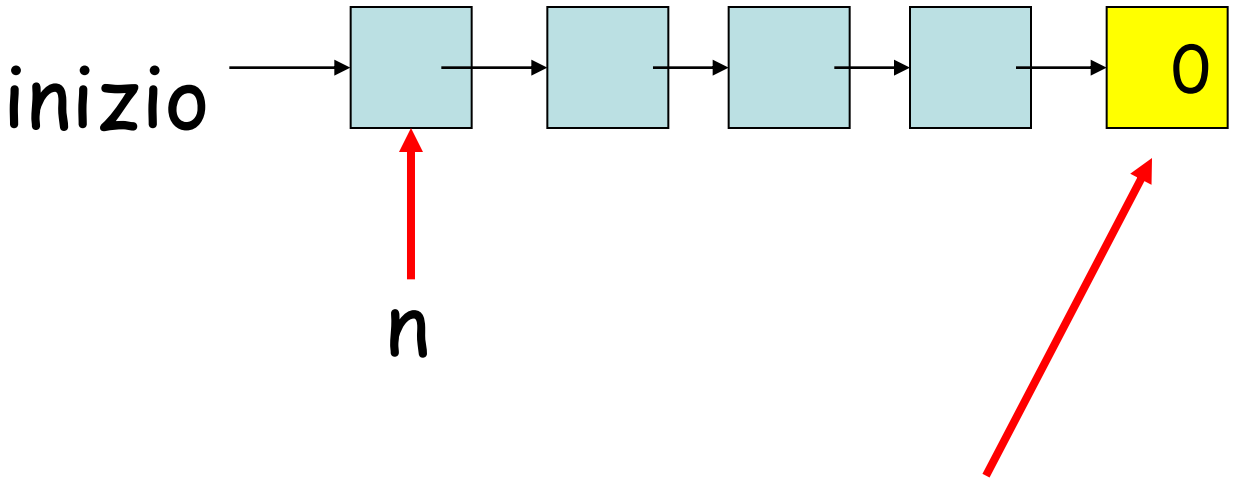
# Liste concatenate 2

# altro esempio: eliminare ultimo nodo di una lista

# soluzioni col passaggio di n per valore




n si deve fermare qui, quando n->next==0

buttare via questo nodo e restituire 0

# costruiamo la lista al ritorno

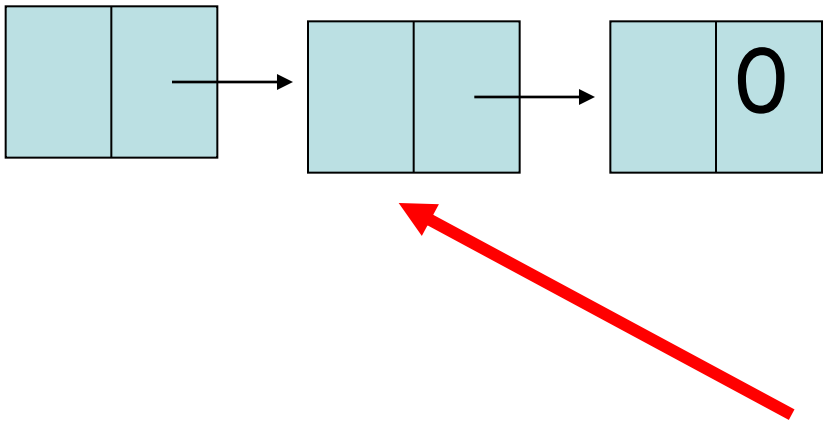
```
nodo * del_L(nodo *n)
{if( ! n→next )
{delete n; return 0; }
n→next=del_L(n→next);
return n;
}
```

```
invocazione:
if(inizio)  inizio=del_L(inizio);
```

non ha senso se la lista è vuota

è possibile fare l'operazione solo all'andata ?

a quale nodo deve fermarsi la ricorsione ?




poco generale
BRUTTA !!

dobbiamo fermarci qui

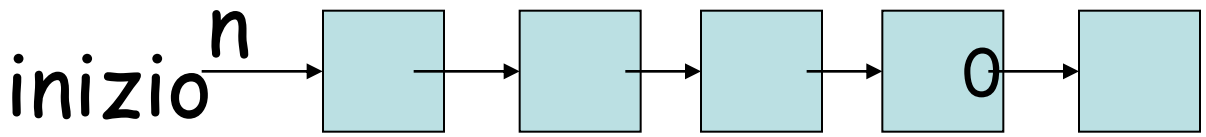
n→next→next ==0

funziona solo per liste con almeno 2 nodi !!!!!

