Liste

Rappresentazione di sequenze ....

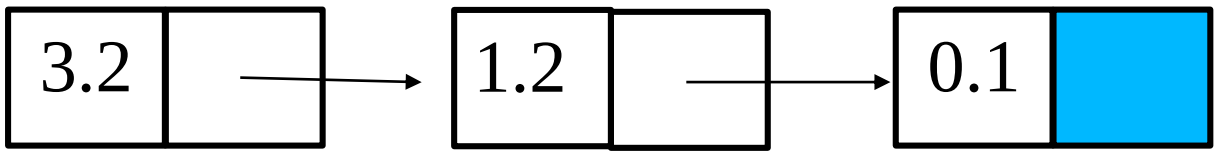
- Ritorniamo al nostro problema di rappresentare le sequenze, se vi ricordate avevano sottolineato un paio di problemi della rappresentazione di sequenze usando array statici
  - Gestire sequenze con numero di elementi non noto a tempo di compilazione
    - Abbiamo visto come questo si può risolvere con l'allocazione al costo di lunghe copie dell'array
    - Si può fare meglio ?
  - Gestire sequenze in cui gli elementi vengono inseriti e cancellati elementi in mezzo alla sequenza
    - Anche per questo possiamo pensare a creare nuove copie ma
    - Anche in questo caso si può fare meglio ?

Un piccolo passo indietro

- Gli array sono quello che in informatica si definisce una *struttura dati* ovvero un modo di organizzare i dati
- In informatica ci sono diverse modi di organizzare i dati, ovvero diverse *strutture dati*, ognuna con le proprie caratteristiche
  - Generalmente ogni struttura è più adatta a determinati algoritmi e problemi
- In questa parte del corso intruduciamo alcune strutture importanti: liste, alberi, hash
  - Metteremo in evidenza i problemi per cui sono piu' adeguate degli array

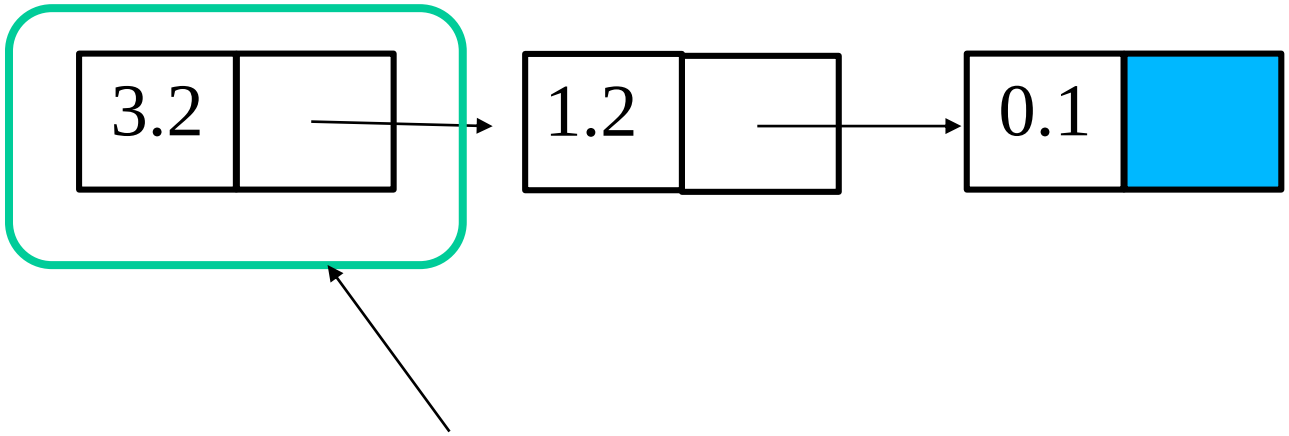
Liste (concatenate)

- Una lista concatenata è una sequenza lineare di elementi (detti anche nodi),
  - Ciascun elemento contiene un valore e l'indirizzo di memoria (puntatore) dell'elemento successivo nella sequenza
  - Ad esempio: una sequenza di double rappresentata con una lista è *logicamente* qualcosa di questo tipo



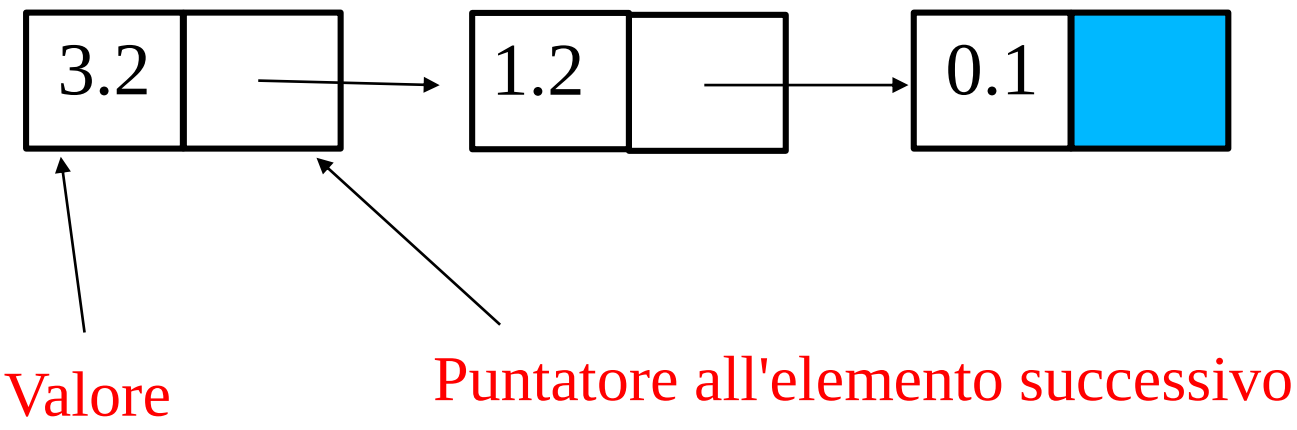
Liste (concatenate)

- Una lista concatenata è una sequenza lineare di elementi (detti anche nodi),
  - Ciascun elemento contiene un valore e l'indirizzo di memoria (puntatore) dell'elemento successivo nella sequenza
  - Ad esempio: una sequenza di double rappresentata con una lista è *logicamente* qualcosa di questo tipo



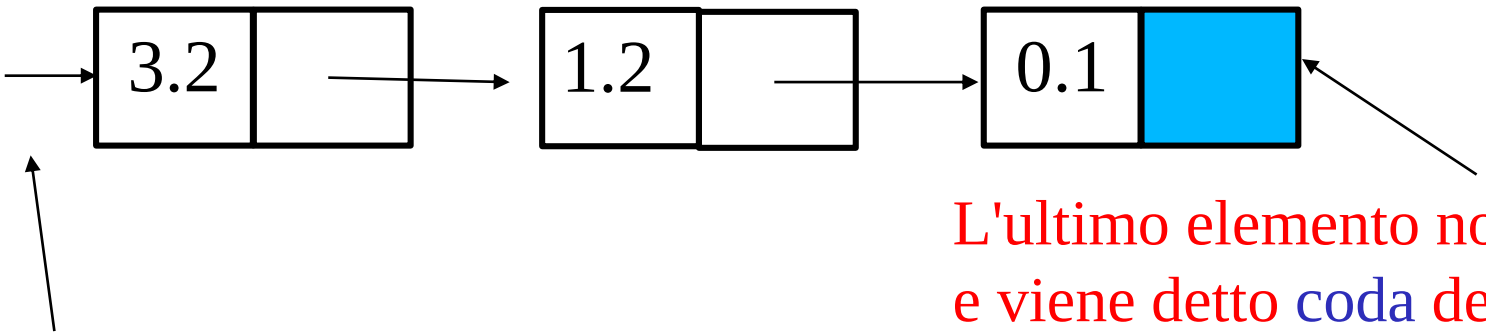
Liste (concatenate)

- Una lista concatenata è una sequenza lineare di elementi (detti anche nodi),
  - Ciascun elemento contiene un valore e l'indirizzo di memoria (puntatore) dell'elemento successivo nella sequenza
  - Ad esempio: una sequenza di double rappresentata con una lista è *logicamente* qualcosa di questo tipo



Liste (concatenate)

- Una lista concatenata è una sequenza lineare di elementi (detti anche nodi),
  - Ciascun elemento contiene un valore e l'indirizzo di memoria (puntatore) dell'elemento successivo nella sequenza
  - Ad esempio: una sequenza di double rappresentata con una lista è *logicamente* qualcosa di questo tipo



L'ultimo elemento non ha puntatori e viene detto coda della lista

Il puntatore al primo elemento è detto testa della lista

Liste in C

- Come si realizza una lista in C ?
  - Ogni elemento è una struttura che può contenere al suo interno un puntatore al suo stesso tipo, ad esempio per i double posso utilizzare:

```
typedef struct lista_d {
        double val;
        struct lista_d * next;
    } lista_d_t ;
```

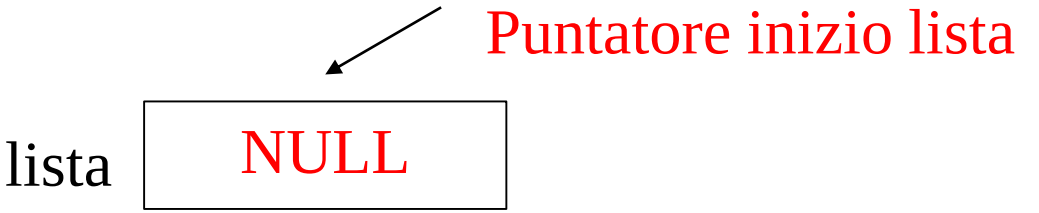
  - La testa della lista è un puntatore alla prima struttura (tipo **`lista_d_t *`**)
  - La coda della lista utilizza il puntatore nullo (**`NULL`**) nel campo **`next`** per indicare che non ci sono altri elementi da scandire

Liste in C

- Vediamo prima un esempio di come si crea una lista
  - Consideriamo la nostra solita sequenza di double letti dallo standard input
  - Vediamo come possiamo inserirli in una struttura di tipo lista secondo il seguente algoritmo:
    - Creo una lista, inizialmente vuota
    - Per ogni elemento presente in input:
      1. Leggo il valore
      2. Creo un elemento in cui inserisco il valore letto
      3. Aggiungo l'elemento alla lista

Liste in C

- Graficamente:
  - Iniziamo con la lista vuota ....



Liste in C

- Graficamente:
  - Leggiamo 3.45 da stdin

lista | NULL

?	?

Creiamo un nuovo elemento
(non inizializzato)

Liste in C

- Graficamente:
  - Leggiamo 3.45 da stdin

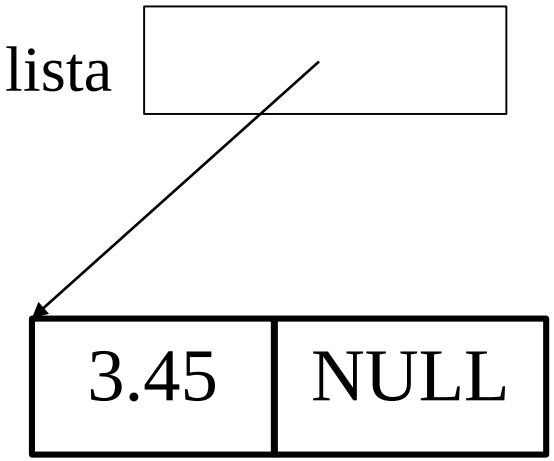
lista | NULL

Inseriamo il valore

3.45	?

