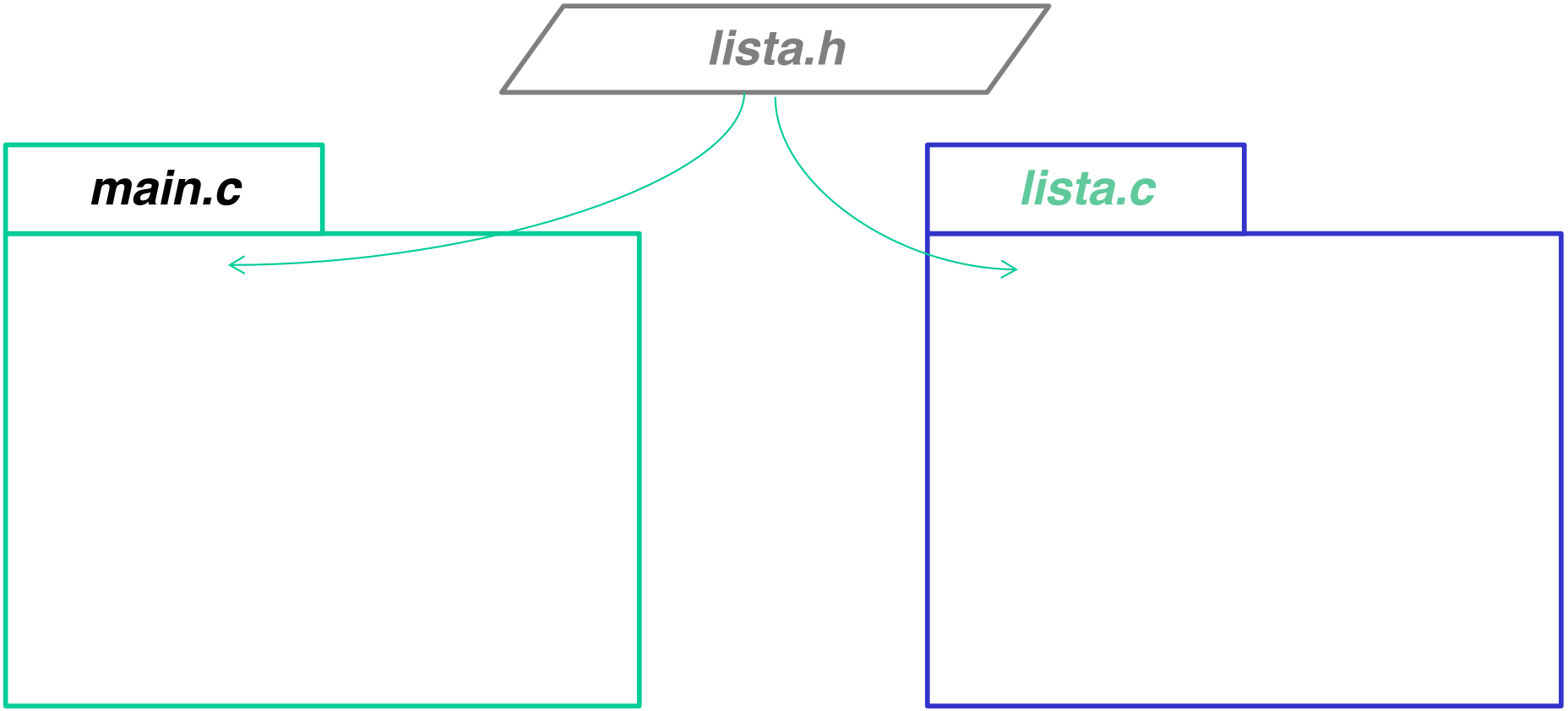
Liste semplici - ADT

Obiettivi:

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

ADT Lista semplice



lista.h, tipo **list** e prototipi delle funzioni (e tipo degli elementi in lista)
lista.c, codice delle funzioni

ADT LISTA (1)

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

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

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

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

- D può essere qualunque (negli esempi visti **`int`**)
- $\mathfrak{I}$ = { cons, head, tail, emptyList }
- Π = { empty }

ADT LISTA (2)

