Liste concatenate e allocazione dinamica

Laboratorio di Programmazione I

Corso di Laurea in Informatica
A.A. 2019/2020

Argomenti del Corso

Ogni lezione consta di una spiegazione assistita da slide, e seguita da esercizi in classe

- Introduzione all'ambiente Linux
- Introduzione al C
- Tipi primitivi e costrutti condizionali
- Costrutti iterativi ed array
- Funzioni, stack e visibilità variabili
- Puntatori e memoria
- Debugging
- Tipi di dati utente
- Liste concatenate e librerie
- Ricorsione

Sommario

Outline

Liste Concatenate

Fin'ora abbiamo visto come gli **array** ci permettano di rappresentare sequenze di elementi dello stesso tipo. Esempi: array di interi, array di caratteri, ma anche array di `struct`.

Liste Concatenate

Fin'ora abbiamo visto come gli **array** ci permettano di rappresentare sequenze di elementi dello stesso tipo. Esempi: array di interi, array di caratteri, ma anche array di `struct`.

Hanno dei vantaggi (accesso diretto, ossia tramite indice, e allocazione contigua in memoria) ma anche degli svantaggi:

Liste Concatenate

Fin'ora abbiamo visto come gli **array** ci permettano di rappresentare sequenze di elementi dello stesso tipo. Esempi: array di interi, array di caratteri, ma anche array di `struct`.

Hanno dei vantaggi (accesso diretto, ossia tramite indice, e allocazione contigua in memoria) ma anche degli svantaggi:

- hanno una dimensione fissata;

Liste Concatenate

Fin'ora abbiamo visto come gli **array** ci permettano di rappresentare sequenze di elementi dello stesso tipo. Esempi: array di interi, array di caratteri, ma anche array di `struct`.

Hanno dei vantaggi (accesso diretto, ossia tramite indice, e allocazione contigua in memoria) ma anche degli svantaggi:

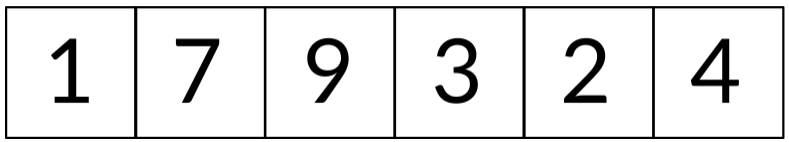
- hanno una dimensione fissata;
- inserire/eliminare un elemento all'interno di un array può essere un'operazione costosa:

Liste Concatenate

Fin'ora abbiamo visto come gli **array** ci permettano di rappresentare sequenze di elementi dello stesso tipo. Esempi: array di interi, array di caratteri, ma anche array di `struct`.

Hanno dei vantaggi (accesso diretto, ossia tramite indice, e allocazione contigua in memoria) ma anche degli svantaggi:

- hanno una dimensione fissata;
- inserire/eliminare un elemento all'interno di un array può essere un'operazione costosa:

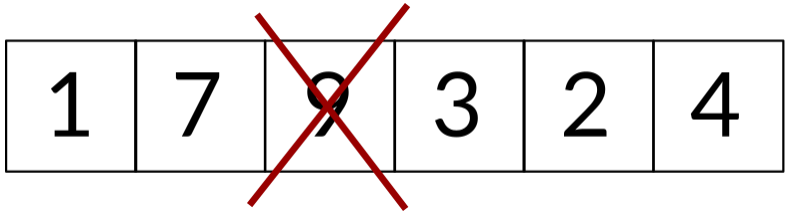


Liste Concatenate

Fin'ora abbiamo visto come gli **array** ci permettano di rappresentare sequenze di elementi dello stesso tipo. Esempi: array di interi, array di caratteri, ma anche array di `struct`.

Hanno dei vantaggi (accesso diretto, ossia tramite indice, e allocazione contigua in memoria) ma anche degli svantaggi:

- hanno una dimensione fissata;
- inserire/eliminare un elemento all'interno di un array può essere un'operazione costosa:

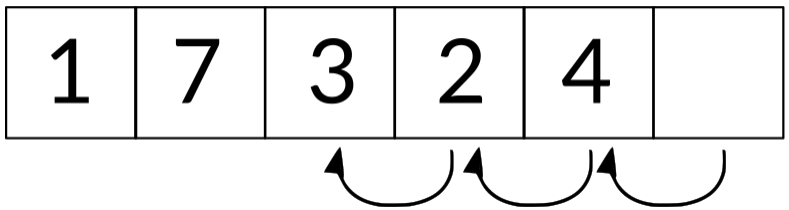


Liste Concatenate

Fin'ora abbiamo visto come gli **array** ci permettano di rappresentare sequenze di elementi dello stesso tipo. Esempi: array di interi, array di caratteri, ma anche array di `struct`.

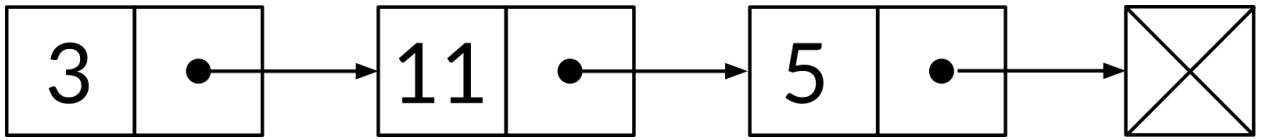
Hanno dei vantaggi (accesso diretto, ossia tramite indice, e allocazione contigua in memoria) ma anche degli svantaggi:

- hanno una dimensione fissata;
- inserire/eliminare un elemento all'interno di un array può essere un'operazione costosa:



Liste Concatenate

Una **lista concatenata** (*linked list*) è una sequenza di nodi, ciascuno dei quali memorizza un dato e contiene un riferimento (puntatore) al nodo successivo.

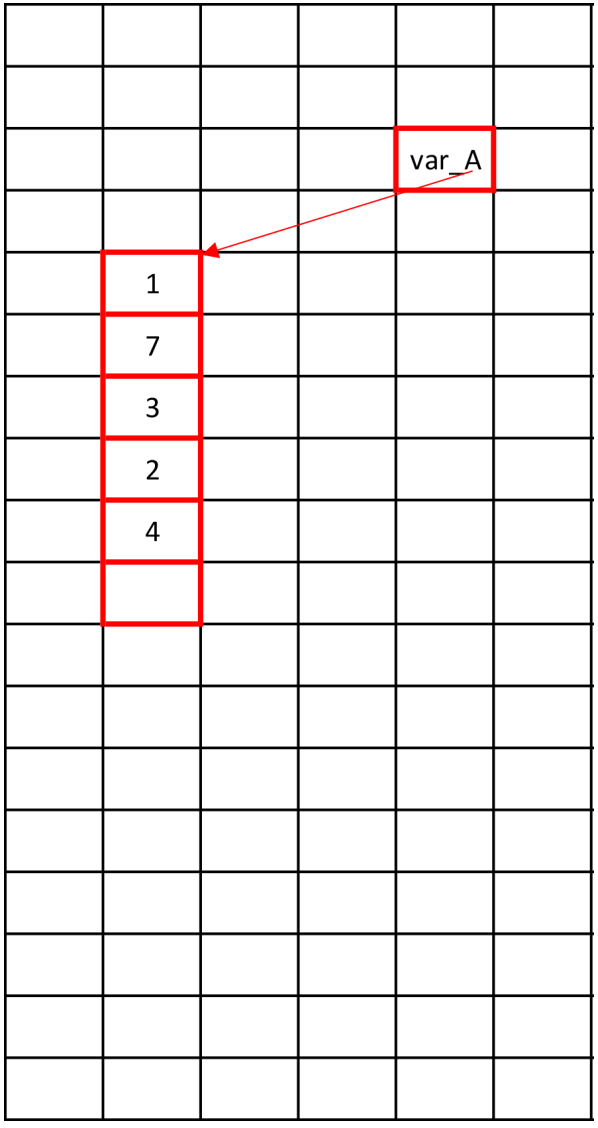


Sono strutture dinamiche:

- possiamo gestirle facilmente in maniera tale che crescano/decrescano in accordo alla quantità di dati che vogliamo memorizzare;
- in particolare per aggiungere o cancellare nodi in una qualunque posizione basta aggiustare i riferimenti all'interno della lista, senza toccare gli elementi non coinvolti.

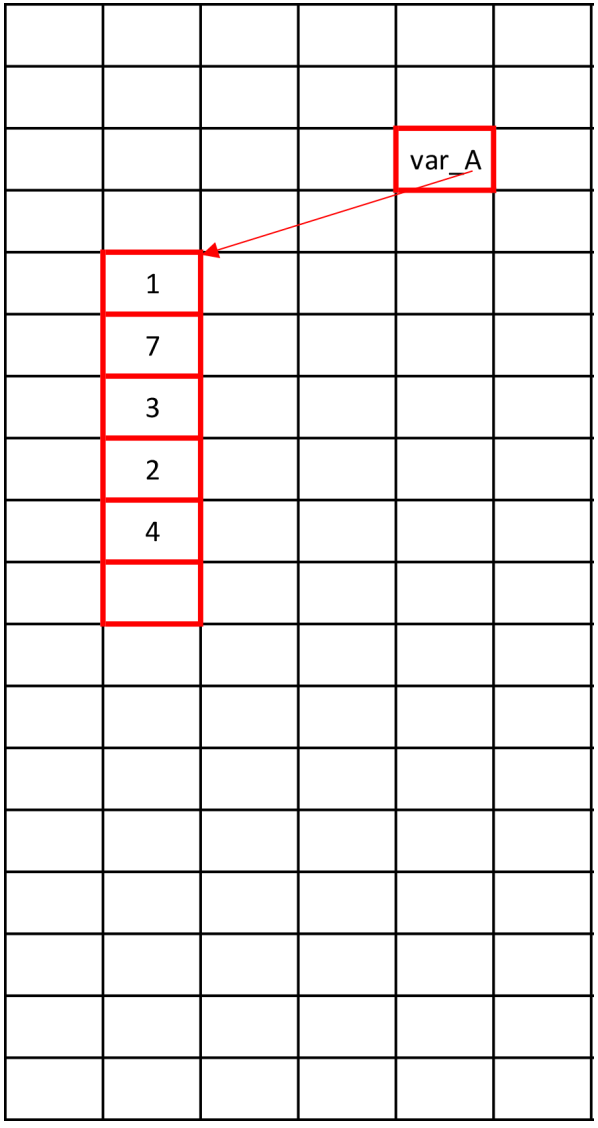
Modello di memoria - Liste concatenate

Array:

