

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 \implies non servono strutture dati aggiuntive
- ▶ 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)

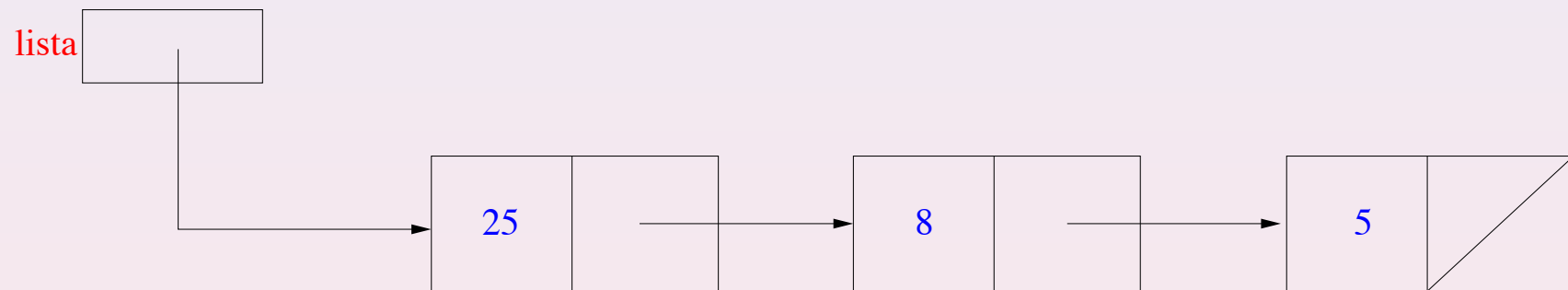
2. Rappresentazione collegata

- ▶ 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 identico 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.

Definizione ricorsiva di lista

- ▶ Possiamo definire ricorsivamente una lista come segue.
- ▶ Una lista è una struttura definita su un insieme di elementi che:
 - ▶ non contiene nessun elemento (lista vuota $[]$), oppure
 - ▶ contiene un elemento EL detto *testa* (head) della lista) seguito dal resto della lista L , detta *coda*: $([EL, L])$
- ▶ La definizione usata dal C riflette proprio questa definizione. Una variabile di tipo lista può valere NULL (che rappresenta la lista vuota), oppure può essere un puntatore a una struttura che contiene un dato più un puntatore, che rappresenta un'altra lista.

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)

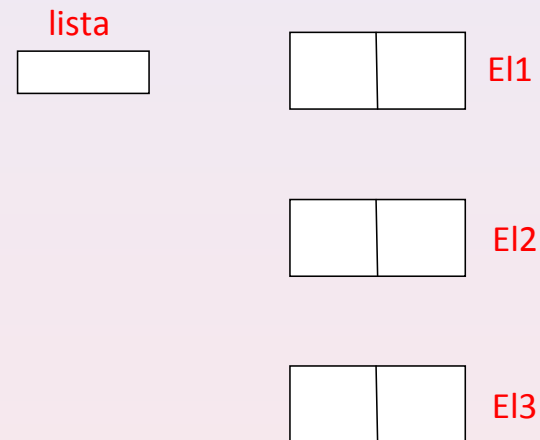
```
ElementoLista E11,E12,E13;  
ListaDiElementi lista;          /* puntatore al primo elemento della lista */
```

```
lista=&E11;
```

```
E11.info = 8;  
E11.next = &E12;
```

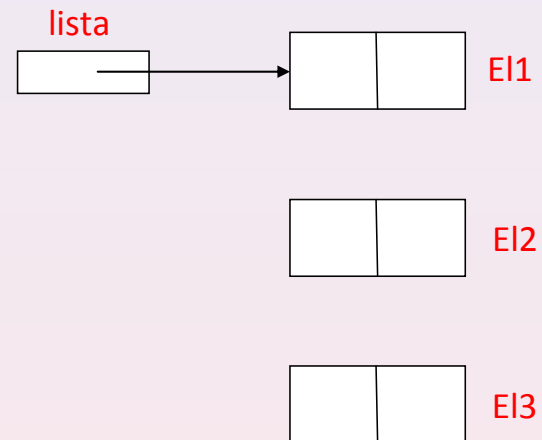
```
E12.info = 3;  
E12.next = &E13;
```

```
E13.info = 15;  
E13.next = NULL;
```



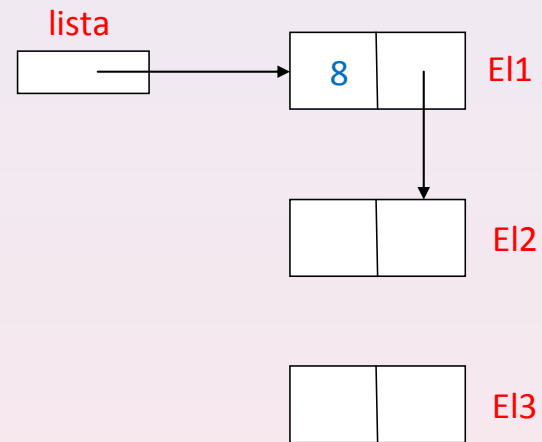
Esempio: Creazione di una lista di tre interi fissati: (8, 3, 15)

```
ElementoLista E11,E12,E13;  
ListaDiElementi lista;          /* puntatore al primo elemento della lista */  
  
lista=&E11;  
  
E11.info = 8;  
E11.next = &E12;  
  
E12.info = 3;  
E12.next = &E13;  
  
E13.info = 15;  
E13.next = NULL;
```



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

```
ElementoLista E1,E2,E3;  
ListaDiElementi lista;          /* puntatore al primo elemento della lista */  
  
lista=&E1;  
  
E1.info = 8;  
E1.next = &E2;  
  
E2.info = 3;  
E2.next = &E3;  
  
E3.info = 15;  
E3.next = NULL;
```



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

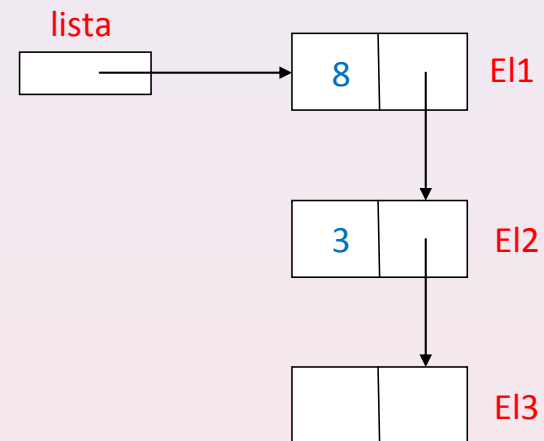
```
ElementoLista E11,E12,E13;  
ListaDiElementi lista;          /* puntatore al primo elemento della lista */
```

```
lista=&E11;
```

```
E11.info = 8;  
E11.next = &E12;
```

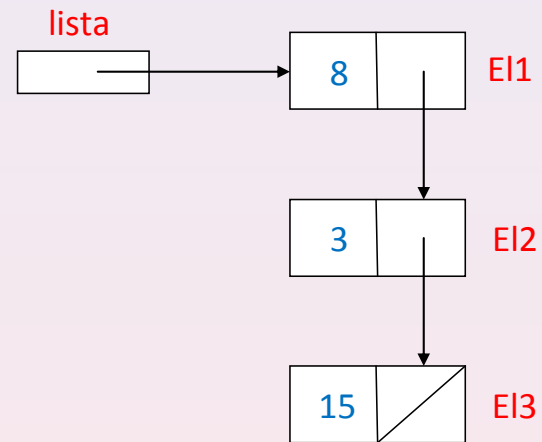
```
E12.info = 3;  
E12.next = &E13;
```

```
E13.info = 15;  
E13.next = NULL;
```



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

```
ElementoLista E1,E2,E3;  
ListaDiElementi lista;          /* puntatore al primo elemento della lista */  
  
lista=&E1;  
  
E1.info = 8;  
E1.next = &E2;  
  
E2.info = 3;  
E2.next = &E3;  
  
E3.info = 15;  
E3.next = NULL;
```



Aliasing

- ▶ Si parla di **aliasing** quando si utilizzano due puntatori (**alias**) per far riferimento allo stesso valore.
- ▶ Se si modifica il valore puntato da uno dei due, implicitamente (come **effetto collaterale**) si modifica anche il valore puntato dall'altro, essendo lo stesso.
- ▶ Questo è un fenomeno particolarmente rilevante quando si manipolano liste.

- ▶ Nell'esempio visto prima, se avessi:

```
lista2=&E12;
```

```
lista2 --> info = 9
```

allora avrei anche che la condizione

```
((E11-->next)-->info == 9) sarebbe vera
```

Ricordiamo che `lista2 --> info` equivale a `(*lista2).info`

- ▶ 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. Possiamo ricorrere all'allocazione dinamica della memoria.

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 */

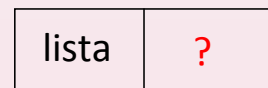
lista->info = 8;

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)

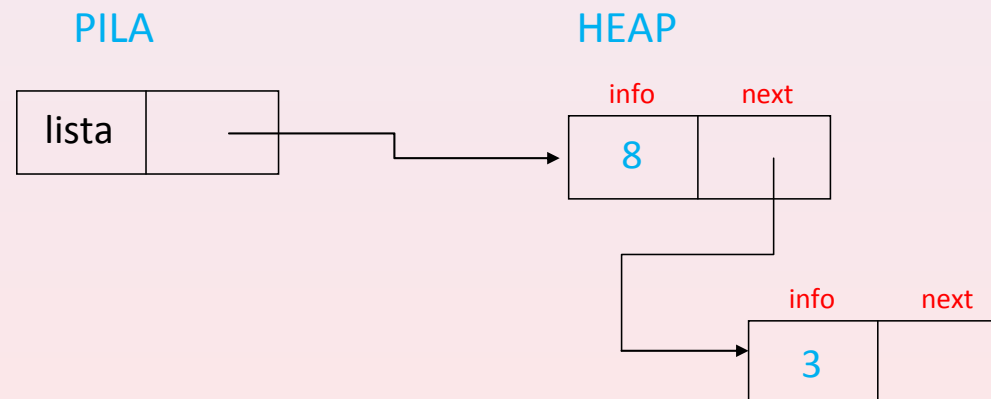
```
ListaDiElementi lista;          /* puntatore al primo elemento della lista */

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)

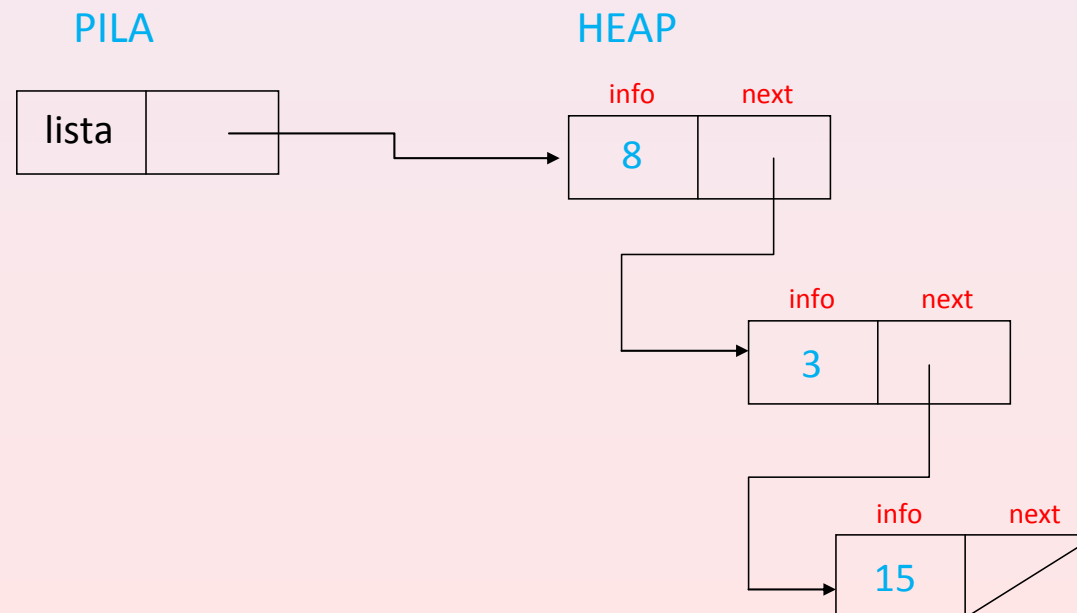
```
ListaDiElementi lista;          /* puntatore al primo elemento della lista */

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;
```



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

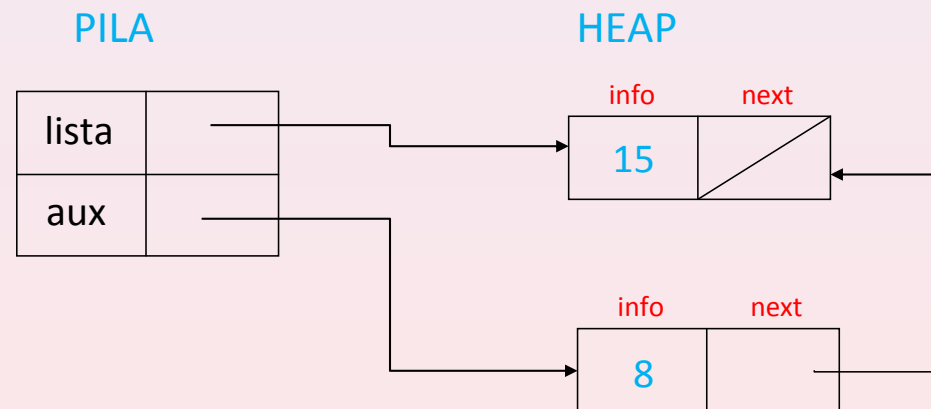
lista	
aux	?


```
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;
```

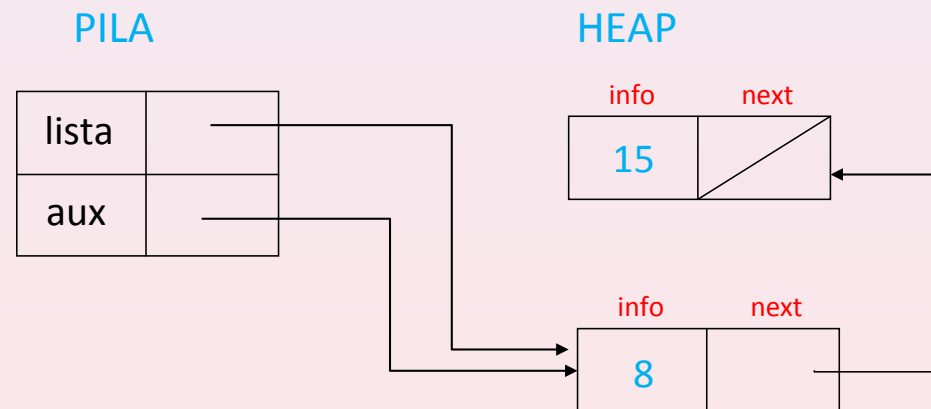


```
ListaDiElementi aux, lista = NULL;

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;
```

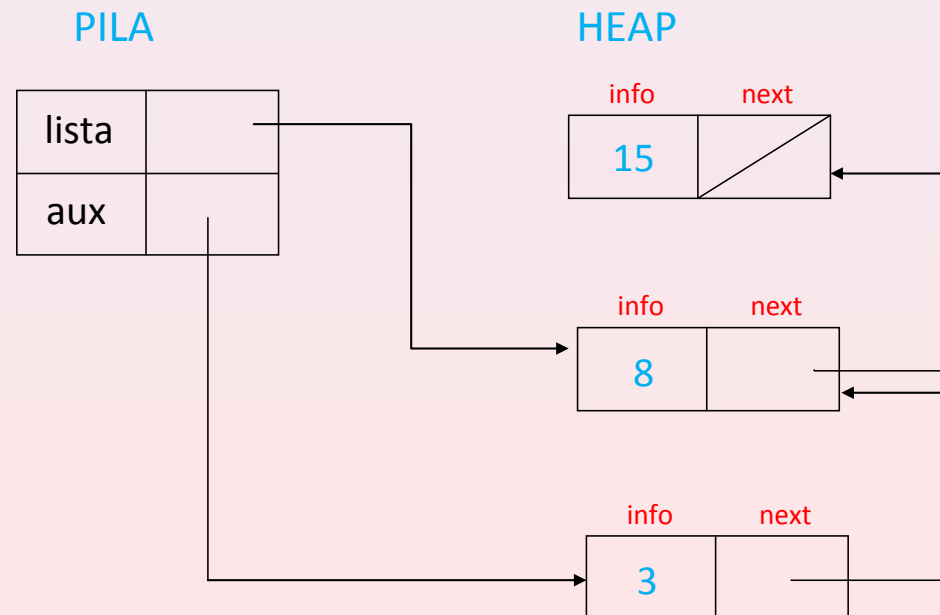



