

Liste



Algoritmi e strutture dati
Ugo de'Liguoro, Andras Horvath

Sommario

- Obiettivi
 - Studiare le liste e gli algoritmi sulle liste dal punto di vista della loro complessità
- Argomenti
 - Liste semplici
 - Inserimento, ricerca e cancellazione
 - Copia ed inversione
 - Ordinamento

Che cosa è una lista

Una lista è una sequenza finita di valori:


$$L = [a_1, \dots, a_k]$$

dove $a_1, \dots, a_k \in A$, per qualche insieme A .

Se $k = 0$, allora $L = []$, o anche $L = \text{nil}$, è la lista vuota.

Le parti di una lista

Sia $k \neq 0$:

$$L = [a_1, a_2, \dots, a_k]$$


$$\text{Head}(L) = a_1$$

$$\text{Tail}(L) = [a_2, \dots, a_k]$$

$$\text{Cons}(a, [a_1, \dots, a_k]) = [a, a_1, \dots, a_k]$$

$$\text{Cons}(\text{Head}(L), \text{Tail}(L)) = L \text{ per ogni } L \neq \text{nil}$$

Una definizione induttiva

Fissato un insieme di elementi A , l'insieme delle liste su A , $\text{List}(A)$ o semplicemente List , è il più piccolo tale che:

- $[]$ (ovvero nil) $\in \text{List}$
- se $a \in A$, $L \in \text{List}$ allora $\text{Cons}(a, L) \in \text{List}$

Confrontiamo questa definizione con quella dei numeri naturali, Nat , che è il più piccolo insieme tale che:

- $0 \in \text{Nat}$
- se $n \in \text{Nat}$ allora $\text{Succ}(n) \in \text{Nat}$ (dove $\text{Succ}(n) = n + 1$)

Il tipo delle liste

In C++ definiremo un tipo List di valori di tipo T (negli esempi $T = \text{int}$); possiamo allora definire il tipo delle costanti e degli operatori sulle liste:

- $\text{nil} : \text{List}$ (nil ha tipo List)
- $\text{Cons} : T \times \text{List} \rightarrow \text{List}$
- $\text{Head} : \text{List} \rightarrow T$ ($\text{Head}(\text{nil}) = \perp$)
- $\text{Tail} : \text{List} \rightarrow \text{List}$ ($\text{Tail}(\text{nil}) = \perp$)

\perp = indefinito



Algebra delle liste

A questi operatori ne possiamo aggiungere altri definendoli ricorsivamente secondo lo schema

$$f(\text{nil}, x_1, \dots, x_k) = g(x_1, \dots, x_k)$$

$$f(\text{Cons}(z, L), x_1, \dots, x_k) = \\ h(f(L, x_1, \dots, x_k), z, L, x_1, \dots, x_k)$$

supposte note g ed h .



Ricorsione primitiva sulle
liste

Algebra delle liste

Esempi:

$\text{IsEmpty} : \text{List} \rightarrow \text{Bool}$ (predicato “L è vuota”)

$\text{IsEmpty}(\text{nil}) = \text{true}$

$\text{IsEmpty}(\text{Cons}(z, L)) = \text{false}$

$\text{Length} : \text{List} \rightarrow \text{Int}$ (funzione lunghezza)

$\text{Length}(\text{nil}) = 0$

$\text{Length}(\text{Cons}(z, L)) = 1 + \text{Length}(L)$

Algebra delle liste

Dati gli operatori Head e Tail ed il predicato IsEmpty, e supposto di avere un operatore a tre posti

if-then-else : $\text{Bool} \times T \times T \rightarrow T$ per ogni T

possiamo trasformare lo schema di ricorsione in una definizione più vicina al codice C++:

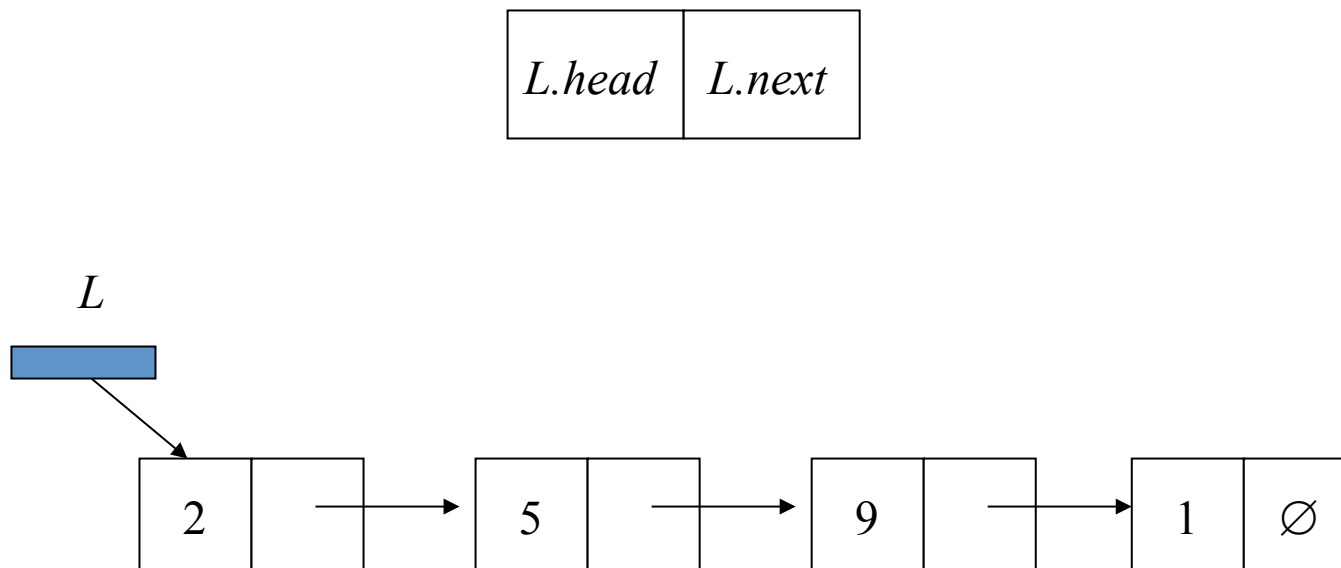
$f(L, x_1, \dots, x_k) =$
if IsEmpty (L) then $g(x_1, \dots, x_k)$
else $h(f(\text{Tail}(L), x_1, \dots, x_k), \text{Head}(L), \text{Tail}(L), x_1, \dots, x_k)$

Esempio:

$\text{Length}(L) = \text{if IsEmpty } (L) \text{ then } 0 \text{ else } 1 + \text{Length}(\text{Tail}(L))$

