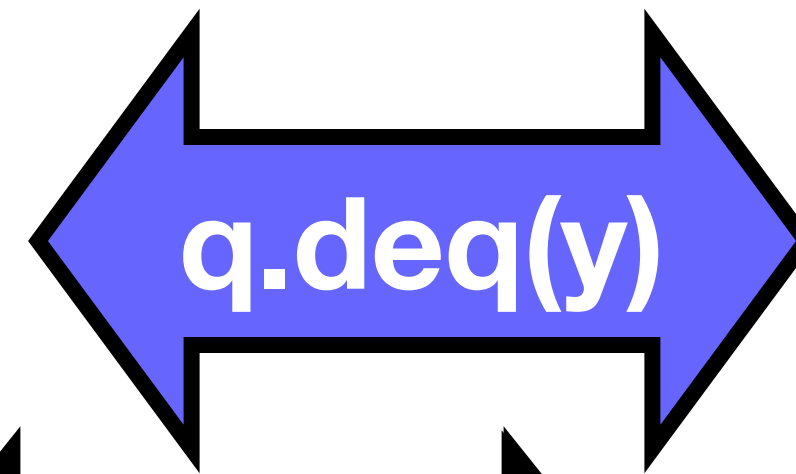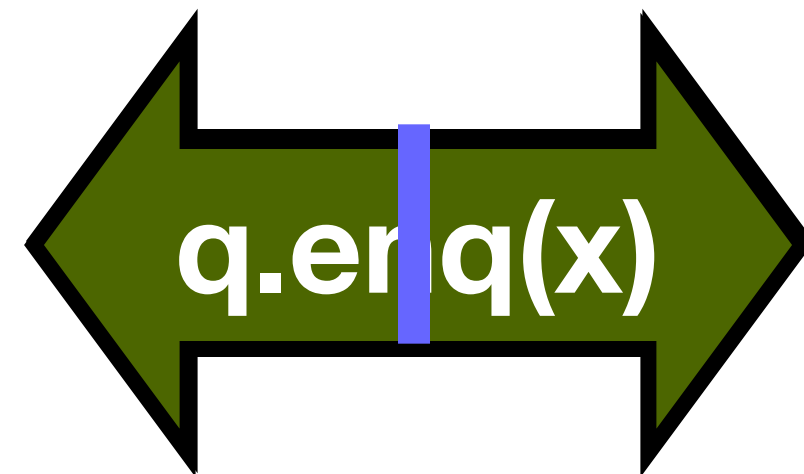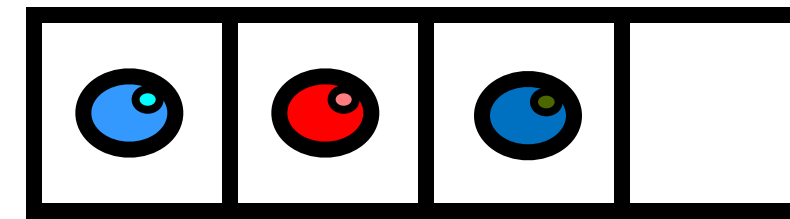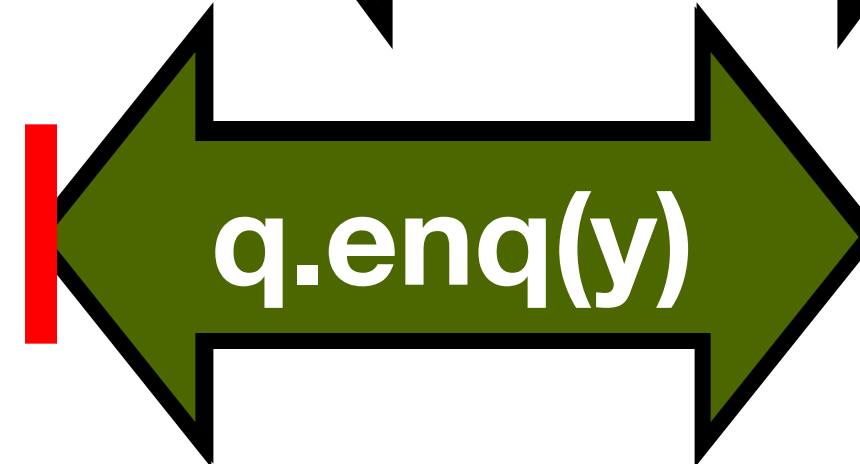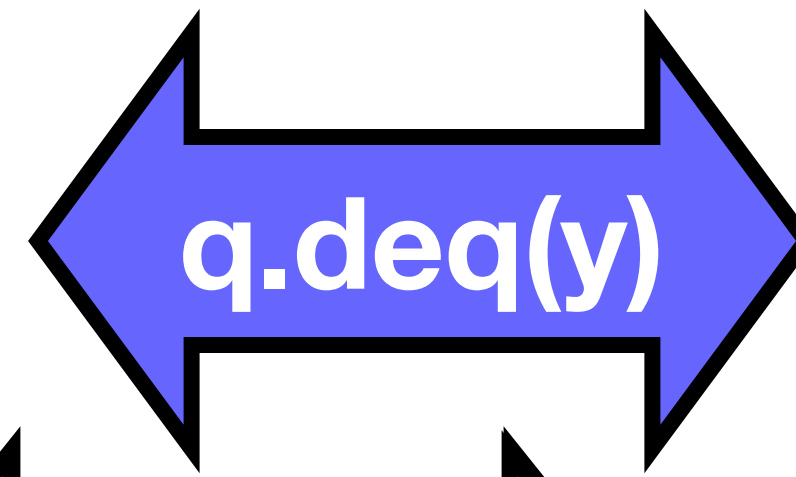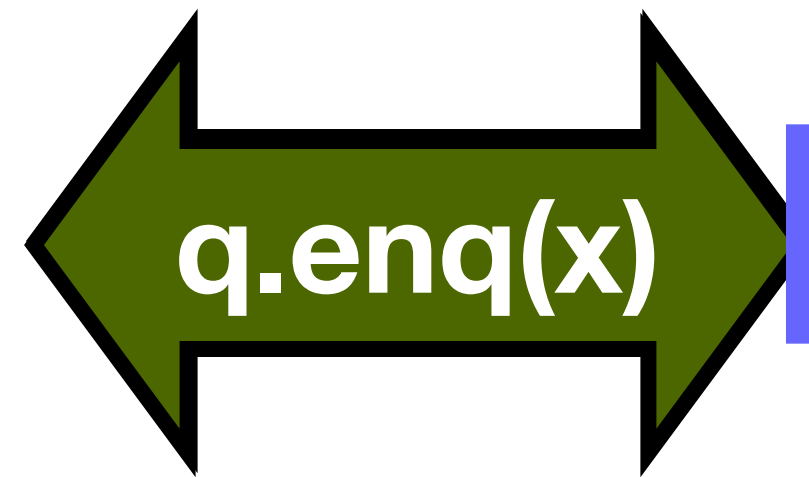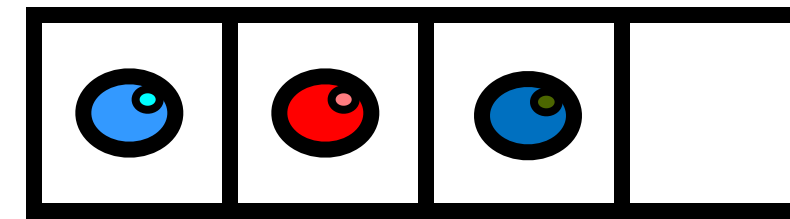
time

