

Puntatori e strutture dati dinamiche: allocazione della memoria e modularità in linguaggio C

Capitolo 4: Le liste

G. Cabodi, P. Camurati, P. Pasini, D. Patti, D. Vendraminetto





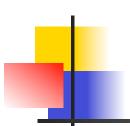
Sequenze lineari

 Insieme finito di elementi di tipo generico Item disposti consecutivamente in cui a ogni elemento è associato univocamente un indice

$$a_0, a_1, \ldots, a_i, \ldots, a_{n-1}$$

Sulle coppie di elementi è definita una relazione predecessore/successore:

$$a_{i+1} = succ(a_i)$$
 $a_i = pred(a_{i+1})$
 $\exists succ(a_{n-1})$ $\exists pred(a_0)$



- La sequenza è:
 - ordinata
 - non ordinata
- L'accesso avviene:
 - in base ad una chiave
 - in base alla posizione nella sequenza
 - diretto, costo O(1)
 - sequenziale, costo O(n)



Modalità di memorizzazione:

dati contigui in memoria

Accesso diretto:

- dato l'indice i, si accede all'elemento a_i senza dover scorrere la sequenza lineare
- il costo dell'accesso non dipende dalla posizione dell'elemento nella sequenza lineare, quindi è O(1)



Modalità di memorizzazione:

dati non contigui in memoria

Accesso sequenziale:

- dato l'indice i, si accede all'elemento a_i scorrendo la sequenza lineare a partire da uno dei suoi 2 estremi, solitamente quello di SX
- il costo dell'accesso dipende dalla posizione dell'elemento nella sequenza lineare, quindi è O(n) nel caso peggiore

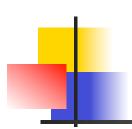


Operazioni sulle liste

- ricerca di un elemento il cui campo chiave di ricerca è uguale a una chiave data
- inserzione di un elemento:
 - in testa alla lista non ordinata
 - in coda alla lista non ordinata
 - nella posizione tale da garantire l'invarianza della proprietà di ordinamento per una lista ordinata

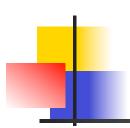


- cancellazione di un elemento:
 - che si trova in testa alla lista non ordinata
 - che si trova in una posizione arbitraria della lista non ordinata
 - che ha un campo con contenuto uguale a quello di una chiave di cancellazione (tale operazione richiede solitamente una ricerca preventiva dell'elemento da cancellare)
 - con o senza restituzione dell'elemento cancellato (estrazione).



Le liste possono essere realizzate mediante vettori:

- se è noto o stimabile il numero massimo di elementi
- sfruttando la contiguità fisica degli elementi (elemento all'indice i successore di quello all'indice i-1 e predecessore di quello all'incice i+1)
- disaccoppiando contiguità fisica e relazione predecssore/successore mediante indici



Le liste possono essere realizzate mediante strutture ricorsive:

- se non è noto o stimabile il numero massimo di elementi
- se la relazione è da predecessore a successore si hanno liste concatenate semplici
- se è in entrambi i versi si hanno liste concatenate doppie.

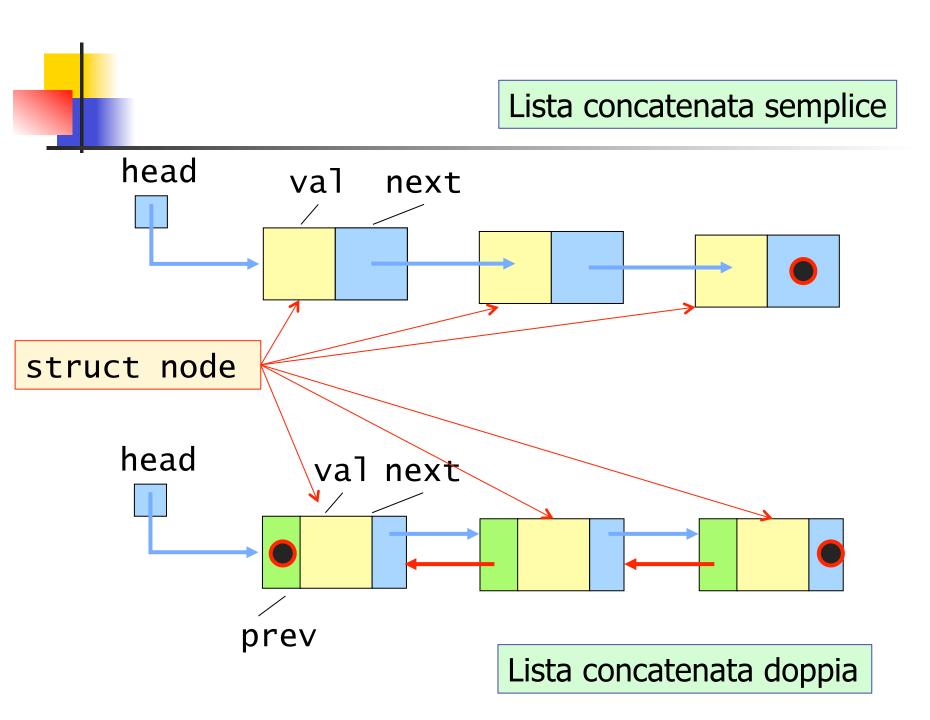


Le liste concatenate

Strutture dati dinamiche come sequenze di nodi.

In C ogni nodo è una struct con:

- un numero arbitrario di dati, generalmente racchiusi in un campo val di tipo Item
- uno o due riferimenti ("link") che puntano al nodo successivo e/o precedente





Allocazione dei nodi

Definizione dei dati:

dato in lista

```
typedef ... Item;
typedef ... Key;
```

typedef int Item;

```
chiave del dato
```

```
typedef struct {
    char nome[40];
    data_t nascita;
} Item;
```

```
typedef int Key;
```

```
typedef char *Key;
```



Definizione delle funzioni:

```
Key KEYget(Item d);
int KEYeq(Key k1, Key k2);
int KEYless(Key k1, Key k2);
int KEYgreater(Key k1, Key k2);
```

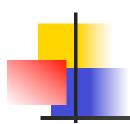


Allocazione dei nodi

Diversi modi per definire il nodo:

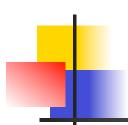
1. senza typedef, definendo il puntatore next mentre si definisce il tipo struct node

```
struct node {
   Item val;
   struct node *next;
};
```



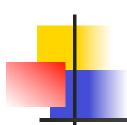
 con typedef, definendo sia un alias node_t per struct node, sia un alias link per il puntatore a oggetto di tipo struct node

```
typedef struct node {
   Item val;
   struct node *next;
} node_t, *link;
```



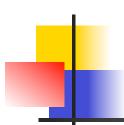
3. con separazione tra typedef e dichiarazione della struct node, dichiarando un alias link per il puntatore a oggetto di tipo struct node. Nella dichiarazione di tipo struct node si usa il tipo link appena definito

```
typedef struct node *link;
struct node {
   Item val;
   link next;
};
```



4. con separazione tra typedef e dichiarazione della struct node, dichiarando un alias node_t per struct node. Nella dichiarazione di struct node si dichiara next come puntatore a oggetto di tipo node_t

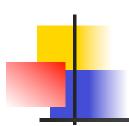
```
typedef struct node node_t;
struct node {
   Item val;
   node_t next;
};
```



5. con separazione tra typedef e dichiarazione della struct node, dichiarando un alias link per un puntatore a struct node e un alias node_t per struct node. Nella dichiarazione di struct node si usa link

```
typedef struct node *link, node_t;

struct node {
   Item val;
   link next;
};
```



Con la terza modalità:

si dichiara un puntatore x a un nodo come:

```
link x;
```

si alloca il nodo come:

```
x = malloc(sizeof *x);
```

 Ω

```
x = malloc(sizeof(struct node));
```



Operazioni atomiche su liste

Creazione mediante generazione del puntatore alla testa:

head

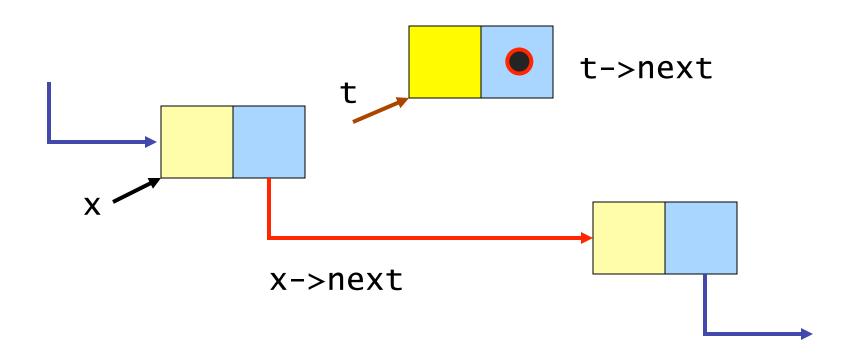
```
link head = NULL;
```

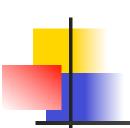


Test se la lista è vuota

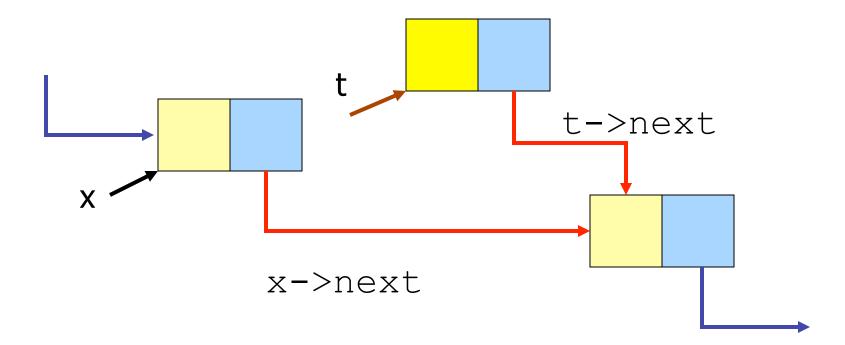
Inserimento

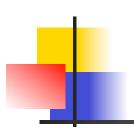
Inserimento del nodo puntato da t dopo il nodo puntato da x in lista esistente:





t->next = x->next;