```
ListaDiElementi aux, lista = NULL;

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;
```

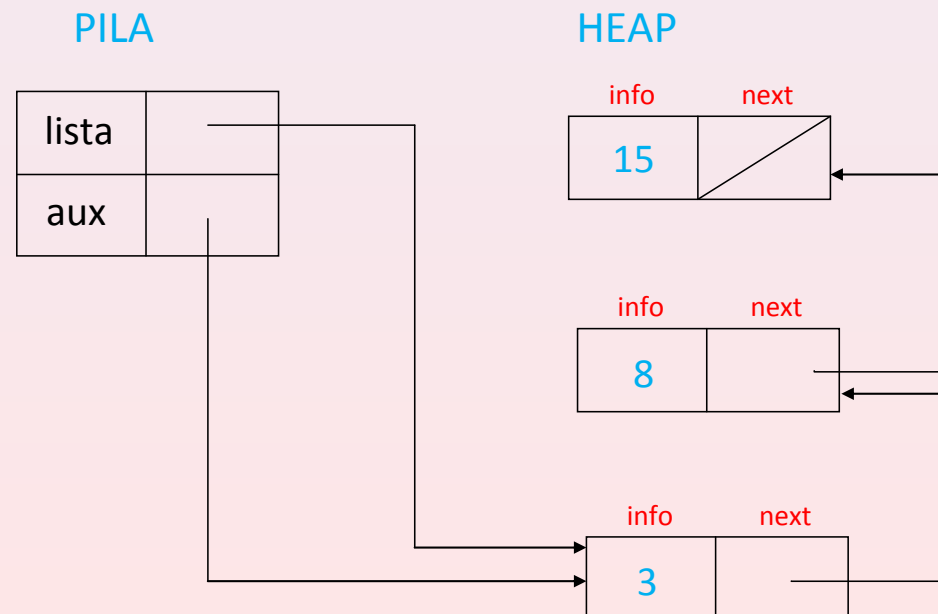


```
ListaDiElementi aux, lista = NULL;

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

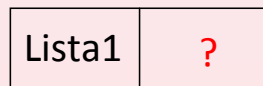
- ▶ Definiamo una procedura che inizializza una lista assegnando il valore `NULL` alla variabile **testa della lista**.
- ▶ Tale variabile deve essere modificata e quindi passata per **indirizzo**.
- ▶ Ciò provoca, nell'intestazione della procedura, la presenza di un puntatore a puntatore.

```
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



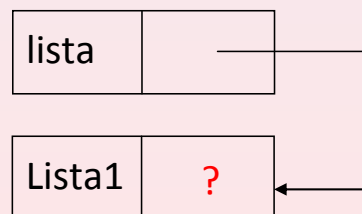
```
void Inizializza(ListaDiElementi *lista)
{
    *lista=NULL;
}
```

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

```
ListadiElementi Lista1;
Inizializza(&Lista1);
```

PILA

RDA Inizializza



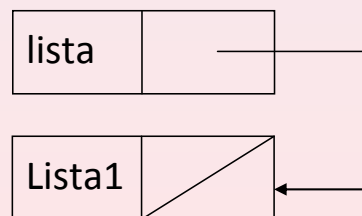
```
void Inizializza(ListaDiElementi *lista)
{
    *lista=NULL;
}
```

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

```
ListadiElementi Lista1;
Inizializza(&Lista1);
```

PILA

RDA Inizializza

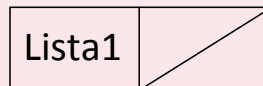


```
void Inizializza(ListaDiElementi *lista)
{
    *lista=NULL;
}
```

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

```
ListadiElementi Lista1;
Inizializza(&Lista1);
```

PILA

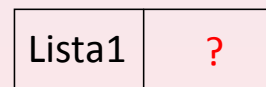


Cosa succederebbe se passassimo il parametro **per valore**?

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

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

PILA



Cosa succederebbe se passassimo il parametro **per valore**?

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

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

PILA

RDA Inizializza

lista	?
-------	---

Lista1	?
--------	---

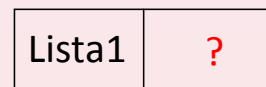
Cosa succederebbe se passassimo il parametro **per valore**?

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

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

PILA

RDA Inizializza

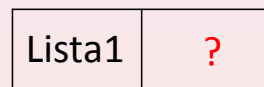


Cosa succederebbe se passassimo il parametro **per valore**?

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

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

PILA



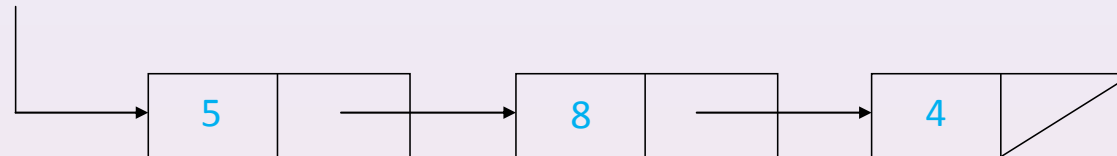
Controllo lista vuota

```
boolean ListaVuota(ListaDiElementi lista)
{
    return (lista==NULL);
}
```

A `lista` viene passato il valore contenuto nella variabile testa di lista e quindi punta al primo elemento della lista considerata.

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 lista
- ▶ **Attenzione:** 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("//");
}

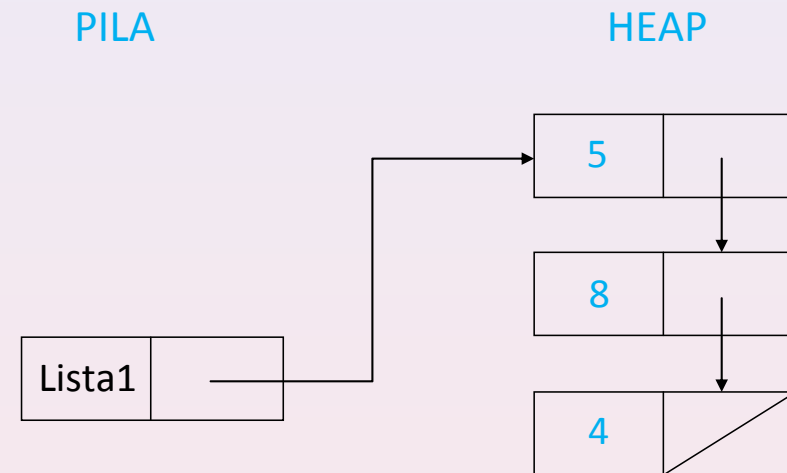
main()
{
    ListaDiElementi Lista1;
    ...
    /* costruzione lista 5 --> 8 --> 4 */
    ...
    StampaLista(Lista1);
    ...
}
```

```

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);
    ...
}

```

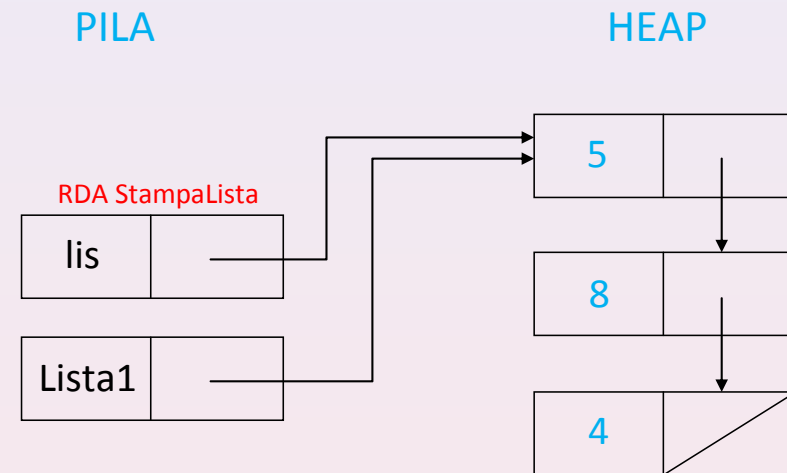



```

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);
    ...
}

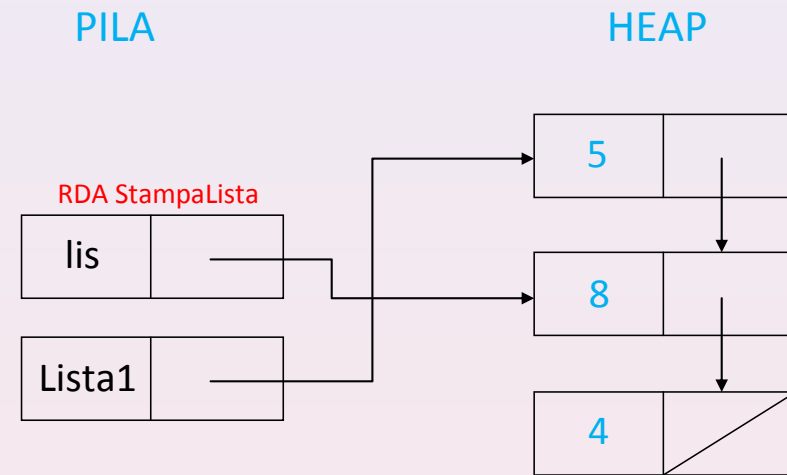
```



```

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

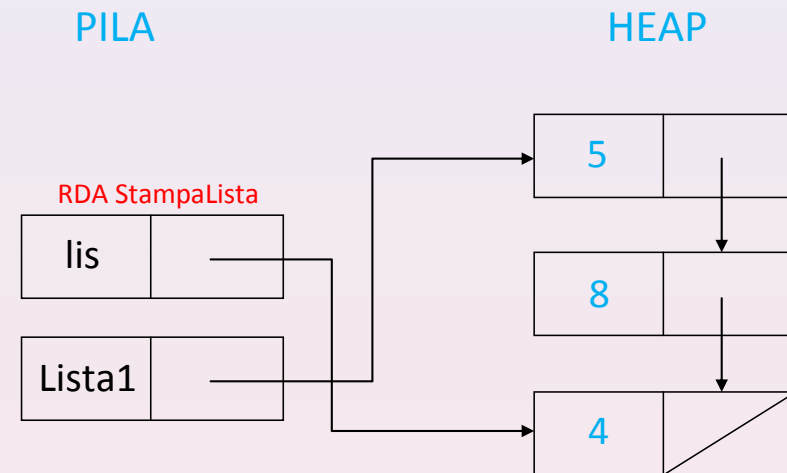
5 -->

```

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

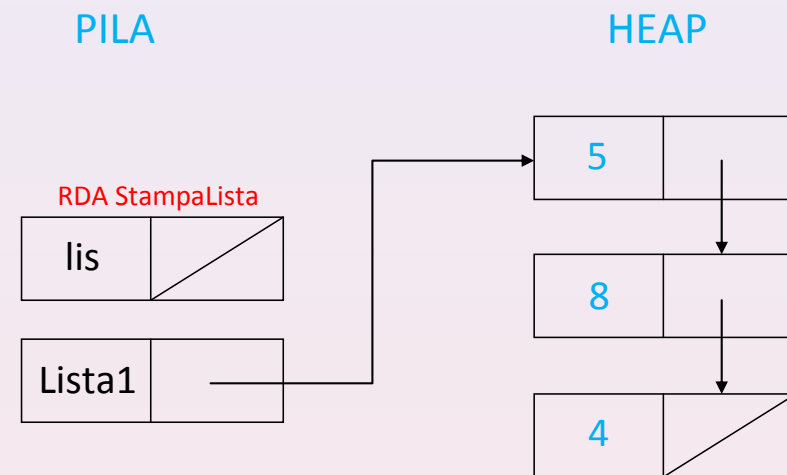
5 --> 8 -->

```

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

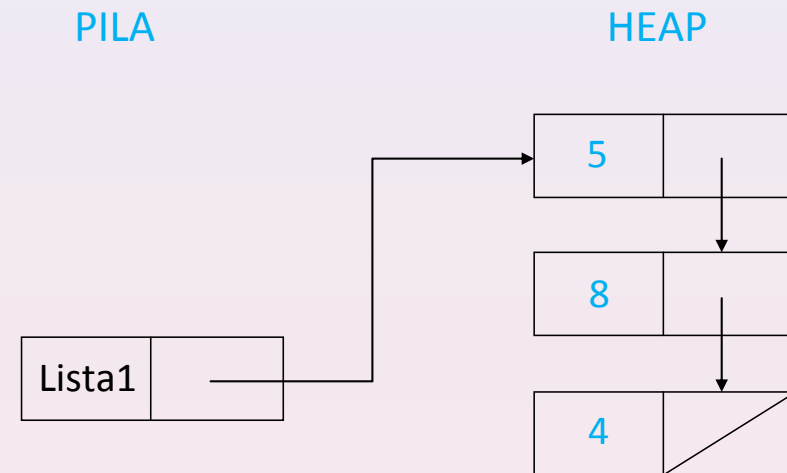
```
5 --> 8 --> 4 --> //
```

```

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);
    ...
}

```



Output

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

Cosa sarebbe successo passando il parametro per indirizzo?

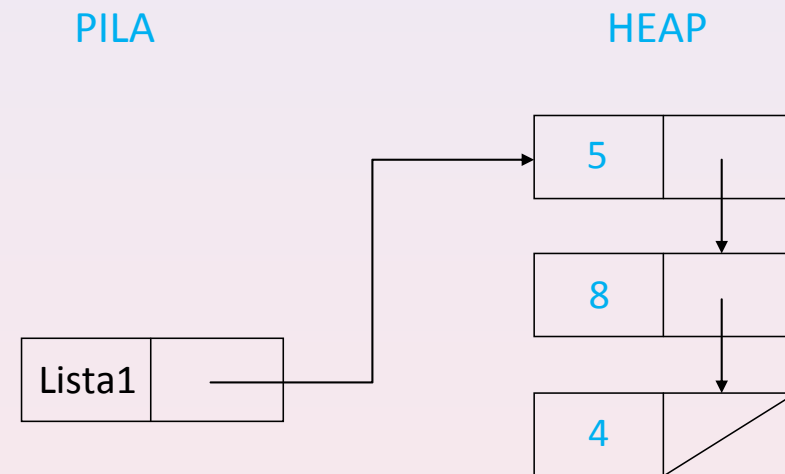
```
void StampaLista(ListaDiElementi *lis)
{
    while (*lis != NULL)
        {
            printf("%d -->", *lis->info);
            *lis = *lis->next;
        }
    printf("//");
}

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

Cosa sarebbe successo passando il parametro per **indirizzo**?

