Programmazione ricorsiva

Laboratorio di Programmazione I

Corso di Laurea in Informatica
A.A. 2018/2019

Programmazione ricorsiva

- In quasi tutti i linguaggi di programmazione è ammessa la possibilità di definire funzioni ricorsive: durante l'esecuzione di una funzione `F` è possibile chiamare la funzione `F` stessa.
- Ciò può avvenire
  - direttamente: il corpo di `F` contiene una chiamata a `F` stessa.
  - indirettamente (ricorsione annidata):`F` contiene una chiamata a `G` che a sua volta contiene una chiamata a `F`.

Programmazione ricorsiva

- La programmazione ricorsiva si basa sull'idea che per molti problemi la soluzione per un caso generico può essere ricavata sulla base della soluzione di un altro caso, generalmente più semplice, dello stesso problema.
- La soluzione di un problema viene individuata supponendo di saperlo risolvere su casi più semplici.
- Bisogna poi essere in grado di risolvere direttamente il problema sui casi più semplici di qualunque altro.

Programmazione ricorsiva

- Le funzioni ricorsive sono convenienti per implementare funzioni matematiche definite in modo induttivo.
- Funzione fattoriale - definizione iterativa:
  $fatt(n) = n \cdot (n-1) \cdot (n-2) \cdots 2 \cdot 1$
- Funzione fattoriale - definizione induttiva:

$$fatt(n) = \begin{cases} 1 & \text{se } n = 0 \quad \text{(caso base)} \\ n \cdot fatt(n-1) & \text{se } n > 0 \quad \text{(caso induttivo)} \end{cases}$$

- È essenziale il fatto che, applicando ripetutamente il caso induttivo, ci riconduciamo prima o poi al caso base.

$$\begin{aligned} fatt(3) = \quad & 3 \cdot \overline{fatt(2)} = \\ & 3 \cdot (2 \cdot \overline{fatt(1)}) = \\ & 3 \cdot (2 \cdot (1 \cdot \overline{fatt(0)})) = \\ & 3 \cdot (2 \cdot (1 \cdot 1)) = \\ & 3 \cdot (2 \cdot 1) = \\ & 3 \cdot 2 = \\ & 6 \end{aligned}$$

Fattoriale

- Versione iterativa:

```
int fatt(int n) {
        int i,ris;
        ris=1;
        for (i=1;i<=n;i++)
          ris=ris*i;
        return ris;
    }
```

- Versione ricorsiva:

```
int fattric(int n) {
      if (n == 0)
        return 1;
      else
        return n * fattric(n-1);
     }
```

Fattoriale

- Uso della funzione ricorsiva:

```
int fattric (int);

int main(){
    int x, f;
    scanf("%d", &x);
    f = fattric(x);
    printf("Fattoriale di %d:  %d\n", x, f);
    return 0;
}

int fattric(int n) {
  int ris;
  if (n == 0)
    ris = 1;
  else
    ris = n * fattric(n-1);
  return ris;
}
```

Evoluzione della pila (supponendo x=3)

x	3
f	?

Evoluzione della pila (supponendo x=3)

x	3
f	?

n	3
ris	?

x	3
f	?

Evoluzione della pila (supponendo x=3)

x	3
f	?

n	3
ris	?

x	3
f	?

n	2
ris	?

n	3
ris	?

x	3
f	?

Evoluzione della pila (supponendo x=3)

x	3
f	?

n	3
ris	?

x	3
f	?

n	2
ris	?

n	3
ris	?

x	3
f	?

n	1
ris	?

n	2
ris	?

n	3
ris	?

x	3
f	?

Evoluzione della pila (supponendo x=3)

x	3
f	?

n	3
ris	?
x	3
f	?

n	2
ris	?
n	3
ris	?
x	3
f	?

n	1
ris	?
n	2
ris	?
n	3
ris	?
x	3
f	?

n	0
ris	?
n	1
ris	?
n	2
ris	?
n	3
ris	?
x	3
f	?

Evoluzione della pila (supponendo x=3)

x	3
f	?

n	3
ris	?

x	3
f	?

n	2
ris	?

n	3
ris	?

x	3
f	?

n	1
ris	?

n	2
ris	?

n	3
ris	?

x	3
f	?

n	0
ris	?

n	1
ris	?

n	2
ris	?

n	3
ris	?

x	3
f	?

n	0
ris	1

n	1
ris	?

n	2
ris	?

n	3
ris	?

x	3
f	?

Evoluzione della pila (supponendo x=3)

x	3
f	?

n	3
ris	?
x	3
f	?

n	2
ris	?
n	3
ris	?
x	3
f	?

n	1
ris	?
n	2
ris	?
n	3
ris	?
x	3
f	?

n	0
ris	?
n	1
ris	?
n	2
ris	?
n	3
ris	?
x	3
f	?

n	0
ris	1
n	1
ris	?
n	2
ris	?
n	3
ris	?
x	3
f	?

n	1
ris	1
n	2
ris	?
n	3
ris	?
x	3
f	?

Evoluzione della pila (supponendo x=3)

x	3
f	?

n	3
ris	?
x	3
f	?

n	2
ris	?
n	3
ris	?
x	3
f	?

n	1
ris	?
n	2
ris	?
n	3
ris	?
x	3
f	?

n	0
ris	?
n	1
ris	?
n	2
ris	?
n	3
ris	?
x	3
f	?

n	0
ris	1
n	1
ris	?
n	2
ris	?
n	3
ris	?
x	3
f	?

n	1
ris	1
n	2
ris	?
n	3
ris	?
x	3
f	?

n	2
ris	2
n	3
ris	?
x	3
f	?

Evoluzione della pila (supponendo x=3)

x	3
f	?

n	3
ris	?
x	3
f	?

n	2
ris	?
n	3
ris	?
x	3
f	?

n	1
ris	?
n	2
ris	?
n	3
ris	?
x	3
f	?

n	0
ris	?
n	1
ris	?
n	2
ris	?
n	3
ris	?
x	3
f	?

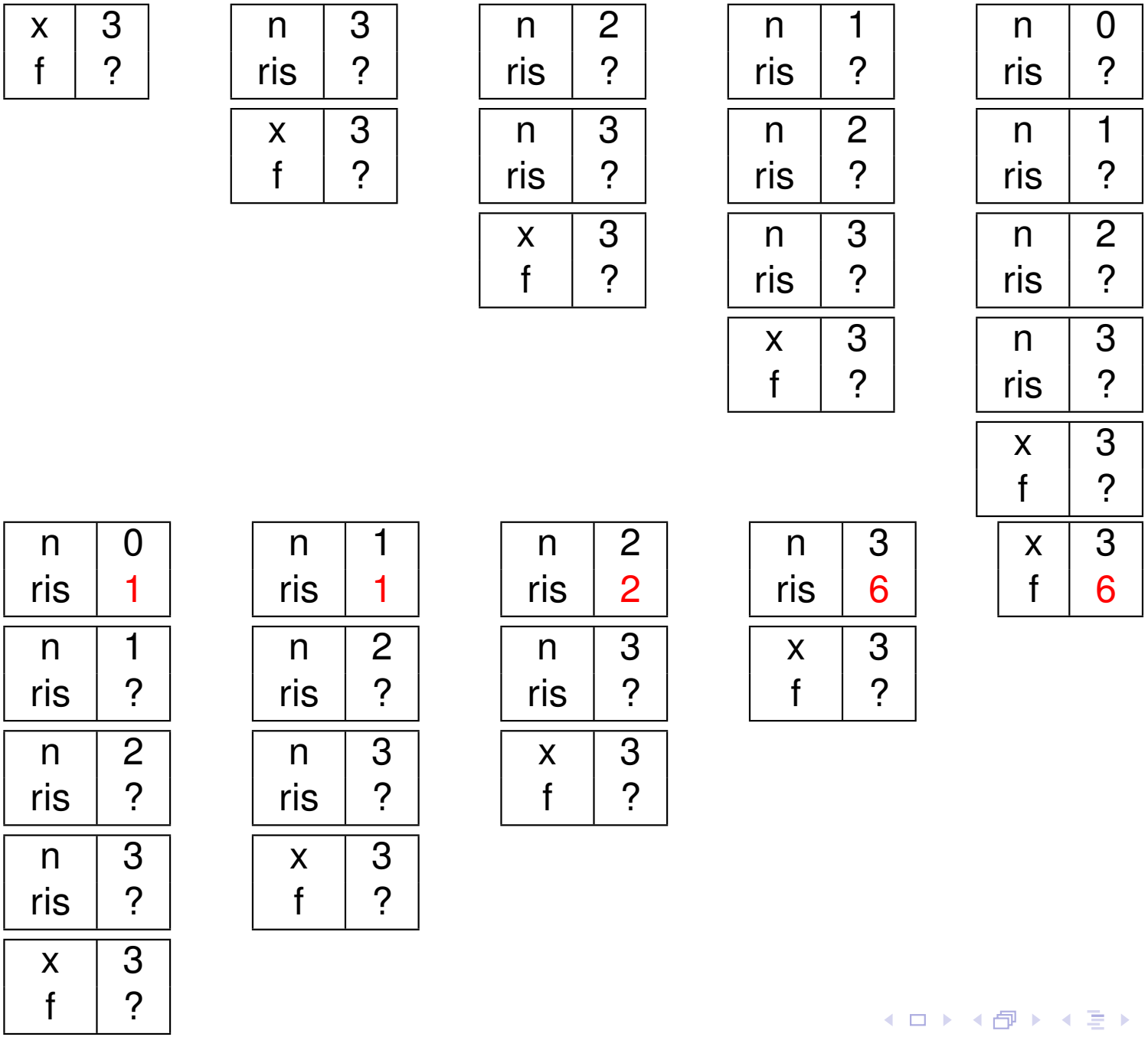
n	0
ris	1
n	1
ris	?
n	2
ris	?
n	3
ris	?
x	3
f	?

n	1
ris	1
n	2
ris	?
n	3
ris	?
x	3
f	?

n	2
ris	2
n	3
ris	?
x	3
f	?

n	3
ris	6
x	3
f	?

Evoluzione della pila (supponendo x=3)



Esempio: sequenza invertita

Leggere una sequenza di caratteri terminata da `'\n'` e stamparla invertita. Ad esempio: `casa` $\rightarrow$ `asac`

- Problema: prima di poter iniziare a stampare dobbiamo aver letto e memorizzato tutta la sequenza:
  - usando una struttura dati opportuna ma dinamica (liste, le vedremo più avanti)
  - usando un procedimento ricorsivo.
    - leggiamo un carattere della sequenza, `c1`, leggiamo e stampiamo ricorsivamente il resto della sequenza `c2...cn` e infine stampiamo `c1`;
    - il caso base è rappresentato dalla lettura del carattere di fine sequenza.

