Programmazione I

Il Linguaggio C

Strutture Dati - Liste

Daniel Riccio

Università di Napoli, Federico II

01 dicembre 2021

Sommario

- Argomenti
  - Tipi ricorsivi
  - Le liste
  - Liste semplicemente concatenate
  - Operazioni su liste semplicemente concatenate

Tipi ricorsivi

Definire tipi ricorsivi:

```
typedef struct {
  char nome[20];
  char cognome[20];
  int peso;
  Persona padre;
} Persona;
```

concettualmente sbagliato. ricorsione infinita

Tipi ricorsivi

Definire tipi ricorsivi:

```
typedef struct {
  char nome[20];
  char cognome[20];
  int peso;
  Persona *padre;
} Persona;
```

Concettualmente giusto.

Ma non compila perché al momento della dichiarazione del campo **padre** il tipo **Persona** non esiste ancora.

Tipi ricorsivi

Definire tipi ricorsivi:

```
typedef struct P {
  char nome[20];
  char cognome[20];
  int peso;
  struct P *padre;
} Persona;

Persona a,b;
a.padre = &b;
```

Tipi ricorsivi

Definire tipi ricorsivi:

```
typedef struct Persona {
  char nome[20];
  char cognome[20];
  int peso;
  struct Persona *padre;
};

struct Persona a,b;
a.padre = &b;
```

Le liste

Una **lista** è una **struttura dati ricorsiva**

(formata da elementi dello stesso tipo e collegati insieme) la cui lunghezza può variare dinamicamente.

I suoi elementi sono variabili dinamiche e vengono creati e/o distrutti a tempo di esecuzione producendo una struttura dati che cresce o diminuisce a seconda delle esigenze del programma in esecuzione.

È possibile implementare liste anche tramite array, ma ciò può avvenire solo quando si conoscono esattamente le dimensioni della lista.

Le liste

Ogni elemento di una lista è definito come una struttura costituita da uno o più campi dati e da un campo puntatore contenente l'indirizzo dell'elemento successivo.

```
struct elem
{
    int info;
    struct elem * succ;
};
```

Strutture concatenate

Una struttura è detta **concatenata** quando è costituita, oltre che dai suoi normali **membri**, anche da uno o più membri aggiuntivi, dichiarati come **puntatori** alla **struttura** stessa.

```
struct rec
{
    int info;
    struct rec * next;
};
```

La definizione di una **struttura concatenata** è di solito accompagnata da un certo numero di **funzioni**, che hanno il compito di **gestirla**, cioè eseguire le operazioni di **inserimento**, di **eliminazione** e di **ricerca** di **oggetti**.