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

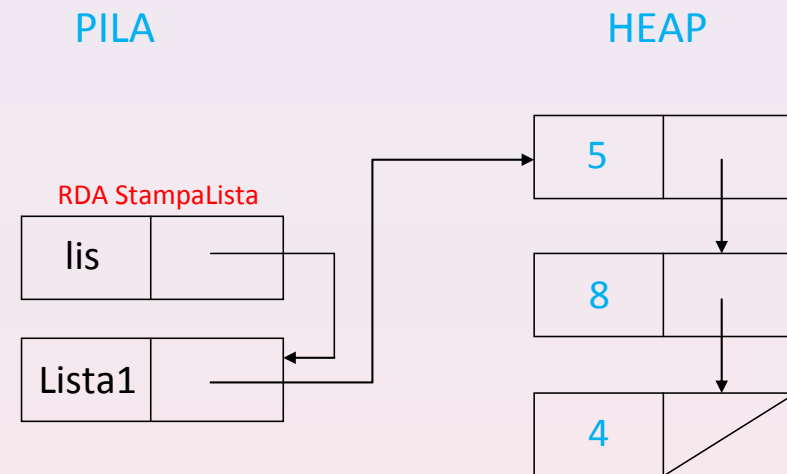
main()
{
    ListaDiElementi Lista1;
    ...
    /* costruzione lista 5 --> 8 --> 4 */
    ...
    StampaLista(&Lista1);
    ...
}
```



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);
    ...
}
```



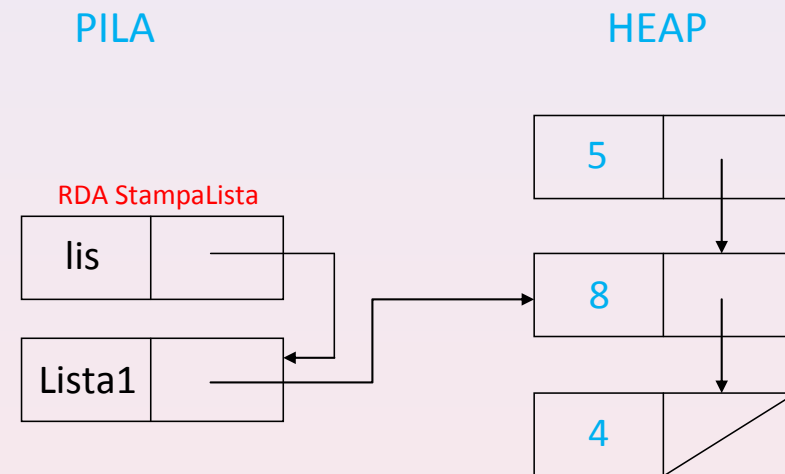
Output



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);
    ...
}
```



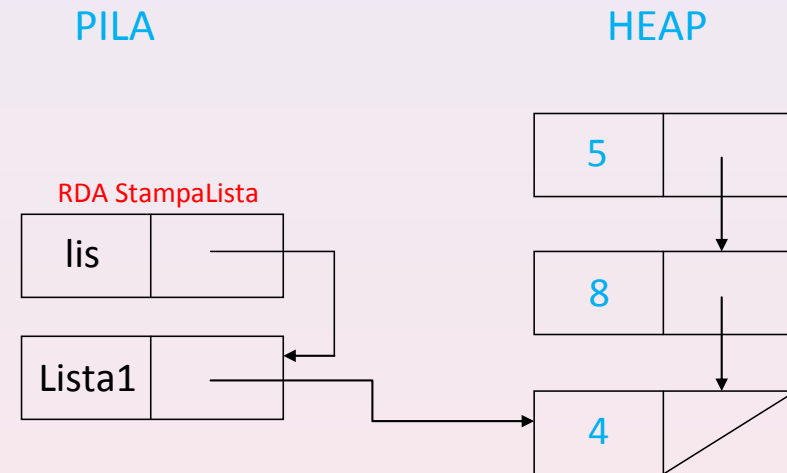
Output

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);
    ...
}
```



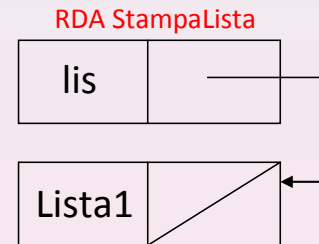
Output
5 --> 8 -->

Cosa sarebbe successo passando il parametro per indirizzo?

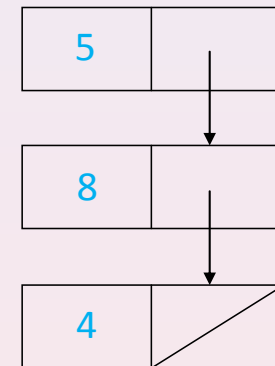
```
void StampaLista(ListaDiElementi *lis)
{
    while (*lis != NULL)
    {
        printf("%d -->", *lis->info);
        *lis = *lis->next;
    }
    printf("//");
}
```

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

PILA



HEAP



Output

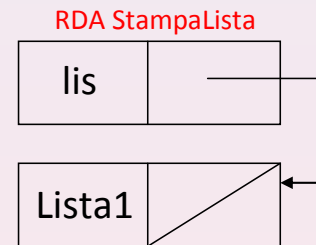
5 --> 8 --> 4 -->

Cosa sarebbe successo passando il parametro per indirizzo?

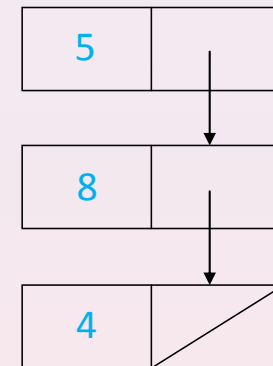
```
void StampaLista(ListaDiElementi *lis)
{
    while (*lis != NULL)
    {
        printf("%d -->", *lis->info);
        *lis = *lis->next;
    }
    printf("//");
}
```

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

PILA



HEAP



Output

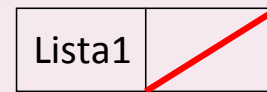
5 --> 8 --> 4 --> //

Cosa sarebbe successo passando il parametro per indirizzo?

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

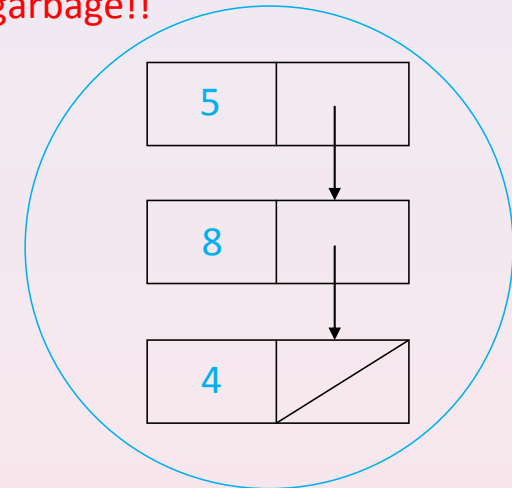
```
main()
{
    ListaDiElementi Lista1;
    ...
    /* costruzione lista 5 --> 8 --> 4 */
    ...
    StampaLista(&Lista1);
    ...
}
```

PILA



HEAP

garbage!!



Output

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

Versione ricorsiva

Se la lista non è vuota stampa il primo elemento, quindi stampa la lista rimanente

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

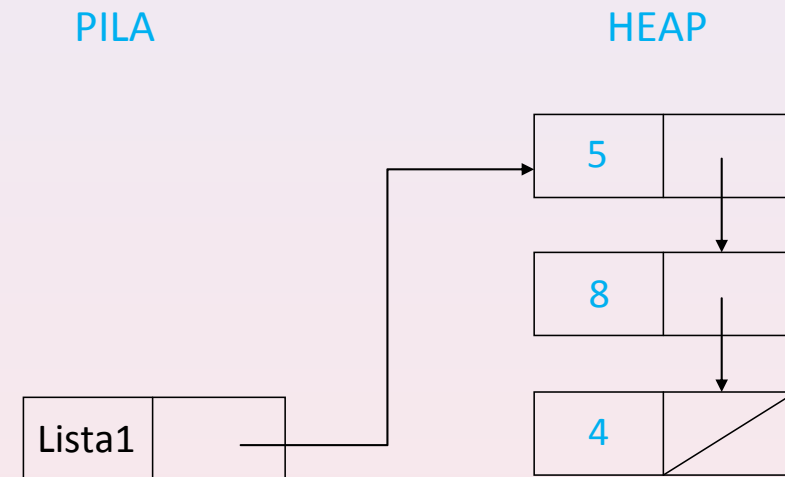
main()
{
    ListaDiElementi Lista1;
    ...
    /* costruzione lista 5 --> 8 --> 4 */
    ...
    StampaListaRic(Lista1);
    ...
}
```

Versione ricorsiva

Se la lista non è vuota stampa il primo elemento, quindi stampa la lista rimanente

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



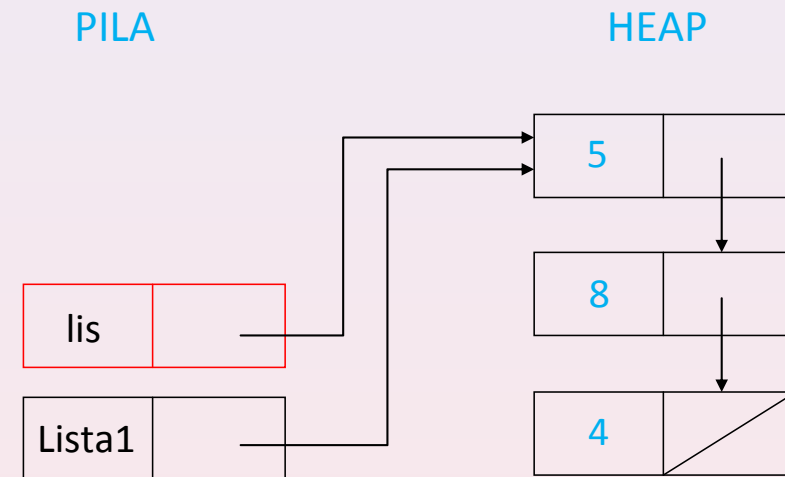
Output

Versione ricorsiva

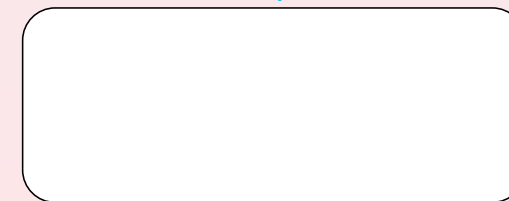
Se la lista non è vuota stampa il primo elemento, quindi stampa la lista rimanente

```
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

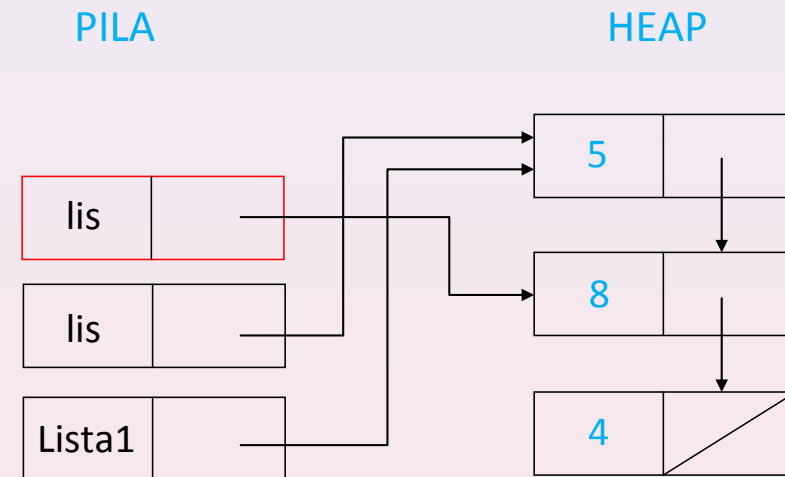


Versione ricorsiva

Se la lista non è vuota stampa il primo elemento, quindi stampa la lista rimanente

```
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

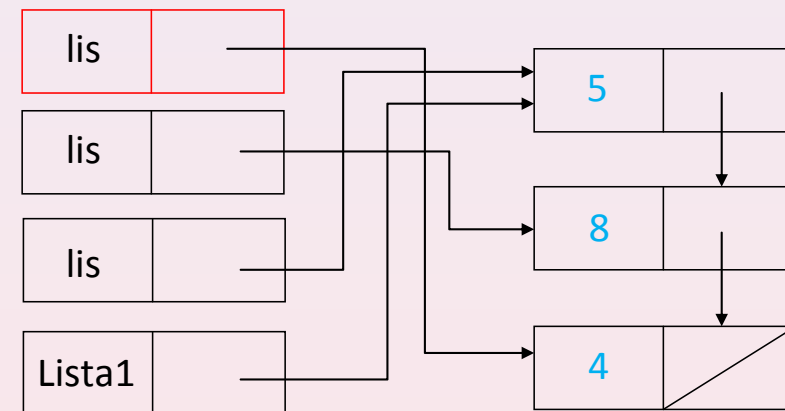
Se la lista non è vuota stampa il primo elemento, quindi stampa la lista rimanente

```
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

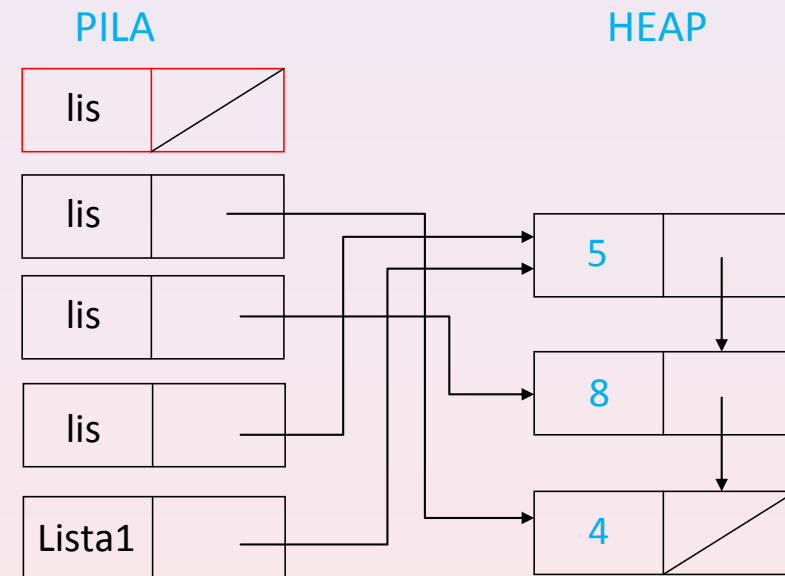
5 --> 8 -->

Versione ricorsiva

Se la lista non è vuota stampa il primo elemento, quindi stampa la lista rimanente

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

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



Output

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

Versione ricorsiva

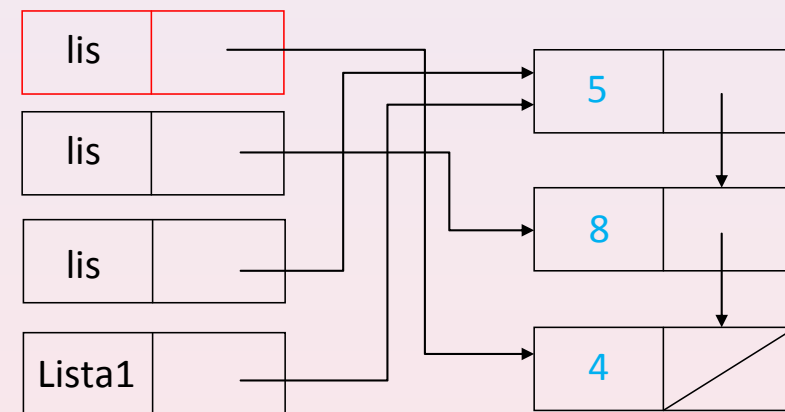
Se la lista non è vuota stampa il primo elemento, quindi stampa la lista rimanente

```
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

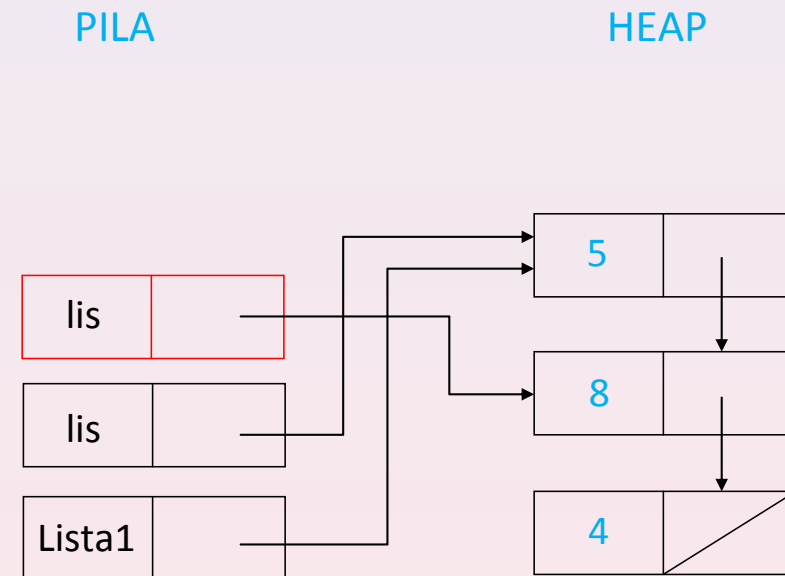
5 --> 8 --> 4 --> //

Versione ricorsiva

Se la lista non è vuota stampa il primo elemento, quindi stampa la lista rimanente

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

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



Output

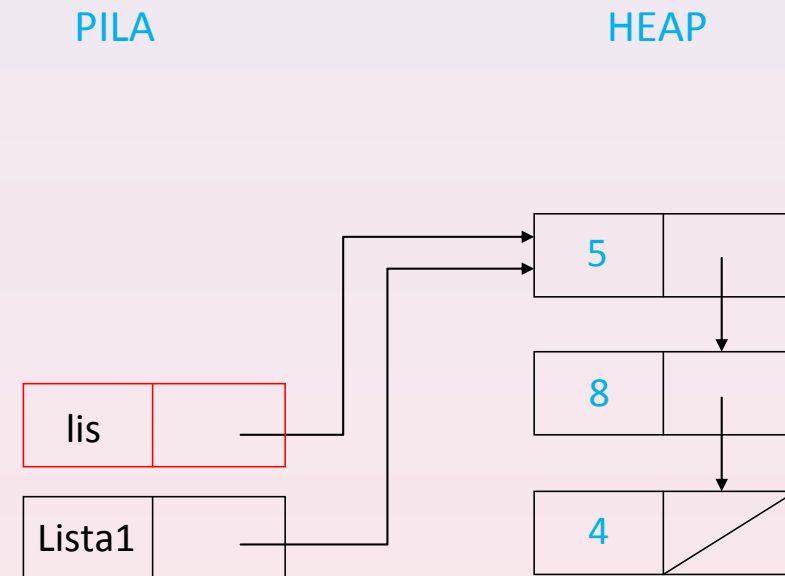
5 --> 8 --> 4 --> //

Versione ricorsiva

Se la lista non è vuota stampa il primo elemento, quindi stampa la lista rimanente

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

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



Output

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

Versione ricorsiva

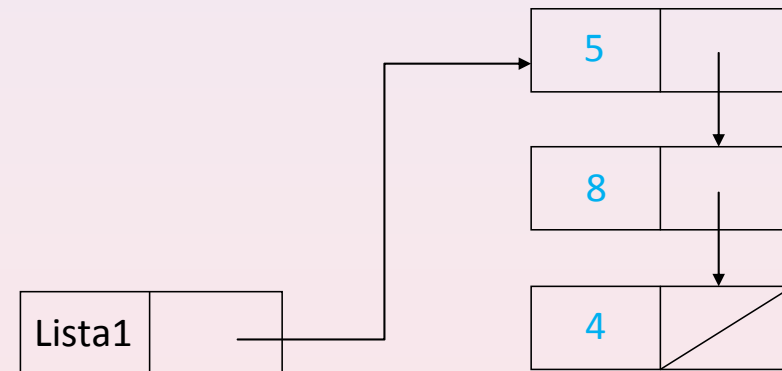
Se la lista non è vuota stampa il primo elemento, quindi stampa la lista rimanente

