

# Fondamenti di Informatica

Allievi Automatici

A.A. 2015-16

Memoria Dinamica

# Allocazione della memoria

- In C i dati hanno una **dimensione** nota a **tempo di compilazione** ( `sizeof(...)` )
  - La quantità di memoria necessaria per eseguire una funzione è nota al compilatore
    - dimensione di un record di attivazione
  - Non si conosce, però, il numero di "esemplari" da allocare (esempio: ricorsione)
- Come si gestiscono i dati la cui dimensione è nota solo a **tempo di esecuzione**?
  - Es: invertiamo una sequenza di interi letti da stdin

# Soluzione con array

- Si **prealloca** un'area dati **sovradimensionata** rispetto all'effettivo utilizzo
  - Si tiene traccia di quanta parte di essa è effettivamente occupata
    - Nell'esempio della sequenza, l'indice che progredisce "conta" i valori validi inseriti
- **Problema: grande spreco di memoria**
- **Soluzione: MEMORIA DINAMICA**

# Memoria dinamica: motivazioni

- Dimensionamento "fisso" iniziale (ad esempio di array) – problemi tipici:
  - Spreco di memoria se a runtime i dati sono pochi
  - Violazione di memoria se i dati sono più del previsto
    - Un accesso oltre il limite dell'array ha effetti imprevedibili
  - Spreco di tempo per *ricompattare/spostare* i dati
    - Cancellazione di un elemento intermedio in un array ordinato
      - occorre far scorrere "indietro" tutti gli elementi successivi
    - Inserimento di un elemento intermedio in un array ordinato
      - occorre far scorrere "in avanti" i dati per creare spazio

# Variabili statiche, automatiche, dinamiche

- *Statiche*
  - *allocate prima dell'esecuzione del programma*
  - *restano allocate per tutta l'esecuzione*
- *Automatiche*
  - *allocate e deallocate automaticamente*
  - *gestione della memoria a stack (LIFO)*
- **Dinamiche**
  - **Allocate e deallocate esplicitamente a run-time dal programma (= dal programmatore)**
  - **Accessibili solo tramite puntatori**
  - **Referenziabili "da ogni ambiente"**
    - A patto di disporre di un puntatore che punti ad esse

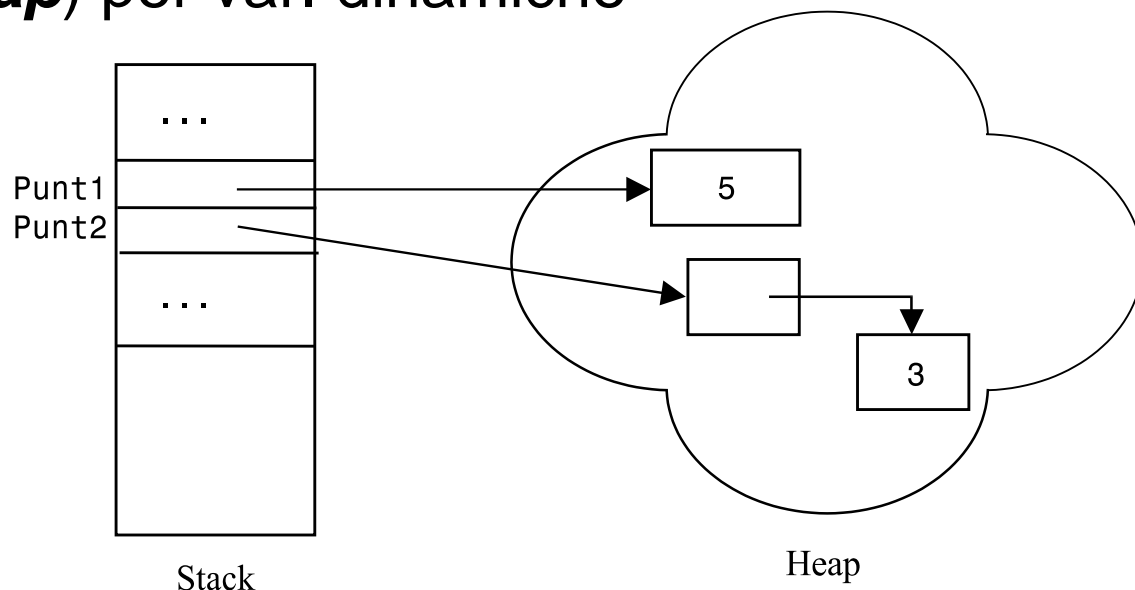
# Gestione della memoria

La memoria riservata ai dati del programma è partizionata in due "zone"

- pila (***stack***) per var. statiche e automatiche
- mucchio (***heap***) per var. dinamiche

Esempio

```
int * Punt1;  
int ** Punt2;
```



# Allocazione e Rilascio di memoria

- Apposite funzioni definite nella standard library `<stdlib.h>` si occupano della gestione della memoria dinamica:
  - `malloc(...)`      *memory **allocation*** - per l'allocazione
  - `free(...)`      per il rilascio
- Il programma le può invocare in qualsiasi momento per agire sullo heap

# Puntatori e `<stdlib.h>`

- La libreria `stdlib.h` contiene:
  - I prototipi delle funzioni di allocazione dinamica della memoria
    - `malloc(...)`
    - `free(...)`
  - La dichiarazione della costante `NULL`
    - puntatore nullo
    - non punta ad alcuna area significativa di memoria
      - ANSI impone che rappresenti il valore 0



# Allocazione: malloc()

- La funzione **malloc(...)**
  - Prototipo: `void * malloc( int );`
  - **Riceve** come parametro il numero di **byte** da allocare
    - Normalmente si usa la funzione **sizeof()** per indicare la dimensione dei dati da allocare
  - **Restituisce** un puntatore di tipo **void \***
    - il puntatore di tipo **void \*** può essere poi assegnato a qualsiasi altro puntatore per usare la nuova variabile
    - se non c'è più memoria disponibile (perché lo heap è già pieno), malloc() restituisce **NULL**

# Allocazione: malloc()

```
typedef ...any definition... TipoDato;  
typedef TipoDato * PTD;  
PTD ref;  
...  
ref = (PTD) malloc( sizeof(TipoDato) );
```

- **Alloca** nello heap **una variabile dinamica** (grande quanto un **TipoDato**) e restituisce l'indirizzo della prima cella occupata da tale variabile
  - **LA VARIABILE DI PER SÉ È ANONIMA!!!**
  - Ovviamente ref perde il valore precedente, e punta alla nuova variabile, che è accessibile per dereferenziazione ( \*ref )

# Attenzione

- Lo spazio allocato da malloc() è per la nuova variabile, di tipo TipoDato
- Non è per il puntatore ref, che già esisteva!
  - ref è una variabile **STATICA**
- **ref = (PTD) malloc( sizeof(TipoDato) );**
  - Il cast esplicito specifica al compilatore che il programmatore è consapevole che il puntatore è convertito **da** void \* **a** PTD (cioè a puntatore a TipoDato)
  - Nel seguito a volte sarà omesso (come nel libro di testo)
    - Si tenga comunque presente che alcune piattaforme non segnalano nulla, altre segnalano un warning, altre ancora ne considerano l'omissione un vero e proprio errore

