

## Programmazione ricorsiva.

- ▶ In quasi tutti i linguaggi di programmazione è ammessa la possibilità di definire funzioni/procedure **ricorsive**: durante l'esecuzione di una funzione **F** è possibile chiamare la funzione **F** stessa.

## Programmazione ricorsiva.

- ▶ In quasi tutti i linguaggi di programmazione è ammessa la possibilità di definire funzioni/procedure **ricorsive**: durante l'esecuzione di una funzione **F** è possibile chiamare la funzione **F** stessa.
- ▶ Ciò può avvenire

## Programmazione ricorsiva.

- ▶ In quasi tutti i linguaggi di programmazione è ammessa la possibilità di definire funzioni/procedure **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.

# Programmazione ricorsiva.

- ▶ In quasi tutti i linguaggi di programmazione è ammessa la possibilità di definire funzioni/procedure **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**: **F** contiene una chiamata a **G** che a sua volta contiene una chiamata a **F**.

## Programmazione ricorsiva.

- ▶ In quasi tutti i linguaggi di programmazione è ammessa la possibilità di definire funzioni/procedure **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**:  $F$  contiene una chiamata a  $G$  che a sua volta contiene una chiamata a  $F$ .
- ▶ Può sembrare strano: se pensiamo che una funzione è destinata a risolvere un sottoproblema  $\mathcal{P}$ , una definizione ricorsiva sembra indicare che per risolvere  $\mathcal{P}$  dobbiamo . . . saper risolvere  $\mathcal{P}$ !

- ▶ 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 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.

- ▶ 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.



- ▶ Le funzioni ricorsive sono convenienti per implementare funzioni matematiche definite in modo **induttivo**.

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

- ▶ 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.

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

- ▶ 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.

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

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

- ▶ 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.

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

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

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

## Il codice delle due diverse versioni

► definizione iterativa:

```
int fatt(int n) {
    int i,ris;

    ris=1;
    for (i=1;i<=n;i++)
        ris=ris*i;
    return ris;
}
```

## Il codice delle due diverse versioni

► definizione iterativa:

```
int fatt(int n) {
    int i,ris;

    ris=1;
    for (i=1;i<=n;i++)
        ris=ris*i;
    return ris;
}
```

► definizione ricorsiva:

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

**Esempio:** Programma che usa una funzione ricorsiva.

```
#include <stdio.h>

int fattoric (int);

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

int fattoric(int n) {
  int ris;
  if (n == 0)
    ris = 1;
  else
    ris = n * fattoric(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	?

x	3
f	6

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

- ▶ Problema: prima di poter iniziare a stampare dobbiamo aver letto e memorizzato tutta la sequenza:

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

- ▶ Problema: prima di poter iniziare a stampare dobbiamo aver letto e memorizzato tutta la sequenza:
  1. usando una struttura dati opportuna ma **dinamica** come le liste

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

- ▶ Problema: prima di poter iniziare a stampare dobbiamo aver letto e memorizzato tutta la sequenza:
  1. usando una struttura dati opportuna ma **dinamica** come le liste
  2. usando un procedimento ricorsivo.

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

- ▶ Problema: prima di poter iniziare a stampare dobbiamo aver letto e memorizzato tutta la sequenza:
  1. usando una struttura dati opportuna ma **dinamica** come le liste
  2. usando un procedimento ricorsivo.
    - ▶ leggiamo un carattere della sequenza, `c1`, leggiamo e stampiamo ricorsivamente il resto della sequenza `c2...cn` e infine stampiamo `c1`;

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

- ▶ Problema: prima di poter iniziare a stampare dobbiamo aver letto e memorizzato tutta la sequenza:
  1. usando una struttura dati opportuna ma **dinamica** come le liste
  2. 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:** Leggere una sequenza di caratteri terminata da `'\n'` e stamparla invertita. Ad esempio: `casa`  $\implies$  `asac`

- ▶ Problema: prima di poter iniziare a stampare dobbiamo aver letto e memorizzato tutta la sequenza:
  1. usando una struttura dati opportuna ma **dinamica** come le liste
  2. 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.

