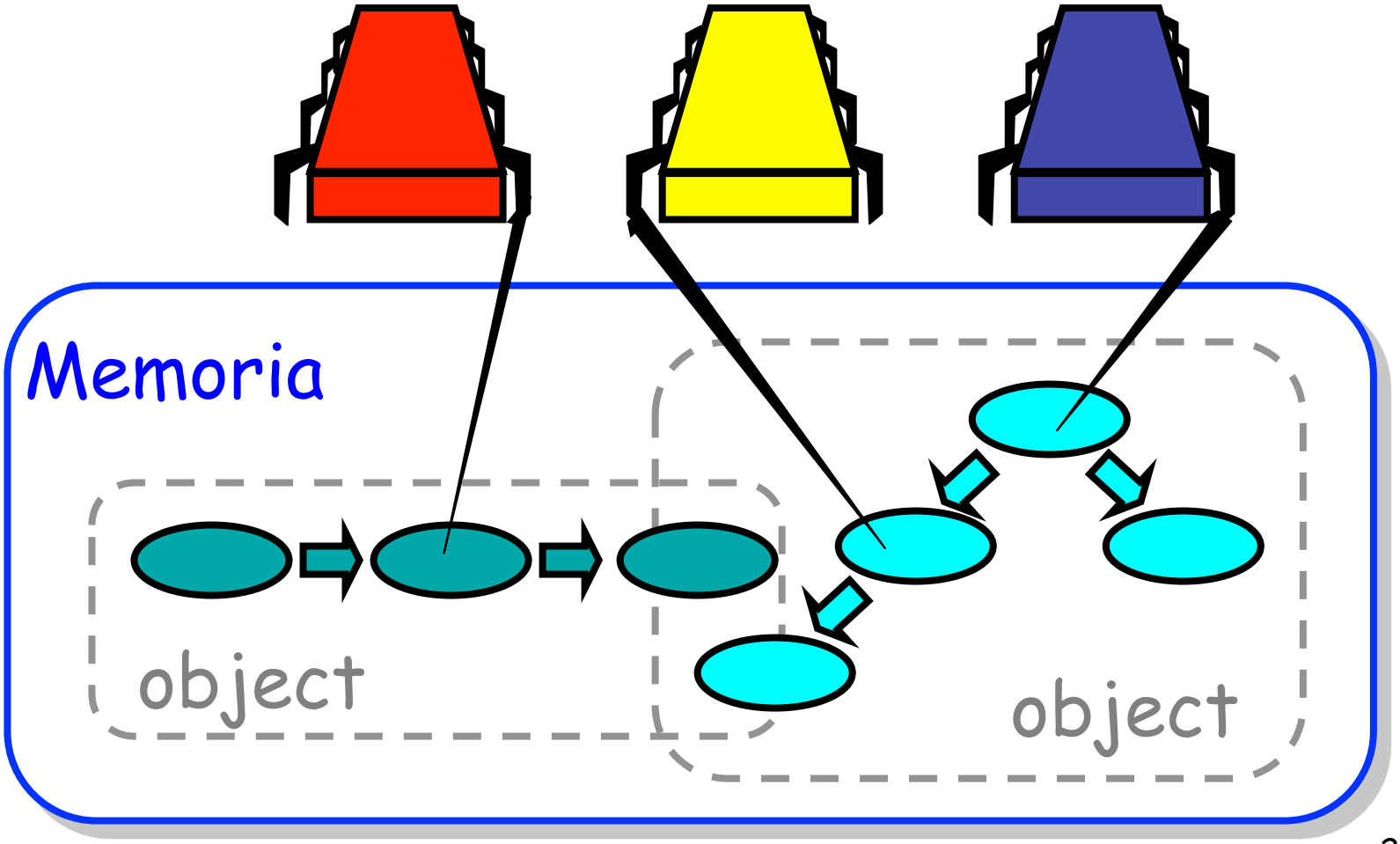
Concurrent Objects

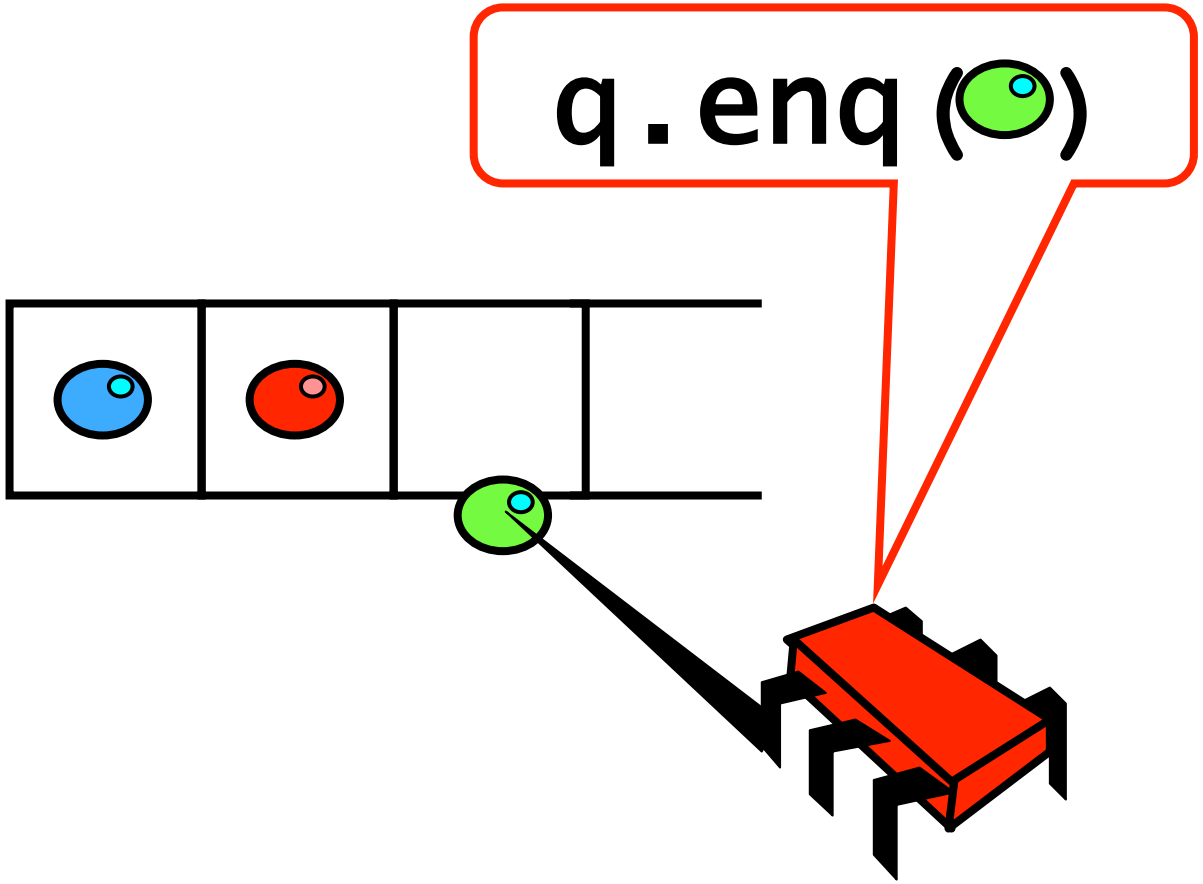
La concorrenza



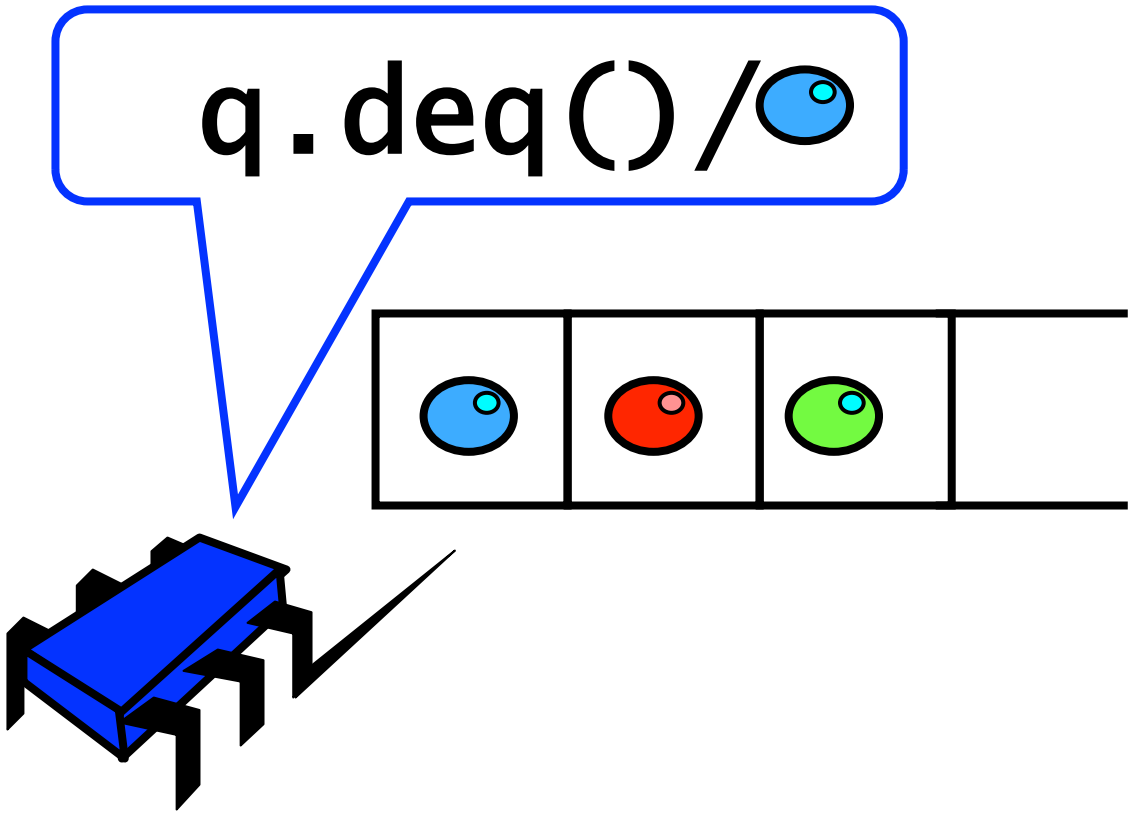
Il problema che vogliamo affrontare

- Cosa e' un oggetto concorrente?
  - In quale modo lo **descriviamo?**
  - In quale modo lo **implementiamo**?
  - In quale modo dimostriamo **la correttezza**?

FIFO Queue: Enqueue Method



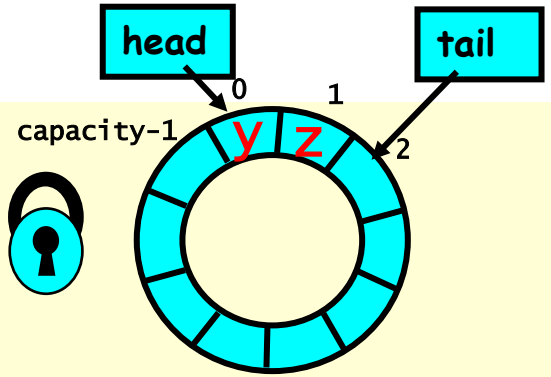
FIFO Queue: Dequeue Method



Lock-Based Queue

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

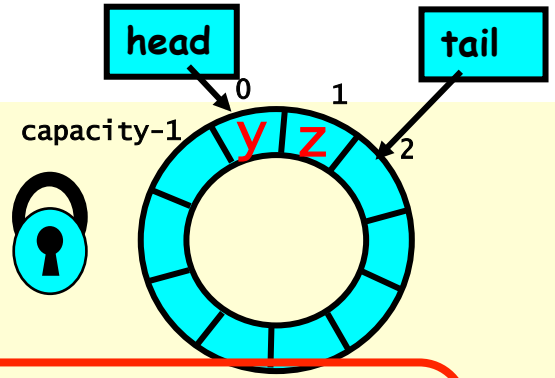
Lock-Based Queue




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

Queue fields protected by single shared lock

Lock-Based Queue

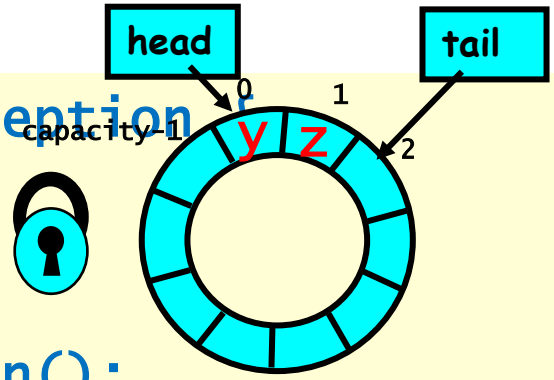



```
class LockBasedQueue<T> {
  int head, tail;
  T[] items;
  Lock lock;
  public LockBasedQueue(int capacity) {
    head = 0; tail = 0;
    lock = new ReentrantLock();
    items = (T[]) new Object[capacity];
}
```

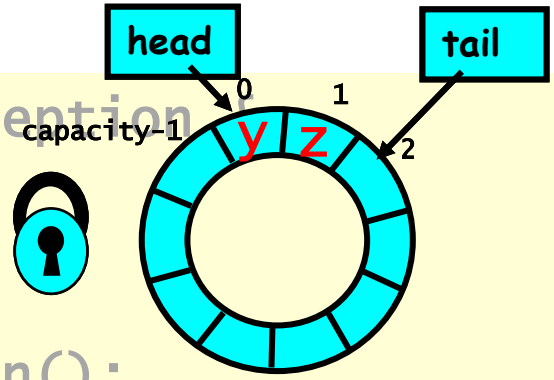
Initially head = tail

Implementation: Deq

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



Implementation: Deq

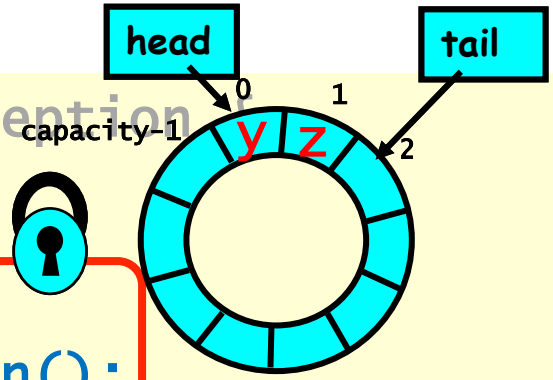



```
public T deq() throws EmptyException {
  lock.lock();
  try {
    if (tail == head)
       throw new EmptyException();
    T x = items[head % items.length];
    head++;
    return x;
  } finally {
    lock.unlock();
  }
}
```

Method calls mutually exclusive

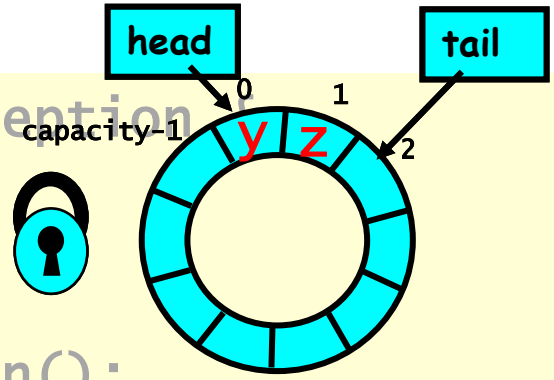
Implementation: Deq

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



If queue empty
throw exception

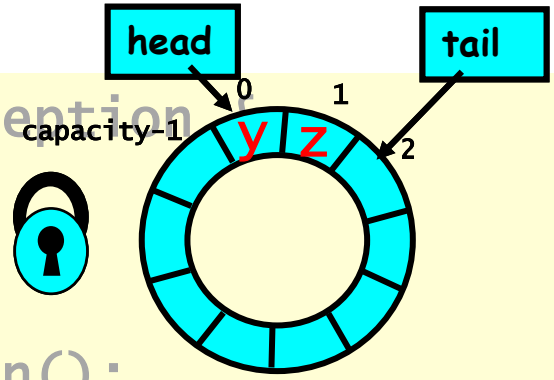
Implementation: Deq




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

Queue not empty: remove item and update head

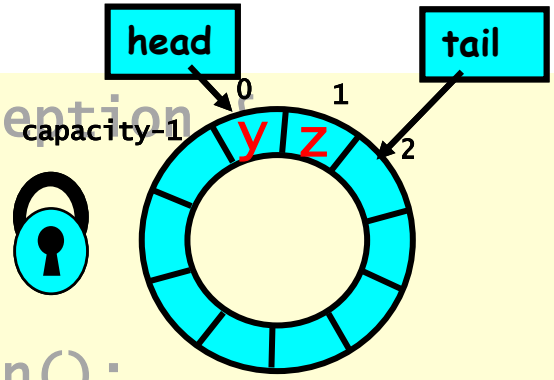
Implementation: Deq




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

Return result

Implementation: Deq




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

Release lock no matter what!

