Herlihy’s Hierarchy

Concurrent Algorithms 2018

Karolos Antoniadis

Herlihy's Hierarchy

and why it collapses in practice …

Concurrent Algorithms 2018

Karolos Antoniadis

Roadmap

- Herlihy's hierarchy
- Herlihy's hierarchy collapses in practice
- Universal construction without *CAS*
- Conclusion

Herlihy's Hierarchy

Herlihy's "Wait-free synchronization" (TOPLAS '91)

The **consensus number** of an object `X` is the largest *k* such that:

> there exists an algorithm that implements **wait-free** consensus among *k* processes using any number of `X` instances and `registers`.

Herlihy's Hierarchy

object	consensus number
	1
	2
…	
	∞

Herlihy’s Hierarchy

object	consensus number
register	**1**
fetch-and-increment, test-and-set	**2**
…	
CAS, LL/SC	**∞**

Hierarchy’s Selling Point

“We have tried to suggest here that the resulting theory has a rich structure, yielding a number of unexpected results with **consequences** for algorithm design, **multiprocessor architectures**, and real-time systems.”

Wait-Free Synchronization (TOPLAS ’91)

Hierarchy's Selling Point

"**Imagine you are in charge of designing a new multiprocessor. What kinds of atomic instructions should you include?** *[…]* Supporting them all would be complicated and inefficient, **but supporting the wrong ones could make it difficult or even impossible to solve important synchronization problems**."

The Art of Multiprocessor Programming - Revised Reprint (2012)

Therefore …

- Multiprocessors **should** provide objects with infinite consensus number (e.g., *CAS*).
- Herlihy's hierarchy had an impact in industry.

Hierarchy's Impact (i)

"*[...]* **the unsolvability of consensus in asynchronous shared memory systems** *[...]* **led to manufacturers including more powerful primitives such as compare&swap into their architectures.**"

Impossibility Results for Distributed Computing (2012)

Hierarchy’s Impact (ii)

“*[…]* multiprocessor architects abandoned operations with low consensus number *[…]* **I wouldn’t be surprised to learn that these architects were influenced, at least in part, by Herlihy’s discovery that, from the perspective of wait-free synchronisation, much more is possible with operations such as compare-and-swap** *[…]* than with operations such as test-and-set”

A Quarter-Century of Wait-Free Synchronization (SIGACT News ’15)

Then, in 2016 …

A Complexity-Based Hierarchy for Multiprocessor Synchronization
(PODC ’16)

Then, in 2016 …

“However, key to this hierarchy is treating these instructions as distinct objects, **an approach that is far from the real-world**, where multiprocessor programs apply synchronization instructions to collections of arbitrary memory locations. **We were surprised to realize that, when considering instructions applied to memory locations, the computability based hierarchy collapses.**”

A Complexity-Based Hierarchy for Multiprocessor Synchronization (PODC ’16)

Roadmap

- Herlihy's hierarchy
- Herlihy's hierarchy collapses in practice
- Universal construction without *CAS*
- Conclusion

Herlihy’s Hierarchy

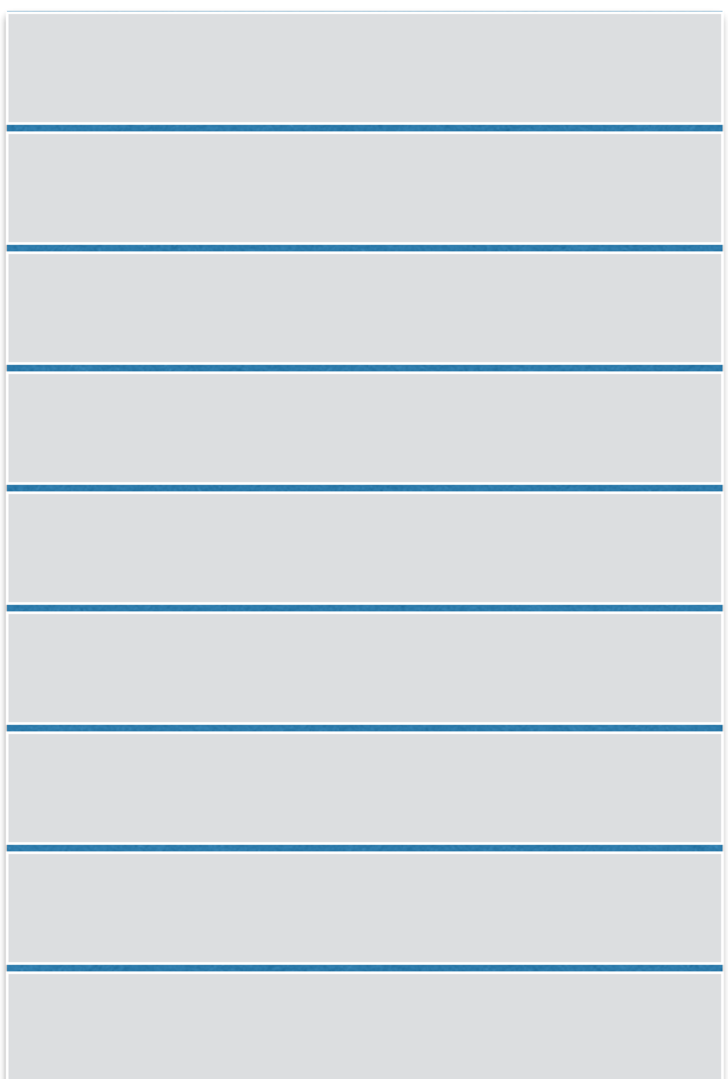
Herlihy’s hierarchy is treating instructions as **distinct objects**.

objects

The hierarchy considers objects that only support some restricted set of operations.

For example, `fetch-and-add`, `CAS`

Memory - In reality …

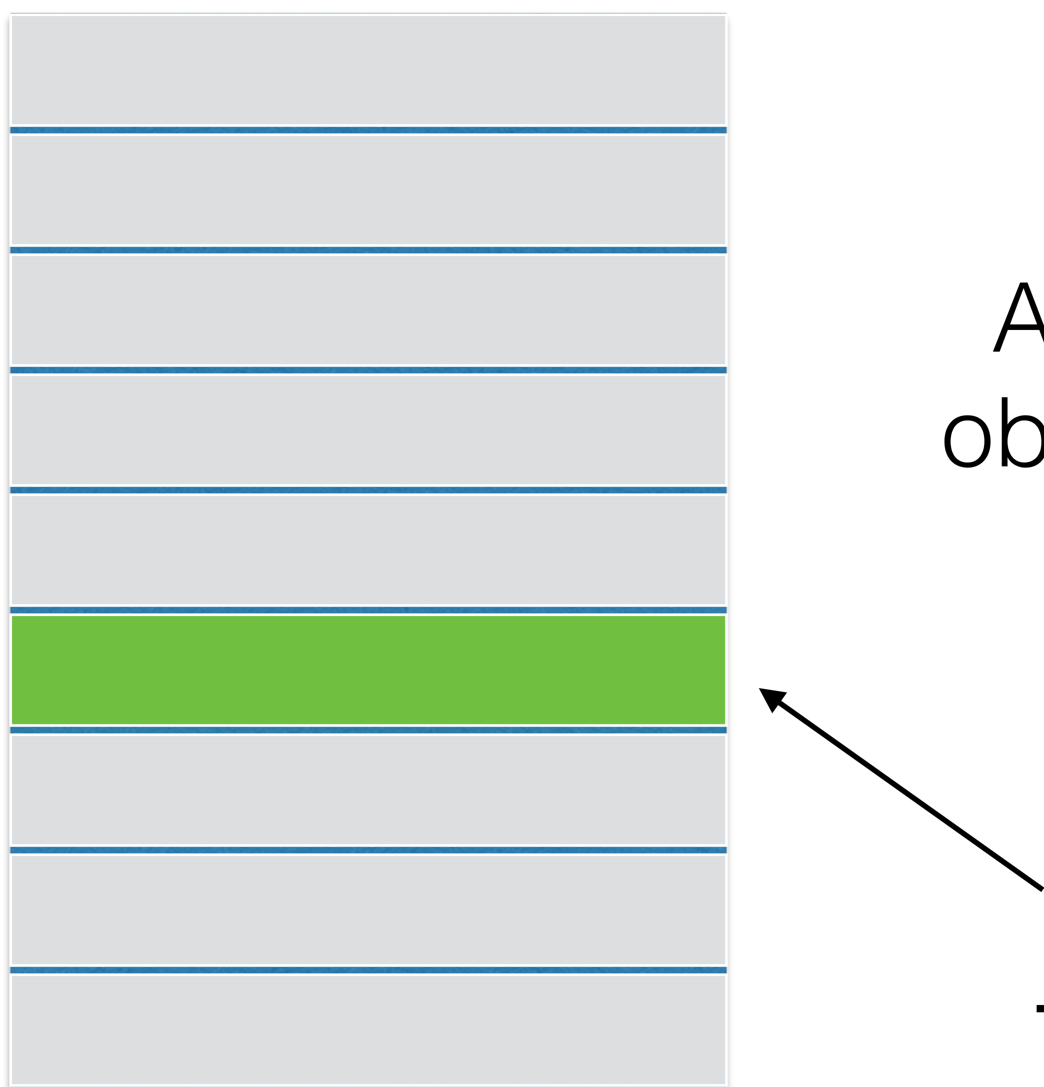


memory

We don't have distinct objects.
We have memory locations.

We don't have memory locations that **only** support `CAS` or **only** support `fetch-and-increment`.

Memory - In reality …



memory

A memory location is a single object that supports a multitude of operations.

You can apply:
`read`, `write`, `CAS`, `fetch-and-increment`, `xor`, `not`, …

Example: `test-and-set`

```
test-and-set()
    ret := x
    if (x = 0) x := 1
    return ret
```

Example: `test-and-set`

```
test-and-set()
   ret := x
   if (x = 0) x := 1
   return ret
```

```
// i in {0, 1}
propose(v)
   reg[i] := v
   res := test-and-set()
   if (res = 0) return v
   else return reg[1 - i]
```

Example: `fetch-and-add2`

```
fetch-and-add2()
    ret := x
    x := x + 2
    return ret
```

Example: `fetch-and-add2`

```
fetch-and-add2()
   ret := x
   x := x + 2
   return ret
```

```
// i in {0, 1}
propose(v)
   reg[i] := v
   res := fetch-and-add2()
   if (res = 2) return v
   else return reg[1 - i]
```

`test-and-set` + `fetch-and-add2` = ?

Having objects that support
either `test-and-set`
or `fetch-and-add2`
we cannot solve consensus for > 2 processes.

Having one object that supports
both `test-and-set`
and `fetch-and-add2` we can!

test-and-set + fetch-and-add2 = ∞

```
propose(v)
    if (v = 1)
        a := X.test-and-set()
        if (a = 0 or a mod 2 = 1)
            return 1
        else
            return 0
    else // v = 0
        a := X.fetch-and-add2()
        if (a mod 2 = 0)
            return 0
        else
            return 1
```

test-and-set + fetch-and-add2 = ∞

```
propose(v)
    if (v = 1)
        a := X.test-and-set()
        if (a = 0 or a mod 2 = 1)
            return 1
        else
            return 0
    else // v = 0
        a := X.fetch-and-add2()
        if (a mod 2 = 0)
            return 0
        else
            return 1
```

First process that executes `test-and-set` or `fetch-and-add` “wins” the consensus.

Space Hierarchy

Instructions $\mathcal{I}$	$\mathcal{SP}(\mathcal{I}, n)$
$\{read(), test\text{-}and\text{-}set()\}$, $\{read(), write(1)\}$	∞
$\{read(), write(1), write(0)\}$	n (lower), $O(n \log n)$ (upper)
$\{read(), write(x)\}$	n
$\{read(), test\text{-}and\text{-}set(), reset()\}$	$\Omega(\sqrt{n})$ (lower), $O(n \log n)$ (upper)
$\{read(), swap(x)\}$	$\Omega(\sqrt{n})$ (lower), $n-1$ (upper)
$\{\ell\text{-}buffer\text{-}read(), \ell\text{-}buffer\text{-}write(x)\}$	$\lceil \frac{n-1}{\ell} \rceil$ (lower), $\lceil \frac{n}{\ell} \rceil$ (upper)
$\{read(), write(x), increment()\}$ $\{read(), write(x), fetch\text{-}and\text{-}increment()\}$	2 (lower), $O(\log n)$ (upper)
$\{read\text{-}max(), write\text{-}max(x)\}$	2
$\{compare\text{-}and\text{-}swap(x, y)\}$ $\{read(), set\text{-}bit(x)\}$ $\{read(), add(x)\}$, $\{read(), multiply(x)\}$ $\{fetch\text{-}and\text{-}add(x)\}\}$, $\{fetch\text{-}and\text{-}multiply(x)\}$	1

Table 1: Space Hierarachy

SP(I, n)*:* minimum number of memory locations supporting ***I*** that are needed to solve ***n***-consensus.

Space Hierarchy

Instructions $\mathcal{I}$	$\mathcal{SP}(\mathcal{I}, n)$
$\{read(), test\text{-}and\text{-}set()\}$, $\{read(), write(1)\}$	∞
$\{read(), write(1), write(0)\}$	n (lower), $O(n \log n)$ (upper)
$\{read(), write(x)\}$	n
$\{read(), test\text{-}and\text{-}set(), reset()\}$	$\Omega(\sqrt{n})$ (lower), $O(n \log n)$ (upper)
$\{read(), swap(x)\}$	$\Omega(\sqrt{n})$ (lower), $n - 1$ (upper)
$\{\ell\text{-}buffer\text{-}read(), \ell\text{-}buffer\text{-}write(x)\}$	$\lceil \frac{n-1}{\ell} \rceil$ (lower), $\lceil \frac{n}{\ell} \rceil$ (upper)
$\{read(), write(x), increment()\}$ $\{read(), write(x), fetch\text{-}and\text{-}increment()\}$	2 (lower), $O(\log n)$ (upper)
$\{read\text{-}max(), write\text{-}max(x)\}$	2
$\{compare\text{-}and\text{-}swap(x, y)\}$ $\{read(), set\text{-}bit(x)\}$ $\{read(), add(x)\}$, $\{read(), multiply(x)\}$ $\{fetch\text{-}and\text{-}add(x)\}\}$, $\{fetch\text{-}and\text{-}multiply(x)\}$	1

Table 1: Space Hierarachy

SP(I, n)*:* minimum number of memory locations supporting ***I*** that are needed to solve ***n***-consensus.