```
void invertInputRic()
{ char ch;

scanf("%c", &ch);
if (ch != '\n')
{
invertInputRic();
printf("%c", ch);
}
else
printf("Sequenza invertita: ");
}
```

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

Vediamo come evolve la pila per l'input ABC\n



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

Vediamo come evolve la pila per l'input ABC\n

ch	A
----	---

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

Vediamo come evolve la pila per l'input ABC\n

ch	A
----	---

ch	B
----	---

ch	A
----	---

```

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

```

Vediamo come evolve la pila per l'input ABC\n

ch	A
----	---

ch	B
----	---

ch	A
----	---

ch	C
----	---

ch	B
----	---

ch	A
----	---

```

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

```

Vediamo come evolve la pila per l'input ABC\n

ch	A
----	---

ch	B
ch	A

ch	C
ch	B
ch	A

ch	\n
ch	C
ch	B
ch	A

```

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

```

Vediamo come evolve la pila per l'input ABC\n

ch	A
----	---

ch	B
ch	A

ch	C
ch	B
ch	A

ch	\n
ch	C
ch	B
ch	A

ch	C
ch	B
ch	A

```

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

```

Vediamo come evolve la pila per l'input ABC\n

ch	A
----	---

ch	B
----	---

ch	C
----	---

ch	\n
----	----

ch	A
----	---

ch	B
----	---

ch	C
----	---

ch	A
----	---

ch	B
----	---

ch	A
----	---

ch	C
----	---

ch	B
----	---

ch	A
----	---

ch	B
----	---

ch	A
----	---

ch	A
----	---

```

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

```

Vediamo come evolve la pila per l'input ABC\n

ch	A
----	---

ch	B
----	---

ch	C
----	---

ch	\n
----	----

ch	A
----	---

ch	B
----	---

ch	C
----	---

ch	A
----	---

ch	B
----	---

ch	A
----	---

ch	C
----	---

ch	B
----	---

ch	A
----	---

ch	B
----	---

ch	A
----	---

ch	A
----	---

L'output prodotto è il seguente

Sequenza invertita: CBA

Possiamo usare anche la ricorsione per operare sugli array:

Calcolare ricorsivamente la somma degli elementi di un array  $v$  di dimensione  $dim$ .

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



Calcolare ricorsivamente il numero di occorrenze dell'elemento  $x$  nell' array  $v$  di dimensioni  $dim$ .

```
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;
}
```

Calcolare ricorsivamente il numero di occorrenze dell'elemento  $x$  nell' array  $v$  di dimensioni  $dim$ .

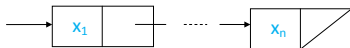
```
int occorrenze (int *v, int from, int to)
{
    int occ;

    if (from > to)
        occ= 0;
    else
        if (v[from]!=x)
            occ = occorrenze(v,from+1,to,x);
        else
            occ = 1+occorrenze(v,from+1,to,x);
    return occ;
}
```

## Visione ricorsiva delle liste

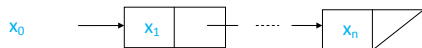
- ▶ Una lista di elementi è una struttura dati ricorsiva per sua natura

## Visione ricorsiva delle liste



- ▶ Una lista di elementi è una struttura dati ricorsiva per sua natura
  1. data una lista  $L$  di elementi  $x_1, \dots, x_n$

## Visione ricorsiva delle liste



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

## Visione ricorsiva delle liste



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

## Procedure e funzioni ricorsive su liste

Consistono, in genere, nel:

- ▶ individuare il **caso base** e risolverlo direttamente: di solito **lista vuota**.

Ad esempio per la stampa di una lista

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



## Procedure e 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(ListaDiElementi lis)
{
    if (lis != NULL)
    {
        printf("%d ", lis->info);
        StampaListaRic(lis->next);
    }
    else
        printf("//");
}
```

