Tipi user-defined

- Il C mette a disposizione un insieme di tipi di dato predefiniti (tipi built-in) e dei meccanismi per definire nuovi tipi (tipi user-defined)
- Vediamo le regole generali che governano la definizione di nuovi tipi e quindi i costrutti linguistici (costruttori) che il C mette a disposizione.
- Tutti i tipi non predefiniti utilizzati in un programma devono essere dichiarati come ogni altro elemento del programma. Una dichiarazione di tipo viene fatta di solito nella parte dichiarativa del programma.
  - parte dichiarativa globale:
    - dichiarazioni di costanti
    - dichiarazioni di tipi
    - dichiarazioni di variabili
    - prototipi di funzioni/procedure

Dichiarazione di tipo

- Una dichiarazione di tipo (type declaration) consiste nella parola chiave typedef seguita da:
  - la rappresentazione o costruzione del nuovo tipo (ovvero la specifica di come è costruito a partire dai tipi già esistenti)
  - il nome del nuovo tipo
  - il simbolo ; che chiude la dichiarazione

  Esempio: `typedef int anno;`
- Una volta definito e nominato un nuovo tipo, è possibile utilizzarlo per dichiarare nuovi oggetti (ad es. variabili) di quel tipo.

  Esempio:

  ```
  float x;
  anno a;
  ```
- **Nota:** In C si possono anche definire tipi senza usare `typedef`. Quest'ultima consente l'associazione di un nome (identificatore) a un nuovo tipo. Per uniformità e leggibilità del codice useremo spesso `typedef` per definire nuovi tipi.

Tipi semplici user-defined

Ridefinizione: Un nuovo tipo può essere definito rinominando un tipo già esistente (cioè creandone un alias)

typedef TipoEsistente NuovoTipo;

dove `TipoEsistente` può essere un tipo built-in o user-defined.

Esempio:

```
typedef int anno;
typedef int naturale;

typedef char carattere;
```

Enumerazione: Consente di definire un nuovo tipo enumerando i suoi valori, con la seguente sintassi

typedef enum {v1, v2, ... , vk} NuovoTipo;

Esempio:

```
typedef enum {lun, mar, mer, gio, ven, sab, dom} Giorno;
typedef enum {gen, feb, mar, apr, mag, giu,
              lug, ago, set, ott, nov, dic} Mese;

typedef enum {m, f} sesso;
```


- I valori elencati nella definizione di un nuovo tipo enumerato, sono identificatori che rappresentano costanti di quel tipo (esattamente come `0, 1, 2, ...` sono costanti del tipo `int`, o `'a', 'b', ...` sono costanti del tipo `char`).
- Dunque, se dichiariamo una variabile
  `Giorno g;`
  possiamo scrivere l'assegnamento
  `g = mar;`
- Le costanti dei tipi enumerati non vanno racchiuse tra virgolette o tra apici!

  N.B. Il compilatore associa ai nomi utilizzati per denotare le costanti dei tipi enumerati valori naturali progressivi.

  Esempio: il valore associato a `g` dopo l'assegnamento `g=mar` è il numero naturale (intero) `1`.

  $\Longrightarrow$ mancanza di astrazione: è possibile fare riferimento alla rappresentazione dei valori.

- La relazione tra interi e tipi enumerati consente di applicare a questi ultimi le seguenti operazioni:
  - operazioni aritmetiche: `+,-,*,/,%`
  - uguaglianza e disuguaglianza: `=, !=`
  - confronto: `<,<=,>,>=`
- Si noti che la relazione di precedenza tra i valori (che determina l'esito delle operazioni di confronto) dipende dall'ordine in cui vengono elencati i valori del tipo al momento della sua definzione.

  Esempio: Con le dichiarazioni viste in precedenza `lun < gio` è vero (un intero diverso da `0`) `apr <= feb` è falso (il valore intero `0`)
- Il C tratta questi tipi come ridefinizione di `int`

Tipi fai da te: i booleani

Soluzione 1

```
#define FALSE 0;
#define TRUE 1;...
typedef int Boolean;
Boolean b;
...
```

Soluzione 2

```
typedef enum {FALSE, TRUE} Boolean;...
Boolean b;
...
```

N.B. I valori vanno elencati come sopra, rispettando la convenzione adottata dal `C`: il valore `0` rappresenta falso.

Esempio:

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

boolean even (int n)
{
if (n % 2 == 0)
    return true;
else
    return false;
}


boolean implies (boolean p, boolean q)
{
    if (p)
        return q;
    else
        return true;
}
```


Esempio: Uso del costrutto `switch` con tipi enumerati

```
typedef enum {lun, mar, mer, gio, ven, sab, dom} Giorno;
Giorno g;
...
switch (g) {
case lun: case mar: case mer: case gio: case ven:
       printf("Giorno lavorativo");
       break;
case sab: case dom:
       printf("Week-end");
       break;
}


void stampaGiorno(Giorno g) {
switch (g) {
case lun: printf("lun");
          break;
...

case dom: printf("dom");
          break;
}
```

Tipi strutturati user-defined

- Il C non possiede tipi strutturati built-in, ma fornisce dei costruttori che permettono di definire tipi strutturati anche piuttosto complessi.
- Array e puntatori possono essere visti come costruttori di tipo (definiscono un tipo di dato non semplice a partire da tipi esistenti).

Uso di `typedef` con array e puntatori

- In generale, una dichiarazione di tipo mediante `typedef` ha la forma di una dichiarazione di variabile preceduta dalla parola chiave `typedef`, e con il nome di tipo al posto del nome della variabile.
- Nel caso di array e puntatori:

```
typedef TipoElemento TipoArray[Dimensione];
typedef TipoPuntato *TipoPuntatore;
```

Esempio:

```
typedef int ArrayDieciInteri[10];
typedef int MatriceTreXQuattro[3][4];
typedef int *PuntIntero;
ArrayDieciInteri vet;           /*  int vet[10]; */
PunIntero p;                    /*  int *p; */
MatriceTreXQuattro mat, mat1;   /*  int mat[3][4]; int mat1[3][4]; */
```


Il costruttore struct

- Una struttura è un'aggregazione di elementi che possono essere eterogenei (di tipo diverso).

Esempio:

```
struct persona  {
     char nome[15];
     char cognome[20];
     int eta;
     sesso s;   }
```

  - la parola chiave `struct` introduce la definizione della struttura
  - `persona` è l'etichetta della struttura, attribuisce un nome alla definizione della struttura
  - `nome`, `cognome`, `eta`, `s` sono detti campi della struttura
- È anche possibile definire strutture con campi omogenei

```
struct complex  {
 double real;
 double imag;  }
```

Campi di una struttura

- devono avere nomi univoci all'interno di una struttura
- strutture diverse possono avere campi con lo stesso nome
- i nomi dei campi possono coincidere con altri nomi già utilizzati (es. per variabili o funzioni)

Esempio:

```
int x;
struct a { char x; int y; };
struct b { int w; float x; };
```

- possono essere di tipo diverso (semplice o altre strutture)
- un campo di una struttura non può essere del tipo struttura che si sta definendo
- un campo può però essere di tipo puntatore alla struttura

Esempio:

```
struct s { int a;
           struct s *p;  };
```


Dichiarazione di variabili di tipo struttura

- La definizione di una struttura non provoca allocazione di memoria, ma introduce un nuovo tipo di dato.

  Esempio: `struct persona tizio, docenti[10], *p;`
  - `tizio` è una variabile di tipo `struct persona`
  - `docenti` è un vettore di 10 elementi di tipo `struct persona`
  - `p` è un puntatore a una `struct persona`
  - N.B.: `persona tizio;` Errore!
- Una variabile di tipo struttura può essere dichiarata contestualmente alla definizione della struttura.

  Esempio:

```
struct studente {          |      struct  {
  char nome[20];           |        char nome[20];
  long matricola;          |        long matricola;
  struct data ddn;         |        struct data ddn;
} s1, s2;                  |      } s1, s2;
```

- In questo caso si può anche omettere l'etichetta di struttura.

Uso di typedef con strutture

- Attraverso `typedef` è possibile associare un nome ad un tipo definito mediante il costruttore `struct`.

  Esempio:

```
struct data { int giorno, mese, anno; };

typedef struct data Data;
```

- `Data` è un sinonimo di `struct data`, che può essere utilizzato nelle dichiarazioni di variabili.

```
Data d1, d2;
Data appelli[10], *pd;
```


Operazioni sulle strutture

- Si possono assegnare variabili di tipo struttura a variabili dello stesso tipo struttura.

  Esempio:

```
Data d1, d2;
...
d1 = d2;
```

- Non è possibile invece effettuare il confronto tra due variabili di tipo struttura.

  Esempio:

```
struct data d1, d2;
if (d1 == d2) ...          Errore!
```

- L'equivalenza di tipo tra strutture è per nome.

  Esempio:

```
struct s1 { int i; };
struct s2 { int i; };
struct s1 a, b;
struct s2 c;
```

`a = b;`	OK	`a` e `b` sono dello stesso tipo
`a = c;`	Errore!	`a` e `c` non sono dello stesso tipo

- Si può ottenere l'indirizzo di una variabile di tipo struttura tramite l'operatore `&`.
- Si può rilevare la dimensione di una struttura con `sizeof`.

  Esempio: `sizeof(struct data)`
- Attenzione: non è detto che la dimensione di una struttura sia pari alla somma delle dimensioni dei singoli campi.

Accesso ai campi di una struttura

- I campi di una struttura si comportano come variabili del tipo corrispondente. L'accesso avviene tramite l'operatore punto

```
Data oggi;
oggi.giorno = 11; oggi.mese = 5; oggi.anno = 2009;
printf("%d %d %d", oggi.giorno, oggi.mese, oggi.anno);
```

- Accesso tramite un puntatore alla struttura.

```
Data oggi, *pd;
pd = &oggi;
(*pd).giorno = 11; (*pd).mese = 5; (*pd).anno = 2009;
```

  N.B. Ci vogliono le **()** perché "." ha priorità più alta di "*".

- Operatore freccia: combina il dereferenziamento e l'accesso al campo della struttura.

```
pd->giorno =11; pd->mese = 5; pd->anno = 2009;
```

- N.B.: `pd->giorno` è una abbreviazione per `(*pd).giorno`.

Esempio: Accesso al campo di una struttura che è a sua volta campo di un'altra struttura.

```
struct dipendente
   { Persona datiDip;
     Data dataAssunzione;
     int stipendio;
   };
typedef struct dipendente Dipendente;

Dipendente dip, *p;
...
dip.dataAssunzione.giorno = 3;
dip.dataAssunzione.mese = 4;
dip.dataAssunzione.anno = 1997;
...
(p->dataAssunzione).giorno = 5;
(p->stipendio) = (p->stipendio) + 120;
```


Inizializzazione di strutture

- Può avvenire, come per i vettori, con un elenco di inizializzatori.

  Esempio: `Data oggi = { 11, 5, 2009 }`

- Se ci sono meno inizializzatori di campi della struttura, i campi rimanenti vengono inizializzati a 0 (o al valore speciale `NULL`, se il campo è un puntatore).

Passaggio di parametri di tipo struttura

- È come per i parametri di tipo semplice:
  - il passaggio è per valore $\Longrightarrow$ viene fatta una copia dell'intera struttura dal parametro attuale a quello formale
  - è comunque possibile simulare il passaggio per indirizzo attraverso un puntatore

Nota: per passare per valore ad una funzione un vettore (il vettore, non il puntatore al suo primo elemento) è sufficiente racchiuderlo in una struttura.

Esempio:

```
struct dipendente
   { Persona datiDip;
     Data dataAssunzione;
     int stipendio;
   };
typedef struct dipendente Dipendente;

void aumento(Dipendente *p, int percentuale)
{
    int incremento;
    incremento = (p -> stipendio) * percentuale / 100;
    p -> stipendio = p -> stipendio + incremento;
}
```


Liste

- È molto comune dover rappresentare sequenze di elementi tutti dello stesso tipo e fare operazioni su di esse.

  Esempi: sequenza di interi (23 46 5 28 3)
  sequenza di caratteri ('x' 'r' 'f')
  sequenza di persone con nome e data di nascita

- Finora abbiamo usato gli array per realizzare tali strutture, nonostante ciò porti talvolta a un impiego inefficiente della memoria.
- Vediamo adesso un modo basato sull'allocazione dinamica di variabili, che ci permette di realizzare liste di elementi in maniera che la memoria fisica utilizzata corrisponda meglio a quella astratta, cioè al numero di elementi della sequenza che vogliamo rappresentare.

Diversi modi di rappresentare sequenze di elementi

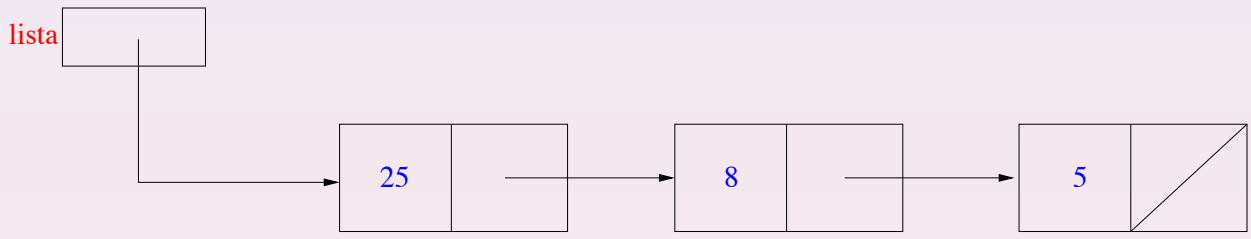
1. Rappresentazione sequenziale: tramite array
   - Vantaggi:
     - l'accesso agli elementi è diretto (tramite indice) ed efficiente
     - l'ordine degli elementi è quello in memoria $\Longrightarrow$ non servono strutture dati addizionali
     - è semplice manipolare l'intera struttura (copia, ordinamento, ...)
   - Svantaggi:
     - dobbiamo avere un'idea precisa della dimensione della sequenza
     - inserire o eliminare elementi è complicato ed inefficiente (comporta un numero di spostamenti che nel caso peggiore può essere dell'ordine del numero degli elementi della struttura)

2. Rappresentazione collegata
   - Una lista concatenata è una sequenza lineare di nodi, ciascuno dei quali memorizza un valore e contiene un riferimento (puntatore) al nodo successivo nella sequenza.
   - Per aggiungere e cancellare nodi in qualunque posizione semplicemente aggiustando il sistema di puntatori senza operare sui nodi non interessati dalla aggiunta o dalla cancellazione.
   - L'accesso agli elementi è di tipo sequenziale: per accedere al generico nodo, si deve scandire la lista, dato che l'accesso ad un elemento è possibile attraverso il puntatore contenuto nell'elemento precedente.

2. Rappresentazione collegata (continua)

- La sequenza di elementi viene rappresentata da una struttura di dati collegata, realizzata tramite strutture e puntatori.
- Ogni elemento è rappresentato con una struttura C:
  - un campo (o più campi se necessario) per l'elemento (ad es. `int`)
  - un campo puntatore alla struttura che rappresenta l'elemento successivo (ovviamente, tale struttura ha tipo indentico a quello della struttura corrente)
- L'ultimo elemento non ha un elemento successivo
  - il campo puntatore ha valore `NULL` che assume quindi il significato di "fine lista".
- L'inizio della lista è individuato da una variabile del tipo dei puntatori ai vari elementi.
  - Sarà nostra abitudine attribuire a questa variabile il nome stesso della lista, identificando il concetto di "inizio lista" (o "testa della lista") con la lista stessa.
- l'accesso a una lista avviene attraverso il puntatore al primo elemento.

Graficamente



- La variabile lista, di tipo puntatore, è utilizzata per accedere alla sequenza.

Esempio: Sequenze di interi.

