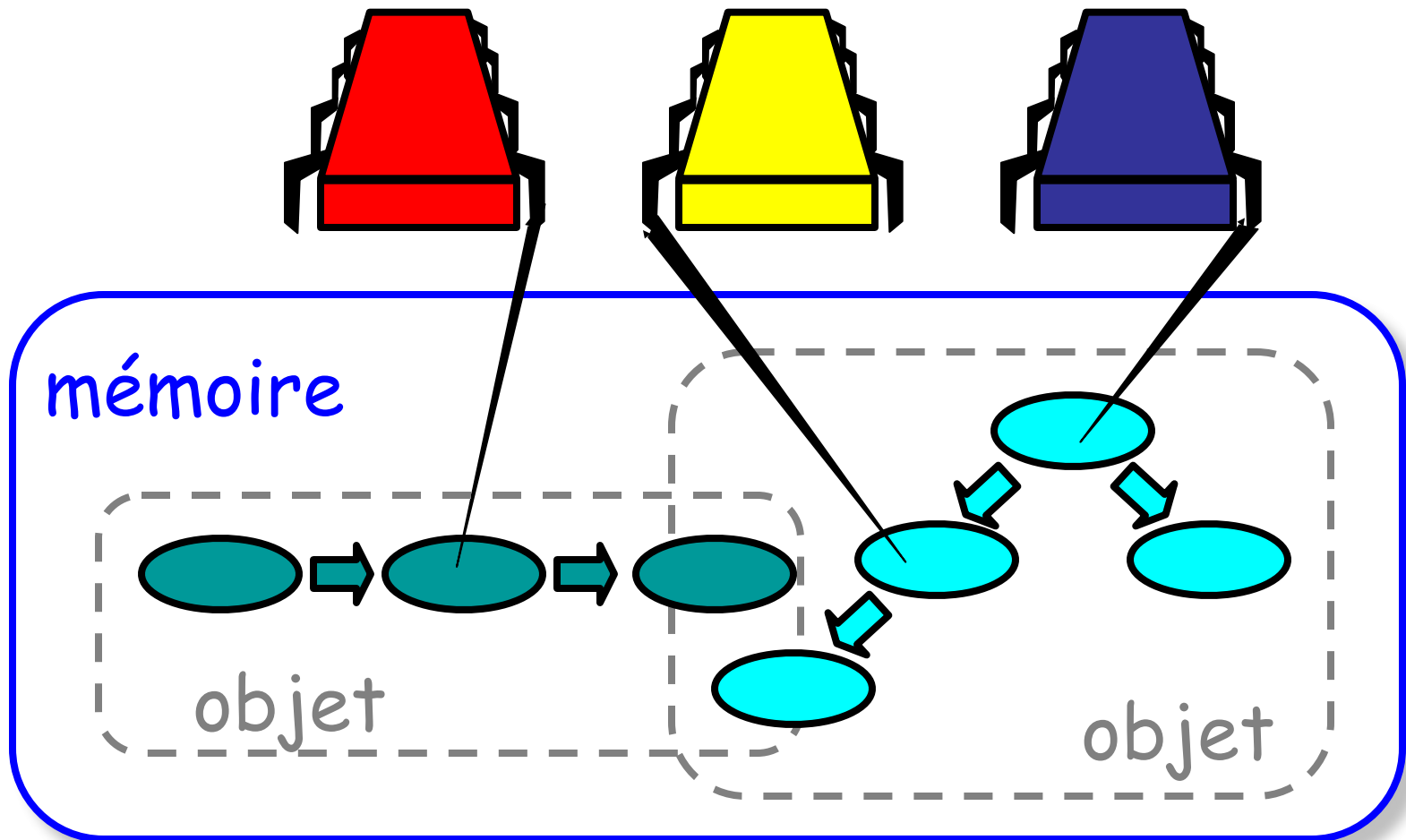


Linéarisabilité

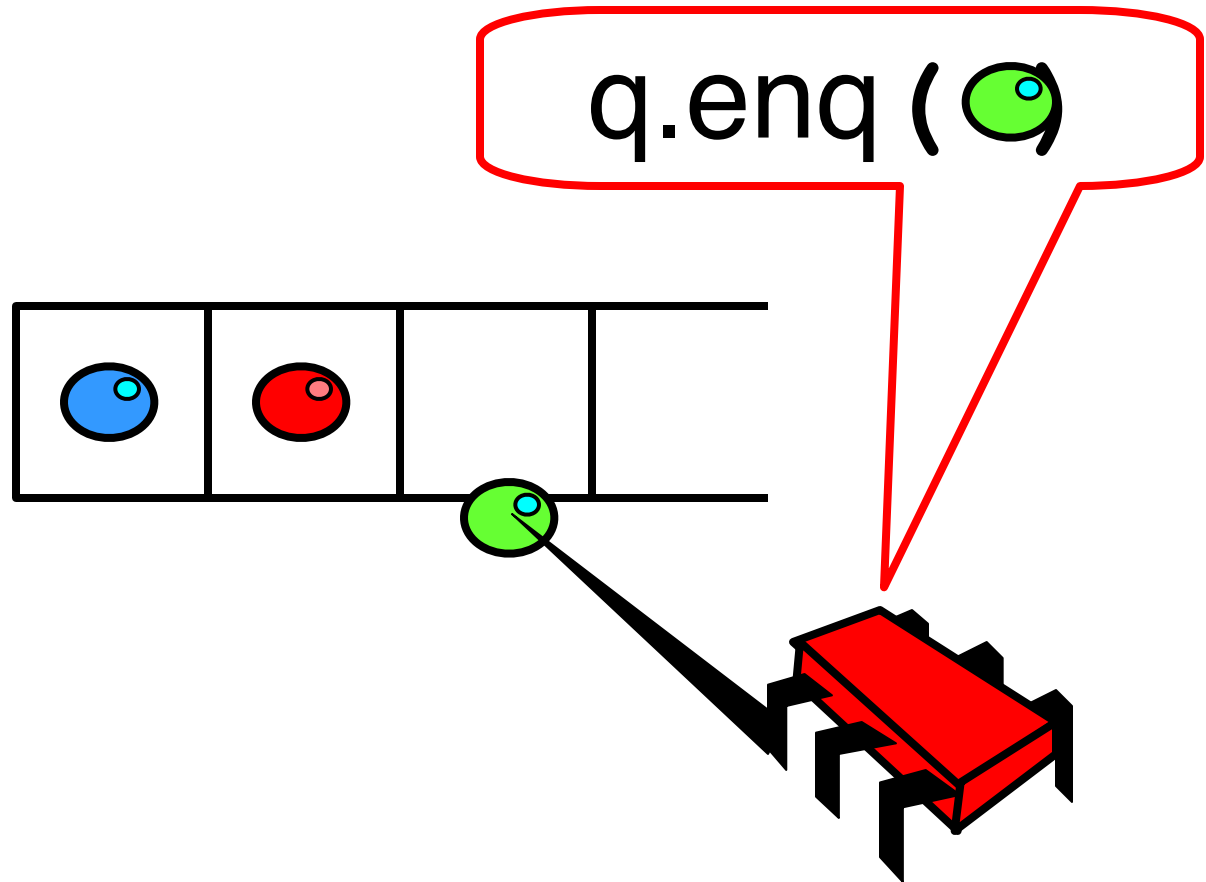
Concurrence



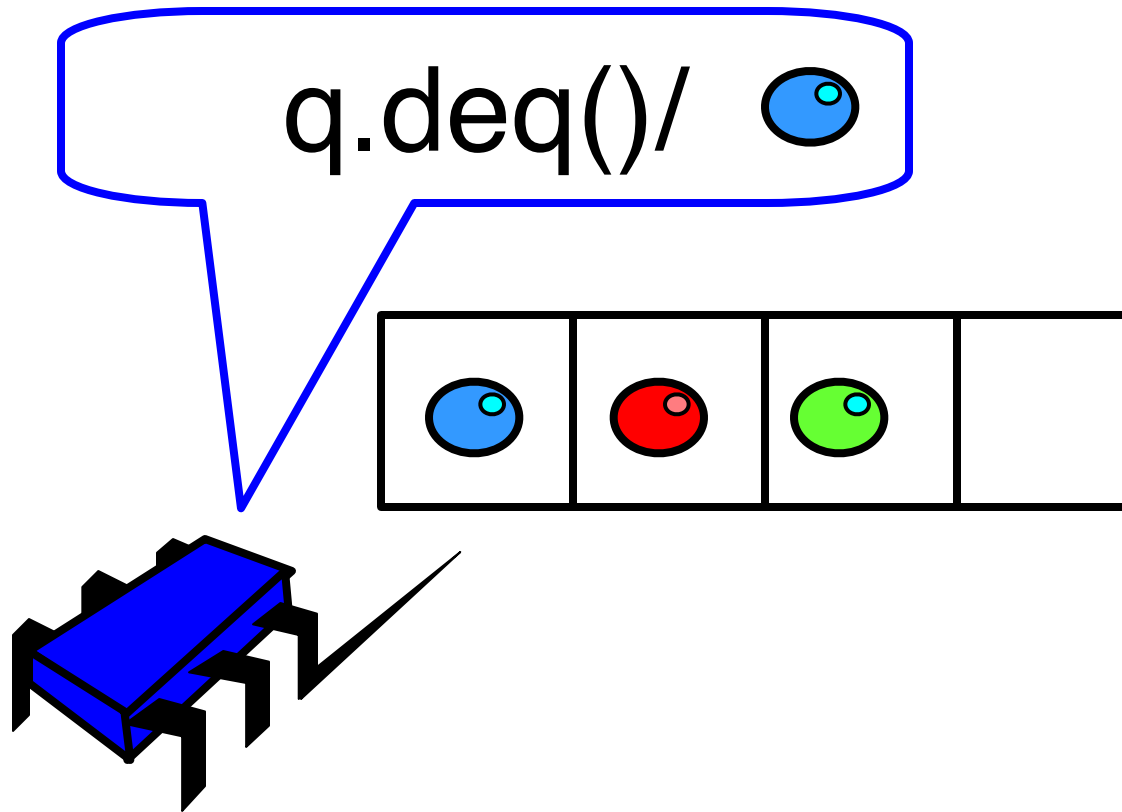
Objectifs

- Qu'est ce qu'un objet concurrent?
 - Comment le **décrire**?
 - Comment **l'implémenter**?
 - Comment vérifier que c'est **correct**?

FIFO Queue: Enqueue



FIFO Queue: Dequeue

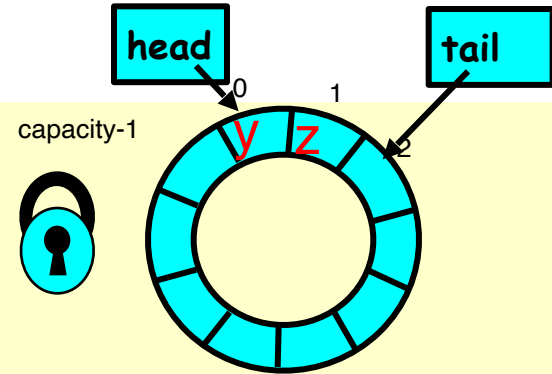


Avec des Locks

```
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];  
    }  
}
```

Avec des Locks

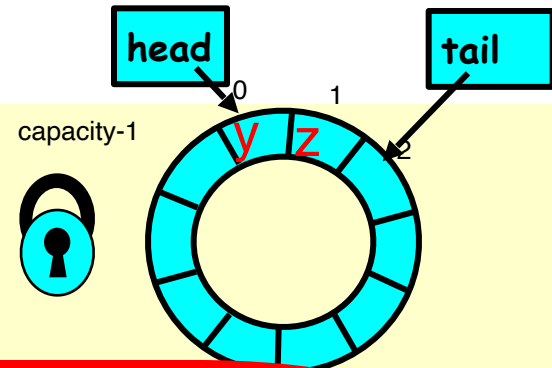
```
class LockBasedQueue<T> {  
    int head, tail;  
    T[] items;  
    Lock lock;  
    public LockBasedQueue(int capacity) {  
        head = 0; tail = 0;  
        lock = new ReentrantLock();  
        items = (T[]) new Object[capacity];  
    }  
}
```



les champs sont
protégées par un
lock partagé

A Lock-Based Queue

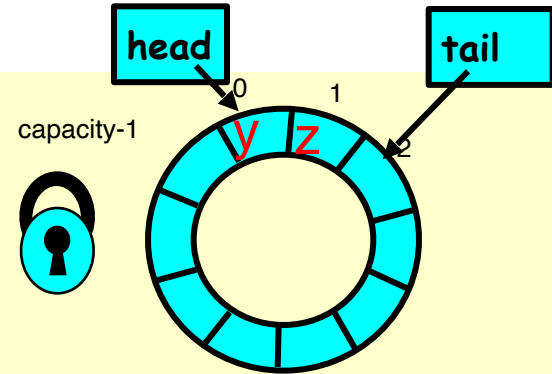
```
class LockBasedQueue<T> {  
    int head, tail;  
    T[] items;  
    Lock lock;  
    public LockBasedQueue(int capacity) {  
        head = 0; tail = 0;  
        lock = new ReentrantLock();  
        items = (T[]) new Object[capacity];  
    }  
}
```



Initialement head = tail

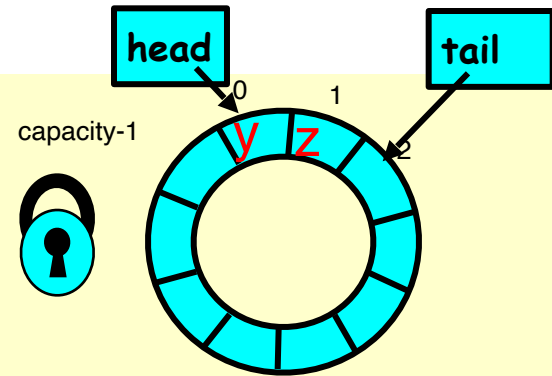
Implementation: Deq

```
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

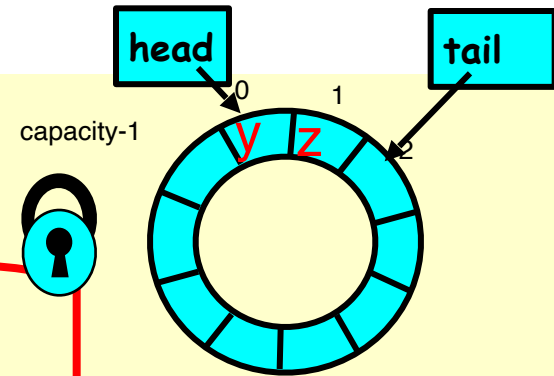
```
public T deq() throws EmptyException {  
    lock.lock();  
    try {  
        if (tail == head)  
            throw new EmptyException();  
        T x = items[head % items.length];  
        head++;  
        return x;  
    } finally {  
        lock.unlock();  
    }  
}
```



Appel de la méthode
en exclusion mutuelle

Implementation: Deq

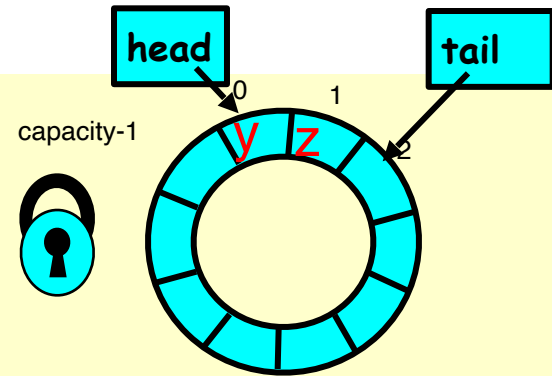
```
public T deq() throws EmptyException {  
    lock.lock();  
    try {  
        if (tail == head)  
            throw new EmptyException();  
        T x = items[head % items.length];  
        head++;  
        return x;  
    } finally {  
        lock.unlock();  
    }  
}
```



si la queue est vide:
exception!

Implementation: Deq

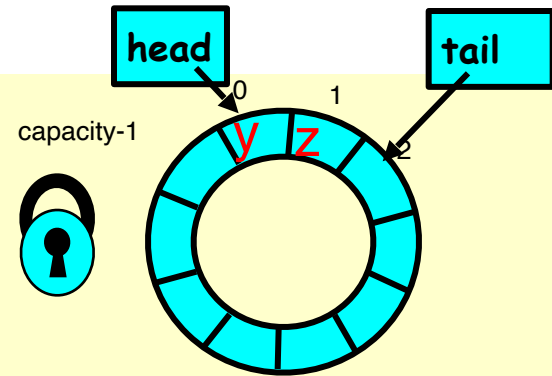
```
public T deq() throws EmptyException {  
    lock.lock();  
    try {  
        if (tail == head)  
            throw new EmptyException();  
        T x = items[head % items.length];  
        head++;  
        return x;  
    } finally {  
        lock.unlock();  
    }  
}
```



Queue non vide:
prendre l'item mettre à
jour head

Implementation: Deq

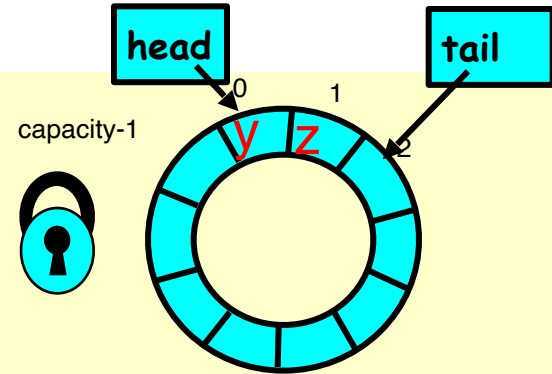
```
public T deq() throws EmptyException {  
    lock.lock();  
    try {  
        if (tail == head)  
            throw new EmptyException();  
        T x = items[head % items.length];  
        head++;  
        return x;  
    } finally {  
        lock.unlock();  
    }  
}
```



Retourner l'item

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();  
    }  
}
```



toujours relâcher le
lock!

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: Enq

```
public void enq(Item x) throws FullException {  
    lock.lock();  
    try {  
        if (tail - head == capacity )  
            throw new FullException();  
        items[tail % capacity] = x; tail++;  
    } finally {  
        lock.unlock();  
    }  
}
```

devrait être correct car toutes
les modifications se font en
exclusion mutuelle

Une autre implementation

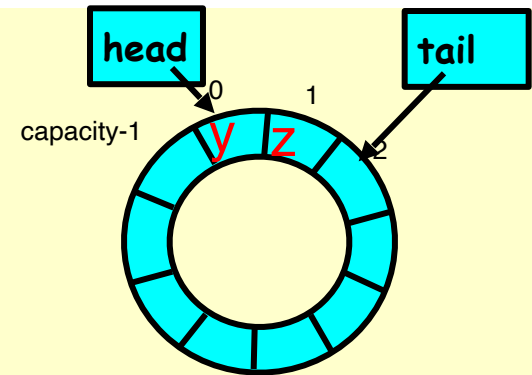
- sans exclusion mutuelle
- Avec seulement deux threads
 - Une thread ne fait que des **enq**
 - L'autre ne fait que des **deq**

« Wait-free » 2-Thread Queue

```
public class WaitFreeQueue {  
  
    volatile int head = 0, tail = 0;  
    volatile items = (T[]) new Object[capacity];  
  
    public void enq(Item x) {  
        while (tail-head == capacity); // busy-wait  
        items[tail % capacity] = x; tail++;  
    }  
    public Item deq() {  
        while (tail == head); // busy-wait  
        Item item = items[head % capacity]; head++;  
        return item;  
    }  
}
```

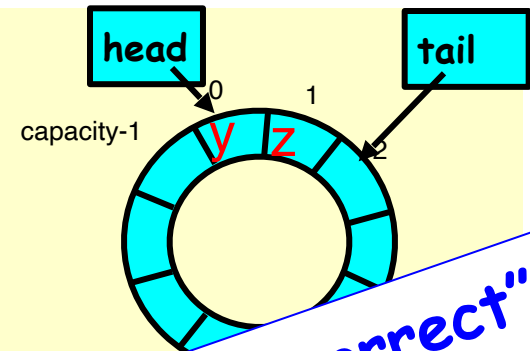
Wait-free 2-Thread Queue

```
public class LockFreeQueue {  
  
    volatile int head = 0, tail = 0;  
    volatile items = (T[]) new Object[capacity];  
  
    public void enq(Item x) {  
        while (tail-head == capacity); // busy-wait  
        items[tail % capacity] = x; tail++;  
    }  
  
    public Item deq() {  
        while (tail == head); // busy-wait  
        Item item = items[head % capacity]; head++;  
        return item;  
    }  
}
```



Lock-free 2-Thread Queue

```
public class LockFreeQueue {  
  
    volatile int head = 0, tail = 0;  
    volatile items = (T[])new Object[capacity];  
  
    public void enq(Item x) {  
        while (tail == head); // busy wait  
        items[tail % capacity] = x; tail++;  
    }  
  
    public Item deq() {  
        while (tail == head); // busy-wait  
        Item item = items[head % capacity]; head++;  
        return item;  
    }  
}
```



Que signifie "correct" si la modification ne sont pas n exclusion mutuelle?

Queue modifiée sans lock!

Que doit être une implémentation d'une queue concurrente?

- Il faut spécifier une queue concurrente
- Il faut pouvoir prouver que l'algorithme implémente la spécification de l'objet

Correction et Progression

- safety
- liveness
- définir
 - quand une implémentation est correcte
 - les conditions qui garantissent la progression

Objet séquentiel

- Un objet a un **état**
 - donné par les **champs** de l'objet
 - Queue: la séquence des items
- Un objet a un ensemble de **méthodes**
 - On ne peut accéder à l'objet que par ces méthodes
 - Queue: enq **et** deq

Spécification Séquentielle

- If (precondition)
 - l'objet est dans un certaine état
 - avant l'appel de la méthode,
- Then (postcondition)
 - la méthode retournera une valeur particulière
 - ou « throw » une exception particulière.
- Et (postcondition)
 - l'object sera dans un nouvel état quand la méthode retournera

Pre et PostConditions pour Dequeue

- Precondition:
 - La Queue est non-vide
- Postcondition:
 - Retourne le premier item dans la queue
- Postcondition:
 - Supprime ce premier item de la queue

Pre et PostConditions pour Dequeue

- Precondition:
 - Queue est vide
- Postcondition:
 - Throws Empty exception
- Postcondition:
 - L'état de la Queue est inchangé

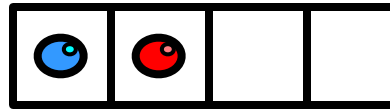
Avantages de la spécification séquentielle

- Décrit toutes les interactions entre méthodes par les effets sur l'état de l'objet
- Documentation
 - chaque méthode est décrite en isolation
 - on peut ajouter de nouvelles méthodes sans changer la descriptions des anciennes

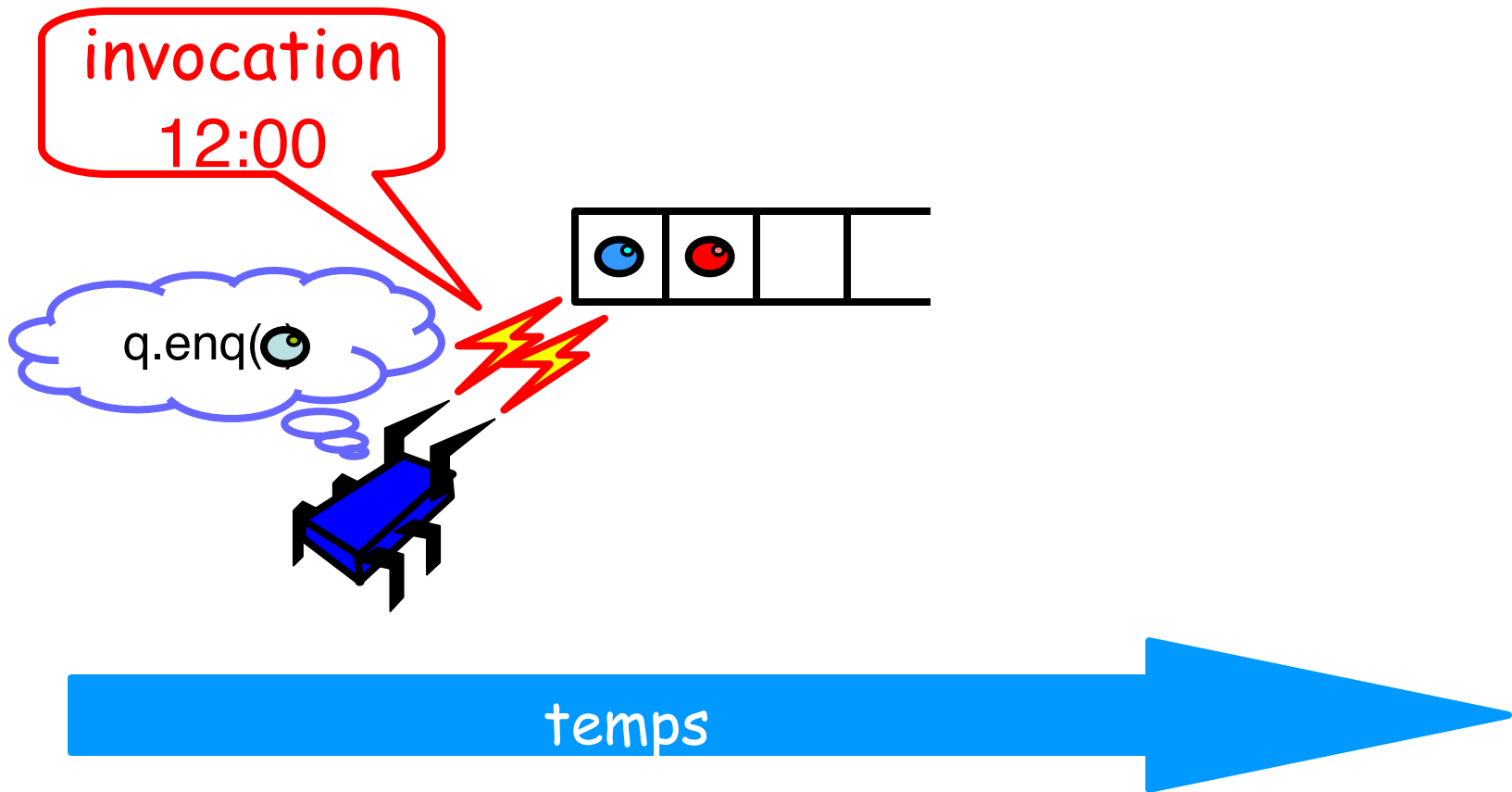
Et pour des spécifications concurrentes ?

- Methodes?
- Documentation?
- Ajouter des nouvelles méthodes?