## Versione ricorsiva

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

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

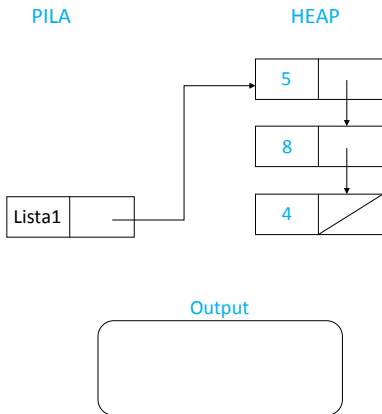
## Versione ricorsiva

```

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

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

```



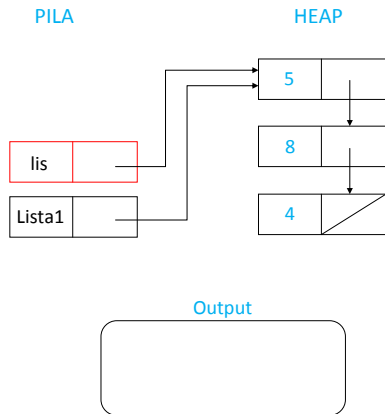
## Versione ricorsiva

```

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

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

```



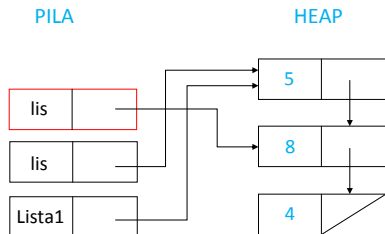
## Versione ricorsiva

```

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

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

```



Output

5 -->

## Versione ricorsiva

```

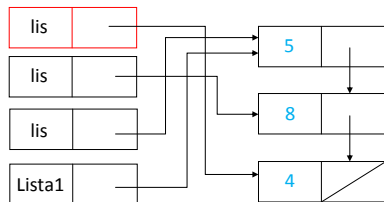
void StampaListaRic(ListaDiElementi lis)
{
    if (lis != NULL)
    {
        printf("%d ", lis->info);
        StampaListaRic(lis->next);
    }
    else
        printf("//");
}

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

```

PILA

HEAP



Output

5 --&gt; 8 --&gt;

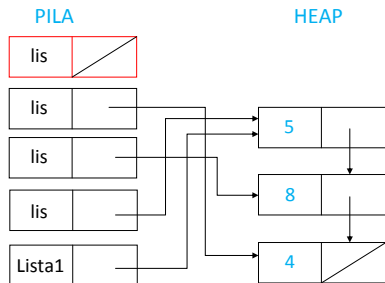
## Versione ricorsiva

```

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

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

```



Output

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

## Versione ricorsiva

```

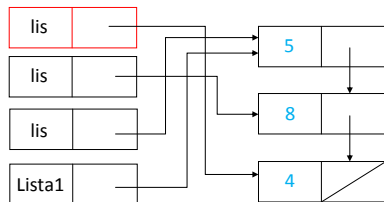
void StampaListaRic(ListaDiElementi lis)
{
    if (lis != NULL)
    {
        printf("%d ", lis->info);
        StampaListaRic(lis->next);
    }
    else
        printf("//");
}

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

```

PILA

HEAP



Output

5 --&gt; 8 --&gt; 4 --&gt; //



## Versione ricorsiva

```

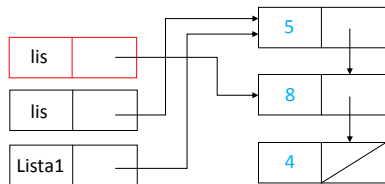
void StampaListaRic(ListaDiElementi lis)
{
    if (lis != NULL)
    {
        printf("%d ", lis->info);
        StampaListaRic(lis->next);
    }
    else
        printf("//");
}

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

