

Liste semplici - ADT

Obiettivi:

- Discutere la realizzazione collegata (puntatori a strutture) di liste semplici
- Introdurre l'ADT **lista semplice** e le operazioni tipiche su essa

ADT LISTA (1)

In generale, un ***tipo di dato astratto*** T è definito come:

$$T = \{D, \mathfrak{F}, \Pi\}$$

- ***dominio-base*** D
- insieme di ***funzioni*** $\mathfrak{F} = \{F_1, \dots, F_n\}$ sul dominio D
- insieme di ***predicati*** $\Pi = \{F_1, \dots, F_n\}$ sul dominio D

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

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

ADT LISTA (2)

cons: D x list $\rightarrow$ list (*costruttore*)
cons(6, [7,11,21,3,6]) $\rightarrow$ [6,7,11,21,3,6]

head: list $\rightarrow$ D (*selettore “testa”*)
head([6,7,11,21,3,6]) $\rightarrow$ 6

tail: list $\rightarrow$ list (*selettore “coda”*)
tail([6,7,11,21,3,6]) $\rightarrow$ [7,11,21,3,6]

emptyList: $\rightarrow$ list (*costante “lista vuota”*)

empty: list $\rightarrow$ boolean (*test di “lista vuota”*)
empty([]) $\rightarrow$ true

ADT LISTA (3)

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

OPERAZIONI PRIMITIVE

Operazione	Descrizione
cons: D x list -> list	Costruisce una nuova lista, aggiungendo l'elemento fornito in testa alla lista data
head: list -> D	Restituisce primo elemento della lista data
tail: list -> list	Restituisce la coda della lista data
emptyList: -> list	Restituisce (costruisce) la lista vuota
empty: list -> boolean	Restituisce vero se la lista data è vuota, falso altrimenti

ADT LISTA: altre operazioni non primitive

Operazione	Descrizione
member: D x list -> boolean	Restituisce vero o falso a seconda se l'elemento dato è presente nella lista data
length: list -> int	Calcola il numero di elementi della lista data
append: list x list -> list	Restituisce una lista che è concatenamento delle due liste date
reverse: list -> list	Restituisce una lista che è l'inverso della lista data
copy: list -> list	Restituisce una lista che è copia della lista data
insord: D x list -> list	Inserimento ordinato di un elemento del dominio D in una lista ordinata

ADT LISTA (4)

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

Tutte le altre operazioni, *quali ad esempio inserimento (ordinato) di elementi, concatenamento di liste, stampa degli elementi di una lista, ribaltamento 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**

ESERCIZIO: `showList` come derivata

Stampa di una lista (come derivata)

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

Realizzata come funzione derivata la `showList` *non varia* anche *se varia la rappresentazione concreta* del tipo `list`.

Serve però l'implementazione delle funzioni primitive (selettori) *head* e *tail* per questa nuova realizzazione del tipo *list* e della funzione predicativa *empty*

Creazione di una lista di interi (segue)

```
int head(list l) {  
    if (empty(l)) abort();  
    else return l->value; }
```

```
list tail(list l) {  
    if (empty(l)) abort();  
    else return l->next; }
```

```
int empty(list l) {  
    return (l==NULL); }
```

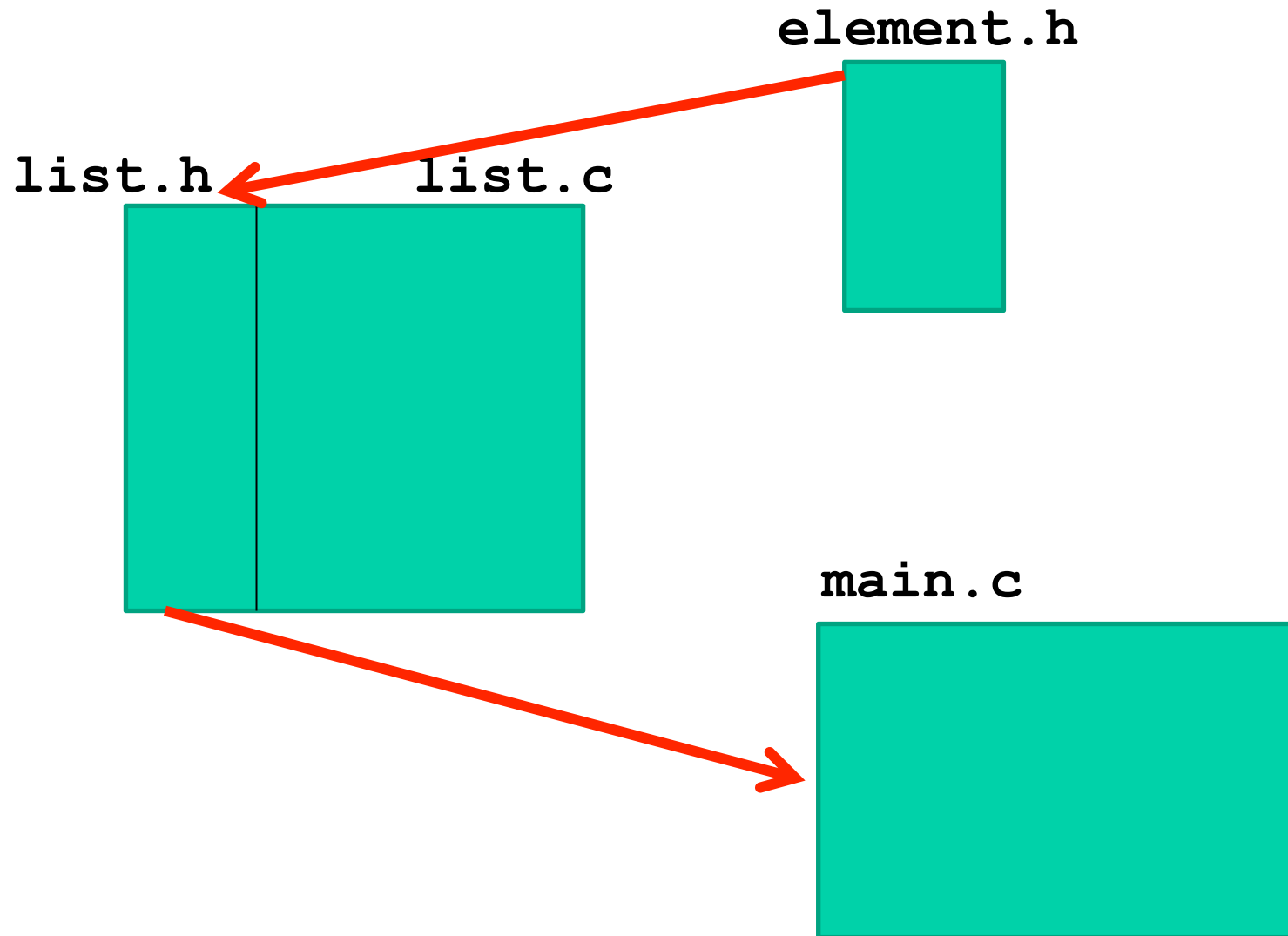
COSTRUZIONE ADT LISTA (1)

Incapsulare la ***rappresentazione concreta*** (che utilizza puntatori e strutture) e esportare sotto forma di 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 (in Laboratorio ...)***

COMPONENTI



COSTRUZIONE ADT LISTA (2)

LINEE GUIDA:

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

ADT LISTA: header file (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)) abort();
    else return l->value; }

list  tail(list l) {
    if (empty(l)) abort();
    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; }
```

Continua ... (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");
}
```

NOTA: **printf("%d", ...)** è specifica per gli interi – poi generalizzeremo e introdurremo anche operazioni ad hoc per il tipo *element* nei file *element.h* e *element.c*

ADT LISTA: il cliente (main.c)