**CAST ESPLICITO**

# Deallocazione: free()

- La funzione **free()**

- Prototipo: `void free( void * );`
- Libera la memoria allocata tramite la **malloc**, che dopo l'esecuzione è pronta ad essere riusata
- Riceve un puntatore **void \*** come argomento
- `free( ref );`

- N.B.: non serve specificare la dimensione in byte, che è derivabile automaticamente

# malloc() e free()

- Esempio: allocare una var. dinamica di tipo char, assegnarle 'a', stamparla e infine deallocarla

```
char * ptr;  
ptr = (char *) malloc( sizeof(char) );  
*ptr = 'a';  
printf("Carattere: %c\n", *ptr);  
free( ptr );
```

- Attenzione:
  - ptr **NON** è eliminato, e può essere riutato per una nuova malloc
    - INFATTI ptr E' UNA VARIABILE STATICA, QUINDI NON DEALLOCABILE

# Confrontare ...

```
char c = 'a'; /* varibile char STATICA */  
printf("Carattere: %c\n", c);
```

con

```
char c; /* varibile char STATICA */  
void * ptr; /* puntatore "buono per tutti gli usi" */  
ptr = &c; *ptr = 'a';  
printf("Carattere: %c\n", *ptr);
```

e

```
void * ptr; /* puntatore "buono per tutti gli usi" */  
ptr = malloc( sizeof(char) ); /* var. char DINAMICA */  
*ptr = 'a';  
printf("Carattere: %c\n", *ptr);  
free( ptr );
```

# A volte ritornano: inversione di una sequenza di interi

- Avevamo imparato studiando questo problema che l'uso degli array può semplificare assai la scrittura dei programmi
- Restava però “irrisolto” il problema di dover scegliere a priori quante variabili allocare (la dimensione dell'array)
- Ora possiamo pensare di allocare un array **dinamico** “piccolo” e sostituirlo con uno più grande solo se necessario
- ATTENZIONE:
  - La malloc() alloca blocchi contigui di memoria ad ogni invocazione, ma invocazioni diverse restituiscono blocchi totalmente scorrelati
  - Quando un vettore si riempie, quindi, occorre ricopiare nel nuovo vettore la sequenza memorizzata fino a quel punto

```
#define SENTINELLA -1
#define DIM_INIZIALE 100
#define INCREMENTO 50
```

```
int main() { int n, *v, lung_max = DIM_INIZIALE, i=0;
             v = (int *) malloc(lung_max*sizeof(int));
             scanf("%d", &n);
             while( n != SENTINELLA ) {
                 v[i++] = n;
                 if( i == lung_max ) {
                     v = replace( v, lung_max, INCREMENTO );
                     lung_max += INCREMENTO;
                 }
                 scanf("%d", &n);
             }
             printReverse(v, i-1);
             return 0; }
```



```
int * replace( int * v, int l_max, int inc ) {  
    int * vet, i;  
    vet = (int *) malloc( sizeof(int)*(l_max+inc) );  
    for( i=0; i<l_max; i++ )  
        vet[i] = v[i];  
    free( v );  
    return vet;  
}
```

```
void printReverse( int v[], int len ) {  
    while( i>0 )  
        printf("%d", v[i--]);  
}
```

# A volte ritornano: inversione di una sequenza di interi

- Ora sappiamo usare (quasi) solo la memoria realmente necessaria per la memorizzazione
- Lo schema di incremento è piuttosto rigido
  - Si può migliorare per cercare di limitare il numero di ricopiature, man mano che la sequenza si allunga
    - Incrementare / raddoppiare l'incremento a ogni incremento
    - Incrementare ogni volta di una percentuale (fissa o variabile)
- *Possiamo ancora migliorare questa soluzione:*  
impareremo come allocare le variabili una alla volta, senza ricopiare mai la sequenza

# Produzione di "spazzatura"

- La memoria allocata dinamicamente può diventare **inaccessibile** se nessun puntatore punta più ad essa
  - Risulta **sprecata**, e non è recuperabile
    - Per invocare free() è necessario un puntatore
  - È "spazzatura" (**garbage**) che non si può smaltire

- Esempio banale

```
TipoDato *P, *Q;
```

```
P = (TipoDato *) malloc(sizeof(TipoDato));
```

```
Q = (TipoDato *) malloc(sizeof(TipoDato));
```

```
P = Q;  /* la variabile che era puntata da P è garbage */
```

# Puntatori "ciondolanti"

- Detti abitualmente *dangling references*
- Sono puntatori a zone di memoria deallocate (→ a variabili dinamiche "non più esistenti")
  - $P = Q$ ;
  - `free(Q);`    */\* ora accedere a \*P causa un errore \*/*
- Sono **più gravi** della produzione di garbage: portano a veri e propri errori
  - Alcuni linguaggi (**Java!**) non hanno una operazione `free()`, ma un ***garbage collector***
    - Un componente della macchina astratta che trova e riutilizza la memoria inaccessibile (non più referenziata)

# Un intermezzo: con i puntatori...

- **...è possibile programmare molto male**
  - in modo "criptico"
  - generando effetti difficili da "tracciare"
  - in modo che il funzionamento del programma dipenda da come uno specifico sistema gestisce la memoria
    - Lo stesso programma, se scritto "male", può funzionare in modo diverso su macchine diverse
- Si possono fare danni considerevoli
  - Non sempre la macchina reale si comporta come il modello suggerirebbe
- Vediamo due "esempi" di cosa "si riesce" a fare..

# Puntatori a variabili automatiche

```
#include <stdio.h>
```

```
int * p;
```

```
void boh() {  
    int x = 55;  
    p = &x;  
}
```

```
int main() {  
    int x = 1;  
    boh();  
    printf("risultato= %d", *p);  
    return 0;  
}
```

p è dangling dopo  
la chiamata di boh  
  
in pratica, però, stampa 55  
**Perché?**

22

SONO ESEMPI DI QUELLO CHE **NON** SI  
DEVE FARE CON I PUNTATORI  
(indipendentemente dalla memoria dinamica)

```
#include <stdio.h>
```

```
int * p;
```

```
void boh() {    int x = 55;  
              p = &x; }
```

```
void bohboh() { int y = 100; }
```

```
int main() {  
    int x = 1;  
    p = &x;  
    boh();  
    bohboh();  
    printf("risultato= %d\n", *p);  
    return 0;  
}
```

## Che cosa fa?

p è assegnato all'indirizzo della x "statica" del main, che ha valore 1. Poi la chiamata di boh() lo riassegna alla x "automatica" (p è globale!)