```

PILA

HEAP



Output

5 --&gt; 8 --&gt; 4 --&gt; //

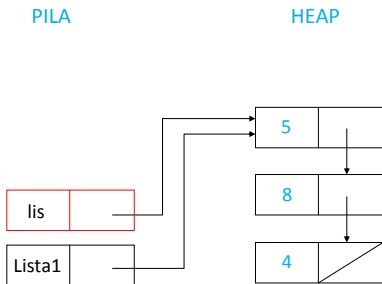
## Versione ricorsiva

```

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

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

```



Output

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

## Versione ricorsiva

```

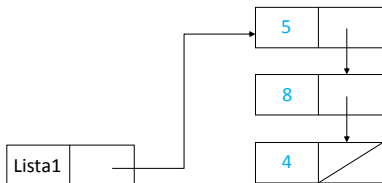
void StampaListaRic(ListaDiElementi lis)
{
    if (lis != NULL)
    {
        printf("%d ", lis->info);
        StampaListaRic(lis->next);
    }
    else
        printf("//");
}

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

```

PILA

HEAP



Output

5 --&gt; 8 --&gt; 4 --&gt; //

## Cancellazione di una lista: versione iterativa

```
void CancellaLista(ListaDiElementi *lista)
{
    ListaDiElementi 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**
  1. 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**
  1. la cancellazione della lista vuota non richiede alcuna azione
  2. 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**
  1. la cancellazione della lista vuota non richiede alcuna azione
  2. la cancellazione della lista ottenuta come concatenazione dell'elemento  $x$  e della lista  $L$  richiede l'eliminazione di  $x$  e la cancellazione di  $L$
- ▶ la traduzione in **C** è immediata



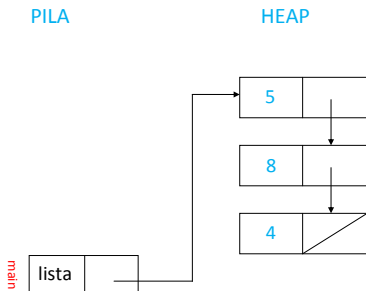
## Cancellazione di una lista: versione ricorsiva

- ▶ Sfruttiamo la visione ricorsiva della struttura dati lista per realizzare la cancellazione in modo **ricorsivo**
  1. la cancellazione della lista vuota non richiede alcuna azione
  2. la cancellazione della lista ottenuta come concatenazione dell'elemento  $x$  e della lista  $L$  richiede l'eliminazione di  $x$  e la cancellazione di  $L$
- ▶ la traduzione in **C** è immediata

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

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

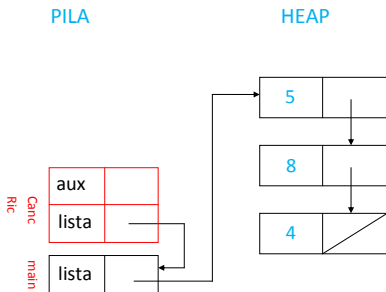
```
void CancellaListaRic(ListaDiElementi *lista)
{
    ListaDiElementi aux;
    if (*lista != NULL)
    {
        aux = *lista;
        *lista = (*lista)->next;
        free(aux);
        CancellaListaRic(lista);
    }
}
```