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

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);
    printf("%d", lenght(l1));
}
```

ADT LISTA: altre operazioni non primitive

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

ADT LISTA: il predicato member

member(el, l) = falso

vero

member(el, tail(l))

se empty(l)

se el == head(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;
}
```

```
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;
}
```

ADT LISTA: il predicato member

member (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 1;
        else l = tail(l);
    } return 0;
}
```

```
// VERSIONE RICORSIVA
boolean member(element el, list l) {
    if (empty(l)) return 0;
    else if (el == head(l)) return 1;
    else return member(el, tail(l));
}
```

E' una funzione *tail ricorsiva (ottimizzazione sullo stack)*

ADT LISTA: la funzione length

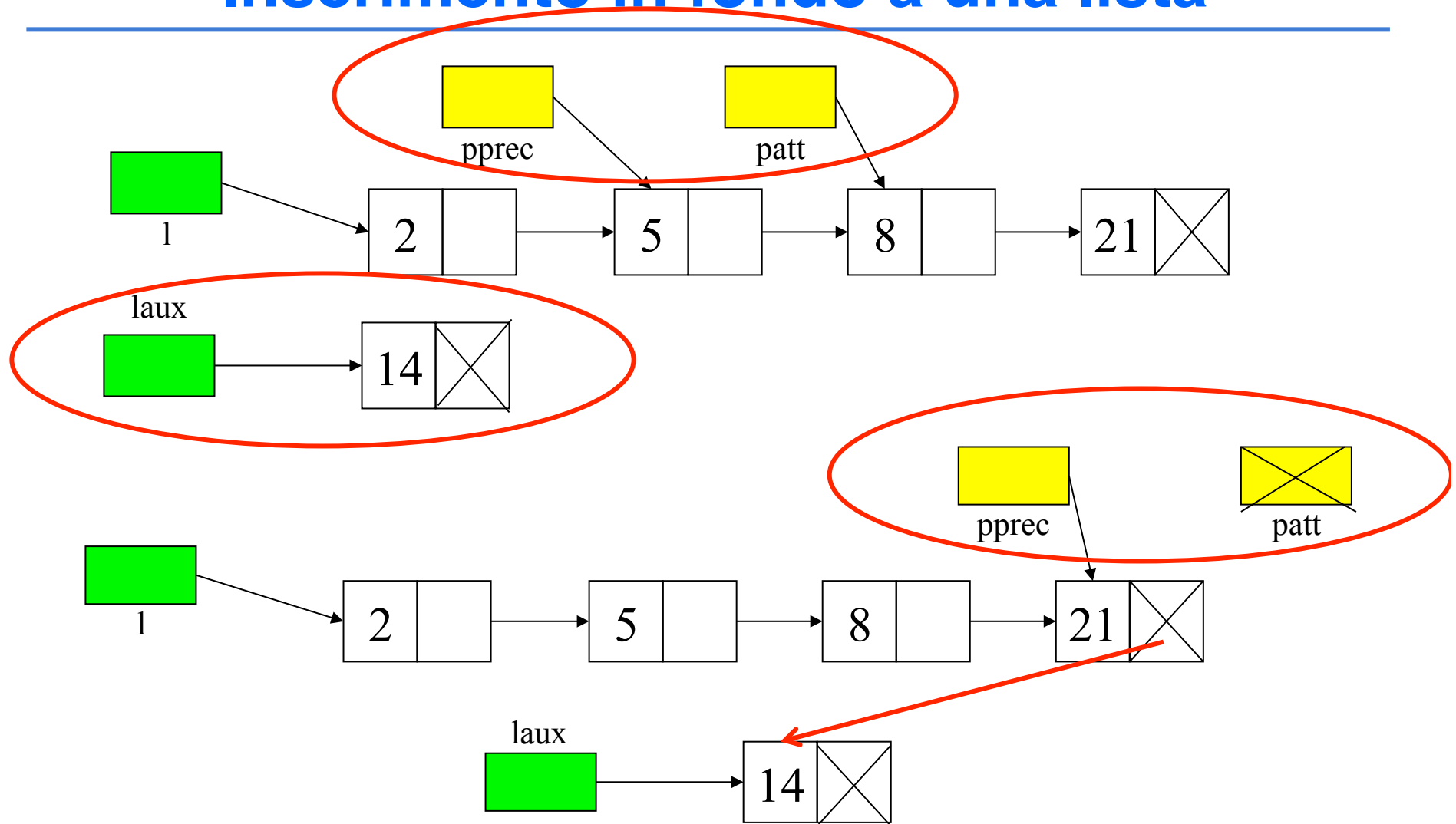
length(l) =	0	se empty(l)
	1 + length(tail(l))	altrimenti

```
// 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));
}
```

NOTA: **NON** è una funzione ***tail ricorsiva***, somma dopo la chiamata ricorsiva

Inserimento in fondo a una lista



ADT LISTA: (list.c)

```
// FUNZIONE CHE INSERISCE IN FONDO - PRIMITIVA ITERATIVA

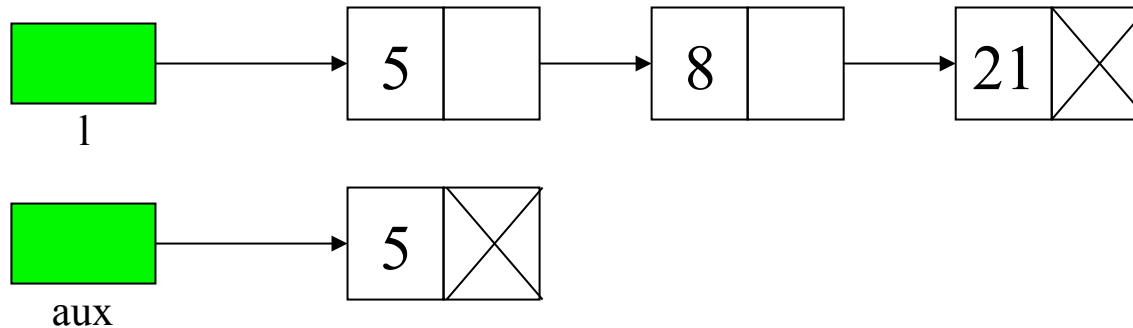
list cons_tail(element e, list l) {
    list prec, aux;
    list patt=l;

    aux=(list)malloc(sizeof(item));          // ALLOCA NODO
    aux->value=e;
    aux->next=NULL;
    if (l==NULL) return aux;                // INSERISCE IN LISTA VUOTA
    else
        { while (patt!=NULL)                // NON FINE LISTA
            { prec=patt ;
              patt=patt->next; }

          prec->next=aux;                    // AGGIUNGE IN FONDO
          return l;                         // RESTITUISCE RADICE l
        }
}
```

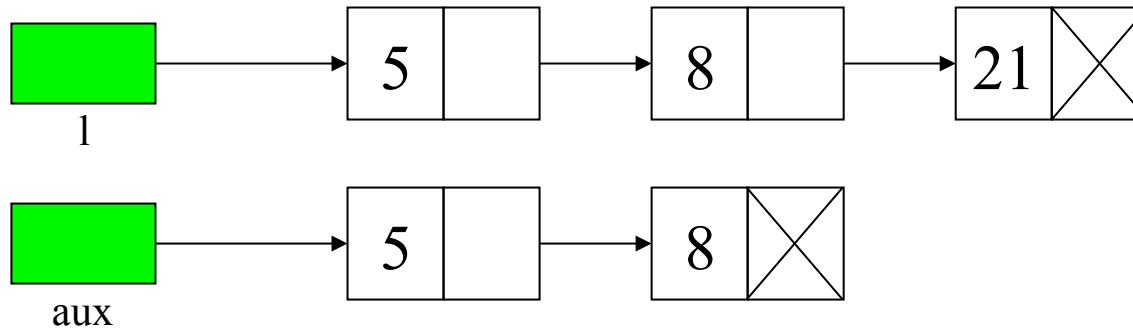
ADT LISTA: la funzione copy iterativa

