

# PUNTATORI A STRUTTURE

---

È possibile utilizzare i puntatori per accedere a variabili di tipo struct

Ad esempio:

```
typedef struct { int Campo_1, Campo_2;
                } TipoDato;

TipoDato S, *P;
P = &S;
```

Operatore . di selezione campo ha precedenza su \*

=> necessarie parentesi tonde      `(*P).Campo_1=75;`

**Operatore ->** consente l'accesso ad un campo di una struttura referenziata da puntatore in modo più sintetico: `P->Campo_1=75;`

# IL CONCETTO DI LISTA

---

**Lista come sequenza** (*multi-insieme finito e ordinato*) di elementi dello *stesso tipo*

**La lista può contenere in teoria un numero illimitato di oggetti: l'unica limitazione è data dai limiti di memoria del calcolatore. Non possiamo quindi fissare una capienza massima a priori per la lista.**

**Multi-insieme:** insieme in cui un medesimo elemento può comparire più volte

Notazione (astratta, formale, non del C):

$L = [el_1, el_2, \dots, el_N]$

$['a', 'b', 'c']$  denota la lista dei caratteri 'a', 'b', 'c'

$[5, 8, 5, 21, 8]$  denota una lista di 5 interi

$[]$  denota la lista vuota

# ADT LISTA (1)

---

Come ogni **tipo di dato astratto**  $T = \{D, \mathfrak{S}, \Pi\}$  la lista è definita in termini di:

- **dominio** dei suoi elementi (dominio di base)  $D$
- **operazioni** (**costruzione**, **selezione**,...)  $\mathfrak{S}$  e **predicati** sul tipo lista  $\Pi$

Una lista semplice è un tipo di dato astratto tale che:

- $D$  può essere qualunque
- $\mathfrak{S} = \{ \text{cons}, \text{head}, \text{tail}, \text{emptyList} \}$
- $\Pi = \{ \text{empty} \}$

# ADT LISTA (2)

---

ESEMPI:

cons:            D x list -> list            (***costruttore***)  
cons(6, [7,11,21,3,6]) -> [6,7,11,21,3,6]

head:           list -> D            (***selettore “testa”***)  
head([6,7,11,21,3,6]) -> 6

tail:           list -> list            (***selettore “coda”***)  
tail([6,7,11,21,3,6]) -> [7,11,21,3,6]

emptyList:      -> list            (***costruttore “lista vuota”***)

empty:          list -> boolean        (***test di “lista vuota”***)  
empty( [] ) -> true

# ADT LISTA (3)

Pochi linguaggi forniscono il tipo *lista* fra i tipi predefiniti (LISP, Prolog); per gli altri, ***ADT lista si costruisce a partire da altre strutture dati*** (in C tipicamente vettori o puntatori)

## OPERAZIONI PRIMITIVE DA REALIZZARE

| Operazione                       | Descrizione   |
|----------------------------------|---|
| <b>cons: D x list -&gt; list</b> | Costruisce una <i>nuova lista</i> , aggiungendo l'elemento fornito in testa alla lista data |
| <b>head: list -&gt; D</b>        | Restituisce <i>primo elemento</i> della lista data  |
| <b>tail: list -&gt; list</b>     | Restituisce la <i>coda</i> della lista data   |
| <b>emptyList: -&gt; list</b>     | Restituisce la <i>lista vuota</i>   |
| <b>empty: list -&gt; boolean</b> | Restituisce <i>vero</i> se la lista data è vuota, <i>falso</i> altrimenti                   |

# ADT LISTA (4)

---

Concettualmente, le operazioni precedenti costituiscono un ***insieme minimo completo*** per operare sulle liste

Tutte le altre operazioni, *ad es. inserimento (ordinato) di elementi, concatenamento di liste, stampa degli elementi di una lista, rovesciamento di una lista*, si possono ***definire in termini delle primitive precedenti***

***NOTA - Tipo list è definito in modo induttivo:***

- Esiste la costante “lista vuota” (risultato di *emptyList*)
- È fornito un costruttore (*cons*) che, dato un elemento e una lista, produce una nuova lista

Questa caratteristica renderà naturale esprimere le ***operazioni derivate*** (non primitive) mediante **algoritmi ricorsivi**

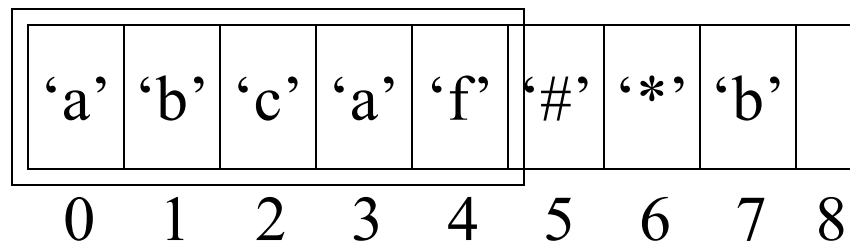
# IMPLEMENTAZIONE CONCRETA DI LISTE

## 1) RAPPRESENTAZIONE STATICA

Utilizzare un **vettore** per memorizzare gli elementi della lista uno dopo l'altro (**rappresentazione sequenziale**)

- **primo** memorizza l'indice del vettore in cui è inserito primo elemento
- **lunghezza** indica da quanti elementi è composta la lista
- Le componenti del vettore con indice pari o successivo a (primo + lunghezza) non sono significative

•ESEMPIO: ['a', 'b', 'c', 'a', 'f']



0 primo

5 lunghezza

# IMPLEMENTAZIONE CONCRETA DI LISTE COME ARRAY

---

## Vantaggi:

- Implementazione molto semplice
- Accesso agli elementi tramite indice molto rapido

## Inconvenienti:

- ***dimensioni fisse*** del vettore: L'array deve essere ridimensionato (all'occorrenza) per non fissare a priori il numero massimo di elementi (gestione dinamica della memoria)
- ***costosi inserimento e cancellazione***  
(con spostamento di un certo numero di elementi)  
copia dell'intera struttura dati)



