Operazioni sulle liste

- Definiamo una serie di procedure e funzioni per operare sulle liste.

Operazioni sulle liste

- Definiamo una serie di procedure e funzioni per operare sulle liste.
- Usiamo liste di interi ma tutte le operazioni si definiscono analogamente su liste di altro tipo

Operazioni sulle liste

- Definiamo una serie di procedure e funzioni per operare sulle liste.
- Usiamo liste di interi ma tutte le operazioni si definiscono analogamente su liste di altro tipo
- Facciamo riferimento alle dichiarazioni dei tipi viste a lezione

```
struct elemento {
  int info;
  struct elemento *next;
};
typedef struct elemento ElementoDiLista;
typedef ElementoDiLista *ListaDiElementi;
```

Inizializzazione

- Definiamo una procedura che inizializza una lista assegnando il valore `NULL` alla variabile testa della lista.

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.

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 come segue :

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


  int main()
  {
   ListaDiElementi Lista1;
   Inizializza(&Lista1);
   return 0;     }
```

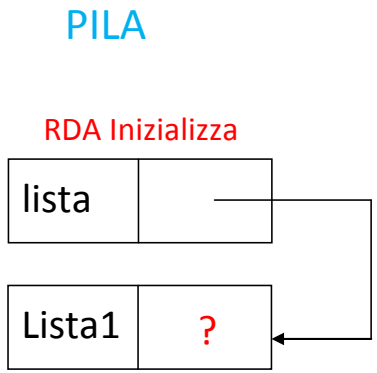
- Supponiamo ora che `Inizializza` sia chiamata come segue :

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


  int main()
  {
   ListaDiElementi Lista1;
   Inizializza(&Lista1);
   return 0;     }
```

- Supponiamo ora che `Inizializza` sia chiamata come segue :

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


  int main()
  {
   ListaDiElementi Lista1;
   Inizializza(&Lista1);
   return 0;     }
```

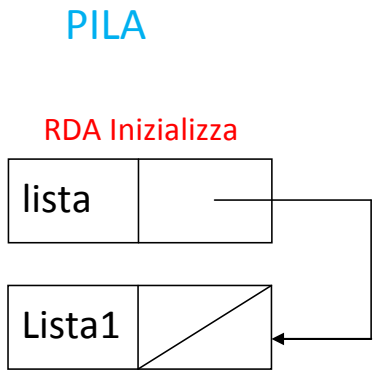
Ricorda che RDA e' il record di attivazione



- Supponiamo ora che `Inizializza` sia chiamata come segue :

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


  int main()
  {
   ListaDiElementi Lista1;
   Inizializza(&Lista1);
   return 0;     }
```

Ricorda che RDA e' il record di attivazione

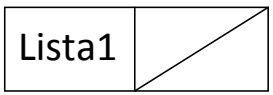


- Supponiamo ora che `Inizializza` sia chiamata come segue :

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


  int main()
  {
   ListaDiElementi Lista1;
   Inizializza(&Lista1);
   return 0;     }
```

Ricorda che RDA e' il record di attivazione

PILA



Cosa succederebbe se passassimo il parametro per valore?

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

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

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	?

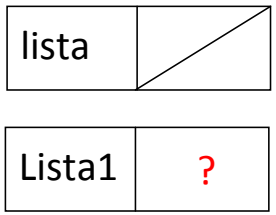
Lista1	?

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?

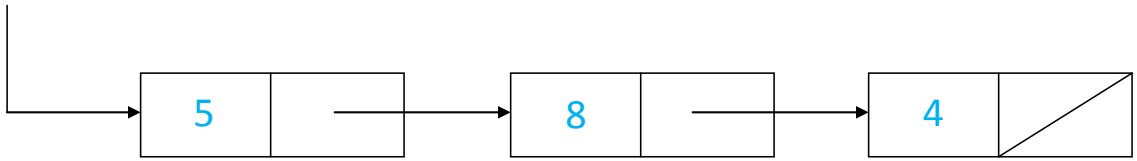
```
void Inizializza(ListaDiElementi lista)
{
  lista=NULL;
}

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

PILA

Lista1	?

