

# 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* i dati
    - Cancellazione di un elemento intermedio in un array ordinato
      - occorre far scorrere "indietro" tutti gli elementi successivi
  - Spreco di tempo per *spostare* i dati
    - Inserimento di un elemento intermedio in un array ordinato
      - occorre far scorrere "in avanti" i dati per creare spazio

# Variabili statiche, automatiche e 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 che si disponga di un puntatore che le punti

# Gestione della memoria

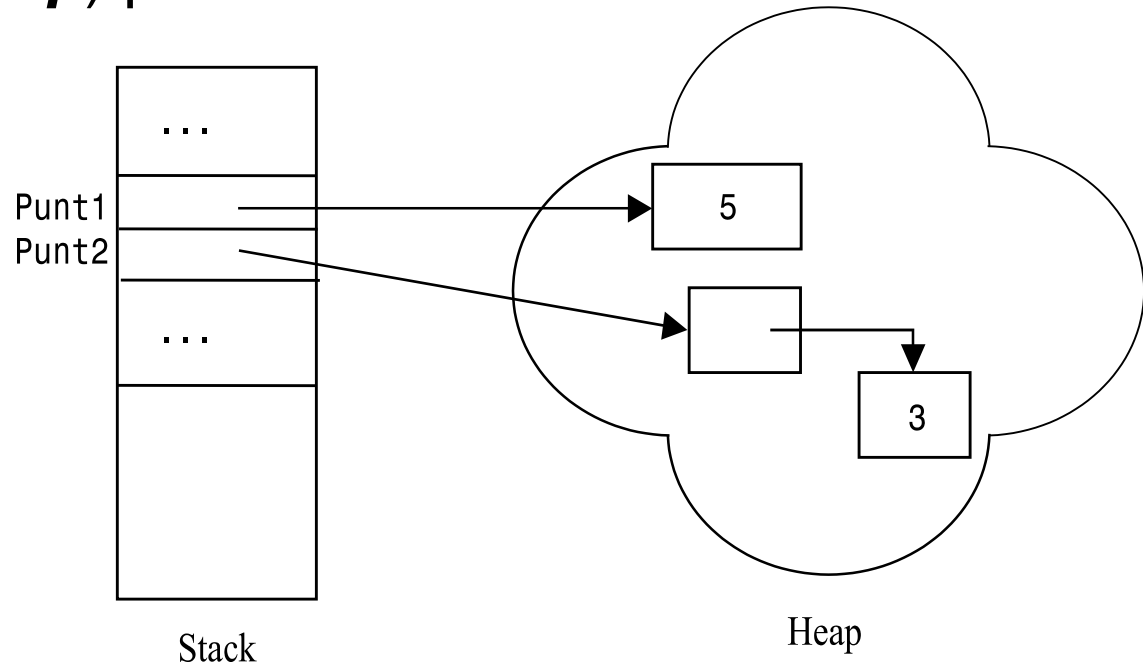
La memoria riservata ai dati del programma è partizionata in due "zone"

- 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 in **<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
- In **<stdlib.h>**: 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 dimensione);`
  - **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 TipoDato * PTD;
```

```
PTD ref;
```

```
...
```

```
ref = (PTD) malloc( sizeof(TipoDato) );
```

**CAST ESPLICITO**



- **Alloca** nello heap **una variabile dinamica** di tipo **TipoDato** e restituisce l'indirizzo della prima cella di memoria 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** )
  - N.B.: ref è una variabile **STATICA**

# 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 intrinsecamente derivabile dal tipo della variabile allocata



# malloc() e free()

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

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

- Attenzione:
  - ptr **NON** è eliminato, e può essere riusato per una nuova malloc
  - Si poteva equivalentemente dichiarare `char * ptr;`
  - In questo modo però ptr può essere riusato per altri tipi di dato

# Dereferenziazione di un puntatore void

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
    void * ptr;    /* puntatore generico */
    ptr = (char *) malloc (sizeof (char));
    *((char*) ptr) = 'a';
    printf ("Carattere: %c\n", *((char*) ptr));
    free (ptr);
}
```

- Attenzione:
  - Per dereferenziare un puntatore void devo fare un cast
  - Come farebbe altrimenti il compilatore a sapere quanti byte devono essere letti...

