Wait Free Synchronization

Maurice Herlihy, ACM TOPLAS, Jan 1991

Original slides by Tengyu Ma, 2010

Lightly updated by John Mellor-Crummey 21 March 2019

OUTLINE

- Motivation
- Wait-free object model
- Consensus problem
- Wait-free solutions to the consensus problem
- Impossibility proofs
- Universal construction

OUTLINE

- Motivation
- Wait-free object model
- Consensus problem
- Wait-free solutions to the consensus problem
- Impossibility proofs
- Universal construction

MOTIVATION

Concurrent objects in shared memory

- Traditional approach: mutual exclusion using locks
- Some problems with mutual exclusion
  - no fault tolerance
    - a thread may fail in the critical section
  - a slow thread may delay others

Objects without waiting?

- New approach: wait-free concurrent object
  - a thread can proceed independent of others
- Questions:
  - what wait-free objects are impossible?
  - how can we implement wait-free objects?

The Main Problem

- Given two concurrent objects X, Y.
- Is it possible to implement X by using Y?

OUTLINE

- Motivation
- Wait-free object model
- Consensus problem
- Wait-free solutions to the consensus problem
- Impossibility proofs
- Universal construction

Wait Free Concurrent Objects

Definition: A concurrent object is wait-free if every thread completes a method in finite number of steps

The Main Question

- How to implement concurrent object X by Y?
- Previous work
  - from single-writer single-reader boolean safe register, we can build multi-writer multi-reader atomic register
- This paper shows that an atomic register is a weak concurrent object

Understanding the Possibilities

- *Theorem: It is impossible to build a wait-free queue from atomic registers*
- How can one prove theorem like this?
- Basic idea:
    - determine a consensus number for each type of concurrent object
    - show that objects with low consensus number cannot implement ones with high consensus number

OUTLINE

- Motivation
- Wait-free object model
- Consensus problem
- Wait-free solutions to the consensus problem
- Impossibility proofs
- Universal construction

Consensus Problem

- Suppose there are n threads
- Each thread starts with an input value
- By executing some protocol, each outputs a value
- Three requirements:
    - Consistency: all threads decide the same value
    - Wait free: every thread eventually decides some value
    - Validity: the value decided is from the set of inputs

Parameters for Consensus

- Two factors that should be specified
  - What shared data-structure is used?
  - How many threads?

Consensus Number

Object ⟷ Supports n-thread consensus protocol

The <u>consensus number (CN)</u> for object type X is the largest number n, for which there exists a consensus protocol of n threads using objects of type X and atomic registers.

Consensus Number

- Consensus number measures synchronization power
- Classify objects by consensus number (CN)
  - objects with different CN in different classes
  - object with CN N cannot implement any objects with CN of M > N.

Consensus Hierarchy

consensus number	Objects
1	register
2	test&set, swap, fetch&add, queue,stack
...	......
2n-2	n-register assignment
....	......
∞	memory to memory move and swap, augmented queue, compare&swap,fetch&cons, sticky byte

OUTLINE

- Motivation
- Wait-free object model
- Consensus problem
- Wait-free solutions to the consensus problem
- Impossibility proofs
- Universal construction

Queue Consensus Number

- ***Theorem:*** *Queue has consensus number at least 2*

Queue Consensus Number

- ***Theorem:*** *Queue has consensus number at least 2*
- **Proof**
  - Queue initially with two entries 0,1
  - Two shared **atomic** registers prefer[0], prefer[1]

Queue Consensus Number

- ***Theorem:*** *Queue has consensus number at least 2*
- **Proof**
  - Queue initially with two entries 0,1
  - Two shared **atomic** registers prefer[0], prefer[1]

Algorithm 1 Algorithm for P_i with input value v_i

prefer[i] := v_i
if deque() = 0 **then**
    return prefer[i]
else
    return prefer[1-i]
end if

Queue Consensus Number

- ***Theorem:*** *Queue has consensus number at least 2*
- **Proof**
  - Queue initially with two entries 0,1
  - Two shared **atomic** registers prefer[0], prefer[1]

Algorithm 1 Algorithm for P_i with input value v_i

prefer[i] := v_i
if deque() = 0 **then**
  return prefer[i]
else
  return prefer[1-i]
end if

If deque() = 1, the thread can always read the value written by the other. Why?

Queue Consensus Number

- ***Theorem:*** *Queue has consensus number at least 2*
- **Proof**
  - Queue initially with two entries 0,1
  - Two shared **atomic** registers prefer[0], prefer[1]

Is it wait-free ?

Algorithm 1 Algorithm for P_i with input value v_i

prefer[i] := v_i
if deque() = 0 **then**
  return prefer[i]
else
  return prefer[1-i]
end if

If deque() = 1, the process can always read the value written by the other. Why?

Augmented Queue Consensus Number

Definition: *An augmented queue is a FIFO queue with a peek operation, which returns the head of the queue without changing it.*

Augmented Queue Consensus Number

Definition: *An augmented queue is a FIFO queue with a peek operation, which returns the head of the queue without changing it.*

Theorem: *Augmented Queue has infinite consensus number*

Augmented Queue Consensus Number

Definition: *An augmented queue is a FIFO queue with a peek operation, which returns the head of the queue without changing it.*

Theorem: *Augmented Queue has infinite consensus number*

Algorithm 1 Algorithm for P_i

Require: P_i has input v_i

1: enq(v_i)
2: **return** peek()

Augmented Queue Consensus Number

Definition: *An augmented queue is a FIFO queue with peek operation which return the head of the queue without changing it.*

Theorem: *Augmented Queue has infinite consensus number*

Algorithm 1 Algorithm for P_i

Require: P_i has input v_i

1: enq(v_i)
2: **return** peek()

Is it wait-free?

N-Register Assignment

- ***Definition***(*Multiple Assignment): The expression*

$$r_1, r_2, \ldots, r_n := v_1, \ldots, v_n$$

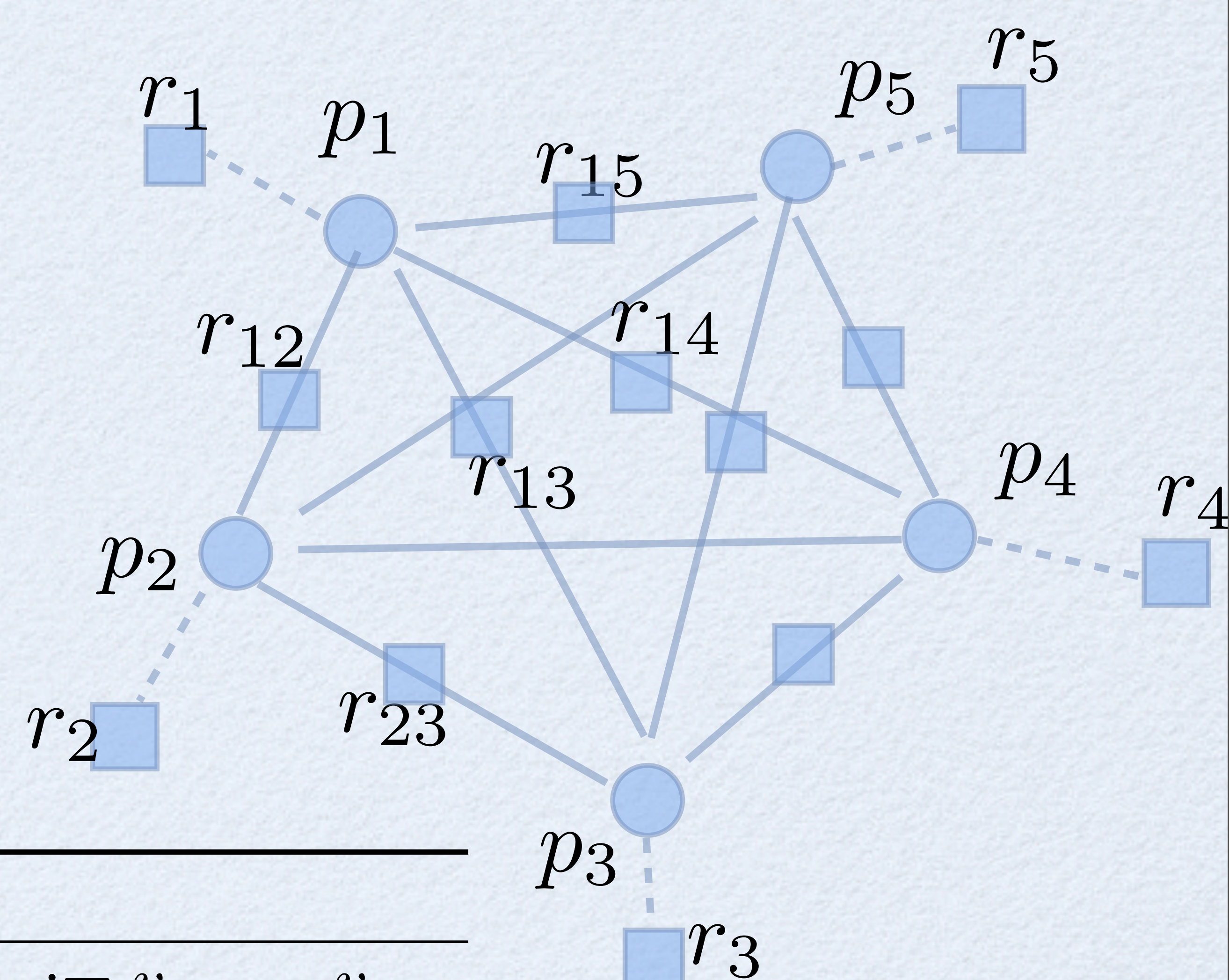
atomically assign each value v_i *to each register* r_i

Theorem: *Registers with atomic m-assignment have consensus number at least m*

N-Register Assignment Cont'd

Proof:

- Each thread has a single-writer register.
- Each two threads share a multi-writer register



Algorithm 1 Algorithm for P_i

---

atomically assign $r_i, r_{i1}, r_{i2}, \ldots, r_{in} := v_i, \ldots, v_i$
return determineFirstAssignment()

N-Register Assignment

Algorithm 1 Algorithm for p_i

atomically assign $r_i, r_{i1}, \ldots, r_{in} := v_1, \ldots, v_i$
return determineFirstAssignment()

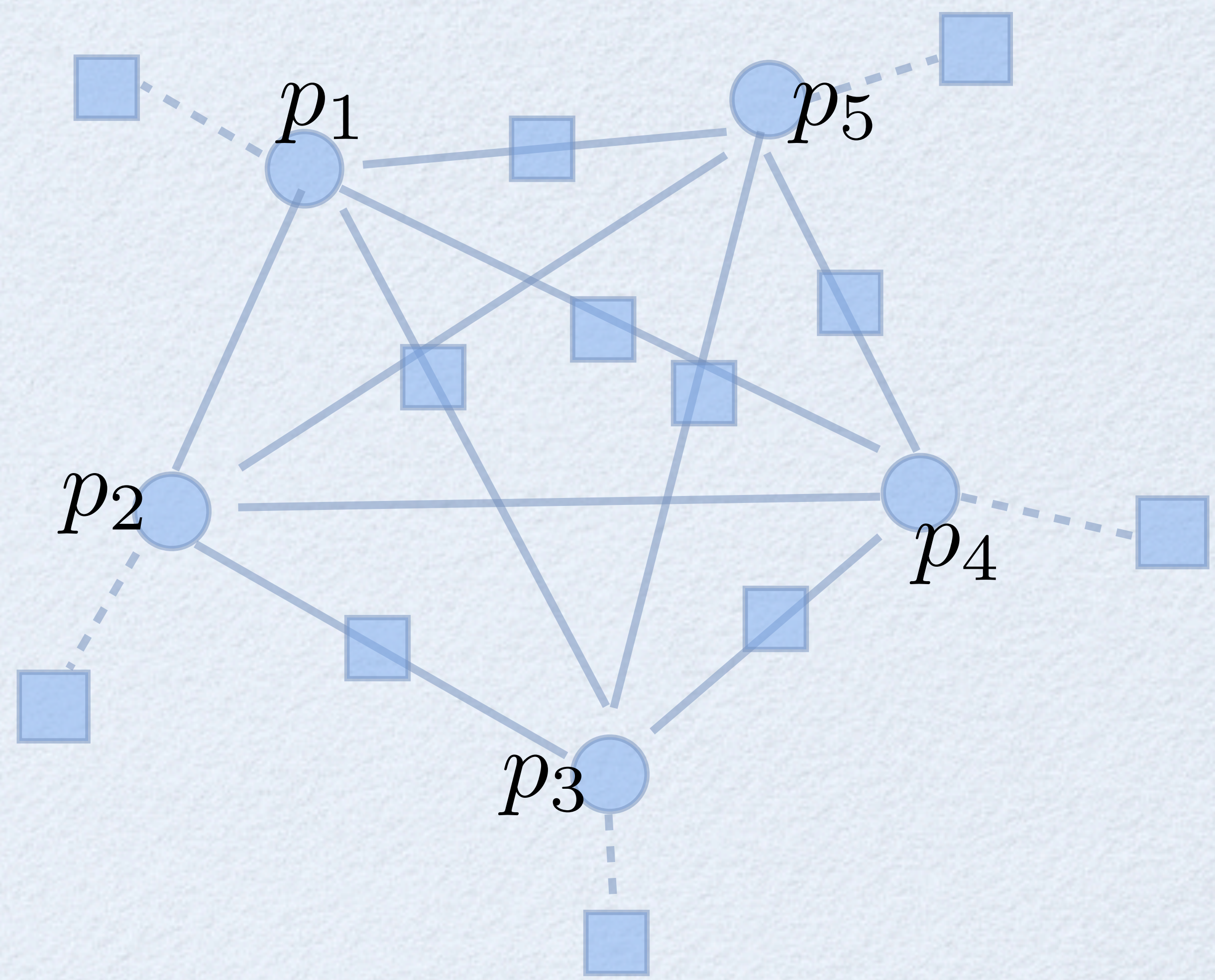
Algorithm 2 determineFirstAssignment

for all $1 \leq i < j \leq n$ **do**
  determineOrder(i,j)
end for

Algorithm 3 determineOrder(i,j)

Ensure: determine the order between occurred assignment

if r_{ij} has not been initialized **then**
  assignments by p_i, p_j has not occurred.
else if r_i is not initialized but r_j is initialized **then**
  p_j precedes p_i
else if r_j is not initialized but r_i is initialized **then**
  p_i precedes p_j
else
  if $r_i = r_{ij}$ **then**
    p_j precedes p_i
  else
    p_i precedes p_j
  end if
end if



N-Register Assignment

Algorithm 1 Algorithm for p_i

```
atomically assign r_i, r_i1, ..., r_in := v_1, ..., v_i
return determineFirstAssignment()
```

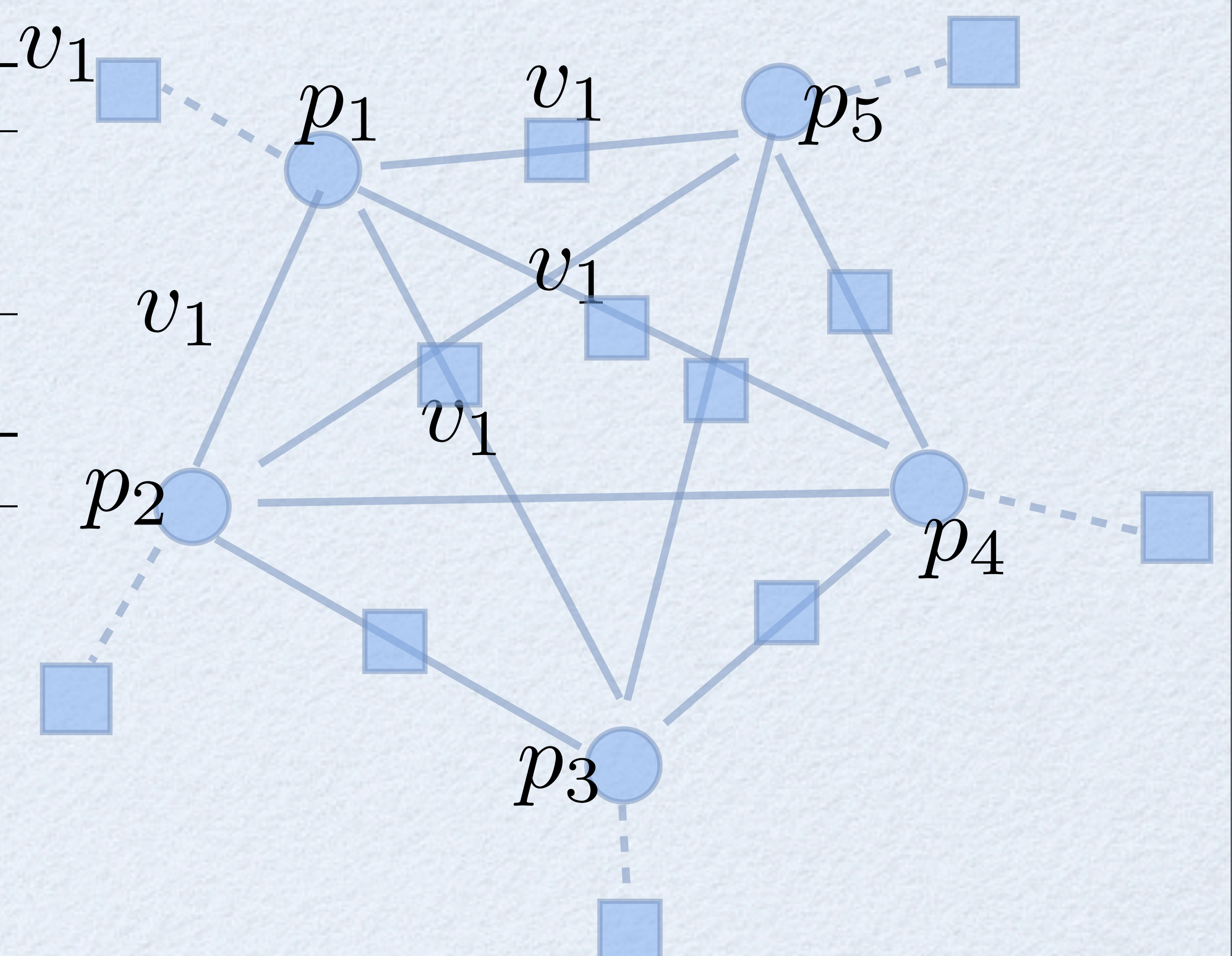
Algorithm 2 determineFirstAssignment

```
for all 1 ≤ i < j ≤ n do
    determineOrder(i,j)
end for
```

Algorithm 3 determineOrder(i,j)

Ensure: determine the order between occurred assignment

```
if r_ij has not been initialized then
    assignments by p_i, p_j has not occurred.
else if r_i is not initialized but r_j is initialized then
    p_j precedes p_i
else if r_j is not initialized but r_i is initialized then
    p_i precedes p_j
else
    if r_i = r_ij then
        p_j precedes p_i
    else
        p_i precedes p_j
    end if
end if
```



N-Register Assignment

Algorithm 1 Algorithm for p_i

atomically assign $r_i, r_{i1}, \ldots, r_{in} := v_1, \ldots, v_i$
return determineFirstAssignment()

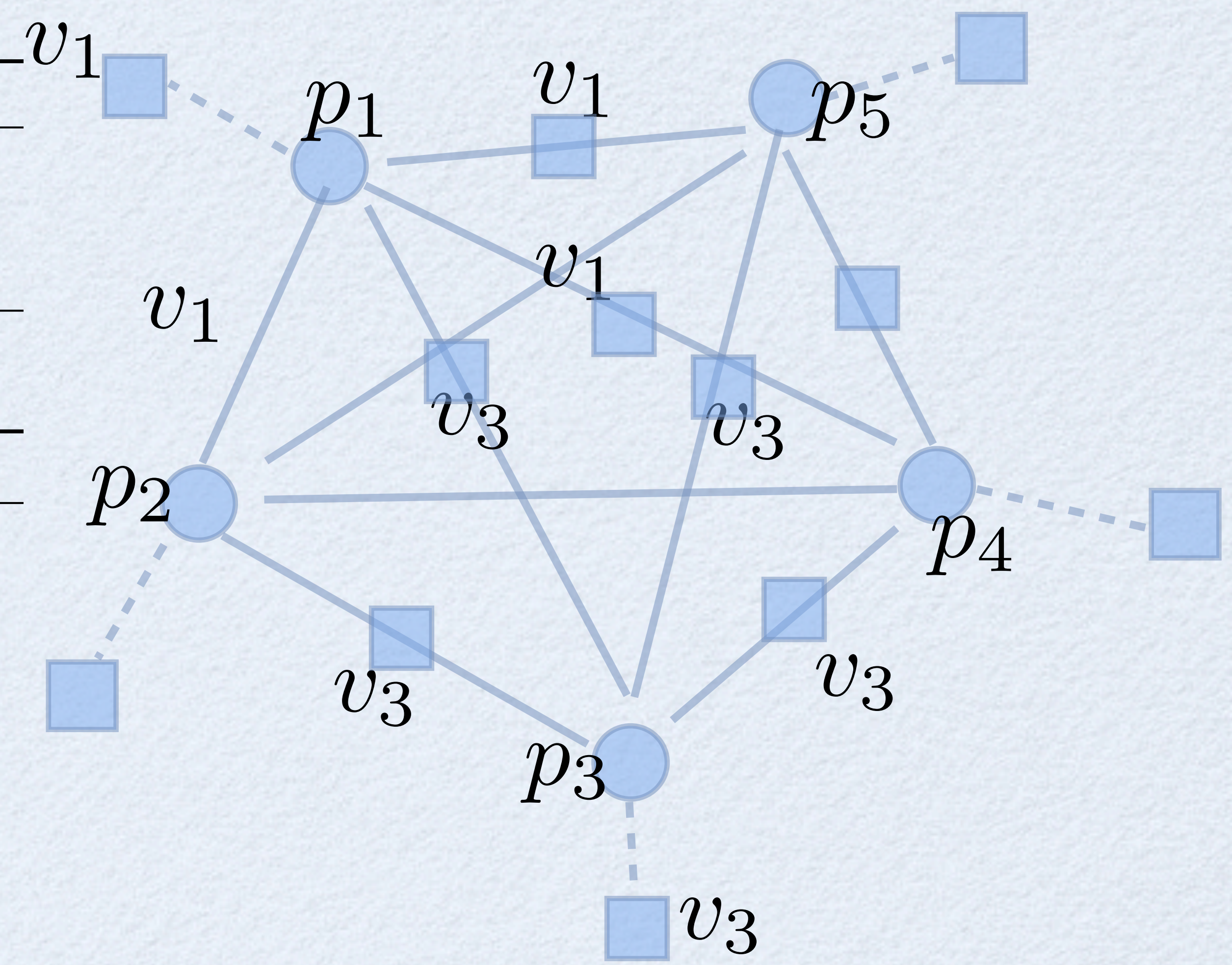
Algorithm 2 determineFirstAssignment

for all $1 \leq i < j \leq n$ **do**
    determineOrder(i,j)
end for

Algorithm 3 determineOrder(i,j)

Ensure: determine the order between occurred assignment

if r_{ij} has not been initialized **then**
    assignments by p_i, p_j has not occurred.
else if r_i is not initialized but r_j is initialized **then**
    p_j precedes p_i
else if r_j is not initialized but r_i is initialized **then**
    p_i precedes p_j
else
    if $r_i = r_{ij}$ **then**
        p_j precedes p_i
    else
        p_i precedes p_j
    end if
end if



N-Register Assignment

Algorithm 1 Algorithm for p_i

atomically assign $r_i, r_{i1}, \ldots, r_{in} := v_1, \ldots, v_i$
return determineFirstAssignment()

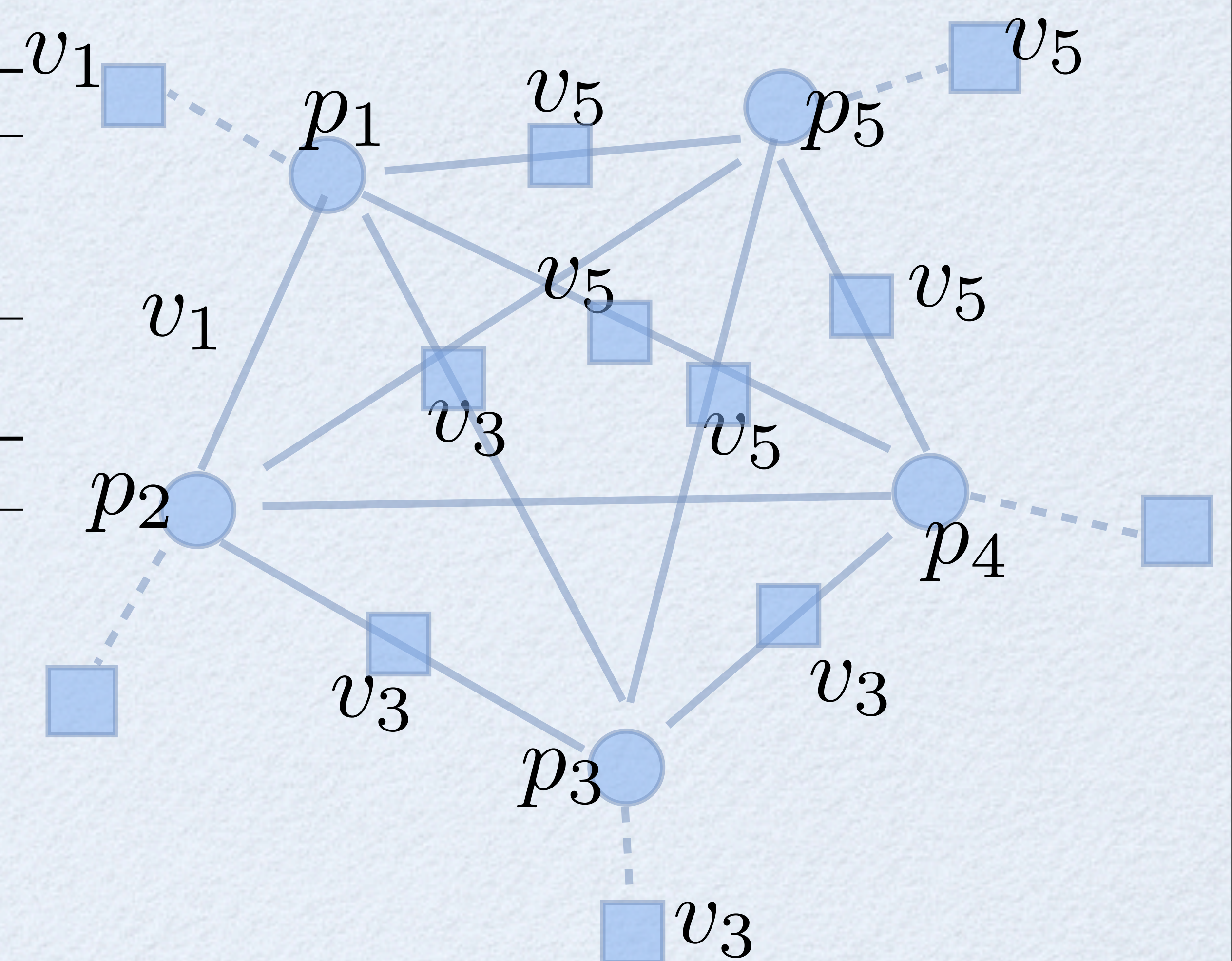
Algorithm 2 determineFirstAssignment

for all $1 \leq i < j \leq n$ **do**
  determineOrder(i,j)
end for

Algorithm 3 determineOrder(i,j)

Ensure: determine the order between occurred assignment

if r_{ij} has not been initialized **then**
  assignments by p_i, p_j has not occurred.
else if r_i is not initialized but r_j is initialized **then**
  p_j precedes p_i
else if r_j is not initialized but r_i is initialized **then**
  p_i precedes p_j
else
  if $r_i = r_{ij}$ **then**
    p_j precedes p_i
  else
    p_i precedes p_j
  end if
end if



OUTLINE

- Motivation
- Wait-free object model
- Consensus problem
- Wait-free solutions to the consensus problem
- Impossibility proofs
- Universal construction

Impossibility Results

Proof Terminology

- Protocol state: The states of all the concurrent objects and the internal states of the algorithms run in every processes
- A state is *bivalent* if starting from this state, any decision is still possible.
- A state is *x-valent* if starting from this state, the only possible decision value is x.
- A state is *univalent* if it is x-valent for some value x

Examples

- A state is *x-valent* if starting from this state, the only possible decision value is x.
- A state is *bivalent* if starting from this state, either decision is still possible.
- A state is *univalent* if it is x-valent for some value x

Examples

- A state is *x-valent* if starting from this state, the only possible decision value is x.
- A state is *bivalent* if starting from this state, either decision is still possible.
- A state is *univalent* if it is x-valent for some value x

Algorithm 1 Algorithm for P_i with input value v_i

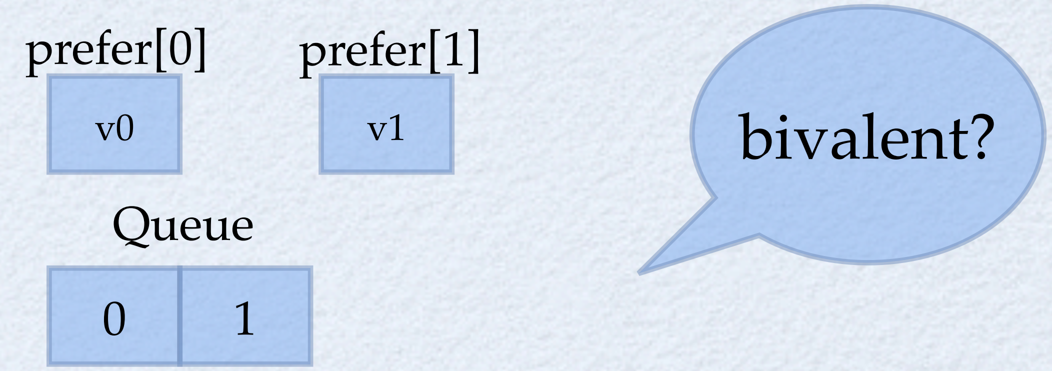
prefer[i] := v_i
if deque() = 0 **then**
    return prefer[i]
else
    return prefer[1-i]
end if

Examples

- A state is *x-valent* if starting from this state, the only possible decision value is x.
- A state is *bivalent* if starting from this state, either decision is still possible.
- A state is *univalent* if it is x-valent for some value x

Algorithm 1 Algorithm for P_i with input value v_i

```
prefer[i] := v_i
if deque() = 0 then
   return prefer[i]
else
   return prefer[1-i]
end if
```



Examples

- A state is *x-valent* if starting from this state, the only possible decision value is x.
- A state is *bivalent* if starting from this state, either decision is still possible.
- A state is *univalent* if it is x-valent for some value x

Algorithm 1 Algorithm for P_i with input value v_i

prefer[i] := v_i
if deque() = 0 **then**
    return prefer[i]
else
    return prefer[1-i]
end if

Decision Step

- A decision step is an operation which carries the protocol from a bivalent state to a univalent state.
- *Proposition: There exists a state, such that every feasible operation on it is a decision step.*
- *The state should be reachable from the initial state.*

Decision Step

- A decision step is an operation which carries the protocol from a bivalent state to a univalent state.

Algorithm 1 Algorithm for P_i with input value v_i

```
prefer[i] := v_i
if deque() = 0 then
    return prefer[i]
else
    return prefer[1-i]
end if
```

- *Proposition: There exists a state, such that every feasible operation on it is a decision step.*

- *The state should be reachable from the initial state.*

Decision Step

- A decision step is an operation which carries the protocol from a bivalent state to a univalent state.

- *Proposition: There exists a state, such that every feasible operation on it is a decision step.*

- *The state should be reachable from the initial state.*

Algorithm 1 Algorithm for P_i with input value v_i

prefer[i] := v_i
if deque() = 0 **then**
    return prefer[i]
else
    return prefer[1-i]
end if

The first deque() operation is the decision step

Decision Step

- A decision step is an operation which carries the protocol from a bivalent state to a univalent state.

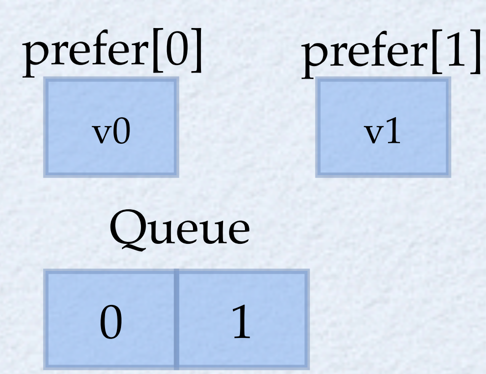
Algorithm 1 Algorithm for P_i with input value v_i

```
prefer[i] := v_i
if deque() = 0 then
    return prefer[i]
else
    return prefer[1-i]
end if
```

The first deque() operation is the decision step

- *Proposition: There exists a state, such that every feasible operation on it is a decision step.*

- *The state should be reachable from the initial state.*



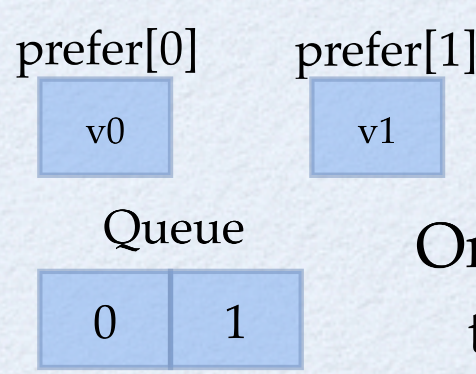
DECISION STEP

- A decision step is an operation which carries the protocol from a bivalent state to a univalent state.
- *Proposition: There exists a state, such that every feasible operation on it is a decision step.*
- *The state should be reachable from the initial state.*

Algorithm 1 Algorithm for P_i with input value v_i

```
prefer[i] := v_i
if deque() = 0 then
    return prefer[i]
else
    return prefer[1-i]
end if
```

The first deque() operation is the decision step



Only two feasible operations: the deque() of P1, and the deque() of P2

Critical State

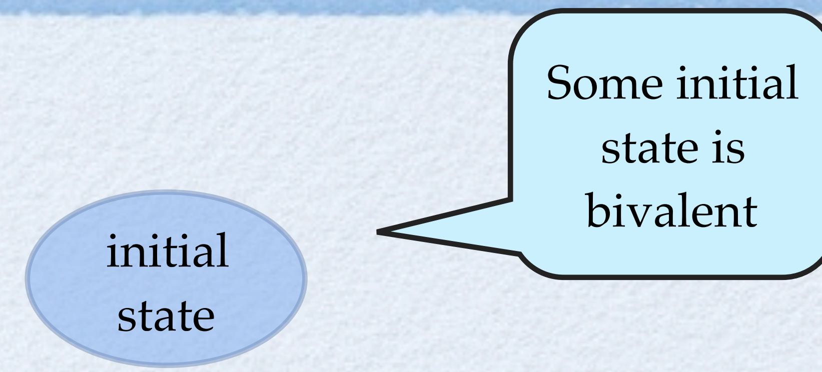
- ***Proposition***: *There exists a state, which can be reached from the initial state, such that every feasible operation on it is a decision step.*

- We call this state critical state

Critical State

- ***Proposition**: There exists a state, which can be reached from the initial state, such that every feasible operation on it is a decision step.*

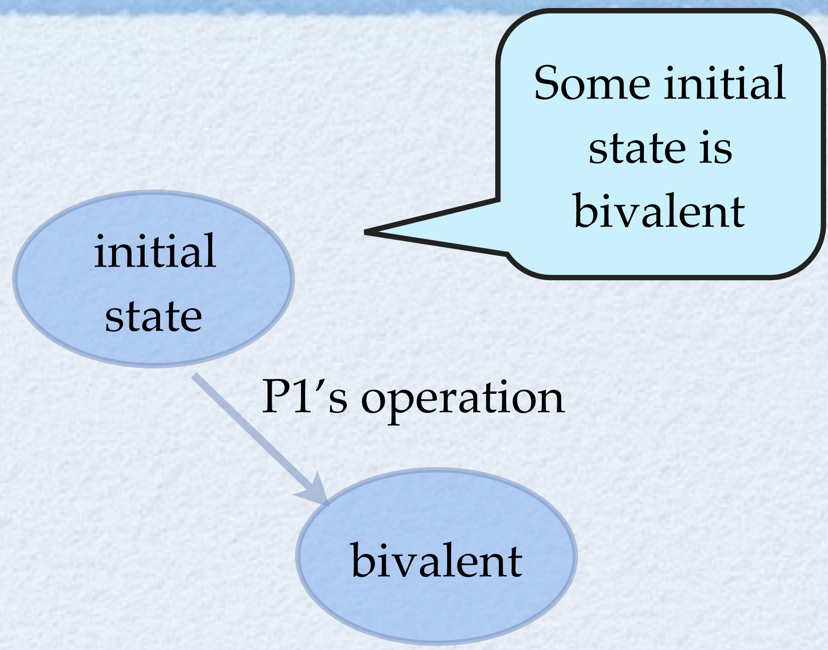
- We call this state critical state



Critical State

- ***Proposition**: There exists a state, which can be reached from the initial state, such that every feasible operation on it is a decision step.*

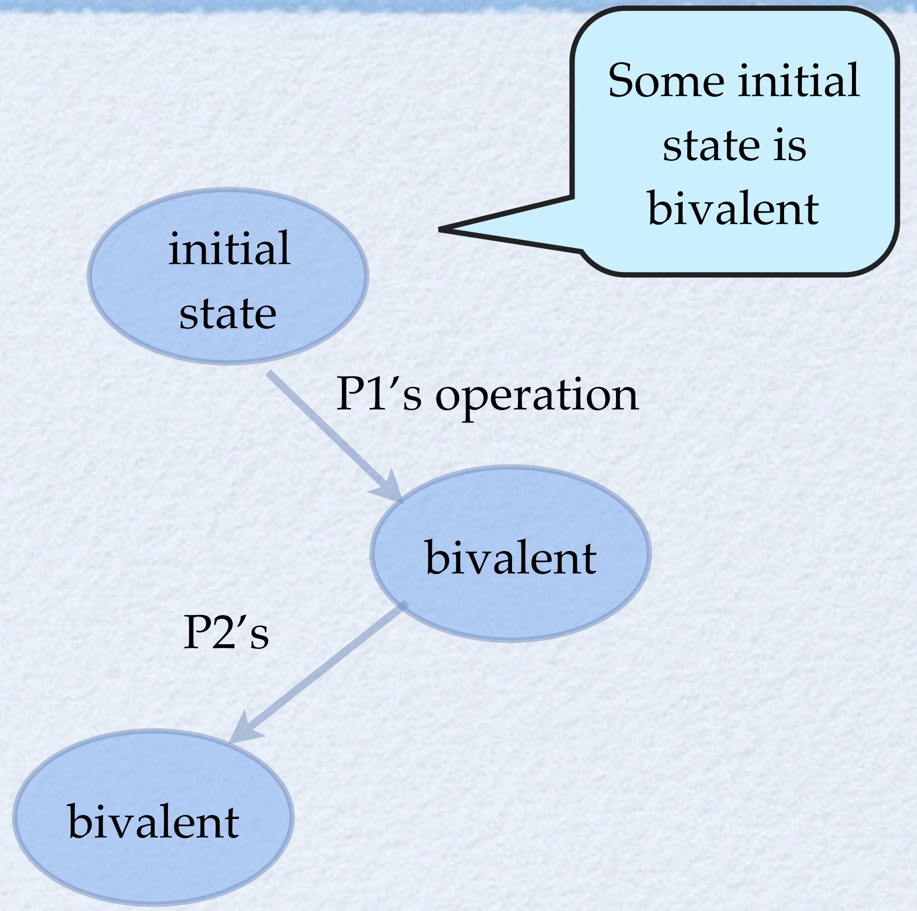
- We call this state critical state



Critical State

- ***Proposition**: There exists a state, which can be reached from the initial state, such that every feasible operation on it is a decision step.*

