Generalized Compare-and-Swap and Space-Efficient Universal Constructions for the Infinite-Arrival Model

Vassos Hadzilacos, **Myles Thiessen**, Sam Toueg

University of Toronto

Compare-and-Swap Object O

Compare-and-Swap Object O

CAS(val1, val2)

Compare-and-Swap Object O

```
CAS(val1, val2)
  current := O
```

Compare-and-Swap Object O

```
CAS(val1, val2)
  current := O
  if current = val1 then
```

Compare-and-Swap Object O

```
CAS(val1, val2)
  current := O
  if current = val1 then
    O := val2
```

Compare-and-Swap Object O

```
CAS(val1, val2)
  current := O
  if current = val1 then
    O := val2
    return True
```

Compare-and-Swap Object O

```
CAS(val1, val2)
  current := O
  if current = val1 then
    O := val2
    return True
  return False
```

Generalized Compare-and-Swap Object O

```
CAS(val1, val2)
  current := O
  if current = val1 then
    O := val2
    return True
  return False
```

```
GCAS(C, val1, val2)
  current := O
  if current C val1 then
    O := val2
    return True
  return False
```

Generalized Compare-and-Swap Object O

```
CAS(val1, val2)
  current := O
  if current = val1 then
    O := val2
    return True
  return False
```

```
GCAS(C, val1, val2)
  current := O
  if current C val1 then
    O := val2
    return True
  return False
```

For our purposes C is in {<, =, >}

Generalized Compare-and-Swap Object O

```
CAS(val1, val2)
  current := O
  if current = val1 then
    O := val2
    return True
  return False
```

```
GCAS(C, val1, val2)
  current := O
  if current C val1 then
    O := val2
    return True
  return False
```

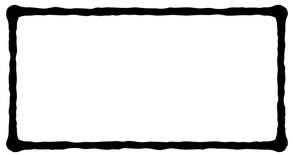
For our purposes C is in {<, =, >}, so the additional cost of implementing GCAS in hardware should be small relative to CAS...

So what does GCAS give us?

A simple **priority-based helping mechanism** for wait-freedom

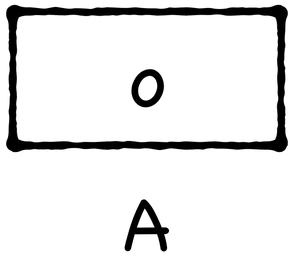
Help Mechanism — Basic Idea

Help Mechanism — Basic Idea

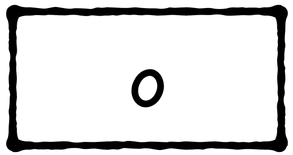


A

Help Mechanism — Basic Idea



Help Mechanism — Basic Idea

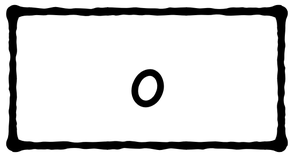


A

All processes **help** to execute the operation o currently in A

Help Mechanism — Basic Idea

To execute an operation o' process p

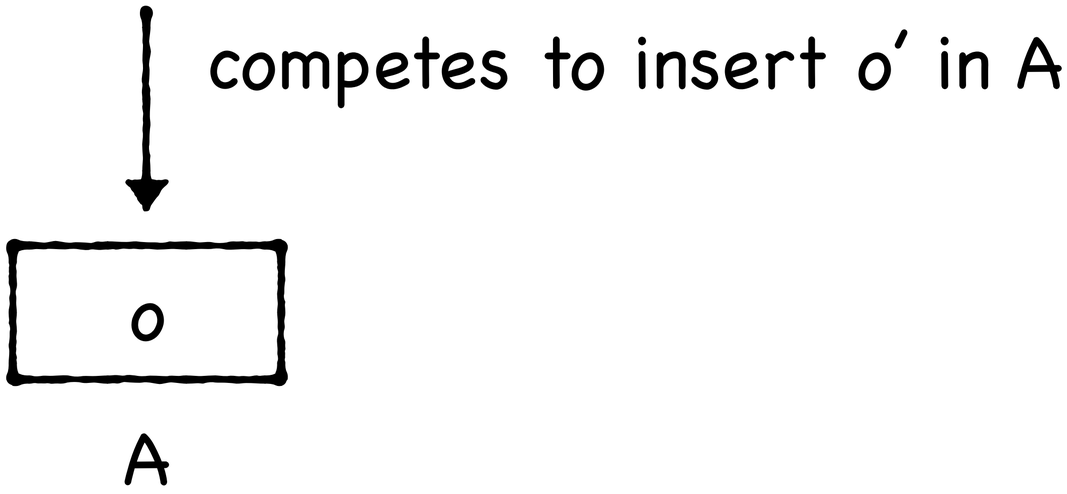


A

All processes **help** to execute the operation o currently in A

Help Mechanism — Basic Idea

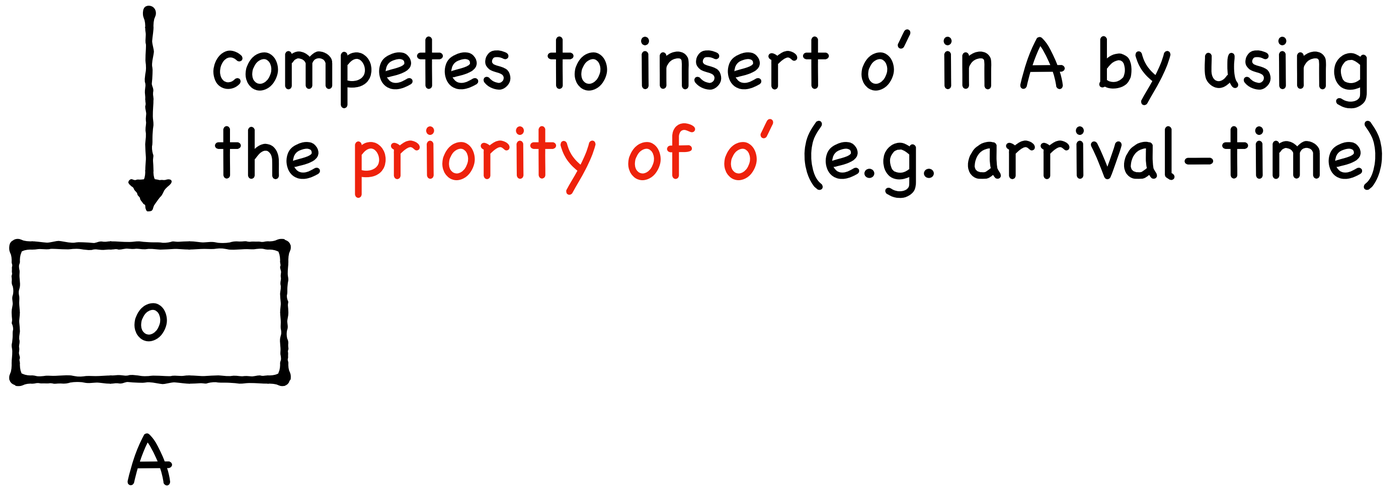
To execute an operation o' process p



All processes **help** to execute the operation o currently in A

Help Mechanism — Basic Idea

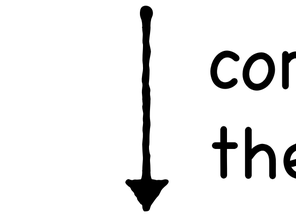
To execute an operation o' process p



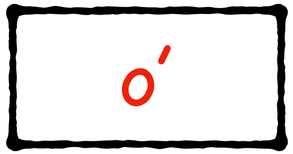
All processes **help** to execute the operation o currently in A

Help Mechanism — Basic Idea

To execute an operation o′ process p



competes to insert o′ in A by using the priority of o′ (e.g. arrival-time)

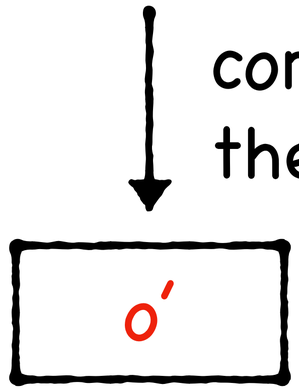


A

All processes **help** to execute the operation o′ currently in A

Help Mechanism — Basic Idea

To execute an operation o′ process p



competes to insert o′ in A by using the priority of o′ (e.g. arrival-time)

All processes **help** to execute the operation o′ currently in A

Eventually the highest priority operation stays in A until it is "done"

Help Mechanism — Using GCAS to Compete

Help Mechanism — Using GCAS to Compete

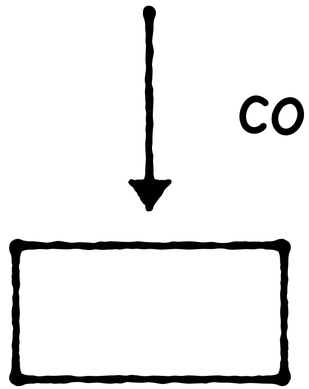
To execute an operation o′ process p

Help Mechanism — Using GCAS to Compete

To execute an operation (t', o') process p

Help Mechanism — Using GCAS to Compete

To execute an operation (t′, o′) process p

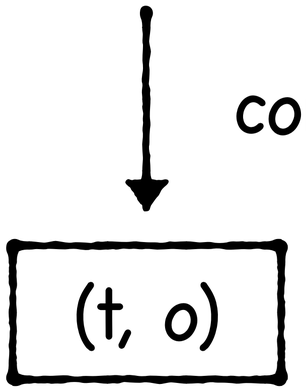


competes to place (t′, o′) into

GCAS "announcement" object A

Help Mechanism — Using GCAS to Compete

To execute an operation (t′, o′) process p

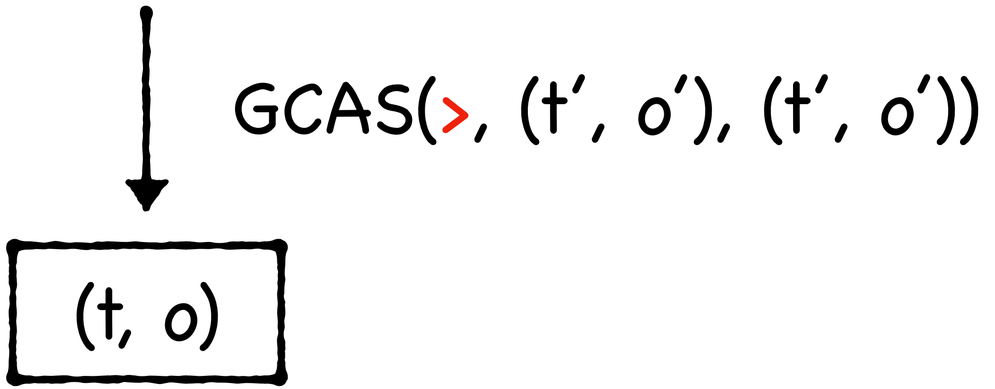


competes to place (t′, o′) into

GCAS "announcement" object A

Help Mechanism — Using GCAS to Compete

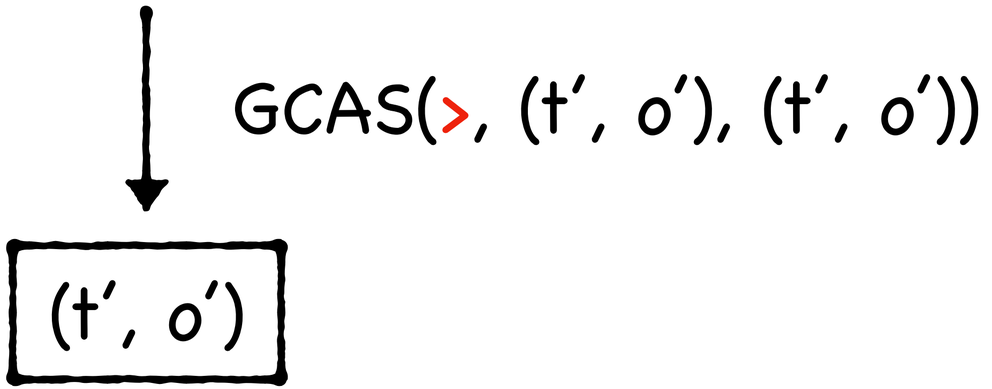
To execute an operation (t′, o′) process p



GCAS "announcement" object A

Help Mechanism — Using GCAS to Compete

To execute an operation (t′, o′) process p

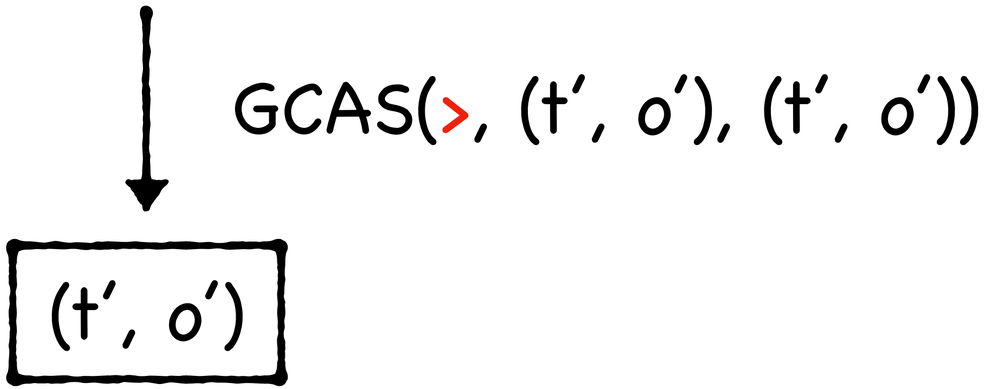


GCAS "announcement" object A

- Eventually (t′, o′) enters A and stays there until o′ is "done"

Help Mechanism — Using GCAS to Compete

To execute an operation (t′, o′) process p

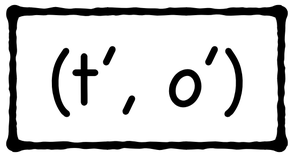


GCAS "announcement" object A

- Eventually (t′, o′) enters A and stays there until o′ is "done"
- But how do processes **remove** (t′, o′) from A after o′ is "done"?

Help Mechanism — Using GCAS to Compete

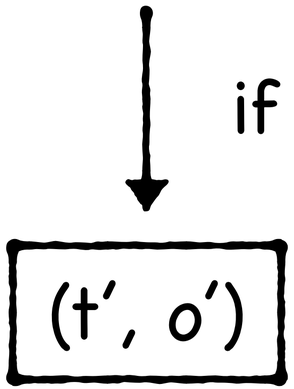
To execute an operation (t*, o*) process q



GCAS "announcement" object A

Help Mechanism — Using GCAS to Compete

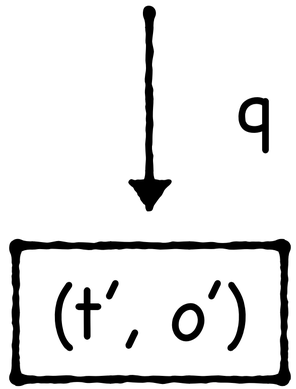
To execute an operation (t*, o*) process q



GCAS "announcement" object A

Help Mechanism — Using GCAS to Compete

To execute an operation (t*, o*) process q

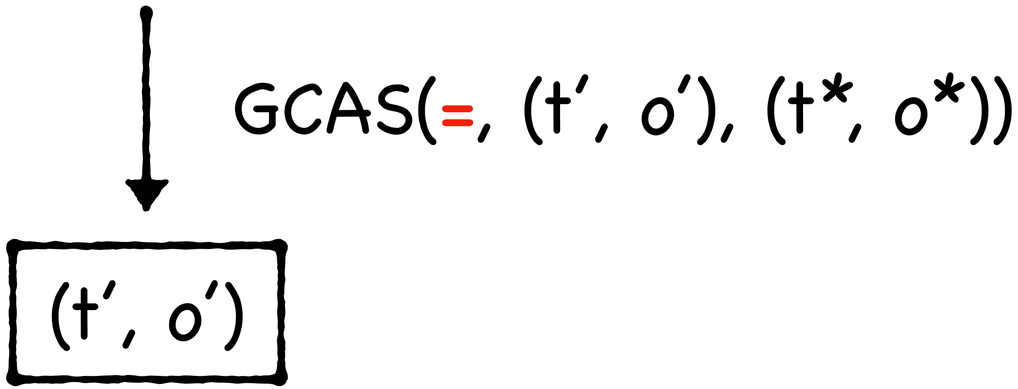


q tries to write (t*, o*) into A

GCAS "announcement" object A

Help Mechanism — Using GCAS to Compete

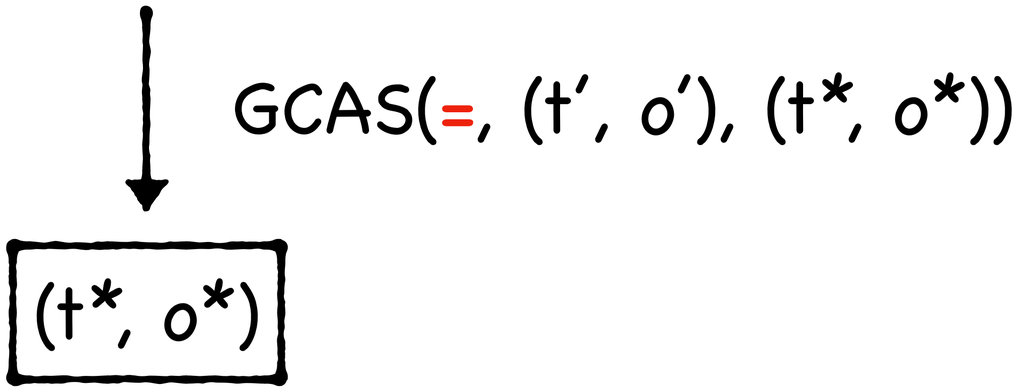
To execute an operation (t*, o*) process q



GCAS "announcement" object A

Help Mechanism — Using GCAS to Compete

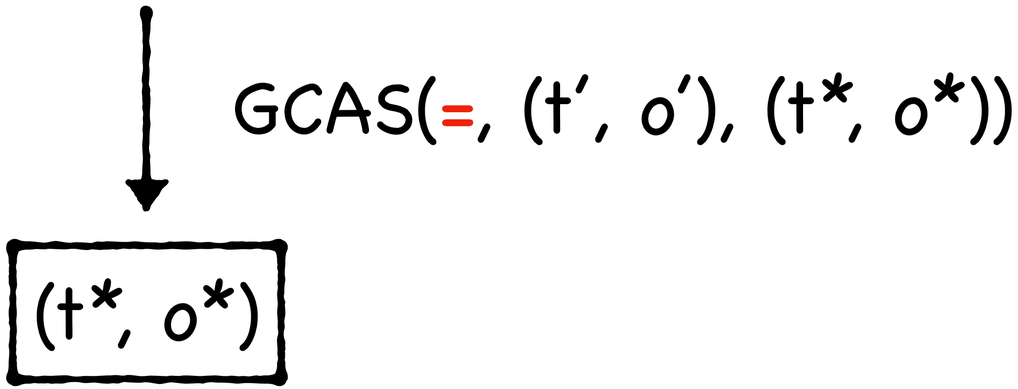
To execute an operation (t*, o*) process q



GCAS "announcement" object A

Help Mechanism — Using GCAS to Compete

To execute an operation (t*, o*) process q

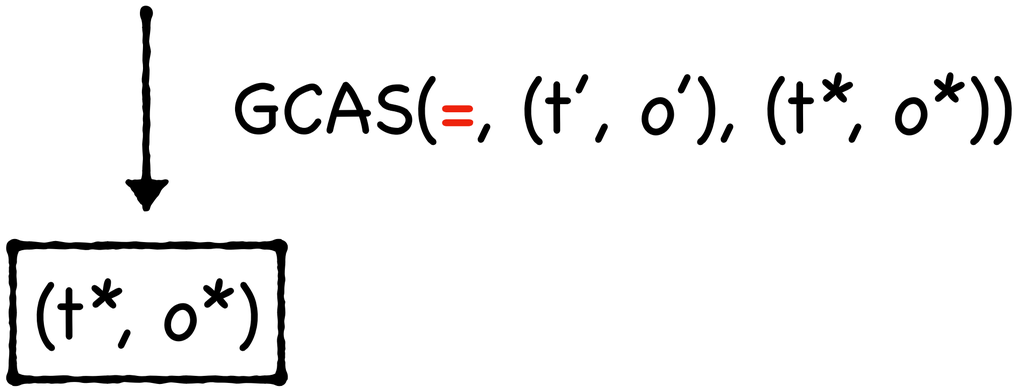


GCAS "announcement" object A

Note: we used both the > and = comparators of GCAS

Help Mechanism — Using GCAS to Compete

To execute an operation (t*, o*) process q



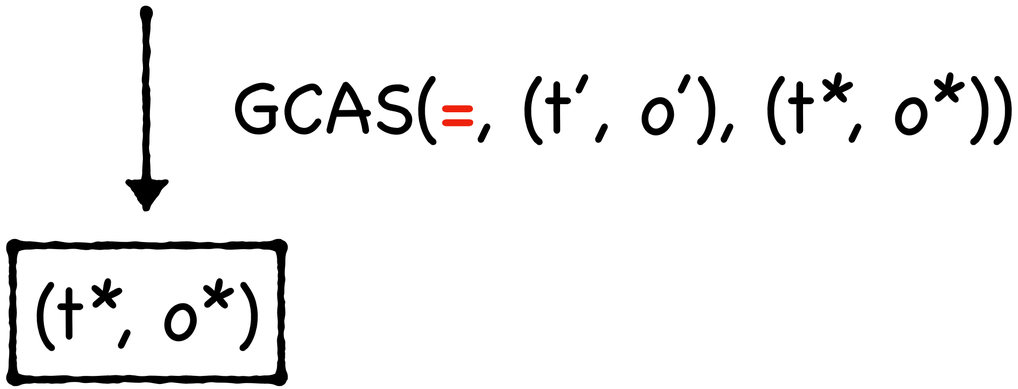
GCAS "announcement" object A

Note: we used both the > and = comparators of GCAS

- Comparator > : to compete on priorities

Help Mechanism — Using GCAS to Compete

To execute an operation (t*, o*) process q



GCAS "announcement" object A

Note: we used both the > and = comparators of GCAS

- Comparator > : to compete on priorities
- Comparator = : to dislodge an operation that was "done"

Two Space-Efficient Wait-free Universal Constructions for the Infinite-Arrival Model

Two Space-Efficient Wait-free Universal Constructions for the **Infinite-Arrival** Model

Two Space-Efficient Wait-free Universal Constructions for the **Infinite-Arrival** Model

There are **infinitely many processes**

Two Space-Efficient Wait-free Universal Constructions for the **Infinite-Arrival** Model

There are **infinitely many processes**

So we do **not** assume the number n of processes is known or fixed

Two Space-Efficient Wait-free Universal Constructions for the Infinite-Arrival Model

(1) Space linear in the # of processes that have participated so far

Two Space-Efficient Wait-free Universal Constructions for the Infinite-Arrival Model

(1) Space linear in the # of processes that have participated so far

(2) Space linear in the point contention

Two Space-Efficient Wait-free Universal Constructions for the Infinite-Arrival Model

(1) Space linear in the # of processes that have participated so far

(2) Space linear in the point contention

- uses a novel memory recycling scheme

Two Space-Efficient Wait-free Universal Constructions for the Infinite-Arrival Model

(1) Space linear in the # of processes that have participated so far

(2) Space linear in the point contention

- uses a novel memory recycling scheme
- but assumes bounded concurrency

Two Space-Efficient Wait-free Universal Constructions for the Infinite-Arrival Model

(1) Space linear in the # of processes that have participated so far

(2) Space linear in the point contention

- uses a novel memory recycling scheme
- but assumes bounded concurrency

Two Space-Efficient Wait-free Universal Constructions for the Infinite-Arrival Model

(1) Space linear in the # of processes that have participated so far

(2) Space linear in the point contention

- uses a novel memory recycling scheme
- but assumes bounded concurrency

Two Space-Efficient Wait-free Universal Constructions for the Infinite-Arrival Model

(1) Space linear in the # of processes that have participated so far

(2) Space linear in the point contention

- uses a novel memory recycling scheme
- but assumes bounded concurrency

Two Space-Efficient Wait-free Universal Constructions for the Infinite-Arrival Model

(1) Space linear in the # of processes that have participated so far

- **Time** linear in the point contention

(2) Space linear in the point contention

- uses a novel memory recycling scheme
- but assumes bounded concurrency

Related Work

Universal Constructions for the Infinite-Arrival Model

Universal Constructions for the Infinite-Arrival Model

[Aspnes,Shah,Shah PODC'02]: space infinite in every run

Universal Constructions for the Infinite-Arrival Model

[Aspnes,Shah,Shah PODC'02]: space infinite in every run

[Perrin,Mostéfaoui,Bonin PODC'20]: space linear in the # of operations

Universal Constructions for the Infinite-Arrival Model

[Aspnes,Shah,Shah PODC'02]: space infinite in every run

[Perrin,Mostéfaoui,Bonin PODC'20]: space linear in the # of operations

[Bédin,Lépine,Mostéfaoui,Perez,Perrin DISC'21]: assumes automatic memory recycling scheme for the infinite-arrival model

Universal Constructions for the Infinite-Arrival Model

[Aspnes,Shah,Shah PODC'02]: space infinite in every run

[Perrin,Mostéfaoui,Bonin PODC'20]: space linear in the # of operations

[Bédin,Lépine,Mostéfaoui,Perez,Perrin DISC'21]: assumes automatic memory recycling scheme for the infinite-arrival model

Memory Recycling (a lot of work on this topic!)

Universal Constructions for the Infinite-Arrival Model

[Aspnes,Shah,Shah PODC'02]: space infinite in every run

[Perrin,Mostéfaoui,Bonin PODC'20]: space linear in the # of operations

[Bédin,Lépine,Mostéfaoui,Perez,Perrin DISC'21]: assumes automatic memory recycling scheme for the infinite-arrival model

Memory Recycling (a lot of work on this topic!)

[Herlihy,Luchangco,Moir'02], [Michael'02+04], [Brown'15], [Anderson,Blelloch,Wei'21]: do not work in the infinite-arrival model

Universal Constructions for the Infinite-Arrival Model

[Aspnes,Shah,Shah PODC'02]: space infinite in every run

[Perrin,Mostéfaoui,Bonin PODC'20]: space linear in the # of operations

[Bédin,Lépine,Mostéfaoui,Perez,Perrin DISC'21]: assumes automatic memory recycling scheme for the infinite-arrival model

Memory Recycling (a lot of work on this topic!)

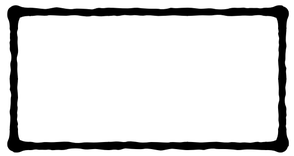
[Herlihy,Luchangco,Moir'02], [Michael'02+04], [Brown'15], [Anderson,Blelloch,Wei'21]: do not work in the infinite-arrival model

[Detlefs,Martin,Moir,Steele'01], [Herlihy,Luchangco,Martin,Moir'05], [Nikolaev,Ravindran'21]: work for this model but are not wait-free

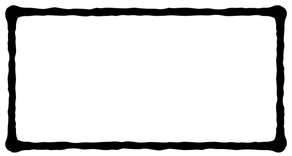
Our first universal construction

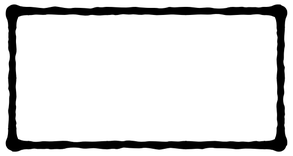
Base Objects

Base Objects

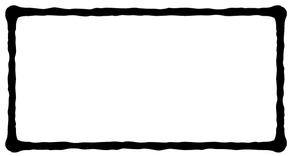
C  a Fetch&Inc to timestamp operations

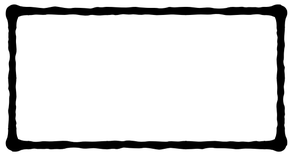
Base Objects

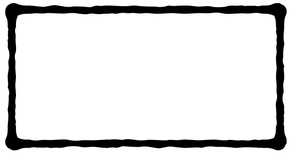
C  a Fetch&Inc to timestamp operations

R_p  a CAS to store the response of the last operation of p

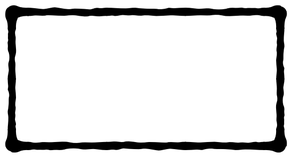
Base Objects

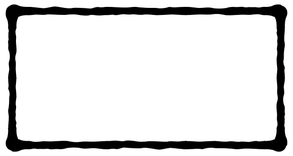
C  a Fetch&Inc to timestamp operations

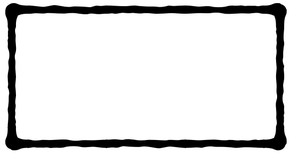
R_p  a CAS to store the response of the last operation of p

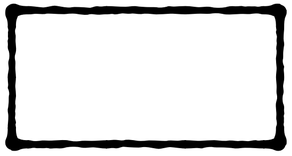
A  a GCAS "announcement" object

Base Objects

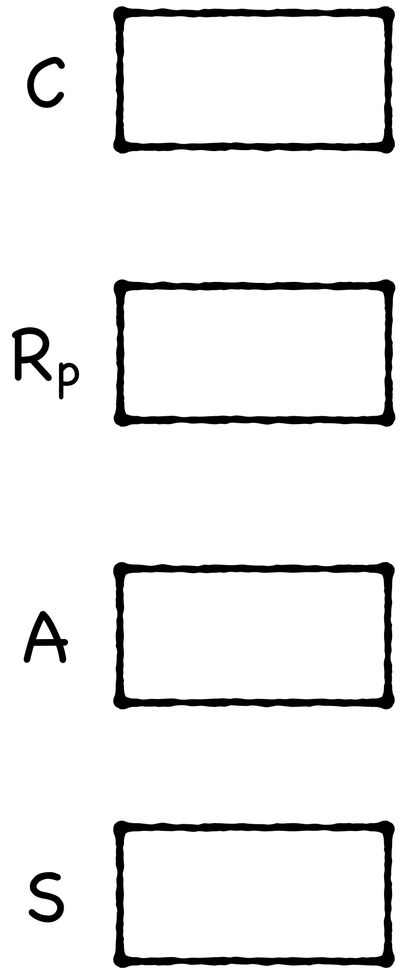
C  a Fetch&Inc to timestamp operations

R_p  a CAS to store the response of the last operation of p

A  a GCAS "announcement" object

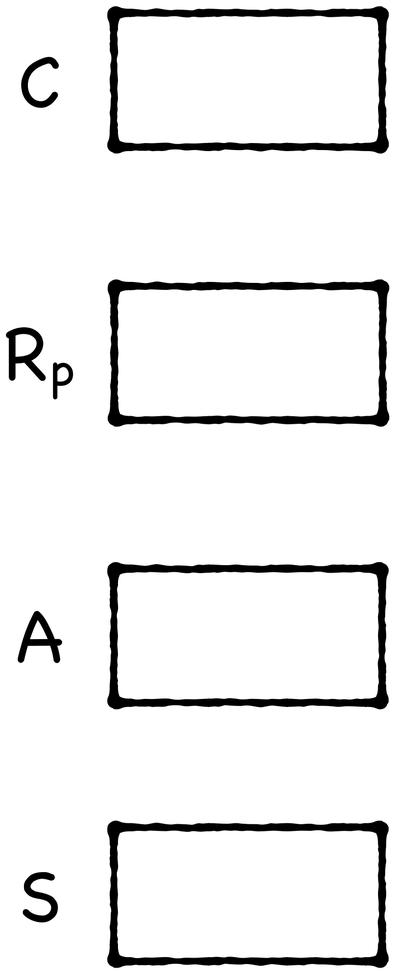
S  a CAS to store the state of the implemented object

Base Objects




```
DoOp(o) for process p
  if o is the first op of p then Rp := get object from manager
  t := C.FI()
  Rp := (t, NIL)
  while Rp = (t, NIL) do
    (t*, s*, r*, R*) := S
    R*.CAS((t*, NIL), (t*, r*))
    A.GCAS(>, (t, o, Rp), (t, o, Rp))
    (t', o', R') := A
    if R' = (t', NIL) then
      (s', r') := apply(o', s*)
      S.CAS((t*, s*, r*, R*), (t', s', r', R'))
    else
      A.GCAS(=, (t', o', R'), (t, o, Rp))
  r := Rp.response
  return r
```

Base Objects




```
DoOp(o) for process p
  if o is the first op of p then Rp := get object from manager
  t := C.FI()
  Rp := (t, NIL)
  while Rp = (t, NIL) do
    (t*, s*, r*, R*) := S
    R*.CAS((t*, NIL), (t*, r*))
    A.GCAS(>, (t, o, Rp), (t, o, Rp))
    (t', o', R') := A
    if R' = (t', NIL) then
      (s', r') := apply(o', s*)
      S.CAS((t*, s*, r*, R*), (t', s', r', R'))
    else
      A.GCAS(=, (t', o', R'), (t, o, Rp))
  r := Rp.response
  return r
```

Base Objects

C

R_p

A

S

DoOp(o) for process p

if o is the **first** op of p then R_p := get object from manager

t := C.FI()

R_p := (t, NIL)

while R_p =

(t*, s*

R*.CA

A.GC

(t', o

if R'

(s', r

S.CAS(

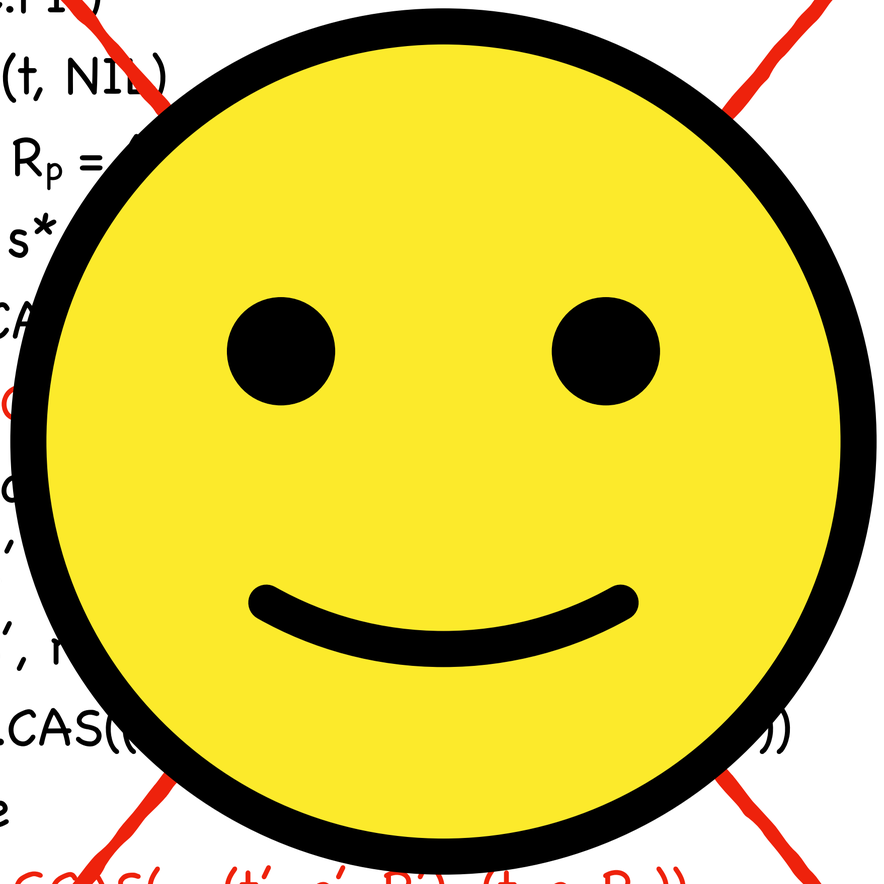
))

else

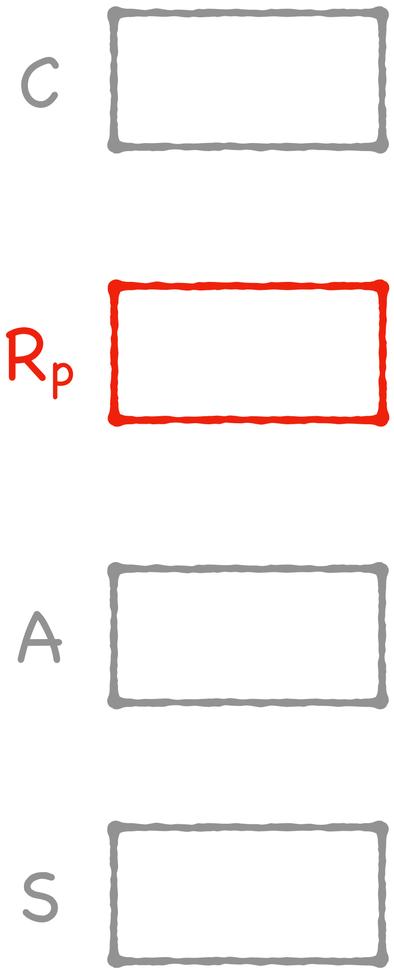
A.GCAS(=, (t', o', R'), (t, o, R_p))

r := R_p.response

return r

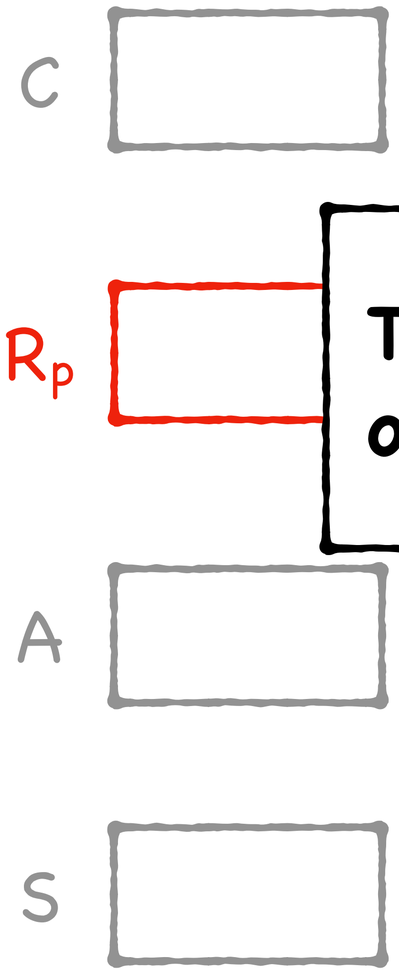


Base Objects




```
DoOp(o) for process p
  if o is the first op of p then Rp := get object from manager
  t := C.FI()
  Rp := (t, NIL)
  while Rp = (t, NIL) do
    (t*, s*, r*, R*) := S
    R*.CAS((t*, NIL), (t*, r*))
    A.GCAS(>, (t, o, Rp), (t, o, Rp))
    (t', o', R') := A
    if R' = (t', NIL) then
      (s', r') := apply(o', s*)
      S.CAS((t*, s*, r*, R*), (t', s', r', R'))
    else
      A.GCAS(=, (t', o', R'), (t, o, Rp))
  r := Rp.response
  return r
```

Base Objects




```
DoOp(o) for process p
  if o is the first op of p then R_p := get object from manager
  t := C.FI()
  R_p := (t, NIL)
  while R_p = (t, NIL) do
    (t*, s*, r*, R*) := S
    (s', r') := apply(o', s*)
    S.CAS((t*, s*, r*, R*), (t', s', r', R'))
  else
    A.GCAS(=, (t', o', R'), (t, o, R_p))
  r := R_p.response
  return r
```

The space complexity is linear in the number of processes that have participated so far

Memory recycling in our second universal construction

Memory Manager

Memory Manager

Processes can:

Memory Manager

Processes can:

(1) get "cells" from the memory manager, and

Memory Manager

Processes can:

(1) get "cells" from the memory manager, and

(2) free "cells" by returning them to the memory manager

Memory Manager

Processes can:

(1) get "cells" from the memory manager, and

(2) free "cells" by returning them to the memory manager

Warning:

(1) the memory manager may give any freed cell to another application, and this application may write "junk" into it, so

Memory Manager

Processes can:

(1) get "cells" from the memory manager, and

(2) free "cells" by returning them to the memory manager

Warning:

(1) the memory manager may give any freed cell to another application, and this application may write "junk" into it, so

(2) processes should never access a cell that has been freed

Memory Manager

Processes can:

(1) get "cells" from the memory manager, and

(2) free "cells" by returning them to the memory manager

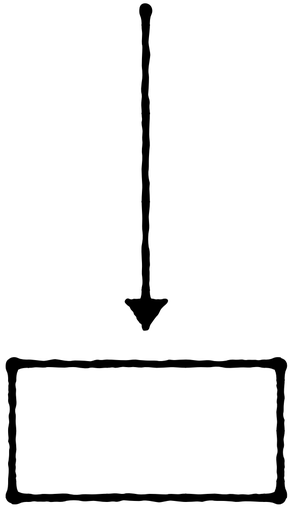
Warning:

(1) the memory manager may give any freed cell to another application, and this application may write "junk" into it, so

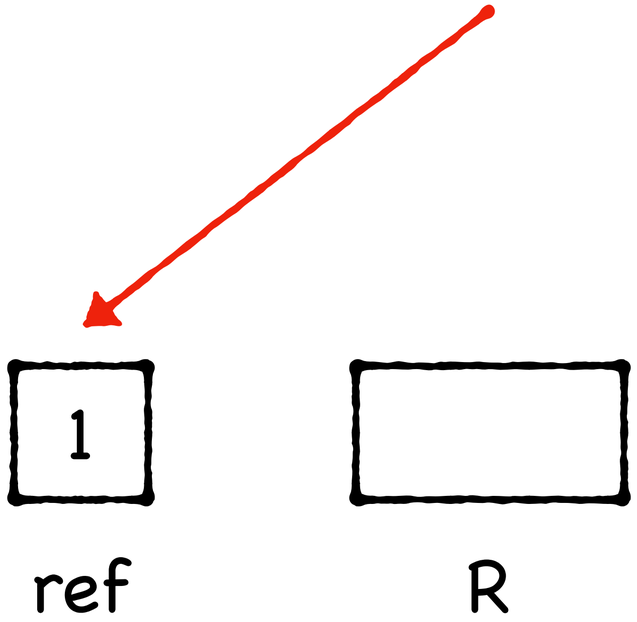
(2) processes should never access a cell that has been freed (as otherwise it may return a wrong value)

Reference Counters

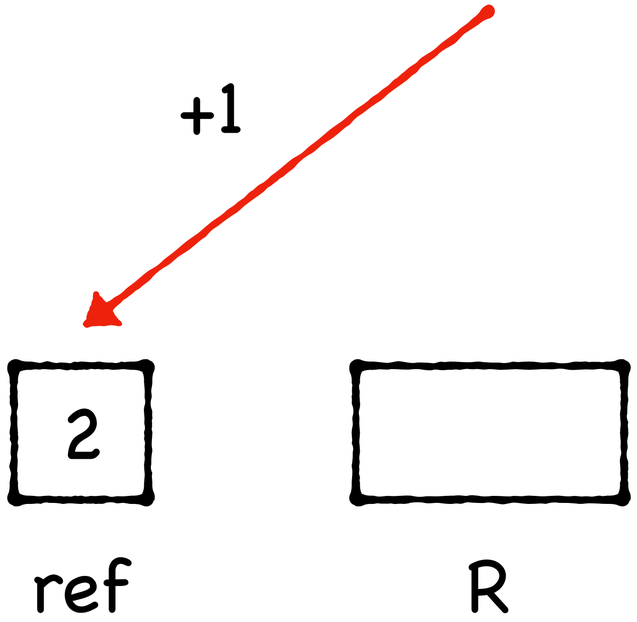
Before process p accesses cell R it must gain the "right" to access R



p does so by incrementing the reference counter for R

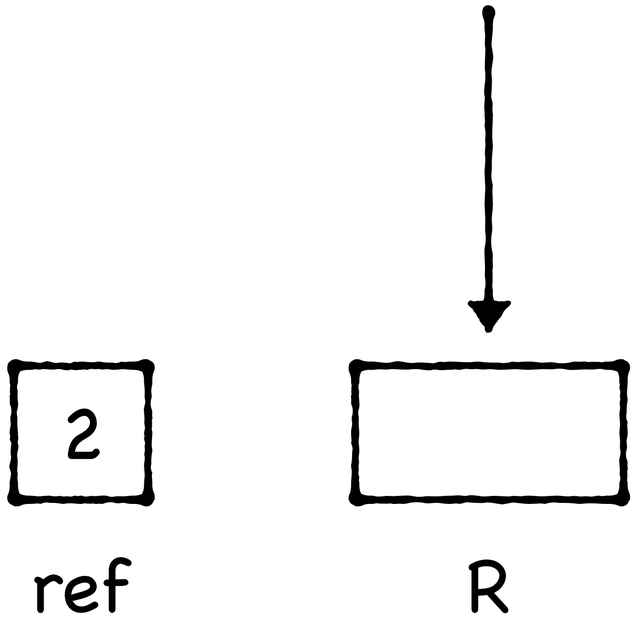


p does so by incrementing the reference counter for R

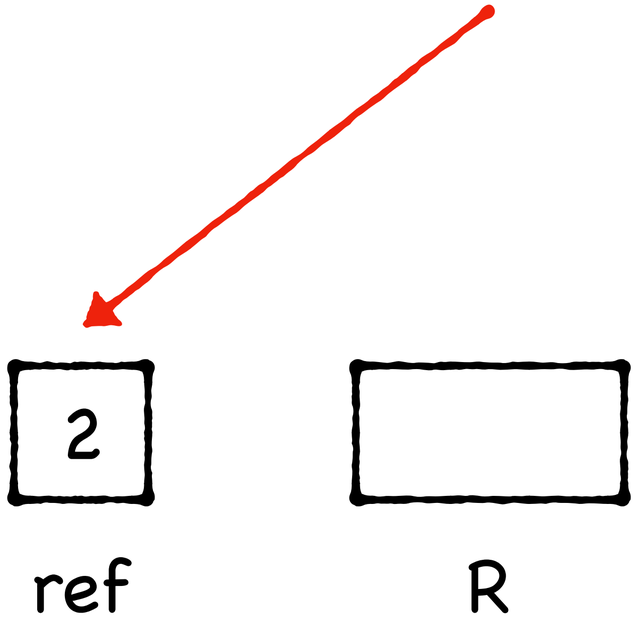


Now p has the "right" to access R

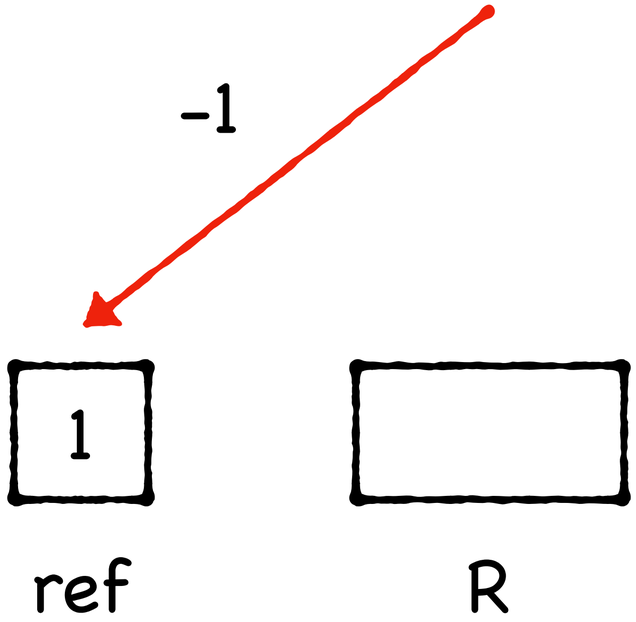
After p is done accessing cell R it relinquishes its "right" to access R



p does so by decrementing the reference counter for R

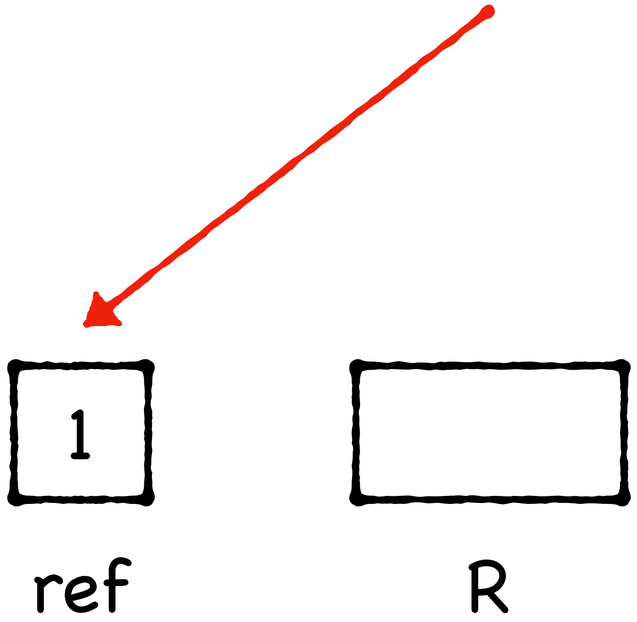


p does so by decrementing the reference counter for R

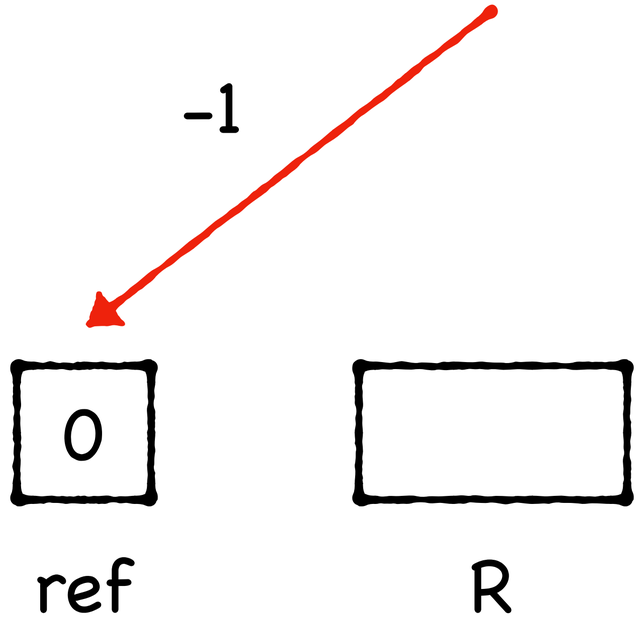


Now p relinquished access to R

To relinquishes its access to R, q decrements the reference counter for R

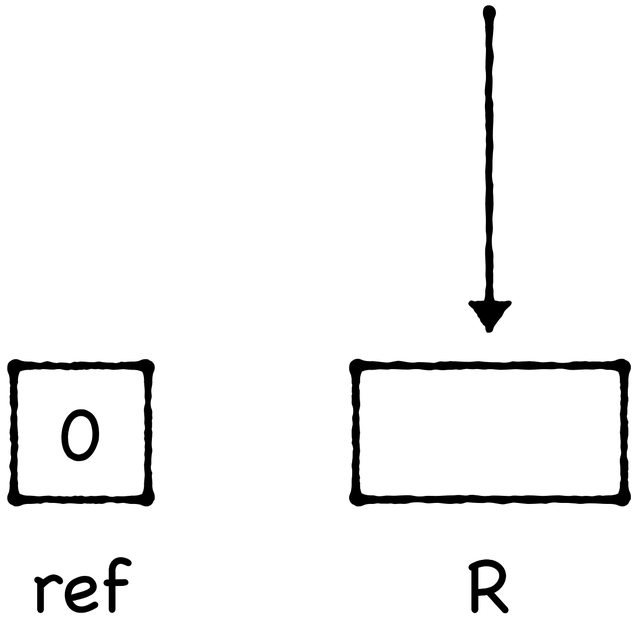


To relinquishes its access to R, q decrements the reference counter for R

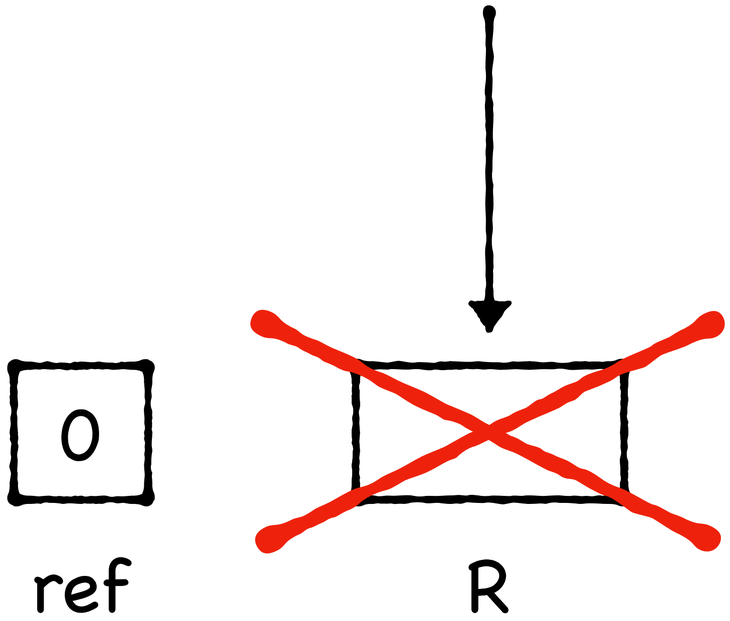


Now q relinquished access to R

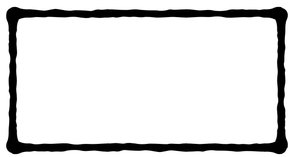
Now no process has the right to access R



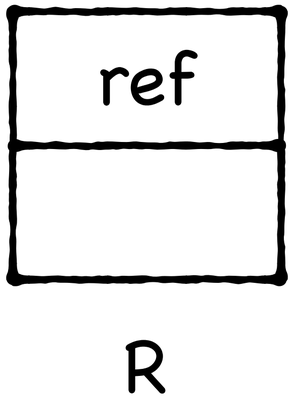
Now no process has the right to access R, so q can safely recycle it



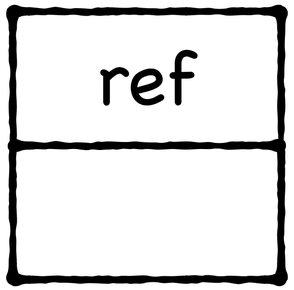
But where do we put the reference counter for R?