# Confrontare ...

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

con

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

e

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

# Produzione di "spazzatura"

- La memoria allocata dinamicamente può diventare **inaccessibile** se nessun puntatore punta più ad essa
  - Risulta **sprecata**, e non è recuperabile
    - Ho bisogno di un puntatore per invocare free()
  - 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 */
```

- Alcuni linguaggi (**Java!**) hanno un **garbage collector**
  - Un componente della macchina astratta che trova e riutilizza la memoria inaccessibile (non più referenziata)

# 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

# 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);  
}
```

p è dangling dopo  
la chiamata di boh

in pratica, però, stampa 55  
**Perché?**

```
#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", *p);  
}
```

## Che cosa fa?

A p è assegnato l'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 PUÒ  
SUCCEDERE CON I  
PUNTATORI**

**(e non c'entrano niente con  
la memoria dinamica)**

# Avvertimento

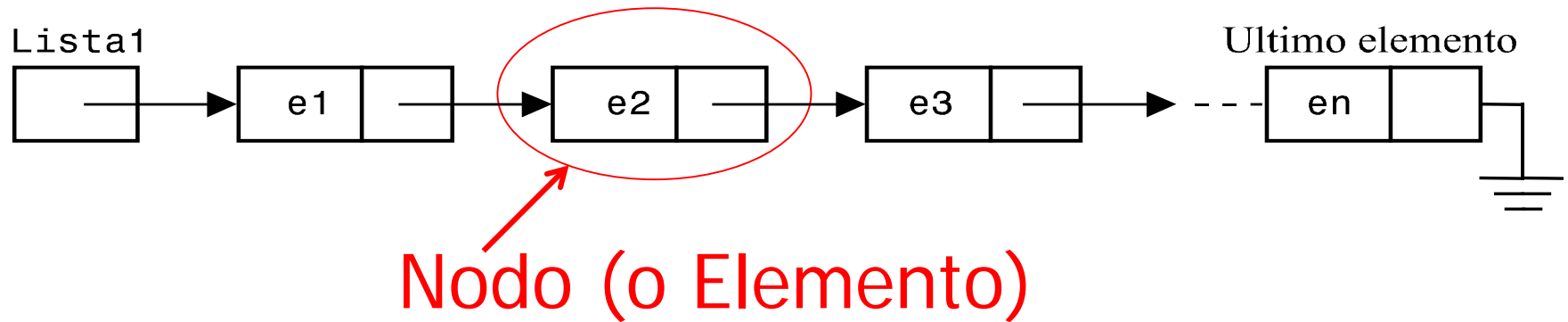
- Puntatori e variabili dinamiche portano a programmare a basso livello e "pericolosamente"
- Sono da usare con parsimonia, solo quando è strettamente necessario, e cioè:
  - per passare parametri per indirizzo
  - per costruire strutture dati complesse
    - Liste, alberi, grafi, ... (che studiamo subito)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