Example

Example

q.enq(x)

q.deq(y)

time

Example

q.enq(x)

q.deq(y)

q.enq(y)

time
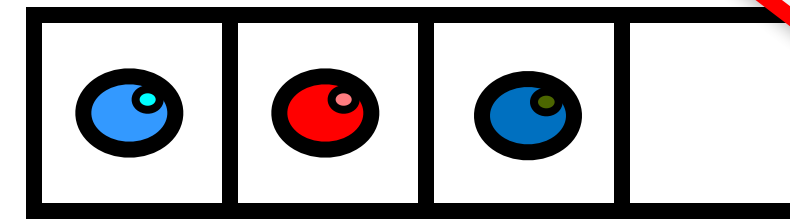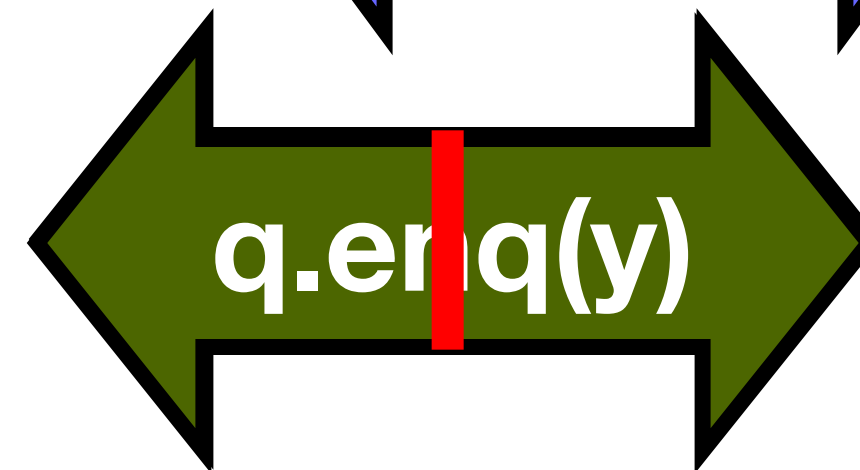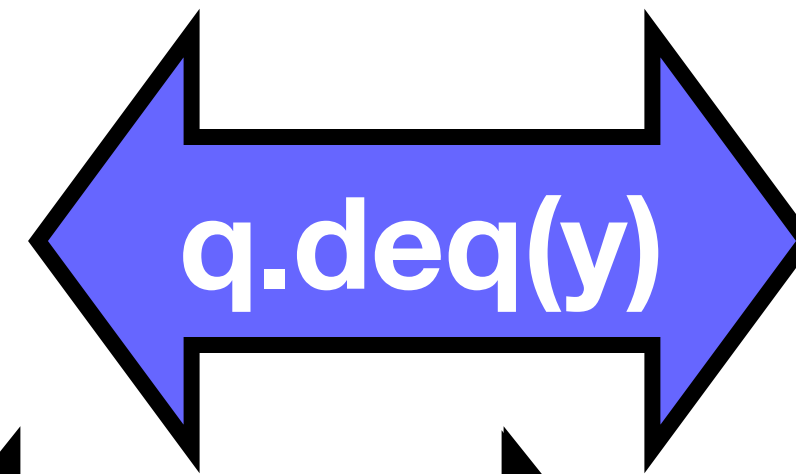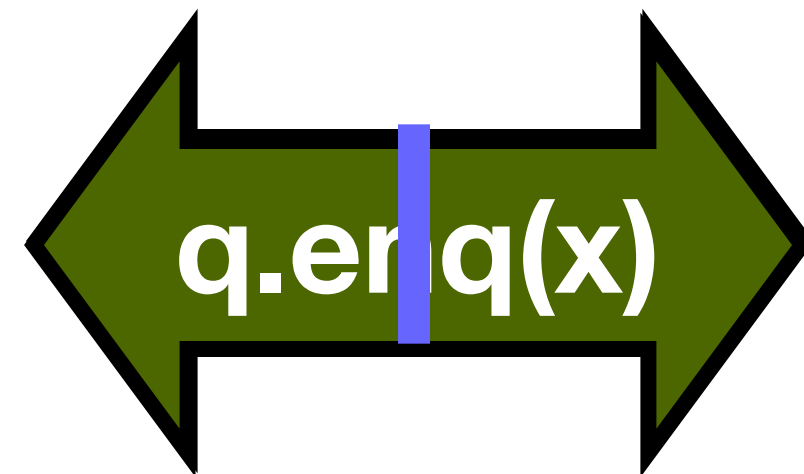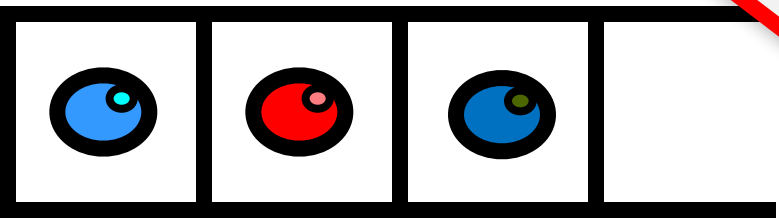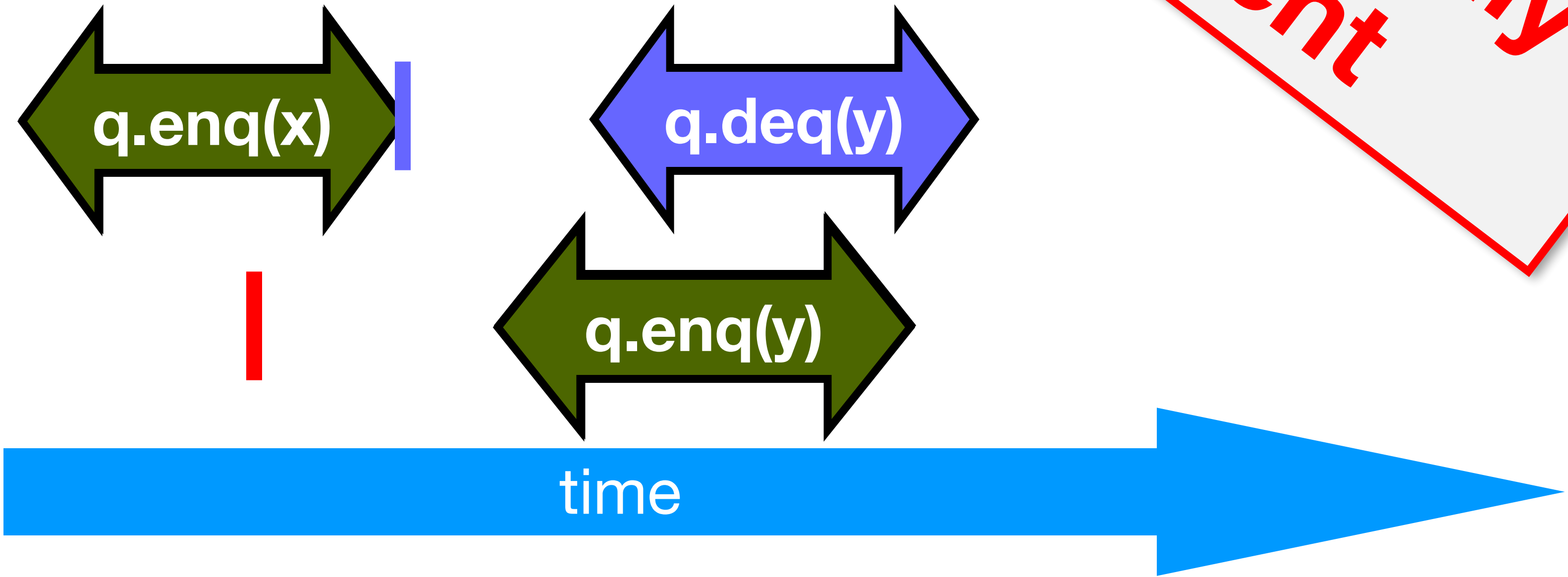
Example

