Programmazione I

Il Linguaggio C

Strutture Dati - Liste

Daniel Riccio

Università di Napoli, Federico II

6 dicembre 2021

Sommario

- Argomenti
  - Liste doppiamente concatenate

Liste semplicemente concatenate

Pro

Permettono di utilizzare in modo efficiente la memoria in fase di esecuzione. Viene allocata solo la memoria richiesta per rappresentare il contenuto attuale della lista

Le operazioni di inserimento e rimozione sono efficienti. Non è necessario riorganizzare completamente la struttura dati.

Contro

È possibile attraversare la lista in una sola direzione: dalla testa verso la coda. Operazioni che richiedano l'attraversamento dalla coda verso la testa sono complesse e generalmente costose.

Liste doppiamente concatenate

Le liste **doppiamente concatenate** estendono la rappresentazione delle liste concatenate, introducendo un puntatore al nodo precedente.

A differenza delle liste concatenate semplici, le liste doppiamente concatenate possono essere attraversate facilmente in entrambe le direzioni.

Le implementazioni di alcune funzioni sono semplificate dall'introduzione del nuovo puntatore.

Aggiungono un ulteriore overhead di memoria alla struttura dati: è necessario memorizzare due puntatori distinti in ogni nodo.

Non offrono vantaggi rispetto alle liste concatenate semplici relativamente all'accesso alla coda della lista.

Liste doppiamente concatenate

Possiamo rappresentare il tipo di dato lista come puntatore a struttura contente:

- Un campo (dato) di tipo in int
- Un campo puntatore (prev) ad una struttura dello stesso tipo.
- Un campo puntatore (next) ad una struttura dello stesso tipo.

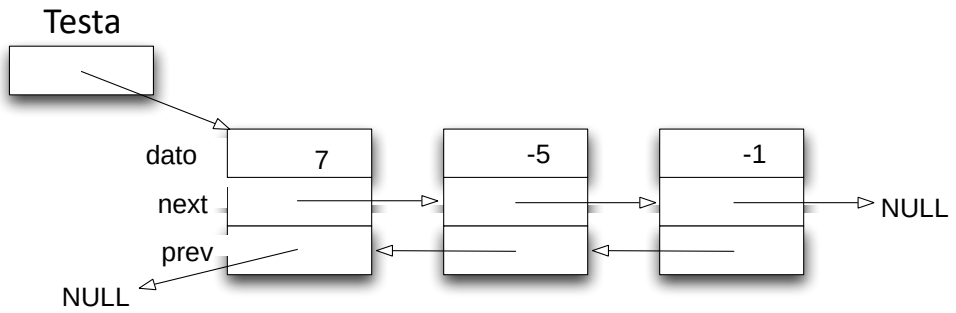
Struttura del nodo:

```
typedef struct Nodo_DL {
    int dato;
    struct Nodo_DL *prev;
    struct Nodo_DL *next;
} Nodo_DL;
```

Struttura della lista:

```
typedef struct Lista_DL {
  Nodo_DL *next;
} Lista_DL;
```

Da un nodo è possibile accedere sia al nodo successivo (campo next) che al nodo precedente (campo prev) nella lista



Creazione della lista

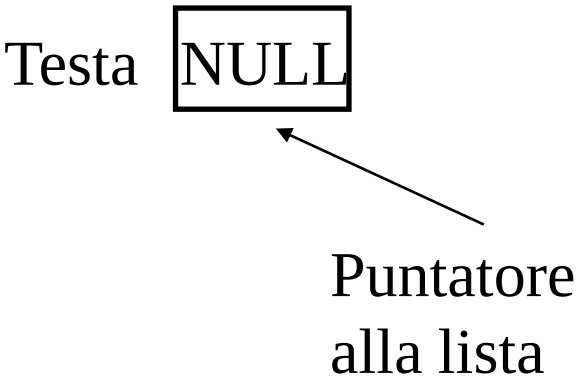
Per creare una lista, basta definirla, ovvero è sufficiente creare il modo di riferirsi ad essa.