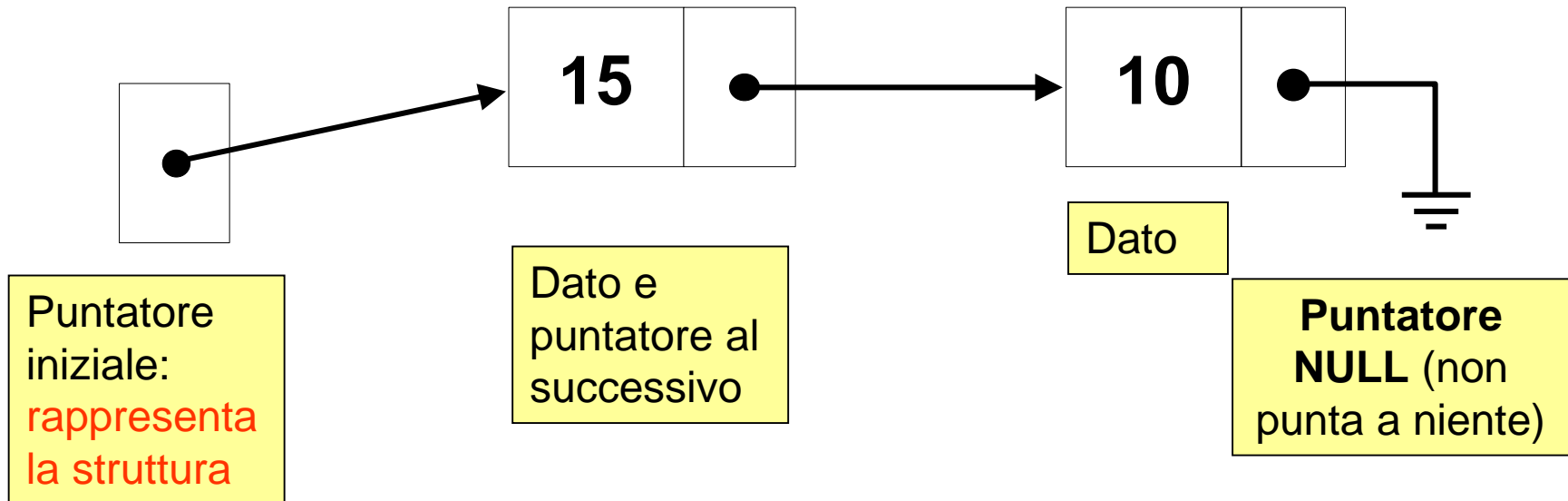
# Lista concatenata (Linked list)



- Composta da **elementi** allocati dinamicamente (**accessibili tramite puntatori**), il cui numero cambia durante l'esecuzione
- Ogni elemento contiene un **puntatore (link) al prossimo** elemento della lista
  - Il primo (**testa della lista**) deve essere puntato "a parte" (non ha un precedente)
- Testa: punto di accesso della lista
  - Gli elementi successivi al primo (**la coda della lista**) si raggiungono attraversando i link da un oggetto all'altro
- L'ultimo non ha un successivo: **punta a NULL**
- NULL è interpretabile anche come "lista vuota"

