The dynamic multithreading model (part 2)

CSE 6230, Fall 2014

August 28

Georgia Tech College of Computing
Computational Science and Engineering

```
function Y ← qsort(X) // |X| = n
if |X| ≤ 1 then
    return Y ← X
else
    X_L, Y_M, X_R ← partition-seq(X) // Pivot
    Y_L ← spawn qsort(X_L)
    Y_R ← qsort(X_R)
    sync
    return Y ← Y_L ∪ Y_M ∪ Y_R
endif
```

Recall: Quicksort

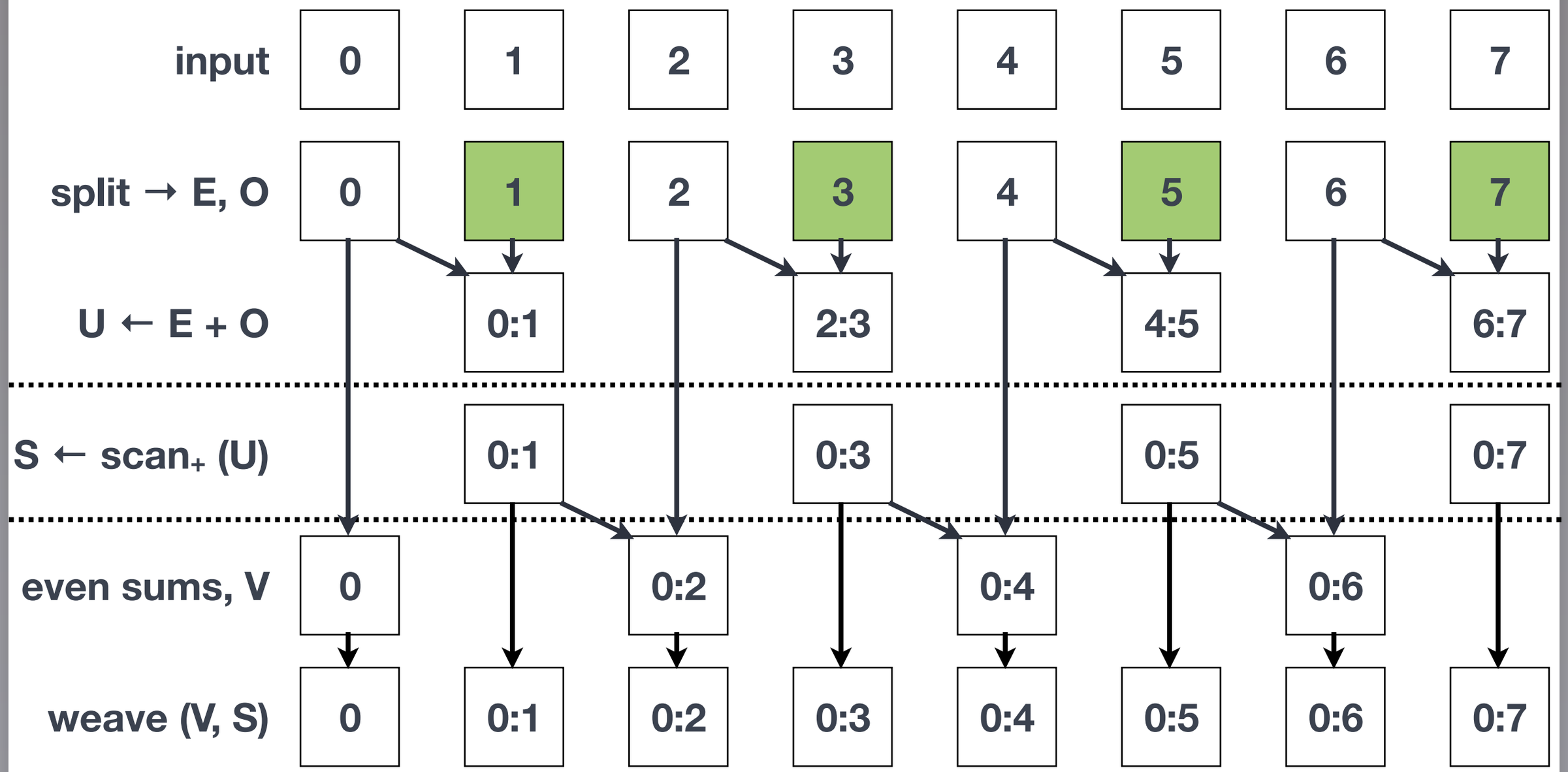
```
parallel-for i ← 1 to n do
    f(i)
```

Work and span?

Recall: Parallel loops

```
{"Add-scan" or "prefix sum" operation}
for i ← 1 to n − 1 do
    A[i] ← A[i − 1] + A[i]
end for
```

Compute: **scan$_+$** (x) = [**x[0]**, **x[0]+x[1]**, **x[0]+x[1]+x[2]**, …]



Scans in parallel

Seemingly sequential computations may have parallel algorithms.

What did we assume? What are *W*, *D*?

```
function y ← scan_+(x) // Pre: |x| is power of 2
if |x| ≥ 2 then
    x_e, x_o ← evens(x), odds(x)
    u ← x_e + x_o
    y_o ← scan_+(u)
    y_e ← y_o − x_o
    return y ← weave(y_e, y_o)
else
    return y ← x
endif
```

```
function y ← scan_+(x) // Pre: |x| is power of 2
if |x| ≥ 2 then
    x_e, x_o ← evens(x), odds(x)
    u ← x_e + x_o
    y_o ← scan_+(u)
    y_e ← y_o − x_o
    return y ← weave(y_e, y_o)
else
    return y ← x
endif
```

```
function y ← scan_+(x) // Pre: |x| is power of 2
if |x| ≥ 2 then
    x_e, x_o ← evens(x), odds(x)
    u ← x_e + x_o
    y_o ← scan_+(u)
    y_e ← y_o − x_o
    return y ← weave(y_e, y_o)
else
    return y ← x
endif
```

```
function y ← scan_+(x) // Pre: |x| is power of 2
if |x| ≥ 2 then
    x_e, x_o ← evens(x), odds(x)
    u ← x_e + x_o
    y_o ← scan_+(u)
    y_e ← y_o − x_o
    return y ← weave(y_e, y_o)
else
    return y ← x
endif
```

```
function y ← scan_+(x) // Pre: |x| is power of 2
if |x| ≥ 2 then
    x_e, x_o ← evens(x), odds(x)
    u ← x_e + x_o
    y_o ← scan_+(u)
    y_e ← y_o − x_o
    return y ← weave(y_e, y_o)
else
    return y ← x
endif
```

```
function y ← scan_+(x) // Pre: |x| is power of 2
if |x| ≥ 2 then
    x_e, x_o ← evens(x), odds(x)
    u ← x_e + x_o
    y_o ← scan_+(u)
    y_e ← y_o − x_o
    return y ← weave(y_e, y_o)
else
    return y ← x
endif
```

$$
\begin{aligned}
W_{\text{scan}}(n) &= \Theta(n) \\
D_{\text{scan}}(n) &= \Theta(\log n)
\end{aligned}
$$

$$\Downarrow$$

$$
p_0 \equiv \frac{W}{D} = \Theta\left(\frac{n}{\log n}\right)
$$

Cilk / [Intel Cilk Plus]

R. D. Blumofe, et al. Cilk: An efficient multithreaded runtime system. In *Proc. ACM Conf. on Principles and Practice of Parallel Programming (PPoPP)*, pp. 207–216, Santa Barbara, CA, USA, July 1995.

For documentation, see also: http://software.intel.com/en-us/intel-cilk-plus

Cilk Plus in a nutshell

- C/C++ extensions to support nested task and data parallelism

```
int fib (int n) {
  if (n < 2) return 1;
  else {
    int x, y;
    x = fib (n-1);
    y = fib (n-2);

    return x + y;
  }
}
```

```
int fib (int n) {
  if (n < 2) return 1;
  else {
    int x, y;
    x = _Cilk_spawn fib (n-1);
    y = fib (n-2);
    _Cilk_sync;
    return x + y;
  }
}
```

Parallel for loops

```
// Computes A += B, sequentially
void vadd (double* A, double* B, int n) { … }

_Cilk_for (int k=0; k < n; ++k)
  vadd (A+k, B+k, 1);
```


Source: Cilk Arts tutorial

Reducers

- Naïve (incorrect) Cilk Plus:

```
cilk_for (i = 0; i < n; ++i)
  s += x[i];  // Data race!
```

- Correct Cilk: Use “reducer” object

```
cilk::reducer_opadd<float> r;
cilk_for (i = 0; i < n; ++i)
  r += x[i];
s += r.get_value ();
```

More interesting example

```
const Node* target = …;
list<Node *> collisions;  // Stores collisions

void findCollisions (Node* x) {
  if (x->isLeaf ()  &&  target->collidesWith (x))
    collisions->append (x);
  else
    for (Node::iterator child = x->begin (); child != x->end (); ++child)
      findCollisions (child);
}
```


Source: M. Frigo, CScADS 2009 talk

More interesting example

```
const Node* target = …;
list<Node *> collisions;  // Stores collisions

void findCollisions (Node* x) {
  if (x->isLeaf ()  &&  target->collidesWith (x))
    collisions->append (x);
  else
    _Cilk_for (Node::iterator child = x->begin (); child != x->end (); ++child)
      findCollisions (child);
}
```


Source: M. Frigo, CScADS 2009 talk

More interesting example

```
Node* target = …;
list<Node *> collisions;  // Stores collisions

void findCollisions (Node* x) {
  if (x->isLeaf ()  &&  target->collidesWith (x))
    collisions->append (x);  // Data race!
  else
    _Cilk_for (Node::iterator child = x->begin (); child != x->end (); ++child)
      findCollisions (child);
}
```


Source: M. Frigo, CScADS 2009 talk

More interesting example

```
Node* target = …;
list<Node *> collisions;  // Stores collisions

void findCollisions (Node* x) {
  if (x->isLeaf ()  &&  target->collidesWith (x)) {
    cilk::mutex m;
    m.lock ();
    collisions->append (x);
    m.unlock ();
  }
  else
    _Cilk_for (Node::iterator child = x->begin (); child != x->end (); ++child)
      findCollisions (child);
}
```


Source: M. Frigo, CScADS 2009 talk

More interesting example

```
Node* target = …;
list<Node *> collisions;  // Stores collisions

void findCollisions (Node* x) {
  if (x->isLeaf ()  &&  target->collidesWith (x)) {
    cilk::mutex m;
    m.lock ();            // Performance issue – lock contention
    collisions->append (x);
    m.unlock ();
  }
  else
    _Cilk_for (Node::iterator child = x->begin (); child != x->end (); ++child)
      findCollisions (child);
}
```


Source: M. Frigo, CScADS 2009 talk

More interesting example

```
Node* target = …;
cilk::reducer_list_append<Node *> collisions;  // Stores collisions

void findCollisions (Node* x) {
  if (x->isLeaf ()  &&  target->collidesWith (x)) {
    collisions->append (x);
  }
  else
    _Cilk_for (Node::iterator child = x->begin (); child != x->end (); ++child)
      findCollisions (child);
}
```


Source: M. Frigo, CScADS 2009 talk

DAG view of execution

Cilk Plus

```
A ();
_Cilk_spawn B ();
C ();
_Cilk_sync;
D ();
```

Serial elision

```
A ();
B ();
C ();

D ();
```

DAG

Source: M. Frigo, CScADS 2009 talk

Hyperobjects

"Regular" objects: Same view

Hyperobjects: Observer-dependent view

Source: M. Frigo, CScADS 2009 talk

Reduction over list concatenation

```
x.append (0);
_Cilk_spawn x.append (1);
x.append (2);
x.append (3);
_Cilk_sync;
```


Source: M. Frigo, CScADS 2009 talk

Reduction over list concatenation

```
x.append (0);
_Cilk_spawn x.append (1);
x.append (2);
x.append (3);
_Cilk_sync;
```

Serial execution

Source: M. Frigo, CScADS 2009 talk

Reduction over list concatenation

```
x.append (0);
_Cilk_spawn x.append (1);
x.append (2);
x.append (3);
_Cilk_sync;
```

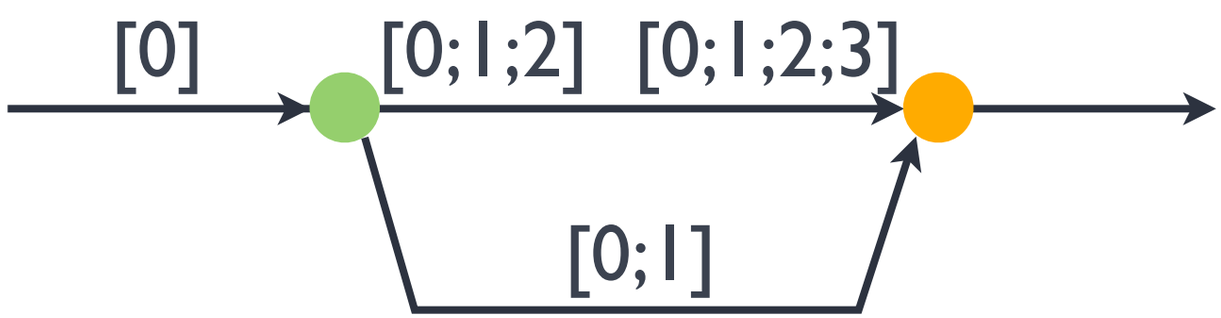
Serial execution

Source: M. Frigo, CScADS 2009 talk

Reduction over list concatenation

```
x.append (0);
_Cilk_spawn x.append (1);
x.append (2);
x.append (3);
_Cilk_sync;
```

Serial execution

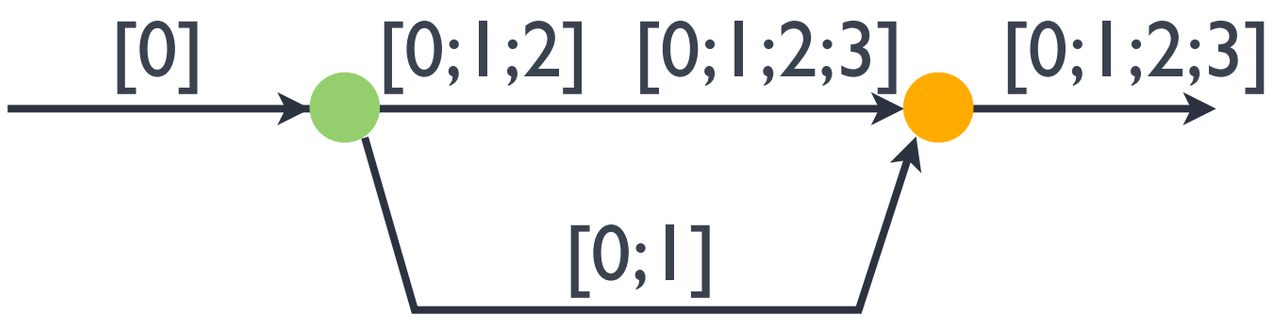


Source: M. Frigo, CScADS 2009 talk

Reduction over list concatenation

```
x.append (0);
_Cilk_spawn x.append (1);
x.append (2);
x.append (3);
_Cilk_sync;
```

Serial execution

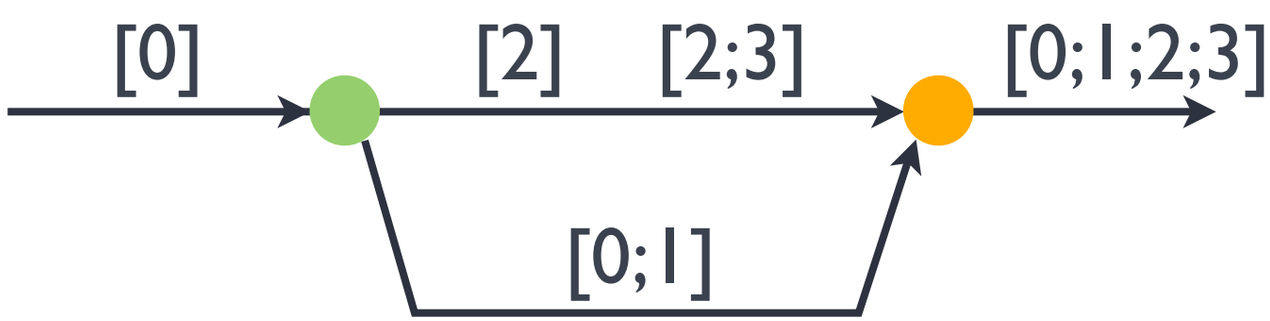


Source: M. Frigo, CScADS 2009 talk

Reduction over list concatenation

```
x.append (0);
_Cilk_spawn x.append (1);
x.append (2);
x.append (3);
_Cilk_sync;
```

Parallel execution: Reduction over monoids is deterministic.



Source: M. Frigo, CScADS 2009 talk

Additional reducers

- Provided
  - Lists (append, prepend)
  - min, max, {min, max}_index
  - opadd
  - ostream
  - basic_string
- May build your own

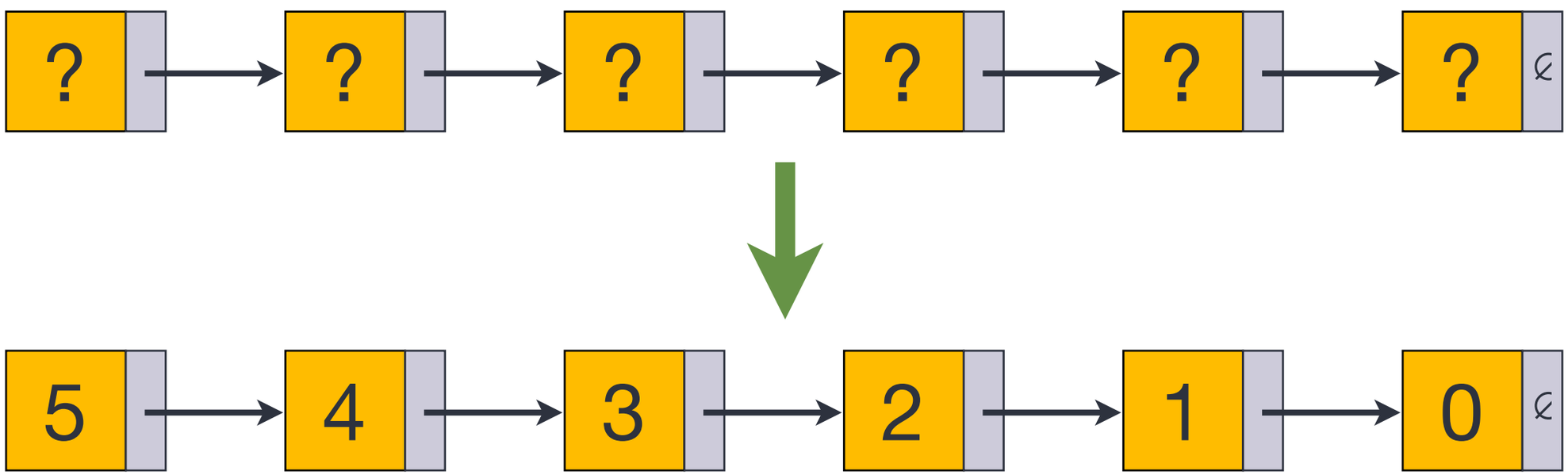
Additional features and caveats

- Fences (Cilk_fence) & locking (Cilk_lock)
- Automatic clean-up for local dynamic allocation (Cilk_alloca)
- Aborts
- “Inlets” support use of spawned results in arbitrary expressions
- Race detector: Cilkscreen
- Beware sharing through pointer-passing, deadlocks

Parallel list processing

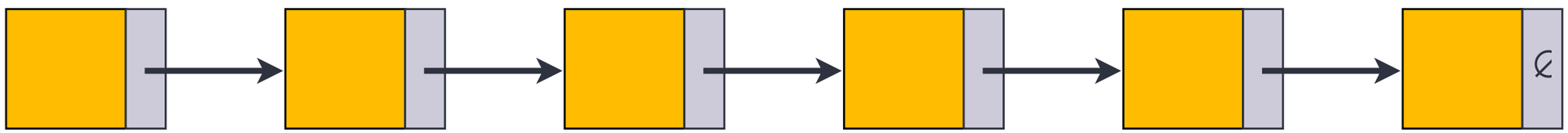
Parallel list processing algorithms

- Data parallel processing of arrays seems easy. What about lists?
- Example: List ranking: Determine distance to tail

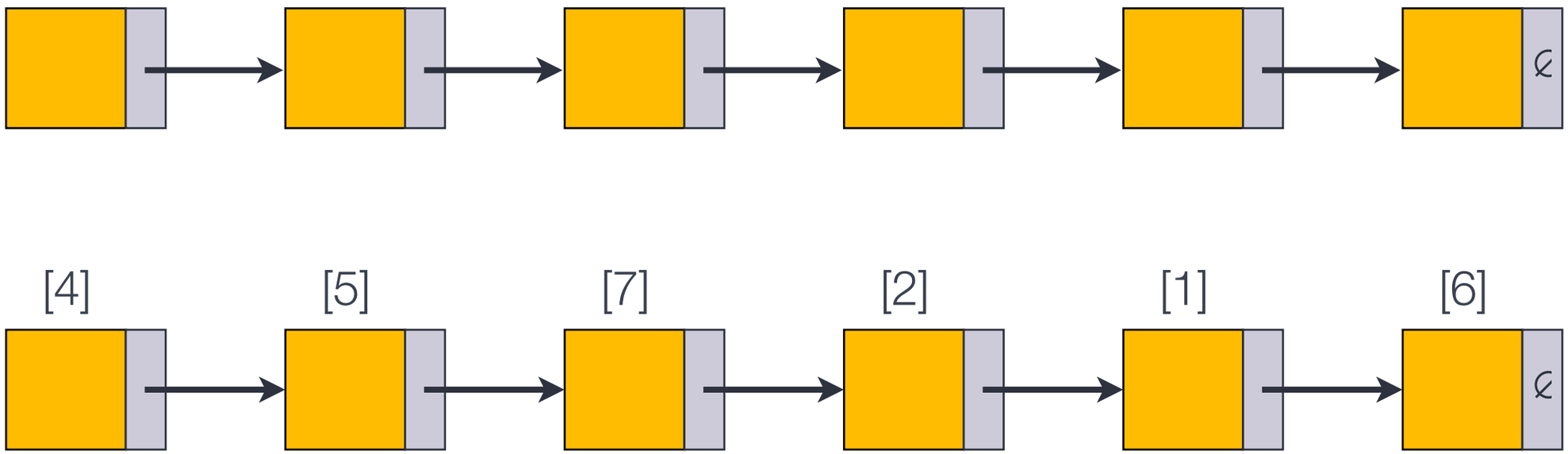


- In principle, this is a parallel prefix/scan problem. (*Why*?)

