Esercitazioni di Prog. II (funzioni su Liste e Pile)

Chiara Petrioli

Esercizi su liste

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

struct elemento_lista{
int elem;
struct elemento_lista *next;
};

typedef struct elemento_lista      L_ELEM;
typedef L_ELEM     *L_PTR;

L_PTR inserisci_in_testa(L_PTR,int);
void inserisci_in_testa2 (L_PTR *,int);
void stampalista(L_PTR);
void stampalista_iter(L_PTR);
int sum_elementi(L_PTR);
int num_occorrenze(L_PTR,int);
int verifica_presenza(L_PTR,int);
L_PTR inserisci_in_coda (L_PTR, int);
int verifica_se_ordinata (L_PTR, int);
L_PTR Rinvertilista (L_PTR);
```

Main

```
main()
{
int i,n;
L_PTR L=NULL;
L_PTR L2=NULL;
int dato;

printf("inserisci numero di elementi della prima lista \n");
scanf("%d",&n);
for (i=0;i<n;i++)
{
printf("inserisci l' %d -esimo elemento della prima lista (inserimento in coda)\n", i+1);
scanf("%d",&dato);
L=inserisci_in_coda(L,dato);
}
printf("la prima lista inserita da input e' \n");
stampalista(L);
if (verifica_se_ordinata (L,0))
printf ("la prima lista e' ordinata \n");
else
printf ("la prima lista non e' ordinata \n");
printf("la prima lista invertita e' \n");
stampalista(Rinvertilista(L));
```

Main

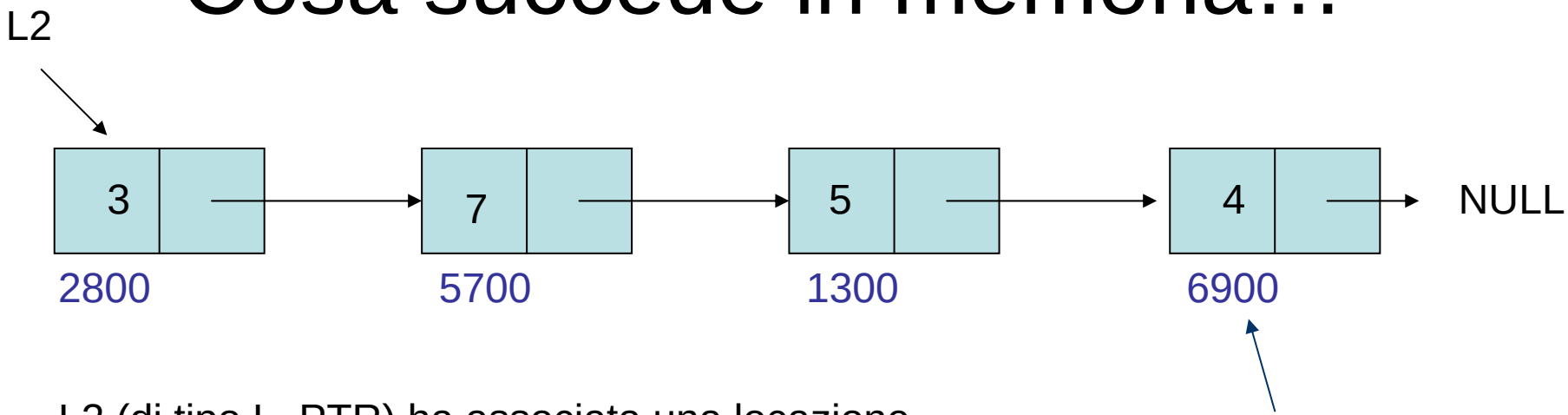
```
printf("inserisci numero di elementi della seconda  lista \n");
scanf("%d",&n);
for (i=0;i<n;i++)
{
printf("inserisci l' %d -esimo elemento della seconda lista (inserimento in testa)\n", i+1);
scanf("%d",&dato);
inserisci_in_testa2(&L2,dato);
}
printf("la seconda lista inserita da input e' \n");
stampalista_iter(L2);
if (verifica_se_ordinata (L2,0))
printf ("la seconda lista e' ordinata \n");
else
printf ("la seconda lista non e' ordinata \n");
printf("la sommtoria degli elementi della seconda lista e' %d \n", sum_elementi(L2));
printf("inserisci elemento da cercare nella seconda lista \n");
scanf("%d", &dato);
if (verifica_presenza(L2,dato))
printf ("il valore inserito da input compare nella seconda lista \n");
else
printf ("il valore inserito da input non compare nella seconda lista \n");
printf("il numero di occorrenze dell'elemento nella lista e' quindi %d \n", num_occorrenze(L2, dato));
}
```

Esercizio 1