```

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

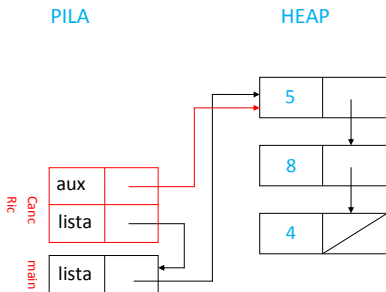
```



```

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

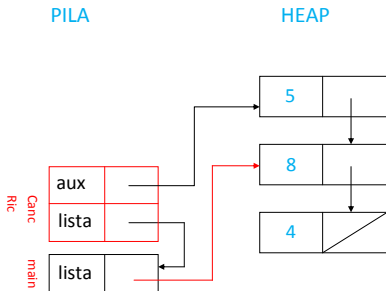
```



```

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

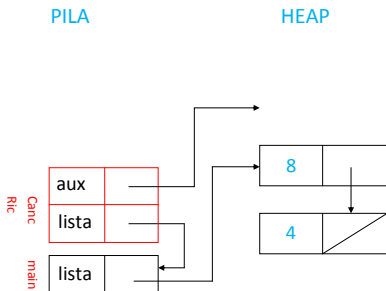
```



```

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

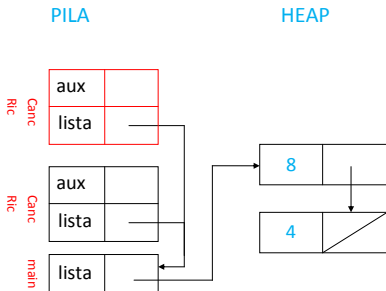
```



```

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

```

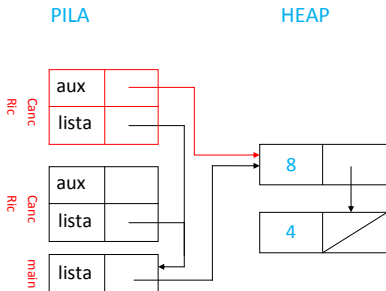




```

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

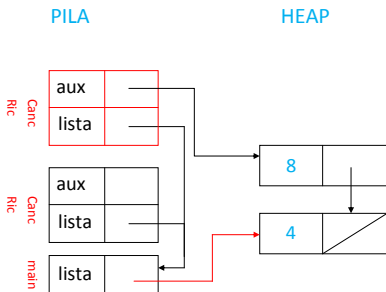
```



```

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

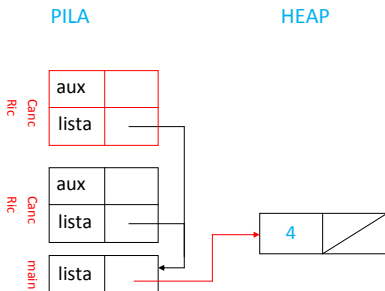
```



```

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

```



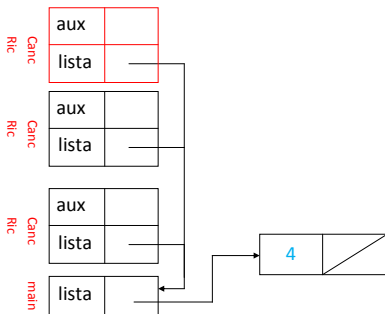
```

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

```

PILA

HEAP



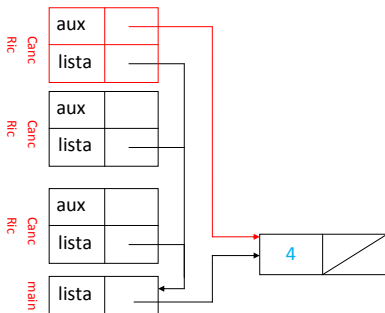
```

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

```

PILA

HEAP



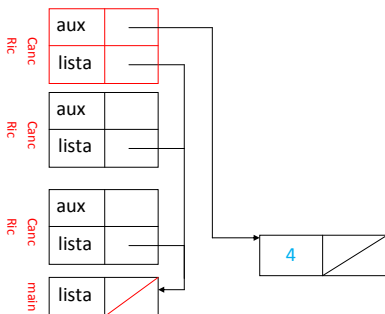
```

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

```

PILA

HEAP



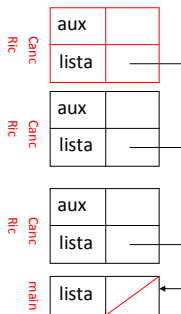
```