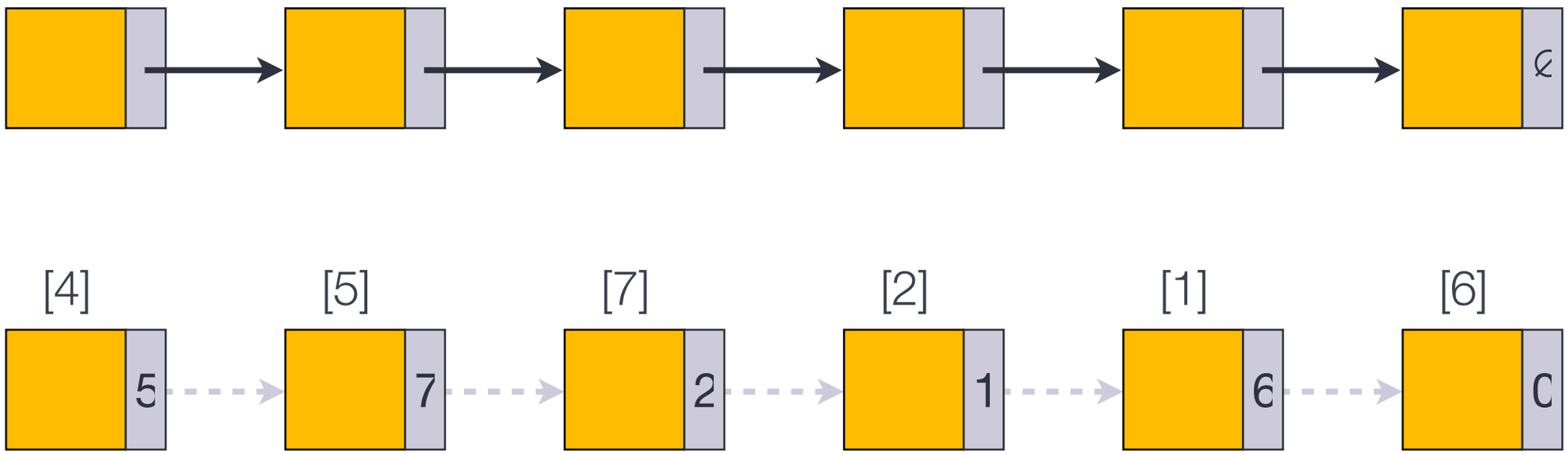
List storage: Simplifying assumption



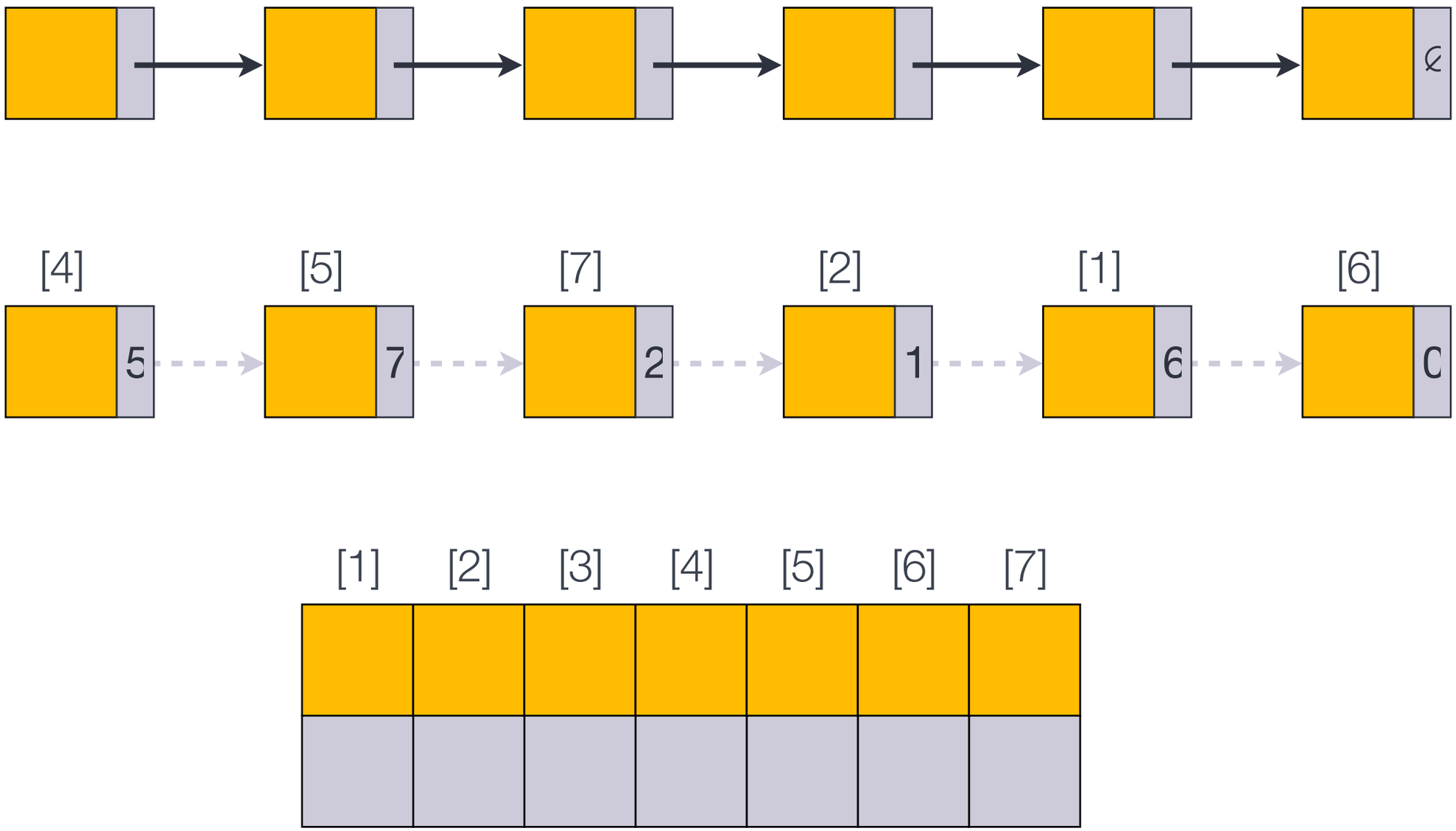
List storage: Simplifying assumption



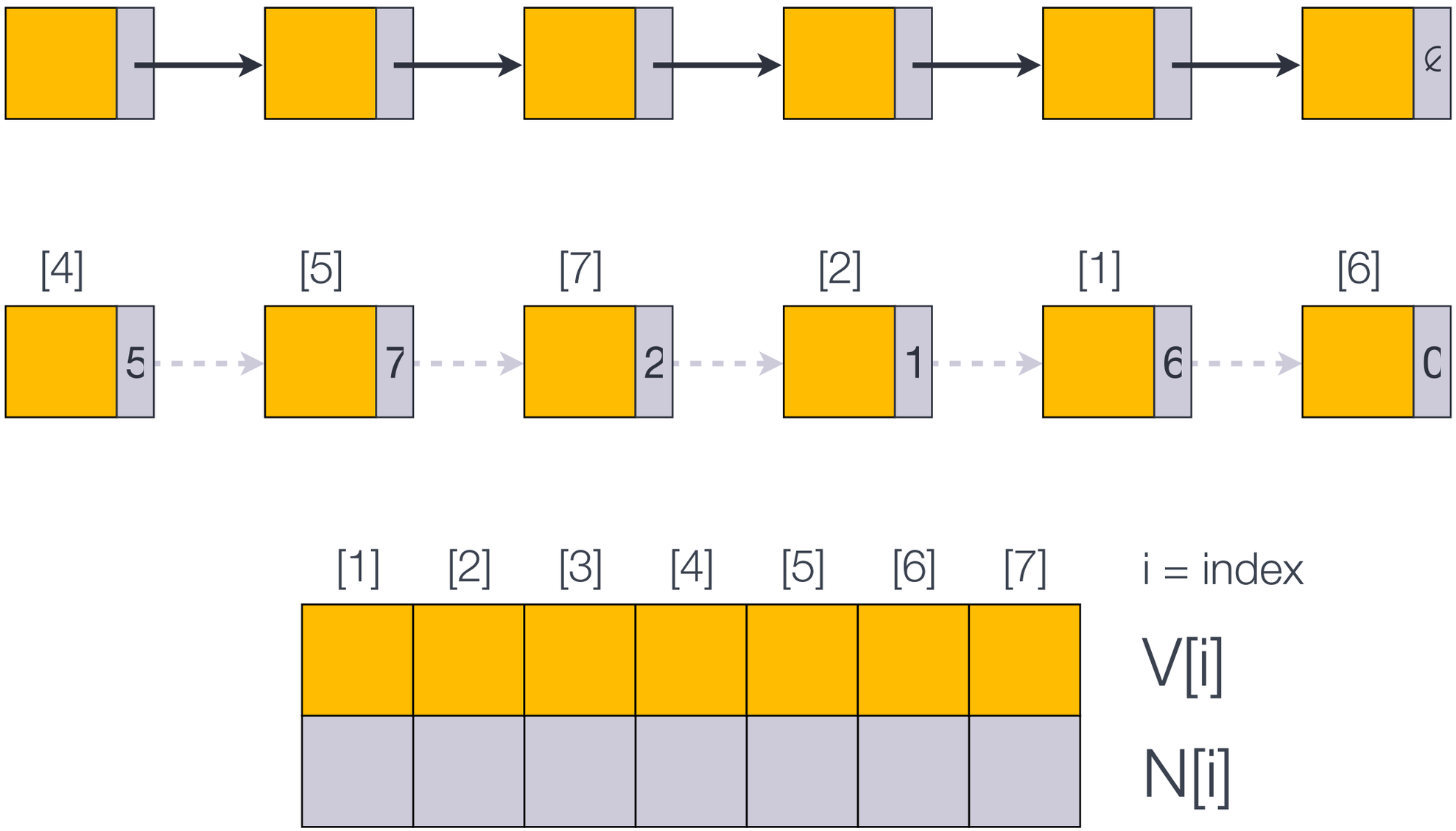
List storage: Simplifying assumption



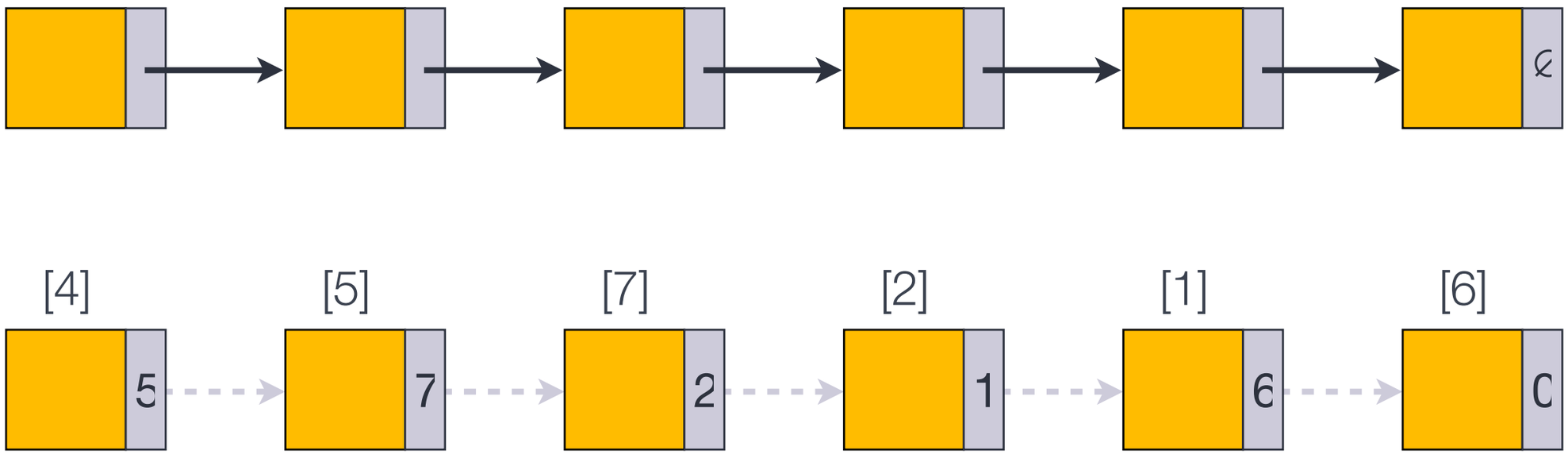
List storage: Simplifying assumption



List storage: Simplifying assumption

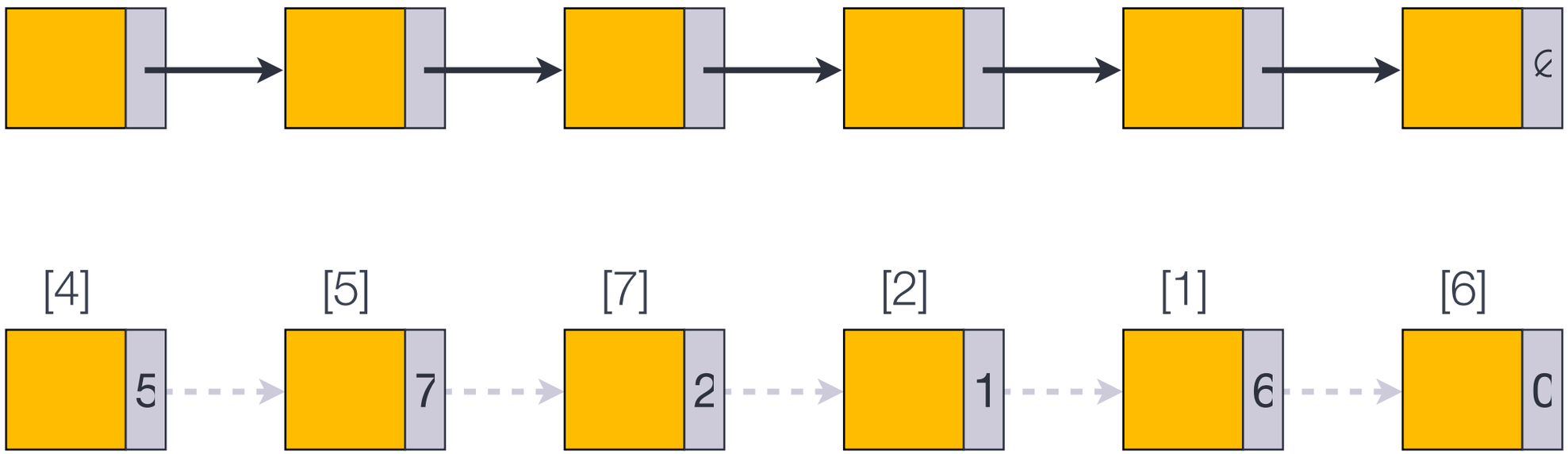


List storage: Simplifying assumption



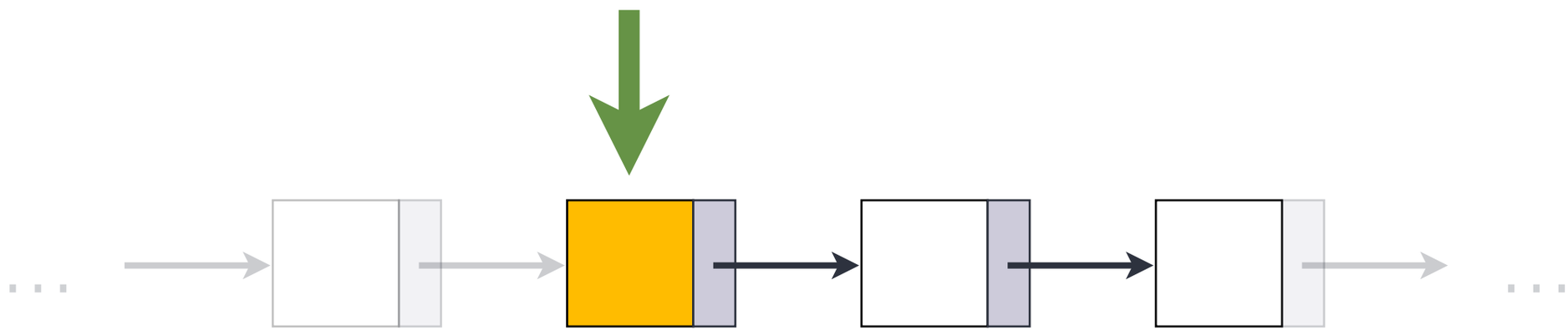
[1]	[2]	[3]	[4]	[5]	[6]	[7]	i = index
		?					V[i]
6	1	?	5	7	0	2	N[i]

List storage: Simplifying assumption

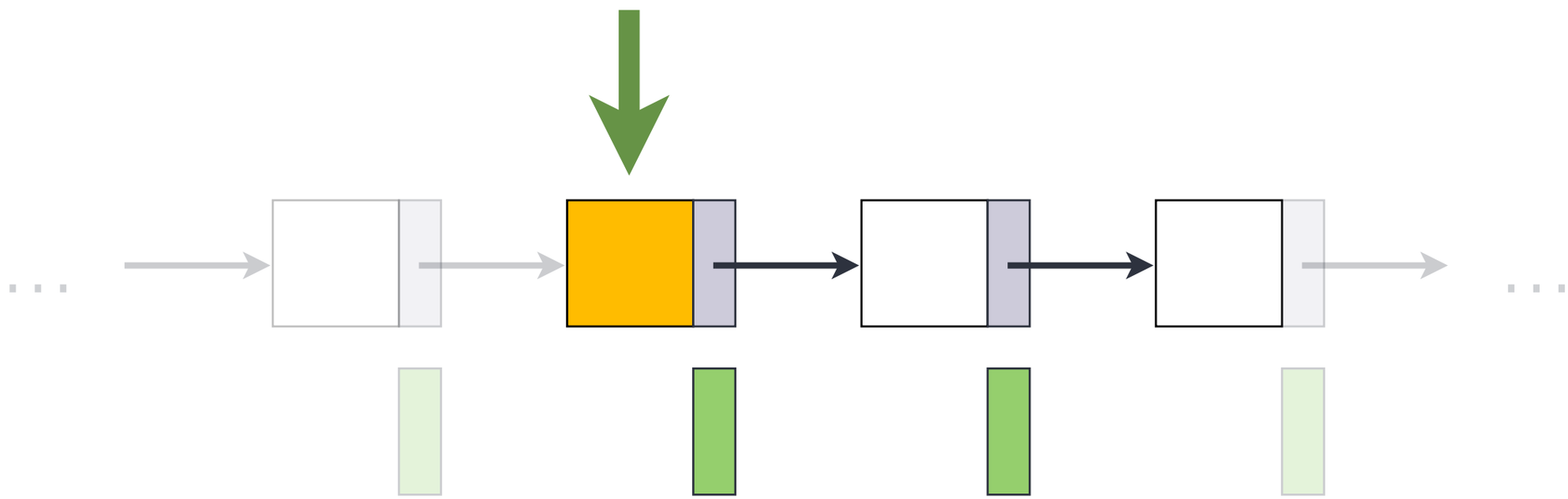


[1]	[2]	[3]	[4]	[5]	[6]	[7]	i = index
		?					V[i]
6	1	0	5	7	0	2	N[i]

