

# *Tecniche di Programmazione Funzionale e Imperativa*

---

*Ivano Salvo*

*Strutture dati dinamiche in C*  
*Liste*

---

Corso di Laurea in **Informatica**, III anno



**SAPIENZA**  
UNIVERSITÀ DI ROMA

Lezione **23**, 17 maggio 2022

# *Lezione 23a:*

## *Tipi di Dato in C Struct (Record)*

# Costruttori di tipo in C

Il C è un linguaggio piuttosto primitivo, tuttavia permette di costruire nuovi tipo di dato, usando i costruttori \* (puntatore), [] (vettore) e **struct {}** (record).

I record permettono di **aggregare molti dati in uno solo**.

Mentre i vettori sono sequenze di dati omogenei, i record permettono di aggregare più informazioni usualmente di **tipo diverso** in un unico dato strutturato.

L'esempio tipico è l'anagrafica di una persona in un database.

```
struct AP {  
    char nome[15];  
    char cognome[15];  
    data dataNascita;  
    char luogoNascita[15];  
    char cf[14];  
    ...  
};
```

*Tipo definito dall'utente*

*I campi possono essere a loro volta **array** o **struct***

# Tipi di dato: rappresentazione+operazioni

In C è possibile definire nuovi **nomi di tipo** con l'istruzione **typedef** che ha la sintassi:

**typedef** type-definition new-type-name

Ad esempio, il tipo `data` può essere definito come segue:

```
typedef struct D {
    int anno;
    int mese;
    int giorno;
} data;
```

Così definita, il tipo `data` potrebbe sembrare un sottoinsieme di  $\mathbb{Z}^3$ : è bene ricordare che tuttavia ci sono **vincoli di integrità** (il giorno è un numero compreso tra 1 e 31, il mese tra 1 e 12, inoltre...).

Un tipo di dato è caratterizzato dalle **operazioni** (o funzioni) definite su quel dato. Nel caso delle date, operazioni tipiche sono: *verifica della legalità, distanza tra due date, determinare una data tra un certo numero di giorni, etc.*

# Accesso ai campi di un record

Vediamo come esempio, alcune semplici funzioni che manipolano date. Cominciamo con le stampe:

```
void printData(data d){
    printf("%2d %s %4d\n",d.g,m[d.m],d.a);
}

void printDataGGMMAAAA(data d){
    if (d.m<10) printf("%2d/0%1d/%4d\n",d.g,d.m,d.a);
    else printf("%2d/%2d/%4d\n",d.g,d.m,d.a);
}

void printDataGGMMAA(data d){
    if (d.m<10) printf("%2d/0%1d",d.g,d.m);
    else printf("%2d/%2d",d.g,d.m);
    if (d.a%100<10) printf("/0%1d\n",d.a%100);
    else printf("/%2d\n",d.a%100);
}
```

*Dei vettori globali  
possono rendere i  
programmi più  
eleganti*

*Si usa il . (detta  
'dot notation')  
per accedere ai  
campi*

```
char* m[13]={"", "gennaio", "febbraio", "marzo", "aprile", "maggio",
    "giugno", "luglio", "agosto", "settembre", "ottobre", "novembre",
    "dicembre"};
```

```
int dm[13]={0, 31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31};
```

# Struct e puntatori

Più interessante vedere la lettura da input di una data (occorre **controllarne la legalità**)

```
void inserisciData(data *d){
    do {
        printf("\ninserisci il giorno : ");
        scanf("%d",&(d->g));
        printf("\ninserisci il mese : ");
        scanf("%d",&(d->m));
        printf("\ninserisci l'anno : ");
        scanf("%d",&(d->a));
        } while (!legale(*d));
}
```

*La notazione **d->g** significa **(\*d).g** e ne faremo largo uso*

*Uso del vettore delle lunghezze dei mesi (evita if)*

```
int bisestile(int anno){
    if (!(anno%4) && (anno%100 || !(anno%400)))
        return 1;
    return 0;
}

int legale(data d){
    int maxg;

    if (d.a<0 || d.m<1 || d.m>12 || d.g<1)
        return 0;
    if (d.m==feb && bisestile(d.a)) maxg=29;
    else maxg=dm[d.m];
    if (d.g>maxg) return 0;
    return 1;
}
```

# *Lezione 23b:*

## *Liste*

# Liste: visione algebrica

---

Da un punto di vista algebrico, le liste sono definite a partire dai costruttori e sono analoghe quelle **che abbiamo visto in Haskell**, cioè lista vuota `[]` e inserimento in testa `:`.

