Liste concatenate 2

esercizio: eliminare ultimo nodo di una lista

PRE=(L(n) è lista corretta e non vuota, vL(n)=L(n))

POST=(restituisce vL(n) – ultimo nodo)

caso base ?

lista con un solo nodo e deve diventare la lista vuota

PRE=(L(n) è lista corretta e non vuota, vL(n)=L(n))

```
if(! n->next)
{delete n; return 0;}
```

POST=(restituisce vL(n) – ultimo nodo)

caso ricorsivo: lista con almeno 2 nodi:

si deve:

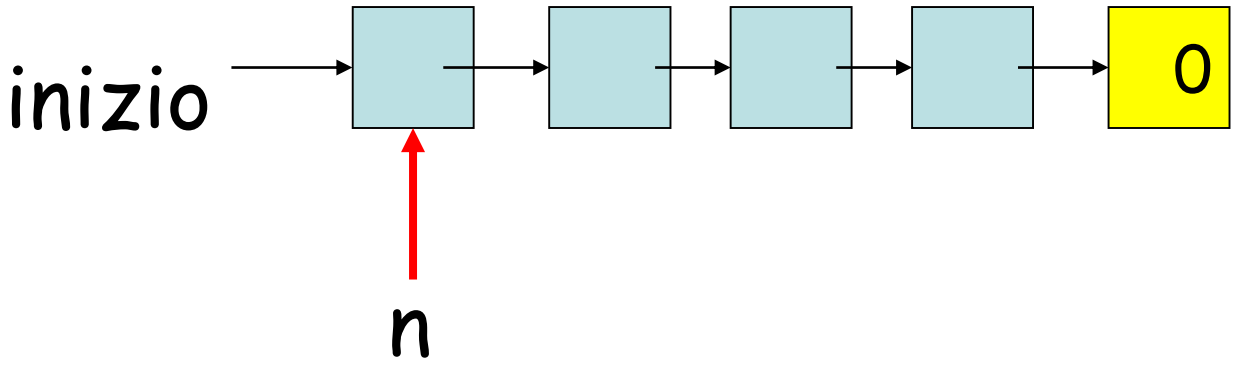
1) fare l'invocazione ricorsiva sul resto della lista e
2) appendere la lista che ritorna al nodo corrente

PRE_ric=(L(n->next) è corretta e non vuota)

n->next=del(n->next); return n;

POST_ric=(restituisce vL(n->next)-ult nodo)

I soluzione: quello che succede in pratica



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

=> deallocare questo nodo e restituire 0

riassumendo

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

invocazione:

```
if(inizio)  inizio=del(inizio);
```

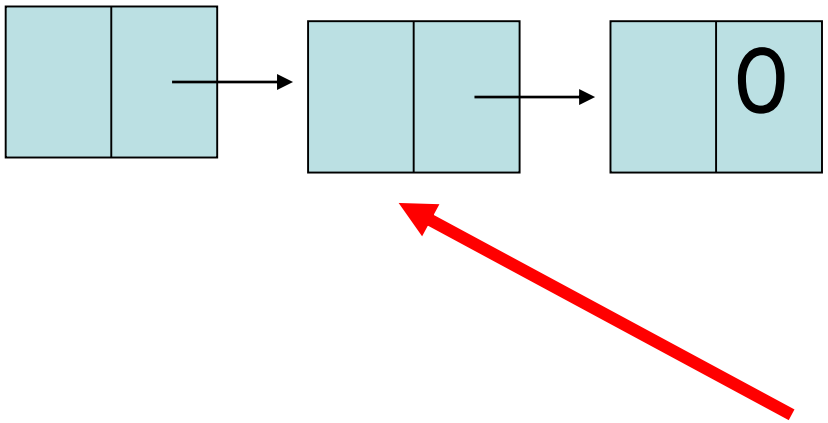
non ha senso se la lista è vuota

ci sono operazioni «inutili» ?

