Il linguaggio C
Strutture

Moreno Marzolla
Dipartimento di Informatica—Scienza e Ingegneria (DISI)
Università di Bologna
http://www.moreno.marzolla.name/

Ringraziamenti

- Questi lucidi si basano su materiale fornito dal prof. Mirko Viroli, Università di Bologna

Nuovi tipi in C: `struct`

- Le strutture (struct) sono costituite da un insieme di campi
- Ogni campo ha un nome e un tipo
- Es: **`struct complex`** ha i campi
  - **`re`** di tipo *double* (parte reale)
  - **`im`** di tipo *double* (parte complessa)

```
/* complex.c */
#include <stdio.h>

struct complex {
     double re;
     double im;
};

int main( void )
{
     struct complex v1, v2, s;
     v1.re = 1.0; v1.im = -3.0;
     v2.re = 2.0; v2.im = 2.0;
     s.re = v1.re + v2.re;
     s.im = v1.im + v2.im;
     printf("%f + (%fi)\n",
            s.re, s.im);
     return 0;
}
```

Attenzione...

Dichiarare variabili di tipo struttura

- Dichiarazione senza inizializzazione

```
struct complex val;
```

- Dichiarazione con inizializzazione

```
struct complex val = {1.0, 2.0};
```

ANSI C richiede che i valori dell'inizializzazione delle strutture siano dei letterali (quindi niente variabili)

```
#include <stdio.h>

struct complex { double re, im; };

int main( void )
{
    struct complex v1 = {1.0, -3.0}, v2 = {2.0, 2.0}, s;
    s.re = v1.re + v2.re;
    s.im = v1.im + v2.im;
    printf("%f + (%fi)\n", s.re, s.im);
    return 0;
}
```

Strutture come parametri di funzioni

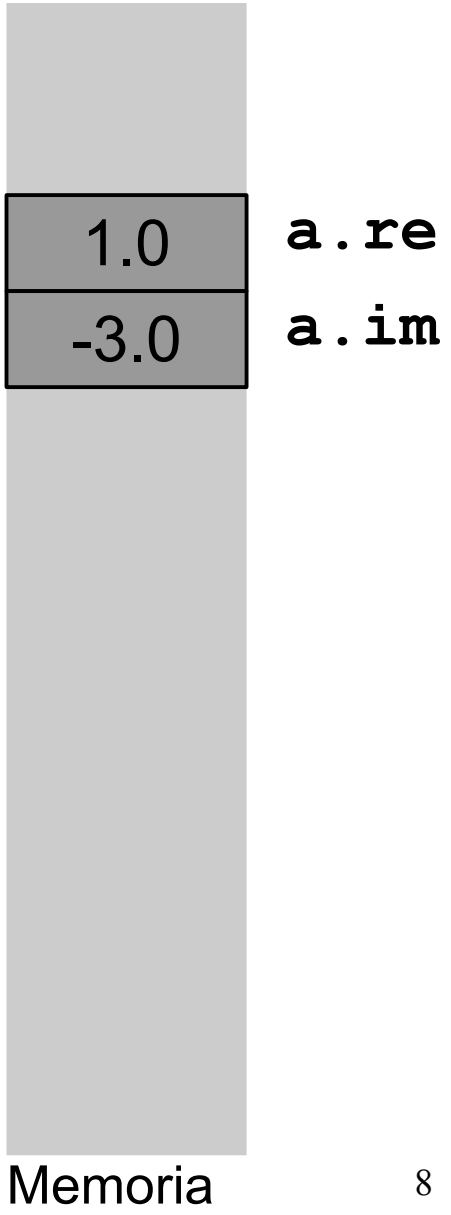
```
/* somma-complex.c: somma due numeri complessi */
#include <stdio.h>

struct complex {
    double re, im;
};

struct complex somma(struct complex v1, struct complex v2)
{
    struct complex r;
    r.re = v1.re + v2.re;
    r.im = v1.im + v2.im;
    return r;
}

int main( void )
{
    struct complex a = {1.0, 2.0};
    struct complex b = {-1.0, 3.0};
    struct complex s = somma(a, b);
    printf("%f+(%fi)\n", s.re, s.im);
    return 0;
}
```

Puntatori a strutture

```
/* incr-complex.c: passaggio di puntatori a strutture */
#include <stdio.h>

struct complex {
    double re, im;
};

void incr(struct complex *v)
{
     (*v).re = (*v).re + 1.0;
     (*v).im = (*v).im + 1.0;
}

int main( void )
{
     struct complex a = {1.0, -3.0};
     incr( &a );
     printf("%f+(%fi)\n", a.re, a.im);
     return 0;
}
```

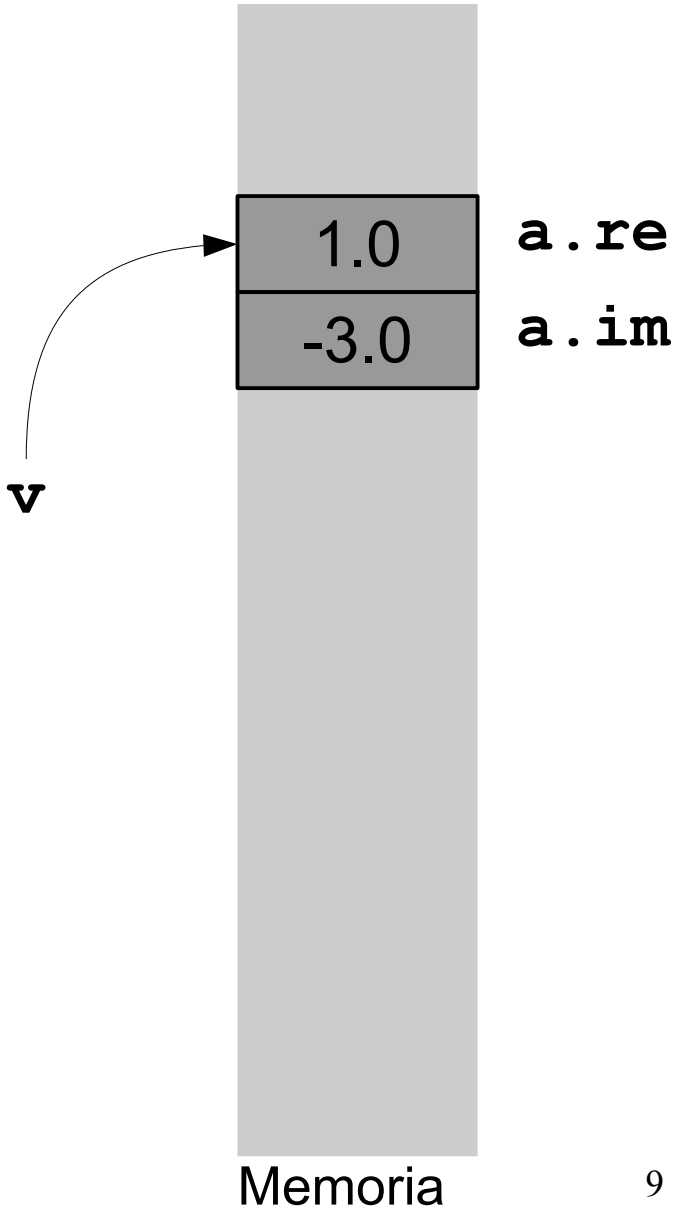
Puntatori a strutture

```
/* incr-complex.c: passaggio di puntatori a strutture */
#include <stdio.h>

struct complex {
    double re, im;
};

void incr(struct complex *v)
{
     (*v).re = (*v).re + 1.0;
     (*v).im = (*v).im + 1.0;
}

int main( void )
{
     struct complex a = {1.0, -3.0};
     incr( &a );
     printf("%f+(%fi)\n", a.re, a.im);
     return 0;
}
```