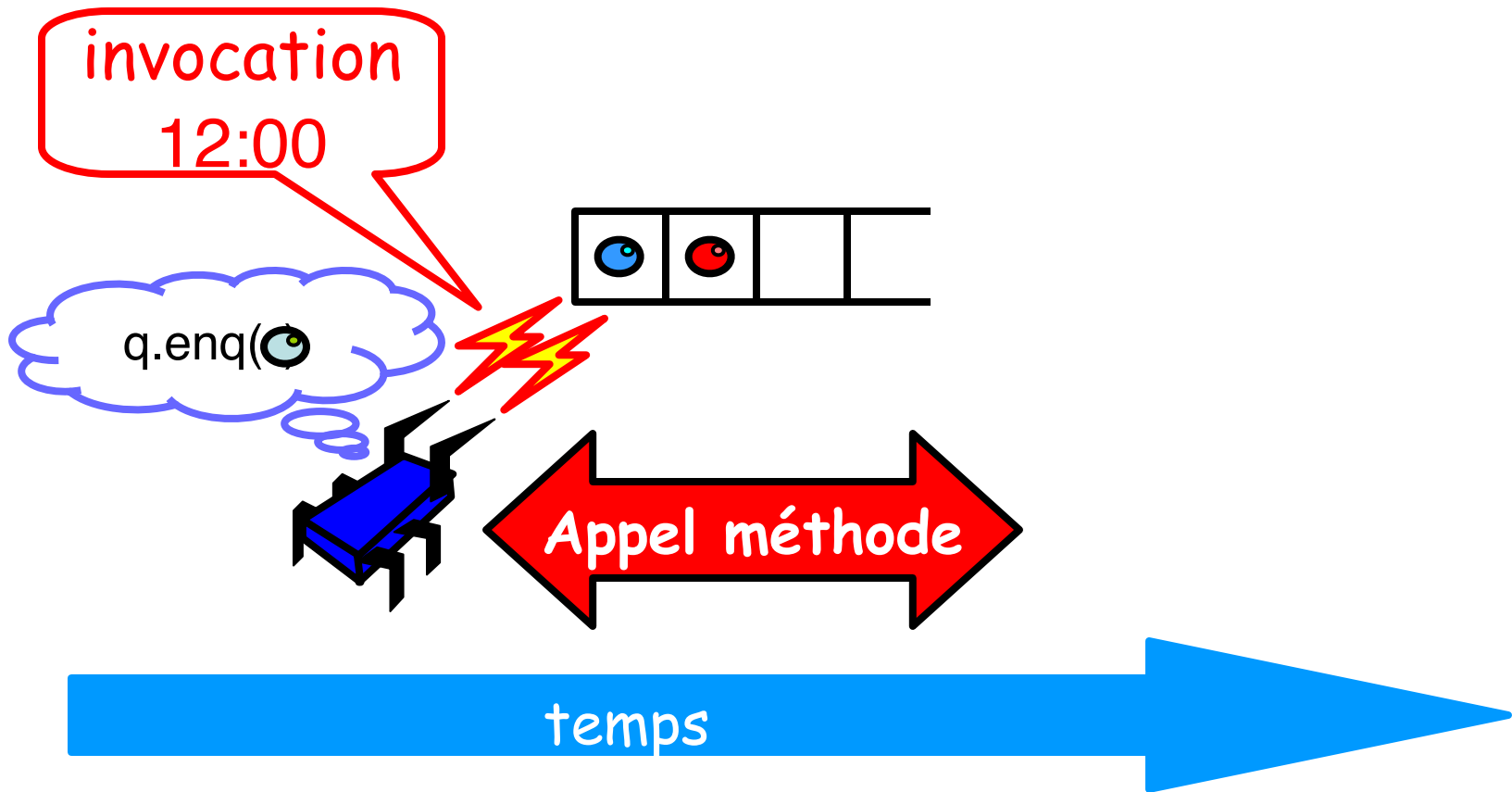
Les Méthodes prennent du temps



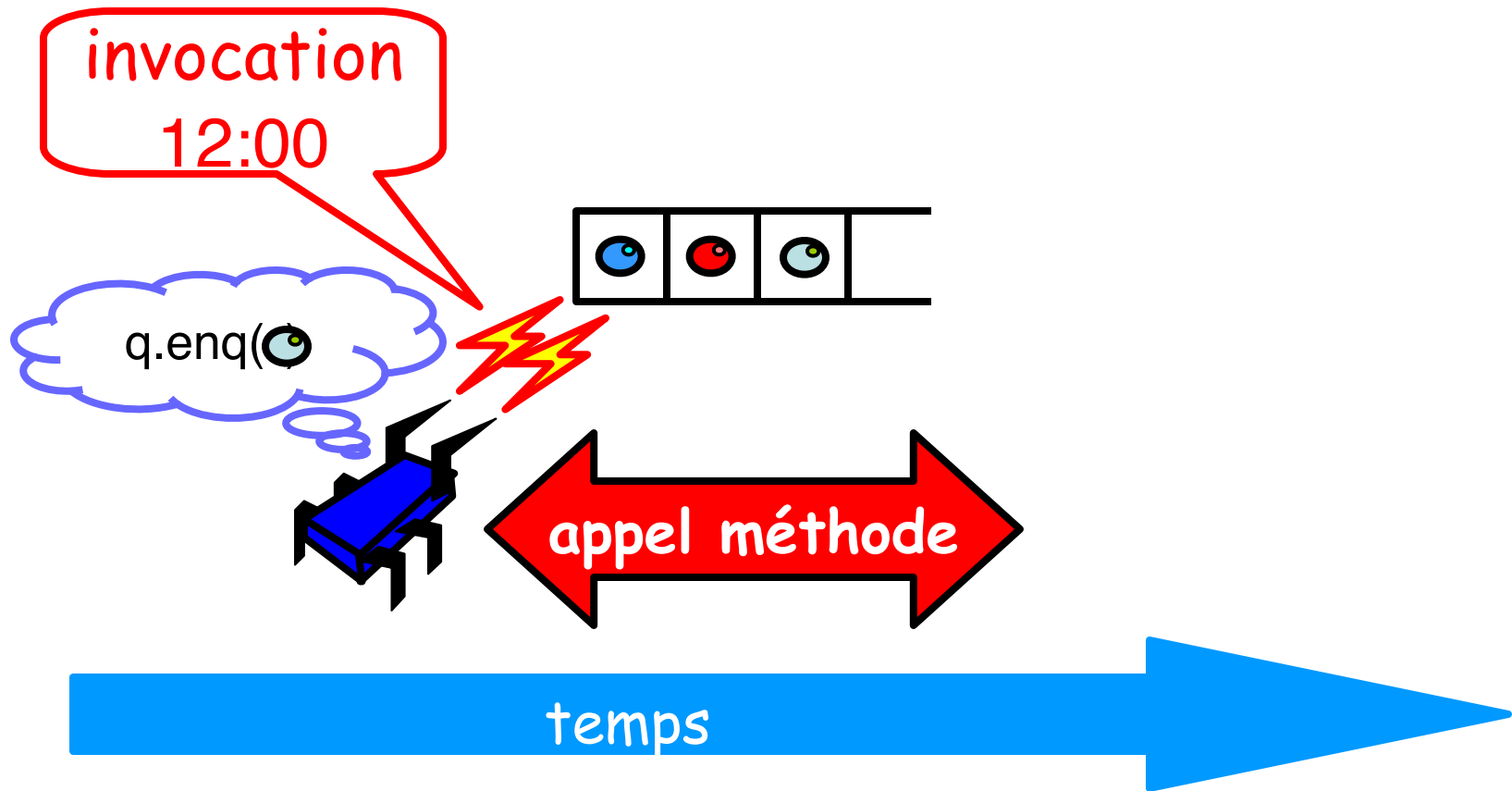
Les Méthodes prennent du temps



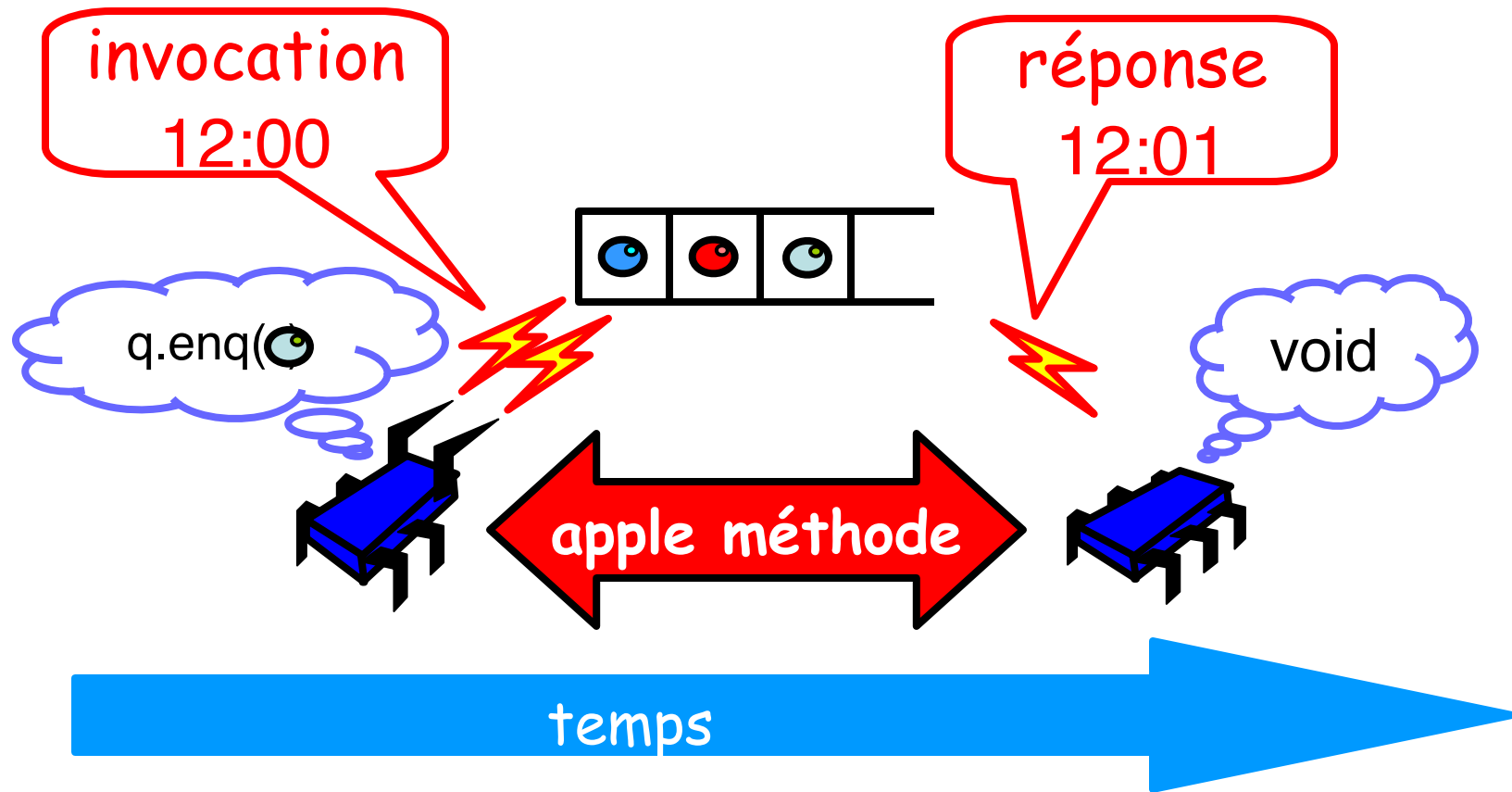
Les Méthodes prennent du temps



Les Méthodes prennent du temps



Les Méthodes prennent du temps

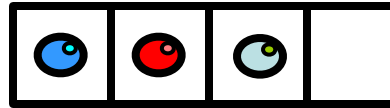


Séquentiel vs Concurrent

- Séquentiel
 - Méthodes prennent du temps ? Qui s'en préoccupe?
- Concurrent
 - L'appel d'une méthode n'est pas un événement
 - L'appel d'une méthode est un intervalle.

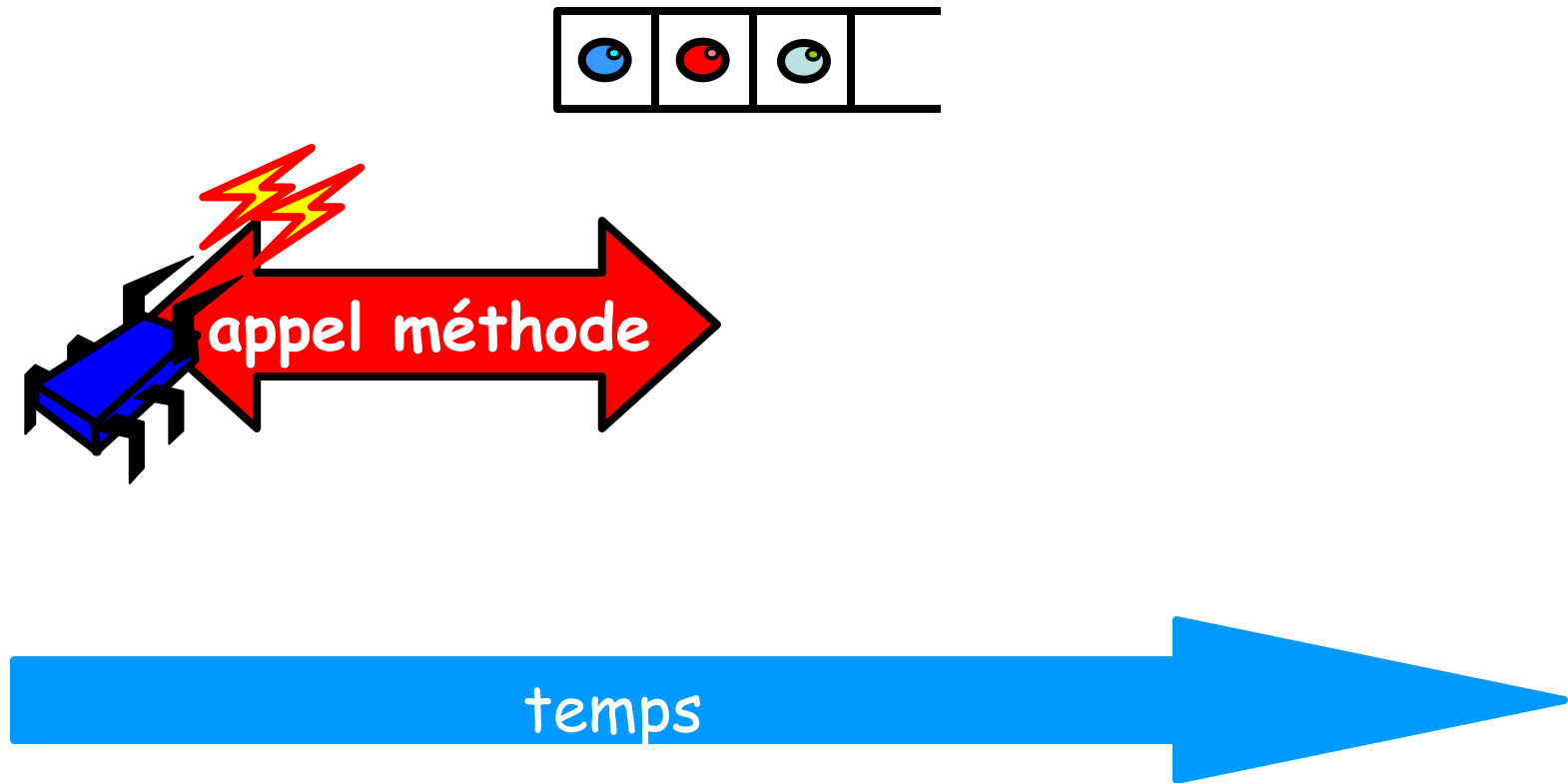
Les méthodes concurrentes prennent du temps

Recouvrement



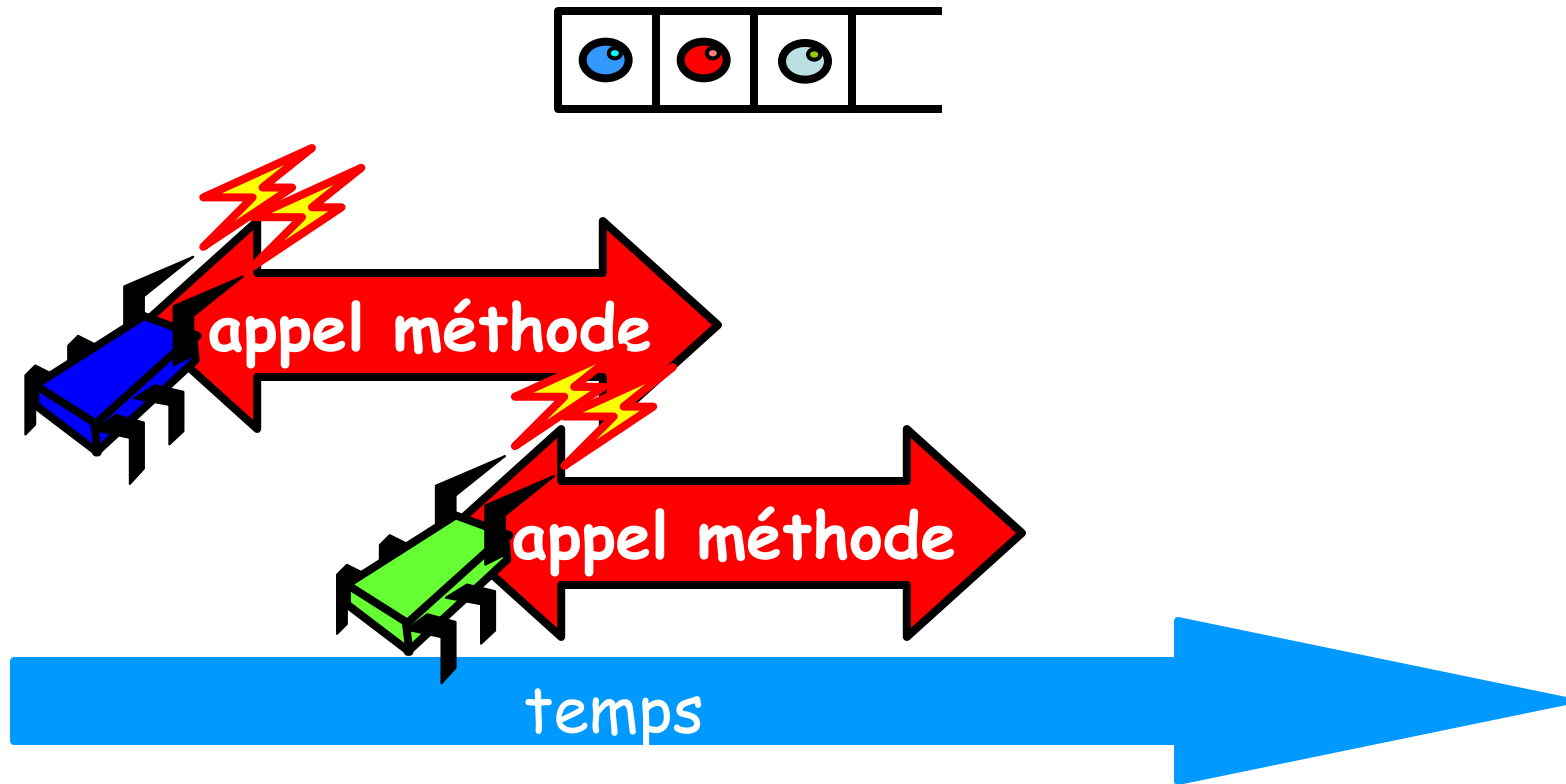
Les méthodes concurrentes prennent du temps

Recouvrement



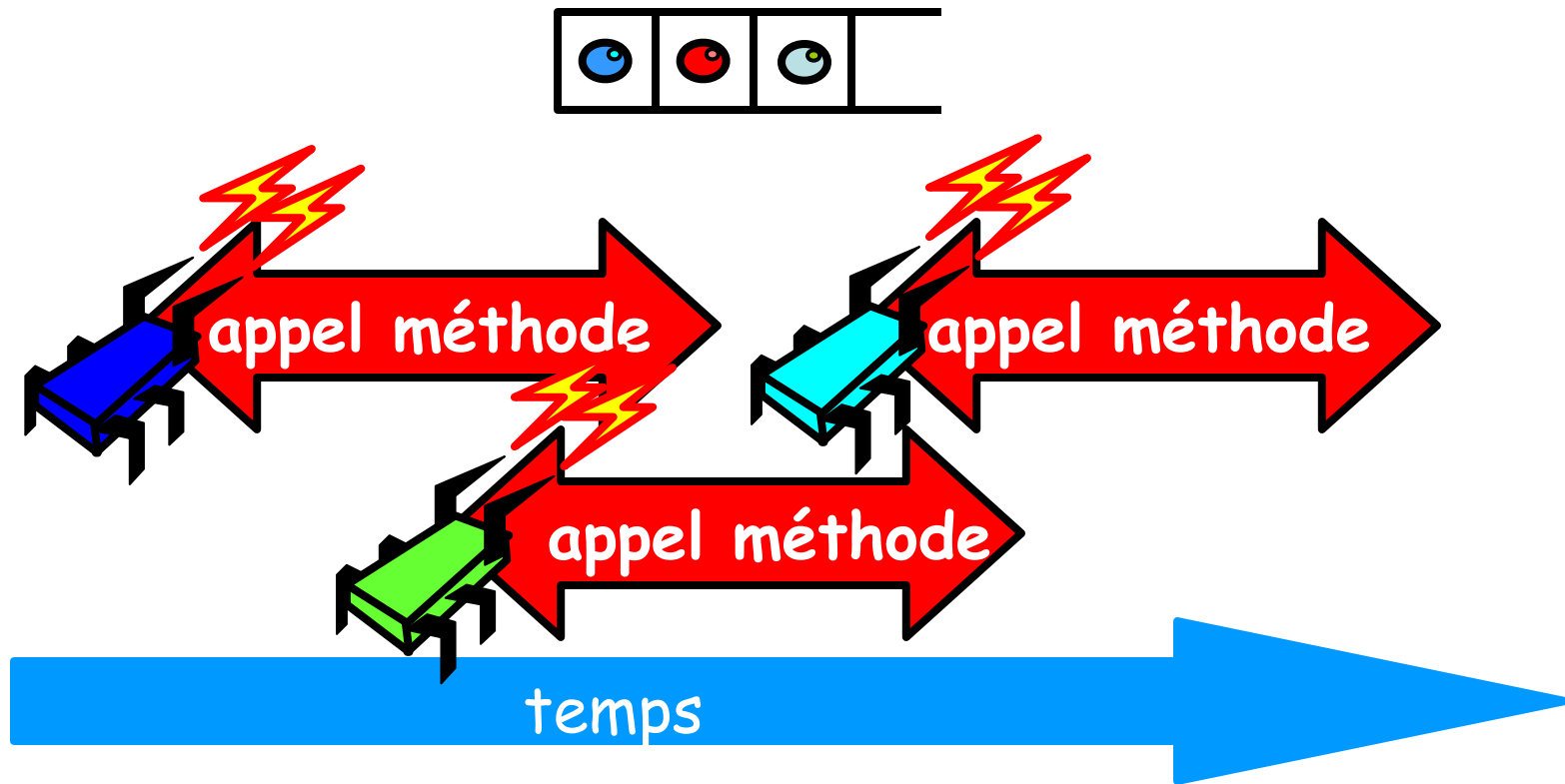
Les méthodes concurrentes prennent du temps

Recouvrement



Les méthodes concurrentes prennent du temps

Recouvrement



Séquentiel vs Concurrent

- Séquentiel:
 - L'état d'un objet n'est significatif qu'entre des appels de méthodes
- Concurrent
 - Comme les appels des méthodes peuvent se recouvrir un objet peut ne jamais être entre des appels de méthodes

Séquentiel vs Concurrent

- Séquentiel:
 - chaque méthode est décrite en isolation
- Concurrent
 - On doit caractériser toutes les interactions possibles entre les appels concurrents
 - Si deux enq se recouvrent?
 - deux deqs? enq et deq? ...

Séquentiel vs Concurrent

- Séquentiel:
 - On peut ajouter de nouvelles méthodes (sans effet sur les anciennes)
- Concurrent:
 - Tout peut potentiellement interagir

Question

- Que signifie qu'un objet concurrent est correct?
- Qu'est ce qu'une file FIFO concurrente?
- FIFO suppose un ordre strict temporel
 - Concurrent signifie un ordre temporel ambiguë

Intuitivement...

```
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();  
    }  
}
```

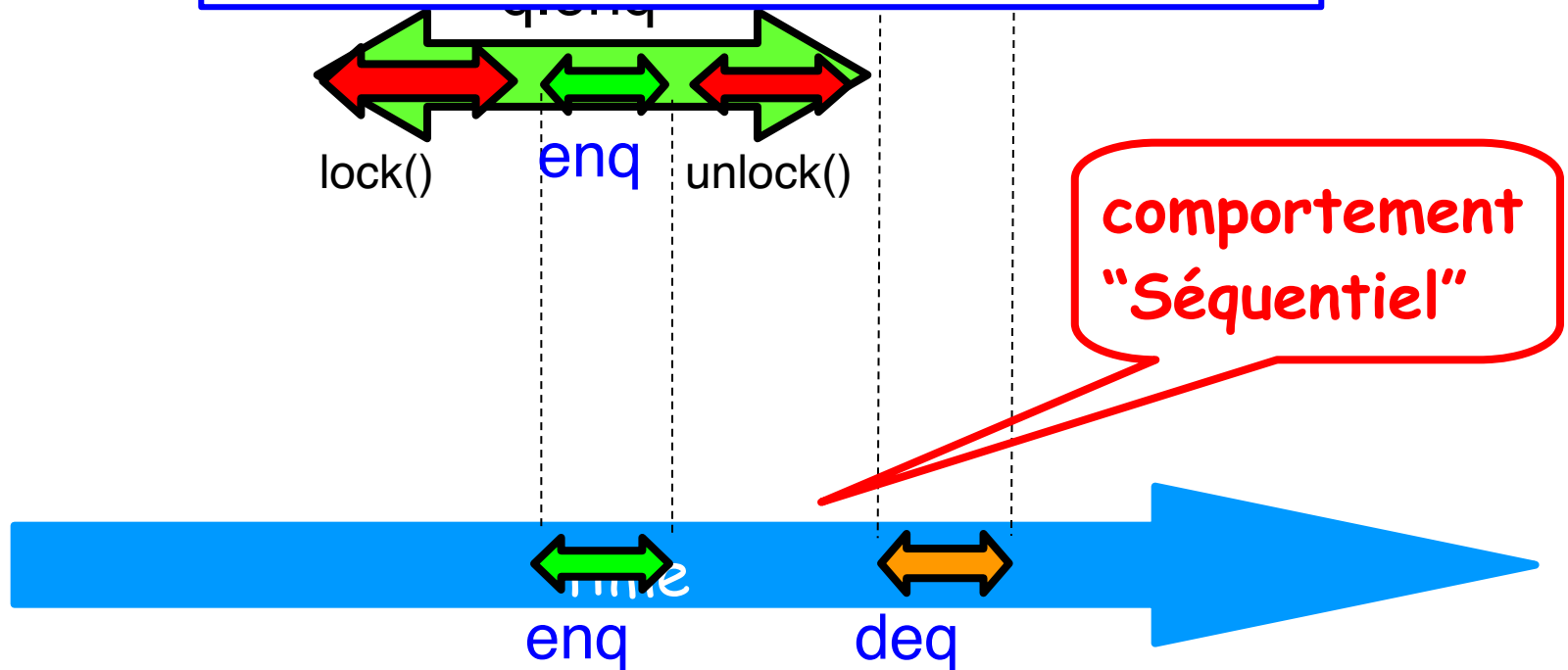
Intuitivement...

```
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();  
    }  
}
```

toutes les
modifications de la
file se font en
exclusion mutuelle

Intuitivement

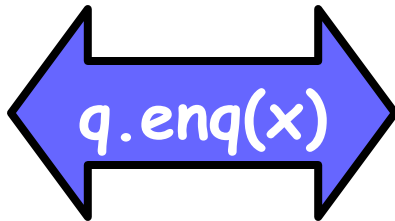
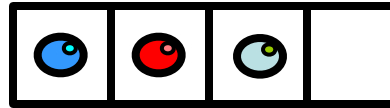
décrire le comportement concurrent
par le comportement séquentiel



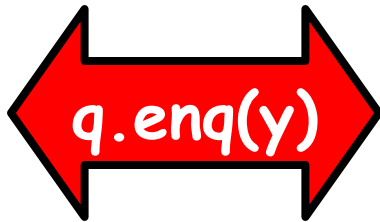
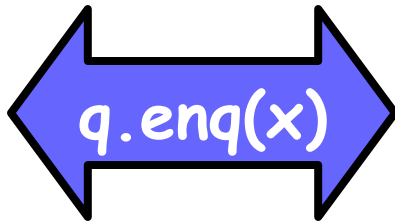
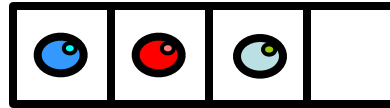
Propriété de l'objet?

- Chaque méthode doit:
 - “prendre effet” (take effect)
 - Instantanément
 - Entre l'invocation et la response
- Propriété d'une exécution...
- Un objet linéarisable: toutes les exécutions peuvent être linéarisées

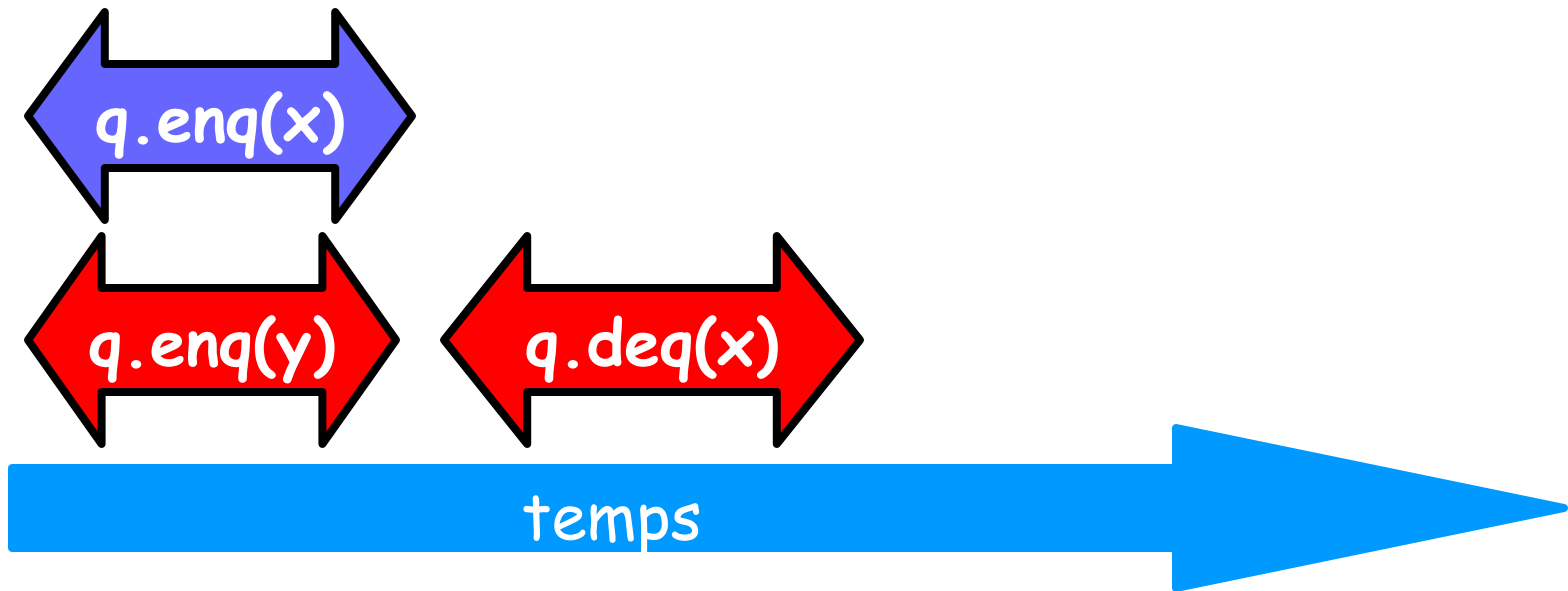
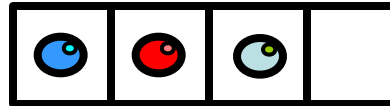
Exemple



Exemple

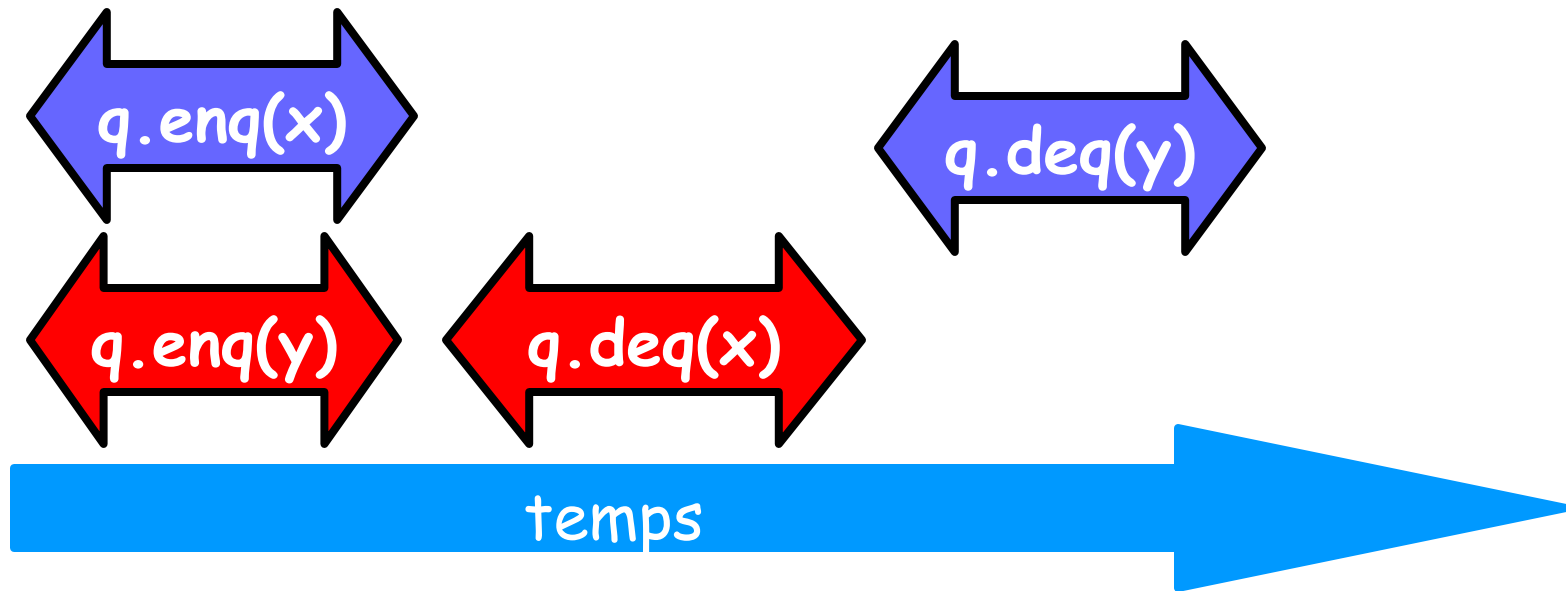
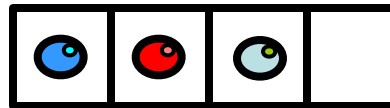