Di fatto, le liste finite sono il **minimo insieme chiuso** rispetto ai costruttori.

Useremo la notazione Haskell come specifica delle funzioni.

D'altra parte, occorre **trovare una rappresentazione** in C.

Una lista non vuota `x:xs` contiene sempre l'elemento  $x$  (**testa** della lista, head) seguito dalla lista `xs` (**coda** della lista, tail).

È naturale quindi pensare che una lista sia rappresentata in C da una **struct con due campi**.



# Liste: rappresentazione in C

Tuttavia, la definizione:

```
struct L {  
    A x;  
    struct L xs;  
}
```

**non funziona** in quanto, data la ricorsività, il compilatore **non potrebbe determinare quanto spazio riservare per una struct L** (che dovrebbe essere infinito, visto che una parte di una struct L è esso stesso una struct L).

Tuttavia, **un puntatore è sempre un puntatore...**

```
typedef struct L {  
    int val;  
    struct L* next;  
} listanode;  
  
typedef listanode* lista;
```

*Usando l'escamotage dei puntatori, posso determinare la memoria necessaria*

*Osservare che il tipo lista è un tipo pointer*

# Liste: funzioni costruttori in C

Essendo il tipo lista un tipo puntatore, abbiamo una comoda rappresentazione per la **lista vuota** [ ]: il puntatore **NULL**.

Al fine di concentrarci **sulla logica** delle liste (piuttosto che sulla loro rappresentazioni a struct e puntatori), definiamo delle funzioni che implementano i costruttori [ ] e ::.

La funzione `lista_emptyList()` torna semplicemente **NULL**.  
Mentre `::` viene implementato usando una funzione che per motivi storici chiameremo `cons`:

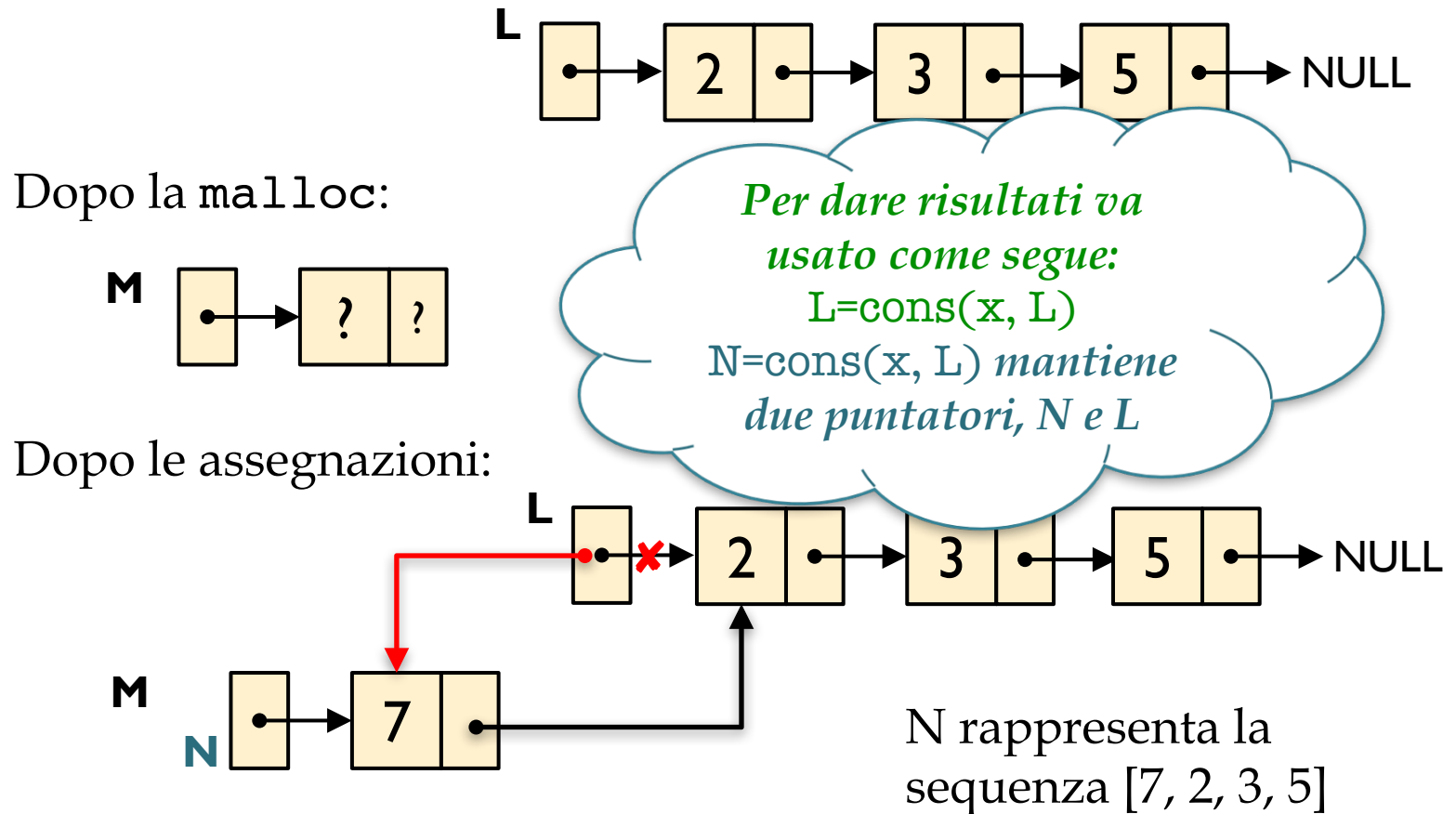
```
lista cons(int x, lista L){
    /* alloco memoria per un nuovo nodo */
    lista M =
        (lista) malloc(sizeof(listanode));
    M->val = x;    /* M = [x] */
    M->next = L;  /* M = x::L */
    return M;
}
```

