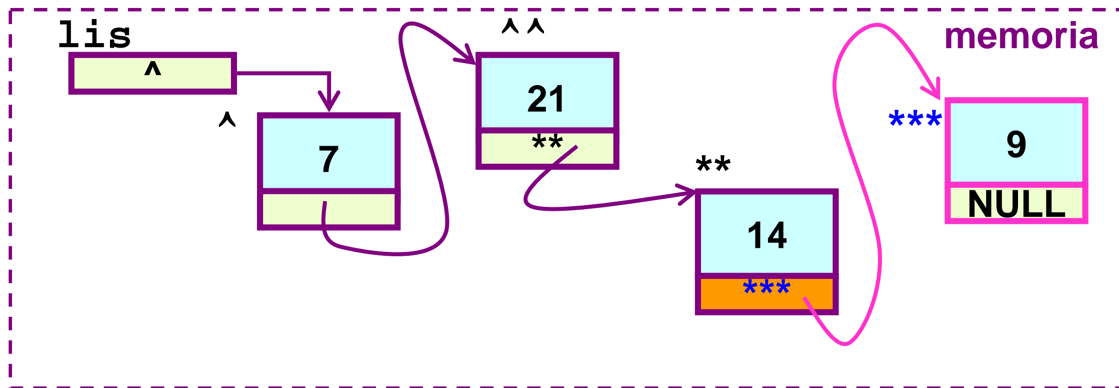
Previously on TDP …



LISTA rappresentata mediante "struct e puntatori"

```
TipoLista lis
```

DISEGNARE

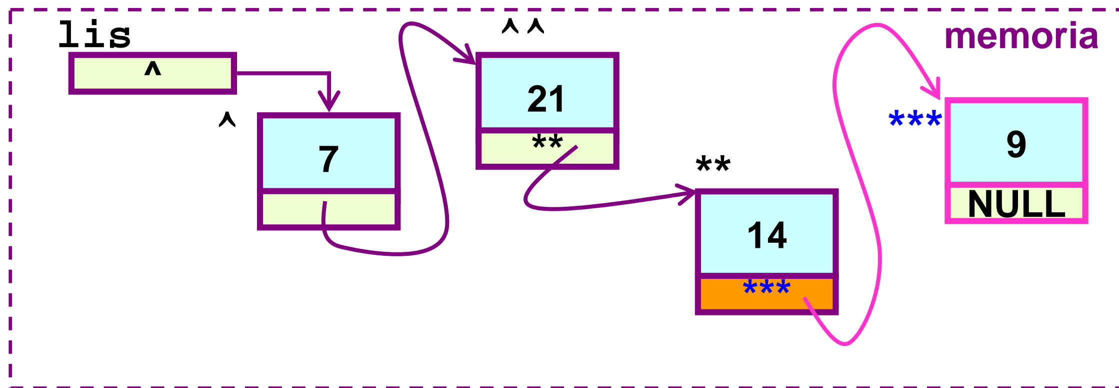
```
aux
```

in modo che

```
aux->info
```

sia il campo info contenente "21"

Previously on TDP …



LISTA rappresentata mediante "struct e puntatori"

`TipoLista lis`

DISEGNARE

aux

in modo che

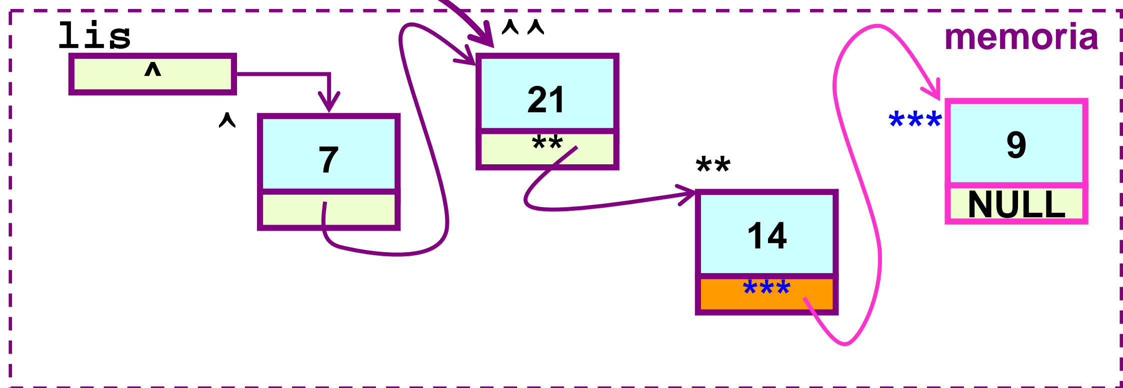
aux->info

sia il campo info contenente "21"

aux

???

Previously on TDP …



LISTA rappresentata mediante "struct e puntatori"

`TipoLista lis`

`lis->info ==` quanto vale? ☺

`lis->next == ^^;` /*cosa è ^^ ?*/ ☺

che cosa è `(*lis)`? ☺

aux

^^

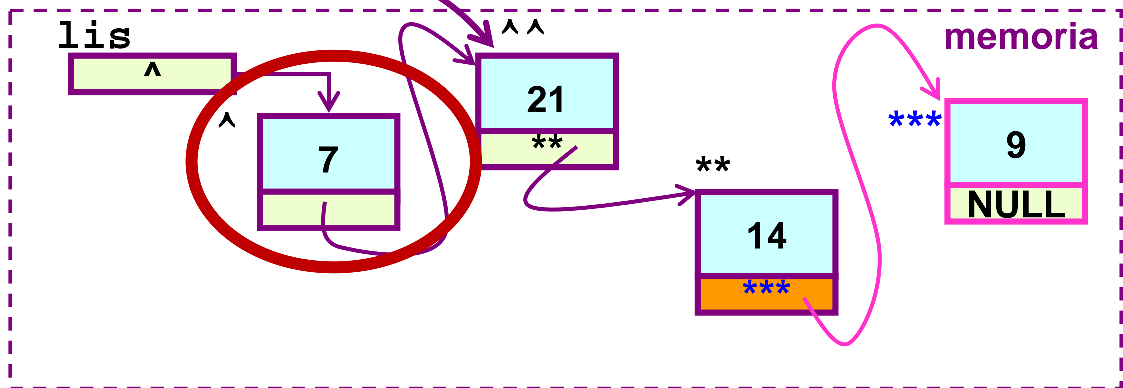
DISEGNARE

aux

in modo che

`aux->info`

sia il campo info contenente "21"



LISTA rappresentata mediante "struct e puntatori"

`TipoLista lis`

`lis->info ==` quanto vale? 7

`lis->next == ^^;` /* ^^ è l'indirizzo del secondo nodo */

che cosa è (*lis)?

DISEGNARE

aux

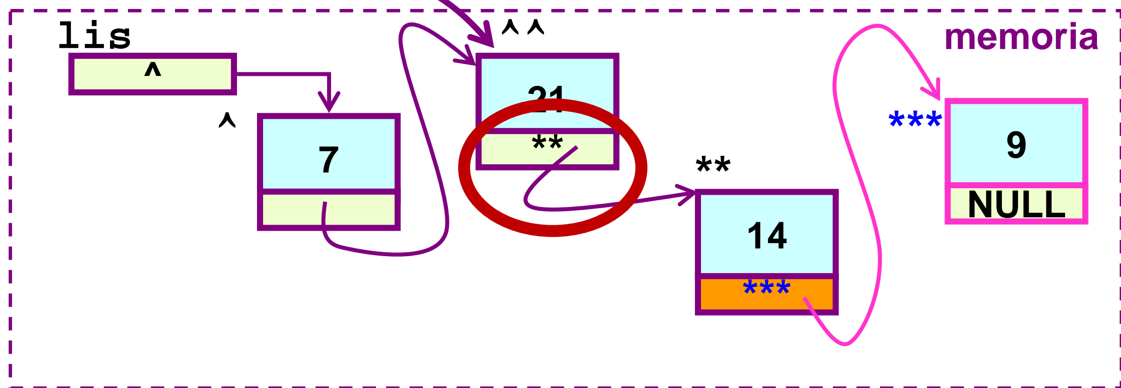
in modo che

aux->info

sia il campo info contenente "21"

Allora cosa contiene

aux->next ??



LISTA rappresentata mediante "struct e puntatori"

TipoLista lis

lis->info == ? 7

lis->next == ^^; /* ^^ è l'indirizzo del secondo nodo */

che cosa è (*lis)?

DISEGNARE

aux

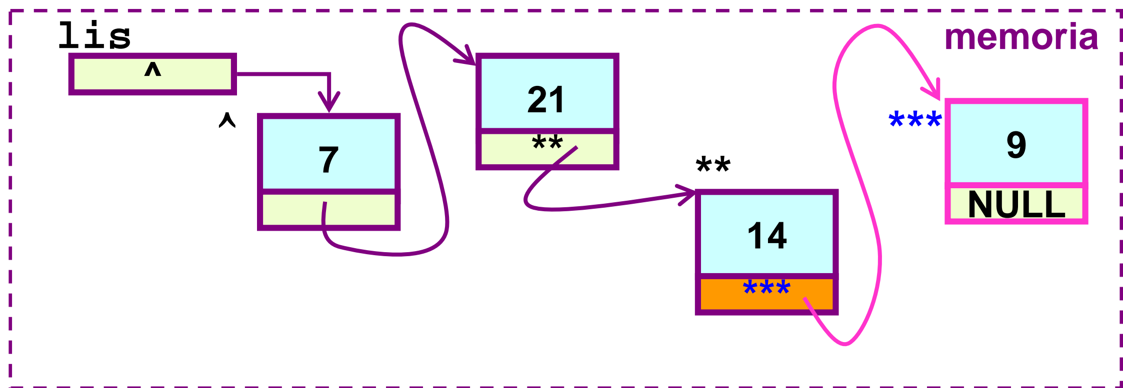
in modo che

aux->info

sia il campo info contenente "21"

Allora cosa contiene aux->next ??

aux->next contiene **



LISTA rappresentata mediante "struct e puntatori"

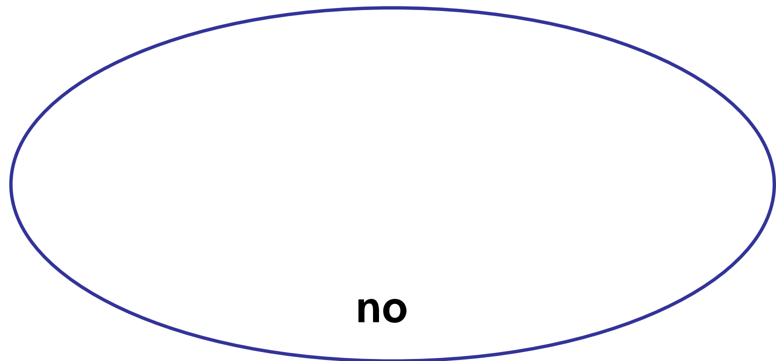
```
TipoLista lis
```

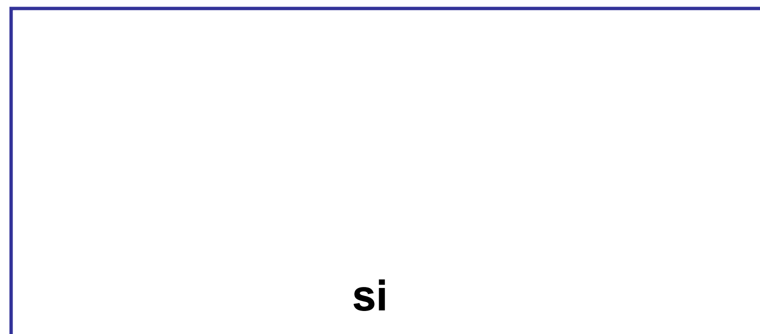
```
lis->info == cosa?

lis->next == ^^; /*cosa è ^^  ?*/

che cosa è (*lis)?
```

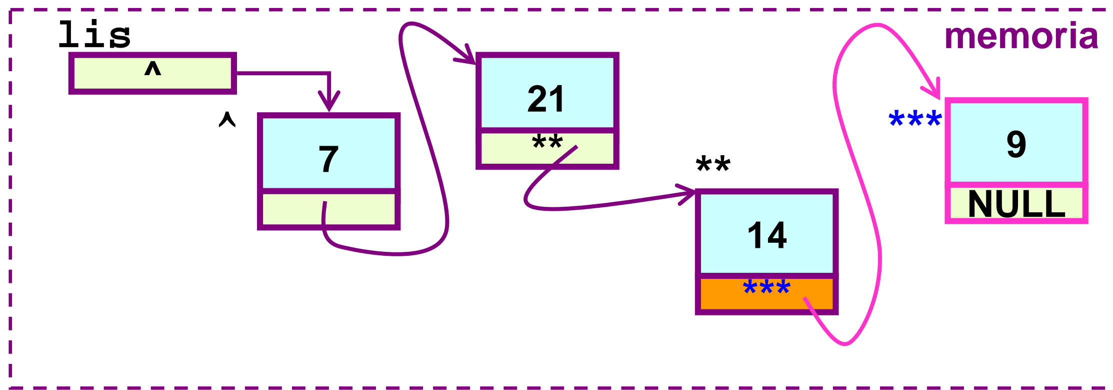
Dichiarare

aux

(quella di prima)







LISTA rappresentata mediante "struct e puntatori"

`TipoLista lis`

`lis->info ==` cosa?

`lis->next == ^^;` /*cosa è ^^ ?*/

che cosa è `(*lis)`?

Dichiarare

aux

(quella di prima)

no

si

0) resLista = NULL (init lista risultato)

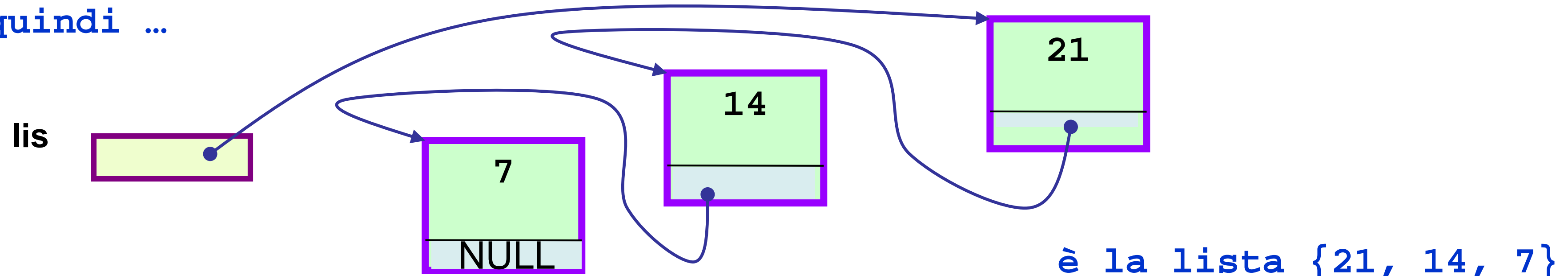
1) lettura dato

2) inserimento del dato in testa a resLista

} n volte

```
TipoLista costrLista (int n) {
  TipoLista resLista = NULL;                    /* 0) */
  int i;    TipoElem dato;

 for (i=1; i<=n; i++) {
   printf (“nuovo elem (intero): “);
   leggiElem(&dato);
      /* o magari scanf("%d", &dato); se sono interi */    /* 1) */
   insTestaLista(&resLista, dato);               /* 2) */
 }
return resLista;
}
```

Prima viene inserito 7, poi, in testa, viene inserito 4, e poi 21, quindi …



è la lista {21, 14, 7}

And now … **Lezione 23 (LISTE-2)**

Rappresentazione concreta del Tipo di Dati Astratto LISTA.

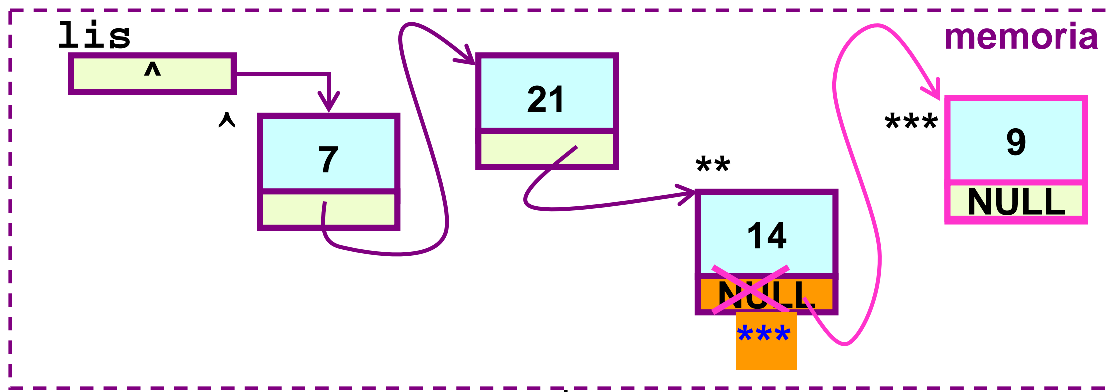
- **inserimento in lista**
  - **tecnica di aggiunta di un nodo in coda alla lista**
  - **costruzione di una lista di n nodi con ins. in coda**

- **gestione di una lista con la tecnica del *record generatore***
  - **costruzione di una lista con ins. in coda**

- **deallocazione di lista**
  - **funzione di deallocazione**

- **ancora inserimento in lista**
  - **funzione di inserimento in coda alla lista**
  - **gestione di una lista con la tecnica del *record generatore***

LISTA "struct e puntatori": Tecnica di inserimento in coda



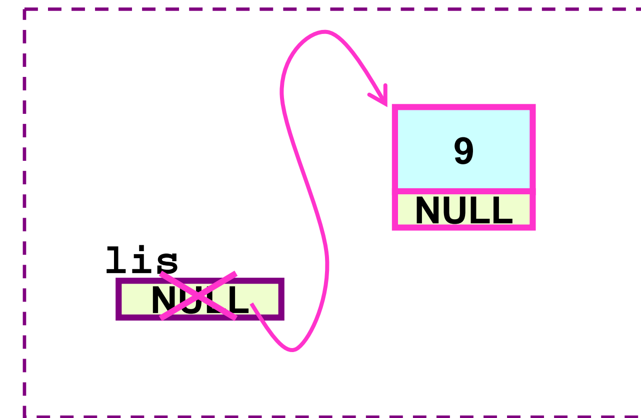
Immaginiamo di aver già costruito la lista **(7, 21, 14)**: aggiungiamo **9**