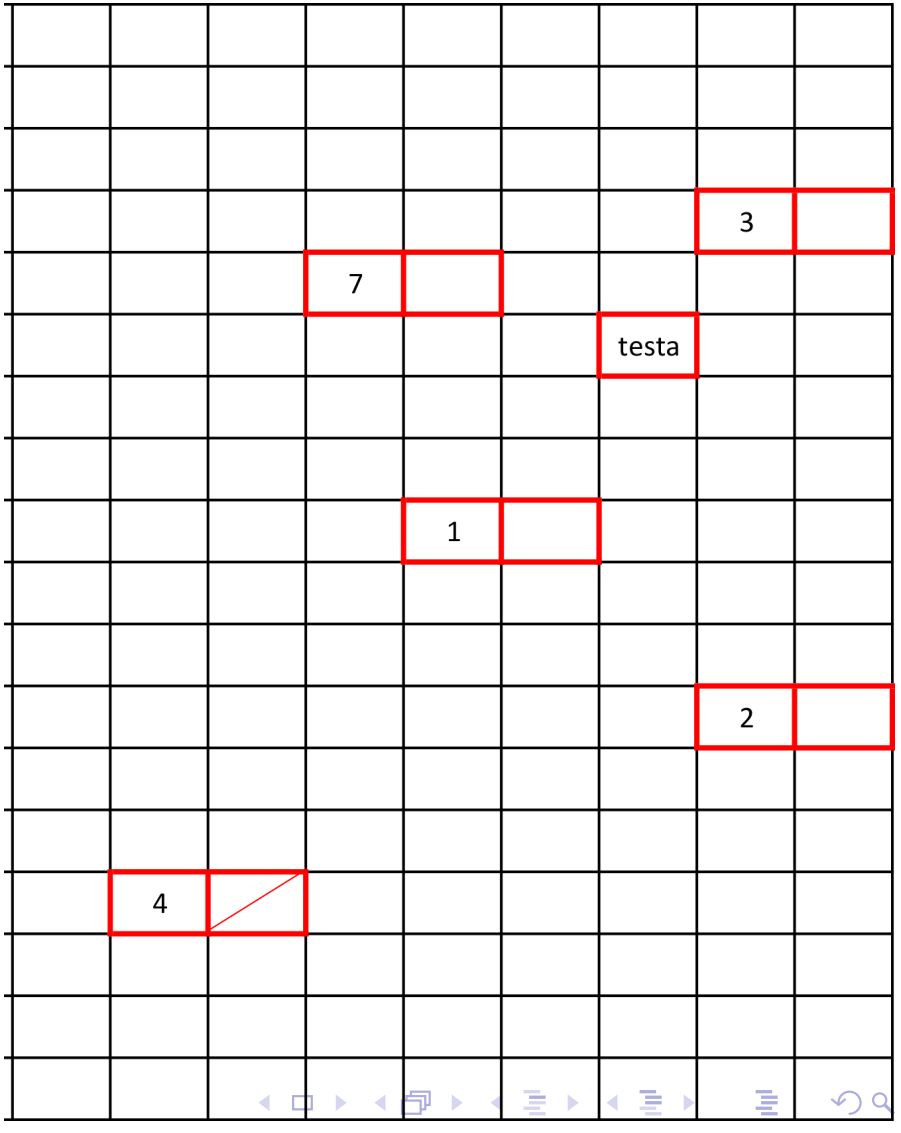


Modello di memoria - Liste concatenate

Array:

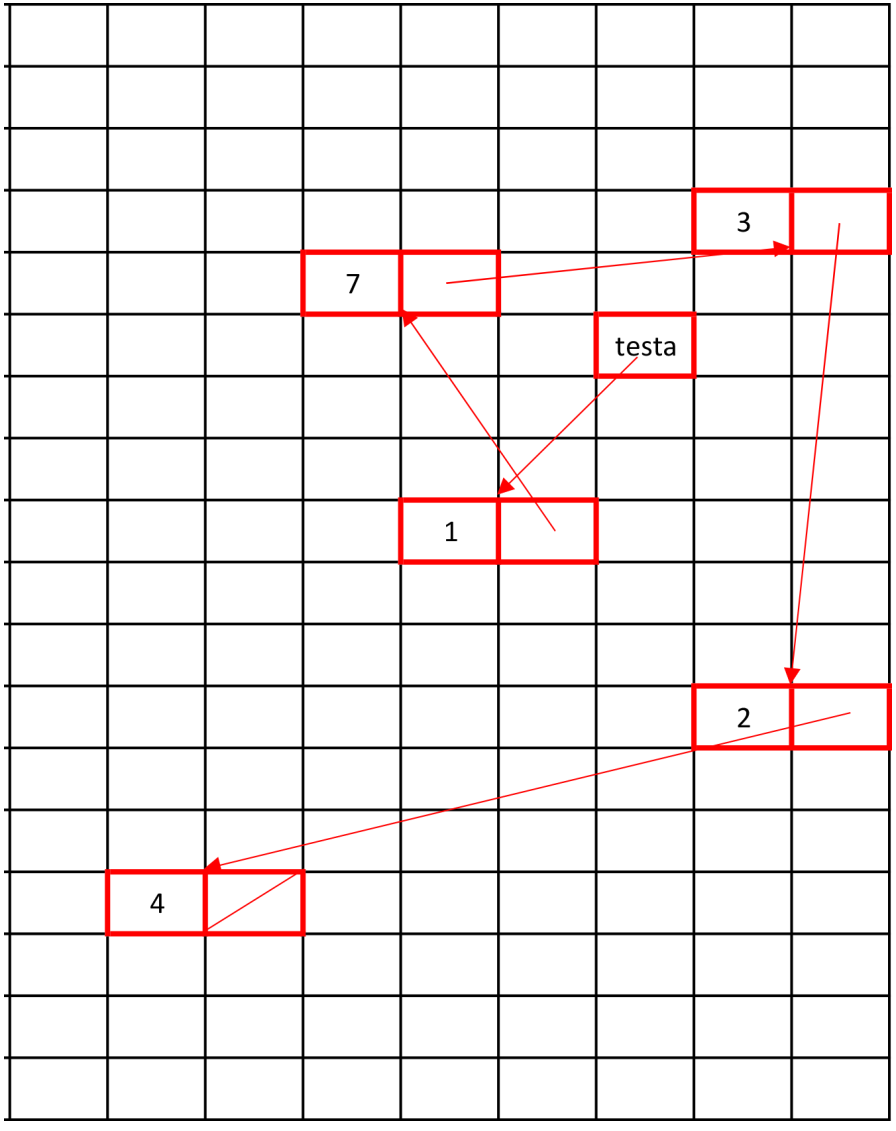


Strutture di interi:



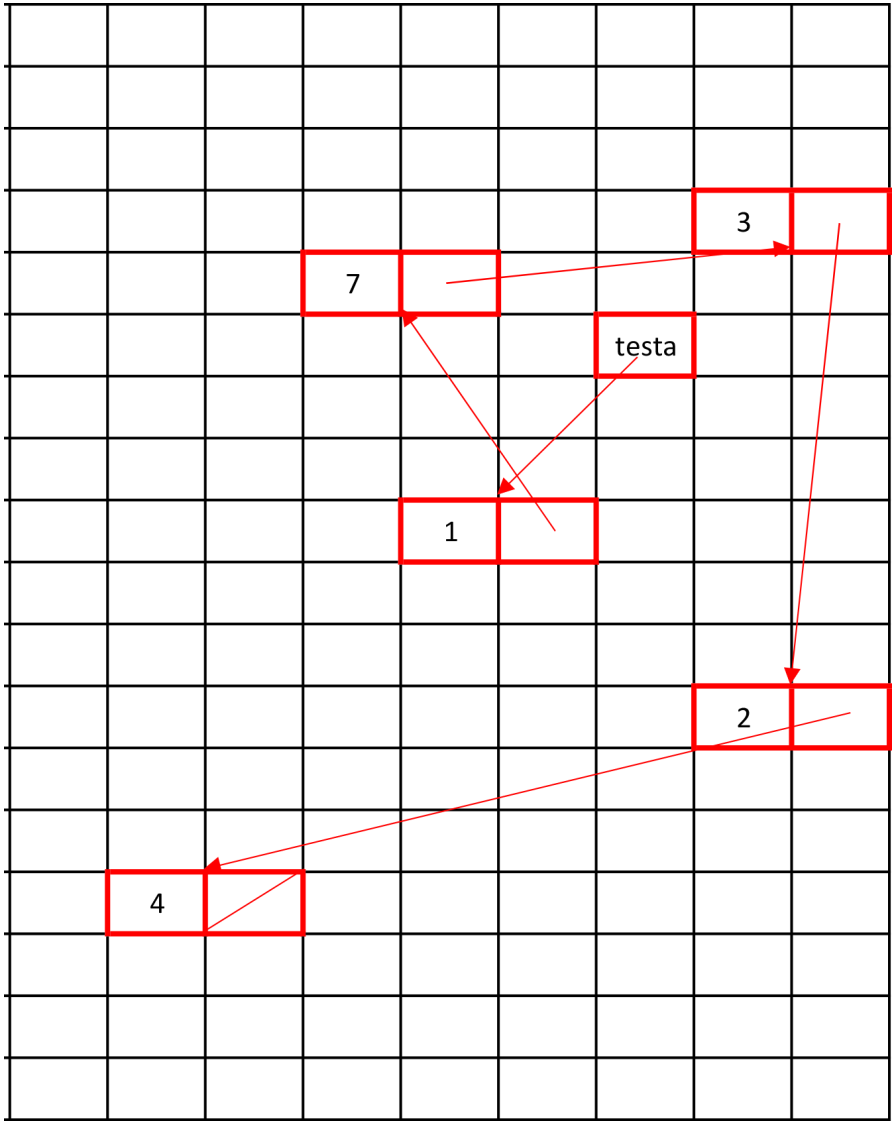
Modello di memoria

Lista concatenata:



Modello di memoria

Lista concatenata:

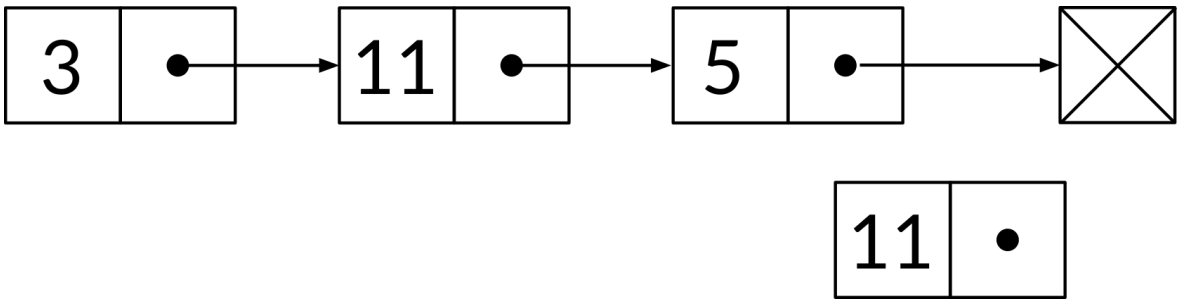


Rappresentazione semplificata della lista concatenata:

Da ora in poi useremo questa rappresentazione!