Stampa degli elementi di una lista

- Data la lista



vogliamo che venga stampato:

5 –> 8 –> 4 –> //

Stile di programmazione

- In alcuni casi possiamo scegliere se definire funzioni o procedure che realizzino l'operazione principale sulle liste

Stile di programmazione

- In alcuni casi possiamo scegliere se definire funzioni o procedure che realizzino l'operazione principale sulle liste
- se decidiamo di usare le funzioni, la lista modificata dall'operazione viene restituita come valore di ritorno della funzione

Stile di programmazione

- In alcuni casi possiamo scegliere se definire funzioni o procedure che realizzino l'operazione principale sulle liste
- se decidiamo di usare le funzioni, la lista modificata dall'operazione viene restituita come valore di ritorno della funzione
- se decidiamo di usare una procedura, la lista modificata dall'operazione deve venire modificata direttamente dalla procedura

Stile di programmazione

- In alcuni casi possiamo scegliere se definire funzioni o procedure che realizzino l'operazione principale sulle liste
- se decidiamo di usare le funzioni, la lista modificata dall'operazione viene restituita come valore di ritorno della funzione
- se decidiamo di usare una procedura, la lista modificata dall'operazione deve venire modificata direttamente dalla procedura
- il primo approccio corrisponde ad uno stile di programmazione funzionale

Stile di programmazione

- In alcuni casi possiamo scegliere se definire funzioni o procedure che realizzino l'operazione principale sulle liste
- se decidiamo di usare le funzioni, la lista modificata dall'operazione viene restituita come valore di ritorno della funzione
- se decidiamo di usare una procedura, la lista modificata dall'operazione deve venire modificata direttamente dalla procedura
- il primo approccio corrisponde ad uno stile di programmazione funzionale
- il secondo ad uno stile di programmazione di linguaggio imperativo

Inserimento di un nuovo elemento in testa con una funzione

1. allochiamo una nuova struttura per l'elemento (`malloc`)

Inserimento di un nuovo elemento in testa con una funzione

1. allochiamo una nuova struttura per l'elemento (`malloc`)
2. assegnamo il valore da inserire al campo `info` della struttura

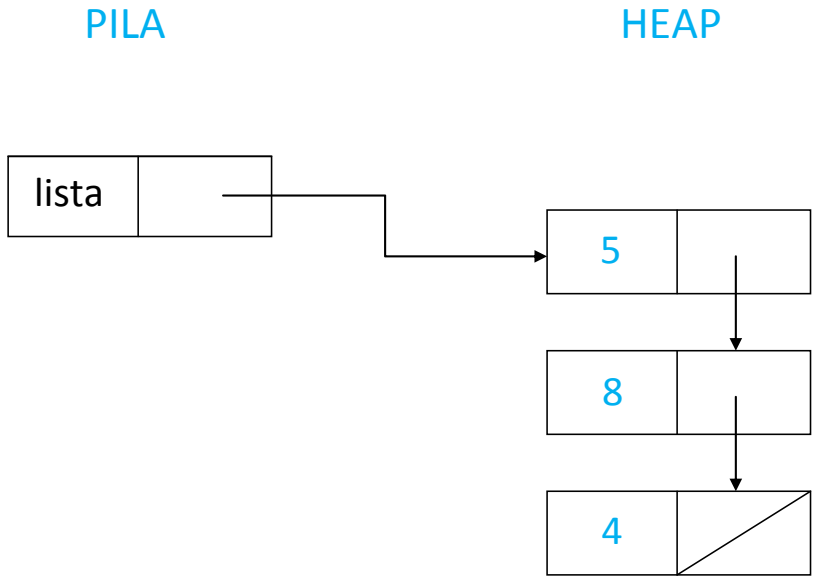
Inserimento di un nuovo elemento in testa con una funzione

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

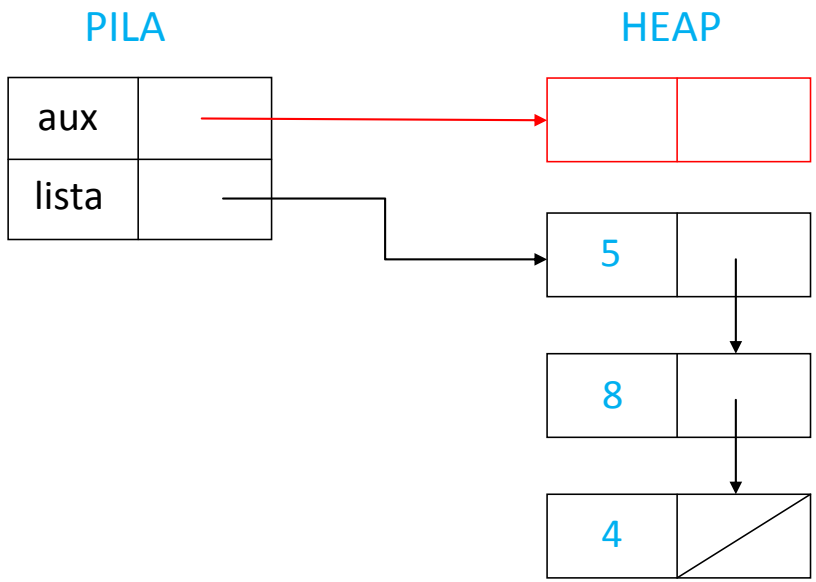
Inserimento di un nuovo elemento in testa con una funzione

