ALGORITMI E STRUTTURE DATI

Prof. Manuela Montangero

A.A. 2022/23

STRUTTURE DATI: liste, pile e code

UNIMORE
UNIVERSITÀ DEGLI STUDI DI MODENA E REGGIO EMILIA

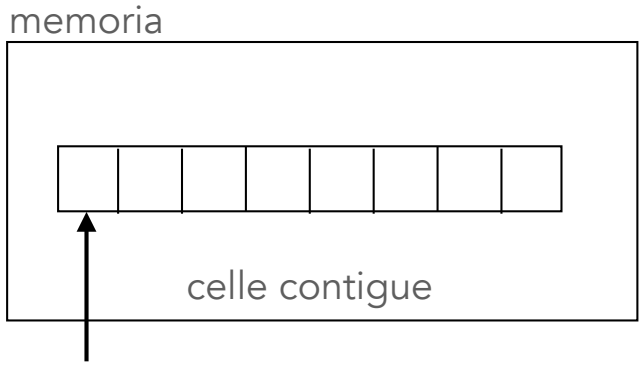
Strutture Dati

SEQUENZA LINEARE

Elementi disposti consecutivamente, associati alla loro posizione

ACCESSO DIRETTO

ARRAY



nome Array = posizione 1a cella

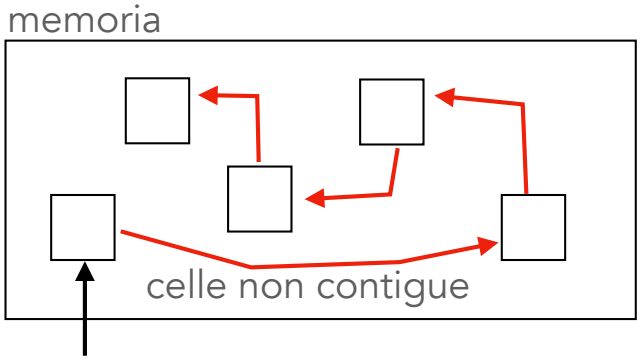
accesso cella indice *i* = posizione 1a cella + *i* $O(1)$

inserzione o cancellazione di elementi in testa $O(n)$

SEQUENZE STATICHE

ACCESSO SEQUENZIALE

LISTA



nome Lista = posizione 1o elemento

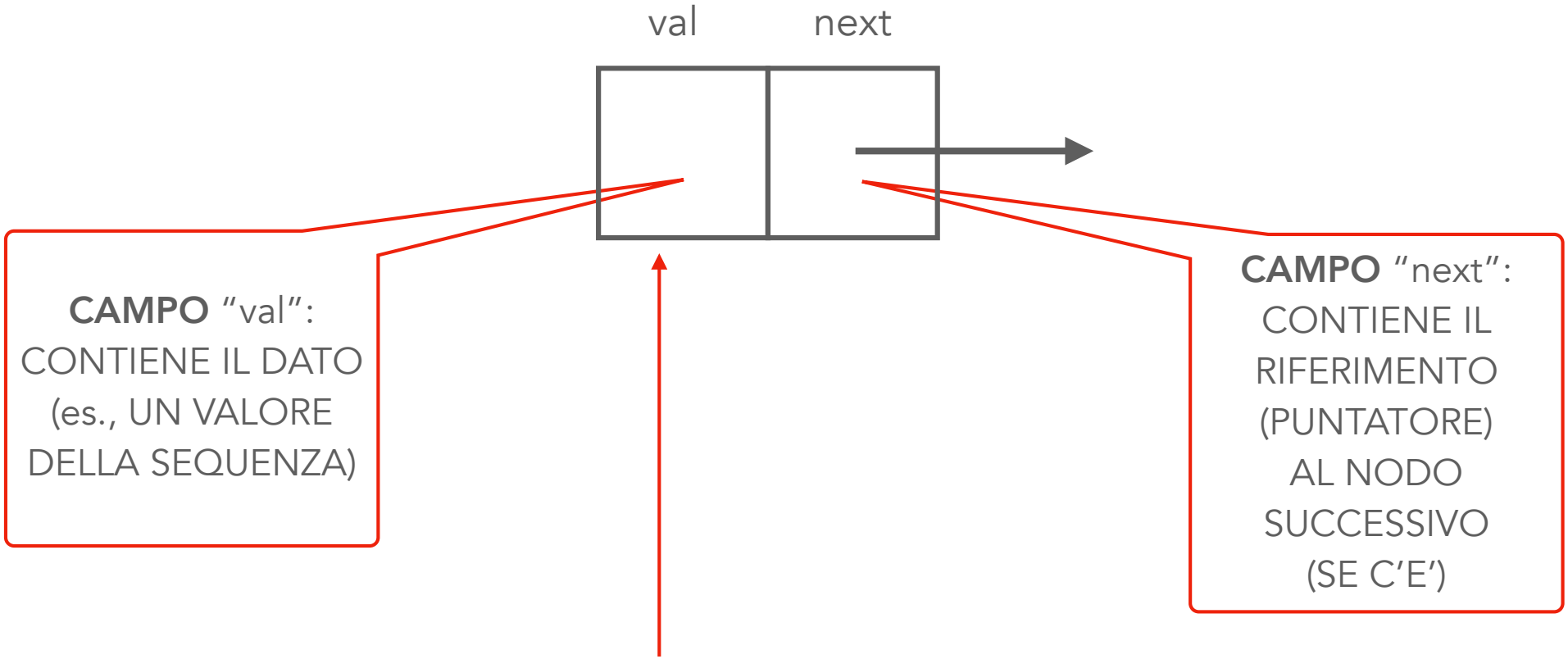
accesso all' *i*-esimo elemento = posizione 1a elemento + scansione altri elementi fino ad *i* $O(i)$

inserzione o cancellazione di elementi in testa $O(1)$

SEQUENZE DINAMICHE

LISTE

NODO



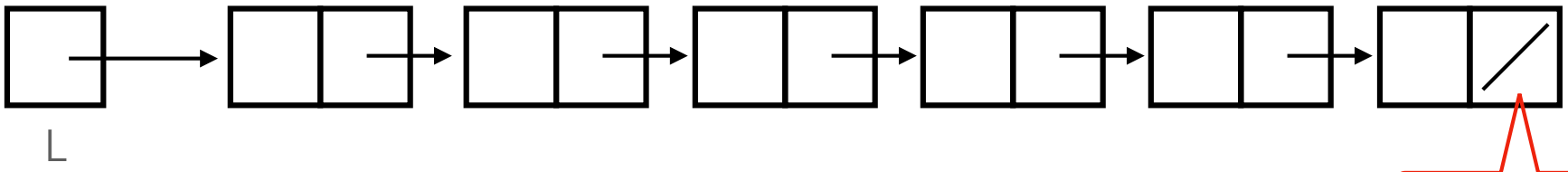
DATO UN PUNTATORE ***e*** AD UN NODO DELLA LISTA

e.val è il dato associato a tale nodo (es. valore della sequenza)

e.next è il **puntatore** al nodo successivo

LISTA (struttura dati elementare)

LISTA: puntatore al primo nodo

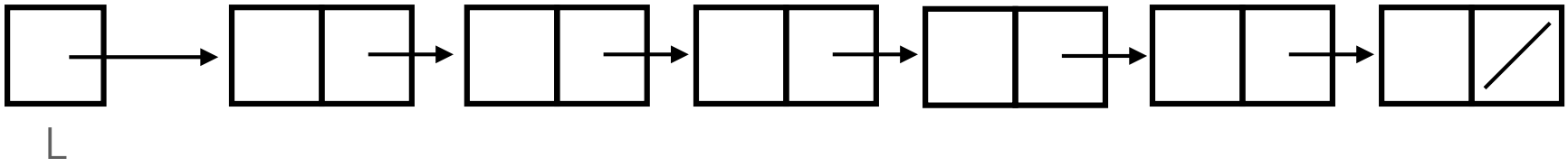


Puntatore NIL (o NULL), non c'e' un nodo successivo

PRIMITIVE della lista:

1. Crea una nuova lista vuota: `new_list()`
2. Test lista vuota: `is_empty_list(L)`
3. Inserimento di un nodo in testa: `insert_head(L,e)`
4. Inserimento di un nodo dopo un nodo dato: `insert_next(p,e)`
5. Ricerca del nodo in posizione *i*: *`search(L,i)`*
6. Inserimento di un nodo in posizione *i*: *`insert_pos(L,e,i)`*
7. Cancellazione di un nodo dato: `delete(L,p)`

LISTA: implementazione primitive



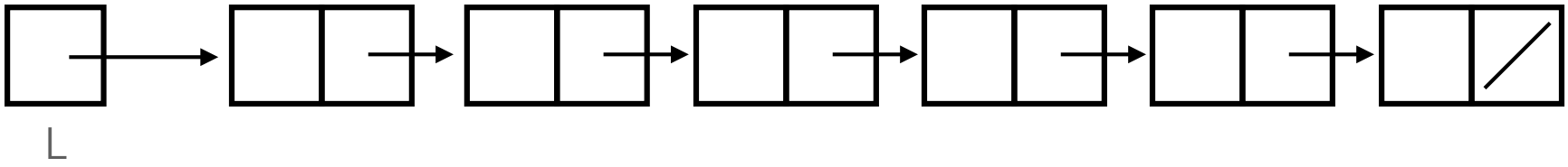
1. **Crea una nuova lista**: restituisce un puntatore ad una lista vuota