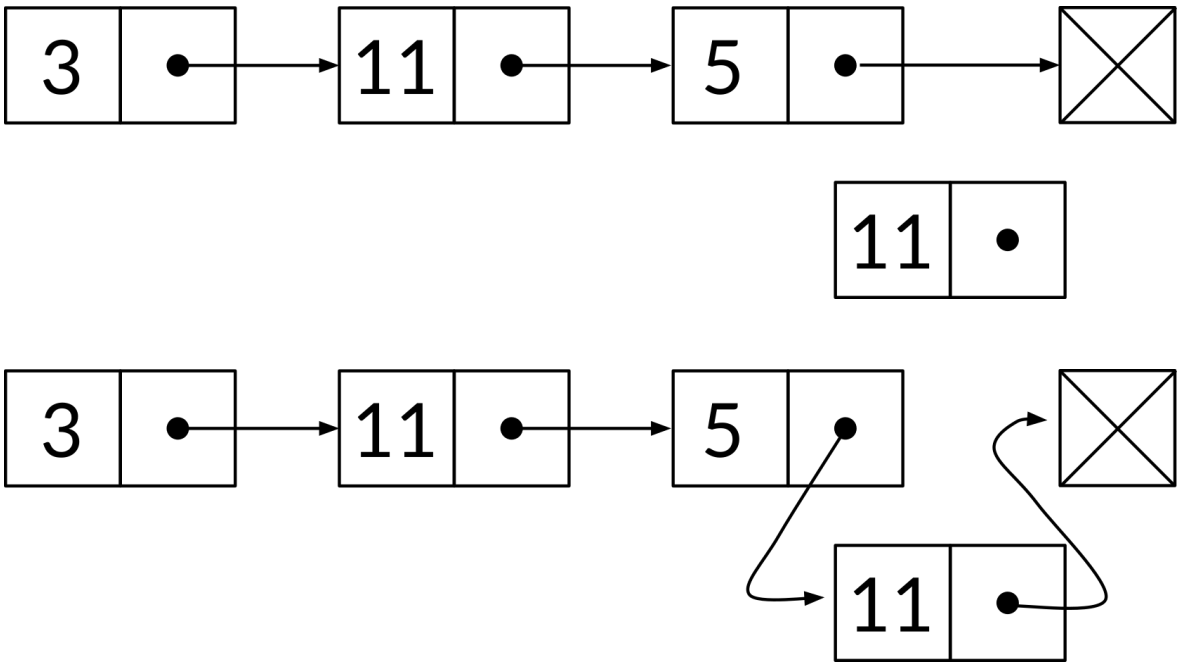
Inserzione/Cancellazione

Inserzione:



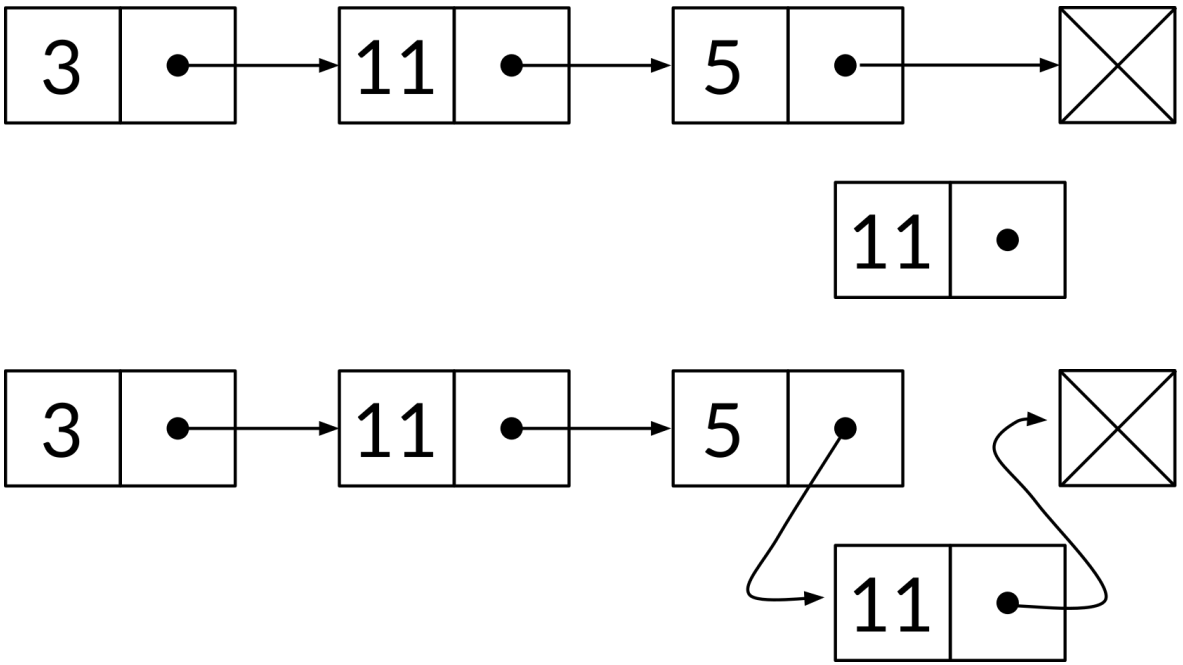
Inserzione/Cancellazione

Inserzione:

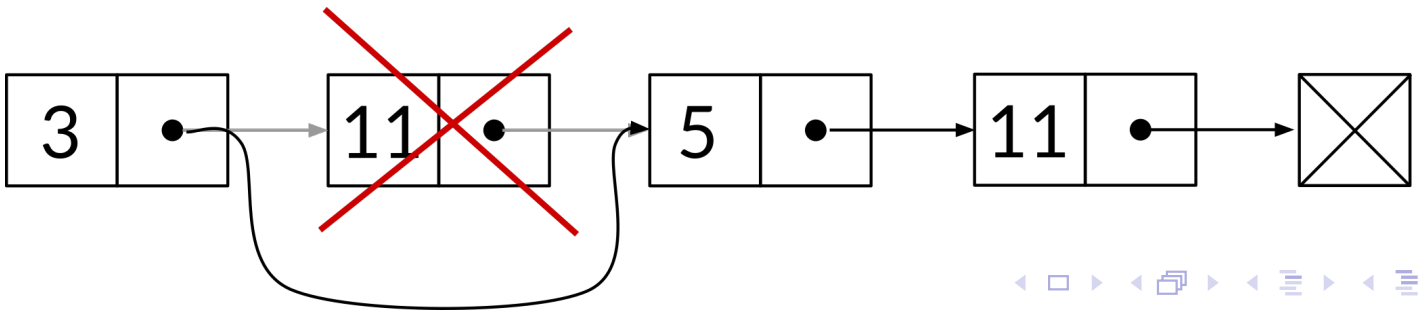


Inserzione/Cancellazione

Inserzione:



Rimozione:



Rappresentazione in C

In C le liste vengono realizzate mediante **struct**. Andiamo a definire il generico nodo:

```
struct elemento{
        //dato
        int info;
        //puntatore al prossimo elemento
        struct elemento* next;
};
```

Rappresentazione in C

In C le liste vengono realizzate mediante **struct**. Andiamo a definire il generico nodo:

```
struct elemento{
        //dato
        int info;
        //puntatore al prossimo elemento
        struct elemento* next;
};
typedef struct elemento ElementoDiLista;
```

Rappresentazione in C

In C le liste vengono realizzate mediante **struct**. Andiamo a definire il generico nodo:

```
struct elemento{
        //dato
        int info;
        //puntatore al prossimo elemento
        struct elemento* next;
};
typedef struct elemento ElementoDiLista;
```

- l'ultimo elemento della lista non ha successore e quindi il suo campo `next` sarà uguale a `NULL`;

Rappresentazione in C

In C le liste vengono realizzate mediante **struct**. Andiamo a definire il generico nodo:

```
struct elemento{
        //dato
        int info;
        //puntatore al prossimo elemento
        struct elemento* next;
};
typedef struct elemento ElementoDiLista;
```

- l'ultimo elemento della lista non ha successore e quindi il suo campo `next` sarà uguale a `NULL`;
- l'inizio della lista è individuato da una variabile puntatore chiamata `head` (tipo `ElementoDiLista*`) che punta al primo elemento della lista. L'accesso alla lista avviene attraverso il puntatore al primo elemento.

Sommario

Perchè usare l'allocazione dinamica

Ma con quello che sappiamo adesso per creare una lista di 3 elementi devo dichiarare 3 variabili di tipo `ElementoDiLista`:

```
ElementoDiLista el1, el2, el3;
```

Perchè usare l'allocazione dinamica

Ma con quello che sappiamo adesso per creare una lista di 3 elementi devo dichiarare 3 variabili di tipo `ElementoDiLista`:

```
ElementoDiLista el1, el2, el3;
```

- Il numero degli elementi della lista deve essere deciso a tempo di compilazione.

Perchè usare l'allocazione dinamica

Ma con quello che sappiamo adesso per creare una lista di 3 elementi devo dichiarare 3 variabili di tipo `ElementoDiLista`:

```
ElementoDiLista el1, el2, el3;
```

- 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?

Perchè usare l'allocazione dinamica

Ma con quello che sappiamo adesso per creare una lista di 3 elementi devo dichiarare 3 variabili di tipo `ElementoDiLista`:

```
ElementoDiLista el1, el2, el3;
```

- 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. . . .