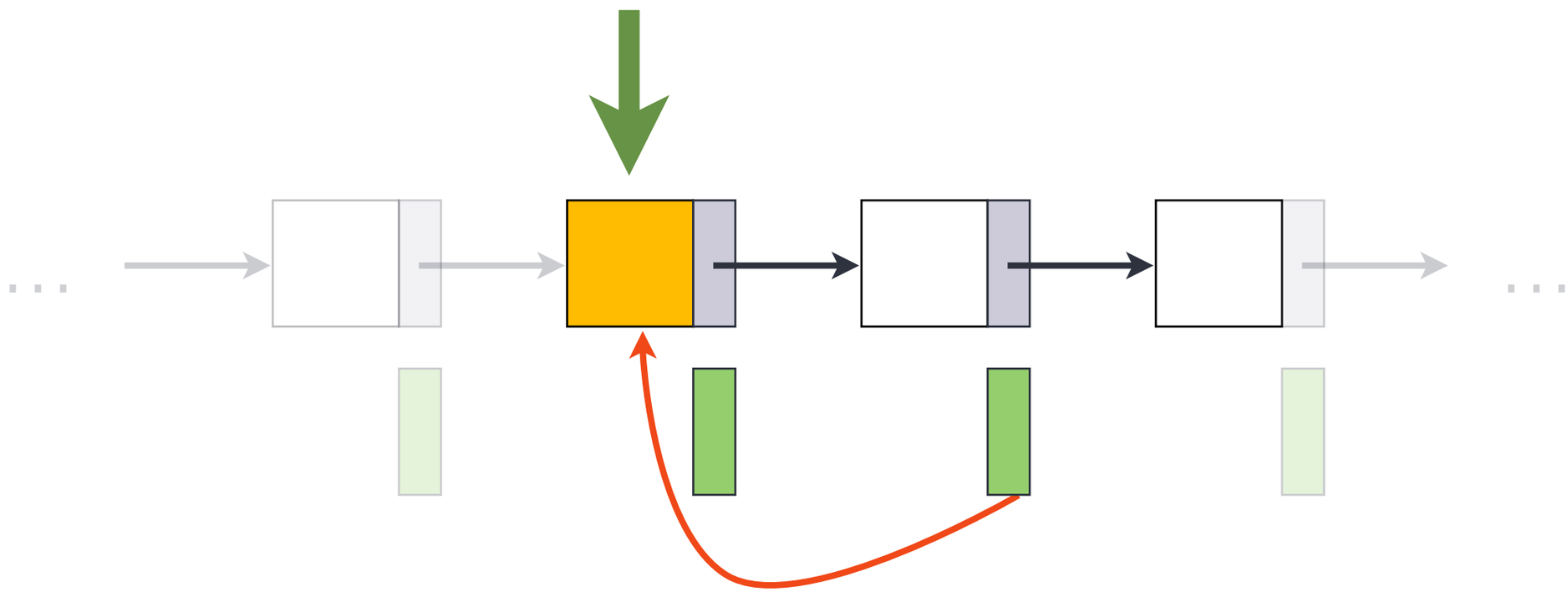
Parallel list processing: Reversals



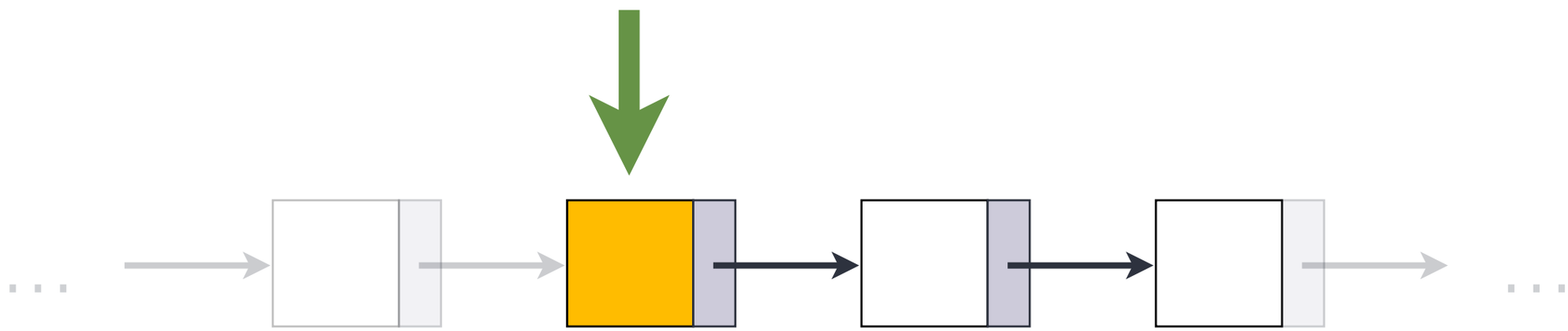
Parallel list processing: Reversals



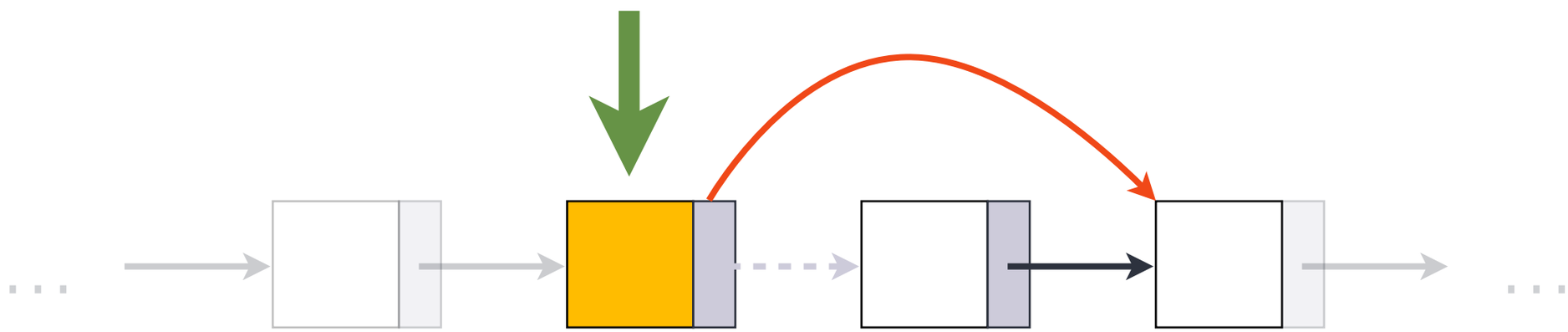
Parallel list processing: Reversals



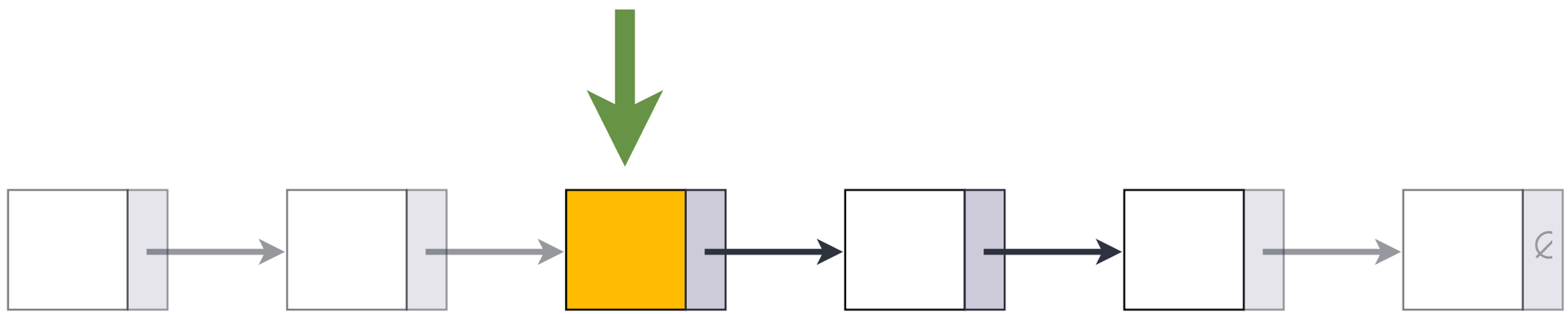
Parallel list processing: Jumps



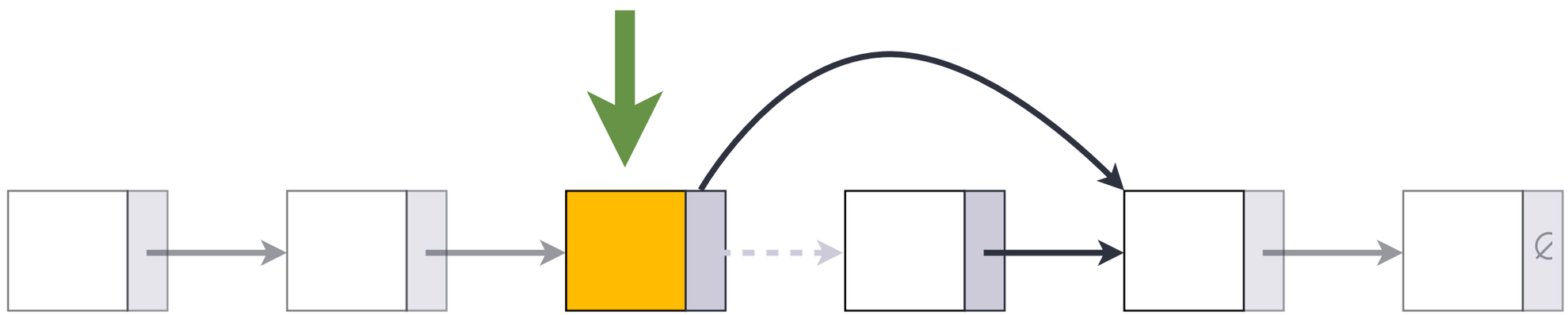
Parallel list processing: Jumps



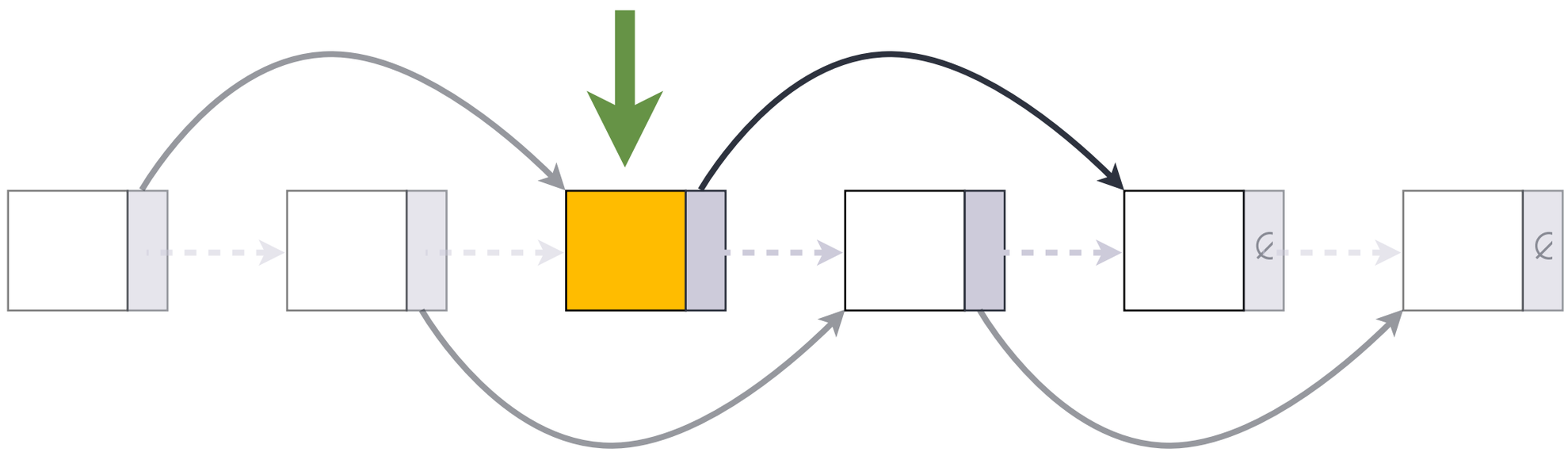
Parallel list processing: Jumps



Parallel list processing: Jumps

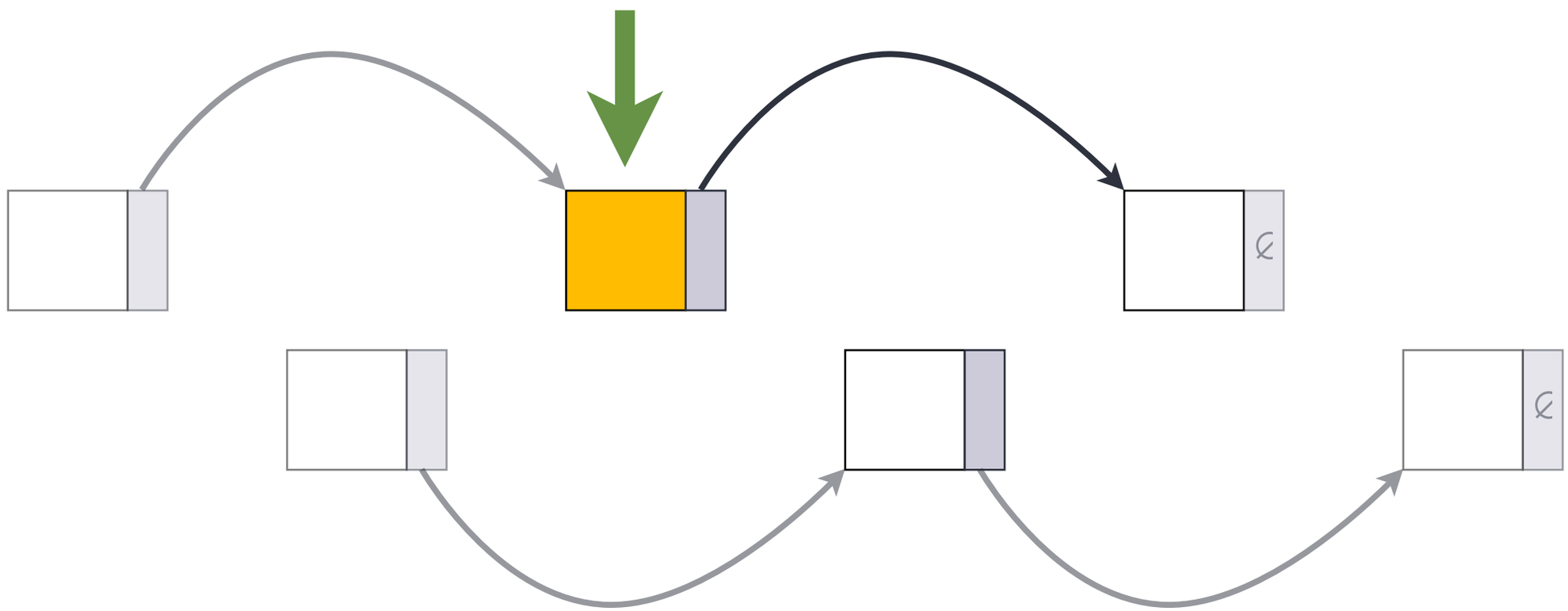


Parallel list processing: Jumps

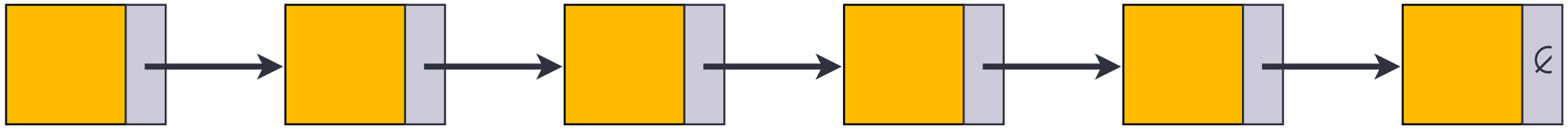


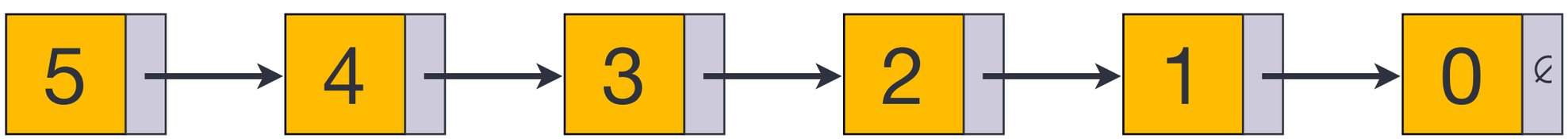
Note: Need synchronized execution

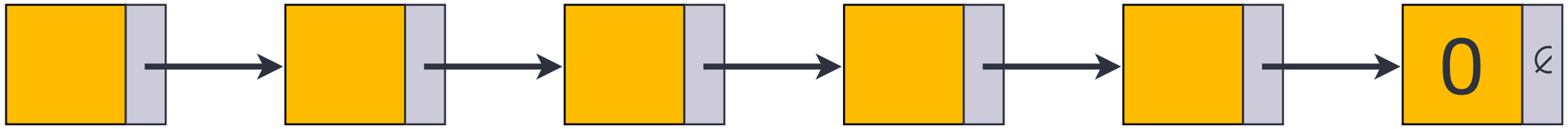
Parallel list processing: Jumps

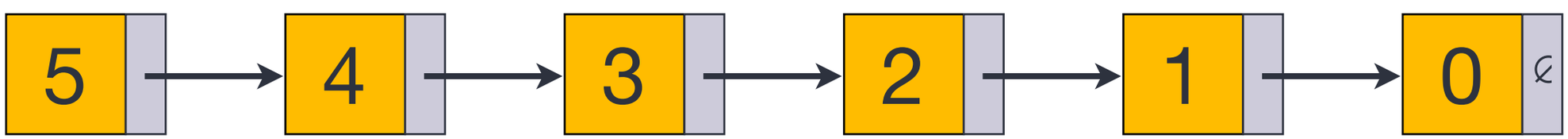


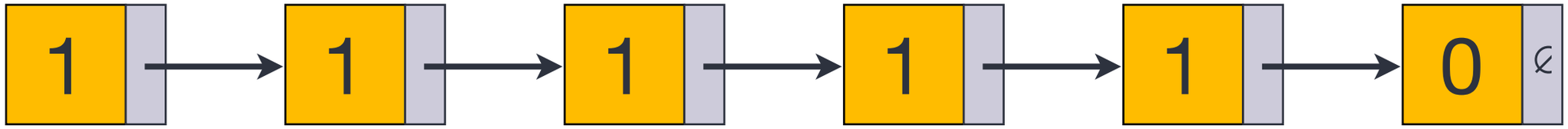
Parallel sublists!

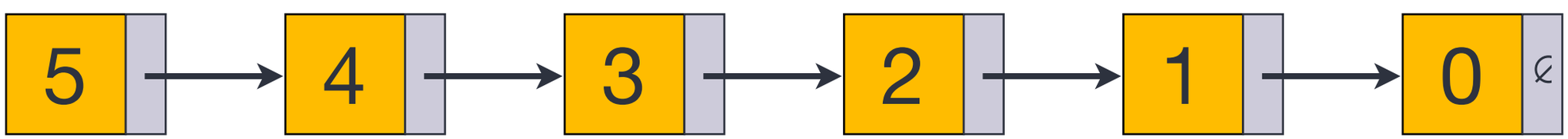


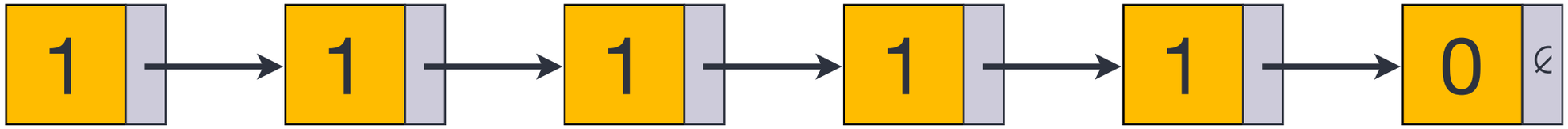

5
4
3
2
1
0


0

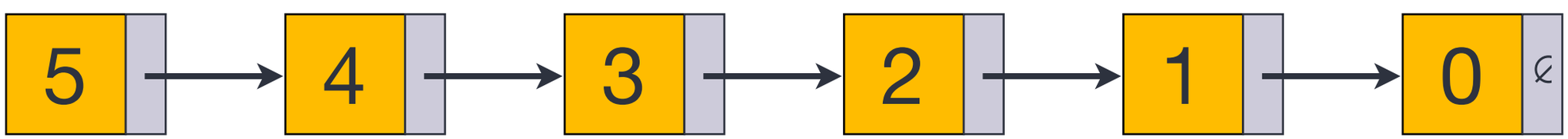

5
4
3
2
1
0


1
1
1
1
1
0
¢

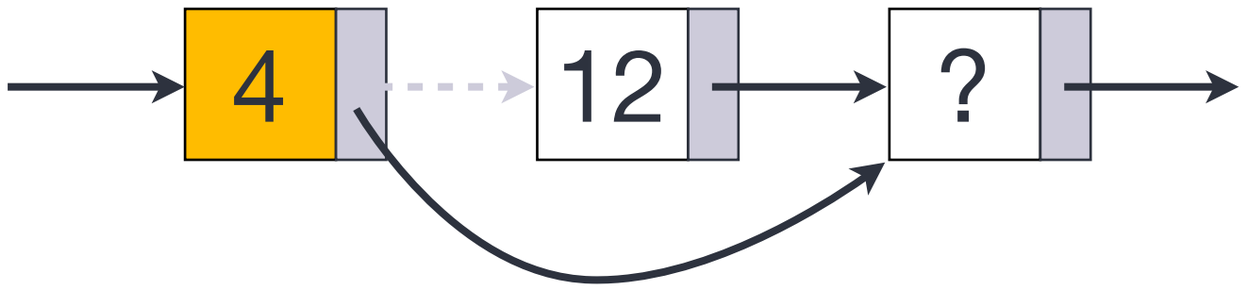

5
4
3
2
1
0
¢



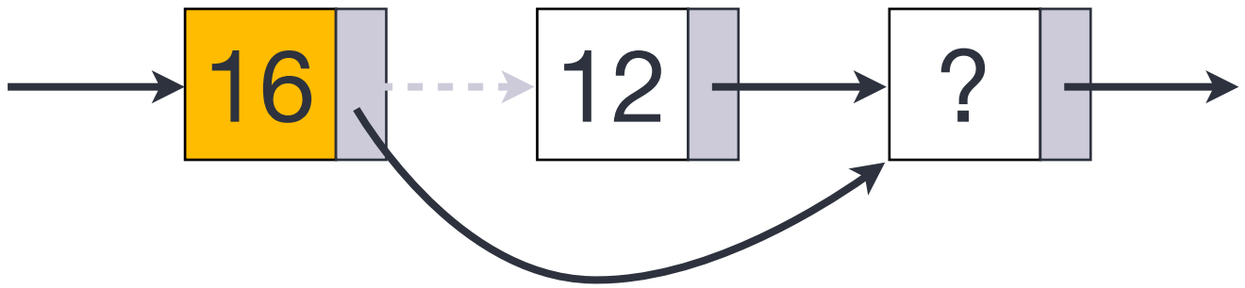
Candidate invariant: $$\mathrm{rank}(i) = \sum_{k \leftarrow i}^{\mathrm{tail}} V[k]$$



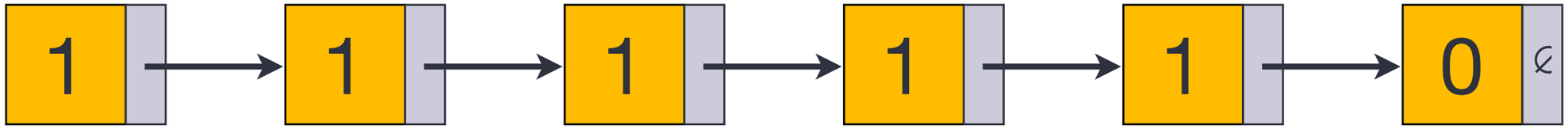
Candidate invariant: $$\mathrm{rank}(i) = \sum_{k \leftarrow i}^{\mathrm{tail}} V[k]$$

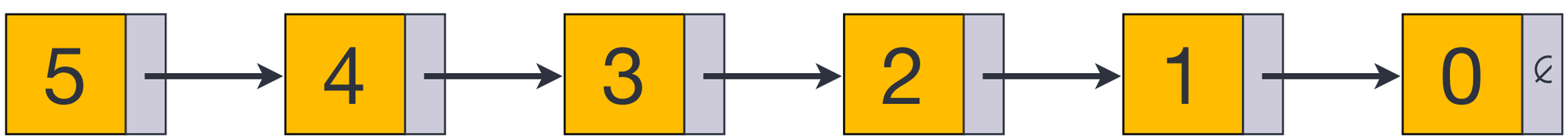


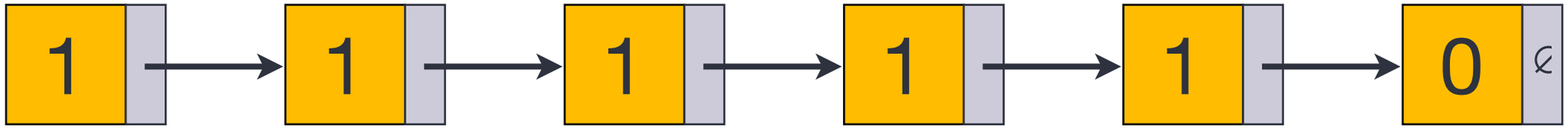
Candidate invariant: $$\mathrm{rank}(i) = \sum_{k \leftarrow i}^{\mathrm{tail}} V[k]$$

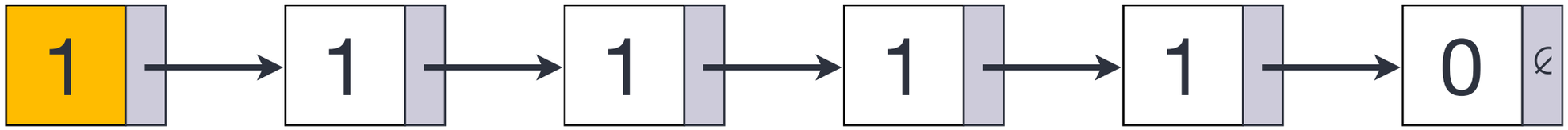


Candidate invariant: $$\mathrm{rank}(i) = \sum_{k \leftarrow i}^{\mathrm{tail}} V[k]$$