# IMPLEMENTAZIONE CONCRETA DI LISTE mediante liste concatenate

## 2) RAPPRESENTAZIONE COLLEGATA

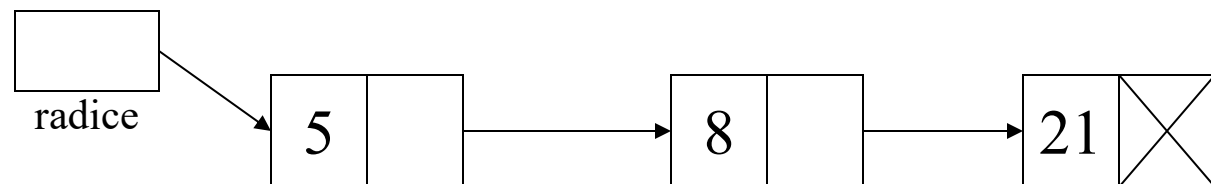
A ogni elemento si associa l'informazione *(indice, riferimento)* che permette di individuare la posizione dell'elemento successivo

=> **Sequenzialità** degli elementi della lista non è più rappresentata mediante **l'adiacenza delle locazioni di memorizzazione**

### NOTAZIONE GRAFICA

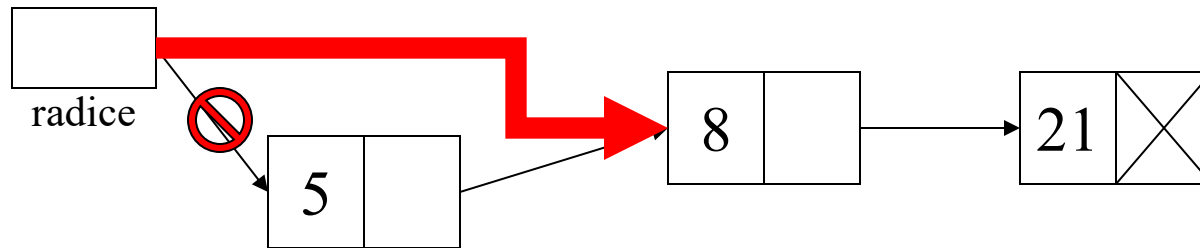
- elementi della lista come **nodi**
- riferimenti (indici) come **archi**

ESEMPIO  
[5, 8, 21]

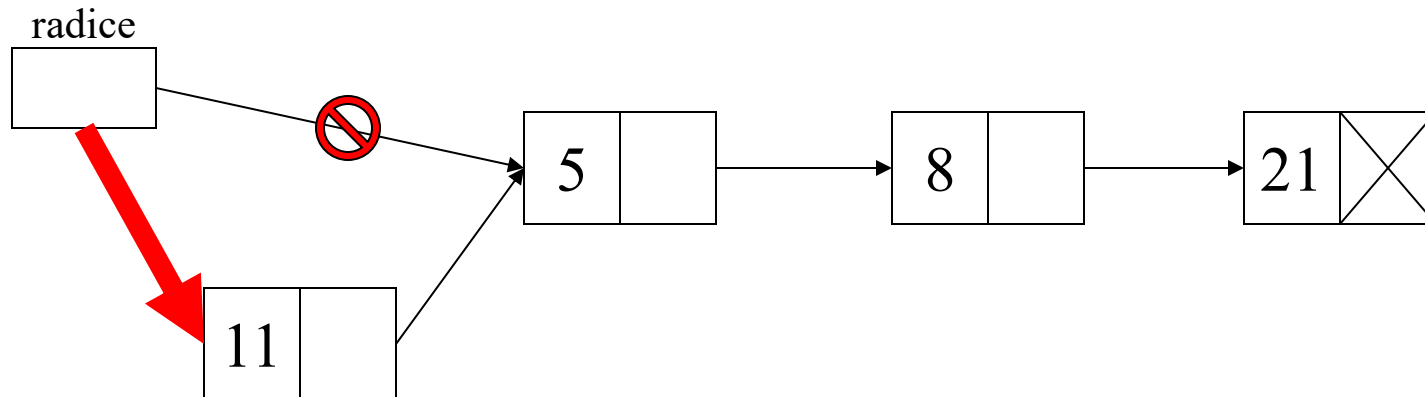


# RAPPRESENTAZIONE COLLEGATA

## ELIMINAZIONE PRIMO ELEMENTO (tail)

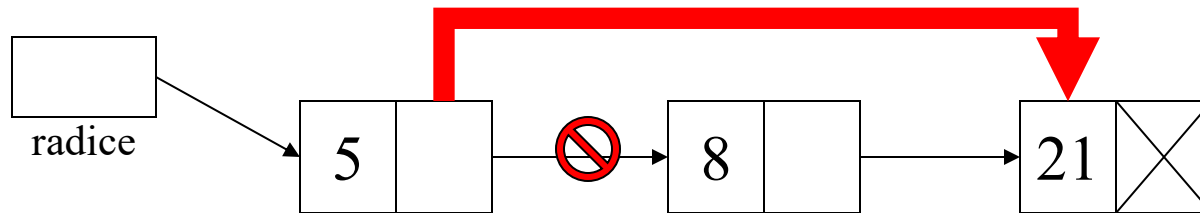


## INSERIMENTO IN TESTA (cons)

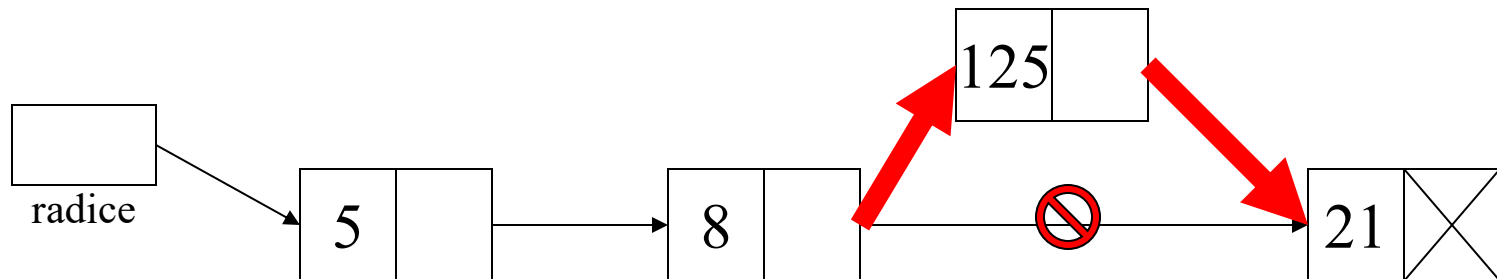


# RAPPRESENTAZIONE COLLEGATA

## ELIMINAZIONE DI UN ELEMENTO



## INSERIMENTO DI UN ELEMENTO



# IMPLEMENTAZIONE CONCRETA DI LISTE COLLEGATE

---

## **Vantaggi:**

- Uno degli approcci più comuni per l'implementazione di liste
- E' una struttura altamente dinamica in cui i nodi della lista non sono necessariamente contigui in memoria.
- L'inserimento o cancellazione degli elementi non comporta la riorganizzazione dell'intera struttura con lo spostamento di elementi ma solo l'aggiornamento di puntatori.