Primitive

- Il C mette a disposizione delle primitive per la gestione dinamica della memoria
- Incluse in `stdlib.h`
- Primitive per allocare dinamicamente memoria
  - `malloc`, `calloc`, `realloc` (in questo corso vedremo solo la malloc)
- `free` - primitiva di deallocazione della memoria

Primitive - `malloc`

```
void * malloc(size_t size)
```

Primitive - `malloc`

```
void * malloc(size_t size)
```

- Alloca `size` bytes di memoria
  - `size_t` è semplicemente un tipo intero senza segno

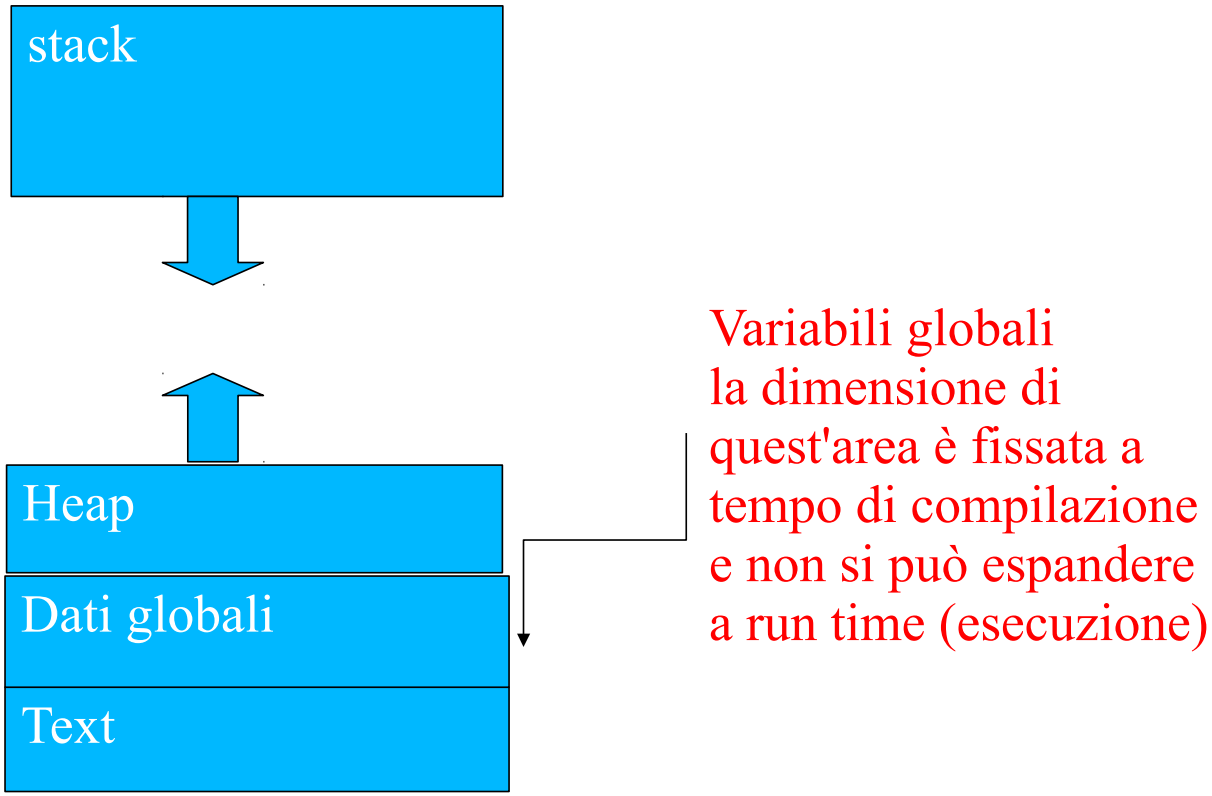
Primitive - `malloc`

```
void * malloc(size_t size)
```

- Alloca `size` bytes di memoria
  - `size_t` è semplicemente un tipo intero senza segno
- Restituisce un puntatore all'inizio dell'area di memoria allocata
  - `void*` definisce un puntatore ad un tipo generico che puó essere castato implicitamente od esplicitamente a qualsisi tipo puntatore
  - Se l'allocazione di memoria fallisce, `malloc` restituisce `NULL`

Heap

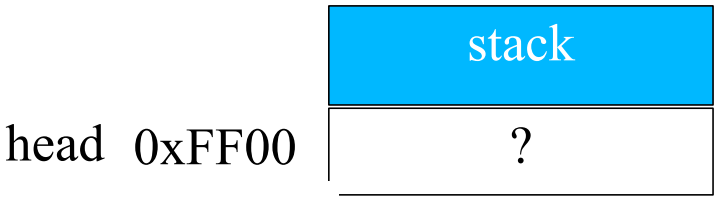
- Ricordiamo come vede la memoria un programma C in esecuzione



Allocazione dinamica: malloc

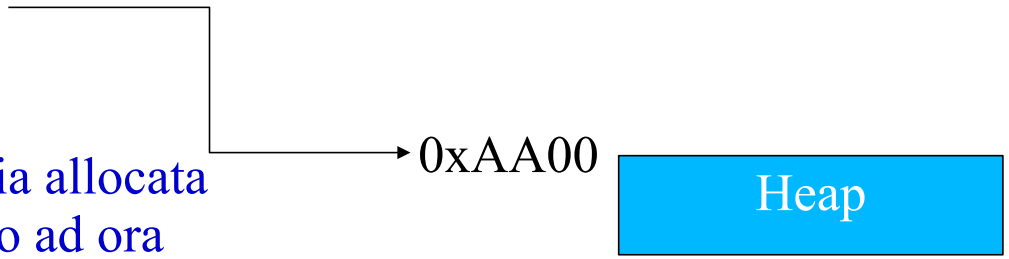
- Come si usa tipicamente:

```
ElementoDiLista* head;
```




```
head= malloc (sizeof (ElementoDiLista));
head->info=12;
head->next=NULL;
```

Primo indirizzo disponibile

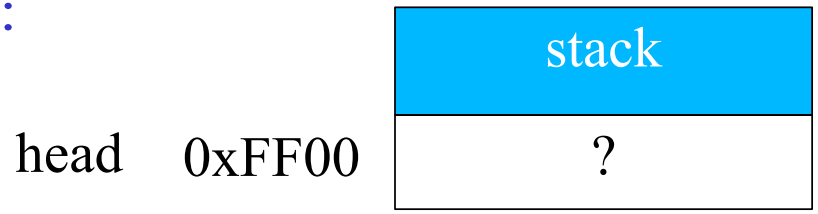


Contiene la memoria allocata
Dinamicamente fino ad ora

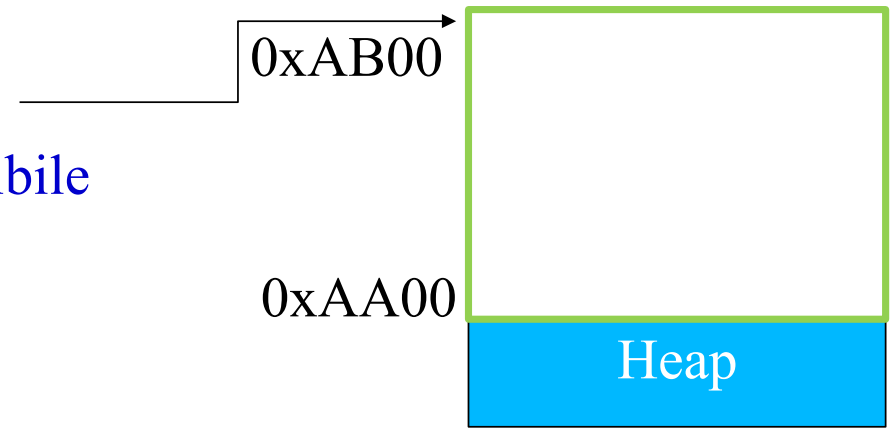
Allocazione dinamica: malloc

- Come si usa tipicamente:

```
ElementoDiLista* head;
```



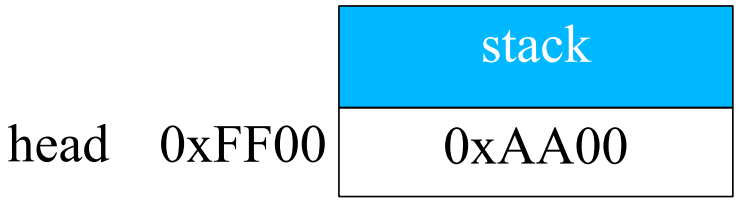

```
head=malloc (sizeof(ElementoDiLista));)
head->info=12;
head->next=NULL;
```



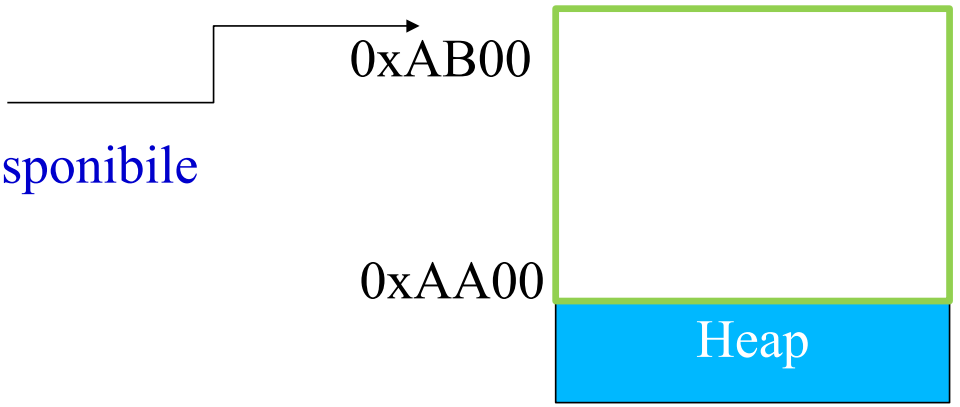
Allocazione dinamica: malloc

- Situazione di partenza:

```
ElementoDiLista* head;
```




```
head= malloc (sizeof(ElementoDiLista));
head->info=12;
head->next=NULL;
```

