IL CONCETTO DI LISTA

Una lista è una sequenza (cioè un multi-insieme finito e ordinato) di elementi di un certo tipo.

Per denotare strutture a lista useremo la seguente notazione "a parentesi quadre":

L = [*elemento*, *elemento*, ... , *elemento*]

Ad esempio,

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

[5,8,5,21,8] denota una lista di cinque interi.

Una lista è un ***multi-insieme***, ossia un insieme in cui un medesimo elemento può comparire più volte.

Come ogni tipo di dato astratto, anche la lista è definita in termini di

- dominio dei suoi elementi (dominio-base)
- operazioni di ***costruzione*** sul tipo lista
- operazioni di ***selezione*** sul tipo lista

Nei due esempi sopra, il dominio base è, rispettivamente, l'insieme dei caratteri e quello degli interi.

L'ADT LISTA

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

- un dominio-base, D
- un insieme di funzioni $\Im = \{F_1,...,F_n\}$ sul dominio D
- un insieme di predicati $\prod = \{P_1,...,P_m\}$ sul dominio D

$$\mathbf{T = \{ D, \ \Im, \ \prod \}}$$

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

- **D** può essere qualunque
- $\Im$ = { cons, head, tail, emptylist }

  cons: $D \times list \rightarrow list$ (*costruttore*)

  head: $list \rightarrow D$ (*selettore* "*testa*")

  tail: $list \rightarrow list$ (*selettore "coda*")

  emptylist: $\rightarrow list$ (costante "*lista vuota*")
- $\prod$ = { empty }

  empty: $list \rightarrow boolean$ (test di lista vuota)

L'ADT LISTA (II)

ESEMPI

head([6,7,11,21,3,6]) $\rightarrow$ 6

tail([6,7,11,21,3,6]) $\rightarrow$ [7,11,21,3,6]

cons(6, [7,11,21,3,6]) $\rightarrow$ [6,7,11,21,3,6]

empty([6,7,11,21,3,6]) $\rightarrow$ false

empty([ ]) $\rightarrow$ true

Pochi linguaggi forniscono il tipo lista fra quelli predefiniti (LISP, Prolog).
Per gli altri, l'ADT lista si costruisce a partire da altre strutture dati (in C, tipicamente *vettori* o *puntatori*).

LE OPERAZIONI PRIMITIVE DA REALIZZARE

operazione	*descrizione*
cons : $D \times list \rightarrow list$	Costruisce una nuova lista, aggiungendo in testa alla lista data l'elemento fornito
head : $list \rightarrow D$	Restituisce il primo elemento della lista data
tail : $list \rightarrow list$	Restituisce la coda della lista data
emptylist : $\rightarrow list$	Restituisce la lista vuota
empty : $list \rightarrow boolean$	Restituisce *vero* se la lista data è vuota, *falso* altrimenti

L'ADT LISTA (III)

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 un lista, ribaltamento di una lista, etc. - si possono definire in termini delle primitive precedenti.

UNA RIFLESSIONE

Il tipo *list* è definito in modo induttivo:

- **esiste la costante "lista vuota"**
  (ottenibile dalla funzione *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.*

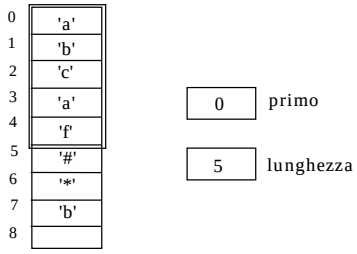
RAPPRESENTAZIONE CONCRETA DI LISTE

1) RAPPRESENTAZIONE *STATICA*

Una prima forma di rappresentazione, molto "banale", consiste nell'utilizzare un **vettore** per memorizzare gli elementi della lista uno dopo l'altro (***rappresentazione sequenziale***).

- La variabile ***primo*** memorizza l'indice del vettore in cui è inserito il primo elemento.
- La variabile ***lunghezza*** indica da quanti elementi è composta la lista.

ESEMPIO: ['a','b','c','a','f']



Le componenti del vettore con indice pari o successivo a (*primo+lunghezza*) non sono significative.