È una FUNZIONE!

```
new_list()
  return NIL
```

Costo computazionale
$\Theta(1)$

LISTA: implementazione primitive



2. **Test lista vuota**: restituisce VERO se la lista è vuota, FALSO altrimenti

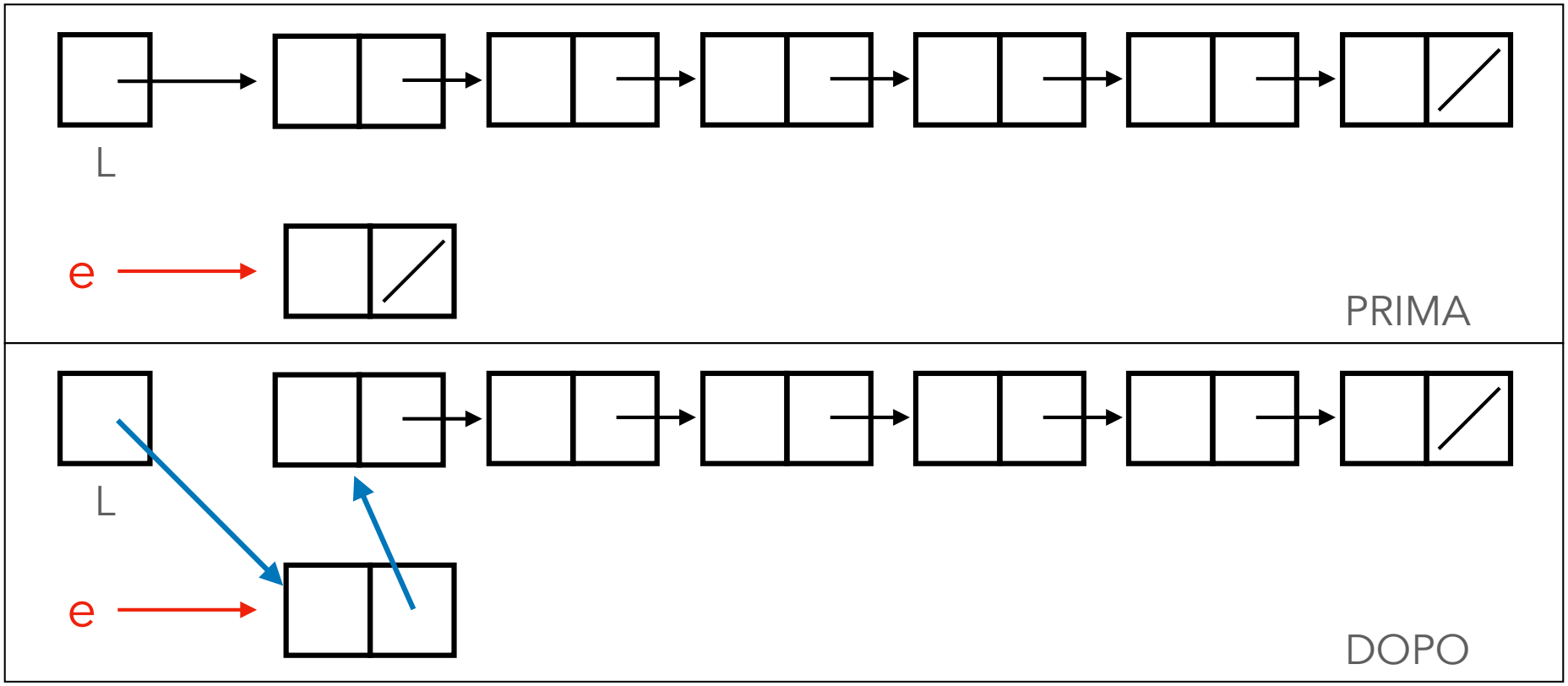
È una FUNZIONE!

```
is_empty_list(L)
  return L = NIL
```

Costo computazionale
$\Theta(1)$

LISTA: implementazione primitive

3. **Inserimento** di un nodo in testa (dato un puntatore **e** al nodo)




```
insert_head(L,e)
 e.next := L
 L := e
```

Costo computazionale

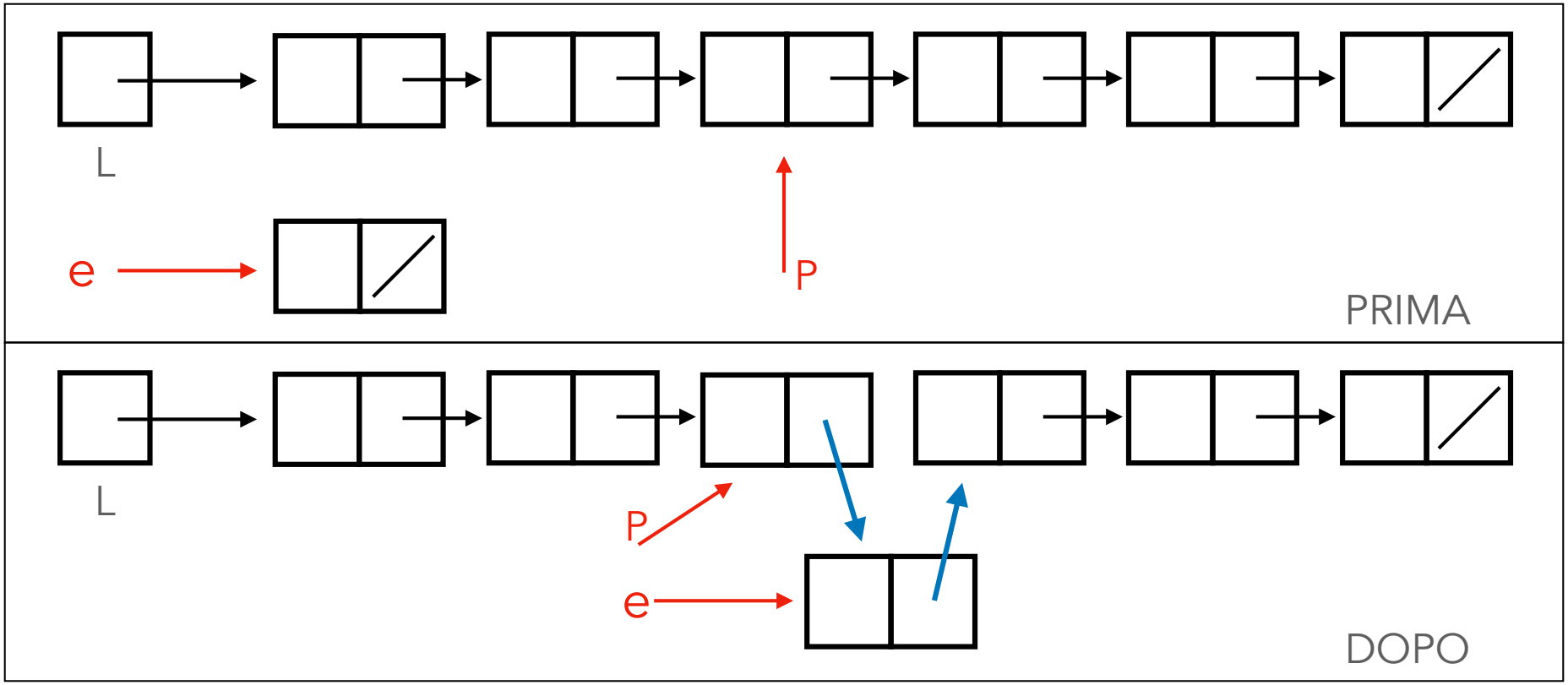
$\Theta(1)$

Funziona anche se L è vuota

È una PROCEDURA!

LISTA: implementazione primitive

4. **Inserimento** di un nodo dopo un nodo dato (dati un puntatore **e** al nodo da inserire e uno al nodo **p** che lo deve precedere nella lista)




```
insert_next(p,e)
 e.next := p.next
 p.next := e
```

Costo computazionale

$\Theta(1)$

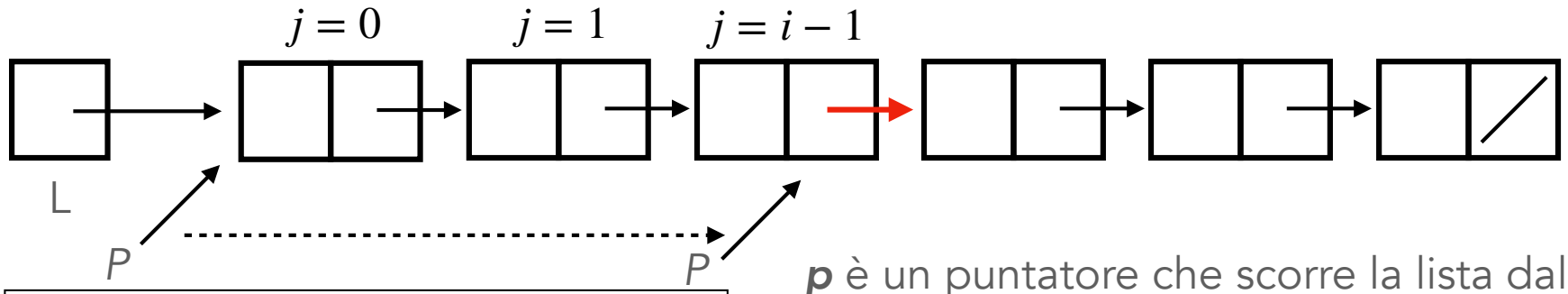
È una PROCEDURA!