Space Hierarchy

Instructions $\mathcal{I}$	$\mathcal{SP}(\mathcal{I}, n)$
$\{read(), test\text{-}and\text{-}set()\}$, $\{read(), write(1)\}$	∞
$\{read(), write(1), write(0)\}$	n (lower), $O(n \log n)$ (upper)
$\{read(), write(x)\}$	n
$\{read(), test\text{-}and\text{-}set(), reset()\}$	$\Omega(\sqrt{n})$ (lower), $O(n \log n)$ (upper)
$\{read(), swap(x)\}$	$\Omega(\sqrt{n})$ (lower), $n-1$ (upper)
$\{\ell\text{-}buffer\text{-}read(), \ell\text{-}buffer\text{-}write(x)\}$	
$\{read(), write(x), increment()\}$ $\{read(), write(x), fetch\text{-}and\text{-}increment()\}$	
$\{read\text{-}max(), write\text{-}max(x)\}$	
$\{compare\text{-}and\text{-}swap(x, y)\}$ $\{read(), set\text{-}bit(x)\}$ $\{read(), add(x)\}$, $\{read(), multiply(x)\}$ $\{fetch\text{-}and\text{-}add(x)\}\}$, $\{fetch\text{-}and\text{-}multiply(x)\}$	1

Table 1: Space Hierarachy

Space hierarchy asks for an obstruction-free solution to consensus.

SP(I, n): minimum number of memory locations supporting ***I*** that are needed to solve ***n***-consensus.

Roadmap

- Herlihy's hierarchy
- Herlihy's hierarchy collapses in practice
- Universal construction without *CAS*
- Conclusion

Log Object

Log object supports 2 operations:

`append(item)` // appends item to the log

`getLog()` // returns appended items in order

Log object $\iff$ Universal Construction

Towards Reduced Instruction Sets for Synchronization (DISC ’17)

Universal Construction

Local objects

```
localCopy := initState
appliedOps := ∅
```

Shared object

```
log := emptyLog
```

```
do(operation)
    log.append(operation)
    allOps := log.getLog()
    for op in allOps:
        if (op not in appliedOps)
            res := localCopy.apply(op)
            appliedOps.add(op)
            if (operation = op)
                return res
```

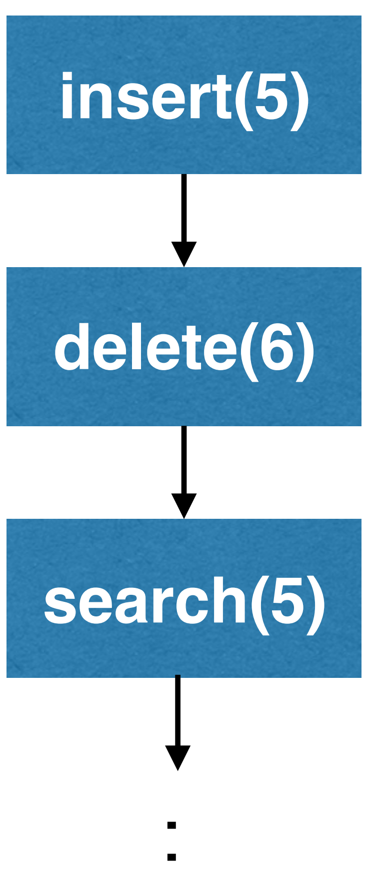
Idea

Order all operations using the *log* object.

Execute them in this order.

Universal Construction

log



Local objects

```
localCopy := initState
appliedOps := ∅
```

Shared object

```
log := emptyLog
```

```
do(operation)
    log.append(operation)
    allOps := log.getLog()
    for op in allOps:
        if (op not in appliedOps)
            res := localCopy.apply(op)
            appliedOps.add(op)
            if (operation = op)
                return res
```

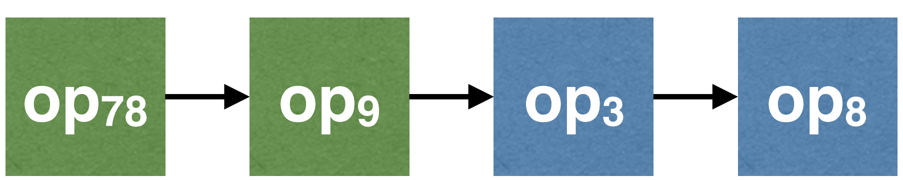
Universal Construction

process p_0

```
do(operation)
    log.append(operation)
    allOps := log.getLog()
    for op in allOps:
        if (op not in appliedOps)
            res := localCopy.apply(op)
            appliedOps.add(op)
            if (operation = op)
                return res
```

`localCopy`

`log`



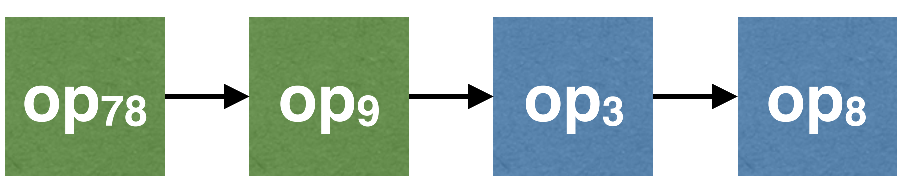
Universal Construction

process p_0

calls **do(op$_{92}$)**


```
do(operation)
    log.append(operation)
    allOps := log.getLog()
    for op in allOps:
        if (op not in appliedOps)
            res := localCopy.apply(op)
            appliedOps.add(op)
            if (operation = op)
                return res
```

log



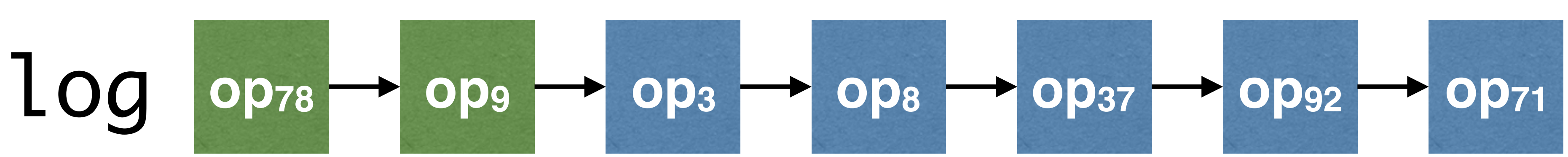
Universal Construction

process p_0

calls **do(op$_{92}$)**

`localCopy`

```
do(operation)
    log.append(operation)
    allOps := log.getLog()
    for op in allOps:
        if (op not in appliedOps)
            res := localCopy.apply(op)
            appliedOps.add(op)
            if (operation = op)
                return res
```



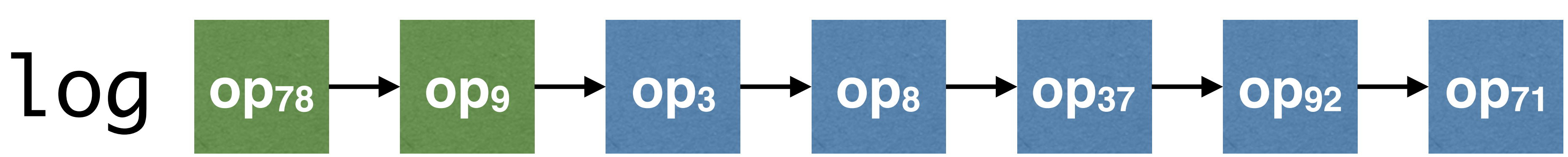
Universal Construction

process p_0

calls **do(op$_{92}$)**

localCopy

```
do(operation)
    log.append(operation)
    allOps := log.getLog() //{op78, …, op71}
    for op in allOps:
        if (op not in appliedOps)
            res := localCopy.apply(op)
            appliedOps.add(op)
            if (operation = op)
                return res
```



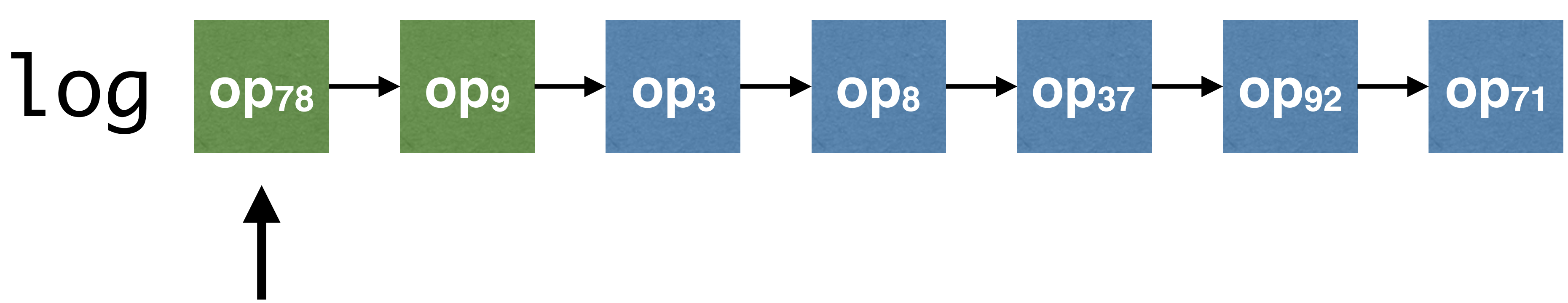
Universal Construction

process p_0

calls **do(op_{92})**

localCopy

```
do(operation)
    log.append(operation)
    allOps := log.getLog() //{op78, …, op71}
    for op in allOps:
        if (op not in appliedOps)
            res := localCopy.apply(op)
            appliedOps.add(op)
            if (operation = op)
                return res
```



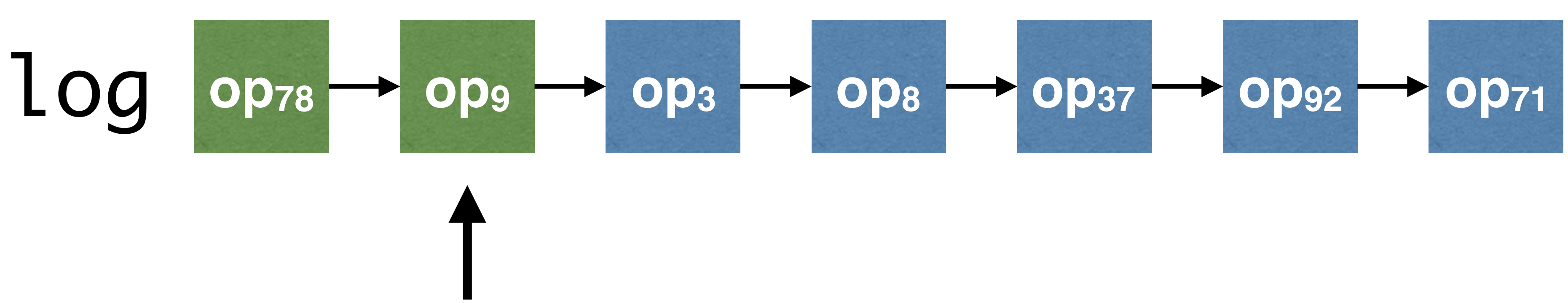
Universal Construction

process p_0

calls **do(op_{92})**

localCopy

```
do(operation)
    log.append(operation)
    allOps := log.getLog() //{op78, …, op71}
    for op in allOps:
        if (op not in appliedOps)
            res := localCopy.apply(op)
            appliedOps.add(op)
            if (operation = op)
                return res
```



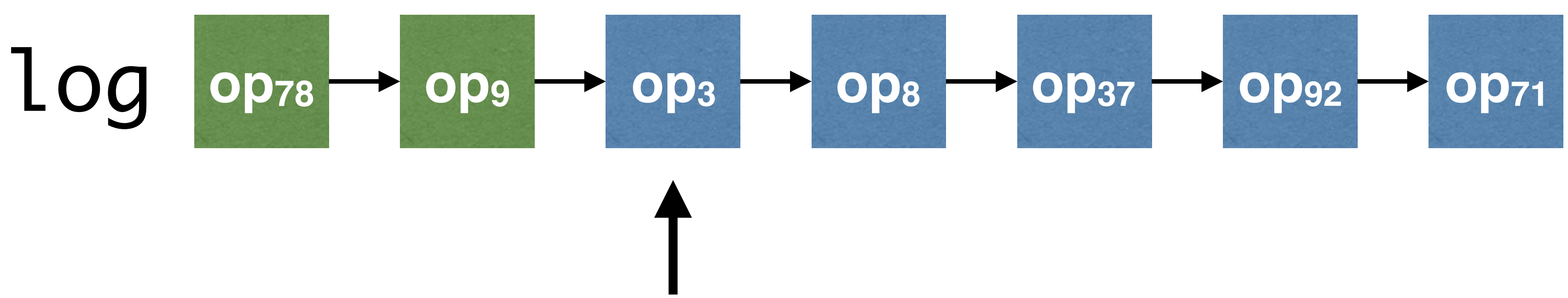
Universal Construction

process p_0

calls **do(op_{92})**

localCopy

```
do(operation)
    log.append(operation)
    allOps := log.getLog() //{op78, …, op71}
    for op in allOps:
        if (op not in appliedOps)
            res := localCopy.apply(op)
            appliedOps.add(op)
            if (operation = op)
                return res
```