Liste in C

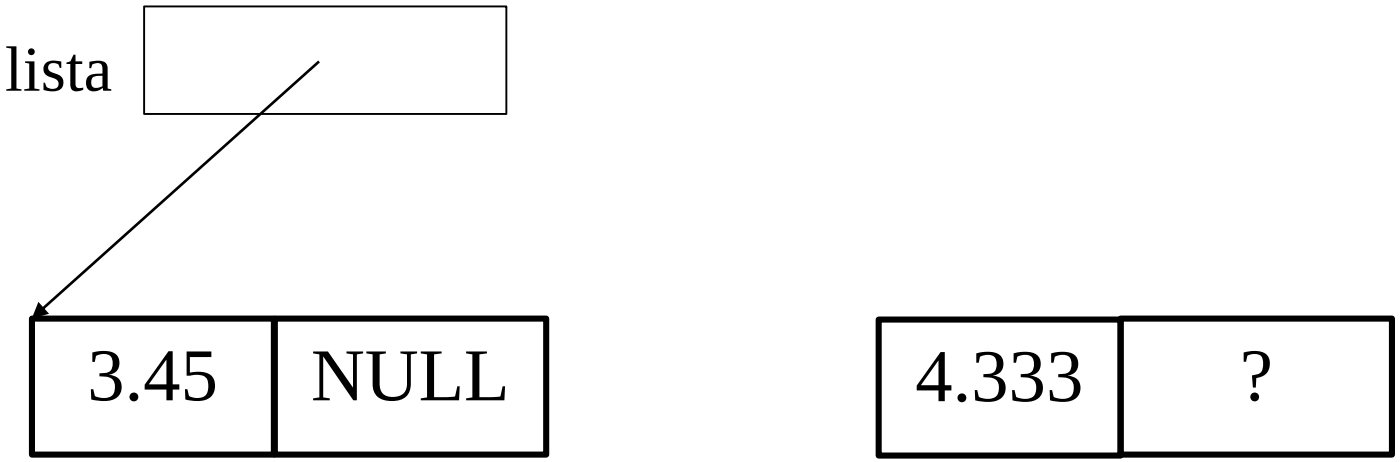
- Graficamente:
  - Leggiamo 3.45 da stdin



Colleghiamo alla lista

Liste in C

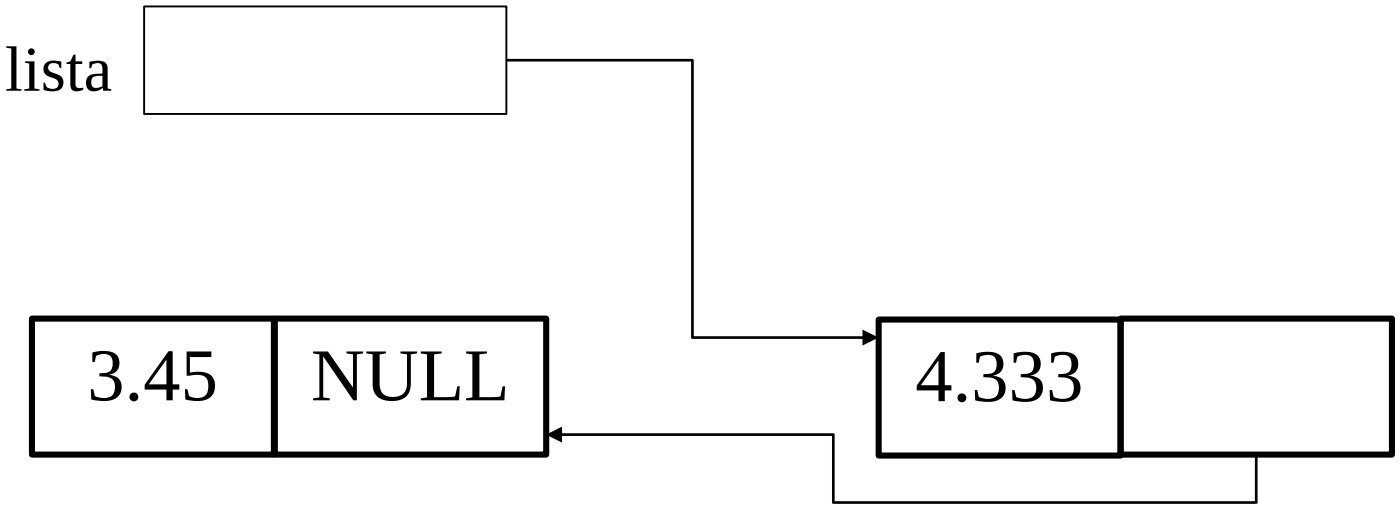
- Graficamente:
  - Leggiamo 4.333 da stdin



Creiamo un nuovo elemento e inseriamo il valore letto ....

Liste in C

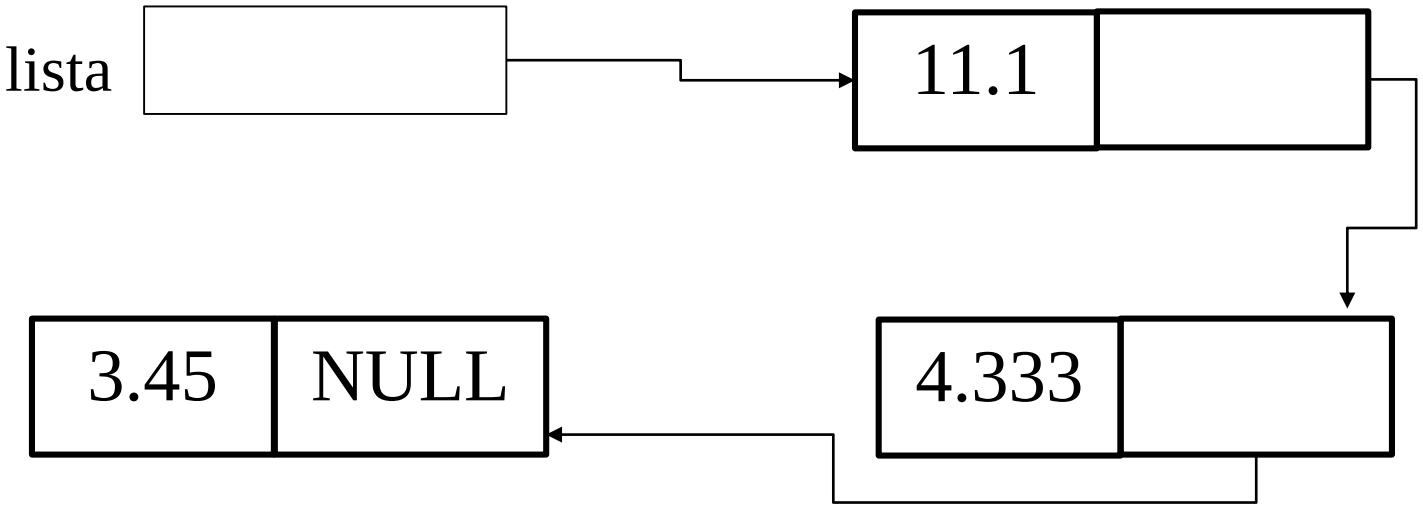
- Graficamente:
  - Leggiamo 4.333 da stdin



Colleghiamo alla lista

Liste in C

- Graficamente:
  - Leggiamo 11.1 da stdin ed inseriamo .....



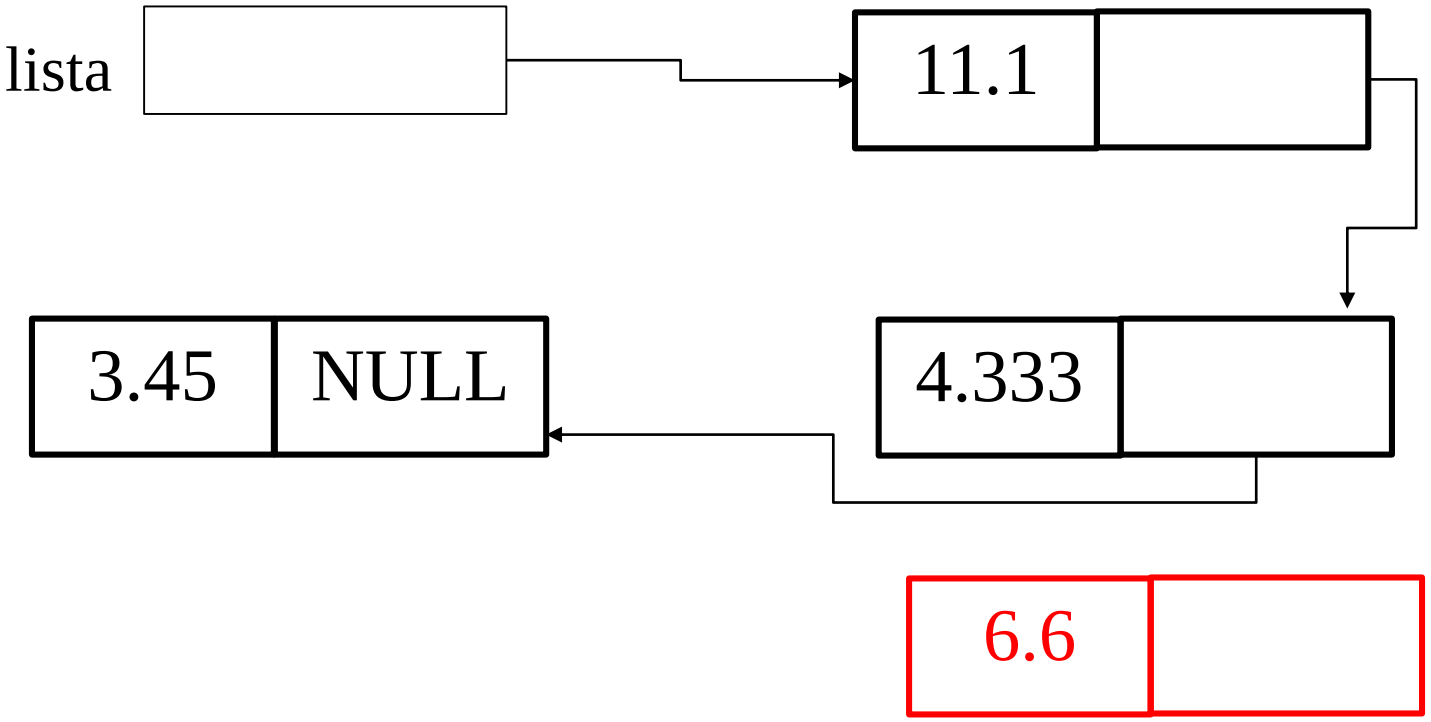
Etc ....

Liste in C

- Prima di vedere come codificare questo in C, discutiamo i vantaggi:
  - Possiamo espandere la struttura indefinitamente senza costose copie
  - Possiamo inserire semplicemente in mezzo alla struttura ....

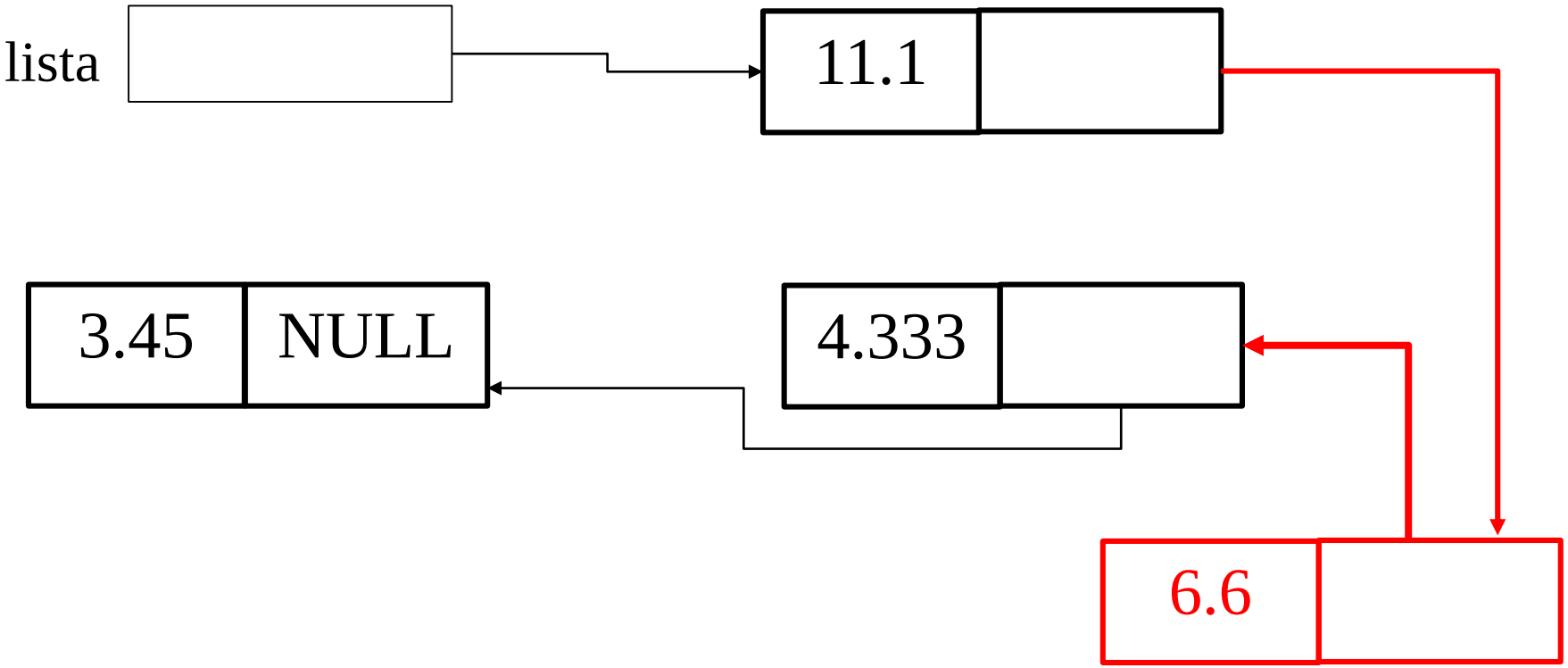
Liste in C

- Inserzione esempio:
  - Leggiamo 6.6 da stdin e vogliamo mantenere la lista ordinata in modo decrescente ....



Liste in C

- Inserzione esempio:
  - Leggiamo 6.6 da stdin e vogliamo mantenere la lista ordinata in modo decrescente ....