Inconvenienti:

- le *dimensioni del vettore sono fisse*
- sono *dispendiose le operazioni di inserimento e cancellazione* (si copia l'intera struttura dati).

ESEMPIO – Liste di caratteri come vettori

DICHIARAZIONE PRIMITIVE (`listVet.h`)

```
#define N 100

typedef struct {  int primo, lunghezza;
                  char El[N];
          }  list;

typedef int boolean;

list emptylist();
boolean empty(list);
char head(list);
list tail(list);
list cons(char, list);
```

IMPLEMENTAZIONE PRIMITIVE (`listVet.c`)

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

list  emptylist() {
  list l;
  l.primo = -1;    l.lunghezza=0;
  return l;
}

boolean empty(list l) {
  return (l.primo == -1);
}

char head(list l) {
  if (empty(l)) abort();
  else return (l.El[l.primo]);
}
```


ESEMPIO – Liste di caratteri come vettori (II)

IMPLEMENTAZIONE PRIMITIVE (`listVet.c`) (segue)

```
list tail(list l) {
  /* copia il vettore */
  list t=l;
  if (empty(l)) abort();
  else { t.primo++; t.lunghezza--; return t; }
}

list  cons(char e, list l){
  list t;
  int i;
  t.primo=0;   t.lunghezza=1;
  t.El[t.primo]=e;
  for(i=1; i<=l.lunghezza; i++) {
     t.El[i]=l.El[i-1]; t.lunghezza++;
  }
  return t;
}
```

DICHIARAZIONE NON PRIMITIVE (`listVet2.h`)

```
void showlist(list);
```

IMPLEMENTAZIONE (come primitiva!) (`listVet2.c`)