```
struct EL {
  int info;
  struct EL *next;
};
typedef struct EL ElementoLista;
typedef ElementoLista *ListaDiElementi;
```

1. La prima dichiarazione `struct EL` definisce un primo campo, `info`, di tipo `int` e permette di dichiarare il campo `next` come puntatore al tipo strutturato che si sta definendo;
2. la seconda dichiarazione utilizza `typedef` per ridenominare il tipo `struct EL` come `ElementoLista`;
3. la terza dichiarazione definisce il tipo `ListaDiElementi` come puntatore al tipo `ElementoLista`.

- A questo punto possiamo definire variabili di tipo lista:

```
ListaDiElementi Lista1, Lista2;
```

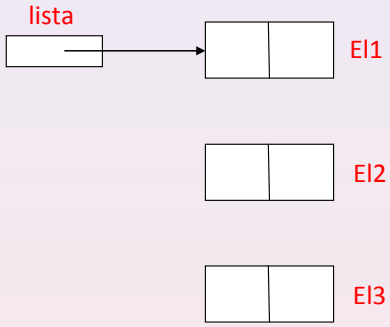

Esempio: Creazione di una lista di tre interi fissati: (8, 3, 15)

```
ElementoLista El1,El2,El3;
ListaDiElementi lista;        /* puntatore al primo elemento della lista */

lista=&El1;

El1.info = 8;
El1.next = &El2;

El2.info = 3;
El2.next = &El3;

El3.info = 15;
El3.next = NULL;
```

Nota: per poter usare la costante `NULL` dobbiamo importare il file `stdlib.h`

Esempio: Creazione di una lista di tre interi fissati: (8, 3, 15)

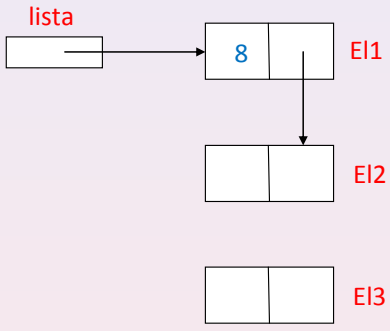
```
ElementoLista El1,El2,El3;
ListaDiElementi lista;        /* puntatore al primo elemento della lista */

lista=&El1;

El1.info = 8;
El1.next = &El2;

El2.info = 3;
El2.next = &El3;

El3.info = 15;
El3.next = NULL;
```



Nota: per poter usare la costante `NULL` dobbiamo importare il file `stdlib.h`

Esempio: Creazione di una lista di tre interi fissati: (8, 3, 15)

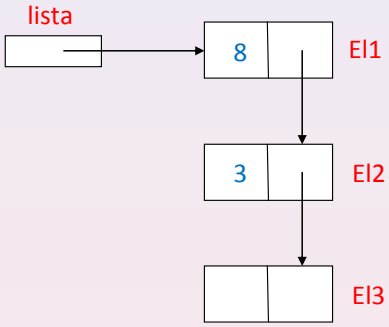
```
ElementoLista El1,El2,El3;
ListaDiElementi lista;        /* puntatore al primo elemento della lista */

lista=&El1;

El1.info = 8;
El1.next = &El2;

El2.info = 3;
El2.next = &El3;

El3.info = 15;
El3.next = NULL;
```



Nota: per poter usare la costante `NULL` dobbiamo importare il file `stdlib.h`

Esempio: Creazione di una lista di tre interi fissati: (8, 3, 15)

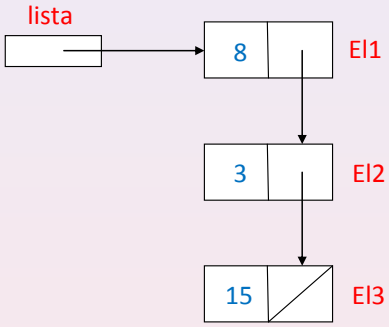
```
ElementoLista El1,El2,El3;
ListaDiElementi lista;          /* puntatore al primo elemento della lista */

lista=&El1;

El1.info = 8;
El1.next = &El2;

El2.info = 3;
El2.next = &El3;

El3.info = 15;
El3.next = NULL;
```



Nota: per poter usare la costante `NULL` dobbiamo importare il file `stdlib.h`

Esempio: Creazione di una lista di tre interi fissati: (8, 3, 15)

```
ElementoLista El1,El2,El3;
ListaDiElementi lista;          /* puntatore al primo elemento della lista */

lista=&El1;

El1.info = 8;
El1.next = &El2;

El2.info = 3;
El2.next = &El3;

El3.info = 15;
El3.next = NULL;
```



Nota: per poter usare la costante `NULL` dobbiamo importare il file `stdlib.h`

- Nel programma che abbiamo appena visto per creare una lista di tre elementi dobbiamo dichiarare tre variabili di tipo `ElementoLista`.
- Il numero degli elementi della lista deve essere deciso a tempo di compilazione.
- Non è possibile, per esempio, creare una lista con un numero di elementi letti in ingresso. Ma allora qual è il vantaggio rispetto all'array?
- Quello che abbiamo visto non è l'unico modo. . . .

Allocazione Dinamica della memoria

- L'allocazione dinamica della memoria è possibile in `C` grazie all'utilizzo di alcune funzioni messe a disposizione dalla libreria standard (standard library). Infatti è richiesta l'inclusione del file header `<stdib.h>`
- Le due funzioni principali sono
  - malloc: consente di allocare dinamicamente memoria per una variabile di un tipo specificato
  - free: consente di rilasciare dinamicamente memoria (precedentemente allocata con malloc)
- I tipi di dato sono ancora statici, ovvero hanno una dimensione fissata a priori. Le variabili di un certo tipo di dato possono invece essere create.

malloc

- La chiamata di funzione

  malloc(sizeof(TipoDato));

  crea in memoria una variabile di tipo TipoDato, e restituisce come risultato l'indirizzo della variabile creata.
- Se p è una variabile di tipo puntatore a TipoDato, l'istruzione

  p=malloc(sizeof(TipoDato));

  assegna l'indirizzo restituito dalla funzione malloc a p che punta quindi alla nuova variabile (p già esiste).
- Una variabile creata dinamicamente è necessariamente anonima: a essa si può fare riferimento solo tramite un puntatore

  a differenza di una variabile dichiarata mediante un proprio identificatore, che può essere riferita sia direttamente sia tramite un puntatore

free

- Se p è l'indirizzo di una variabile allocata dinamicamente, la chiamata

  free(p);

  rilascia lo spazio di memoria puntato da p

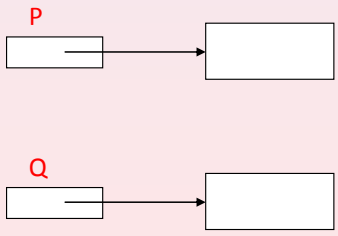
  la corrispondente memoria fisica è resa disponibile per qualsiasi altro uso.
- free deve ricevere come parametro attuale un puntatore al quale era stato assegnato come valore l'indirizzo restituito da una funzione di allocazione dinamica di memoria (cioè malloc).

Heap

- Poiché le variabili dinamiche possono essere create e distrutte in un qualsiasi punto del programma esse non possono essere allocate sullo stack.
- Vengono allocate in un'altra zona di memoria chiamata heap (mucchio). La loro gestione risulta molto più inefficiente.

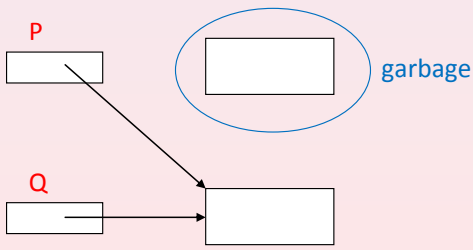
Produzione di garbage (spazzatura)

- Si verifica quando la memoria allocata dinamicamente risulta logicamente inaccessibile, e quindi sprecata, perché non esiste alcun riferimento ad essa.
  Esempio:
  ```
  P=malloc(sizeof(TipoDato));
  ...
  P=Q;
  ```
- In questo modo la cella puntata da `P` subito dopo l'assegnamento `P=Q` perde ogni possibilità di accesso (da cui il termine spazzatura).



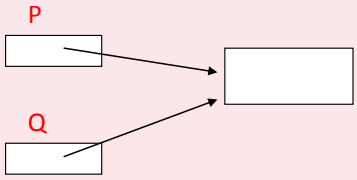
Produzione di garbage (spazzatura)

- Si verifica quando la memoria allocata dinamicamente risulta logicamente inaccessibile, e quindi sprecata, perché non esiste alcun riferimento ad essa.
  Esempio:
  ```
  P=malloc(sizeof(TipoDato));
  ...
  P=Q;
  ```
- In questo modo la cella puntata da `P` subito dopo l'assegnamento `P=Q` perde ogni possibilità di accesso (da cui il termine spazzatura).



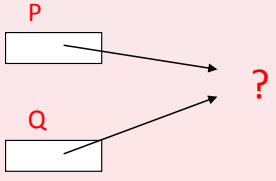
Riferimenti fluttuanti (dangling references)

- Simmetrico al problema precedente: consiste nel creare riferimenti fasulli a zone di memoria logicamente inesistenti.
  Esempio:
  ```
  P=Q;
  free(Q);
  ```
- L'operazione `free(Q)` provoca il rilascio della memoria allocata per la variabile cui `Q` punta
- `P` punta a una zona di memoria non più significativa (può essere riusata in futuro).
- `*P` comporterebbe l'accesso all'indirizzo fisico puntato da `P` e l'interpretazione del suo contenuto come un valore del tipo di `*P` con risultati imprevedibili.



Riferimenti fluttuanti (dangling references)

- Simmetrico al problema precedente: consiste nel creare riferimenti fasulli a zone di memoria logicamente inesistenti.
  Esempio:
  ```
  P=Q;
  free(Q);
  ```
- L'operazione `free(Q)` provoca il rilascio della memoria allocata per la variabile cui `Q` punta
- `P` punta a una zona di memoria non più significativa (può essere riusata in futuro).
- `*P` comporterebbe l'accesso all'indirizzo fisico puntato da `P` e l'interpretazione del suo contenuto come un valore del tipo di `*P` con risultati imprevedibili.



- Produzione di garbage e riferimenti fluttuanti hanno svantaggi simmetrici:
  - la prima comporta spreco di memoria
  - la seconda comporta risultati imprevedibili e scorretti.
- La seconda è più pericolosa della prima e in alcuni linguaggi non è prevista l'istruzione `free`.
- Viene lasciato al supporto del linguaggio l'onere di effettuare garbage collection ("raccolta rifiuti").

Esempio di allocazione dinamica

```
#include <stdio.h>
#include <stdlib.h>
main()
{
int x = 10, *P1, *P2;

P1 = malloc(sizeof(int));
*P1 = 2*x;
P2 = P1;
*P2= 3*(*P1);
printf("x=%d  *P1=%d  *P2=%d \n", x, *P1, *P2);
free(P1);
}
```

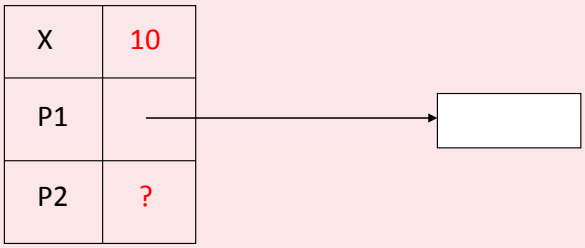
PILA HEAP

X	10
P1	?
P2	?

Esempio di allocazione dinamica

```
#include <stdio.h>
#include <stdlib.h>
main()
{
int x = 10, *P1, *P2;

P1 = malloc(sizeof(int));
*P1 = 2*x;
P2 = P1;
*P2= 3*(*P1);
printf("x=%d  *P1=%d  *P2=%d \n", x, *P1, *P2);
free(P1);
}
```

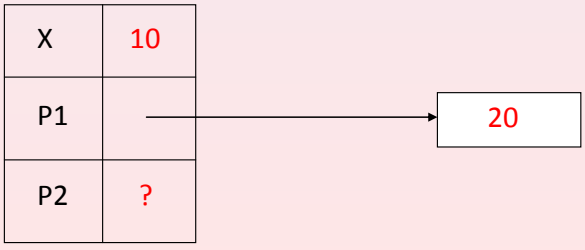
PILA HEAP



Esempio di allocazione dinamica

```
#include <stdio.h>
#include <stdlib.h>
main()
{
int x = 10, *P1, *P2;

P1 = malloc(sizeof(int));
*P1 = 2*x;
P2 = P1;
*P2= 3*(*P1);
printf("x=%d  *P1=%d  *P2=%d \n", x, *P1, *P2);
free(P1);
}
```

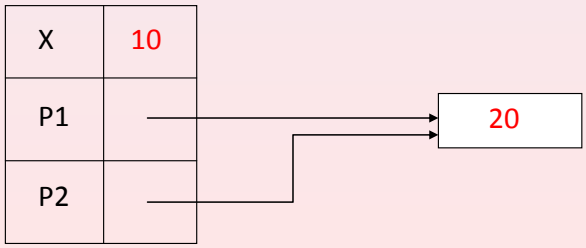
PILA HEAP



Esempio di allocazione dinamica

```
#include <stdio.h>
#include <stdlib.h>
main()
{
int x = 10, *P1, *P2;

P1 = malloc(sizeof(int));
*P1 = 2*x;
P2 = P1;
*P2= 3*(*P1);
printf("x=%d  *P1=%d  *P2=%d \n", x, *P1, *P2);
free(P1);
}
```

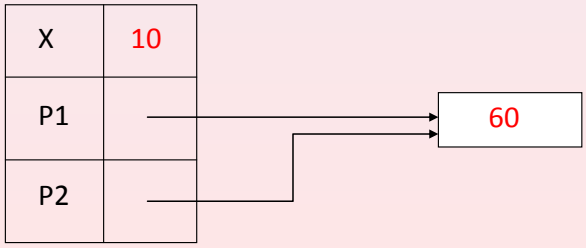
PILA HEAP



Esempio di allocazione dinamica

```
#include <stdio.h>
#include <stdlib.h>
main()
{
int x = 10, *P1, *P2;

P1 = malloc(sizeof(int));
*P1 = 2*x;
P2 = P1;
*P2= 3*(*P1);
printf("x=%d  *P1=%d  *P2=%d \n", x, *P1, *P2);
free(P1);
}
```

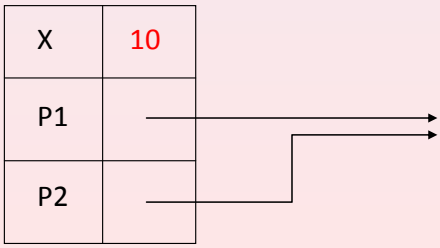
PILA HEAP



Esempio di allocazione dinamica

```
#include <stdio.h>
#include <stdlib.h>
main()
{
int x = 10, *P1, *P2;

P1 = malloc(sizeof(int));
*P1 = 2*x;
P2 = P1;
*P2= 3*(*P1);
printf("x=%d  *P1=%d  *P2=%d \n", x, *P1, *P2);
free(P1);
}
```

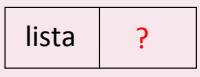
PILA HEAP



Creazione di una lista di tre interi fissati: (8, 3, 15)

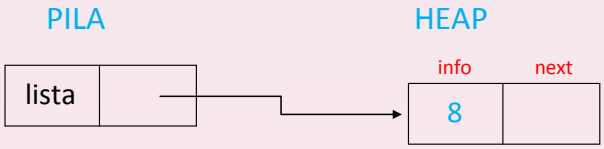
```
ListaDiElementi lista;          /* puntatore al primo elemento della lista */

lista = malloc(sizeof(ElementoLista)); /* allocazione primo elemento */

lista->info = 8;

lista->next = malloc(sizeof(ElementoLista)); /* secondo elemento */

lista->next->info = 3;
lista->next->next = malloc(sizeof(ElementoLista));       /* terzo elemento */
lista->next->next->info = 15;
lista->next->next->next = NULL;
```

PILA HEAP



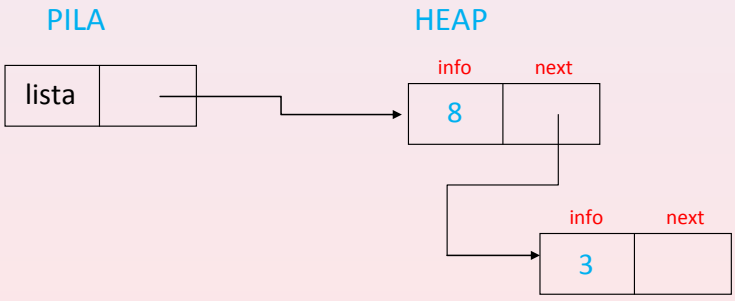
Creazione di una lista di tre interi fissati: (8, 3, 15)