Example

Theorem

Sequential Consistency is not Composable
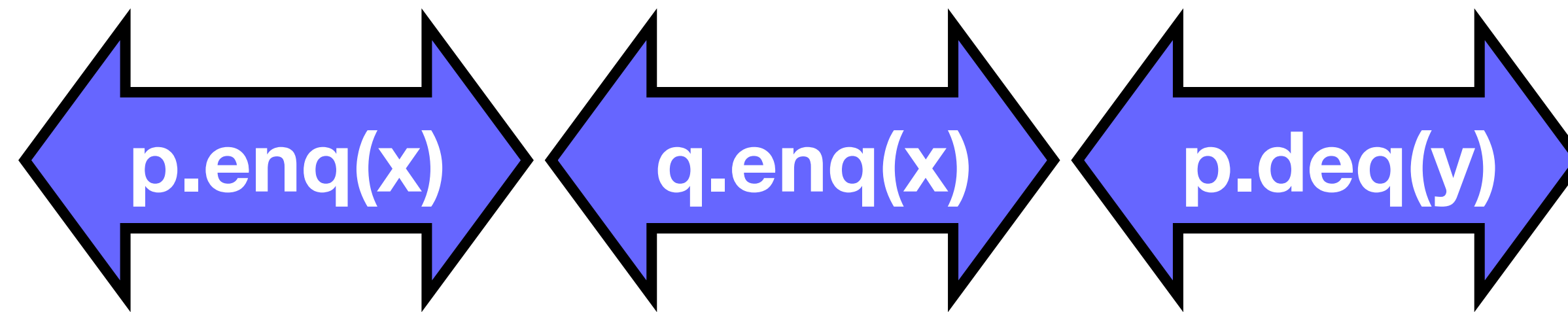
FIFO Queue Example