LISTA: implementazione primitive

5. **Ricerca** del nodo in posizione **i**: restituisce il puntatore al nodo o NIL se non lo trova (contando le posizioni partendo da zero)

Restituisce il puntatore nel campo *next* del nodo in posizione (i-1)

j conta le posizioni



p è un puntatore che scorre la lista dalla testa fino a quando punta al nodo i-1

```
search(L,i)
 p := L
 J := 0
 while (j < i AND p ≠ NIL)
  p := p.next
  J := j + 1
 return p
```

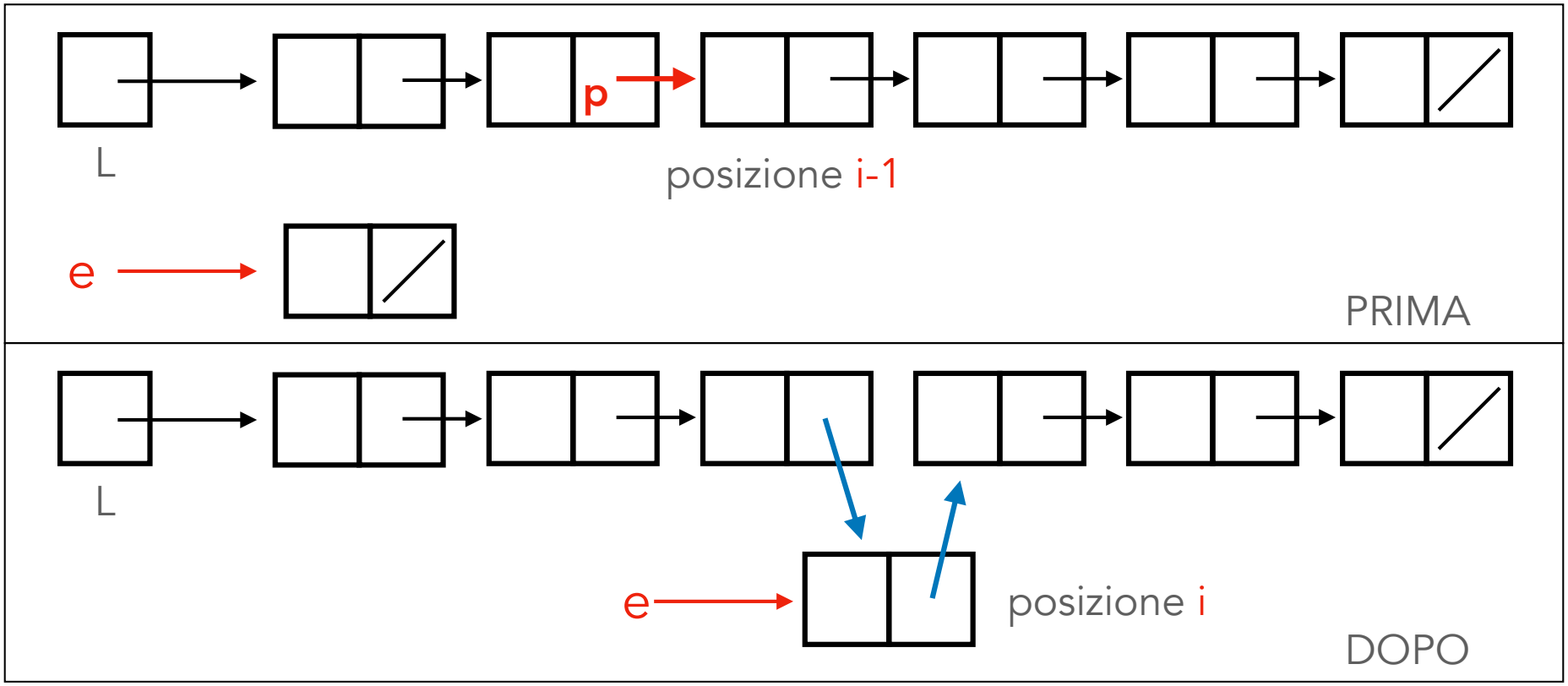
Costo computazionale

$\Theta(i)$

È una FUNZIONE!

LISTA: implementazione primitive

6. **Inserimento** di un nodo in una posizione data **i** (dato un puntatore **e** al nodo da inserire e contando le posizioni a partire da zero)




```
insert_pos(L,e,i)
 p := search(L,i-1)
 if NOT(p = NIL)
  then
    insert_next(p,e)
```

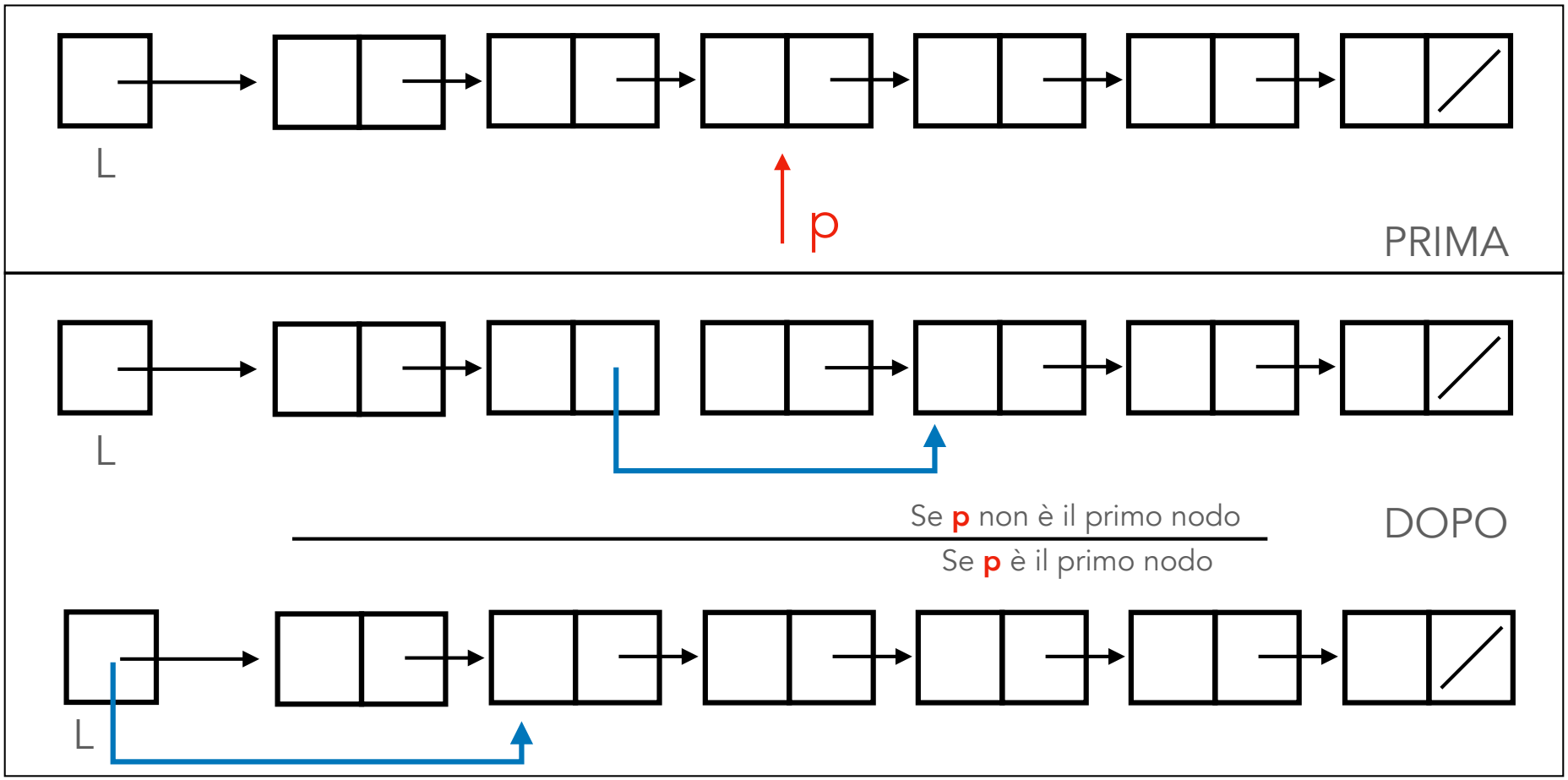
Costo computazionale

$\Theta(i)$

È una PROCEDURA!

LISTA: implementazione primitive

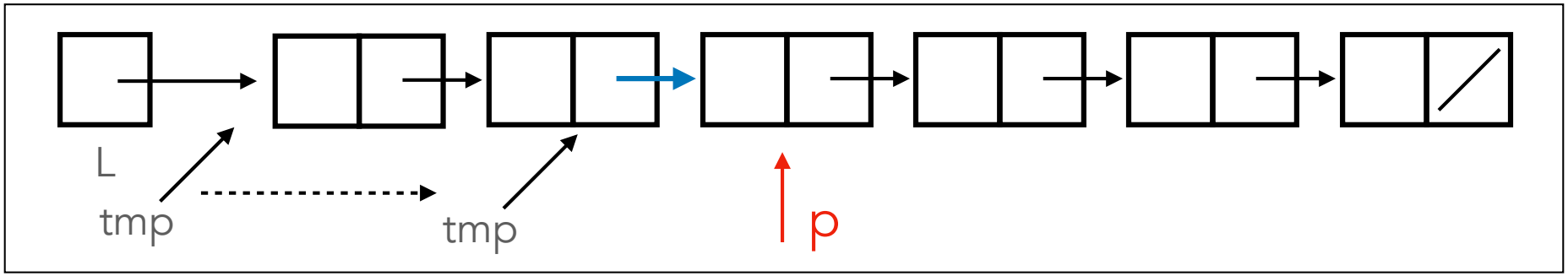
7. **Cancellazione** di un nodo dato (dato un puntatore **p** al nodo):
se il nodo non è nella lista, questa non deve essere modificata.