```
ListaDiElementi lista;          /* puntatore al primo elemento della lista */

lista = malloc(sizeof(ElementoLista)); /* allocazione primo elemento */

lista->info = 8;

lista->next = malloc(sizeof(ElementoLista)); /* secondo elemento */

lista->next->info = 3;
lista->next->next = malloc(sizeof(ElementoLista));       /* terzo elemento */
lista->next->next->info = 15;
lista->next->next->next = NULL;
```



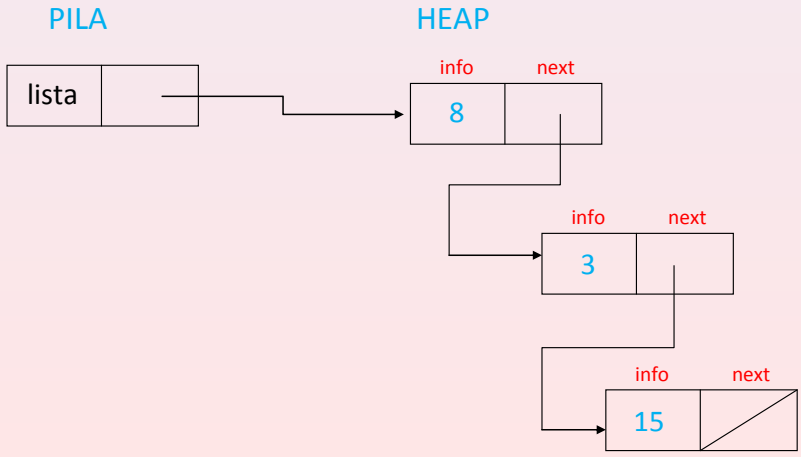
Creazione di una lista di tre interi fissati: (8, 3, 15)

```
ListaDiElementi lista;          /* puntatore al primo elemento della lista */

lista = malloc(sizeof(ElementoLista)); /* allocazione primo elemento */

lista->info = 8;

lista->next = malloc(sizeof(ElementoLista)); /* secondo elemento */

lista->next->info = 3;
lista->next->next = malloc(sizeof(ElementoLista));       /* terzo elemento */
lista->next->next->info = 15;
lista->next->next->next = NULL;
```



Creazione di una lista di tre interi fissati: (8, 3, 15)

```
ListaDiElementi lista;          /* puntatore al primo elemento della lista */

lista = malloc(sizeof(ElementoLista)); /* allocazione primo elemento */

lista->info = 8;

lista->next = malloc(sizeof(ElementoLista)); /* secondo elemento */

lista->next->info = 3;
lista->next->next = malloc(sizeof(ElementoLista));       /* terzo elemento */
lista->next->next->info = 15;
lista->next->next->next = NULL;
```

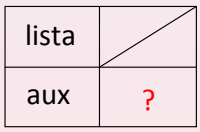


Osservazioni:

- `lista` è di tipo `ListaDiElementi`, quindi è un puntatore e non una struttura
- la zona di memoria per ogni elemento della lista (non per ogni variabile di tipo `ListaDiElementi`) deve essere allocata esplicitamente con `malloc`
- Esiste un modo più semplice di creare la lista di 3 elementi?
- Creiamo la lista a partire dal fondo!

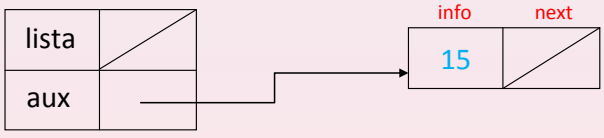
```
ListaDiElementi aux, lista = NULL;

aux = malloc(sizeof(ElementoLista));
aux->info = 15;     aux->next = lista;
lista = aux;

aux = malloc(sizeof(ElementoLista)
aux->info = 8;      aux->next = lista;
lista = aux;

aux = malloc(sizeof(ElementoLista));
aux->info = 3;      aux->next = lista;
lista = aux;
```

PILA HEAP

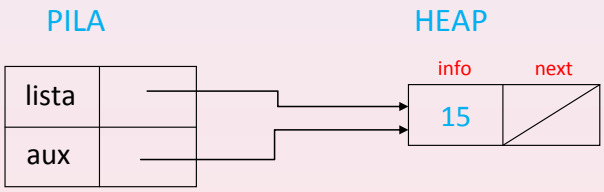



```
ListaDiElementi aux, lista = NULL;

aux = malloc(sizeof(ElementoLista));
aux->info = 15;     aux->next = lista;
lista = aux;

aux = malloc(sizeof(ElementoLista)
aux->info = 8;      aux->next = lista;
lista = aux;

aux = malloc(sizeof(ElementoLista));
aux->info = 3;      aux->next = lista;
lista = aux;
```

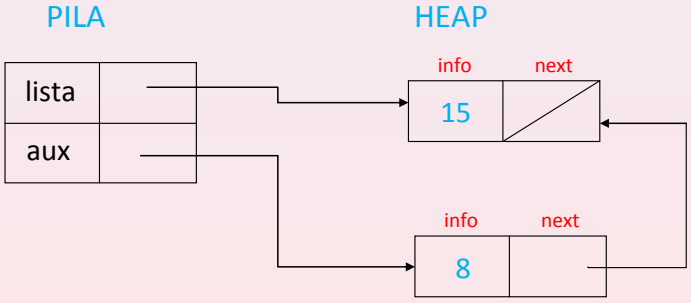
PILA HEAP



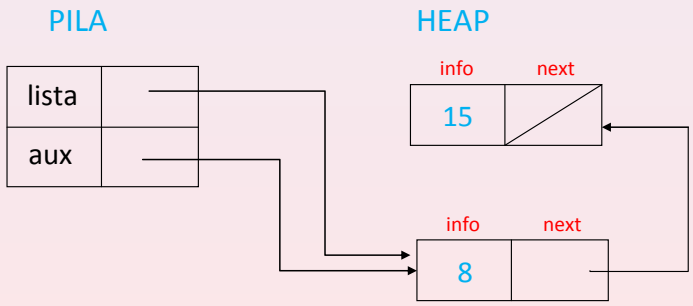
```
ListaDiElementi aux, lista = NULL;

aux = malloc(sizeof(ElementoLista));
aux->info = 15;     aux->next = lista;
lista = aux;

aux = malloc(sizeof(ElementoLista)
aux->info = 8;      aux->next = lista;
lista = aux;

aux = malloc(sizeof(ElementoLista));
aux->info = 3;      aux->next = lista;
lista = aux;
```



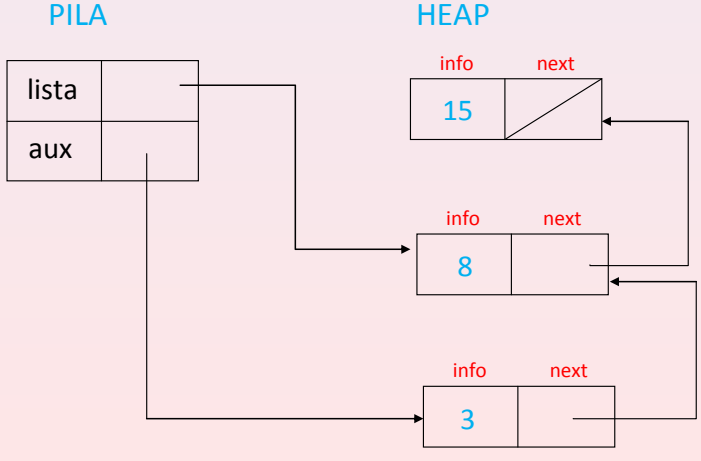

```
ListaDiElementi aux, lista = NULL;

aux = malloc(sizeof(ElementoLista));
aux->info = 15;     aux->next = lista;
lista = aux;

aux = malloc(sizeof(ElementoLista)
aux->info = 8;      aux->next = lista;
lista = aux;

aux = malloc(sizeof(ElementoLista));
aux->info = 3;      aux->next = lista;
lista = aux;
```



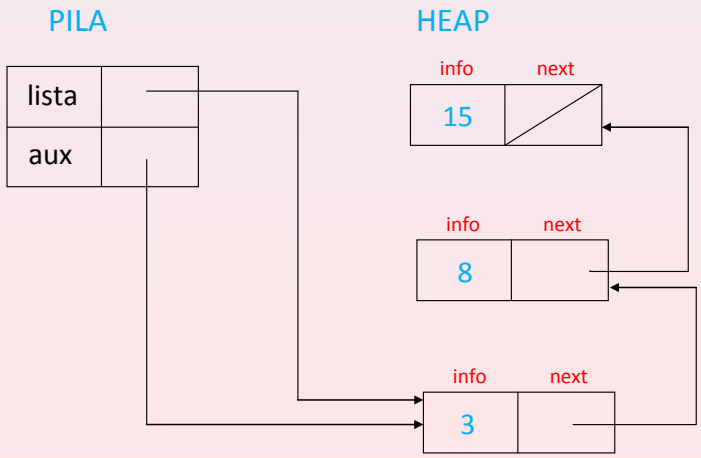
```
ListaDiElementi aux, lista = NULL;

aux = malloc(sizeof(ElementoLista));
aux->info = 15;     aux->next = lista;
lista = aux;

aux = malloc(sizeof(ElementoLista)
aux->info = 8;      aux->next = lista;
lista = aux;

aux = malloc(sizeof(ElementoLista));
aux->info = 3;      aux->next = lista;
lista = aux;
```




```
ListaDiElementi aux, lista = NULL;

aux = malloc(sizeof(ElementoLista));
aux->info = 15;     aux->next = lista;
lista = aux;

aux = malloc(sizeof(ElementoLista)
aux->info = 8;      aux->next = lista;
lista = aux;

aux = malloc(sizeof(ElementoLista));
aux->info = 3;      aux->next = lista;
lista = aux;
```

```
ListaDiElementi aux, lista = NULL;

aux = malloc(sizeof(ElementoLista));
aux->info = 15;    aux->next = lista;
lista = aux;

aux = malloc(sizeof(ElementoLista)
aux->info = 8;     aux->next = lista;
lista = aux;

aux = malloc(sizeof(ElementoLista));
aux->info = 3;     aux->next = lista;
lista = aux;
```



Operazioni sulle liste

- Definiamo una serie di procedure e funzioni per operare sulle liste.
- Usiamo liste di interi per semplicità, ma tutte le operazioni sono realizzabili in modo del tutto analogo su liste di altro tipo (salvo rare eccezioni)
- Facciamo riferimento alle dichiarazioni dei tipi `ElementoLista` e `ListaDiElementi` viste in precedenza

Inizializzazione

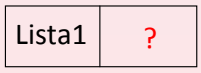
- Definiamo una procedura che inizializza una lista assegnando il valore `NULL` alla variabile testa della lista.
- Tale variabile deve essere modificata e quindi passata per indirizzo.
- Ciò provoca, nell'intestazione della procedura, la presenza di un puntatore a puntatore.

```
void Inizializza(ListaDiElementi *lista)
{
 *lista=NULL;
}
```

- Supponiamo ora che `Inizializza` sia chiamata passando come parametro l'indirizzo della variabile `Lista1` di tipo `ListaDiElementi`, ad esempio:

```
ListaDiElementi Lista1;
Inizializza(&Lista1);
```

PILA

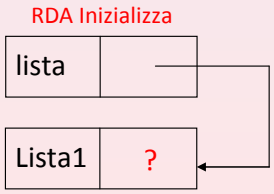



```
void Inizializza(ListaDiElementi *lista)
{
 *lista=NULL;
}
```

- Supponiamo ora che `Inizializza` sia chiamata passando come parametro l'indirizzo della variabile `Lista1` di tipo `ListaDiElementi`, ad esempio:

```
ListaDiElementi Lista1;
Inizializza(&Lista1);
```


PILA

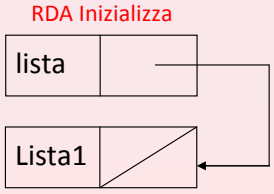


```
void Inizializza(ListaDiElementi *lista)
{
 *lista=NULL;
}
```

- Supponiamo ora che `Inizializza` sia chiamata passando come parametro l'indirizzo della variabile `Lista1` di tipo `ListaDiElementi`, ad esempio:

```
ListaDiElementi Lista1;
Inizializza(&Lista1);
```

PILA

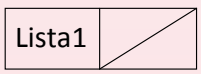



```
void Inizializza(ListaDiElementi *lista)
{
 *lista=NULL;
}
```

- Supponiamo ora che `Inizializza` sia chiamata passando come parametro l'indirizzo della variabile `Lista1` di tipo `ListaDiElementi`, ad esempio:

```
ListaDiElementi Lista1;
Inizializza(&Lista1);
```


PILA



Cosa succederebbe se passassimo il parametro per valore?

```
void Inizializza(ListaDiElementi lista)
{
  lista=NULL;
}

main() {
   ListaDiElementi Lista1;
   Inizializza(Lista1);
   ...
}
```

PILA

Lista1	?

Cosa succederebbe se passassimo il parametro per valore?

```
void Inizializza(ListaDiElementi lista)
{
  lista=NULL;
}

main() {
   ListaDiElementi Lista1;
   Inizializza(Lista1);
   ...
}
```

PILA

RDA Inizializza

lista	?

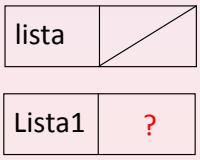
Lista1	?

Cosa succederebbe se passassimo il parametro per valore?

```
void Inizializza(ListaDiElementi lista)
{
  lista=NULL;
}

main() {
   ListaDiElementi Lista1;
   Inizializza(Lista1);
   ...
}
```

PILA

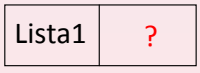
RDA Inizializza



Cosa succederebbe se passassimo il parametro per valore?

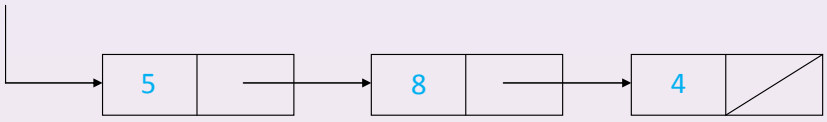
```
void Inizializza(ListaDiElementi lista)
{
  lista=NULL;
}

main() {
   ListaDiElementi Lista1;
   Inizializza(Lista1);
   ...
}
```

PILA



Stampa degli elementi di una lista

- Data la lista



vogliamo che venga stampato:

5 –> 8 –> 4 –> //

Versione iterativa:

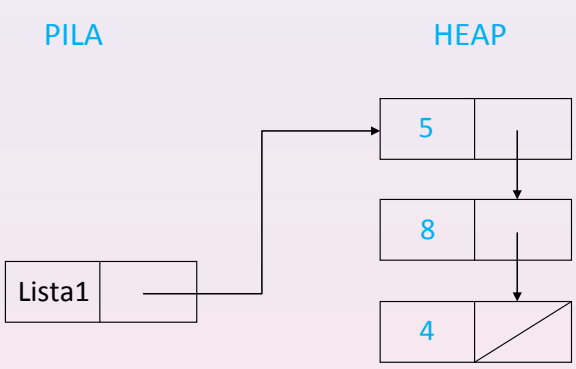
```
void StampaLista(ListaDiElementi lis)
{
   while (lis != NULL)
    {
     printf("%d -->", lis->info);
     lis = lis->next;
    }
printf("//");
}
```

N.B.: `lis = lis->next` fa puntare `lis` all'elemento successivo della listaAttenzione: Possiamo usare `lis` per scorrere la lista perché, avendo utilizzato il passaggio per valore, le modifiche a `lis` non si ripercuotono sul parametro attuale.

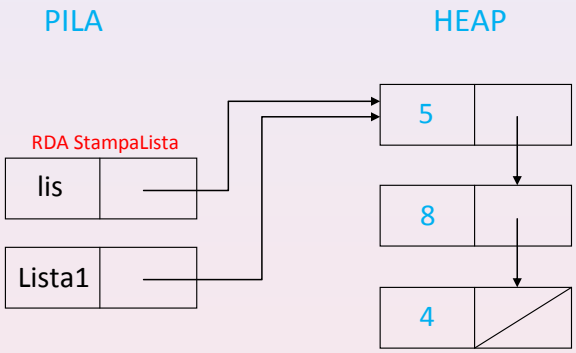
```
void StampaLista(ListaDiElementi lis)
{
  while (lis != NULL)
    {
      printf("%d -->", lis->info);
      lis = lis->next;
     }
  printf("//");
}