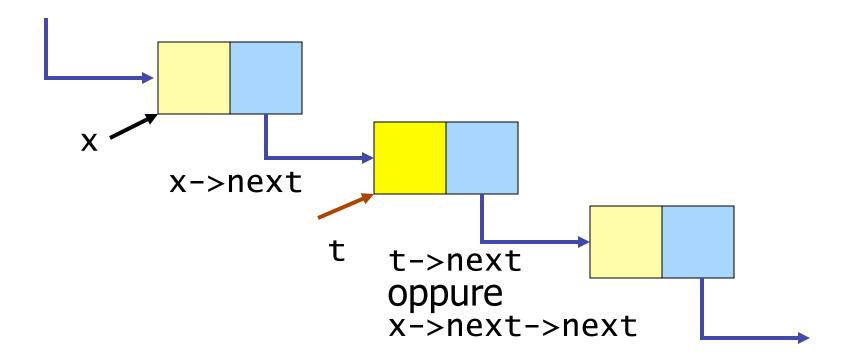


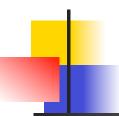




Cancellazione

Cancellazione del nodo puntato da t, successore del nodo puntato da x:





```
x->next = x->next->next;
oppure
x->next = t->next;
                 x->next
            x->next->next;
            oppure
            t->next;
```

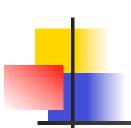


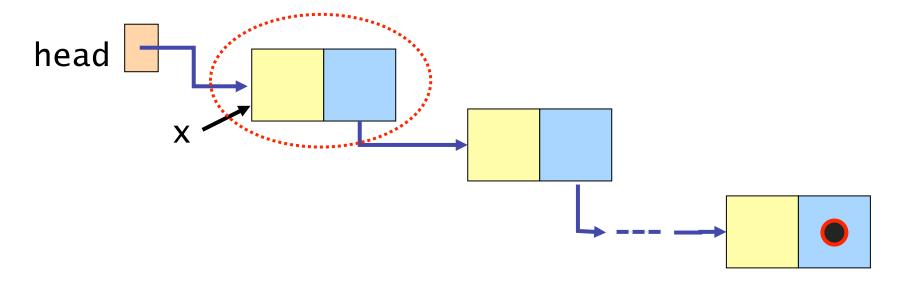
Attraversamento (I)

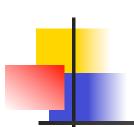
Con elaborazioni semplici basta il puntatore al nodo corrente x:

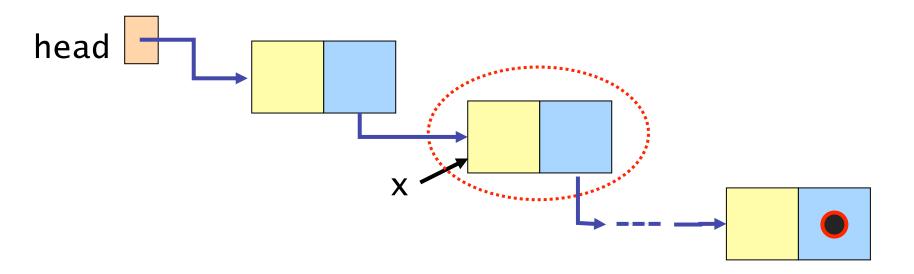
non si modifica la lista: es. ricerca, visualizzazione, conteggio, ...

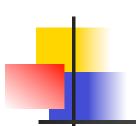
```
link x, head;
for (x=head; x!=NULL; x=x->next) {...}
```

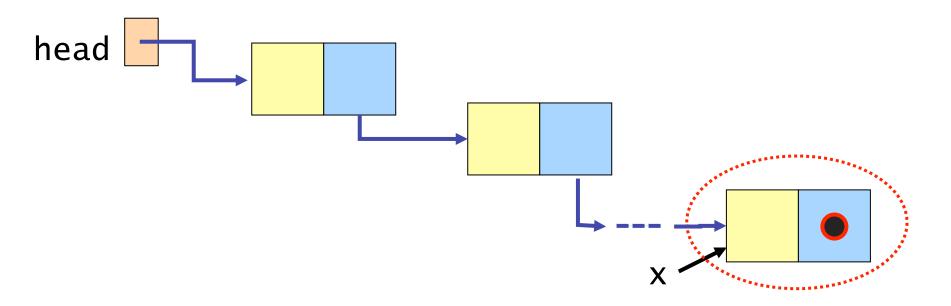














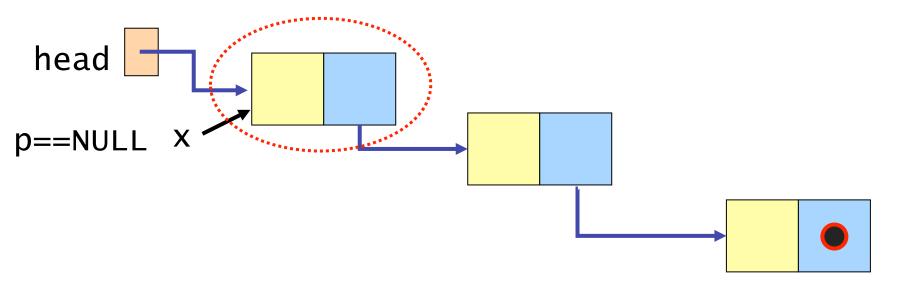
Attraversamento (II)

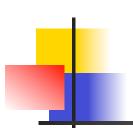
Con elaborazioni complesse serve il puntatore al nodo corrente x e al o predecessore p:

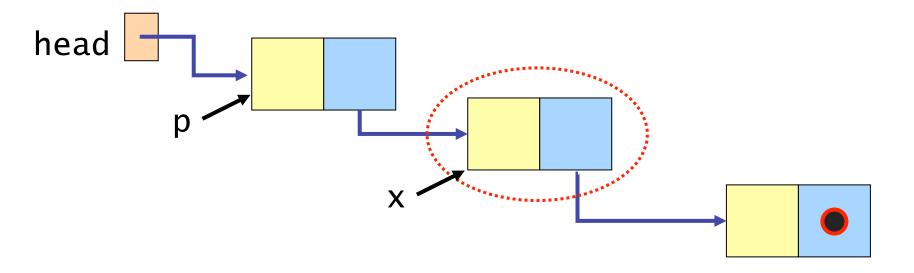
si modifica la lista: es. inserzione, cancellazione ...

```
link x, p, head;
p = NULL;
for (x=head; x!=NULL; p = x, x=x->next) {...}
```

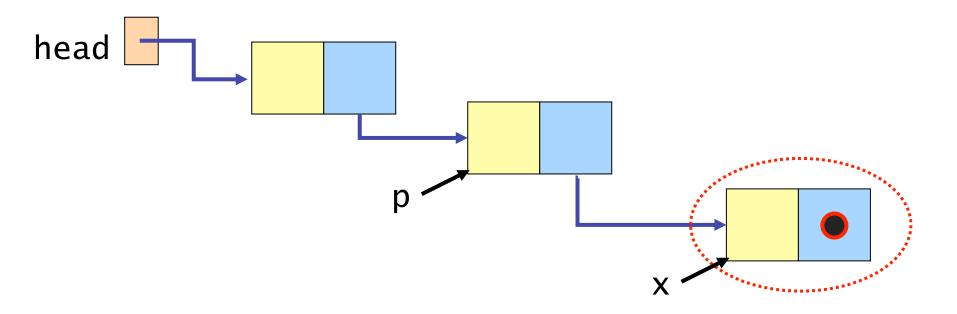


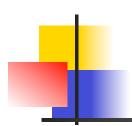








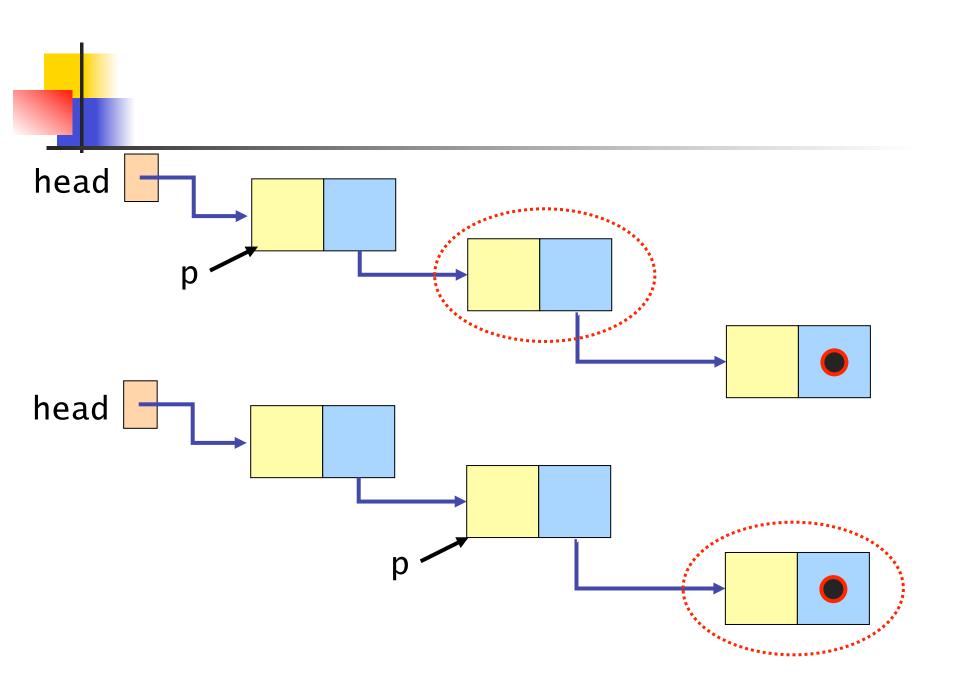




Il puntatore al nodo corrente x è inutile se:

- si tratta il nodo in testa fuori dall'iterazione
- si comincia l'iterazione dal secondo elemento, inizializzando p con head
- per terminare si testa che la lista non sia vuota e che il prossimo elemento non sia l'ultimo
- si scorre la lista aggiornando p

```
link p, head;
/* gestione separata nodo in testa */
for (p=head;p!=NULL && p->next!=NULL;p=p->next)
{...}
```