Nel caso in cui **p** non sia il primo della lista, è necessario cercare il puntatore nodo che lo precede

LISTA: implementazione primitive

7. **Cancellazione** di un nodo dato (dato un puntatore **p** al nodo):
se il nodo non è nella lista, questa non deve essere modificata.



tmp è un puntatore che scorre la lista e si posiziona
sul nodo che precede **p**

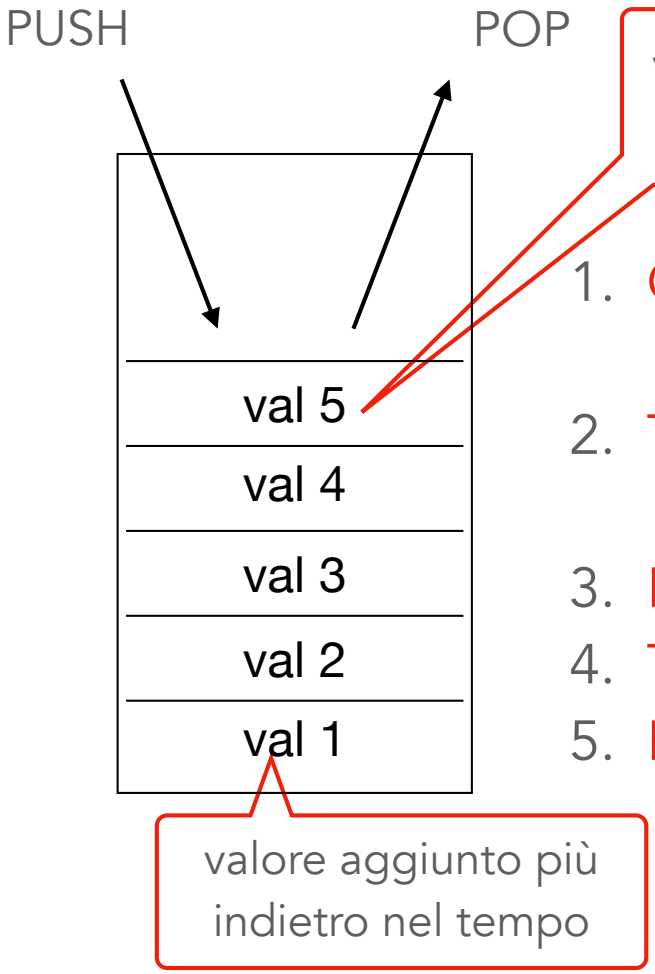
```
delete(L,p)
 if p = L
  then L := L.next
  else
   tmp := L
   while (tmp.next ≠ p AND tmp ≠ NIL)
    tmp := tmp.next
   if tmp ≠ NIL
    then tmp.next := p.next
```

Costo computazionale
$\Theta(n)$

PILA (STACK)

1) I valori memorizzati nella pila sono tutti dello stesso tipo
2) Nuovi valori vengono aggiunti in testa
3) Un'estrazione restituisce il valore in testa (l'ultimo inserito)

Implementa usa strategia: **LIFO** (Last In First Out)



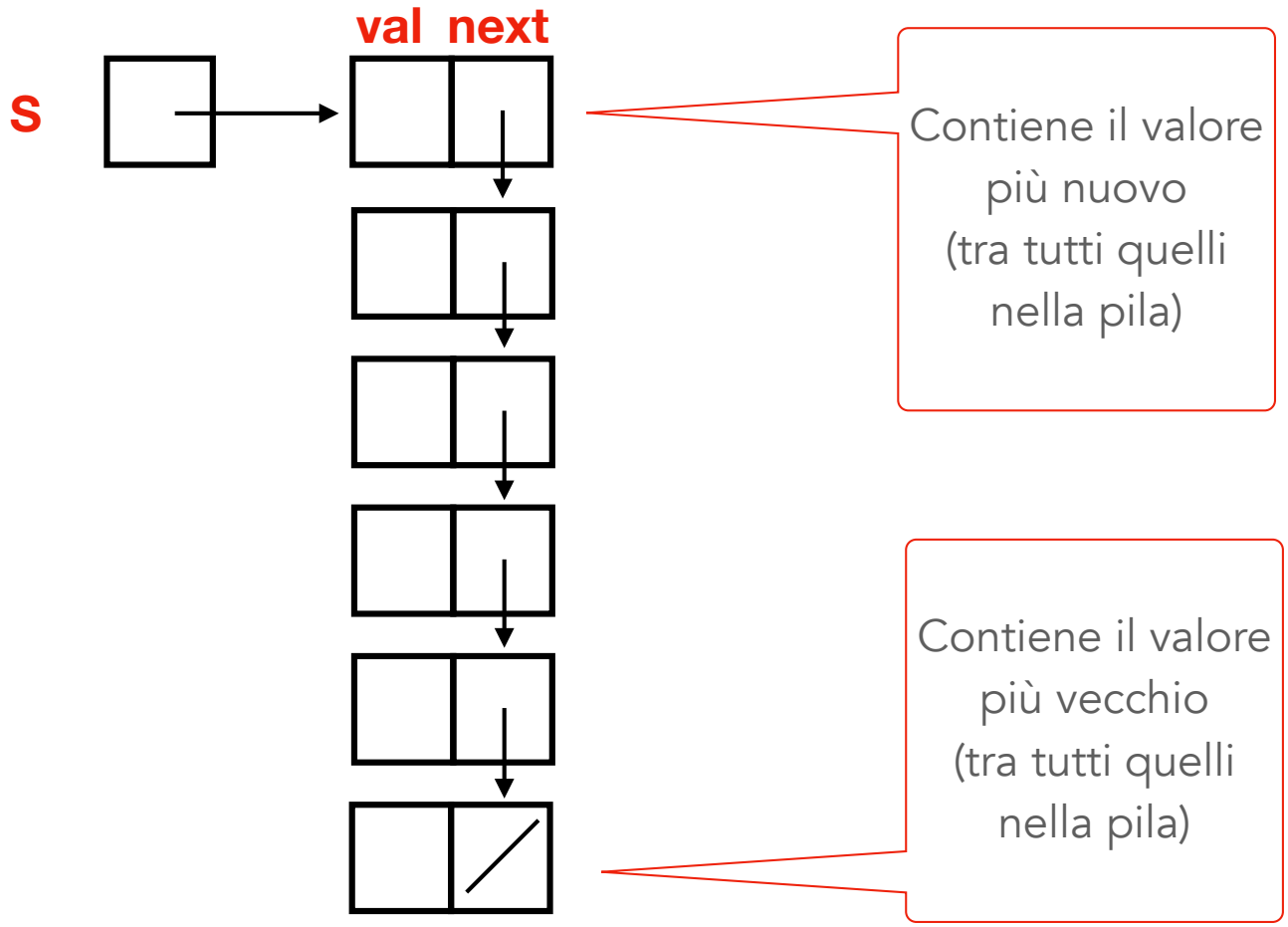
PRIMITIVE della **PILA**:

1. Crea una nuova pila: restituisce una nuova pila vuota: `new_stack()`
2. Test pila vuota: restituisce Vero se la pila non contiene valori, Falso altrimenti: `is_empty_stack(S)`
3. PUSH: inserimento di un valore in testa: `push(S,x)`
4. TOP: restituisce il valore in testa: `top(S)`
5. POP: restituisce il valore in testa alla pila e lo elimina dalla pila: `pop(S)`

PILA (STACK)

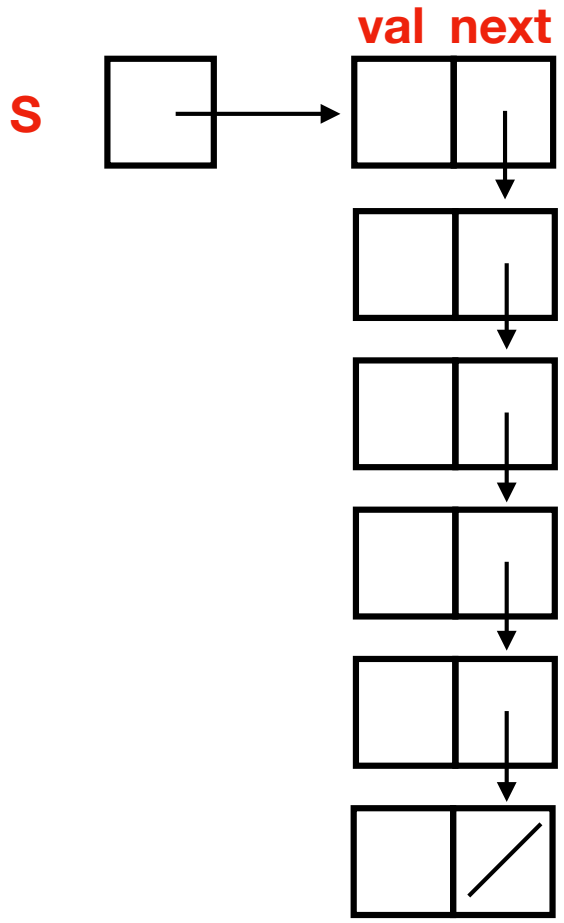
REALIZZAZIONE CON LISTE

PILA (STACK): puntatore al primo nodo di una lista che memorizzano i valori della pila