Le liste semplici

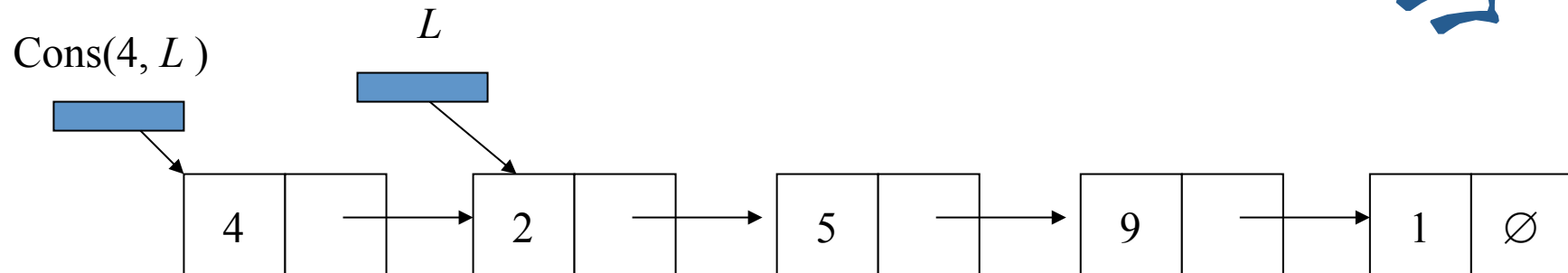
Come struttura dati una lista è una sequenza di record, ciascuno dei quali contiene un campo che punta al successivo:



La funzione Cons

$\text{CONS}(x, L)$
 $N.\text{head} \leftarrow x$
 $N.\text{next} \leftarrow L$
return N

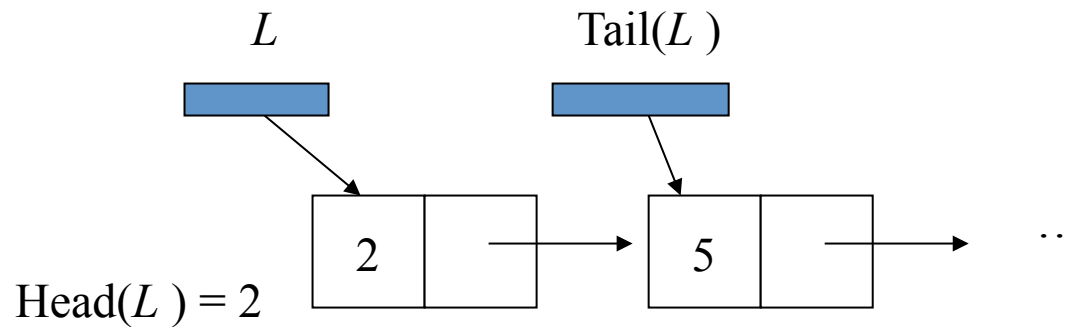
Nelle implementazioni
Cons è un allocatore



Le funzioni Head e Tail

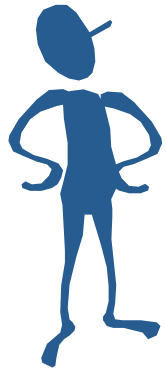
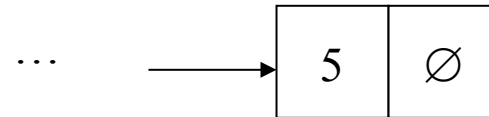
```
HEAD(L)  
return L.head
```

```
TAIL(L)  
return L.next
```



La lista vuota

IsEmpty(L)
return $L = nil$



Al fondo di ogni lista c'è
la lista vuota, ossia un
puntatore a *nil*

THEEMPTYLIST()
return *nil*

Lunghezza di una lista

Versione ricorsiva:

```
LENGTH-REC( $L$ )  
if  $IsEmpty(L)$  then  
    return 0  
else  
    return  $1 + LENGTH-REC(Tail(L))$   
end if
```



Lunghezza di una lista

LENGTH-TAIL-REC(L, n) \triangleright post: ritorna $length(L) + n$

```
if ISEMPTY( $L$ ) then
    return  $n$ 
else
    return LENGTH-TAIL-REC(TAIL( $L$ ),  $n + 1$ )
end if
```

LENGTH-ITER(L)

```
 $n \leftarrow 0$ 
while not ISEMPTY( $L$ ) do
     $L \leftarrow$  TAIL( $L$ )
     $n \leftarrow n + 1$ 
end while
return  $n$ 
```

Le versioni ricorsiva di
coda ed iterativa sono
fortemente equivalenti



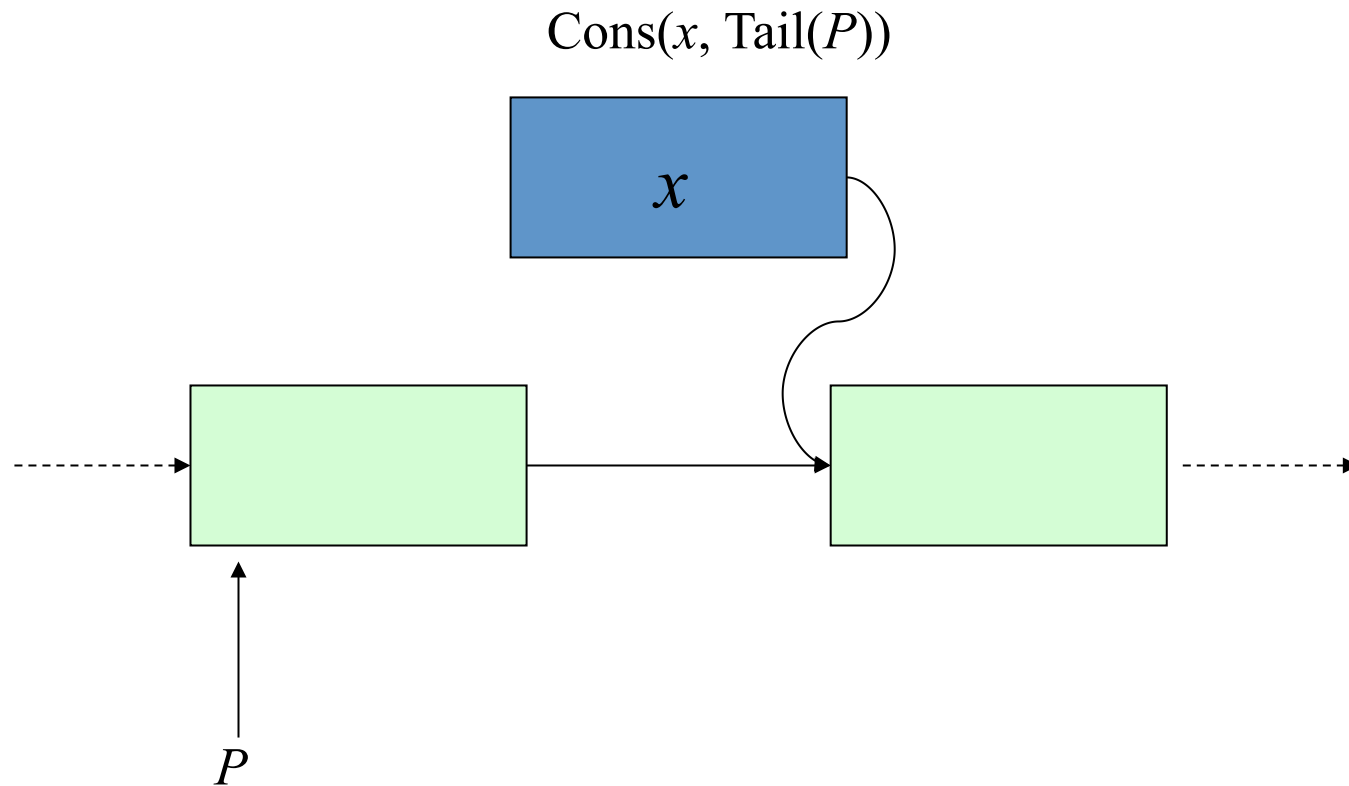
Ricerca in una lista

```
SEARCH( $x, L$ )  
  if ISEMPY( $L$ ) then  
    return false  
  else  
    return HEAD( $L$ ) =  $x$  or SEARCH( $x$ , TAIL( $L$ ))  
  end if
```