Poi boh termina, e il suo record di attivazione è sovrascritto da quello di bohboh, che è strutturalmente uguale

Quindi la y "cade" dove prima c'era la x, e la printf **stampa** il valore **100**

SONO ESEMPI DI QUELLO CHE **NON** SI  
DEVE FARE CON I PUNTATORI  
(indipendentemente dalla memoria dinamica)

# Avvertimento

- Puntatori e variabili dinamiche portano a programmare a basso livello e "pericolosamente"
- Sono da usare con parsimonia, solo quando è strettamente necessario, e cioè:
  - per passare parametri per indirizzo
  - per costruire strutture dati complesse
    - Liste, alberi, grafi, ... (che studiamo subito)
  - In pochi altri casi di uso della mem. dinamica



# Strutture dati dinamiche

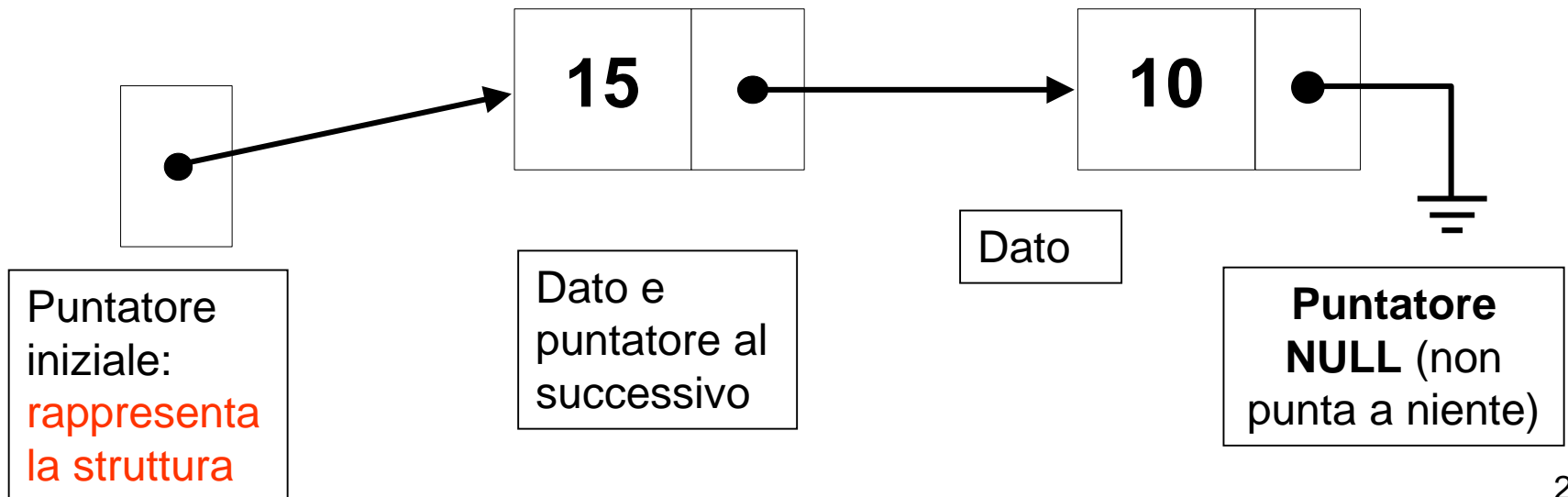
**Crescono e decrescono** durante l'esecuzione:

- ***Lista concatenata (linked list)***
  - Inserimenti/cancellazioni facili in qualsiasi punto
- ***Pila (stack)***
  - Inserimenti/cancellazioni solo in cima (accesso **LIFO**)
- ***Coda (queue)***
  - inserimenti "in coda" e cancellazioni "in testa" (**FIFO**)
- ***Albero binario (binary tree)***
  - ricerca e ordinamento veloce di dati
  - rimozione efficiente dei duplicati

# Strutture dati ricorsive

(o auto-referenziali)

- **Strutture con puntatori a strutture dello stesso tipo**
- Si possono **concatenare** per ottenere strutture dati utili come: liste, code, pile, alberi, ...
- "terminano" con **NULL**



# La Lista

- Composta da ***elementi*** allocati dinamicamente, il cui numero cambia durante l'esecuzione
- Si accede agli elementi tramite puntatori
- Ogni elemento contiene un **puntatore al prossimo** elemento della lista
  - Il primo deve essere puntato "a parte"
    - Non ha un precedente
  - L'ultimo non deve puntare "a niente"
    - Non ha un successivo

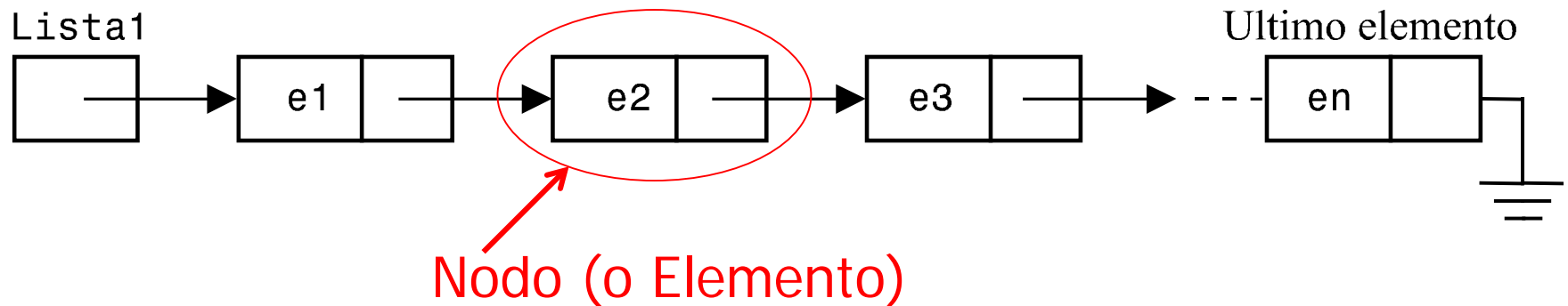
# La Lista

**Inizio** della lista:

- Variabile di tipo *puntatore a elemento della lista*

**Fine** della lista:

- Puntatore nell'ultimo elemento vale NULL
- NULL è interpretabile anche come "lista vuota"



# Strutture dati ricorsive (dichiaraz.)

- Si definiscono il tipo del nodo...

```
typedef struct EL {  
    TipoElemento info;  
    struct EL * prox;  
} ElemLista;
```

Notare la sintassi!

- ...e il tipo del puntatore

```
typedef ElemLista * ListaDiElem;
```

Definizione Ricorsiva!!!

