

IL CONCETTO DI LISTA

Una lista è una sequenza (cioè un multi-insieme finito e ordinato) di elementi tutti dello stesso tipo

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

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

$L = [\text{elemento}, \text{elemento}, \dots, \text{elemento}]$

Ad esempio,

$['a', 'b', 'c']$ denota la lista dei caratteri 'a', 'b', 'c'
 $[5, 8, 5, 21, 8]$ denota una lista di cinque interi

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

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

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

ADT LISTA

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

- un **dominio-base** D
- un insieme di **funzioni** $\mathcal{S} = \{F_1, \dots, F_n\}$ sul dominio D
- un insieme di **predicati** $\Pi = \{P_1, \dots, P_m\}$ sul dominio D

$T = \{ D, \mathcal{S}, \Pi \}$

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

- D può essere qualunque
- $\mathcal{S} = \{ \text{cons}, \text{head}, \text{tail}, \text{emptylist} \}$
 - cons: $D \times \text{list} \rightarrow \text{list}$ (costruttore)
 - head: $\text{list} \rightarrow D$ (selettore "testa")
 - tail: $\text{list} \rightarrow \text{list}$ (selettore "coda")
 - emptyList: $\rightarrow \text{list}$ (costante "lista vuota")
- $\Pi = \{ \text{empty} \}$
 - empty: $\text{list} \rightarrow \text{boolean}$ (test di lista vuota)

ADT LISTA

ESEMPI

```
head([ 6,7,11,21,3,6 ]) → 6
tail([ 6,7,11,21,3,6 ]) → [ 7,11,21,3,6 ]
cons(6, [ 7,11,21,3,6 ]) → [ 6,7,11,21,3,6 ]
empty([ 6,7,11,21,3,6 ]) → false
empty([]) → 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
$\text{cons} : D \times \text{list} \rightarrow \text{list}$	Costruisce una nuova lista, aggiungendo l'elemento fornito in testa alla lista data
$\text{head} : \text{list} \rightarrow D$	Restituisce il primo elemento della lista data
$\text{tail} : \text{list} \rightarrow \text{list}$	Restituisce la coda della lista data
$\text{emptyList} : \rightarrow \text{list}$	Restituisce la lista vuota
$\text{empty} : \text{list} \rightarrow \text{boolean}$	Restituisce vero se la lista data è vuota, <i>false</i> altrimenti

L'ADT LISTA

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*, 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***

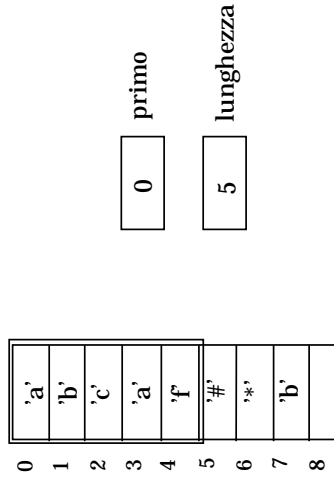
RAPPRESENTAZIONE CONCRETA DI LISTE

1) RAPPRESENTAZIONE STATICA

Una prima forma di rappresentazione 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 **costose le operazioni di inserimento e cancellazione** (copia dell'intera struttura dati)

ESEMPIO – Liste di caratteri come vettori

DICHIARAZIONE PRIMITIVE (listVet.h)

```
#define N 100
typedef struct {int primo, lunghezza;
char E[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.E[l.primo]);
}

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

ESEMPIO – Liste di caratteri come vettori

IMPLEMENTAZIONE PRIMITIVE (listVet.c) (segue)

```
list cons(char e, list l){
    list t;
    int i;
    t.primo=0; t.lunghezza=1;
    t.E[t.primo]=e;
    for(i=1; i<=l.lunghezza; i++) {
        t.E[i]=l.E[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=1.primo; i<l.lunghezza; i++){
        printf("%c", l.E[i]);
        if (i<l.lunghezza-1) printf(",");
    }
    printf("]\n");
}
```

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

2) RAPPRESENTAZIONE COLLEGATA

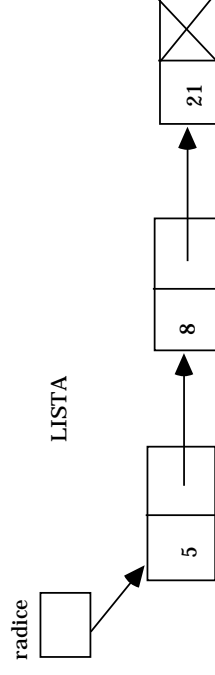
A ogni elemento si associa l'informazione ("indice", "riferimento") che permette di individuare la posizione dell'elemento successivo (*rappresentazione collegata*)