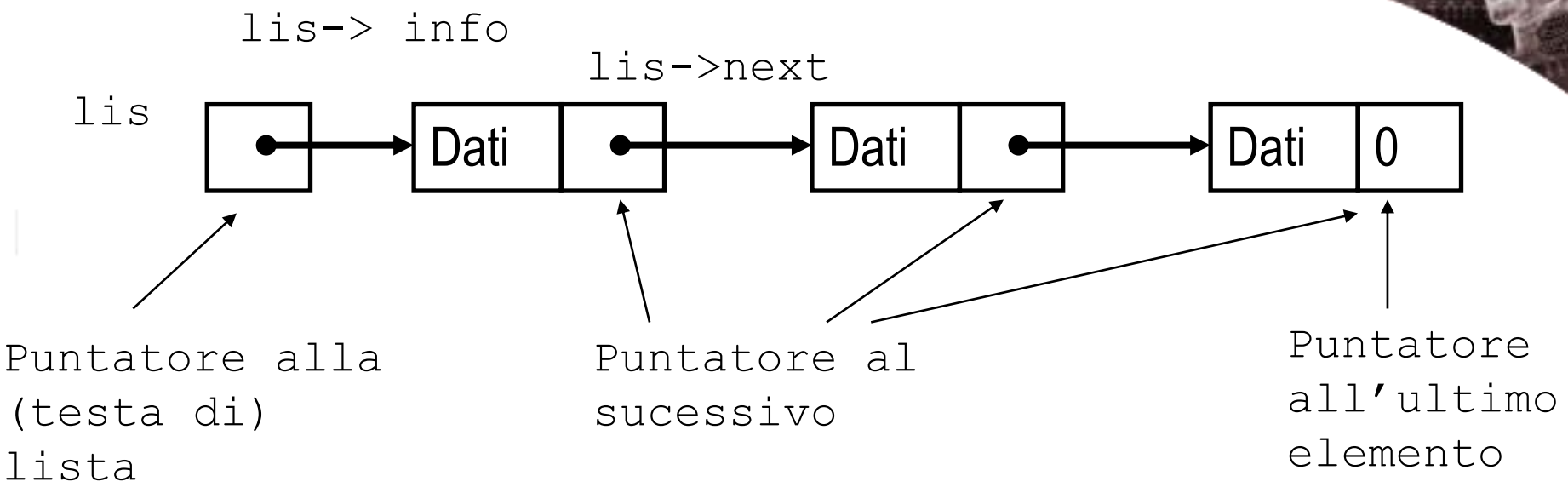
Liste

Ogni lista è definita da una variabile puntatore che punta al primo elemento della lista.

Nel caso di assenza di elementi (lista vuota) tale variabile puntatore assume valore **`NULL`**.

In una lista il campo puntatore dell'ultimo elemento assume sempre valore **`NULL`**.

Liste




```
struct rec
{
    int info;
    struct rec* next;
};
```

Allocazione dinamica di liste

L’allocazione dinamica della memoria si presta alla gestione di liste di oggetti, quando il loro numero non è definito a priori.

Queste liste possono aumentare e diminuire di dimensioni dinamicamente in base al flusso del programma, e quindi devono essere gestite in un modo più efficiente dell’allocazione di memoria permanente sotto forma di array

Allocazione dinamica di liste

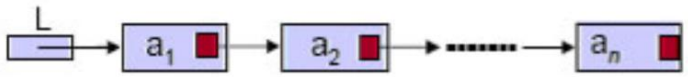
Una **lista concatenata** (linked list) è un insieme di **oggetti**, caratterizzati dal fatto di essere **istanze** di una **struttura concatenata**.

In ogni oggetto, i membri puntatori alla struttura contengono l'indirizzo di altri oggetti della lista, creando così un **legame** fra gli oggetti e rendendo la stessa lista **percorribile**, anche se gli oggetti non sono allocati consecutivamente in memoria.

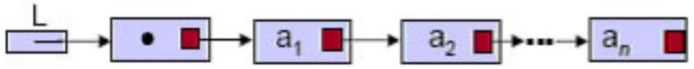
Se la struttura possiede un solo membro puntatore a se stessa, la lista è detta **single-linked** (o **monodirezionale**), se ne possiede due, è detta **double-linked** (o **bidirezionale**).

Tipi di liste

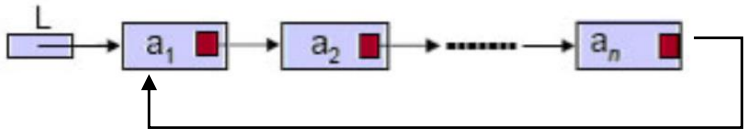
Lista monodirezionale (può essere visitata in un solo senso -> presenza di un puntatore per nodo)



Lista monodirezionale con sentinella (ha una cella in più detta **sentinella**, che è direttamente indirizzata da L, ma non contiene i valore di alcun elemento.

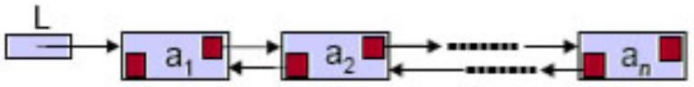


Lista monodirezionale circolare (l'ultimo puntatore non ha valore **NULL**, ma punta al primo nodo)

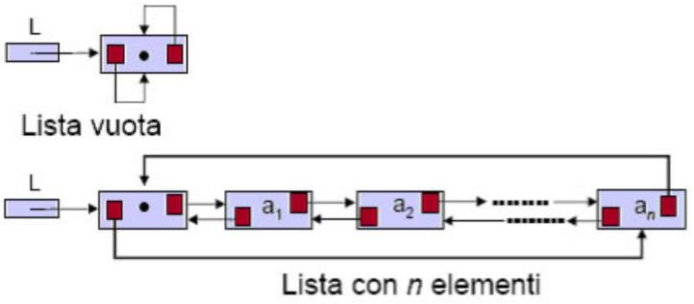


Tipi di liste

Lista bidirezionale (può essere visitata nei due sensi -> presenza di due puntatori per nodo)