```
list copy_it(list l) {  
    list aux=emptylist();           // aux=NULL  
  
    while (!empty(l))  
        { aux=cons_tail(head(l),aux);  
          l = tail(l); }  
  
    return aux;  
}
```



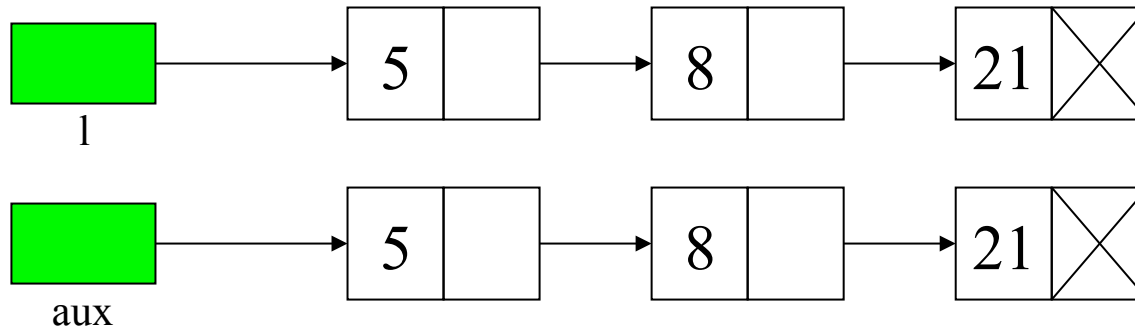
ADT LISTA: la funzione copy iterativa

```
list copy_it(list l) {  
    list aux=emptylist();           // aux=NULL  
  
    while (!empty(l))  
        { aux=cons_tail(head(l),aux);  
          l = tail(l); }  
  
    return aux;  
}
```



ADT LISTA: la funzione copy iterativa

```
list copy_it(list l) {  
    list aux=emptylist();           // aux=NULL  
  
    while (!empty(l))  
        { aux=cons_tail(head(l),aux);  
          l = tail(l); }  
  
    return aux;  
}
```

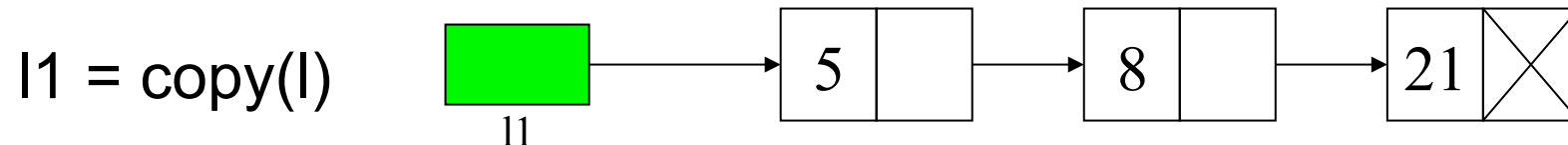
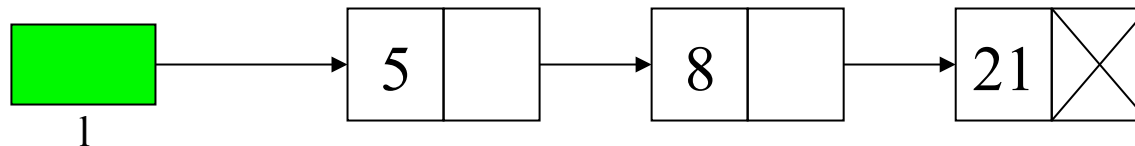


ADT LISTA: la funzione copy

Seconda versione che si basa sulle funzioni primitive (ma meno efficiente della precedente)

Si tratta di impostare un ciclo (o una funzione ricorsiva tail) 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)));  
}
```



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, ...

LISTE ORDINATE: la funzione insord (1)

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 nella coda** della lista data

I primi due casi sono operazioni elementari

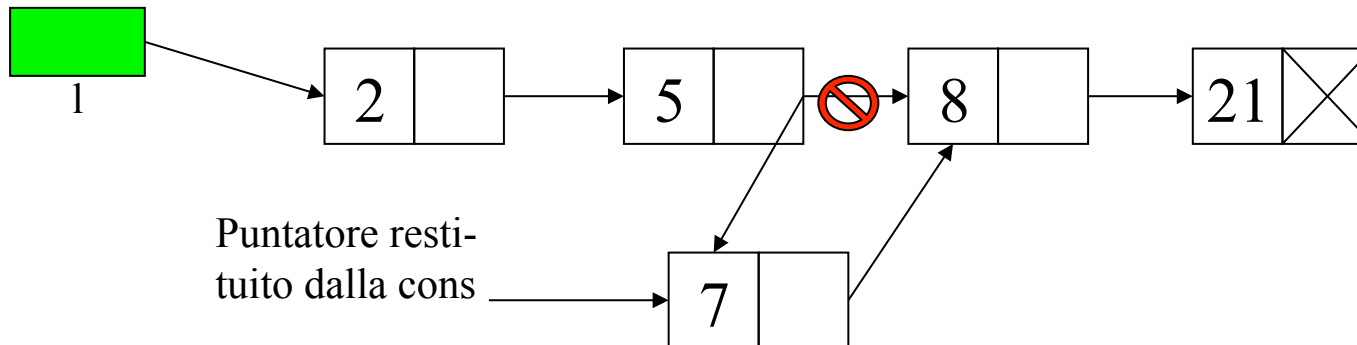
Il terzo caso riconduce il problema allo **stesso problema in un caso più semplice**: alla fine si potrà effettuare o un inserimento in testa o ci si ricondurrà alla lista vuota

```
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>  
}
```

La funzione insord (4)

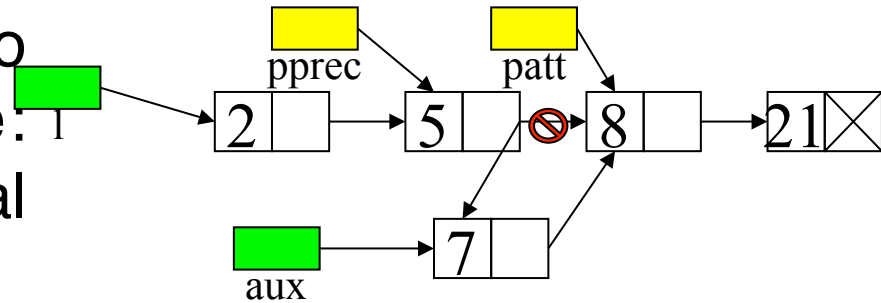
Se non ci basiamo sulle funzioni primitive, ma ***utilizziamo direttamente l'accesso ai campi*** dei nodi della lista, possiamo effettuare l'inserimento in modo efficiente, e ricorsivo:

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



Inserimento ordinato - iterativo

Il posto giusto del nuovo nodo è **prima** del nodo attuale: occorre anche il 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; }  
}
```

ADT LISTA: altre operazioni (non svolto)

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

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*** ad append

ADT LISTA: la funzione append (2)

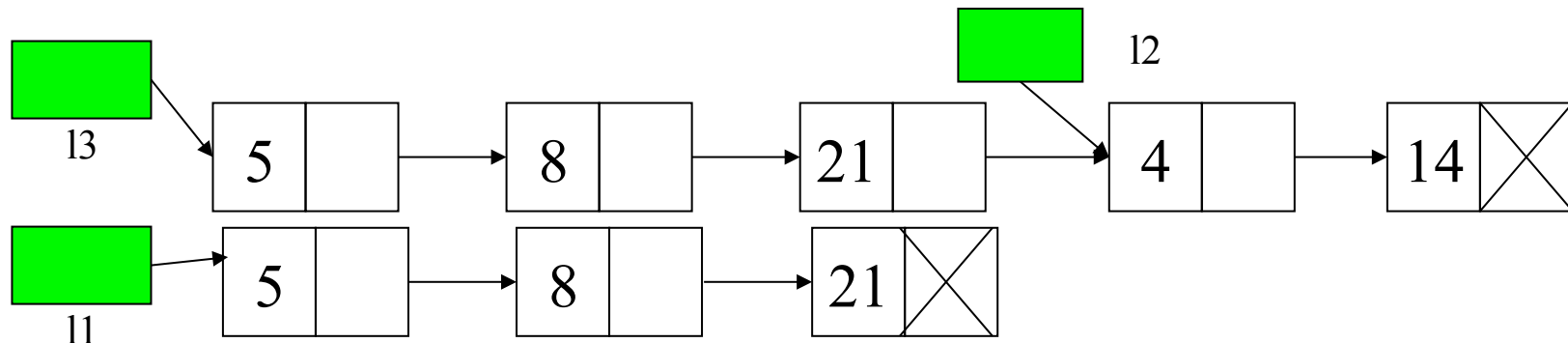
append(l1, l2) = l2 se empty(l1)
 cons(head(l1), append(tail(l1), l2)) altrimenti

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

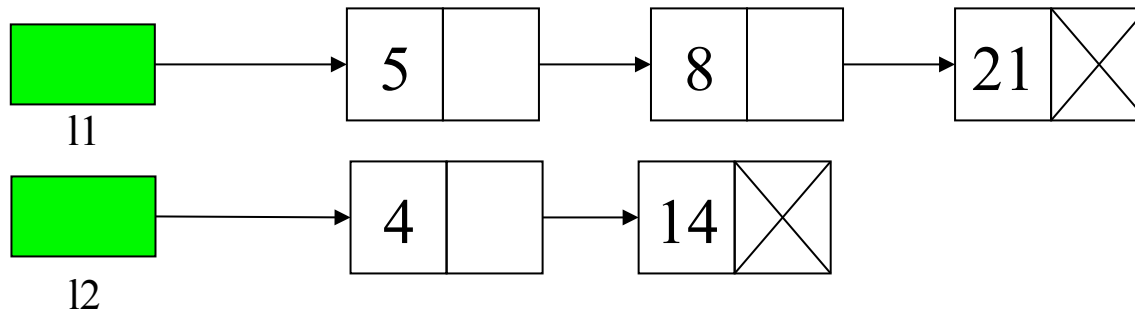
NOTA: quando l1 diventa vuota, **append** restituisce direttamente l2, **non una sua copia** -> l1 è duplicata, ma **l2 rimane condivisa**

Structure sharing (parziale)

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

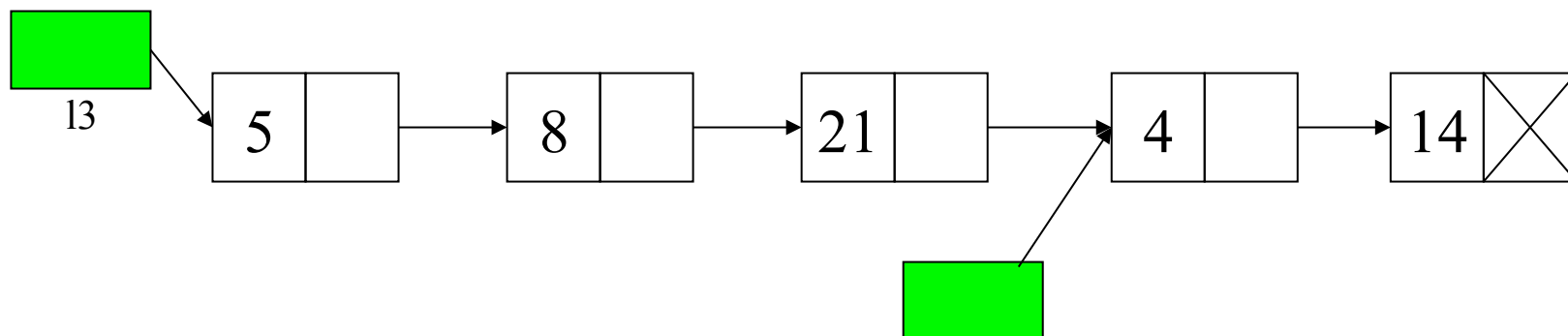


ADT LISTA: la funzione append (3)



Se si vuole evitare lo ***structure sharing (parziale)***

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



Restituito da **copy(l2)**

ADT LISTA: la funzione reverse (1)

Per ottenere una lista **ribaltamento** 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)

```
reverse(l) = emptyList()           se empty(l)
append(reverse(tail(l)), cons(head(l), emptyList())) altrimenti
```

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

ADT LISTA: la funzione reverse (2)

Esercizio: scrivere una versione ***ricorsiva tail*** di reverse

Esercizio: scrivere una versione ***iterativa*** di reverse

ADT LISTA: la funzione reverse (2)

Esercizio: scrivere una versione **ricorsiva tail** di reverse

Per passare da una versione ricorsiva ad una ricorsiva tail:

- **funzione di interfaccia** che richiami la funzione tail-ricorsiva con necessari parametri supplementari
- definire la funzione tail-ricorsiva con un parametro lista in più, inizialmente vuota, in cui inserire in testa gli elementi via via prelevati dalla lista data

```
list reverse(list l) {  
    return rev(emptyList(), l);  
}  
list rev(list l2, list l1) {  
    if (empty(l1)) return l2;  
    else return rev(cons(head(l1), l2), tail(l1));  
}
```

ADT LISTA: la funzione reverse (3)

Esercizio: scrivere una versione *iterativa* di reverse

```
list reverse_it(list l) {  
    list aux=emptyList();  
  