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 (***costante "lista vuota"***)

empty: list -> boolean (***test di "lista vuota"***)

```
empty( [] ) -> 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***

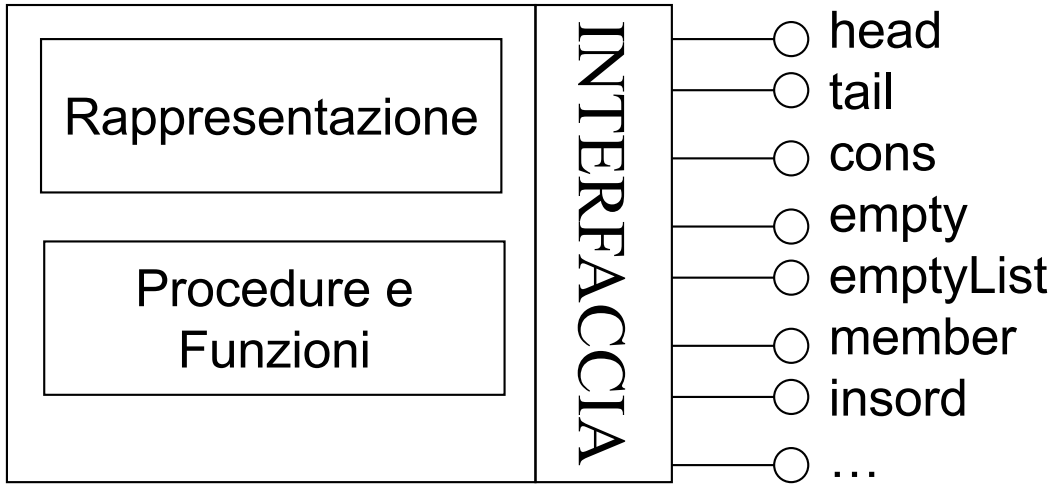
NOTA - Tipo list è definito in modo induttivo:

- 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***

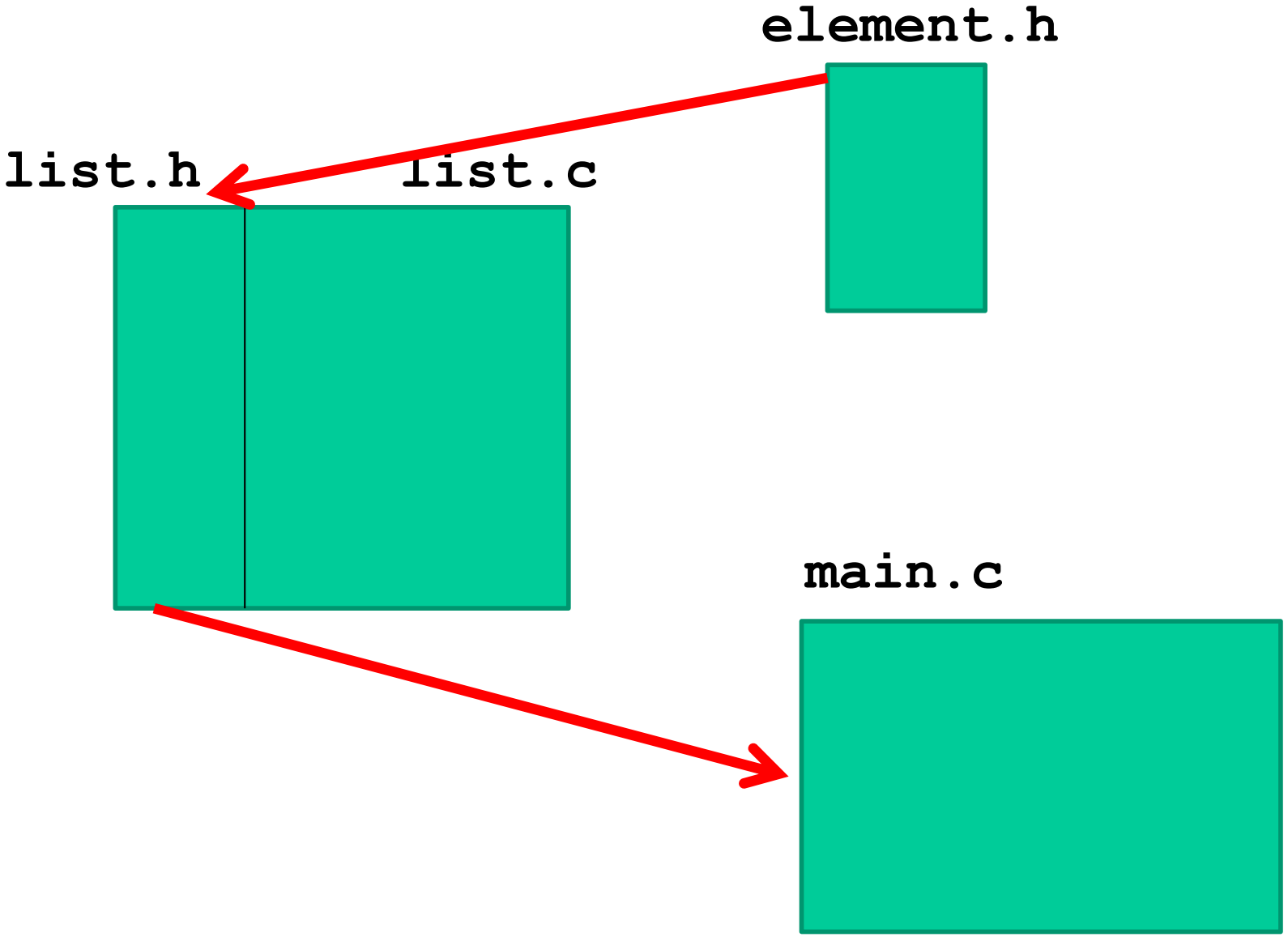
IMPLEMENTAZIONE ADT LISTA (1)

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***

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 — 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

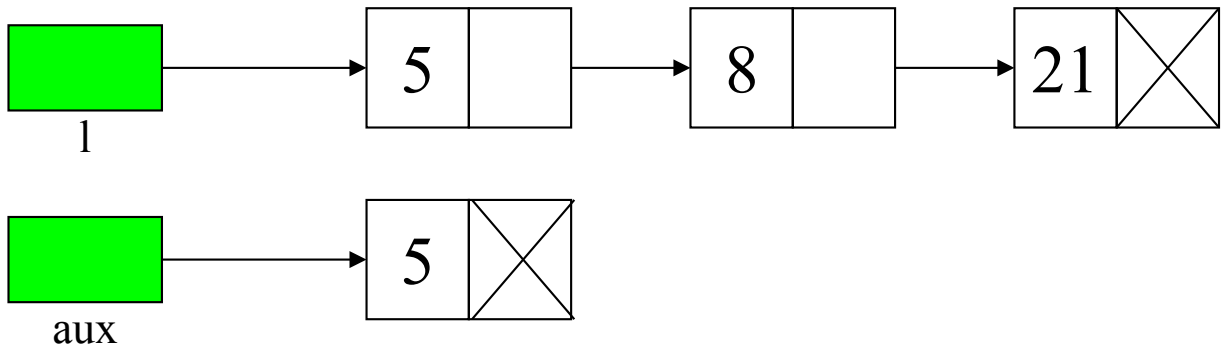
```
// VERSIONE ITERATIVA
int length(list l) {
        int n = 0;
        while (!empty(l)) {
                n++; l = tail(l); }
        return n;
}
```