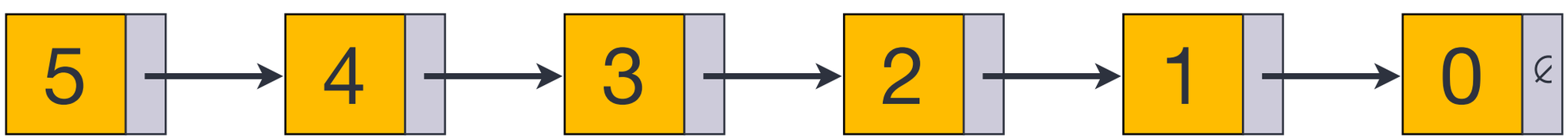

1
1
1
1
1
0
¢

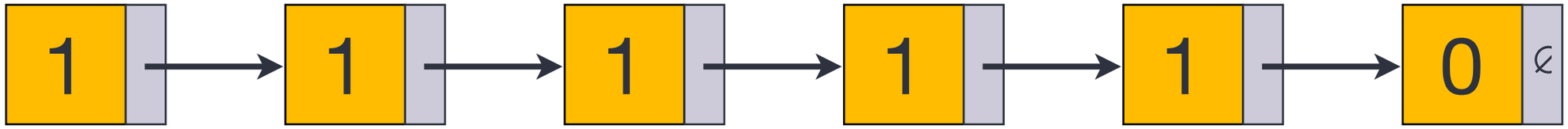

5
4
3
2
1
0
¢

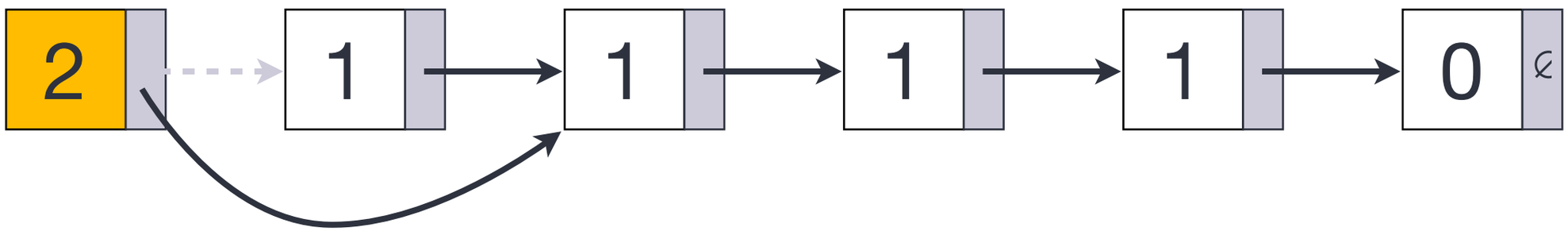




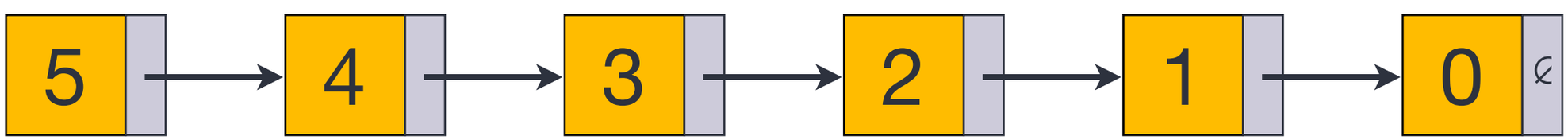
Candidate invariant: $$\mathrm{rank}(i) = \sum_{k \leftarrow i}^{\mathrm{tail}} V[k]$$

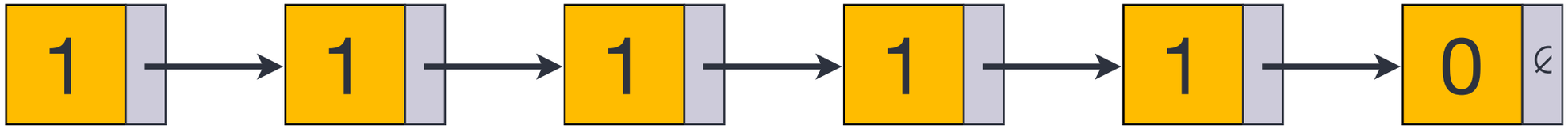


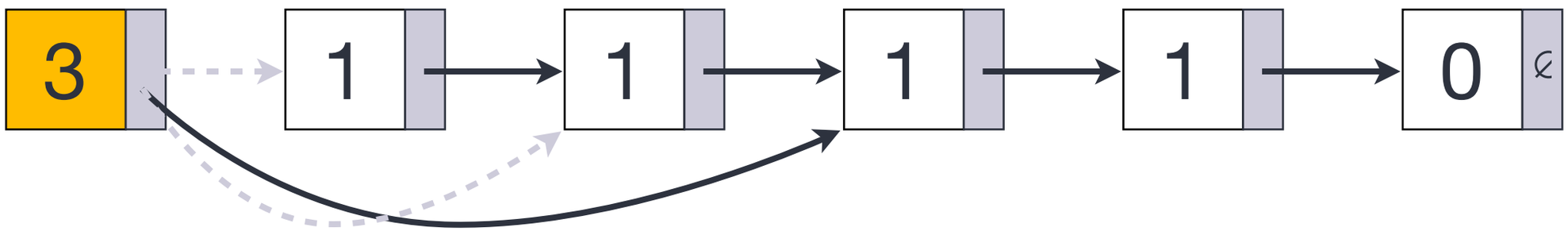




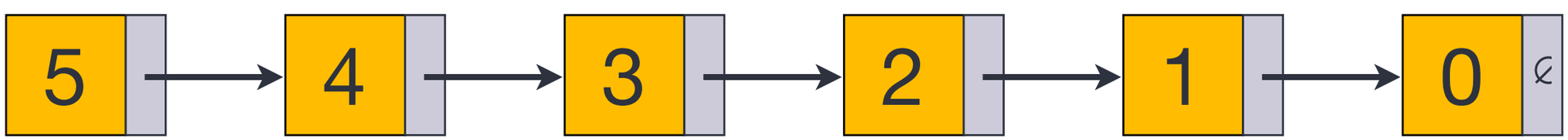
Candidate invariant: $$\mathrm{rank}(i) = \sum_{k \leftarrow i}^{\mathrm{tail}} V[k]$$

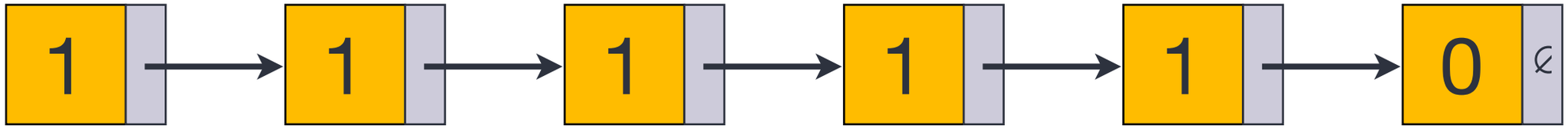


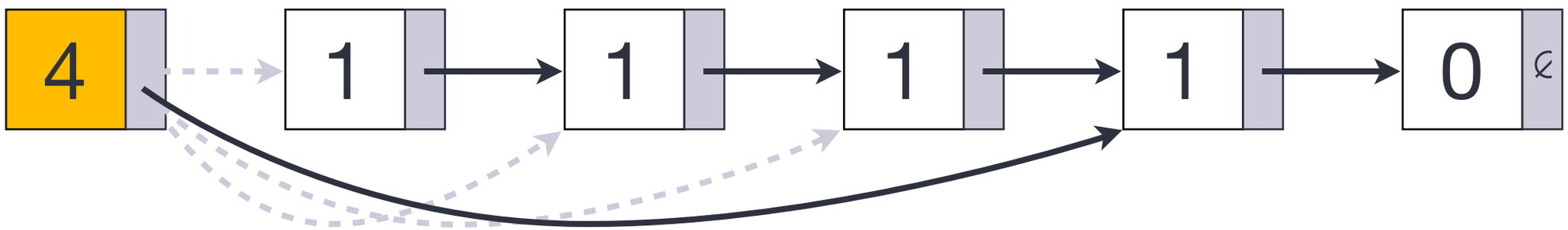




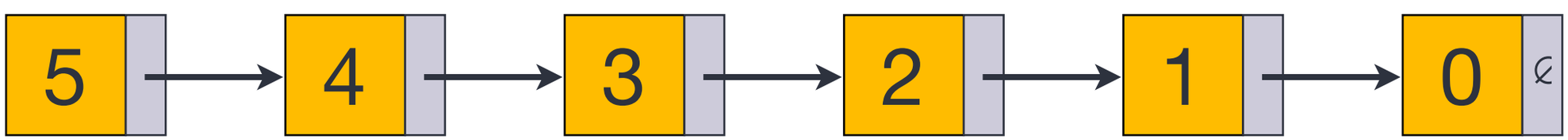
Candidate invariant: $$\mathrm{rank}(i) = \sum_{k \leftarrow i}^{\mathrm{tail}} V[k]$$

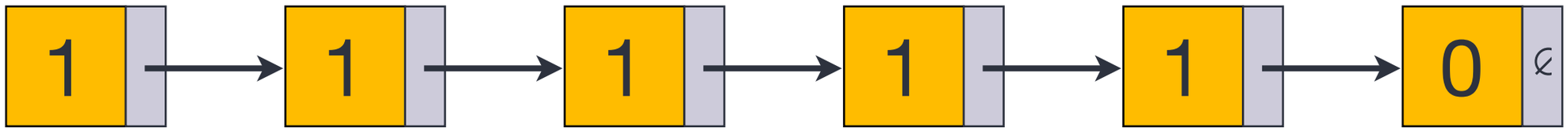


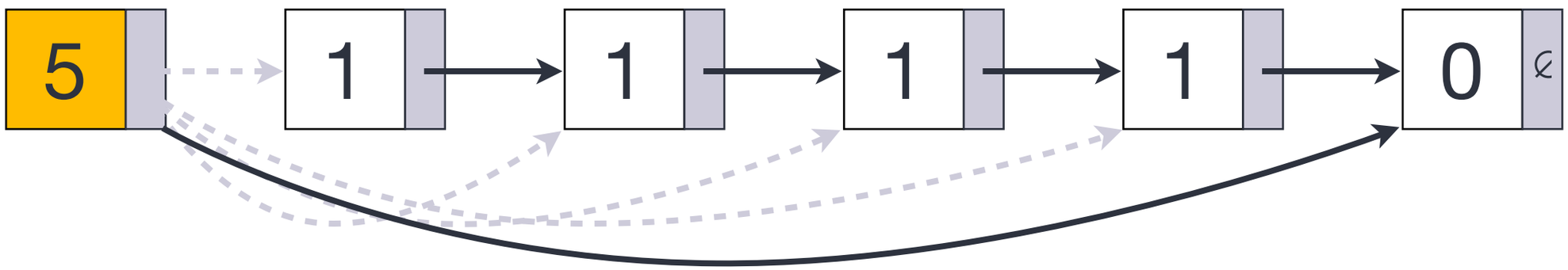




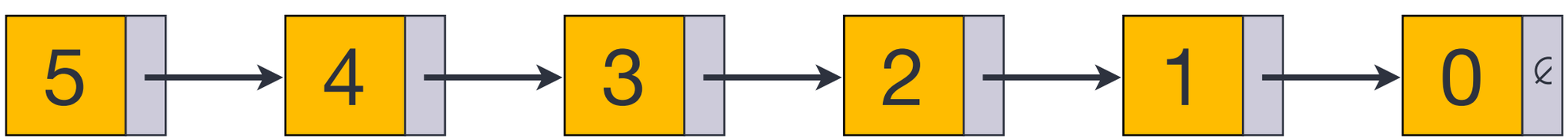
Candidate invariant: $$\mathrm{rank}(i) = \sum_{k \leftarrow i}^{\mathrm{tail}} V[k]$$

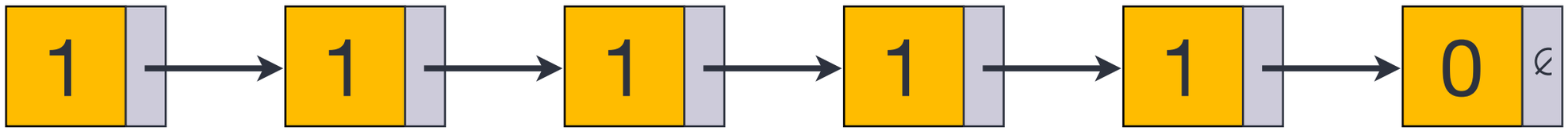


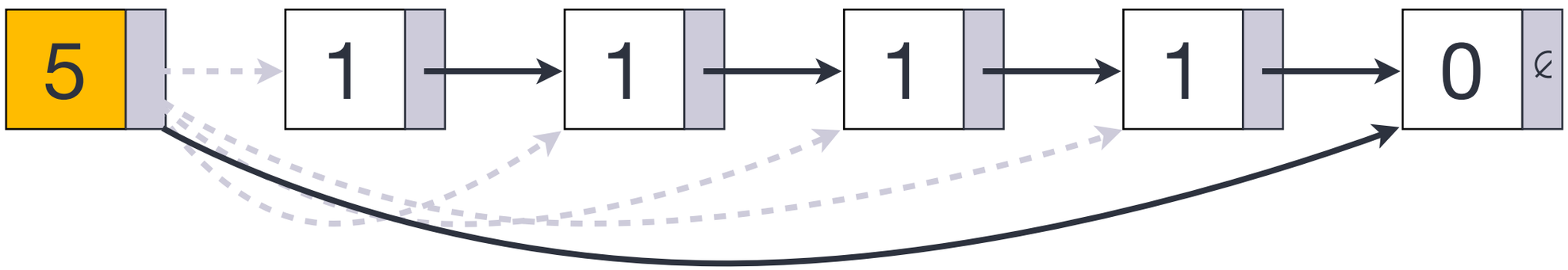




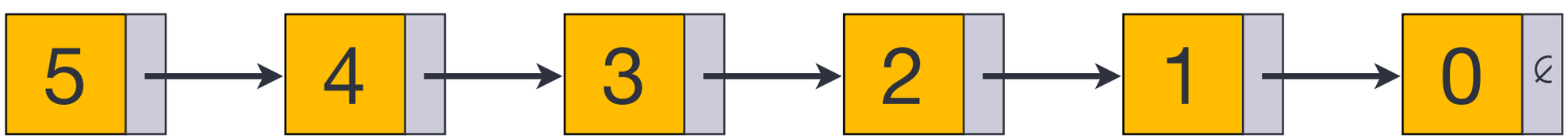
Candidate invariant: $$\mathrm{rank}(i) = \sum_{k \leftarrow i}^{\mathrm{tail}} V[k]$$

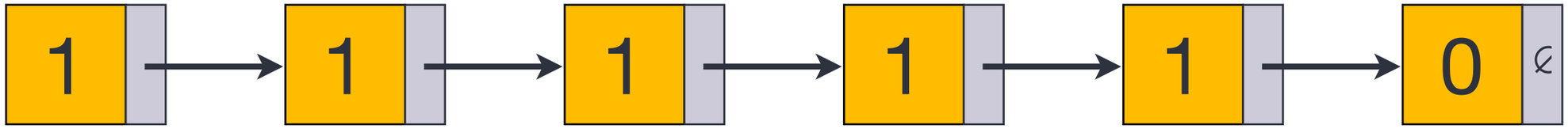


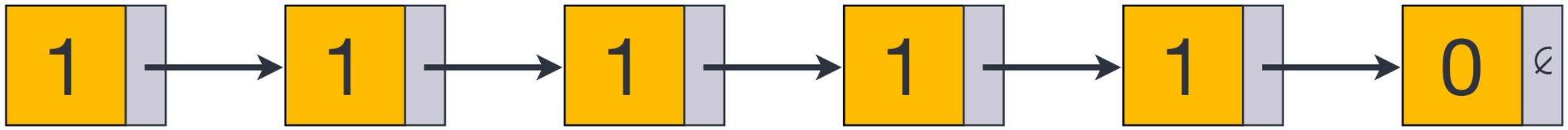




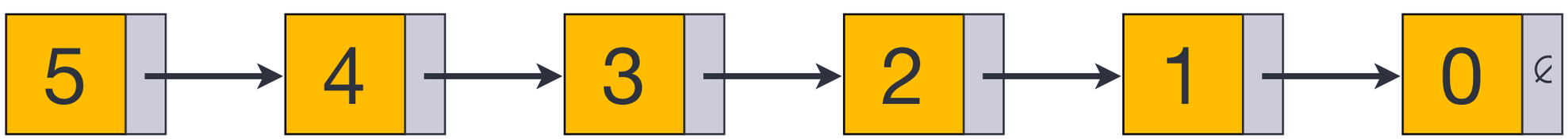
Parallelism?

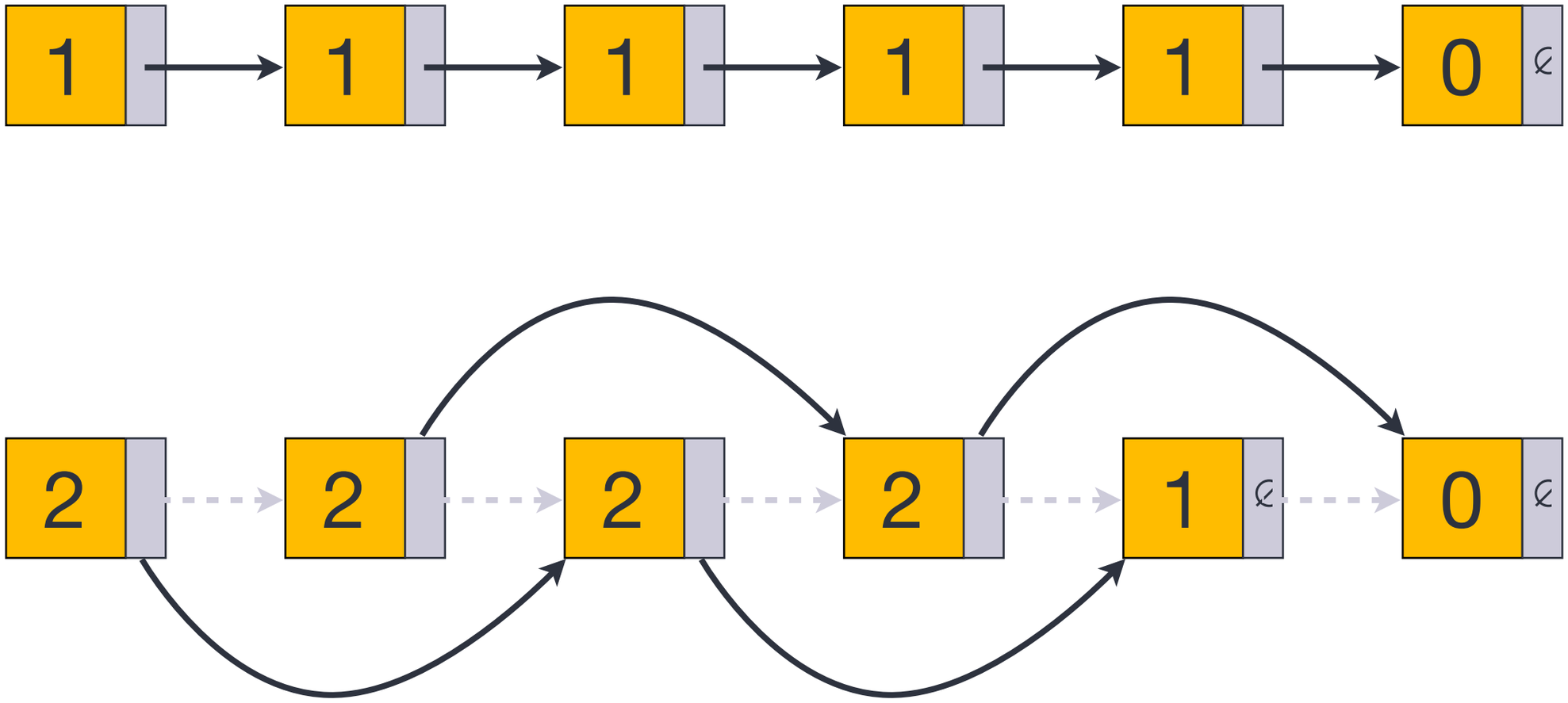




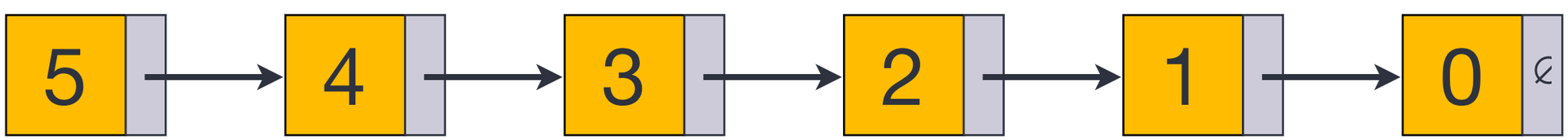


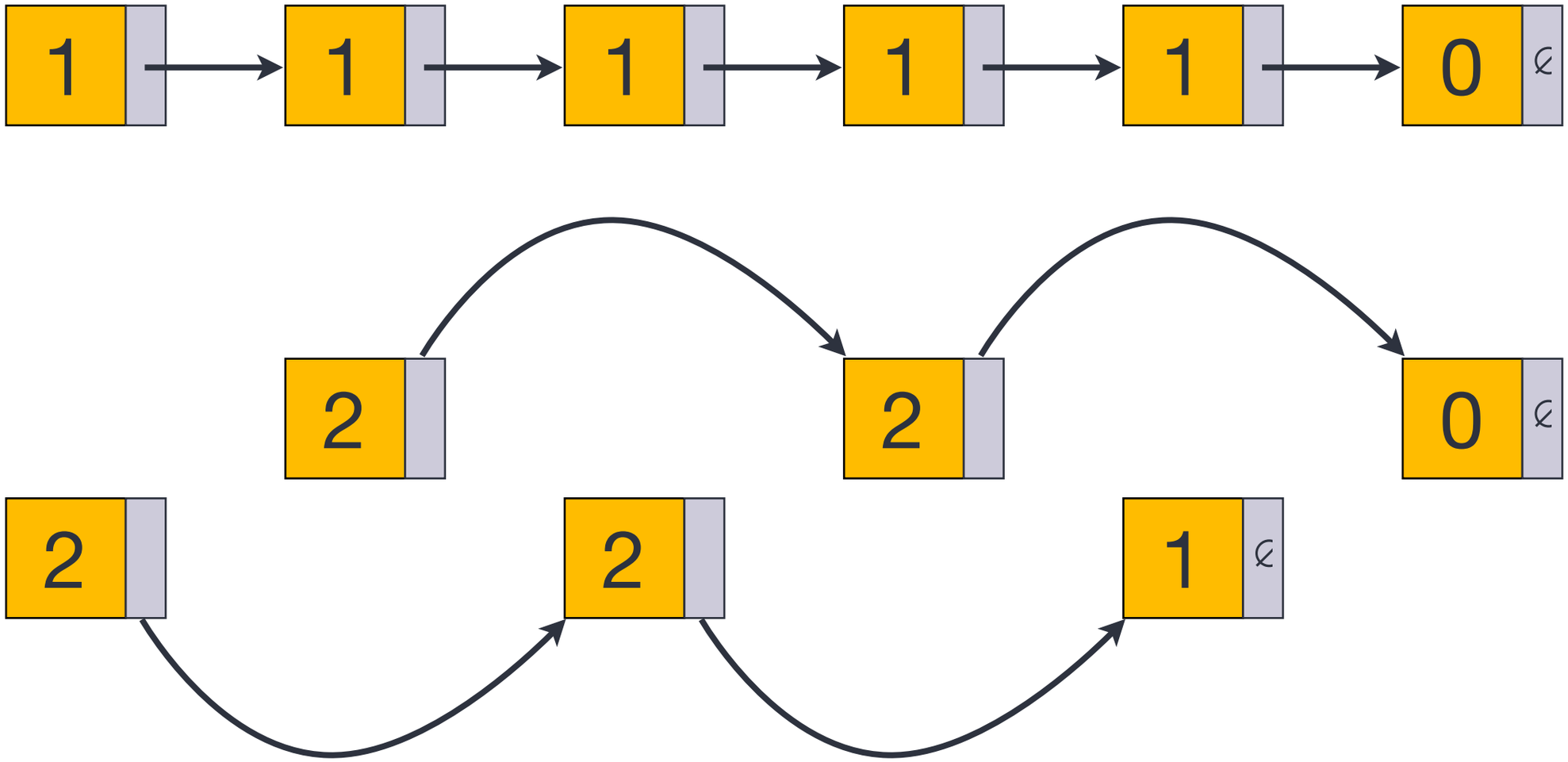
Parallelism?