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. ritorniamo il puntatore al nuovo elemento della lista

Inserimento di un nuovo elemento in testa con una funzione

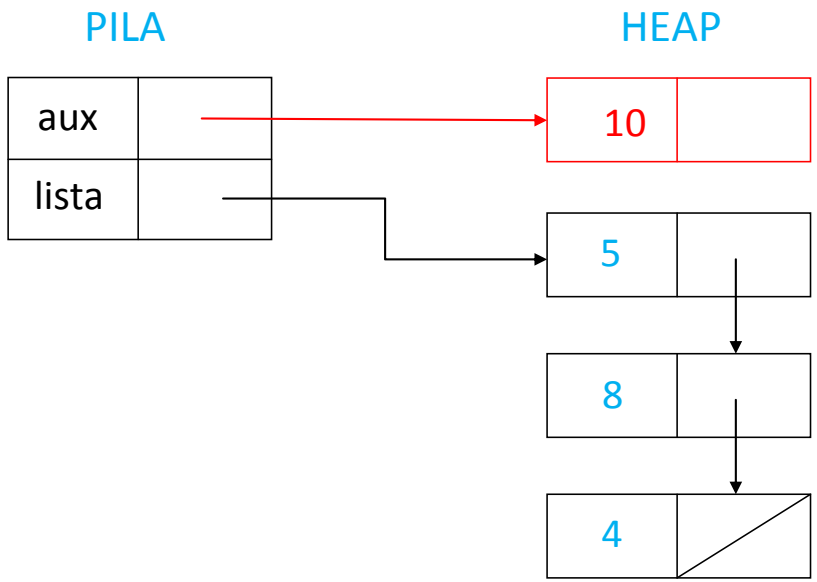


Inserimento di un nuovo elemento in testa con una funzione



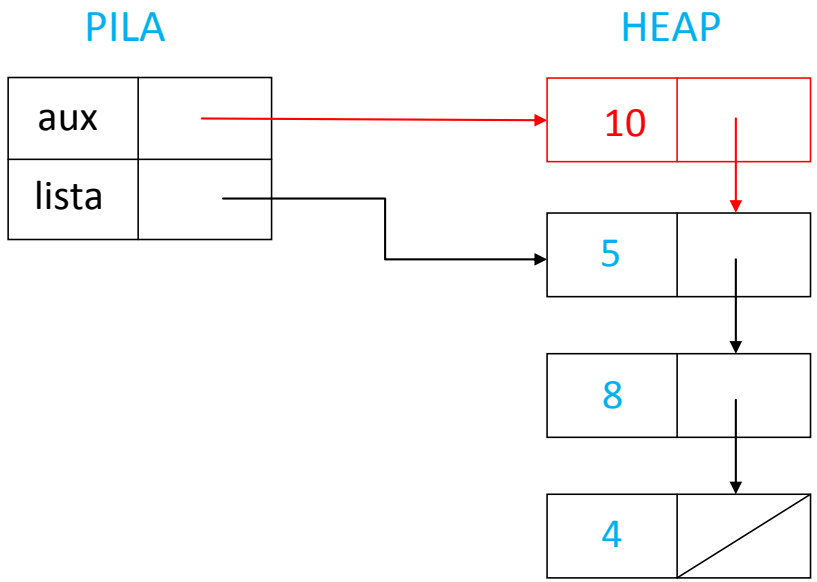
1. allochiamo una nuova struttura per l'elemento (`malloc`)

Inserimento di un nuovo elemento in testa con una funzione



1. allochiamo una nuova struttura per l'elemento (`malloc`)
2. assegnamo il valore da inserire al campo `info` della struttura

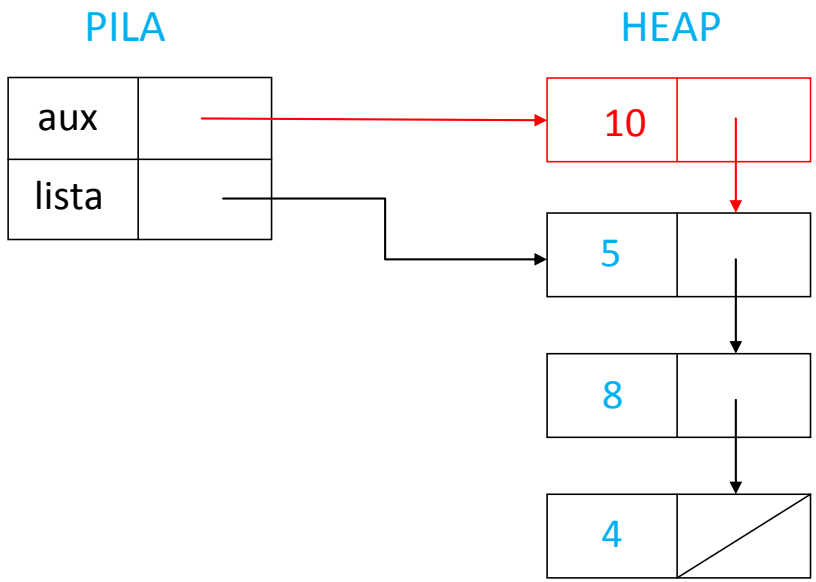
Inserimento di un nuovo elemento in testa con una funzione