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

è possibile fare solo l'operazione che serve ?

dobbiamo fermare la ricorsione al penultimo nodo



poco generale
BRUTTA !!

dobbiamo fermarci qui

n→next→next ==0

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

II soluzione

PRE=(L(n) è corretta con almeno 2 nodi)

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

 else

  del(n→next);
```

} POST=(L(n)=vL(n)-ultimo nodo)

=> primo nodo di L(n) e uguale a quello di vL(n)

III soluzione: col passaggio per riferimento

passando n per riferimento, arriviamo all'ultimo nodo avendo un alias del next del nodo precedente (se c'è e, se non c'è della variabile che punta all'inizio della lista)

PRE=(L(n) è corretta non vuota)

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

POST=(L(n)=vL(n) – ultimo nodo)

invocazione: if(inizio) del(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 finora era:

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

altro esercizio distruggere l'intera lista

PRE=(L(x) è corretta)

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

} POST=(tutti i nodi di vL(x) sono stati deallocate)

invocazione: del_all(inizio); inizio=0;

l'ordine delle operazioni 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 la stampa inversa?

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

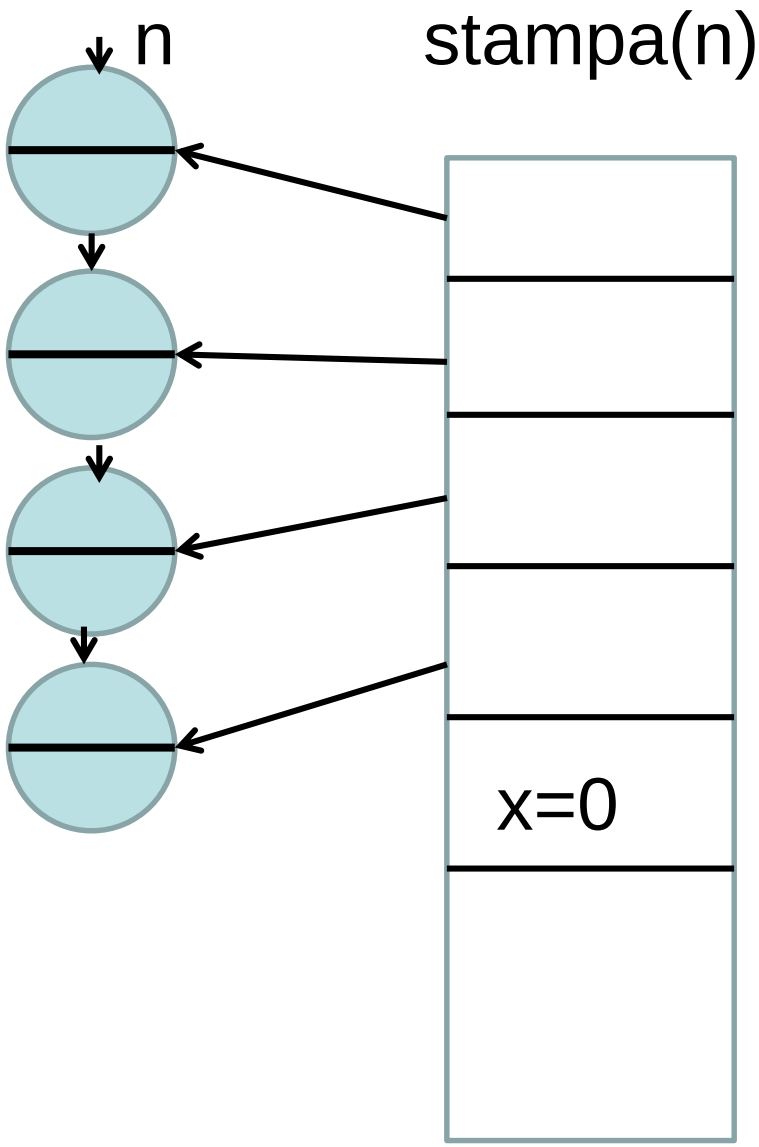
è ricorsivo terminale, facile farlo con l'iterazione