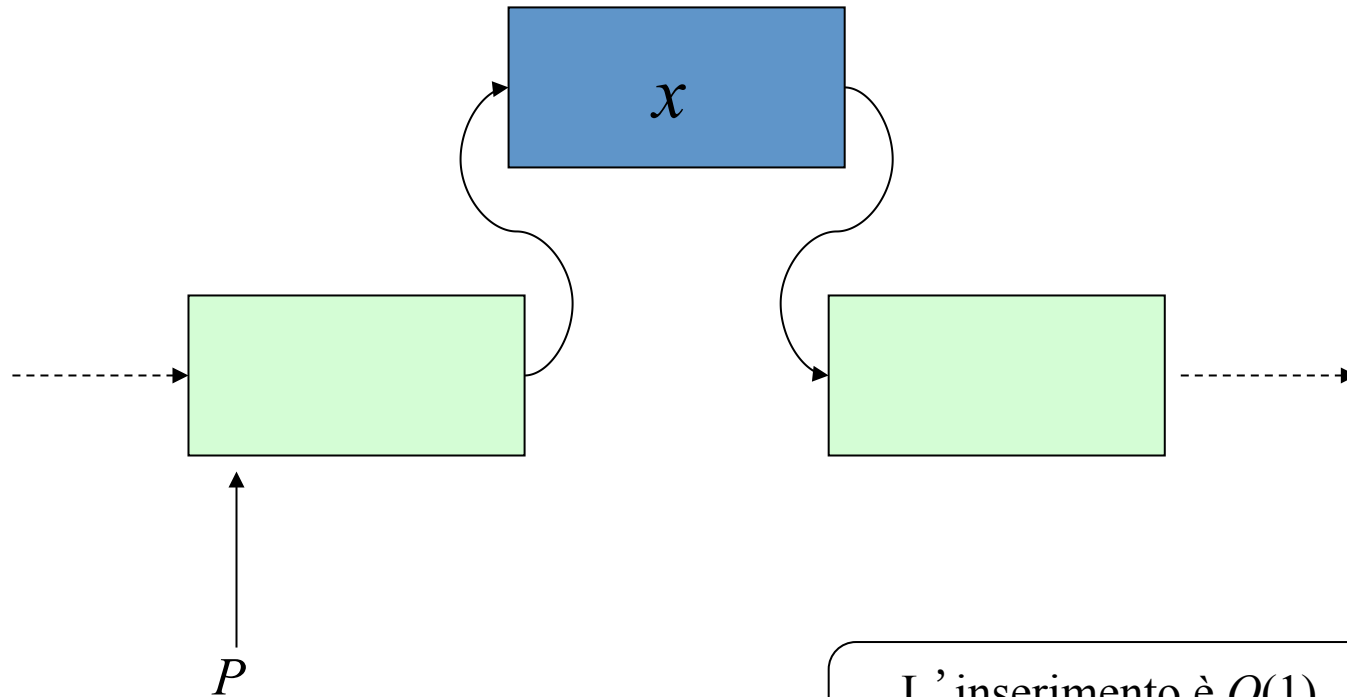
Se $L \neq \emptyset$ allora $x \in L$ solo se
 $x = \text{Head}(L)$ oppure $x \in \text{Tail}(L)$

Inserimento in una lista



Inserimento in una lista

$$P.next = \text{Cons}(x, \text{Tail}(P))$$



L' inserimento è $O(1)$
posto che si conosca P



Un metodo ricorsivo

INSERT(x, i, L)

▷ post: x inserito davanti all'el. di posto i in L

if $i = 1$ **then**

return CONS(x, L)

else

$L.next \leftarrow$ INSERT($x, i - 1, \text{TAIL}(L)$)

return L

end if

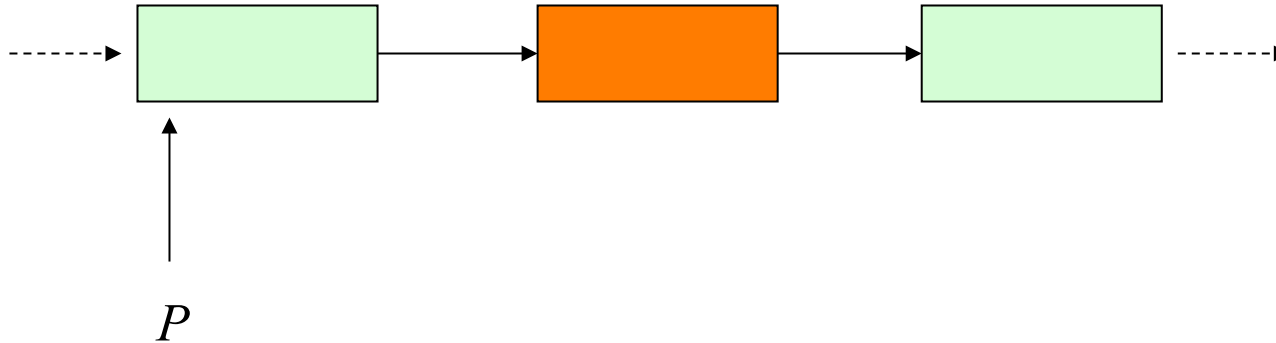


Come sarà la versione
iterativa?