```
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

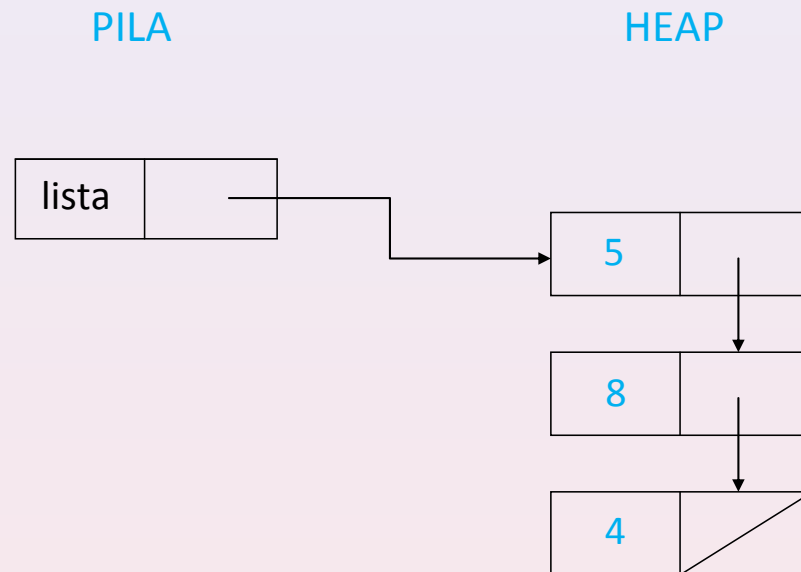
HEAP



Output

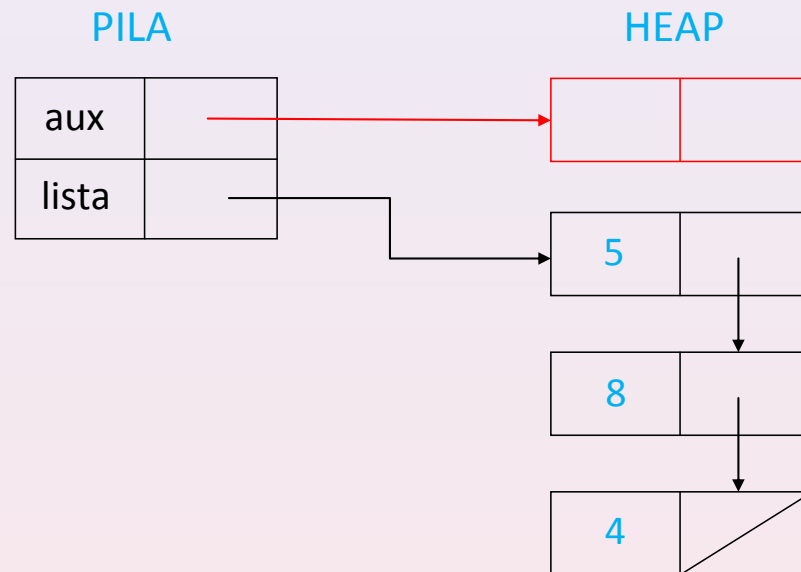
5 --> 8 --> 4 --> //

Inserimento di un nuovo elemento in testa



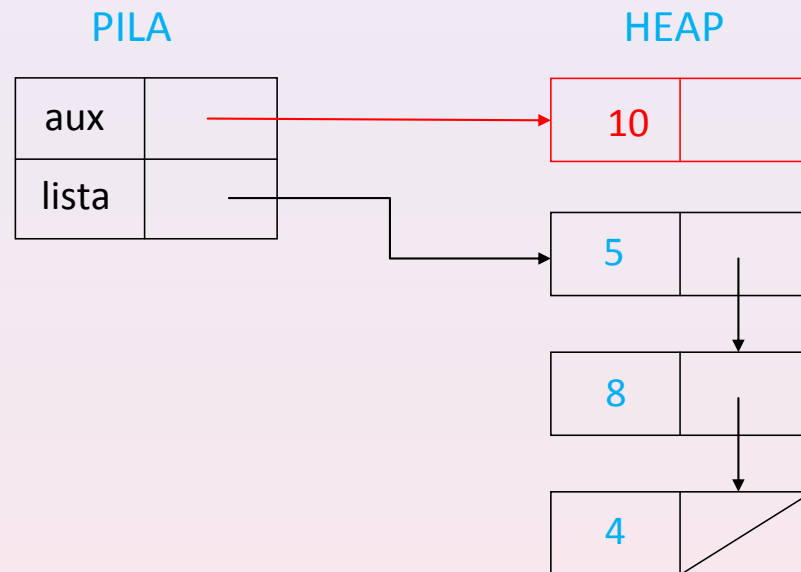
1. allochiamo una nuova struttura per l'elemento (`malloc`)
2. assegniamo il valore da inserire al campo `info` della struttura
3. concateniamo la nuova struttura con la vecchia lista
4. il puntatore iniziale della lista punta alla nuova struttura
⇒ la lista da modificare deve essere passata per `indirizzo`

Inserimento di un nuovo elemento in testa



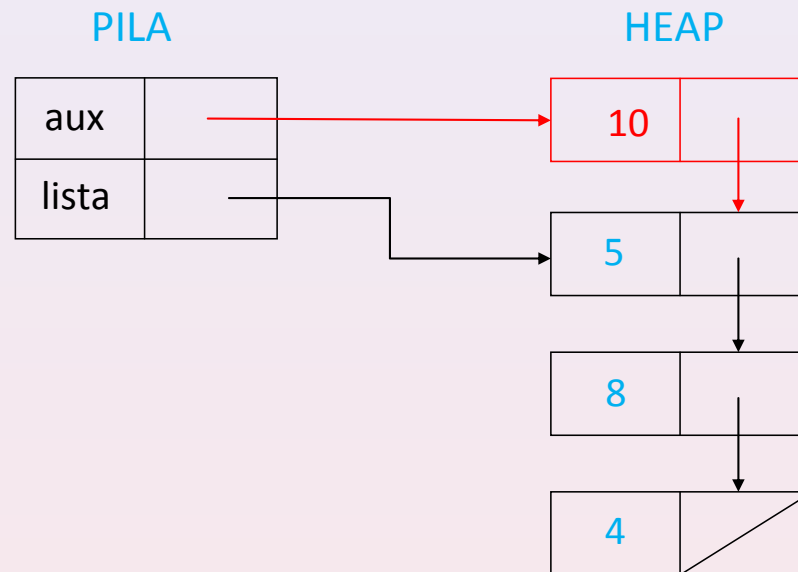
1. allochiamo una nuova struttura per l'elemento (`malloc`)
2. assegniamo il valore da inserire al campo `info` della struttura
3. concateniamo la nuova struttura con la vecchia lista
4. il puntatore iniziale della lista punta alla nuova struttura
⇒ la lista da modificare deve essere passata per `indirizzo`

Inserimento di un nuovo elemento in testa



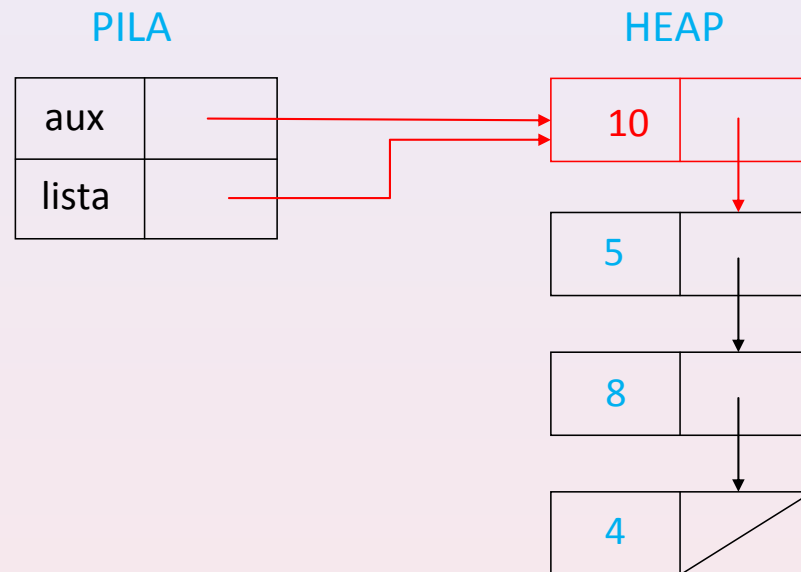
1. allochiamo una nuova struttura per l'elemento (`malloc`)
2. assegniamo il valore da inserire al campo `info` della struttura
3. concateniamo la nuova struttura con la vecchia lista
4. il puntatore iniziale della lista punta alla nuova struttura
⇒ la lista da modificare deve essere passata per `indirizzo`

Inserimento di un nuovo elemento in testa



1. allochiamo una nuova struttura per l'elemento (`malloc`)
2. assegniamo il valore da inserire al campo `info` della struttura
3. concateniamo la nuova struttura con la vecchia lista
4. il puntatore iniziale della lista punta alla nuova struttura
⇒ 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. assegniamo il valore da inserire al campo `info` della struttura
3. concateniamo la nuova struttura con la vecchia lista
4. il puntatore iniziale della lista punta alla nuova struttura
⇒ 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)

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)

Esercizio

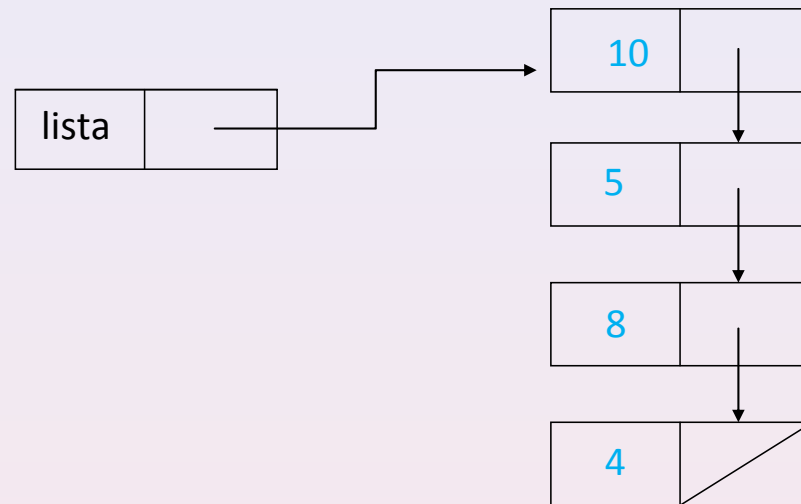
Impostare una chiamata alla procedura e tracciare l'evoluzione di pila e heap

- ▶ 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

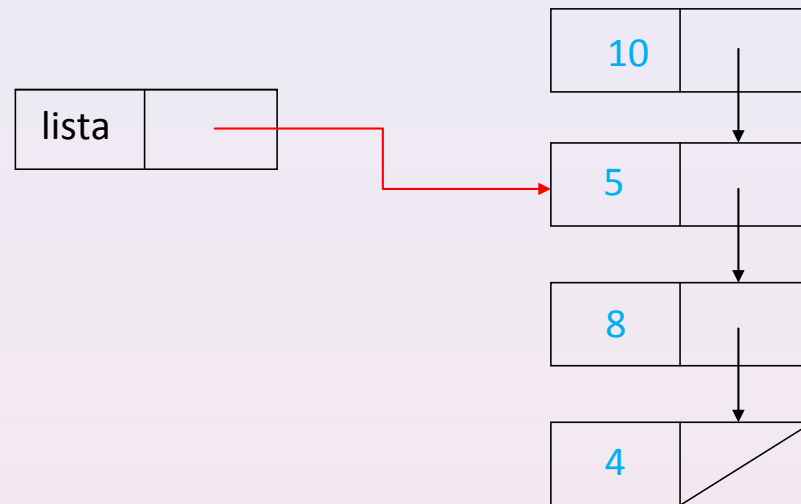
Impostare una chiamata alla procedura e tracciare l'evoluzione di pila e heap

Cancellazione del primo elemento HEAP



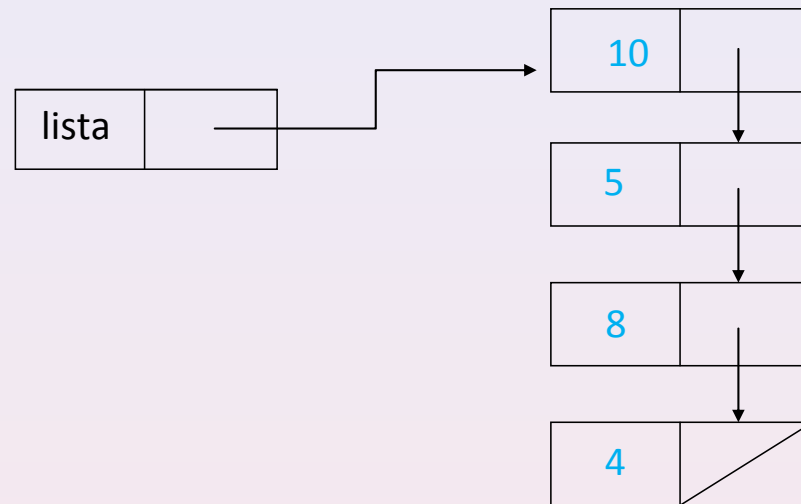
- ▶ se la lista è vuota non facciamo nulla
- ▶ altrimenti eliminiamo il primo elemento
 - ⇒ la lista deve essere passata per indirizzo
- ▶ **cancellare** significa anche **deallocare** la memoria occupata dall'elemento
 - ⇒ dobbiamo invocare **free** passando il puntatore all'elemento da cancellare
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