*Le liste vengono allocate dinamicam.*

*Viene creato un nuovo nodo e ritornato*

# Aspetto logico e memoria

Spesso le liste vengono disegnate come trenini, con dei vagoncini legati dai puntatori: questa rappresentazione centra l'attenzione su cosa effettivamente avviene in memoria. Vediamo l'effetto di una `cons(7, L)` con `L=[2, 3, 5]`:



# Distruttori in C (1)

Un distruttore permette di decomporre un elemento di un insieme induttivo accedendo alle sue componenti: questo serve a definire **proprietà/funzioni** per **induzione/ricorsione**.

Ancora in analogia con i naturali, il distruttore deve distinguere tra 0 e un naturale successore  $n+1$  e in questo caso, accedere a  $n$ .

Nelle liste, il distruttore deve distinguere la lista vuota `[]` da una lista non vuota `x::xs` e nel secondo caso dare accesso alle componenti: l'elemento in testa `x` e la coda `xs`.

```
int isEmpty(lista L){
    /* torna 1 se L=[]
     * 0 altrimenti */
    if (L==NULL) return 1;
    return 0;
    /* anche: return !L */
}
```

```
int head(lista L){
    /* REQ: L<>[]*/
    return L->val;
}
```

```
lista tail(lista L){
    /* REQ: L<>[]*/
    return L->next;
}
```

# *Distruttori in C (2)*

Anche se più spesso ci limiteremo a confrontare puntatori, probabilmente il distruttore **più fedele** in C sarebbe **la funzione** che **contemporaneamente** distingue liste vuote e non vuote e in quest'ultimo caso estrarre testa e coda:

```
int isNotEmpty(lista L, int* x, lista* xs){  
    /* torna 1 se L=[]  
     * 0 altrimenti */  
    if (L){  
        *x = L->val;  
        *xs = L->next;  
        return 1;  
    } else return 0;  
}
```

*Determina se la lista è  
vuota e se non lo è  
carica sui parametri  
testa e coda*

# Uso dei Distruttori in C (1)

Vediamo come possiamo codificare in C alcune funzioni base sulle liste:

```
int length(lista L){
    lista xs;
    int x;
    if (isNotEmpty(L, &x, &xs))
        return 1 + length(xs);
    return 0;
}
```

*purtroppo in C questo necessita del fastidio di definire due variabili*

```
int maxL(lista L){
    /* REQ: L <> [] */
    if (isEmpty(tail(L)))
        return head(L);
    return max(head(L), maxL(tail(L)));
}
```

```
int sum(lista L){
    if (isEmpty(L)) return 0;
    return head(L)+sum(tail(L));
}
```