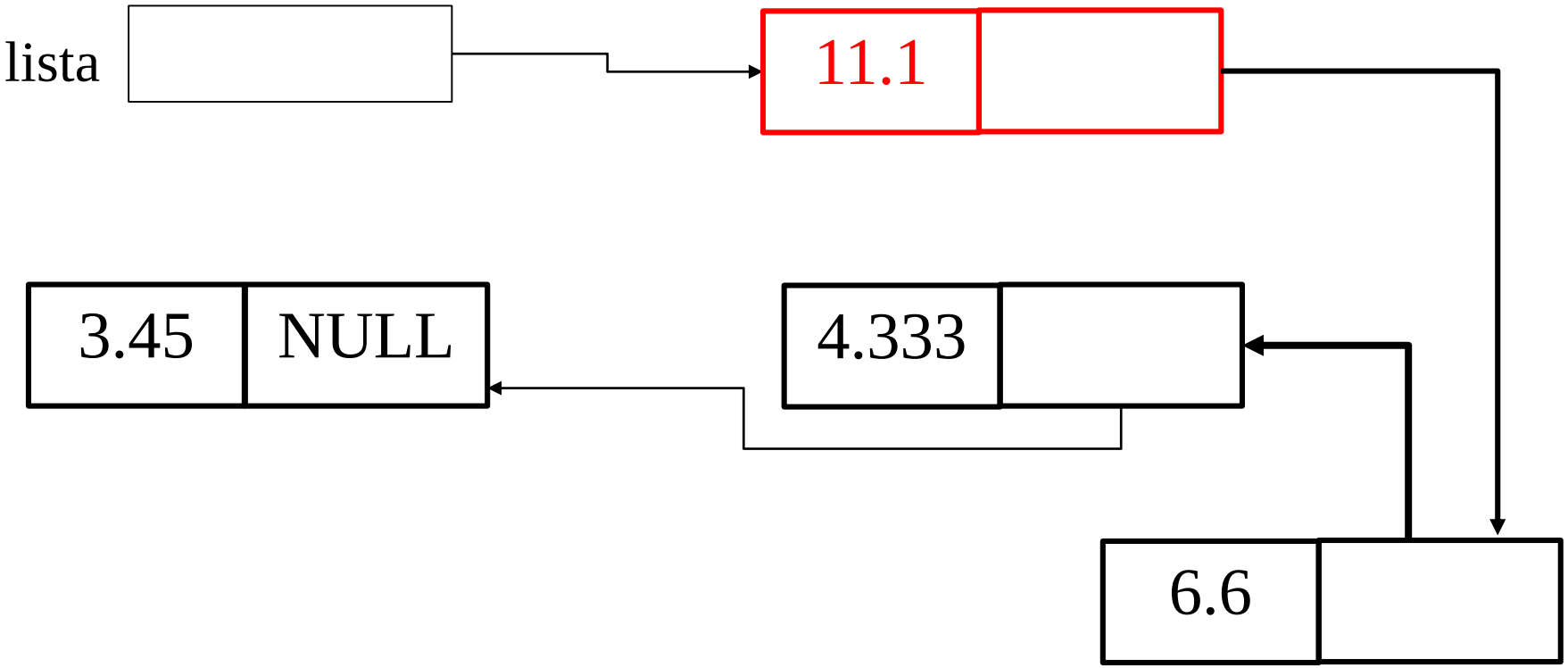


Liste in C

- Prima di vedere come codificare questo in C, discutiamo i vantaggi:
  - Possiamo espandere la struttura indefinitamente senza costose copie
  - Possiamo inserire semplicemente in mezzo alla struttura senza dover ricopiare tutto come accadrebbe con un array ....
  - Possiamo anche eliminare senza dover ricopiare tutto ...

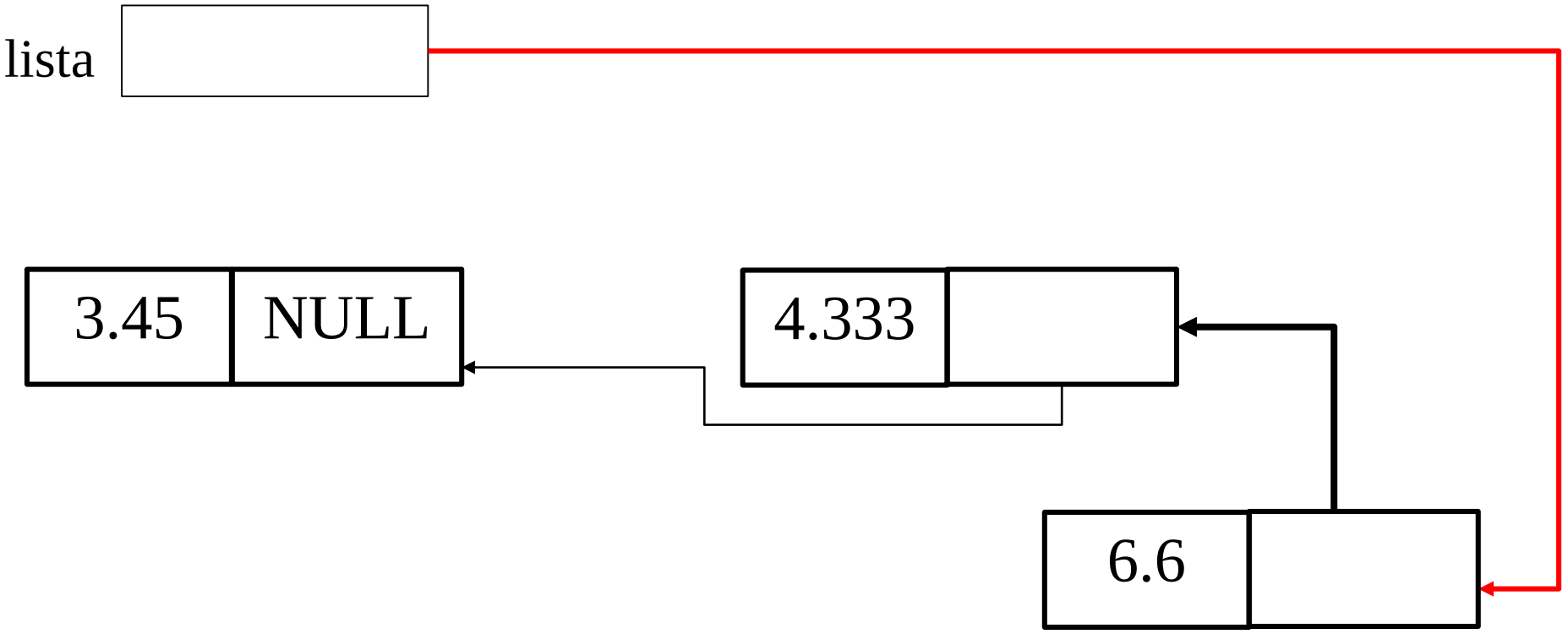
Liste in C

- Eliminazione esempio:
  - Vogliamo togliere 11.1 ....



Liste in C

- Eliminazione esempio:
  - Vogliamo togliere 11.1 ....



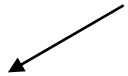
Liste in C

- Prima di vedere come codificare questo in C, discutiamo i vantaggi:
  - Possiamo espandere la struttura indefinitamente senza costose copie
  - Possiamo inserire semplicemente in mezzo alla struttura senza dover ricopiare tutto come accadrebbe con un array ....
  - Possiamo anche eliminare senza dover ricopiare tutto ...
  - Vedremo come questo può essere un grande vantaggio in situazioni in cui i dati hanno molta dinamicità ...

Sequenze come liste

```
/* lettura e memorizzazione di una sequenza di lunghezza non
   nota, valori reali terminati da 0.0 (come lista) */

#include <stdlib.h>

int main( void ) {

   /* creazione della lista vuota */

   lista_d_t * lista = NULL;

   lista_d_t * elem; double valore;

   do {

    valore = leggi_nuovo_valore(); /* leggo valore */

    if ( valore == 0 ) break;   /* ho finito ? */

    elem = malloc(sizeof(lista_d_t)); /* creo */

    elem -> val = valore; /* assegno il valore */

    elem->next = lista; /*aggiungo inizio lista (in testa)*/

    lista = elem; /* aggiorno il puntatore alla lista */

 } while ( true );  /* tipo enum visto */

/* altre eleborazioni e fine main */
```

Sequenze come liste



Puntatore inizio lista

Iterazione 1

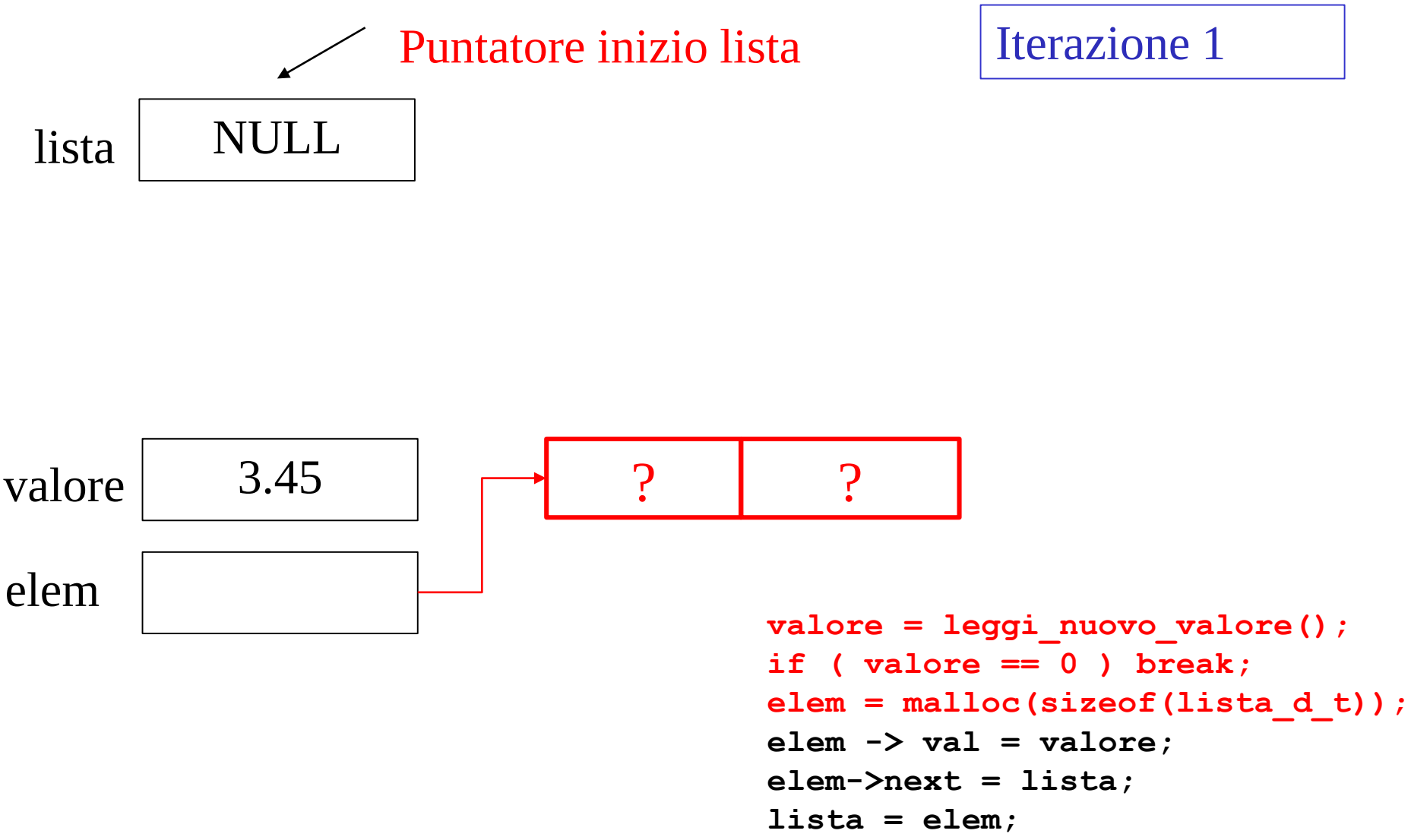
lista | NULL

valore | 3.45

elem | ?

```
valore = leggi_nuovo_valore();
if ( valore == 0 ) break;
elem = malloc(sizeof(lista_d_t));
elem -> val = valore;
elem->next = lista;
lista = elem;
```

Sequenze come liste

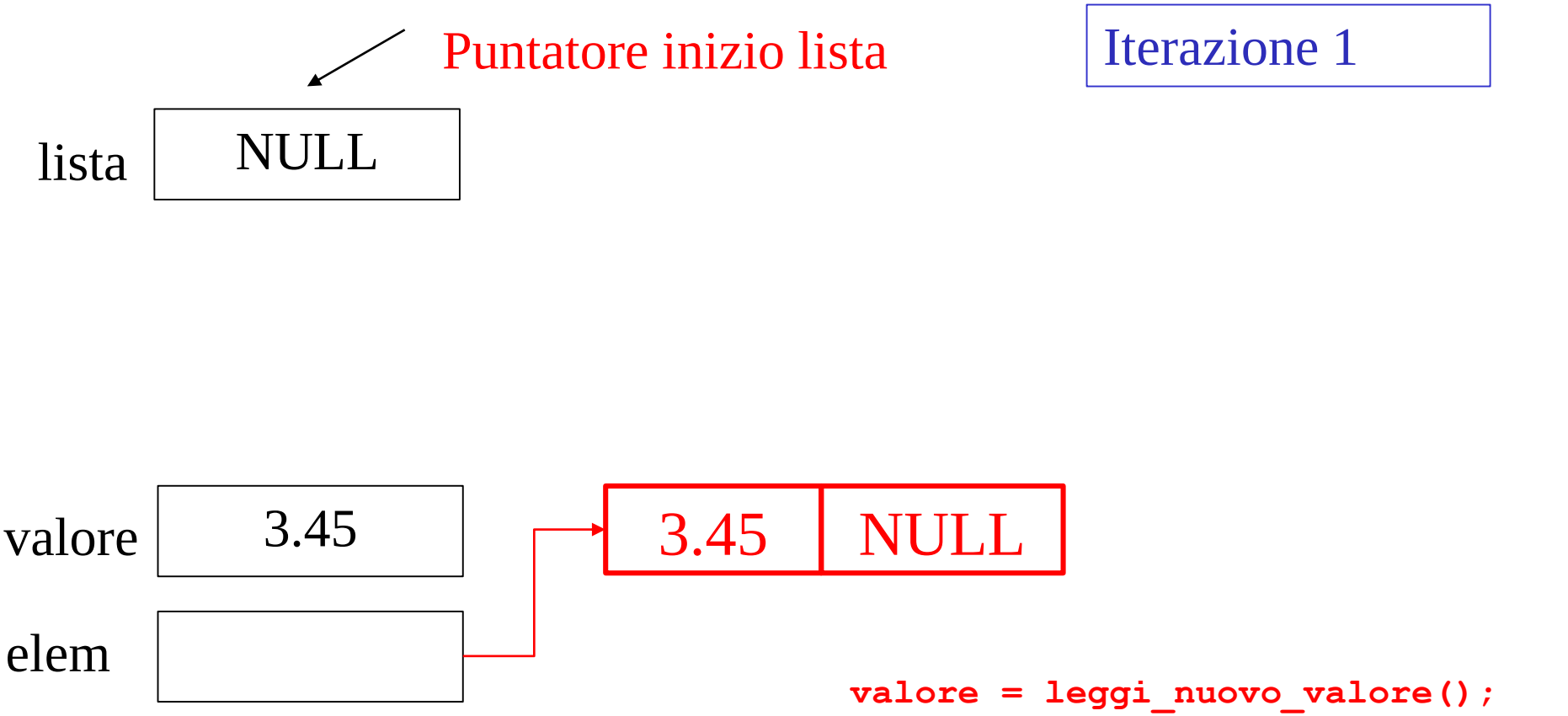



```
valore = leggi_nuovo_valore();
if ( valore == 0 ) break;
elem = malloc(sizeof(lista_d_t));
elem -> val = valore;
elem->next = lista;
lista = elem;
```