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

```

PILA

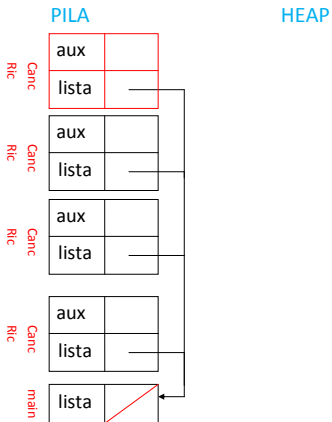
HEAP



```

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

```





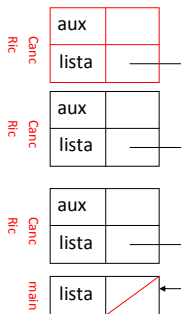
```

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

```

PILA

HEAP



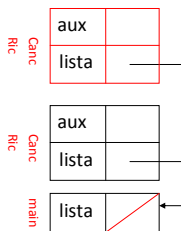
```

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

```

PILA

HEAP



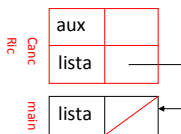
```

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

```

PILA

HEAP

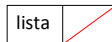


```
void CancellaListaRic(ListaDiElementi *lista)
{
    ListaDiElementi aux;
    if (*lista != NULL)
    {
        aux = *lista;
        *lista = (*lista)->next;
        free(aux);
        CancellaListaRic(lista);
    }
}
```

PILA

HEAP

main



## Appartenenza di un elemento ad una lista: versione iterativa

```
typedef enum {false,true} boolean;

boolean Appartiene(int elem, ListaDiElementi lista)
{
    boolean trovato = false;

    while (lista != NULL && !trovato)
        if (lista->info==elem)
            trovato = true;
        else
            lista = lista->next;
    return trovato;
}
```

## Appartenenza di un elemento: versione ricorsiva

## Appartenenza di un elemento: versione ricorsiva

- ▶ Un elemento `elem`

## Appartenenza di un elemento: versione ricorsiva

- ▶ Un elemento `elem`
  - ▶ non appartiene alla lista vuota



## Appartenenza di un elemento: 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: 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: versione ricorsiva

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

- ▶ 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`

## Inserimento di un elemento in coda ad una lista: versione ricorsiva

## Inserimento di un elemento in coda ad una lista: versione ricorsiva

Caratterizzazione **induttiva** dell'inserimento in coda

## Inserimento di un elemento in coda ad una lista: versione ricorsiva

Caratterizzazione **induttiva** dell'inserimento in coda Sia **nuovaLista** la lista ottenuta inserendo **elem** in coda a **lista**.

- ▶ se **lista** è vuota, allora **nuovaLista** è costituita dal solo **elem** (**caso base**)

## Inserimento di un elemento in coda ad una lista: versione ricorsiva

Caratterizzazione **induttiva** dell'inserimento in coda Sia **nuovaLista** la lista ottenuta inserendo **elem** in coda a **lista**.

- ▶ 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 di un elemento in coda ad una lista: versione ricorsiva

Caratterizzazione **induttiva** dell'inserimento in coda Sia **nuovaLista** la lista ottenuta inserendo **elem** in coda a **lista**.

- ▶ 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(ListaDiElementi *lista, int elem)
{if (*lista == NULL)
  {
    *lista = malloc(sizeof(ElementoLista));
    (*lista)->info = elem;
    (*lista)->next = NULL;
  }
else

InserisciInCoda(      ??      , elem);
}}
```



## Inserimento di un elemento in coda ad una lista: versione ricorsiva

Caratterizzazione **induttiva** dell'inserimento in coda Sia `nuovaLista` la lista ottenuta inserendo `elem` in coda a `lista`.