```
void showlist(list l) { /* VER. ITERATIVA */
  int i;
  printf("[");
  for (i==l.primo; i<l.lunghezza; i++){
     printf("%c",l.El[i]);
     if (i<l.lunghezza-1) printf(",\t"); }
  printf("]\n");
}
```

Esercizio: implementare `showlist()` come *non-primitiva*, evitando accesso diretto alla rappresentazione fisica, e usando invece solo le primitive sopra definite.

RAPPRESENTAZIONE CONCRETA DI LISTE

2) RAPPRESENTAZIONE *COLLEGATA*

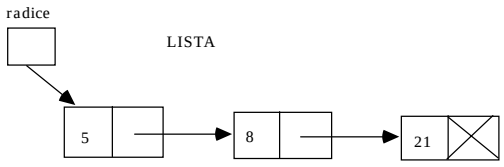
In questa seconda forma di rappresentazione, a ogni elemento si associa l'informazione ("indice", "riferimento") che permette di individuare la posizione dell'elemento successivo (***rappresentazione collegata***).

⇒ La sequenzialità degli elementi della lista non è più rappresentata mediante l'adiacenza delle locazioni di memoria in cui sono memorizzati.

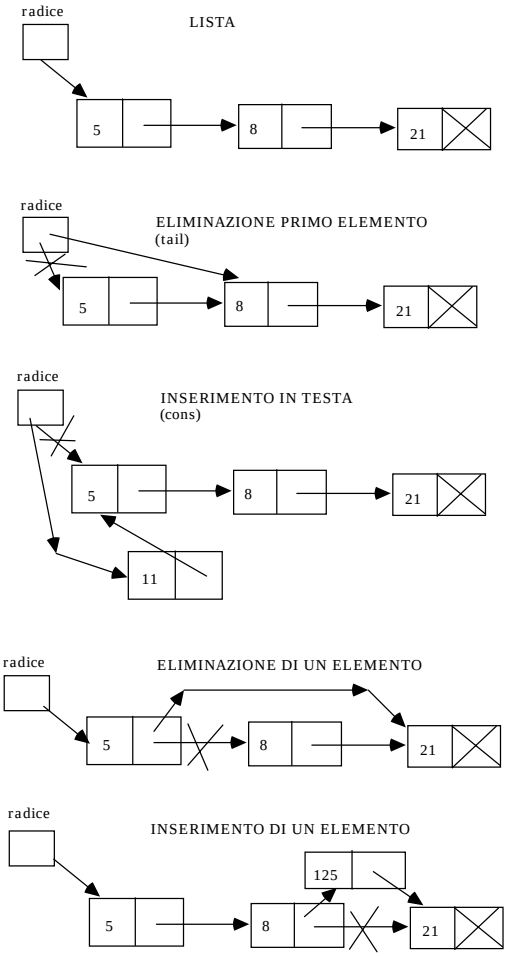
NOTAZIONE GRAFICA

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

Ad esempio, la lista [5, 8, 21] risulta così rappresentata:



La figura seguente illustra a livello di principio come possono essere realizzare le varie operazioni.



RAPPRESENTAZIONE COLLEGATA

IMPLEMENTAZIONE MEDIANTE VETTORI

Ogni elemento del vettore deve mantenere:

- il valore dell'elemento della lista (dato)
- un riferimento (indice) al prossimo elemento (next).

VANTAGGI

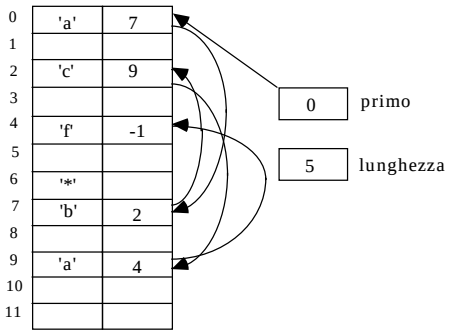
Cancellazione e inserimento sono *più efficienti*, in quanto non richiedono lo spostamento fisico di elementi (ma solo il "riaggiustamento" degli indici o riferimenti).

SVANTAGGI

Si occupa più spazio senza superare il problema dell'esistenza di una dimensione massima della lista.

Occorre inoltre gestire la *lista libera*, cioè le componenti libere del vettore (per inserimento e cancellazione).

Esempio: ['a','b','c','a','f']



RAPPRESENTAZIONE COLLEGATA

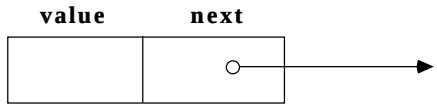
IMPLEMENTAZIONE MEDIANTE PUNTATORI

Per ovviare al problema dell'esistenza di una dimensione massima del vettore occorre adottare un approccio basato sull'*allocazione dinamica* della memoria.

Ciascun nodo della lista è una struttura di due campi:

- **il *valore* dell'elemento**
- **un *puntatore* al nodo successivo** della lista (NULL nel caso dell'ultimo elemento).

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



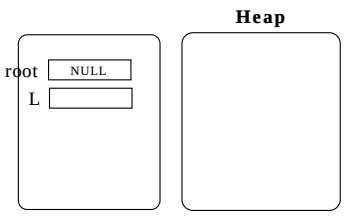

```
typedef node* list;
```

NOTA:

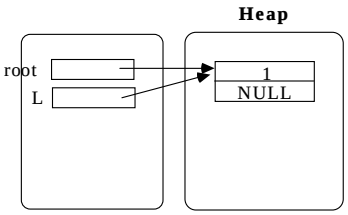
per la prima volta ***l'etichetta*** (`list_element`) nella dichiarazione della `struct` **è indispensabile**, altrimenti sarebbe impossibile definire un tipo ricorsivamente (in termini di se stesso).

ESERCIZIO 1

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

typedef item *list;

main(){
   list root=NULL, L;
```



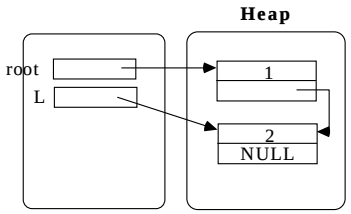

```
   root = (list) malloc(sizeof(item));
   root->value = 1;
   root->next = NULL;
   L = root;
```



