



Strutture Dati Dinamiche: Le Liste

Informatica A AA 2023 / 2024

Giacomo Boracchi

13 Novembre 2023

giacomo.boracchi@polimi.it

Slide credits Prof. Alessandro Campi

Strutture dati dinamiche

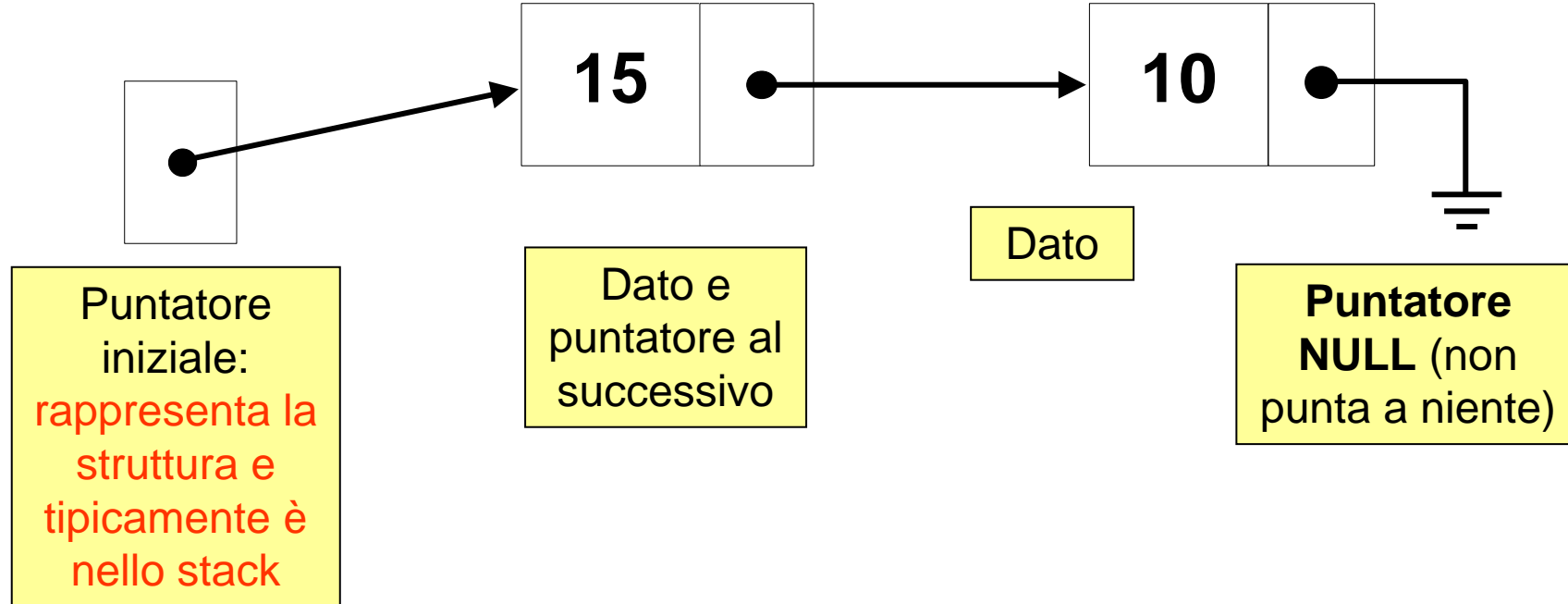
Crescono e decrescono durante l'esecuzione del programma:

- **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)

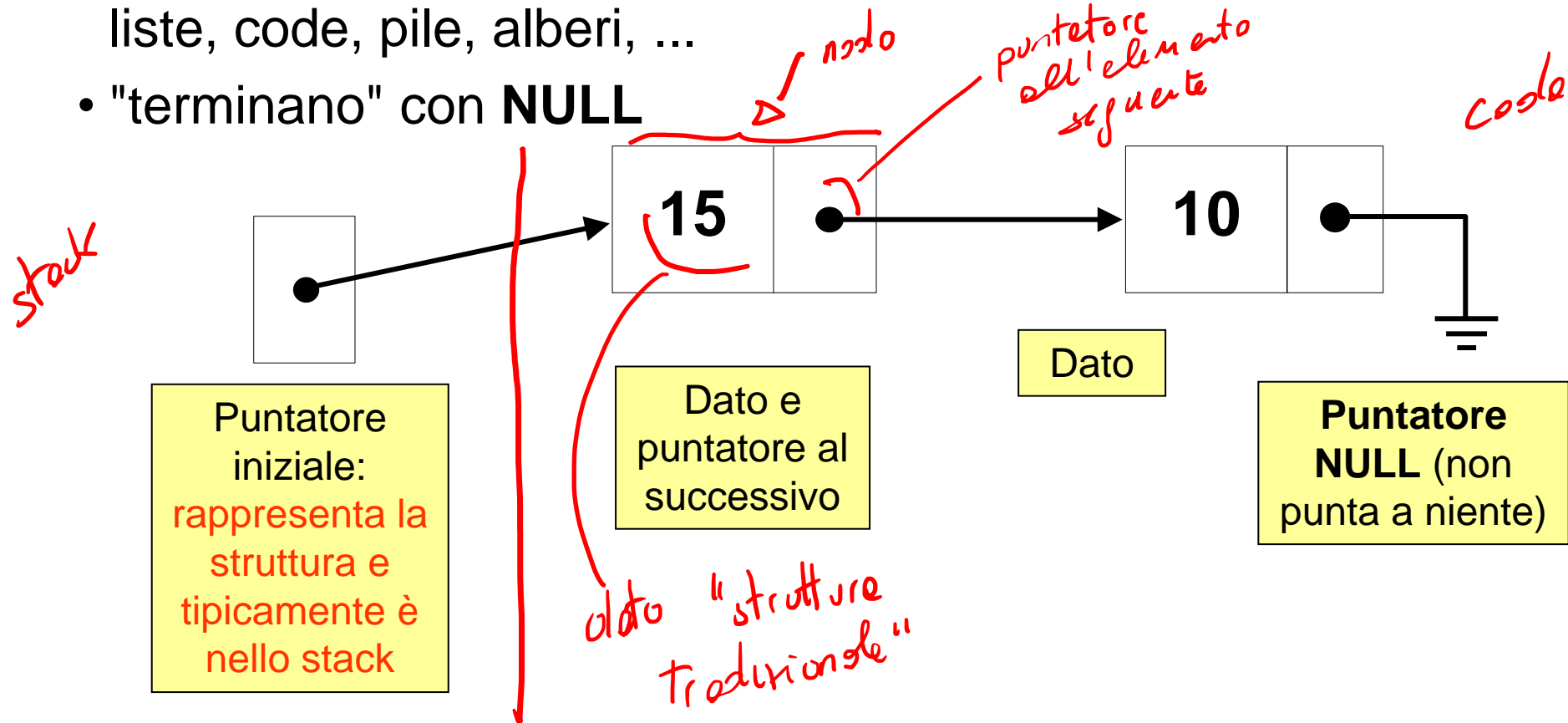
- **Strutture con puntatori a strutture dello stesso tipo**
- Si possono **concatenare** per ottenere strutture dati utili come: liste, code, pile, alberi, ...
- "terminano" con **NULL**



Strutture dati ricorsive

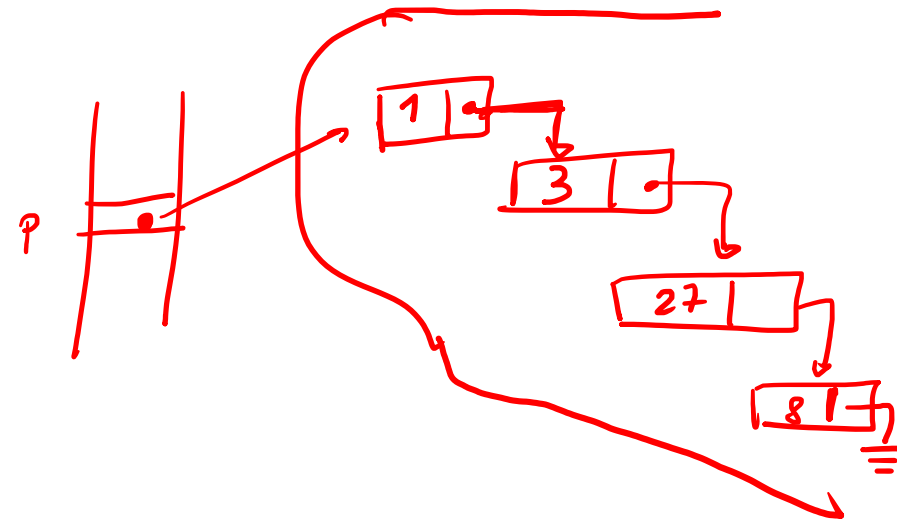
(o auto-referenziali)

- **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 (è in memoria dinamica)
- 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



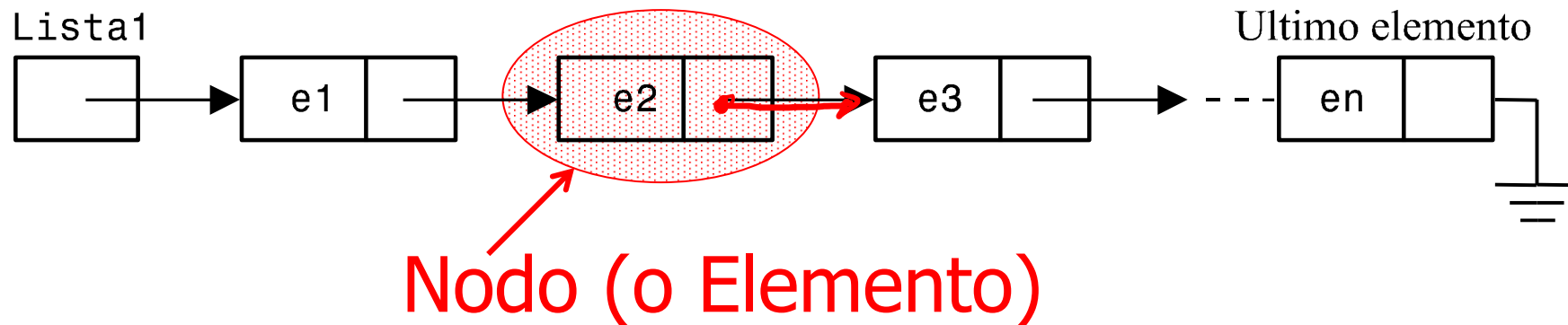
La Lista

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...

La userp

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

Notare la sintassi!

- ...e il tipo del puntatore

```
typedef ElemLista * ListaDiElem;
```

Definizione Ricorsiva!!!

```
ListaDiElem l;
```

Strutture dati ricorsive (dichiaraz.)

- Si definiscono il tipo del nodo...

La userp

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

Notare la sintassi!

*puntatore
al prossimo
elemento della lista*

- ...e il tipo del puntatore

```
typedef ElemLista * ListaDiElem;
```

Definizione Ricorsiva!!!

```
ListaDiElem l;
```

*per evitare
di usare
un puntatore!*

Strutture dati ricorsive (variante)

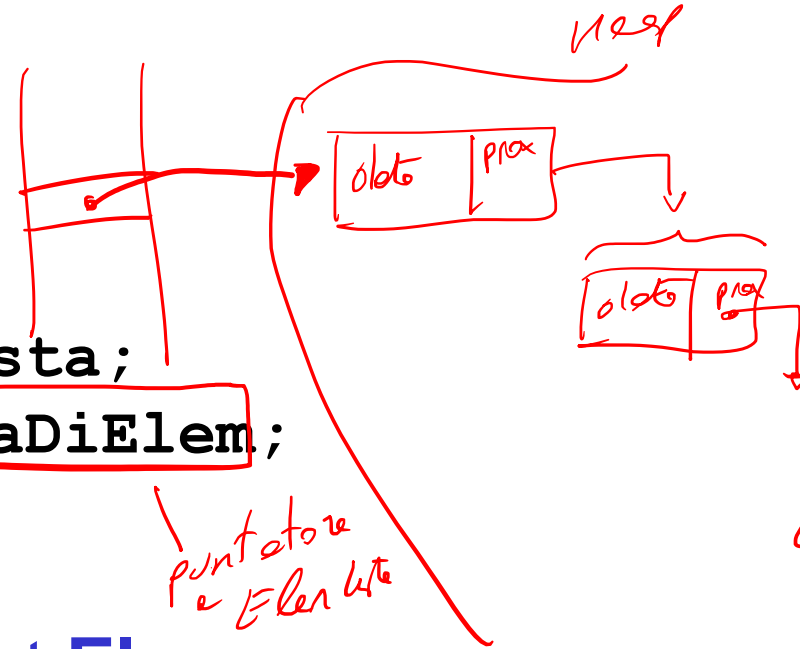
```
struct El {  
    TipoElemento dato;  
    struct El * prox;  
};  
typedef struct El ElemLista;  
typedef struct El * ListaDiElem;
```

Il puntatore **prox**:

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

Strutture dati ricorsive (variante)

```
struct E1 {  
    TipoElemento dato;  
    struct E1 * prox;  
};  
typedef struct E1 ElemLista;  
typedef struct E1 * ListaDiElem;
```



Il puntatore **prox**:

- punta a un oggetto di tipo **struct E1**
- si chiama **link**
- lega oggetti di tipo **struct E1** tra di loro

Liste concatenate (linked lists)

Lista concatenata:

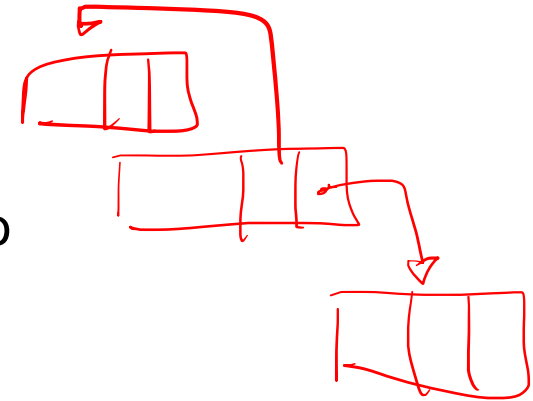
- Collezione lineare di oggetti di tipo auto-referenziale, chiamati ***nodi***, collegati tramite puntatori (***link***)
- Vi si accede mediante un puntatore al primo nodo della lista (la ***testa*** della lista)
- **Gli elementi successivi al primo** (la ***coda*** della lista) **si raggiungono attraversando 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 costo 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



Recall: Struct e puntatori

```
typedef struct {  
    int    PrimoCampo;  
    char   SecondoCampo;  
} TipoDato;  
TipoDato t;  
TipoDato * p = &t;
```

*Sintassi per accedere
ai campi di una struct
tramite un puntatore p*

```
p->PrimoCampo    = 12;  
(*p).PrimoCampo = 12;
```

} *equivalenti*

Come accedere ai campi di una struttura tramite puntatori

```
int main()
{
    SovraStruttura *p, n;
    p = &n;
    printf("inserisci v");
    scanf("%d", &p->v); // equivale a scanf("%d", &n.v); e scanf("%d", &(*n).v);
    printf("inserisci x");
    scanf("%d", &p->s.x); //equivale a scanf("%d", &n.s.x);
    fflush(stdin);
    printf("inserisci stringa");
    scanf("%s", p->s.stringa); //scanf("%d", n.s.stringa);

    printf("\ncon punt:\np->v = %d, p->s.x = %d, p->s.stringa = %s", p->v, p->s.x, p->s.stringa);
    printf("\ncon var :\nn.v = %d, n.s.x = %d, n.s.stringa = %s", n.v, n.s.x, n.s.stringa);
    return 0 ;
}
```

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

typedef struct {
    int x;
    char stringa[10];
}Struttura;

typedef struct {
    int v;
    Struttura s;
}SovraStruttura;
```

Creare un singolo nodo di una lista

```
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(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 */
```

...

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

...

```
ptrNodo 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 */
```

...

```
#include <stdlib.h>
// definizione del tipo
typedef struct EL{
    int info;
    struct EL * next;
}Nodo;
// è comodo anche definire il tipo puntatore a nodo
typedef Nodo *PNodo;
void stampaLista(PNodo);

int main()
{
    // dichiaro 3 puntatori a nodi per creare questi elementi in heap,
    // dichiaro un altro puntatore p che userò per operare nella lista
    PNodo ptesta, pcorpo, pcoda, p;
    int i;

    // creo uno spazio per la testa nello heap
    ptesta = (Nodo *) malloc(sizeof(Nodo));
    // creo uno spazio per il corpo
    pcorpo = (Nodo *) malloc(sizeof(Nodo));
    // creo uno spazio per la coda
    pcoda = (Nodo *) malloc(sizeof(Nodo));
}
```

```
// popolo i nodi assegnando ai campi di una struttura a cui accedo con un puntatore
p testa->info = 7;
p testa->next = p corpo; // il primo nodo punta alla coda
p corpo->info = 8;
p corpo->next = p coda;
p coda->info = 9;
p coda->next = NULL;

// se mettessi p coda->next = p testa; farei una lista ciclica, la stampa non terminerebbe
// scorro e stampo il contenuto la lista
// prendo un puntatore che sia "spostabile" e lo faccio puntare alla testa
// NON sposto p testa perchè se no non posso più tornare indietro.
p = p testa; i = 0;
while(p->next != NULL)
{
    printf("\nlista[%d] = %d", i, p->info);
    p = p->next; // non si può fare l'aritmetica del puntatore p++ perché siamo
                // nello heap!

    i++;
}
// per stampare anche il valore della coda
printf("\nlista[%d] = %d", i, p->info);
}
```

Stampa del contenuto di una lista

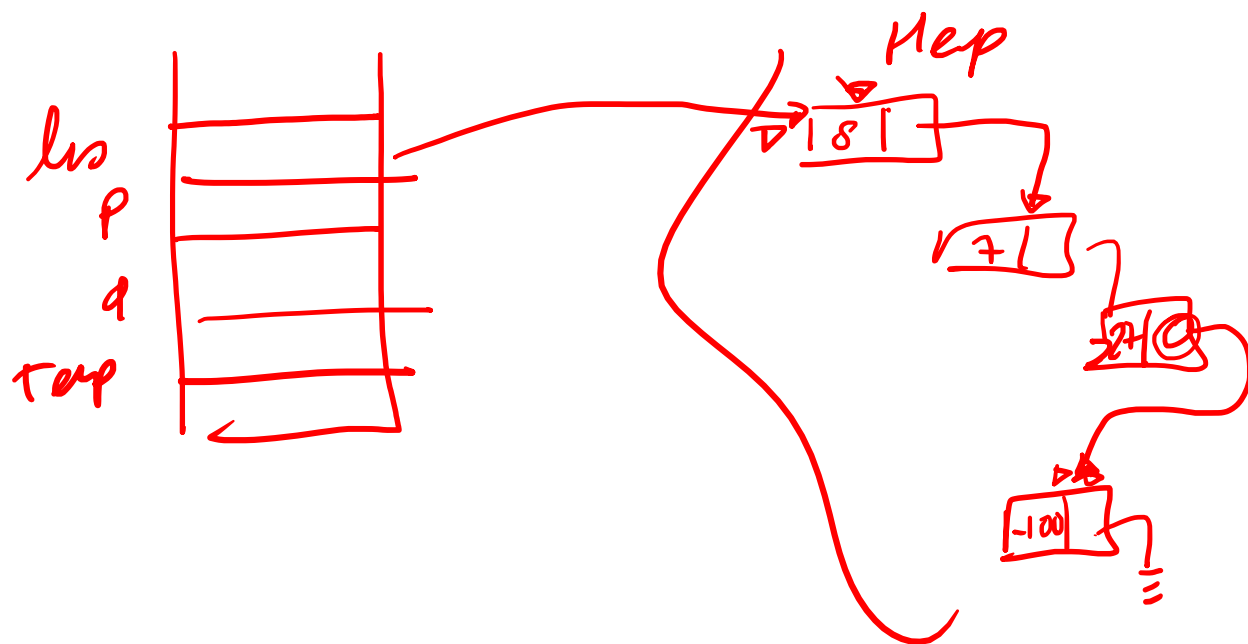
```
// questa funzione modifica p che è una copia del puntatore alla testa, non c'è bisogno
// di fare un'altra copia locale... tanto p non viene restituita e viene distrutta al
// termine dell'invocazione
```

```
void stampaLista(PNodo p)
{
    int i = 0;
    while (p!=NULL)
    {
        printf("\nlista[%d] = %d", i, p->info);
        p = p->next;
        i++;
    }
}
```

Stampa del contenuto di una lista con una funzione, non serve usare un «puntare copia», si può mandare il parametro a NULL, tanto questo non modifica l'indirizzo a cui punta il puntatore nel record di attivazione del main

```
// popolo i nodi assegnando ai campi di una struttura a cui accedo con un puntatore
    ptesta->info = 7;
    ptesta->next = pcorpo; // il primo nodo punta alla coda
    pcorpo->info = 8;
    pcorpo->next = pcoda;
    pcoda->info = 9;
    pcoda->next = NULL;

// OSS, la funzione stampaLista ha accesso al heap tramite un puntatore locale
// che raccoglie la lista
    stampaLista(ptesta);
    printf("\n");
// ho modificato la copia del puntatore locale alla funzione, non ptesta, quindi posso
stampare anche due volte
    stampaLista(ptesta);
}
```



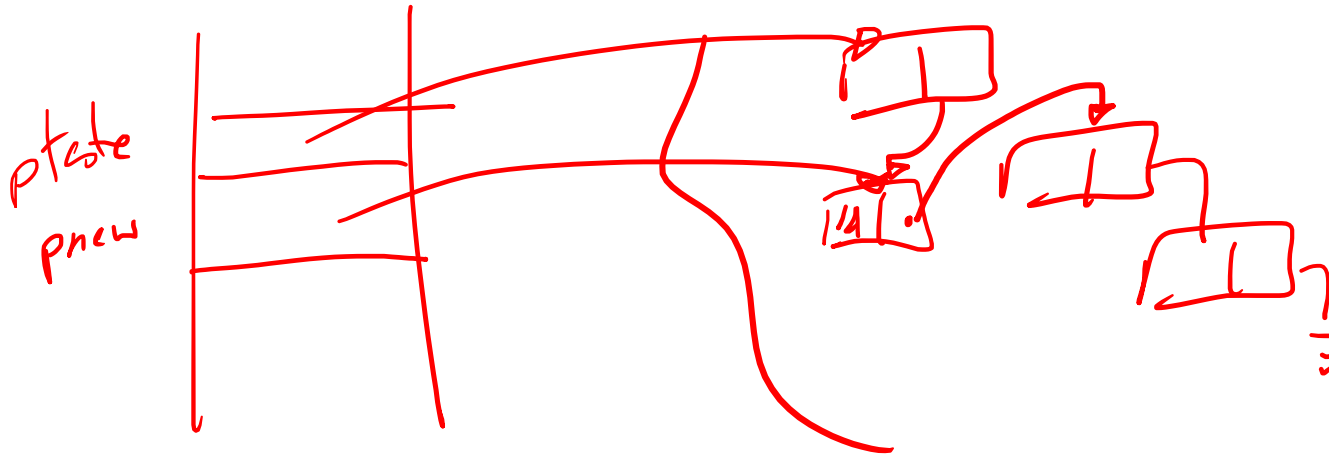
$p_{new} = (N_{ob} \times) \text{ malloc } (6 \dots);$
 $p_{new} \rightarrow data = 8;$
 $p_{new} \rightarrow next = p;$
 $P = p_{new};$

Inserimenti in testa senza funzione

```
// inserisco elemento in testa alla lista
// 1) creo un nuovo elemento
    pnew = (PNodo) malloc(sizeof(Nodo));
    pnew->info = 6;
// 2) attacco il nuovo davanti alla lista
    pnew->next = ptesta;
// 3) modifico il puntatore al primo elemento della lista
    ptesta = pnew;
// OSS: nell'assegnamento 2 ho copiato l'indirizzo di ptesta in pnew->next
//      quindi se modifico poi ptesta, questo non si propaga
```


Inserimento in seconda posizione senza funzione

```
// aggiungo un nuovo elemento in seconda posizione a ptesta
//1) preparo il nuovo nodo (OSS è la stessa istruzione di sopra, ma crea un altro Nodo,
non sovrascrive e quello sopra non è andato perso perchè è raggiungibile dalla lista)
    pnew = (PNodo) malloc(sizeof(Nodo));
    pnew->info = m;
//2) aggancio pnew al successivo di p
    pnew->next = ptesta->next;
//3) faccio raggiungere pnew da p
    ptesta->next = pnew;
// OSS, se avessi fatto 3) prima di 2) avrei creato garbage in quanto tutto quello che
segue da p (prima dell'inserimento) sarebbe diventato irraggiungibile
```



Inserimenti in coda senza funzione

```
// inserisco un elemento in coda
// 1) creo un nuovo elemento
    pnew = (PNodo) malloc(sizeof(Nodo));
    pnew->info = 10;
// 2) attacco il nuovo elemento in coda (qui so qual'è la coda, se no dovrei cercarla)
    pcoda->next = pnew;
// 3) faccio in modo che sia una coda l'ultimo inserito
    pnew->next = NULL;
```

Richiede tuttavia un puntatore alla coda (pcoda) nello stack. Nel caso in cui questo non ci fosse occorre scorrere la lista fino a trovare un nodo che punta a NULL!

Inserimento in prima posizione con funzioni

```
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*)

Nota bene: questa funzione esegue le operazioni corrette anche nel caso in cui la lista passata in ingresso fosse NULL

Inserimento in ultima posizione (funzione iterativa)

```
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 ( lista==NULL )
        return punt;           /* => punt è la nuova lista */
    else {
        while(cur->prox!= NULL ) /* Trova l'ultimo nodo */
            cur = cur->prox;
        cur->prox = punt;       /* Aggancio all'ultimo nodo */
    }
    return lista;
}
```

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

Inserimento in ultima posizione (funzione iterativa)

```
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 ( lista==NULL )
        return punt;           /* => punt è la nuova lista */
    else {
        while(cur->prox!= NULL ) /* Trova l'ultimo nodo */
            cur = cur->prox;
        cur->prox = punt;       /* Aggancio all'ultimo nodo */
    }
    return lista;
}                               Chiamata : lista1 = InsInFondo( lista1, elemento );
```

Serve che la funzione **InsInFondo** restituisca la nuova lista perché la lista passata in ingresso potrebbe essere vuota.

In questo caso occorrerebbe quindi un inserimento in testa.

Altrimenti modifiche all'interno o in coda alla lista possono essere inserite anche con una funzione void.

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 ( lista==NULL )          /* => 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** = InsInFondo(**lista1**, elemento);

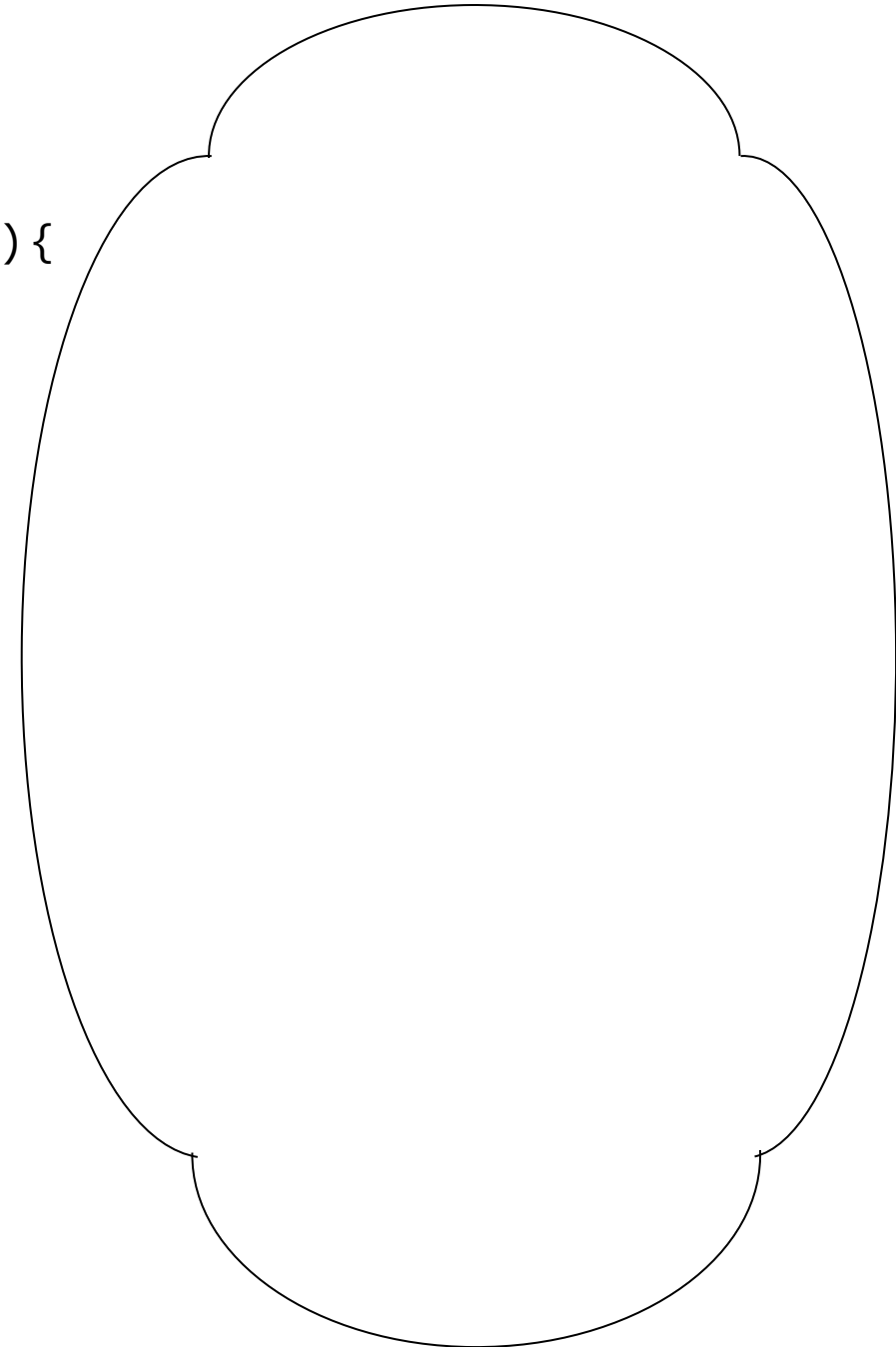
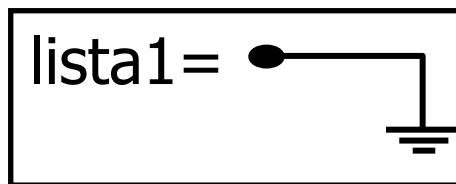
È necessario mantenere il puntatore alla testa (lista in questo caso) che viene restituito senza essere modificato. Cur invece viene modificato per scorrere la lista e quindi se lo restituissi, perderei la lista nell'invocazione

```

int main() {
    ListaDiElem lista1;
    ... ..
    lista1 = InsInFondo( lista1, 6 );
    ... ..
}

ListaDiElem InsInFondo( ListaDiElem lista, int elem ) {
    ListaDiElem punt, cur = lista;
    punt = (ListaDiElem) malloc( sizeof(ElemLista) );
    punt->prox = NULL;
    punt->info = elem;
    if ( lista==NULL )
        return punt;
    else {
        while( cur ->prox != NULL )
            cur = cur->prox;
        cur ->prox = punt;
    }
    return lista;
}

```

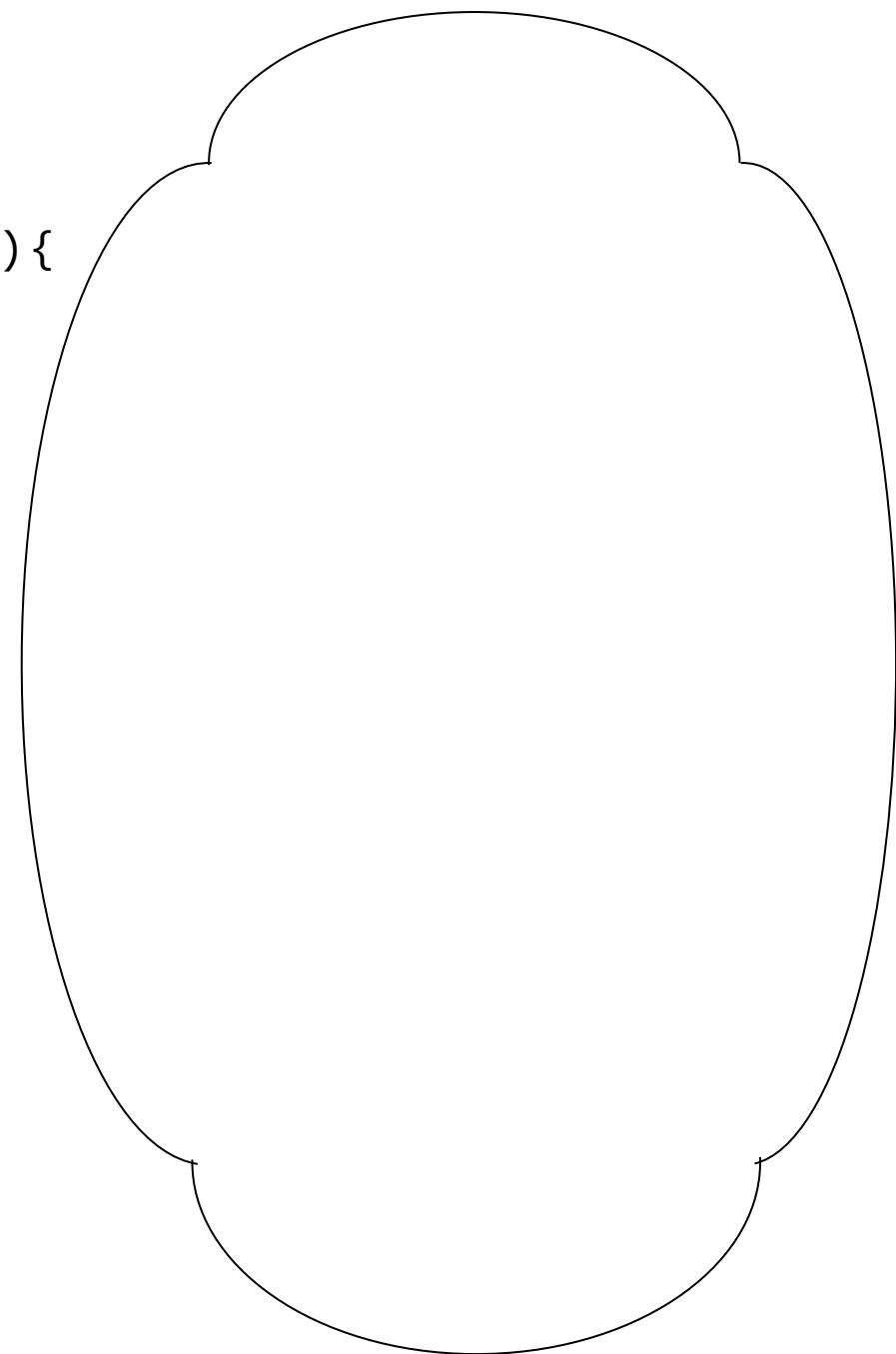
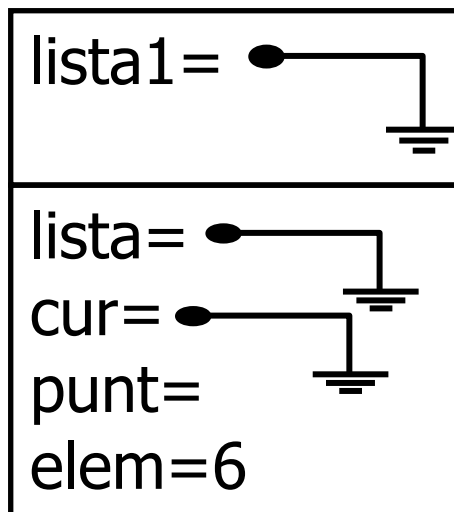


```

int main() {
    ListaDiElem lista1;
    ... ..
    lista1 = InsInFondo( lista1, 6 );
    ... ..
}

ListaDiElem InsInFondo( ListaDiElem lista, int elem ) {
    ListaDiElem punt, cur = lista;
    punt = (ListaDiElem) malloc( sizeof(ElemLista) );
    punt->prox = NULL;
    punt->info = elem;
    if ( lista==NULL )
        return punt;
    else {
        while( cur ->prox != NULL )
            cur = cur->prox;
        cur ->prox = punt;
    }
    return lista;
}

```

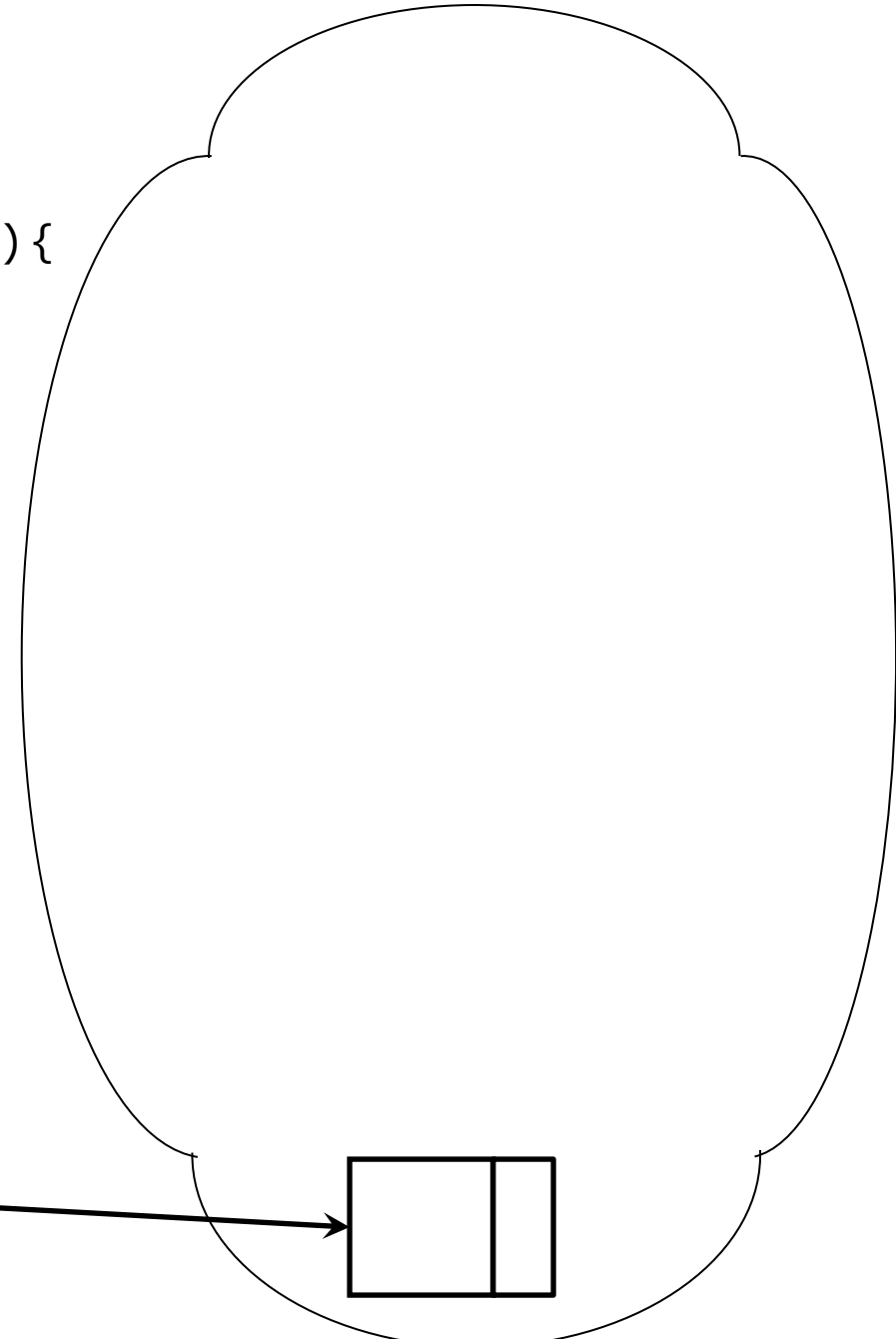
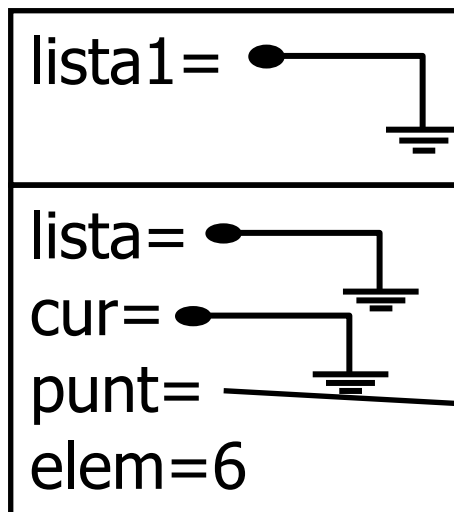



```

int main() {
    ListaDiElem lista1;
    ... ..
    lista1 = InsInFondo( lista1, 6 );
    ... ..
}

ListaDiElem InsInFondo( ListaDiElem lista, int elem ) {
    ListaDiElem punt, cur = lista;
    punt = (ListaDiElem) malloc( sizeof(ElemLista) );
    punt->prox = NULL;
    punt->info = elem;
    if ( lista==NULL )
        return punt;
    else {
        while( cur ->prox != NULL )
            cur = cur->prox;
        cur ->prox = punt;
    }
    return lista;
}

```

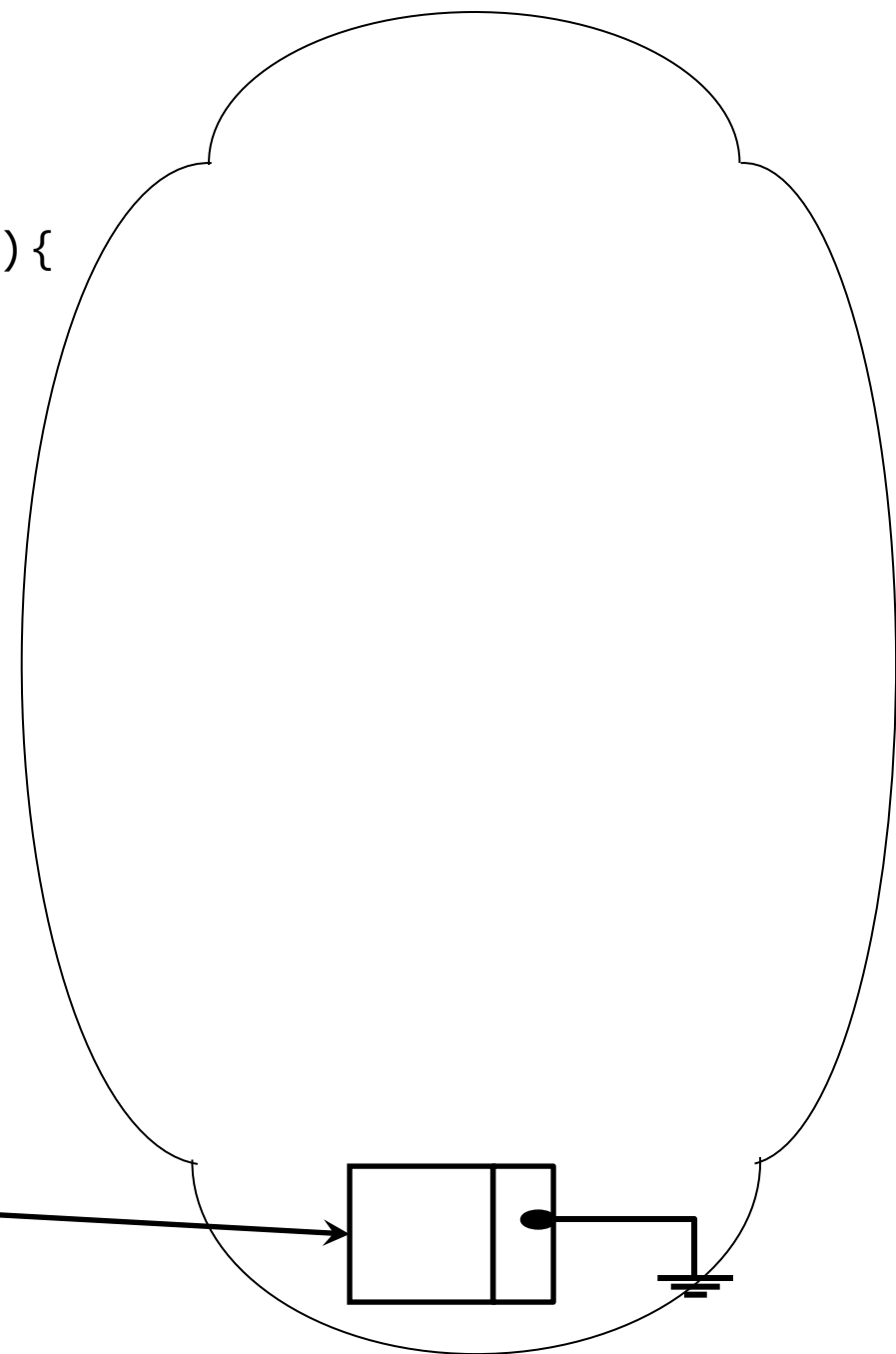
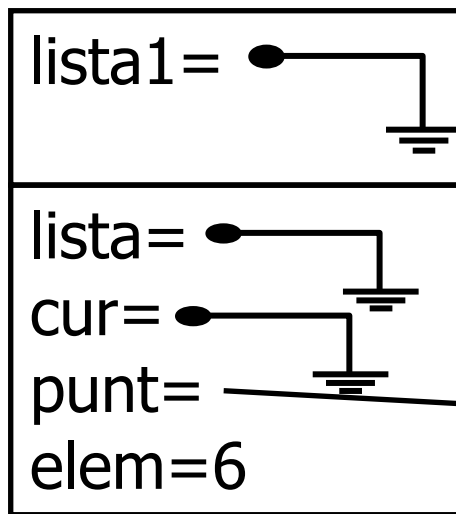


```

int main() {
    ListaDiElem lista1;
    ... ..
    lista1 = InsInFondo( lista1, 6 );
    ... ..
}

ListaDiElem InsInFondo( ListaDiElem lista, int elem ) {
    ListaDiElem punt, cur = lista;
    punt = (ListaDiElem) malloc( sizeof(ElemLista) );
    punt->prox = NULL;
    punt->info = elem;
    if ( lista==NULL )
        return punt;
    else {
        while( cur ->prox != NULL )
            cur = cur->prox;
        cur ->prox = punt;
    }
    return lista;
}

```

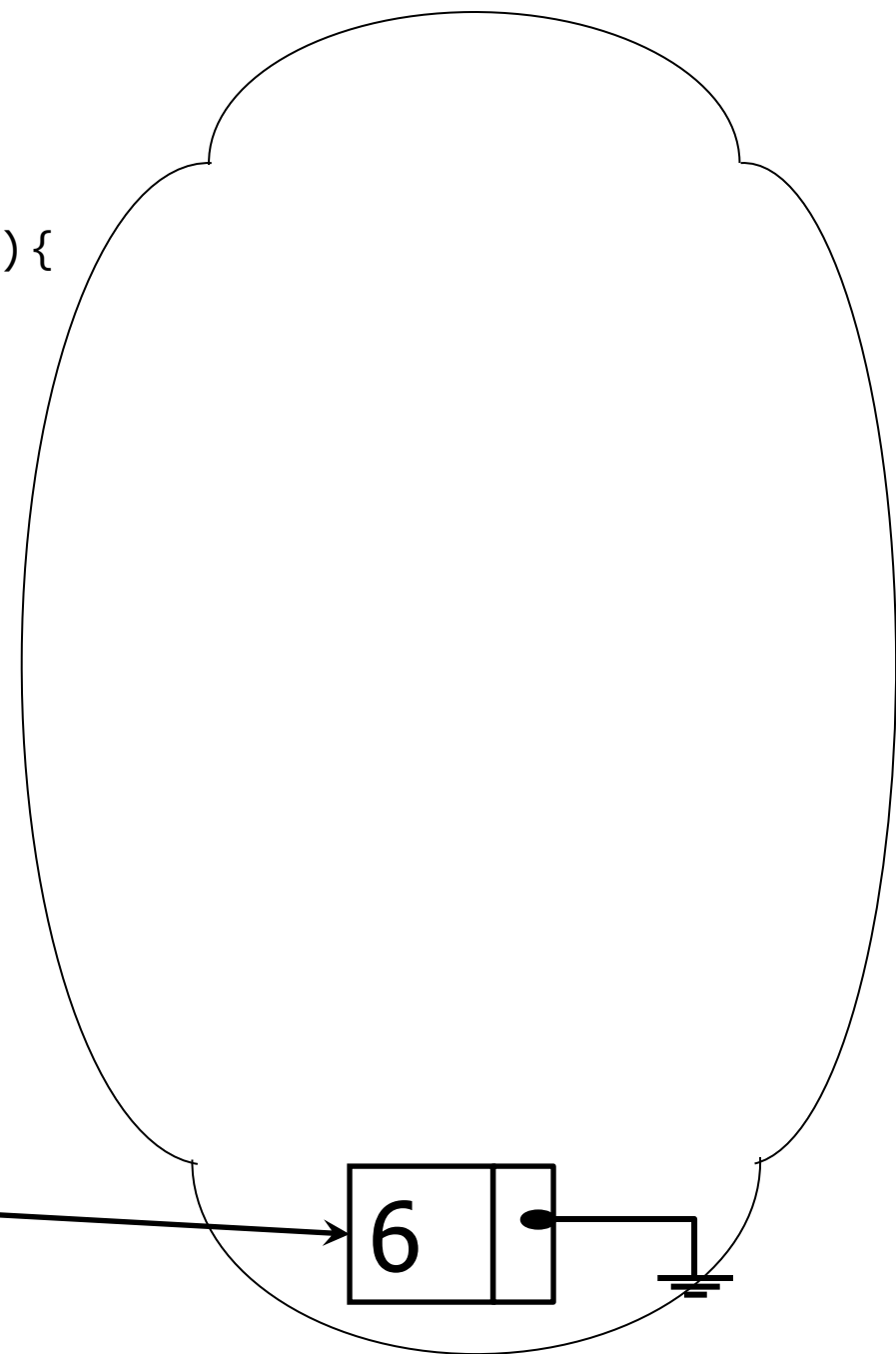
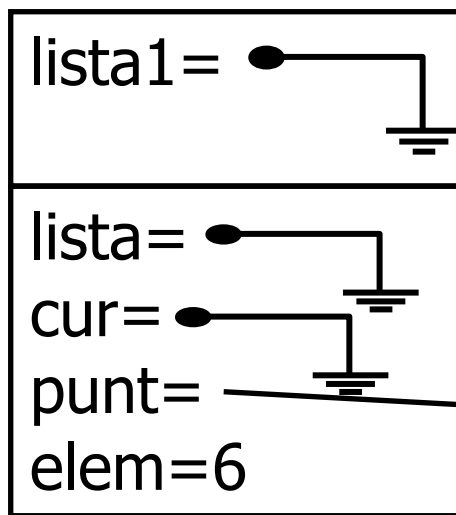


```

int main() {
    ListaDiElem lista1;
    ... ..
    lista1 = InsInFondo( lista1, 6 );
    ... ..
}

ListaDiElem InsInFondo( ListaDiElem lista, int elem ) {
    ListaDiElem punt, cur = lista;
    punt = (ListaDiElem) malloc( sizeof(ElemLista) );
    punt->prox = NULL;
    punt->info = elem;
    if ( lista==NULL )
        return punt;
    else {
        while( cur ->prox != NULL )
            cur = cur->prox;
        cur ->prox = punt;
    }
    return lista;
}