- ▶ 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(ListaDiElementi *lista, int elem)
{if (*lista == NULL)
  {
    *lista = malloc(sizeof(ElementoLista));
    (*lista)->info = elem;
    (*lista)->next = NULL;
  }
else

InserisciInCoda(      ??      , elem);
}}
```

## Inserimento di un elemento in coda ad una lista: versione ricorsiva

Caratterizzazione **induttiva** dell'inserimento in coda Sia **nuovaLista** la lista ottenuta inserendo **elem** in coda a **lista**.

- ▶ 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(ListaDiElementi *lista, int elem)
{if (*lista == NULL)
  {
    *lista = malloc(sizeof(ElementoLista));
    (*lista)->info = elem;
    (*lista)->next = NULL;
  }
else

InserisciInCoda(      ??      , elem);
}}

InserzioneInCoda(&((*lista)->next), elem);
```

## Inserimento di un elemento in coda ad una lista: versione ricorsiva

Caratterizzazione **induttiva** dell'inserimento in coda Sia **nuovaLista** la lista ottenuta inserendo **elem** in coda a **lista**.

- ▶ 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(ListaDiElementi *lista, int elem)
{if (*lista == NULL)
  {
    *lista = malloc(sizeof(ElementoLista));
    (*lista)->info = elem;
    (*lista)->next = NULL;
  }
else

InserisciInCoda(      ??      , elem);
}}

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
  1. l'elemento è il primo della lista: si aggiorna il puntatore iniziale in modo che punti all'elemento successivo

## 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
  1. l'elemento è il primo della lista: si aggiorna il puntatore iniziale in modo che punti all'elemento successivo  
⇒ 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
  1. l'elemento è il primo della lista: si aggiorna il puntatore iniziale in modo che punti all'elemento successivo  
⇒ passaggio per indirizzo!!
  2. 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
  1. l'elemento è il primo della lista: si aggiorna il puntatore iniziale in modo che punti all'elemento successivo  
⇒ passaggio per indirizzo!!
  2. 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
  3. l'elemento è l'ultimo: come (2), solo che il campo `next` dell'elemento precedente viene posto a `NULL`

## Cancellazione della prima occorrenza di un elemento

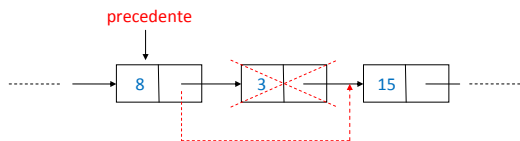
- ▶ si scandisce la lista alla ricerca dell'elemento
- ▶ se l'elemento non compare non si fa nulla
- ▶ altrimenti, a seconda di dove si trova l'elemento, si distinguono tre casi
  1. l'elemento è il primo della lista: si aggiorna il puntatore iniziale in modo che punti all'elemento successivo  
⇒ passaggio per indirizzo!!
  2. 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
  3. l'elemento è l'ultimo: come (2), 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

## Osservazioni:

- ▶ per poter aggiornare il campo `next` dell'elemento precedente, bisogna **fermare la scansione sull'elemento precedente** (e non su quello da cancellare)

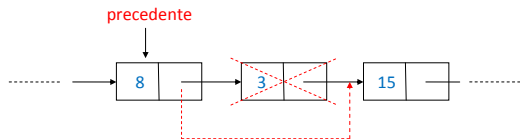
## Osservazioni:

- ▶ per poter aggiornare il campo **next** dell'elemento precedente, bisogna **fermare la scansione sull'elemento precedente** (e non su quello da cancellare)



## 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

```
void CancellaElementoLista(ListaDiElementi *lista, TipoElementoLista elem)
{
    ListaDiElementi prec;    /* puntatore all'elemento precedente */
    ListaDiElementi corr;    /* puntatore all'elemento corrente */
    boolean trovato;        /* usato per terminare la scansione */

    if (*lista != NULL)
        if ((*lista)->info==elem)
            { /* cancella il primo elemento */
                CancellaPrimo(lista);
            }
        else /* scansione della lista e cancellazione dell'elemento */
            prec = *lista; corr = prec->next; trovato = false;
            while (corr != NULL && !trovato)
                if (corr->info == elem)
                    { /* cancella l'elemento */
                        trovato = true; /* provoca l'uscita dal ciclo */
                        prec->next = corr->next;
                        free(corr); }
                    else {
                        prec = prec->next; /* avanzamento dei due puntatori */
                        corr = corr->next; }
}
}
```

