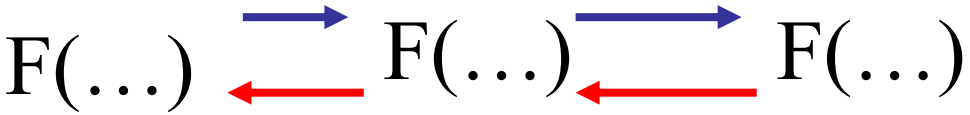
liste concatenate
e ricorsione

ricordare che nella ricorsione c'è:

l'andata



e il ritorno

si può fare calcoli sia all'andata che al ritorno

se non si fa nulla al ritorno allora ricorsione terminale ➔ facile trasformarla in WHILE

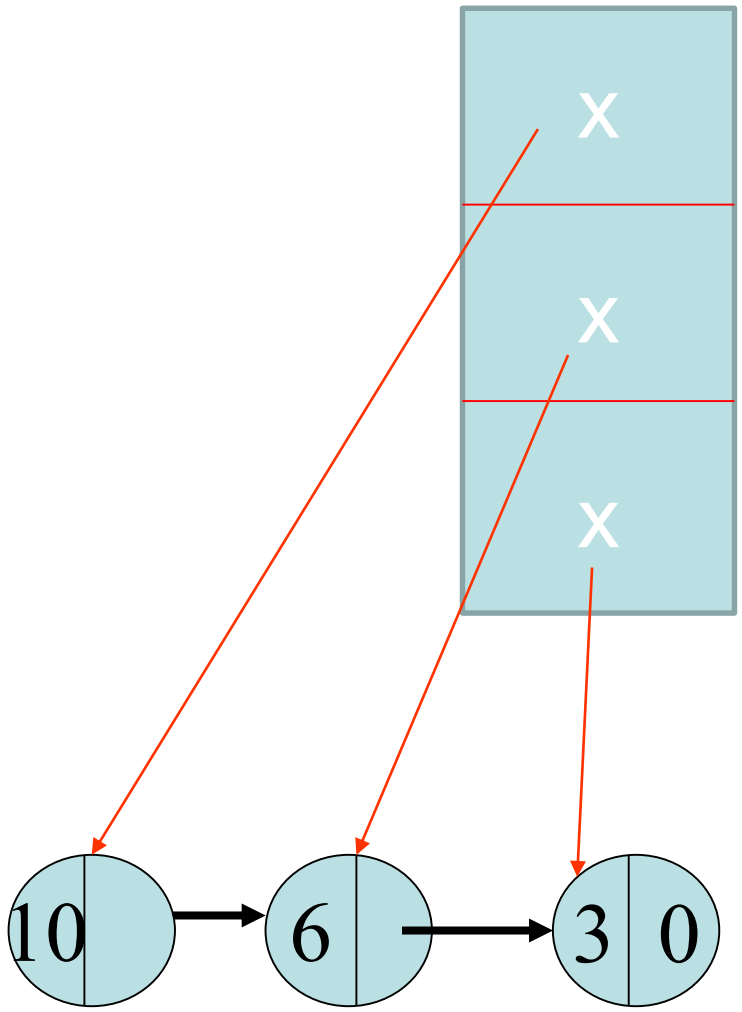
```
struct nodo{char info; nodo* next;
nodo(char a='\0', nodo*b=0){info=a; next=b;} };
```

si accede ad una lista attraverso una variabile
nodo* n
l'intera lista puntata da n è indicata con Lista(n) o L(n)
Lista(n) è ben formata o corretta se
--n è 0
--oppure se n punta ad un nodo il cui campo next è una lista ben formata (detto il resto di L(n))

liste non ben formate si producono per errore

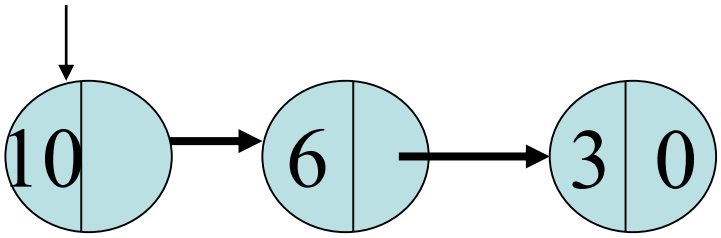
```
void stampa(nodo *x)
{
 if(x)
        {
          cout<< x->info;
          stampa(x->next);
        }
}
```

ogni invocazione ha una x
che punta ad un nodo
ma non serve!!
è ricorsione terminale



si può fare facilmente anche col while

```
nodo *x=inizio;
while(x!=0){
                cout<< x->info<<endl;
                 x=x->next;
              }
```