FIFO Queue Example

time
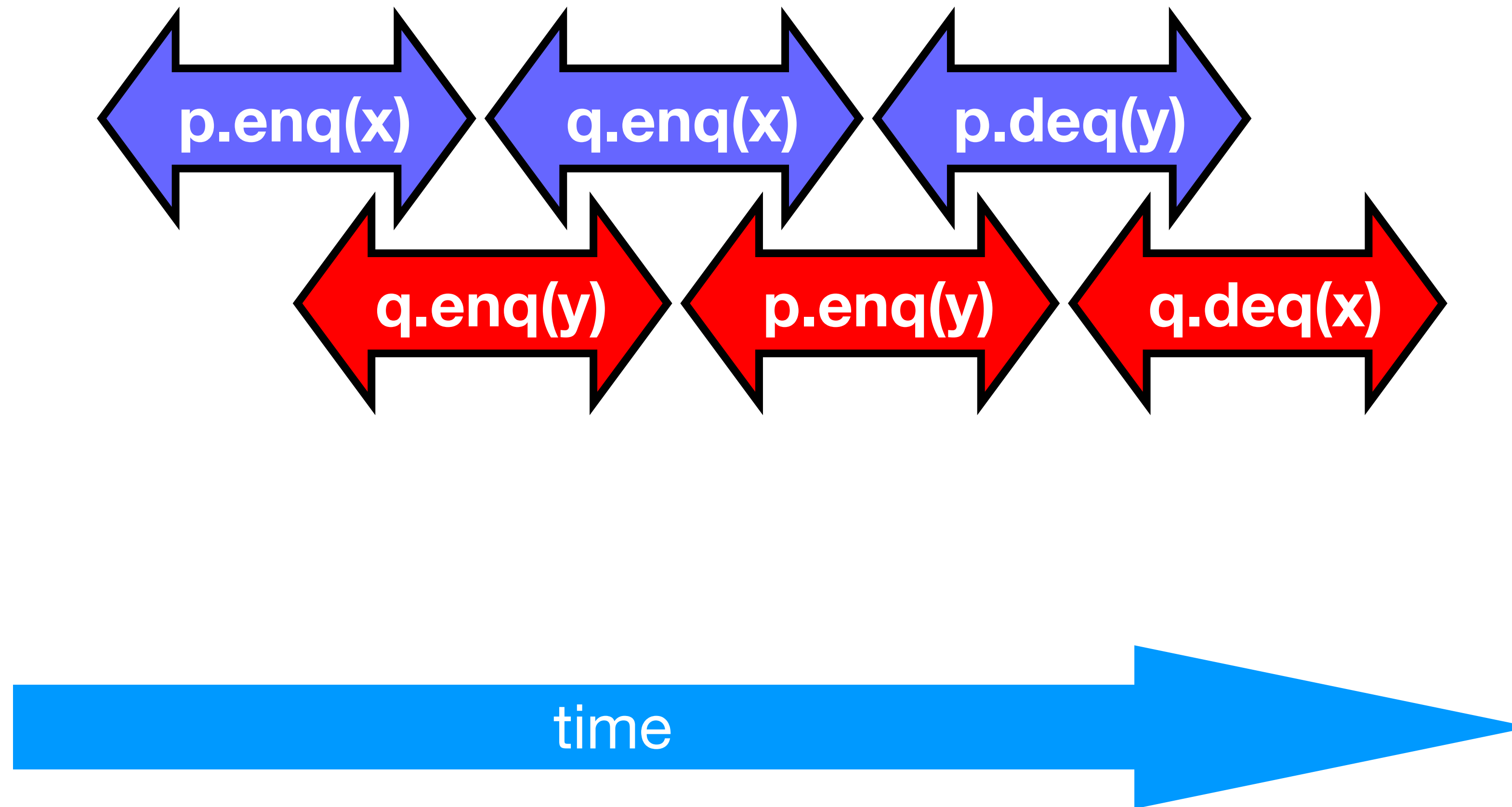
FIFO Queue Example
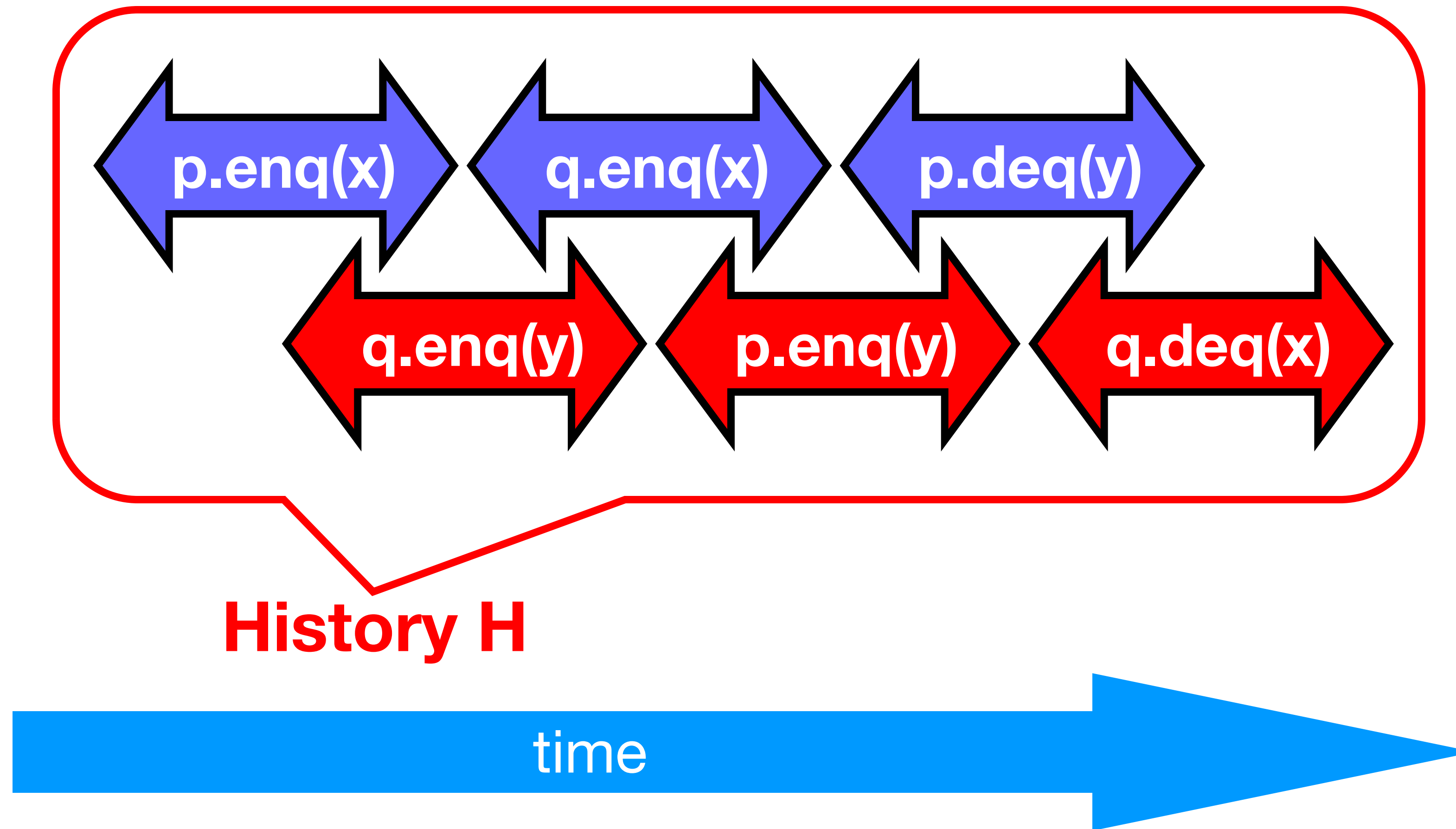
H|p Sequentially Consistent