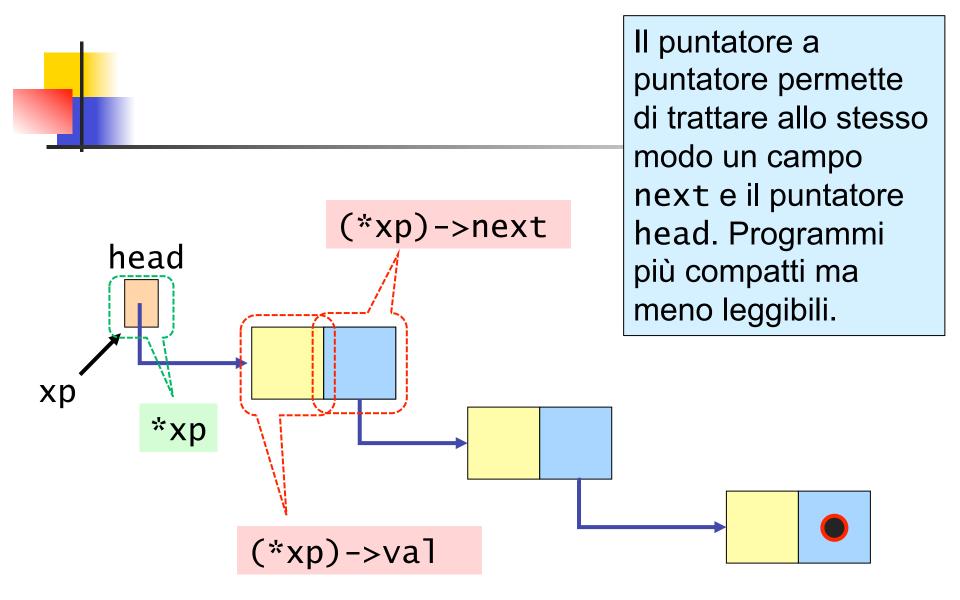




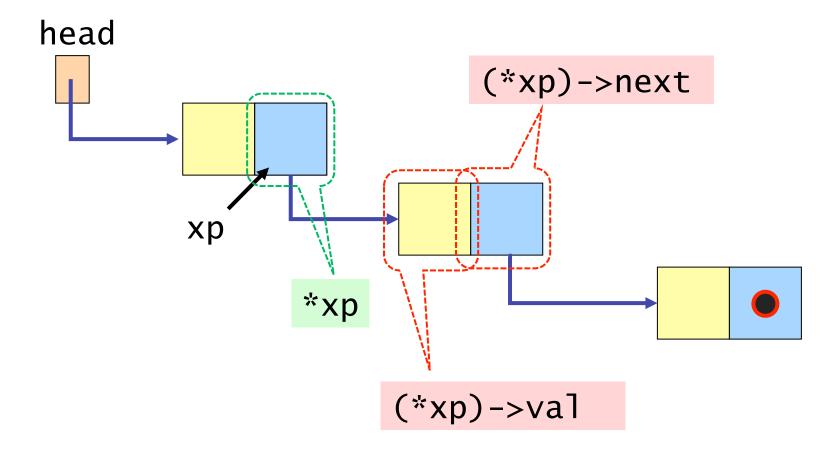
Attraversamento (III)

Con puntatore a puntatore a nodo xp per accedere al campo puntatore a successore della struct:

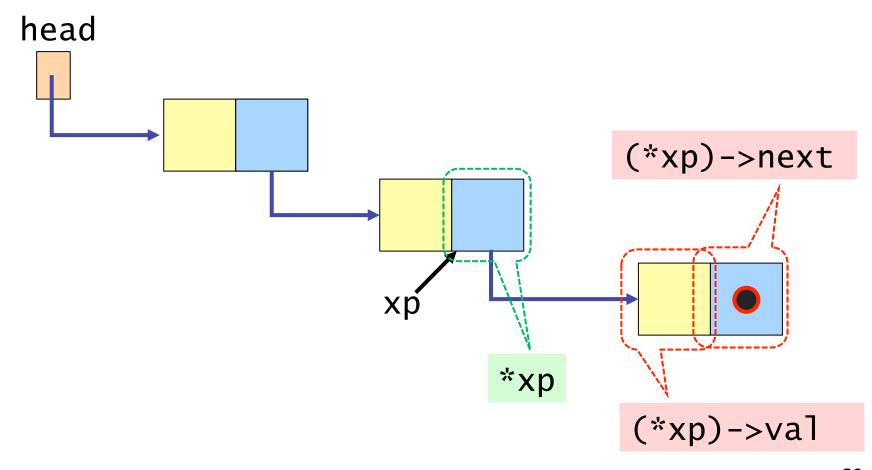
```
link *xp, head;
for (xp=&head; *xp!=NULL; xp=&((*xp)->next)) {
    ...
}
```













Attraversamento (IV)

Ricorsivo: è immediato anche l'attraversamento all'indietro senza bisogno di puntatore al predecessore:

```
void listTravR(link h) {
  if (h == NULL) return;
  ITEMdisplay(h->val);
  listTravR(h->next);
}
```

```
void listRevTravR(link h) {
  if (h == NULL) return;
  listRevTravR(h->next);
  ITEMdisplay(h->val);
}
```



Operazioni sulle liste

- Creazione nodo
- Liste non ordinate:
 - inserimento in testa
 - inserimento in coda
 - ricerca di chiave
 - cancellazione dalla testa
 - estrazione dalla testa
 - cancellazione di nodo con chiave data.



- Liste ordinate:
 - Inserimento
 - ricerca di chiave
 - cancellazione di nodo con chiave data



Creazione nodo

```
link newNode(Item val, link next) {
  link x = malloc(sizeof *x);
  if (x==NULL)
    return NULL;
  else {
    x->val = val;
    x->next = next;
  return x;
```



Inserzione in testa

Soluzione 1: parametri di ingresso: val e puntatore alla testa h. Valore di ritorno: nuovo puntatore alla testa, che il main assegnerà a head:

```
link listInsHead (link h, Item val) {
  h = newNode(val,h);
  return h:
/* main */
link head = NULL;
Item d;
head = listInsHead(head, d);
```



Inserzione in testa

Soluzione 2: parametri di ingresso: val e puntatore al puntatore alla testa hp. La funzione modifica direttamente il puntatore alla testa *hp. In chiamata si passa l'indirizzo del puntatore alla testa &head:

```
void listInsHeadP(link *hp, Item val) {
   *hp = newNode(val, *hp);
}
/* main */
...
link head = NULL;
Item d;
...
listInsHead(&head, d);
```



Serve il puntatore all'ultimo nodo:

- ricavato con un attraversamento di costo O(n)
- mantenuto con costo O(1)

Soluzione 1:

- lista vuota: inserzione in testa con modifica di head
- lista non coda vuota: ciclo di attraversamento per raggiungere l'ultimo nodo, creazione di un nuovo nodo e aggancio come successore dell'ultimo nodo, head rimane invariato.





```
link listInsTail(lin' n, Item val) {
  link x;
  if (h==NULL)
    return newNode(val, NULL);
  for (x=h; x->next!=NULL; x=x->next);
  x->next = newNode(val, NULL);
  return h;
                          crea nuovo nodo e
/* main */
                          aggancialo
                          all'ultimo
link head=NULL;
Item d;
head = listInsTail(head, d);
```



Soluzione 2:

parametri di ingresso: val e puntatore al puntatore alla testa hp

- x = *hp identfica la testa della lista
- lista vuota: inserzione in testa con modifica di *hp
- lista non coda vuota: ciclo di attraversamento per raggiungere l'ultimo nodo, creazione di un nuovo nodo e aggancio come successore dell'ultimo nodo, *hp rimane invariato.



```
void listInsTailP(link
                         np, Item val) {
  link x=*hp;
  if (x==NULL)
    *hp = newNode (val, NULL);
  else {
    for (; x->next!=NULL; x=x->next);
    x->next = newNode(val, NULL);
                          crea nuovo nodo e
 *main */
                          aggancialo
                          all'ultimo
link head=NULL;
Item d;
listInsTailP(&head, d);
```



Soluzione 3:

- attraversamento con variabile xp di tipo puntatore a puntatore a nodo che punta al campo puntatore a successore della struct
- unificazione dei casi di inserimento in lista vuota e non vuota.



cerca ultimo nodo (comprende il caso di lista vuota

```
void listInsTailP(link np, Item val) {
  link *xp = hp;
  while (*xp != NULL)
    xp = \&((*xp)->next);
    *xp = newNode(val, NULL);
/* main */
link head=NULL;
Item d;
listInsTailP(&head, d);
```



Soluzione 4:

- uso di 2 variabili di tipo puntatore a puntatore a nodo hp e tp per accedere a primo e ultimo nodo
- *hp identifica la testa della lista, *tp la coda
- lista vuota: inserzione in testa con modifica di *hp e di *tp
- lista non coda vuota: creazione di un nuovo nodo e aggancio come successore dell'ultimo nodo, *hp rimane invariato, *tp viene aggiornato.

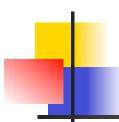


```
void listInsTFast(link *hp,link *tp,Item val)
  if (*hp==NULL)
    *hp = *tp = newNode(val, NULL);
  else {
    (*tp)->next = newNode(val, NULL);
    *tp = (*tp)->next;
  main */
link head=NULL, tail=NULL;
Item d;
listInsTailFast(&head, &tail, d);
```



Ricerca di una chiave

- Non essendo modificata la lista, basta un solo puntatore per l'attraversamento.
- Se la chiave c'è, si ritorna il dato che la contiene
- Se la chiave non c'è, si ritorna il dato nullo tramite chiamata alla funzione ITEMsetvoid.