Sequenze come liste

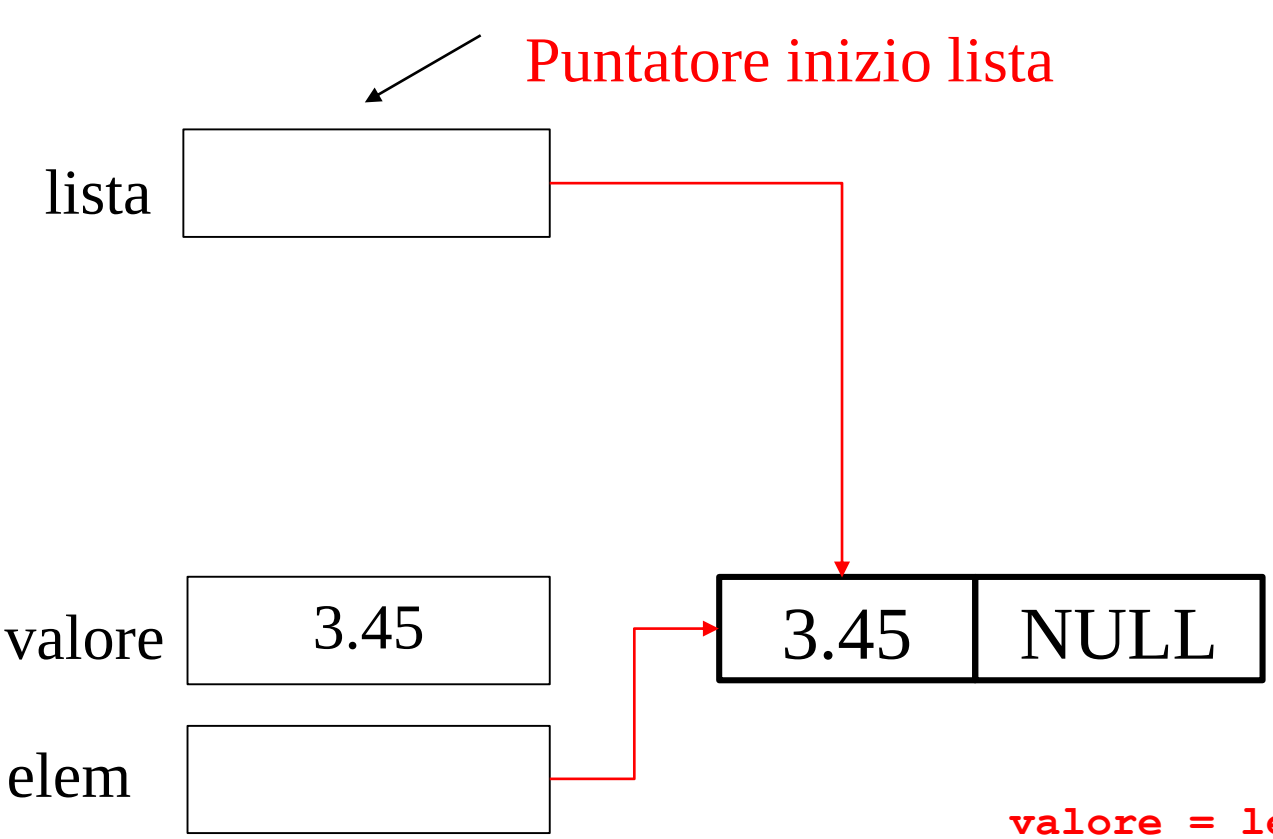

```
valore = leggi_nuovo_valore();
if ( valore == 0 ) break;
elem = malloc(sizeof(lista_d_t));
elem -> val = valore;
elem->next = lista;
lista = elem;
```

Sequenze come liste



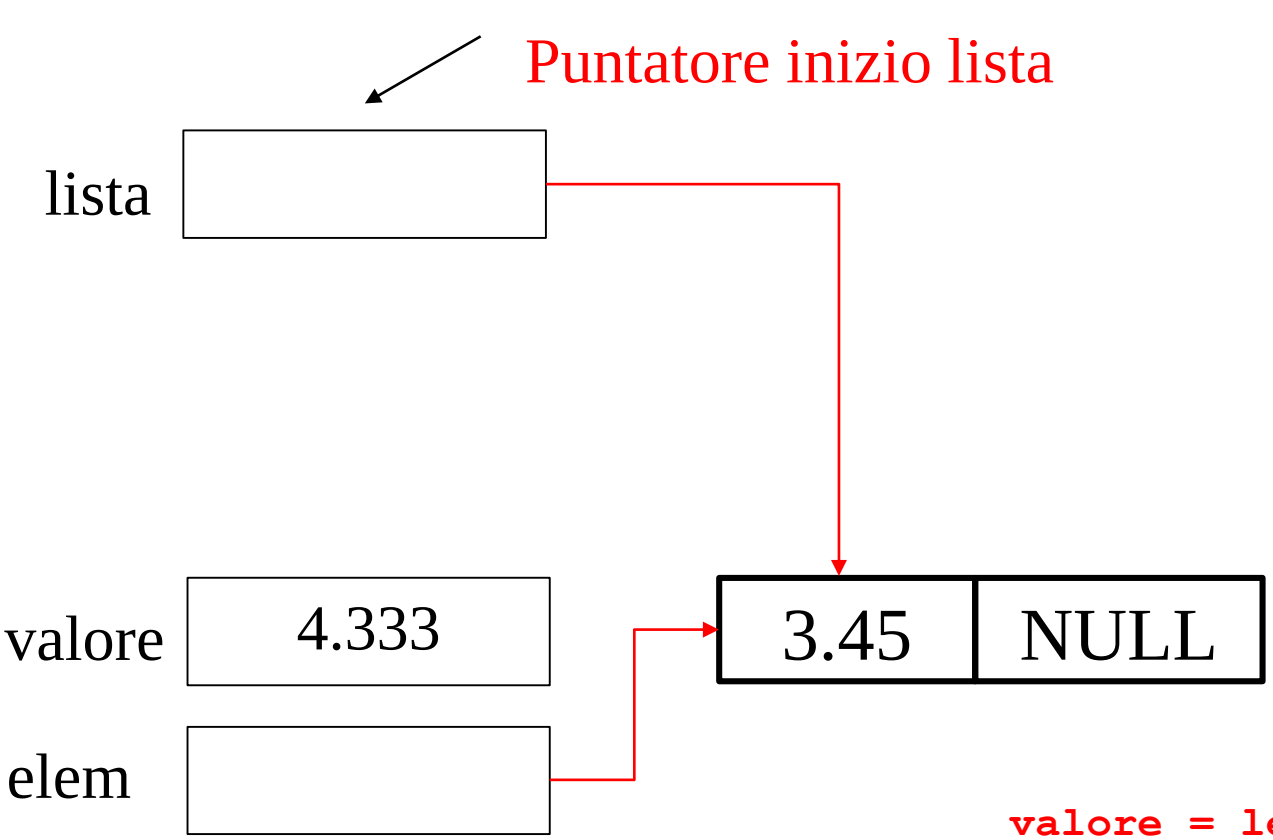

```
valore = leggi_nuovo_valore();
if ( valore == 0 ) break;
elem = malloc(sizeof(lista_d_t));
elem -> val = valore;
elem->next = lista;
lista = elem;
```

Sequenze come liste



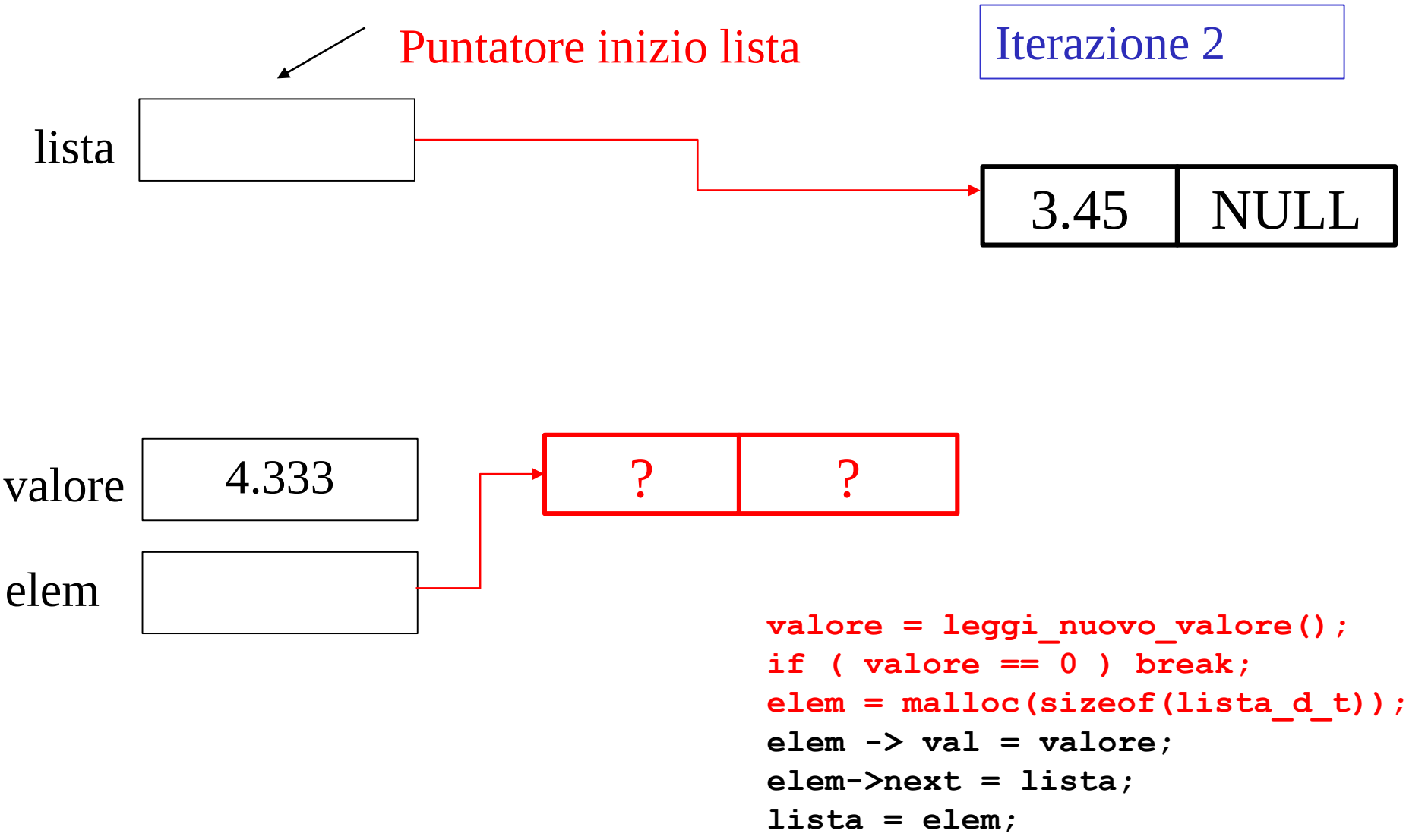

```
valore = leggi_nuovo_valore();
if ( valore == 0 ) break;
elem = malloc(sizeof(lista_d_t));
elem -> val = valore;
elem->next = lista;
lista = elem;
```