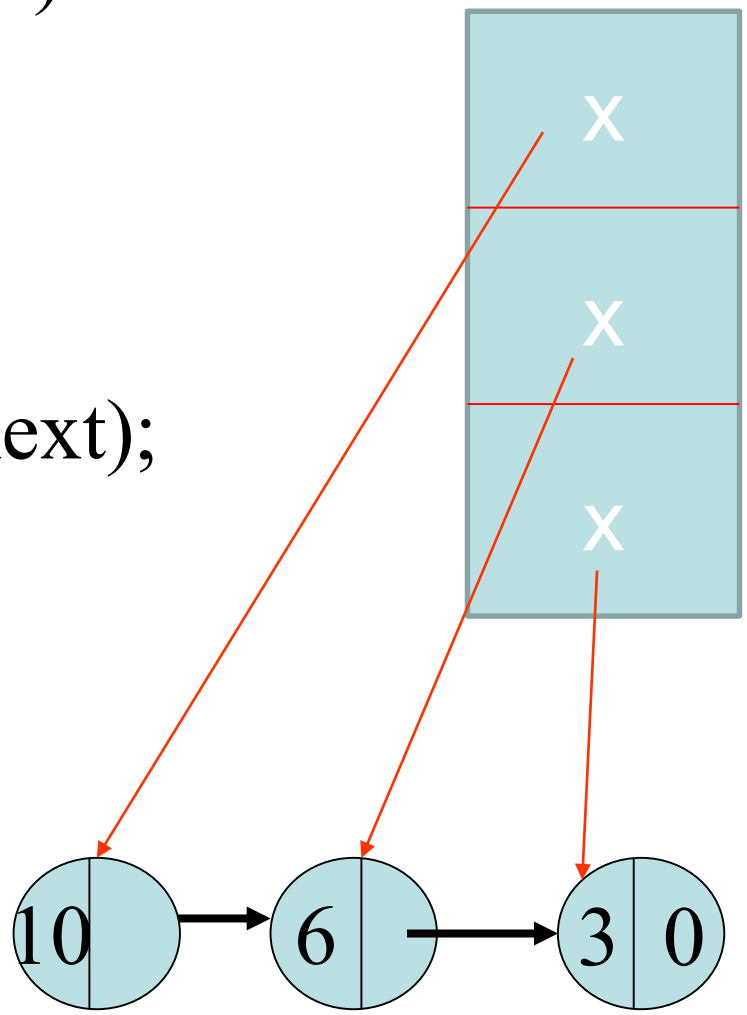


la variabile x scorre i 3 nodi, il ciclo si ferma quando x=0. cioè quando «esce» dalla lista

stampa dal fondo

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

servono le 3 x !!
non è terminale !!



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

ricorsione terminale
equivalente a while

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

non terminale
più complicato da
simulare col while

PRE= Lista(L) ben formata

```
void stampa_LR_iter(nodo*L)
{       int k=length(L);
        nodo* X=new nodo*[k];
        for(int i=0; i<k; i++) //andata
                {X[i]=L; L=L->next;}
        for(int i=k-1; i>=0; i--) //ritorno
                cout << X[i]->info<<‘ ‘;
        cout<<endl;
}
```

POST= stampa i campi info di Lista(L) da destra a sinistra

lunghezza di una lista

PRE=(Lista(L) è ben formata)

```
int lung(nodo* L)
{
   if(!L) return 0;
   return 1 + lung(L->next);
}
```

POST=(restituisce la lunghezza di Lista(L))

costruiamo una lista di lunghezza m >=0

PRE=(m>=0)

```
nodo* build(int m)
{
   if(m==0) return 0;
   return new nodo(m,build(m-1));
}
```