# 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**



# 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!!!

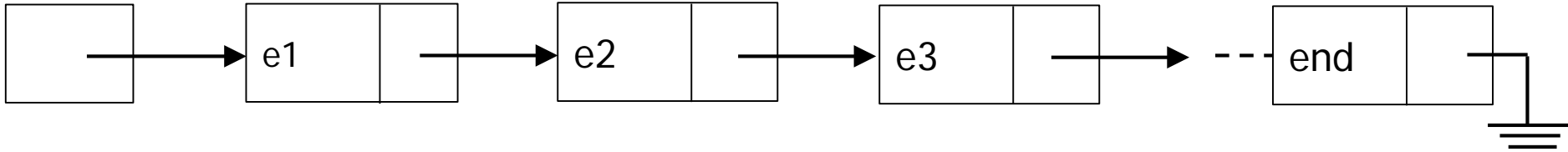
# Uso

- 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

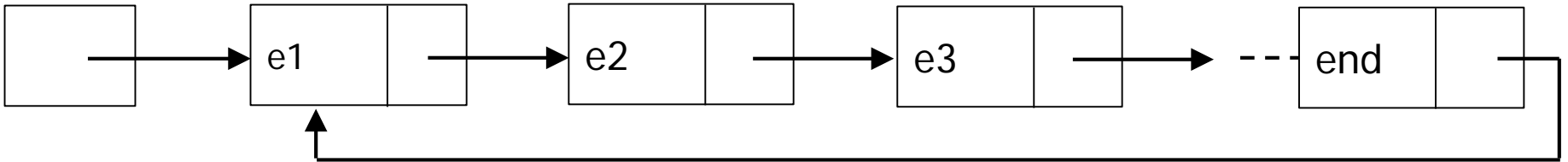
Lista1

**Lista semplice**



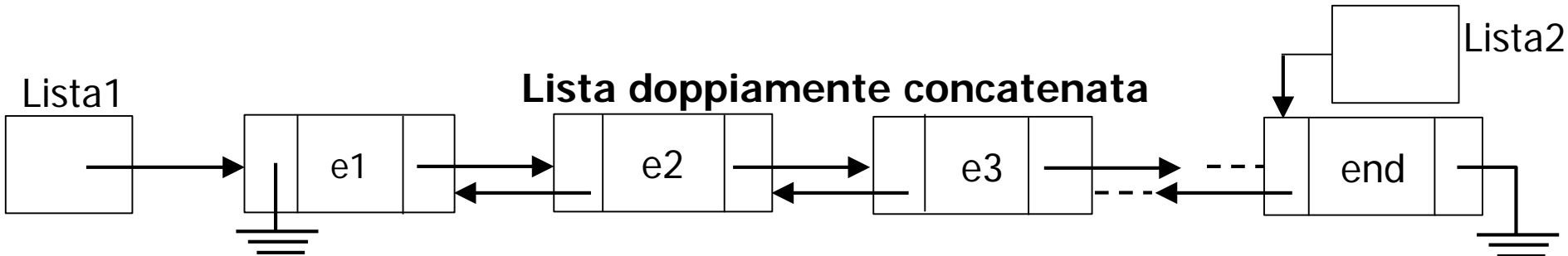
Lista1

**Lista semplice circolare**



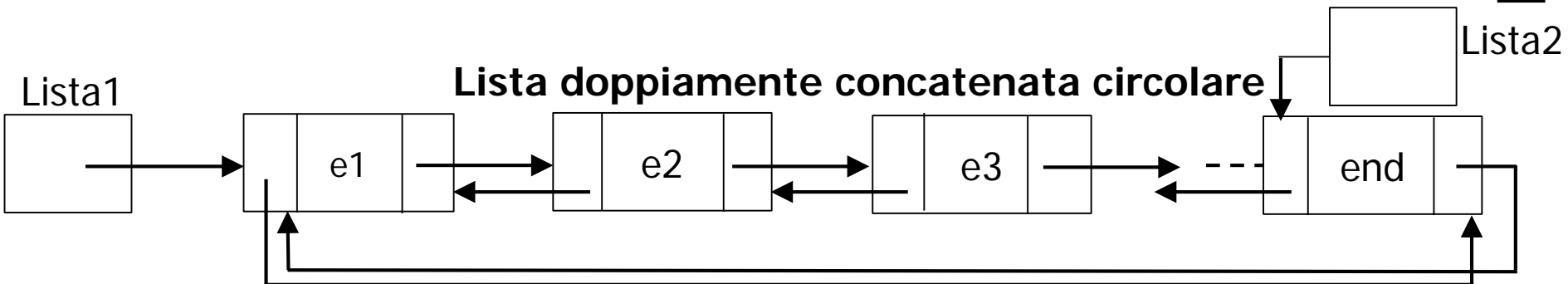
Lista1

**Lista doppiamente concatenata**



Lista1

**Lista doppiamente concatenata circolare**





# Creare un singolo nodo

```
typedef struct N {  
    int dato;  
    struct N * next;  
} Nodo;  
typedef Nodo * ptrNodo;
```