Cancellazione da una lista

$temp \leftarrow Tail(P)$

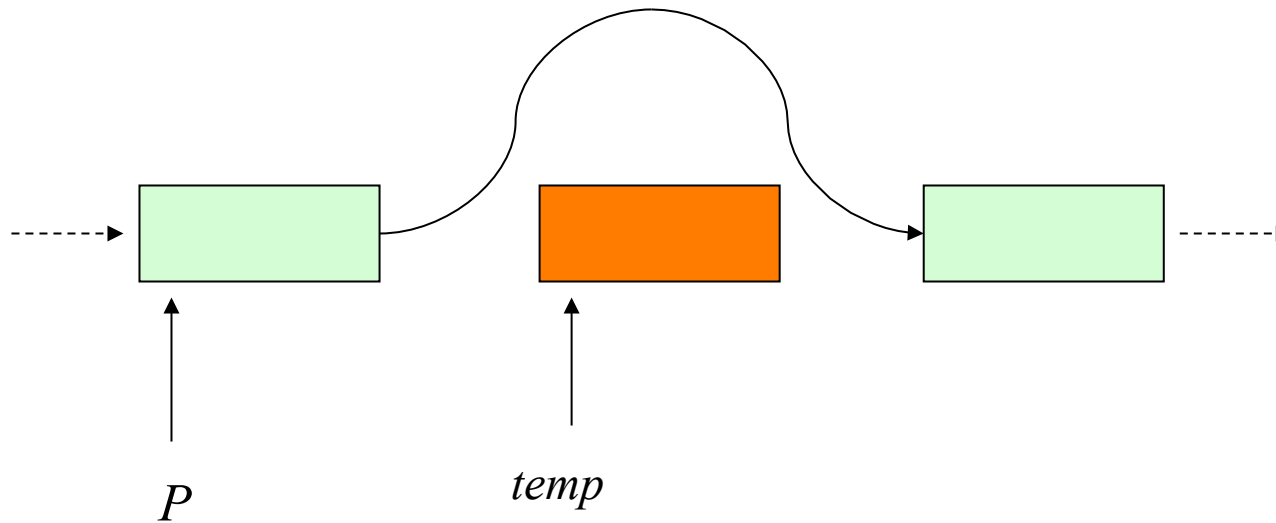
$P.next \leftarrow Tail(Tail(P))$



Cancellazione da una lista

$temp \leftarrow Tail(P)$

$P.next \leftarrow Tail(Tail(P))$



Cancellazione da una lista

```
DELETEALL( $x, L$ )  
  ▷ post: ogni occ. di  $x$  è rimossa da  $L$   
  if ISEMPTY( $L$ ) then  
    return nil  
  else  
    if  $x = \text{HEAD}(L)$  then  
      return DELETEALL( $x, \text{TAIL}(L)$ )  
    else  
       $L.\text{next} \leftarrow \text{DELETEALL}(x, \text{TAIL}(L))$   
      return  $L$   
    end if  
  end if
```



Altri algoritmi sulle liste

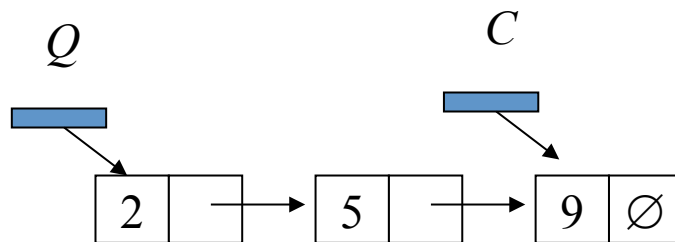
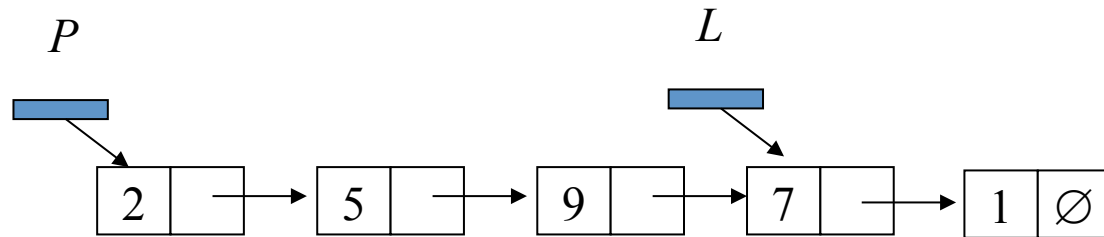
Per apprendere come manipolare (ricorsivamente) le liste consideriamo:

- Copia di una lista
- Inversione dell'ordine degli elementi in una lista
- Ricerca, inserimento e cancellazione in una lista ordinata
- Ordinamento di una lista per fusione (Merge-Sort), dopo averla divisa in due metà (circa)

Copia di una lista

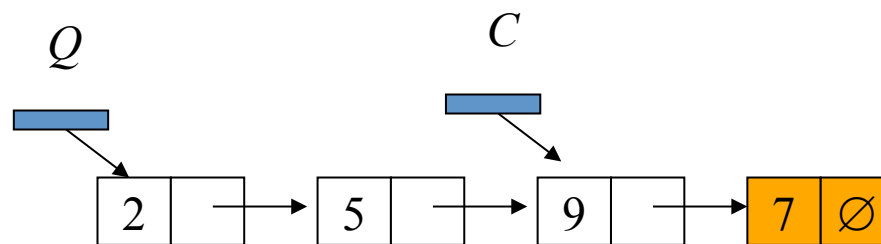
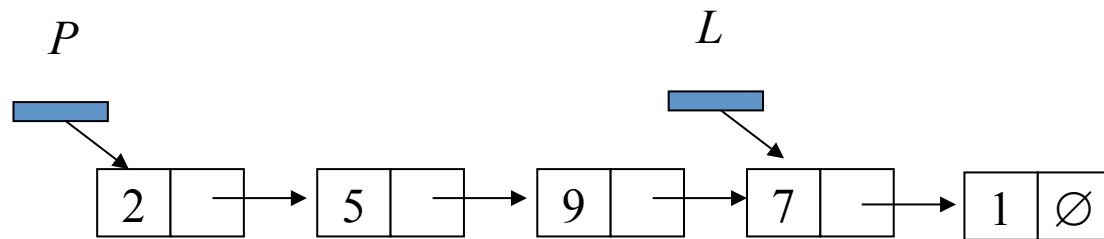
- Una lista è una struttura dati persistente, che può subire modifiche durante l'esecuzione di una procedura
- È allora utile poter duplicare una lista quando si vuole che l'effetto delle modifiche sia solo temporaneo