```
// 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

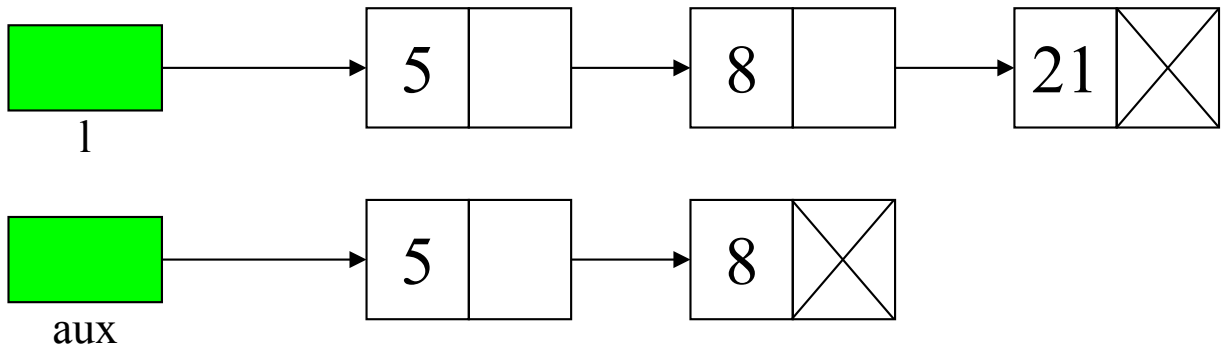
ADT LISTA: la funzione copy iterativa

```
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;
}
```

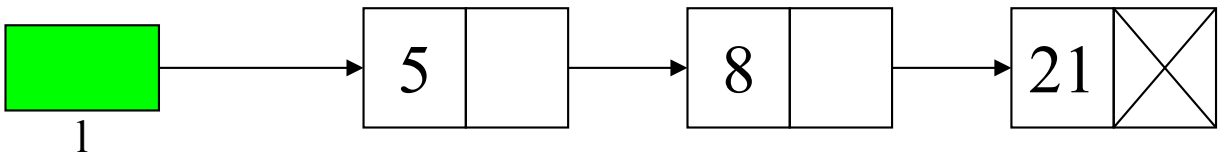


ADT LISTA: la funzione copy

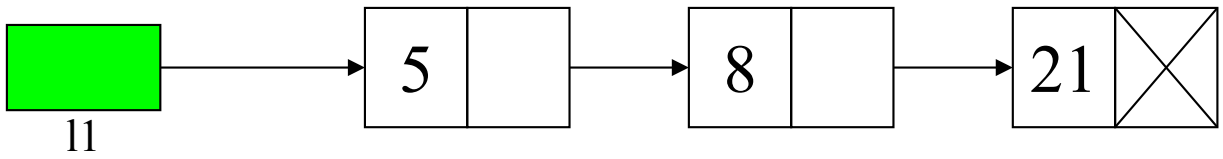
Versione (ricorsiva) che si basa sulle funzioni primitive