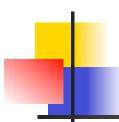


```
Item listSearch(link h, Key k) {
  link x;
  for (x=h; x!=NULL; x=x->next)
    if (KEYeq(KEYget(x->val), k))
      return x->val;
  return ITEMsetvoid();
   main */
link head=NULL;
Item d; Key k;
d = listSearch(head,k);
```



Cancellazione dalla testa

- Se la lista non è vuota, aggiorna la testa della lista con il puntatore al secondo dato che diventa il primo
- Ricorda il primo dato per poi liberarlo con free
- Il main assegna a head il nuovo puntatore alla testa.



```
link listDelHead(link h) {
 link t = h;
 if (h == NULL)
   return NULL;
 h = h->next;
  free(t);
  return h;
/* main */
link head = NULL;
head = listDelHead(head);
```



Estrazione dalla testa

- Per aggiornare la testa della lista si deve usare il puntatore al puntatore alla testa hp poiché il valore di ritorno della funzione è il dato
- Se la lista è vuota, si ritorna il dato nullo tramite chiamata alla funzione ITEMsetvoid, altrimenti si memorizza il primo dato per poi ritornarlo
- Si ricorda il primo dato per poi liberarlo con free

```
Item listExtrHeadP(link *hp) {
  link t = *hp;
  Item tmp;
  if (t == NULL)
    return ITEMsetvoid();
  tmp = t->val;
  *hp = t->next;
  free(t);
  return tmp;
/* main */
link head = NULL;
Item d;
d = listExtrHeadP(&head);
```



Cancellazione di nodo con chiave data

A seguito della cancellazione, il puntatore alla testa della lista può essere:

- NULL perché la lista era vuota
- il puntatore al secondo dato, se la chiave si trovava nel primo
- invariato se la lista non è vuota, la chiave non è il primo dato o non c'è in lista. Un ciclo di attraversamento con 2 puntatori identifica il nodo da cancellare.

```
link listDelKey(link h, Key k) {
 link x, p;
  if (h == NULL)
    return NULL;
  for (x=h, p=NULL; x!=NULL; p=x, x=x->next) {
    if (KEYeq(KEYget(x->val),k)) {
      if (x==h)
        h = x->next;
      else
        p->next = x->next;
      free(x);
      break;
  return h;
```



Cancellazione di nodo con chiave data

Versione ricorsiva:

- terminazione: si punta al nodo vuoto
- se il nodo corrente non contiene la chiave, si ricorre sulla lista che ha come testa il nodo successore
- se il nodo corrente contiene la chiave, si salva il puntatore al suo successore, si cancella il nodo corrente e si ritorna il puntatore al successore che nell'istanza ricorsiva chiamante viene assegnato come successore del nodo corrente realizzando il bypass.



```
link listDelKeyR(link x, Key k) {
 link t;
  if (x == NULL)
    return NULL;
  if (KEYeq(KEYget(x->val), k)) {
    t = x->next;
    free(x);
    return t;
  x->next = listDelKeyR(x->next, k);
  return x;
```

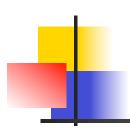


Estrazione di nodo con chiave data

L'estrazione può alterare il puntatore alla testa nel caso la chiave di ricerca sia nel primo dato.

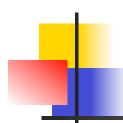
La funzione deve:

- ritornare:
 - il dato nullo tramite chiamata alla funzione ITEMsetvoid se la lista è vuota o la chiave non è presente
 - il dato se la chiave è presente
- aggiornare il puntatore alla testa della lista se si estrae il primo dato.



Si opera con la tecnica del puntatore a puntatore xp, inizializzato al puntatore al puntatore alla testa della lista hp.

Nel ciclo di attraversamento si verifica se si trova la chiave, in caso affermativo se ne salva il puntatore e il dato, si avanza nella lista ed infine si libera il nodo estratto.



```
Item listExtrKeyP(link *hp, Key k) {
  link *xp, t;
  Item i = ITEMsetvoid();
  for (xp=hp;(*xp)!=NULL;xp=&((*xp)->next)) {
    if (KEYeq(KEYget((*xp)->val),k)){
      t = *xp:
      *xp = (*xp)->next;
      i = t->val;
      free(t);
      break;
  return i;
```



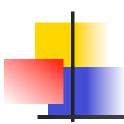
- Dati di tipo Item ordinati in base a chiave
- Inserimento (O(N)) con ricerca della posizione
- Cancellazione (O(N)) con ricerca, può decidere "non trovato" senza percorrere tutta la lista



Richiede:

- aggiornamento del puntatore alla testa per inserzione in lista vuota o inserzione di dato con chiave minima (massima)
- ricerca della posizione in cui inserire, cioè identificazione nodo predecessore con tecnica del doppio puntatore.





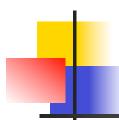
```
link x, p;
 Key k = KEYget(val);
 if (h==NULL | | KEYgreater(KEYget(h->val),k))
   return newNode(val, h);
 for (x=h->next, p=h;
      x!=NULL && KEYgreater(k, KEYget(x->val));
      p=x, x=x->next);
   p->next = newNode(val, x);
                              attraversamento
  return h;
                              per ricerca
                              posizione
```





Essendo l'accesso ai dati della lista lineare, anche se sono ordinati, non si usa la ricerca dicotomica.

La ricerca è identica a quella in lista non ordinata con eventuale interruzione anticipata.



```
Item SortListSearch(link h, Key k) {
    link x;
    for (x=h;
        x!=NULL && KEYgeq(k, KEYget(x->val));
        x=x->next)
    if (KEYeq(KEYget(x->val), k))
        return x->val;
    return ITEMsetvoid();
}
```





Cancellazione di nodo con chiave data

Si aggiunge una condizione di interruzione anticipata al ciclo di attraversamento.



```
link SortListDel(link h, Key k) {
 link x, p;
 if (h == NULL) return NULL;
 for (x=h, p=NULL;
       x!=NULL && KEYgeq(k,KEYget(x->val));
       p=x, x=x->next) {
   if (KEYeq(KEYget(x->val),k)){
      if (x==h) h = x->next;
     else
             p->next = x->next;
      free(x);
      break;
  return h;
```



Liste concatenate particolari

- Uso di nodi fittizi per semplificare i test di lista vuota
- Adiacenza logica di nodo in testa e in coda per ottenere una lista circolare
- Attraversamento in entrambe le direzioni con operazioni tipo cancellazione semplificate: liste concatenate doppie.



Liste con nodi fittizi (sentinelle)

- Nodo con dato fittizio (in testa e/o coda), usato per rimuovere casi speciali:
 - lista vuota
 - inserimento/cancellazione del primo o ultimo nodo

Lista con nodo fittizio in testa

inizializza	<pre>h = malloc(sizeof *h); h->next = NULL;</pre>
inserisci t dopo X	t->next = x->next; x->next = t;
cancella dopo X	<pre>t = x->next; x->next = t->next;</pre>
ciclo di attraversamento	<pre>for (t = h->next; t != NULL; t = t->next)</pre>
testa se lista vuota	if (h->next == NULL)

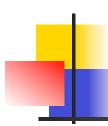
Lista con nodi fittizi in testa e coda

inizializza	<pre>h = malloc(sizeof *h); z = malloc(sizeof *z);</pre>
	h->next = z;
	z->next = z;
inserisci t dopo x	t->next = x->next;
	x->next = t;
cancella dopo X	x->next = x->next->next;
ciclo di attraversamento	<pre>for (t = h->next;</pre>
	t != z; t = t->next)
testa se lista vuota	<pre>if (h->next == z)</pre>

Lista circolare

- L'ultimo nodo punta al primo
- Utilizzata per gestire casi di servizi a "rotazione"

prima inserzione	h->next = h;
inserisci t dopo X	t->next = x->next; x->next = t;
cancella dopo X	x->next = x->next->next;
ciclo di attraversamento	<pre>t = h; do { t = t->next; } while (t != h)</pre>
testa singolo elemento	<pre>if (h->next == h)</pre>

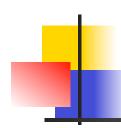


Lista concatenata doppia

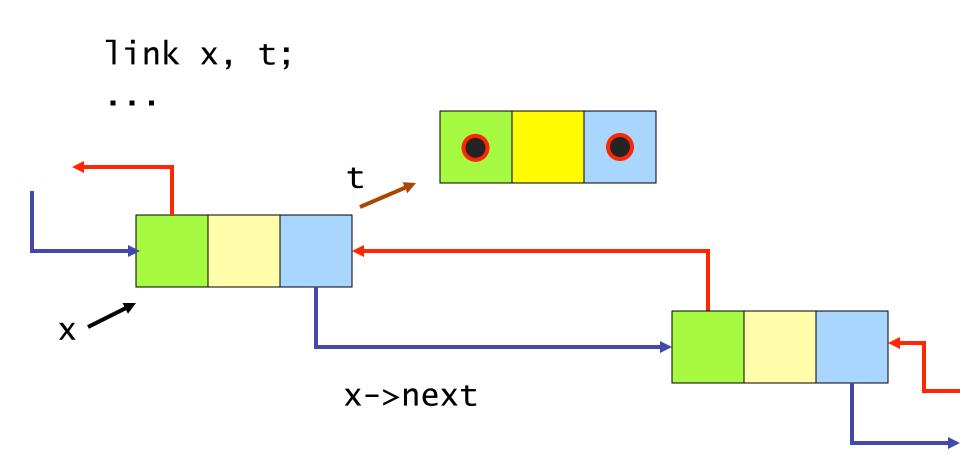
- Un puntatore in più (al nodo precedente)
- Facilita cancellazione (senza ricerca) dato il puntatore al nodo da cancellare

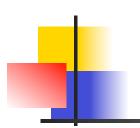
```
typedef struct node *link, node_t;

struct node {
   Item val;
   link next;
   link prev;
};
```