POST=(restituisce una lista con m nodi con info= m..1)

Induzione ?

una lista con i campi info crescenti

PRE=(0<=m<= dim)

```
nodo* build1(int m, int dim)
{
        if(m==dim) return 0;
        return new nodo(m, build1(m+1,dim));
}
```

POST=(restituisce una lista con dim - m nodi e campi info = m..dim-1)

lista con campi letti da cin

```
nodo* build2(int m)
{
    if(!m) return 0;
    int x;
    cin >> x;
    return new nodo(x, build2(m-1));
}
```

distruzione di una lista, Right to Left

```
void del(nodo*L)
{
  if(L)
      {
       del(L->next);
       delete L;
      }
}
```

non ricorsiva
terminale

ma potremmo anche farla Left to Right

```
void del(nodo*L)
{
  if(L)
      {
        nodo*x=L->next;
        delete L;
        del(x);
       }
}
```

ricorsiva terminale

iterativamente

```
while (L)
{
   nodo*x=L->next;
  delete L;
   L=x;
}
```

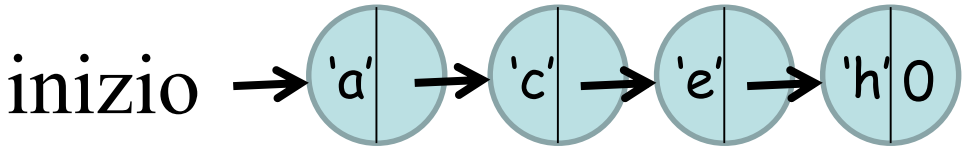
```
void del(nodo*L)
{
 if(L)
      {
        nodo*x=L->next;
        delete L;
        del(x);
      }
}
```

distruggere una lista iterativamente RL:

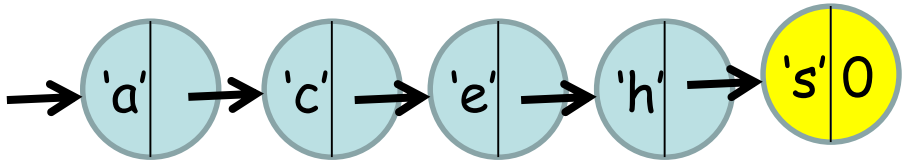
```
int k=lung(L), int i;
nodo* pila= new nodo* [k];
for( int i=0; i<k; i++)                    andata
  {pila[i]=L; L=L->next;}
for(int i=k-1; i>=0; i--)                  ritorno
        delete pila[i];
```

pila simula la pila di RA della ricorsione

Operazione: inserire un elemento alla fine di una lista



insEnd(inizio, 's') restituisce



vediamo 3 modi di realizzare questa operazione

ci serviranno da guida per il futuro!

I soluzione

diamo la lista iniziale in input e otteniamo la nuova lista come risultato restituito col return

nodo* insEnd(nodo*, char)