Lista bidirezionale circolare (il puntatore all'elemento successivo dell'ultimo nodo punta al primo elemento della lista e il puntatore all'elemento precedente del primo punta all'ultimo elemento della lista)



Operazioni su liste

- Creazione della lista (vuota e successivi inserimenti)
- Lettura di una lista
- Stampa di una lista
- Cancellazione di una lista
- Inserimento in lista
- Estrazione da lista

Struttura del nodo:

```
typedef struct Nodo_SL {
    int dato;
    struct Nodo_SL *next;
} Nodo_SL;
```

Struttura della lista:

```
typedef struct Lista_SL {
  Nodo_SL *next;
} Lista_SL;
```

Creazione della lista

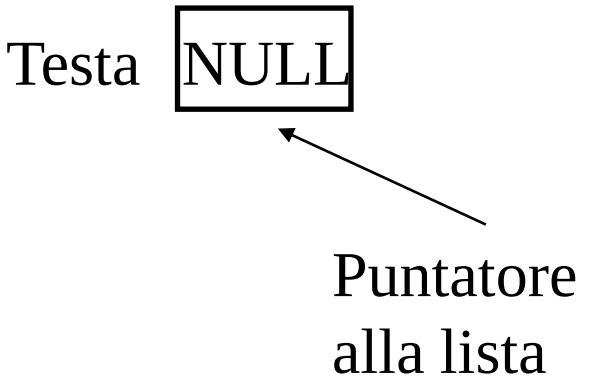
Per creare una lista, basta definirla, ovvero è sufficiente creare il modo di riferirsi ad essa.

L'unica cosa che esiste sempre della lista è la sua **testa** (o **radice**) ossia il puntatore al suo primo elemento.

Questa è l'unica componente **allocata staticamente** ed è inizializzata a **NULL** poiché all'inizio (creazione della lista) non punta a niente in quanto non ci sono elementi.

Es.:

```
Lista_SL Testa;

Testa.next = NULL;
```

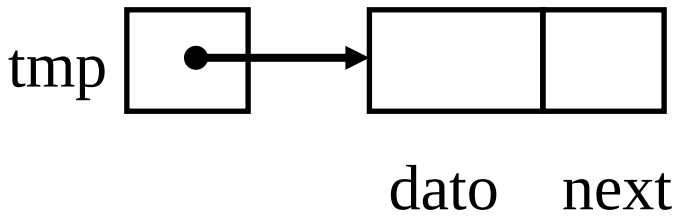


Creazione di un nuovo nodo

La creazione di un **nuovo nodo** (in qualunque fase dell'esistenza di una lista) avviene creando una nuova istanza della struttura tramite **allocazione dinamica**, utilizzando di solito un puntatore di appoggio (**tmp**)

Es.:

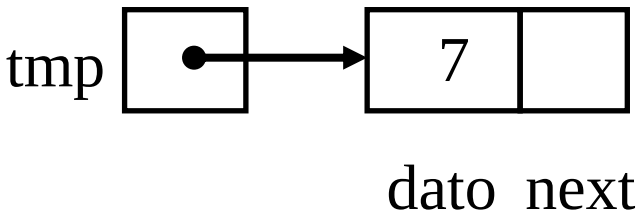
```
Nodo_SL *tmp = (Nodo_SL *)malloc(sizeof(Nodo_SL));
```



Assegnazione di valori ai campi dati

L'assegnazione di valori ai campi dati si ottiene dereferenziando il puntatore al nodo e accedendo ai singoli dati, ovvero utilizzando direttamente l'operatore ->

```
Nodo_SL *CreaNodo_SL(int dato)
{
   Nodo_SL *tmp;

   tmp = (Nodo_SL *)malloc(sizeof(Nodo_SL));
   if(!tmp)
     return NULL;
   else {
     tmp->next = NULL;
     tmp->dato = dato;
   }

   return tmp;
}
```



Assegnazione di valori ai campi dati

Le operazioni di inserimento di un elemento (ed analogamente quelle di cancellazione) possono avvenire secondo diverse modalità, (ovvero in diverse posizioni della lista) assumendo di volta in volta caratteristiche specifiche.