```
/*inserisci un nuovo elemento in testa alla lista*/
L_PTR inserisci_in_testa(L_PTR L1, int val)
{
L_PTR temp_ptr;
temp_ptr=malloc(sizeof(L_ELEM));
if (temp_ptr!=NULL)
  {
   temp_ptr->elem=val;
   temp_ptr->next=L1;
   L1=temp_ptr;
  }
else
  printf("memoria non disponibile per l'elemento della lista \n");
return L1;
}
```

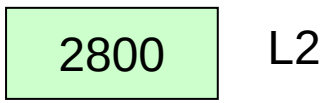
Si scriva una funzione che, data una lista di Interi ed un valore inserisca un nuovo Elemento contenente Il valore in testa alla lista

Cosa succede in memoria…



L2 (di tipo L_PTR) ha associata una locazione di memoria che contiene l'indirizzo del primo elemento della lista

Indirizzo di memoria
In cui e' memorizzato
Questo elemento

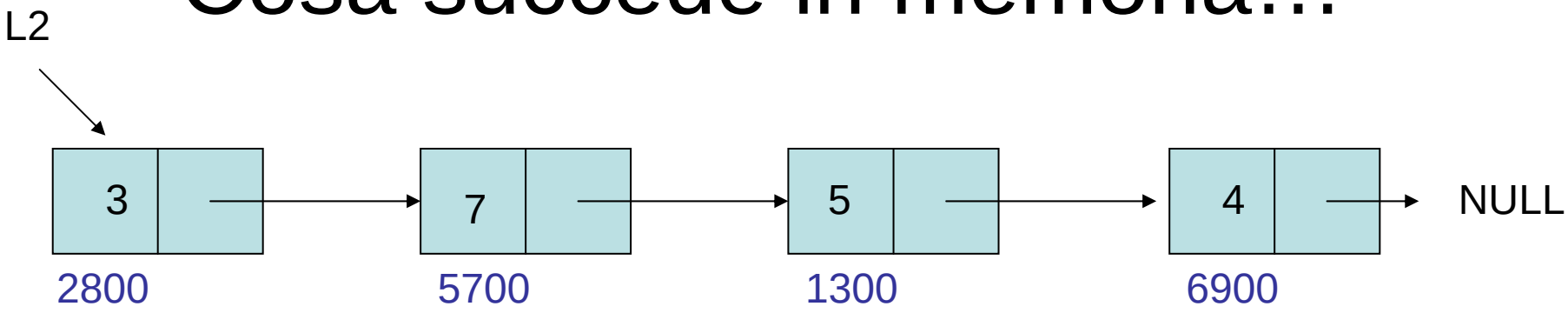


L2->next contiene l'indirizzo di memoria del prossimo elemento della lista (5700)

Vediamo ora cosa succede se dal main viene invocato:
L2=inserisci_in_testa(L2,v) dove v vale 5

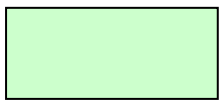
```
L_PTR inserisci_in_testa(L_PTR L1, int val)
{……….
}
```

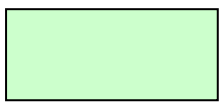
Cosa succede in memoria…



Vediamo ora cosa succede se dal main viene invocato:
L2=inserisci_in_testa(L2,v) dove v vale 5….
Quando viene invocata la funzione viene allocata memoria per gli argomento della funzione (L1 e val)

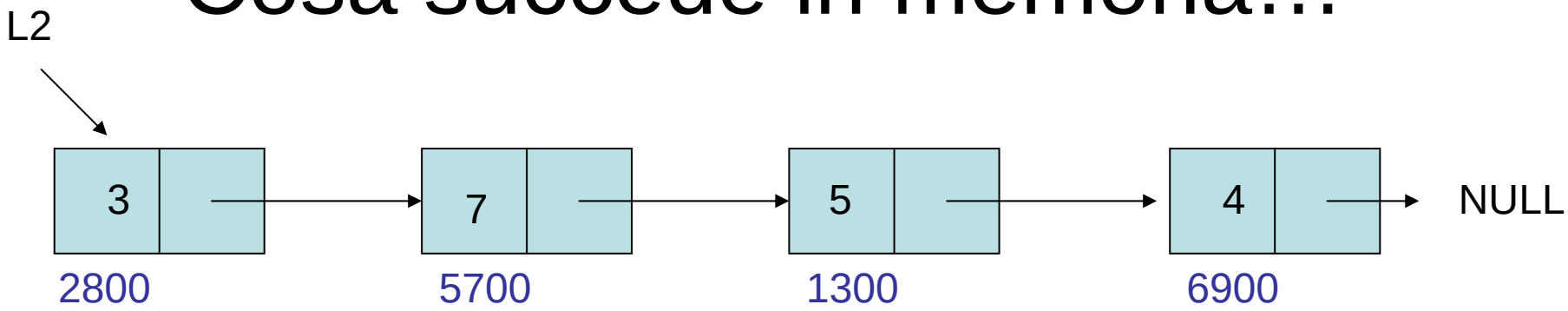
Indirizzo di memoria
In cui e' memorizzato
Questo elemento


L1


val

L_PTR inserisci_in_testa(L_PTR L1, int val)
{……
}

Cosa succede in memoria…

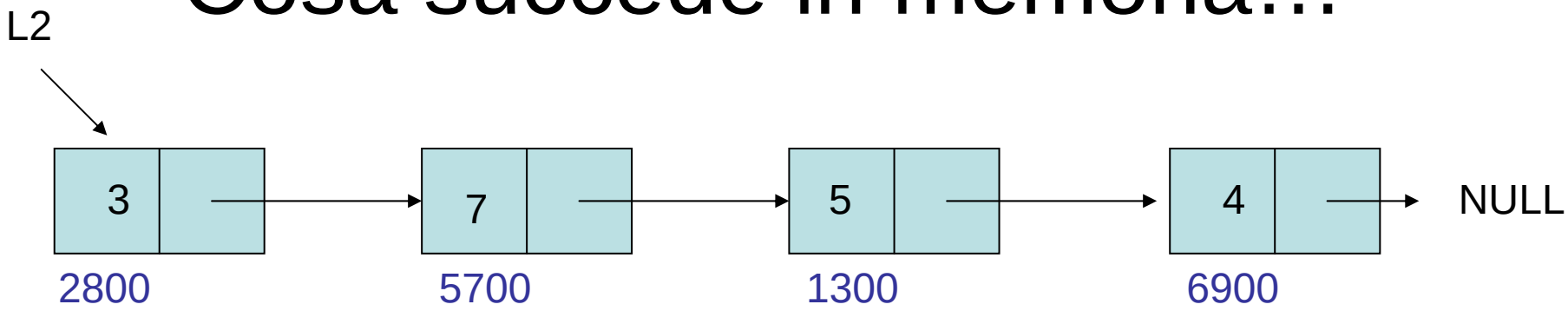


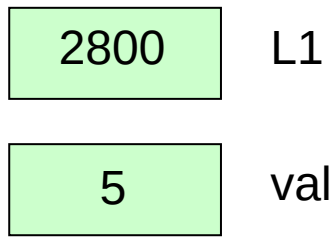
Vediamo ora cosa succede se dal main viene invocato:
L2=inserisci_in_testa(L2,v) dove v vale 5….
Quando viene invocata la funzione viene allocata memoria per gli argomento della funzione (L1 e val).

2800	L1	In L1 viene copiato il valore di L2
5	val	In val viene copiato il valore di v

L_PTR inserisci_in_testa(L_PTR L1, int val)
{……
}

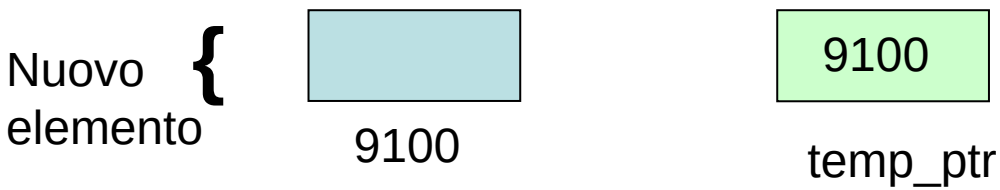
Cosa succede in memoria…





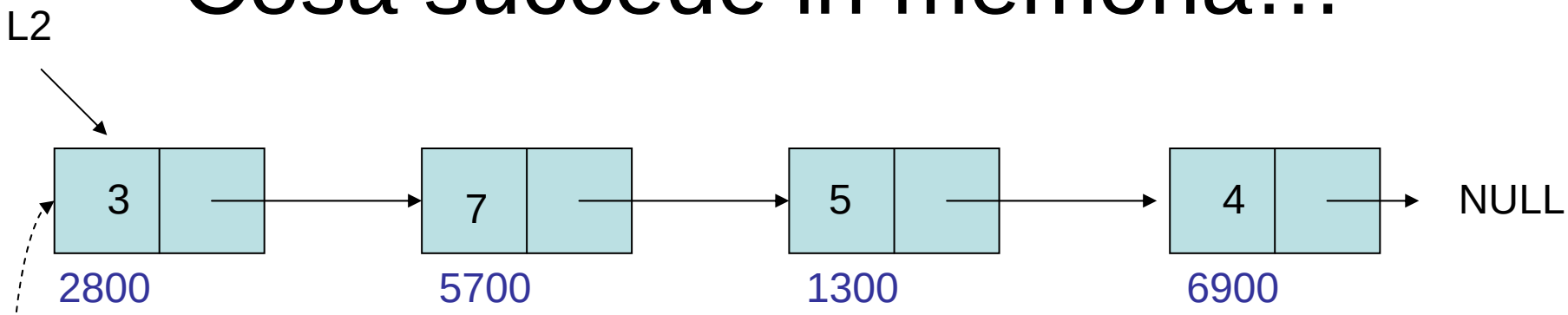