CASO GENERICO (`list!=NULL`)
in questo caso bisogna sapere chi è l'ultimo nodo della lista per accodare il nuovo elemento (assumiamo di usare un puntatore **`ultimo`** per mantenere l'informazione su *"chi è l'ultimo"*)

```
ultimo->next = malloc(sizeof(TipoNodo));
ultimo = ultimo->next;
ultimo->info = 9;
ultimo->next = NULL;
```

CASO LISTA VUOTA (lis==NULL)
in questo caso bisogna solo creare un nuovo nodo da attaccare a list

```
lis = malloc(sizeof(TipoNodo));
lis->info = 9;
lis->next = NULL;
```



costruzione lista di n nodi interi (con inserimento in coda)

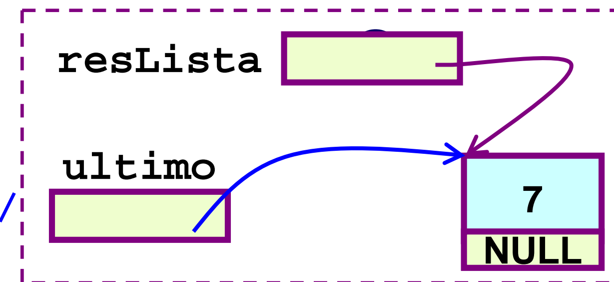
```
...typedef int TipoElem;...
```

restituisce alla funzione chiamante il puntatore al nodo iniziale della lista costruita in memoria

```
TipoLista costrCoda (int n) {
                    TipoLista resLista = NULL;
        0)          TipoNodo * ultimo;
                          int i;      TipoElem dato;

    1)
      1.1) se n == 0 ? O negativo!! … che restituire?
      1.2) lettura dato

    2)
      2.1) inserimento CASO LISTA VUOTA (in resLista)
      2.2) assegnazione ultimo

    3) lettura dato                     }
                                        } n-1 volte
    4) inserimento CASO GENERICO        }
       (con assestamento ultimo
}
```

lista di n nodi interi (con inserimento in coda): **primo caso - lista vuota**

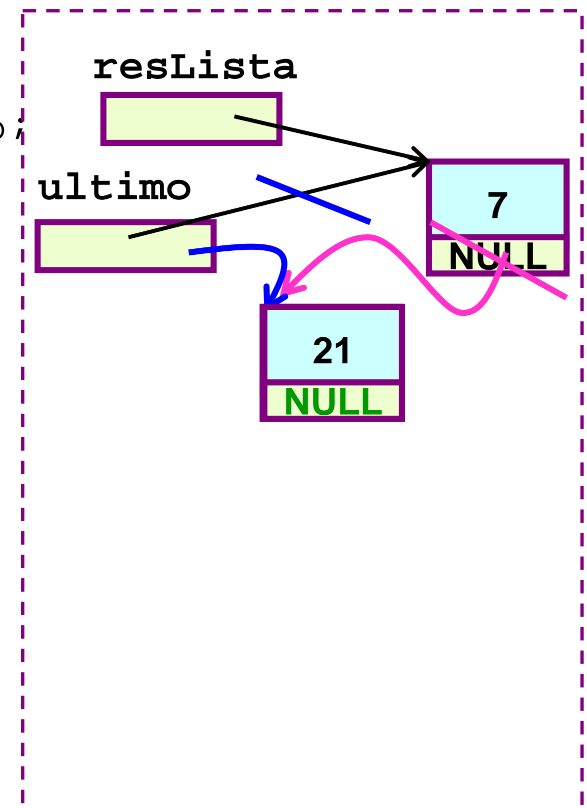
```
...typedef int TipoElem;...

TipoLista costrCoda (int n) {
  TipoLista resLista = NULL;       TipoNodo * ultimo;
  int i;                           TipoElem dato;

if (n<=0) return NULL;                                   /* 1.1) */

 printf("primo elem? "); leggiElem(&dato);               /* 1.2) */

 resLista = malloc(sizeof(TipoNodo));                    /* 2.1) */
 resLista->info = dato;
 resLista->next = NULL;
```




```
 ultimo = resLista;        /* 2.2) ultimo deve puntare all'ultimo
                                    nodo della lista, sempre*/
...   /* resto della costruzione … passi 3 e 4 */
return resLista;
}
```

lista di n nodi interi (con inserimento in coda): caso generico

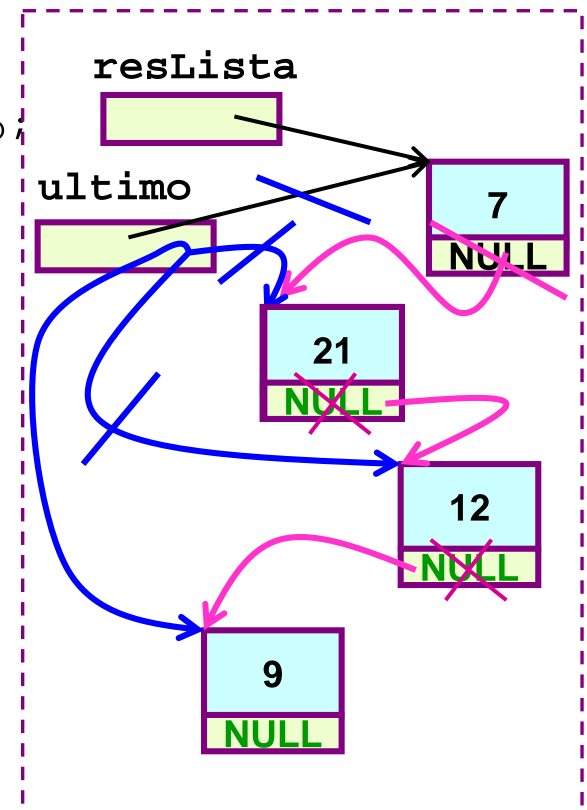
```
...typedef int TipoElem;...

TipoLista costrCoda (int n) {
  TipoLista resLista = NULL;  TipoNodo * ultimo;
  int i;                      TipoElem dato;
 if (n<=0) return NULL;
               ...                     /* 1)*/
               ...                     /* 2) */
 resLista->info = dato;
 resLista->next = NULL;
 ultimo = resLista;                    /*ultimo …*/

               /* cosa succede nel passo 3) */

    ultimo->next = malloc(sizeof(TipoDato));      /* 4) */
    ultimo = ultimo->next;
    ultimo->info = dato;
    ultimo->next = NULL;

return resLista; }
```



si nota che tutti i NULL inseriti nei campi next, a meno dell'ultimo, vengono cancellati da altro contenuto, praticamente appena dopo essere stati asssegnati ... il cambiamento nella prossima slide evita queste assegnazioni inutili

lista di n nodi interi (con inserimento in coda): caso generico

```
...typedef int TipoElem;...

TipoLista costrCoda (int n) {
  TipoLista resLista = NULL;   TipoNodo * ultimo;
  int i;                       TipoElem dato;
 if (n<=0) return NULL;
                ...                           /* 1)*/
                ...                           /* 2)  */
 resLista->info = dato;
 resLista->next = NULL;
 ultimo = resLista;                           /*ultimo …*/


 for (i=0; i<n-1; i++) {
    printf("elem? ");                         /* 3) */
    scanf("%d", &dato);
    ultimo->next = malloc(sizeof(TipoDato));        /* 4) */
    ultimo = ultimo->next;
    ultimo->info = dato;
    ultimo->next = NULL;
 }
return resLista; }
```

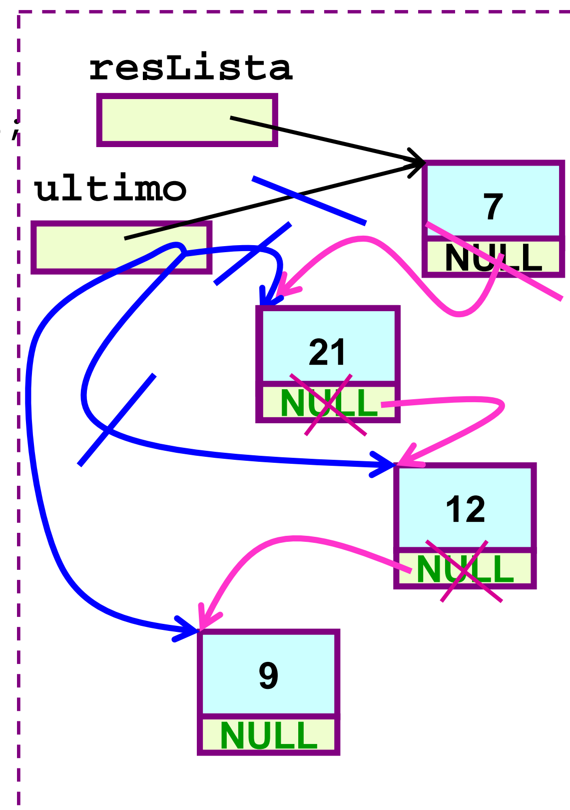


si nota che tutti i NULL inseriti nei campi next, a meno dell'ultimo, vengono cancellati da altro contenuto, praticamente appena dopo essere stati asssegnati ... il cambiamento nella prossima slide evita queste assegnazioni inutili

lista di n nodi interi (con inserimento in coda): caso generico

```
...typedef int TipoElem;...

TipoLista cost
  TipoLista re                                  timo;
  int i;                                        ;
 if (n<=0) ret
                                                */
   ...                                    /* 2) */
 resLista->info = dato;
 resLista->next = NULL;
 ultimo = resLista;                       /*ultimo …*/

 for (i=0; i<n-1; i++) {
    printf("elem? ");                     /* 3) */
    scanf("%d", &dato);
    ultimo->next = malloc(sizeof(TipoDato));        /* 4) */
    ultimo = ultimo->next;
    ultimo->info = dato;
    ultimo->next = NULL;
 }
return resLista; }
```

osservazione, per migliorare il codice

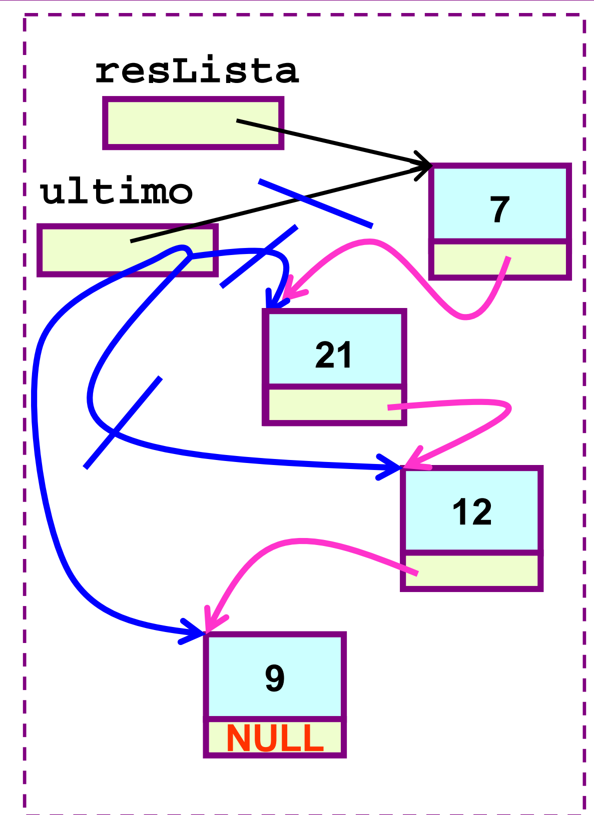


si nota che tutti i NULL inseriti nei campi next, a meno dell'ultimo, vengono cancellati da altro contenuto, praticamente appena dopo essere stati asssegnati ... il cambiamento nella prossima slide evita queste assegnazioni inutili

costruzione lista di n nodi interi (con inserimento in coda) - modifica

Nella versione attuale sovrascriviamo, ad ogni inserimento di un nuovo nodo, il campo next dell'elemento appena aggiunto: se ci limitiamo ad assegnare NULL al campo next del solo ultimo nodo immesso (cioè eseguire l'assegnazione ad ultimo->next appena usciti dal ciclo di inserimenti), risparimiamo un po' di assegnazioni...

```
            ...   ...
resLista-info = dato;
 ~~resLista->next = NULL;~~
 ultimo = resLista;

 for (i=0; i<n-1; i++) {
    printf("primo elem? ");
    scanf("%d", &dato);
    ultimo->next =
            malloc(sizeof(TipoDato));
    ultimo = ultimo->next;
    ultimo->info = dato;
    ~~ultimo->next = NULL;~~
 }
ultimo->next = NULL;
return resLista;
}
```



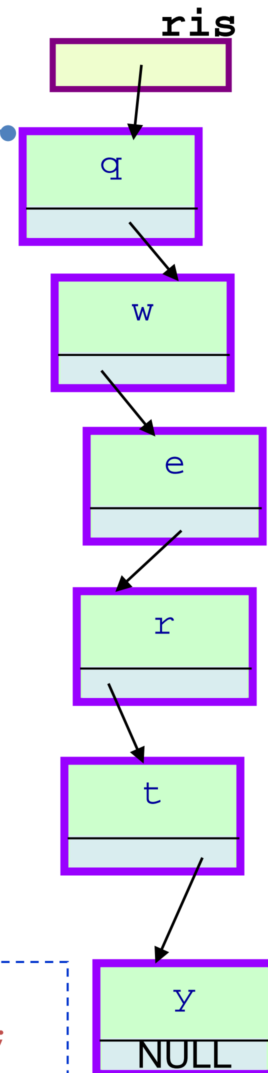
Es. funzione che legge una riga e produce la lista dei suoi caratteri

```
...typedef char  TipoElem;...

TipoLista costrCoda2 () {
     TipoLista ris = NULL;
0)   TipoNodo * ultimo;
     TipoElem dato;

1) lettura primo car. (dato)
2) ins caso LISTA VUOTA

3) lettura secondo (dato)

4) ins caso GEN.
5) lettura dato (car) successivo

6) return ris;

}
```

mentre dato!='\n'

Assumiamo almeno un carattere diverso da '\n' in input

I = **qwerty\n**



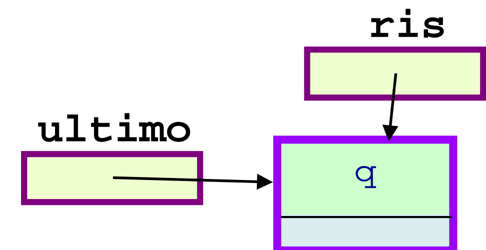
NB
Non sapendo all'inizio quanti elementi dovremo processare, adottiamo un algoritmo che 1) processa l'elemento disponibile (ultimo letto) e poi legge il prossimo

```
int main () { …
 ... list = costrCoda2();
 stampaLista (list);
...
```

Es. funzione che legge una riga e produce la lista dei suoi caratteri

```
...typedef char  TipoElem;...
TipoLista costrCoda2 () {
  TipoLista ris = NULL;
  TipoNodo * ultimo;
  TipoElem dato;
```

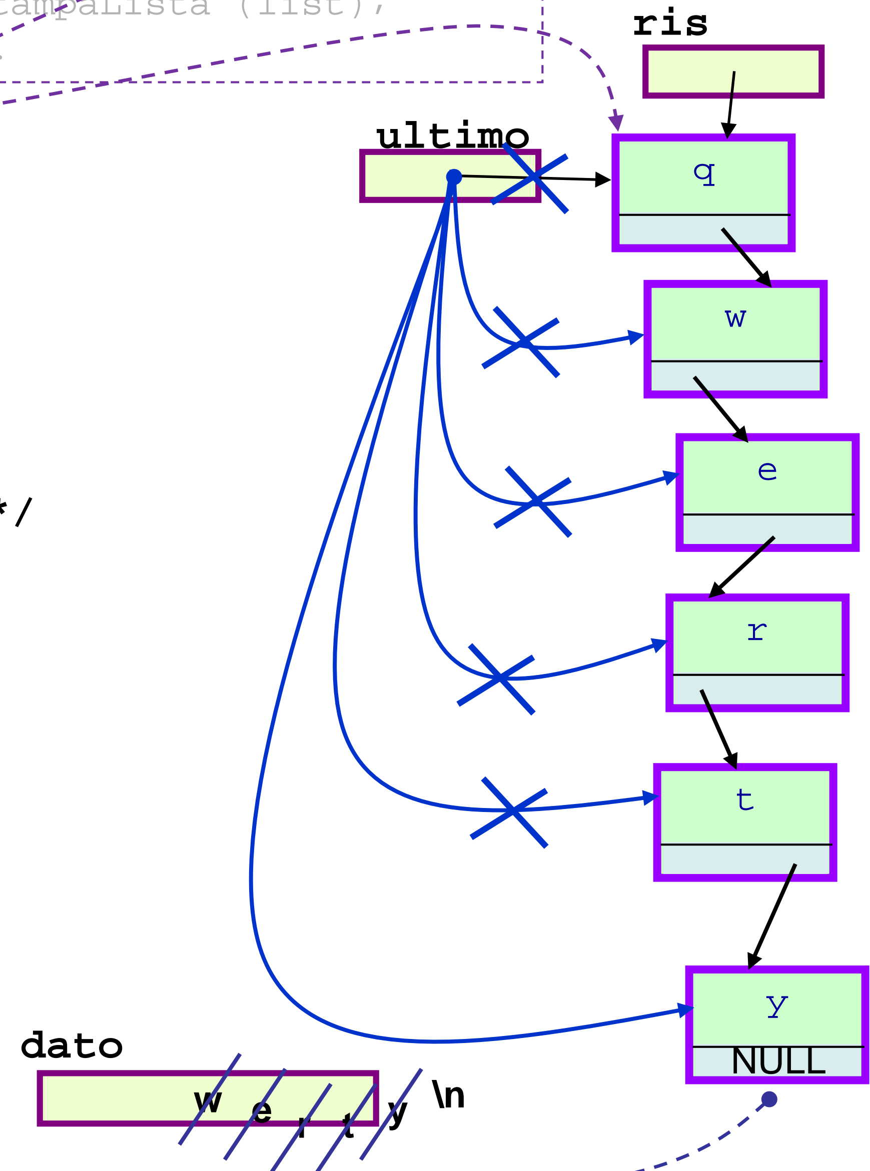
Assumiamo almeno un carattere diverso da '\n' in input
INPUT = qwerty\n




```
printf("scrivi una riga"); /* 1) */
scanf("%c", &dato);         /* (è la 'q')*/
ris = malloc(sizeof(TipoDato));
ris->info=dato;
ultimo=ris;

scanf("%c", &dato);         /* 3) (è la 'w')*/

...
}
```