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



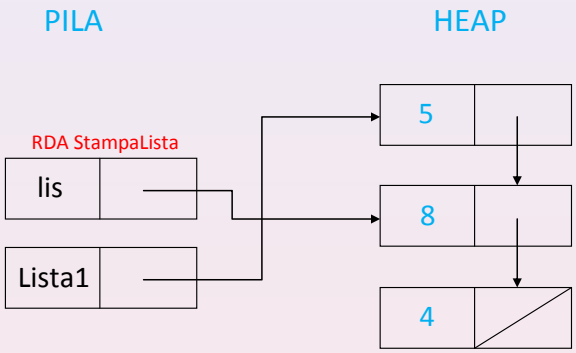
```
void StampaLista(ListaDiElementi lis)
{
  while (lis != NULL)
    {
      printf("%d -->", lis->info);
      lis = lis->next;
     }
  printf("//");
}

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



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

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



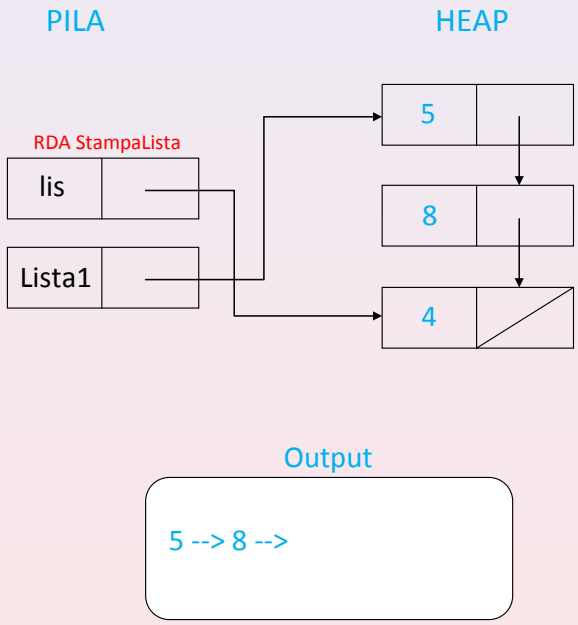

```
void StampaLista(ListaDiElementi lis)
{
  while (lis != NULL)
    {
      printf("%d -->", lis->info);
      lis = lis->next;
     }
  printf("//");
}

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



Output

5 -->

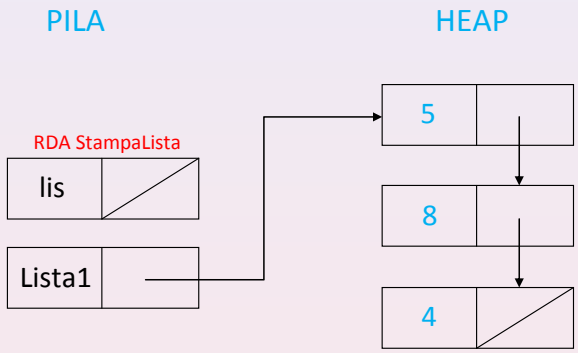
```
void StampaLista(ListaDiElementi lis)
{
  while (lis != NULL)
    {
      printf("%d -->", lis->info);
      lis = lis->next;
     }
  printf("//");
}

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



Output

5 --> 8 -->

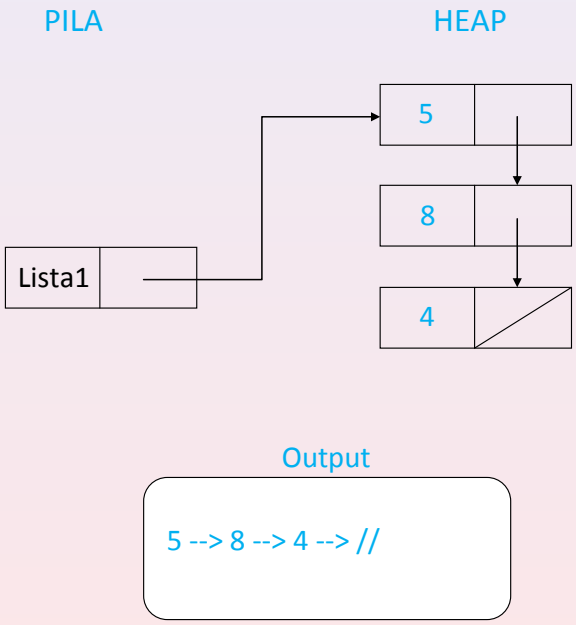

```
void StampaLista(ListaDiElementi lis)
{
  while (lis != NULL)
    {
      printf("%d -->", lis->info);
      lis = lis->next;
     }
  printf("//");
}

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



Output

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

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

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

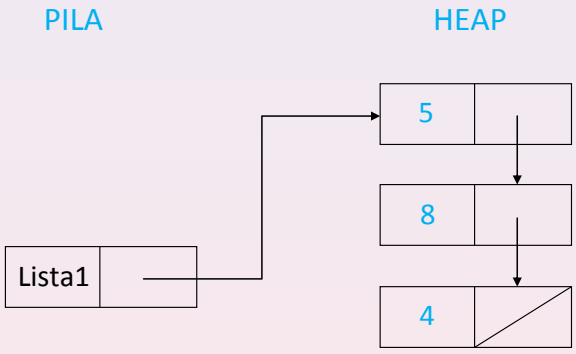


Cosa sarebbe successo passando il parametro per indirizzo?

```
void StampaLista(ListaDiElementi *lis)
{
  while (*lis != NULL)
    {
      printf("%d -->", *lis->info);
      *lis = *lis->next;
     }
  printf("//");
}

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

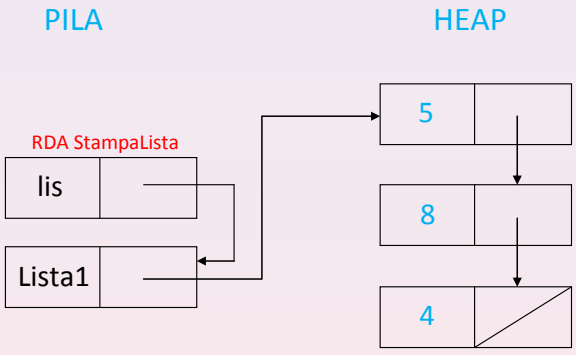
Cosa sarebbe successo passando il parametro per indirizzo?

```
void StampaLista(ListaDiElementi *lis)
{
  while (*lis != NULL)
    {
      printf("%d -->", *lis->info);
      *lis = *lis->next;
     }
  printf("//");
}

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



Cosa sarebbe successo passando il parametro per indirizzo?

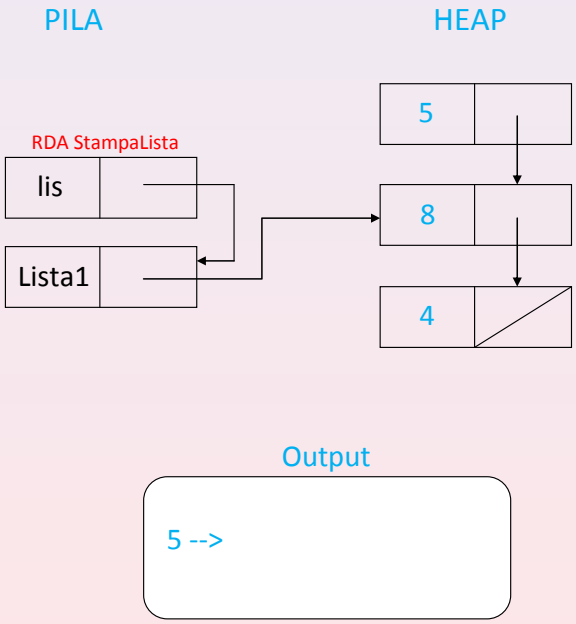
```
void StampaLista(ListaDiElementi *lis)
{
  while (*lis != NULL)
    {
      printf("%d -->", *lis->info);
      *lis = *lis->next;
     }
  printf("//");
}

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



Output

Cosa sarebbe successo passando il parametro per indirizzo?

```
void StampaLista(ListaDiElementi *lis)
{
  while (*lis != NULL)
    {
      printf("%d -->", *lis->info);
      *lis = *lis->next;
     }
  printf("//");
}

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

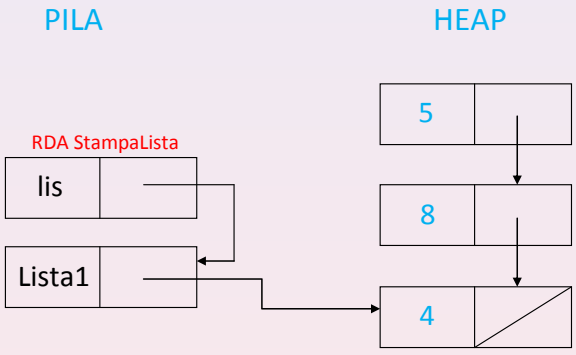


Output

5 -->

Cosa sarebbe successo passando il parametro per indirizzo?

```
void StampaLista(ListaDiElementi *lis)
{
  while (*lis != NULL)
    {
      printf("%d -->", *lis->info);
      *lis = *lis->next;
     }
  printf("//");
}

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

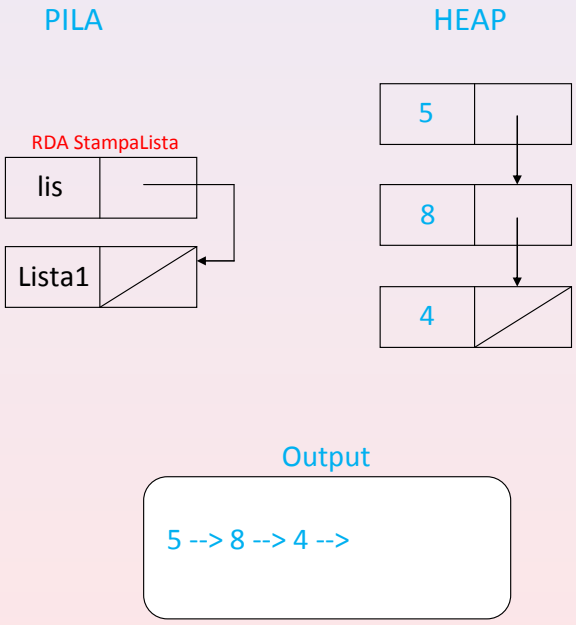


Output

5 --> 8 -->

Cosa sarebbe successo passando il parametro per indirizzo?

```
void StampaLista(ListaDiElementi *lis)
{
  while (*lis != NULL)
    {
      printf("%d -->", *lis->info);
      *lis = *lis->next;
     }
  printf("//");
}

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

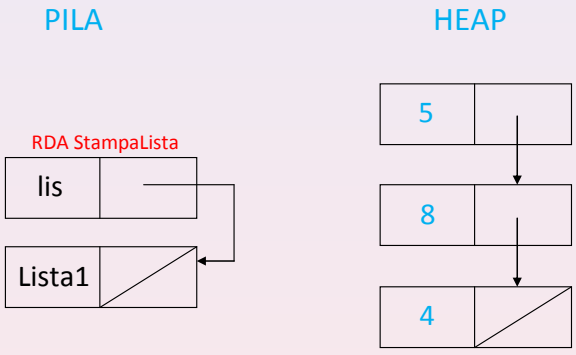


Output

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

Cosa sarebbe successo passando il parametro per indirizzo?

```
void StampaLista(ListaDiElementi *lis)
{
  while (*lis != NULL)
    {
      printf("%d -->", *lis->info);
      *lis = *lis->next;
     }
  printf("//");
}

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

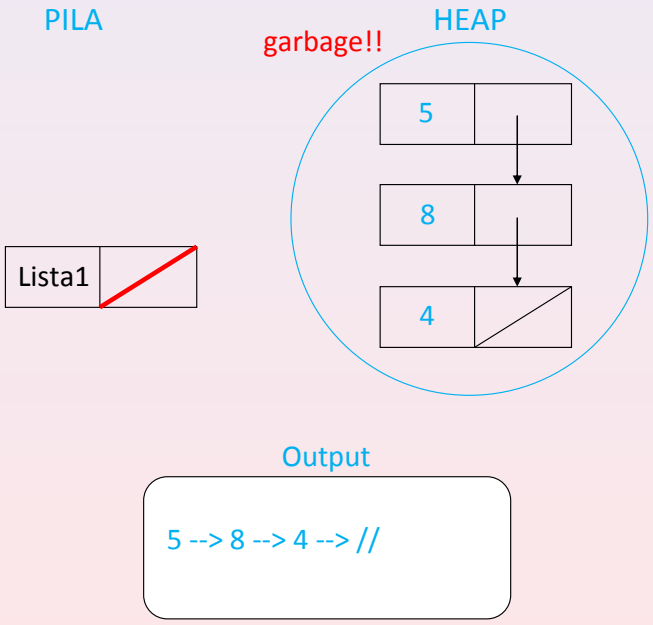


Output

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

Cosa sarebbe successo passando il parametro per indirizzo?

```
void StampaLista(ListaDiElementi *lis)
{
  while (*lis != NULL)
    {
      printf("%d -->", *lis->info);
      *lis = *lis->next;
     }
  printf("//");
}

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

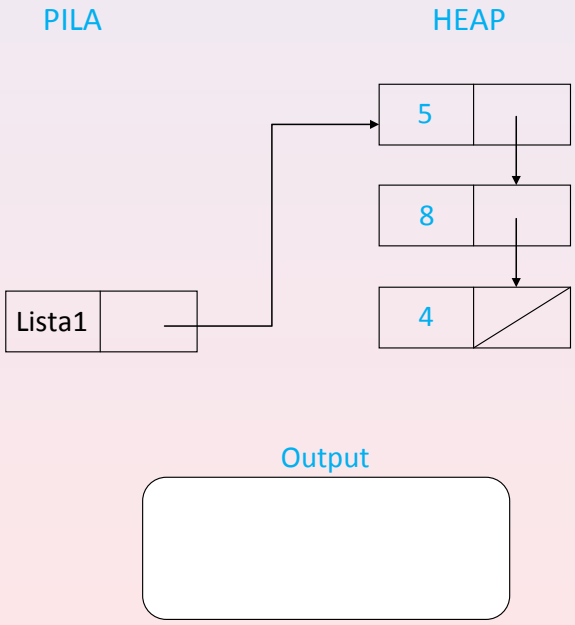


Versione ricorsiva

```
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);
  ...
}
```

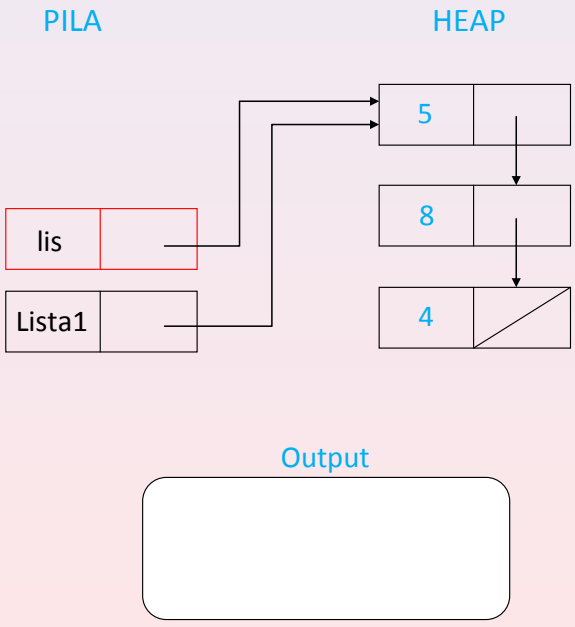
Versione ricorsiva

```
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);
  ...
}
```