Exemple

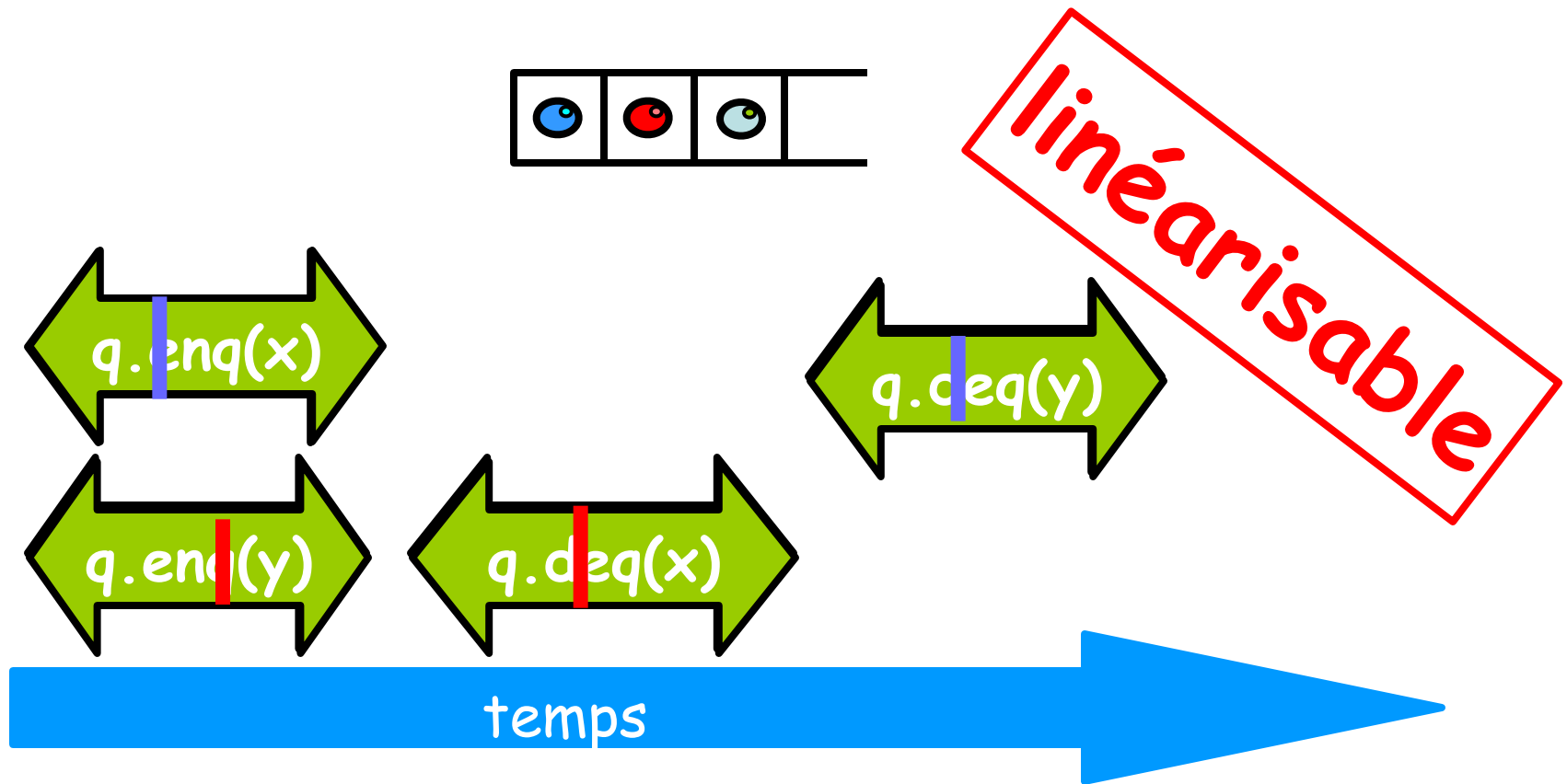




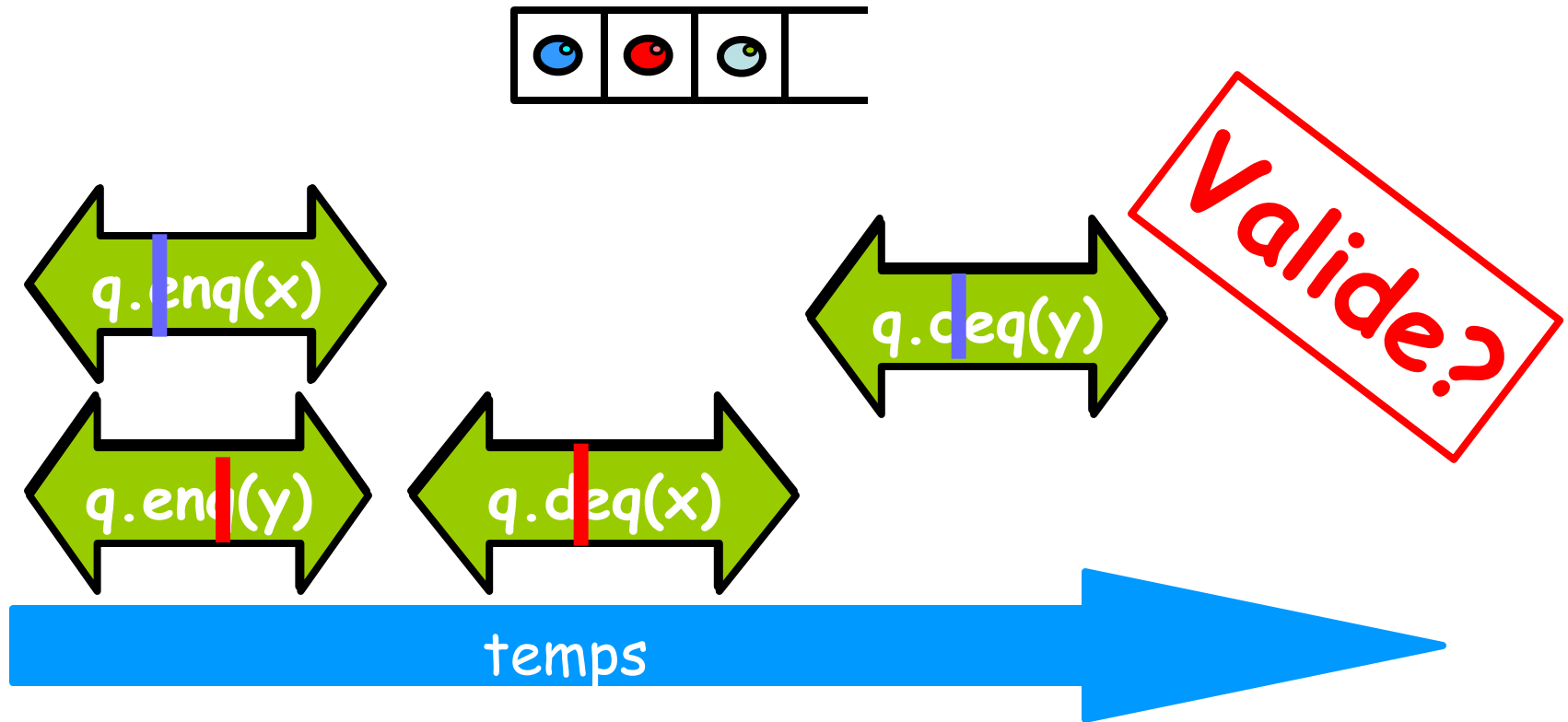
Exemple



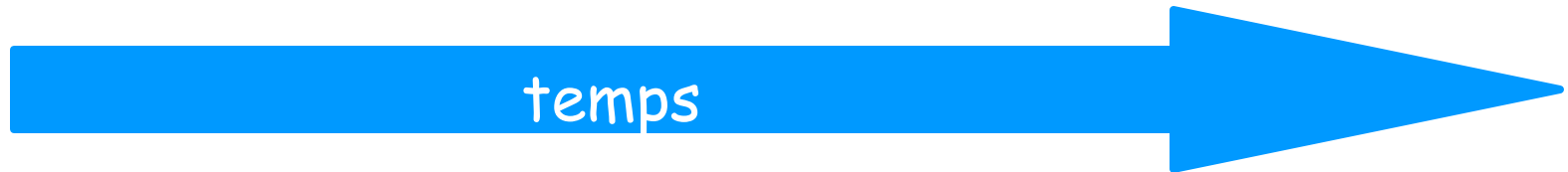
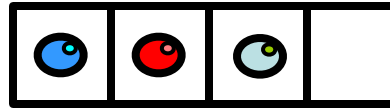
Exemple



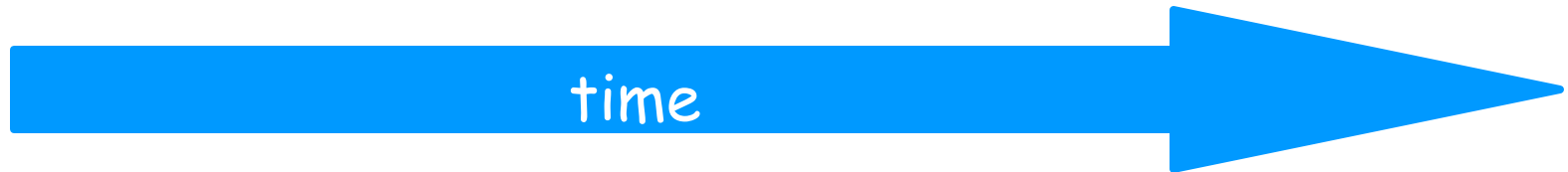
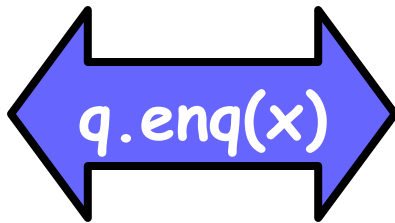
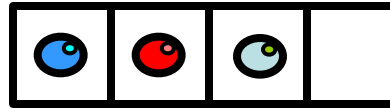
Exemple



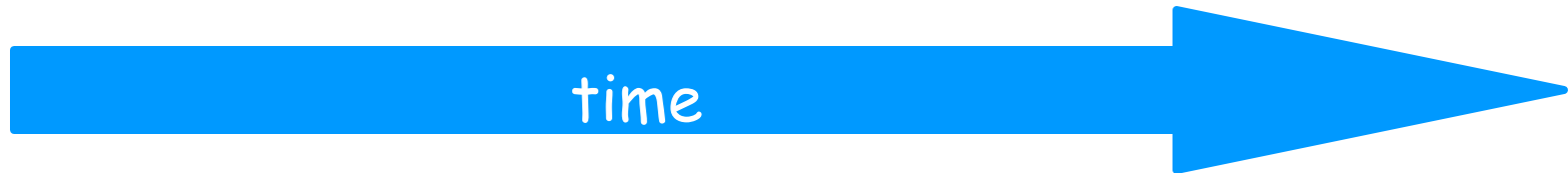
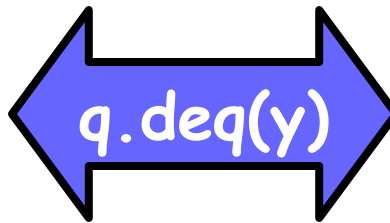
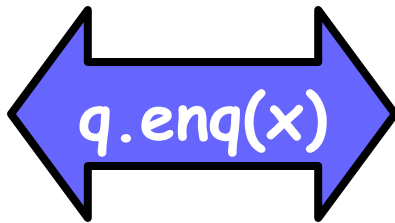
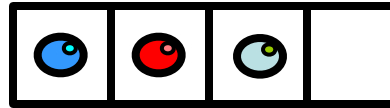
Exemple



Example

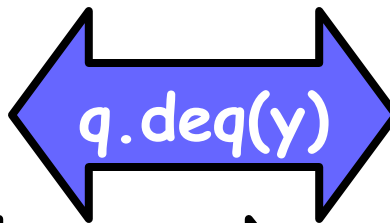
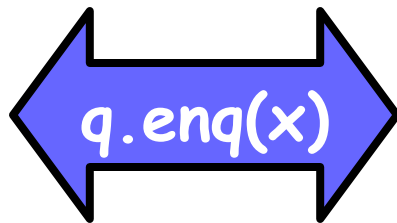
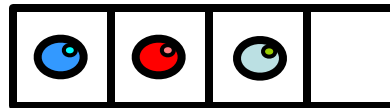


Exemple



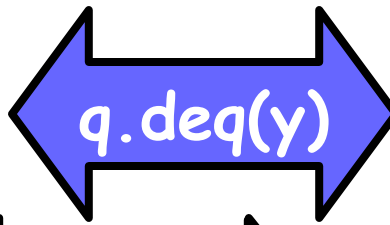
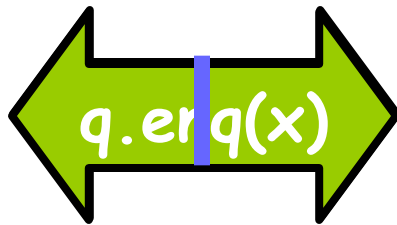
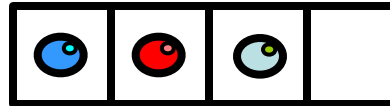


Exemple





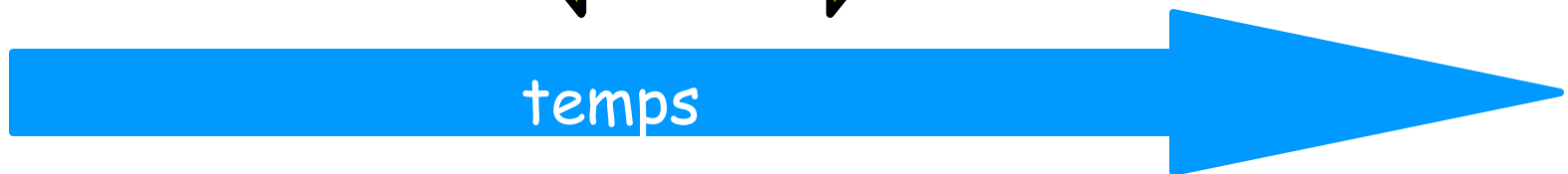
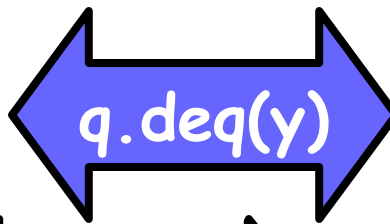
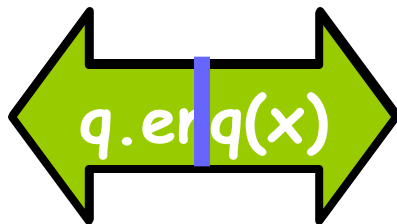
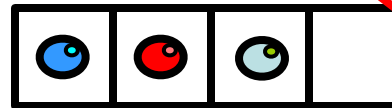
Exemple



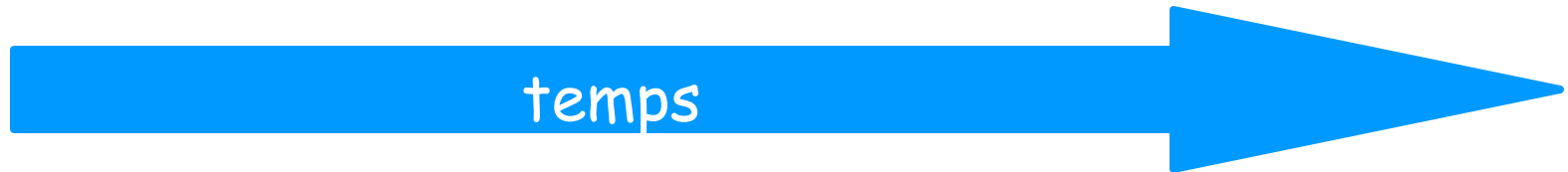
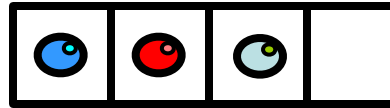


Exemple

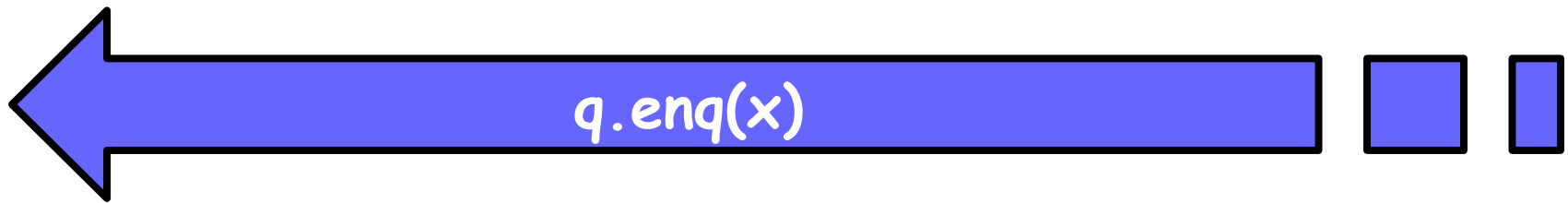
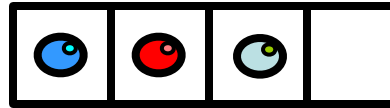
non linéarisable



Exemple

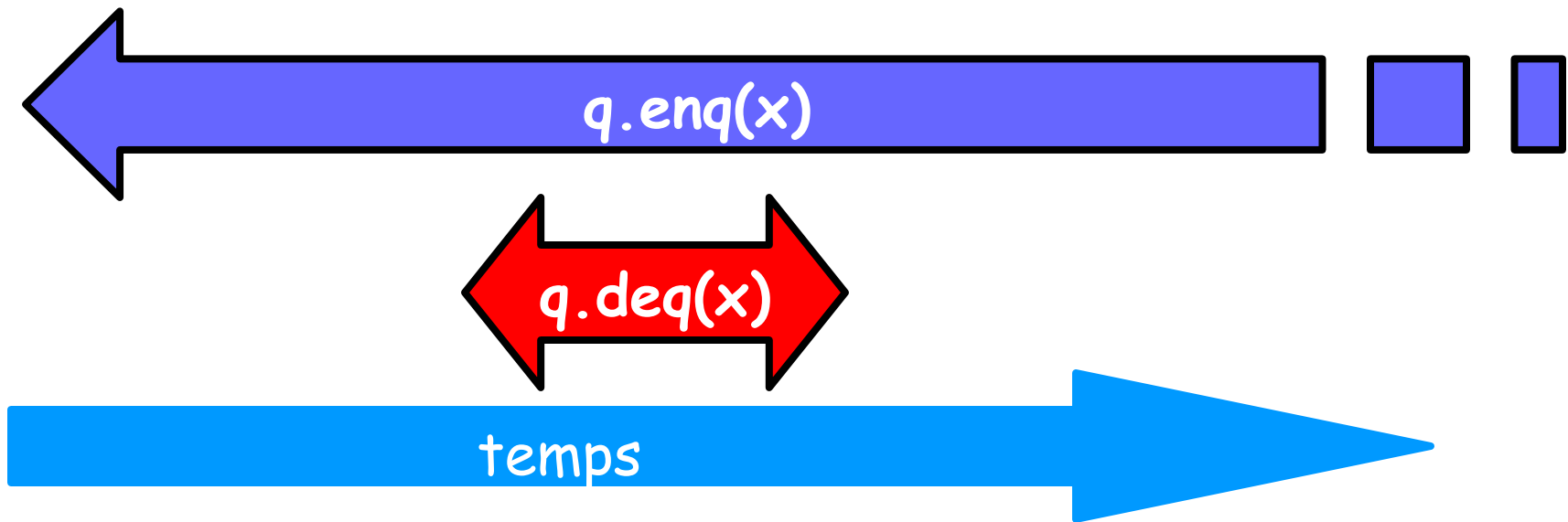
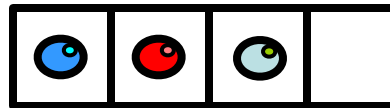


Exemple



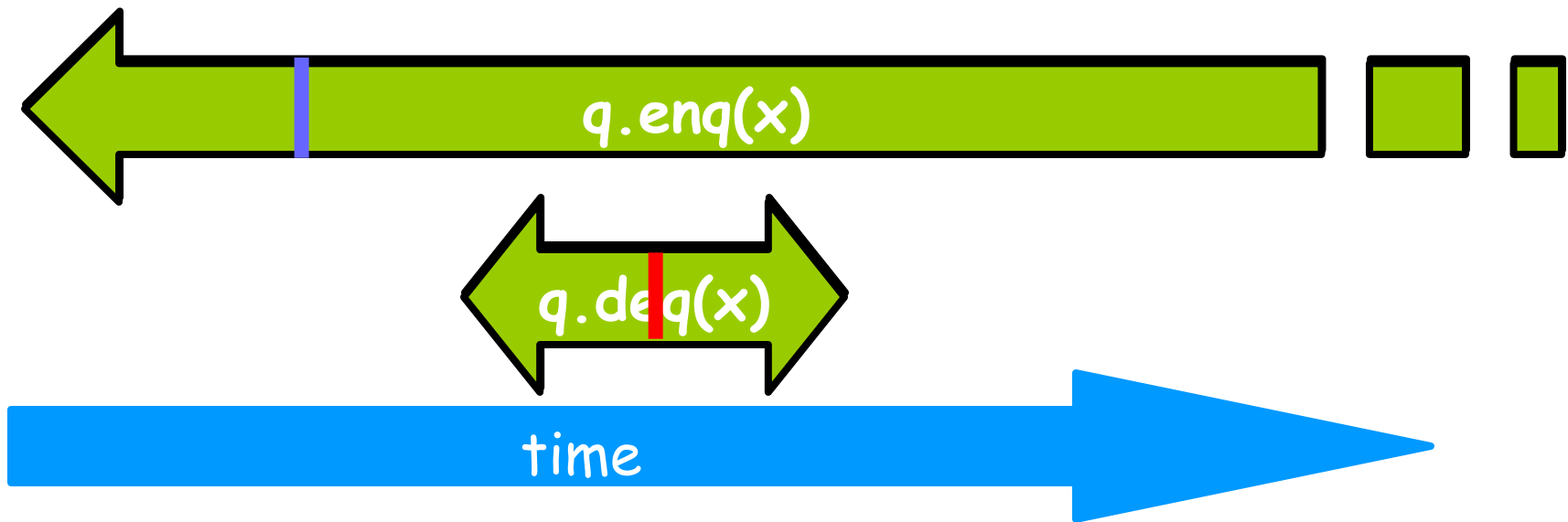
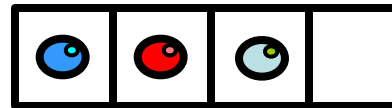


Exemple



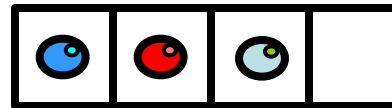


Exemple





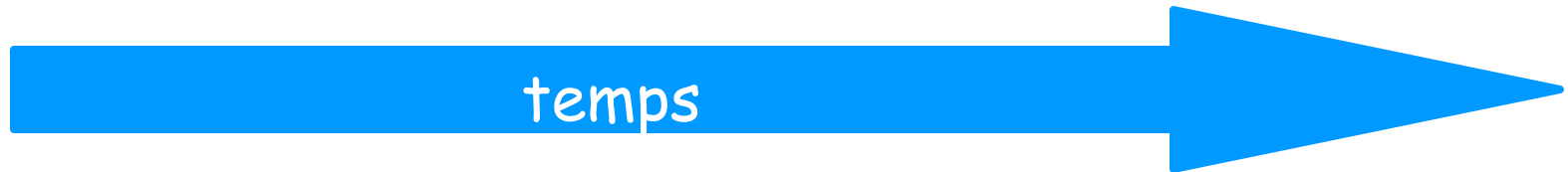
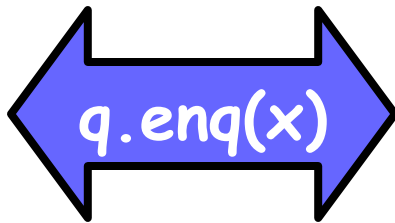
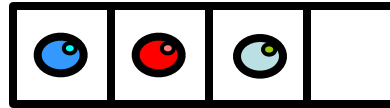
Exemple



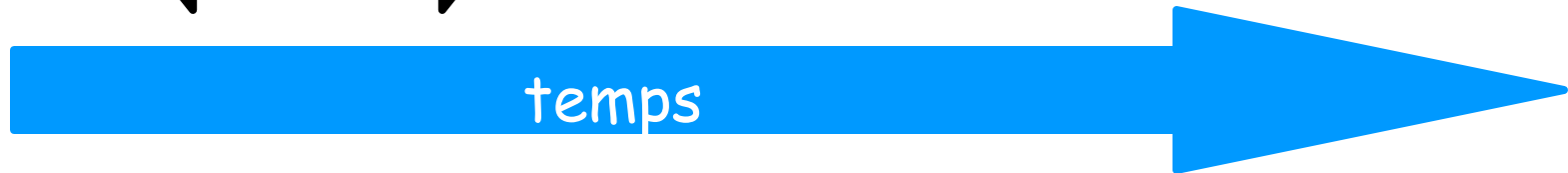
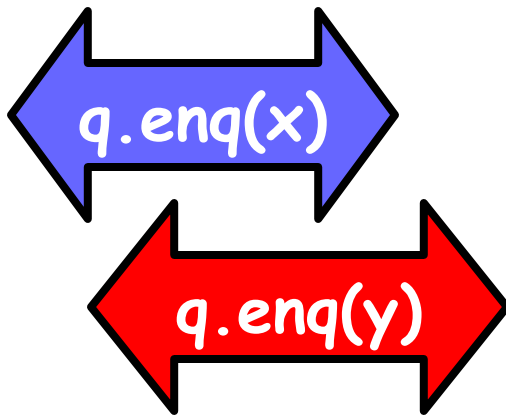
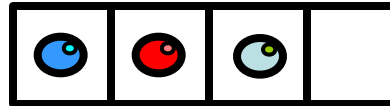
linéarisable