Es. funzione che legge una riga e produce la lista dei suoi caratteri

```
...typedef char  TipoElem;...
TipoLista costrCoda2 () {
  TipoLista ris = NULL;
  TipoNodo * ultimo;
  TipoElem dato;
printf("scrivi una riga"); /* 1) */
scanf("%c", &dato);
ris = malloc(sizeof(TipoDato));
ris->info=dato;
ultimo=ris;
scanf("%c", &dato);  /* 3) (è la 'w')*/

 while(dato!='\n') {
  ultimo->next = malloc (…);
  ultimo = ultimo->next;
  ultimo->info = dato;
  scanf("%c", &dato);
 }
ultimo->next = NULL;
return ris;
}
```



Tecnica del record generatore

(introdotta qui per scrivereun'altra funzione che costruisce una lista con inserimenti in coda … poi pero` questa è una tecnica utile in altri casi …)

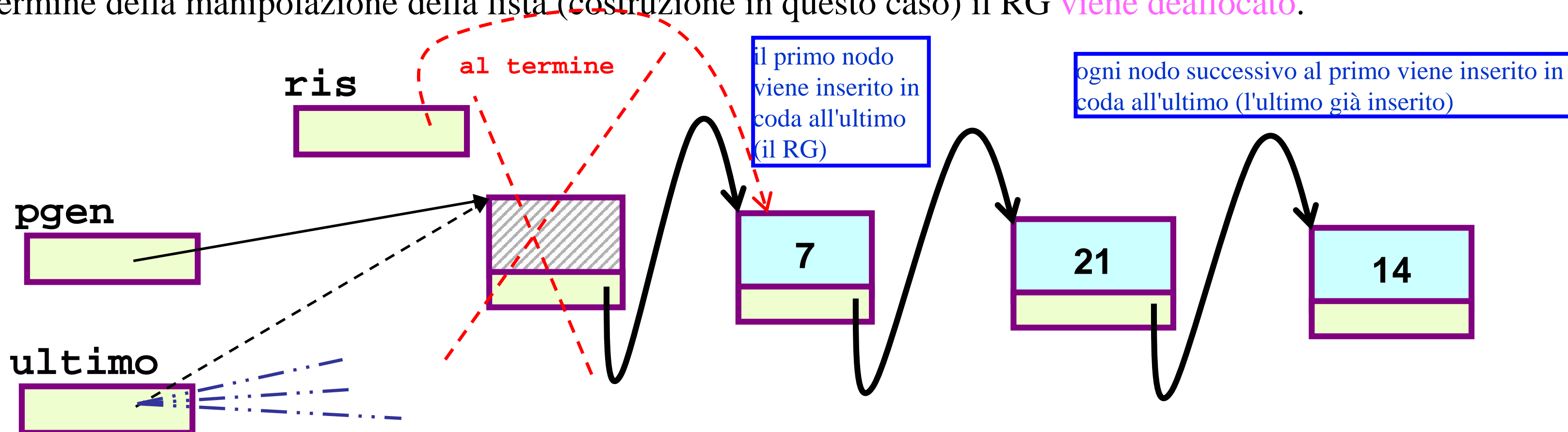
La differenza tra i due casi (INS. IN LISTA VUOTA, INS. GENERICO) è nel fatto che **`ultimo`** non può essere assegnato se la lista è vuota. ***Nel caso LISTA VUOTA, `ultimo->next` non c'è***.
Quindi non si puo` fare `ultimo->next=malloc(…).`
Per questo bisogna scrivere un codice (istruzioni) per un caso e altro codice, diverso, per il secondo caso.

Pero`, possiamo fare in modo che `ultimo` ci sia sempre, unificando i due casi: se facciamo cosi`, saremo in grado di scrivere un unico gruppo di istruzioni, funzionante in entrambi i casi.

`pgen` = puntatore al record generatore

RECORD GENERATORE = record fittizio, che non farà parte integrante della lista ma esisterà solo durante la sua manipolazione (perche' alla fine lo cancelleremo, ma con affetto…).

Durante la costruzione della lista il RG è il primo della lista (ma non contiene informazioni rilevanti nel campo info). Al termine della manipolazione della lista (costruzione in questo caso) il RG viene deallocato.



**All'inizio (lista vuota) viene creato il RG; e ultimo punta li`: la lista è vuota ma ultimo->next esiste!
Adesso possiamo usare sempre il codice del CASO GENERICO di INS.**

```
0) pgen, ultimo, …
1) allocazione record generatore (RG)
   pgen = malloc (sizeof(TipoNodo))
2) ultimo = pgen
3) lettura dato (elemento da inserire)

4) inserimento dopo l'ultimo
   4.1) ultimo->next = malloc(…)
   4.2) ultimo = ultimo->next
   4.3) ultimo->info = dato
5) lettura prossimo elemento

   } mentre non finito
     (es. dato!='\n')

6) chiusura lista
   ultimo->next = NULL

7) ris = pgen->next     (il vero primo elemento della
                                                lista)
8) eliminazione RG
```

lista da riga di testo in input

```
(typedef char TipoElem;)

struct strutturalista {
    TipoElem info;
    struct strutturalista * next;
}

typedef struct strutturalista
        TipoNodo;

typedef TipoNodo *
        TipoLista

TipoLista costruisciListaCoda() {
   TipoLista ris = NULL;
   TipoNodo * ultimo, *pgen;       /* 0) */
   TipoElem dato;
```

```
0) pgen, ultimo, …
1) allocazione record generatore (RG)
   pgen = malloc (sizeof(TipoNodo))
2) ultimo = pgen
3) lettura dato (elemento da inserire)

4) inserimento dopo l'ultimo
   4.1) ultimo->next = malloc(…)        mentre non finito
   4.2) ultimo = ultimo->next           (es. dato!='\n')
   4.3) ultimo->info = dato
5) lettura prossimo elemento

6) chiusura lista
   ultimo->next = NULL

7) ris = pgen->next     (il vero primo elemento dell
                                                 li
8) eliminazione RG
```

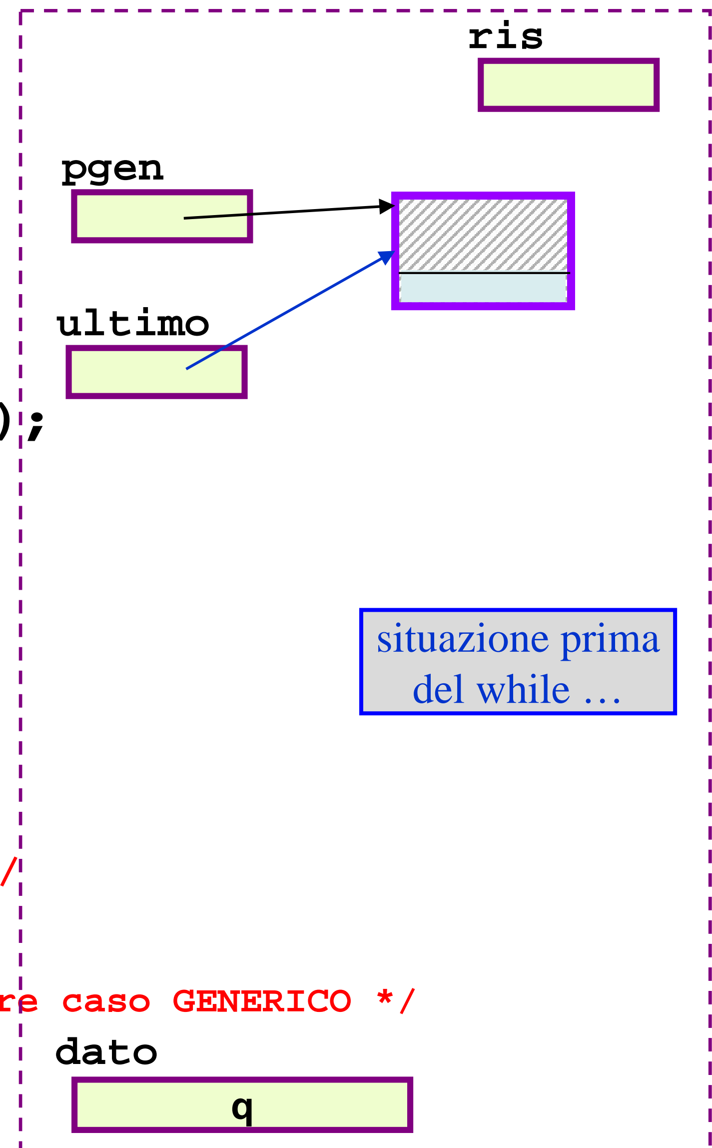
lista da riga di testo in input

```
TipoLista costruisciListaCoda() {
   TipoLista ris = NULL;
   TipoNodo * ultimo, *pgen; TipoElem dato;

                          /* 1) */
 pgen = malloc (sizeof(TipoNodo));
 if (pgen==NULL) {
  printf ("\nerrore in allocazione del RG\n");
  return NULL;
 }

 ultimo = pgen;           /* 2) */

 printf("scrivi una riga\n");
 scanf("%c",&dato);       /* 3) */

/* 4, 5; INS. sempre caso GENERICO */
 while (dato!='\n') {
   ultimo->next = malloc ...
...
```

```
0) pgen, ultimo, …
1) allocazione record generatore (RG)
   pgen = malloc (sizeof(TipoNodo))
2) ultimo = pgen
3) lettura dato (elemento da inserire)

4) inserimento dopo l'ultimo
   4.1) ultimo->next = malloc(…)         mentre non finito
   4.2) ultimo = ultimo->next            (es. dato!='\n')
   4.3) ultimo->info = dato
5) lettura prossimo elemento

6) chiusura lista
   ultimo->next = NULL

7) ris = pgen->next      (il vero primo elemento dell
                                                    li
8) eliminazione RG
```

lista da riga di testo in input

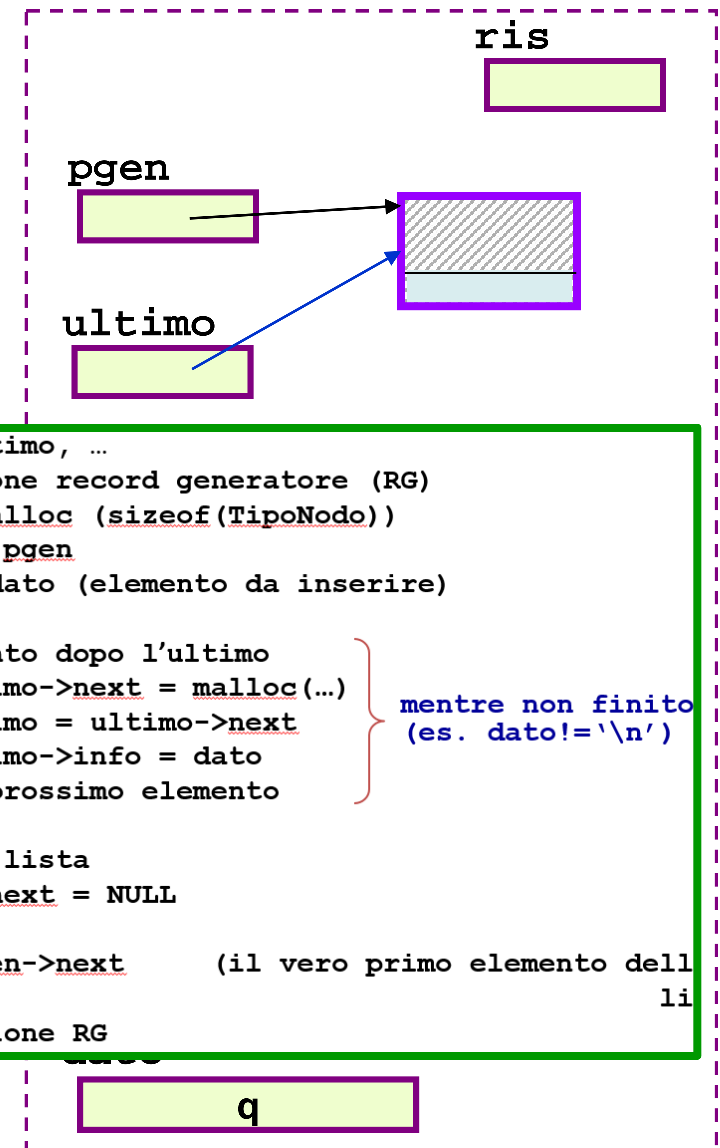
```
TipoLista costruisciListaCoda() {
    TipoLista ris = NULL;
    TipoNodo * ultimo, *pgen;
    TipoElem dato;
 /* 1) */
 pgen = malloc (sizeof(TipoNodo));
 if (pgen==NULL) {
  printf ("\nerrore in allocazione del RG\n");
  return NULL;
 }

 ultimo = pgen;            /* 2) */

 printf("scrivi una riga\n");
 scanf("%c",&dato);              /* 3) */

 while (dato!='\n') { /* 4, 5; INS. sempre caso GENERICO */
   ultimo->next = malloc ...
...
```



lista da riga di testo in input

INPUT = qwe enter

```
TipoLista costruisciListaCoda() {
   ...
 ultimo = pgen;
 printf("scrivi una riga\n");
 scanf("%c",&dato);
 while (dato!='\n') {     /*4)*/
   ultimo->next = malloc (…);
   ultimo=ultimo->next;
   ultimo->info = dato;
   scanf("%c",&dato); /*5)*/
 }
                    /* 6) */
 ultimo->next = NULL;

 ris = pgen->next;/* 7) */

 free (pgen); /* 8) */
 return ris;
}
```



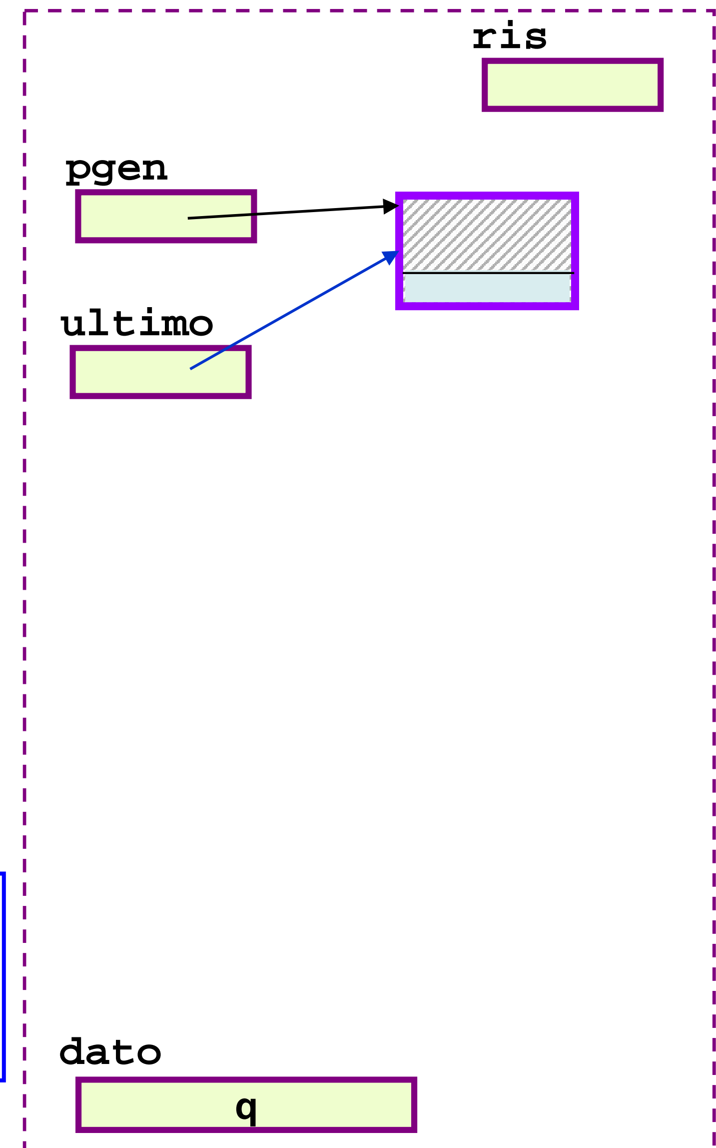

```
0) pgen, ultimo, …
1) allocazione record generatore (RG)
   pgen = malloc (sizeof(TipoNodo))
2) ultimo = pgen
3) lettura dato (elemento da inserire)

4) inserimento dopo l'ultimo
   4.1) ultimo->next = malloc(…)
   4.2) ultimo = ultimo->next
   4.3) ultimo->info = dato
5) lettura prossimo elemento
        } mentre non finito (es. dato!='\n')

6) chiusura lista
   ultimo->next = NULL

7) ris = pgen->next      (il vero primo elemento dell
                                                  li
8) eliminazione RG
```

lista da riga di testo in input

INPUT = qwe enter

```
TipoLista costruisciListaCoda() {
   ...
 ultimo = pgen;
 printf("scrivi una riga\n");
 scanf("%c",&dato);
 while (dato!='\n') {
   ultimo->next = malloc (…);
   ultimo=ultimo->next;
   ultimo->info = dato;
   scanf("%c",&dato);
 }
 ultimo->next = NULL;
 ris = pgen->next;
 free (pgen);
 return ris;
}
```