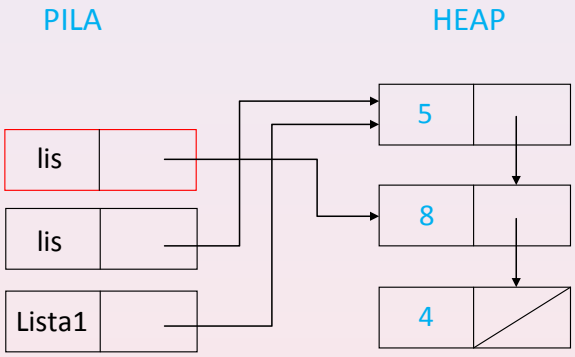
Versione ricorsiva

```
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);
  ...
}
```



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);
  ...
}
```

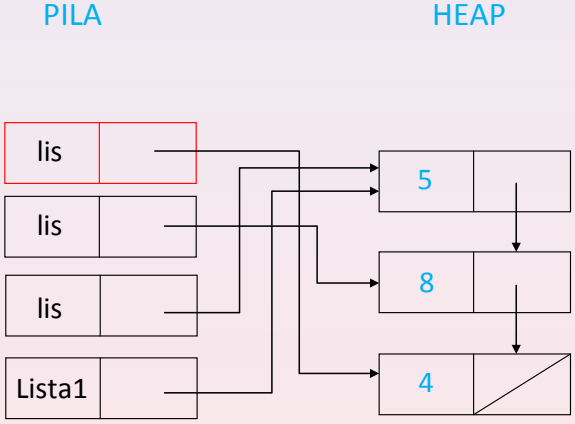


Output

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);
  ...
}
```

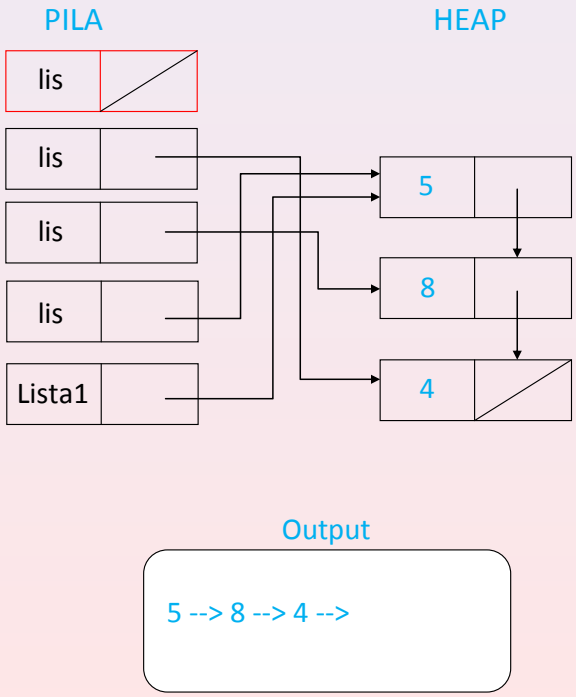


Output

5 --> 8 -->

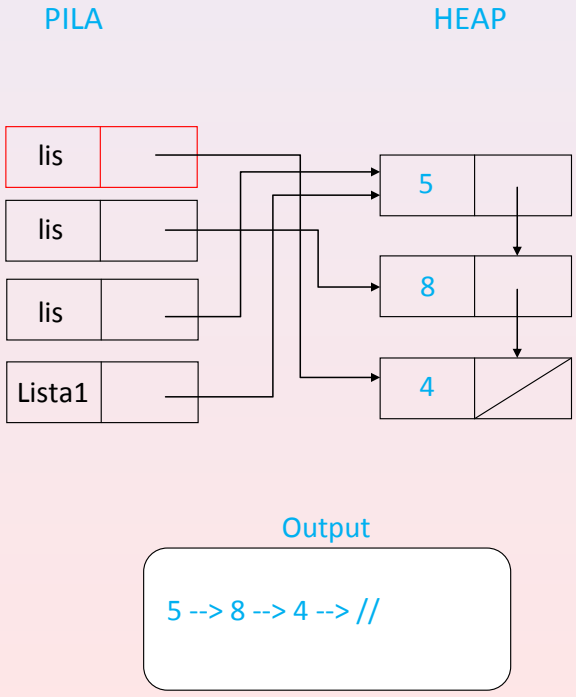
Versione ricorsiva

```
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

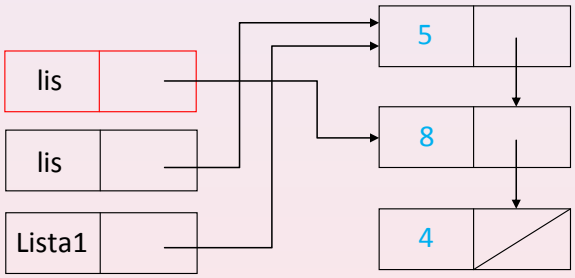
```
void StampaListaRic(ListaDiElementi lis)
{
   if (lis != NULL)
   {
      printf("%d ", lis->info);
      StampaListaRic(lis->next);
   }
   else
      printf("//");
}

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);
  ...
}
```

PILA HEAP



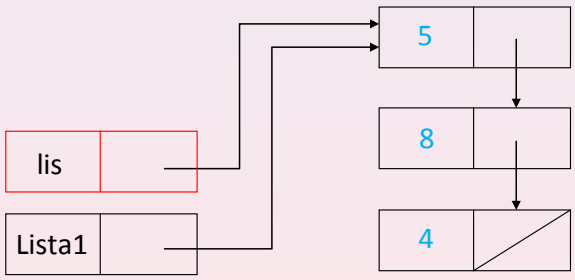
Output

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

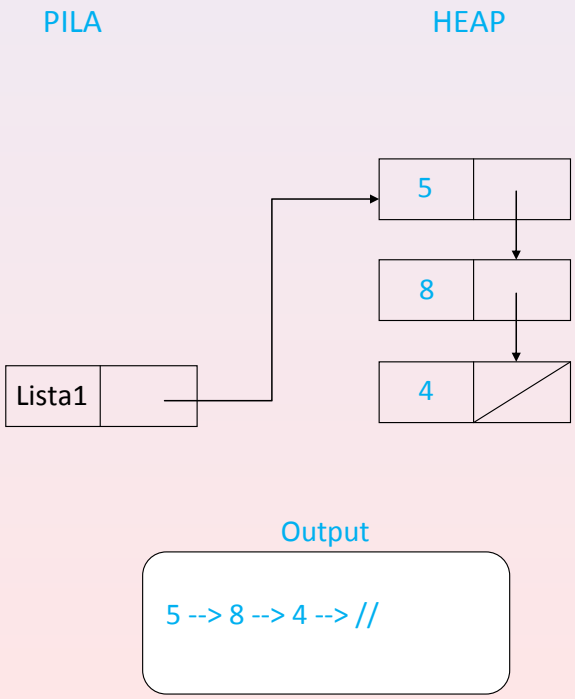


Output

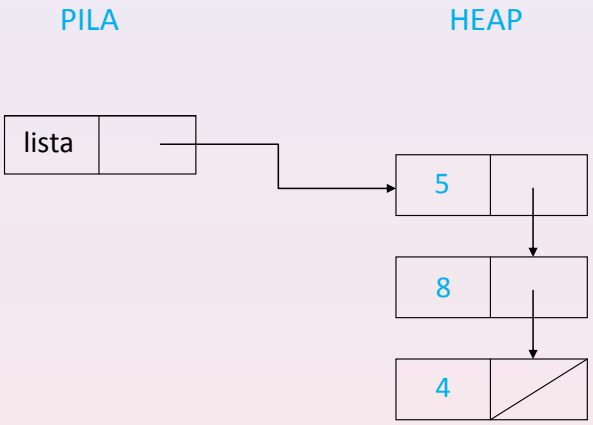
5 --> 8 --> 4 --> //

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);
  ...
}
```

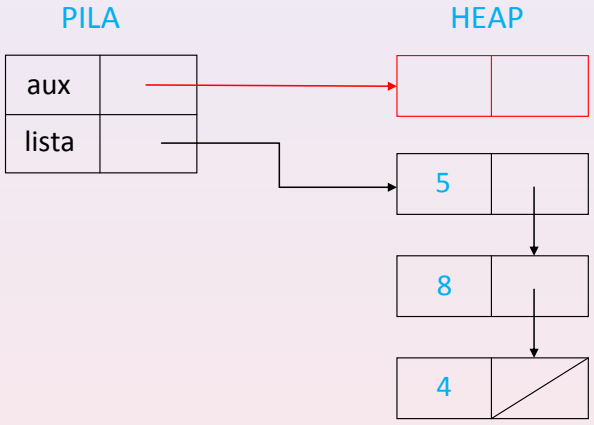


Inserimento di un nuovo elemento in testa



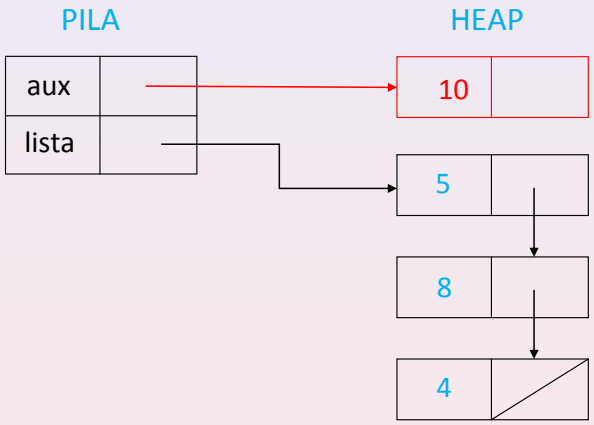
1. allochiamo una nuova struttura per l'elemento (`malloc`)
2. assegnamo il valore da inserire al campo `info` della struttura
3. concateniamo la nuova struttura con la vecchia lista
4. il puntatore iniziale della lista viene fatto puntare alla nuova struttura
   $\Longrightarrow$ la lista da modificare deve essere passata per indirizzo

Inserimento di un nuovo elemento in testa



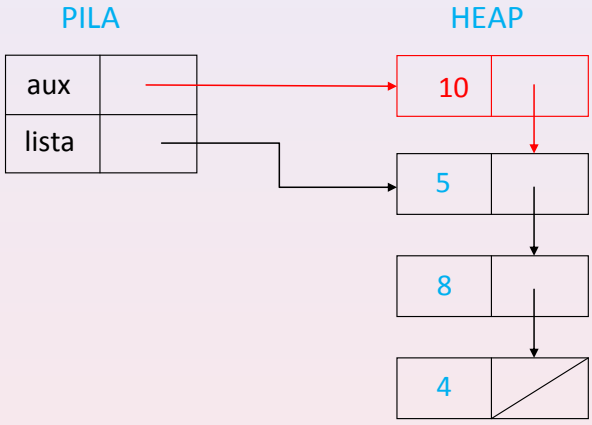
1. allochiamo una nuova struttura per l'elemento (`malloc`)
2. assegnamo il valore da inserire al campo `info` della struttura
3. concateniamo la nuova struttura con la vecchia lista
4. il puntatore iniziale della lista viene fatto puntare alla nuova struttura
   $\Longrightarrow$ la lista da modificare deve essere passata per indirizzo

Inserimento di un nuovo elemento in testa



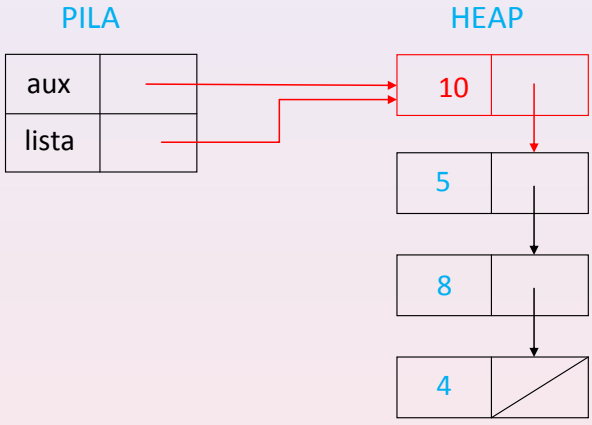
1. allochiamo una nuova struttura per l'elemento (`malloc`)
2. assegnamo il valore da inserire al campo `info` della struttura
3. concateniamo la nuova struttura con la vecchia lista
4. il puntatore iniziale della lista viene fatto puntare alla nuova struttura
   $\Longrightarrow$ la lista da modificare deve essere passata per indirizzo

Inserimento di un nuovo elemento in testa



1. allochiamo una nuova struttura per l'elemento (`malloc`)
2. assegnamo il valore da inserire al campo `info` della struttura
3. concateniamo la nuova struttura con la vecchia lista
4. il puntatore iniziale della lista viene fatto puntare alla nuova struttura
   $\Longrightarrow$ la lista da modificare deve essere passata per indirizzo

Inserimento di un nuovo elemento in testa



1. allochiamo una nuova struttura per l'elemento (`malloc`)
2. assegnamo il valore da inserire al campo `info` della struttura
3. concateniamo la nuova struttura con la vecchia lista
4. il puntatore iniziale della lista viene fatto puntare alla nuova struttura
   $\Longrightarrow$ la lista da modificare deve essere passata per indirizzo

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

  aux = malloc(sizeof(ElementoLista));
  aux->info = elem;
  aux->next = *lista;
  *lista = aux;
}
```

- il primo parametro è la lista da modificare (passata per indirizzo)
- il secondo parametro è l'elemento da inserire (passato per indirizzo)
  - Attenzione: nel caso di liste di tipo `TipoElemLista` la procedura può essere generalizzata se su tale tipo è definito l'assegnamento

Esercizio

Impostare una chiamata alla procedura e tracciare l'evoluzione di pila e heap

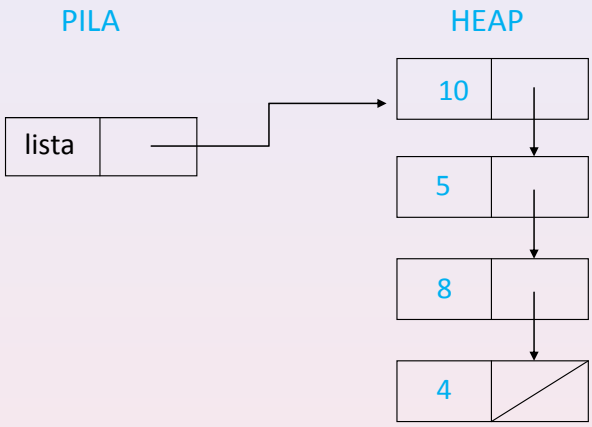