# eliminare ultimo nodo all'andata

```
void     del_L(nodo *n)
{

if( ! n→next→next )
{  delete n→next;

    n→next=0; }

 else

  del_L(n→next);

}
```

# eliminare ultimo nodo con passaggio per riferimento




```
void del_LR(nodo *& n)
{
if(! n→next)
        {delete n; n=0; }
else
        del_LR(n →next);
}
```

il passaggio per riferimento ci permette di arrivare all'ultimo nodo e di avere un alias del next del nodo precedente

invocazione: if(inizio) del_LR(inizio);

## ricorsione e passaggio per riferimento:

```
void f(... int & x ....)
{
....f(...x...)...
}
```

tutte le invocazioni di f condividono la variabile x : le modifiche di x si ripercuotono su tutte le invocazioni

negli esercizi visti prima era diverso:

```
void ins(nodo * & n....)
{
....ins( n->next...)
}
```

# altro esercizio distruggere l'intera lista

```
void del_all(nodo *x)
{if(x)
{del_all(x→next);
delete x;
} }
```

invocazione:

```
del_all(inizio);

inizio=0;
```

l'ordine delle cose è MOLTO importante !!

percorrere le liste con un ciclo:

stampare i campi info dei nodi:

```
nodo*X=L; // non perdiamo l'inizio
while(X)
{
cout<< X->info<<' ';
X=X->next;
}
```

e all'incontrario?

```
void stampa(nodo* x)
{
  if(x)
    {
      cout<< x->info<<' ';
      stampa(x->next);
    }
}
```

è ricorsivo terminale, facile farlo con l'iterazione

```
while(X)
{
cout<< X->info<<' ';
X=X->next;
}
```

```
void stampa(nodo* x)
{
  if(x)
    {
      stampa(x->next);
      cout<< x->info<<' ';
    }
}
```

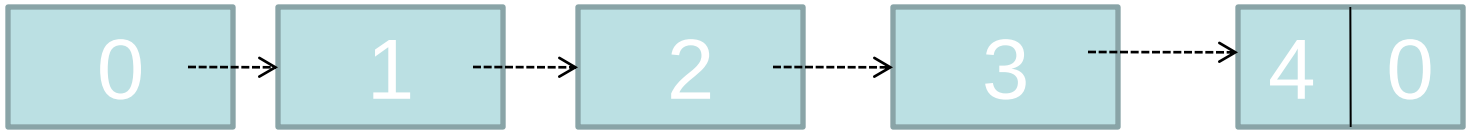
ma questa?

```
while(X)
{
X=X->next;
cout<< X->info<<' ';
}
```

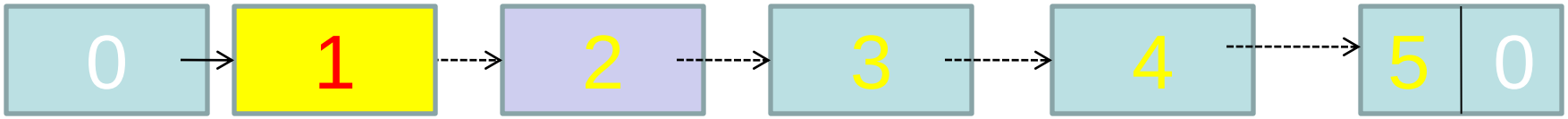
```
int l=lung(L); nodo*X=L;
nodo**K=new nodo*[l];

for(int i=0; i<l; i++)    // andata
  {K[i]=X; X=X->next;}

for(int i=l-1; i>=0; i--) //ritorno
  cout <<K[i]->info<<' ';
```