# Strutture dati ricorsive (variante)

```
struct El {  
    TipoElemento dato;  
    struct El * prox;  
};  
typedef struct El ElemLista;  
typedef struct El * ListaDiElem;
```

Il puntatore **prox**:

- punta a un oggetto di tipo **struct El**
- si chiama ***link***
- lega oggetti di tipo **struct El** tra di loro

# Liste concatenate (linked lists)

- Lista concatenata:
  - Collezione lineare di oggetti di tipo auto-referenziale, chiamati **nodi**, collegati tramite puntatori (**link**)
  - Vi si accede mediante un puntatore al primo nodo della lista (la **testa** della lista)
  - Gli elementi successivi al primo (la **coda** della lista) si raggiungono attraversando i puntatori (link) da un oggetto all'altro
    - Osservazione utile in seguito: la coda di una lista è una lista
  - Il puntatore (link) contenuto **nell'ultimo** elemento ha valore **NULL**

# Liste concatenate (linked lists)

- Si usano (al posto degli array) quando:
  - Il numero degli elementi non è noto a priori, e/o
  - La lista deve essere mantenuta ordinata
- Al prezzo di una gestione un po' più complessa, risolviamo i problemi di "spreco" di spazio e di tempo descritti all'inizio
- **NOTA:** gli elementi di una lista **non** sono necessariamente **memorizzati in modo contiguo!**
  - I nodi sono anzi di solito "sparpagliati" nello heap, e i link li "cuciono" in una sequenza che dalla testa arriva all'ultimo nodo



# Modi di concatenare le liste

- *Liste **semplicemente** concatenate:*
  - Comincia con un puntatore al primo
  - Termina col puntatore nullo
  - Si attraversa solo in un solo verso (dalla testa fino in fondo)
- *Liste **semplicemente** concatenate **circolari**:*
  - Il puntatore contenuto nell'ultimo nodo punta di nuovo al primo
- *Liste **doppiamente** concatenate:*
  - Due puntatori di "inizio", uno al primo e uno all'ultimo elemento
  - Ogni nodo ha un puntatore "in avanti" e uno "indietro"
  - Permette l'attraversamento nelle due direzioni
- *Liste **doppiamente** concatenate **circolari**:*
  - Il puntatore "in avanti" dell'ultimo nodo punta al primo nodo
  - Il puntatore "indietro" del primo nodo punta all'ultimo nodo

# Creare un singolo nodo

```
typedef struct Nd {  
    int dato;  
    struct Nd * next;  
} Nodo;  
  
typedef Nodo * ptrNodo;  
  
ptrNodo ptr;                               /* puntatore a nodo */  
ptr = malloc(sizeof(Nodo)); /* crea nodo */  
ptr->dato = 10;      /* inizializza nodo (dato) */  
ptr->next = NULL;    /* inizializza nodo (link) */
```

# Creare una lista di **due** nodi

...

```
ptrNode Lista;    /* puntatore alla testa della lista */
ptrNode ptr;      /* puntatore ausiliario a nodo */
Lista = malloc(sizeof(Nodo));    /* crea 1° nodo */
Lista->dato = 10;    /* inizializza 1° nodo */
ptr = malloc(sizeof(Nodo));    /* crea 2° nodo */
ptr->dato = 20;    /* inizializza 2° nodo */
Lista->next = ptr;    /* collega il 1° al 2° */
ptr->next = NULL;    /* "chiusura" lista al 2° nodo */
```

...

# Creare una lista di **due** nodi (variante)

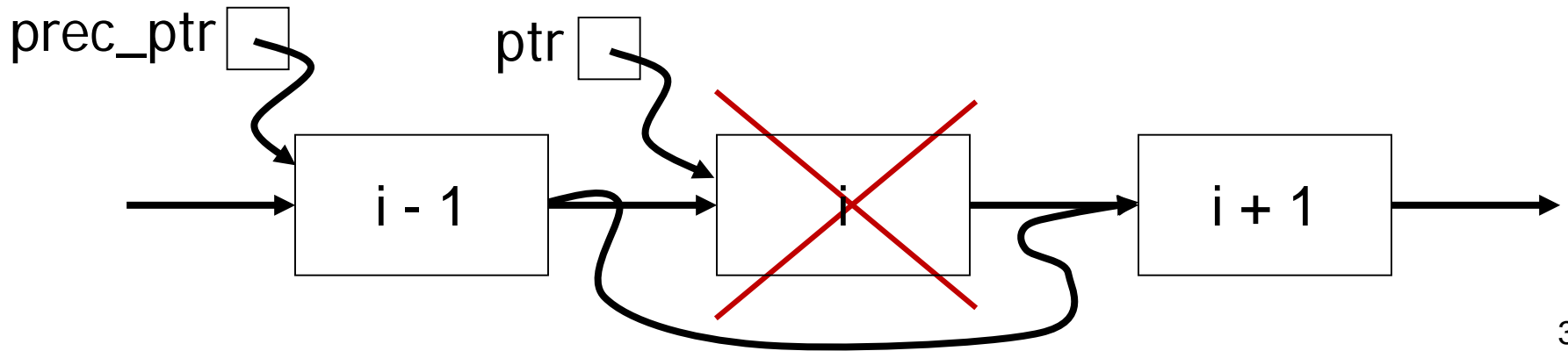
...

```
ptrNode Lista;  /* puntatore alla testa della lista */
Lista = malloc(sizeof(Nodo));      /* crea 1° nodo */
Lista->dato = 10;                  /* inizializza 1° nodo */
Lista->next = malloc(sizeof(Nodo);
    /* crea E ATTACCA il 2° nodo in coda al primo */
Lista->next->dato = 20;             /*inizializza 2° nodo */
Lista->next->next = NULL; /*"chiusura" al 2° nodo */
...
```

# Cancellare un nodo interno

...

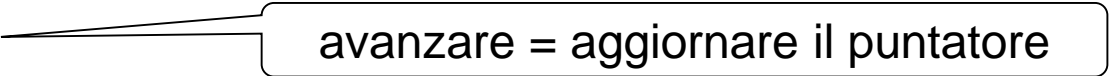
```
ptrNode ptr; /* puntatore al nodo i° da cancellare */
ptrNode prec_ptr; /* puntatore al nodo (i-1)° che
                  precede il nodo i° da cancellare */
... /* qui si inizializzano ptr e prec_ptr ... */
prec_ptr->next = ptr->next;
/* collega il nodo (i-1)° all' (i+1)°, saltando il nodo i° */
free (ptr); /* elimina il nodo i° */
```



# Cercare un nodo nella lista

```
int d;           /* il dato da cercare */
ptrNode Lista;  /* puntatore alla radice della lista */
ptrNode ptr;    /* puntatore ausiliario a nodo */
...            /* Lista e d sono inizializzati (omesso) */

ptr = Lista;
while( ptr != NULL && ptr->dato != d )
    /* entra nel ciclo se ptr NON punta al dato cercato */
    ptr = ptr->next;
/* all'uscita ptr vale NULL o punta al dato cercato */
```



---

```
int d;
ptrNode Lista, ptr;
...
for( ptr=Lista; ptr!=NULL && ptr->dato!=d; ptr=ptr->next )
    ;
/* Variante sintattica: con FOR invece che con WHILE */
```