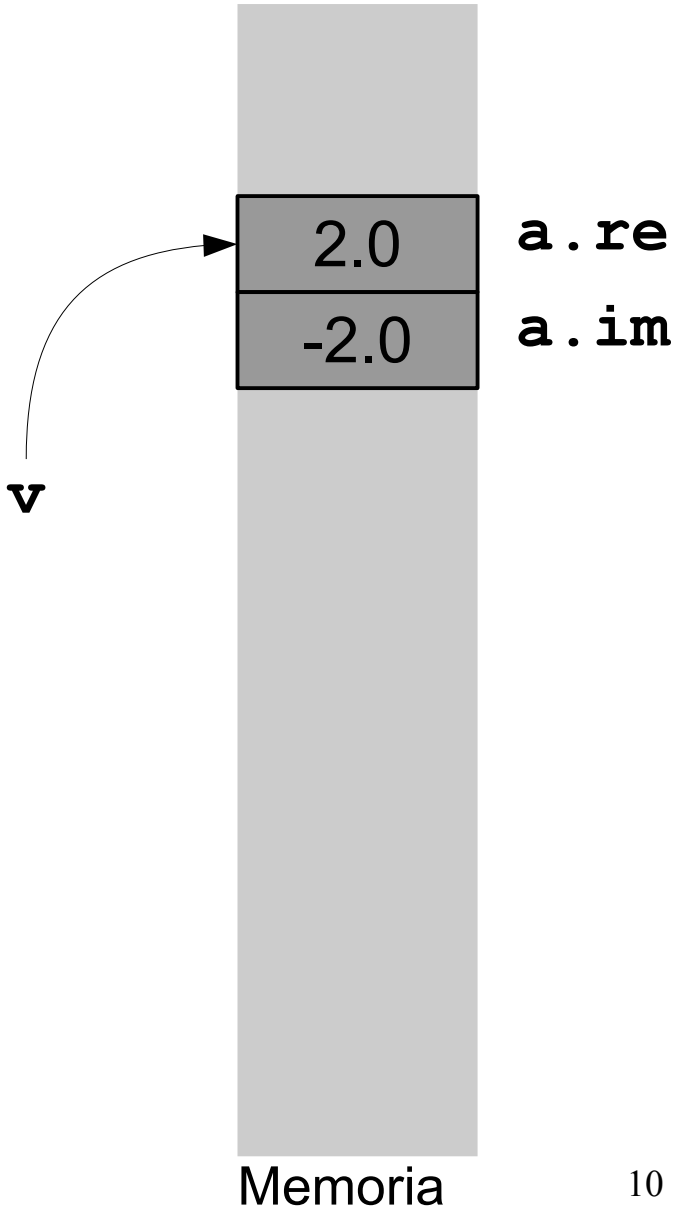
Puntatori a strutture

```
/* incr-complex.c: passaggio di puntatori a strutture */
#include <stdio.h>

struct complex {
    double re, im;
};

void incr(struct complex *v)
{
     (*v).re = (*v).re + 1.0;
     (*v).im = (*v).im + 1.0;
}

int main( void )
{
     struct complex a = {1.0, -3.0};
     incr( &a );
     printf("%f+(%fi)\n", a.re, a.im);
     return 0;
}
```



Puntatori a strutture

```
/* incr-complex.c: passaggio di puntatori a strutture */
#include <stdio.h>

struct complex {
    double re, im;
};

void incr(struct complex *v)
{
     (*v).re = (*v).re + 1.0;
     (*v).im = (*v).im + 1.0;
}

int main( void )
{
     struct complex a = {1.0, -3.0};
     incr( &a );
     printf("%f+(%fi)\n", a.re, a.im);
     return 0;
}
```



Allocare strutture con `malloc()`

```
/* struct-malloc.c : allocare strutture con malloc() */
#include <stdio.h>
#include <stdlib.h> /* per usare malloc() */

struct complex {
    double re, im;
};

void incr(struct complex *v)
{
     (*v).re = (*v).re + 1.0;
     (*v).im = (*v).im + 1.0;
}

int main( void )
{
     struct complex *pa;
     pa = (struct complex*)malloc(sizeof(struct complex));
     (*pa).re = 1.0; (*pa).im = -2.0;
     incr( pa );
     printf("%f+(%fi)\n", (*pa).re, (*pa).im);
     free(pa);
     return 0;
}
```

Note

- In C esiste una notazione più comoda per accedere ai campi di un puntatore a struttura
- Al posto di

  ```
  (*pa).re
  ```

  si può scrivere

  ```
  pa->re
  ```

Allocare strutture con `malloc()`

```
/* struct-malloc.c : allocare strutture con malloc() */
#include <stdio.h>
#include <stdlib.h> /* per usare malloc() */

struct complex {
    double re, im;
};

void incr(struct complex *v)
{
    v->re = v->re + 1.0;
    v->im = v->im + 1.0;
}

int main( void )
{
    struct complex *pa;
    pa = (struct complex*)malloc(sizeof(struct complex));
    pa->re = 1.0; pa->im = -2.0;
    incr( a );
    printf("%f+(%fi)\n", pa->re, pa->im);
    free(pa);
    return 0;
}
```

Varianti di dichiarazione/uso

```
/* varianti-struct.c */
#include <stdio.h>
#include <stdlib.h>

int main( void )
{
     struct complex {
        double re, im;
     } *c1, *c2;

     c1 = (struct complex *)malloc(sizeof(struct complex));
     c1->re=1.0; c1->im=2.0;
     c2 = (struct complex *)malloc(sizeof(struct complex));
     c2->re=2.0; c2->im=-7.0;
     free(c1);
     free(c2);
     return 0;
}
```

Dichiarazione locale di struct complex: il tipo è visibile solo dentro il blocco in cui è dichiarato

Dichiarazione contestuale di variabili (puntatori) struct complex;

Array di strutture

```
/* array-struct.c : Esempio di array di strutture */
#include <stdio.h>
#include <stdlib.h>

struct complex { double re, im; };
#define N 5

int main( void )
{
    struct complex v[N];
    int i;
    for (i=0; i<N; i++) {
        v[i].re = i; v[i].im = i;
        printf("%f+(%fi)\n", v[i].re, v[i].im);
    }
    return 0;
}
```

```
0.000000+(0.000000i)
1.000000+(1.000000i)
2.000000+(2.000000i)
3.000000+(3.000000i)
4.000000+(4.000000i)
```

Array di strutture

- Dichiarazione senza inizializzazione

```
struct complex v[3];
```

- Dichiarazione con inizializzazione

```
struct complex v[3]={ {1.0,1.0}, {1.0,-3.0}, {7.5,3.2} };
```

  oppure omettendo la dimensione dell'array

```
struct complex v[] = { {1.0,1.0}, {1.0,-3.0}, {7.5,3.2} };
```