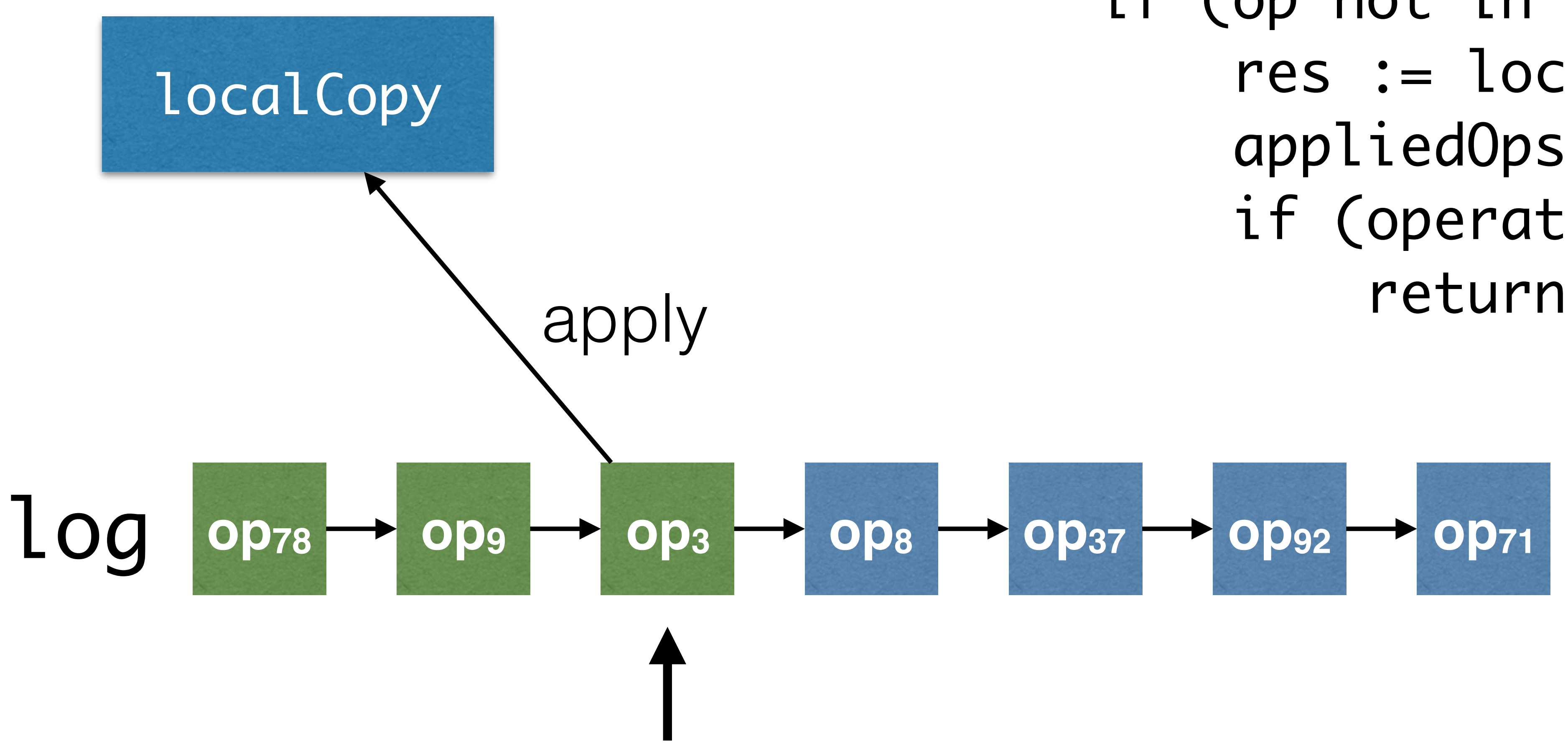


Universal Construction

process p_0

calls **do(op_{92})**

```
do(operation)
    log.append(operation)
    allOps := log.getLog() //{op78, …, op71}
    for op in allOps:
        if (op not in appliedOps)
            res := localCopy.apply(op)
            appliedOps.add(op)
            if (operation = op)
                return res
```



Universal Construction

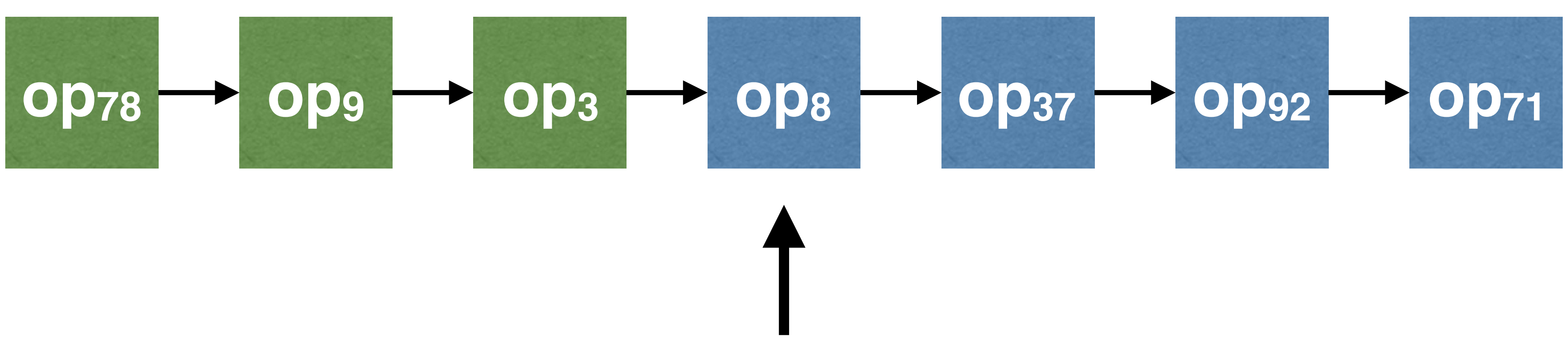
process p_0

calls **do(op$_{92}$)**

localCopy

```
do(operation)
    log.append(operation)
    allOps := log.getLog() //{op78, …, op71}
    for op in allOps:
        if (op not in appliedOps)
            res := localCopy.apply(op)
            appliedOps.add(op)
            if (operation = op)
                return res
```

log

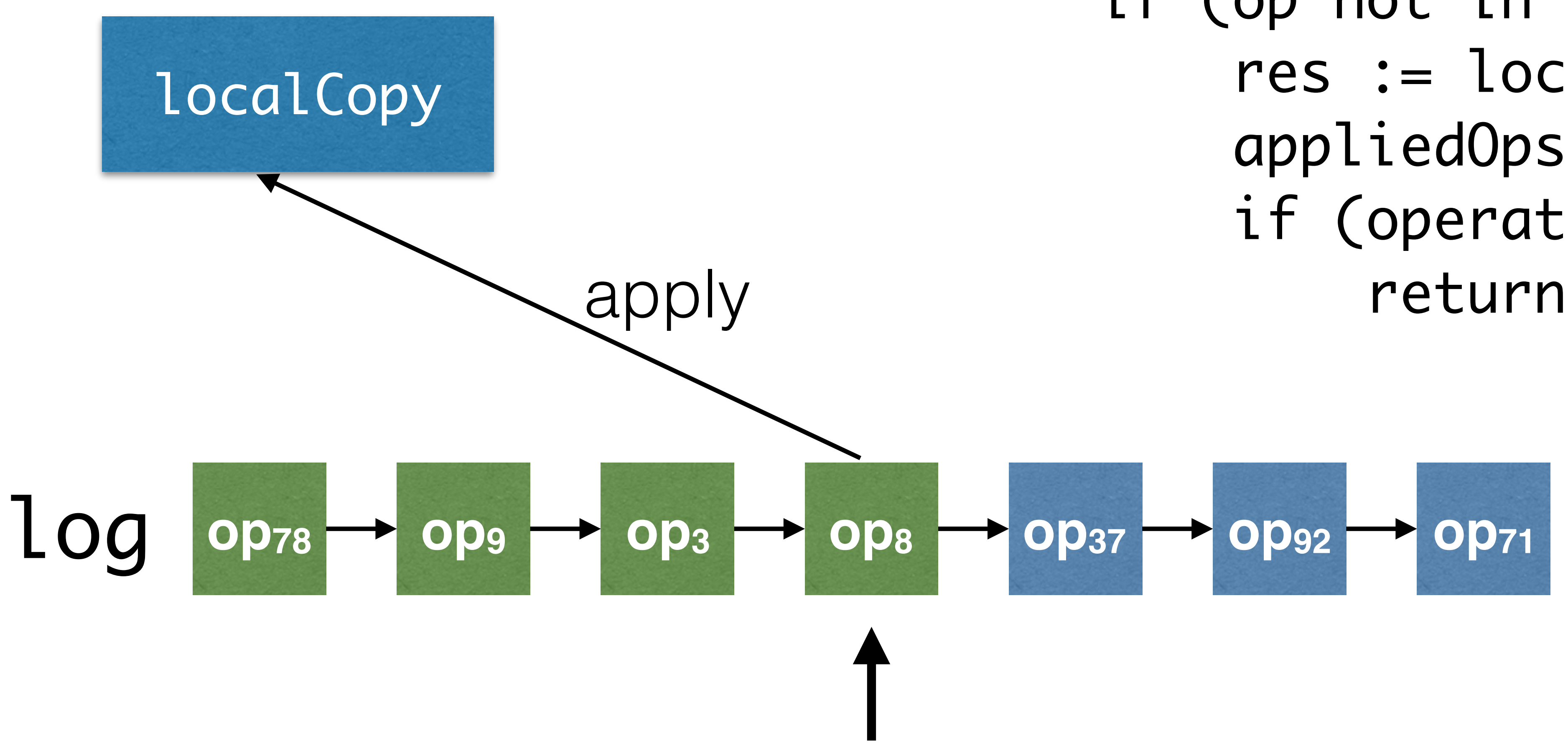


Universal Construction

process p_0

calls **do(op92)**

```
do(operation)
    log.append(operation)
    allOps := log.getLog() //{op78, …, op71}
    for op in allOps:
        if (op not in appliedOps)
            res := localCopy.apply(op)
            appliedOps.add(op)
            if (operation = op)
                return res
```



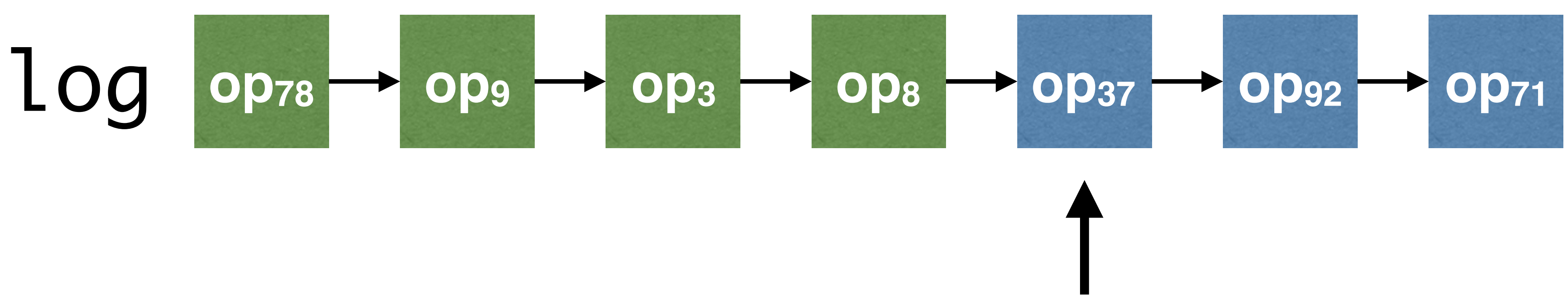
Universal Construction

process p_0

calls **do(op_{92})**

localCopy

```
do(operation)
    log.append(operation)
    allOps := log.getLog() //{op78, …, op71}
    for op in allOps:
        if (op not in appliedOps)
            res := localCopy.apply(op)
            appliedOps.add(op)
            if (operation = op)
                return res
```

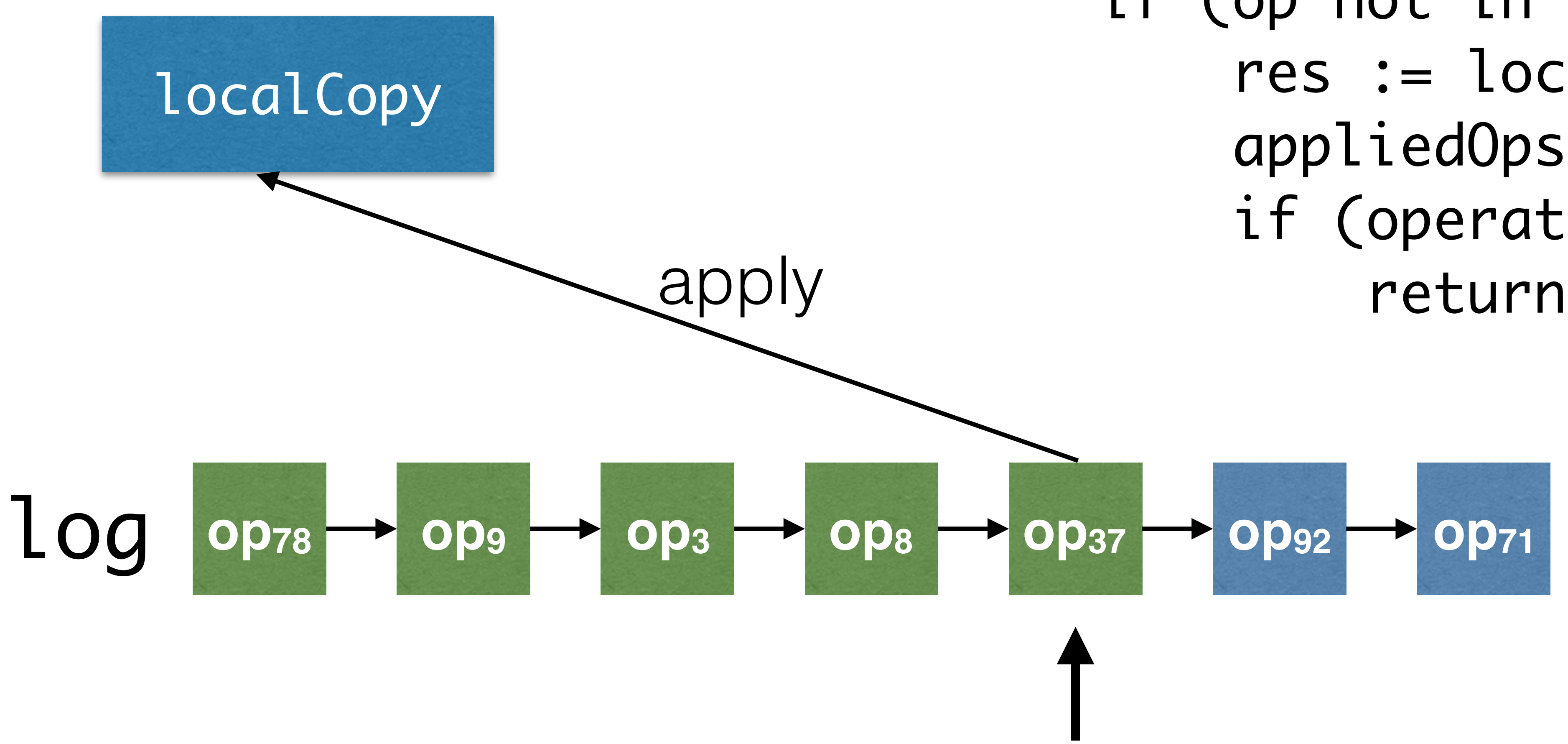


Universal Construction

process p_0

calls **do(op_{92})**

```
do(operation)
    log.append(operation)
    allOps := log.getLog() //{op78, …, op71}
    for op in allOps:
        if (op not in appliedOps)
            res := localCopy.apply(op)
            appliedOps.add(op)
            if (operation = op)
                return res
```