# Lunghezza della lista

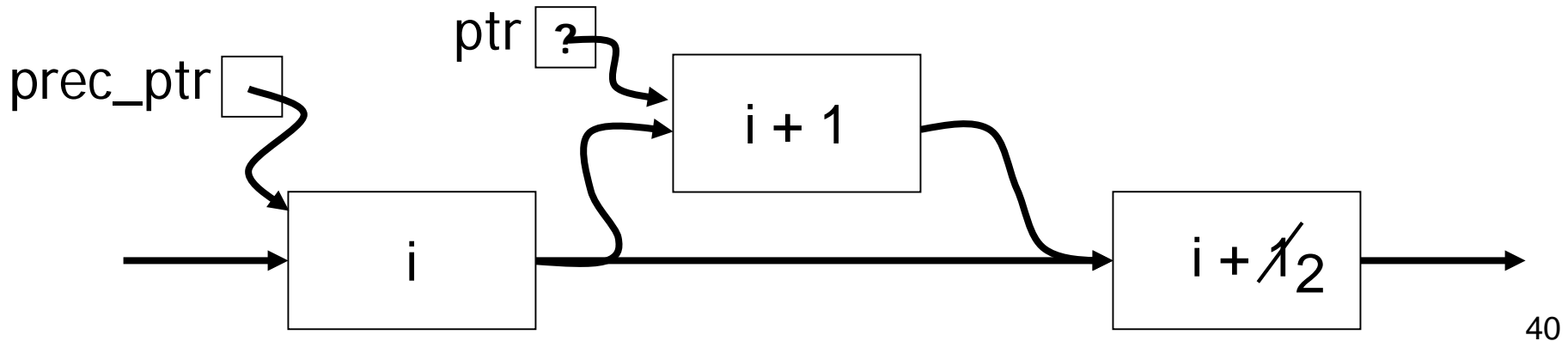
```
int numeronodi = 0;  
ptrNode Lista, ptr;  
Lista = ... /* costruzione della lista */  
for( ptr=Lista; ptr!=NULL; ptr=ptr->next )  
    numeronodi++;
```

- Per **CONTARE** i nodi dobbiamo necessariamente **SCANDIRE** la lista
- Anche per accedere a ogni nodo occorre partire dall'inizio, se si dispone soltanto del puntatore alla testa
- Non è possibile accedere alla lista se non scandendola in ordine, seguendo i puntatori

# Inserire un nodo interno alla lista

...

```
ptrNodp prec_ptr; /* puntatore al nodo  $i^{\text{esimo}}$ , che precede  
                  il nuovo nodo da inserire */  
  
ptrNodo ptr; /* puntatore ausiliario a nodo */  
... /* qui prec_ptr è inizializzato (trovare il nodo) */  
ptr = malloc (sizeof (Nodo));  
ptr->next = prec_ptr->next;  
prec_ptr->next = ptr;
```





# Gestione degli errori

...

```
ptrNode ptr;                                /* puntatore a nodo */
ptr = malloc(sizeof(Nodo));                  /* alloca un nodo */
if( ptr == NULL ) {
    printf("malloc: memoria insufficiente!\n");
} else {
    ptr->dato = 10;                           /* inizializza dato */
    ptr->next = NULL;                         /* inizializza link */
}
```

# Attenzione ....

```
...  
ptrNodo ptr;  
ptr = malloc(sizeof(Nodo));  
if( ptr == NULL ) {  
    ptr->dato = 10; /* ERRORE GRAVE !!!!!!! */  
    ...  
}
```

- SI STA TENTANDO DI APPLICARE L'OPERATORE "FRECCIA" A UN PUNTATORE NULL, OVVERO SI STA TENTANDO DI ACCEDERE A UN CAMPO DI UNA STRUCT INESISTENTE!
- Dereferenziare un puntatore a NULL genera un errore

# Le liste e la ricorsione...

- Che cos'è una lista (di nodi)??
- Dicesi **lista**:
  - Il **niente**, se è una lista vuota!
    - Questo è un caso veramente **base!!!**  
altrimenti...
  - Un **nodo**, seguito da... una **lista**!
    - Questo è un passo veramente... **induttivo!**

**UNA LISTA È UNA STRUTTURA RICORSIVA**

# Operazioni su liste (un "TDA"!)

*(su liste semplicemente concatenate)*

- Inizializzazione
- Inserimento
  - in prima posizione
  - in ultima posizione
  - ordinato
- Eliminazione

# Come facciamo?

- Le operazioni sono **tutte funzioni**
- Ricevono come parametro un puntatore al primo elemento (la *testa* della lista su cui operare)
- Le scriviamo in modo che, se la lista deve essere modificata, *restituiscano* al programma chiamante *un puntatore alla testa della lista modificata*
  - Questo impatta sul modo in cui faremo le chiamate
- Così tutti i parametri sono passati per valore

# Usiamo questa formulazione

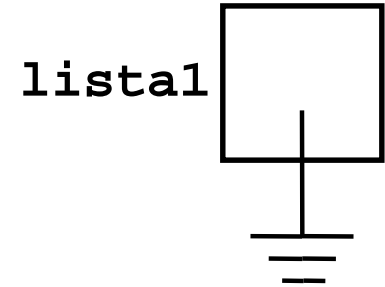
```
typedef struct EL {  
    TipoElemento info;  
    struct EL * prox;  
} ElemLista;
```

```
typedef ElemLista * ListaDiElem;
```



# Inizializzazione

```
ListaDiElem Inizializza( void ) {  
    return NULL;  
}
```



## Esempio di chiamata:

```
...  
ListaDiElem lista1;  
...  
lista1 = Inizializza();
```

## NOTA BENE

1. Se voglio di inizializzare **diversamente...** basta cambiare la funzione **Inizializza e non il resto del programma!**
2. Se Lista1 puntava a una lista, dopo Inizializza quella lista diventa **garbage**

# Controllo lista vuota

```
int ListaVuota( ListaDiElem lista ) {  
    if ( lista == NULL )  
        return 1;  
    else  
        return 0;  
}
```

*Oppure, più direttamente:*

```
int ListaVuota( ListaDiElem lista ) {  
    return lista == NULL;  
}
```



# Dimensione della lista (iter. e ric.)

```
int Dimensione(ListaDiElem lista) {  
    int count = 0;  
    while( ! ListaVuota(lista) ) {  
        lista = lista->prox;    /* "distruggiamo" il parametro */  
        count++;  
    }  
    return count;  
}
```

```
int DimensioneRic(ListaDiElem lista) {  
    if ( ListaVuota(lista) )  
        return 0;  
    return 1 + DimensioneRic( lista->prox );  
}
```

# Controllo presenza di un elemento

```
int VerificaPresenza (ListaDiElem lista, TipoElemento elem) {  
    ListaDiElem cursore;  
    if ( ! ListaVuota(lista) ) {  
        cursore = lista;    /* La lista non è vuota */  
        while( ! ListaVuota(cursore) ) {  
            if ( cursore->info == elem )  
                return 1;  
            cursore = cursore->prox;  
        }  
    }  
    return 0;    /* Falso: l'elemento Elem non c'è */  
}
```

# Versione ricorsiva !

```
int VerificaPresenza(ListaDiElem lista, TipoElemento elem) {  
    if( ListaVuota( lista ) )  
        return 0;  
    if( lista->info == elem )  
        return 1;  
    return VerificaPresenza( lista->prox, elem );  
}
```