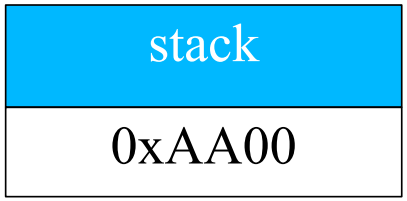


Primo indirizzo disponibile
Per l'allocazione

Allocazione dinamica: malloc

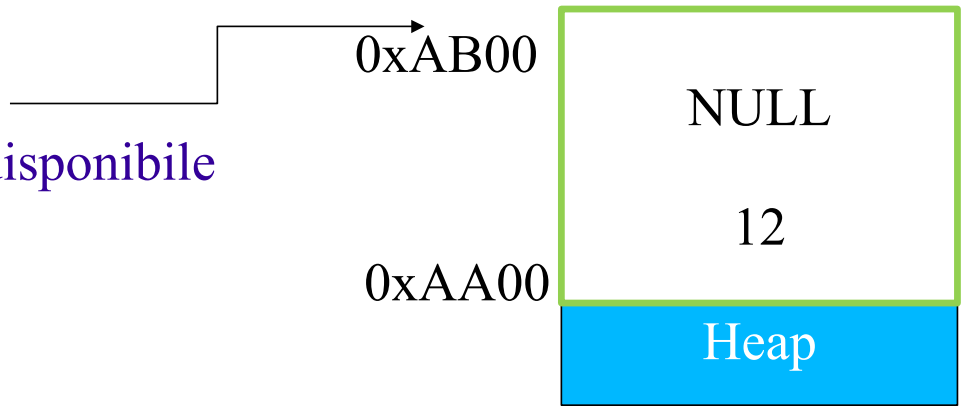
- Situazione di partenza:

```
ElementoDiLista* head;
```




```
head= malloc(sizeof(ElementoDiLista));
head->info=12;
head->next=NULL;
```

Primo indirizzo disponibile
Per l'allocazione



Primitive - `free`

In C, la gestione dello heap è lasciata al programmatore

- Lo spazio allocato sullo heap non viene deallocato all'uscita delle funzioni
- La deallocazione deve essere gestita esplicitamente

Primitive - `free`

In C, la gestione dello heap è lasciata al programmatore

- Lo spazio allocato sullo heap non viene deallocato all'uscita delle funzioni
- La deallocazione deve essere gestita esplicitamente

```
void free(void *ptr)
```

Primitive - `free`

In C, la gestione dello heap è lasciata al programmatore

- Lo spazio allocato sullo heap non viene deallocato all'uscita delle funzioni
- La deallocazione deve essere gestita esplicitamente

```
void free(void *ptr)
```

Dealloca lo spazio puntato da `ptr` ed allocato usando `malloc`

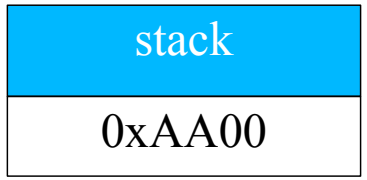
```
ElementoDiLista*  head;
head=(ElementoDiLista*) malloc(sizeof(ElementoDiLista));

    ...
free(head);
```

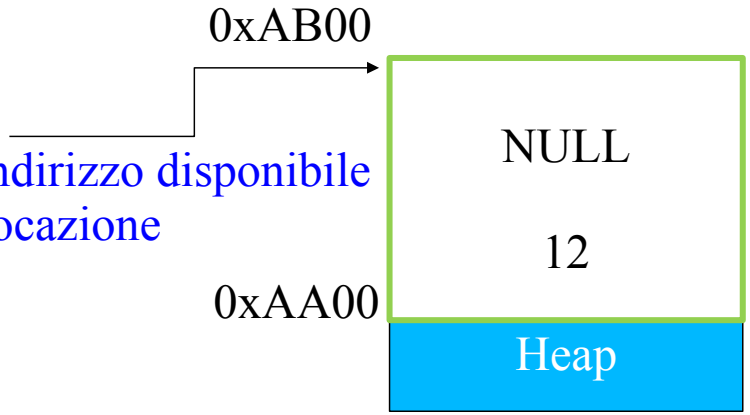
Allocazione dinamica: free

- Situazione di partenza:

```
ElementoDiLista* head;
```




```
head=malloc(sizeof(ElementoDiLista));
head->info=12;
head->next=NULL;
....
free(head);
```

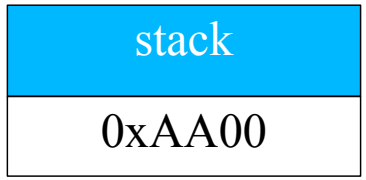


Primo indirizzo disponibile
Per l'allocazione

Allocazione dinamica: free

- Situazione di partenza:

```
ElementoDiLista* head;
```




```
head= malloc (sizeof(ElementoDiLista));
head->info=12;
head->next=NULL
....
free(head);
```

Primo indirizzo disponibile per l'allocazione

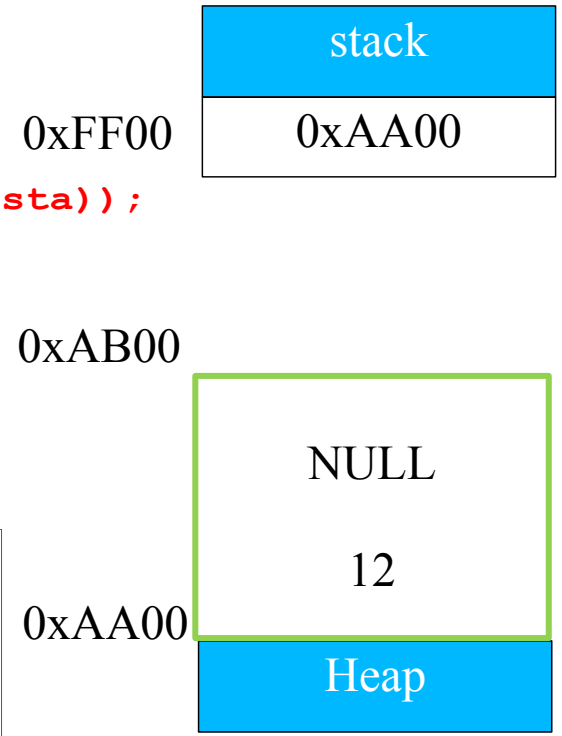
Allocazione dinamica: free

- Situazione di partenza:

```
ElementoDiLista* head;
```

```
head=malloc(sizeof(ElementoDiLista));
head->info=12;
head->next=NULL
....
free(head);
```

Notate che il puntatore **head non** viene modificato, contiene ancora l'indirizzo e può essere usato!
Il risultato però di tale accesso è indeterminato ....



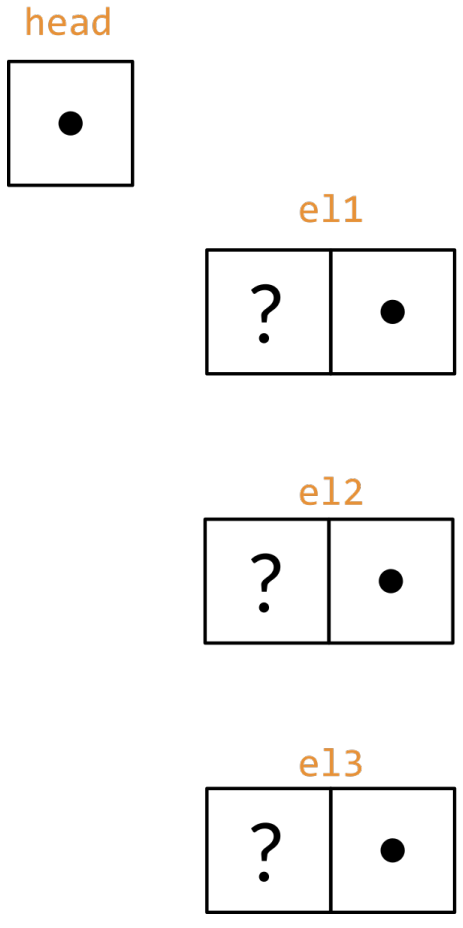
Liste concatenate - codice

Esempio: creazione di una lista con tre elementi $\{5, 1, 20\}$:

```
ElementoDiLista* head;
ElementoDiLista* el1,*el2,*el3;
el1=malloc(sizeof(ElementoDiLista));
el2=malloc(sizeof(ElementoDiLista));
el3=malloc(sizeof(ElementoDiLista));
```

Liste concatenate - codice

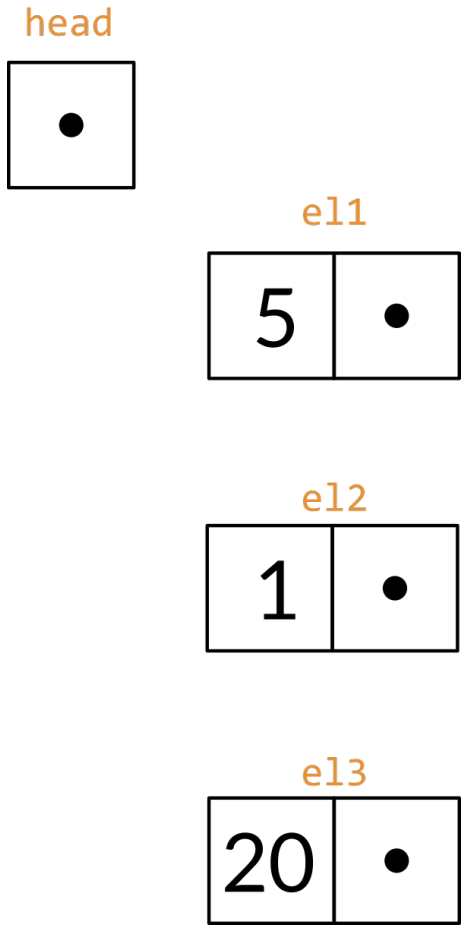
Esempio: creazione di una lista con tre elementi $\{5, 1, 20\}$:



Liste concatenate - codice

Esempio: creazione di una lista con tre elementi $\{5, 1, 20\}$:

```
el1->info = 5;
el2->info = 1;
el3->info = 20;
```

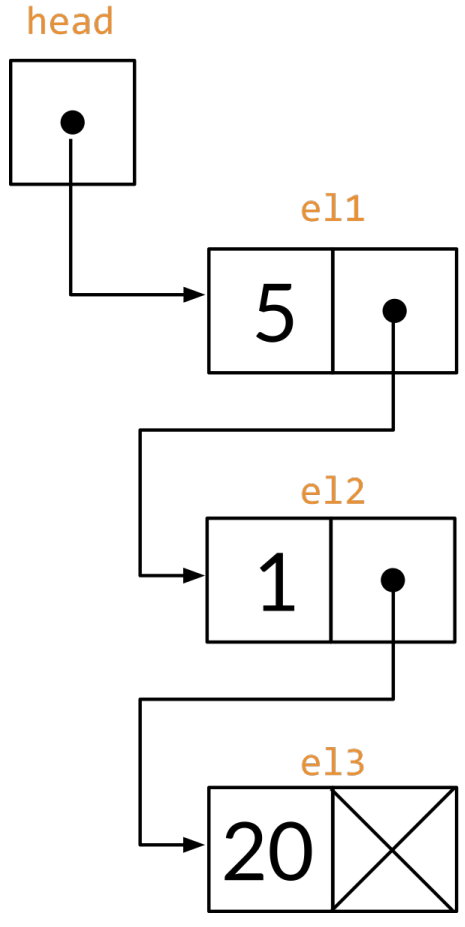


Liste concatenate - codice

Esempio: creazione di una lista con tre elementi $\{5, 1, 20\}$:

```
el1->info = 5;
el2->info = 1;
el3->info = 20;


head = el1;
el1->next = el2;
el2->next = el3;
el3->next = NULL;
```



Array vs. Liste

Anche le liste hanno i propri vantaggi e svantaggi:

- inserzione e rimozione molto piu' semplice che negli array (non c'e' bisogno di riorganizzare tutta la struttura);
- sono strutture dinamiche, possiamo crescere e decrescere facilmente durante l'esecuzione del programma.