```
L_PTR inserisci_in_testa(L_PTR L1, int val)
{L_PTR temp_ptr;
temp_ptr=malloc(sizeof(L_ELEM));
temp_ptr->elem=val;
   temp_ptr->next=L1;
   L1=temp_ptr;
return L1;
}
```

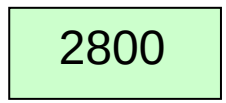
Alloca memoria
per il nuovo elemento



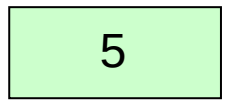
Temp_ptr contiene l'indirizzo della posizione di memoria allocata per
Il nuovo elemento

Cosa succede in memoria…





L1

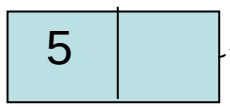


val

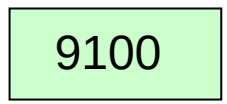
```
L_PTR inserisci_in_testa(L_PTR L1, int val)
{L_PTR temp_ptr;
temp_ptr=malloc(sizeof(L_ELEM));
temp_ptr->elem=val;
   temp_ptr->next=L1;
   L1=temp_ptr;
return L1;
}
```

Viene inserito il valore val nel nuovo elemento
Il campo next del nuovo elemento punta a quello che era il primo elemento della lista

Nuovo elemento {

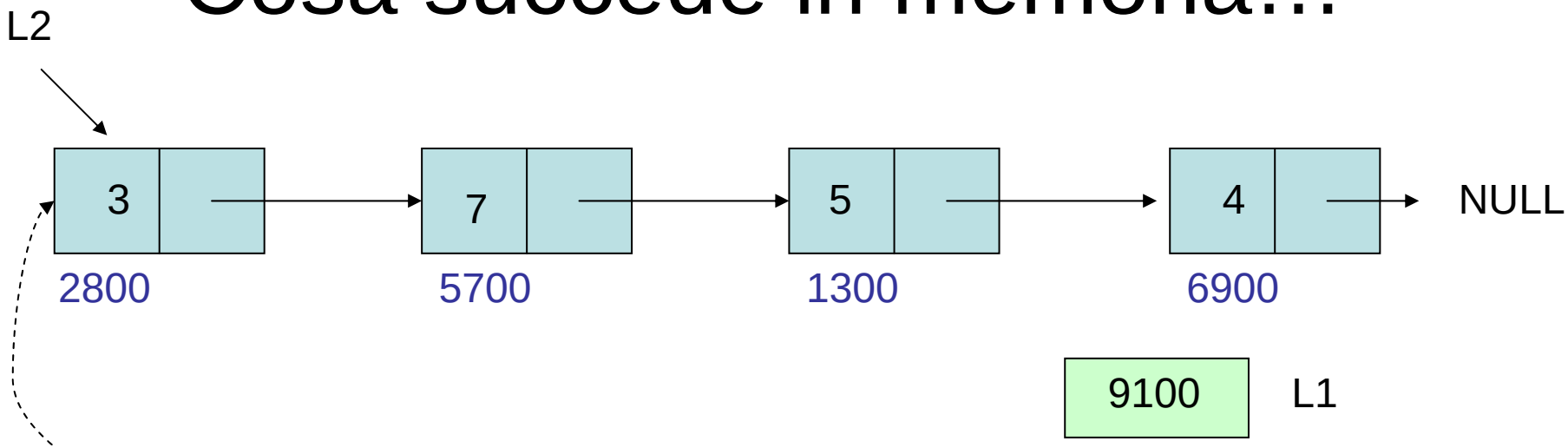


9100



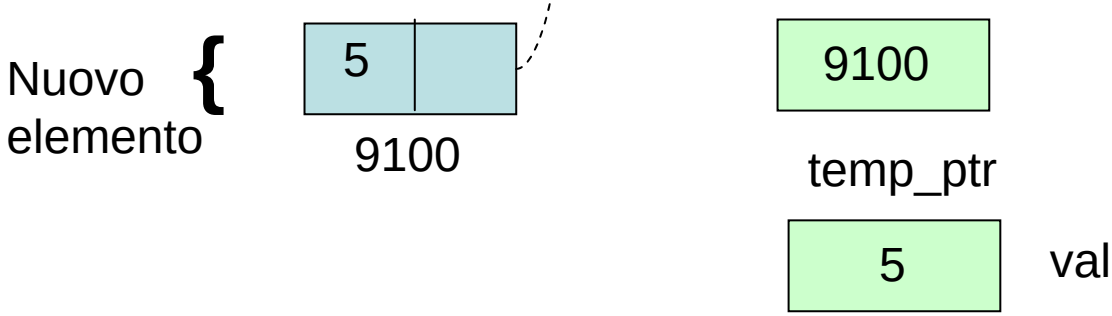
temp_ptr

Cosa succede in memoria…

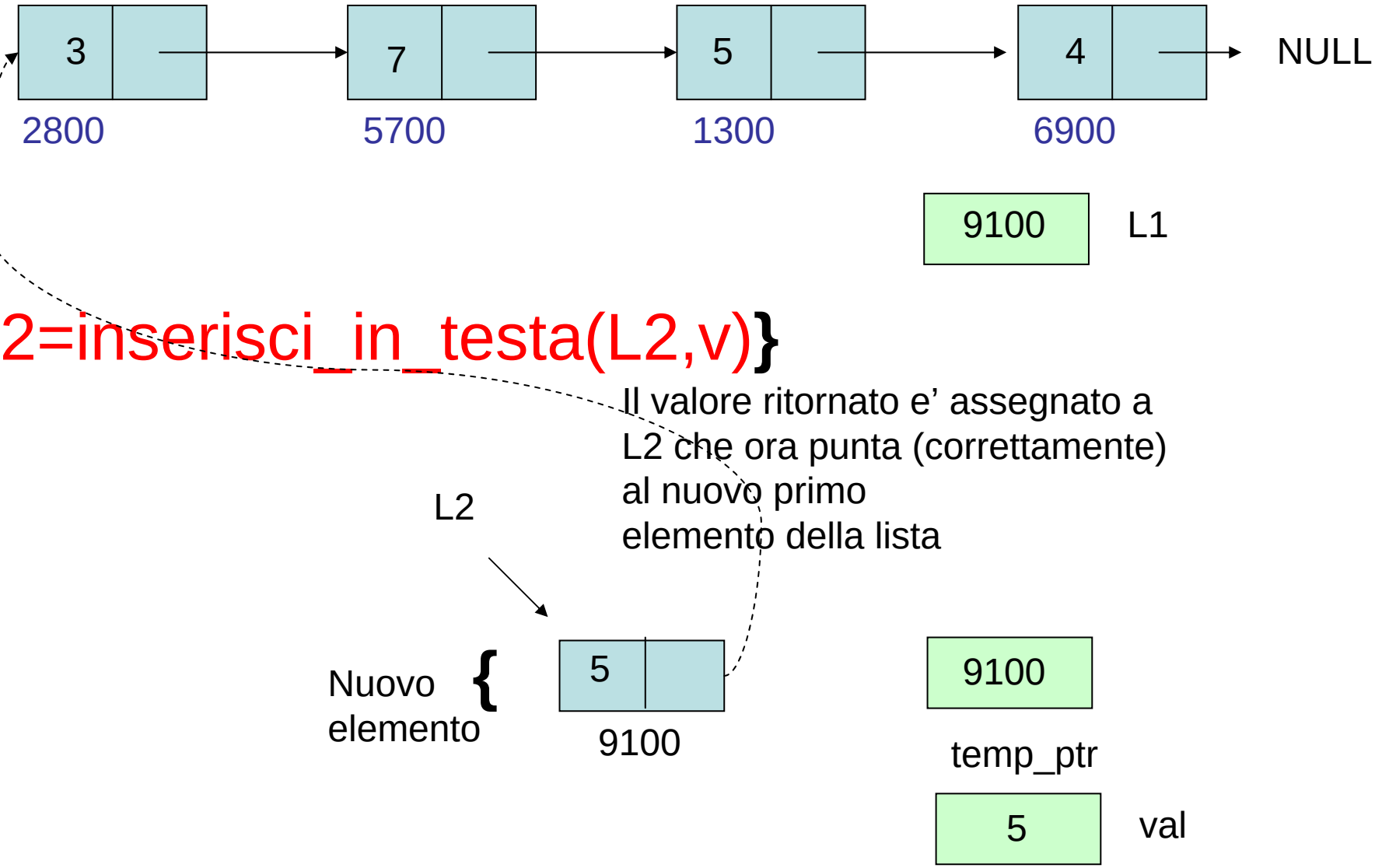



```
L_PTR inserisci_in_testa(L_PTR L1, int val)
{L_PTR temp_ptr;
temp_ptr=malloc(sizeof(L_ELEM));
temp_ptr->elem=val;
   temp_ptr->next=L1;
   L1=temp_ptr;
return L1;
}
```

L1 viene aggiornato con Quello che e' l'indirizzo Della locazione di memoria Del nuovo primo elemento della Lista. Tale valore viene restituito



Cosa succede in memoria…



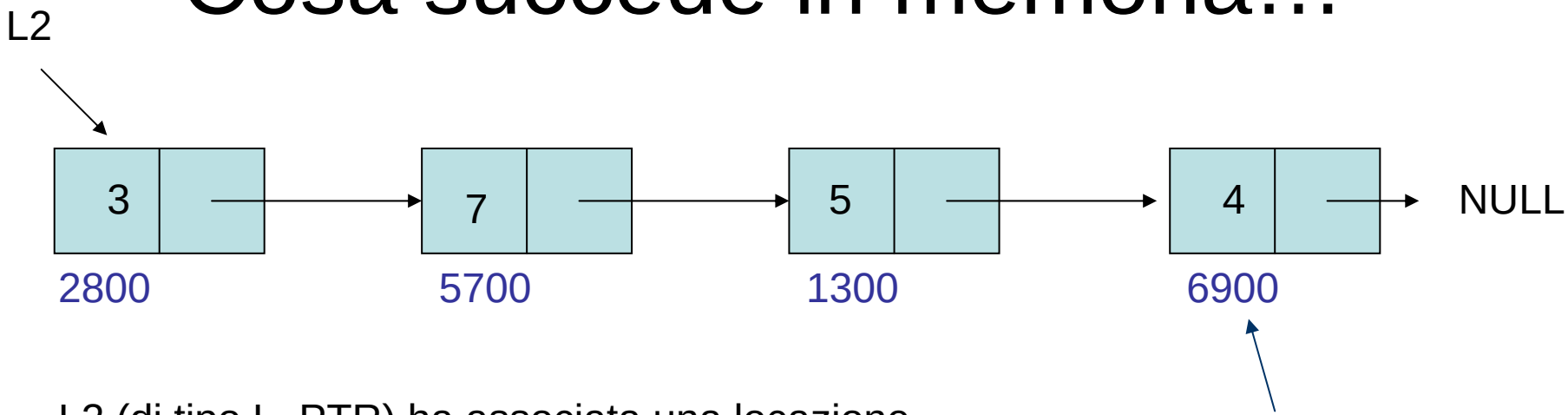
Esercizio 1-bis

