

# Parte 1 - Liste

## Liste



P. Picasso – Guernica, 1937

# Struttura dati

1. Modo sintetico per **organizzare** i dati
2. Insieme di operatori o **primitive** per la manipolazione degli elementi della struttura

**ARRAY**: struttura dati che memorizza elementi omogenei in *sequenza* (lineare), ad *accesso diretto*, e che ha *dimensione fissa*

*Accesso diretto*: Dato un indice  $i$ , si accede direttamente all'elemento in posizione  $i$ -esima

*Dimensione fissa*: La dimensione non può cambiare nel tempo

*Operatori*: inserimento di un elemento, cancellazione di un elemento, ricerca (dicotomica) di un elemento, ...

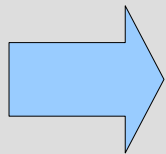
# Strutture dati dinamiche

- Per la risoluzione di alcuni problemi vi è la necessità di utilizzare *strutture dati dinamiche* ovvero la cui dimensione può cambiare nel tempo
- Si consideri ad esempio un problema che operi su una *sequenza di valori, il cui numero non è noto a priori, effettuando inserimenti ed estrazioni frequenti*
  - *Contatti, To-do, ...*
- Finora per memorizzare **elementi omogenei** abbiamo utilizzato gli array

# Limiti degli array (1)

## *Occupazione di memoria*

- La dimensione dell'array deve essere definita nella parte dichiarativa del programma o comunque al momento dell'allocazione (se dinamico)
- Se il numero di valori non è noto a priori è necessario *sovrastimarne*

 Occupazione di memoria potenzialmente molto maggiore del necessario

- Il vettore può *saturare*
- Come posso aggiungere un nuovo valore?

# Limiti degli array (2)

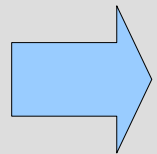
## *Velocità di esecuzione*

**Caso di inserimento in posizione i-esima:**

Shift in avanti di tutti gli elementi uguali o successivi alla posizione i-esima

**Caso di estrazione in posizione i-esima:**

Shift all'indietro di tutti gli elementi dall'i-esimo all'ultimo



Inefficiente per frequenti inserimenti/estrazioni

# Soluzioni?

- I limiti degli array sono dovuti alla *scarsa flessibilità* della loro struttura
- Elementi successivi dell'array devono essere collocati in locazioni contigue di memoria

**LISTA: Sequenza a dimensione variabile di  
elementi omogenei ad accesso sequenziale**

*Dimensione variabile:* La dimensione può variare nel tempo

*Accesso sequenziale:* Per accedere ad un elemento, devo prima accedere a tutti gli elementi che lo precedono nella sequenza

# Implementazione delle liste

- Alcuni linguaggi prevedono il tipo di dato Lista (ad esempio Python)
- Il C++ non ha il dato primitivo Lista

**Come possiamo implementarlo?**

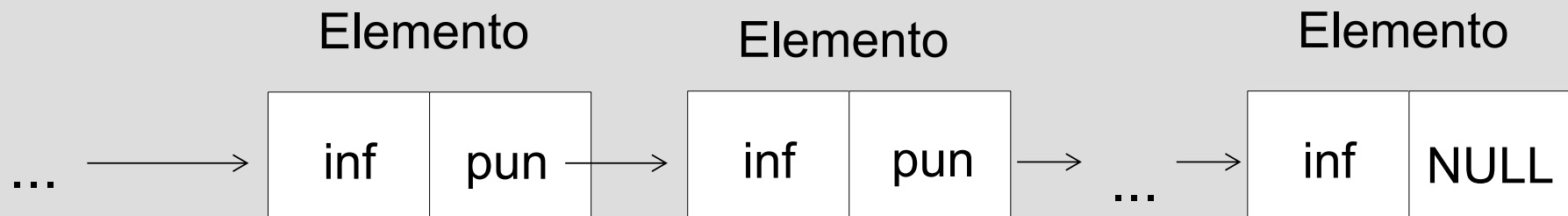
**ALLOCAZIONE DINAMICA + PUNTATORI**

**LINKED LIST**

- Ogni volta che dobbiamo aggiungere un elemento alla lista allochiamo un nuovo elemento
- Ogni volta che dobbiamo estrarre un elemento dalla lista, lo deallochiamo
- Gli elementi della lista non occupano posizioni contigue in memoria
- Usiamo i puntatori per collegare i vari elementi

# Linked list

- Ciascun elemento contiene un ***campo informazione*** (di qualunque tipo) ed un ***campo puntatore*** all'elemento successivo
- Il ***puntatore dell'ultimo elemento*** della lista non fa riferimento a nessun elemento (valore ***NULL***)





# Struttura dati in C++

*Elemento* implementato attraverso una *struct*:

```
struct elem {  
    int inf;        // o qualsiasi tipo semplice  
    elem* pun;      // definizione ricorsiva  
};
```

# Testa e coda della lista

**Testa** (o **head**) della lista = primo elemento della lista

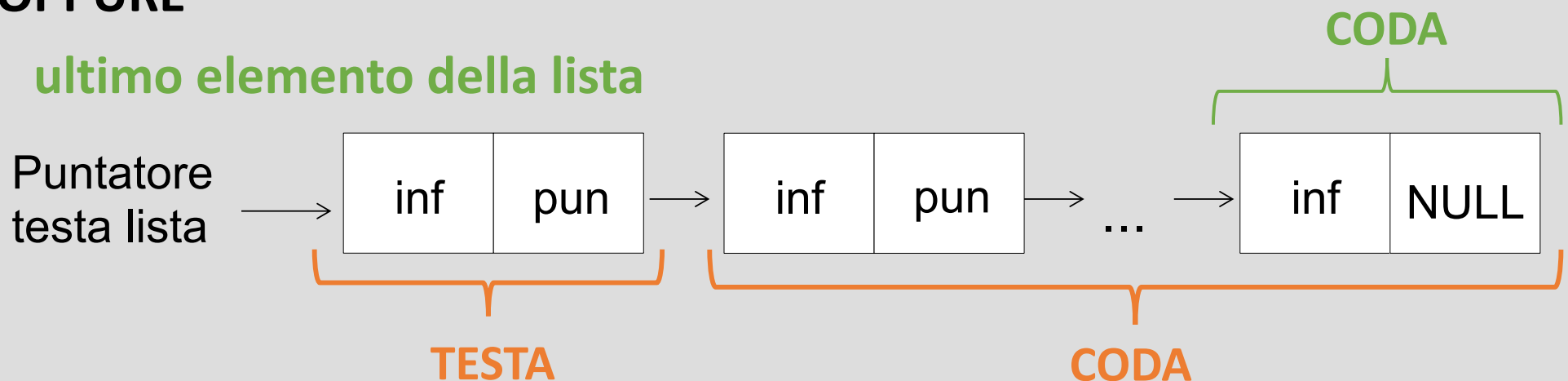
- L'accesso alla lista avviene attraverso il **puntatore alla testa della lista**
- Deve essere mantenuto in una variabile