Sequenze come liste



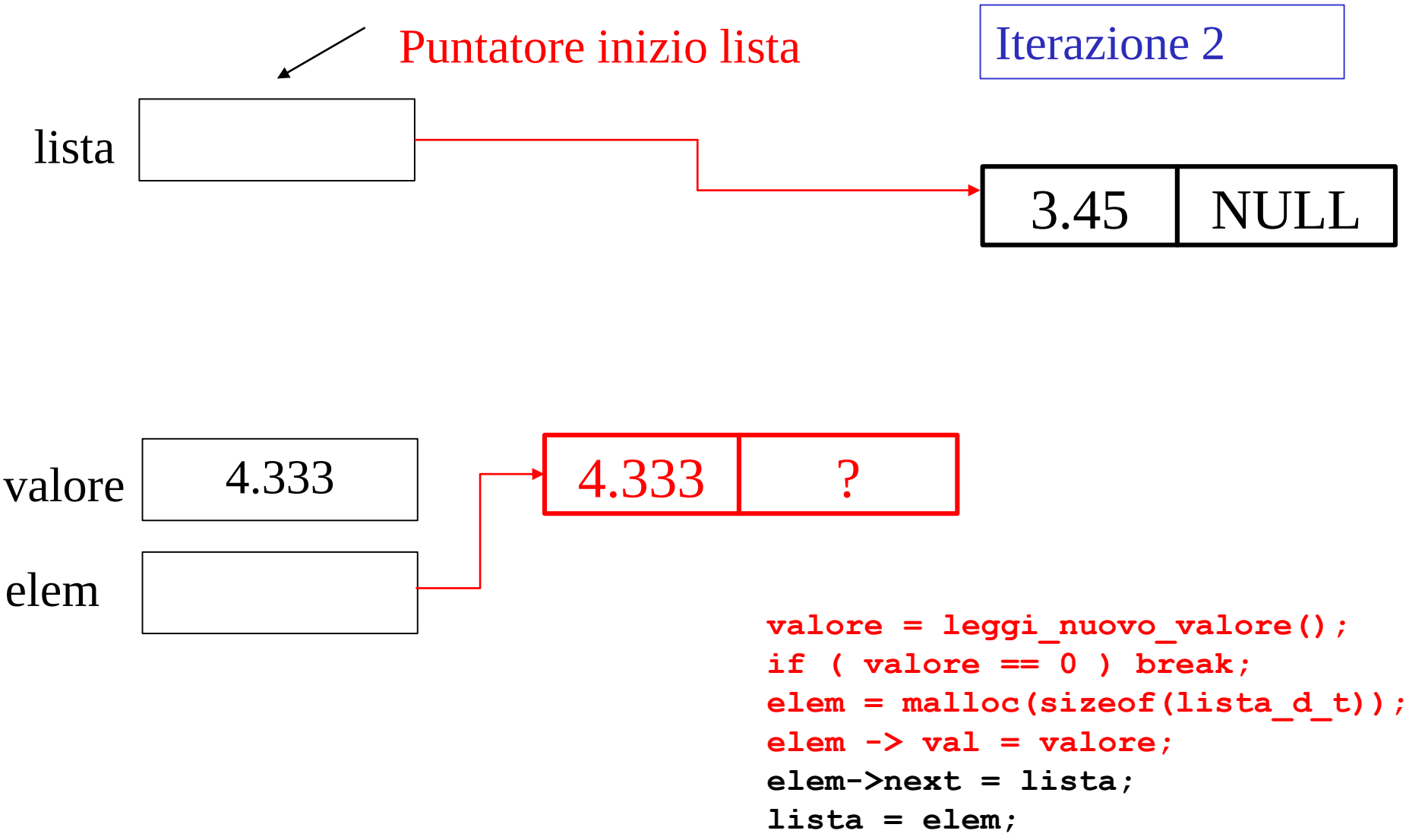

```
valore = leggi_nuovo_valore();
if ( valore == 0 ) break;
elem = malloc(sizeof(lista_d_t));
elem -> val = valore;
elem->next = lista;
lista = elem;
```

Sequenze come liste



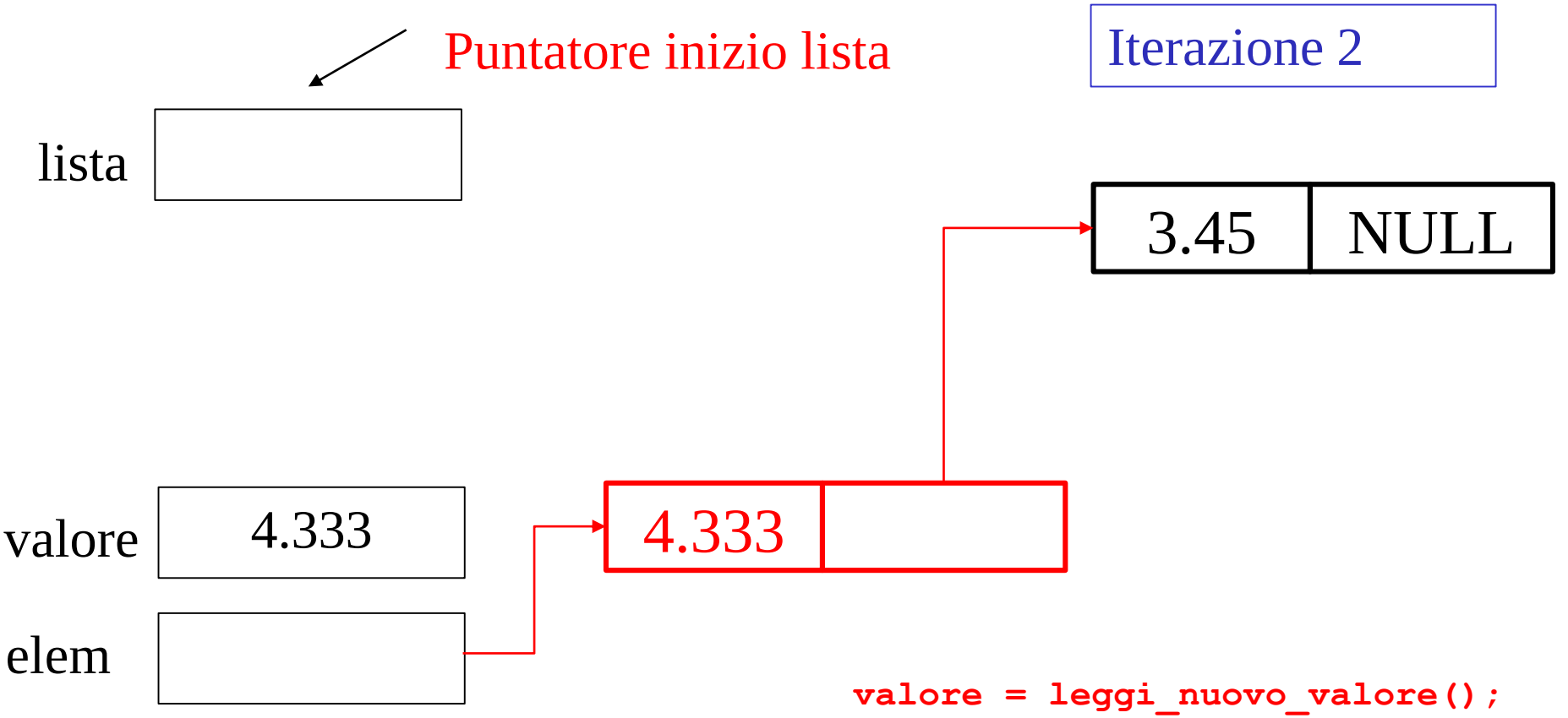

```
valore = leggi_nuovo_valore();
if ( valore == 0 ) break;
elem = malloc(sizeof(lista_d_t));
elem -> val = valore;
elem->next = lista;
lista = elem;
```

Sequenze come liste



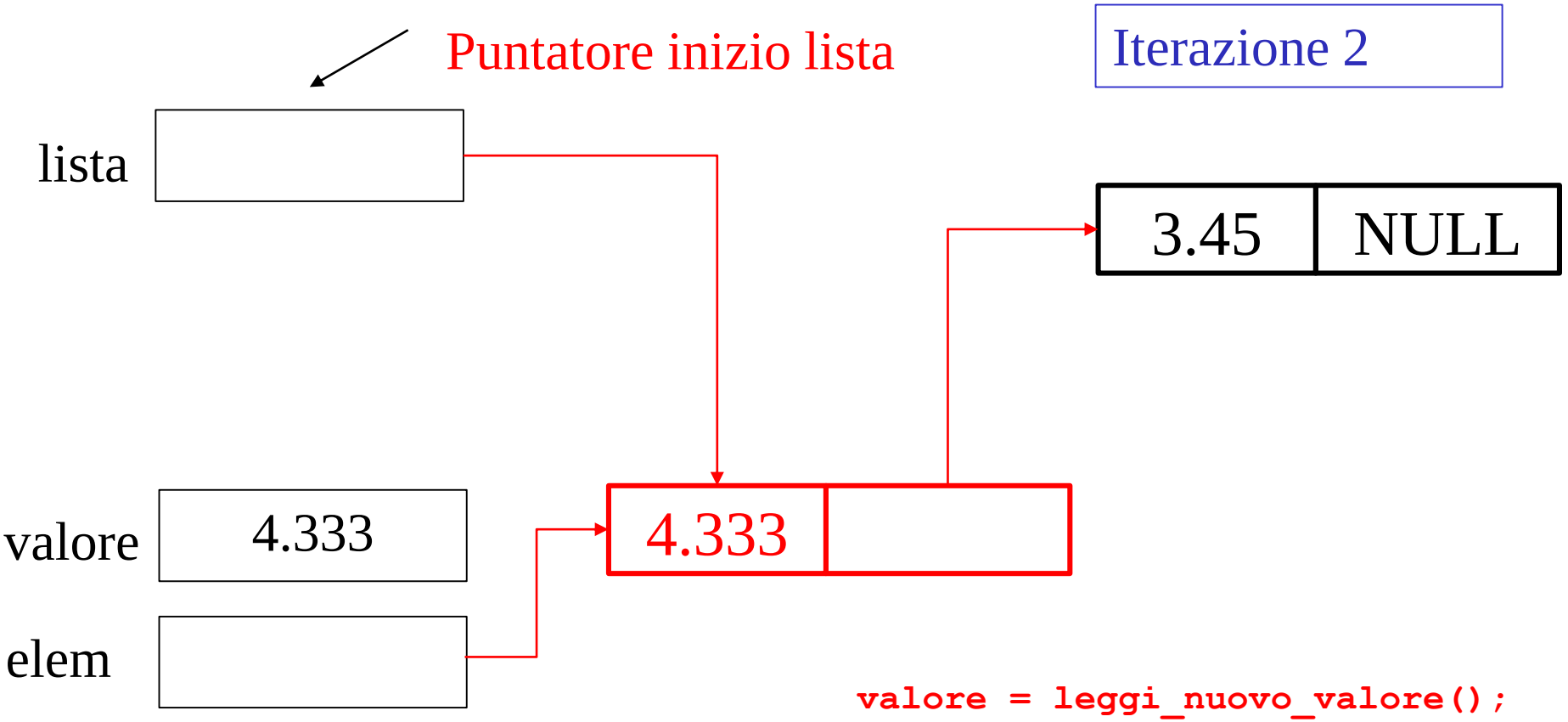

```
valore = leggi_nuovo_valore();
if ( valore == 0 ) break;
elem = malloc(sizeof(lista_d_t));
elem -> val = valore;
elem->next = lista;
lista = elem;
```

Sequenze come liste



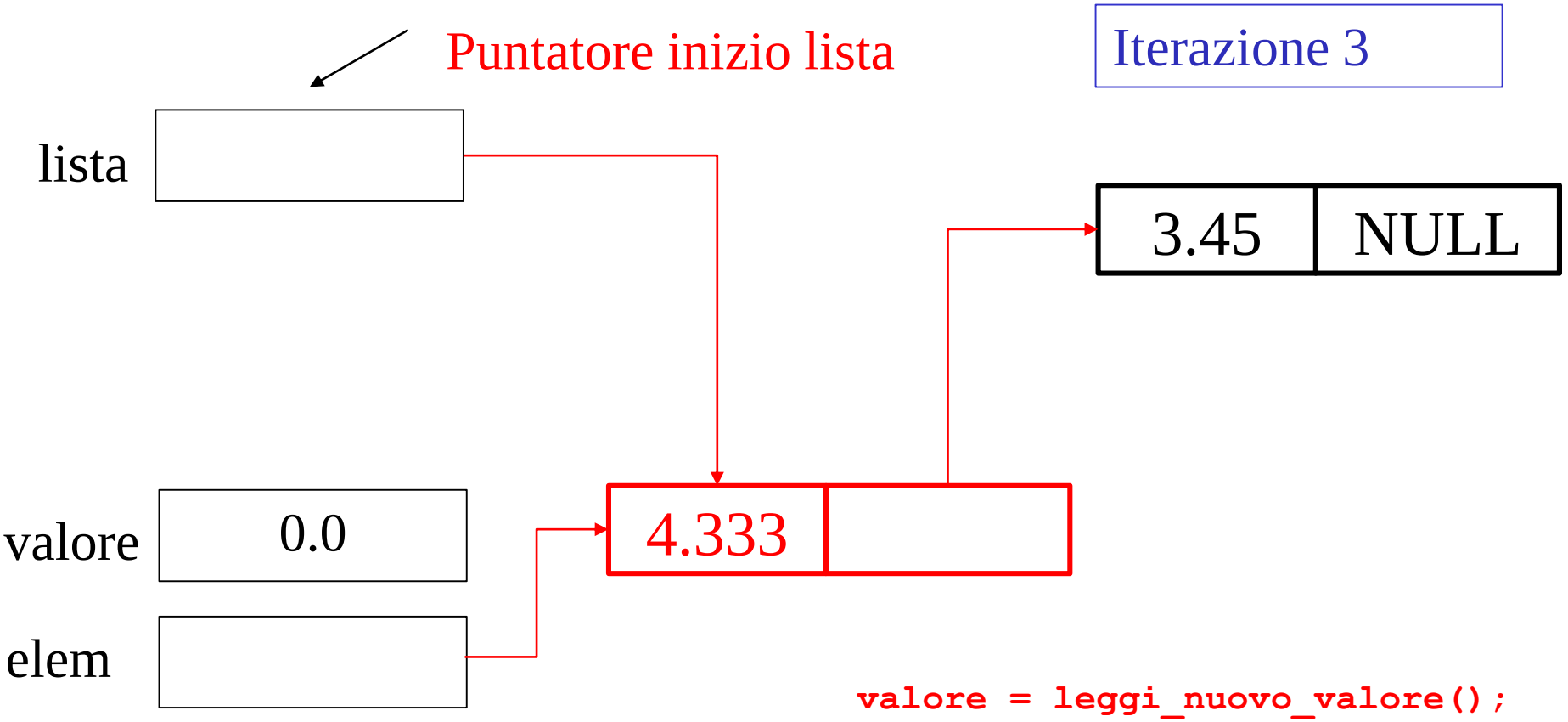

```
valore = leggi_nuovo_valore();
if ( valore == 0 ) break;
elem = malloc(sizeof(lista_d_t));
elem -> val = valore;
elem->next = lista;
lista = elem;
```