```
/*inserisci un nuovo elemento in testa alla lista*/
void inserisci_in_testa(L_PTR L1, int val)
{
L_PTR temp_ptr;
temp_ptr=malloc(sizeof(L_ELEM));
if (temp_ptr!=NULL)
  {
   temp_ptr->elem=val;
   temp_ptr->next=L1;
   L1=temp_ptr;
  }
else
  printf("memoria non disponibile per l'elemento della lista \n");
}
```

Si scriva una funzione che, data una lista di Interi ed un valore inserisca un nuovo Elemento contenente Il valore in testa alla lista

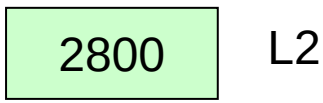
SBAGLIATO!!!
Capiamo il perche’

Cosa succede in memoria…



L2 (di tipo L_PTR) ha associata una locazione di memoria che contiene l'indirizzo del primo elemento della lista

Indirizzo di memoria
In cui e' memorizzato
Questo elemento

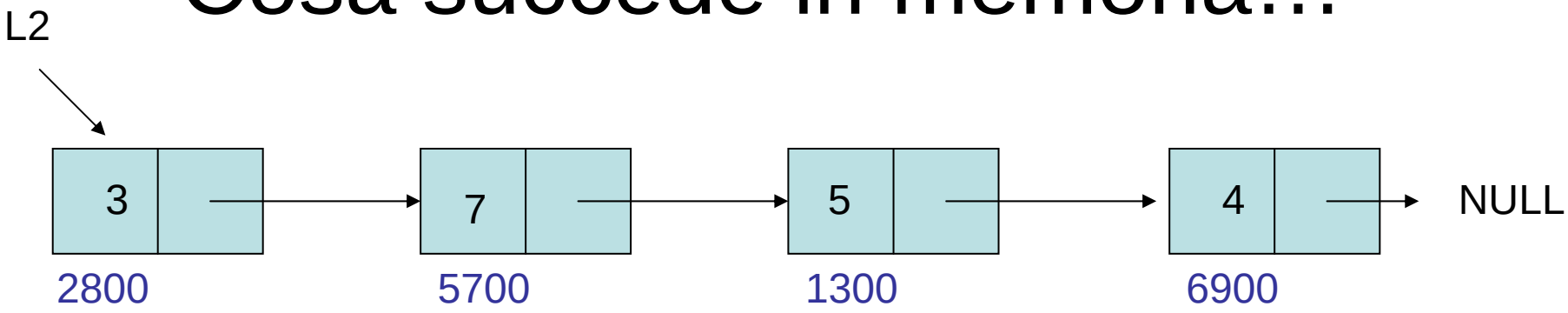


L2->next contiene l'indirizzo di memoria del prossimo elemento della lista (5700)

Vediamo ora cosa succede se dal main viene invocato:
inserisci_in_testa(L2,v) dove v vale 5

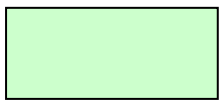
Void inserisci_in_testa(L_PTR L1, int val)
{……
}

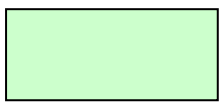
Cosa succede in memoria…



Vediamo ora cosa succede se dal main viene invocato:
inserisci_in_testa(L2,v) dove v vale 5….
Quando viene invocata la funzione viene allocata memoria per gli argomento della funzione (L1 e val)

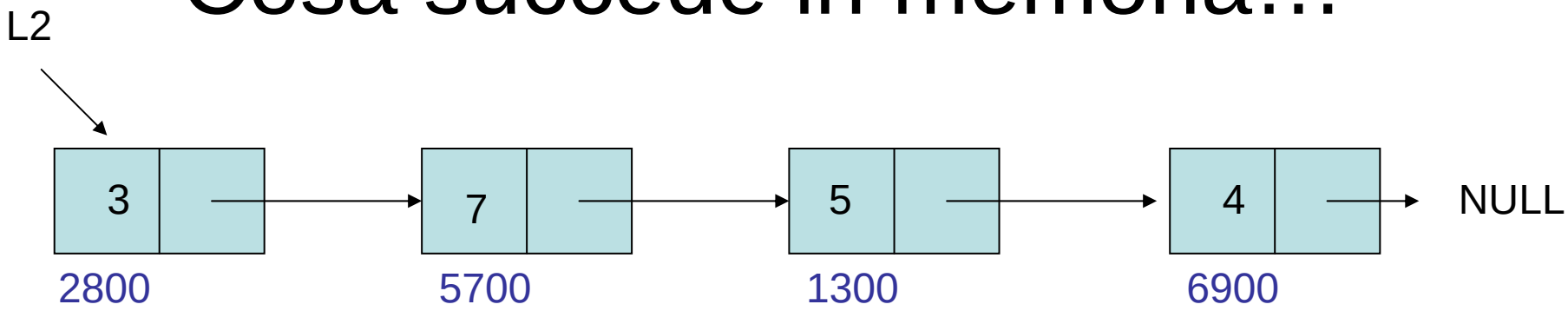
Indirizzo di memoria
In cui e' memorizzato
Questo elemento


L1


val

```
 void inserisci_in_testa(L_PTR L1, int val)
{…….
}
```

Cosa succede in memoria…

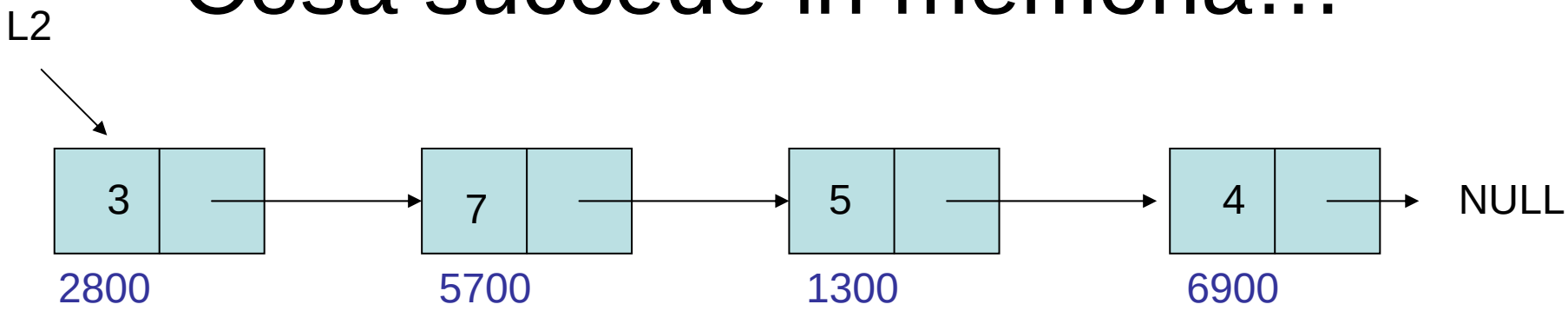


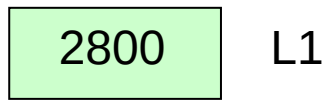
Vediamo ora cosa succede se dal main viene invocato:
inserisci_in_testa(L2,v) dove v vale 5….
Quando viene invocata la funzione viene allocata memoria per gli argomento della funzione (L1 e val).

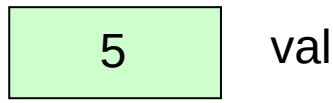
2800	L1	In L1 viene copiato il valore di L2
5	val	In val viene copiato il valore di v

Void inserisci_in_testa(L_PTR L1, int val)
{……
}

Cosa succede in memoria…

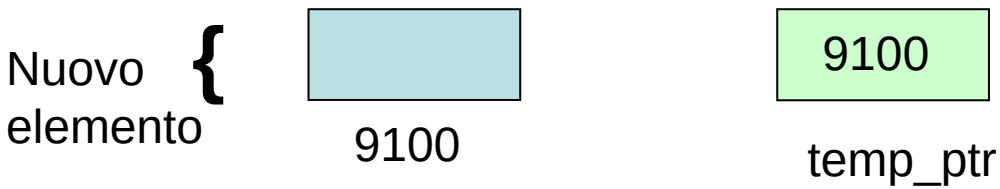






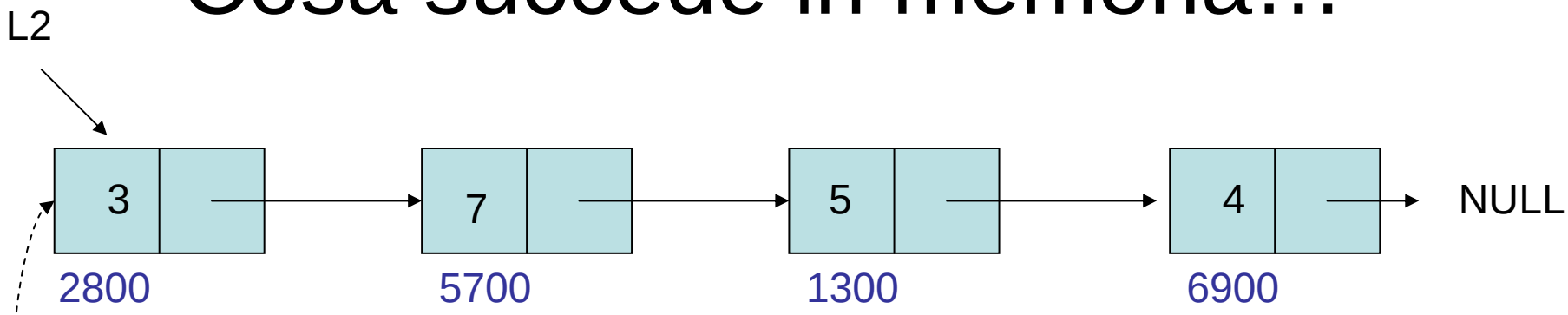