## **Inconvenienti:**

- L'accesso ad un elemento non può avvenire in modo diretto, ma richiede l'attraversamento sequenziale di un certo numero di nodi della lista.

# RAPPRESENTAZIONE COLLEGATA

---

## ***IMPLEMENTAZIONE MEDIANTE PUNTATORI***

Problema della dimensione massima del vettore: adottare un approccio basato su ***allocazione dinamica*** memoria

Ciascun nodo della lista è una struttura di due campi:

- ***valore*** dell'elemento
- un ***puntatore*** nodo successivo lista (NULL se ultimo elemento)

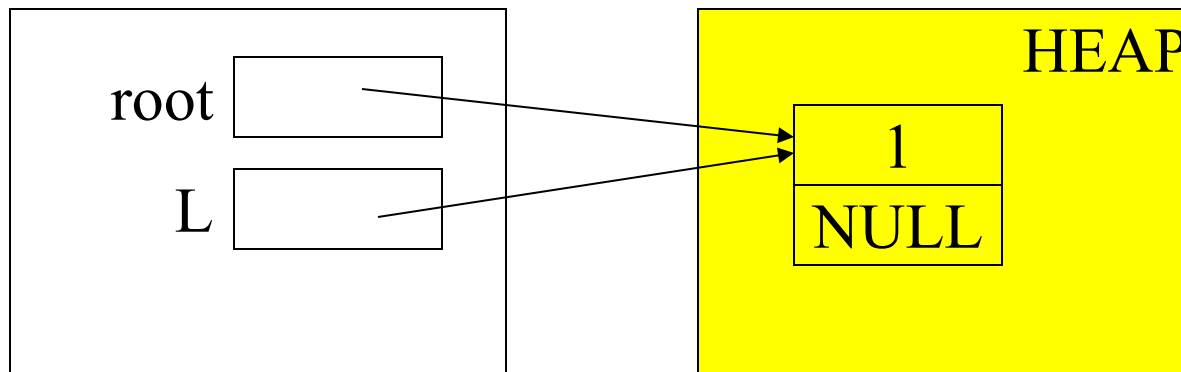
```
typedef struct list_element {  
    int value;  
    struct list_element *next;  
} node;  
typedef node *list;
```

NOTA: ***etichetta list\_element*** in dich. struct è in questo caso ***indispensabile***, altrimenti sarebbe impossibile definire il tipo ricorsivamente

# ESERCIZIO 1

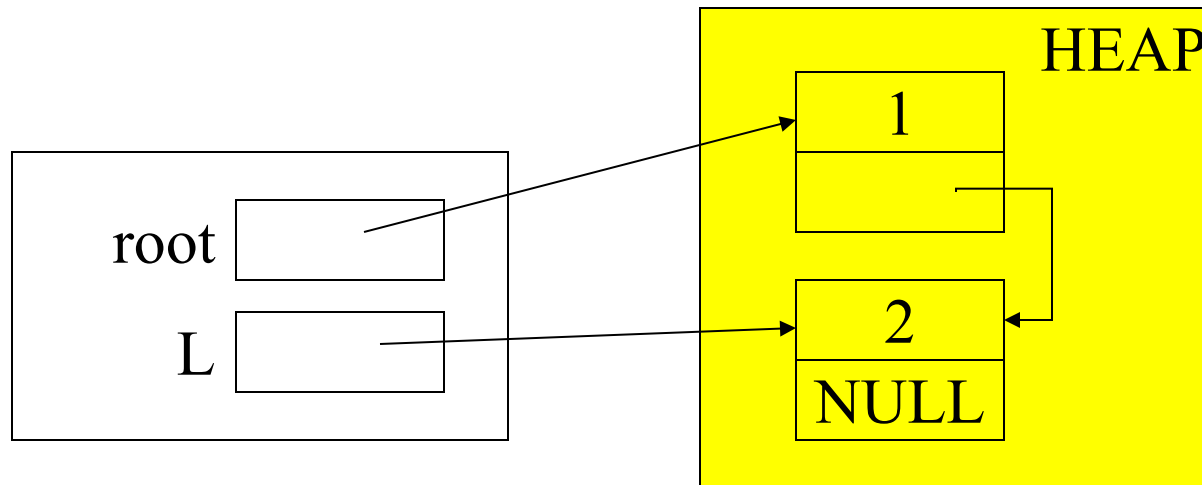
```
#include <stdlib.h>
typedef struct list_element {
    int value;
    struct list_element *next;
} item;
typedef item *list;

int main() {
    list root = NULL, L;
    root = (list) malloc(sizeof(item));
    root->value = 1;
    root->next = NULL;
    L = root;
```



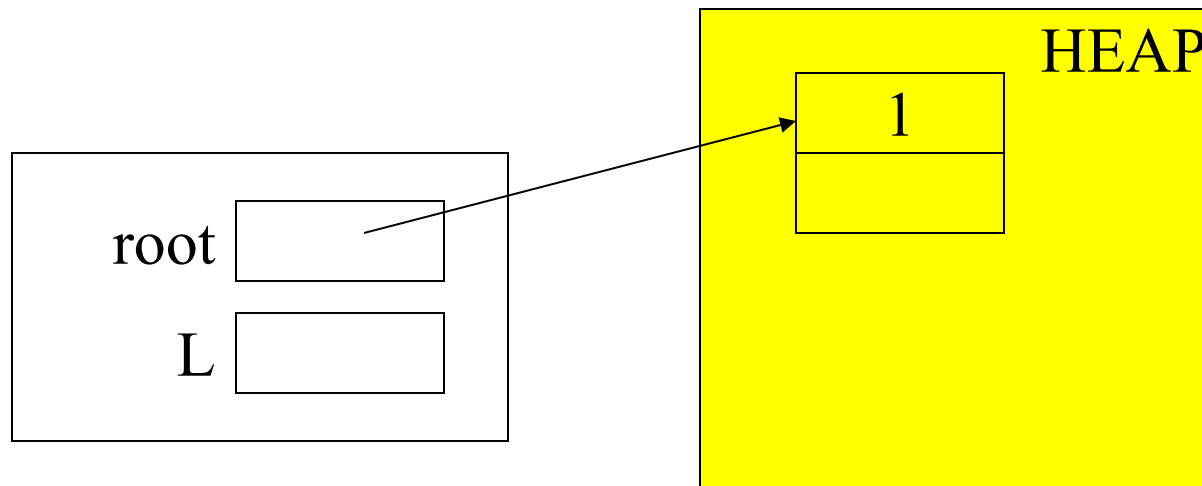
# ESERCIZIO 1 (segue)

```
...  
L = (list) malloc(sizeof(item));  
L->value = 2;  
L->next = NULL;  
root->next = L;
```