In ogni caso l'inserimento di un nuovo elemento nella lista prevede sempre i seguenti passi:

1) Creazione di un nuovo nodo (allocazione dinamica)
2) Assegnazione di valori ai campi dati
3) Collegamento del nuovo elemento alla lista esistente
   - aggiornamento del campo puntatore del nodo
   - aggiornamento dei puntatori della lista

Queste due ultime operazioni caratterizzeranno la tipologia dell'inserimento

Inserimento in testa

Il caso più semplice è costituito dall'**inserimento in testa**, in quanto si dispone di un riferimento esplicito a questa (la testa della lista **Testa**).

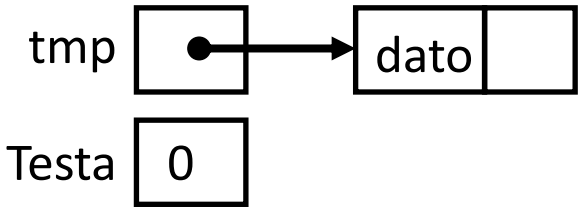
- Il campo **next** del nuovo nodo punterà allo stesso valore a cui punta il campo **next** di **Testa**

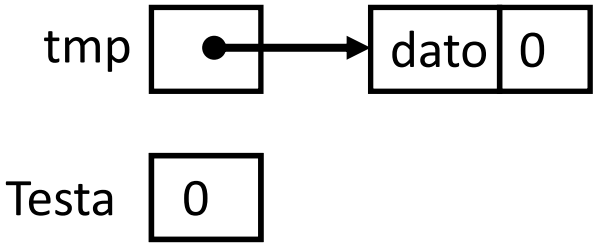
- **Testa** sarà collegato al nuovo nodo

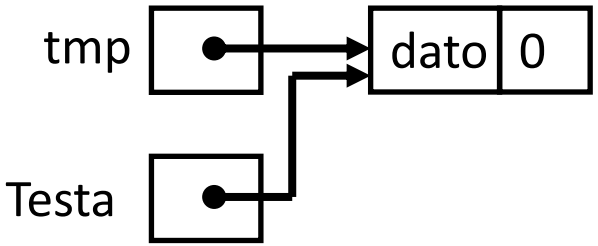
```
tmp->next = Testa.next;
Testa.next = tmp;
```

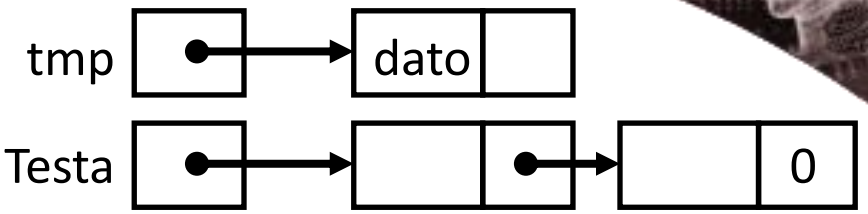
NB. Funziona anche se la lista è vuota!