```
void inserisci_in_testa(L_PTR L1, int val)
{L_PTR temp_ptr;
temp_ptr=malloc(sizeof(L_ELEM));
temp_ptr->elem=val;
temp_ptr->next=L1;
L1=temp_ptr;
}
```

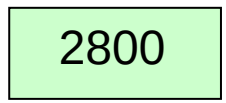
Alloca memoria
per il nuovo elemento



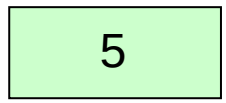
Temp_ptr contiene l'indirizzo della posizione di memoria allocata per
Il nuovo elemento

Cosa succede in memoria…





L1

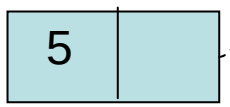


val

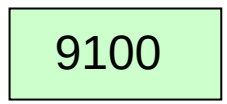
```
L_PTR inserisci_in_testa(L_PTR L1, int val)
{L_PTR temp_ptr;
temp_ptr=malloc(sizeof(L_ELEM));
temp_ptr->elem=val;
temp_ptr->next=L1;
L1=temp_ptr;
}
```

Viene inserito il valore val nel nuovo elemento
Il campo next del nuovo elemento punta a quello che era il primo elemento della lista

Nuovo elemento {

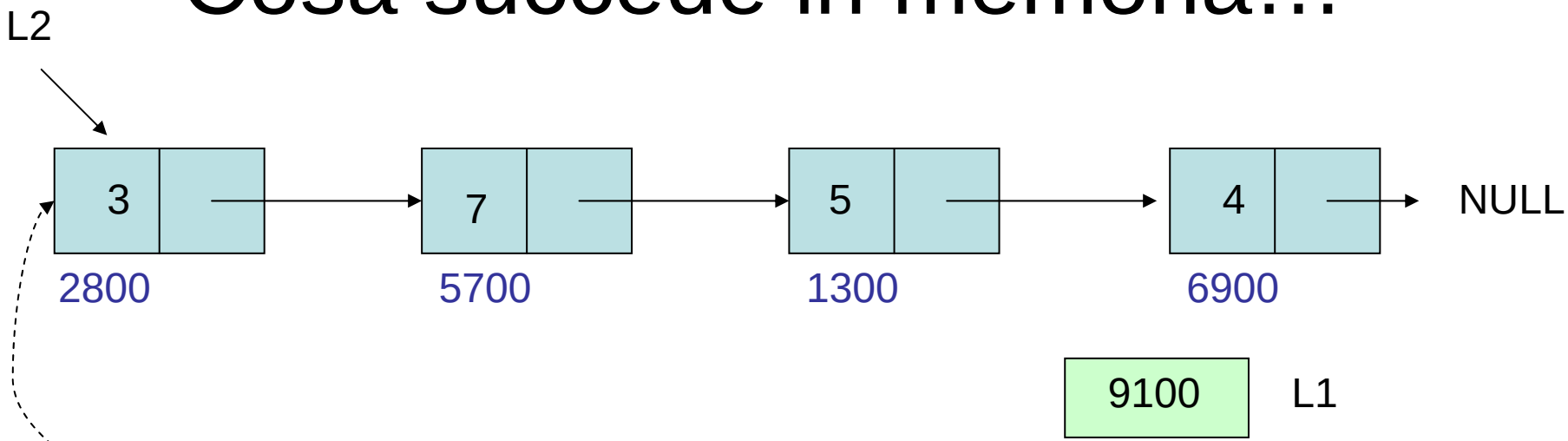


9100



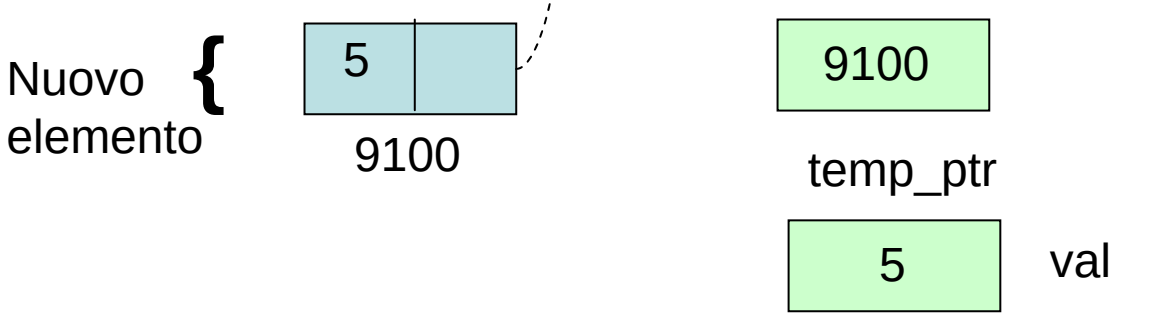
temp_ptr

Cosa succede in memoria…

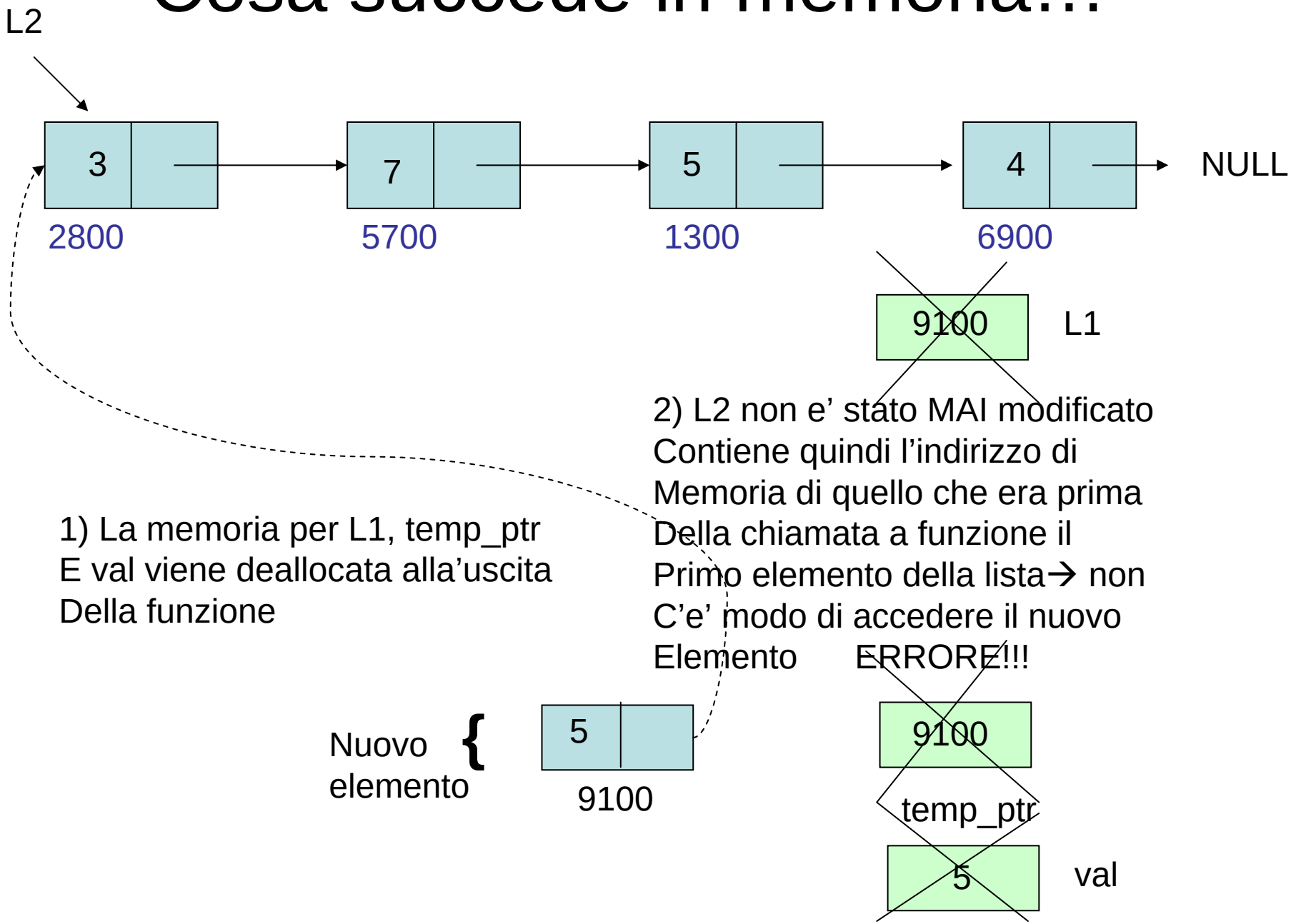