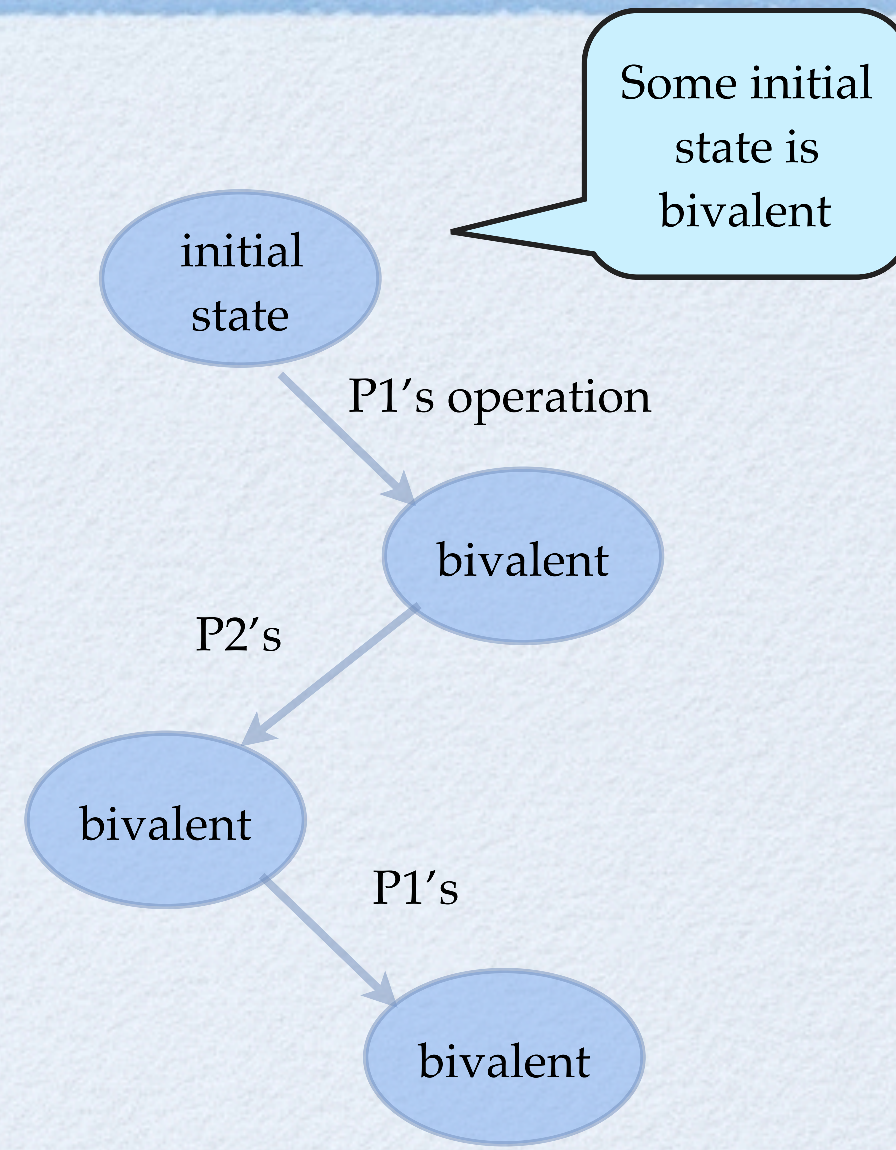
- We call this state critical state



Critical State

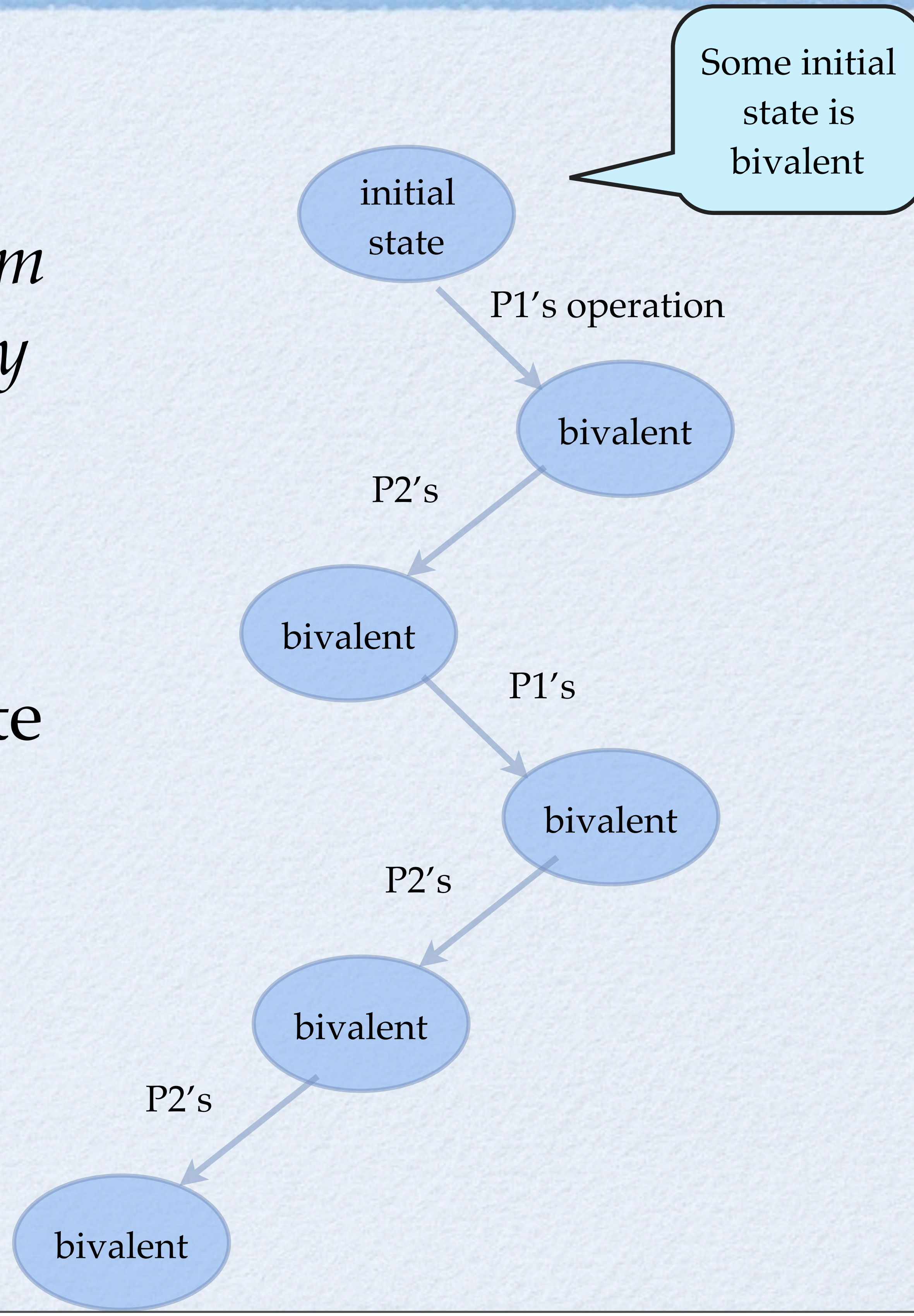
- ***Proposition***: *There exists a state, which can be reached from the initial state, such that every feasible operation on it is a decision step.*

- We call this state critical state



Critical State

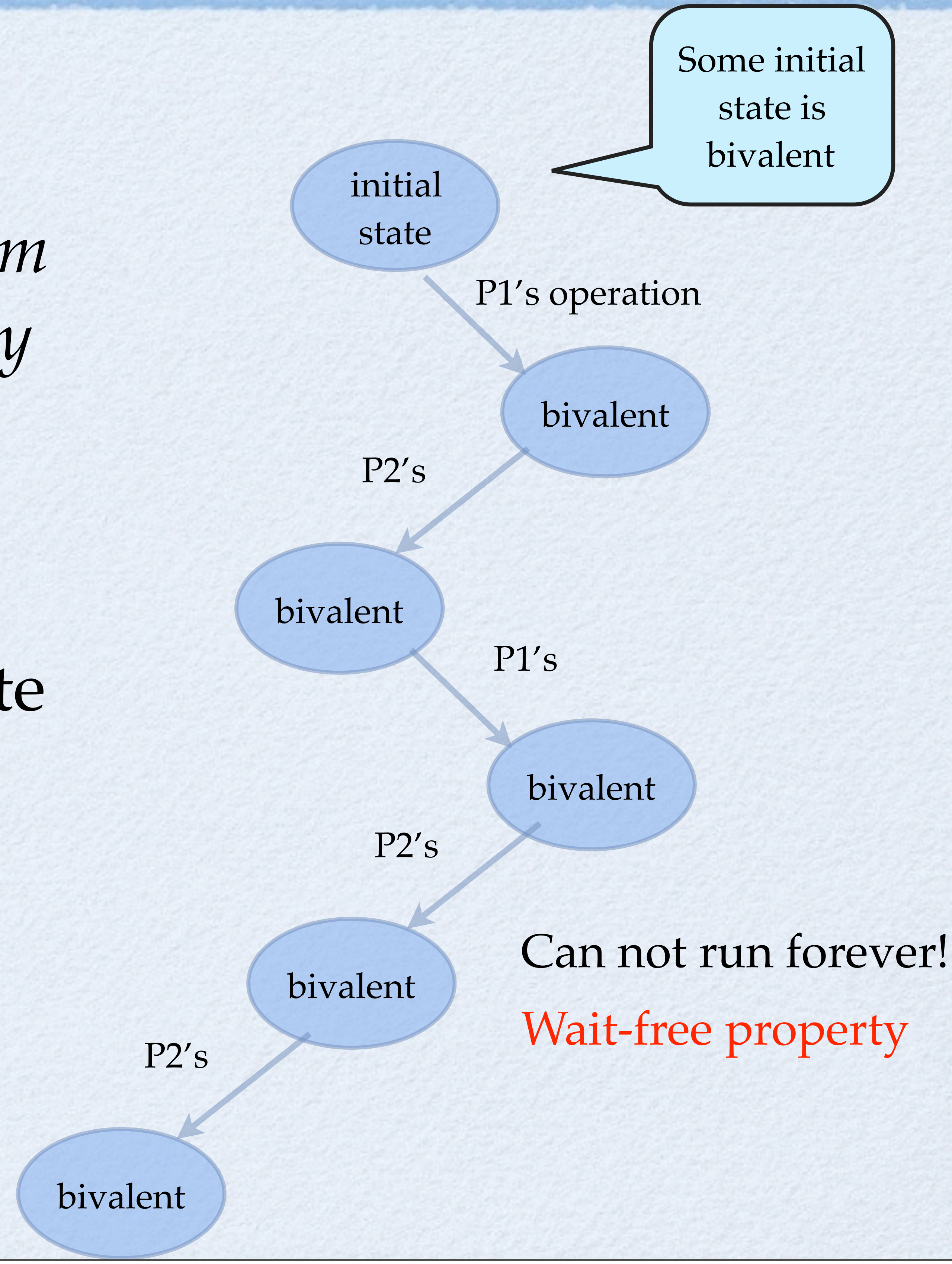
- ***Proposition***: *There exists a state, which can be reached from the initial state, such that every feasible operation on it is a decision step.*
- We call this state critical state



Critical State

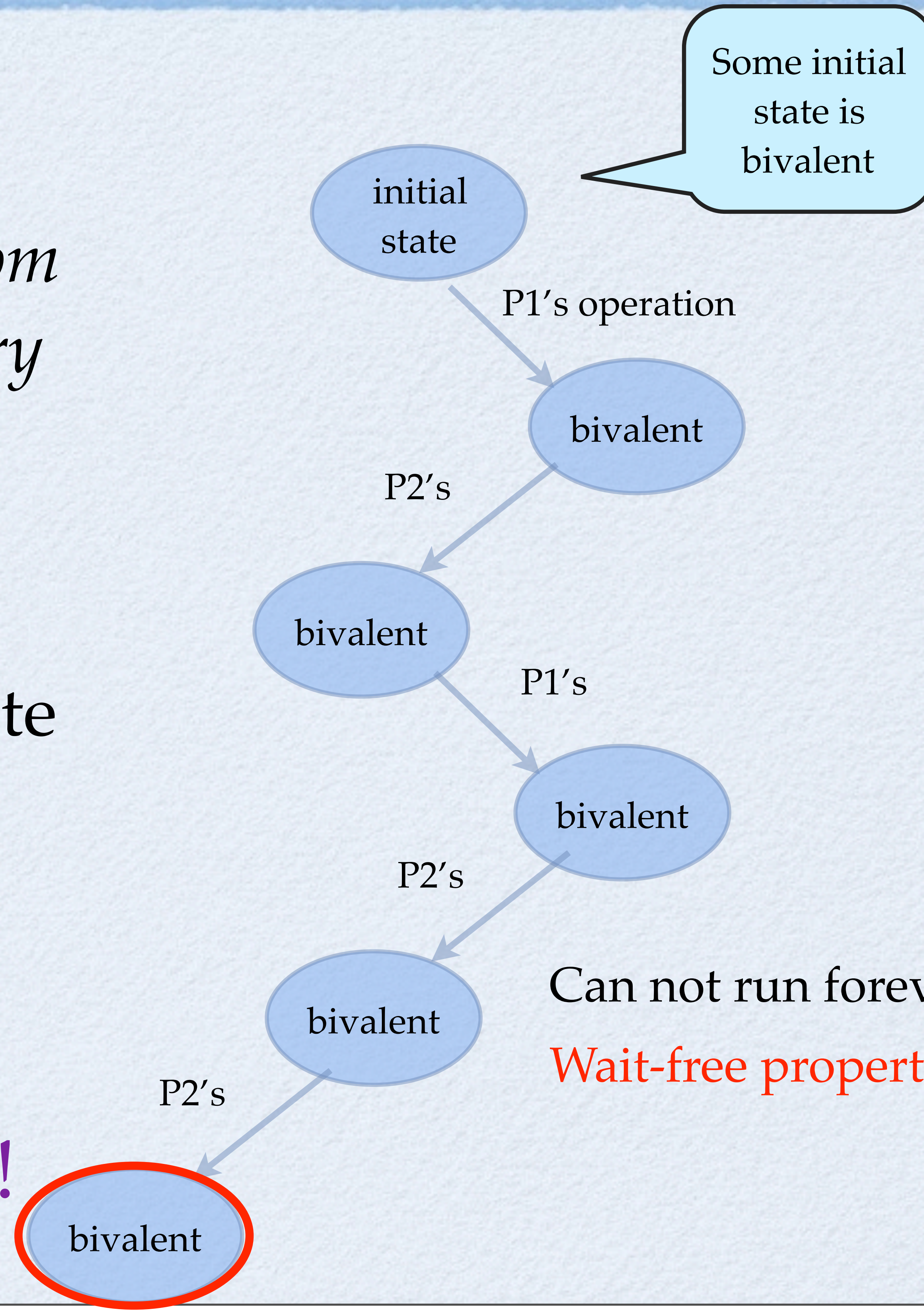
- ***Proposition****: There exists a state, which can be reached from the initial state, such that every feasible operation on it is a decision step.*

- We call this state critical state



Critical State

- ***Proposition***: *There exists a state, which can be reached from the initial state, such that every feasible operation on it is a decision step.*
- We call this state critical state



Can not run forever!

Wait-free property

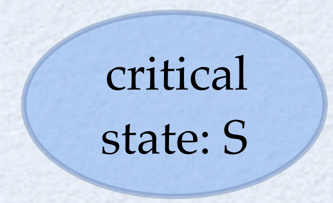
Finally reach a critical state!

Impossibility Results

- ***Theorem**: atomic registers cannot simulate 2-processes consensus protocol.*

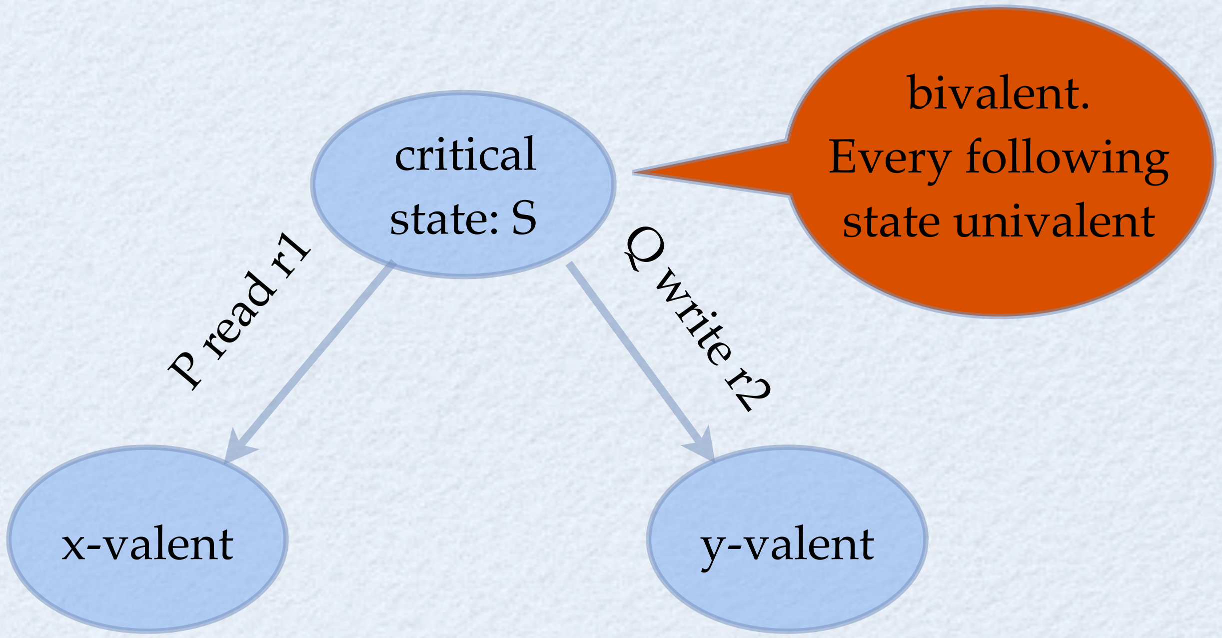
- Proof structure: Assume that there exists a protocol. Find the critical state (the state for which every operation on it is decision step).

- Enumerate all the possible cases of the operations following this state.



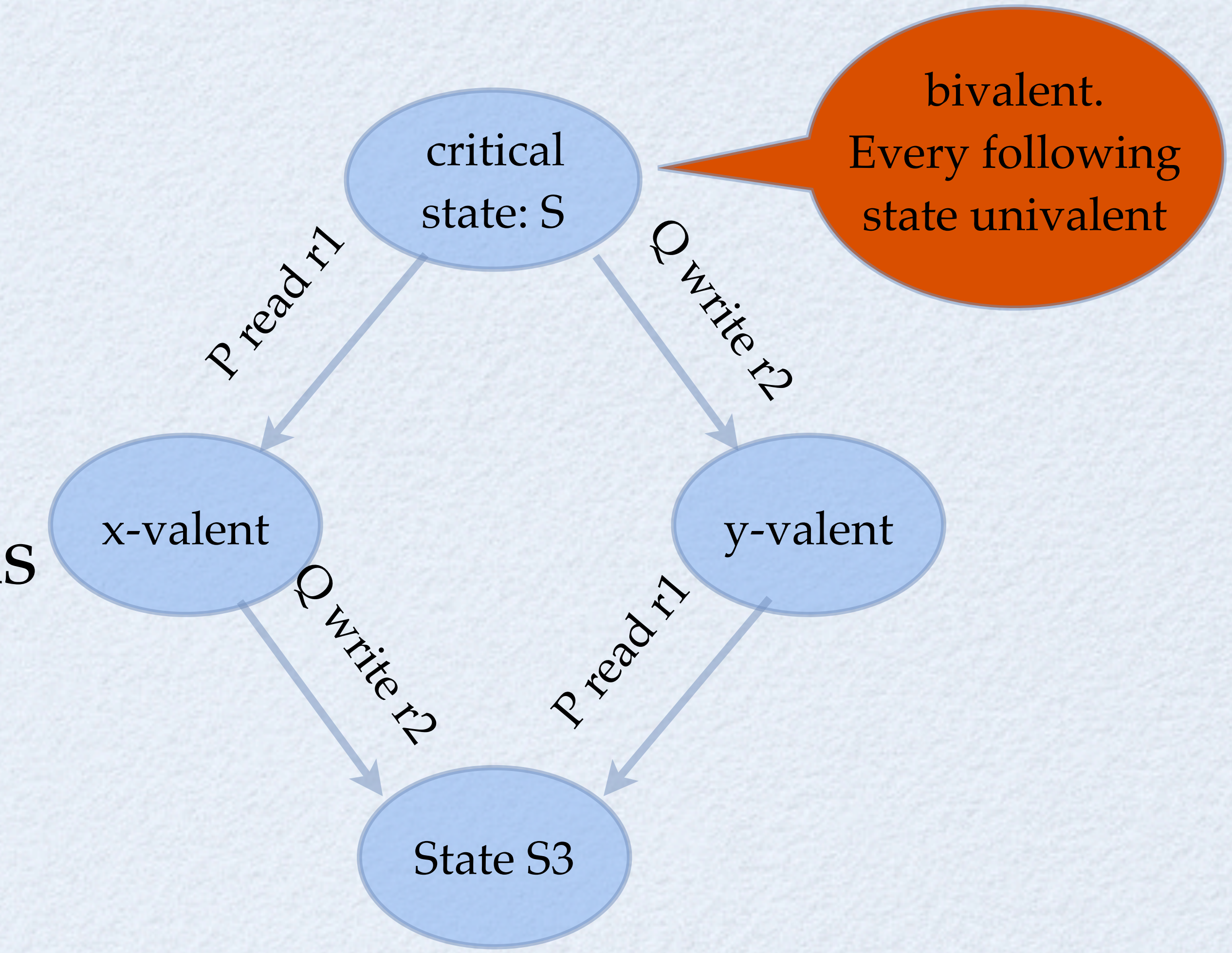
Atomic Registers - 1/3

- Consider: Two operations following the critical state are on different registers.
- Trivial since the two operations on different objects can be commuted without changing the final state.
- Every feasible operation on the critical state is on the same base object (register).



Atomic Registers - 1/3

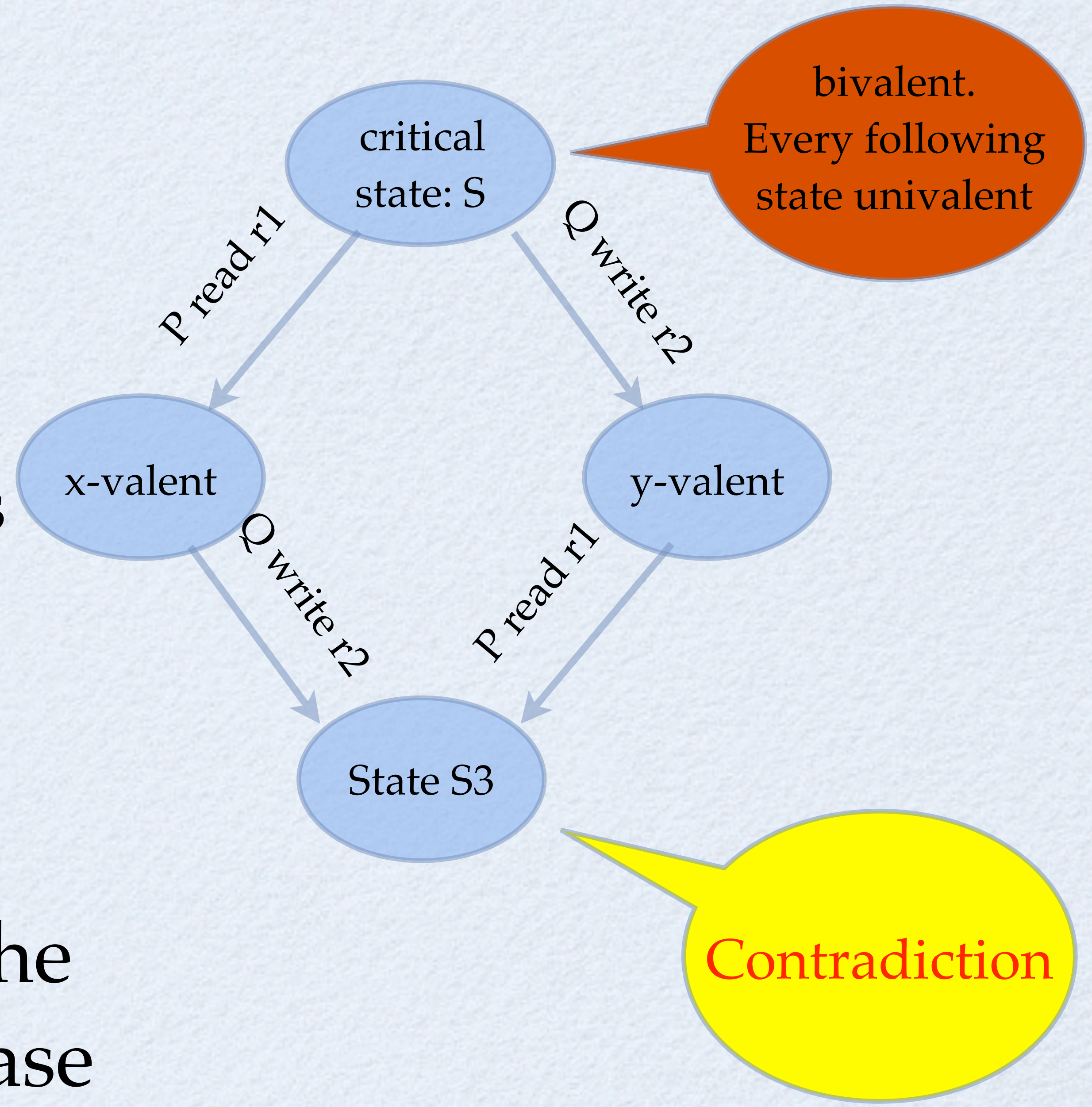
- Consider: Two operations following the critical state are on different registers.



- Trivial since the two operations on different objects can be commuted without changing the final state.
- Every feasible operation on the critical state is on the same base object (register).

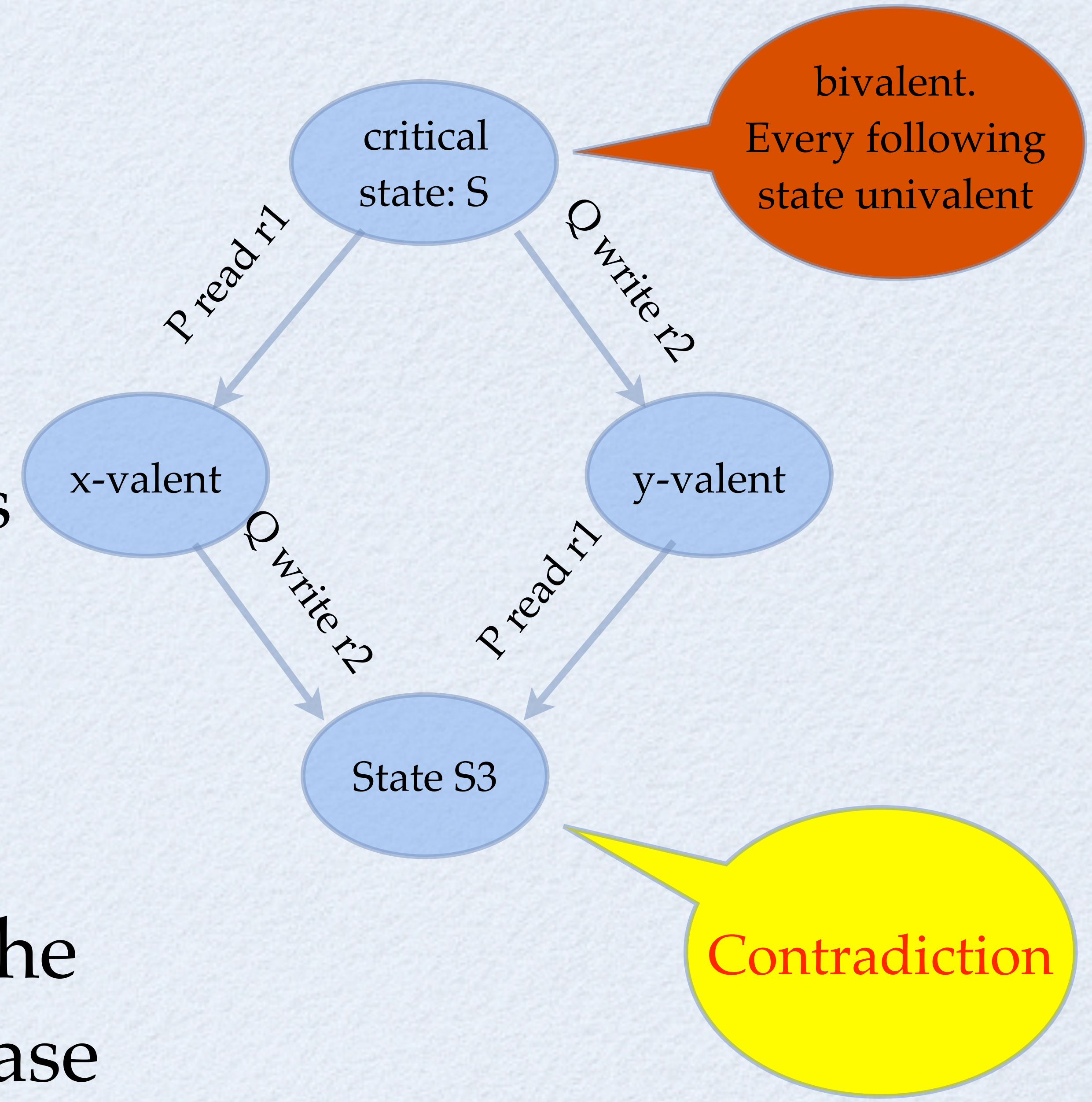
Atomic Registers - 1/3

- Consider: Two operations following the critical state are on different registers.
- Trivial since the two operations on different objects can be commuted without changing the final state.
- Every feasible operation on the critical state is on the same base object (register).



Atomic Registers - 1/3

- Consider: Two operations following the critical state are on different registers.
- Trivial since the two operations on different objects can be commuted without changing the final state.
- Every feasible operation on the critical state is on the same base object (register).



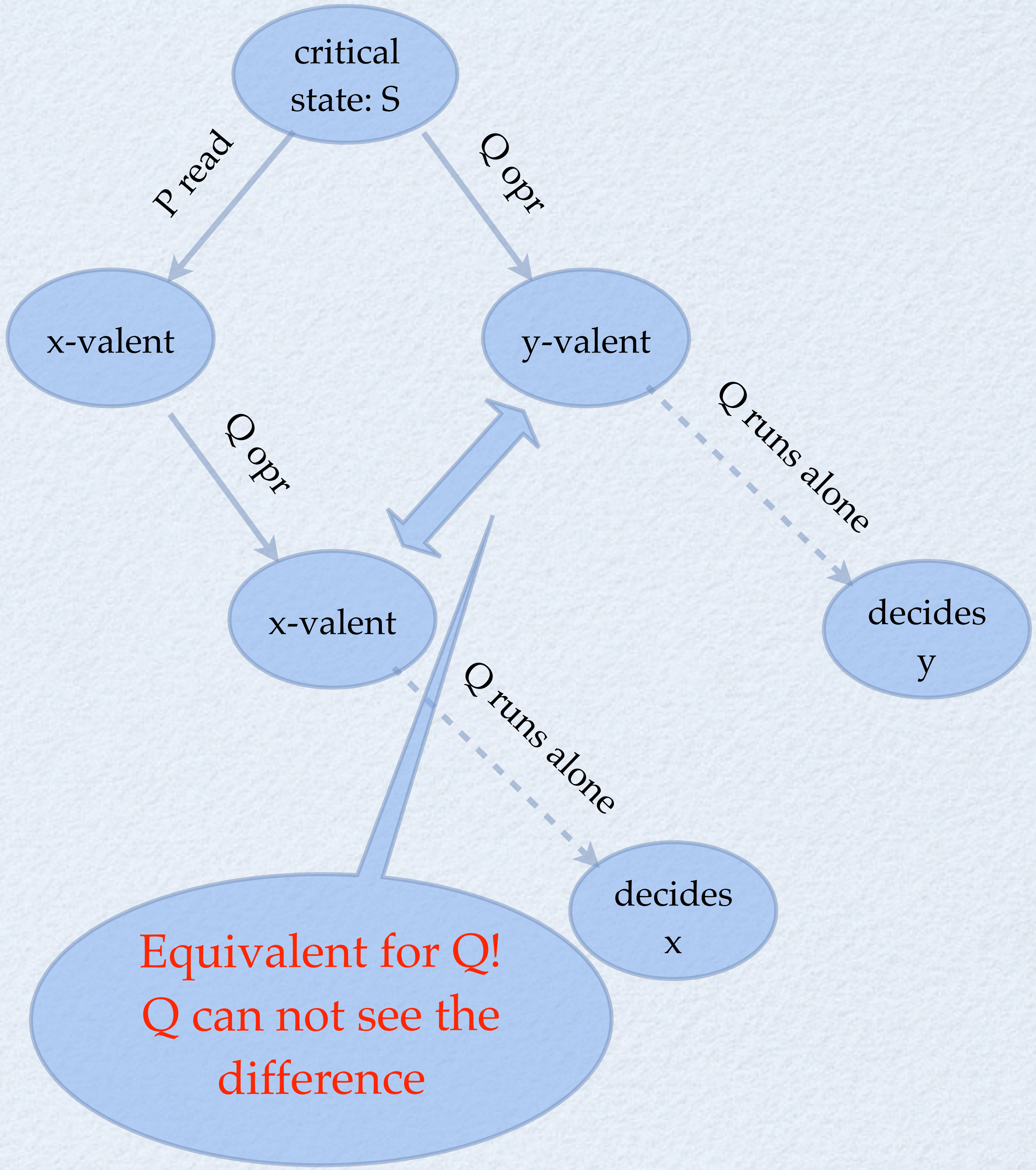
this argument is valid in every impossibility proof

Atomic Registers - 2/3

Two operations on the same register.

1. <u>one of the operations is read</u>

2. each of the operations is write

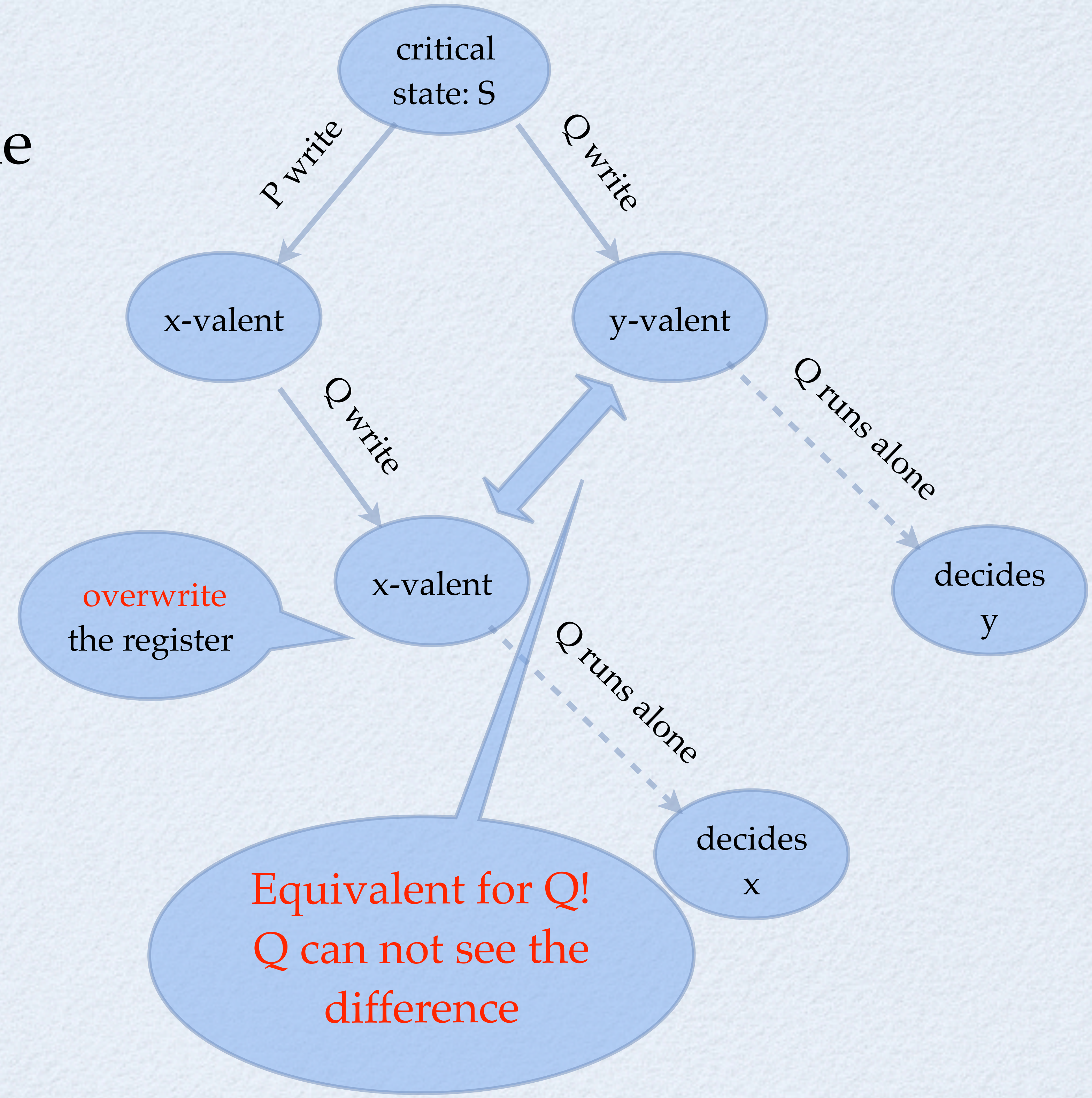


ATOMIC REGISTERS - 3/3

Two operations on the same register.

1. one of the operations is read

2. <u>each of the operations is write</u>



OUTLINE

- Motivation
- Wait-free object model
- Consensus problem
- Wait-free solutions to the consensus problem
- Impossibility proofs
- Universal construction

Universality Results

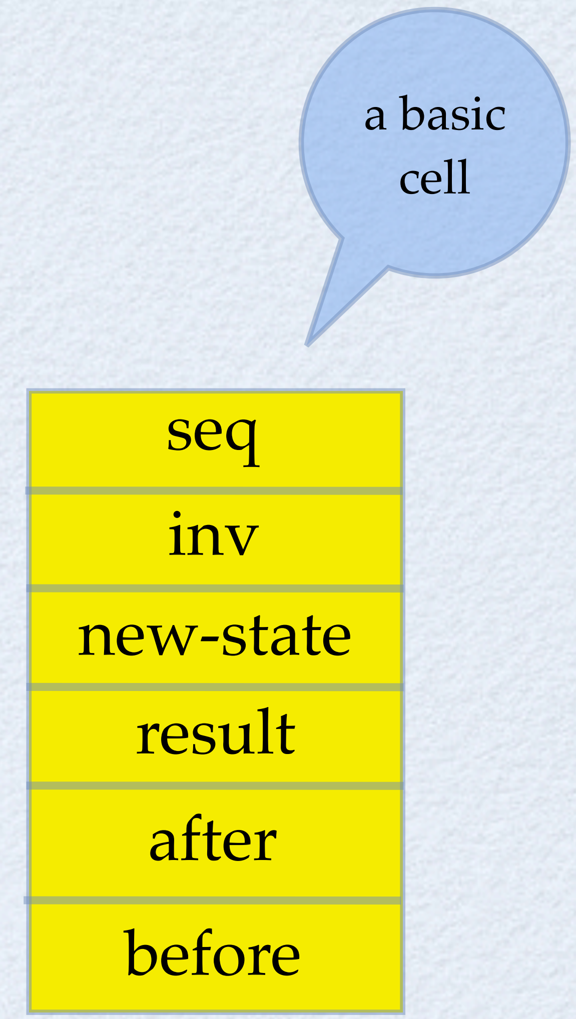
- Every object with consensus number n, can implement any other concurrent object within a system of n threads
- Consensus object
  - a consensus protocol with a register where the decision value is written
  - has a function decide(value: input). a thread calls decide to invoke the consensus protocol and get the decision value as result.
  - every object with consensus number n can implement the consensus object within a system of n threads.

Universality Results

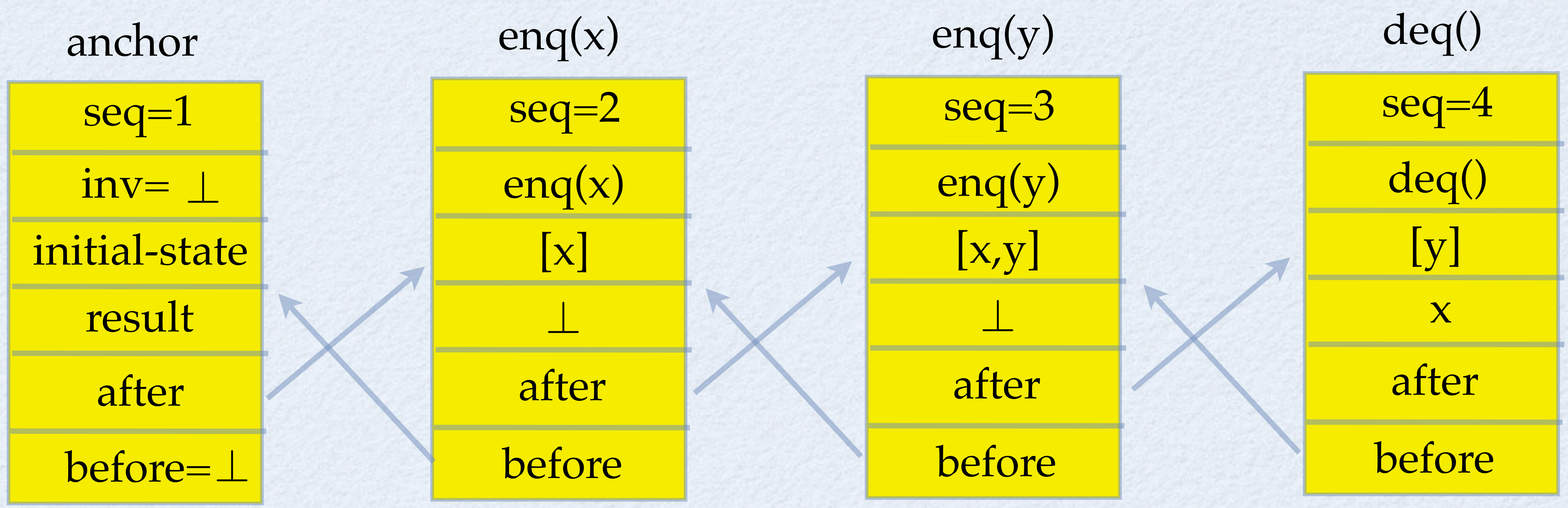
- Implement a concurrent object by consensus objects and atomic registers
- General idea: An execution of a concurrent object can be presented as a linked list of cells.