Cancellazione del primo elemento HEAP



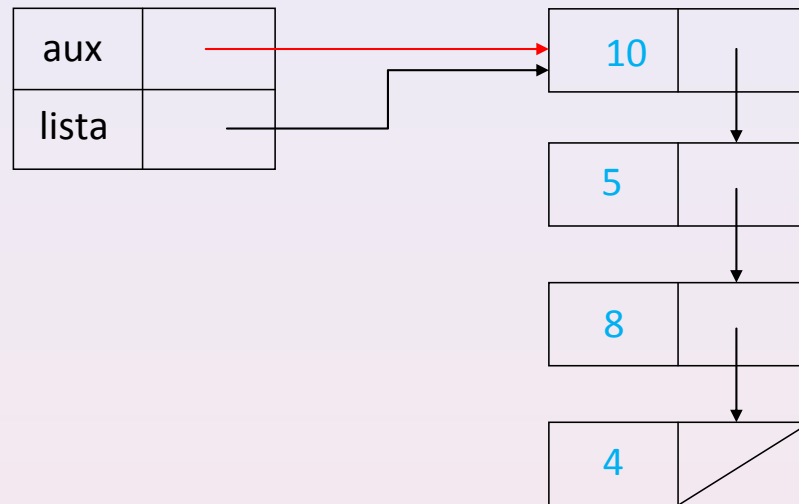
- ▶ se la lista è vuota non facciamo nulla
- ▶ altrimenti eliminiamo il primo elemento
 - ⇒ la lista deve essere passata per indirizzo
- ▶ **cancellare** significa anche **deallocare** la memoria occupata dall'elemento
 - ⇒ dobbiamo invocare **free** passando il puntatore all'elemento da cancellare
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

Cancellazione del primo elemento HEAP



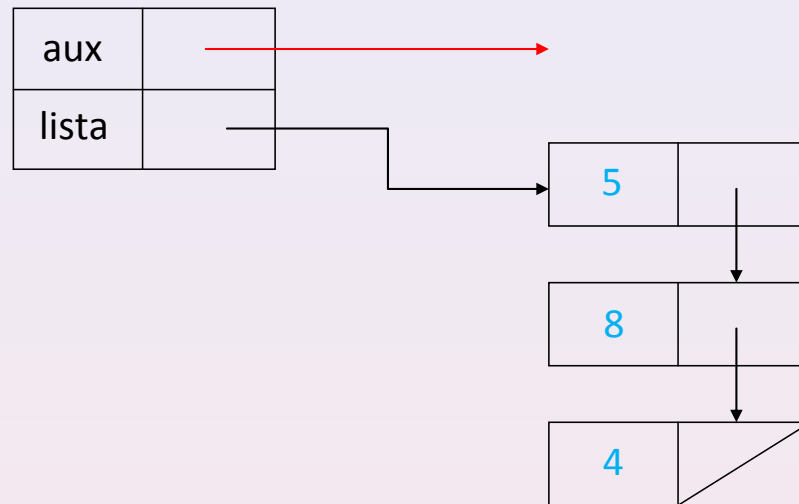
- ▶ se la lista è vuota non facciamo nulla
- ▶ altrimenti eliminiamo il primo elemento
 - ⇒ la lista deve essere passata per indirizzo
- ▶ **cancellare** significa anche **deallocare** la memoria occupata dall'elemento
 - ⇒ dobbiamo invocare **free** passando il puntatore all'elemento da cancellare
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

Cancellazione del primo elemento HEAP



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

Cancellazione del primo elemento HEAP



- ▶ se la lista è vuota non facciamo nulla
- ▶ altrimenti eliminiamo il primo elemento
 - ⇒ la lista deve essere passata per indirizzo
- ▶ **cancellare** significa anche **deallocare** la memoria occupata dall'elemento
 - ⇒ dobbiamo invocare **free** passando il puntatore all'elemento da cancellare
 - ⇒ 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);
        }
}

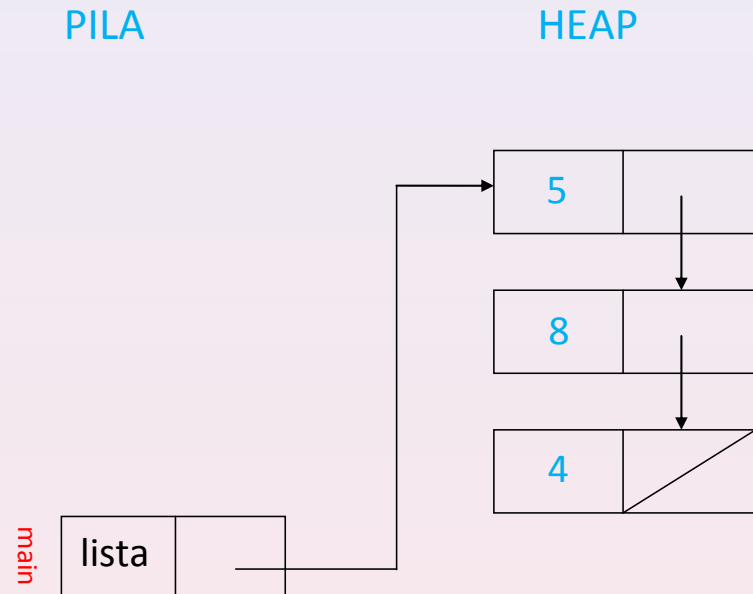
void CancellaLista(ListaDiElementi *lista)
{
    while (*lista != NULL)
        CancellaPrimo(lista);
}

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);
    ...
}
```



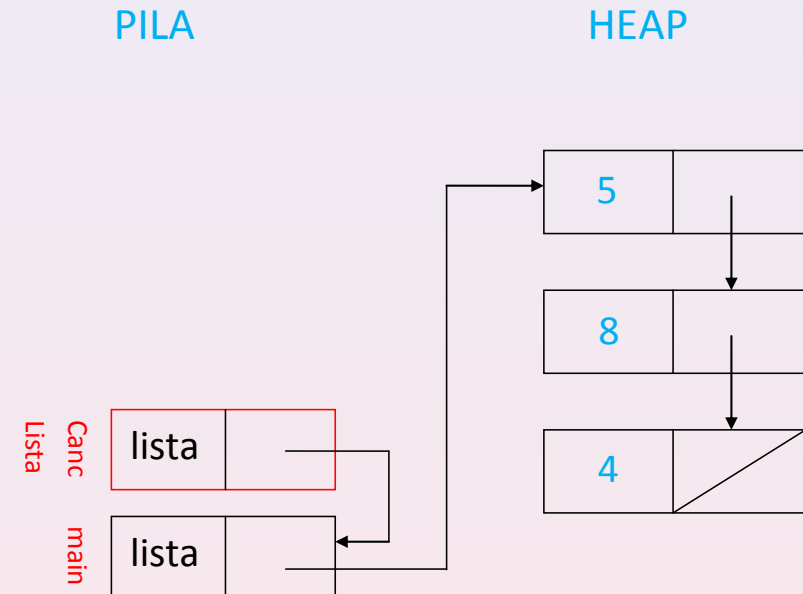

```

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);
    ...
}

```



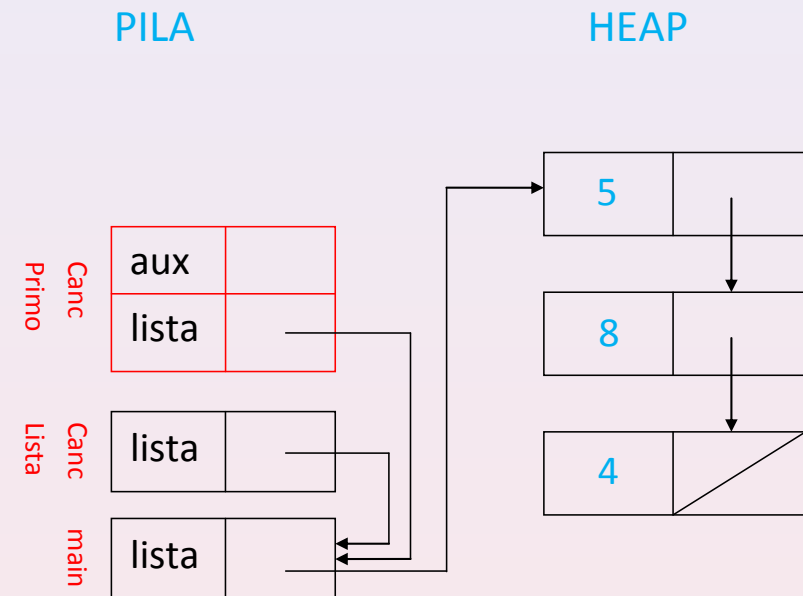
```

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);
    ...
}

```



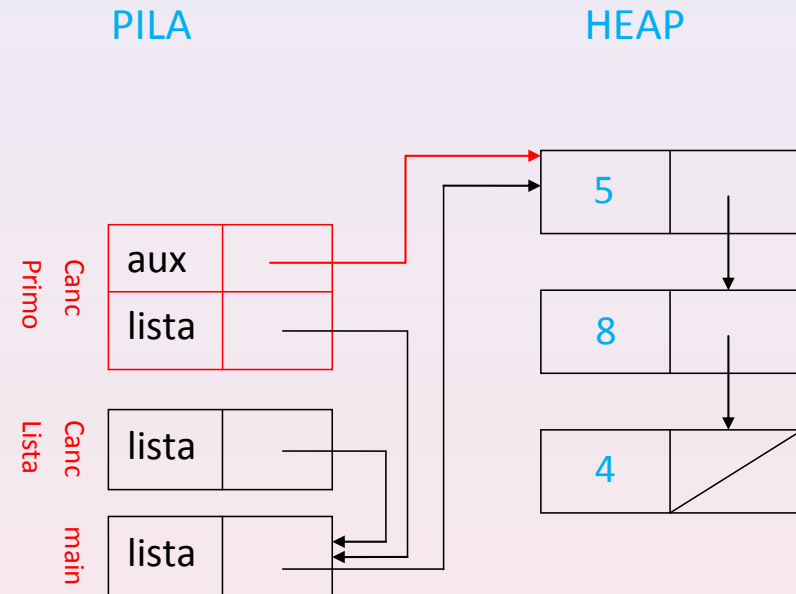
```

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);
    ...
}

```



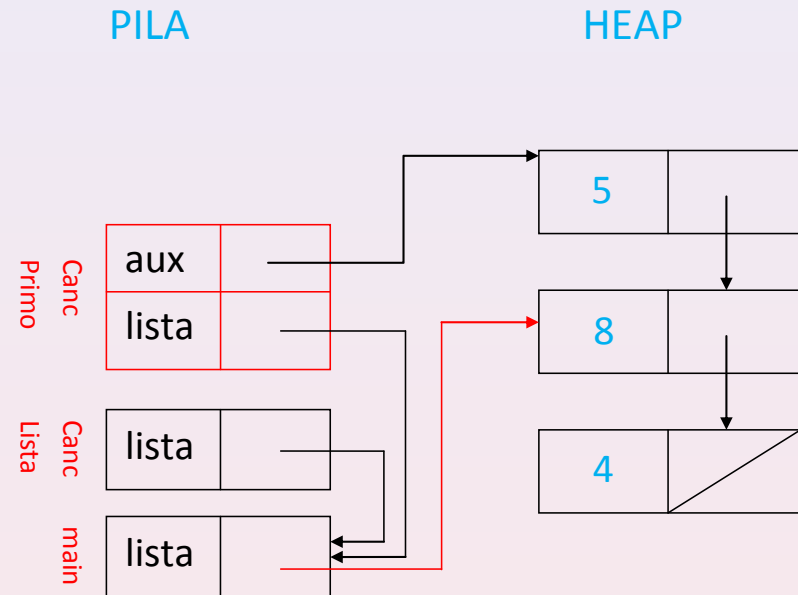
```

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);
        }
}

```

```

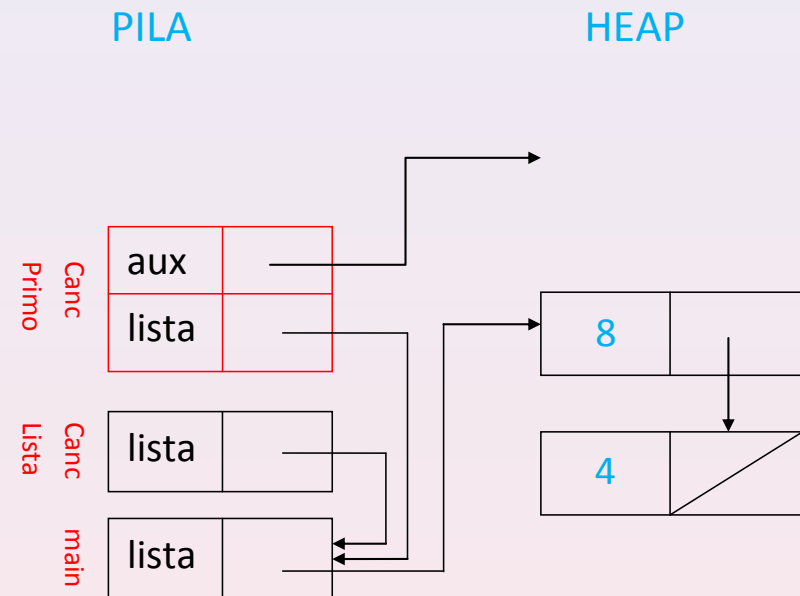
void CancellaLista(ListaDiElementi *lista)
{
    while (*lista != NULL)
        CancellaPrimo(lista);
}

```

```

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

```



```

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

```

```

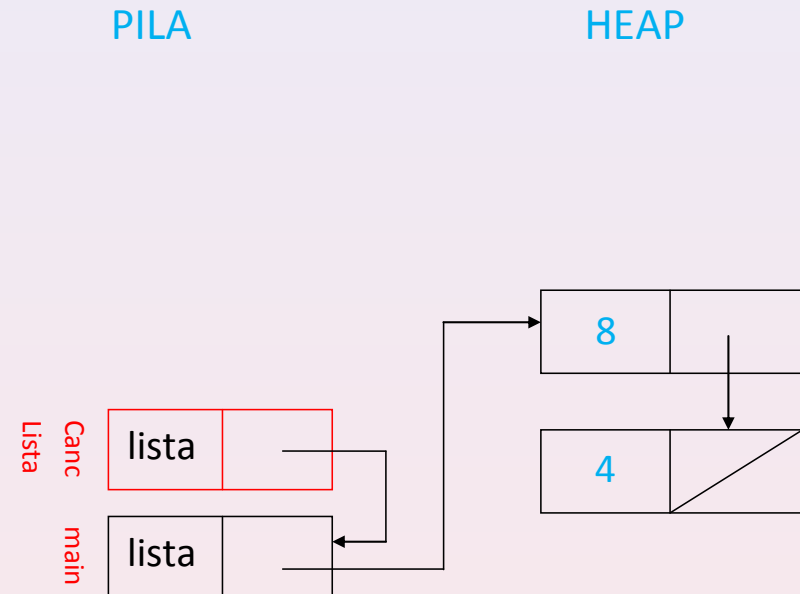
void CancellaLista(ListaDiElementi *lista)
{
    while (*lista != NULL)
        CancellaPrimo(lista);
}

```

```

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

```



```

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

```

```

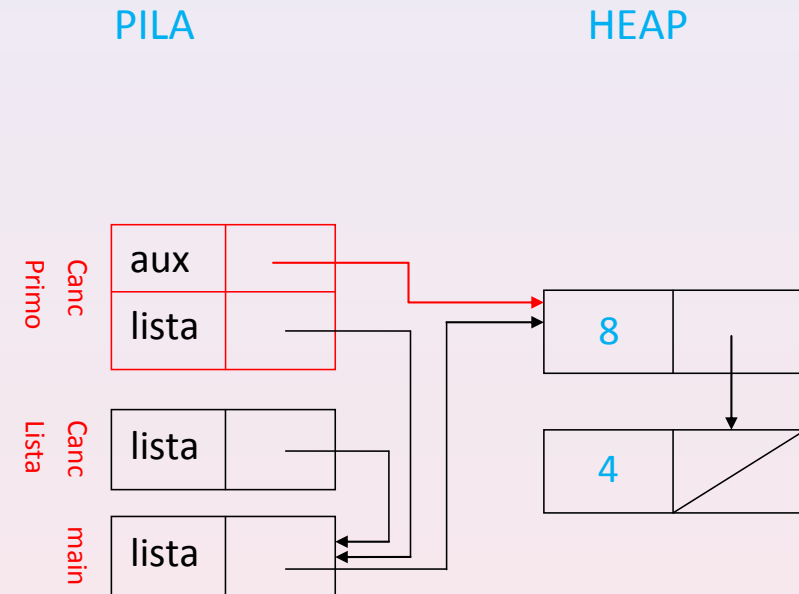
void CancellaLista(ListaDiElementi *lista)
{
    while (*lista != NULL)
        CancellaPrimo(lista);
}

```