Inserimento in testa

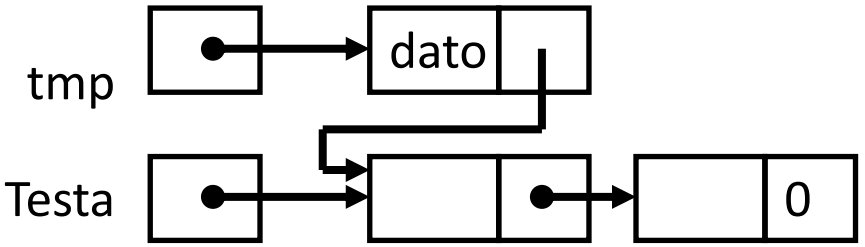




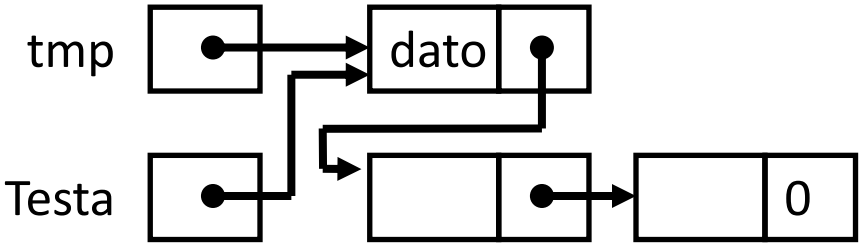




tmp->next = **Testa.next;**

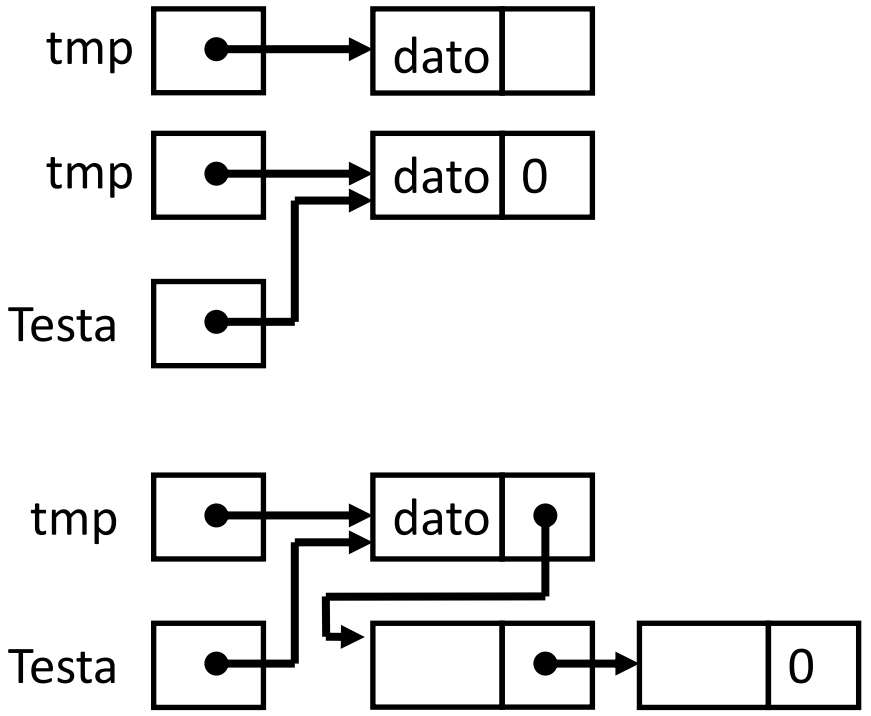


Testa.next = tmp;



Assegnazione di valori ai campi dati

```
void InserisciInTesta_SL(Lista_SL *Testa, int dato)
{
   Nodo_SL *temp = NULL;

   if(!Testa->next){
      temp = CreaNodo_SL(dato);

      if(temp){
         Testa->next = temp;
         return;
      }
   } else {
      temp = CreaNodo_SL(dato);

      if(temp){
         temp->next = Testa->next;
         Testa->next = temp;
      }
   }
   return;
}
```



Ricerca dell'ultimo elemento

Per cercare l'ultimo elemento, possiamo scorrere la lista tramite un puntatore ausiliario **p**, inizializzato a **Testa.next**.

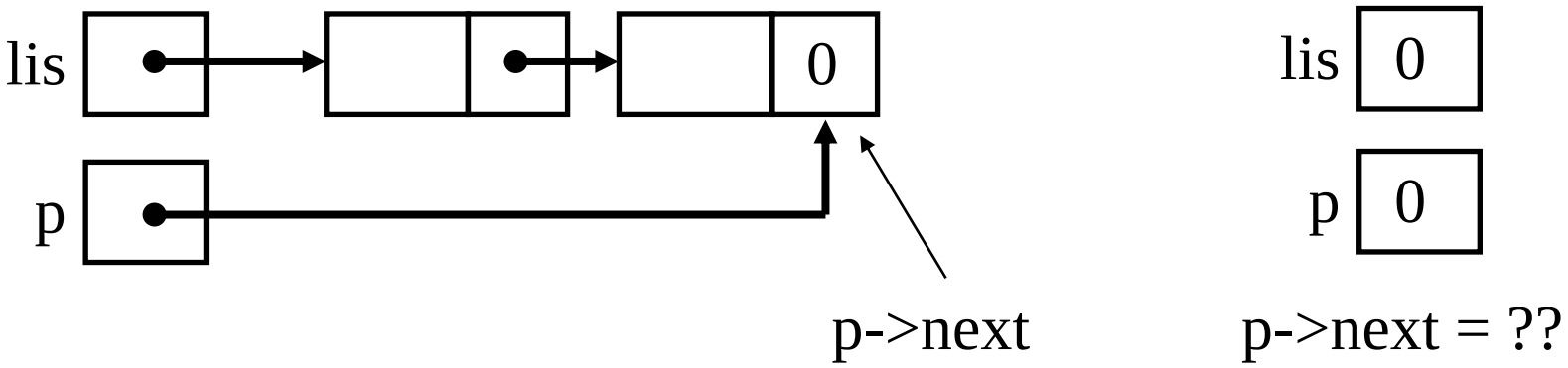
Es:

```
Nodo_SL *p = Testa.next;
while (p->next != NULL)
   p = p->next;
```

Dove è l'errore?