Representing a concurrent object

- The general idea: An execution of a concurrent object can be presented as a linked list of cells.
- A cell has the following fields
  - seq: sequence number indicating the order of the operations. Increase by 1 for successive cells
  - inv: invocation (operation name, argument name)
  - new-state: the new state of the object
  - new-result: the result value of the operation
  - before, after: point to the cell previous and next to it.

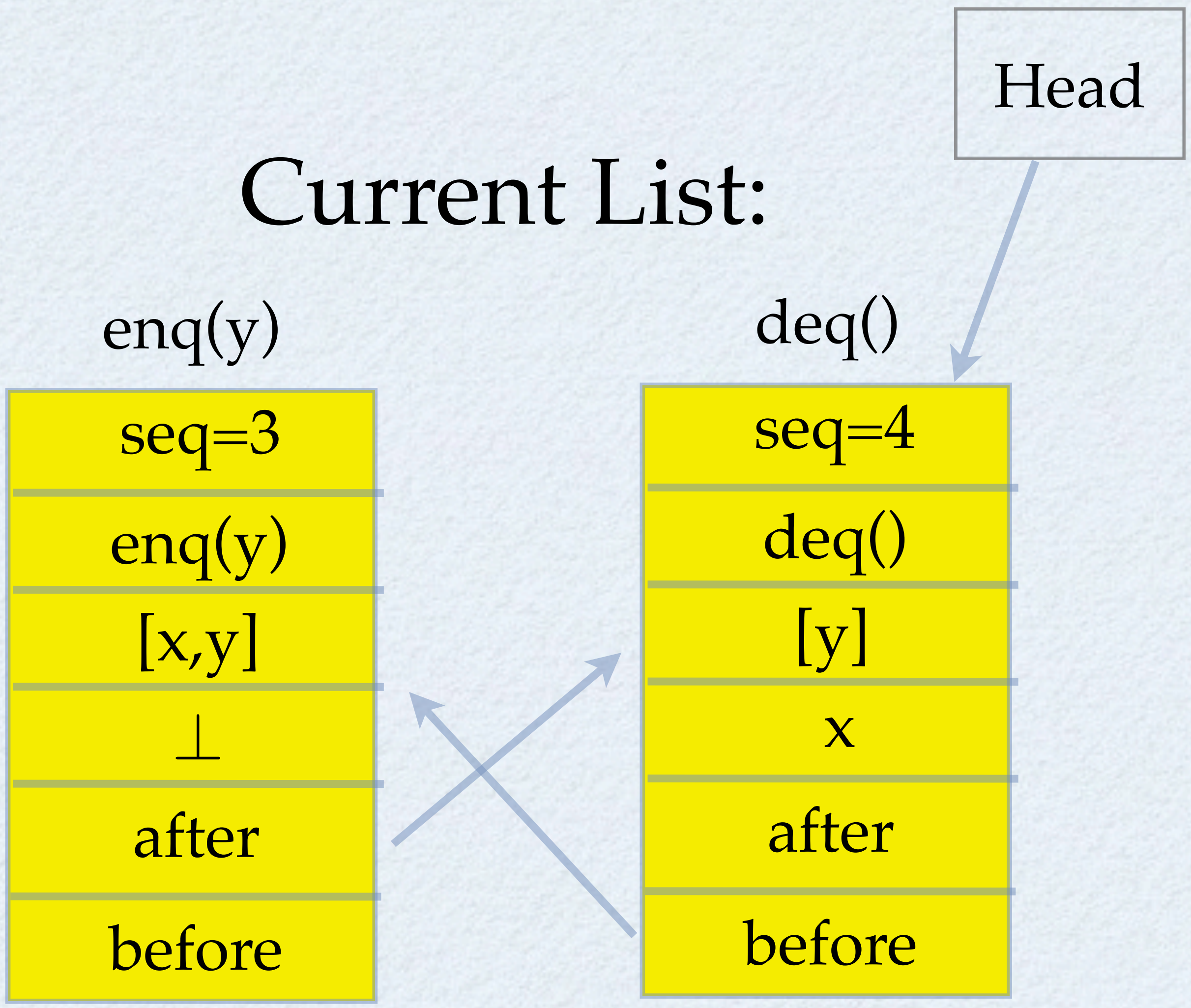


A Linked List of Cells

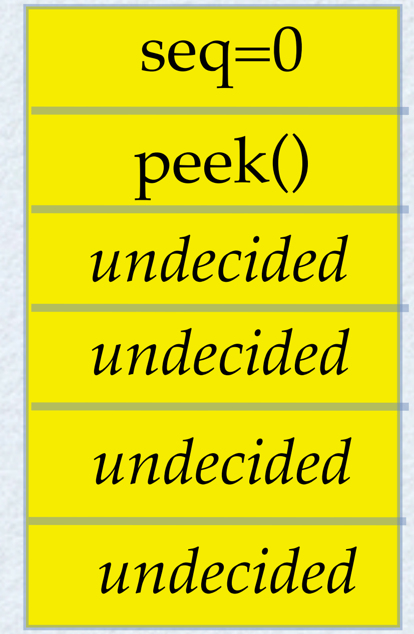


- When a thread invokes an operation, it creates a cell with operation information and sequence number 0.
- Maintains a linked list of cells

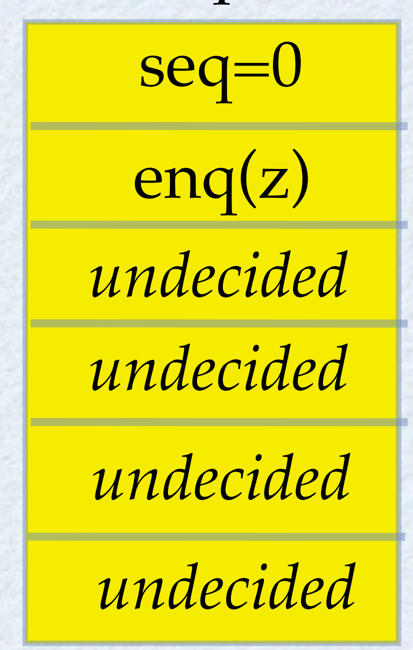
Current List:



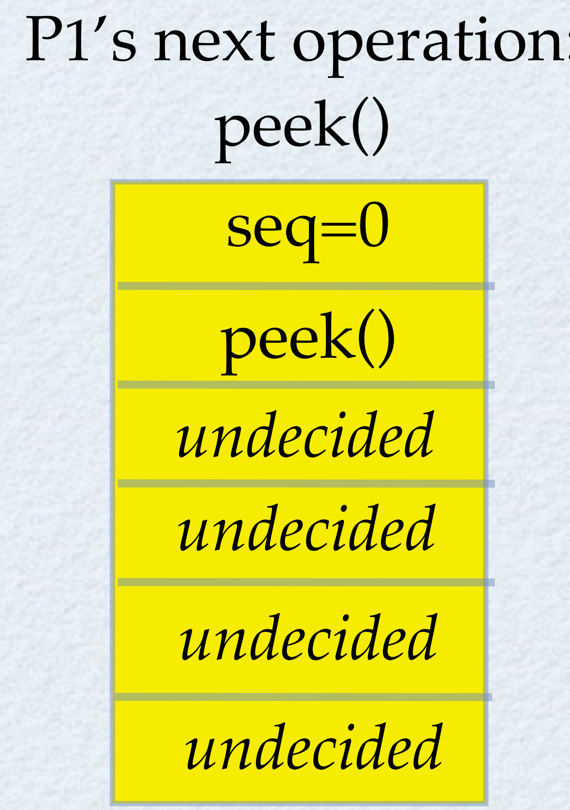
P1′s next operation: peek()

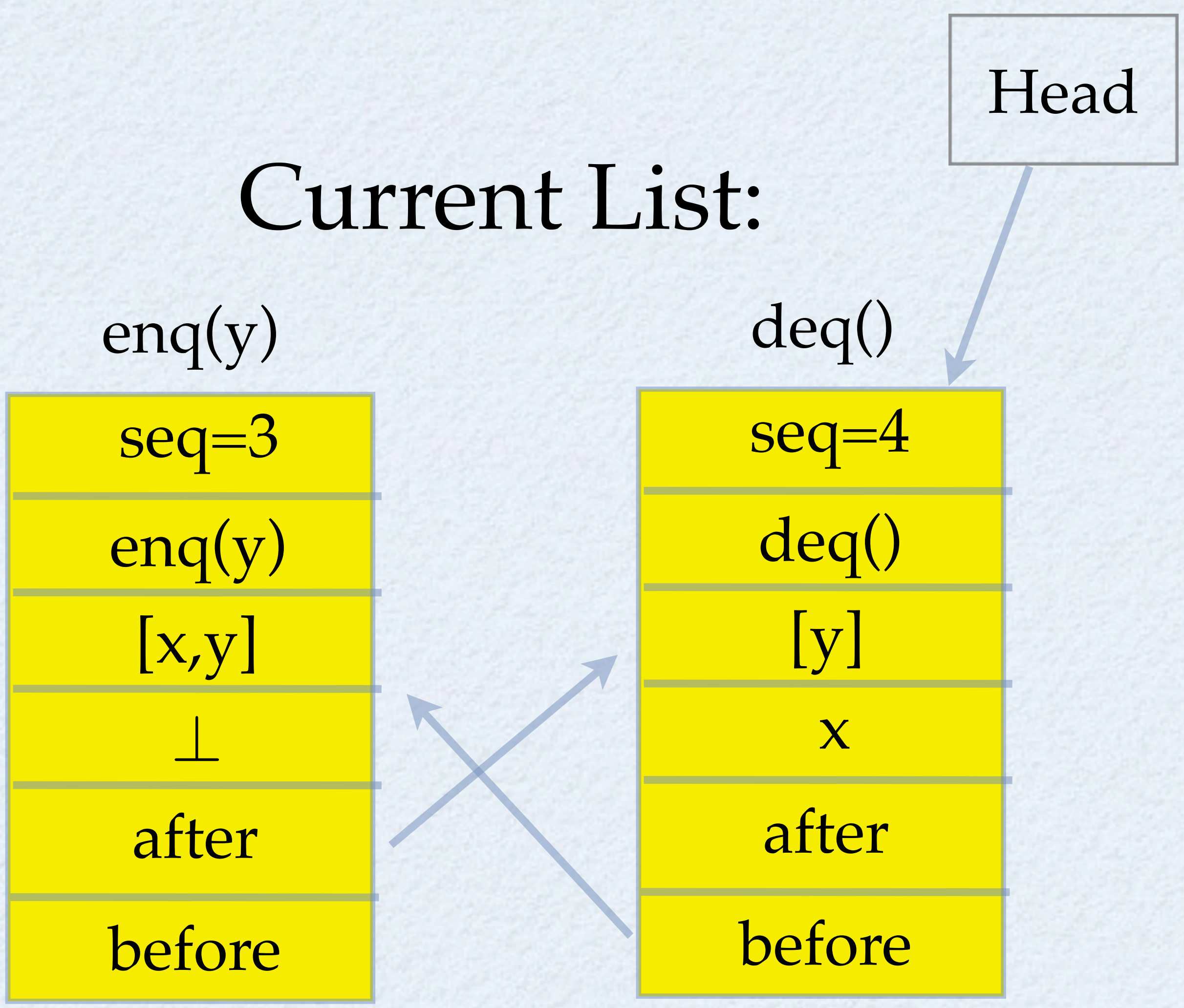


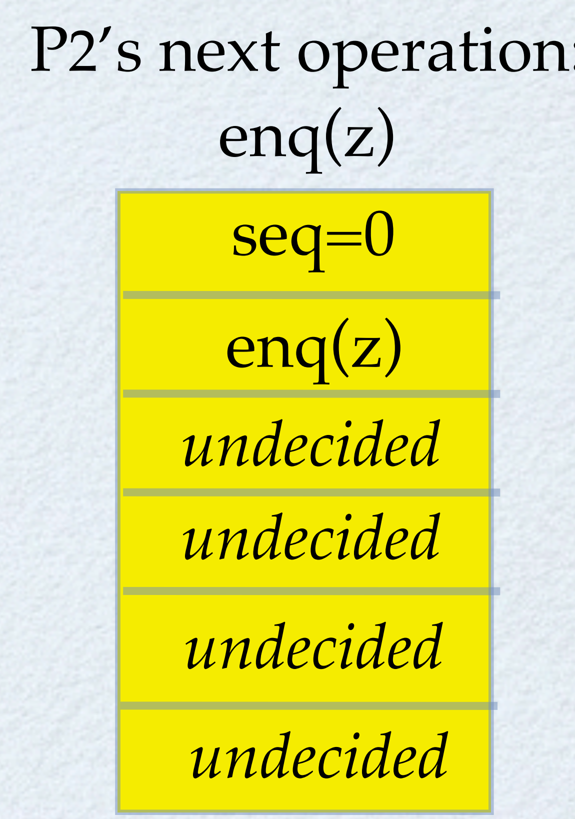
P2′s next operation: enq(z)



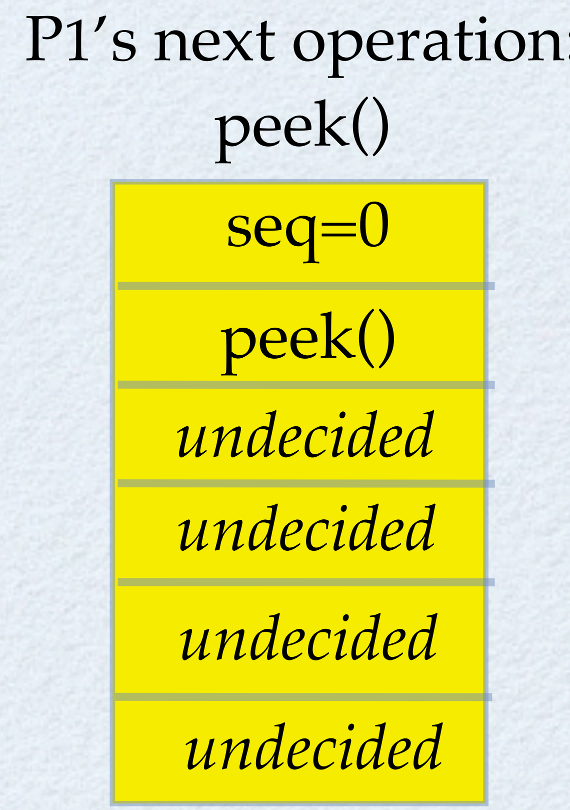
- We say a thread *threads a cell* if it adds the cell into the linked list.
- Naive idea: use a consensus protocol to decide which cell should be threaded next.

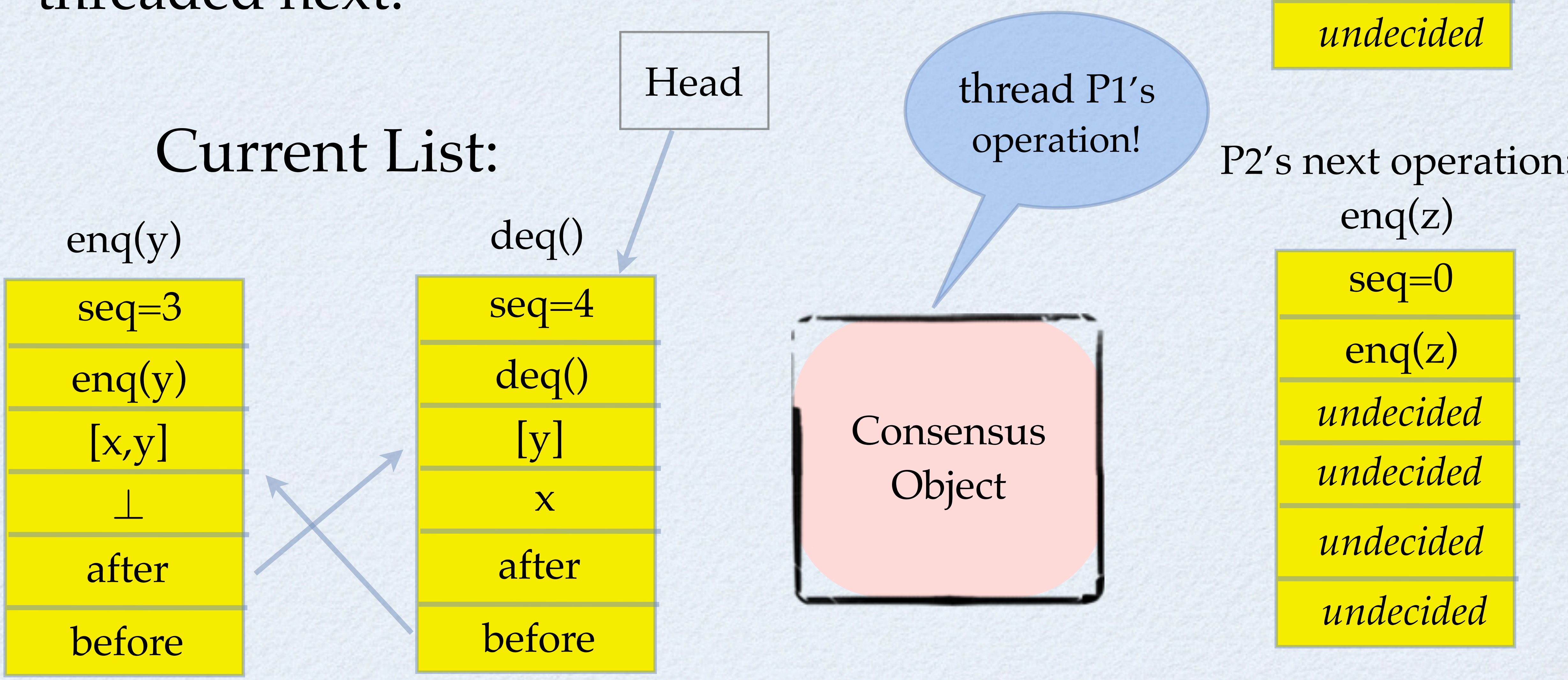




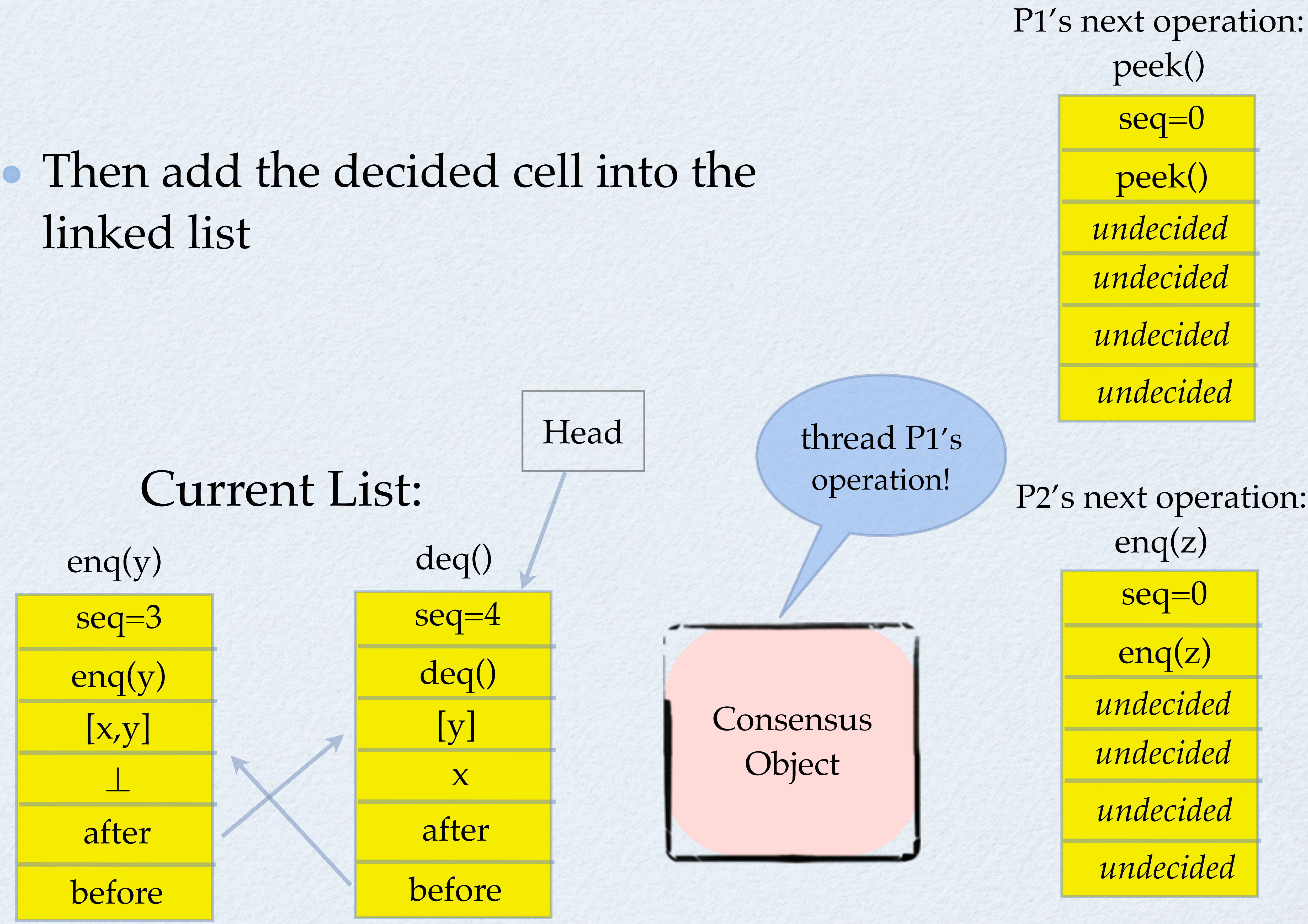


- We say a process *threads a cell* if it adds the cell into the linked list.
- Naive idea: use a consensus protocol to decide which cell should be threaded next.

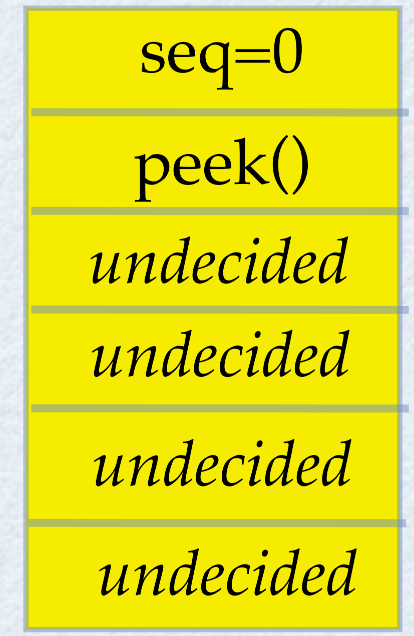




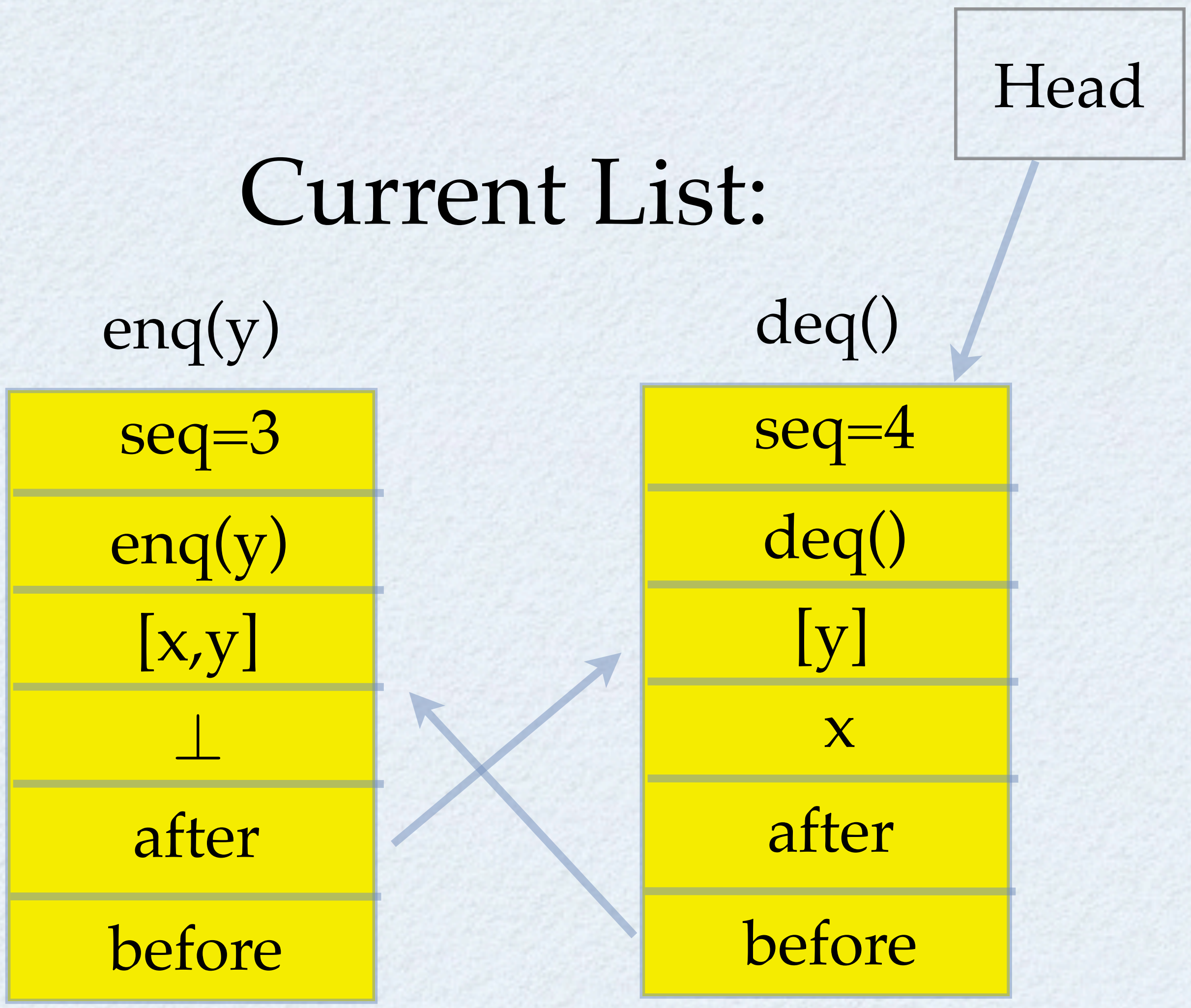
- Then add the decided cell into the linked list



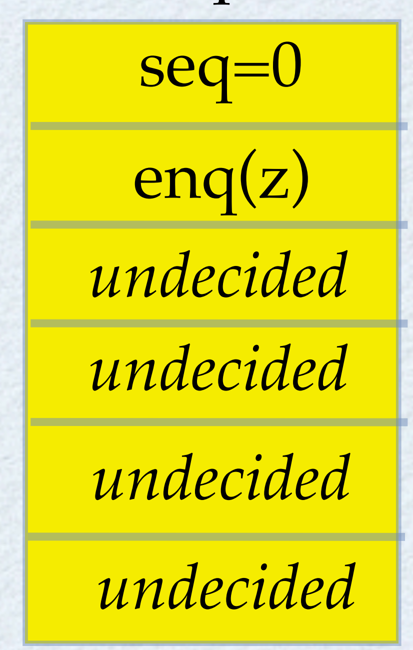
- Then add the decided cell into the linked list



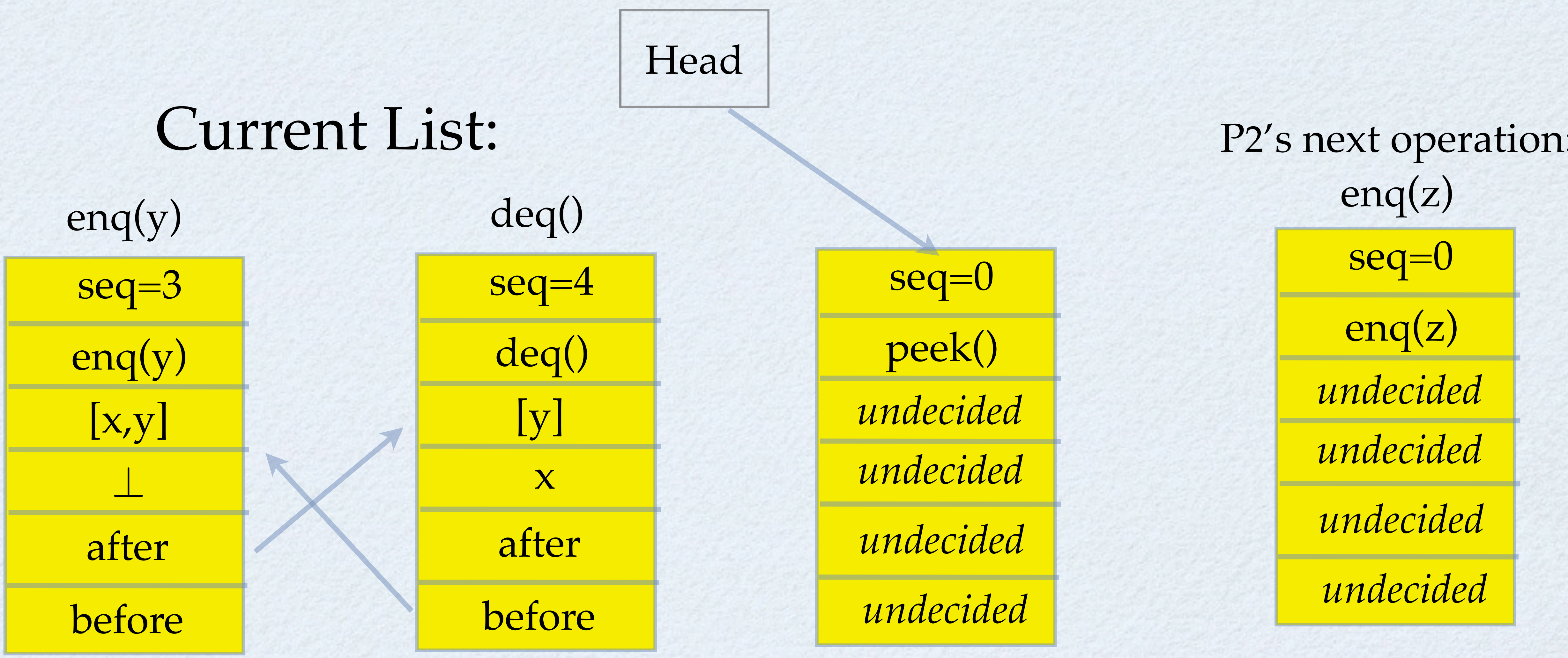
Current List:



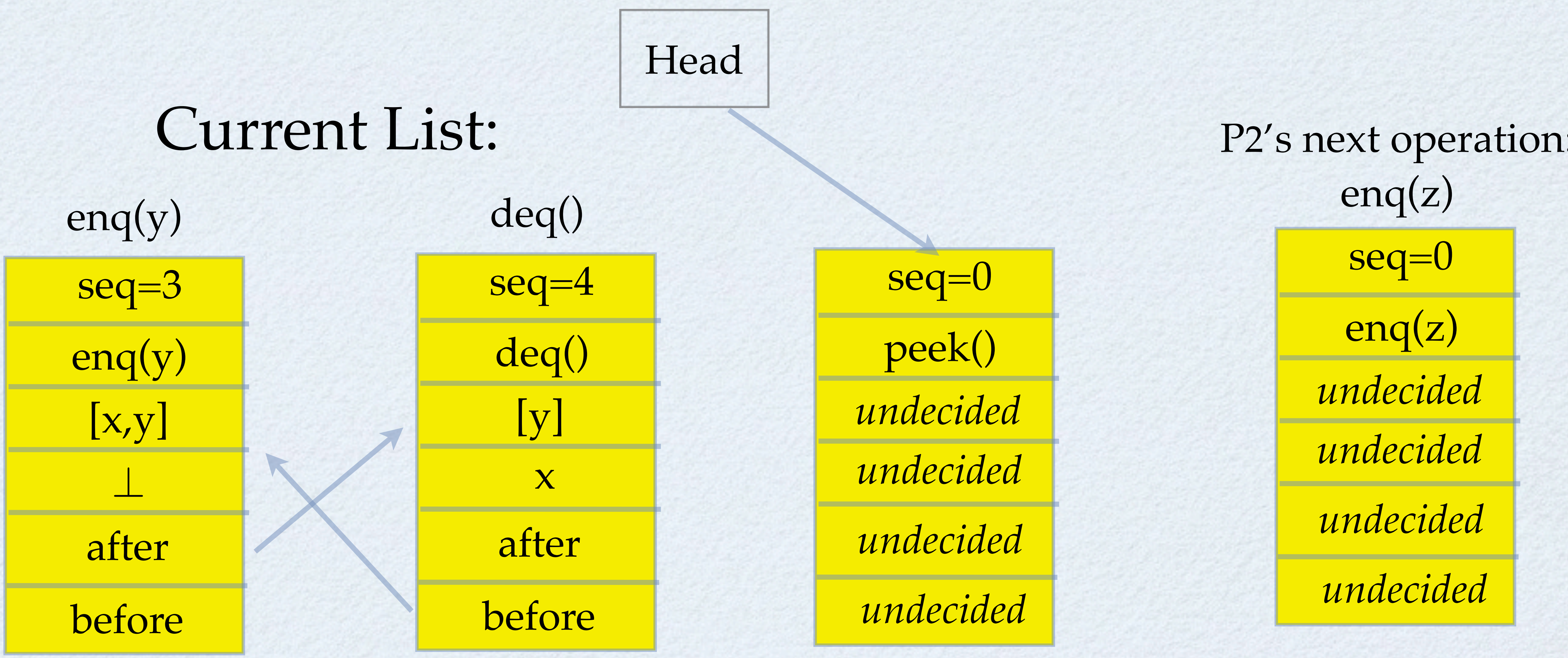
P2's next operation:
enq(z)



- Then add the decided cell into the linked list

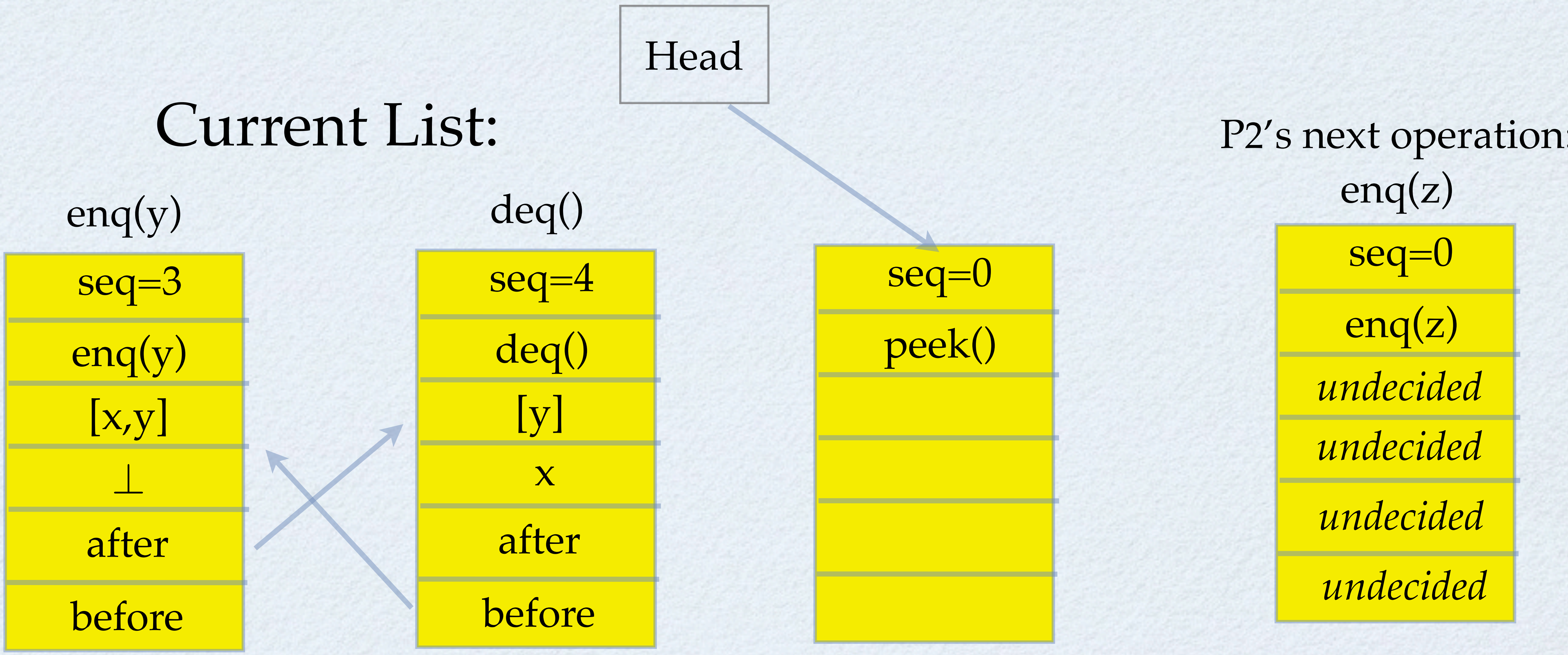


- Then add the decided cell into the linked list
- update fields of the cell



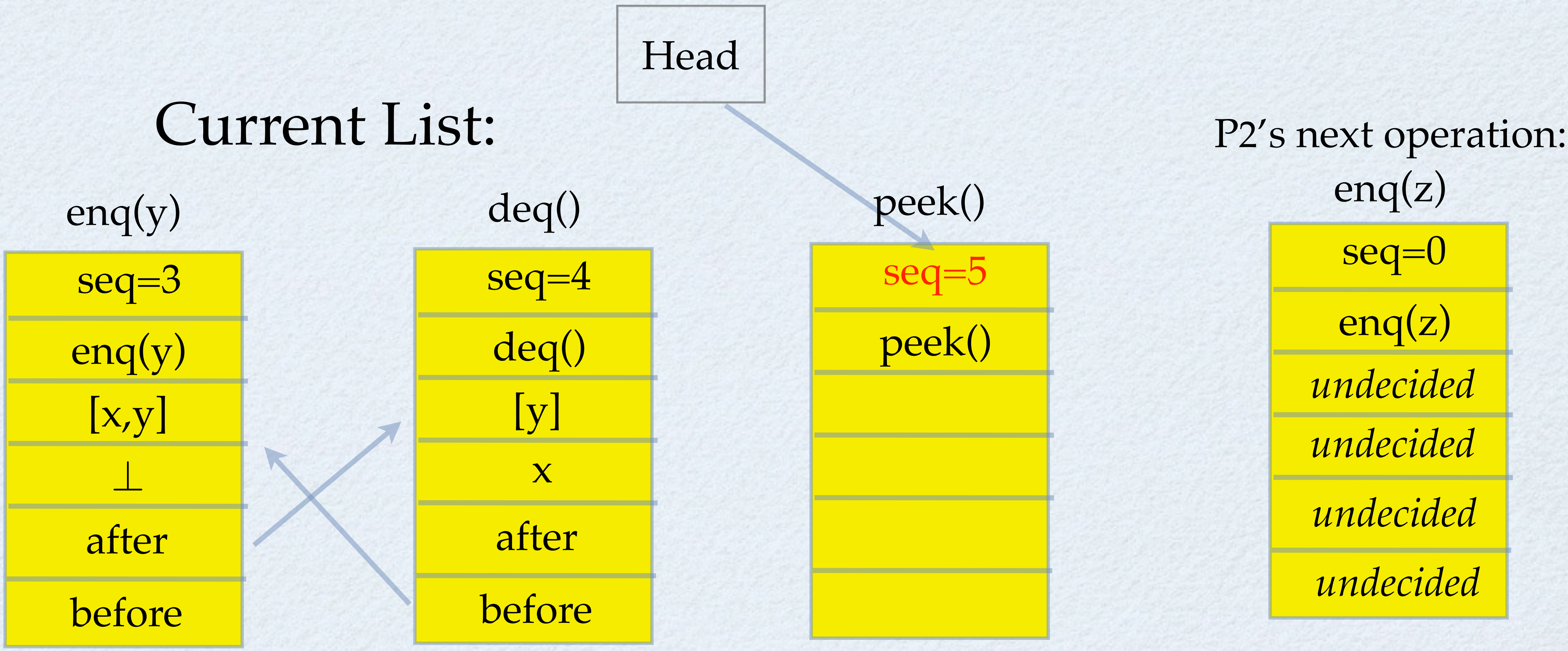
- Then add the decided cell into the linked list