```

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

```



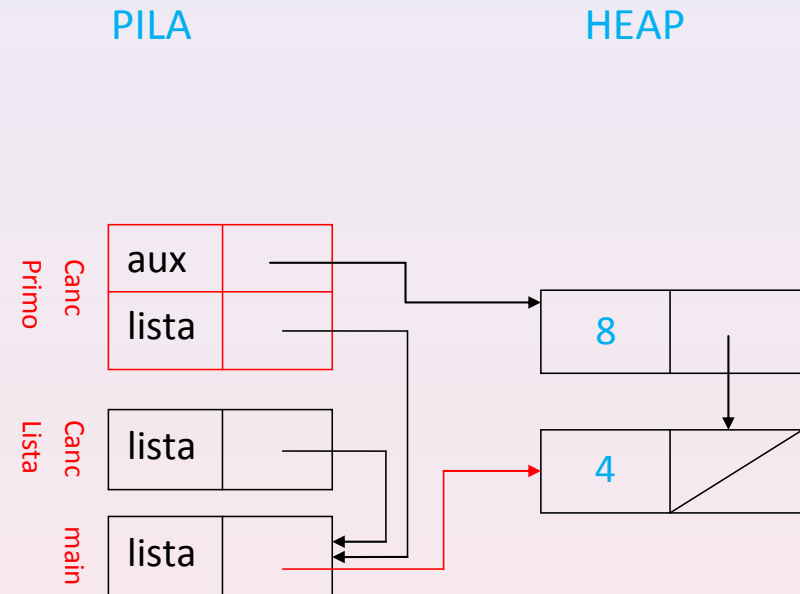
```

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);
    ...
}

```



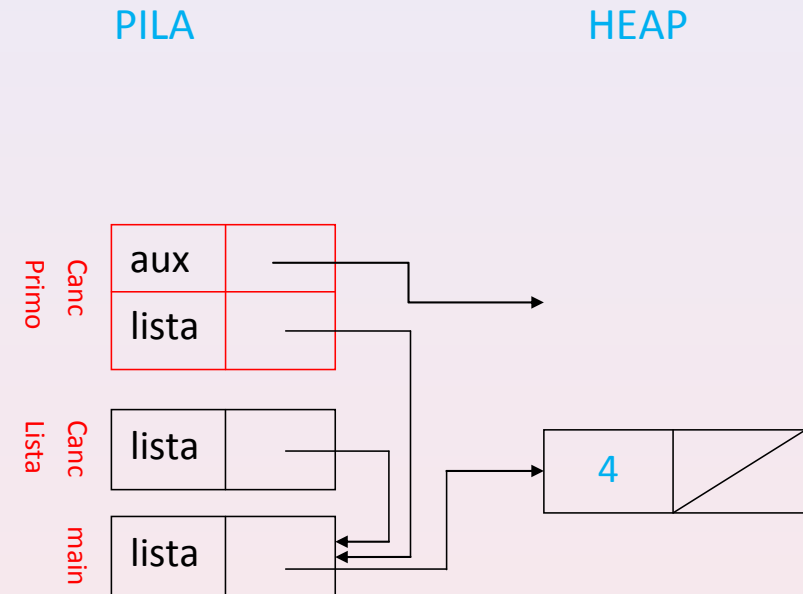

```

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);
}

```

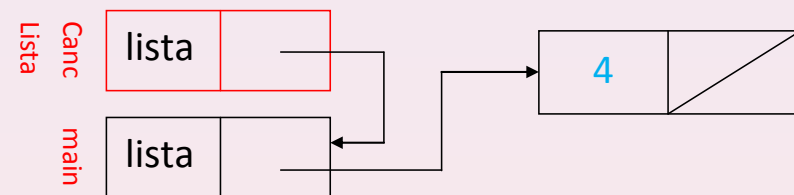
```

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

```

PILA

HEAP



```

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

```

```

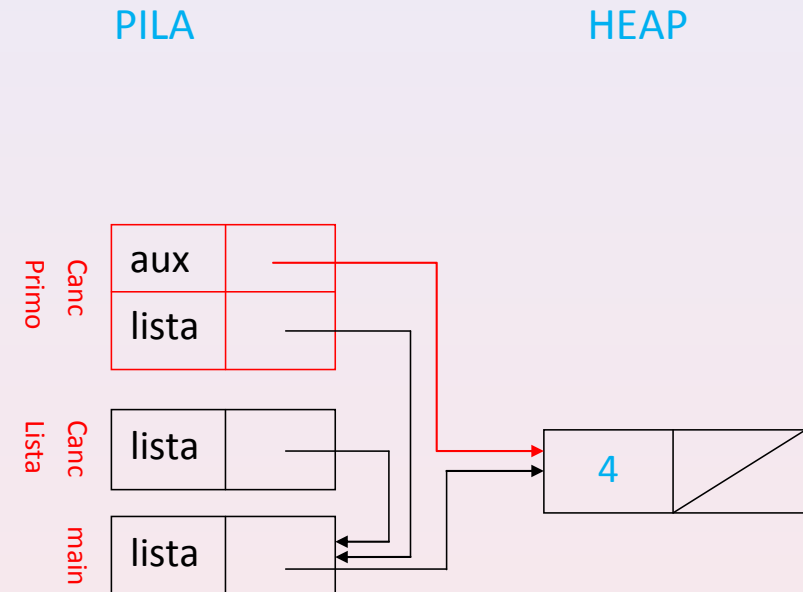
void CancellaLista(ListaDiElementi *lista)
{
    while (*lista != NULL)
        CancellaPrimo(lista);
}

```

```

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

```



```

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

```

```

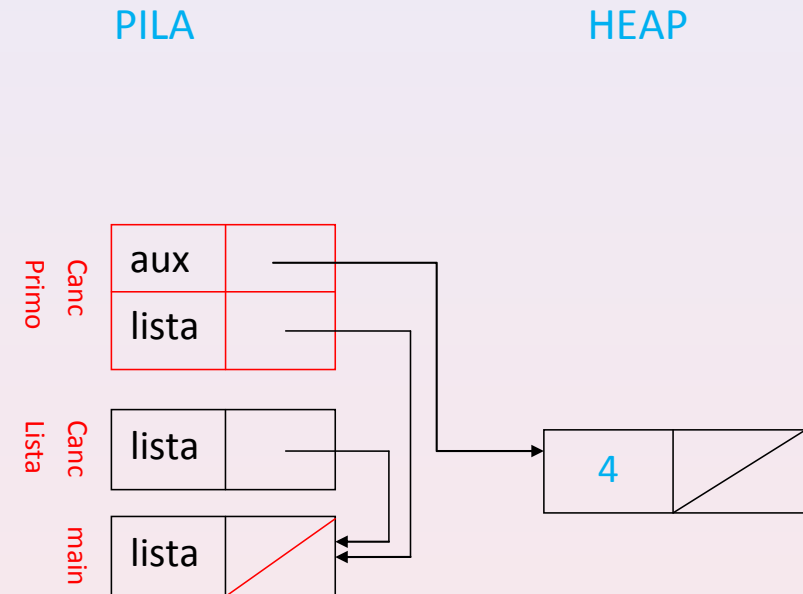
void CancellaLista(ListaDiElementi *lista)
{
    while (*lista != NULL)
        CancellaPrimo(lista);
}

```

```

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

```



```

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

```

```

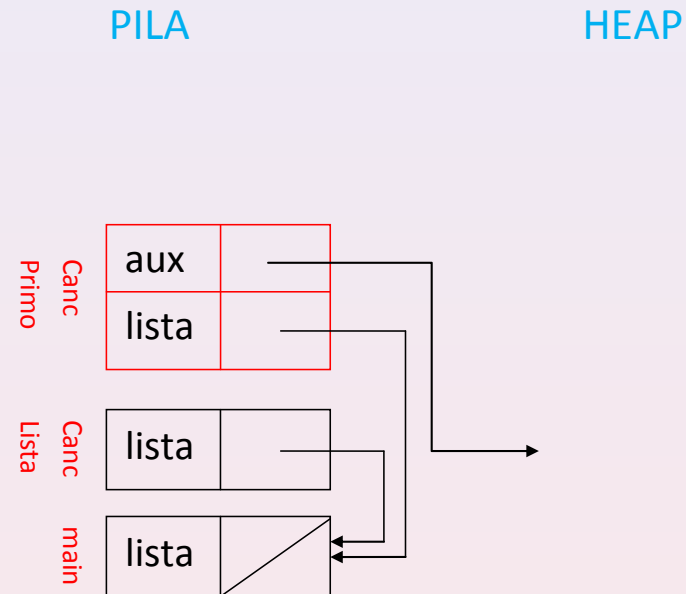
void CancellaLista(ListaDiElementi *lista)
{
    while (*lista != NULL)
        CancellaPrimo(lista);
}

```

```

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);
}

```

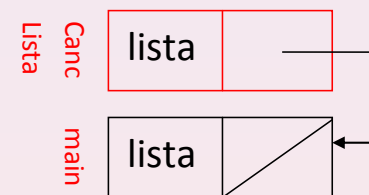
```

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

```

PILA

HEAP



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

PILA

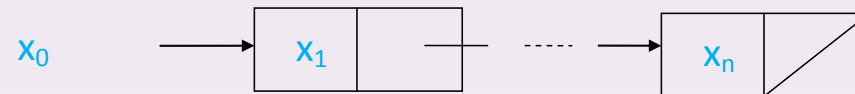
HEAP

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



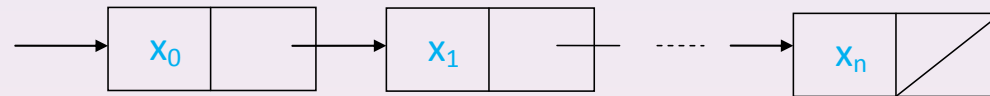
```
main()
{
    ListaDiElementi lista;
    ...
    CancellaLista(&lista);
    ...
}
```

Visione ricorsiva delle liste



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

Visione ricorsiva delle liste



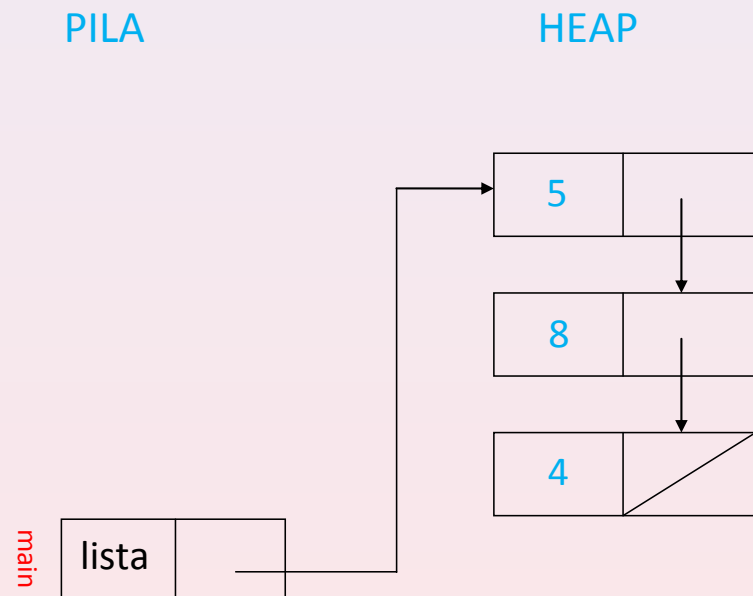
- ▶ Una lista di elementi è una struttura dati ricorsiva per sua natura
 1. data una lista L di elementi x_1, \dots, x_n
 2. dato un ulteriore elemento x_0
 3. anche la **concatenazione** di x_0 e L è una lista
- ▶ Si noti che in 1. L può anche essere la lista vuota

Cancellazione lista: versione ricorsiva

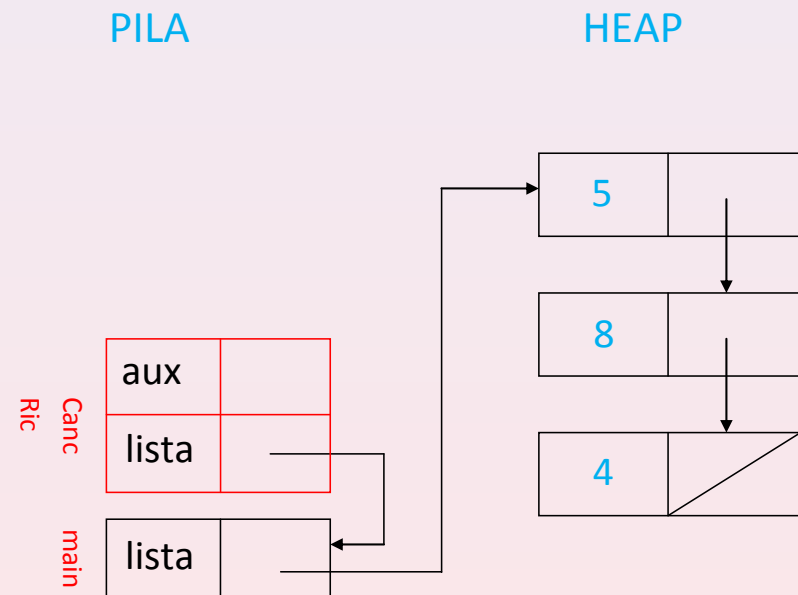
- ▶ Sfruttiamo la visione ricorsiva della struttura dati lista per realizzare la cancellazione in modo **ricorsivo**
 1. la cancellazione della lista vuota non richiede alcuna azione
 2. la cancellazione della lista ottenuta come concatenazione dell'elemento x e della lista L richiede l'eliminazione 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);
    }
}
```



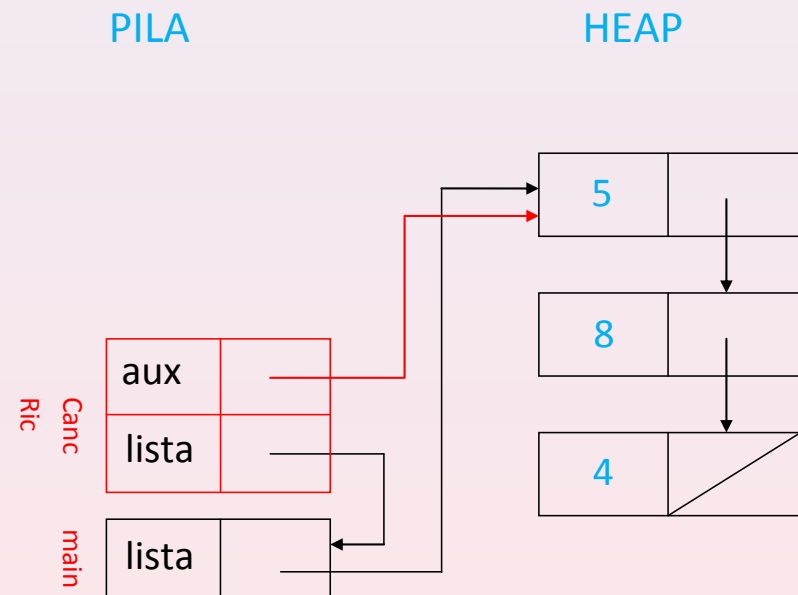
```
void CancellaListaRic(ListaDiElementi *lista)
{
    ListaDiElementi aux;
    if (*lista != NULL)
    {
        aux = *lista;
        *lista = (*lista)->next;
        free(aux);
        CancellaListaRic(lista);
    }
}
```



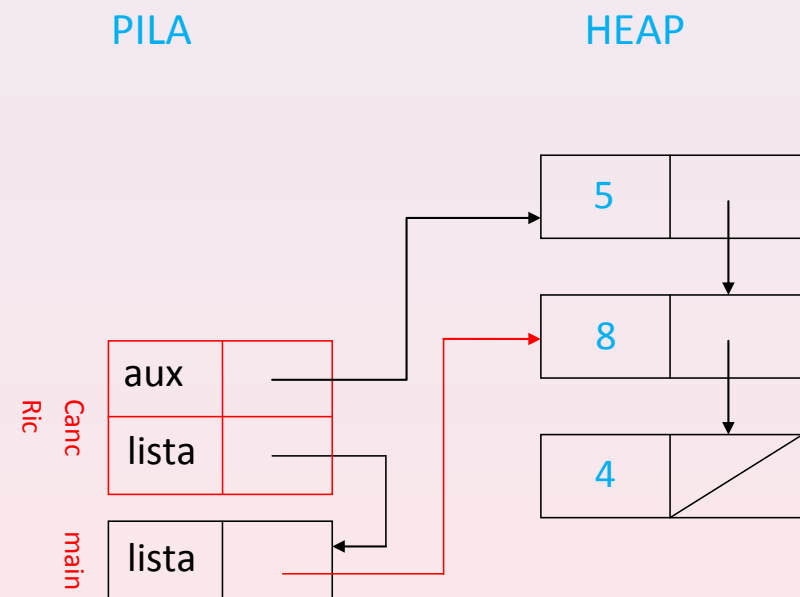
```

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