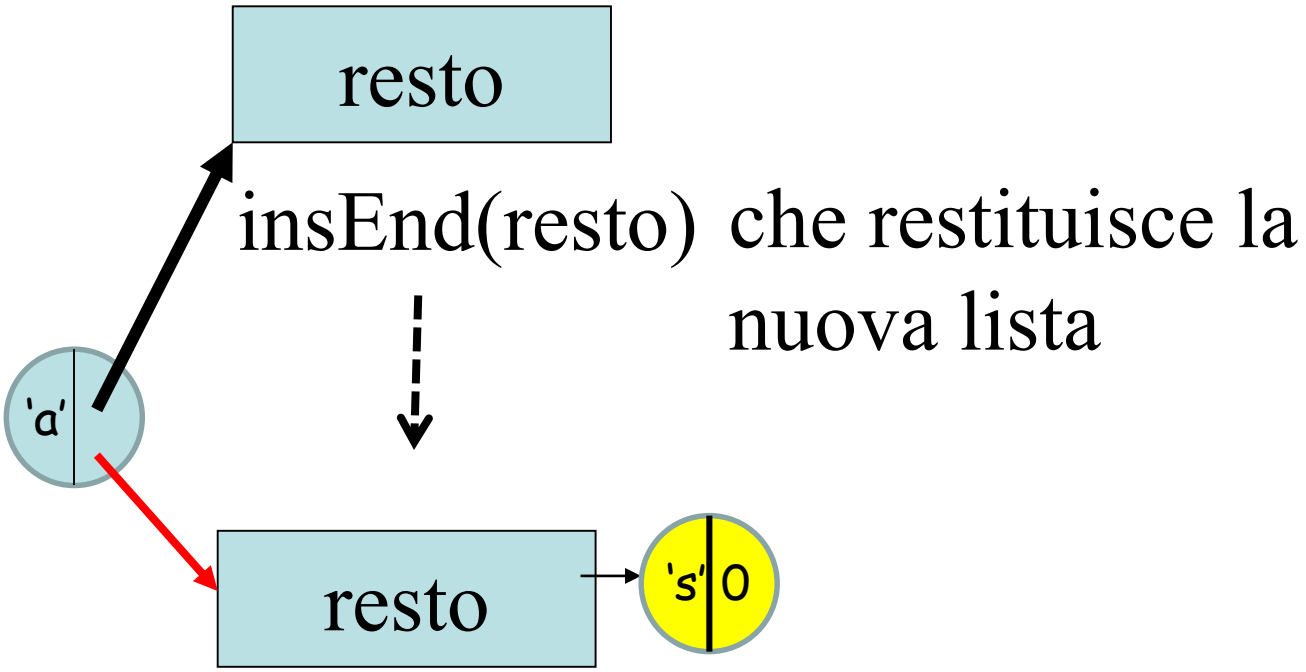
tutto passato e restituito per valore

soluzione + semplice: sempre la prima da tentare

caso base: lista vuota

return new nodo(‘s’,0);

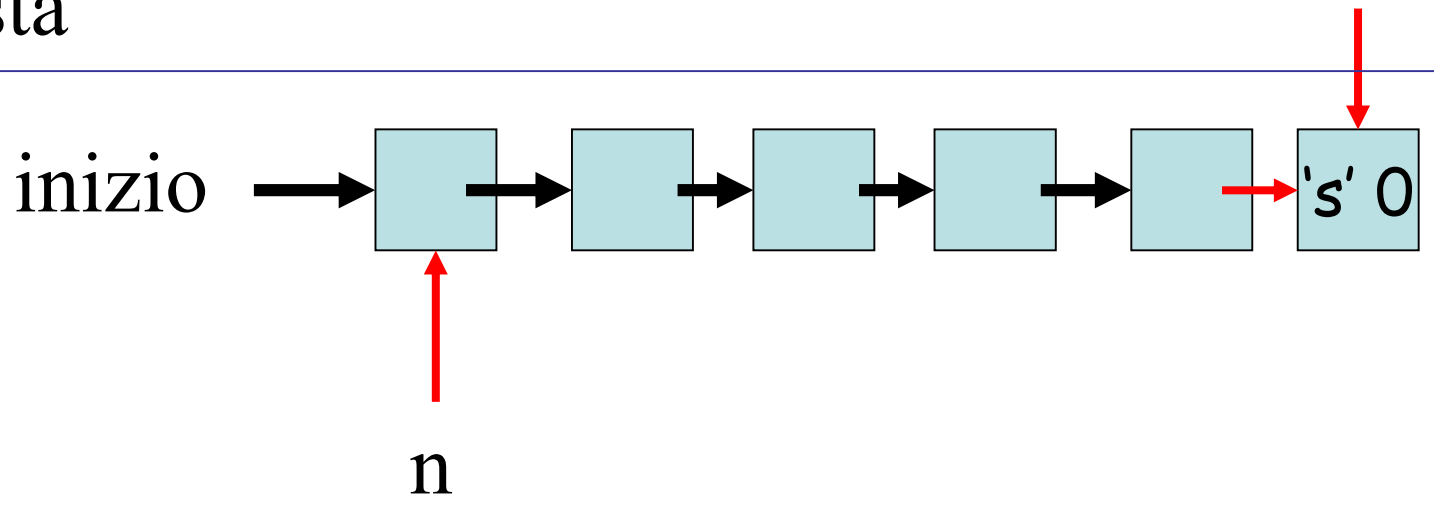
caso ricorsivo: lista con un nodo almeno