Esempio: sequenza invertita

```
void invertInputRic(){
    char ch;
    ch = getchar();
    if (ch != '\n'){
        invertInputRic();
        putchar(ch);
    }
    else
        printf("Sequenza invertita: ");
}

int main(){
  printf("Immetti una sequenza di caratteri\n");
  invertInputRic();
  printf("\n");
  return 0;
}
```

Esempio: sequenza invertita

- Evoluzione della pila:

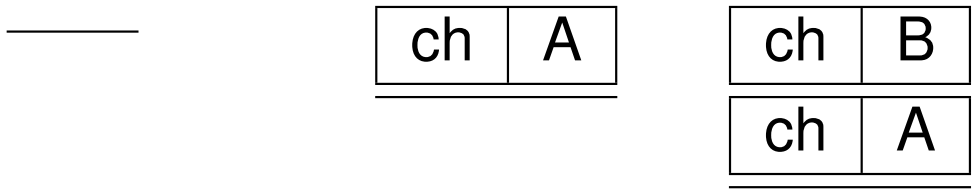
  _____

Esempio: sequenza invertita

- Evoluzione della pila:

Esempio: sequenza invertita

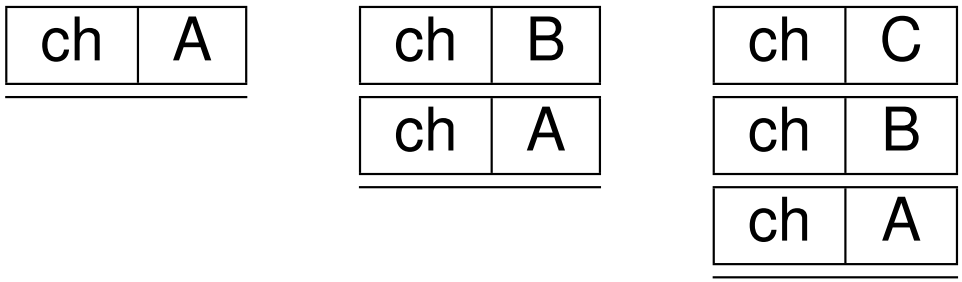
- Evoluzione della pila:



Esempio: sequenza invertita

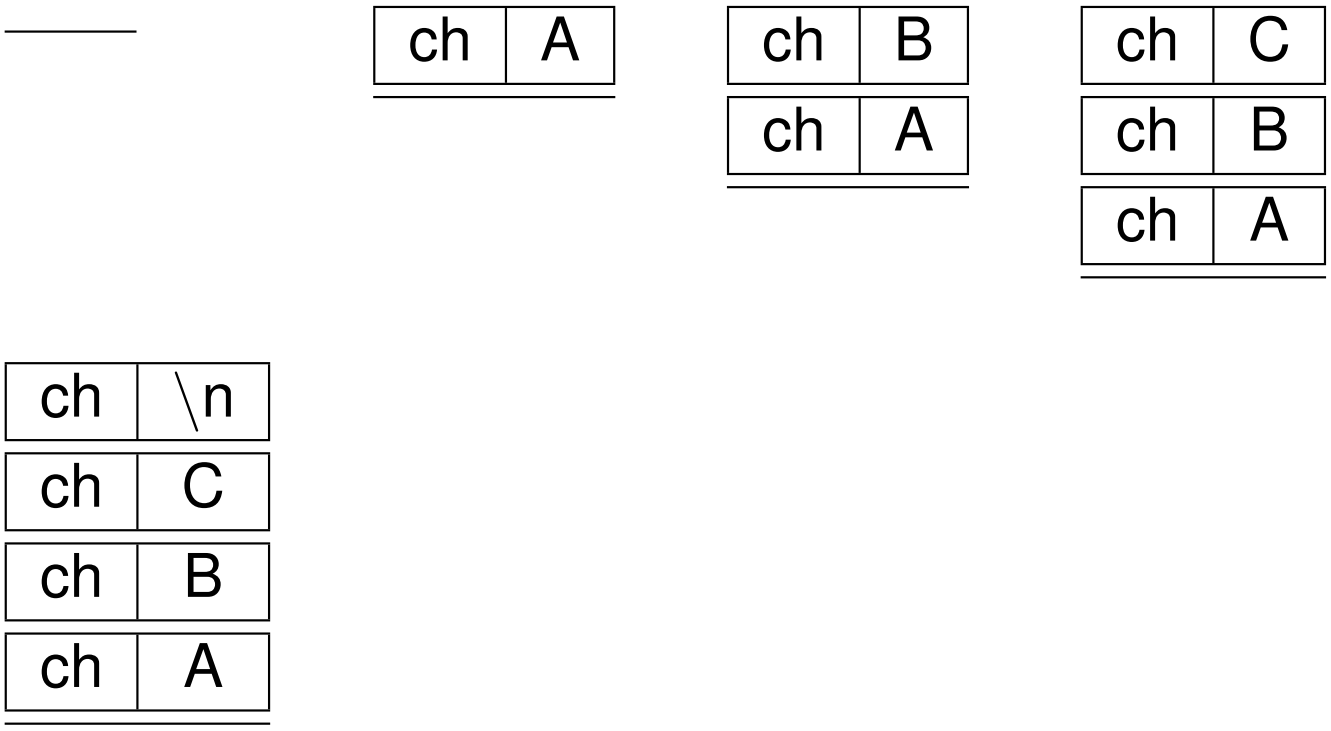
- Evoluzione della pila:

———



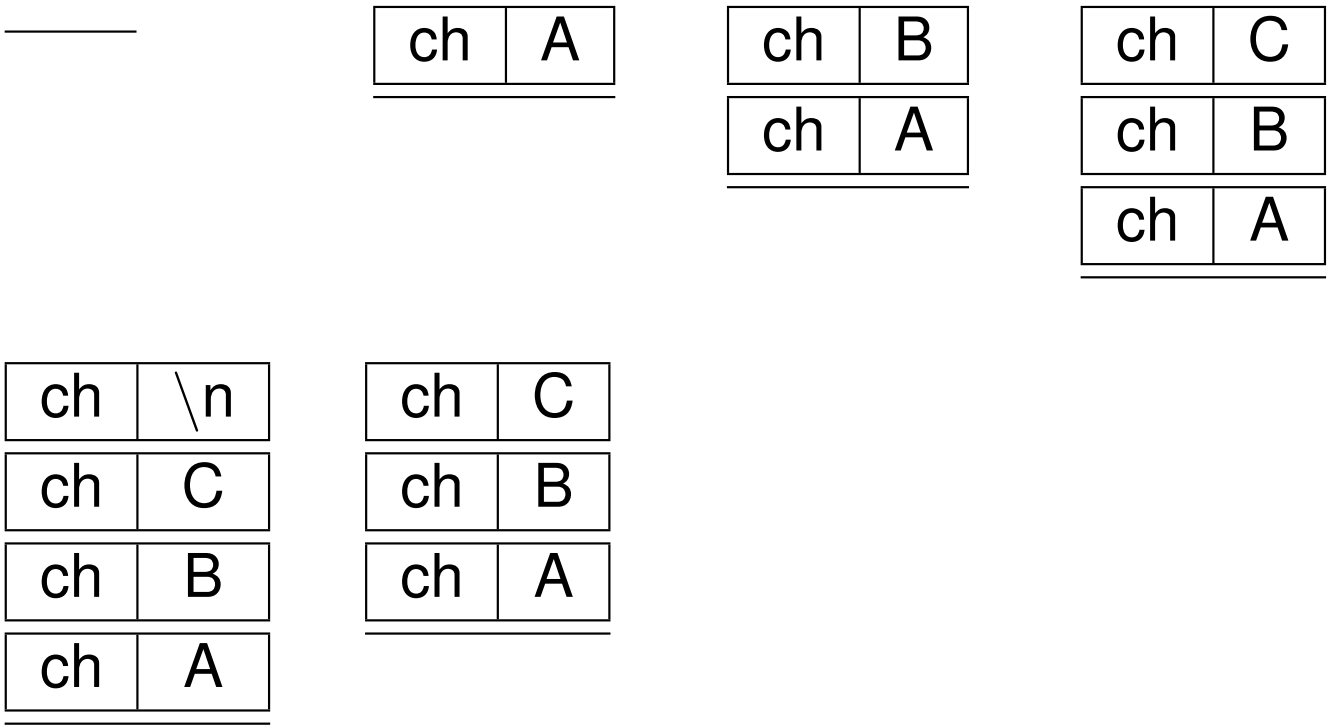
Esempio: sequenza invertita

- Evoluzione della pila:



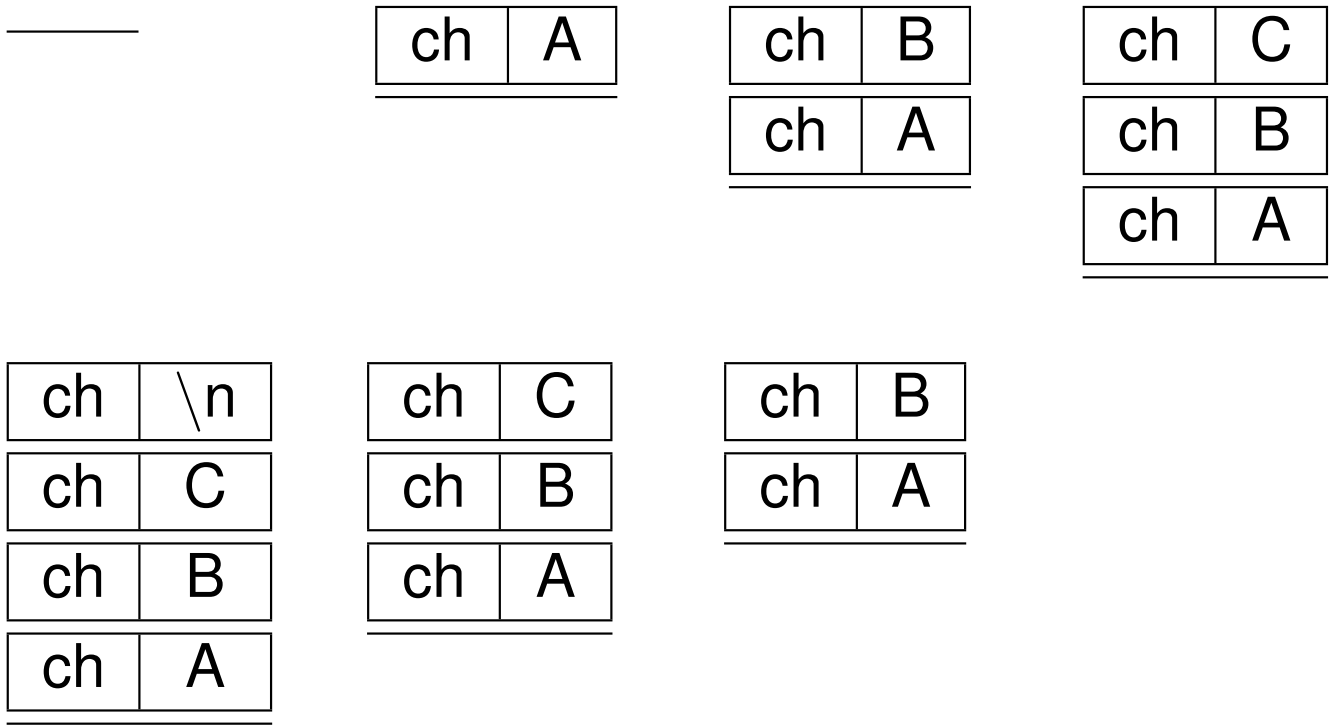
Esempio: sequenza invertita

- Evoluzione della pila:



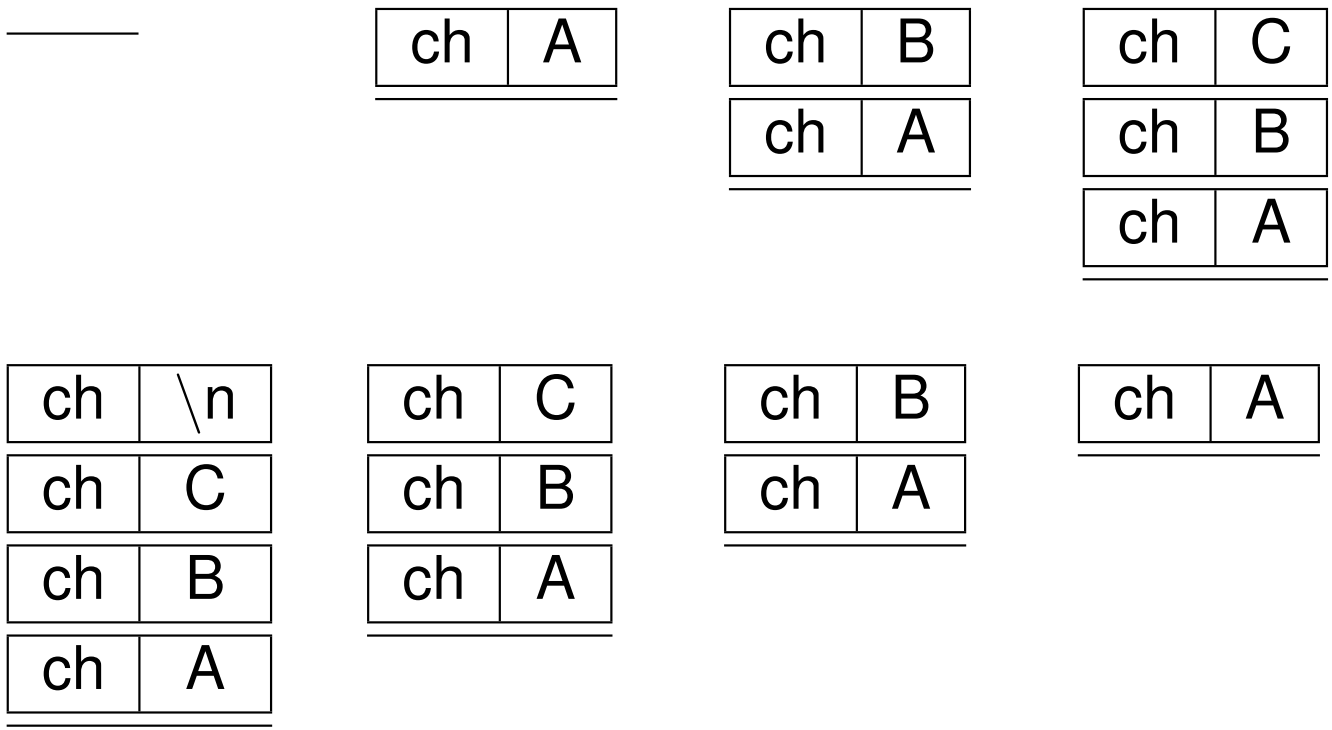
Esempio: sequenza invertita

- Evoluzione della pila:



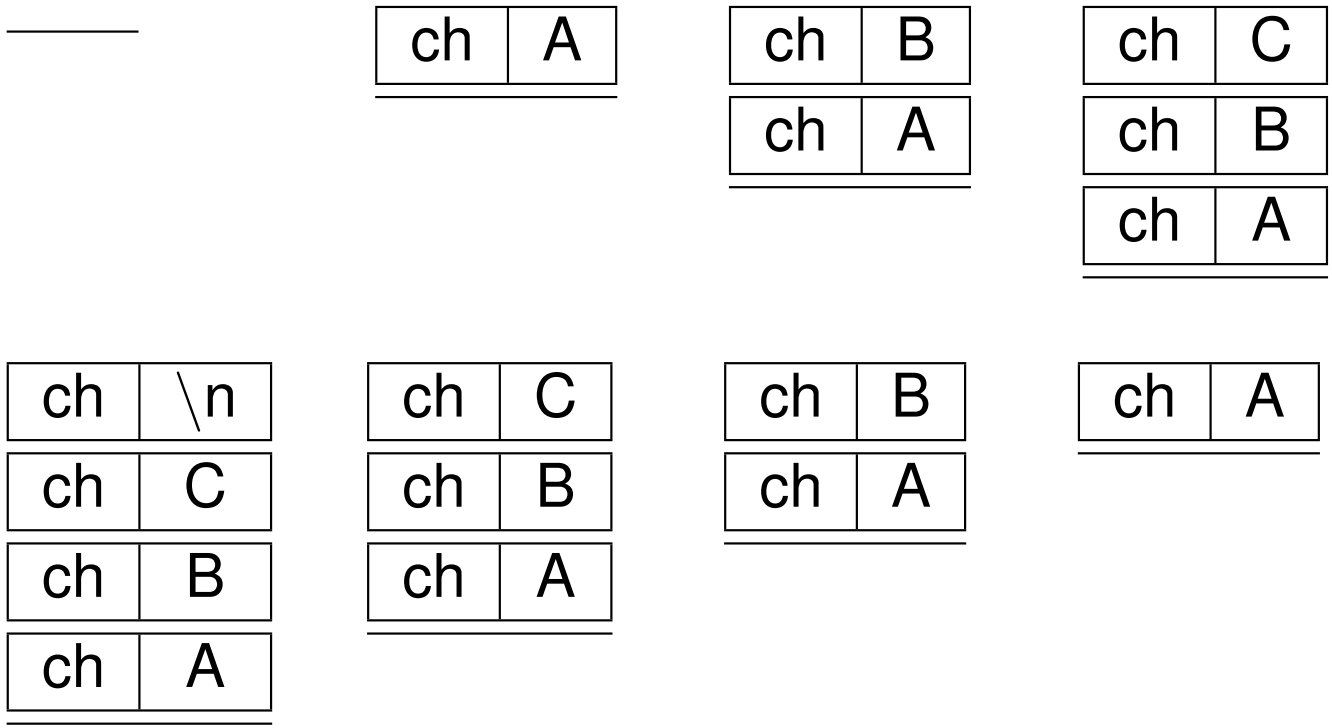
Esempio: sequenza invertita

- Evoluzione della pila:



Esempio: sequenza invertita

- Evoluzione della pila:



- L'output prodotto è `Sequenza invertita:  CBA`

Array e ricorsione

- Calcolare la somma degli elementi di un array:

```
int sumVet(int *v, int dim){
    if (dim==0)
        return 0;
    else
        return v[0] + sumVet(v+1,dim-1);
}
```

Array e ricorsione

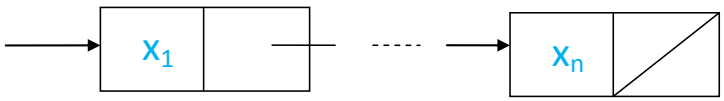
- Calcolare il numero di occorrenze dell'elemento `x` nell' array `v`:

```
int occorrenze (int *v, int dim, int x){
    int occ;
    if (dim==0)
        occ= 0;
    else
        if (v[0]!=x)
            occ = occorrenze(v+1,dim-1,x);
        else
            occ = 1+occorrenze(v+1,dim-1,x);
    return occ;
}
```

Visione ricorsiva delle liste

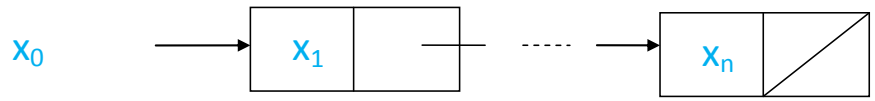
- Una lista di elementi è una struttura dati ricorsiva per sua natura

Visione ricorsiva delle liste



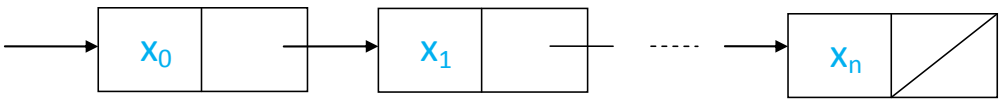
- Una lista di elementi è una struttura dati ricorsiva per sua natura
  1. data una lista **L** di elementi $\mathbf{x_1, \ldots, x_n}$

Visione ricorsiva delle liste



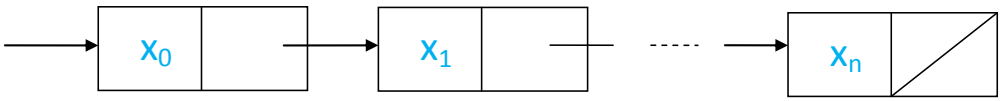
- Una lista di elementi è una struttura dati ricorsiva per sua natura
  1. data una lista **L** di elementi $\mathbf{x_1, \ldots, x_n}$
  2. dato un ulteriore elemento $\mathbf{x_0}$

Visione ricorsiva delle liste



- Una lista di elementi è una struttura dati ricorsiva per sua natura
  1. data una lista **L** di elementi $\mathbf{x_1, \ldots, x_n}$
  2. dato un ulteriore elemento $\mathbf{x_0}$
  3. anche la concatenazione di $\mathbf{x_0}$ e **L** è una lista

Visione ricorsiva delle liste



- Una lista di elementi è una struttura dati ricorsiva per sua natura
  1. data una lista **L** di elementi $\mathbf{x_1, \ldots, x_n}$
  2. dato un ulteriore elemento $\mathbf{x_0}$
  3. anche la concatenazione di $\mathbf{x_0}$ e **L** è una lista
- Si noti che in **1.**, **L** può anche essere la lista vuota

Funzioni ricorsive su liste

- Consistono, in genere, nel:
  - individuare il caso base e risolverlo direttamente: di solito lista vuota.
  - costruire la soluzione del caso ricorsivo: di solito operando sul primo elemento della lista e componendo la soluzione con il risultato dell'invocazione della procedura/funzione sul resto della lista.
- Ad esempio per la stampa di una lista:

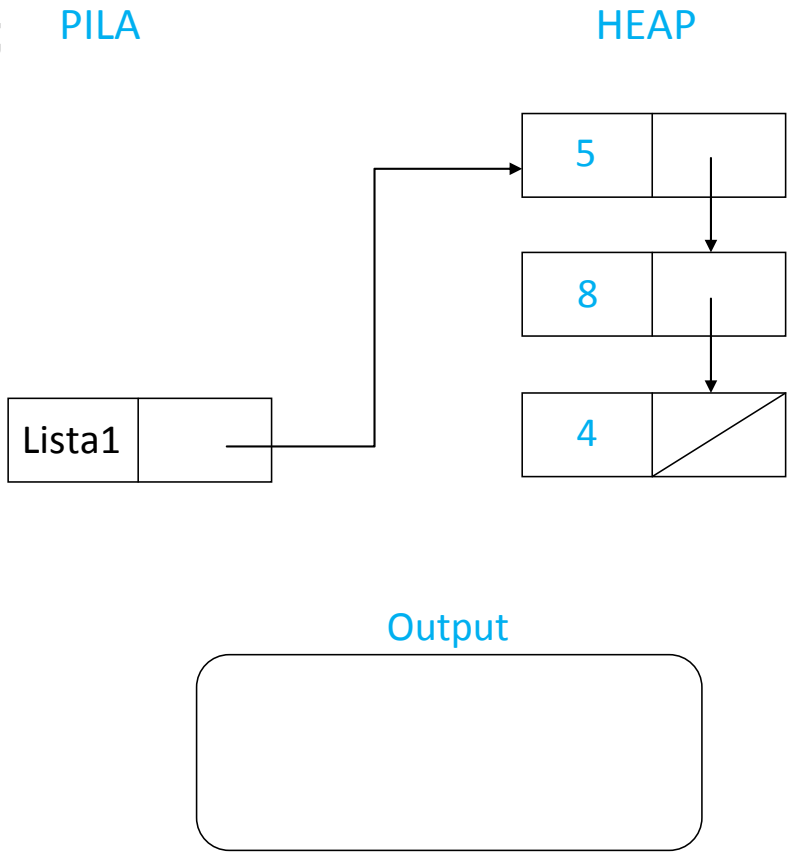
```
void StampaListaRic(ElementoDiLista* lis)
{
    if (lis != NULL){
        printf("%d ", lis->info);
        StampaListaRic(lis->next);
    }
    else
        printf("//");
}
```

Stampa ricorsiva di una lista

```
void StampaListaRic(ElementoDiLista* lis)
{
    if (lis != NULL){
        printf("%d ", lis->info);
        StampaListaRic(lis->next);
    }
    else
        printf("//");
}

int main(){
    ElementoDiLista* Lista1;
    ...
    /* costruzione lista
      5 --> 8 --> 4 */
    ...
    StampaListaRic(Lista1);
    ...
}
```