```



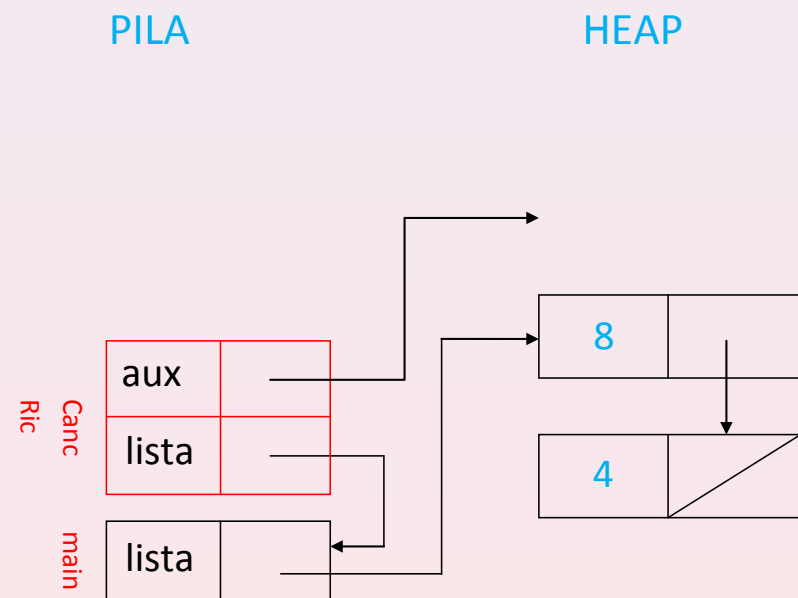
```
void CancellaListaRic(ListaDiElementi *lista)
{
    ListaDiElementi aux;
    if (*lista != NULL)
    {
        aux = *lista;
        *lista = (*lista)->next;
        free(aux);
        CancellaListaRic(lista);
    }
}
```



```

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

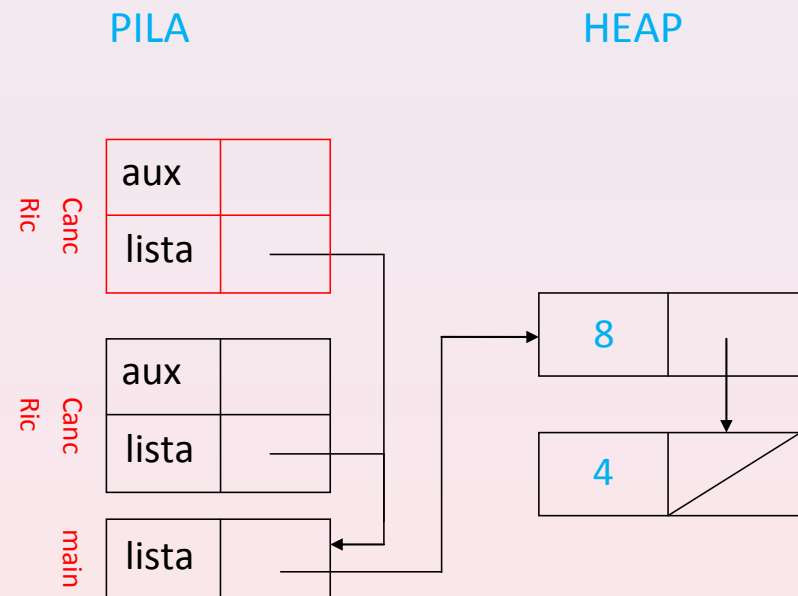
```



```

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

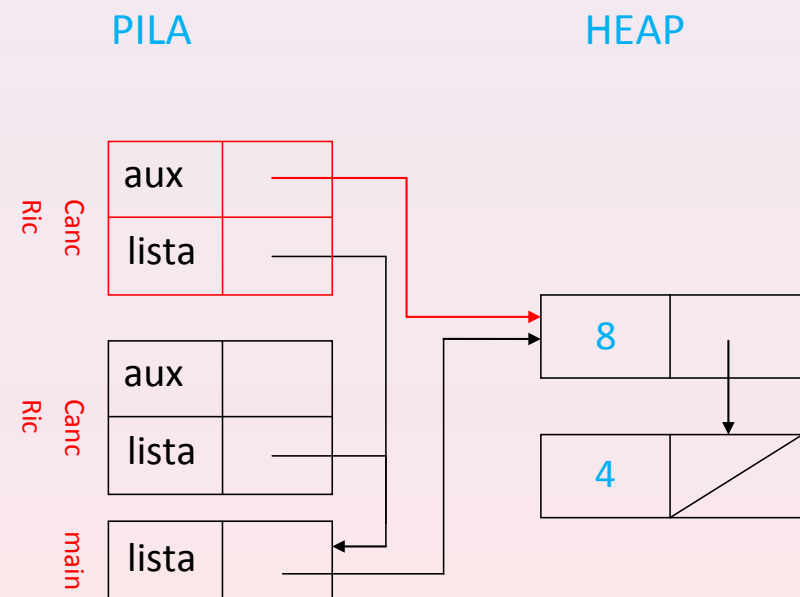
```




```

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

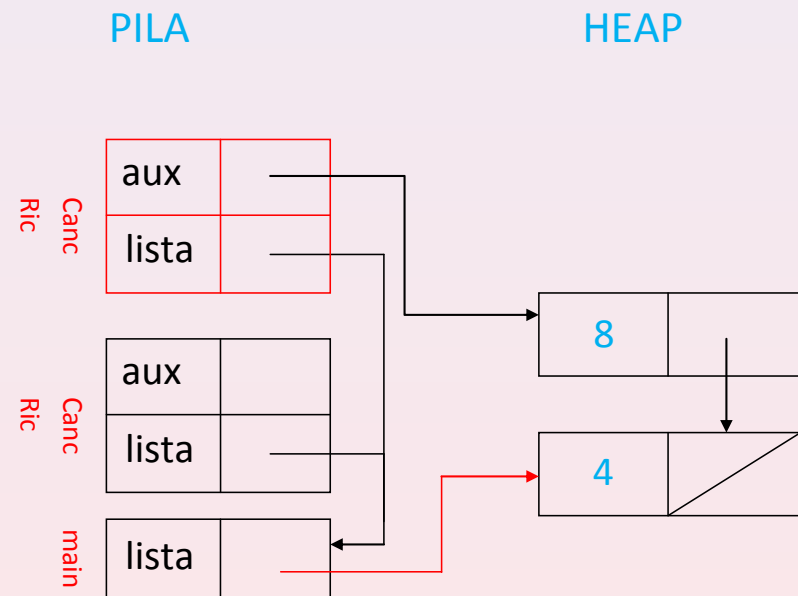
```



```

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

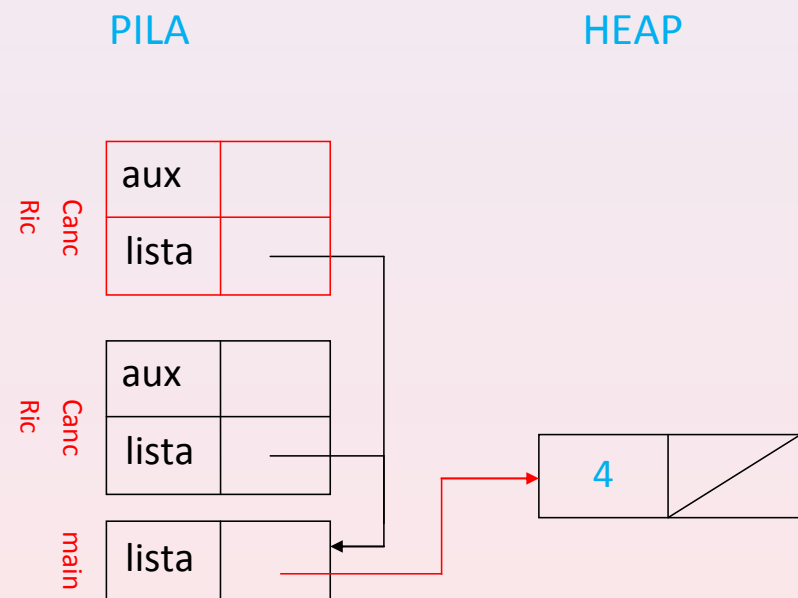
```



```

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

```

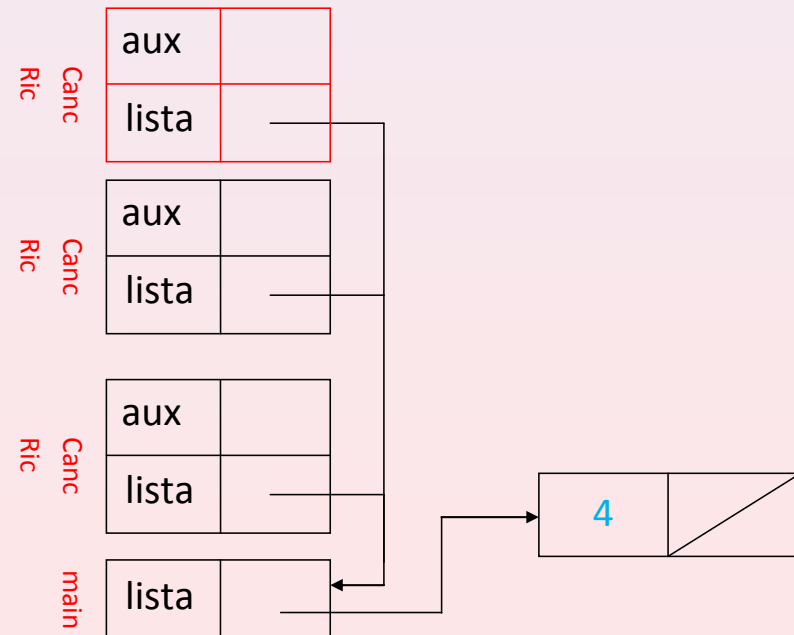


```

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

PILA

HEAP

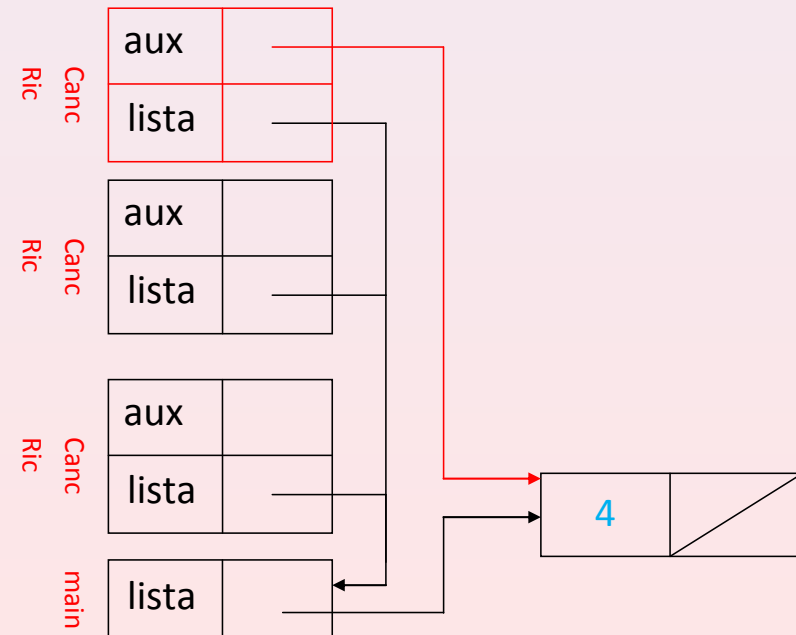


```

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

PILA

HEAP

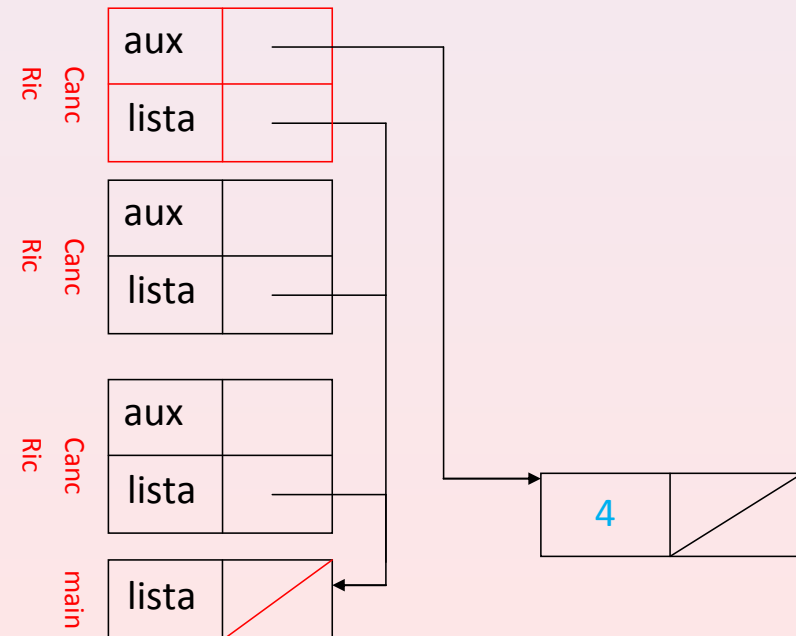


```

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

PILA

HEAP

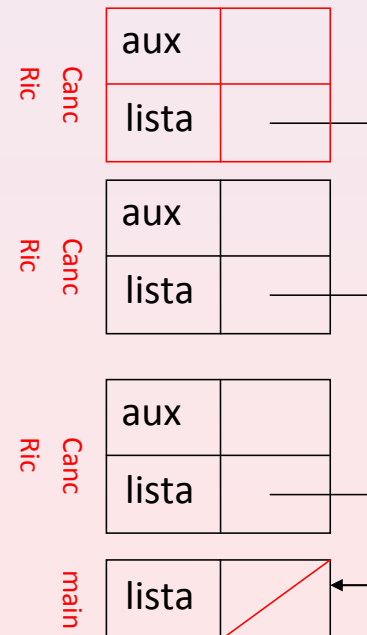


```

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

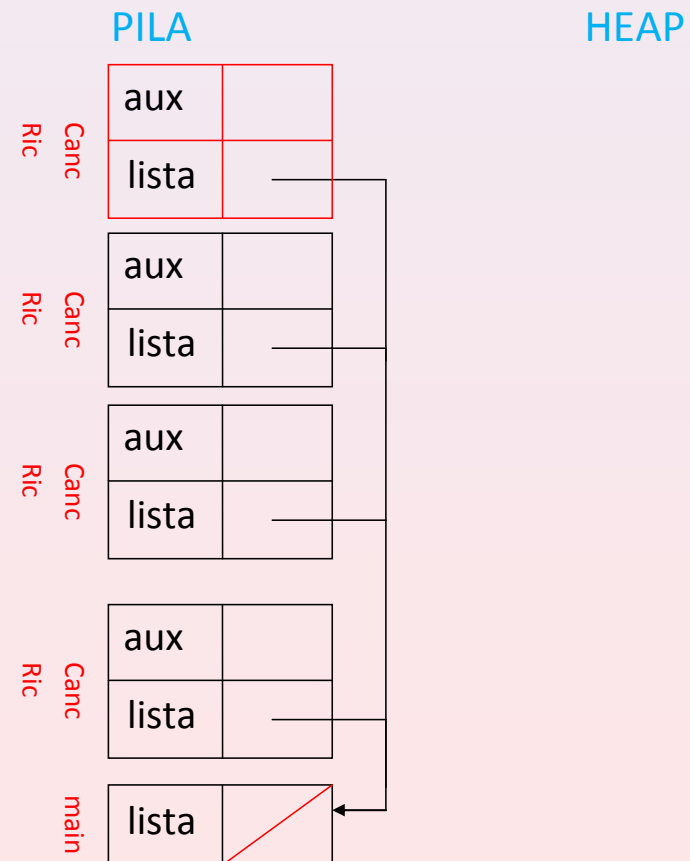
PILA

HEAP



```

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

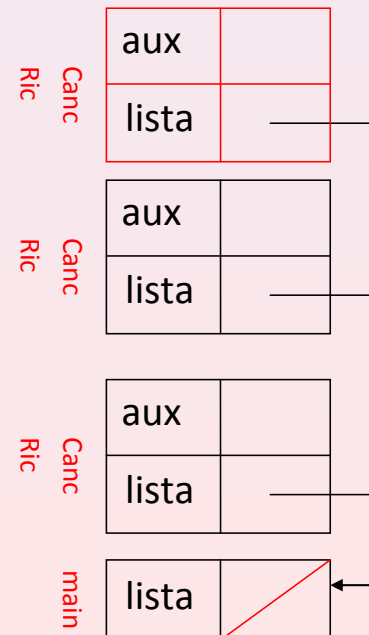