```
Void inserisci_in_testa(L_PTR L1, int val)
{L_PTR temp_ptr;
temp_ptr=malloc(sizeof(L_ELEM));
temp_ptr->elem=val;
   temp_ptr->next=L1;
   L1=temp_ptr;
}
```

L1 viene aggiornato con
Quello che e' l'indirizzo
Della locazione di memoria
Del nuovo primo elemento della
Lista. Si esce dalla funzione..



Cosa succede in memoria…



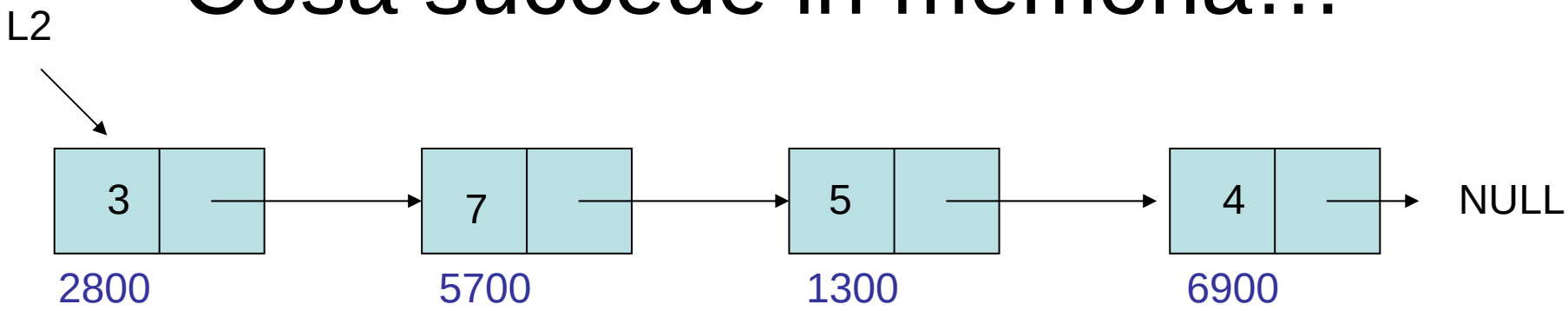
Esercizio 2

```
/*inserisci un nuovo elemento in testa alla lista */
void inserisci_in_testa2 (L_PTR * LISTAPTR, int val)
{
L_PTR temp_ptr;
temp_ptr=malloc(sizeof(L_ELEM));
if (temp_ptr !=NULL)
   {
    temp_ptr->elem=val;
    temp_ptr->next=*LISTAPTR;
    *LISTAPTR=temp_ptr;
   }
else
printf("memoria non sidponibile per l'elemento della lista \n");
}
```

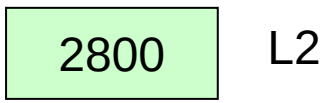
Si scriva una funzione che, data una lista di Interi ed un valore inserisca un nuovo Elemento contenente Il valore in testa alla lista

LISTA_PTR e' un puntatore a puntatore
Ovvero una variabile che contiene l'indirizzo di memoria di una variabile di tipo puntatore (che a sua volta contiene l'indirizzo di memoria di un Elemento della lista)

Cosa succede in memoria…



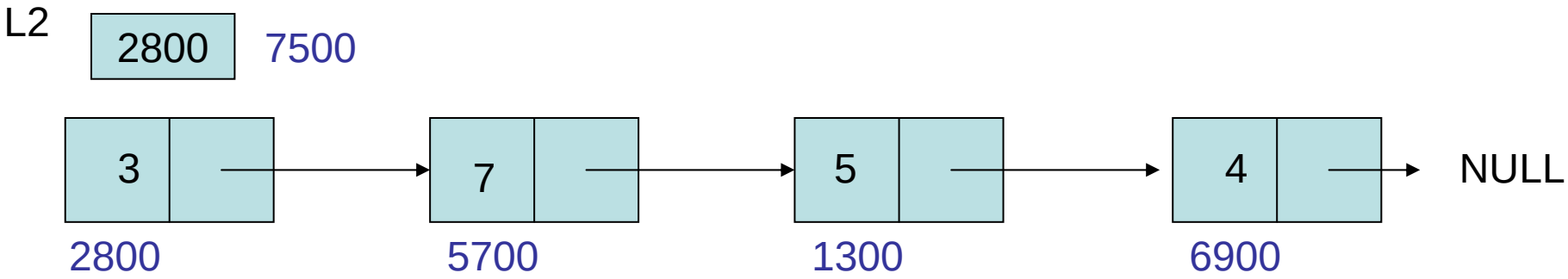
L2 (di tipo L_PTR) ha associata una locazione di memoria che contiene l'indirizzo del primo elemento della lista



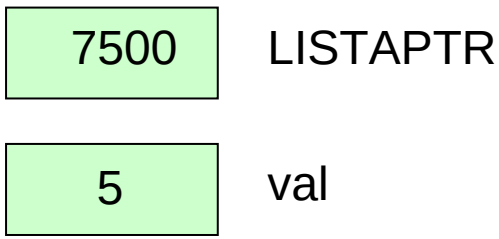
Vediamo ora cosa succede se dal main viene invocato:
inserisci_in_testa(&L2,v) dove v vale 5

```
Void inserisci_in_testa(L_PTR *LISTAPTR, int val)
{.............
}
```

Cosa succede in memoria…

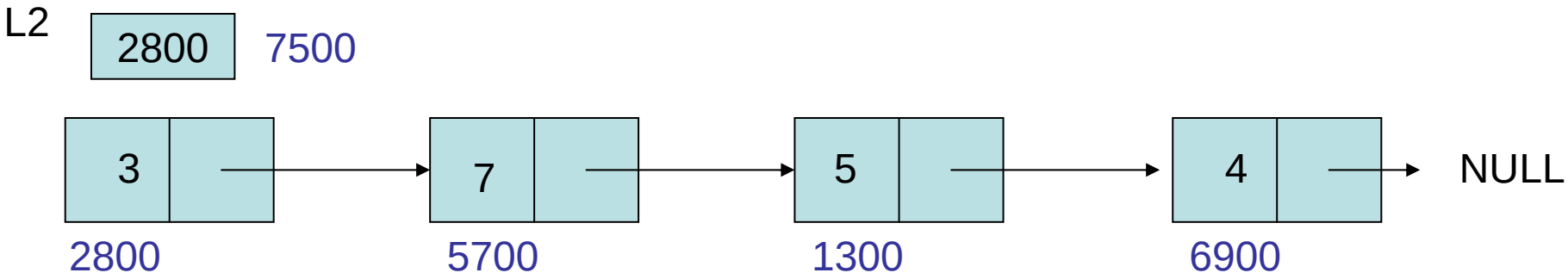


Vediamo ora cosa succede se dal main viene invocato:
inserisci_in_testa(&L2,v) dove v vale 5….
Quando viene invocata la funzione viene allocata memoria
per gli argomento della funzione (LISTAPTR e val). LISTAPTR e' un puntatore a
Puntatore ovvero contiene l'indirizzo di memoria di un puntatore. In LISTAPTR viene
Copiato l'indirizzo di L2 (dato che la funzione e' invocata con argomento &L2)