R

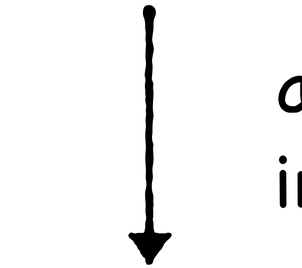

ref
R

To acquire “the right” to access R

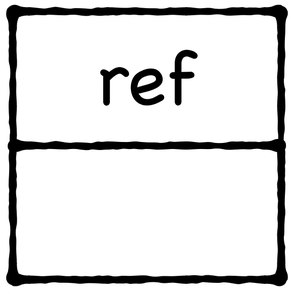


R

To acquire "the right" to access R

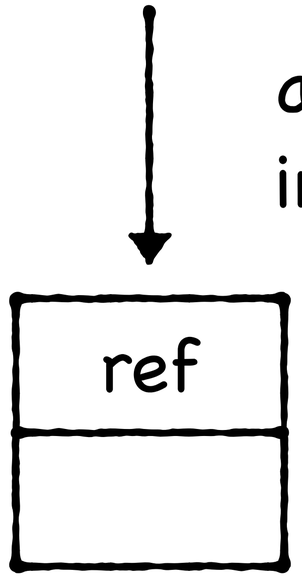


a process must access R to increment its reference counter!



R

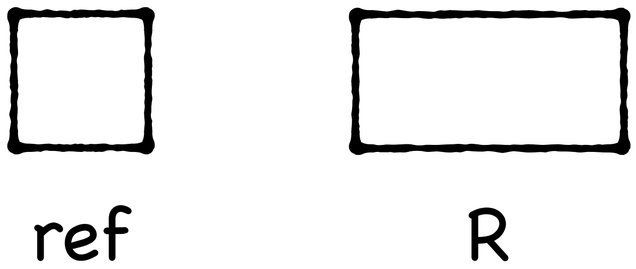
To acquire "the right" to access R



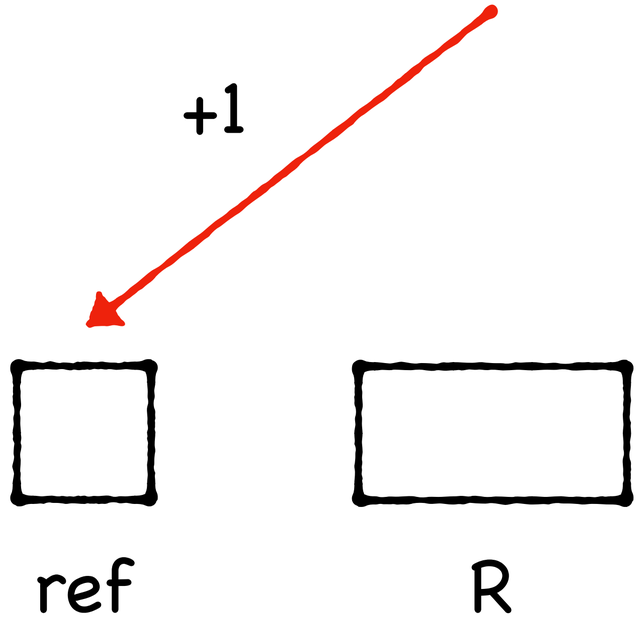
a process must access R to increment its reference counter!

R

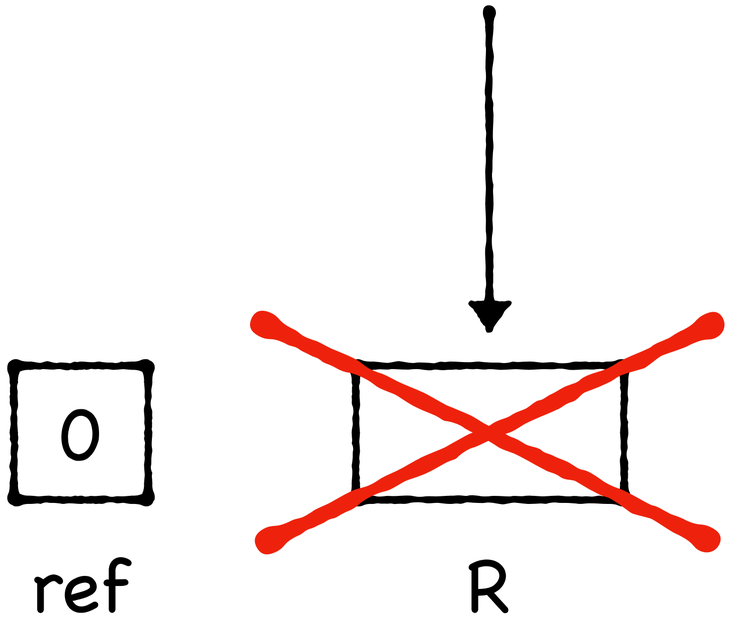
A chicken-and-egg situation


ref
R

To acquire "the right" to access R we no longer need to access R!

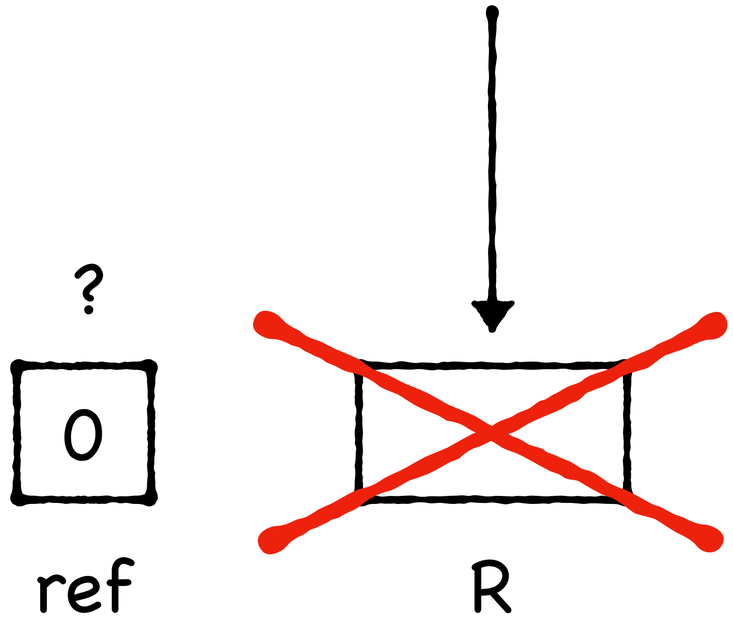


We can safely recycle a cell when its reference counter is zero



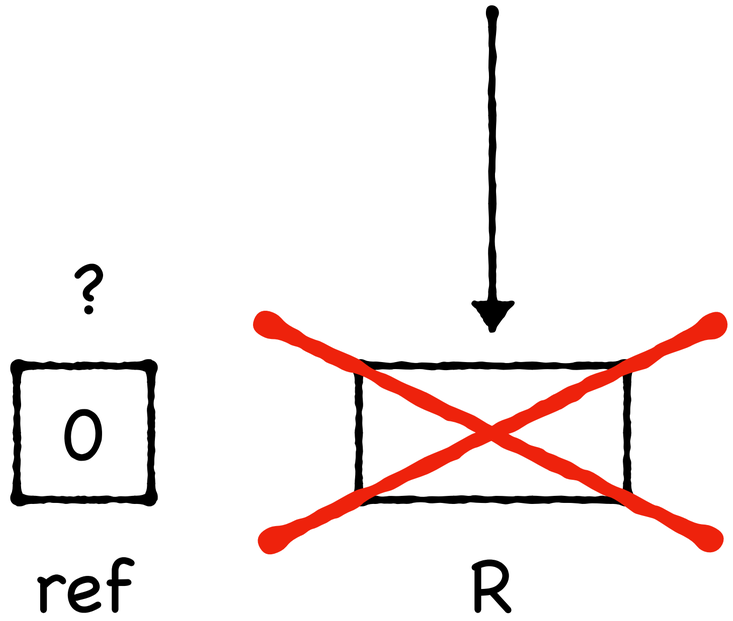
We can safely recycle a cell when its reference counter is zero

The problem: when can we safely recycle its reference counter?



We can safely recycle a cell when its reference counter is zero

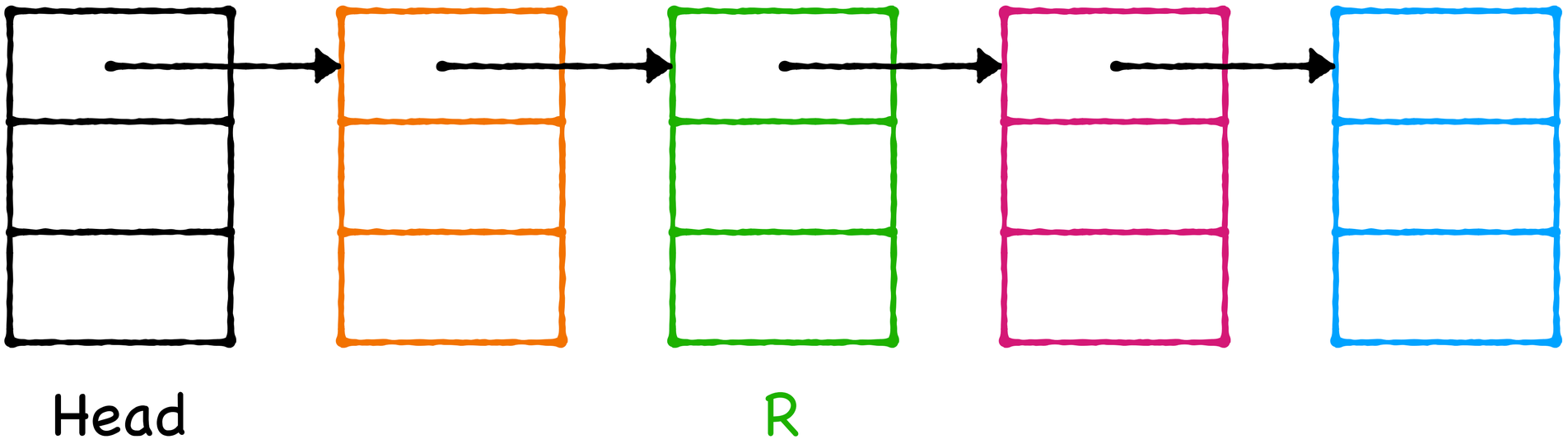
The problem: when can we safely recycle its reference counter?



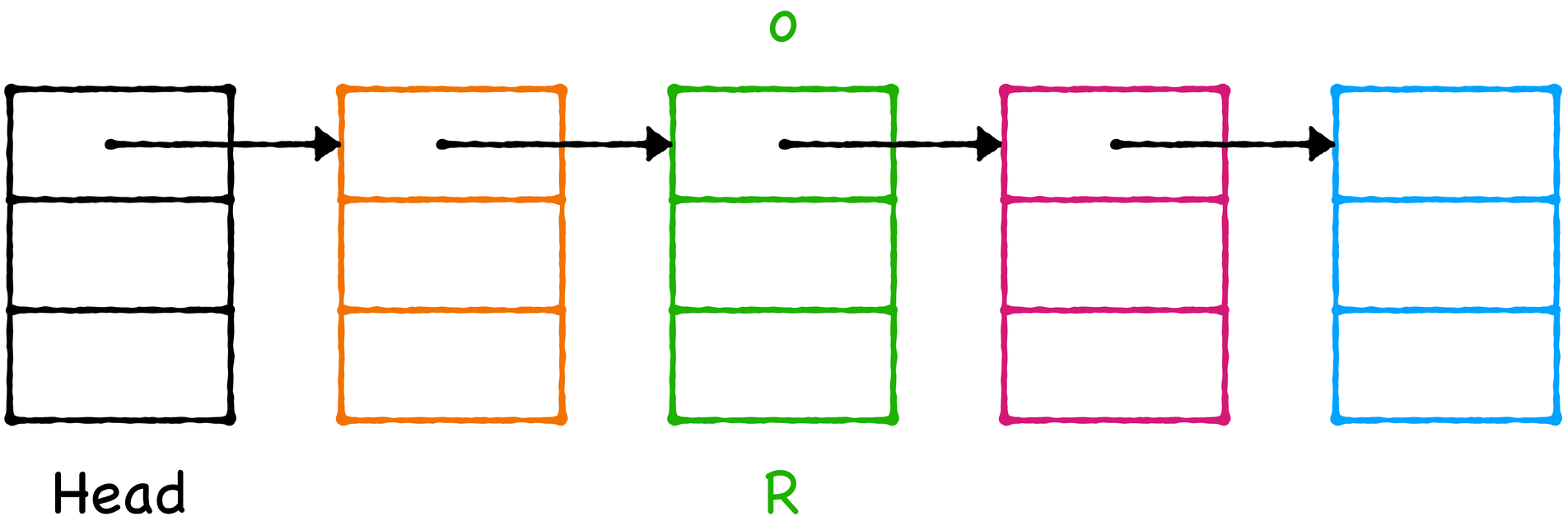
A different kind of chicken-and-egg situation!

Solving the chicken-and-egg problems

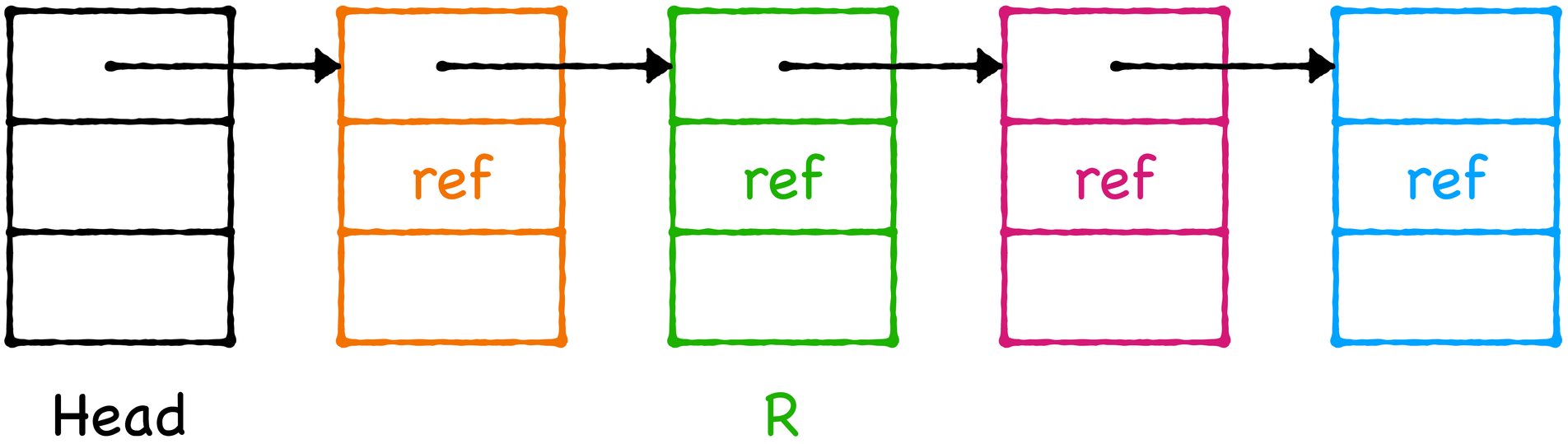
- Thread the response cells of pending operations into a list



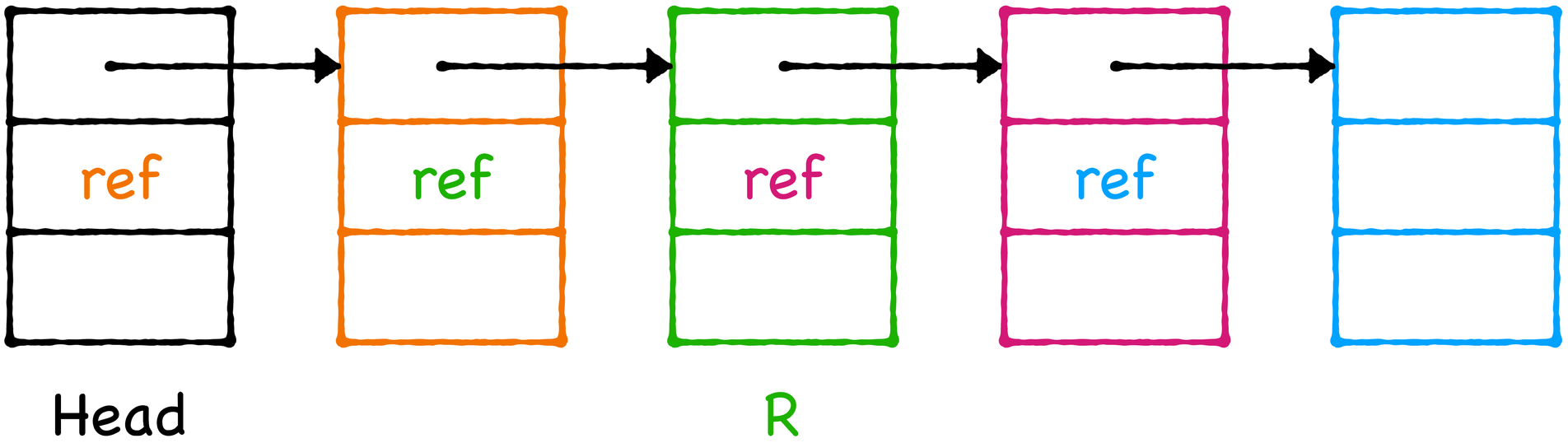
- Thread the response cells of pending operations into a list



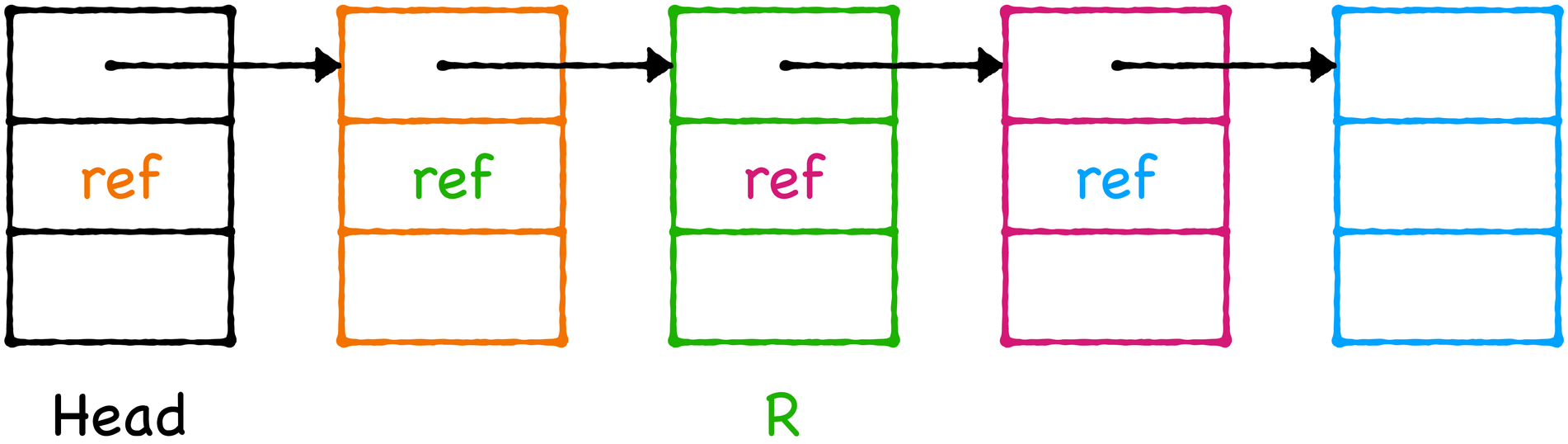
- Thread the response cells of pending operations into a list
- Put the reference counter of each cell into the cell itself



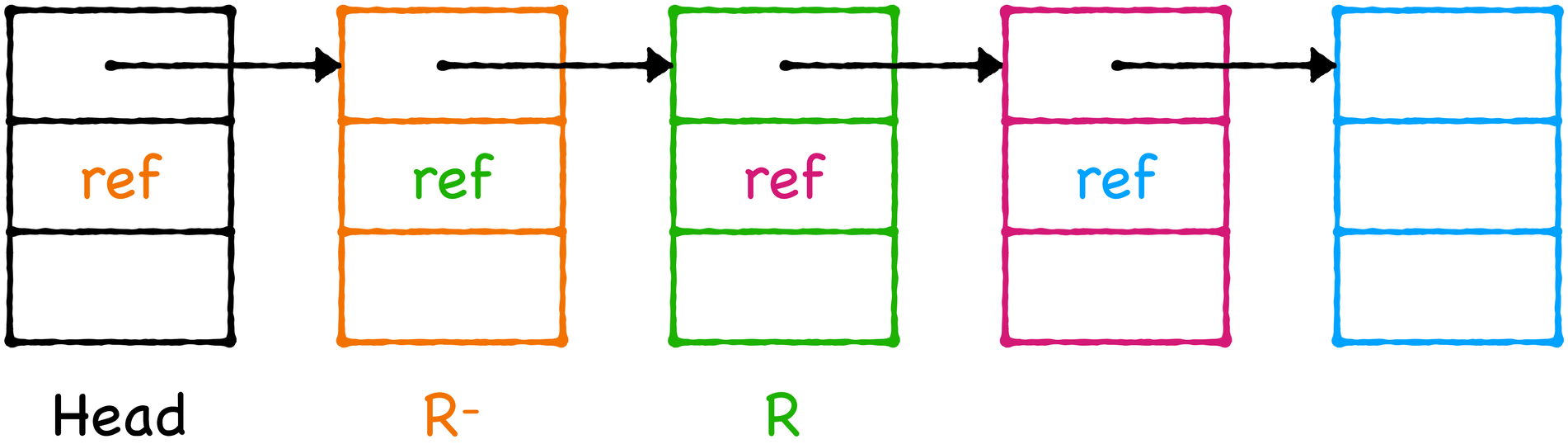
- Thread the response cells of pending operations into a list
- Put the reference counter of each cell into the cell itself

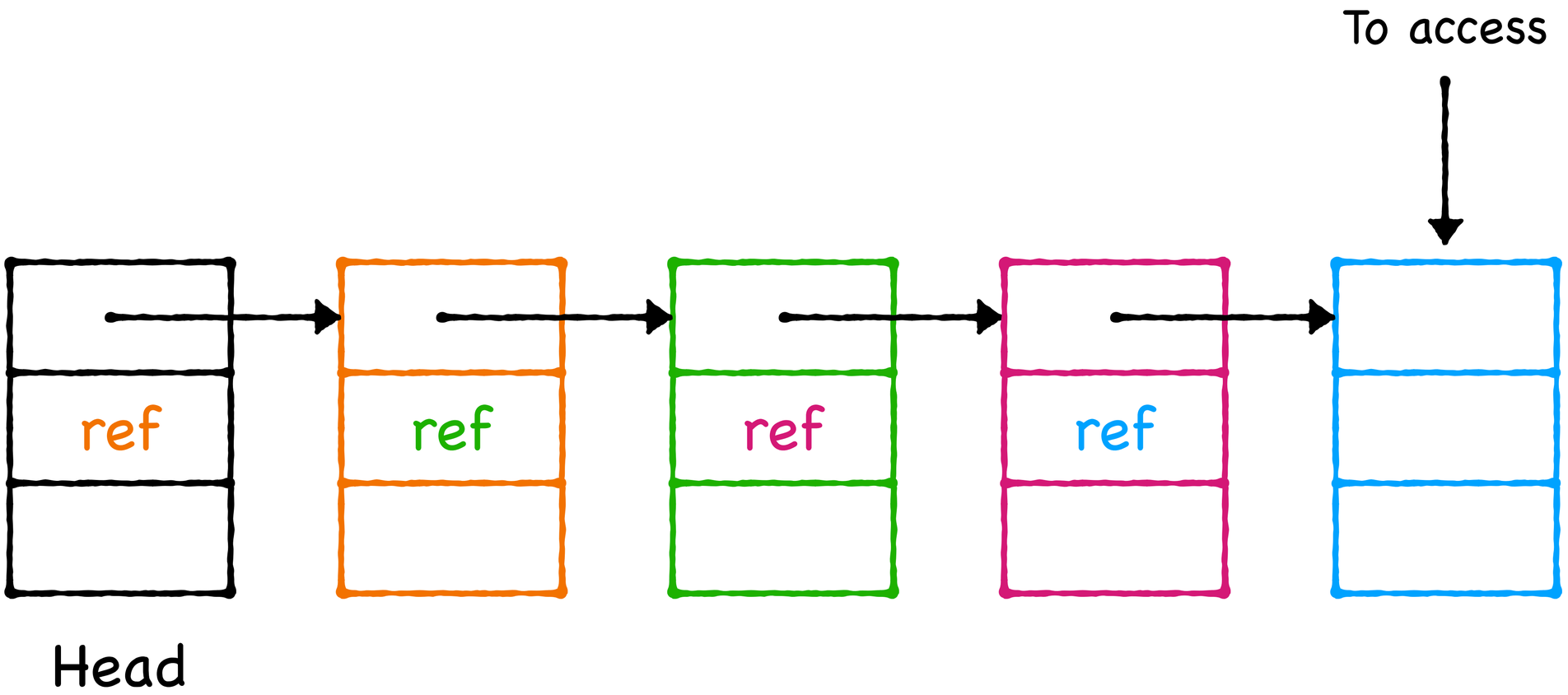


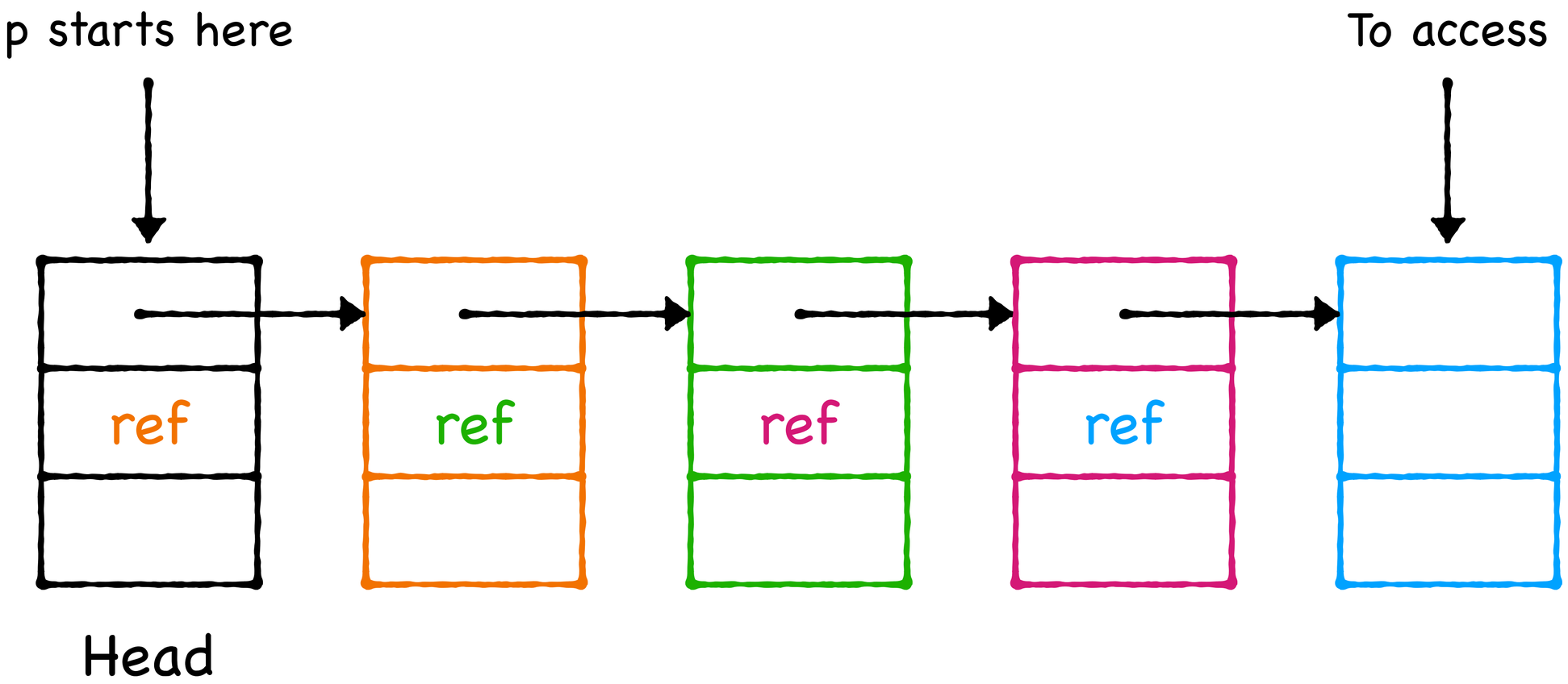
- Thread the response cells of pending operations into a list
- Put the reference counter of each cell into ~~the cell itself~~ its predecessor!

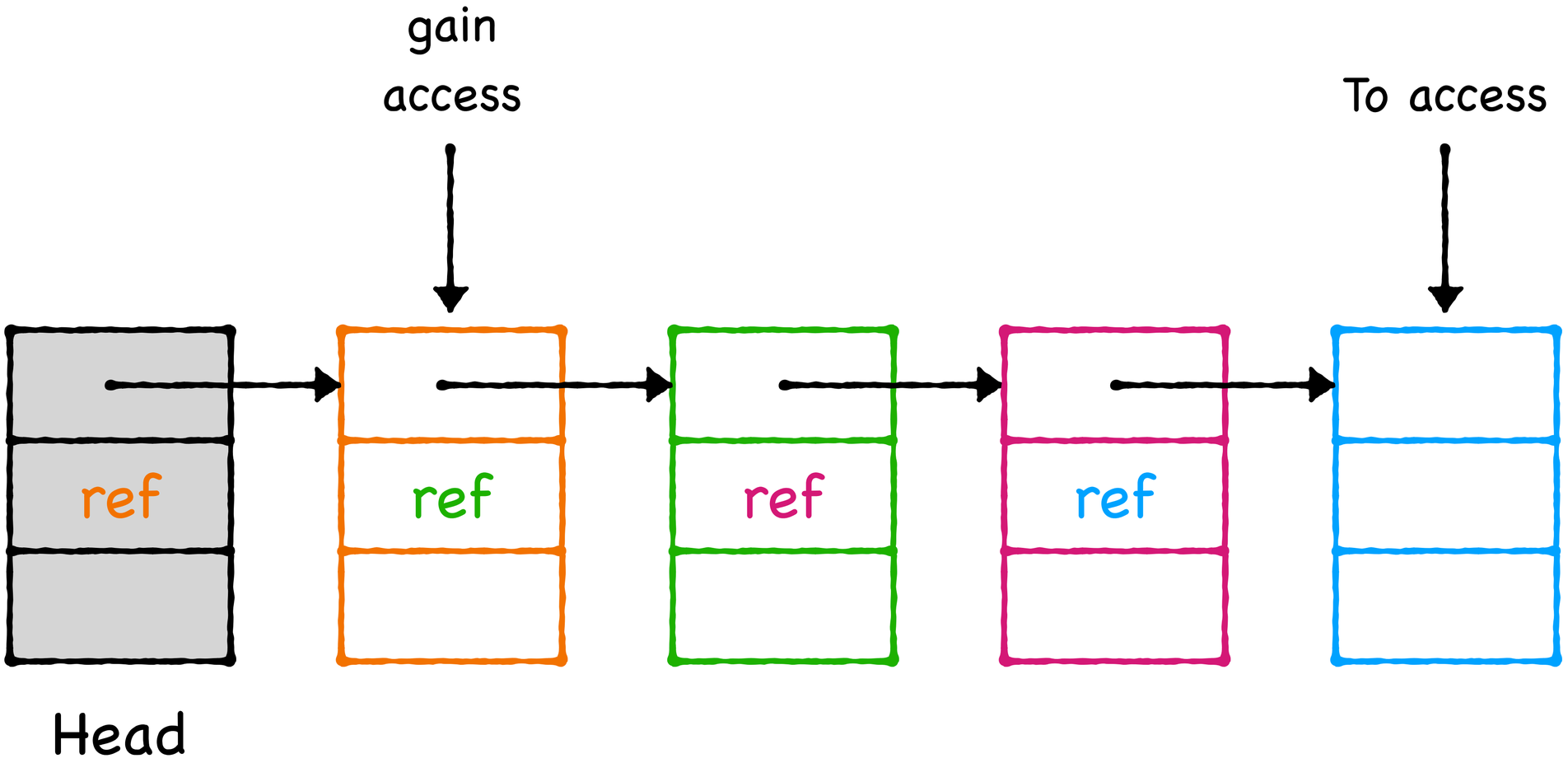


- Thread the response cells of pending operations into a list
- Put the reference counter of each cell into ~~the cell itself~~ its predecessor!