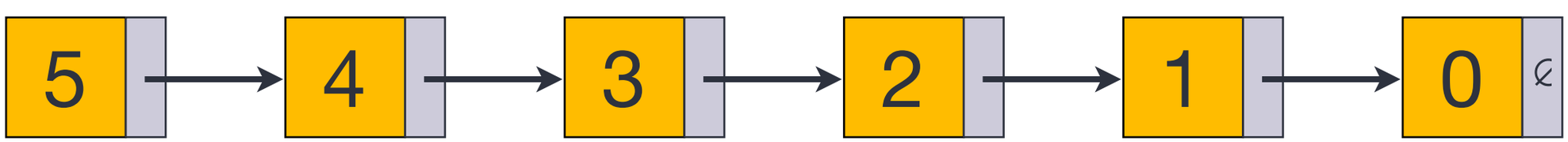


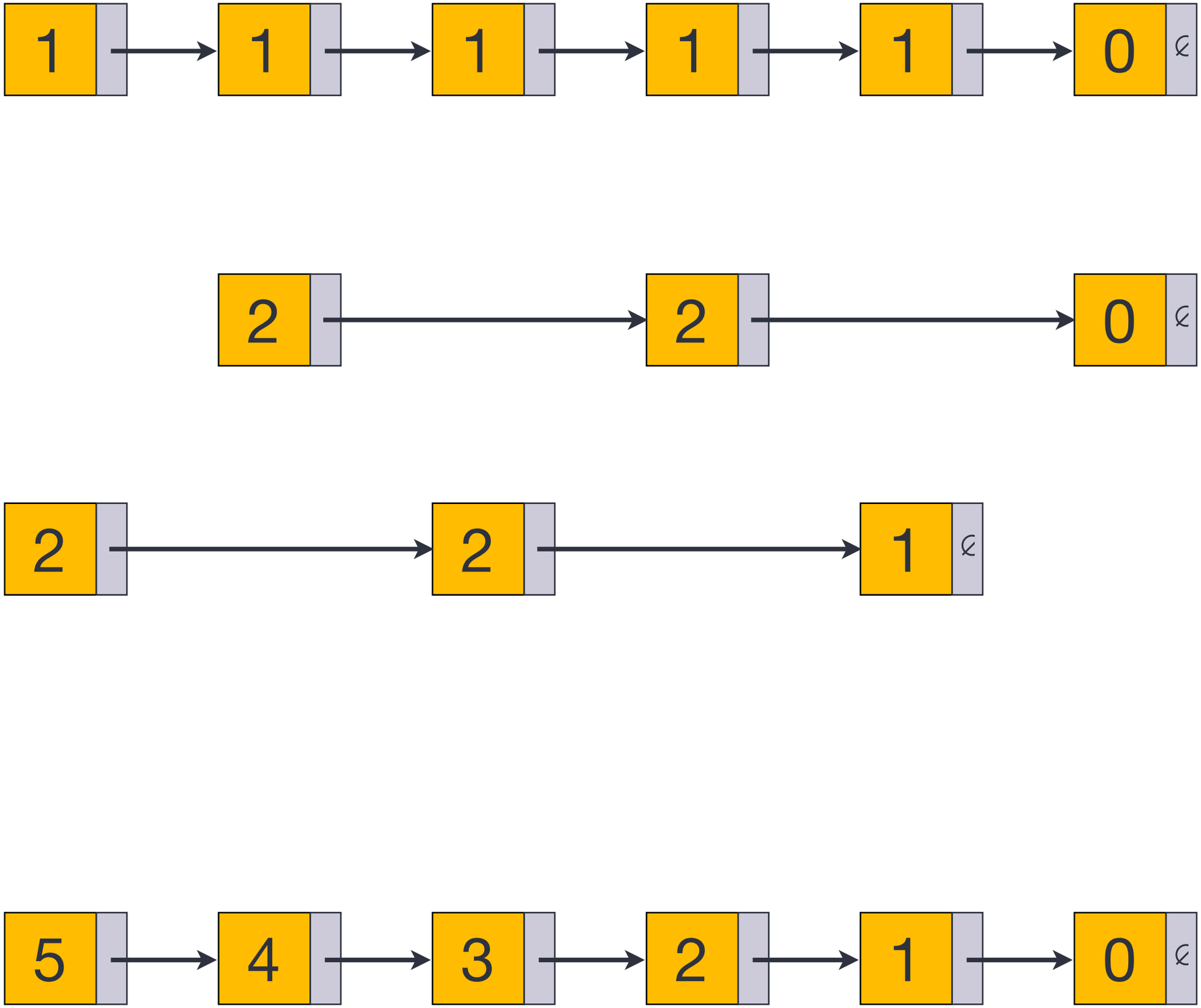
Jump! (And maintain invariant)

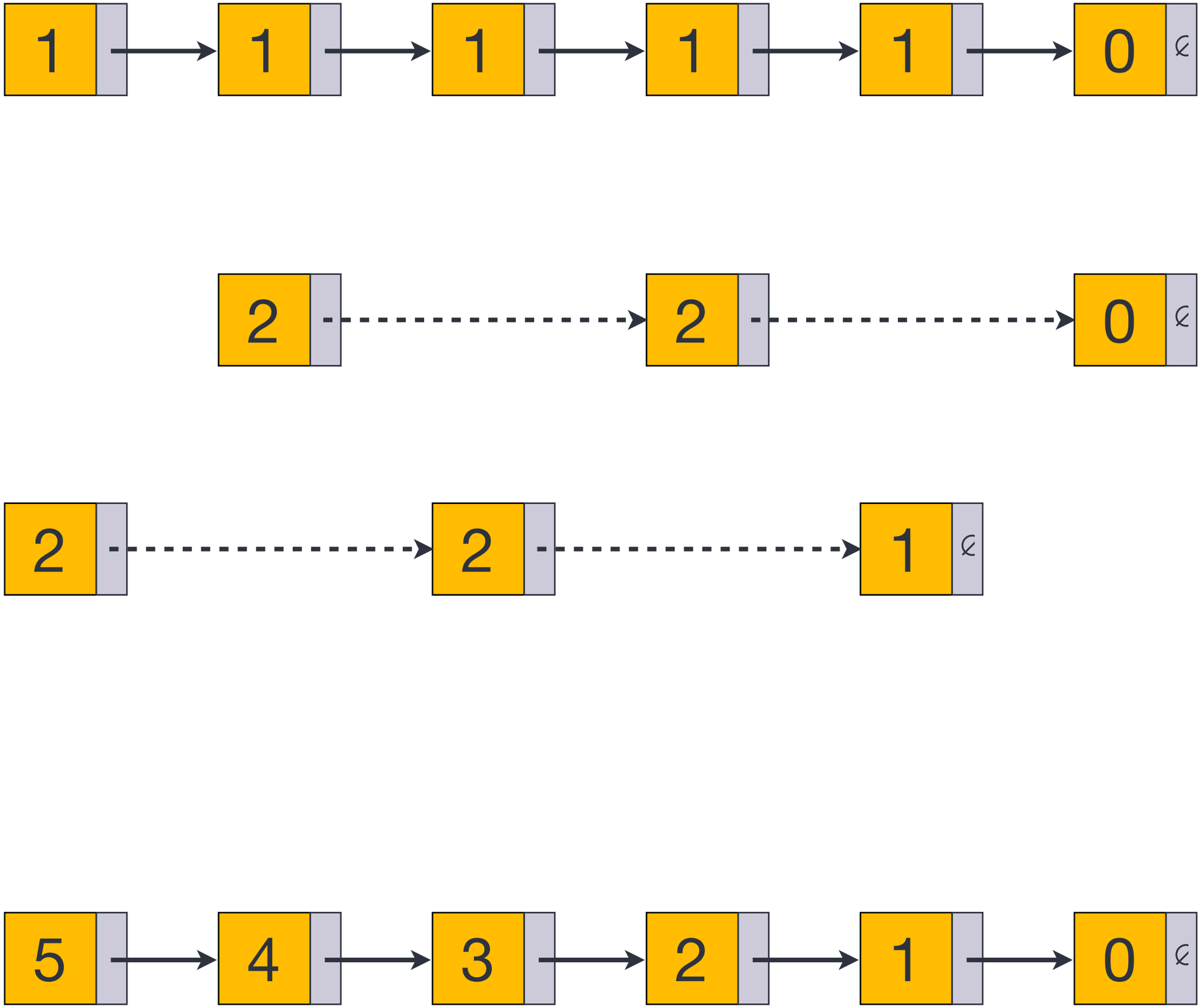


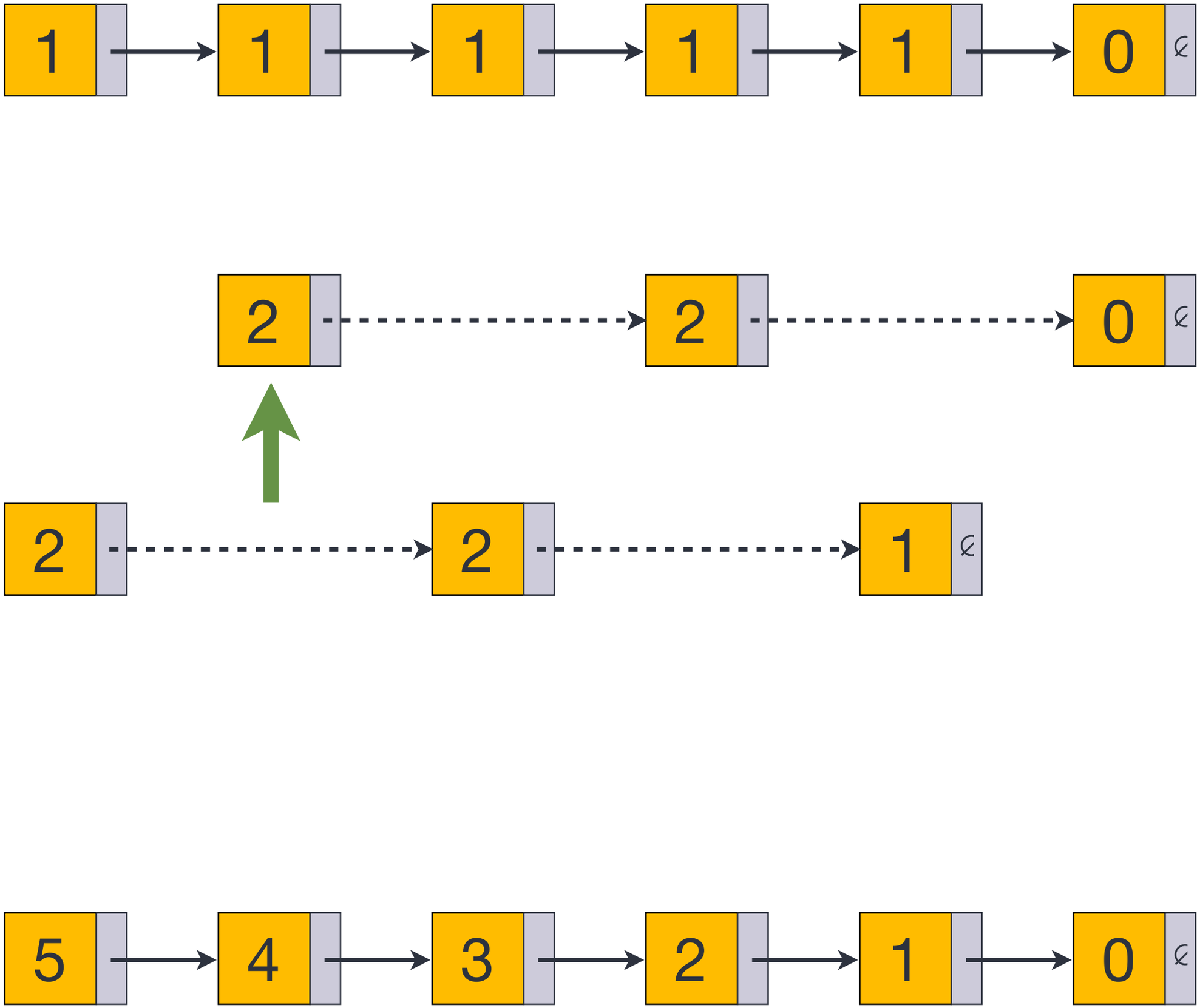


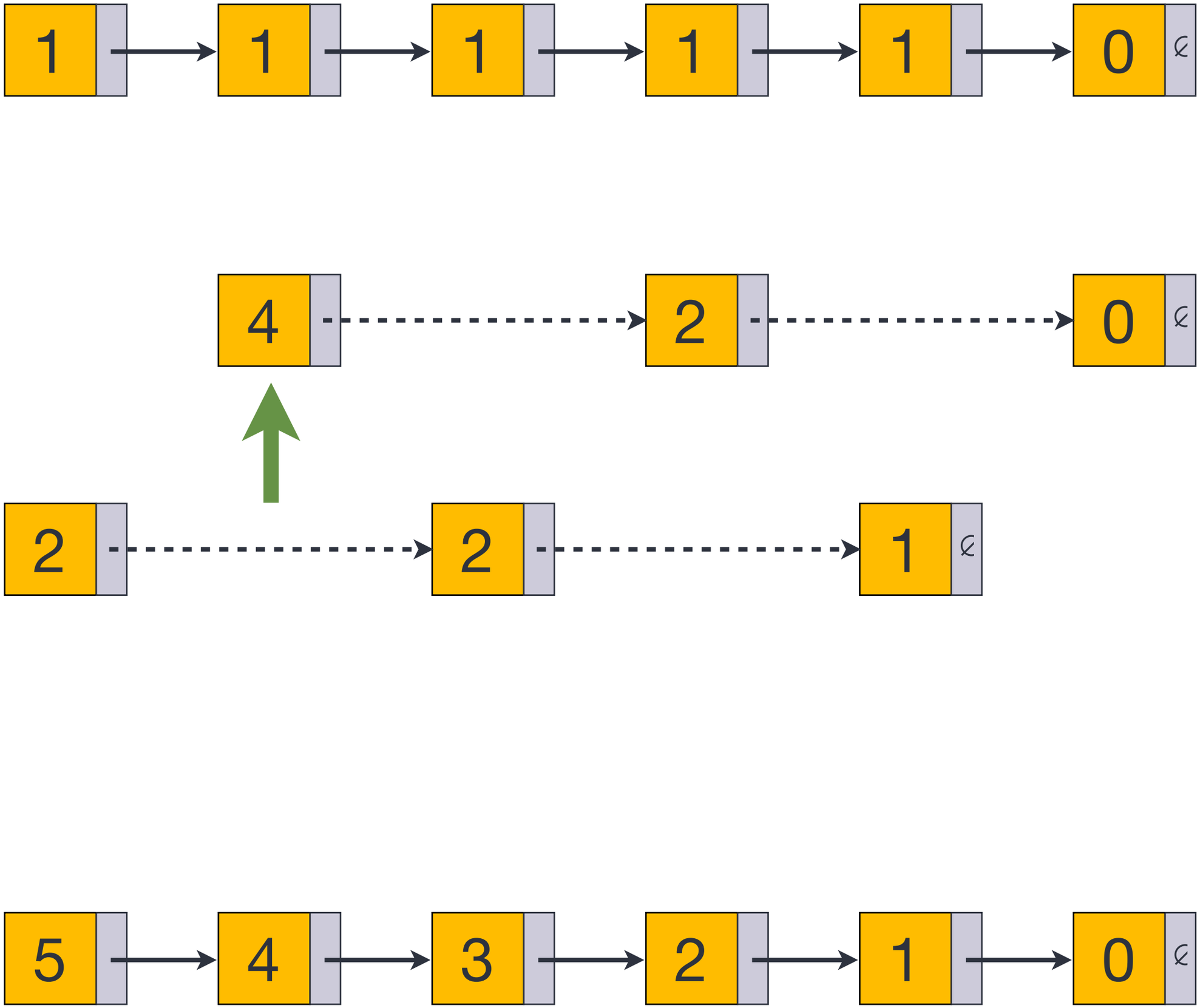
Parallelism!