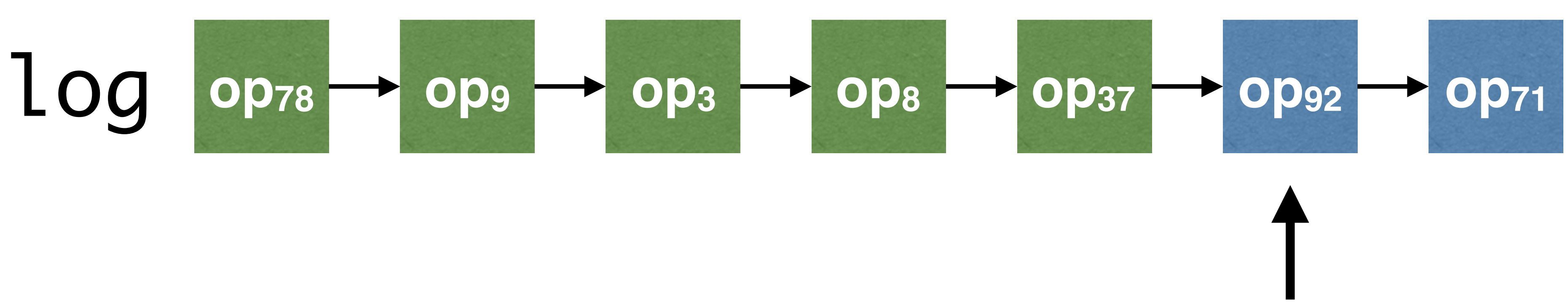


Universal Construction

process p_0

calls **do(op$_{92}$)**


```
do(operation)
    log.append(operation)
    allOps := log.getLog() //{op78, …, op71}
    for op in allOps:
        if (op not in appliedOps)
            res := localCopy.apply(op)
            appliedOps.add(op)
            if (operation = op)
                return res
```

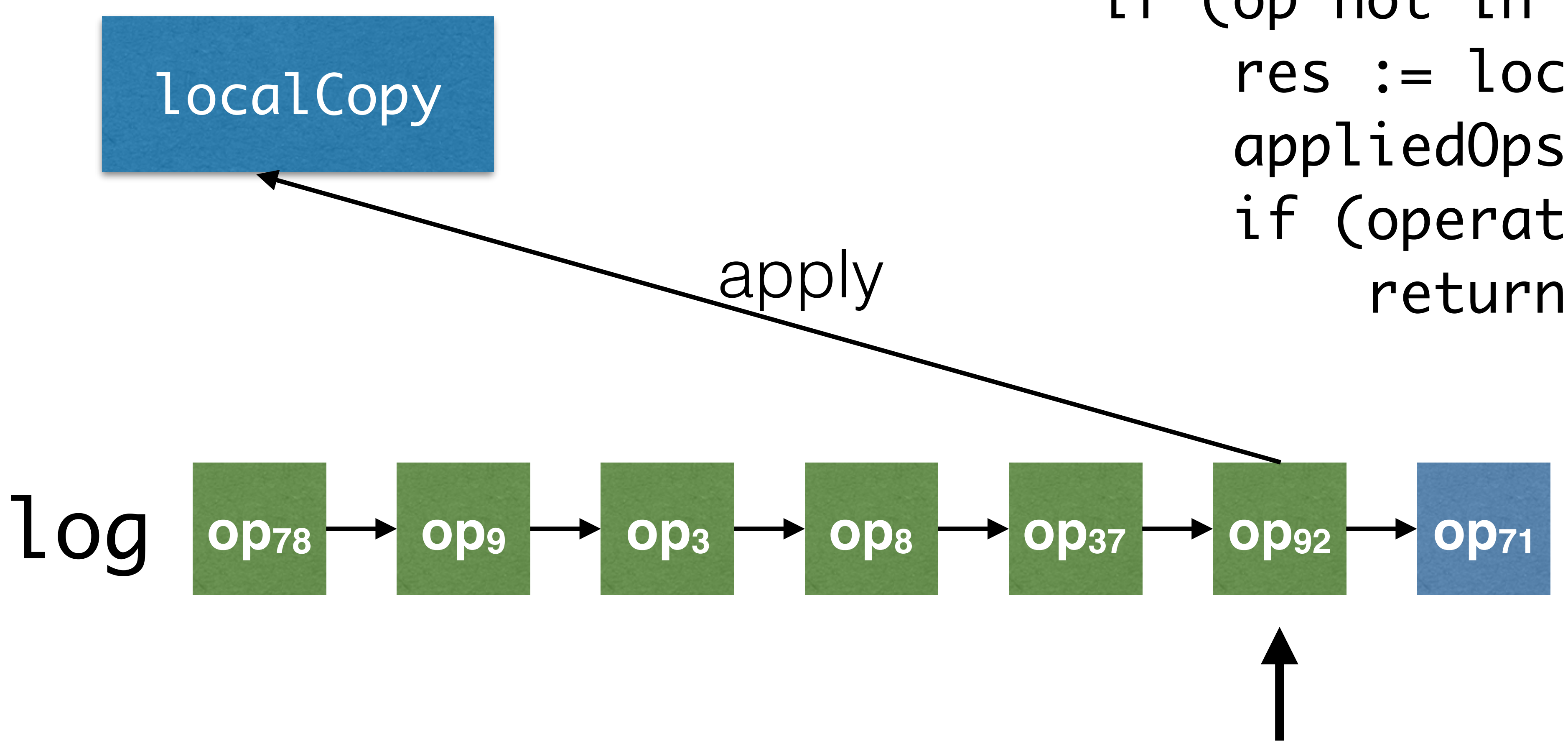


Universal Construction

process p_0

calls **do(op_{92})**

```
do(operation)
    log.append(operation)
    allOps := log.getLog() //{op78, …, op71}
    for op in allOps:
        if (op not in appliedOps)
            res := localCopy.apply(op)
            appliedOps.add(op)
            if (operation = op)
                return res
```



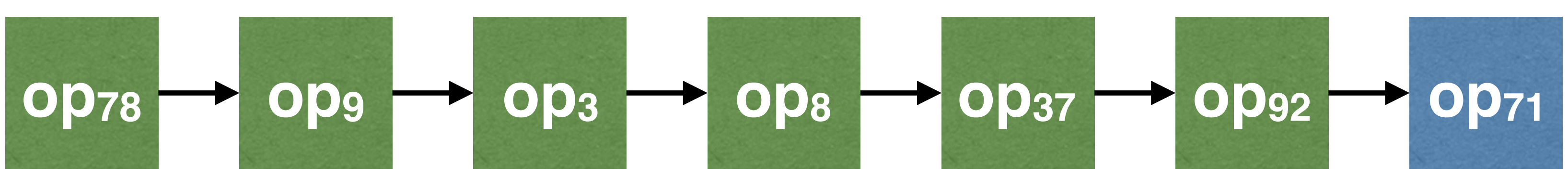
Universal Construction

process p_0

calls **do(op_{92})**


```
do(operation)
    log.append(operation)
    allOps := log.getLog() //{op78, …, op71}
    for op in allOps:
        if (op not in appliedOps)
            res := localCopy.apply(op)
            appliedOps.add(op)
            if (operation = op)
                return res
```

log



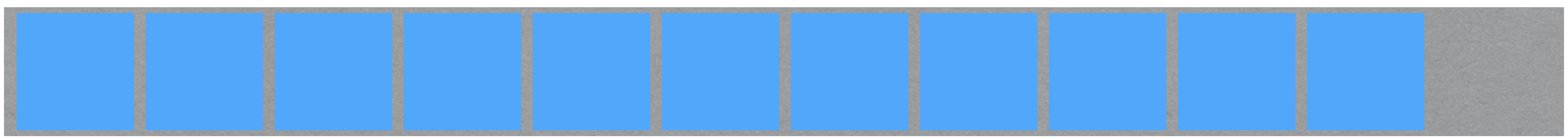
Log Object - Implementation

Log object implementation using only:
`read`
`xor`
`decrement`
`fetch-and-increment`

Provides similar performance to a `CAS` solution.

Log Object - Implementation

An unbounded array `A` of *b*-bit integers.



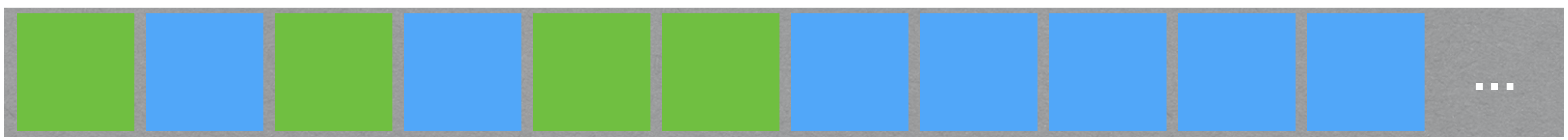
`read`, `xor` and `decrement`

C counter (`read`, `fetch-and-increment`)

Log Object - Implementation

<u>*append* **Idea** (2 phases)</u>

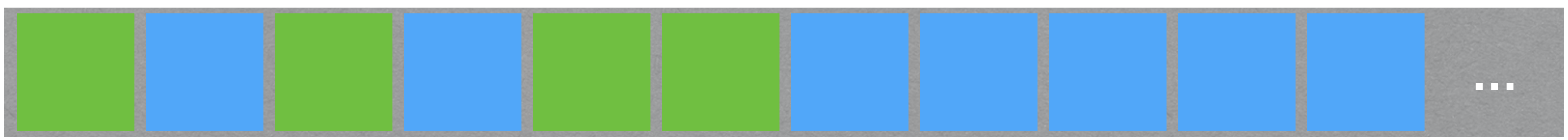
1. “**Record**”(i.e., store) them in array **A**.
2. **Invalidate** elements.



Log Object - Implementation

<u>*append* **Idea** (2 phases)</u>

1. "**Record**"(i.e., store) them in array **A**.
2. **Invalidate** elements.

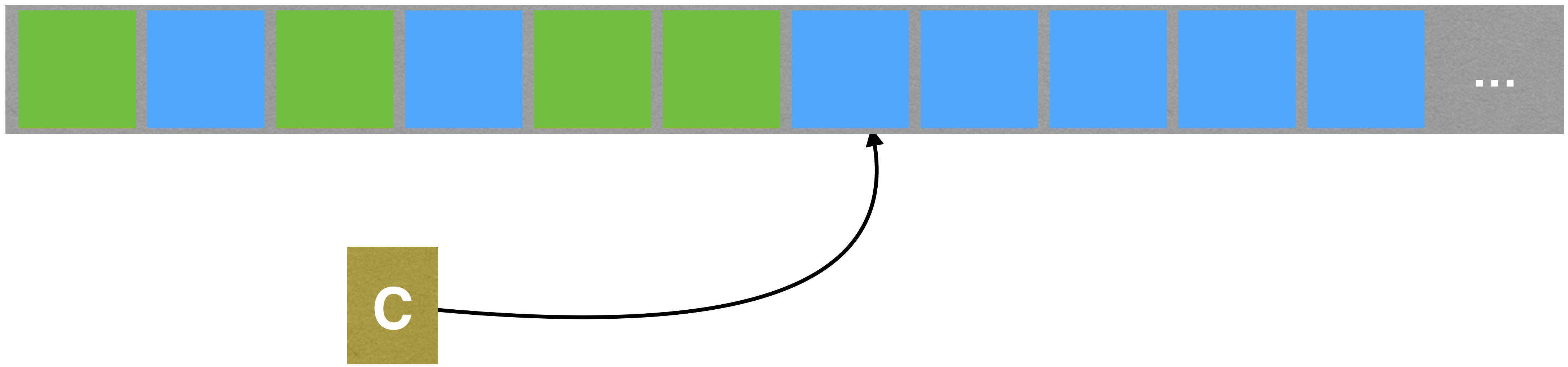


Get a position in the array to store (**record**) an item using `fetch-and-increment`.

Log Object - Implementation

<u>*append* **Idea** (2 phases)</u>

1. "**Record**"(i.e., store) them in array **A**.
2. **Invalidate** elements.

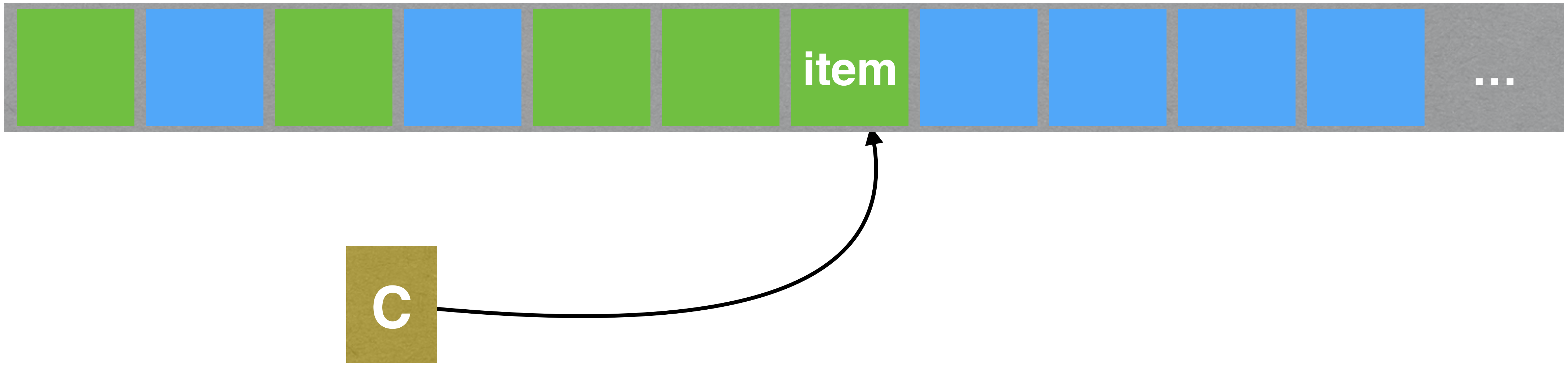


Get a position in the array to store (**record**) an item using `fetch-and-increment`.