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_DL Testa;

Testa.next = NULL;
```

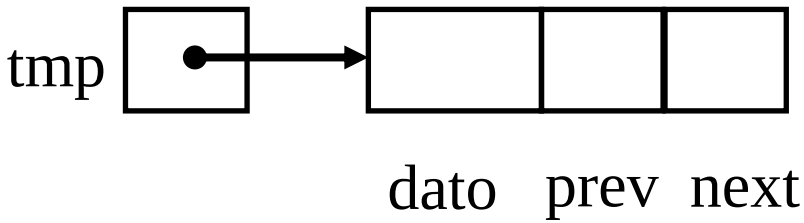


Creazione di un nuovo nodo

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_DL *tmp = (Nodo_DL *)malloc(sizeof(Nodo_DL));
```



Creazione di un nodo

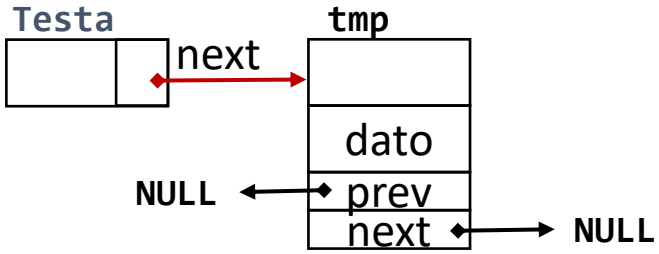
L'assegnazione di valori ai campi dati si ottiene dereferenziando il puntatore al nodo e accedendo ai singoli dati, ovvero utilizzando direttamente l'operatore ->

```
Nodo_DL *CreaNodo_DL(int dato)
{
   Nodo_DL *tmp;

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

   return tmp;
}
```

La funzione **CreaNodo_DL** è essenzialmente implementa come visto per liste concatenate. Il nodo di una lista concatenata contiene il puntatore al nodo precedente, oltre al puntatore al nodo successivo, per cui è necessario inizializzare a **NULL** entrambi i puntatori nella struttura **Nodo_DL**.

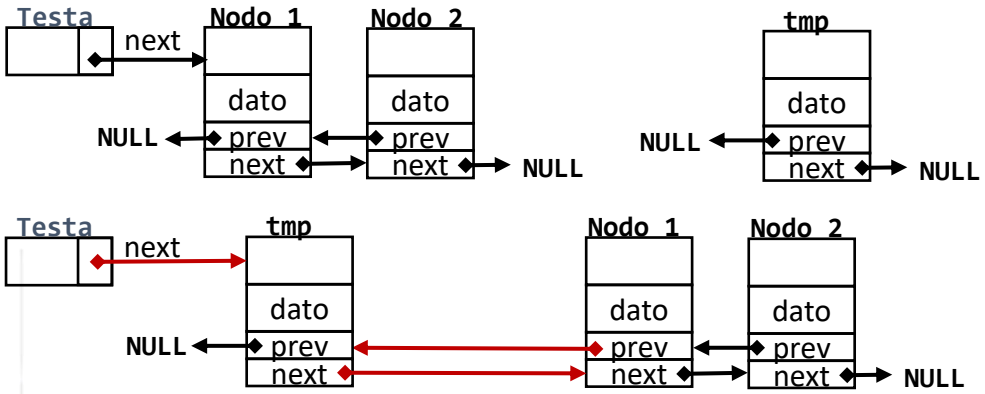
Inserimento in testa

Se la lista è vuota, si inserisce il nuovo nodo come unico nodo puntato da **Testa**.**next**.




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

Se la lista non è vuota, si inserisce il nuovo nodo come primo nodo puntato da **Testa**.**next**, e si aggiornano i puntatori del nodo in testa e del nodo appena inserito



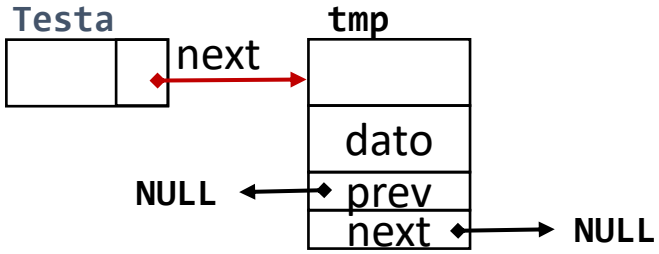

```
tmp->next = Testa.next;
Testa.next->prev = tmp;
Testa.next = tmp;
```

Inserimento in testa

```
void InserisciInTesta_DL(Lista_DL *Testa, int dato)
{
   Nodo_DL *temp = NULL;

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

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

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

   return;
}
```

Inserimento in coda

Se la lista è vuota, si inserisce il nuovo nodo come unico nodo puntato da **Testa.next**.