- update fields of the cell

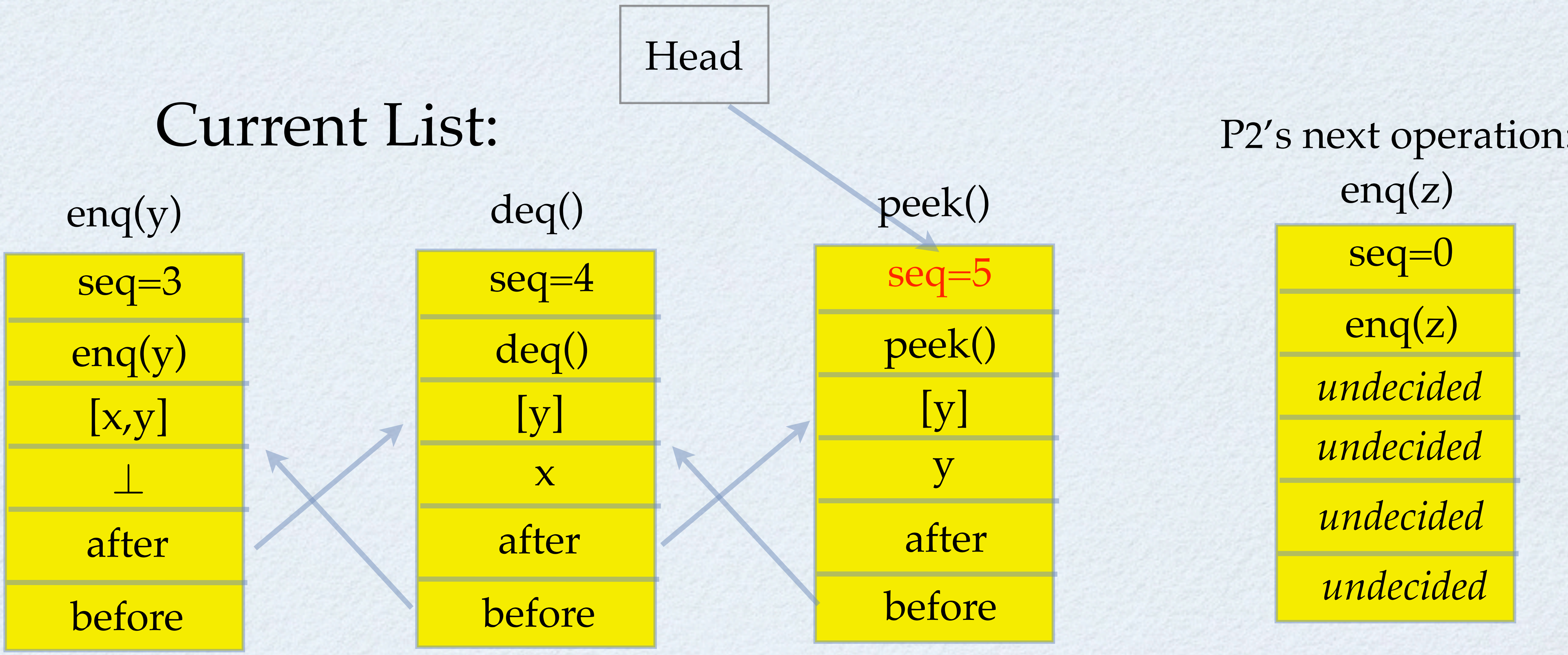


- Then add the decided cell into the linked list

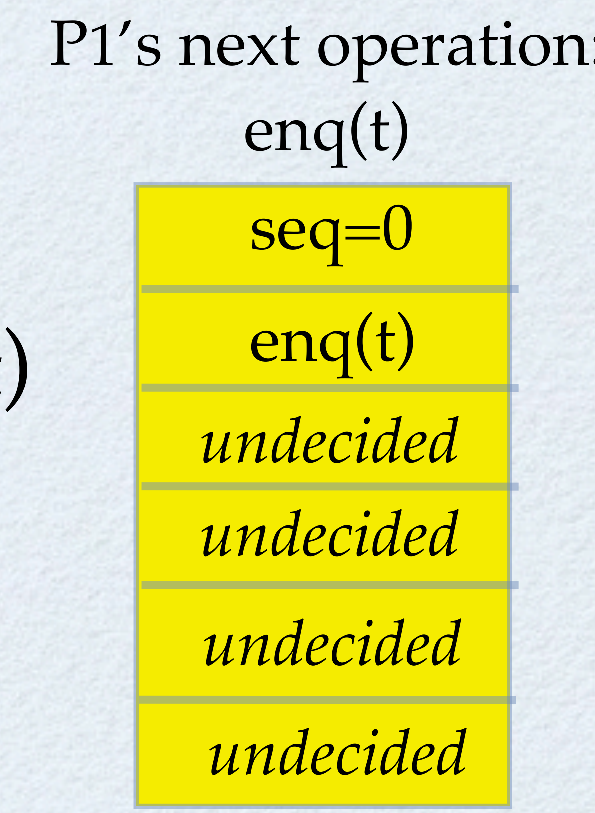
- update fields of the cell

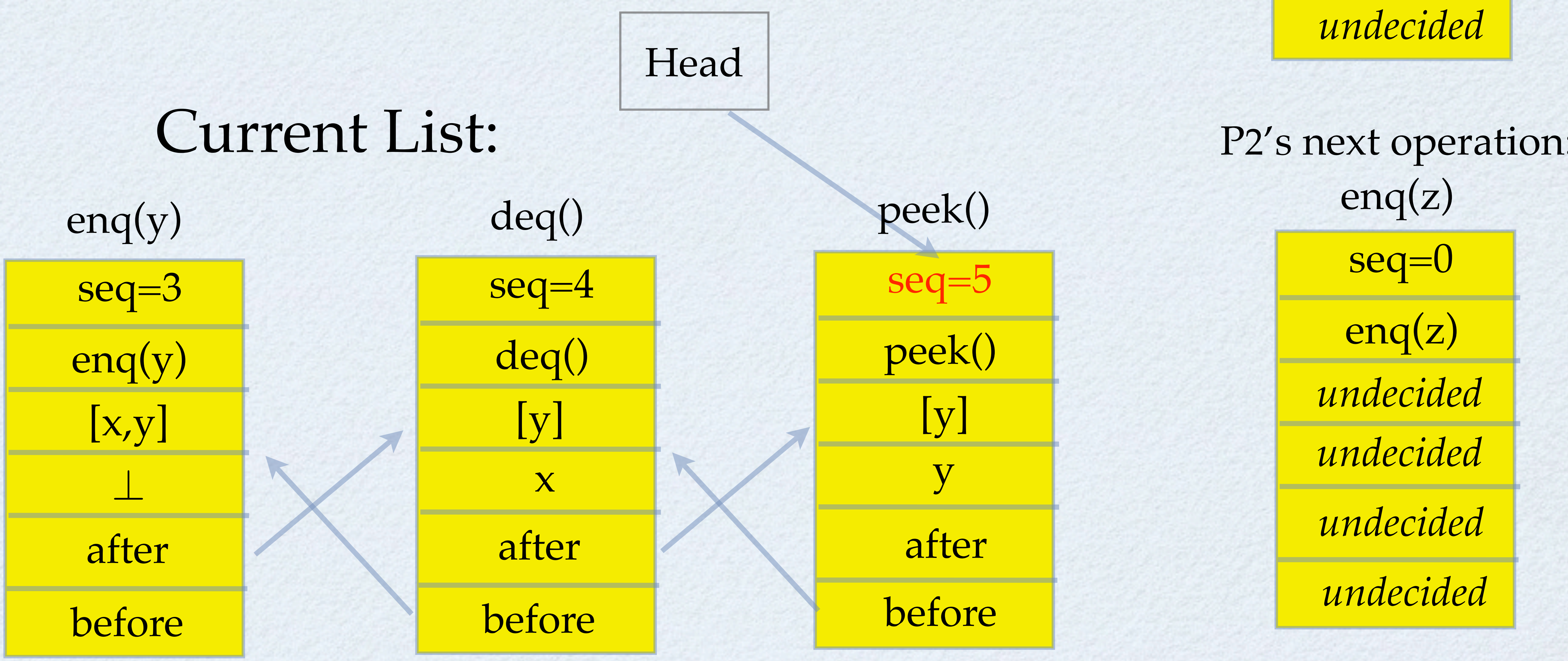


- Then add the decided cell into the linked list
- update fields of the cell



P1 announces another cell for operation enq(t)



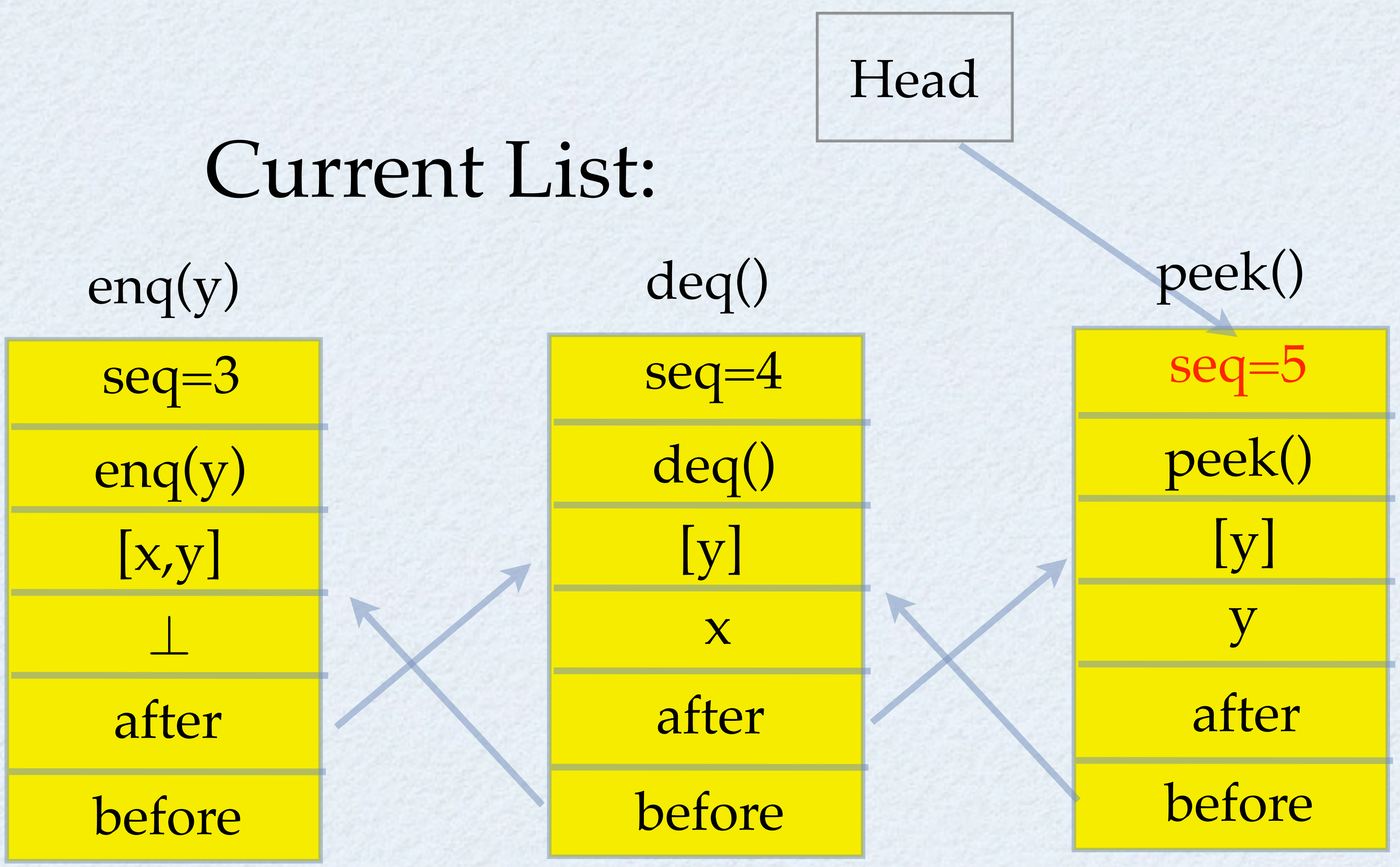


P1's next operation:
enq(t)

seq=0
enq(t)
undecided
undecided
undecided
undecided

P1 announces another cell for operation enq(t)

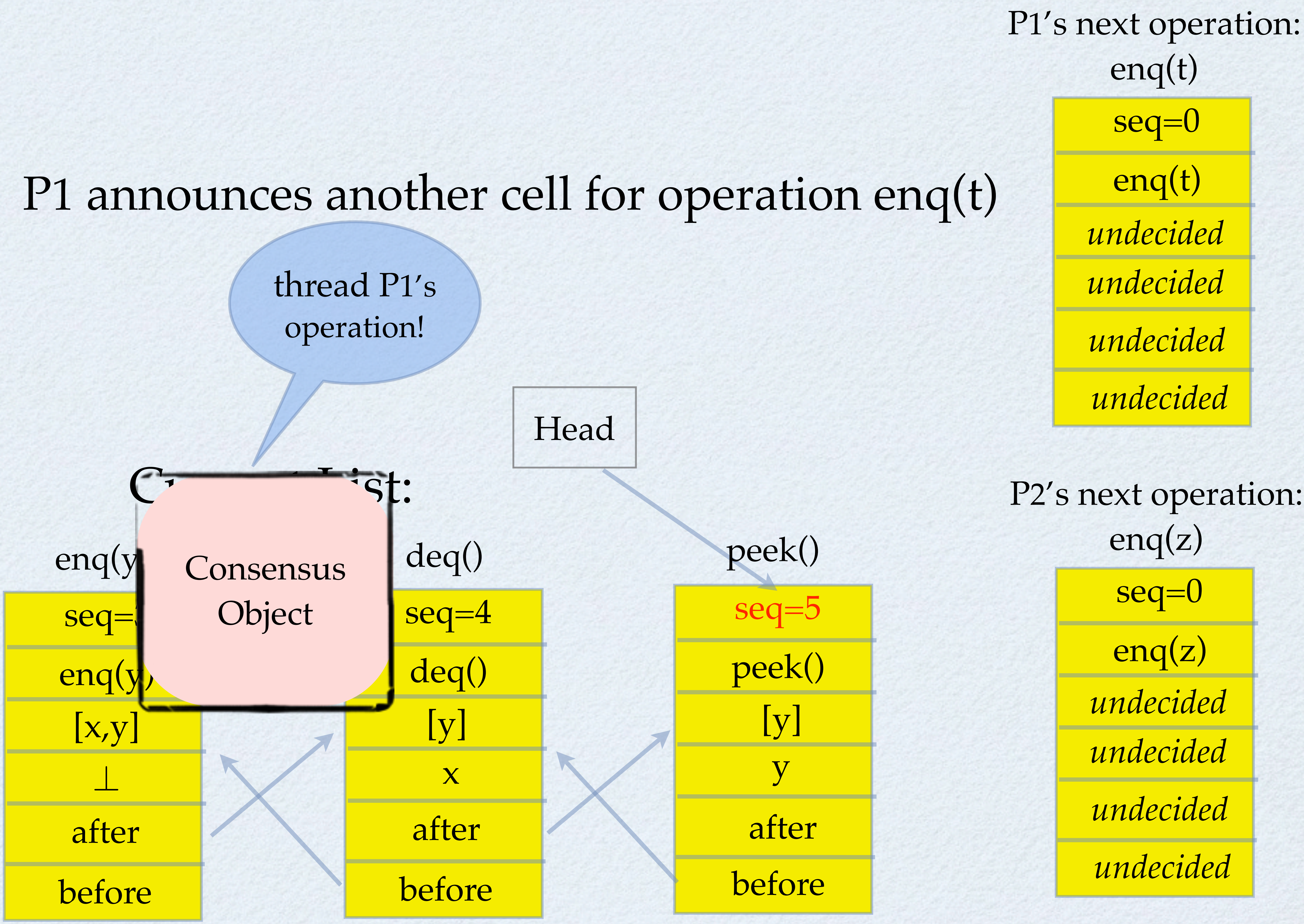
Current List:



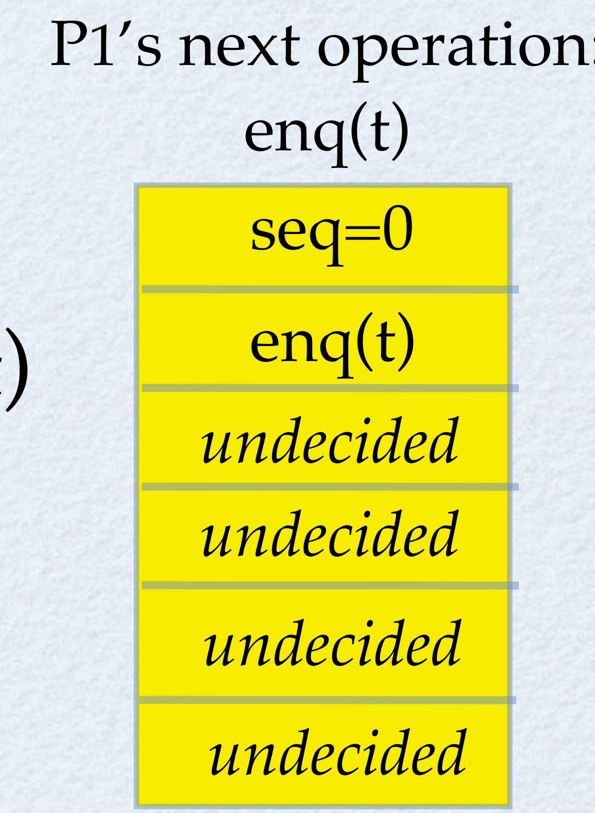
P2's next operation:
enq(z)

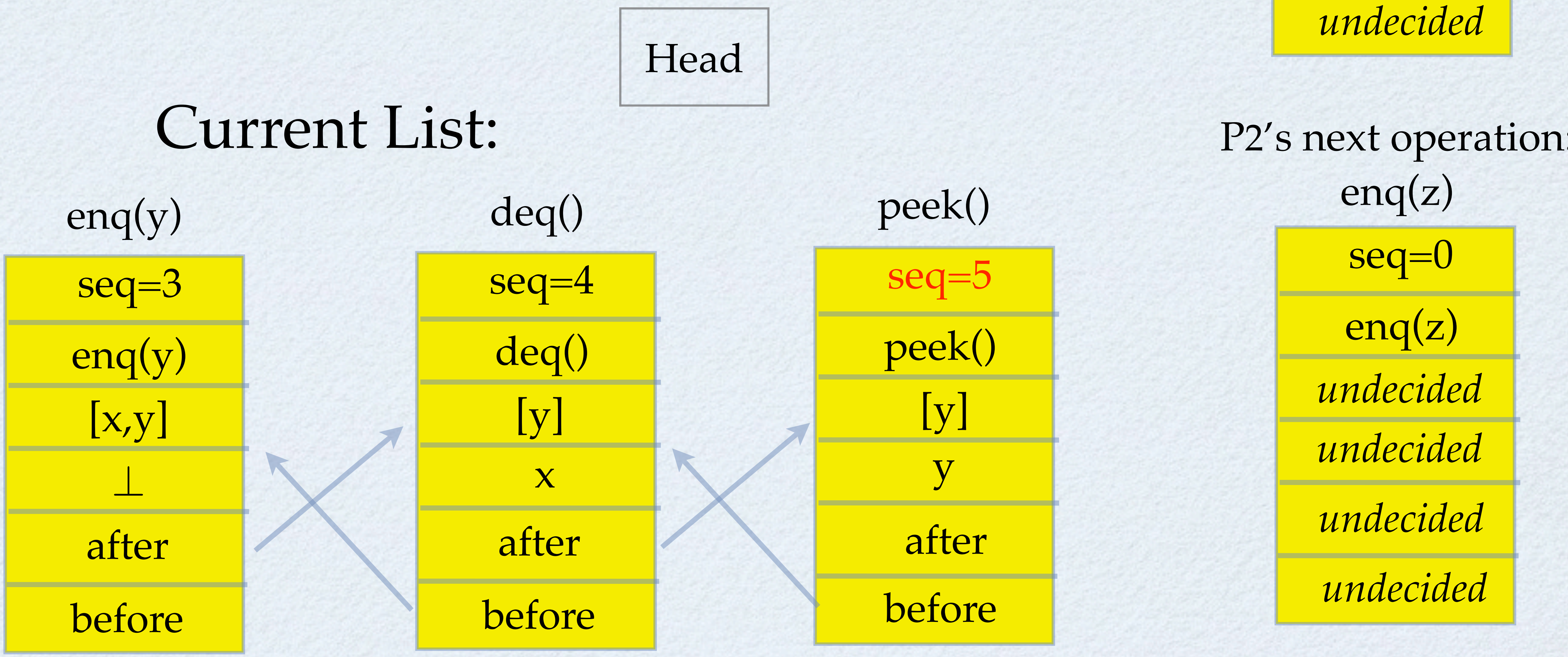
seq=0
enq(z)
undecided
undecided
undecided
undecided

P1 announces another cell for operation enq(t)

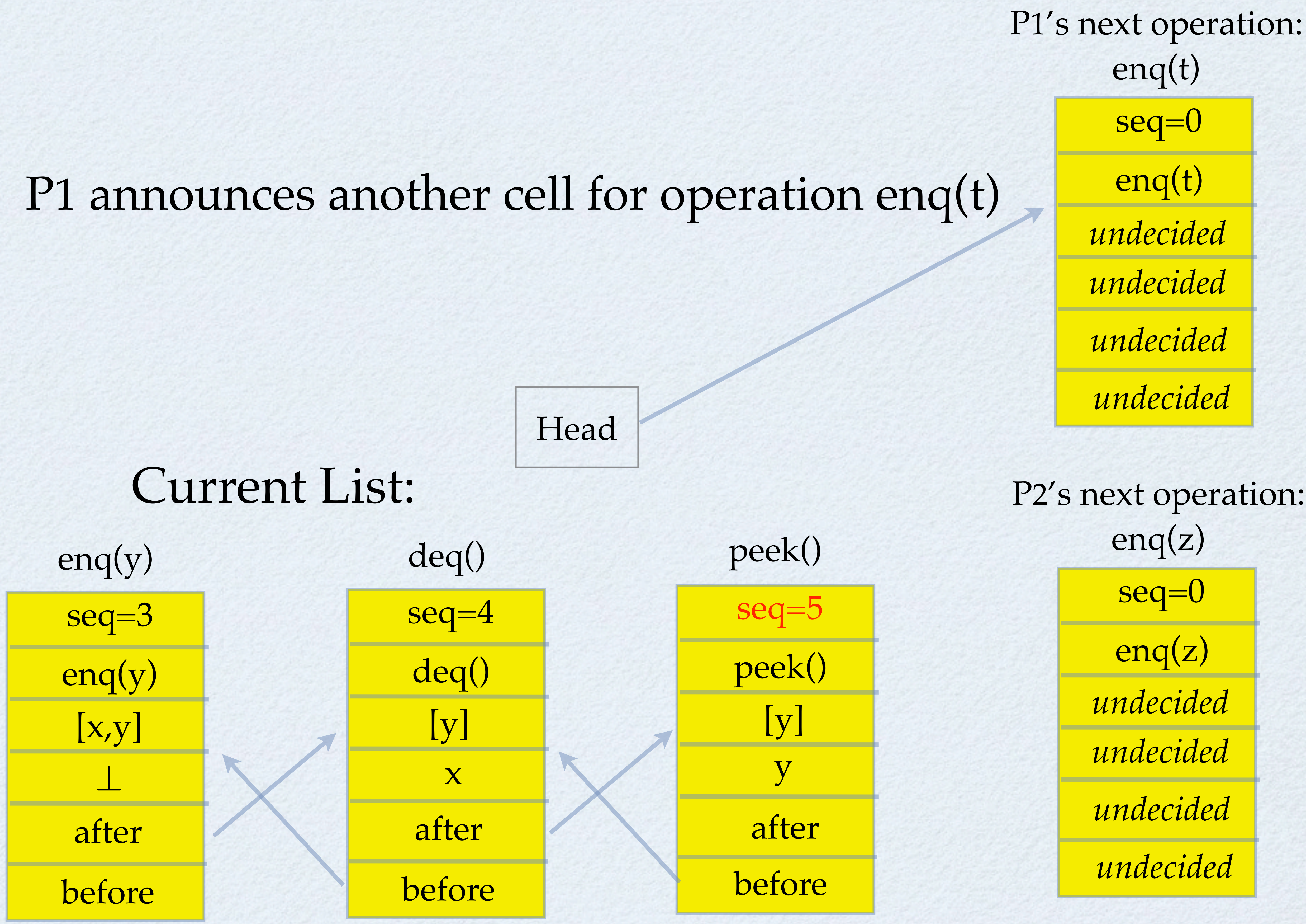


P1 announces another cell for operation enq(t)

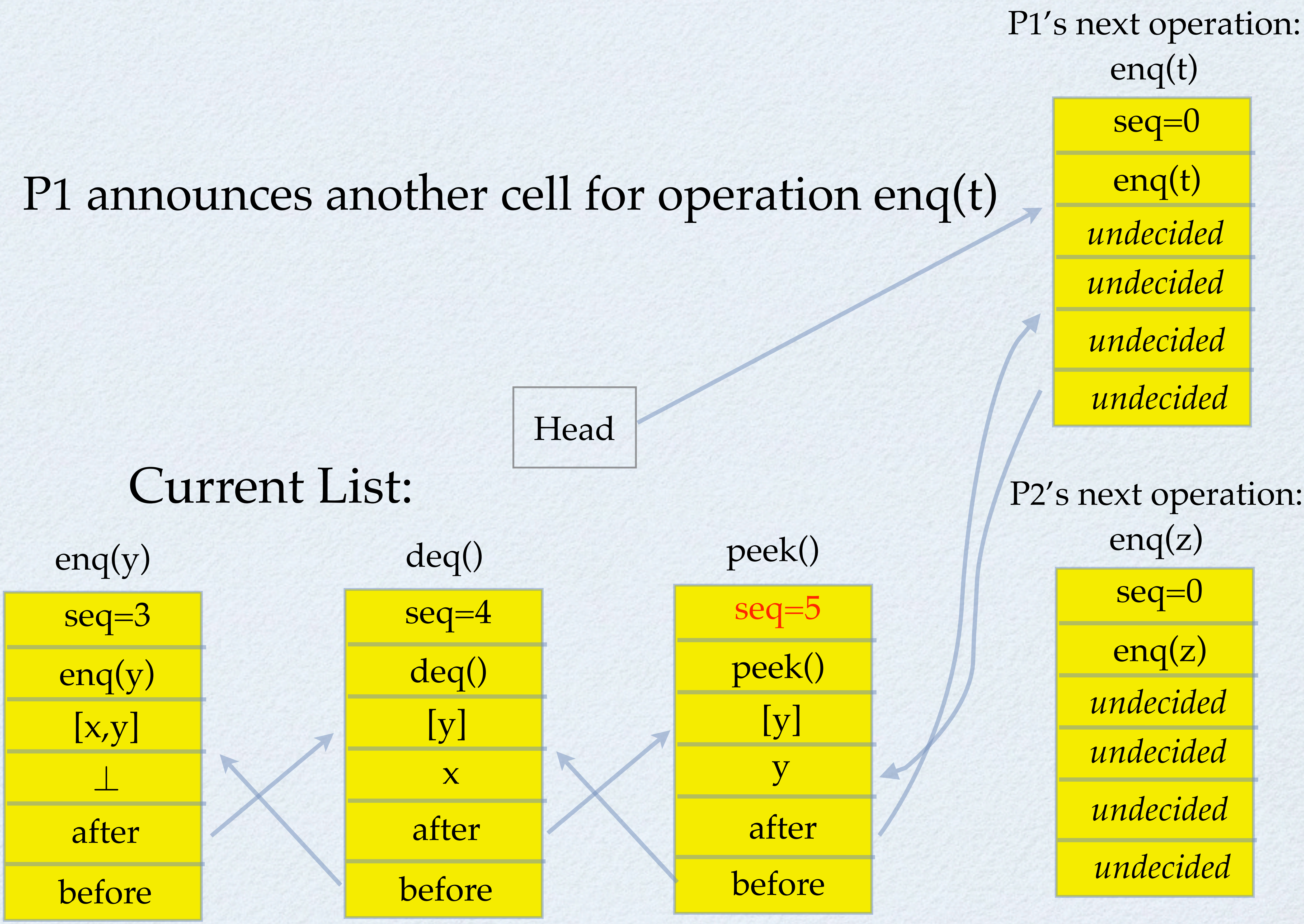




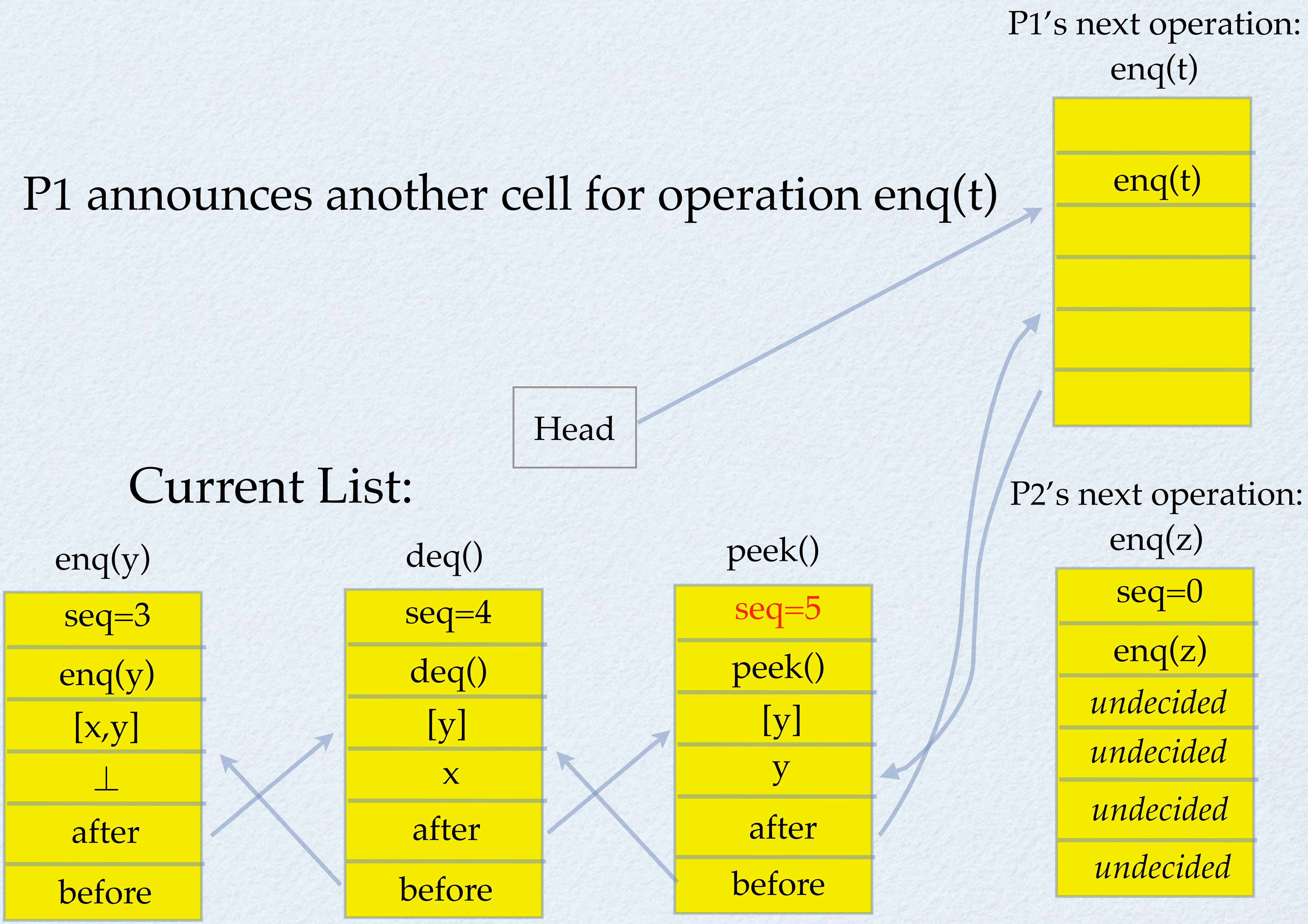
P1 announces another cell for operation enq(t)

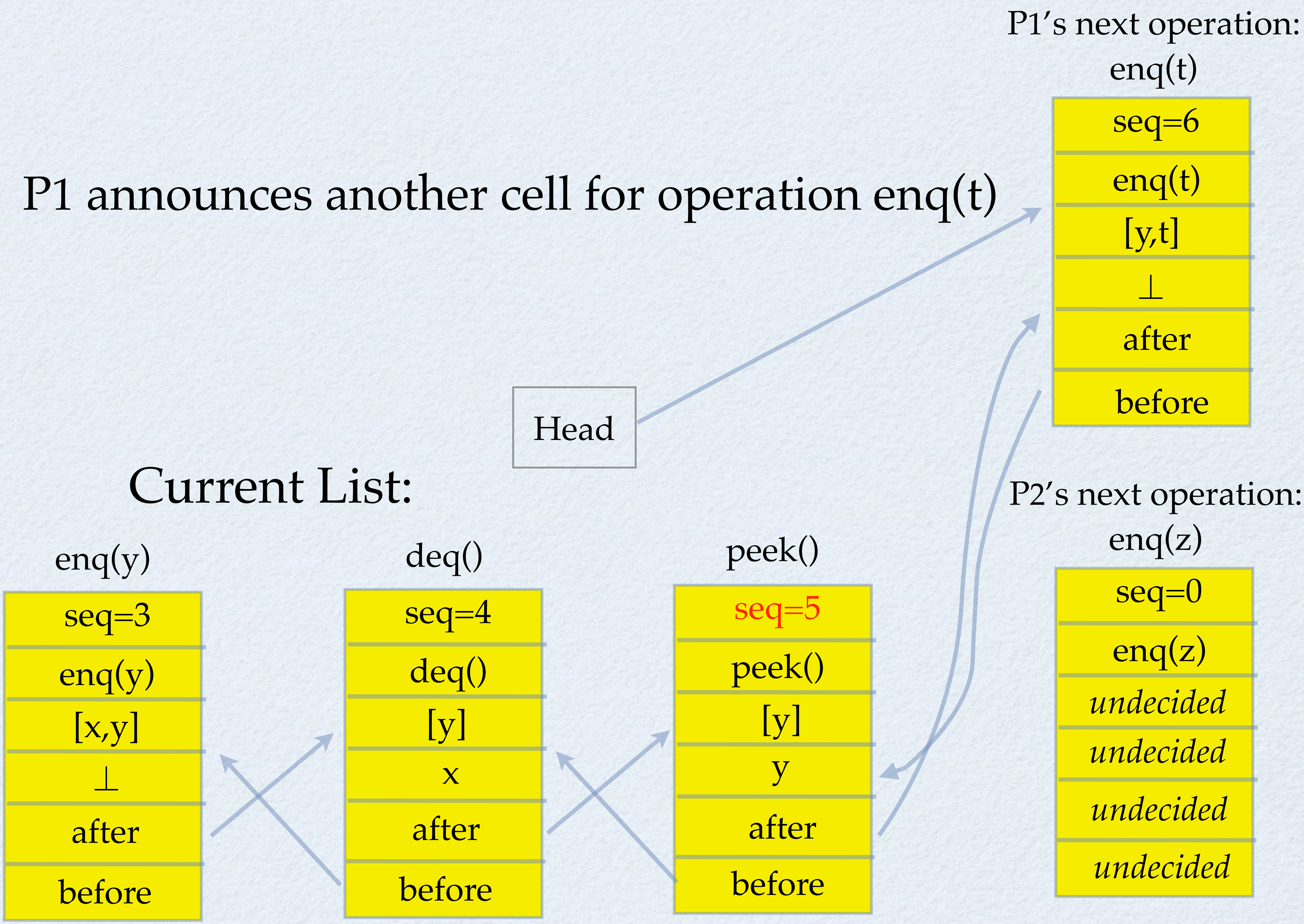


P1 announces another cell for operation enq(t)



P1 announces another cell for operation enq(t)

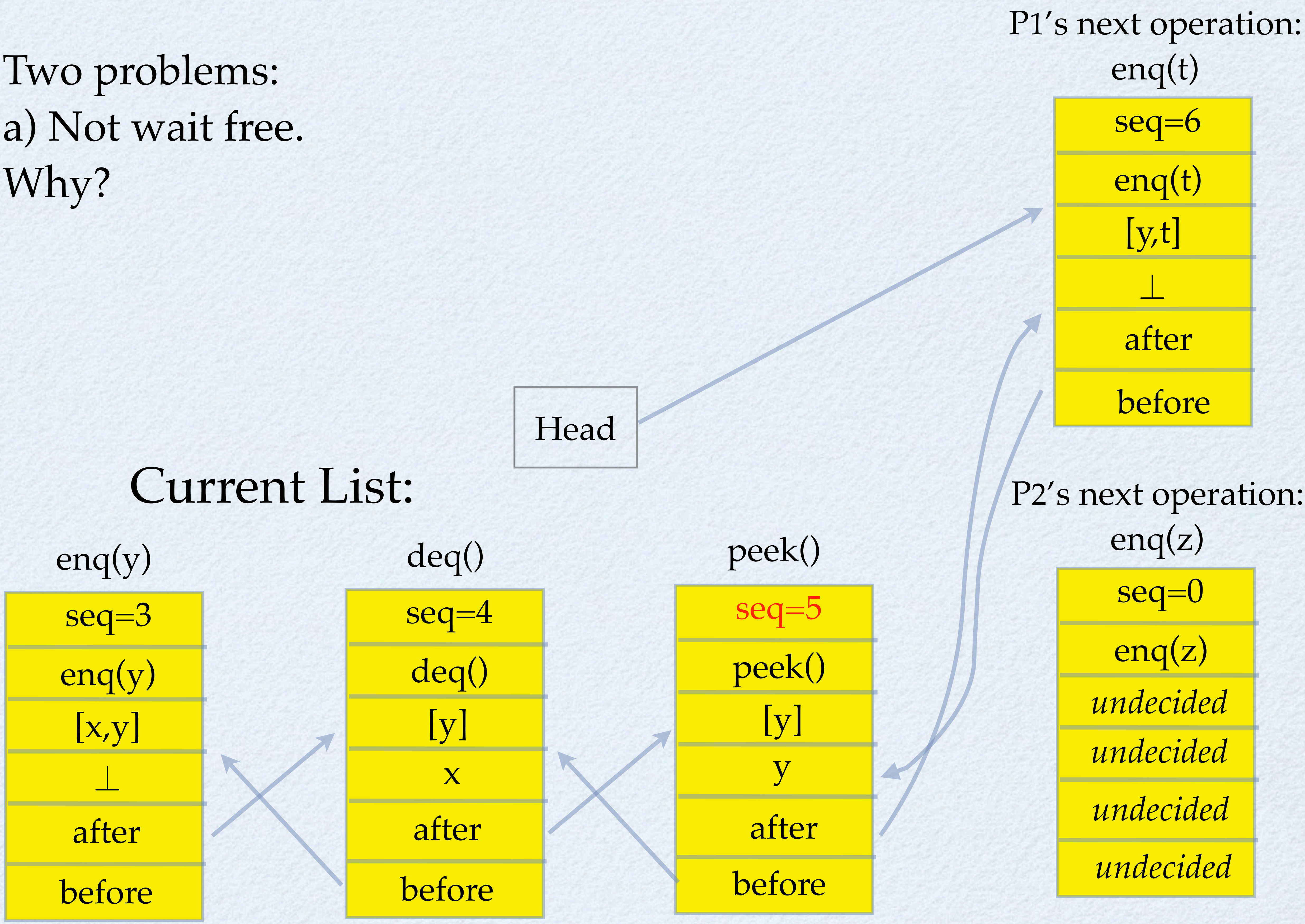



P1 announces another cell for operation enq(t)
P1's next operation:
enq(t)
seq=6
enq(t)
[y,t]
⊥
after
before
Head
Current List:
P2's next operation:
enq(z)
seq=0
enq(z)
undecided
undecided
undecided
undecided
enq(y)
seq=3
enq(y)
[x,y]
⊥
after
before
deq()
seq=4
deq()
[y]
x
after
before
peek()
seq=5
peek()
[y]
y
after
before

Two problems:

a) Not wait free.

Why?



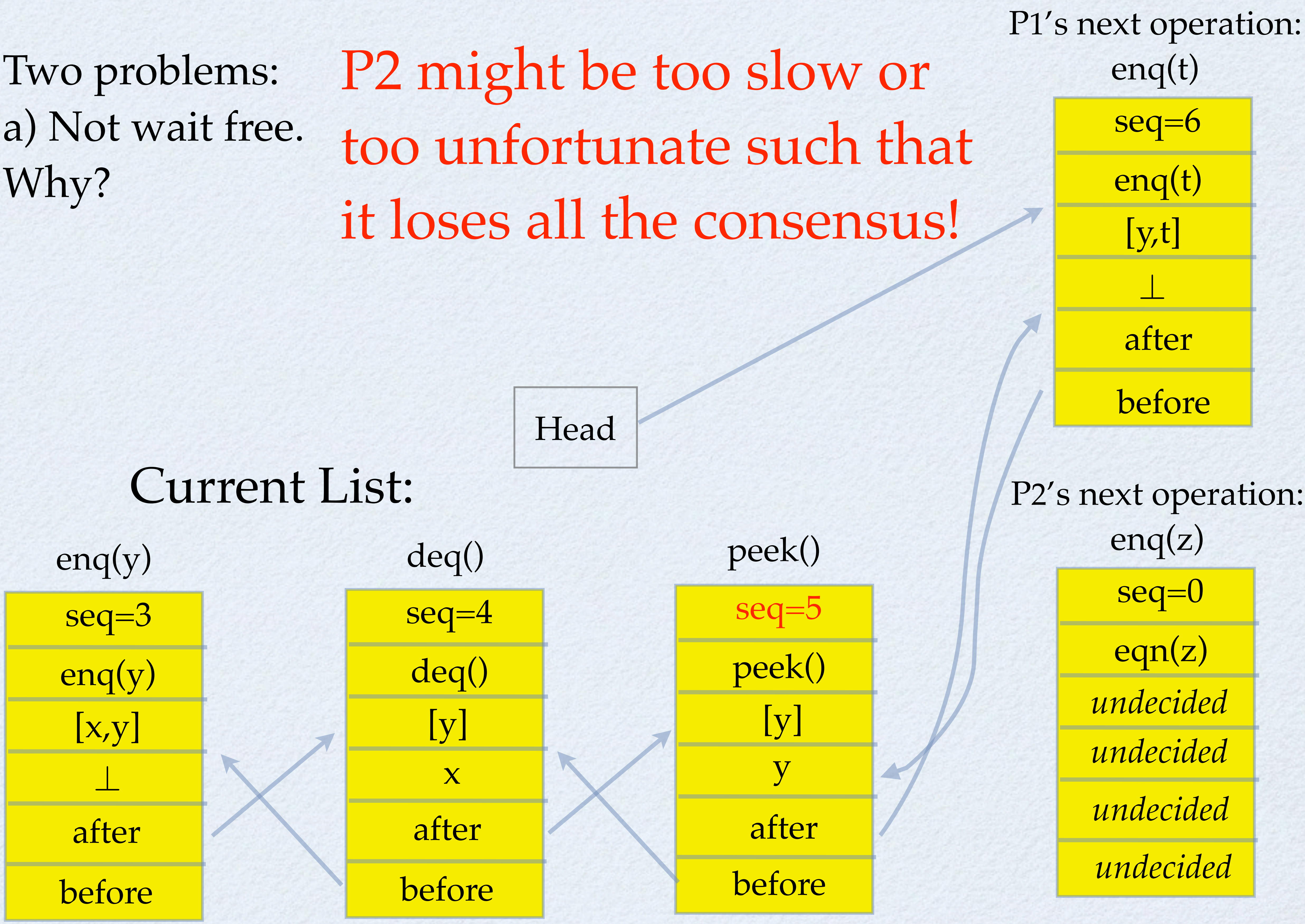
Two problems:

a) Not wait free.