```

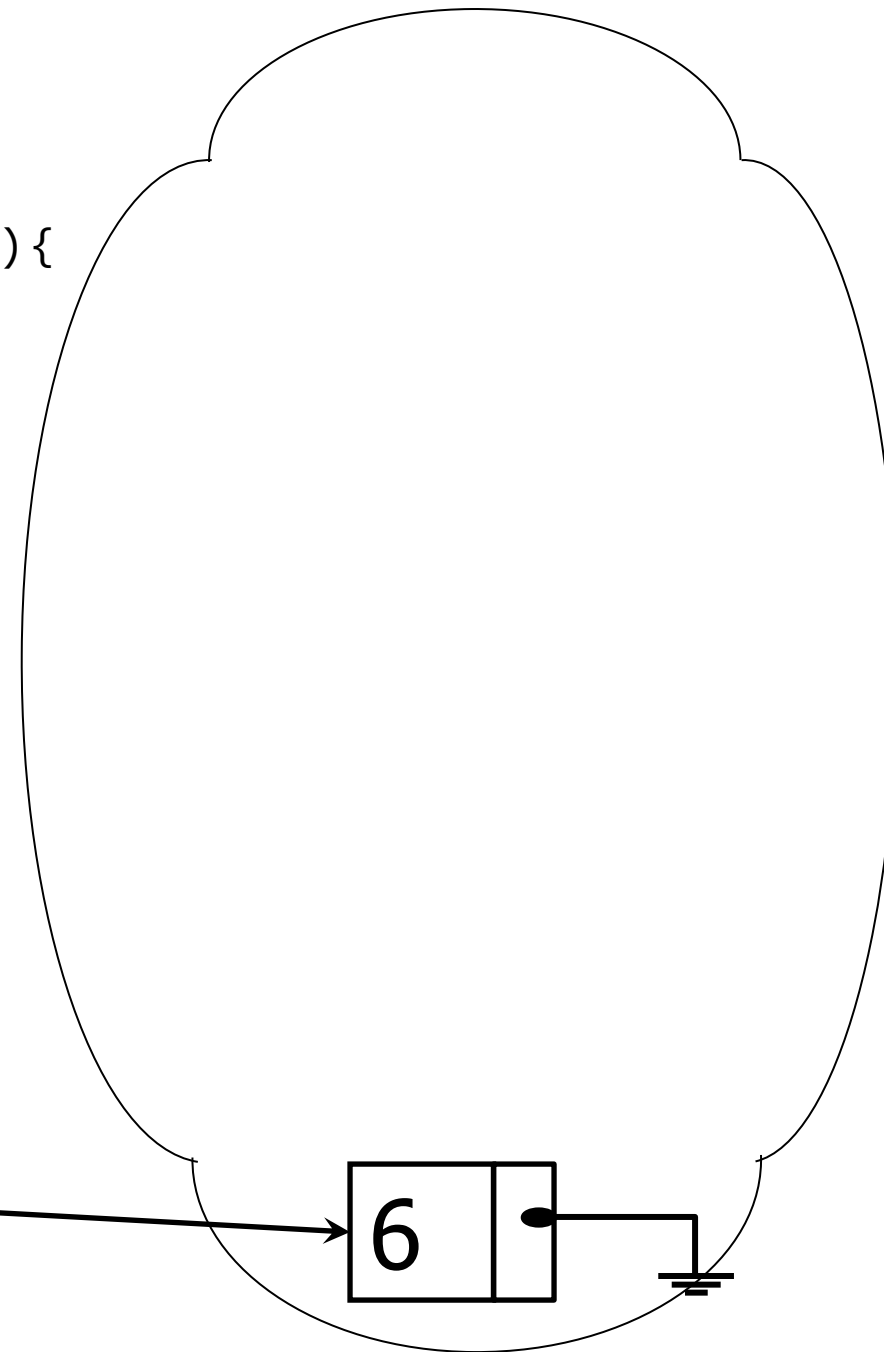
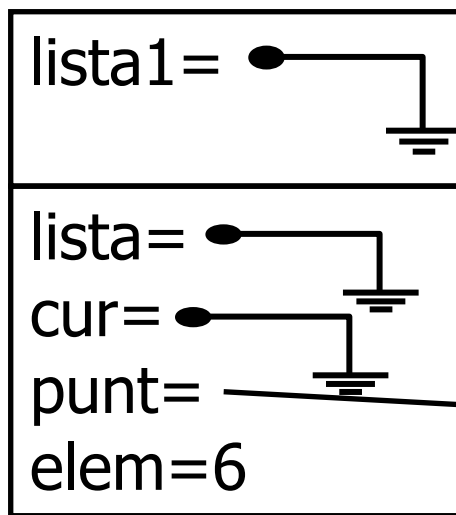


```

int main() {
    ListaDiElem lista1;
    ... ..
    lista1 = InsInFondo( lista1, 6 );
    ... ..
}

ListaDiElem InsInFondo( ListaDiElem lista, int elem ) {
    ListaDiElem punt, cur = lista;
    punt = (ListaDiElem) malloc( sizeof(ElemLista) );
    punt->prox = NULL;
    punt->info = elem;
    if ( lista==NULL )
        return punt;
    else {
        while( cur ->prox != NULL )
            cur = cur->prox;
        cur ->prox = punt;
    }
    return lista;
}

```

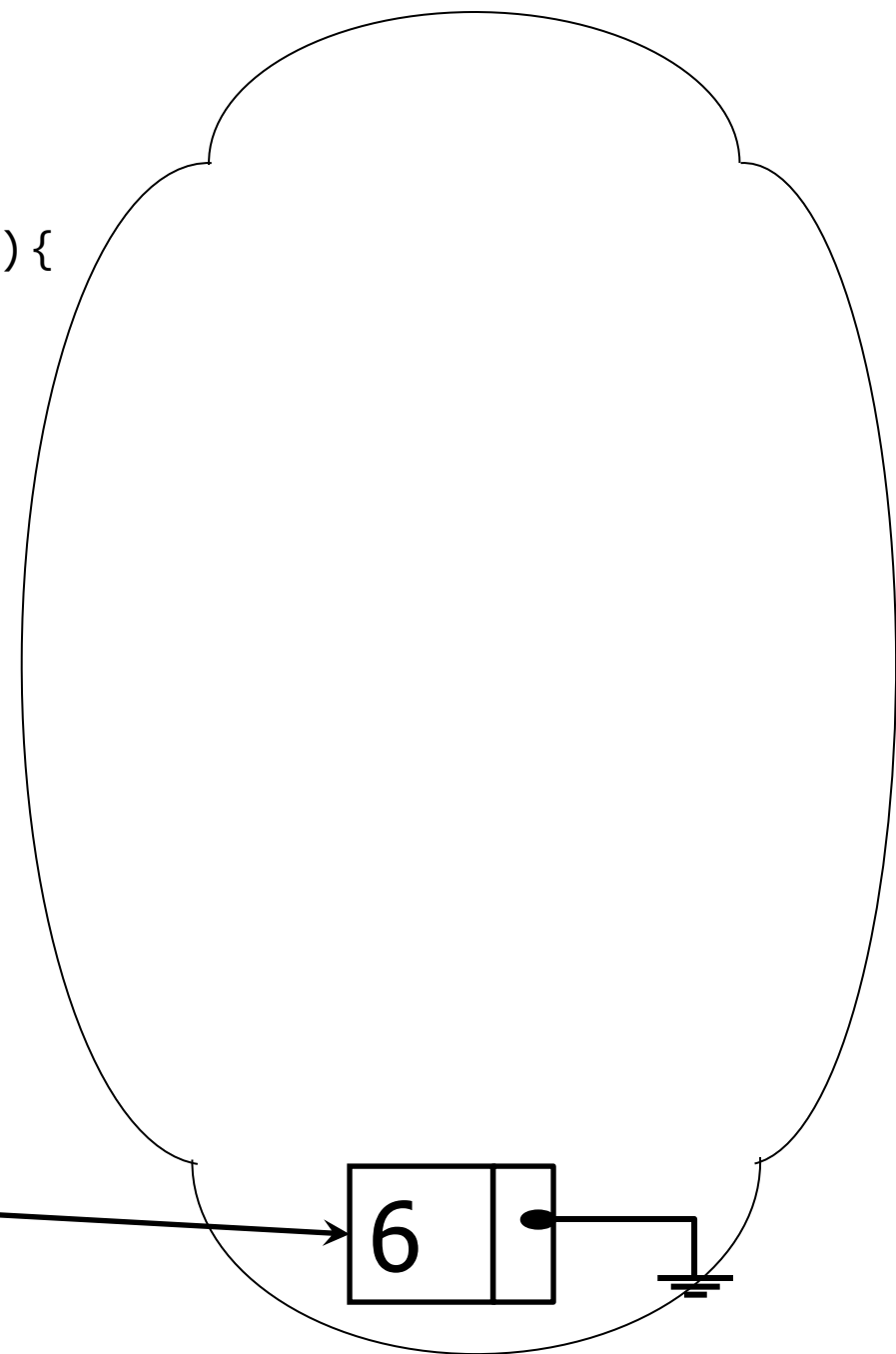
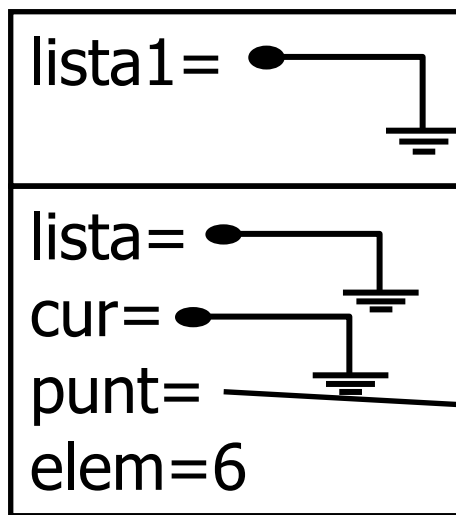


```

int main() {
    ListaDiElem lista1;
    ... ..
    lista1 = InsInFondo( lista1, 6 );
    ... ..
}

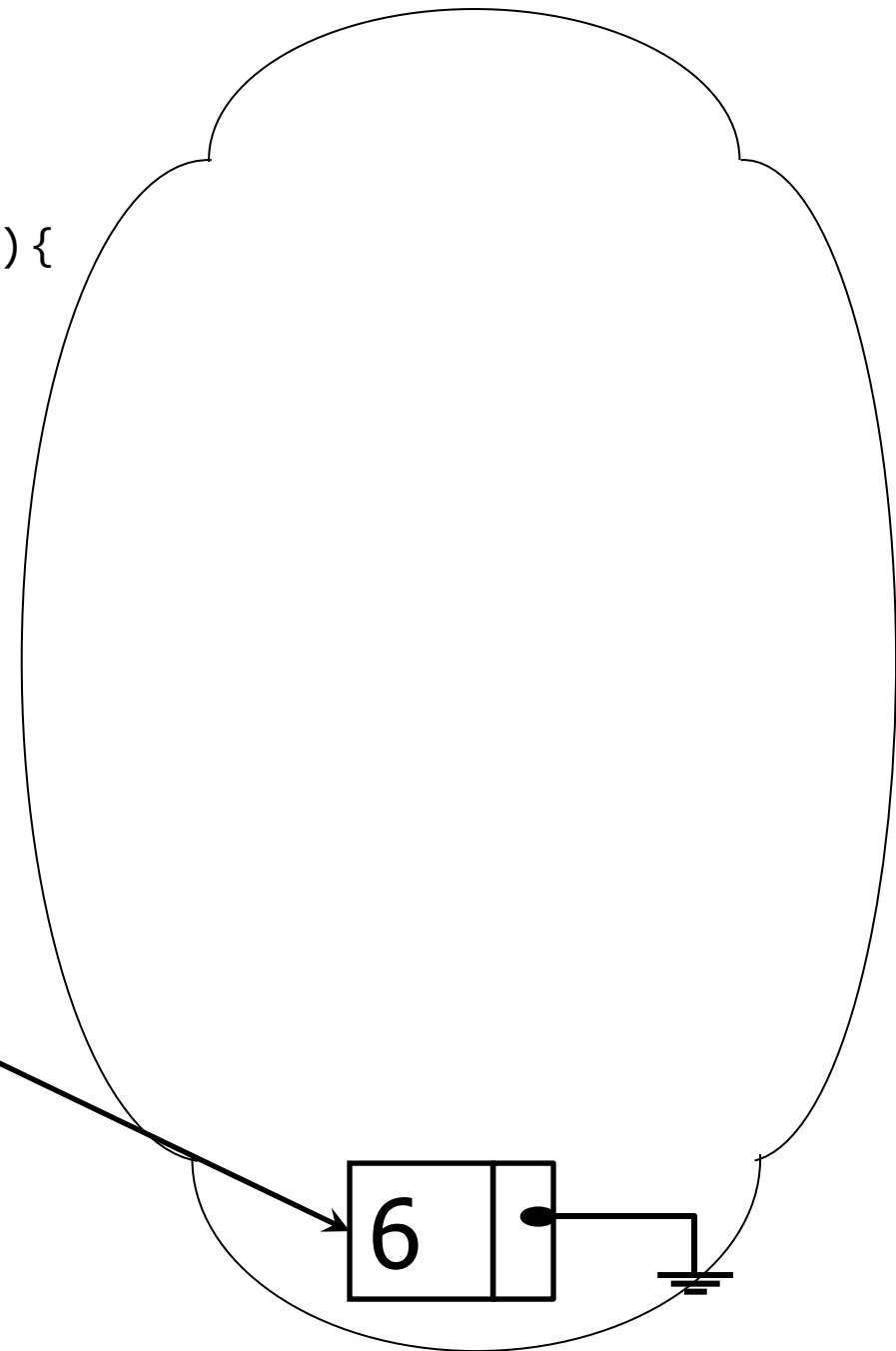
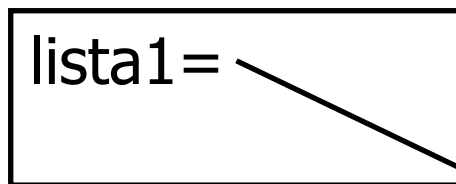
ListaDiElem InsInFondo( ListaDiElem lista, int elem ) {
    ListaDiElem punt, cur = lista;
    punt = (ListaDiElem) malloc( sizeof(ElemLista) );
    punt->prox = NULL;
    punt->info = elem;
    if ( lista==NULL )
        return punt;
    else {
        while( cur ->prox != NULL )
            cur = cur->prox;
        cur ->prox = punt;
    }
    return lista;
}

```



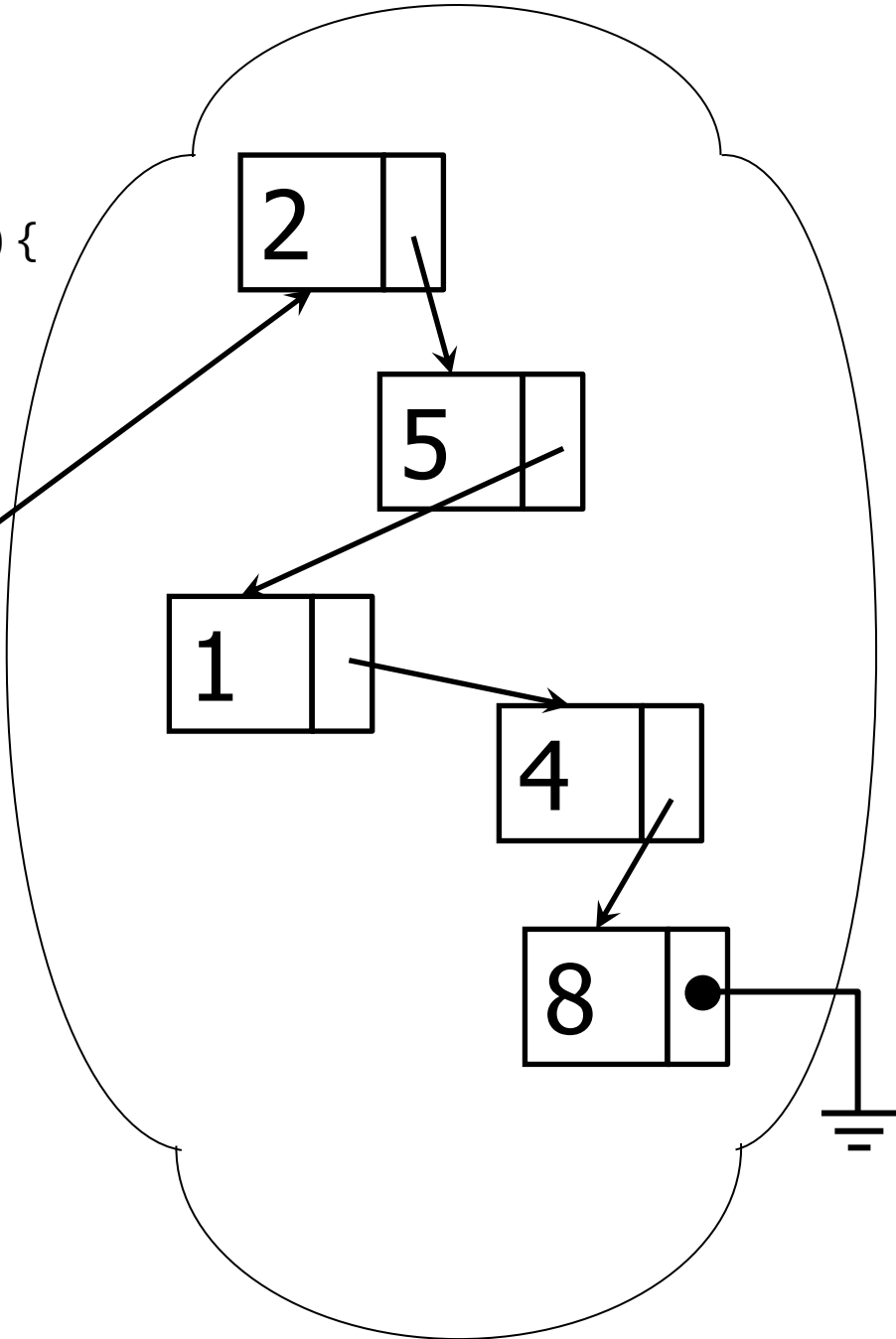
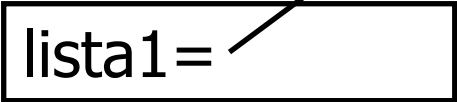
```
int main() {
    ListaDiElem lista1;
    ... ..
    lista1 = InsInFondo( lista1, 6 );
    ... ..
}

ListaDiElem InsInFondo( ListaDiElem lista, int elem ) {
    ListaDiElem punt, cur = lista;
    punt = (ListaDiElem) malloc( sizeof(ElemLista) );
    punt->prox = NULL;
    punt->info = elem;
    if ( lista==NULL )
        return punt;
    else {
        while( cur ->prox != NULL )
            cur = cur->prox;
        cur ->prox = punt;
    }
    return lista;
}
```



```
int main() {
    ListaDiElem lista1;
    ... ..
    lista1 = InsInFondo( lista1, 6 );
    ... ..
}

ListaDiElem InsInFondo( ListaDiElem lista, int elem ) {
    ListaDiElem punt, cur = lista;
    punt = (ListaDiElem) malloc( sizeof(ElemLista) );
    punt->prox = NULL;
    punt->info = elem;
    if ( lista==NULL )
        return punt;
    else {
        while( cur ->prox != NULL )
            cur = cur->prox;
        cur ->prox = punt;
    }
    return lista;
}
```

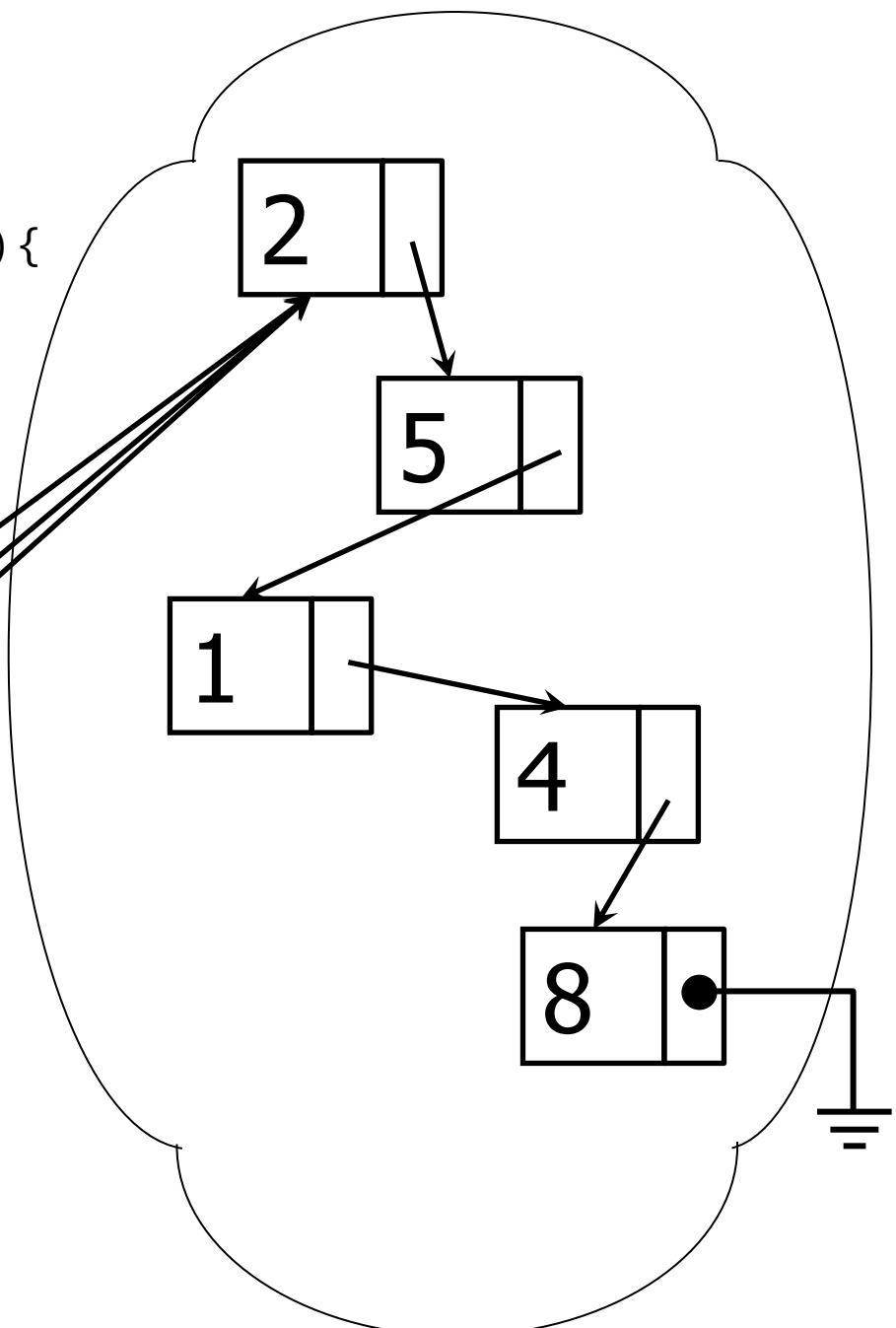
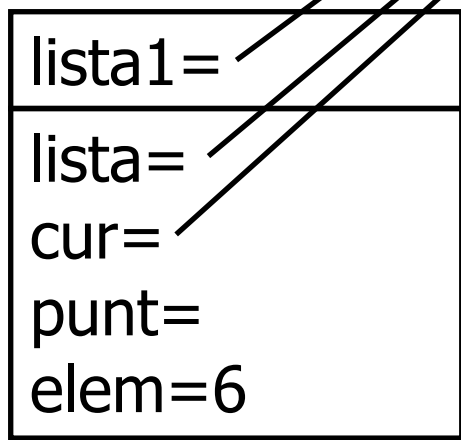


```

int main() {
    ListaDiElem lista1;
    ... ..
    lista1 = InsInFondo( lista1, 6 );
    ... ..
}

ListaDiElem InsInFondo( ListaDiElem lista, int elem ) {
    ListaDiElem punt, cur = lista;
    punt = (ListaDiElem) malloc( sizeof(ElemLista) );
    punt->prox = NULL;
    punt->info = elem;
    if ( lista==NULL )
        return punt;
    else {
        while( cur->prox != NULL )
            cur = cur->prox;
        cur->prox = punt;
    }
    return lista;
}

```

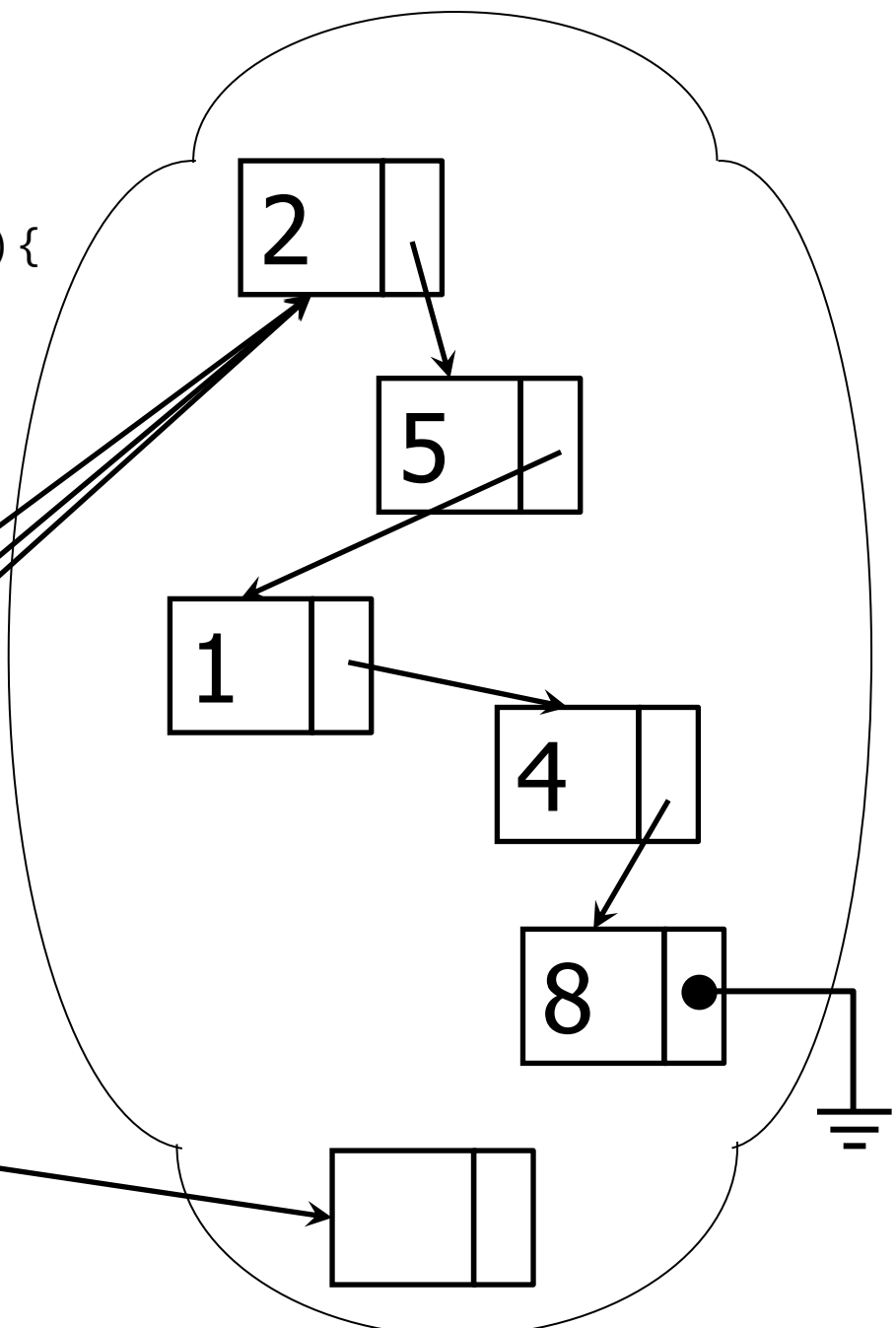
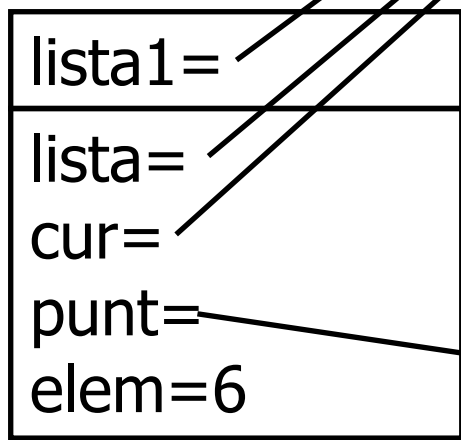



```

int main() {
    ListaDiElem lista1;
    ... ..
    lista1 = InsInFondo( lista1, 6 );
    ... ..
}

ListaDiElem InsInFondo( ListaDiElem lista, int elem ) {
    ListaDiElem punt, cur = lista;
    punt = (ListaDiElem) malloc( sizeof(ElemLista) );
    punt->prox = NULL;
    punt->info = elem;
    if ( lista==NULL )
        return punt;
    else {
        while( cur->prox != NULL )
            cur = cur->prox;
        cur->prox = punt;
    }
    return lista;
}

```

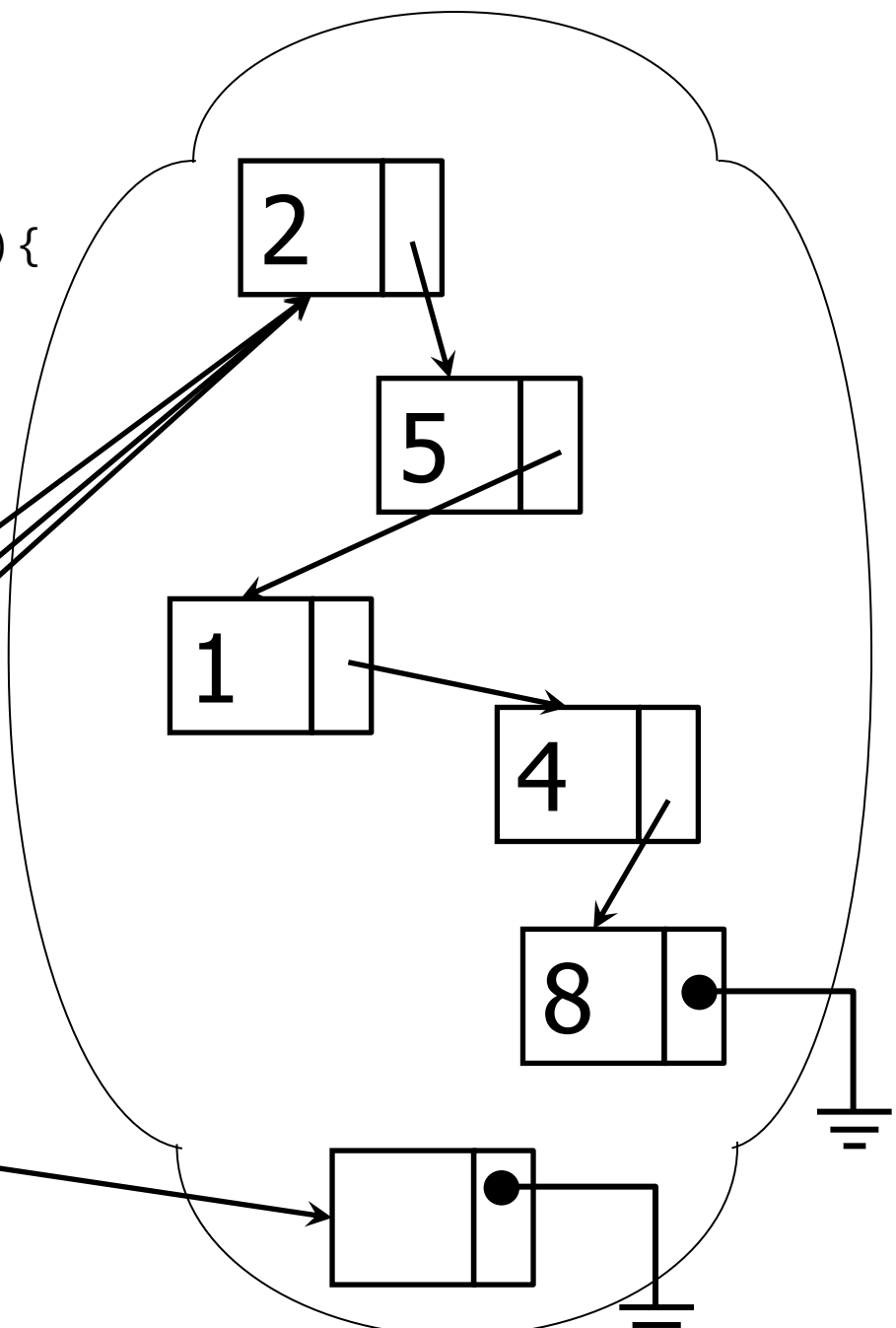
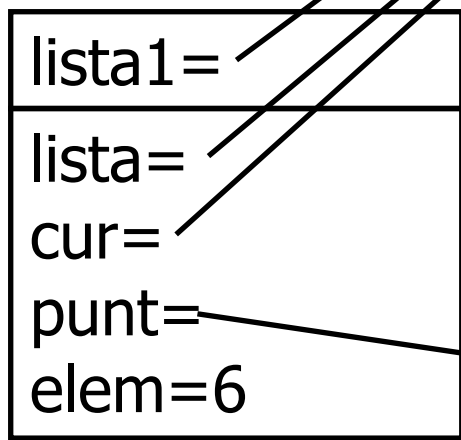


```

int main() {
    ListaDiElem lista1;
    ... ..
    lista1 = InsInFondo( lista1, 6 );
    ... ..
}

ListaDiElem InsInFondo( ListaDiElem lista, int elem ) {
    ListaDiElem punt, cur = lista;
    punt = (ListaDiElem) malloc( sizeof(ElemLista) );
    punt->prox = NULL;
    punt->info = elem;
    if ( lista==NULL )
        return punt;
    else {
        while( cur->prox != NULL )
            cur = cur->prox;
        cur->prox = punt;
    }
    return lista;
}

```

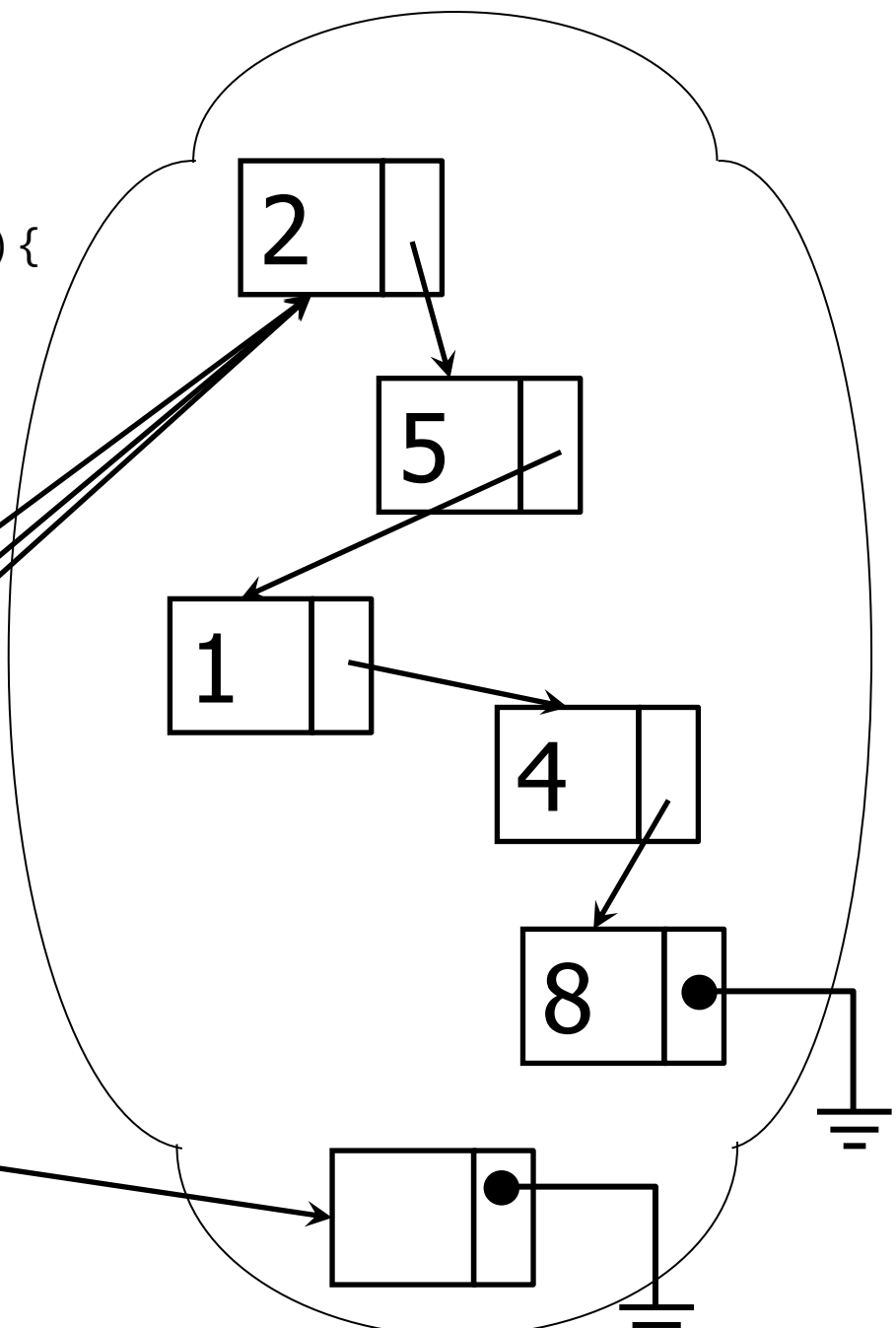
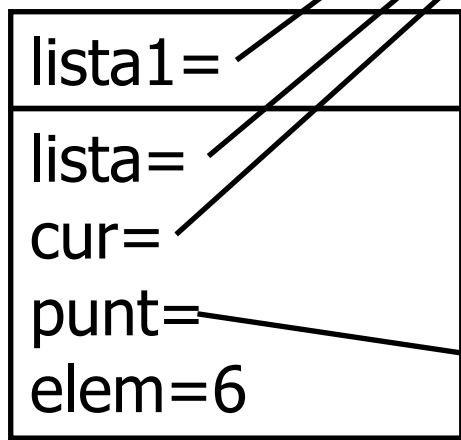


```

int main() {
    ListaDiElem lista1;
    ... ..
    lista1 = InsInFondo( lista1, 6 );
    ... ..
}

ListaDiElem InsInFondo( ListaDiElem lista, int elem ) {
    ListaDiElem punt, cur = lista;
    punt = (ListaDiElem) malloc( sizeof(ElemLista) );
    punt->prox = NULL;
    punt->info = elem;
    if ( lista==NULL )
        return punt;
    else {
        while( cur->prox != NULL )
            cur = cur->prox;
        cur->prox = punt;
    }
    return lista;
}

```

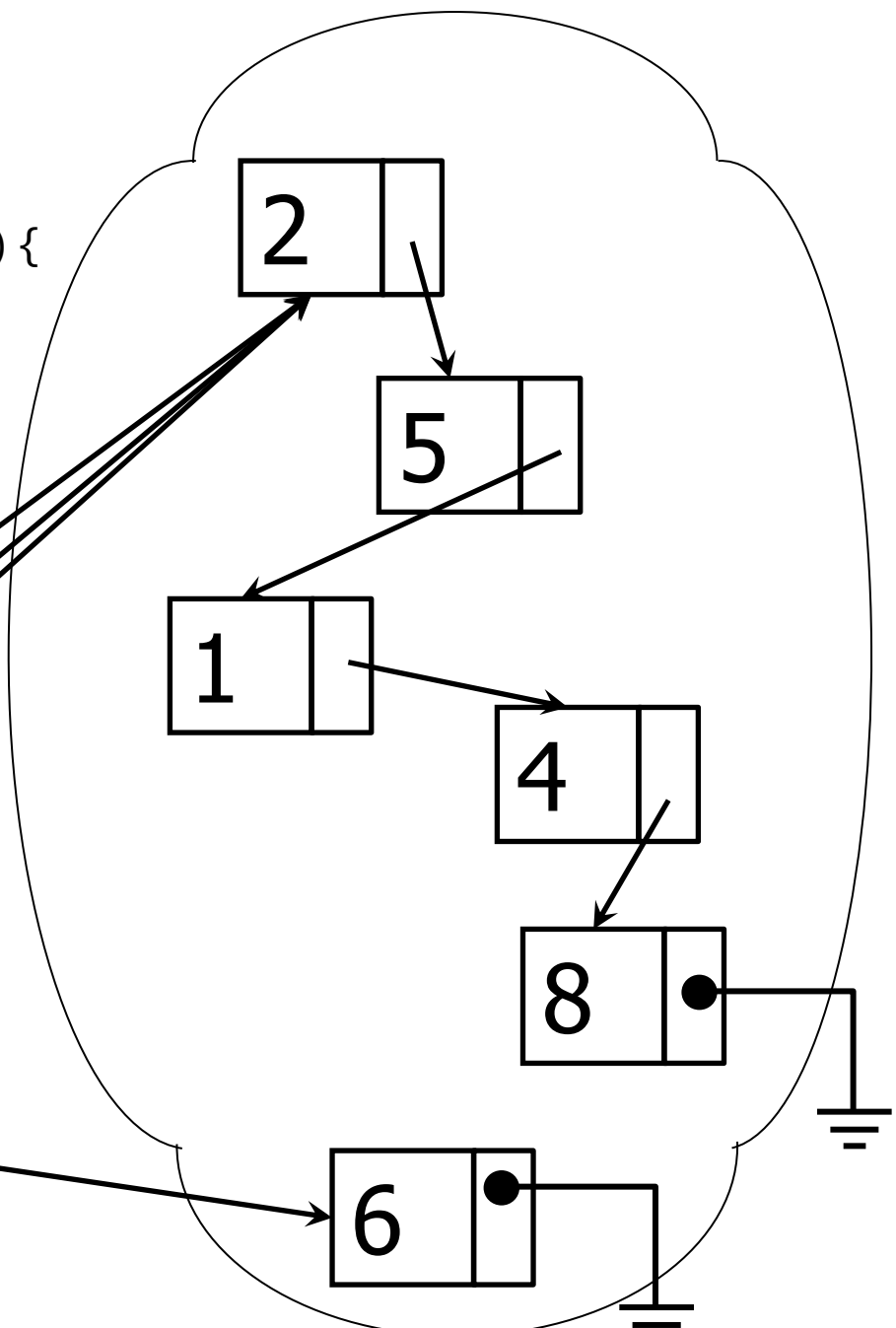
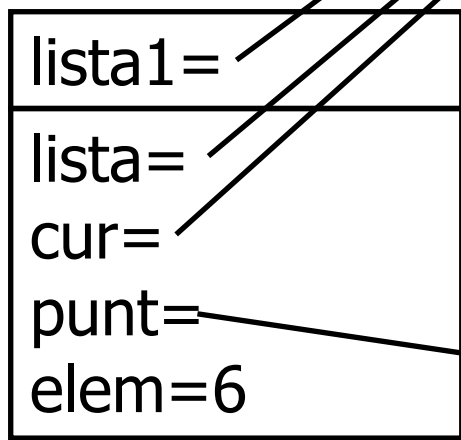


```

int main() {
    ListaDiElem lista1;
    ... ..
    lista1 = InsInFondo( lista1, 6 );
    ... ..
}

ListaDiElem InsInFondo( ListaDiElem lista, int elem ) {
    ListaDiElem punt, cur = lista;
    punt = (ListaDiElem) malloc( sizeof(ElemLista) );
    punt->prox = NULL;
    punt->info = elem;
    if ( lista==NULL )
        return punt;
    else {
        while( cur->prox != NULL )
            cur = cur->prox;
        cur->prox = punt;
    }
    return lista;
}

```

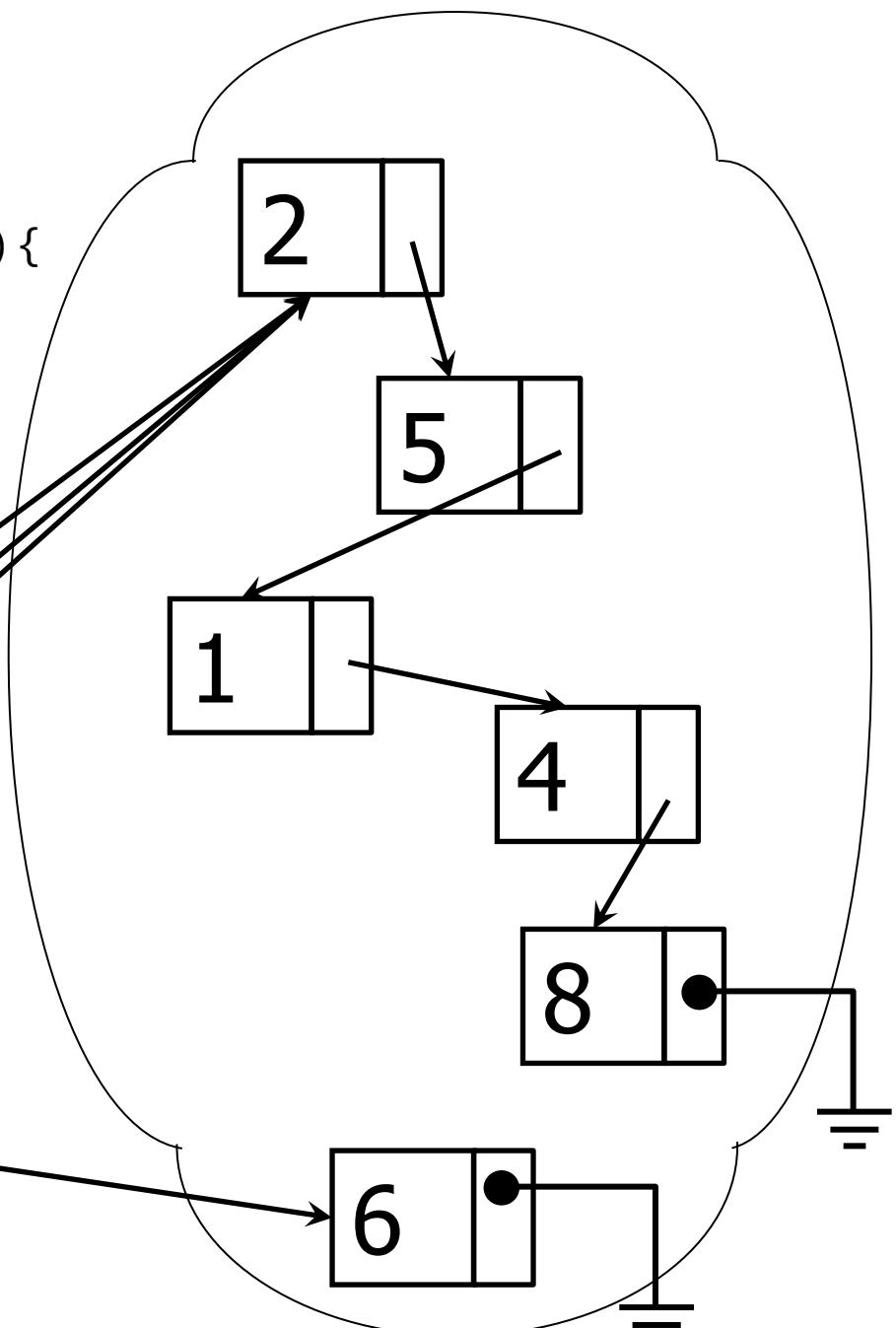
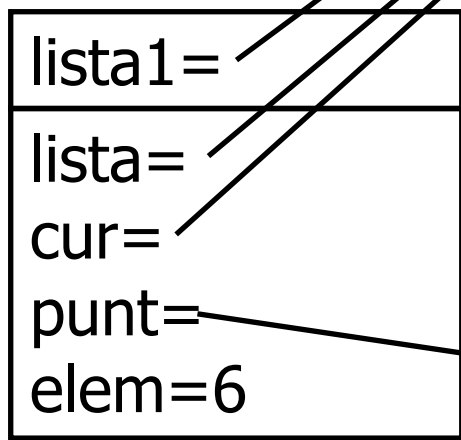


```

int main() {
    ListaDiElem lista1;
    ... ..
    lista1 = InsInFondo( lista1, 6 );
    ... ..
}

ListaDiElem InsInFondo( ListaDiElem lista, int elem ) {
    ListaDiElem punt, cur = lista;
    punt = (ListaDiElem) malloc( sizeof(ElemLista) );
    punt->prox = NULL;
    punt->info = elem;
    if ( lista==NULL )
        return punt;
    else {
        while( cur->prox != NULL )
            cur = cur->prox;
        cur->prox = punt;
    }
    return lista;
}

