Liste semplici - ADT

Obiettivi:

- Introdurre l'ADT lista semplice e le operazioni tipiche su essa come componente separato
- Vedere altre operazioni su liste

Programmazione modulare

- Suddivisione di un progetto software in parti indipendenti
  - moduli sviluppabili *separatamente*, purché le modalità di interazione siano ben definite (*interfacce*)

- Principio di *Information hiding*
  - ovvero nascondere i dettagli realizzativi, esportando solo – a livello di *interfaccia* – i nomi (gli identificatori) dei tipi e delle operazioni
  - *...*vedremo che in linguaggio C è *parzialmente possibile*

Programmazione modulare in C

Dichiarazioni di funzioni, def. di tipi, etc.

file.h **file.c**

```
#include "file.h"
```

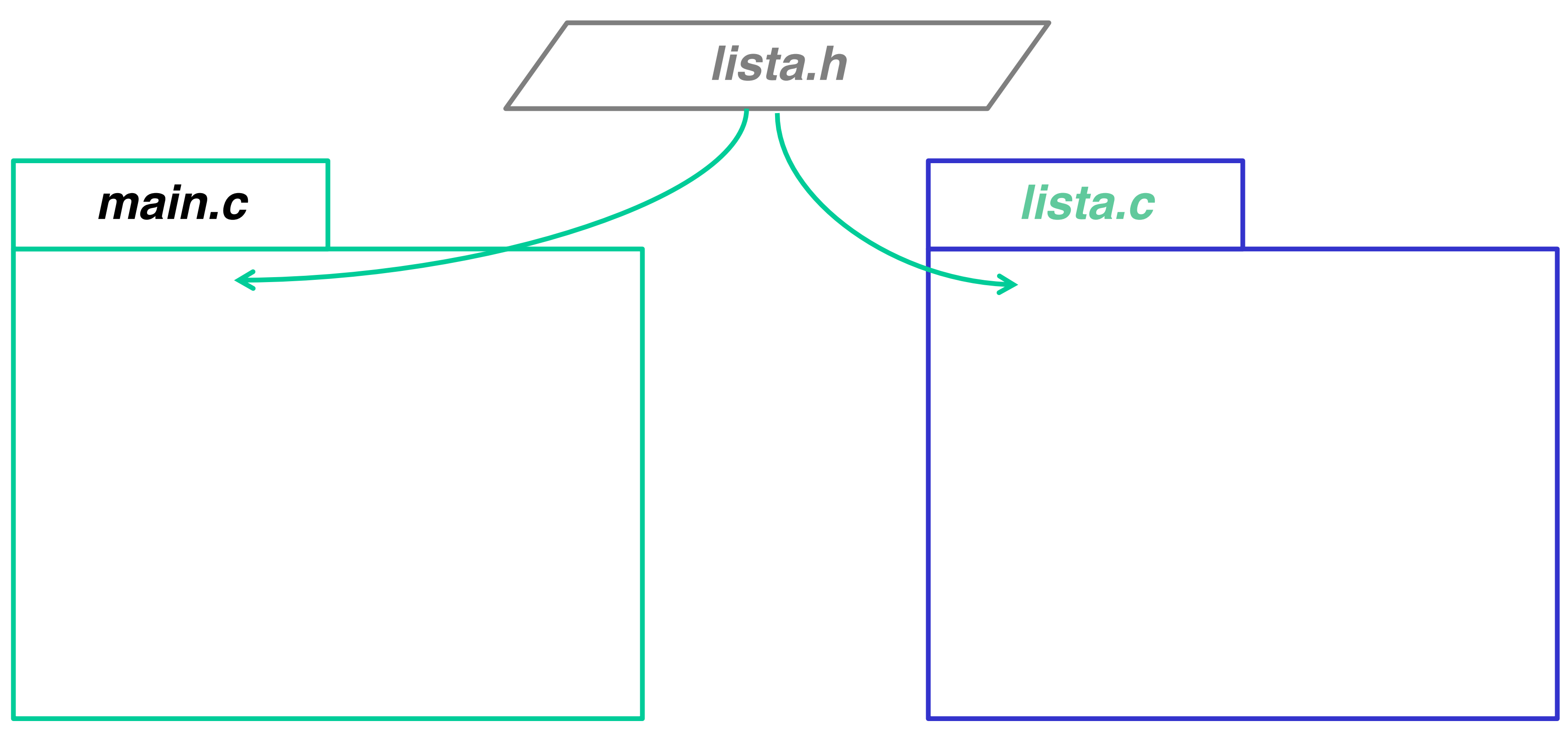
Definizioni di funzioni, identificatori di tipo, variabili, etc