- Accesso agli elementi

```
v[2].re = 0.2;
```

Esempio

```
/* struct-persona.c */
#include <stdio.h>

struct persona {
     char *nome;
     int eta;
};
struct persona crea(char *n, int a) {
    struct persona p;
    p.nome = n; p.eta = a;
    return p;
}
void stampa(struct persona *p) {
    printf("Nome = %s, Eta' = %d\n", p->nome, p->eta);
}
int main(void)
{
    struct persona p1 = crea("Mario", 50);
    struct persona p2 = crea("Luca",  30);
    stampa(&p1);
    stampa(&p2);
    return 0;
}
```

Strutture con strutture

- È possibile definire strutture i cui membri siano a loro volta strutture

```
/* struct-intervallo.c */
struct orario {
    int hh, mm, ss;
};
struct intervallo {
    struct orario inizio;
    struct orario fine;
};
struct intervallo i1 = { {10,5,0}, {12,20,30} };
struct intervallo i2;
i2.inizio.hh = 10; i2.inizio.mm =  5; i2.inizio.ss =  0;
i2.fine.hh   = 12; i2.fine.mm   = 20; i2.fine.ss   = 30;
```

Copiare strutture

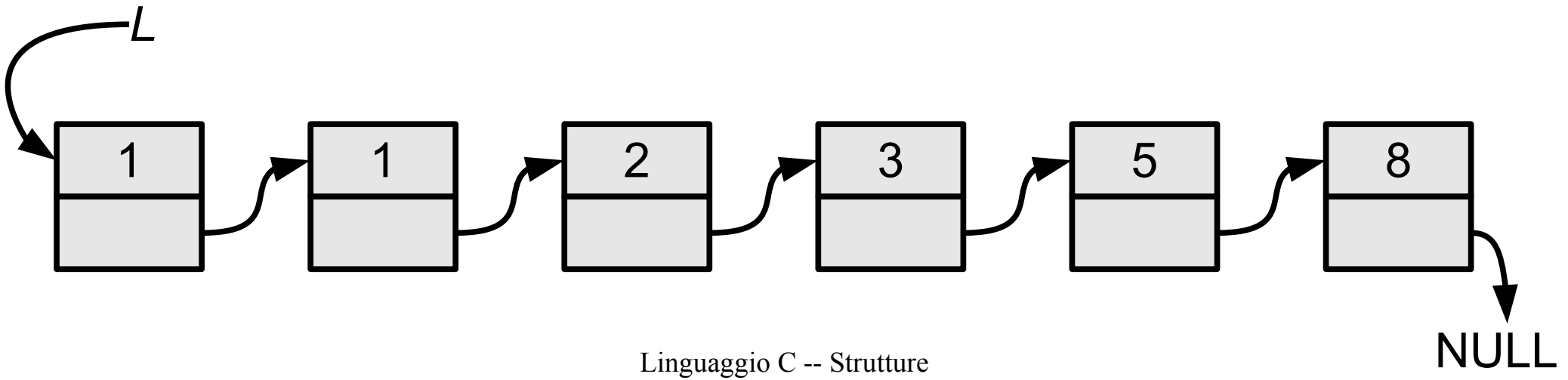
- È possibile utilizzare l'operatore di assegnamento (=) per copiare i valori dei campi di strutture

```
struct orario {
    int hh, mm, ss;
};
struct intervallo {
    struct orario inizio;
    struct orario fine;
};
struct intervallo i1 = { {10,5,0}, {12,20,30} };
struct intervallo i2 = i1; /* ok, i campi di i2 hanno ora
lo stesso valore dei campi di i1 */
```

- Nel caso di campi di tipo puntatore, l'operatore = copia il valore del puntatore, non l'oggetto puntato!

Le Liste

- Una lista è una sequenza (anche vuota) di nodi
- Ogni nodo contiene
  - Una informazione (es., un intero)
  - Un puntatore al nodo successivo, oppure a NULL se non c'è nodo successivo
- Esempio: lista *L* che contiene i valori (1, 1, 2, 3, 5, 8)

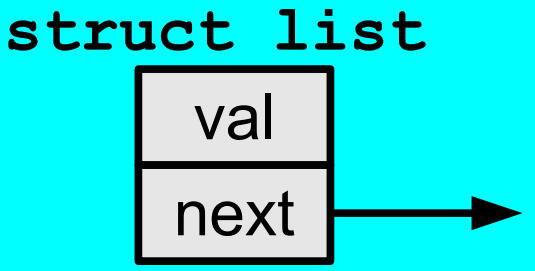


Array vs Liste

Array	Liste
Lunghezza fissa definita all'atto della definizione o allocazione con `malloc()`	Lunghezza variabile: è possibile aggiungere o rimuovere nodi dalla lista a piacimento
È possibile l'accesso diretto all'i-esimo elemento	In generale non è possibile l'accesso diretto all'i-esimo elemento
Non è possibile inserire o cancellare un elemento	È possibile inserire o cancellare nodi dalla lista in qualunque posizione

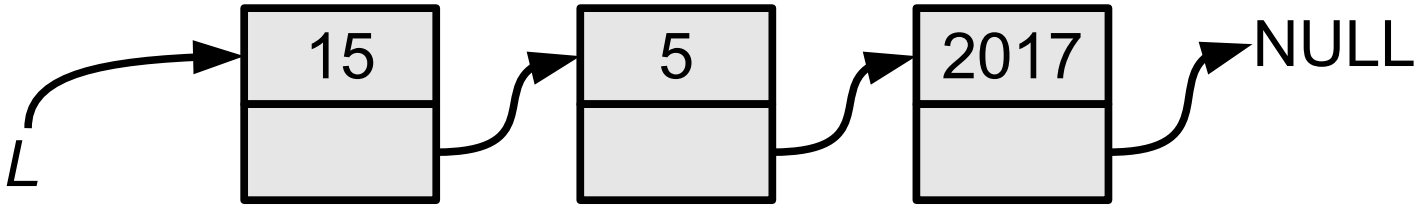
Liste in C

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

struct list {
    int val;
    struct list *next;
};

struct list  *list_create(int v, struct list *n)
{
    struct list *r=(struct list *)malloc(sizeof(struct list));
    r->val = v;
    r->next = n;
    return r;
}
```



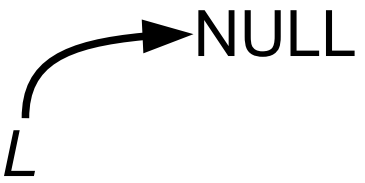
Uso delle liste

- Usando solo la funzione **`list_create()`**, come facciamo a creare una lista *L* come la seguente?



Uso delle liste

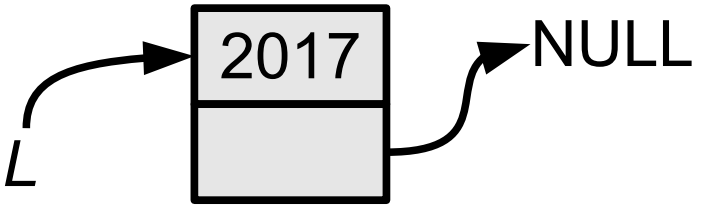
- Usando solo la funzione **`list_create()`**, come facciamo a creare una lista *L* come la seguente?