Realizzazione:

--all'andata della ricorsione troviamo la fine della lista => caso base: lista vuota

--al ritorno costruiamo la lista allungata collegando ogni nodo con il nuovo resto della lista



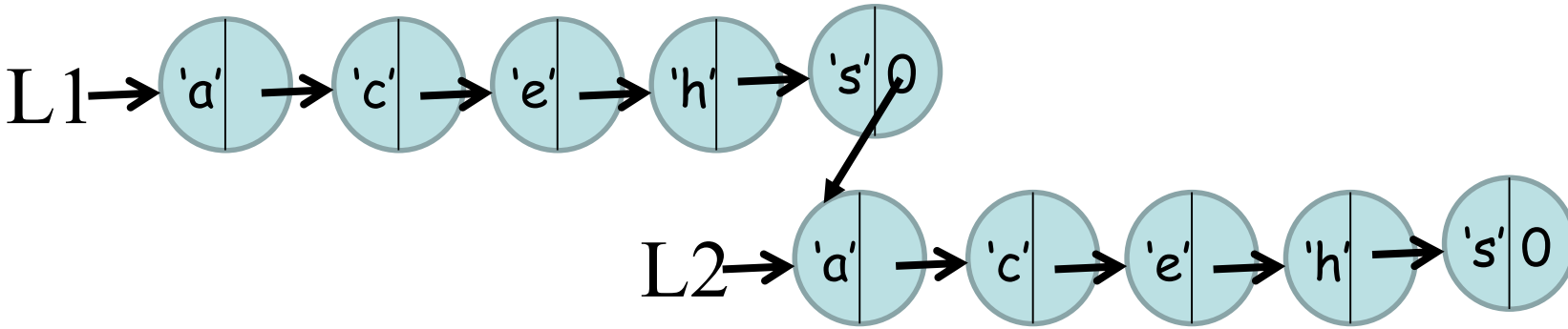
nella funzione che stiamo per descrivere

usiamo il fatto che :

puntatore = 0 = false

puntatore !=0 = true

la concatenazione di due liste è indicata con L1@ L2



PRE=(Lista(n) è ben formata, vLista(n)=Lista(n)
)

```
nodo * insEnd1(nodo *n, char c)
{
        if(! n) return new nodo(c,0);
        else
        {
            n→next=insEnd1(n→next,c);
            return n;
        }
}
```

POST=(restituisce vLista(n)@nodo(c,0))

prova induttiva, *usiamo L(n) al posto di Lista(n)*

base: vL(n) è vuota, vL(n)@nodo(c,0)=nodo(c,0)

passo induttivo: vL(n)=n@resto

da PRE = (L(n) ben formato) e n != 0 segue
PRE_ric=(resto è ben formato) => POST_ric

n->next= insEnd(n→next,c)

n@resto@nodo(c,0)

vL(n) @ nodo(c,0) => POST

invocazione iniziale:

inizio=insEnd(inizio,’s’);

PRE=(Lista(n) è ben formata, vLista(n)=Lista(n))

```
nodo * insEnd1(nodo *n, char c)
{
        if(! n) return new nodo(c,0);
        else
        {
            n→next=insEnd1(n→next,c);
            return n;
        }
}
```

POST=(restituisce vLista(n)@nodo(c,0))