main.c

```
#include "file.h"
```

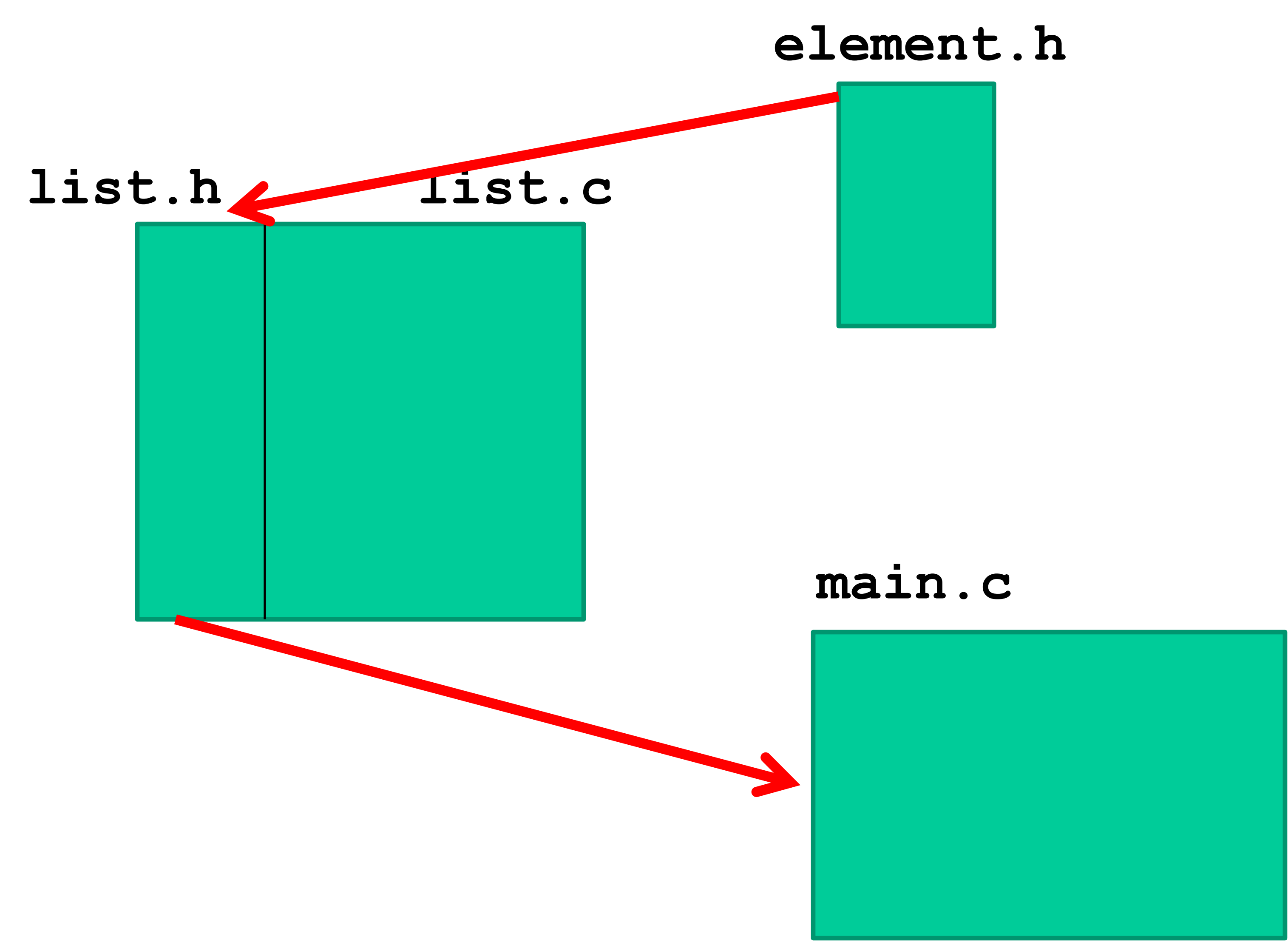
Uso di funzioni, identificatori di tipo, etc.

Tipo di Dato Atratto (ADT) Lista semplice



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

COMPONENTI



COSTRUZIONE ADT LISTA

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;

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

Continua ... (`list.c`)

```
void showList(list l) {                    //      NON
PRIMITIVE
      printf("[");
      while (l!=NULL) {
            printf("%d", l->value);
            l = l->next;
            if (l!=NULL) 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 = NULL;
element 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));
}
```

IL PROBLEMA DELLA GENERICITÀ

Le funzioni che abbiamo scritto per operare su liste sono ancora DIPENDENTI DAL TIPO DEGLI ELEMENTI ...