    while (!empty(l))  
        { aux = cons(head(l), aux);  
          l=tail(l); }  
    return aux;  
}
```

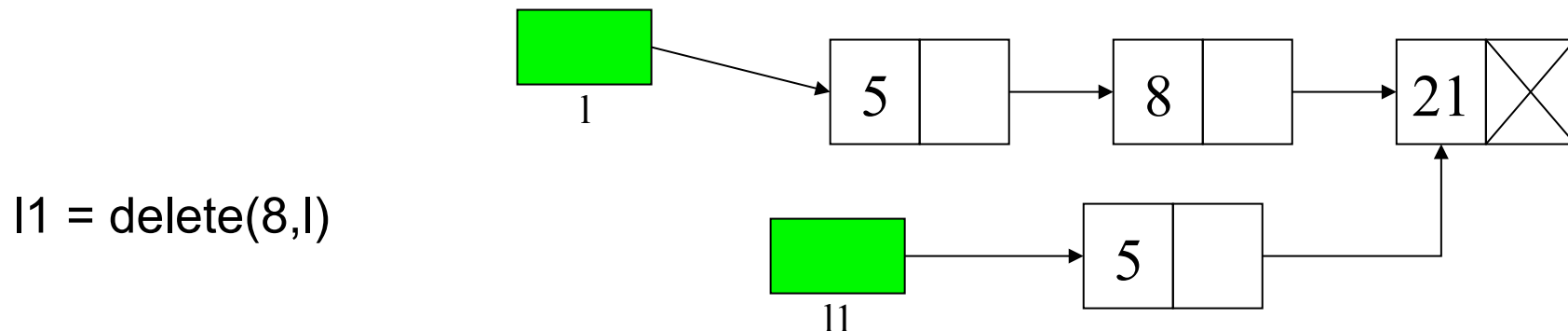
ADT LISTA: la funzione delete (1)

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

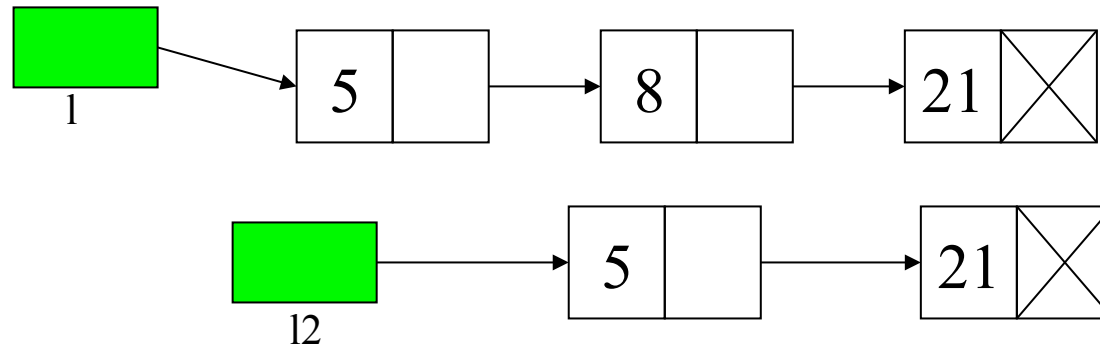
```
list delete(element el, list l) {  
    if (empty(l)) return emptyList();  
    else if (el == head(l)) return tail(l);  
    else return cons(head(l), delete(el, tail(l)));  
}
```



ADT LISTA: la funzione delete (2)

Per non avere condivisione:

L2 = delete (8,copy(l))



CONCLUSIONE

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

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

ESERCIZIO: scrivere una **versione iterativa** di delete

ADT LISTA: la funzione delete (3)

Esercizio: scrivere una versione *iterativa* di delete

La funzione delete vers. iterativa (3)

Esercizio: scrivere una versione *iterativa* di delete

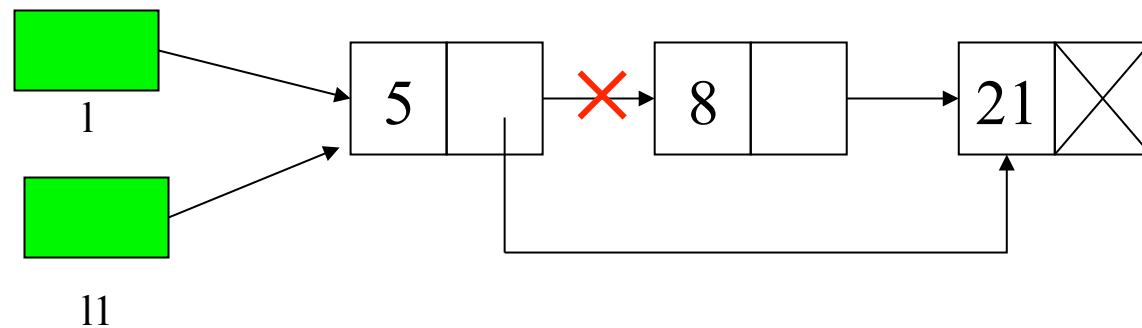
```
list  delete_it(element el, list l)
{ list prec, current, aux;
  if (empty(l)) return(emptylist());
  else
  { current=l;
    do
    { if(!(head(current)==el)
      { prec=current;
        current=tail(current); }
      while ( (!empty(current)) || !(head(current)==el) );
    }
    if (!empty(current))
      if (current==l) return tail(l);    // DELETE PRIMO
      else
      { prec->next=current->next;
        return l;           }          // DELETE IN MEZZO
    else return l;
  }
```

ADT LISTA: la funzione delete_it (4)

```
list delete_it(element el, list l) {  
    ...  
}
```

l1 = delete_it(8,l)

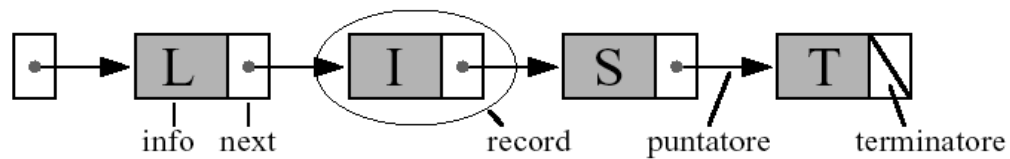
Anche l è cambiata!!



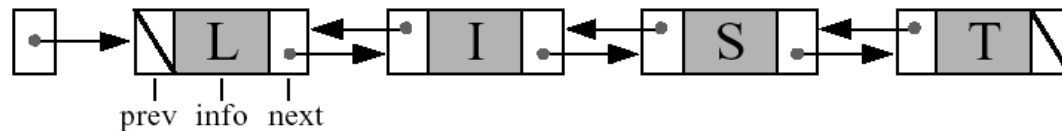
l1 = delete_it(8,copy(l)) Per evitare structure sharing

ESERCIZIO: includere il rilascio della memoria del nodo cancellato

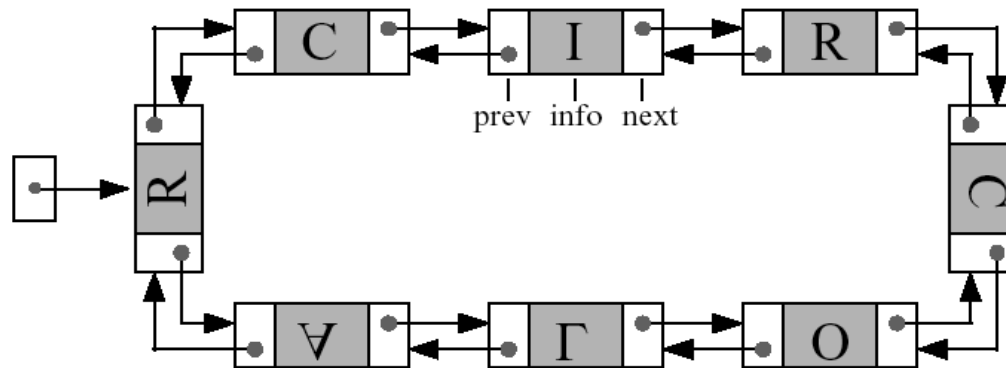
Altri esempi di strutture dati collegate



lista semplice



lista doppiamente collegata



lista circolare doppiamente

e anche ... alberi binari, alberi n-ari (liste di liste)

PARTE II: 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 ***qualunque tipo di elementi***

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

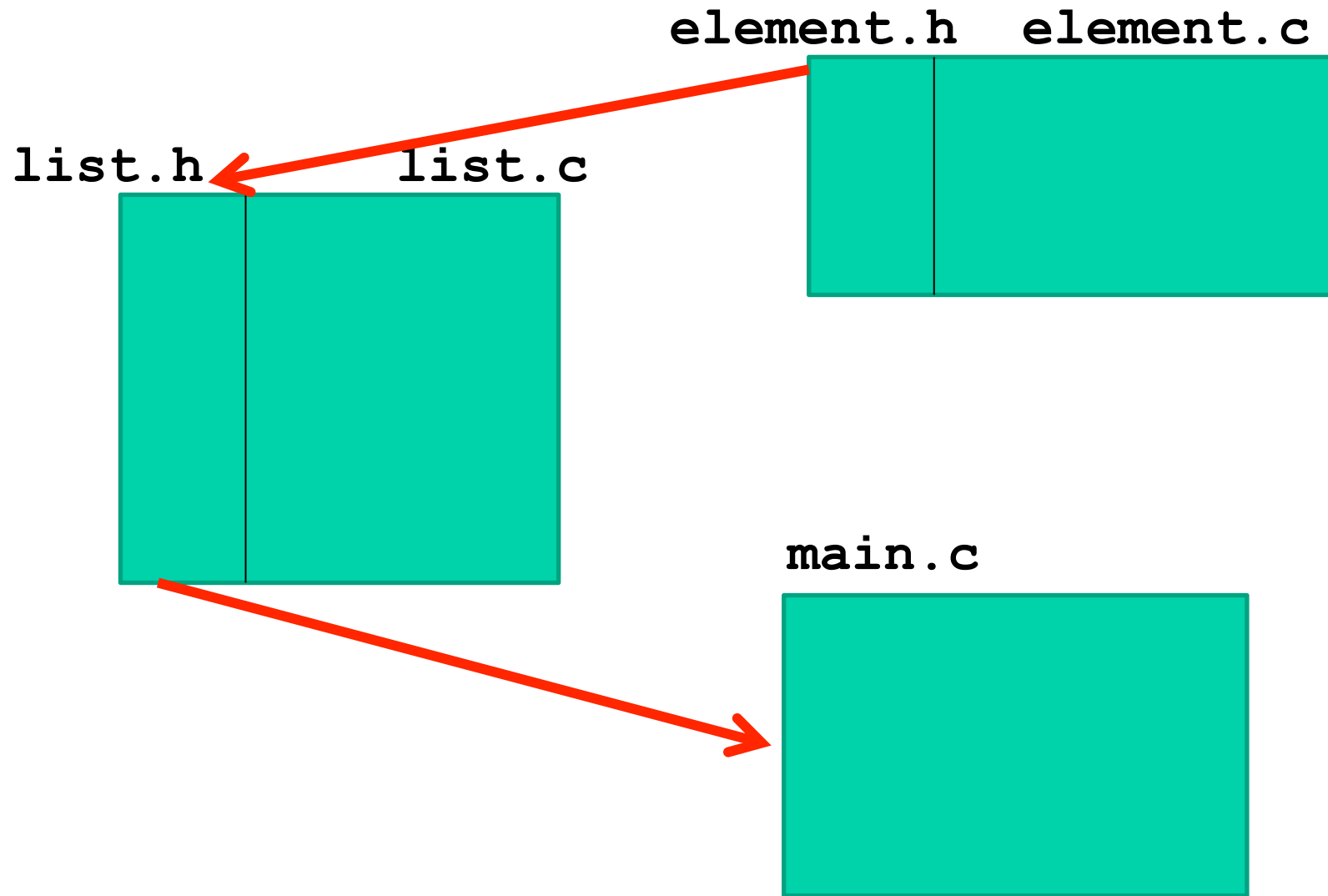
Osservazioni:

- ***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

COMPONENTI



ADT ELEMENT: element.h

Header element.h deve contenere

- ***definizione*** del tipo element
- ***dichiarazioni*** delle varie funzioni fornite

Poiché contiene una ***definizione***, header dovrà essere protetto dal ***problema delle inclusioni multiple***