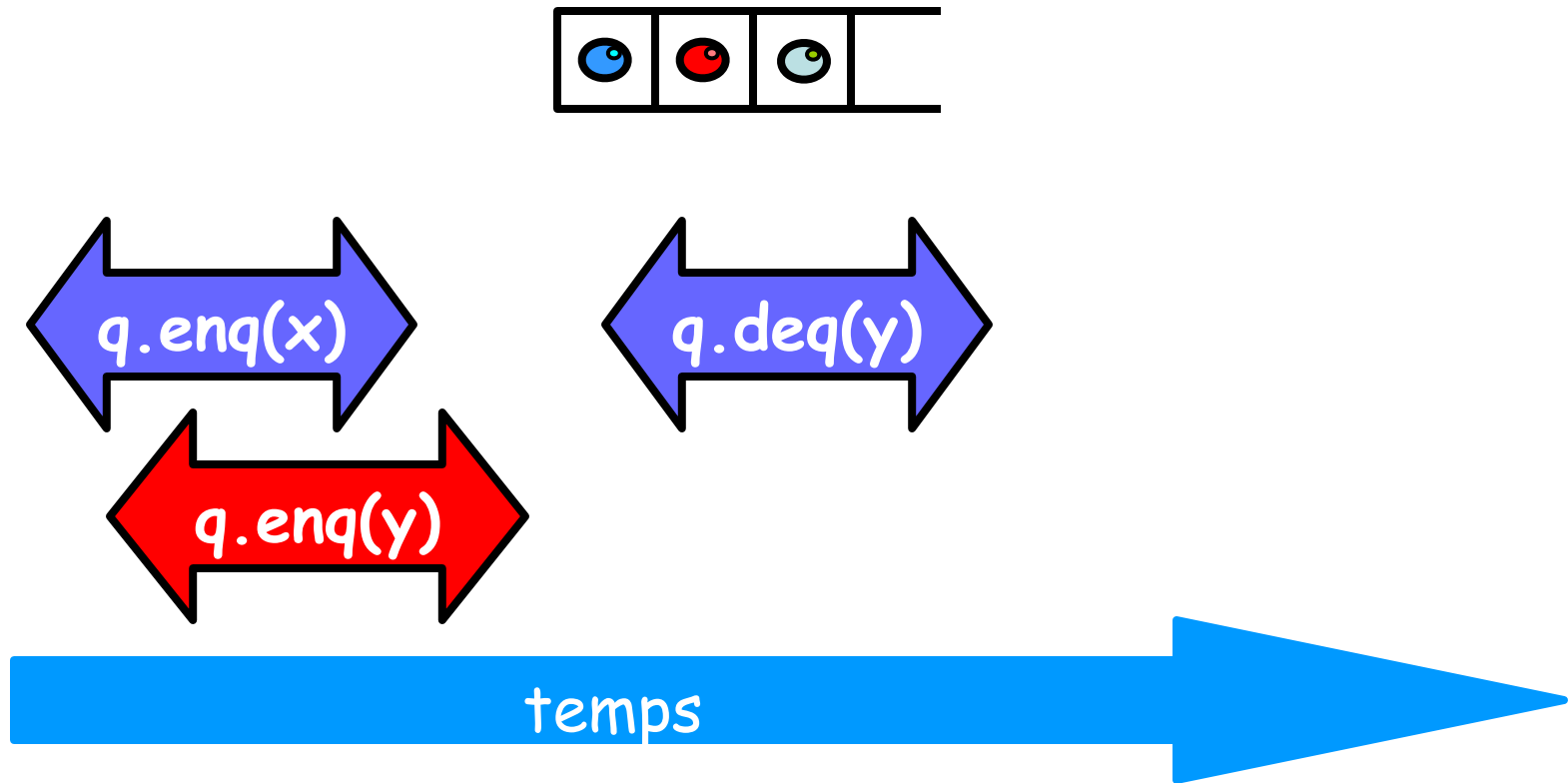
Exemple



Exemple

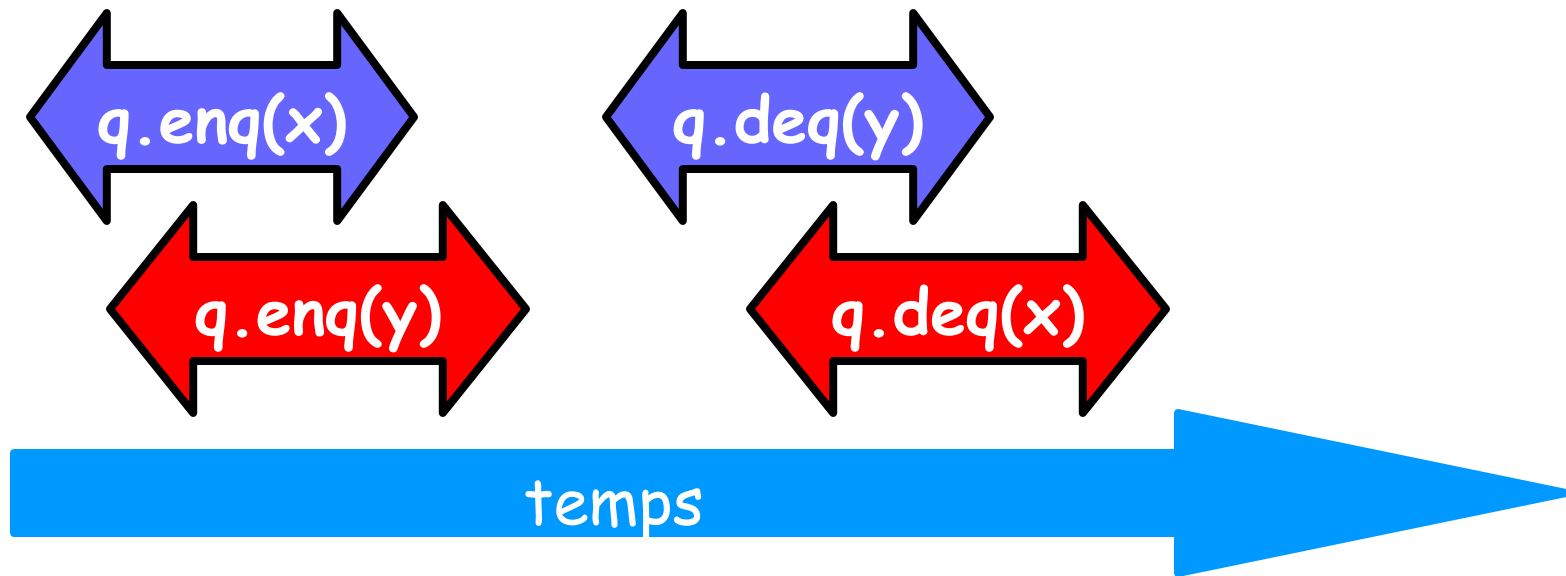
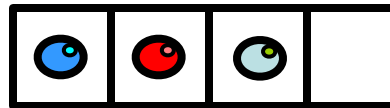


Exemple

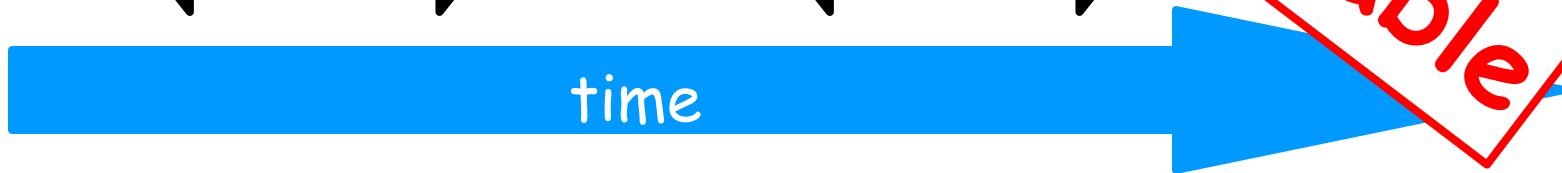
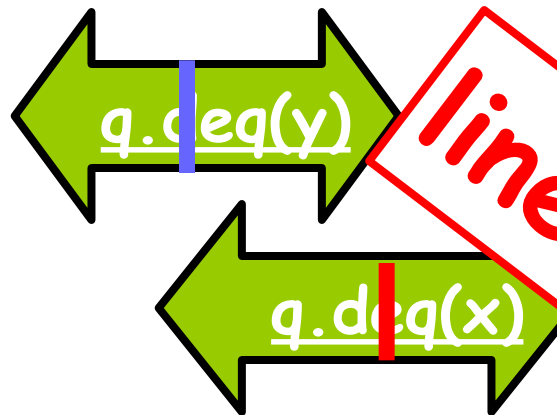
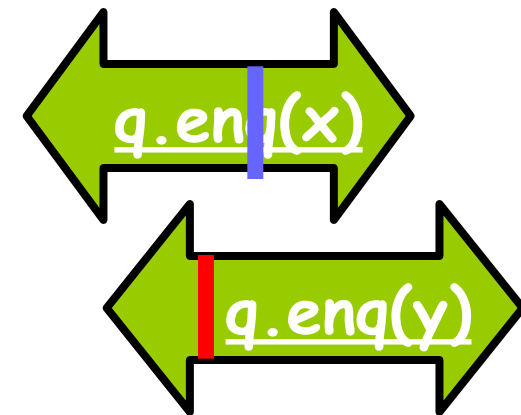
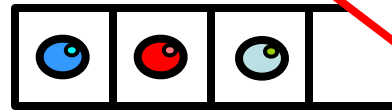