**Coda** (o **tail**) della lista =

resto della lista puntata dalla testa

**OPPURE**

ultimo elemento della lista



# Struttura dati in C++

*Elemento* implementato attraverso una *struct*:

```
struct elem {  
    int inf;        // o qualsiasi tipo semplice  
    elem* pun;     // definizione ricorsiva  
};  
  
elem *testa; // puntatore alla testa della  
             lista  
  
// oppure:  
typedef elem* lista;  
lista testa;
```

# Osservazioni (1)

Definito il tipo:

```
struct elem {  
    int inf;  
    elem* pun;  
};
```

La seguente istruzione:

```
elem *p;
```

crea un oggetto dinamico di tipo elem ?

*No, crea solo il puntatore p*

## Osservazioni (2)

Dunque l'istruzione

```
elem *p;
```

deve essere seguita da un'allocazione

```
p = new elem;
```

(che alloca un elemento dinamico) prima di poter accedere ai campi dell'elemento puntato da ***p***:

```
p->inf      p->pun
```

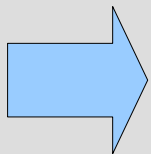
*Prima di dereferenziare un puntatore assicurarsi che punti ad un oggetto dinamico correttamente allocato!*

# Terminologia

***Lista semplice o singolarmente concatenata (singly linked list):***  
ogni elemento contiene un unico puntatore che lo collega all'elemento successivo

***Lista doppia o doppiamente concatenata (doubly linked list):***  
ogni elemento contiene due puntatori, uno all'elemento precedente e uno al successivo

***Lista vuota (empty list)*** = lista senza elementi identificata da un puntatore alla testa della lista avente valore **NULL**



In queste lezioni vedremo le *liste semplici*, il prossimo argomento riguarderà le *liste doppie*

# Caratteristiche di una lista semplice

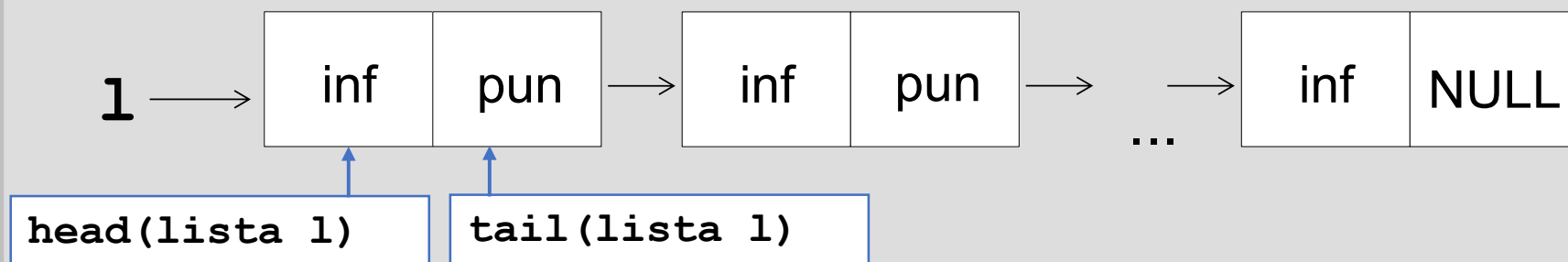
- Diversamente dagli array, gli elementi in memoria occupano *posizioni non sequenziali*
- Diversamente dagli array, in cui l'ordine è determinato dagli indici, l'*ordine è determinato da un puntatore in ogni elemento*
- Per determinare la fine della lista è *necessario il segnale di fine lista NULL*
- Non esiste modo di risalire da un elemento al suo antecedente → *scansione possibile solo da un elemento verso il successivo*

# Accesso alla lista: le primitive `head` e `tail`

- Per accedere agli elementi di una lista è sufficiente disporre di due primitive implementate come funzioni:

`int head(lista l)` restituisce la testa di `l`

`lista tail(lista l)` restituisce la coda di `l`



Il tipo  
restituito  
corrisponde  
al tipo di `inf`

```
int head(lista l){return l->inf;}
```

```
lista tail(lista l){return l->pun;}
```



# Stampa di una lista

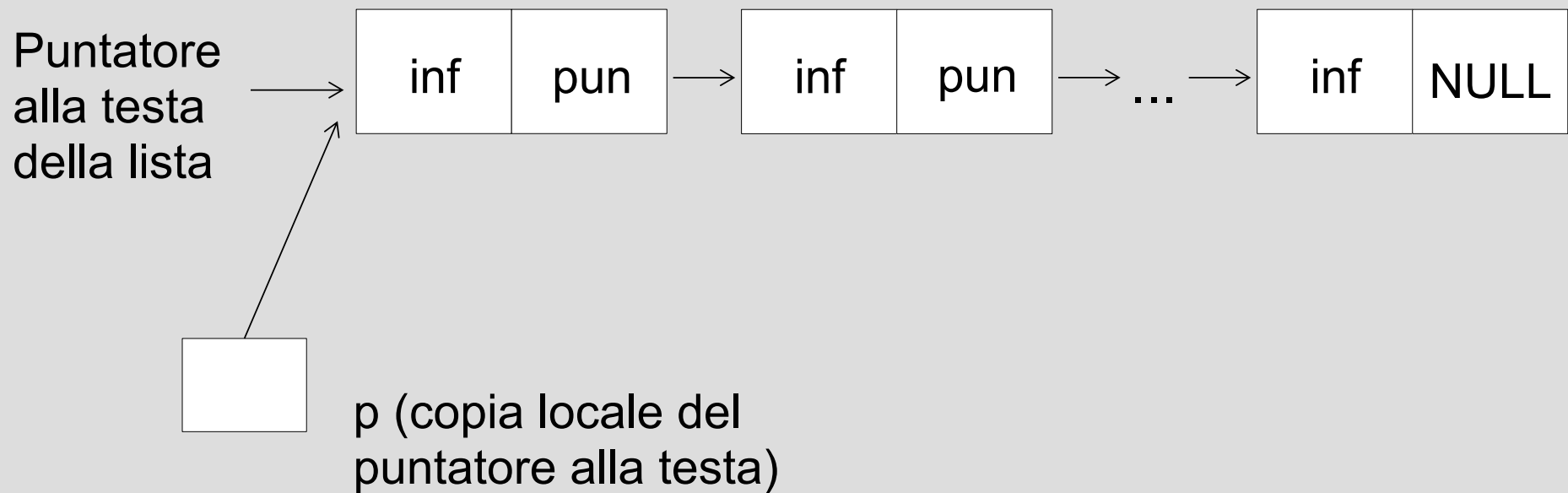
Problema da risolvere: *scandire una lista esistente fino alla fine (usando head e tail)*

```
void stampalista(lista p)
{ while (p != NULL) {
    cout << head(p) << " " ; // stampa valore in testa
    p = tail(p);    // spostamento sulla coda di p
}
cout << endl ;
}
```