Array vs. Liste

Anche le liste hanno i propri vantaggi e svantaggi:

+ inserzione e rimozione molto piu' semplice che negli array (non c'e' bisogno di riorganizzare tutta la struttura);
+ sono strutture dinamiche, possiamo crescere e decrescere facilmente durante l'esecuzione del programma.
- sono una sequenza di dati, quindi per arrivare ad un particolare nodo (es. l'ultimo) e' necessario percorrere tutta la lista;
- non sono memorizzate in maniera contigua in memoria (piu' tempo per passare da un elemento all'altro)

Array vs. Liste

Anche le liste hanno i propri vantaggi e svantaggi:

+ inserzione e rimozione molto piu' semplice che negli array (non c'e' bisogno di riorganizzare tutta la struttura);
+ sono strutture dinamiche, possiamo crescere e decrescere facilmente durante l'esecuzione del programma.
- sono una sequenza di dati, quindi per arrivare ad un particolare nodo (es. l'ultimo) e' necessario percorrere tutta la lista;
- non sono memorizzate in maniera contigua in memoria (piu' tempo per passare da un elemento all'altro)

Da questo punto di vista **array** e **liste** si complementano

Outline

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 del tipo `ElementoDiLista` visto in precedenza

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.

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(ElementoDiLista **lista)
{
        *lista=NULL;
}
```

Inizializzazione

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

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

int main()
{
     ElementoDiLista* Lista1;
     Inizializza(&Lista1);
     ...
}
```

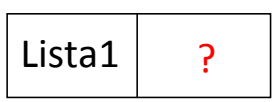
Inizializzazione

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

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

int main()
{
     ElementoDiLista* Lista1;
     Inizializza(&Lista1);
     ...
}
```

PILA

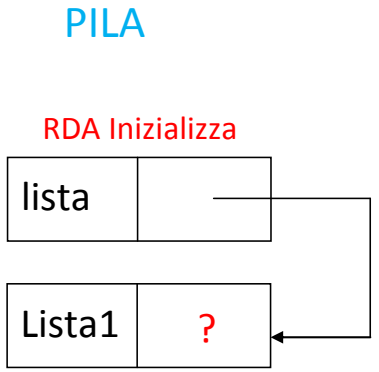


Inizializzazione

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

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

int main()
{
     ElementoDiLista* Lista1;
     Inizializza(&Lista1);
     ...
}
```

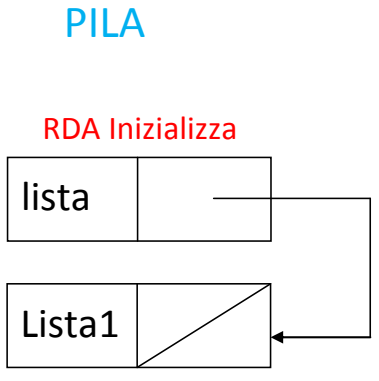


Inizializzazione

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

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

int main()
{
     ElementoDiLista* Lista1;
     Inizializza(&Lista1);
     ...
}
```



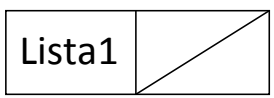
Inizializzazione

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

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

int main()
{
     ElementoDiLista* Lista1;
     Inizializza(&Lista1);
     ...
}
```

PILA



Inizializzazione

Cosa succederebbe se passassimo il parametro per valore?

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

int main()
{
     ElementoDiLista* Lista1;
     Inizializza(Lista1);
     ...
}
```

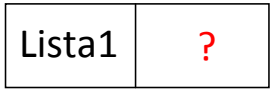
Inizializzazione

Cosa succederebbe se passassimo il parametro per valore?

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

int main()
{
     ElementoDiLista* Lista1;
     Inizializza(Lista1);
     ...
}
```

PILA



Inizializzazione

Cosa succederebbe se passassimo il parametro per valore?

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

int main()
{
     ElementoDiLista* Lista1;
     Inizializza(Lista1);
     ...
}
```

PILA

RDA Inizializza

lista	?

Lista1	?

Inizializzazione

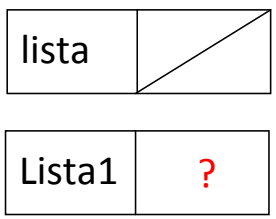
Cosa succederebbe se passassimo il parametro per valore?

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

int main()
{
     ElementoDiLista* Lista1;
     Inizializza(Lista1);
     ...
}
```

PILA

RDA Inizializza



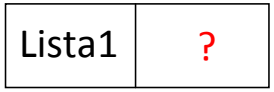
Inizializzazione

Cosa succederebbe se passassimo il parametro per valore?

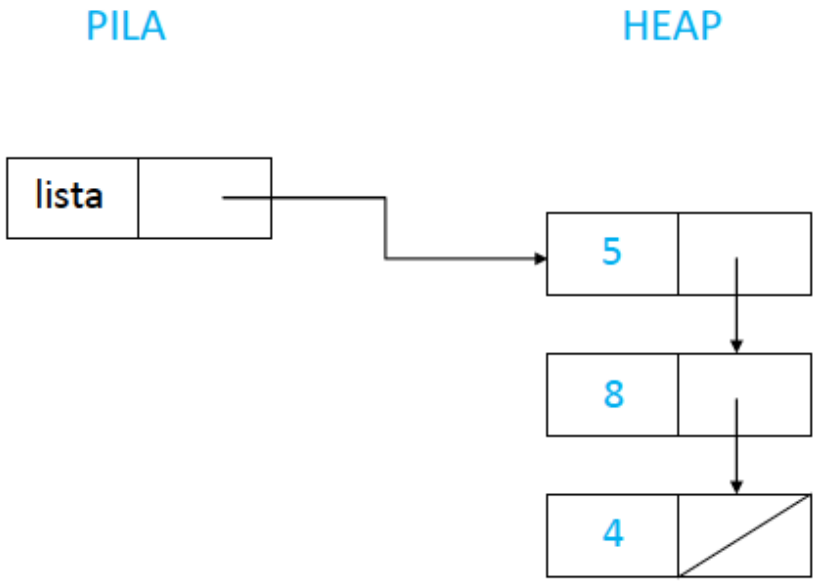
```
void Inizializza(ElementoDiLista* lista)
{
      lista=NULL;
}

int main()
{
     ElementoDiLista* Lista1;
     Inizializza(Lista1);
     ...
}
```