Log Object - Implementation

<u>*append* **Idea** (2 phases)</u>

1. "**Record**"(i.e., store) them in array **A**.
2. **Invalidate** elements.



Get a position in the array to store (**record**) an item using `fetch-and-increment`.

Log Object - Implementation

<u>*append* **Idea** (2 phases)</u>

1. "**Record**"(i.e., store) them in array **A**.
2. **Invalidate** elements.

Invalidate: Make sure that no new items will be added before you after the ***append*** terminates.

Log Object - Implementation

<u>**append** **Idea** (2 phases)</u>

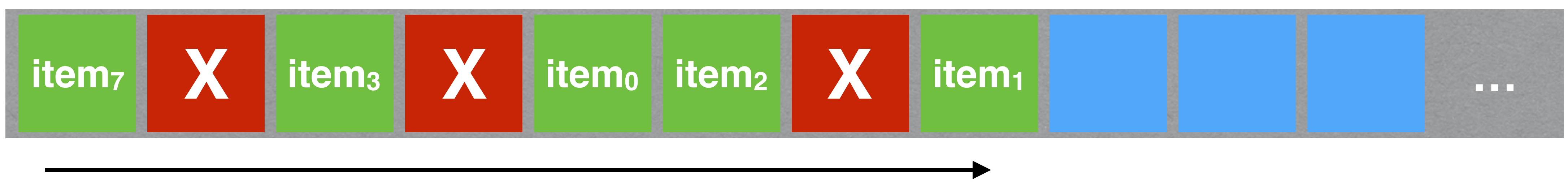
1. "**Record**"(i.e., store) them in array **A**.
2. **Invalidate** elements.

Invalidate: Make sure that no new items will be added before your item after the **append** terminates.

Log Object - Implementation

`getLog` Idea

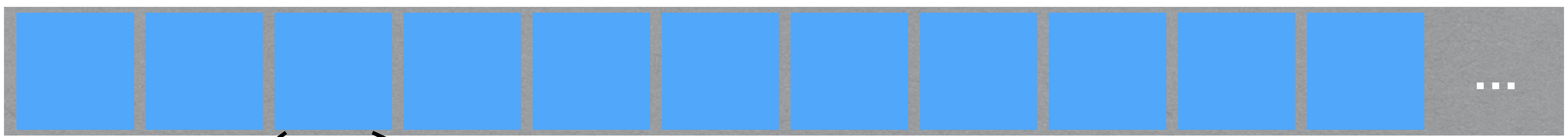
Traverse the array A to get all the non-invalidated items.

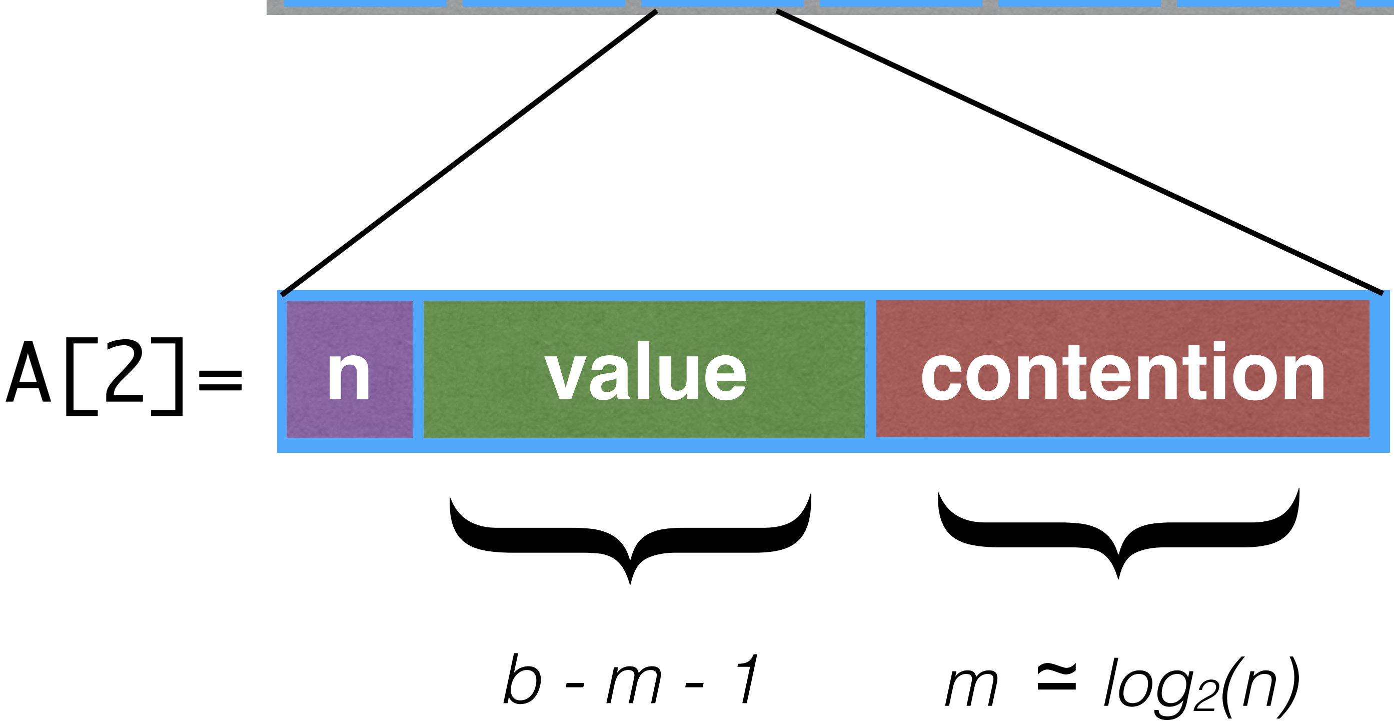


Traverse and return {$item_7$, $item_3$, $item_0$, $item_2$, $item_1$}

Log Object - Implementation

An unbounded array **A** of *b*-bit integers.





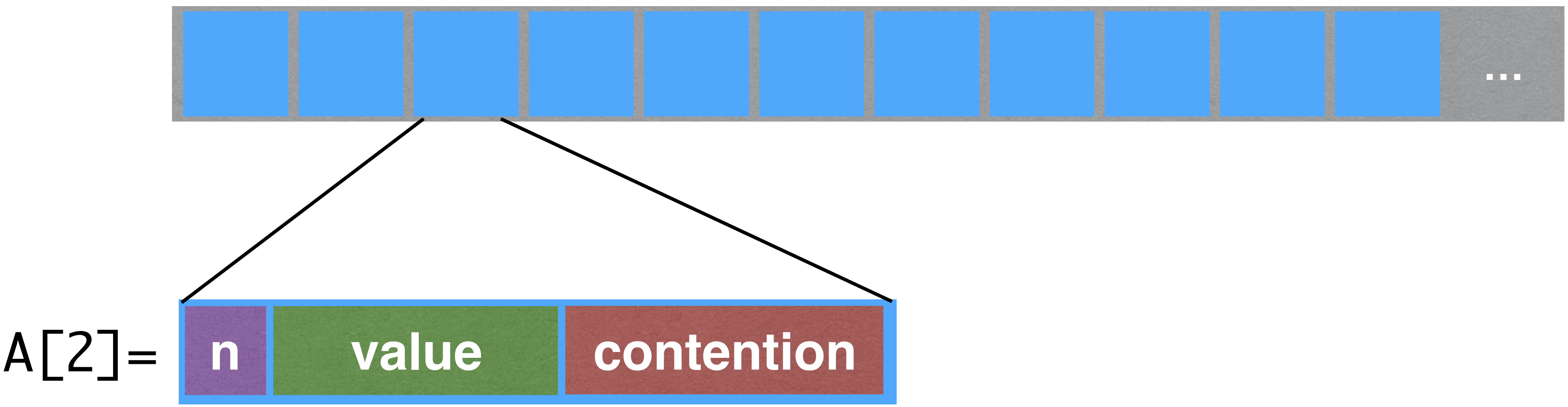
An array element `A[i]` is:

- **invalid** if `A[i] < 0 (n = 1)`
- **empty** if `A[i] = 0`
- **valid** otherwise

We can either **record** or **invalidate** an array element.

Log Object - Implementation

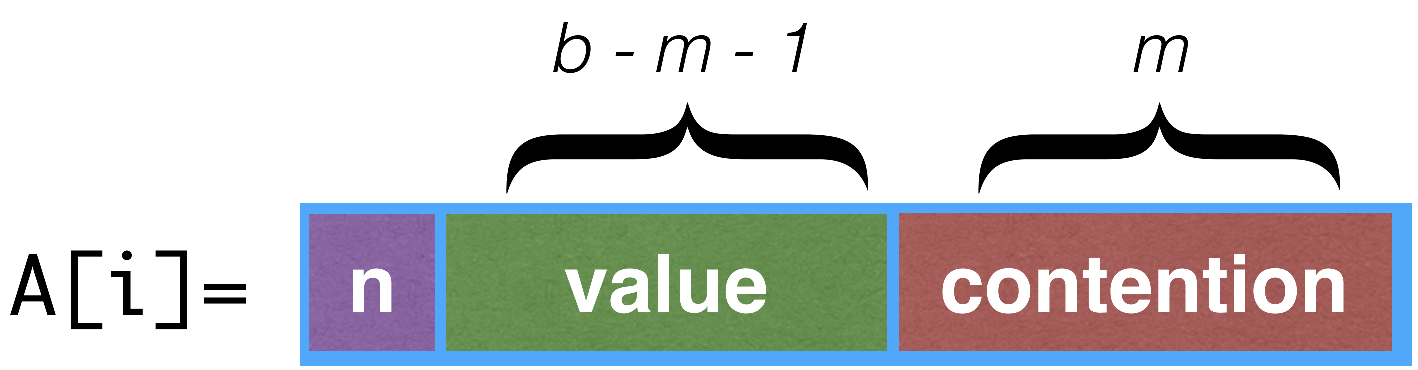
An unbounded array `A` of *b*-bit integers.



For every `A[i]`:

- at **most one** process will ever attempt to record a value
- at **most *n–1*** processes will attempt to invalidate it

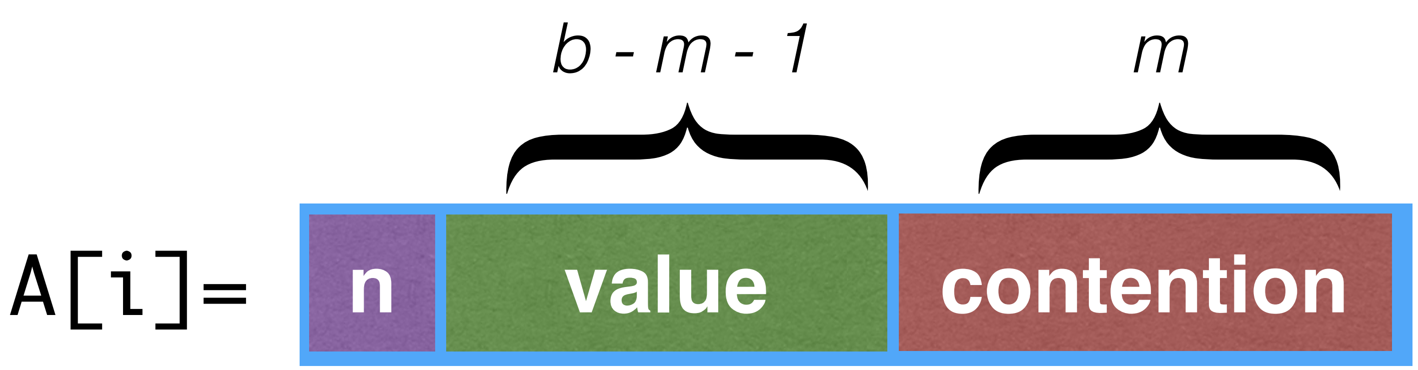
Log Object - Record



An array element `A[i]` is:

- **invalid** if `A[i] < 0`
- **empty** if `A[i] = 0`
- **valid** otherwise

Log Object - Record



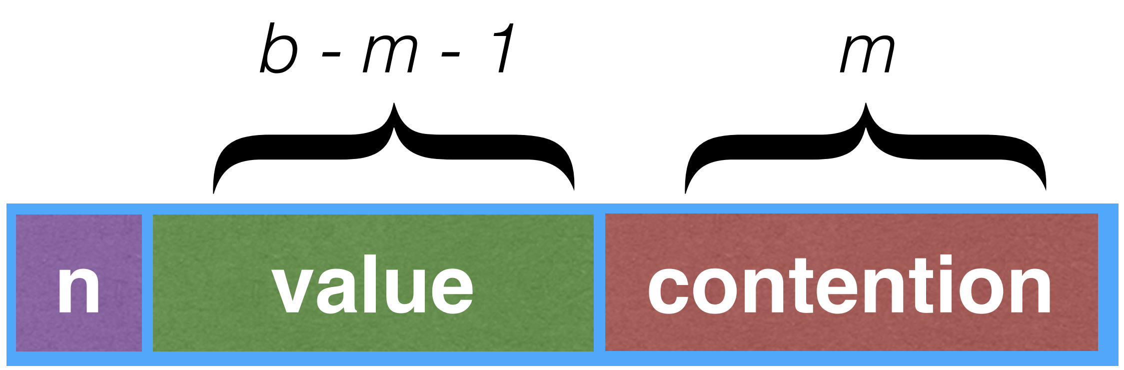
An array element `A[i]` is:

- **invalid** if `A[i] < 0`
- **empty** if `A[i] = 0`
- **valid** otherwise

To record a value `v` in `A[i]`, we apply `xor(v')` to `A[i]`, where `v'` is `v` shifted to the left by *m* bits + (2^m - 1).