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

# Creare una lista di **due** nodi

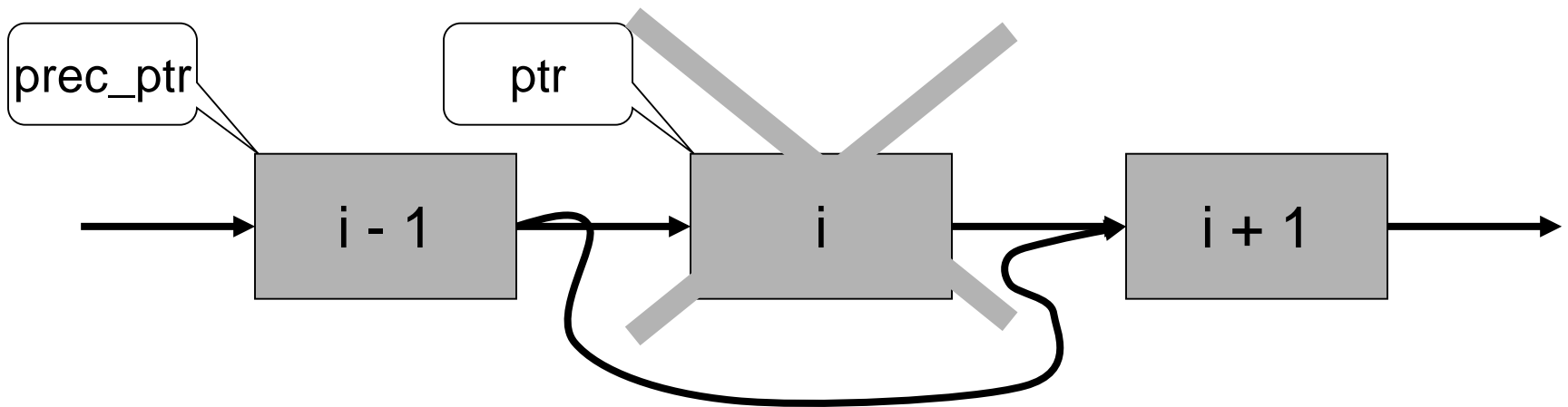
...

```
ptrNode Lista;    /* puntatore alla testa della lista */
ptrNode ptr;      /* puntatore ausiliario a nodo */
Lista = malloc (sizeof(Node));    /* crea 1°  nodo */
Lista->dato = 10;    /* inizializza 1°  nodo */
ptr = malloc (sizeof(Node));    /* 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 */
```

...

# Cancellare un nodo interno

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



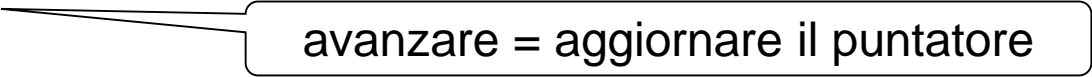
# Cercare un nodo nella lista

```
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

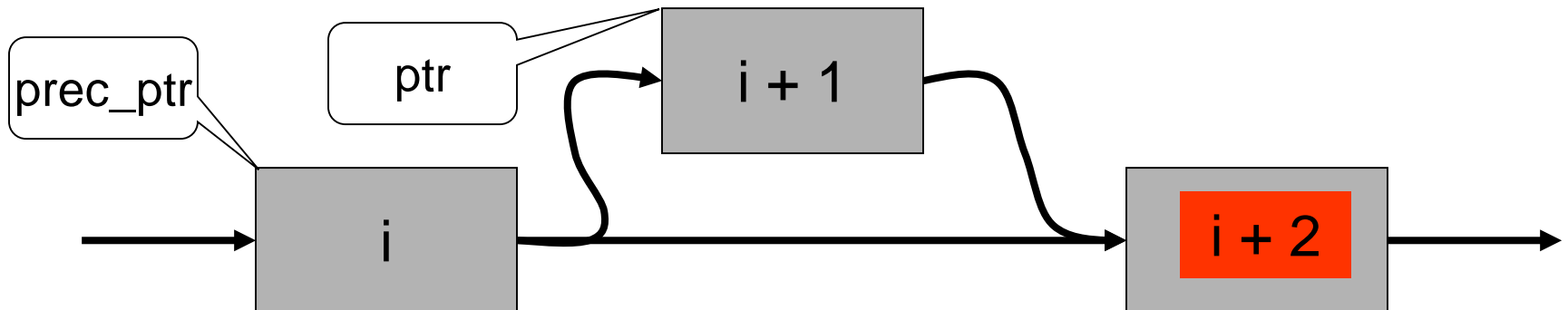
```
int numeronodi = 0;  
ptrNode Lista, ptr;  
Lista = ... /* costruzione della lista */  
for( ptr=Lista; ptr!=NULL; ptr=ptr->next )  
    numeronodi++;
```

- 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

*(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

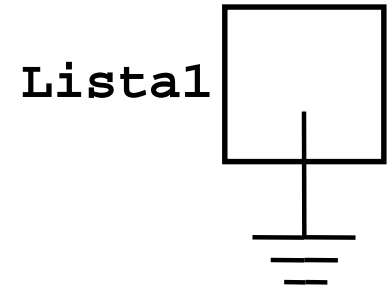
# Definizione dei tipi