Copia di una lista



$C.next = \text{Cons}(\text{Head}(L), \text{nil})$

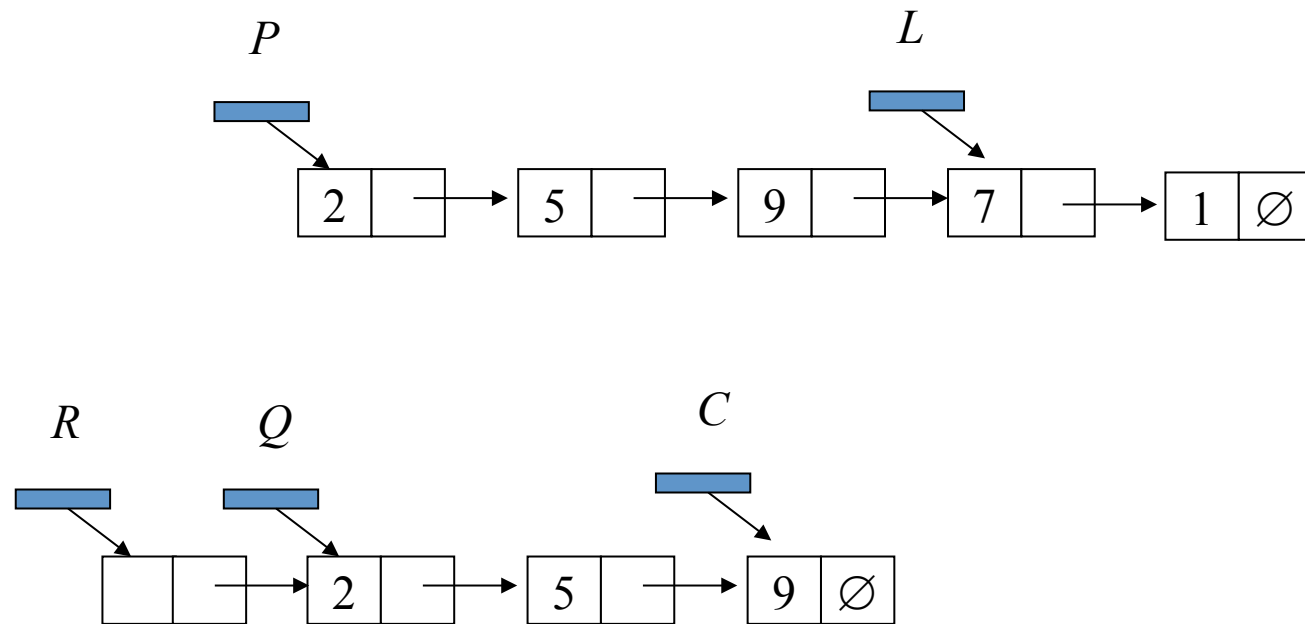
Copia di una lista



Ma come si
comincia?

$C.next = \text{Cons}(\text{Head}(L), \text{nil})$

Copia di una lista

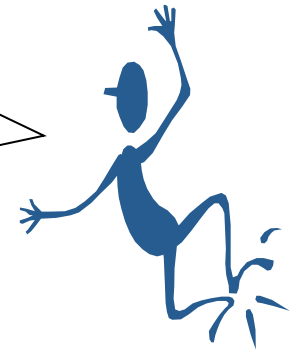


Record fittizio sulla cui coda
costruiamo la copia di P

Copia di una lista

```
CLONE-ITER(L)  
R ← CONS(–, nil)  
C ← R  
while L ≠ nil do  
    C.next ← CONS(HEAD(L), nil)  
    C ← TAIL(C)  
    L ← TAIL(L)  
end while  
return TAIL(R)
```

La versione
ricorsiva è più
chiara e concisa



```
CLONE-REC(L)  
if ISEMPY(L) then  
    return nil  
else  
    return CONS(HEAD(L), CLONE-REC(TAIL(L)))  
end if
```

Inversione di una lista

L'inversa della lista:

[1, 2, 3, 4, 5]

è la lista:

[5, 4, 3, 2, 1]

Esiste una soluzione iterativa?



Inversione di una lista: iterazione

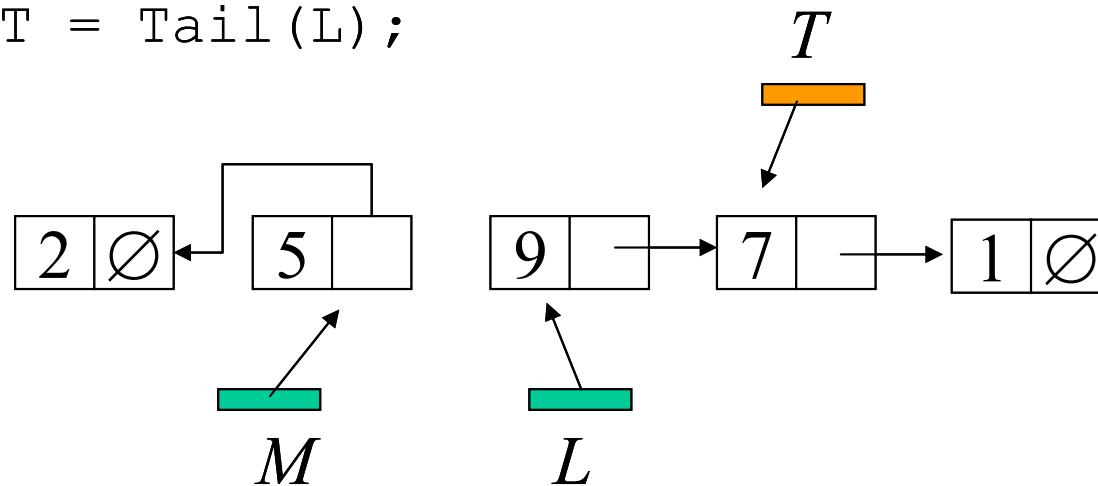
Problema: una lista semplice ha un verso solo (quindi l'algoritmo di inversione per un vettore non lo posso adattare)