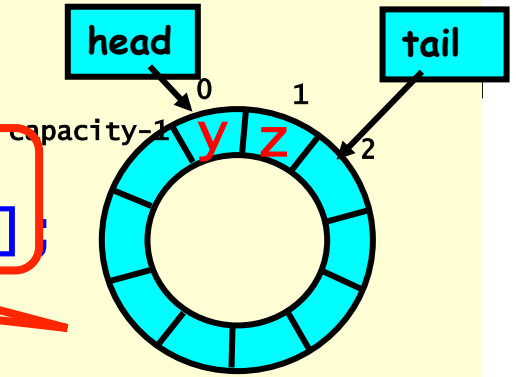
Implementation: Deq

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

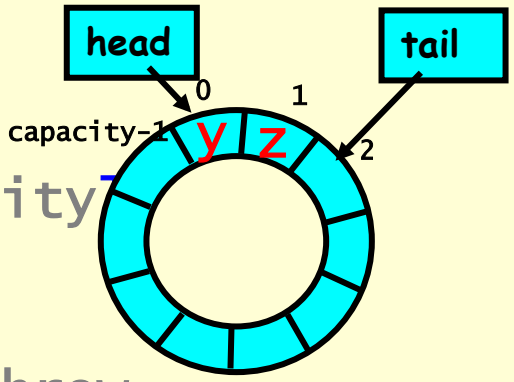
E' corretta? Le modifiche avvengono in mutua esclusione...

```
public class WaitFreeQueue {

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

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