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)));
}
```



l1 = copy(l)



Inserimento ordinato in lista

- Iterativa (vers. 3), *l'abbiamo già vista*

  Scandisce la lista con due puntatori ausiliari, fino al punto in cui va inserito l'elemento, e aggiunge un nodo sistemando i puntatori

- Ricorsiva
  - Senza duplicazione (no structure copying), vers. 1
  - Con duplicazione di parte della lista (con structure copying), vers. 2

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 nella coda*** della lista data

I primi due casi sono operazioni elementari

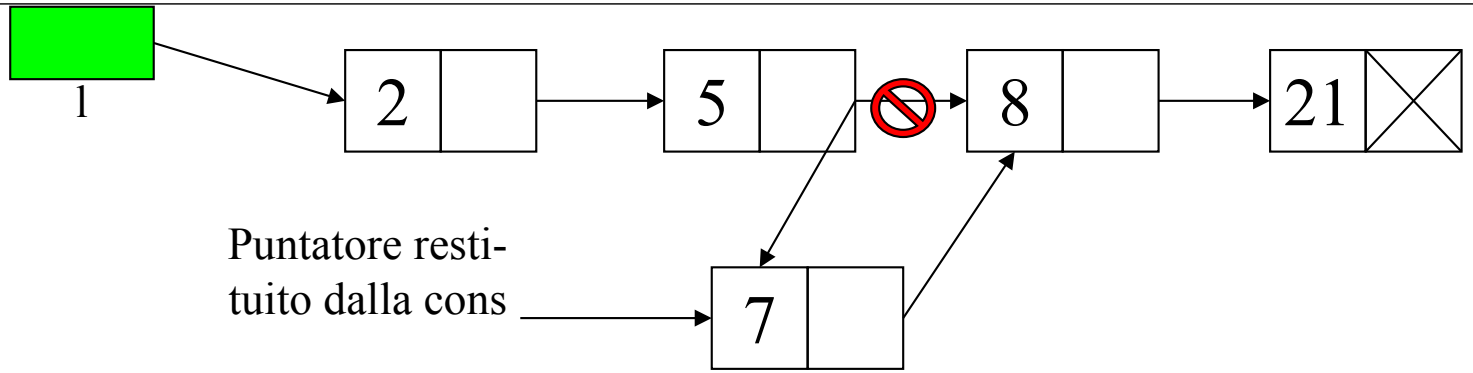
Il terzo caso riconduce il problema allo ***stesso problema in un caso più semplice***: alla fine si potrà effettuare o un inserimento in testa o ci si ricondurrà alla lista vuota

```
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 (vers. 1)

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



LISTE ORDINATE: la funzione insord (vers. 2)

Basandosi sulle funzioni primitive, non esistendo primitive di modifica, in questo approccio 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)));
}
```

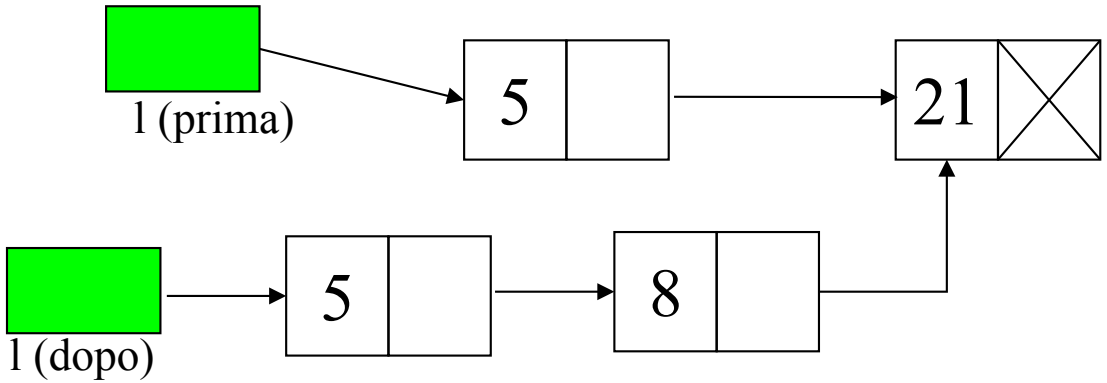
- tutta la parte iniziale della lista viene ***duplicata***;
- parte successiva al punto inserimento è invece ***condivisa***

insord (vers. 2)

- tutta la parte iniziale della lista viene *duplicata*;
- parte successiva al punto inserimento è invece *condivisa* ***(structure sharing)***