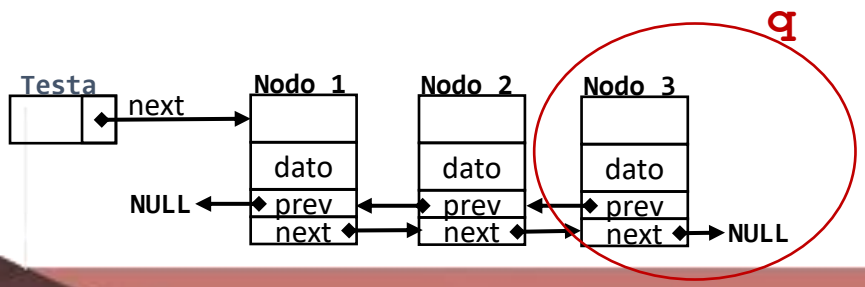


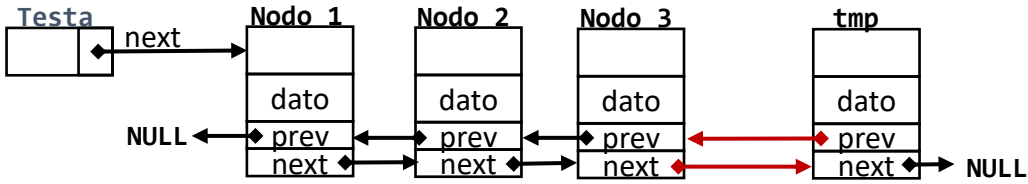

```
Testa.next = tmp;
```

Se la lista non è vuota, troviamo il nodo di coda scorrendo la lista a partire da **Testa.next**.

```
Nodo_DL *q = Testa.next;
while(q!=NULL && q->next!=NULL)
    q=q->next;
```

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





Inserimento in coda

```
void InserisciInCoda_DL(Lista_DL *Testa, int dato)
{
   Nodo_DL *temp = NULL;
   Nodo_DL *q = Testa->next;

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

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

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

      q->next = temp;
      temp->prev = q;
   }

   return;
}
```

Ricerca di un dato specifico

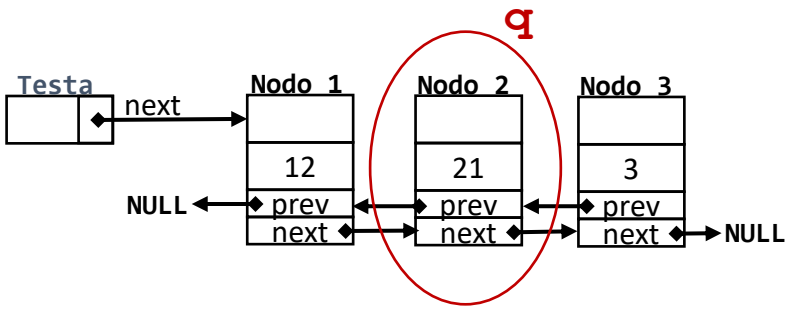
```
Nodo_DL *CercaElemento_DL(Lista_DL Testa, int dato)
{
   Nodo_DL *q = Testa.next;

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

   return q;
}
```

Inserimento dopo un elemento

Se la lista è vuota, l'elemento non esiste e il nuovo nodo non viene inserito. In alternativa, se l'elemento viene trovato, si crea un nuovo nodo e si inserisce dopo tale elemento, aggiornando i puntatori.

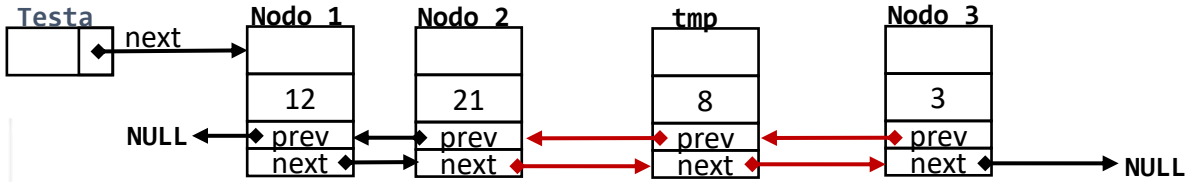
```
InserisciDopoElemento_DL(Testa, 21, 8);
```

```
q = CercaElemento_DL(Testa, 21);
```