```
#ifndef ELEMENT_H
#define ELEMENT_H

typedef int element;                //DEFINIZIONI
typedef enum { false, true } boolean;

boolean isLess(element, element);   //DICHIARAZIONI
boolean isEqual(element, element);
element getElement(void);
void printElement(element);

#endif
```

ADT ELEMENT: element.c

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

boolean isEqual(element e1, element e2) {
    return (e1==e2); }

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

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

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

TO DO

Si **definiscano** i file ***element.c*** e ***element.h*** che realizzano l'ADT ***element*** (come intero)



Si modifichino i file ***list.h*** e ***list.c*** già disponibili, generalizzando le loro operazioni in funzione di quelle esportate dall'ADT ***element*** (showList, insord)

Il ***main*** da realizzare deve leggere la sequenza e inserire ogni elemento letto in una ***lista con inserimento in testa*** e infine stampare la lista creata usando le funzionalità dell'ADT ***element*** e ***list***

COSA CAMBIA NELL' ADT LISTA?

Ridefinendo in funzione delle operazioni esportate da `element.h` il codice delle operazioni dell'ADT lista cerchiamo di aumentarne la riusabilità

Ad esempio, prima ...

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

NOTA: **printf**("%d", ...) è specifica per gli interi

ADT list.c *prima ...*

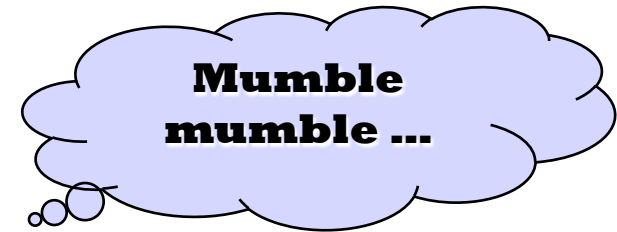
insord iterativa

```
list insord(element el, list l) {  
    list pprec, patt = l, paux;  
    boolean trovato = false;  
    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; }  
}
```



Realizzare *l'ADT element* (intero e operazioni, si veda *element.h*)

Modificare *list.h* e *list.c* già disponibili



Il *main* da realizzare deve leggere la sequenza e inserire ogni elemento letto in una *lista ordinata* e stamparla

...

(vediamo poi la soluzione insieme)

ADT ELEMENT: element.c

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

boolean isEqual(element e1, element e2) {
    if (e1==e2) return true;
    else return false; }

boolean isLess(element e1, element e2) {
    if(e1<e2) return true;
    else return false; }

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

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

ADT LISTA: *ora*

```
void showList(list l) {  
    while (!empty(l)) {  
        printElement(head(l));  
        l = tail(l);    }  
}
```

NOTA: **printElement** stampa su stdout l'elemento in testa ad l,
head(l)

ADT list.c ora ... più genericità!

insord iterativa

```
list insord(element el, list l) {  
    list pprec, patt = l, paux;  
    boolean trovato = 0;  
    while (patt!=NULL && !trovato) {  
        if (isLess(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; }  
}
```

ADT LISTA: il cliente (main.c)

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

main() {
    list l1 = emptyList();
    element el;
    do { printf("\n Introdurre valore:\t");
        el=getElement();
        l1 = insord(el, l1);
    } while (!isEqual(el,0));

    showList(l1);
}
```

GENERALIZZIAMO ULTERIORMENTE ...

Un file sequenziale di tipo testo (ESPR.TXT) contiene stringhe (una per linea) costituite dai caratteri {a,b,c,*,+}. Si realizzi un programma C che:

a) riconosca le stringhe del file uguali a "a*b+c" oppure a "a+b*c" e le inserisca con i loro numeri di linea in una lista a puntatori, L1, ***ordinata sul campo stringa (a parità di stringa, si ordini in base alla posizione)***;

b) produca, a partire dalla lista L1 generata precedentemente, un file (UNICHE.TXT) di tipo testo in cui ogni stringa accettata compare seguita, sulla stessa linea, dalla posizione originale nel file ESPR.TXT

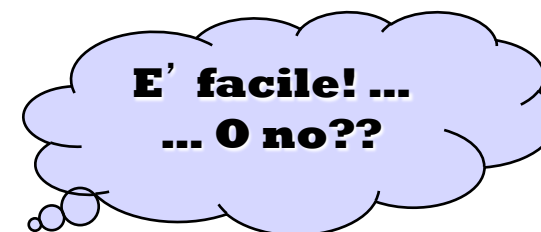
Esempio:

ESPR.TXT:

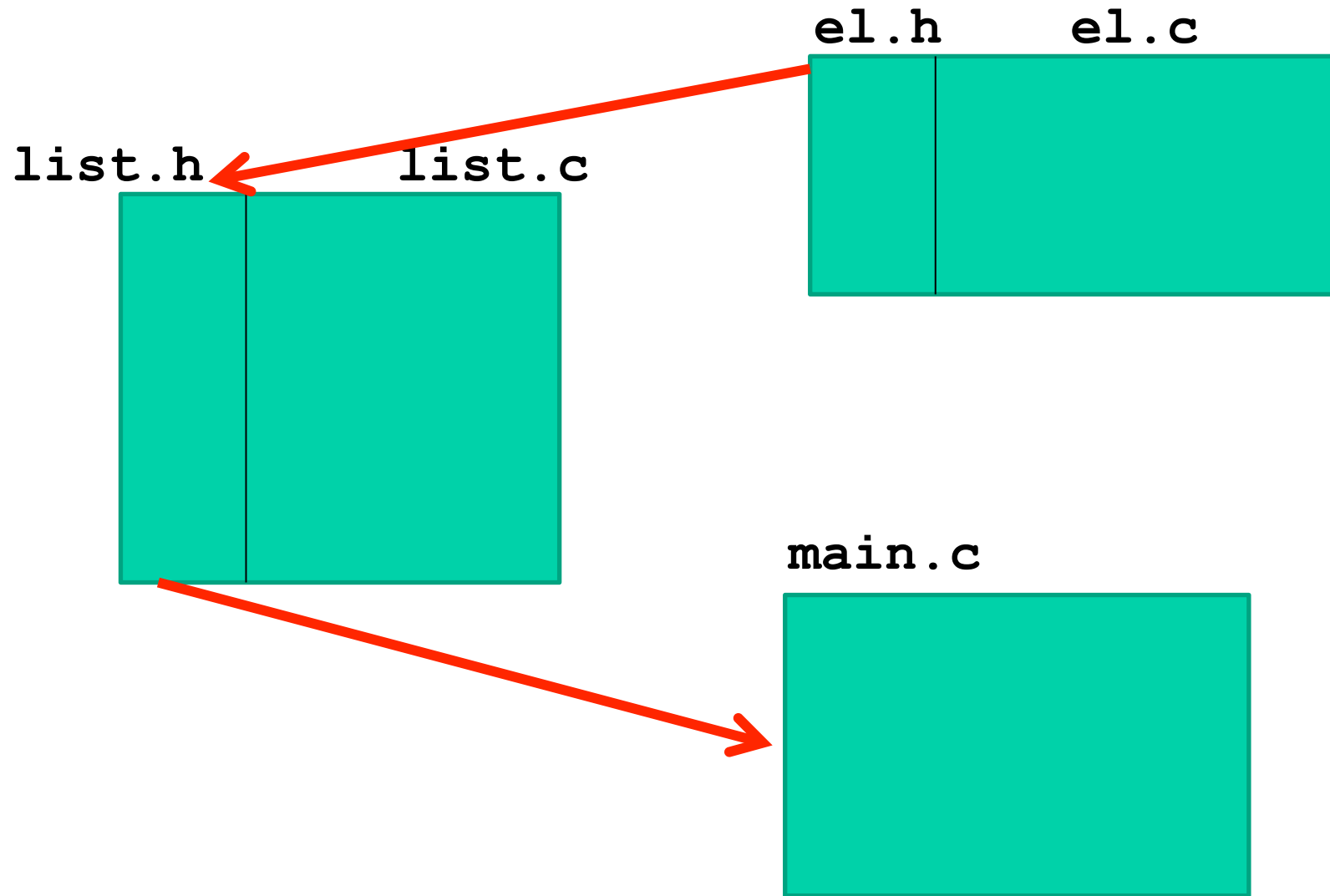
a*b+c
a**b+
a+b*c
a*b+c
a+b*c

UNICHE:

a*b+c 1
a*b+c 4
a+b*c 3
a+b*c 5



COMPONENTI



Interfaccia modulo elementi: el.h

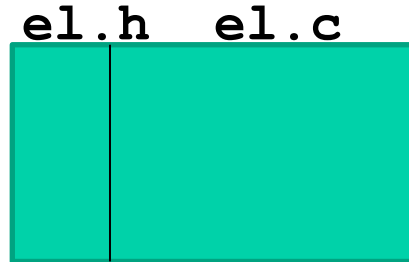
```
/* ELEMENT TYPE - file el.h*/
#ifndef ELEMENT_H
#define ELEMENT_H

#define N 30
typedef struct { char stringa[N];
                int  pos; } element;
typedef enum {false, true} boolean;

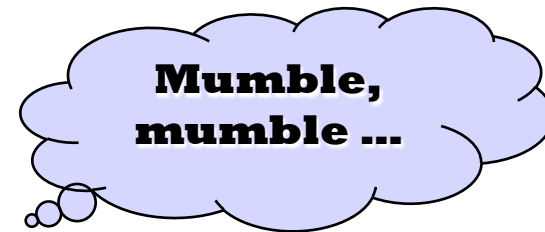
boolean isLess(element, element);
boolean isEqual(element, element);
element getElement(char *, int);
void printElement(FILE *, element);
#endif
```

TO DO: el.c

Realizzare la parte implementazione del nuovo ADT degli elementi (strutture)



Ove necessario rivedere le funzioni dell' ADT lista



...

(vediamo poi la soluzione insieme)

ADT LISTA: list.h

FILE HEADER (list.h)

non cambia di molto

```
#include "el.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 fshowList(FILE *, list);    // NON PRIMITIVE
boolean member(element, list);
...
```

Implementazione modulo elementi: el.c

```
#include "el.h"
#include <stdio.h>
#include <string.h>

boolean isEqual(element e1, element e2) {
    return (!strcmp(e1.stringa, e2.stringa)); }

boolean isLess(element e1, element e2) {
    if (strcmp(e1.stringa, e2.stringa))
        return (strcmp(e1.stringa, e2.stringa) < 0);
    else return (e1.pos < e2.pos); }

element getElement(char *s, int pos) {
    element el;
    strcpy(el.stringa, s);
    el.pos = pos;
    return el; }

void printElement(FILE *f, element el) {
    fprintf(f, "%s\t%d\n", el.stringa, el.pos); }
```

II main (1)

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

void main (void)
{
    list L1 = emptylist();
    element EL;
    char s[N];
    int pos=0;
    FILE *f, *g;

    f = fopen("ESPR.TXT",rt");
    if(f==NULL)
    {
        printf("errore apertura espr.txt \n");
        exit(-1);
    }
```

Il main (2)

//DOMANDA a) ciclo di scansione del file

```
while (fscanf(f, "%s", s) !=EOF)
{   pos++;
    if ( (strcmp(s,"a*b+c")==0) ||
        (strcmp(s,"a+b*c")==0) )
    {   EL = getElement(s,pos);

        // e inserimento ordinato in L1
        L1 = insord(EL,L1);
    }
}
fclose(f);
```

II main (3)

```
// DOMANDA b)
```

```
g = fopen("UNICHE.TXT", "wt");  
if (g==NULL)  
    { printf("errore apertura uniche.txt \n");  
      exit(-2);    }  
  
fshowList(g,L1);    // MA ANCHE: showList(L1);  
fclose(g);  
}
```

```
a*b+c 1  
a*b+c 4  
a+b*c 3  
a+b*c 5
```

ADT LISTA: *ora*

```
void fshowList(FILE *f, list l) {  
    while (!empty(l)) {  
        printElement(f, head(l));  
        l = tail(l);    }  
}
```

NOTA: **printElement** stampa su file testo le componenti di head(l):