```
void inserisci_in_testa(L_PTR* LISTAPTR, int val)
{.............
}
```

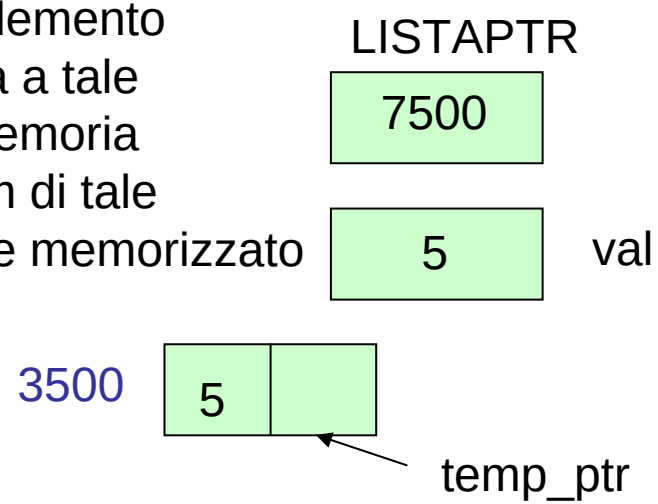
Cosa succede in memoria…



Vediamo ora cosa succede se dal main viene invocato:
inserisci_in_testa(&L2,v) dove v vale 5….

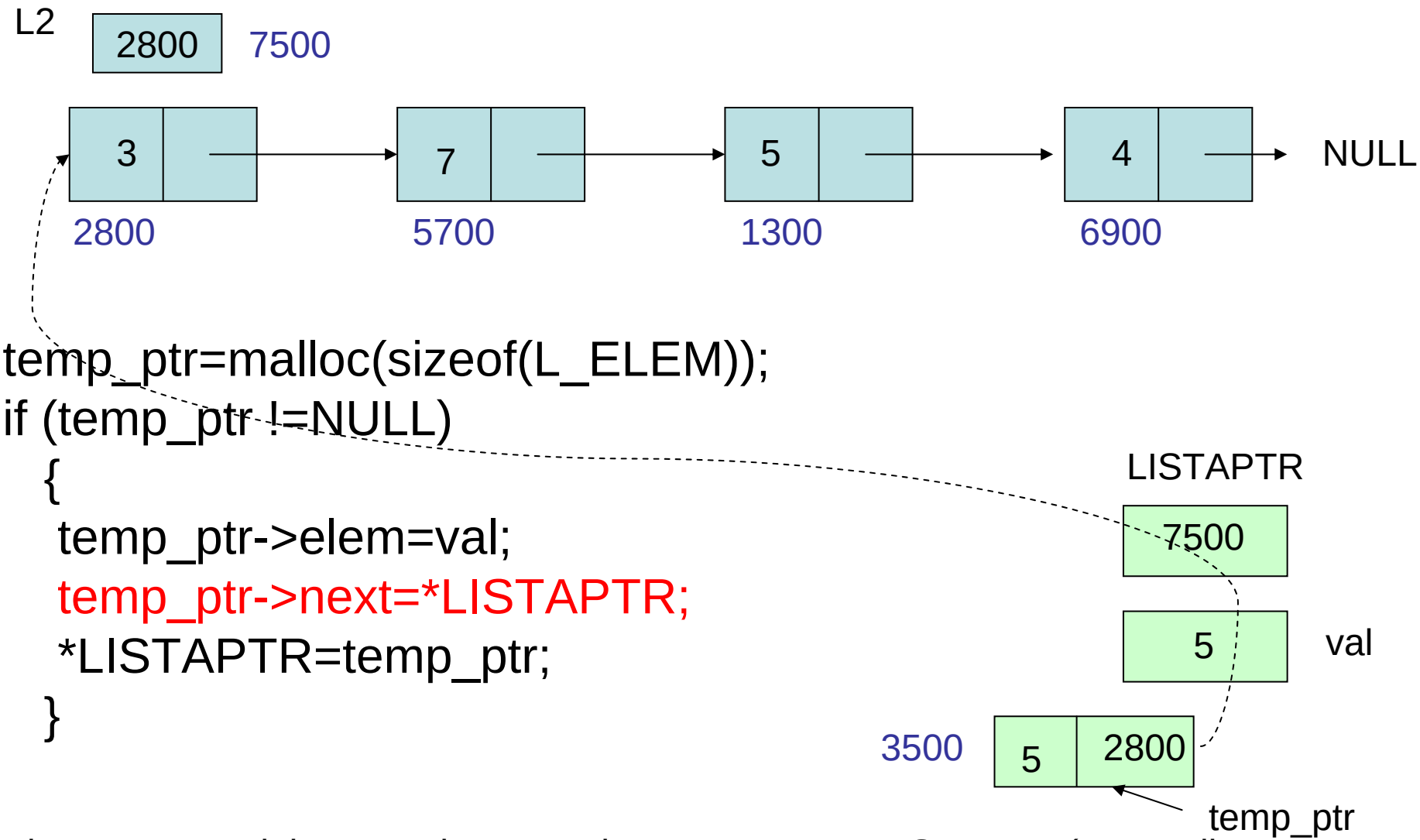
```
temp_ptr=malloc(sizeof(L_ELEM));
if (temp_ptr !=NULL)
  {
   temp_ptr->elem=val;
   temp_ptr->next=*LISTAPTR;
   *LISTAPTR=temp_ptr;
  }
```

Viene allocata memoria
Per un nuovo elemento
Temp_ptr punta a tale
Locazione di memoria
Nel campo elem di tale
Locazione viene memorizzato
Il valore di val



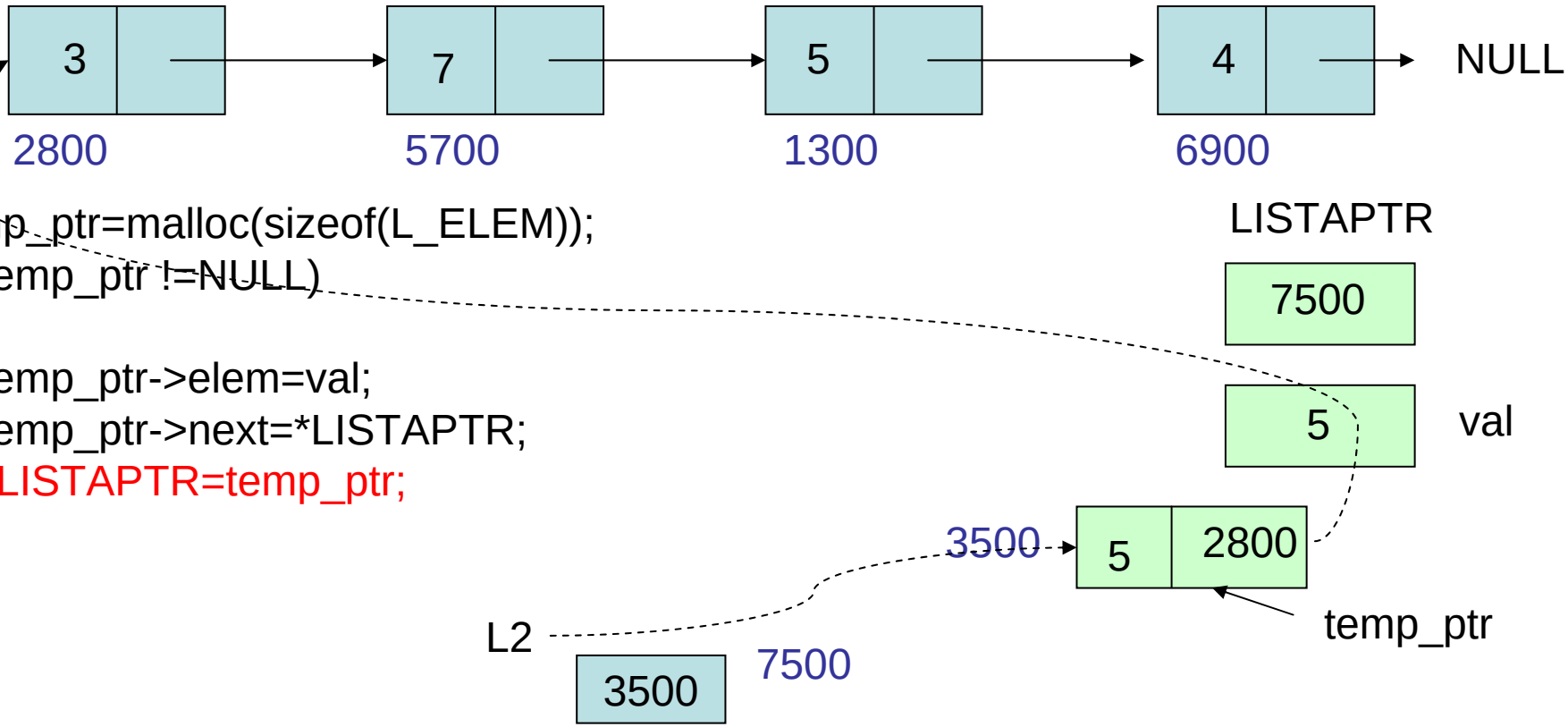
Void inserisci_in_testa(L_PTR* LISTAPTR, int val)
{……
}

Cosa succede in memoria…



Al campo next del nuovo elemento viene assegnato *LISTAPTR (ovvero il contenuto Della locazione di memoria il cui indirizzo e' in LISTAPTR → ovvero l'indirizzo del Primo elemento della lista)

Cosa succede in memoria…



*LISTAPTR e' il contenuto della locazione di memoria il cui indirizzo
E' memorizzato in LISTAPTR, ovvero il valore di L2.
Con l'istruzione *LISTAPTR=temp_ptr viene modificato L2 che viene fatto puntare al nuovo primo elemento. All'uscita dalla funzione L2 puntera' quindi correttamente al nuovo primo elemento della lista.

Esercizio 3 –stampa di una lista

```
/*stampa ricorsiva di una lista*/
void stampalista (L_PTR L1)
{
if (L1!=NULL)
{
printf("--> %d ",L1->elem);
stampalista(L1->next);
}
else
printf("-->NULL \n");
}
```

Si scriva una procedura che data una lista Ne stampi gli elementi ®

```
/*stampa di una lista (versione iterativa)*/
void stampalista_iter (L_PTR L1)
{
while (L1 !=NULL)
{
printf("--> %d", L1->elem);
L1=L1->next;
}
printf("-->NULL \n");
}
```

Si scriva una procedura che data una lista Ne stampi gli elementi (iterativa)

Esercizio 4-somma degli elementi di una lista

```
/*calcola la somma degli elementi di una lista */
int sum_elementi(L_PTR L1)
{
int sum=0;
while (L1 != NULL)
{
sum+=L1->elem;
L1=L1->next;
}
return sum;
}
```

Si scriva una funzione Che dato una lista Calcoli la somma dei Suoi elementi (iterativa)