PILA (STACK)

1. Crea una nuova pila: restituisce una nuova pila vuota: `new_stack()`
2. Test pila vuota: restituisce Vero se la pila non contiene valori, Falso altrimenti: `is_empty_stack(S)`



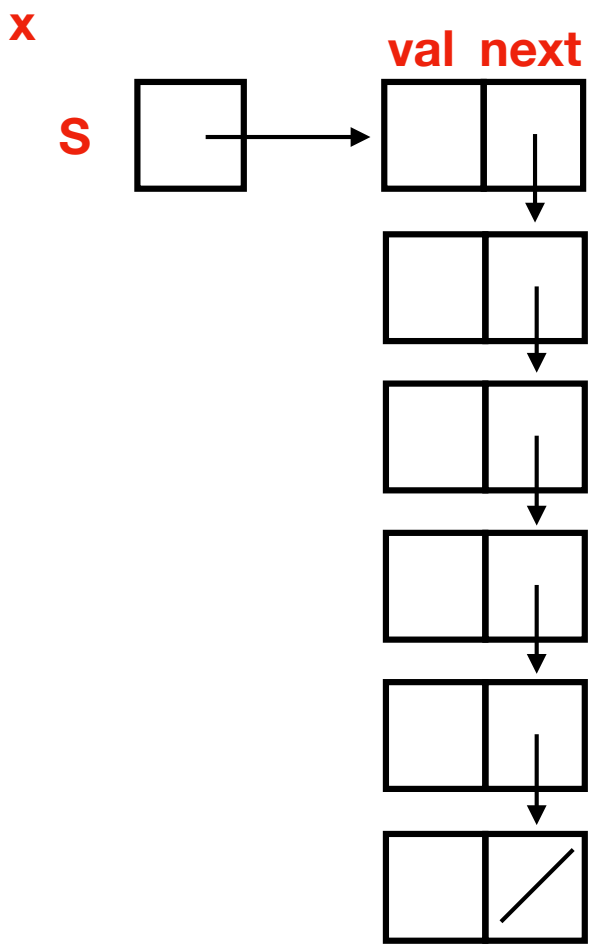

```
new_stack()
  return new_list()
```


```
is_empty_stack(S)
  return (S = NIL)
```

Costo computazionale $\Theta(1)$

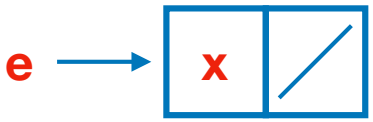
PILA (STACK)

3. PUSH: inserimento di un **valore** in testa: `push(S,x)`

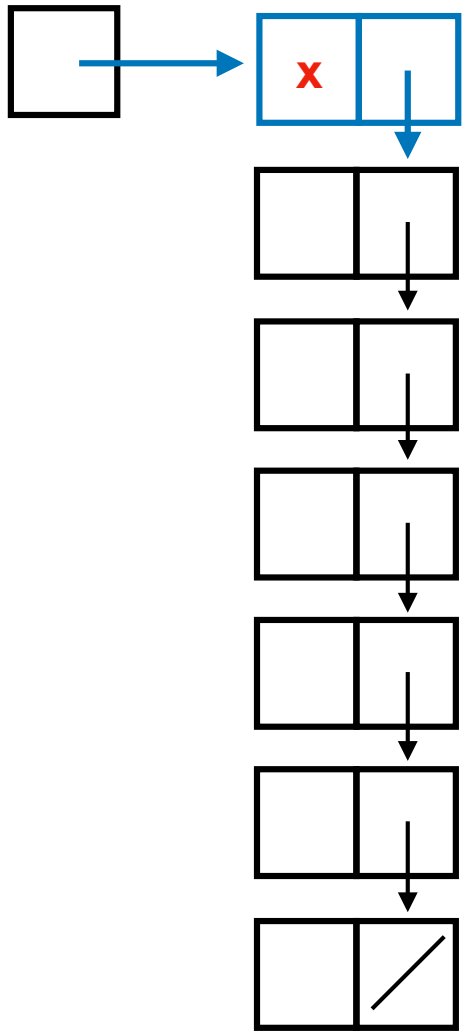


È passata per riferimento, alla fine la pila è stata modificata

```
push(S,x)
  if NOT S=NIL
    then
      e := new_list_node()
      e.val := x
      e.next := S
      S := e
    else
      return error
```

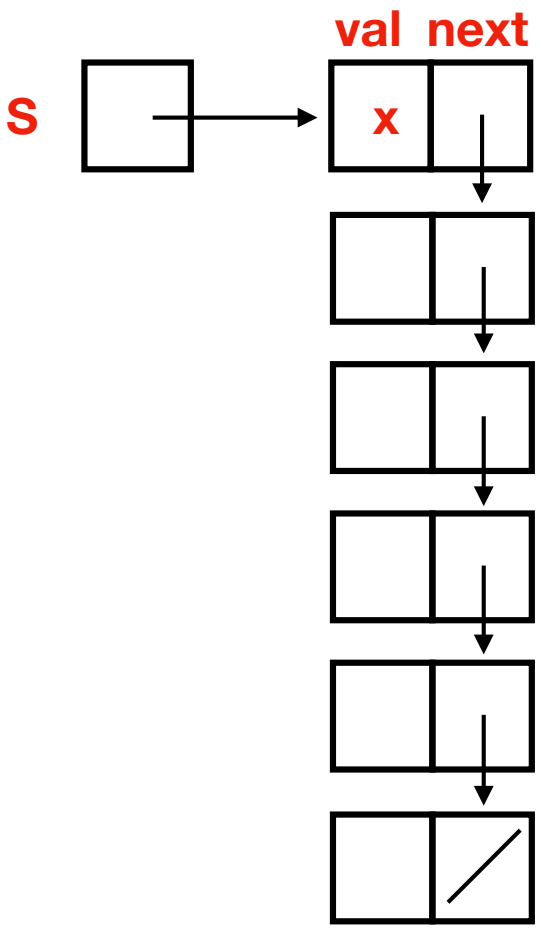


Costo computazionale $\Theta(1)$



PILA (STACK)

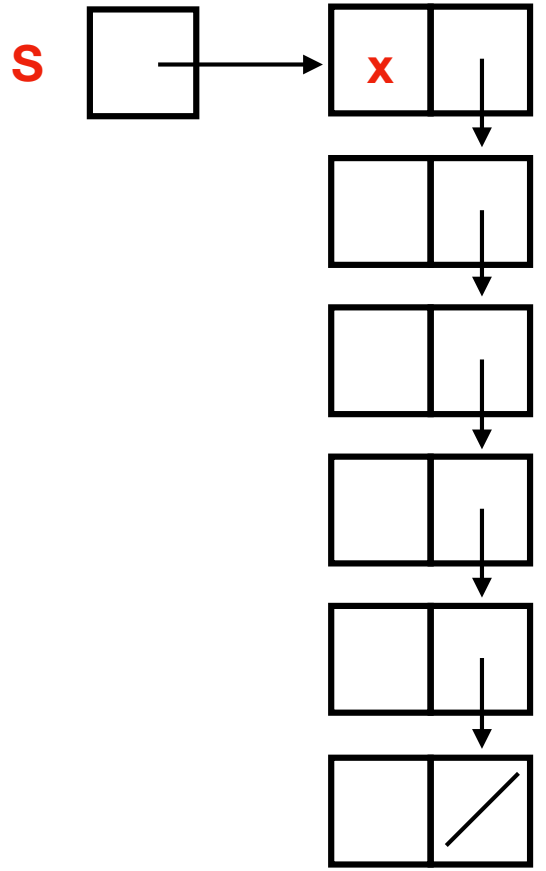
4. TOP: restituisce **il valore** in testa: `top(S)`




```
top(S)
 if NOT S = NIL
  then
   return S.val
  else
   return error
```

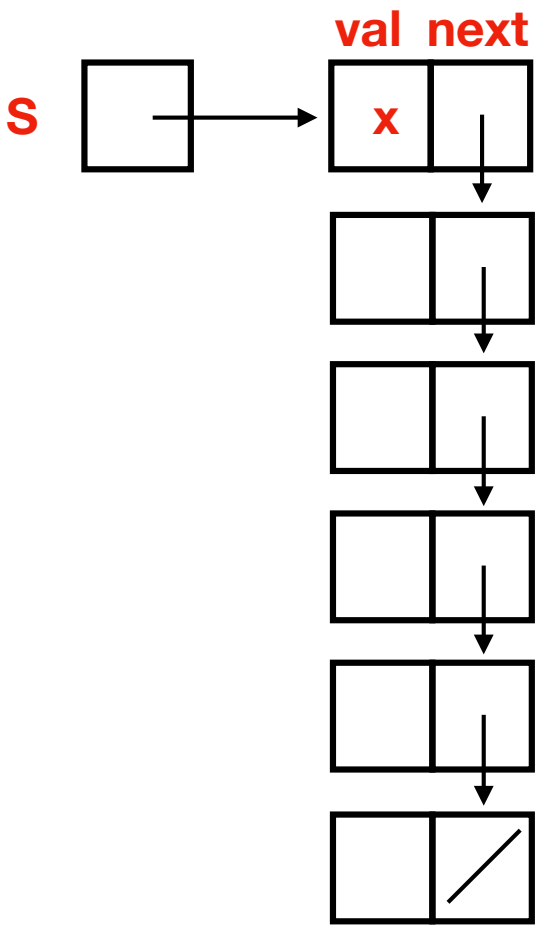
Costo computazionale $\Theta(1)$

Restituisce **x**



PILA (STACK)