Sequenze come liste




```
valore = leggi_nuovo_valore();
if ( valore == 0 ) break;
elem = malloc(sizeof(lista_d_t));
elem -> val = valore;
elem->next = lista;
lista = elem;
```

Sequenze come liste




```
valore = leggi_nuovo_valore();
if ( valore == 0 ) break;
elem = malloc(sizeof(lista_d_t));
elem -> val = valore;
elem->next = lista;
lista = elem;
```

Inseriamo in coda

```
/* vediamo come aggiungere in coda alla lista .... */
#include <stdlib.h>
int main( void) ) {
   /* creazione della lista vuota */
   lista_d_t * lista = NULL;
   lista_d_t * elem, *p;
   double valore;
   do {
    valore = leggi_nuovo_valore(); /* leggo valore */
    if ( valore == 0 ) break;   /* ho finito ? */
    elem = malloc(sizeof(lista_d_t)); /* creo */
    elem -> val = valore; /* assegno il valore */
    /* questa prima parte è uguale all'inserzione in testa */
```

Inseriamo in coda

```
/* vediamo la parte dell'inserzione in coda ....*/
    /* cerco l'ultimo elemento */
    if ( lista == NULL )
       /* lista vuota, aggiorno il puntatore alla lista */
       lista = elem;
    else {
    p = lista ;
    while ( p -> next != NULL )
       p = p->next;
    p->next = elem; /* adesso l'ultimo punta a elem */
    elem->next = NULL; /* terminatore di lista */
   }
 } while ( true );
/* altre e leborazionie fine main */
}
```

Inseriamo in coda

```
/** inserisce in coda
   \param l puntatore alla lista
   \param v valore da inserire
   \retval ll puntatore alla nuova lista (con il il valore) */
lista_d_t * inserisci_coda ( lista_d_t * l, double v) {
   lista_d_t* elem, *p;
   elem = malloc(sizeof(lista_d_t));
   elem->val = v; elem ->next = NULL;
   if ( l == NULL ) return elem;
   p = l ;
   while ( p -> next != NULL ) p = p->next;
   p->next = elem;
   return l;
}
```

Inseriamo in coda

```
/* main usando la funzione inserisci_coda ... */
int main( void ) {
   lista_d_t * lista = NULL;
   double valore;
   valore = leggi_nuovo_valore(); /* leggo primo valore */
   while ( valore != 0  ) {
     lista = inserisci_coda(lista,valore);
     valore = leggi_nuovo_valore();
   }
  /*
    uso della lista ....
  */
  return 0;
} /* fine main */
```

Liste in C

- Prima di andare avanti lavoriamo ancora un po' su questo esempio ....
  - Funzione inserzione in testa e inserzione ordinata
  - Effetto della ricorsione ....
  - ... E non scordiamoci di dellocare la lista

- Continuiamo a tralasciare tutta la gestione errori, la implementeremo in laboratorio ....

Funzione inserisci in testa

```
/** inserisce in testa
   \param l puntatore alla lista
   \param v valore da inserire
   \retval ll puntatore alla nuova lista (con il il valore) */
lista_d_t * inserisci_testa ( lista_d_t * l, double v) {
   lista_d_t* elem;
   elem = malloc(sizeof(lista_d_t));
   elem->val = v;
   elem ->next = l;
   return elem;
}
```

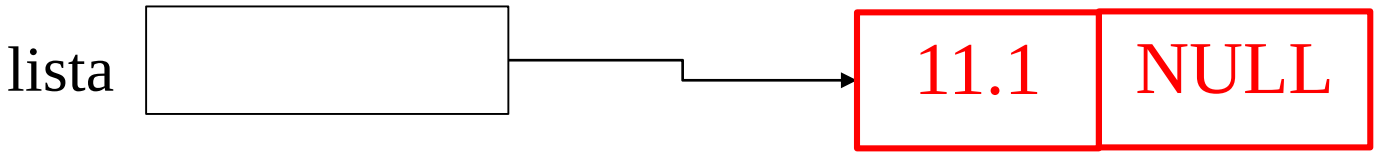
Funzione inserisci ordinato ...

Dobbiamo prima di tutto chiarirci i casi possibili :

- lista vuota: in questo creiamo un elemento e lo inseriamo in testa alla lista

Inserisci_ord: lista vuota

Inserisci_ord: lista vuota



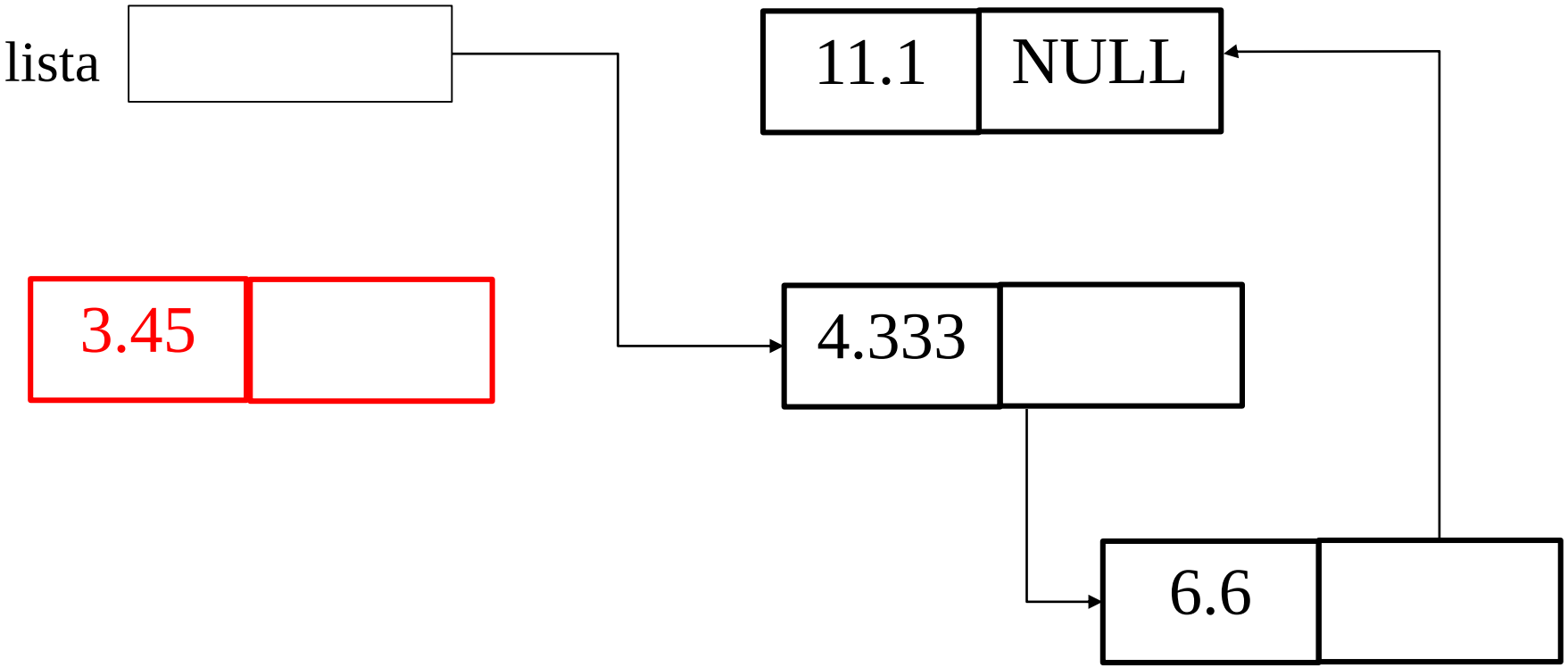
Funzione inserisci ordinato ...

Dobbiamo prima di tutto chiarirci i casi possibili :

- lista vuota: in questo creiamo un elemento e lo inseriamo in testa alla lista
- lista non vuota: abbiamo tre sottocasi:
  1. Inserimento in testa: il valore è minore di tutti quelli presenti
  2. Inserimento in coda: il valore è maggiore di tutti quelli presenti
  3. Inserimento in mezzo: esiste un valore minore o uguale ed uno successivo maggiore dentro la lista