```

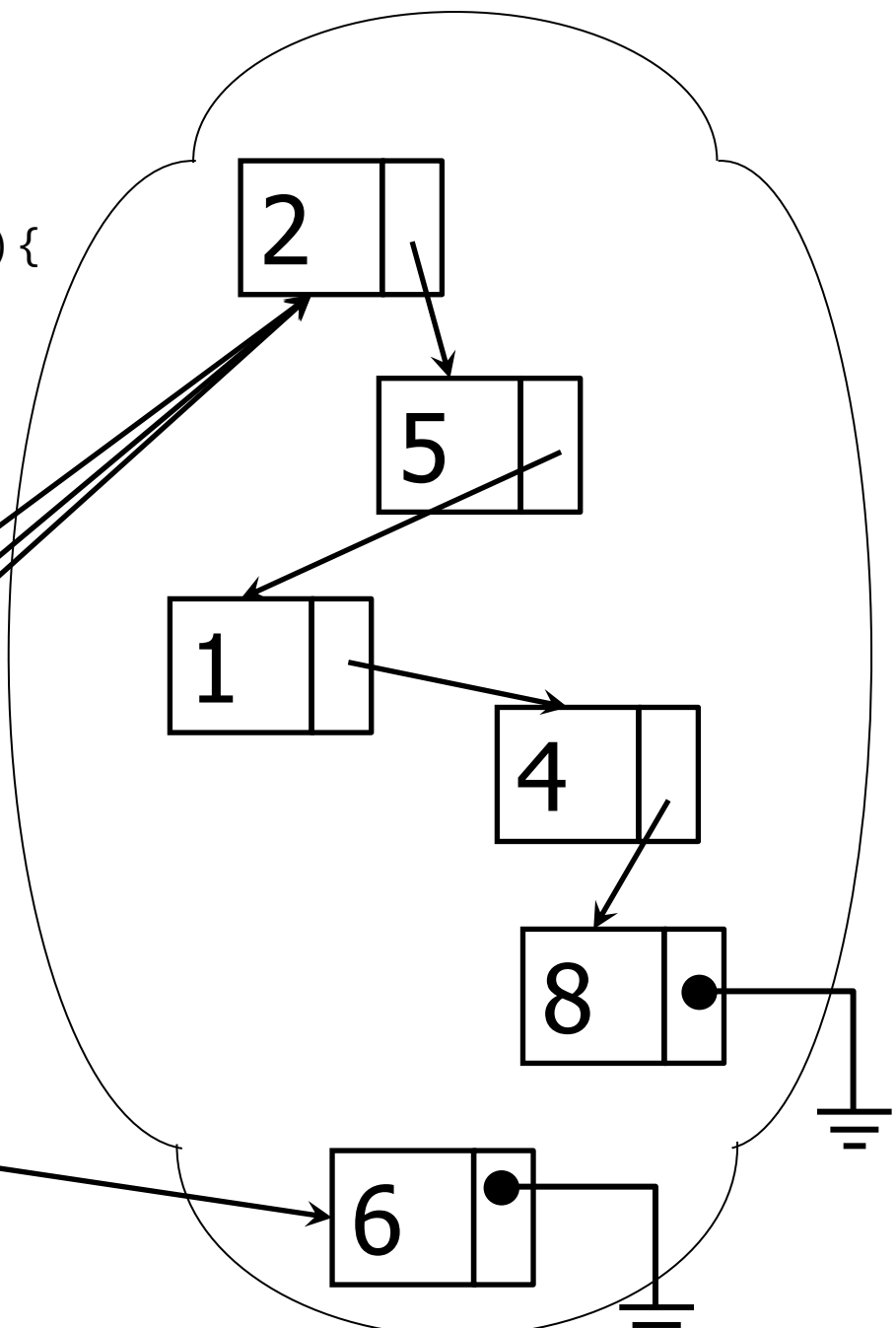
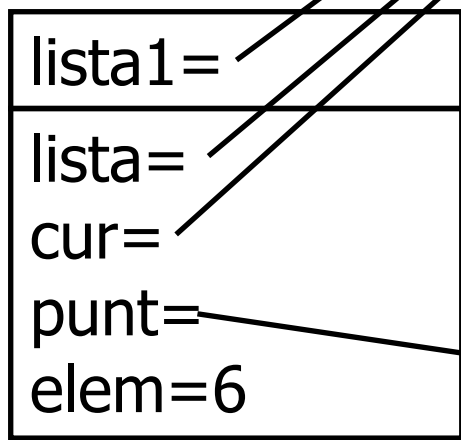


```

int main() {
    ListaDiElem lista1;
    ... ..
    lista1 = InsInFondo( lista1, 6 );
    ... ..
}

ListaDiElem InsInFondo( ListaDiElem lista, int elem ) {
    ListaDiElem punt, cur = lista;
    punt = (ListaDiElem) malloc( sizeof(ElemLista) );
    punt->prox = NULL;
    punt->info = elem;
    if ( lista==NULL )
        return punt;
    else {
        while( cur->prox != NULL )
            cur = cur->prox;
        cur->prox = punt;
    }
    return lista;
}

```

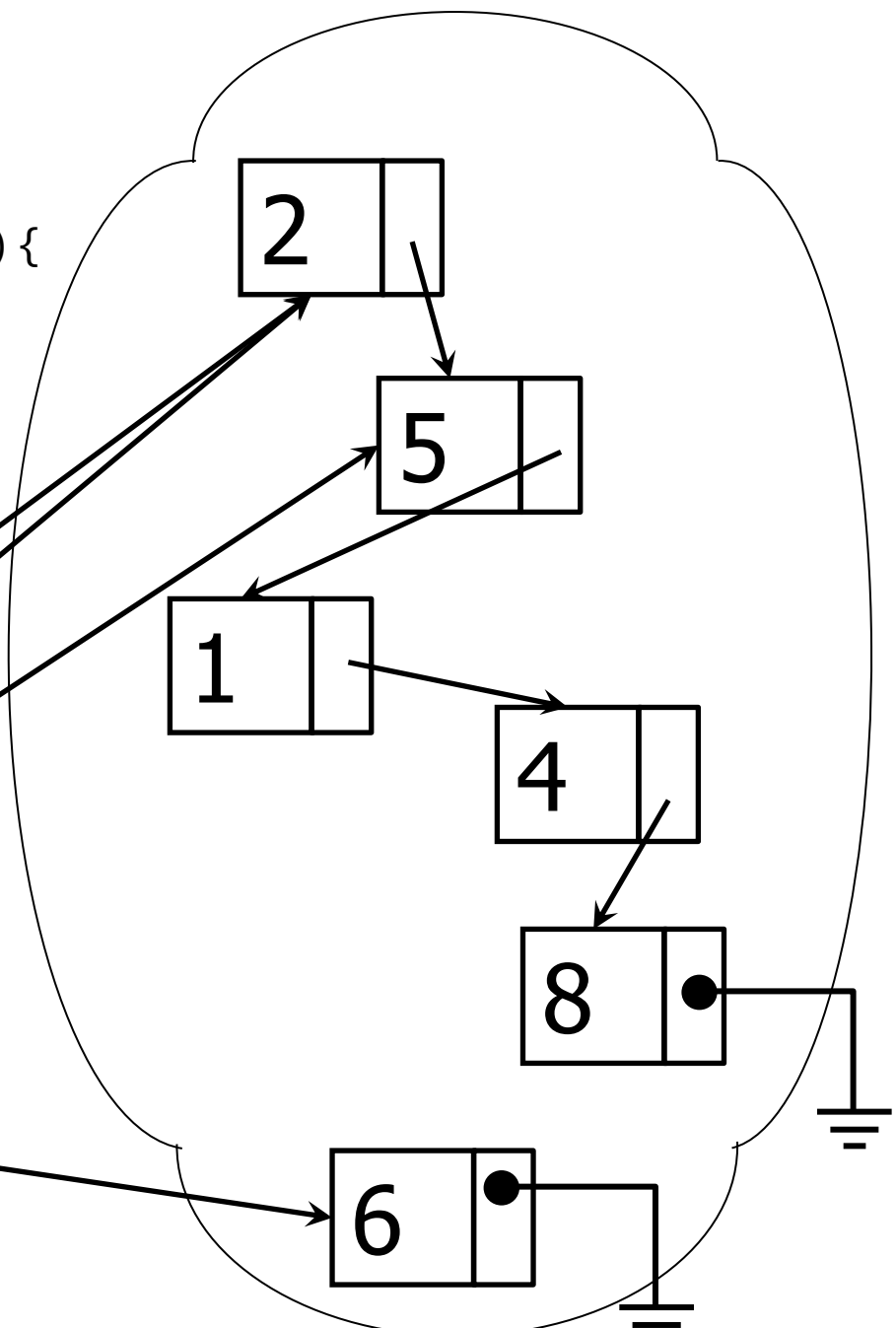
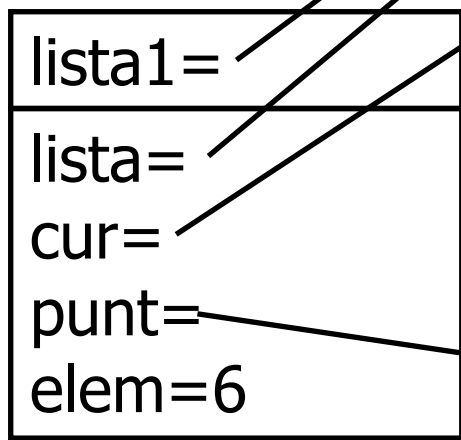


```

int main() {
    ListaDiElem lista1;
    ... ..
    lista1 = InsInFondo( lista1, 6 );
    ... ..
}

ListaDiElem InsInFondo( ListaDiElem lista, int elem ) {
    ListaDiElem punt, cur = lista;
    punt = (ListaDiElem) malloc( sizeof(ElemLista) );
    punt->prox = NULL;
    punt->info = elem;
    if ( lista==NULL )
        return punt;
    else {
        while( cur->prox != NULL )
            cur = cur->prox;
        cur->prox = punt;
    }
    return lista;
}

```

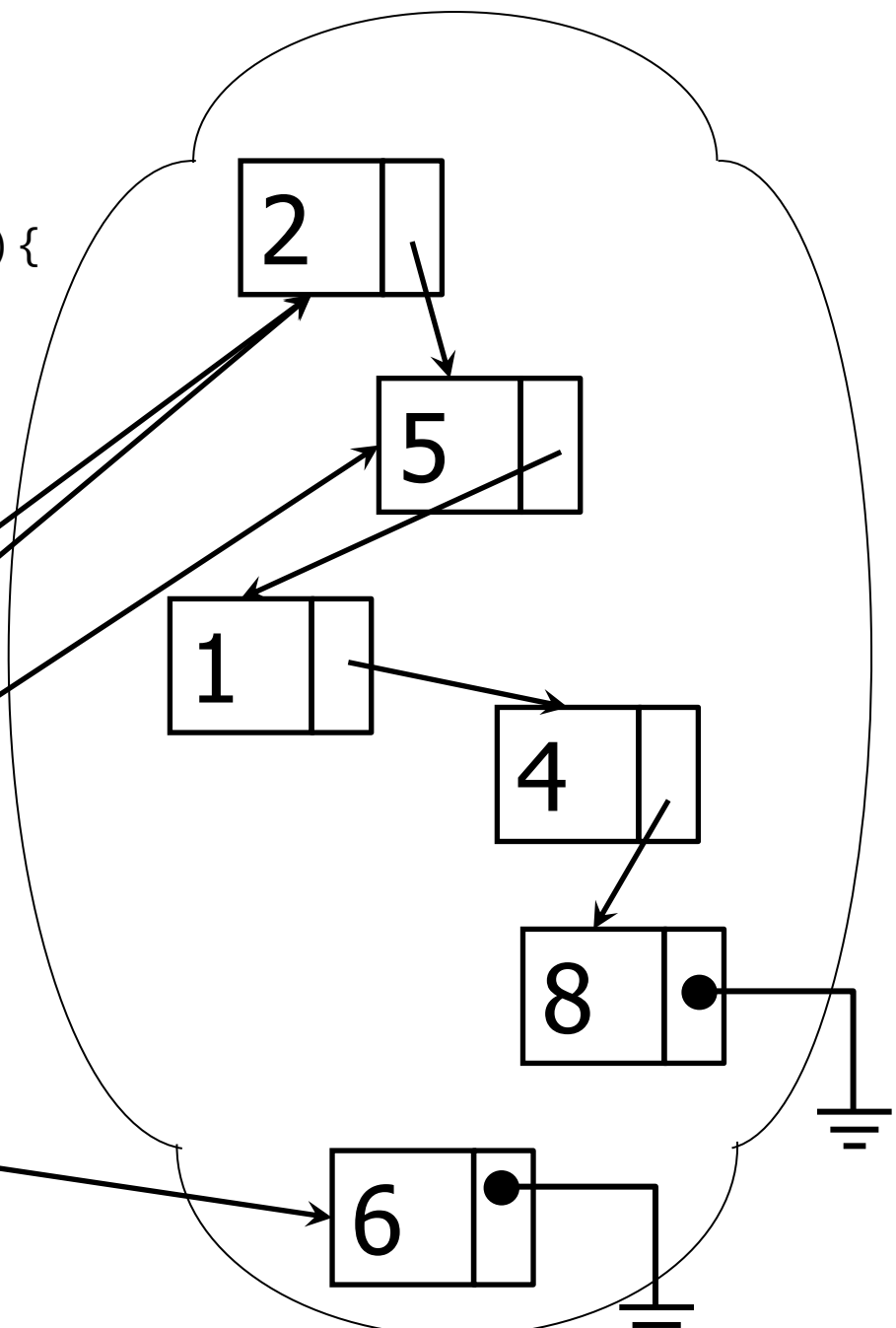
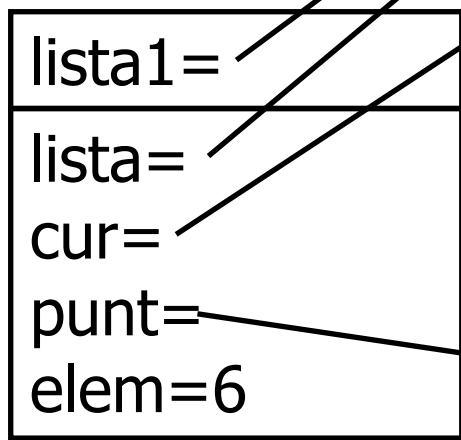


```

int main() {
    ListaDiElem lista1;
    ... ..
    lista1 = InsInFondo( lista1, 6 );
    ... ..
}

ListaDiElem InsInFondo( ListaDiElem lista, int elem ) {
    ListaDiElem punt, cur = lista;
    punt = (ListaDiElem) malloc( sizeof(ElemLista) );
    punt->prox = NULL;
    punt->info = elem;
    if ( lista==NULL )
        return punt;
    else {
        while( cur->prox != NULL )
            cur = cur->prox;
        cur->prox = punt;
    }
    return lista;
}

```

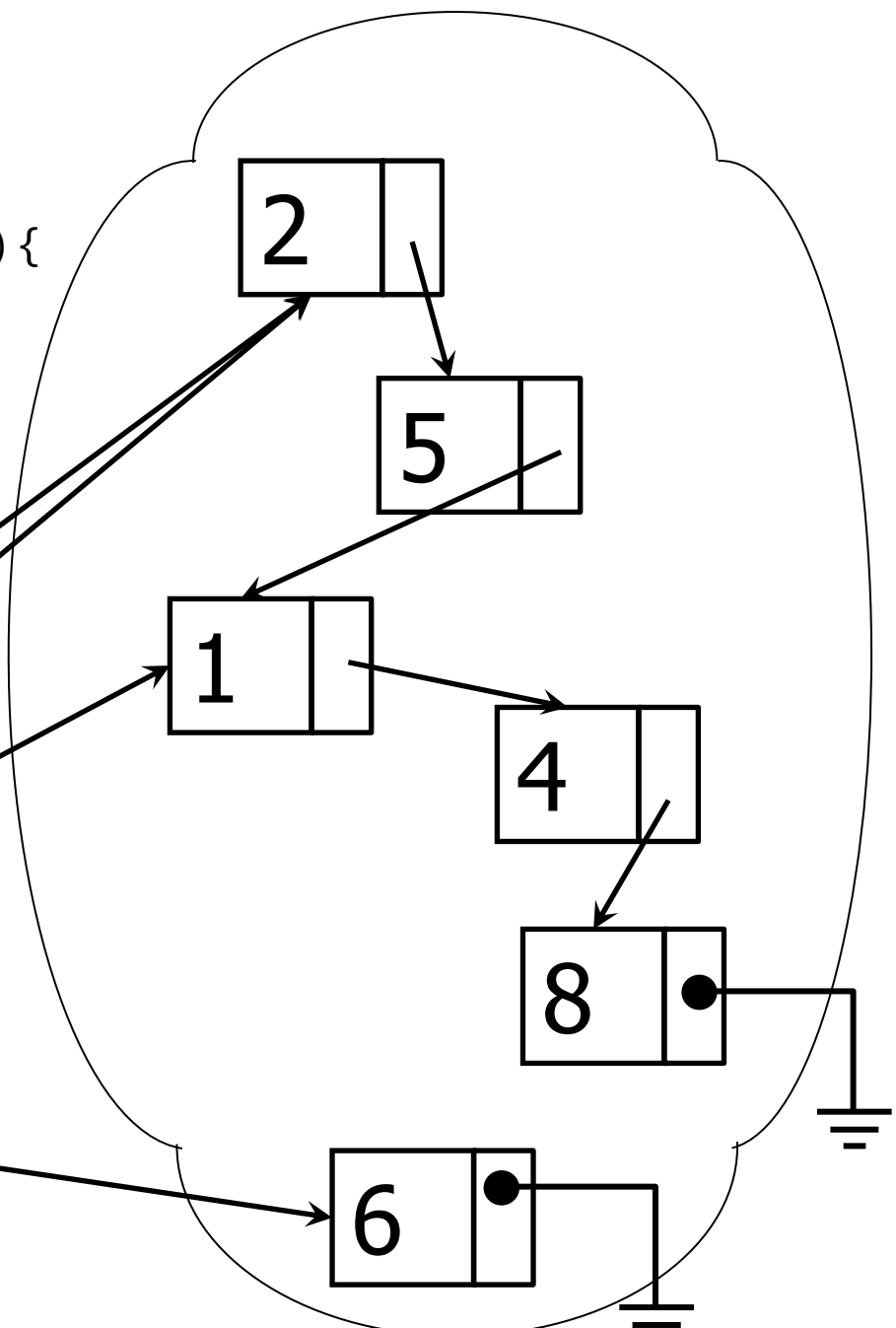
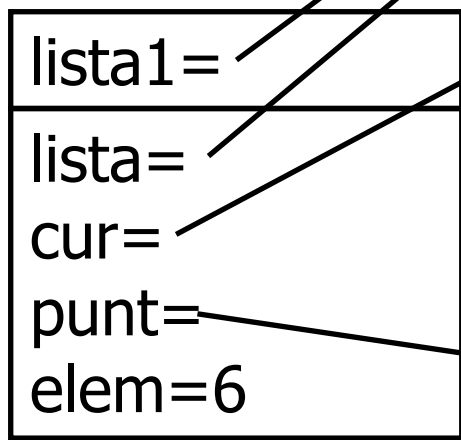



```

int main() {
    ListaDiElem lista1;
    ... ..
    lista1 = InsInFondo( lista1, 6 );
    ... ..
}

ListaDiElem InsInFondo( ListaDiElem lista, int elem ) {
    ListaDiElem punt, cur = lista;
    punt = (ListaDiElem) malloc( sizeof(ElemLista) );
    punt->prox = NULL;
    punt->info = elem;
    if ( lista==NULL )
        return punt;
    else {
        while( cur->prox != NULL )
            cur = cur->prox;
        cur->prox = punt;
    }
    return lista;
}

```

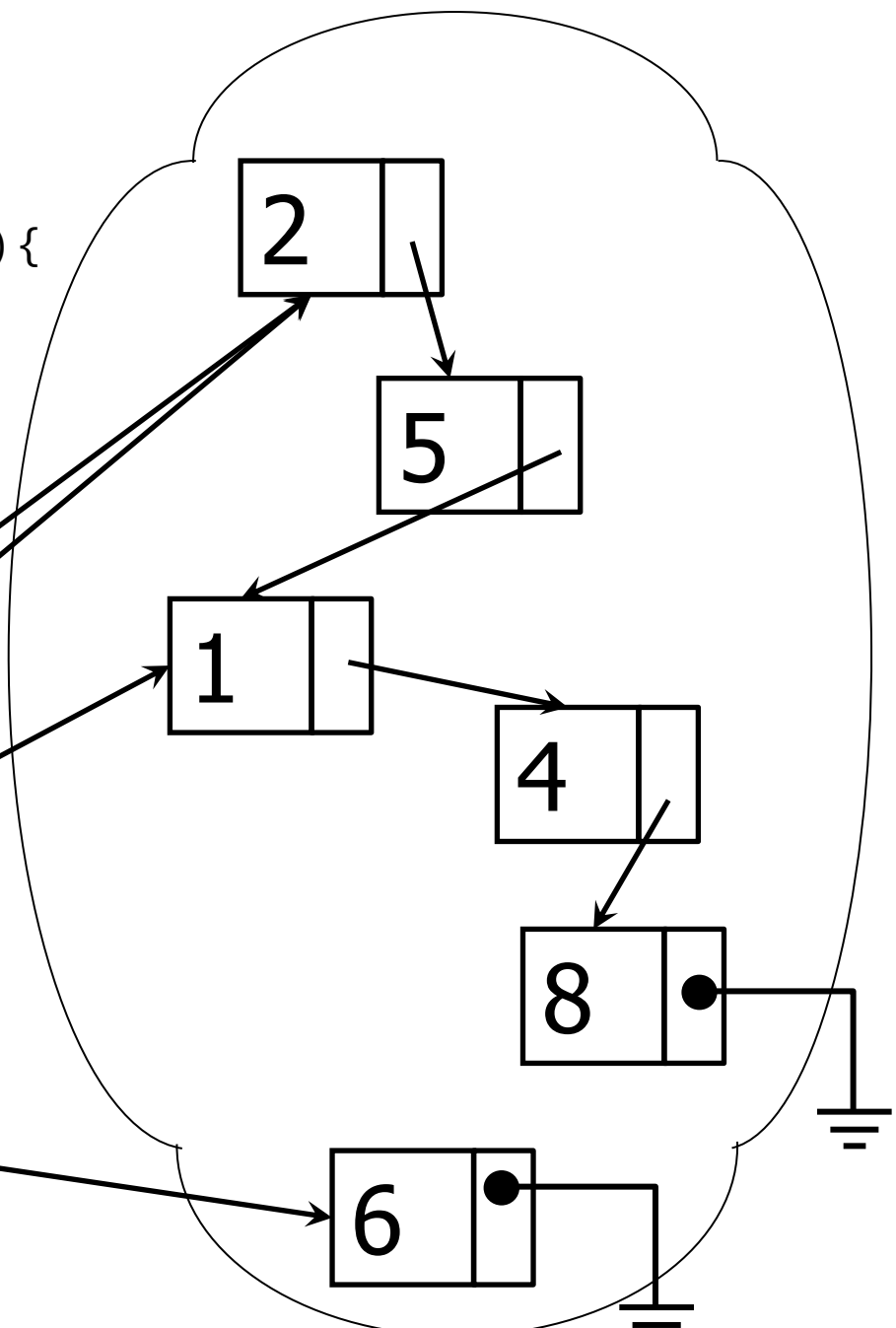
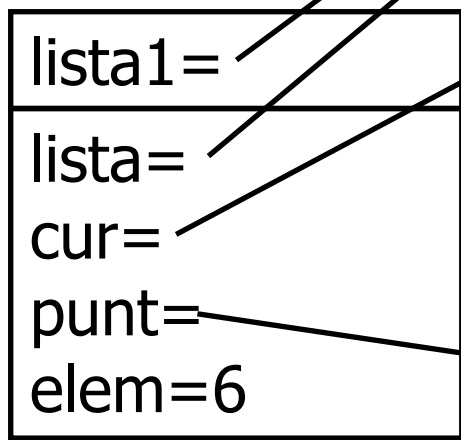


```

int main() {
    ListaDiElem lista1;
    ... ..
    lista1 = InsInFondo( lista1, 6 );
    ... ..
}

ListaDiElem InsInFondo( ListaDiElem lista, int elem ) {
    ListaDiElem punt, cur = lista;
    punt = (ListaDiElem) malloc( sizeof(ElemLista) );
    punt->prox = NULL;
    punt->info = elem;
    if ( lista==NULL )
        return punt;
    else {
        while( cur->prox != NULL )
            cur = cur->prox;
        cur->prox = punt;
    }
    return lista;
}

```

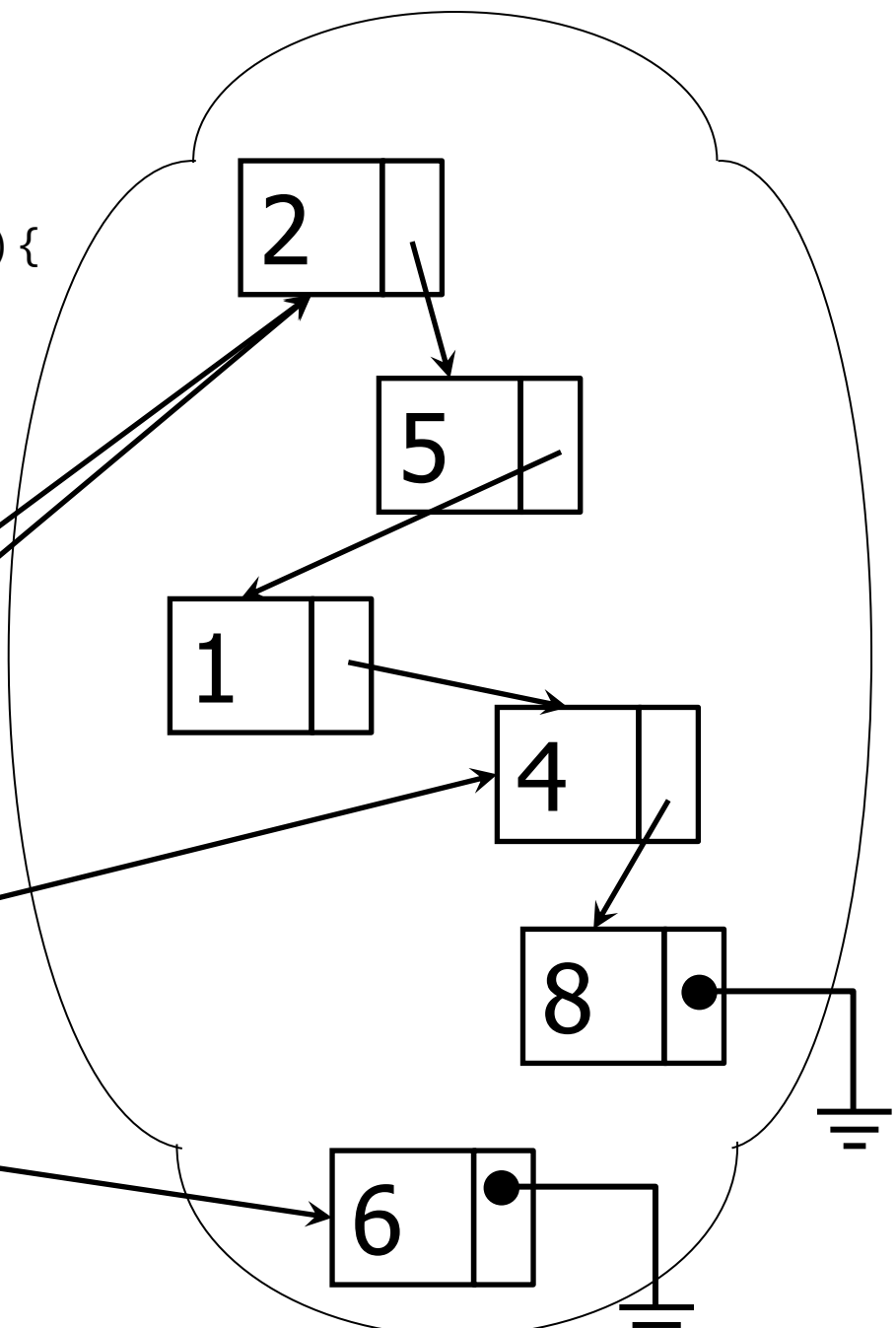
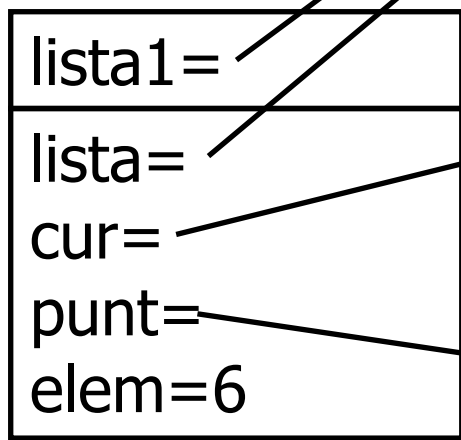


```

int main() {
    ListaDiElem lista1;
    ... ..
    lista1 = InsInFondo( lista1, 6 );
    ... ..
}

ListaDiElem InsInFondo( ListaDiElem lista, int elem ) {
    ListaDiElem punt, cur = lista;
    punt = (ListaDiElem) malloc( sizeof(ElemLista) );
    punt->prox = NULL;
    punt->info = elem;
    if ( lista==NULL )
        return punt;
    else {
        while( cur->prox != NULL )
            cur = cur->prox;
        cur->prox = punt;
    }
    return lista;
}

```

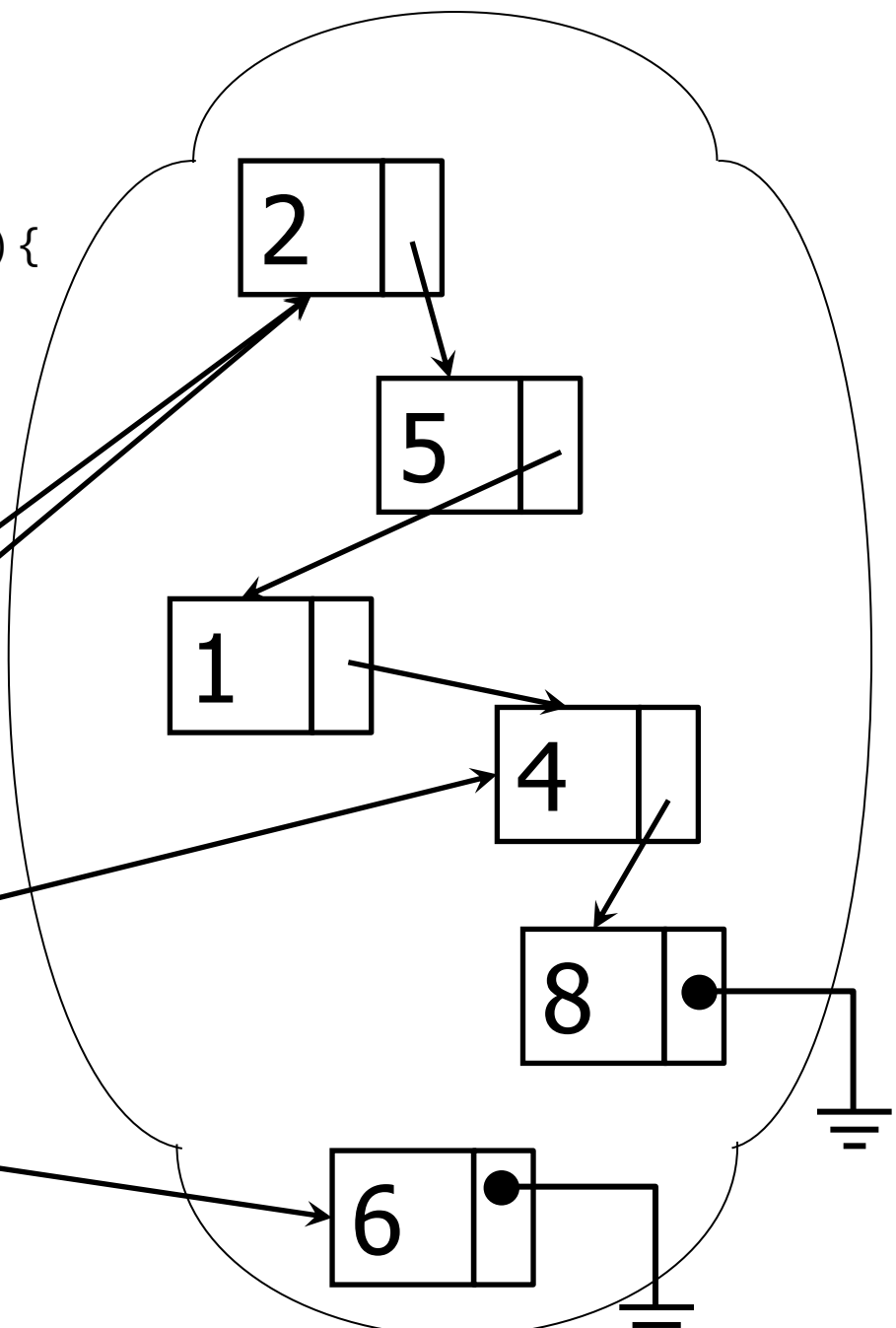
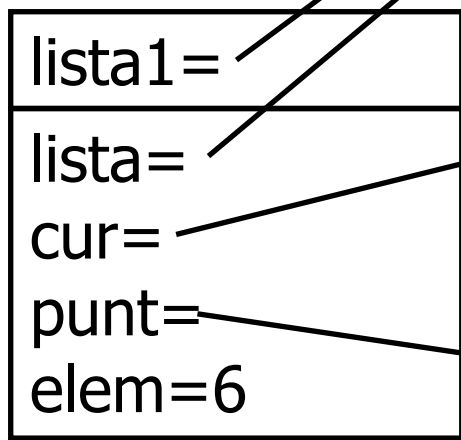


```

int main() {
    ListaDiElem lista1;
    ... ..
    lista1 = InsInFondo( lista1, 6 );
    ... ..
}

ListaDiElem InsInFondo( ListaDiElem lista, int elem ) {
    ListaDiElem punt, cur = lista;
    punt = (ListaDiElem) malloc( sizeof(ElemLista) );
    punt->prox = NULL;
    punt->info = elem;
    if ( lista==NULL )
        return punt;
    else {
        while( cur->prox != NULL )
            cur = cur->prox;
        cur->prox = punt;
    }
    return lista;
}

```

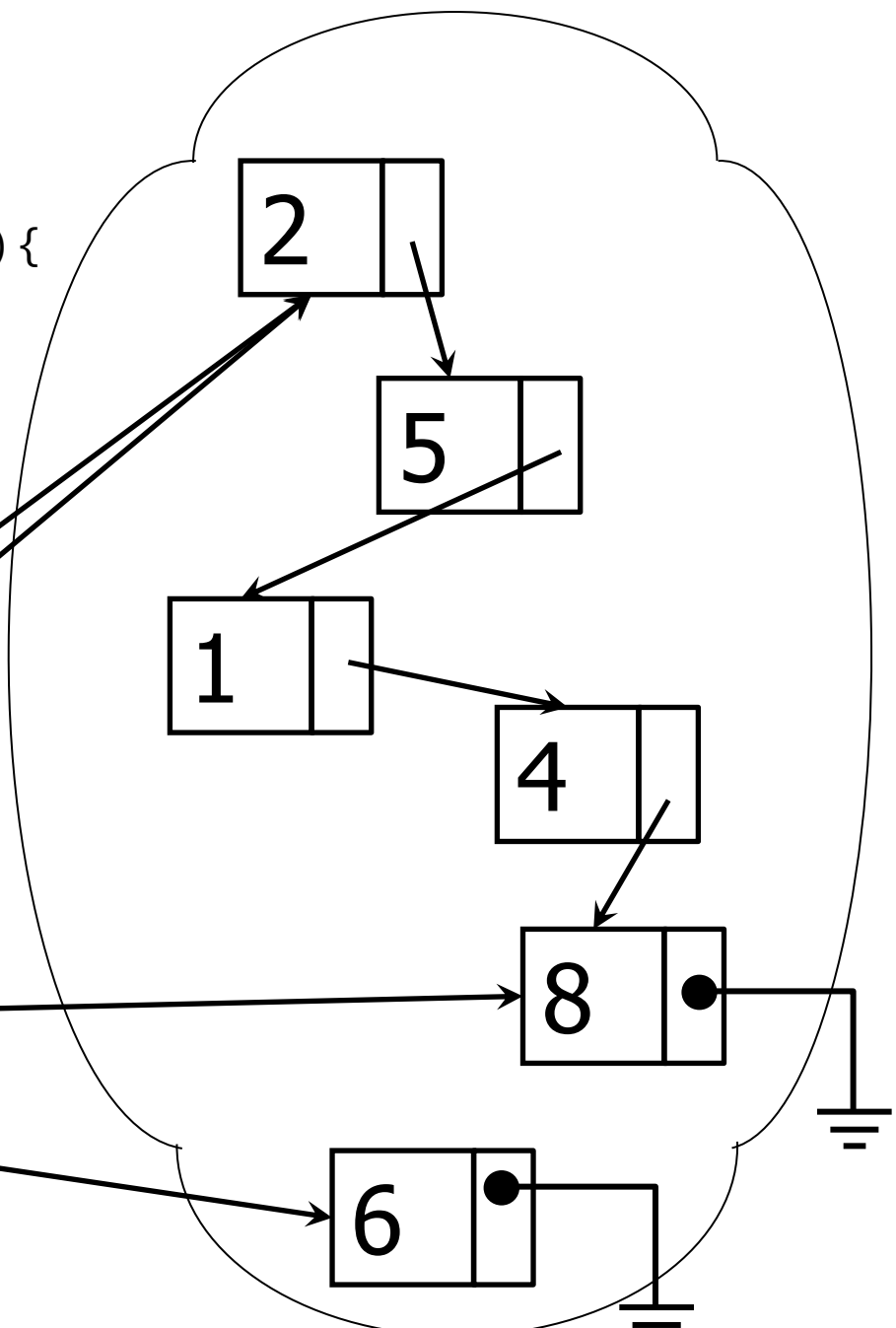
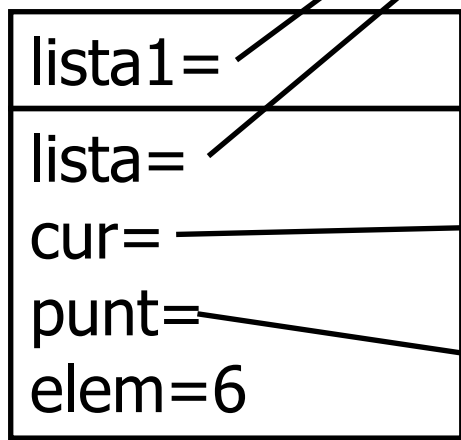


```

int main() {
    ListaDiElem lista1;
    ... ..
    lista1 = InsInFondo( lista1, 6 );
    ... ..
}

ListaDiElem InsInFondo( ListaDiElem lista, int elem ) {
    ListaDiElem punt, cur = lista;
    punt = (ListaDiElem) malloc( sizeof(ElemLista) );
    punt->prox = NULL;
    punt->info = elem;
    if ( lista==NULL )
        return punt;
    else {
        while( cur->prox != NULL )
            cur = cur->prox;
        cur->prox = punt;
    }
    return lista;
}

```

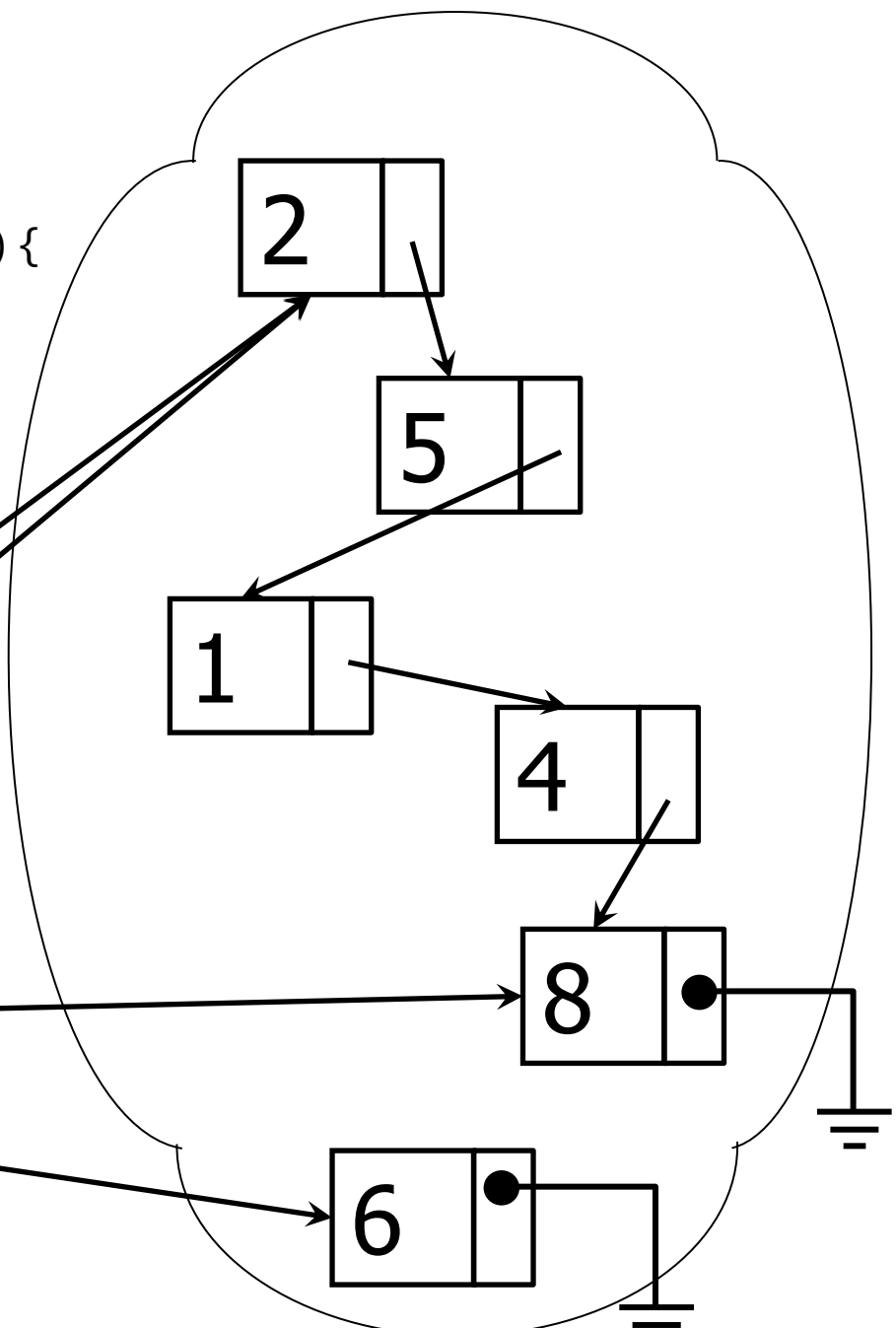
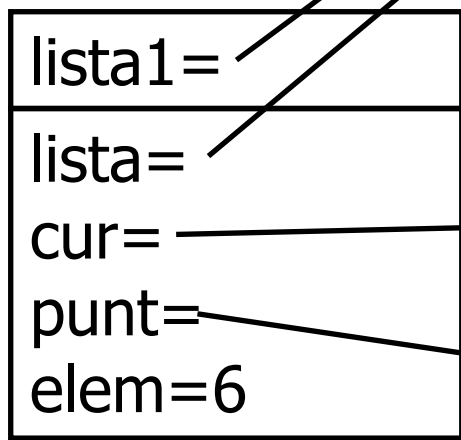


```

int main() {
    ListaDiElem lista1;
    ... ..
    lista1 = InsInFondo( lista1, 6 );
    ... ..
}

ListaDiElem InsInFondo( ListaDiElem lista, int elem ) {
    ListaDiElem punt, cur = lista;
    punt = (ListaDiElem) malloc( sizeof(ElemLista) );
    punt->prox = NULL;
    punt->info = elem;
    if ( lista==NULL )
        return punt;
    else {
        while( cur->prox != NULL )
            cur = cur->prox;
        cur->prox = punt;
    }
    return lista;
}

```

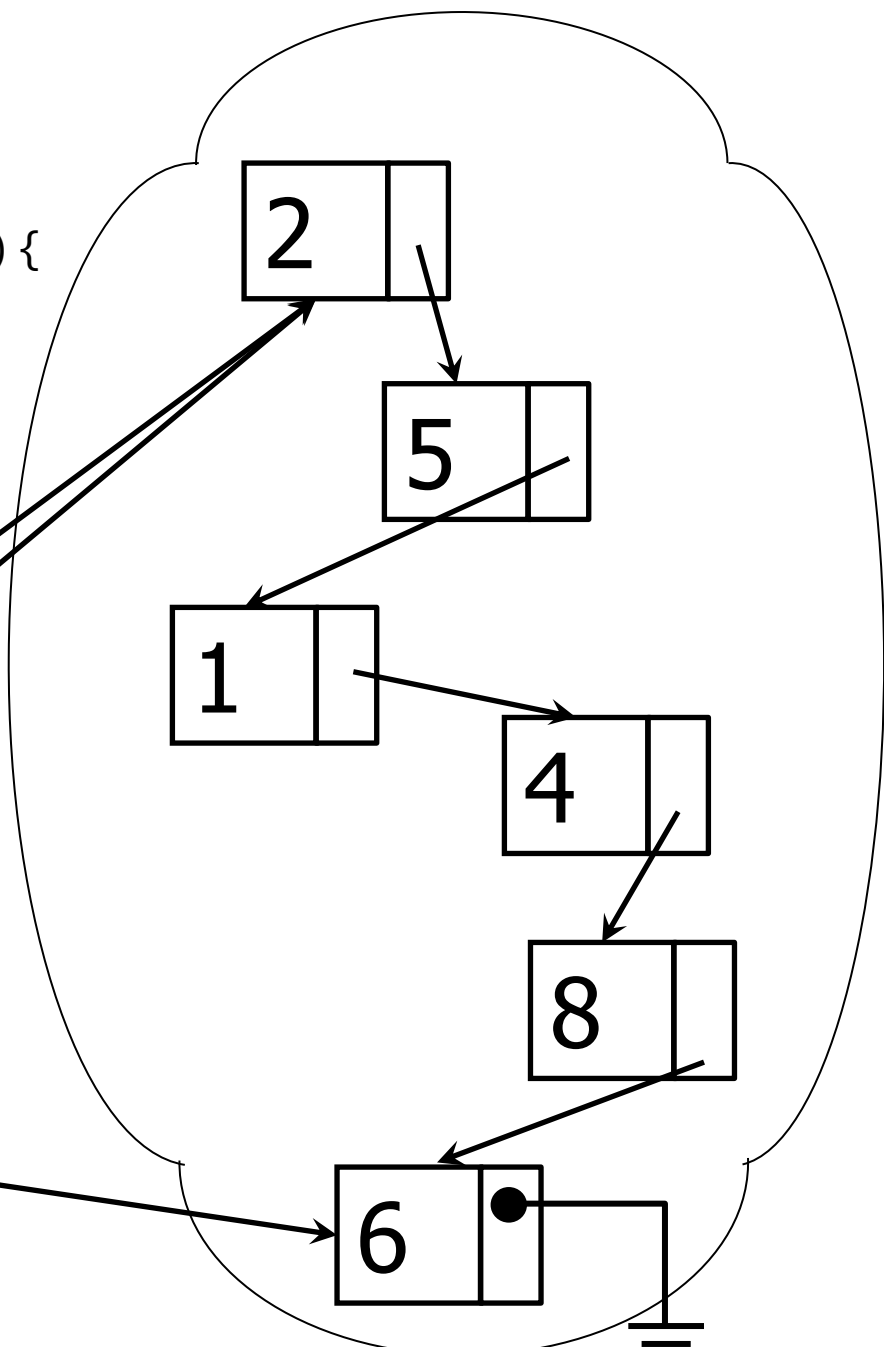
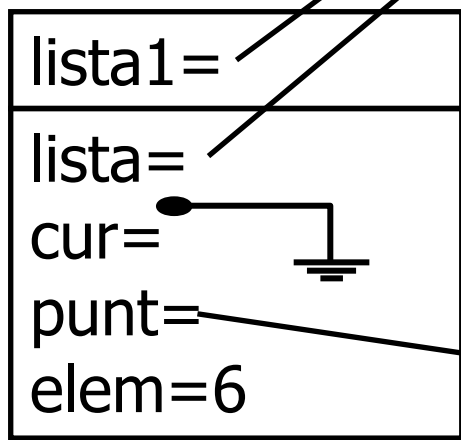


```

int main() {
    ListaDiElem lista1;
    ... ..
    lista1 = InsInFondo( lista1, 6 );
    ... ..
}

ListaDiElem InsInFondo( ListaDiElem lista, int elem ) {
    ListaDiElem punt, cur = lista;
    punt = (ListaDiElem) malloc( sizeof(ElemLista) );
    punt->prox = NULL;
    punt->info = elem;
    if ( lista==NULL )
        return punt;
    else {
        while( cur->prox != NULL )
            cur = cur->prox;
        cur->prox = punt;
    }
    return lista;
}