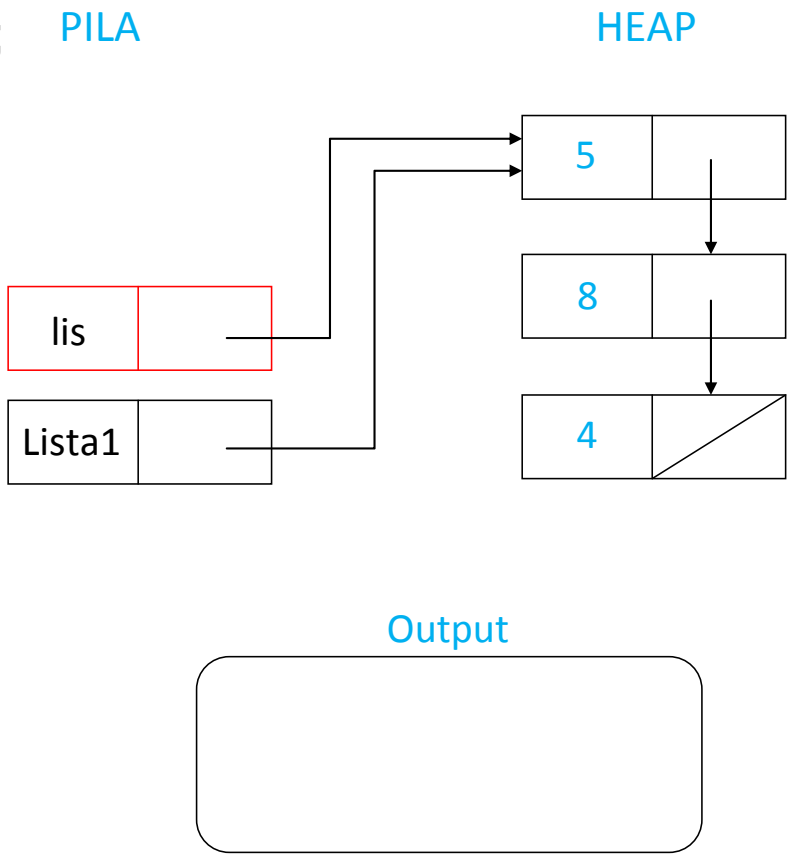
Stampa ricorsiva di una lista

```
void StampaListaRic(ElementoDiLista* lis)
{
    if (lis != NULL){
        printf("%d ", lis->info);
        StampaListaRic(lis->next);
    }
    else
        printf("//");
}

int main(){
    ElementoDiLista* Lista1;
    ...
    /* costruzione lista
      5 --> 8 --> 4 */
    ...
    StampaListaRic(Lista1);
    ...
}
```



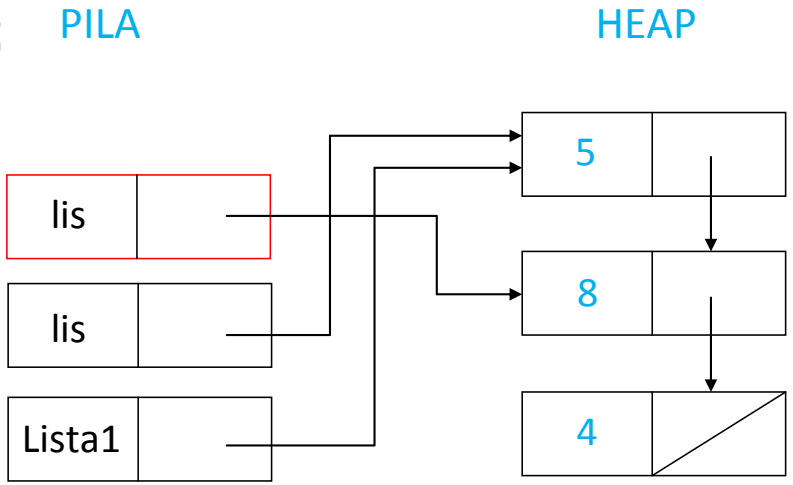
Stampa ricorsiva di una lista

```
void StampaListaRic(ElementoDiLista* lis)
{
    if (lis != NULL){
        printf("%d ", lis->info);
        StampaListaRic(lis->next);
    }
    else
        printf("//");
}

int main(){
    ElementoDiLista* Lista1;
    ...
    /* costruzione lista
       5 --> 8 --> 4 */
    ...
    StampaListaRic(Lista1);
    ...
}
```



Stampa ricorsiva di una lista

```
void StampaListaRic(ElementoDiLista* lis)
{
    if (lis != NULL){
        printf("%d ", lis->info);
        StampaListaRic(lis->next);
    }
    else
        printf("//");
}

int main(){
    ElementoDiLista* Lista1;
    ...
    /* costruzione lista
       5 --> 8 --> 4 */
    ...
    StampaListaRic(Lista1);
    ...
}
```



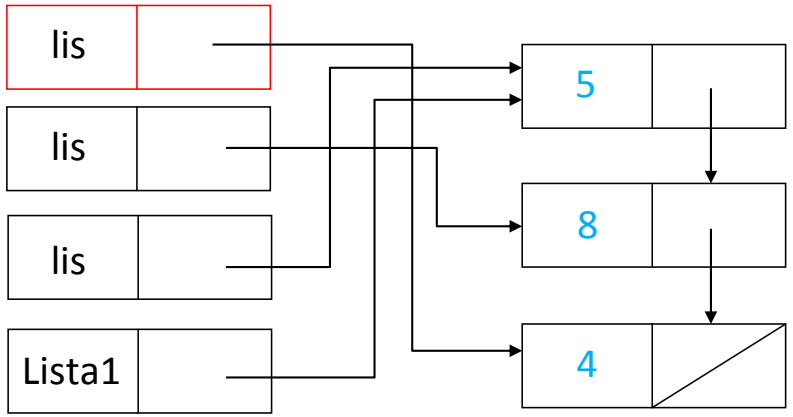
Output

5 -->

Stampa ricorsiva di una lista

```
void StampaListaRic(ElementoDiLista* lis)
{
    if (lis != NULL){
        printf("%d ", lis->info);
        StampaListaRic(lis->next);
    }
    else
        printf("//");
}

int main(){
    ElementoDiLista* Lista1;
    ...
    /* costruzione lista
       5 --> 8 --> 4 */
    ...
    StampaListaRic(Lista1);
    ...
}
```

PILA HEAP

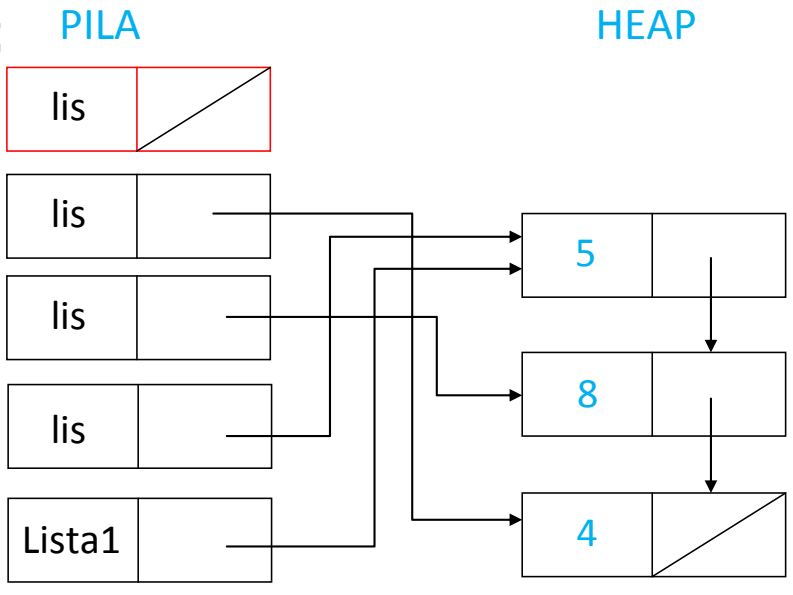


Output

5 --> 8 -->

Stampa ricorsiva di una lista

```
void StampaListaRic(ElementoDiLista* lis)
{
    if (lis != NULL){
        printf("%d ", lis->info);
        StampaListaRic(lis->next);
    }
    else
        printf("//");
}

int main(){
    ElementoDiLista* Lista1;
    ...
    /* costruzione lista
       5 --> 8 --> 4 */
    ...
    StampaListaRic(Lista1);
    ...
}
```



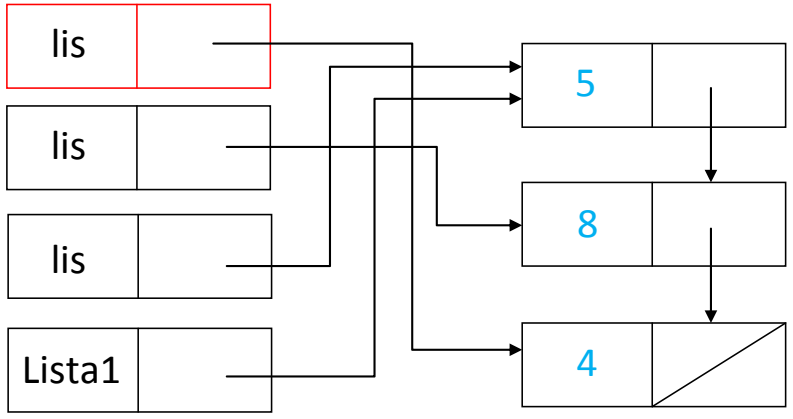
Output

5 --> 8 --> 4 -->

Stampa ricorsiva di una lista

```
void StampaListaRic(ElementoDiLista* lis)
{
    if (lis != NULL){
        printf("%d ", lis->info);
        StampaListaRic(lis->next);
    }
    else
        printf("//");
}

int main(){
    ElementoDiLista* Lista1;
    ...
    /* costruzione lista
       5 --> 8 --> 4 */
    ...
    StampaListaRic(Lista1);
    ...
}
```

PILA HEAP



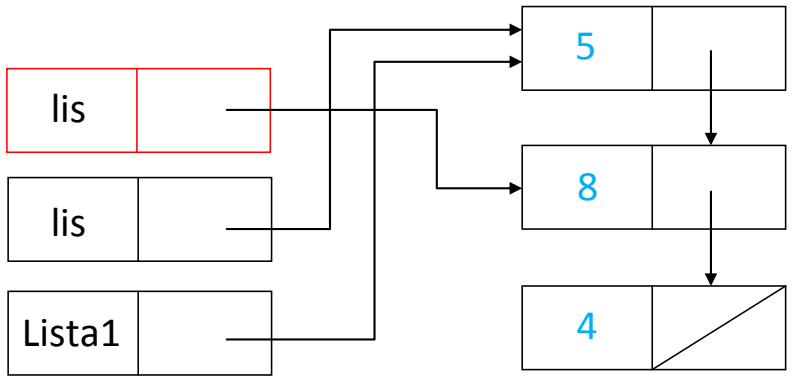
Output

5 --> 8 --> 4 --> //

Stampa ricorsiva di una lista

```
void StampaListaRic(ElementoDiLista* lis)
{
    if (lis != NULL){
        printf("%d ", lis->info);
        StampaListaRic(lis->next);
    }
    else
        printf("//");
}

int main(){
    ElementoDiLista* Lista1;
    ...
    /* costruzione lista
       5 --> 8 --> 4 */
    ...
    StampaListaRic(Lista1);
    ...
}
```

PILA HEAP



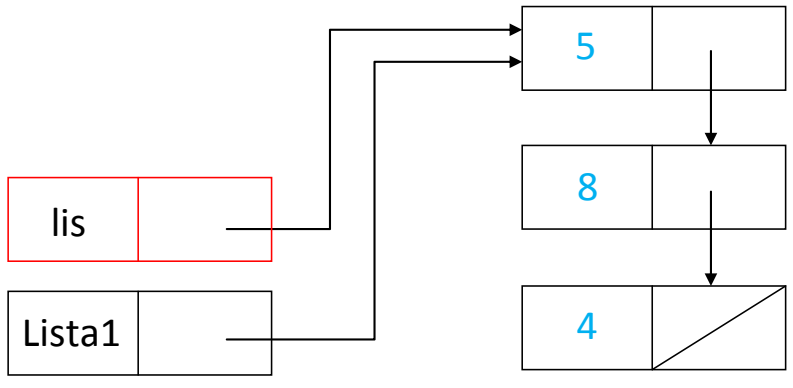
Output

5 --> 8 --> 4 --> //

Stampa ricorsiva di una lista

```
void StampaListaRic(ElementoDiLista* lis)
{
    if (lis != NULL){
        printf("%d ", lis->info);
        StampaListaRic(lis->next);
    }
    else
        printf("//");
}

int main(){
    ElementoDiLista* Lista1;
    ...
    /* costruzione lista
       5 --> 8 --> 4 */
    ...
    StampaListaRic(Lista1);
    ...
}
```