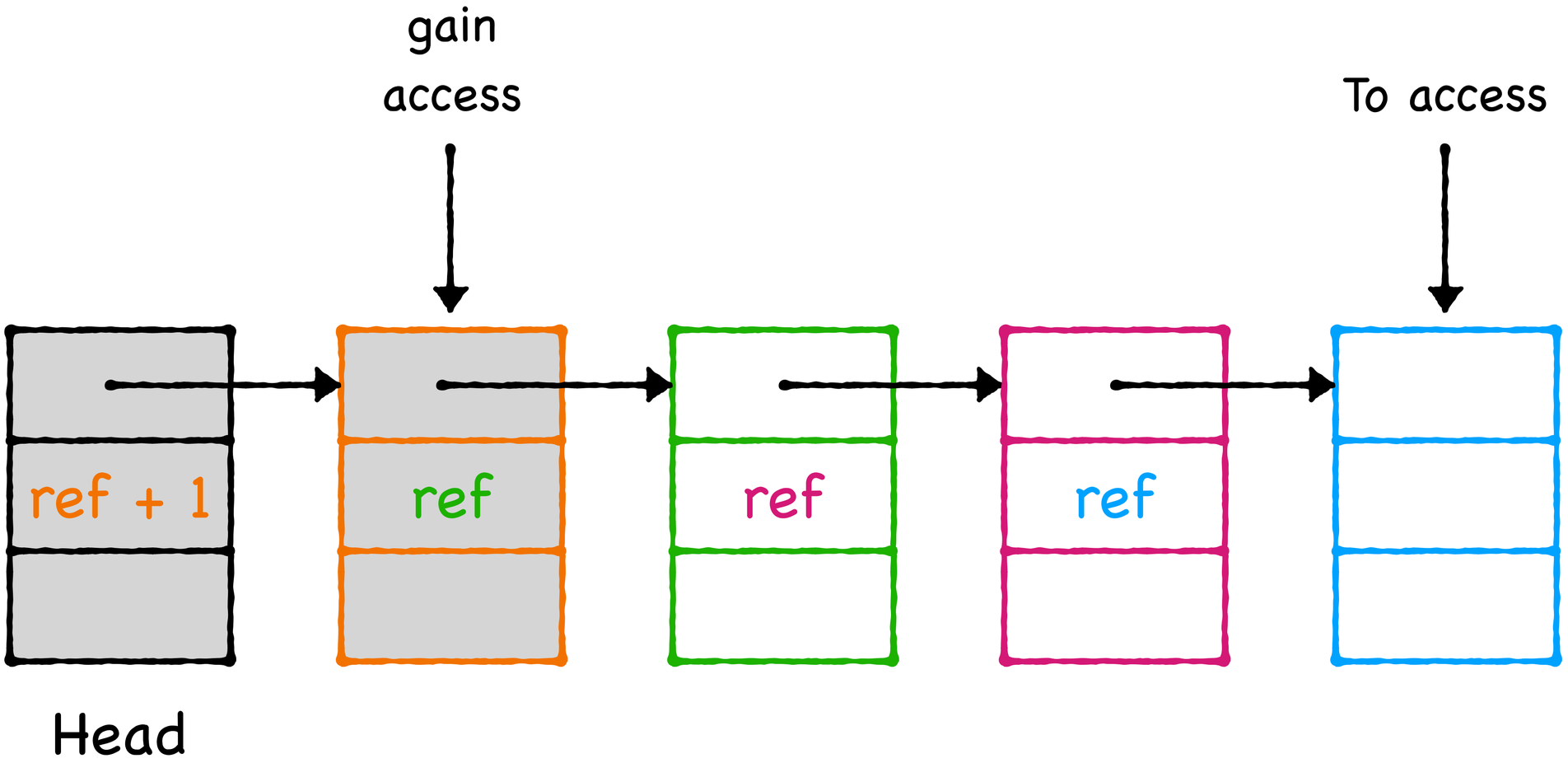



To access
ref
ref
ref
ref
Head

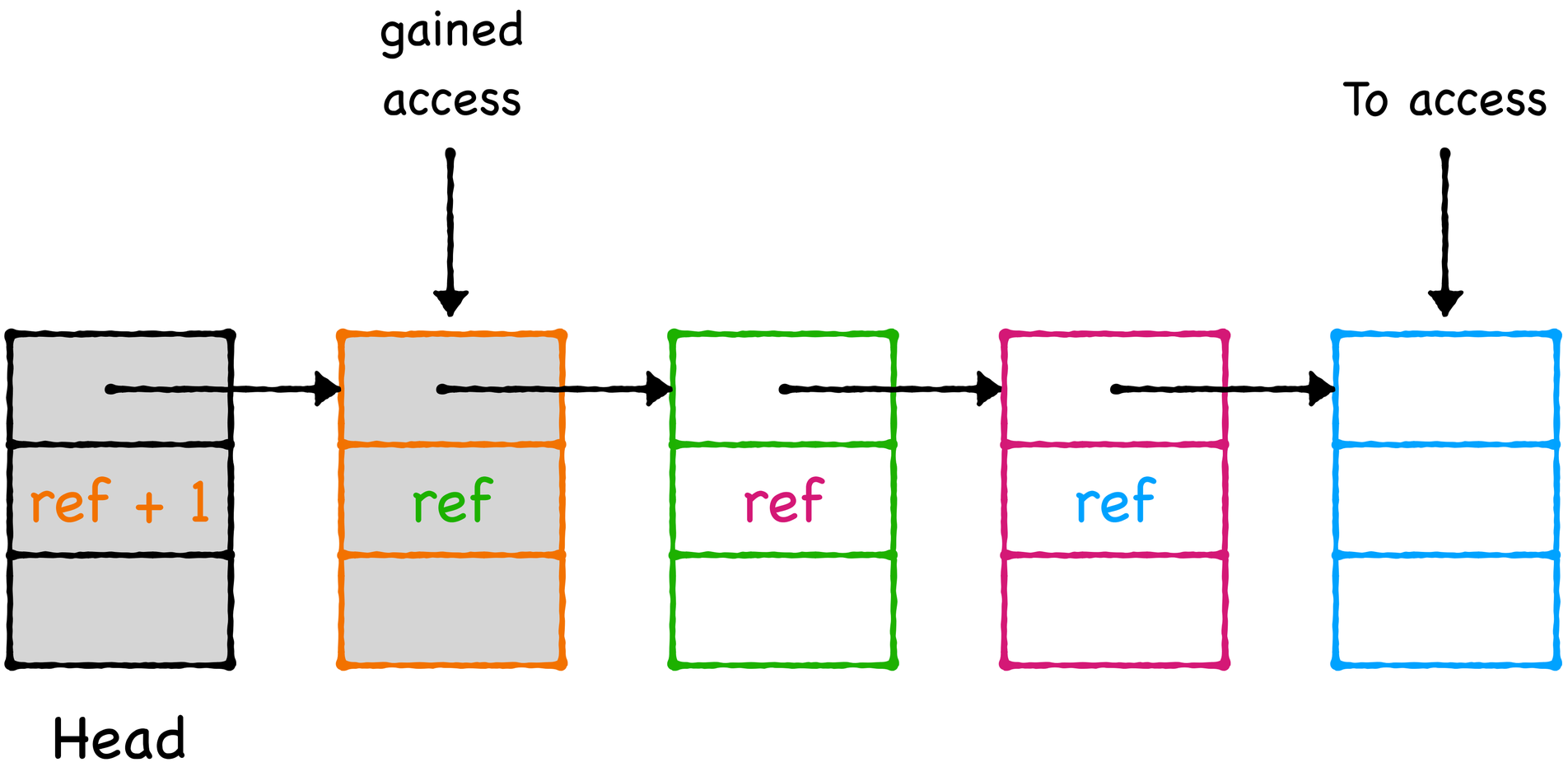

p starts here
To access
ref
ref
ref
ref
Head



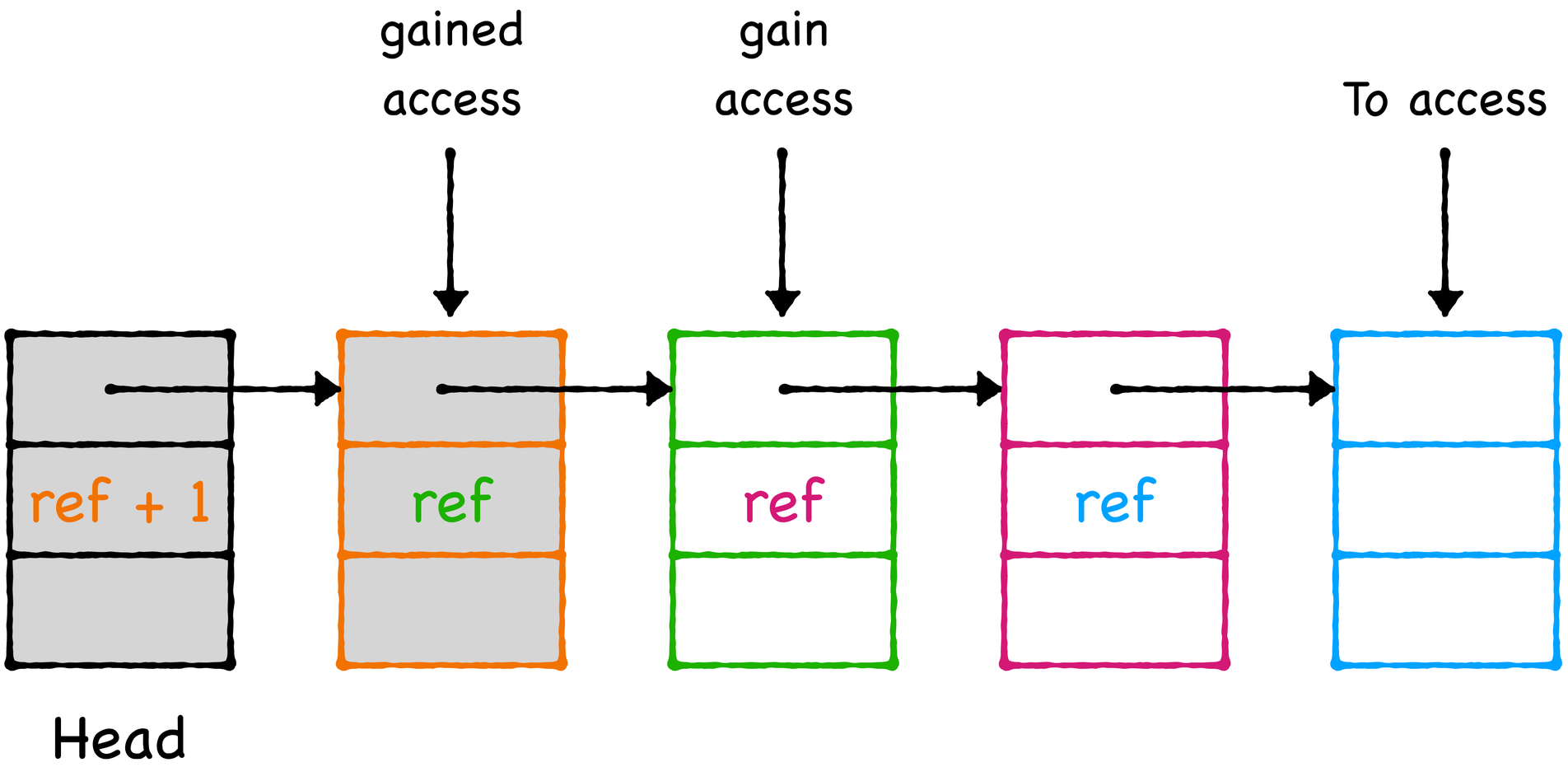
# of acquired cells: 1



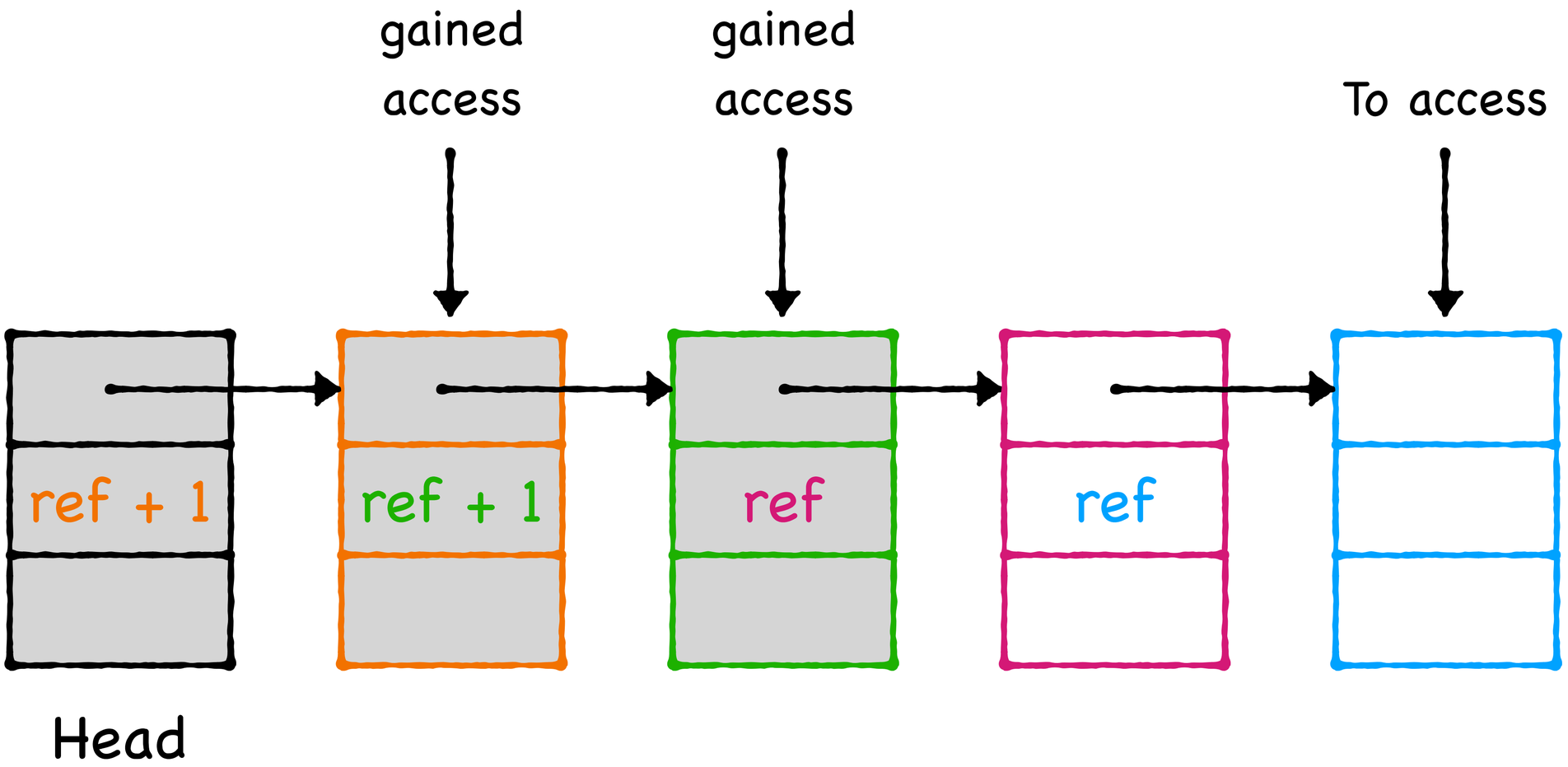
# of acquired cells: 2



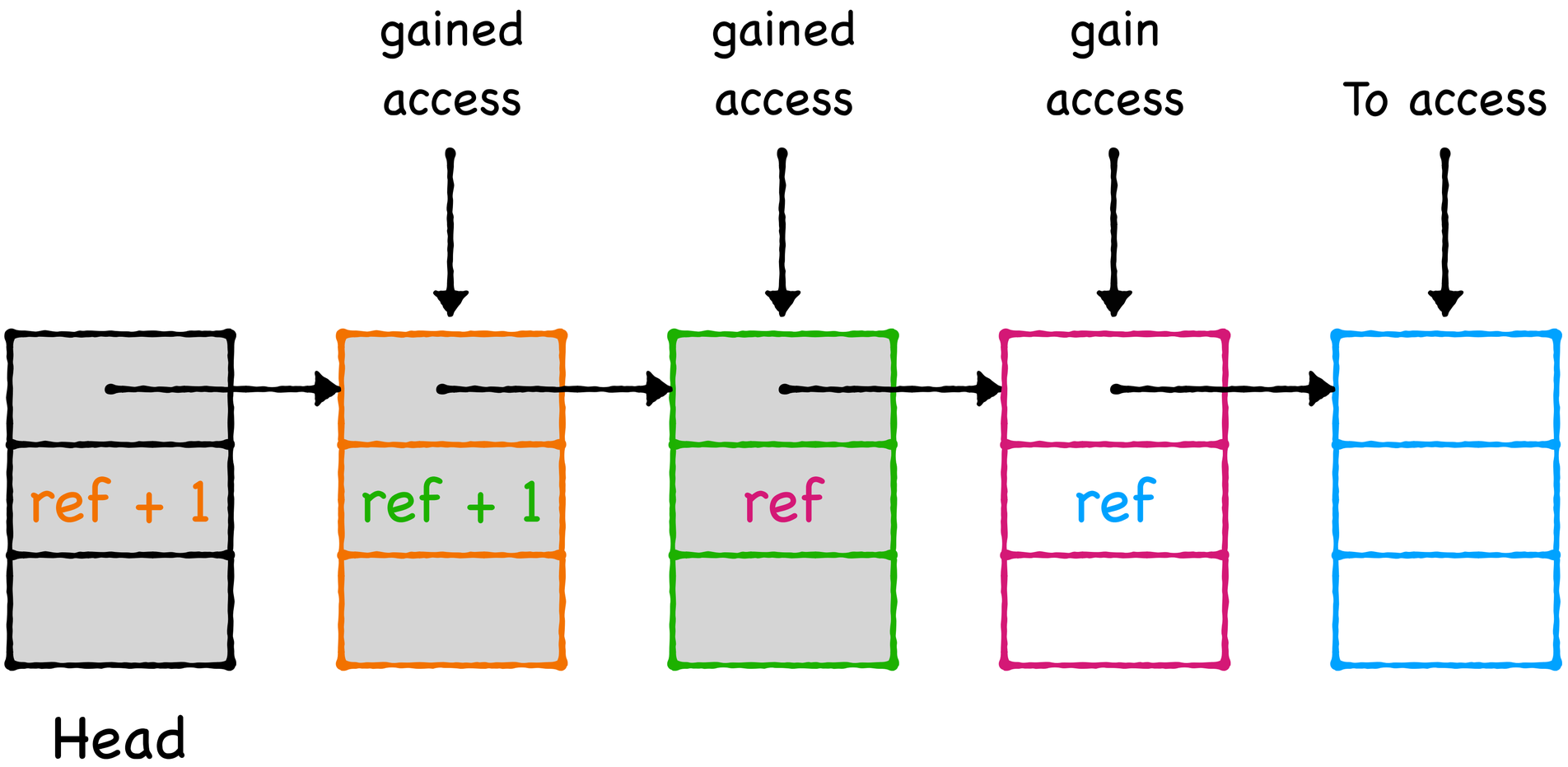
# of acquired cells: 2



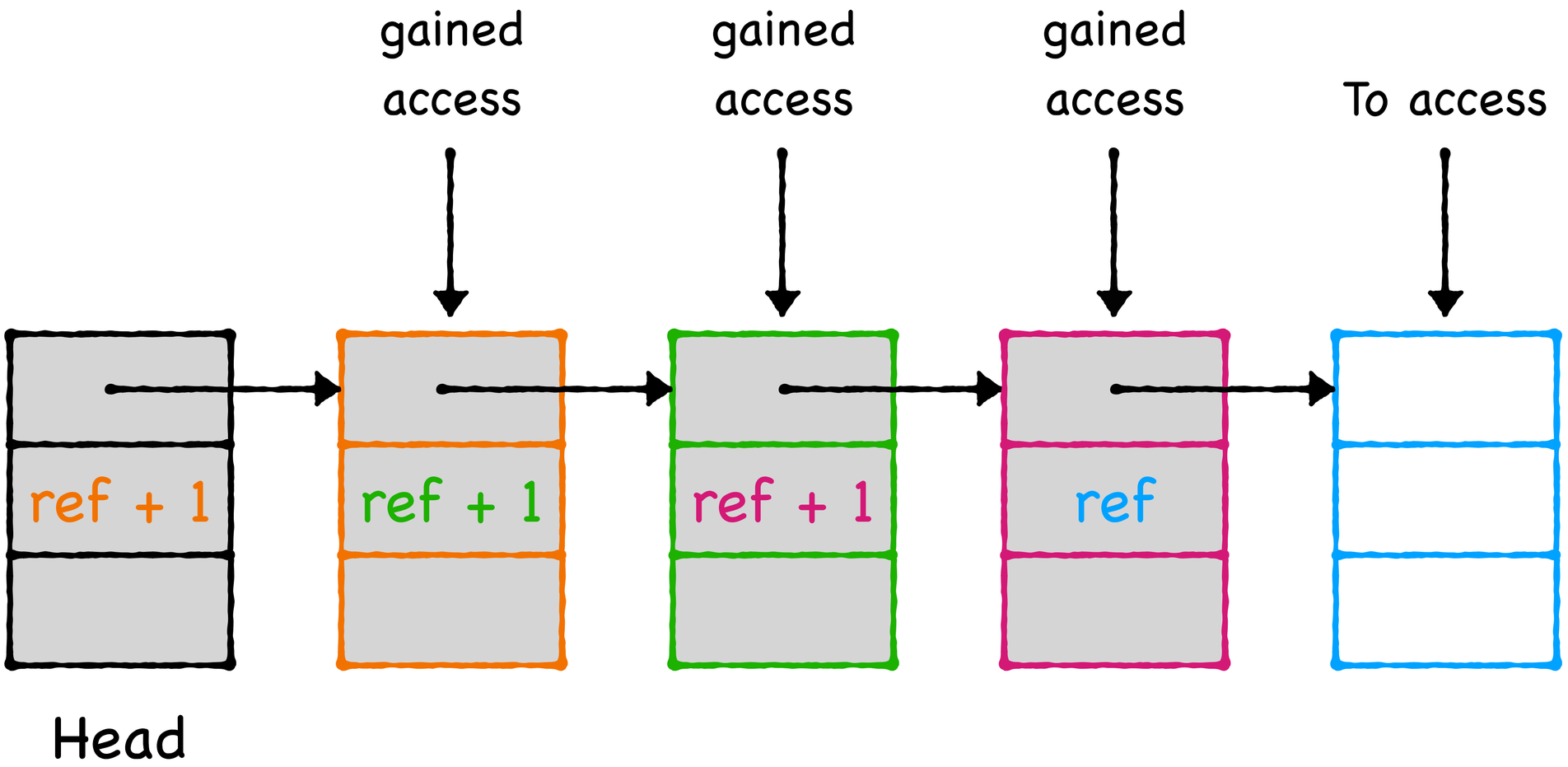
# of acquired cells: 2



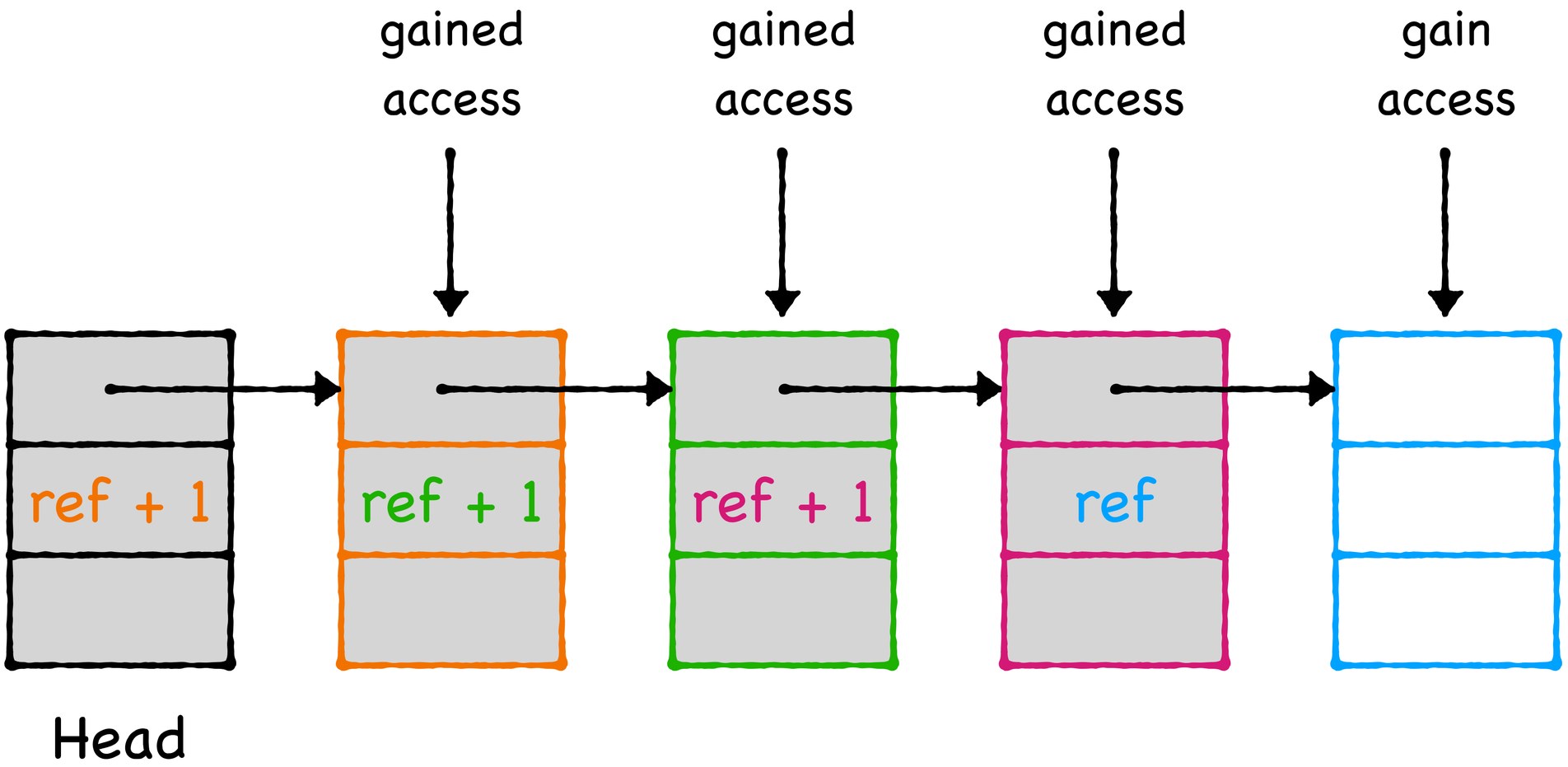
# of acquired cells: 3



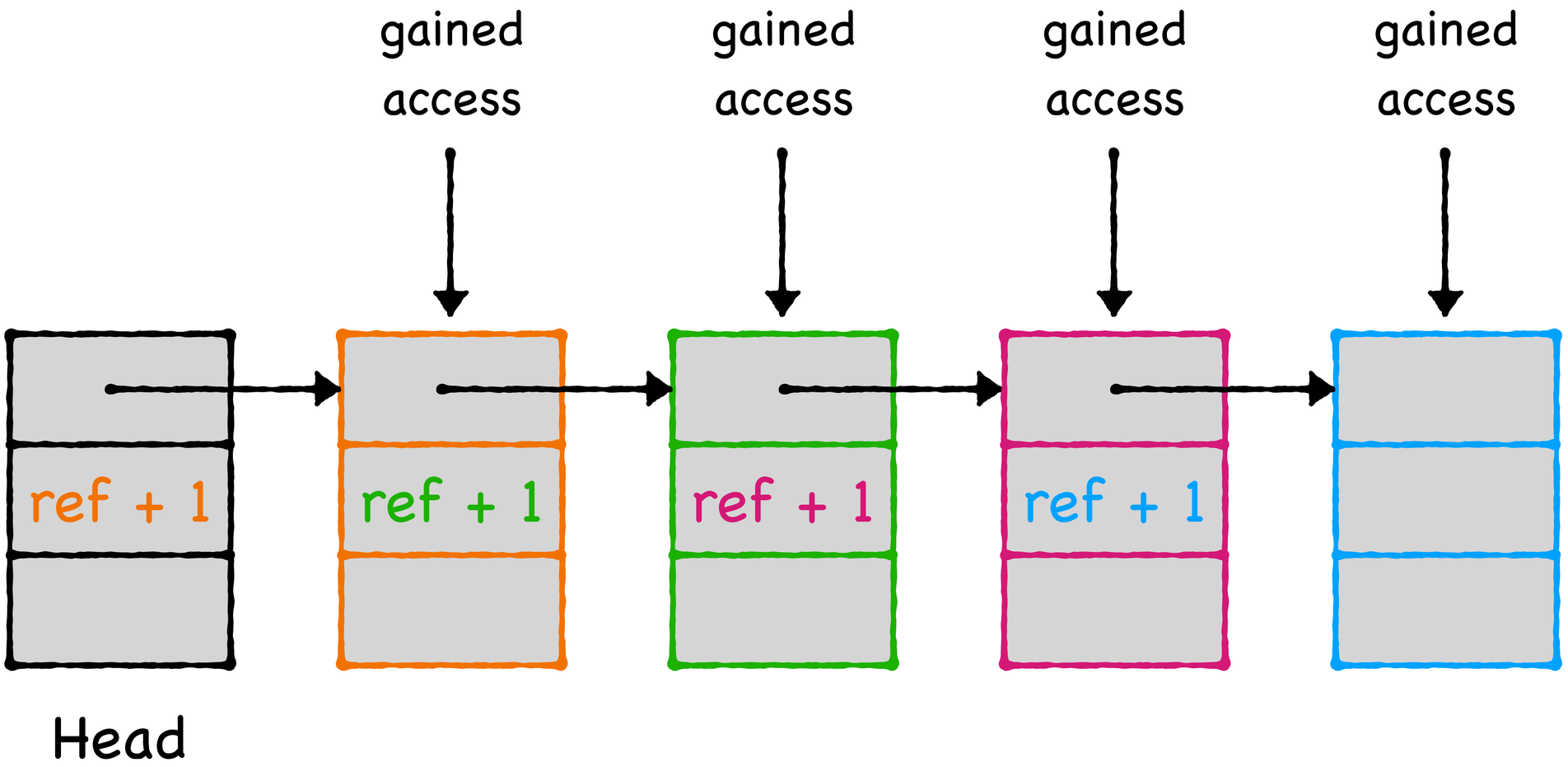
# of acquired cells: 3



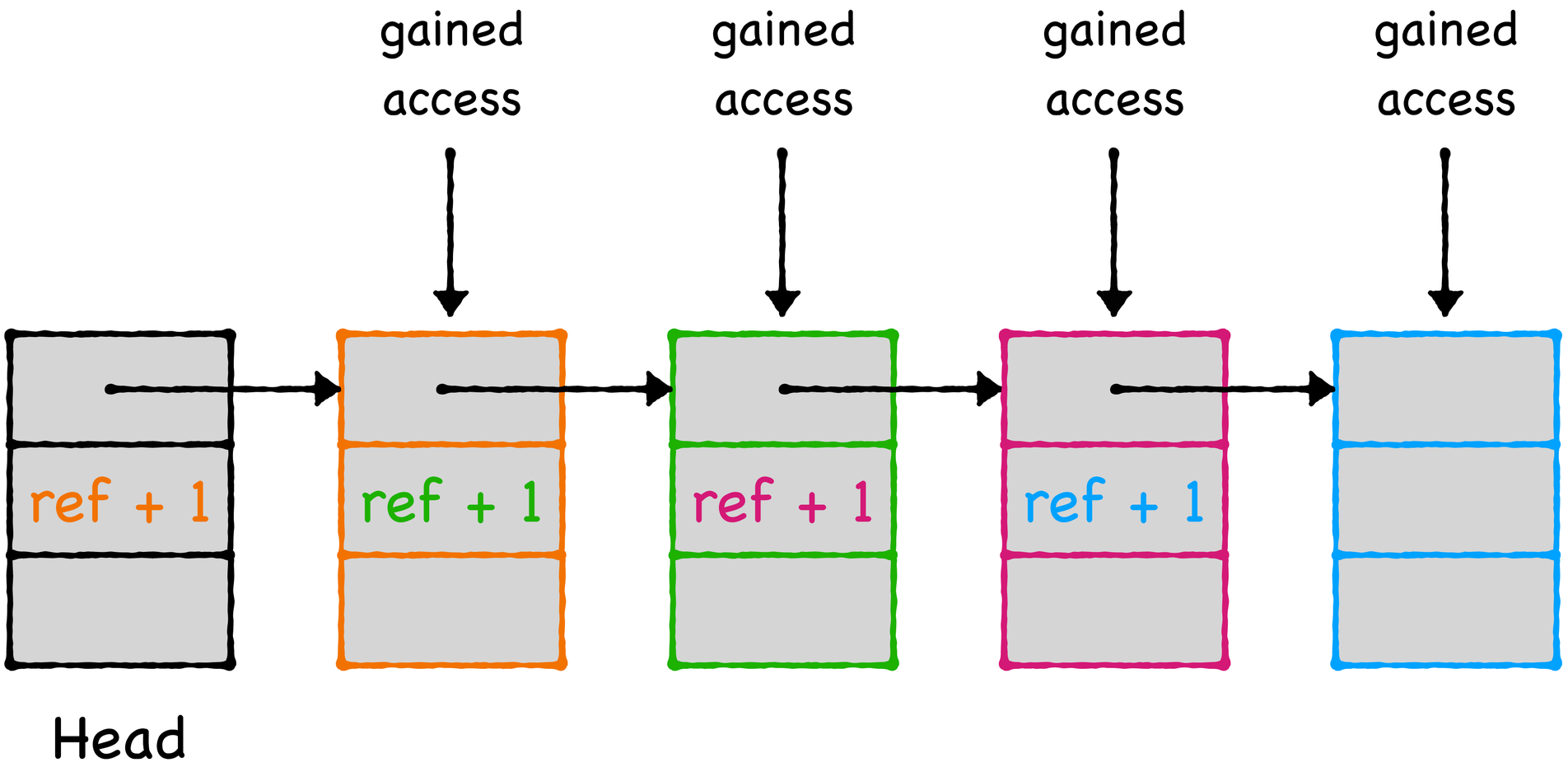
# of acquired cells: 4



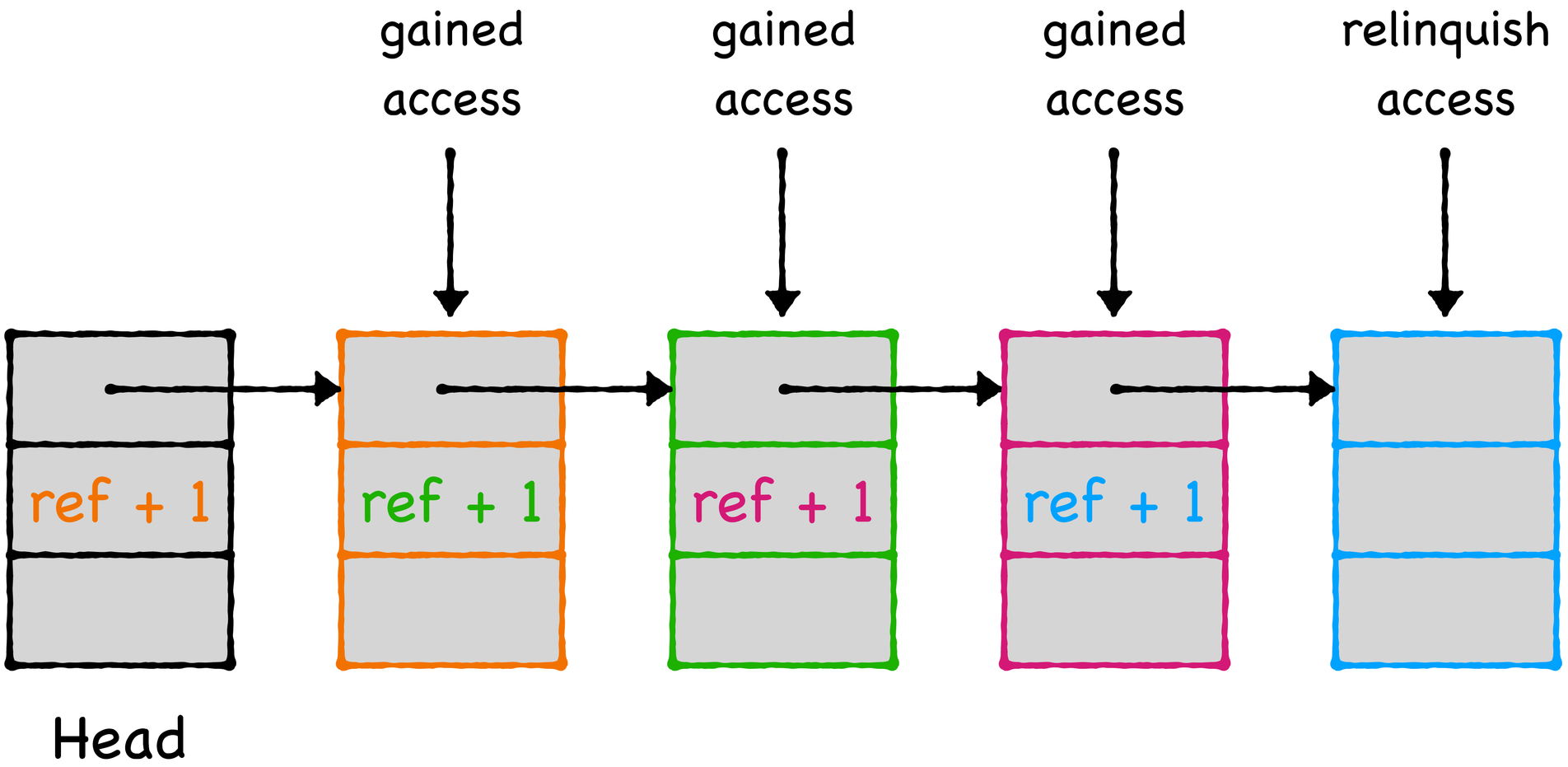
# of acquired cells: 4



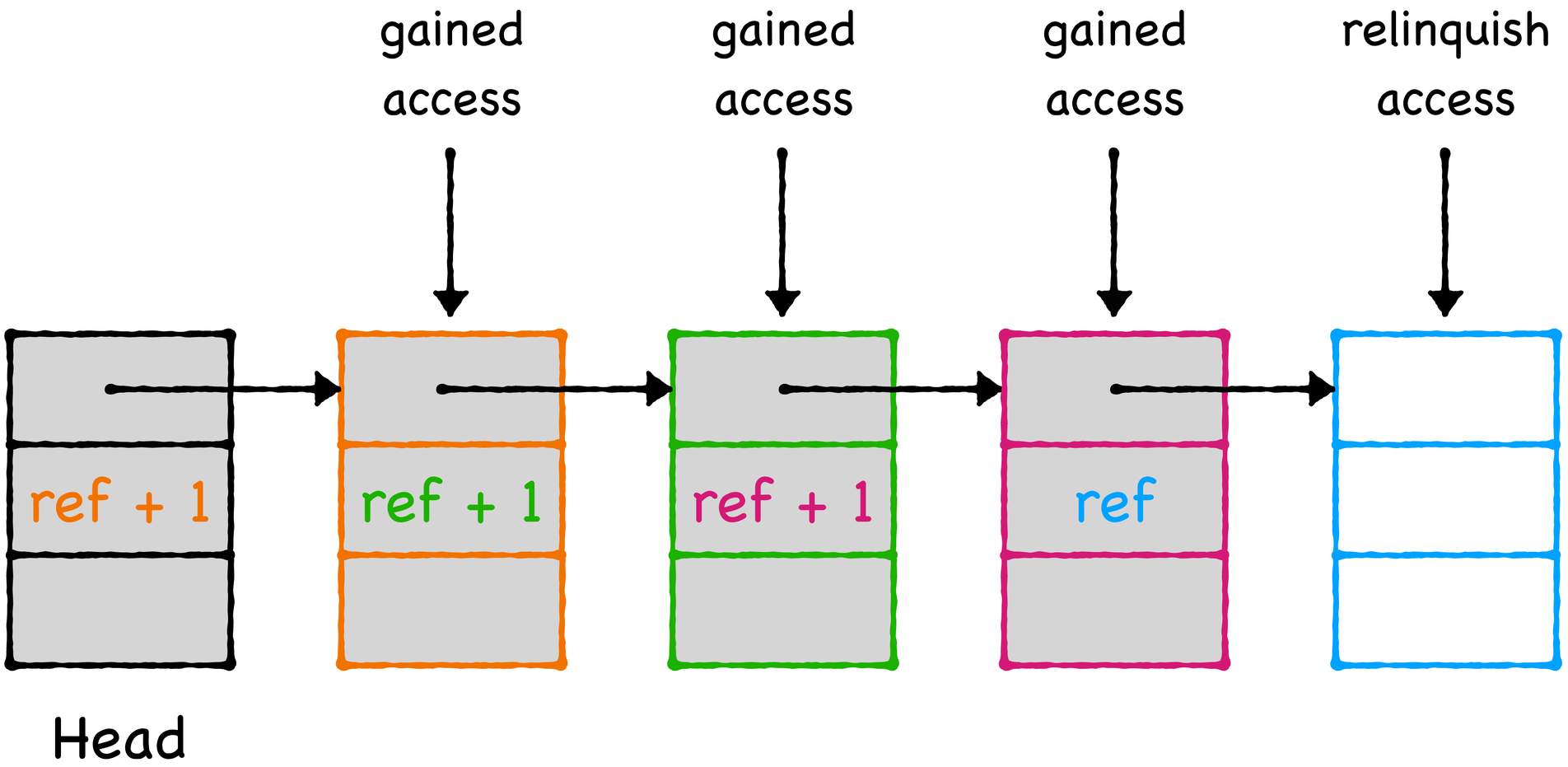
# of acquired cells: 5



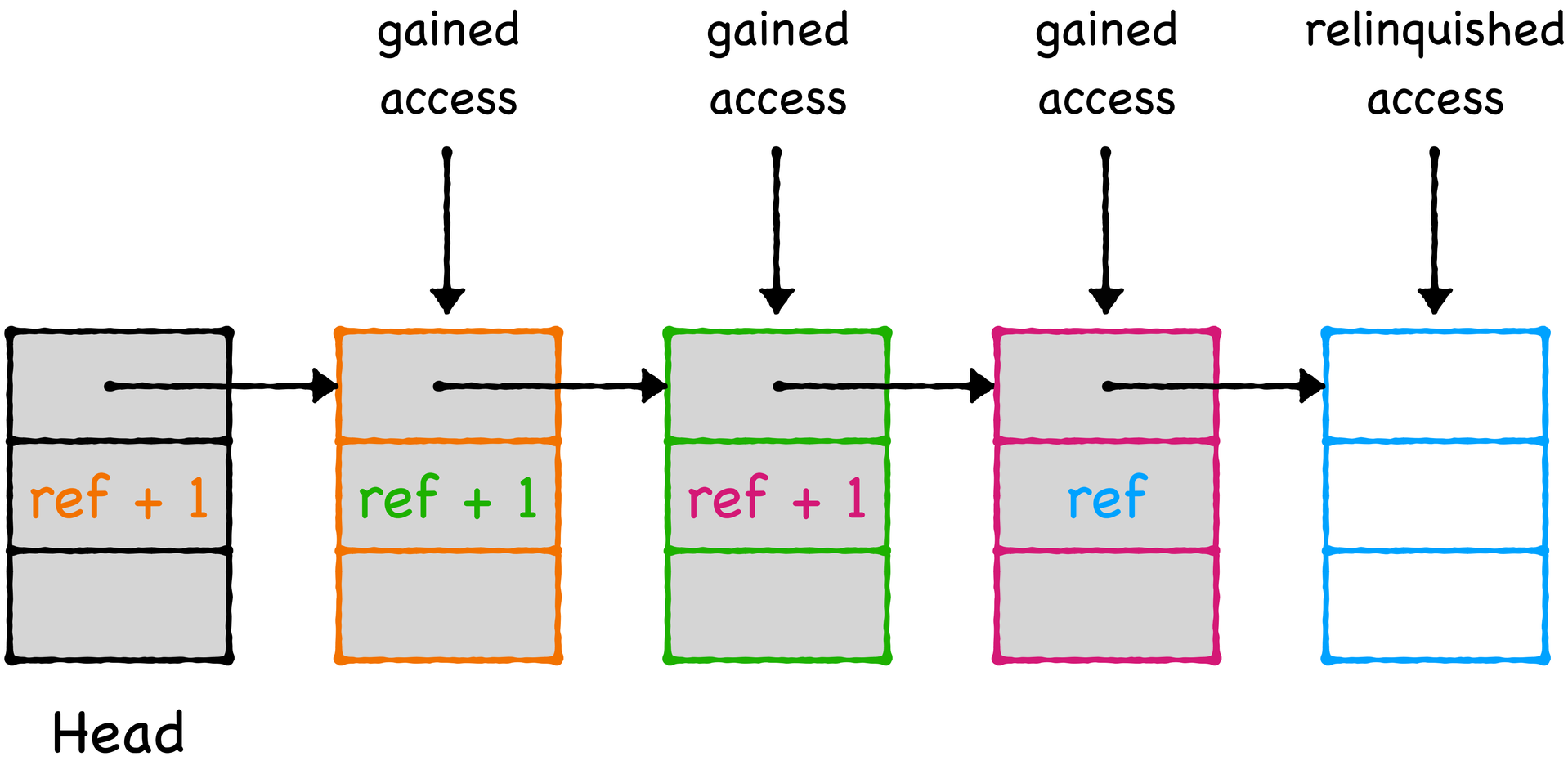
Observe: at this point p has the "right" to access every cell