```
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 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 Dimensione(ListaDiElem lista) {  
    if ( ListaVuota(lista) )  
        return 0;  
    return 1 + Dimensione( lista->Prox );  
}
```

# Controllo presenza di un elemento

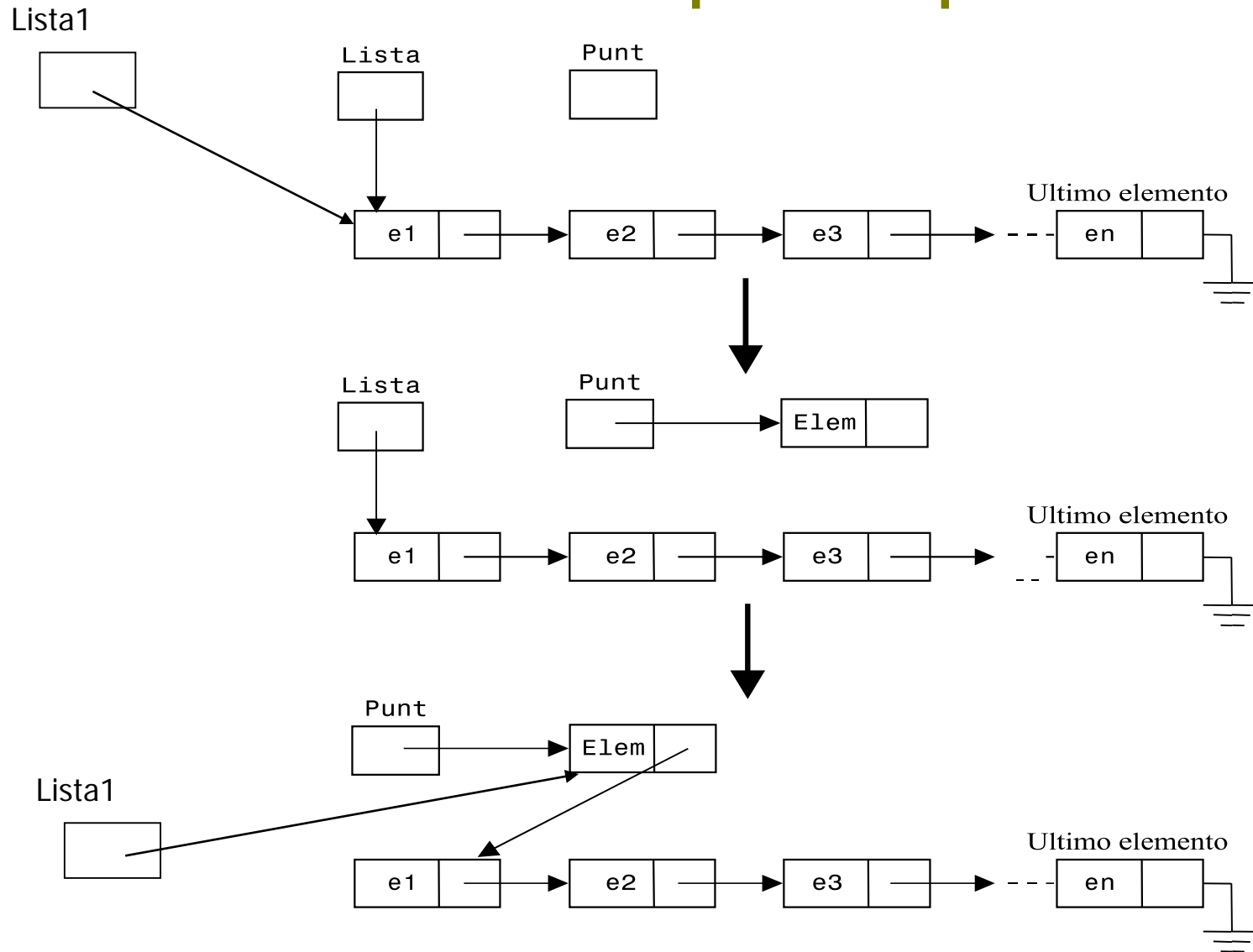
```
int VerificaPresenza (ListaDiElem Lista, TipoElemento Elem) {  
    ListaDiElem cursore = Lista; /* La lista non è vuota */  
    while ( cursore != NULL ) {  
        if ( cursore->Info == Elem )  
            return 1;  
        cursore = cursore->Prox;  
    }  
    return 0; /* Falso: l'elemento 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



# Inserimento in prima posizione

```
ListaDiElem InsInTesta ( ListaDiElem Lista,  
                          TipoElemento Elem ) {  
    ListaDiElem Punt;  
    Punt = 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*)

# Inserimento in ultima posizione (it.)