Ricerca dell'ultimo elemento

La condizione (`p->next != NULL`) nel caso la lista sia vuota conduce ad un **errore fatale** in quanto si sta dereferenziando un puntatore nullo.

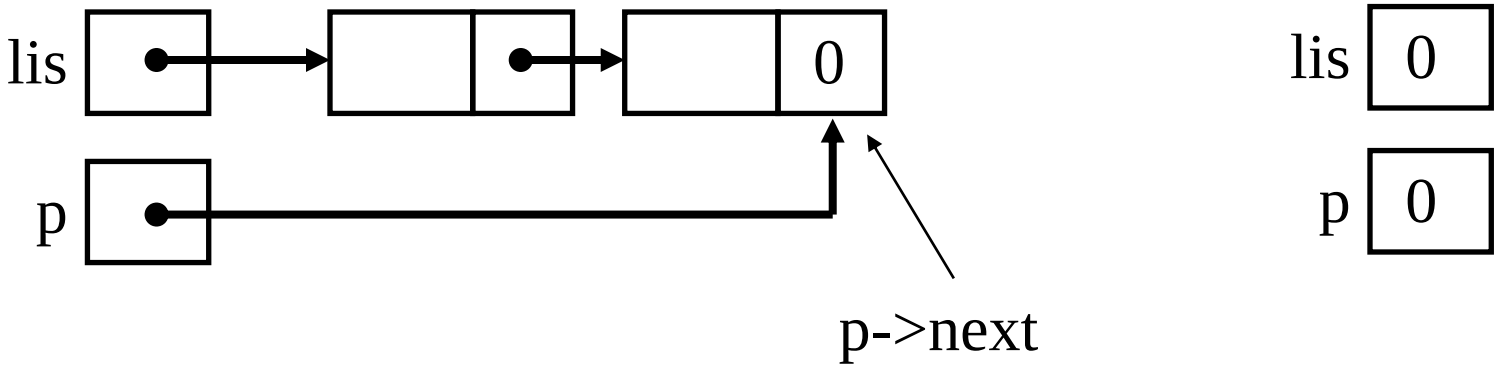


Questo è un **tipico** errore che si fa nella gestione delle strutture dinamiche!

Ricerca dell'ultimo elemento

La procedura corretta è:

```
Nodo_SL *p= Testa.next;
while(p!=NULL && p->next!=NULL)
    p = p->next;
```



Inserimento in coda

L'**inserimento in coda** è più complesso, in quanto non abbiamo un puntatore esplicito all'ultimo elemento, ma dobbiamo prima scorrere la lista per cercarlo.

Supponiamo di averlo trovato e che sia il puntatore **p**:
Il campo **next** del nuovo nodo punterà a **NULL** (in quanto è l'ultimo)

Il campo **next** dell'ex ultimo nodo punterà al nuovo nodo

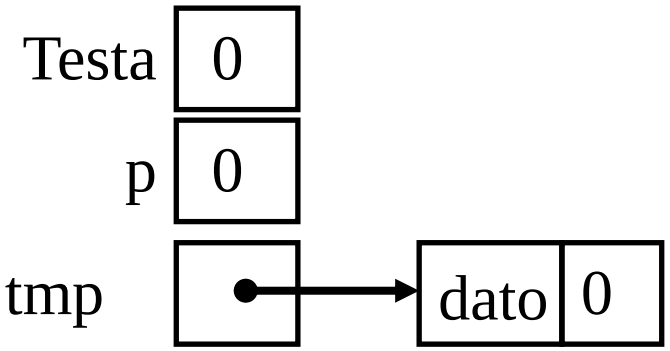
Es.:

```
tmp->next = NULL;
p->next = tmp;
```