```
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)));
}
```

l = insord(8,l)



ADT LISTA: altre operazioni (laboratorio)

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

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 (*cont.*)

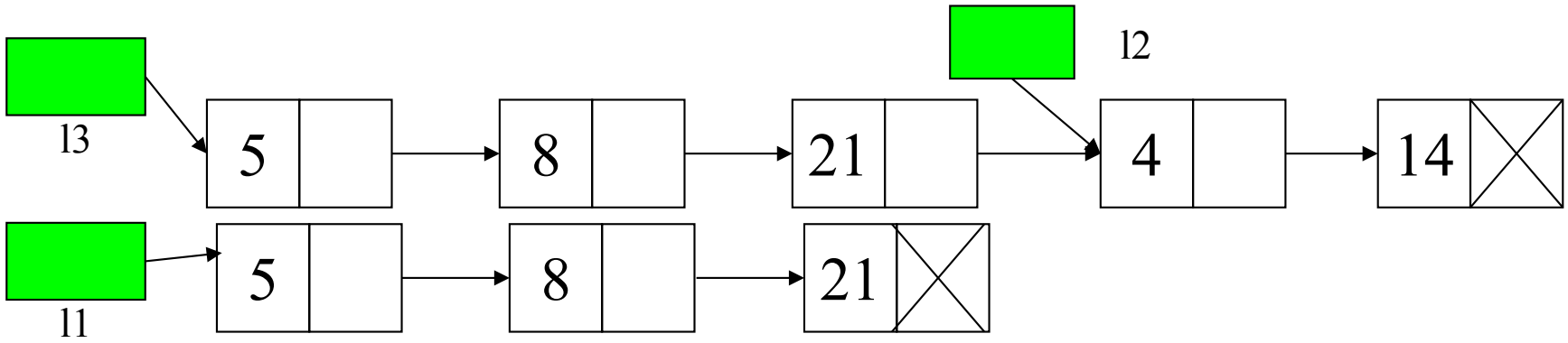
append(l1, l2) = l2 se empty(l1)
cons(head(l1), append(tail(l1), l2)) altrimenti

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

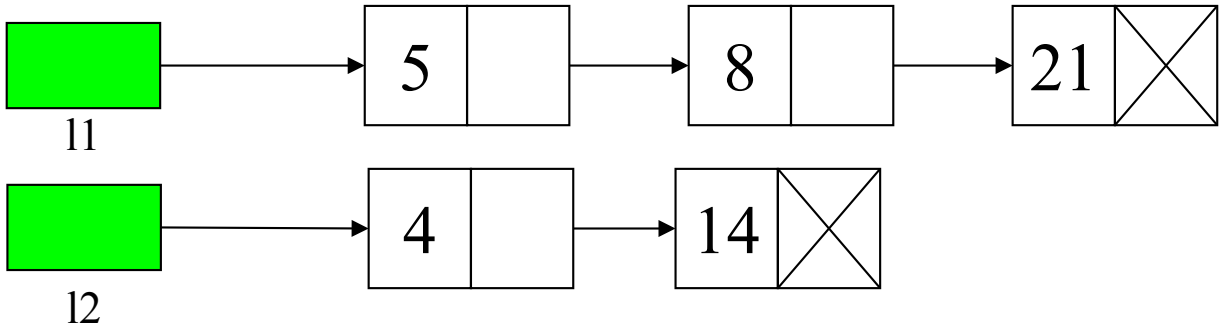
NOTA: quando l1 diventa vuota, ***append*** restituisce direttamente l2, ***non una sua copia*** -> l1 è duplicata, ma ***l2 rimane condivisa***

Structure sharing (parziale)

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

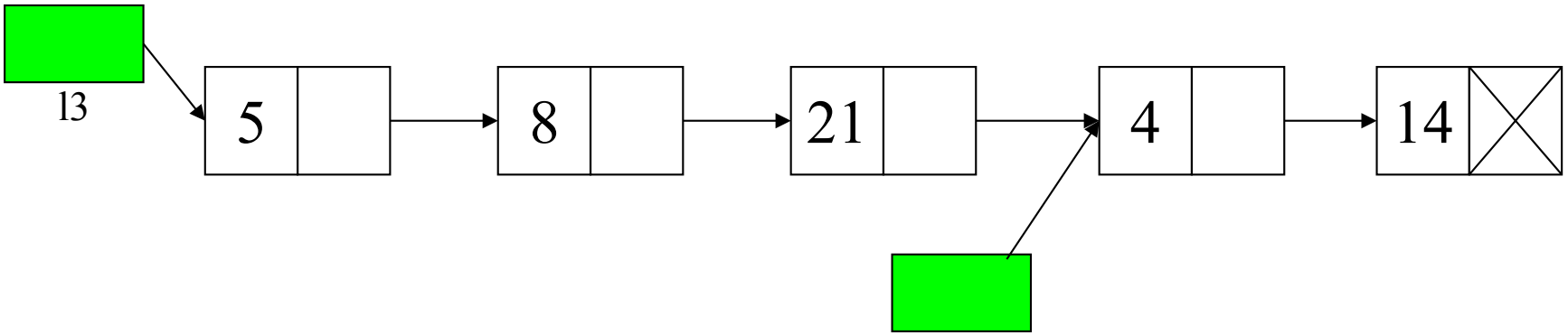


ADT LISTA: la funzione append (*cont.*)



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

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



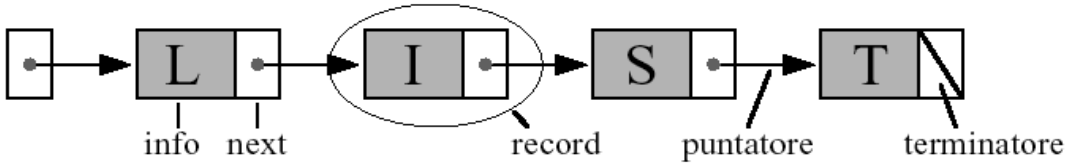
Restituito da **copy(l2)**

ADT LISTA: la funzione reverse

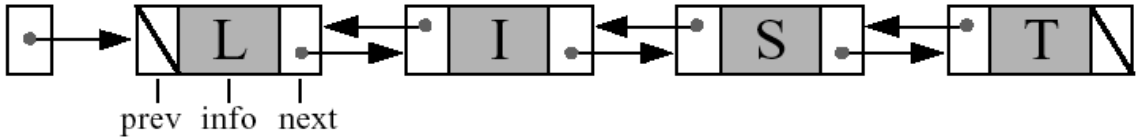
Esercizio: scrivere una versione ***iterativa*** di reverse