```

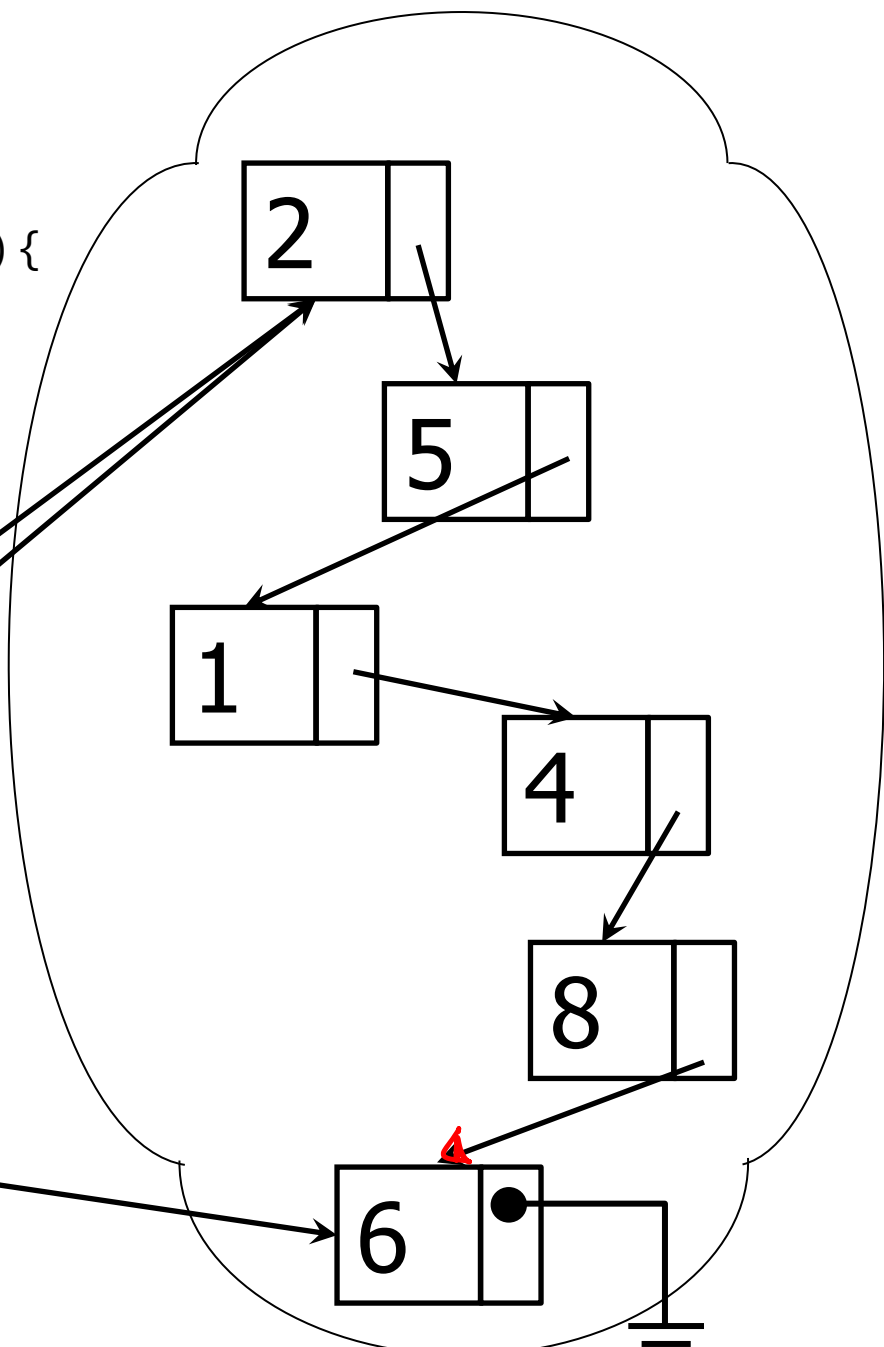
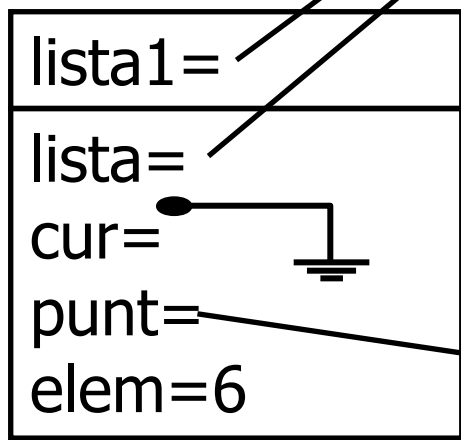


```

int main() {
    ListaDiElem lista1;
    ... ..
    lista1 = InsInFondo( lista1, 6 );
    ... ..
}

ListaDiElem InsInFondo( ListaDiElem lista, int elem ) {
    ListaDiElem punt, cur = lista;
    punt = (ListaDiElem) malloc( sizeof(ElemLista) );
    punt->prox = NULL;
    punt->info = elem;
    if ( lista==NULL )
        return punt;
    else {
        while( cur->prox != NULL )
            cur = cur->prox;
        cur->prox = punt;
    }
    return lista;
}

```

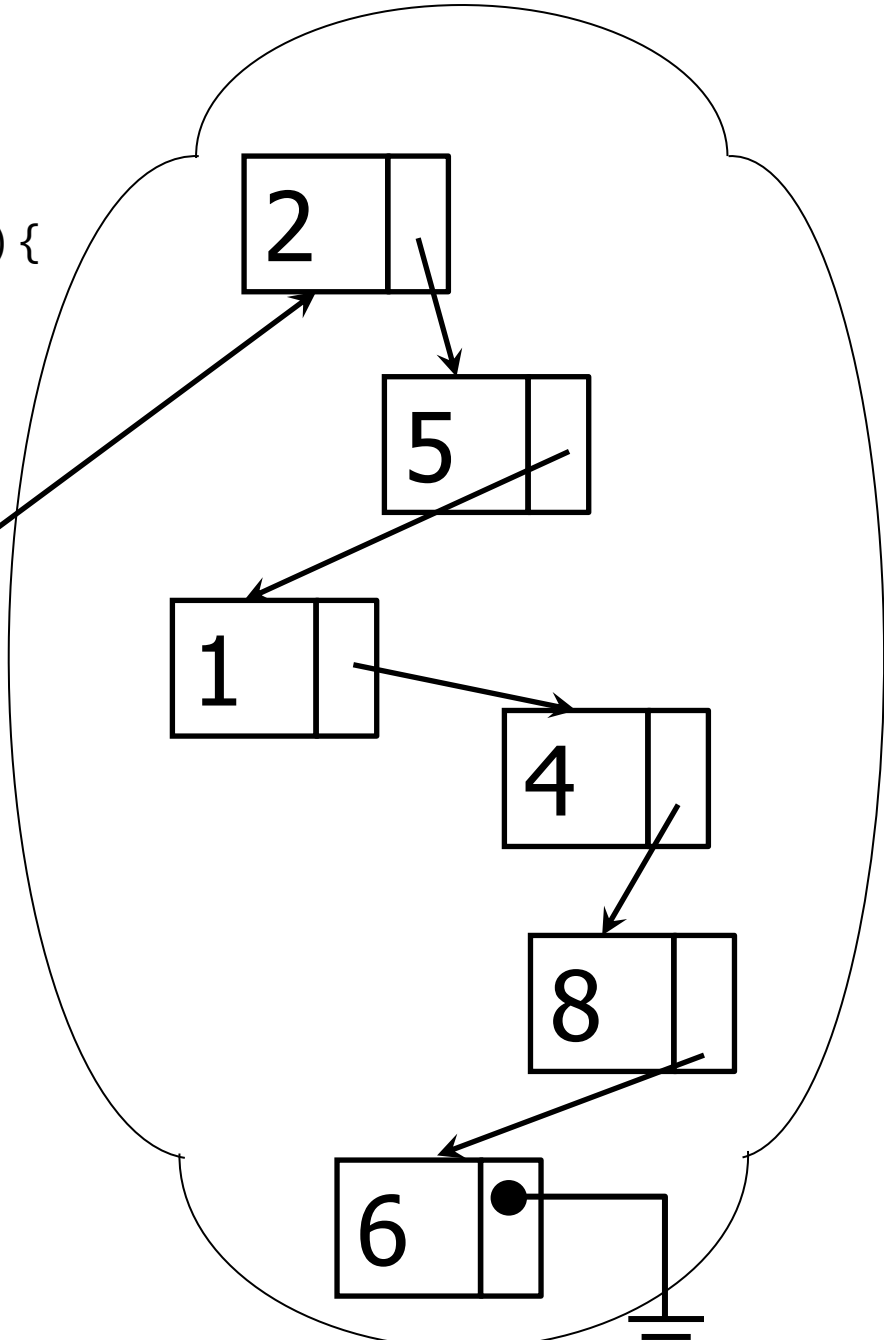
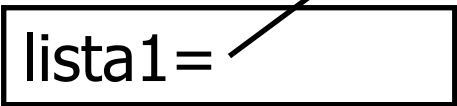



```

int main() {
    ListaDiElem lista1;
    ... ..
    lista1 = InsInFondo( lista1, 6 );
    ... ..
}

ListaDiElem InsInFondo( ListaDiElem lista, int elem ) {
    ListaDiElem punt, cur = lista;
    punt = (ListaDiElem) malloc( sizeof(ElemLista) );
    punt->prox = NULL;
    punt->info = elem;
    if ( lista==NULL )
        return punt;
    else {
        while( cur->prox != NULL )
            cur = cur->prox;
        cur->prox = punt;
    }
    return lista;
}

```



Lunghezza della lista

```
int numeronodi = 0;
Lista lis, ptr;
lis = ... /*costruzione della lista*/

for( ptr=lis; ptr!=NULL; ptr=ptr->next )
    numeronodi++;
```

avanzare = aggiornare il puntatore

- 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

Lunghezza Lista con Funzione

// questa funzione modifica p che è una copia del puntatore alla testa,

```
int calcolaLunghezza(PNodo p)
```

```
{
```

```
    int i = 0;
```

```
    while(p!=NULL)
```

```
    {
```

```
        p = p->next;
```

```
        i++;
```

```
    }
```

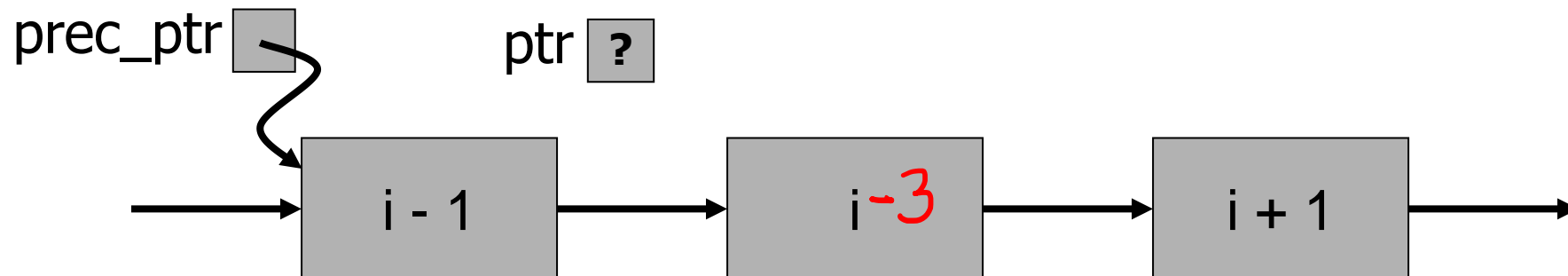
```
    return i;
```

```
}
```

Cancellare un nodo interno

...

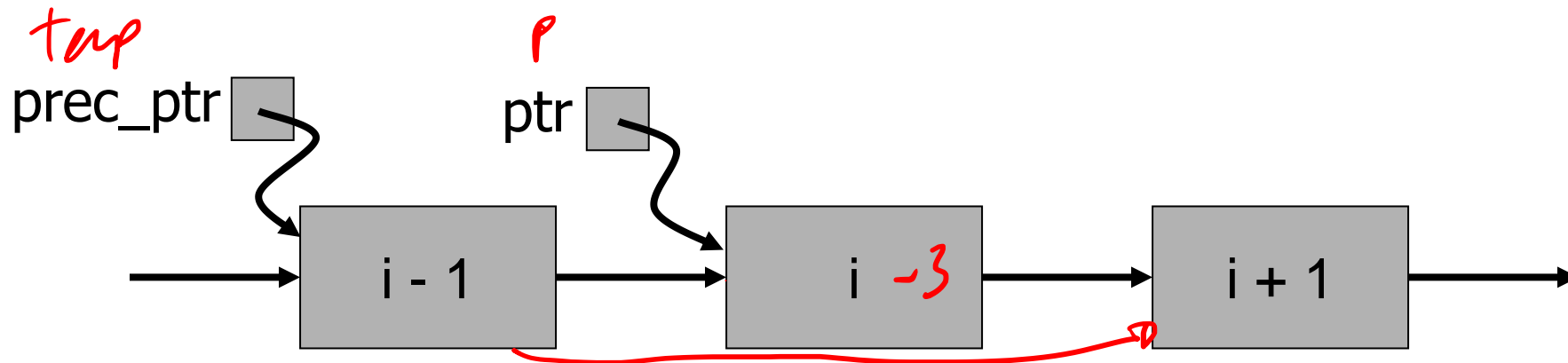
```
ptrNode ptr; /* puntatore al nodo  $i^{\circ}$  da cancellare */
ptrNode prec_ptr; /* puntatore al nodo  $(i-1)^{\circ}$  che
                  precede il nodo  $i^{\circ}$  da cancellare */
... /* qui si inizializzano ptr e prec_ptr ... */
prec_ptr->next = ptr->next;
/* collega il nodo  $(i-1)^{\circ}$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



Cancellare un nodo interno

...

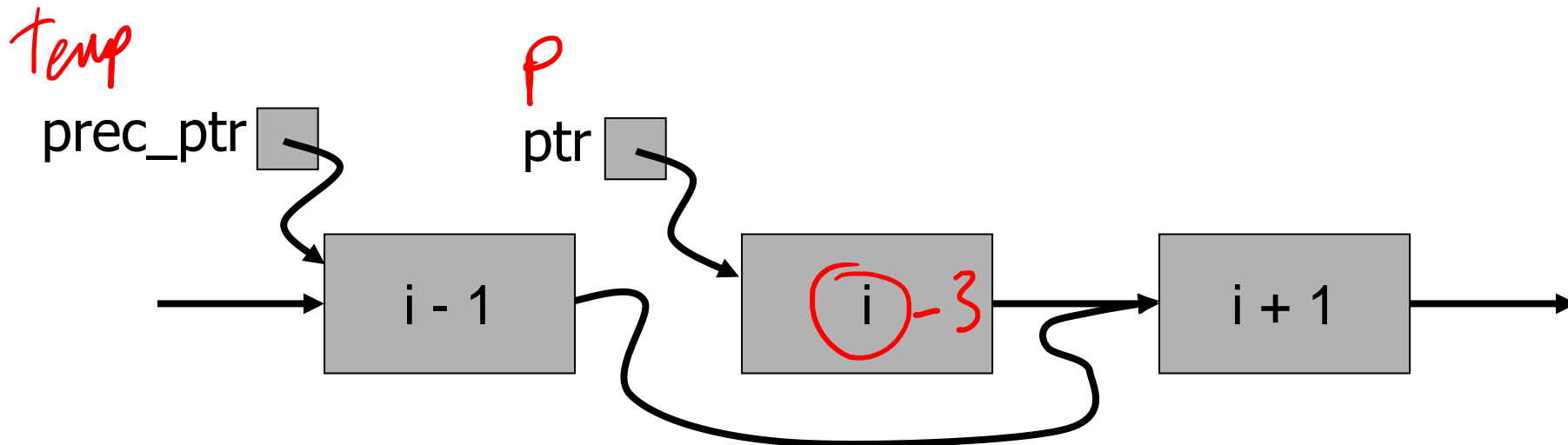
```
ptrNode ptr; /* puntatore al nodo  $i^{\circ}$  da cancellare */
ptrNode prec_ptr; /* puntatore al nodo  $(i-1)^{\circ}$  che
                  precede il nodo  $i^{\circ}$  da cancellare */
... /* qui si inizializzano ptr e prec_ptr ... */
prec_ptr->next = ptr->next;
/* collega il nodo  $(i-1)^{\circ}$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



Cancellare un nodo interno

...

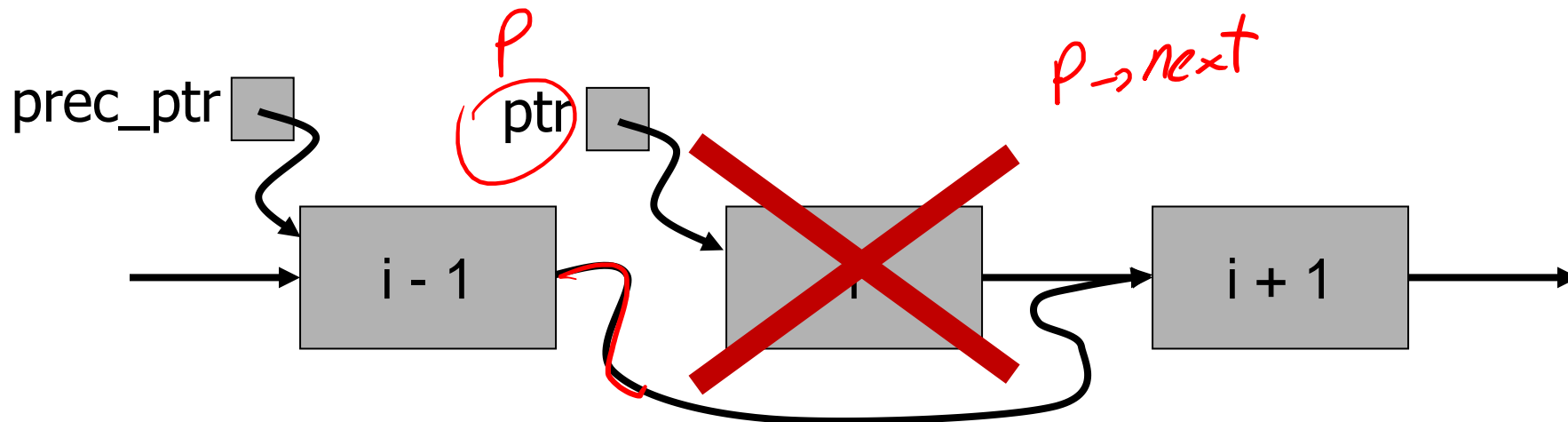
```
ptrNode ptr; /* puntatore al nodo  $i^{\circ}$  da cancellare */
ptrNode prec_ptr; /* puntatore al nodo  $(i-1)^{\circ}$  che
                  precede il nodo  $i^{\circ}$  da cancellare */
... /* qui si inizializzano ptr e prec_ptr ... */
prec_ptr->next = ptr->next;
/* collega il nodo  $(i-1)^{\circ}$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



Cancellare un nodo interno

...

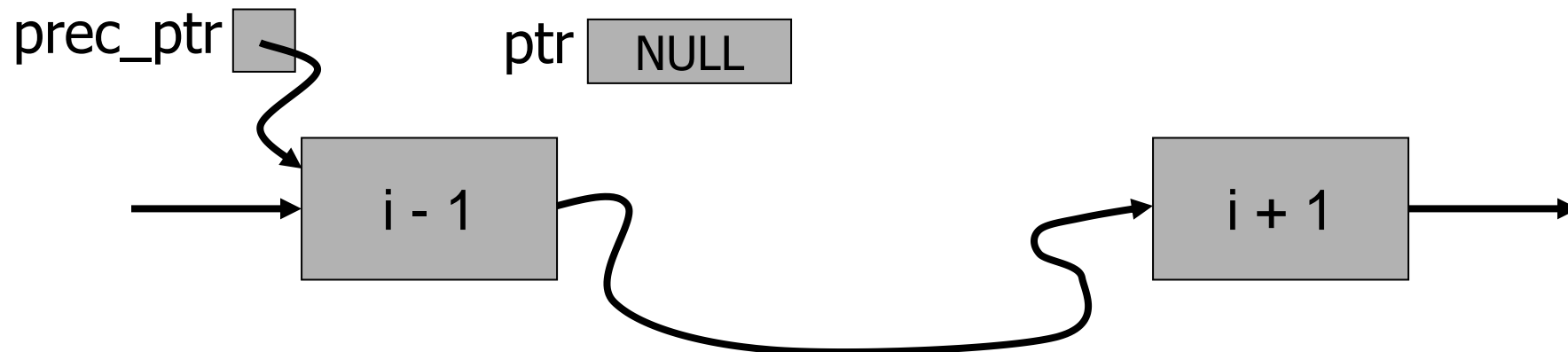
```
ptrNode ptr; /* puntatore al nodo i° da cancellare */
ptrNode prec_ptr; /* puntatore al nodo (i-1)° che
                  precede il nodo i° da cancellare */
... /* qui si inizializzano ptr e prec_ptr ... */
prec_ptr->next = ptr->next;
/* collega il nodo (i-1)° all' (i+1)°, saltando il nodo i° */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



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° */
ptr = NULL;
```



Cercare un nodo nella lista in base al valore

```
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 */
```

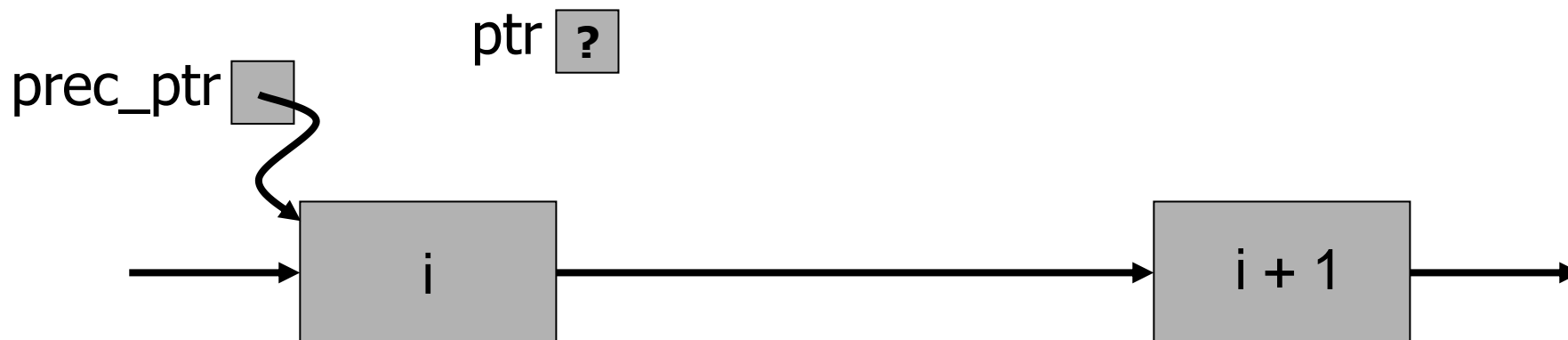
avanzare = aggiornare il puntatore

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

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;
```



Inserire un nodo interno alla lista

...

```
ptrNodp prec_ptr; /* puntatore al nodo  $i^{\text{esimo}}$ , che precede  
                  il nuovo nodo da inserire */
```

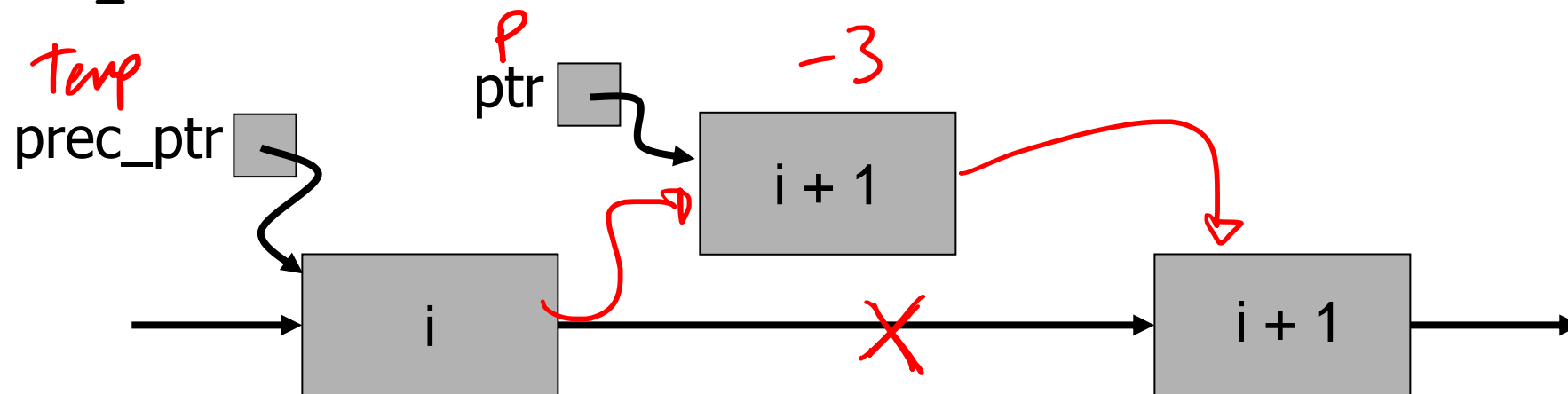
```
ptrNodo ptr; /* puntatore ausiliario a nodo */
```

```
... /* qui prec_ptr è inizializzato (trovare il nodo) */
```

```
ptr = malloc (sizeof (Nodo));
```

```
ptr->next = prec_ptr->next;
```

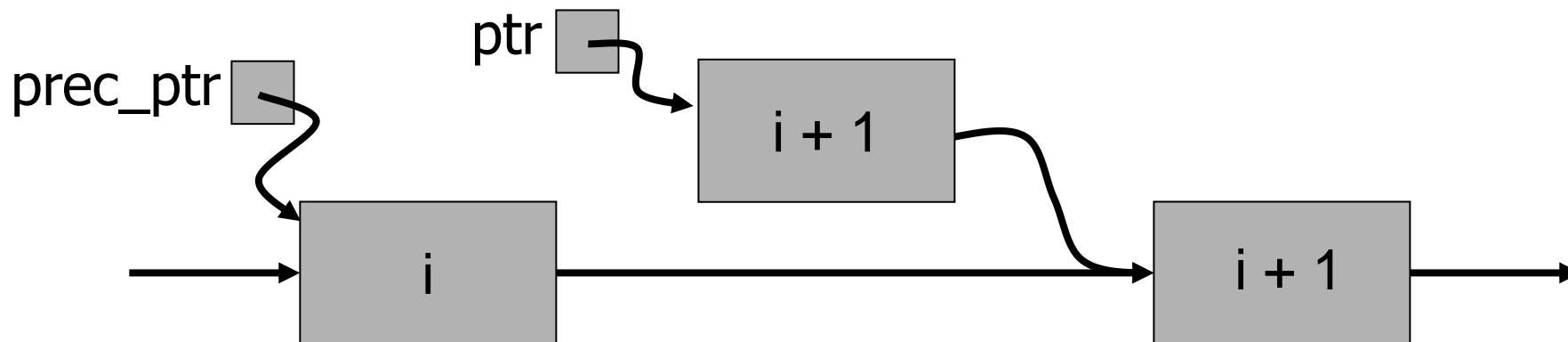
```
prec_ptr->next = ptr;
```



Inserire un nodo interno alla lista

...

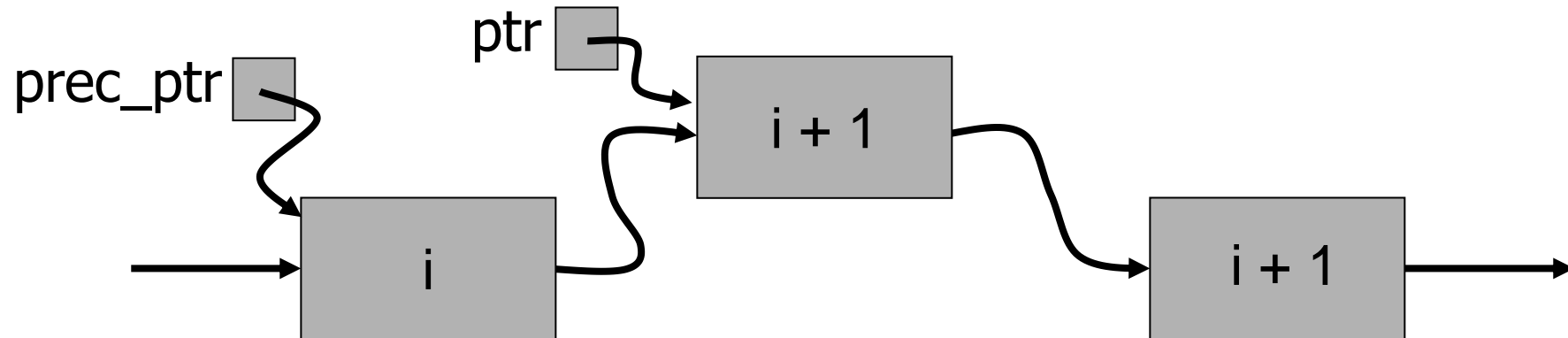
```
ptrNodp prec_ptr; /* puntatore al nodo  $i^{\text{esimo}}$ , che precede  
                  il nuovo nodo da inserire */  
  
ptrNodo ptr; /* puntatore ausiliario a nodo */  
... /* qui prec_ptr è inizializzato (trovare il nodo) */  
ptr = malloc (sizeof (Nodo));  
ptr->next = prec_ptr->next;  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



Inserire un nodo interno alla lista

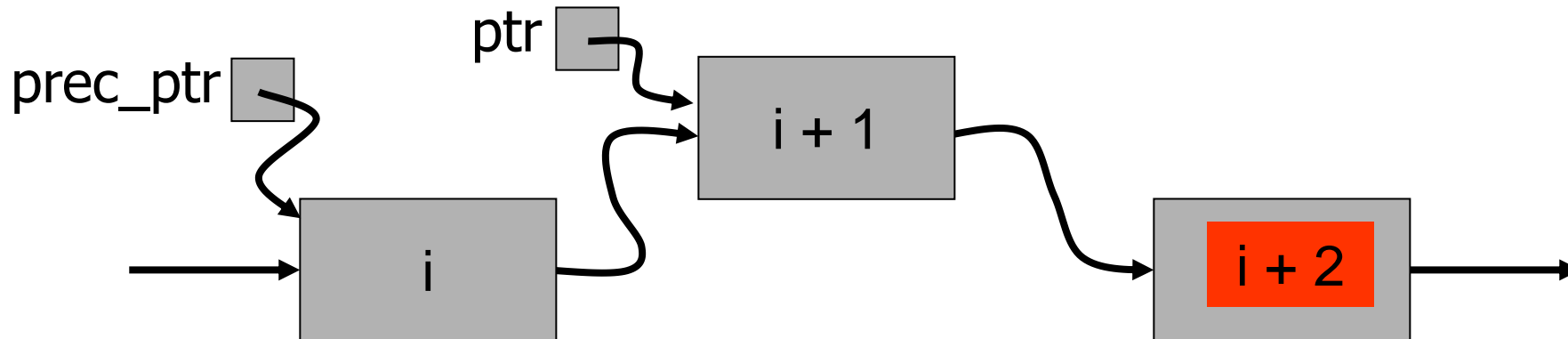
...

```
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;
```



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

Attenzione

...

```
ptrNode ptr;
```

```
ptr = (Node*)malloc(sizeof(Node)); /* alloca  
un 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

Liste come Tipi di Dato Astratti

cambiamo prospettiva...

Operazioni su liste (un "TDA"!)

TDA = Tipo di Dato Astratto

(su liste semplicemente concatenate)

- Inizializzazione
- Calcolo della dimensione
- Ricerca di un elemento
- Inserimento di un elemento
 - in prima posizione
 - in ultima posizione
 - ordinato
- Eliminazione di un elemento

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

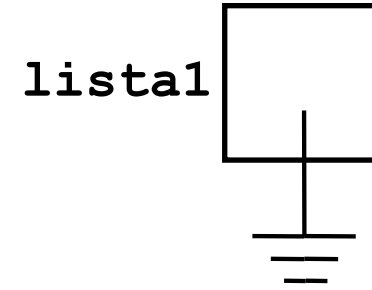
Usiamo questa formulazione

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

```
typedef ElemLista * ListaDiElem;
```



Inizializzazione



```
...  
ListaDiElem lista1;
```

```
...  
lista1 = NULL;
```

```
...  
ListaDiElem lista1;
```

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

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

Controllo lista vuota

```
if ( lista == NULL )  
    //è vuota  
else  
    //non è vuota
```

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

Dimensione della lista (iter. e ric.)

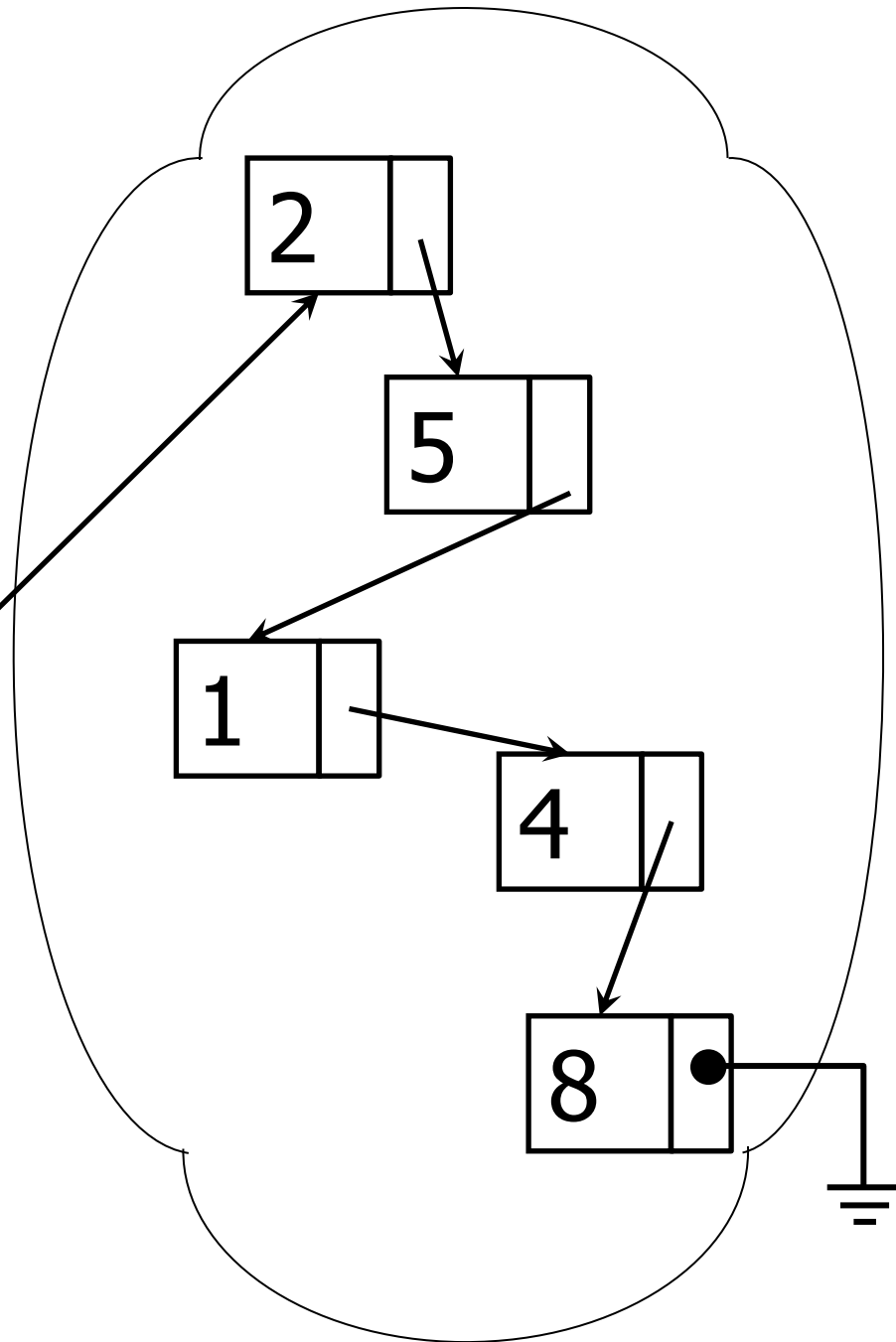
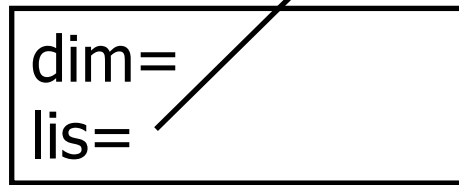
```
int Dimensione(ListaDiElem lista) {  
    int count = 0;  
    while( lista!=NULL) {  
        lista = lista->prox;    /* "distruggiamo" il parametro */  
        count++;  
    }  
    return count;  
}
```

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



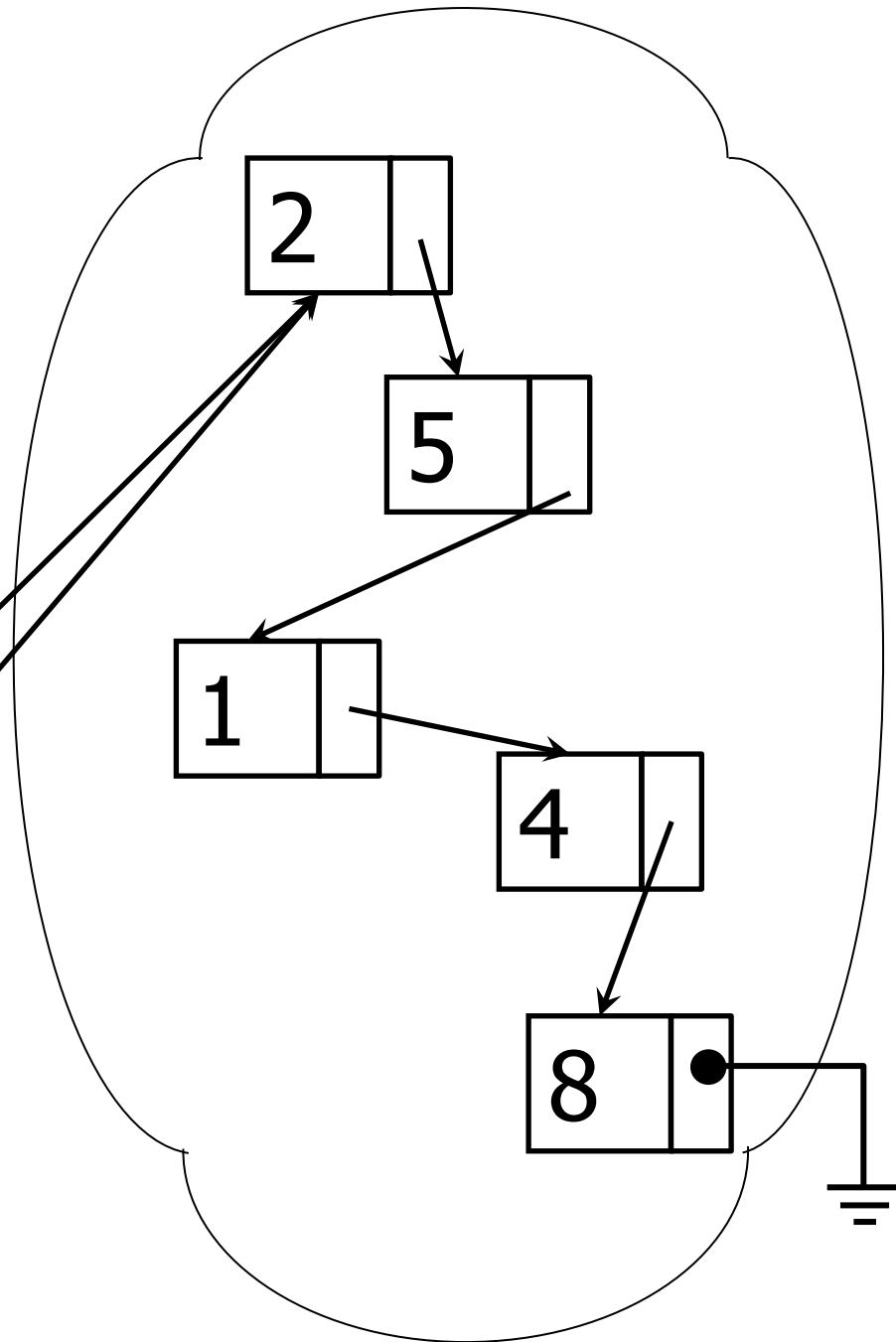
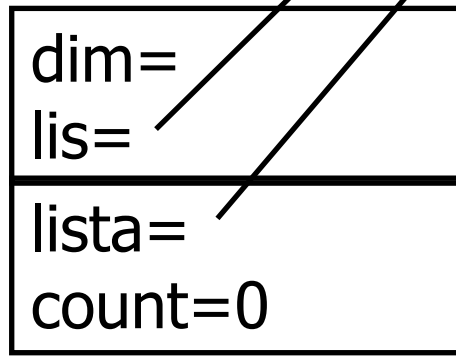
```
int main() {
    int dim;
    ListaDiElem lis;
    ... ..
    dim=Dimensione(lis);
    ... ..
}
```

```
int Dimensione(ListaDiElem lista)
{
    int count = 0;
    while( lista != NULL ) {
        lista = lista->prox;
        count++;
    }
    return count;
}
```



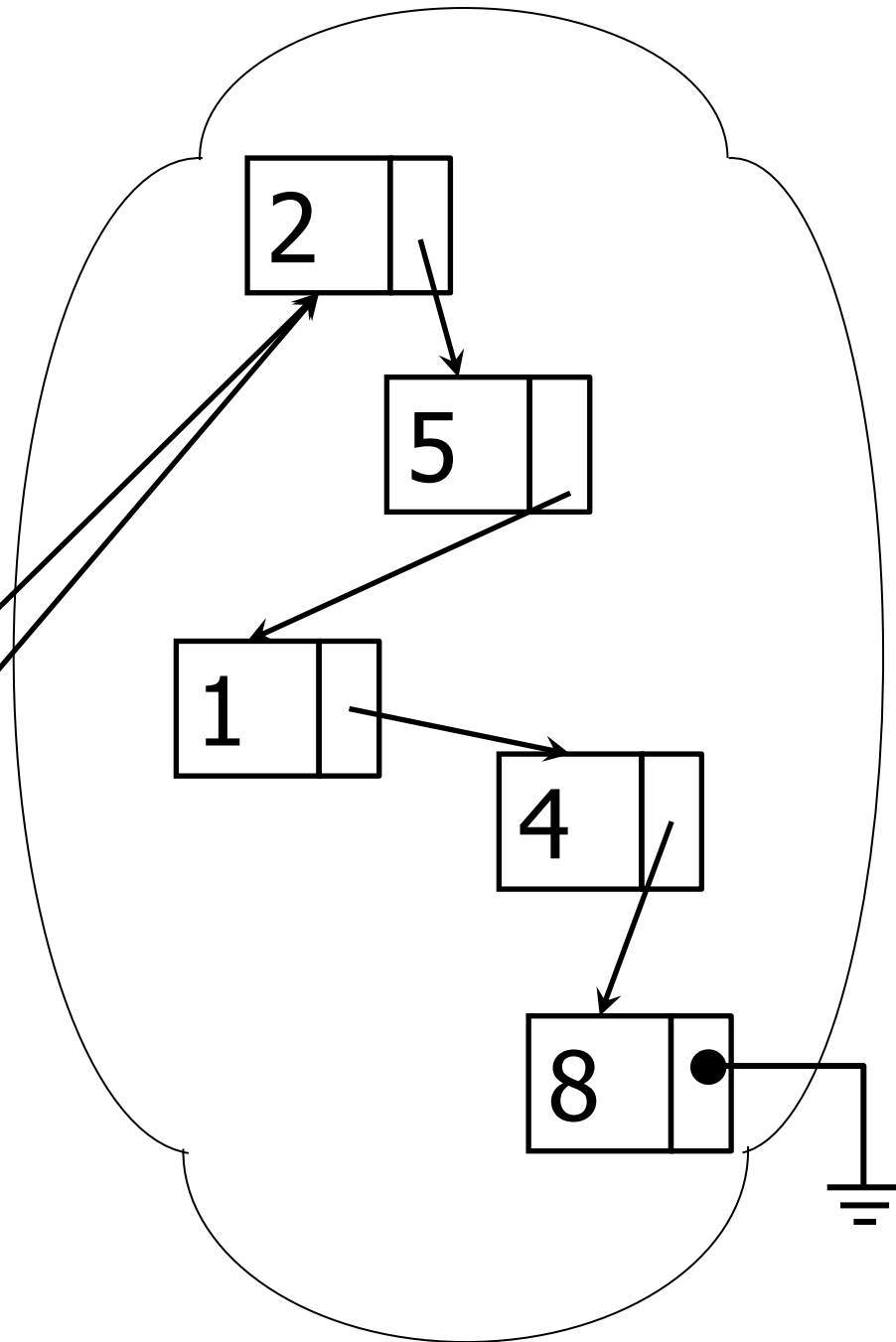
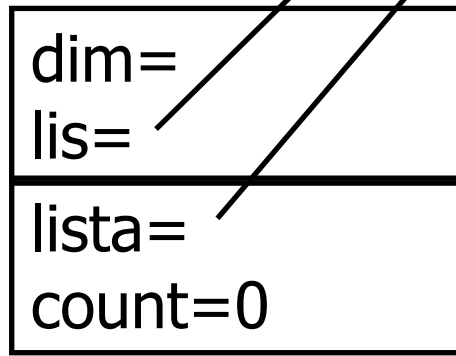
```
int main() {  
    int dim;  
    ListaDiElem lis;  
    ... ..  
    dim=Dimensione(lis);  
    ... ..  
}
```

```
int Dimensione(ListaDiElem lista)  
{  
    int count = 0;  
    while( lista != NULL ) {  
        lista = lista->prox;  
        count++;  
    }  
    return count;  
}
```



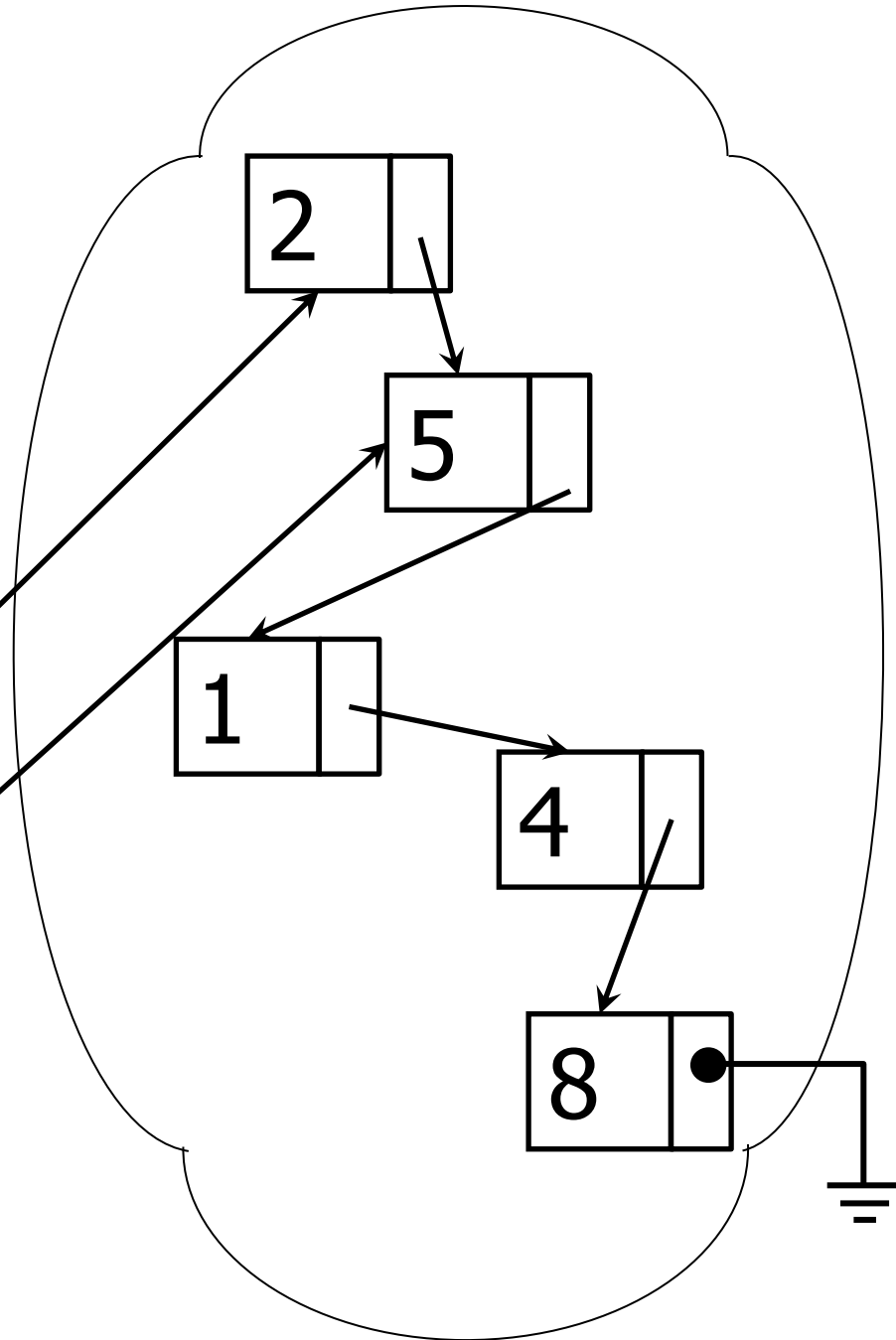
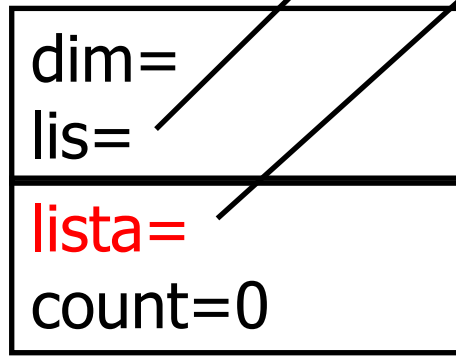
```
int main() {
    int dim;
    ListaDiElem lis;
    ... ..
    dim=Dimensione(lis);
    ... ..
}
```

```
int Dimensione(ListaDiElem lista)
{
    int count = 0;
    while( lista != NULL ) {
        lista = lista->prox;
        count++;
    }
    return count;
}
```