# Inserimento in prima posizione

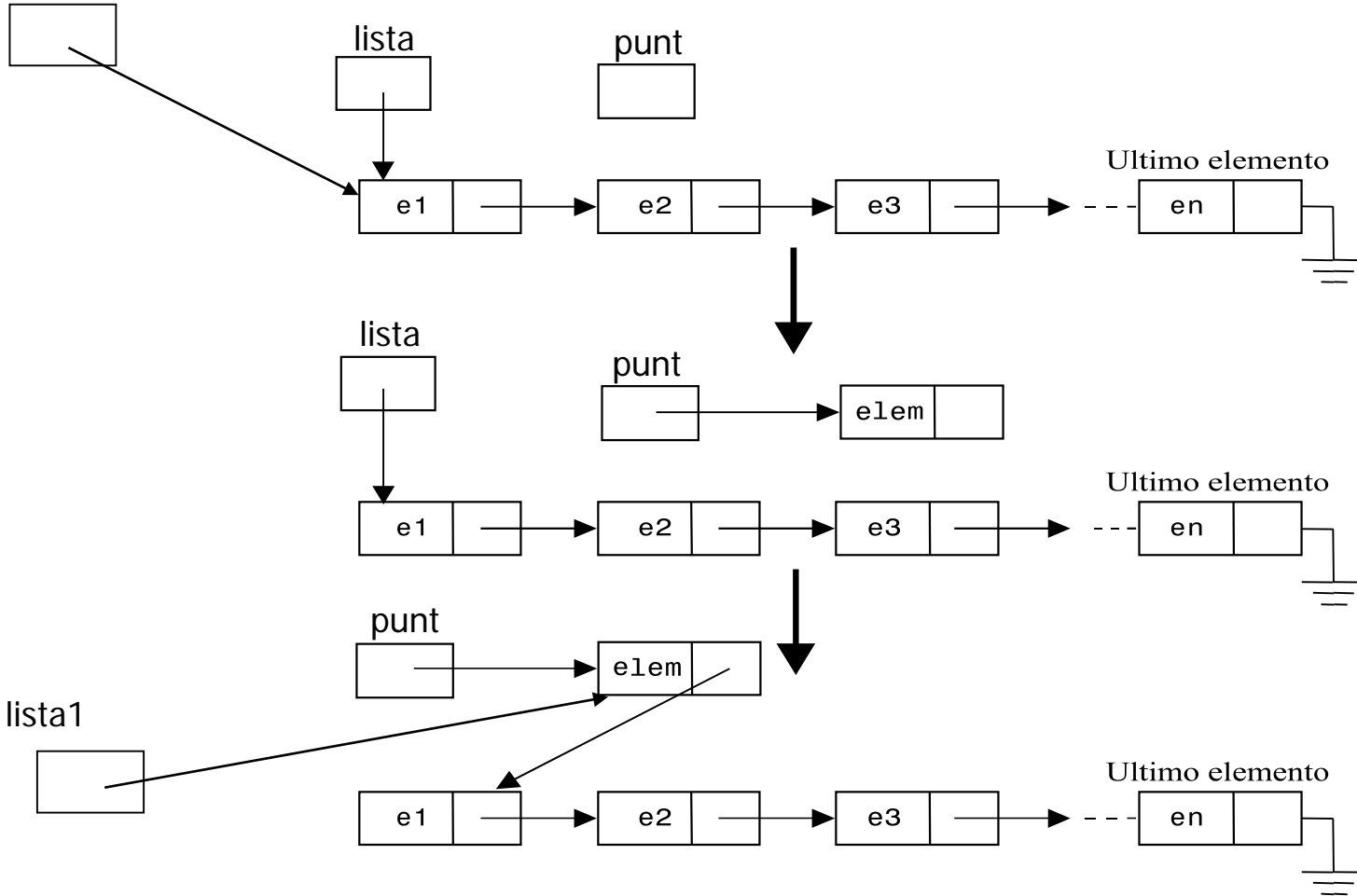
```
ListaDiElem InsInTesta ( ListaDiElem lista,  
                          TipoElemento elem ) {  
    ListaDiElem punt;  
    punt = (ListaDiElem) malloc(sizeof(ElemLista));  
    punt->info = elem;  
    punt->prox = lista;  
    return punt;  
}
```

Chiamata: **lista1** = InsInTesta( **lista1**, elemento );

**ATTENZIONE: l'inserimento modifica la lista**  
(non solo in quanto aggiunge un nodo, ma anche in quanto deve modificare  
il valore del puntatore al primo elemento *nell'ambiente del main*)

# Visualizzazione

lista1



# Inserimento in ultima posizione (iter.)

```
ListaDiElem InsInFondo( ListaDiElem lista, TipoElemento elem ) {  
    ListaDiElem punt, cur = lista;  
    punt = (ListaDiElem) malloc( sizeof(ElemLista) );  
    punt->prox = NULL;  
    punt->info = elem;                                /* Crea il nuovo nodo */  
    if ( ListaVuota(lista) )                          /* => punt è la nuova lista */  
        return punt;  
    else {  
        while( cur->prox != NULL )                  /* Trova l'ultimo nodo */  
            cur = cur->prox;  
        cur->prox = punt;                            /* Aggancio all'ultimo nodo */  
    }  
    return lista;  
}
```

Chiamata : **lista1** = InsInCoda( **lista1**, elemento );

# Inserimento in ultima posizione (ric.)

ListaDiElem **InsInFondo**( ListaDiElem lista, TipoElemento elem )

{

ListaDiElem punt;

if( ListaVuota(lista) ) {

punt = malloc( sizeof(ElemLista) );

punt->prox = NULL;

punt->info = elem;

return punt;

}

else { lista->prox = **InsInFondo**( lista->prox, elem );

return lista;

}

}

*Alternativa:* return InsInTesta( lista, elem );

Chiamata : **lista1** = InsInFondo( **lista1**, Elemento );

# Inserimento in ultima posizione (ric.)

```
ListaDiElem InsInFondo( ListaDiElem lista, TipoElemento elem ) {  
    if( ListaVuota(lista) )  
        return InsInTesta( lista, elem );  
    lista->prox = InsInFondo( lista->prox, elem );  
    return lista;  
}
```



# Inserimento in lista ordinata

```
ListaDiElem InsInOrd( ListaDiElem lista, TipoElemento elem ) {  
    ListaDiElem punt, puntCor = lista, puntPrec = NULL;  
    while ( puntCor != NULL && elem > puntCor->info ) {  
        puntPrec = puntCor;  
        puntCor = puntCor->prox;  
    }  
    punt = (ListaDiElem) malloc(sizeof(ElemLista));  
    punt->info = elem;  
    punt->prox = PuntCor;  
    if ( puntPrec != NULL ) {          /* Inserimento interno alla lista */  
        puntPrec->prox = punt;  
        return lista;  
    } else                             /* Inserimento in testa alla lista */  
        return punt;  
}
```

Chiamata : **lista1** = InsInOrd( **lista1**, elemento );

57

**ESERCIZIO** : scriverne una versione ricorsiva

# Una riflessione sulle liste ordinate

- Se consideriamo una lista inizialmente vuota e operiamo sempre e solo inserimenti ordinati...
  - In ogni momento la lista sarà ordinata
  - Questa assunzione può essere sfruttata
- Ma...
  - Se anche una sola volta facciamo un inserimento in testa o in coda
  - Se la lista inizialmente non è vuota
  - ...
- Allora (*nel caso più generale*)
  - Non vale più l'assunzione che la lista sia ordinata
  - L'effetto di "InsInOrd" non è nemmeno ben definito !