Why?

P2 might be too slow or too unfortunate such that it loses all the consensus!

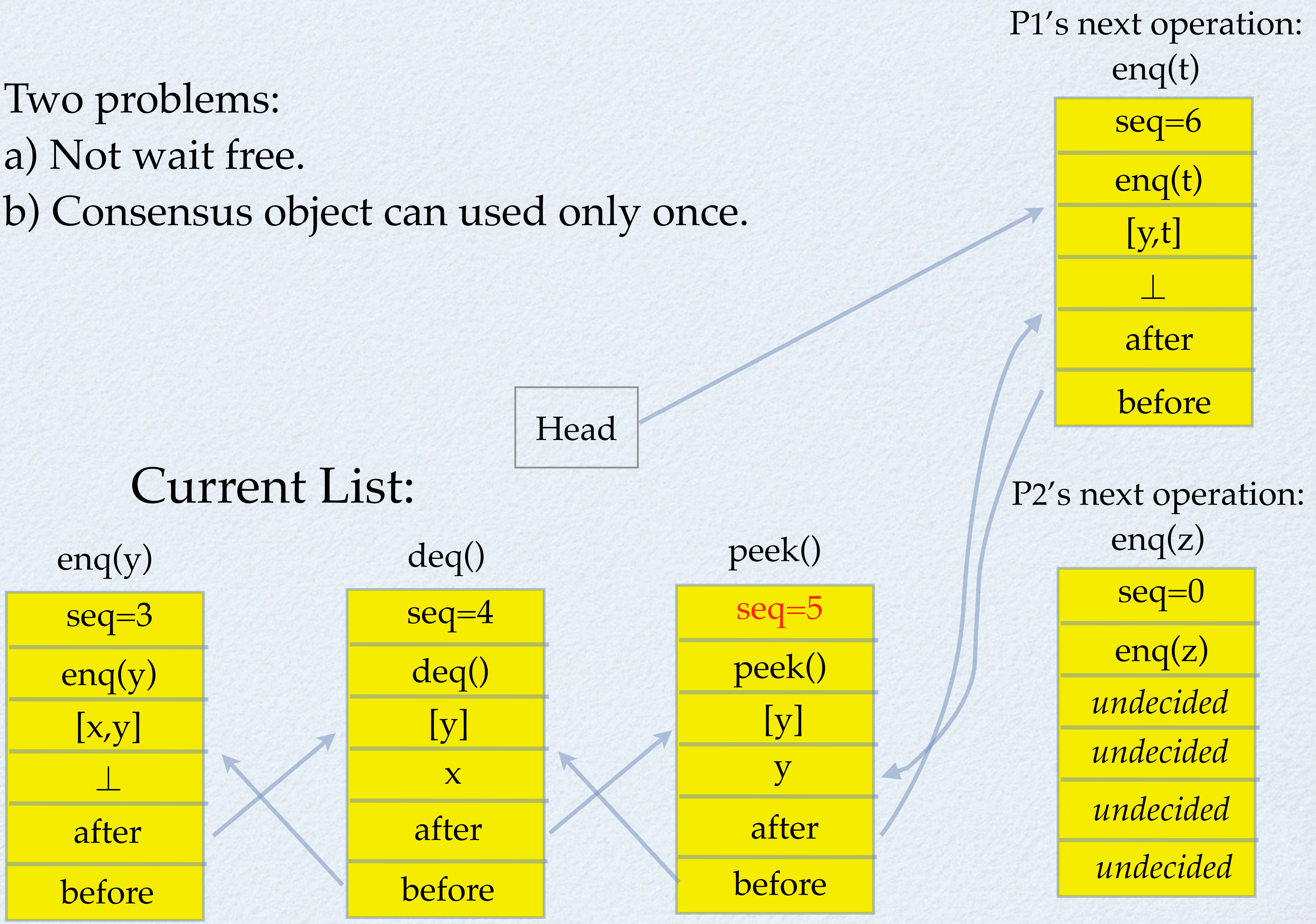
Current List:



Two problems:

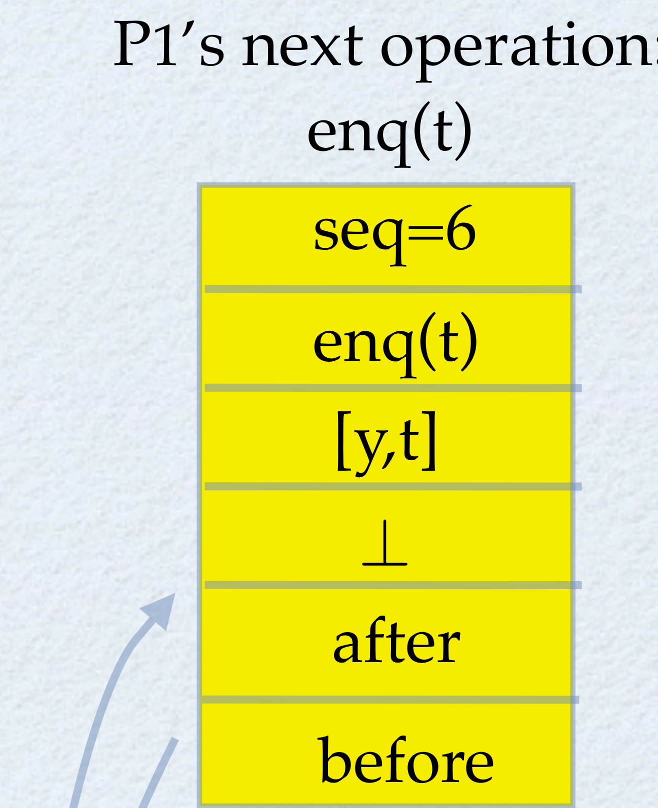
a) Not wait free.

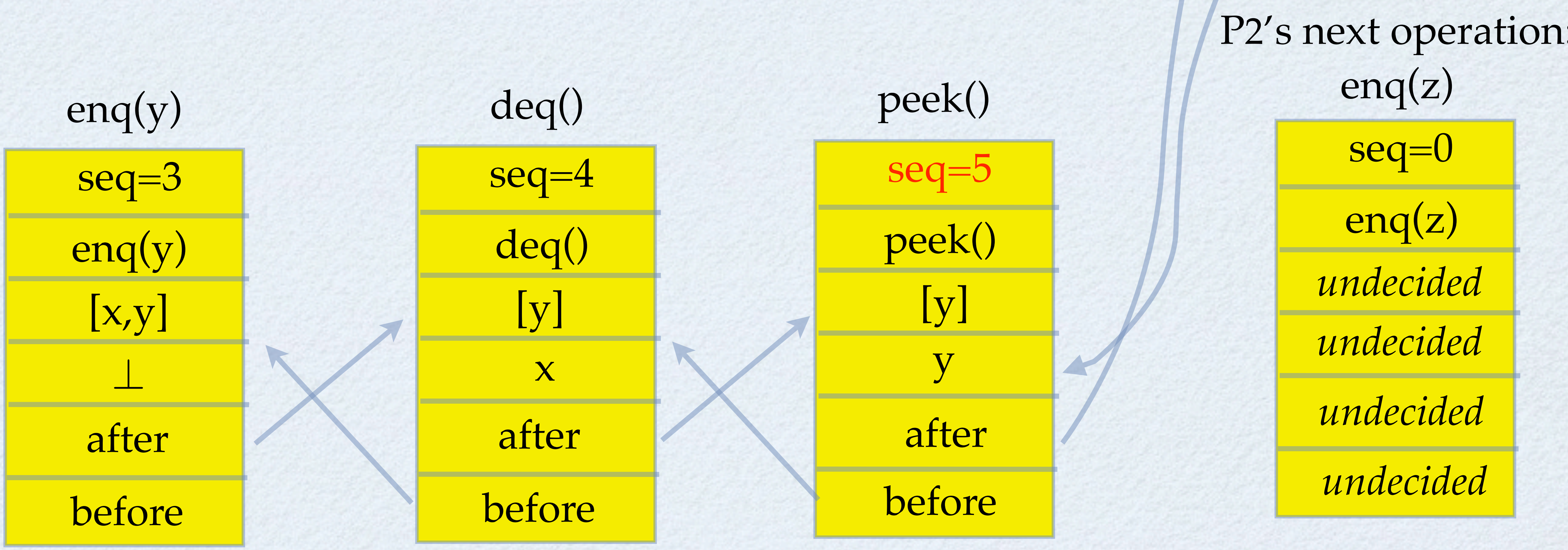
b) Consensus object can used only once.

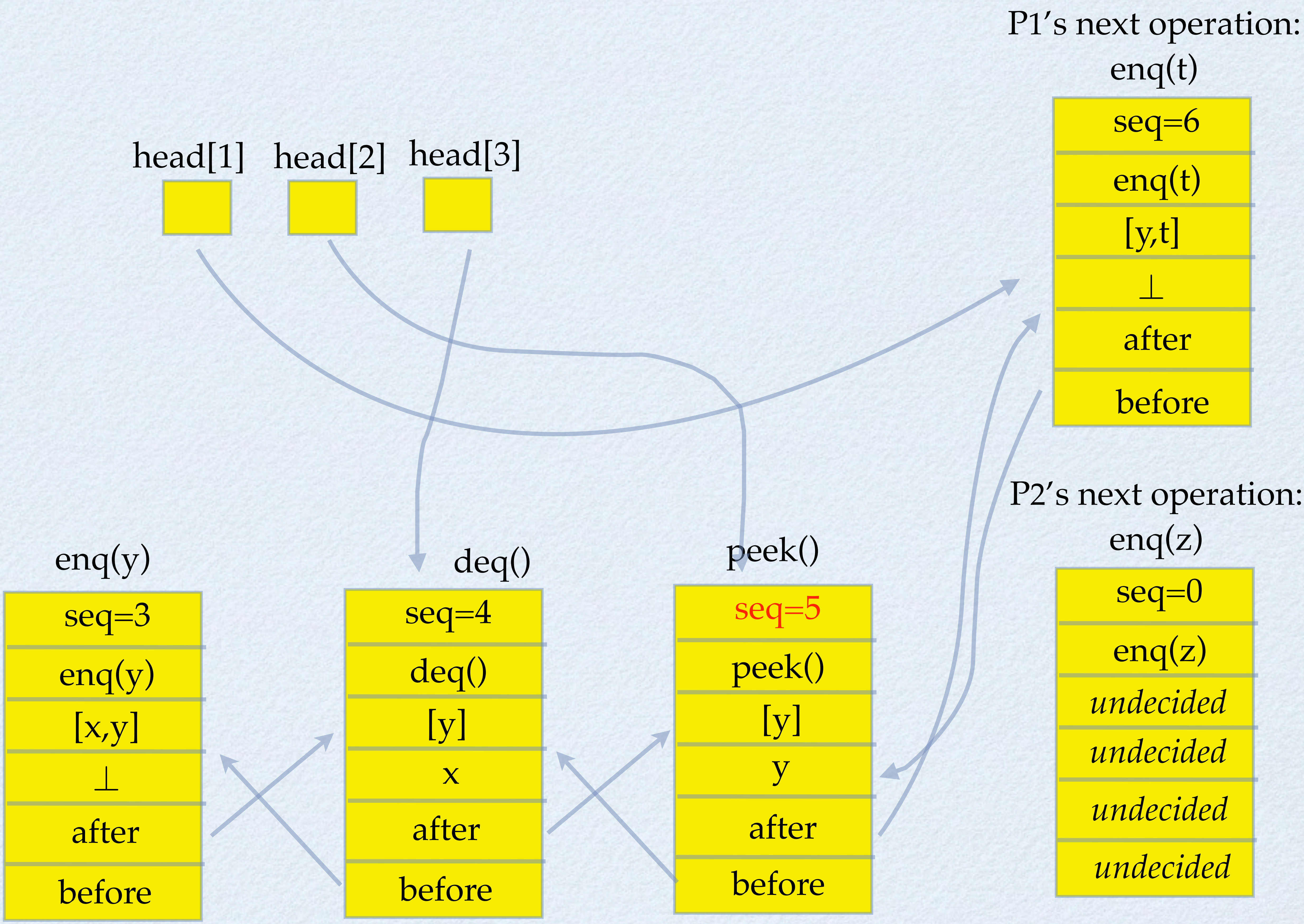


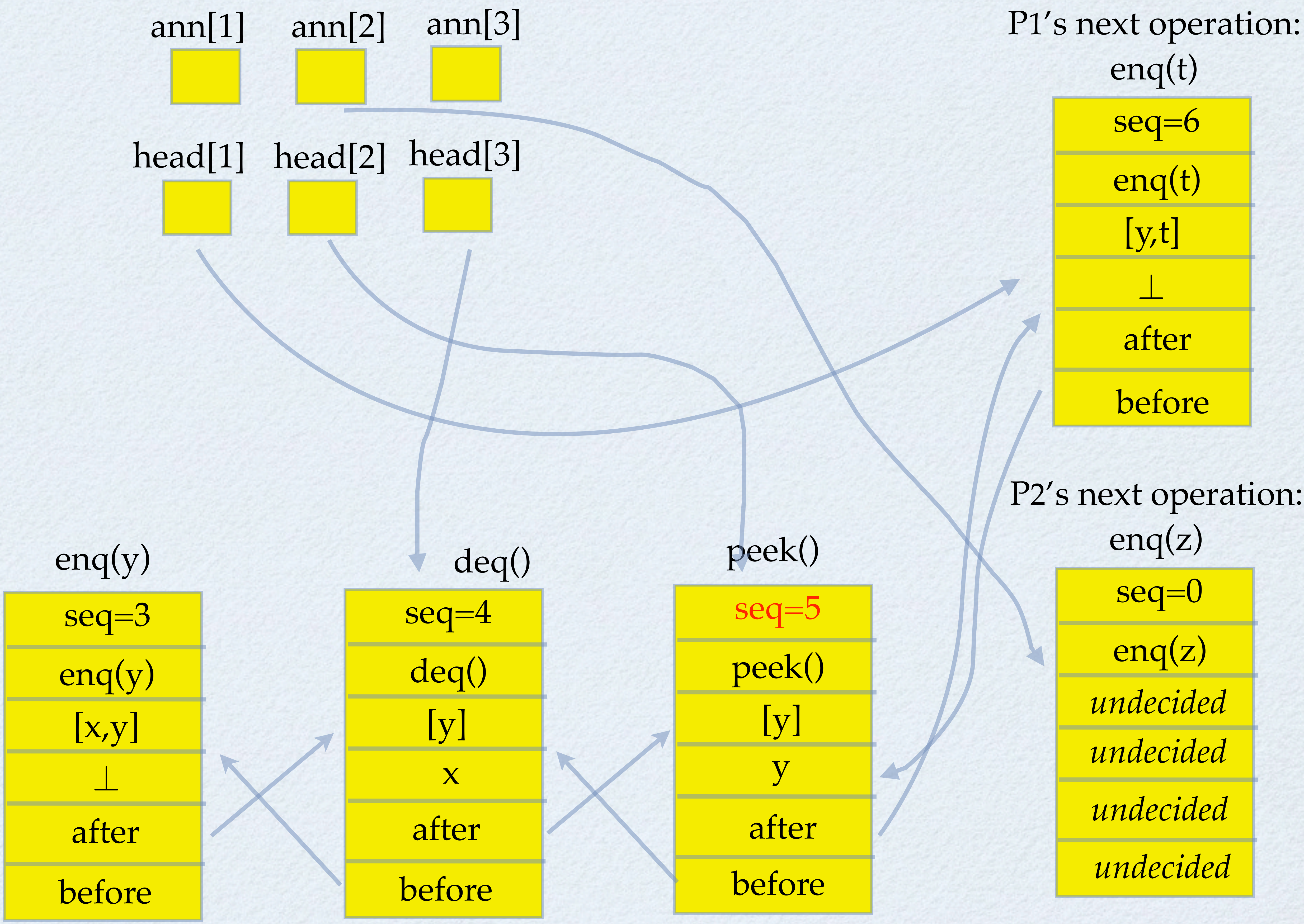
Solution:

1. an array of atomic registers head[] pointing to the latest cell each process has seen
2. an array of atomic registers announce[] pointing to cells to be threaded

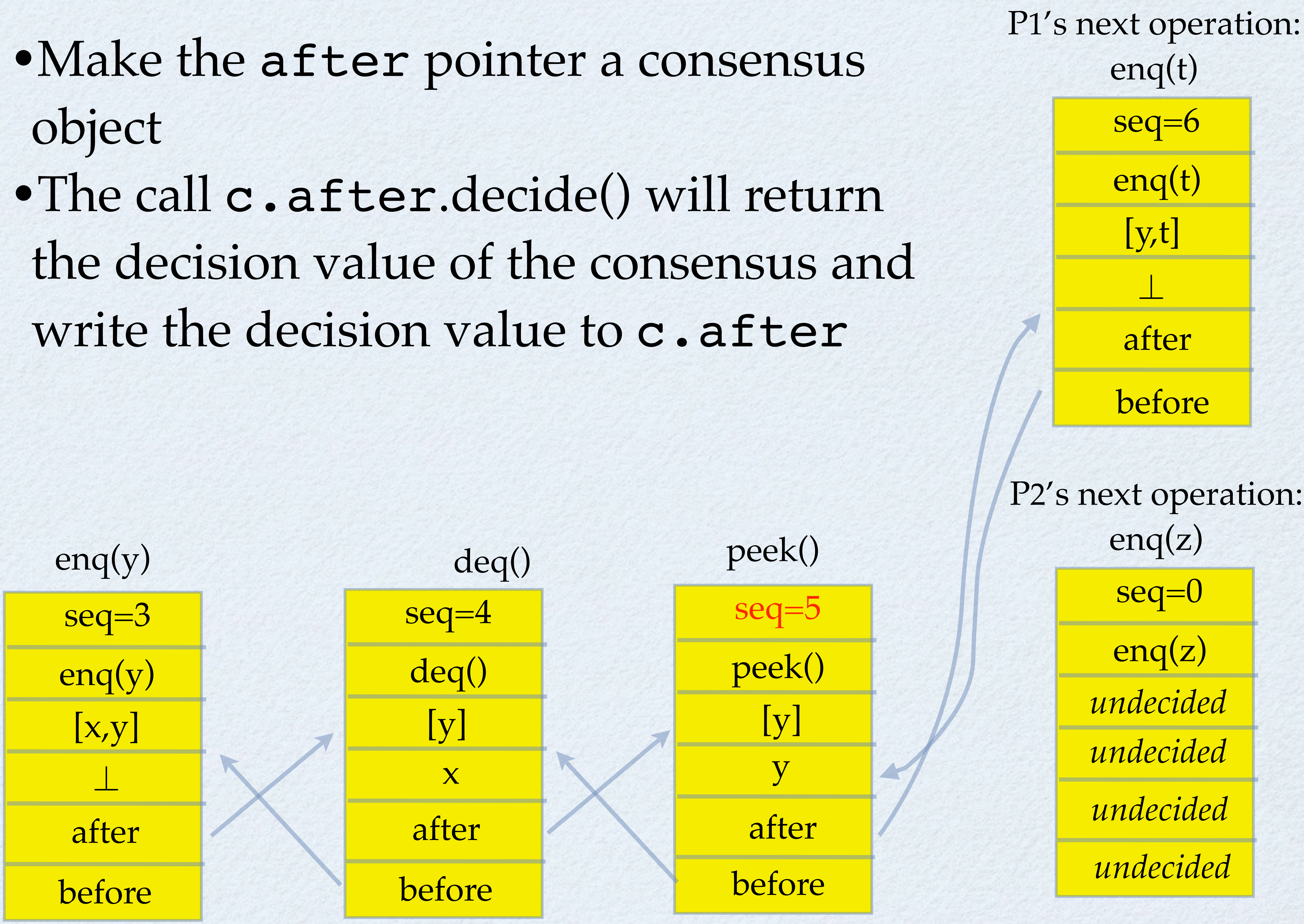





P1's next operation:
enq(t)
seq=6
enq(t)
[y,t]
⊥
after
before
head[1]
head[2]
head[3]
P2's next operation:
enq(z)
seq=0
enq(z)
undecided
undecided
undecided
undecided
enq(y)
seq=3
enq(y)
[x,y]
⊥
after
before
deq()
seq=4
deq()
[y]
x
after
before
peek()
seq=5
peek()
[y]
y
after
before


ann[1]
ann[2]
ann[3]
head[1]
head[2]
head[3]
P1's next operation:
enq(t)
seq=6
enq(t)
[y,t]
⊥
after
before
P2's next operation:
enq(z)
seq=0
enq(z)
undecided
undecided
undecided
undecided
enq(y)
seq=3
enq(y)
[x,y]
⊥
after
before
deq()
seq=4
deq()
[y]
x
after
before
peek()
seq=5
peek()
[y]
y
after
before

- Make the `after` pointer a consensus object
- The call `c.after`.decide() will return the decision value of the consensus and write the decision value to `c.after`



```
initialize the cell with seq = 0
let announce[P] point to it.
head[P] = max{head[1], ..., head[n]}
while announce[P].seq = 0 do
   c = head[P]
   h = announce[c.seq mod n + 1]
   if h.seq = 0 then
      prefer = h
   else
      prefer = announce[P]
   end if
   d = c.after.decide(prefer)
   d.seq = c.seq + 1
   update the field of d according to c.inv, c.new-state
   head[P] = d
end while
return announce[P].result
```

```
initialize the cell with seq = 0
let announce[P] point to it.
head[P] = max{head[1], ..., head[n]}
while announce[P].seq = 0 do
   c = head[P]
   h = announce[c.seq mod n + 1]
   if h.seq = 0 then
      prefer = h
   else
      prefer = announce[P]
   end if
   d = c.after.decide(prefer)
   d.seq = c.seq + 1
   update the field of d according to c.inv, c.new-state
   head[P] = d
end while
return announce[P].result
```

seq = 0 indicates that the cell has not been threaded

```
initialize the cell with seq = 0
let announce[P] point to it.
head[P] = max{head[1], . . . , head[n]}
while announce[P].seq = 0 do
   c = head[P]
   h = announce[c.seq mod n + 1]
   if h.seq = 0 then
      prefer = h
   else
      prefer = announce[P]
   end if
   d = c.after.decide(prefer)
   d.seq = c.seq + 1
   update the field of d according to c.inv, c.new-state
   head[P] = d
end while
return announce[P].result
```

```
initialize the cell with seq = 0
let announce[P] point to it.
head[P] = max{head[1], . . . , head[n]}
while announce[P].seq = 0 do
   c = head[P]
   h = announce[c.seq mod n + 1]
   if h.seq = 0 then
      prefer = h
   else
      prefer = announce[P]
   end if
   d = c.after.decide(prefer)
   d.seq = c.seq + 1
   update the field of d according to c.inv, c.new-state
   head[P] = d
end while
return announce[P].result
```

make the head be as close to the end of the list as possible

```
initialize the cell with seq = 0
let announce[P] point to it.
head[P] = max{head[1], . . . , head[n]}
while  announce[P].seq = 0 do
   c = head[P]
   h = announce[c.seq mod n + 1]
   if  h.seq = 0 then
      prefer = h
   else
      prefer = announce[P]
   end if
   d = c.after.decide(prefer)
   d.seq = c.seq + 1
   update the field of d according to c.inv, c.new-state
   head[P] = d
end while
return  announce[P].result
```

actually it is a loop.
just for brevity. no
atomicity requirement

```
initialize the cell with seq = 0
let announce[P] point to it.
head[P] = max{head[1], ..., head[n]}
while announce[P].seq = 0 do
   c = head[P]
   h = announce[c.seq mod n + 1]
   if h.seq = 0 then
      prefer = h
   else
      prefer = announce[P]
   end if
   d = c.after.decide(prefer)
   d.seq = c.seq + 1
   update the field of d according to c.inv, c.new-state
   head[P] = d
end while
return announce[P].result
```

The main loop. iterates as long as the cell is not threaded

```
initialize the cell with seq = 0
let announce[P] point to it.
head[P] = max{head[1], . . . , head[n]}
while  announce[P].seq = 0 do
   c = head[P]
   h = announce[c.seq mod n + 1]
   if h.seq = 0 then
      prefer = h
   else
      prefer = announce[P]
   end if
   d = c.after.decide(prefer)
   d.seq = c.seq + 1
   update the field of d according to c.inv, c.new-state
   head[P] = d
end while
return  announce[P].result
```

```
initialize the cell with seq = 0
let announce[P] point to it.
head[P] = max{head[1], ..., head[n]}
while announce[P].seq = 0 do
  c = head[P]
  h = announce[c.seq mod n + 1]
  if h.seq = 0 then
    prefer = h
  else
    prefer = announce[P]
  end if
  d = c.after.decide(prefer)
  d.seq = c.seq + 1
  update the field of d according to c.inv, c.new-state
  head[P] = d
end while
return announce[P].result
```

h is the cell that the thread tries to help when its head pointer points to c

```
initialize the cell with seq = 0
let announce[P] point to it.
head[P] = max{head[1], . . . , head[n]}
while announce[P].seq = 0 do
  c = head[P]
  h = announce[c.seq mod n + 1]
  if h.seq = 0 then
    prefer = h
  else
    prefer = announce[P]
  end if
  d = c.after.decide(prefer)
  d.seq = c.seq + 1
  update the field of d according to c.inv, c.new-state
  head[P] = d
end while
return announce[P].result
```

```
initialize the cell with seq = 0
let announce[P] point to it.
head[P] = max{head[1], . . . , head[n]}
while announce[P].seq = 0 do
   c = head[P]
   h = announce[c.seq mod n + 1]
   if h.seq = 0 then
      prefer = h
   else
      prefer = announce[P]
   end if
   d = c.after.decide(prefer)
   d.seq = c.seq + 1
   update the field of d according to c.inv, c.new-state
   head[P] = d
end while
return announce[P].result
```

check if h needs help, or if h has not yet been threaded.

```
initialize the cell with seq = 0
let announce[P] point to it.
head[P] = max{head[1], . . . , head[n]}
while announce[P].seq = 0 do
   c = head[P]
   h = announce[c.seq mod n + 1]
   if h.seq = 0 then
      prefer = h
   else
      prefer = announce[P]
   end if
   d = c.after.decide(prefer)
   d.seq = c.seq + 1
   update the field of d according to c.inv, c.new-state
   head[P] = d
end while
return announce[P].result
```

otherwise try to thread own cell

```
initialize the cell with seq = 0
let announce[P] point to it.
head[P] = max{head[1], . . . , head[n]}
while  announce[P].seq = 0 do
   c = head[P]
   h = announce[c.seq mod n + 1]
   if h.seq = 0 then
      prefer = h
   else
      prefer = announce[P]
   end if
   d = c.after.decide(prefer)
   d.seq = c.seq + 1
   update the field of d according to c.inv, c.new-state
   head[P] = d
end while
return  announce[P].result
```

Observe that c.after is a consensus object and however many times decide() is called, the return value is the same.

```
initialize the cell with seq = 0
let announce[P] point to it.
head[P] = max{head[1], ..., head[n]}
while announce[P].seq = 0 do
   c = head[P]
   h = announce[c.seq mod n + 1]
   if h.seq = 0 then
      prefer = h
   else
      prefer = announce[P]
   end if
   d = c.after.decide(prefer)
   d.seq = c.seq + 1
   update the field of d according to c.inv, c.new-state
   head[P] = d
end while
return announce[P].result
```

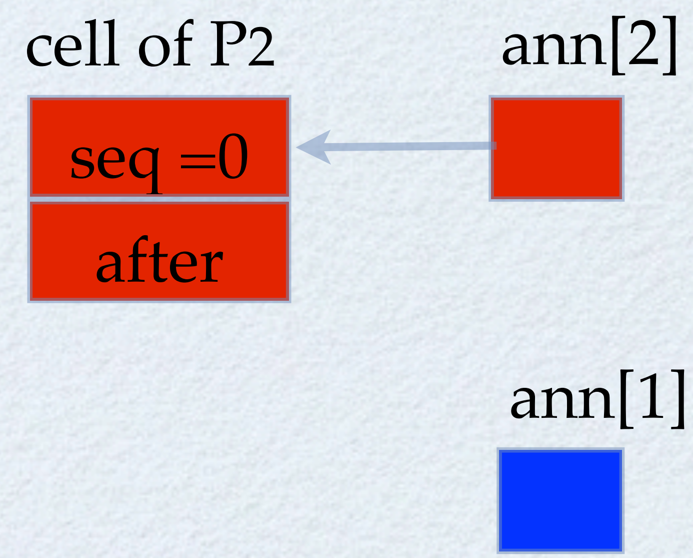
```
initialize the cell with seq = 0
let announce[P] point to it.
head[P] = max{head[1], ..., head[n]}
while announce[P].seq = 0 do
   c = head[P]
   h = announce[c.seq mod n + 1]
   if h.seq = 0 then
      prefer = h
   else
      prefer = announce[P]
   end if
   d = c.after.decide(prefer)
   d.seq = c.seq + 1
   update the field of d according to c.inv, c.new-state
   head[P] = d
end while
return announce[P].result
```

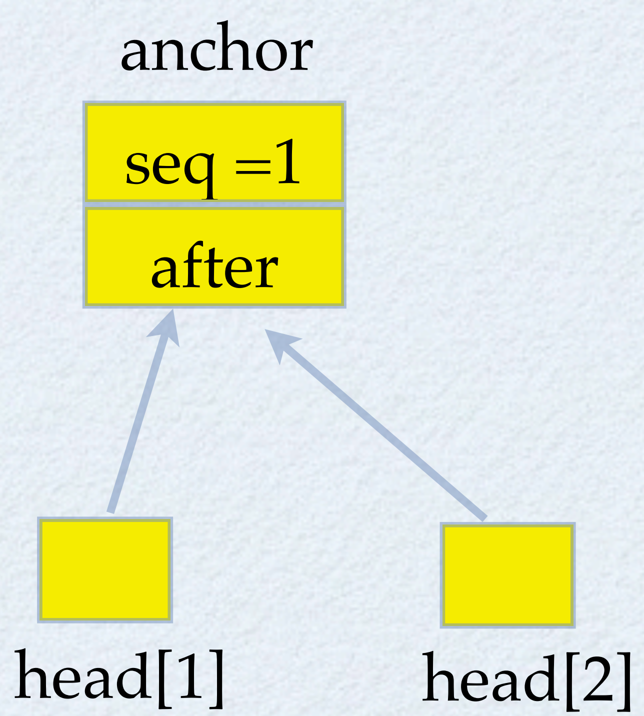
```
initialize the cell with seq = 0
let announce[P] point to it.
head[P] = max{head[1], . . . , head[n]}
while announce[P].seq = 0 do
   c = head[P]
   h = announce[c.seq mod n + 1]
   if h.seq = 0 then
      prefer = h
   else
      prefer = announce[P]
   end if
   d = c.after.decide(prefer)
   d.seq = c.seq + 1
   update the field of d according to c.inv, c.new-state
   head[P] = d
end while
return announce[P].result
```

However many times d is updated by different processes, the result is the same!!

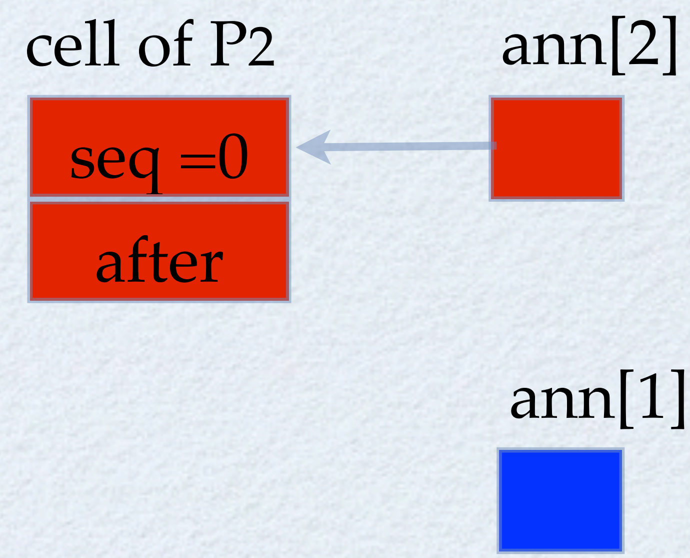
```
initialize the cell with seq = 0
let announce[P] point to it.
head[P] = max{head[1], . . . , head[n]}
while announce[P].seq = 0 do
   c = head[P]
   h = announce[c.seq mod n + 1]
   if h.seq = 0 then
      prefer = h
   else
      prefer = announce[P]
   end if
   d = c.after.decide(prefer)
   d.seq = c.seq + 1
   update the field of d according to c.inv, c.new-state
   head[P] = d
end while
return announce[P].result
```

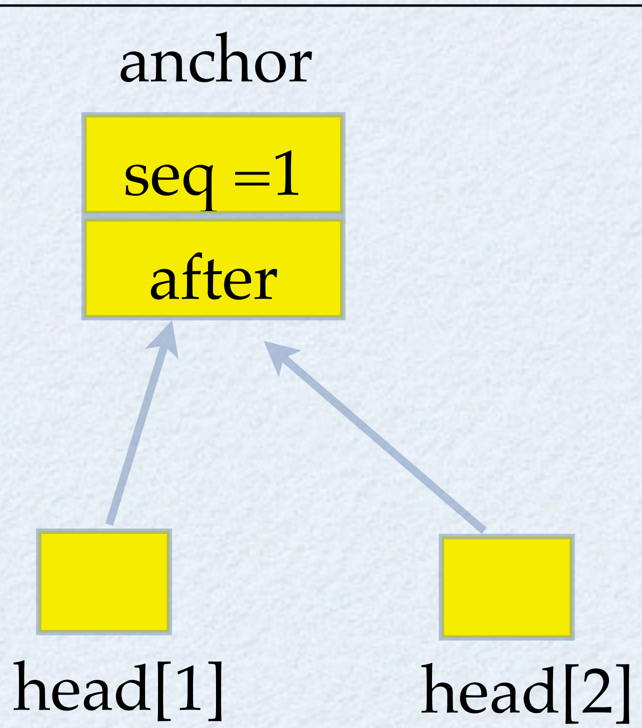
```
initialize the cell with seq = 0
let announce[P] point to it.
head[P] = max{head[1], ..., head[n]}
while announce[P].seq = 0 do
   c = head[P]
   h = announce[c.seq mod n + 1]
   if h.seq = 0 then
      prefer = h
   else
      prefer = announce[P]
   end if
   d = c.after.decide(prefer)
   d.seq = c.seq + 1
   update the field of d according to c.inv, c.new-state
   head[P] = d
end while
return announce[P].result
```



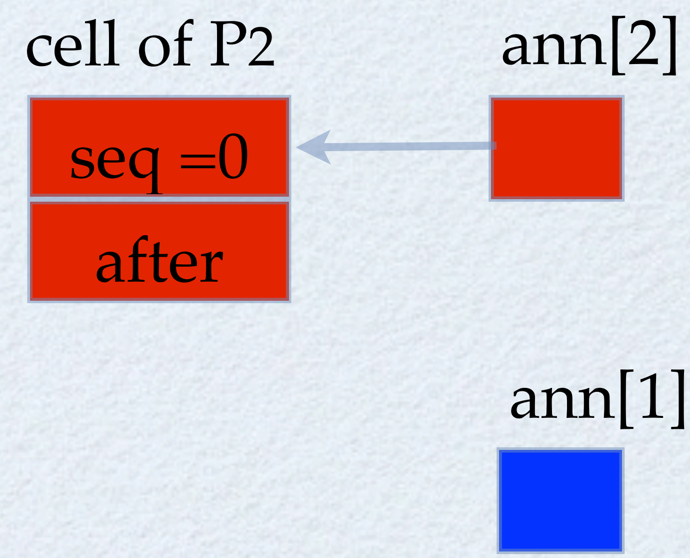


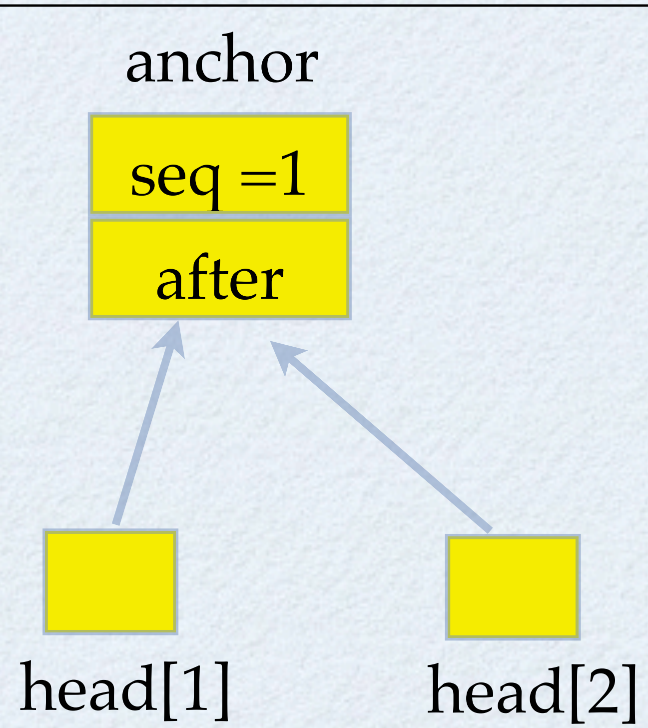
```
initialize the cell with seq = 0
let announce[P] point to it.
head[P] = max{head[1], ..., head[n]}
while announce[P].seq = 0 do
   c = head[P]
   h = announce[c.seq mod n + 1]
   if h.seq = 0 then
      prefer = h
   else
      prefer = announce[P]
   end if
   d = c.after.decide(prefer)
   d.seq = c.seq + 1
   update the field of d according to c.inv, c.new-state
   head[P] = d
end while
return announce[P].result
```



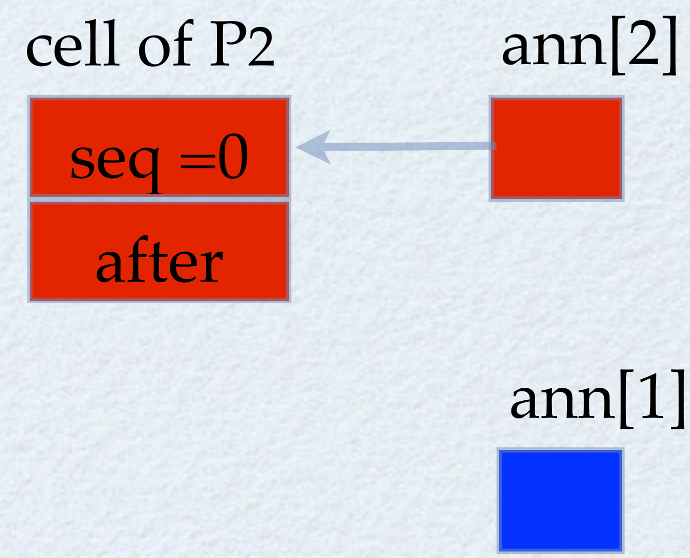


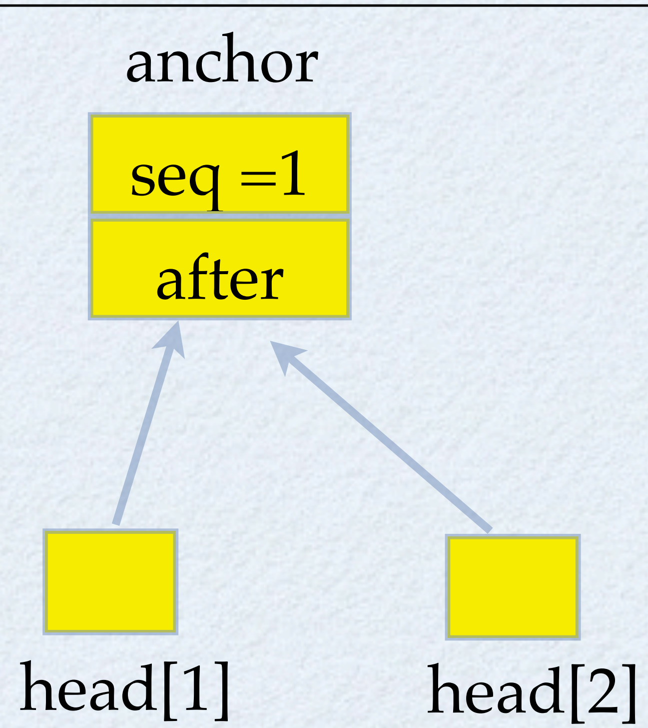
```
initialize the cell with seq = 0
let announce[P] point to it.
head[P] = max{head[1], . . . , head[n]}
while  announce[P].seq = 0 do
   c = head[P]
   h = announce[c.seq mod n + 1]
   if h.seq = 0 then
      prefer = h
   else
      prefer = announce[P]
   end if
   d = c.after.decide(prefer)
   d.seq = c.seq + 1
   update the field of d according to c.inv, c.new-state
   head[P] = d
end while
return  announce[P].result
```