# ESERCIZIO 1 (segue)

```
...  
free(L);  
}
```



ATTENZIONE: pericoloso l'utilizzo di `free()` in presenza di ***structure sharing***



## ESERCIZIO 2: creazione di una lista di interi

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

typedef struct list_element {
    int value;
    struct list_element *next;
} item;
typedef item *list;

list insert(int e, list l) { list t;
    t = (list) malloc(sizeof(item));
    t->value = e; t->next = l;
    return t; }

int main() {
    list root = NULL, l; int i;
    do { printf("\nIntrodurre valore: \t");
        scanf("%d", &i);
        root = insert(i, root);
    } while (i!=0);
```

# Creazione di una lista di interi (segue)

---

```
...  
l = root;                                /* stampa */  
while (l!=NULL) {  
    printf("\nValore estratto: \t%d", l->value);  
    l = l->next; }  
}
```

# ESERCIZIO 3: ricerca in una lista

---

```
int ricerca(int e, list l) {  
    int trovato = 0;  
    while ((l!=NULL)&&!trovato)  
        if (l->value == e) trovato = 1;  
        else l = l->next;  
    return trovato;  
}
```

È una ***scansione sequenziale***:

- nel caso ***peggiore***, occorre scandire l'intera lista
- nel caso ***migliore***, è il primo elemento

Esercizio proposto: progettare e implementare  
una soluzione ricorsiva al medesimo problema

# ESERCIZIO 4

---

Definire una funzione **subList()** che, dato un intero positivo *n* e una lista *l*, restituisca una lista che rappresenti ***la sotto-lista inclusa in quella data a partire dall'elemento n-esimo***

ESEMPIO: *l* = [1, 13, 7, 9, 10, 1]                      subList(3, *l*) = [7, 9, 10, 1]

## ***Versione iterativa:***

```
list subList(int n, list l) {  
    int i = 1;  
    while ((l!=NULL) && (i<n)) {  
        l = l->next; i++;  
    }  
    return l; }
```

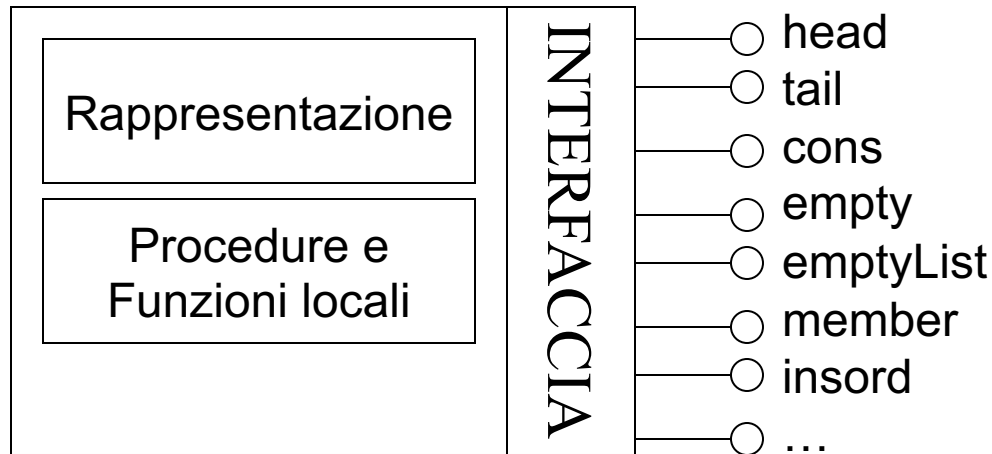
## ***Versione ricorsiva:***

```
list subList(int n, list l) {  
    if ((n == 1) || (l==NULL)) return l;  
    else return (subList(n-1, l->next)); }
```

# COSTRUZIONE ADT LISTA (1)

---

Incapsulare la ***rappresentazione concreta*** (che utilizza puntatori e strutture) e esportare come file header solo ***definizioni di tipo e dichiarazioni delle operazioni***



Funzionamento di lista ***non dipende dal tipo*** degli elementi di cui è composta -> ***soluzione generale***

# COSTRUZIONE ADT LISTA (2)

---

## LINEE GUIDA:

- definire un tipo ***element*** per rappresentare un generico tipo di elemento (con le sue proprietà)
- realizzare ***ADT lista*** in termini di ***element***

## Il tipo *element*

File element.h contiene la definizione di tipo:

```
typedef int element;
```

(il file element.c non è necessario per ora)

Inoltre: `typedef enum { false, true } boolean;`

SCELTA DI PROGETTO: pericoloso uso di free() con structure sharing  
=> NO DEALLOCAZIONE memoria

- ***Inefficiente*** per spazio occupato a meno che non ci sia un garbage collector (Java, Prolog, LISP, ...)
- ***Sicura***, mai effetti collaterali sulle strutture condivise

# ADT LISTA

---

## FILE HEADER (list.h)

```
#include "element.h"

typedef struct list_element {
    element value;
    struct list_element *next;
} item;
typedef item *list;

list emptyList(void);           // PRIMITIVE
boolean empty(list);
element head(list);
list tail(list);
list cons(element, list);

void showList(list);           // NON PRIMITIVE
boolean member(element, list);
...
```

# ADT LISTA: file di implementazione (list.c)

---