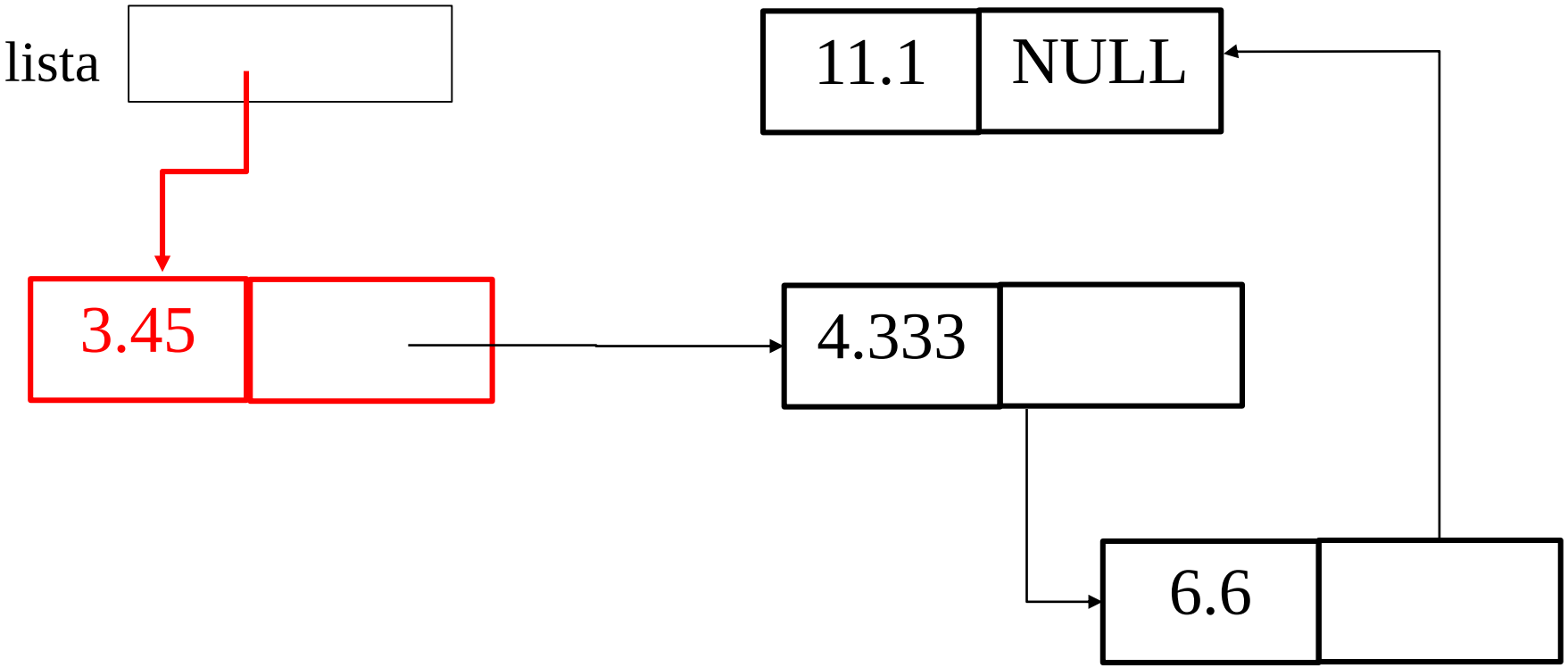
Inserisci_ord: lista non vuota

- Inserimento in testa, leggo 3.45



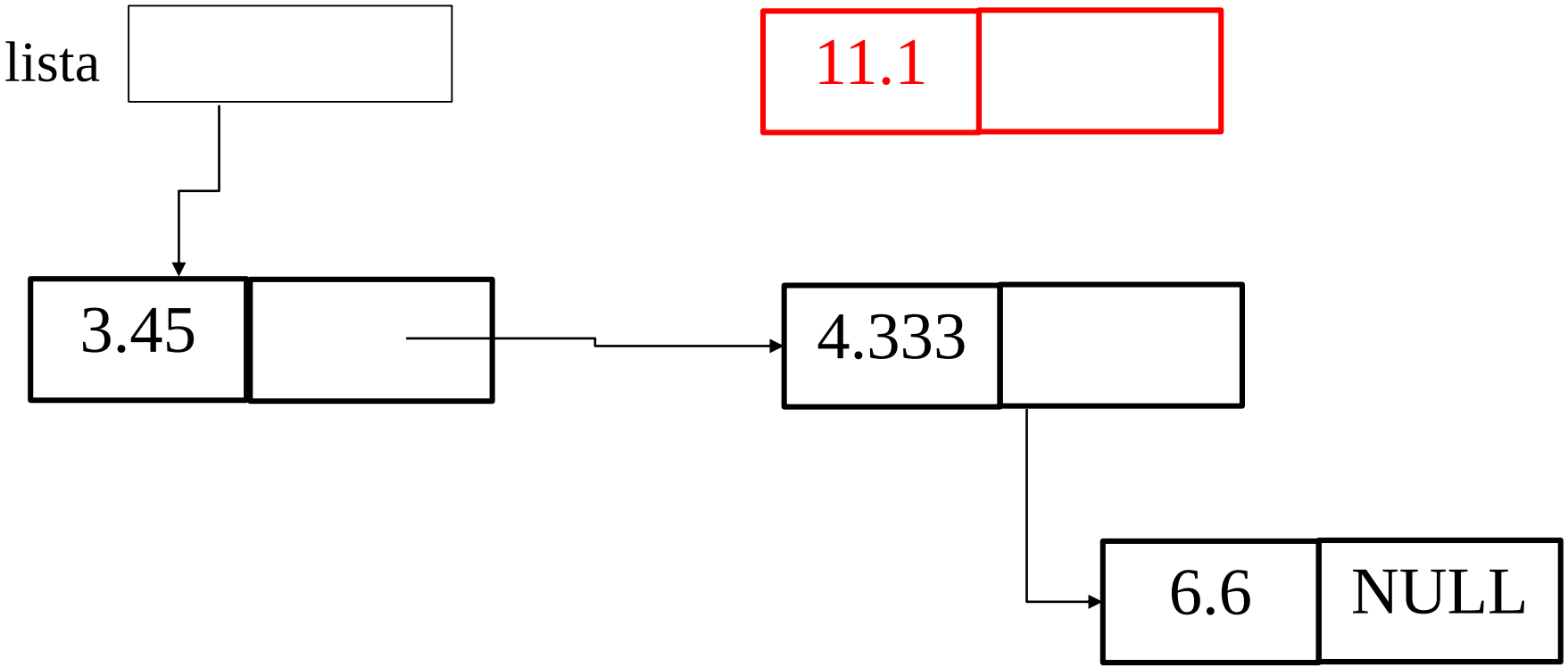
Inserisci_ord: lista non vuota

- Inserimento in testa, leggo 3.45
- È minore del primo elemento ...



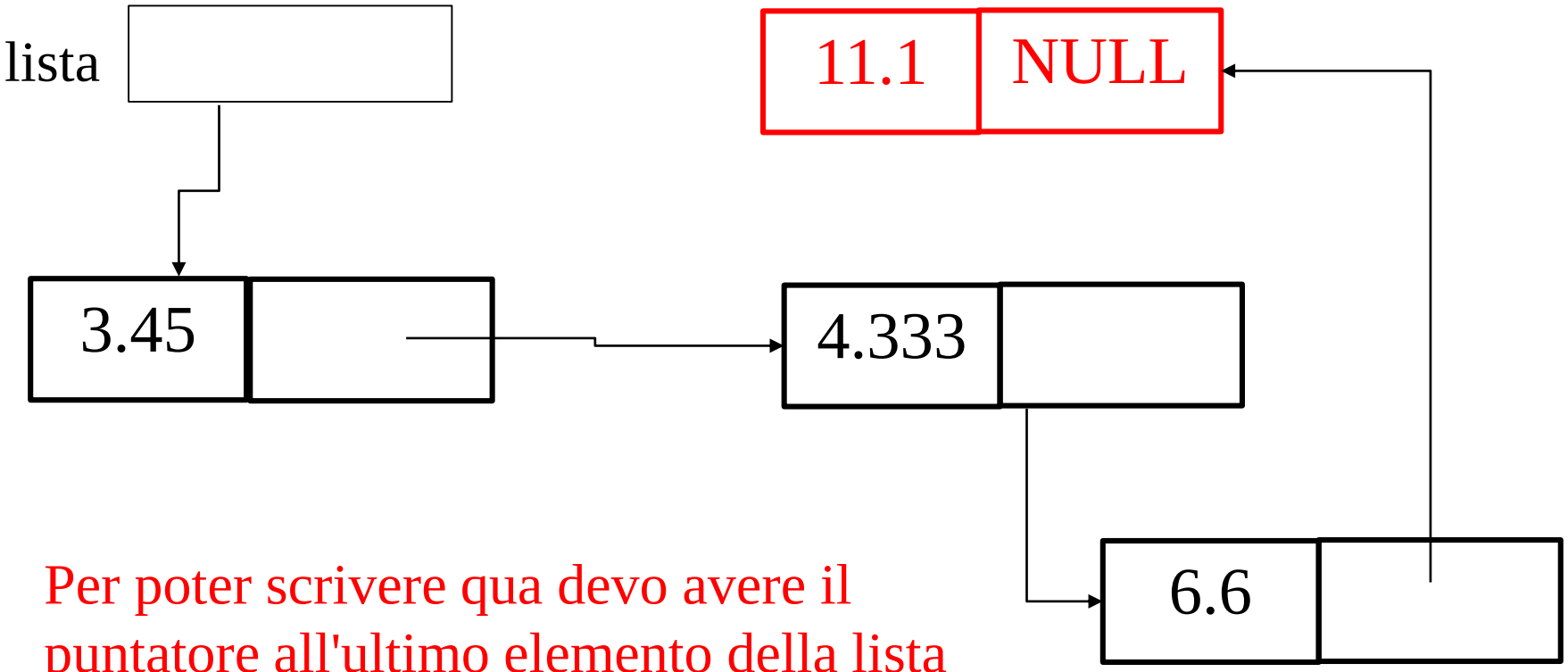
Inserisci_ord: lista non vuota

- Inserimento coda, leggo 11.1
- Scorro la lista trovando tutti valori minori



Inserisci_ord: lista non vuota

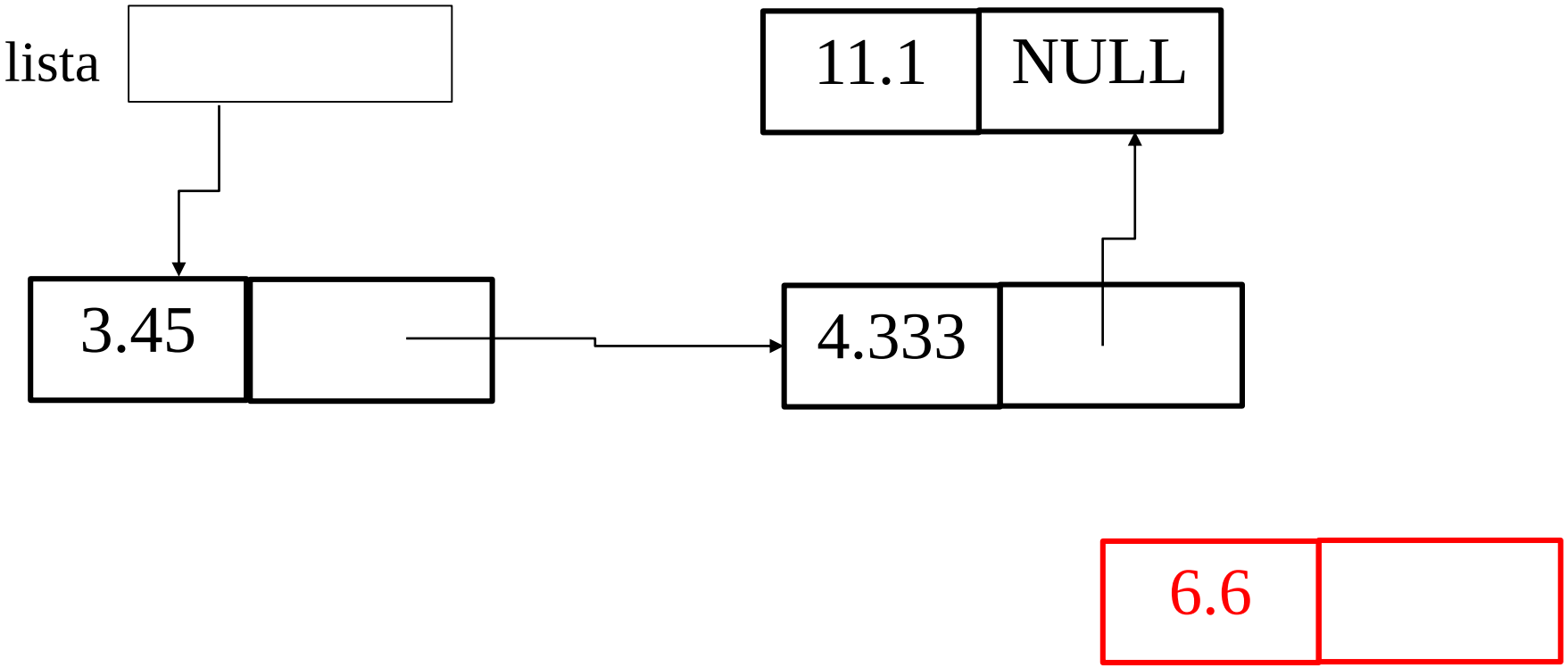
- Inserimento coda, leggo 11.1
- Scorro la lista trovando tutti valori minori



Per poter scrivere qua devo avere il puntatore all'ultimo elemento della lista

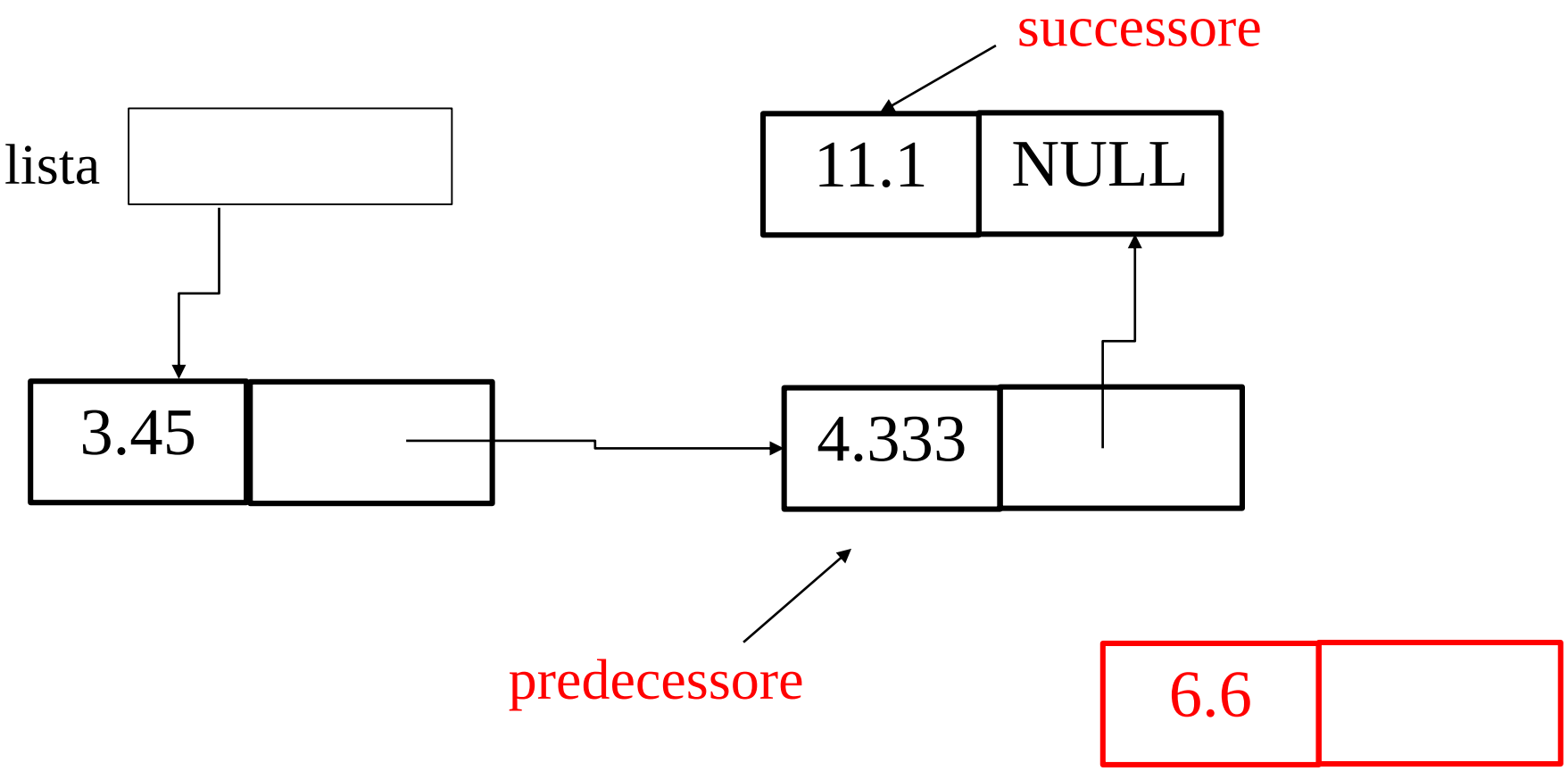
Inserisci_ord: lista non vuota

- Inserimento in mezzo, leggo 6,6
- Scorro la lista e trovo il valore minore e quello maggiore