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

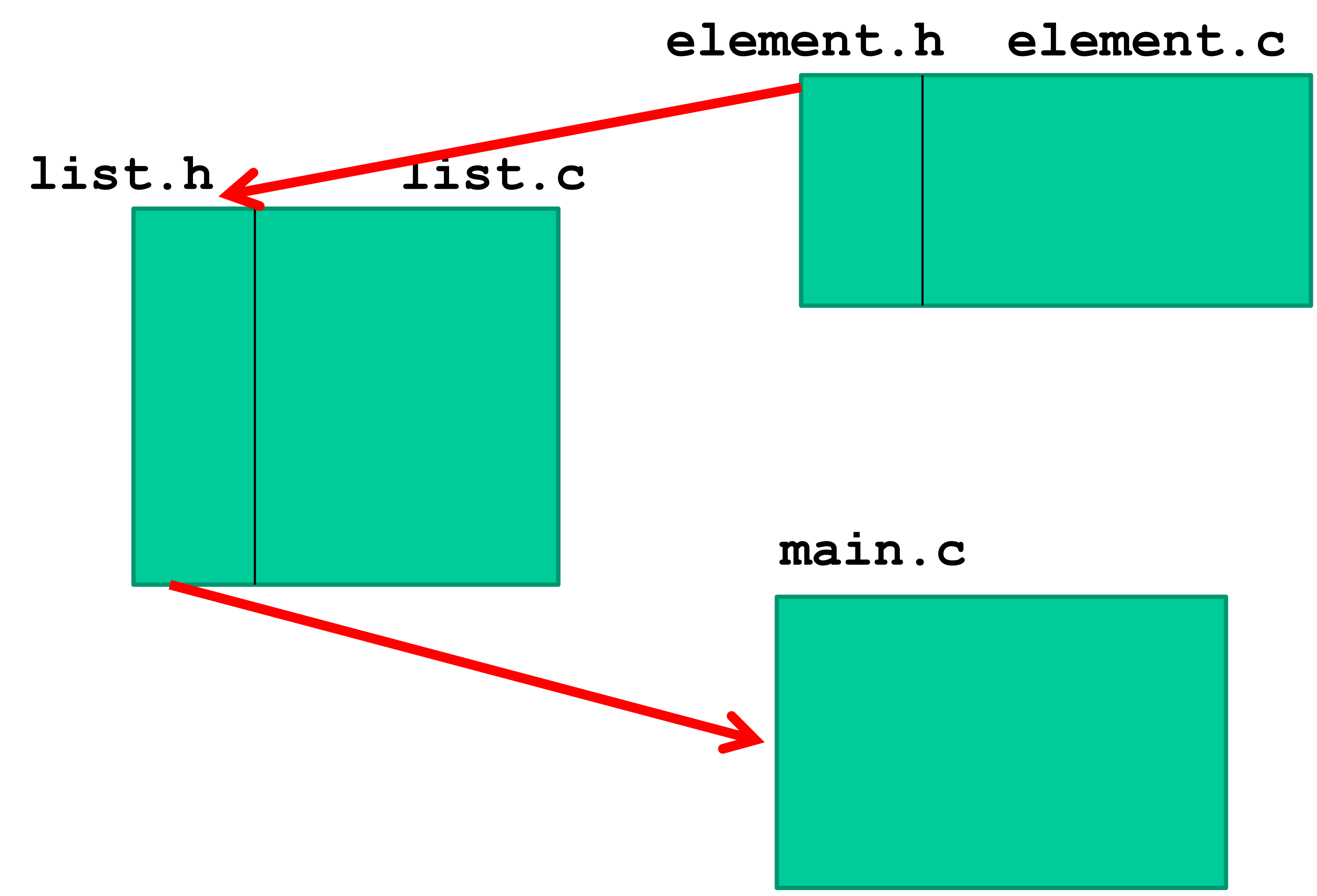
Osservazioni:

- ***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)

Facile?
Difficile? ...