Exemple



Comme ci Exemple
Comme ça

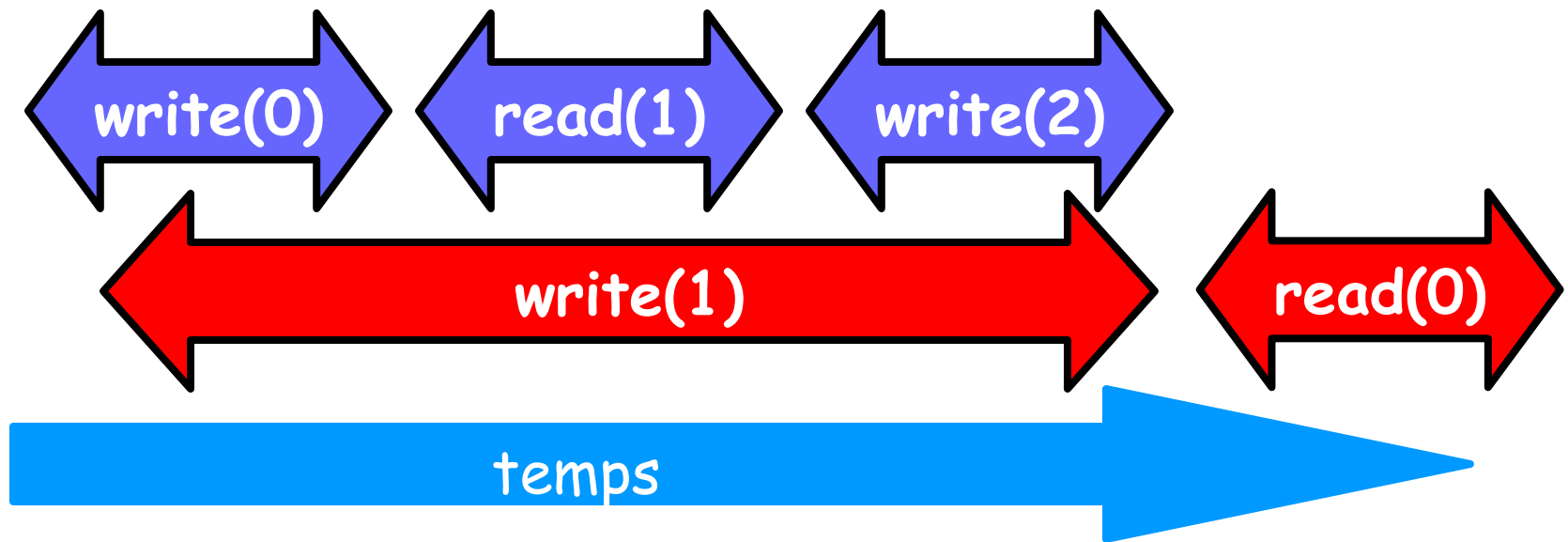


ordres multiples OK

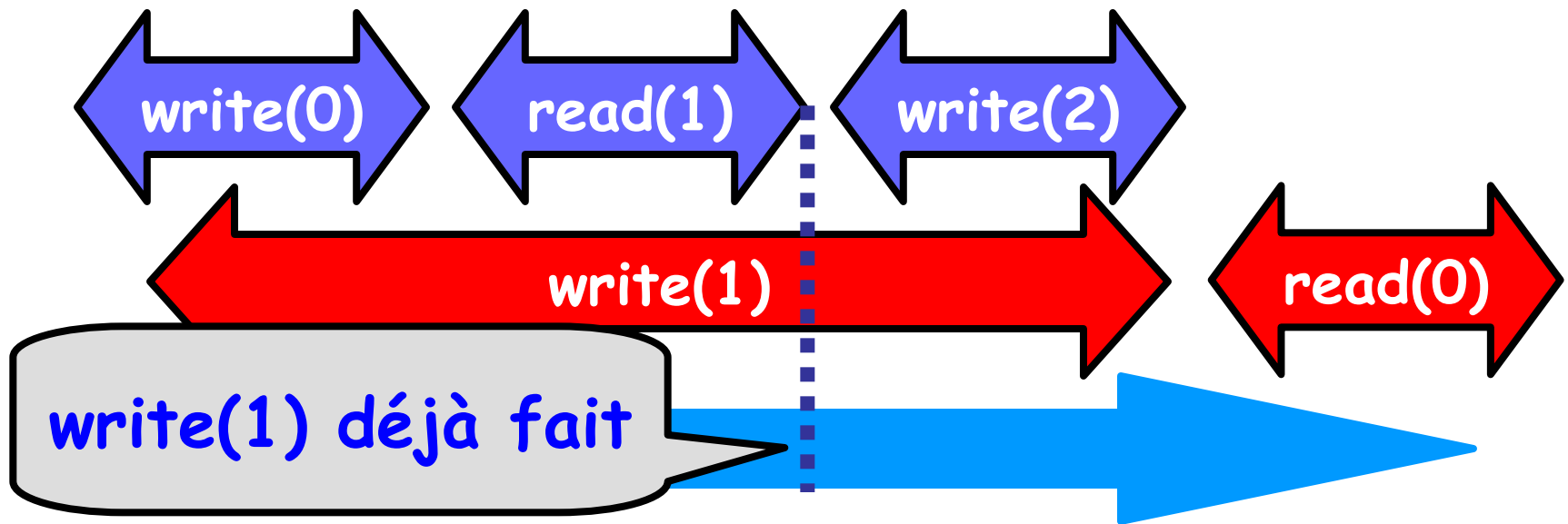
linéarisable

Exemple simple : registres

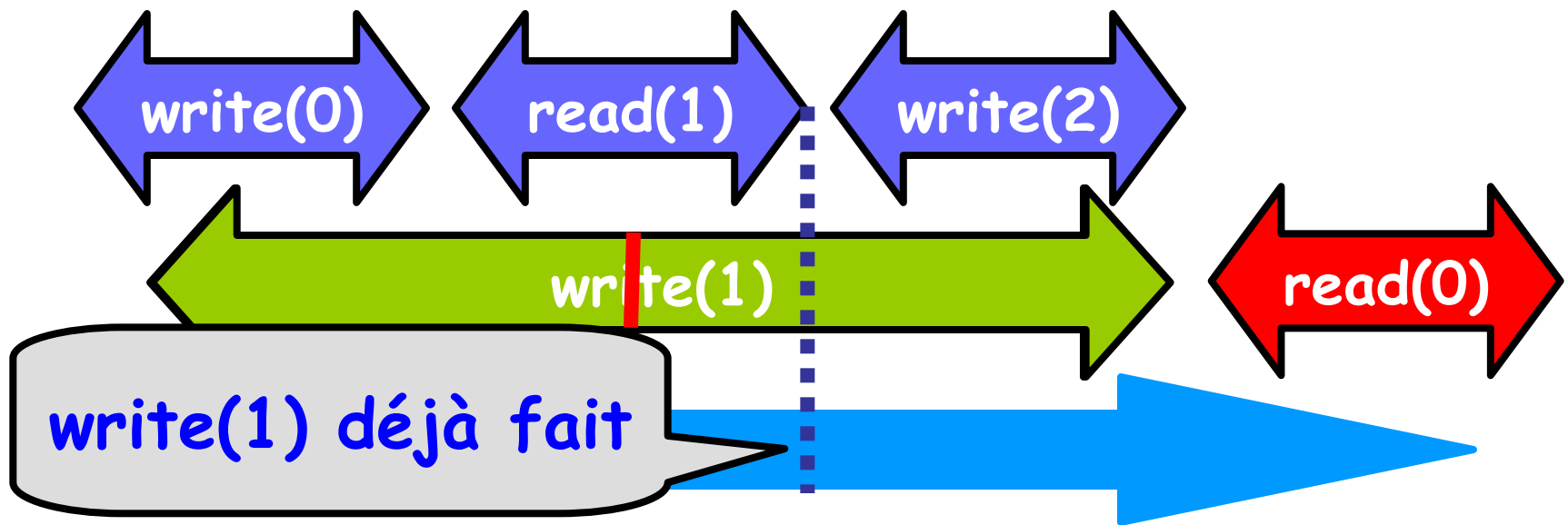
Read/Write Registre Exemple



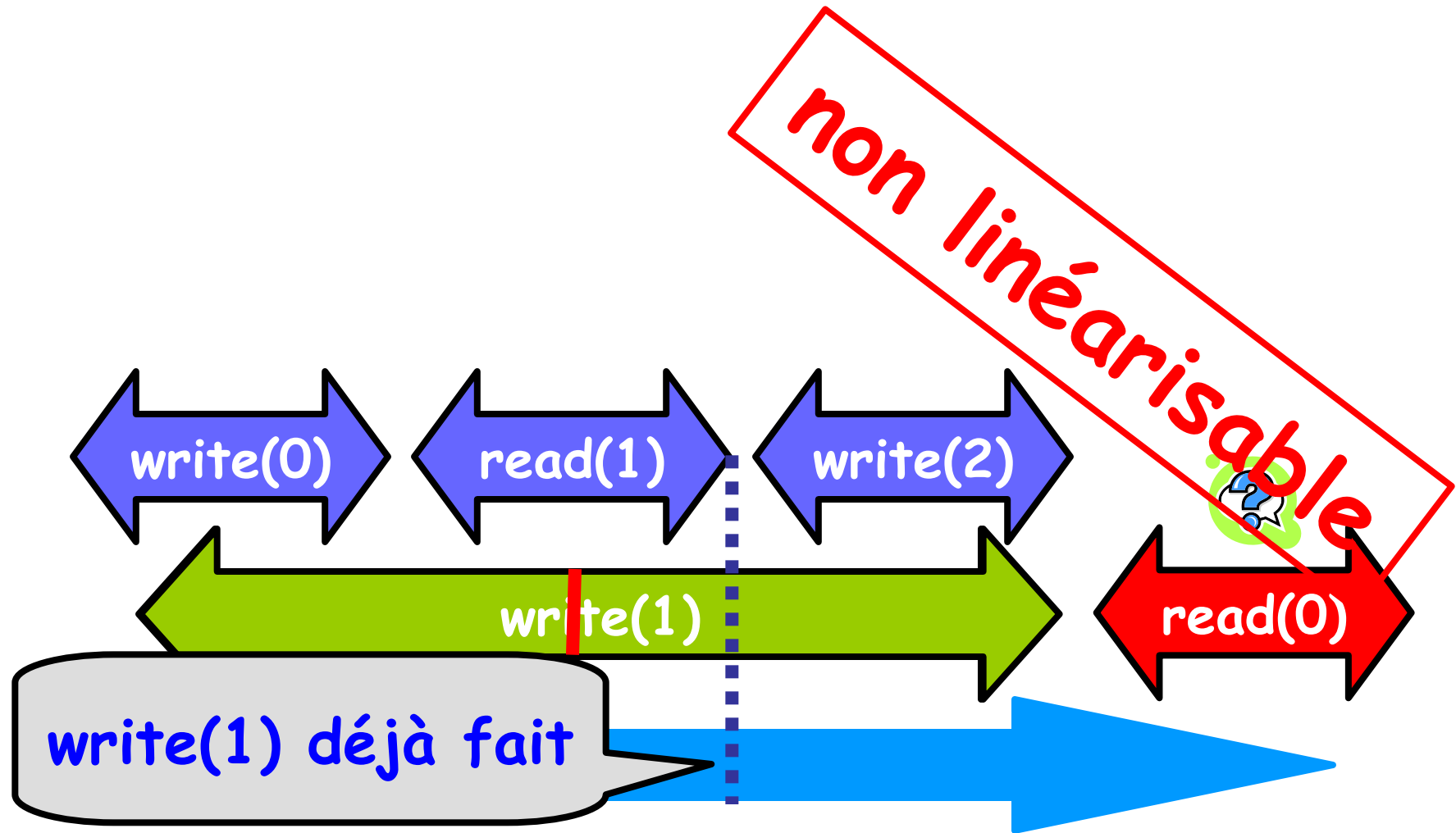
Read/Write Registre Exemple



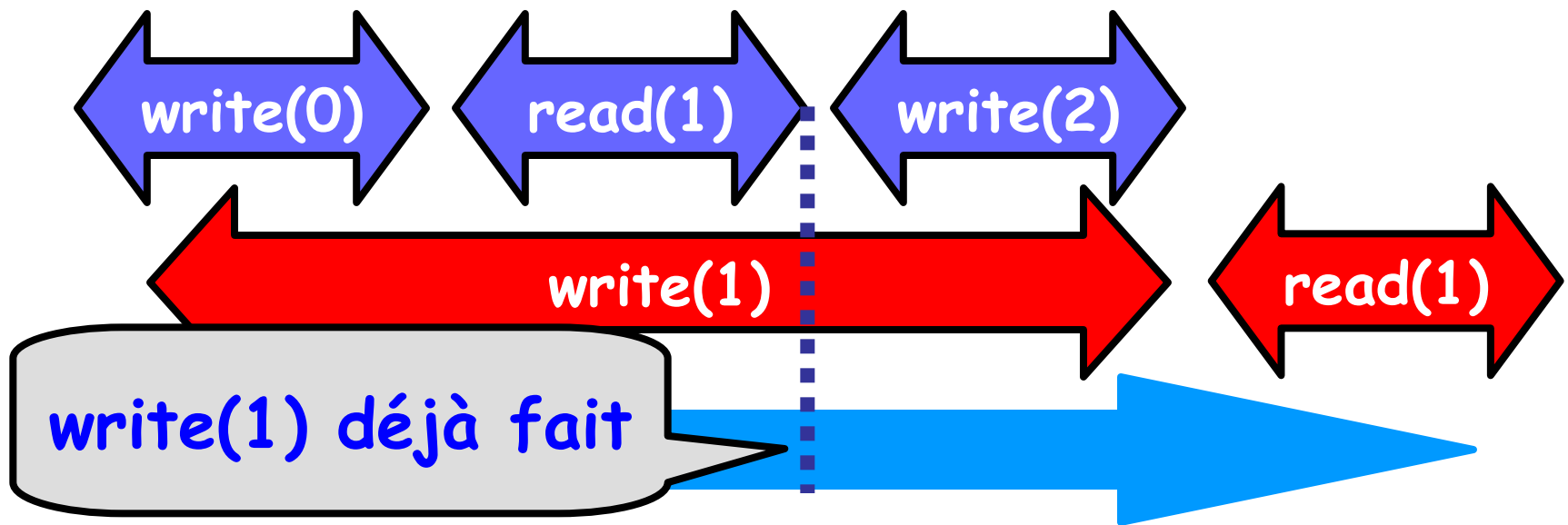
Read/Write Registre Exemple



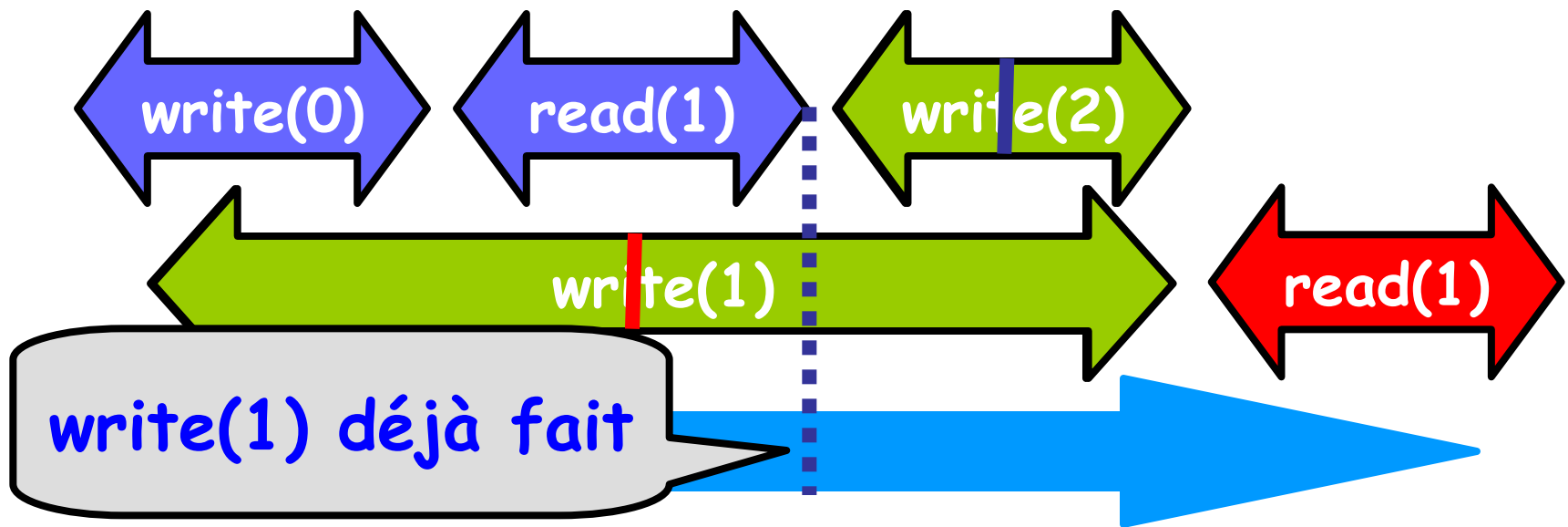
Read/Write Registre Exemple



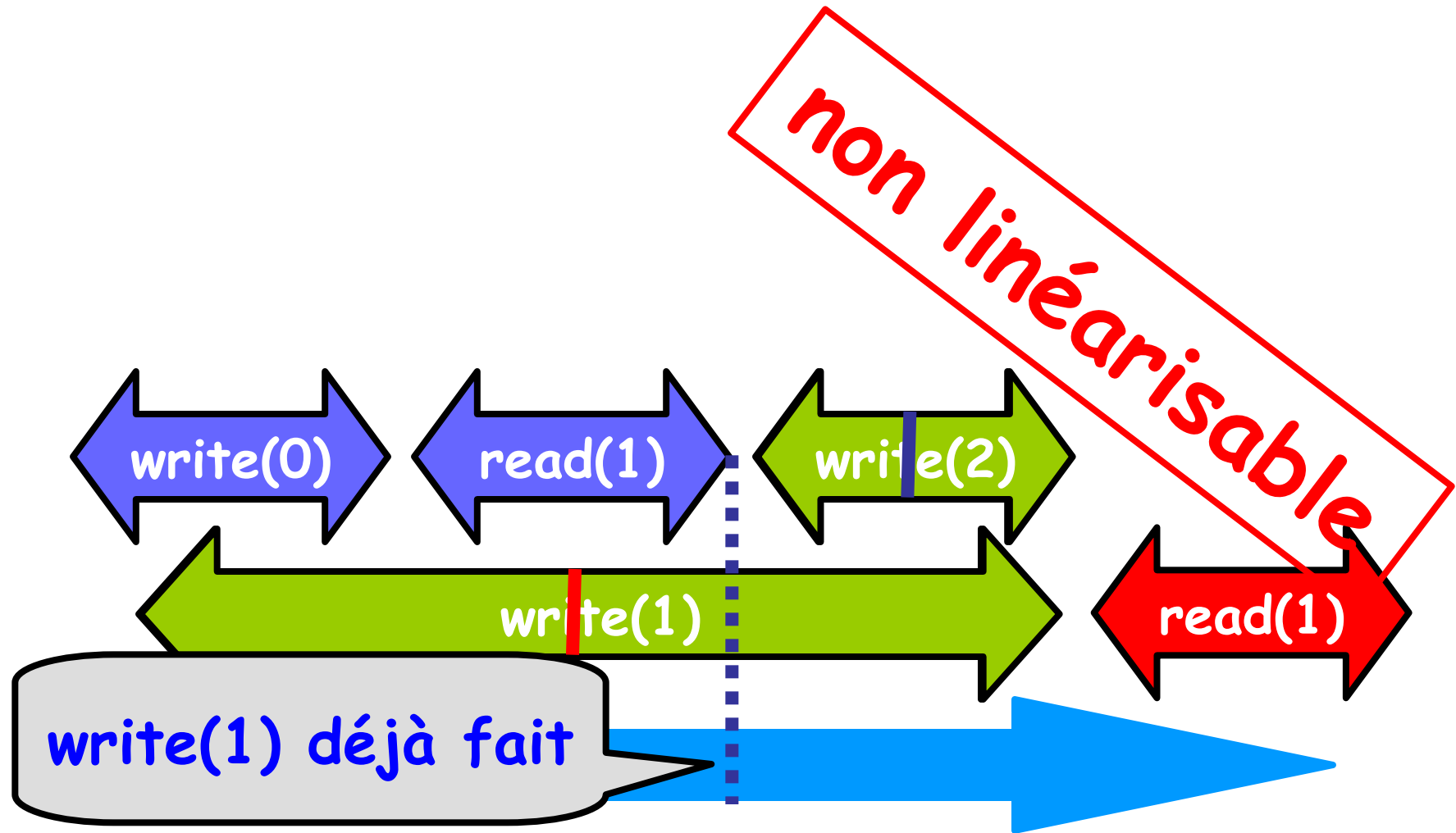
Read/Write Registre Exemple



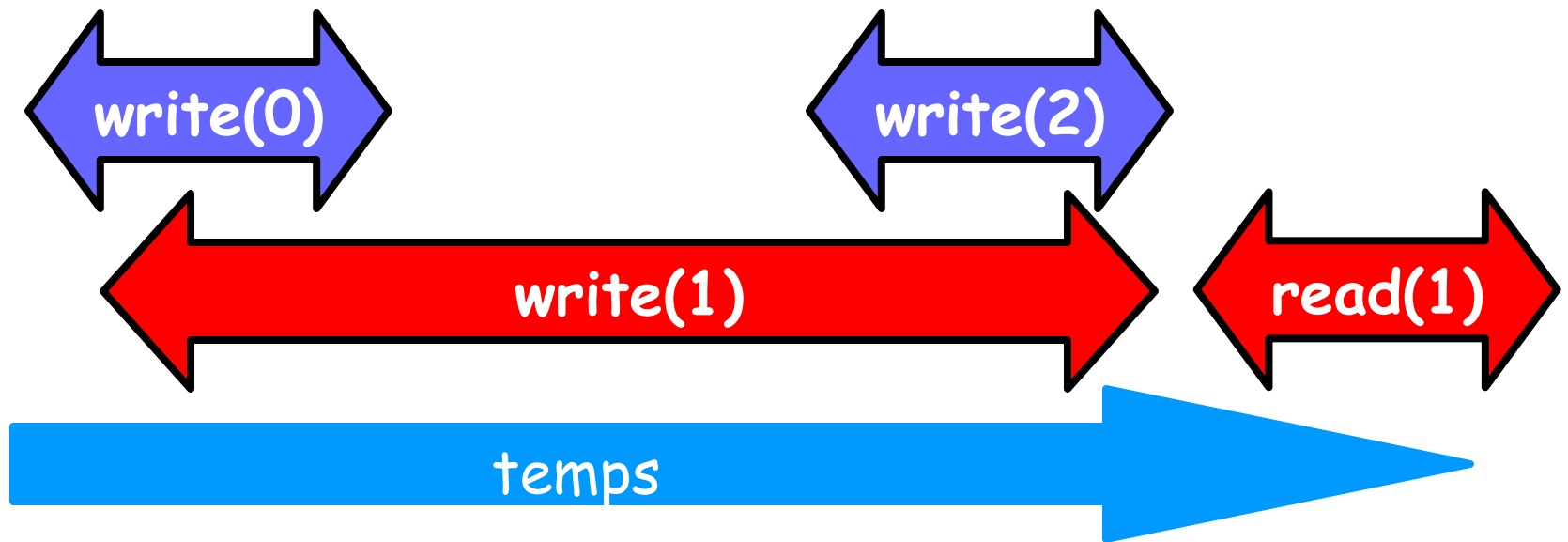
Read/Write Registre Exemple



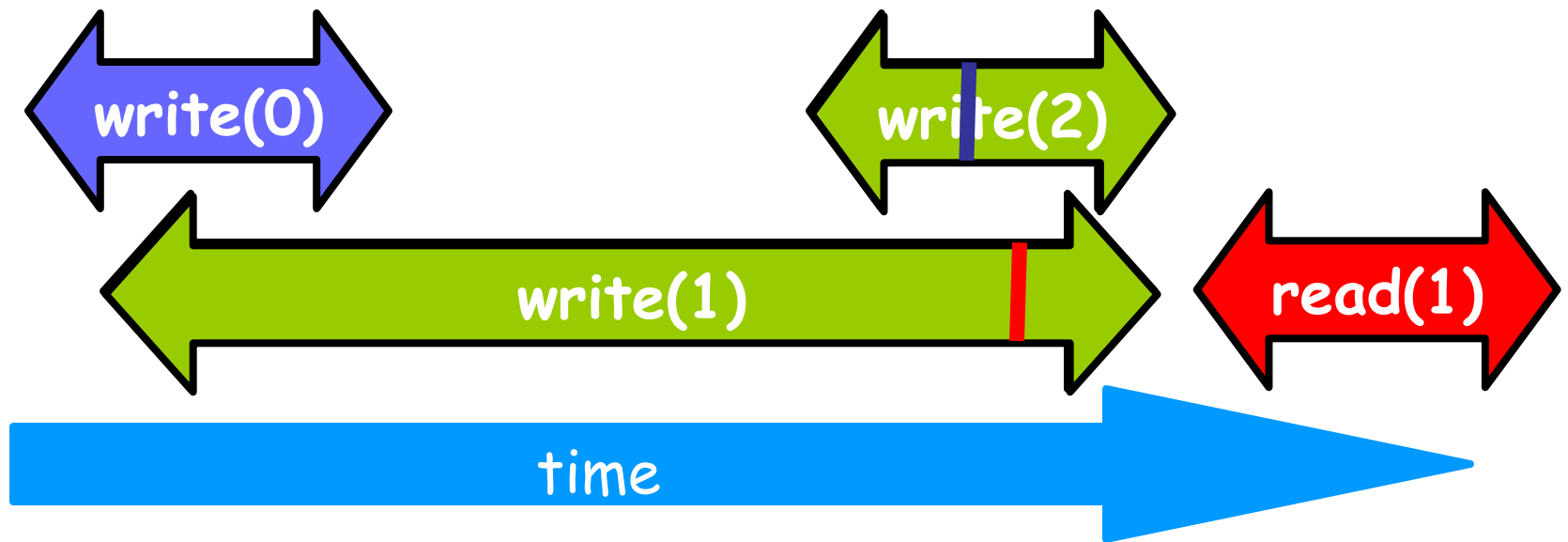
Read/Write Registre Exemple



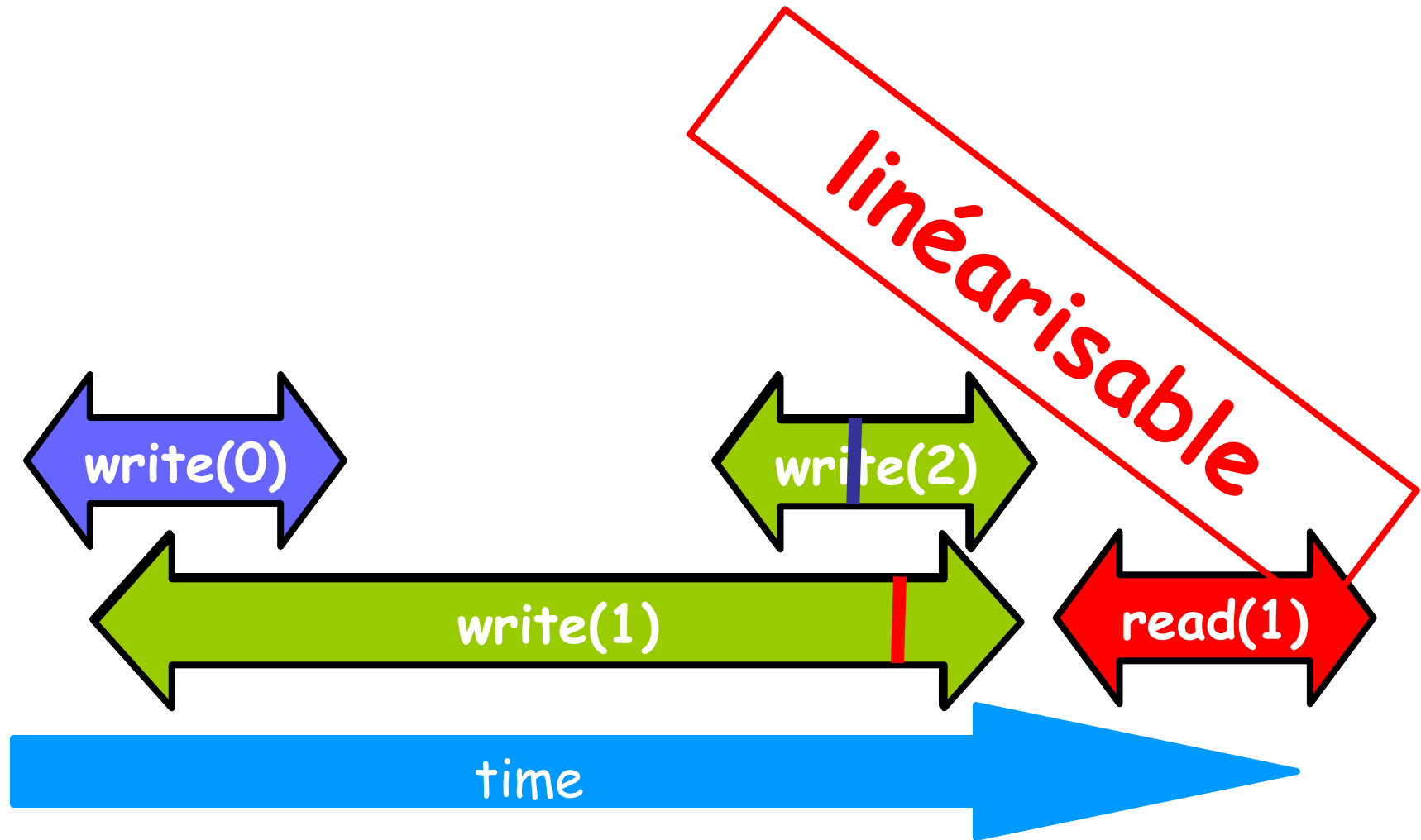
Read/Write Registre Exemple



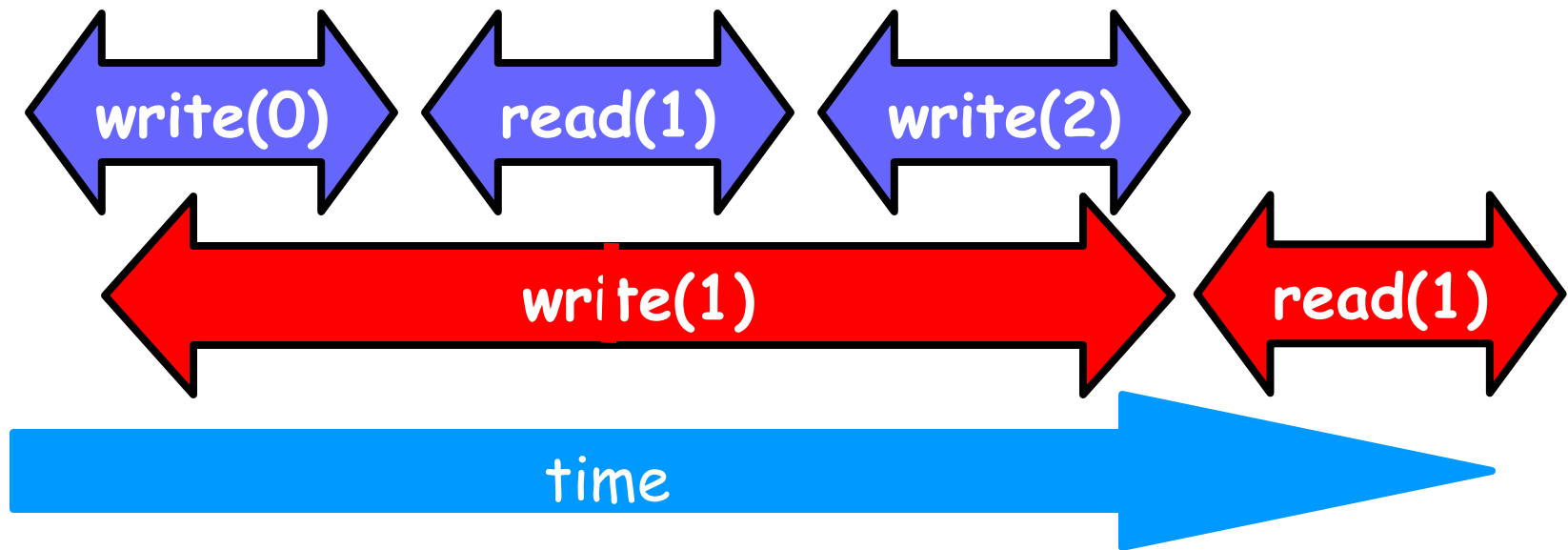
Read/Write Registre Exemple



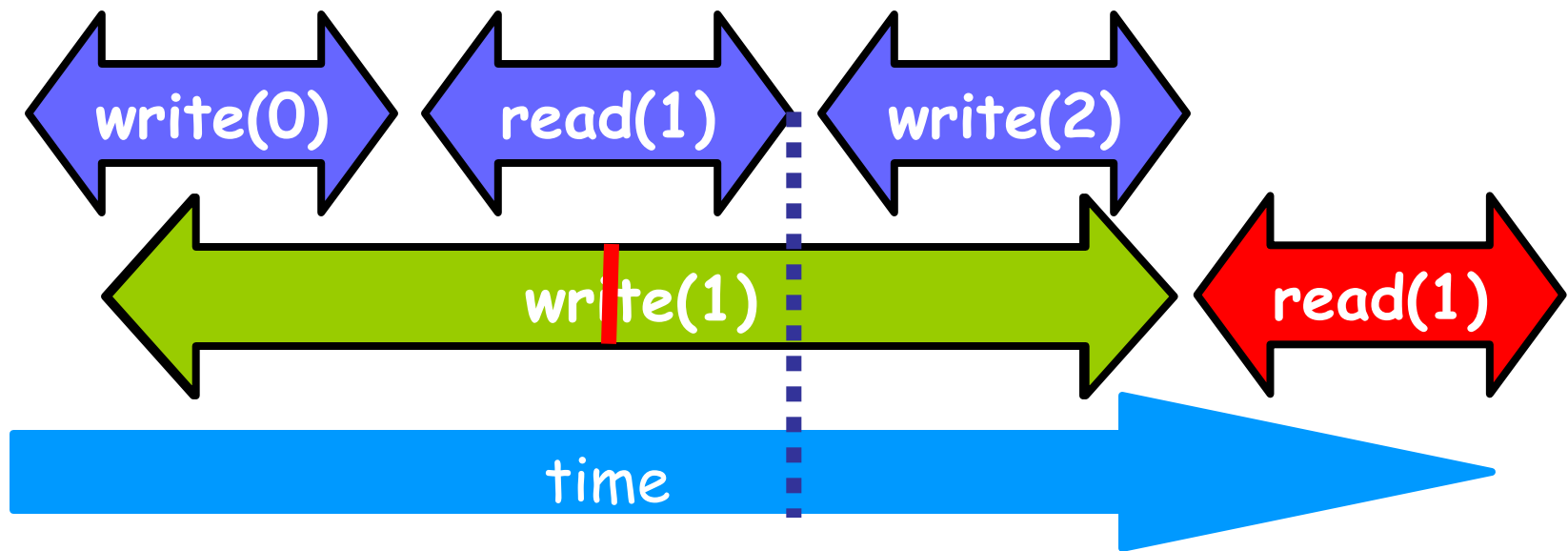
Read/Write Registre Exemple



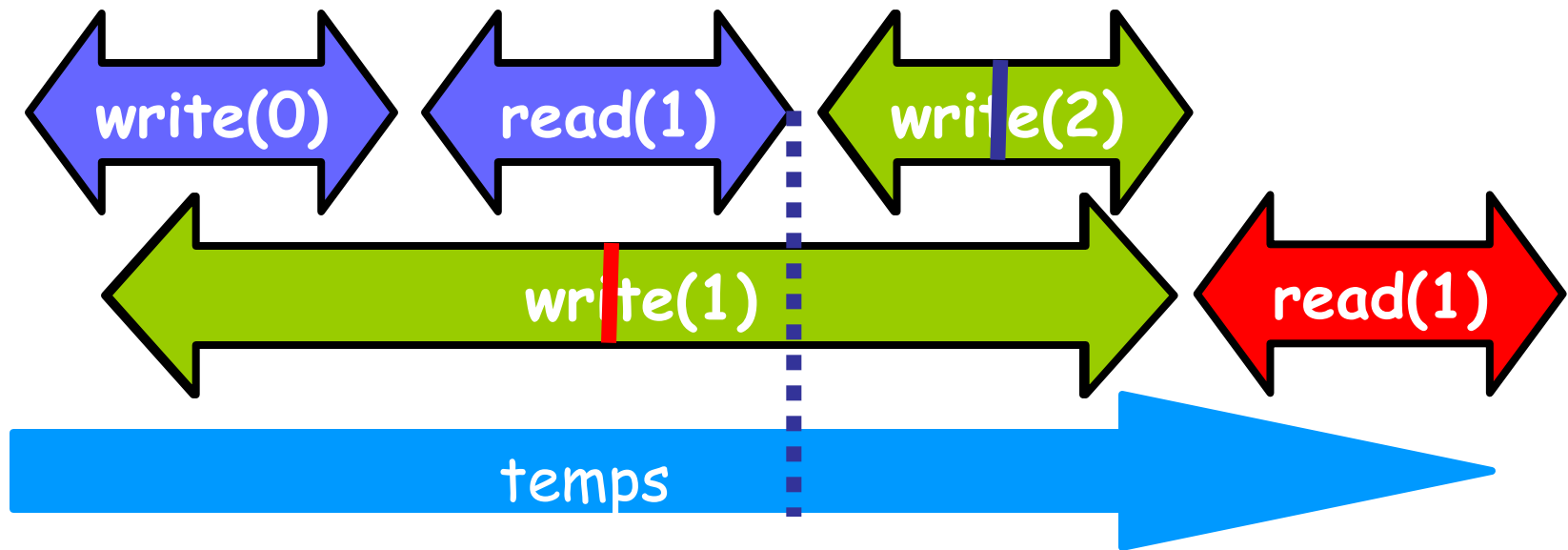
Read/Write Registre Exemple



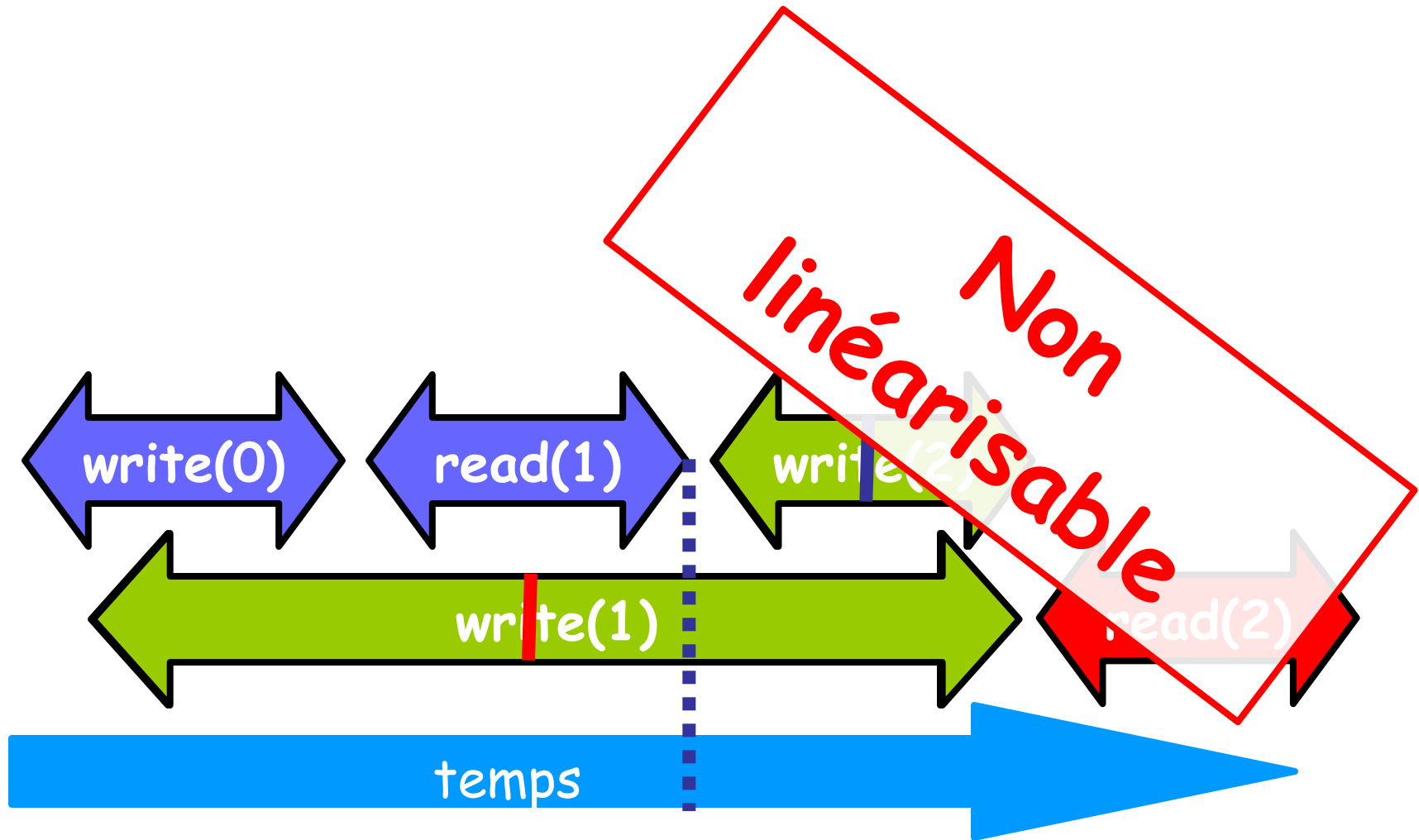
Read/Write Registre Exemple



Read/Write Registre Exemple



Read/Write Register Example



Modèle..

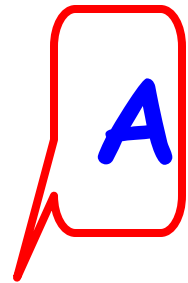
Appel de méthodes: 2 événements

- Invocation
 - Methode name & args
 - q.enq(x)
- Réponse
 - résultat (ou exception)
 - q.enq(x) retourne void
 - q.deq() retourne x
 - q.deq() throws empty

Invocation

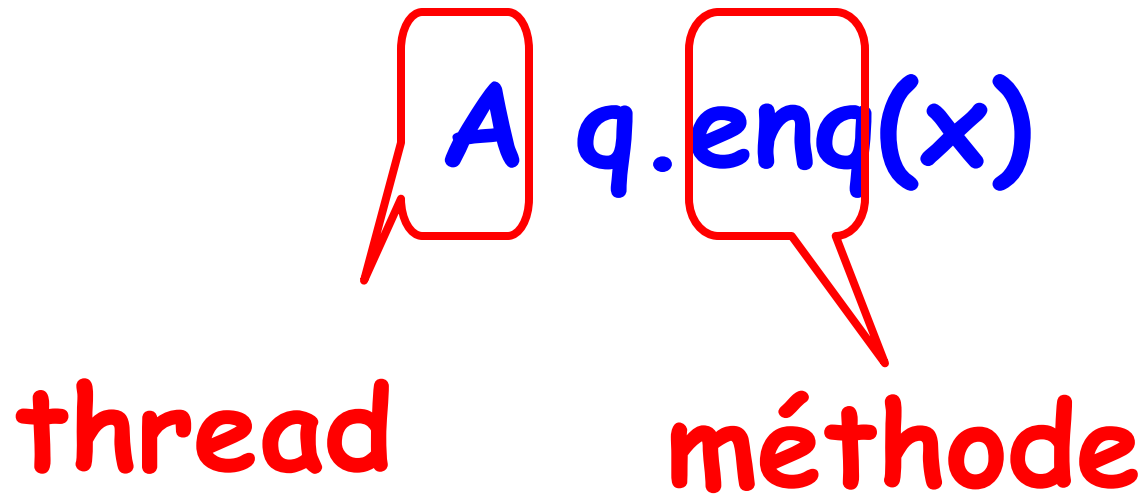
A q.enq(x)