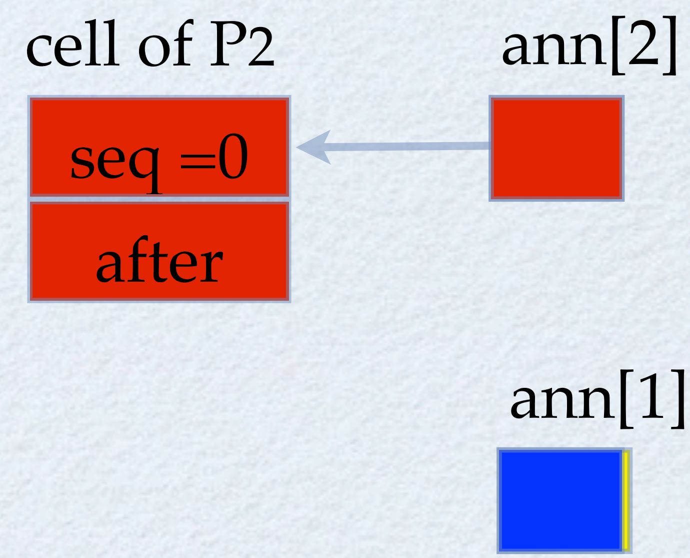


```
initialize the cell with seq = 0
let announce[P] point to it.
head[P] = max{head[1], ..., head[n]}
while announce[P].seq = 0 do
   c = head[P]
   h = announce[c.seq mod n + 1]
   if h.seq = 0 then
      prefer = h
   else
      prefer = announce[P]
   end if
   d = c.after.decide(prefer)
   d.seq = c.seq + 1
   update the field of d according to c.inv, c.new-state
   head[P] = d
end while
return announce[P].result
```

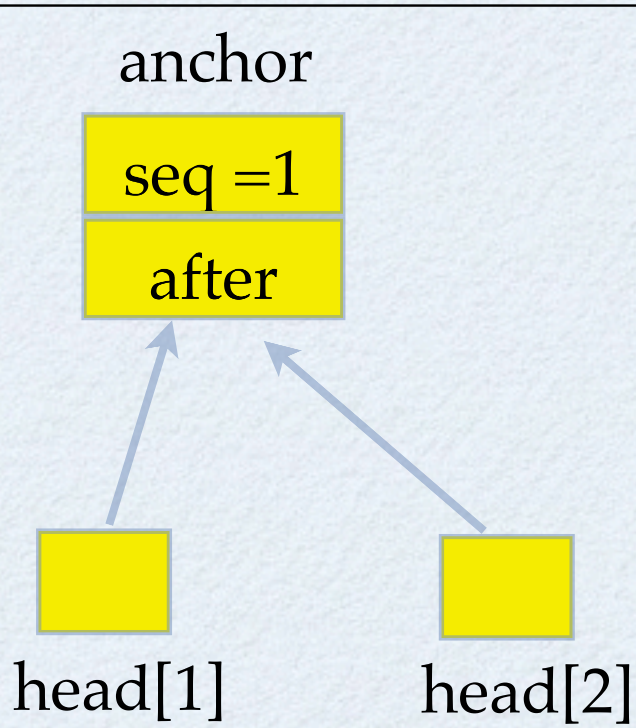




```
initialize the cell with seq = 0
let announce[P] point to it.
head[P] = max{head[1], ..., head[n]}
while  announce[P].seq = 0 do
   c = head[P]
   h = announce[c.seq mod n + 1]
   if h.seq = 0 then
      prefer = h
   else
      prefer = announce[P]
   end if
   d = c.after.decide(prefer)
   d.seq = c.seq + 1
   update the field of d according to c.inv, c.new-state
   head[P] = d
end while
return  announce[P].result
```

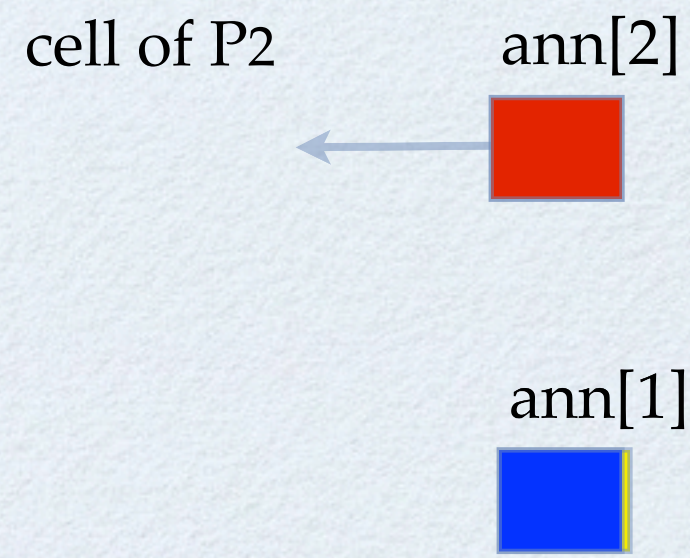


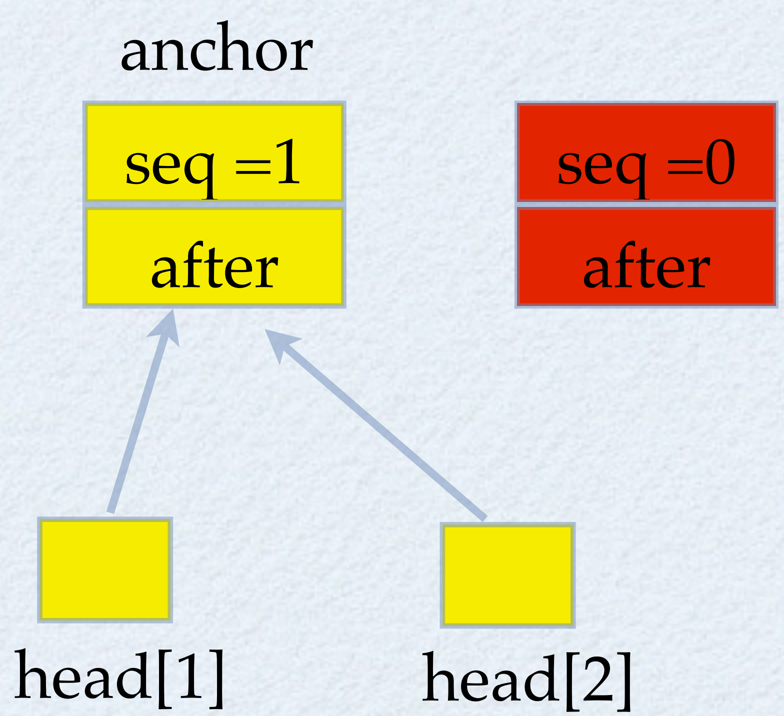
wait-free P2 should decide



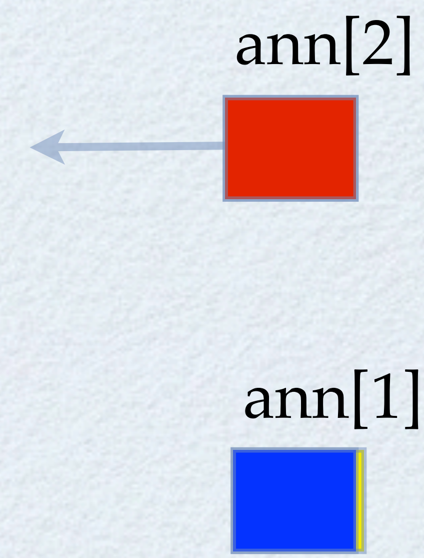
```
initialize the cell with seq = 0
let announce[P] point to it.
head[P] = max{head[1], ..., head[n]}
while announce[P].seq = 0 do
   c = head[P]
   h = announce[c.seq mod n + 1]
   if h.seq = 0 then
      prefer = h
   else
      prefer = announce[P]
   end if
   d = c.after.decide(prefer)
   d.seq = c.seq + 1
   update the field of d according to c.inv, c.new-state
   head[P] = d
end while
return announce[P].result
```

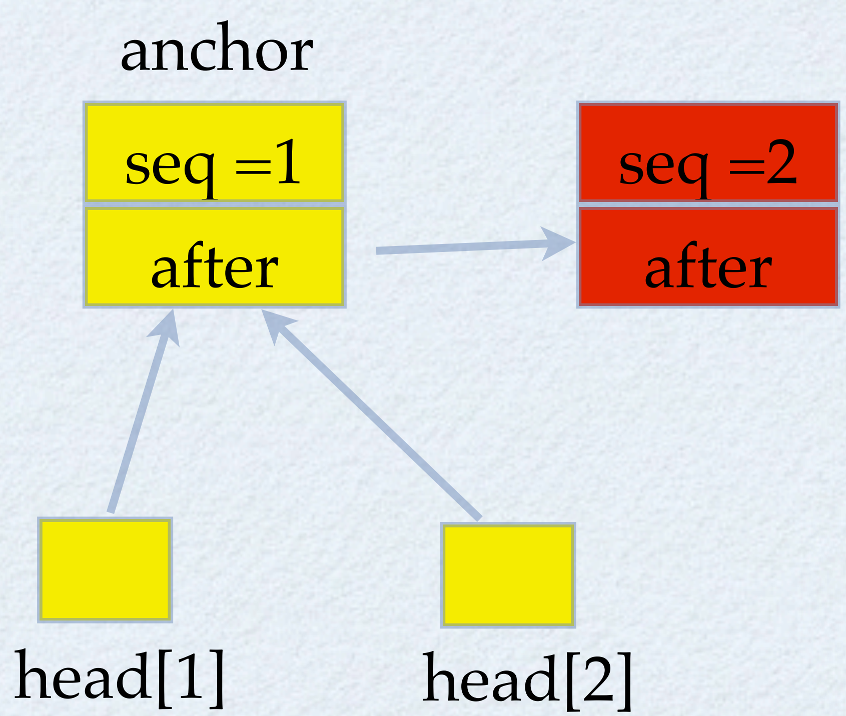
wait-free
P2 should
decide



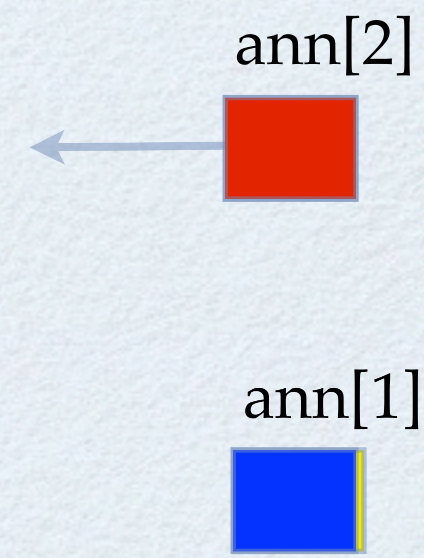


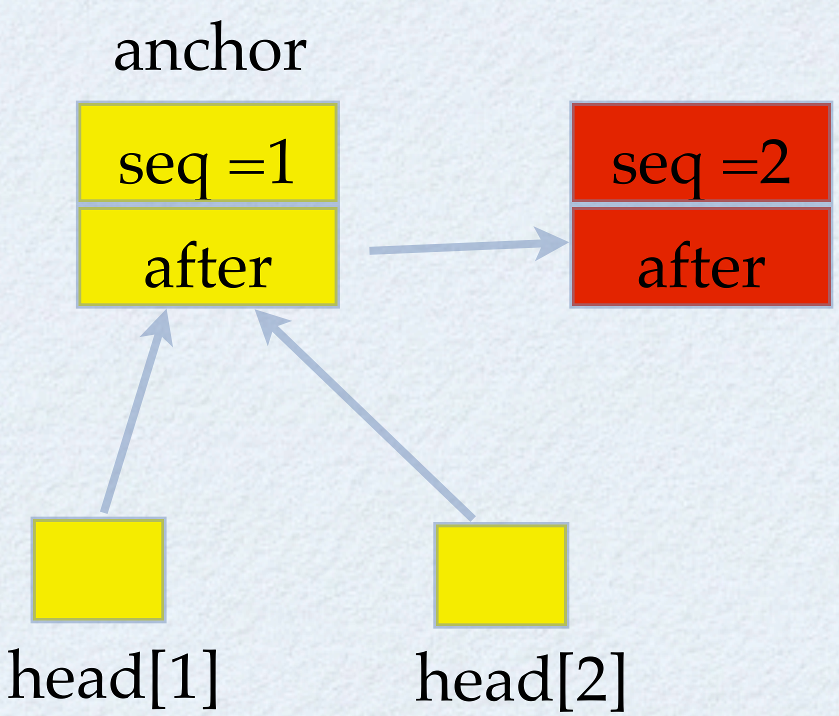
```
initialize the cell with seq = 0
let announce[P] point to it.
head[P] = max{head[1], . . . , head[n]}
while  announce[P].seq = 0 do
   c = head[P]
   h = announce[c.seq mod n + 1]
   if h.seq = 0 then
      prefer = h
   else
      prefer = announce[P]
   end if
   d = c.after.decide(prefer)
   d.seq = c.seq + 1
   update the field of d according to c.inv, c.new-state
   head[P] = d
end while
return  announce[P].result
```



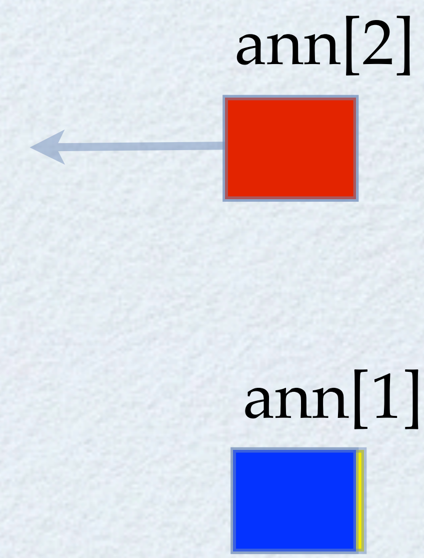


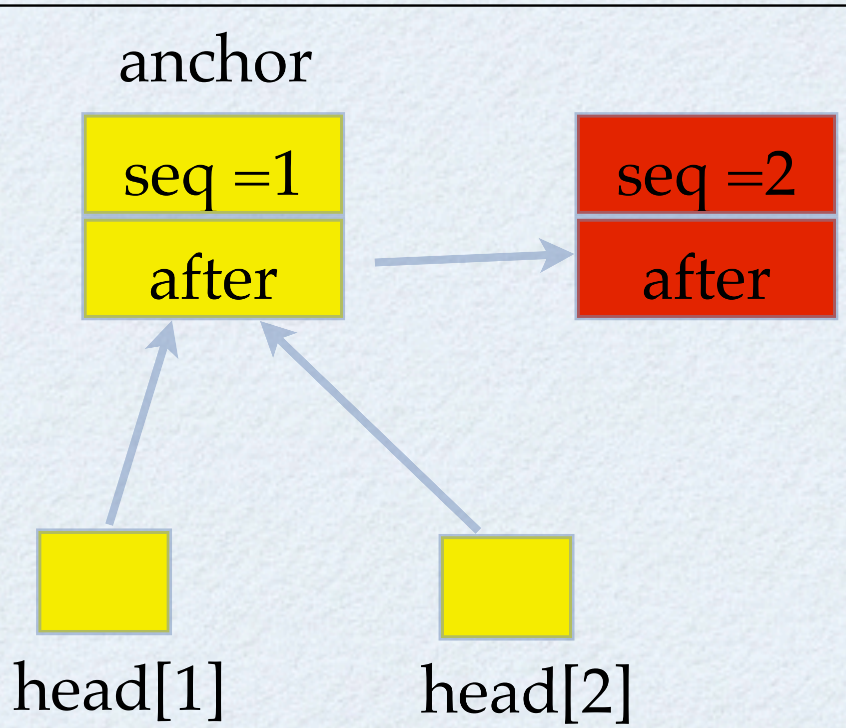
```
initialize the cell with seq = 0
let announce[P] point to it.
head[P] = max{head[1], . . . , head[n]}
while announce[P].seq = 0 do
   c = head[P]
   h = announce[c.seq mod n + 1]
   if h.seq = 0 then
      prefer = h
   else
      prefer = announce[P]
   end if
   d = c.after.decide(prefer)
   d.seq = c.seq + 1
   update the field of d according to c , c.new-state
   head[P] = d
end while
return announce[P].result
```



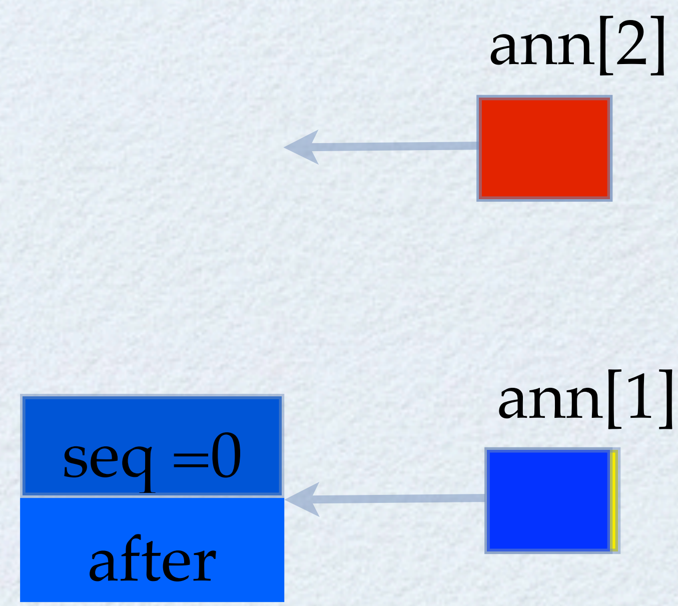


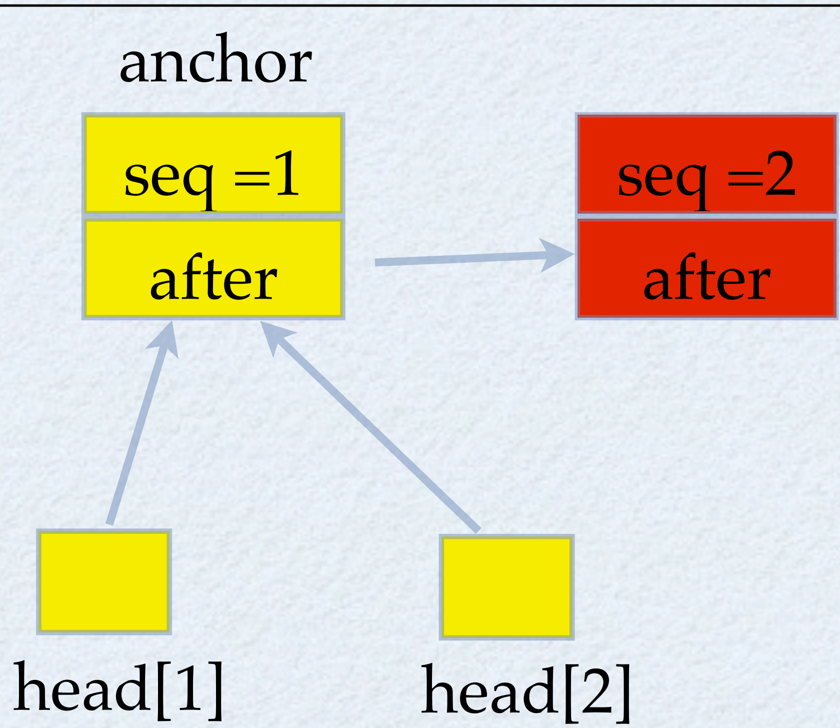
```
initialize the cell with seq = 0
let announce[P] point to it.
head[P] = max{head[1], ..., head[n]}
while  announce[P].seq = 0 do
   c = head[P]
   h = announce[c.seq mod n + 1]
   if h.seq = 0 then
      prefer = h
   else
      prefer = announce[P]
   end if
   d = c.after.decide(prefer)
   d.seq = c.seq + 1
   update the field of d according to [illegible], c.new-state
   head[P] = d
end while
return  announce[P].result
```



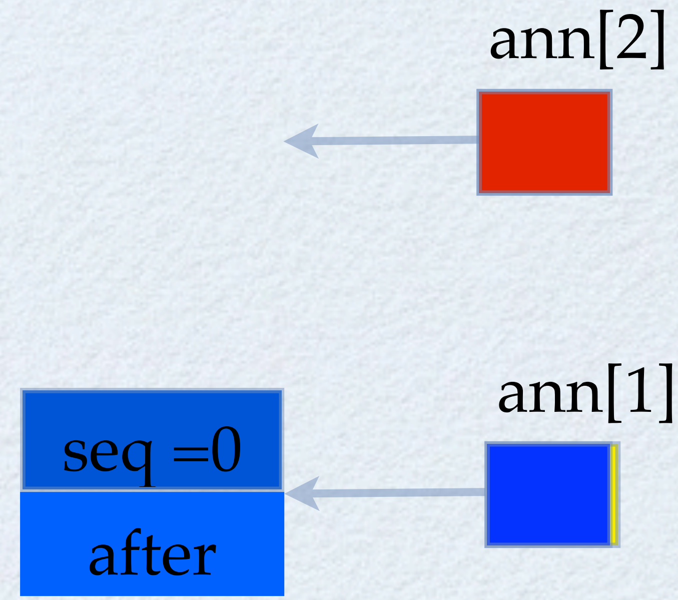


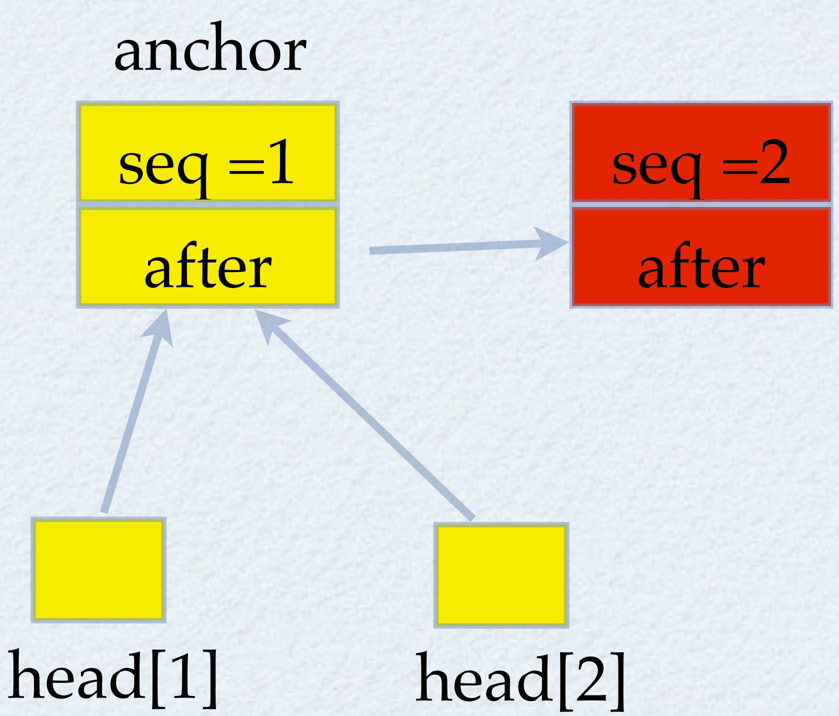
initialize the cell with $seq = 0$
let announce[P] point to it.
head[P] = $\max\{head[1], \ldots, head[n]\}$
while announce[P].seq = 0 **do**
  c = head[P]
  h = announce[c.seq mod n + 1]
  if h.seq = 0 **then**
    prefer = h
  else
    prefer = announce[P]
  end if
  d = c.after.decide(prefer)
  d.seq = c.seq + 1
  update the field of d according to [illegible], c.new-state
  head[P] = d
end while
return announce[P].result



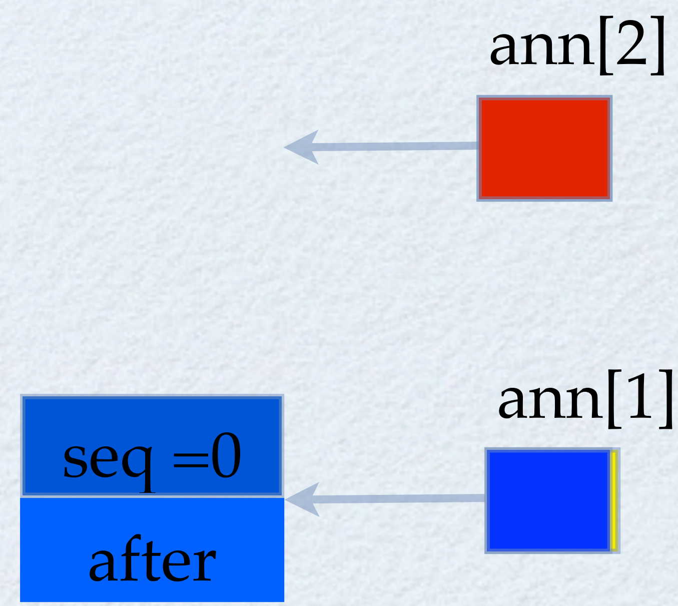


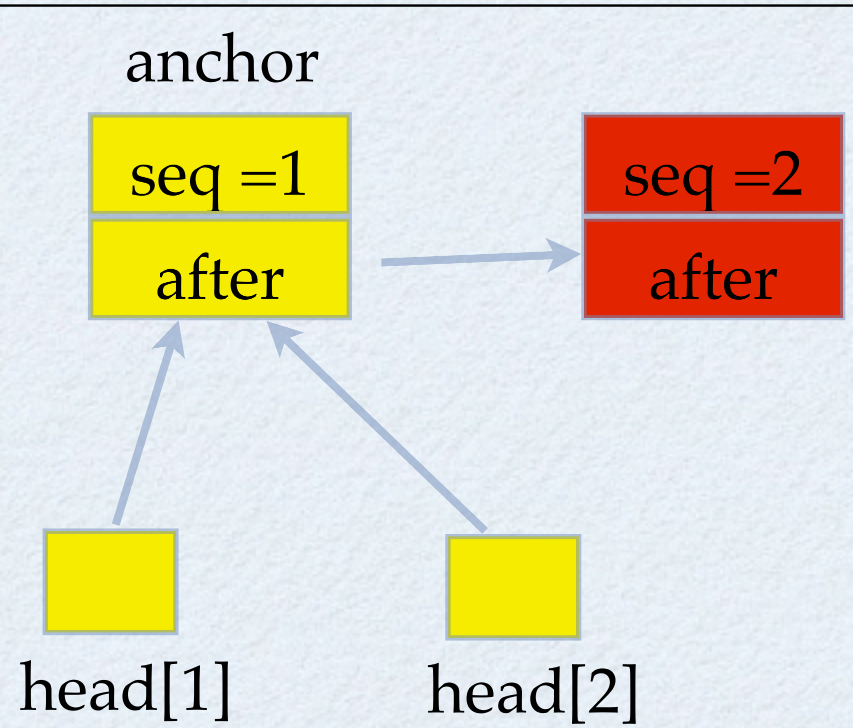
initialize the cell with $seq = 0$
let announce[P] point to it.
head[P] $= \max\{head[1], \ldots, head[n]\}$
while announce[P].seq = 0 **do**
  c = head[P]
  h = announce[c.seq mod n + 1]
  if h.seq = 0 **then**
    prefer = h
  else
    prefer = announce[P]
  end if
  d = c.after.decide(prefer)
  d.seq = c.seq + 1
  update the field of d according to c[illegible], c.new-state
  head[P] = d
end while
return announce[P].result



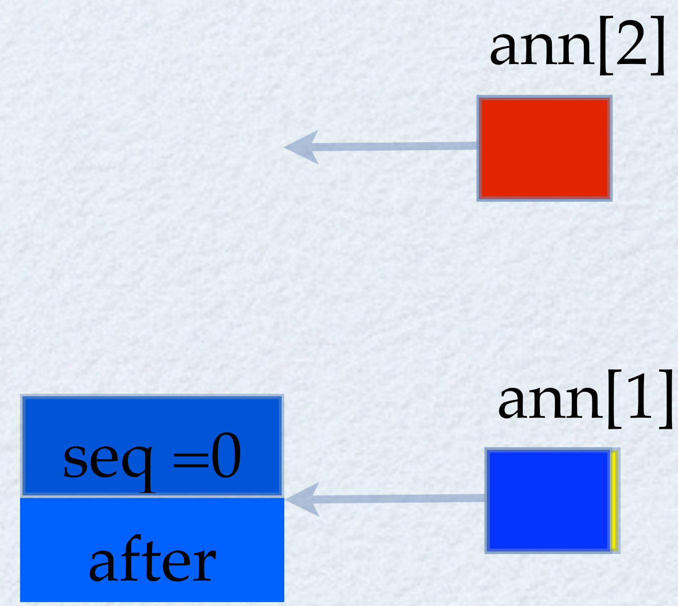


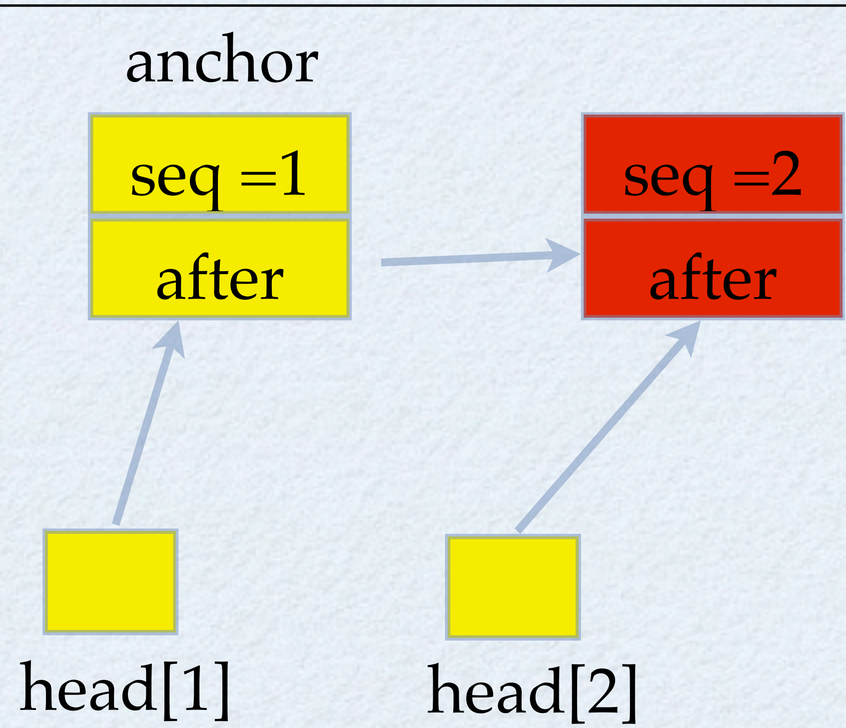
```
initialize the cell with seq = 0
let announce[P] point to it.
head[P] = max{head[1], ..., head[n]}
while announce[P].seq = 0 do
   c = head[P]
   h = announce[c.seq mod n + 1]
   if h.seq = 0 then
      prefer = h
   else
      prefer = announce[P]
   end if
   d = c.after.decide(prefer)
   d.seq = c.seq + 1
   update the field of d according to c.inv, c.new-state
   head[P] = d
end while
return announce[P].result
```



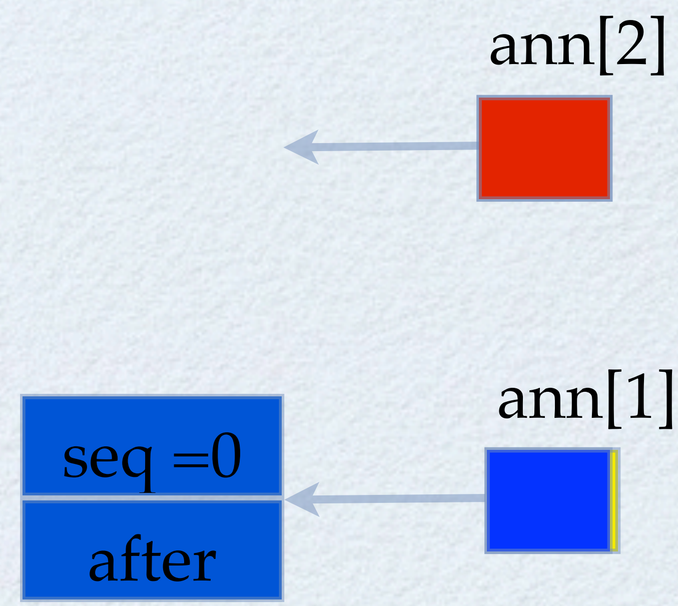


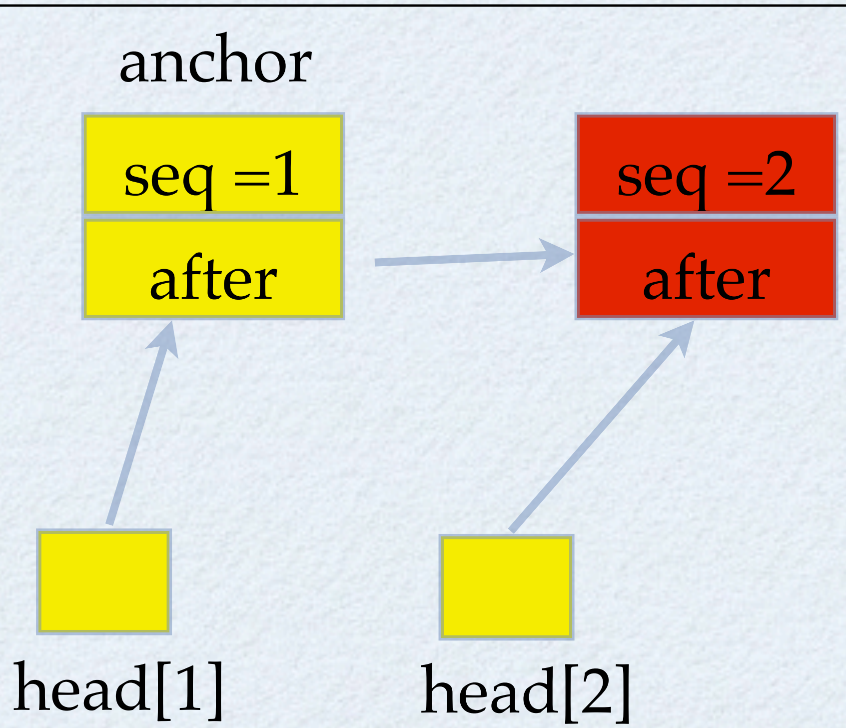
initialize the cell with $seq = 0$
let announce[P] point to it.
head[P] = $\max\{head[1], \ldots, head[n]\}$
while announce[P].seq = 0 **do**
  c = head[P]
  h = announce[c.seq mod n + 1]
  if h.seq = 0 **then**
    prefer = h
  else
    prefer = announce[P]
  end if
  d = c.after.decide(prefer)
  d.seq = c.seq + 1
  update the field of d according to c.inv, c.new-state
  head[P] = d
end while
return announce[P].result



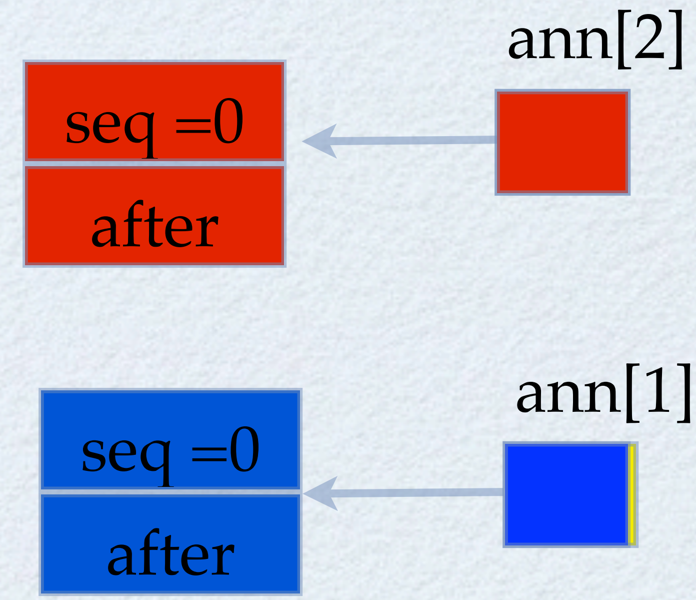


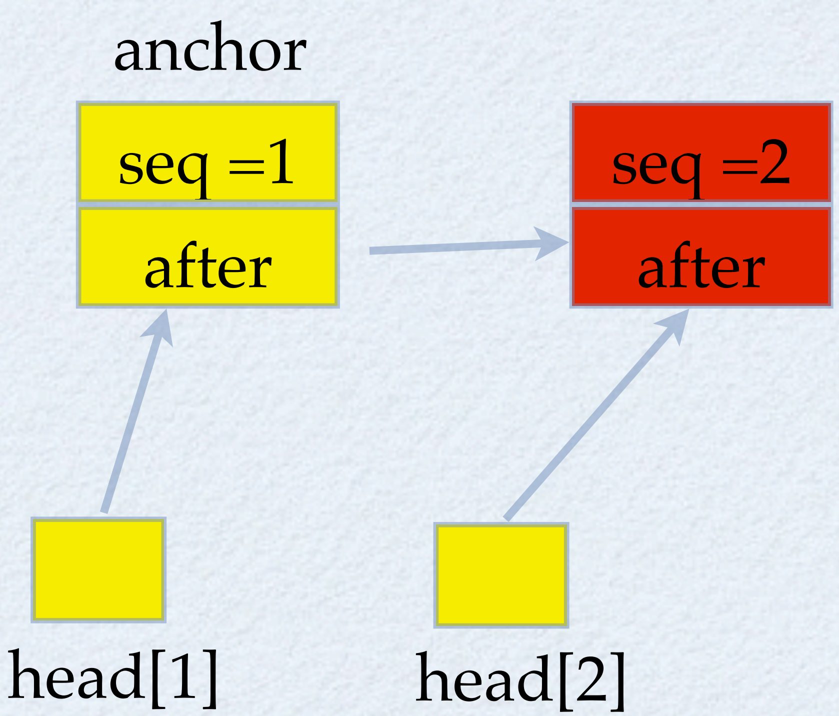
```
initialize the cell with seq = 0
let announce[P] point to it.
head[P] = max{head[1], . . . , head[n]}
while announce[P].seq = 0 do
   c = head[P]
   h = announce[c.seq mod n + 1]
   if h.seq = 0 then
      prefer = h
   else
      prefer = announce[P]
   end if
   d = c.after.decide(prefer)
   d.seq = c.seq + 1
   update the field of d according to c.inv, c.new-state
   head[P] = d
end while
return announce[P].result
```



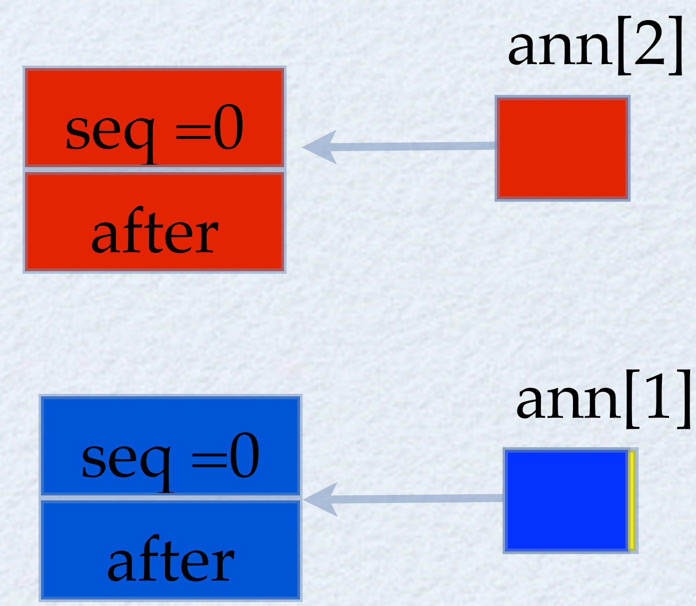


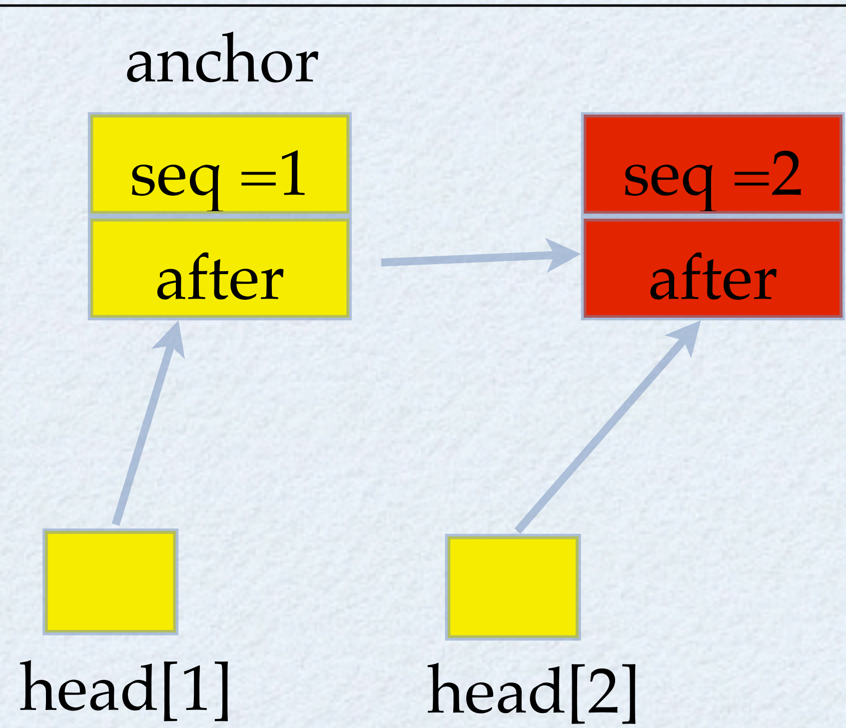
```
initialize the cell with seq = 0
let announce[P] point to it.
head[P] = max{head[1], ..., head[n]}
while announce[P].seq = 0 do
   c = head[P]
   h = announce[c.seq mod n + 1]
   if h.seq = 0 then
      prefer = h
   else
      prefer = announce[P]
   end if
   d = c.after.decide(prefer)
   d.seq = c.seq + 1
   update the field of d according to c.inv, c.new-state
   head[P] = d
end while
return announce[P].result
```