```
struct list *L = NULL;
L = list_create(2017, L); /* lista 2017 → NULL          */
L = list_create(5, L);    /* lista 5 → 2017 → NULL     */
L = list_create(15, L);   /* lista 15 → 5 → 2017 → NULL */
```

Uso delle liste

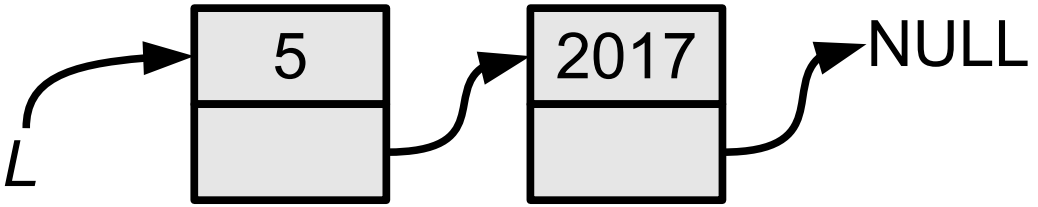
- Usando solo la funzione **`list_create()`**, come facciamo a creare una lista *L* come la seguente?




```
struct list *L = NULL;
L = list_create(2017, L); /* lista 2017 → NULL          */
L = list_create(5, L);    /* lista 5 → 2017 → NULL     */
L = list_create(15, L);   /* lista 15 → 5 → 2017 → NULL */
```

Uso delle liste

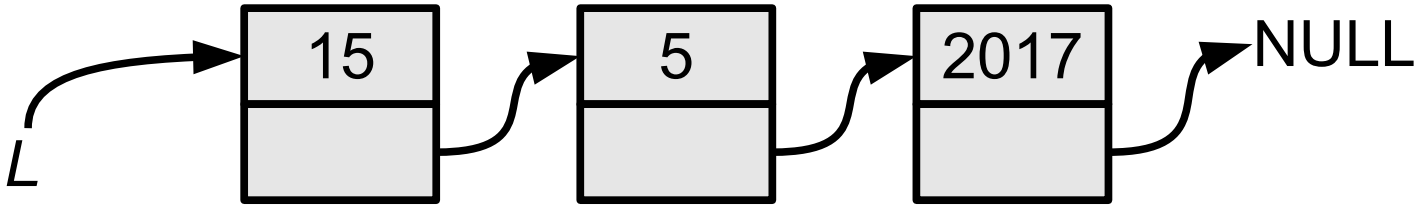
- Usando solo la funzione **`list_create()`**, come facciamo a creare una lista *L* come la seguente?




```
struct list *L = NULL;
L = list_create(2017, L); /* lista 2017 → NULL          */
L = list_create(5, L);    /* lista 5 → 2017 → NULL     */
L = list_create(15, L);   /* lista 15 → 5 → 2017 → NULL */
```

Uso delle liste

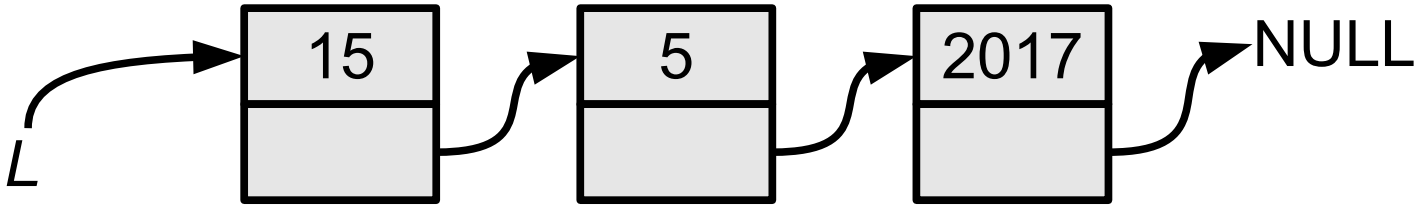
- Usando solo la funzione **`list_create()`**, come facciamo a creare una lista *L* come la seguente?




```
struct list *L = NULL;
L = list_create(2017, L); /* lista 2017 → NULL          */
L = list_create(5, L);    /* lista 5 → 2017 → NULL     */
L = list_create(15, L);   /* lista 15 → 5 → 2017 → NULL */
```

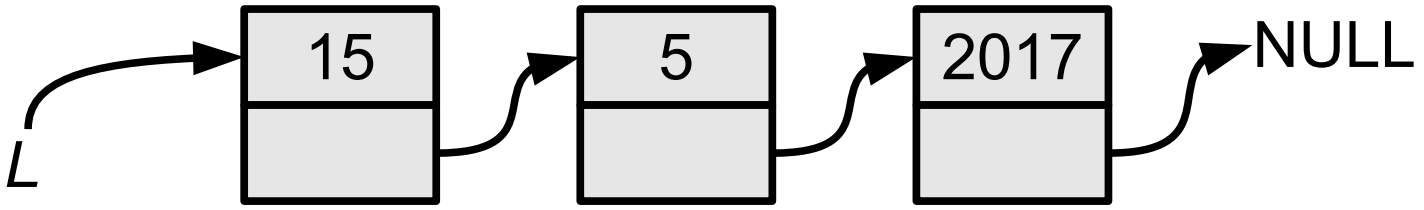
Uso delle liste

- Usando solo la funzione **`list_create()`**, come facciamo a creare una lista *L* come la seguente?



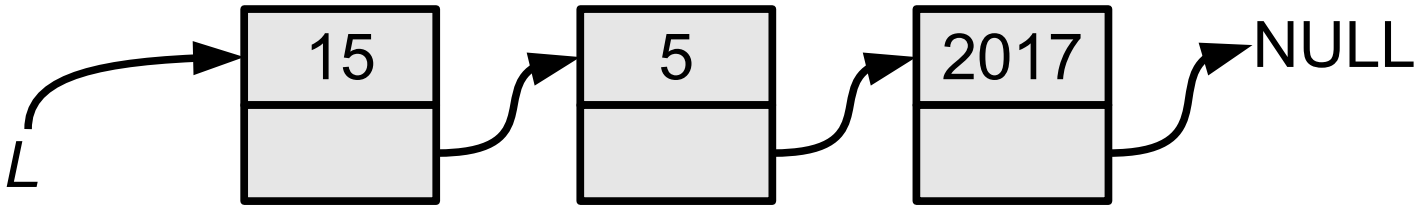

```
struct list *L = list_create(15,
                     list_create(5,
                         list_create(2017, NULL)));
```

Uso delle liste




```
/* Creazione della lista L di elementi (15, 5, 2017) */
struct list *L = NULL;
L = list_create(2017, L);
L = list_create(5, L);
L = list_create(15, L);

/* Stampa il secondo elemento di L */
printf("%d\n", L->next->val);

/* Modifica del terzo elemento in L; diventa (15, 5, 11) */
L->next->next->val = 11;