Invocation

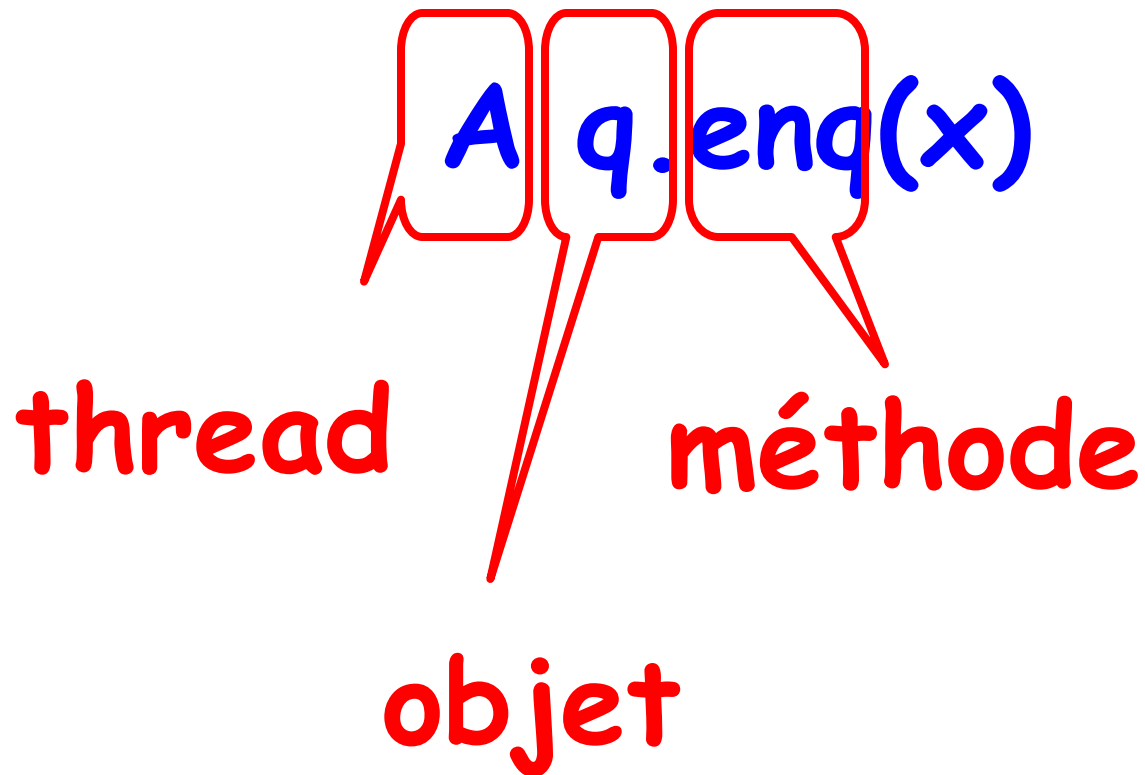
 `q.enq(x)`

thread

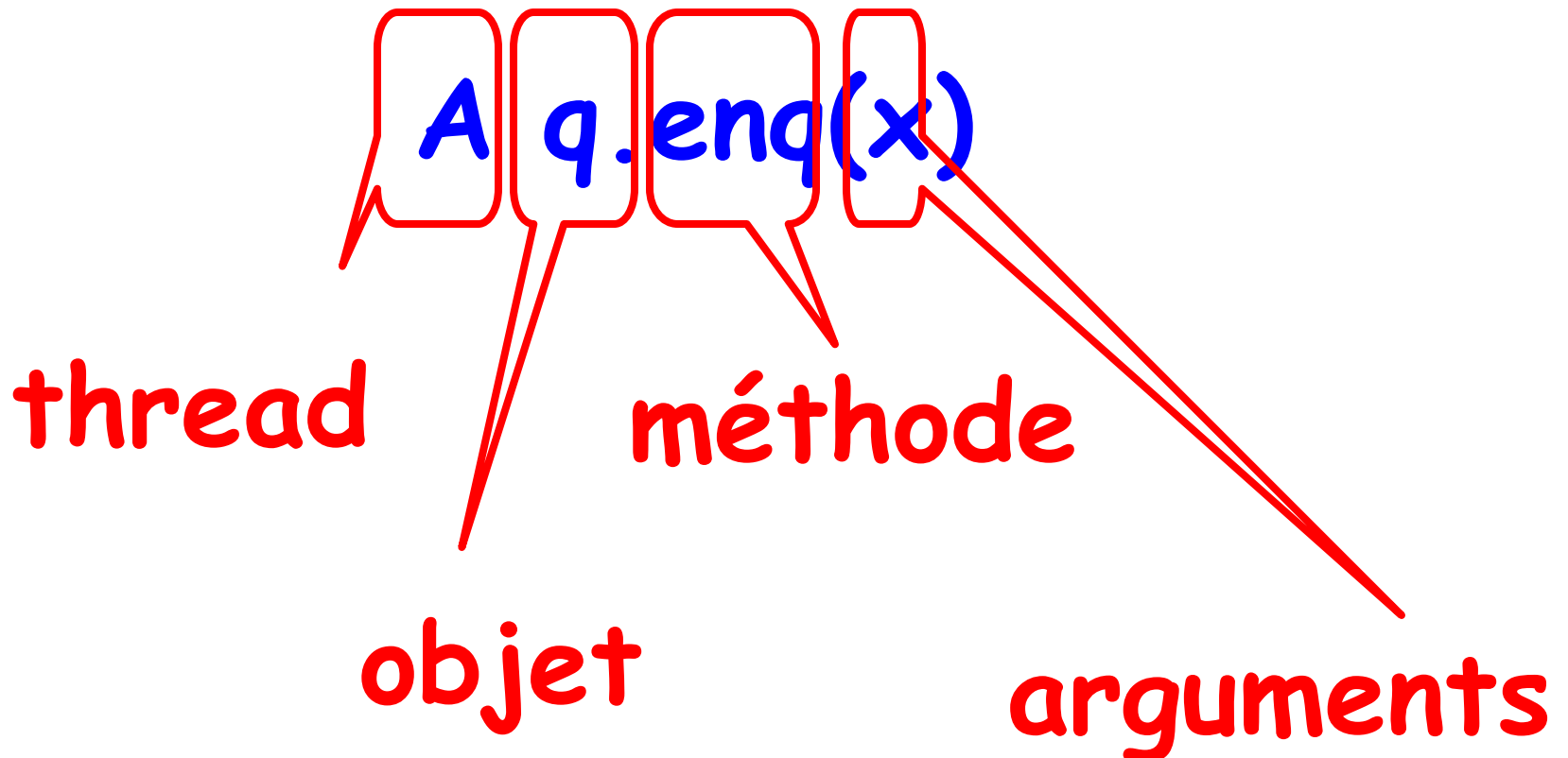
Invocation



Invocation



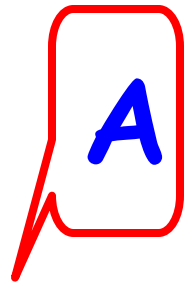
Invocation Notation



Réponse

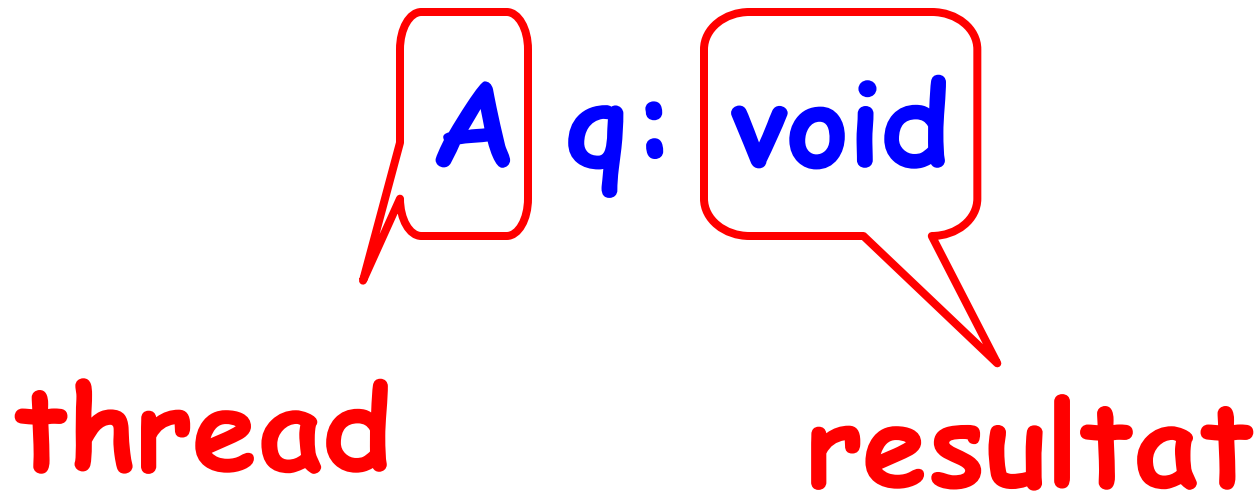
A q: void

Réponse

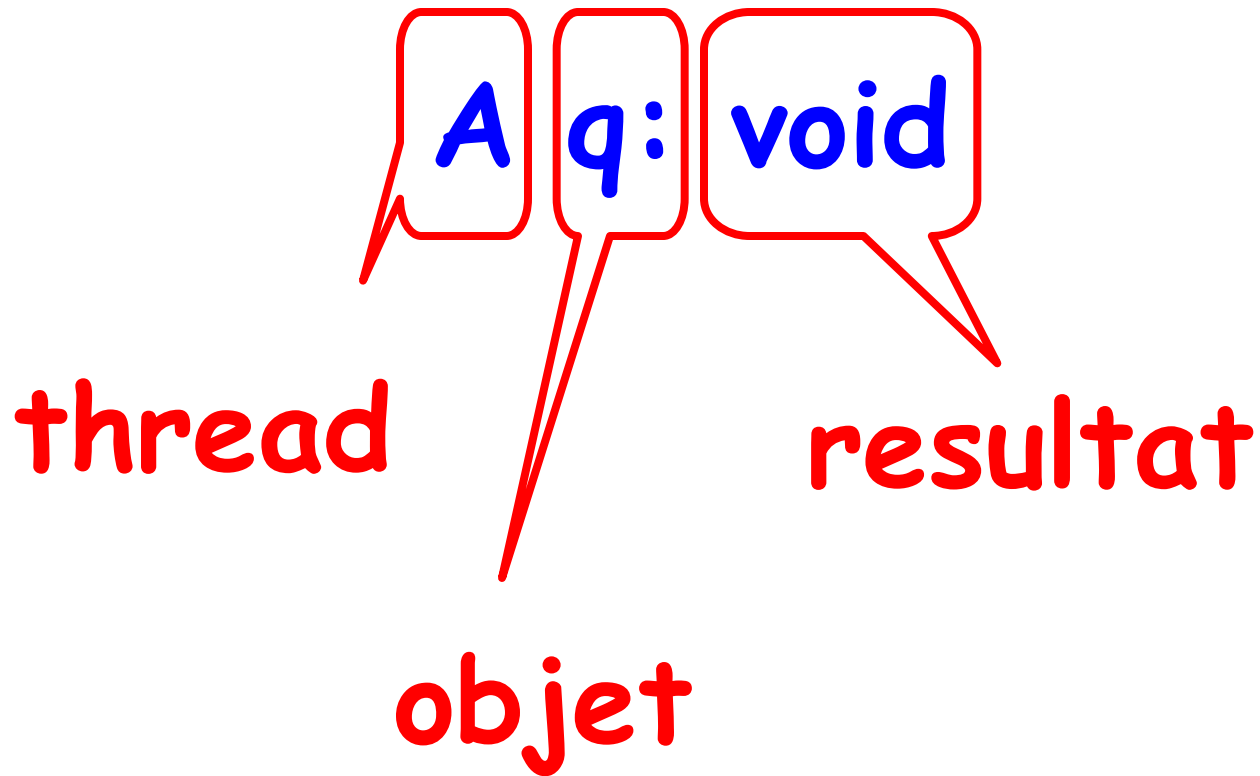
 **A q: void**

thread

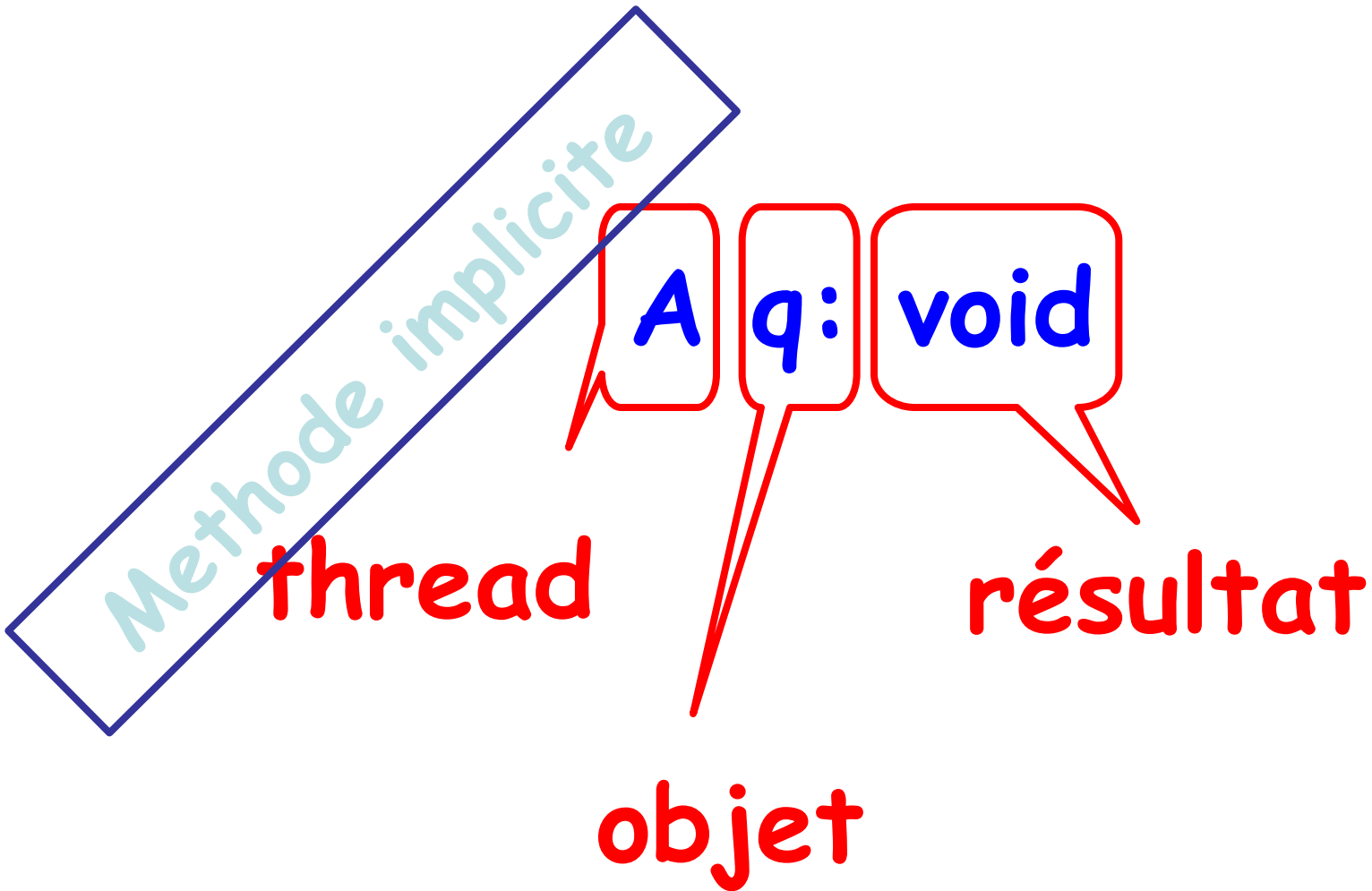
Réponse



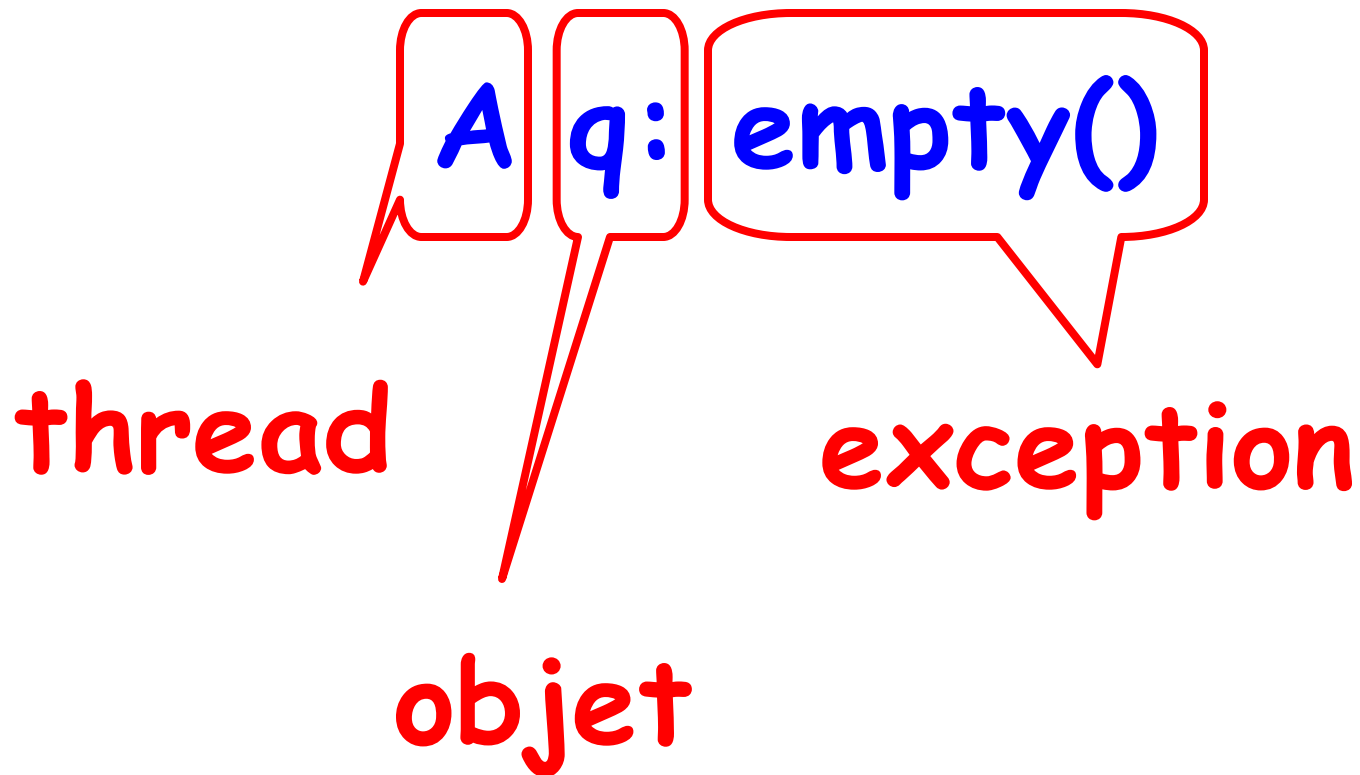
Réponse



Réponse



Réponse Notation



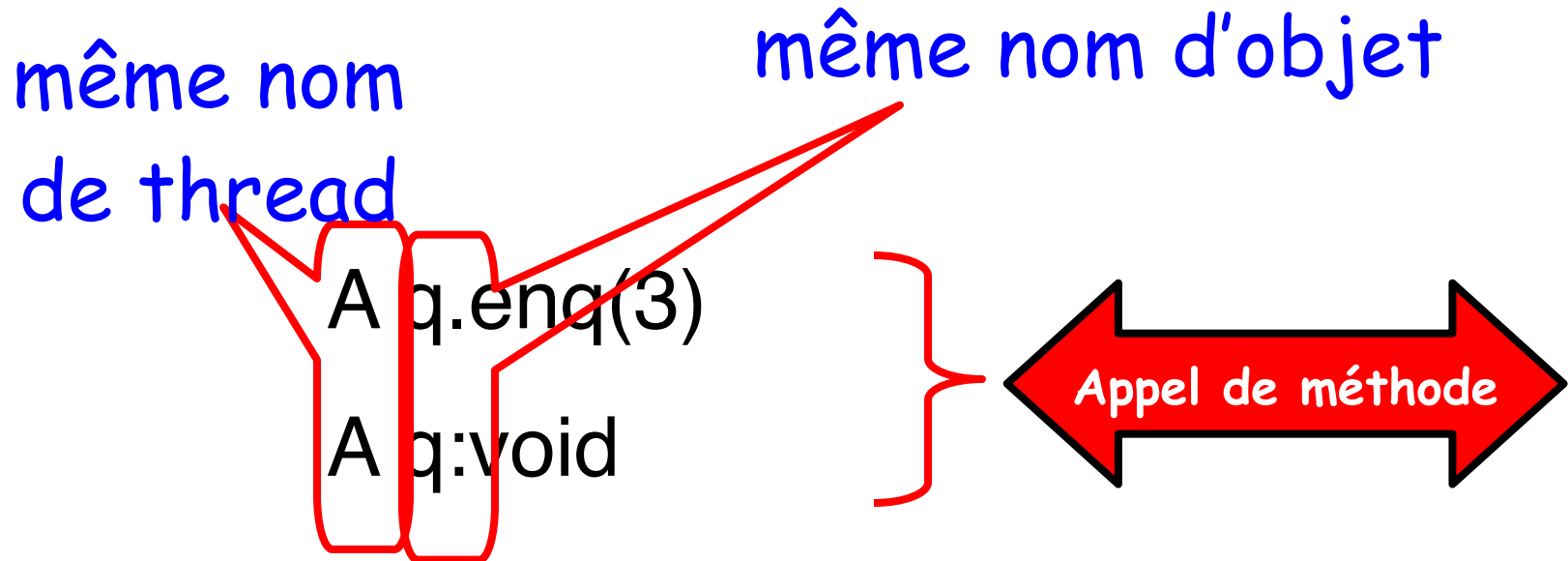
Histoires-

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

**Sequence
d'invocations et de
réponses**

Définition

- Invocation & réponse **se correspondent** si



Projections

$H =$

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

Projections sur les objets

A q.enq(3)

A q:void

$H|q =$

B q.deq()

B q:3

Projections sur les processus

$H =$

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

Projections sur les processus

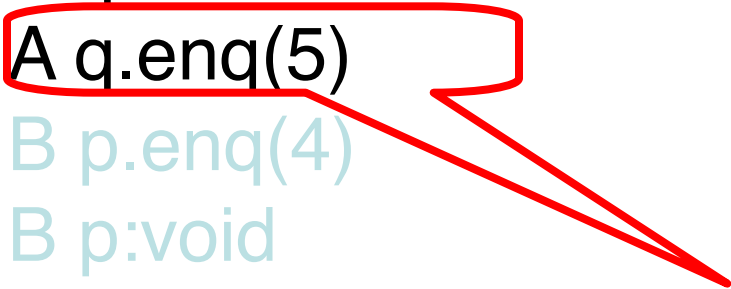
$H|B =$
B p.enq(4)
B p:void
B q.deq()
B q:3

Sous-histoire complète

$H =$

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

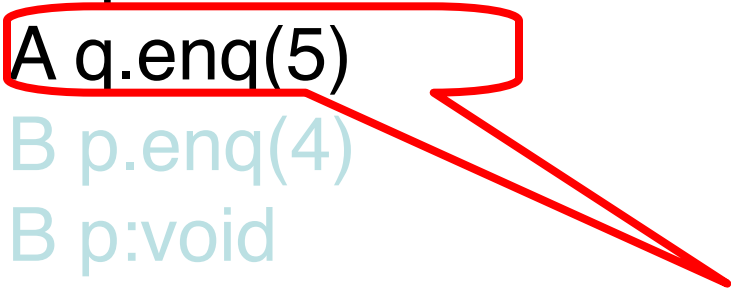
**invocation pendant
pas de réponse
correspondante**



Sous-histoire complète

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

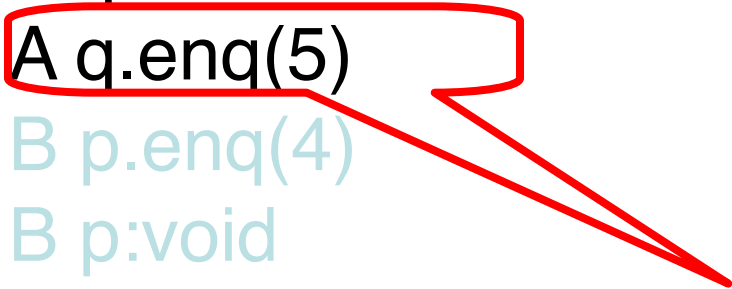
A pris effet ou non



Sous-histoire complète

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

**ignorer les
invocations
pendantes**



Sous-histoire complète

A q.enq(3)
A q:void

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

Sous-histoire complète

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

Histoires séquentielles

A q.enq(3)

A q:void

B p.enq(4)

B p:void

B q.deq()

B q:3

A q:enq(5)

Histoires séquentielles

A q.enq(3)

A q:void

B p.enq(4)

B p:void

B q.deq()

B q:3

A q:enq(5)

Correspondance

Histoires séquentielles

A q.enq(3)
A q:void

Correspondance

B p.enq(4)
B p:void

Correspondance

B q.deq()

B q:3

A q:enq(5)

Histoires séquentielles

A q.enq(3)
A q:void

Correspondance

B p.enq(4)
B p:void

Correspondance

B q.deq()
B q:3

Correspondance

A q:enq(5)

Histoires séquentielles

A q.enq(3)
A q:void

Correspondance

B p.enq(4)
B p:void

Correspondance

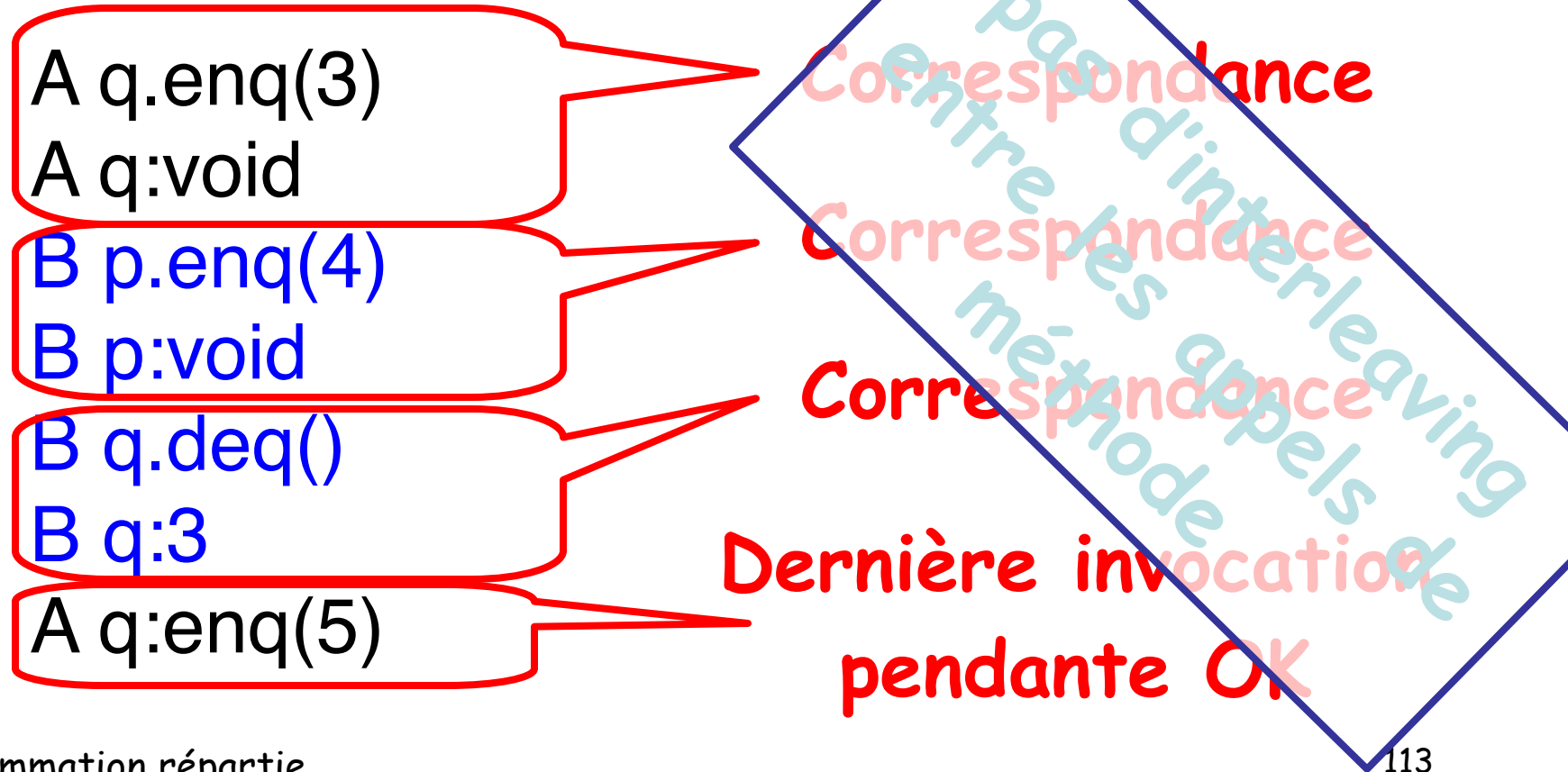
B q.deq()
B q:3

Correspondance

A q:enq(5)