# esercizio

Inserire un nodo in posizione k=0,1,...




per esempio k=1



Su quale nodo ci si deve fermare ?

per inserire in posizione k:

-passiamo il nodo k-1 : serve andata e ritorno

- ci fermiamo sul nodo k-1: basta l'andata

conviene introdurre la seguente notazione:
L(k)= lista che consiste  dei primi k nodi di L, cioè dal nodo 0 al k-1

attenzione ➔ L(0)= lista vuota

il nodo finale di L(k) è in posizione k-1

quindi se L=L(k)@R, fermarsi alla posizione k significa alla prima di R
fermarsi alla k-1 significa all'ultima di L(k)

-passiamo il nodo k-1:

PRE=(L corretta poss. vuota e k>=0)

```
nodo* ins(nodo*L, int k, int c)
{
  if(!k)
    return new nodo(c,L);
  else
    if(L)
    {L->next=ins(L->next,k-1,c); return L;}
    else
      return 0;
}
```

POST=(se L=L(k)@R, restituisce L(k)@(nodo(c,→)::R), altrimenti L)

-fermarsi sul nodo in posizione k-1:

PRE=(L corretta e non vuota e k>0, L=vL)

```
void ins(nodo*L, int k, int c)
{if(k==1)
   L->next=new nodo(c,L->next);
 else // k>1
   if(L->next) //garantisce PRE_ric
     ins(L->next,k-1,c);
}
```

POST=(se vL=L(k)@R, costruisce lista(L)=L(k)@(nodo(c,→)::R), altrimenti vL)

Esercizio 11.6.3
Scrivere una funzione ricorsiva
nodo* alt mix(nodo* P, nodo* Q),
che riceva per valore due liste corrette (possibilmente vuote) P e Q e restituisca col return una lista composta da un nodo di P, seguito da un nodo di Q, poi
di nuovo un nodo di P e così via. Se una delle 2 liste finisce prima dell'altra,
i nodi di quella rimasta verrano appesi alla fine della lista da costruire.

caso base: una delle 2 liste è vuota, allora restituisco l'altra

caso ricorsivo:
P=a::P'
Q=b::Q'
restituisco
a::b::alt_mix(P',Q')

ALT(P,Q)

POST =(produce la lista che alterna gli elementi di P e Q e quando una delle 2 finisce si attacca l'altra)  meglio produce ALT(P,Q)

PRE=(P e Q liste corrette anche vuote)

```
nodo* alt_mix(nodo* P, nodo* Q)
{
  if(!P) return Q;
  if(!Q) return P;
  nodo* x=P->next;
  nodo* y=Q->next;
  P->next=Q;
  Q->next=alt_mix(x,y);
  return P;
}
```