5. POP: restituisce **il valore** in testa e lo elimina dalla pila: `pop(S)`

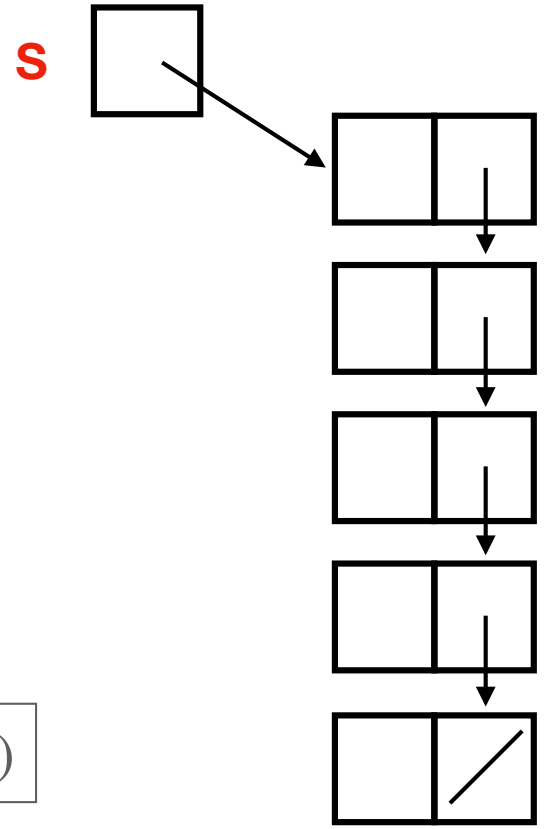


È passata per riferimento, alla fine la pila è stata modificata

```
pop(S)
if NOT S = NIL
 then
   x := top(S)
   S := S.next
   return x
 else
   return error
```

Costo computazionale $\Theta(1)$

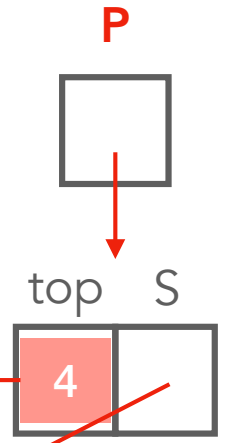
Restituisce **x**

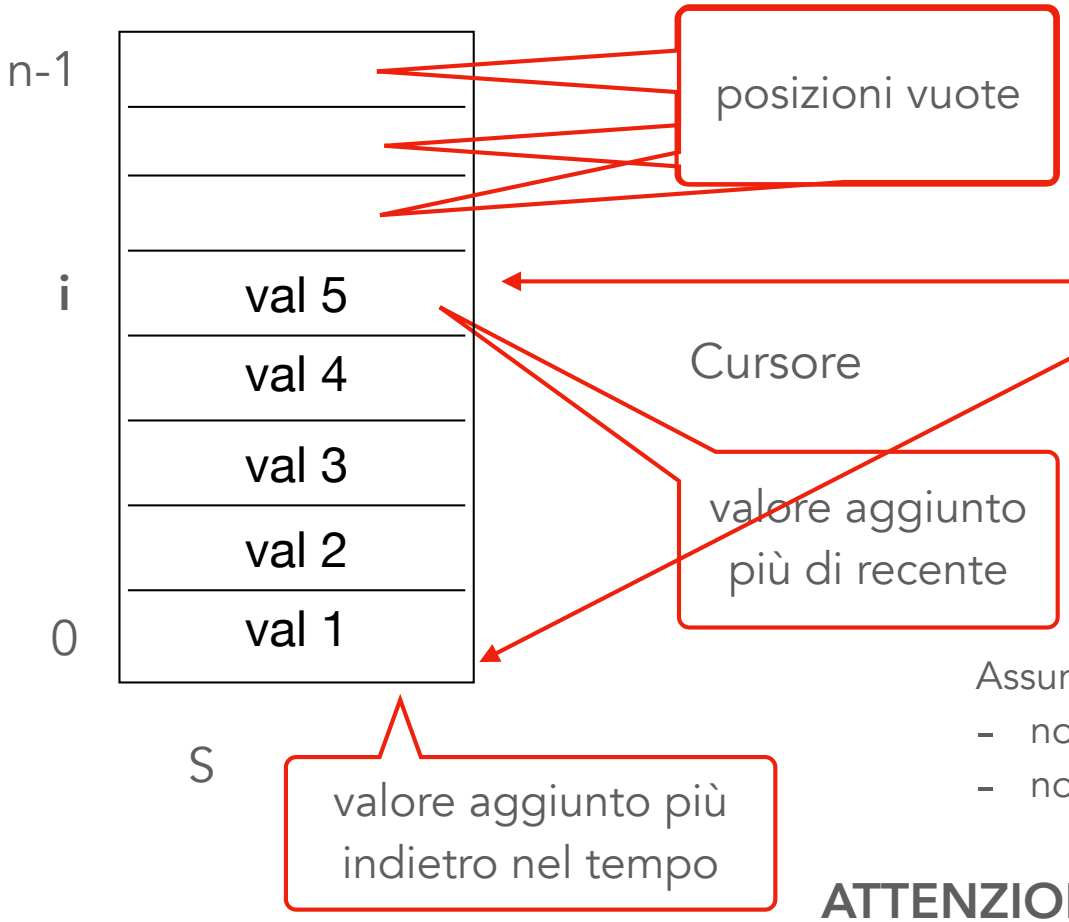


PILA (STACK) - esercizio

REALIZZAZIONE CON **ARRAY**

PILA (STACK): - array di n elementi S
- cursore alla testa (top) della pila





PENSATECI SOPRA PRIMA DI SBIRCIARE IL LIBRO O IL WEB!!

- P.S: array
- P.top: indice della cella che contiene il più recente valore della pila
- P.S[P.top]: valore più recente nella pila

Assunzioni:
- non è possibile inserire un elemento in una pila "piena"
- non è possibile eliminare un elemento da una pila vuota

ATTENZIONE: aggiungere primitiva `is_stack_full`

CODA (QUEUE)

1) I valori memorizzati nella coda sono tutti dello stesso tipo
2) Nuovi valori vengono aggiunti in fondo alla coda
3) Un'estrazione restituisce il primo valore in coda

Implementa usa strategia: **FIFO** (First In First Out)

PRIMITIVE della **CODA**:

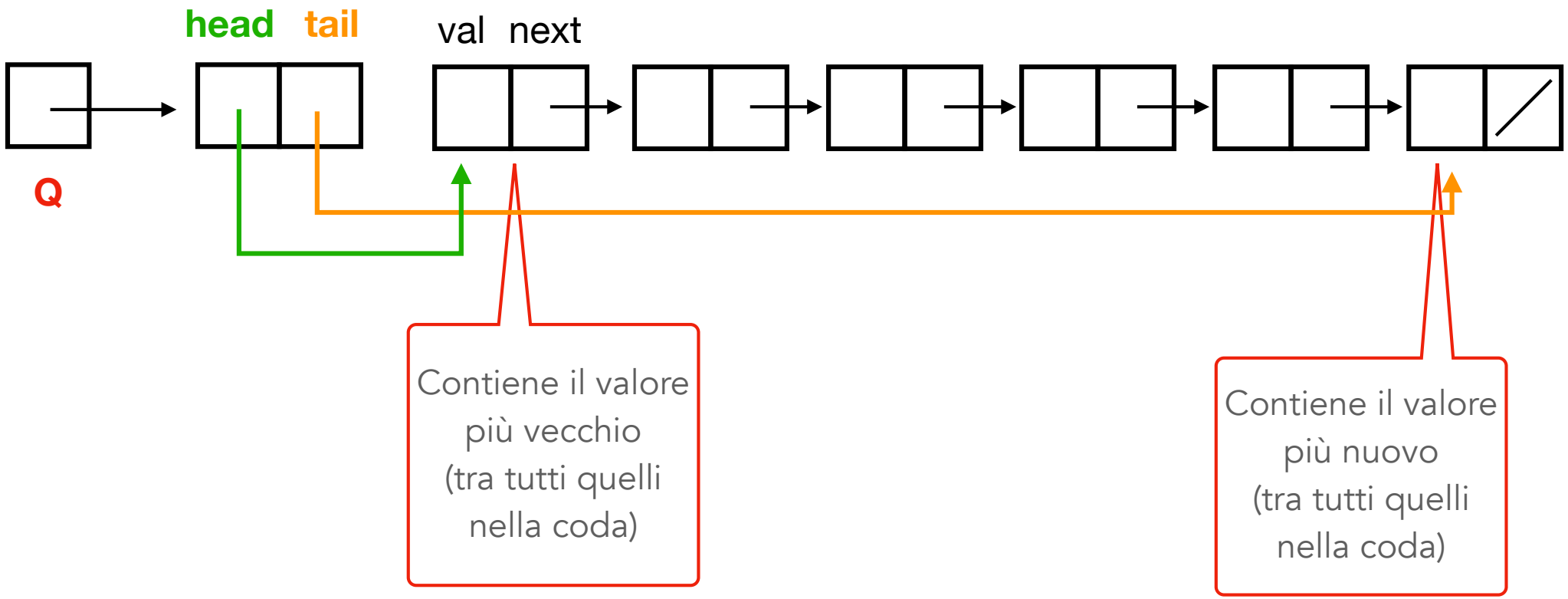
1. Crea una nuova coda: restituisce una nuova coda vuota: `new_queue()`
2. Test coda vuota: restituisce Vero se la coda non contiene valori, Falso altrimenti: `is_empty_queue(Q)`
3. ENQUEUE: inserimento di un valore in fondo alla coda: `enqueue(Q,x)`
4. FIRST: restituisce il valore in testa alla coda: `first(Q)`
5. DEQUEUE: restituisce il valore in testa alla coda e lo elimina dalla coda: `dequeue(Q)`