```
ListaDiElem InsInCoda(ListaDiElem Lista, TipoElemento Elem)
{
    ElemLista *Punt,*cur=Lista;
    Punt = malloc( sizeof(ElemLista) );
    Punt->Prox = NULL;
    Punt->Info = Elem;
    if ( ListaVuota(Lista) )
        return Punt;
    else { while( cur->Prox != NULL )
            cur = cur->Prox;
          cur->Prox = Punt;
        }
    return Lista;
}
```

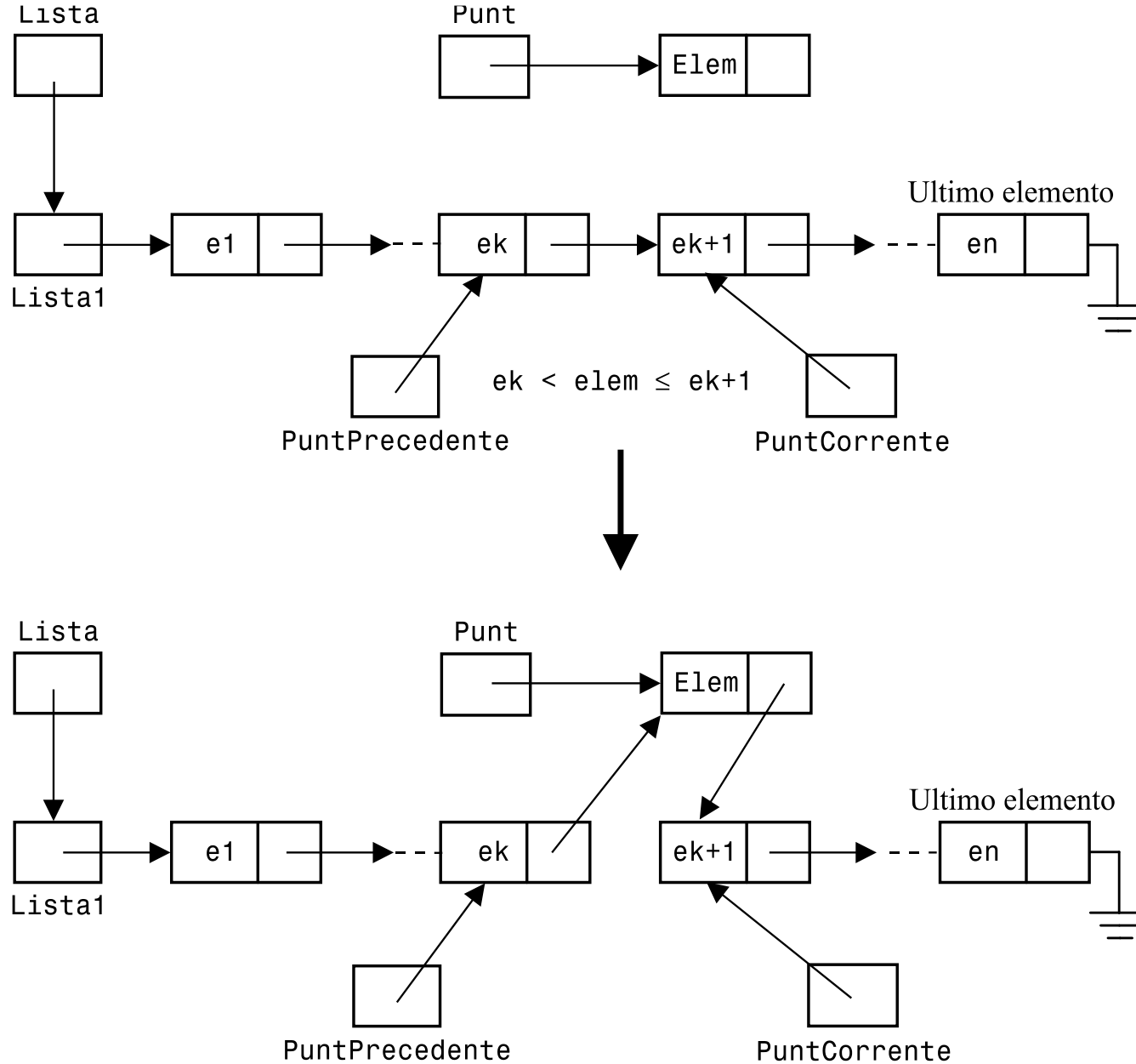
Chiamata :    **Lista1** = InsInCoda (**Lista1**, Elemento);

# Inserimento in ultima posizione (ric.)