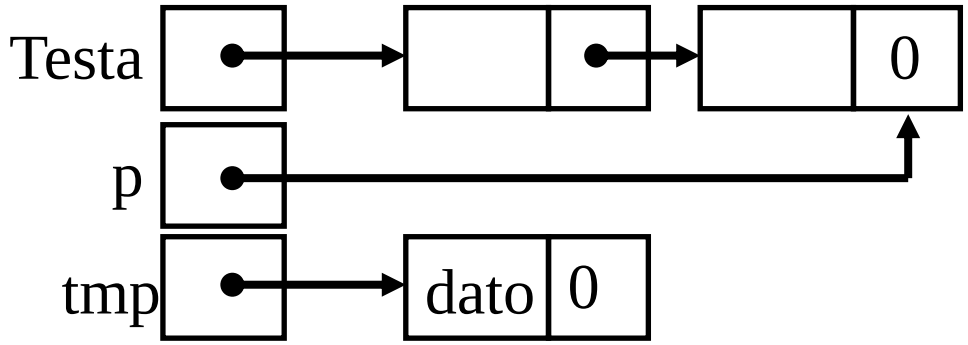
La lista vuota va gestita come un caso particolare!

Inserimento in coda

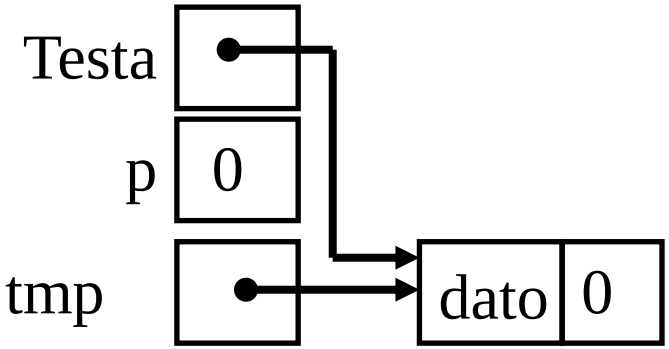
(NULL)



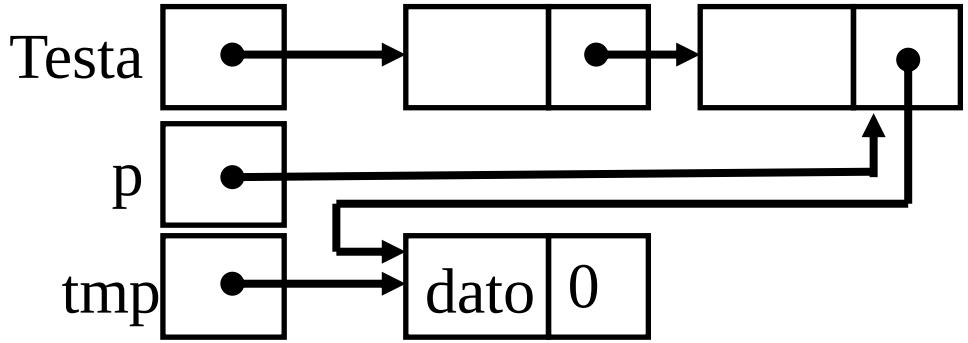
tmp->next = p;



Testa.next = tmp;