```
public class WaitFreeQueue {

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

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



```
public class WaitFreeQueue {

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

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



Le modifiche non sono mutuamente esclusive

Progresso

- In un contesto concorrente si deve specificare sia le proprieta' di invarianza (safety) che le proprieta' di liveness
- Correttezza
  - Quando una implementazione e' corretta
  - Le condizioni che garantiscono il progresso

Oggetti sequenziali

- Ogni astrazione ha un proprio ***state***
  - Le variabili di istanza ***fields***
  - Queue: vettore di items
- Ogni astrazione possiede dei metodi ***methods***
  - Descrivono come operare sullo statep
  - Queue: **enq** e **deq**

Specifica

- (precondition)
  - Prima di invocare un metodo l'oggetto e' nello stato,
- (postcondition)
  - Il metodo modifica lo stato oppure solleva una eccezione.

Dequeue

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

Pre – Post Conditions Dequeue

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

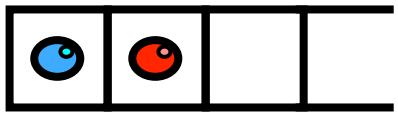
Sequenziale

- Le interazioni tra i metodi sono catturate tramite effetti layerali sullo stato degli oggetti
  - Invariante di rappresentazione a questo serve!!!
- Ogni metodo e' descritto singolarmente
- Refinement: possiamo aggiungere metodi senza modificare la descrizione dei vecchi metodi.

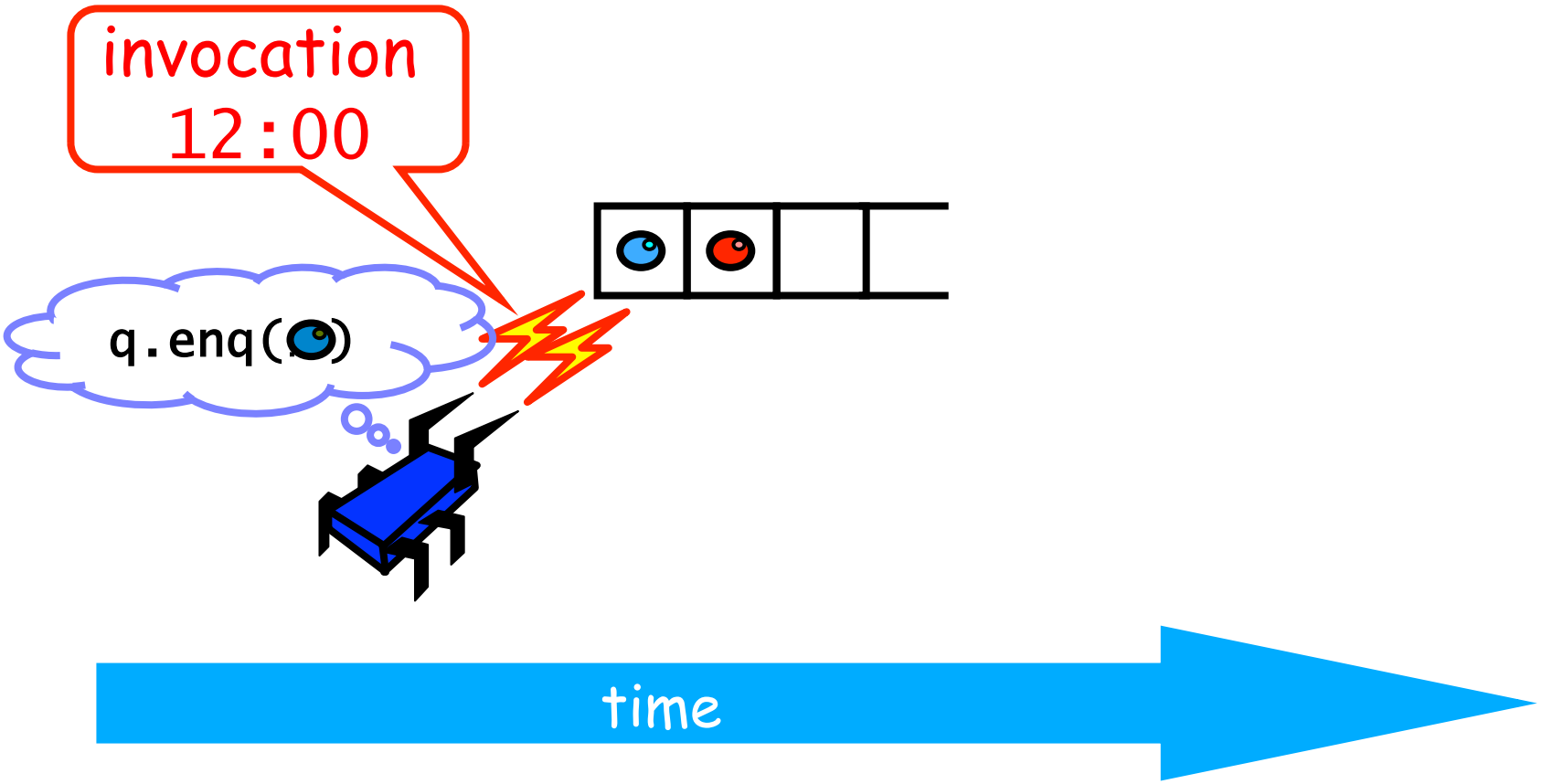
Cosa cambia con la concorrenza?

- Metodi?
- La descrizione del metodo?
- Refinement?

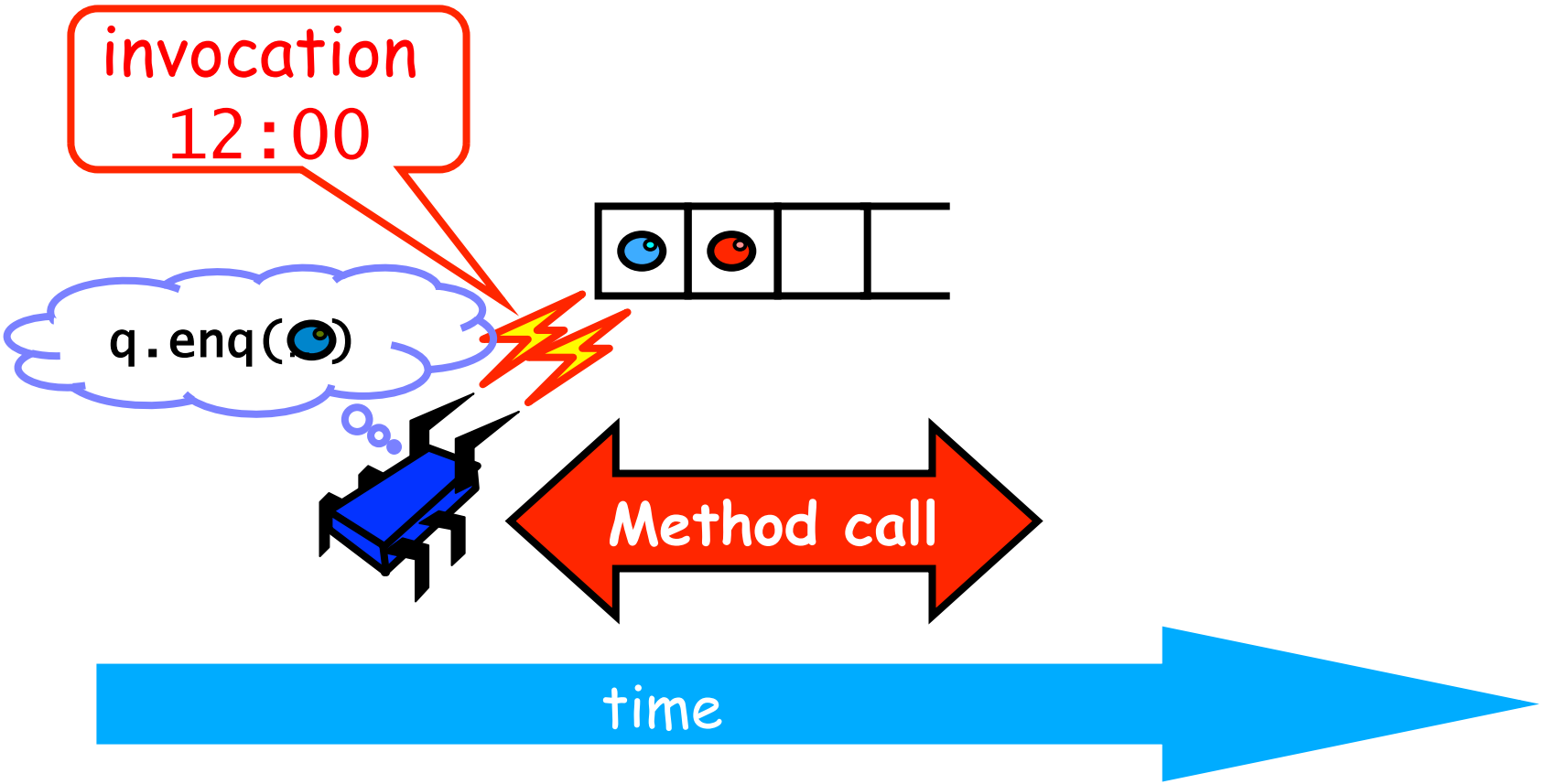
"Methods Take Time"



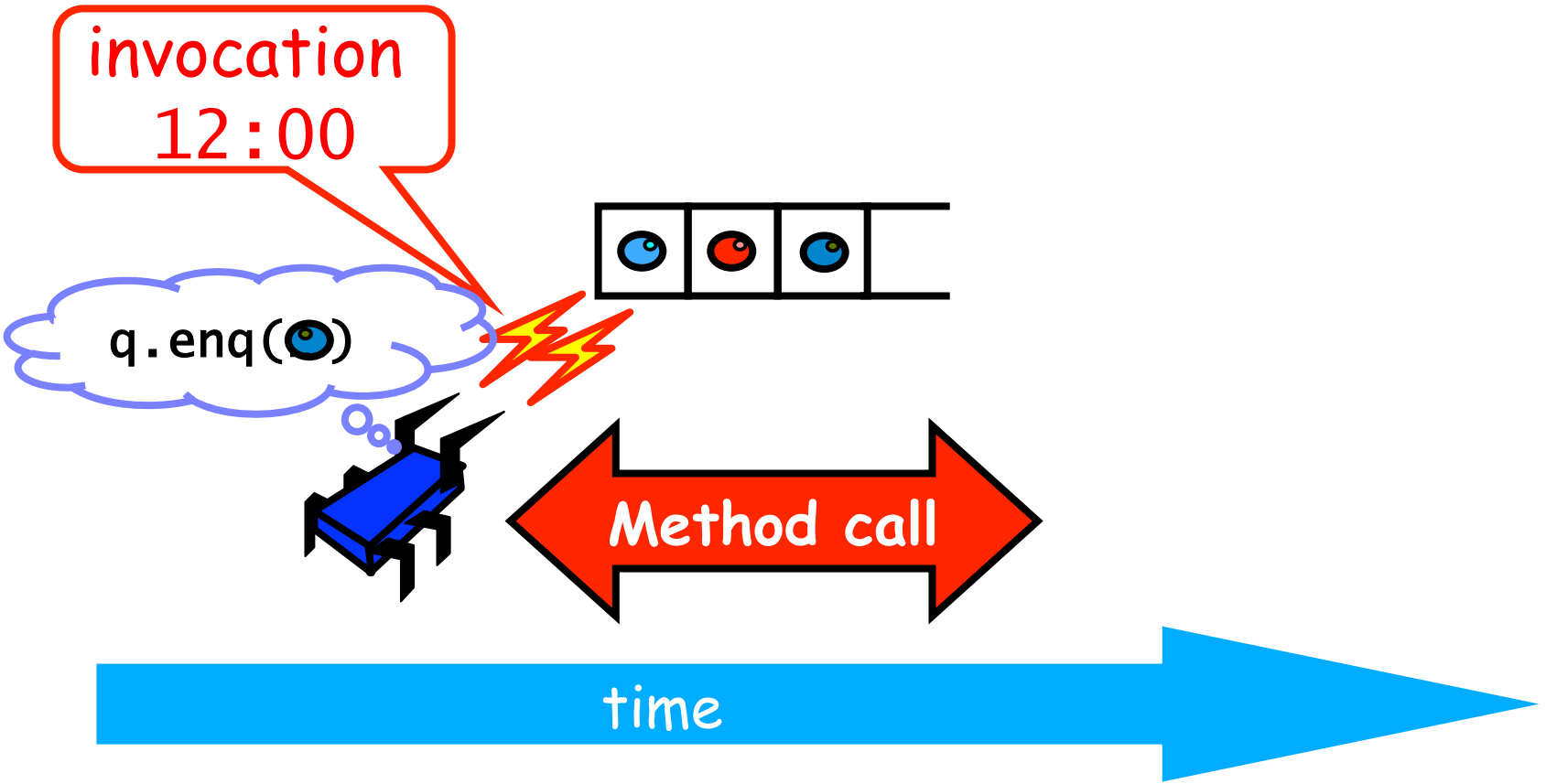
Methods Take Time



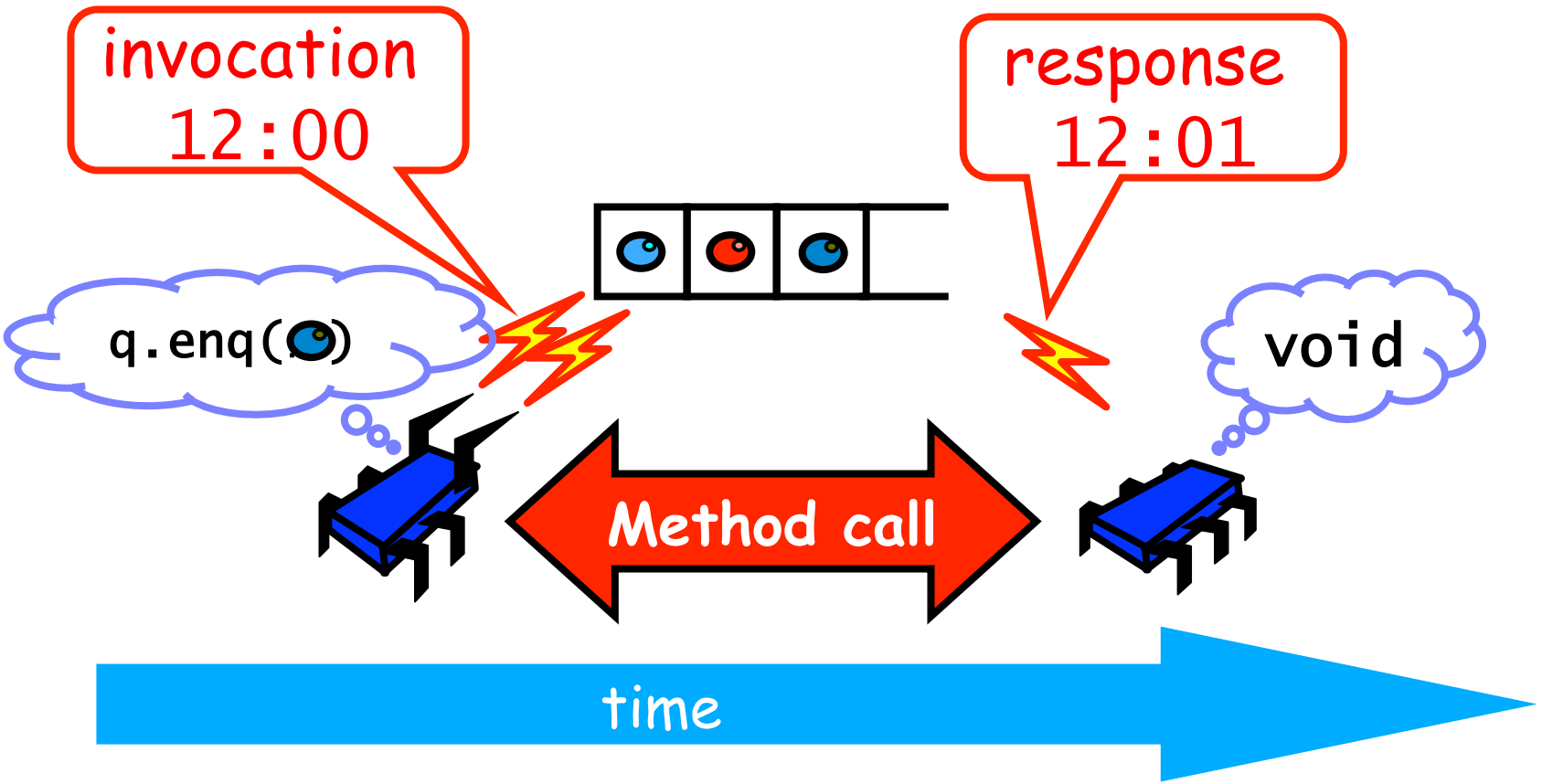
Methods Take Time



Methods Take Time



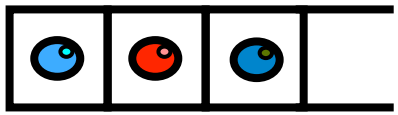
Methods Take Time