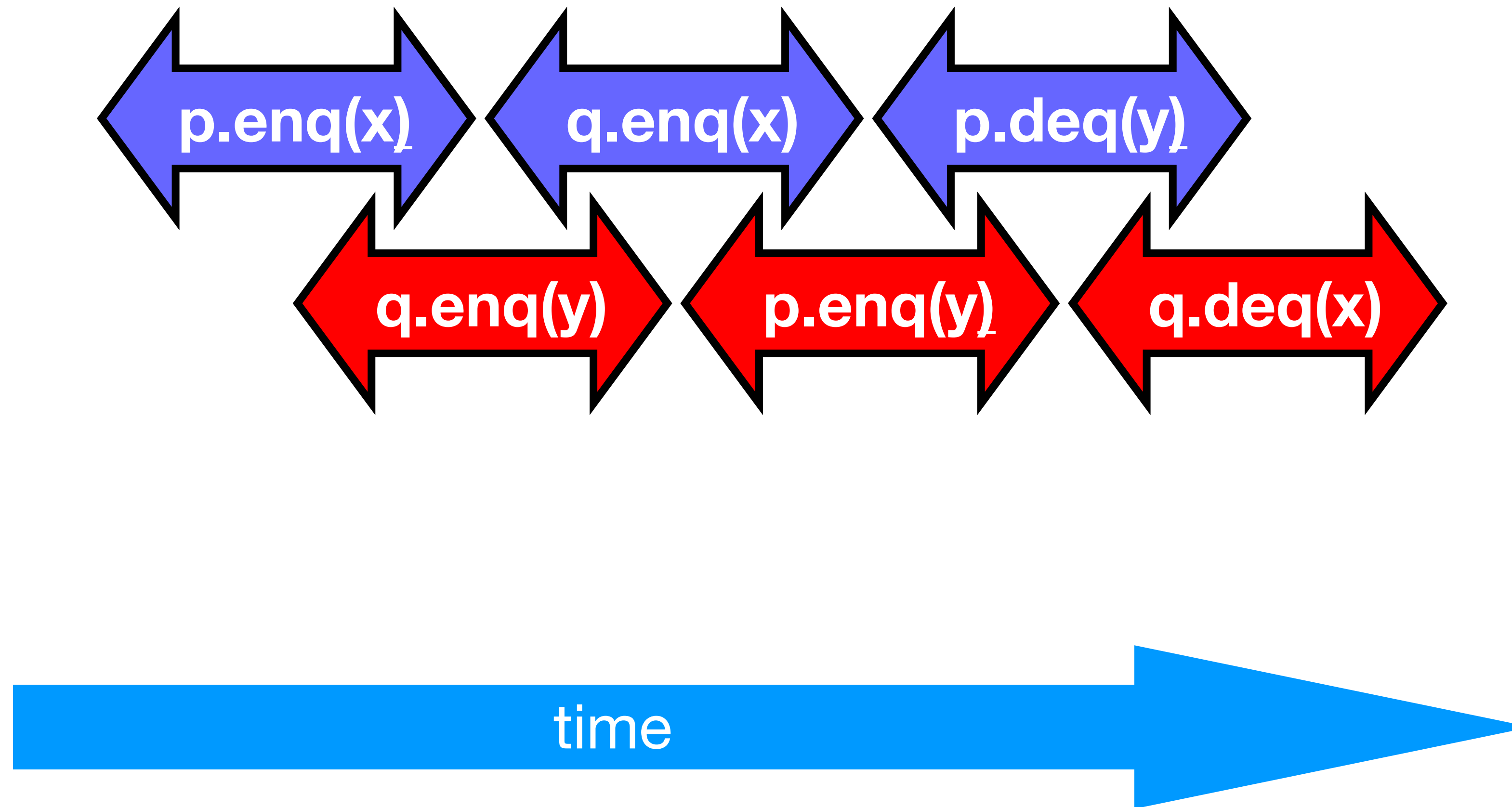
H|p Sequentially Consistent
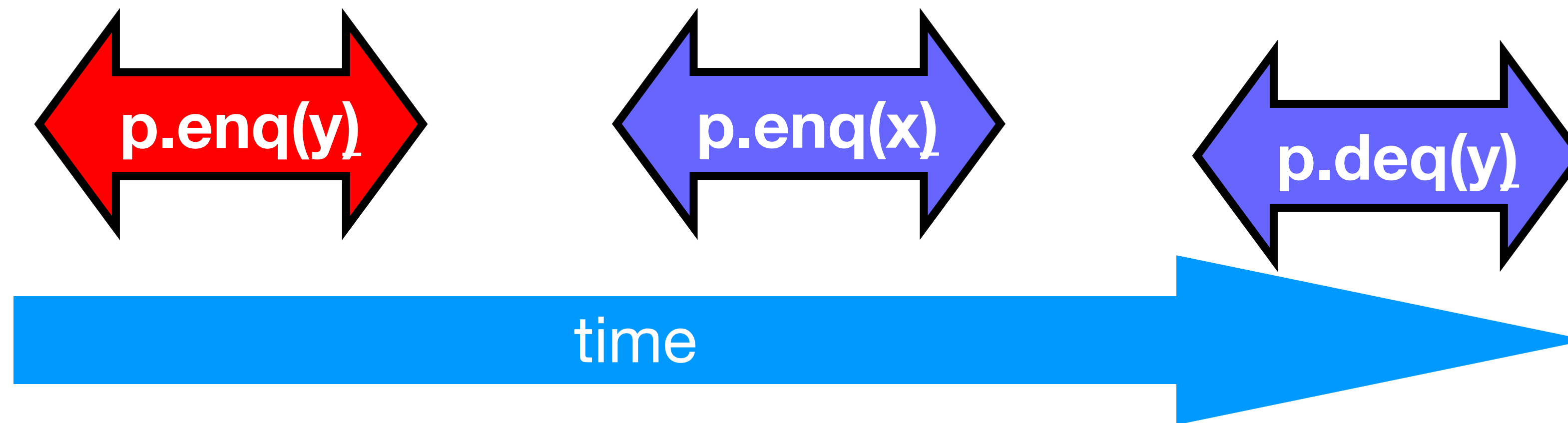
H|q Sequentially Consistent
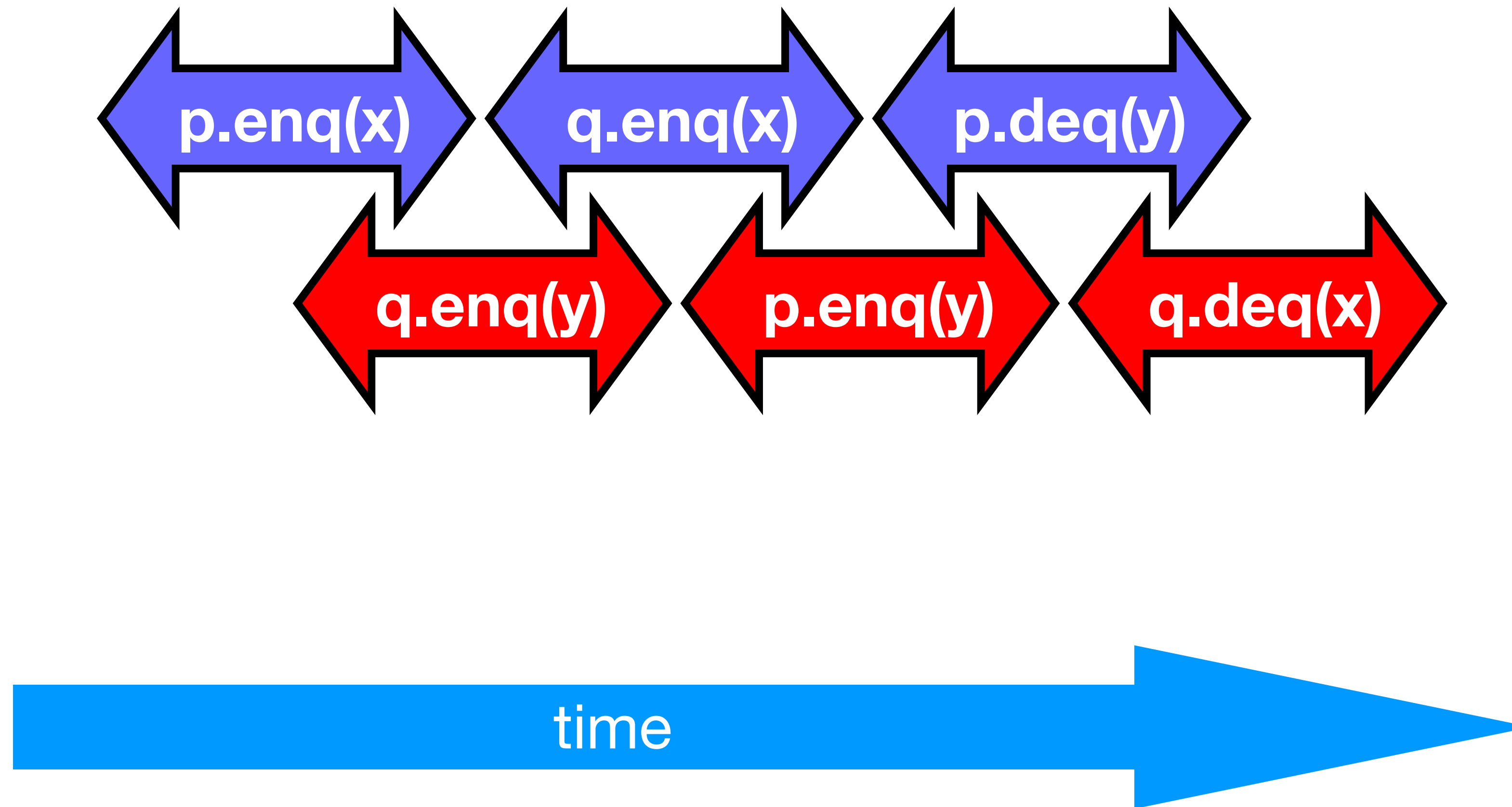