PILA HEAP



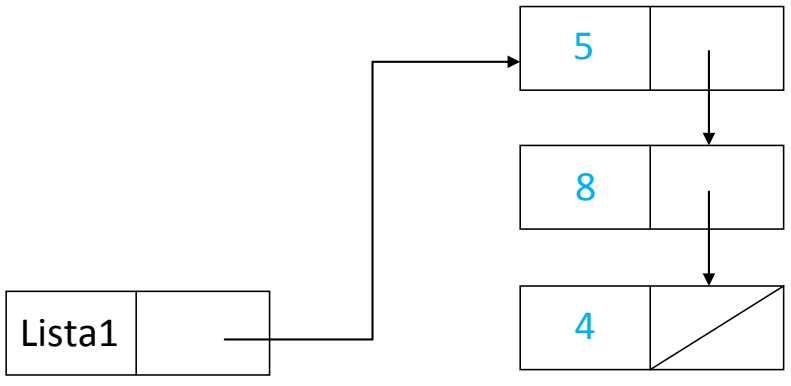
Output

5 --> 8 --> 4 --> //

Stampa ricorsiva di una lista

```
void StampaListaRic(ElementoDiLista* lis)
{
    if (lis != NULL){
        printf("%d ", lis->info);
        StampaListaRic(lis->next);
    }
    else
        printf("//");
}

int main(){
    ElementoDiLista* Lista1;
    ...
    /* costruzione lista
       5 --> 8 --> 4 */
    ...
    StampaListaRic(Lista1);
    ...
}
```

PILA HEAP



Output

5 --> 8 --> 4 --> //

Cancellazione di una lista

- Versione iterativa:

```
void CancellaLista(ElementoDiLista **lista)
{
    ElementoDiLista* aux;
    while (*lista != NULL) {
        aux = *lista;
        *lista = (*lista)->next;
        free(aux);
    }
}
```

Cancellazione di una lista

- Versione ricorsiva: sfruttiamo la visione ricorsiva della struttura dati lista per realizzare la cancellazione in modo ricorsivo

Cancellazione di una lista

- Versione ricorsiva: sfruttiamo la visione ricorsiva della struttura dati lista per realizzare la cancellazione in modo ricorsivo
  - la cancellazione della lista vuota non richiede alcuna azione

Cancellazione di una lista

- Versione ricorsiva: sfruttiamo la visione ricorsiva della struttura dati lista per realizzare la cancellazione in modo ricorsivo
  - la cancellazione della lista vuota non richiede alcuna azione
  - la cancellazione della lista ottenuta come concatenazione dell'elemento `x` e della lista `L` richiede l'eliminazione di `x` e la cancellazione di `L`.

Cancellazione di una lista

- Versione ricorsiva: sfruttiamo la visione ricorsiva della struttura dati lista per realizzare la cancellazione in modo ricorsivo
  - la cancellazione della lista vuota non richiede alcuna azione
  - la cancellazione della lista ottenuta come concatenazione dell'elemento `x` e della lista `L` richiede l'eliminazione di `x` e la cancellazione di `L`.

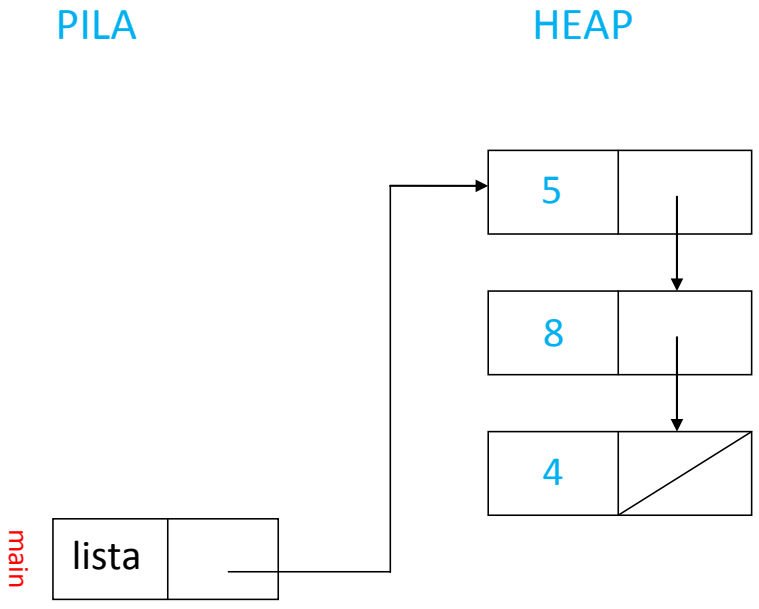
```
void CancellaListaRic(ElementoDiLista **lista)
{
    ElementoDiLista* aux;
    if (*lista != NULL)
    {
        aux = *lista;
       *lista = (*lista)->next;
        free(aux);
        CancellaListaRic(lista);
    }
}
```

Cancelazione di una lista

```
 void CancellaListaRic(ElementoDiLista **lista)
{
    ElementoDiLista* aux;
    if (*lista != NULL)
    {
        aux = *lista;
       *lista = (*lista)->next;
        free(aux);
        CancellaListaRic(lista);
    }
 }
```

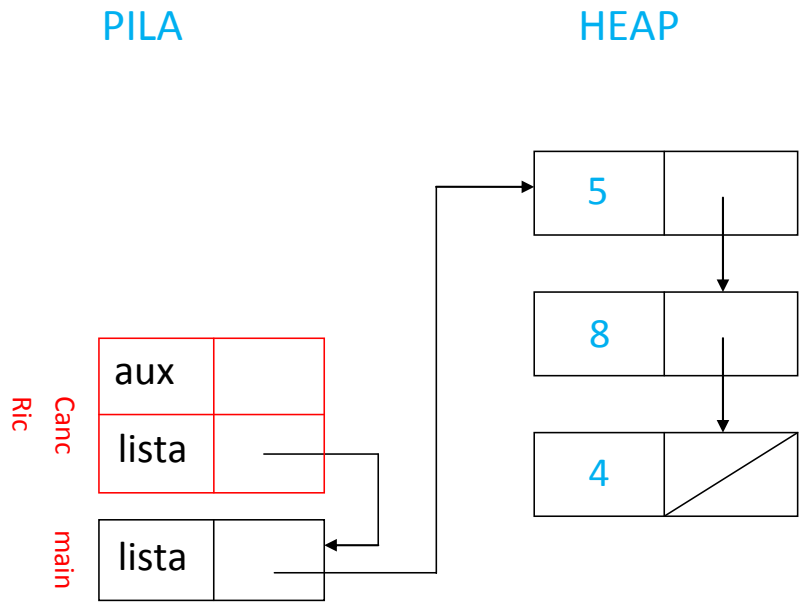
Cancelazione di una lista

```
void CancellaListaRic(ElementoDiLista **lista)
{
    ElementoDiLista* aux;
    if (*lista != NULL)
    {
        aux = *lista;
       *lista = (*lista)->next;
        free(aux);
        CancellaListaRic(lista);
    }
}
```



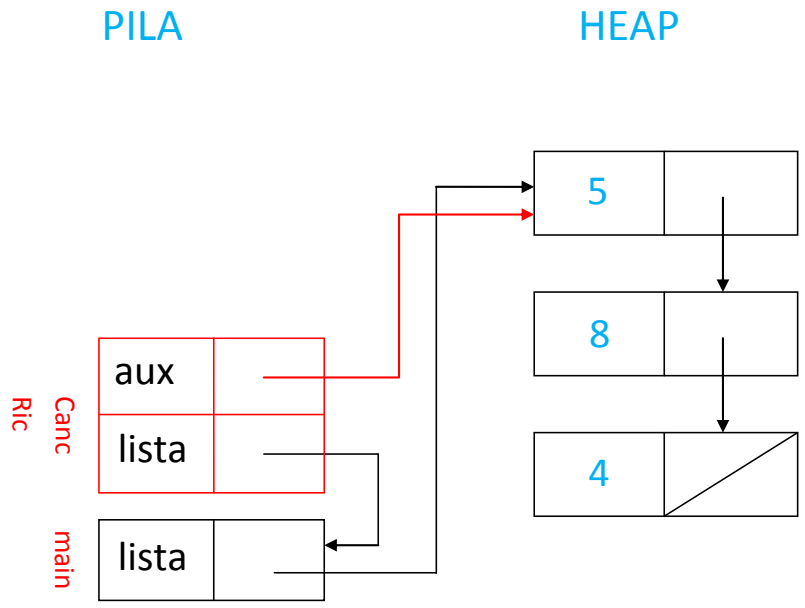
Cancelazione di una lista

```
void CancellaListaRic(ElementoDiLista **lista)
{
    ElementoDiLista* aux;
    if (*lista != NULL)
    {
        aux = *lista;
       *lista = (*lista)->next;
        free(aux);
        CancellaListaRic(lista);
    }
}
```



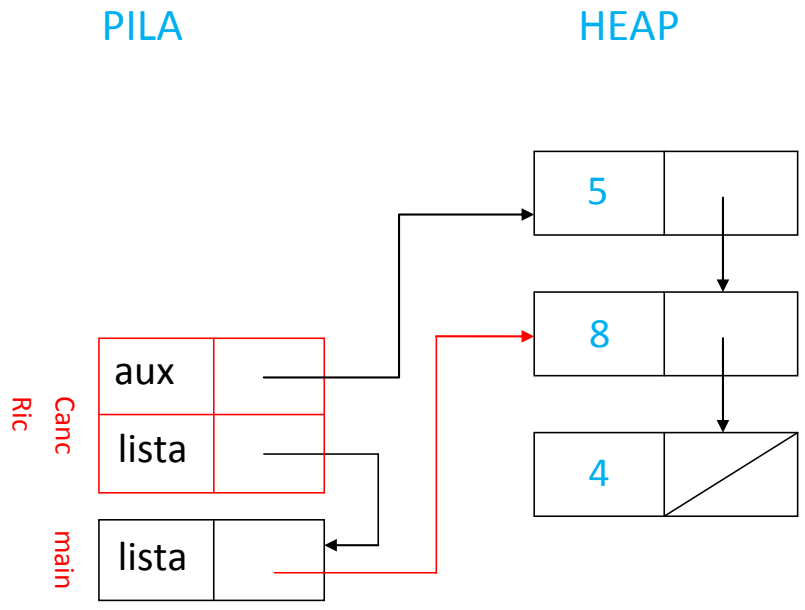
Cancelazione di una lista

```
void CancellaListaRic(ElementoDiLista **lista)
{
    ElementoDiLista* aux;
    if (*lista != NULL)
    {
        aux = *lista;
       *lista = (*lista)->next;
        free(aux);
        CancellaListaRic(lista);
    }
}
```



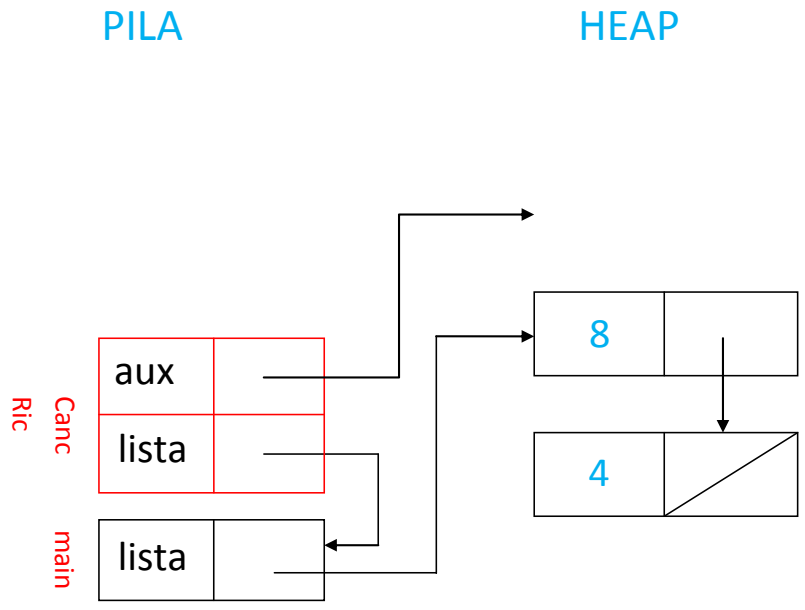
Cancelazione di una lista

```
void CancellaListaRic(ElementoDiLista **lista)
{
    ElementoDiLista* aux;
    if (*lista != NULL)
    {
        aux = *lista;
       *lista = (*lista)->next;
        free(aux);
        CancellaListaRic(lista);
    }
}
```