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 (l!=NULL) {
            printf("%d", l->value);
            l = l->next;
            if (l!=NULL) 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 (l!=NULL) {

          printElement(l->value);

          l = l->next;     }
}
```

NOTA: **printElement** stampa su stdout l'elemento in testa ad l, l->value

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

showList(l1);
}
```

Altre operazioni su liste collegate

- Cancellare un elemento

```
list delete(int el, list L)

list delete(element el, list L)
```

- Concatenare due liste

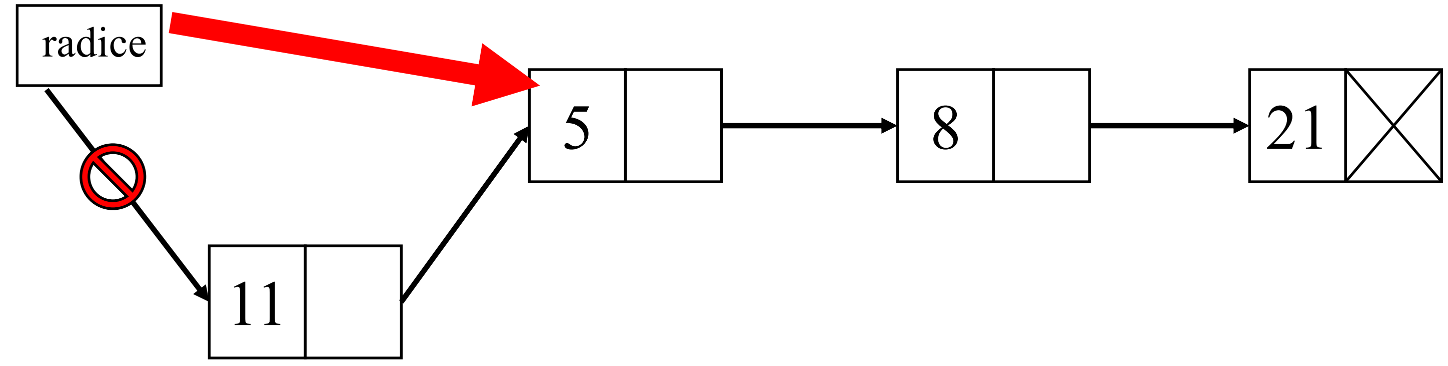
```
list append(list L1, list L2)
```

- Fondere (merge) due liste

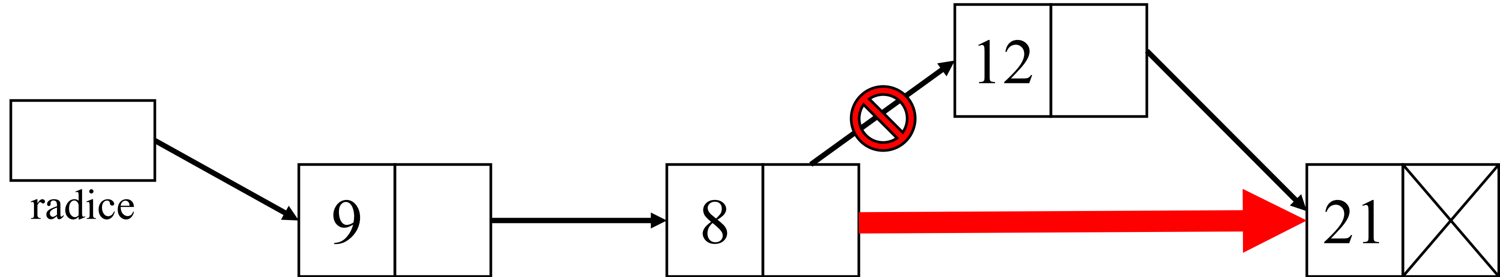
```
list merge(list L1, list L2)
```

Cancellare un nodo da una lista collegata

Primo nodo (in testa o …)



(… in un punto intermedio)



Cancellare un nodo da una lista collegata

Dato un elemento (chiave, o intero nei nostri esempi di liste di interi), cancellare la prima occorrenza dell'elemento dalla lista:

1) Trovare il nodo da cancellare, se è in testa alla lista basta aggiornare il puntatore radice