```
void InserisciTestaLista(ListaDiElementi *lista, TipoElemLista elem)
{
  ListaDiElementi aux;

  aux = malloc(sizeof(ElementoLista));
  aux->info = elem;
  aux->next = *lista;
  *lista = aux;
}
```

- il primo parametro è la lista da modificare (passata per indirizzo)
- il secondo parametro è l'elemento da inserire (passato per indirizzo)
  - Attenzione: nel caso di liste di tipo `TipoElemLista` la procedura può essere generalizzata se su tale tipo è definito l'assegnamento

Esercizio

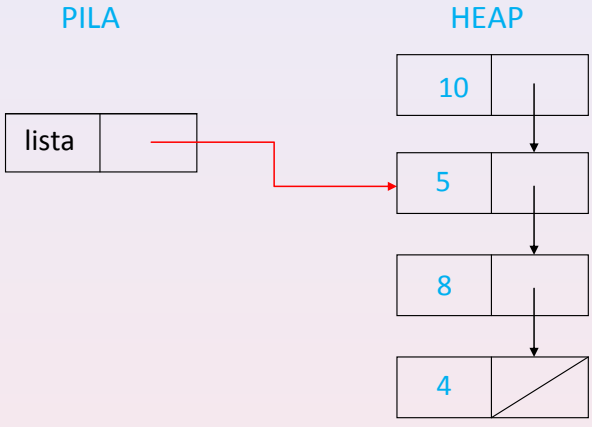
Impostare una chiamata alla procedura e tracciare l'evoluzione di pila e heap

Cancellazione del primo elemento



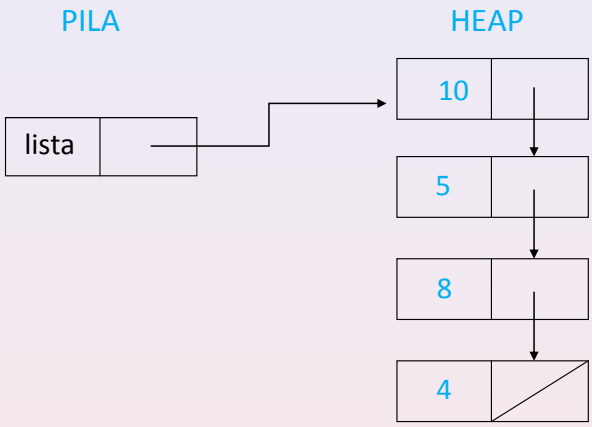
- se la lista è vuota non facciamo nulla
- altrimenti eliminiamo il primo elemento
  $\Longrightarrow$ la lista deve essere passata per indirizzo
- cancellare significa anche deallocare la memoria occupata dall'elemento
  $\Longrightarrow$ dobbiamo invocare `free` passando il puntatore all'elemento da cancellare
  $\Longrightarrow$ è necessario un puntatore ausiliario

Cancellazione del primo elemento



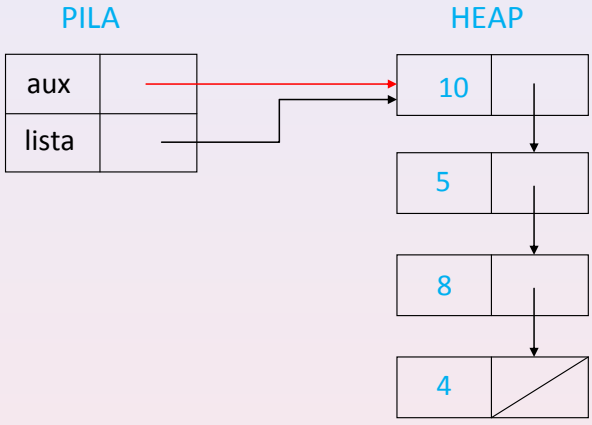
- se la lista è vuota non facciamo nulla
- altrimenti eliminiamo il primo elemento
  $\Longrightarrow$ la lista deve essere passata per indirizzo
- cancellare significa anche deallocare la memoria occupata dall'elemento
  $\Longrightarrow$ dobbiamo invocare `free` passando il puntatore all'elemento da cancellare
  $\Longrightarrow$ è necessario un puntatore ausiliario

Cancellazione del primo elemento



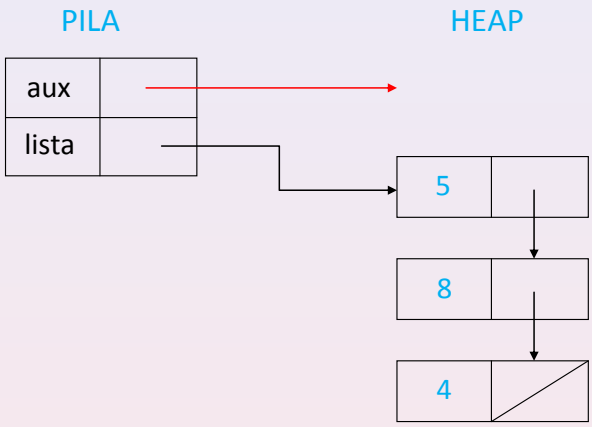
- se la lista è vuota non facciamo nulla
- altrimenti eliminiamo il primo elemento
  $\Longrightarrow$ la lista deve essere passata per indirizzo
- cancellare significa anche deallocare la memoria occupata dall'elemento
  $\Longrightarrow$ dobbiamo invocare `free` passando il puntatore all'elemento da cancellare
  $\Longrightarrow$ è necessario un puntatore ausiliario

Cancellazione del primo elemento



- se la lista è vuota non facciamo nulla
- altrimenti eliminiamo il primo elemento
  $\Longrightarrow$ la lista deve essere passata per indirizzo
- cancellare significa anche deallocare la memoria occupata dall'elemento
  $\Longrightarrow$ dobbiamo invocare `free` passando il puntatore all'elemento da cancellare
  $\Longrightarrow$ è necessario un puntatore ausiliario

Cancellazione del primo elemento



- se la lista è vuota non facciamo nulla
- altrimenti eliminiamo il primo elemento
  $\Longrightarrow$ la lista deve essere passata per indirizzo
- cancellare significa anche deallocare la memoria occupata dall'elemento
  $\Longrightarrow$ dobbiamo invocare `free` passando il puntatore all'elemento da cancellare
  $\Longrightarrow$ è necessario un puntatore ausiliario


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

Cancellazione di tutta una lista

```
void CancellaLista(ListaDiElementi *lista)
{
  ListaDiElementi aux;

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

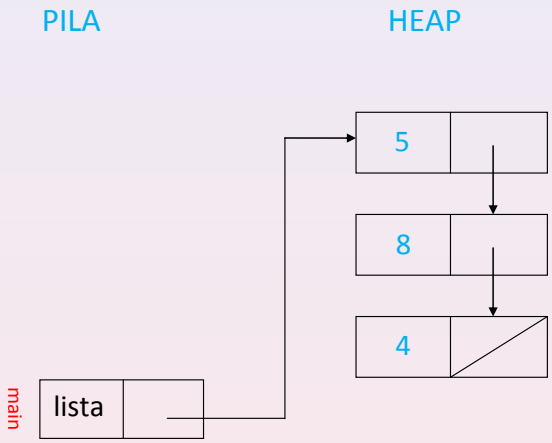
Osserviamo che il corpo del ciclo corrisponde alle azioni della procedura `CancellaPrimo`. Possiamo allora scrivere:

```
void CancellaLista(ListaDiElementi *lista)
{
  while (*lista != NULL)
    CancellaPrimo(lista);
}
```

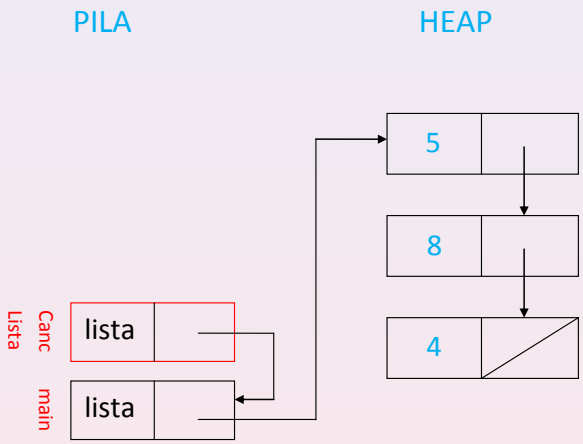
Si noti il parametro attuale della chiamata a `CancellaPrimo`, che è `lista` (di tipo `ListaDiElementi *`) e non `&lista`


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

void CancellaLista(ListaDiElementi *lista)
{
  while (*lista != NULL)
    CancellaPrimo(lista);
}

main()
{
  ListaDiElementi lista;
  ...
  CancellaLista(&lista);
  ...
}
```

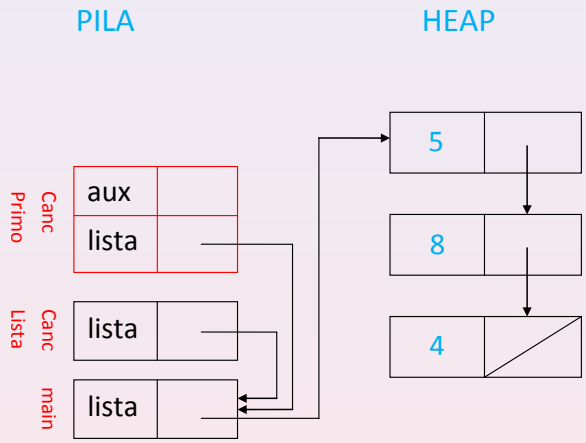
```
void CancellaPrimo(ListaDiElementi *lista)
{
  ListaDiElementi aux;
  if (*lista != NULL)
     {
        aux = *lista;
        *lista = (*lista)->next;
        free(aux);
     }
}

void CancellaLista(ListaDiElementi *lista)
{
  while (*lista != NULL)
    CancellaPrimo(lista);
}

main()
{
  ListaDiElementi lista;
  ...
  CancellaLista(&lista);
  ...
}
```



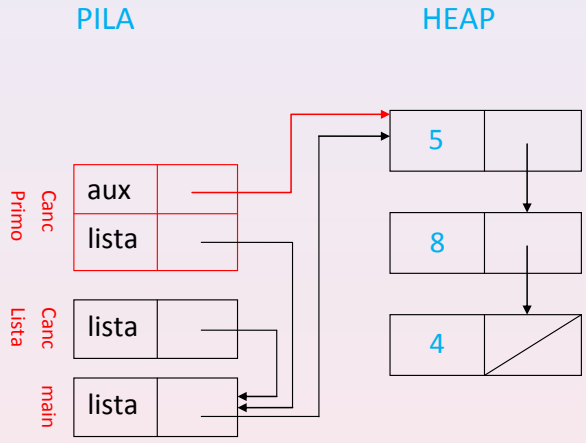

```
void CancellaPrimo(ListaDiElementi *lista)
{
  ListaDiElementi aux;
  if (*lista != NULL)
     {
        aux = *lista;
        *lista = (*lista)->next;
        free(aux);
     }
}

void CancellaLista(ListaDiElementi *lista)
{
  while (*lista != NULL)
    CancellaPrimo(lista);
}

main()
{
  ListaDiElementi lista;
  ...
  CancellaLista(&lista);
  ...
}
```



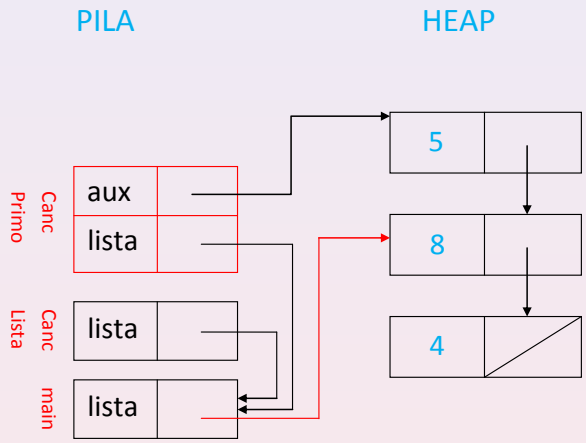
```
void CancellaPrimo(ListaDiElementi *lista)
{
  ListaDiElementi aux;
  if (*lista != NULL)
     {
        aux = *lista;
        *lista = (*lista)->next;
        free(aux);
     }
}

void CancellaLista(ListaDiElementi *lista)
{
  while (*lista != NULL)
    CancellaPrimo(lista);
}

main()
{
  ListaDiElementi lista;
  ...
  CancellaLista(&lista);
  ...
}
```



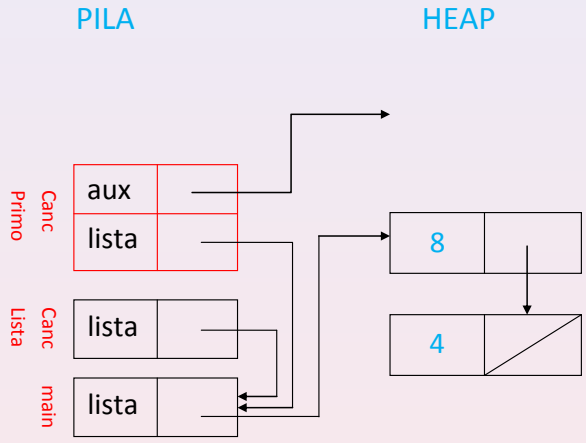

```
void CancellaPrimo(ListaDiElementi *lista)
{
  ListaDiElementi aux;
  if (*lista != NULL)
     {
        aux = *lista;
        *lista = (*lista)->next;
        free(aux);
     }
}

void CancellaLista(ListaDiElementi *lista)
{
  while (*lista != NULL)
    CancellaPrimo(lista);
}

main()
{
  ListaDiElementi lista;
  ...
  CancellaLista(&lista);
  ...
}
```

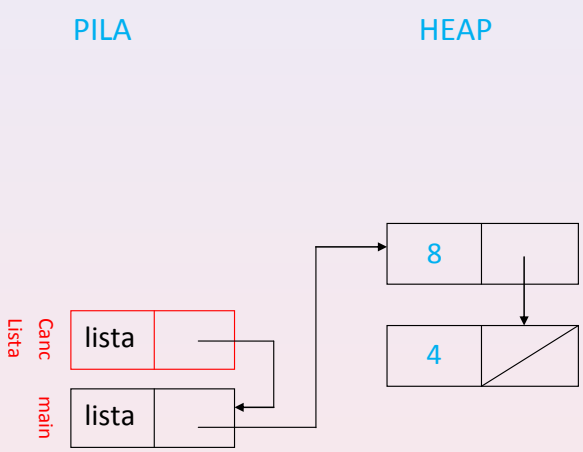
```
void CancellaPrimo(ListaDiElementi *lista)
{
  ListaDiElementi aux;
  if (*lista != NULL)
     {
        aux = *lista;
        *lista = (*lista)->next;
        free(aux);
     }
}

void CancellaLista(ListaDiElementi *lista)
{
  while (*lista != NULL)
    CancellaPrimo(lista);
}

main()
{
  ListaDiElementi lista;
  ...
  CancellaLista(&lista);
  ...
}
```



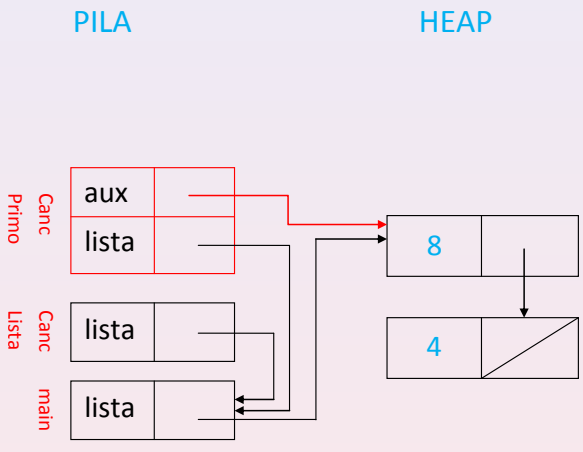

```
void CancellaPrimo(ListaDiElementi *lista)
{
  ListaDiElementi aux;
  if (*lista != NULL)
     {
        aux = *lista;
        *lista = (*lista)->next;
        free(aux);
     }
}

void CancellaLista(ListaDiElementi *lista)
{
  while (*lista != NULL)
    CancellaPrimo(lista);
}

main()
{
  ListaDiElementi lista;
  ...
  CancellaLista(&lista);
  ...
}
```

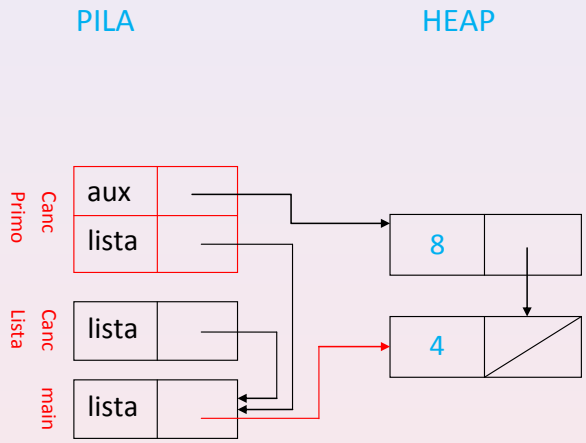
```
void CancellaPrimo(ListaDiElementi *lista)
{
  ListaDiElementi aux;
  if (*lista != NULL)
     {
        aux = *lista;
        *lista = (*lista)->next;
        free(aux);
     }
}

void CancellaLista(ListaDiElementi *lista)
{
  while (*lista != NULL)
    CancellaPrimo(lista);
}

main()
{
  ListaDiElementi lista;
  ...
  CancellaLista(&lista);
  ...
}
```



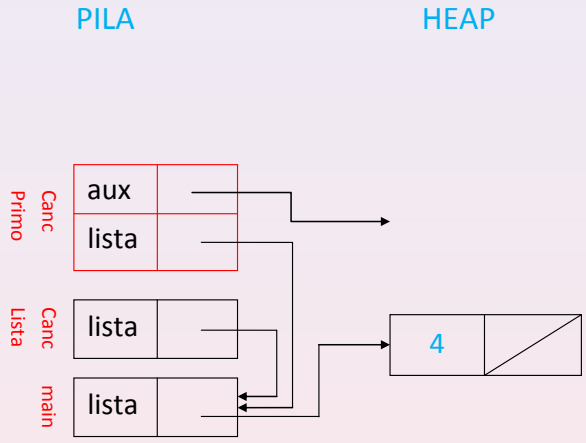