**Dernière invocation
pendante OK**

Histoires séquentielles



Histoires bien formées

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

Histoires bien formées

projections par
processus séquentiels

HIB=

H=

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

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

Histoires bien formées

projections par
processus séquentiels

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

Histoires équivalentes

les processus voient la même chose

$\left\{ \begin{array}{l} HIA = GIA \\ HIB = GIB \end{array} \right.$

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
```

Histoires légales

- Une histoire séquentielle H est légale if
 - Pour tout objet x
 - $H|x$ est légale (conforme) pour la spécification séquentielle de x

Précédence

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

Un appel de méthode en
précède un autre si la
réponse du premier
précède l'invocation de
l'autre



Non-Précédence

A q.enq(3)

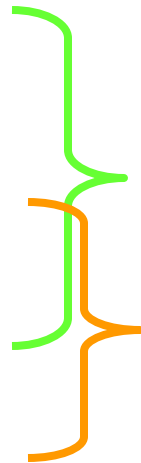
B p.enq(4)

B p.void

B q.deq()

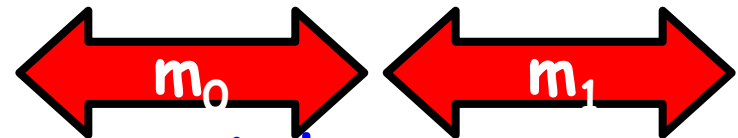
A q:void

B q:3



Notation

- Soit
 - Histoire H
 - méthodes m_0 et m_1 dans H
- $m_0 \rightarrow_H m_1$, si
 - m_0 précède m_1
- $m_0 \rightarrow_H m_1$ est un ordre partiel et un ordre total si H est séquentiel



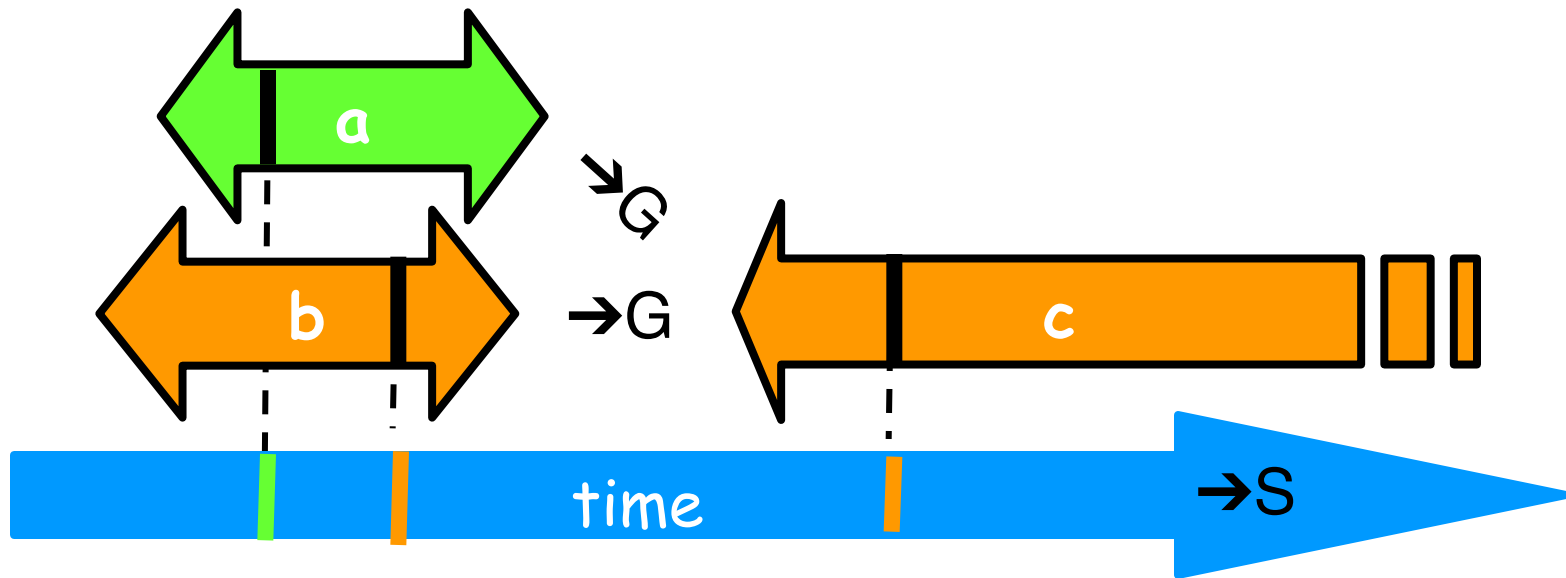
Linéarisabilité

- L'histoire H est **linéarisable** si elle peut être étendue à G en
 - Ajoutant des réponses à des invocations pendantes
 - Ignorant les autres invocations pendantes
- De façon à ce que G soit équivalente à
 - Une histoire séquentielle légale S
 - telle que $\rightarrow_G \subset \rightarrow_S$

$$\rightarrow_G \subset \rightarrow_S$$

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

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



Remarques

- Pour les invocations pendantes:
 - on garde celles qui ont pris effet
 - on élimine les autres
- La condition $\rightarrow_G \subset \rightarrow_S$
 - signifie que **S** respecte l'ordre “temps réel” de **G**

Exemple

A q.enq(3)

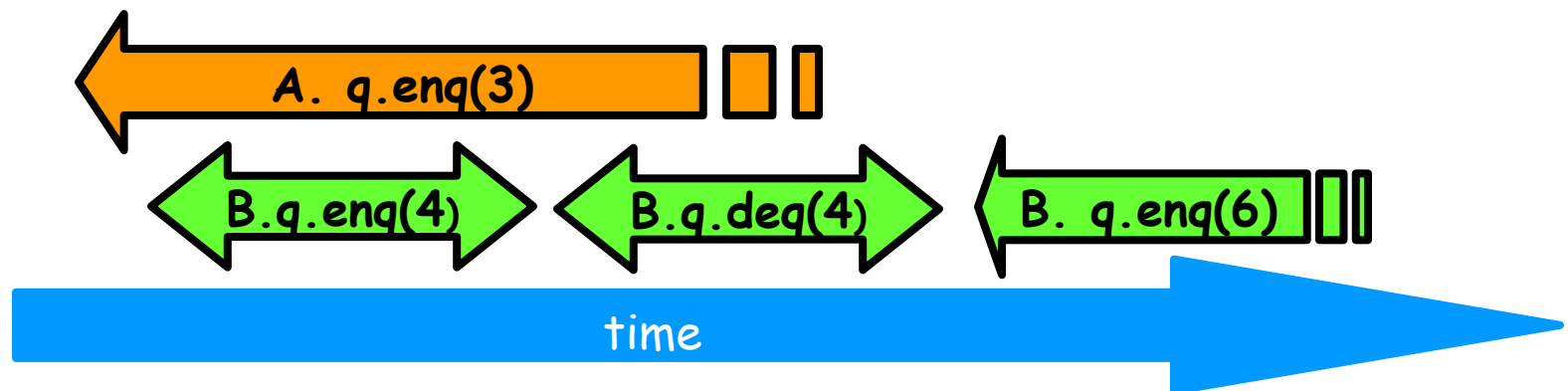
B q.enq(4)

B q:void

B q.deq()

B q:4

B q:enq(6)



Exemple

A q.enq(3)

B q.enq(4)

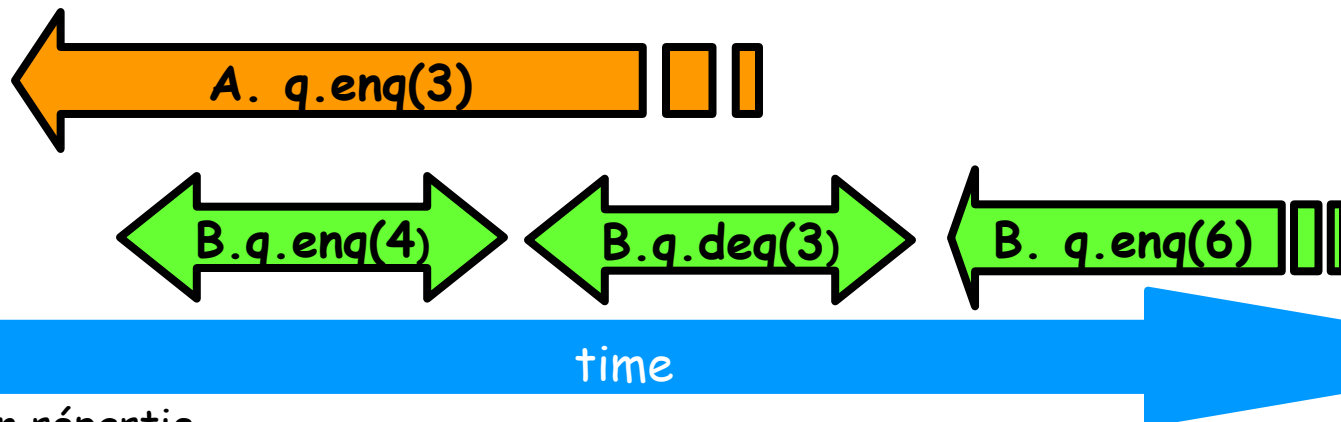
B q:void

B q.deq()

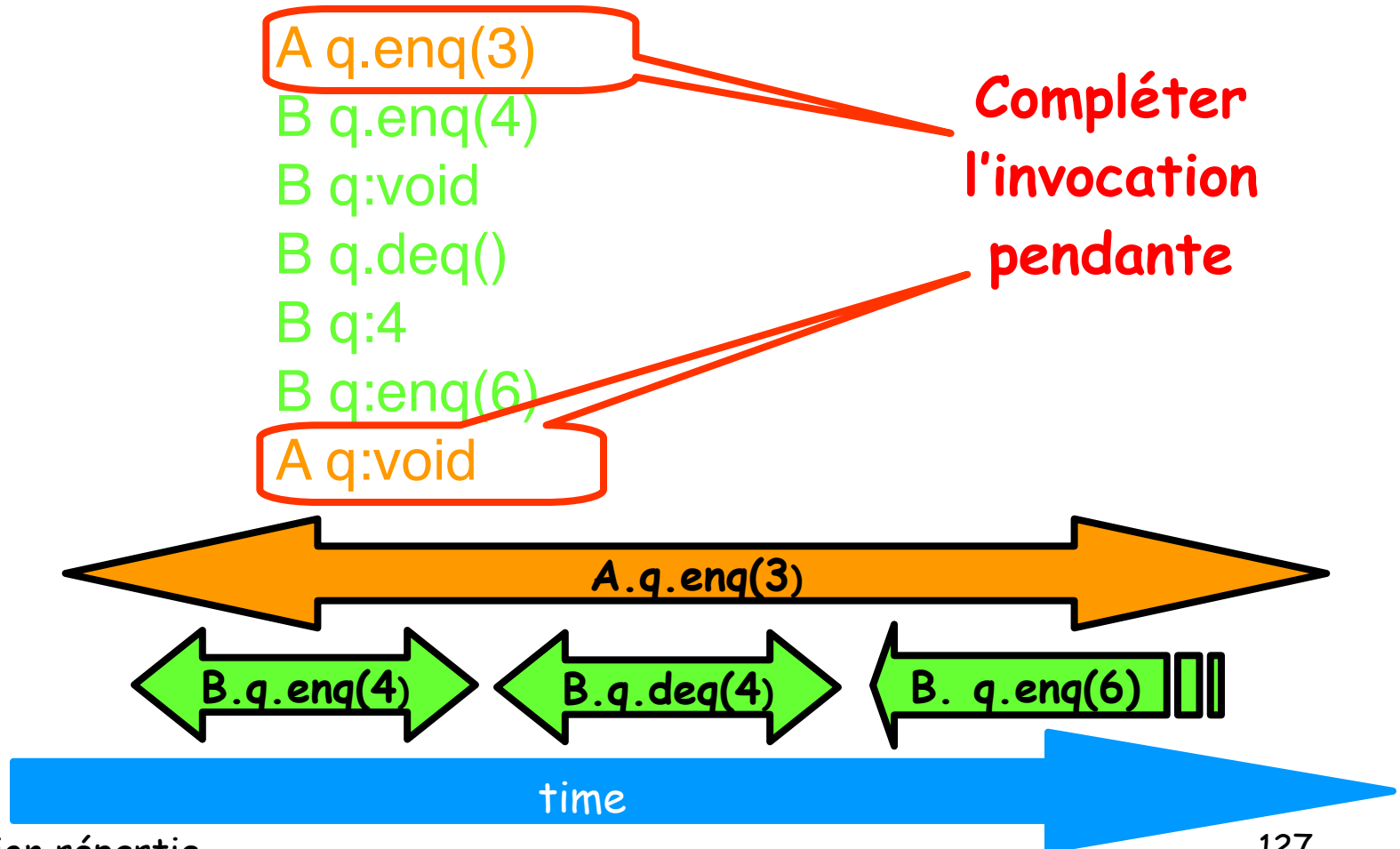
B q:4

B q:enq(6)

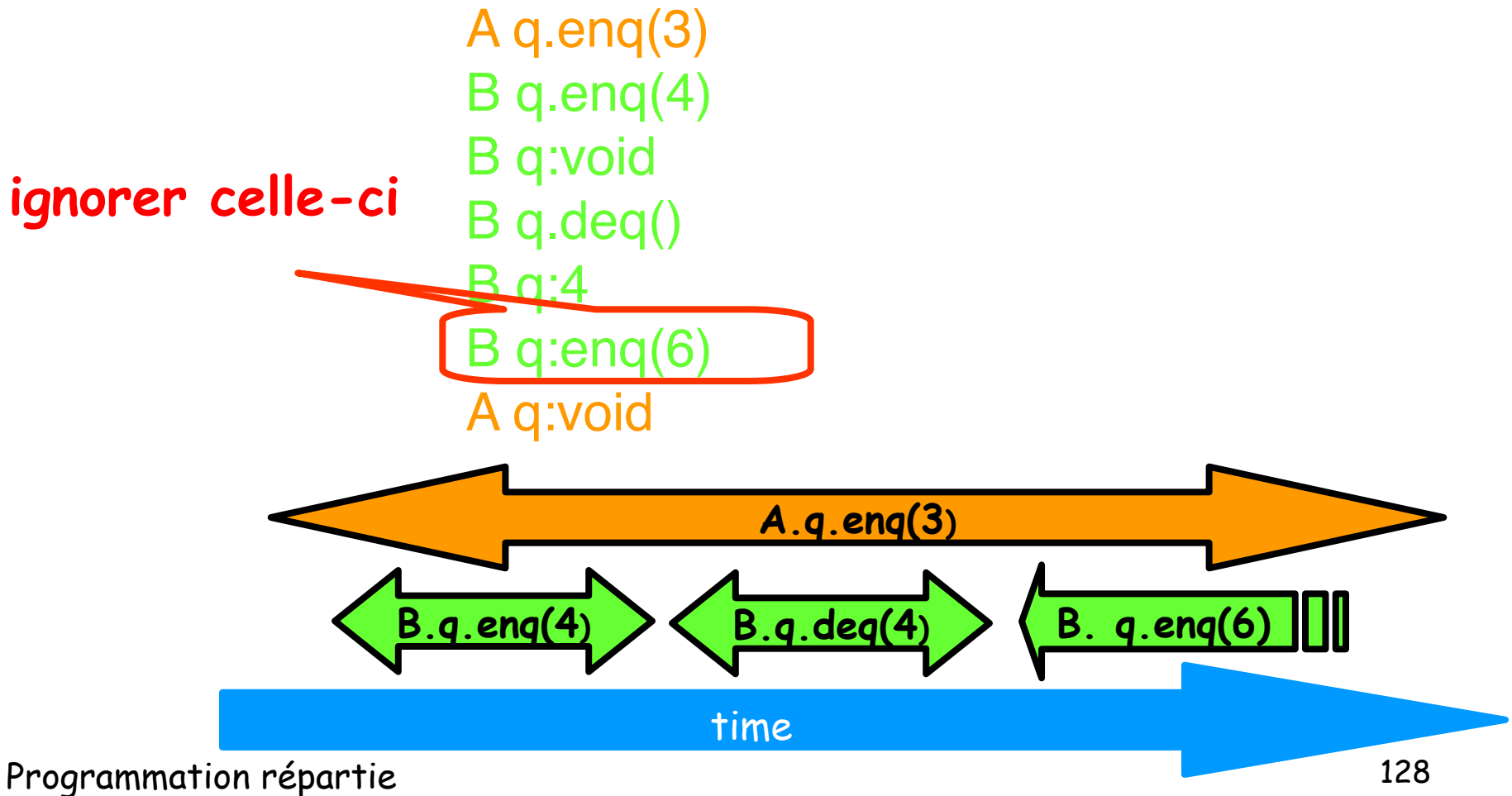
Compléter
l'invocation
pendante



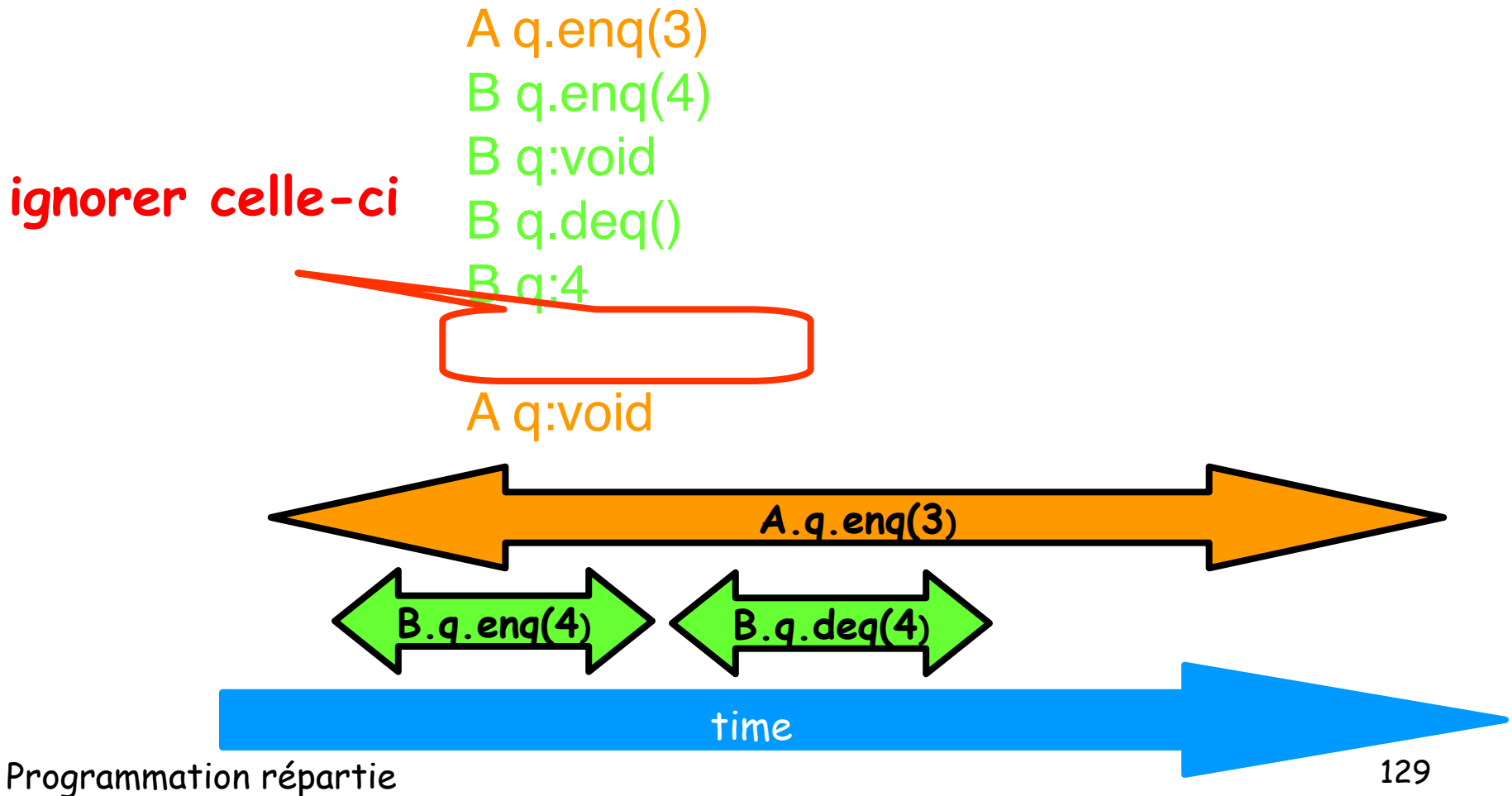
Exemple



Exemple



Exemple



Example

A q.enq(3)

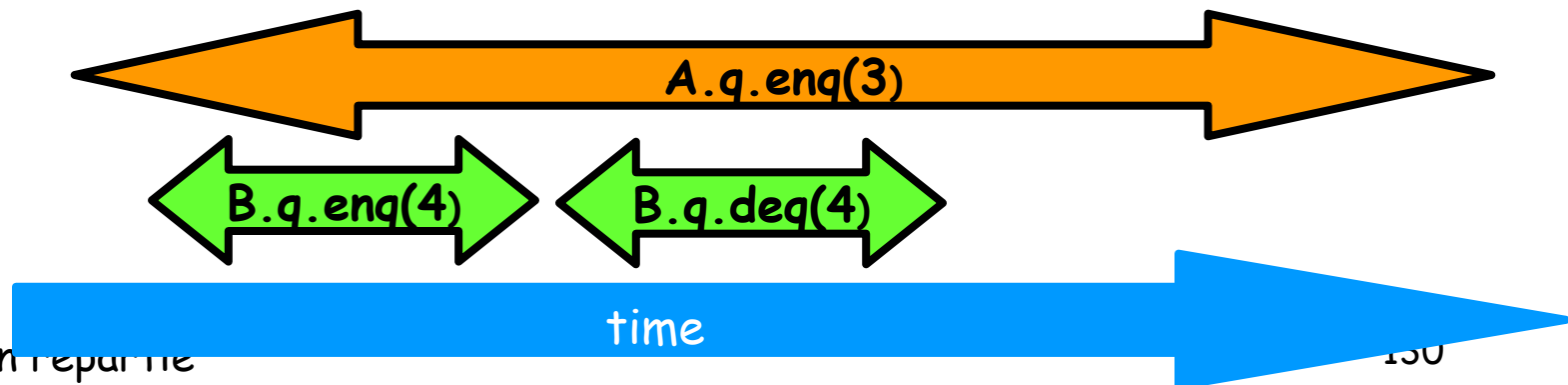
B q.enq(4)

B q:void

B q.deq()

B q:4

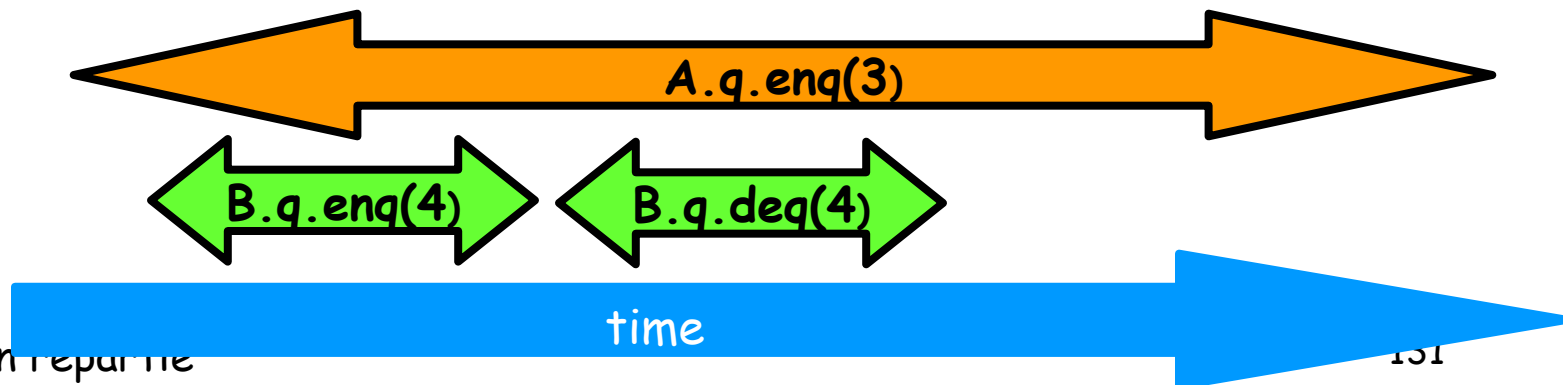
A q:void



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

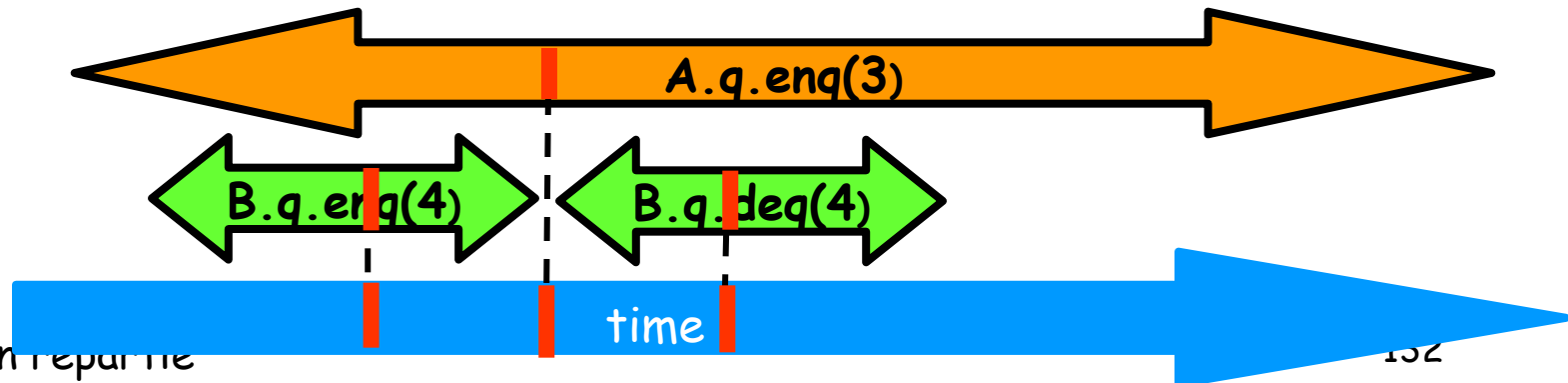


Exemple

Histoire séquentielle équivalente

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



Concurrence

- Quelle concurrence la linearizabilité permet-elle?
- Quand l'invocation d'une méthode doit-elle bloquer?

Concurrence

- On s'intéresse aux méthodes **totales**
 - Défini dans chaque état
- Exemple:
 - `deq()` qui lance `Empty exception`
 - Version `deq()` qui attends...
- Pourquoi?
 - Sinon blocage sans liaison avec la synchronisation

Concurrency

- **Question:** Quand la linéarization demande t elle à une invocation de méthode de bloquer?
- **Réponse:** jamais.
- Linearizabilité est non-bloquante

Théorem de non-blocage

Si invocation

$A \ q.inv(...)$

est pendante dans l'histoire H , alors il
existe une réponse

$A \ q.res(...)$

tel que

$H + A \ q.res(...)$

Est linéarisable

Preuve

- Soit S une linéarisation de H
- Si S contient déjà
 - Invocation $A\ q.inv(\dots)$ et sa réponse ,
 - C'est ok.
- Sinon, prendre une réponse tel que
 - $S + A\ q.inv(\dots) + A\ q.res(\dots)$
 - c'est possible car l'objet est **total**.

Théorème de composition (localité)

- Une histoire H est linéarisable

si et seulement si:

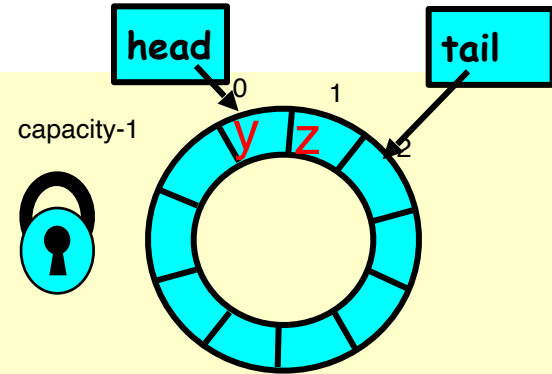
- $H|x$ est linéarisable pour tout objet x

Compositionnalité?

- Modularité
- On peut prouver la linéarisabilité des objets en isolation
- On peut composer des objets

Linéarisabilité: Locking

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



Linéarisabilité: Locking

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

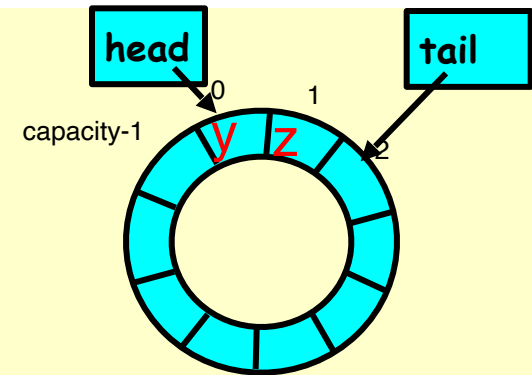
Points de linéarisation
quand le lock est
relâché

Implementation: **locking**

```
public void enq(Item x) throws FullException {  
    lock.lock();  
    try {  
        if (tail - head == capacity )  
            throw new FullException();  
        items[tail % capacity] = x; tail++;  
    } finally {  
        lock.unlock();  
    }  
}
```

Linéarisabilité: Lockfree (2 threads une met dans la file, l'autre enleve)

```
public class LockFreeQueue {  
  
    volatile int head = 0, tail = 0;  
    volatile items = (T[]) new Object[capacity];  
  
    public void enq(Item x) {  
        while (tail-head == capacity); // busy-wait  
        items[tail % capacity] = x; tail++;  
    }  
  
    public Item deq() {  
        while (tail == head); // busy-wait  
        Item item = items[head % capacity]; head++;  
        return item;  
    }  
}
```



Mor... reasoning

attention il n'y a qu'un thread qui enfile et qu'un thread qui défile

Linéarisation: ordre des modifications de head et tail

```
public class L...  
  
volatile int head, tail = 0;  
volatile Object[] items = new Ob...  
  
public void enq(Item x) {  
    while (tail-head == capacity); // busy-wait  
    items[tail % capacity] = x; tail++;  
}  
  