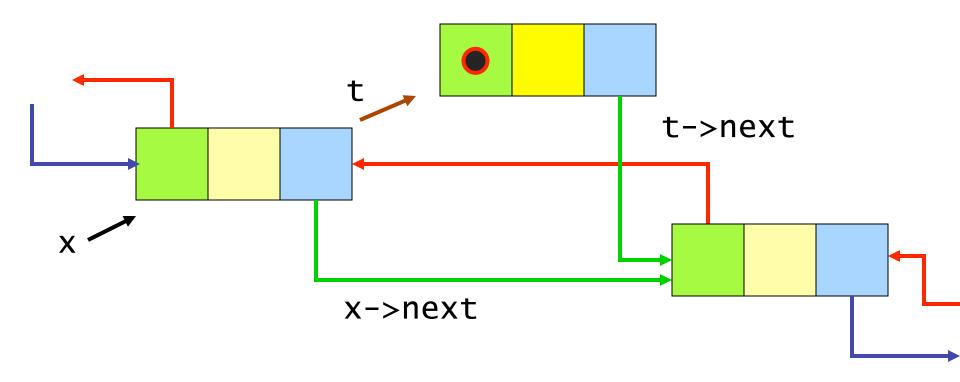


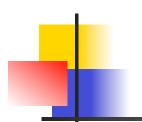
Inserimento di nodo t dopo nodo x



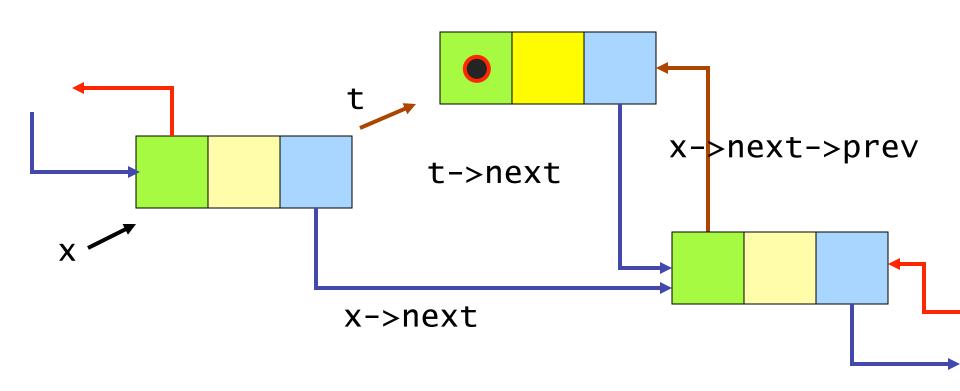


t->next = x->next;



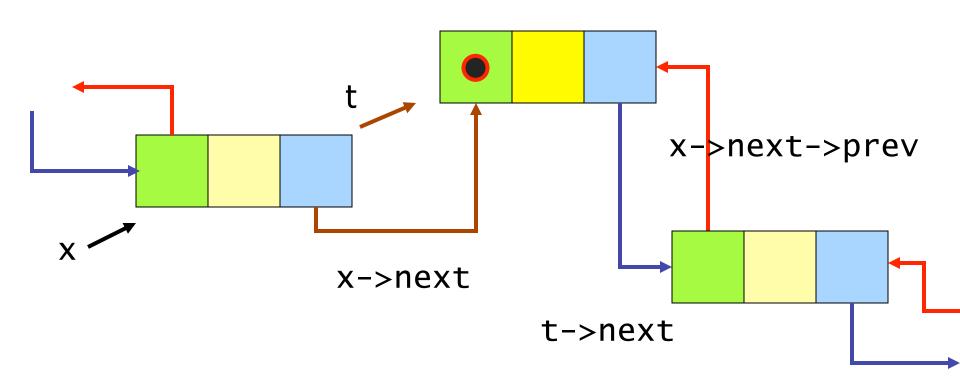


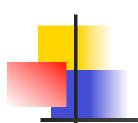
$$x->next->prev = t;$$



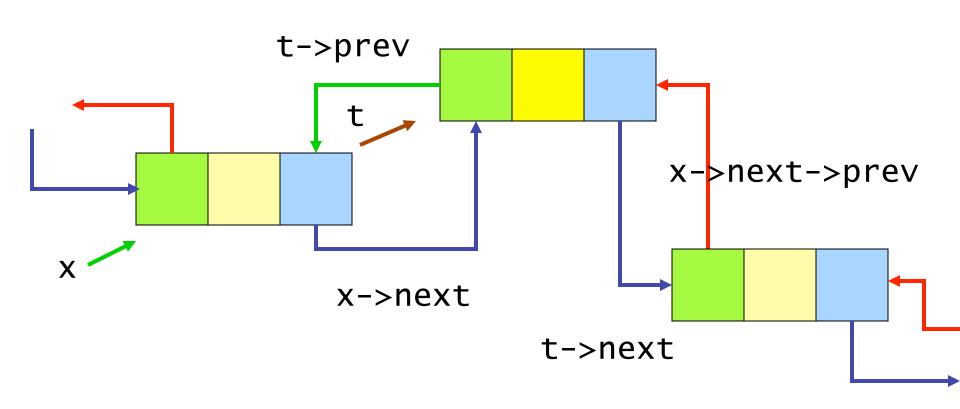


$$x->next = t;$$





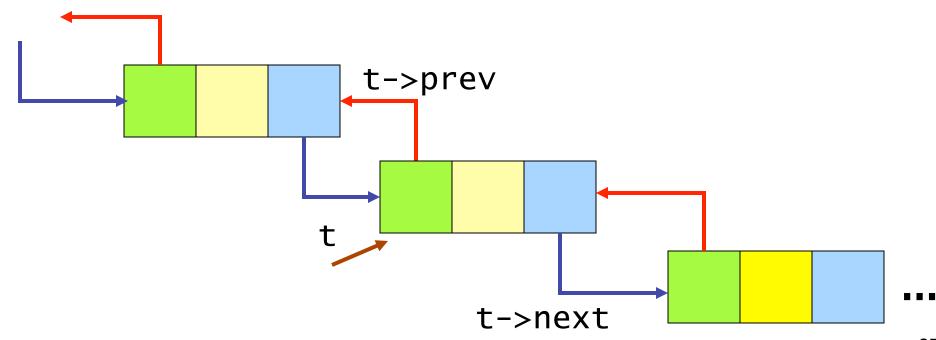
$$t->prev = x;$$

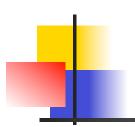


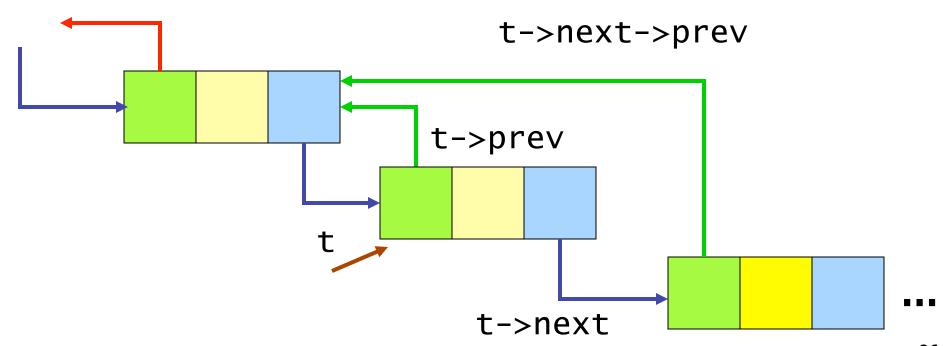


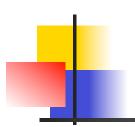
Cancellazione del nodo t

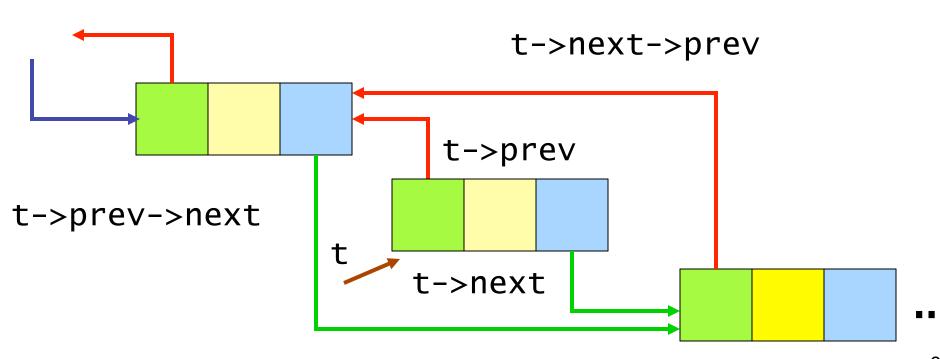
```
link t;
```

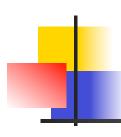












Problemi semplici su liste

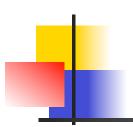
- Inversione di lista
- Insertion sort su lista
- Elenchi di canzoni



Inversione di lista

Data una lista, invertirla:

- x: puntatore alla testa della lista
- r: puntatore alla testa della lista già invertita (ultimo nodo già sistemato). Inizialmente r=NULL
- y: puntatore alla porzione di lista da invertire (primo nodo ancora da sistemare). Inizialmente y=x
- t: puntatore al nodo successivo al primo nodo ancora da sistemare (puntato da y)



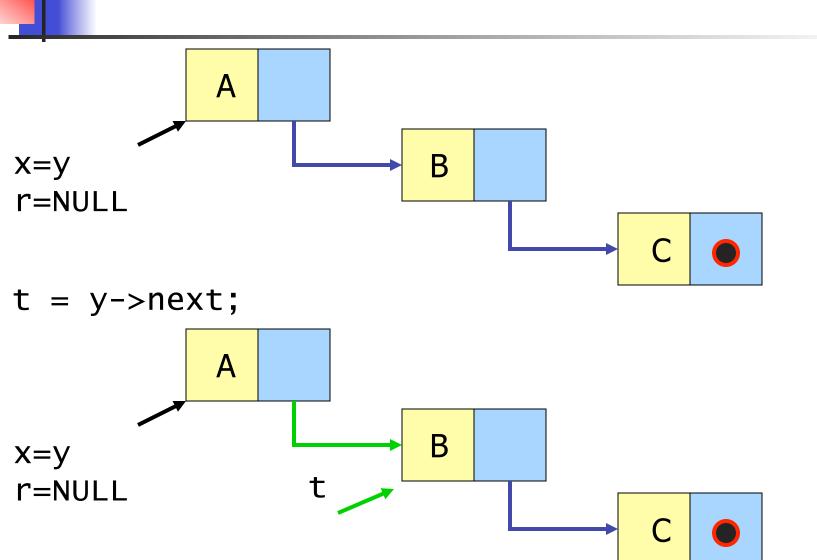
Finché esiste una porzione non vuota di lista da invertire (y!=NULL):