Posso “girare” i puntatori mentre scandisco la lista con due puntatori

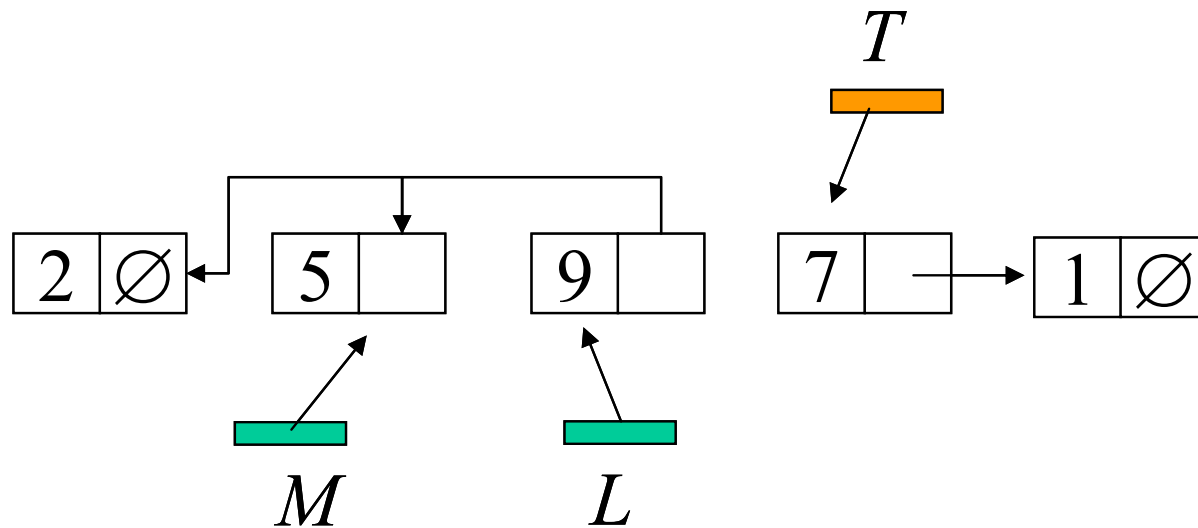
Inversione di una lista: iterazione

`T = Tail(L);`



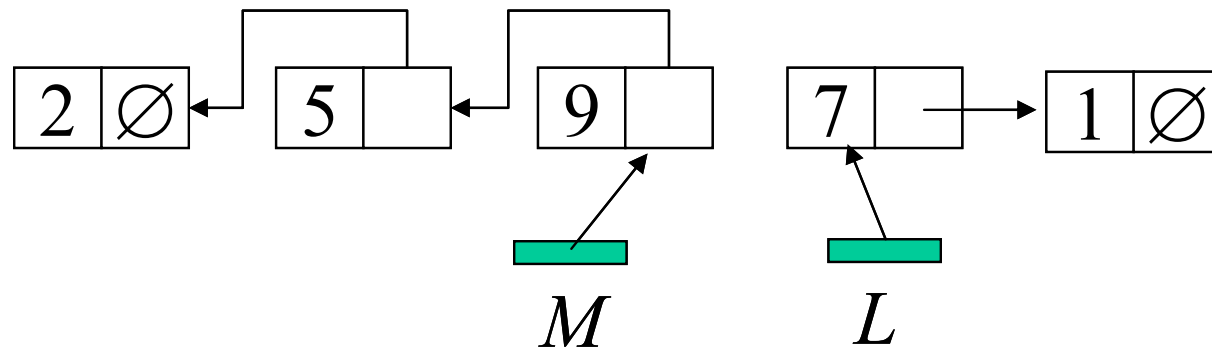
Inversione di una lista: iterazione

$\text{Tail}(L) = M;$



Inversione di una lista: iterazione

$M = L; \quad L = T;$



Inversione di una lista: iterazione

```
REVERSE-ITER( $L$ )  
 $M \leftarrow nil$   
while not ISEMPY( $L$ ) do  
     $T \leftarrow \text{TAIL}(L)$   
     $L.next \leftarrow M$   
     $M \leftarrow L$   
     $L \leftarrow T$   
end while  
return  $M$ 
```



Inversione di una lista: ricorsione



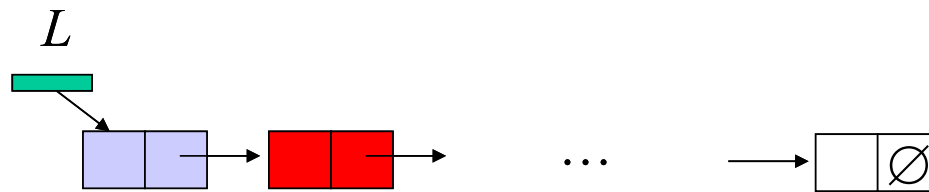
Come posso definire l'inversa
induttivamente?

- L'inversa della lista vuota o con un solo elemento è la lista stessa
- L'inversa di una lista $[a|L]$ con almeno due elementi, è l'inversa di L con a aggiunto in fondo

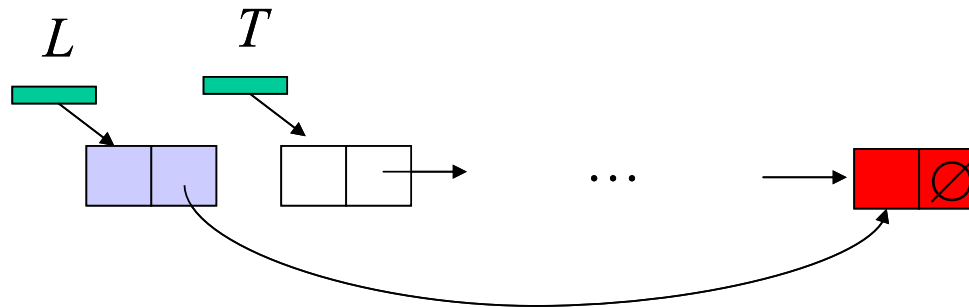
Per aggiungere in fondo devo
scandire l'inversa di L ?

Inversione di una lista: ricorsione

Se in una prima fase inverte ricorsivamente la coda di:

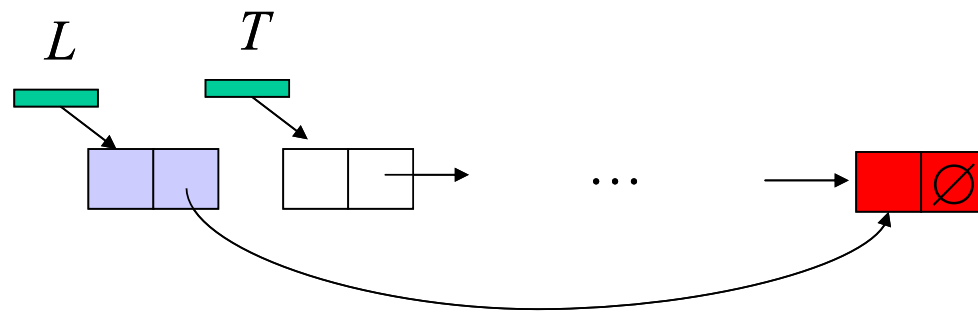


ottengo:

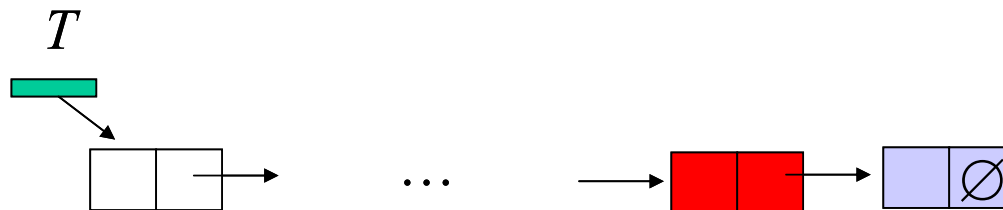


Inversione di una lista: ricorsione

Allora è facile passare da:



a:



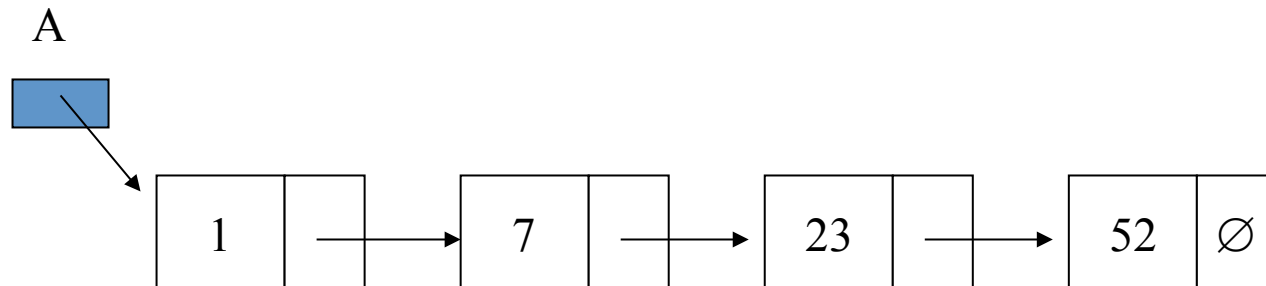
Inversione di una lista: ricorsione

```
REVERSE-REC( $L$ )  
if ISEMPTY( $L$ ) or ISEMPTY(TAIL( $L$ )) then  
    return  $L$   
else  
     $R \leftarrow$  REVERSE-REC(TAIL( $L$ ))  
     $L.next.next \leftarrow L$   
     $L.next \leftarrow nil$   
    return  $R$   
end if
```



Liste ordinate

$$A = \{1, 7, 23, 52\}$$



Possiamo rappresentare insiemi con liste ordinate

Ricerca ordinata

INORD(x, L)

▷ Pre: L ordinata

▷ Post: ritorna **true** se $x \in L$

if ISEMPTY(L) **or** $x < \text{HEAD}(L)$ **then**

return false

else

if $x = \text{HEAD}(L)$ **then**

return true

else

return INORD($x, \text{TAIL}(L)$)

end if

end if

or valutato da sin. a des.

Se $x < \text{Head}(L)$ allora $x \notin L$

... allora $x > \text{Head}(L)$ quindi $x \in L$
sse $x \in \text{Tail}(L)$

Inserimento ordinato

INSERT(x, L)

▷ Pre: L ordinata

▷ Post: L ordinata e $x \in L$

if ISEMPTY(L) **or** $x < \text{HEAD}(L)$ **then**

return CONS(x, L)

else

if $x = \text{HEAD}(L)$ **then**

return L

else

$L.\text{next} \leftarrow \text{INSERT}(x, \text{TAIL}(L))$

return L

end if

end if

Inserimento distruttivo nella coda

Cancellazione ordinata

DELETE(x, L)

▷ Pre: L ordinata

▷ Post: L ordinata e $x \notin L$

if ISEMPTY(L) **or** $x < \text{HEAD}(L)$ **then**
 return L

else

if $x = \text{HEAD}(L)$ **then**
 return TAIL(L)

Questo valore serve per
“ricucire” la lista

else

$L.\text{next} \leftarrow \text{DELETE}(x, \text{TAIL}(L))$

return L

end if

end if

In questo punto la lista
viene ricucita

Unione

UNION(L, M)

▷ Pre: L, M ordinate

▷ Post: ritorna la nuova lista $L \cup M$

if ISEMPTY(L) **then**

return CLONE(M)

else

if ISEMPTY(M) **then**

return CLONE(L)

else

if HEAD(L) = HEAD(M) **then**

return CONS(HEAD(L), UNION(TAIL(L), TAIL(M)))

else

if HEAD(L) < HEAD(M) **then**

return CONS(HEAD(L), UNION(TAIL(L), M))

else ▷ HEAD(L) > HEAD(M)

return CONS(HEAD(M), UNION(L , TAIL(M)))

end if

end if

end if

end if

Questo algoritmo è $O(n)$

Ordinamento per fusione

Come per i vettori; la fusione è più semplice, un po' più complicata la divisione:

Mergesort(L):

dividi L in due parti uguali, L ed M

$L = \text{Mergesort}(L)$

$M = \text{Mergesort}(M)$

ritorna la fusione ordinata di L ed M

Merge Sort

MERGESORT(L)

▷ Post: L ordinata

if ISEMPTY(L) **or** ISEMPTY(TAIL(L)) **then**

return L

else

$M \leftarrow \text{SPLIT}(L)$ ▷ L ed M sono le due metà di L

$L \leftarrow \text{MERGESORT}(L)$

$M \leftarrow \text{MERGESORT}(M)$

return MERGE(L, M)

end if

Fondi (Merge)

MERGE(L, M)

▷ Pre: L, M ordinate

▷ Post: ritorna una lista N ordinata modificando i puntatori in L ed M tale che $Len(N) = Len(L) + Len(M)$

Molto simile a Union: dove sono le differenze?