# Cancellazione di un elemento

```
ListaDiElem Cancella( ListaDiElem lista, TipoElemento elem ) {  
    ListaDiElem puntTemp;  
    if( ! ListaVuota(lista) )  
        if( lista->info == elem ) {  
            puntTemp = lista->prox;  
            free( lista );  
            return puntTemp;  
        }  
    else  
        lista->prox = Cancella( lista->prox, elem );  
    return lista;  
}
```

*Che cosa succede se  
nella lista ci sono  
valori duplicati?*

Chiamata : **lista1** = Cancella( **lista1**, elemento );

# Variante: elimina *tutte* le occorrenze

```
ListaDiElem Cancella( ListaDiElem lista, TipoElemento elem ) {  
    ListaDiElem puntTemp;  
    if( ! ListaVuota(lista) )  
        if( lista->info == elem ) {  
            puntTemp = lista->prox;  
            free( lista );  
            return Cancella(PuntTemp, Elem);  
        }  
        else  
            lista->prox = Cancella( lista->prox, elem );  
    return lista;  
}
```

# Deallocare completamente la lista

```
void DistruggiLista( ListaDiElem lista ) {  
    ListaDiElem temp;  
    while( lista != NULL ) {  
        temp = lista->prox;  
        free( lista );  
        lista = temp;  
    } }
```

```
void DistruggiListaRic( ListaDiElem lista ) {  
    if ( ! ListaVuota(lista) )  
        DistruggiListaRic( lista->prox );  
    free( lista );  
}
```

# Visualizzare la lista

```
void VisualizzaLista( ListaDiElem lista ) {  
    if ( ListaVuota(lista) )  
        printf(" ---| \n");  
    else {  
        printf(" %d\n ---> ", lista->info);  
        VisualizzaLista( lista->prox );  
    }  
}
```

*È un po' "brutto" il rendering dell'ultimo elemento...  
Esercizio: migliorarlo*

# A volte ritornano: inversione di una sequenza di interi

- Utilizzando una lista, possiamo memorizzare la sequenza allocando un nodo per ogni intero
- Dove inseriamo i nodi via via che leggiamo gli interi?
  - In coda? (ultima posizione)
    - Ma per generare la sequenza invertita....
  - In testa? (prima posizione)
    - Infatti per generare la sequenza invertita....

```
#define SENTINELLA -1
```

```
typedef int TipoElemento;
```

```
int main() {
```

```
    int n;
```

```
    ListaDiElem lista = Inizializza();
```

```
    scanf("%d", &n);
```

```
    while( n != SENTINELLA ) {
```

```
        lista = InsInTesta( lista, n );
```

```
        scanf("%d", &n);
```

```
    }
```

```
    VisualizzaLista(lista);
```

```
    return 0;
```

```
}
```

*Questo è un programma che, mentre acquisisce la sequenza, ha l'accortezza di memorizzarla "al contrario"*

*Possiamo sfruttare il principio per una funzione che realizzi l'inversione di una lista data?*



# Reverse di lista

```
ListaDiElem Reverse1( ListaDiElem lista, int keepSource ) {  
    ListaDiElem temp = Inizializza(), curr = lista;  
    while( ! ListaVuota(curr) )  
        temp = InsInTesta( temp, curr->info );  
        curr = curr->prox;  
}  
if( ! keepSource )  
    DistruggiLista( lista );  
return temp;  
}
```

*Questa versione **alloca**, un nodo alla volta, una **nuova lista** ricopiando via via i valori del campo info nei nuovi nodi.*

*Alla fine, si può deallocare la lista originale o conservarla, in base alla scelta effettuata dal programma chiamante.*

Chiamate: ListaDiElem s1, s2, s3;  
          s1 = Reverse1( s1, **0** );   s2 = Reverse1( s3, **1** );

# Reverse di lista

```
ListaDiElem Reverse2( ListaDiElem lista ) {  
    ListaDiElem temp, prec = NULL;  
    if( ! ListaVuota(lista) ) {  
        while( lista->prox != NULL ) {  
            temp = prec;  
            prec = lista;  
            lista = lista->prox;  
            prec->prox = temp;  
        }  
        lista->prox = prec;  
    }  
    return lista;  
}
```

*Questa versione **riusa** i nodi della lista passata come parametro, e li "rimonta" in ordine inverso*

# Inversione RICORSIVA...

- Se la lista ha 0 o 1 elementi, allora è pari alla sua inversa (e la restituiamo inalterata)
- Diversamente... **supponiamo** di saper invertire la coda (riduciamo il problema da "N" a "N-1"!!!)
  - 1–2–3–4–5–6–\
  - 1     6–5–4–3–2–\
  - Dobbiamo inserire il primo elemento in fondo alla coda invertita
  - Scriviamo una versione che sfrutti bene i puntatori...
    - **Prima** della chiamata ricorsiva possiamo mettere da parte un puntatore al **secondo** elemento [2], confidando che dopo l'inversione esso [2] sarà diventato l'**ultimo** elemento della "coda invertita", e attaccargli (in coda) il primo elemento [1]

```

ListaDiElem ReverseRic( ListaDiElem lista ) {
    ListaDiElem p, ris;
    if ( ListaVuota(lista) || ListaVuota(lista->prox) )
        return lista;
    else {
        p = lista->prox;
        ris = ReverseRic( p );
        p->prox = lista;
        lista->prox = NULL;
        return ris;
    }
}

```

# Liste e array

- Quando si deve operare su una lista di elementi di dimensione ignota
  - se si usa un array
    - occorre fissare una dimensione massima
    - si spreca la memoria non usata
    - ...ma la gestione è semplice
  - se si usa una lista con puntatori
    - vale **esattamente** il viceversa !
      - Si usa solamente la memoria strettamente necessaria, ma la sua gestione può essere complicata

# A volte ritornano (*poi però basta*): inversione di una sequenza di interi

- Vediamo come si può invertire la sequenza “SENZA MEMORIZZARLA” (cioè... senza usare né array né liste)

```
void inverti() {           // non ci sono né array né liste
    int n;                 // e neanche assegnamenti
    scanf("%d", &n);
    if( n != SENTINELLA )
        inverti();
    printf("%d ", n);
}
```

- Ma... è proprio vero che la sequenza non è stata memorizzata?  
Qual è lo stato dello stack dei record di attivazione nel momento in cui si esegue la prima printf?