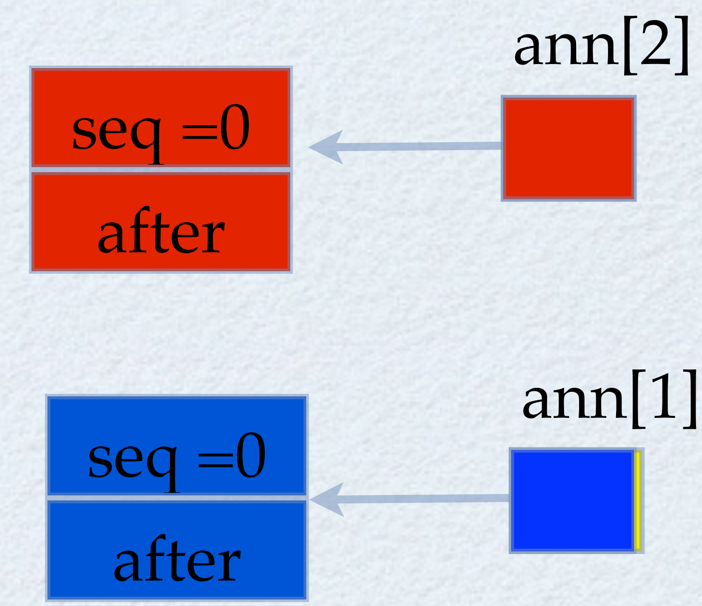
```
initialize the cell with seq = 0
let announce[P] point to it.
head[P] = max{head[1], ..., head[n]}
while  announce[P].seq = 0 do
   c = head[P]
   h = announce[c.seq mod n + 1]
   if h.seq = 0 then
      prefer = h
   else
      prefer = announce[P]
   end if
   d = c.after.decide(prefer)
   d.seq = c.seq + 1
   update the field of d according to c.inv, c.new-state
   head[P] = d
end while
return  announce[P].result
```

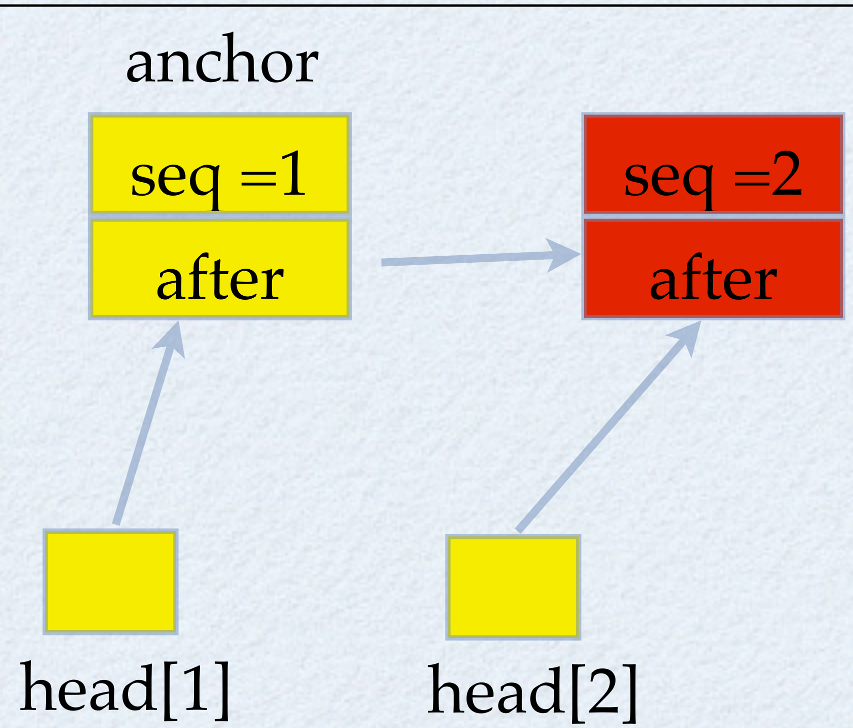




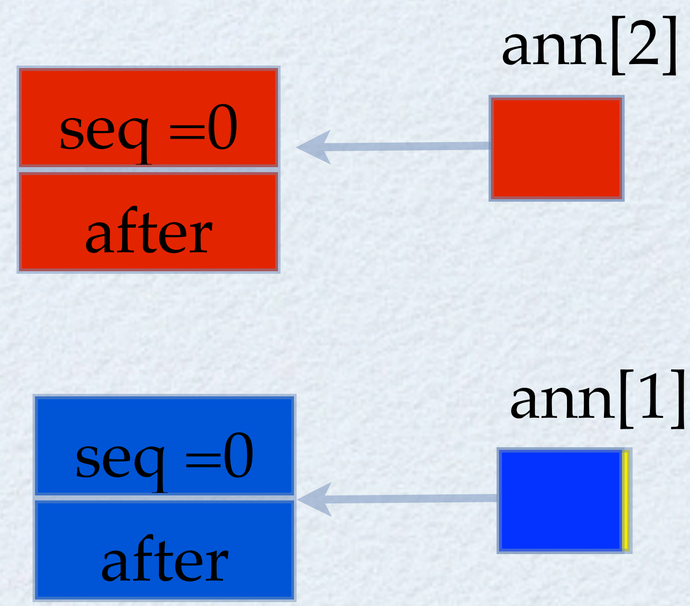
initialize the cell with $seq = 0$
let announce[P] point to it.
head[P] = $\max\{head[1], \ldots, head[n]\}$
while announce[P].seq = 0 **do**
  c = head[P]
  h = announce[c.seq mod n + 1]
  if h.seq = 0 **then**
    prefer = h
  else
    prefer = announce[P]
  end if
  d = c.after.decide(prefer)
  d.seq = c.seq + 1
  update the field of d according to c.inv, c.new-state
  head[P] = d
end while
return announce[P].result

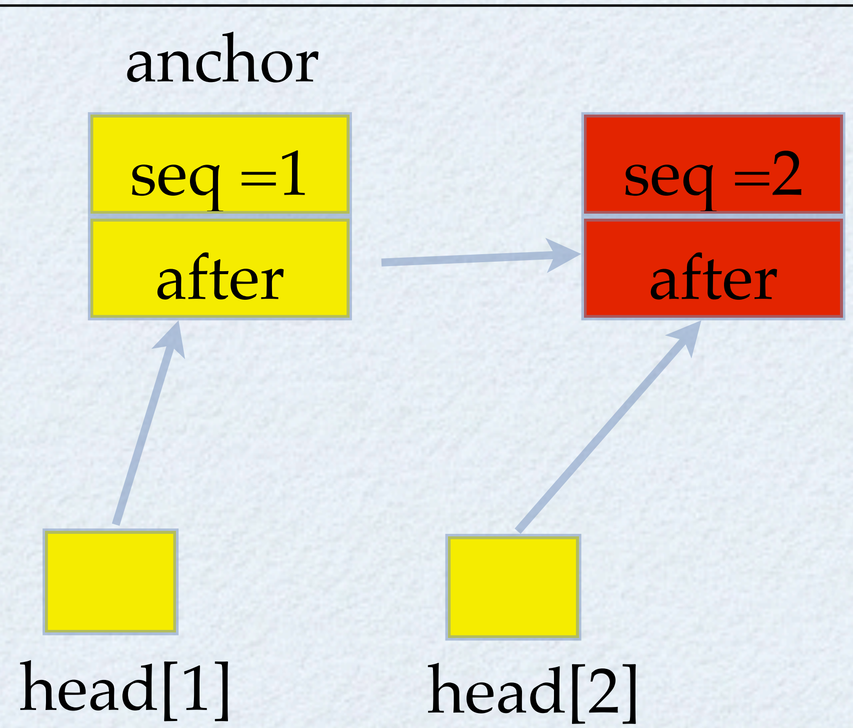
P2 tries to help P1



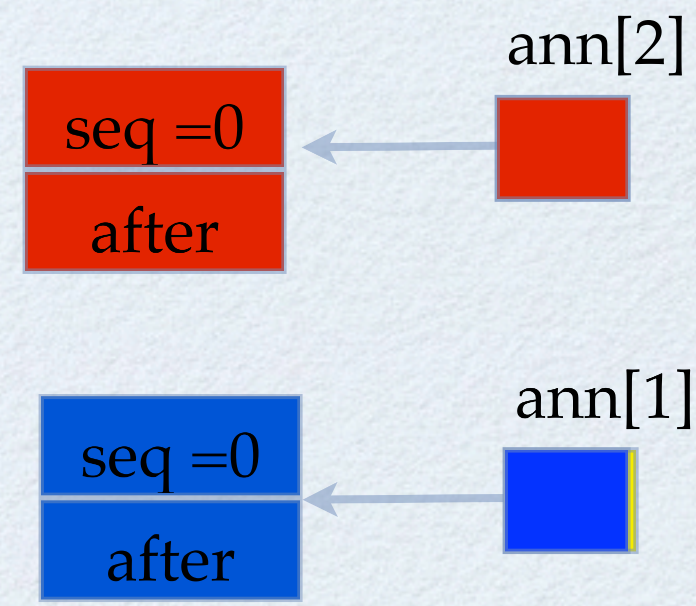


initialize the cell with $seq = 0$
let announce[P] point to it.
head[P] $= \max\{head[1], \ldots, head[n]\}$
while announce[P].seq = 0 **do**
  c = head[P]
  h = announce[c.seq mod n + 1]
  if h.seq = 0 **then**
    prefer = h
  else
    prefer = announce[P]
  end if
  d = c.after.decide(prefer)
  d.seq = c.seq + 1
  update the field of d according to c.inv, c.new-state
  head[P] = d
end while
return announce[P].result

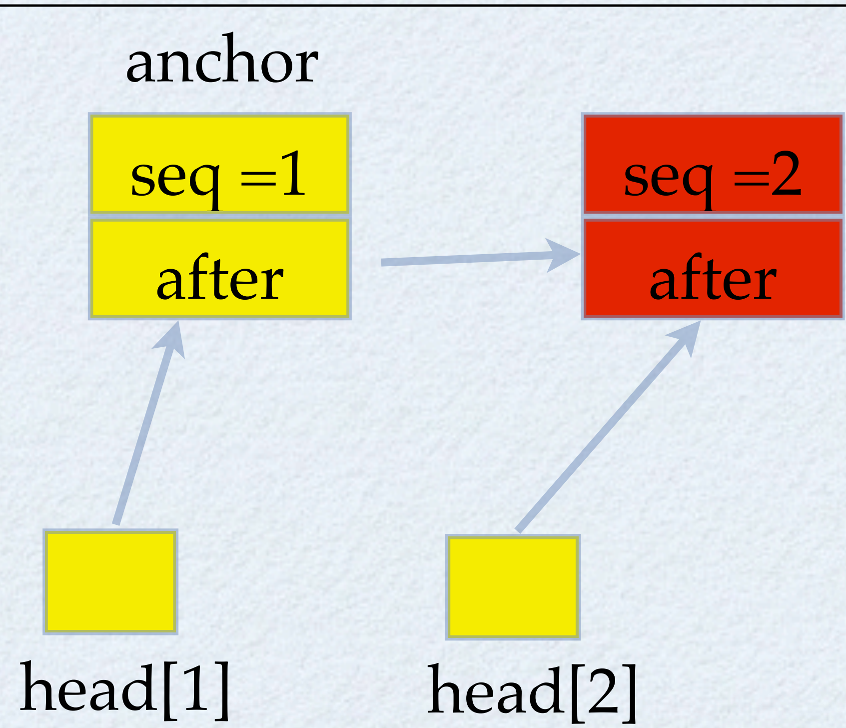




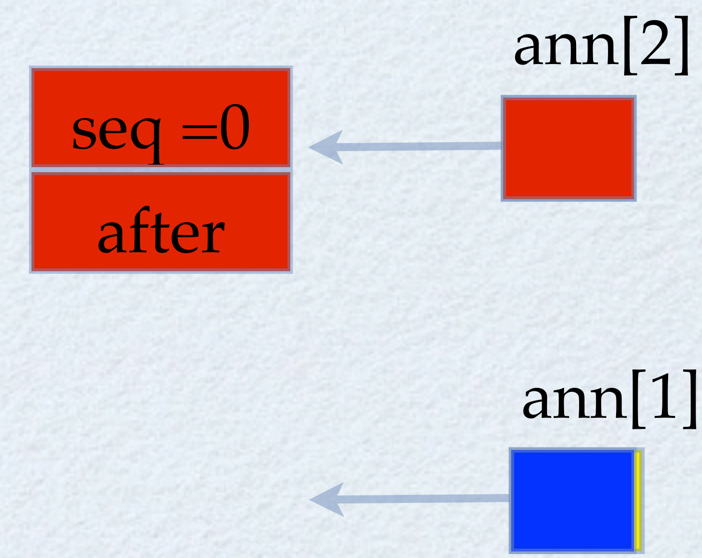
```
initialize the cell with seq = 0
let announce[P] point to it.
head[P] = max{head[1], ..., head[n]}
while announce[P].seq = 0 do
   c = head[P]
   h = announce[c.seq mod n + 1]
   if h.seq = 0 then
      prefer = h
   else
      prefer = announce[P]
   end if
   d = c.after.decide(prefer)
   d.seq = c.seq + 1
   update the field of d according to c.inv, c.new-state
   head[P] = d
end while
return announce[P].result
```

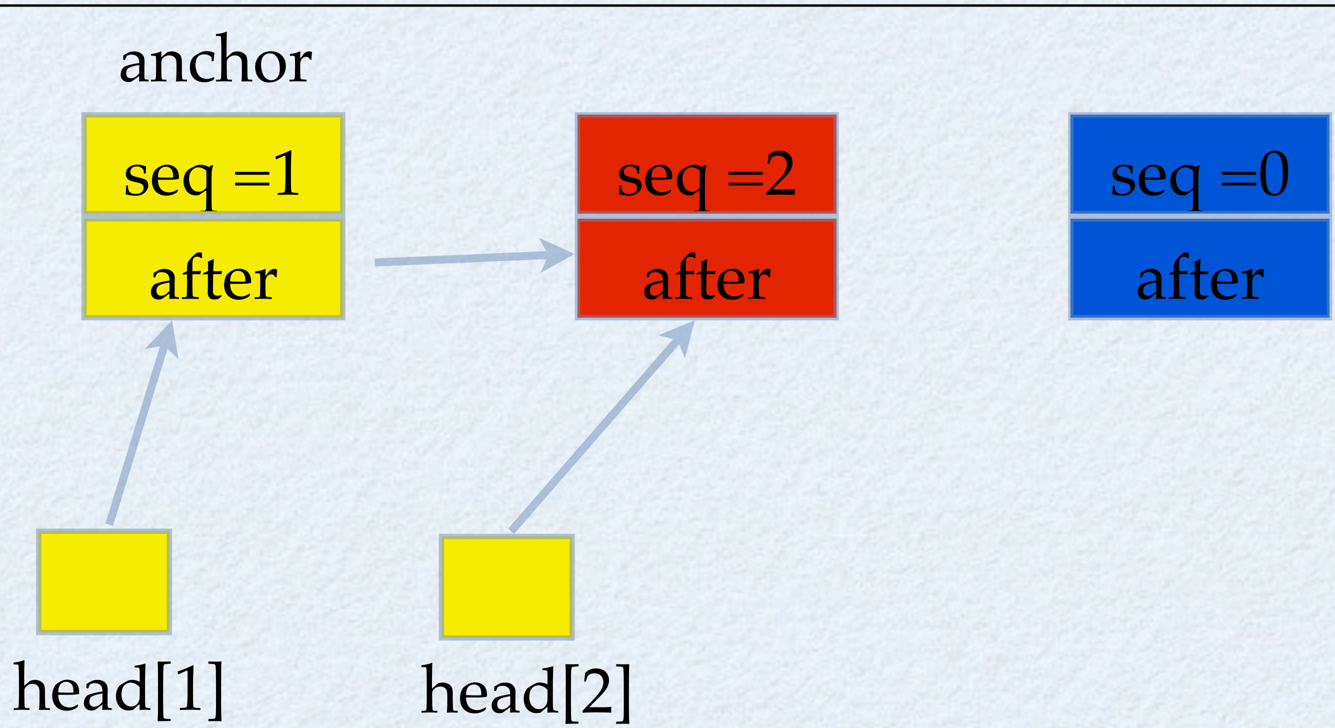


P1 is really slow, P2 will not wait for it. decide ann[1].

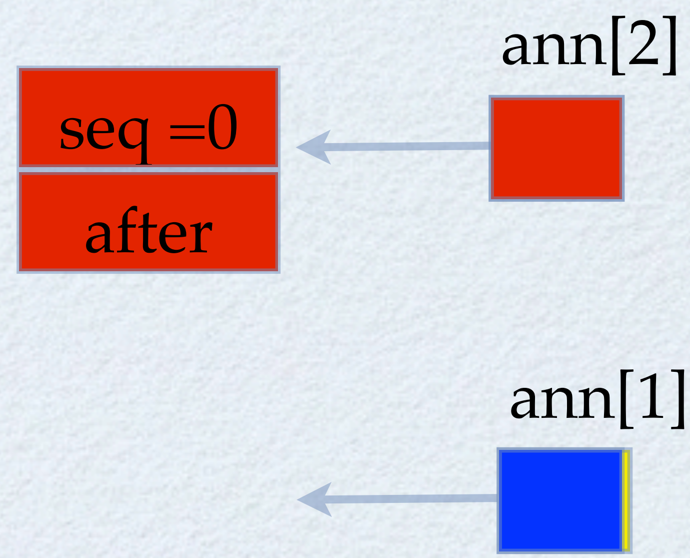


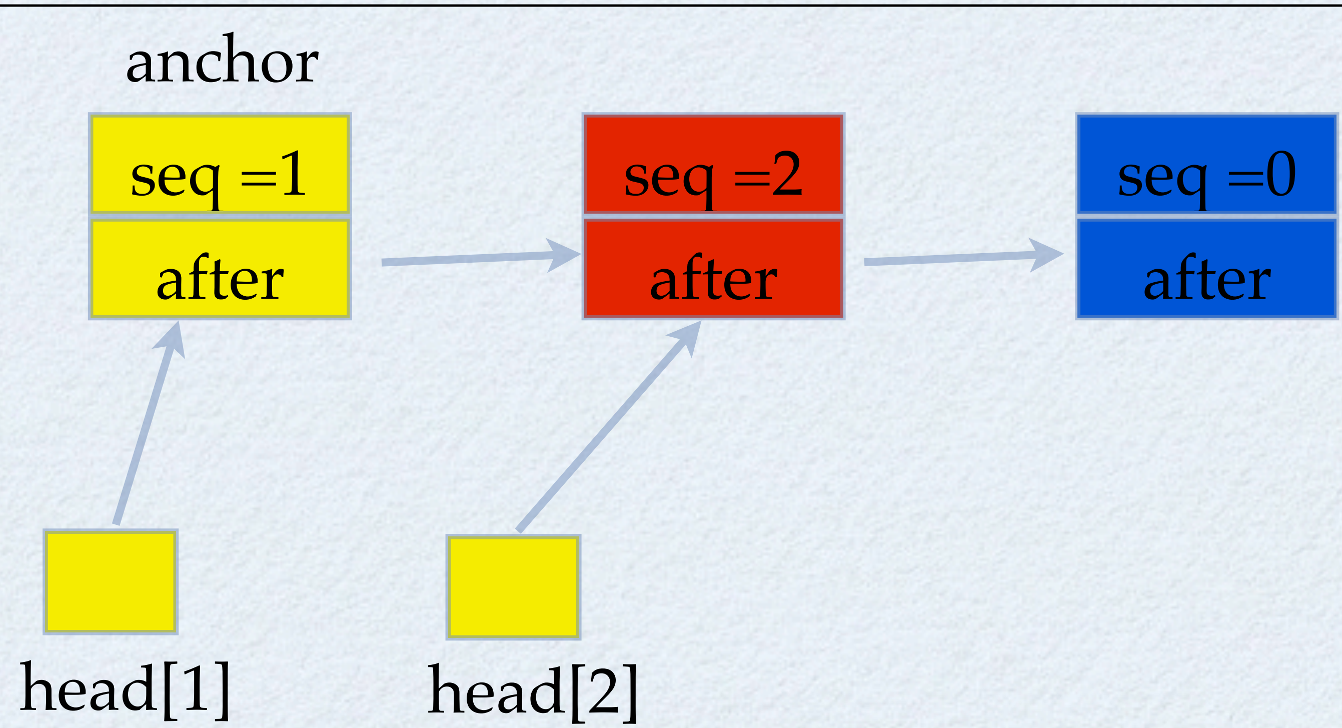
```
initialize the cell with seq = 0
let announce[P] point to it.
head[P] = max{head[1], ..., head[n]}
while announce[P].seq = 0 do
   c = head[P]
   h = announce[c.seq mod n + 1]
   if h.seq = 0 then
      prefer = h
   else
      prefer = announce[P]
   end if
   d = c.after.decide(prefer)
   d.seq = c.seq + 1
   update the field of d according to c.inv, c.new-state
   head[P] = d
end while
return announce[P].result
```



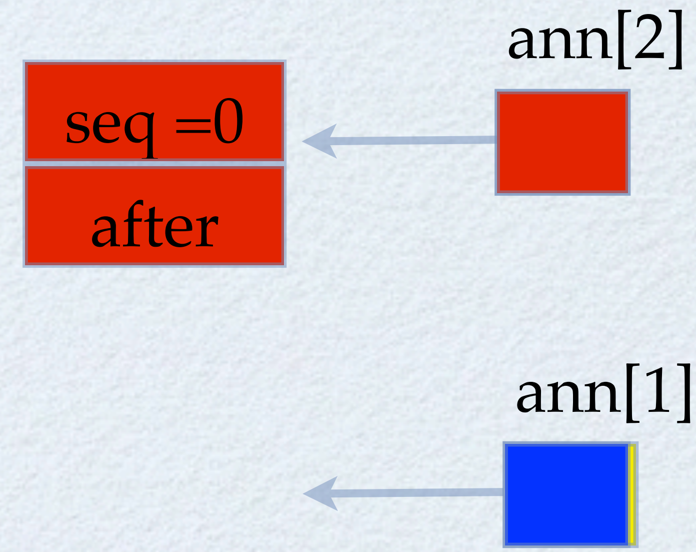


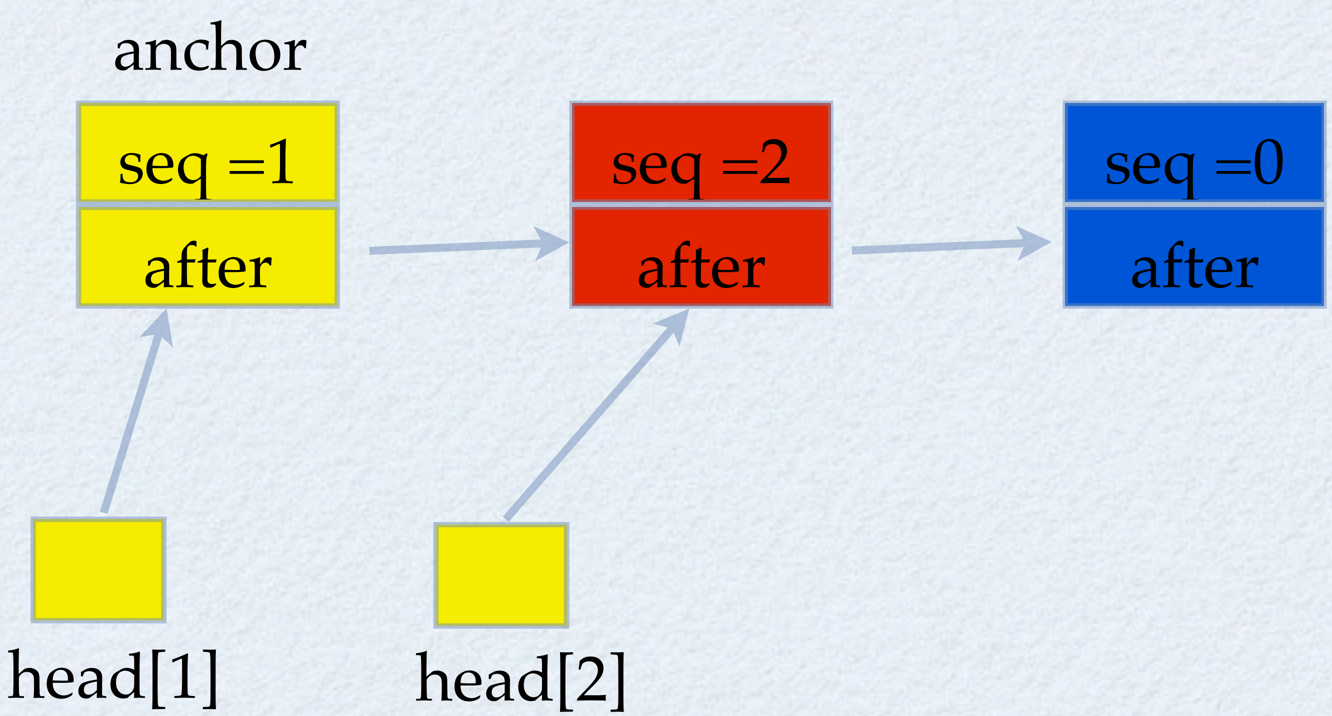
```
initialize the cell with seq = 0
let announce[P] point to it.
head[P] = max{head[1], ..., head[n]}
while  announce[P].seq = 0 do
   c = head[P]
   h = announce[c.seq mod n + 1]
   if h.seq = 0 then
      prefer = h
   else
      prefer = announce[P]
   end if
   d = c.after.decide(prefer)
   d.seq = c.seq + 1
   update the field of d according to c.inv, c.new-state
   head[P] = d
end while
return  announce[P].result
```



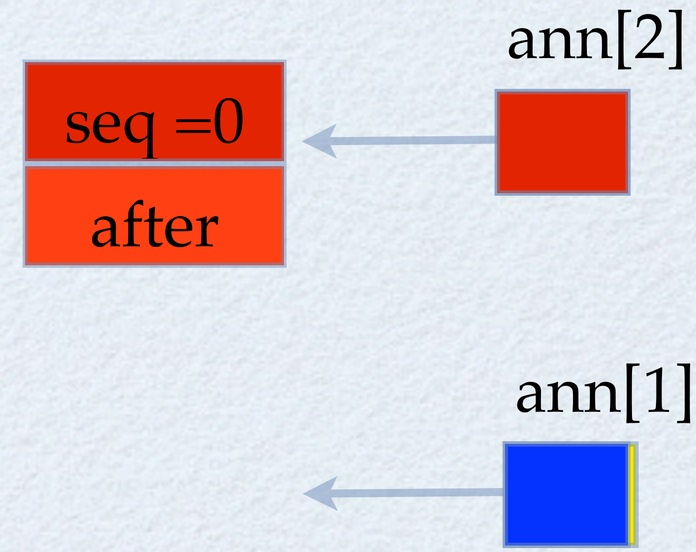


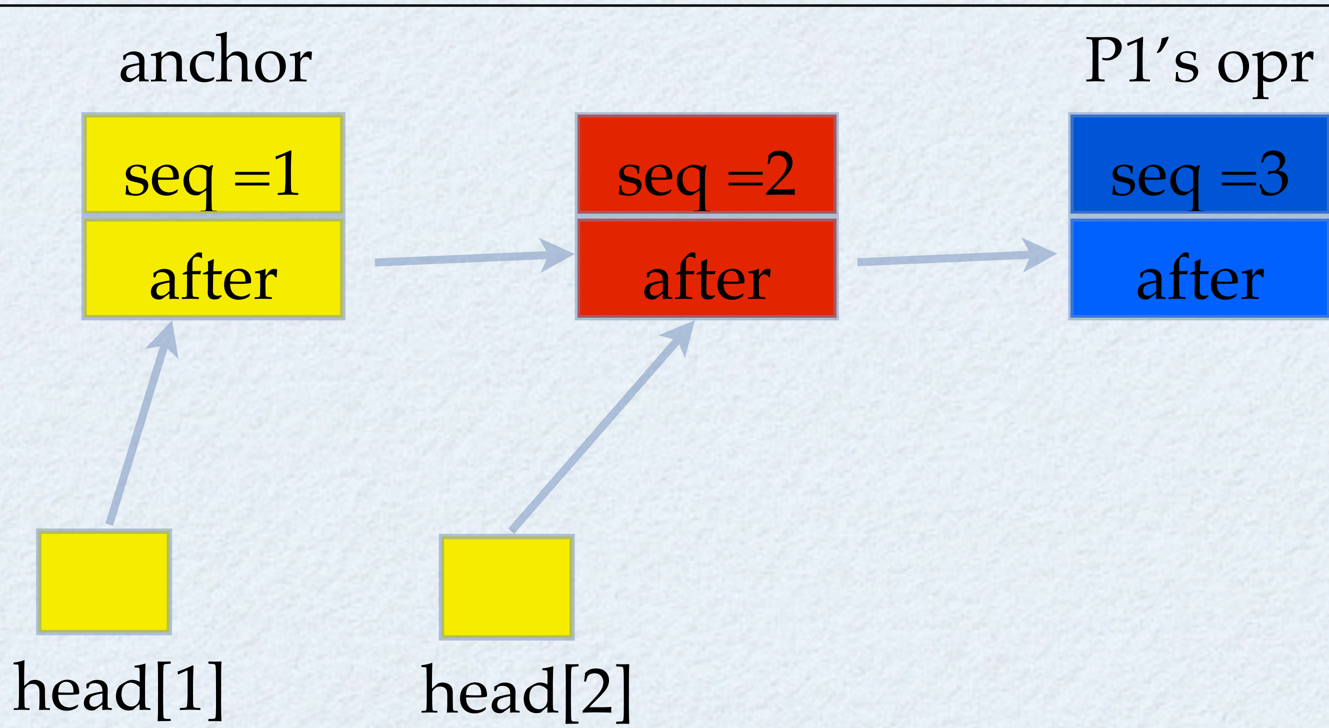
```
initialize the cell with seq = 0
let announce[P] point to it.
head[P] = max{head[1], . . . , head[n]}
while  announce[P].seq = 0 do
   c = head[P]
   h = announce[c.seq mod n + 1]
   if h.seq = 0 then
      prefer = h
   else
      prefer = announce[P]
   end if
   d = c.after.decide(prefer)
   d.seq = c.seq + 1
   update the field of d according to c.inv, c.new-state
   head[P] = d
end while
return  announce[P].result
```



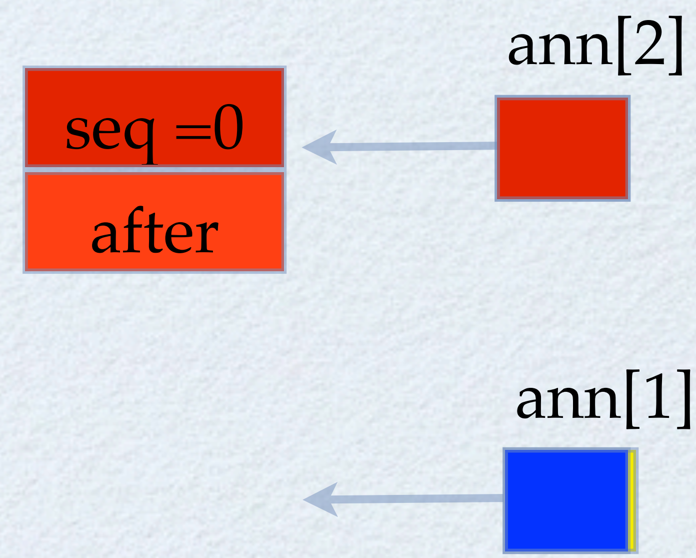


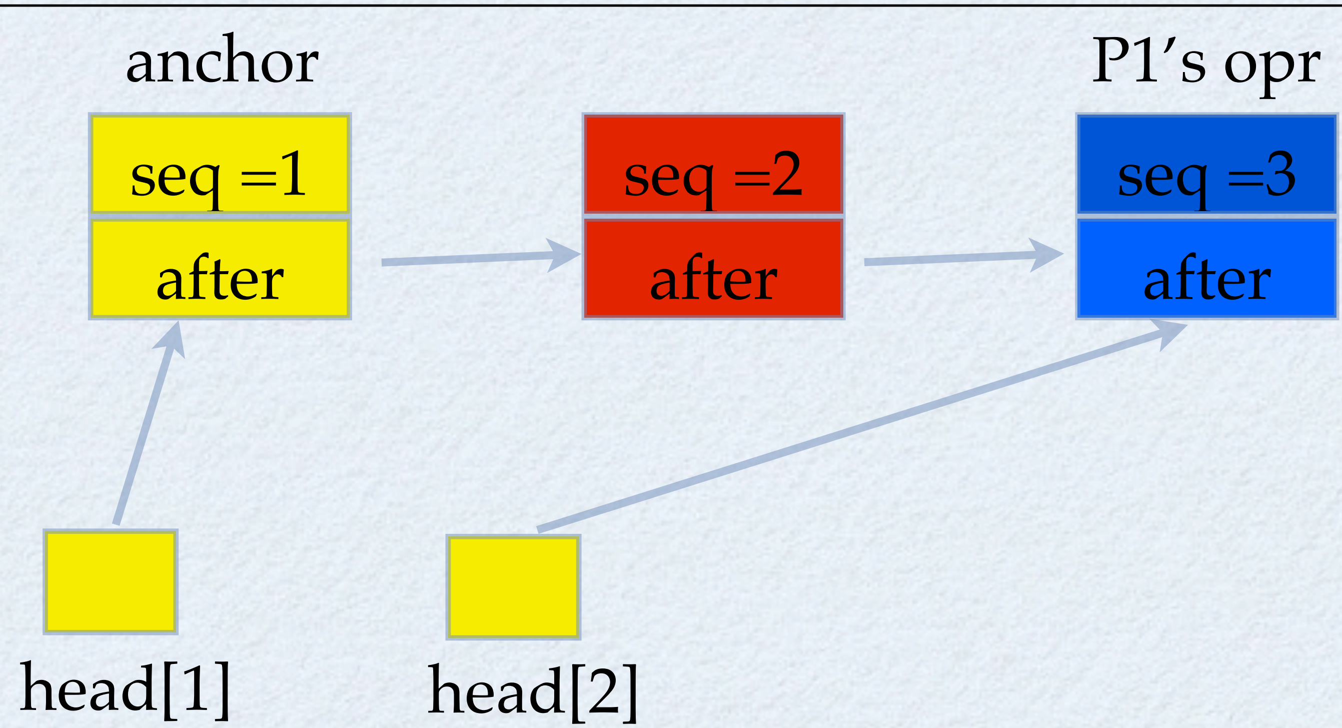
```
initialize the cell with seq = 0
let announce[P] point to it.
head[P] = max{head[1], ..., head[n]}
while announce[P].seq = 0 do
   c = head[P]
   h = announce[c.seq mod n + 1]
   if h.seq = 0 then
      prefer = h
   else
      prefer = announce[P]
   end if
   d = c.after.decide(prefer)
   d.seq = c.seq + 1
   update the field of d according to c.inv, c.new-state
   head[P] = d
end while
return announce[P].result
```



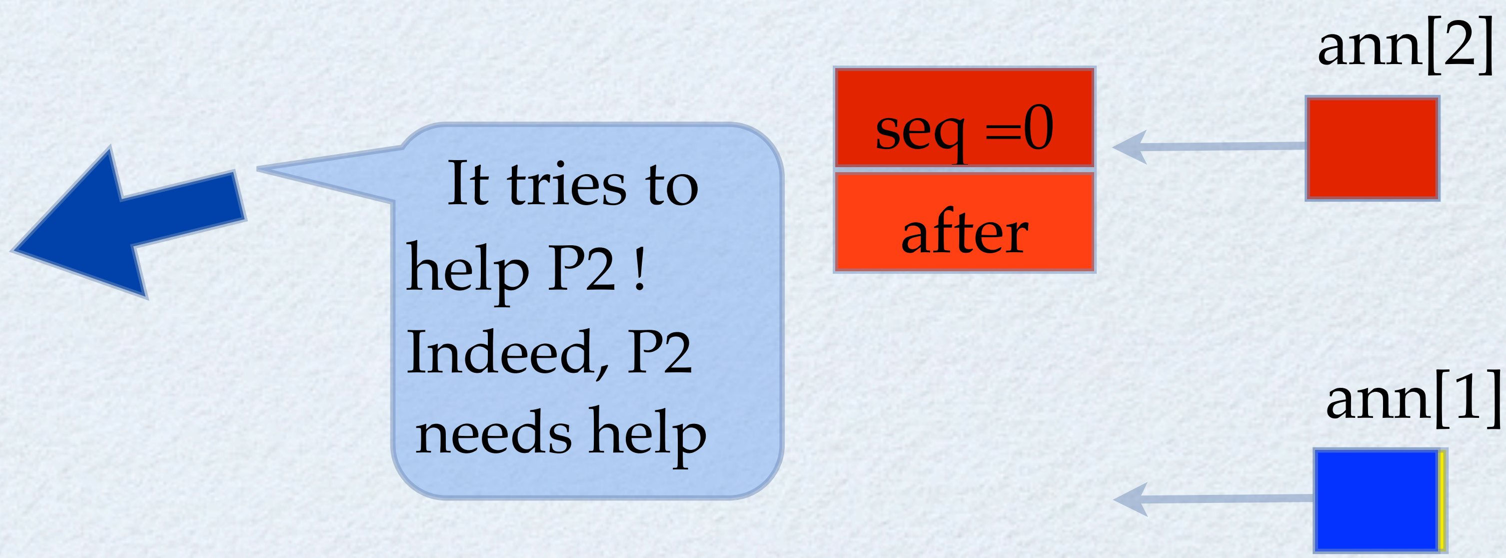


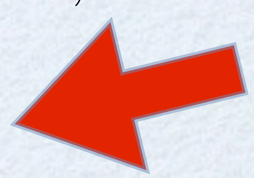
```
initialize the cell with seq = 0
let announce[P] point to it.
head[P] = max{head[1], . . . , head[n]}
while announce[P].seq = 0 do
   c = head[P]
   h = announce[c.seq mod n + 1]
   if h.seq = 0 then
      prefer = h
   else
      prefer = announce[P]
   end if
   d = c.after.decide(prefer)
   d.seq = c.seq + 1
   update the field of d according to c.inv, c.new-state
   head[P] = d
end while
return announce[P].result
```

