Programmazione I

Il Linguaggio C

Strutture Dati - Liste

Daniel Riccio

Università di Napoli, Federico II

03 dicembre 2021

Sommario

- Argomenti
  - Liste semplicemente concatenate

Inserimento in una posizione specifica

L’inserimento in una posizione specifica richiede preventivamente l’individuazione di tale posizione all’interno della lista e dipende dalla condizione che si vuole verificare, per cui dobbiamo prima scorrere la lista per determinarla.

Nel caso di inserimento in ordine crescente, la lista risultante deve rimanere in ogni momento ordinata.

Pertanto, all’inserimento di un nuovo valore, si dovrà scorrere la lista fino alla posizione corretta per l’inserimento (fin quando cioè il campo **`dato`** dei nodi esistenti risulta minore del dato da inserire).

Ricerca di un elemento qualsiasi

La condizione più sicura da utilizzare in una ricerca è riferirsi direttamente al puntatore all'elemento nella condizione di scorrimento.

In tal modo però **si sorpassa l'elemento cercato**.

Per questo nella ricerca della posizione di inserimento si usano di solito due puntatori, **p** e **q**, che puntano rispettivamente all'elemento precedente e al successivo.

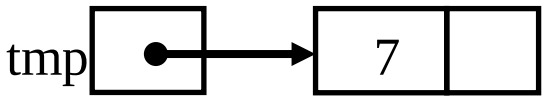
Es:

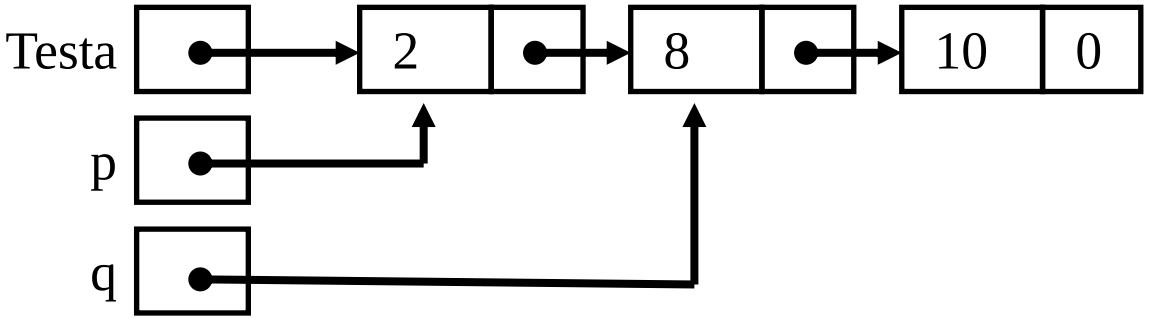
```
Nodo_SL *q = Testa.next;
Nodo_SL *p = Testa.next;
  while(q!=NULL && q->dato<dato){
     p=q;
     q=q->next;
  }
```

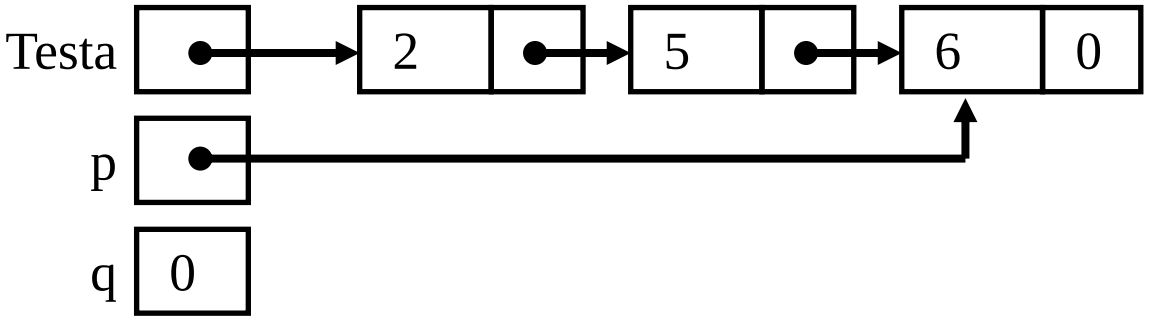
Ricerca di una posizione specifica

Es:

```
Nodo_SL *q = Testa.next;
Nodo_SL *p = Testa.next;
  while(q!=NULL && q->dato<dato){
      p=q;
      q=q->next;
  }
```