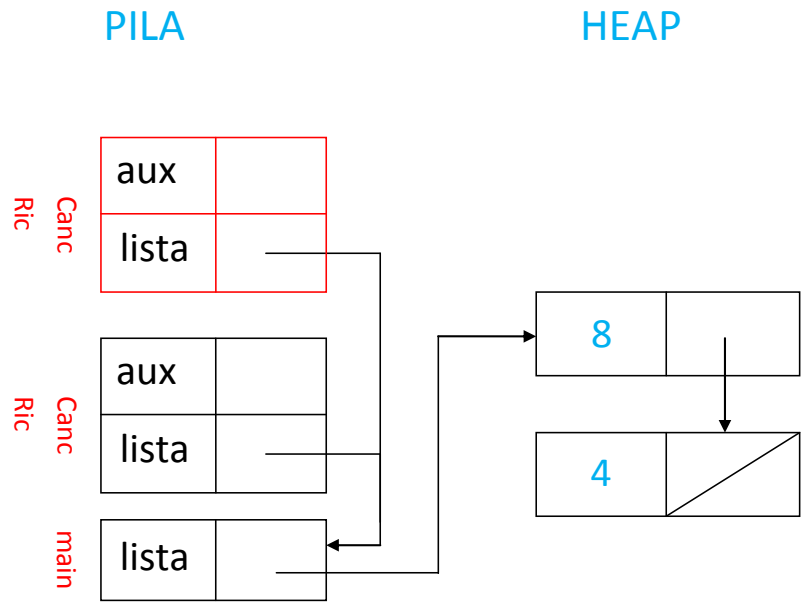
Cancelazione di una lista

```
void CancellaListaRic(ElementoDiLista **lista)
{
    ElementoDiLista* aux;
    if (*lista != NULL)
    {
        aux = *lista;
       *lista = (*lista)->next;
        free(aux);
        CancellaListaRic(lista);
    }
}
```



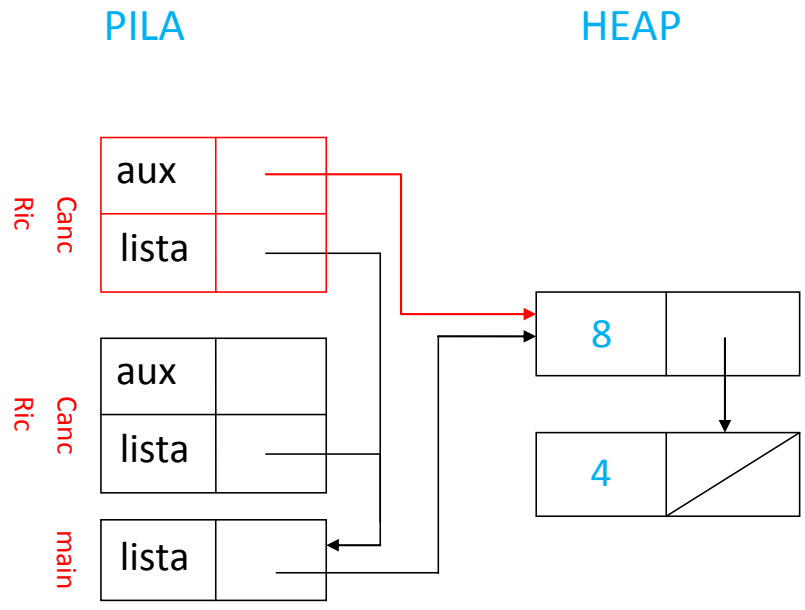
Cancelazione di una lista

```
void CancellaListaRic(ElementoDiLista **lista)
{
    ElementoDiLista* aux;
    if (*lista != NULL)
    {
        aux = *lista;
       *lista = (*lista)->next;
        free(aux);
        CancellaListaRic(lista);
    }
}
```



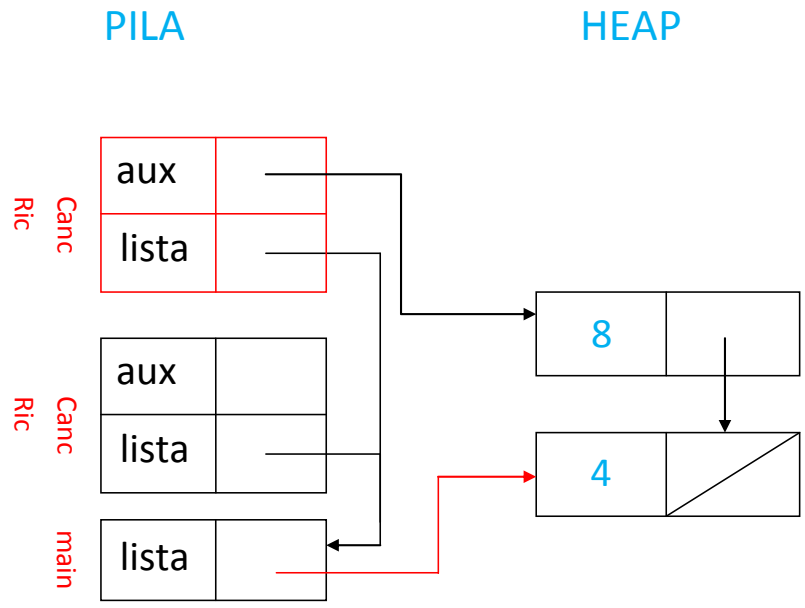
Cancelazione di una lista

```
void CancellaListaRic(ElementoDiLista **lista)
{
    ElementoDiLista* aux;
    if (*lista != NULL)
    {
        aux = *lista;
       *lista = (*lista)->next;
        free(aux);
        CancellaListaRic(lista);
    }
}
```



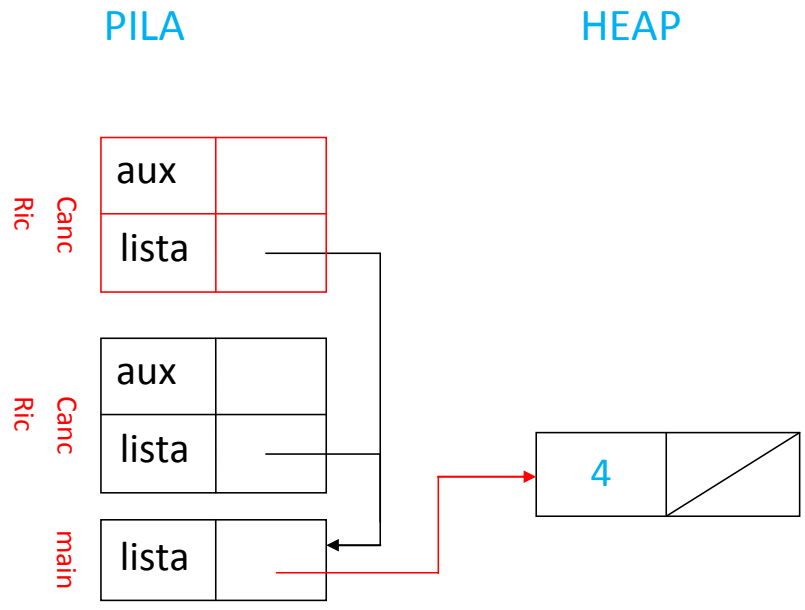
Cancelazione di una lista

```
void CancellaListaRic(ElementoDiLista **lista)
{
    ElementoDiLista* aux;
    if (*lista != NULL)
    {
        aux = *lista;
       *lista = (*lista)->next;
        free(aux);
        CancellaListaRic(lista);
    }
}
```



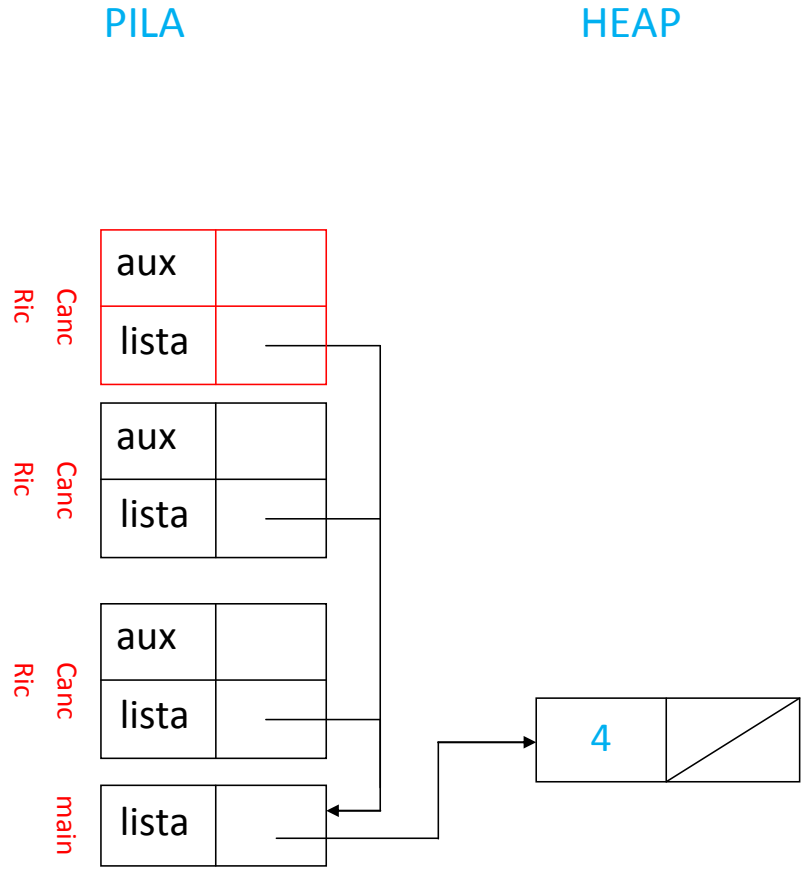
Cancelazione di una lista

```
void CancellaListaRic(ElementoDiLista **lista)
{
    ElementoDiLista* aux;
    if (*lista != NULL)
    {
        aux = *lista;
       *lista = (*lista)->next;
        free(aux);
        CancellaListaRic(lista);
    }
}
```



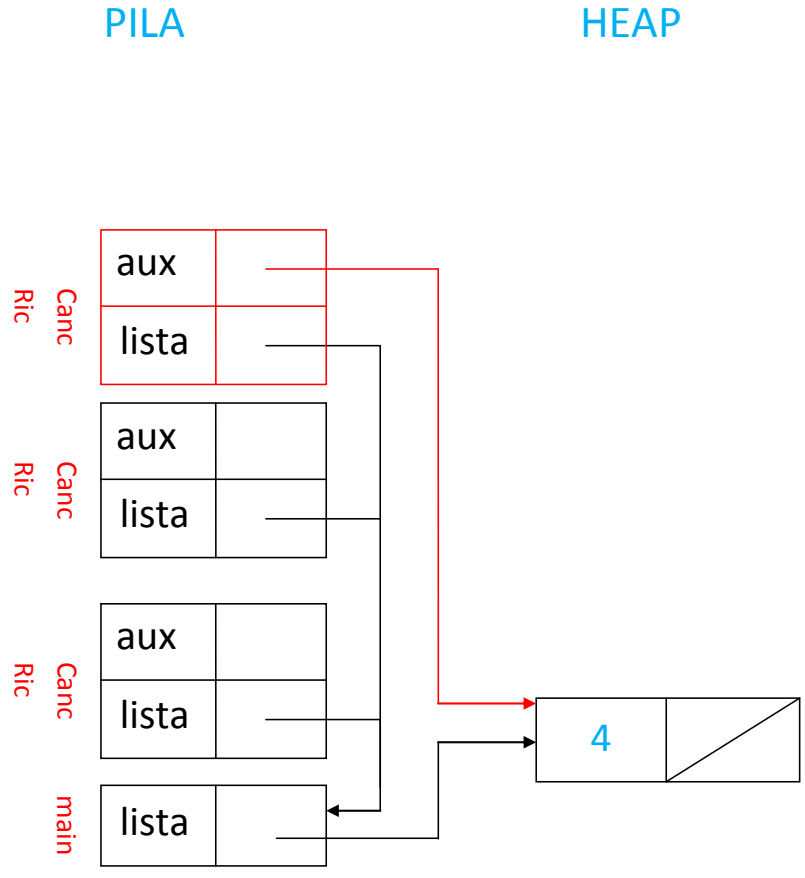
Cancelazione di una lista

```
void CancellaListaRic(ElementoDiLista **lista)
{
    ElementoDiLista* aux;
    if (*lista != NULL)
    {
        aux = *lista;
       *lista = (*lista)->next;
        free(aux);
        CancellaListaRic(lista);
    }
}
```