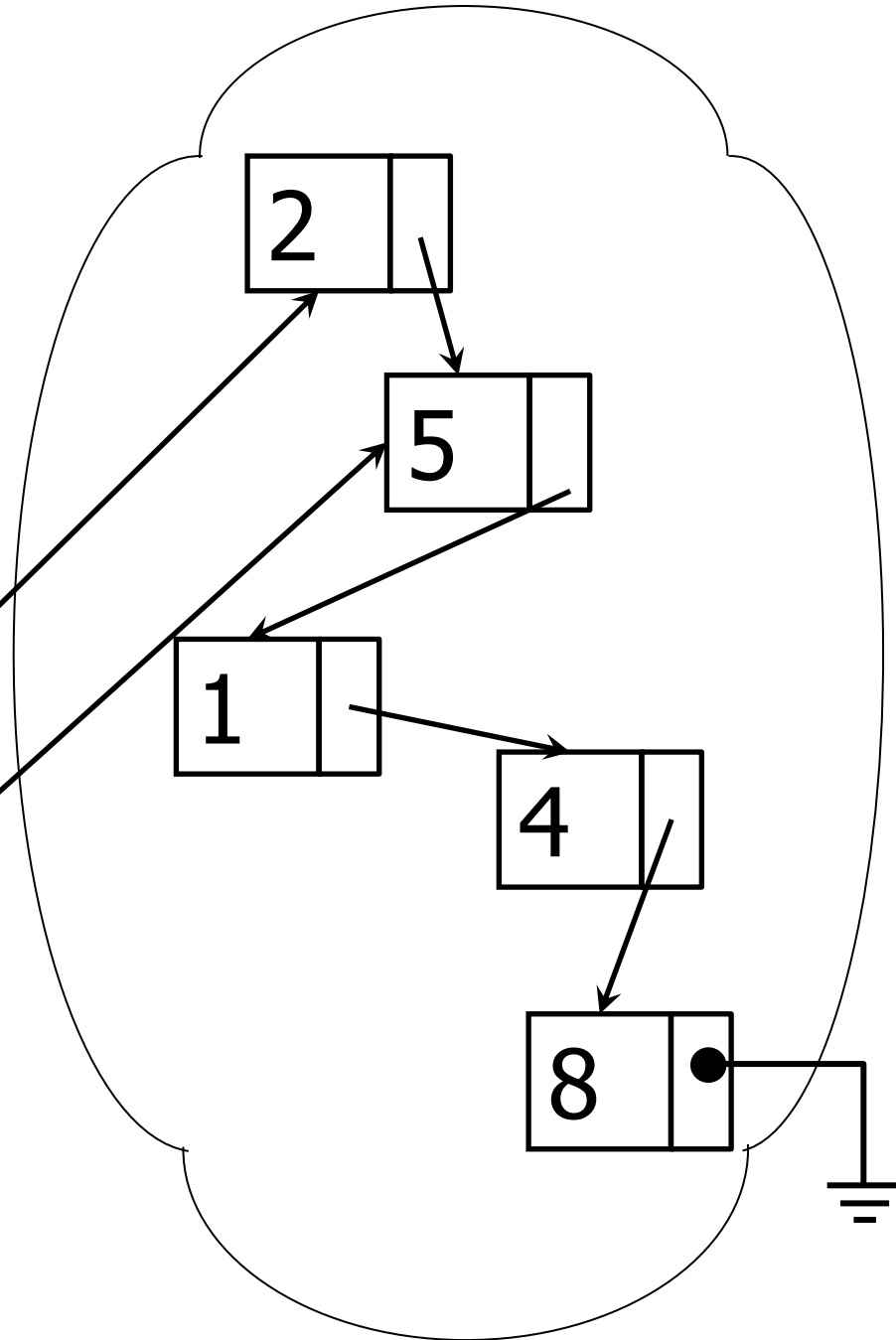
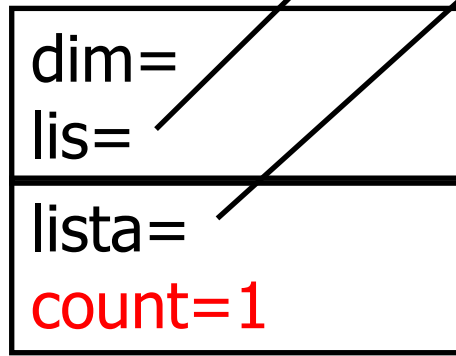
```
int main() {
    int dim;
    ListaDiElem lis;
    ... ..
    dim=Dimensione(lis);
    ... ..
}
```

```
int Dimensione(ListaDiElem lista)
{
    int count = 0;
    while( lista != NULL ) {
        lista = lista->prox;
        count++;
    }
    return count;
}
```



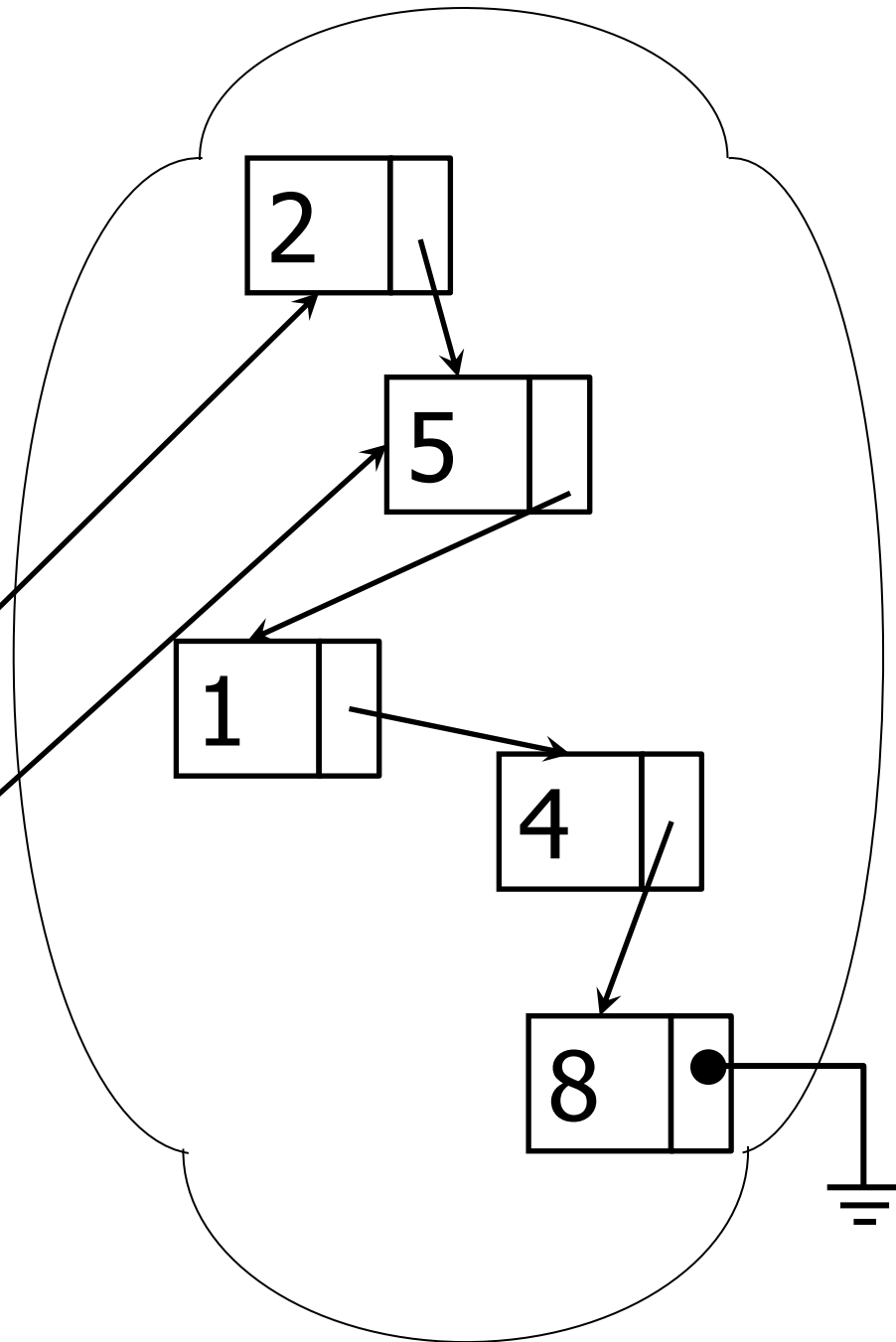
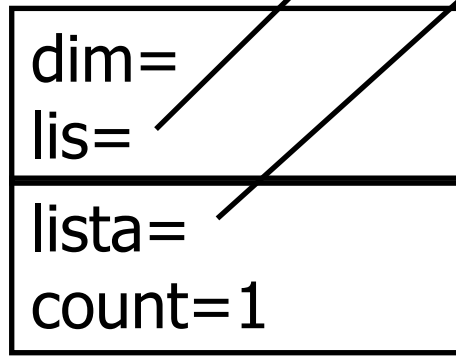
```
int main() {
    int dim;
    ListaDiElem lis;
    ... ..
    dim=Dimensione(lis);
    ... ..
}
```

```
int Dimensione(ListaDiElem lista)
{
    int count = 0;
    while( lista != NULL ) {
        lista = lista->prox;
        count++;
    }
    return count;
}
```



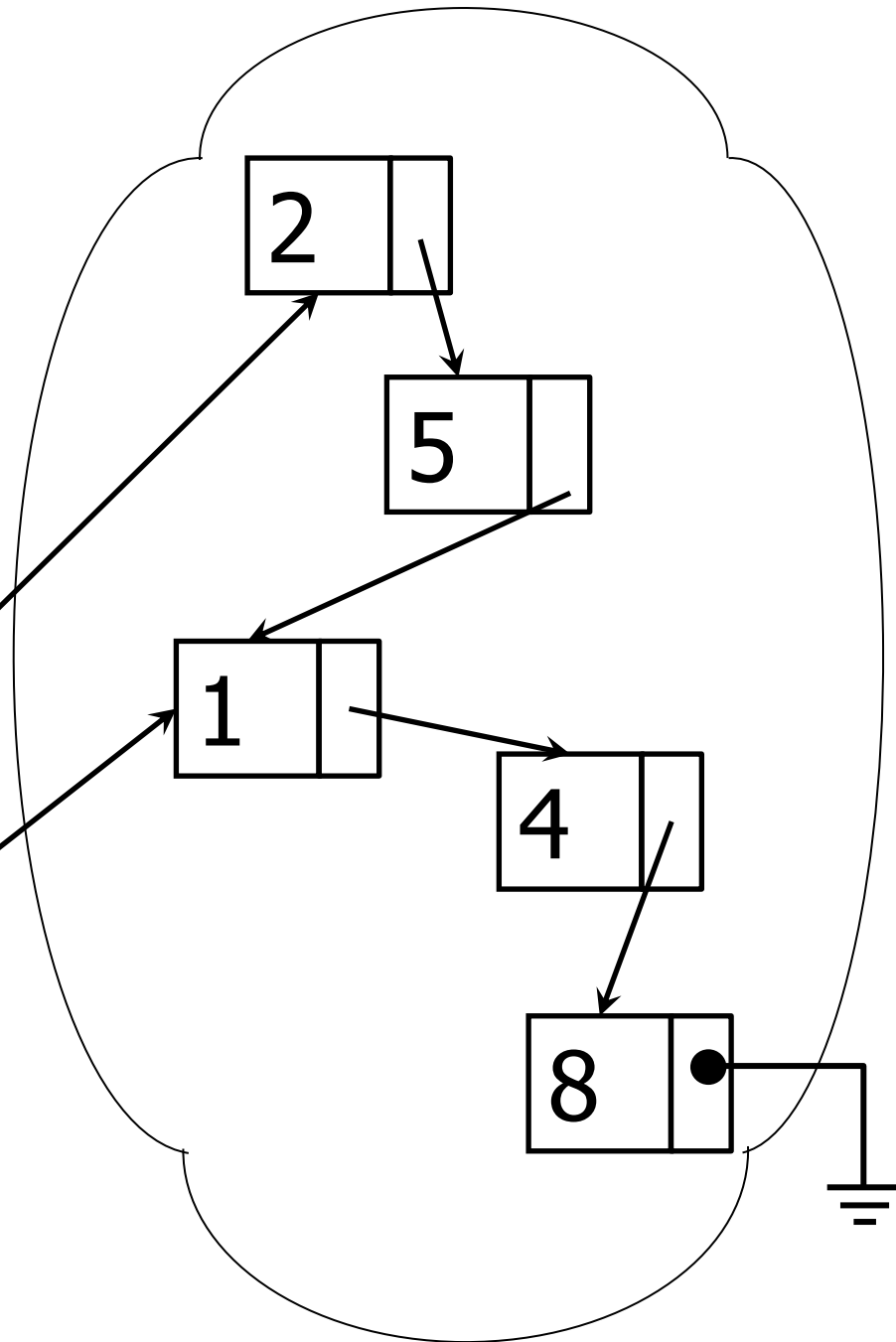
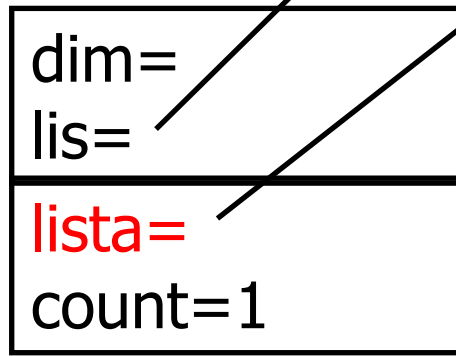
```
int main() {
    int dim;
    ListaDiElem lis;
    ... ..
    dim=Dimensione(lis);
    ... ..
}
```

```
int Dimensione(ListaDiElem lista)
{
    int count = 0;
    while( lista != NULL ) {
        lista = lista->prox;
        count++;
    }
    return count;
}
```



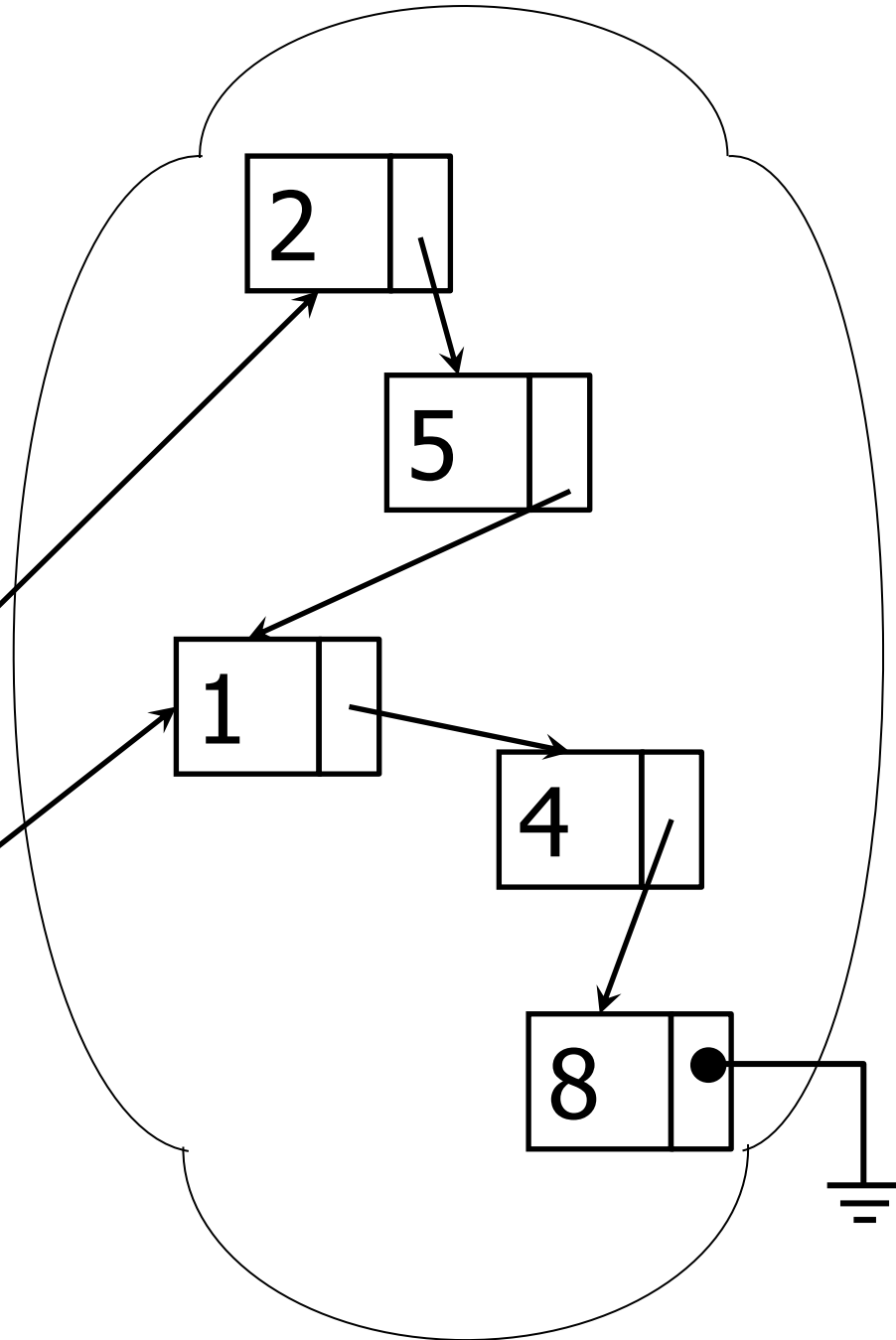
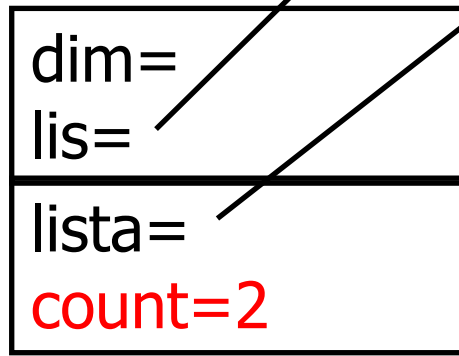
```
int main() {
    int dim;
    ListaDiElem lis;
    ... ..
    dim=Dimensione(lis);
    ... ..
}
```

```
int Dimensione(ListaDiElem lista)
{
    int count = 0;
    while( lista != NULL ) {
        lista = lista->prox;
        count++;
    }
    return count;
}
```



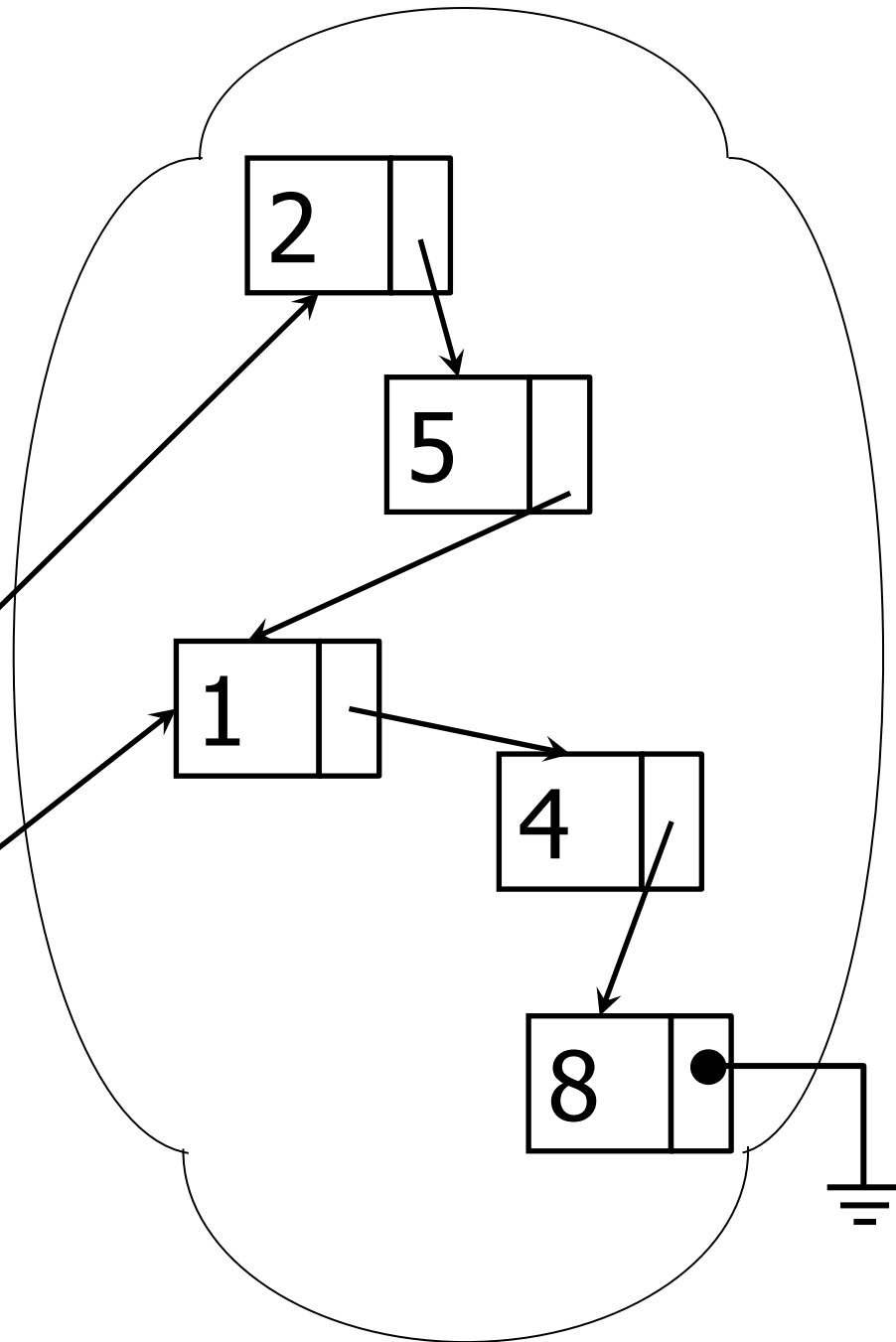
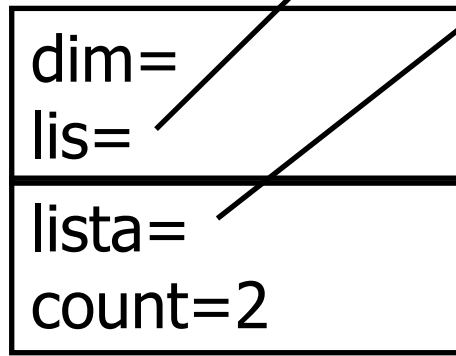
```
int main() {
    int dim;
    ListaDiElem lis;
    ... ..
    dim=Dimensione(lis);
    ... ..
}
```

```
int Dimensione(ListaDiElem lista)
{
    int count = 0;
    while( lista != NULL ) {
        lista = lista->prox;
        count++;
    }
    return count;
}
```



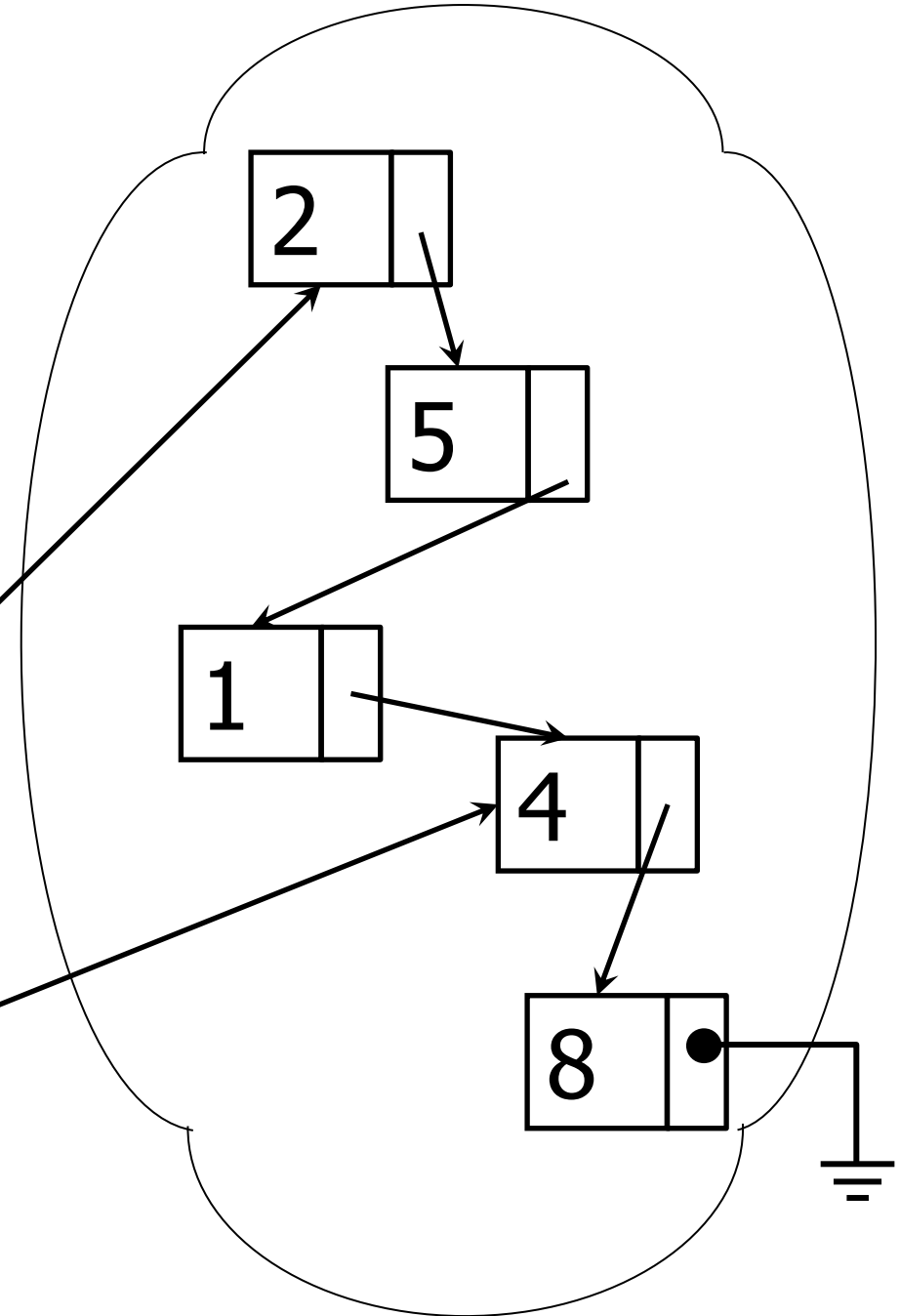
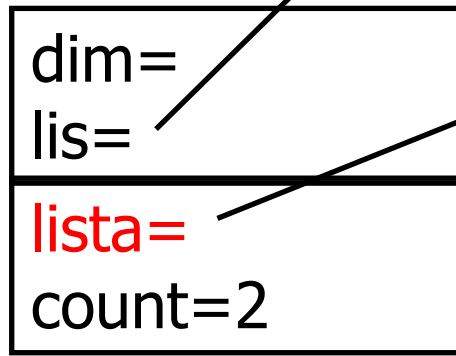

```
int main() {
    int dim;
    ListaDiElem lis;
    ... ..
    dim=Dimensione(lis);
    ... ..
}
```

```
int Dimensione(ListaDiElem lista)
{
    int count = 0;
    while( lista != NULL ) {
        lista = lista->prox;
        count++;
    }
    return count;
}
```



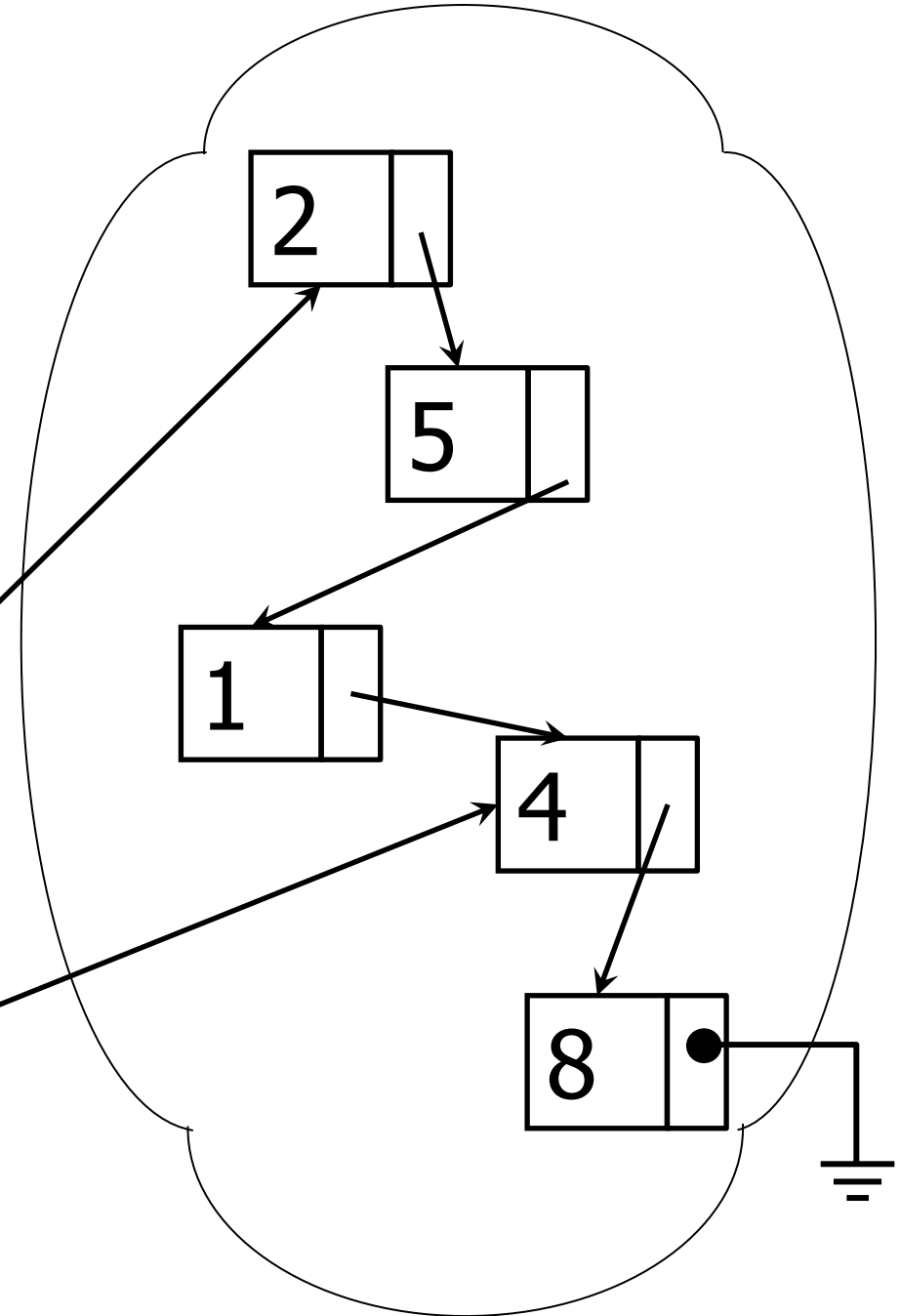
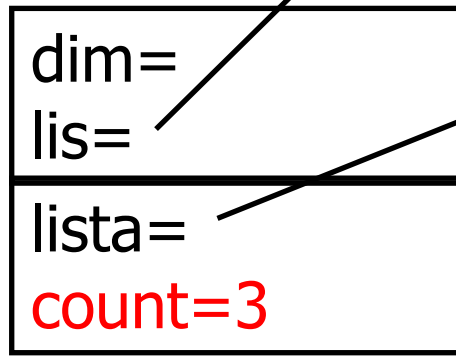
```
int main() {
    int dim;
    ListaDiElem lis;
    ... ..
    dim=Dimensione(lis);
    ... ..
}
```

```
int Dimensione(ListaDiElem lista)
{
    int count = 0;
    while( lista != NULL ) {
        lista = lista->prox;
        count++;
    }
    return count;
}
```



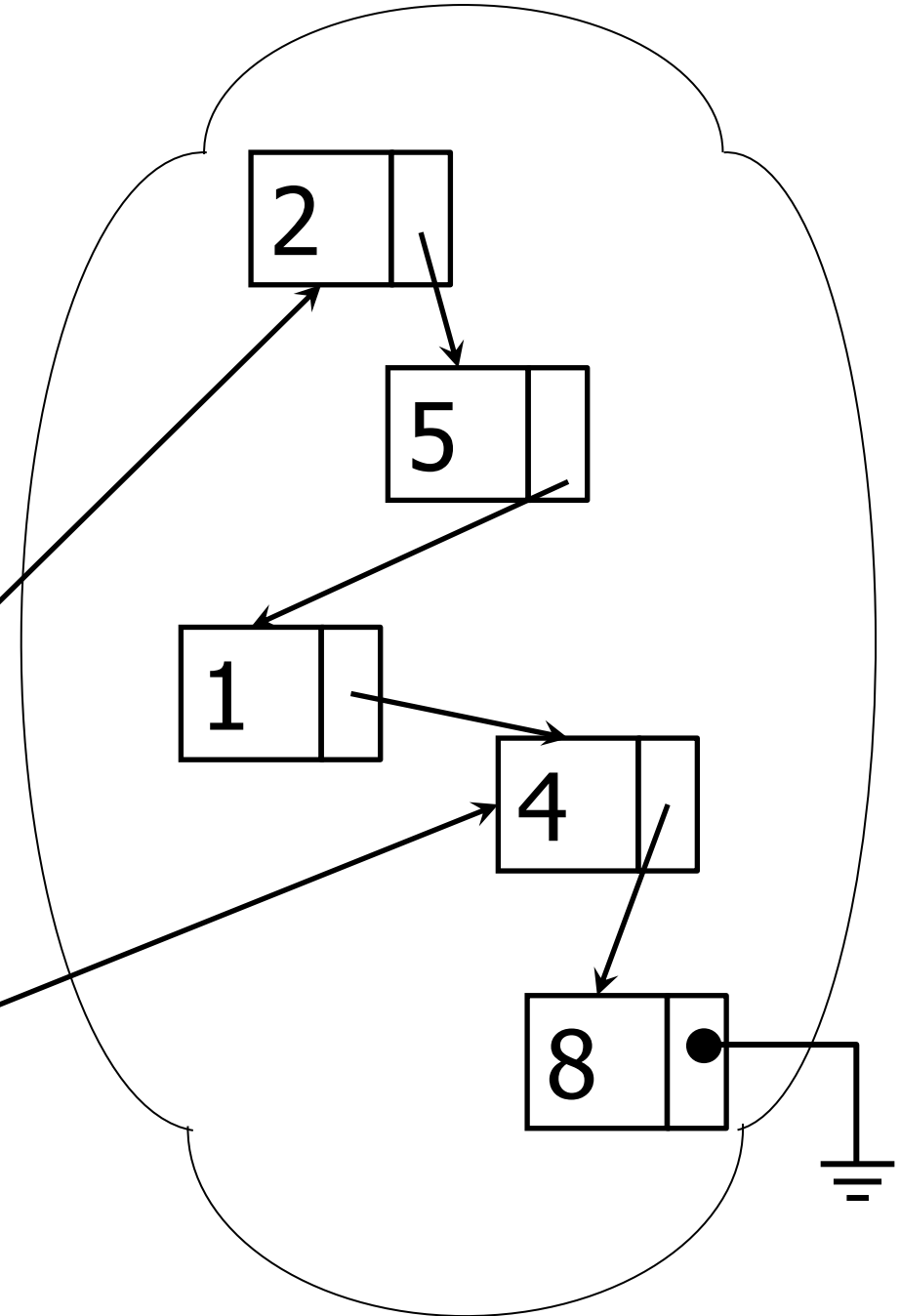
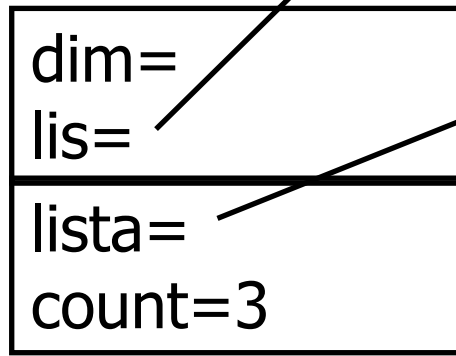
```
int main() {
    int dim;
    ListaDiElem lis;
    ... ..
    dim=Dimensione(lis);
    ... ..
}
```

```
int Dimensione(ListaDiElem lista)
{
    int count = 0;
    while( lista != NULL ) {
        lista = lista->prox;
        count++;
    }
    return count;
}
```



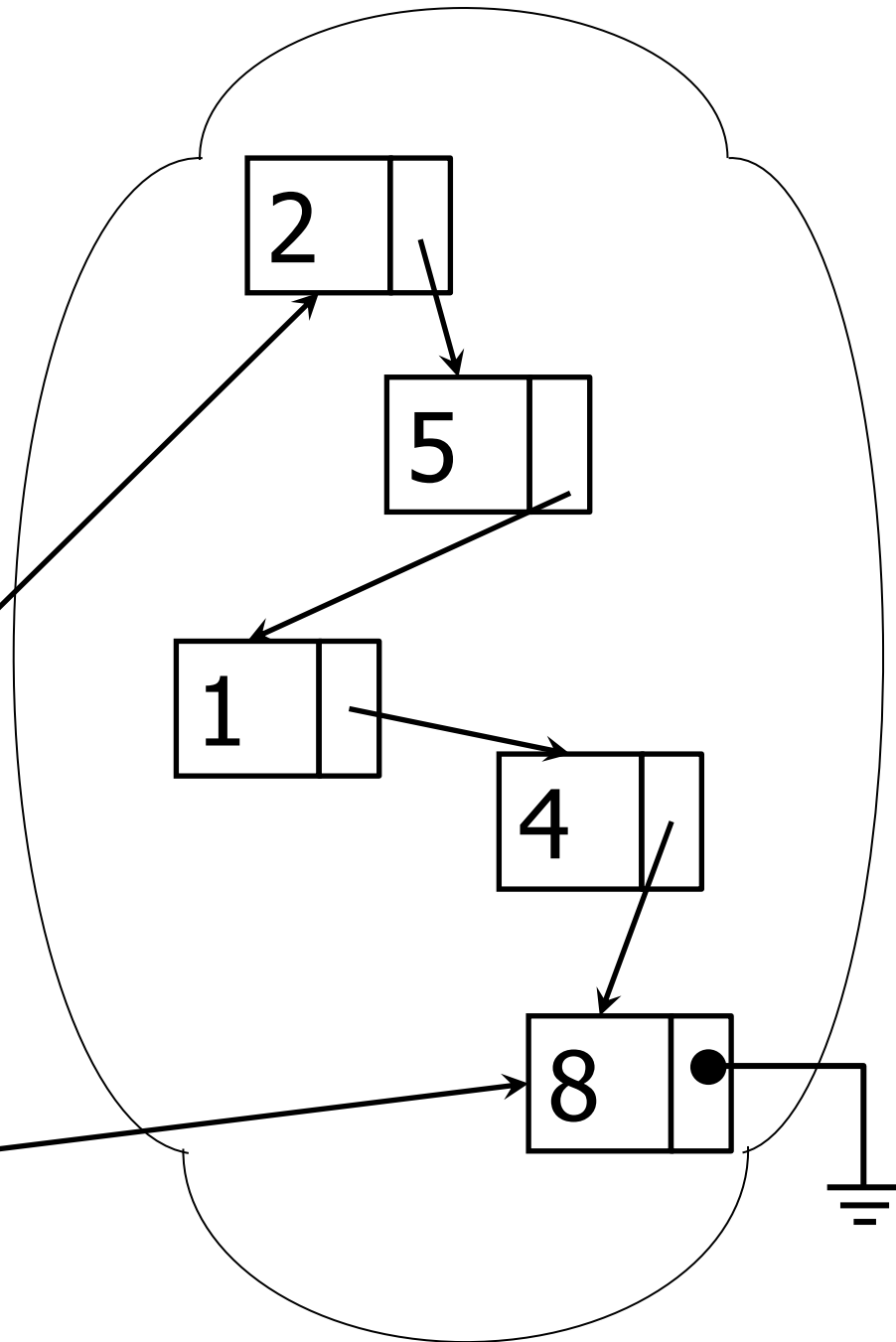
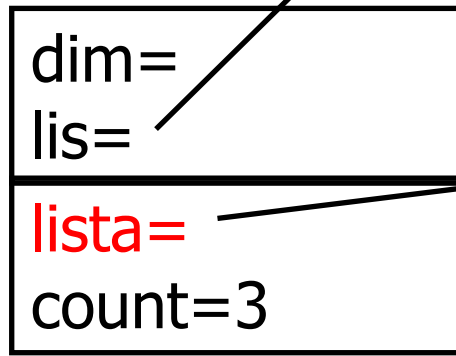
```
int main() {
    int dim;
    ListaDiElem lis;
    ... ..
    dim=Dimensione(lis);
    ... ..
}
```

```
int Dimensione(ListaDiElem lista)
{
    int count = 0;
    while( lista != NULL ) {
        lista = lista->prox;
        count++;
    }
    return count;
}
```



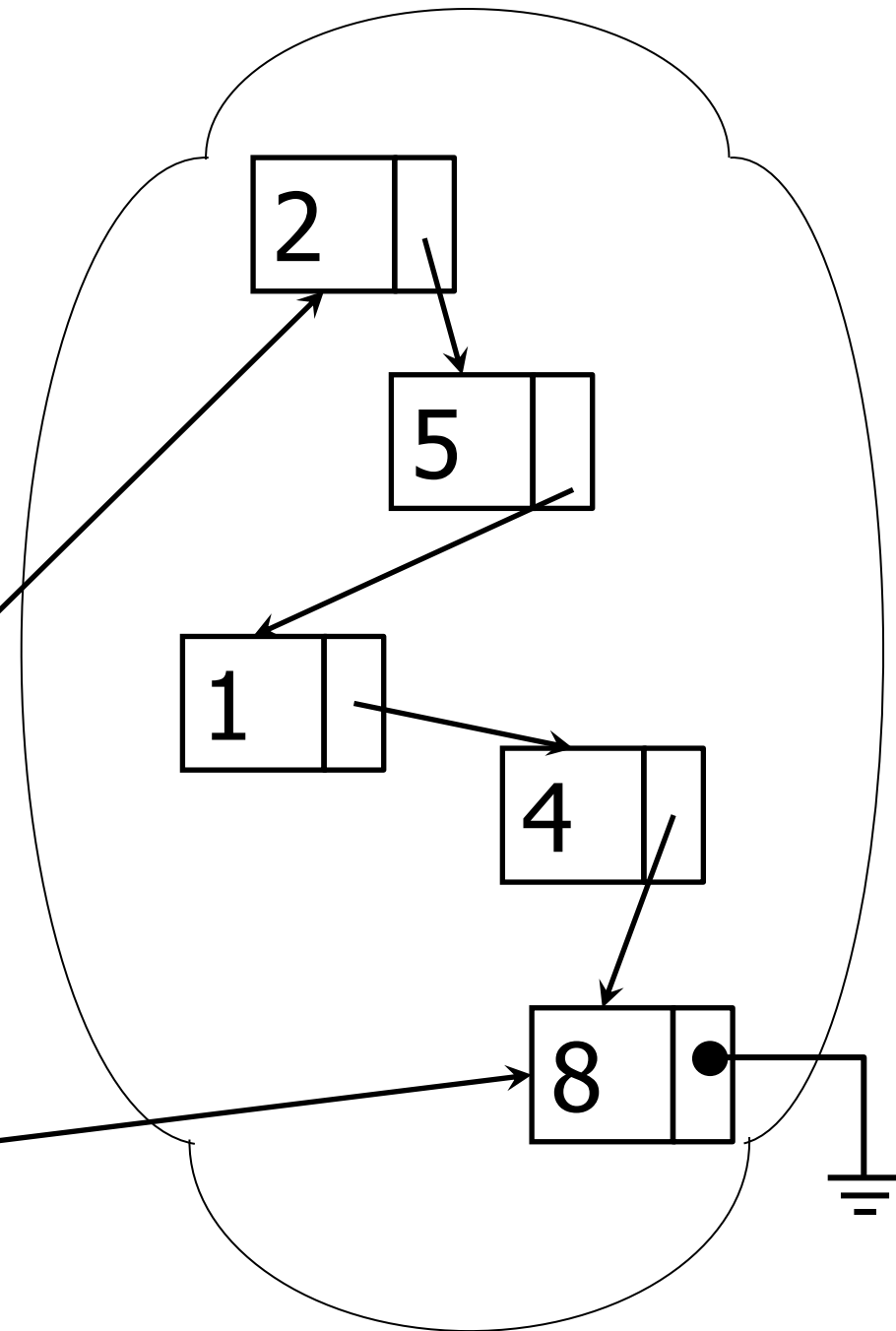
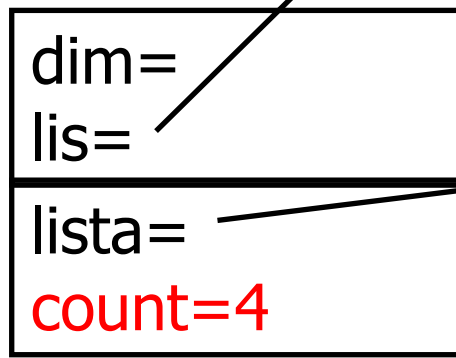
```
int main() {
    int dim;
    ListaDiElem lis;
    ... ..
    dim=Dimensione(lis);
    ... ..
}
```

```
int Dimensione(ListaDiElem lista)
{
    int count = 0;
    while( lista != NULL ) {
        lista = lista->prox;
        count++;
    }
    return count;
}
```



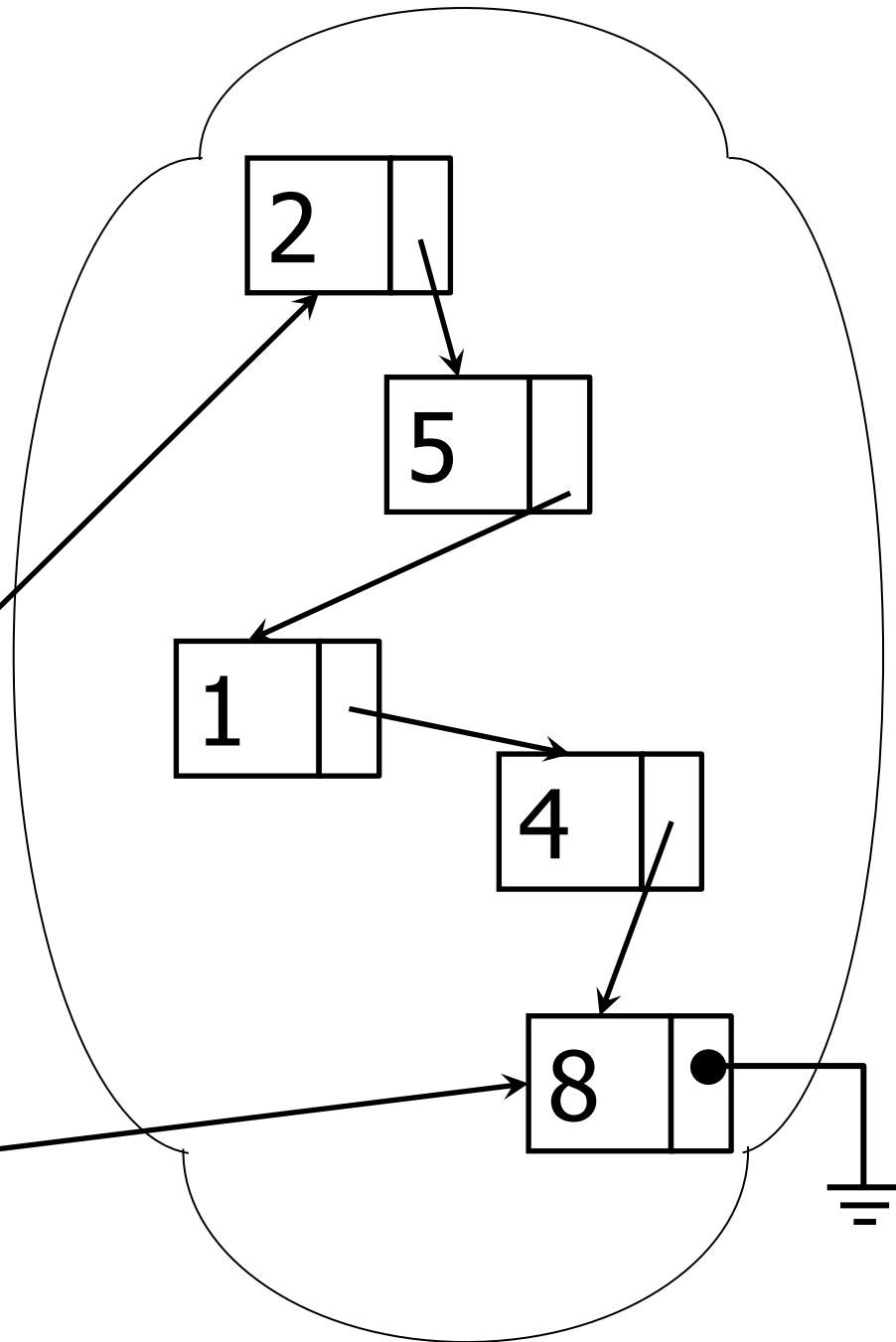
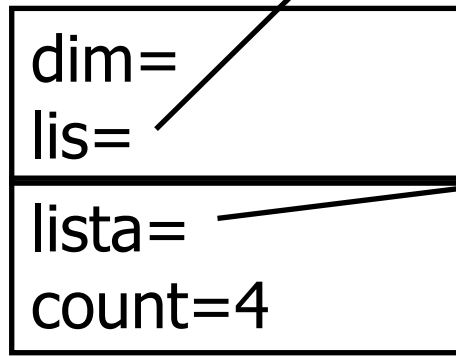
```
int main() {
    int dim;
    ListaDiElem lis;
    ... ..
    dim=Dimensione(lis);
    ... ..
}
```

```
int Dimensione(ListaDiElem lista)
{
    int count = 0;
    while( lista != NULL ) {
        lista = lista->prox;
        count++;
    }
    return count;
}
```