H|q Sequentially Consistent
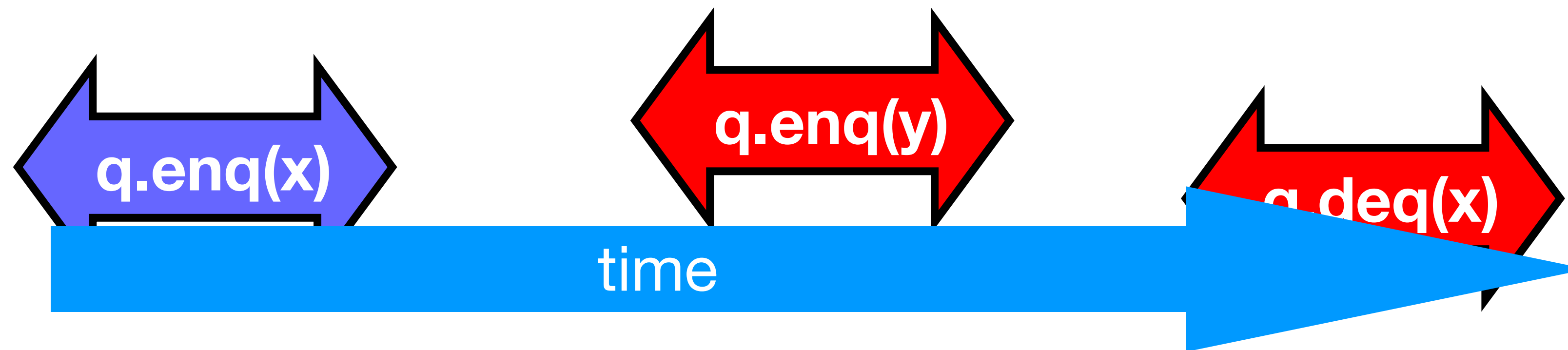
Ordering imposed by p
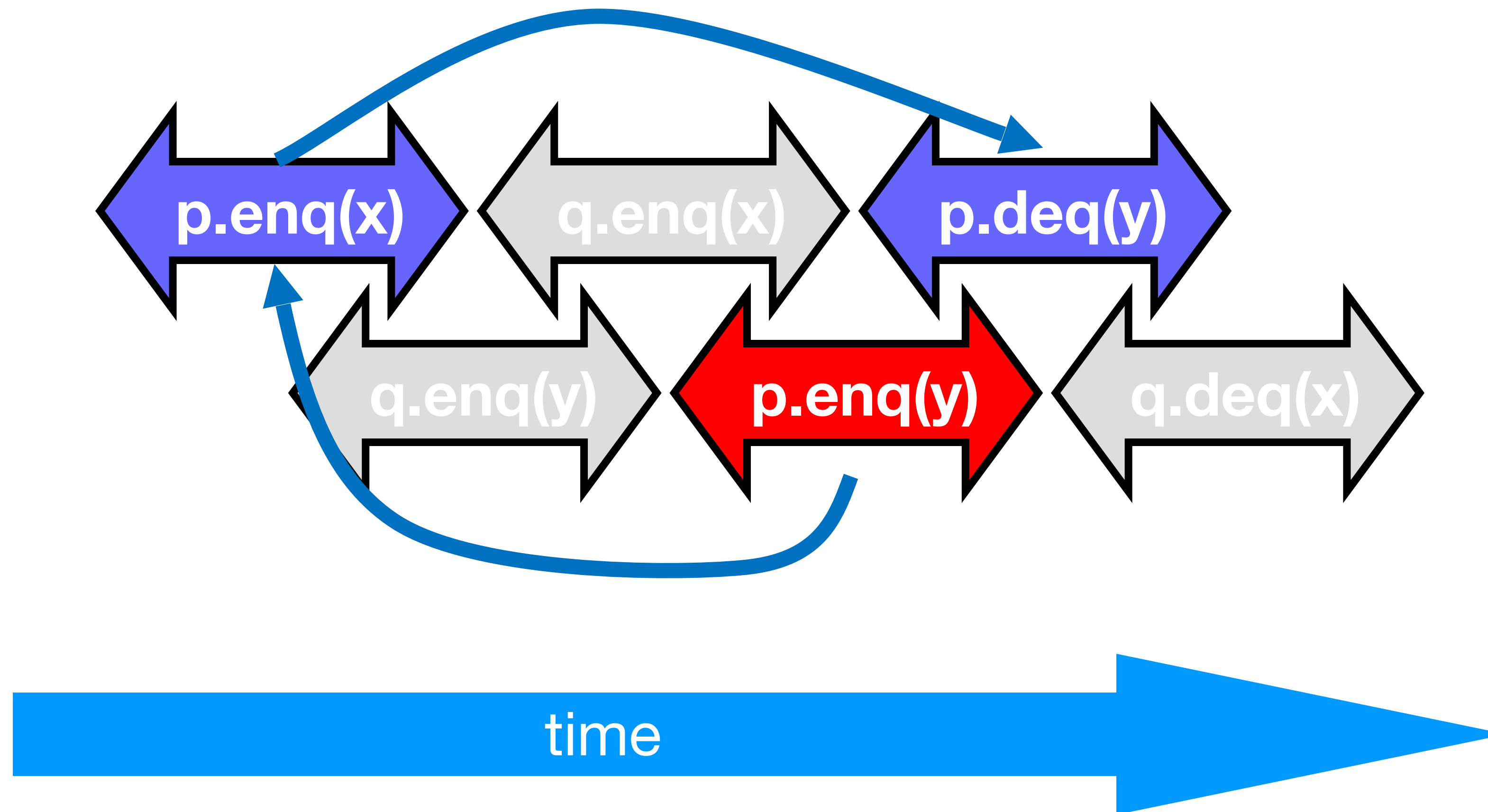
Ordering imposed by q