Log Object - Record

A[i]=

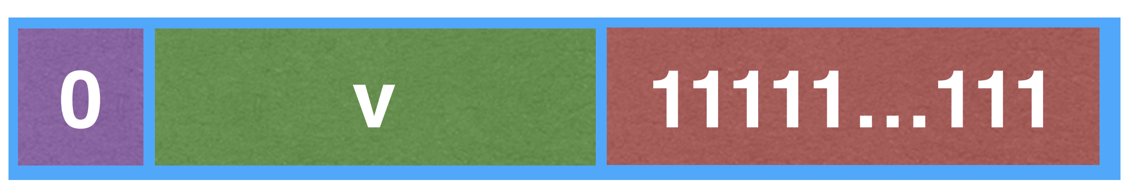


An array element `A[i]` is:

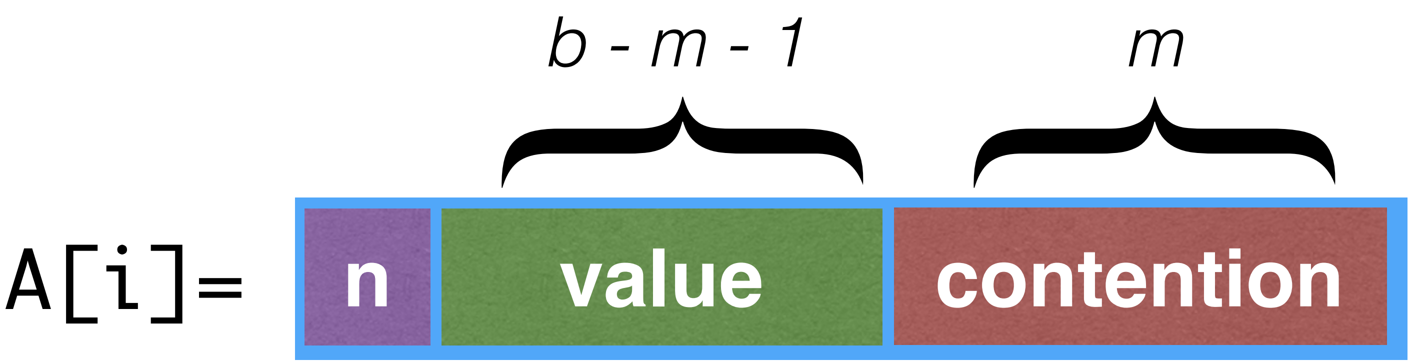
- **invalid** if `A[i] < 0`
- **empty** if `A[i] = 0`
- **valid** otherwise

To record a value `v` in `A[i]`, we apply `xor(v')` to `A[i]`, where `v'` is `v` shifted to the left by *m* bits + (2^m - 1).

v'=



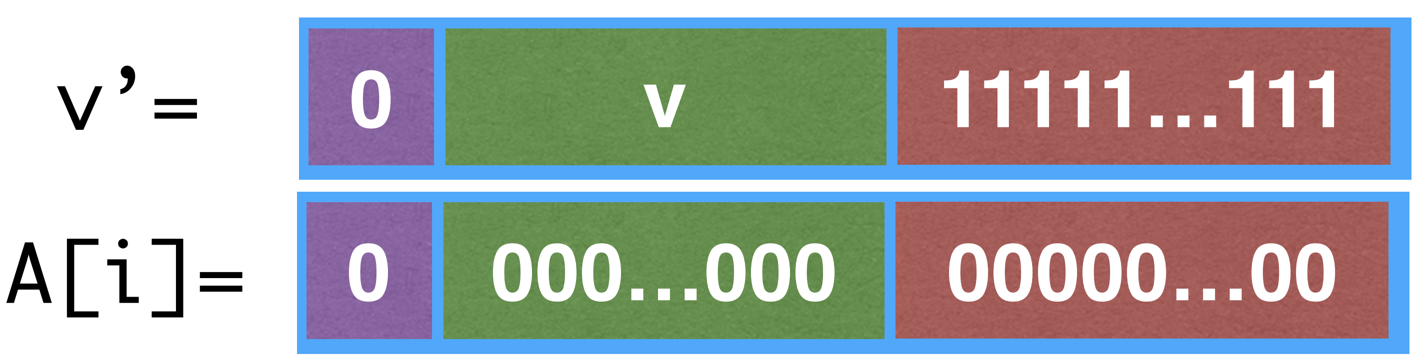
Log Object - Record



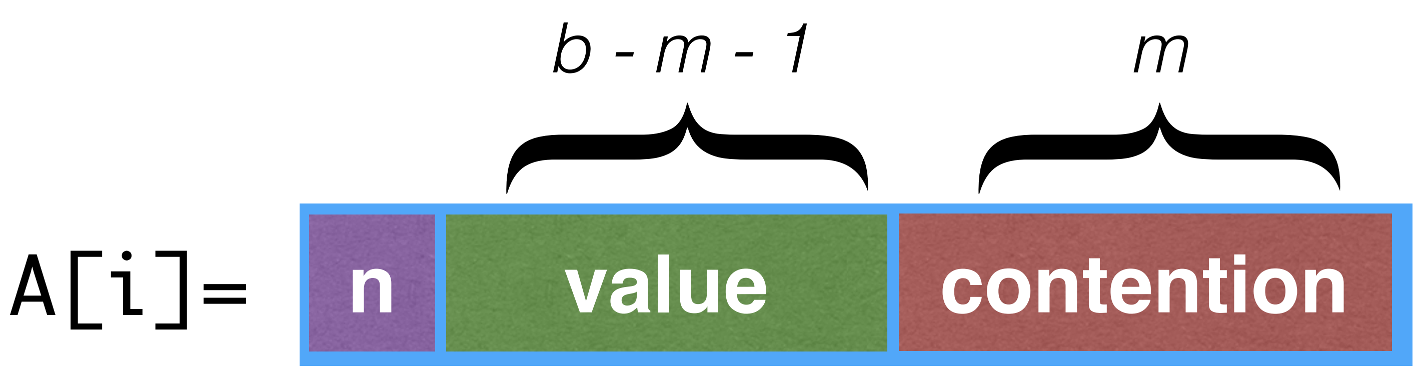
An array element `A[i]` is:

- **invalid** if `A[i] < 0`
- **empty** if `A[i] = 0`
- **valid** otherwise

To record a value **v** in `A[i]`, we apply `xor(v')` to `A[i]`, where `v'` is **v** shifted to the left by *m* bits + (2^m - 1).



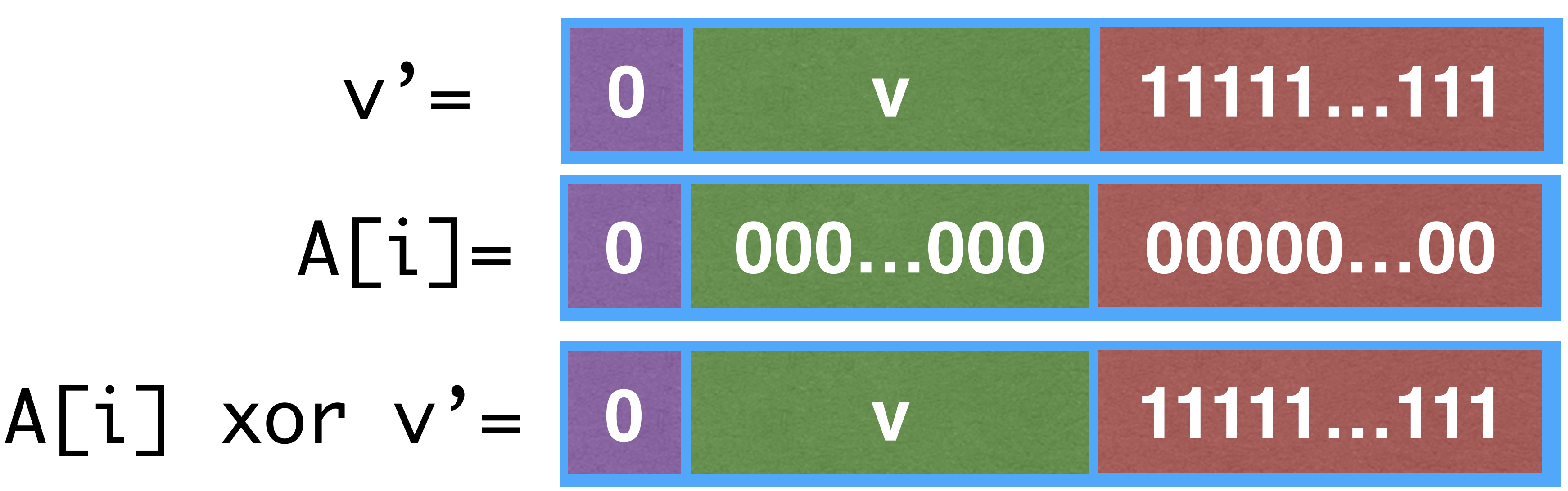
Log Object - Record



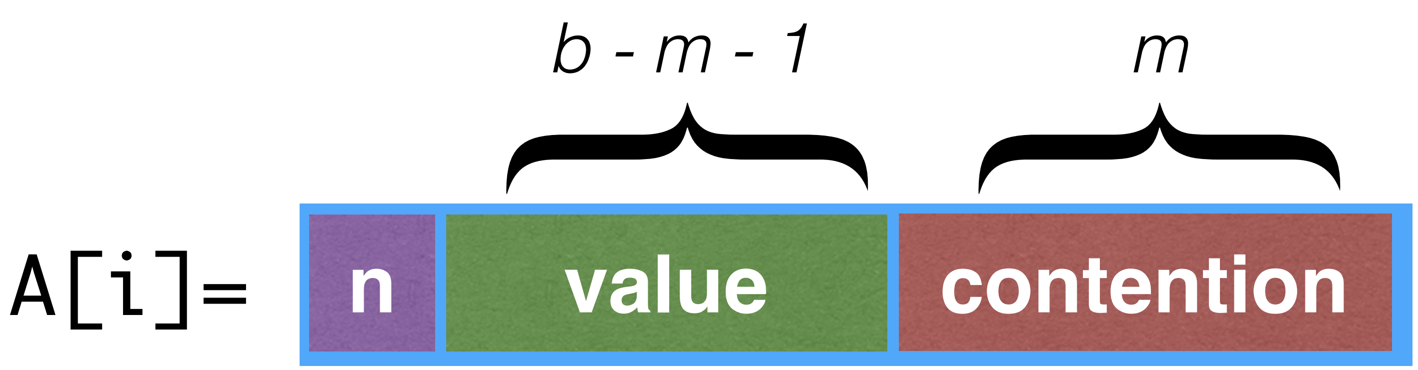
An array element `A[i]` is:

- **invalid** if `A[i] < 0`
- **empty** if `A[i] = 0`
- **valid** otherwise

To record a value **v** in `A[i]`, we apply `xor(v’)` to `A[i]`, where `v’` is **v** shifted to the left by *m* bits + (2^m - 1).



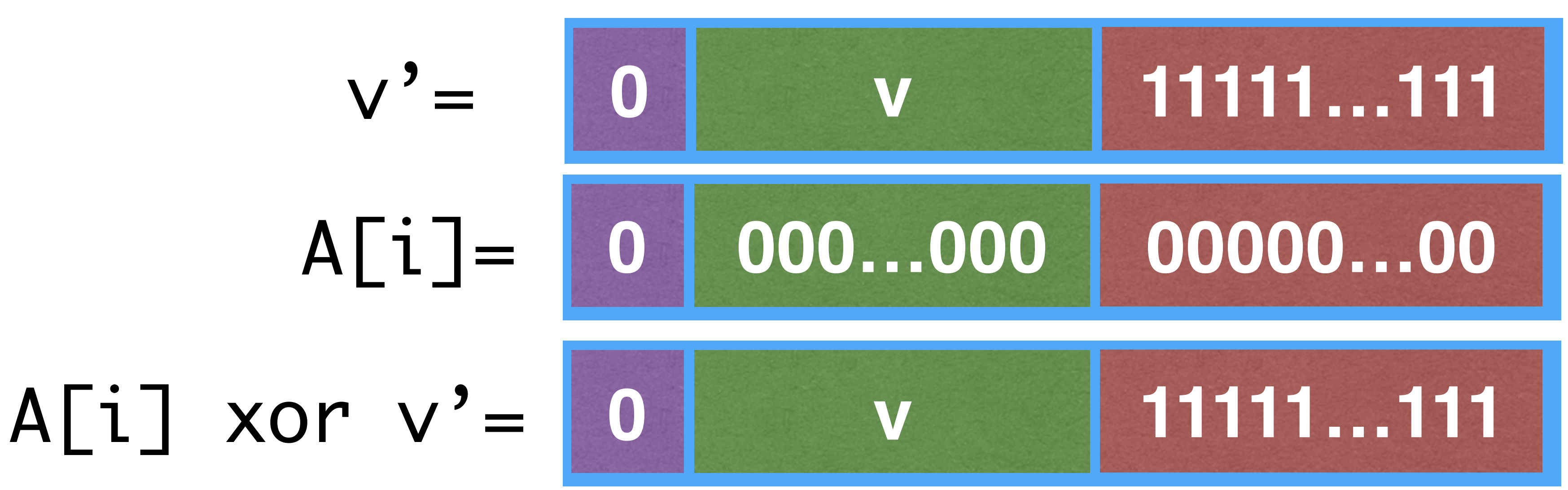
Log Object - Record



An array element `A[i]` is:

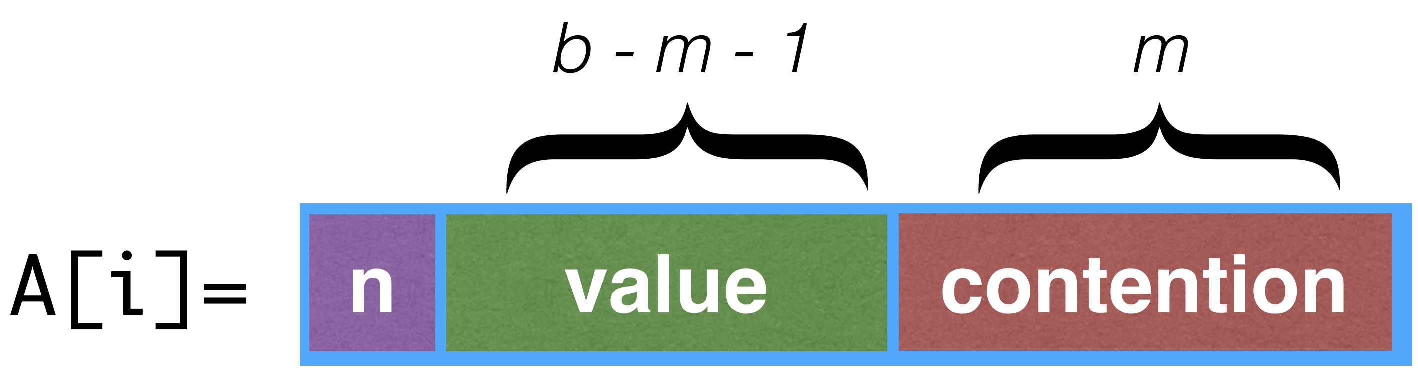
- **invalid** if `A[i] < 0`
- **empty** if `A[i] = 0`
- **valid** otherwise

To record a value `v` in `A[i]`, we apply `xor(v')` to `A[i]`, where `v'` is `v` shifted to the left by *m* bits + (2^m - 1).