```
#include <stdio.h>
#include <stdlib.h>
#include "list.h"          /* ---- PRIMITIVE ---- */

list  emptyList(void)      { return NULL; }

boolean empty(list l) {
    if (l==NULL) return true; else return false; }

element head(list l) {
    if (empty(l)) exit(1);
    else return l->value; }

list tail(list l) {
    if (empty(l)) exit(1);
    else return l->next; }

list cons(element e, list l) {
    list t;
    t = (list) malloc(sizeof(item));
    t->value=e; t->next=l; return t; }
```



# ADT LISTA: file di implementazione (list.c)

---

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

void showList(list l) {                                // NON PRIMITIVE
    printf("[");
    while (!empty(l)) {
        printf("%d", head(l));
        l = tail(l);
        if (!empty(l)) printf(", ");
    } printf("]\n");
}
```

Attenzione!      printf("%d", ...) è specifica per gli interi

# ADT LISTA: il cliente (main.c)

---

```
#include <stdio.h>
#include "list.h"

int main() {
    list l1 = emptyList();
    int el;
    do { printf("\n Introdurre valore:\t");
        scanf("%d", &el);
        l1 = cons(el, l1);
    } while (el!=0);           // condizione arbitraria

    showList(l1);
}
```

# ADT LISTA: altre operazioni non primitive

---

| Operazione                            | Descrizione  |
|---------------------------------------|--|
| <b>member: D x list -&gt; boolean</b> | Restituisce vero o falso a seconda se l'elemento dato è <b>presente</b> nella lista data |
| <b>length: list -&gt; int</b>         | Calcola il <b>numero di elementi</b> della lista data                                    |
| <b>append: list x list -&gt; list</b> | Restituisce una <b>lista</b> che è <b>concatenamento</b> delle due liste date            |
| <b>reverse: list -&gt; list</b>       | Restituisce una <b>lista</b> che è <b>l'inverso</b> della lista data                     |
| <b>copy: list -&gt; list</b>          | Restituisce una <b>lista</b> che è <b>copia</b> della lista data                         |

# ADT LISTA: il predicato member

---

|                               |                  |
|-------------------------------|------------------|
| <b>member</b> (el, l) = falso | se empty(l)      |
| vero                          | se el == head(l) |
| member(el, tail(l))           | altrimenti       |

```
// VERSIONE ITERATIVA
boolean member(element el, list l) {
    while (!empty(l)) {
        if (el == head(l)) return true;
        else l = tail(l);
    } return false;
}
```

```
// VERSIONE RICORSIVA
boolean member(element el, list l) {
    if (empty(l)) return false;
    else if (el == head(l)) return true;
    else return member(el, tail(l));
}
```

# ADT LISTA: la funzione length

---

|                    |                     |             |
|--------------------|---------------------|-------------|
| <b>length(l) =</b> | 0                   | se empty(l) |
|                    | 1 + length(tail(l)) | altrimenti  |

```
// VERSIONE ITERATIVA
int length(list l) {
    int n = 0;
    while (!empty(l)) {
        n++; l = tail(l); }
    return n;
}
```

```
// VERSIONE RICORSIVA
int length(list l) {
    if (empty(l)) return 0;
    else return 1 + length(tail(l));
}
```

# ADT LISTA: la funzione append (1)

---

***append*** (come copy e reverse) non è solo un'operazione di ***analisi*** del contenuto o della struttura della lista, ma implica la ***costruzione di una nuova lista***

Per ottenere una lista che sia il concatenamento di due liste l1 e l2:

- se la lista l1 è vuota, ***il risultato è l2***
- altrimenti occorre ***prendere l1 e aggiungerle in coda la lista l2***

PROBLEMA: come aggiungere una lista in coda a un'altra?

***Nelle primitive non esistono operatori di modifica***

-> l'unico modo è costruire una lista nuova

- con primo elemento (testa), la ***testa della lista l1***
- come coda, una ***nuova lista*** ottenuta ***appendendo l2*** alla coda di l1

=> Serve una ***chiamata ricorsiva*** a append

# ADT LISTA: la funzione append (2)

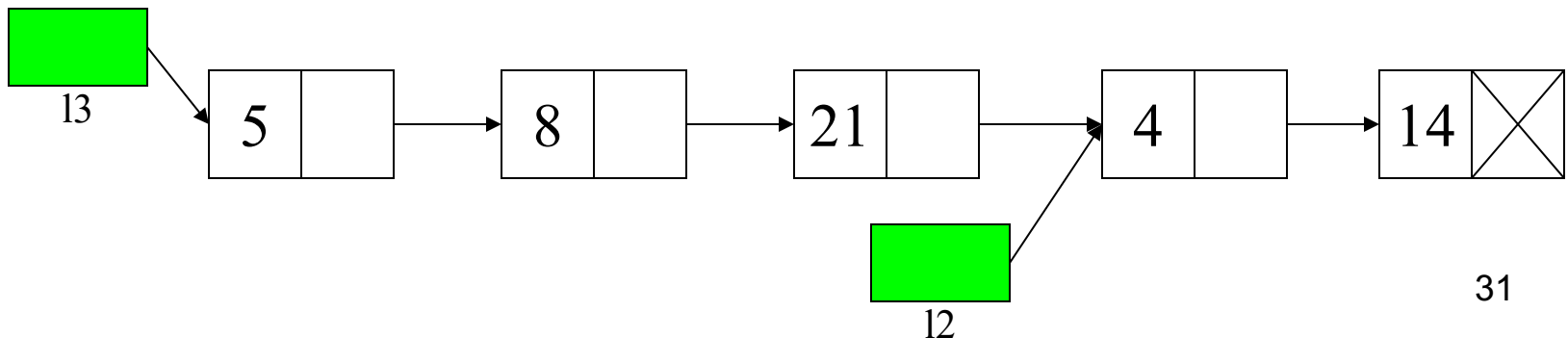
**append(l1, l2) =**  $\begin{matrix} l2 & \text{se empty}(l1) \\ \text{cons}(\text{head}(l1), \text{append}(\text{tail}(l1), l2)) & \text{altrimenti} \end{matrix}$

```
list append(list l1, list l2) {  
    if (empty(l1)) return l2;  
    else return cons(head(l1), append(tail(l1), l2));  
}
```