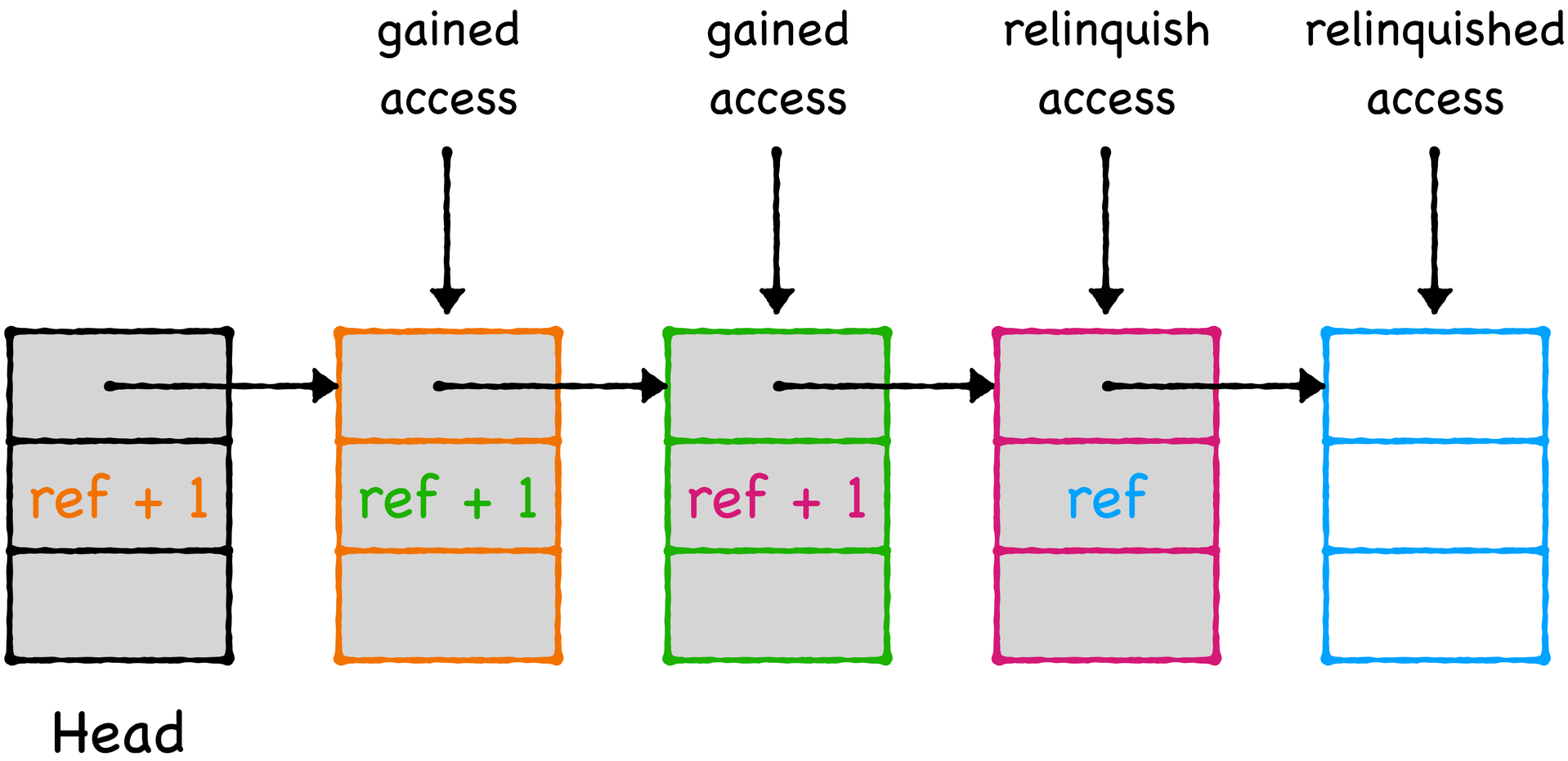
# of acquired cells: 5



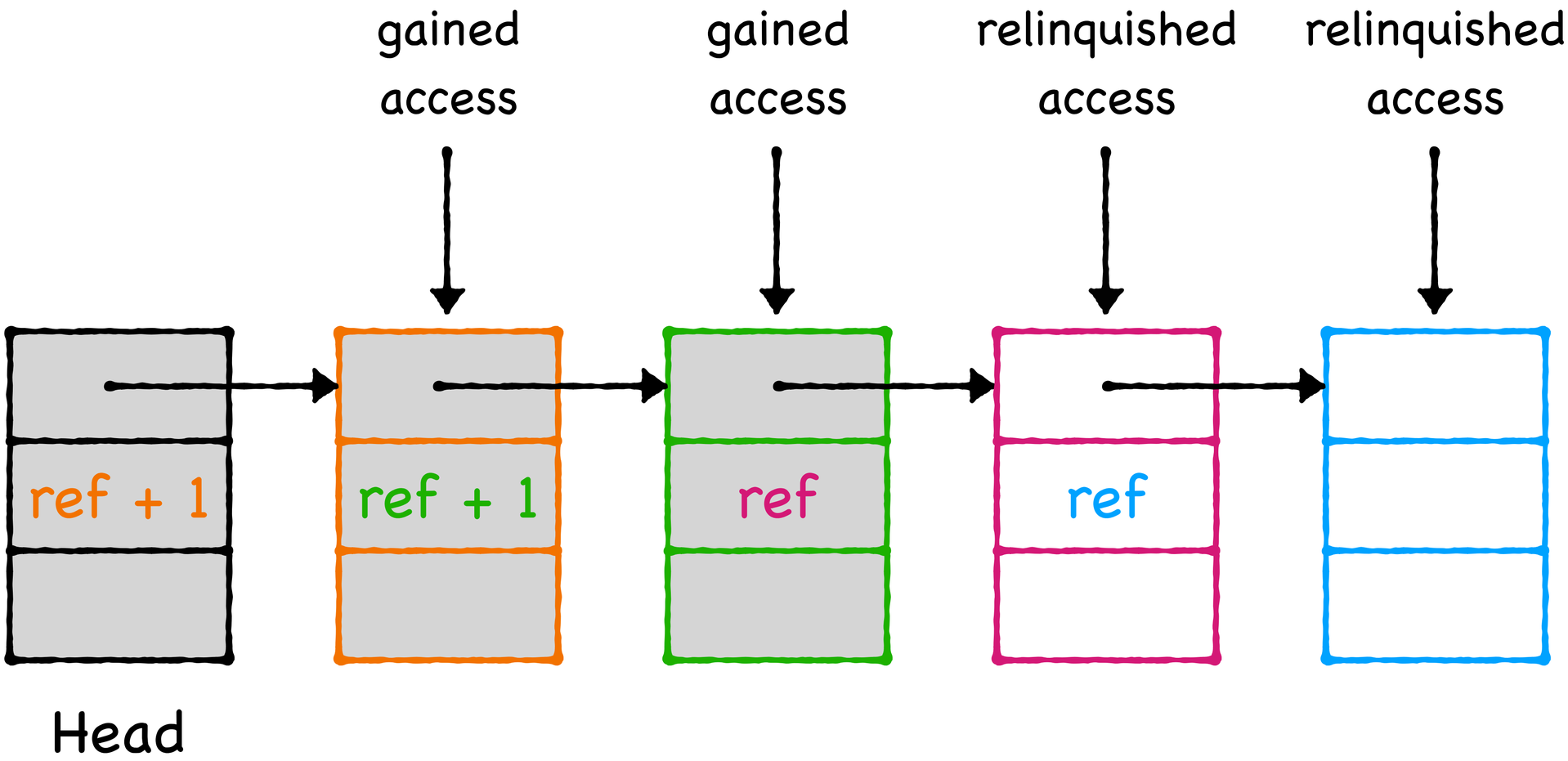
# of acquired cells: 4



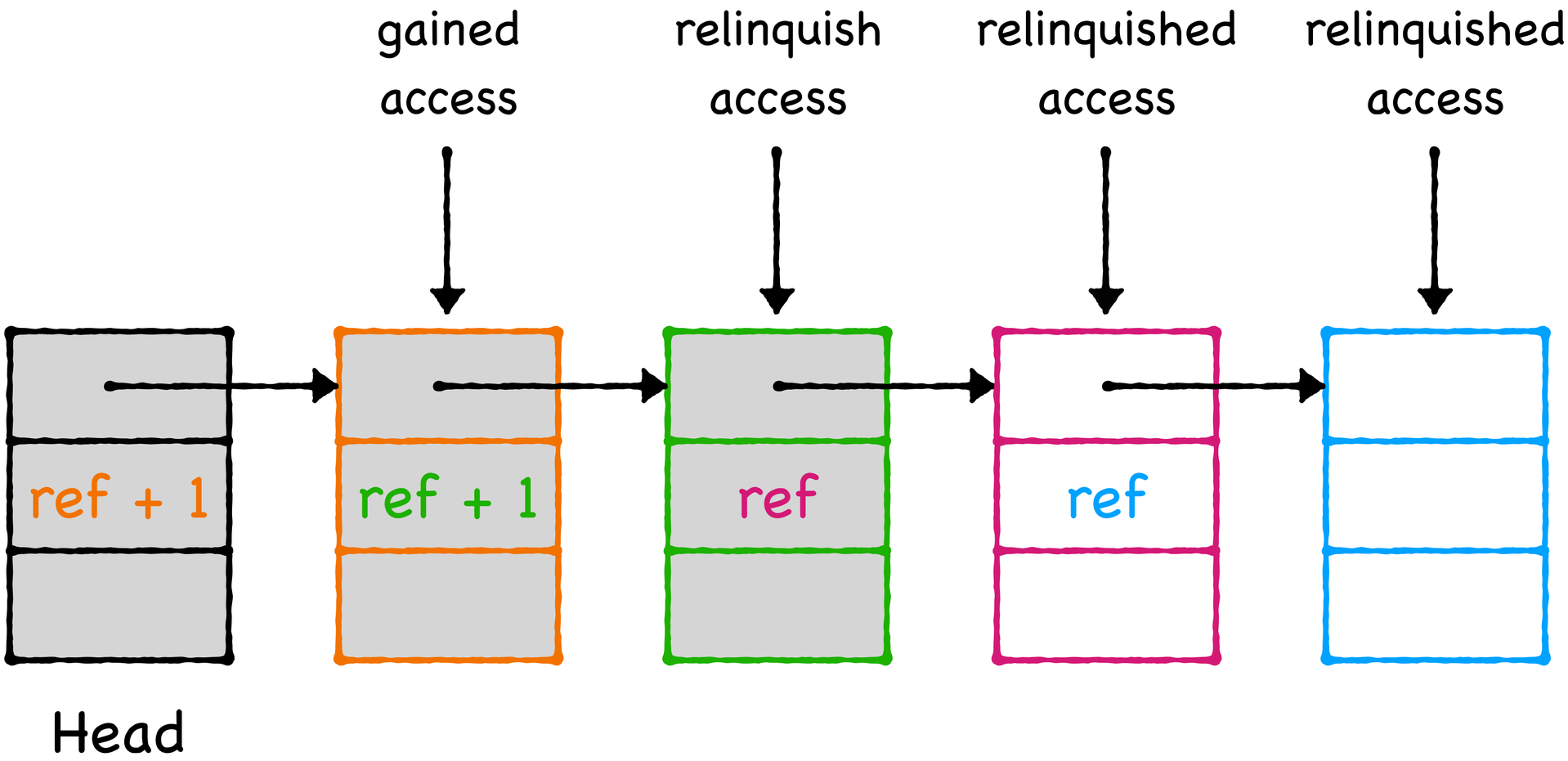
# of acquired cells: 4



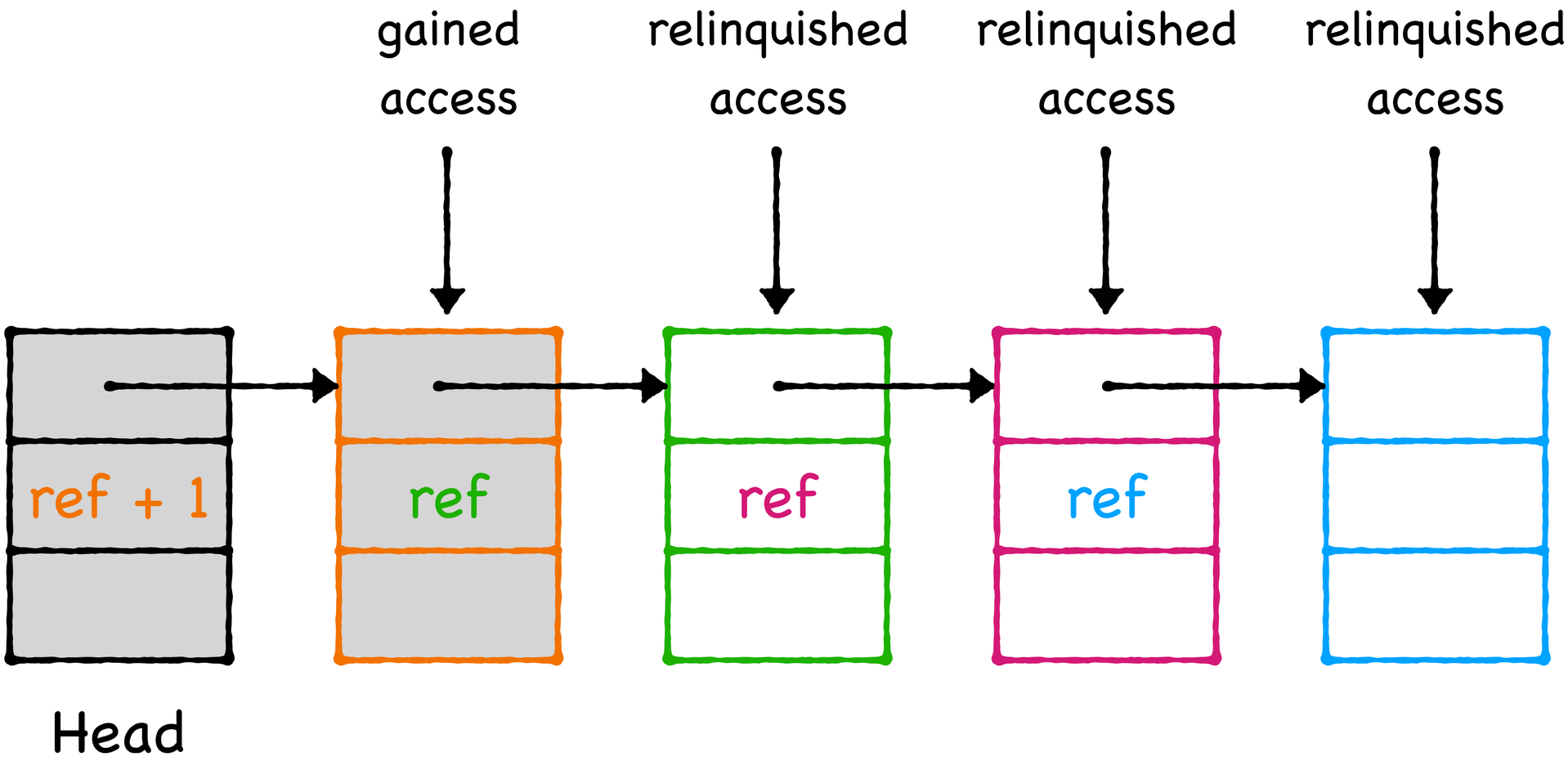
# of acquired cells: 4



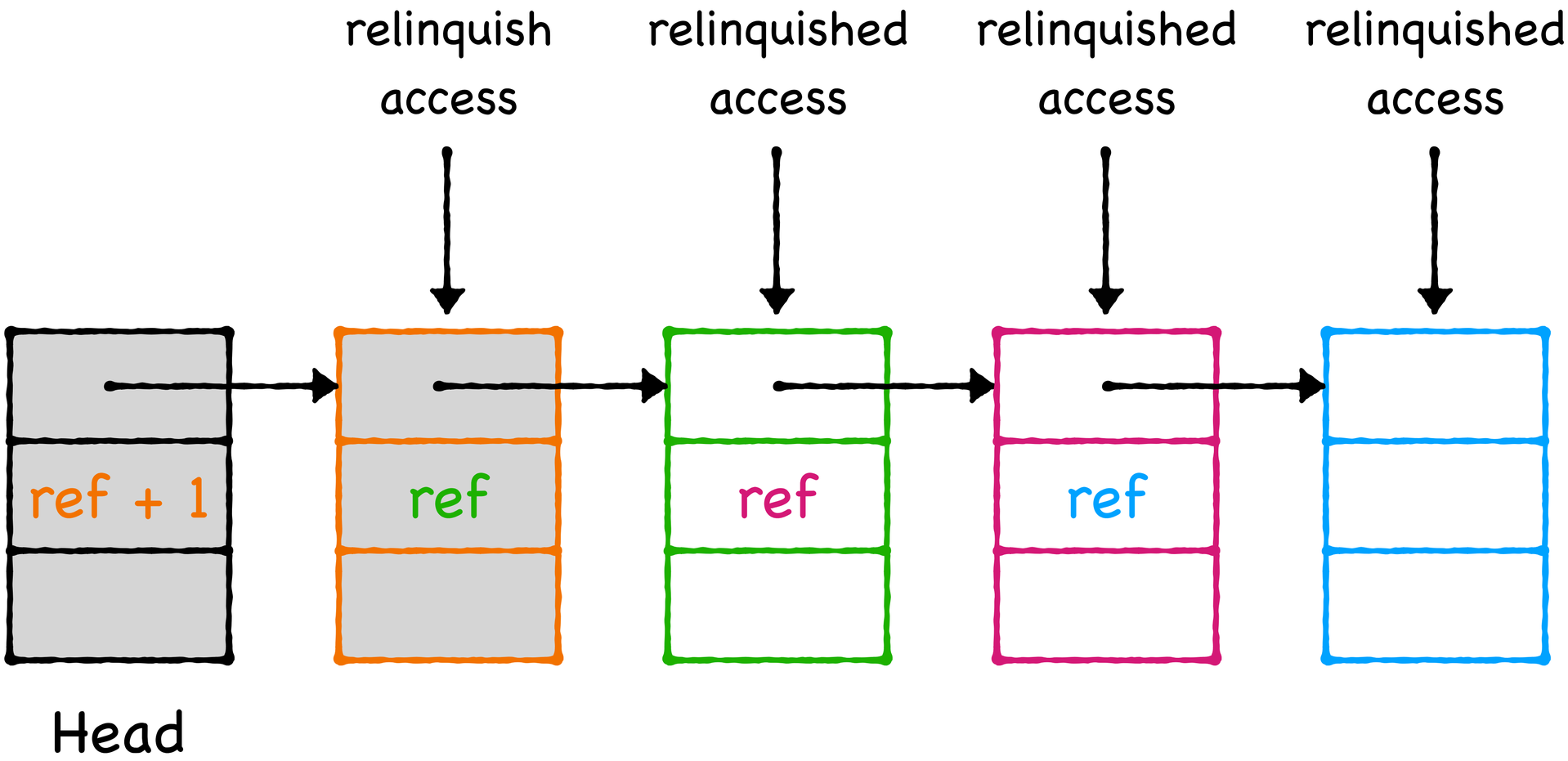
# of acquired cells: 3



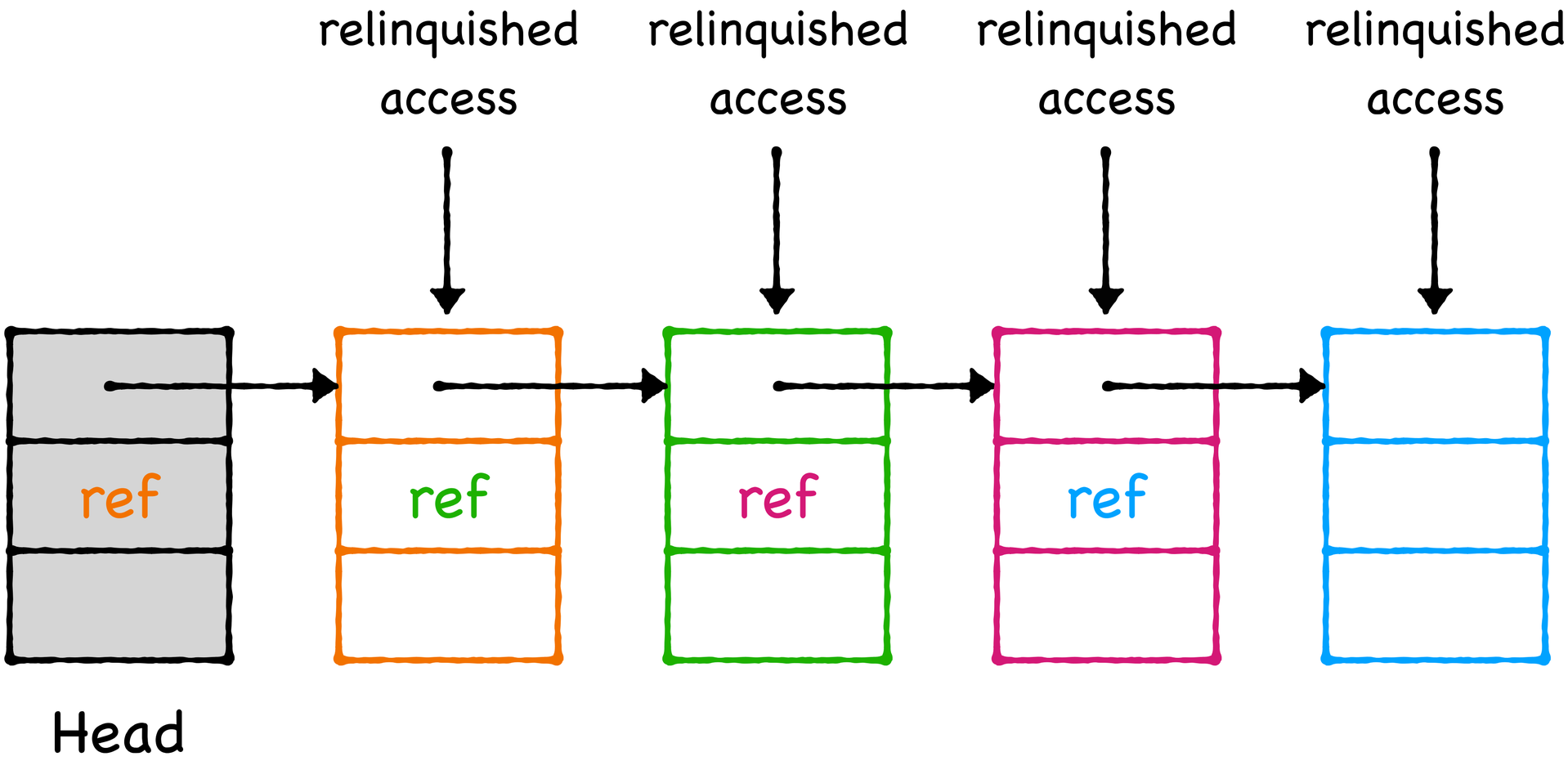
# of acquired cells: 3



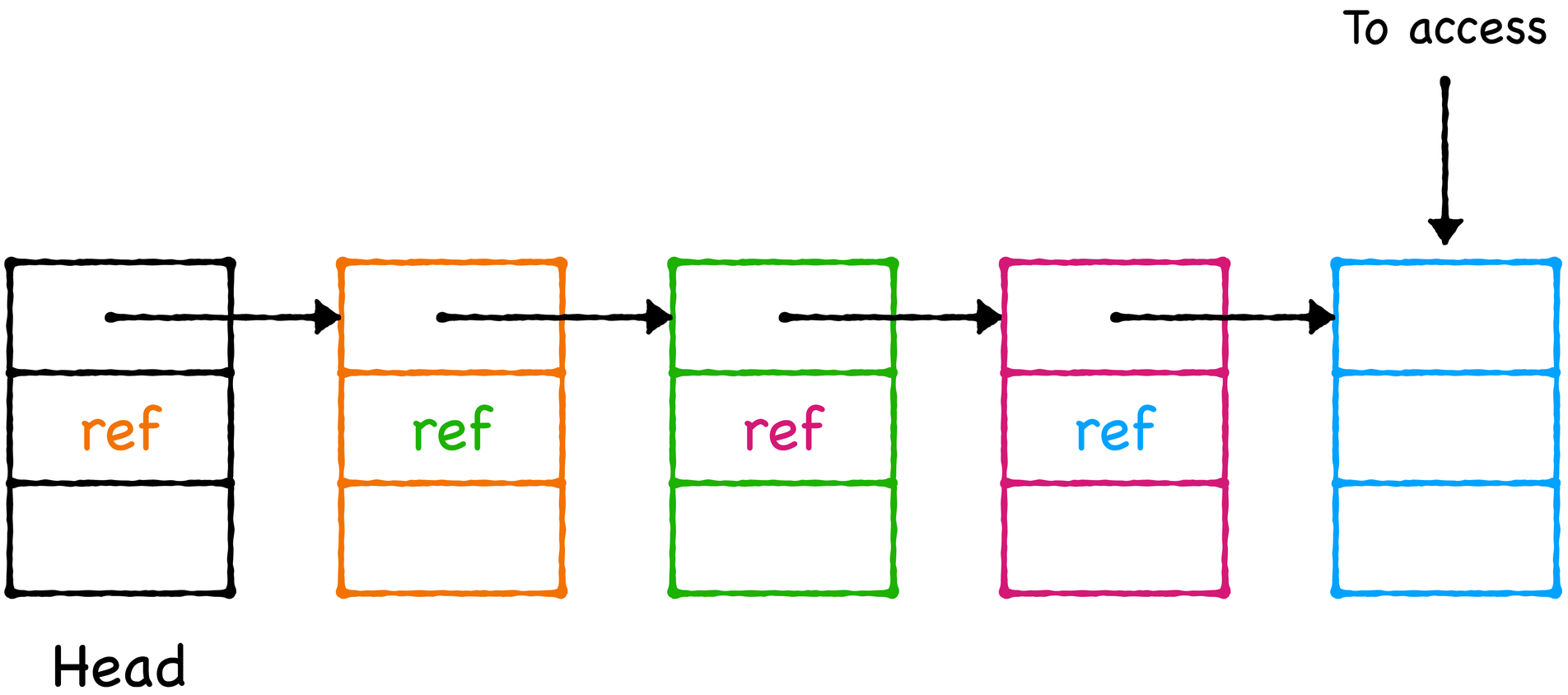
# of acquired cells: 2

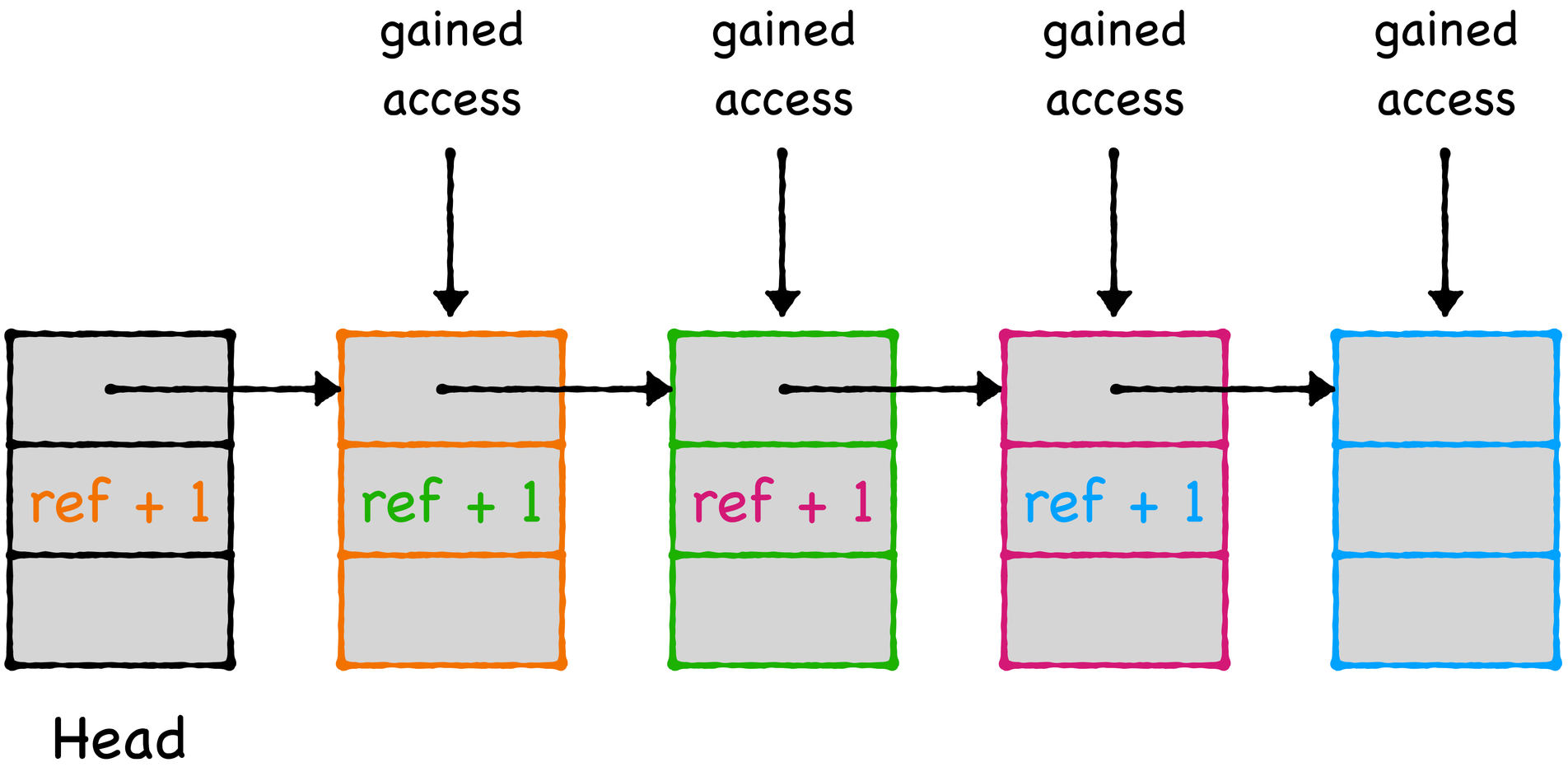


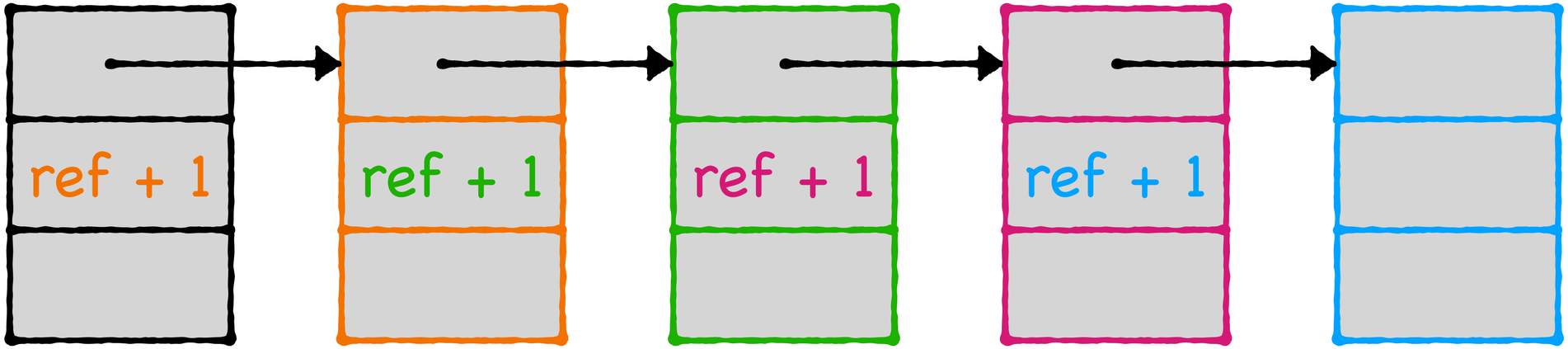
# of acquired cells: 2



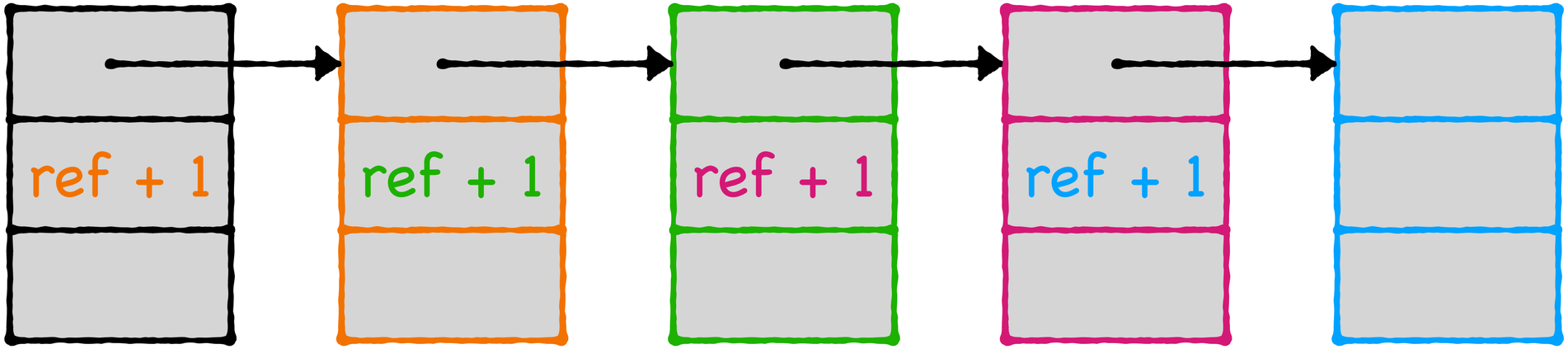
# of acquired cells: 1


To access
ref
ref
ref
ref
Head


gained access
gained access
gained access
gained access
ref + 1
ref + 1
ref + 1
ref + 1
Head

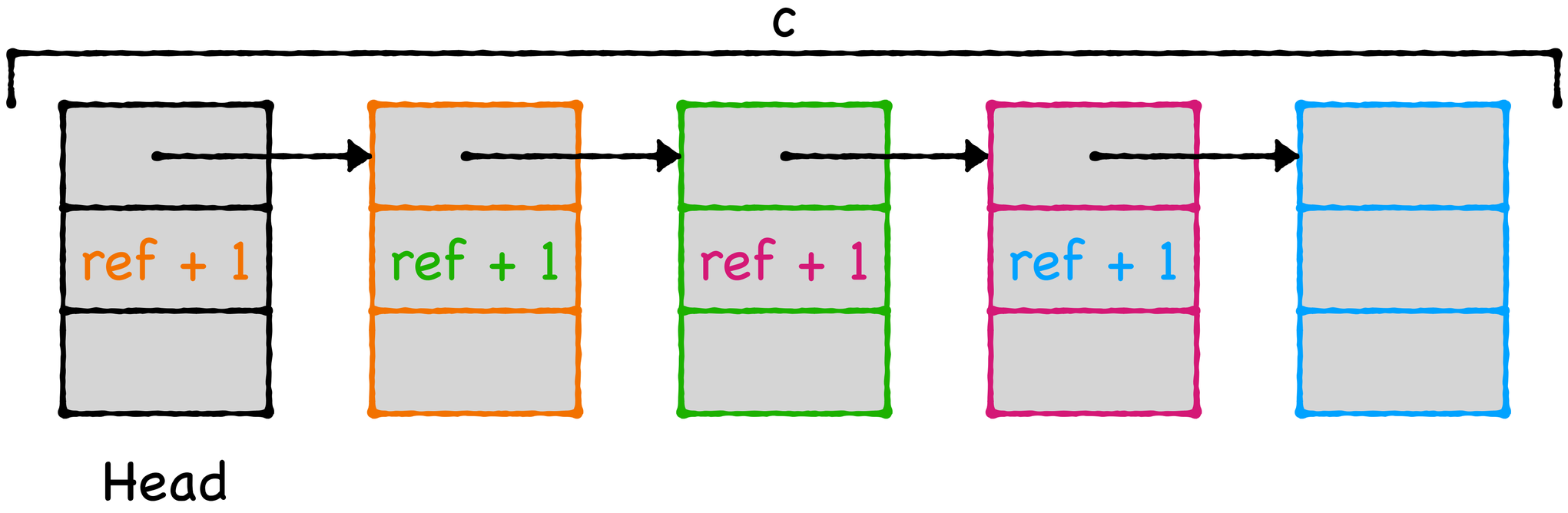


But what if p crashes when it has the **"right" to access every cell?**



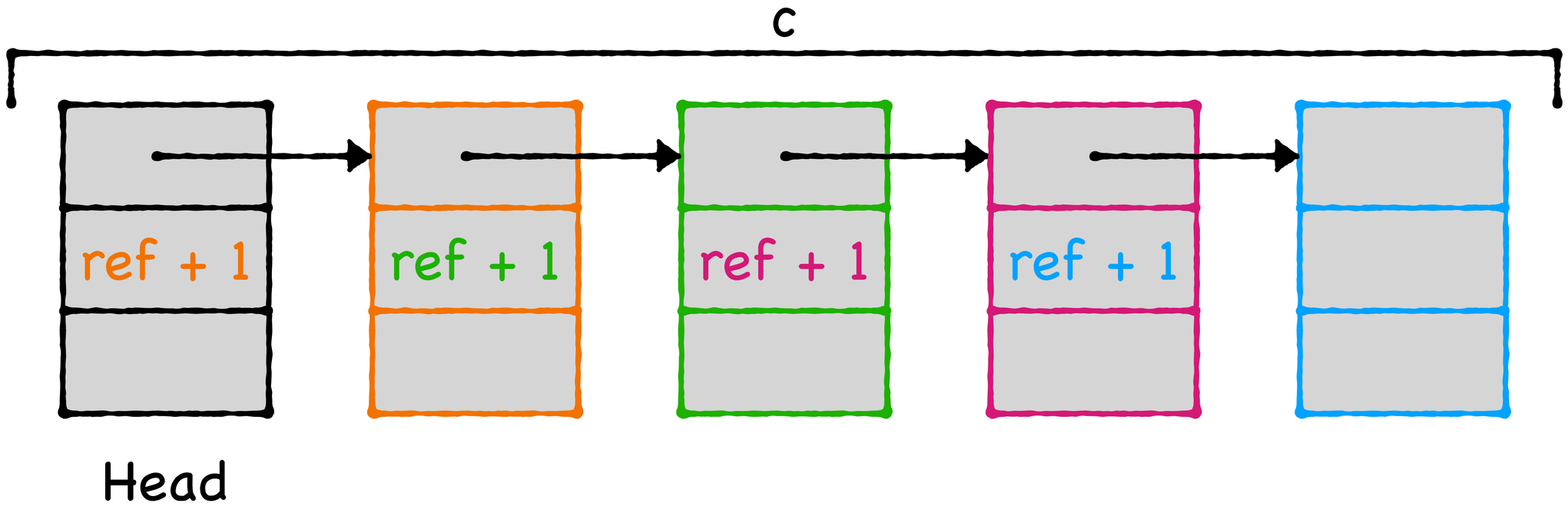
But what if p crashes when it has the **"right" to access every cell?**

None of these cells **can ever be recycled!**



But what if p crashes when it has the **"right" to access every cell?**

None of these cells **can ever be recycled!**



But what if p crashes when it has the **"right" to access every cell?**

None of these cells **can ever be recycled!**

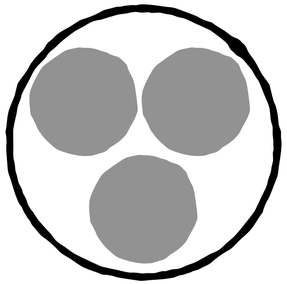
The space complexity is $\Omega(c^2)$ where c is the point contention

Improving the Space Complexity

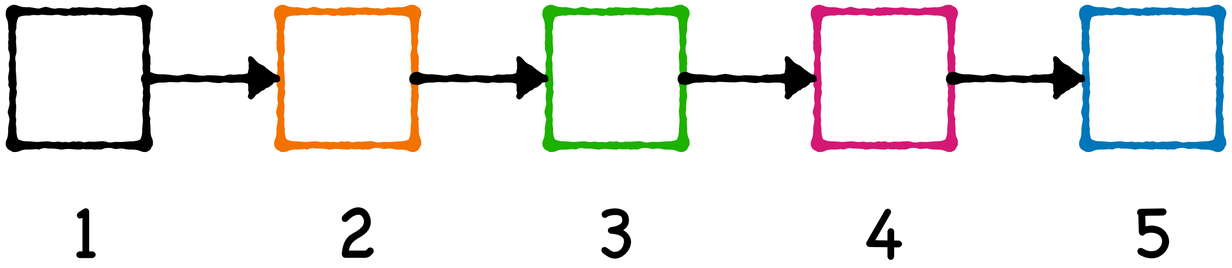
Goal: reduce the number of cells any process has the right to access at any point in time

Yes!

The Reversible Pebble Game [Bennett'89]



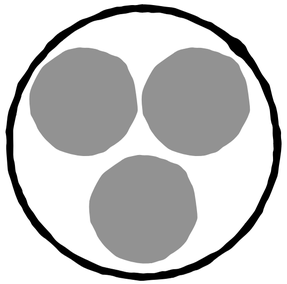
Pebbles



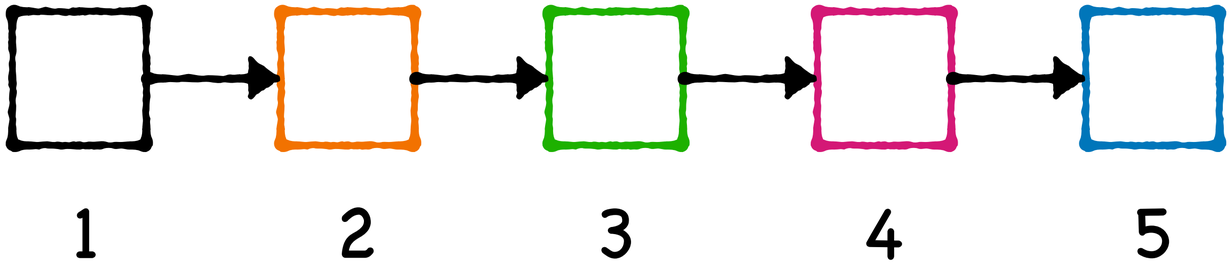
Given:

(1) A list of n nodes (here n = 5)

(2) A set of m pebbles (here m = 3)



Pebbles

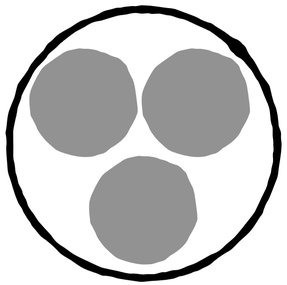


Given:

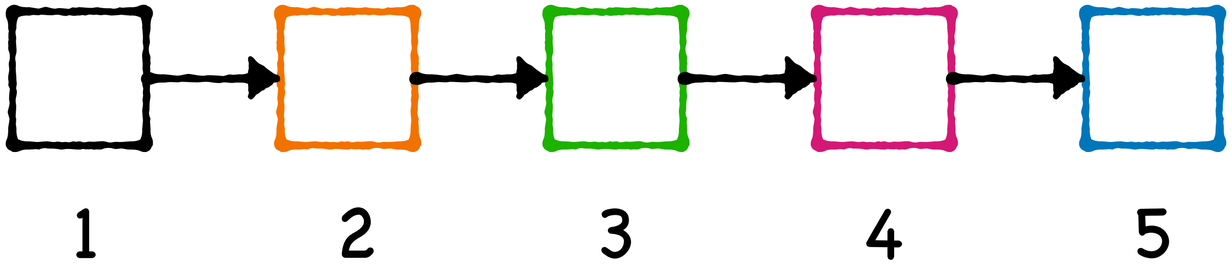
(1) A list of n nodes (here n = 5)

(2) A set of m pebbles (here m = 3)

Goal: (a) pebble the last node and then (b) remove all pebbles



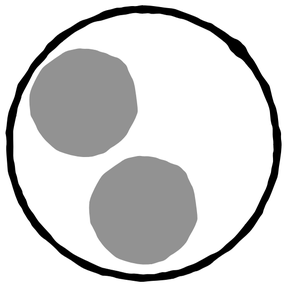
Pebbles



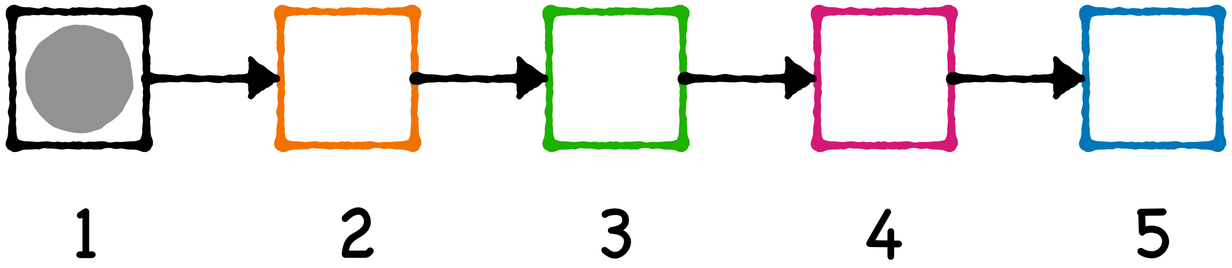
Rules:

(1) You can place or remove a pebble on 1

Goal: (a) pebble the last node and then (b) remove all pebbles



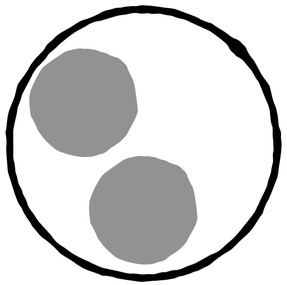
Pebbles



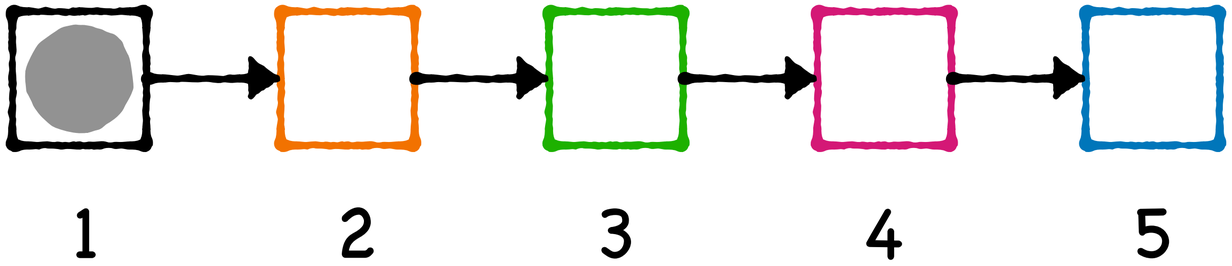
Rules:

(1) You can place or remove a pebble on 1

Goal: (a) pebble the last node and then (b) remove all pebbles



Pebbles

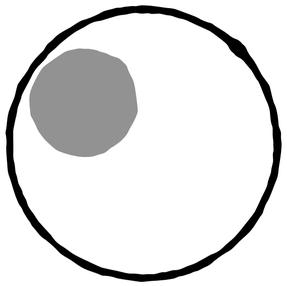


Rules:

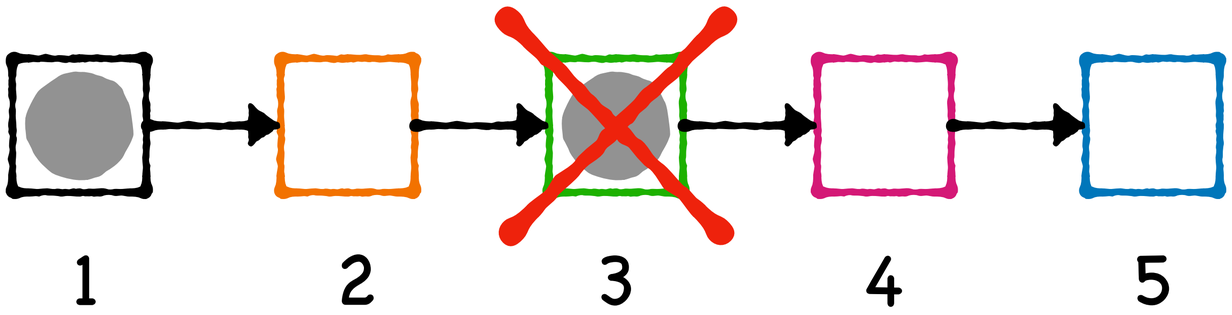
(1) You can place or remove a pebble on 1

(2) You can place or remove a pebble on $i > 1$ if $i - 1$ is pebbled

Goal: (a) pebble the last node and then (b) remove all pebbles



Pebbles

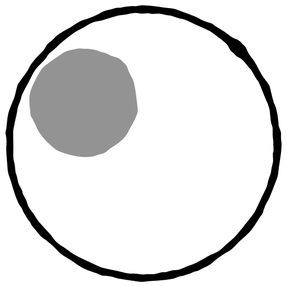


Rules:

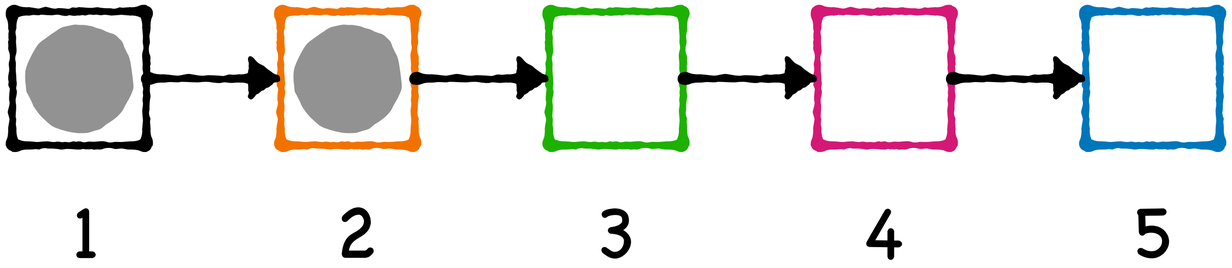
(1) You can place or remove a pebble on 1

(2) You can place or remove a pebble on $i > 1$ if $i - 1$ is pebbled

Goal: (a) pebble the last node and then (b) remove all pebbles



Pebbles

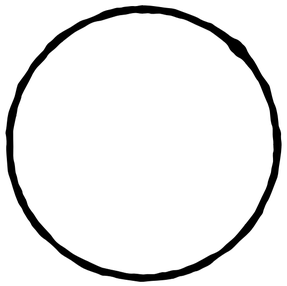


Rules:

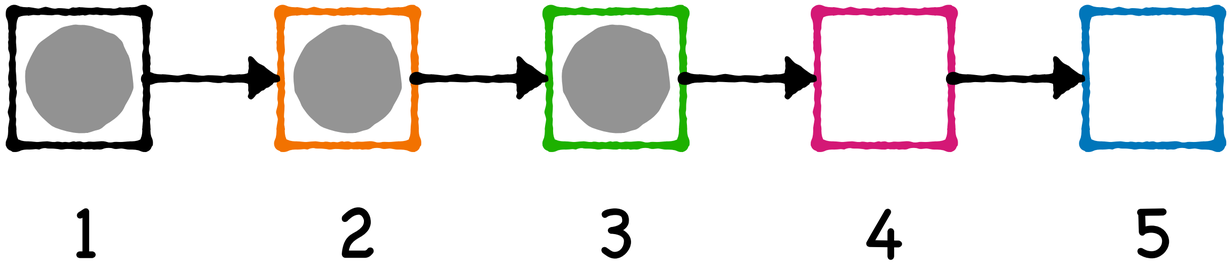
(1) You can place or remove a pebble on 1

(2) You can place or remove a pebble on $i > 1$ if $i - 1$ is pebbled

Goal: (a) pebble the last node and then (b) remove all pebbles



Pebbles

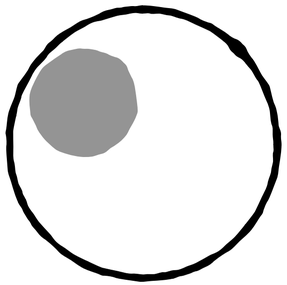


Rules:

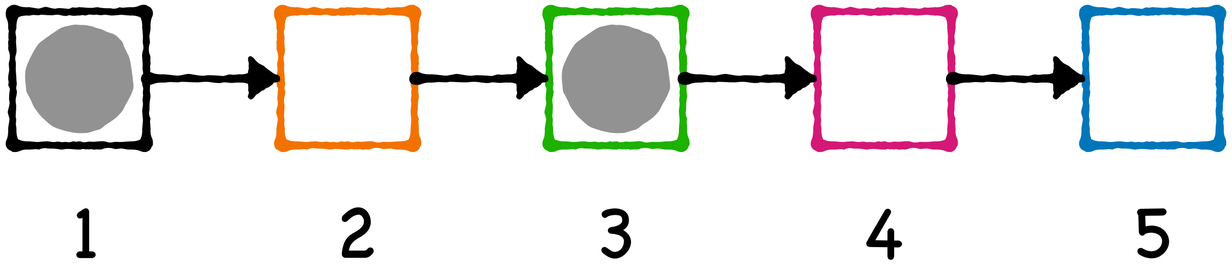
(1) You can place or remove a pebble on 1

(2) You can place or remove a pebble on $i > 1$ if $i - 1$ is pebbled

Goal: (a) pebble the last node and then (b) remove all pebbles



Pebbles

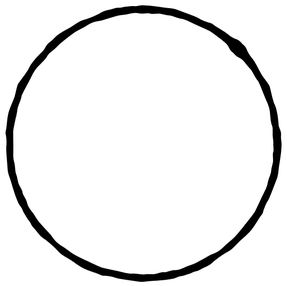


Rules:

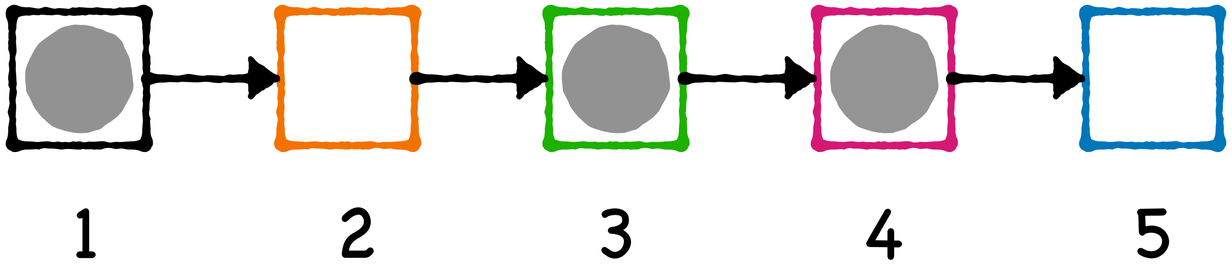
(1) You can place or remove a pebble on 1

(2) You can place or remove a pebble on $i > 1$ if $i - 1$ is pebbled

Goal: (a) pebble the last node and then (b) remove all pebbles



Pebbles

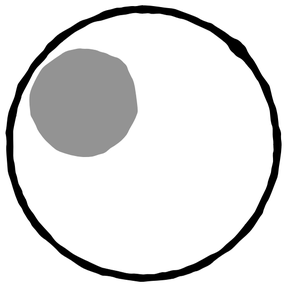


Rules:

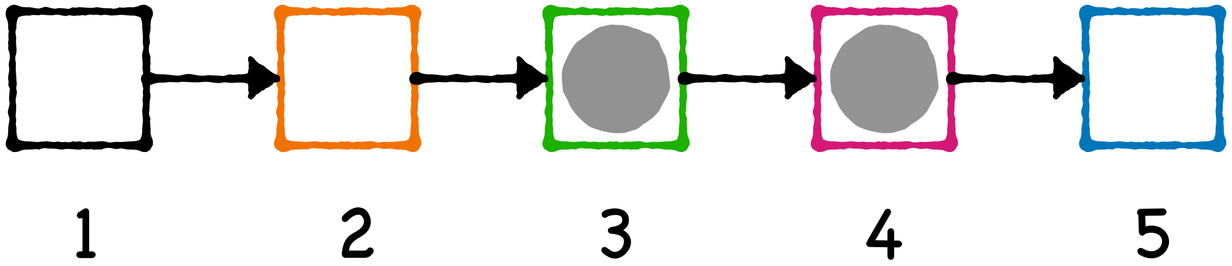
(1) You can place or remove a pebble on 1

(2) You can place or remove a pebble on $i > 1$ if $i - 1$ is pebbled

Goal: (a) pebble the last node and then (b) remove all pebbles



Pebbles

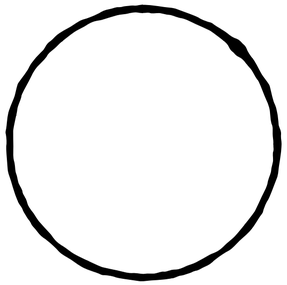


Rules:

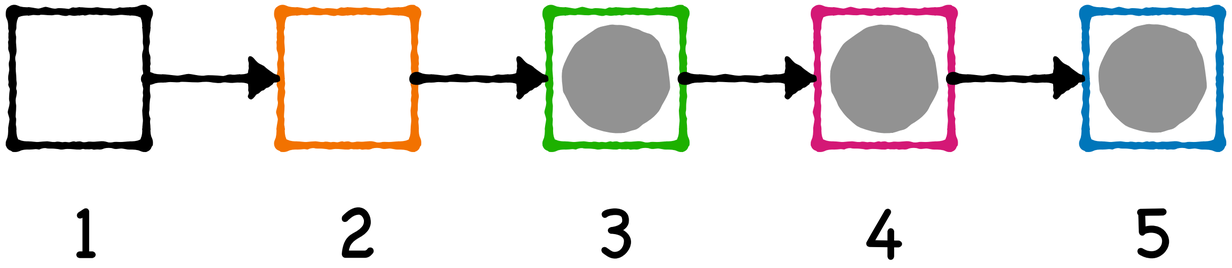
(1) You can place or remove a pebble on 1

(2) You can place or remove a pebble on $i > 1$ if $i - 1$ is pebbled

Goal: (a) pebble the last node and then (b) remove all pebbles



Pebbles

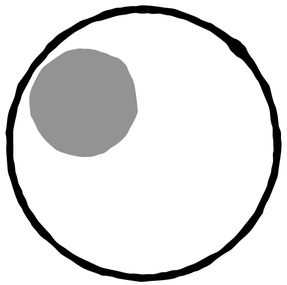


Rules:

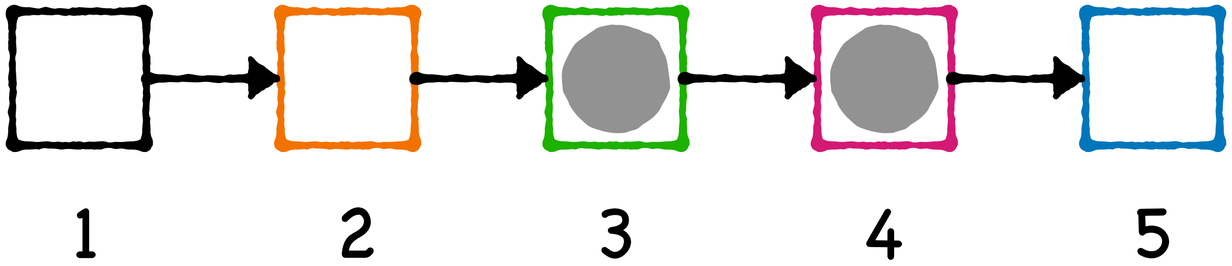
(1) You can place or remove a pebble on 1

(2) You can place or remove a pebble on $i > 1$ if $i - 1$ is pebbled

Goal: (a) pebble the last node and then (b) remove all pebbles



Pebbles

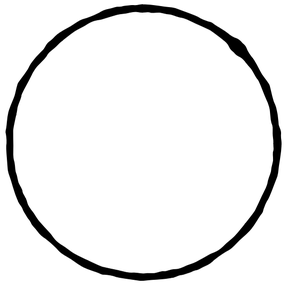


Rules:

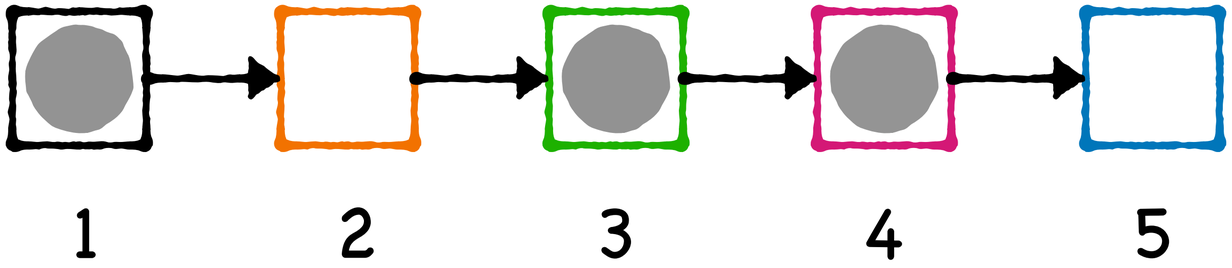
(1) You can place or remove a pebble on 1

(2) You can place or remove a pebble on $i > 1$ if $i - 1$ is pebbled

Goal: (a) pebble the last node and then (b) remove all pebbles



Pebbles

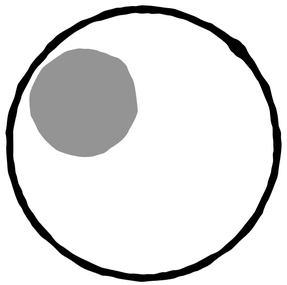


Rules:

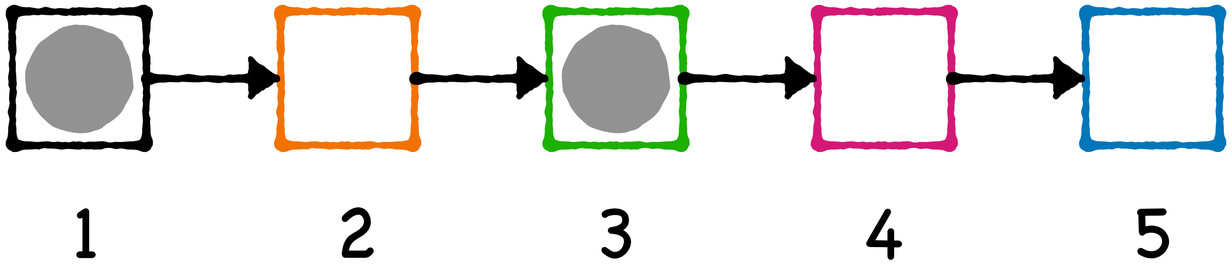
(1) You can place or remove a pebble on 1

(2) You can place or remove a pebble on $i > 1$ if $i - 1$ is pebbled

Goal: (a) pebble the last node and then (b) remove all pebbles



Pebbles

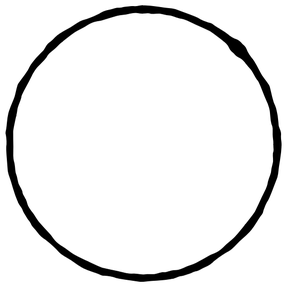


Rules:

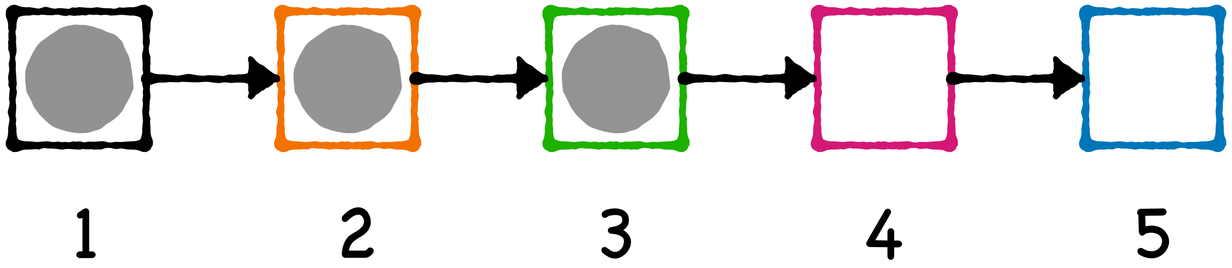
(1) You can place or remove a pebble on 1

(2) You can place or remove a pebble on $i > 1$ if $i - 1$ is pebbled

Goal: (a) pebble the last node and then (b) remove all pebbles



Pebbles

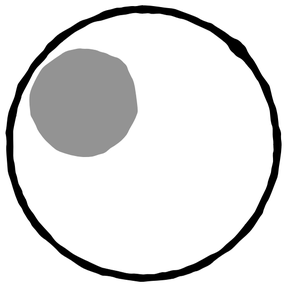


Rules:

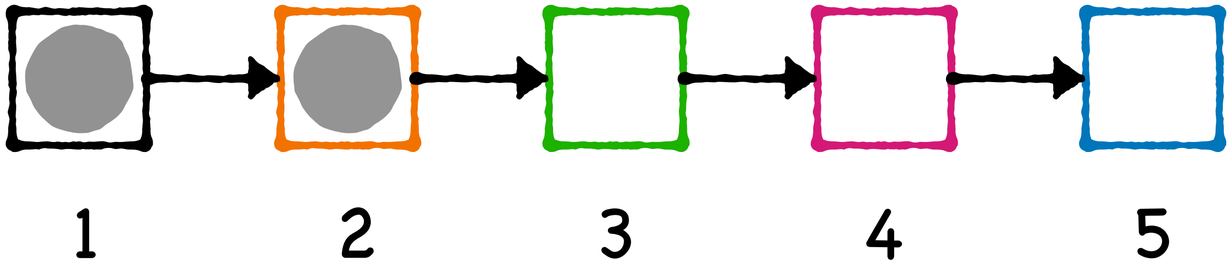
(1) You can place or remove a pebble on 1

(2) You can place or remove a pebble on $i > 1$ if $i - 1$ is pebbled

Goal: (a) pebble the last node and then (b) remove all pebbles



Pebbles

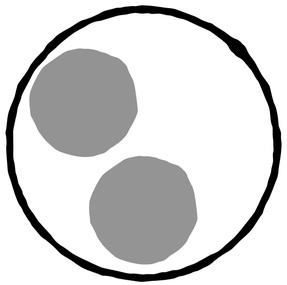


Rules:

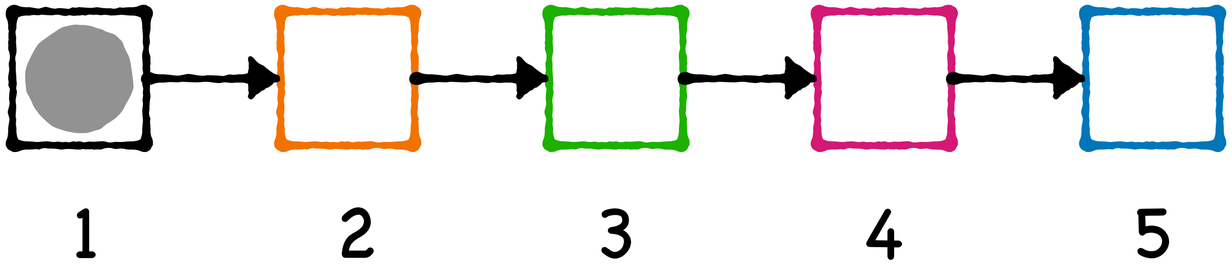
(1) You can place or remove a pebble on 1

(2) You can place or remove a pebble on $i > 1$ if $i - 1$ is pebbled

Goal: (a) pebble the last node and then (b) remove all pebbles



Pebbles

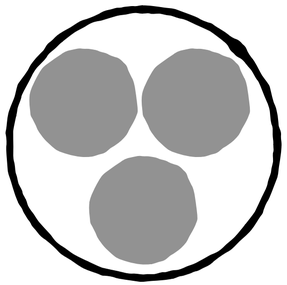


Rules:

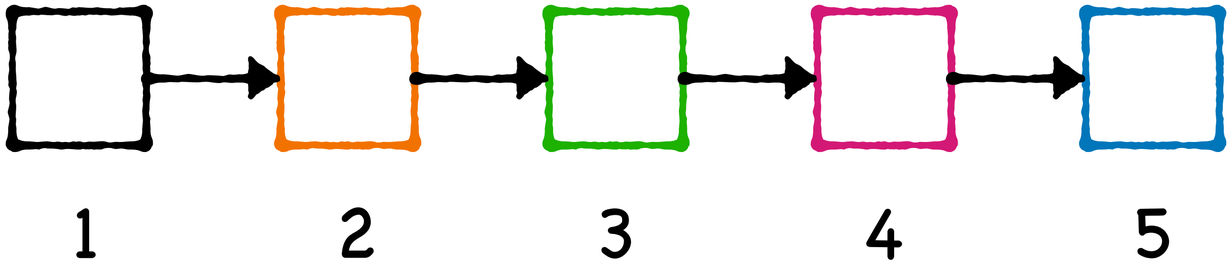
(1) You can place or remove a pebble on 1