# Uso dei Distruttori in C (2)

---

Probabilmente un vero Programmatore C preferisce queste versioni:

```
int length(lista L){
    if (!L) return 0;
    return 1 + length(L->next);
}
```

```
int sum(lista L){
    if (!L) return 0;
    return L->val + sum(L->next);
}
```

```
int maxL(lista L){
    /* REQ: L <> [] */
    if (!L->next) return L->val;
    return max(L->val, maxL(L->next));
}
```

# Attenzione a cosa accade in memoria

---

Consideriamo ora la seguente funzione Haskell:

```
twice [] = []  
twice (x:xs) = 2*x:twice xs  
-- twice xs = map (2*) xs
```

A differenza delle funzioni precedenti `twice` deve tornare una lista. **Nel mondo incantato di Haskell esistono solo valori, ma non esiste la memoria.**

Diversa la situazione nel mondo corrotto della computazione imperativa: c'è una memoria e cosa deve fare `twice`?

- creare una nuova lista?
- modificare la lista in ingresso?

Ovviamente non c'è una risposta, dipende da cosa vuol fare il programmatore.

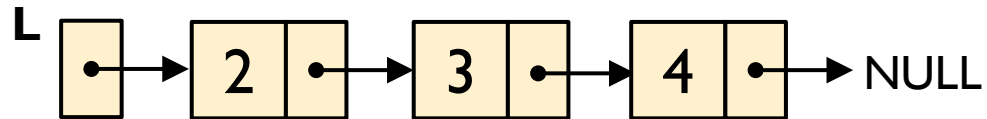
Traducendo le equazioni ricorsive sostituendo `::` con `cons` otteniamo la funzione che genera una nuova lista, perché **cons alloca nuova memoria!**



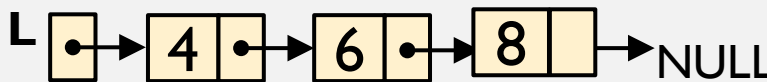
# Attenzione: alla memoria

```
void twiceRec(lista L){  
    if (L){  
        L->val *=2;  
        twiceRec(L->next);  
    }  
}
```

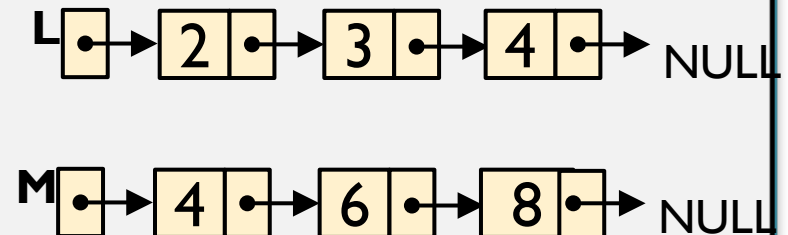
```
lista twiceFun(lista L){  
    if (!L) return L;  
    return cons(2*L->val, twiceFun(L->next));  
}
```



dopo **twiceRec(L)**



dopo **M = twiceFun(L)**



# Attenzione: ai side-effects

I side-effects sono sempre una **risorsa**/**problema** nei linguaggi imperativi, ma diventano particolarmente **sottili e insidiosi** nel caso di memoria concatenata da puntatori. Facciamo un esempio.

La funzione:

```
void tail(lista L){  
    L = L->next;  
}
```

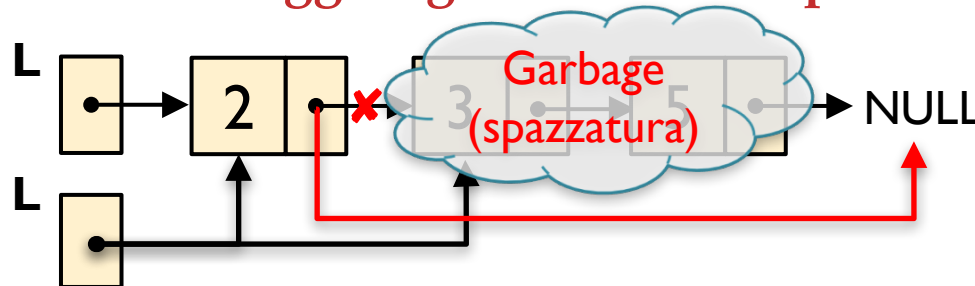
**non fa niente!**

Ma attenzione che:

```
void cutTail(lista L){  
    L->next = NULL;  
}
```

**stacca la coda!**

Infatti, anche se il **pointer di inizio lista è passato per valore**, tutta la **memoria raggiungibile** è di fatto **passata per indirizzo**.