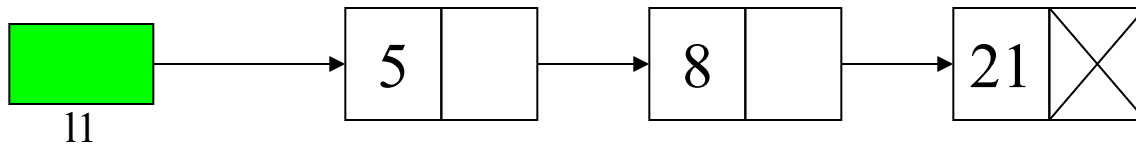
NOTA: quando l1 diventa vuota, **append** restituisce direttamente l2, **non una sua copia** -> l1 è duplicata, ma **l2 rimane condivisa**

## Structure sharing (parziale)

l1=[5,8,21]      l2=[4,14]      l3=append(l1,l2)=[5,8,21,4,14]



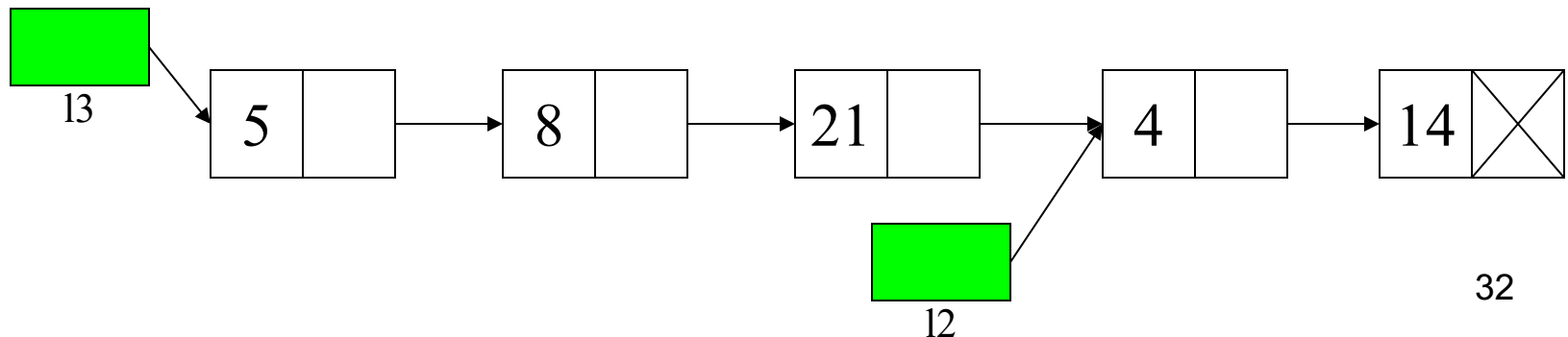
# ADT LISTA: la funzione append (2)



NOTA: quando  $l1$  diventa vuota, ***append*** restituisce direttamente  $l2$ ,  
***non una sua copia*** ->  $l1$  è duplicata, ma ***l2 rimane condivisa***

## ***Structure sharing (parziale)***

$l1 = [5, 8, 21]$        $l2 = [4, 14]$        $l3 = \text{append}(l1, l2) = [5, 8, 21, 4, 14]$





# ADT LISTA: la funzione reverse

---

Per ottenere una lista **rovesciamento** di una lista data *l*, occorre **costruire una nuova lista**, avente:

1. davanti, il risultato del ribaltamento della coda di *l*
2. in fondo, l'elemento iniziale (testa) di *l*

Occorre dunque concatenare la lista ottenuta al punto 1) con l'elemento definito al punto 2) => uso di **append**

append richiede **due liste** => occorre prima costruire una lista *l2* contenente il solo elemento di cui al punto 2)

|   |                    |             |
|---|--------------------|-------------|
| <b>reverse(l) =</b>   | <b>emptyList()</b> | se empty(l) |
| <b>append(reverse(tail(l)), cons(head(l), emptyList()))</b> |                    | altrimenti  |

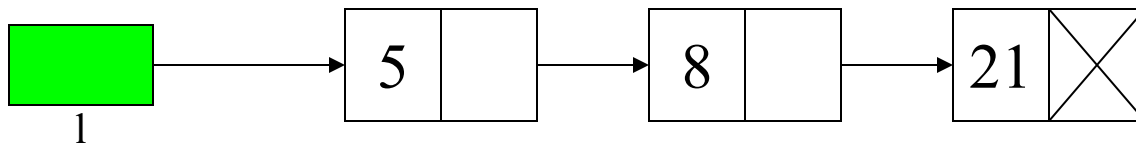
```
list reverse(list l) {  
    if (empty(l)) return emptyList();  
    else return append(reverse(tail(l)),  
                        cons(head(l), emptyList()));  
}
```

# ADT LISTA: la funzione copy

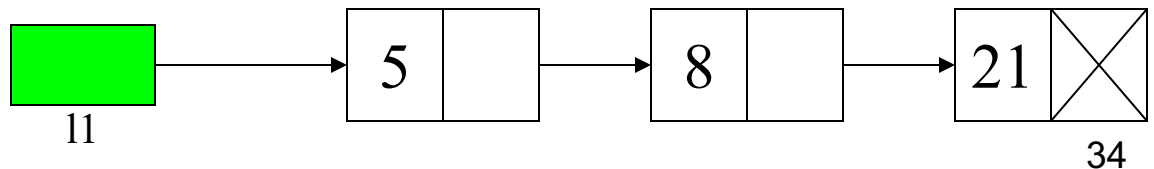
Dato il tipo di operazione, **non** può esservi **condivisione** di **strutture**

Si tratta quindi di impostare un ciclo (o una funzione ricorsiva) che duplichi uno a uno tutti gli elementi

```
list copy(list l) {  
    if (empty(l)) return l;  
    else return cons(head(l), copy(tail(l)));  
}
```



l1 = copy(l)