Dividi (Split): prima versione

$\text{SPLITAFTER}(L, n)$

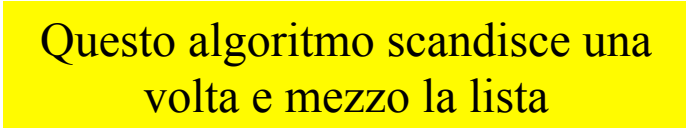
▷ Pre: $0 \leq n \leq \text{Len}(L)$

▷ Post: divide distruttivamente L dopo n elementi ritornandone la seconda parte

$\text{SPLIT}_1(L)$

$n \leftarrow \text{LENGTH}(L)$

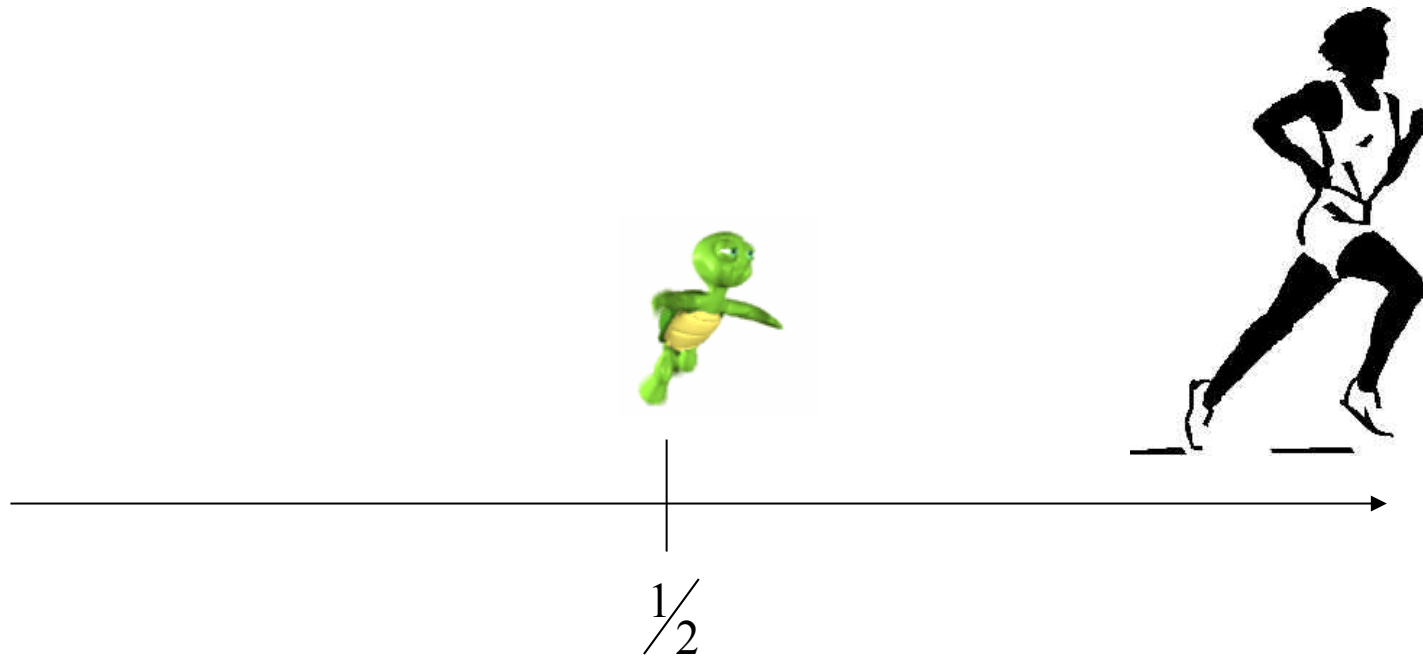
return $\text{SPLITAFTER}(L, n/2)$



Questo algoritmo scandisce una volta e mezzo la lista

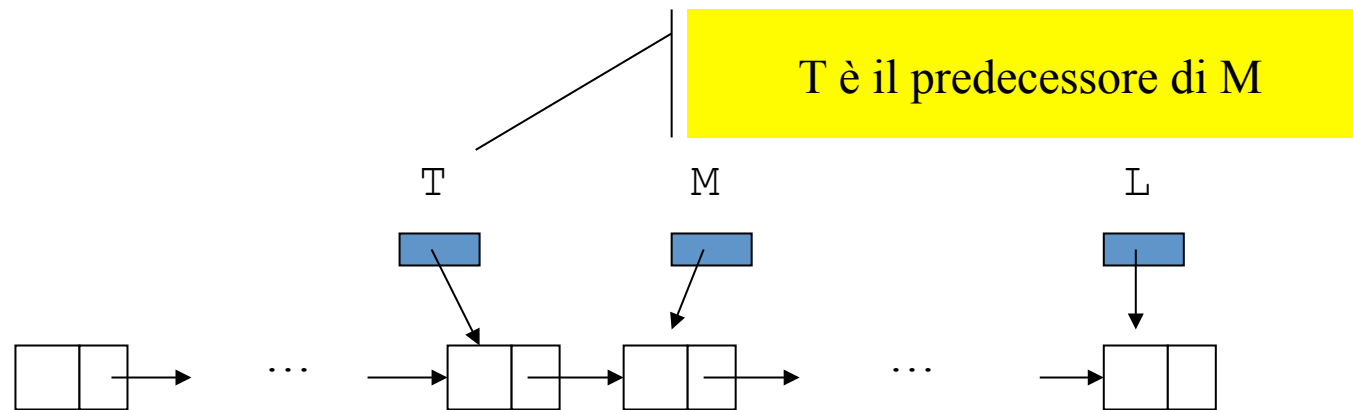
Dividi: seconda versione

Idea: percorriamo la lista con due puntatori, uno dei quali si muova a velocità doppia dell'altro.



Dividi: seconda versione

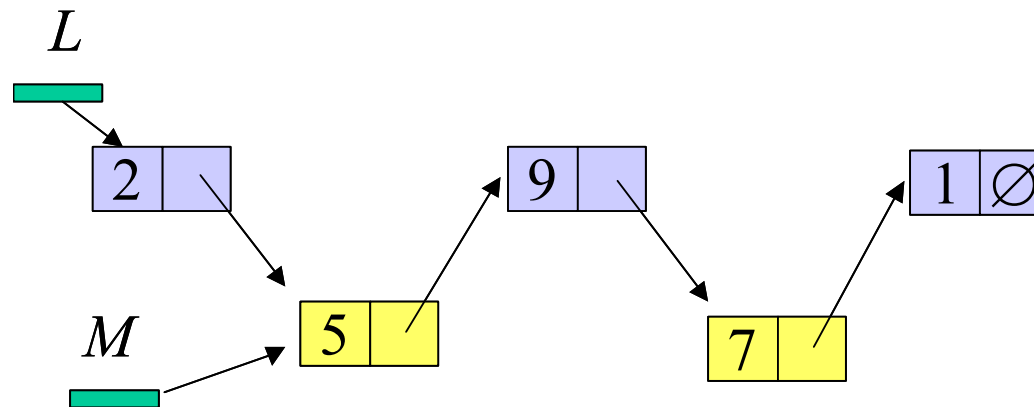
Idea: percorriamo la lista con due puntatori, uno dei quali si muova a velocità doppia dell'altro.



```
M = Tail(T); T.next = nil;
```

Dividi: terza versione

Idea: dividiamo la lista unendo gli elementi di posto pari in una nuova lista, e lasciando quelli di posto dispari.



Dividi: terza versione

Idea: dividiamo la lista unendo gli elementi di posto pari in una nuova lista.