- inserire il nodo puntato da y in testa alla lista puntata da r
- aggiornare la testa della lista invertita r con
- aggiornare y con il suo successore t

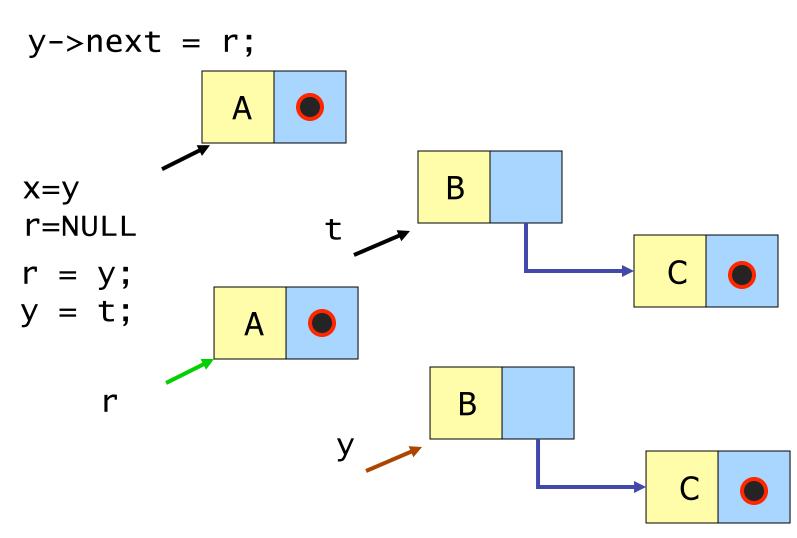


```
link listReverse(link x) {
  link t, y = x, r = NULL;
  while (y != NULL) {
    t = y->next;
    y->next = r;
    r = y;
    y = t;
  return r;
```

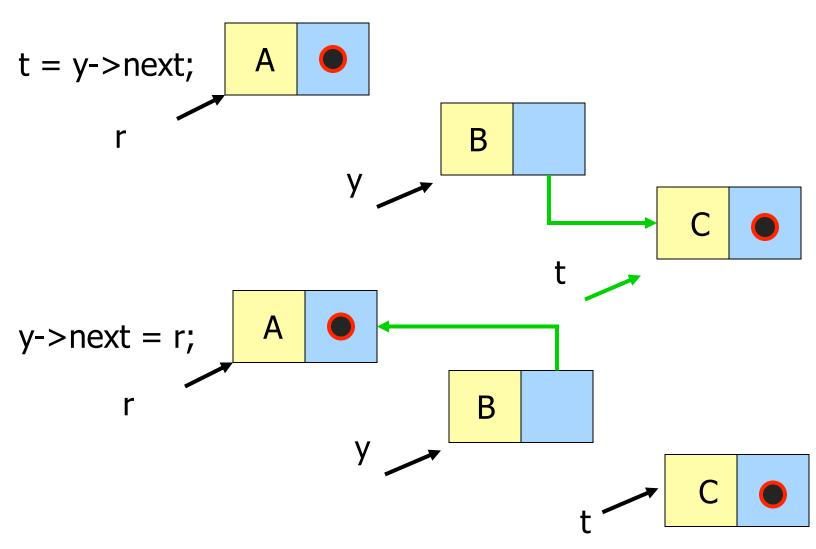




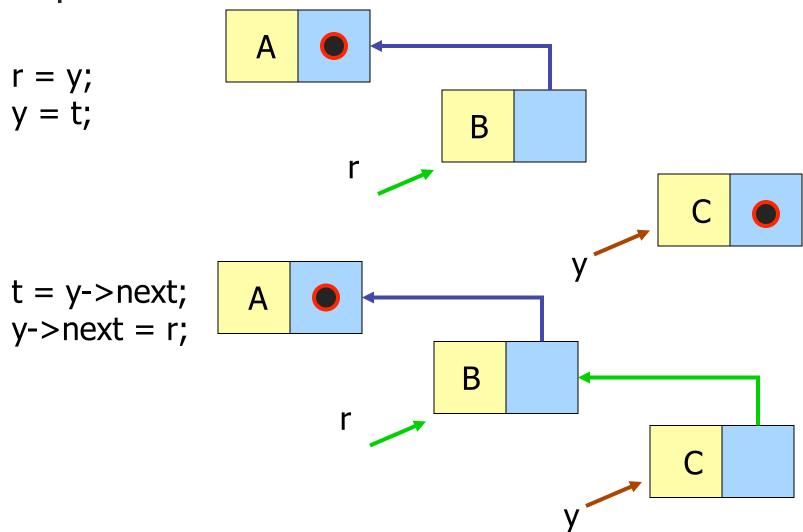




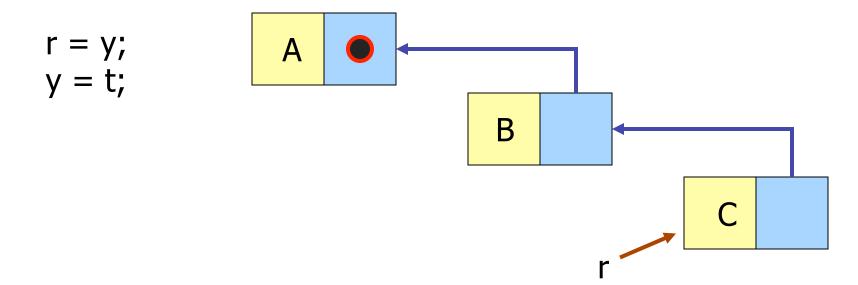












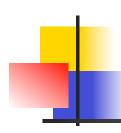


Versione con funzioni su liste:

finché esiste una porzione non vuota di lista da invertire (y!=NULL):

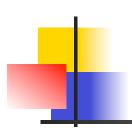
- estrai nodo da testa della lista y
- inserisci nodo in testa alla lista invertita

```
link listReverseF(link x) {
  link y = x, r = NULL;
  Item tmp;
  while (y != NULL) {
    tmp = listExtrHeadP(&y);
    r = listInsHead(r, tmp);
  }
  return r;
}
```



Insertion sort su lista

- Solo ordinamenti quadratici, essendo impossibile l'accesso diretto tipico dei vettori
- Lista contenente N numeri casuali con h puntatore alla testa
- Insertion sort:
 - lista non ordinata (input)
 - lista in output generata mediante inserzioni in lista ordinata

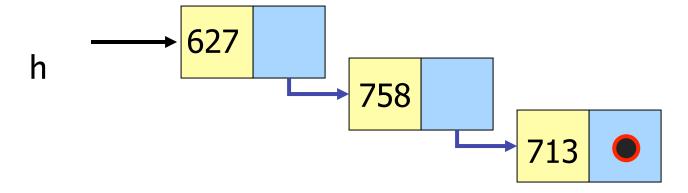


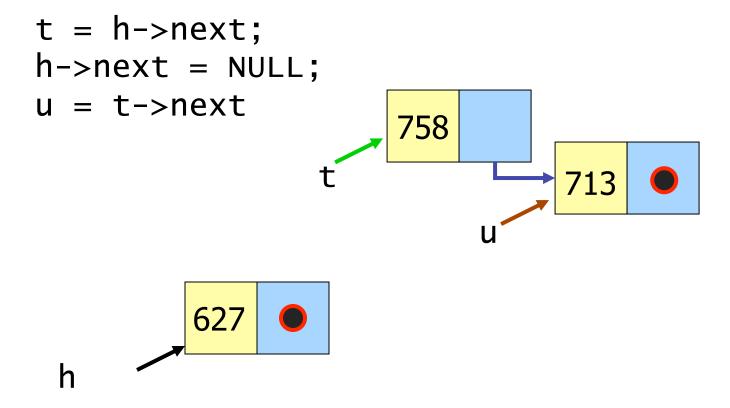
Insertion sort:

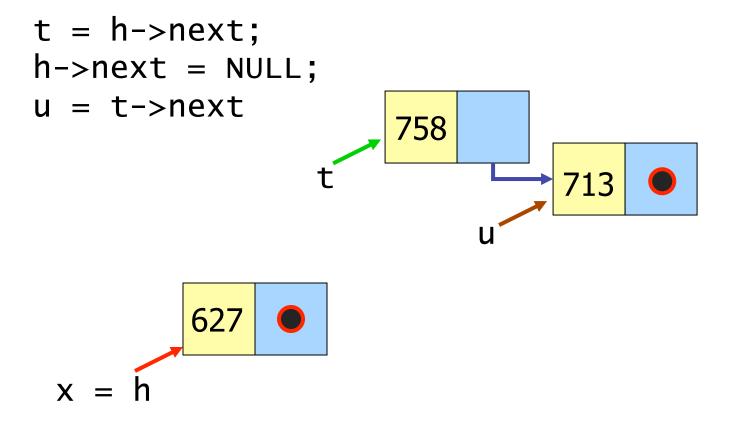
- inizialmente: nodo di testa puntato da h è primo nodo della lista ordinata (h->head=NULL;)
- attraversamento della lista restante a partire dal secondo nodo puntato da t (t=h->next;), mantenimento del puntatore u al nodo successore e scalamento di t e u ad ogni iterazione
- nel corpo dell'iterazione si verifica se il nodo contiene la chiave minima e lo si inserisce in testa o lo si inserisce in ordine.