(2) You can place or remove a pebble on $i > 1$ if $i - 1$ is pebbled

Goal: (a) pebble the last node and then (b) remove all pebbles



Pebbles

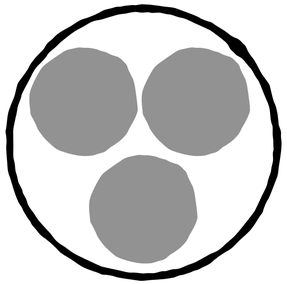


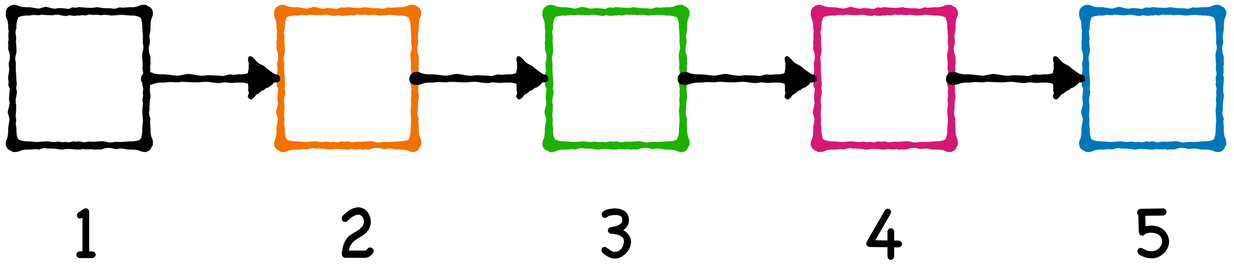
Rules:

(1) You can place or remove a pebble on 1

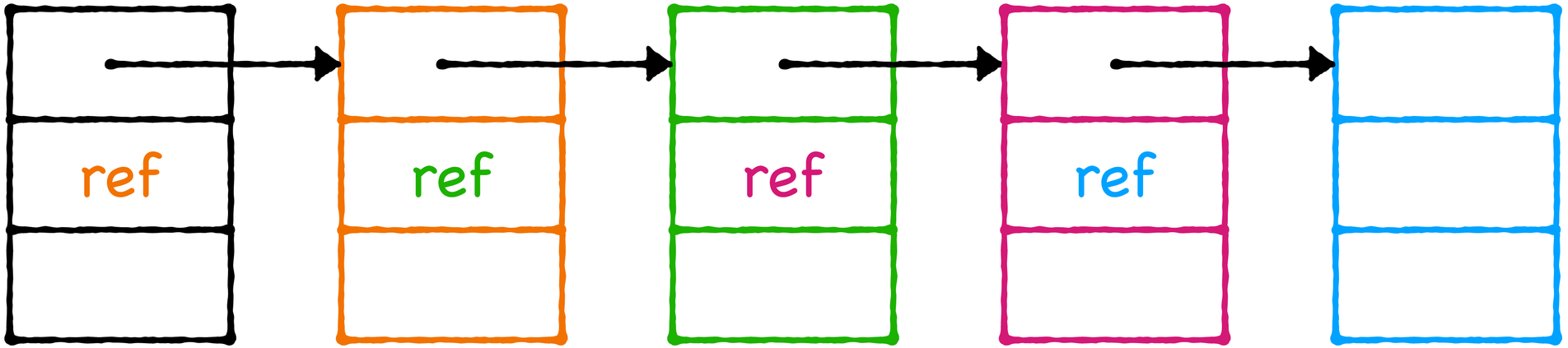
(2) You can place or remove a pebble on $i > 1$ if $i - 1$ is pebbled

