Memoria Dinamica

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

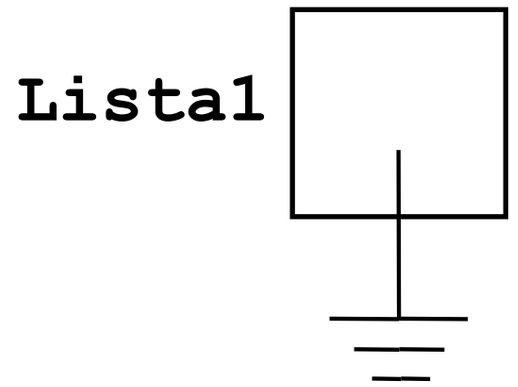
Usiamo questa formulazione

```
typedef struct  EL {
   TipoElemento Info;
   struct EL * Prox;
} ElemLista;
```

typedef ElemLista * **ListaDiElem**;

Inizializzazione

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



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**

<u>Esempio di chiamata</u>:

```
…
ListaDiElem Lista1;
…
Lista1 = Inizializza ();
```

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;
        if ( ! ListaVuota(Lista) ) {
                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

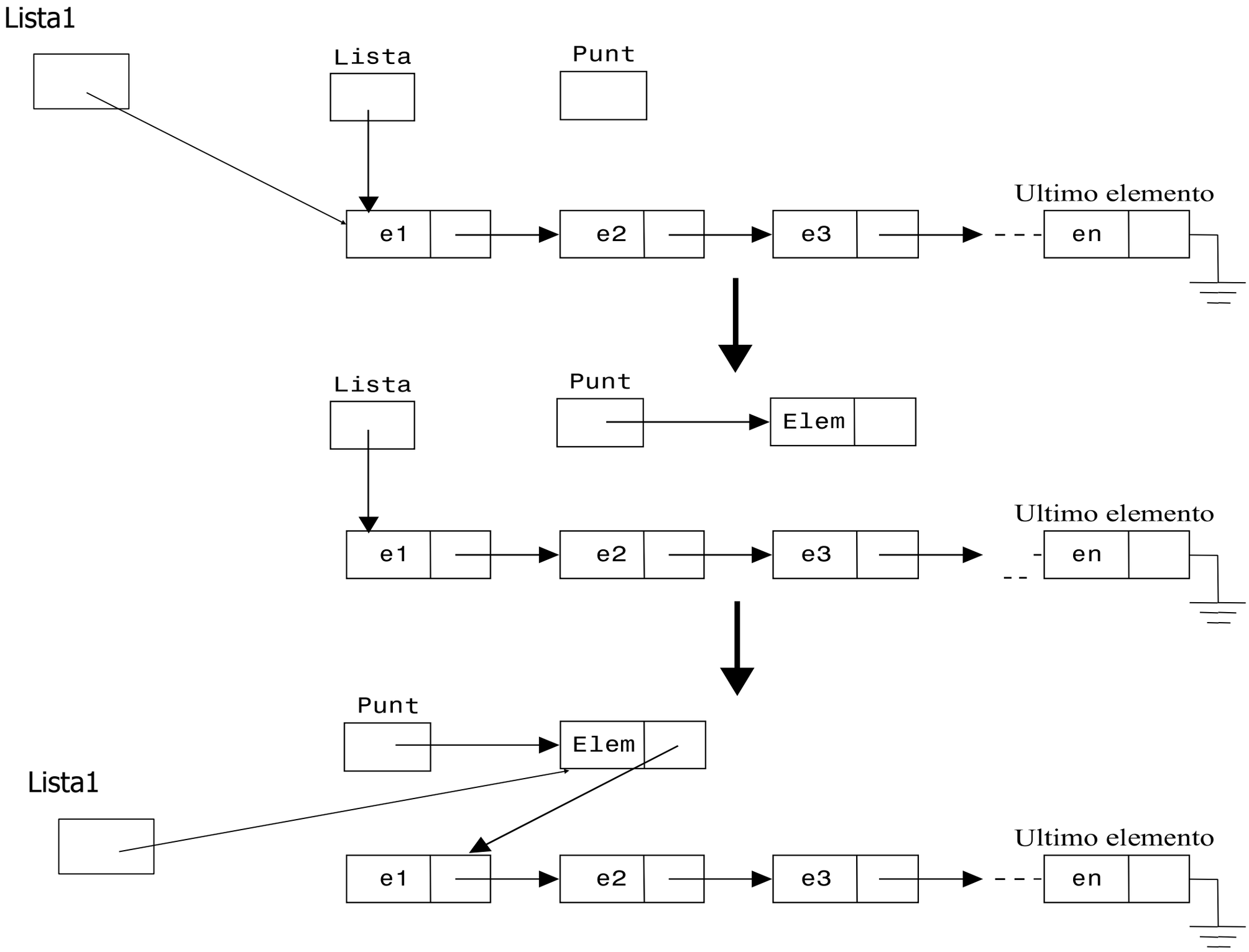
```
ListaDiElem InsInTesta ( ListaDiElem Lista,
                                    TipoElemento Elem ) {
    ListaDiElem Punt;
    Punt = malloc(sizeof(ElemLista));
    Punt–>Info = Elem;
    Punt–>Prox = Lista;
    return  Punt;
}
```