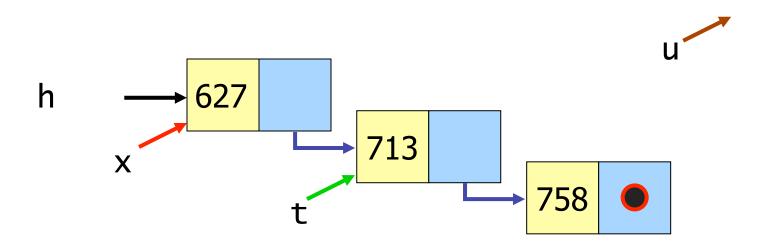
```
for(t=h->next, h->next, h->next, t=u){
  u = t->next;
  if (h->val > t->val) {
    t->next = h;
    h = t;
  else {
    for (x=h; x->next != NULL; x=x->next)
      if (x->next->val > t->val)
        break;
   t->next = x->next;
                              attraversamento
   x->next = t;
                              per ricerca
                              posizione
              inserimento
```







```
t->next = x->next;
x->next = t;
                               713
head -
        → 627
                   758
```





Versione con funzioni su liste:

finché esiste una porzione non vuota della lista originale (y!=NULL):

- estrai nodo da testa della lista y
- inserisci nodo in ordine nella lista ordinata

```
link listSortF(link h) {
  link y = h, r = NULL;
  Item tmp;
  while (y != NULL) {
    tmp = listExtrHeadP(&y);
    r = SortListIns(r, tmp);
  }
  return r;
}
```

Liste con indici

- Se i dati sono contenuti in vettori, essi sono individuabili mediante i loro indici
- Gli indici fungono da riferimenti, come i puntatori, ma nello spazio dei vettori e non in quello degli indirizzi di sistema
- Un indice di valore -1 gioca il ruolo di NULL

Liste con vettori

- Possibile realizzare liste con vettori
- Limiti:
 - allocazione del vettore della dimensione corretta o riallocazione
 - scarsa dinamicità in relazione alle cancellazioni.



Elenchi di canzoni: dati:

- un elenco di cantanti (stringhe), memorizzato in un vettore
- un elenco di canzoni (stringhe per i titoli, interi che indicano l'indice del cantante nel vettore dei cantanti)

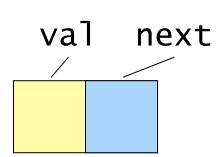
creare per ogni cantante l'elenco delle sue canzoni.



Soluzione 1: con liste concatenate

- Ogni cantante, oltre al nome, ha un puntatore alla testa della lista delle sue canzoni
- La lista delle canzoni è fatta di nodi che contengono la stringa del titolo
- Il tipo Item è un puntatore a carattere e si introduce per uniformità (typedef char *Item)

```
typedef struct nodo_s {
   Item val;
   struct nodo_s *next;
} nodo_t, *link;
```





```
titolo id_cant
```

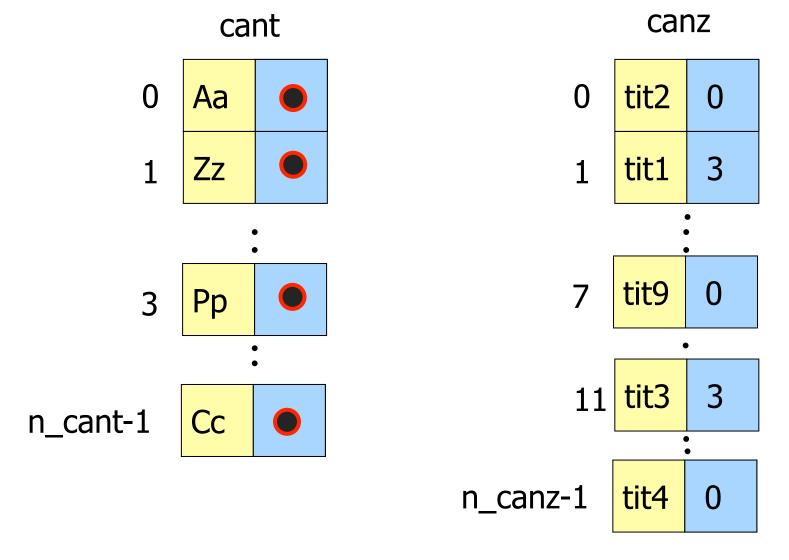
```
typedef struct canz_s{
   char *titolo;
  int id_cant;
} canz_t;
```

```
nome h
```

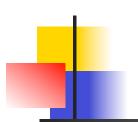
```
typedef struct cant_s{
   char *nome;
   link h;
} cant_t;
```

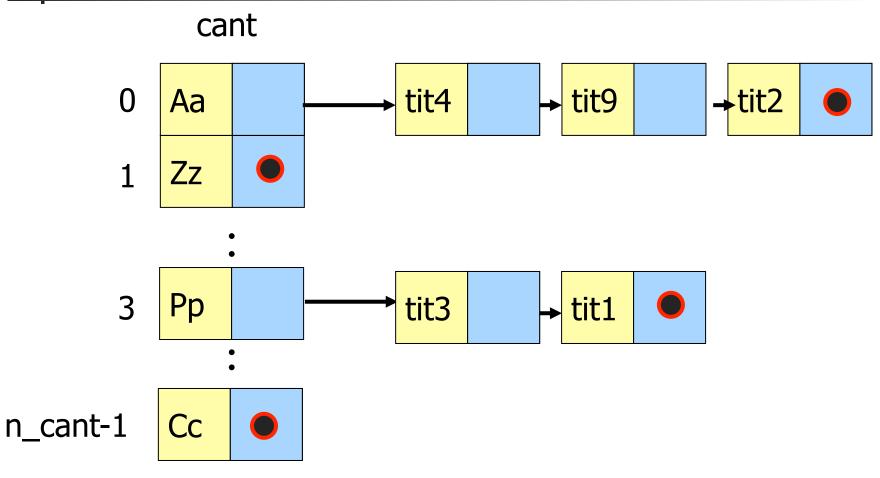
Inizialmente

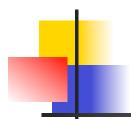












inizializzazione a NULL puntatori a teste delle liste

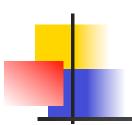
```
, int n_cant,
void genEl(cant_t *c/
                   canz_t *
 int i, id_c;
  Item v;
 for (i=0; i<n_cant; i++)
   cant[i].h = NULL;
 for (i=0; i<n_canz; i++) {
   id_c = canz[i].id_cant;
   v = strdup(canz[i].titolo);
   cant[id_c].h = newNode(v,cant[id_c].h);
```

inserimento in testa del nuovo nodo



Soluzione 2: con liste concatenate

- Ogni cantante, oltre al nome, ha un puntatore alla testa della lista delle sue canzoni
- La lista delle canzoni di un cantante è creata concatenando i nodi del vettore canz che hanno quel cantante come autore
- Non si creano ex novo nodi per la lista, non serve il tipo Item



```
titoloid_cant next
```

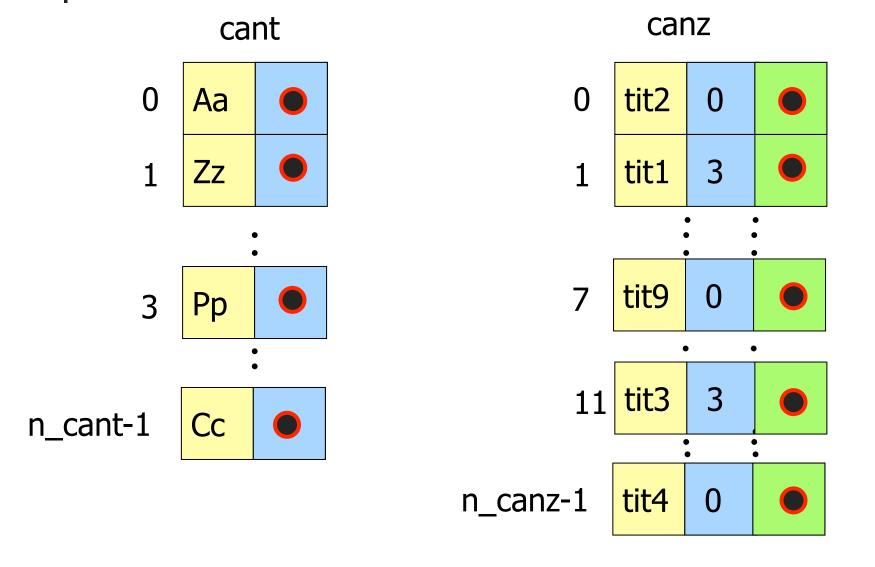
```
typedef struct canz_s{
   char *titolo;
   int id_cant;
   struct canz_s *next;
} canz_t;
```

```
nome h
```

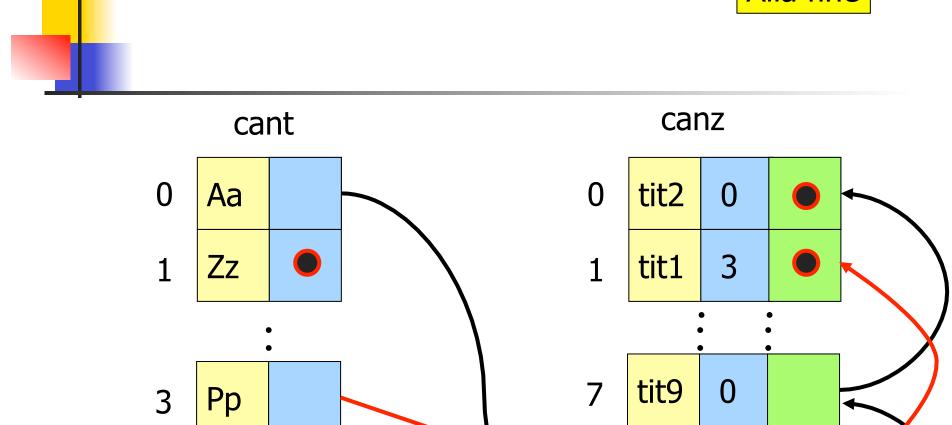
```
typedef struct cant_s{
   char *nome;
   link h;
} cant_t;
```