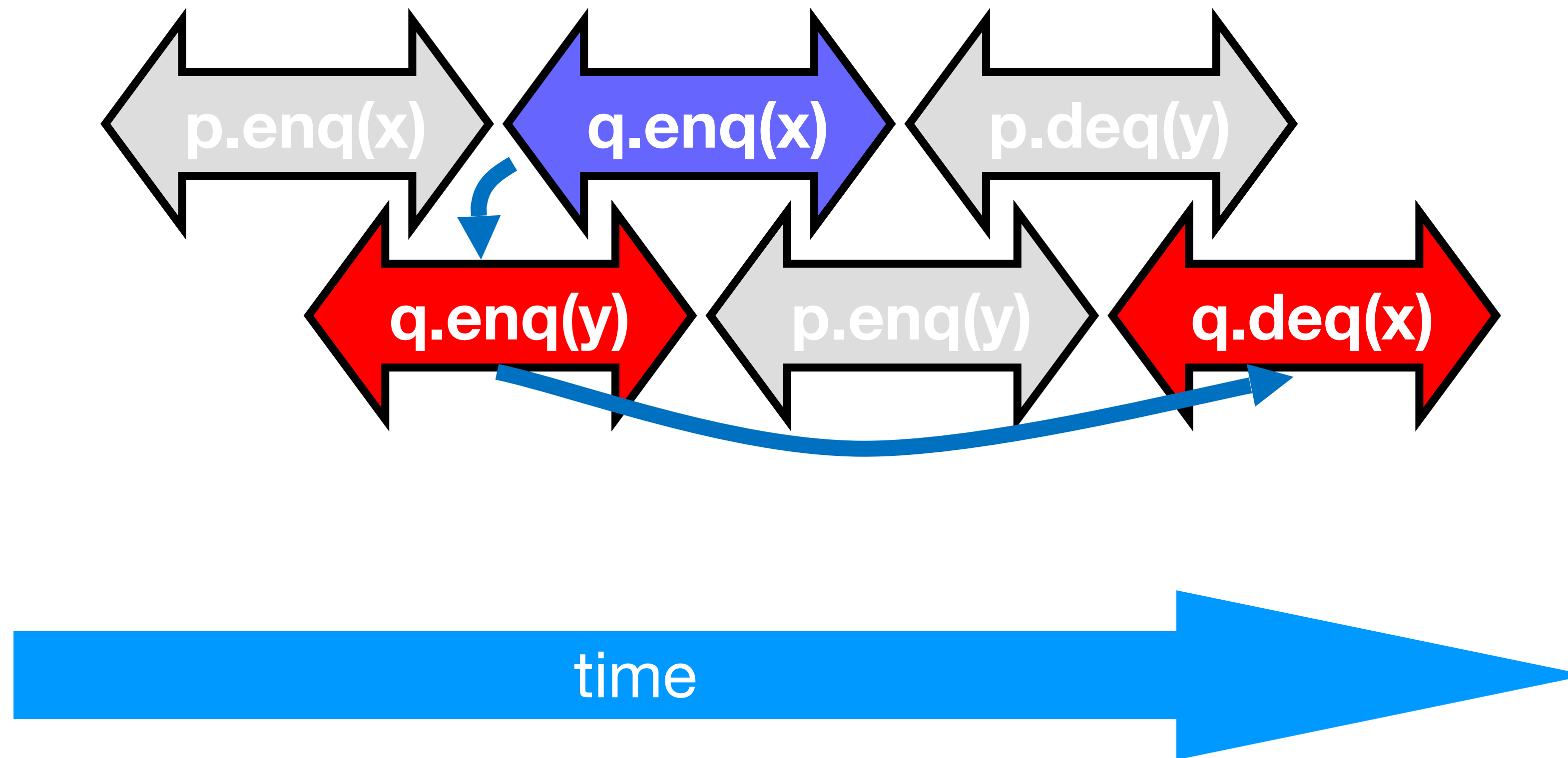
Ordering imposed by both
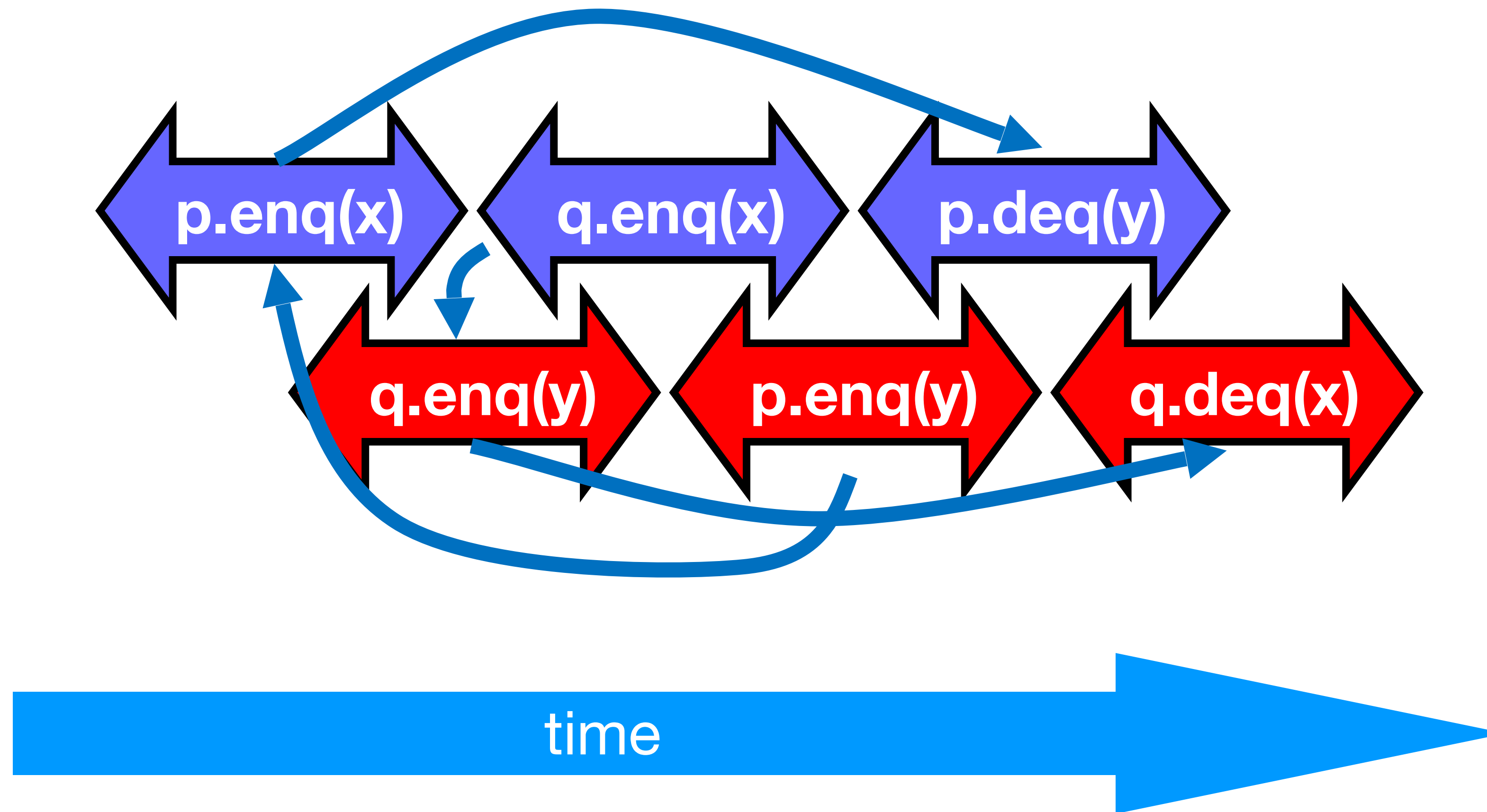
Combining orders
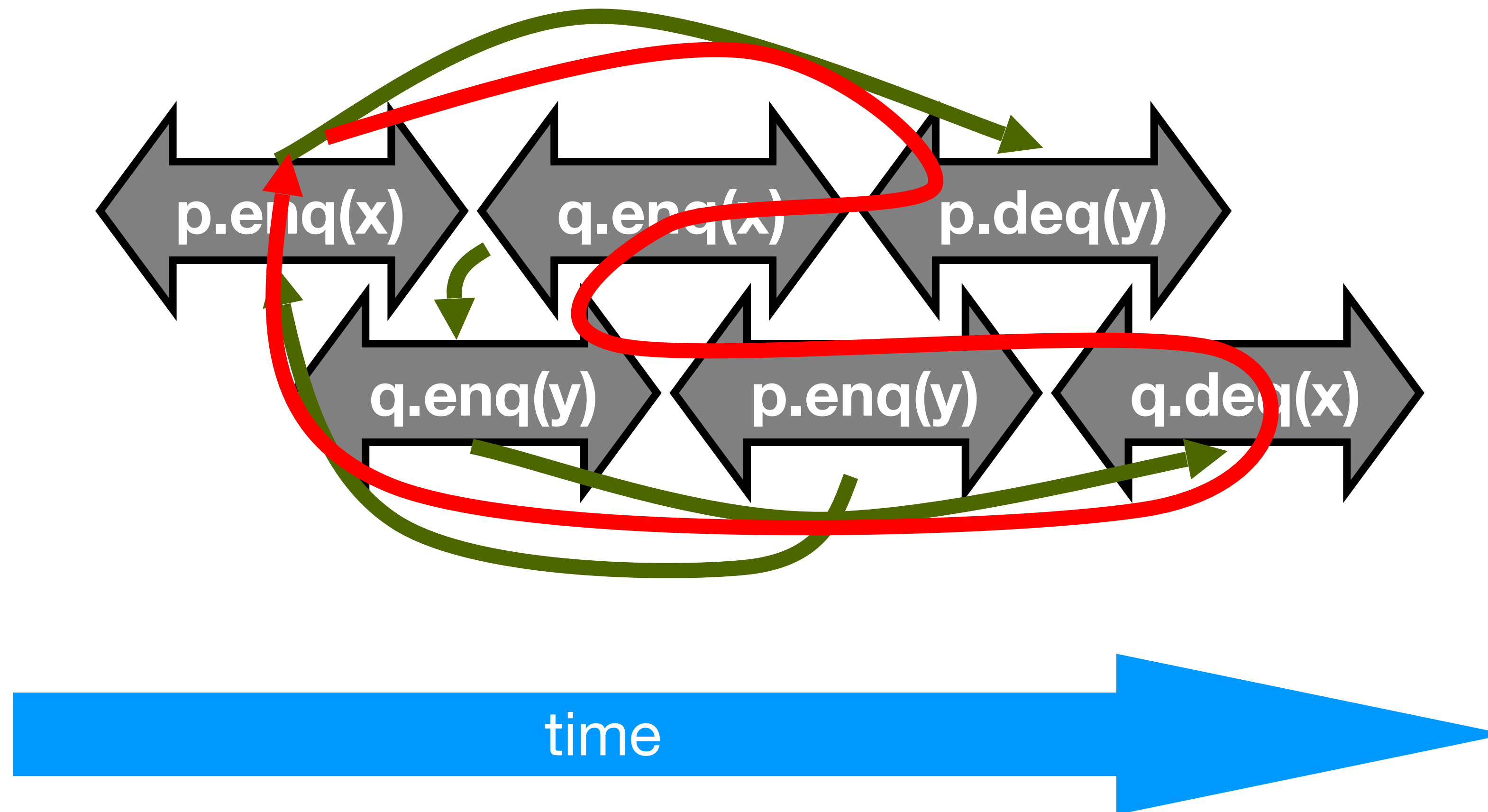
Concurrency Testing