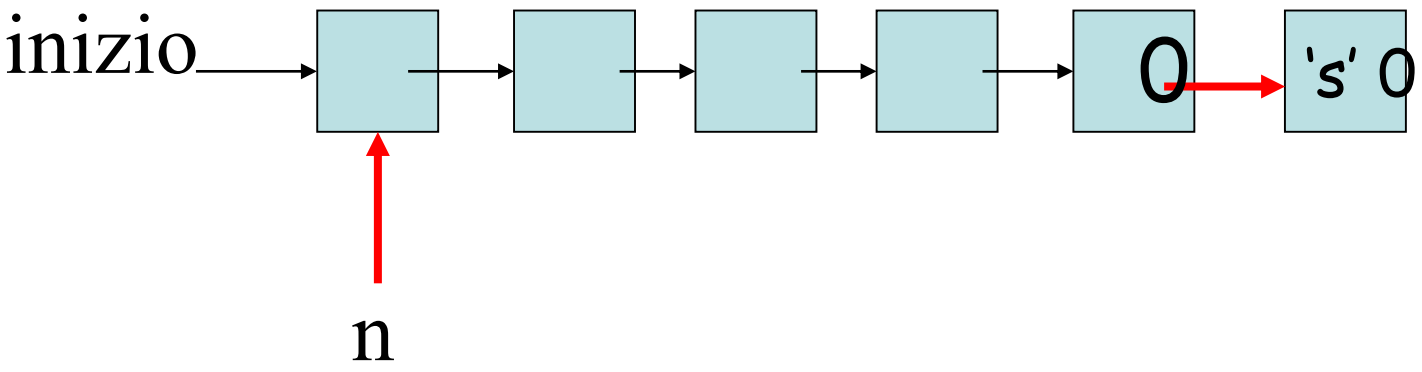
In conclusione per una lista non vuota, le operazioni che insEnd1 esegue sono:
-all'andata:
--- scorrere la lista fino a lista vuota=0
---la creazione del nuovo nodo
-al ritorno:
---suo aggancio a quello che era l'ultimo nodo
---aggancio della nuova lista ai nodi precedenti
-la prima invocazione restituisce la nuova lista completa al chiamante

le operazioni in rosso sono INUTILI, cerchiamo di non farle

II soluzione: all'andata ci fermiamo all'ultimo nodo e gli appendiamo il nuovo nodo

....ma ci deve essere almeno un nodo

=> inizio non cambia mai => la funzione ritorna void



PRE=(L(n) è ben formata e non vuota vL(n)=L(n))

```
void insEnd2(nodo *n, char c)                                    {
        if( ! n→next)  //caso base = ultimo nodo
                n→next=new nodo(c,0);
        else
                insEnd2(n→next,c);
}
```

POST=(L(n) è diventato vL(n)@nodo(c,0))

anche se n non è cambiato

```
void insEnd2(nodo *n, char c)
{       if( !n→next)
                n→next=new nodo(c,0);
        else
                insEnd2(n→next,c);
}
```

da chiamare solo con inizio !=0

```
if(inizio) ins_end(inizio, ‘s’);

else  inizio=new nodo(‘s’,0);
```

Prova induttiva di correttezza:

base: vL(n) ha un solo nodo,
n -> next= new nodo(c,0) trasforma L(n) come richiede la POST
passo ricorsivo: ci sono almeno 2 nodi
vL(n)=n @ resto, con resto non 0
=> vale PRE_RIC
=> vale POST_ric, cioè, dopo insEnd2(n→next,c), resto è diventato resto@nodo(c,0)
ma n punta a resto e quindi vale POST

```
void insEnd2(nodo *n, char c)
{
        if(n)
        {
                if( ! n→next)
                        n→next=new nodo(c,0);
                else
                        insEnd2(n→next,c);
         }
 }
```

riassumiamo :

soluzione I : all'andata si oltrepassa l'ultimo nodo (caso base n = 0) e poi si costruisce la nuova lista al ritorno

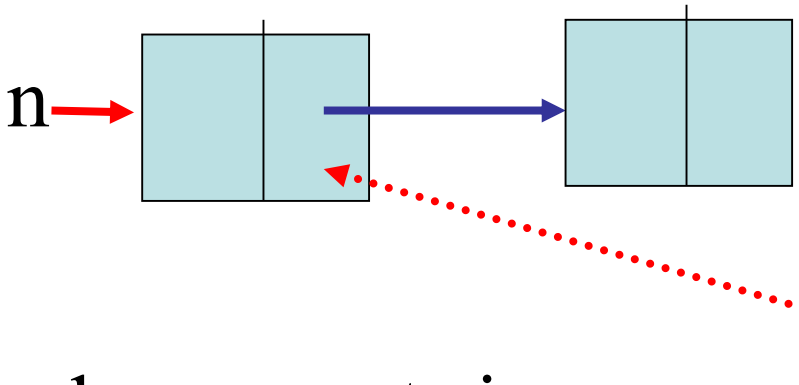
soluzione II: all'andata ci si ferma all'ultimo nodo (caso base n→next =0) e gli si attacca il nuovo nodo. Non si fa nulla al ritorno