```
list reverse_it(list l) {
        list aux=emptyList();

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

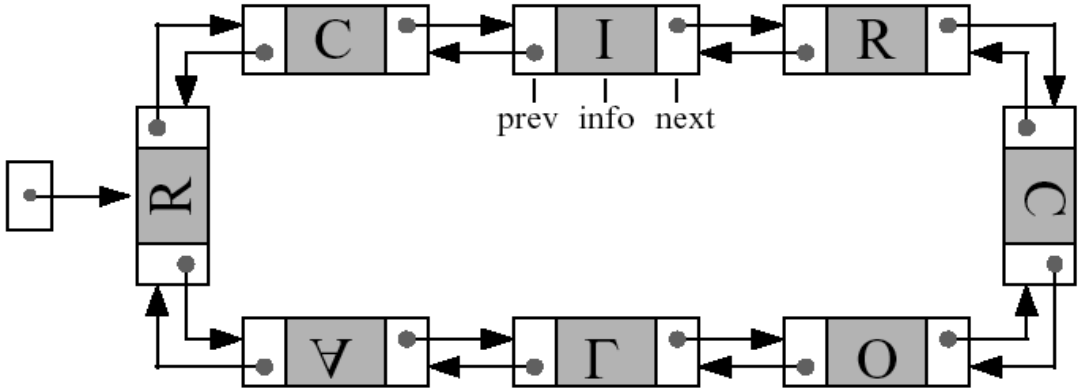
Altri esempi di strutture dati collegate