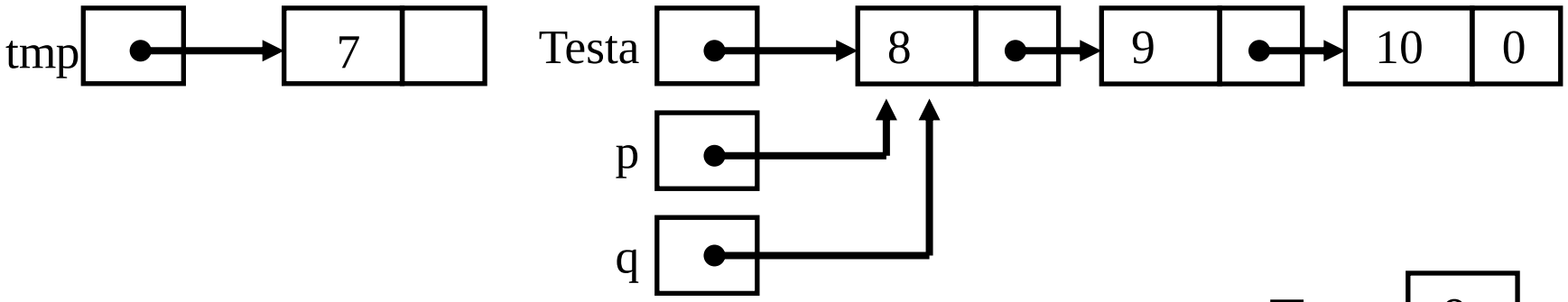


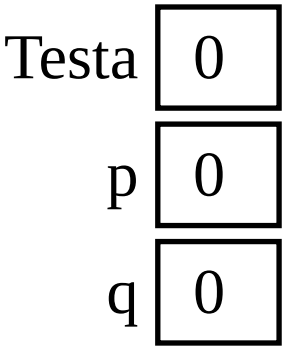


Casi particolari

Es:

```
Nodo_SL *q = Testa.next;
Nodo_SL *p = Testa.next;
  while(q!=NULL && q->dato<dato){
     p=q;
     q=q->next;
  }
```



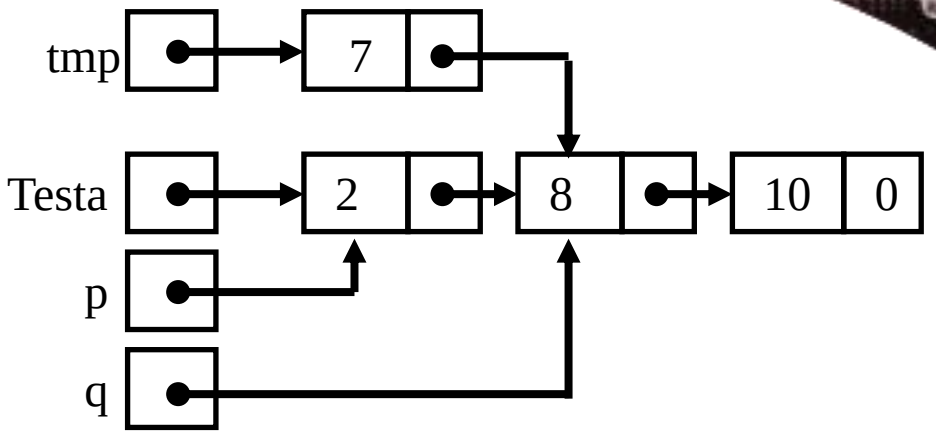


Se **q==Testa.next** allora **p** non contiene l'elemento precedente

Casi particolari

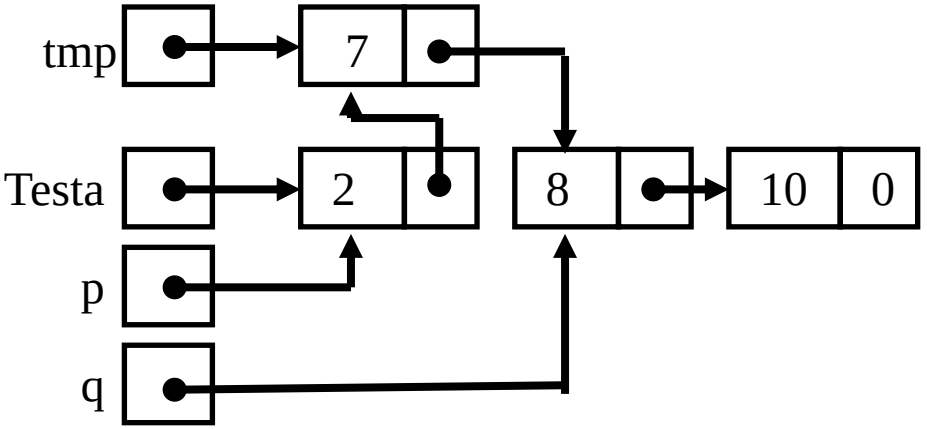
Quindi nel caso generale:

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



ma se `q==Testa->next`
(inserimento in testa)

```
Testa.next = tmp;
```



Ricerca del nodo predecessore

```
Nodo_SL *CercaPredecessore_SL(Lista_SL Testa, int dato)
{
   Nodo_SL *q = Testa.next;
   Nodo_SL *p = Testa.next;

   while(q!=NULL && (q->dato < dato)){
      p=q;
      q=q->next;
   }

   return p;
}
```

Ricerca di un dato specifico

```
Nodo_SL *CercaElemento_SL(Lista_SL Testa, int dato)
{
   Nodo_SL *q = Testa.next;

   while(q!=NULL && (q->dato != dato))
      q=q->next;

   return q;
}
```

Inserimento in ordine crescente

L'inserimento di valori in ordine crescente è effettuata inserendo i nuovi elementi subito dopo averne trovato il predecessore.

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

   if(Testa->next == NULL || Testa->next->dato>dato)
      InserisciInTesta_SL(Testa, dato);
   else {
      nuovo = CreaNodo_SL(dato);
      temp = CercaPredecessore_SL(*Testa, dato);

      nuovo->next = temp->next;
      temp->next = nuovo;
   }

   return;
}
```

Inserimento dopo un elemento

L'inserimento di un valore dopo un elemento specifico è effettuata cercando il valore e inserendo l'elemento come nodo successivo.

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

   if(Testa->next == NULL)
      return;
   else {
        temp = CercaElemento_SL(*Testa, predecessore);

      if(temp){
        nuovo = CreaNodo_SL(dato);
         nuovo->next = temp->next;
         temp->next = nuovo;
      }
   }

   return;
}
```

Ordine decrescente

Supponiamo di voler creare una lista contenente i valori

$$V=\{5, 9, 7, 12, 0\}, n=|V|$$

in ordine decrescente.

```
Lista_SL Testa_tmp;
Lista_SL Testa;

Nodo_SL *tmp = NULL;
```

```
Testa_tmp.next = NULL;
for(i=0; i<5; i++){
   InserisciInOrdine_SL(&Testa_tmp, V[i]);
}

StampaLista_SL(Testa_tmp);

tmp = Testa_tmp.next;
Testa.next = NULL;
while(tmp){
   InserisciInTesta_SL(&Testa, tmp->dato);
   tmp = tmp->next;
}

StampaLista_SL(Testa);
```

Eliminazione di un nodo

L'eliminazione di un nodo dalla lista prevede:

- Ricerca del nodo da eliminare (se necessaria)
- Salvataggio del nodo in una variabile ausiliaria
- Scollegamento del nodo dalla lista (aggiornamento dei puntatori della lista)
- Distruzione del nodo (deallocazione della memoria)

In ogni caso, bisogna verificare inizialmente che la lista non sia già vuota!

```
if(Testa.next != NULL)
```

Eliminazione di un nodo

Dipende dalle esigenze del programma.

Come per l'inserimento, il caso più semplice è costituito dall'eliminazione del nodo di testa, in quanto esiste il puntatore `Testa.next` a questo elemento.