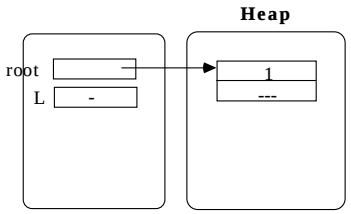
/ segue main */*

ESERCIZIO 1 (II)

```
    /* segue main */

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




```
    free(L);
```




```
}
```

ATTENZIONE: la `free()` in presenza di *structure sharing* è molto pericolosa!

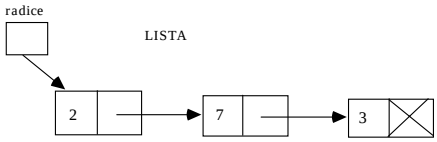
ESERCIZIO 2

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) {
    /* funzione di inserimento in testa */
   list t;
    t=(list)malloc(sizeof(item));
    t->value=e;
    t->next=l;
    return t ;
}

main(){
   list root=NULL, l;
    int i;

    do {
        printf("\n Introdurre valore:\t");
        scanf("%d", &i);
        root = insert(i, root);
    } while (i!=0);

    l=root;        /* stampa */

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


ESERCIZIO 2 (II)

Ricerca in una lista




```
int ricerca(int e, list l) {

    /* funzione di ricerca - iterativa */

   int trovato=0;

    while ((l!=NULL) && !trovato)
        if (l->value == e) trovato=1;
        else l=l->next;

   return trovato;
}
```

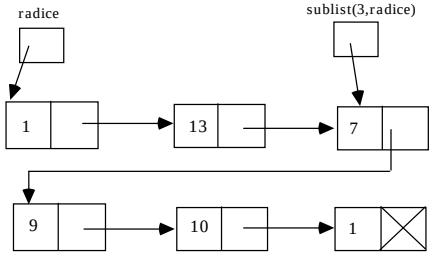
E' una scansione sequenziale
→ nel caso peggiore, occorre scandire l'intera lista
→ nel caso migliore, è il primo elemento

Esercizio: scriverne una versione ricorsiva.

ESERCIZIO 3

Definire una funzione **`sublist`** che, dato un intero positivo `n` e una lista **`l`**, restituisca una lista che rappresenti la sotto-lista di quella data a partire dall' n-esimo elemento.

ESEMPIO: **`newList = sublist(3, radice)`**



Versione ***iterativa***:

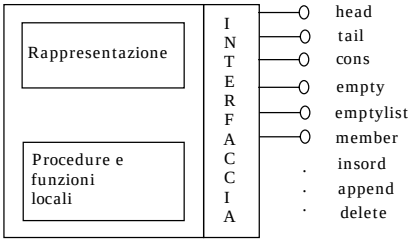
```
list sublist(int n, list l) {
   int i=1;
   while ((l!=NULL)&&(i<n)) { l = l->next; i++; }
   return l;
}
```

Versione ***ricorsiva***:

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

COSTRUZIONE DELL'ADT "LISTA"

Anche per la realizzazione di questa struttura dati conviene incapsulare la ***rappresentazione concreta*** (che utilizza puntatori e strutture) ed esportare, sotto forma di file header, solo le *definizioni di tipo* e le *dichiarazioni delle operazioni*.



Poiché il funzionamento di una lista **non dipende dal tipo degli elementi di cui è composta**, cerchiamo di impostare una soluzione *generica*.

IDEA:

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

IL TIPO `element`

Il file `element.h` contiene la definizione di tipo:

```
typedef  int  element;
```

(Il file `element.c` per ora non è necessario.)

L'ADT "LISTA"

SCELTA DI PROGETTO:

poiché l'uso di `free()` in presenza di *structure sharing* è pericoloso, *non deallocheremo* la memoria allocata.

- *inefficiente in termini di spazio occupato* (lo heap si riempie sempre più!!) *a meno che non ci sia un garbage collector* (Java, Prolog, Lisp)
- *ma sicura* perché garantisce che *non ci saranno mai effetti collaterali* sulle strutture condivise.

FILE HEADER (`list.h`)

```
#include "element.h"

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

typedef  item* list;