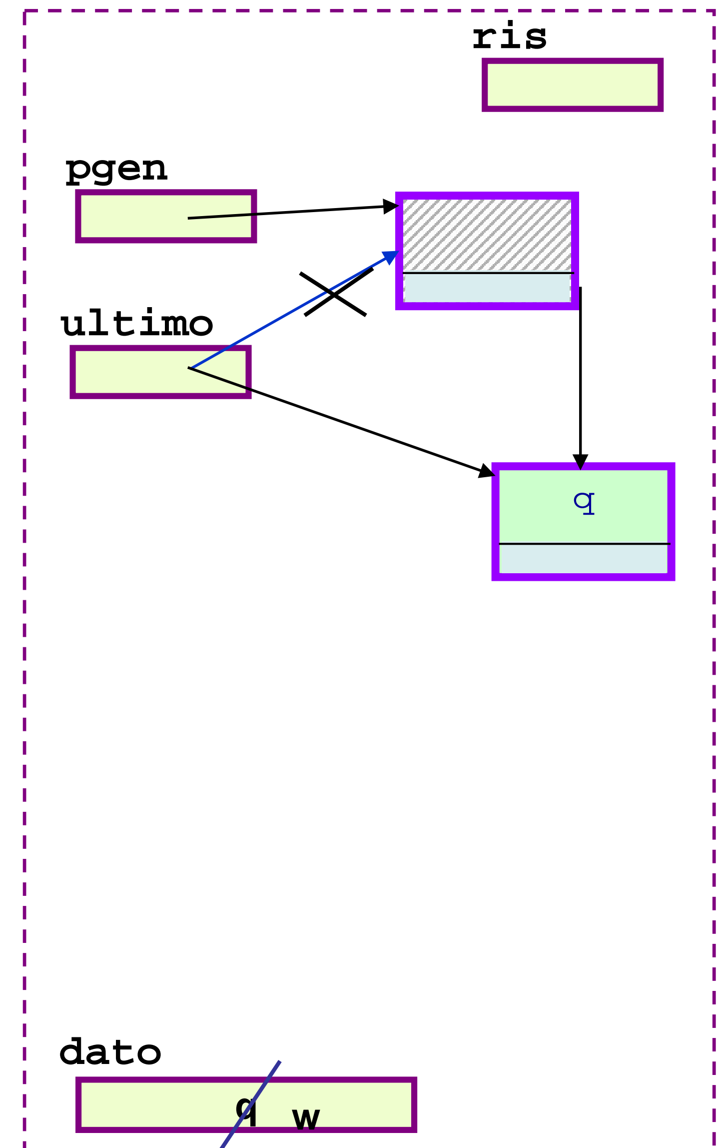
Prima di proseguire completa la figura con il risultato delle prima iterazione del ciclo



lista da riga di testo in input

INPUT = qwe enter

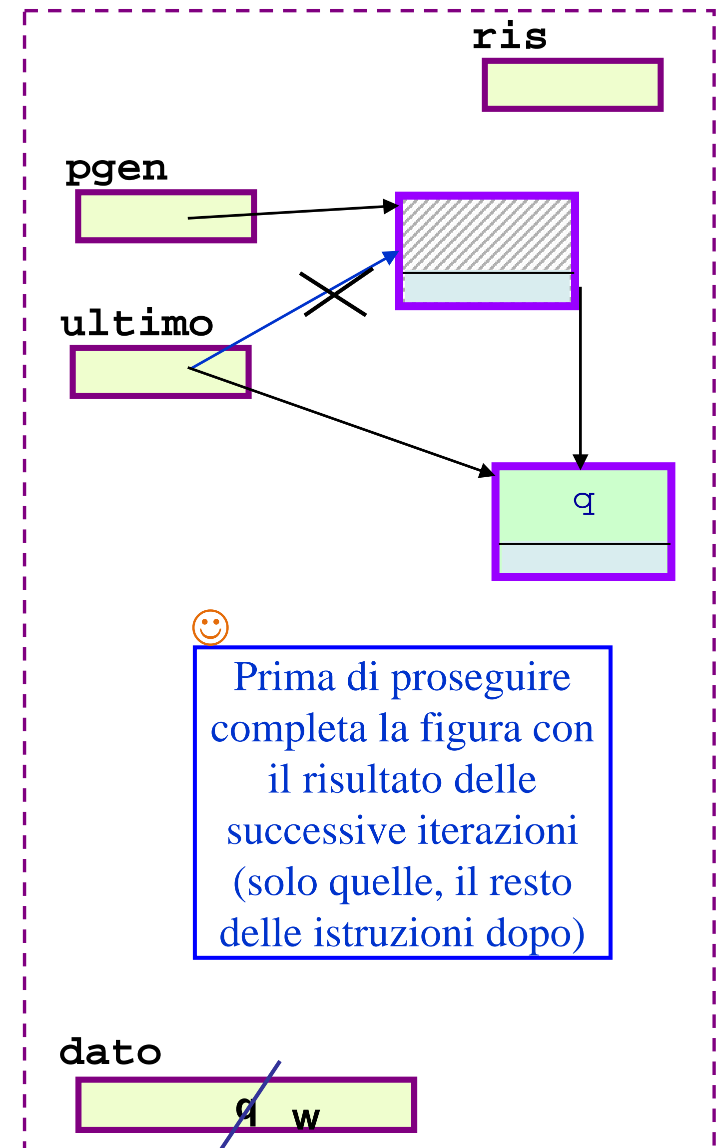
```
TipoLista costruisciListaCoda() {
   ...
 ultimo = pgen;
 printf("scrivi una riga\n");
 scanf("%c",&dato);
 while (dato!='\n') {
   ultimo->next = malloc (…);
   ultimo=ultimo->next;
   ultimo->info = dato;
   scanf("%c",&dato);
 }
 ultimo->next = NULL;
 ris = pgen->next;
 free (pgen);
 return ris;
}
```



lista da riga di testo in input

INPUT = qwe enter

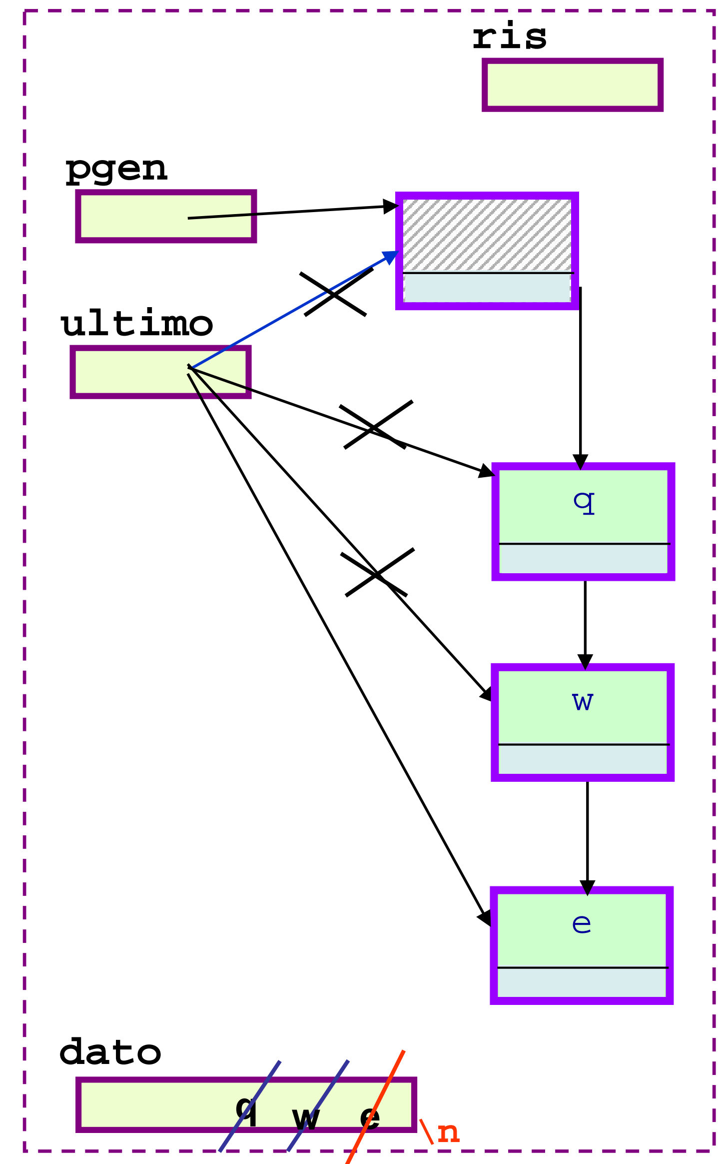
```
TipoLista costruisciListaCoda() {
   ...
 ultimo = pgen;
 printf("scrivi una riga\n");
 scanf("%c",&dato);
 while (dato!='\n') {
    ultimo->next = malloc (…);
    ultimo=ultimo->next;
    ultimo->info = dato;
    scanf("%c",&dato);
 }
 ultimo->next = NULL;
 ris = pgen->next;
 free (pgen);
 return ris;
}
```



lista da riga di testo in input

INPUT = qwe enter

```
TipoLista costruisciListaCoda() {
   ...
 ultimo = pgen;
 printf("scrivi una riga\n");
 scanf("%c",&dato);
 while (dato!='\n') {
   ultimo->next = malloc (…);
   ultimo=ultimo->next;
   ultimo->info = dato;
   scanf("%c",&dato);
 }
 ultimo->next = NULL;
 ris = pgen->next;
 free (pgen);
 return ris;
}
```

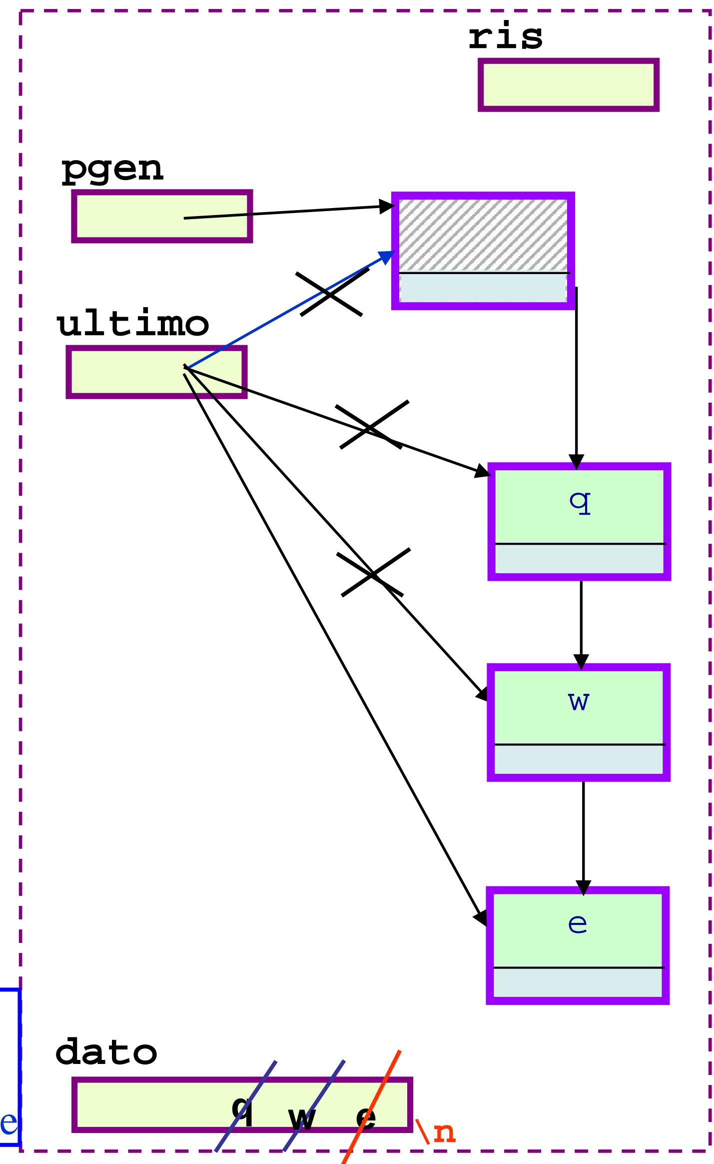


lista da riga di testo in input

INPUT = qwe enter

```
TipoLista costruisciListaCoda() {
   ...
 ultimo = pgen;
 printf("scrivi una riga\n");
 scanf("%c",&dato);
 while (dato!='\n') {
   ultimo->next = malloc (…);
   ultimo=ultimo->next;
   ultimo->info = dato;
   scanf("%c",&dato);
 }
 ultimo->next = NULL;
 ris = pgen->next;
 free (pgen);
 return ris;
}
```

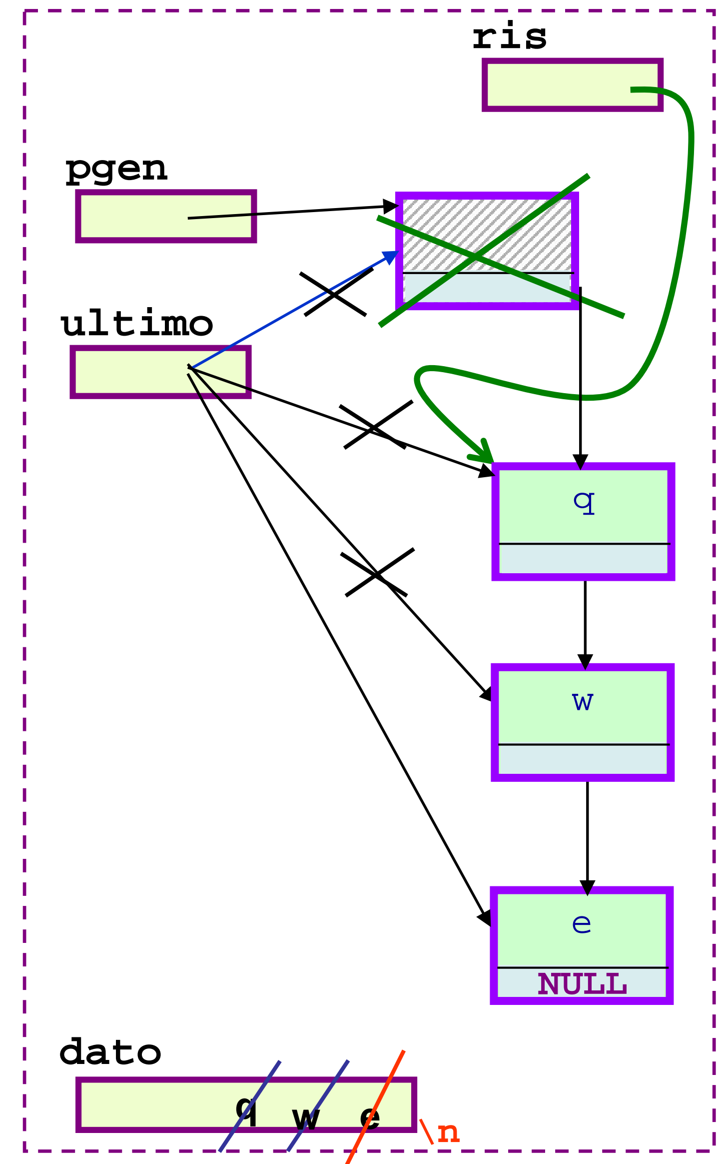
Siamo appena usciti dal while …
Prima di proseguire completa la figura
con il risultato delle istruzioni successive



lista da riga di testo in input

INPUT = qwe enter

```
TipoLista costruisciListaCoda() {
   ...
 ultimo = pgen;
 printf("scrivi una riga\n");
 scanf("%c",&dato);
 while (dato!='\n') {
   ultimo->next = malloc (…);
   ultimo=ultimo->next;
   ultimo->info = dato;
   scanf("%c",&dato);
 }
 ultimo->next = NULL;
 ris = pgen->next;
 free (pgen);
 return ris;
}
```

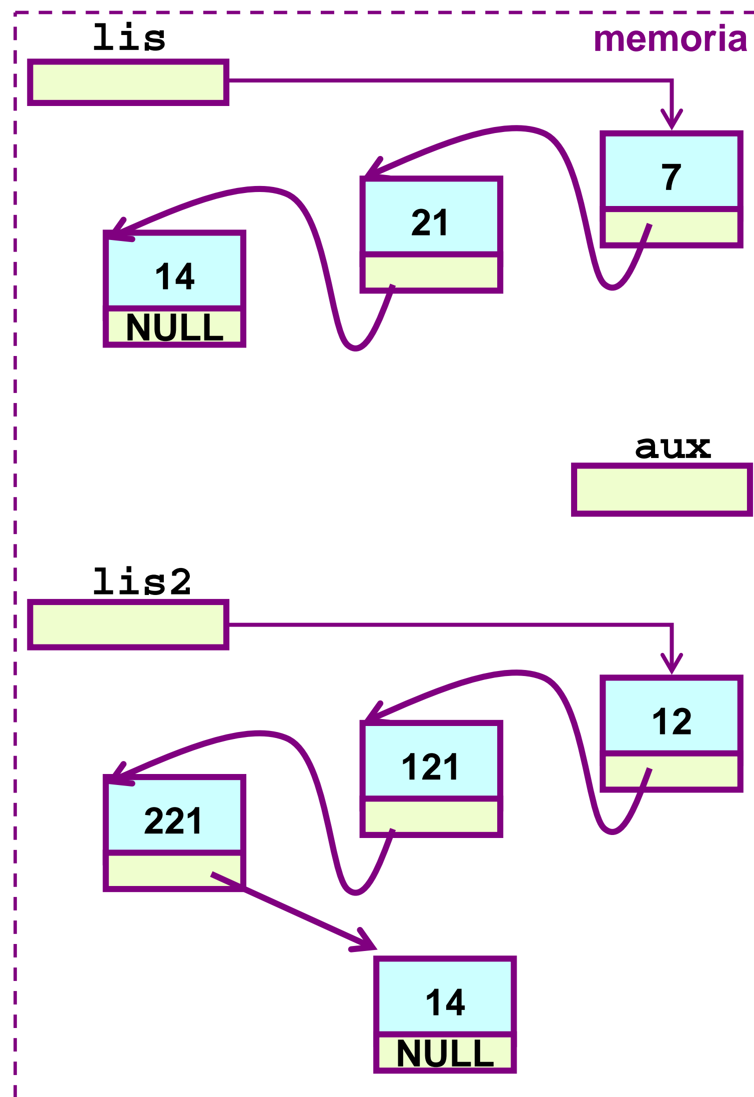


programma che legge e costruisce due liste, `lis`, `lis2`, le usa e poi le dealloca dalla memoria - `lis` con codice nella main(); `lis2` con una chiamata di funzione

```
#include ...
...
int main() {
  TipoLista lis, lis2;
   ...
/* costruzione lis */ ...
/* costruzione lis2 */ ...
...
/* deallocazione prima lista */

          free(lis);
```

No, non è questa la soluzione

per considerazioni identiche a quelle fatte a proposito di deallocazione del primo elemento di una lista