```
ListaDiElem InsInFondo(ListaDiElem Lista, TipoElemento Elem)
{
    ElemLista *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;
    }
}
```

chiamata :    **Lista1** = InsInFondo (**Lista1**, Elemento);

# Inserimento in lista ordinata



# Inserimento in lista ordinata

```
ListaDiElem InsInOrd(ListaDiElem Lista, TipoElemento Elem) {
    ElemLista *Punt, *PuntCor, *PuntPrec;
    PuntPrec = NULL;
    PuntCor = Lista;
    while ( PuntCor != NULL && Elem > PuntCor->Info ) {
        PuntPrec = PuntCor;
        PuntCor = PuntCor->Prox;
    }
    Punt = malloc(sizeof(ElemLista));
    Punt->Info = Elem;
    Punt->Prox = PuntCor;
    if (PuntPrec != NULL ) { /* Inserimento interno alla lista */
        PuntPrec->Prox = Punt;
        return Lista;
    }
    else return Punt; /* Inserimento in testa alla lista */
}
```

Chiamata : **Lista1** = InsInOrd (**Lista1**, Elemento);

# 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;  
        }  
    else  
        return Lista;  
}
```

Salvo il puntatore al nodo  
seguente

Chiamata : **Lista1** = Cancella (**Lista1**, Elemento);



# Visualizza Lista

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

# Visualizza Lista Rovesciata

```
void VisualizzaListaRov (ListaDiElem lista) {  
    if ( ListaVuota(lista) )  
        printf(" |--- \n");  
    else {  
        VisualizzaListaRov ( lista->Prox );  
        printf(" %d\n <--- ", lista->Info);  
    }  
}
```

# Passaggio dei parametri per indirizzo

- Operazioni di *inizializzazione*, *inserimento* e *cancellazione* come delle PROCEDURE
  - cioè funzioni che restituiscono void
- 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)
{
    ElemLista * Punt;
    Punt = 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) {  
    ElemLista * Punt;  
    if (ListaVuota(*Lista)) {  
        Punt = malloc(sizeof(ElemLista));  
        Punt->Prox = NULL;  
        Punt->Info = Elem;  
        *Lista = Punt;  
    }  
    else InsInCoda(&((*Lista)->Prox), Elem);  
}
```

Chiamata : InsInCoda(&Lista1, Elemento);

# Inserimento in ordine

```
void InsInOrd(ListaDiElem * Lista, TipoElemento Elem) {  
    ElemLista * Punt, * PuntCor, * PuntPrec=NULL;  
    PuntCor = *Lista;  
    while (PuntCor != NULL && Elem > PuntCor->Info) {  
        PuntPrec = PuntCor;  
        PuntCor = PuntCor->Prox;  
    }  
    Punt = 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;  
}
```

**Chiamata:** InsInOrdine(&Lista1, Elemento);



# Cancellazione

```
/* Cancella Elem, se esiste, assumendo non vi siano ripetizioni */  
void Cancella(ListaDiElem *Lista, TipoElemento Elem) {  
    ElemLista * PuntTemp;  
    if (ListaVuota (*Lista) == false)  
        if ((*Lista)->Info == Elem) {  
            PuntTemp = *Lista;  
            *Lista = (*Lista)->prox; /*rimuovo il  
            primo nodo della lista*/  
            free(PuntTemp);  
        }  
    else Cancella(&((*Lista)->Prox), Elem);  
}
```