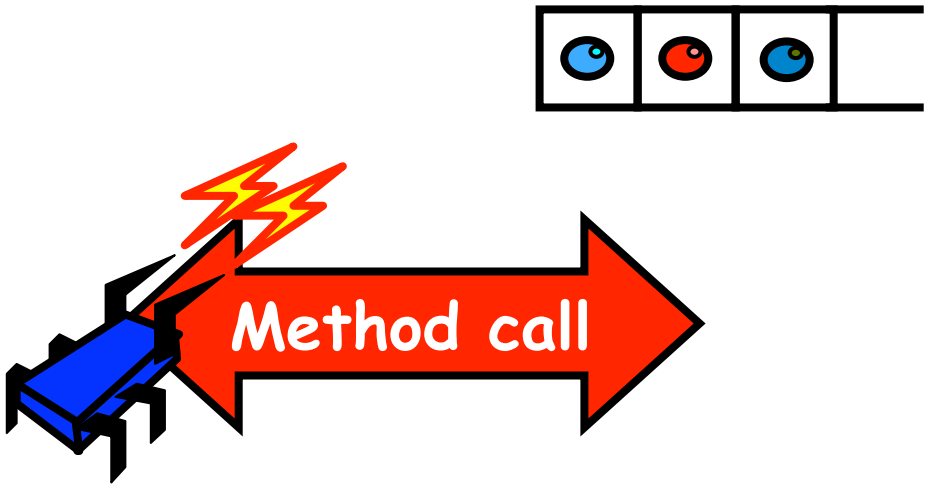
Sequenziale vs Concorrente

- Sequenziale
  - Methods take time?
  - Quando mai!!!
- Concorrente
  - La chiamata di un metodo e' un intervallo.

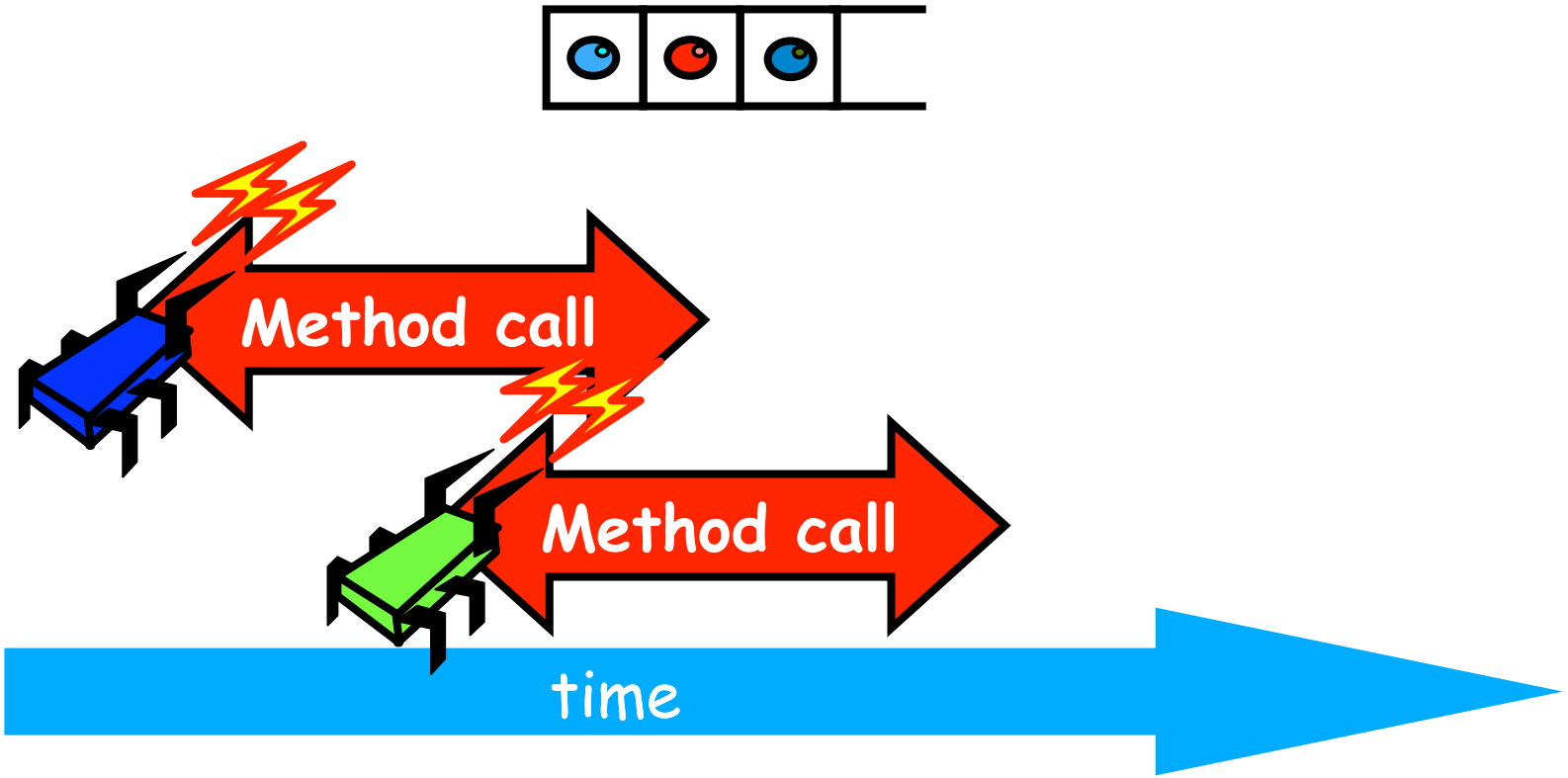
Overlapping Time



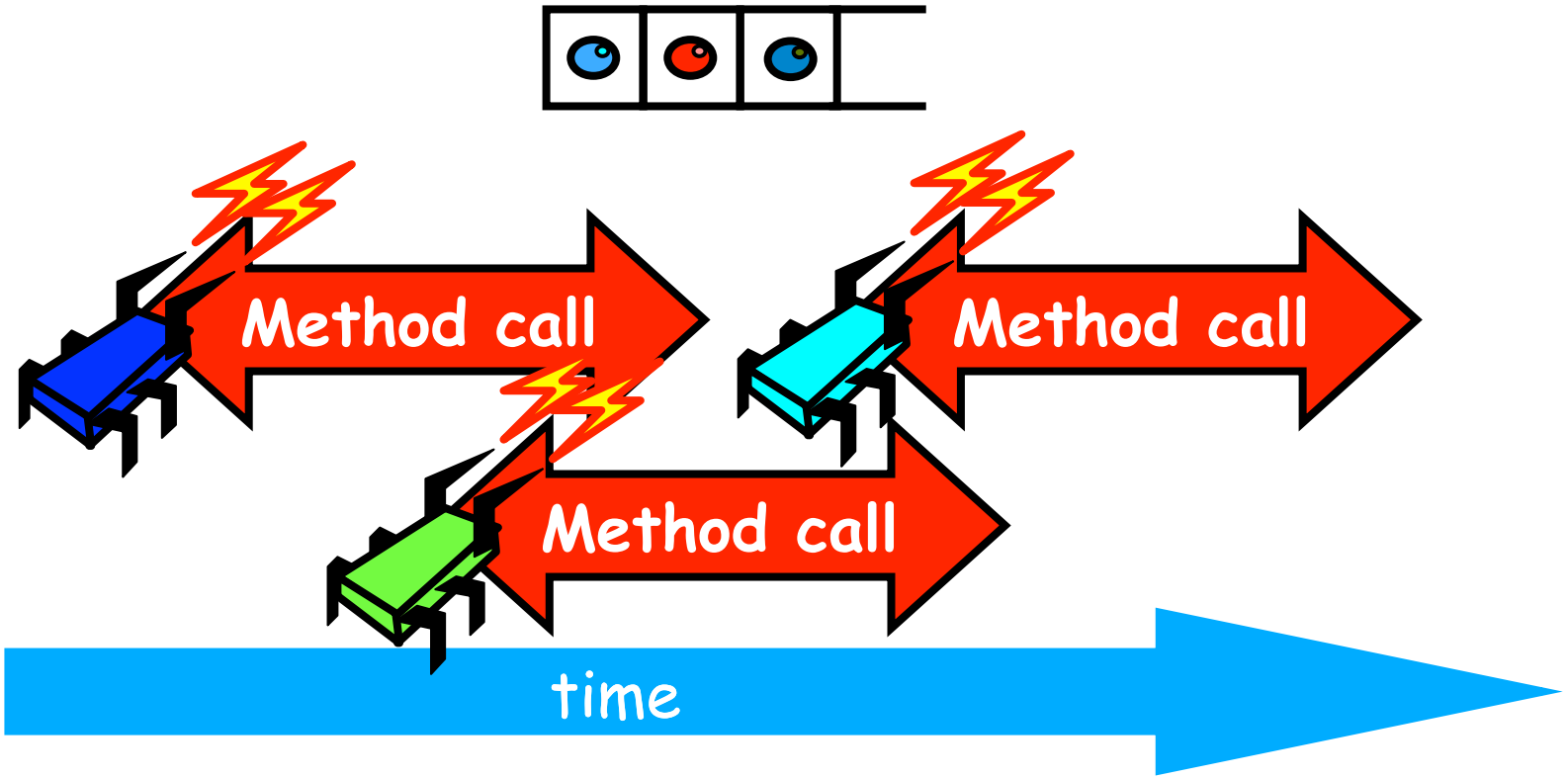
Overlapping Time



Overlapping Time



Overlapping Time



Sequenziale vs Concorrente

- Sequenziale:
  - Stato degli oggetti e' significativo solamente ***tra*** le invocazioni dei metodi
- Concorrente
  - Dato che le chiamate si sovrappongono la stato di un oggetto potrebbe ***non essere mai*** tra le invocazioni dei metodo

Sequenziale vs Concorrente

- Sequenziale:
  - Ogni metodo e' descritto in isolamento
- Concorrente
  - Si devono comprendere ***tutte*** le possiibili interazioni con chiamate concorrenti
    - Cosa succede se due invocazioni di enq si sovrappongono?

Sequenziale vs Concorrente

- Sequenziale:
  - Refinement
- Concorrente:
  - Ogni metodo puo'potenzialemnte interagire con tutti gli altri

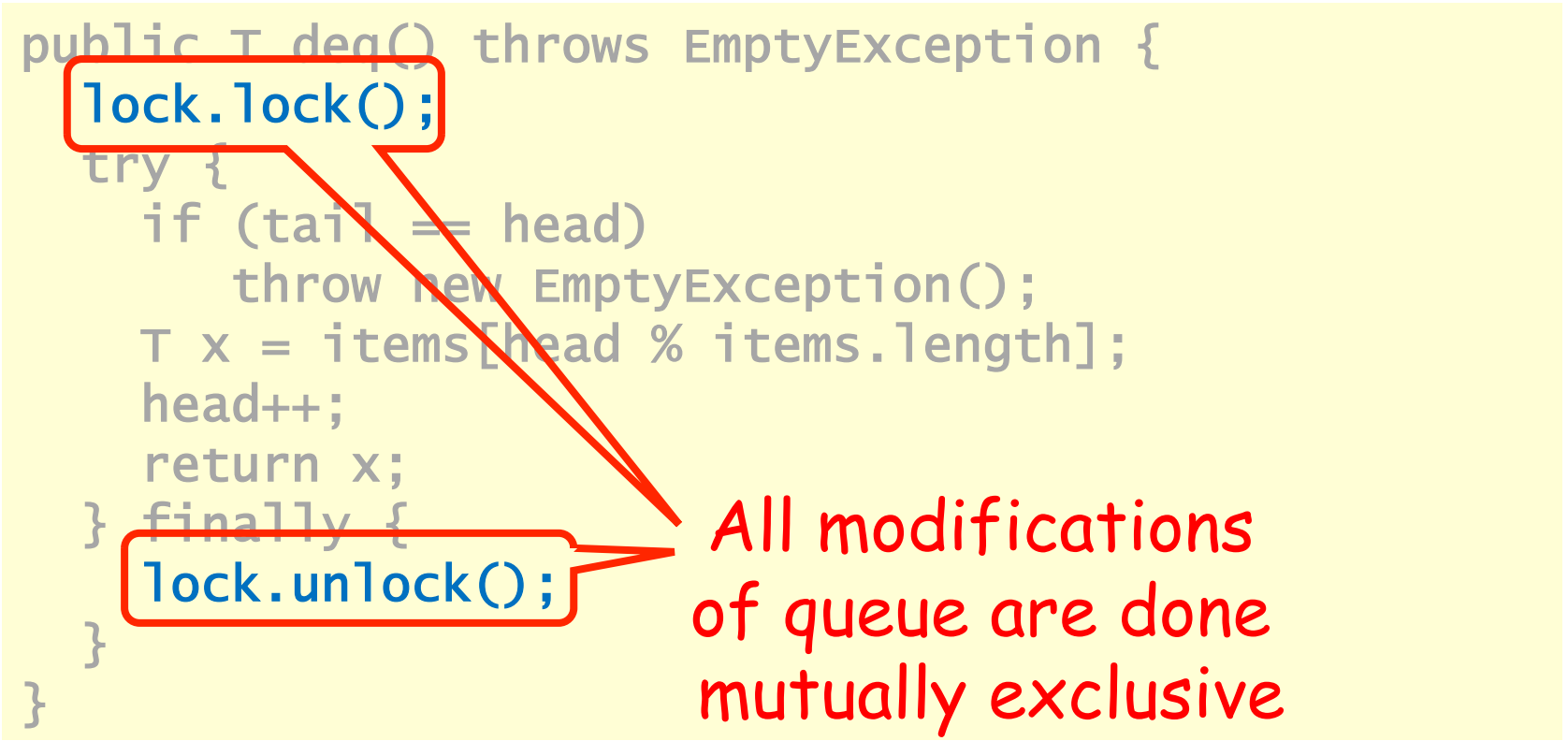
Sequenziale vs Concorrente

- Sequenziale:
  - Refinement
- Concorrente:
  - Ogni metodo puo'potenzialemnte interagire con tutti gli altri

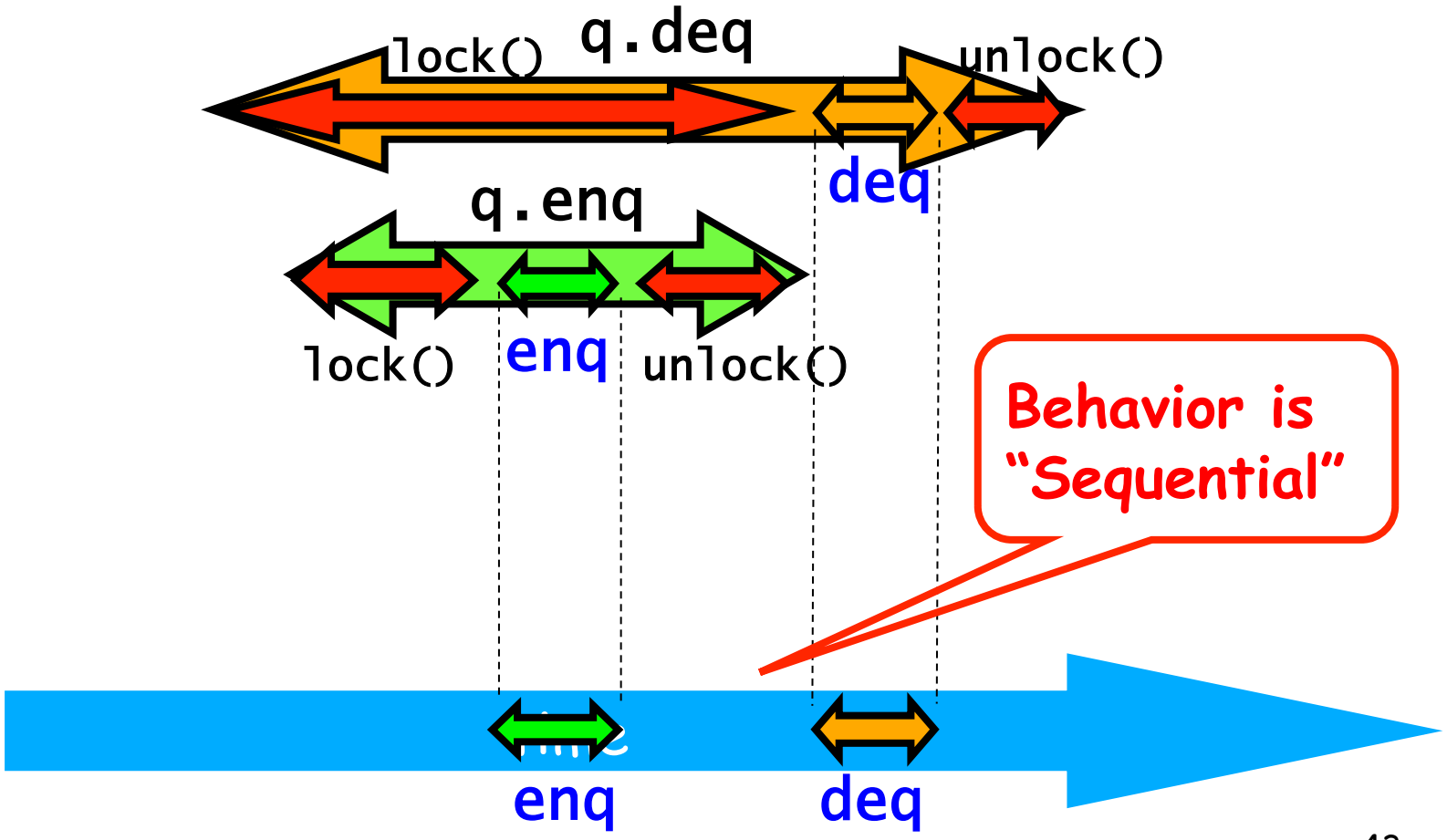
La domanda

- Quale e' la nozione di correttezza nel caso concorrente?
  - FIFO implica ordine temporale strettoo
  - Concorrenza implica un ordine temporale non determinato

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


public T deq() throws EmptyException {
lock.lock();
try {
if (tail == head)
throw new EmptyException();
T x = items[head % items.length];
head++;
return x;
} finally {
lock.unlock();
}
}
All modifications
of queue are done
mutually exclusive

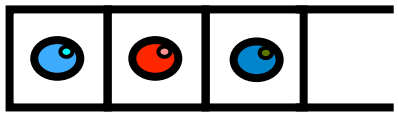
Catturiamo la concorrenza mediante l'ordine Degli eventi



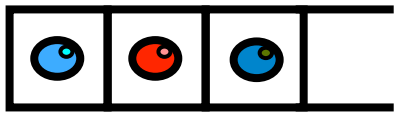
Linearizability

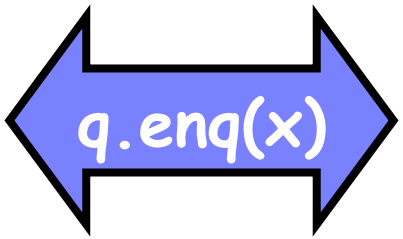