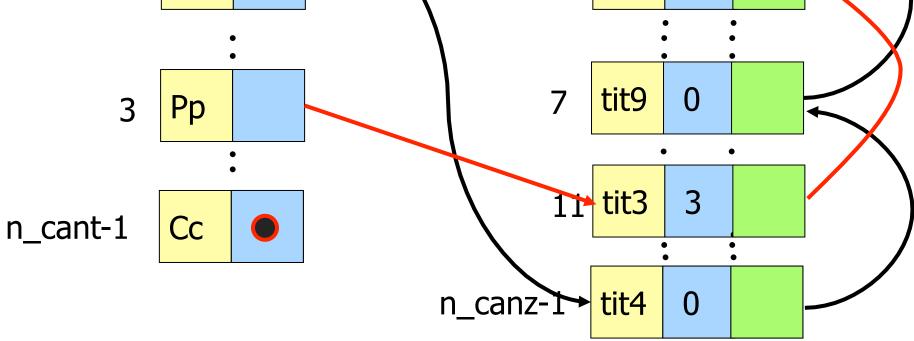














inizializzazione a NULL puntatori a teste delle liste

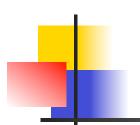
```
void genEl(cant_t *c/
                      _, int n_cant,
           canz_t * anz, int n_canz) {
  int i, id_c;
  for (i=0; i<n_cant; i++)
    cant[i].h = NULL;
  for (i=0; i<n_canz; i++) {
    id_c = canz[i].id_cant;
    canz[i].next = cant[id_c].h;
    cant[id_c].h = &canz[i];
```

inserimento in testa del nuovo nodo



Soluzione 3: con indici

- Ogni cantante, oltre al nome, ha un indice del dato in testa alla lista delle sue canzoni
- Ogni canzone, oltre a titolo e indice del cantante, contiene l'indice della canzone successiva nella lista di quel cantante
- La lista delle canzoni di un cantante è creata concatenando mediante gli indici i nodi del vettore canz che hanno quel cantante come autore
- Non si creano ex novo nodi per la lista, non serve il tipo Item



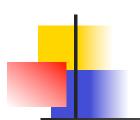
```
titoloid_cant id_next
```

```
typedef struct canz_s{
   char *titolo;
   int id_cant;
   int id_next;
} canz_t;
```

```
nome h
```

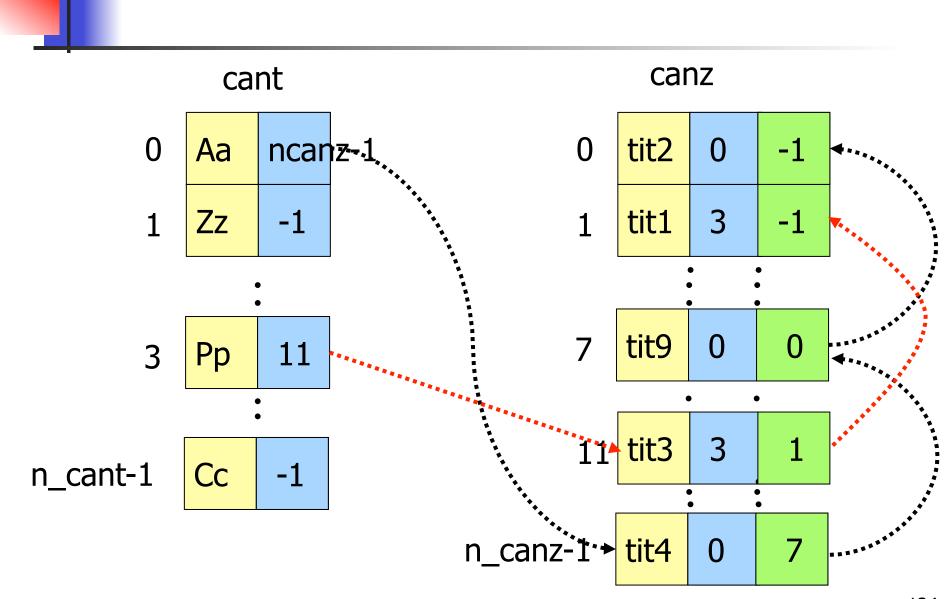
```
typedef struct cant_s{
   char *nome;
   int id_testa;
} cant_t;
```

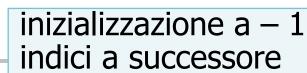
Inizialmente



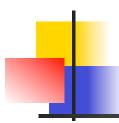
cant			canz				
0	Aa	-1	0	tit2	0	-1	
1	Zz	-1	1	tit1	3	-1	
		•				•	
3	Рр	-1	7	tit9	0	-1	
n_cant-1	Сс	-1	11	tit3	3	-1	
			n_canz-1	tit4	0	-1	







inizializzazione a – 1 indici a teste delle liste



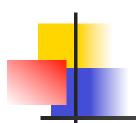
inserimento primo elemento

```
for (i=0; i<n_canz; i++) {
    if(canz[i].id_next == -1)
      id_c = canz[i].id_cant;
      if (cant[id_c].id_testa == -1)
        cant[id_c].id_testa = i;
      else {
        canz[i].id_next = cant[id_c].id_testa;
        cant[id_c].id_testa = i;
             inserimento in testa
```



Soluzione 4: con vettori

- Ogni cantante, oltre al nome, ha un vettore dinamico delle sue canzoni (puntatore) e il numero delle sue canzoni
- Ogni canzone contiene titolo e indice del cantante
- Il numero di canzoni di ogni cantante è calcolato con un'iterazione sulle canzoni
- La lista delle canzoni è realizzata tramite vettore di indici
- Non si creano ex novo nodi per la lista, non serve il tipo Item



```
titoloid_cant
```

```
typedef struct canz_s{
   char *titolo;
  int id_cant;
} canz_t;
```

```
nome lista n_canz
```

```
typedef struct cant_s{
   char *nome;
   int *lista;
   int n_canz;
} cant_t;
```



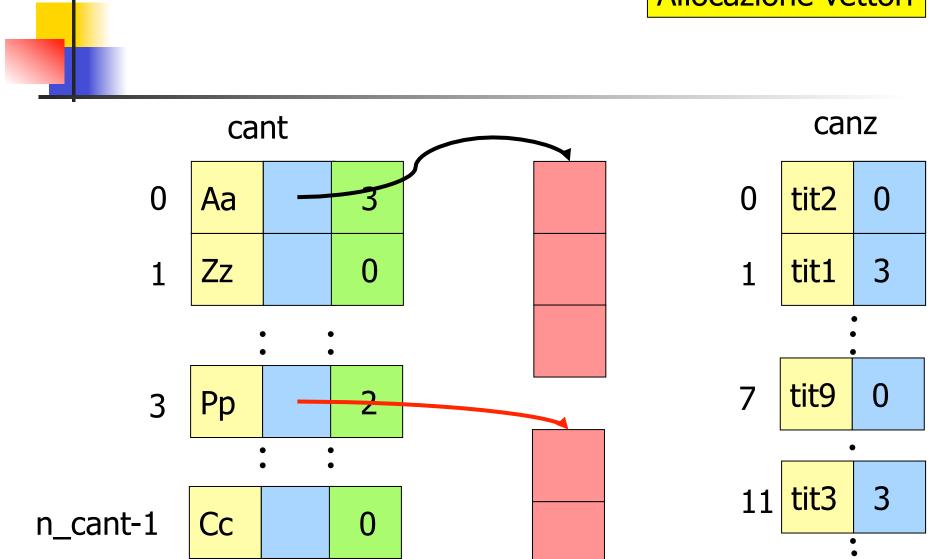


	cai	nt		canz		
0	Aa		0	0	tit2	0
1	Zz		0	1	tit1	3
))			
3	Рр		0	7	tit9	0
	•		•		•	
n_cant-1	Сс		0	11	tit3	3
				n_canz-1	tit4	0

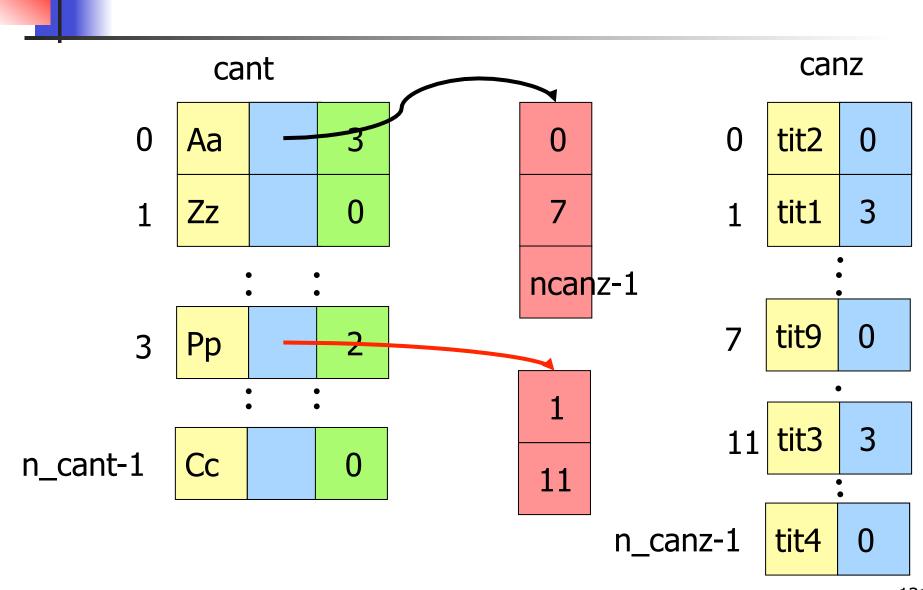
Allocazione vettori

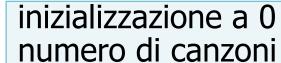
n_canz-1

tit4



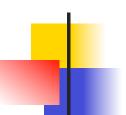








```
void genEl(cant_t *cant, i c n_cant,
            canz_t *canz, int n_canz) {
  int i, id_c;
  for (i=0; i<n_cant; i++)
                              conteggio numero di
    cant[i].n\_canz = 0;
                              canzoni di ogni cantante
  for (i=0; i<n_canz; i++)
    cant[canz[i].id_cant].n_canz++;
  for (i=0; i<n_cant; i++) {
    cant[i].lista = malloc(cant[i].n_canz*
                              sizeof(int));
    if (cant[i].lista == NULL)
      exit(-1);
    cantanti[i].n_canzoni = 0;
            reinizializzazione contatore delle
            canzoni per successivo inserimento
```



creazione lista canzoni di ogni cantante

```
for (i=0; i<n_canz; i++) {
   id_c = canz[i].id_cant;
   cant[id_c].lista[cant[id_c].n_canz] = i;
   cant[id_c].n_canz++;
}
}</pre>
```