# La soluzione del testo

- Considera le operazioni di *inizializzazione*, *inserimento* e *cancellazione* come delle PROCEDURE
  - cioè funzioni che restituiscono void
- Realizza il passaggio per indirizzo della lista su cui si vuole operare, invece di restituire la lista attraverso la return (per le op. di modifica)
  - La chiamata **lista1 = f ( lista1, ... )** diventa
  - **f ( &lista1, ... )** il puntatore è passato per indirizzo
  - Il parametro formale è un **puntatore a puntatore a** elemento

# Inizializzazione

```
void Inizializza( ListaDiElem * lista ) {  
    *lista = NULL;  
}
```

dichiarazione della variabile testa della lista

```
ListaDiElem lista1;
```

Chiamata di Inizializza: Inizializza( &lista1 );



# Controllo di lista vuota

```
boolean ListaVuota( ListaDiElem lista ) {  
    /* true sse la lista parametro è vuota */  
    return lista == NULL;  
}
```

## Chiamata

```
boolean vuota; /*boolean definito come enumerazione*/  
...           /* typedef enum {false, true} boolean */  
vuota = ListaVuota( lista1 );
```

# Inserimento in prima posizione

```
void InsInTesta( ListaDiElem * lista, TipoElemento elem ) {  
    ListaDiElem punt;  
    punt = (ListaDiElem) malloc(sizeof(ElemLista));  
    punt->info = elem;  
    punt->prox = *lista;  
    *lista = punt;  
}
```

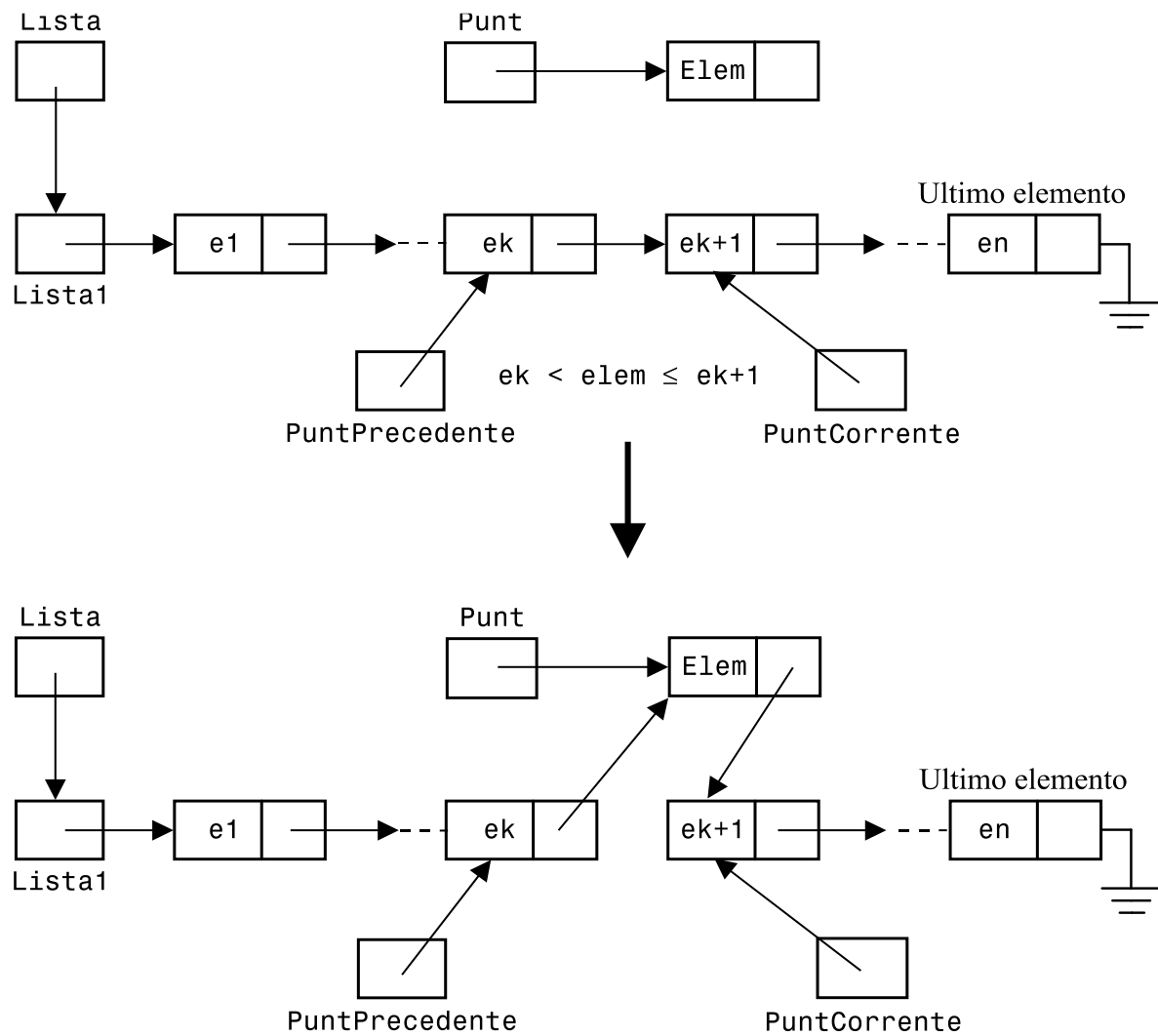
Chiamata : InsInTesta( &lista1, elemento );

# Inserimento in ultima posizione

```
void InsInCoda( ListaDiElem * lista, TipoElemento elem ) {  
    ListaDiElem punt;  
    if ( ListaVuota(*lista) ) {  
        punt = (ListaDiElem) malloc(sizeof(ElemLista));  
        punt->prox = NULL;  
        punt->info = elem;  
        *lista = punt;  
    }  
    else InsInCoda( &((*lista)->prox), elem );  
}
```

# Inserimento in ordine

```
void InsInOrd( ListaDiElem * lista, TipoElemento elem ) {  
    ListaDiElem punt, puntCor, puntPrec=NULL;  
    puntCor = *lista;  
    while ( puntCor != NULL && elem > puntCor->info ) {  
        puntPrec = puntCor;  
        puntCor = puntCor->prox;  
    }  
    punt = (ListaDiElem) malloc(sizeof(ElemLista));  
    punt->info = elem;  
    punt->prox = puntCor;  
    if( puntPrec != NULL )           /* Ins. interno alla lista */  
        puntPrec->prox = punt;  
    else                             /* Ins. in testa alla lista */  
        *lista = punt;  
}
```



# Cancellazione

*/\* Cancella Elem, se esiste, assumendo non vi siano ripetizioni \*/*

```
void Cancella( ListaDiElem *lista, TipoElemento elem ) {  
    ListaDiElem puntTemp;  
    if( ! ListaVuota(*lista) )  
        if( (*lista)->info == elem ) {  
            puntTemp = *lista;  
            *lista = CodaLista(*lista);  
            free( puntTemp );  
        }  
    else Cancella( &((*lista)->prox), elem );  
}
```