



liste concatenate

ricordare che nella ricorsione:

andata

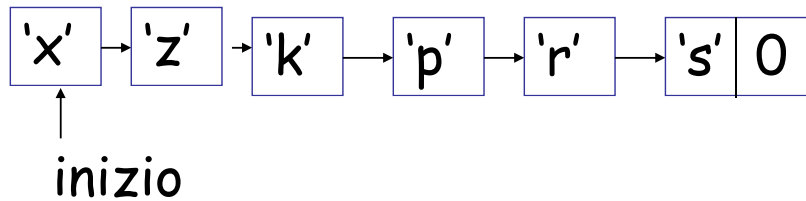
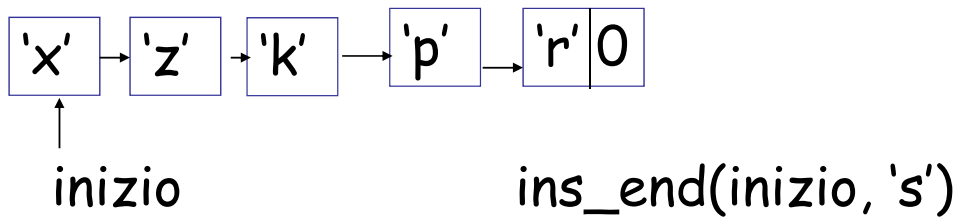
ric()  ric()  ric()

ritorno

calcolare all'andata e/o al ritorno

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

inserire un elemento alla fine della lista

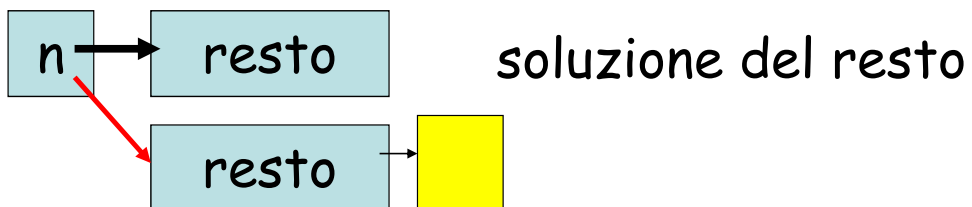


I soluzione

caso base: lista vuota

si tratta di creare il nuovo nodo e restituirlo

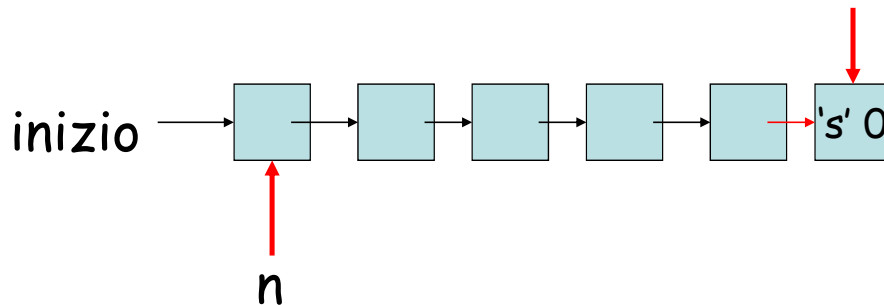
caso ricorsivo: lista con un nodo almeno



Realizzazione:

.all'andata troviamo la fine della lista

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



nella funzione che stiamo per
descrivere

usiamo il fatto che :

puntatore == 0 = false

puntatore != 0 = true

```

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

```

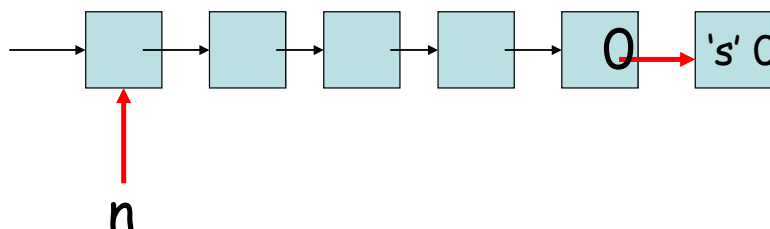
fine
ricorsione

invocazione:

inizio=ins_end(inizio,'s');

invocazioni
intermedie

II soluzione: facciamo tutto all'andata,
significa che la funzione restituisce void
e quindi non possiamo modificare **inizio**
che punta al primo nodo
funziona solo per liste non vuote



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

da chiamare
solo con $n \neq 0$

```
if(inizio) ins_end(inizio, 's');
else inizio=new nodo('s',0);
```

riassumiamo:

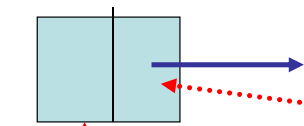
soluzione I : all'andata si passa l'ultimo nodo ($n == 0$) e poi si costruisce la nuova lista al ritorno

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

la II è più semplice, ma non gestisce il caso della lista vuota

vorremmo contemporaneamente
poter modificare il campo next
dell'ultimo nodo ma fermarci con $n == 0$
possiamo ottenerlo passando il nodo
per riferimento

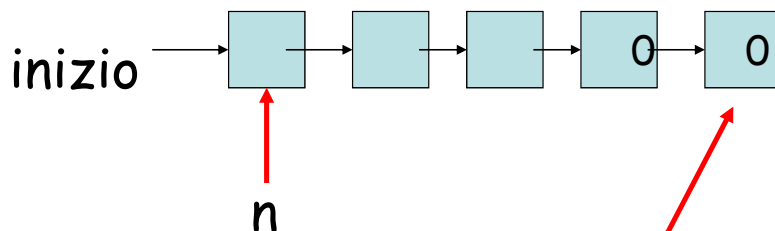
$F(\text{nodo} * \& n) \{ \dots F(n \rightarrow \text{next}) \dots \}$



$F(n) \dots \rightarrow F(n \rightarrow \text{next})$

nella prossima
 $F n$ è un alias di
questo
puntatore

soluzioni col passaggio di n per valore



TROPPO
TARDI !!

ci dobbiamo fermare qui, con $n \rightarrow \text{next} == 0$
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 ?