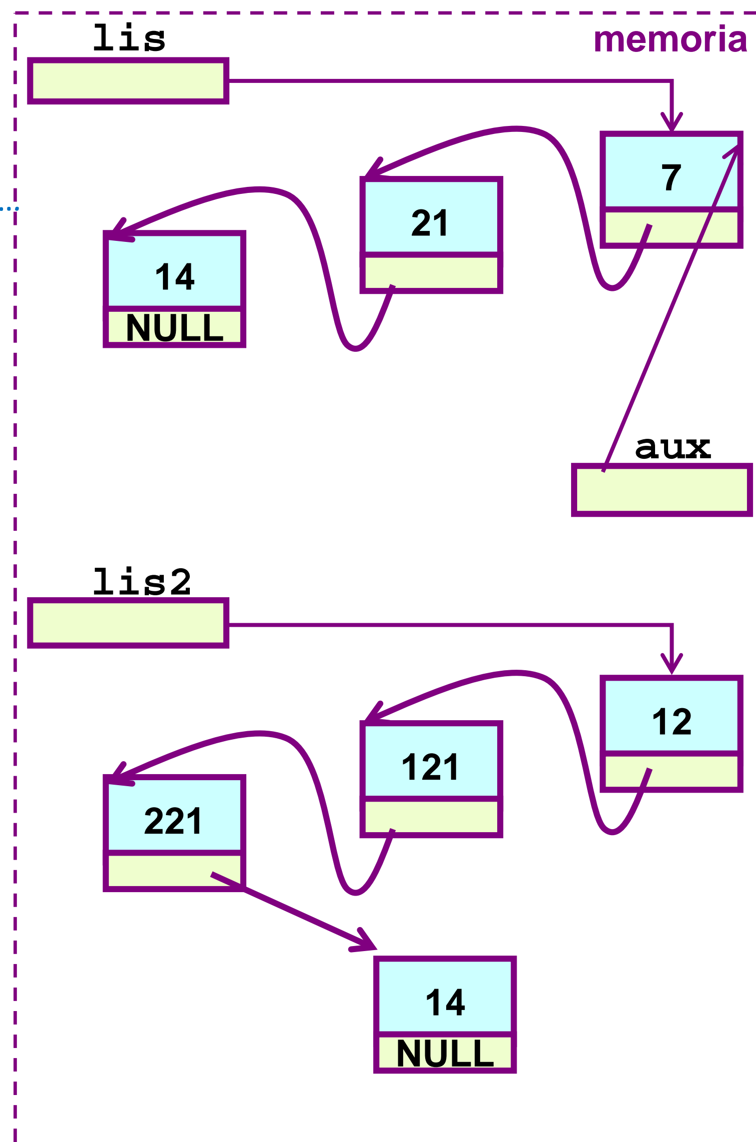


```
...
/* deallocazione prima lista */
```

... pero` , sappiamo eliminare il primo elemento della lista ...

eliminazione (primo) elem. puntato da lis

```
aux=lis;
lis=lis->next;
free(aux);
```




```
    /* deallocazione seconda lista
      mediante chiamata di funzione */
   deallocaLista(&lis2);
              /* side effect previsto */
... FINE
return 0;
}
```

```
...
/* deallocazione prima lista */
```

eliminazione (primo) elem. puntato da lis

```
aux=lis;
lis=lis->next;
free(aux);
```

qui abbiamo eliminato il primo nodo della lista; ora la lista (attuale) parte da quello che prima era il secondo nodo ...

... allora, ripetendo l'eliminazione del primo, in una lista che, progressivamente, perde un elemento per volta dalla testa ... si elimina la lista ... cioè si deallocano tutti i nodi della lista e il puntatore all'inizio della lista diventa NULL

Nella prossima slide viene eliminato il primo nodo della lista attuale (cioè il secondo della lista originale)

```
    /* deallocazione seconda lista
       mediante chiamata di funzione */
   deallocaLista(&lis2);
                  /* side effect previsto */
... FINE
return 0;
}
```

```
...
/* deallocazione prima lista */
```

eliminazione (primo) elem. puntato da lis

```
aux=lis;
lis=lis->next;
free(aux);
```

... allora, ripetendo l'eliminazione del primo, in una lista che, progressivamente, perde un elemento per volta dalla testa ...
si elimina la lista ... cioè si deallocano tutti i nodi della lista e il puntatore all'inizio della lista diventa NULL

qui è stato eliminato il secondo nodo della lista originale – che era il primo della lista attuale ...

e nella prossima slide eliminamo, sempre eseguendo l'algoritmo qui sopra, il terzo nodo della lista originale, che è qui appena diventato il primo della lista attuale ... non so se mi spiego ...)

```
    /* deallocazione seconda lista
       mediante chiamata di funzione */
   deallocaLista(&lis2);
                /* side effect previsto */
... FINE
return 0;
}
```

memoria
lis
14
NULL
21
7
aux
lis2
12
121
221
14
NULL

```
...
/* deallocazione prima lista */
```

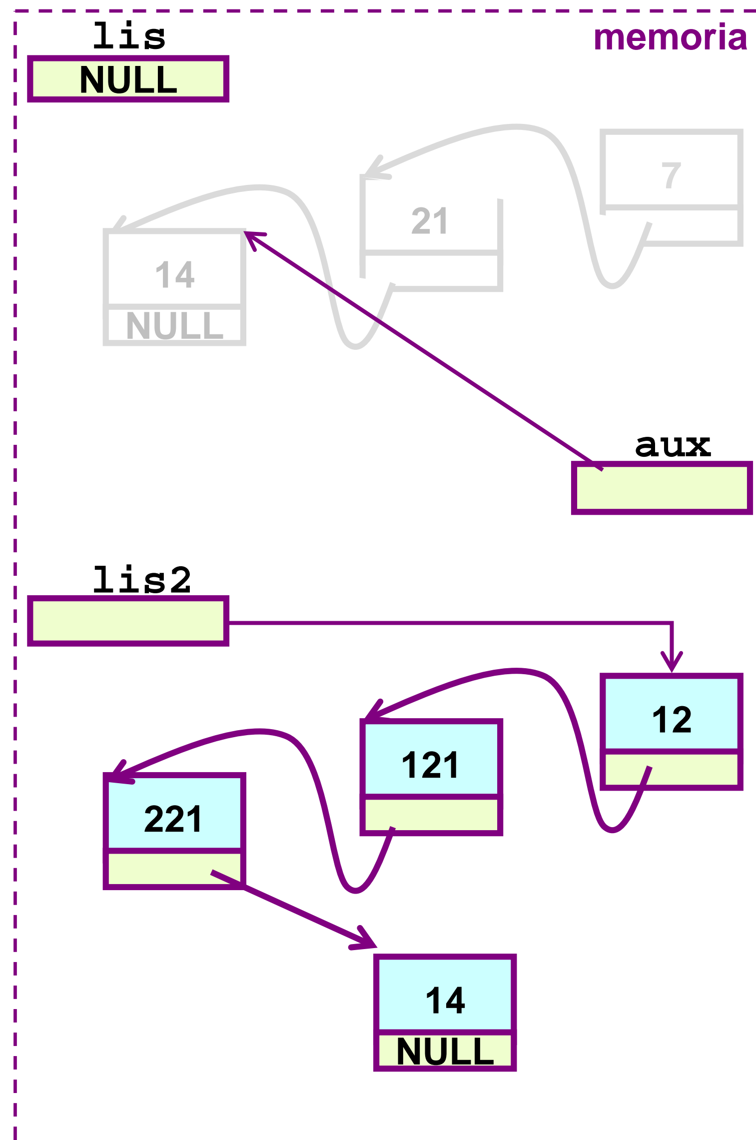
eliminazione (primo) elem. puntato da lis

```
aux=lis;
lis=lis->next;
free(aux);
```

... allora, ripetendo l'eliminazione del primo, in una lista che, progressivamente, perde un elemento per volta dalla testa ... si elimina la lista ... cioè si deallocano tutti i nodi della lista e il puntatore all'inizio della lista diventa NULL

(qui viene eliminato il terzo nodo della lista originale - che era diventato il primo della lista attuale nella slide precedente...)
Nel processo, `lis` diventa NULL
(verificare ... quale istruzione, delle tre qui sopra, fa ottenere questo effetto?)

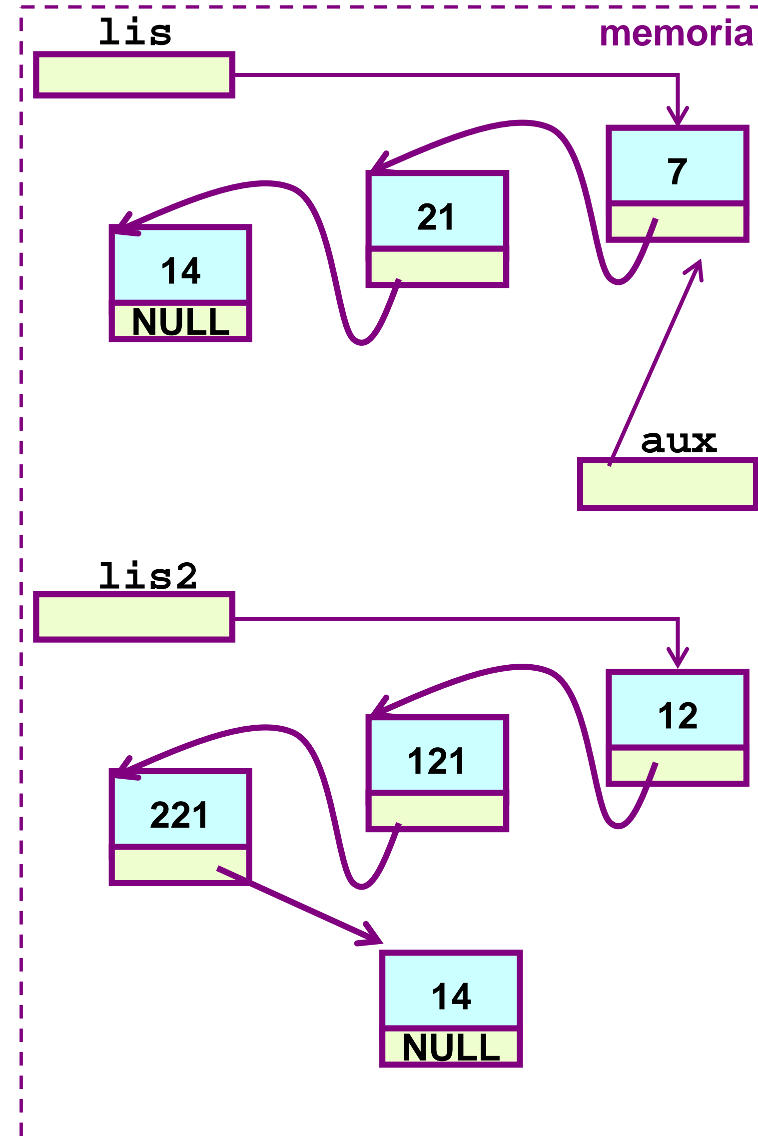
```
    /* deallocazione seconda lista
       mediante chiamata di funzione */
   deallocaLista(&lis2);
                  /* side effect previsto */
... FINE
return 0;
}
```



```
...
/* deallocazione prima lista */
```

eliminazione (primo) elem. puntato da lis

```
aux=lis;
lis=lis->next;
free(aux);
```

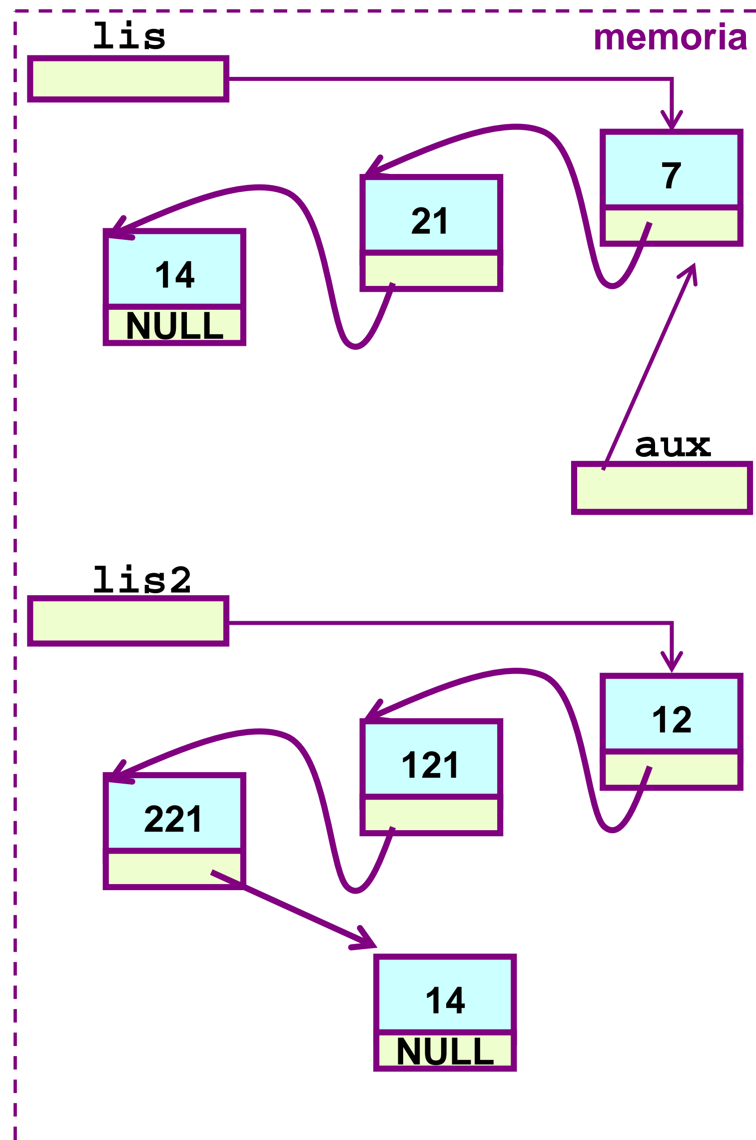



```
/* deallocazione seconda lista
      mediante chiamata di funzione */
   deallocaLista(&lis2);
                            /* side effect previsto */
... FINE
return 0;
}
```

```
...
/* deallocazione prima lista */
```

eliminazione (primo) elem. puntato da lis
```
aux=lis;
lis=lis->next;
free(aux);
```
mentre lis non NULL

```
   while(lis) {
     aux = lis;
     lis = lis->next;
     free(aux);
   }
   /* ora lis==NULL */
```




```
/* deallocazione seconda lista
      mediante chiamata di funzione */
   deallocaLista(&lis2);
                              /* side effect previsto */
... FINE
return 0;
}
```

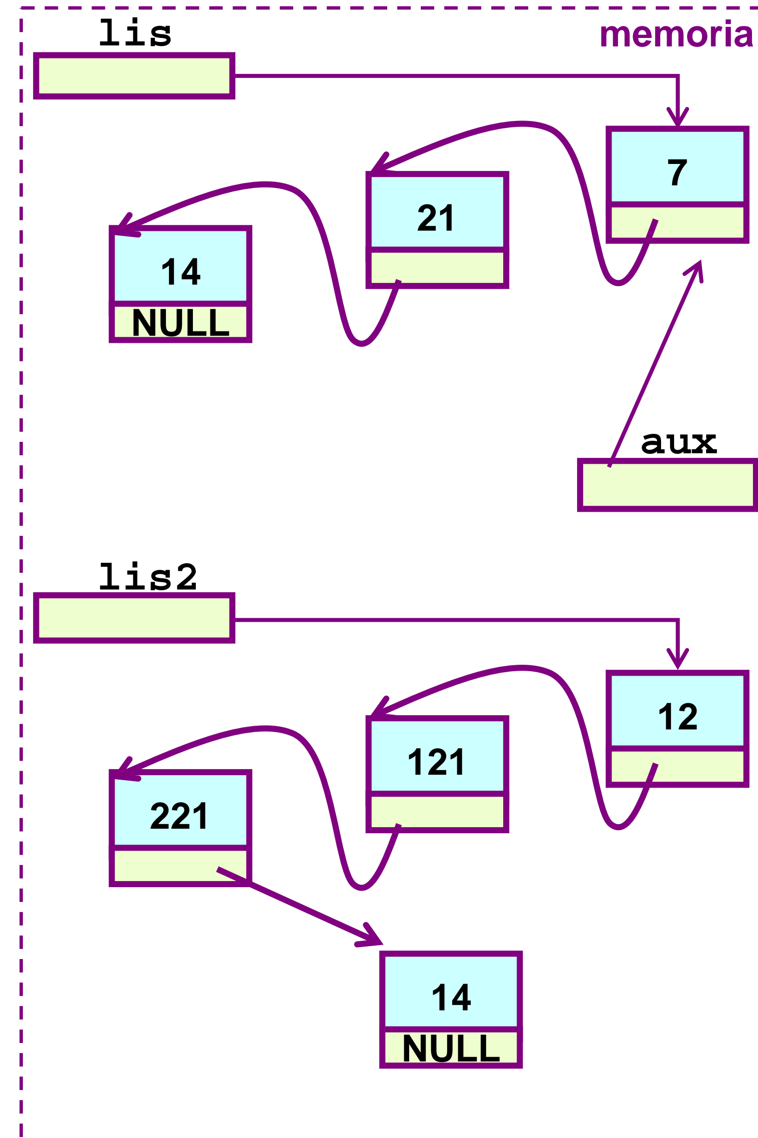
```
...
/* deallocazione prima lista */
```

eliminazione (primo) elem. puntato da lis

```
aux=lis;
lis=lis->next;
free(aux);
```

mentre lis non NULL

```
   while(lis) {
      aux = lis;
      lis = lis->next;
      free(aux);
   }
   /* ora lis==NULL */


   /* deallocazione seconda lista
      mediante chiamata di funzione */
   deallocaLista(&lis2);
               /* side effect previsto */
... FINE
return 0;
}
```