/* Aggiunta di 25 fra il primo e secondo elemento
   L diventa (15, 25, 5, 2017)  */
L->next = list_create(25, L->next);
```

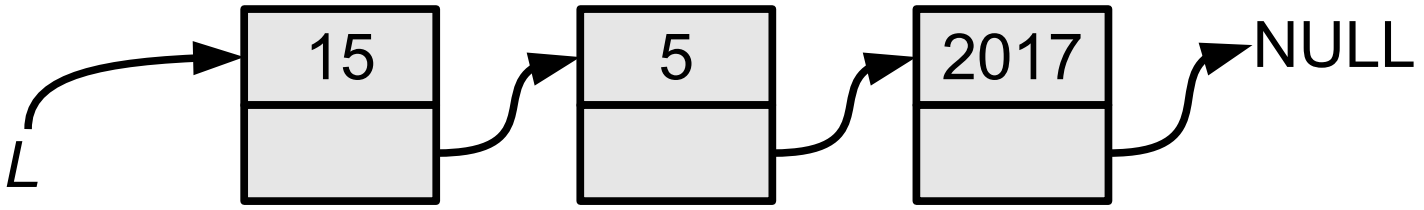
Uso delle liste



- Realizziamo un algoritmo per stampare il contenuto di una lista *L* (anche vuota)
- Iniziamo con uno schema ad alto livello

```
struct list *p = "indirizzo di memoria del primo
nodo di L"
while ("p non è NULL") {
    "stampa il contenuto del nodo puntato da p"
    p = "nodo successivo"
}
```

Uso delle liste



- Realizziamo un algoritmo per stampare il contenuto di una lista *L* (anche vuota)
- Raffinamento

```
struct list *p = L;
while (p != NULL) {
    printf("%d ", p->val);
    p = p->next;
}
```

Altre funzioni che operano su liste

- **`int is_empty(struct list *L)`**
  - ritorna 1 se *L* è la lista vuota (*L* è **`NULL`**), 0 altrimenti
- **`int list_length(struct list *L)`**
  - ritorna la lunghezza (numero di nodi) di *L*; la lista vuota ha lunghezza 0
- **`void list_destroy(struct list *L)`**
  - distrugge (libera la memoria con **`free()`**) tutti i nodi della lista *L*
  - dopo questa operazione il puntatore *L* non deve essere usato perché punta a memoria non più riservata
- **`void list_print(struct list *L)`**
  - Stampa a video il contenuto di L

Altre funzioni che operano su liste

- **`struct list *nth_element(struct list *L, int n)`**
  - restituisce il puntatore all'*n*-esimo nodo della lista (*n*=0 è il primo nodo della lista, *n*=1 è il secondo, ...)
  - restituisce NULL se la lista ha meno di (*n* - 1) nodi
- **`struct list *list_from_array(int v[], int n)`**
  - crea e restituisce il puntatore ad una nuova lista con *n* nodi, in cui il nodo *i*-esimo contiene il valore *v*[*i*] (il nodo 0 è il primo nodo della lista)
- **`struct list *list_concat(struct list *L1, struct list *L2)`**
  - restituisce un puntatore alla lista contenente tutti gli elementi di L1 seguiti dagli elementi di L2
  - dopo questa chiamata, L1 e L2 non andrebbero più usati

Alcune implementazioni

- ...il resto sarà l'oggetto del prossimo laboratorio