If `A[i]` is empty, then `A[i] xor v'` is valid and contains value `v`.

Log Object - Record

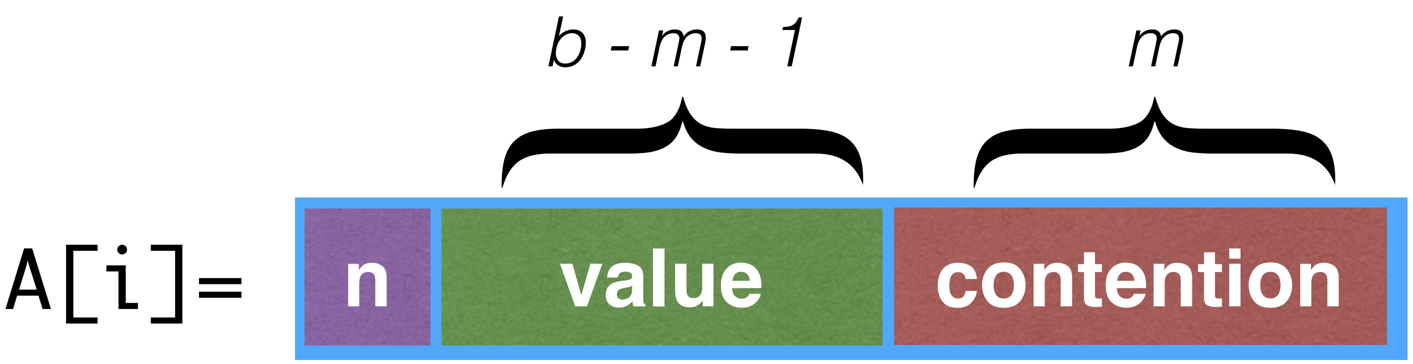


An array element `A[i]` is:

- **invalid** if `A[i] < 0`
- **empty** if `A[i] = 0`
- **valid** otherwise

To record a value **v** in `A[i]`, we apply `xor(v')` to `A[i]`, where `v'` is **v** shifted to the left by *m* bits + (2^m - 1).

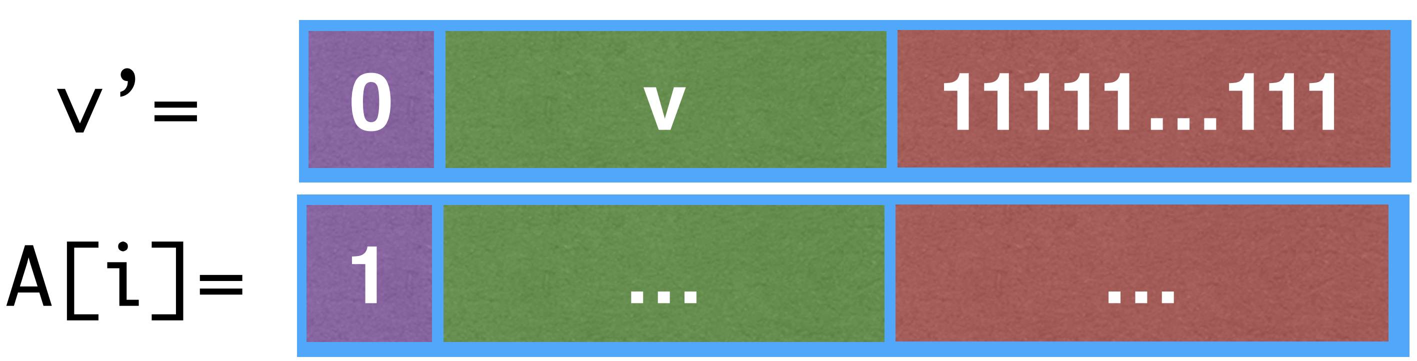
Log Object - Record



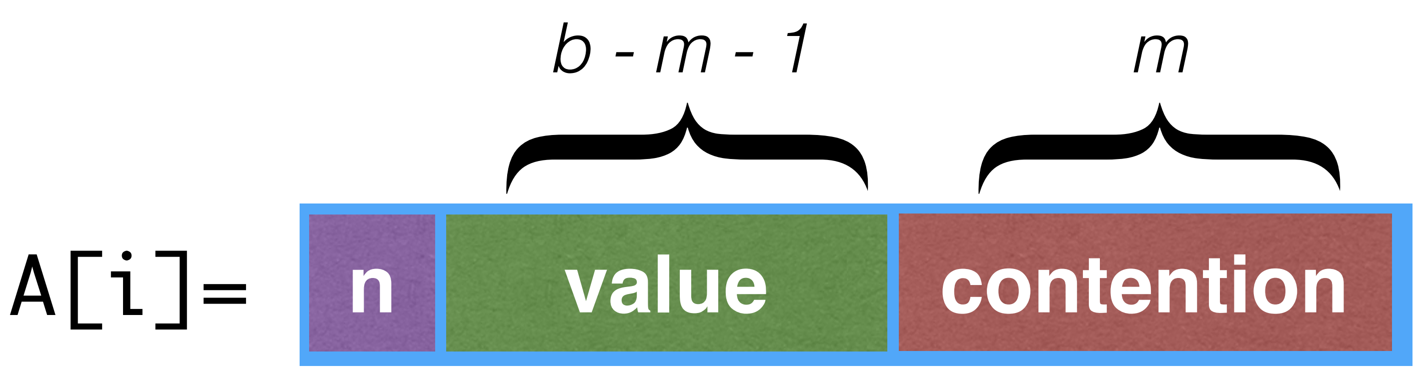
An array element `A[i]` is:

- **invalid** if `A[i] < 0`
- **empty** if `A[i] = 0`
- **valid** otherwise

To record a value `v` in `A[i]`, we apply `xor(v')` to `A[i]`, where `v'` is `v` shifted to the left by *m* bits + (2^m - 1).



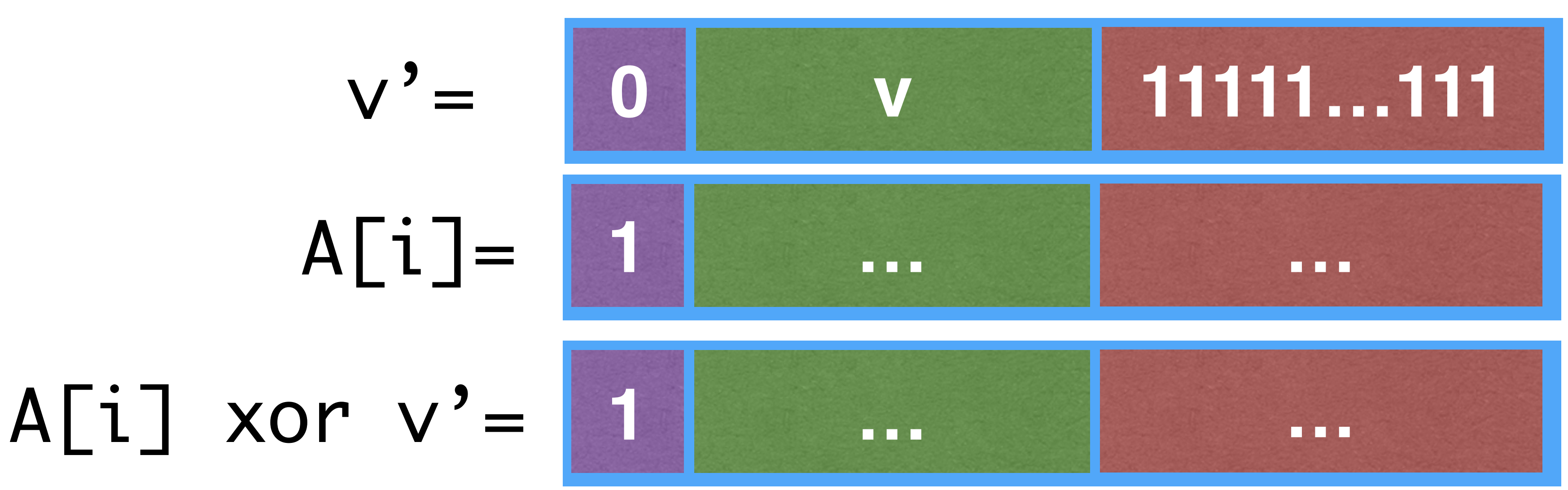
Log Object - Record



An array element `A[i]` is:

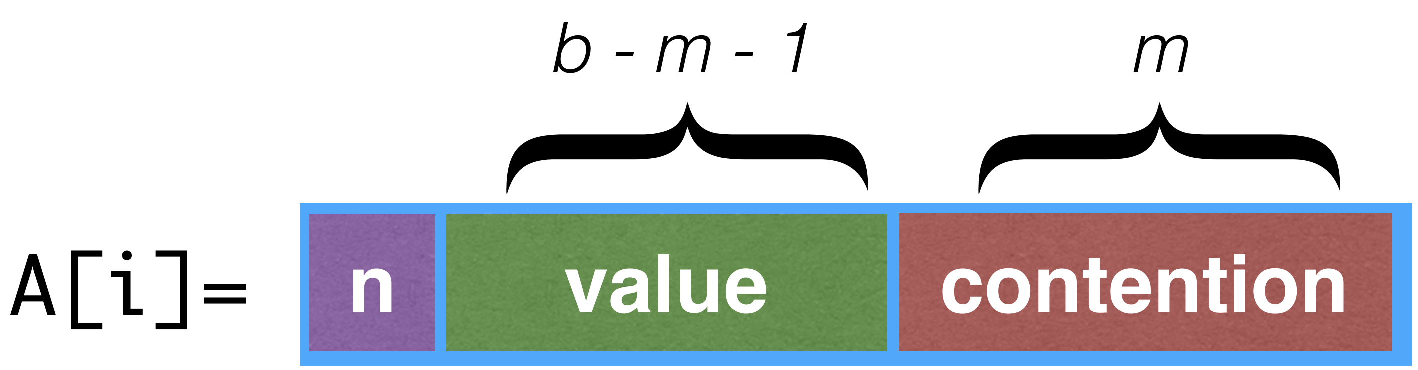
- **invalid** if `A[i] < 0`
- **empty** if `A[i] = 0`
- **valid** otherwise

To record a value **v** in `A[i]`, we apply `xor(v')` to `A[i]`, where `v'` is **v** shifted to the left by *m* bits + (2^m - 1).



If `A[i]` is invalid, then `A[i] xor v'` is invalid.

Log Object - Invalidate

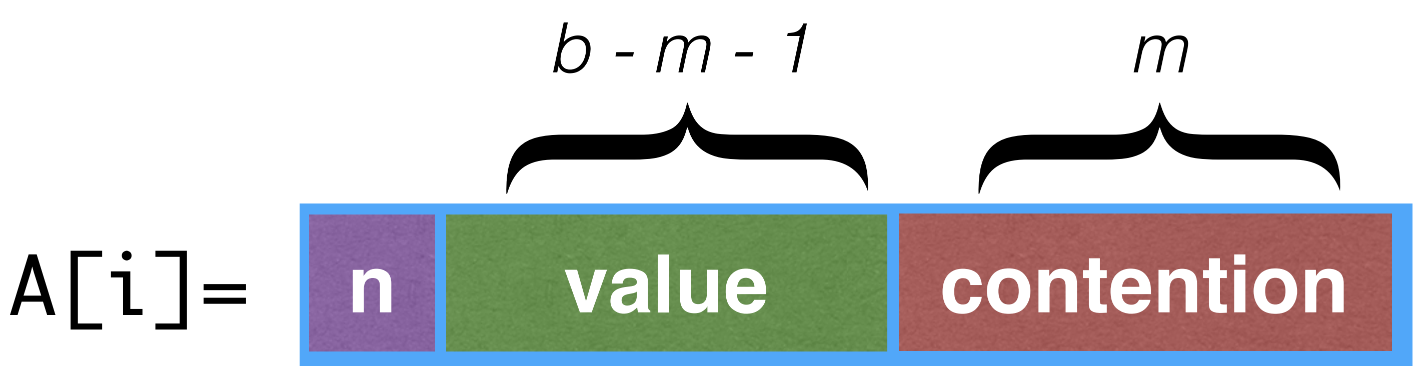


An array element `A[i]` is:

- **invalid** if `A[i] < 0`
- **empty** if `A[i] = 0`
- **valid** otherwise

To invalidate `A[i]`, we `decrement()` `A[i]`.

Log Object - Invalidate



An array element `A[i]` is:

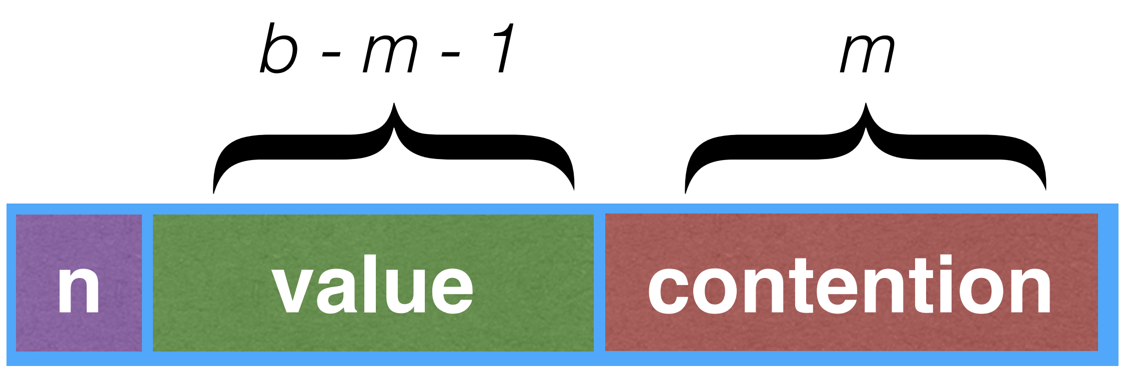
- **invalid** if `A[i] < 0`
- **empty** if `A[i] = 0`
- **valid** otherwise

To invalidate `A[i]`, we `decrement()` `A[i]`.