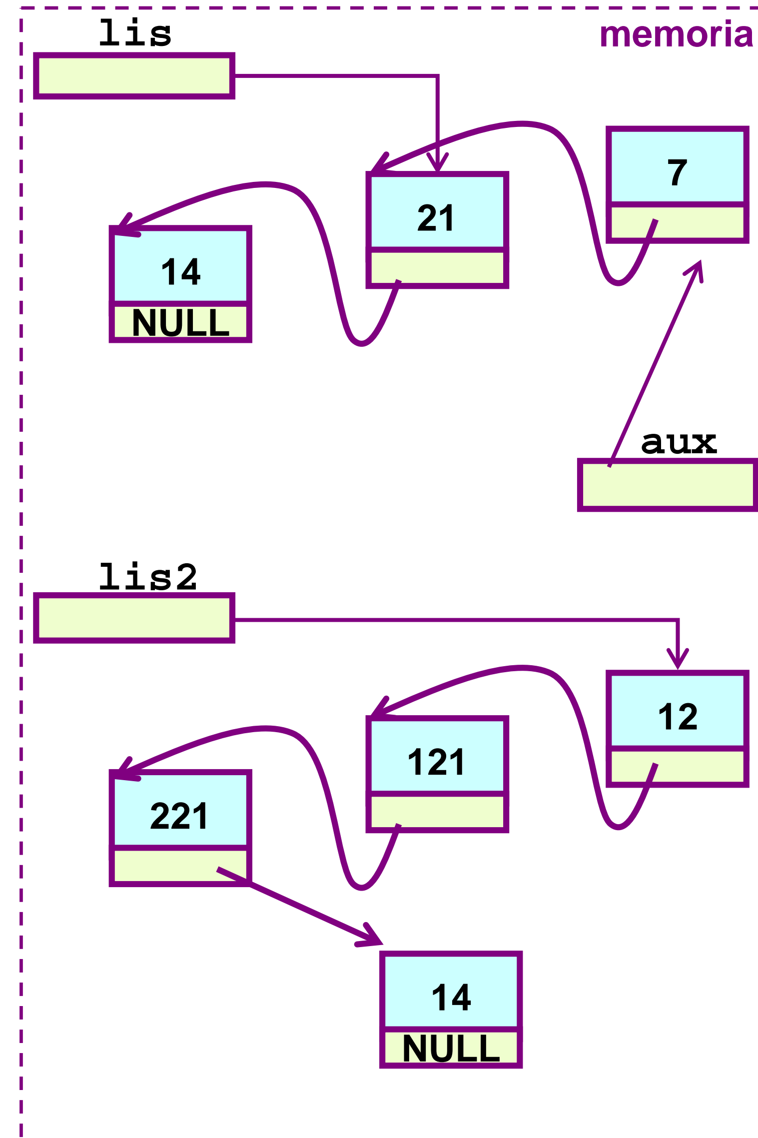
```
...
/* deallocazione prima lista */
```

eliminazione (primo) elem. puntato da lis

```
aux=lis;
lis=lis->next;
free(aux);
```

mentre lis non NULL

```
   while(lis) {
      aux = lis;
      lis = lis->next;
      free(aux);
   }
   /* ora lis==NULL */


   /* deallocazione seconda lista
      mediante chiamata di funzione */
   deallocaLista(&lis2);
                        /* side effect previsto */
... FINE
return 0;
}
```



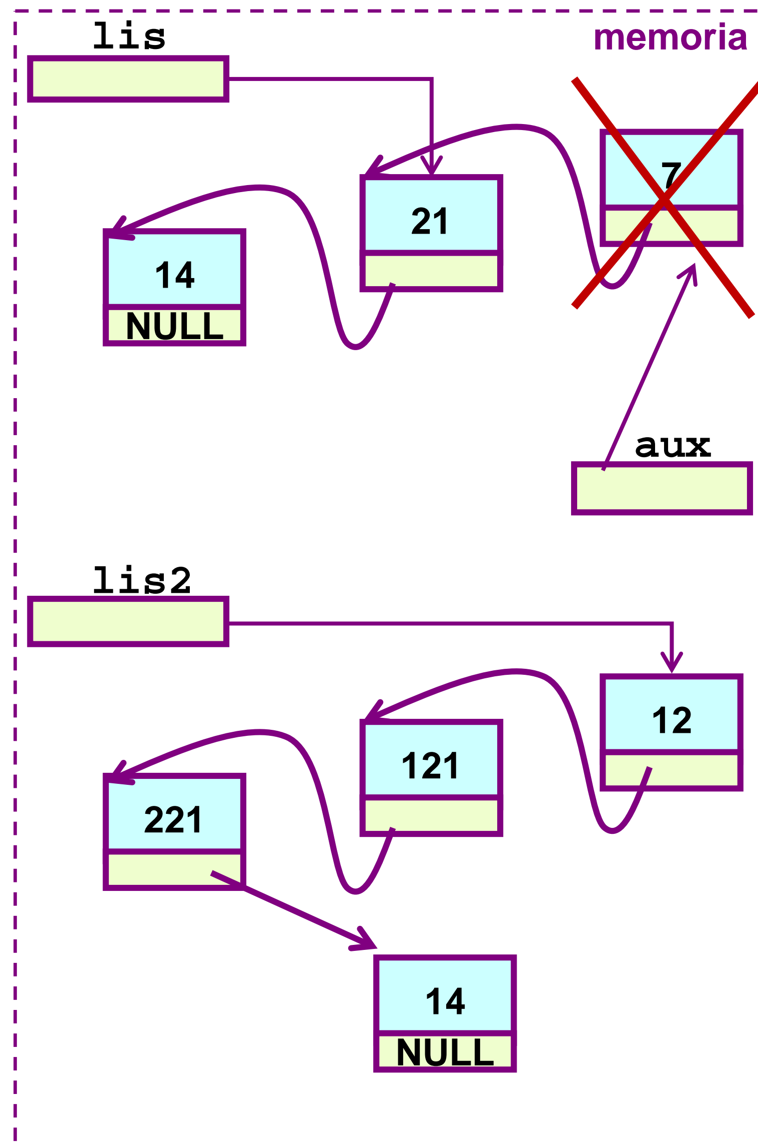
```
...
/* deallocazione prima lista */
```

eliminazione (primo) elem. puntato da lis

```
aux=lis;
lis=lis->next;
free(aux);
```

mentre lis non NULL

```
   while(lis) {
      aux = lis;
      lis = lis->next;
      free(aux);
   }
   /* ora lis==NULL */


   /* deallocazione seconda lista
      mediante chiamata di funzione */
   deallocaLista(&lis2);
                        /* side effect previsto */
... FINE
return 0;
}
```



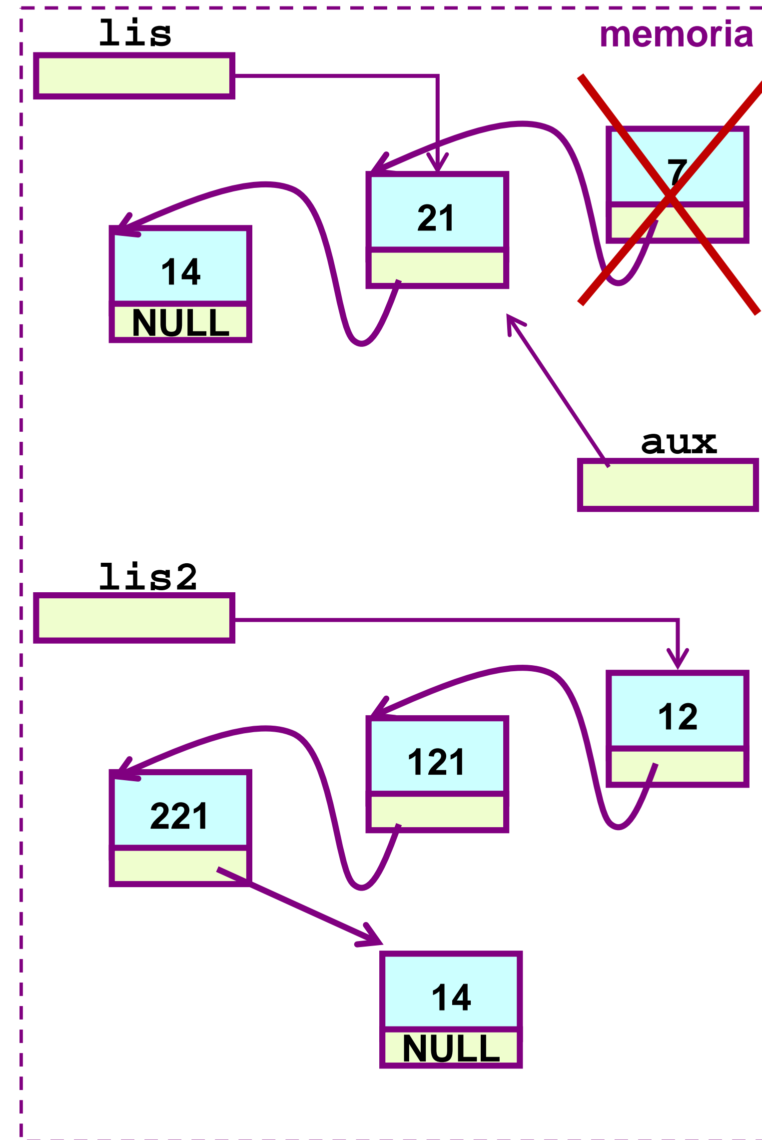
```
...
/* deallocazione prima lista */
```

eliminazione (primo) elem. puntato da lis

```
aux=lis;
lis=lis->next;
free(aux);
```

mentre lis non NULL

```
   while(lis) {
      aux = lis;
      lis = lis->next;
      free(aux);
   }
   /* ora lis==NULL */

   /* deallocazione seconda lista
      mediante chiamata di funzione */
   deallocaLista(&lis2);
                        /* side effect previsto */
... FINE
return 0;
}
```



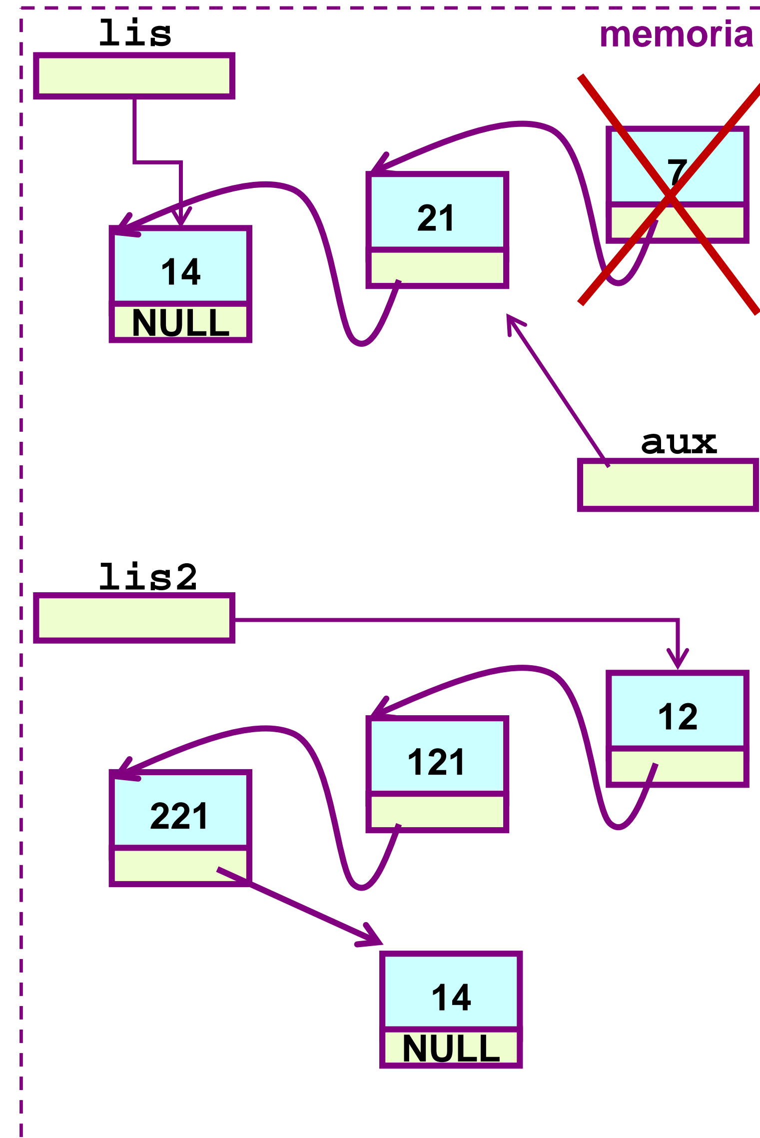
```
...
/* deallocazione prima lista */
```

eliminazione (primo) elem. puntato da lis

```
aux=lis;
lis=lis->next;
free(aux);
```

mentre lis non NULL

```
   while(lis) {
     aux = lis;
     lis = lis->next;
     free(aux);
   }
   /* ora lis==NULL */


   /* deallocazione seconda lista
      mediante chiamata di funzione */
   deallocaLista(&lis2);
                         /* side effect previsto */
... FINE
return 0;
}
```



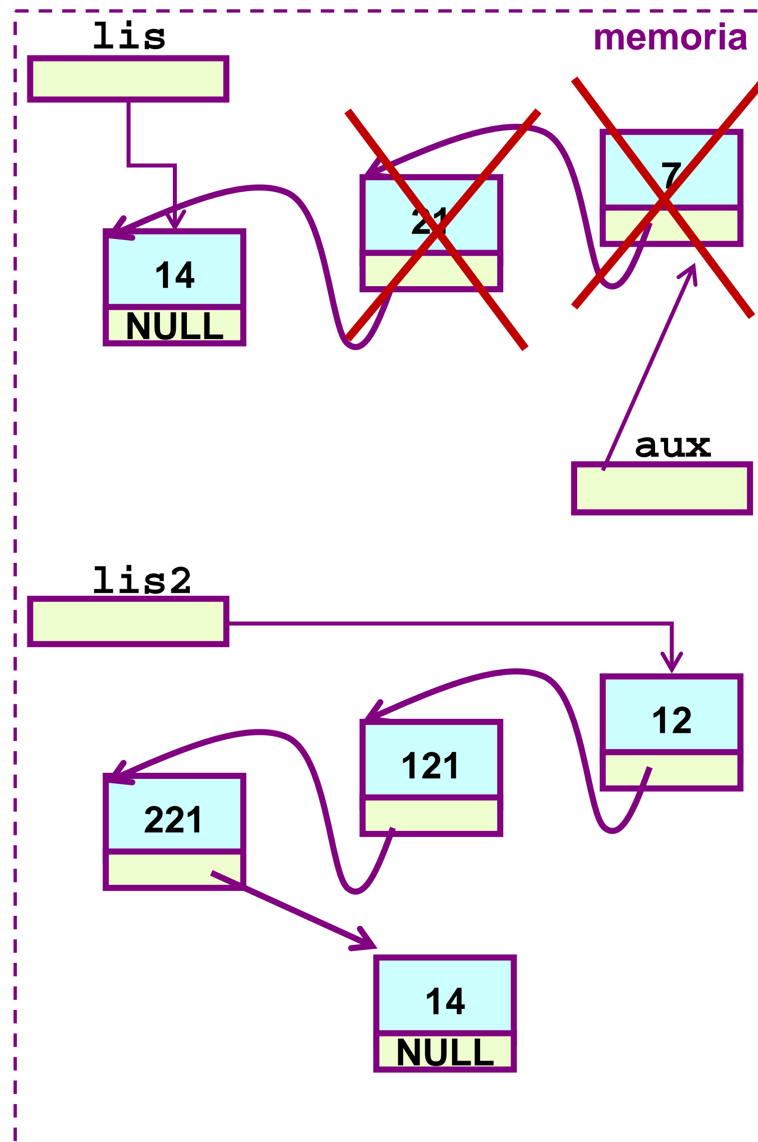
```
...
/* deallocazione prima lista */
```

eliminazione (primo) elem. puntato da lis

```
aux=lis;
lis=lis->next;
free(aux);
```

mentre lis non NULL

```
   while(lis) {
     aux = lis;
     lis = lis->next;
     free(aux);
   }
   /* ora lis==NULL */


   /* deallocazione seconda lista
     mediante chiamata di funzione */
  deallocaLista(&lis2);
                         /* side effect previsto */
... FINE
return 0;
}
```

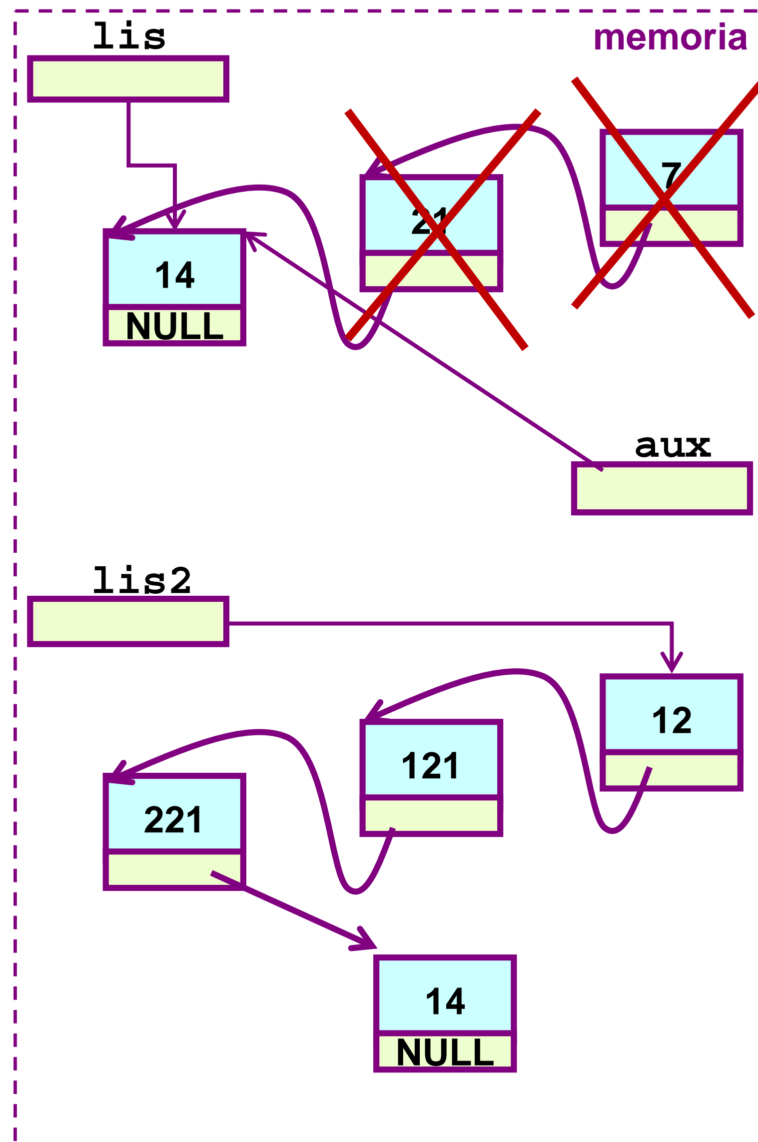


```
...
/* deallocazione prima lista */
```

eliminazione (primo) elem. puntato da lis — **mentre lis non NULL**

```
aux=lis;
lis=lis->next;
free(aux);
```

```
   while(lis) {
      aux = lis;
      lis = lis->next;
      free(aux);
   }
   /* ora lis==NULL */


   /* deallocazione seconda lista
      mediante chiamata di funzione */
   deallocaLista(&lis2);
                          /* side effect previsto */
... FINE
return 0;
}
```