# Attenzione: ai side-effects

Possiamo comunque ottenere la coda della lista in due modi:

La funzione:

```
lista tail(lista L){  
    return L->next;  
}
```

va usata `L = tail(L)!`

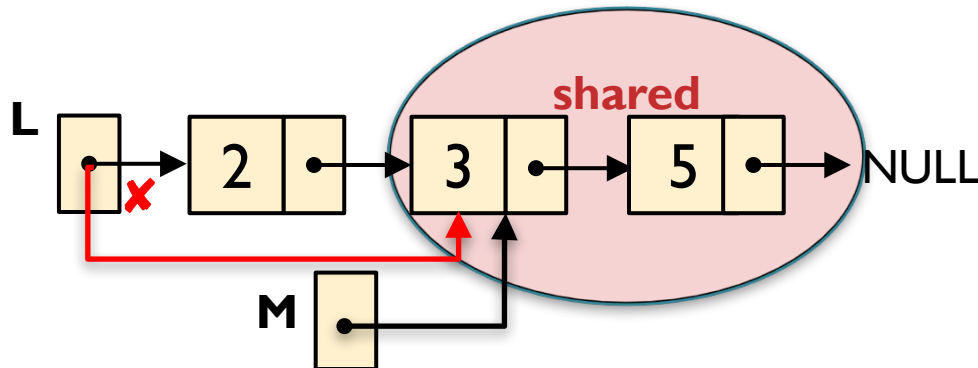
Attenzione che perdetevi il primo nodo!

`M = tail(L)` tiene due pointer alla lista originale (**con uno sharing tra M ed L**).

Oppure:

```
void tailRef(lista *L){  
    *L = *L->next;  
}
```

modifica direttamente L.  
Osservate che L è di tipo `listaNode**`.



# Ovviamente, c'è anche l'iterazione

Vista la natura induttiva delle liste, la **ricorsione**, usualmente si **presta meglio** a scrivere su programmi che manipolano e/o creano liste. Tuttavia, si possono scrivere programmi iterativi equivalenti. **In casi eccezionali, la versione iterativa è più facile.**

```
lista twiceLFunIt(lista L){
  lista lPtr=L;
  lista res=emptyList;
  while (lPtr){
    res = cons(2*head(lPtr), res);
    lPtr = lPtr->next;
  }
  return res;
}
```

```
void twiceLIt(lista L){
  lista lPtr=L;
  while (lPtr){
    lPtr->val *= 2;
    lPtr = lPtr->next;
  }
}
```

Figura 9: Funzione *twiceL* in C (versioni iterativa e ricorsiva)

*Si usa un pointer  
ausiliario per scorrere  
la lista*

# Attenzione tuttavia...

```
lista twiceLFunIt(lista L){
  lista lPtr=L;
  lista res=emptyList;
  while (lPtr){
    res = cons(2*head(lPtr), res);
    lPtr = lPtr->next;
  }
  return res;
}
```

Questa funzione **è errata!** In quanto la lista res risultante **è rovesciata!**

con L = [ 2, 3, 5 ]

res = [ ] (inizializzazione)

res = [ 4 ] (1ma iterazione)

res = [ 6, 4 ] (2da iterazione)

res = [ 10, 6, 4 ] (3za iterazione)

Soluzioni:

- Usare aggiunta in coda invece di cons [**complessità  $O(n^2)$** ]
- Ritornare reverse(res) [**complessità  $O(n)$** ]
- Costruire direttamente il risultato “dritto” [**laborioso**]

# Costruire la lista risultato "dritta"

---

```
int twiceFunIt(lista L){
    /* devo distinguere il caso vuoto */
    if (!L) return L;
    /* creo il primo nodo che va tornato */
    lista res = cons(2*L->val, NULL);
    /* non devo perdere la testa di res */
    lista aux = res;
    L = L->next;
    while (L){
        /* creo il nuovo nodo e lo attacco */
        aux->next = cons(2*L->val, NULL);
        /* avanzo sulle liste */
        L = L->next;
        aux = aux->next;
    }
    return res;
}
```