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 e restituiamo il puntatore aux alla nuova struttura

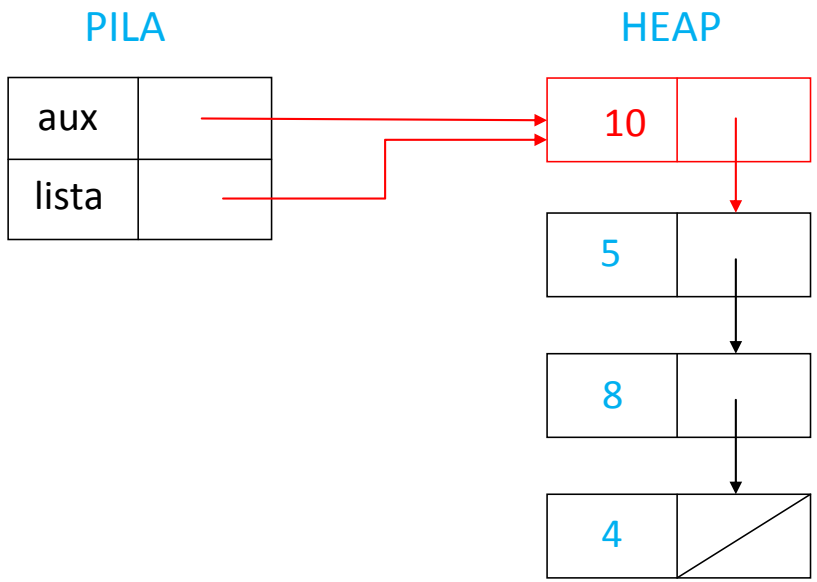
```
ListaDiElementi inserisci_testa ( ListaDiElementi lista, int  elem) {
   ListaDiElementi aux;
   aux = malloc(sizeof(ElementoDiLista));
   aux->info= elem;
   aux ->next = lista;
   return aux;}
```

Inserimento di un nuovo elemento in testa con una procedura



In questo caso il puntatore alla nuova struttura non deve essere restituito ma

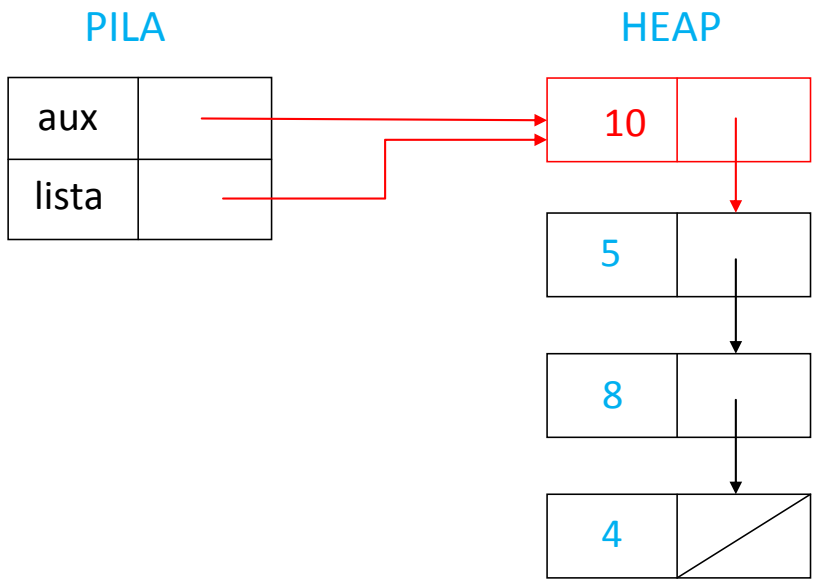
Inserimento di un nuovo elemento in testa con una procedura



In questo caso il puntatore alla nuova struttura non deve essere restituito ma

1. il puntatore iniziale della lista viene fatto puntare alla nuova struttura

Inserimento di un nuovo elemento in testa con una procedura



In questo caso il puntatore alla nuova struttura non deve essere restituito ma

1. il puntatore iniziale della lista viene fatto puntare alla nuova struttura
2. $\Longrightarrow$ la lista da modificare deve essere passata per indirizzo

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

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

- il primo parametro è la lista da modificare (passata per indirizzo)

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

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

- il primo parametro è la lista da modificare (passata per indirizzo)
- il secondo parametro è il campo info dell' elemento da inserire

Inserimento di un elemento in coda

- Se la lista è vuota coincide con l'inserimento in testa

Inserimento di un elemento in coda

- Se la lista è vuota coincide con l'inserimento in testa
  $\Longrightarrow$ se facciamo una procedura è necessario il passaggio per indirizzo!