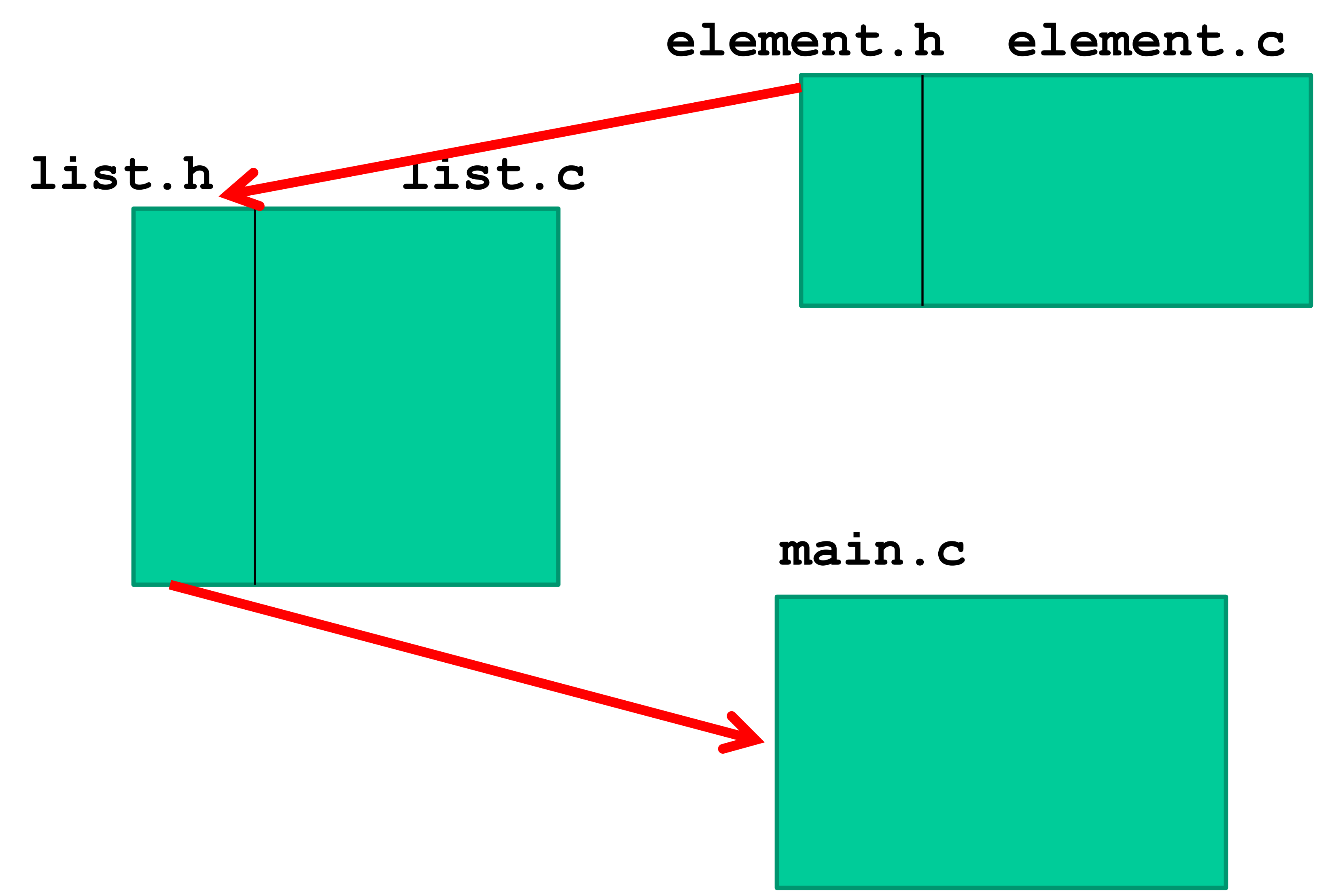
2) Se non è quello in testa alla lista, modificare il campo *next* del nodo predecessore

3) Rilasciare sempre la memoria occupata dal nodo da cancellare

Se non trovo alcun elemento uguale a quello da cancellare, la lista non cambia

```c
list delete(int el, list L)
{ int trovato = 0;
  list aux=L, prev=NULL;
  if (aux != NULL)
    if (aux->value==el)
       { L = aux->next;
         free(aux);  }
    else
    { while ((aux != NULL) && (!trovato))
         if (aux->value==el) trovato=1;
         else
           { prev = aux;
             aux = aux->next;  }
      if (aux != NULL)
         { prev->next = aux->next;
           free(aux);} }
  return L;  //il return qui per tutti i casi
}
```

COMPONENTI



```
list delete(int el, list L)
{ int trovato = 0;
  list aux=L, prev=NULL;
  if (aux != NULL)
    if (aux->value==el)
      { L = aux->next;
        free(aux);  }
    else
    { while ((aux != NULL) && (!trovato))
        if (aux->value==el) trovato=1;
        else
          { prev = aux;
            aux = aux->next;  }
      if (aux != NULL)
        { prev->next = aux->next;
          free(aux);} }
  return L;  //il return qui per tutti i casi
}
```