## Versione ricorsiva:

```
void CancellaElementoLista(ListaDiElementi *lista, TipoElementoLista elem)
{
    if (*lista != NULL)
        if ((*lista)->info== elem)
            { /* cancella il primo elemento */
                CancellaPrimo(lista);
            }
        else /* cancella elem dal resto */
            CancellaElementoLista(&((*lista)->next), elem);
}
```



## Cancellazione di tutte le occorrenze di un elemento

### Versione iterativa

- ▶ analoga alla cancellazione della prima occorrenza

## Cancellazione di tutte le occorrenze di un elemento

### Versione iterativa

- ▶ analoga alla cancellazione della prima occorrenza
- ▶ però, dopo aver trovato e cancellato l'elemento, bisogna continuare la scansione

## Cancellazione di tutte le occorrenze di un elemento

### Versione iterativa

- ▶ analoga alla cancellazione della prima occorrenza
- ▶ però, dopo aver trovato e cancellato l'elemento, bisogna continuare la scansione
- ▶ ci si ferma solo quando si è arrivati alla fine della lista

## Cancellazione di tutte le occorrenze di un elemento

### Versione iterativa

- ▶ analoga alla cancellazione della prima occorrenza
- ▶ però, dopo aver trovato e cancellato l'elemento, bisogna continuare la scansione
- ▶ ci si ferma solo quando si è arrivati alla fine della lista  
⇒ non serve la sentinella booleana per fermare la scansione

## Cancellazione di tutte le occorrenze di un elemento

### Caratterizzazione induttiva

Sia *ris* la lista ottenuta cancellando tutte le occorrenze di *elem* da *lista*.  
Allora:

- ▶ se *lista* è la lista vuota, allora *ris* è la lista vuota (caso base)

## Cancellazione di tutte le occorrenze di un elemento

### Caratterizzazione induttiva

Sia *ris* la lista ottenuta cancellando tutte le occorrenze di *elem* da *lista*.

Allora:

- ▶ 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 di un elemento

### Caratterizzazione induttiva

Sia *ris* la lista ottenuta cancellando tutte le occorrenze di *elem* da *lista*. Allora:

- ▶ 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)

## Versione iterativa

```

void CancellaTuttiLista(ListaDiElementi *lista, int elem)
{
    ListaDiElementi prec;    /* puntatore all'elemento precedente */
    ListaDiElementi corr;    /* puntatore all'elemento corrente */
    boolean trovato = false;
    while ((*lista != NULL) && !trovato) /* cancella le occorrenze */
        if ((*lista)->info!=elem)      /* di elem in testa      */
            trovato = true;
        else CancellaPrimo(lista);

    if (*lista != NULL)
        {
            prec = *lista; corr = prec->next;
            while (corr != NULL)
                if (corr->info == elem)
                    { /* cancella l'elemento */
                        prec->next = corr->next;
                        free(corr);
                        corr = prec->next;}
                else {
                    prec = prec->next; /* avanzamento dei due puntatori */
                    corr = corr->next; }
        }
}

```



## Versione ricorsiva

```
void CancellaTuttiLista(ListaDiElementi *lista, int elem)
{
    ListaDiElementi aux;

    if (*lista != NULL)
        if ((*lista)->info==elem)
            {
                CancellaPrimo(lista);                /* cancellazione del primo elemento */
                CancellaTuttiLista(list, elem);      /* cancellazione di elem dal resto della lista */
            }
        else
            CancellaTuttiLista(&((*lista)->next), elem);
}
```