*stampalista()  
prende in ingresso il  
puntatore alla testa della lista*

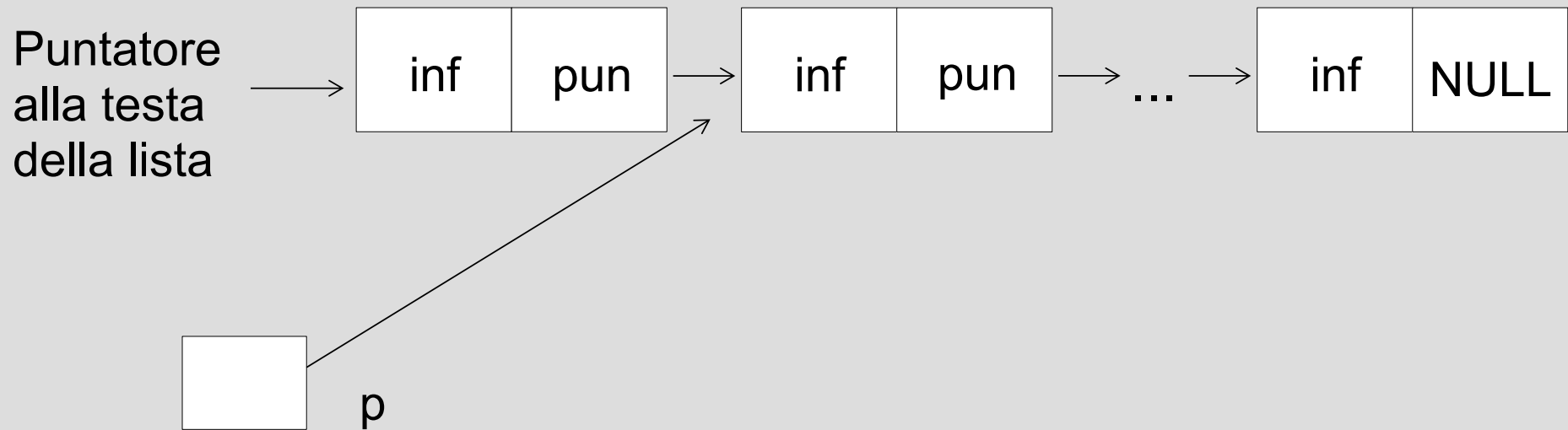
# Iterazioni di stampa

## Prima iterazione



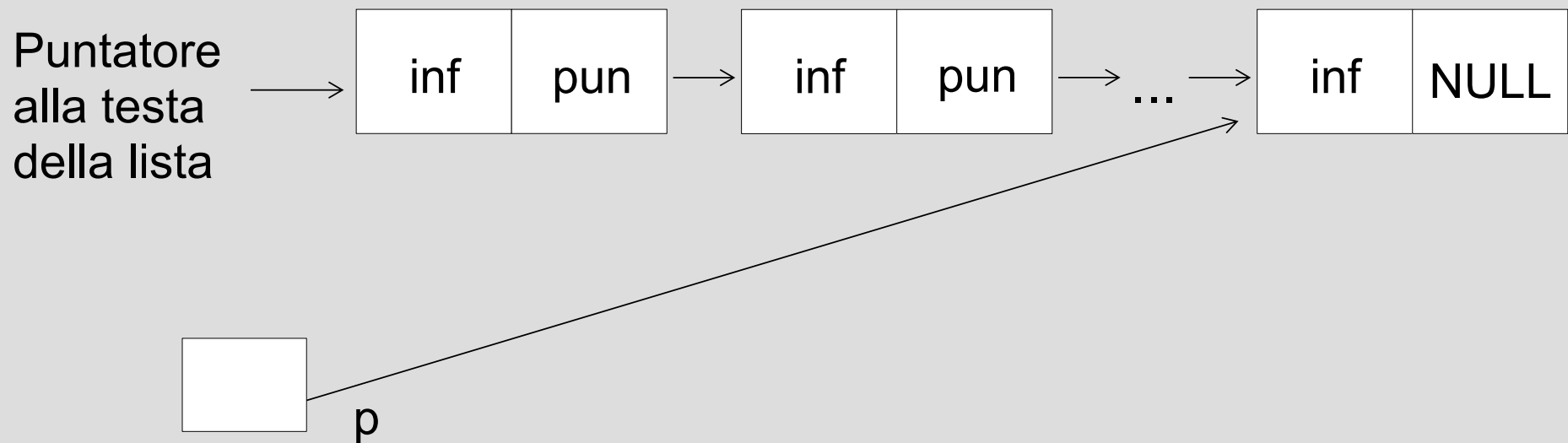
# Iterazioni di stampa

## Seconda iterazione



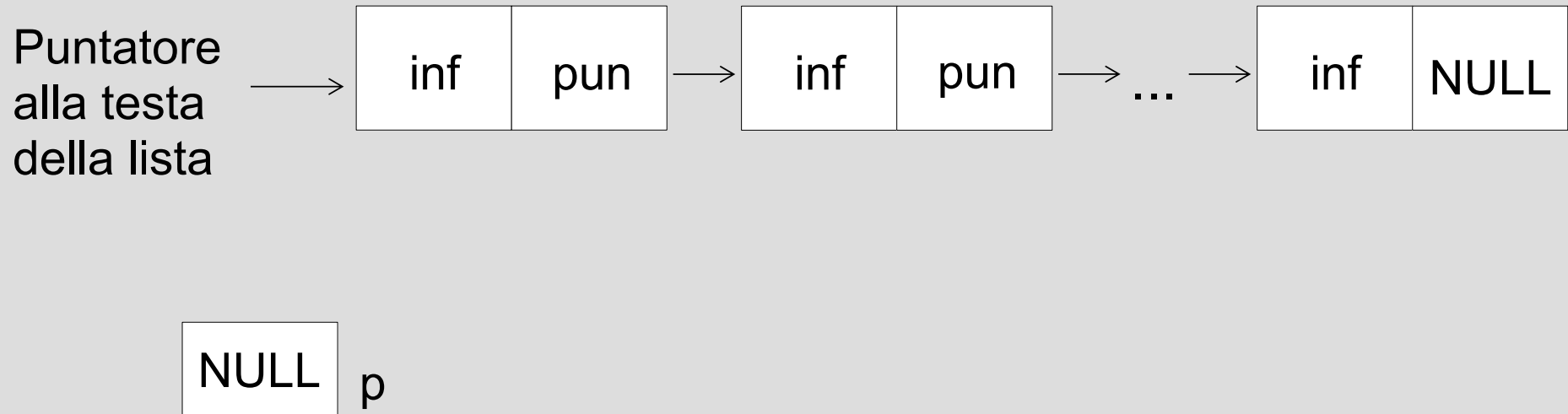
# Iterazioni di stampa

## Ultima iterazione



# Iterazioni di stampa

Dopo l'ultima iterazione



# Esercizio

Scrivere un programma in cui:

- sia definita (a tempo di scrittura del programma stesso) una lista formata da due elementi, contenenti i valori 3 e 7
- si stampi il contenuto della lista mediante la funzione **stampalista**

**Vedi programma** *stampa\_elem2.cc*

# Soluzione

```
main() {  
    lista testa = new elem;  
  
    testa->inf = 3;  
    elem * p = new elem; // creo l'elemento  
    p->inf = 7;  
    p->pun = NULL;  
    testa->pun = p; //aggancio l'elemento  
    stampalista(testa);  
}
```

**Vedi programma** *stampa\_elem2\_sol.cc*

# Inserimento di un elemento: la primitiva `insert_elem`

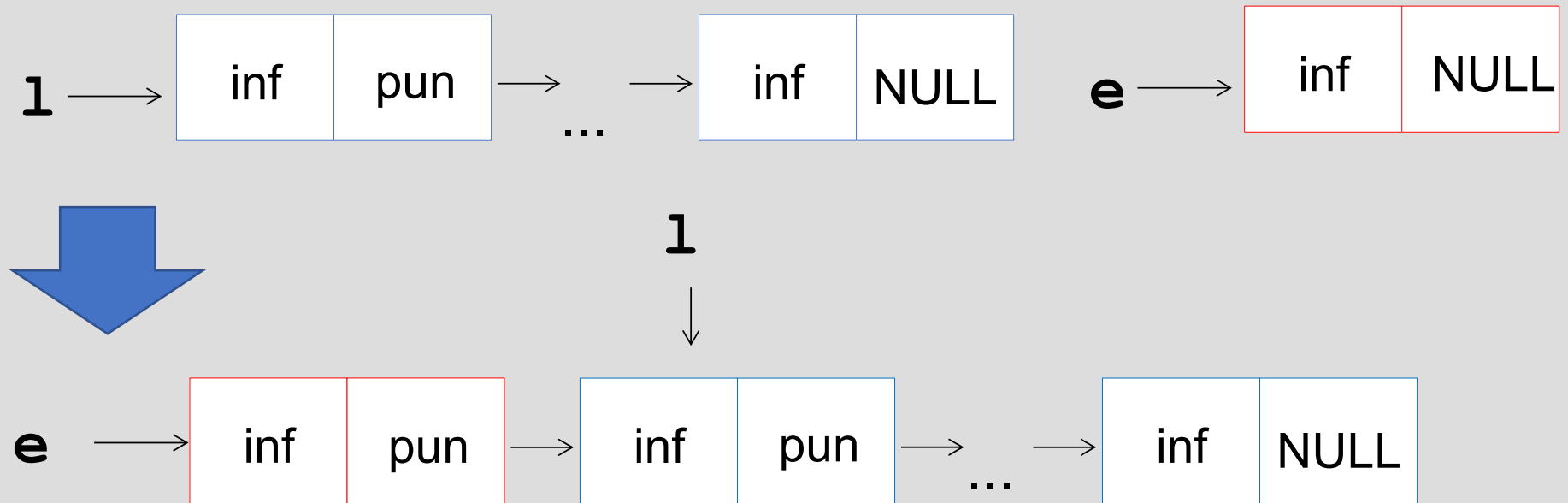
`lista insert_elem(lista l, elem* e)`

Funzione che aggiunge `e` a `l` e restituisce la lista  
aggiornata

- L'inserimento può avvenire
  - **in testa** modalità più semplice ed efficiente, gli elementi compariranno nella lista in **ordine inverso** rispetto all'ordine di inserimento
  - **In fondo (in coda)**: richiede di scorrere tutta la lista, gli elementi compariranno nella lista in **ordine diretto** rispetto all'ordine di inserimento
- Se non ci sono esigenze particolari, l'inserimento normalmente avviene in testa



# La primitiva `insert_elem`



```
lista insert_elem(lista l, elem* e) {  
    e->pun=l;  
    return e;  
}
```

# Inserimento di un elemento in una lista

- La funzione richiede un puntatore **elem\*** **e** all'elemento da aggiungere
- **e** deve essere allocato prima di passarlo a **insert\_elem**

```
    lista a;  
    elem* ele;  
...  
    ele=new elem;  
    cin>>ele->inf;  
    ele->pun = NULL;  
    a=insert_elem(a,ele) ;  
...
```

# Creazione di una lista

Si scriva la funzione **crealista**, che, utilizzando l'operatore **insert\_elem**, prende in ingresso un numero intero  $n$  e crea una lista di  $n$  elementi, inizializzati con valori inseriti dall'utente.