```
/*versione ricorsiva*/
int sum_elementi(L_PTR L1)
{
if (L1!=NULL)
return ((L1->elem)+(sum_elementi(L1->next)));
else
return 0;
}
```

Si scriva una funzione Che dato una lista Calcoli la somma dei Suoi elementi

®

Esercizio 5

```
/*calcola il numero di occorrenze di un dato valore in una lista*/
int num_occorrenze(L_PTR L1, int val)
{
int occorrenze = 0;
while (L1 !=NULL)
{
  if (L1->elem == val)
  occorrenze++;
  L1=L1->next;
}
return (occorrenze);
}
```

Si scriva una funzione che Data una lista ed un valore Calcoli il numero di Occorrenze del valore Nella lista (iterativa)

Si scriva una funzione che Data una lista ed un valore Calcoli il numero di Occorrenze del valore Nella lista ®

```
/*versione ricorsiva*/
int num_occorrenze (L_PTR L1, int val)
{
if (L1 == NULL)
return 0;
else
return ((L1->elem == val)? (1+num_occorrenze(L1->next,val)):(num_occorrenze(L1->next,val)));
}
```

Esercizio 6

```
int verifica_presenza (L_PTR L1, int val)
{
if (L1 == NULL)
return 0;
else
return ((L1->elem==val)||verifica_presenza(L1->next,val));
}
```

Si scriva una funzione
Che data una lista ed
Un valore verifichi
Se il valore compare tra
Gli elementi della lista

Esercizio 7

```
/*prende un elemento e lo inserisce in coda alla lista*/
L_PTR inserisci_in_coda (L_PTR L1, int val)
{
L_PTR tempptr1, tempptr2;
tempptr2=malloc(sizeof(L_ELEM));
tempptr2->elem = val;
tempptr2->next=NULL;
if (L1 == NULL)
   return tempptr2;
else
{
   tempptr1=L1;
   while (tempptr1->next !=NULL)
      tempptr1=tempptr1->next;
   tempptr1->next=tempptr2;
   return L1;
}
}
```

Si scriva una funzione iterativa
Che inserisca un elemento
In coda alla lista

Esercizio 7-bis

```
void insertTAILlista(LISTA *L1,int val)
{
LISTA temp,temp1;
if (*L1==NULL)
{
temp=malloc(sizeof(NODOLISTA));
temp->elem=val;
temp->next=NULL;
*L1=temp;
}
else
{
temp1=*L1;
while (temp1->next!=NULL)
temp1=temp1->next;
temp=malloc(sizeof(NODOLISTA));
temp->elem=val;
temp->next=NULL;
temp1->next=temp;
}
}
```

Si scriva una funzione iterativa
Che inserisca un elemento
In coda alla lista

```
struct node {
int elem;
struct node *next;
};
typedef struct node NODOLISTA;
typedef NODOLISTA *LISTA;
```

Esercizio 7-bis (versione ricorsiva)

```
void RinsertTAILlista(LISTA *L1,int val)
{
LISTA temp;
if (*L1==NULL)
{
temp=malloc(sizeof(NODOLISTA));
temp->next=NULL;
temp->elem=val;
*L1=temp;
}
else RinsertTAILlista (&((*L1)->next),val);
}
```

Si scriva una funzione
Che inserisca un elemento
In coda alla lista
®

```
struct node {
int elem;
struct node *next;
};
typedef struct node NODOLISTA;
typedef NODOLISTA *LISTA;
```

Esercizio 8

```
/*verifica se la lista e' ordinata- E' ordinata se e' ordinata in ordine
    crescente o decrescente.
Nel primo caso ord varra' 1 altrimenti 2. Alla prima chiamata ord sara'
    inizializzato a 0*/
int verifica_se_ordinata (L_PTR L1, int ord)
{
if ((L1 == NULL) || (L1->next == NULL)) return 1;
switch (ord)
{
case 0: if (L1->elem == L1->next->elem)
        return (verifica_se_ordinata(L1->next,0));
        else if (L1->elem < L1->next->elem)
        return (verifica_se_ordinata (L1->next, 1));
        else return (verifica_se_ordinata(L1->next, 2));
        break;
case 1: return ((L1->elem <=L1->next->elem) && (verifica_se_ordinata (L1-
    >next, 1)));
        break;
case 2:return ((L1->elem >= L1->next->elem) && (verifica_se_ordinata (L1-
    >next, 2)));
        break;
default: break;
}
```

Si scriva una funzione che, data una lista verifichi se e' ordinata

®

Esercizio 9

```
L_PTR Rinvertilista(L_PTR L)
{
L_PTR temp;
if ((L==NULL) ||(L->next == NULL))
return L;
else
{
   temp=Rinvertilista(L->next);
   L->next->next=L;
   L->next=NULL;
   return temp;
}
}
```

Si scriva una funzione che, data una lista La inverta

®

Esercizio 10

```
void RdeleteELEMlista(LISTA * L, int val)
{
LISTA temp;
if (*L==NULL) return;
else if (((*L)->elem)==val)
{
temp=*L;
*L=(*L)->next;
free(temp);
RdeleteELEMlista(L,val);
}
else RdeleteELEMlista (&((*L)->next),val);
}
```

Si scriva una funzione Ricorsiva che, data una lista elimini Le occorrenze di val nella lista

```
struct node {
int elem;
struct node *next;
};
typedef struct node NODOLISTA;
typedef NODOLISTA *LISTA;
```

Esercizio 11

```
LISTA eliminaognik (LISTA l1, int k, int val)
{
LISTA temp;
if (l1==NULL) return l1;
else if (k==0)
{
temp=l1;
l1=l1->next;
free (temp);
return eliminaognik(l1,val,val);
}
else
{
l1->next=eliminaognik(l1->next,k-1,val);
return l1;
}
}
```

Si scriva una funzione Ricorsiva che, data una lista elimini Un elemento ogni val

```
struct node {
int elem;
struct node *next;
};
typedef struct node NODOLISTA;
typedef NODOLISTA *LISTA;
```

Esercizio 12

```
void eliminaognikpuntpunt (LISTA *l1, int k, int val)
{
LISTA temp;
if (*l1==NULL) return;
else if (k==0)
{
temp=*l1;
*l1=(*l1)->next;
free (temp);
eliminaognikpuntpunt(l1,val,val);
return;
}
else
{
eliminaognikpuntpunt(&((*l1)->next),k-1,val);
return;
}
}
```

Si scriva una funzione Ricorsiva che, data una lista elimini Un elemento ogni val

```
struct node {
int elem;
struct node *next;
};
typedef struct node NODOLISTA;
typedef NODOLISTA *LISTA;
```

Esercizio 12

```
void eliminapari(LISTA *l1)
{
LISTA temp;
if (*l1==NULL) return;
else if ((*l1)->elem %2)
{
eliminapari(&(*l1)->next);
return;
}
else
{
temp=*l1;
*l1=(*l1)->next;
free(temp);
eliminapari(l1);
return;
}
}
```

Si scriva una funzione Ricorsiva che, data una lista elimini Gli elementi contenenti Valori pari

Esercizi su pile

```
struct node{
int elem;
struct node *next;
};


typedef struct node NODOPILA;
typedef NODOPILA *PILA;

void push(PILA *, int);
PILA pop(PILA *);
void printpila(PILA);
int verifparentesi();
```

Pop e Push

```
/*Post: restituisce un puntatore alla testa della pila a cui e'
stato acchiunto il nuovo elemento*/
void push (PILA *PTRPILA, int val)
{
	PILA temp;
	temp=malloc(sizeof(NODOPILA));
	if (temp==NULL)
		printf("memoria non disponibile \n");
	else
	{
		temp->elem=val;
		temp->next = *PTRPILA;
		*PTRPILA = temp;
	}
}


/*Post: restituisce un puntatore a quello che era l'elemento in testa della
pila, ora cancellato. Se la pila e' vuota restituisce NULL */
PILA pop (PILA *PTRPILA)
{
PILA temp;
	if ((*PTRPILA)!=NULL)
	{
		temp = *PTRPILA;
		*PTRPILA = (*PTRPILA)->next;
		return temp;
	}
	else
		return NULL;
}
```

Verifica della corretta parentesizzazione

```
/*Post: verifica se una linea di parentesi ( e ) prese da input e' ben parentesizzata*/
int verifparentesi()
{
    PILA L=NULL;
    PILA t;

    int c;
    while ((c=getchar())!='\n')
    {
        switch(c)
        {
            case '(':push(&L,c);
                break;
            case ')':t=pop(&L);
                if (t==NULL) return 0;
                break;
            default:break;
        }
    }
    return (L==NULL);
}
```

Si scriva una funzione Che prende da input una sequenza di Parentesi tonde aperte E chiuse e verifica che Tale sequenza sia ben parentesizzata