```
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<<' ';
}
```




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

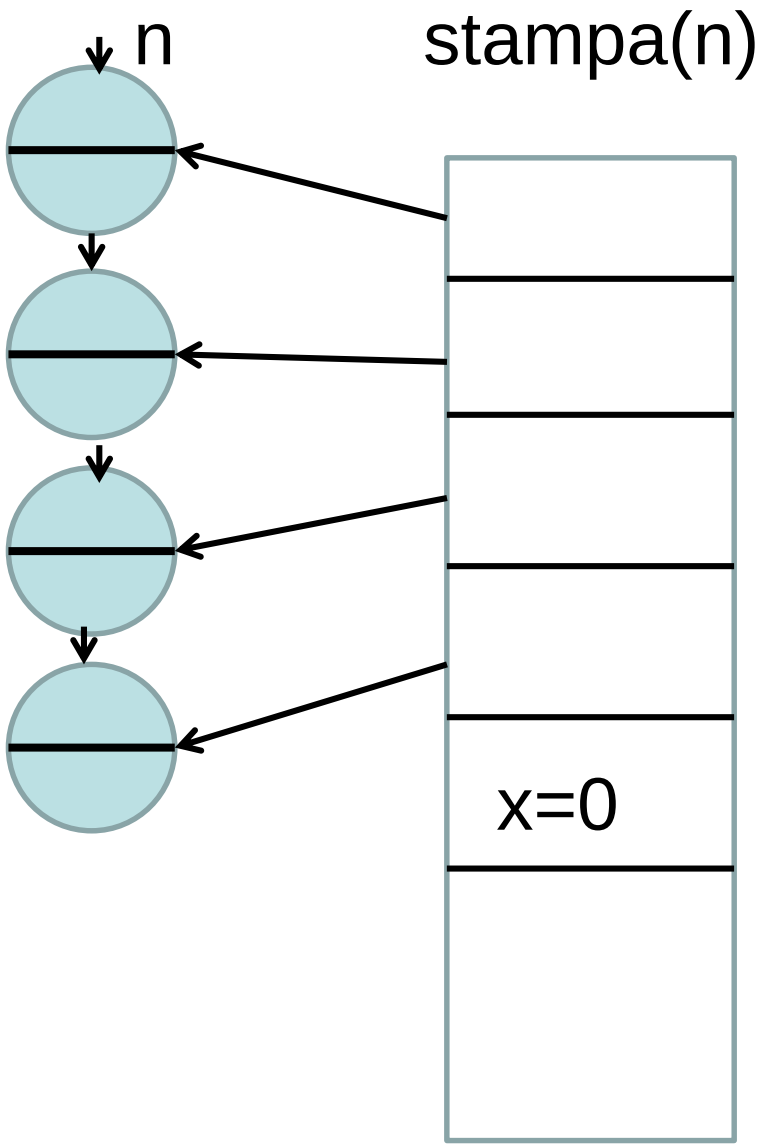
stack dei Record di Attivazione

```
int l=lung(n); nodo*x=n;
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<<' ';
```

la figura di prima ci aiuta a capire perché la ricorsione terminale è facile da simulare con l’iterazione



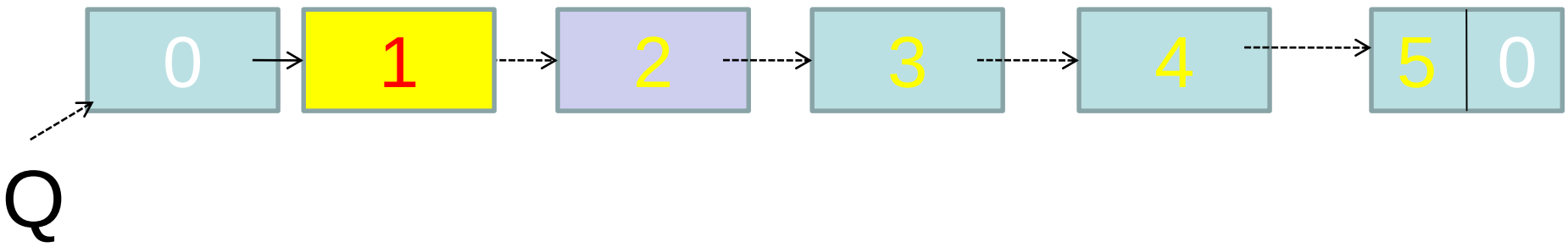