Goal: (a) pebble the last node and then (b) remove all pebbles



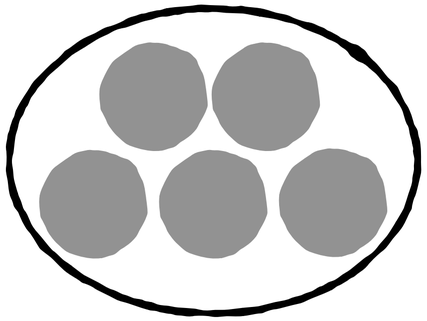


Pebbles

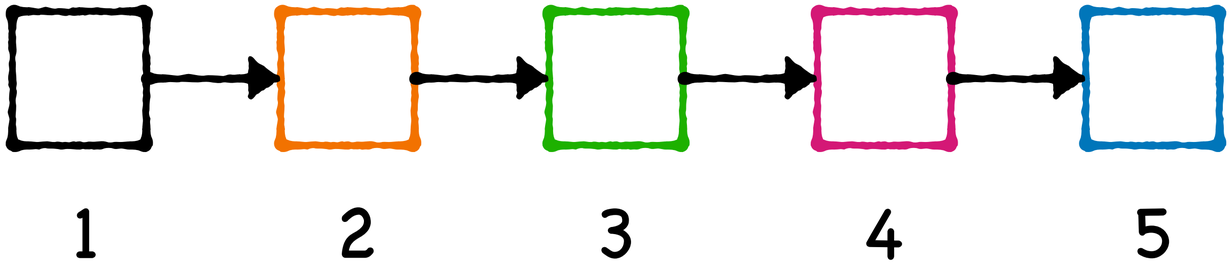


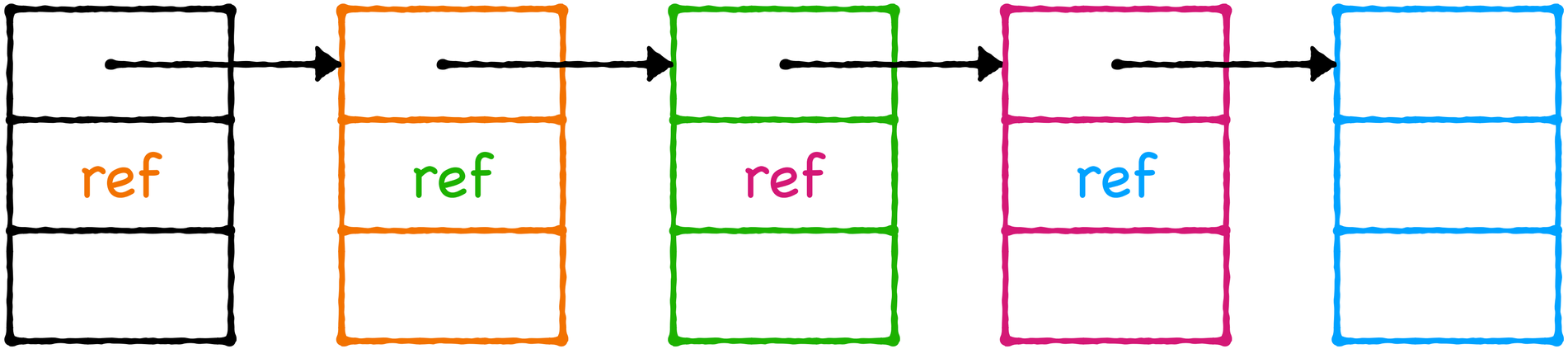
Head

# of acquired cells: 0



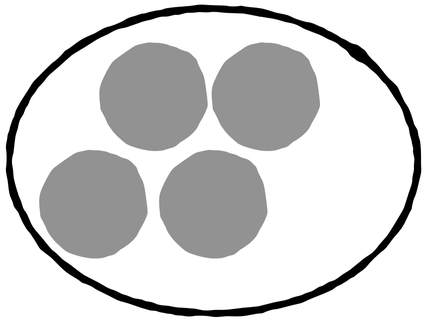
Pebbles



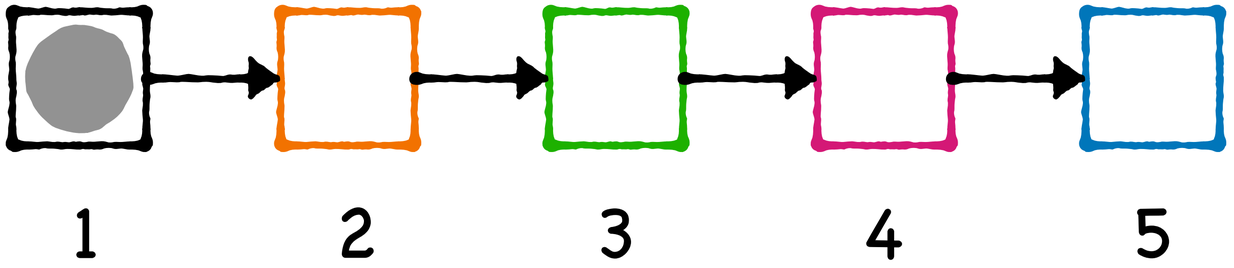


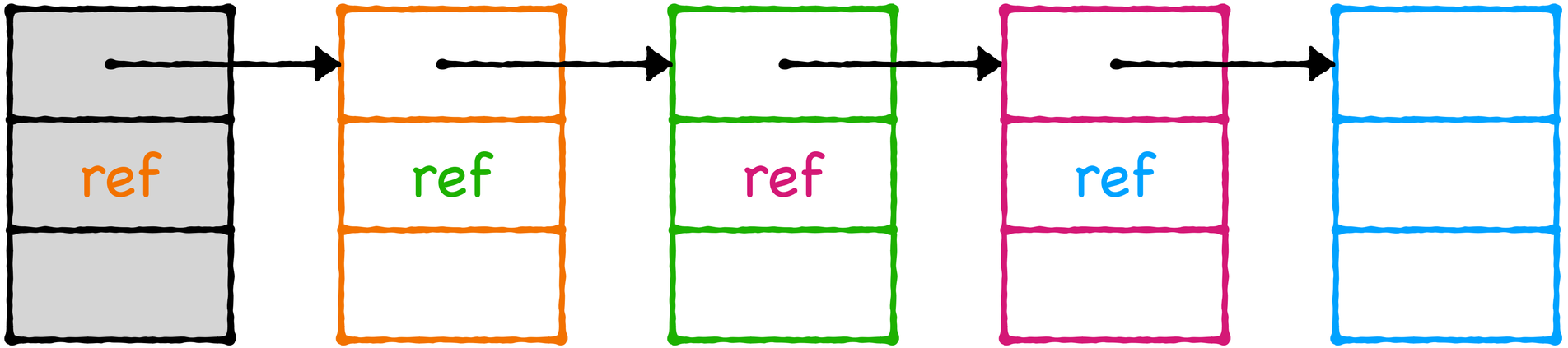
Head

# of acquired cells: 0



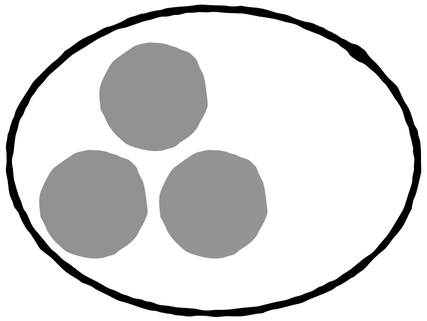
Pebbles



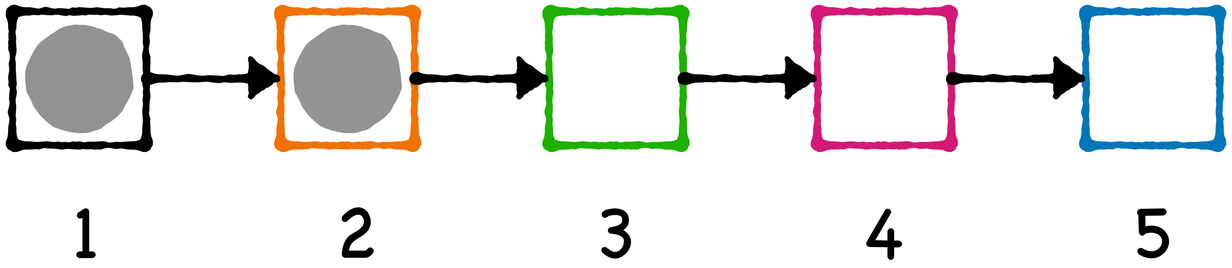


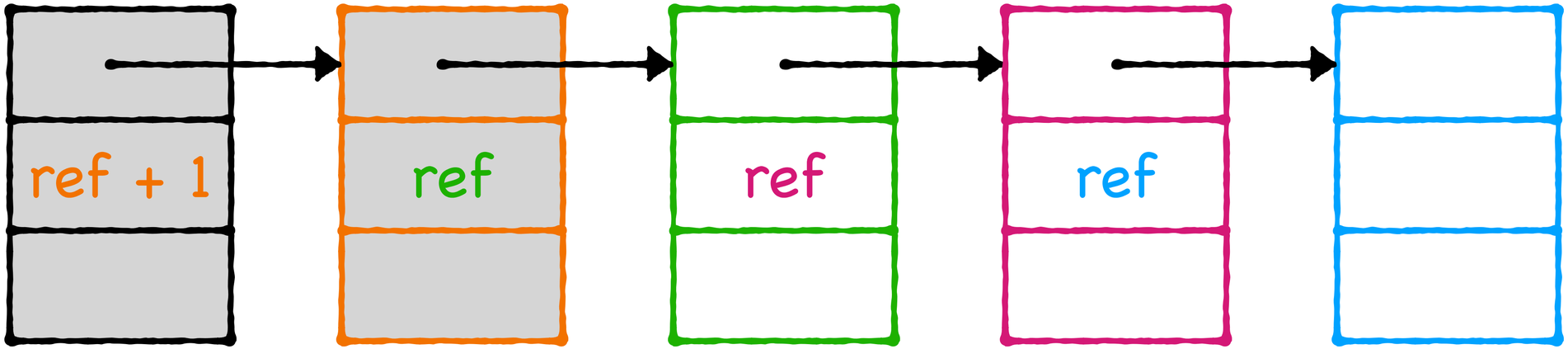
Head

# of acquired cells: 1



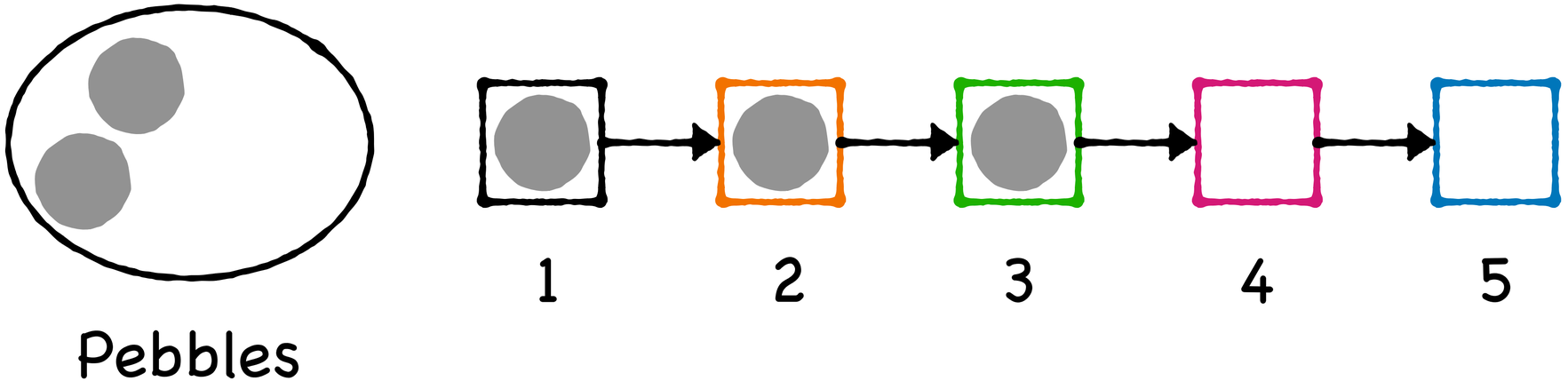
Pebbles

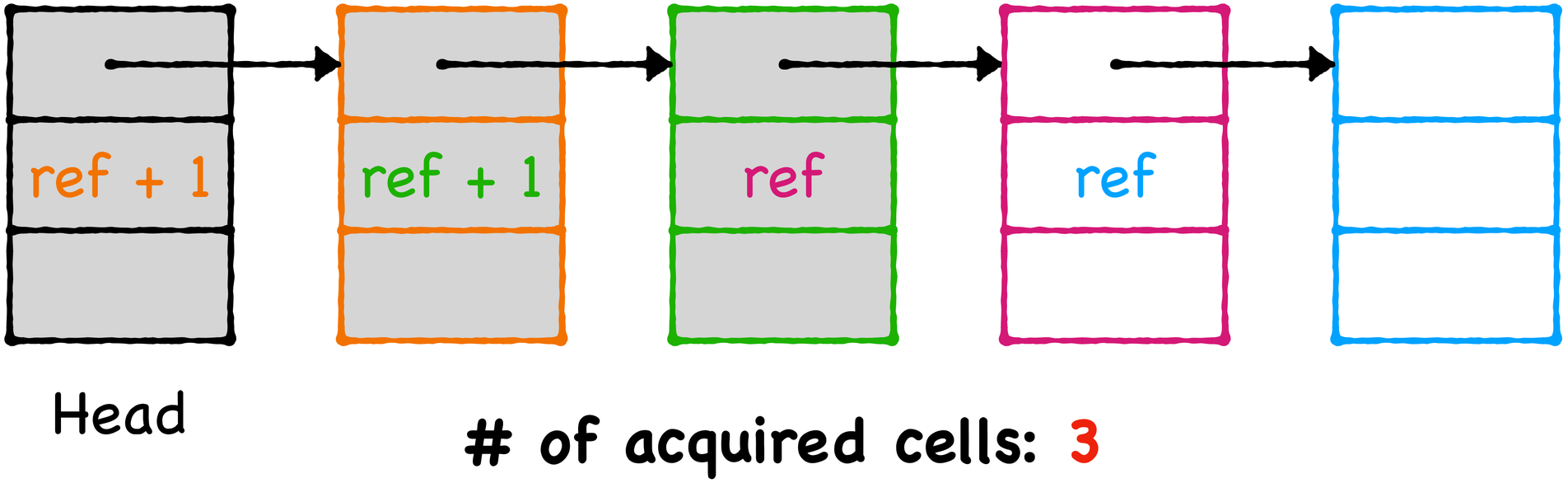




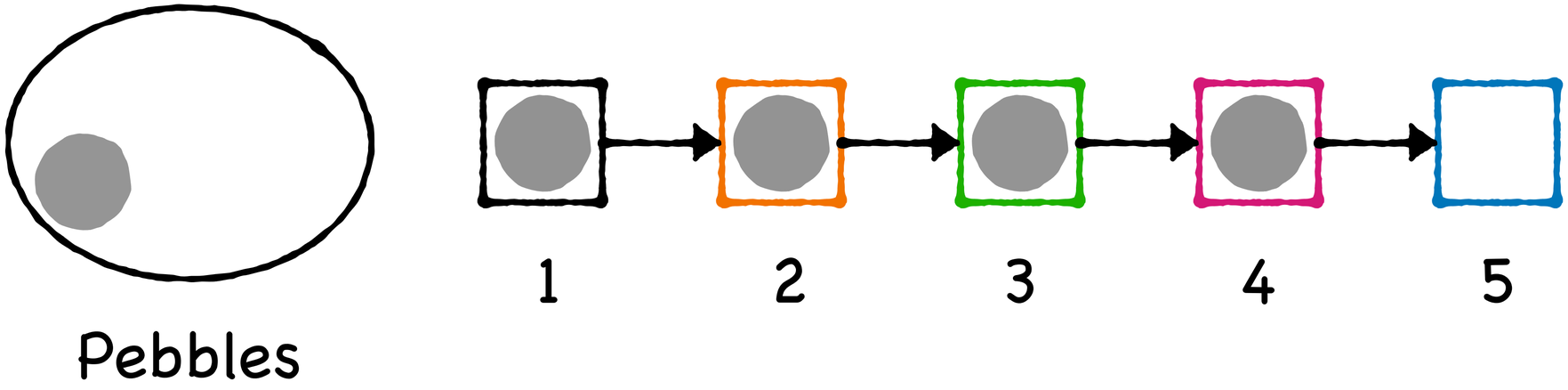
Head

# of acquired cells: 2

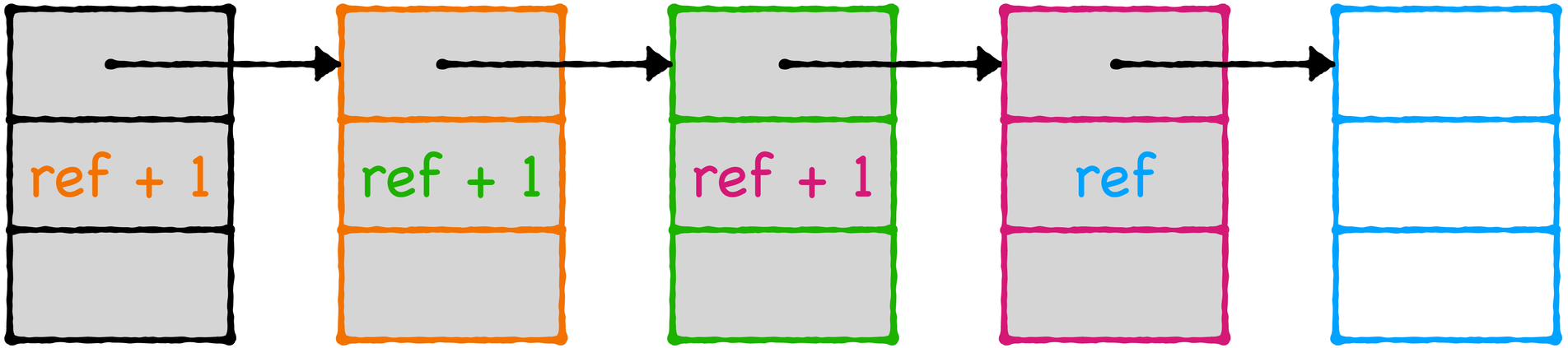




# of acquired cells: 3

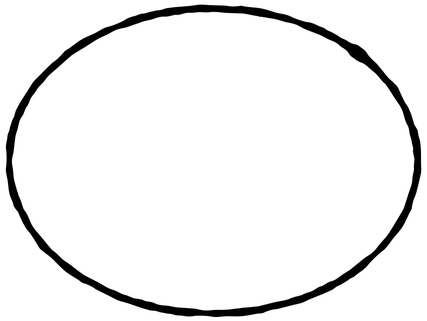


Pebbles

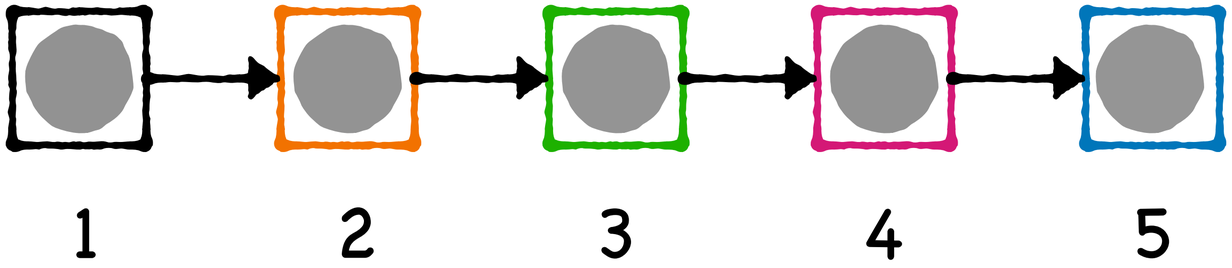


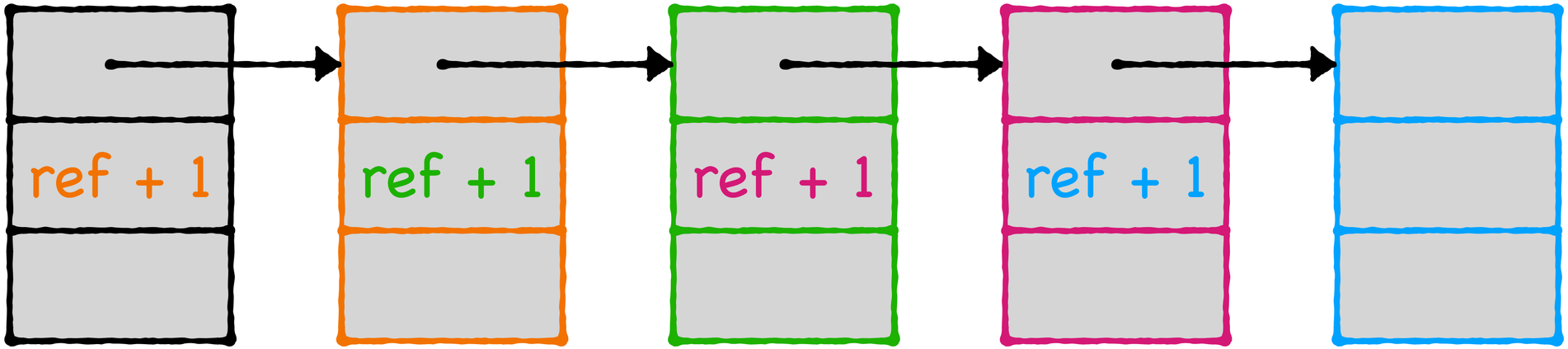
Head

# of acquired cells: 4



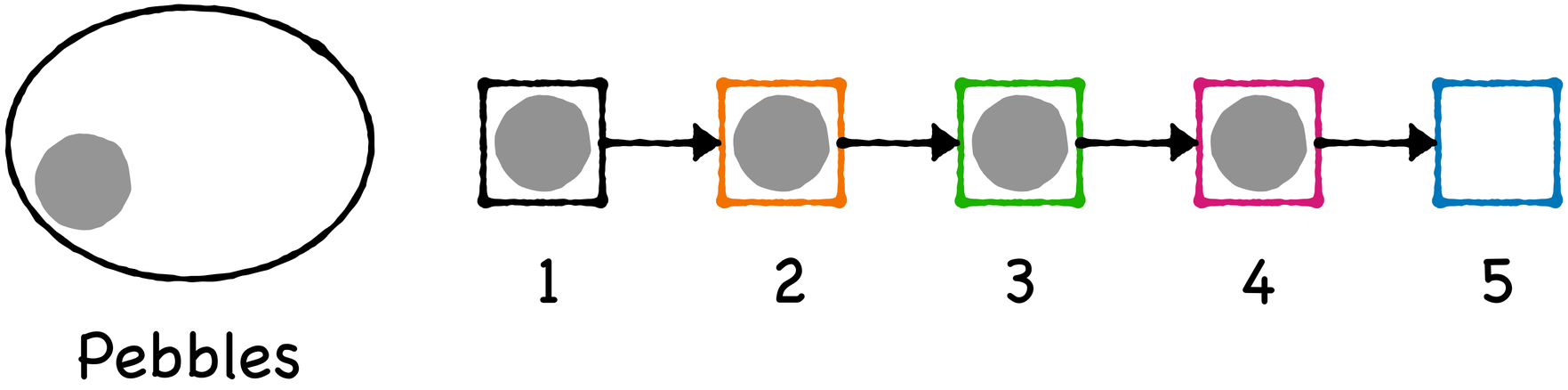
Pebbles



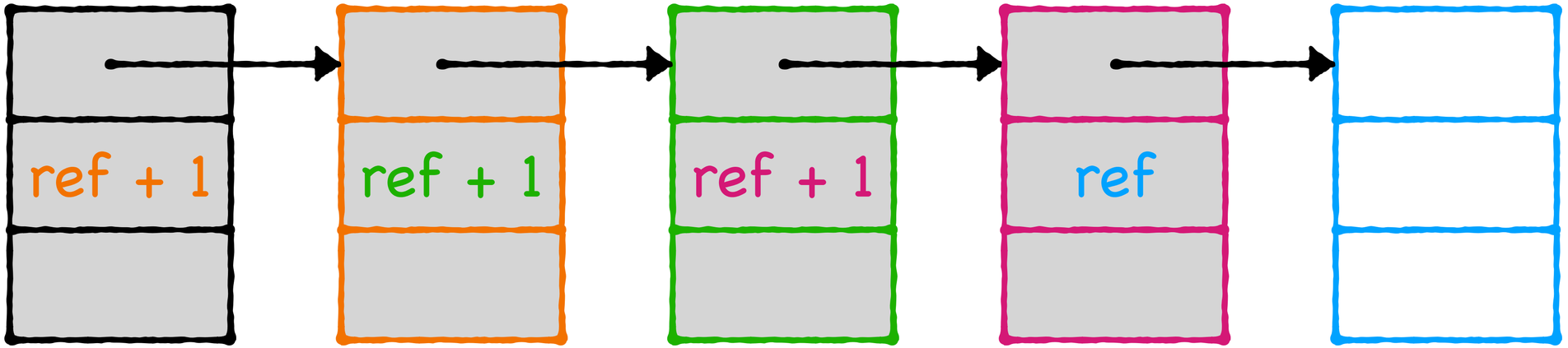


Head

# of acquired cells: 5

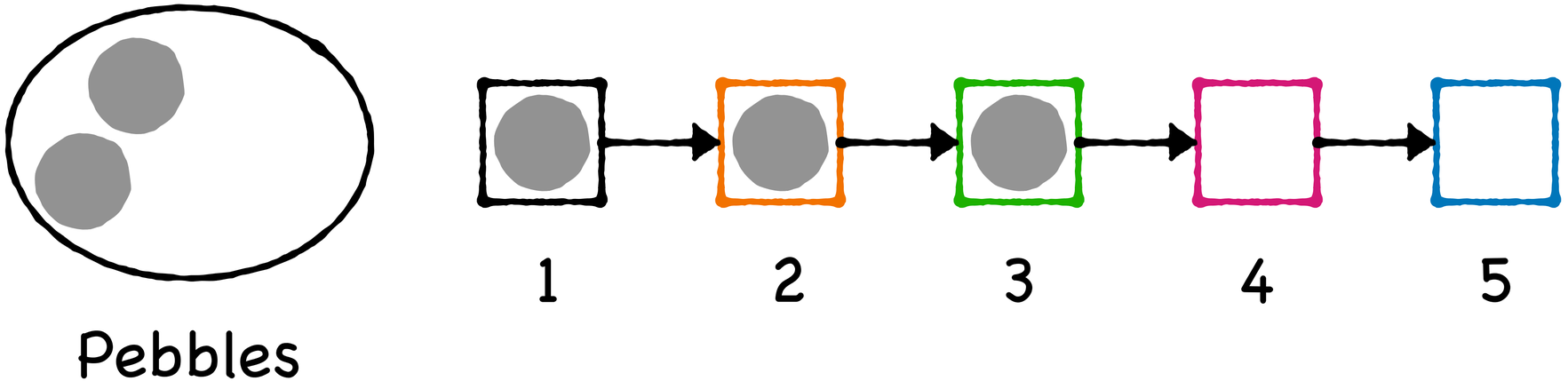


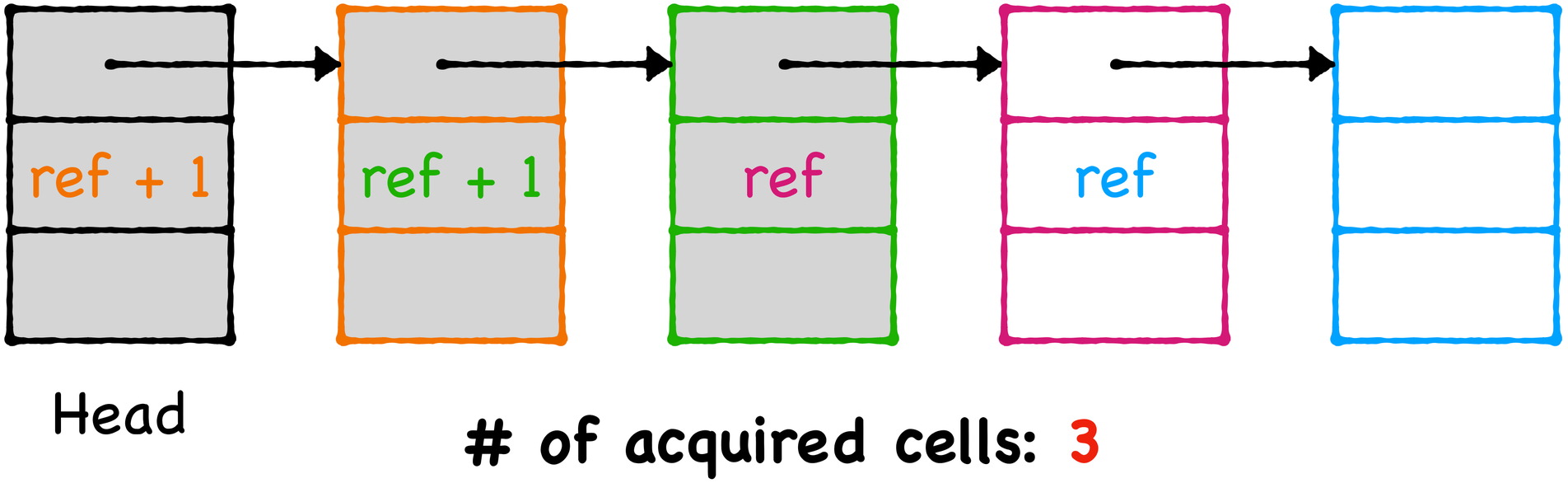
Pebbles



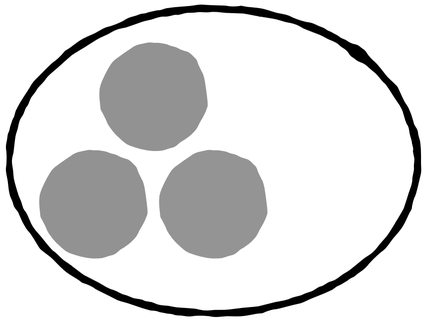
Head

# of acquired cells: 4

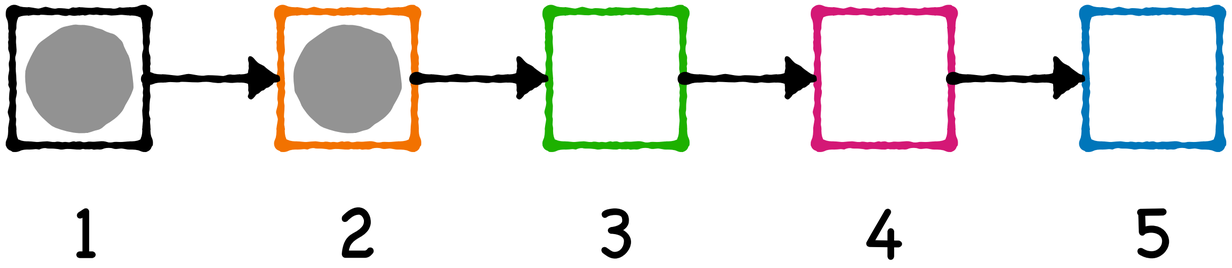


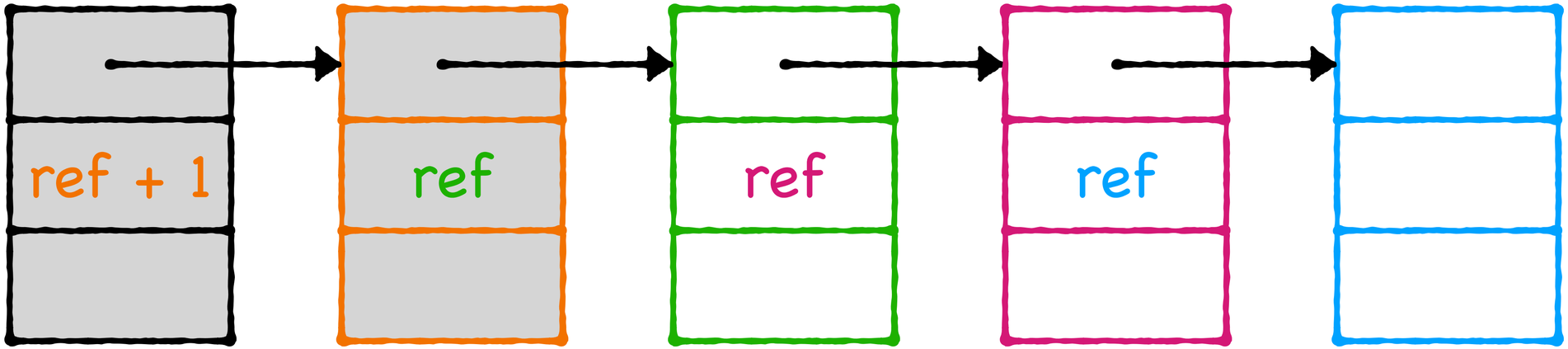


# of acquired cells: 3



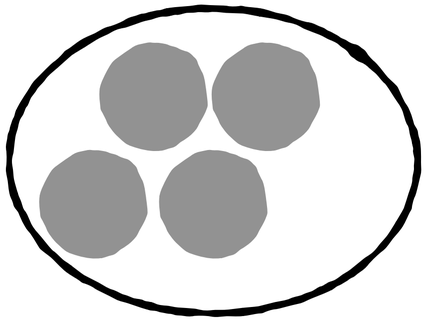
Pebbles



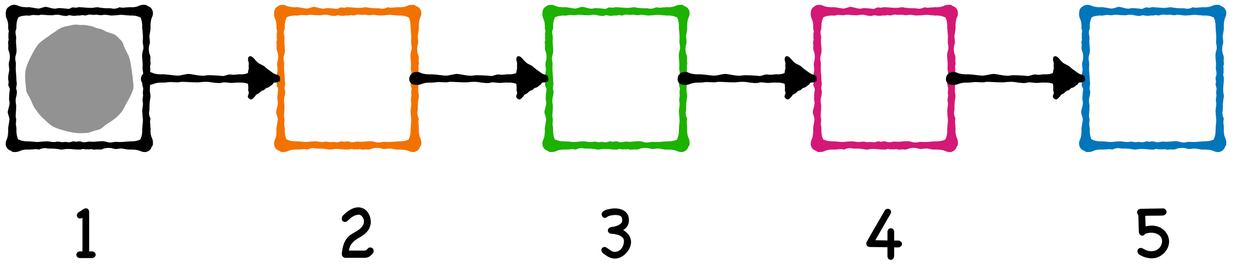


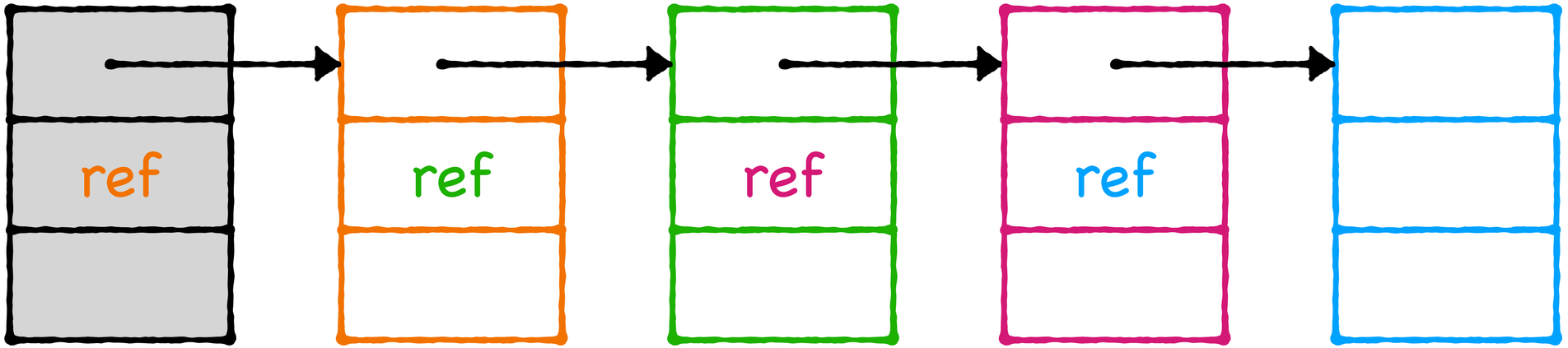
Head

# of acquired cells: 2



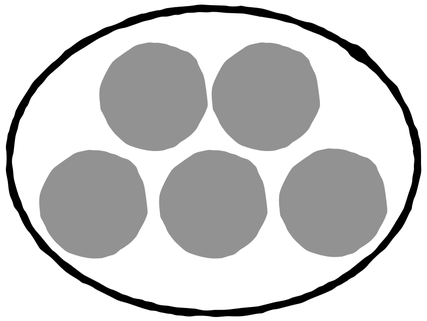
Pebbles



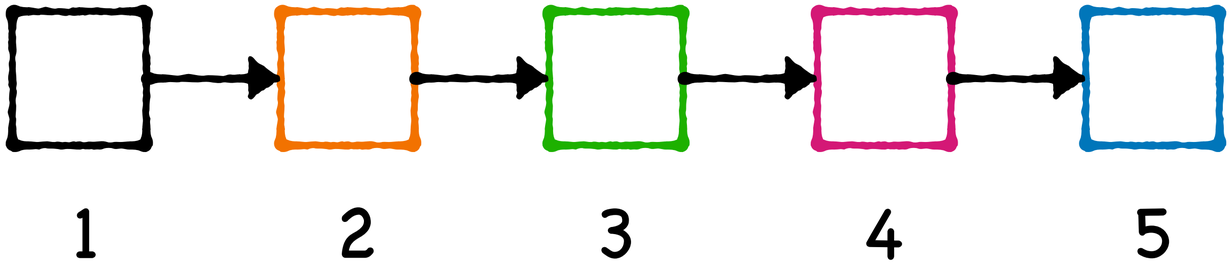


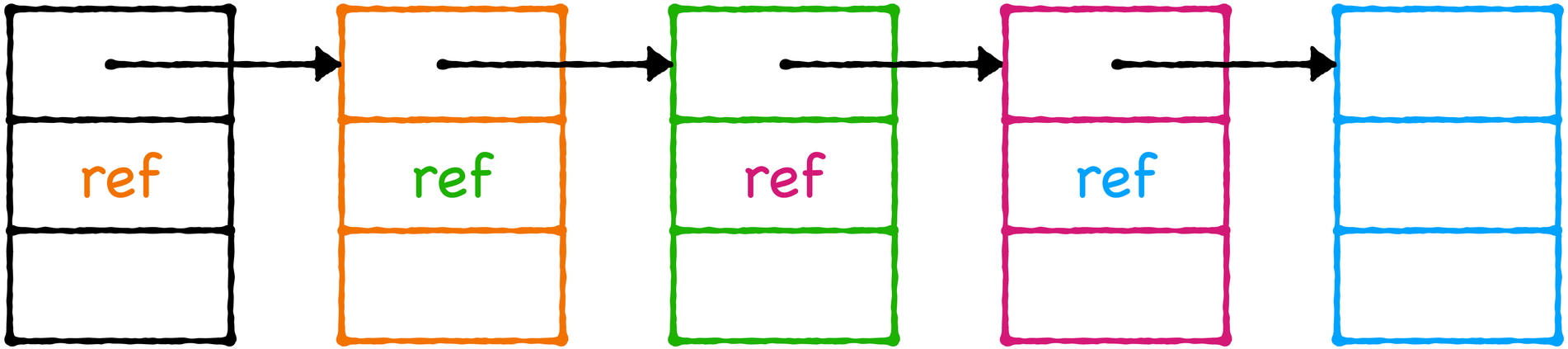
Head

# of acquired cells: 1



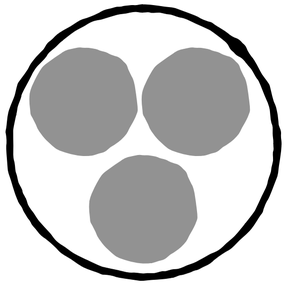
Pebbles

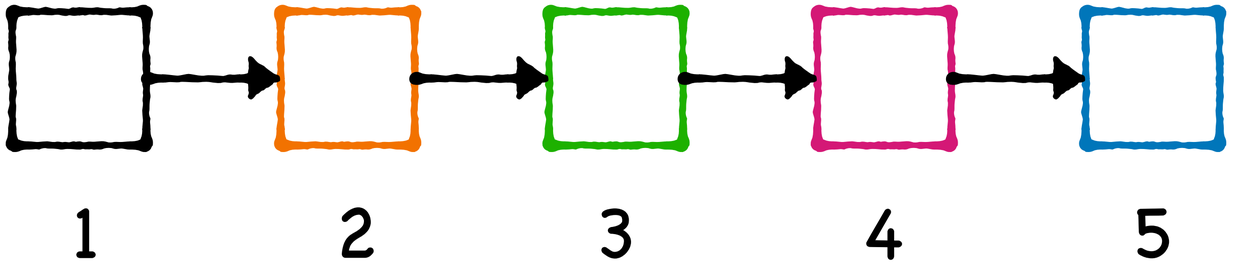




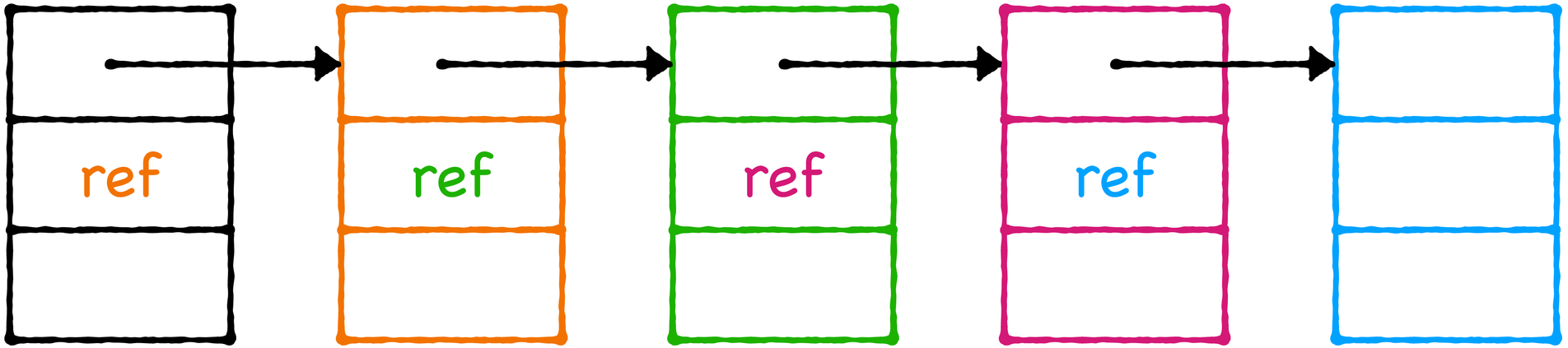
Head

# of acquired cells: 0



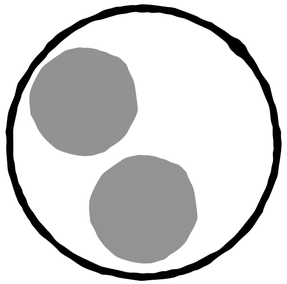


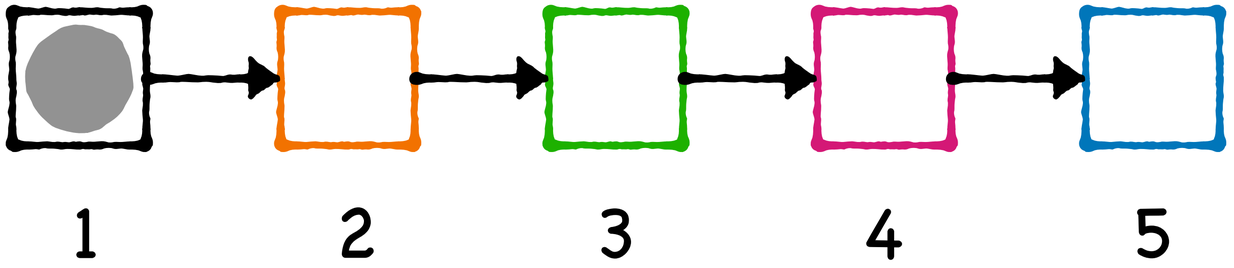
Pebbles



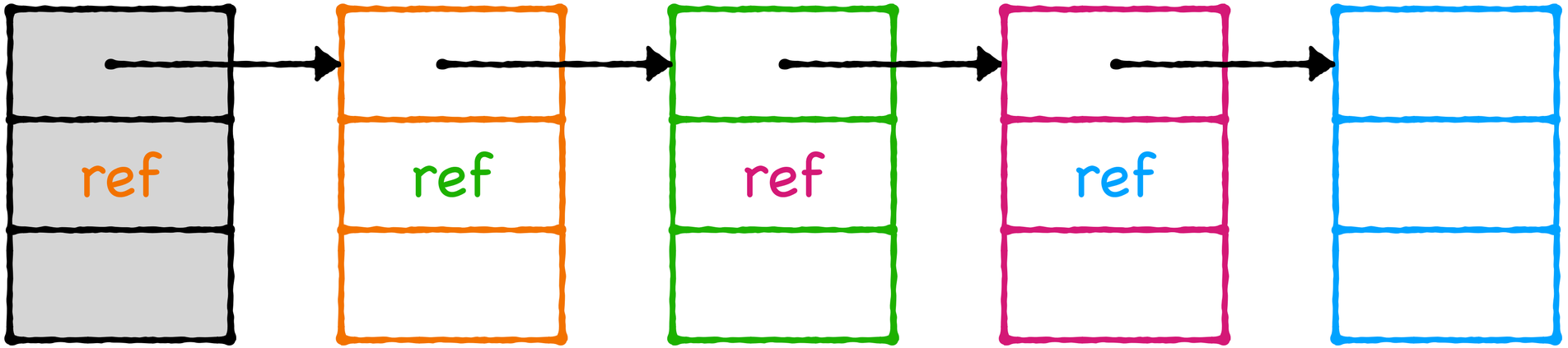
Head

# of acquired cells: 0



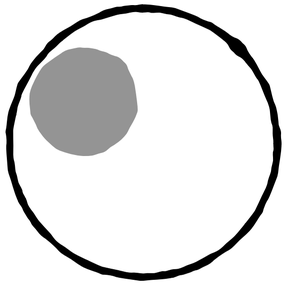


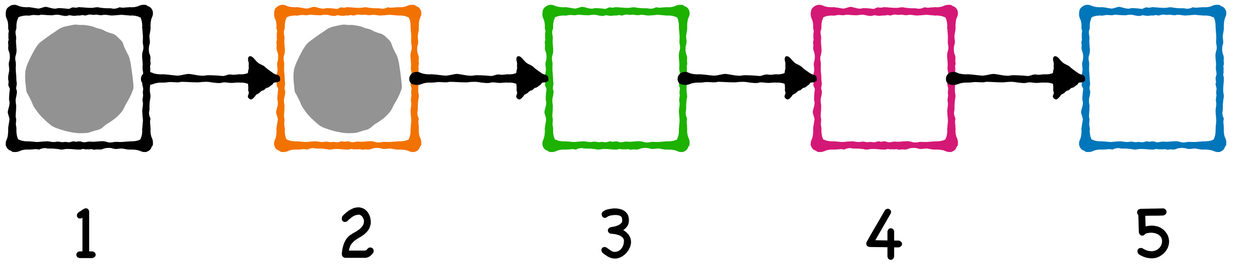
Pebbles



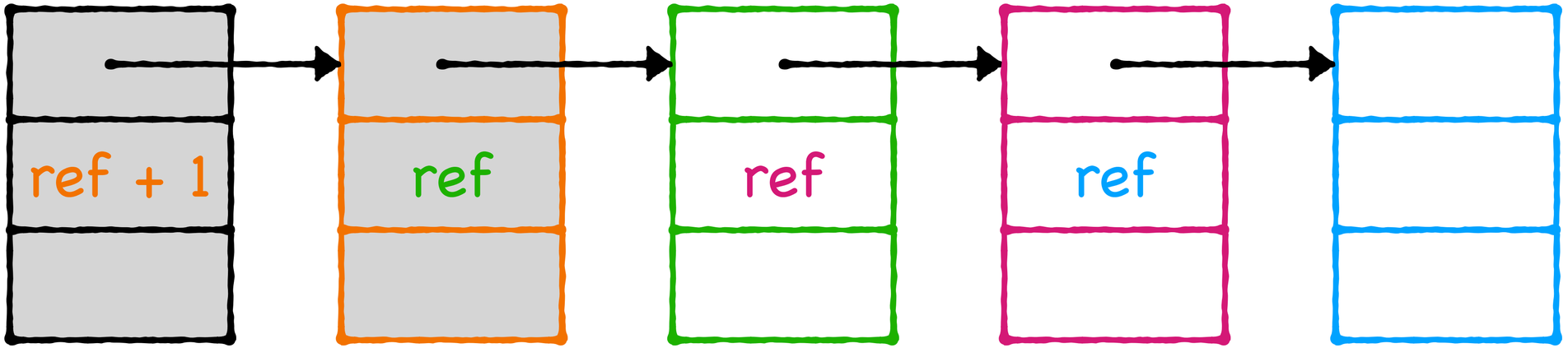
Head

# of acquired cells: 1



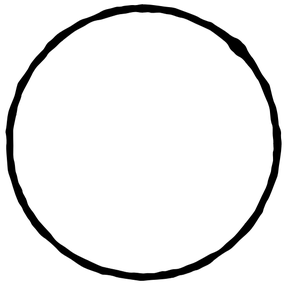


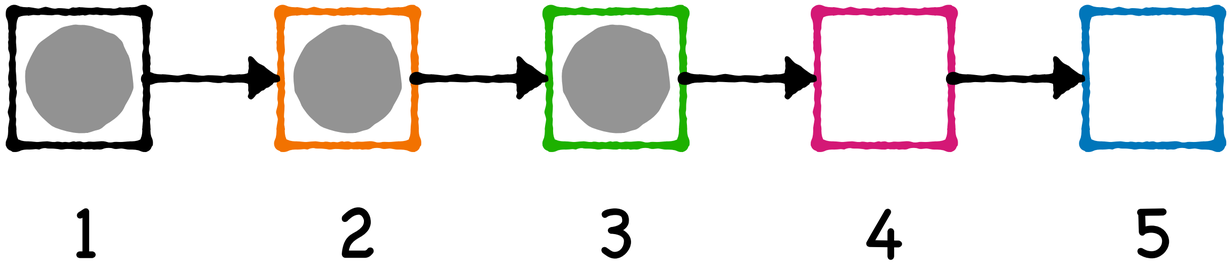
Pebbles



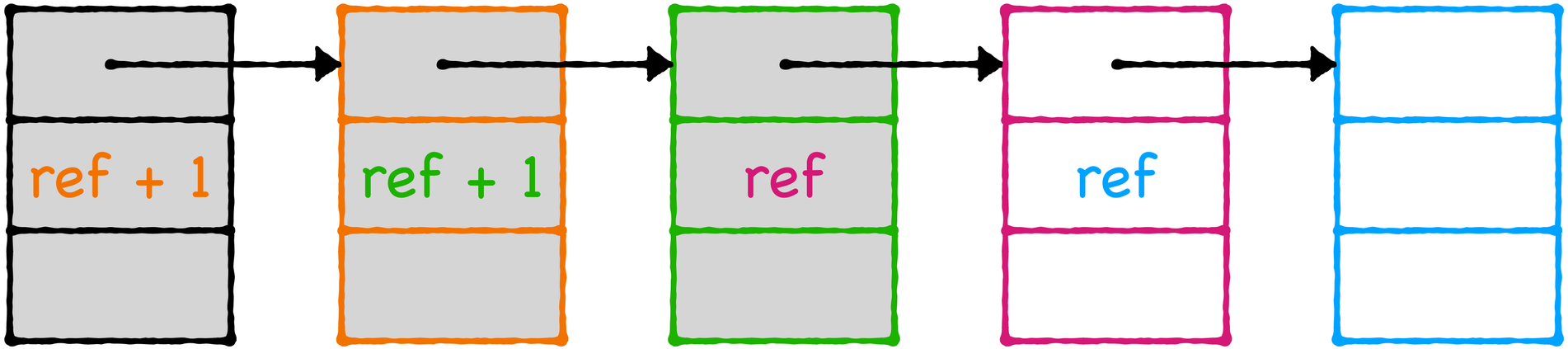
Head

# of acquired cells: 2



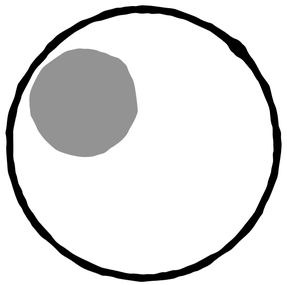


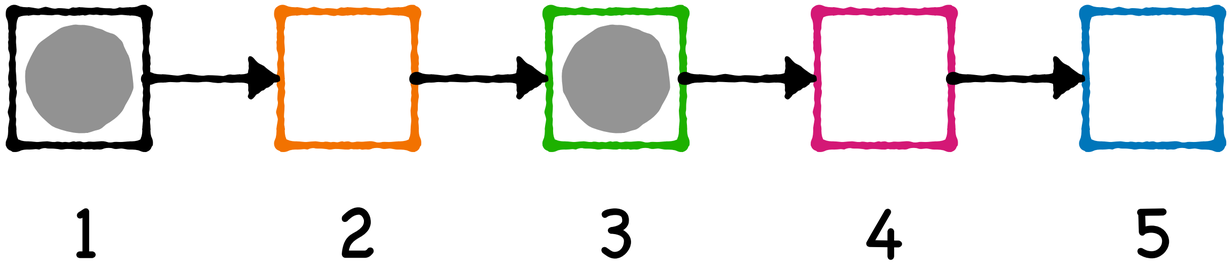
Pebbles



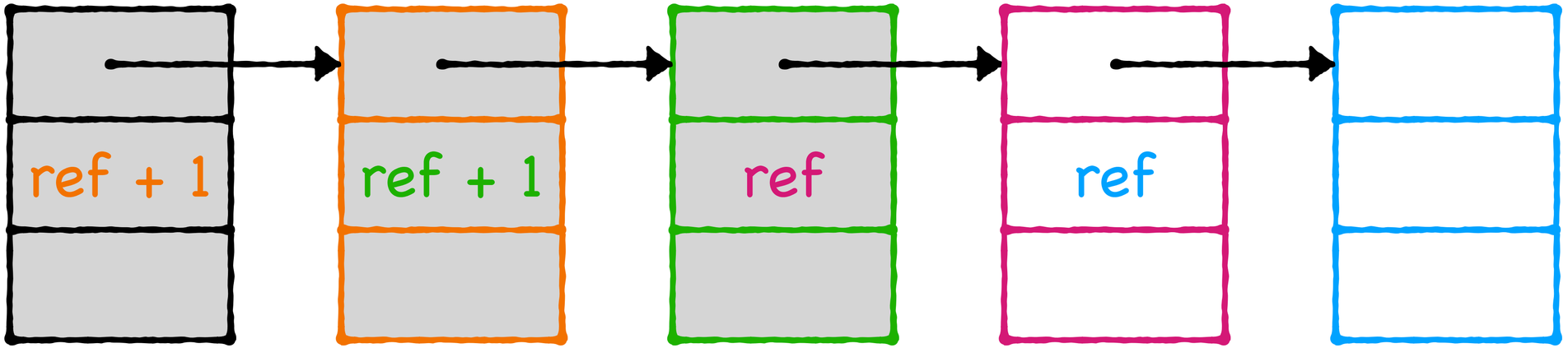
Head

# of acquired cells: 3



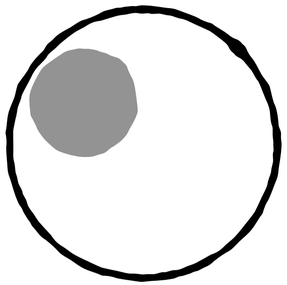


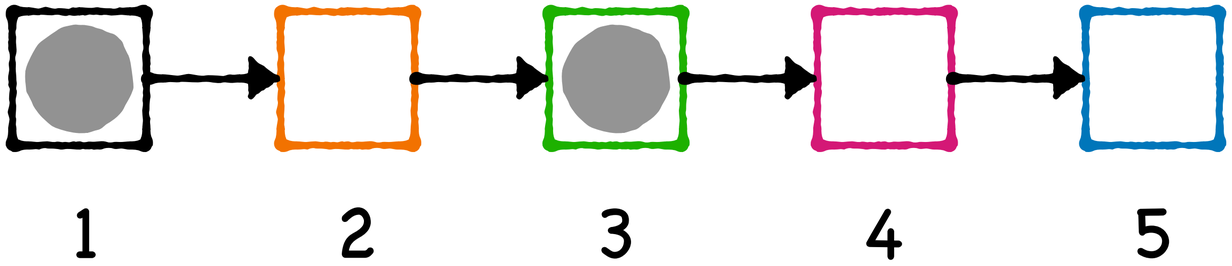
Pebbles



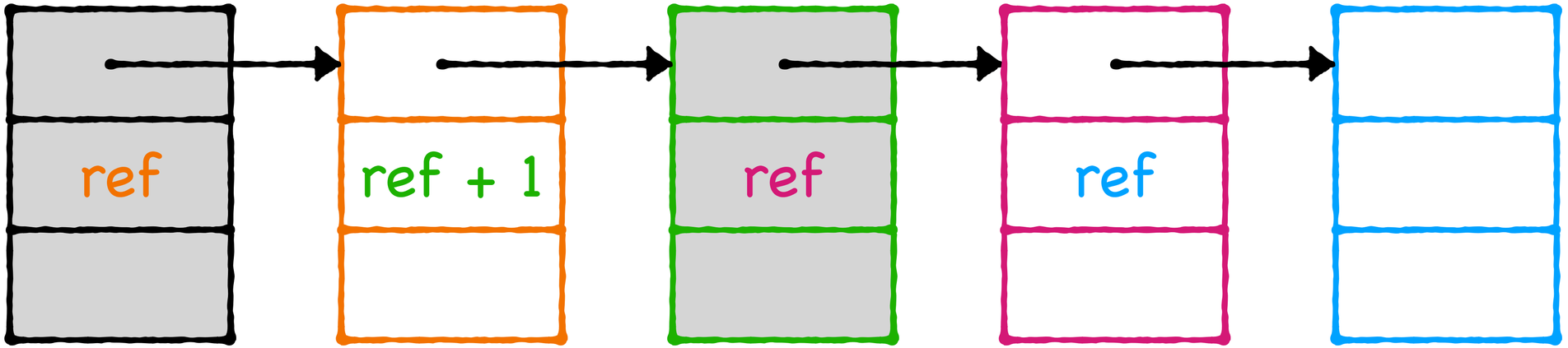
Head

# of acquired cells: 3



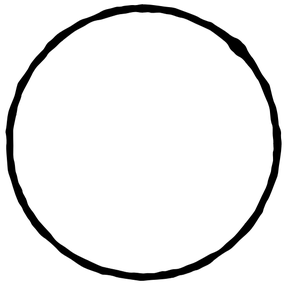


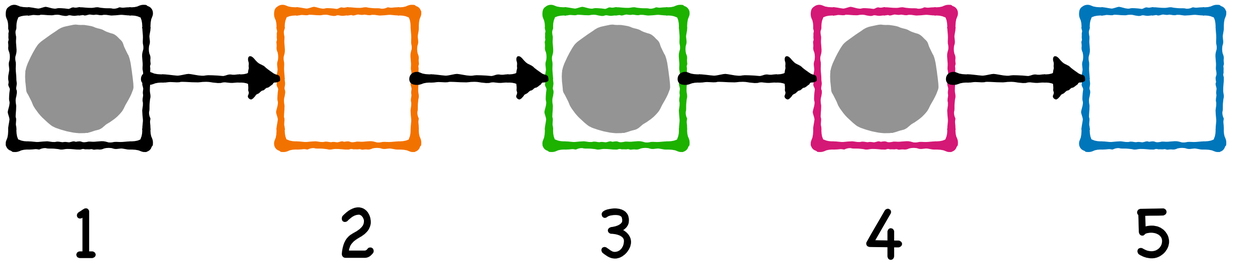
Pebbles



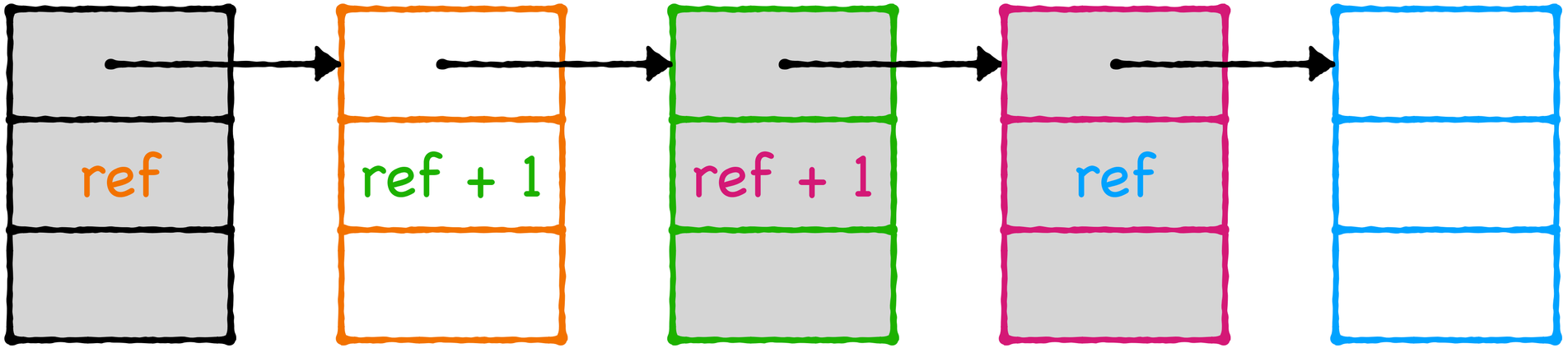
Head

# of acquired cells: 2



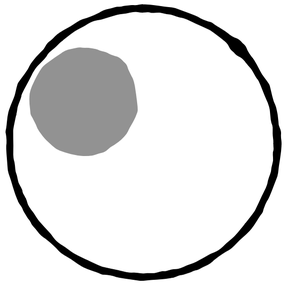


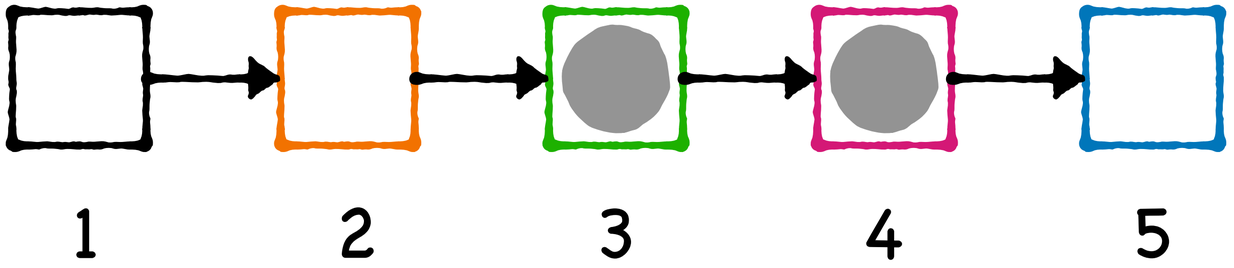
Pebbles



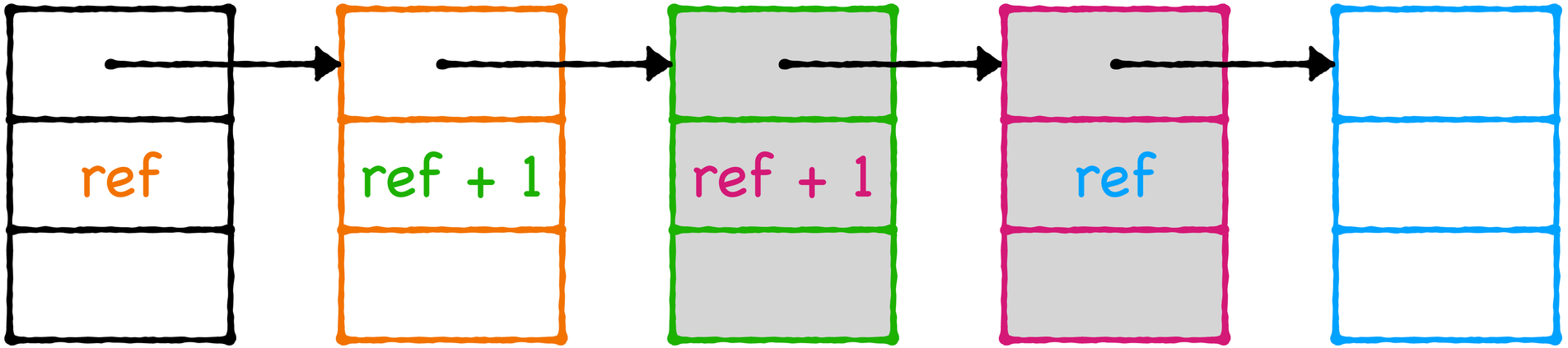
Head

# of acquired cells: 3



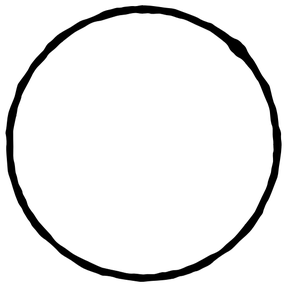


Pebbles

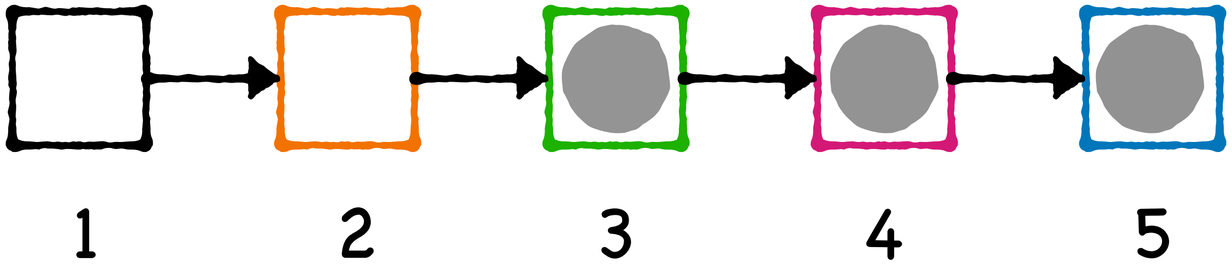


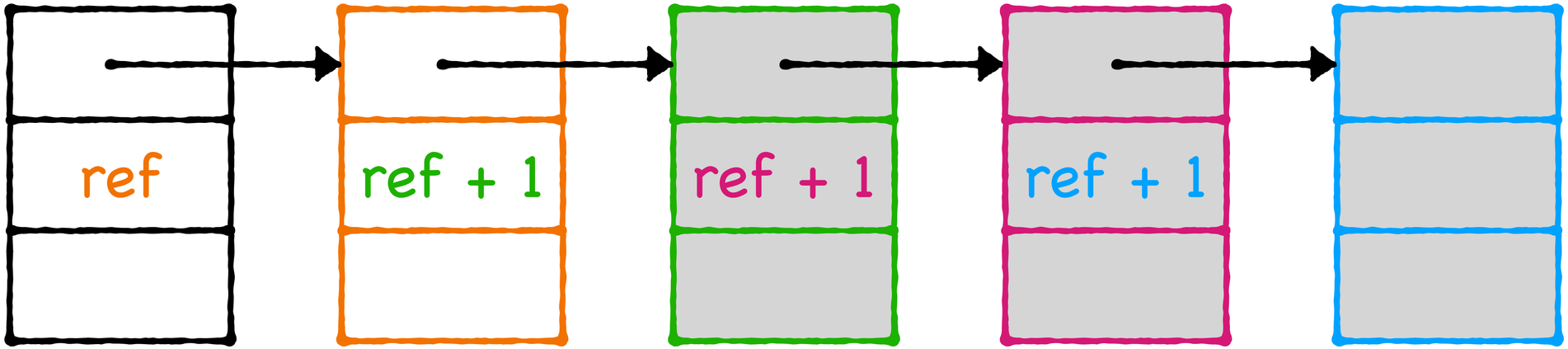
Head

# of acquired cells: 2



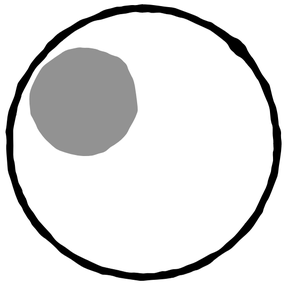
Pebbles

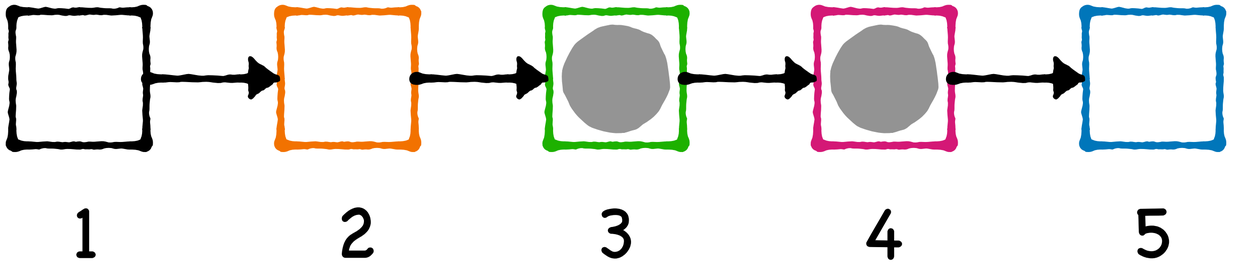




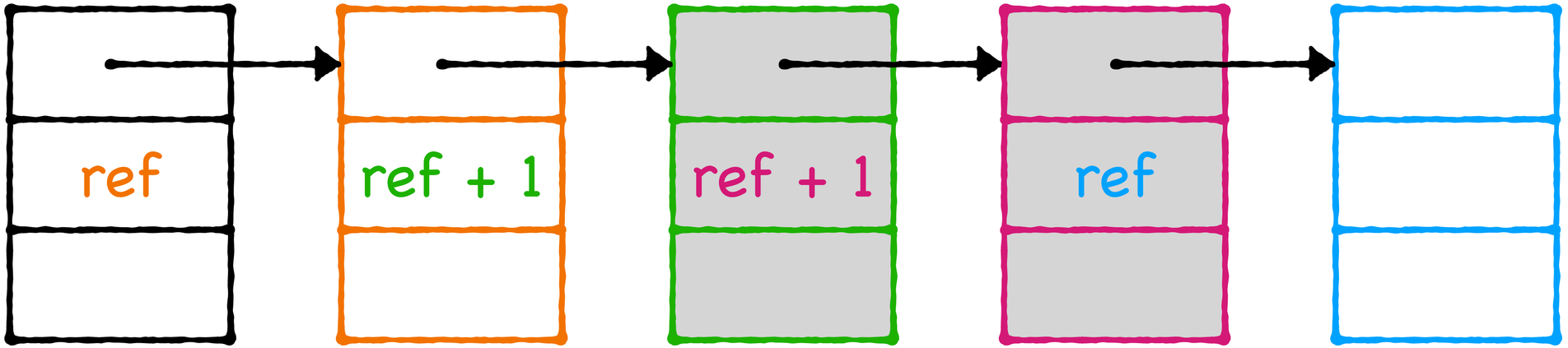
Head

# of acquired cells: 3



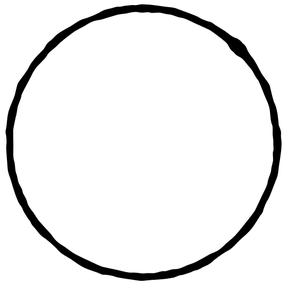


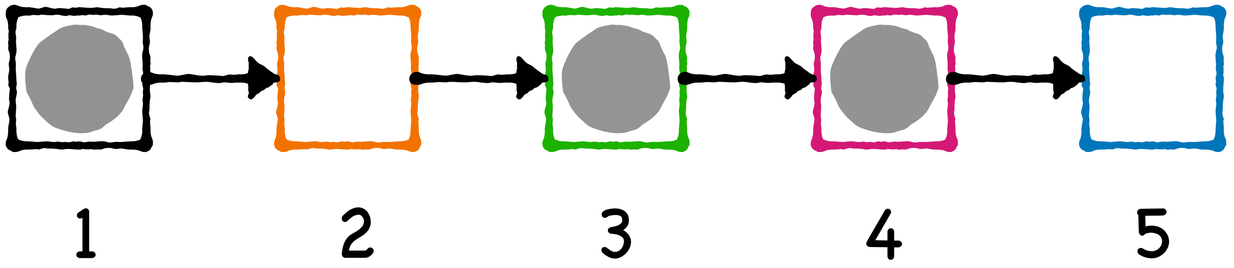
Pebbles



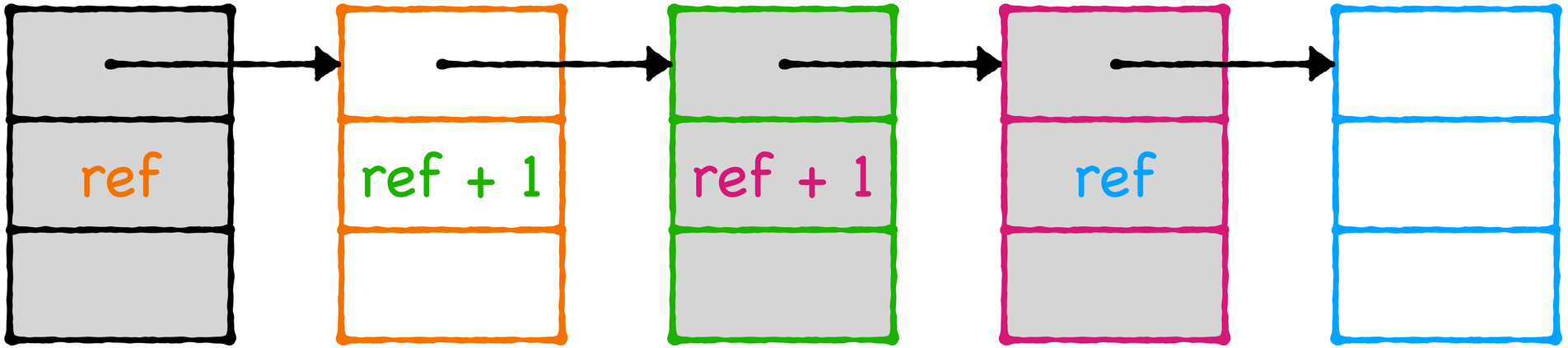
Head

# of acquired cells: 2



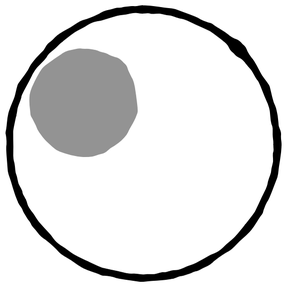


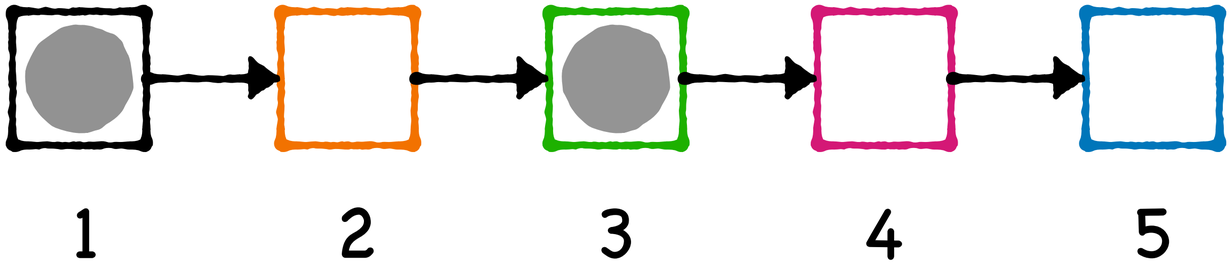
Pebbles



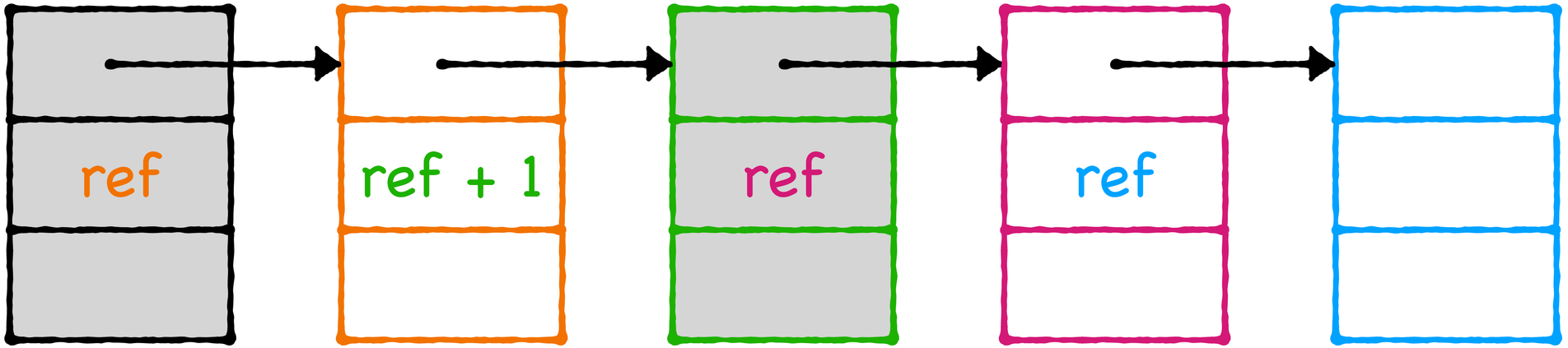
Head

# of acquired cells: 3



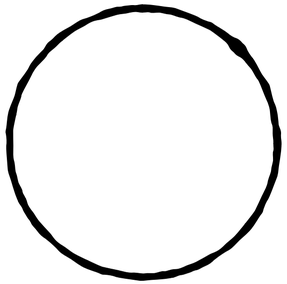


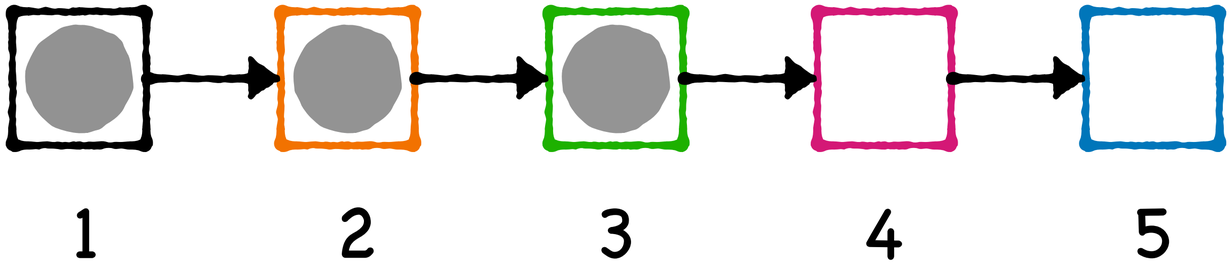
Pebbles



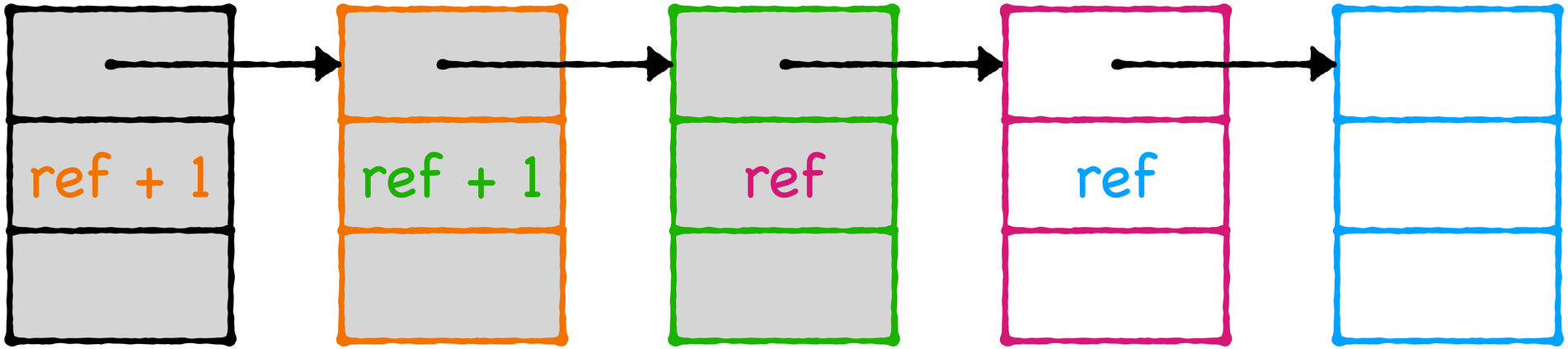
Head

# of acquired cells: 2



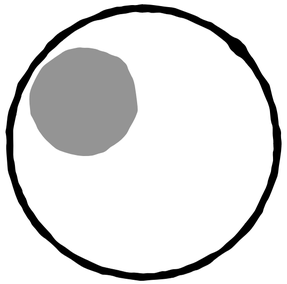


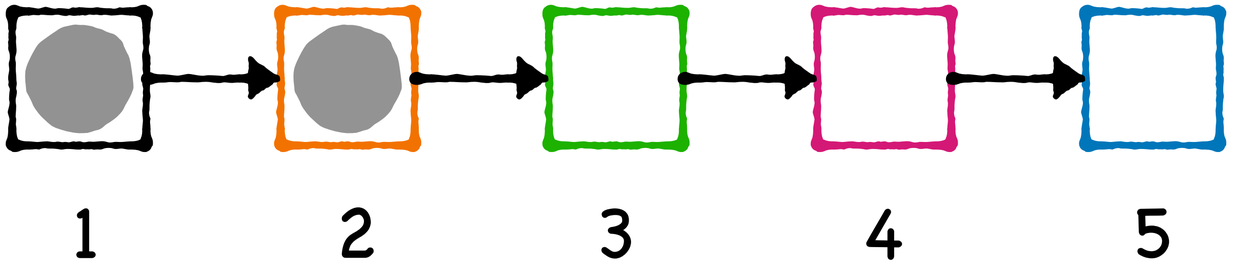
Pebbles



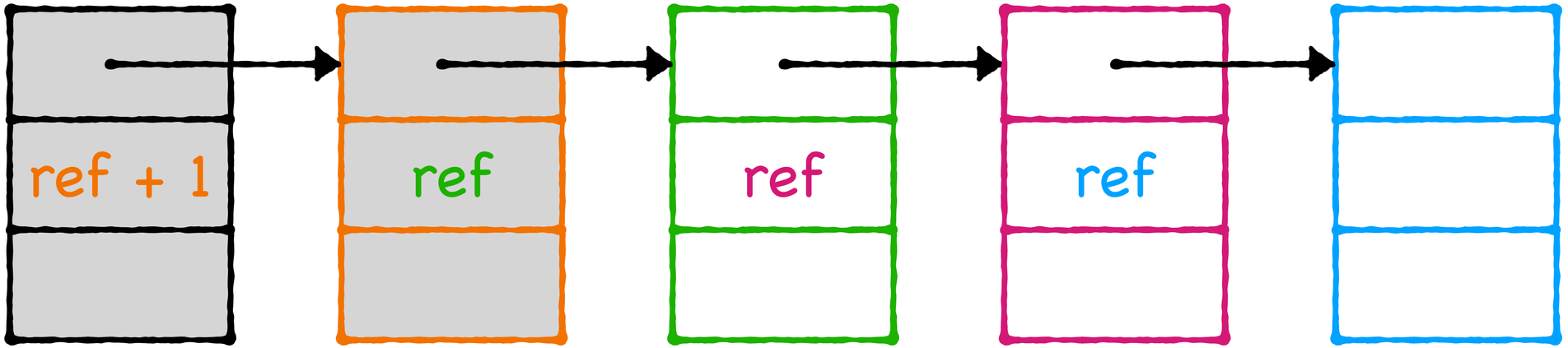
Head

# of acquired cells: 3



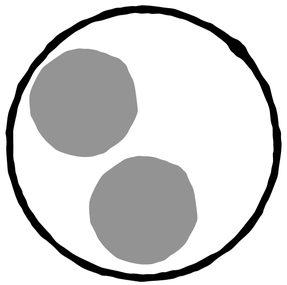


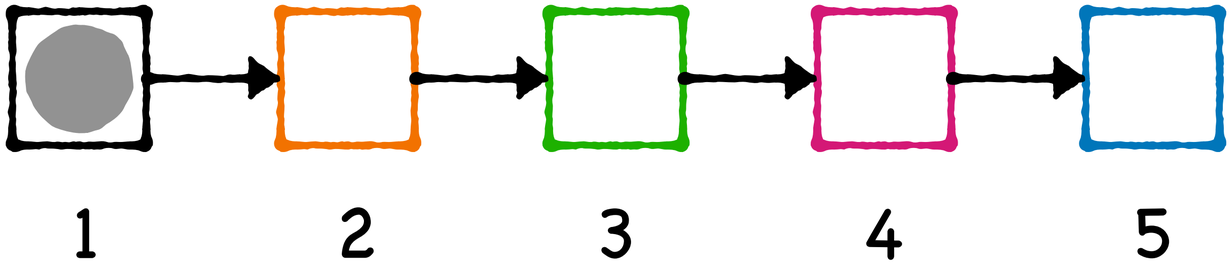
Pebbles



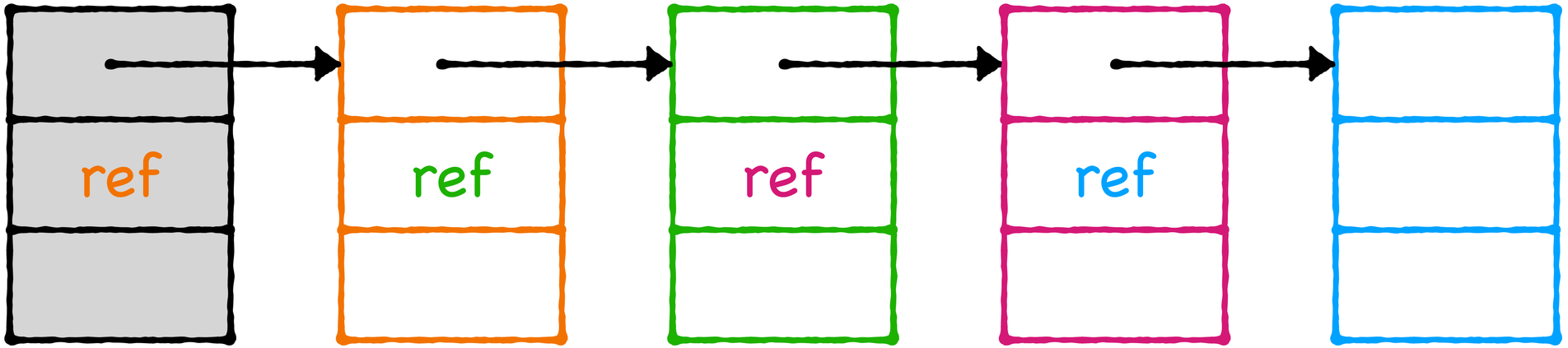
Head

# of acquired cells: 2



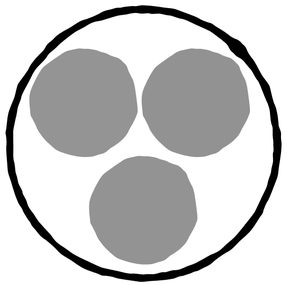


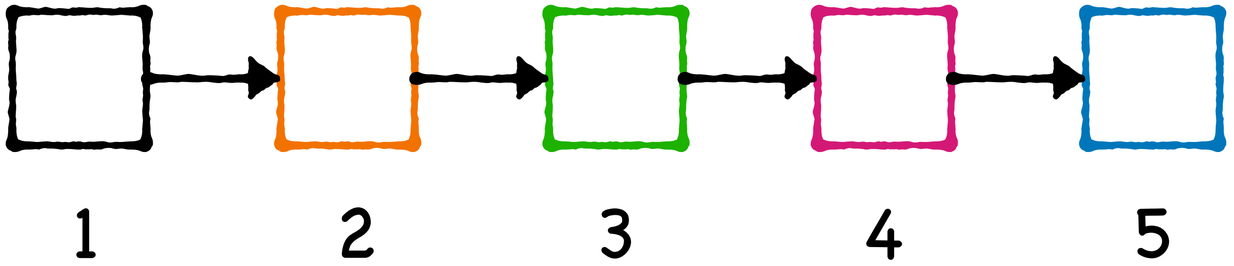
Pebbles



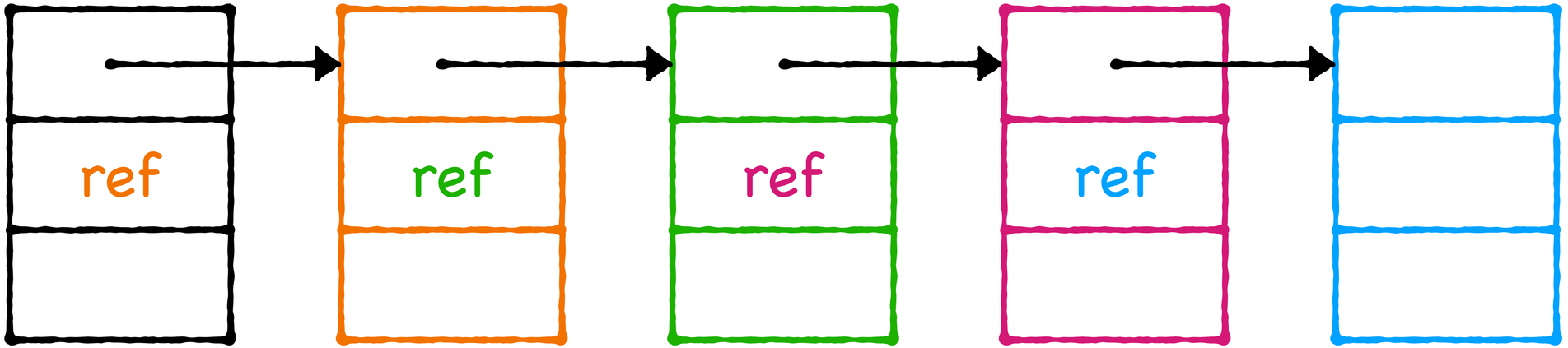
Head

# of acquired cells: 1





Pebbles



Head

# of acquired cells: 0

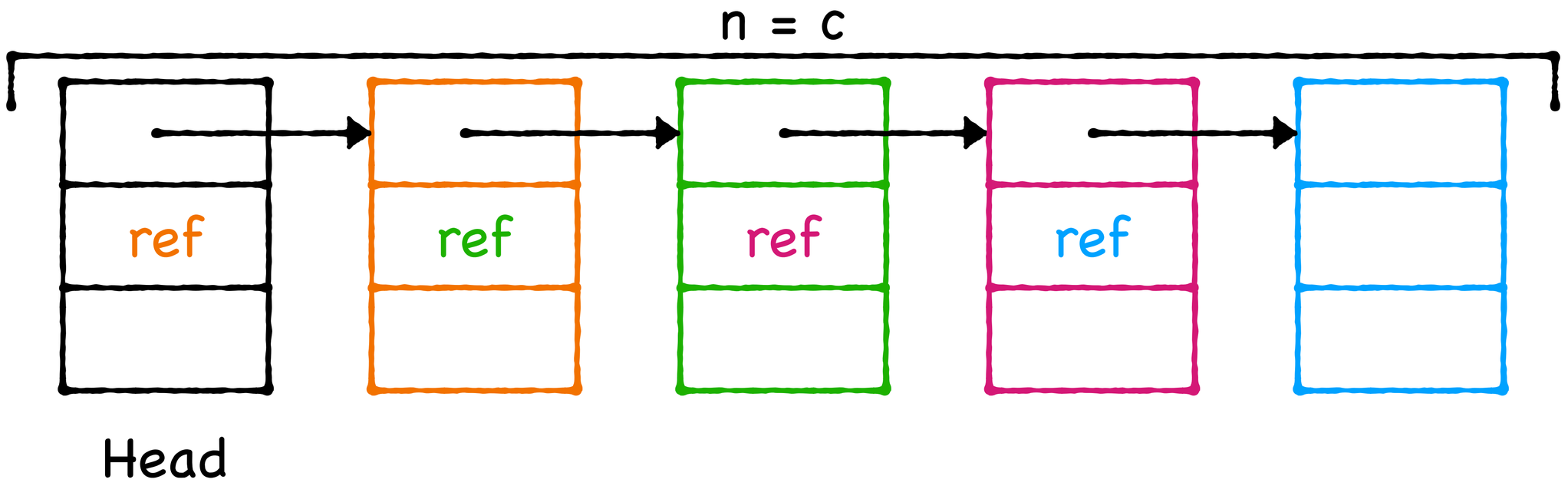
Lower Bound on the Number of Pebbles Needed

Lower Bound on the Number of Pebbles Needed

Theorem[Li&Vitanyi'97]: You can win the game for a list of n nodes with m pebbles **only if** $m > \log_2(n)$

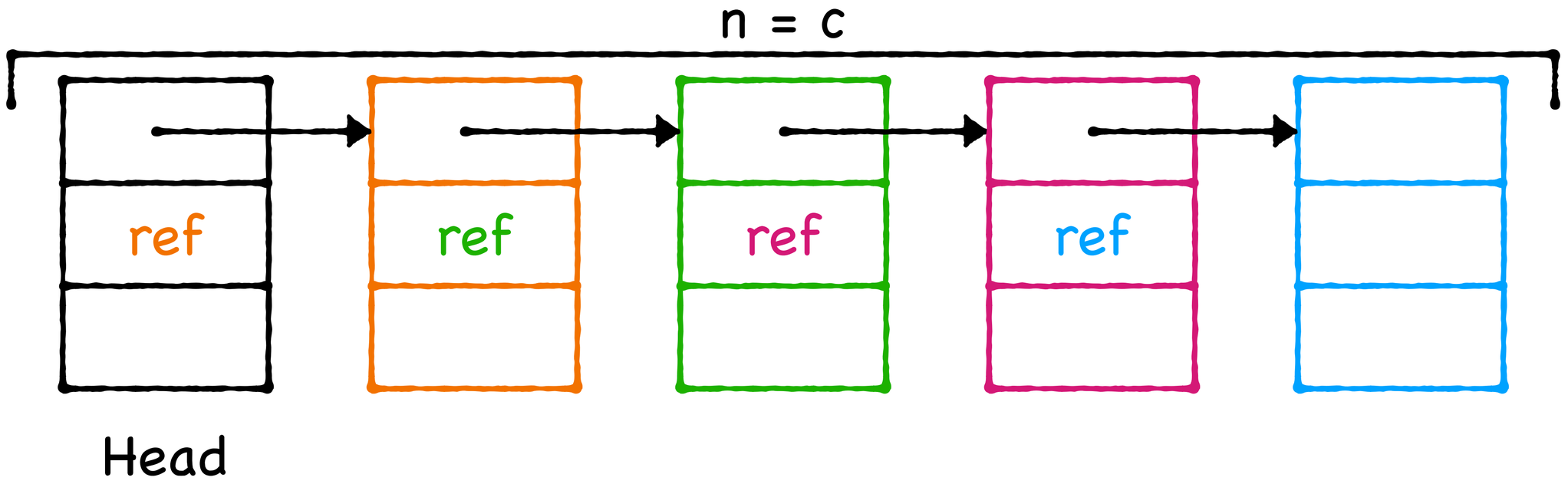
Lower Bound on the Number of Pebbles Needed

Theorem[Li&Vitanyi'97]: You can win the game for a list of n nodes with m pebbles **only if** $m > \log_2(n)$



Lower Bound on the Number of Pebbles Needed

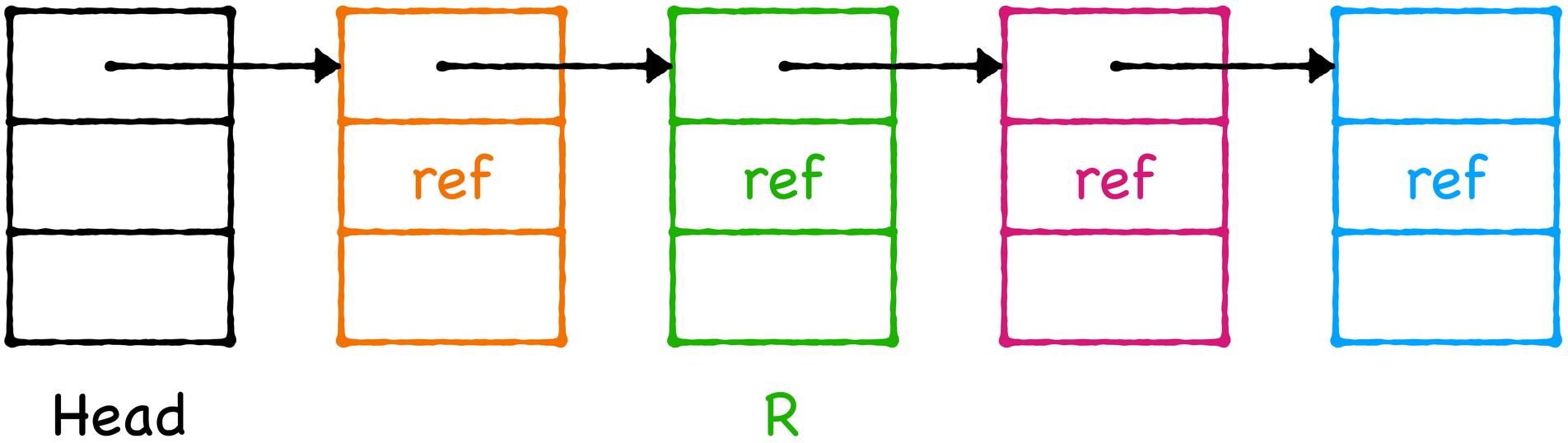
Theorem[Li&Vitanyi'97]: You can win the game for a list of n nodes with m pebbles **only if** $m > \log_2(n)$



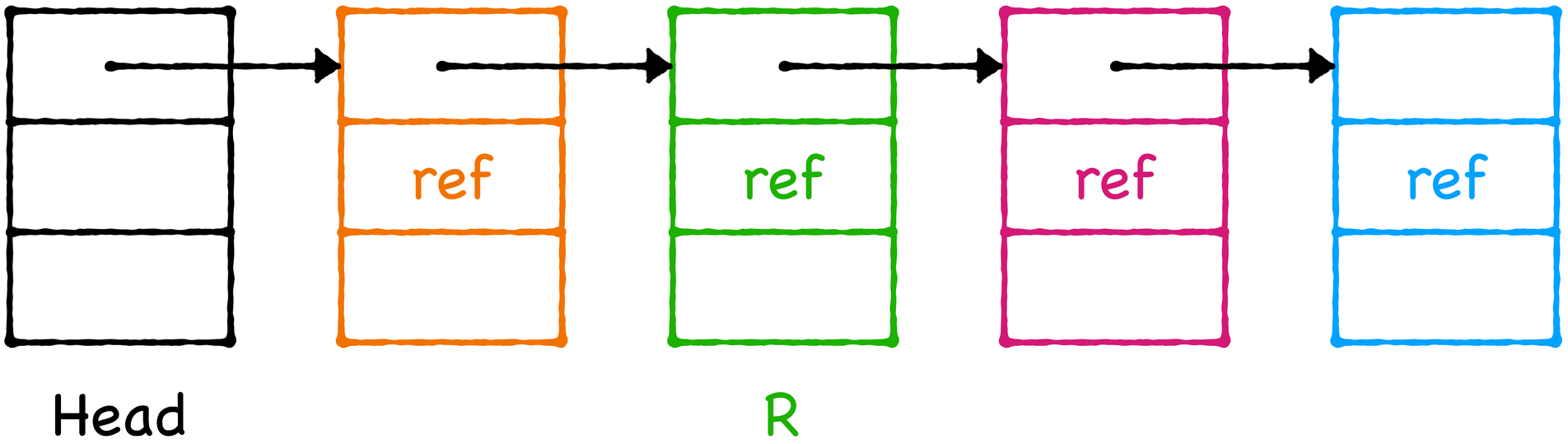
The space complexity is $\Omega(c\log(c))$ where c is the point contention

This is not good enough:
We want O(c) not O(clog(c))!