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

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

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

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



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_DL(Lista_DL *Testa, int predecessore, int dato)
{
   Nodo_SL *temp = NULL;
   Nodo_SL *nuovo = NULL;

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

      if(temp){
         nuovo = CreaNodo_DL(dato);
         nuovo->prev = temp;
         temp->next->prev = nuovo;
         temp->next = nuovo;
         nuovo->next = temp->next;
      }
   }
   return;
}
```

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_DL *tmp = Testa.next;
```

aggiornamento della lista:

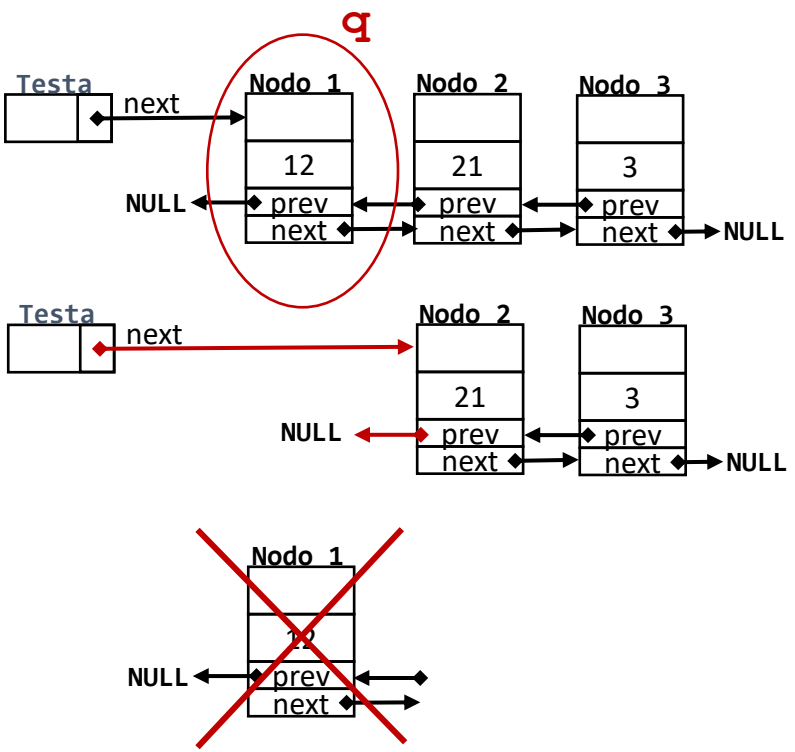
```
Testa.next = tmp->next;
tmp->next->prev = NULL;
```

distruzione del nodo:

```
free(tmp);
```

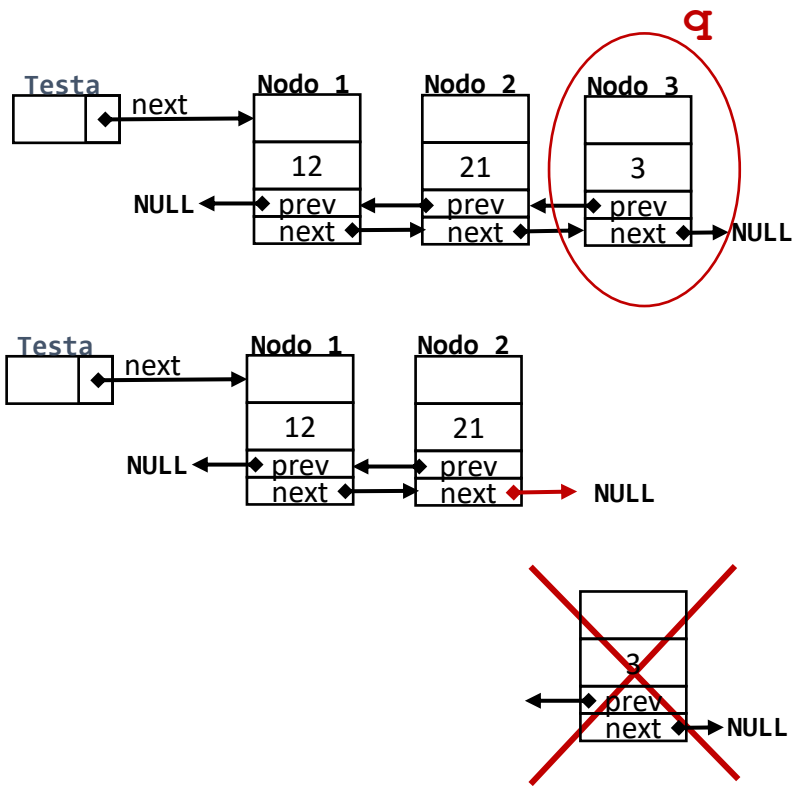
Eliminazione del nodo di testa

```
void CancellaInTesta_DL(Lista_DL *Testa)
{
   Nodo_DL *q = Testa->next;

   if(!q)
      return;
   else {

      if(q->next){
         Testa->next = q->next;
         Testa->next->prev = NULL;
      }else

      Testa->next = NULL;

      free(q);
   }
}
```