Negli altri casi, si procede come per l'inserimento

Eliminazione del nodo di testa

Bisogna aggiornare il puntatore alla testa `Testa.next` che dovrà puntare al nodo successivo a quello da eliminare.

salvataggio del nodo da eliminare:

```
Node_SL *tmp = Testa.next;
```

aggiornamento della lista:

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

distruzione del nodo:

```
free(tmp);
```

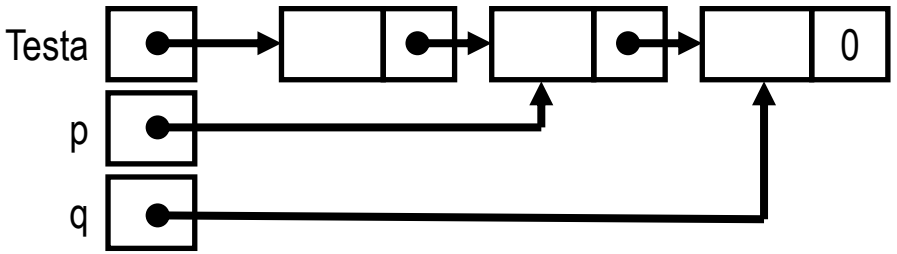
Eliminazione del nodo di coda

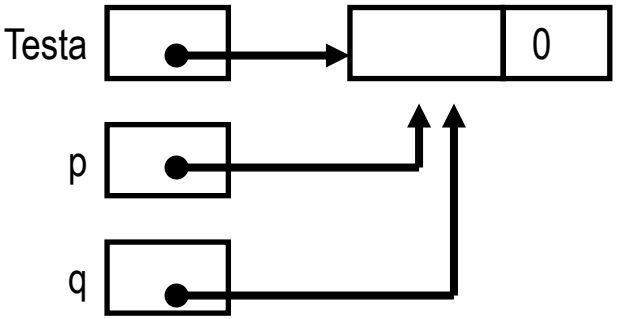
Bisogna aggiornare il campo **next** relativo al penultimo nodo, che ora diventa l'ultimo (e quindi assume valore **NULL**).

Per questo nella ricerca della posizione di eliminazione si usano di solito due puntatori, **p** e **q**, che puntano rispettivamente all'elemento precedente e al successivo.

Es:

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



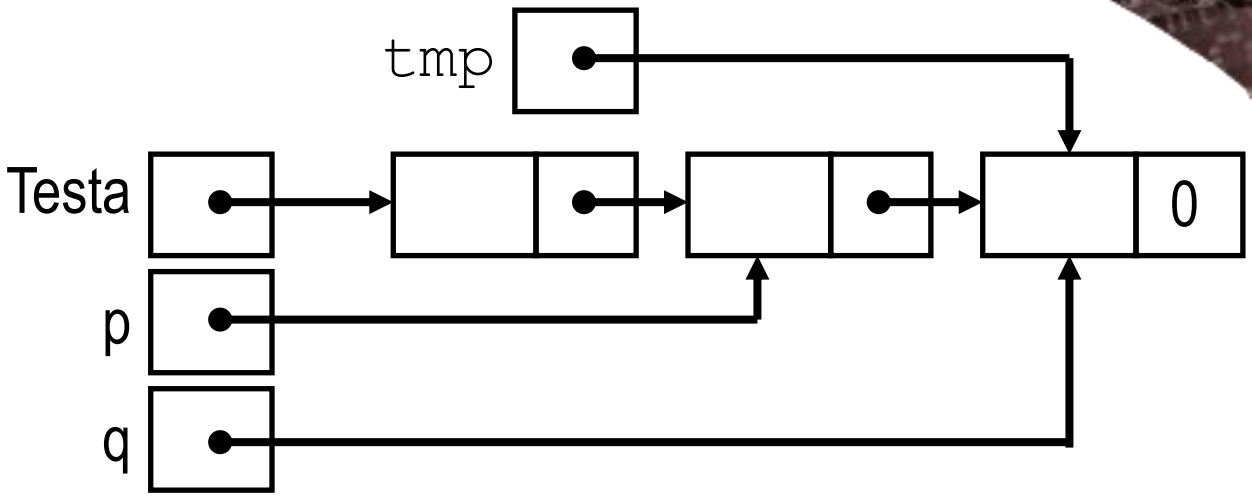


Casi particolari:

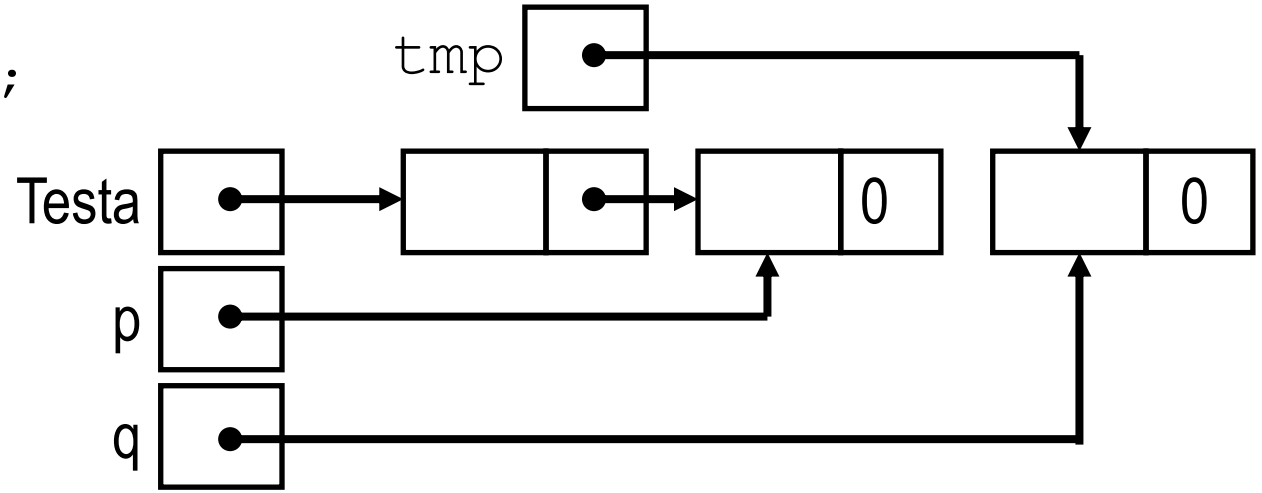
l'elemento da eliminare è l'unico della lista.

Eliminazione del nodo di coda

```
Nodo_SL *tmp=q;
```




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

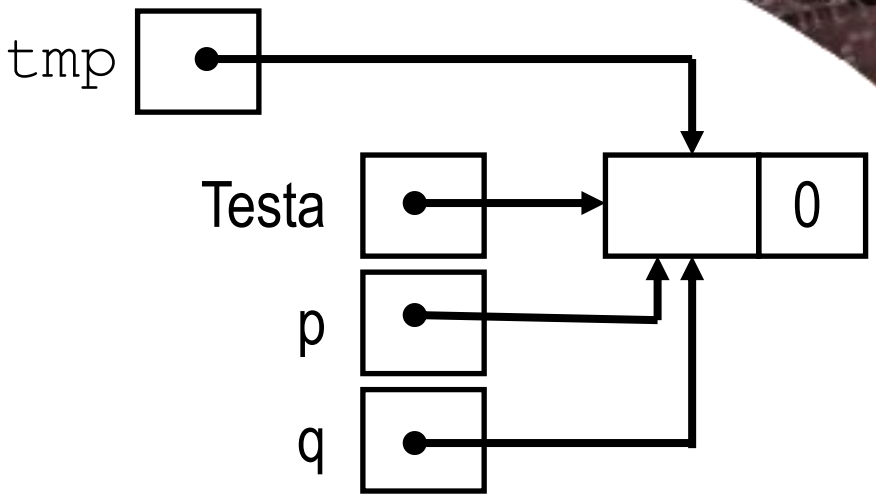


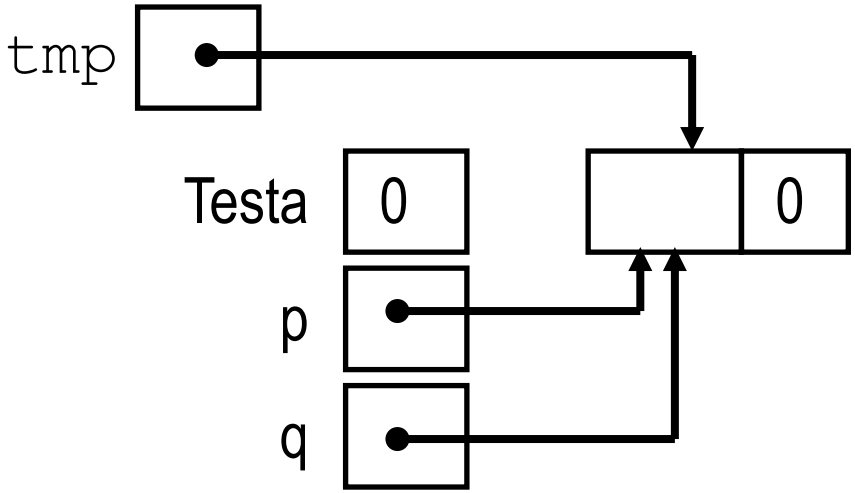
Eliminazione del nodo di coda

Se **q==Testa.next** allora **p** non contiene l'elemento successivo

```
Nodo_SL *tmp=q;
```