```
void stampa(nodo* x)
{
  if(x)
    {
      cout<< x->info<<' ';
      stampa(x->next);
    }
}
```

stack dei Record di Attivazione

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:

-I soluzione: passiamo il nodo k-1, serve andata e ritorno

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

conviene introdurre la seguente notazione:
dato nodo* Q con L(Q) lista corretta,
L_k(Q)= lista che consiste dei primi k nodi di L, cioè dal nodo 0 al k-1
attenzione ➔ L_0(Q)= lista vuota

il nodo finale di L_k(Q) è quello in posizione k-1, quindi se L(Q)=L_k(Q)@R

- passare il nodo k-1 significa fermarsi alla prima di R
- fermarsi al nodo k-1 significa all'ultimo nodo di L_k(Q)

-passiamo il nodo k-1:

PRE=(L(Q) corretta,k>=0, vL(Q)=L(Q))

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

POST=(se vL(Q)=vL_k(Q)@R, restituisce vL_k(Q)@(nodo(c)->R), altrimenti vL(Q))

-fermarsi sul nodo in posizione k-1:

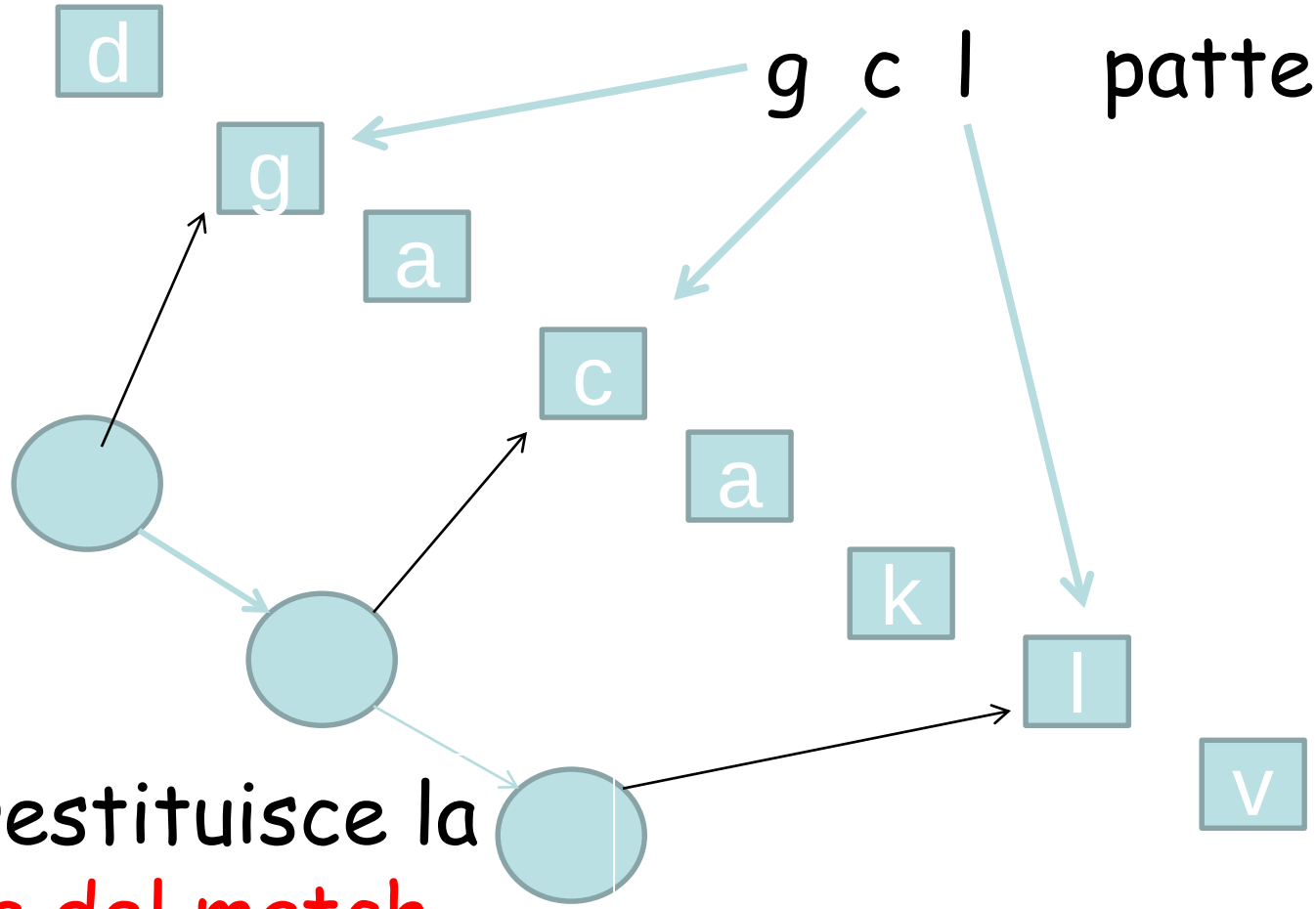
PRE=(L(Q) corretta e non vuota,k>0,
vL(Q)=L(Q))

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