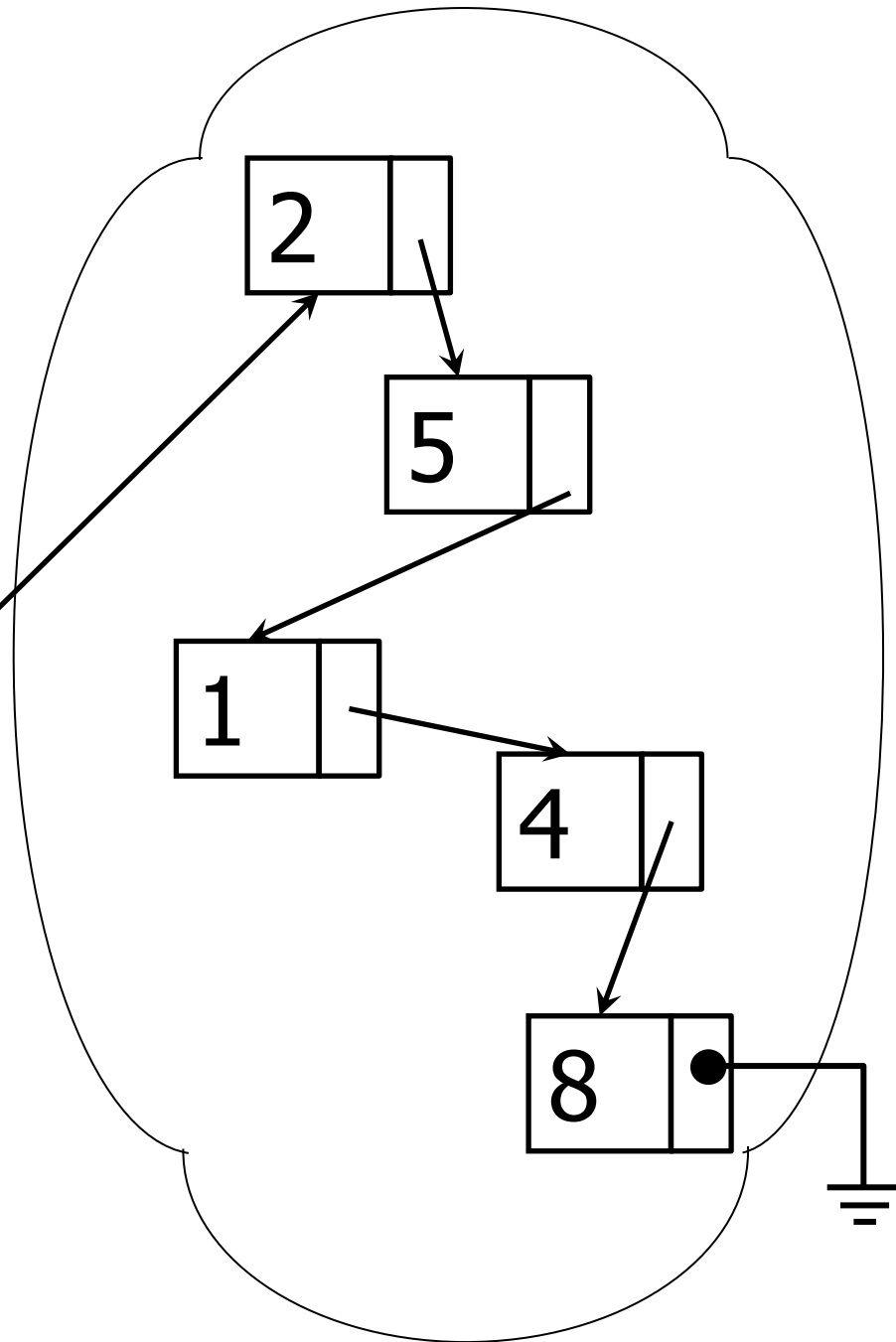
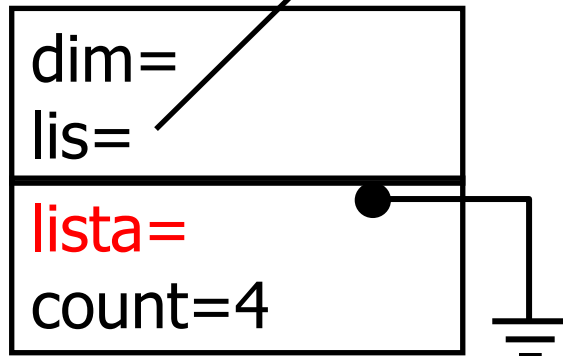
```
int main() {
    int dim;
    ListaDiElem lis;
    ... ..
    dim=Dimensione(lis);
    ... ..
}
```

```
int Dimensione(ListaDiElem lista)
{
    int count = 0;
    while( lista != NULL ) {
        lista = lista->prox;
        count++;
    }
    return count;
}
```



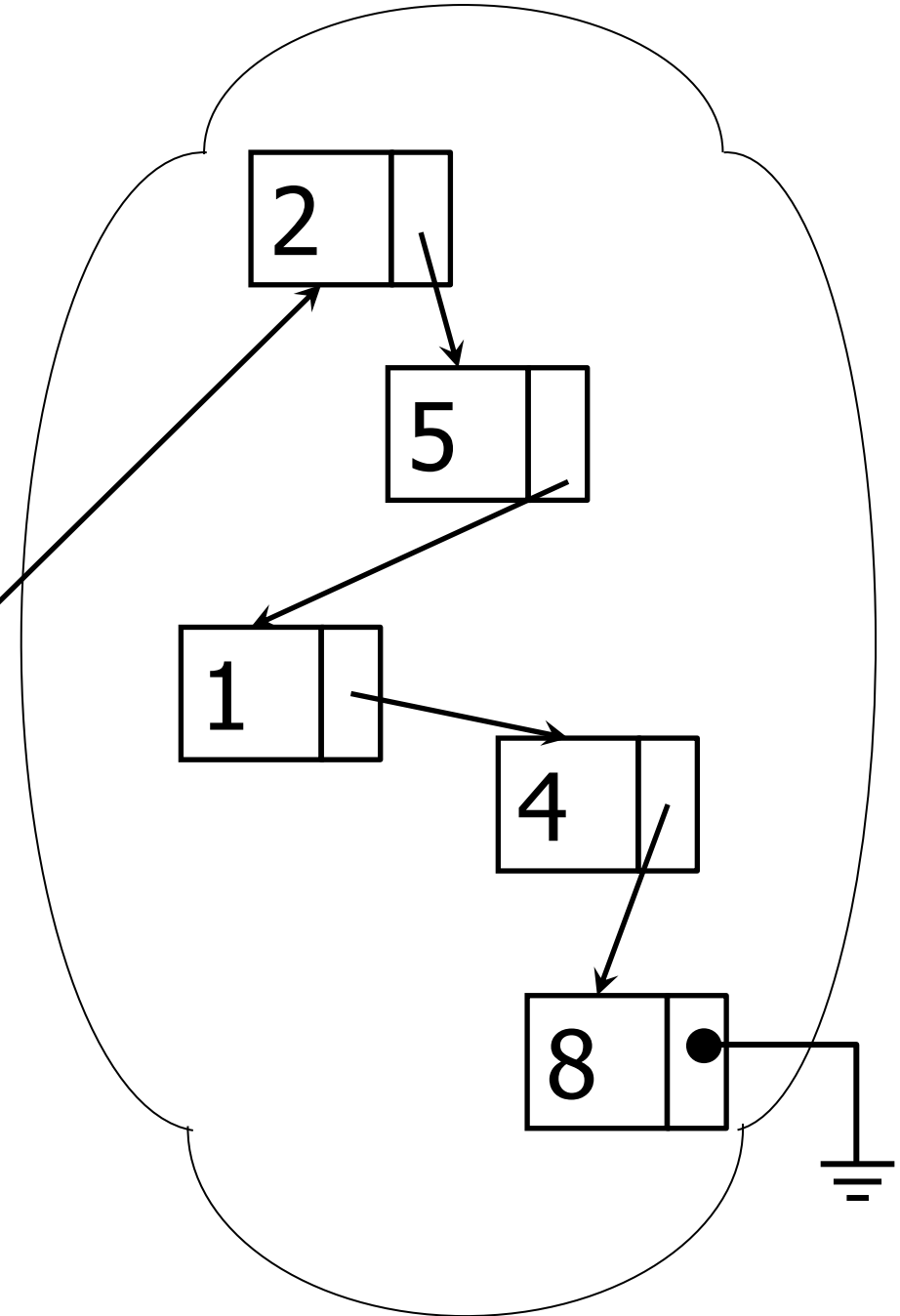
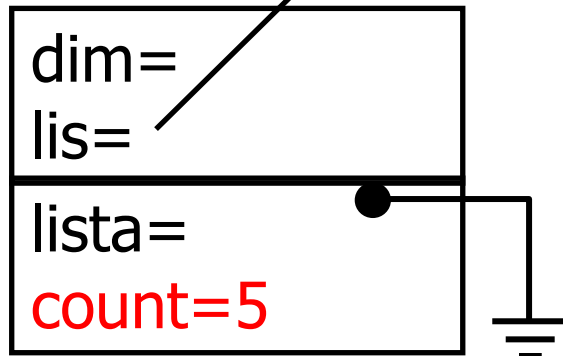
```
int main() {
    int dim;
    ListaDiElem lis;
    ... ..
    dim=Dimensione(lis);
    ... ..
}
```

```
int Dimensione(ListaDiElem lista)
{
    int count = 0;
    while( lista != NULL ) {
        lista = lista->prox;
        count++;
    }
    return count;
}
```



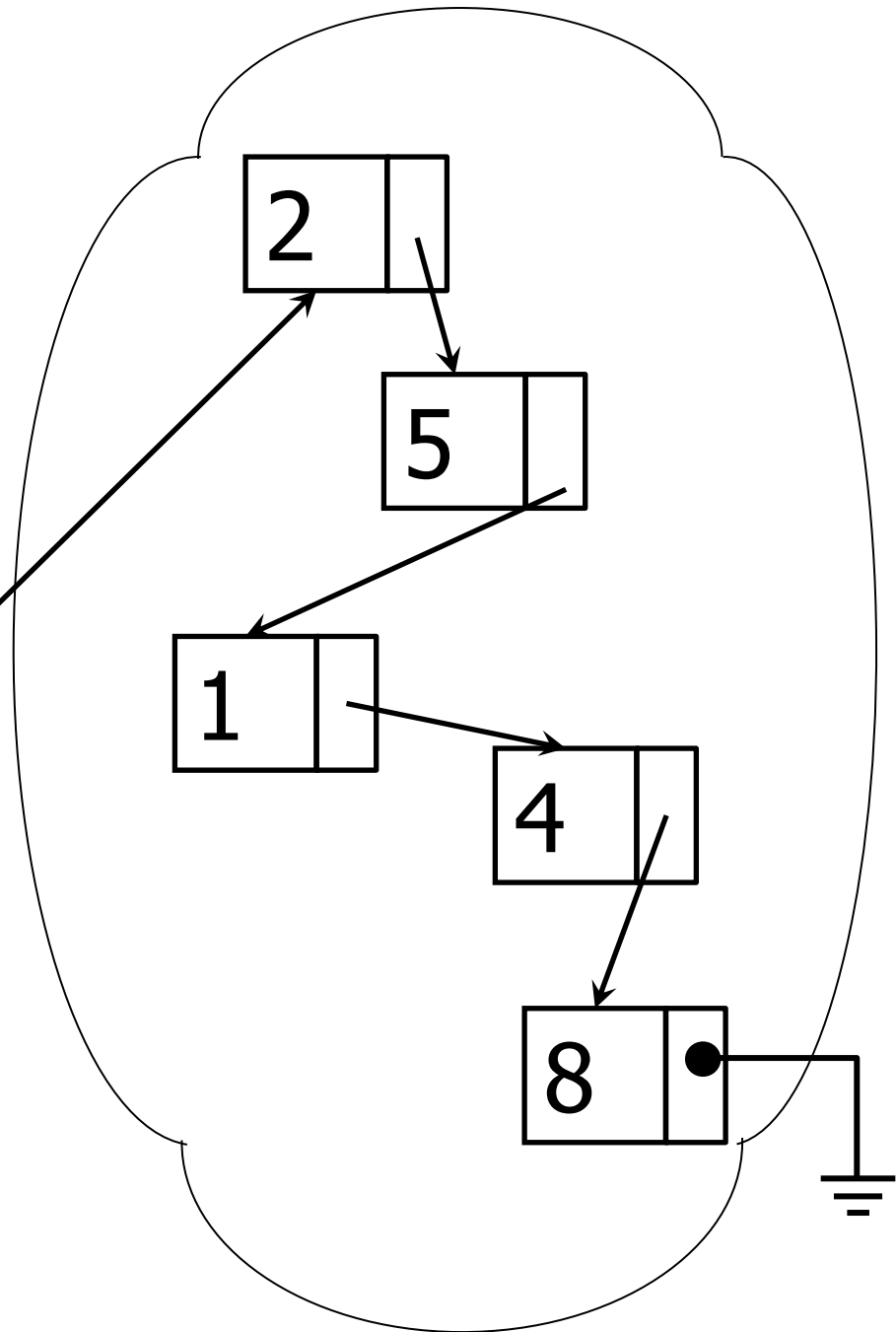
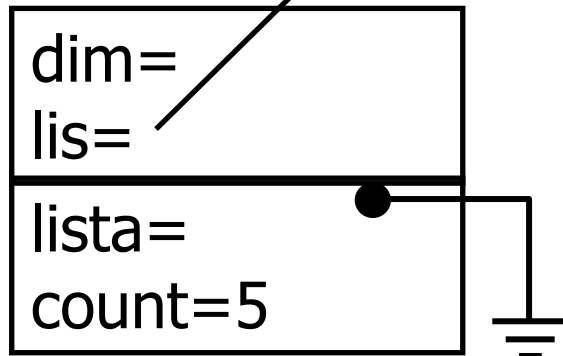

```
int main() {
    int dim;
    ListaDiElem lis;
    ... ..
    dim=Dimensione(lis);
    ... ..
}
```

```
int Dimensione(ListaDiElem lista)
{
    int count = 0;
    while( lista != NULL ) {
        lista = lista->prox;
        count++;
    }
    return count;
}
```



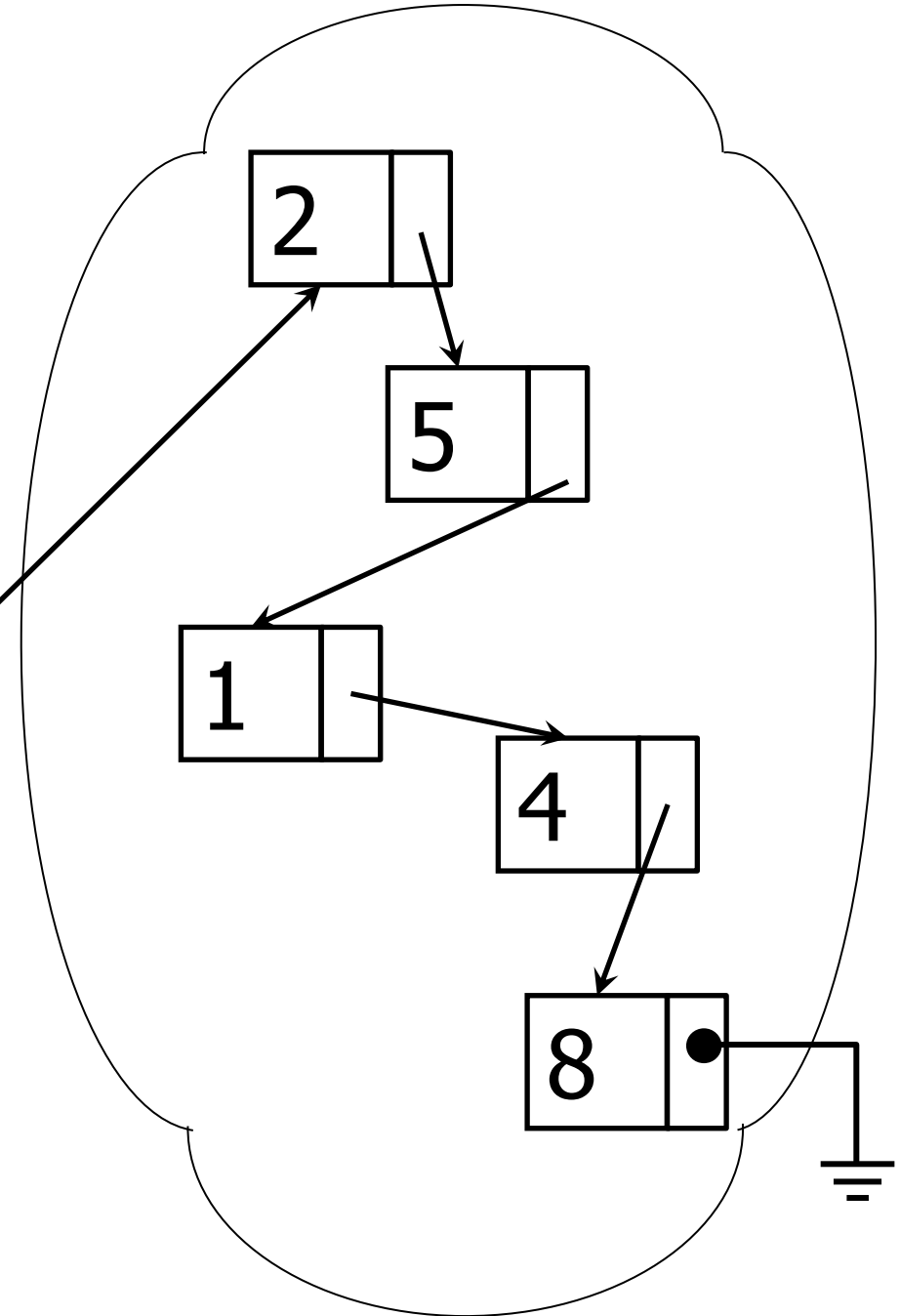
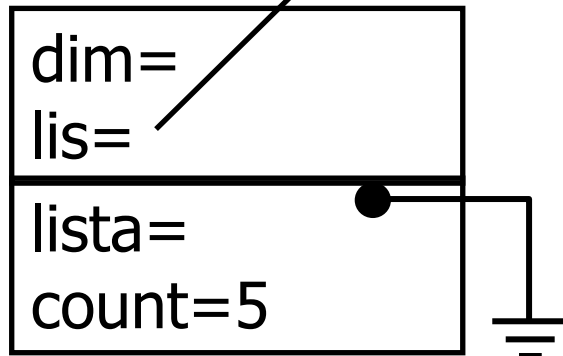
```
int main() {
    int dim;
    ListaDiElem lis;
    ... ..
    dim=Dimensione(lis);
    ... ..
}
```

```
int Dimensione(ListaDiElem lista)
{
    int count = 0;
    while( lista != NULL ) {
        lista = lista->prox;
        count++;
    }
    return count;
}
```



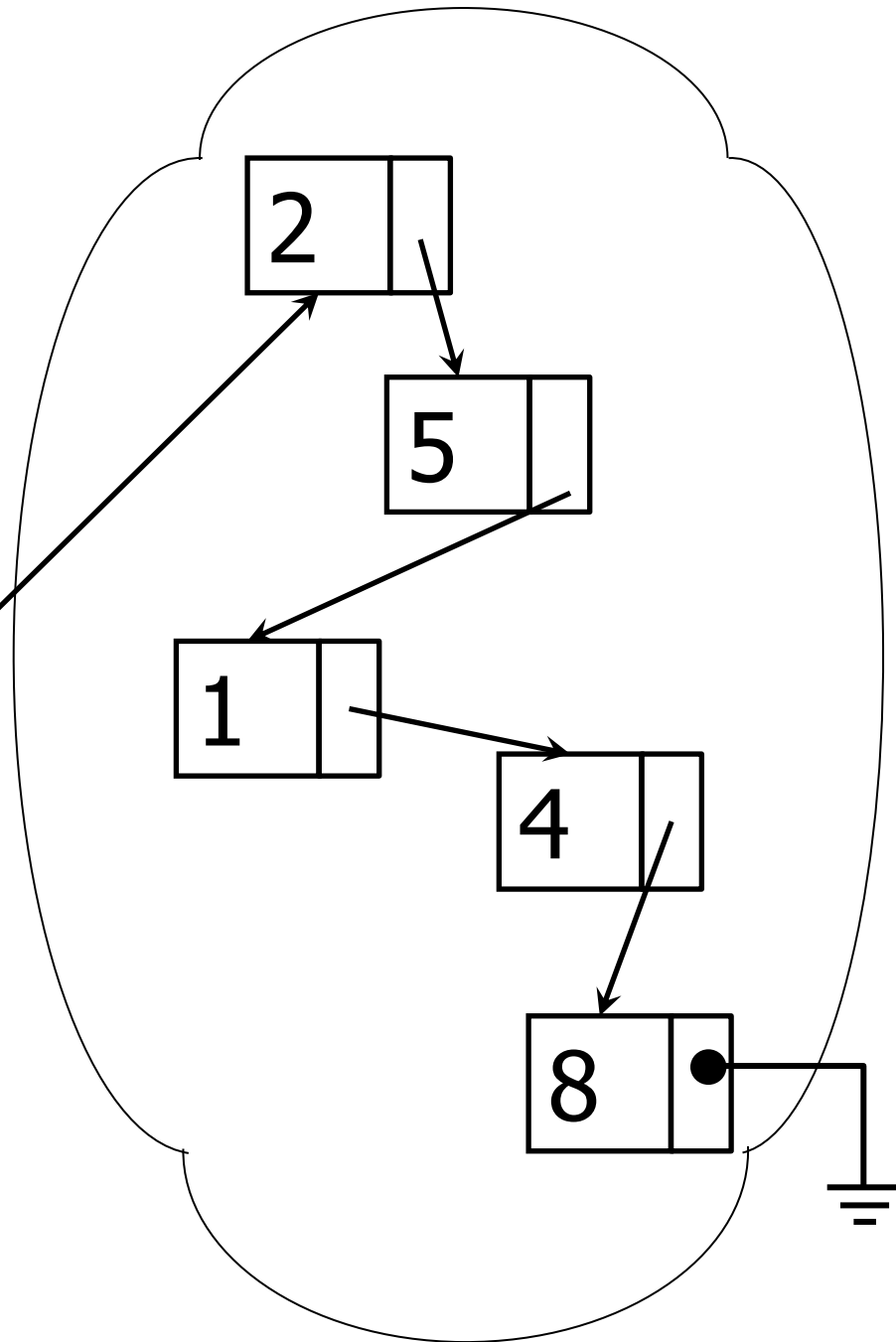
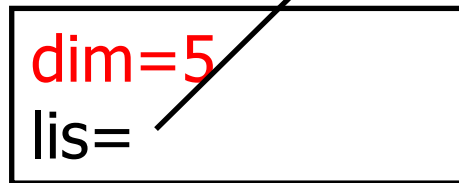
```
int main() {  
    int dim;  
    ListaDiElem lis;  
    ... ..  
    dim=Dimensione(lis);  
    ... ..  
}
```

```
int Dimensione(ListaDiElem lista)  
{  
    int count = 0;  
    while( lista != NULL ) {  
        lista = lista->prox;  
        count++;  
    }  
    return count;  
}
```



```
int main() {  
    int dim;  
    ListaDiElem lis;  
    ... ..  
    dim=Dimensione(lis);  
    ... ..  
}
```

```
int Dimensione(ListaDiElem lista)  
{  
    int count = 0;  
    while( lista != NULL ) {  
        lista = lista->prox;  
        count++;  
    }  
    return count;  
}
```



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;  
}
```

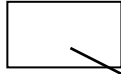
Chiamata: **lista1** = InsInTesta(**lista1**, elemento);

ATTENZIONE: l'inserimento modifica la lista

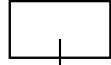
(non solo in quanto aggiunge un nodo, ma anche in quanto deve modificare il valore del puntatore al primo elemento *nell'ambiente del main*)

Visualizzazione

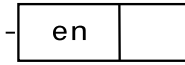
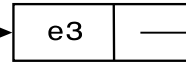
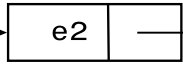
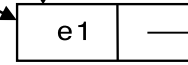
lista1



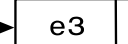
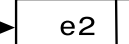
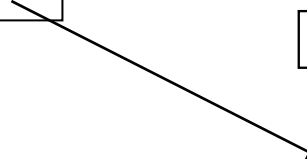
lista



punt

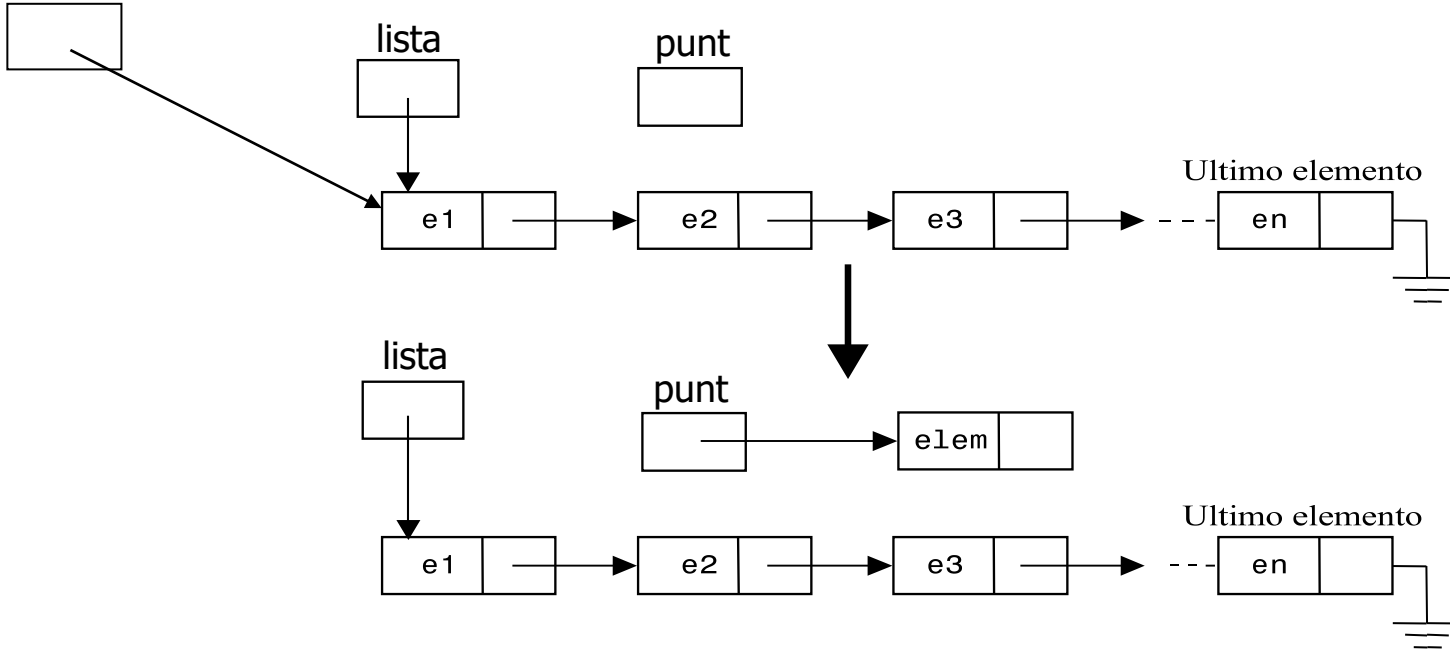


Ultimo elemento



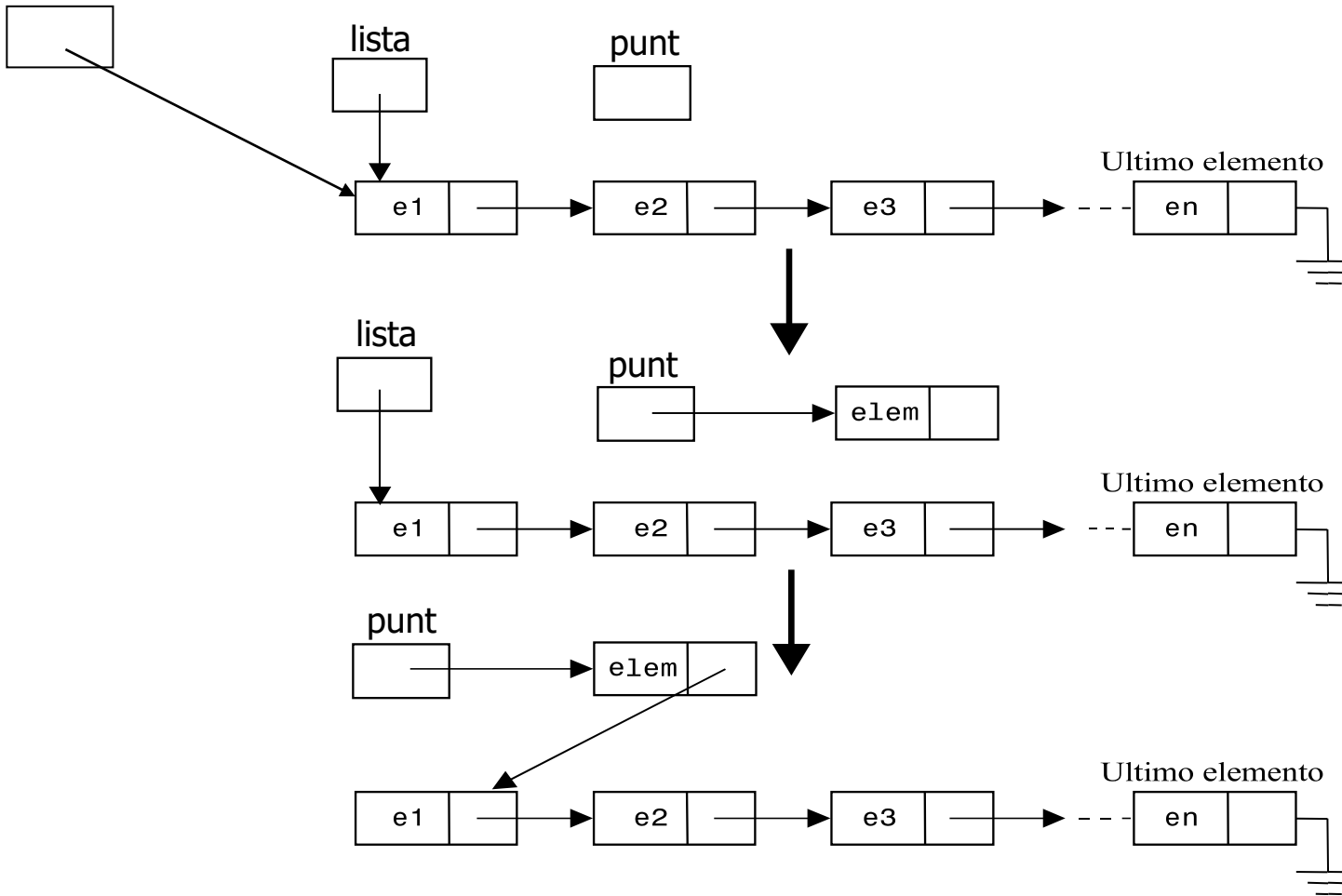
Visualizzazione

lista1



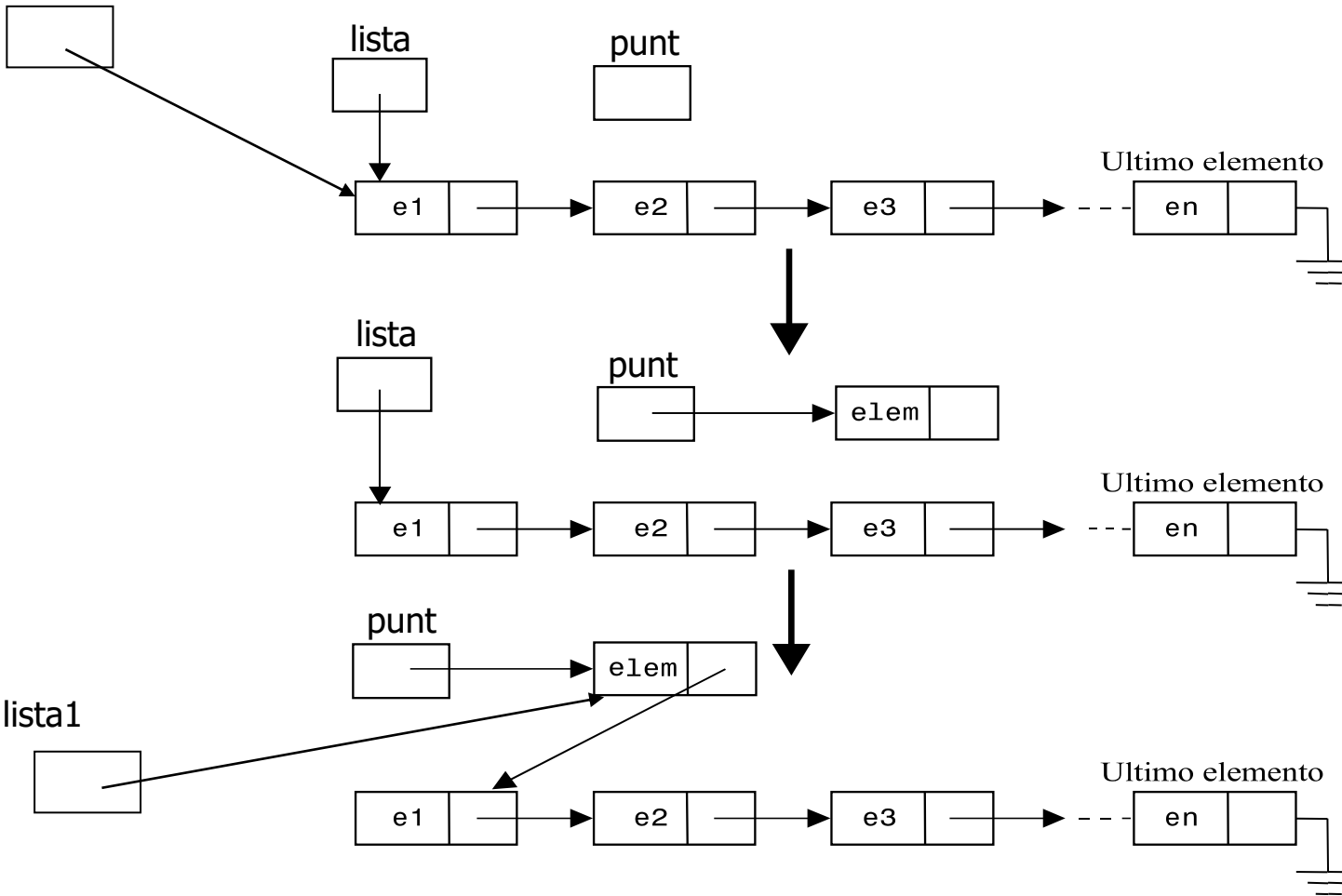
Visualizzazione

lista1



Visualizzazione

lista1



```
void inserisciInTestaRef(PNodo *p, int x) // questo è un doppio puntatore PNodo è già un puntatore!  
{  
    PNodo p_new;  
    p_new = (PNodo) malloc(sizeof(Nodo));  
    p_new->info = x;  
  
    p_new->next = *p; // p qui punta alla cella dello stack che contiene l'indirizzo della testa  
    della lista nello heap. Quindi, *p è l'indirizzo della cella dello stack che contiene l'indirizzo  
    lista nell'heap  
  
    // ora devo cambiare la testa della lista nello stack nel record di attivazione del main  
    // *p è l'indirizzo della cella dello stack nel record di attivazione del main che contiene il  
    puntatore alla lista.  
    // p_new è l'indirizzo della nuova testa e può essere scritto in quella cella  
    *p = p_new;  
}
```

**L'alternativa è passare per riferimento la testa della lista,
questo richiede di utilizzare un doppio puntatore.**

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 ( lista==NULL )
        return punt;           /* => punt è la nuova lista */
    else {
        while(cur->prox!= NULL ) /* Trova l'ultimo nodo */
            cur = cur->prox;
        cur->prox = punt;       /* Aggancio all'ultimo nodo */
    }
    return lista;
}
```

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

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 ( lista==NULL )          /* => 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** = InsInFondo(**lista1**, elemento);

Serve che la funzione **InsInFondo** restituisca la nuova lista perché la lista passata in ingresso potrebbe essere vuota.

In questo caso occorrerebbe quindi un inserimento in testa.

Altrimenti modifiche all'interno o in coda alla lista possono essere inserite anche con una funzione void.

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 ( lista==NULL )          /* => 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** = InsInFondo(**lista1**, elemento);

È necessario mantenere il puntatore alla testa (lista in questo caso) che viene restituito senza essere modificato. Cur invece viene modificato per scorrere la lista e quindi se lo restituissi, perderei la lista nell'invocazione

Inserimento in ultima posizione (ric.)

```
ListaDiElem InsInFondo( ListaDiElem lista, TipoElemento elem )
{
    ListaDiElem punt;
    if( lista==NULL ) {
        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 ultima posizione (ric.)

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

L'invocazione garantisce che la lista rimanga concatenate, perchè il nodo restituito dalla chiamata ricorsiva, viene agganciato al nodo corrente!

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

Inserimento in ultima posizione (ric.)

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

return lista deve andare in una riga separata, altrimenti non riesci a concatenare e restituire!

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

Inserimento in ultima posizione (ric.)

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

```
{
```

```
    ListaDiElem punt;
```

```
    if( lista==NULL ) {
```

```
        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);

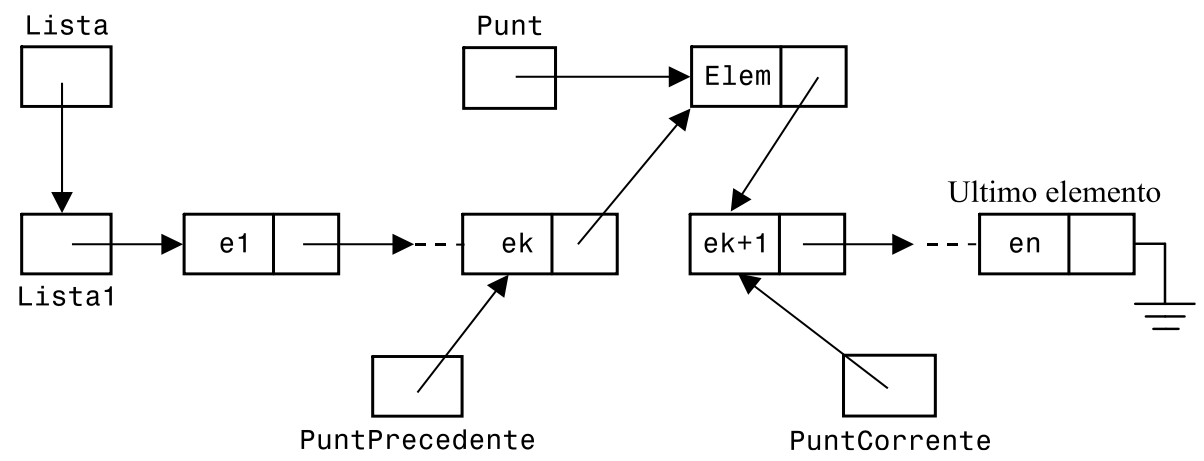
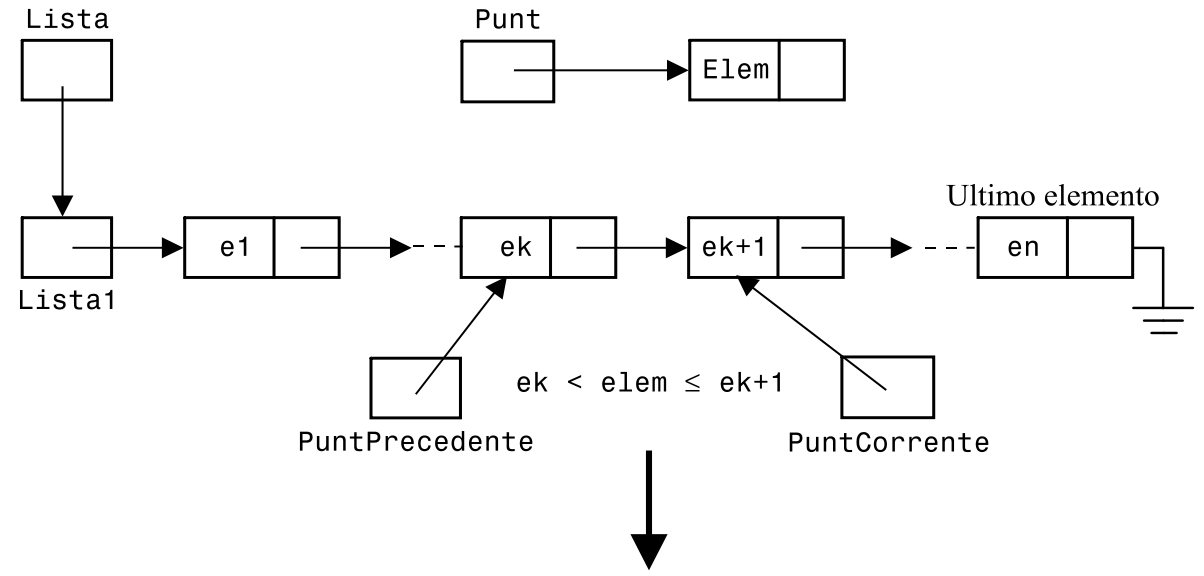
Inserimento in ultima posizione (ric.)

```
ListaDiElem InsInFondo( ListaDiElem lista, TipoElemento elem ) {  
    if( lista==NULL )  
        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;
    /* Cerco la posizione giusta per inserire il nodo elem */
    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
        return punt;                  /* Inserimento in testa alla lista */
}
```

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



Inserimento in lista ordinata no duplicati (iter)

```
Lista ins_ord(Lista l, int val)
{
    Lista prec = NULL, curr = l, p;
    if (l == NULL)// lista vuota
        return ins_testa(l, val);
    // popolo in ordine crescente, continua a cercare finchè curr non è maggiore,
    while(curr != NULL && curr->v < val)    {
        prec = curr;
        curr = curr->next;
    }
    if(curr->v == val) // in questo modo non inserisco duplicati
        return l;

    p = (Lista) malloc(sizeof(Nodo));
    p->v = val;
    p->next = curr;

    // questo vuol dire che devo scrivere nella testa perchè curr->v < val
    if(prec == NULL)
        return p;
    prec->next = p; // concateno se non era un caso precedente
    return l;
}
```

Inserimento in lista ordinata (ric.)

```
Lista ins_ord_ric(Lista l, int val)
{
    // primo caso base, la lista è vuota, inserisco
    if (l == NULL)
        return ins_testa(l, val);

    // secondo caso base, la testa di l ha un valore maggiore, devo inserire in testa per fare
    // inserimento crescente
    if(l->v > val)
    {
        Lista p = (Lista) malloc(sizeof(Nodo));
        p->v = val;
        p->next = l;
        return p;
    }

    // terzo caso base (per evitare duplicati) non inserisco se trovo già il valore
    if(l->v == val)
        return l;

    // altrimenti, devo continuare l'inserimento sulla lista che segue
    l->next = ins_ord_ric(l->next, val);
    return l;
}
```


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

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

Cancellazione di un elemento

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

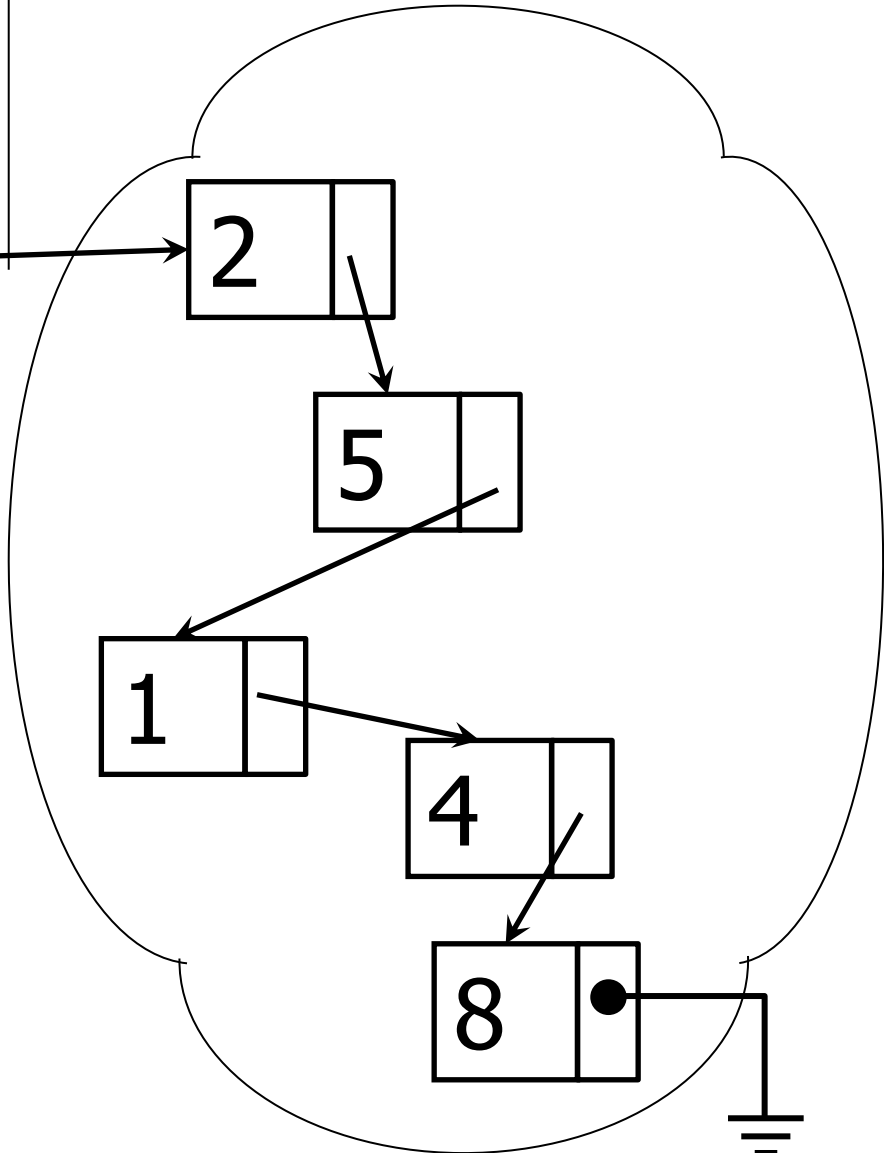
*Che cosa succede se
nella lista ci sono
valori duplicati?*

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

```
ListaDiElem Cancellala( ListaDiElem lista, int elem ) {  
    ListaDiElem puntTemp;  
    if( lista!=NULL)  
        if( lista->info == elem ) {  
            puntTemp = lista->prox;  
            free( lista );  
            return puntTemp;  
        } else  
            lista->prox = Cancellala( lista->prox, elem );  
    return lista;  
}
```

```
int main() {  
    ListaDiElem lista1;  
    ... ..  
    lista1 = Cancellala( lista1, 4 );  
    ... ..  
}
```

lista1 =



```

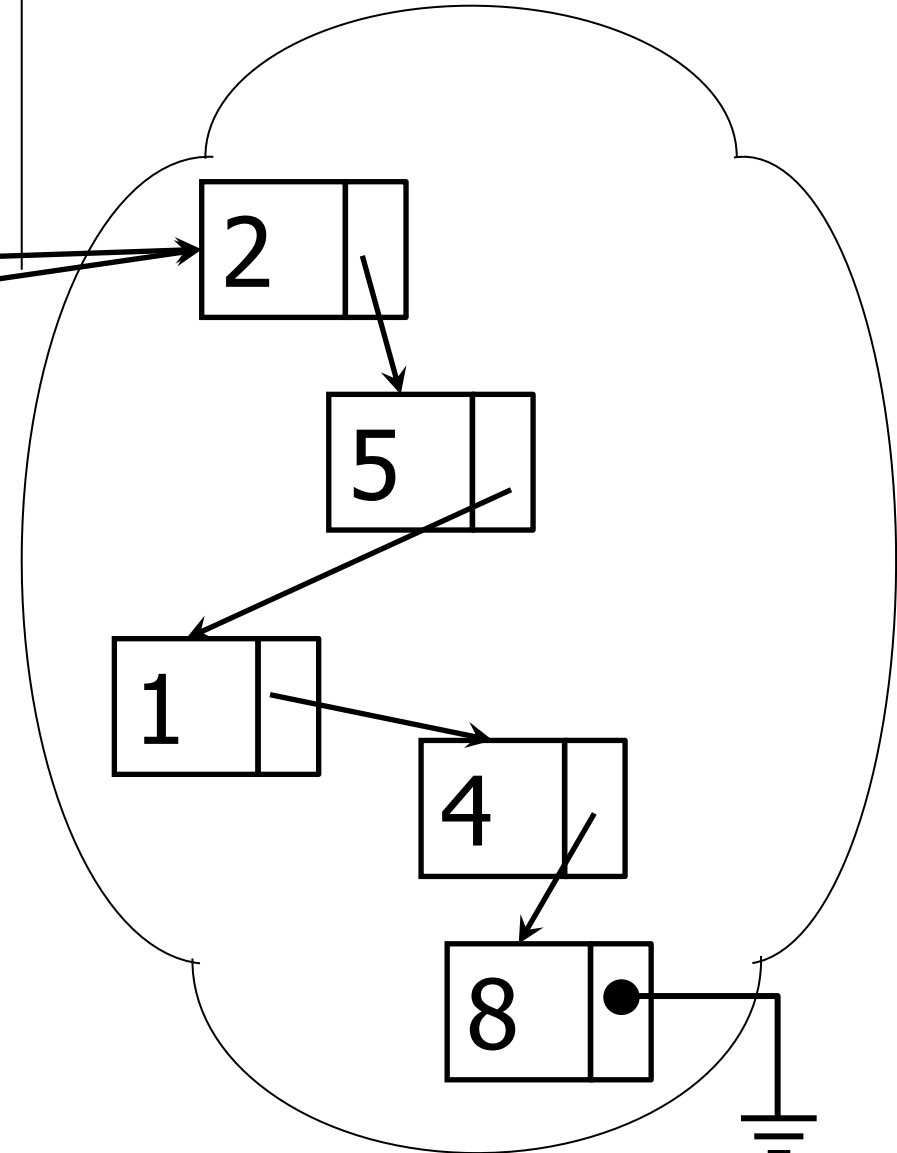
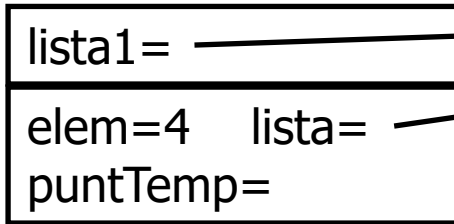
ListaDiElem Cancella( ListaDiElem lista, int elem ) {
    ListaDiElem puntTemp;
    if( lista!=NULL)
        if( lista->info == elem ) {
            puntTemp = lista->prox;
            free( lista );
            return puntTemp;
        } else
            lista->prox = Cancella( lista->prox, elem );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