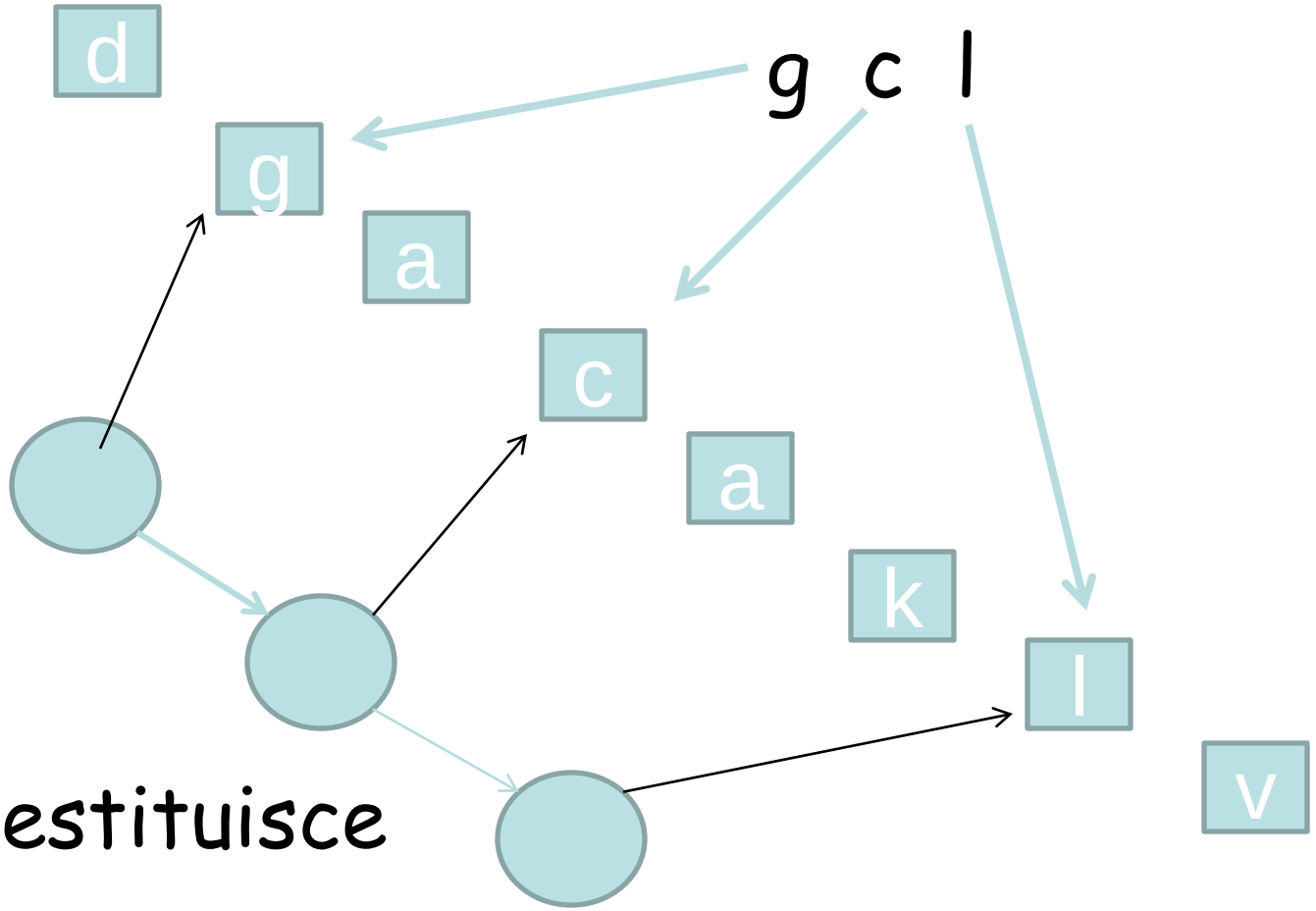
POST=(restituisce ALT(P,Q))

PRE=(P e Q corrette, P=vP, Q=vQ)

```
void alt_mix(nodo* & P, nodo* Q)
{
  if(!P) {P=Q; return;}
  if(!Q) return;
  nodo* x=P->next;
  nodo* y=Q->next;
  P->next=Q;
   alt_mix(x,y);
  Q->next= x;
 }
```