Inserimento di un elemento in coda

- Se la lista è vuota coincide con l'inserimento in testa
  $\Longrightarrow$ se facciamo una procedura è necessario il passaggio per indirizzo!
- Se la lista non è vuota, bisogna scandirla fino in fondo

Inserimento di un elemento in coda

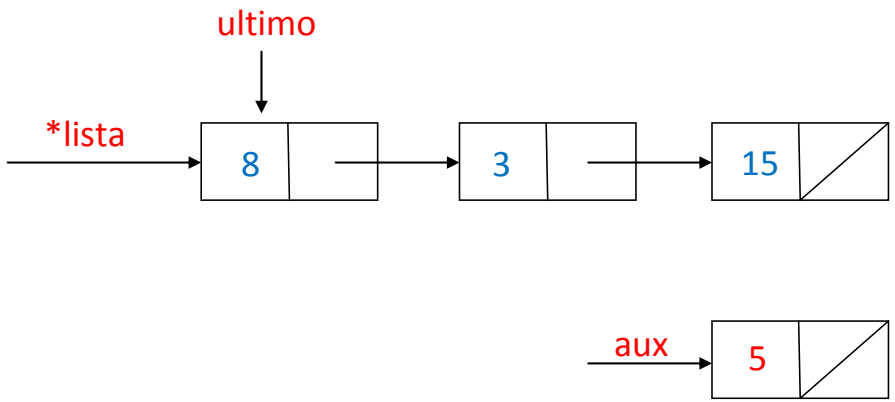
- Se la lista è vuota coincide con l'inserimento in testa
  $\Longrightarrow$ se facciamo una procedura è necessario il passaggio per indirizzo!
- Se la lista non è vuota, bisogna scandirla fino in fondo
  $\Longrightarrow$ dobbiamo usare un puntatore ausiliario per la scansione sia per la funzione che per la procedura

Inserimento di un elemento in coda

- Se la lista è vuota coincide con l'inserimento in testa
  $\Longrightarrow$ se facciamo una procedura 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 sia per la funzione che per la procedura
- La scansione deve terminare in corrispondenza dell'ultimo elemento al quale va collegato quello nuovo

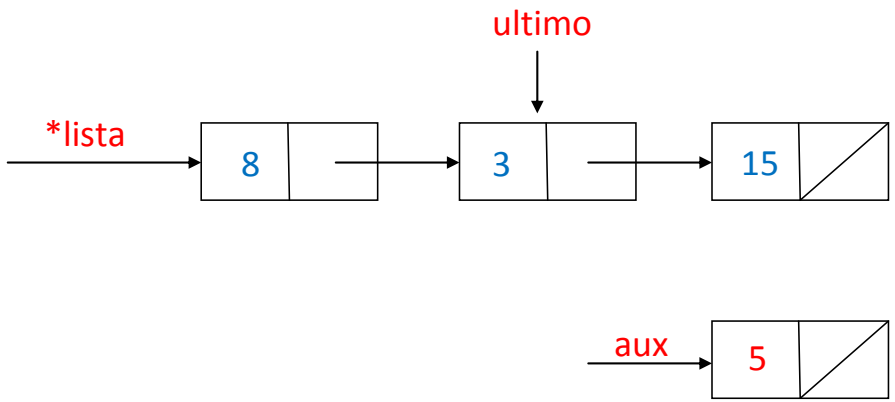
Inserimento di un elemento in coda

- Se la lista è vuota coincide con l'inserimento in testa
  $\Longrightarrow$ se facciamo una procedura 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 sia per la funzione che per la procedura
- La scansione deve terminare in corrispondenza dell'ultimo elemento al quale va collegato quello nuovo