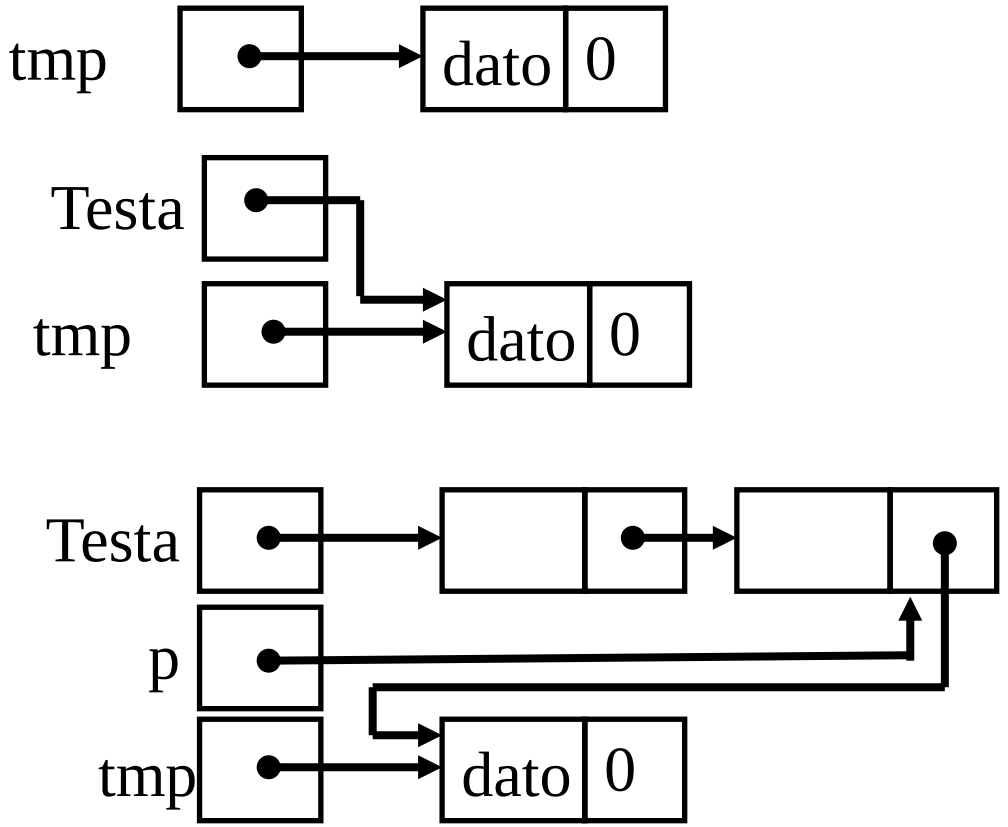


p->next = tmp;



Assegnazione di valori ai campi dati

```
void InserisciInCoda_SL(Lista_SL *Testa, int dato)
{
   Nodo_SL *temp = NULL;
   Nodo_SL *p = Testa->next;


   if(!Testa->next){
      temp = CreaNodo_SL(dato);

      if(temp){
         Testa->next = temp;
         return;
      }

   } else {
      temp = CreaNodo_SL(dato);

      while(p->next)
         p = p->next;

      p->next = temp;
   }
   return;
}
```



Liberare la lista

Liberare la lista consiste nel deallocare sequenzialmente tutti i nodi presenti nella lista.

```
void LiberaLista_SL(Lista_SL *Testa)
{
   Nodo_SL *tmp = Testa->next;

   while(Testa->next){
      tmp = Testa->next;
      Testa->next = tmp->next;
      printf("\nElimino %d", tmp->dato);
      free(tmp);
   }

   Testa->next = NULL;
}
```

Stampa degli elementi nella lista

Stampare la lista consiste nel mostrare a video sequenzialmente tutti i campi nei nodi della lista.

```
void StampaLista_SL(Lista_SL *Testa)
{
    Nodo_SL *tmp = Testa->next;

    while(tmp && tmp->next){
        printf("%d -> ", tmp->dato);
        tmp = tmp->next;
    }

    if(tmp)
        printf("%d -|", tmp->dato);
}
```

Costruiamo una libreria per le liste

Header file

```
#ifndef _LISTE_H
#define _LISTE_H

#include <stdio.h>
#include <stdlib.h>

typedef struct Nodo_SL {
   int dato;
   struct Nodo_SL *next;
} Nodo_SL;

typedef struct Lista_SL {
   Nodo_SL *next;
} Lista_SL;

Nodo_SL *CreaNodo_SL(int dato);
void StampaLista(Lista_SL Testa);
void InserisciInTesta(Lista_SL *Testa, int dato);
void InserisciInCoda(Lista_SL *Testa, int dato);
void LiberaLista(Lista_SL *Testa);

#endif
```

Costruiamo una libreria per le liste

Main

```c
#include <stdio.h>
#include <stdlib.h>
#include "liste.h"

int main(int argc, char *argv[])
{
   int i;
   Lista_SL Testa;
   Nodo_SL *tmp = NULL;


   for(i=1; i<argc; i++){
      InserisciInTesta_SL(&Testa,
                          atoi(argv[i]));
   }

   StampaLista_SL(Testa);

   LiberaLista_SL(&Testa);
```

```
>Lista.exe 4 7 6 9
```

```
[Lista]-> 9 -> 6 -> 7 -> 4 -|
Elimino 9
Elimino 6
Elimino 7
Elimino 4
[Lista]-> 4 -> 7 -> 6 -> 9 -|
Elimino 4
Elimino 7
Elimino 6
Elimino 9
```

```c
   for(i=1; i<argc; i++){
       InserisciInCoda_SL(&Testa,
                          atoi(argv[i]));
   }

   StampaLista_SL(Testa);

   LiberaLista_SL(&Testa);

   return 0;
}
```