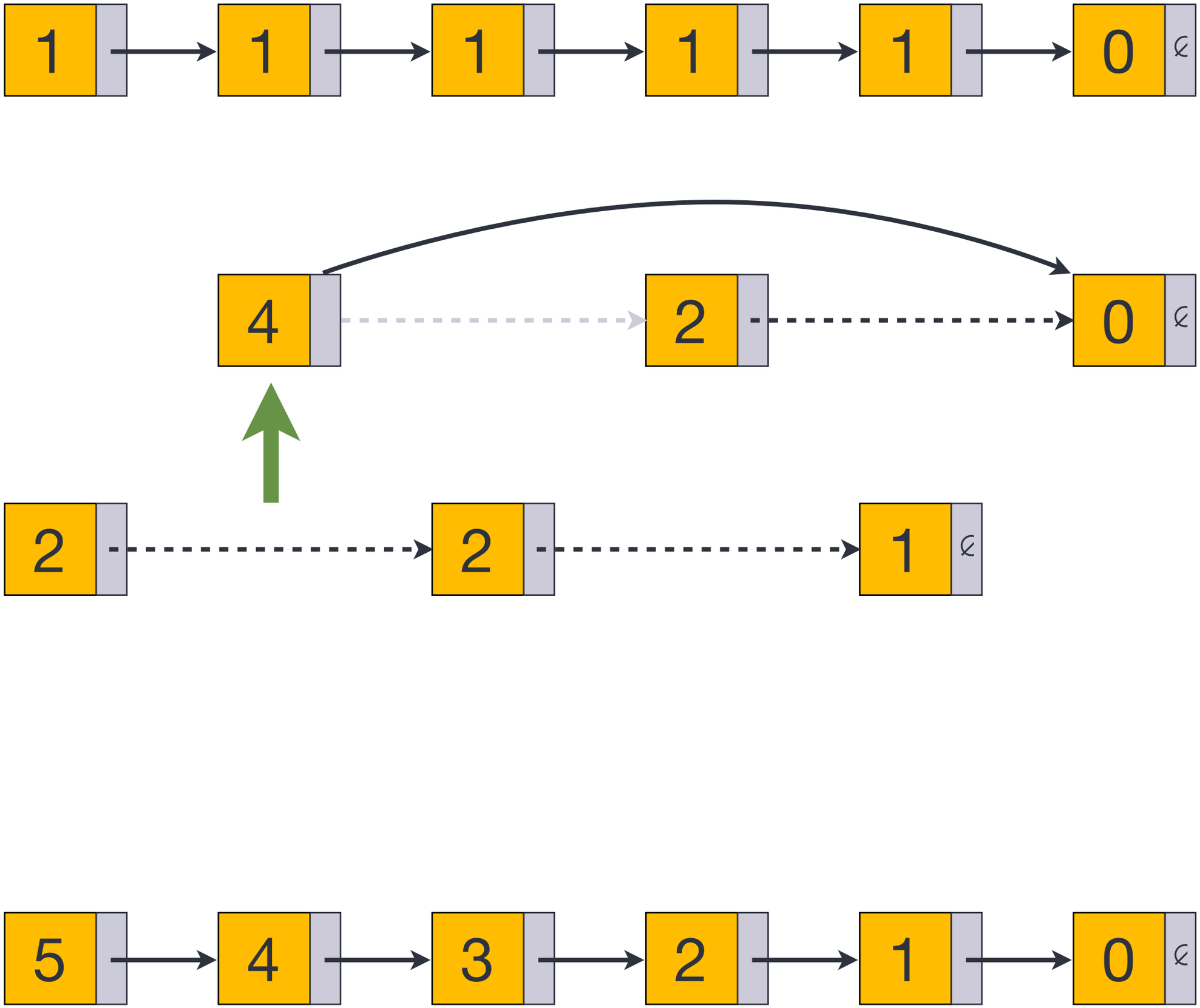


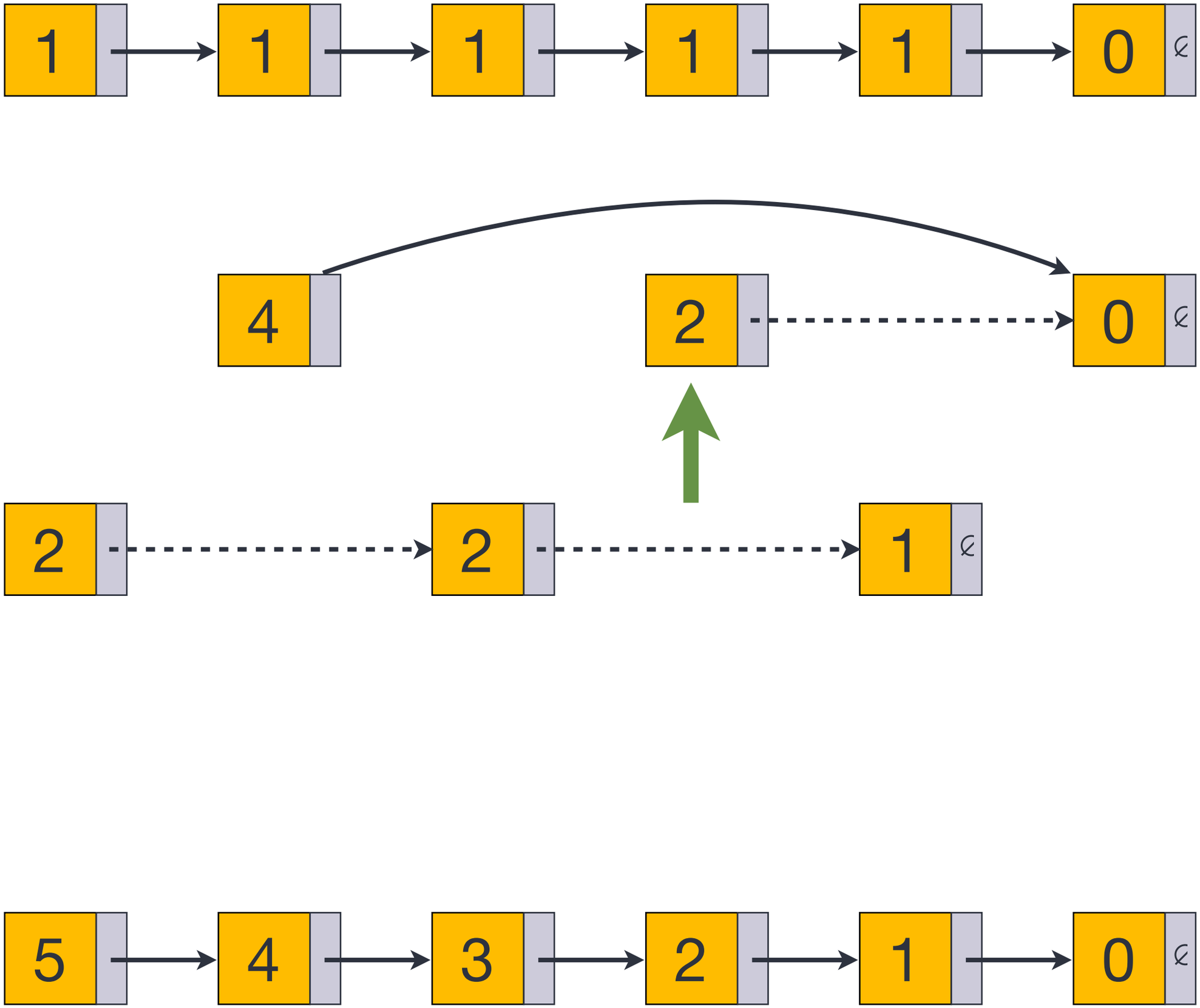

1
1
1
1
1
0
2
2
0
2
2
1
5
4
3
2
1
0

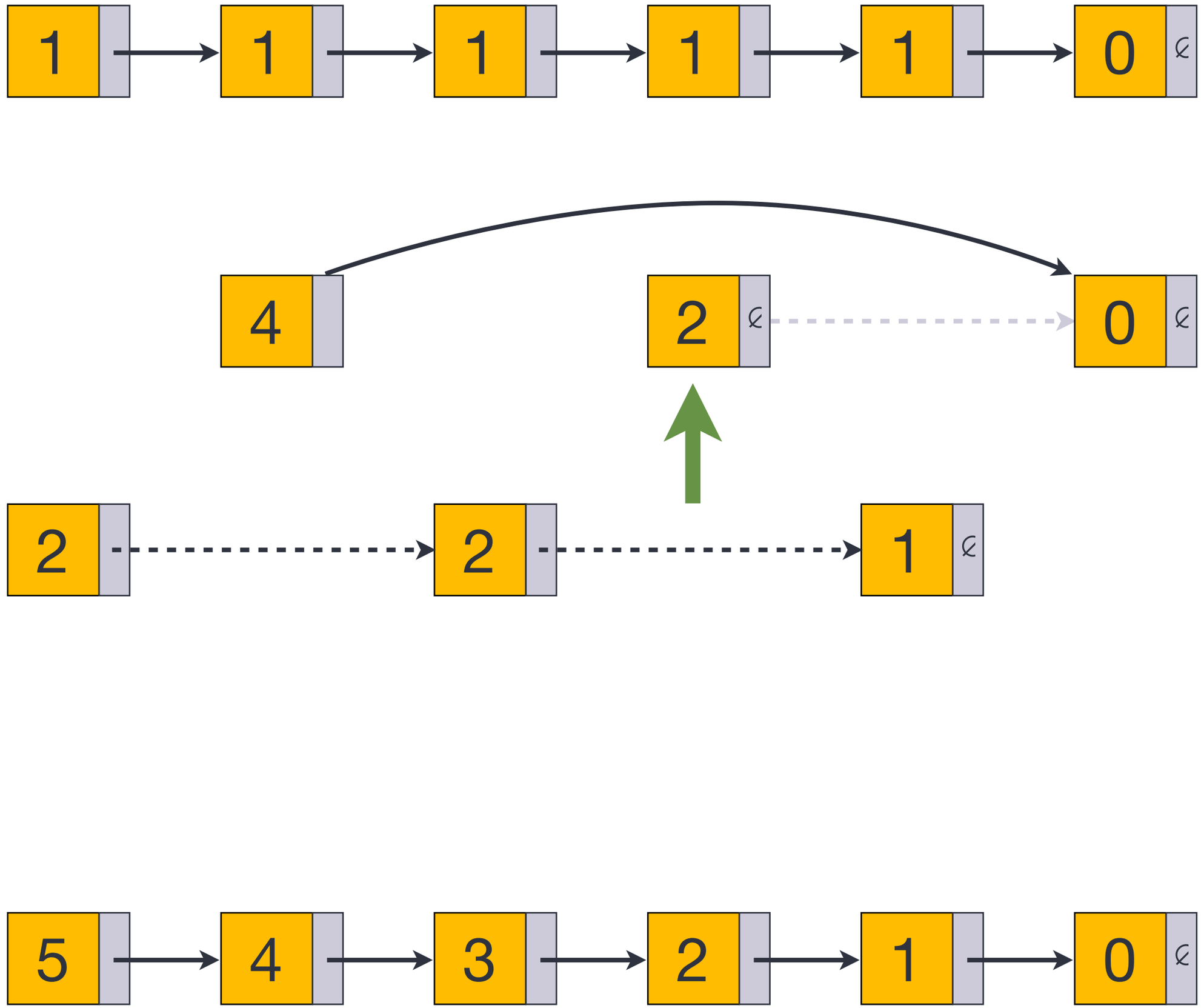

1
1
1
1
1
0
2
2
0
2
2
1
5
4
3
2
1
0

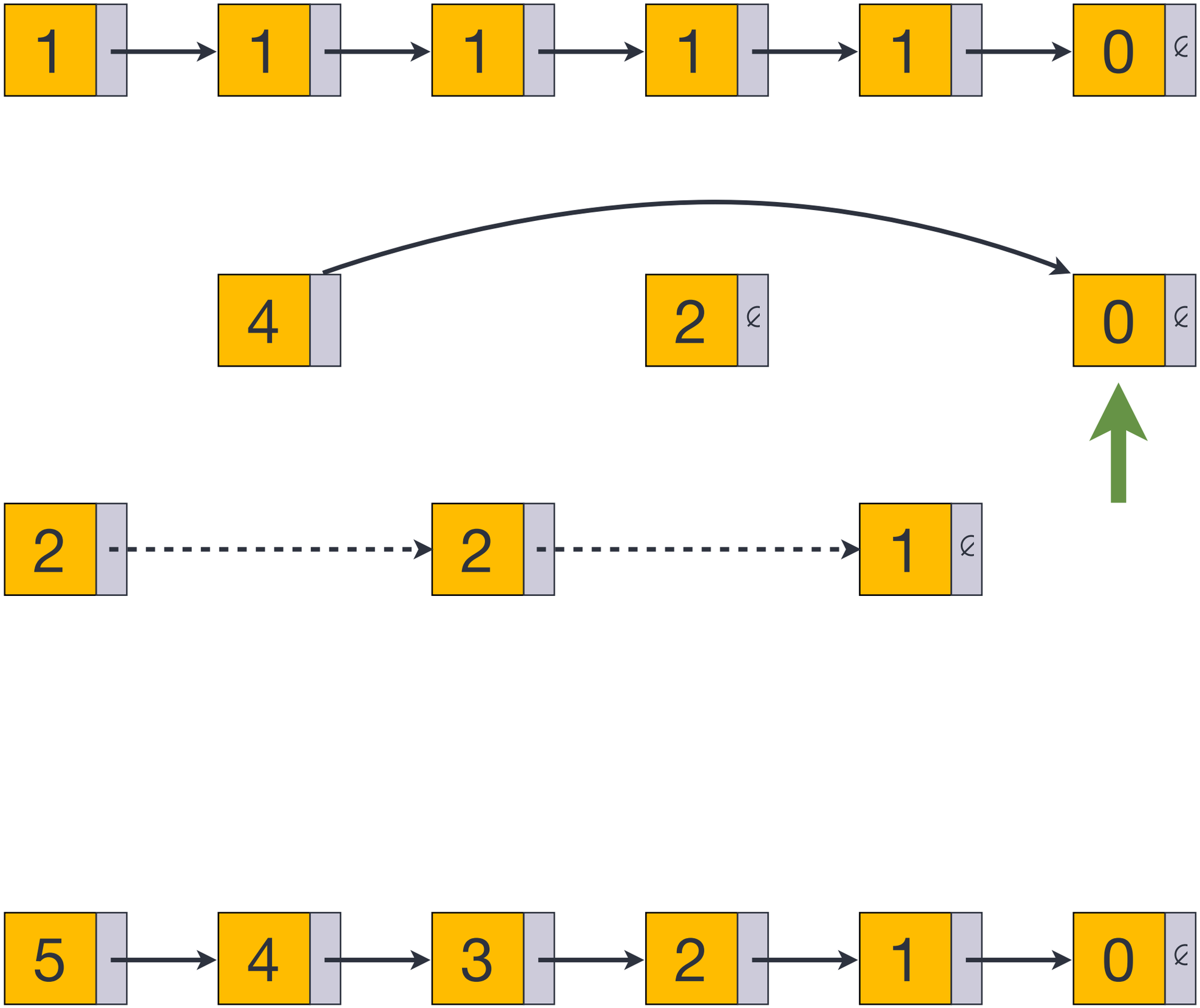

1
1
1
1
1
0
2
2
0
2
2
1
5
4
3
2
1
0

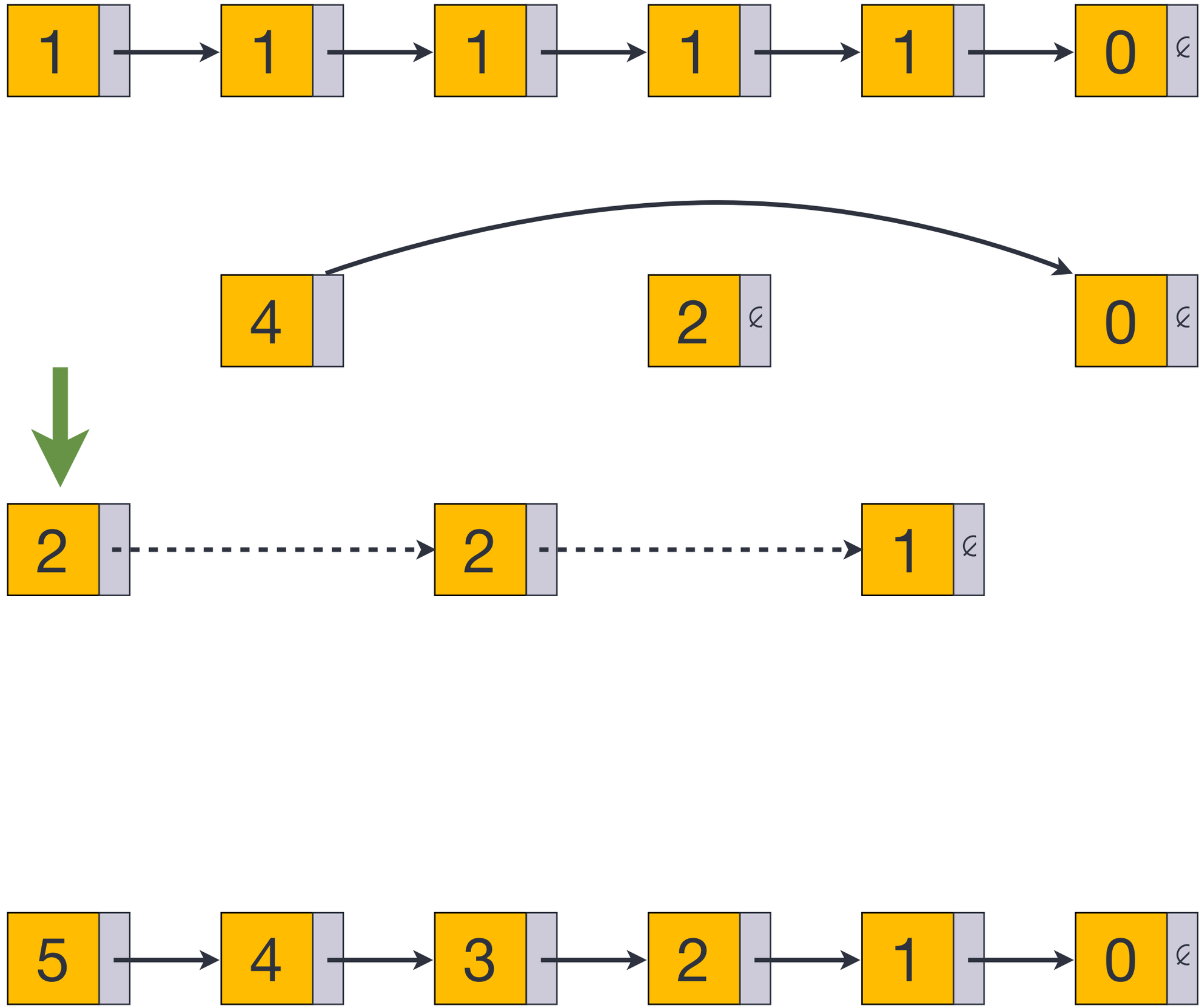

1
1
1
1
1
0
4
2
0
2
2
1
5
4
3
2
1
0

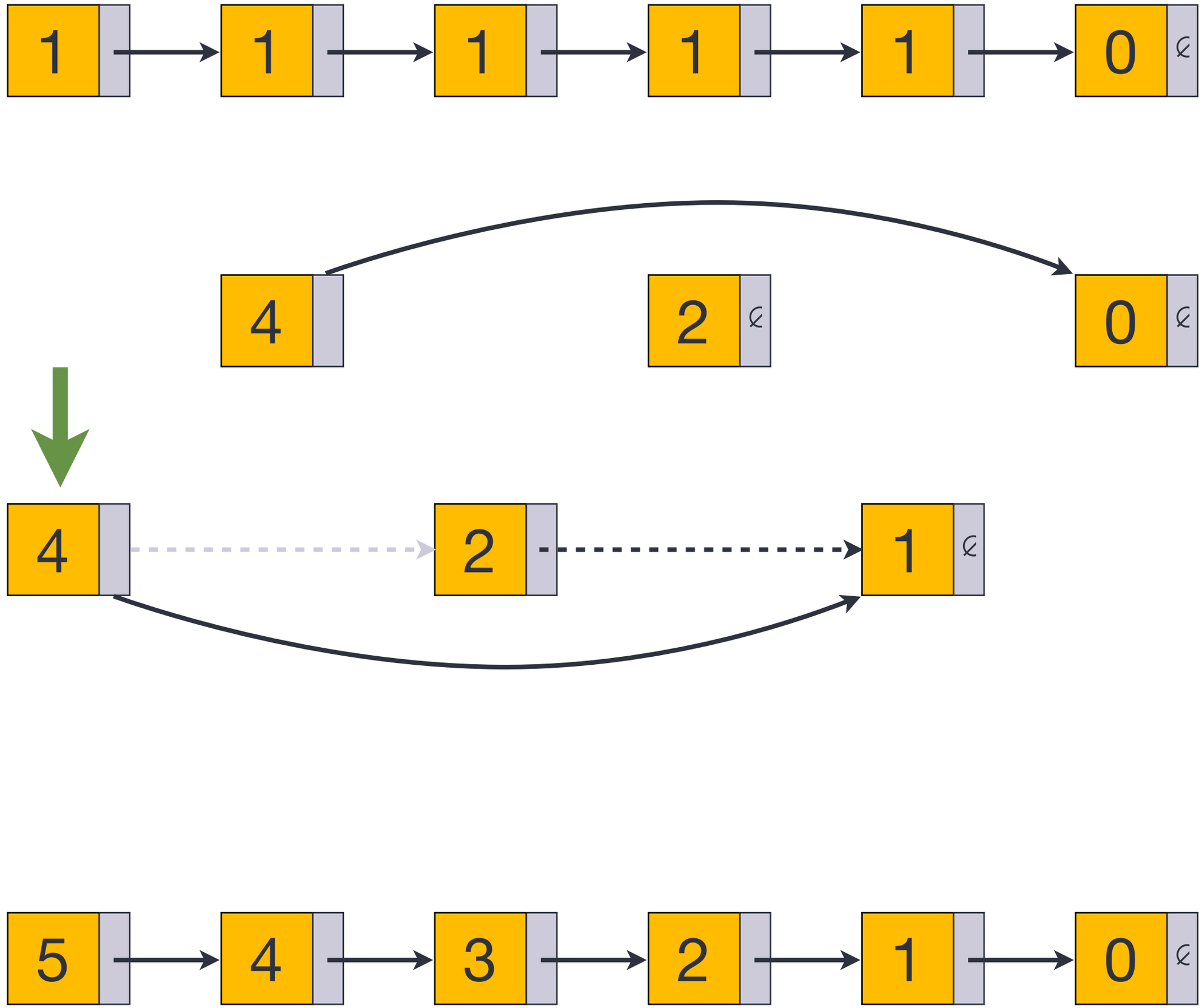

1
1
1
1
1
0
4
2
0
2
2
1
5
4
3
2
1
0

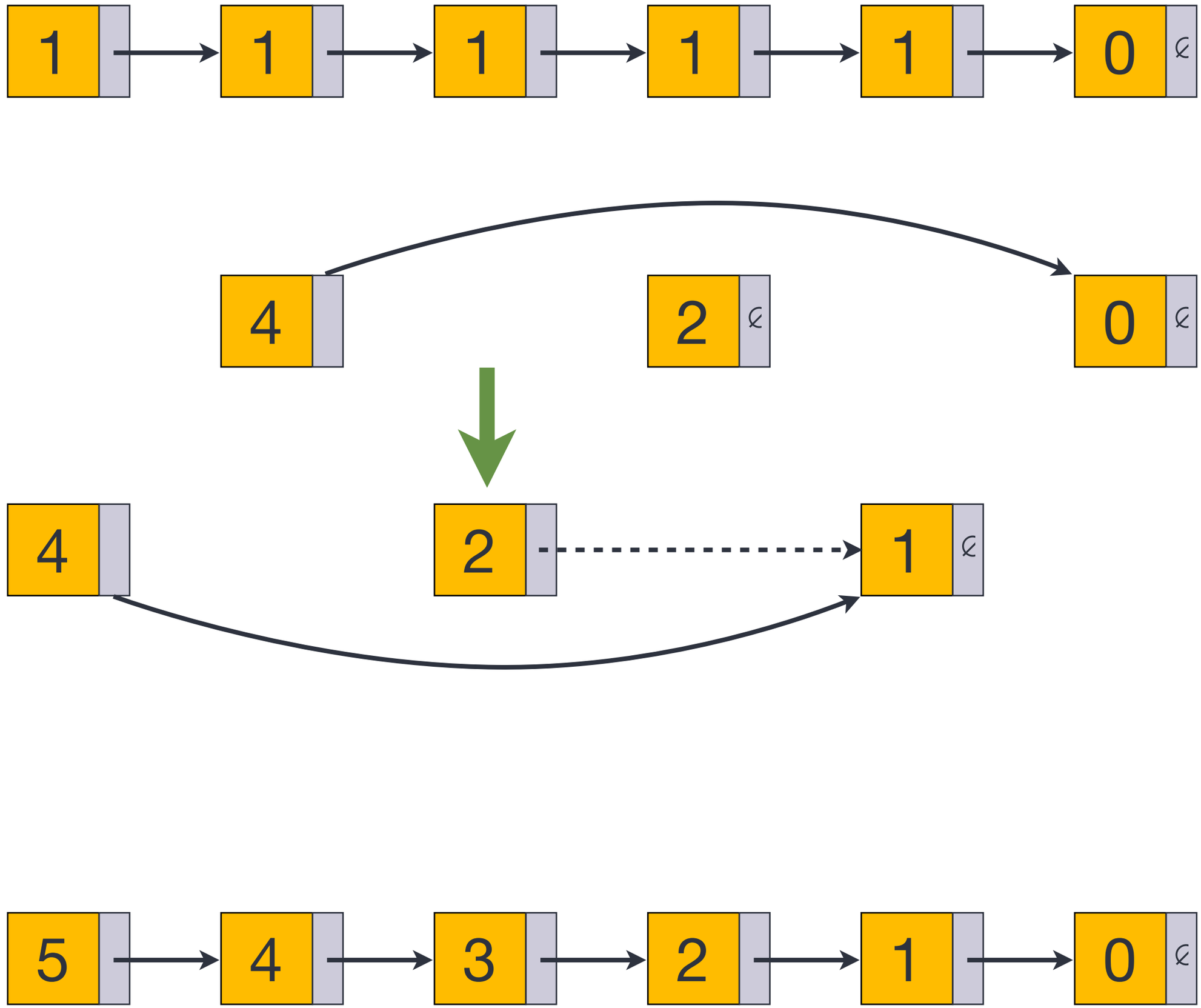

1
1
1
1
1
0
4
2
0
2
2
1
5
4
3
2
1
0

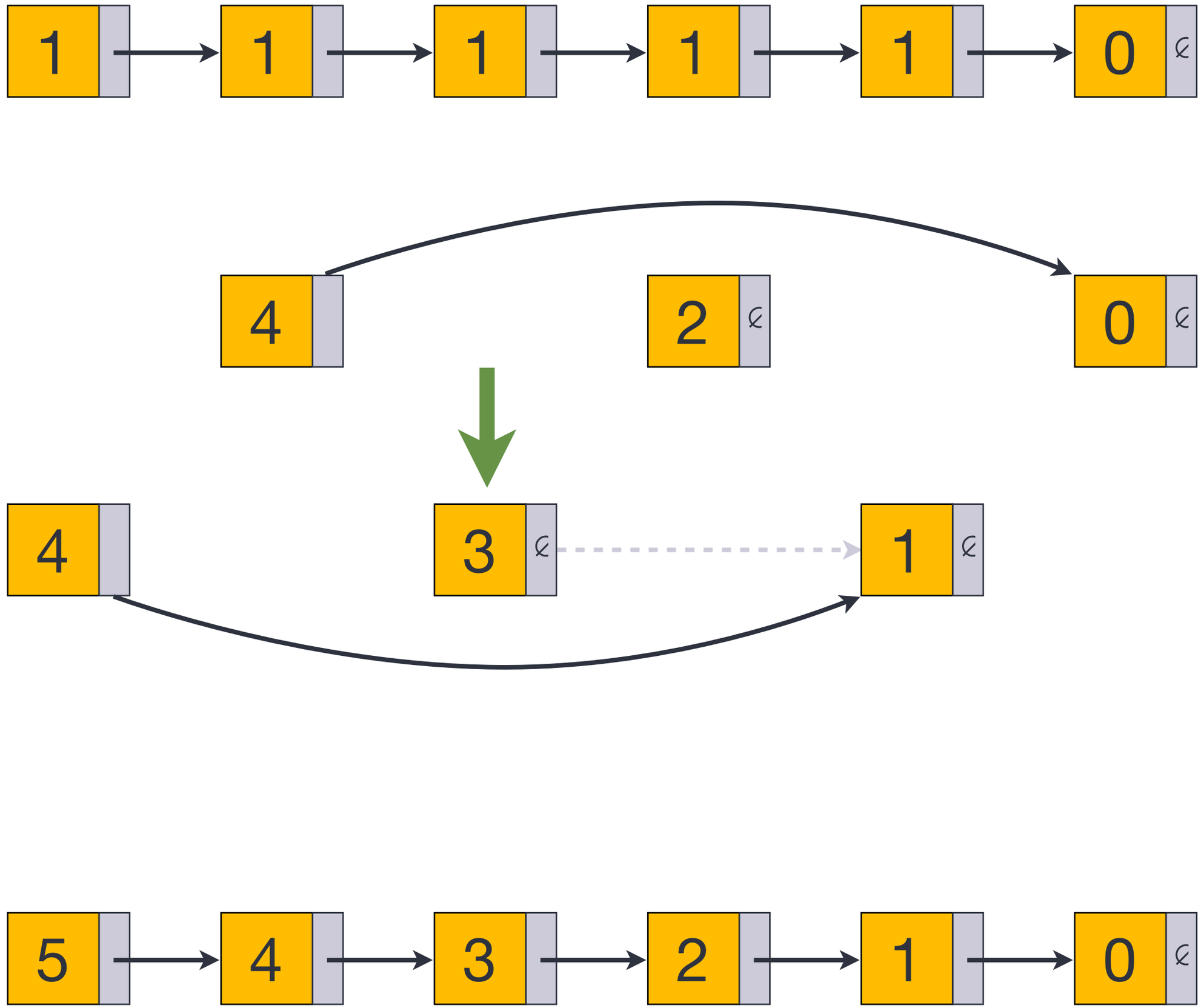

1
1
1
1
1
0
4
2
0
2
2
1
5
4
3
2
1
0

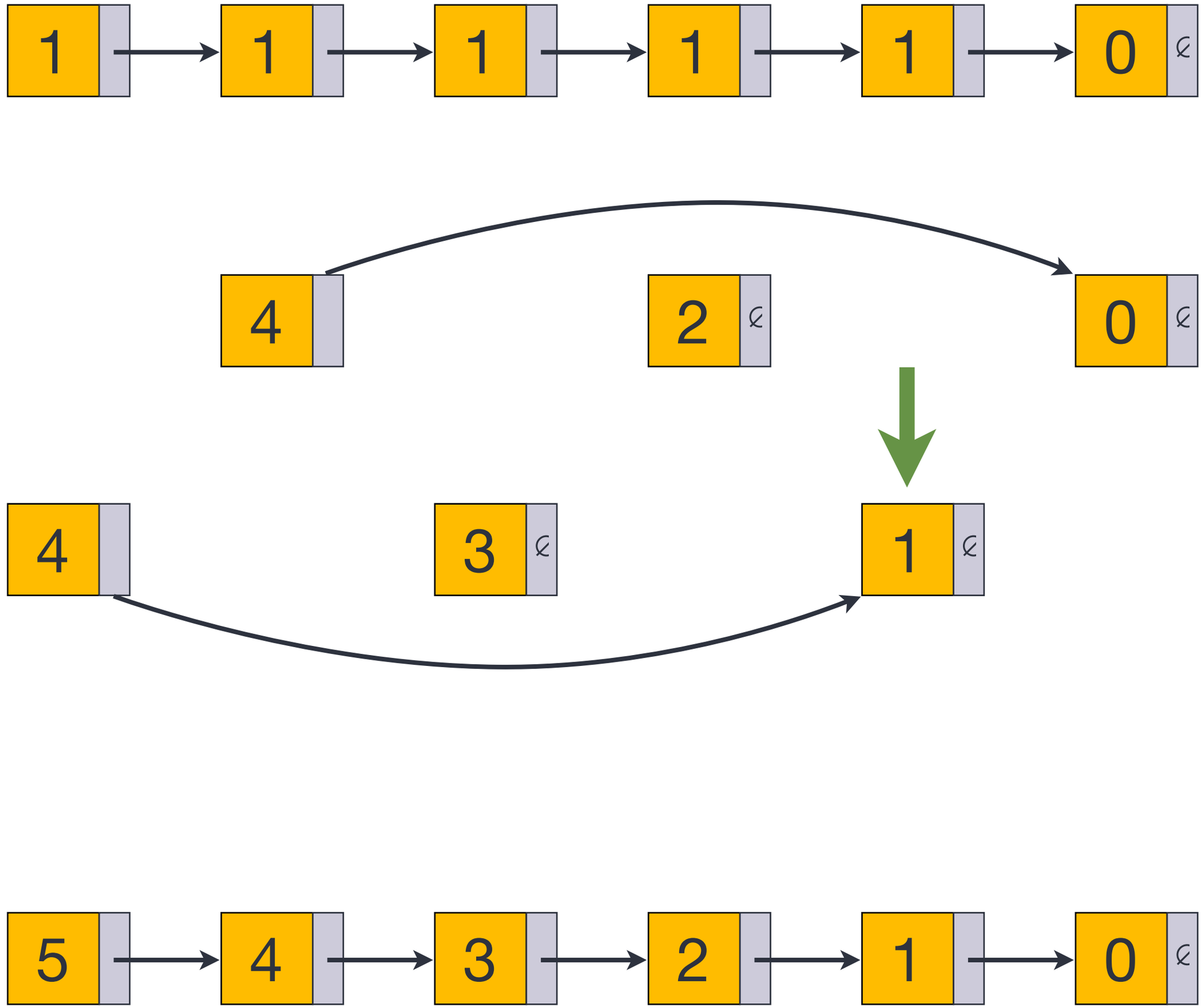

1
1
1
1
1
0
4
2
0
2
2
1
5
4
3
2
1
0

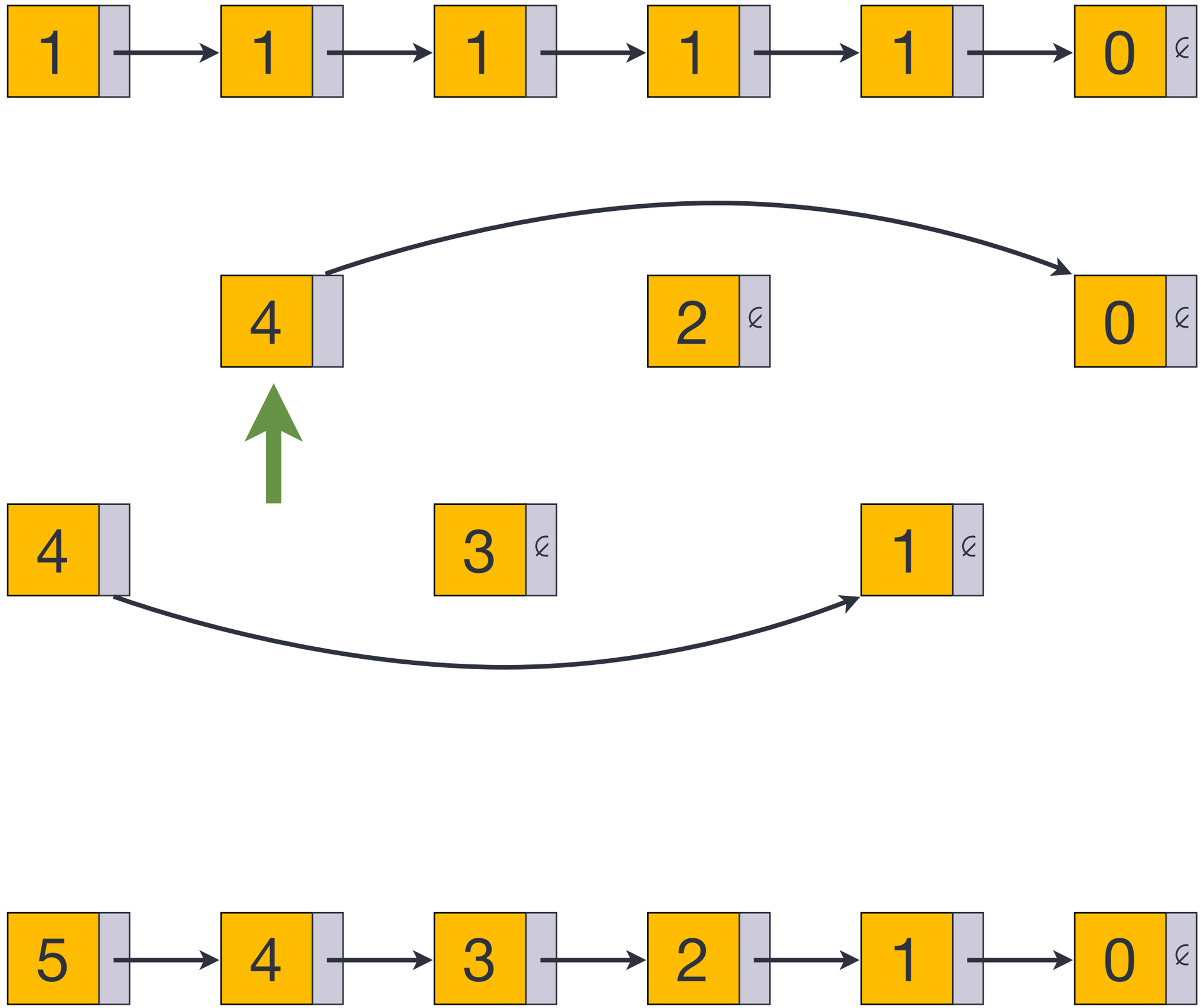

1
1
1
1
1
0
4
2
0
2
2
1
5
4
3
2
1
0

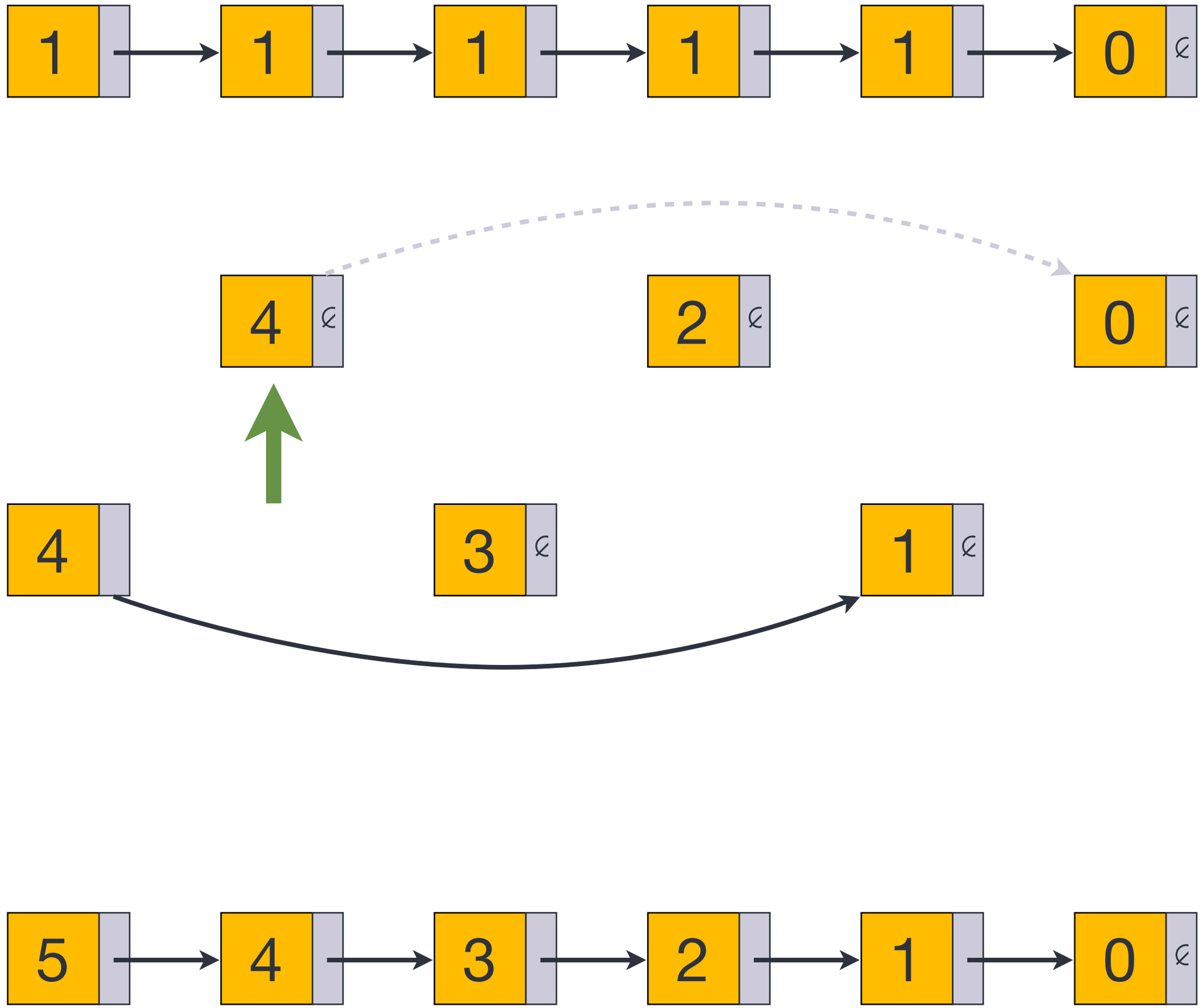

1
1
1
1
1
0
4
2
0
4
2
1
5
4
3
2
1
0

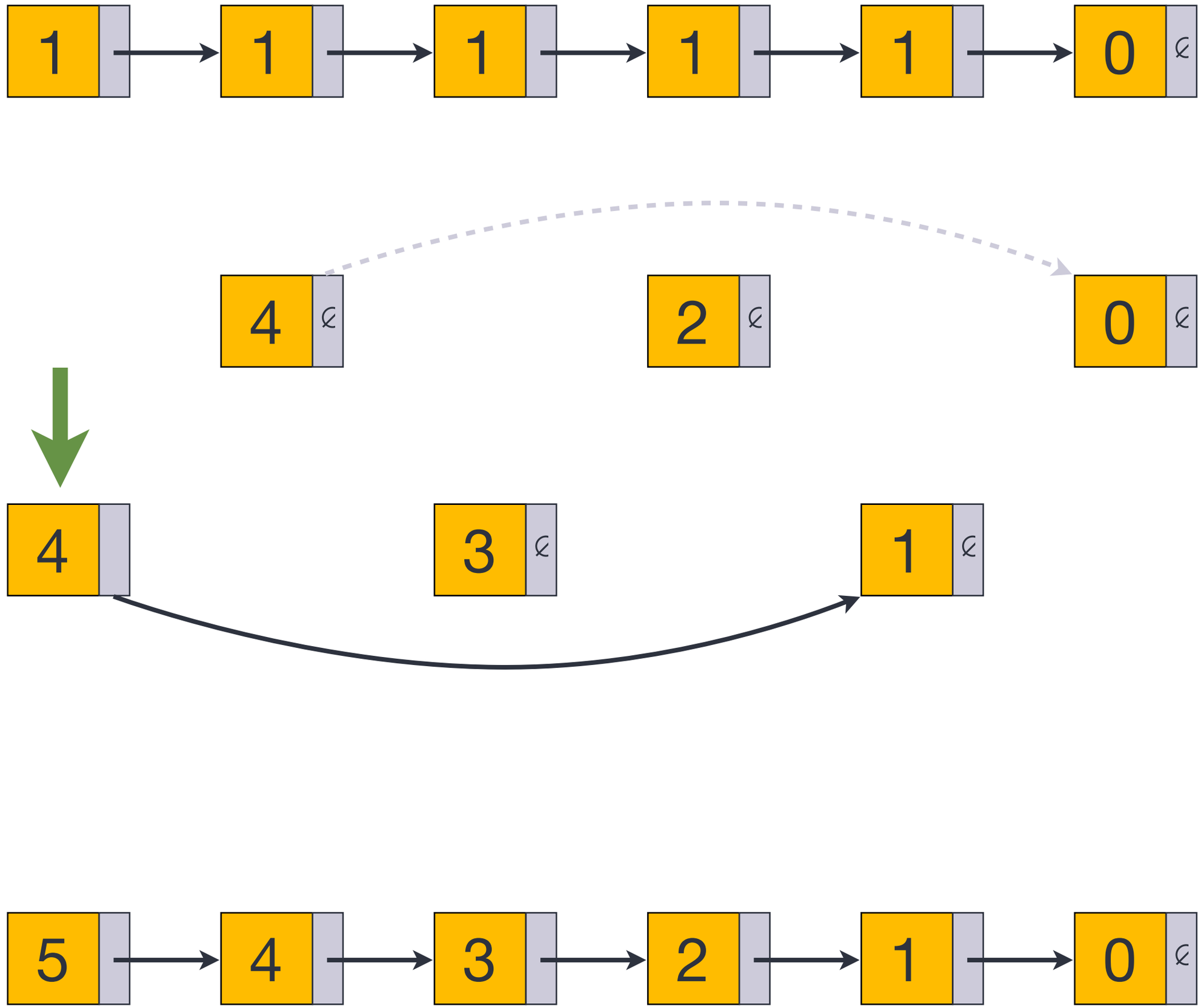

1
1
1
1
1
0
4
2
0
4
2
1
5
4
3
2
1
0

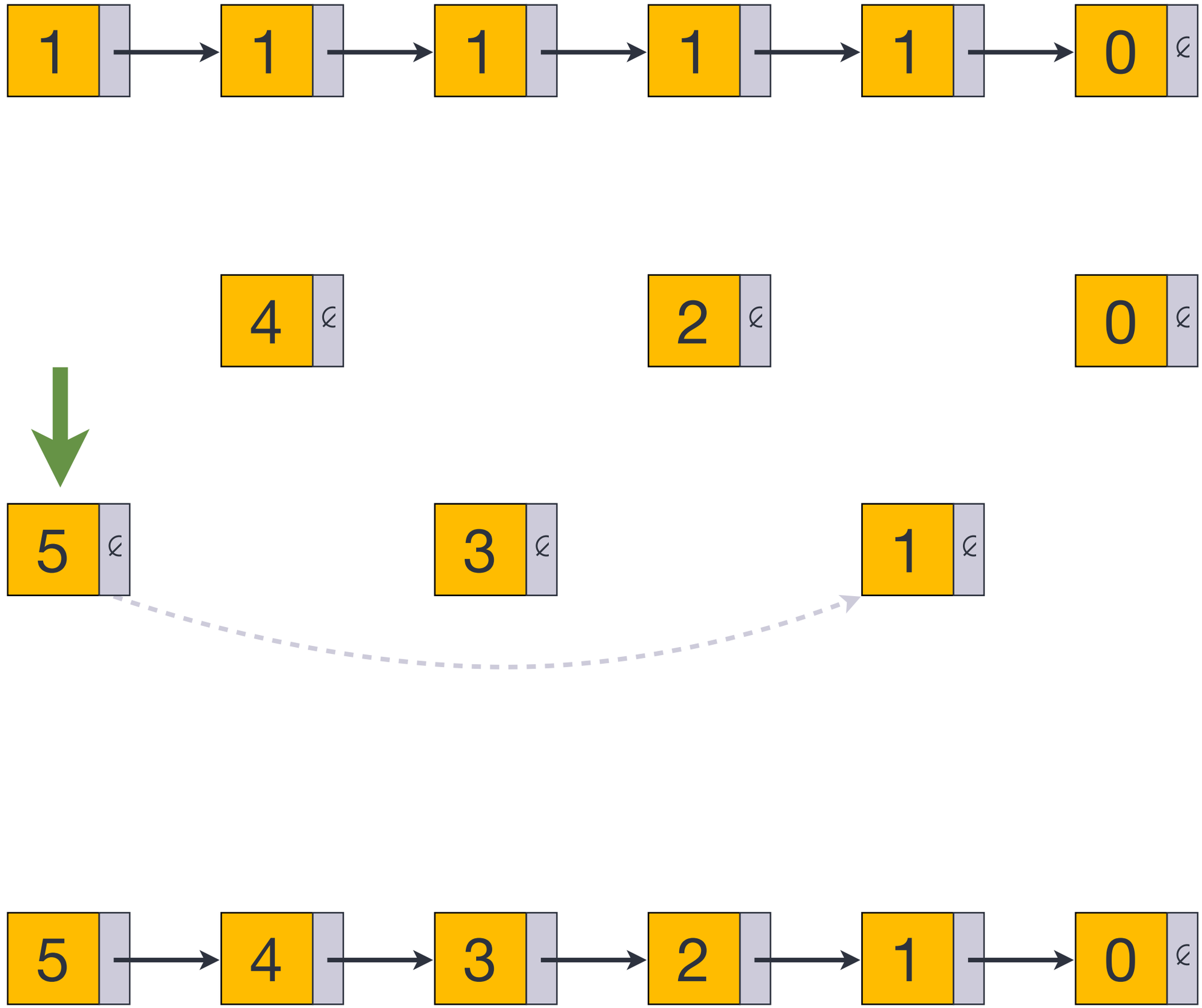

1
1
1
1
1
0
4
2
0
4
3
1
5
4
3
2
1
0

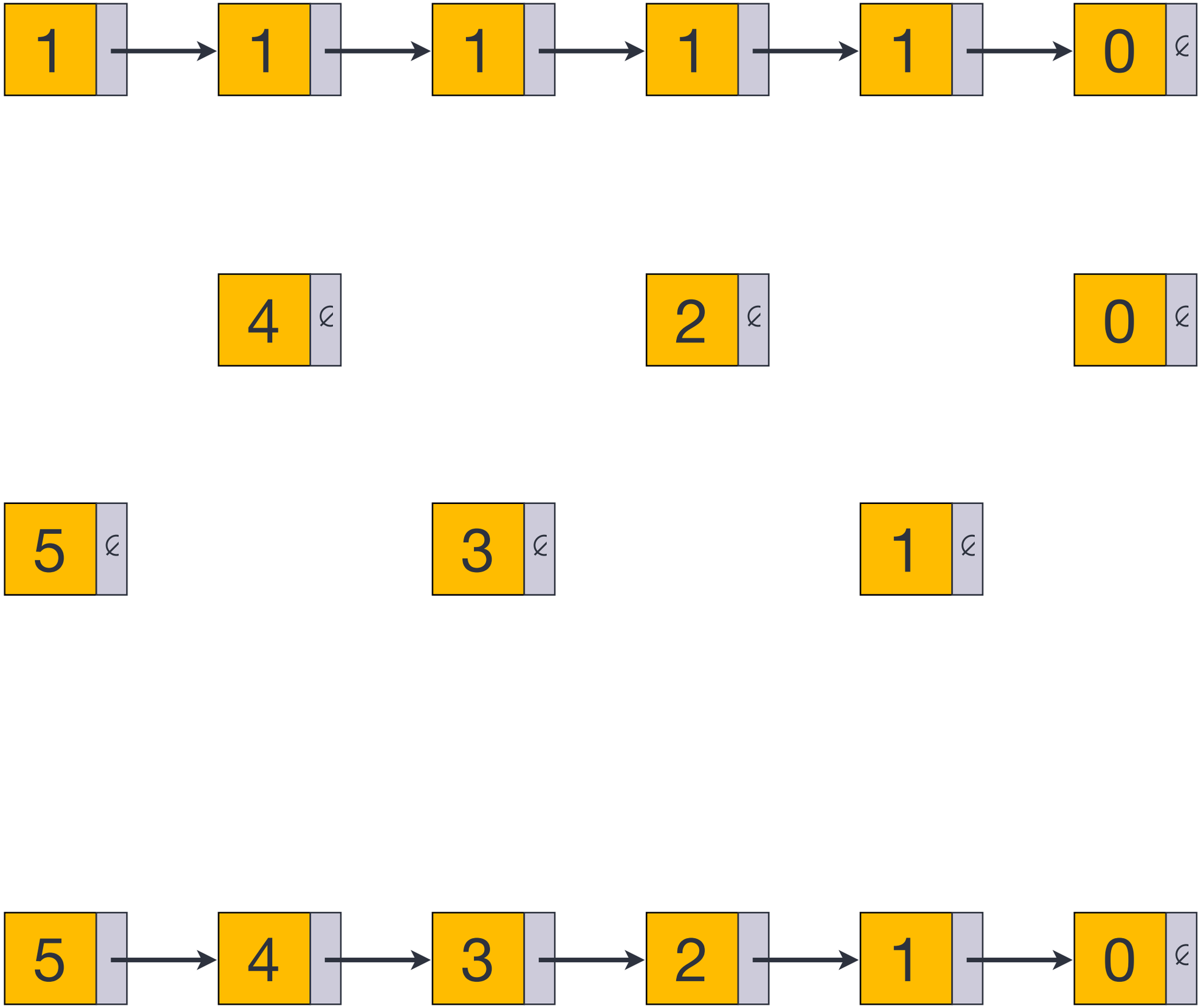

1
1
1
1
1
0
4
2
0
4
3
1
5
4
3
2
1
0

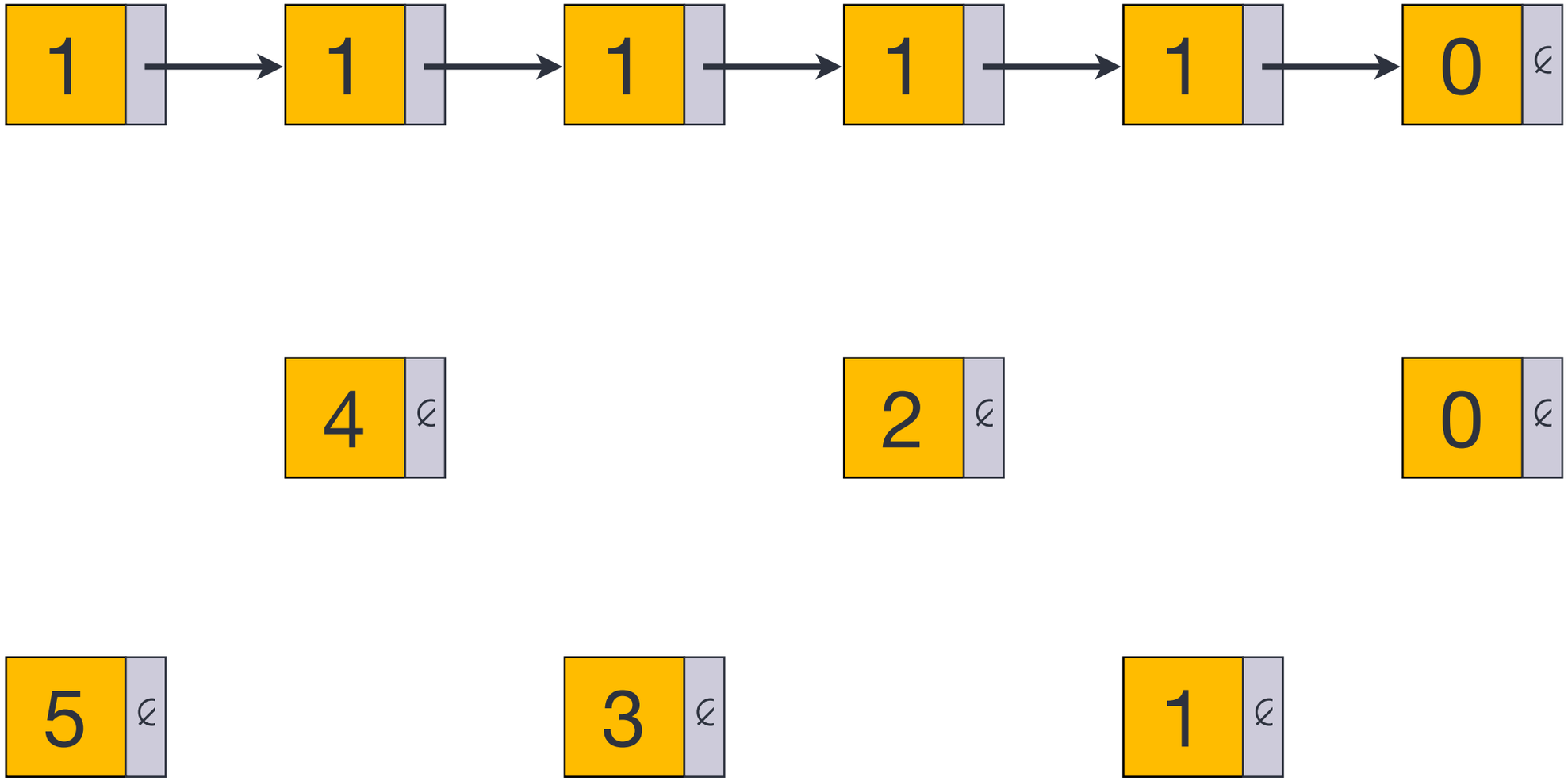