la II evita operazioni inutili, ma non gestisce il caso della lista vuota

vorremmo contemporaneamente poter modificare il campo next dell'ultimo nodo ma fermare la ricorsione con n=0

possiamo ottenerlo passando il nodo n per riferimento

F(nodo * **&** n){…F(n→next)…}



passando n->next si passa un alias di questo campo

III soluzione: con n passato per riferimento

PRE=(L(n) è ben formata, vL(n)=L(n))

```
void insEnd3(nodo*& n , char c)
{
 if(!n)
    n=new nodo(c,0);
 else
    insEnd3(n->next, c);
}
```

POST=(L(n) = vL(n) @ nodo(c,0))

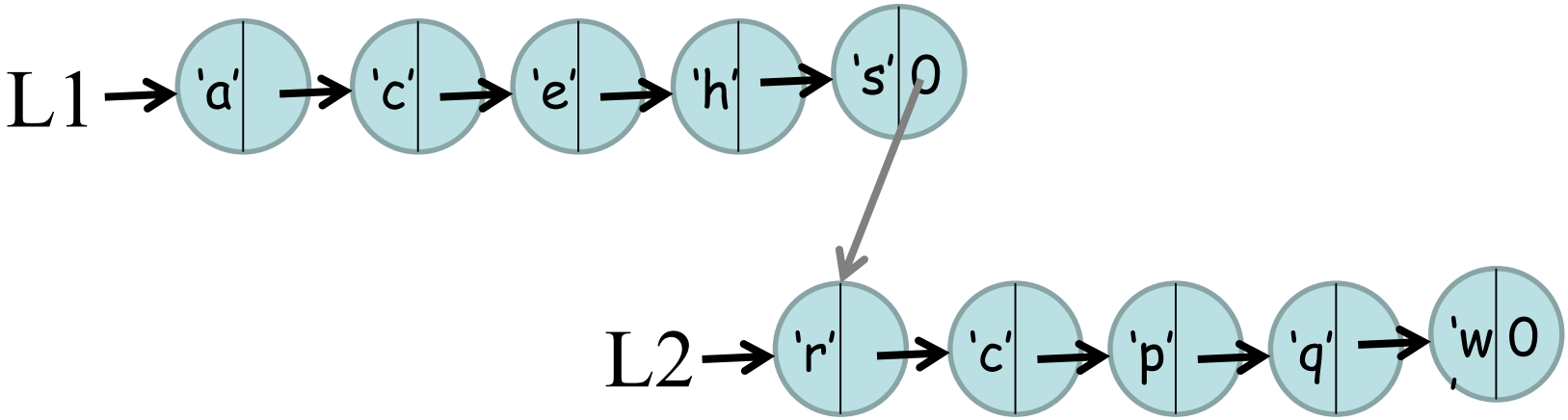
il passaggio per riferimento garantisce che la nuova lista sia restituita al chiamante

Prova induttiva di correttezza:

base: n=0 : n= new nodo(c,0)
L(n)=nodo(c,0) = POST

passo ricorsivo: vL(n)=n @ vL(n->next)
vale PRE_ric => usiamo POST_ric =L(n->next)=
vL(n->next) @ nodo(c,0)
n->next è il campo next di *n e quindi
L(n)= n @ L(n->next) = vL(n) @ nodo(c,0)
$\Rightarrow$ POST

concatenazione di liste:



potrebbe essere che L1 e/o L2 siano vuote

<u>usiamo la notazione</u>: Lista(L1) @ Lista(L2)

funzione concatenzazione: usiamo metodo 1

PRE=(Lista(L1) e Lista(L2) ben formate)

```
nodo* conc(nodo* L1, nodo* L2)
{
  if(!L1) return L2;
  L1->next=conc(L1->next, L2);
  return L1;
}
```

POST=(restituisce Lista(L1)@Lista(L2))