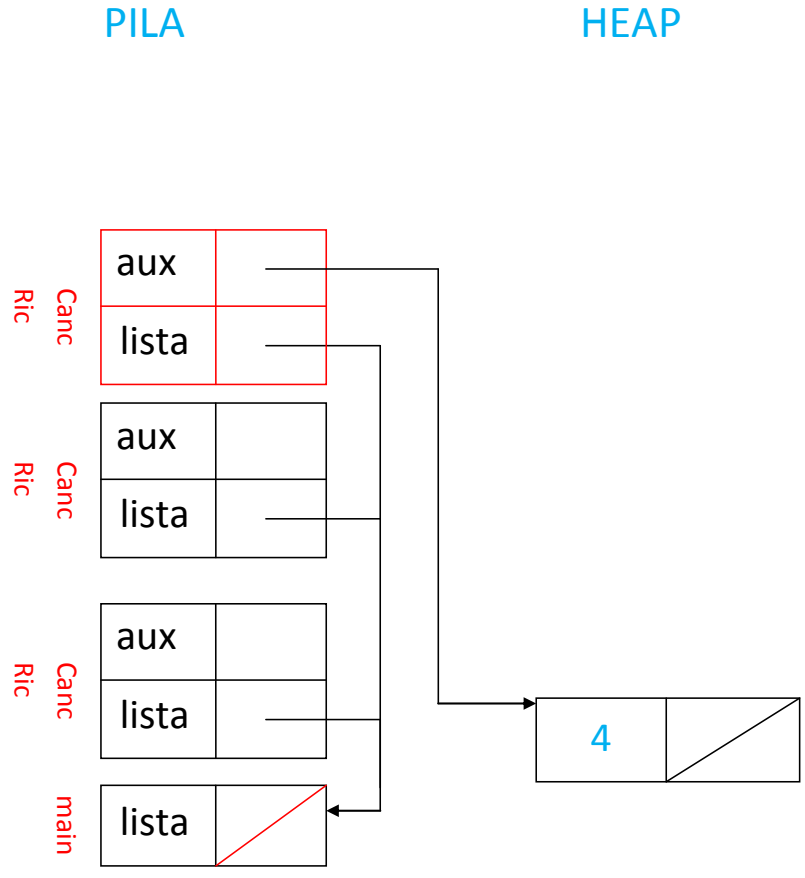
Cancelazione di una lista

```
void CancellaListaRic(ElementoDiLista **lista)
{
    ElementoDiLista* aux;
    if (*lista != NULL)
    {
        aux = *lista;
       *lista = (*lista)->next;
        free(aux);
        CancellaListaRic(lista);
    }
}
```



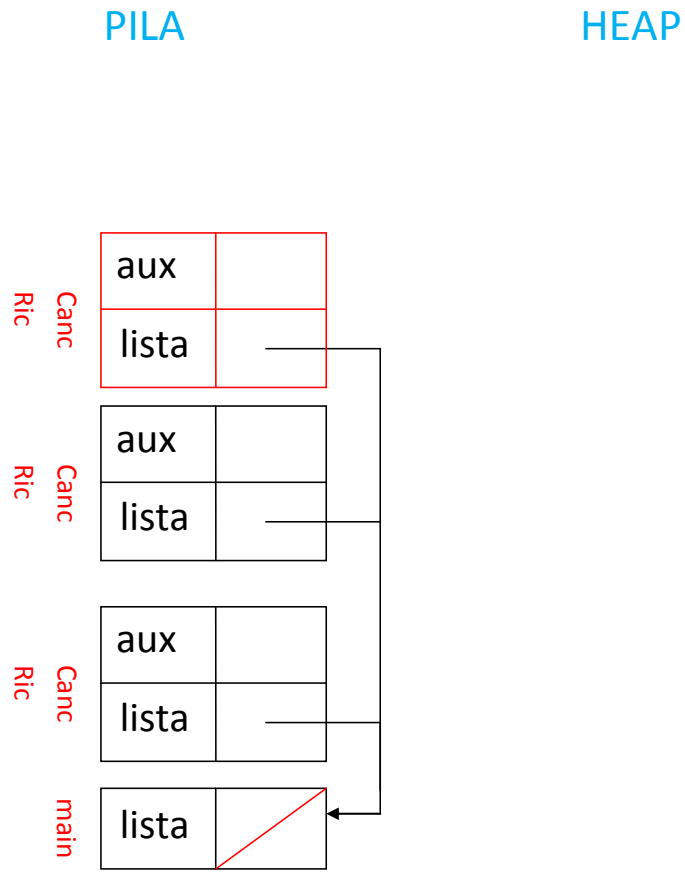
Cancelazione di una lista

```
void CancellaListaRic(ElementoDiLista **lista)
{
    ElementoDiLista* aux;
    if (*lista != NULL)
    {
        aux = *lista;
       *lista = (*lista)->next;
        free(aux);
        CancellaListaRic(lista);
    }
}
```



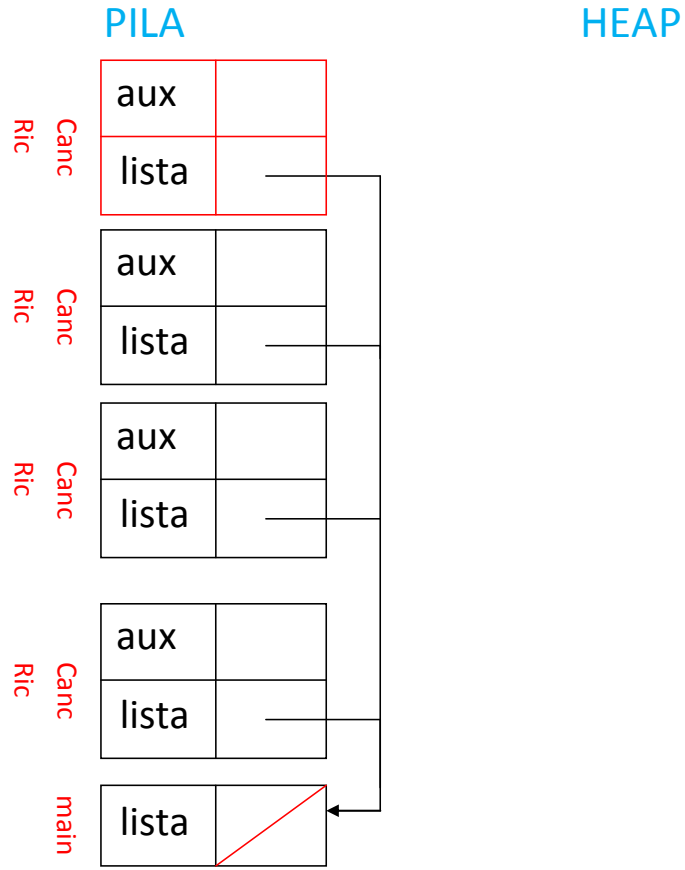
Cancelazione di una lista

```
void CancellaListaRic(ElementoDiLista **lista)
{
    ElementoDiLista* aux;
    if (*lista != NULL)
    {
        aux = *lista;
       *lista = (*lista)->next;
        free(aux);
        CancellaListaRic(lista);
    }
}
```



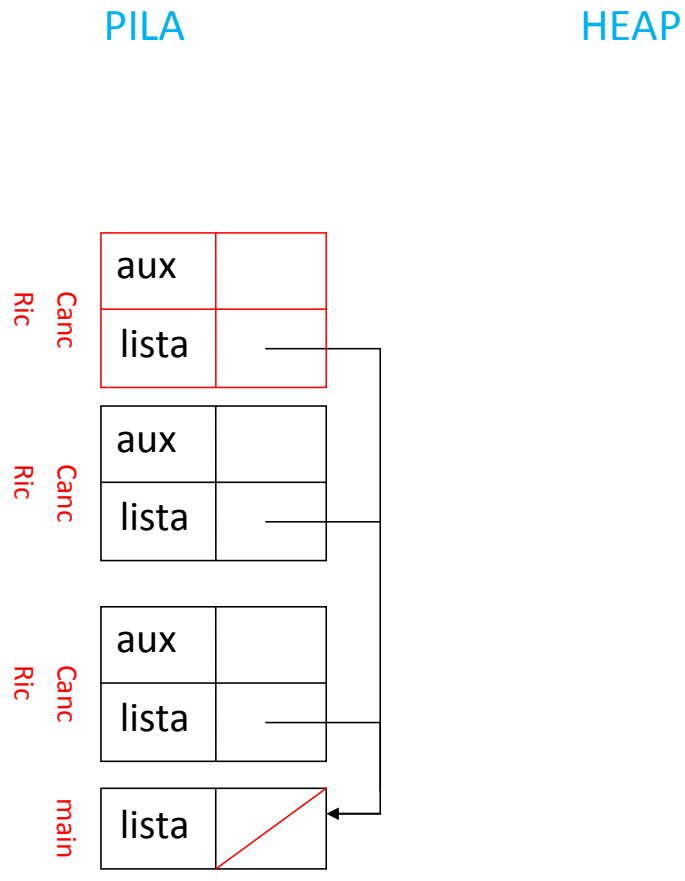
Cancelazione di una lista

```
void CancellaListaRic(ElementoDiLista **lista)
{
    ElementoDiLista* aux;
    if (*lista != NULL)
    {
        aux = *lista;
       *lista = (*lista)->next;
        free(aux);
        CancellaListaRic(lista);
    }
}
```



Cancelazione di una lista

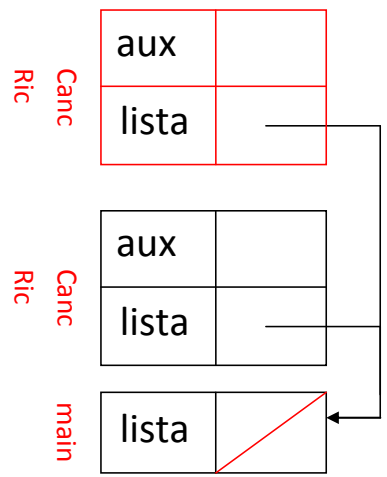
```
void CancellaListaRic(ElementoDiLista **lista)
{
    ElementoDiLista* aux;
    if (*lista != NULL)
    {
        aux = *lista;
       *lista = (*lista)->next;
        free(aux);
        CancellaListaRic(lista);
    }
}
```



Cancelazione di una lista

```
void CancellaListaRic(ElementoDiLista **lista)
{
    ElementoDiLista* aux;
    if (*lista != NULL)
    {
        aux = *lista;
       *lista = (*lista)->next;
        free(aux);
        CancellaListaRic(lista);
    }
}
```

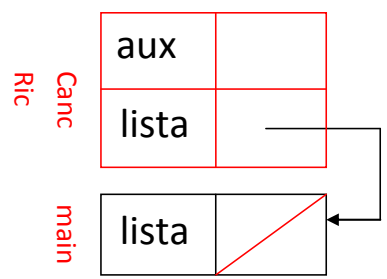
PILA HEAP



Cancelazione di una lista

```
void CancellaListaRic(ElementoDiLista **lista)
{
    ElementoDiLista* aux;
    if (*lista != NULL)
    {
        aux = *lista;
       *lista = (*lista)->next;
        free(aux);
        CancellaListaRic(lista);
    }
}
```

PILA HEAP



Cancelazione di una lista

```
void CancellaListaRic(ElementoDiLista **lista)
{
    ElementoDiLista* aux;
    if (*lista != NULL)
    {
        aux = *lista;
       *lista = (*lista)->next;
        free(aux);
        CancellaListaRic(lista);
    }
}
```

PILA HEAP

Appartenenza di un elemento a una lista

- Versione iterativa:

```
typedef enum {false,true} boolean;
boolean Appartiene(int elem, ElementoDiLista* lista)
{
    boolean trovato = false;
    while (lista != NULL && !trovato)
        if (lista->info==elem)
            trovato = true;
        else
            lista = lista->next;
    return trovato;
}
```