1
1
1
1
1
0
4
2
0
4
3
1
5
4
3
2
1
0


1
1
1
1
1
0
4
2
0
4
3
1
5
4
3
2
1
0

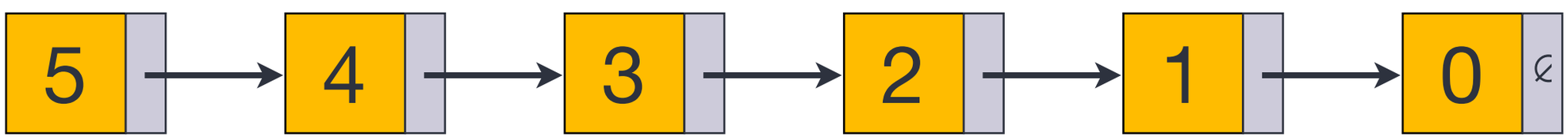

1
1
1
1
1
0
4
2
0
4
3
1
5
4
3
2
1
0


1
1
1
1
1
0
4
2
0
5
3
1
5
4
3
2
1
0

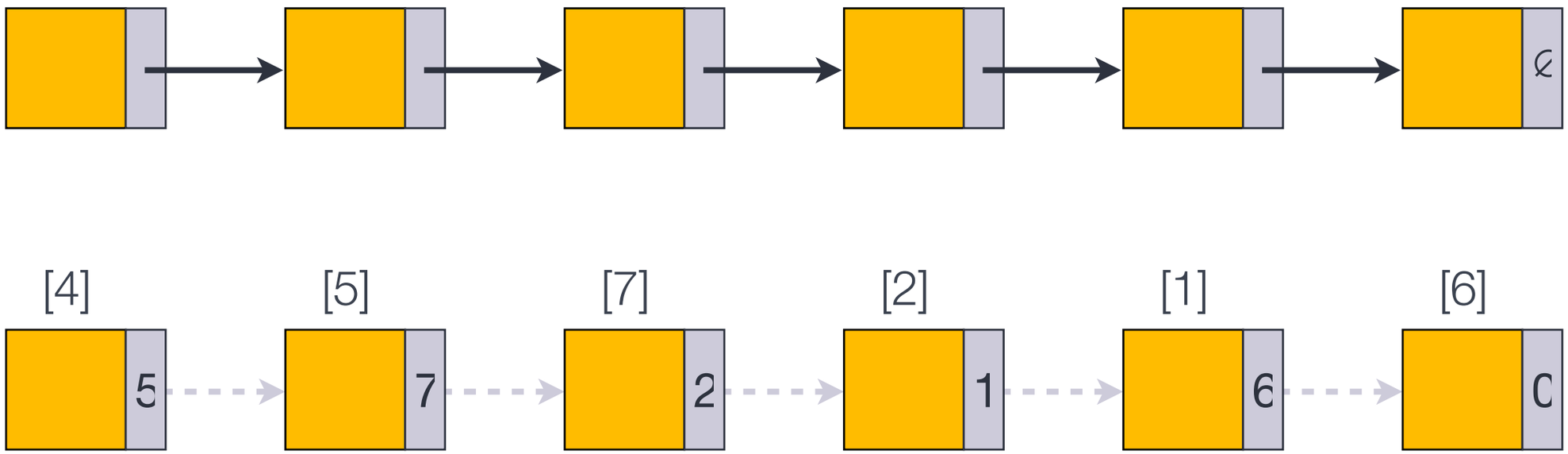

1
1
1
1
1
0
4
2
0
5
3
1
5
4
3
2
1
0



Depth? Work?

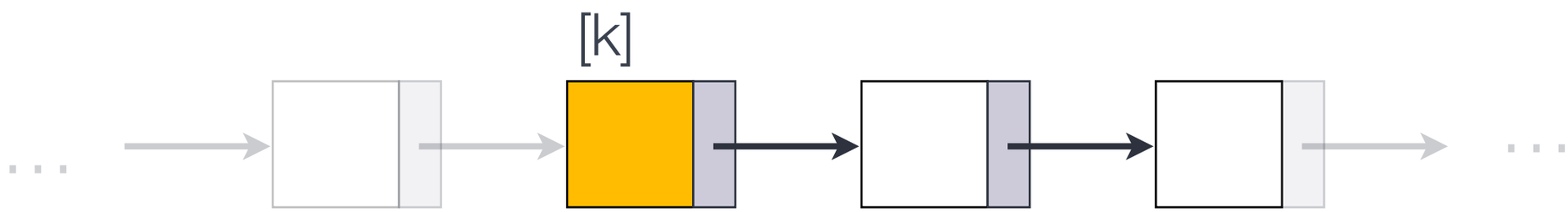


Recall: List storage

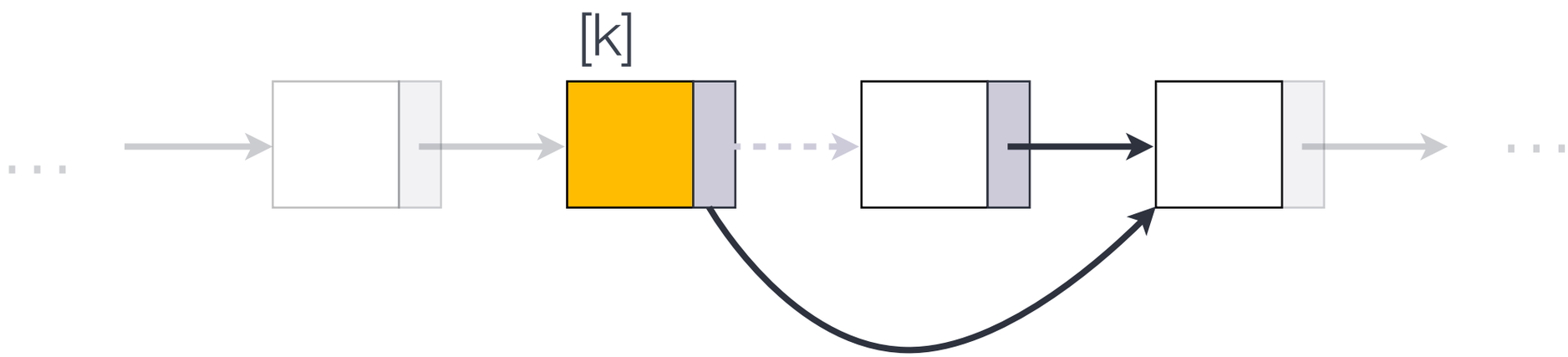


[1]	[2]	[3]	[4]	[5]	[6]	[7]	i = index
		?					V[i]
6	1	0	5	7	0	2	N[i]

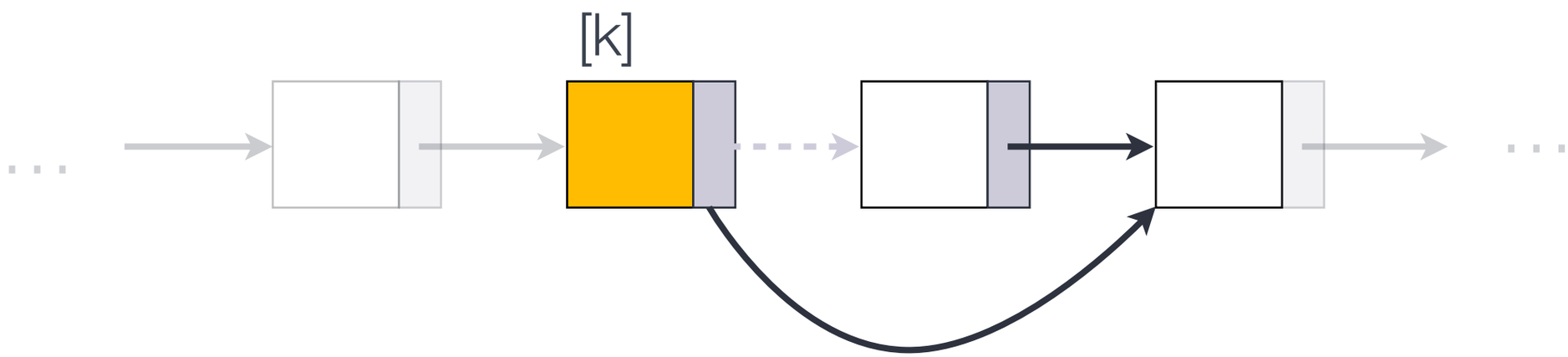
Recall: Jump



Recall: Jump

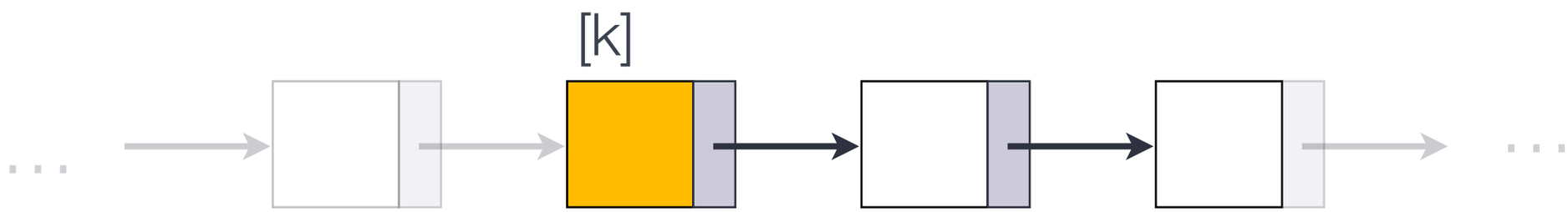


Recall: Jump

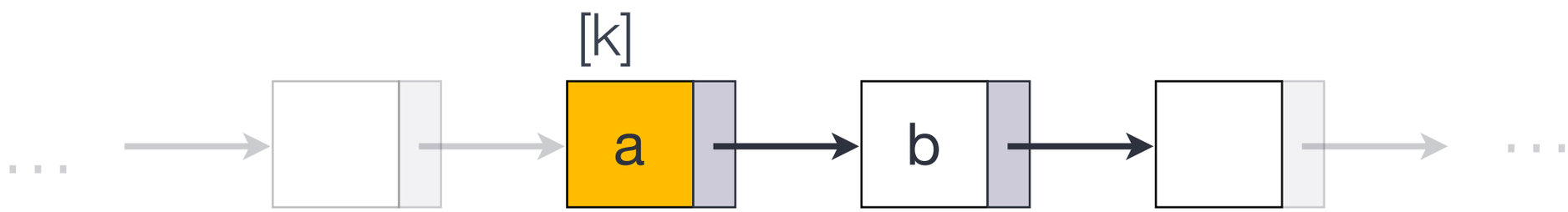


if N[k] ≠ ∅ then
  N[k] ← N[N[k]]

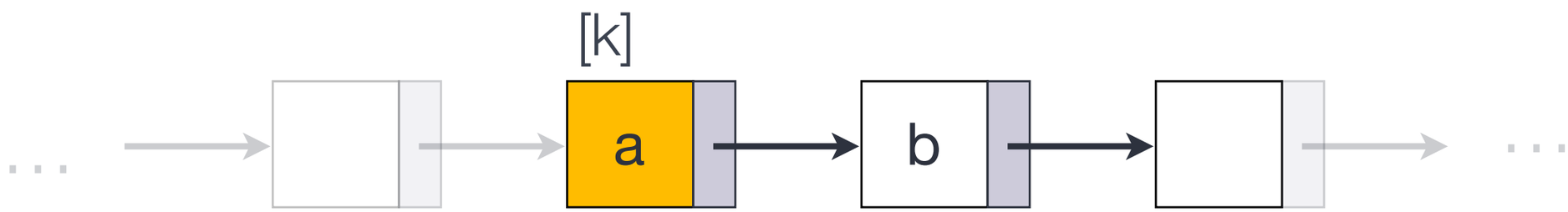
Recall: Jump + invariant update



Recall: Jump + invariant update

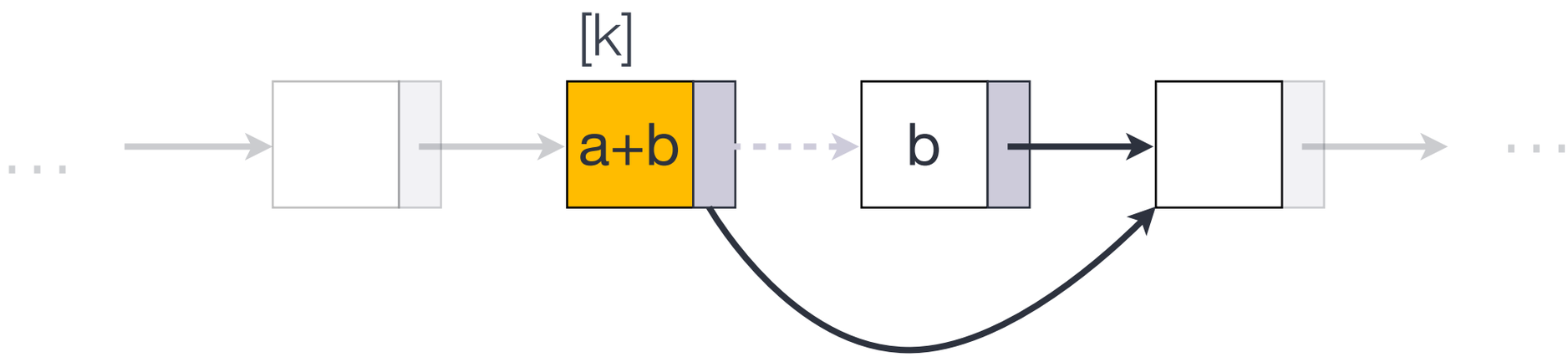


Recall: Jump + invariant update



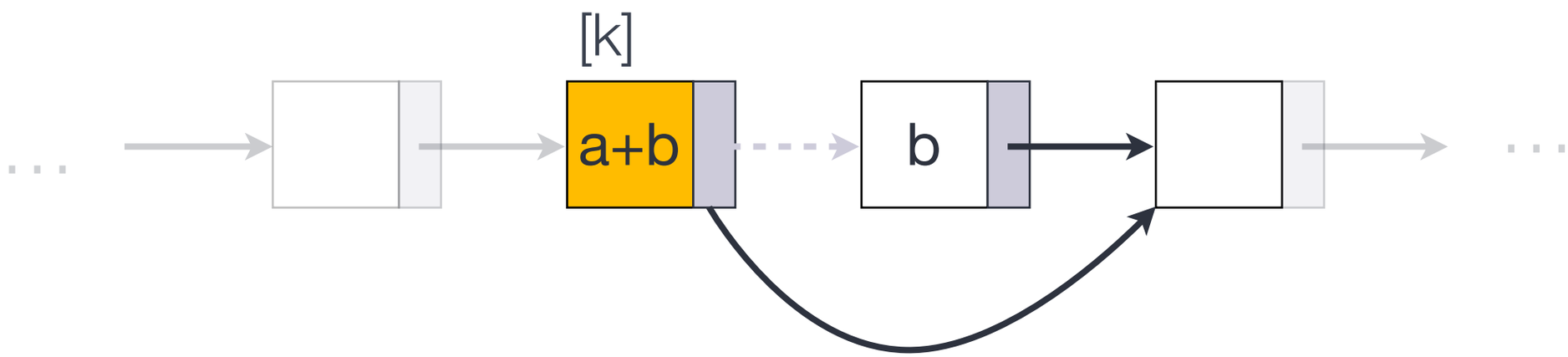
if N[k] ≠ ∅ then
  N[k] ← N[N[k]]

Recall: Jump + invariant update



if N[k] ≠ ∅ then
  N[k] ← N[N[k]]

Recall: Jump + invariant update



if N[k] ≠ ∅ then
  V[k] ← V[k] + V[N[k]]
  N[k] ← N[N[k]]

```