```
int list_length(struct list *L)
{
    if ( NULL == L ) {
        return 0;
    } else {
        return (1 + list_length(L->next));
    }
}
```

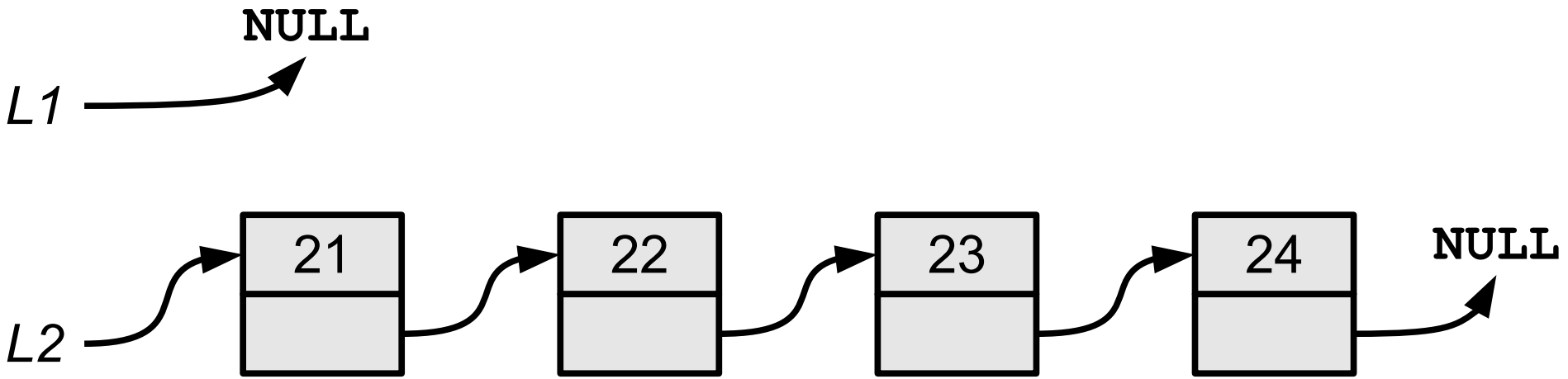
```
void list_destroy(struct list *L)
{
    if ( L != NULL ) {
        list_destroy(L->next);
        L->next = NULL; /* non necessario... */
        free(L);
    }
}
```

Alcuni suggerimenti

- Per molti problemi sulle liste esiste una soluzione ricorsiva "semplice"
  - più semplice della corrispondente soluzione iterativa
- Esempio: **`list_concat(L1, L2)`**
  - Restituisce un puntatore all'inizio della lista costituita da L1 seguita dagli elementi di L2
  - La funzione deve modificare L1 e L2; dopo questa chiamata, L1 e L2 non andranno usate, ma occorre usare solo la lista restituita da **`list_concat()`**

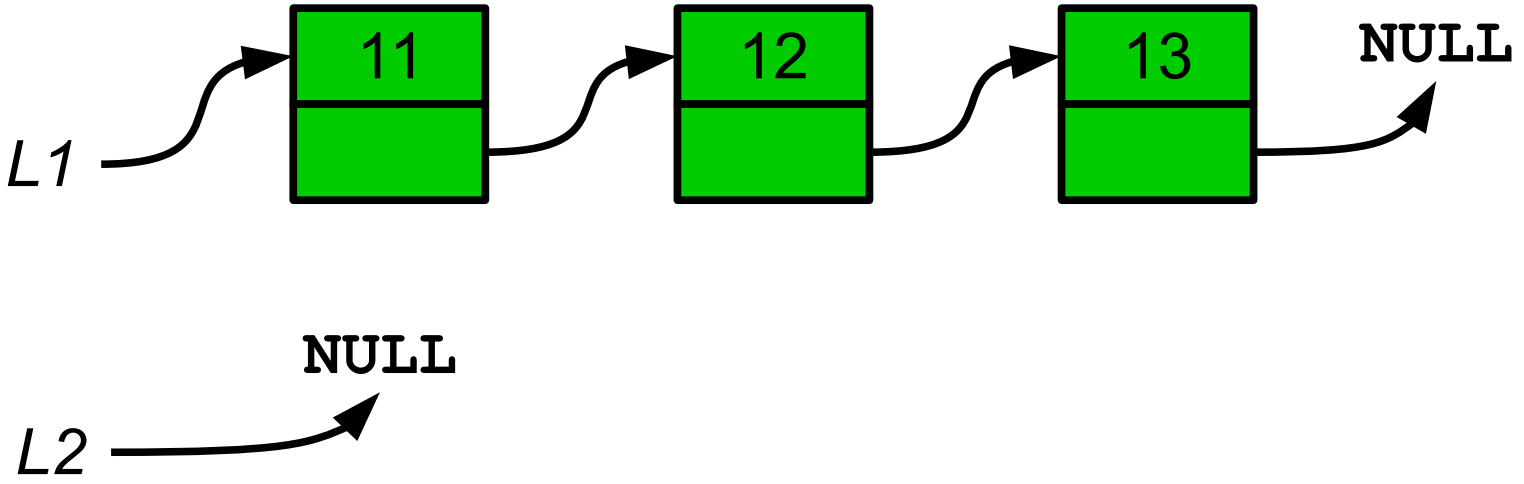