# *Lezione 23c:*

*Programmi su Liste:  
aggiunta e concatenazione*

# Inserimento di elementi in coda

---

La funzione `cons` implementa il costruttore `:` e inserisce un elemento in testa a una lista. Per aggiungere in coda cominciamo con **specificare le funzioni** con **equazioni ricorsive**:

```
addTail [] y = [y]
addTail (x:xs) y = x : addTail xs y
```

da cui si ricava immediatamente del codice C, traducendo `:` con `cons` e usando i destruttori per accedere alle componenti della lista (nella parte sinistra delle equazioni):

```
lista addTailFun(int x, lista L){
    if (!L) return cons(x, NULL);
    return cons(L->val,
                addTailFun(x, L->next);
}
```

Attenzione che questa funzione, usando ripetutamente `cons`, **alloca una nuova lista con un elemento in più in testa.**

Chiameremo queste funzioni con suffisso `Fun` (perché è il comportamento tipico dei **linguaggi funzionali**)



# Inserimento di elementi in coda

---

Ovviamente possiamo scrivere una funzione che si limita a creare un nuovo nodo. Vediamo la versione ricorsiva:

```
lista addTailRec(int x, lista L){
    /* fa una sola chiamata a cons */
    if (!L) return cons(x, NULL);
    L->next = addTailRec(x, L->next);
    return L;
}
```

Ossevate che l'assegnazione `L->next=addTailRec(L->next, x)` **è sempre inutile tranne che nell'ultimo nodo** (quando `L->next` vale `NULL`), cioè nell'unico caso in cui effettivamente cambia la testa della lista risultato.

Stesso dicasi dell'uso della funzione. **Se sappiamo che L non è la lista vuota**, possiamo chiamare questa funzione semplicemente con `addTailRec(L, x)` in quanto essa in questo caso non modifica il puntatore di inizio lista. Altrimenti, occorre chiamarla riassegnando `L`: `L=addTailRec(L, x)`.

# Inserimento iterativo in coda

Ovviamente possiamo scrivere una funzione iterativa: occorre posizionarsi sull'ultimo nodo (quindi **bisogna fermarsi in tempo!** Non bisogna arrivare al pointer NULL) e inserire il nuovo nodo.

```
int addTailIt(int x, lista L){
    lista lAux = L;
    /* creo il nuovo nodo da inserire */
    lista temp = cons(x, NULL);
    if (lAux){
        /* mi posiziono sull'ultimo nodo */
        while (lAux->next) lAux = lAux->next;
        /* aggancio l'ultimo nodo */
        lAux->next = temp;
        return L;
    } else return temp;
    /* solo qui cambia la testa */
}
```

Scorro la lista,  
evitando di  
perdere la testa

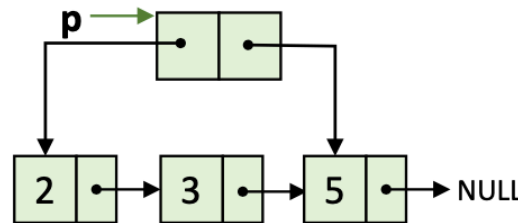
Mi fermo sull'ultimo  
nodo, *sapendo lAux*  
*non vuota*

Se L è vuota, ritorno  
il *nuovo* nodo

# Inserimento in coda in tempo costante

---

Per inserire in tempo costante, posso pensare a una versione evoluta delle liste, in cui una lista contiene due puntatori: uno alla testa e uno alla coda:



Ovviamente, per gestire correttamente questa struttura dati, occorre riprogrammare le funzioni di inserimento in testa e in coda, in modo che i due pointer (diciamo `p->first` e `p->last`) siano **sempre correttamente mantenuti**.

Osservate che in questo caso, **l'operazione "difficile" rimane per esempio la rimozione dell'ultimo elemento**: infatti il puntatore all'ultimo elemento non è sufficiente per risistemare il campo `next` del penultimo elemento a `NULL`.

# Concatenazione di Liste

Cominciamo sempre con la specifica funzionale di una funzione che chiameremo `append`: si tratta di una funzione binaria e facciamo induzione sul primo parametro:

```
append [] ys = ys
append (x:xs) ys = x : append xs ys
```

Al solito, modulo sintassi, possiamo immediatamente tradurre queste equazioni ricorsive in C ottenendo la versione Fun di `append` (o quasi 😊):

```
lista append(lista L, lista M){
    if (!L) return M;
    return cons(L->val, append(L->next, M));
}
```

In effetti, questa funzione **non merita il suffisso Fun**, perché?

Perché **si comporta in modo ibrido** e lascia la seconda lista inalterata. `L=append(L, M)` crea uno **sharing** tra M ed L.