La funzione deve restituire il puntatore alla lista creata.

Infine, il programma stampa il contenuto della lista mediante la funzione **stampalista**

**Vedi programma da completare** *crea\_stampa\_lista.cc*

# Suggerimento

La funzione `lista_crealista(int n)`

- inizializza la lista a lista vuota
- per n volte
  - Crea un nuovo elemento elemento
  - Aggiorna il campo informazione con dati forniti dall'utente
  - Richiama la primitiva `insert_elem` per aggiungere l'elemento alla lista
- Restituisce la lista creata

**Vedi programma** `crea_stamp_lista_sol.cc`

## ESERCIZIO

Osservare lo stato della lista dopo ogni inserimento usando il debugger.  
Inserire i breakpoint nei punti opportuni.

# Cancellazione: la primitiva `delete_elem`

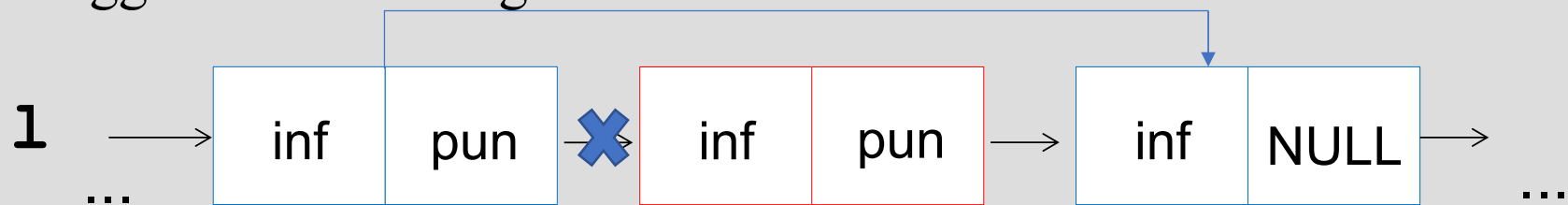
**`lista delete_elem(lista l, elem* e)`**

Funzione che cancella l'elemento **`e`** da **`l`** e restituisce la lista aggiornata

- Si assume che **`e`** sia presente in **`l`**, quindi che **`l`** non sia vuota

Due passi:

1. Aggiorno **`l`** «scollegando» **`e`** dalla lista

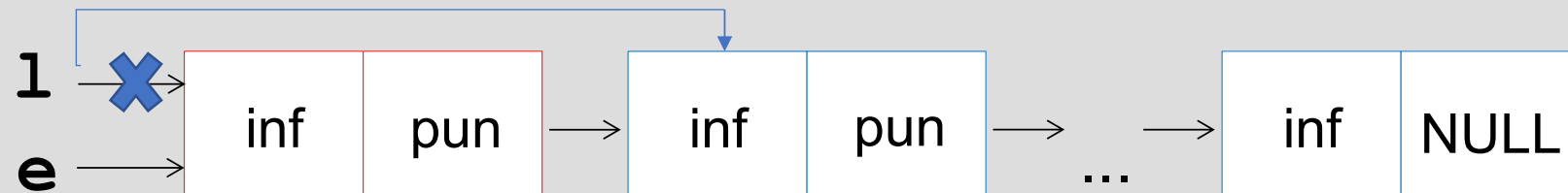


2. dealloco **`e`**

# Aggiornamento della lista

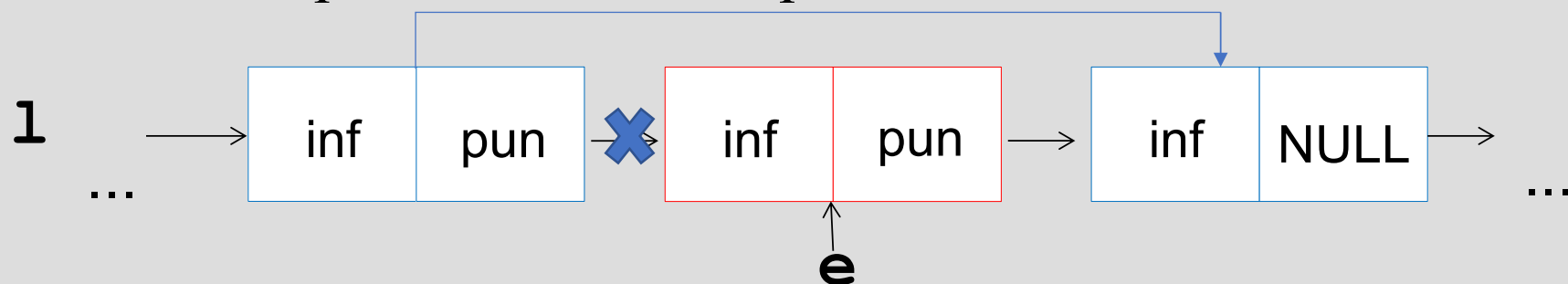
Per l'aggiornamento della lista, ci sono due casi:

- Se **e** coincide con la testa della lista **l**



è necessario modificare **l** affinché punti all'elemento puntato da **e**

- Altrimenti, è necessario aggiornare l'elemento che punta a **e** affinché punti all'elemento puntato da **e**



# Deallocazione di e:

## Uso di delete (2)

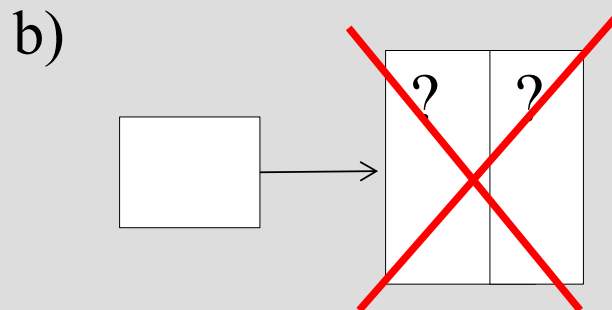
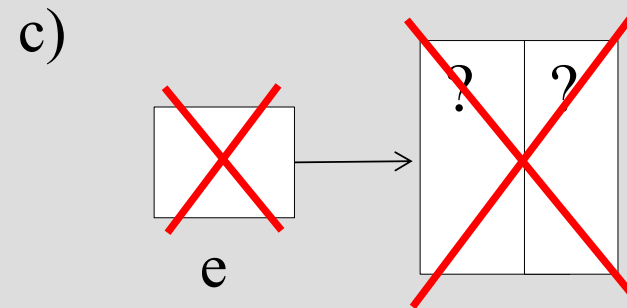
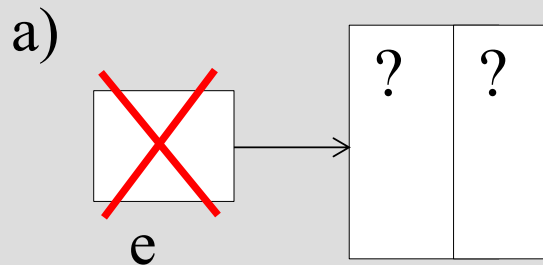
La seguente sequenza di istruzioni è corretta?

```
delete e;
```

*Sì, dealloca correttamente un oggetto  
allocato dinamicamente in memoria*

# Deallocazione di e: Uso di delete (2)

Cosa succede in memoria a seguito delle istruzioni precedenti?