- Un oggetto e' corretto se la sua proiezione sequenziale e' corretta
  - **Linearizable**

Un oggetto e' linearizable: se tuutte le sue possibili esecuzioni sono linearizable

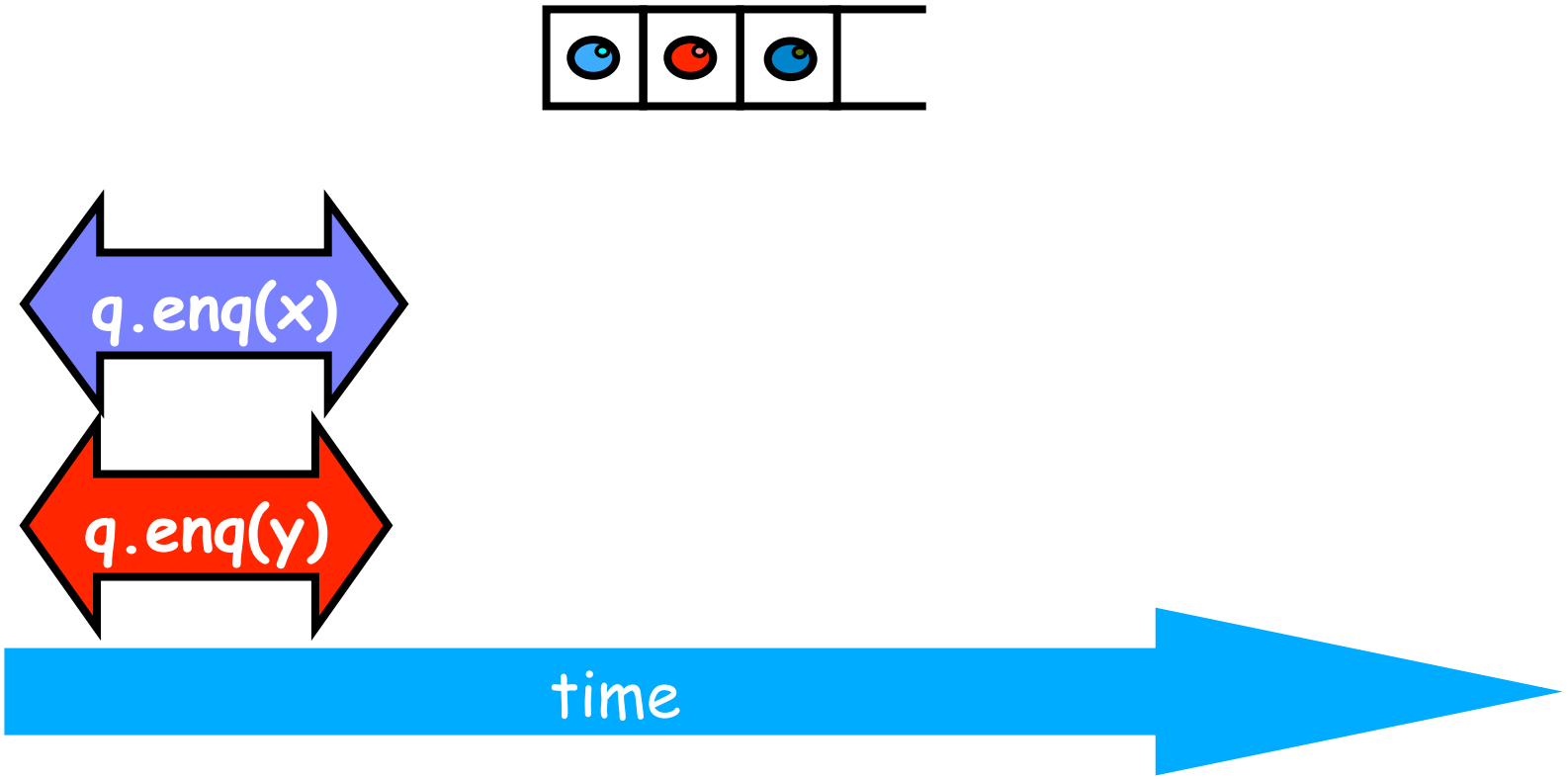


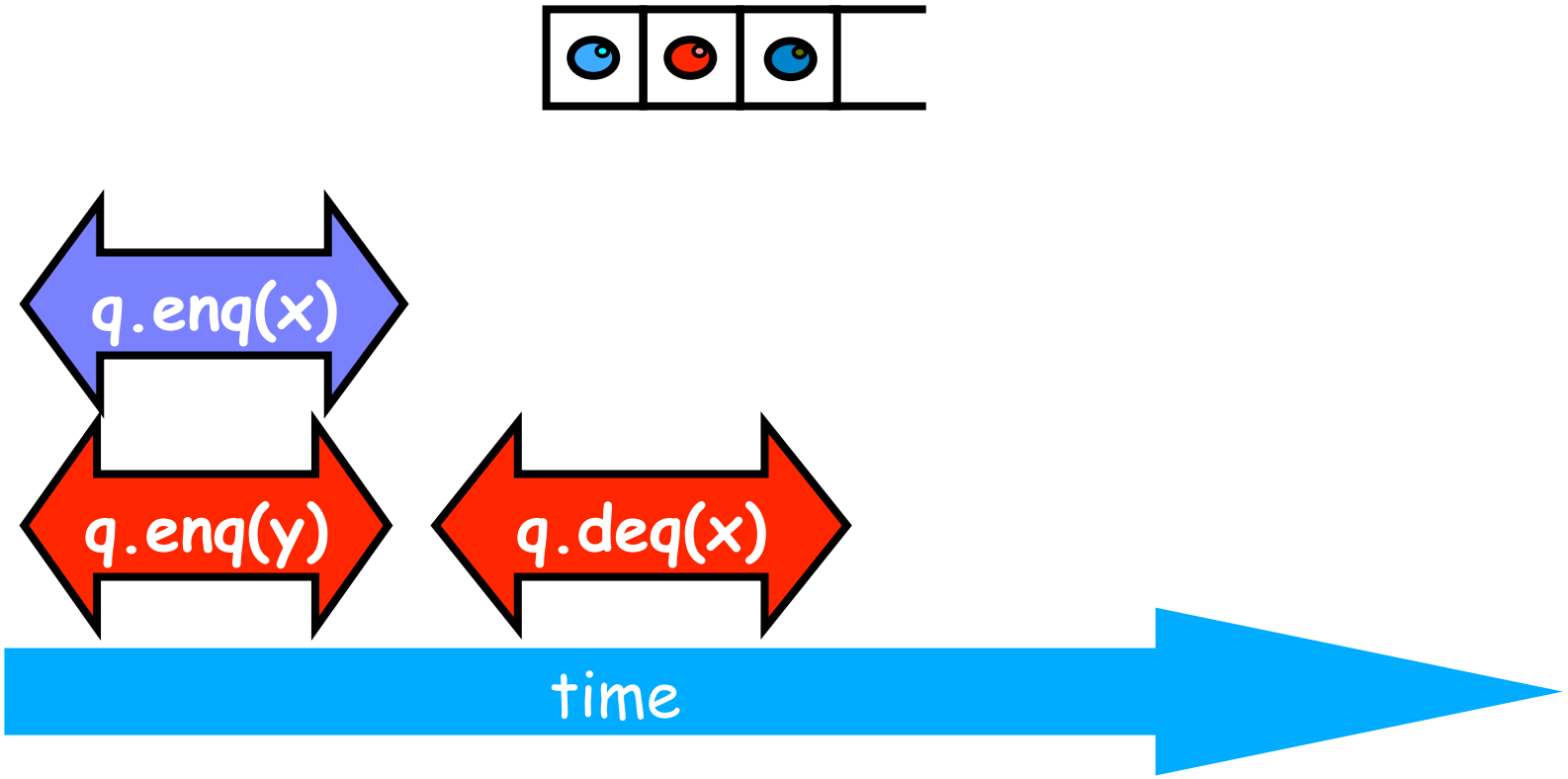

time

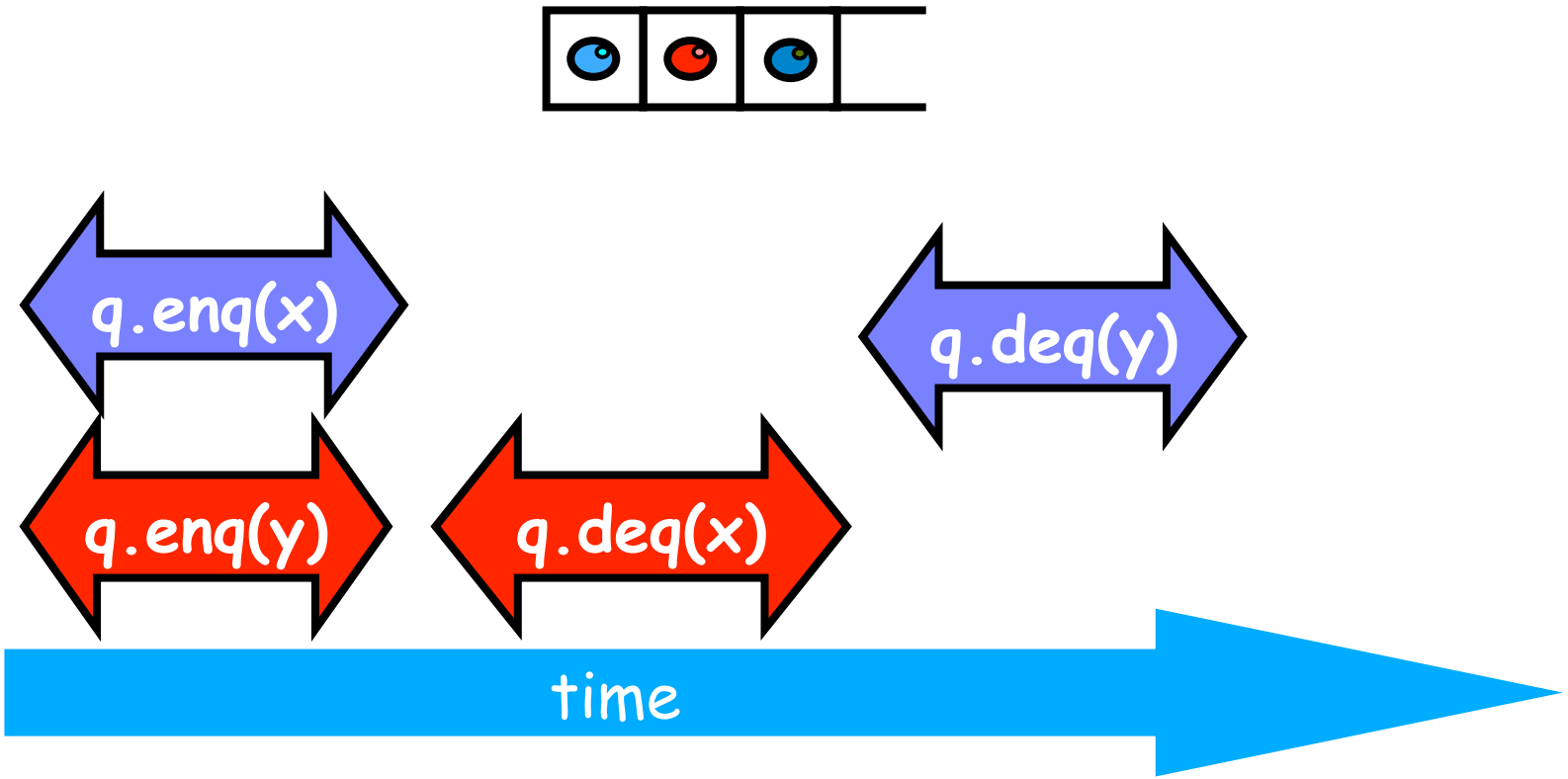


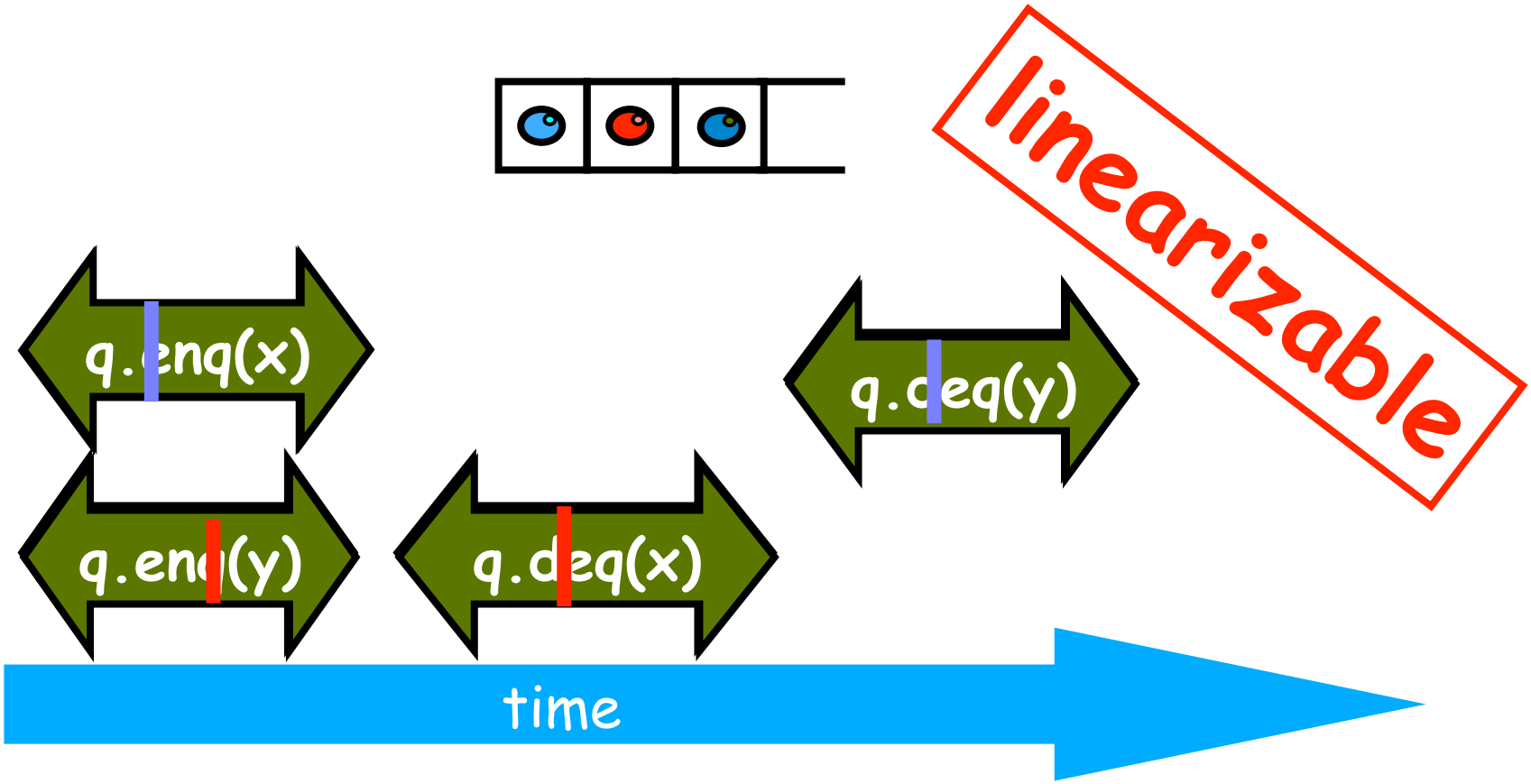

q.enq(x)

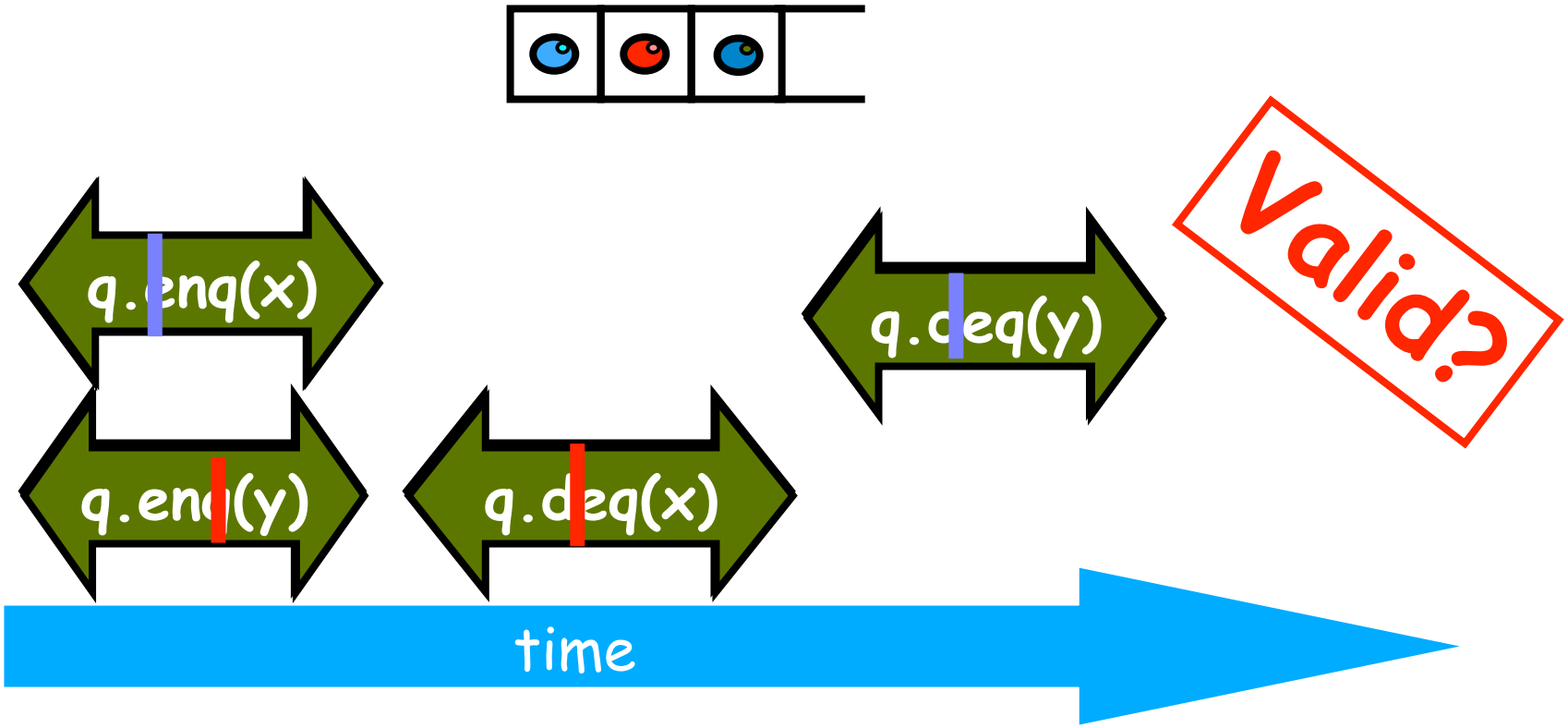

time

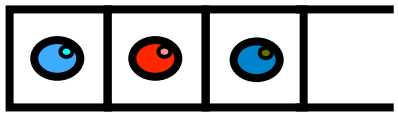

q.enq(x)
q.enq(y)
time


q.enq(x)
q.enq(y)
q.deq(x)
time

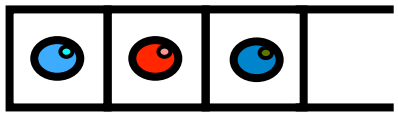

q.enq(x)
q.deq(y)
q.enq(y)
q.deq(x)
time

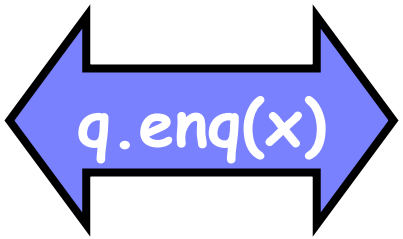

linearizable
q.enq(x)
q.deq(y)
q.enq(y)
q.deq(x)
time


q.enq(x)
q.deq(y)
Valid?
q.enq(y)
q.deq(x)
time

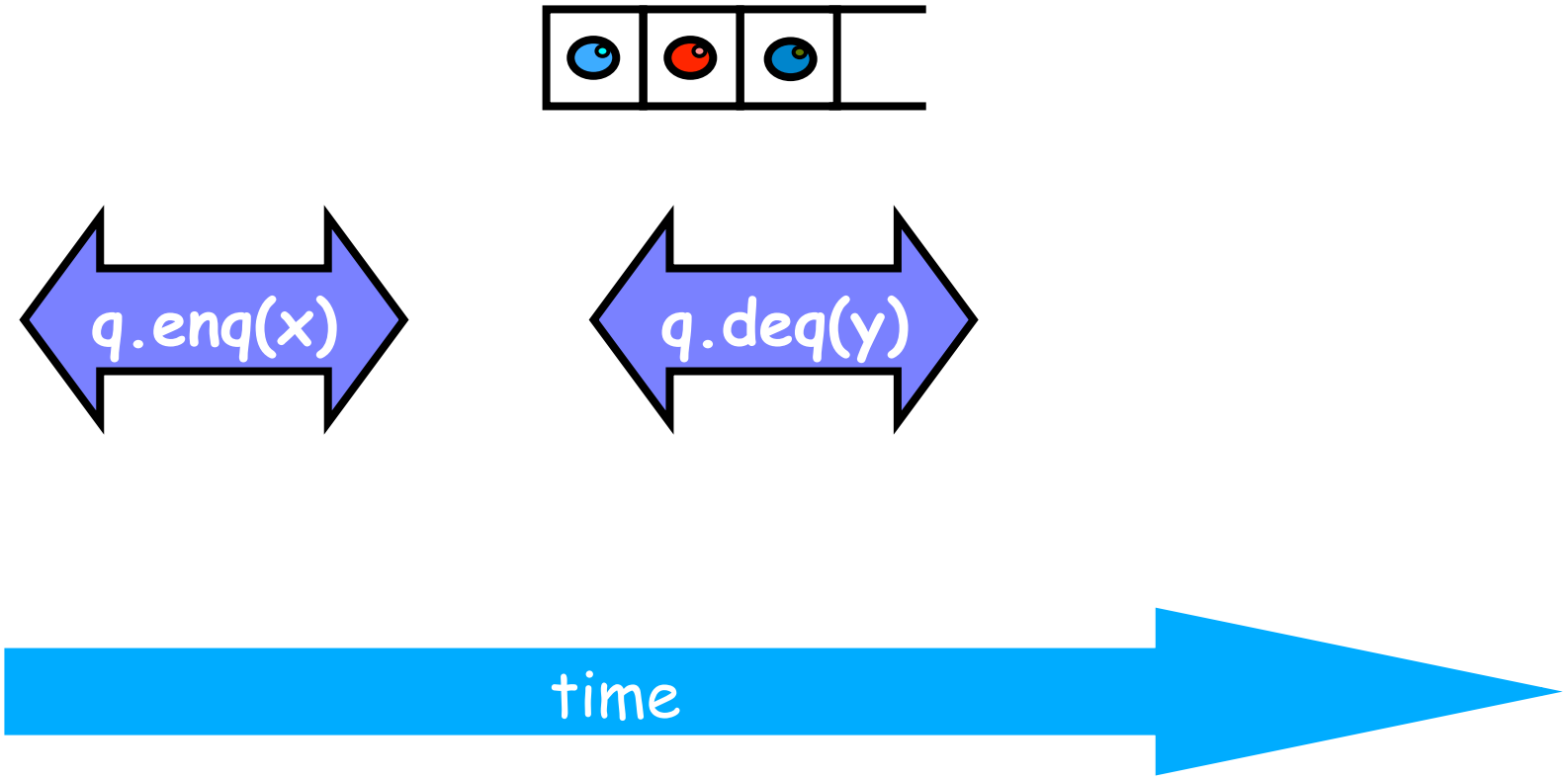


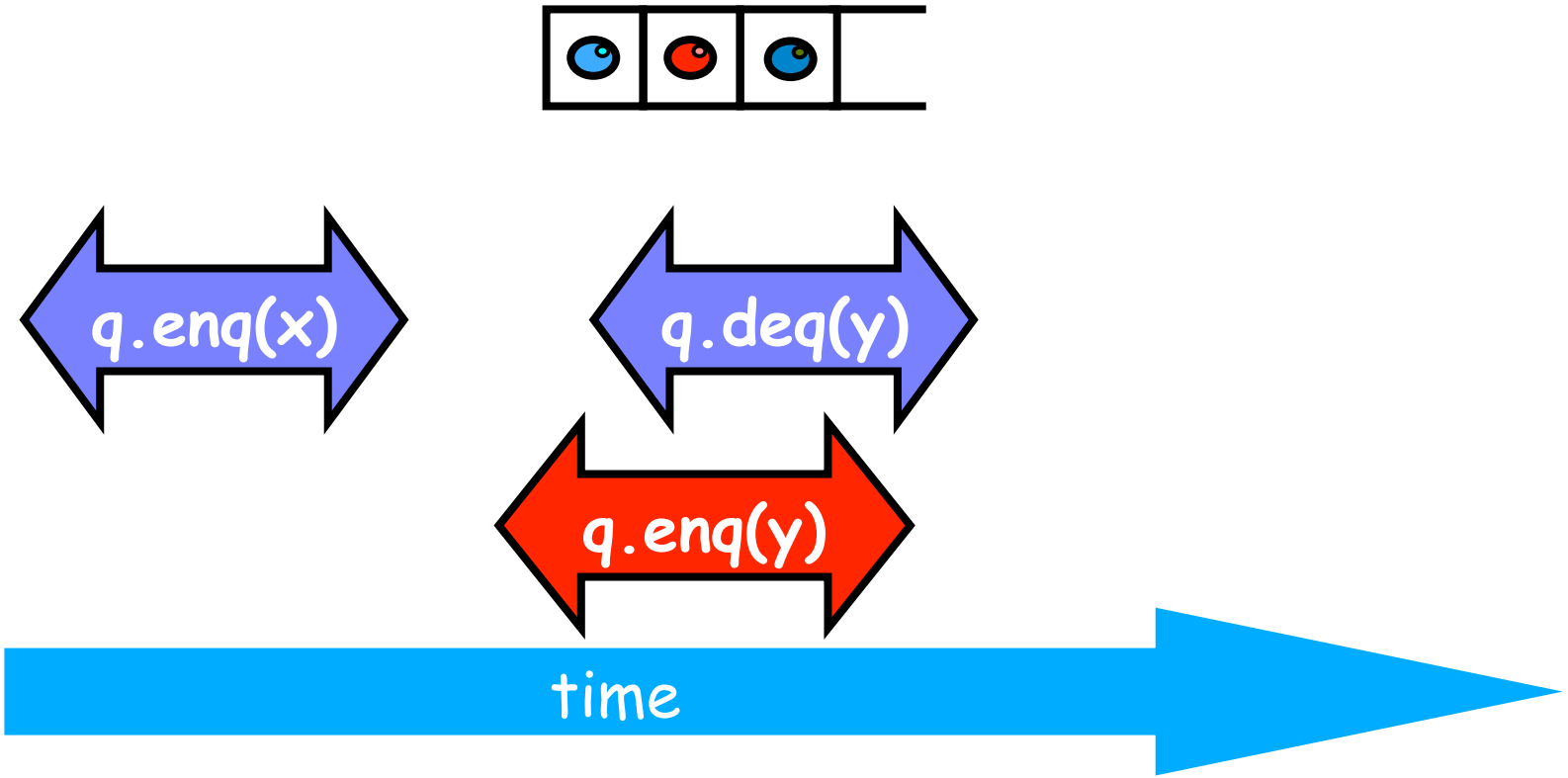

time

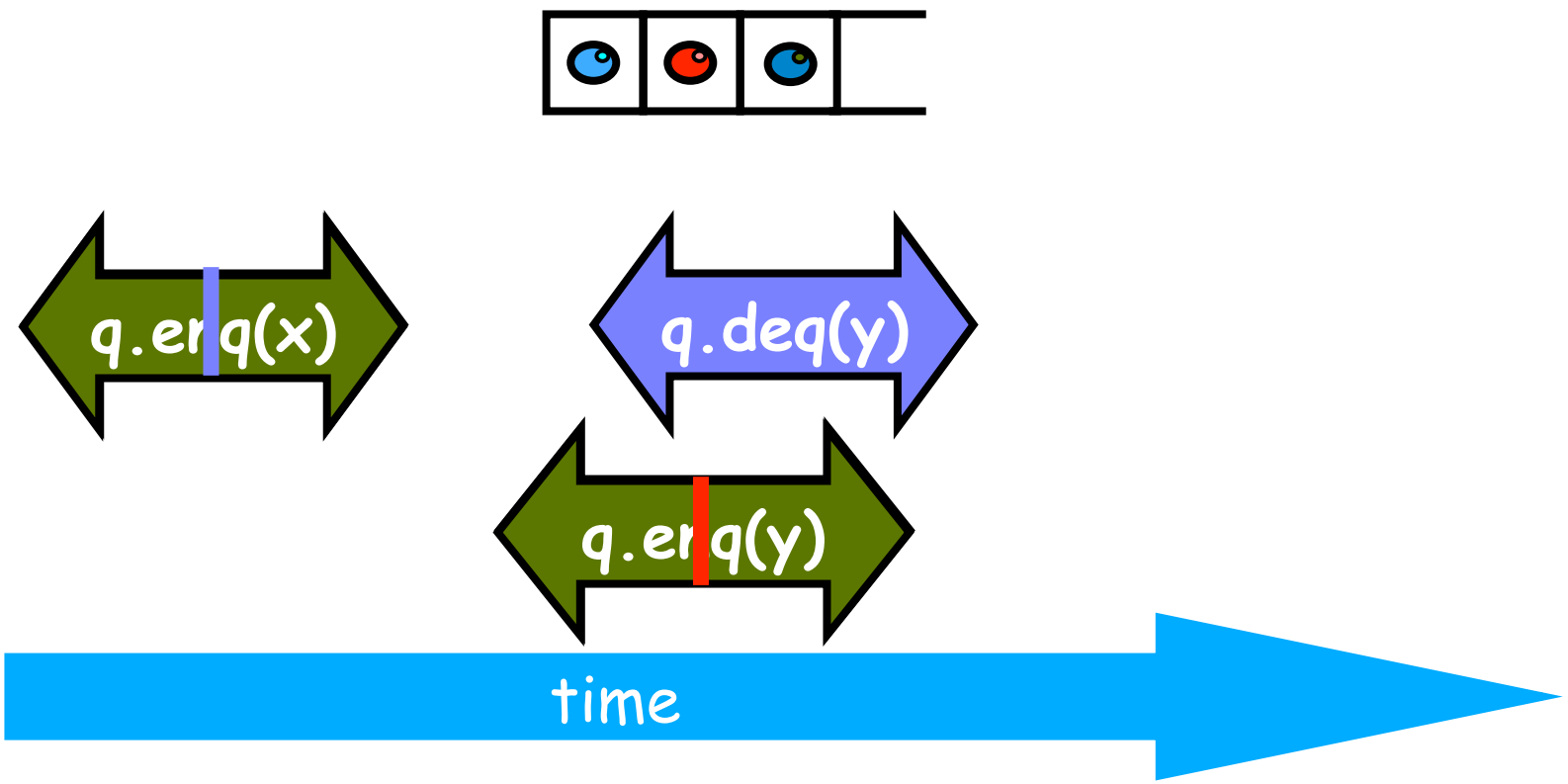


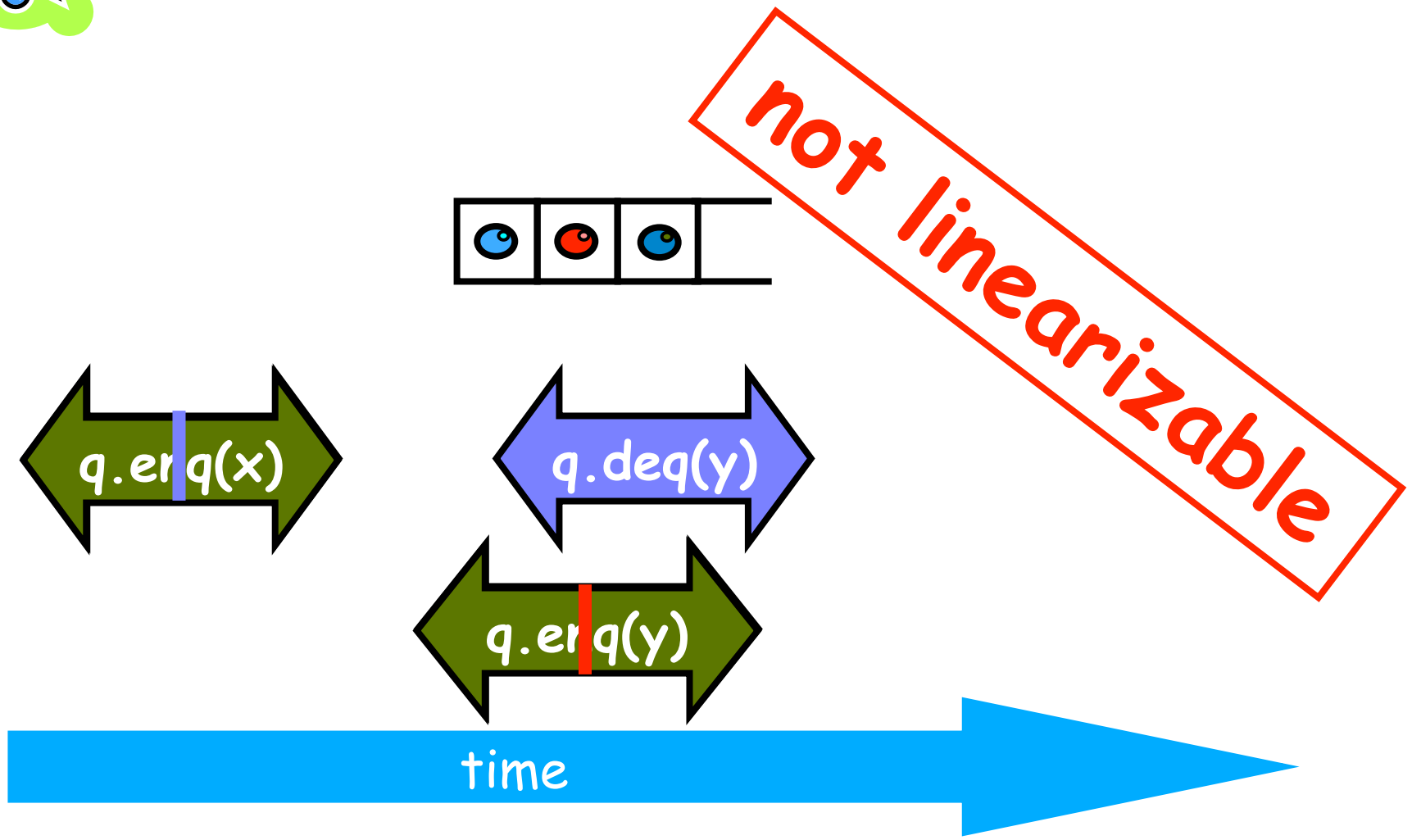