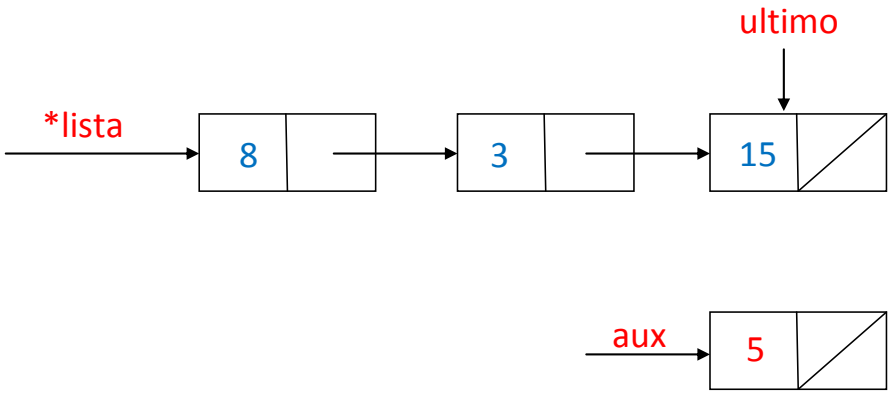
Inserimento di un elemento in coda

- Se la lista è vuota coincide con l'inserimento in testa
  $\Longrightarrow$ se facciamo una procedura 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 sia per la funzione che per la procedura
- La scansione deve terminare in corrispondenza dell'ultimo elemento al quale va collegato quello nuovo



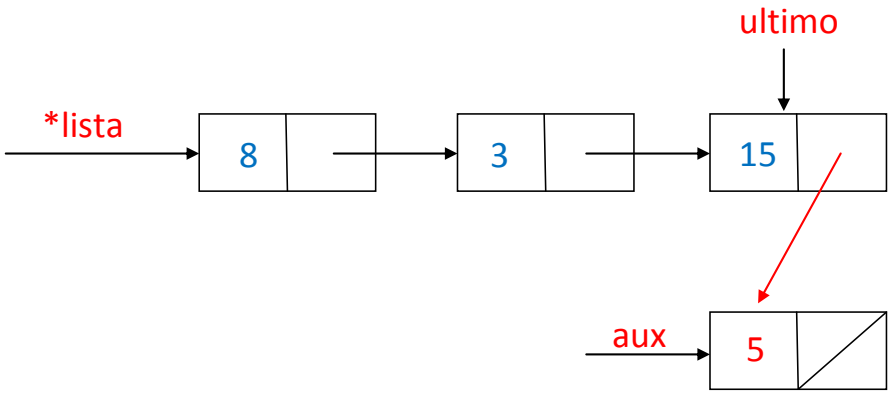
Inserimento di un elemento in coda

- Se la lista è vuota coincide con l'inserimento in testa
  $\Longrightarrow$ se facciamo una procedura 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 sia per la funzione che per la procedura
- 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
  $\Longrightarrow$ se facciamo una procedura 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 sia per la funzione che per la procedura
- La scansione deve terminare in corrispondenza dell'ultimo elemento al quale va collegato quello nuovo