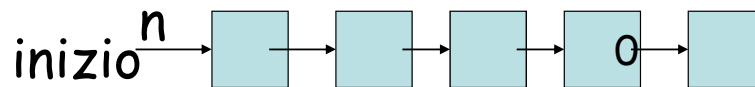
poco generale
BRUTTA !!

dobbiamo fermarci qui

$n \rightarrow next \rightarrow next == 0$

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

soluzione 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...)
```

```
}
```

esercizio

realizzare la cancellazione
dell'ultimo nodo di una lista in
modo iterativo

prima cosa da pensare: a quale nodo si
deve fermare il ciclo ?

altro esercizio distruggere l'intera lista

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

l'ordine delle
cose è
MOLTO
importante !!

invocazione:

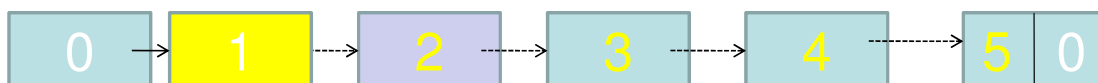
del_all(inizio);

inizio=0;

esercizio

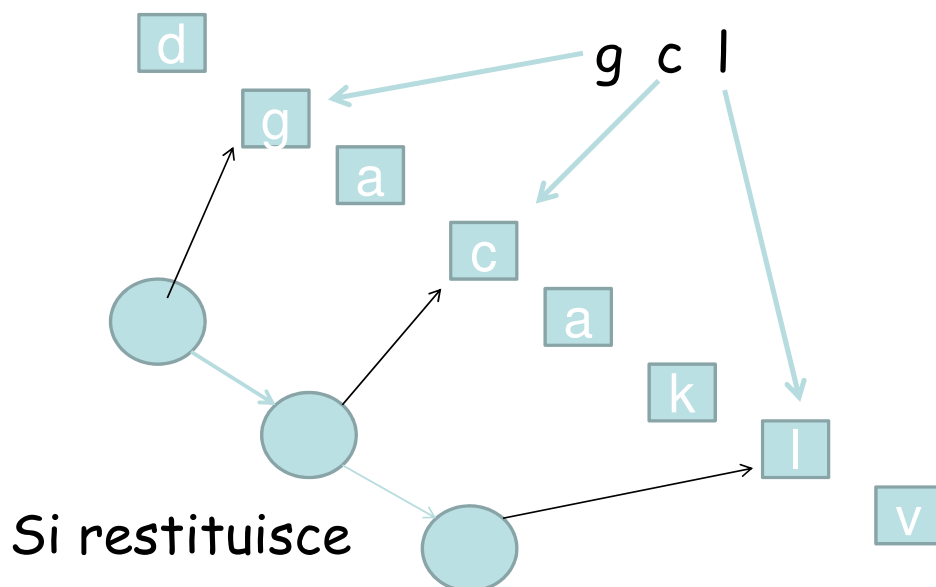


Inserire un nodo in posizione 1

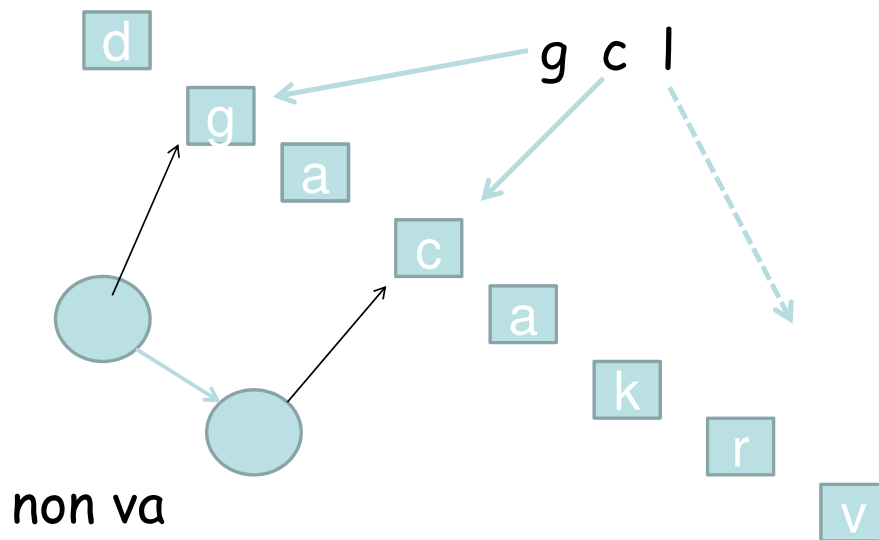


Su quale nodo ci si deve fermare ?
Se inseriamo in posizione k:
-sul nodo in posizione k
-sul nodo precedente k-1

Pattern matching su una lista :



Ma potrebbe anche non funzionare



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

```
nodoP * PM(nodo*n, char *P, int dimP, bool & b)
{if(dimP)
    if(n)
        if(n->info==*P) {
            nodoP*m =PM(n->next,P+1,dimP-1,b);
            if(b) return new nodoP(n,m);
            else return 0;}
        else return PM(n->next,P,dimP,b);
    else {b=false; return 0;} // fallimento
    {b=true; return 0;} // successo
}
```

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)