```
void CancellaPrimo(ListaDiElementi *lista)
{
  ListaDiElementi aux;
  if (*lista != NULL)
     {
        aux = *lista;
        *lista = (*lista)->next;
        free(aux);
     }
}

void CancellaLista(ListaDiElementi *lista)
{
  while (*lista != NULL)
    CancellaPrimo(lista);
}

main()
{
  ListaDiElementi lista;
  ...
  CancellaLista(&lista);
  ...
}
```



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

void CancellaLista(ListaDiElementi *lista)
{
  while (*lista != NULL)
    CancellaPrimo(lista);
}

main()
{
  ListaDiElementi lista;
  ...
  CancellaLista(&lista);
  ...
}
```

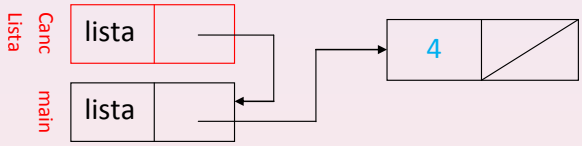



```
void CancellaPrimo(ListaDiElementi *lista)
{
  ListaDiElementi aux;
  if (*lista != NULL)
     {
        aux = *lista;
        *lista = (*lista)->next;
        free(aux);
     }
}

void CancellaLista(ListaDiElementi *lista)
{
  while (*lista != NULL)
    CancellaPrimo(lista);
}

main()
{
  ListaDiElementi lista;
  ...
  CancellaLista(&lista);
  ...
}
```

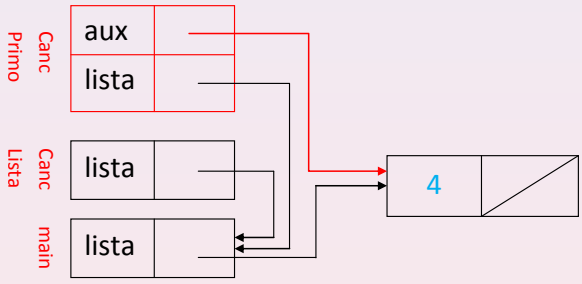
```
void CancellaPrimo(ListaDiElementi *lista)
{
  ListaDiElementi aux;
  if (*lista != NULL)
     {
        aux = *lista;
        *lista = (*lista)->next;
        free(aux);
     }
}

void CancellaLista(ListaDiElementi *lista)
{
  while (*lista != NULL)
    CancellaPrimo(lista);
}

main()
{
  ListaDiElementi lista;
  ...
  CancellaLista(&lista);
  ...
}
```

PILA HEAP



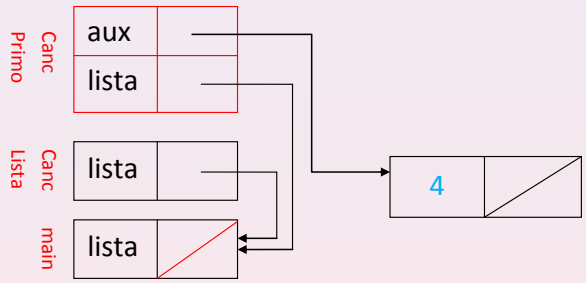

```
void CancellaPrimo(ListaDiElementi *lista)
{
  ListaDiElementi aux;
  if (*lista != NULL)
     {
        aux = *lista;
        *lista = (*lista)->next;
        free(aux);
     }
}

void CancellaLista(ListaDiElementi *lista)
{
  while (*lista != NULL)
    CancellaPrimo(lista);
}

main()
{
  ListaDiElementi lista;
  ...
  CancellaLista(&lista);
  ...
}
```

PILA HEAP



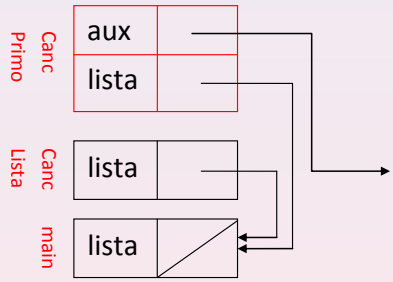
```
void CancellaPrimo(ListaDiElementi *lista)
{
  ListaDiElementi aux;
  if (*lista != NULL)
     {
        aux = *lista;
        *lista = (*lista)->next;
        free(aux);
     }
}

void CancellaLista(ListaDiElementi *lista)
{
  while (*lista != NULL)
    CancellaPrimo(lista);
}

main()
{
  ListaDiElementi lista;
  ...
  CancellaLista(&lista);
  ...
}
```

PILA HEAP



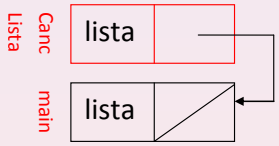

```
void CancellaPrimo(ListaDiElementi *lista)
{
  ListaDiElementi aux;
  if (*lista != NULL)
     {
        aux = *lista;
        *lista = (*lista)->next;
        free(aux);
     }
}

void CancellaLista(ListaDiElementi *lista)
{
  while (*lista != NULL)
    CancellaPrimo(lista);
}

main()
{
  ListaDiElementi lista;
  ...
  CancellaLista(&lista);
  ...
}
```

PILA HEAP



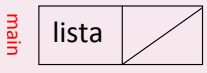
```
void CancellaPrimo(ListaDiElementi *lista)
{
  ListaDiElementi aux;
  if (*lista != NULL)
     {
        aux = *lista;
        *lista = (*lista)->next;
        free(aux);
     }
}

void CancellaLista(ListaDiElementi *lista)
{
  while (*lista != NULL)
    CancellaPrimo(lista);
}

main()
{
  ListaDiElementi lista;
  ...
  CancellaLista(&lista);
  ...
}
```

PILA HEAP

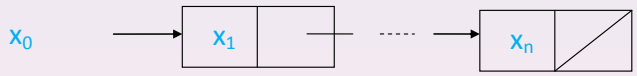



```
void CancellaPrimo(ListaDiElementi *lista)
{
  ListaDiElementi aux;
  if (*lista != NULL)
     {
        aux = *lista;
        *lista = (*lista)->next;
        free(aux);
     }
}

void CancellaLista(ListaDiElementi *lista)
{
  while (*lista != NULL)
    CancellaPrimo(lista);
}

main()
{
  ListaDiElementi lista;
  ...
  CancellaLista(&lista);
  ...
}
```

PILA HEAP

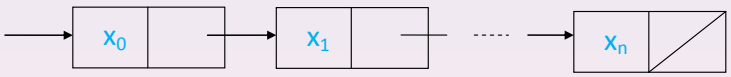


Visione ricorsiva delle liste



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

Visione ricorsiva delle liste



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

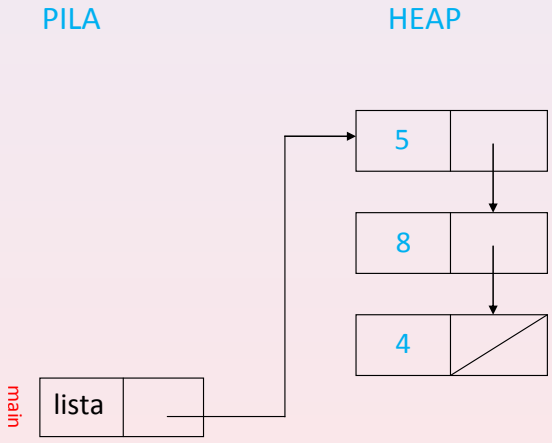
Cancellazione lista: versione ricorsiva

- 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

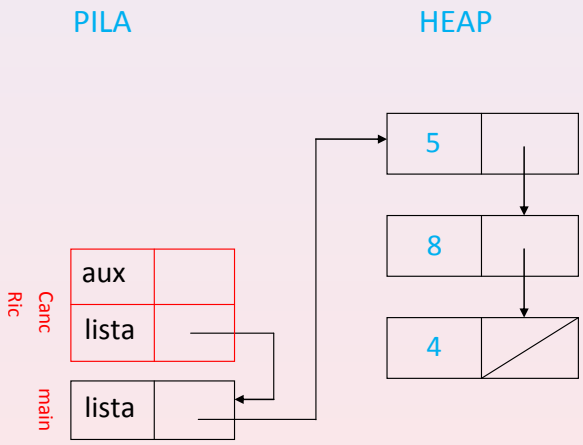
```
void CancellaListaRic(ListaDiElementi *lista)
{
  ListaDiElementi aux;
  if (*lista != NULL)
  {
    aux = *lista;
    *lista = (*lista)->next;
    free(aux);
    CancellaListaRic(lista);
  }
}
```



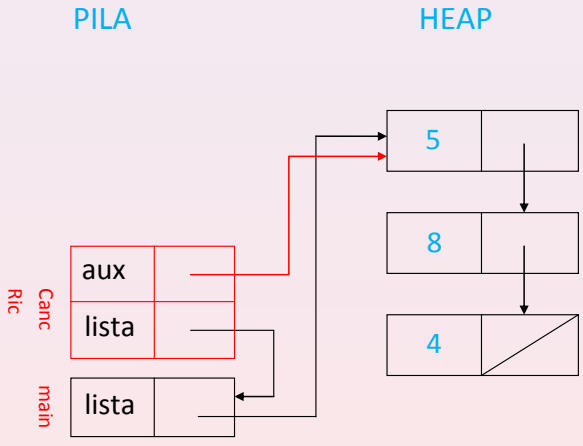
```
void CancellaListaRic(ListaDiElementi *lista)
{
  ListaDiElementi aux;
  if (*lista != NULL)
  {
    aux = *lista;
    *lista = (*lista)->next;
    free(aux);
    CancellaListaRic(lista);
  }
}
```

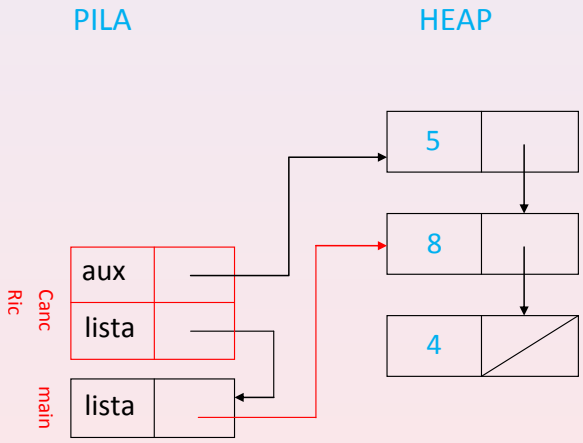
```
void CancellaListaRic(ListaDiElementi *lista)
{
  ListaDiElementi aux;
  if (*lista != NULL)
  {
    aux = *lista;
    *lista = (*lista)->next;
    free(aux);
    CancellaListaRic(lista);
  }
}
```



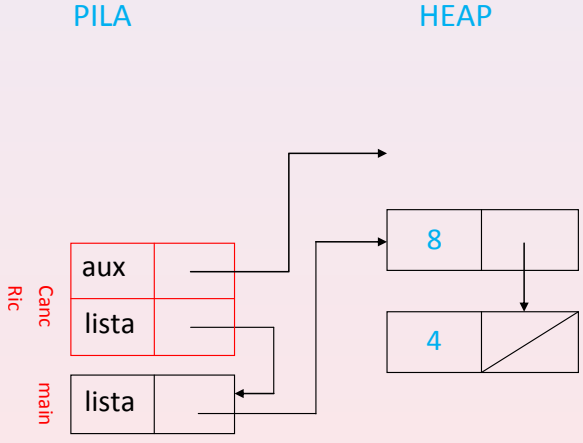

```
void CancellaListaRic(ListaDiElementi *lista)
{
  ListaDiElementi aux;
  if (*lista != NULL)
  {
    aux = *lista;
    *lista = (*lista)->next;
    free(aux);
    CancellaListaRic(lista);
  }
}
```



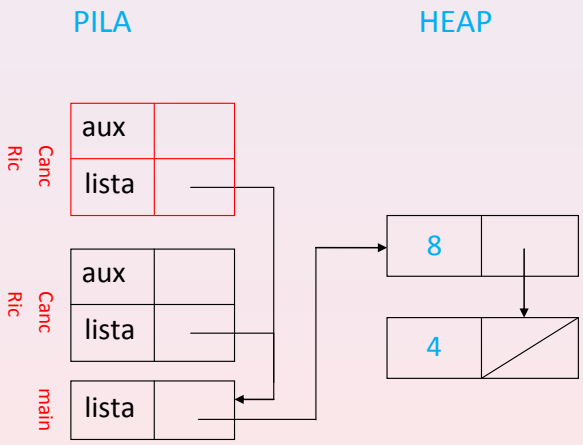
```
void CancellaListaRic(ListaDiElementi *lista)
{
  ListaDiElementi aux;
  if (*lista != NULL)
  {
    aux = *lista;
    *lista = (*lista)->next;
    free(aux);
    CancellaListaRic(lista);
  }
}
```



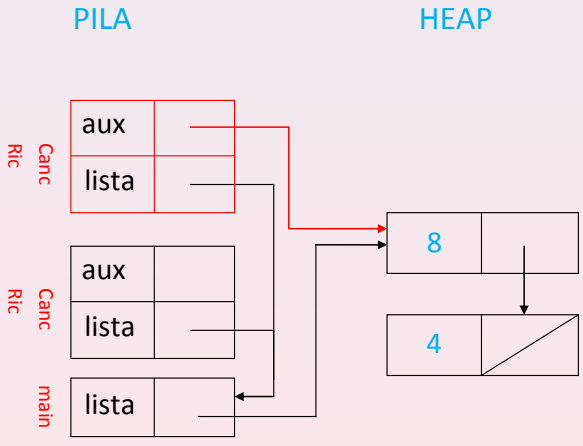

```
void CancellaListaRic(ListaDiElementi *lista)
{
  ListaDiElementi aux;
  if (*lista != NULL)
  {
    aux = *lista;
    *lista = (*lista)->next;
    free(aux);
    CancellaListaRic(lista);
  }
}
```



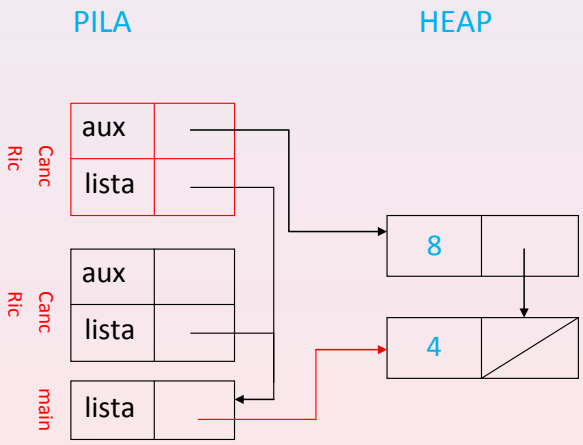
```
void CancellaListaRic(ListaDiElementi *lista)
{
  ListaDiElementi aux;
  if (*lista != NULL)
  {
    aux = *lista;
    *lista = (*lista)->next;
    free(aux);
    CancellaListaRic(lista);
  }
}
```



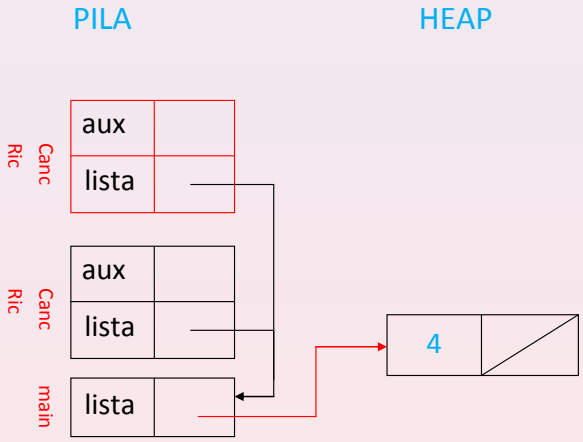

```
void CancellaListaRic(ListaDiElementi *lista)
{
  ListaDiElementi aux;
  if (*lista != NULL)
  {
    aux = *lista;
    *lista = (*lista)->next;
    free(aux);
    CancellaListaRic(lista);
  }
}
```