POST=(se vL(Q)=vL_k(Q)@R,
L(Q)=vL_k(Q)@(nodo(c)->R), altrimenti =vL(Q))

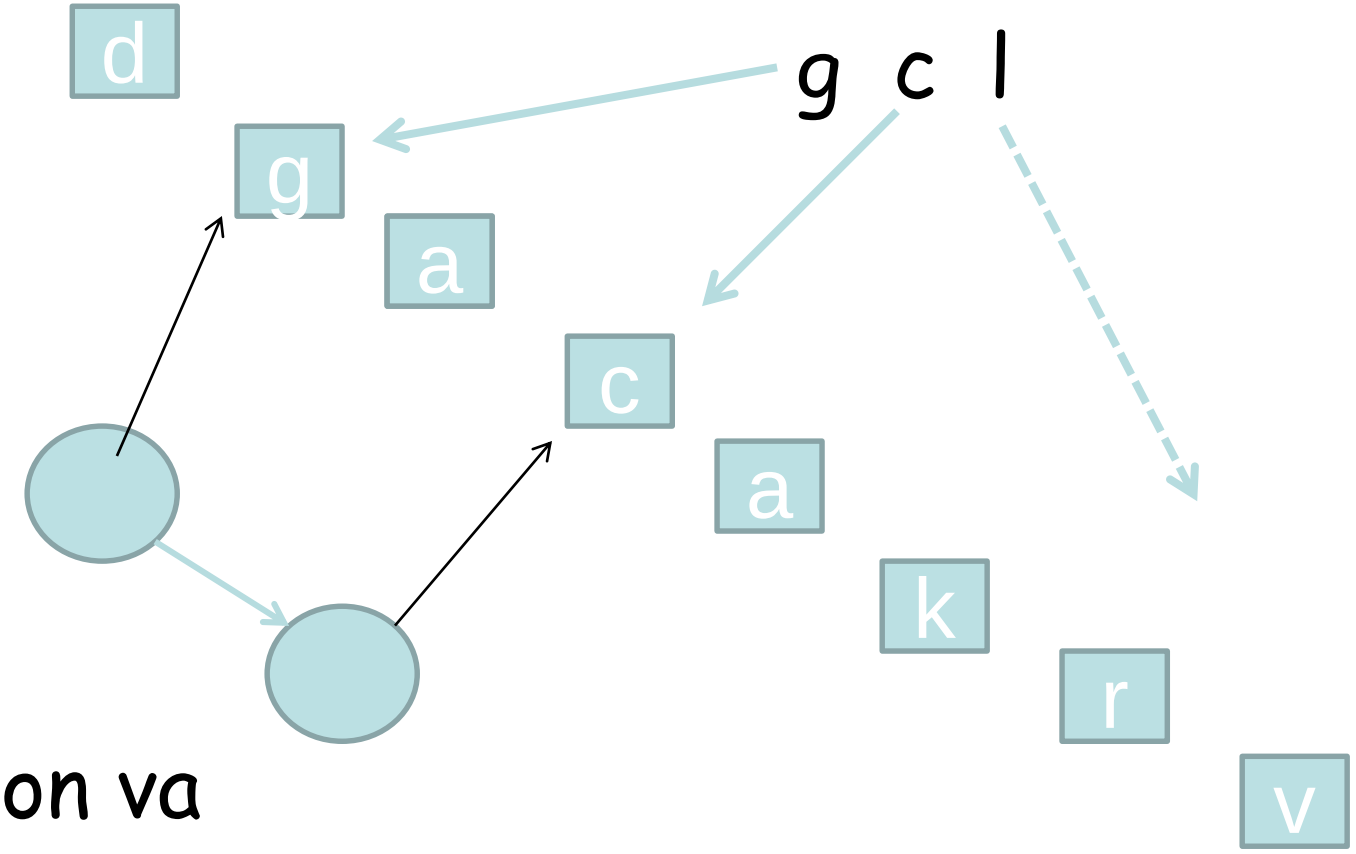
Pattern matching completo su una lista :

testo



Si restituisce la lista del match

Ma potrebbe anche non funzionare



non va

ci serve un nuovo tipo di nodo

quello solito è:

struct nodo{char info; nodo* next;}

quello nuovo è

struct nodop{nodo* info; nodop* next;}

idea: costruiamo la lista del match solo se c'è match

All'andata percorriamo la lista e verifichiamo i match possibili
e al ritorno :

1-facciamo sapere se il match ha avuto successo o no

2-solo se il match ha avuto successo si costruisce la lista del match

per far sapere se il match ha avuto successo ci sono 3 possibili tecniche:

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

2: Comunichiamo il successo/insuccesso con la stessa lista del match

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

complica il caso base

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

PRE=(P[0..dimP-1] def., dimP>=0,L(Q) lista corretta, ok=false)