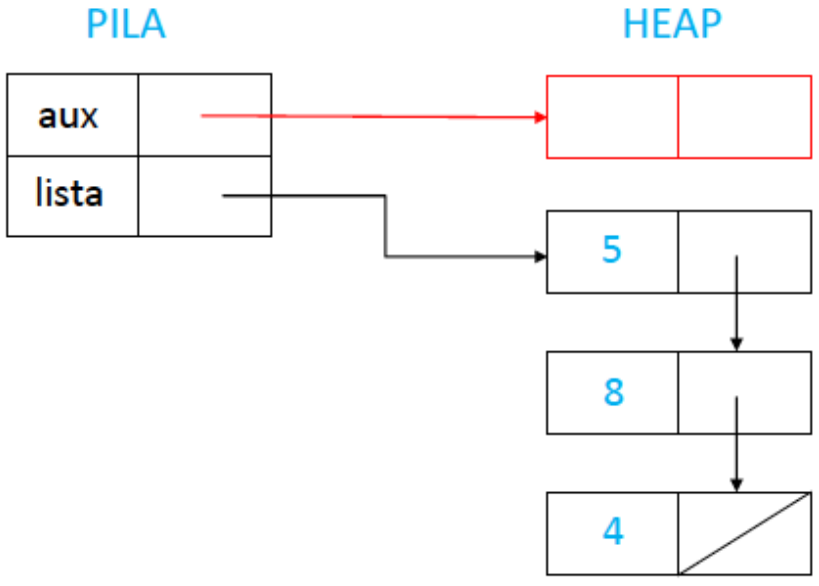
PILA



Inserimento in testa - passaggio per valore

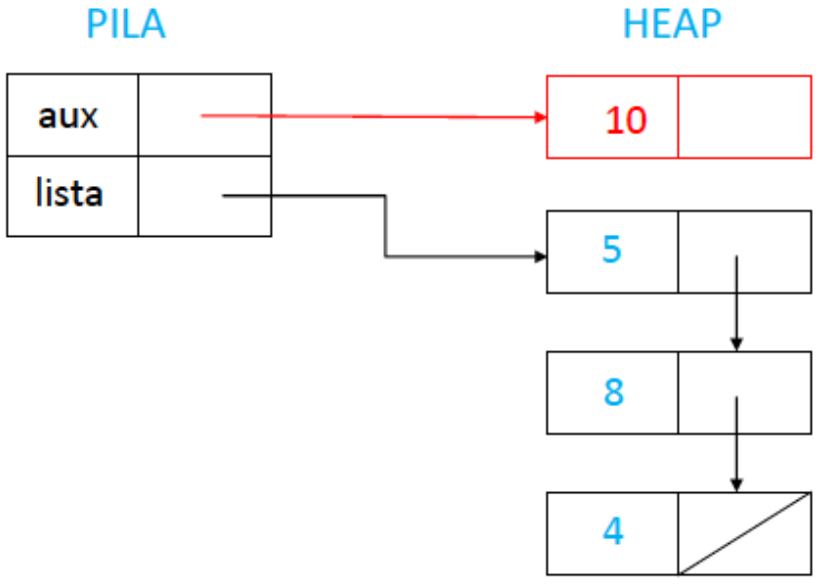


Inserimento in testa - passaggio per valore



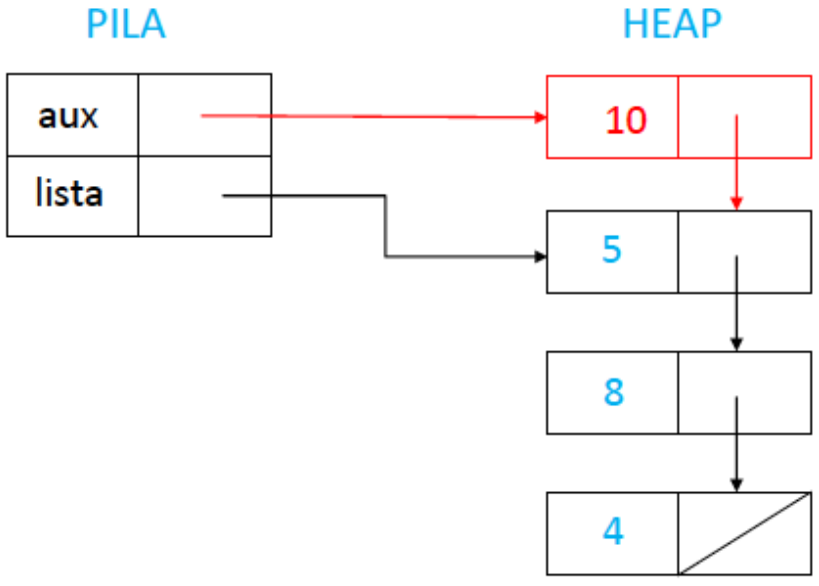
- allochiamo una nuova struttura per l'elemento (`malloc`)

Inserimento in testa - passaggio per valore



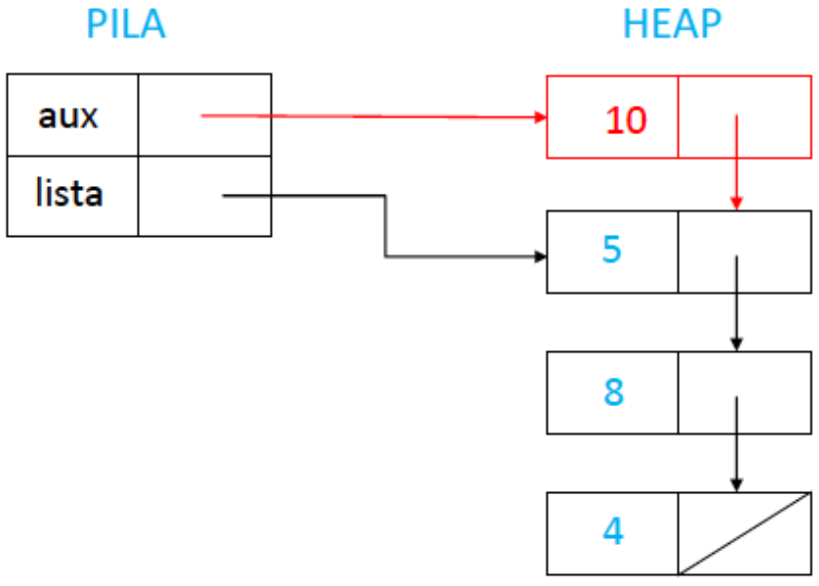
- allochiamo una nuova struttura per l'elemento (`malloc`)
- assegnamo il valore da inserire al campo `info` della struttura

Inserimento in testa - passaggio per valore



- allochiamo una nuova struttura per l'elemento (`malloc`)
- assegnamo il valore da inserire al campo `info` della struttura
- concateniamo la nuova struttura con la vecchia lista

Inserimento in testa - passaggio per valore

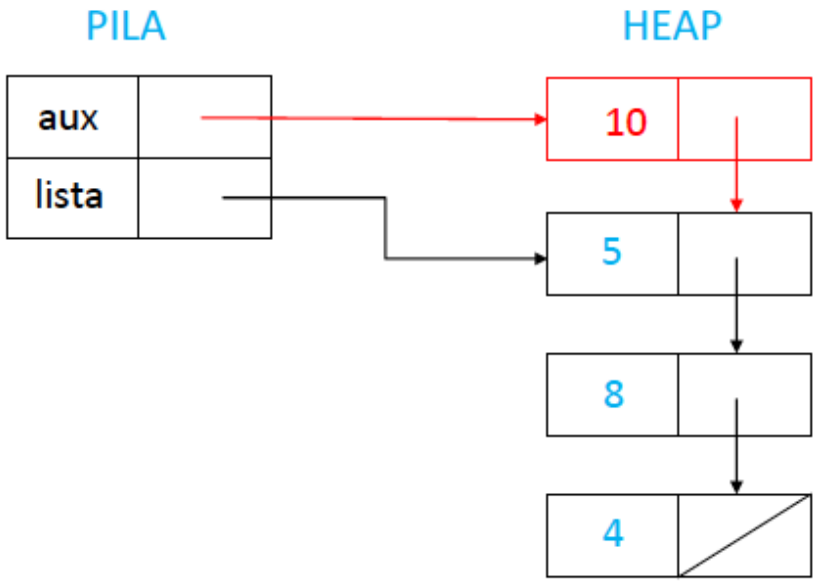


- allochiamo una nuova struttura per l'elemento (`malloc`)
- assegnamo il valore da inserire al campo `info` della struttura
- concateniamo la nuova struttura con la vecchia lista
- restituiamo la nuova testa della lista `aux`.

Inserimento in testa - passaggio per valore

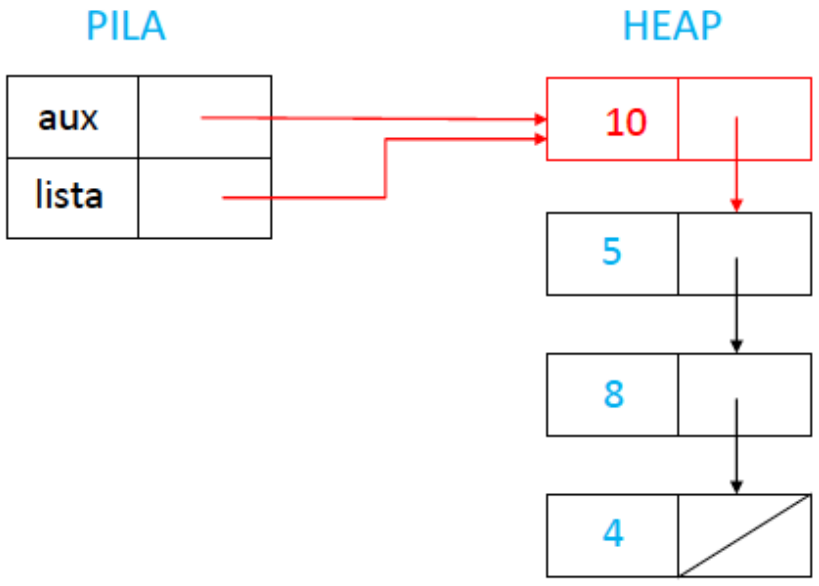
```
ElementoDiLista* inserisci_testa ( ElementoDiLista*
    lista , int elem) {
    ElementoDiLista* aux;
    aux = malloc(sizeof(ElementoDiLista));
    aux->info= elem;
    aux ->next = lista;
    return aux;
}
```

Inserimento in testa - passaggio per indirizzo



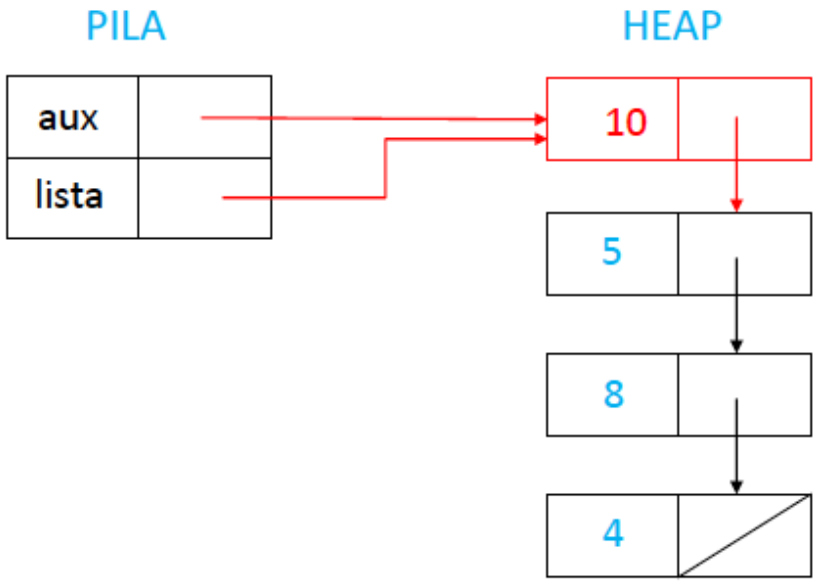
- in questo caso il puntatore alla nuova struttura non deve essere restituito

Inserimento in testa - passaggio per indirizzo



- in questo caso il puntatore alla nuova struttura non deve essere restituito
- il puntatore iniziale della lista viene fatto puntare alla nuova struttura

Inserimento in testa - passaggio per indirizzo



- in questo caso il puntatore alla nuova struttura non deve essere restituito
- il puntatore iniziale della lista viene fatto puntare alla nuova struttura
- la lista da modificare deve essere passata per indirizzo