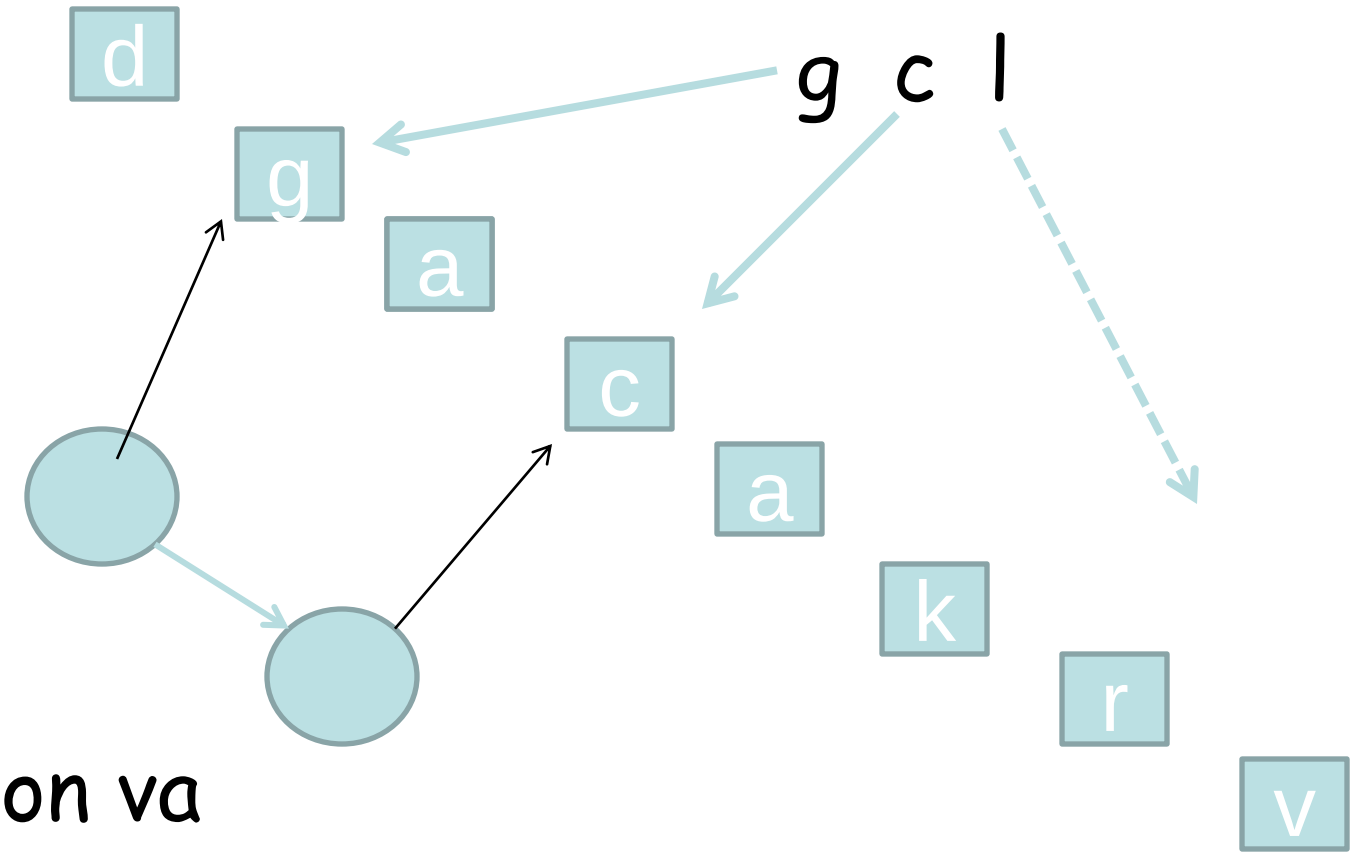
POST=(P contiene ALT(vP,vQ))

# Pattern matching completo su una lista :




Si restituisce

# Ma potrebbe anche non funzionare




non va

All'andata percorriamo la lista e facciamo i match possibili
Al ritorno dobbiamo fare due cose:

-far sapere se match fallisce o no

-se c'è match allora si deve costruire la lista risultato

Così lo costruiamo solo se c'è match

3 possibili soluzioni:

Soluzione 1) Usiamo un bool passato per riferimento (visto che deve comunicare alle invocazioni che "stanno sopra" se è stato trovato match o no)

caso base match finito

soluzione 2:

Comunichiamo il successo/insuccesso con la stessa lista

- -se ritorno 0 fallimento
- -se ritorno != 0 successo

complica il caso base

soluzione 3: sfruttando le eccezioni in caso di fallimento del match → throw

# variazioni dell'esercizio: match contigui

- calcolare numero di match contigui e completi esistenti (anche sovrapposti)

- n. match contigui e completi e non sovrapposti

- lunghezza massima dei match contigui ma possibilmente incompleti (anche sovrapposti)