Inserimento in coda

- realizzato con una procedura

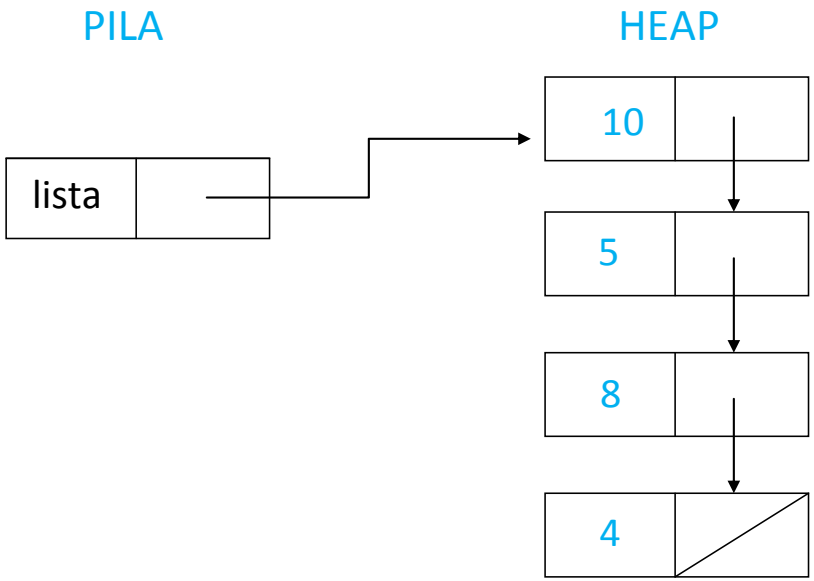
```
void InserzioneInCoda(ListaDiElementi *lista, int  elem)
{
  ListaDiElementi ultimo;   /* puntatore usato per la scansione */
  ListaDiElementi new_elem== malloc(sizeof(ElementoDiLista));
                            /* creazione del nuovo elemento */

  new_elem->info = elem;
  new_elem->next = NULL;

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

Inserimento in coda

- realizzato con una funzione

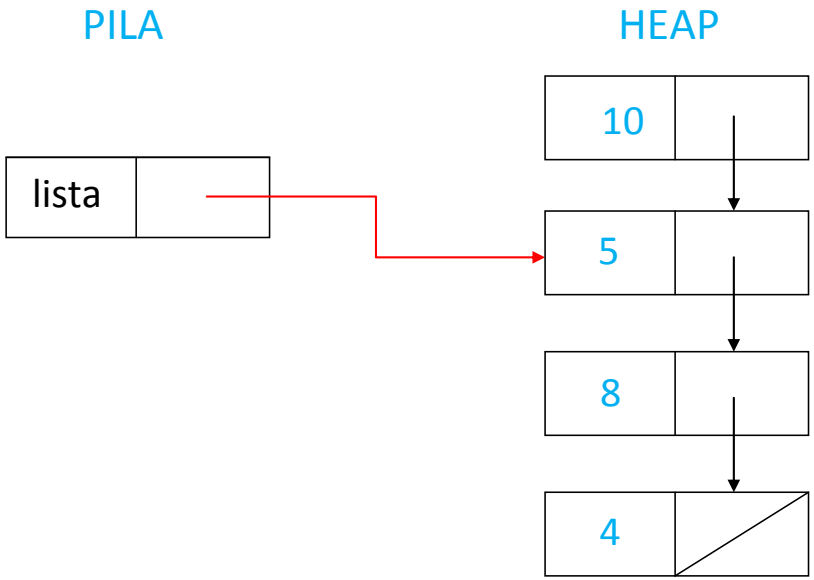
```
ListaDiElementi insertCodaFun(ListaDiElementi lista, int elem)
{
        ListaDiElementi ultimo=lista;
         ListaDiElementi new_elem=malloc(sizeof(ElementoDiLista));
         new_elem->info=elem;
         new_elem->next=NULL;
         if(lista==NULL)  //lista vuota
                 return new_elem;

         while(ultimo->next!=NULL)  //scorriamo la lista
                 ultimo=ultimo->next;