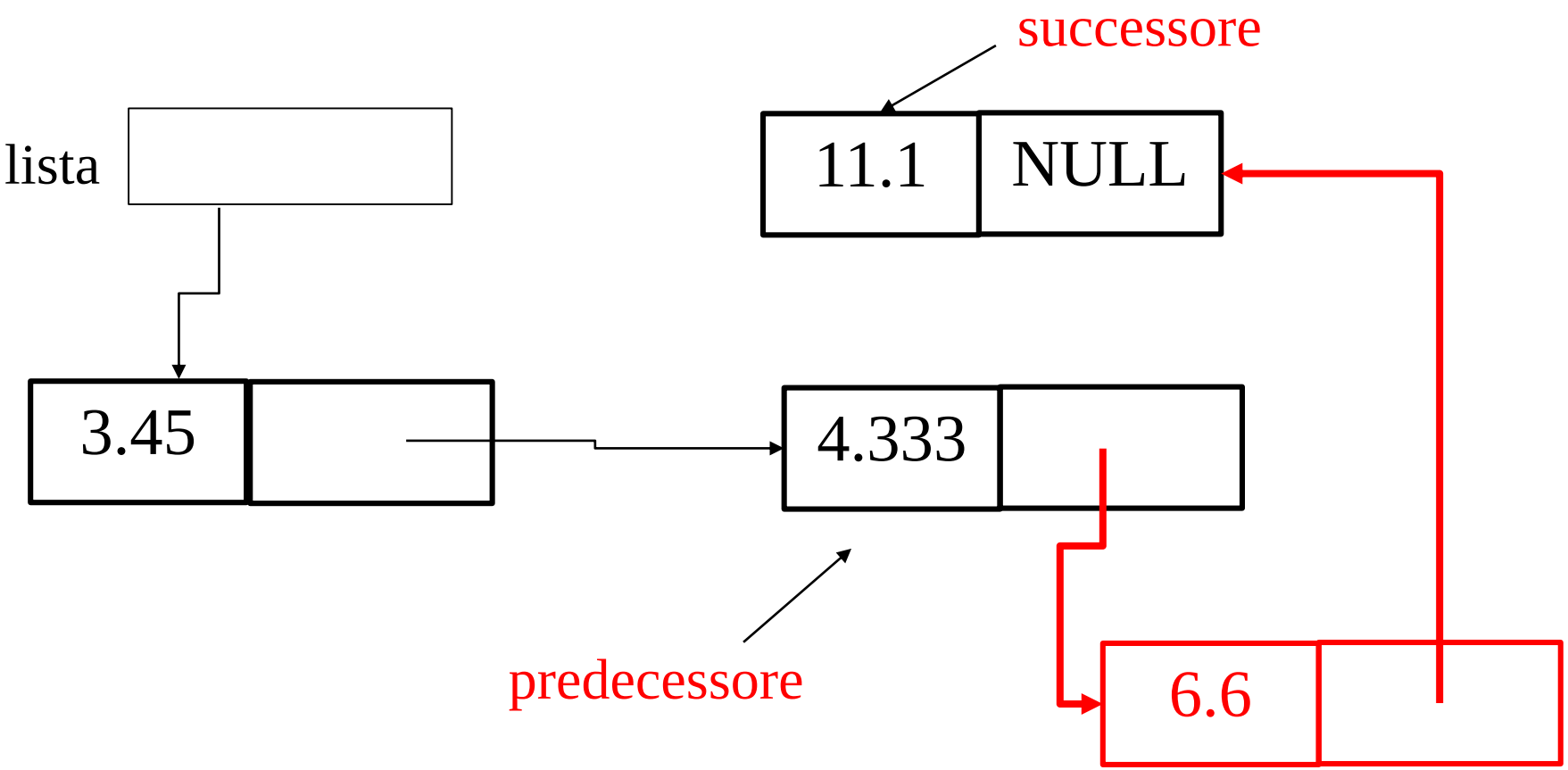
Inserisci_ord: lista non vuota

- Inserimento in mezzo, leggo 6,6
- Scorro la e trovo il valore minore e quello maggiore



Inserisci_ord: lista non vuota

- Inserimento in mezzo, leggo 6,6
- Scorro la e trovo il valore minore e quello maggiore



Funzione inserisci ordinato ...

```
/** inserisce in ordine crescente
   \param l puntatore alla lista
   \param v valore da inserire
   \retval l puntatore alla nuova lista (con il il valore) */
lista_d_t * inserisci_ord ( lista_d_t * l, double v) {
   lista_d_t* elem, *p, *q;
   /* creazione nuovo elemento */
   elem = malloc(sizeof(lista_d_t));
   elem->val = v; elem->next = NULL;
   /* caso lista vuota */
   if ( l == NULL ) return elem;
   /* segue */
```

Funzione inserisci ordinato ...

```
/* caso lista non vuota:
   cerco predecessore (q) e successore (p) */
  p = l ;
  q = NULL;
  while ( p != NULL && p -> val <= v ) {
    q = p;
    p = p->next;
  }
/* caso 1: inserzione in testa */
if ( q == NULL  ) {
  elem -> next = p;
  return elem ;
}
 /* segue */
```

Funzione inserisci ordinato ...

```
/* caso 2: inserzione in coda, q punta all'ultimo elemento */
if ( p == NULL  ) {
  elem -> next = NULL;
  q->next = elem ;
  return l ;
}
/* caso 3: inserzione in mezzo, q punta al precedente p al
successivo */
 elem-> next = p;
 q -> next = elem ;
 return l;
}
```

Funzione inserisci ordinato ricorsiva

```
lista_d_t * inserisci_ord_r ( lista_d_t * l, double v) {
   lista_d_t* elem;
   /* lista vuota */
   if ( l == NULL ) { elem = malloc(sizeof(lista_d_t));
     elem->val = v;
     elem ->next = NULL;
     return elem;
   }
 /* segue */
```

Funzione inserisci ordinato ricorsiva

```
lista_d_t * inserisci_ord_r ( lista_d_t * l, double v) {
   lista_d_t* elem;
   /* lista vuota */
   if ( l == NULL ) { elem = malloc(sizeof(lista_d_t));
     elem->val = v; elem ->next = NULL;
     return elem;
   }
   /* inserisco in testa */
   if ( l->val >= v ) { elem = malloc(sizeof(lista_d_t));
     elem->val = v;
     elem ->next = l;
     return elem;
   }
 /* segue */
  }
```

Funzione inserisci ordinato ricorsiva

```
lista_d_t * inserisci_ord_r ( lista_d_t * l, double v) {
   lista_d_t* elem;
   /* lista vuota */
   if ( l == NULL ) { elem = malloc(sizeof(lista_d_t));
     elem->val = v; elem ->next = NULL;
     return elem;
   }
   /* inserisco in testa */
   if ( l->val >= v ) { elem = malloc(sizeof(lista_d_t));
     elem->val = v; elem ->next = l;
     return elem;
   }
  /* inserisco in mezzo o in coda */
  l->next = inserisci_ord_r(l->next,v);
  return l;
}
```

Funzione inserisci ordinato ricorsiva

```
lista_d_t * inserisci_ord_r ( lista_d_t * l, double v) {
   lista_d_t* elem;
   /* lista vuota */
   if ( l == NULL ) { elem = malloc(sizeof(lista_d_t));
     elem->val = v; elem ->next = NULL;
     return elem;
   }
   /* inserisco in testa */
   if ( l->val >= v ) { elem = malloc(sizeof(lista_d_t));
     elem->val = v; elem ->next = l;
     return elem;
   }
  /* inserisco in mezzo o in coda */
  l->next = inserisci_ord_r(l->next,v);
  return l;
}
```

Si può migliorare ?

Funzione inserisci ordinato ricorsiva

```
lista_d_t * inserisci_ord_r ( lista_d_t * l, double v) {
   lista_d_t* elem;
   /* lista vuota */
   if ( l == NULL ) { elem = malloc(sizeof(lista_d_t));
     elem->val = v; elem ->next = NULL;
     return elem;
   }
   /* inserisco in testa */
   if ( l->val >= v ) { elem = malloc(sizeof(lista_d_t));
     elem->val = v; elem ->next = l;
     return elem;
   }
  /* inserisco in mezzo */
  l->next = inserisci_ord_r(l->next,v);
  return l;
}
```

Si può migliorare ?

Funzione inserisci ordinato ricorsiva

```
lista_d_t * inserisci_ord_r ( lista_d_t * l, double v) {
   lista_d_t* elem;
   /* lista vuota o inserimento in testa */
   if ( l == NULL  || l->val >= v ) {
     elem = malloc(sizeof(lista_d_t));
     elem->val = v; elem ->next = l;
     return elem;
   }
  /* inserisco in mezzo */
  l->next = inserisci_ord_r(l->next,v);
  return l;
}
```

Sviluppo di funzioni su liste

- Chiarirsi sempre (carta e penna!) quali sono i casi che possono accadere, generalmente
  - Lista vuota
  - Testa, centro e coda della lista non vuota ...

- Quando tutto è chiaro iniziare a codificare ed eventualmente ottimizzare accorpando i casi uguali
  - Evitare di ottimizzare se il codice diventa troppo difficile da comprendere
  - Codice chiaro anche se più lungo è molto più facile da modificare e verificare
  - Usare i commenti per chiarire ulteriormente ....

Funzione free_list ricorsiva

- Casi possibili
  - Lista vuota : non devo deallocare niente (caso base)
  - Lista non vuota di *n* elementi :
    - Suppongo di saper deallocare una lista di *n-1* elementi
    - In questo caso posso chiamare la funzione **`free_list()`** sulla lista che contiene dal secondo elemento in poi
    - E dopo deallocare il primo elemento con una sola **`free()`**

Funzione free_list ricorsiva

```
/** dealloca la lista
   \param l puntatore alla lista
*/
void free_list ( lista_d_t * l) {
   /* lista vuota */
   if ( l == NULL ) return;
   /* libero prima il resto della lista (dal secondo elemento
      in poi) con la chiamata ricorsiva */
   free_list(l->next);
   /* e poi la testa */
   free(l);
   return;
}
```

Esempio: stampa di una lista

- Casi possibili (versione ricorsiva)
  - Lista vuota : non devo stampare niente (caso base)
  - Lista non vuota di *n* elementi :
    - Suppongo di saper stampare una lista di *n-1* elementi
    - In questo caso posso stampare il primo elemento con una sola **`printf()`**
    - E dopo chiamare la funzione **`print_list()`** sulla lista che contiene dal secondo elemento in poi per stampare il resto

Funzione print_list ricorsiva

```
/** stampa la lista
   \param l puntatore alla lista
*/
void print_list ( lista_d_t * l) {
   /* lista vuota */
   if ( l == NULL ) return;
   /* stampo la testa */
   printf("%d ", l->val);
   /* stampo il resto della lista (dal secondo elemento
      in poi) con la chiamata ricorsiva */
   print_list(l->next);
return;
}
```

Esempio: stampa di una lista

- Casi possibili (versione iterativa)
  - Se p è il puntatore alla testa della lista
  - Finchè **`p`** non diventa **`== NULL`**:
    - Stampa il primo elemento con una sola **`printf()`**
    - Passa al prossimo elemento (**`p = p -> next`**)

Funzione print_list iterativa

```
/** stampa la lista
   \param l puntatore alla lista
*/
void print_list ( lista_d_t * l) {
   lista_d_t* p = t;
   while ( p != NULL ) {
     /* stampo la testa */
     printf("%d ", p->val);
     p = p->next; /* p punta adesso al prossimo elemento */
   }
  return;
}
```

Esempio: cancellazione di un elemento

- Voglio cancellare l'elemento di valore **`x`**
- Casi possibili (versione ricorsiva)
  - Se la lista è vuota non dobbiamo fare niente
  - Se la lista non è vuota, supponimo di sapere eleminare un elemento di valore x da una lista di **`n-1`** elementi
    - Controlla se il primo elemento **`p->val == x`** , se si cancella l'elemento
    - Richiamiamo ricorsivamente la funzione sulla lista puntata da **`p->next`**

Funzione remove_list ricorsiva

```
/** elimina dalla lista tutti gli elementi con i valori uguali ad x
   \param l puntatore alla lista
   \paral x il valore da eliminare
   \retval p il puntatore alla nuova lista */
lista_d_t* remove_list ( lista_d_t* l, double x) {
   lista_d_t* p;
   if ( l == NULL ) return l; /* lista vuota */
   l->next = remove_list(l->next,x); /* rimuovo dal resto della
  lista */
   /* controlla la testa */
   if ( x == l->val) { /* rimuovo  e libero memoria */
       p = l;
       l = l->next;
       free(p);   }
   return l;
}
```

Liste in C

- Riassumendo ....
  - La lista risolve efficientemente il problema di sequenze di lunghezza arbitraria e di composizione dinamica
  - È però più complessa da programmare, dobbiamo sempre aver chiari tutti i casi
  - La ricorsione può semplificare molto!
  - Dobbiamo ricordarci di deallocare la lista

Liste in C

- Liste o array ?
  - Se conosco la lunghezza e questa rimane fissa l'array e' molto piu' efficiente
  - Accedere all'i-esimo elemento di un array costa una lettura in memoria
  - Accedere all'i-esimo elemento di una lista costa *i* letture in memoria (dobbiamo seguire tutti i puntatori)
- In laboratorio:
  - Implementeremo altre funzioni e gestiremo gli errori di allocazione