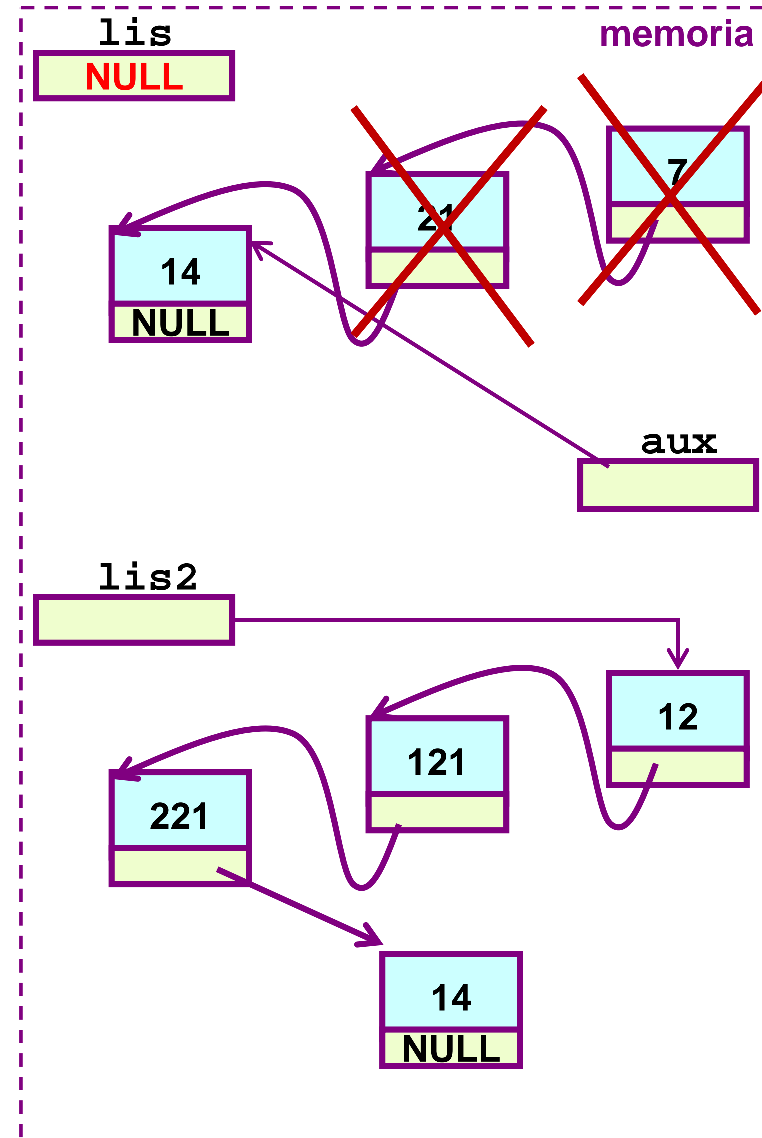
```
...
/* deallocazione prima lista */
```

eliminazione (primo) elem. puntato da lis

```
aux=lis;
lis=lis->next;
free(aux);
```

mentre lis non NULL

```
   while(lis) {
      aux = lis;
      lis = lis->next;
      free(aux);
   }
   /* ora lis==NULL */


   /* deallocazione seconda lista
     mediante chiamata di funzione */
  deallocaLista(&lis2);
                        /* side effect previsto */
... FINE
return 0;
}
```



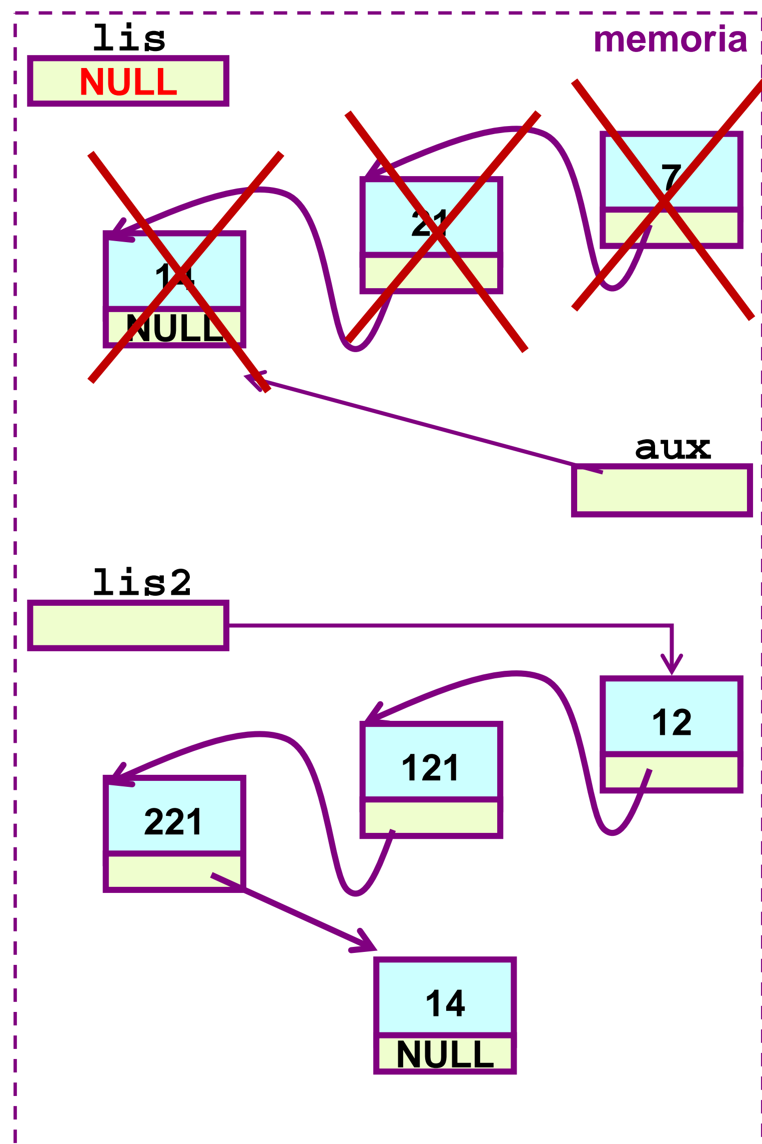
```
...
/* deallocazione prima lista */
```

eliminazione (primo) elem. puntato da lis

```
aux=lis;
lis=lis->next;
free(aux);
```

mentre lis non NULL

```
   while(lis) {
     aux = lis;
     lis = lis->next;
     free(aux);
   }
   /* ora lis==NULL */

    /* deallocazione seconda lista
      mediante chiamata di funzione */
   deallocaLista(&lis2);
                    /* side effect previsto */
... FINE
return 0;
}
```

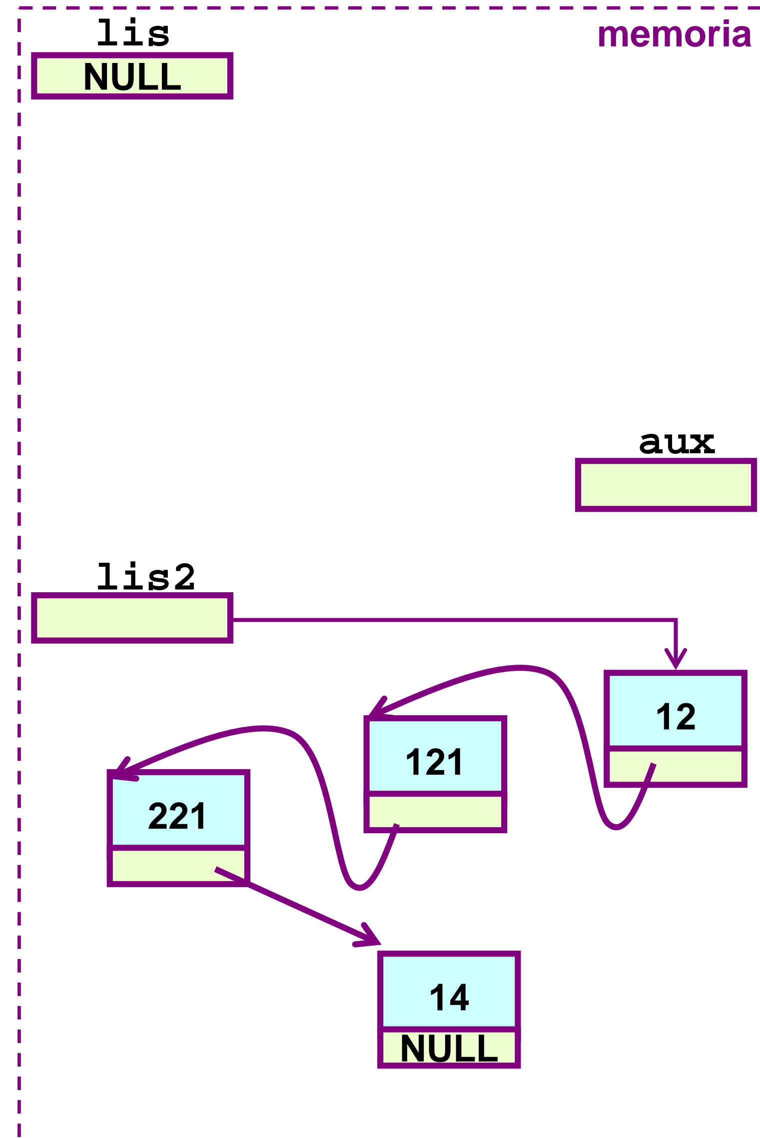


```
...
/* deallocazione prima lista */
```

eliminazione (primo) elem. puntato da lis
`aux=lis;`
`lis=lis->next;`
`free(aux);`

mentre lis non NULL

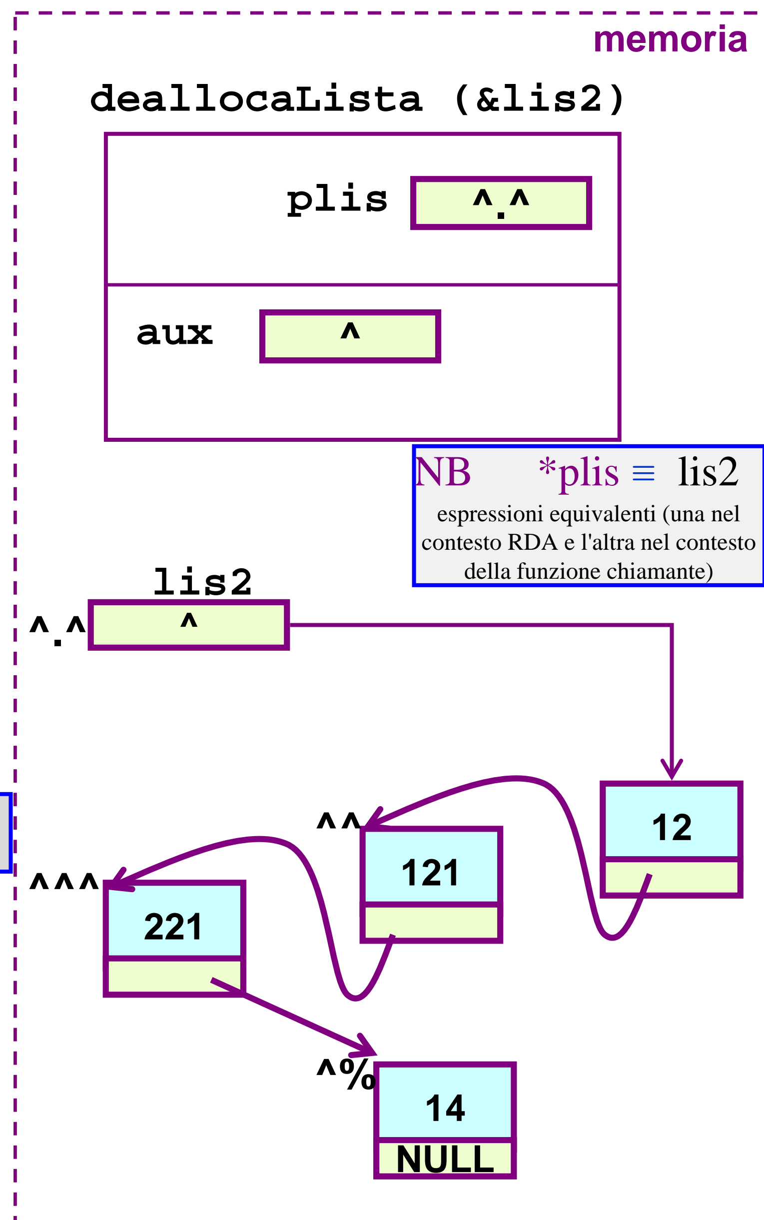
```
   while(lis) {
     aux = lis;
     lis = lis->next;
     free(aux);
   }
   /* ora lis==NULL */

   /* deallocazione seconda lista
      mediante chiamata di funzione */
   deallocaLista(&lis2);
             /* previsto side effect */
... FINE
return 0;
}
```



algoritmo di deallocazione applicato mediante una funzione: la funzione provoca un side effect sul puntatore alla lista, rendendolo NULL alla fine delle deallocazioni di nodo.

```
void deallocaLista(TipoLista *plis) {
     TipoNodo * aux;

 while(*plis) {
      aux = *plis;
      *plis = (*plis)->next;
      free(aux);
   }
return;
}
```

situazione al momento della chiamata …



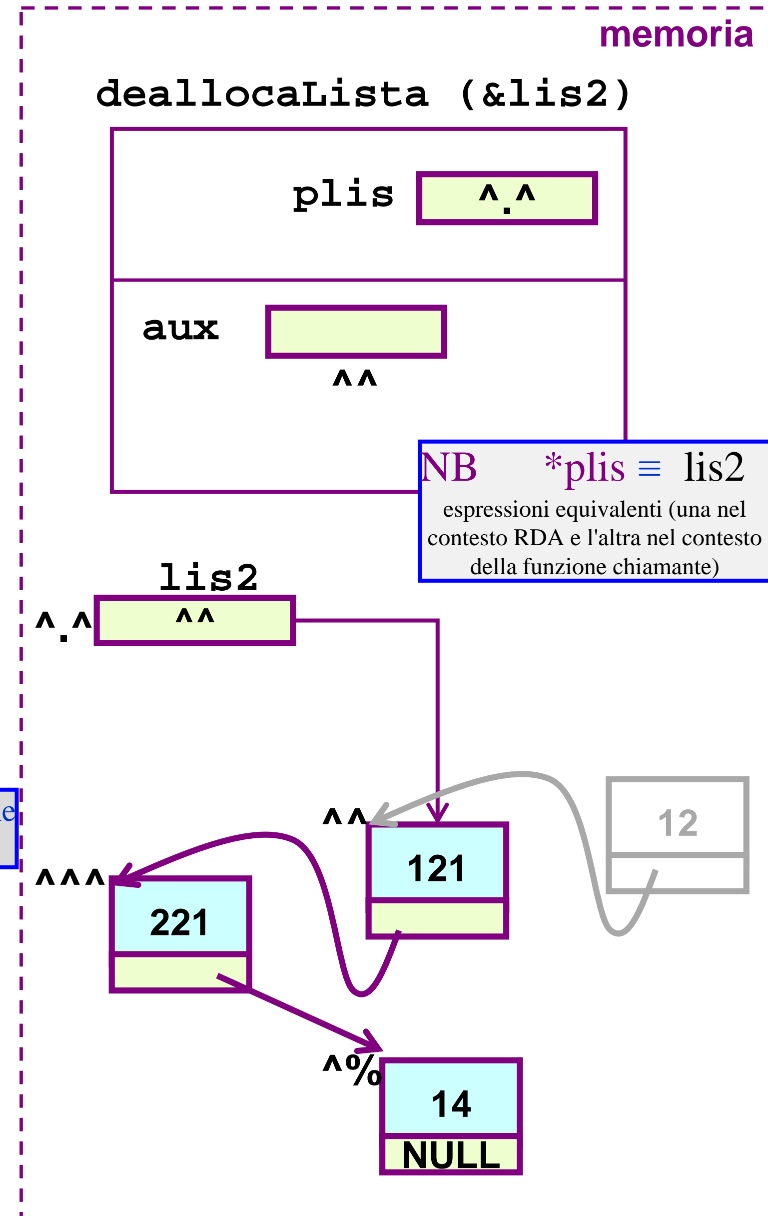
l'implementazione, nel caso sia la funzione che "possiede" il puntatore lis era

```
while(lis) {
   aux = lis;
   lis = lis->next;
   free(aux);      }
```

algoritmo di deallocazione applicato mediante una funzione: la funzione provoca un side effect sul puntatore alla lista, rendendolo NULL alla fine delle deallocazioni di nodo.

```
void deallocaLista(TipoLista *plis) {
     TipoNodo * aux;

 while(*plis) {
      aux = *plis;
      *plis = (*plis)->next;
      free(aux);
    }
return;
}
```

dopo la prima iterazione del ciclo



algoritmo di deallocazione applicato mediante una funzione: la funzione provoca un side effect sul puntatore alla lista, rendendolo NULL alla fine delle deallocazioni di nodo.