         ultimo->next=new_elem;
         return lista;}
```

Cancellazione del primo elemento



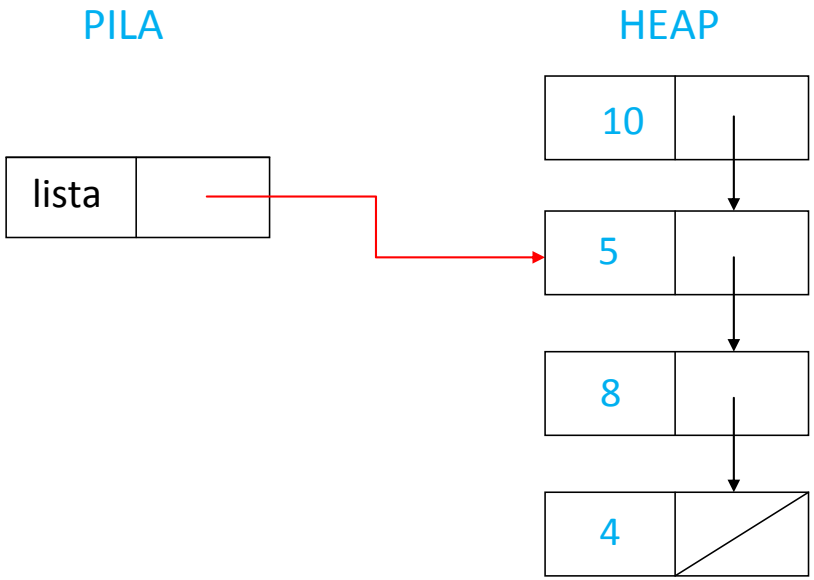
- se la lista è vuota non facciamo nulla

Cancellazione del primo elemento



- se la lista è vuota non facciamo nulla
- altrimenti eliminiamo il primo elemento

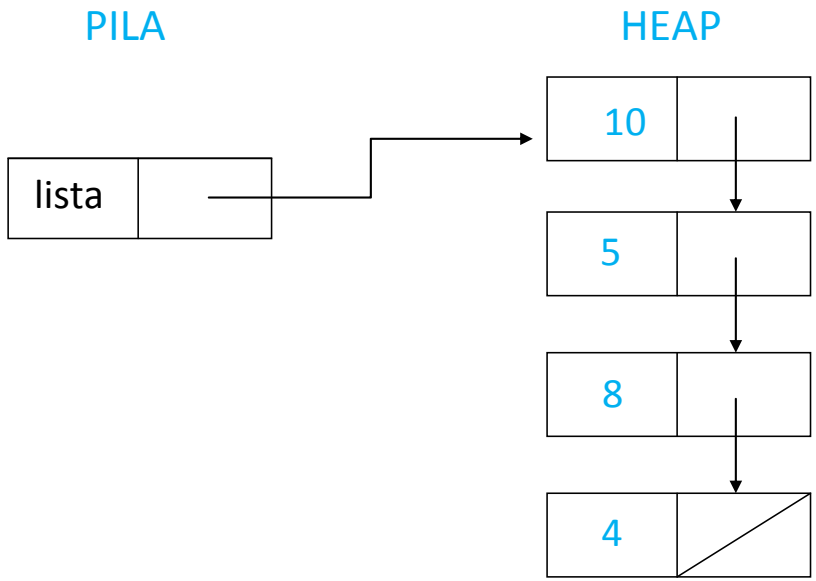
Cancellazione del primo elemento



- se la lista è vuota non facciamo nulla
- altrimenti eliminiamo il primo elemento

  $\Longrightarrow$ la lista deve essere passata per indirizzo

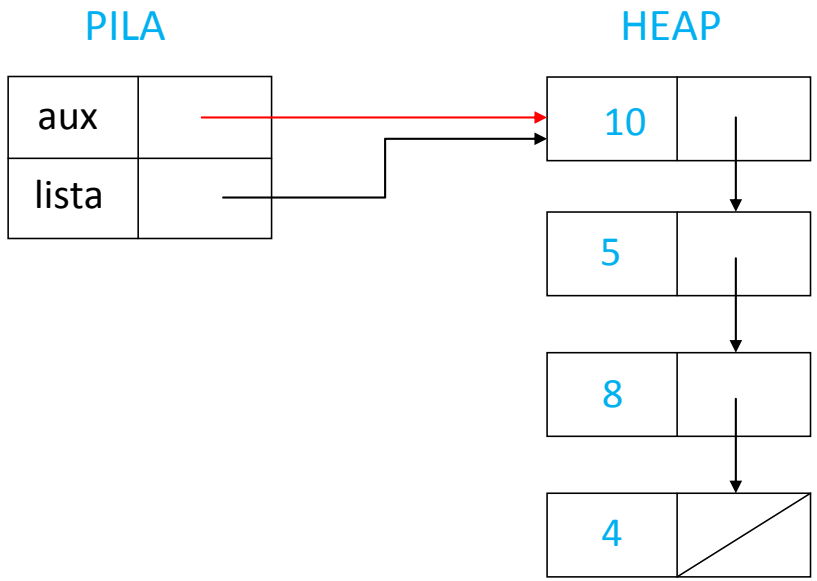
Cancellazione del primo elemento



- 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

Cancellazione del primo elemento

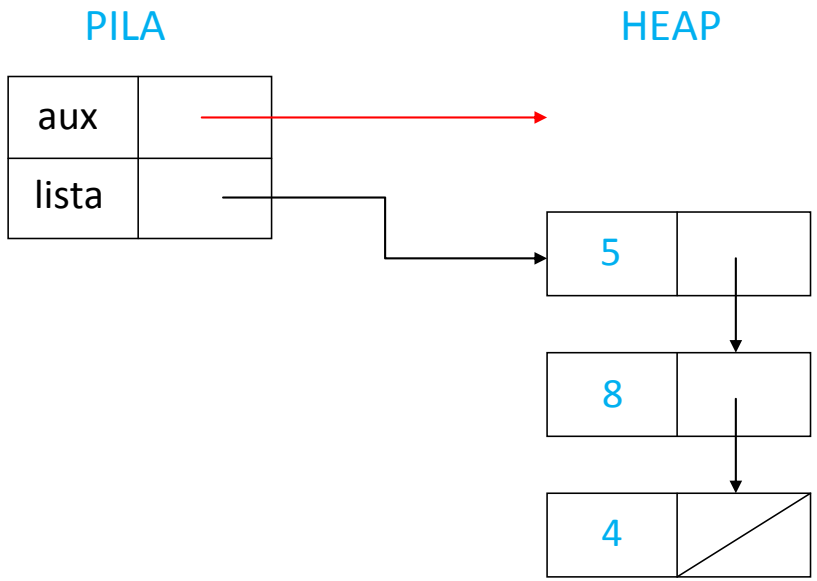


- se la lista è vuota non facciamo nulla
- altrimenti eliminiamo il primo elemento

  $\Longrightarrow$ 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

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);
     }
}
```

Appartenenza di un elemento ad una lista

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


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

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

- Non c'è bisogno di un puntatore ausiliario per scorrere la lista: il passaggio per valore consente di scorrere utilizzando il parametro formale!