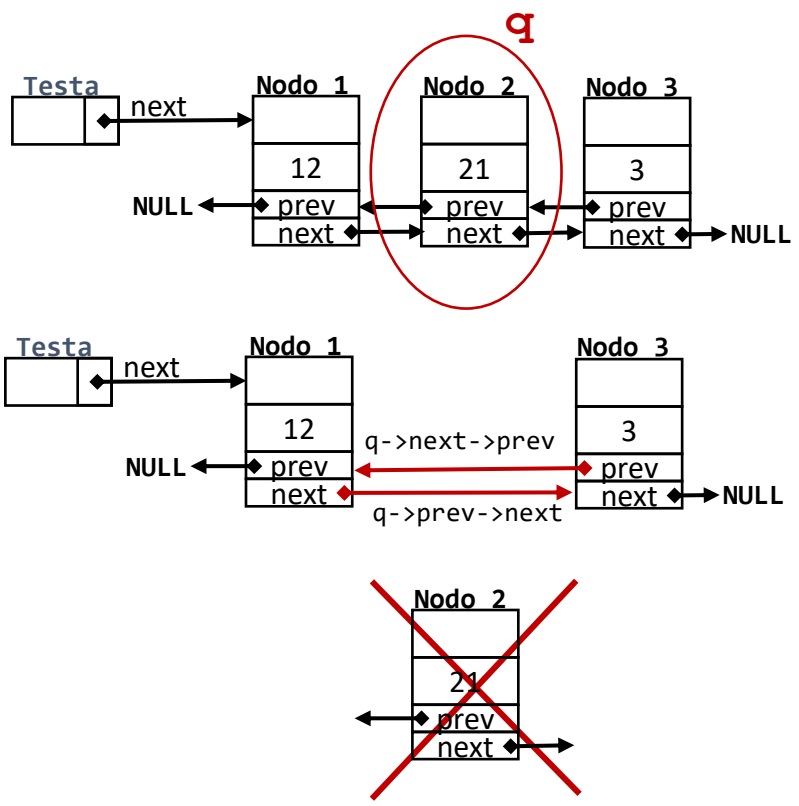
Eliminazione del nodo di coda

```
void CancellaInCoda_DL(Lista_DL *Testa)
{
   Nodo_DL *q = Testa->next;

   if(!q)
      return;
   else {

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

      if(q->prev)
       q->prev->next = NULL;

      free(q);
   }
}
```



Eliminazione di un nodo generico

```
void CancellaElemento_DL(Lista_DL *Testa, int dato)
{
   Nodo_DL *temp = NULL;

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

      if(temp){
         if(temp->next==NULL)
            CancellaInCoda_DL(Testa);
         else if(temp->prev==NULL)
            CancellaInTesta_DL(Testa);
         else {
            temp->prev->next = temp->next;
            temp->next->prev = temp->prev;
            free(temp);
         }
      }
   }
   return;
}
```



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;

typedef struct Nodo_DL {
int dato;
struct Nodo_DL *prev;
struct Nodo_DL *next;
} Nodo_DL;

typedef struct Lista_DL {
Nodo_DL *next;
} Lista_DL;
```

```
// Liste concatenate semplici
Nodo_SL *CreaNodo_SL(int dato);
void CreaLista_SL(Lista_SL *Testa);
void StampaLista_SL(Lista_SL Testa);
void InserisciInTesta_SL(Lista_SL *Testa, int dato);
void InserisciInCoda_SL(Lista_SL *Testa, int dato);
void InserisciInOrdine_SL(Lista_SL *Testa, int dato);
void InserisciDopoElemento_SL(Lista_SL *Testa, int predecessore, int dato);
void CancellaInCoda_SL(Lista_SL *Testa);
void CancellaInTesta_SL(Lista_SL *Testa);
void LiberaLista_SL(Lista_SL *Testa);
Nodo_SL *CercaPredecessore_SL(Lista_SL Testa, int dato);
Nodo_SL *CercaElemento_SL(Lista_SL Testa, int dato);