q.enq(x)

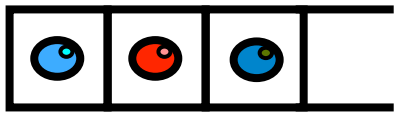

time


q.enq(x)
q.deq(y)
time

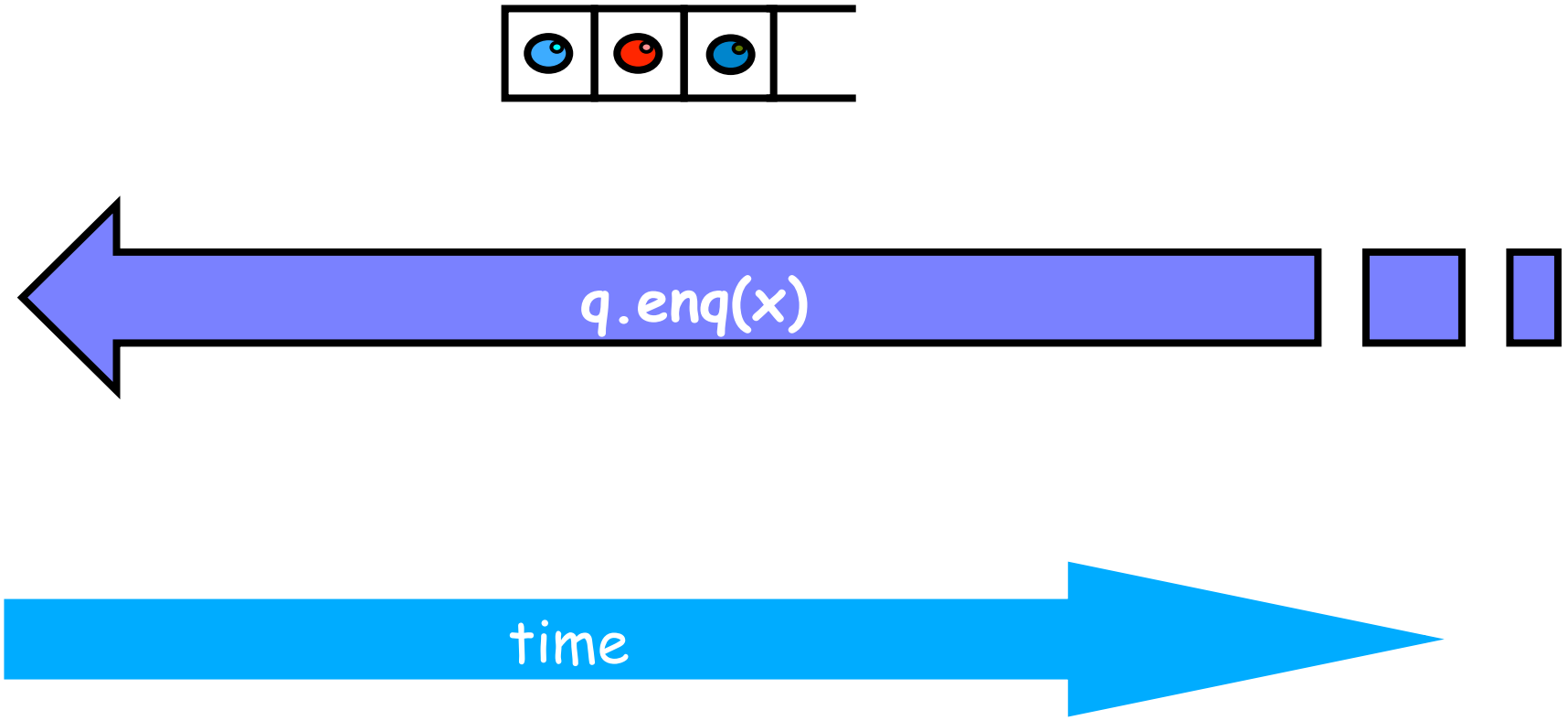

q.enq(x)
q.deq(y)
q.enq(y)
time

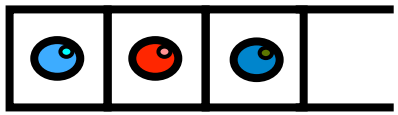

q.enq(x)
q.deq(y)
q.enq(y)
time

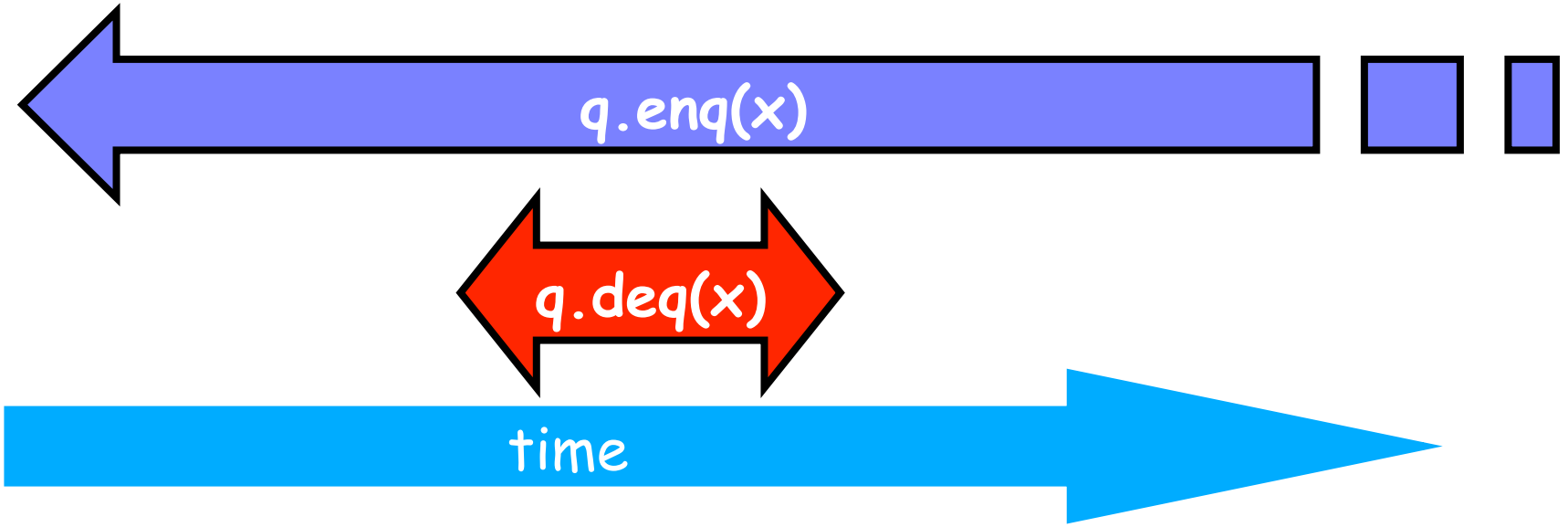

not linearizable
q.enq(x)
q.deq(y)
q.enq(y)
time

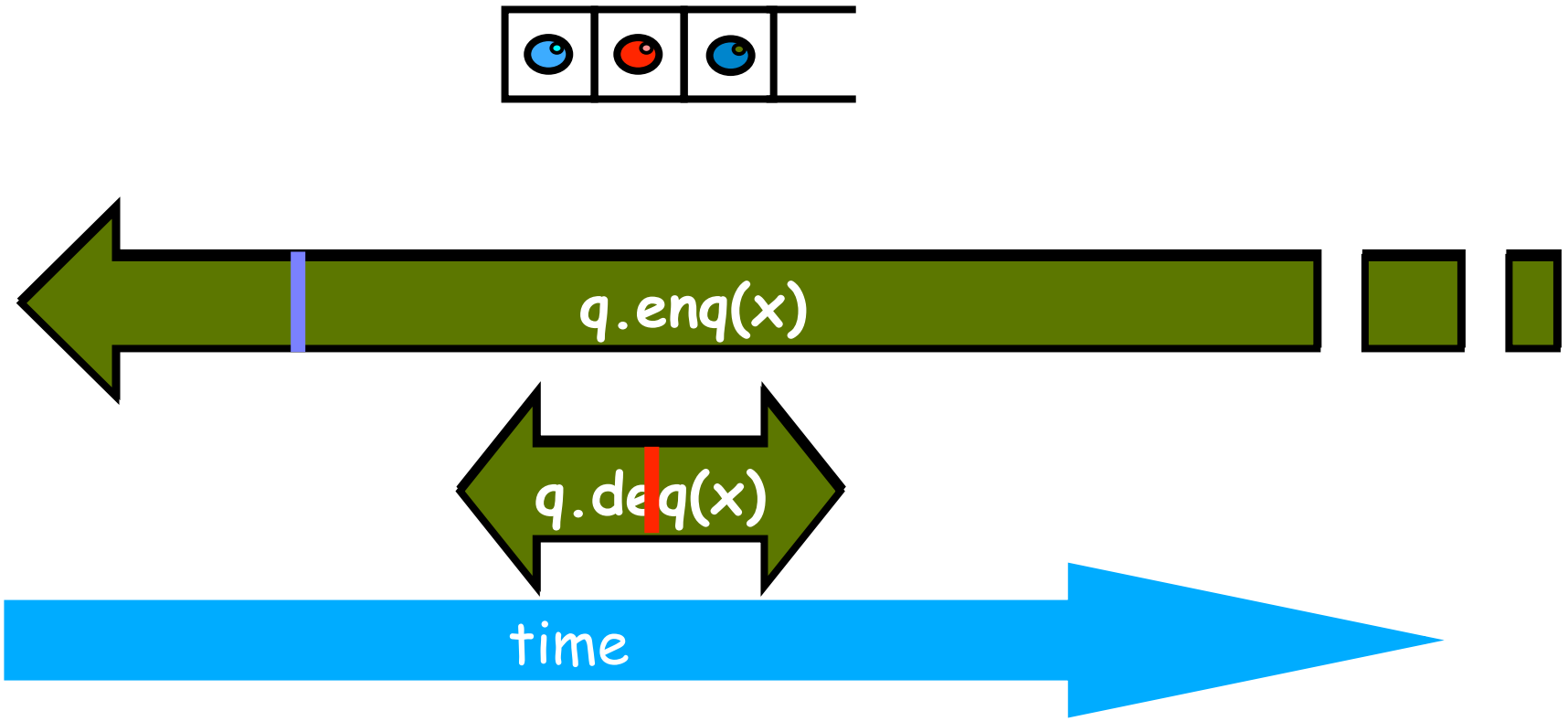


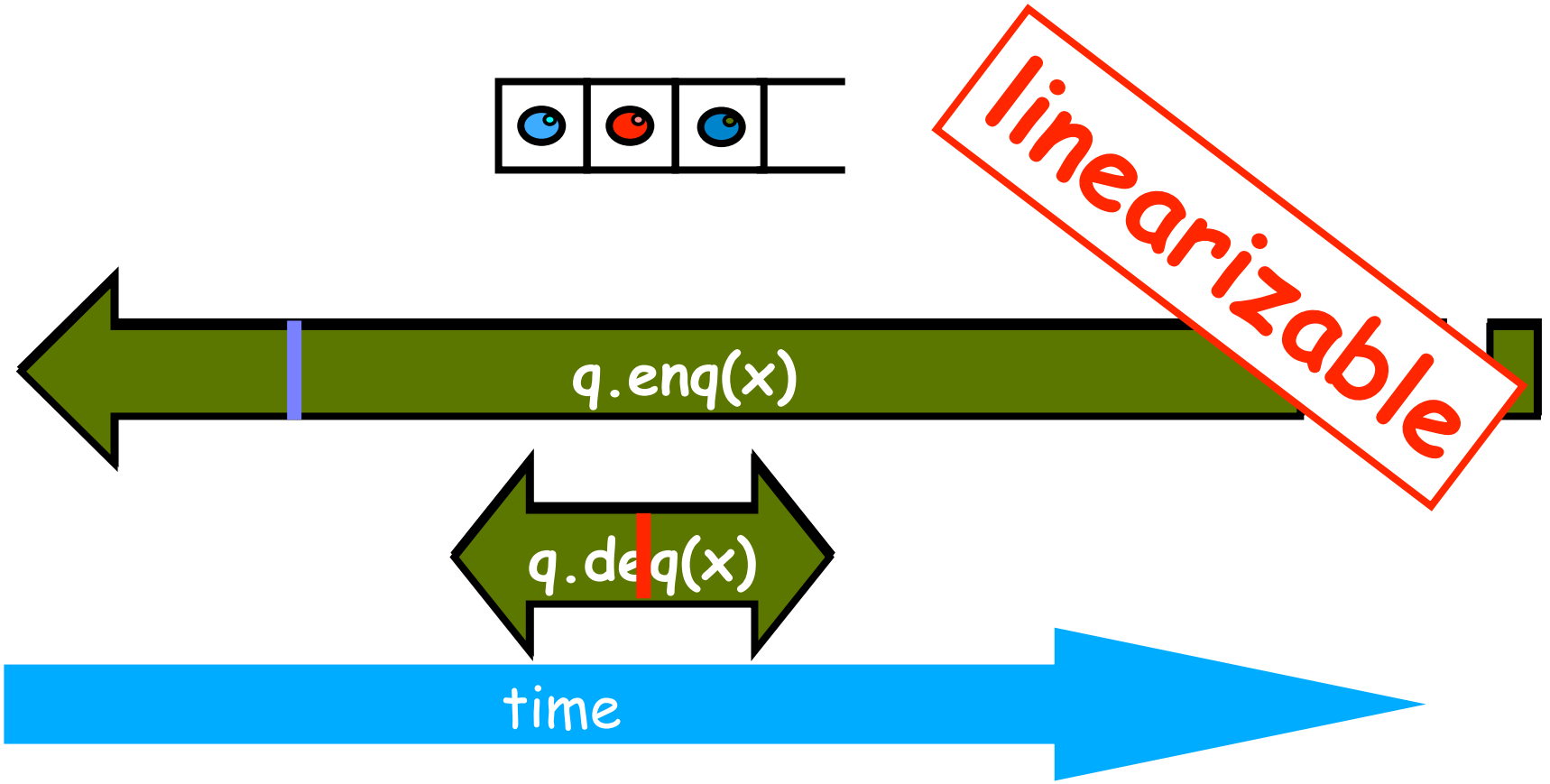

time

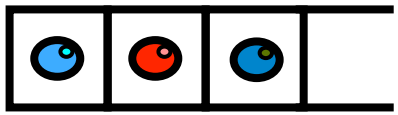

q.enq(x)
time

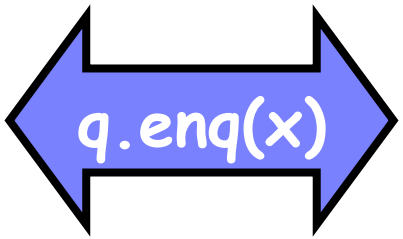



q.enq(x)
q.deq(x)
time

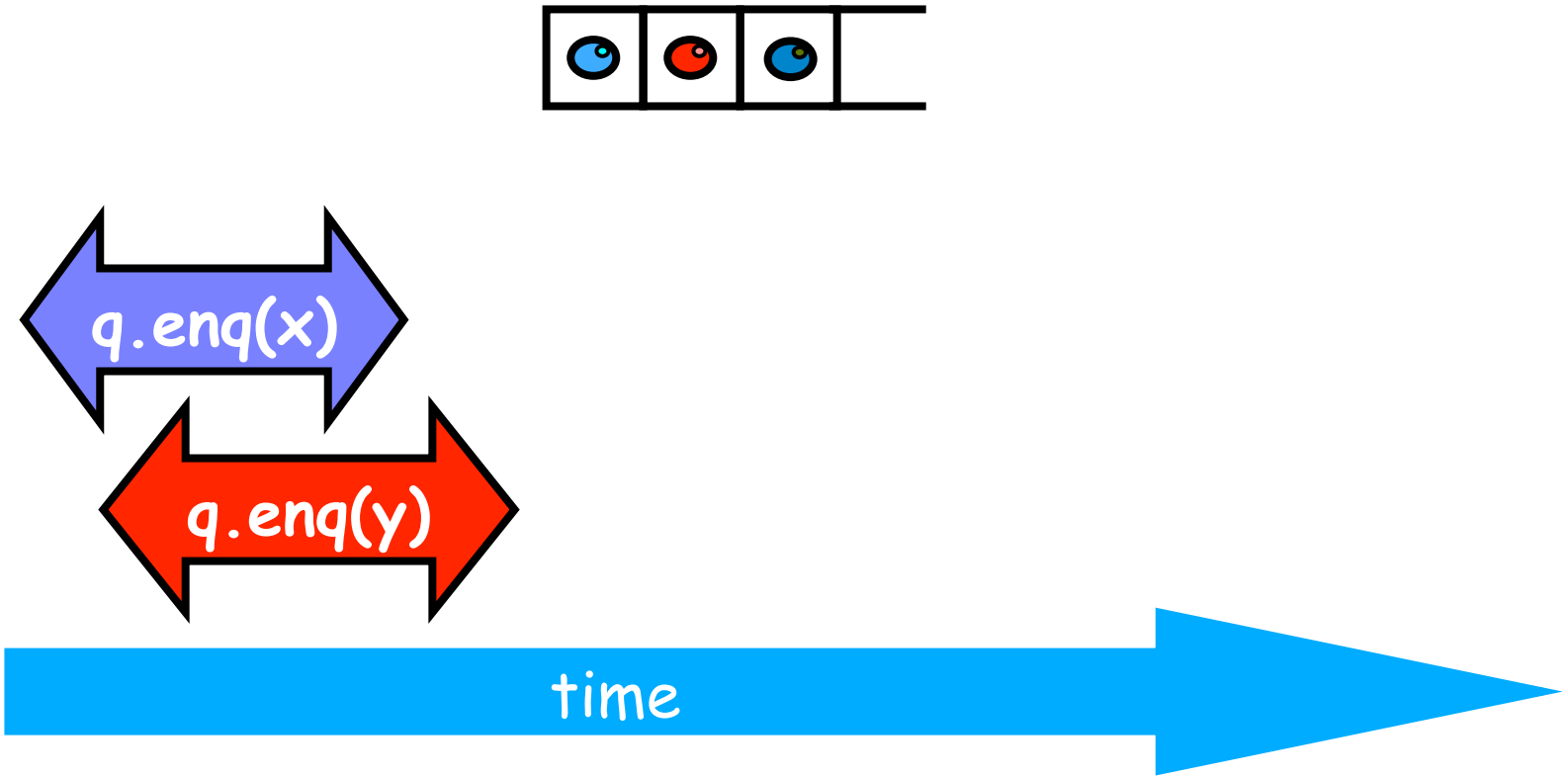

q.enq(x)
q.deq(x)
time

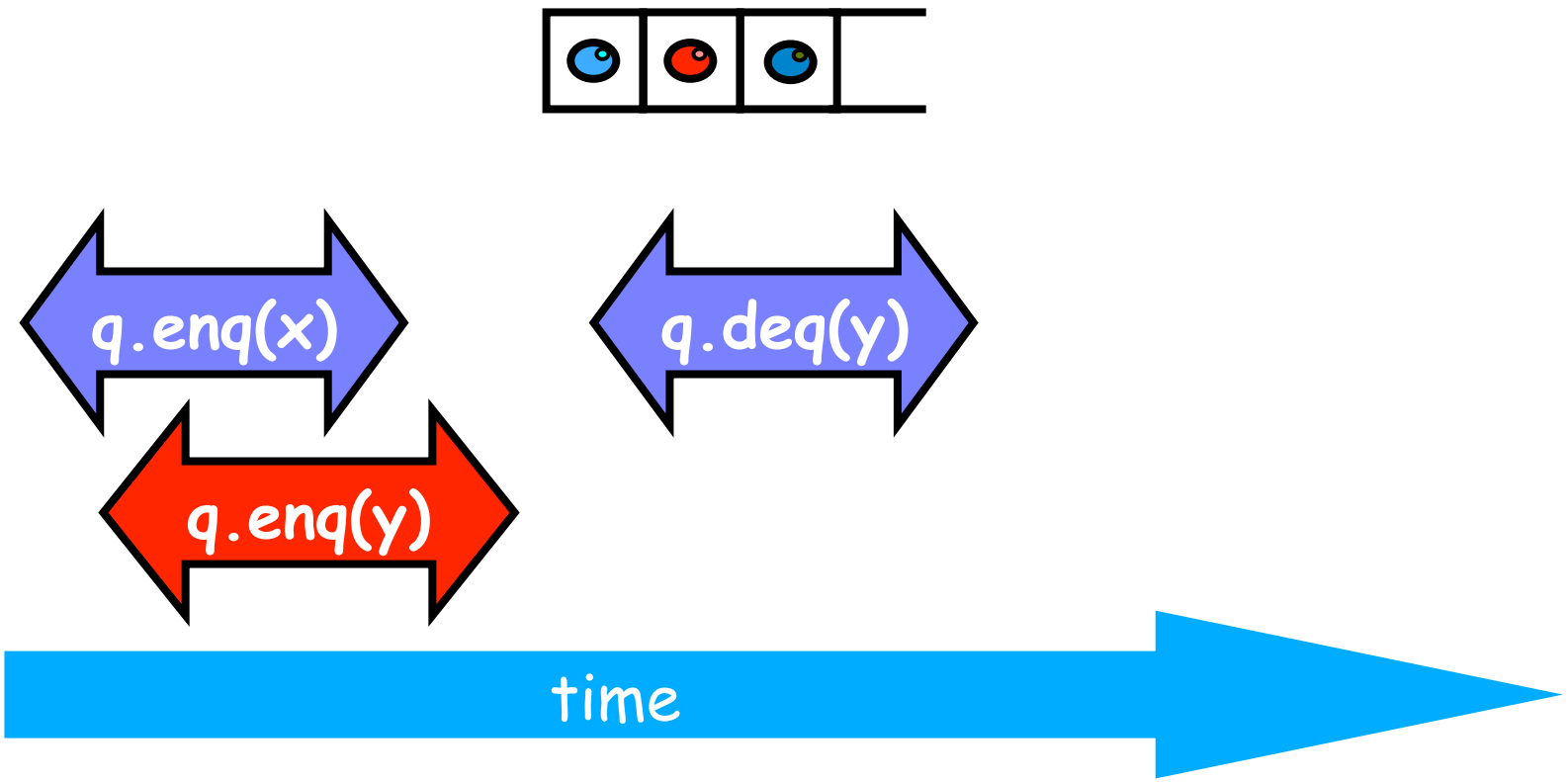

linearizable
q.enq(x)
q.deq(x)
time

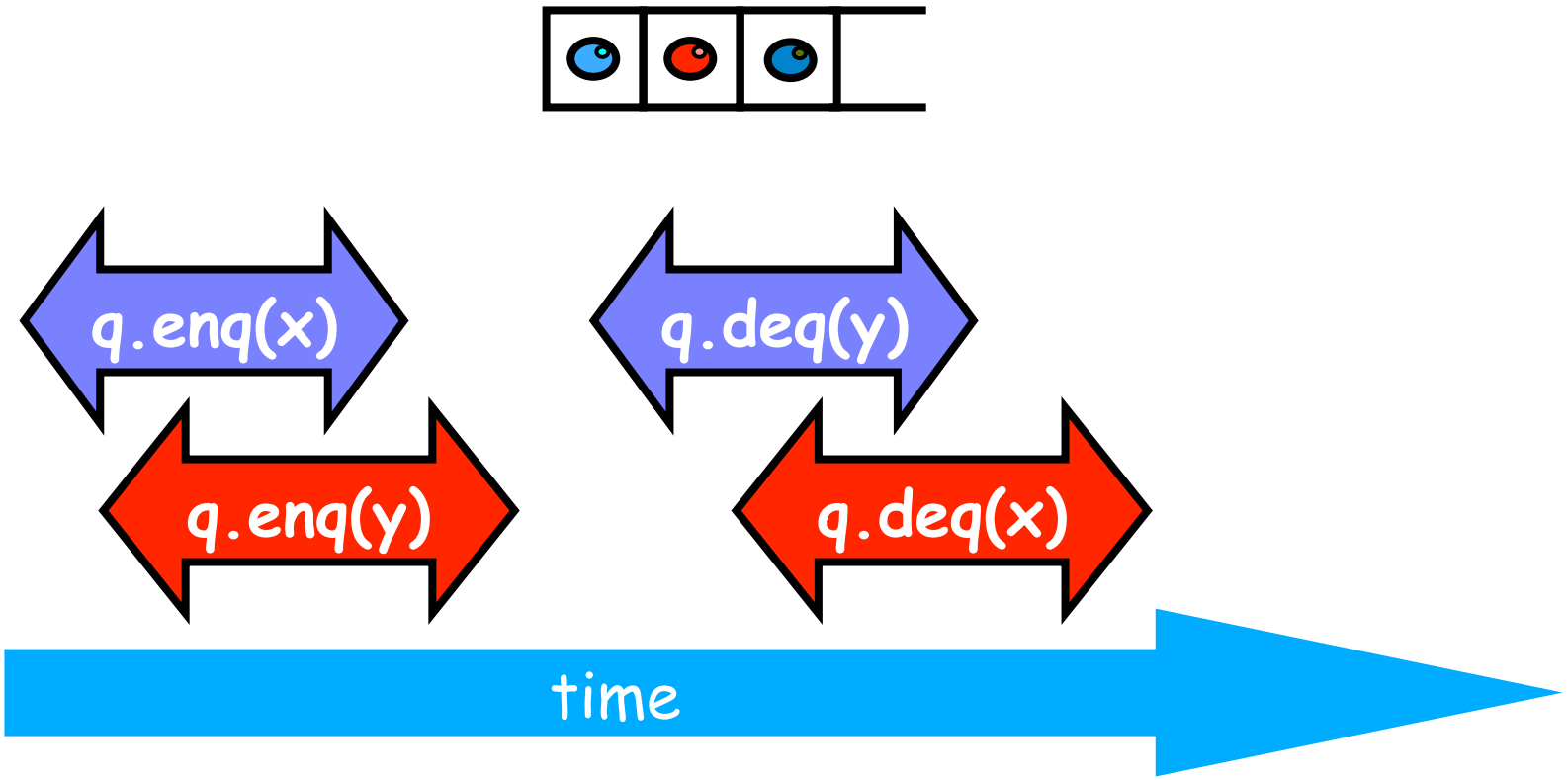


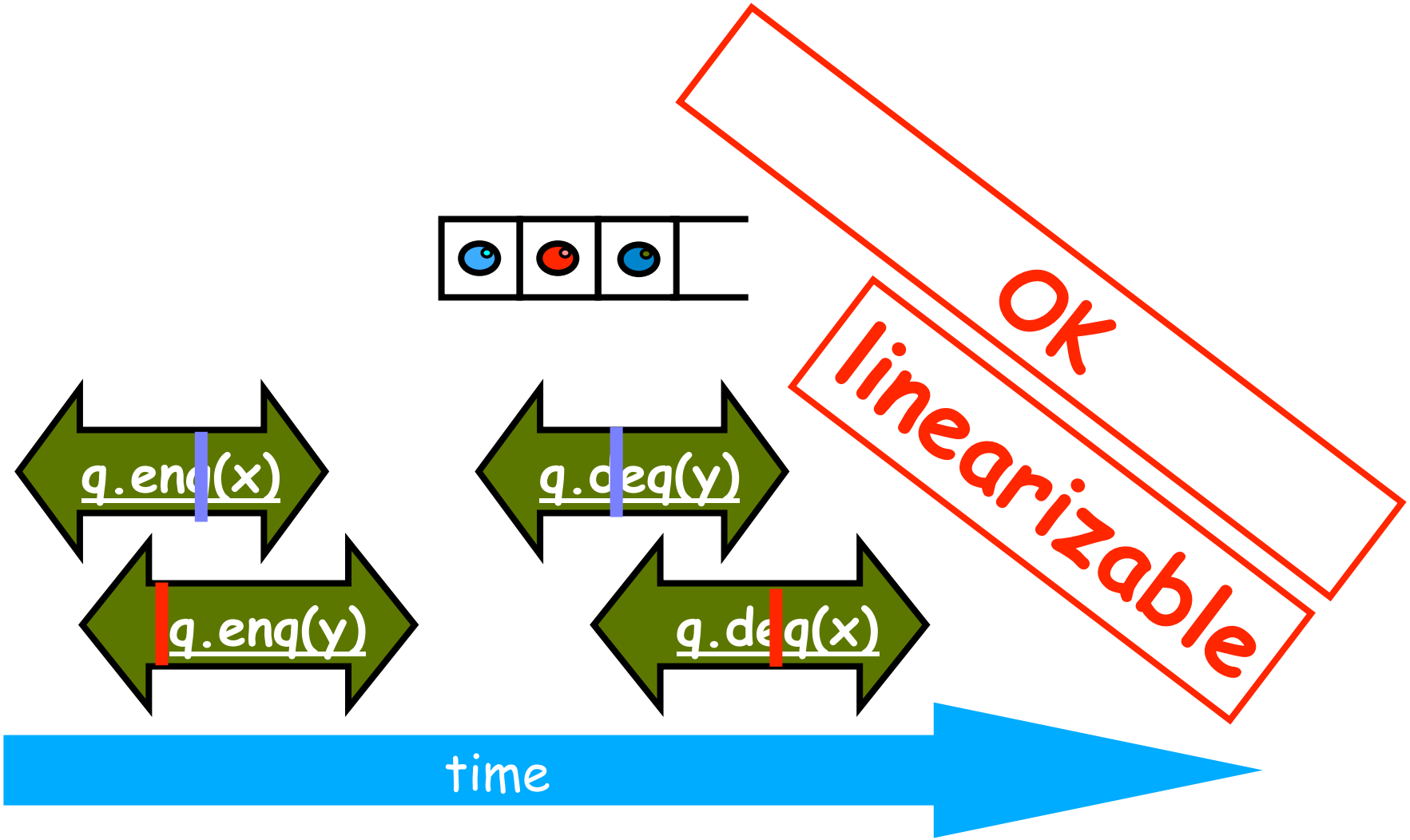

q.enq(x)


time


q.enq(x)
q.enq(y)
time


q.enq(x)
q.deq(y)
q.enq(y)
time


q.enq(x)
q.deq(y)
q.enq(y)
q.deq(x)
time


Ok
linearizable
q.enq(x)
q.deq(y)
q.enq(y)
q.deq(x)
time