CODA (QUEUE)

REALIZZAZIONE CON LISTE

CODA (QUEUE): **puntatore** ad un nodo che contiene due puntatori:

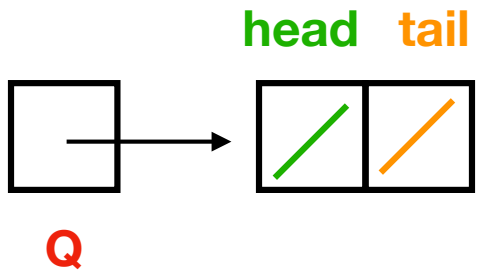
- **head**: al primo nodo di una lista che memorizza i valori della coda
- **tail**: all'ultimo nodo della stessa lista



CODA (QUEUE)

1. Crea una nuova coda: restituisce una nuova coda vuota: `new_queue()`
2. Test coda vuota: restituisce Vero se la coda non contiene valori, Falso altrimenti: `is_empty_queue(Q)`

```
new_queue()
  Q := new_queue_node()
  Q.head := NIL
  Q.tail := NIL
  return Q
```

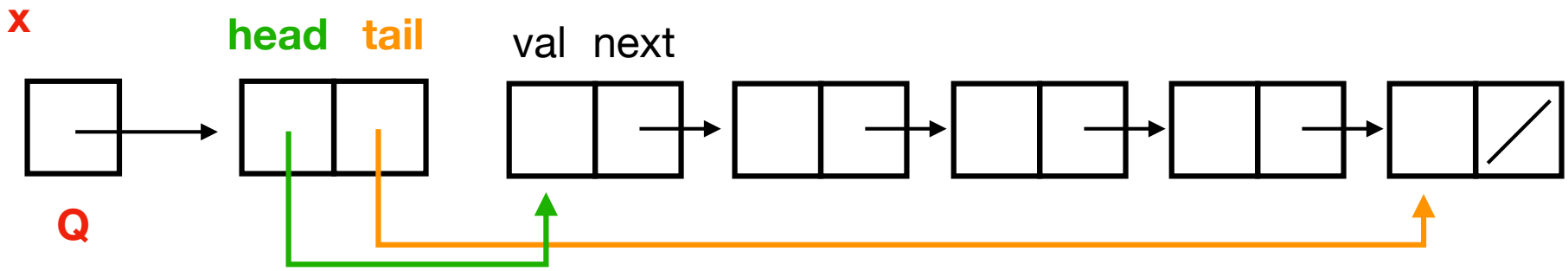



```
is_empty_queue(Q)
  return (Q.head = NIL)
```

Costo computazionale $\Theta(1)$

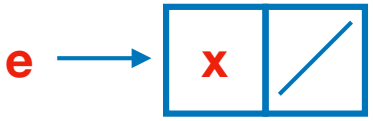
CODA (QUEUE)

3. ENQUEUE: inserimento di un **valore** in fondo alla coda: `enqueue(Q,x)`



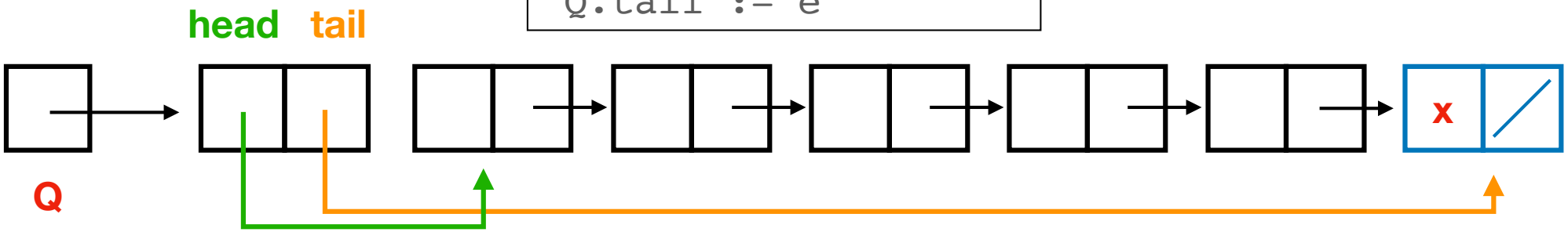
È passata per riferimento, alla fine la coda è stata modificata

```
enqueue(Q,x)
 e := new_list_node()
 e.val := x
 e.next := NIL
 if Q.head = NIL
  then
   Q.head := e
  else
   Q.tail.next := e
 Q.tail := e
```



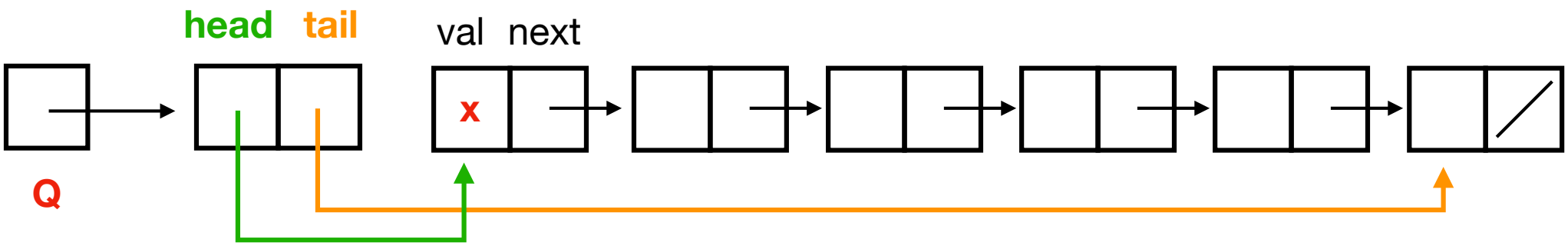
Costo computazionale

$\Theta(1)$



CODA (QUEUE)

4. FIRST: restituisce **il valore** in testa alla coda: `first(Q)`



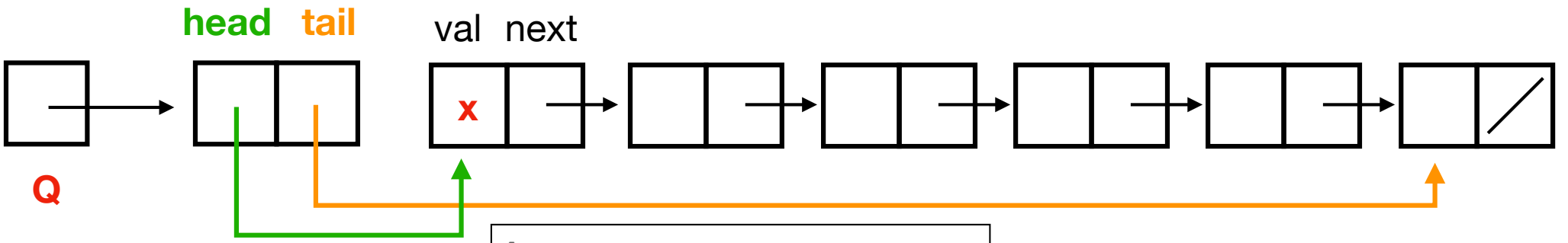

```
first(Q)
 if Q.head ≠ NIL
  then return Q.head.val
  else error
```

Restituisce **x**

Costo computazionale $\Theta(1)$

CODA (QUEUE)

5. DEQUEUE: restituisce **il valore** in testa alla coda e lo elimina dalla coda: `dequeue(Q)`



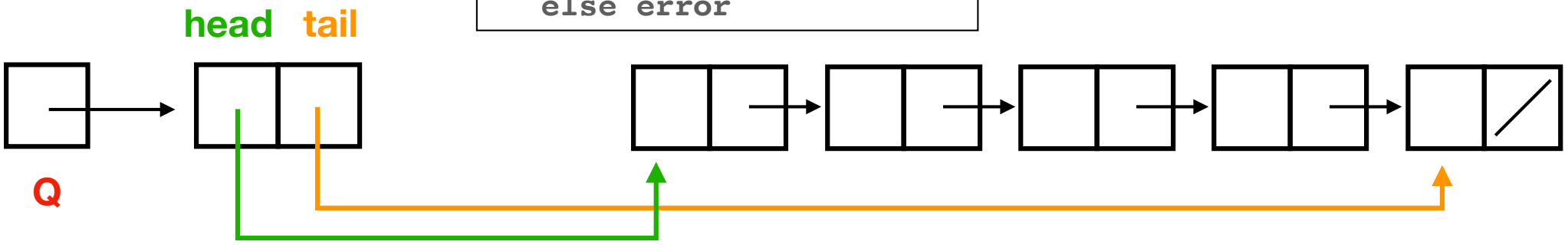
È passata per riferimento, alla fine la coda è stata modificata

```
dequeue(Q)
  if NOT is_empty_queue(Q)
    then
      x := first(Q)
      Q.head := Q.head.next
      if Q.head = NIL
        then
          Q.tail := NIL
      return x
    else error
```