lista semplice



lista doppiamente collegata



lista circolare doppiamente

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

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

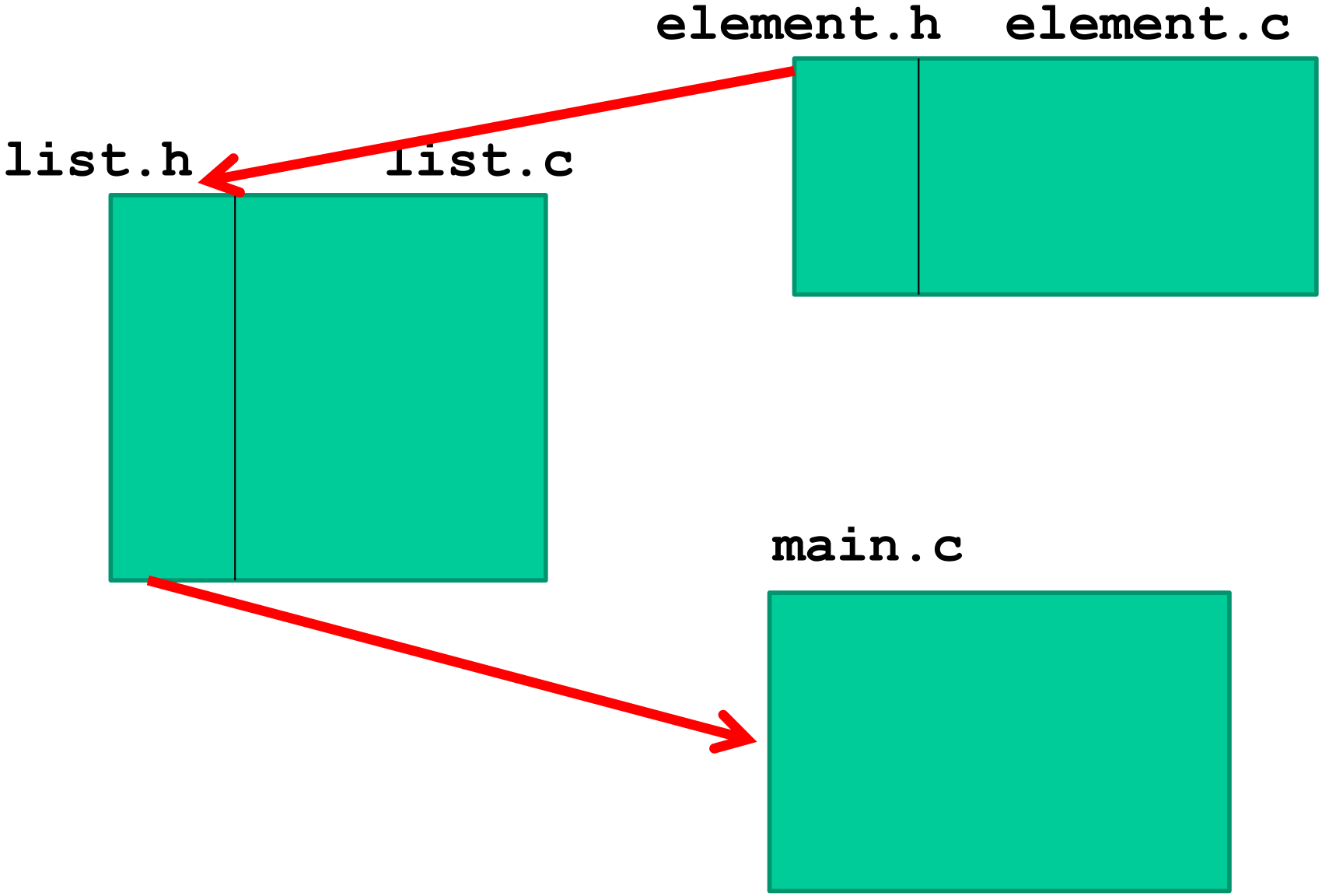
Osservazioni:

- ***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***

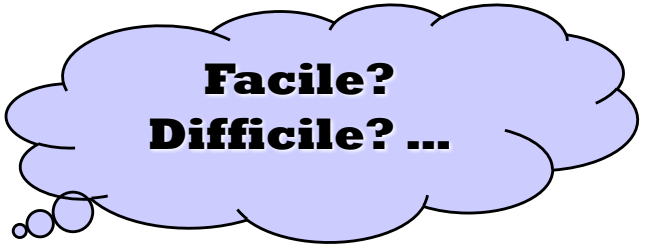
```
#ifndef ELEMENT_H
#define ELEMENT_H

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 el;
      scanf(“%d”, &el);
      return el; }

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

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; }
}
```

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)