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

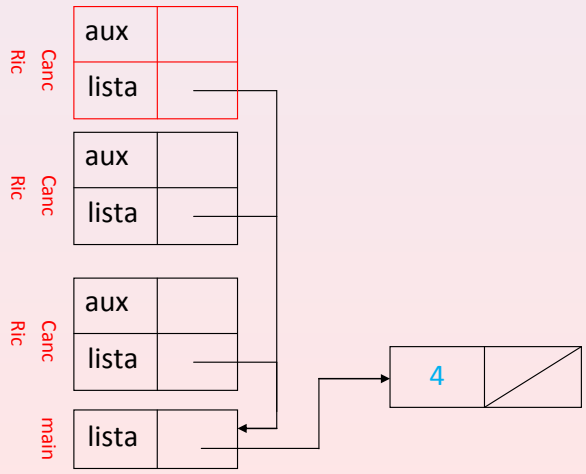



```
void CancellaListaRic(ListaDiElementi *lista)
{
  ListaDiElementi aux;
  if (*lista != NULL)
  {
    aux = *lista;
    *lista = (*lista)->next;
    free(aux);
    CancellaListaRic(lista);
  }
}
```



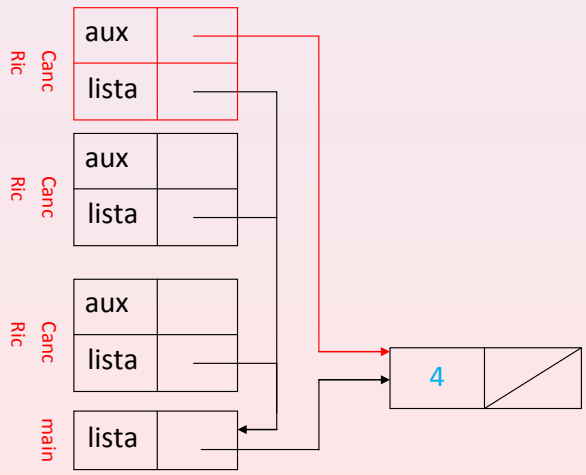
```
void CancellaListaRic(ListaDiElementi *lista)
{
  ListaDiElementi aux;
  if (*lista != NULL)
  {
    aux = *lista;
    *lista = (*lista)->next;
    free(aux);
    CancellaListaRic(lista);
  }
}
```

PILA HEAP



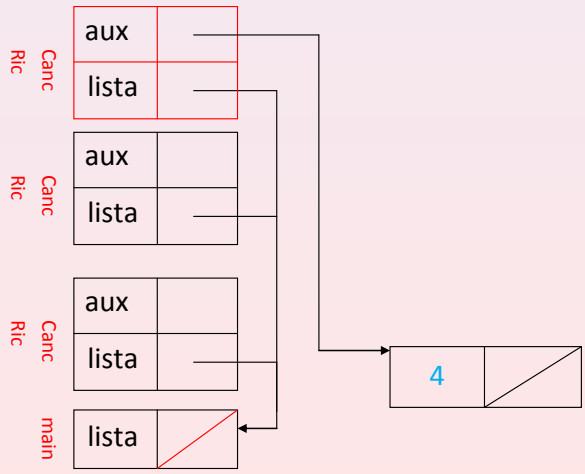

```
void CancellaListaRic(ListaDiElementi *lista)
{
  ListaDiElementi aux;
  if (*lista != NULL)
  {
    aux = *lista;
    *lista = (*lista)->next;
    free(aux);
    CancellaListaRic(lista);
  }
}
```

PILA HEAP



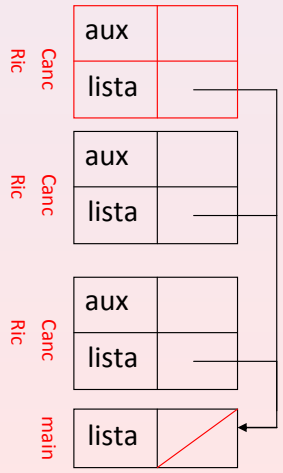
```
void CancellaListaRic(ListaDiElementi *lista)
{
  ListaDiElementi aux;
  if (*lista != NULL)
  {
    aux = *lista;
    *lista = (*lista)->next;
    free(aux);
    CancellaListaRic(lista);
  }
}
```

PILA HEAP

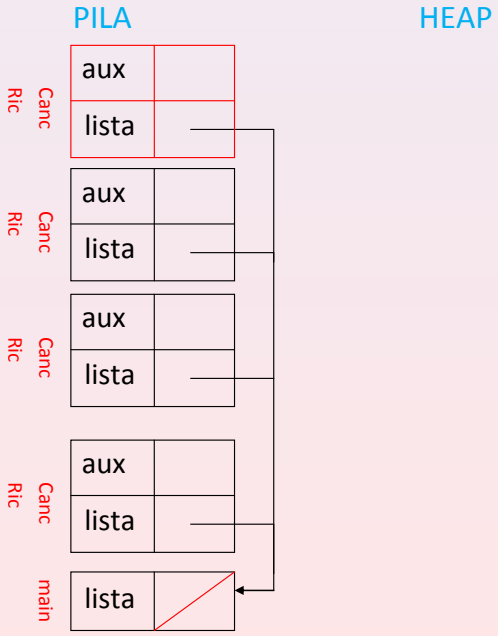



```
void CancellaListaRic(ListaDiElementi *lista)
{
  ListaDiElementi aux;
  if (*lista != NULL)
  {
    aux = *lista;
    *lista = (*lista)->next;
    free(aux);
    CancellaListaRic(lista);
  }
}
```

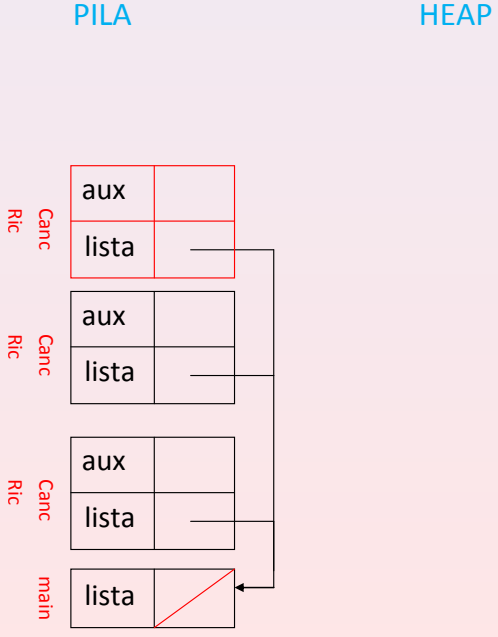
PILA HEAP



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

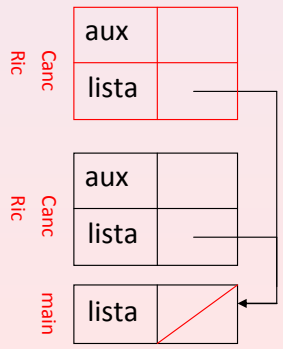



```
void CancellaListaRic(ListaDiElementi *lista)
{
  ListaDiElementi aux;
  if (*lista != NULL)
  {
    aux = *lista;
    *lista = (*lista)->next;
    free(aux);
    CancellaListaRic(lista);
  }
}
```

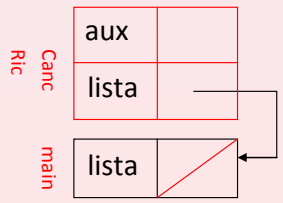
```
void CancellaListaRic(ListaDiElementi *lista)
{
  ListaDiElementi aux;
  if (*lista != NULL)
  {
    aux = *lista;
    *lista = (*lista)->next;
    free(aux);
    CancellaListaRic(lista);
  }
}
```

PILA HEAP



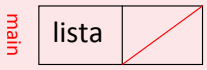

```
void CancellaListaRic(ListaDiElementi *lista)
{
  ListaDiElementi aux;
  if (*lista != NULL)
  {
    aux = *lista;
    *lista = (*lista)->next;
    free(aux);
    CancellaListaRic(lista);
  }
}
```

PILA HEAP



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

PILA HEAP



Appartenenza di un elemento ad una lista

- Ricordiamo la ricerca lineare incerta su vettori
- sostituiamo l'indice i con un puntatore p
- scorriamo la lista attraverso p
- l'elemento corrente è quello puntato da p
- Incapsuliamo questo codice in una funzione a valori booleani

Appartenenza di un elemento ad una lista

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

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

- Non c'è bisogno di un puntatore ausiliario per scorrere la lista
  $\Longrightarrow$ il passaggio per valore consente di scorrere utilizzando il parametro formale!
- Abbiamo assunto che sul tipo `TipoElementoLista` sia definito l'operatore di uguaglianza ==

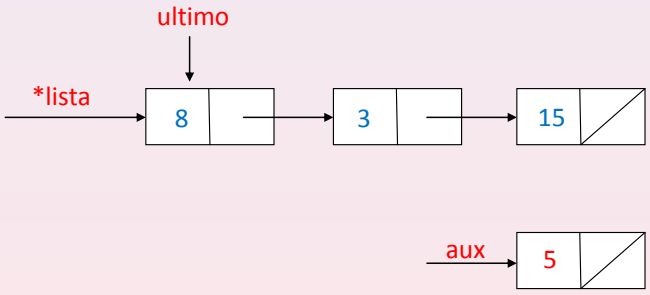
Versione ricorsiva

```
boolean Appartiene(TipoElementoLista 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

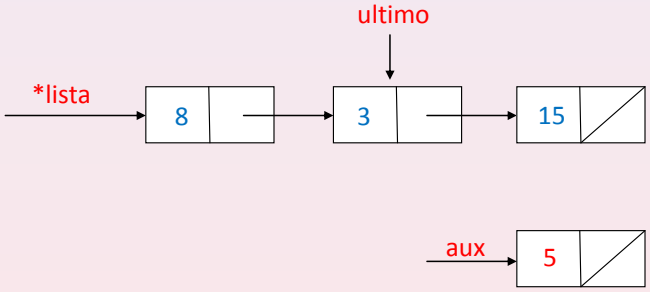
Inserimento di un elemento in coda

- Se la lista è vuota coincide con l'inserimento in testa
  $\Longrightarrow$ è necessario il passaggio per indirizzo!
- Se la lista non è vuota, bisogna scandirla fino in fondo
  $\Longrightarrow$ dobbiamo usare un puntatore ausiliario per la scansione
- La scansione deve terminare in corrispondenza dell'ultimo elemento al quale va collegato quello nuovo



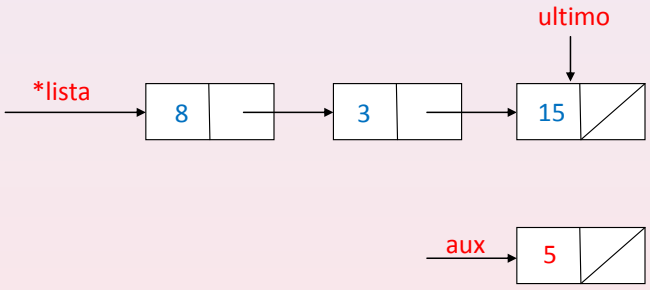
Inserimento di un elemento in coda

- Se la lista è vuota coincide con l'inserimento in testa
  $\Longrightarrow$ è necessario il passaggio per indirizzo!
- Se la lista non è vuota, bisogna scandirla fino in fondo
  $\Longrightarrow$ dobbiamo usare un puntatore ausiliario per la scansione
- La scansione deve terminare in corrispondenza dell'ultimo elemento al quale va collegato quello nuovo



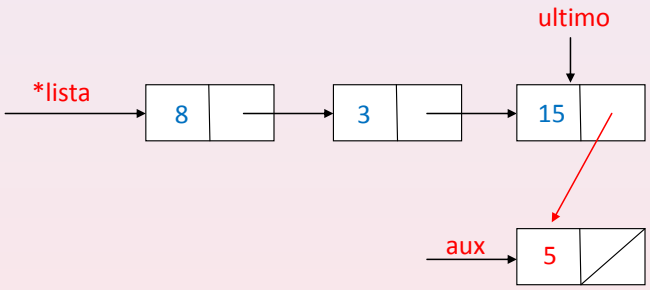
Inserimento di un elemento in coda

- Se la lista è vuota coincide con l'inserimento in testa
  $\Longrightarrow$ è necessario il passaggio per indirizzo!
- Se la lista non è vuota, bisogna scandirla fino in fondo
  $\Longrightarrow$ dobbiamo usare un puntatore ausiliario per la scansione
- La scansione deve terminare in corrispondenza dell'ultimo elemento al quale va collegato quello nuovo



Inserimento di un elemento in coda

- Se la lista è vuota coincide con l'inserimento in testa
  $\Longrightarrow$ è necessario il passaggio per indirizzo!
- Se la lista non è vuota, bisogna scandirla fino in fondo
  $\Longrightarrow$ dobbiamo usare un puntatore ausiliario per la scansione
- La scansione deve terminare in corrispondenza dell'ultimo elemento al quale va collegato quello nuovo



Codice della versione iterativa

```
void InserzioneInCoda(ListaDiElementi *lista, TipoElementoLista elem)
{
  ListaDiElementi ultimo;   /* puntatore usato per la scansione */
  ListaDiElementi aux;
                            /* creazione del nuovo elemento */
  aux = malloc(sizeof(ElementoLista));
  aux->info = elem;
  aux->next = NULL;

  if (*lista == NULL)
    *lista = aux;
  else {
    ultimo = *lista;
    while (ultimo->next != NULL)
      ultimo = ultimo->next;
                      /* concatenazione del nuovo elemento in coda alla lista */
    ultimo->next = aux;
  }
}
```


Inserimento ricorsivo di un elemento in coda

- Caratterizzazione induttiva dell'inserimento in coda

  Sia nuovaLista la lista ottenuta inserendo elem in coda a lista.

  1. se lista è vuota, allora nuovaLista è costituita dal solo `elem` (caso base)
  2. altrimenti nuovaLista è ottenuta da lista facendo l'inserimento di elem in coda al resto di lista (caso ricorsivo)

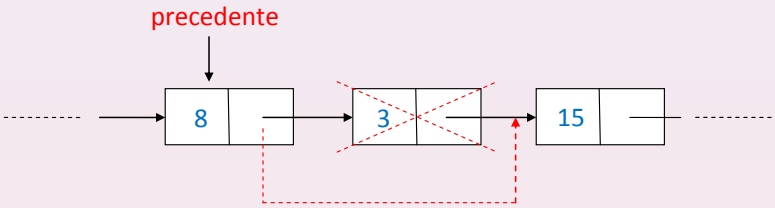
```
InserzioneInCoda(&((*lista)->next), elem);
```

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
     $\Longrightarrow$ 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)



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

  $\Longrightarrow$ 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:

1. se lista è la lista vuota, allora ris è la lista vuota (caso base)
2. 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)
3. altrimenti ris è ottenuta da lista cancellando tutte le occorrenze di elem dal resto di lista (caso ricorsivo)

Esercizio

Implementare le due versioni

Versione iterativa

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

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

Inserimento di un elemento in una lista ordinata

- Data una lista (ad es. di interi) già ordinata (in ordine crescente), si vuole inserire un nuovo elemento mantenendo l'ordinamento.

  Versione iterativa: per **esercizio**

  Versione ricorsiva

- Caratterizziamo il problema induttivamente
- Sia ris la lista ottenuta inserendo l'elemento elem nella lista ordinata lista.
  1. se lista è la lista vuota, allora ris è costituita solo da elem (caso base)
  2. se il primo elemento di lista è maggiore o uguale a elem, allora ris è ottenuta da lista inserendo elem in testa (caso base)
  3. altrimenti ris è ottenuta da lista inserendo ordinatamente elem nel resto di lista (caso ricorsivo)