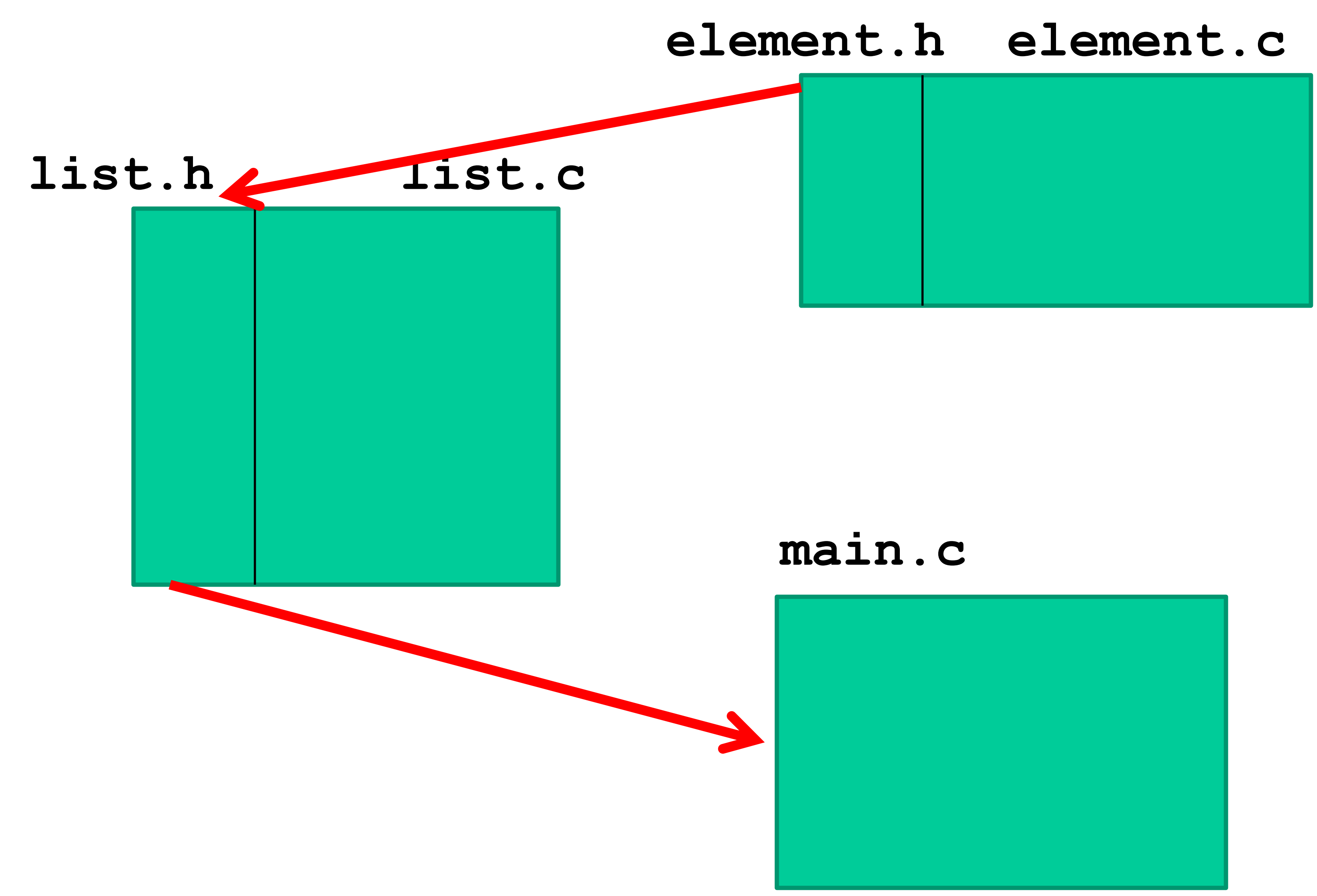
```
list delete(element el, list L)
{ boolean trovato = false;
  list aux=L, prev=NULL;
  if (aux != NULL)
    if (isEqual(aux->value,el))
       { L = aux->next;
         free(aux);  }
    else
    { while ((aux != NULL) && (!trovato))
         if (isEqual(aux->value,el)) trovato=true;
          else
            { prev = aux;
              aux = aux->next;  }
        if (aux != NULL)
          { prev->next = aux->next;
            free(aux);} }
  return L;  //il return qui per tutti i casi
}
```