/* ---- PRIMITIVE ---- */

list    emptylist(void);
boolean empty(list);
element head(list);
list    tail(list);
list    cons(element, list);

/* ---- NON PRIMITIVE ---- */

void    showlist(list);
boolean member(element, list);
...
```


L'ADT "LISTA" (II)

FILE IMPLEMENTAZIONE (`list.c`)

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

/* ---- PRIMITIVE ---- */

list  emptylist(void) { return NULL; }

boolean  empty(list l) { return (l==NULL); }

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;
}
```

(segue implementazione non primitive)

L'ADT "LISTA" (III)

FILE IMPLEMENTAZIONE (`list.c`) *(segue)*

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

/* ---- NON PRIMITIVE ---- */

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

Critica: `printf("%d",...)` è *specifica per gli interi !!*

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);   /* condiz. arbitraria */

 showlist(l1);
}
```

L'ADT "LISTA" (IV)

ALTRE OPERAZIONI (NON PRIMITIVE)

operazione		*descrizione*
member :	D × 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 × list → list	Restituisce una lista che è il 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 una copia della lista data

L'ADT "LISTA" (V)

L'OPERAZIONE 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 (TAIL) 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) );
}
```

Nota: la funzione ricorsiva è una trascrizione pari-pari della specifica data in alto → codifica *di estrema semplicità.*

Esercizio: scriverne una versione *primitiva* per maggiore efficienza.

L'ADT "LISTA" (VI)

L'OPERAZIONE 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, perché la somma viene eseguita *dopo* la chiamata ricorsiva.

Esercizio: scriverne una versione *primitiva* per maggiore efficienza.

L'ADT "LISTA" (VII)

L'OPERAZIONE append

L'operazione *append* (come anche le successive *copy* e *reverse*) non è più solo un'operazione di *analisi* del contenuto o della struttura della lista data (come le precedenti), ma implica la ***"sintesi" di una nuova lista.***

Per ottenere una lista che sia il concatenamento di due liste date L1 e L2 si può ragionare così:

- se la lista L1 è vuota, il risultato è semplicemente 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* !

↓

il solo modo per ottenere una lista diversa è *costruirsene una nuova.*

Dunque, per aggiungere in coda a L1 la lista L2 dovremo **costruire una nuova lista** avente

- **come primo elemento (testa), *la testa della lista L1***
- **come coda, *una nuova lista* ottenuta appendendo L2 alla coda di L1.**

Il secondo punto implica evidentemente una chiamata ricorsiva ad *append* medesima, nel caso più semplice in cui L1 sia ridotta alla sua coda (e sia quindi più corta).

L'ADT "LISTA" (VIII)

L'OPERAZIONE append *(segue)*

La definizione di `append` può quindi essere data così:

append(L1,L2) =
L2, se empty(L1)
cons(head(L1), append(tail(L1), L2)) *altrimenti*

CODIFICA

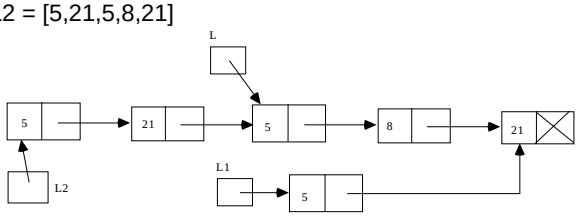
```
list append (list l1, list l2) {
  if (empty(l1)) return l2;
  else
  return cons(head(l1),append(tail(l1),l2));
}
```

IMPORTANTE
quando L1 diventa vuota, `append` restituisce direttamente L2, *non una sua copia !*
→ L1 viene duplicata, ma L2 *rimane condivisa*

Si ha dunque **structure sharing (*parziale*):** ad esempio,

L = [5,8,21] L1 = [5,21] L2=append(L1,L)

L2 = [5,21,5,8,21]



L'ADT "LISTA" (IX)

L'OPERAZIONE reverse

Per ottenere una lista che sia il 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) → funzione `append()`

Peraltro, la `append()` richiede *due liste*, non una lista e un elemento → occorre prima costruire una lista L2 contenente il solo elemento di cui al punto (2).