Concatenazione di liste

- **`list_concat(L1, L2)`**
  - Se L1 è la lista vuota, il risultato è L2;



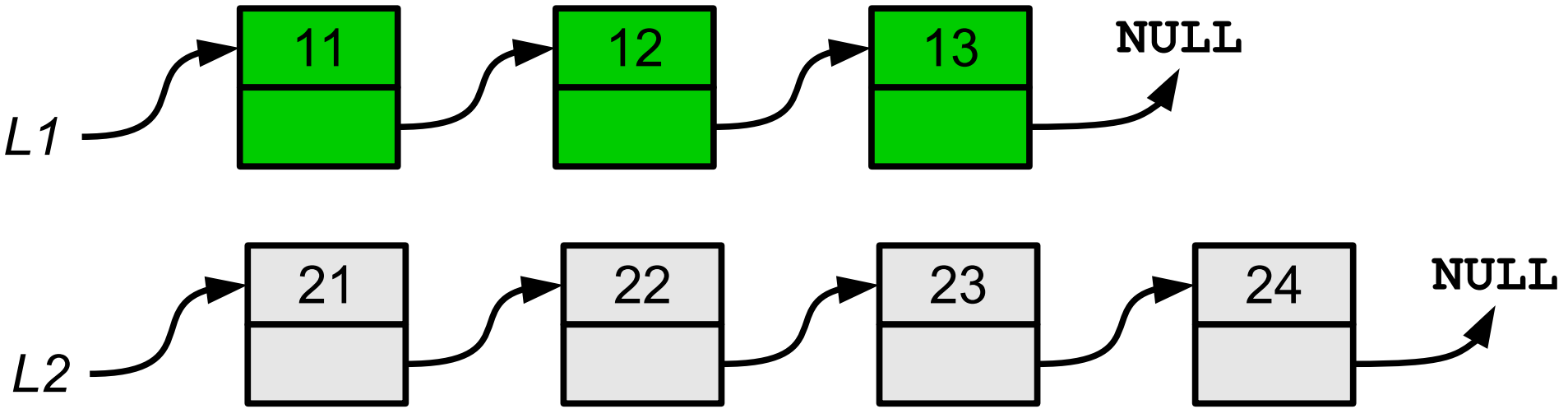
Concatenazione di liste

- **`list_concat(L1, L2)`**
  - Se L1 è la lista vuota, il risultato è L2;
  - Altrimenti, se L2 è la lista vuota, il risultato è L1;



Concatenazione di liste

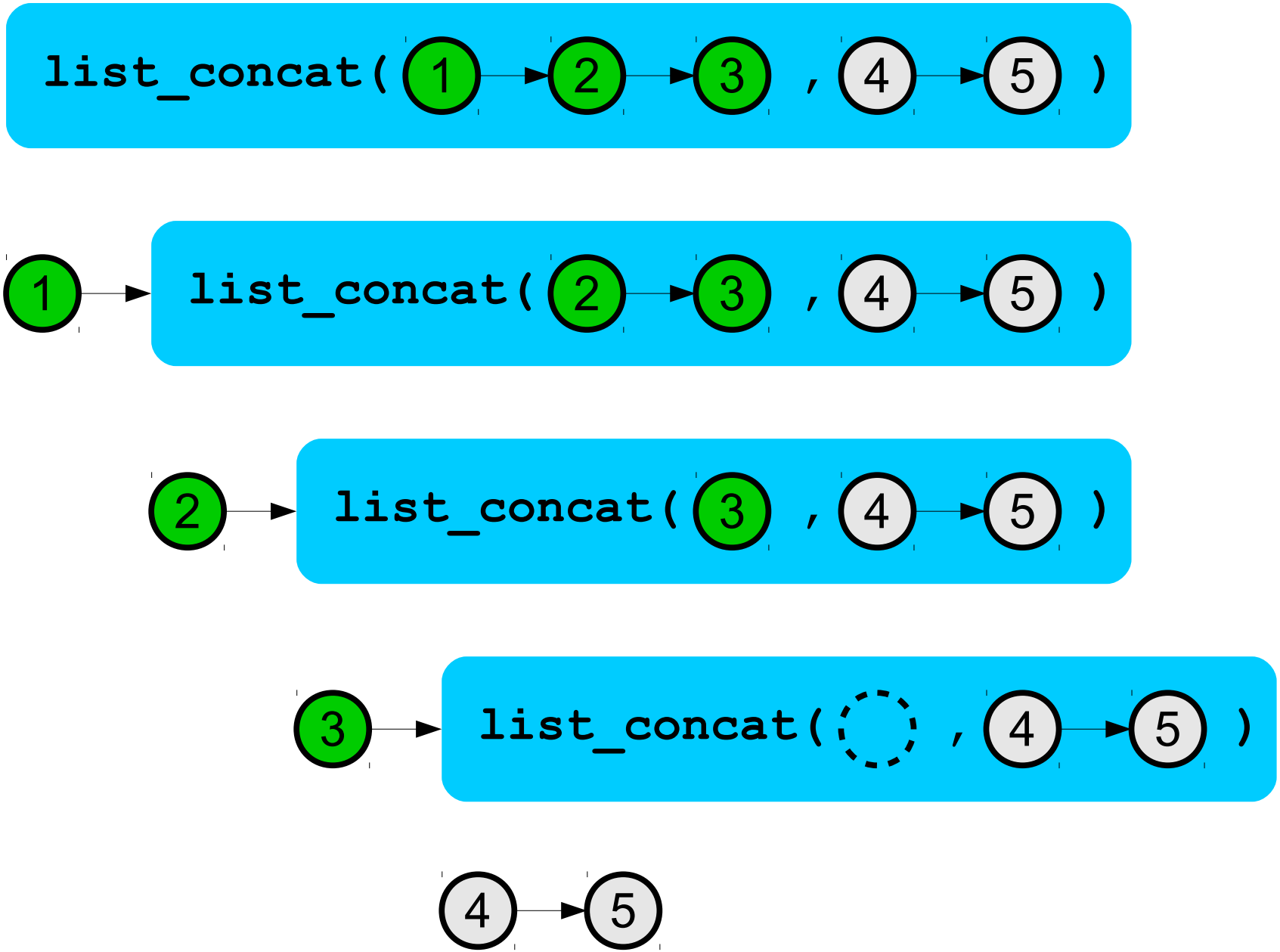
- **`list_concat(L1, L2)`**
  - Se L1 è la lista vuota, il risultato è L2;
  - Altrimenti, se L2 è la lista vuota, il risultato è L1;
  - Altrimenti, il risultato è il primo nodo di L1, seguito dal resto di L1 concatenato con L2

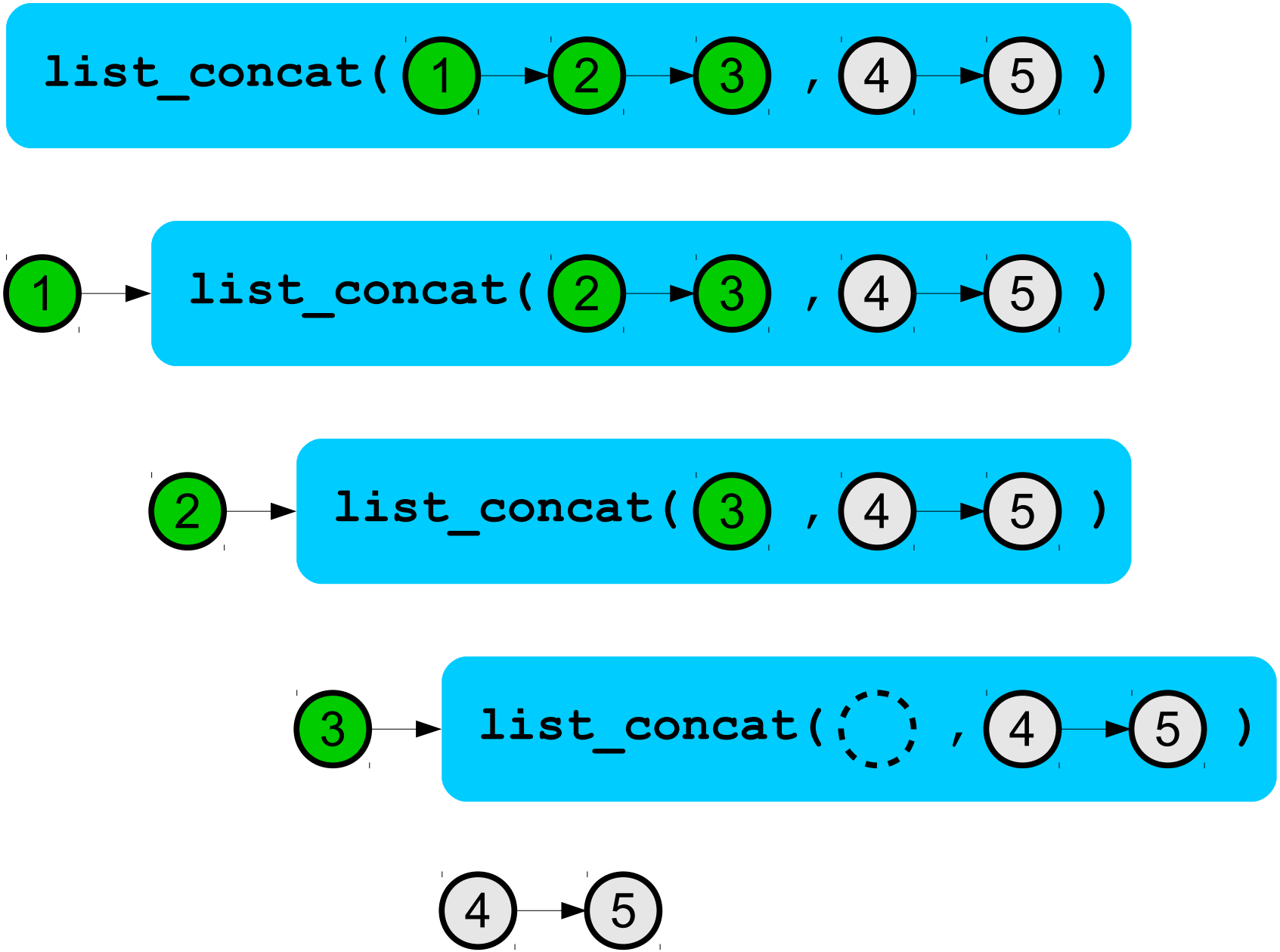


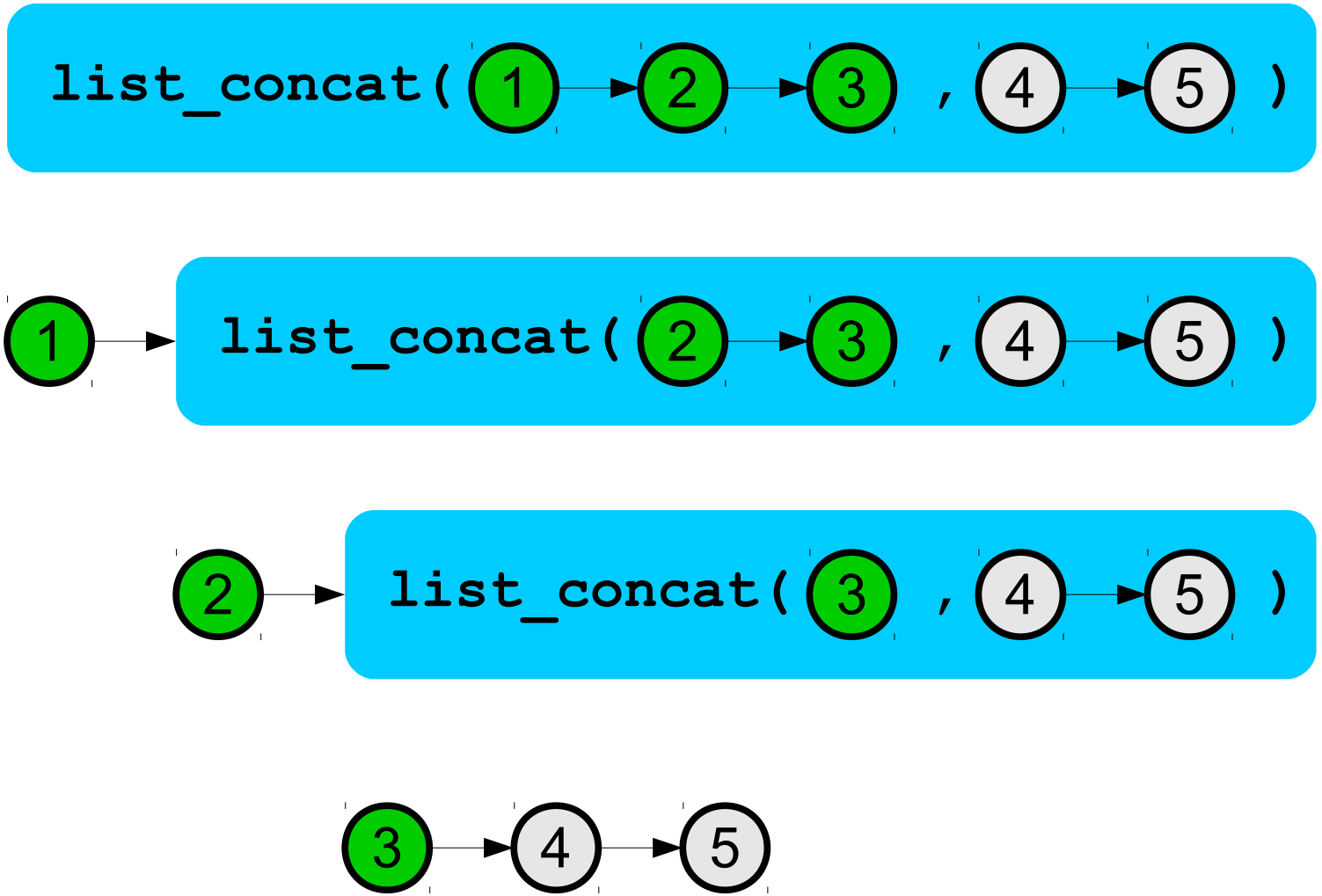
Concatenazione di liste

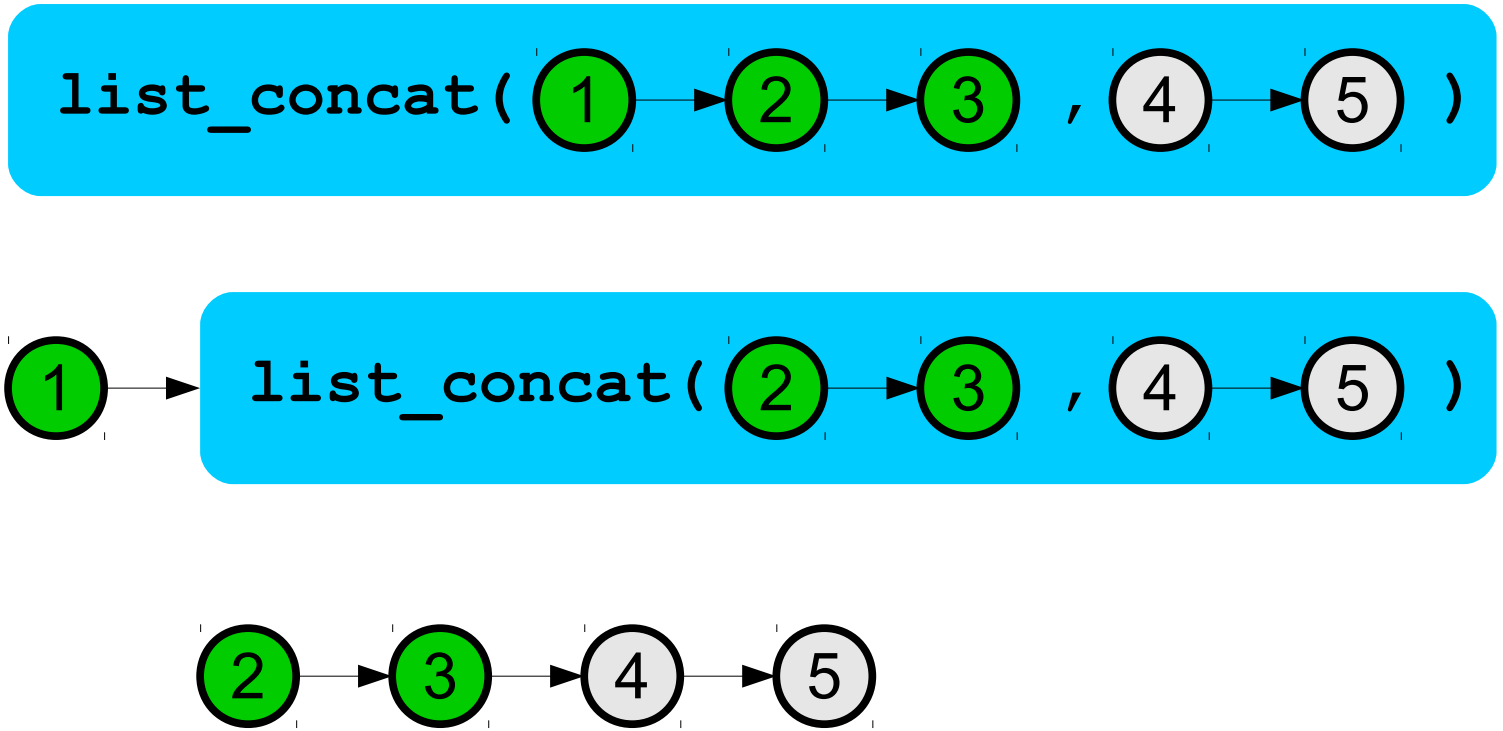
- **`list_concat(L1, L2)`**
  - Se L1 è la lista vuota, il risultato è L2;
  - Altrimenti, se L2 è la lista vuota, il risultato è L1;