public Item deq() {  
    while (tail == head); // busy-wait  
    Item item = items[head % capacity]; head++;  
    return item;  
}}
```


Alternative: Consistence séquentielle

- L'histoire H est **Consistente séquentiellement** si elle peut être étendue en G en
 - Ajoutant des réponses aux invocations pendantes
 - en ignorant les autre invocations pendantes
- De façon à ce que G soit équivalente à une histoire séquentielle S avec $G|p=S|p$ pour toutes les threads

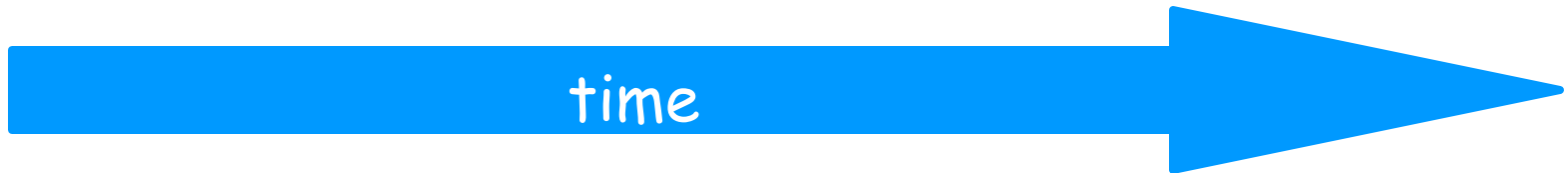
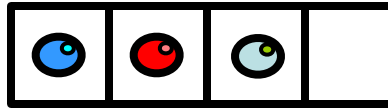
- Avec $\rightarrow G \subset \rightarrow S$

≠ linéarisabilité

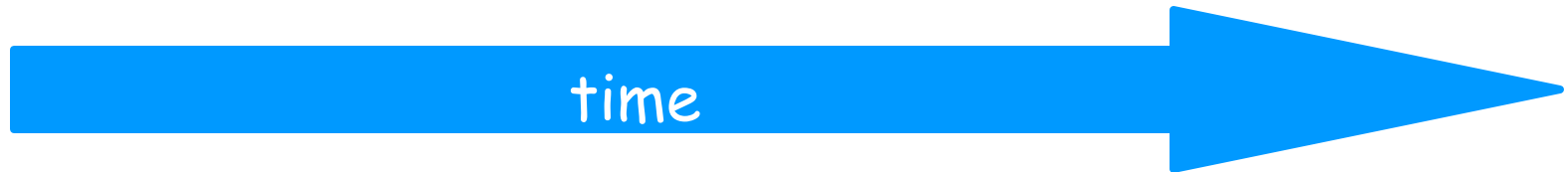
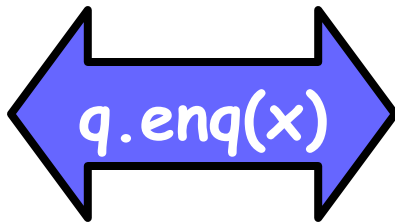
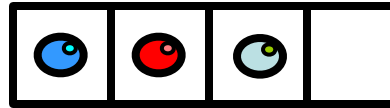
Alternative: Consistence séquentielle

- Pas de préservation du temps réel :
 - On ne peut pas réordonner les opérations faites sur le même thread
 - On peut réordonner des opérations sur des threads différents
- (Utilisé souvent sur des architectures multi-processeurs)

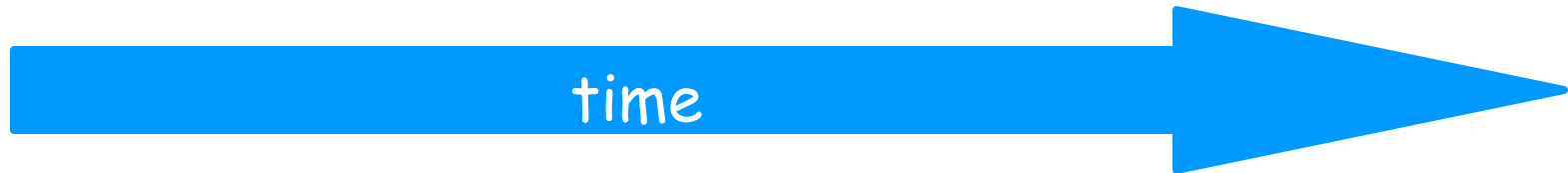
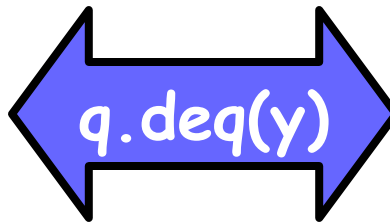
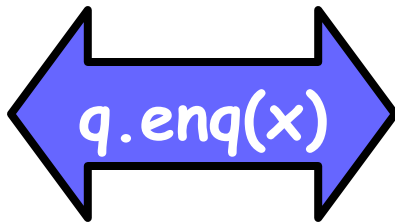
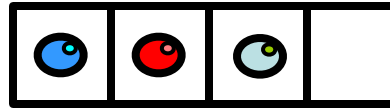
Exemple



Exemple

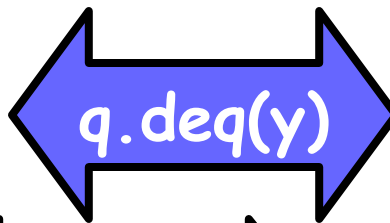
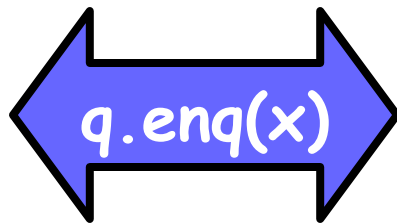
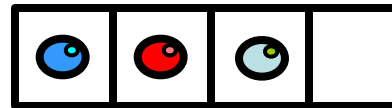


Exemple



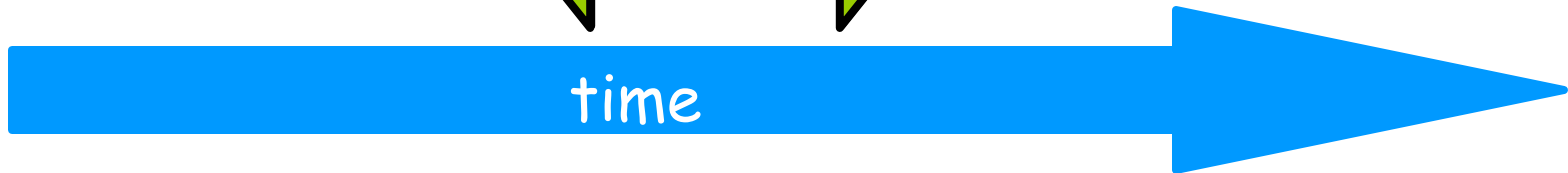
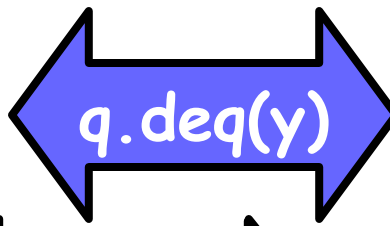
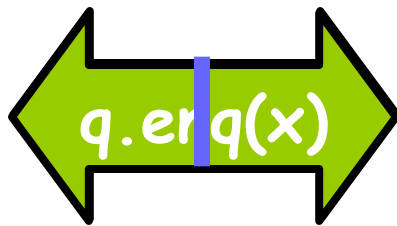
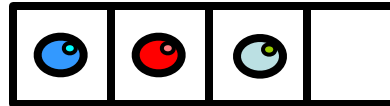


Exemple





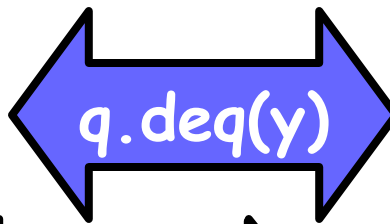
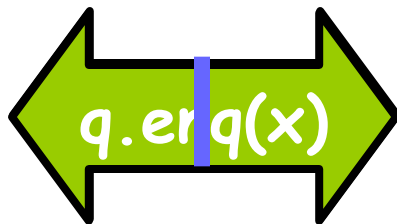
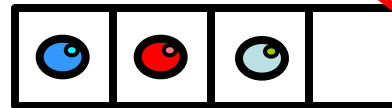
Exemple





Exemple

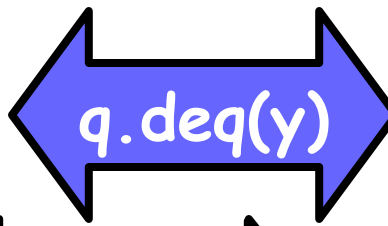
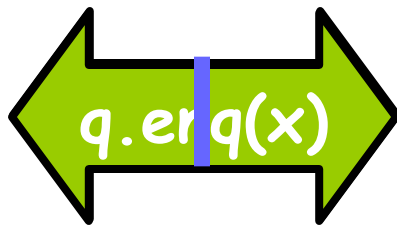
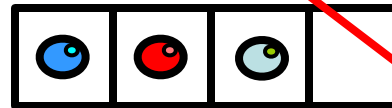
non linéarisable





Exemple

**Mais consistent
séquentiellement**

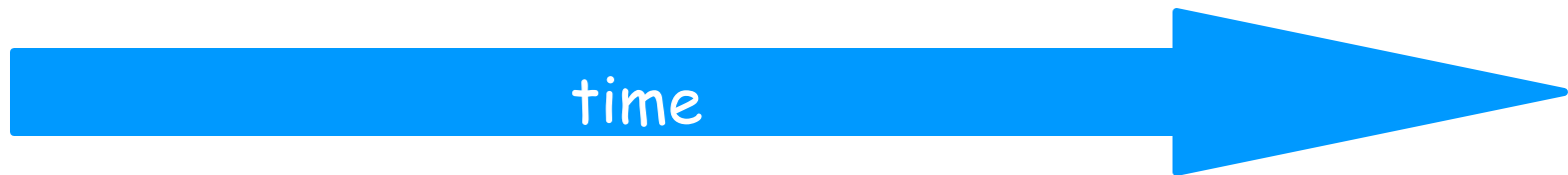
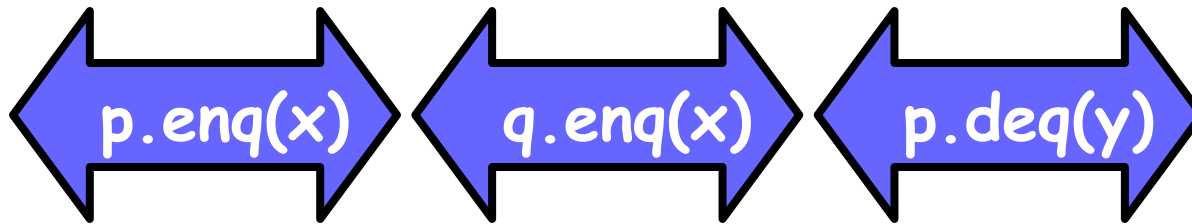


Théorème

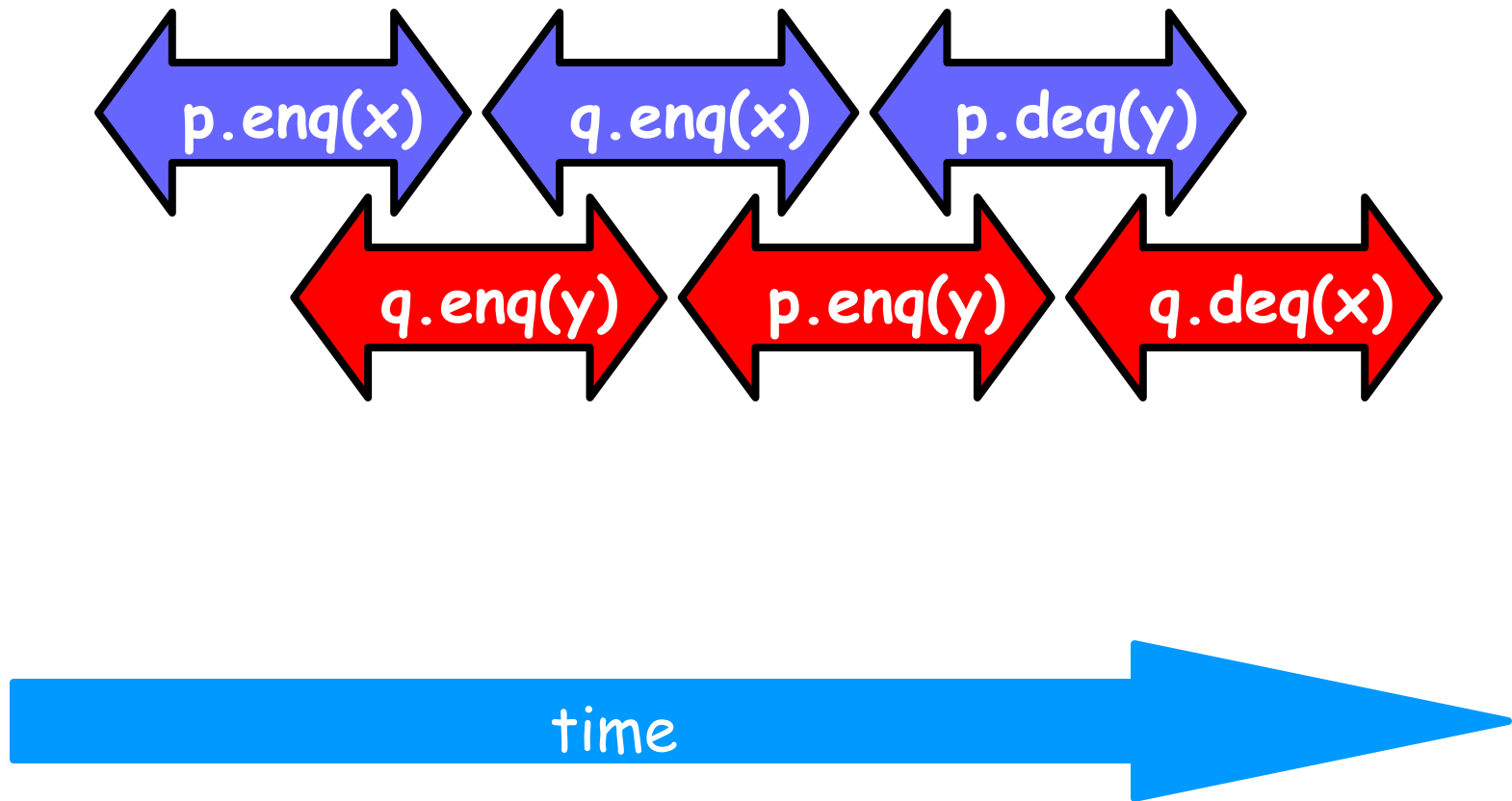
La consistance séquentielle n'est
pas une propriété locale

(pas de compositionnalité...)

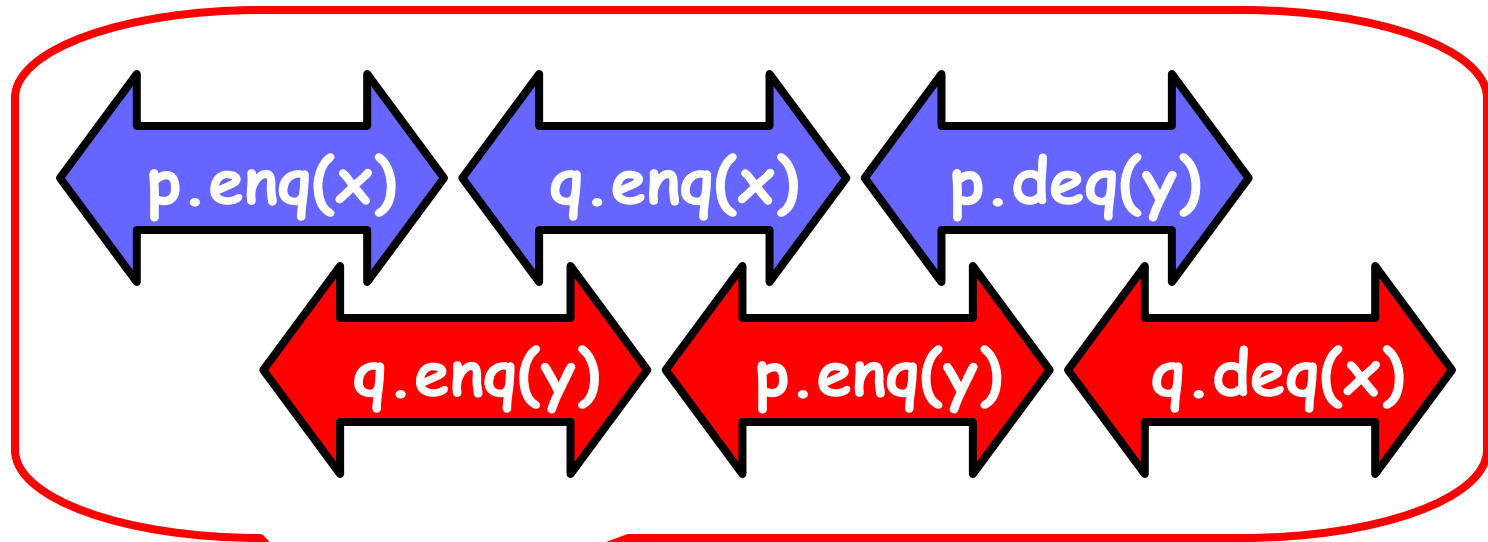
FIFO Queue Example



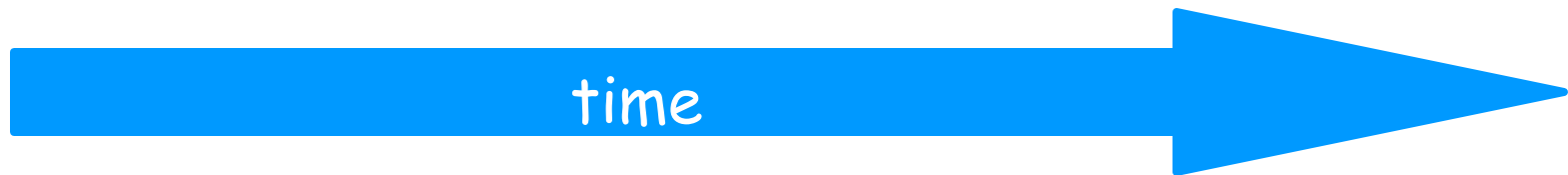
FIFO Queue Example



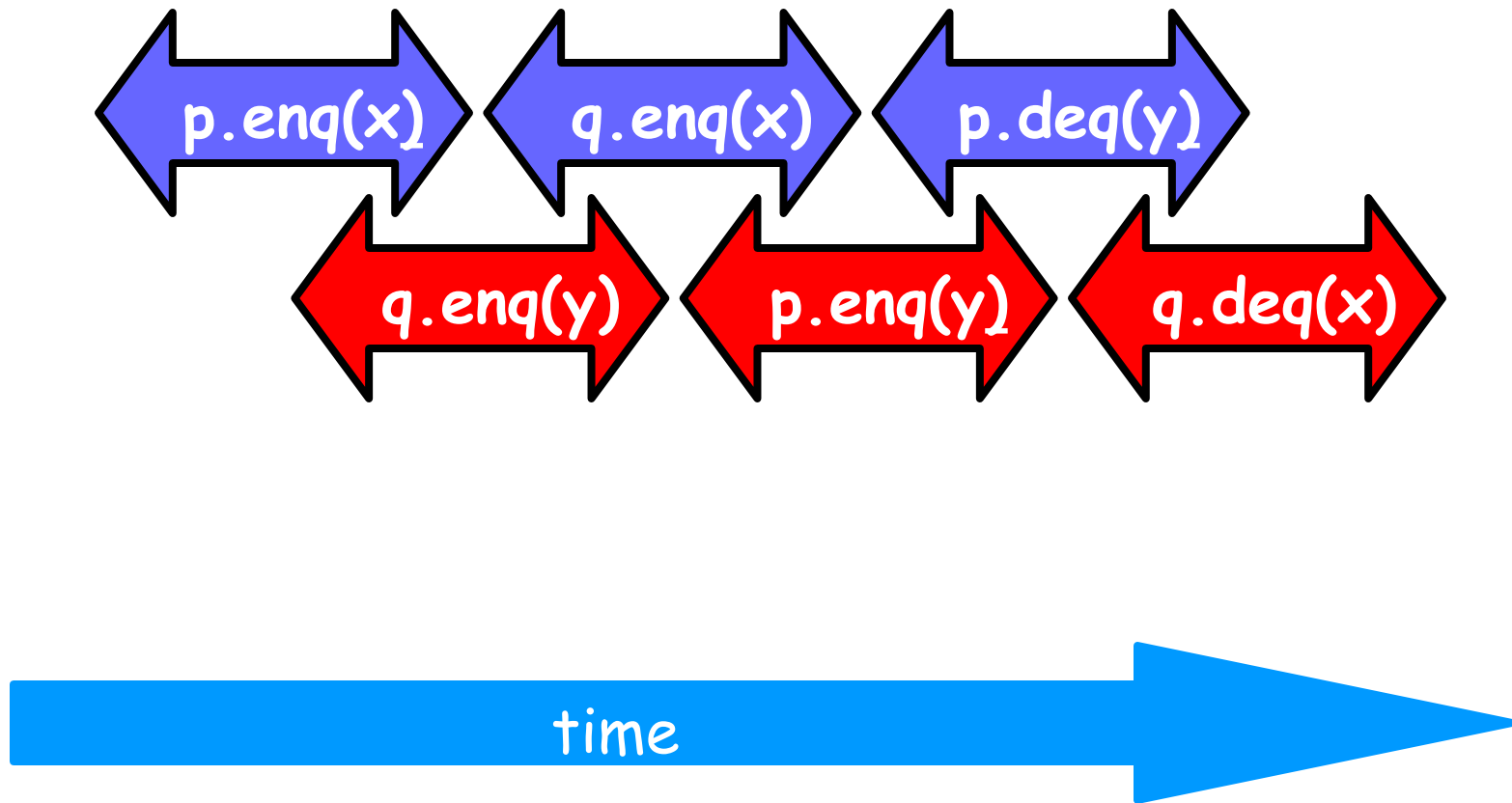
FIFO Queue Example



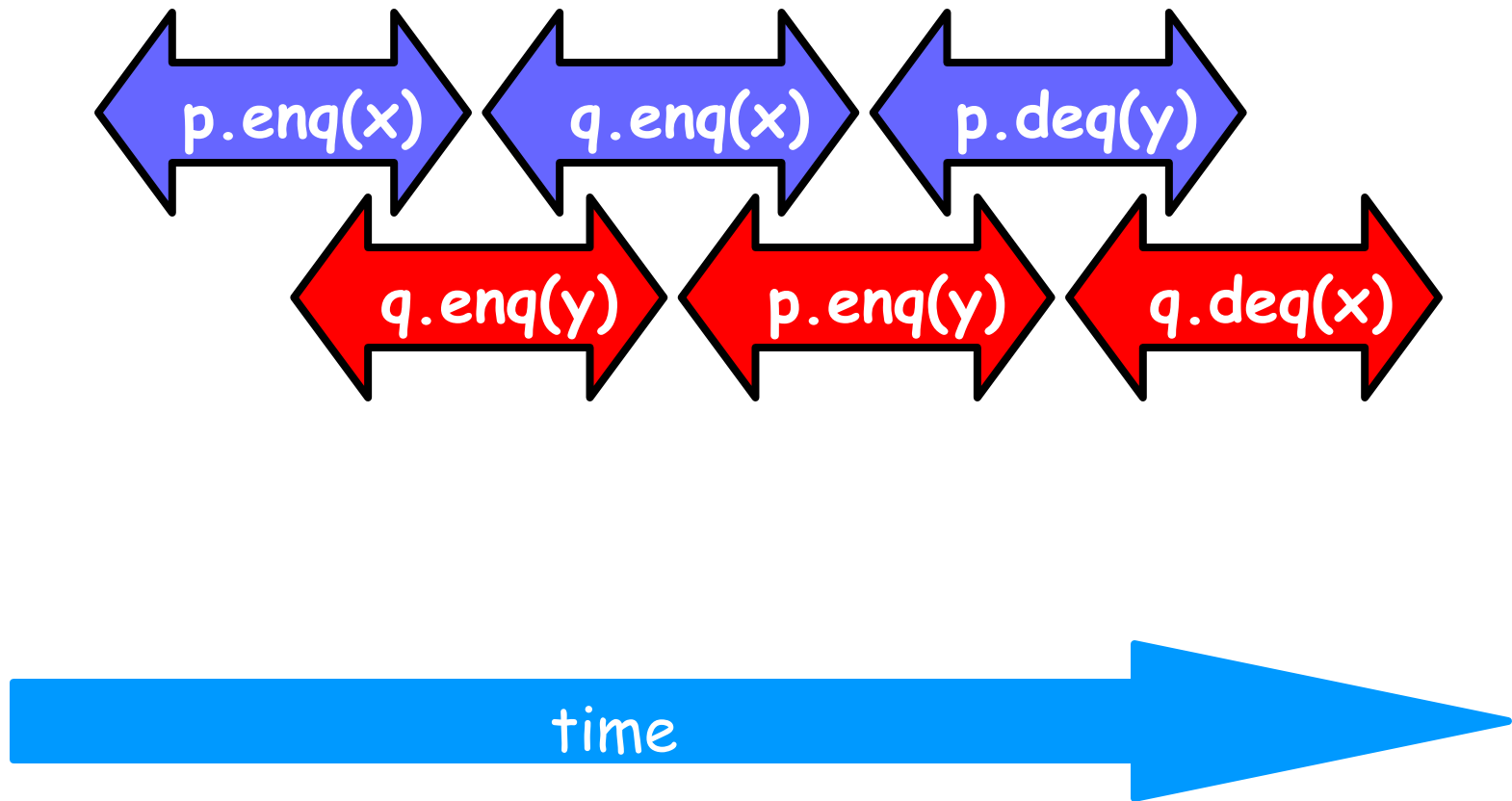
Histoire H



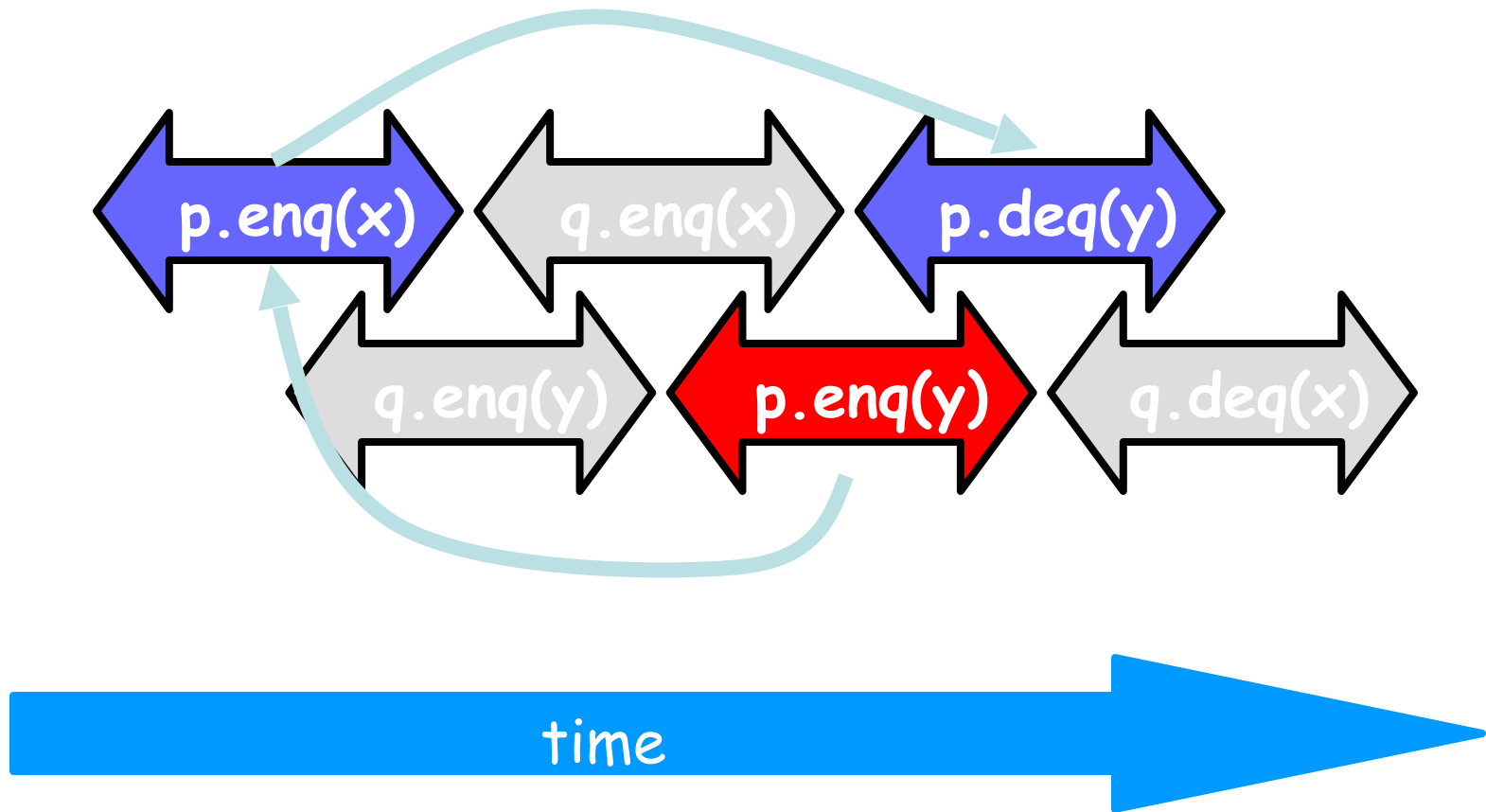
H/p Consistent Séquentiellement



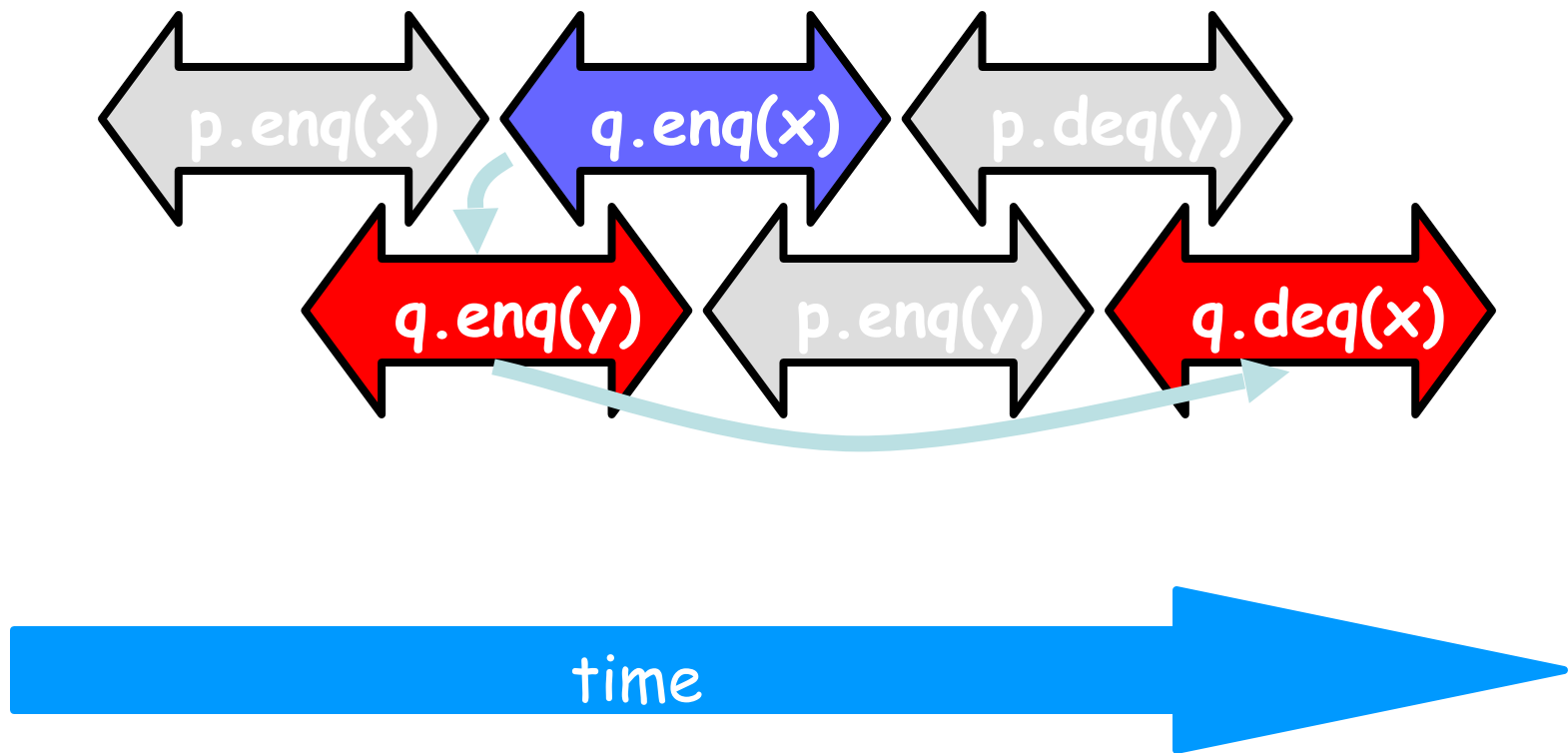
H/q Sequentially Consistent



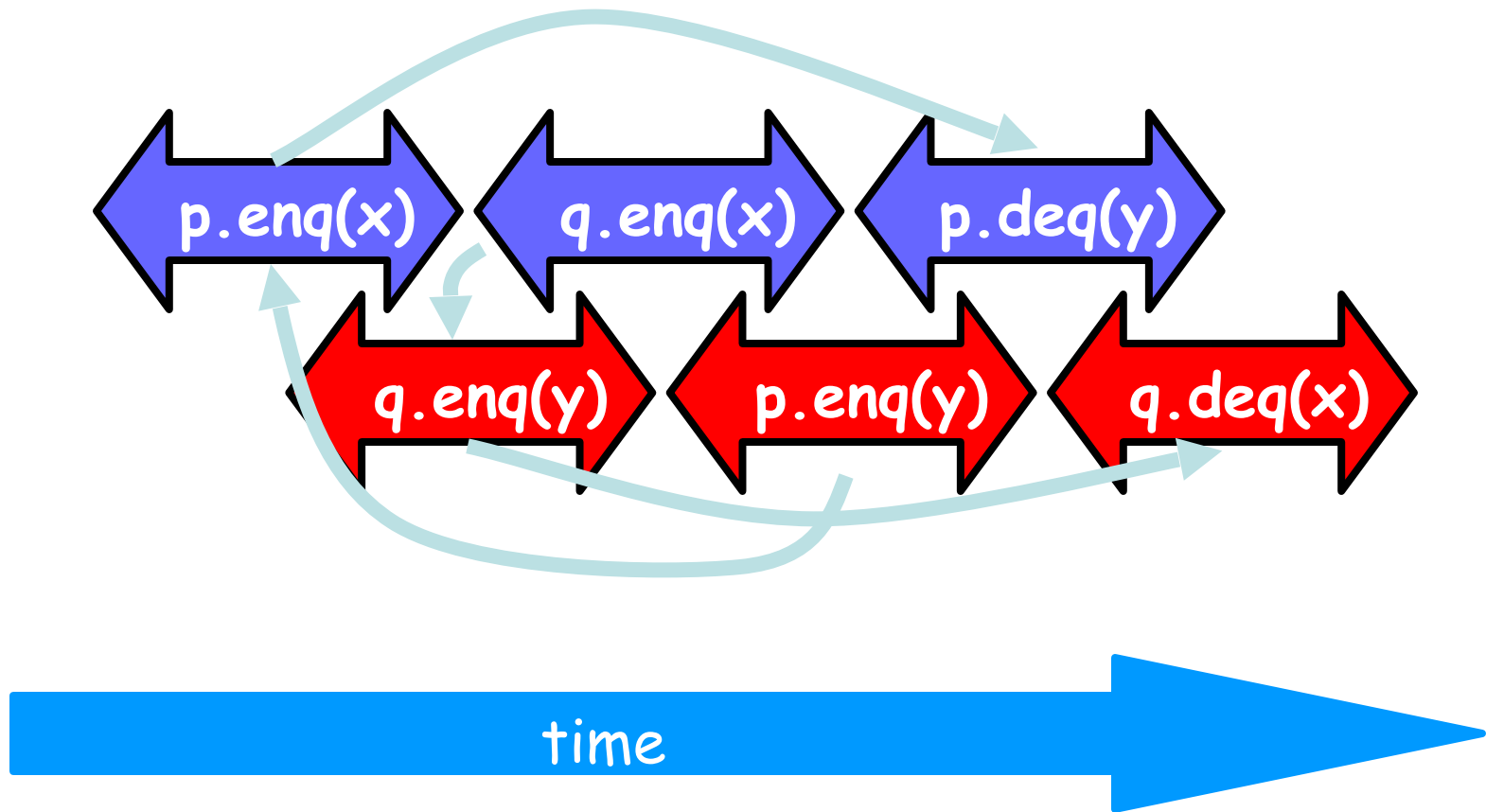
Ordre imposé par p



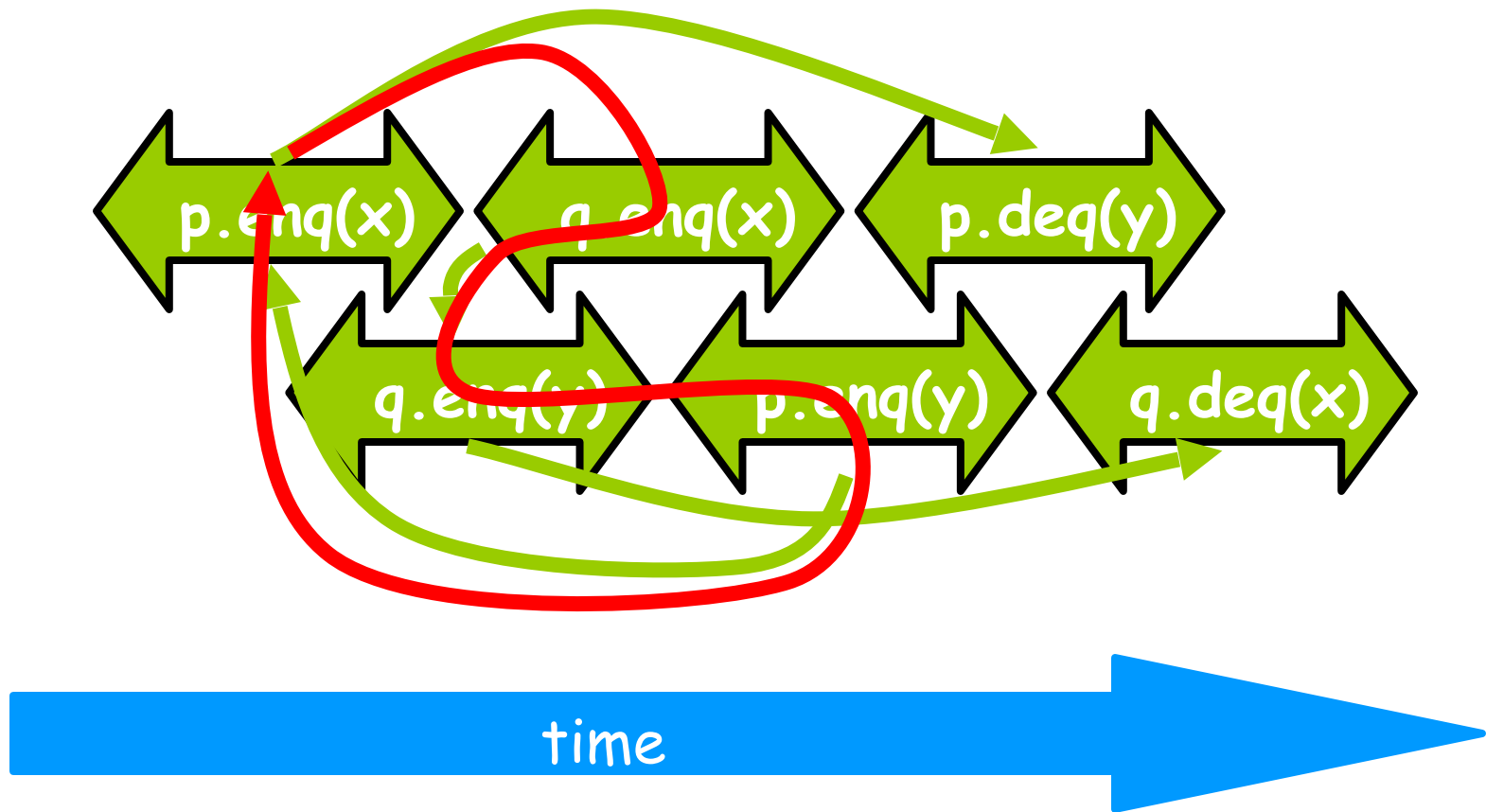
Ordre imposé par q



Ordre imposé par p et q



En combinant..



- Certaines architectures n'assurent même pas la consistance séquentielle
- Parce que les architectes pensent que c'est trop fort....