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





Eliminazione del nodo di coda

```
void CancellaInCoda_SL(Lista_SL *Testa){
   Nodo_SL *q = Testa->next;
   Nodo_SL *p = Testa->next;

   if(!q)
      return;
   else {

      while(q!=NULL && q->next!=NULL){
         p=q;
         q=q->next;
      }

      p->next = NULL;
      free(q);
   }
}
```

Eliminazione di un nodo

Individuato il nodo, bisogna evitare che la sua rimozione spezzi la lista.

salvataggio del nodo da eliminare:

```
Node_SL *pre  = CercaPredecessore_SL(Testa, dato);
Node_SL *nodo = pre->next;
```

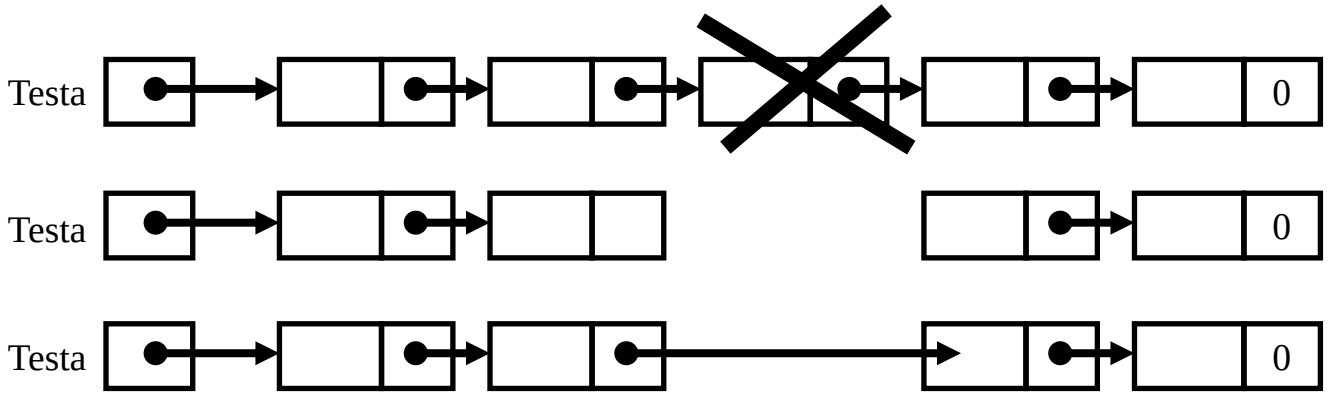
aggiornamento della lista:

```
pre->next = nodo->next;
```

distruzione del nodo:

```
free(nodo);
```

In generale, è necessario aggiornare il puntatore **next** dell'elemento **precedente**



Esercizi – partizione di liste

Scrivere una funzione che prenda un elenco di numeri a linea di comando e li inserisca in una lista. Successivamente, si scriva una funzione che separi la lista originaria in due liste, una con i numeri pari e l'altra con i numeri dispari.

```
void PartizionaLista(Lista_SL Testa, Lista_SL *Lpari, Lista_SL *Ldispari);

void PartizionaLista(Lista_SL Testa, Lista_SL *Lpari, Lista_SL *Ldispari)
{
   Nodo_SL *nodo = NULL;

   while(Testa.next){
      nodo = Testa.next;
      Testa.next = Testa.next->next;
      nodo->next = NULL;

      if(nodo->dato % 2 == 0)
        InserisciInTesta_SL(Lpari, nodo->dato);
      else
        InserisciInTesta_SL(Ldispari, nodo->dato);
   }
}
```

Esercizi – partizione di liste

```
int main(int argc, char *argv[])
{
   int i;
   Lista_SL lista;
   Lista_SL lista_pari;
   Lista_SL lista_dispari;


   lista.next = NULL;
   for(i=1; i<argc; i++){
      InserisciInTesta_SL(&lista, atoi(argv[i]));
   }

   StampaLista_SL(lista);

   lista_pari.next = NULL;
   lista_dispari.next = NULL;
   PartizionaLista(lista, &lista_pari, &lista_dispari);

   StampaLista_SL(lista_pari);
   StampaLista_SL(lista_dispari);
}
```