Inserimento in testa - passaggio per indirizzo

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

Inserimento in coda

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

Inserimento in coda

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

-> è necessario il passaggio per indirizzo o restituire la nuova testa

Inserimento in coda

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

-> è necessario il passaggio per indirizzo o restituire la nuova testa

- Se la lista non è vuota, bisogna scandirla fino in fondo

Inserimento in coda

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

-> è necessario il passaggio per indirizzo o restituire la nuova testa

- Se la lista non è vuota, bisogna scandirla fino in fondo

-> dobbiamo usare un puntatore ausiliario per la scansione

Inserimento in coda

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

-> è necessario il passaggio per indirizzo o restituire la nuova testa

- 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 in coda

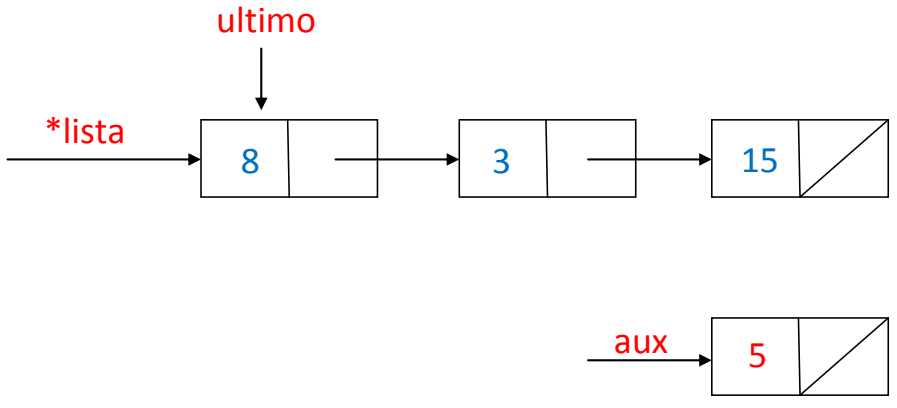
- Se la lista è vuota coincide con l'inserimento in testa

-> è necessario il passaggio per indirizzo o restituire la nuova testa

- 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 in coda

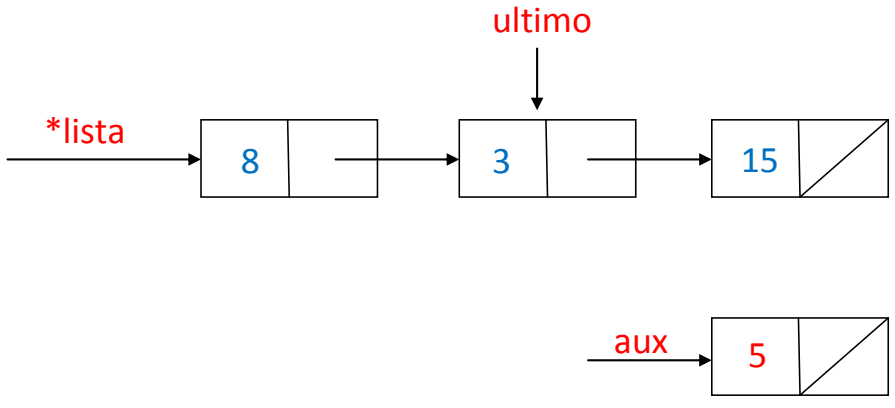
- Se la lista è vuota coincide con l'inserimento in testa

-> è necessario il passaggio per indirizzo o restituire la nuova testa

- 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 in coda

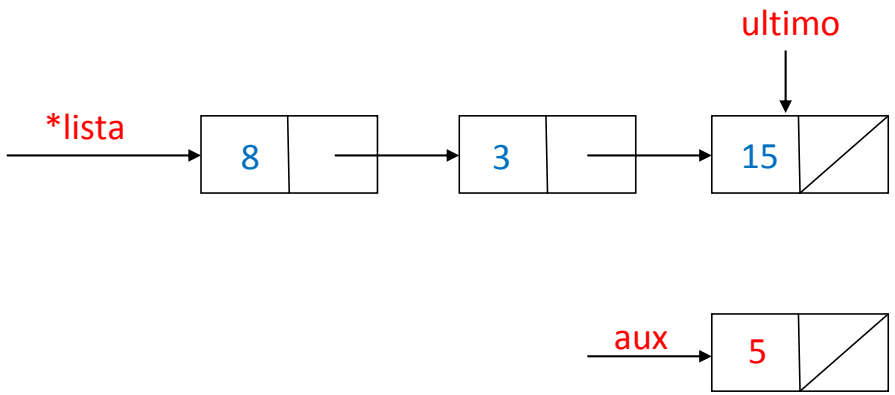
- Se la lista è vuota coincide con l'inserimento in testa

-> è necessario il passaggio per indirizzo o restituire la nuova testa

- 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 in coda

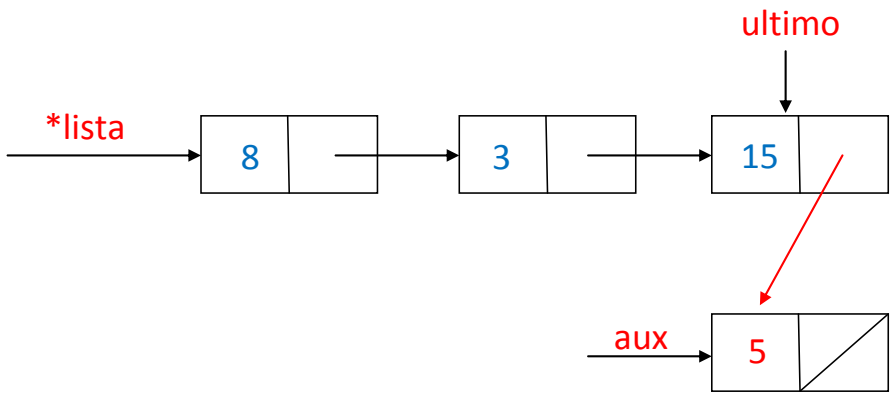
- Se la lista è vuota coincide con l'inserimento in testa

-> è necessario il passaggio per indirizzo o restituire la nuova testa

- 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



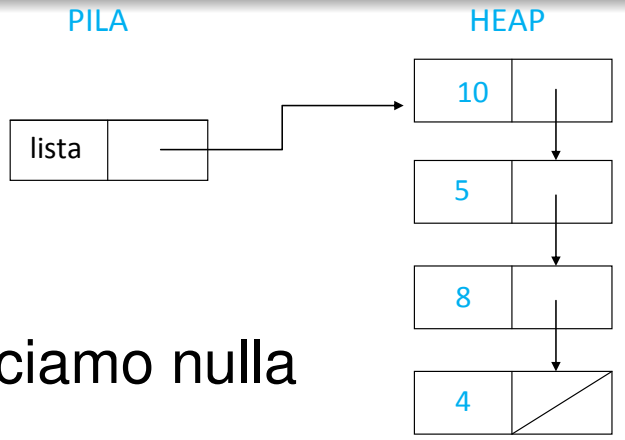
Inserimento in coda - passaggio per valore

```
ElementoDiLista* inserisci_coda(ElementoDiLista* lista,
    int elem)
{
    ElementoDiLista* ultimo=lista;
    ElementoDiLista* 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;
}
```

Inserimento in coda - passaggio per indirizzo

```
void inserisci_coda(ElementoDiLista **lista, int elem)
{
    ElementoDiLista* ultimo; /* puntatore usato per la
    scansione */
    ElementoDiLista* 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;
    }
}
```

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
  - -> la lista deve essere passata per indirizzo o la nuova lista deve essere restituita

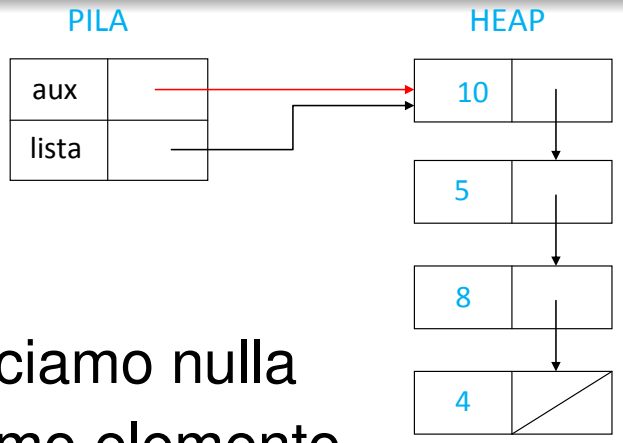
Cancellazione del primo elemento

- se la lista è vuota non facciamo nulla
- altrimenti eliminiamo il primo elemento
  - -> la lista deve essere passata per indirizzo o la nuova lista deve essere restituita
    - cancellare significa anche deallocare la memoria occupata dall'elemento

Cancellazione del primo elemento

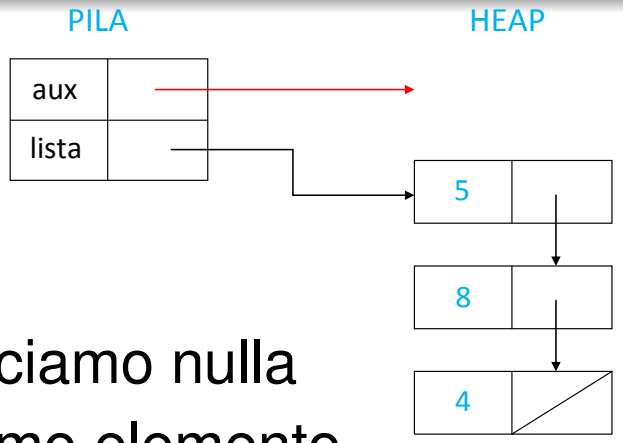
- se la lista è vuota non facciamo nulla
- altrimenti eliminiamo il primo elemento
  - -> la lista deve essere passata per indirizzo o la nuova lista deve essere restituita
    - cancellare significa anche deallocare la memoria occupata dall'elemento
  - -> dobbiamo invocare `free` passando il puntatore all'elemento da cancellare

Cancellazione del primo elemento



- se la lista è vuota non facciamo nulla
- altrimenti eliminiamo il primo elemento
  - -> la lista deve essere passata per indirizzo o la nuova lista deve essere restituita
    - 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



- se la lista è vuota non facciamo nulla
- altrimenti eliminiamo il primo elemento
  - -> la lista deve essere passata per indirizzo o la nuova lista deve essere restituita
    - 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 - passaggio per riferimento

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

Cancellazione del primo elemento - passaggio per valore

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

Appartenenza di un elemento alla lista

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

boolean appartiene(int elem, ElementoDiLista* lista)
{
  boolean trovato = false;

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

- Non c'è bisogno di un puntatore ausiliario per scorrere la lista

-> il passaggio per valore consente di scorrere utilizzando il parametro formale!