  - Altrimenti, il risultato è il primo nodo di L1, seguito dal resto di L1 concatenato con L2

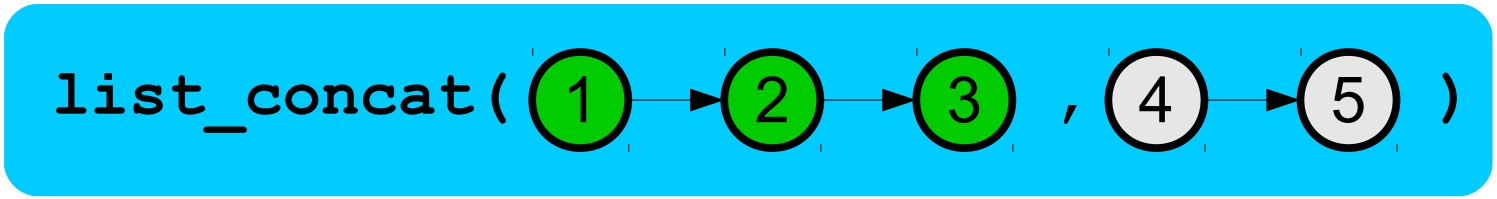
```
struct list* list_concat(struct list *L1, struct list *L2)
{
    if ( NULL == L1 ) {
        return L2;
    } else {
        if ( NULL == L2 ) {
            return L1;
        } else {
            L1->next = list_concat(L1->next, L2);
            return L1;
        }
    }
}
```


list_concat(1 → 2 → 3 , 4 → 5)
1 → list_concat(2 → 3 , 4 → 5)
2 → list_concat(3 , 4 → 5)
3 → list_concat(, 4 → 5)
4 → 5


list_concat(1 → 2 → 3 , 4 → 5)
1 → list_concat(2 → 3 , 4 → 5)
2 → list_concat(3 , 4 → 5)
3 → list_concat(, 4 → 5)
4 → 5


list_concat(1 2 3 , 4 5)
1
list_concat(2 3 , 4 5)
2
list_concat(3 , 4 5)
3 4 5


list_concat(1 2 3 , 4 5)
1
list_concat(2 3 , 4 5)
2 3 4 5


list_concat(1 2 3 , 4 5)


1 2 3 4 5