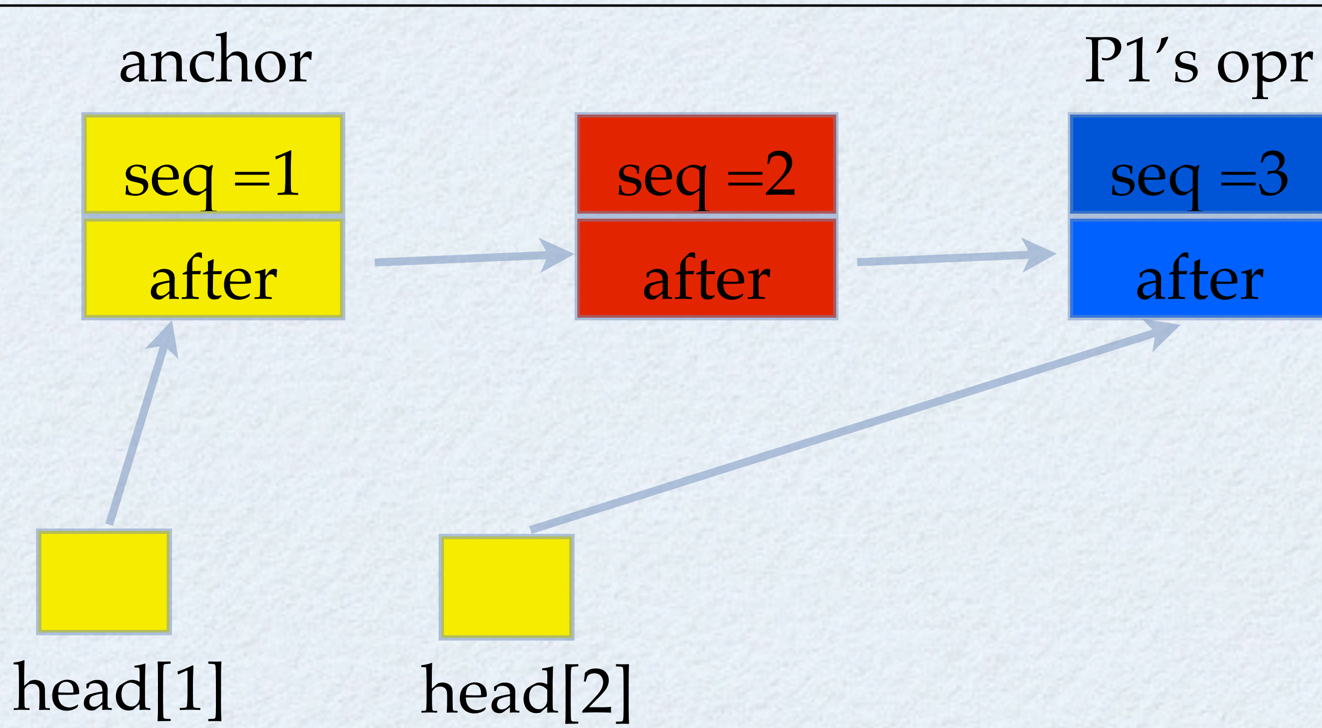




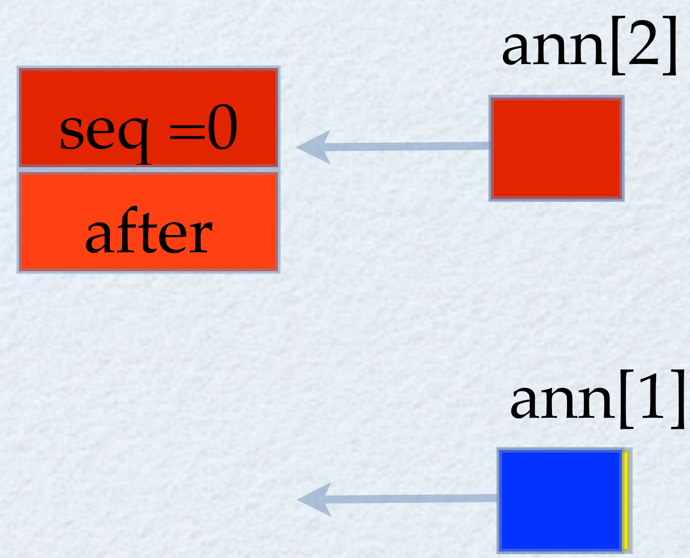
```
initialize the cell with seq = 0
let announce[P] point to it.
head[P] = max{head[1], ..., head[n]}
while  announce[P].seq = 0 do
   c = head[P]
   h = announce[c.seq mod n + 1]
   if h.seq = 0 then
      prefer = h
   else
      prefer = announce[P]
   end if
   d = c.after.decide(prefer)
   d.seq = c.seq + 1
   update the field of d according to c.inv, c.new-state
   head[P] = d
end while
return  announce[P].result
```

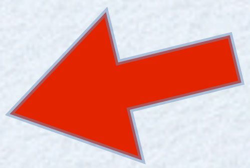


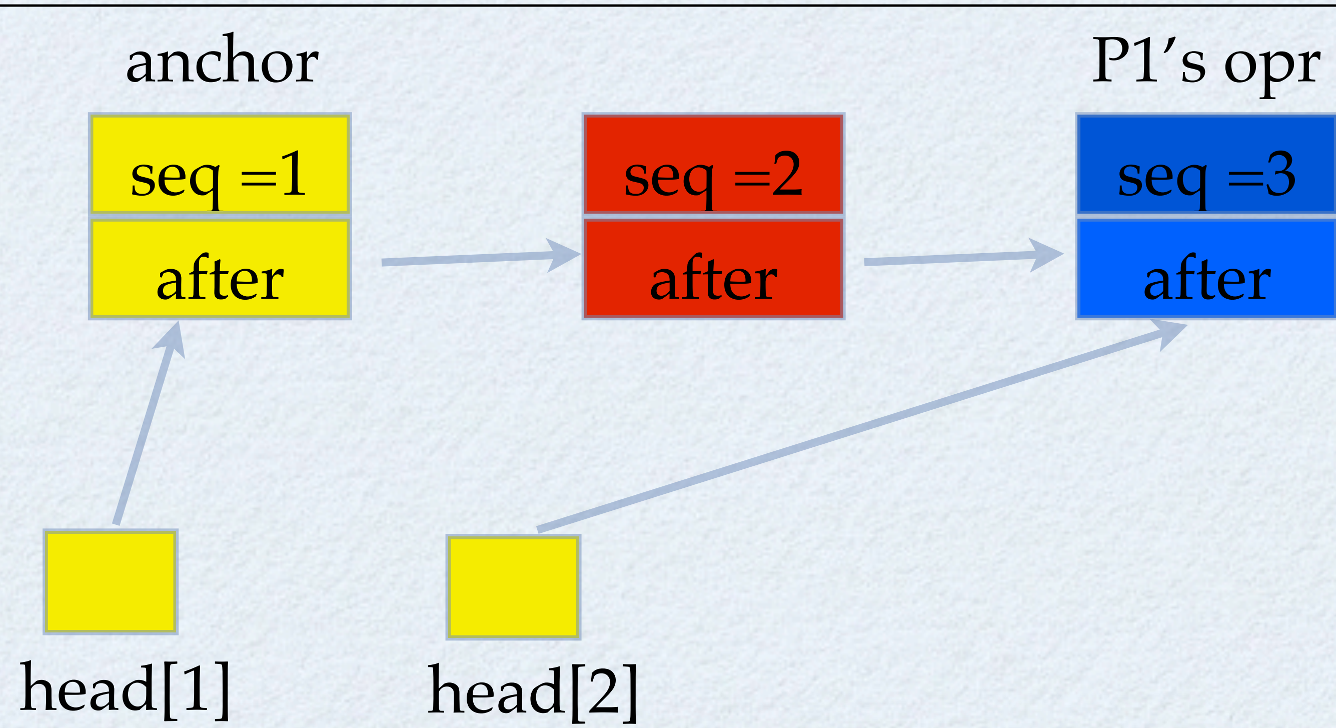




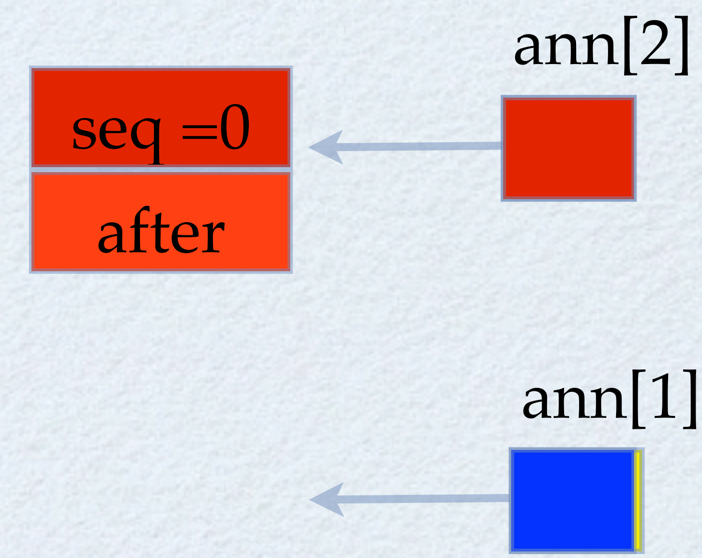
```
initialize the cell with seq = 0
let announce[P] point to it.
head[P] = max{head[1], ..., head[n]}
while announce[P].seq = 0 do
   c = head[P]
   h = announce[c.seq mod n + 1]
   if h.seq = 0 then
      prefer = h
   else
      prefer = announce[P]
   end if
   d = c.after.decide(prefer)
   d.seq = c.seq + 1
   update the field of d according to c.inv, c.new-state
   head[P] = d
end while
return announce[P].result
```



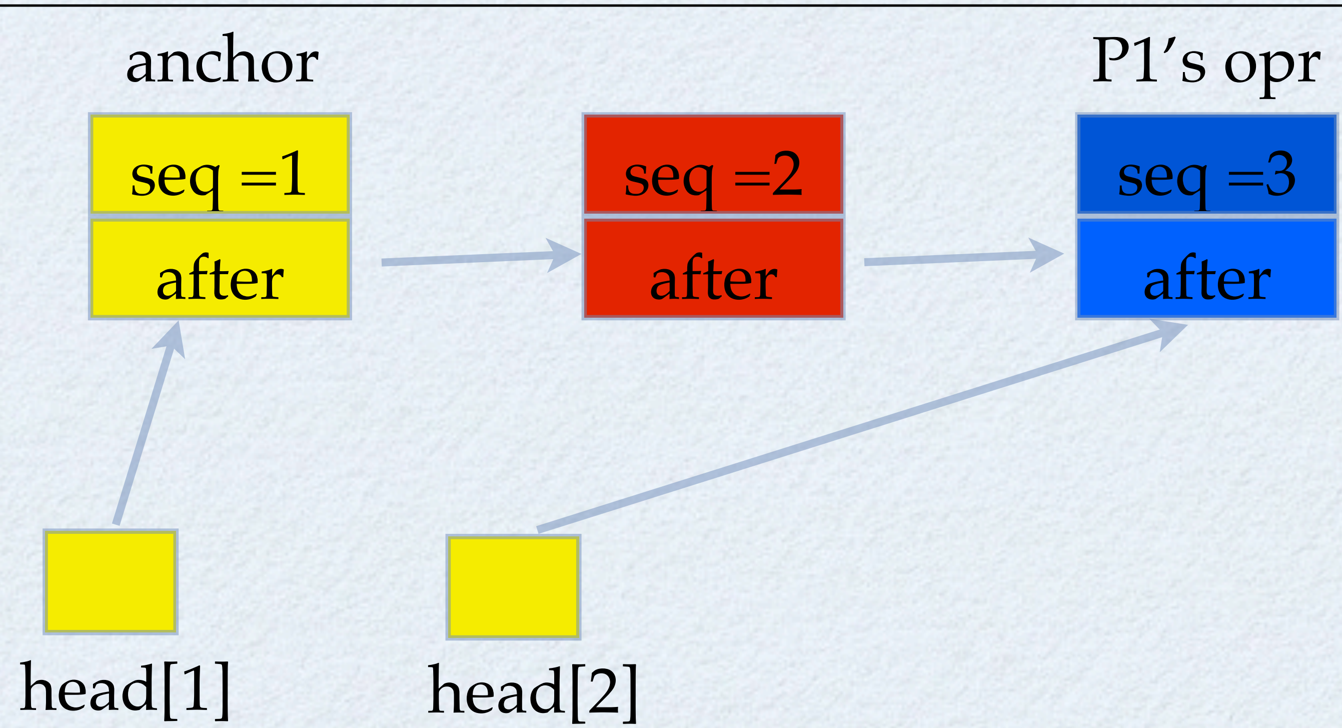




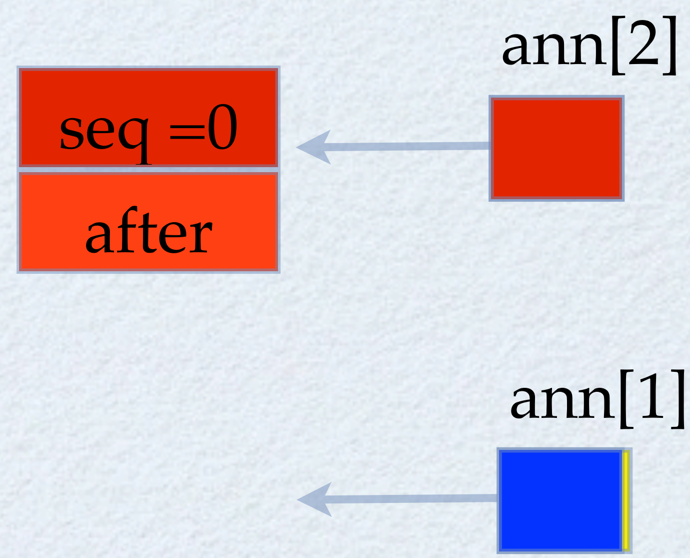
```
initialize the cell with seq = 0
let announce[P] point to it.
head[P] = max{head[1], . . . , head[n]}
while announce[P].seq = 0 do
   c = head[P]
   h = announce[c.seq mod n + 1]
   if h.seq = 0 then
      prefer = h
   else
      prefer = announce[P]
   end if
   d = c.after.decide(prefer)
   d.seq = c.seq + 1
   update the field of d according to c.inv, c.new-state
   head[P] = d
end while
return announce[P].result
```

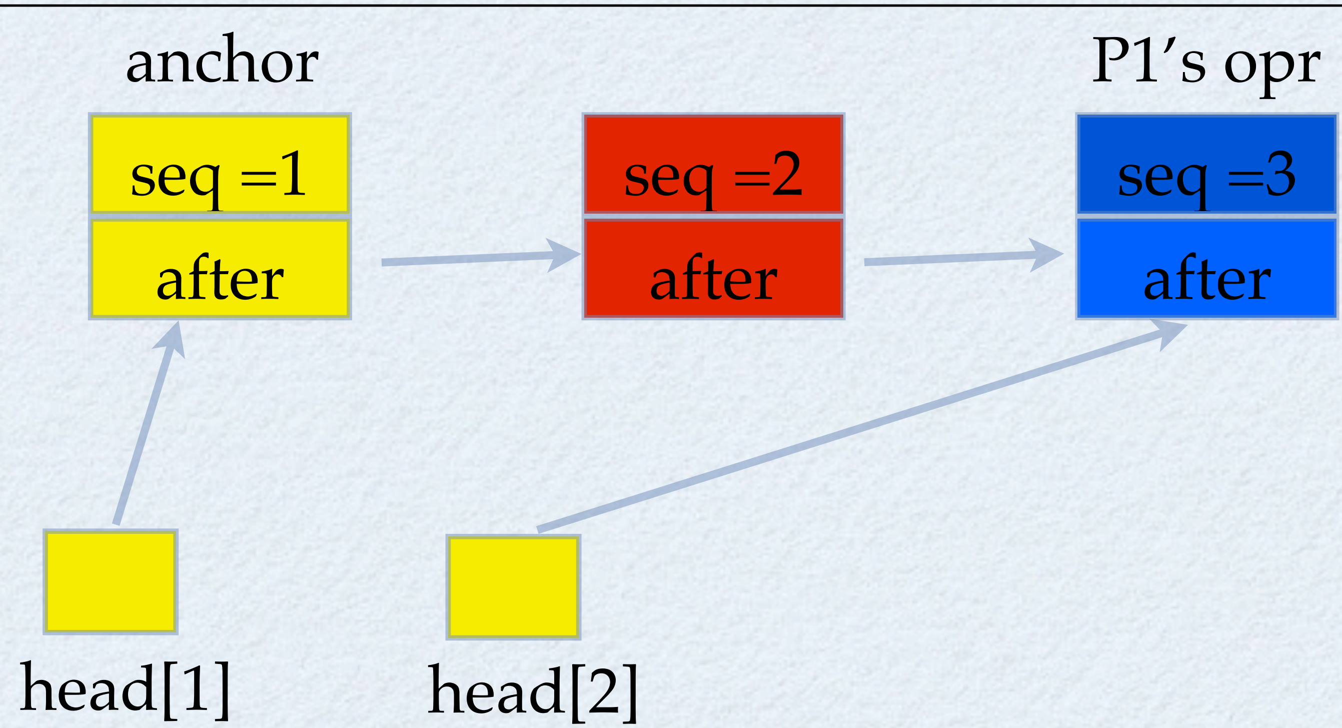


what is the possible result?

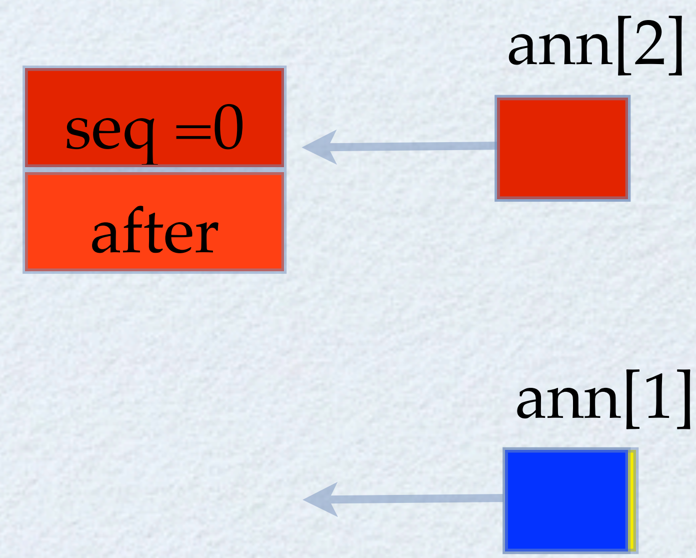


```
initialize the cell with seq = 0
let announce[P] point to it.
head[P] = max{head[1], . . . , head[n]}
while announce[P].seq = 0 do
   c = head[P]
   h = announce[c.seq mod n + 1]
   if h.seq = 0 then
      prefer = h
   else
      prefer = announce[P]
   end if
   d = c.after.decide(prefer)
   d.seq = c.seq + 1
   update the field of d according to c.inv, c.new-state
   head[P] = d
end while
return announce[P].result
```

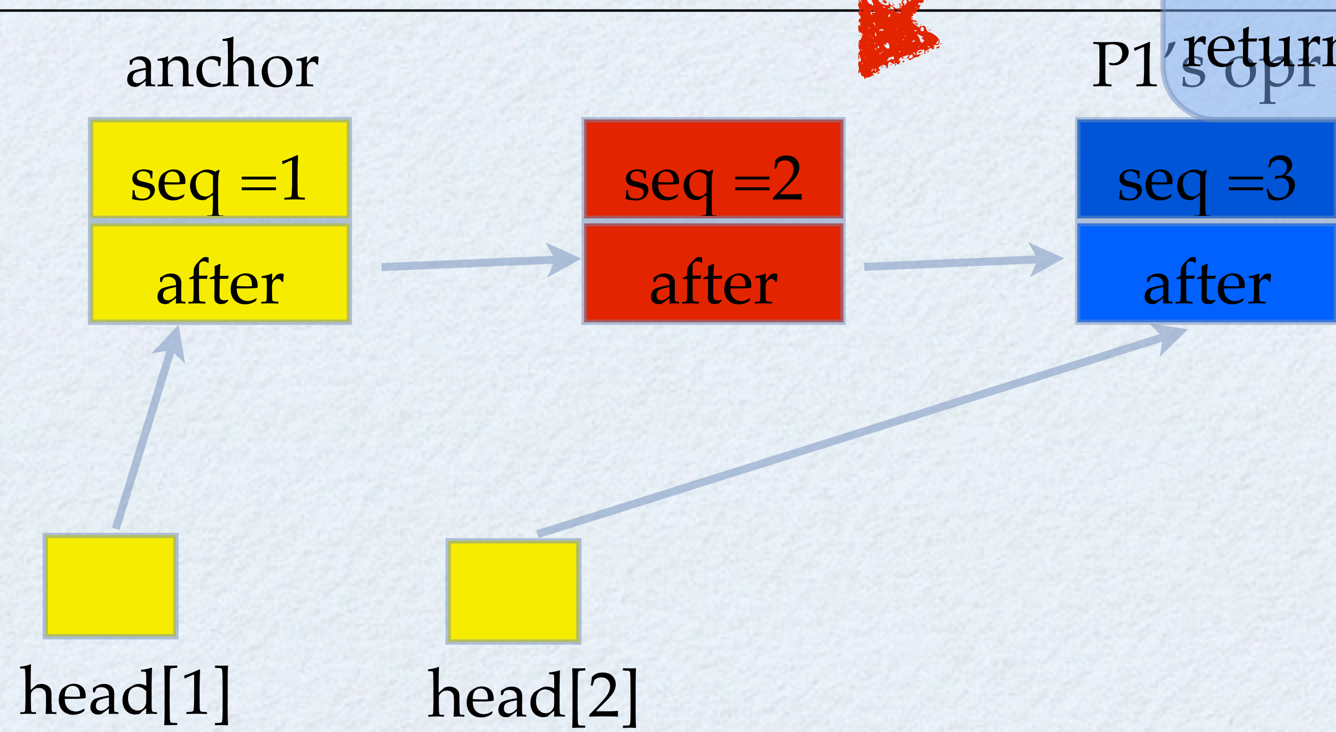




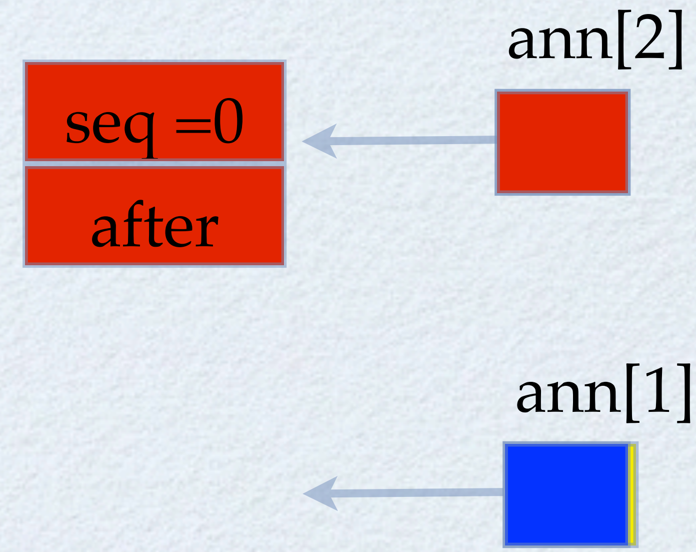
```
initialize the cell with seq = 0
let announce[P] point to it.
head[P] = max{head[1], . . . , head[n]}
while announce[P].seq = 0 do
   c = head[P]
   h = announce[c.seq mod n + 1]
   if h.seq = 0 then
      prefer = h
   else
      prefer = announce[P]
   end if
   d = c.after.decide(prefer)
   d.seq = c.seq + 1
   update the field of d according to c.inv, c.new-state
   head[P] = d
end while
return announce[P].result
```



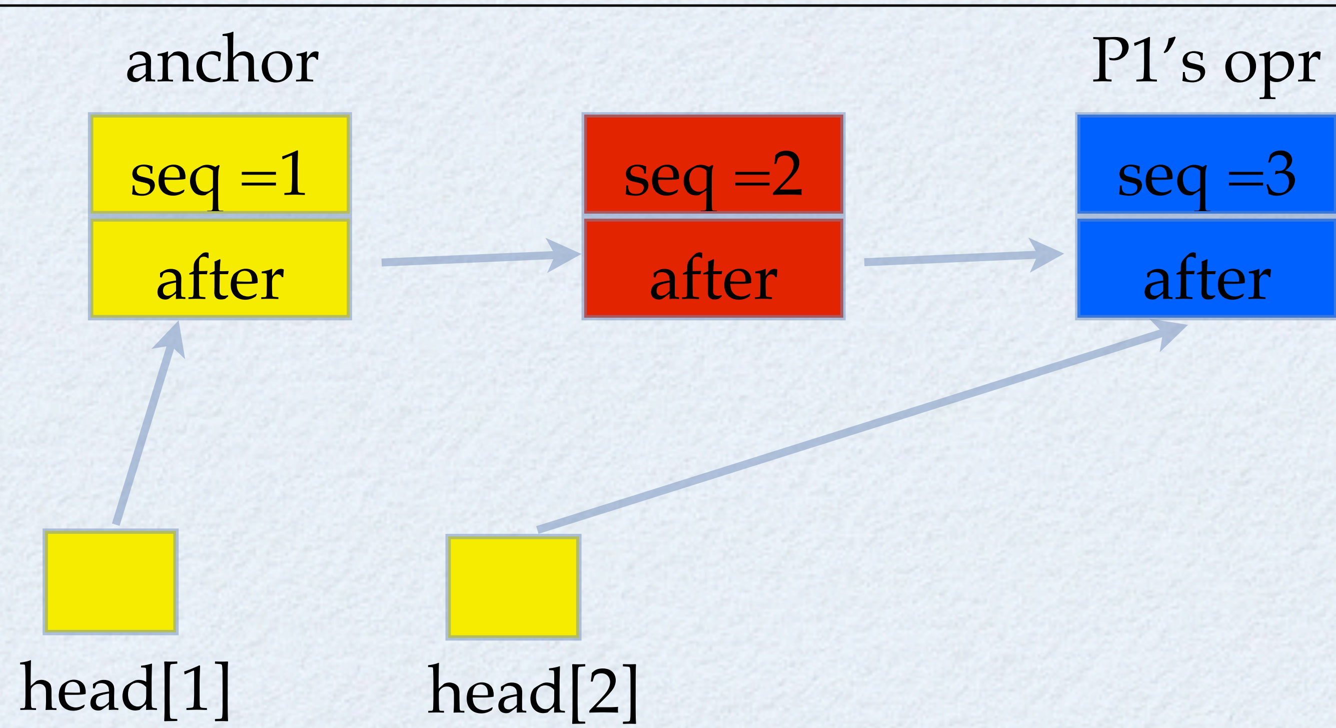
Only the this can be returned!
Since c.after is a consensus object
and now c = head[1] = anchor
anchor.after.decide() has already
returned the value to P2



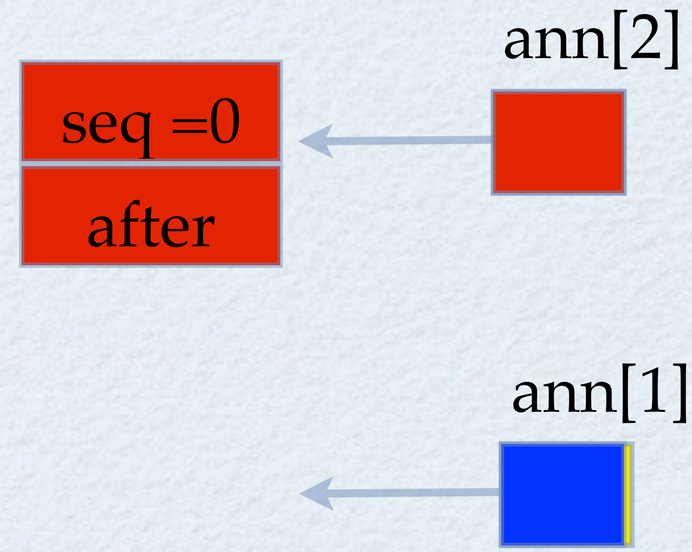
```
initialize the cell with seq = 0
let announce[P] point to it.
head[P] = max{head[1], ..., head[n]}
while announce[P].seq = 0 do
   c = head[P]
   h = announce[c.seq mod n + 1]
   if h.seq = 0 then
      prefer = h
   else
      prefer = announce[P]
   end if
   d = c.after.decide(prefer)
   d.seq = c.seq + 1
   update the field of d according to c.inv, c.new-state
   head[P] = d
end while
return announce[P].result
```



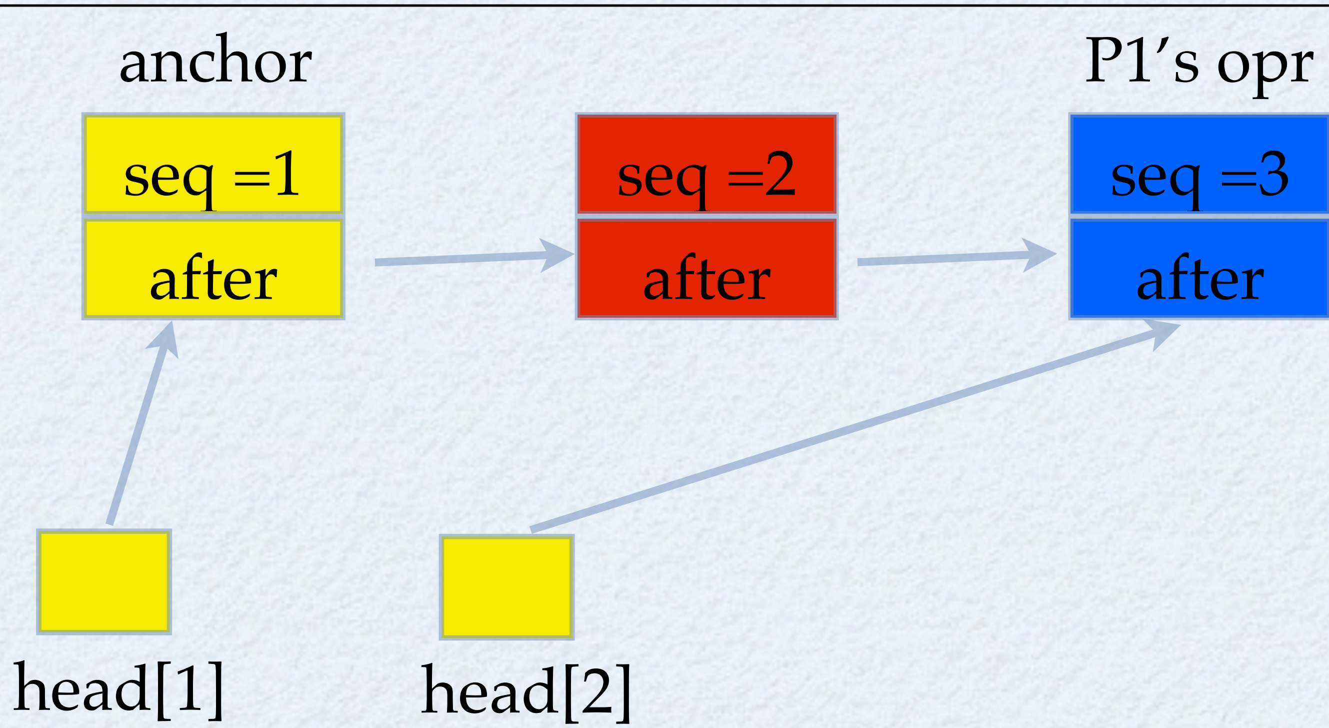
P1 can only rewrite the second cell with the same field value
But it can quit the loop quickly since its cell has been threaded :)



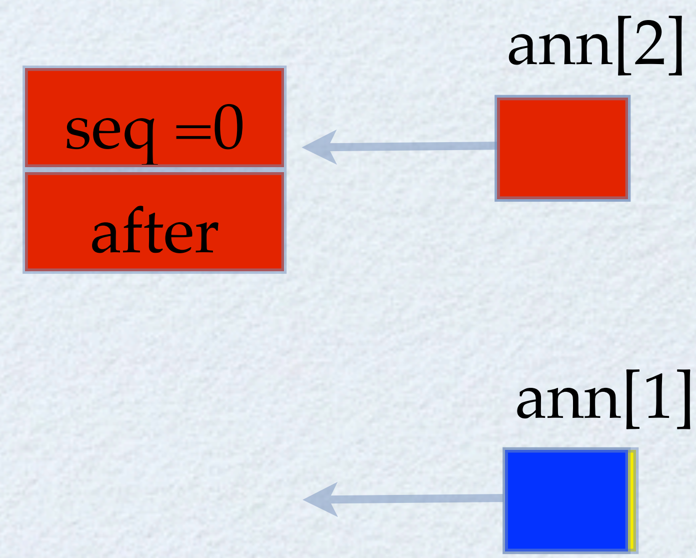
```
initialize the cell with seq = 0
let announce[P] point to it.
head[P] = max{head[1], . . . , head[n]}
while announce[P].seq = 0 do
   c = head[P]
   h = announce[c.seq mod n + 1]
   if h.seq = 0 then
      prefer = h
   else
      prefer = announce[P]
   end if
   d = c.after.decide(prefer)
   d.seq = c.seq + 1
   update the field of d according to c.inv, new-state
   head[P] = d
end while
return announce[P].result
```



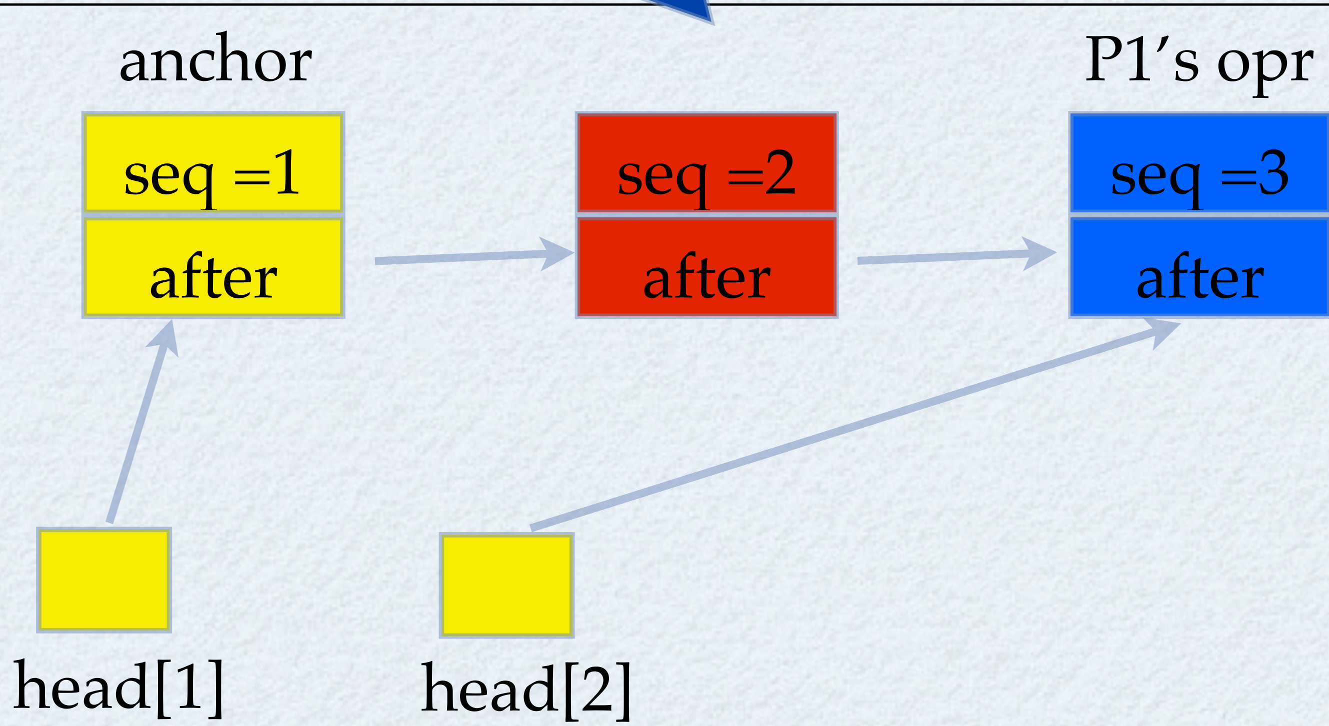
P1 can only rewrite the second cell with the same field value
But it can quit the loop quickly since its cell has been threaded :)



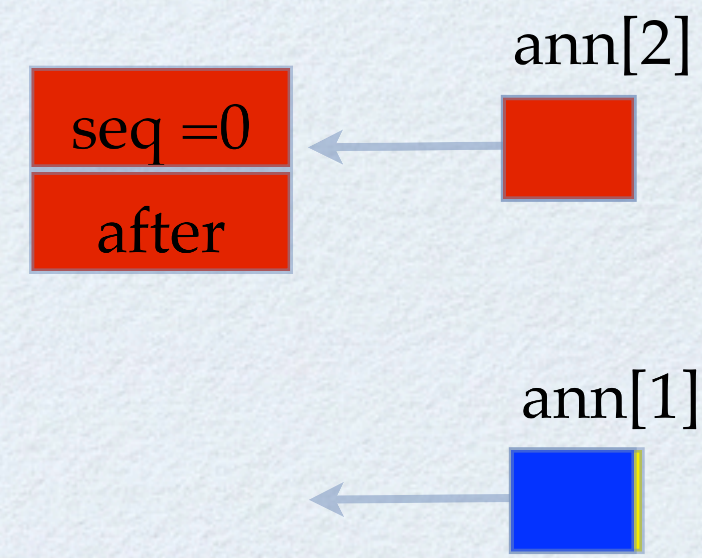
```
initialize the cell with seq = 0
let announce[P] point to it.
head[P] = max{head[1], ..., head[n]}
while announce[P].seq = 0 do
   c = head[P]
   h = announce[c.seq mod n + 1]
   if h.seq = 0 then
      prefer = h
   else
      prefer = announce[P]
   end if
   d = c.after.decide(prefer)
   d.seq = c.seq + 1
   update the field of d according to c.inv, c.new-state
   head[P] = d
end while
return announce[P].result
```



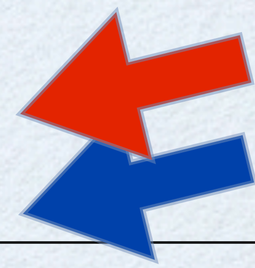
P1 can only rewrite the second cell with the same field value
But it can quit the loop quickly since its cell has been threaded :)

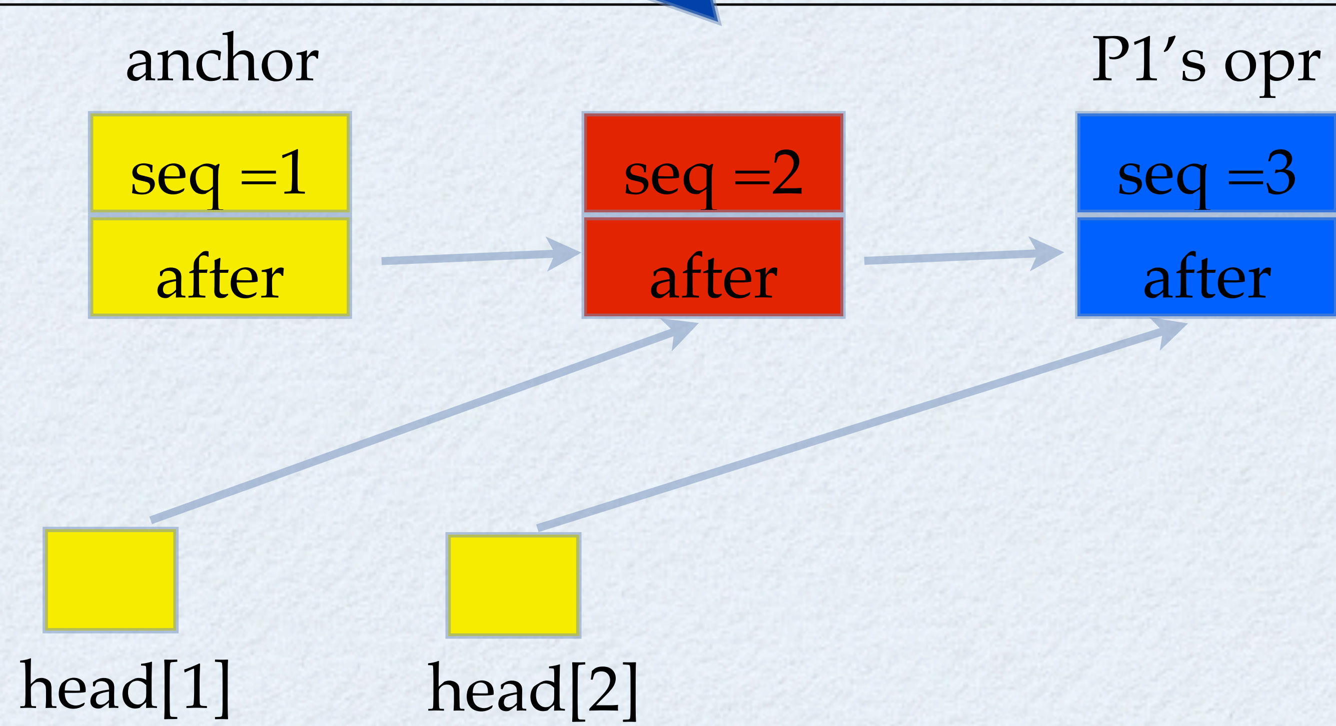


```
initialize the cell with seq = 0
let announce[P] point to it.
head[P] = max{head[1], ..., head[n]}
while announce[P].seq = 0 do
   c = head[P]
   h = announce[c.seq mod n + 1]
   if h.seq = 0 then
      prefer = h
   else
      prefer = announce[P]
   end if
   d = c.after.decide(prefer)
   d.seq = c.seq + 1
   update the field of d according to c.inv, c.new-state
   head[P] = d
end while
return announce[P].result
```



P1 can only rewrite the second cell with the same field value
But it can quit the loop quickly since its cell has been threaded :)





Proof of the Correctness

- Observations
  - non-zero sequence number indicates successful threading
  - the consensus protocols guarantee that the fields of the cells will not be updated with different values.
  - at cell with sequence number k, every thread tries to help thread (k+1) mod n
  - if a cell is announced by thread k+1, after at most n more cells have been threaded
    - everyone will check if process k+1 needs help
    - everyone will help

More Practical Constructions

- New universal construction
  - P. Choung, F. Ellen, V. Ramachandran "A universal construction for wait-free transaction friendly data structure".
  - implements any shared data structure with $\theta(s+p)$ space, where s is the size of the shared data structure and p is the number of processes.
  - uses only CAS and registers as base objects.
- Ad-hoc wait-free data structures
  - lower overhead by using a purpose-built construction

Conclusions

- Wait-free synchronization is possible, practical, and useful!

References

- Maurice Herlihy, “Wait-free synchronization”
- Lynch and Tuttle, “An Introduction to Input/ Output Automata”
- Maurice Herlihy & Nir Shavit “Lecture notes of Art of multiprocessor computing.
- L. Lamport “How to make a multiprocessor computer that correctly executes multiprocess programs.”

Linearizability vs Sequential Consistency

- Linearizability is stronger than sequential consistency.
- sequential consistency is not composable:(not a local property)
  - If two objects are both sequential consistent, the composition of them might be not.
- linearizability has composability.
  - We only need to study isolated object