Dunque:

reverse(L) =
emptylist(), se empty(L)
append(reverse(tail(L)),
cons(head(L),emptylist())), altrimenti

CODIFICA

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


L'ADT "LISTA" (X)

L'OPERAZIONE reverse *(segue)*

Esercizio: scriverne una versione *tail ricorsiva*.

Come sempre, per passare da una struttura ricorsiva a una ricorsiva tail occorre:

- predisporre una funzione di interfaccia che chiami la funzione tail-ricorsiva con i necessari parametri extra
- 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.

Dunque:

```
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));
}
```


L'ADT "LISTA" (XI)

L'OPERAZIONE copy

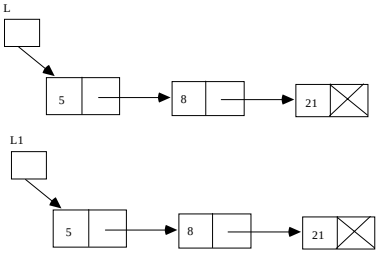
Dato il tipo di operazione, *non può esservi condivisione di strutture* (altrimenti non si copierebbe nulla...).

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

CODIFICA

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

ESEMPIO L1=copy(L)



L'ADT "LISTA" (XII)

L'OPERAZIONE delete

Concettualmente, deve restituire una lista che differisce da quella data solo per l'assenza dell'elemento indicato.

Ancora una volta, non esistendo operatori di modifica, `delete` *deve operare costruendo una nuova lista (almeno per la parte da modificare)*

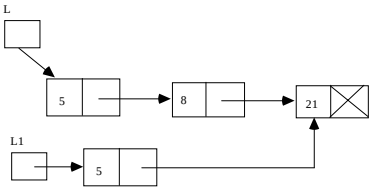
Più esattamente, occorre:

- duplicare tutta la parte iniziale della lista, fino all'elemento da sopprimere (escluso)
- agganciare la lista così creata alla parte rimanente della lista data.

CODIFICA

```
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)));
}
```

ESEMPIO L1= delete(L,8)

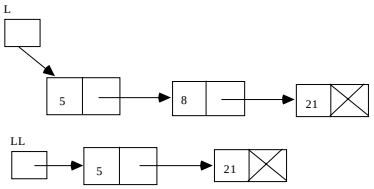


L'ADT "LISTA" (XIII)

L'OPERAZIONE delete *(segue)*

Per non avere condivisione:

```
LL = delete(copy(L),8)
```



Esercizio: scrivere una versione *iterativa*

CONCLUSIONE

Per usare *in modo sicuro* la condivisione di strutture (*structure sharing*) è stato necessario:

- **Non effettuare mai alcuna `free()`** → *uso inefficiente dello heap* in un linguaggio, come il C, privo di garbage collector.

  Ciò evita il rischio di riferimenti pendenti.
- **Realizzare *liste* come *valori* (entità non modificabili)**: ogni modifica comporta così la creazione di una nuova lista.

  Ciò evita però il rischio di effetti collaterali indesiderati, giacché non sarà mai possibile alterare una parte della struttura condivisa.

LISTE ORDINATE

Per poter anche solo parlare di ***liste ordinate***, è necessario che sia definita una *relazione d'ordine sul dominio-base* degli elementi della lista.

IMPORTANTE:
il criterio di ordinamento dipende *dal dominio base* e *dalla specifica necessità applicativa*!

Ad esempio:

- gli interi possono essere ordinati *in senso crescente*, oppure *in senso decrescente*, etc
- le stringhe possono essere ordinate *in ordine alfabetico*, oppure *in base alla loro lunghezza*, etc.
- le persone possono essere ordinate in base all'*ordinamento alfabetico del loro cognome*, oppure in base all'*età*, oppure *in base al codice fiscale*, oppure...

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

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

Questa funzione presuppone l'esistenza di una funzione `insord()` che effettui l'inserimento ordinato di un elemento in una lista data.