```
nodop* match(char*P, int dimP, nodo* Q, bool & ok)
{
  if(!dimP) {ok=true; return 0;}
  if(!Q) return 0;

  if(Q->info==*P)
   {
      nodop* x=match(P+1,dimP-1,Q->next, ok);
      if(ok)
       return new nodop(Q,x);
      else
        return 0;
   }
   else
     return match(P,dimP,Q->next,ok);
}
```

bool per riferimento

POST=(ok sse esiste match di P[0..dimP-1] in L(Q))&&(ok=> restituisce lista di nodop che punta a nodi di L(Q) con match)&&(!ok=> restituisce 0)

PRE=(P[0..dimP-1] def., dimP>0,L(Q) lista corretta)

```
nodop* match(char*P, int dimP, nodo* Q)
 {
   if(!Q) return 0;

  if(Q->info==*P)
   {
      if(dimP==1)  return new nodop(Q,0);
       else
      nodop* x=match(P+1,dimP-1,Q->next);
      if(x)
       return new nodop(Q,x);
      else
        return 0;
   }
   else
     return match(P,dimP,Q->next);
}
```

lista non vuota restituita in caso di successo

POST=(restituisce lista X non 0 sse esiste match di P[0..dimP-1] in L(Q))&&(X=>
X è lista di nodop che punta a nodi di L(Q) con match)

PRE=(P[0..dimP-1] def., dimP>=0,L(Q) lista corretta)

```
nodop* match(char*P, int dimP, nodo* Q)
 {
   if(!dimP) return 0;
   if(!Q) throw (false);

  if(Q->info==*P)

      return new nodop(Q,match(P+1,dimP-1,Q->next));

   else
      return match(P,dimP,Q->next);
}
```

con eccezioni

POST=(restituisce lista X di nodop che punta ai nodi di L(Q) in cui si è trovato il match completo di P[0..dimP-1])

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 di prefissi di P (anche sovrapposti)

- lunghezza massima dei match contigui di prefissi di P (non sovrapposti)