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)

Visualizzazione



Inserimento in ultima posizione (it.)

```
ListaDiElem InsInFondo(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

```
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;
}
```

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

Inversione RICORSIVA…

- Se la lista ha 0 o 1 elementi, è pari alla sua inversa (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;
        }
}
```

Visualizza Lista

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

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)
{
        ElemLista * Punt;
        Punt = malloc(sizeof(ElemLista));
        Punt–>Info = Elem;
        Punt–>Prox = *Lista;
        *Lista = Punt;
}
```

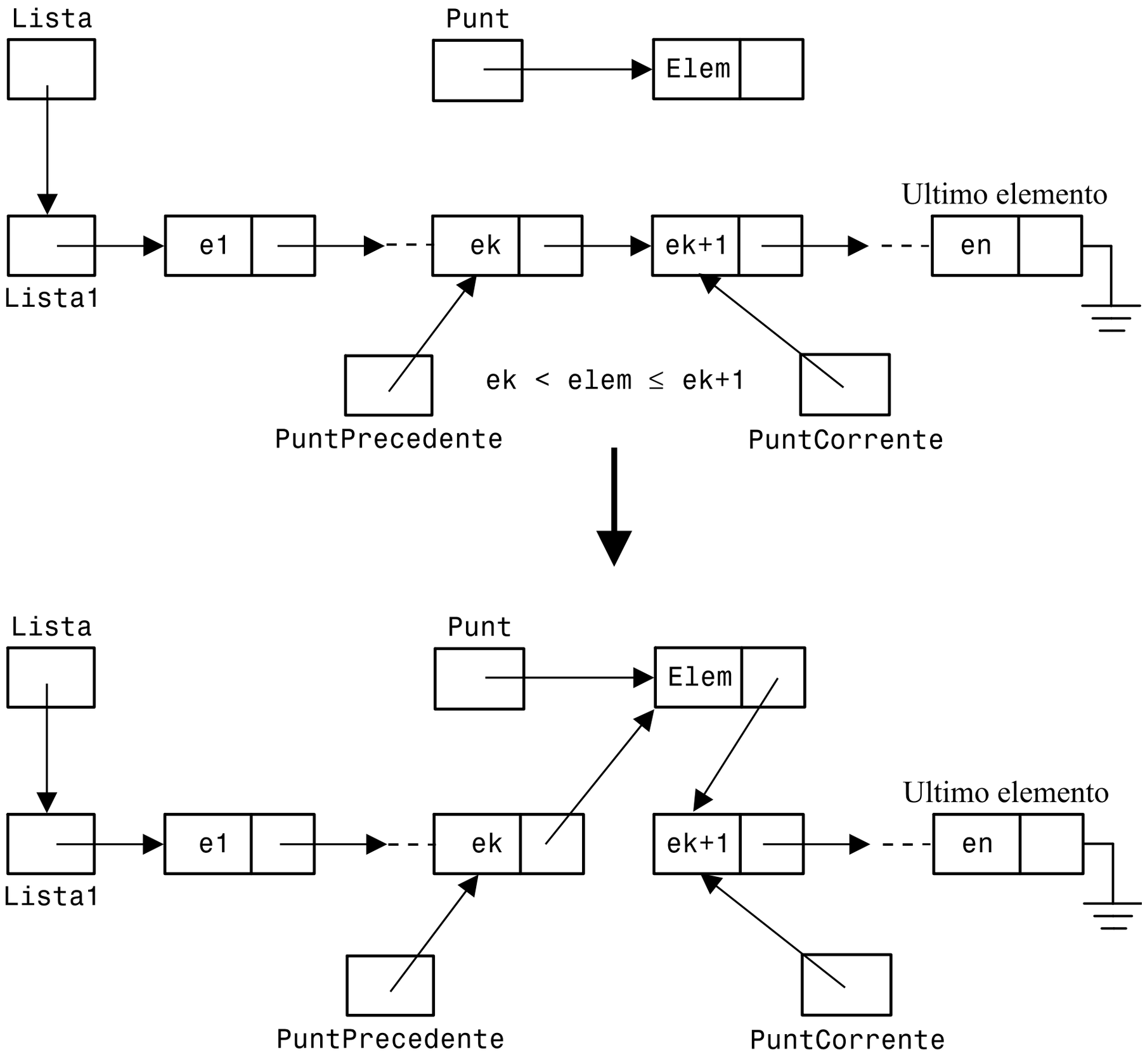
Chiamata : InsInTesta(&Lista1, Elemento);

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);
}
```

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 */
                PuntPrecedente–>Prox = Punt;
        else      /* Ins. in testa alla lista */
                *Lista = Punt;
}
```


Lista
Punt
Elem
Ultimo elemento
e1
ek
ek+1
en
Lista1
ek < elem ≤ ek+1
PuntPrecedente
PuntCorrente

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 = CodaLista(*Lista);
                free(PuntTemp);
            }
            else Cancella(&((*Lista)–>Prox), Elem);
}
```