*Si verifica il caso b) Il  
puntatore non viene  
deallocato*



# Uso di delete (3)

Nel caso:

```
int * e;
```

```
delete e ;
```

cosa viene in realtà passato alla delete?

*La delete si aspetta un indirizzo: alla delete viene passato il valore memorizzato dentro e, che viene interpretato come l'indirizzo dell'oggetto da deallocare dalla memoria dinamica*

# La primitiva delete\_elem

```
lista delete_elem(lista l, elem* e){
```

```
    if (l==e)  
        l=tail(l);
```

**e** è la testa della  
lista

```
    else{
```

```
        lista l1=l;  
        while (tail(l1)!=e)  
            l1=tail(l1);
```

Localizzo  
l'elemento che  
punta a **e**

```
        l1->pun=tail(e);}
```

Aggiorno  
l'elemento  
che punta a **e**

```
    delete e;  
    return l;
```

```
}
```

# La primitiva delete\_elem: OSSERVAZIONE

```
...  
lista l1=l;  
while (l1!=NULL && tail(l1)!=e)  
    l1=tail(l1);  
...
```

- Queste righe di codice localizzano l'elemento che punta a e
- Perché uso la variabile di appoggio l1? Perché non posso usare direttamente l?

```
...  
while (l!=NULL && tail(l)!=e)  
    l=tail(l);  
...
```

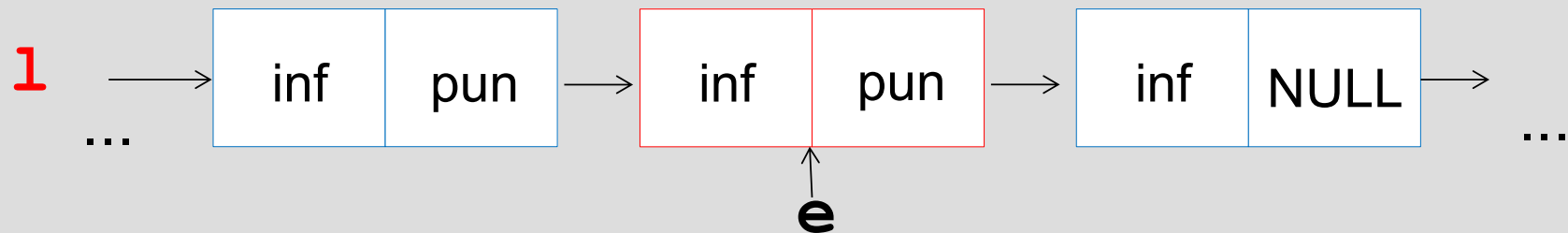
*Perché perdo il  
riferimento alla testa  
della lista*

# La primitiva delete\_elem: OSSERVAZIONE (cont.)

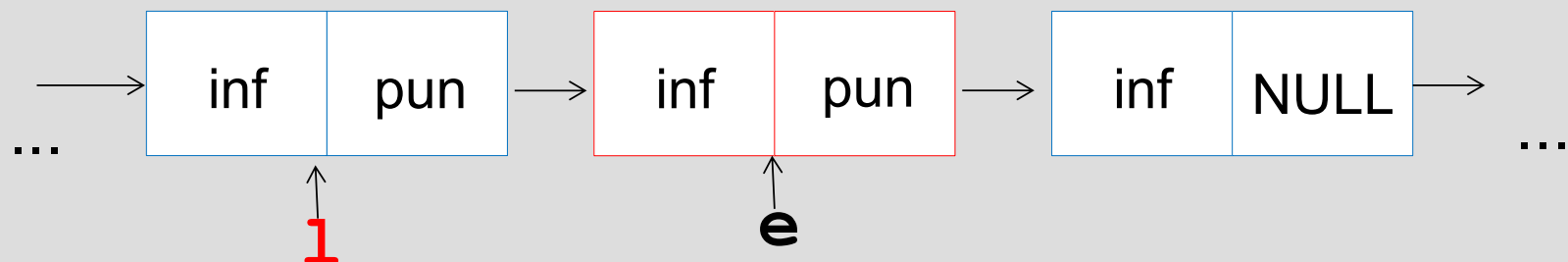
```
...  
while (l!=NULL && tail(l)!=e)  
    l=tail(l);  
...
```

*Perché perdo il  
riferimento alla testa  
della lista*

- Prima dell'ingresso nel ciclo



- Al termine del ciclo



# Eliminazione di una lista

Completare *crea\_stamp\_elim\_lista.cc* scrivendo la funzione **eliminalista**, che riceve come parametro il puntatore alla testa di una lista (passato attraverso un riferimento!) ed elimina la lista stessa usando la primitiva **delete\_elem**

## SUGGERIMENTO

Cancellare la testa della lista fino a quando la lista diventa vuota!