```
void printElement(FILE *f, element el) {  
    fprintf(f, "%s\t%d\n", el.stringa, el.pos); }  
}
```

TO DO ... MORE

c) trovi, a partire dalla lista L1 generata precedentemente, la parola più frequente

Esempio:

ESPR.TXT:

$a*b+c$

$a**b+$

$a+b*c$

$a*b+c$

$a+b*c$

TO DO ... MORE

d) produca, a partire dalla lista L1 generata precedentemente, un file (UNICHE.TXT) di tipo testo in cui ogni stringa accettata compare ***una sola volta*** seguita, sulla stessa linea, dalle posizioni originali nel file ESPR.TXT

Esempio:

ESPR.TXT:

$a*b+c$

$a**b+$

$a+b*c$

$a*b+c$

$a+b*c$

UNICHE:

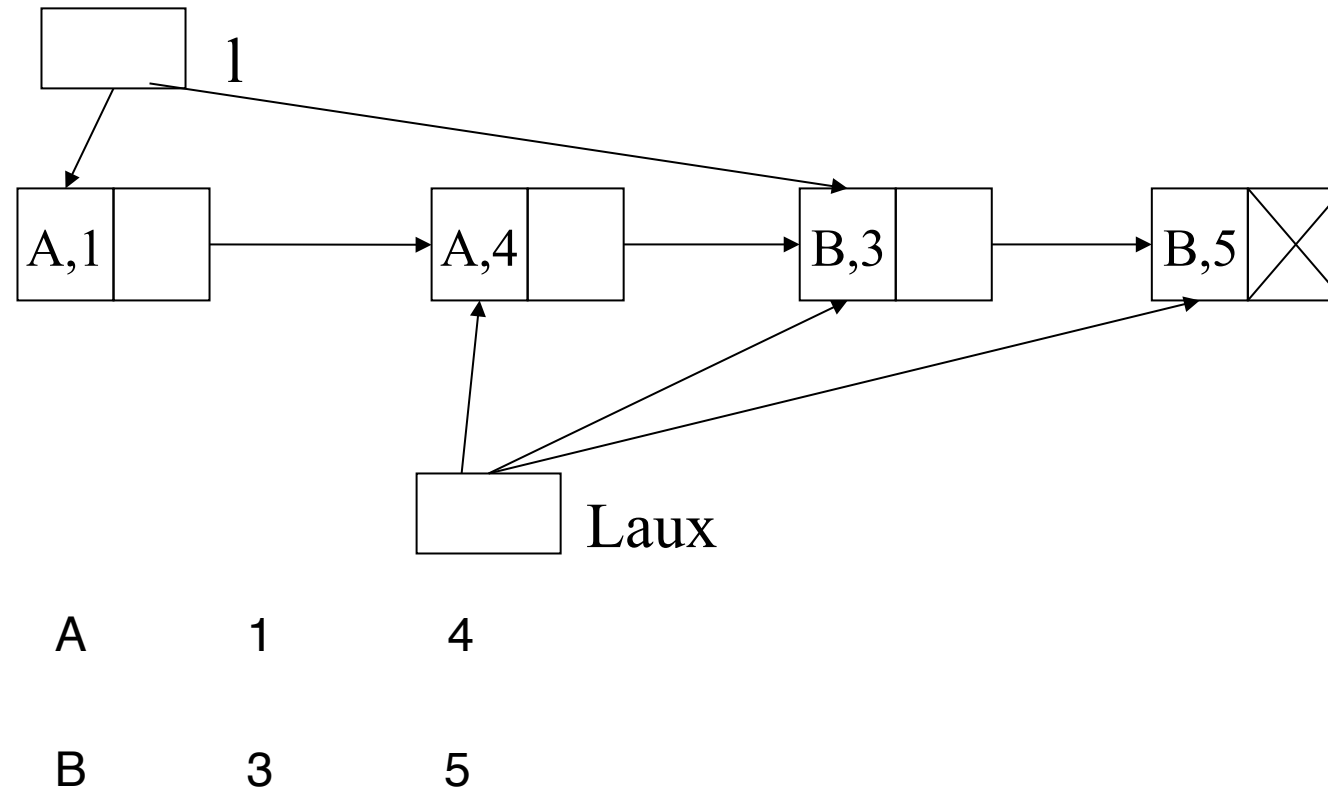
$a*b+c$ 1 4

$a+b*c$ 3 5



**E' già meno
facile da fare ...
Mumble,
mumble ...**

Come scandire la lista?



Scansione innestata, avanzo con Laux (che parte da tail(l)) finché trovo nodi con la stessa stringa (o arrivo alla fine della lista); non appena trovo una stringa diversa, avanzo con l fino a questo nodo e riprendo la scansione innestata (partendo con Laux=tail(l)).



REALIZZATE QUESTA FUNZIONE

```
void fshowList2(FILE *f, list l);
```

Nuova funzione *ad hoc* per questa stampa

```
void fshowList2(FILE *f, list l) {
    list Laux; int fine;
    while (!empty(l))
        { EL=head(l);
          fprintf(f,"%s",EL.stringa);
          fprintf(f,"\t%d", EL.pos);
          Laux=tail(l); fine=0;
          while ((!empty(Laux) && !fine)
              if (!strcmp(Laux->value.stringa,
                          l->value.stringa) )
                  { fprintf(f,"\t%d",Laux->value.pos);
                    Laux=tail(Laux);          }
              else fine=1;
          fprintf(f,"\n");
          l=Laux;
        }
}
```

CONCLUSIONE

Che cosa possiamo trarre come ulteriore conclusione?

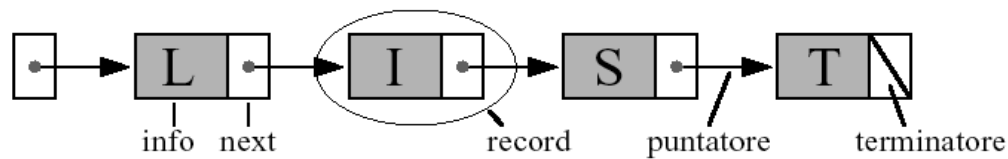


L'obiettivo del corso è presentare strumenti e approcci, e far sì che li assimiliate non che li impariate a memoria ...

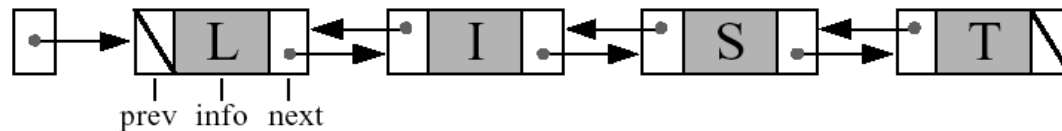
In sede d'esame si valuta se sapete lavorare / gestire le strutture dati viste, anche al di là delle funzioni base o derivate tipiche ...

... una funzione non tipica su liste e/o alberi è **sempre** richiesta nel testo ... fate esercizi, scrivete e fate girare qualche programma!!

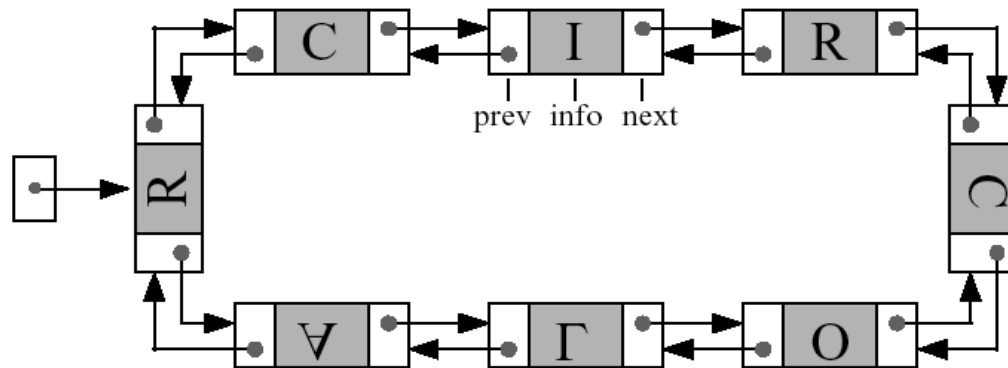
Altri esempi di strutture dati collegate



lista semplice



lista doppiamente collegata



lista circolare doppiamente

e anche ... alberi binari, alberi n-ari (liste di liste)