LISTE ORDINATE (II)

La funzione `insord()`

Per inserire un elemento in modo ordinato in una lista (*che a sua volta si suppone ordinata)*, si può ragionare così:

- se la lista data è vuota, basta costruire una nuova lista contenente il nuovo elemento

  altrimenti
- se l'elemento da inserire è minore del primo elemento (testa) della la lista data, basta 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, facilmente esprimibili con il costruttore `cons()`.

Il terzo caso riconduce il problema *allo stesso problema in un caso più semplice*, perché la coda di una lista è una lista più corta di quella data: perciò, alla fine si arriverà a poter effettuare o un inserimento in testa, o alla lista vuota.

Quindi:

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

LISTE ORDINATE (III)

La funzione `insord()`

Ancora una volta, 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 dovremo **costruire una nuova lista** avente

- **come primo elemento (testa),** ***la testa della lista data***
- **come coda,** ***la coda modificata*** **ottenuta inserendo in essa il 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)));
}
```

Di conseguenza:

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

IL PROBLEMA DELLA GENERICITÀ

Come si è detto all'inizio, il funzionamento di una lista **non dipende - in linea di principio - dal tipo degli elementi di cui è composta**.

Quindi, è ragionevole cercare di costruire un ADT *generico*, che funzioni con *qualunque tipo di elementi* (o quasi...)

A questo fine, abbiamo introdotto l'ADT ausiliario `element`, realizzando poi l'ADT `lista` in termini di esso.

In particolare, osserviamo che:

- `showList()` dipendeva da una `printf()` che svelava il tipo dell'elemento (e la rendeva *dipendente* da esso)
- `insord()` dipende dal tipo dell'elemento *unicamente nel momento del confronto*.

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

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

Le ultime due corrispondono al *modello di coordinazione di `element`*, che ovviamente solo `element` stesso può definire.

L'ADT `element`

Lo header `element.h` deve contenere:

- la **definizione del tipo `element`**
- le **dichiarazioni** delle varie funzioni fornite

Poiché contiene una *definizione*, tale header dovrà essere opportunamente protetto dal problema delle inclusioni multiple nei modi a suo tempo discussi.

`element.h`

```
#ifndef ELEMENT_H
#define ELEMENT_H

 typedef int  element; /* DEFINIZIONE */

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

#endif
```

OSSERVAZIONI:

- a parte la `typedef`, il file header *è invariante rispetto al tipo* effettivo *dell'elemento*
- il file di implementazione, invece, dovrà essere *adattato caso per caso al tipo effettivo dell'elemento considerato.*

L'ADT `element` (II)

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

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

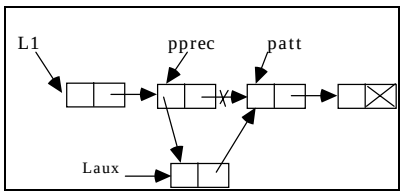
ESERCIZIO 4

Scrivere una versione primitiva della funzione `insord()`.

HP: la lista di partenza è ordinata (e tale deve restare)

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).



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; }
}
```


ESERCIZIO 5

Realizzare (come non-primitiva) la funzione `mergeList()` che, date due liste A e B ordinate, le fonda in un'unica lista C priva di ripetizioni.

Algoritmo:

Si copia la prima lista A in una lista C, poi si scandisce la seconda lista B e, elemento per elemento, si controlla se l'elemento è già presente in C, inserendolo in caso contrario.

```
list mergeList(list A, list B) {
  list C;
  C=copy(A);
  if  (empty(B)) return C;
  else
  if (!member(head(B),C))
        C=insord_p(head(B),C);
  return mergeList(C,tail(B));
}
```

Esercizio:

Realizzarne una versione iterativa. Cercare di aumentare l'efficienza tenendo conto che nell'analisi del successivo elemento di B si può ripartire dall'ultimo elemento di A analizzato.

Esercizio:

Si leggano a terminale alcuni nomi (al massimo di 30 caratteri) e li si inserisca in una lista ordinata alfabeticamente. Si stampi, infine, tale lista.