- Split each reference counter into an acquisition and revocation counter

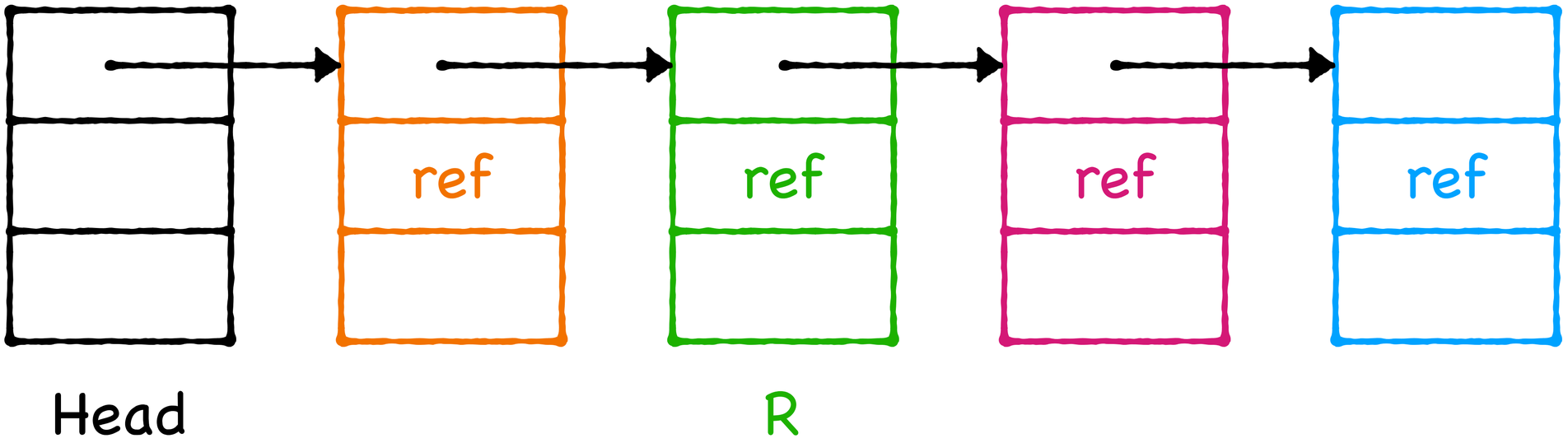


- Split each reference counter into an acquisition and revocation counter



- acquisition: # of processes that have gained the right to access a cell

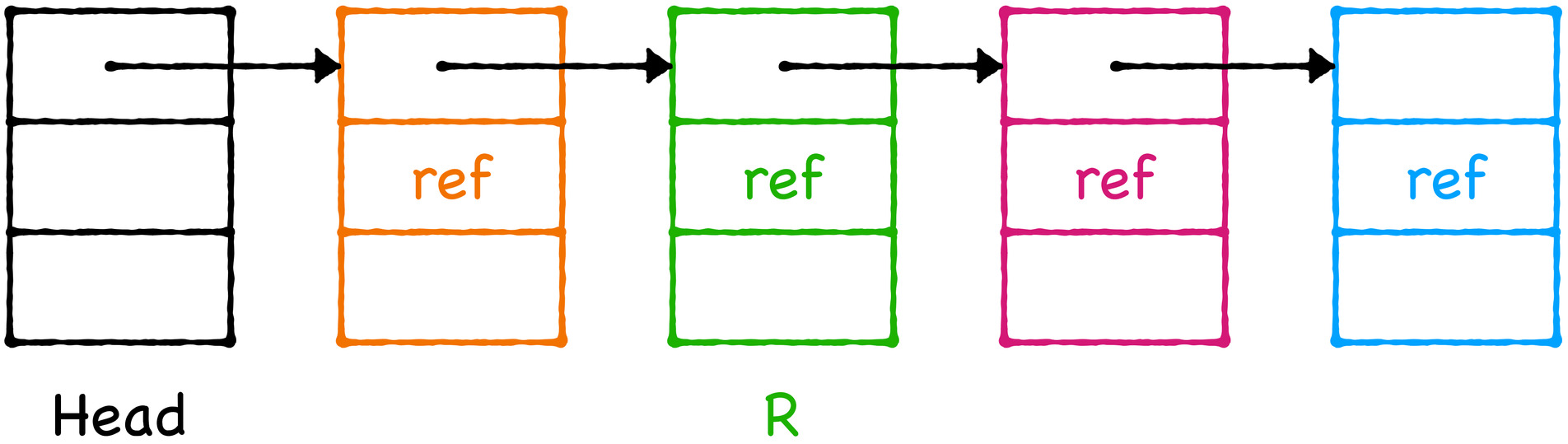
- Split each reference counter into an acquisition and revocation counter



- acquisition: # of processes that have gained the right to access a cell
- revocation: # of processes that have relinquished that right

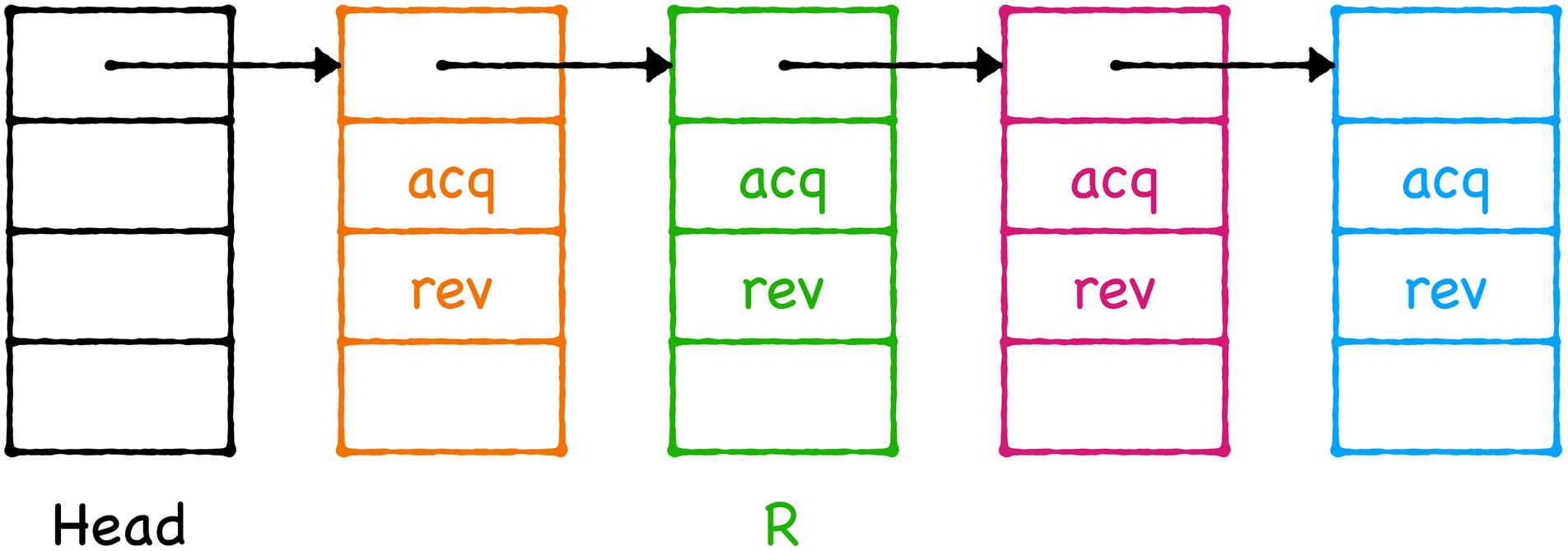
- Split each reference counter into an acquisition and revocation counter

ref = acquisition - revocation

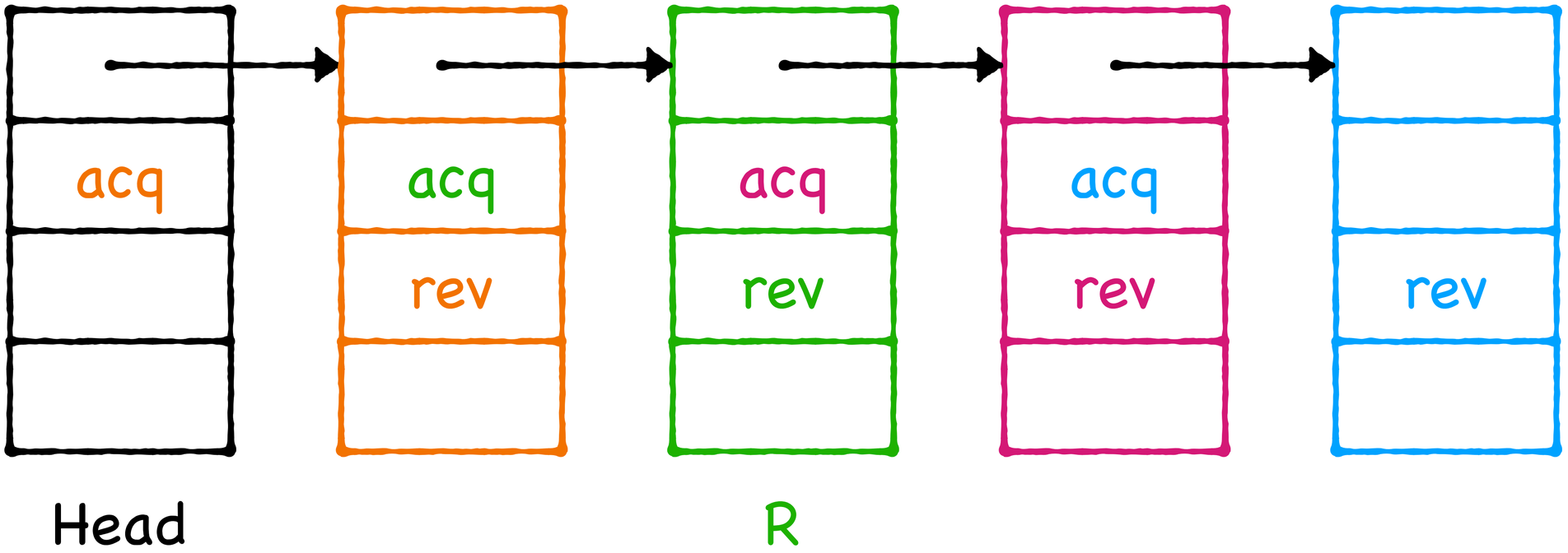


- acquisition: # of processes that have gained the right to access a cell
- revocation: # of processes that have relinquished that right

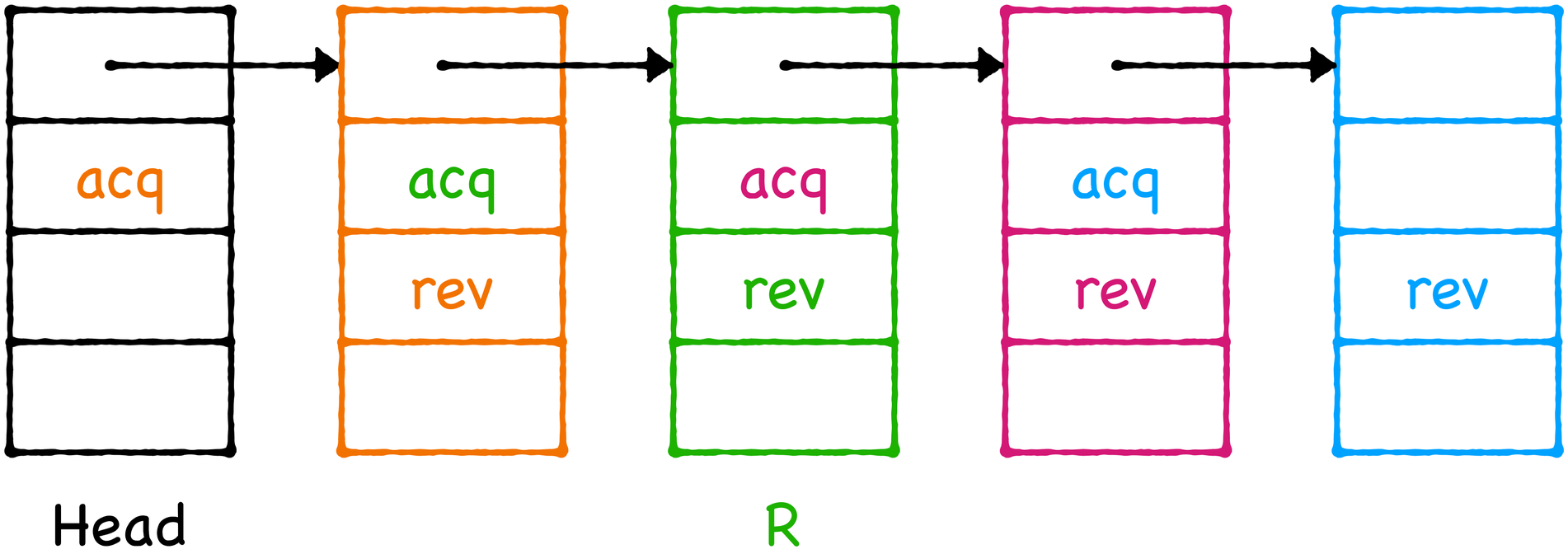
- Split each reference counter into an acquisition and revocation counter



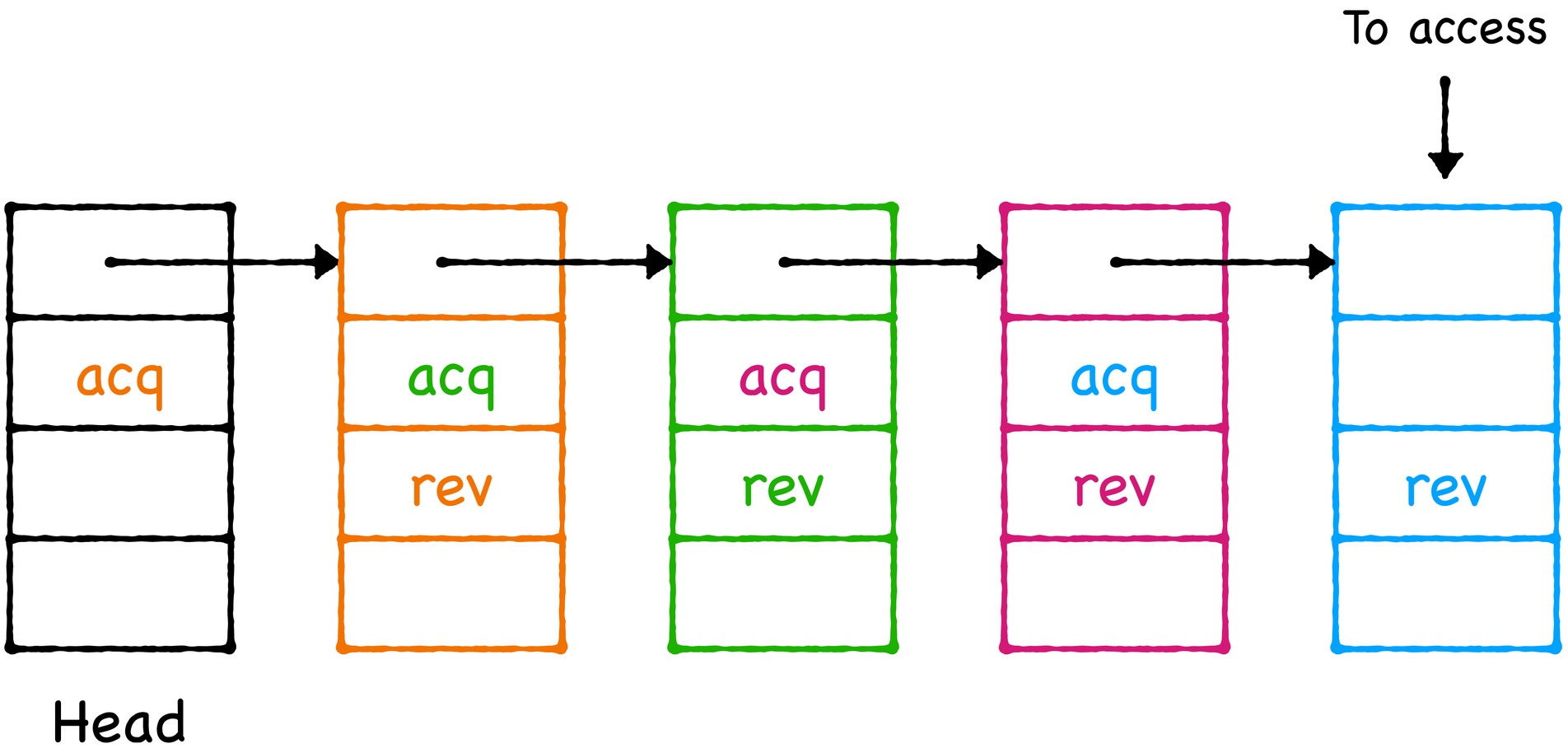
- Split each reference counter into an acquisition and revocation counter
- Put the acquisition counter of each cell into its predecessor

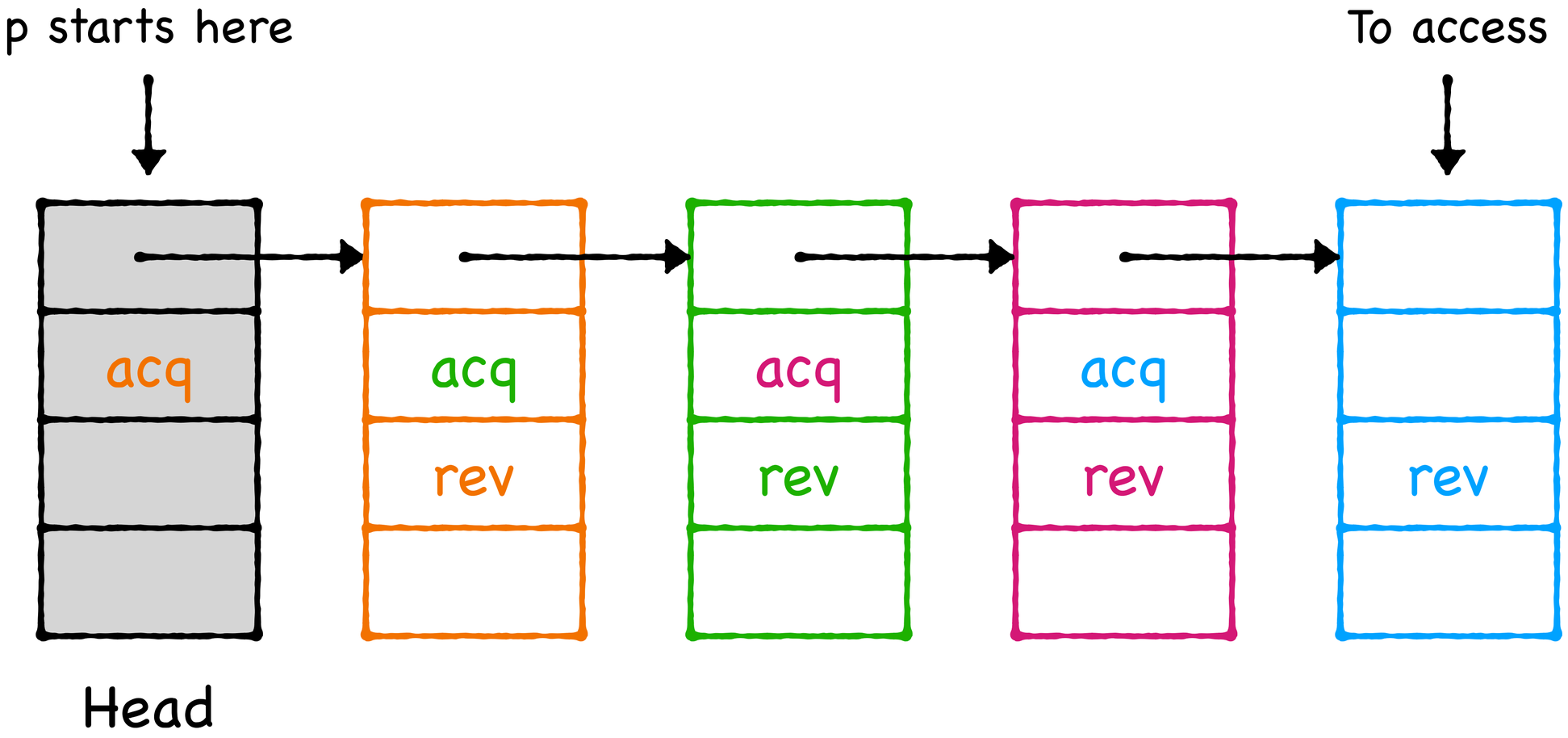


- Split each reference counter into an acquisition and revocation counter
- Put the acquisition counter of each cell into its predecessor

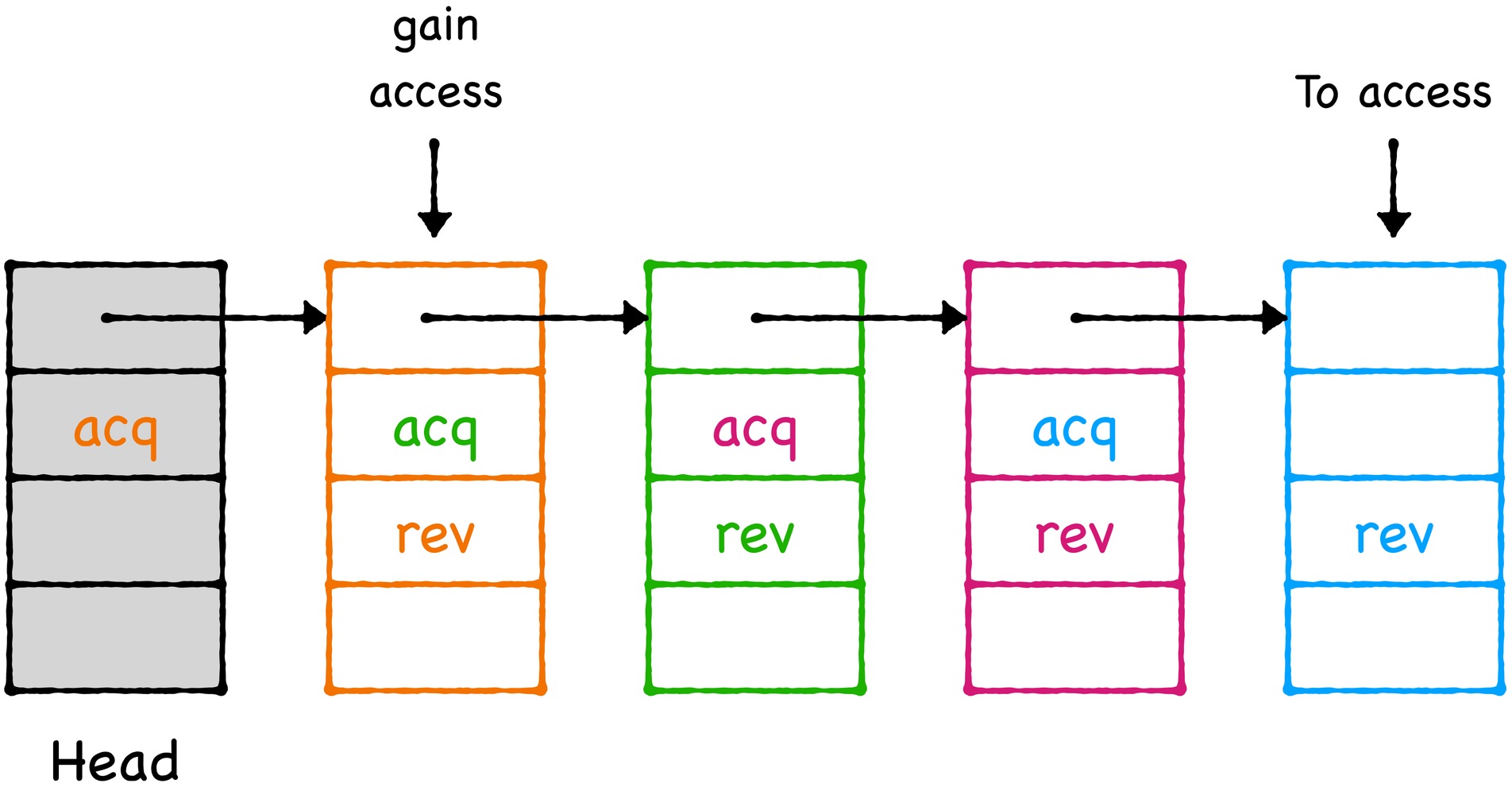


There is no chicken-and-egg issue for relinquishing access to cells!