- Linearizability and Sequential Consistency are good specifications for *testing* the correctness of concurrent data structures.

    - Any observed execution must match a sequential execution

    - Can be exploited for pragmatic testing

- See https://github.com/ocaml-multicore/multicoretests

qcheck-lin

- Checks for sequential consistency violations (despite what the name says)

- *Every sequential consistency violation is a linearizability violation*

- Check that the observed result of a parallel implementation can be observed with a sequential run

qcheck-lin

- Checks for sequential consistency violations (despite what the name says)

- *Every sequential consistency violation is a linearizability violation*

- Check that the observed result of a parallel implementation can be observed with a sequential run

Demo

qcheck-stm

- In *qcheck-lin,* what if the implementation is wrong for the sequential program itself?

  - We're only comparing equivalence.

  - Sequential run of a buggy implementation $\equiv$ Parallel run of a buggy implementation

    - *Is not useful!*

qcheck-stm

- In **qcheck-lin,** what if the implementation is wrong for the sequential program itself?

  - We're only comparing equivalence.

  - Sequential run of a buggy implementation ≡ Parallel run of a buggy implementation

    - *Is not useful!*

- qcheck-stm

  - Write a state-machine *model* of the concurrent object

  - Compare the sequential and parallel executions of the implementation against the state machine model

  - *More work!*

qcheck-stm

- In ***qcheck-lin,*** what if the implementation is wrong for the sequential program itself?

    - We're only comparing equivalence.

    - Sequential run of a buggy implementation ≡ Parallel run of a buggy implementation

        - ***Is not useful!***

- qcheck-stm

    - Write a state-machine ***model*** of the concurrent object

    - Compare the sequential and parallel executions of the implementation against the state machine model

    - ***More work!***

Demo

Summary

- ***Linearizability***

  – The operation takes effect instantaneously between the invocation and the response

  – Uses sequential specification, locality implies composability

- ***Sequential Consistency***

  – Linearizability without real-time ordering

  – Not composable

  – Harder to work with

  – Useful to reason about hardware models (next lecture)

- We will use ***linearizability*** as our consistency condition for reasoning about objects

Progress

- We saw an implementation whose methods were lock-based (deadlock-free)

- We saw an implementation whose methods did not use locks (lock-free)

- How do they relate?

Progress Conditions

- **Deadlock-free:** <u>some</u> thread trying to acquire the lock eventually succeeds.

- **Starvation-free:** <u>every</u> thread trying to acquire the lock eventually succeeds.

- **Lock-free:** <u>some</u> thread calling a method eventually returns.

- **Wait-free:** <u>every</u> thread calling a method eventually returns.

Progress Conditions

- ***Deadlock-free:*** <u>some</u> thread trying to acquire the lock eventually succeeds.

- ***Starvation-free:*** <u>every</u> thread trying to acquire the lock eventually succeeds.

- ***Lock-free:*** <u>some</u> thread calling a method eventually returns.

- ***Wait-free:*** <u>every</u> thread calling a method eventually returns.

	Non-Blocking	Blocking
Everyone makes progress	**Wait-free**	**Starvation-free**
Someone makes progress	**Lock-free**	**Deadlock-free**

Progress Conditions

- ***Deadlock-free:*** <u>some</u> thread trying to acquire the lock eventually succeeds.

- ***Starvation-free:*** <u>every</u> thread trying to acquire the lock eventually succeeds.

- ***Lock-free:*** <u>some</u> thread calling a method eventually returns.

- ***Wait-free:*** <u>every</u> thread calling a method eventually returns.

	Non-Blocking	Blocking
Everyone makes progress	**Wait-free**	**Starvation-free**
Someone makes progress	**Lock-free**	**Deadlock-free**

We will look at linearizable blocking and non-blocking implementations of objects.