Appartenenza di un elemento a una lista

- Versione ricorsiva: un elemento `elem`
  - non appartiene alla lista vuota

Appartenenza di un elemento a una lista

- Versione ricorsiva: un elemento `elem`
  - non appartiene alla lista vuota
  - appartiene alla lista con testa `x` se `elem` coincide con `x`

Appartenenza di un elemento a una lista

- Versione ricorsiva: un elemento `elem`
  - non appartiene alla lista vuota
  - appartiene alla lista con testa `x` se `elem` coincide con `x`
  - appartiene alla lista con testa `x` diversa da `elem` e resto `L` se e solo se appartiene a `L`

Appartenenza di un elemento a una lista

- Versione ricorsiva: un elemento `elem`
  - non appartiene alla lista vuota
  - appartiene alla lista con testa `x` se `elem` coincide con `x`
  - appartiene alla lista con testa `x` diversa da `elem` e resto `L` se e solo se appartiene a `L`

```
boolean Appartiene( int elem, ElementoDiLista* lis)
{
    if (lis == NULL)
        return false;
    else
        if (lis->info==elem)
                return true;
           else
                return (Appartiene(elem, lis->next));
 }
```

Inserimento in coda di un elemento in una lista

- Versione ricorsiva: inseriamo elemento `elem` in lista `lista` ottenendo `nuovaLista`

Inserimento in coda di un elemento in una lista

- Versione ricorsiva: inseriamo elemento `elem` in lista `lista` ottenendo `nuovaLista`
  - se `lista` è vuota, allora `nuovaLista` è costituita dal solo `elem` (caso base)

Inserimento in coda di un elemento in una lista

- Versione ricorsiva: inseriamo elemento `elem` in lista `lista` ottenendo `nuovaLista`
  - se `lista` è vuota, allora `nuovaLista` è costituita dal solo `elem` (caso base)
  - altrimenti `nuovaLista` è ottenuta da `lista` facendo l'inserimento di `elem` in coda al resto di `lista` (caso ricorsivo)

Inserimento in coda di un elemento in una lista

- Versione ricorsiva: inseriamo elemento `elem` in lista `lista` ottenendo `nuovaLista`
  - se `lista` è vuota, allora `nuovaLista` è costituita dal solo `elem` (caso base)
  - altrimenti `nuovaLista` è ottenuta da `lista` facendo l'inserimento di `elem` in coda al resto di `lista` (caso ricorsivo)

```
void InserzioneInCoda(ElementoDiLista**lista, int elem){
    if (*lista == NULL){
        *lista = malloc(sizeof(ElementoDiLista));
        (*lista)->info = elem;
        (*lista)->next = NULL;
    }
    else
        InserzioneInCoda(&((*lista)->next), elem);
}
```

Cancellazione della prima occorrenza di un elemento

- si scandisce la lista alla ricerca dell'elemento

Cancellazione della prima occorrenza di un elemento

- si scandisce la lista alla ricerca dell'elemento
- se l'elemento non compare non si fa nulla

Cancellazione della prima occorrenza di un elemento

- si scandisce la lista alla ricerca dell'elemento
- se l'elemento non compare non si fa nulla
- altrimenti, a seconda di dove si trova l'elemento, si distinguono tre casi

Cancellazione della prima occorrenza di un elemento

- si scandisce la lista alla ricerca dell'elemento
- se l'elemento non compare non si fa nulla
- altrimenti, a seconda di dove si trova l'elemento, si distinguono tre casi
  - l'elemento è il primo della lista: si aggiorna il puntatore iniziale in modo che punti all'elemento successivo $\leftarrow$ passaggio per indirizzo!!

Cancellazione della prima occorrenza di un elemento

- si scandisce la lista alla ricerca dell'elemento
- se l'elemento non compare non si fa nulla
- altrimenti, a seconda di dove si trova l'elemento, si distinguono tre casi
  - l'elemento è il primo della lista: si aggiorna il puntatore iniziale in modo che punti all'elemento successivo $\leftarrow$ passaggio per indirizzo!!
  - l'elemento non è né il primo né l'ultimo: si aggiorna il campo next dell'elemento che precede quello da cancellare in modo che punti all'elemento che segue

Cancellazione della prima occorrenza di un elemento

- si scandisce la lista alla ricerca dell'elemento
- se l'elemento non compare non si fa nulla
- altrimenti, a seconda di dove si trova l'elemento, si distinguono tre casi
  - l'elemento è il primo della lista: si aggiorna il puntatore iniziale in modo che punti all'elemento successivo $\leftarrow$ passaggio per indirizzo!!
  - l'elemento non è né il primo né l'ultimo: si aggiorna il campo next dell'elemento che precede quello da cancellare in modo che punti all'elemento che segue
  - l'elemento è l'ultimo: come il secondo caso, solo che il campo `next` dell'elemento precedente viene posto a `NULL`

Cancellazione della prima occorrenza di un elemento

- si scandisce la lista alla ricerca dell'elemento
- se l'elemento non compare non si fa nulla
- altrimenti, a seconda di dove si trova l'elemento, si distinguono tre casi
  - l'elemento è il primo della lista: si aggiorna il puntatore iniziale in modo che punti all'elemento successivo $\leftarrow$ passaggio per indirizzo!!
  - l'elemento non è né il primo né l'ultimo: si aggiorna il campo next dell'elemento che precede quello da cancellare in modo che punti all'elemento che segue
  - l'elemento è l'ultimo: come il secondo caso, solo che il campo `next` dell'elemento precedente viene posto a `NULL`
- in tutti e tre i casi bisogna liberare la memoria occupata dall'elemento da cancellare

Cancellazione della prima occorrenza di un elemento

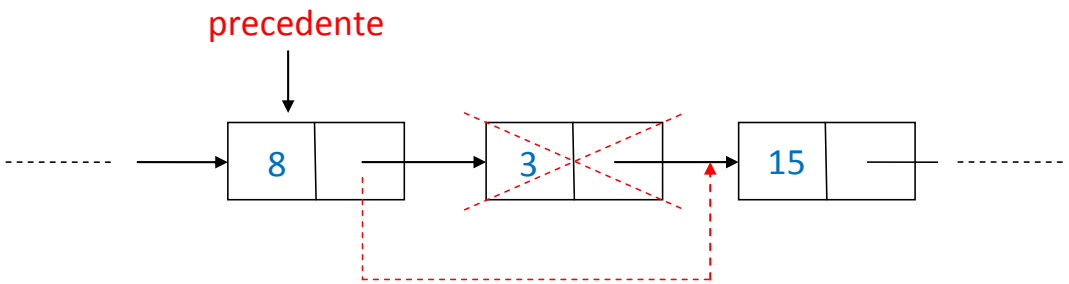
- Versione iterativa.
- Osservazioni:

Cancellazione della prima occorrenza di un elemento

- Versione iterativa.
- Osservazioni:
  - per poter aggiornare il campo `next` dell'elemento precedente, bisogna fermare la scansione sull'elemento precedente (e non su quello da cancellare)

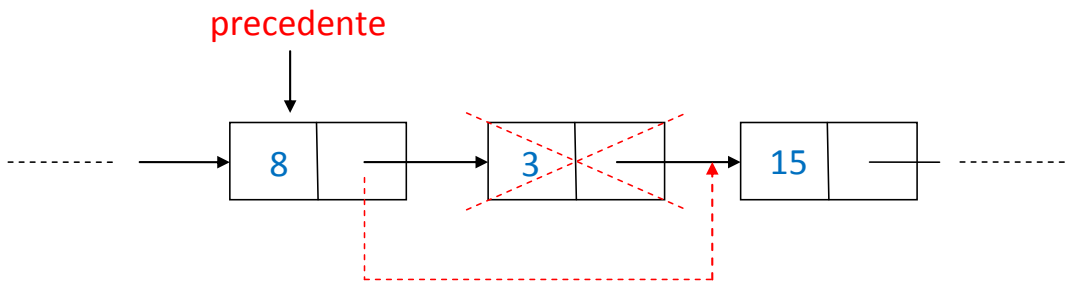
Cancellazione della prima occorrenza di un elemento

- Versione iterativa.
- Osservazioni:
  - per poter aggiornare il campo `next` dell'elemento precedente, bisogna fermare la scansione sull'elemento precedente (e non su quello da cancellare)



Cancellazione della prima occorrenza di un elemento

- Versione iterativa.
- Osservazioni:
  - per poter aggiornare il campo `next` dell'elemento precedente, bisogna fermare la scansione sull'elemento precedente (e non su quello da cancellare)



  - per fermare la scansione dopo aver trovato e cancellato l'elemento, si utilizza una sentinella booleana

Cancellazione della prima occorrenza di un elemento

- Versione iterativa.

```
void CancellaElem(ElementoDiLista **lista, int elem){
    ElementoDiLista* prec; /* elem precedente */
    ElementoDiLista* corr; /* elem corrente */
    boolean trovato; /* per terminare la scansione */
    if (*lista != NULL)
        if ((*lista)->info==elem){ /* cancella primo */
            ElementoDiLista* aux=*lista;
            *lista=*lista->next;
            free(aux);
        }
    else /* scansione della lista e cancellazione */
        prec = *lista; corr = prec->next; trovato = 0;
        while (corr != NULL && !trovato)
            if (corr->info == elem){ /* cancella elem */
                trovato=1; /*provoca uscita dal ciclo*/
                prec->next = corr->next;
                free(corr); }
            else { /* avanzamento dei due puntatori */
                prec = prec->next;
                corr = corr->next;  }
}
```

Cancellazione della prima occorrenza di un elemento

- Versione ricorsiva.

```
void CancellaElemRic(ElementoDiLista **lista,
    TipoElementoLista elem){

    if (*lista != NULL)
        if ((*lista)->info== elem){
            /* cancella il primo elemento */
            ElementoDiLista* aux=*lista;
            *lista=*lista->next;
            free(aux);
    }

    else {
        /* cancella elem dal resto */
        CancellaElemRic(&((*lista)->next), elem);
     }
}
```

Cancellazione di tutte le occorrenze

- Caratterizzazione induttiva: sia `ris` la lista ottenuta cancellando tutte le occorrenze di `elem` da `lista`.

Cancellazione di tutte le occorrenze

- Caratterizzazione induttiva: sia `ris` la lista ottenuta cancellando tutte le occorrenze di `elem` da `lista`.
  - se `lista` è la lista vuota, allora `ris` è la lista vuota (caso base)

Cancellazione di tutte le occorrenze

- Caratterizzazione induttiva: sia `ris` la lista ottenuta cancellando tutte le occorrenze di `elem` da `lista`.
  - se `lista` è la lista vuota, allora `ris` è la lista vuota (caso base)
  - altrimenti, se il primo elemento di `lista` è uguale ad `elem`, allora `ris` è ottenuta da `lista` cancellando il primo elemento e tutte le occorrenze di `elem` dal resto di `lista` (caso ricorsivo)

Cancellazione di tutte le occorrenze

- Caratterizzazione induttiva: sia `ris` la lista ottenuta cancellando tutte le occorrenze di `elem` da `lista`.
  - se `lista` è la lista vuota, allora `ris` è la lista vuota (caso base)
  - altrimenti, se il primo elemento di `lista` è uguale ad `elem`, allora `ris` è ottenuta da `lista` cancellando il primo elemento e tutte le occorrenze di `elem` dal resto di `lista` (caso ricorsivo)
  - altrimenti `ris` è ottenuta da `lista` cancellando tutte le occorrenze di `elem` dal resto di `lista` (caso ricorsivo)

Cancellazione di tutte le occorrenze

- Versione ricorsiva.

```
void CancellaTuttiRic(ElementoDiLista **lista, int elem)
    {
     ElementoDiLista* aux;
     if (*lista != NULL)
          if ((*lista)->info==elem) {
               /* cancellazione del primo elemento */
               *lista=*lista->next;
               free(aux);
               /* cancellazione di elem dal resto della
    lista */
              CancellaTuttiRic(lista, elem);
          }
          else
               CancellaTuttiRic(&((*lista)->next), elem);
}
```