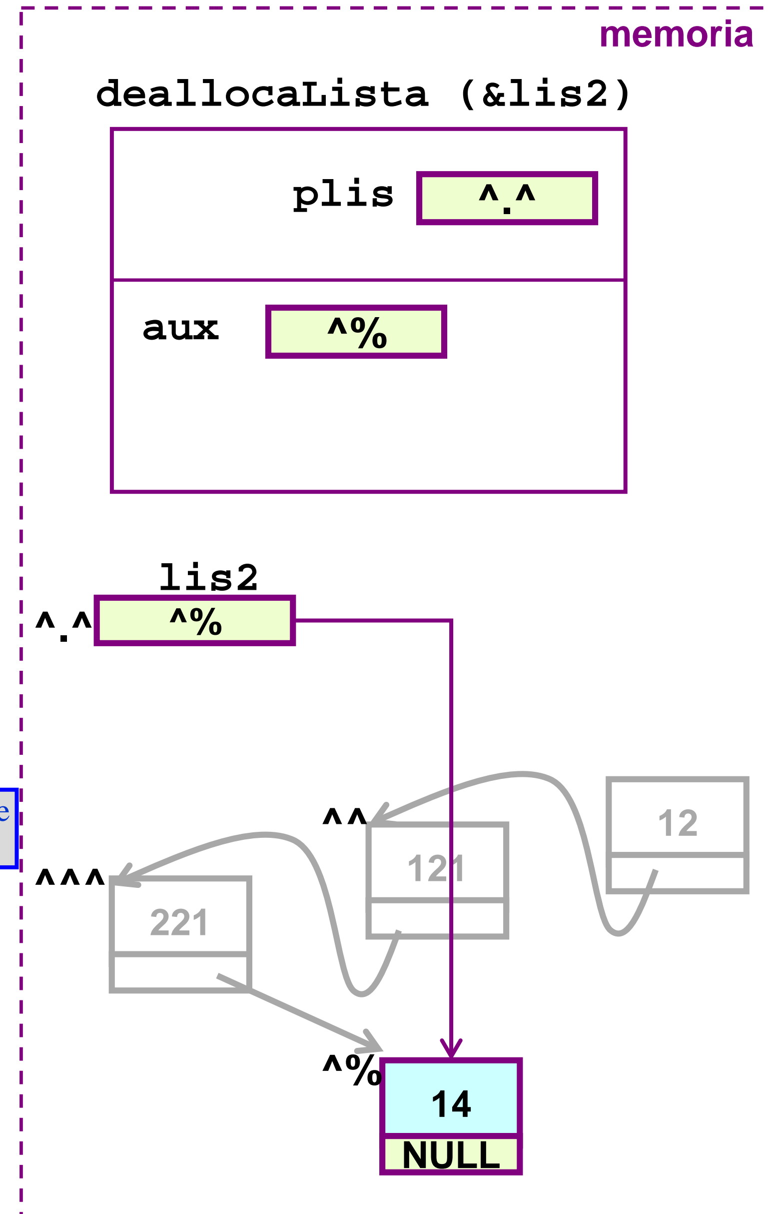
```
void deallocaLista(TipoLista *plis) {
     TipoNodo * aux;

 while(*plis) {
      aux = *plis;
      *plis = (*plis)->next;
      free(aux);
    }
return;
}
```

dopo la seconda iterazione del ciclo

algoritmo di deallocazione applicato mediante una funzione: la funzione provoca un side effect sul puntatore alla lista, rendendolo NULL alla fine delle deallocazioni di nodo.

```
void deallocaLista(TipoLista *plis) {
     TipoNodo * aux;

 while(*plis) {
     aux = *plis;
     *plis = (*plis)->next;
     free(aux);
   }
return;
}
```

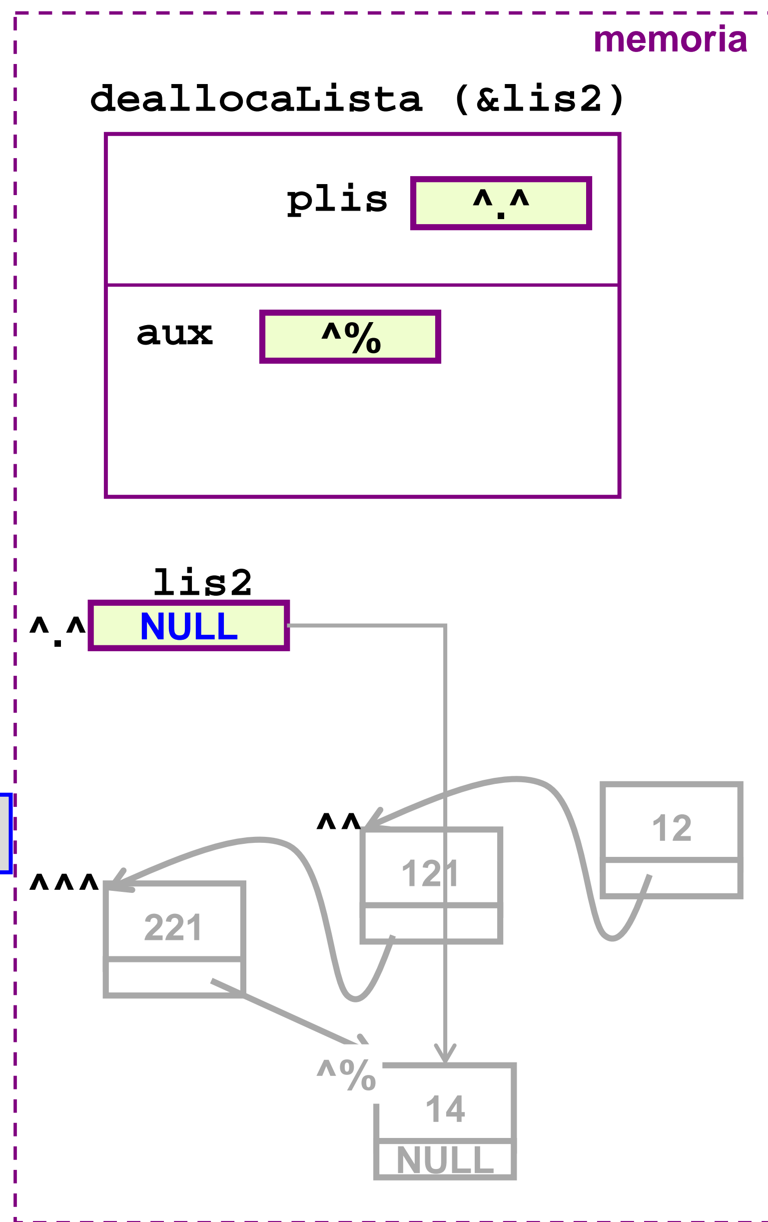
dopo la terza iterazione del ciclo



algoritmo di deallocazione applicato mediante una funzione: la funzione provoca un side effect sul puntatore alla lista, rendendolo NULL alla fine delle deallocazioni di nodo.

```
void deallocaLista(TipoLista *plis) {
     TipoNodo * aux;

  while(*plis) {
      aux = *plis;
      *plis = (*plis)->next;
      free(aux);
    }
return;
}
```

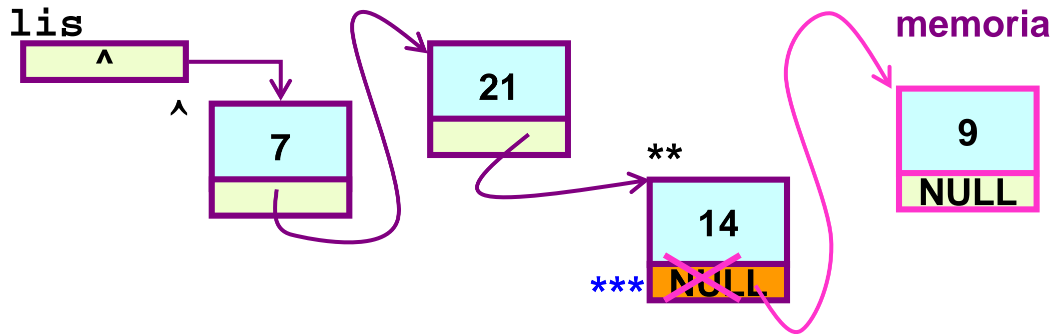
dopo la quarta iterazione del ciclo



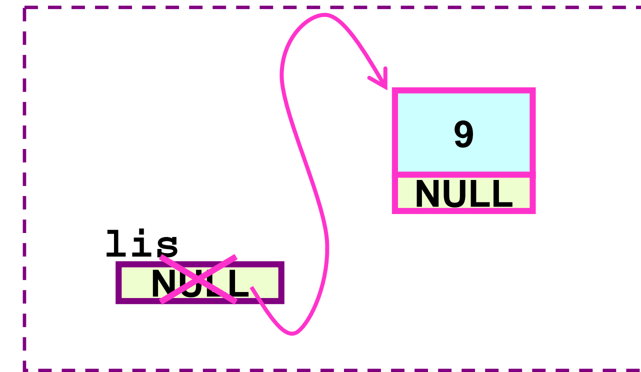
NB in directory pubblica, puo` apparire a volte l'uso del tipo PuntNodoLista,
che qui non usiamo ed è definito come
typedef TipoNodo * PuntNodoLista
esempio: PuntNodoLista aux;

LISTA "struct e puntatori": FUNZIONE di inserimento in coda - 1/6 -

caso generale



caso LISTA VUOTA



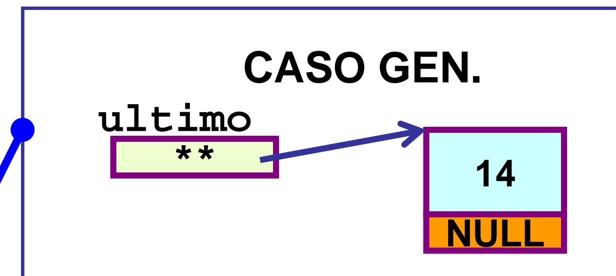
nel caso generico (lista non vuota) l'inserimento in coda non provoca una variazione del puntatore all'inizio della lista; MA NEL CASO DI INSERIMENTO IN LISTA VUOTA, Si`;
quindi la chiamata da effettuare è

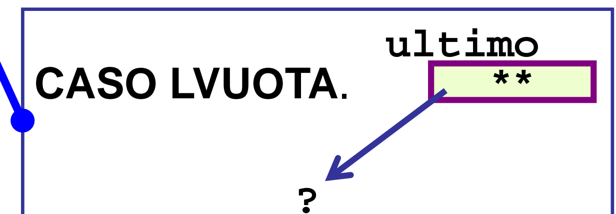
```
insInCoda(&lis, 9)
```

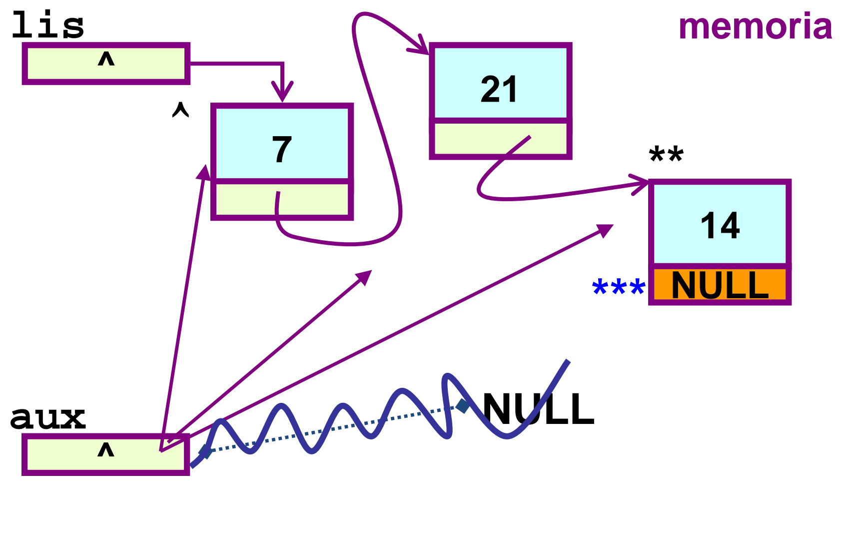
Algoritmo

A) posizionare puntatore `ultimo`, sull'ultimo elemento della lista

B) aggiungere nuovo elemento (con 9) dopo il nodo `*ultimo`







posizionamento di ultimo (punto A)

```
TipoNodo * aux;
aux=lis;
while (aux)
    aux = aux->next;
/* ora chi punta
    sull'ultimo
   nodo? */
```

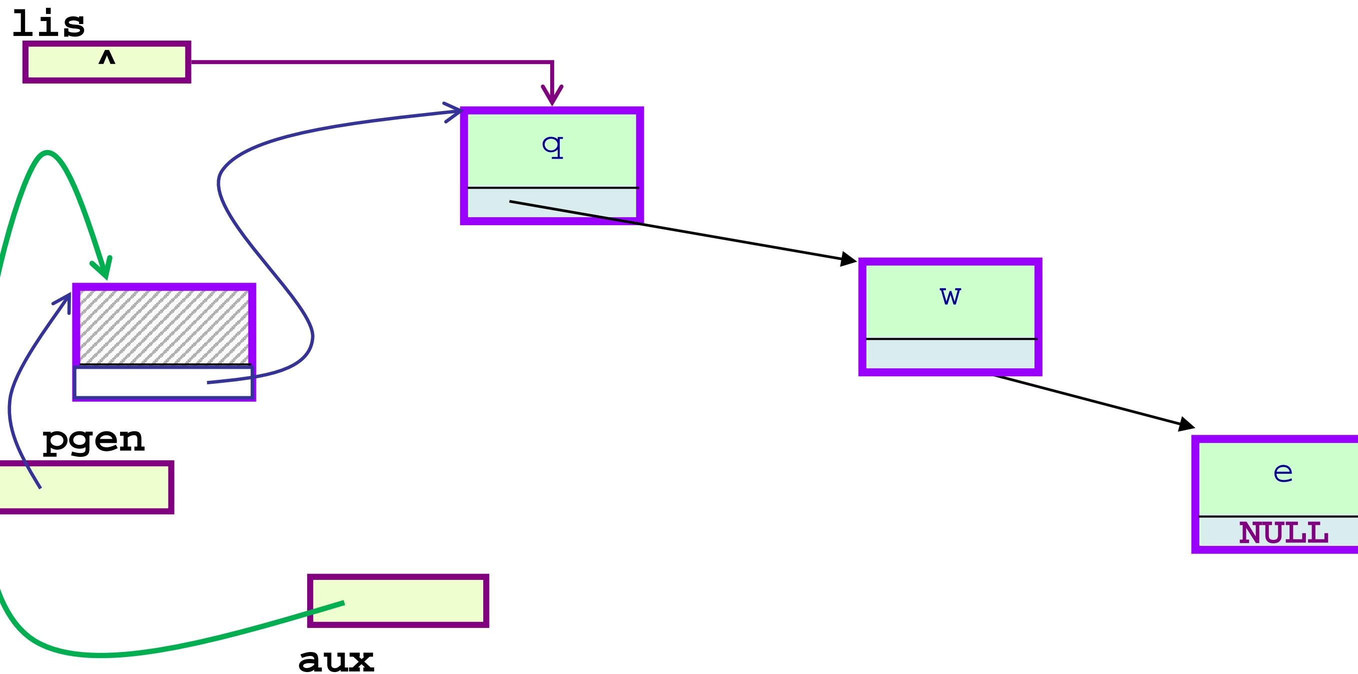
NESSUNO!!!

posizionamento di ultimo (punto A): soluzione giusta - la condizione di avanzamento di aux non è che aux sia non nullo, ma è che aux punti ad un nodo che ha un successore; se il nodo cui punta aux non ha successore, allora lui è l'ultimo (e aux punta all'ultimo)

```
TipoNodo * aux;
aux=lis;
while (aux->next!=NULL)
    aux = aux->next;
/* ora aux punta sull'ultimo nodo */
```

l'unico problema è che nel caso LISTA VUOTA se si inizializza aux con aux=lis, poi non esiste aux->next (il nodo successore) in quanto non esiste nemmeno il nodo puntato da aux ...
Per ovviare a questo problemino si usa la tecnica del RG ...

NB - applicazione tecnica del Record Generatore (RG)




```
TipoNodo * aux;
aux = pgen;
while (aux->next!=NULL)
    aux = aux->next;
/* ora aux punta su un nodo per il quale "->next" esiste … e
                                                contiene NULL */
```

```
void insInCoda(TipoLista * plis, TipoElem el) {
```

(Algoritmo che unifica i casi lista vuota e generico con la tecnica del RG)

0) pgen, ultimo, aux …

1) allocazione record generatore (RG) e suo posizionamento in testa alla lista;

2=A) inizializzazione aux
scansione per portare aux a puntare sull'ultimo
e poi assegnarvi ultimo

3=B) aggiunta dopo *ultimo (il nodo puntato da ultimo)

```
2.1) ultimo->next = malloc(…)
2.2) ultimo = ultimo->next
2.3) ultimo->info = el
```

4) chiusura lista - `ultimo->next = NULL`

5) sistemazione *plis (side effect, assegnandogli `pgen->next`)
6) eliminazione RG

```
void insInCoda(TipoLista * plis, TipoElem el) {
    TipoNodo * aux, *ultimo, *pgen;

 pgen = malloc(sizeof(TipoNodo));    /* 1 */
 if (!pgen) {
    printf (" ... eeeekkk!!!!\n");
    return;
 }

 pgen->next = *plis;                 /* 1 */

 aux = pgen;                         /* 2 */
 while(aux->next)
     aux = aux->next;

 ultimo = aux;      /* 2 - ultimo punta o l'ultimo
                      nodo, se c'è, o il RG */

                 /* 2 inserimento in coda */
                 /* resto algoritmo */
return;
}
```

```
void insInCoda(TipoLista * plis, TipoElem el) {
      TipoNodo * aux *ultimo, *pgen;
 pgen = malloc(sizeof(TipoNodo));                    /* 1 */
 if (!pgen) { printf (" ... eeeekkkk!!!!\n");
              return;
 }
 pgen-> next = *plis;          /* 1 */
 aux = pgen;

 while(aux->next)              /* 2 */
     aux = aux->next;
 ultimo = aux;        /* 2 - ... */

/* 3 - inserimento */
 ultimo->next = malloc(sizeof(TipoNodo));
 if (ultimo->next == NULL)
    printf(" ... problemi in alloc. nuovo nodo");
 else {
     ultimo = ultimo->next;
     ultimo->info = el;
     ultimo->next = NULL;    /* 4 */
 }

 /* 5 sistemazione *plis */
 /* 6 eliminazione RG */
return;
}
```

```
void insInCoda(TipoLista * plis, TipoElem el) {
      TipoNodo * aux *ultimo, *pgen;
 pgen = malloc(sizeof(TipoNodo));           /* 0 */
 if (!pgen) { printf (" ... eeeekkk!!!!\n");
              return;
 }
 pgen-> next = *plis;               /* 1 */
 aux = pgen;

 while(aux->next)                   /* 2 */
     aux = aux->next;
 ultimo = aux;   /* 1 - ultimo punta ... */

 ultimo->next = malloc(sizeof(TipoNodo));  /* 3 */
 if (ultimo->next == NULL)
    printf(" ... problemi in alloc. nuovo nodo */
 else {  ultimo = ultimo->next;
    ultimo->info = el;
    ultimo->next = NULL;  /* 4 */
 }

 *plis = pgen->next;     /* 5 sistemazione *plis */
 free(pgen);             /* 6 eliminazione RG */

return;
}
```