# Soluzione

```
void eliminalista(lista &testa)
{
    while (testa != NULL)
        testa=delete_elem(testa,testa);
}
```

**Vedi programma** *crea\_stampa\_elim\_lista\_sol.cc*

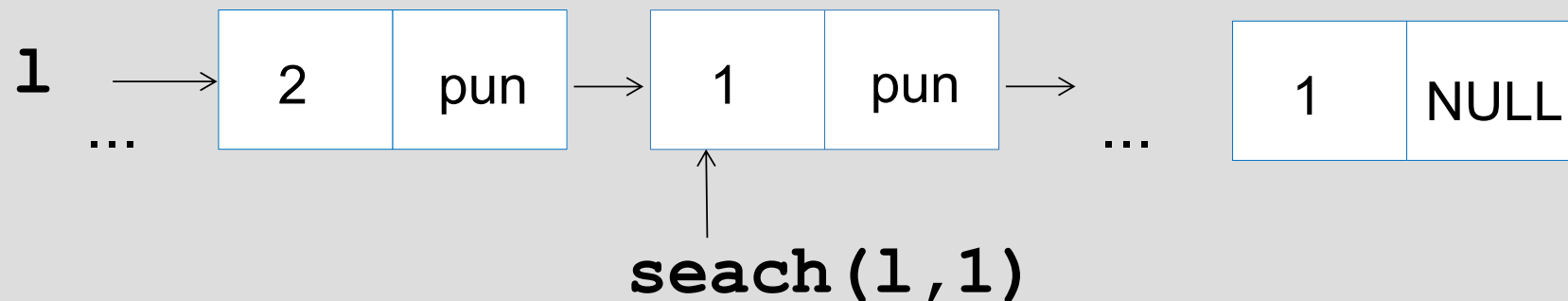
## ESERCIZIO

Osservare lo stato della lista dopo ogni cancellazione usando il debugger

# Ricerca di un elemento: la primitiva `search`

**`elem* search(lista l, int v)`**

Funzione che cerca nella lista **`l`** il valore **`v`** e restituisce il puntatore all'elemento che contiene la *prima occorrenza* di **`v`**, se esiste, **`NULL`**, altrimenti



**`elem* search(lista l, int v)`**

- Il tipo del valore cercato corrisponde al tipo del contenuto informativo di **`elem`**

# Algoritmo per la primitiva `search`

Scorro la lista `l` (attraverso le primitive `head` e `tail`)  
fino ad individuare l'elemento cercato (se esiste)

```
elem* search(lista l, int v){  
    while (l!=NULL)  
        if (head(l)==v)  
            return l;  
        else  
            l=tail(l);  
    return NULL;}
```



# Esercizio

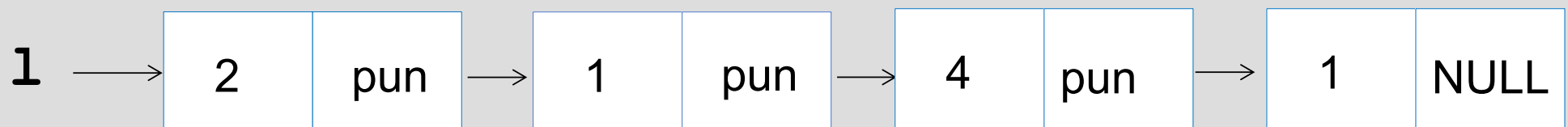
- Creare un progetto con Eclipse con il file sorgente `search.cc`
- Compilare e creare l'eseguibile del progetto
- Verificare il funzionamento della funzione **search** nei due possibili casi (valore presente e assente):
  - Inserendo i breakpoint nei punti opportuni
  - Inserendo i valori opportuni
- Quali valori assume il parametro formale **l** durante l'esecuzione della funzione nei due casi?

# Esercizio

## NUMERO DI OCCORRENZE DI UN VALORE

Scrivere la funzione `int conta(lista l, int v)`, che riceve come parametri il puntatore alla testa di una lista `l` e un valore da cercare `v` e restituisce il numero di occorrenze di `v` in `l`.

La funzione deve usare le primitive delle liste.



`conta(l, 1)` restituisce il valore 2

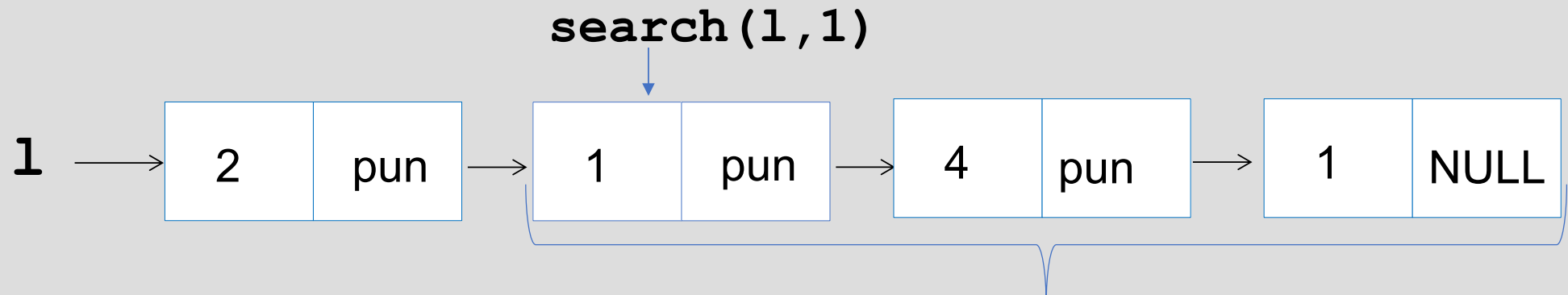
`conta(l, 6)` restituisce il valore 0

**Vedi programma** *conta.cc*

# Esercizio (cont.)

Quali primitive possiamo usare?

- **search**(**l**, **v**) restituisce l'elemento che contiene la prima occorrenza di **v**



Questo elemento è la testa di una lista che è una sottolista di **l**

- Possiamo re-iterare la ricerca sulla coda della sottolista (usando **tail**)
- Il processo continua fino a quando **search** restituisce **NULL**