// Liste doppiamente concatenate
Nodo_DL *CreaNodo_DL(int dato);
void StampaLista_DL(Lista_DL Testa);
void InserisciInTesta_DL(Lista_DL *Testa, int dato);
void InserisciInCoda_DL(Lista_DL *Testa, int dato);
Nodo_DL *CercaElemento_DL(Lista_DL Testa, int dato);
void InserisciDopoElemento_DL(Lista_DL *Testa, int predecessore, int dato);
void CancellaInCoda_DL(Lista_DL *Testa);
void CancellaInTesta_DL(Lista_DL *Testa);
void CancellaElemento_DL(Lista_DL *Testa, int dato);
void LiberaLista_DL(Lista_DL *Testa);

#endif
```

Esempi

Main

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

int main(int argc, char *argv[])
{
   int i;
   Lista_DL Testa;

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

   StampaLista_DL(Testa);

   LiberaLista_DL(&Testa);
```

```
>ListaDL.exe 1 5 7 3 9

[Lista]-> 9 <-> 3 <-> 7 <-> 5 <-> 1 -|

Elimino 9
Elimino 3
Elimino 7
Elimino 5
Elimino 1
```

Esempi

Main

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

```
[Lista]-> 1 <-> 5 <-> 7 <-> 3 <-> 9 -|
```

```
StampaLista_DL(Testa);

LiberaLista_DL(&Testa);

InserisciDopoElemento_DL(&Testa, 7, 14);
```

```
[Lista]-> 1 <-> 5 <-> 7 <-> 14 <-> 3 <-> 9 -|
```

```
CancellaInCoda_DL(&Testa);
```

```
[Lista]-> 1 <-> 5 <-> 7 <-> 14 <-> 3 -|
```

```
CancellaInTesta_DL(&Testa);
```

```
[Lista]-> 5 <-> 7 <-> 14 <-> 3 -|
```

```
CancellaElemento_DL(&Testa, 14);
```

```
[Lista]-> 5 <-> 7 <-> 3 -|
```

Esercizi

Implementare i seguenti prototipi per le liste concatenate semplici.

unsigned int Length_SL(**Lista_SL** lista);
Restituisce il numero di elementi nella lista.

void Complemento_SL(**Lista_SL** *lista1, **Lista_SL** lista2);
Rimuove da L1 tutti gli elementi presenti in L2. Non modifica L2.

void Unione_SL(**Lista_SL** *lista1, **Lista_SL** lista2);
Aggiunge ad L1 tutti gli elementi in L2 non presenti in L1. Non modifica L2.

void Duplica_SL(**Lista_SL** lista1, **Lista_SL** *lista2);
Crea e restituisce una copia di L.

void Merge_SL(**Lista_SL** *lista1, **Lista_SL** lista2);
Aggiunge la lista lista2 in coda alla lista lista1. Non duplica lista2, la collega a lista1.

void Ordina_SL(**Lista_SL** *lista1);
Riorganizza gli elementi in lista1 in ordine crescente.

Esercizi

unsigned int Length_SL(Lista_SL lista);
Restituisce il numero di elementi nella lista.

```
unsigned int Length_SL(Lista_SL lista){
    Nodo_SL *tmp = lista.next;
    int i=0;

    while(tmp){
        ++i;
        tmp = tmp->next;
    }

    return i;
}
```

```
[Lista]-> 5 -> 9 -> 7 -> 12 -> 0 -|
[Lista]-> 9 -> 4 -> 6 -> 7 -> 10 -> 15 -> 12 -> 0 -|

La lista L1 contiene 5 nodi

La lista L2 contiene 8 nodi
```

Esercizi

void Complemento_SL(**Lista_SL** ***lista1**, **Lista_SL** lista2);