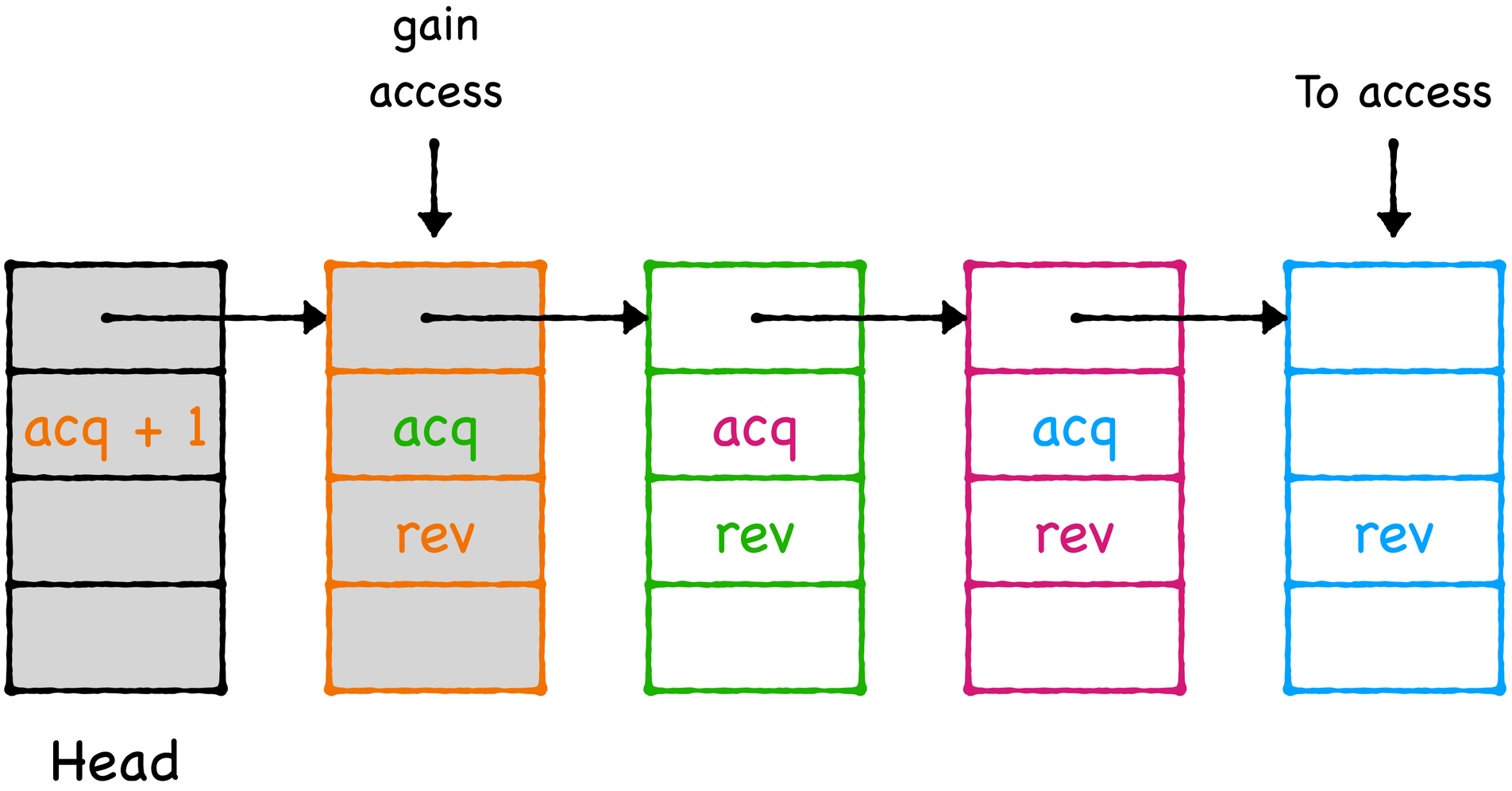

To access
acq
acq
rev
acq
rev
acq
rev
rev
Head



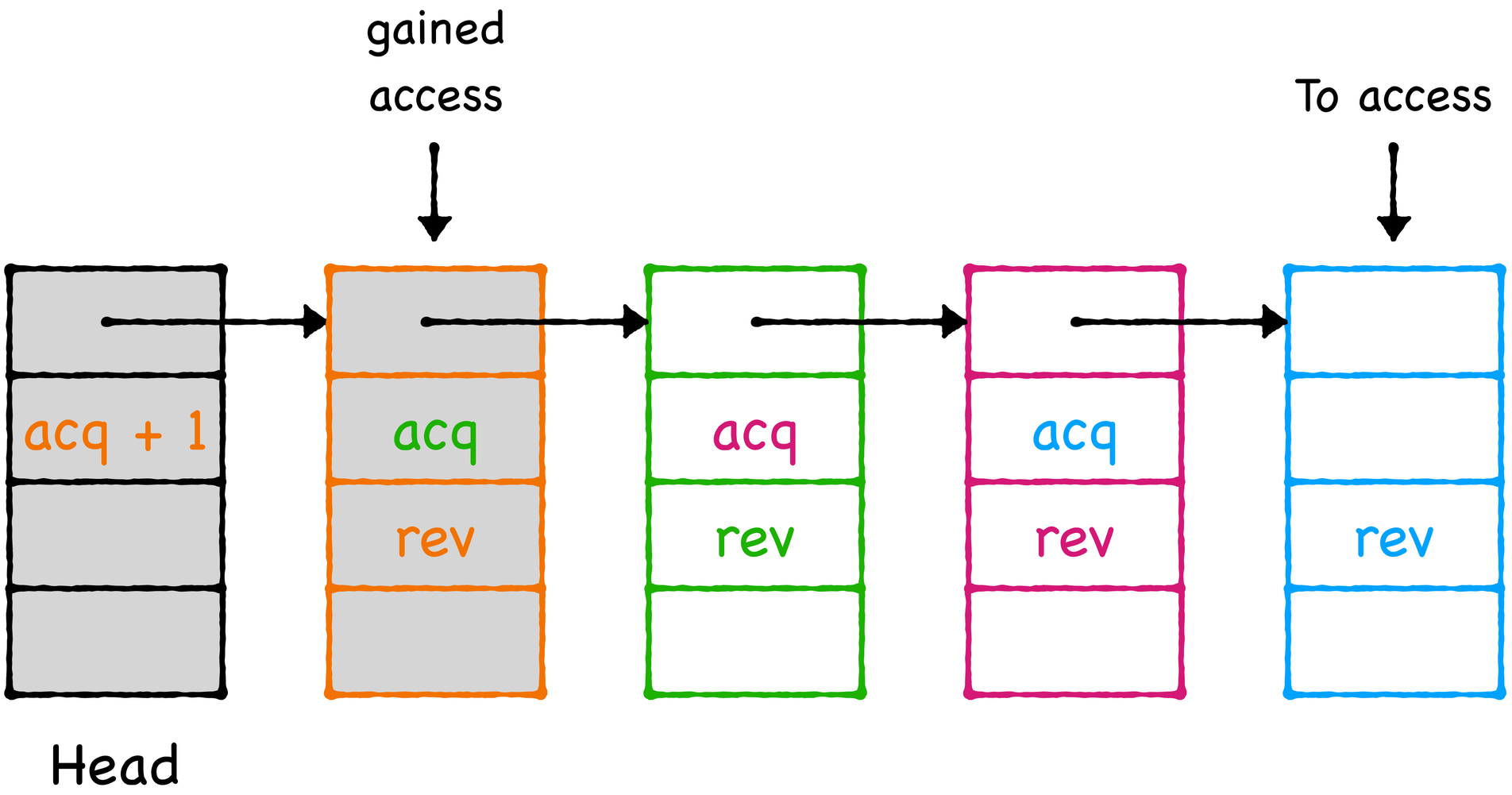
# of acquired cells: 1



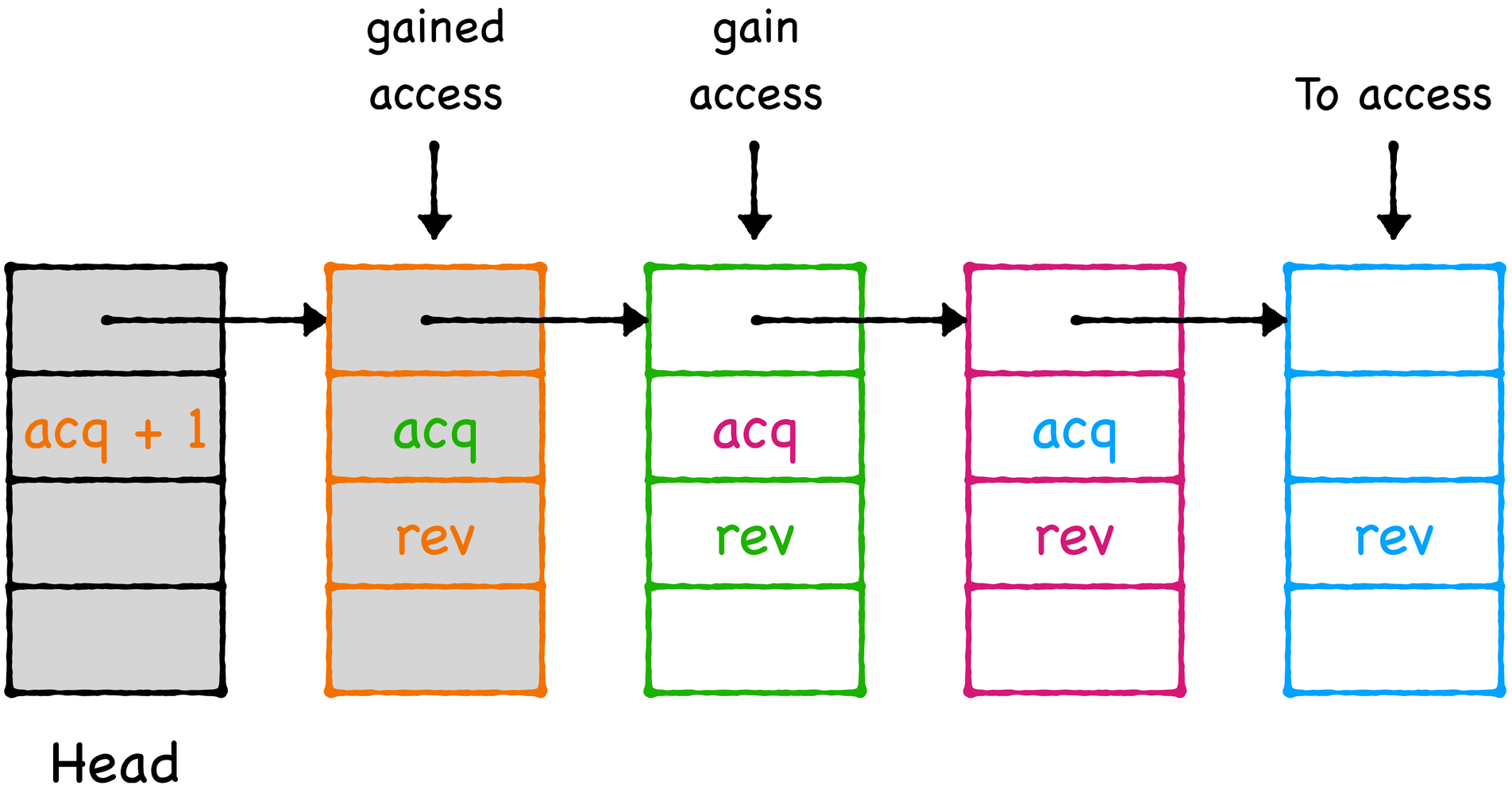
# of acquired cells: 1



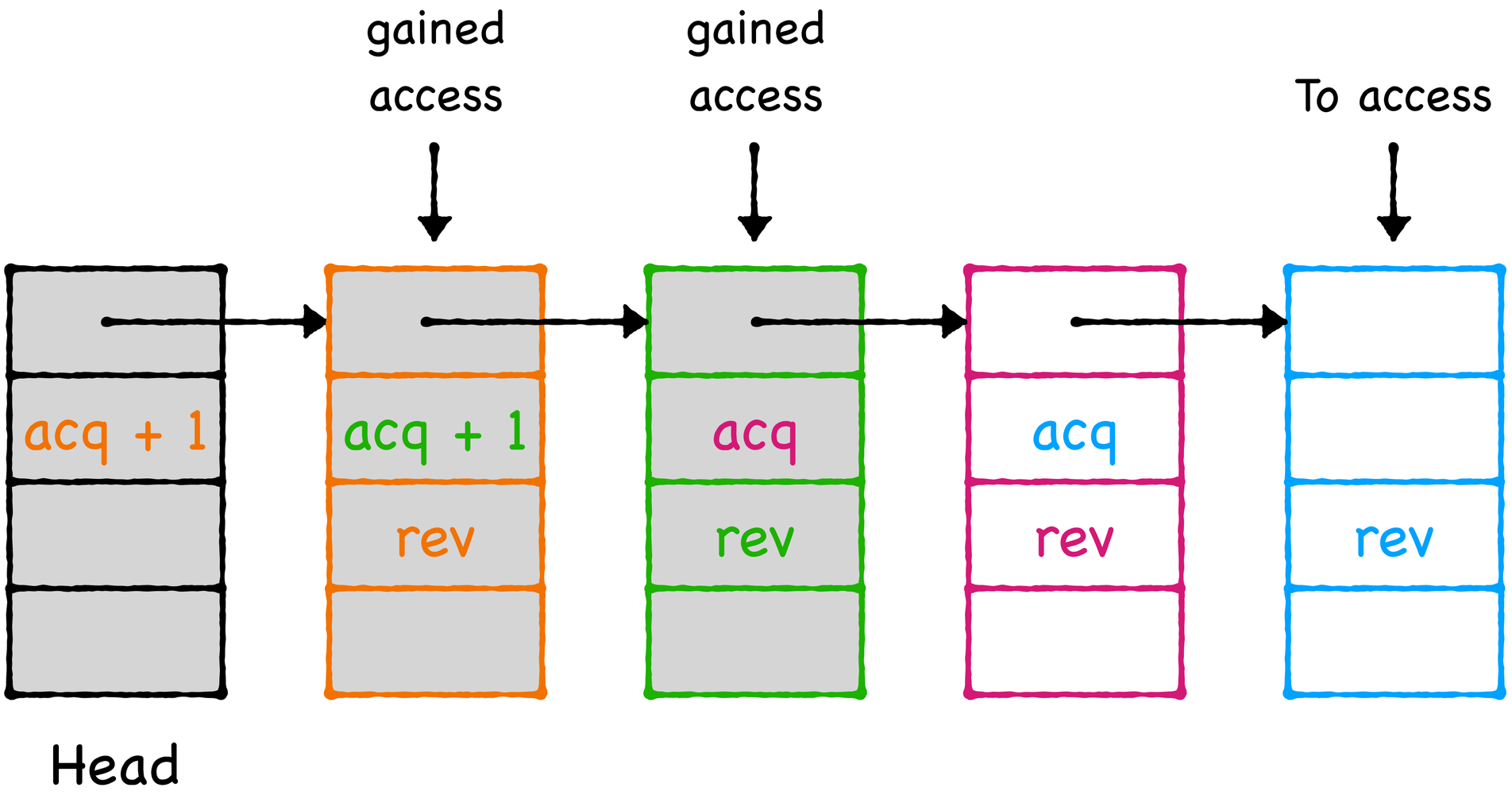
# of acquired cells: 2



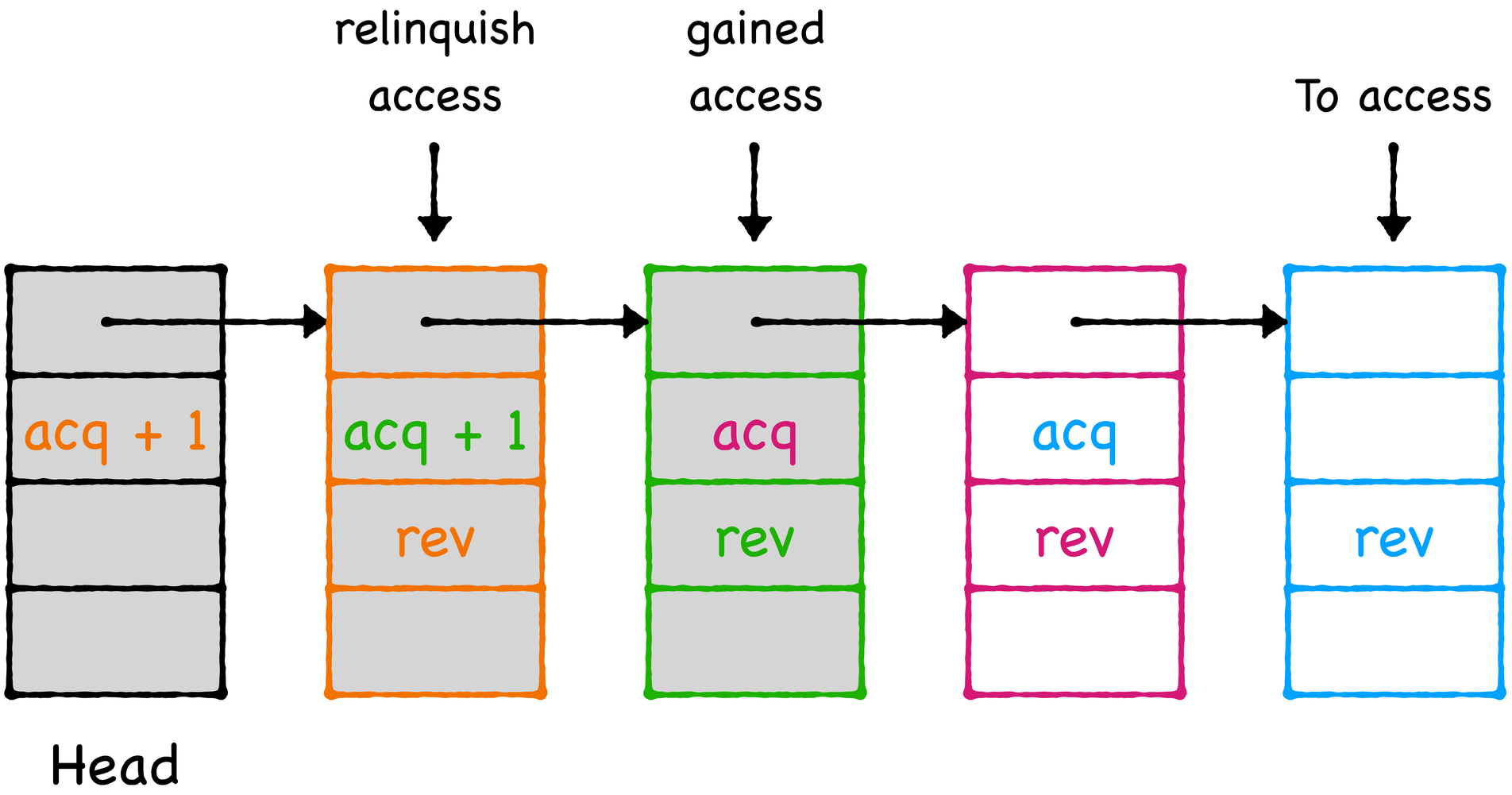
# of acquired cells: 2



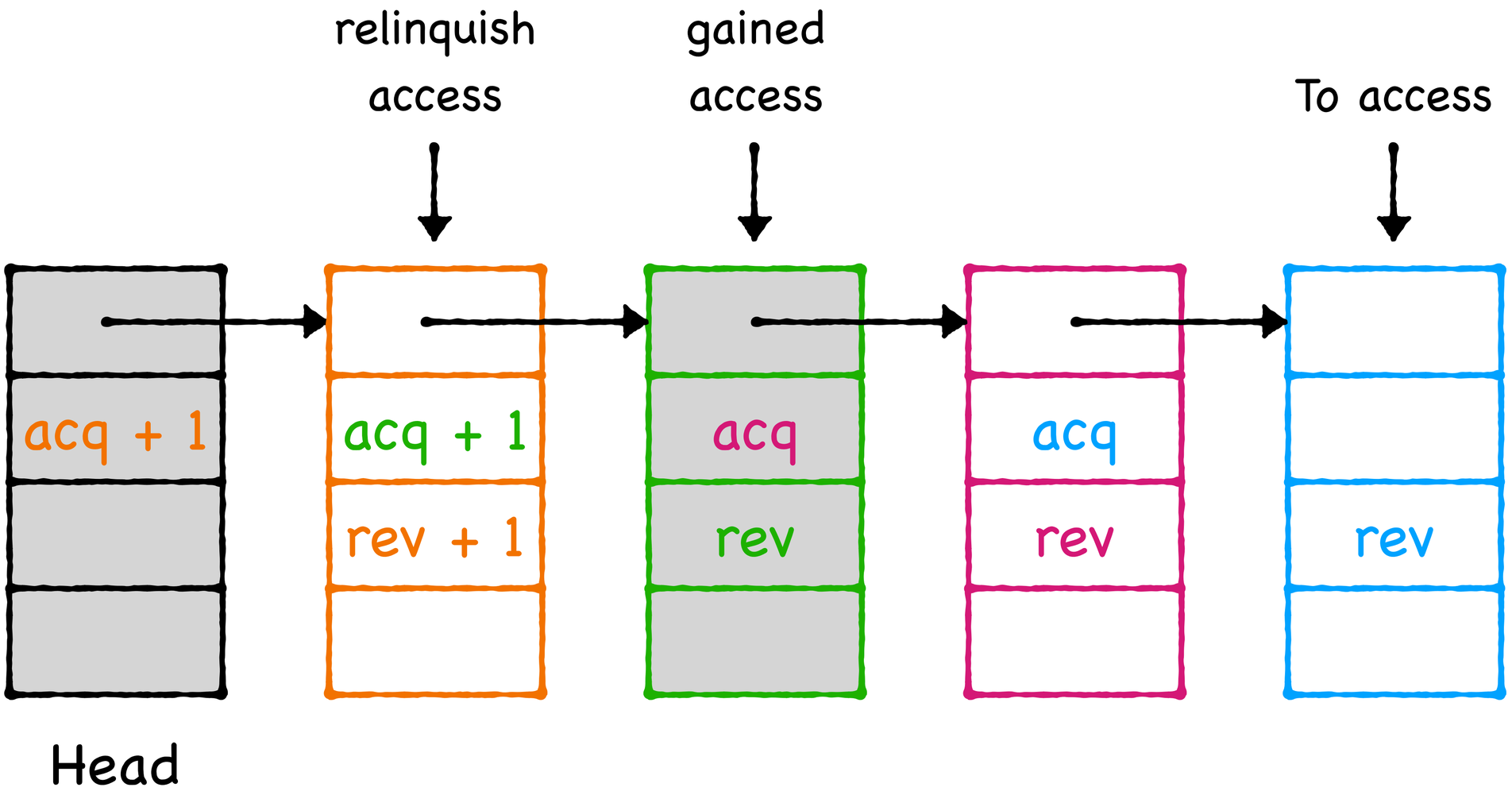
# of acquired cells: 2



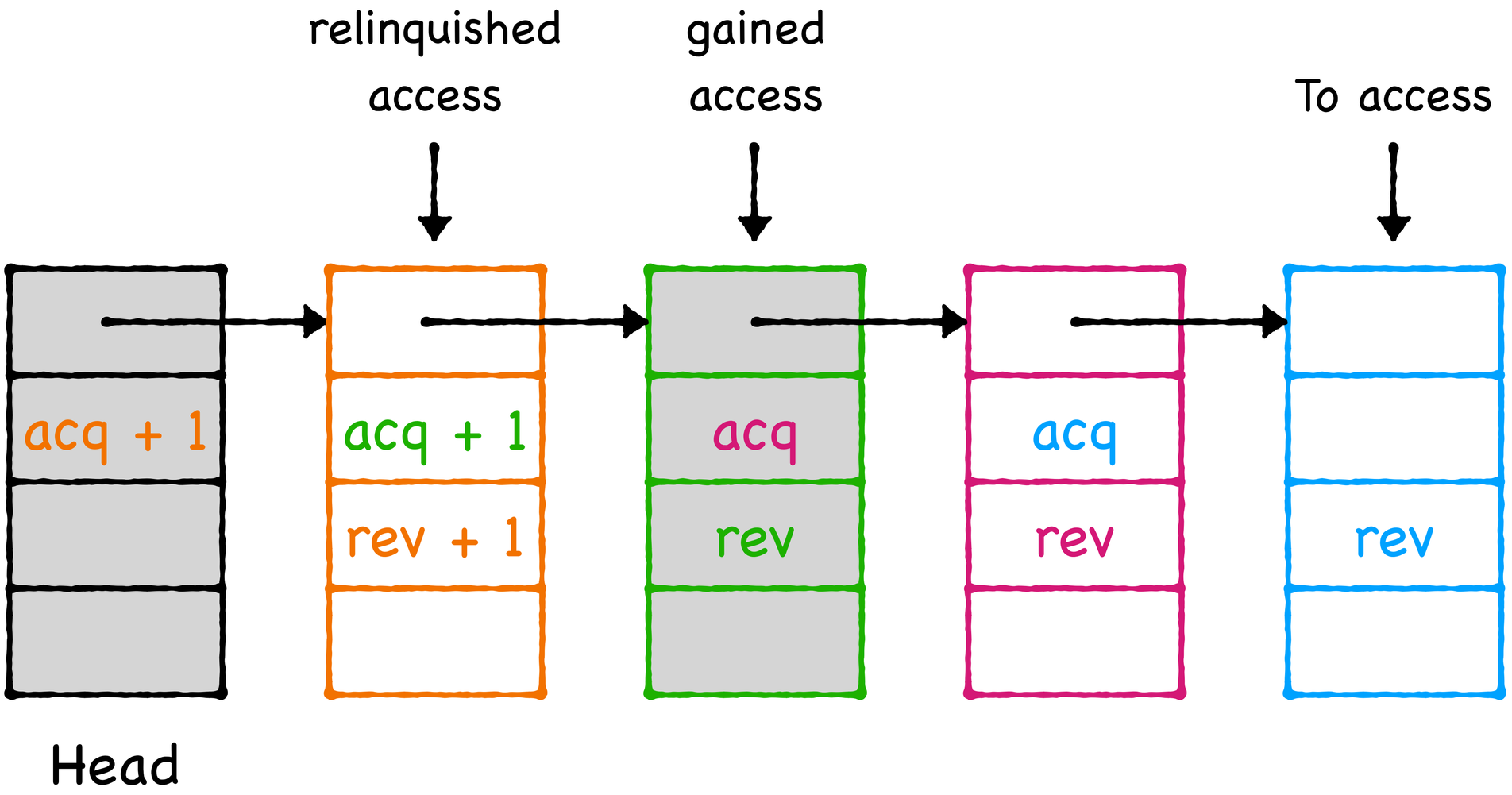
# of acquired cells: 3



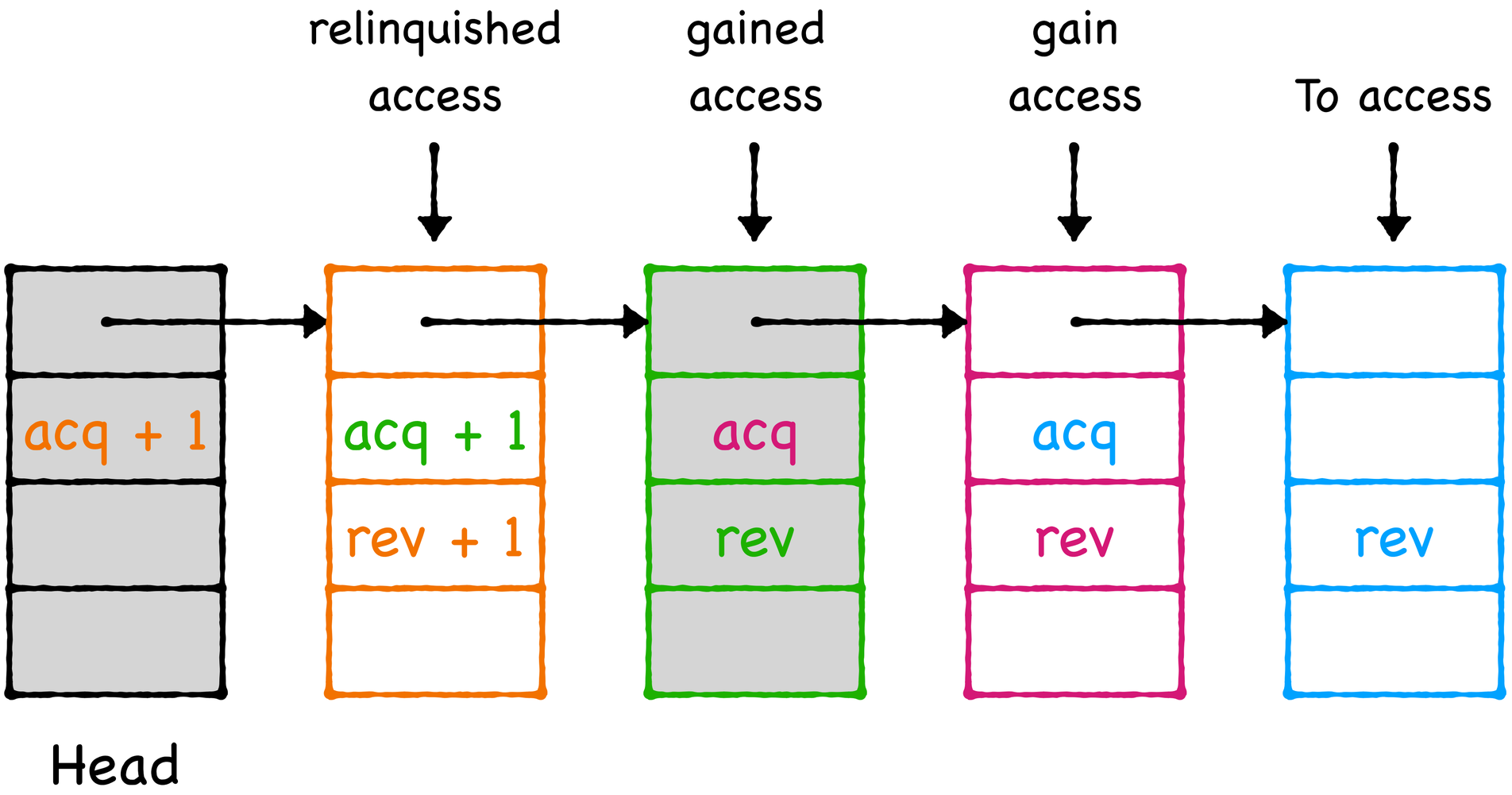
# of acquired cells: 3



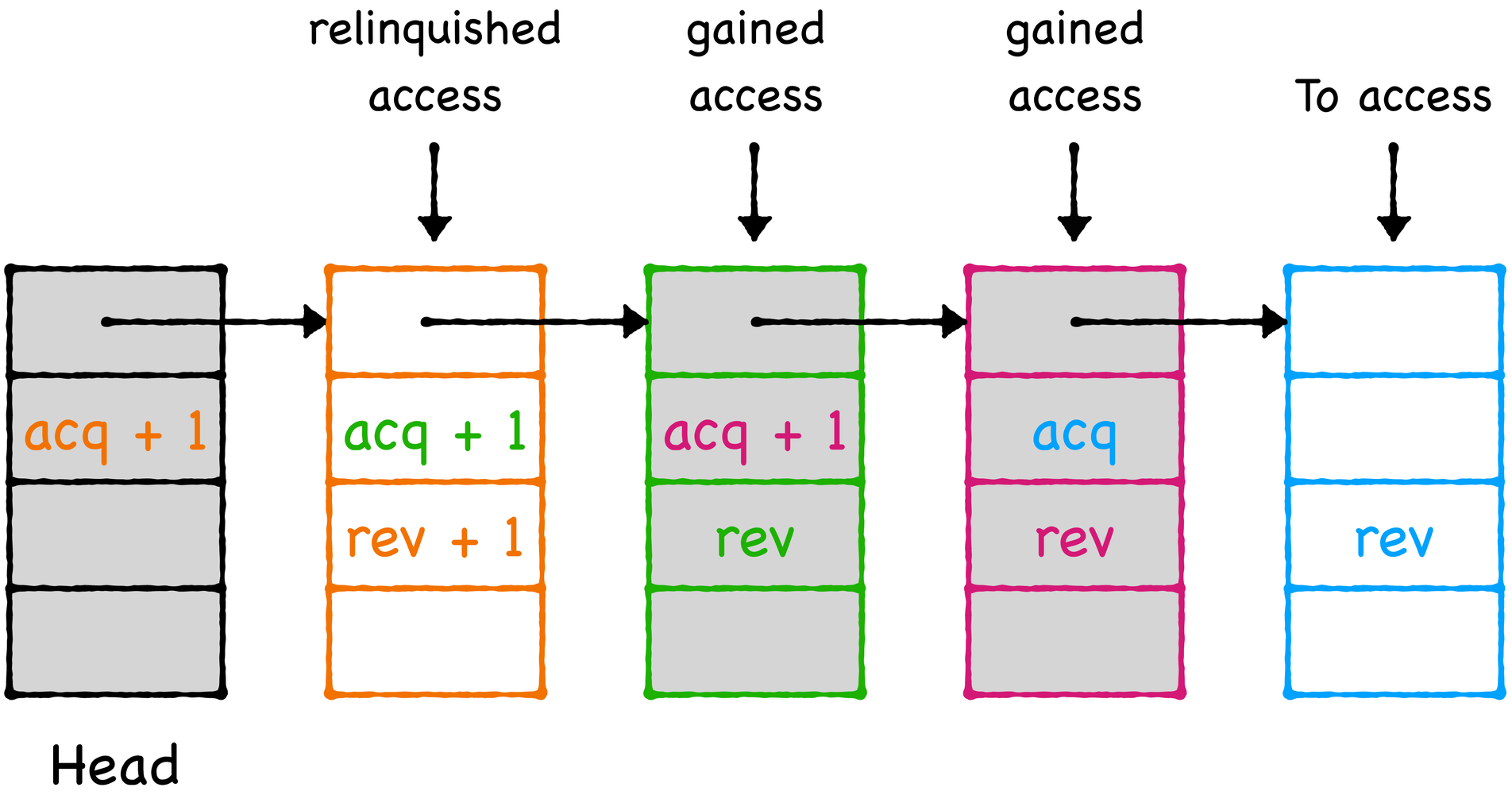
# of acquired cells: 2



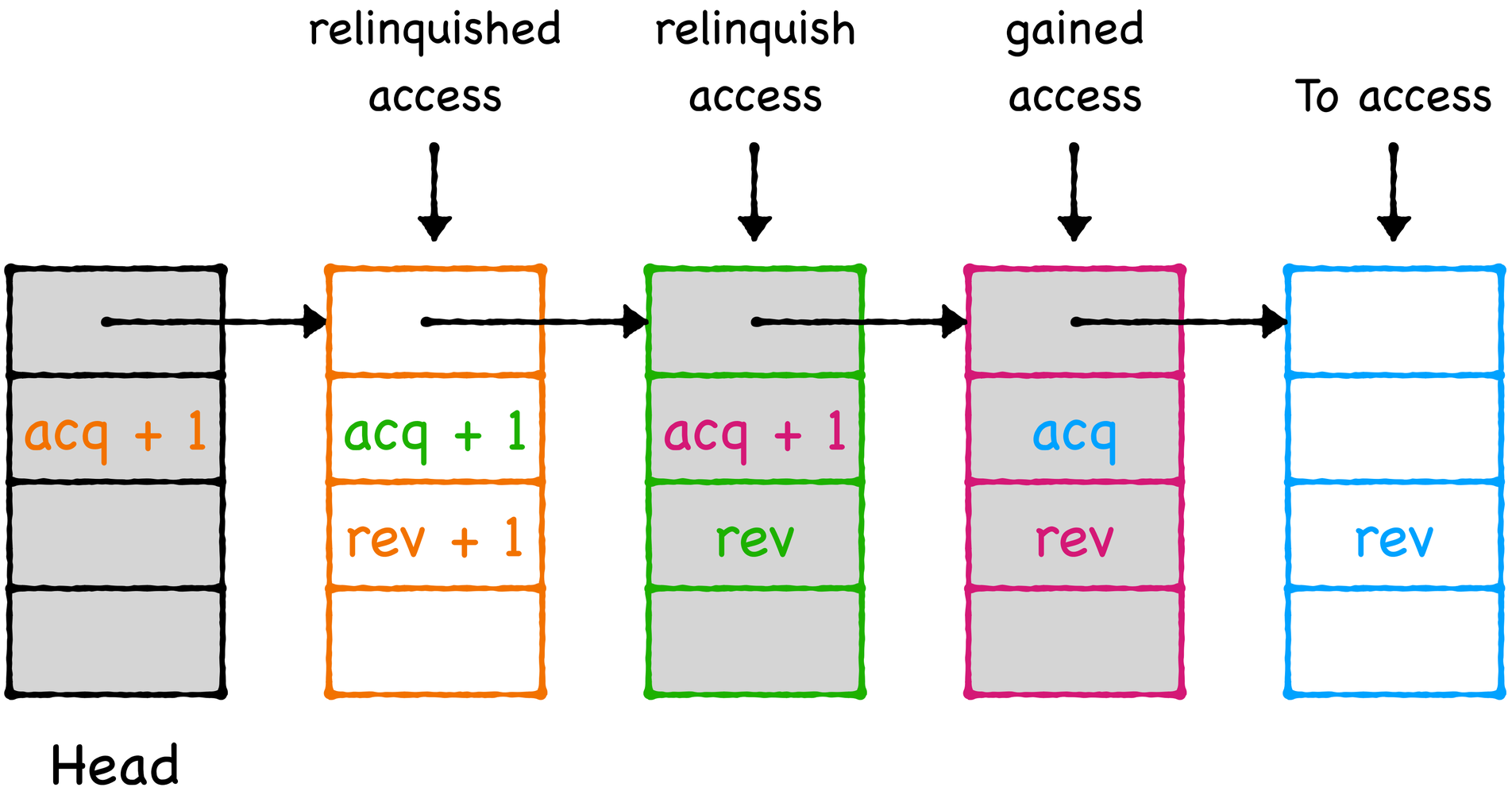
# of acquired cells: 2



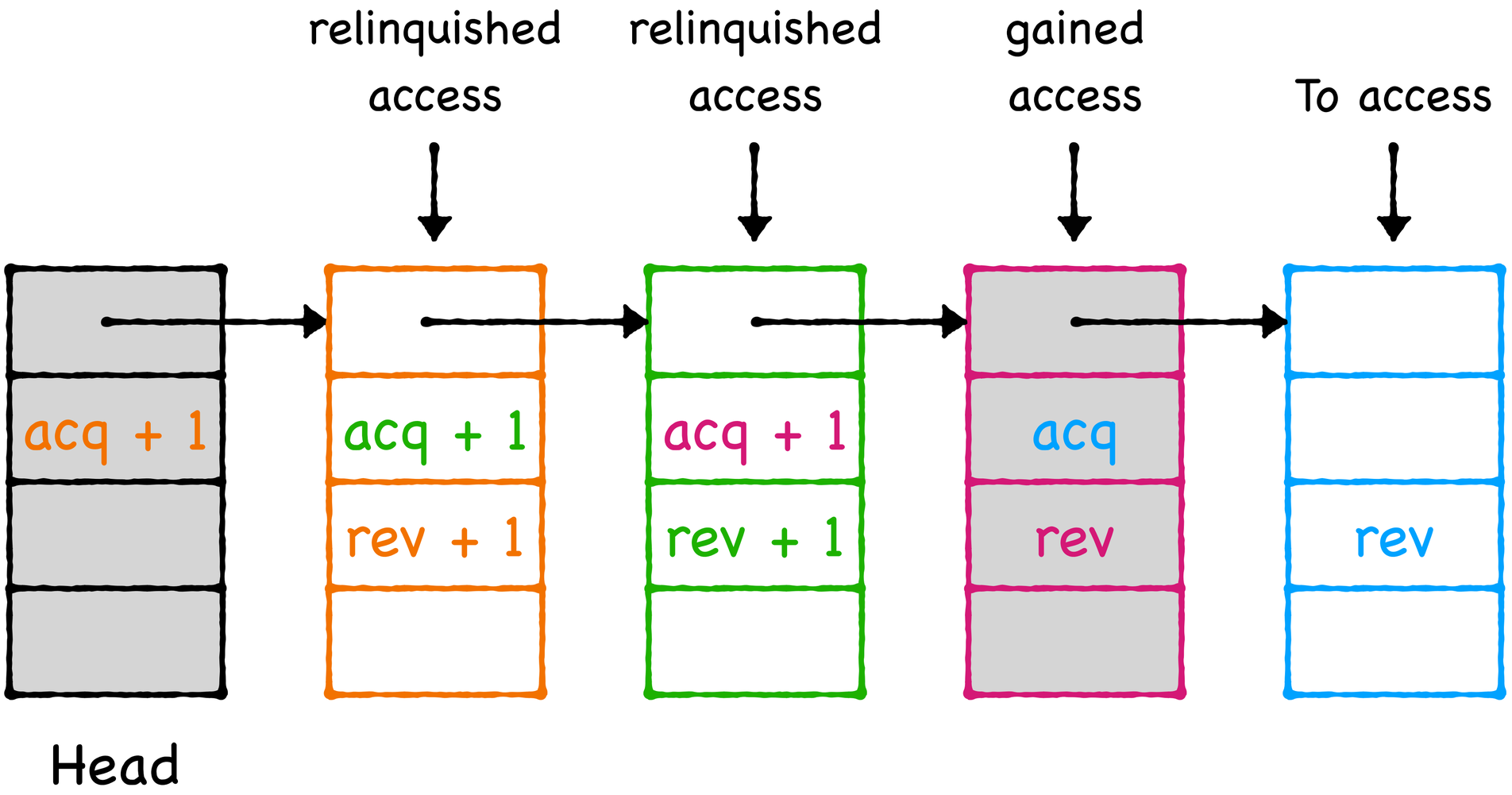
# of acquired cells: 2



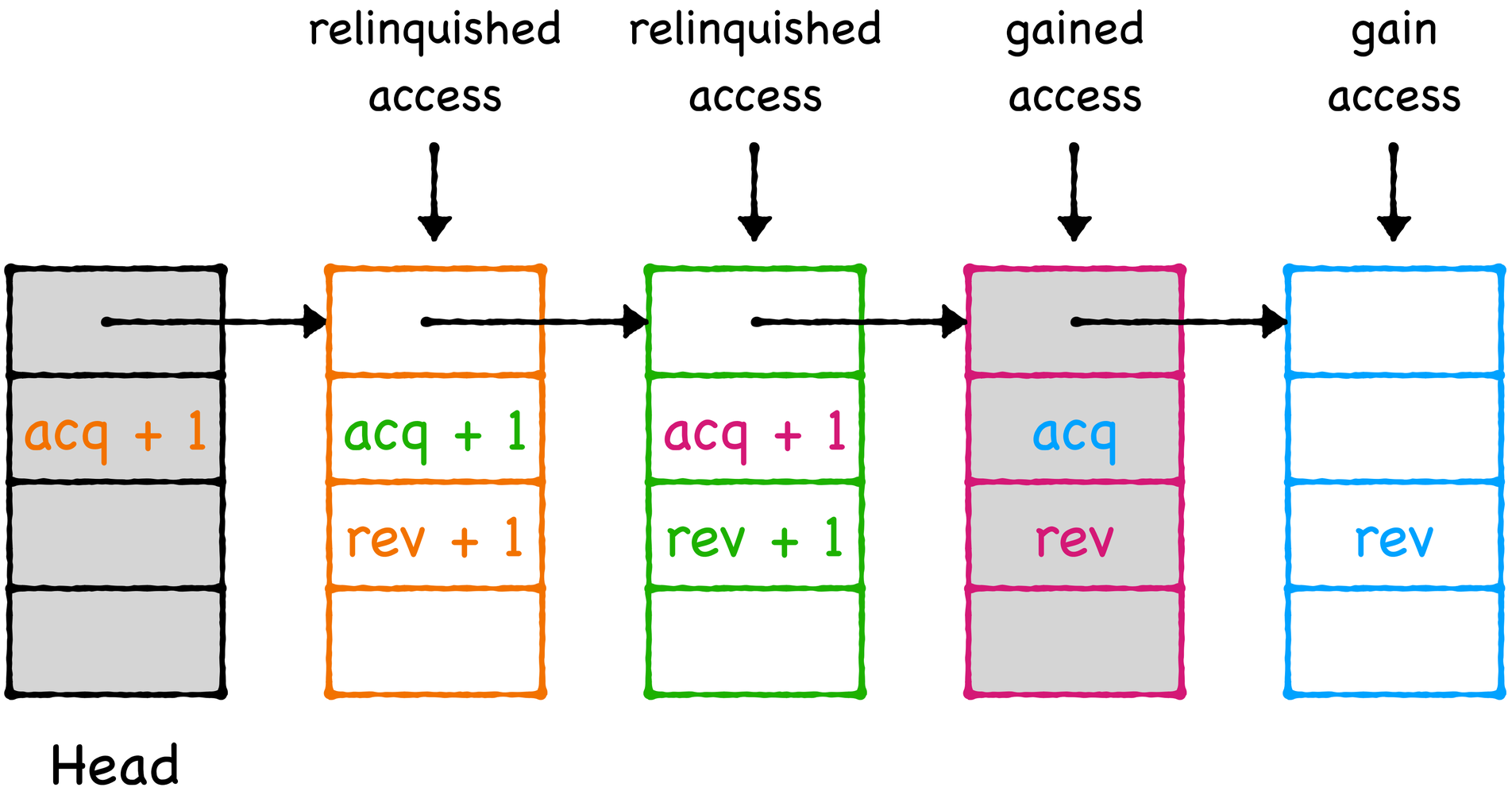
# of acquired cells: 3



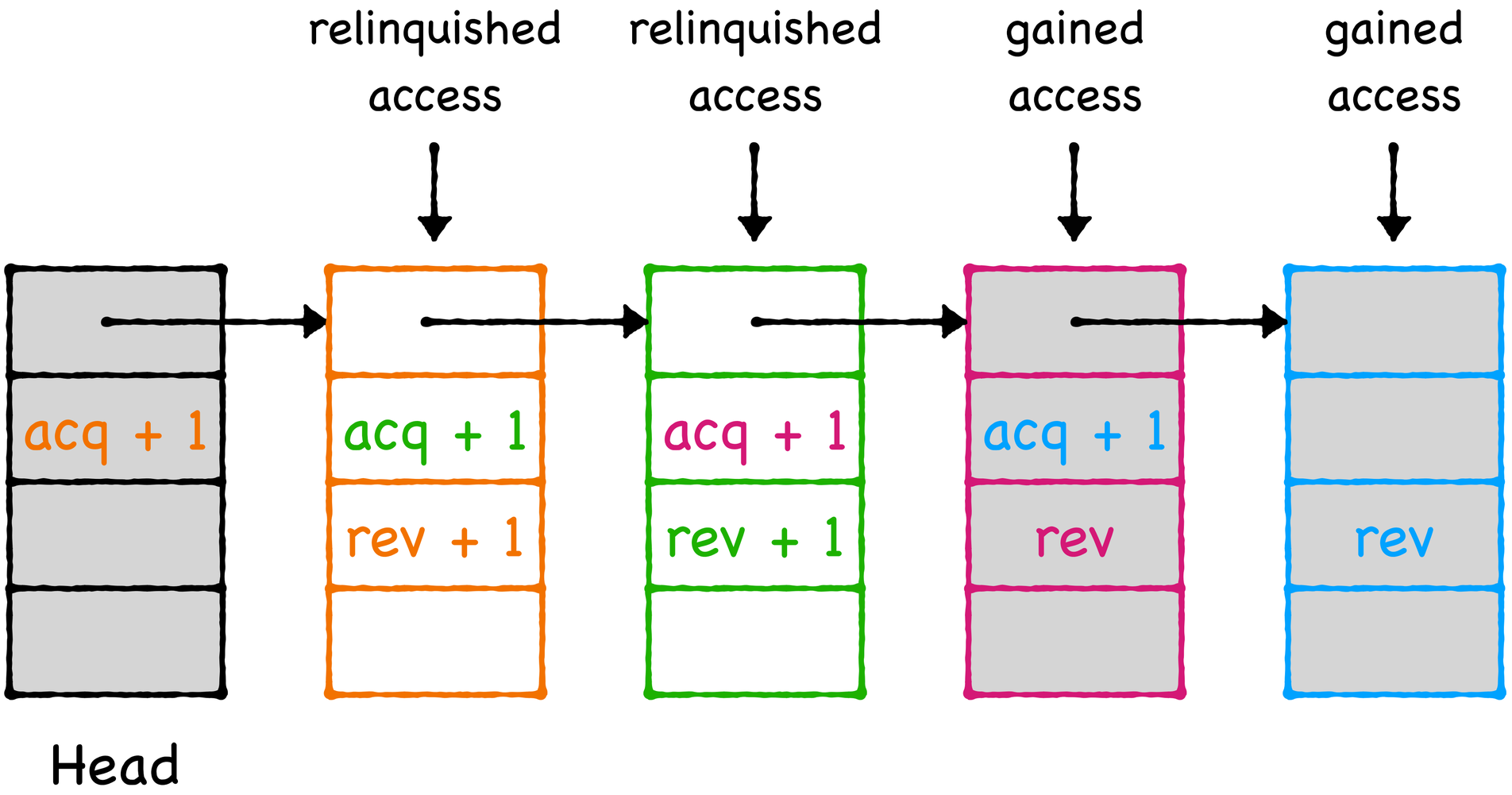
# of acquired cells: 3



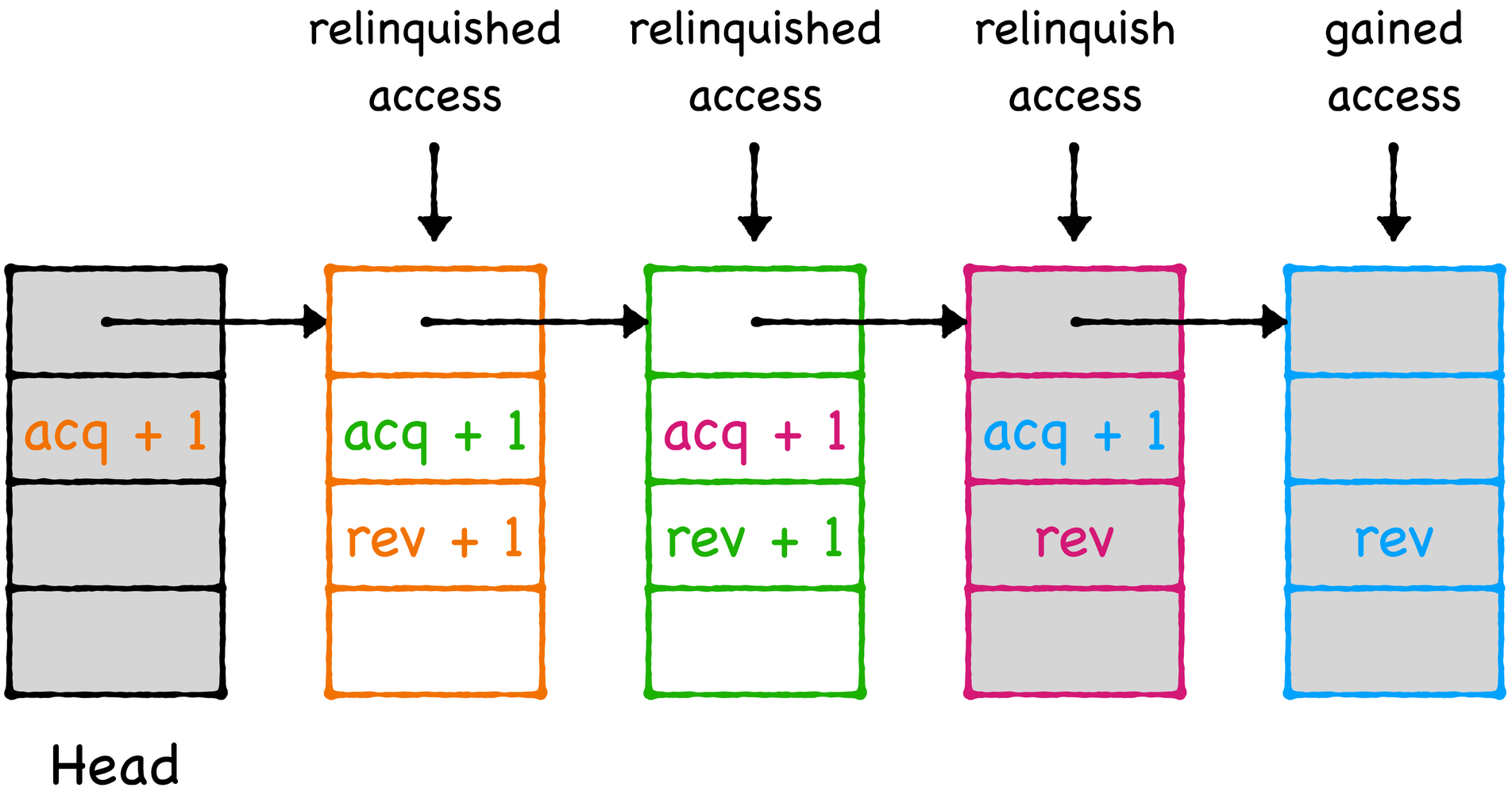
# of acquired cells: 2



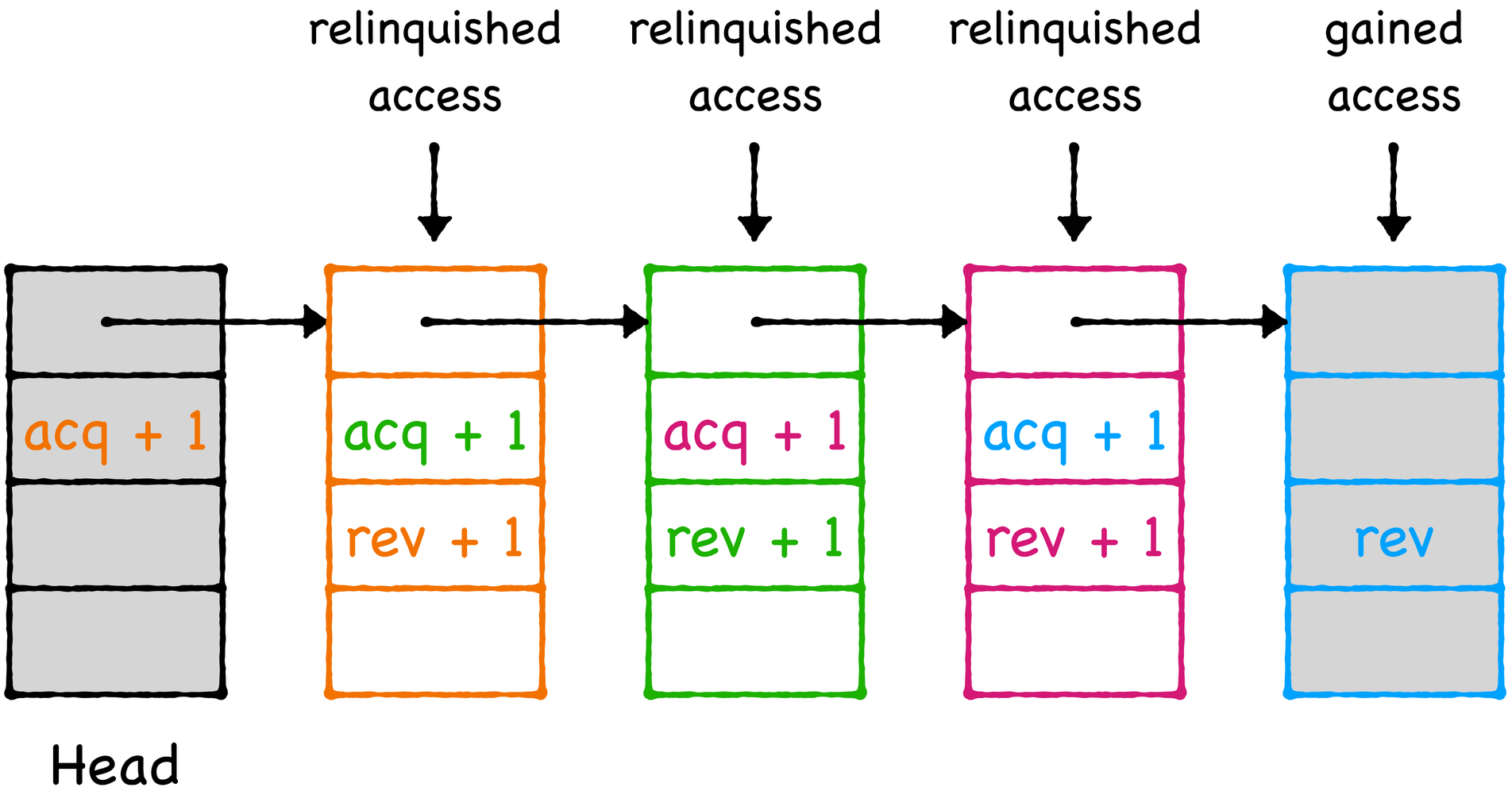
# of acquired cells: 2



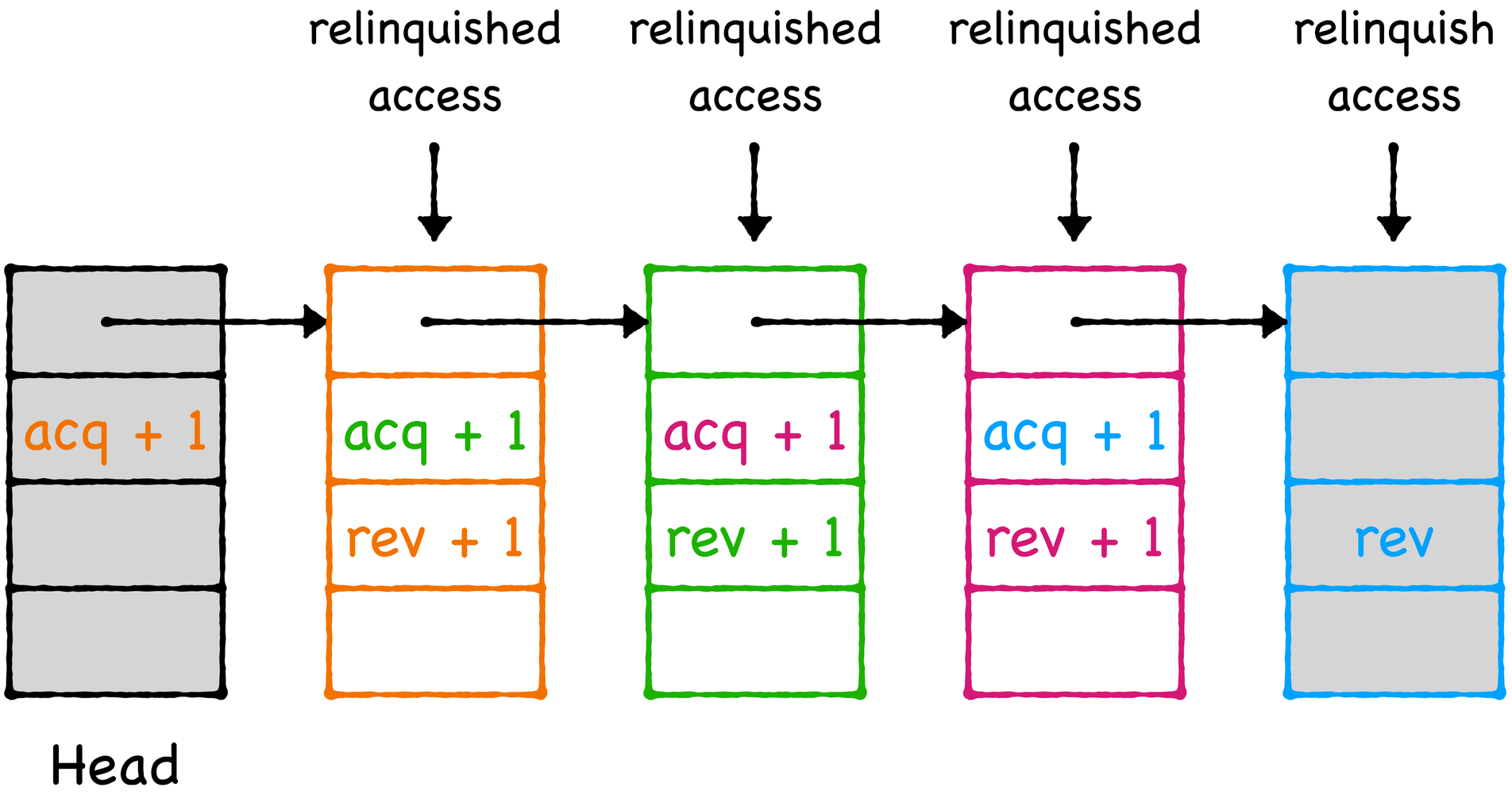
# of acquired cells: 3



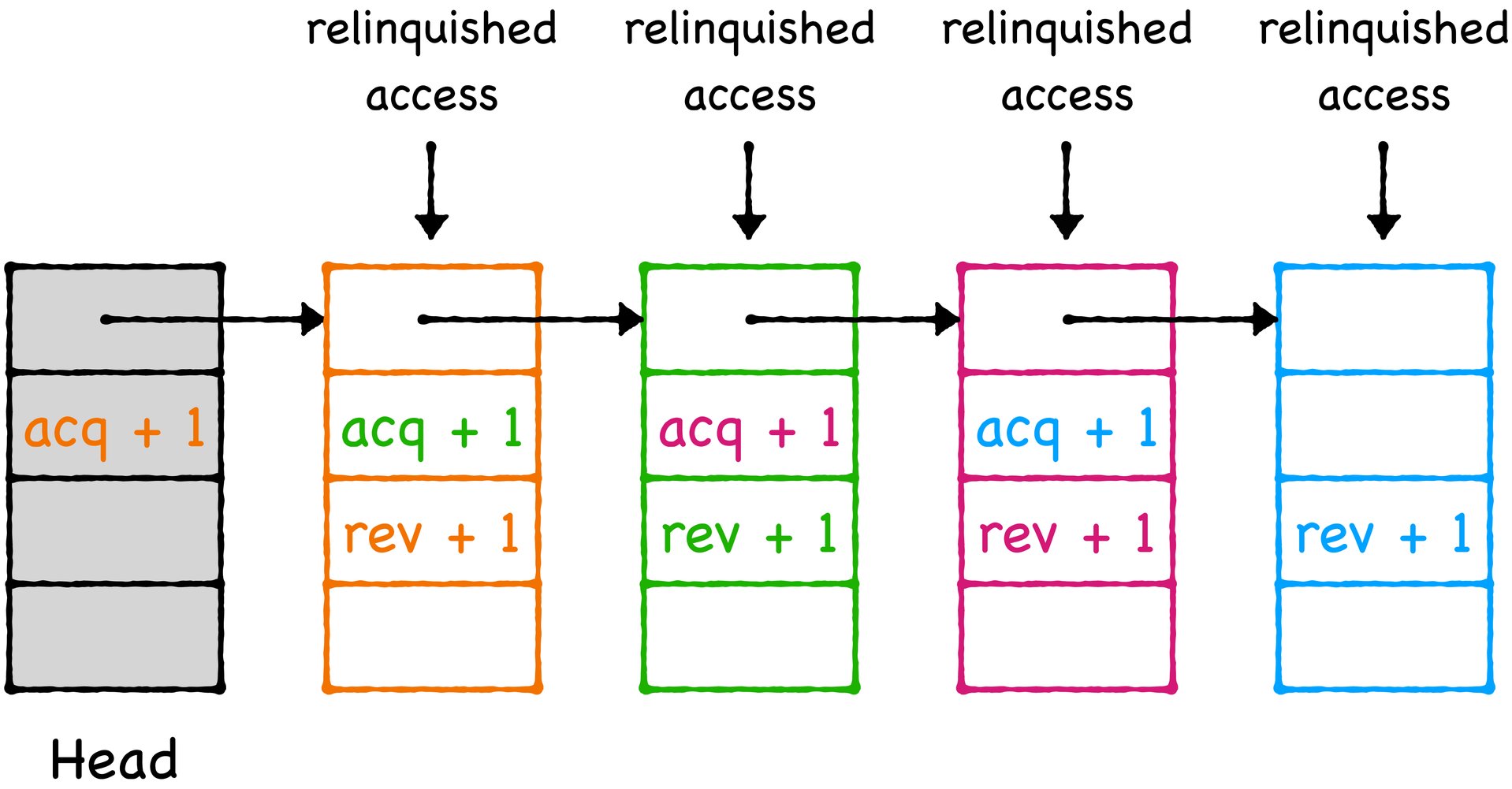
# of acquired cells: 3



# of acquired cells: 2

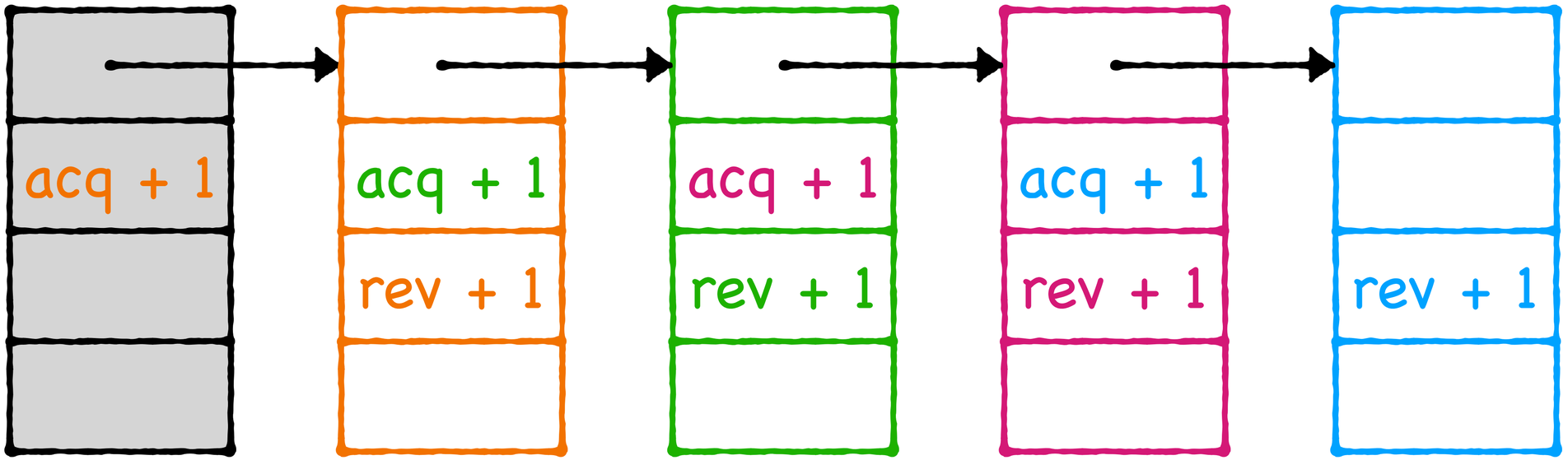


# of acquired cells: 2

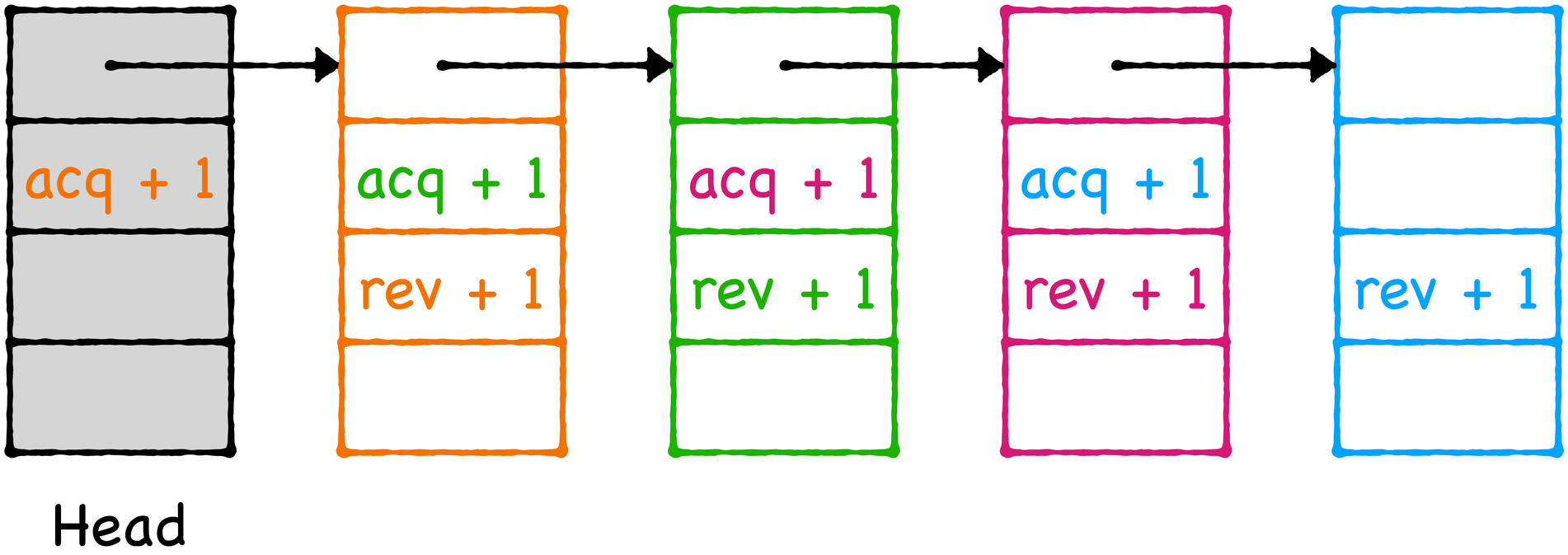


# of acquired cells: 1

We can gain access to any cell in any list while having access to at most three cells simultaneously!



Space complexity O(c), i.e, linear in the point contention c



Recycling Scheme — Core Ideas

Recycling Scheme — Core Ideas

- Response cells are recycled

Recycling Scheme — Core Ideas

- Response cells are recycled
- Current cells are threaded into a list, and

Recycling Scheme — Core Ideas

- Response cells are recycled
- Current cells are threaded into a list, and
- Each reference counter is decomposed into

Recycling Scheme — Core Ideas

- Response cells are recycled
- Current cells are threaded into a list, and
- Each reference counter is decomposed into
  - an acquisition and revocation counter, and

Recycling Scheme — Core Ideas

- Response cells are recycled
- Current cells are threaded into a list, and
- Each reference counter is decomposed into
  - an acquisition and revocation counter, and
  - the acquisition counter is shifted to the left

Recycling Scheme — Core Ideas

- Response cells are recycled
- Current cells are threaded into a list, and
- Each reference counter is decomposed into
  - an acquisition and revocation counter, and
  - the acquisition counter is shifted to the left
- But how do we compute ref = acq - rev atomically?

Our Second Universal Construction

Our Second Universal Construction

- We merged our recycling scheme with our first construction

Our Second Universal Construction

- We merged our recycling scheme with our first construction

- This merging was not easy!

Our Second Universal Construction

- We merged our recycling scheme with our first construction
- This merging was not easy!
- In the paper we explain the challenges and how we solved them

Summary

Summary

(1) GCAS: A simple extension of CAS that replaces the equality test with a parametrized comparator in {<, =, >}

Summary

(1) GCAS: A simple extension of CAS that replaces the equality test with a parametrized comparator in {<, =, >}

(2) Two space-efficient universal constructions for infinite-arrival model:

(a) Space linear in the # of processes that have participated so far

(b) Space linear in the point contention, but assumes bounded concurrency

Summary

(1) GCAS: A simple extension of CAS that replaces the equality test with a parametrized comparator in {<, =, >}

(2) Two space-efficient universal constructions for infinite-arrival model:

(a) Space linear in the # of processes that have participated so far
Time linear in the point contention

(b) Space linear in the point contention, but assumes bounded concurrency

Summary

(1) GCAS: A simple extension of CAS that replaces the equality test with a parametrized comparator in {<, =, >}

(2) Two space-efficient universal constructions for infinite-arrival model:

(a) Space linear in the # of processes that have participated so far
Time linear in the point contention

(b) Space linear in the point contention, but assumes bounded concurrency

(3) A novel memory recycling scheme that may be of more general use

Open Questions

Open Questions

(1) Is GCAS necessary to achieve the space complexity of our universal constructions?

Open Questions

(1) Is GCAS necessary to achieve the space complexity of our universal constructions?

(2) Can one achieve the space complexity of our second universal construction **without assuming bounded concurrency?**

Open Questions

(1) Is GCAS necessary to achieve the space complexity of our universal constructions?

(2) Can one achieve the space complexity of our second universal construction **without assuming bounded concurrency?**

(3) Are there other uses for our memory recycling scheme? In particular, the splitting of the reference counter?

Open Questions

(1) Is GCAS necessary to achieve the space complexity of our universal constructions?

(2) Can one achieve the space complexity of our second universal construction **without assuming bounded concurrency?**

(3) Are there other uses for our memory recycling scheme? In particular, the splitting of the reference counter?

(4) What other applications can benefit from GCAS?

Questions?