✐ TO DO – Es. 4 (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, list)`**

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

COMPONENTI



✎ TO DO - Esercizio

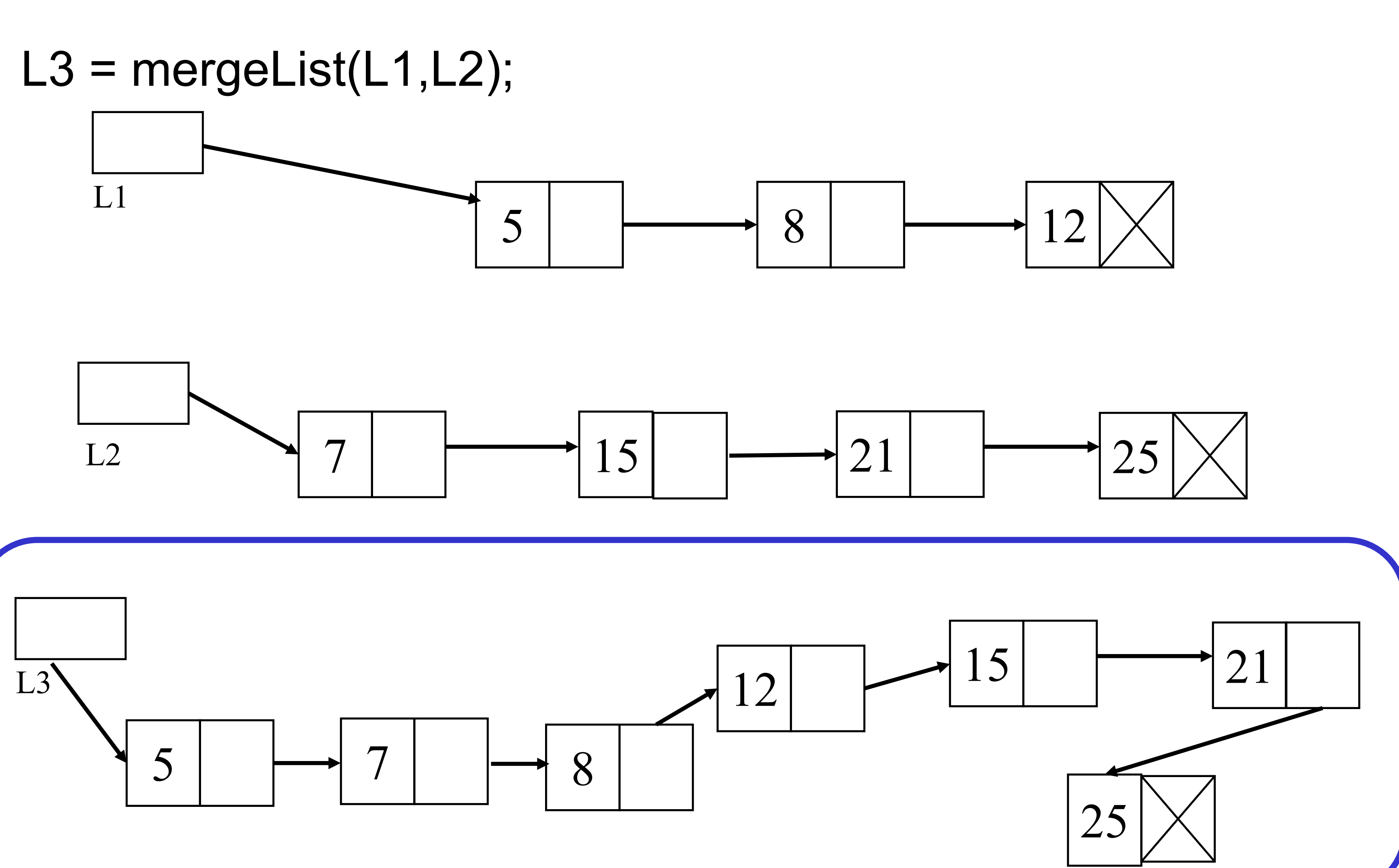
Si ***definiscano i file element.c e element.h*** che realizzano l'ADT ***element*** (come intero)

Facile?
Difficile? ...

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 (append)** 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***

✎ TO DO: Merge di due liste (ordinate)



Merge di liste ordinate

mergeList fonde due liste l1 e l2 ordinate in un' unica lista l3, con ripetizioni