```

```

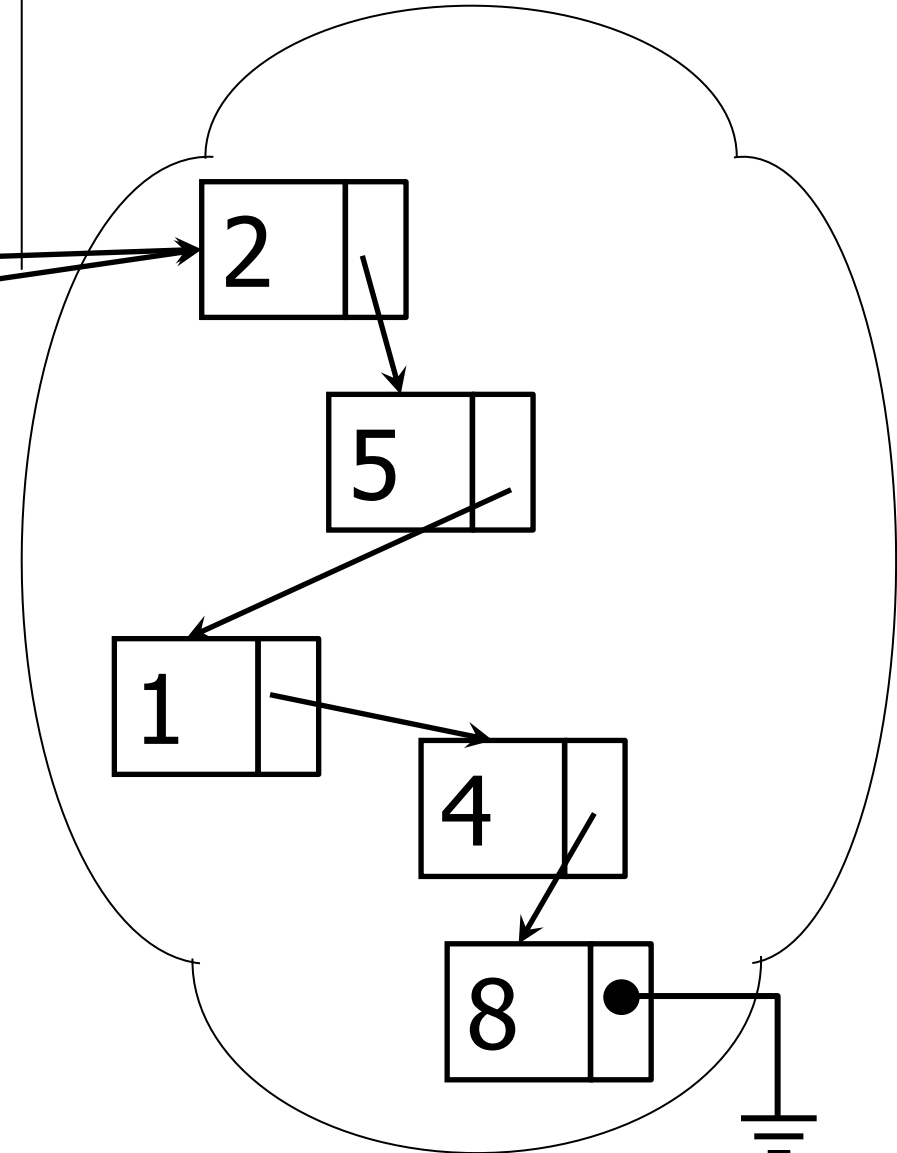
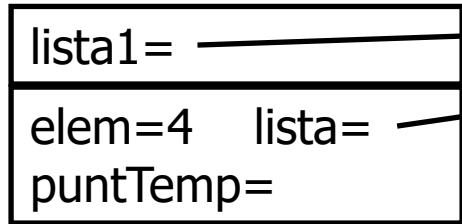
int main() {
    ListaDiElem lista1;
    ... ..
    lista1 = Cancella( lista1, 4 );
    ... ..
}

```



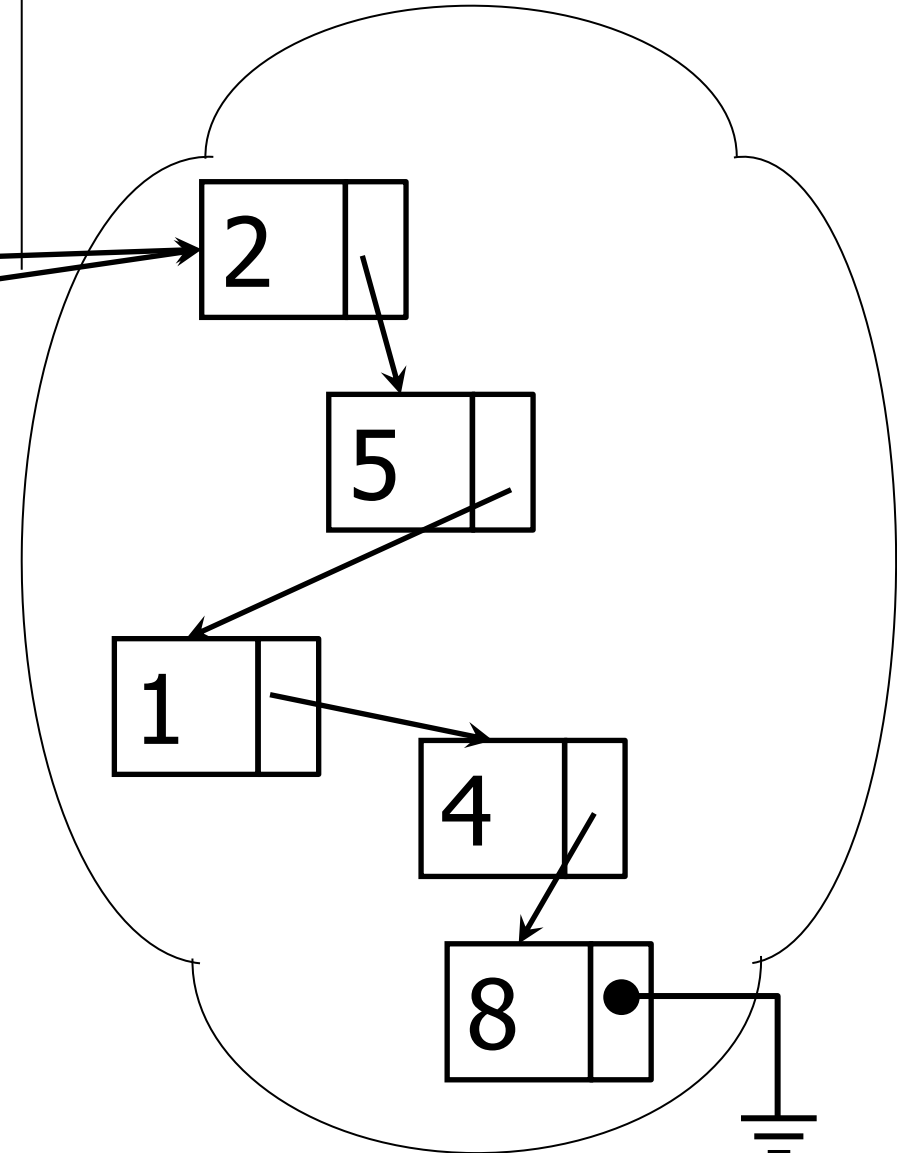
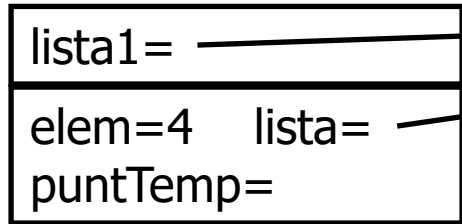
```
ListaDiElem Cancella( ListaDiElem lista, int elem ) {  
    ListaDiElem puntTemp;  
    if( lista!=NULL )  
        if( lista->info == elem ) {  
            puntTemp = lista->prox;  
            free( lista );  
            return puntTemp;  
        } else  
            lista->prox = Cancella(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

```
int main() {  
    ListaDiElem lista1;  
    ... ..  
    lista1 = Cancella( lista1, 4 );  
    ... ..  
}
```



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

```
int main() {  
    ListaDiElem lista1;  
    ... ..  
    lista1 = Cancella( lista1, 4 );  
    ... ..  
}
```



```

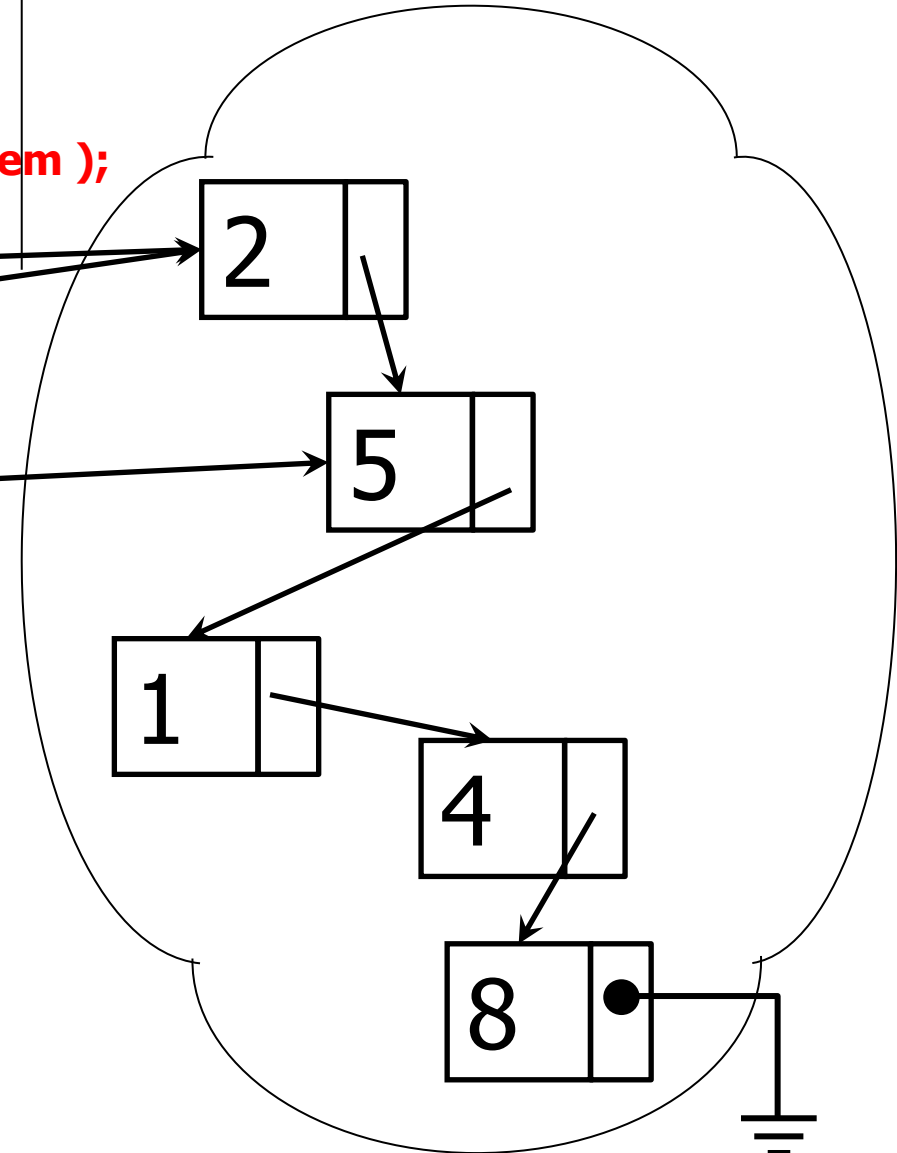
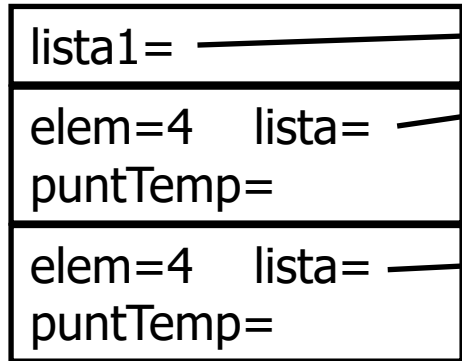
ListaDiElem Cancellala( ListaDiElem lista, int elem ) {
    ListaDiElem puntTemp;
    if( lista!=NULL)
        if( lista->info == elem ) {
            puntTemp = lista->prox;
            free( lista );
            return puntTemp;
        } else
            lista->prox = Cancellala( lista->prox, elem );
    return lista;
}

```

```

int main() {
    ListaDiElem lista1;
    ... ..
    lista1 = Cancellala( lista1, 4 );
    ... ..
}

```




```

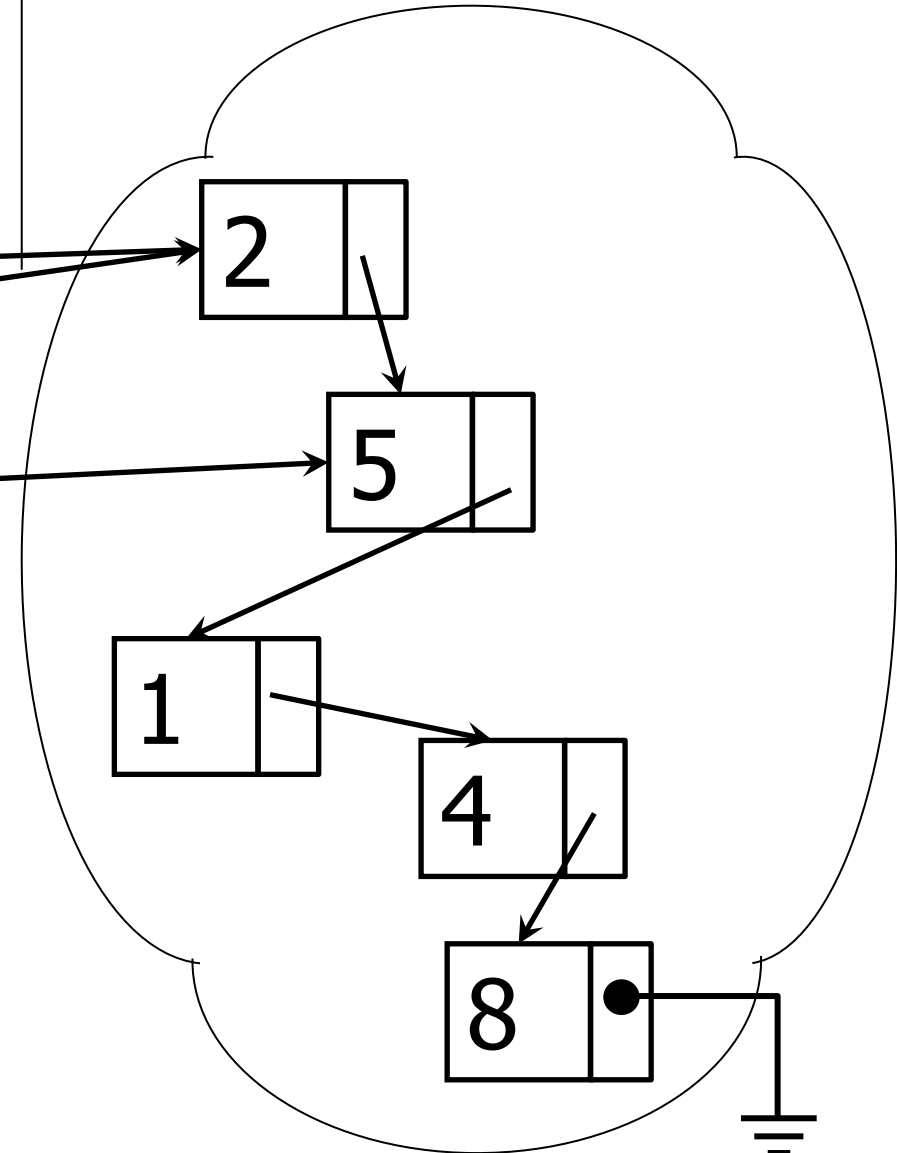
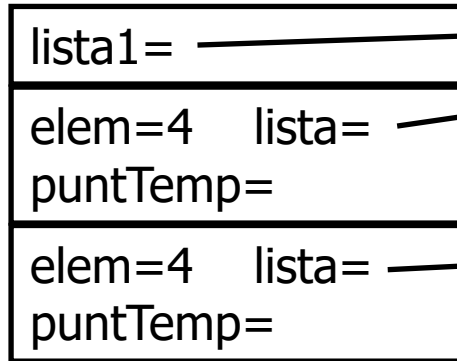
ListaDiElem Cancelli( ListaDiElem lista, int elem ) {
    ListaDiElem puntTemp;
    if( lista!=NULL)
        if( lista->info == elem ) {
            puntTemp = lista->prox;
            free( lista );
            return puntTemp;
        } else
            lista->prox = Cancelli( lista->prox, elem );
    return lista;
}

```

```

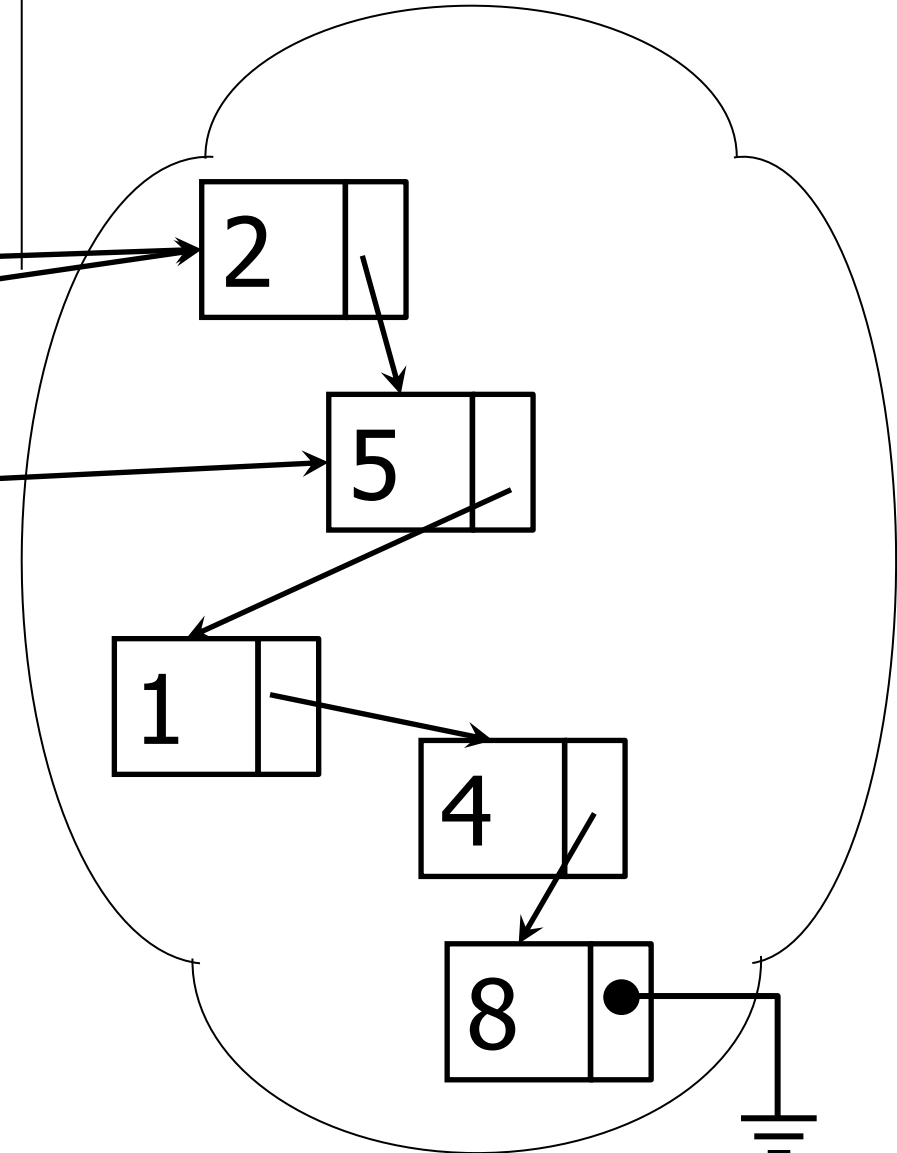
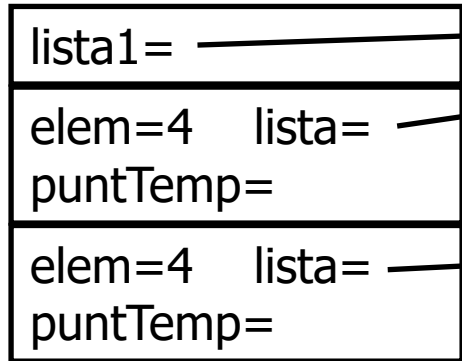
int main() {
    ListaDiElem lista1;
    ... ..
    lista1 = Cancelli( lista1, 4 );
    ... ..
}

```



```
ListaDiElem Cancellazione( ListaDiElem lista, int elem ) {
    ListaDiElem puntTemp;
    if( lista!=NULL)
        if( lista->info == elem ) {
            puntTemp = lista->prox;
            free( lista );
            return puntTemp;
        } else
            lista->prox = Cancellazione( lista->prox, elem );
    return lista;
}
```

```
int main() {
    ListaDiElem lista1;
    ... ..
    lista1 = Cancellazione( lista1, 4 );
    ... ..
}
```



```

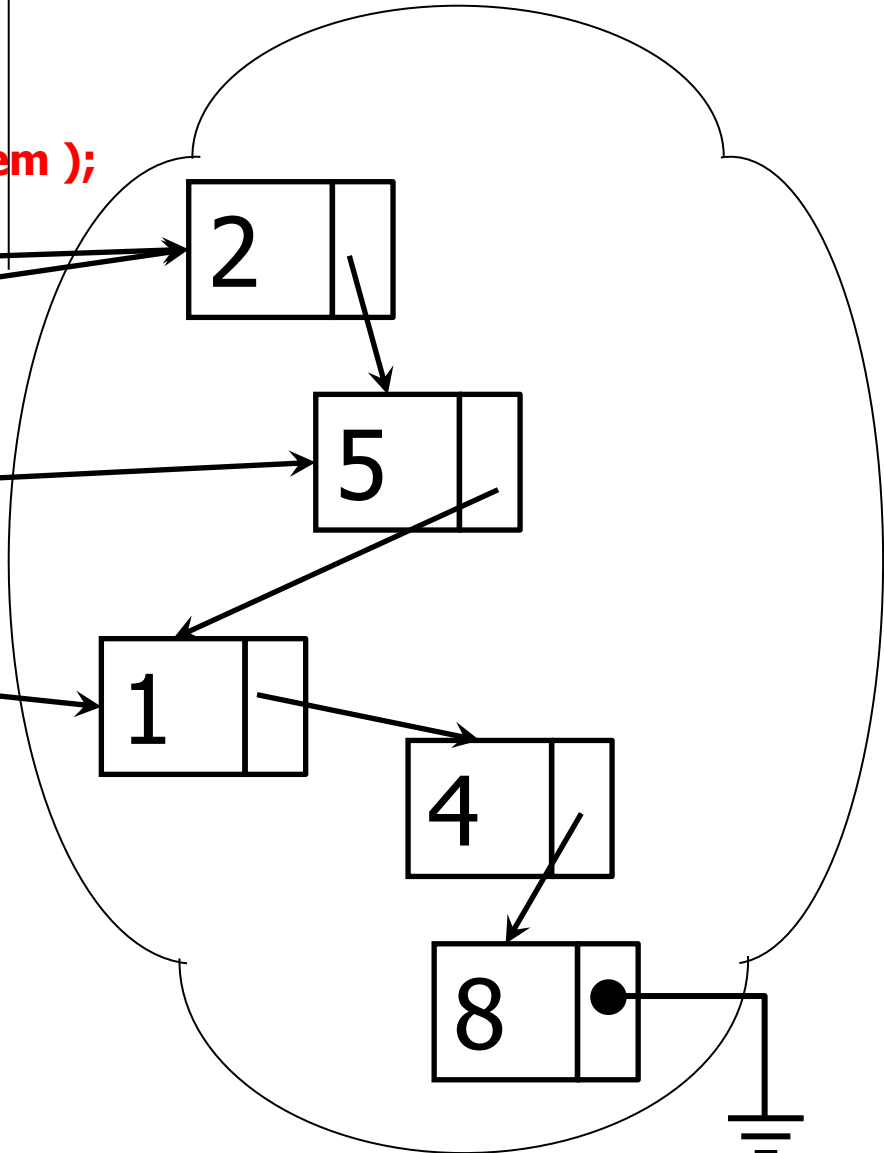
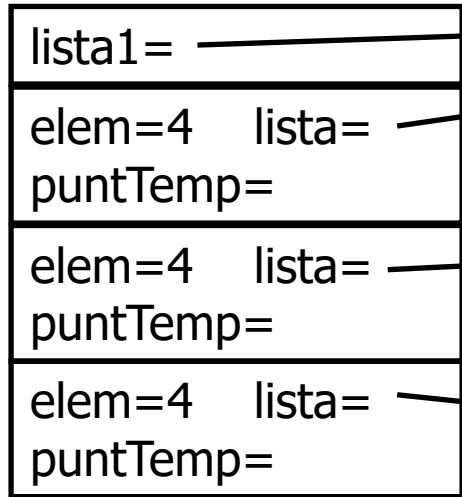
ListaDiElem Cancellala( ListaDiElem lista, int elem ) {
  ListaDiElem puntTemp;
  if( lista!=NULL)
    if( lista->info == elem ) {
      puntTemp = lista->prox;
      free( lista );
      return puntTemp;
    } else
      lista->prox = Cancellala( lista->prox, elem );
  return lista;
}

```

```

int main() {
  ListaDiElem lista1;
  ... ..
  lista1 = Cancellala( lista1, 4 );
  ... ..
}

```



```

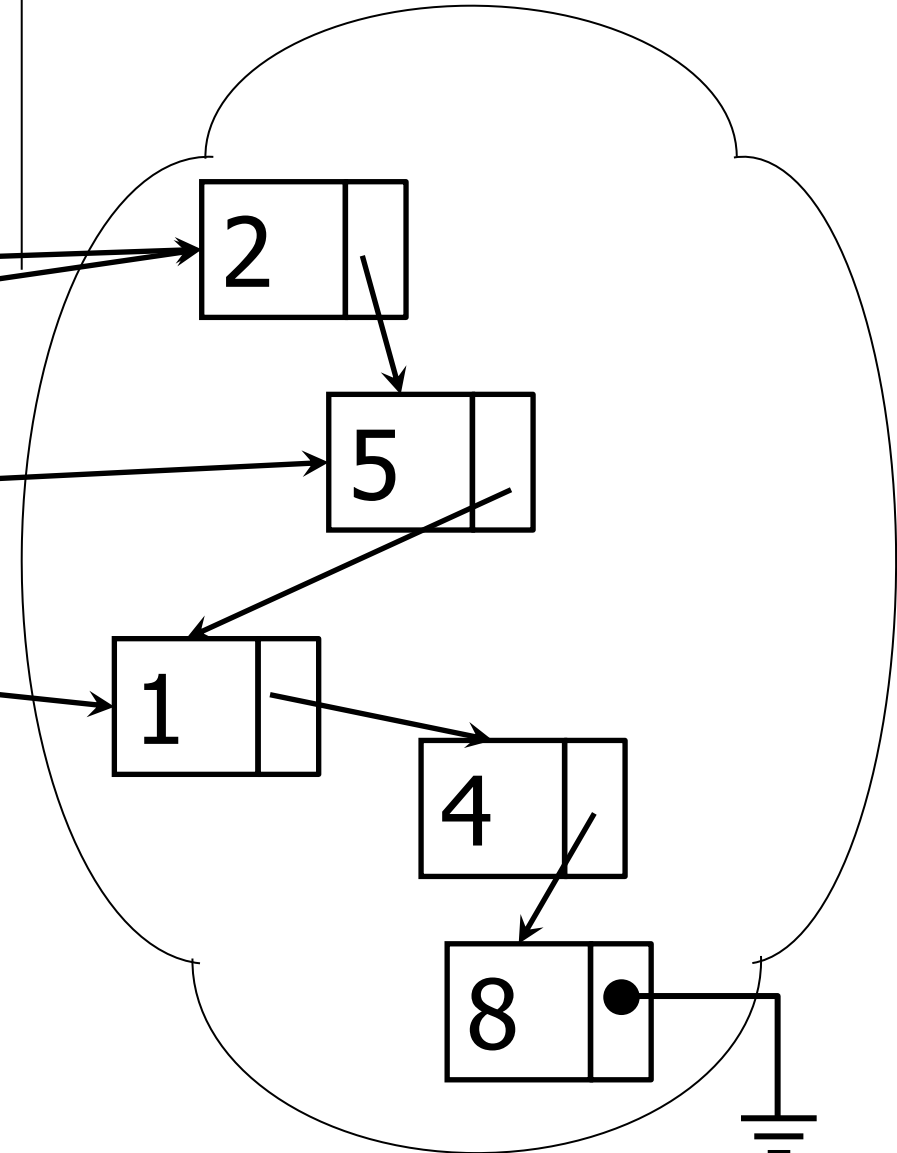
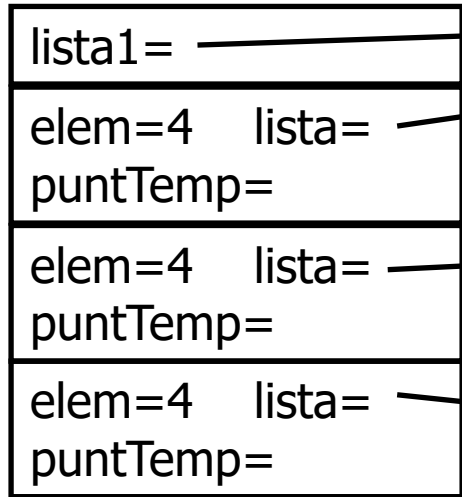
ListaDiElem Cancellala( ListaDiElem lista, int elem ) {
    ListaDiElem puntTemp;
    if( lista!=NULL )
        if( lista->info == elem ) {
            puntTemp = lista->prox;
            free( lista );
            return puntTemp;
        } else
            lista->prox = Cancellala( lista->prox, elem );
    return lista;
}

```

```

int main() {
    ListaDiElem lista1;
    ... ..
    lista1 = Cancellala( lista1, 4 );
    ... ..
}

```



```

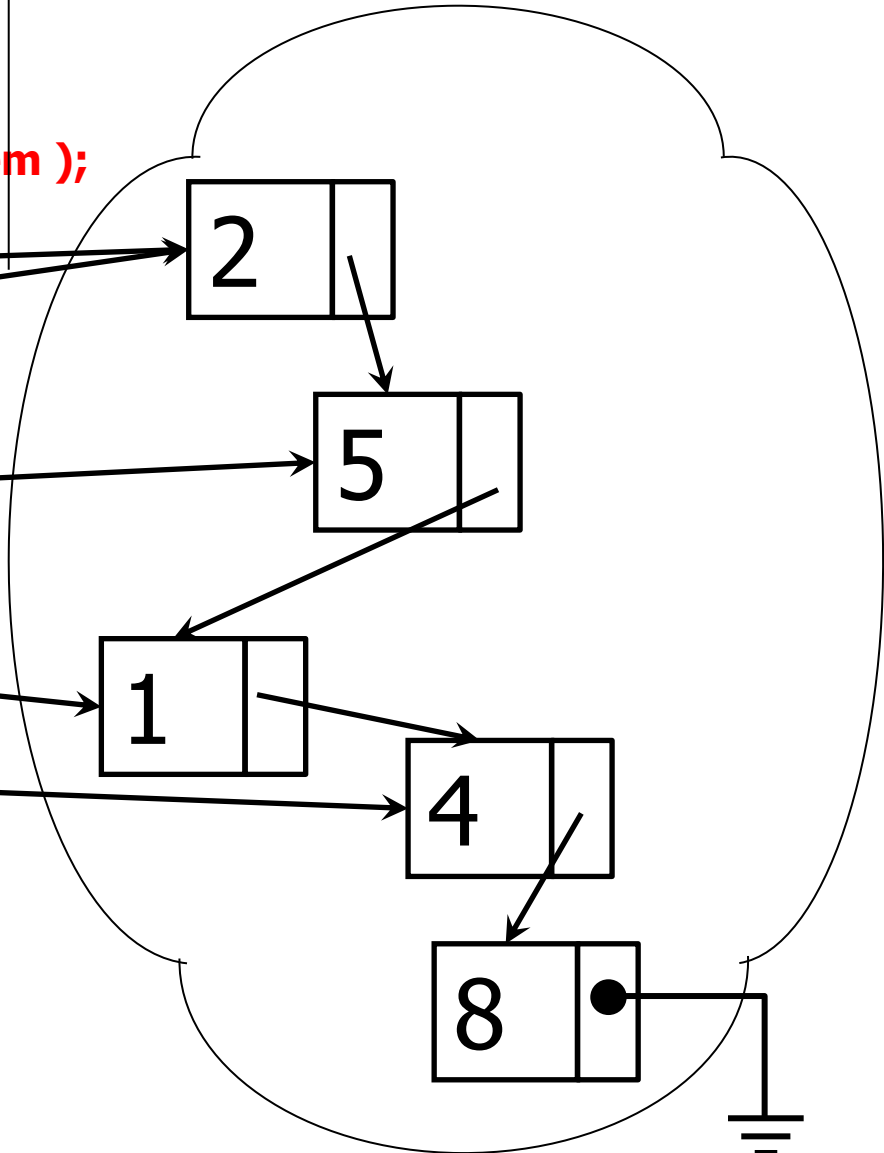
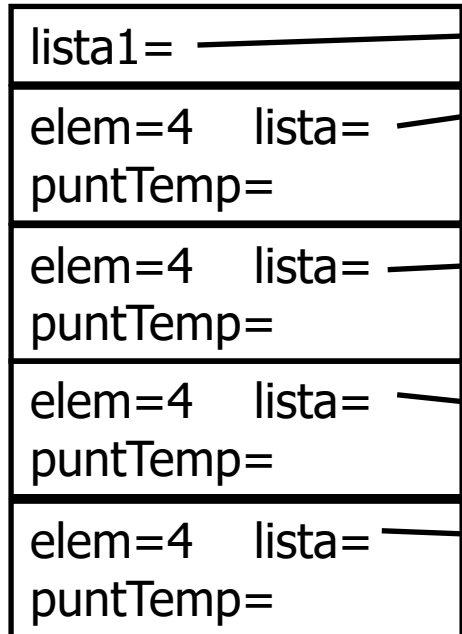
ListaDiElem Cancellala( ListaDiElem lista, int elem ) {
    ListaDiElem puntTemp;
    if( lista!=NULL)
        if( lista->info == elem ) {
            puntTemp = lista->prox;
            free( lista );
            return puntTemp;
        } else
            lista->prox = Cancellala( lista->prox, elem );
    return lista;
}

```

```

int main() {
    ListaDiElem lista1;
    ... ..
    lista1 = Cancellala( lista1, 4 );
    ... ..
}

```



```

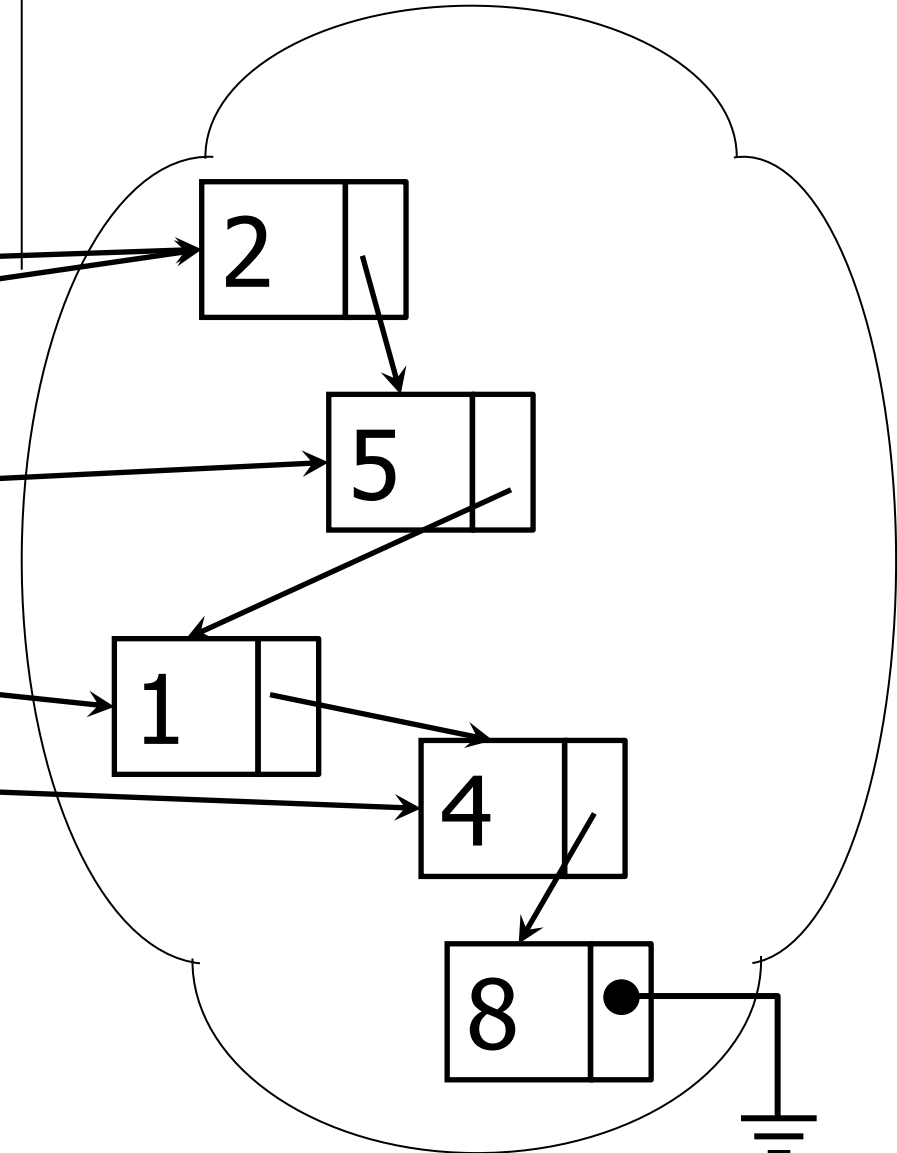
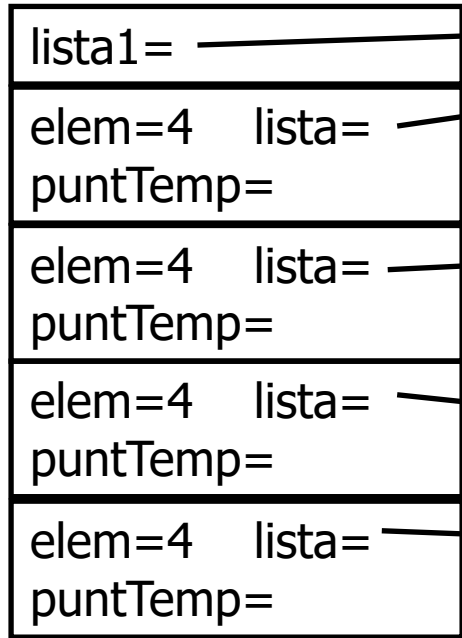
ListaDiElem Cancella( ListaDiElem lista, int elem ) {
    ListaDiElem puntTemp;
    if( lista!=NULL )
        if( lista->info == elem ) {
            puntTemp = lista->prox;
            free( lista );
            return puntTemp;
        } else
            lista->prox = Cancella( lista->prox, elem );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

```

```

int main() {
    ListaDiElem lista1;
    ... ..
    lista1 = Cancella( lista1, 4 );
    ... ..
}

```



```

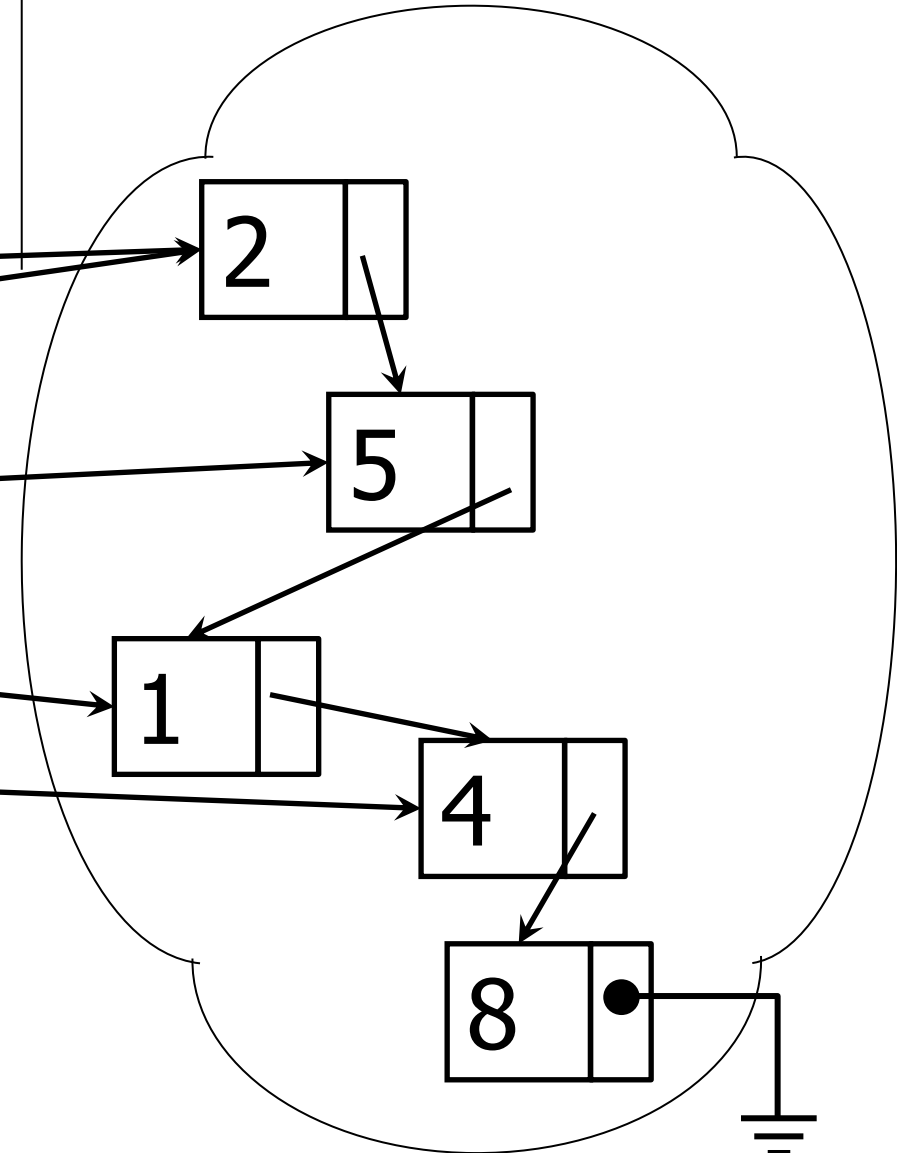
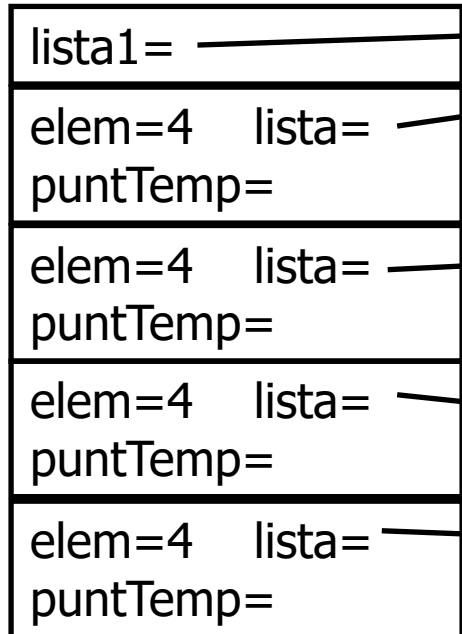
ListaDiElem Cancellala( ListaDiElem lista, int elem ) {
    ListaDiElem puntTemp;
    if( lista!=NULL)
        if( lista->info == elem ) {
            puntTemp = lista->prox;
            free( lista );
            return puntTemp;
        } else
            lista->prox = Cancellala( lista->prox, elem );
    return lista;
}

```

```

int main() {
    ListaDiElem lista1;
    ... ..
    lista1 = Cancellala( lista1, 4 );
    ... ..
}

```