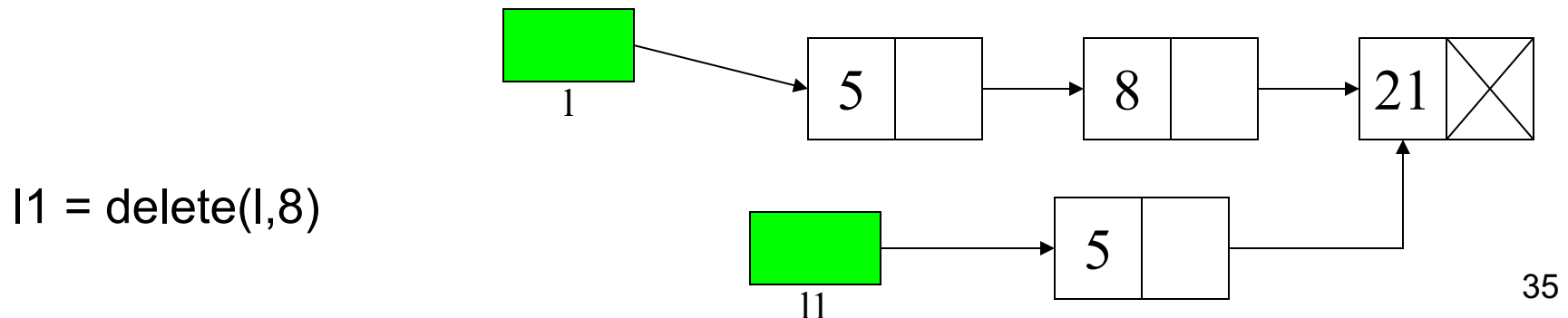
# ADT LISTA: la funzione delete (1)

Deve restituire una lista che differisce da quella data solo per ***l'assenza dell'elemento*** indicato

Non esistendo operatori di modifica, delete deve operare **costruendo una nuova lista** (almeno per la parte da modificare). Occorre:

- duplicare la parte iniziale lista, fino all'elemento da eliminare (escluso)
- agganciare la lista così creata al resto della lista data

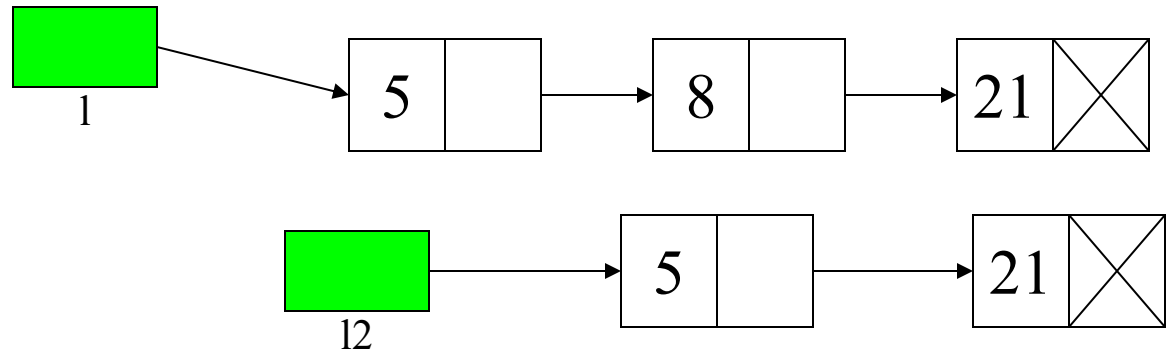
```
list delete(element el, list l) {  
    if (empty(l)) return emptyList();  
    else if (el == head(l)) return tail(l);  
    else return cons(head(l), delete(el, tail(l)));  
}
```



# ADT LISTA: la funzione delete (2)

Per non avere condivisione:

L2 = delete (l, 8)



```
list delete(element el, list l) {  
    if (empty(l)) return emptyList();  
    else if (el == head(l)) return copy(tail(l));  
    else return cons(head(l), delete(el, tail(l)));  
}
```

## CONCLUSIONE

Per usare in modo sicuro la condivisione di strutture, è necessario:

- **NON effettuare free()** -> uso inefficiente heap in linguaggi privi di garbage collection (evitare rischio di riferimenti pendenti)
- **realizzare liste come entità non modificabili**: ogni modifica comporta la creazione di nuova lista (evitare rischio di effetti collaterali indesiderati)

# Deallocazione della memoria per “cancellare” una lista

```
void freelist(list l)
{ if (empty(l)) return;
  else
  {   freelist(tail(l));
      free(l); }
  return; }
```

Non semplicemente `free(l)`

# LISTE ORDINATE

---

Necessario che sia definita una ***relazione d'ordine*** sul ***dominio-base*** degli elementi della lista

NOTA: criterio di ordinamento dipende da ***dominio base*** e dalla specifica ***necessità applicativa***

Ad esempio:

- interi ordinati in senso crescente, decrescente, ...
- stringhe ordinate in ordine alfabetico, in base alla loro lunghezza, ...
- persone ordinate in base all'ordinamento alfabetico del loro cognome, all'età, al codice fiscale, ...

Ad esempio, per costruire una ***lista ordinata di interi*** letti da console si può (*funzione ricorsiva*):

```
list inputordlist(int n) { element e;  
    if (n<0) exit(1);  
    else if (n == 0) return emptyList();  
        else { scanf("%d", &e);  
                return insord(e, inputordlist(n-1));  
        } }
```

# LISTE ORDINATE: la funzione insord

---

Per inserire un elemento in modo ordinato in una lista supposta ordinata:

- se la lista è vuota, costruire una **nuova lista** contenente il nuovo elemento, *altrimenti*
- se l'elemento da inserire è minore della testa della lista, aggiungere il **nuovo elemento in testa** alla lista data, *altrimenti*
- l'elemento andrà **inserito in modo ordinato nella coda** della lista data

I primi due casi sono operazioni elementari

Il terzo caso riconduce il problema allo **stesso problema in un caso più semplice**: alla fine si potrà effettuare o un inserimento in testa o ci si ricondurrà alla lista vuota

```
list insord(element el, list l) {  
    if (empty(l)) return cons(el, l);  
    else if (el <= head(l)) return cons(el, l);  
    else <inserisci el nella coda di l, e restituisci la lista così modificata>  
}
```

# LISTE ORDINATE: la funzione insord

---

Non esistendo primitive di modifica, il solo modo per ottenere una lista diversa è **(ri)costruirla**

Dunque, per inserire un elemento nella coda della lista data occorre **costruire una nuova lista** avente:

- come primo elemento (testa), la **testa della lista data**
- come coda, **coda modificata** (con inserimento del nuovo elemento)

```
list insord(element el, list l) {  
    if (empty(l)) return cons(el, l);  
    else if (el <= head(l)) return cons(el, l);  
        else return cons(head(l), insord(el, tail(l)));  
}
```