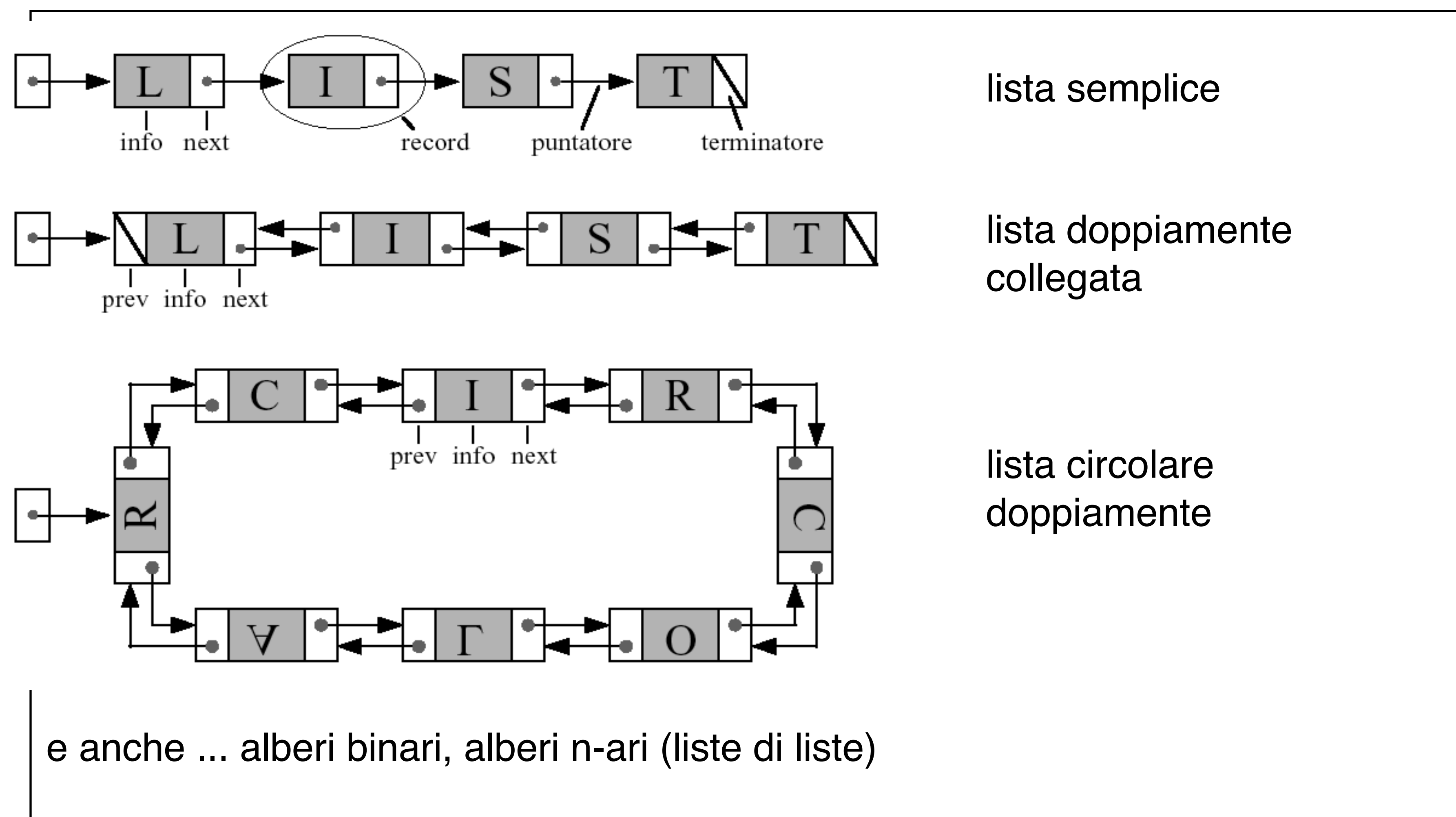
```
list mergeList(list l1, list l2) {
  list l3=NULL:
  while ((l1!=NULL) && (l2!=NULL))
     if ( isLess(l1->value,l2->value) )
        { l3 = cons_tail(l1->value,l3);
          l1 = l1->next; }
     else
        { l3 = cons_tail(l2->value,l3);
          l2 = l2->next; }
  while (l1!=NULL)
       { l3 = cons_tail(l1->value,l3);
         l1 = l1->next; }
  while (l2!=NULL)
       { l3 = cons_tail(l2->value,l3);
         l2 = l2->next; }
  return l3;
}
```

Merge di liste ordinate

mergeList fonde due liste l1 e l2 ordinate in un' unica lista l3, con ripetizioni

```
list mergeList(list l1, list l2) {
  list l3=NULL:
  while ((l1!=NULL) && (l2!=NULL))
     if (l1->value < l2->value )
        { l3 = cons_tail(l1->value,l3);
          l1 = l1->next; }
     else
        { l3 = cons_tail(l2->value,l3);
          l2 = l2->next; }
  while (l1!=NULL)
       { l3 = cons_tail(l1->value,l3);
         l1 = l1->next; }
  while (l2!=NULL)
       { l3 = cons_tail(l2->value,l3);
         l2 = l2->next; }
  return l3;
}
```

Altri esempi di strutture dati collegate



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