# Soluzione

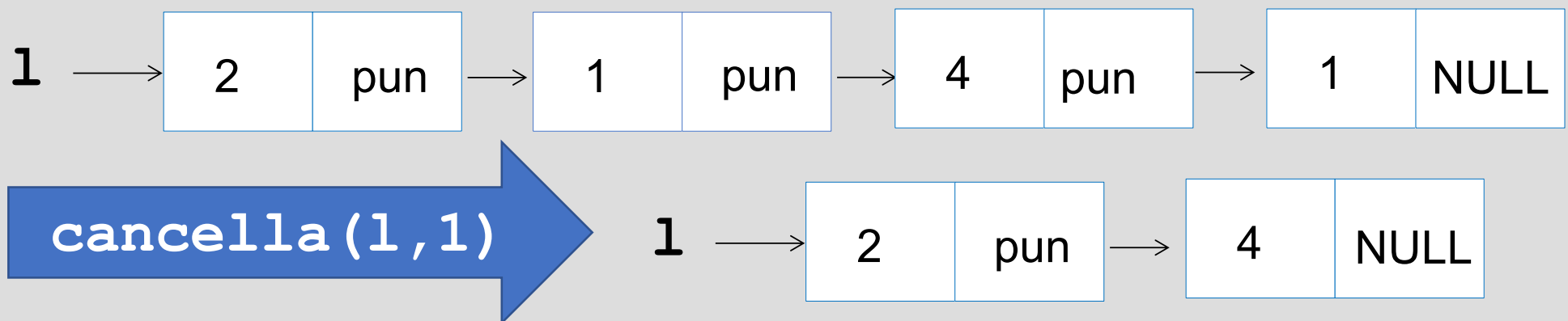
```
int conta(lista l, int v){  
    int occ = 0;  
    while((l=search(l,v))!=NULL){  
        l=tail(l);  
        occ++;  
    }  
    return occ;  
}
```

**Vedi programma** *conta\_sol.cc*

# Esercizio

## CANCELLAZIONE DI UN VALORE

Scrivere la funzione `lista cancella(lista l, int v)`, che riceve come parametri il puntatore alla testa di una lista `l` e un valore da cercare `v` e restituisce la lista priva di tutti gli elementi contenenti `v`



Vedi programma *cancella.cc*

# Esercizio (cont.)

Quali primitive possiamo usare?

- similmente a **conta** possiamo individuare ogni occorrenza di **v** usando la primitive **search**
- A differenza di **conta**, non dobbiamo contare il numero di occorrenze ma cancellare queste occorrenze
- La primitiva **delete\_elem** ci consente di cancellare gli elementi restituiti da **search**

# Soluzione

```
lista cancella(lista l, int v){  
    elem* e;  
    while((e=search(l,v))!=NULL)  
        l=delete_elem(l,e);  
    return l;  
}
```

**Vedi programma** *cancella\_sol.cc*

## OSSERVAZIONE

La funzione cancella è poco efficiente perché per eliminare un elemento scorre due volte la lista: una volta per localizzare l'elemento e l'altra per cancellarlo

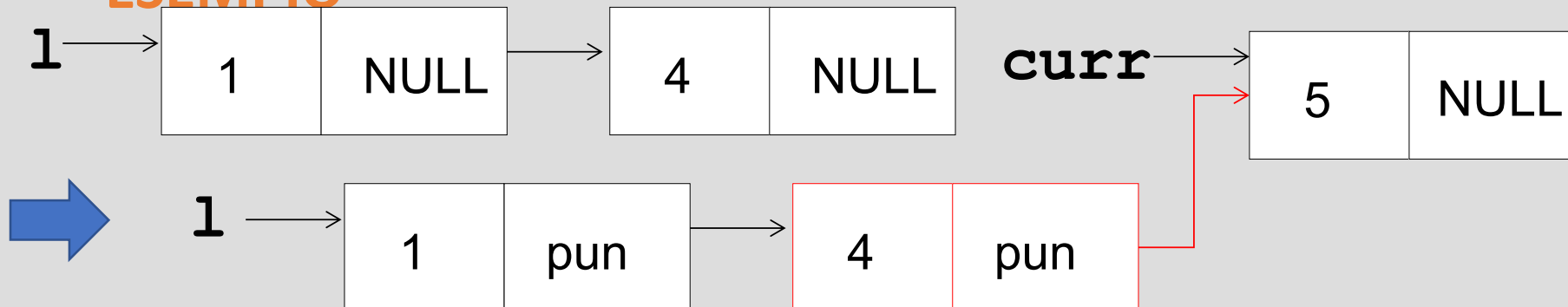
**ESERCIZIO NON RISOLTO** Riscrivere la funzione cancella affinché scorra una sola volta la lista usando le primitive head() e tail()

# La primitiva

## `lista copy(lista l1)`

- Inizializzo la lista `l` a NULL
- Per ogni valore in `l1`
  - Genero un nuovo elemento `curr`
  - Lo appendo alla lista `l`

### ESEMPIO



*Per appendere `curr` a `l`, devo usare un puntatore ausiliario che punti all'ultimo elemento inserito in `l`*



# La primitiva

## lista copy(lista l1)

```
lista copy(lista l1){  
    lista l=NULL;  
    elem* curr;
```

```
    elem* prev=NULL;
```

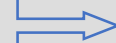


Puntatore ausiliario: punta all'ultimo elemento inserito

```
    while(l1!=NULL){  
        curr = new elem ;  
        curr->inf = head(l1);  
        curr->pun=NULL;  
        if(prev==NULL) /* sto creando la testa */  
            l=curr;
```

```
    else
```

```
        prev->pun=curr;
```



Inserimento in fondo alla lista

```
        prev=curr;
```



Aggiornamento di prev

```
        l1=tail(l1);
```

```
    }
```

```
    return l;
```

```
}
```

# Ricapitolando: il tipo di dato **LISTA**

```
struct elem {  
    int inf;        // contenuto informativo  
    elem* pun;  
};  
typedef elem* lista;
```

## **PRIMITIVE**

- Primitive per accedere alla lista: **head** e **tail**
- Primitive per aggiornare la lista: **insert\_elem** e **delete\_elem**
- Primitiva per la ricerca di un valore informativo: **search**
- Primitiva per la copia di una lista: **copy**

## **NOTE**

- Il contenuto informativo può essere definito a piacere, in base alle esigenze applicative (reali, stringhe, struct,...)
- Il tipo del contenuto informativo è usato nelle primitive **head** e **search** che devono quindi essere cambiate se cambia il tipo

# Esercizi non risolti

- Implementare la primitiva `delete_tail_elem()` che cancella l'ultimo elemento della lista usando la primitiva `tail()`
- Scrivere la primitiva `tail_insert_elem()` che inserisce l'elemento in fondo alla lista invece che in testa
- Date due liste che implementano insieme di valori interi, implementare le funzioni insiemistiche `union`, `intersect` e `difference` che implementano unione insiemistica, intersezione insiemistica e differenza insiemistica, rispettivamente.  
Ogni funzione deve restituire una nuova lista quale risultato dell'operazione. Le funzioni devono far uso delle primitive dei tipi lista per scorrere le due liste e per creare la nuova lista.