- tutta la parte iniziale della lista viene **duplicata**
- parte successiva al punto inserimento è invece **condivisa**



# IL PROBLEMA DELLA GENERICITÀ

---

Funzionamento lista ***non deve dipendere dal tipo degli elementi*** di cui è composta => cercare di costruire ADT generico che funzioni con ***qualsunque tipo di elementi***

=> ADT ausiliario ***element*** e realizzazione dell'ADT lista in termini di element

Osservazioni:

- ***showList*** dipende da printf() che svela il tipo dell'elemento
- ***insord*** dipende dal tipo dell'elemento nel momento del confronto

Può quindi essere utile ***generalizzare queste necessità***, e definire un ADT element che fornisca funzioni per:

- verificare ***relazione d'ordine*** fra due elementi
- verificare ***l'uguaglianza*** fra due elementi
- leggere da ***input*** un elemento
- scrivere su ***output*** un elemento

# ADT ELEMENT: element.h

---

Header element.h deve contenere

- **definizione** del tipo element (e boolean)
- **dichiarazioni** delle varie funzioni fornite

```
typedef int element;      //DEFINIZIONE  
  
boolean isLess(element, element);  
boolean isEqual(element, element);  
element getElement(void);  
void printElement(element);
```

# ADT ELEMENT: element.c

---

```
#include "element.h"
#include <stdio.h>

boolean isEqual(element e1, element e2) {
    return (e1==e2); //non restituisce esattam. true/false}

boolean isLess(element e1, element e2) {
    return (e1<e2); // come sopra }

element getElement() {
    element e1;
    scanf("%d", &e1);
    return e1; }

void printElement() {
    printf("%d", e1); }
```

# ESERCIZIO

Realizzare (utilizzando le funzioni primitive definite nell'ADT) funzione ***mergeList()*** che fonda l1 e l2 ordinate in l3 senza ripetizioni

*Versione ricorsiva*

```
list mergeList(list l1, list l2) {
    if (empty(l1)) return l2;
    else if (empty(l2)) return l1;
    else { if (isLess(head(l1), head(l2)))
            return cons(head(l1), mergeList(tail(l1), l2));
          else if (isEqual(head(l1), head(l2)))
            return cons(head(l1), mergeList(tail(l1), tail(l2)));
          else return cons(head(l2), mergeList(l1, tail(l2)));
        } }
}
```

Esercizio 1: realizzare una ***versione iterativa di mergeList()***

Esercizio 2: si leggano da terminale alcuni ***nomi***, li si inseriscano in una ***lista ordinata alfabeticamente***, e si stampi la lista così ottenuta

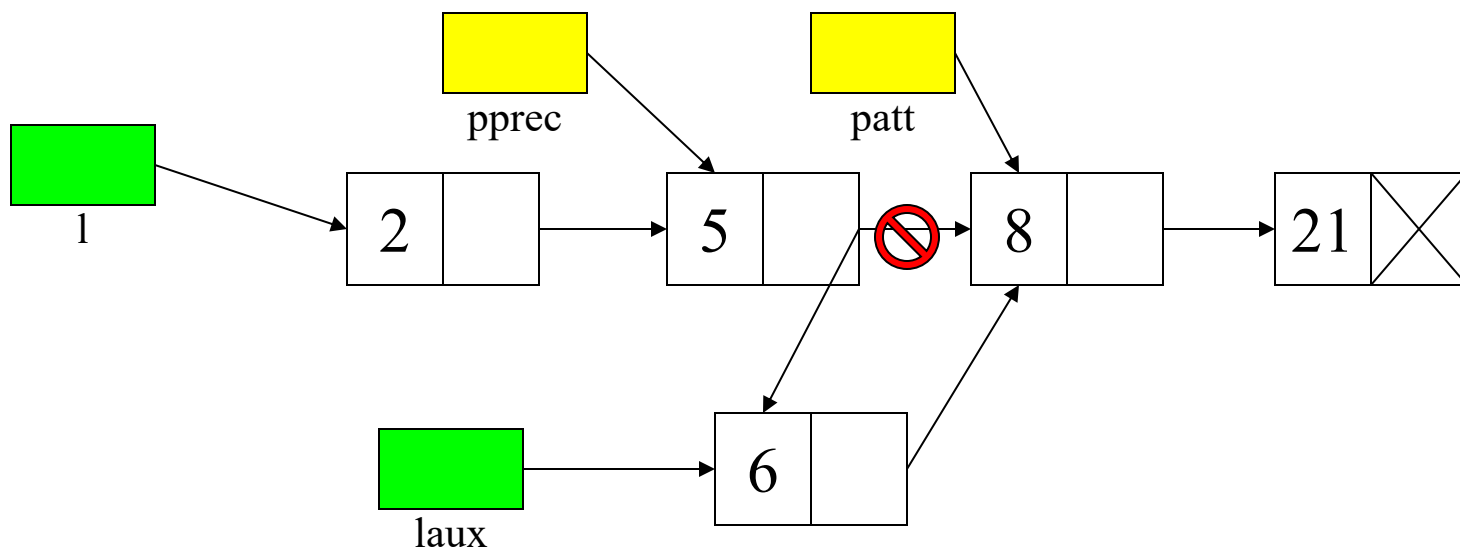
# ESERCIZIO

Scrivere una **versione** della funzione ***insord()*** **non basata sugli operatori primitivi definiti nel ADT lista (notazione a puntatori)**

Ipotesi: la lista di partenza è ordinata

Algoritmo:

- **Scandire la lista** finché si incontra un nodo contenente un **elemento maggiore di quello da inserire**
- **Allocare un nuovo nodo**, con l'elemento da inserire
- **Collegare il nuovo nodo ai due adiacenti** (vedi figura)



## ESERCIZIO (segue)

---

Il posto giusto del nuovo nodo è ***prima*** del nodo attuale -> occorre mantenere un riferimento al ***nodo precedente***

```
list insord_p(element el, list l) {
    list pprec, patt = l, paux;
    boolean trovato = 0;
    while (patt!=NULL && !trovato) {
        if (el < patt->value) trovato = 1;
        else { pprec = patt; patt = patt->next; }
    }
    paux = (list) malloc(sizeof(item));
    paux->value = el; paux->next = patt;
    if (patt==l) return paux;
    else { pprec->next = paux; return l; }
}
```