```

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

PILA

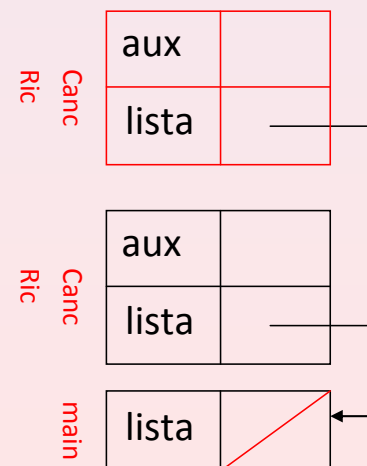
HEAP



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

PILA

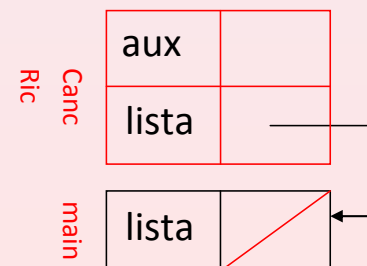
HEAP



```
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



Ricerca di un elemento in una lista

- ▶ Ricordiamo la ricerca lineare incerta su vettori

```
i = 0;      /* indice del primo elemento */
trovato = false;

while (i < DIM && ! trovato)
{
    if (vet[i] == elem)    /* elemento corrente */
        trovato = true;
    else
        i = i + 1;
}
```

- ▶ sostituiamo l'indice **i** con un puntatore alla lista che ci permette di scorrerla
- ▶ Incapsuliamo questo codice in una funzione a valori booleani

Ricerca di un elemento in una lista

Esempio: Versione Iterativa

```
boolean Ricerca(ListaDiElementi lis, TipoElementoLista elem)
{
    boolean trovato = false;
    while (lis != NULL && ! trovato)
    {
        if (lis->info == elem)
            trovato = true;
        else
            lis = lis->next;
    }
    return trovato;
}
```

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

Ricerca di un elemento in una lista

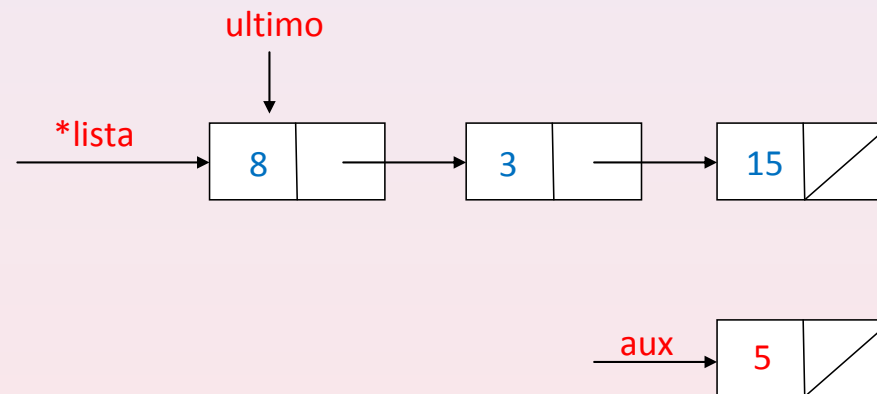
Esempio: Versione Ricorsiva

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

- ▶ 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`

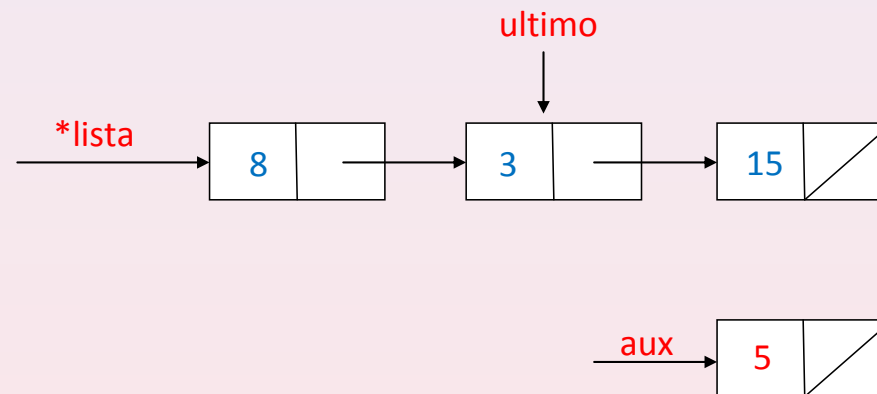
Inserimento di un elemento in coda

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



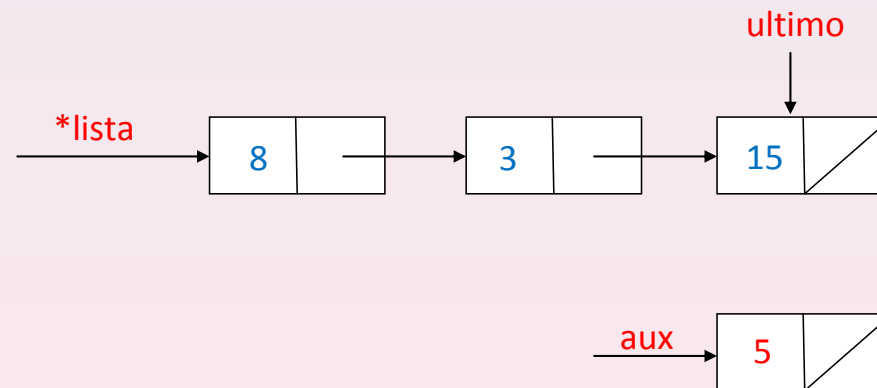
Inserimento di un elemento in coda

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



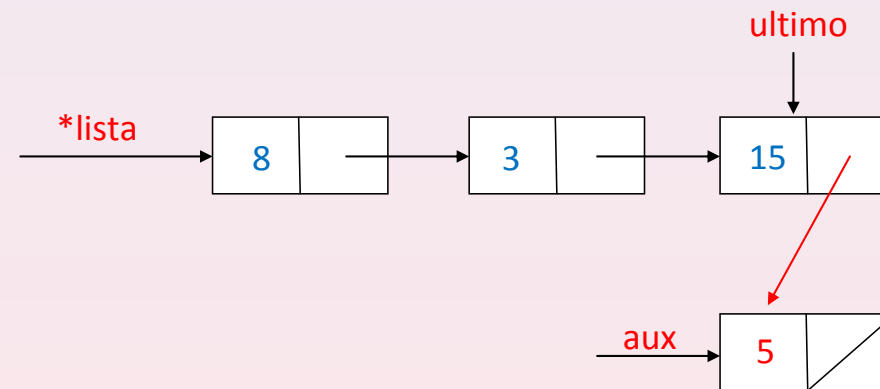
Inserimento di un elemento in coda

- ▶ Se la lista è vuota coincide con l'inserimento in testa
⇒ è necessario il passaggio per indirizzo!
- ▶ Se la lista non è vuota, bisogna scandirla fino in fondo
⇒ 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
⇒ è necessario il passaggio per indirizzo!
- ▶ Se la lista non è vuota, bisogna scandirla fino in fondo
⇒ 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**)

```
void InserzioneInCoda(ListaDiElementi *lista, TipoElementoLista elem)
{
    if (*lista == NULL)
    {
        *lista = malloc(sizeof(ElementoLista));
        (*lista)->info = elem;
        (*lista)->next = NULL;
    }
    else
        InserisciCodaLista(    ??    , elem);
}
```

Inserimento ricorsivo di un elemento in coda

```
void InserzioneInCoda(ListaDiElementi *lista, TipoElementoLista elem)
{
    if (*lista == NULL)
    {
        *lista = malloc(sizeof(ElementoLista));
        (*lista)->info = elem;
        (*lista)->next = NULL;
    }
    else
        InserisciCodaLista( (*lista)->next , elem);
}
```

Inserimento ricorsivo di un elemento in coda

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

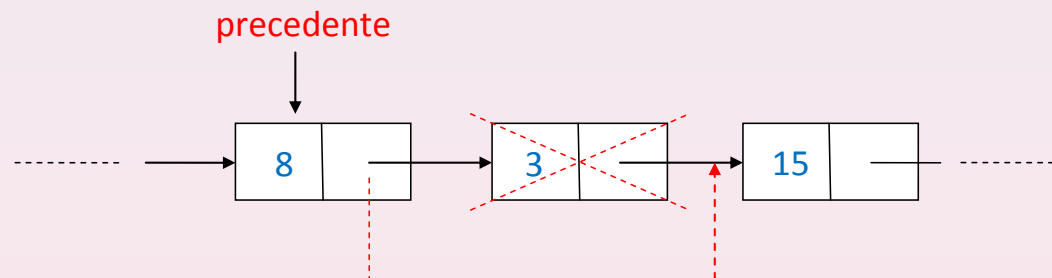
N.B.: Potremmo qui sostituire `*lista == NULL` con `ListaVuota(*lista)`

Cancellazione della prima occorrenza di un elemento

- ▶ si scandisce la lista alla ricerca dell'elemento
- ▶ se l'elemento non compare non si fa nulla
- ▶ altrimenti, a seconda di dove si trova l'elemento, si distinguono tre casi
 1. l'elemento è il primo della lista: si aggiorna il puntatore iniziale in modo che punti all'elemento successivo
⇒ passaggio per indirizzo!!
 2. l'elemento non è né il primo né l'ultimo: si aggiorna il campo `next` dell'elemento che precede quello da cancellare in modo che punti all'elemento che segue
 3. l'elemento è l'ultimo: come (2), solo che il campo `next` dell'elemento precedente viene posto a `NULL`
- ▶ in tutti e tre i casi bisogna liberare la memoria occupata dall'elemento da cancellare

Osservazioni:

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



- ▶ per fermare la scansione dopo aver trovato e cancellato l'elemento, si utilizza una sentinella booleana
- ▶ Seguendo questa idea, fare per esercizio la versione iterativa della cancellazione.

Versione iterativa:

```
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
 - ⇒ 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

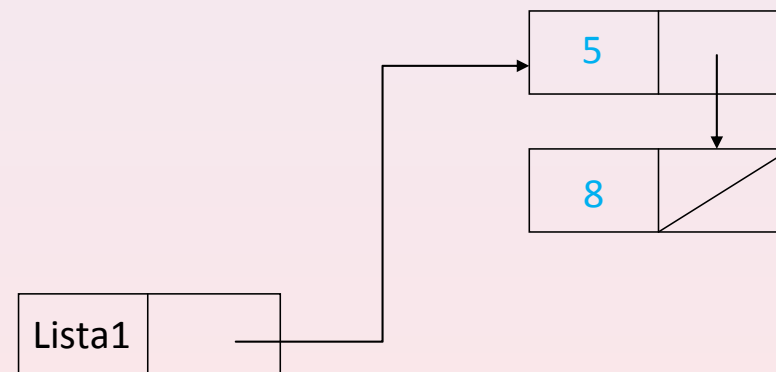
- ▶ Caratterizziamo il problema **induttivamente**
- ▶ Sia **ris** la lista ottenuta inserendo l'elemento **elem** nella lista ordinata **lista**.
 1. se **lista** è la lista vuota, allora **ris** è costituita solo da **elem** (**caso base**)
 2. se il primo elemento di **lista** è maggiore o uguale 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**)


```
void InserzioneOrdinata(ListaDiElementi *lista, int elem)
{
    if (*lista == NULL)
        InserisciTestaLista(lista, elem);
    else
        if ((*lista) --> info >= elem)
            InserisciTestaLista(lista, elem);
        else
            InserzioneOrdinata(&((*lista)->next), elem);
}
```

InserzioneOrdinata(&Lista1, 10)

PILA

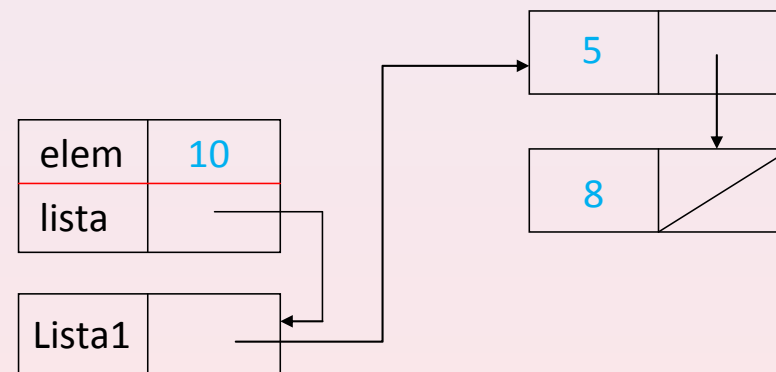
HEAP



InserzioneOrdinata(&Lista1, 10)

PILA

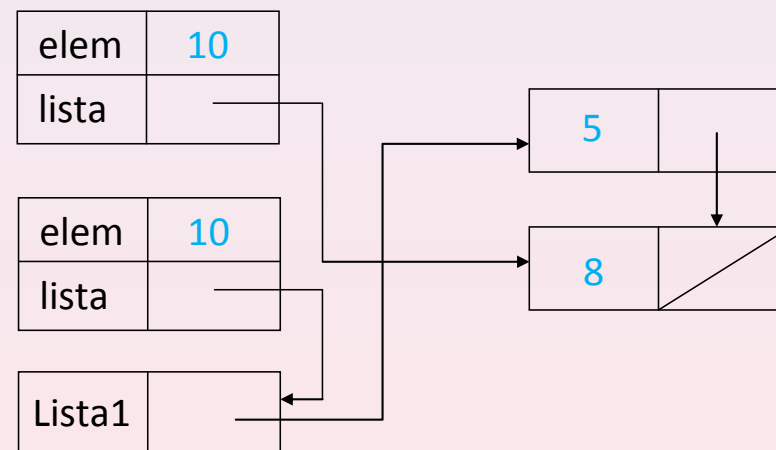
HEAP



InserzioneOrdinata(&Lista1, 10)

PILA

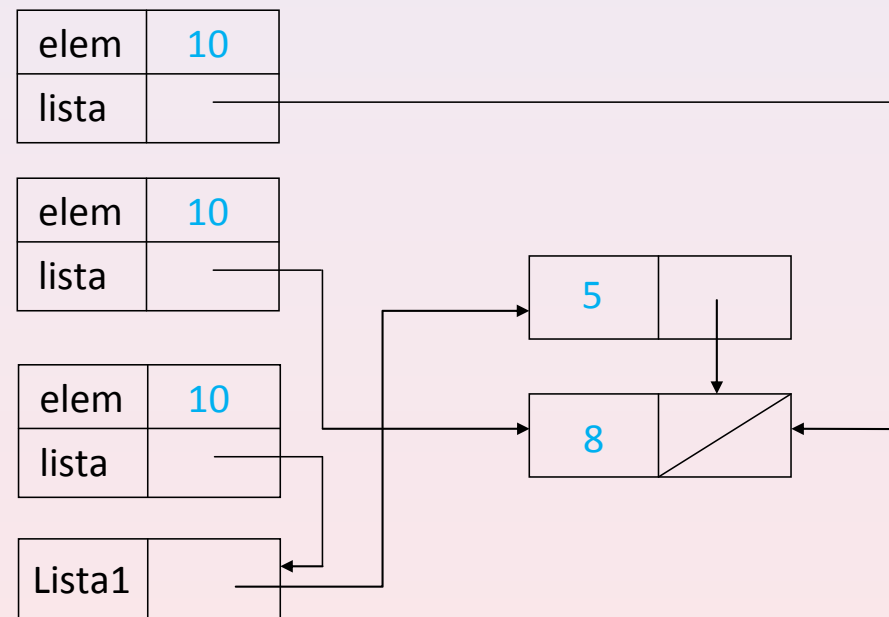
HEAP



InserzioneOrdinata(&Lista1, 10)

PILA

HEAP



InserzioneOrdinata(&Lista1, 10)

PILA

HEAP