Costo computazionale

$\Theta(1)$

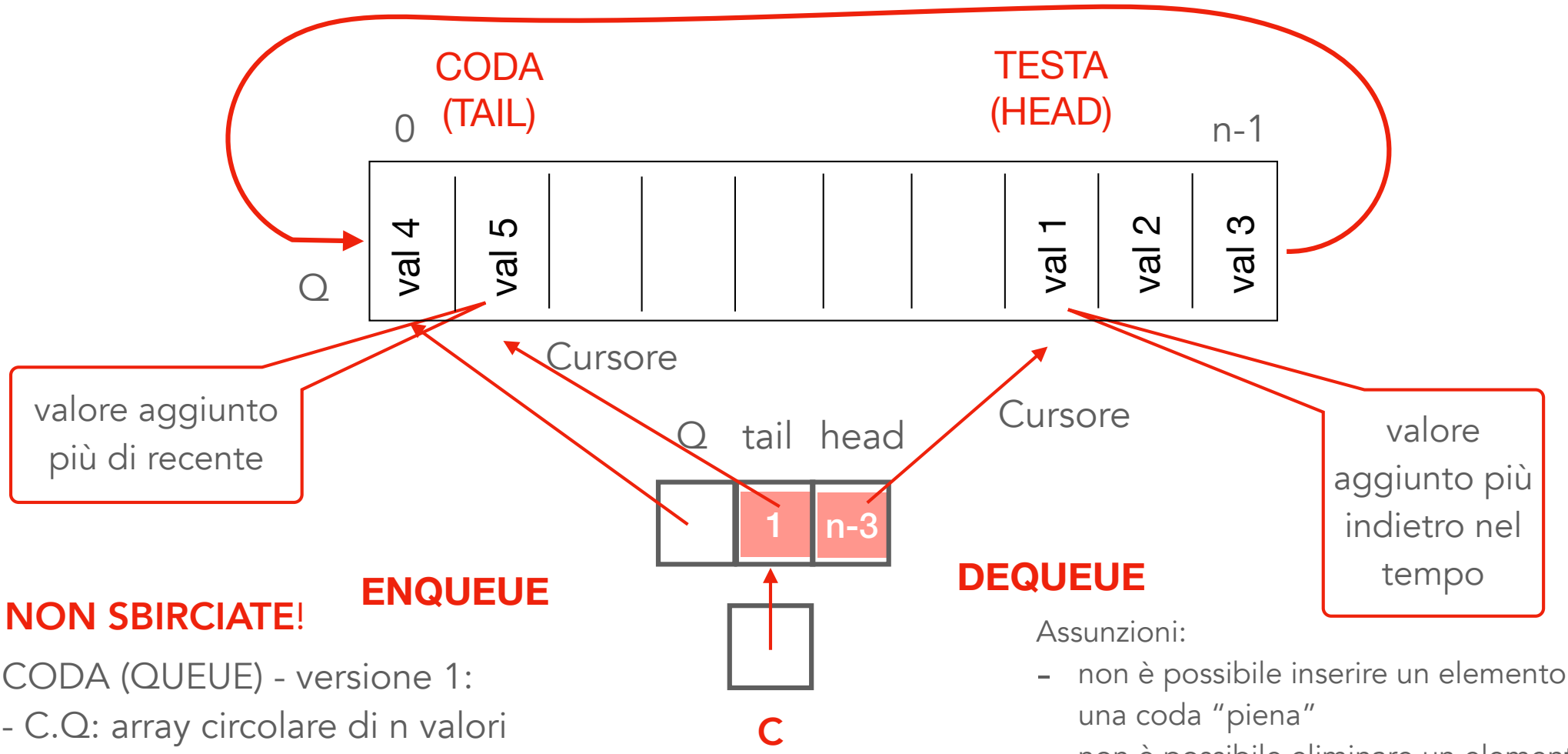
Restituisce **x**



CODA (QUEUE) - esercizio

REALIZZAZIONE CON **ARRAY CIRCOLARE**

(la posizione 0 è considerata la successiva della posizione n-1)



NON SBIRCIATE!

ENQUEUE

CODA (QUEUE) - versione 1:

- C.Q: array circolare di n valori
- C.head: indice della cella che contiene la head della coda
- C.tail: indice della cella che contiene la tail della coda
- usare "+/- 1 mod n" per aggiornare i cursori

DEQUEUE

Assunzioni:

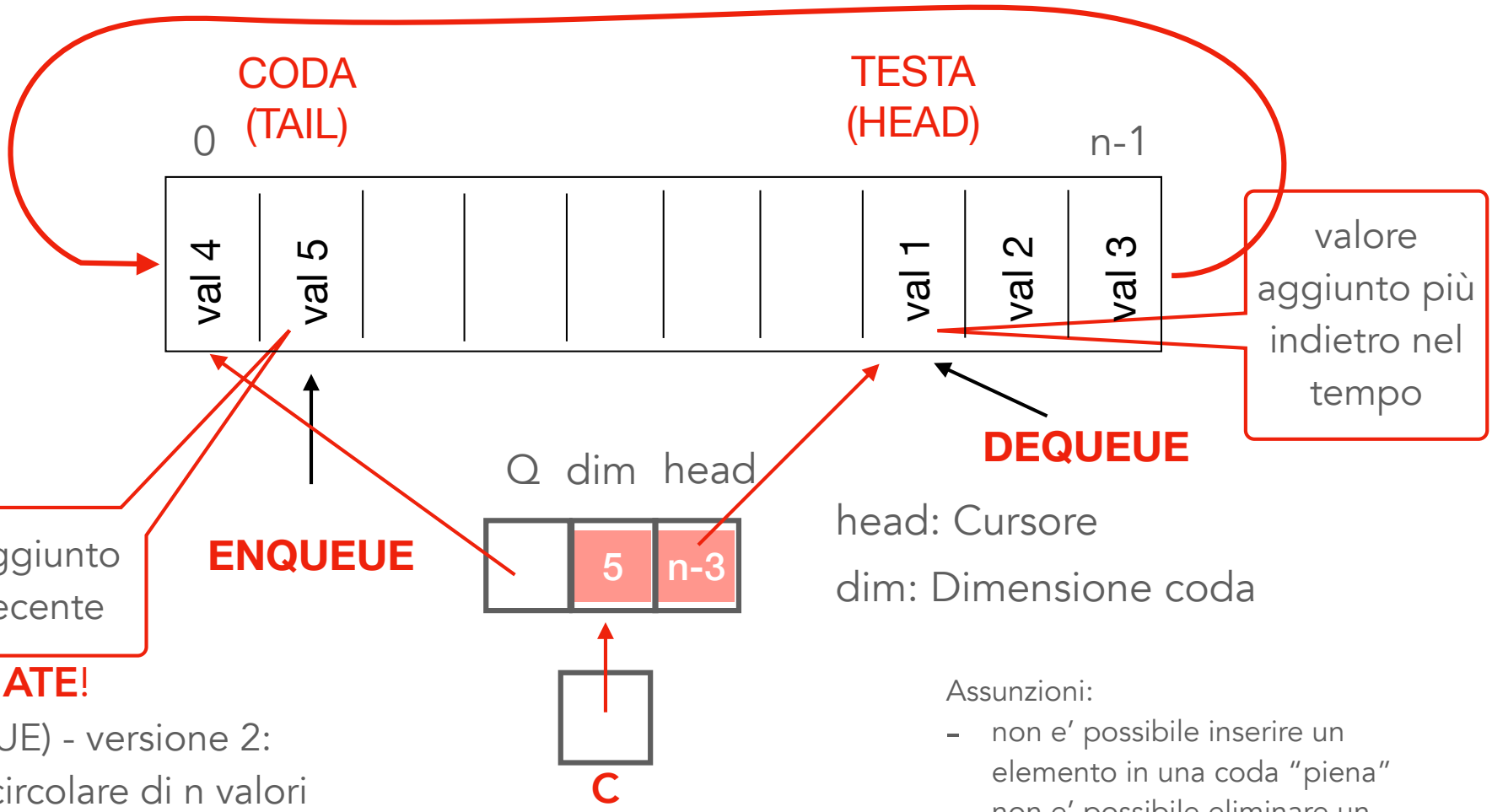
- non è possibile inserire un elemento in una coda "piena"
- non è possibile eliminare un elemento da una coda vuota

ATTENZIONE: aggiungere primitiva `is_queue_full`

CODA (QUEUE) - esercizio

REALIZZAZIONE CON **ARRAY CIRCOLARE**

(la posizione 0 è considerata la successiva della posizione n-1)



head: Cursore

dim: Dimensione coda

CODA (QUEUE) - versione 2:

- C.Q: array circolare di n valori
- C.head: indice della cella che contiene la head della coda
- C.dim: in numero di elementi in coda
- usare "mod n" per aggiornare il cursore e determinare la tail della coda

Assunzioni:

- non e' possibile inserire un elemento in una coda "piena"
- non e' possibile eliminare un elemento da una coda vuota

ATTENZIONE: aggiungere primitiva `is_queue_full`