parallel-for k ← 1 to n do
   if N[k] ≠ ∅
      then V[k] ← 1
      else V[k] ← 0

while any N[k] ≠ ∅ do
   parallel-for k ← 1 to n do
      if N[k] ≠ ∅ then
         V[k] ← V[k] + V[N[k]]
         N[k] ← N[N[k]]
```

Note: Assumes synchronization!

Simple parallel list ranking algorithm

```
parallel-for k ← 1 to n do
   if N[k] ≠ ∅
      then V[k] ← 1
      else V[k] ← 0

while any N[k] ≠ ∅ do
   parallel-for k ← 1 to n do
      if N[k] ≠ ∅ then
         V[k] ← V[k] + V[N[k]]
         N[k] ← N[N[k]]
```

Note: Assumes synchronization!

```
parallel-for k ← 1 to n do
  if N[k] ≠ ∅
    then V[k] ← 1
    else V[k] ← 0

while any N[k] ≠ ∅ do
  parallel-for k ← 1 to n do
    if N[k] ≠ ∅ then
      V[k] ← V[k] + V[N[k]]
      N[k] ← N[N[k]]
```

Note: Assumes synchronization!

```
parallel-for k ← 1 to n do
  if N[k] ≠ ∅
    then V[k] ← 1
    else V[k] ← 0
// Implicit barrier here
while any N[k] ≠ ∅ do
  parallel-for k ← 1 to n do
    if N[k] ≠ ∅ then
      n_cur ← N[k]
      v_next ← V[n_cur]
      n_next ← N[n_cur]
      barrier;
      V[k] ← V[k] + v_next
      N[k] ← n_next
  // Implicit barrier here
```