```

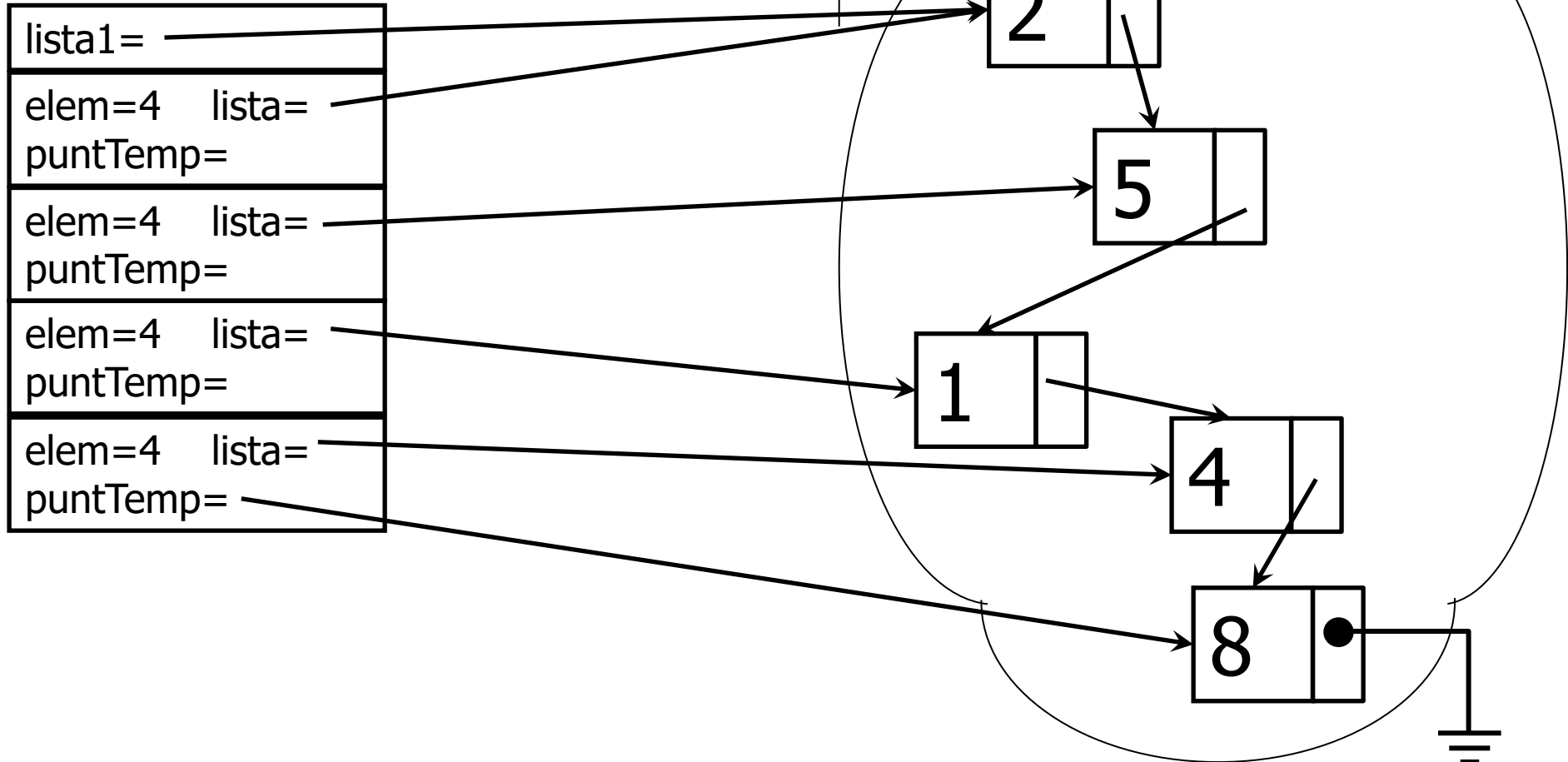
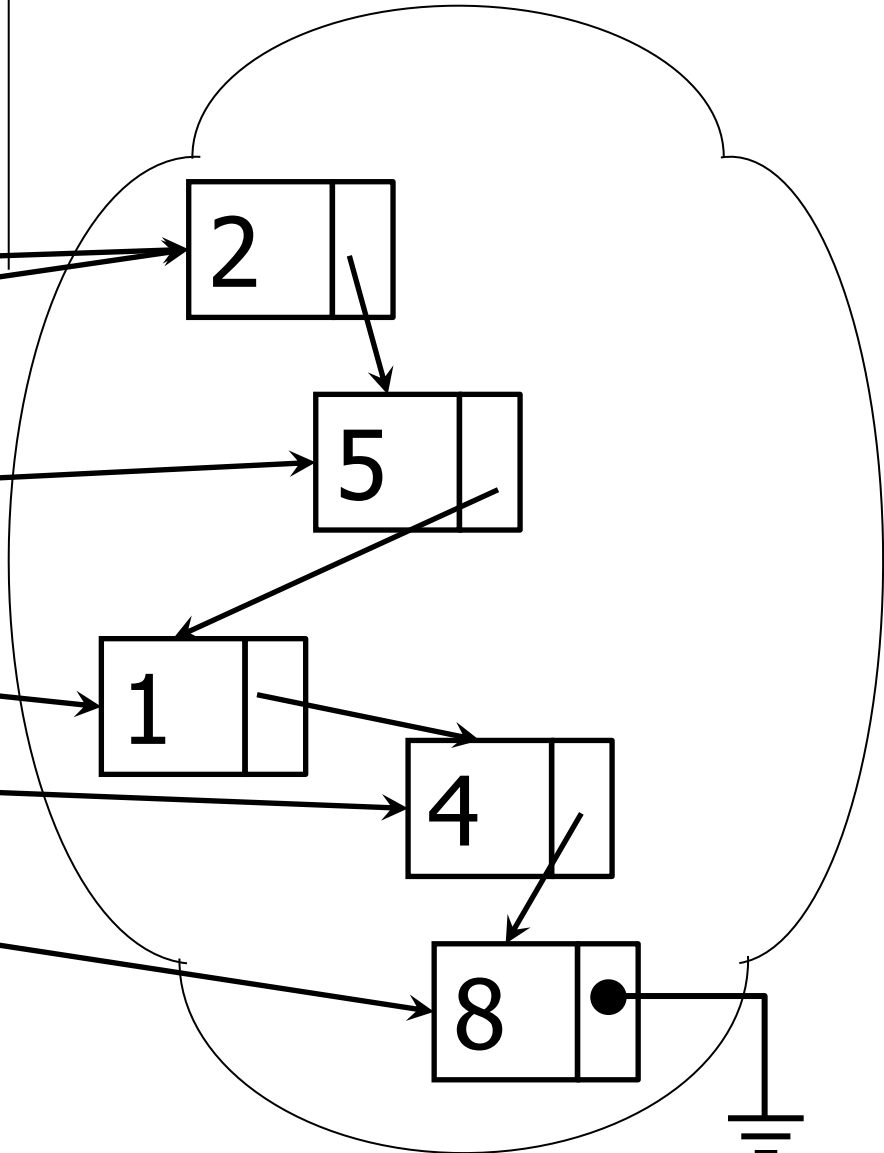
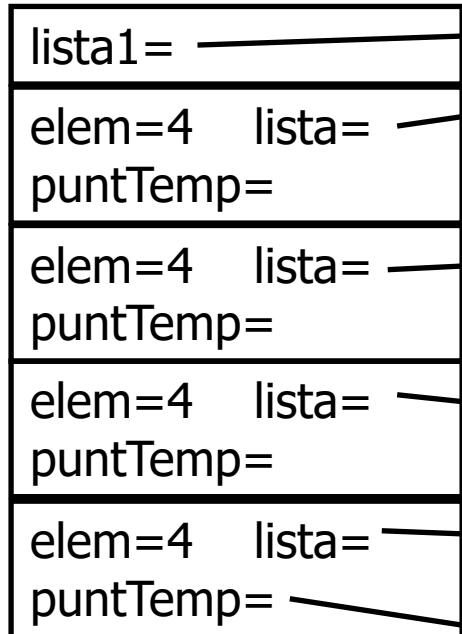
ListaDiElem Cancelli( ListaDiElem lista, int elem ) {
    ListaDiElem puntTemp;
    if( lista!=NULL)
        if( lista->info == elem ) {
            puntTemp = lista->prox;
            free( lista );
            return puntTemp;
        } else
            lista->prox = Cancelli( lista->prox, elem );
    return lista;
}

```

```

int main() {
    ListaDiElem lista1;
    ... ..
    lista1 = Cancelli( lista1, 4 );
    ... ..
}

```




```

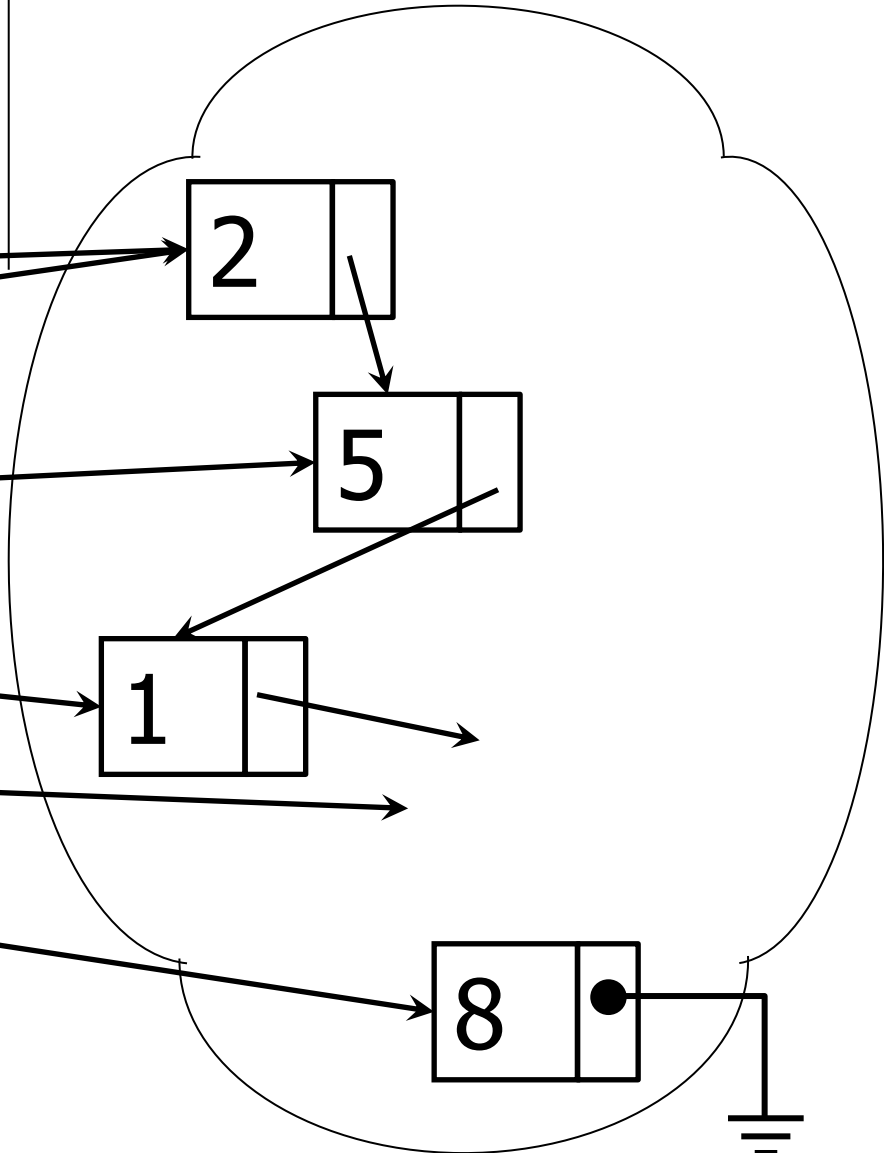
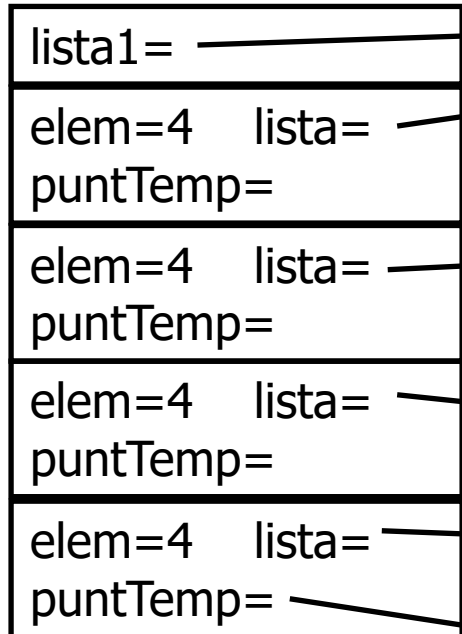
ListaDiElem Cancelli( ListaDiElem lista, int elem ) {
    ListaDiElem puntTemp;
    if( lista!=NULL)
        if( lista->info == elem ) {
            puntTemp = lista->prox;
            free( lista );
            return puntTemp;
        } else
            lista->prox = Cancelli( lista->prox, elem );
    return lista;
}

```

```

int main() {
    ListaDiElem lista1;
    ... ..
    lista1 = Cancelli( lista1, 4 );
    ... ..
}

```



```

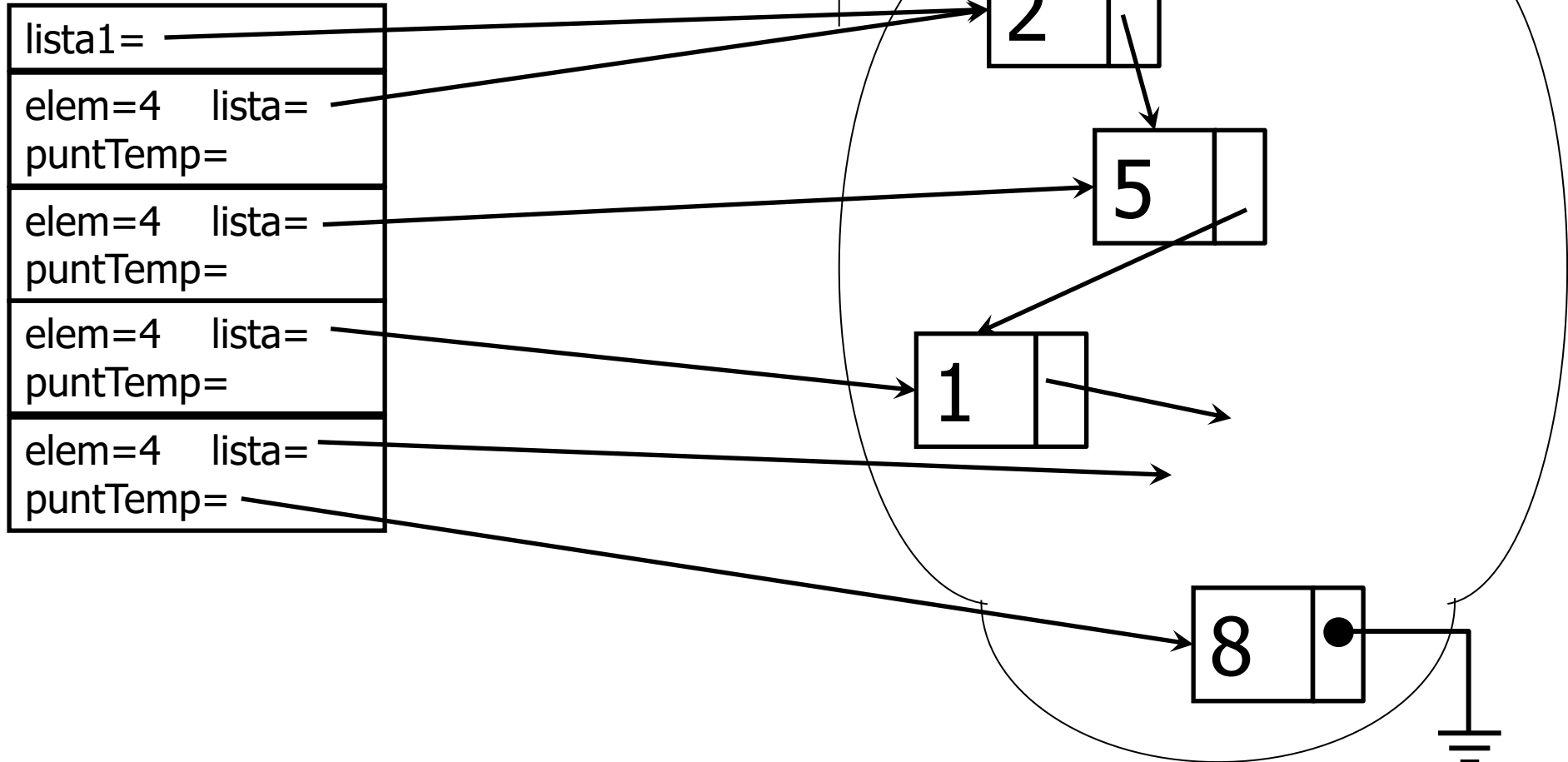
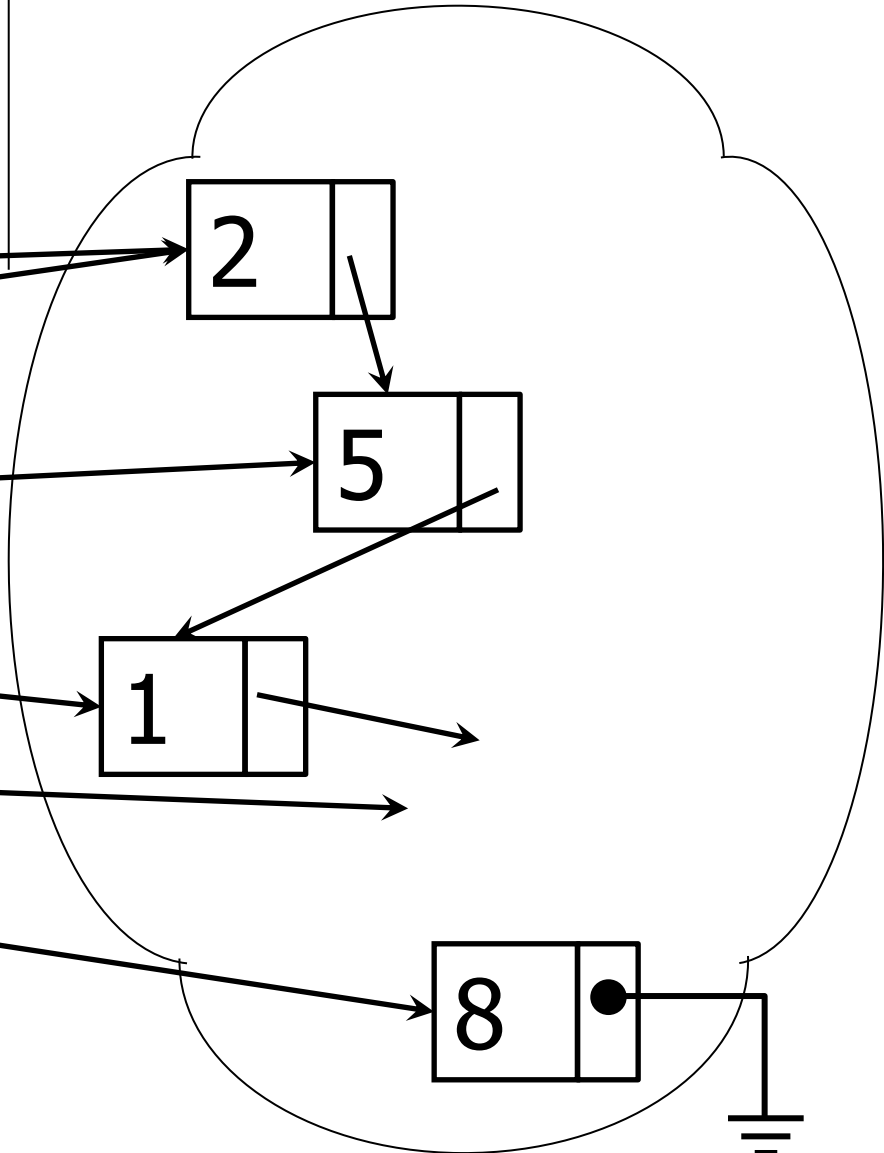
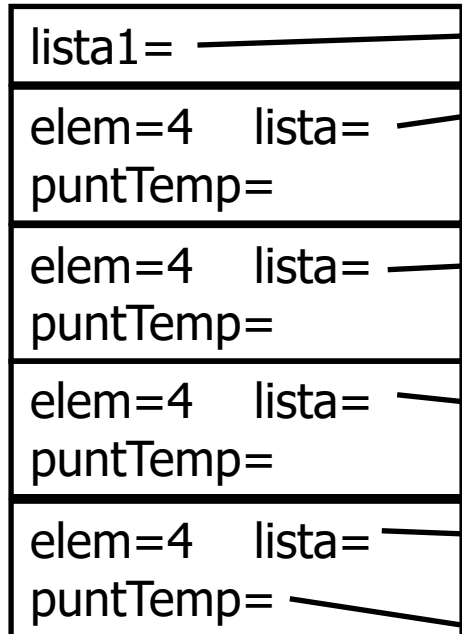
ListaDiElem Cancellala( ListaDiElem lista, int elem ) {
    ListaDiElem puntTemp;
    if( lista!=NULL)
        if( lista->info == elem ) {
            puntTemp = lista->prox;
            free( lista );
            return puntTemp;
        } else
            lista->prox = Cancellala( lista->prox, elem );
    return lista;
}

```

```

int main() {
    ListaDiElem lista1;
    ... ..
    lista1 = Cancellala( lista1, 4 );
    ... ..
}

```



```

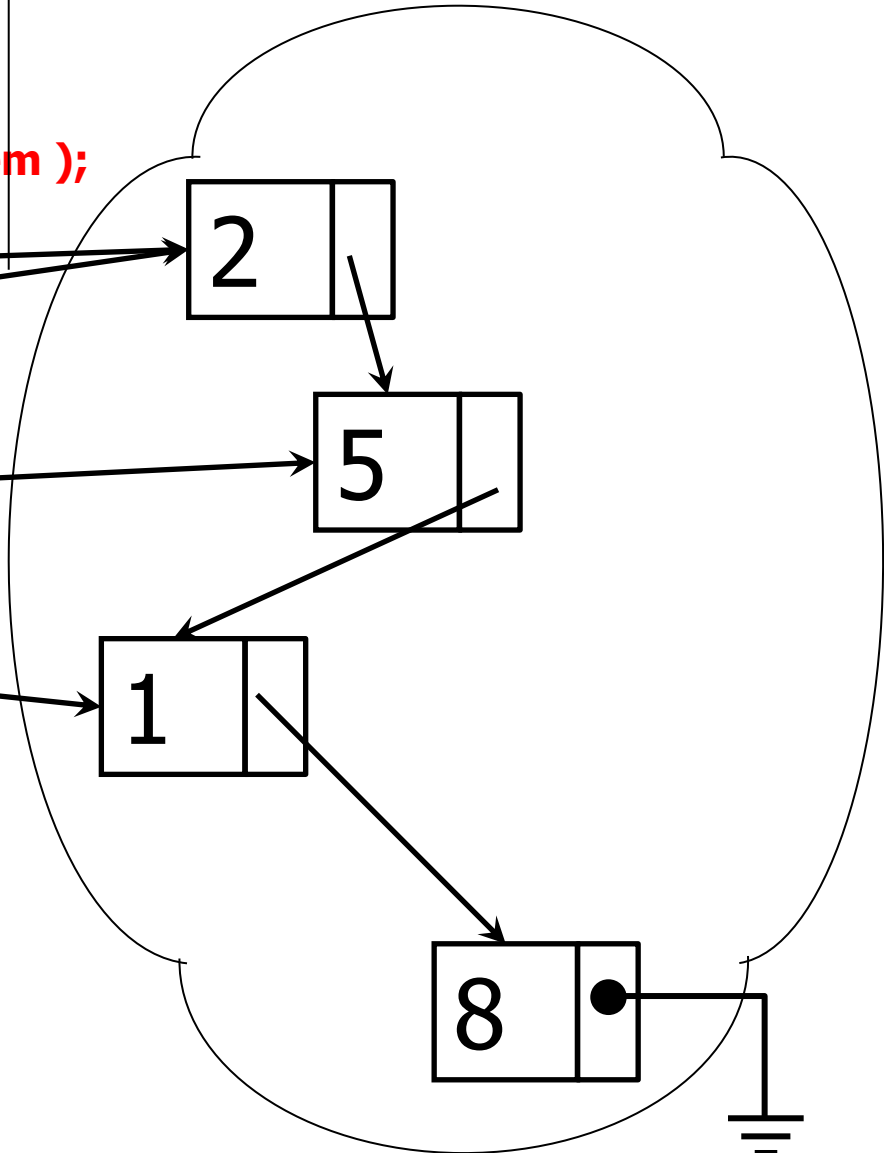
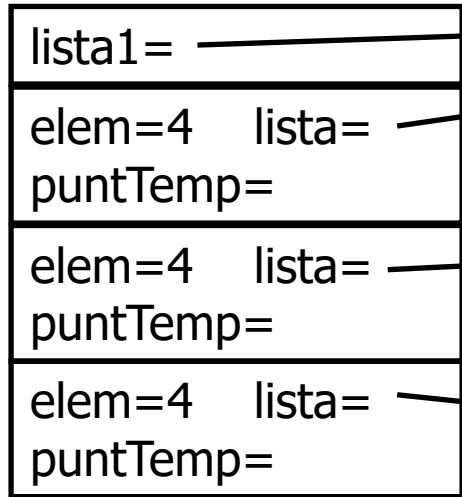
ListaDiElem Cancellala( ListaDiElem lista, int elem ) {
    ListaDiElem puntTemp;
    if( lista!=NULL)
        if( lista->info == elem ) {
            puntTemp = lista->prox;
            free( lista );
            return puntTemp;
        } else
            lista->prox = Cancellala( lista->prox, elem );
    return lista;
}

```

```

int main() {
    ListaDiElem lista1;
    ... ..
    lista1 = Cancellala( lista1, 4 );
    ... ..
}

```



```

ListaDiElem Cancellazione( ListaDiElem lista, int elem ) {
    ListaDiElem puntTemp;
    if( lista!=NULL)
        if( lista->info == elem ) {
            puntTemp = lista->prox;
            free( lista );
            return puntTemp;
        } else
            lista->prox = Cancellazione( lista->prox, elem );
}

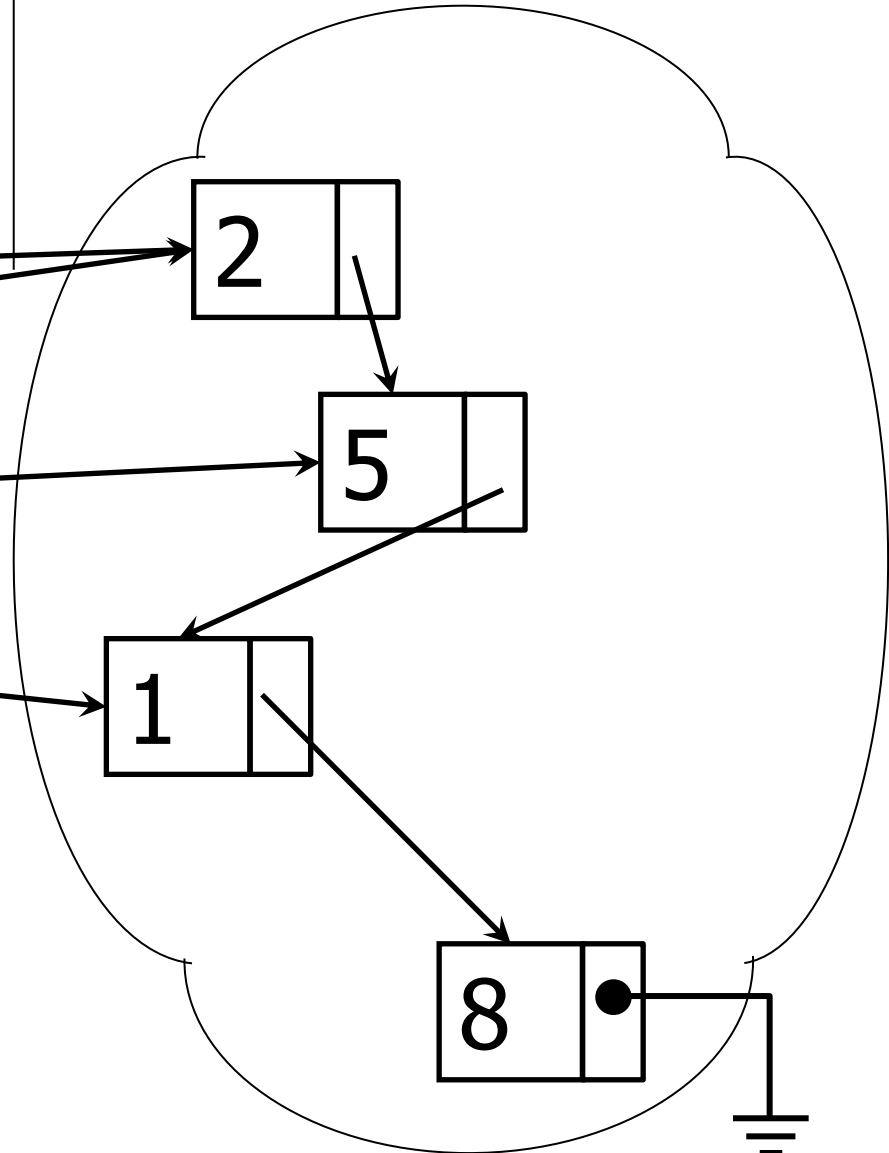
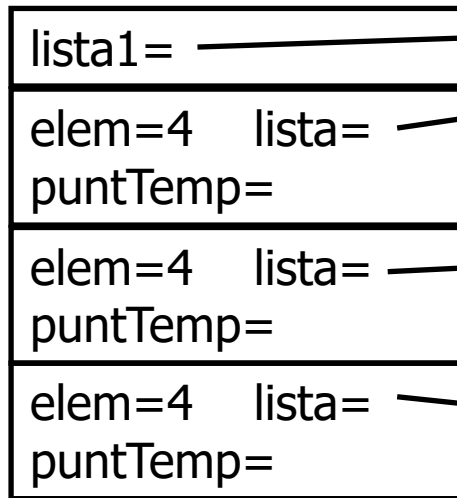
```

```

int main() {
    ListaDiElem lista1;
    ... ..
    lista1 = Cancellazione( lista1, 4 );
    ... ..
}

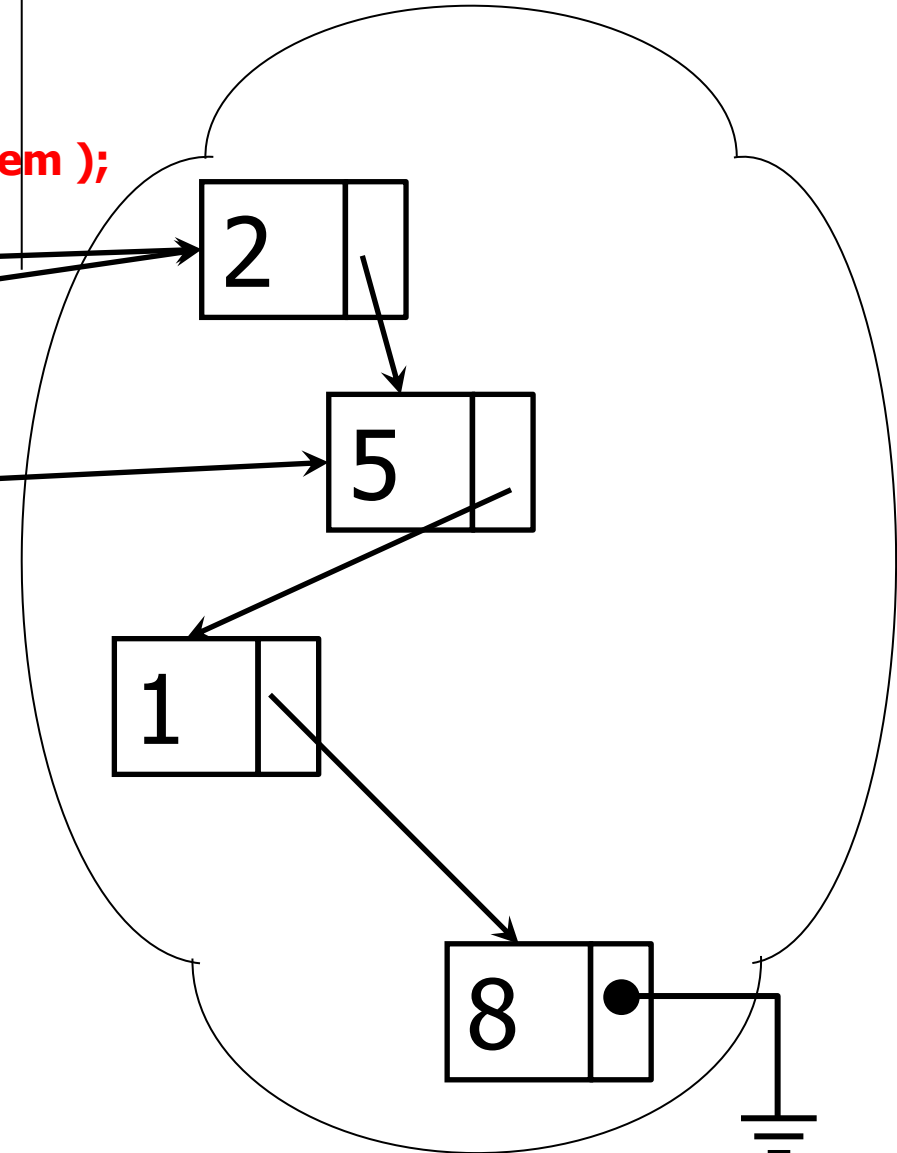
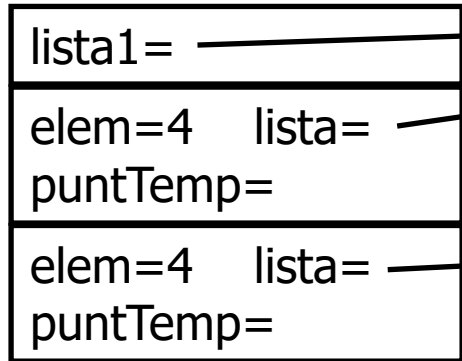
```

return lista;



```
ListaDiElem Cancellala( ListaDiElem lista, int elem ) {  
    ListaDiElem puntTemp;  
    if( lista!=NULL)  
        if( lista->info == elem ) {  
            puntTemp = lista->prox;  
            free( lista );  
            return puntTemp;  
        } else  
            lista->prox = Cancellala( lista->prox, elem );  
    return lista;  
}
```

```
int main() {  
    ListaDiElem lista1;  
    ... ..  
    lista1 = Cancellala( lista1, 4 );  
    ... ..  
}
```



```

ListaDiElem Cancellazione( ListaDiElem lista, int elem ) {
    ListaDiElem puntTemp;
    if( lista!=NULL)
        if( lista->info == elem ) {
            puntTemp = lista->prox;
            free( lista );
            return puntTemp;
        } else
            lista->prox = Cancellazione( lista->prox, elem );
}

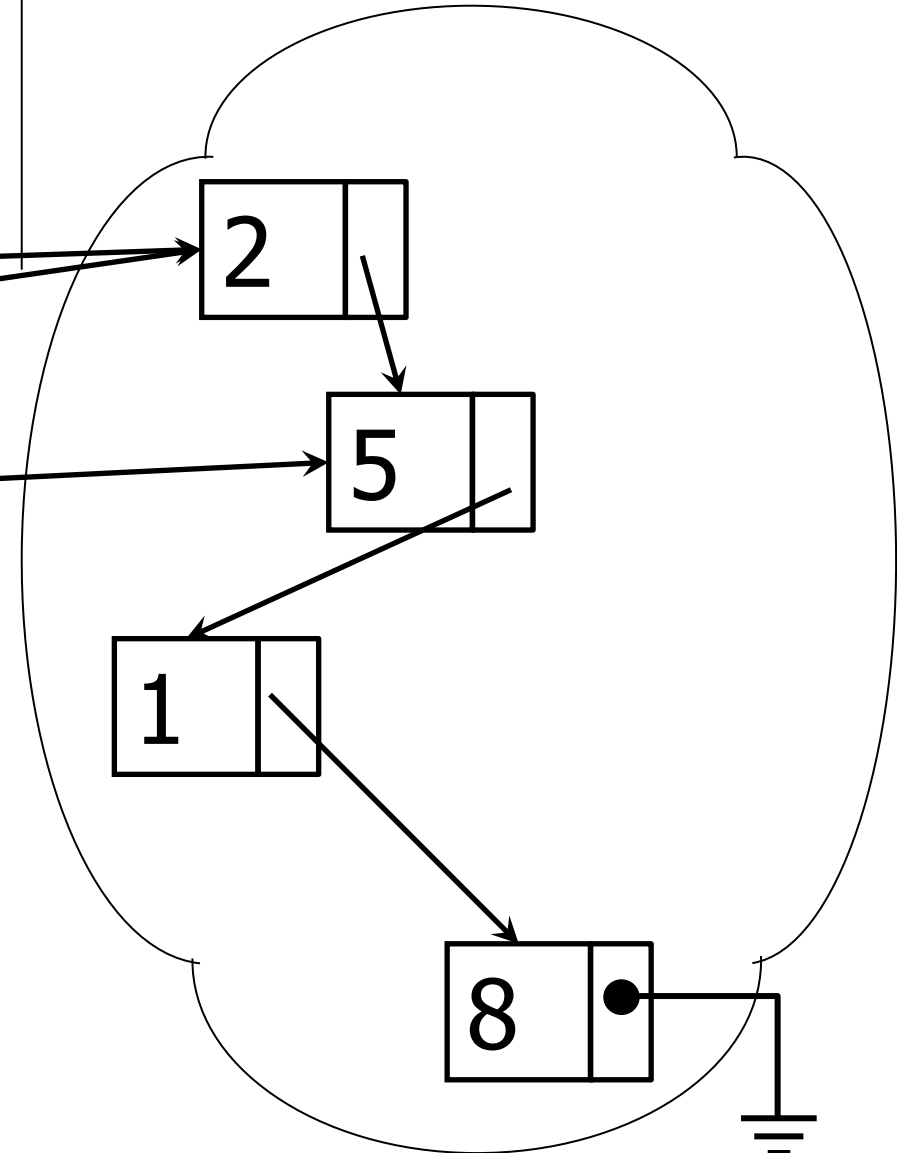
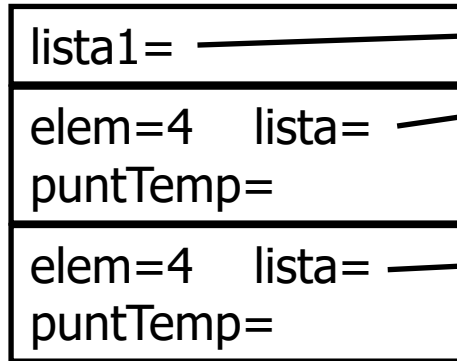
```

```

int main() {
    ListaDiElem lista1;
    ... ..
    lista1 = Cancellazione( lista1, 4 );
    ... ..
}

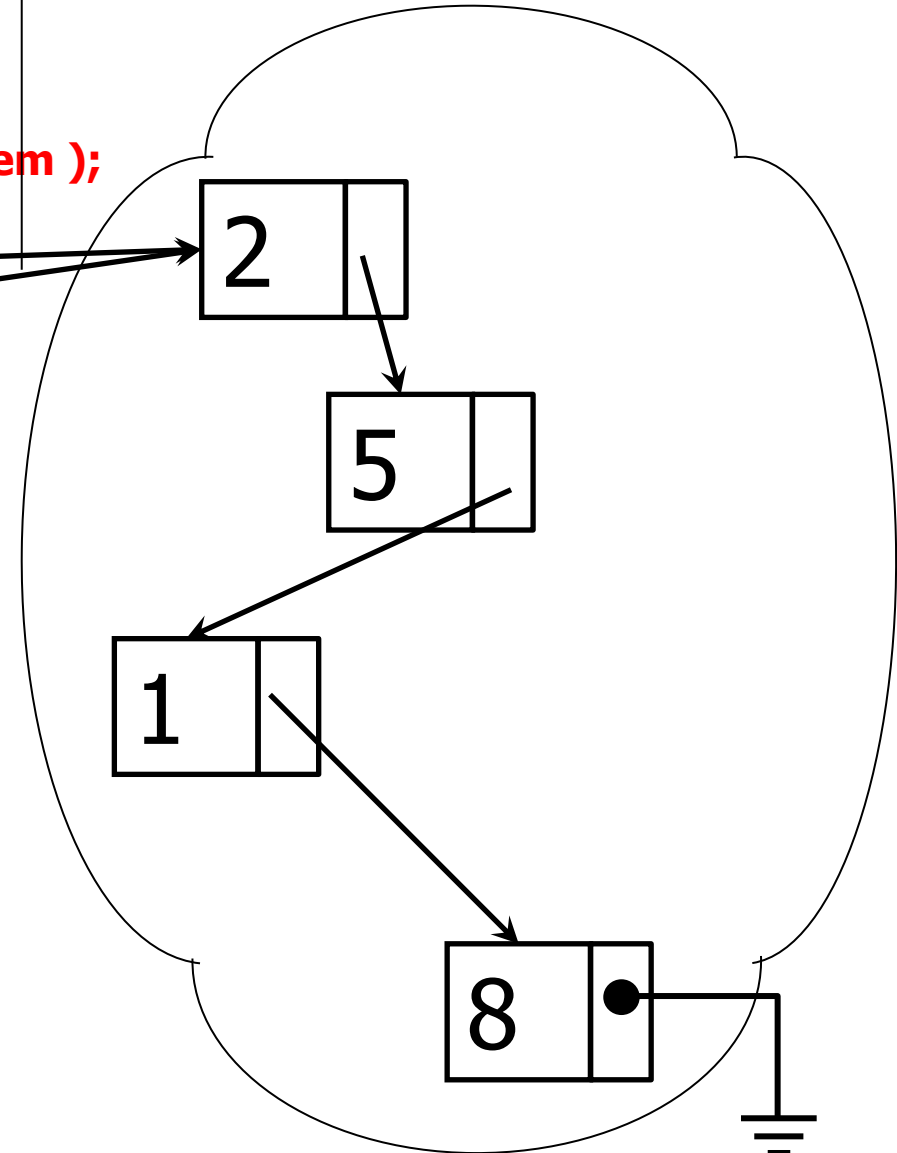
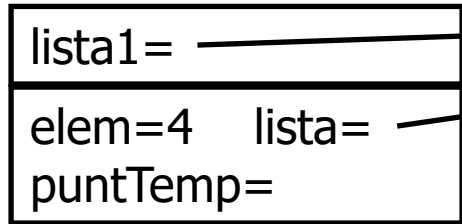
```

return lista;



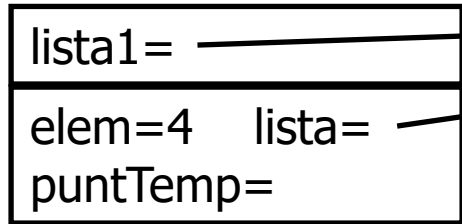
```
ListaDiElem Cancellazione( ListaDiElem lista, int elem ) {  
    ListaDiElem puntTemp;  
    if( lista!=NULL)  
        if( lista->info == elem ) {  
            puntTemp = lista->prox;  
            free( lista );  
            return puntTemp;  
        } else  
            lista->prox = Cancellazione( lista->prox, elem );  
    return lista;  
}
```

```
int main() {  
    ListaDiElem lista1;  
    ... ..  
    lista1 = Cancellazione( lista1, 4 );  
    ... ..  
}
```

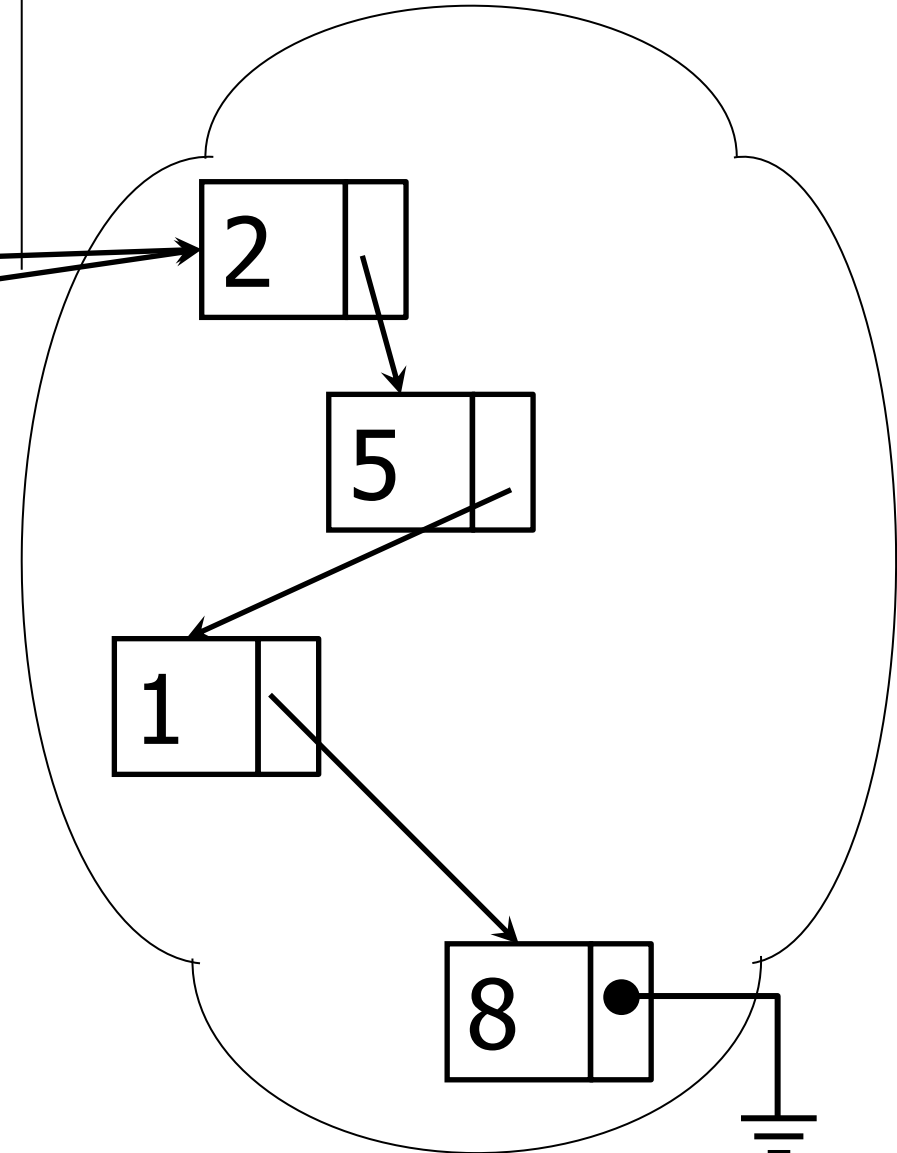


```
ListaDiElem Cancella( ListaDiElem lista, int elem ) {
    ListaDiElem puntTemp;
    if( lista!=NULL)
        if( lista->info == elem ) {
            puntTemp = lista->prox;
            free( lista );
            return puntTemp;
        } else
            lista->prox = Cancella( lista->prox, elem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

return lista;



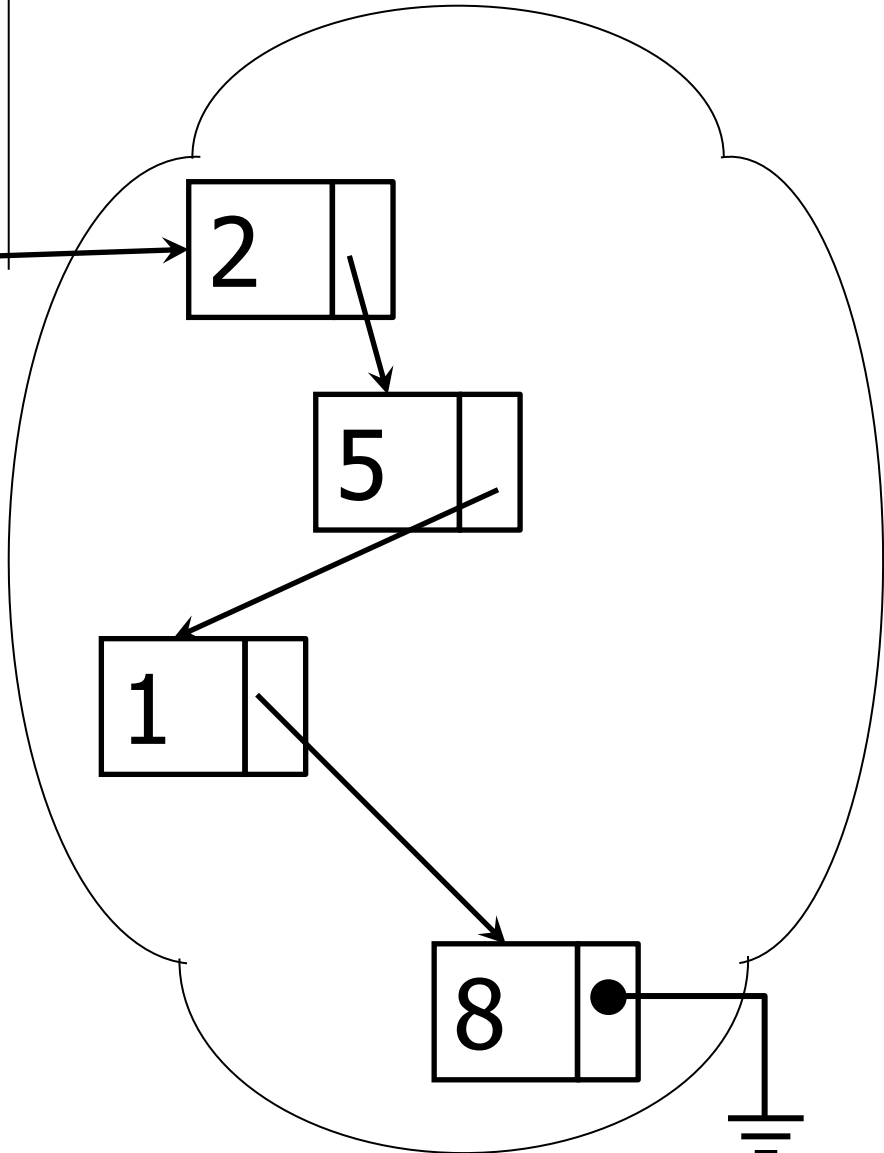
```
int main() {
    ListaDiElem lista1;
    ... ..
    lista1 = Cancella( lista1, 4 );
    ... ..
}
```




```
ListaDiElem Cancellala( ListaDiElem lista, int elem ) {
  ListaDiElem puntTemp;
  if( lista!=NULL)
    if( lista->info == elem ) {
      puntTemp = lista->prox;
      free( lista );
      return puntTemp;
    } else
      lista->prox = Cancellala( lista->prox, elem );
  return lista;
}
```

```
int main() {
  ListaDiElem lista1;
  ... ..
  lista1 = Cancellala( lista1, 4 );
  ... ..
}
```

lista1 =



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 ( lista!=NULL ) {  
        DistruggiListaRic( lista->prox );  
        free( lista );  
    }  
}
```

Visualizzare la lista

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

1 → 2 → 3
1 --> 2 --> 3 --> --|

Visualizzare la lista “al contrario”

```
void VisualizzaListaRovesciata( ListaDiElem lista ) {  
    if ( lista==NULL )  
        printf(" |--- \n");  
    else {  
        VisualizzaListaRovesciata( lista->prox );  
        printf(" %d\n <-- ", lista->info);  
    }  
}
```

1→2→3
|-- 3 <-- 2 <-- 1

*Ma è solo la visualizzazione ad essere invertita
La lista resta inalterata*

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 = NULL;
    scanf("%d", &n);
    while( n != SENTINELLA ) {
        lista = InsInTesta( lista, n );
        scanf("%d", &n);
    }
    VisualizzaLista(lista);
    return 0;
}
```

```
#define SENTINELLA -1
typedef int TipoElemento;
int main() {
    int n;
    ListaDiElem lista = NULL;
    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"*


```
#define SENTINELLA -1
typedef int TipoElemento;
int main() {
    int n;
    ListaDiElem lista = NULL;
    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( curr!=NULL )  
        temp = InsInTesta( temp, curr->info );  
        curr = curr->prox;  
    }  
    if( ! keepSource )  
        DistruggiLista( lista );  
    return temp;  
}
```

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

Reverse di lista

```
ListaDiElem Reverse1( ListaDiElem lista, int keepSource ) {  
    ListaDiElem temp = Inizializza(), curr = lista;  
    while( curr!=NULL )  
        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( lista!=NULL ) {  
        while( lista->prox != NULL ) {  
            temp = prec;  
            prec = lista;  
            lista = lista->prox;  
            prec->prox = temp;  
        }  
        lista->prox = prec;  
    }  
    return lista;  
}
```

Reverse di lista

```
ListaDiElem Reverse2( ListaDiElem lista ) {  
    ListaDiElem temp, prec = NULL;  
    if( lista!=NULL ) {  
        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 ( lista==NULL || lista->prox==NULL )
        return lista;
    else {
        p = lista->prox;
        ris = ReverseRic( p );
        p->prox = lista;
        lista->prox = NULL;
        return ris;
    }
}

```

Prima della chiamata ricorsiva possiamo mettere da parte un puntatore [**p**] al **secondo** elemento [**p=lista->prox**], confidando che dopo l'inversione esso sarà diventato l'**ultimo** elemento della "coda invertita", e attaccargli [**p->prox=**] (in coda) il primo elemento [**lista**]

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)

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);
```

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);
    }
}
```

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?