# Concatenazione di Liste

---

Per ottenere un vero comportamento funzionale, occorre ricopiare la lista M:

```
lista appendFun(lista L, lista M){  
    if (!L) return copia(M);  
    return cons(L->val, appendFun(L->next, M));  
}
```

Dove *copia* è la funzione che genera una copia di una lista:

```
lista copia(lista L){  
    if (!L) return L;  
    return cons(L->val, copia(L->next));  
}
```

Nel mondo incantato delle equazioni ricorsive **copy equivale alla funzione identità**, ma nella memoria di un calcolatore essa ha un effetto preciso, **generare una seconda copia di una lista**.

# Concatenazione di Liste

---

Vediamo la versione che non alloca memoria. Usiamo la solita **tecnica di riassegnare il pointer  $L \rightarrow \text{next}$  per trattare in modo uniforme** il caso in cui sono sull'ultimo nodo e quello in cui sono in mezzo alla prima lista:

```
lista appendRec(lista L, lista M){  
    if (!L) return M;  
    L->next = appendRec(L->next, M);  
    return L;  
}
```

Lasciamo per **Esercizio** la versione iterativa: è facilmente derivabile dalla funzione che fa l'aggiunta iterativa in coda in un nuovo elemento.

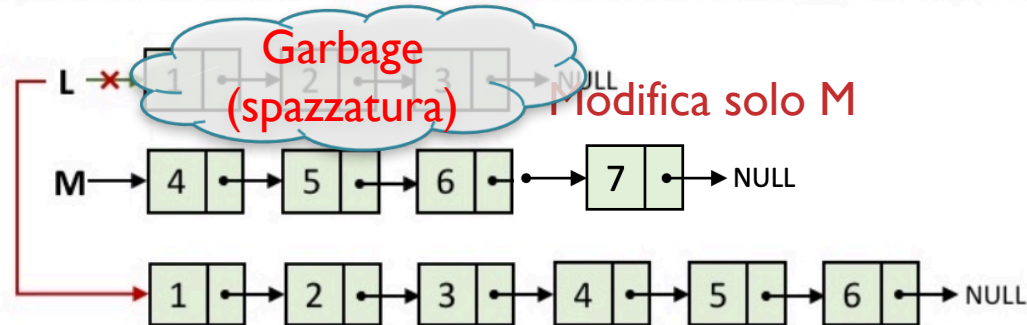
# Concatenazione di Liste

È bene riflettere ancora una volta cosa accade in memoria usando le due funzioni viste:

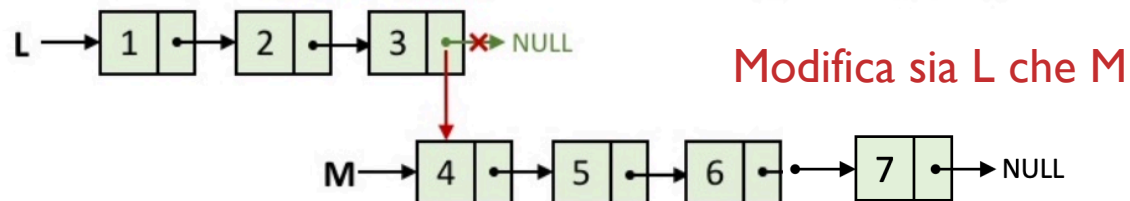
```
L = appendFun(L,M);  
M = addTRec(M,7);
```

```
L = appendRec(L,M);  
M = addTRec(M,7);
```

a) risultato in memoria *prima* e *dopo* l'esecuzione di `L=appendFun(L, M)`



b) risultato in memoria *prima* e *dopo* l'esecuzione di `L=appendRec(L, M)`



# Problemi per casa (beh × casa 🙄)

---

Provate a implementare le strutture dati Pila e Coda.

Fate vedere che per implementare una **pila**, le liste semplici sono abbastanza affinché le operazioni:

```
int pop(stack S);  
void push(stack S, int x);  
int isEmptyStack(stack S);
```

abbiano **complessità costante**.

★Fate vedere che per implementare una **coda**, le liste singolarmente concatenate ma con pointer a inizio e fine lista sono sufficienti affinché le operazioni:

```
int dequeue(queue Q);  
void enqueue(queue Q, int x);  
int isEmptyQueue(queue Q);
```

abbiano **complessità costante**.



# *Lezione 23*

*That's all Folks!*

*Grazie per l'attenzione...*

*...Domande?*