Rimuove da L1 tutti gli elementi presenti in L2. Non modifica L2.

```
void Complemento_SL(Lista_SL *lista1, Lista_SL lista2){
    Nodo_SL *tmp = lista2.next;

    while(tmp){
        CancellaElemento_SL(lista1, tmp->dato);
        tmp = tmp->next;
    }
}
```

```
[Lista]-> 5 -> 9 -> 7 -> 12 -> 0 -|
[Lista]-> 9 -> 4 -> 6 -> 7 -> 10 -> 15 -> 12 -> 0 -|
```

```
[Lista]-> 5 -|
```

Esercizi

void Unione_SL(Lista_SL *lista1, Lista_SL lista2);

Aggiunge ad L1 tutti gli elementi in L2 non presenti in L1. Non modifica L2.

```
void Unione_SL(Lista_SL *lista1, Lista_SL lista2){
   Nodo_SL *tmp = lista2.next;
   Nodo_SL *nodo;

   while(tmp){
      nodo = CercaElemento_SL(*lista1, tmp->dato);

      if(!nodo)
        InserisciInCoda_SL(lista1, tmp->dato);

      tmp = tmp->next;
   }
}
```

```
[Lista]-> 5 -|
[Lista]-> 9 -> 4 -> 6 -> 7 -> 10 -> 15 -> 12 -> 0 -|

[Lista]-> 5 -> 9 -> 4 -> 6 -> 7 -> 10 -> 15 -> 12 -> 0 -|
```

Esercizi

void Duplica_SL(**Lista_SL** lista1, **Lista_SL** ***lista2**);
Crea e restituisce una copia di L.

```
void Duplica_SL(Lista_SL lista1, Lista_SL *lista2){
   Nodo_SL *tmp = lista1.next;

   while(tmp){
      InserisciInCoda_SL(lista2, tmp->dato);
      tmp = tmp->next;
   }
}
```

Esercizi

void Merge_SL(Lista_SL *lista1, Lista_SL lista2);

Aggiunge la lista lista2 in coda alla lista lista1. Non duplica lista2, la collega a lista1.

```
void Merge_SL(Lista_SL *lista1, Lista_SL lista2){
   Nodo_SL *tmp = lista1->next;

   if(!tmp)
      lista1->next = lista2.next;
   else
      while(tmp && tmp->next)
         tmp = tmp->next;

   tmp->next = lista2.next;
}
```

```
[Lista]-> 5 -> 9 -> 4 -> 6 -> 7 -> 10 -> 15 -> 12 -> 0 -|
[Lista]-> 5 -> 9 -> 4 -> 6 -> 7 -> 10 -> 15 -> 12 -> 0 -|

[Lista]-> 5 -> 9 -> 4 -> 6 -> 7 -> 10 -> 15 -> 12 -> 0 -> 5 -> 9 -> 4 -> 6 -> 7 -> 10 -> 15 -> 12 -> 0 -|
```

Esercizi

void Ordina_SL(Lista_SL *lista1);

Riorganizza gli elementi in lista1 in ordine crescente.

```
void Ordina_SL(Lista_SL *lista1){
   Nodo_SL *tmp = lista1->next;
   Lista_SL lista_tmp;

   lista_tmp.next = NULL;

   if(!lista1->next)
      return;

   while(lista1->next){
      tmp = lista1->next;
      InserisciInOrdine_SL(&lista_tmp, tmp->dato);
      lista1->next = lista1->next->next;
      free(tmp);
   }

   lista1->next = lista_tmp.next;
}
```

```
[Lista]-> 5 -> 9 -> 4 -> 6 -> 7 -> 10 -> 15 -> 12 -> 0 -> 5 -> 9 -> 4 -> 6 -> 7 -> 10 -> 15 -> 12 -> 0 -|
[Lista]-> 0 -> 0 -> 4 -> 4 -> 5 -> 5 -> 6 -> 6 -> 7 -> 7 -> 9 -> 9 -> 10 -> 10 -> 12 -> 12 -> 15 -> 15 -|
```

Esercizi

Implementare i seguenti prototipi per le liste concatenate semplici.

void Deduplica_SL(**Lista_SL** ***lista**);
Rimuove tutti i valori duplicati nella lista

void CancellaPrimo_SL(**Lista_SL** ***lista**, **int** dato);
Rimuove la prima occorrenza di dato nella lista.

void CancellaUltimo_SL(**Lista_SL** ***lista**, **int** dato);
Rimuove l'ultima occorrenza di dato nella lista.

void CancellaTutti_SL(**Lista_SL** ***lista**, **int** dato);
Rimuove tutte le occorrenze di dato nella lista.