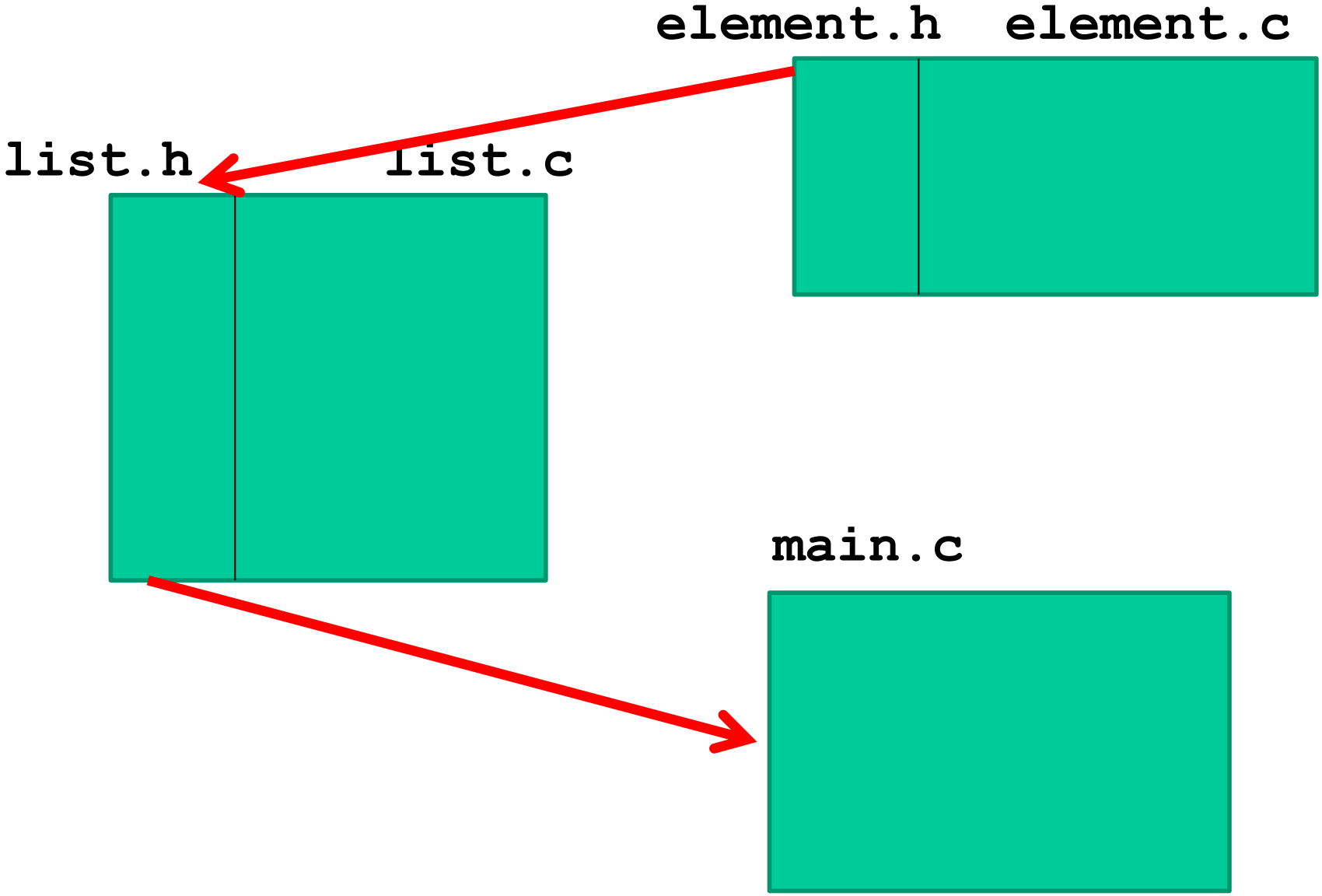
```
#include <stdio.h>
#include "list.h"

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

showList(l1);
}
```

TO DO – Es. 1 LABORATORIO

- Si legga una sequenza di numeri interi da tastiera, dopo ogni inserimento chiedere all'utente se vuole continuare, quindi:
- Creare due liste L1 e L2 con inserimento ordinato;
- Creare una funzione **`append`** che date due liste restituisce una terza lista contenente gli elementi della prima seguiti dagli elementi della seconda; **`list append(list, list)`**
- Creare funzione **`merge`** che date due liste ordinate restituisce una lista ordinata contenente gli elementi delle due liste date (gli elementi possono essere ripetuti); **`list merge, list)`**

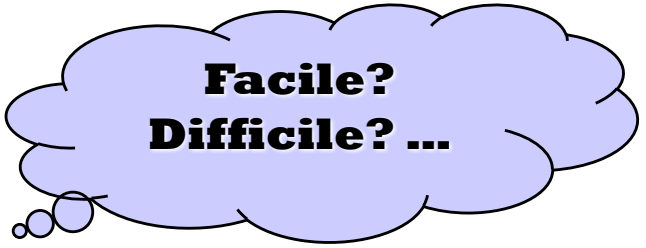
Si astragga il concetto di lista in modo da rendere l'implementazione il più strutturata possibile.

COMPONENTI



TO DO - LABORATORIO

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 ciascuna ***lista con inserimento ordinato,*** concatenare L1 e L2 in L3, e farne il merge in L4, infine stampare le liste L3 e L4 create usando le funzionalità dell'ADT ***element*** e ***list***

Merge di liste ordinate

Realizzare (come non-primitiva) funzione ***mergeList*** che fonda due liste l1 e l2 ordinate in un' unica lista l3 senza ripetizioni

Algoritmo: si copia la lista l1 in una lista l3, poi si scandisce la lista l2 e, elemento per elemento, si controlla se l' elemento è già presente in l3, inserendolo in caso contrario

```
list mergeList(list l1, list l2) {
  list l3 = copy(l1);
  if (empty(l2)) return l3;
  else if (!member(head(l2), l3))
     l3 = insord_p(head(l2), l3);
  return mergeList(l3, tail(l2)); }
```

Realizzare una *versione iterativa di mergeList* (aumentare l' efficienza tenendo conto che nell' analisi del successivo elemento di l2 si può ripartire dall' ultimo elemento di l1 analizzato)