If `A[i]` is empty before the `decrement()`, then `A[i]` is invalid after the `decrement()`.

Log Object - Invalidate

A[i]=



An array element A[i] is:

- **invalid** if A[i] < 0
- **empty** if A[i] = 0
- **valid** otherwise

To invalidate A[i], we decrement() A[i].

If A[i] is empty before the decrement(), then A[i] is invalid after the decrement().

If A[i] is invalid before the decrement(), then A[i] remains invalid.

Log Object - Record vs Invalidate

Initially, `A[i]`'s are empty.

Whether `A[i]` becomes valid or invalid depends on whether the `xor` or the `decrement` takes places first.

Log Object - append

```
append(value)
    while (true)
       l := C.fetch-and-increment()
       A[l] := A[l].xor(value)
       if (A[l] is invalid)
           continue

    j := l - 1
    while (j != -1)
        if (A[j] is empty) // then invalidate
            A[j].decrement()
        j := j - 1

    return ok
```

Log Object - `append`

```
append(value)
    while (true)
       l := C.fetch-and-increment()
       A[l] := A[l].xor(value)
       if (A[l] is invalid)
           continue

    j := l - 1
    while (j != -1)
        if (A[j] is empty) // then invalidate
            A[j].decrement()
        j := j - 1

    return ok
```

`fetch-and-increment` guarantees that each element is `xor`ed at most once.

Log Object - `append`

```
append(value)
    while (true)
       l := C.fetch-and-increment()
       A[l] := A[l].xor(value)
       if (A[l] is invalid)
           continue

    j := l - 1
    while (j != -1)
        if (A[j] is empty) // then invalidate
            A[j].decrement()
        j := j - 1

    return ok
```

Log Object - `append`

```
append(value)
    while (true)
       l := C.fetch-and-increment()
       A[l] := A[l].xor(value)
       if (A[l] is invalid)
           continue

    j := l - 1
    while (j != -1)
        if (A[j] is empty) // then invalidate
            A[j].decrement()
        j := j - 1

    return ok
```

Maybe another process was fast enough and invalidated `A[l]`.

Log Object - append

```
append(value)
    while (true)
       l := C.fetch-and-increment()
       A[l] := A[l].xor(value)
       if (A[l] is invalid)
           continue

    j := l - 1
    while (j != -1)
        if (A[j] is empty) // then invalidate
            A[j].decrement()
        j := j - 1

    return ok
```

Log Object - append

```
append(value)
    while (true)
       l := C.fetch-and-increment()
       A[l] := A[l].xor(value)
       if (A[l] is invalid)
           continue
```

```
j := l - 1
while (j != -1)
    if (A[j] is empty) // then invalidate
        A[j].decrement()
    j := j - 1
```

← Why is this needed?

```
return ok
```

Log Object - `append`

```
append(value)
    while (true)
       l := C.fetch-and-increment()
       A[l] := A[l].xor(value)
       if (A[l] is invalid)
           continue

    j := l - 1
    while (j != -1)
        if (A[j] is empty) // then invalidate
            A[j].decrement()
        j := j - 1

    return ok
```

Why is this needed?

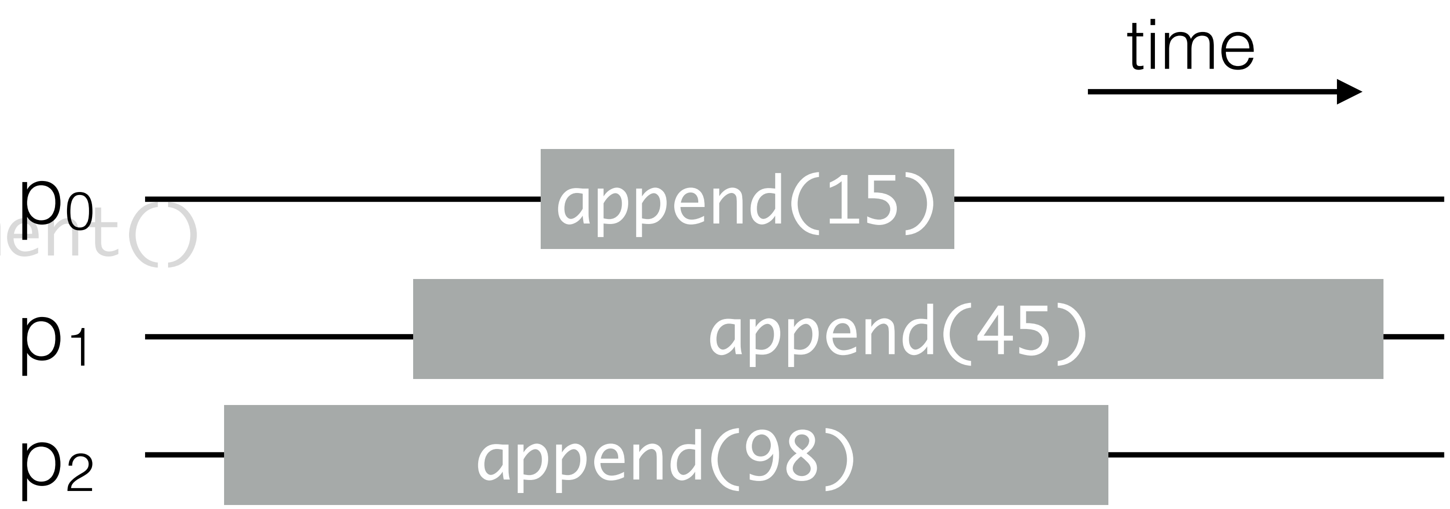
`getLog` might block

Log Object - **append**

```
append(value)
    while (true)
        l := C.fetch-and-increment()
        A[l] := A[l].xor(value)
        if (A[l] is invalid)
            continue

    j := l - 1
    while (j != -1)
        if (A[j] is empty) // then invalidate
            A[j].decrement()
        j := j - 1

    return ok
```



Why is this needed?

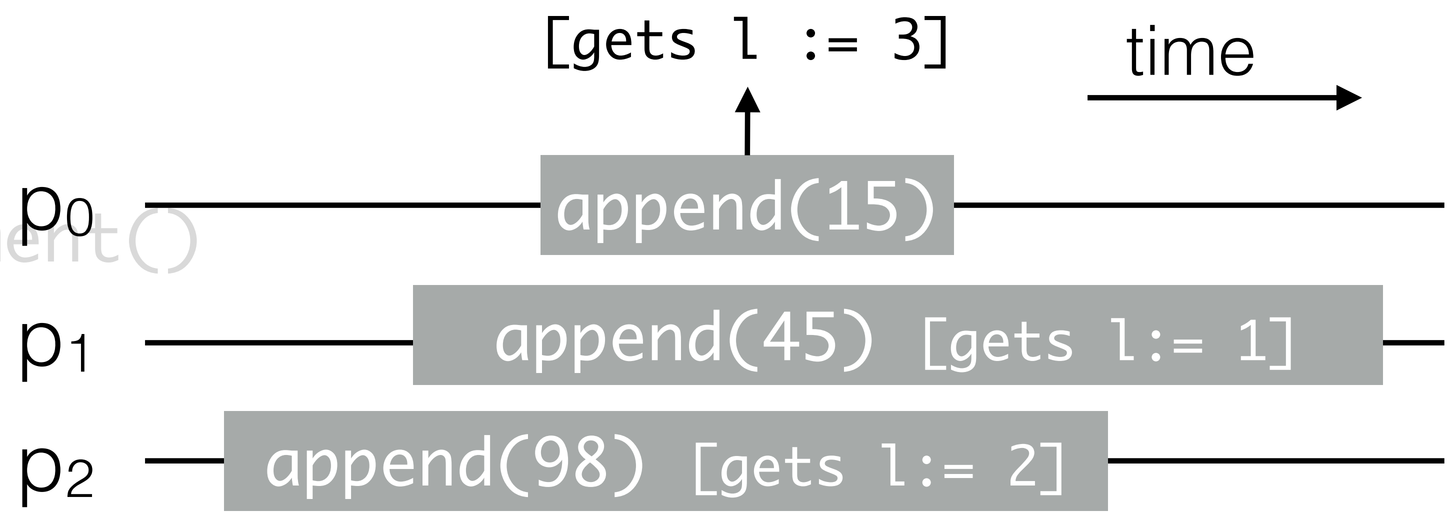
getLog might block

Log Object - **append**

```
append(value)
    while (true)
        l := C.fetch-and-increment()
        A[l] := A[l].xor(value)
        if (A[l] is invalid)
            continue

    j := l - 1
    while (j != -1)
        if (A[j] is empty) // then invalidate
            A[j].decrement()
        j := j - 1

    return ok
```

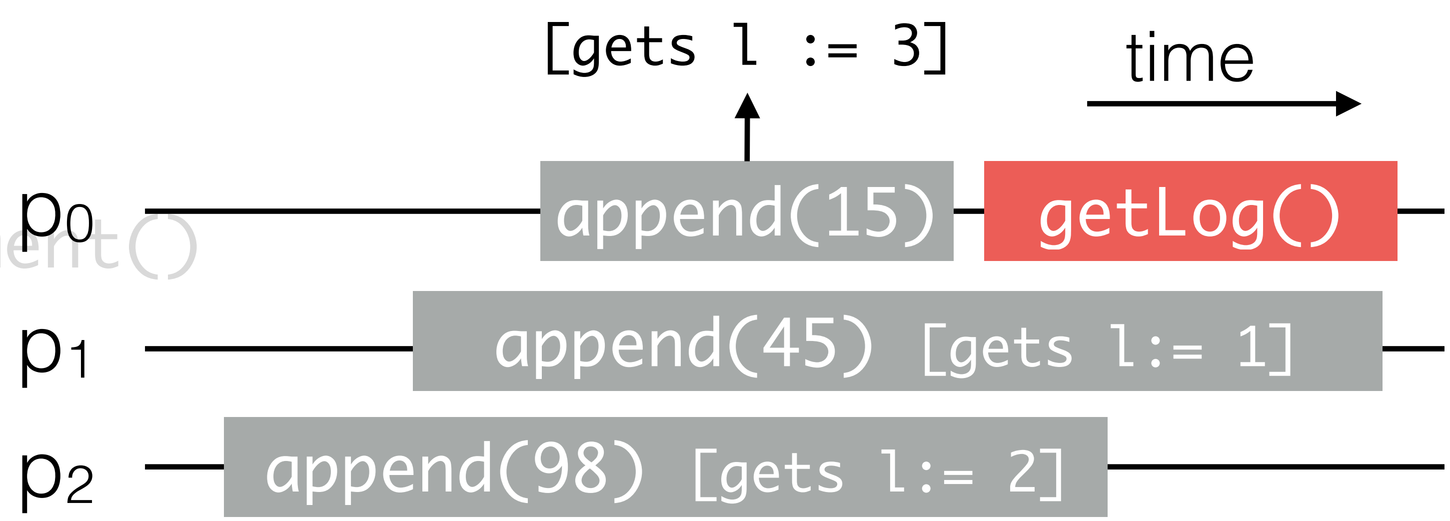


Why is this needed?

getLog might block

Log Object - **append**

```
append(value)
    while (true)
        l := C.fetch-and-increment()
        A[l] := A[l].xor(value)
        if (A[l] is invalid)
            continue
```




```
    j := l - 1
    while (j != -1)
        if (A[j] is empty) // then invalidate
            A[j].decrement()
        j := j - 1

    return ok
```

Why is this needed?

`getLog` might block

Log Object - `getLog`

```
getLog()
    result := ∅

    l := 0
    while (A[l] != 0)
       if (A[l] is valid)
           value = A[l].extract()
           result.add(value)
       l := l + 1

    return result
```

Log Object - `getLog`

```
getLog()
    result := ∅

    l := 0
    while (A[l] != 0)
       if (A[l] is valid)
           value = A[l].extract()
           result.add(value)
       l := l + 1

    return result
```

Is `getLog` wait-free?

Log Object - `getLog`

```
getLog()
    result := ∅

    l := 0
    while (A[l] != 0)
       if (A[l] is valid)
           value = A[l].extract()
           result.add(value)
       l := l + 1

    return result
```

Is `getLog` wait-free?

No.

Log Object - `getLog`

```
getLog()
    result := ∅

    l := 0
    while (A[l] != 0)
        if (A[l] is valid)
            value = A[l].extract()
            result.add(value)
        l := l + 1

    return result
```

Is `getLog` wait-free?

No.

Log Object - `getLog`

```
getLog()
    result := ∅

    l := C.read()
    for (i := 0; i <= l; ++i)
       if (A[i] is valid)
           value = A[i].extract()
           result.add(value)

    return result
```

Is `getLog` wait-free?

No.

How about now?

Log Object - `getLog`

```
getLog()
    result := ∅

    l := C.read()
    for (i := 0; i <= l; ++i)
       if (A[i] is valid)
           value = A[i].extract()
           result.add(value)

    return result
```

Is `getLog` wait-free?

No.

How about now?

Yes.

Conclusion

- Herlihy's hierarchy helps us understand the power of different synchronization primitives …

Conclusion

- Herlihy's hierarchy helps us understand the power of different synchronization primitives …
- … but collapses in practice!

Conclusion

- Herlihy's hierarchy helps us understand the power of different synchronization primitives …
- … but collapses in practice!
- Active research area:
    - ✦ new hierarchies?
    - ✦ can we get better algorithms by using less powerful primitives?

Conclusion

- Herlihy's hierarchy helps us understand the power of different synchronization primitives …
- … but collapses in practice!
- Active research area:
  - ✦ new hierarchies?
  - ✦ can we get better algorithms by using less powerful primitives?

Thanks!