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

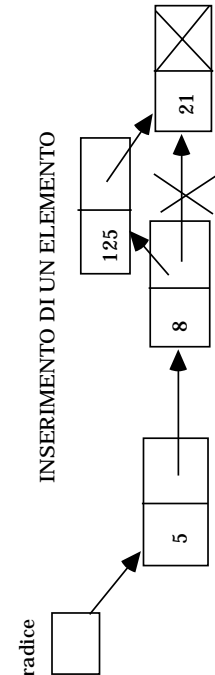
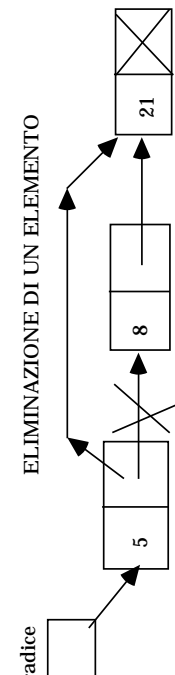
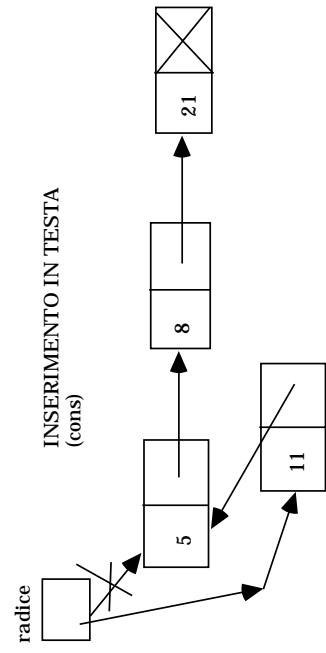
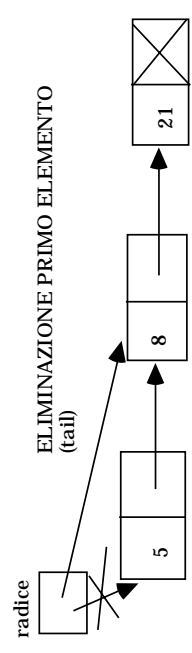
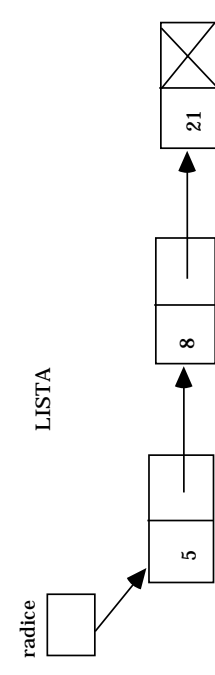
NOTAZIONE GRAFICA

- 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 realizzate 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*)

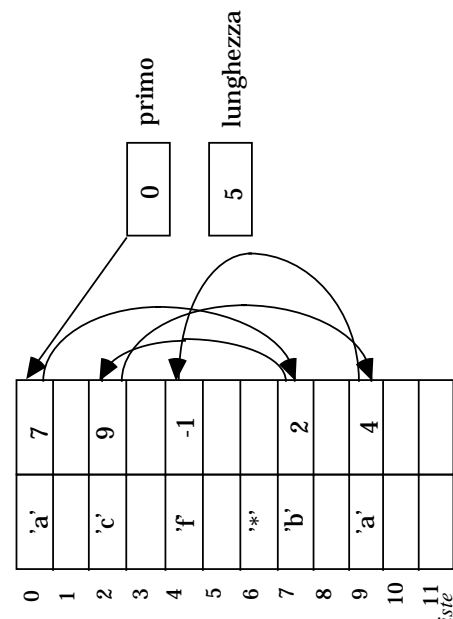
VANTAGGI

Cancelazione e inserimento più efficienti: non richiedono lo spostamento fisico di elementi (ma solo l'aggiornamento dei riferimenti)

SVANTAGGI

- Maggiore spazio occupato
- **dimensione massima** della lista
- gestione della **lista libera**, cioè gli elementi del vettore non ancora occupate da dati (per inserimento e cancellazione)

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



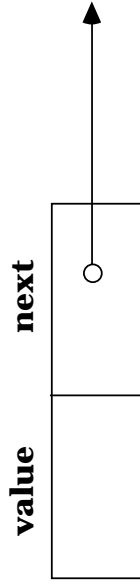
RAPPRESENTAZIONE COLLEGATA IMPLEMENTAZIONE MEDIANTE PUNTORI

Problema della dimensione massima del vettore: occorre adottare un approccio basato **sull'allocazione dinamica** della memoria

Ciascun **node** 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;
```

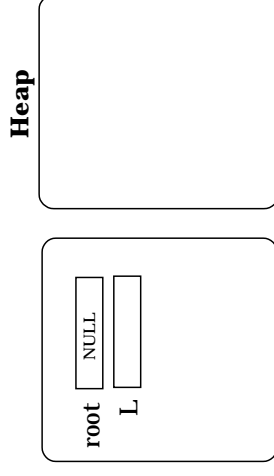
NOTA:

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)

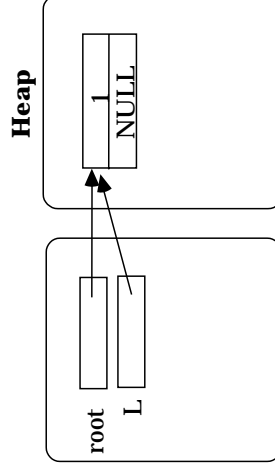
ESERCIZIO 1

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

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



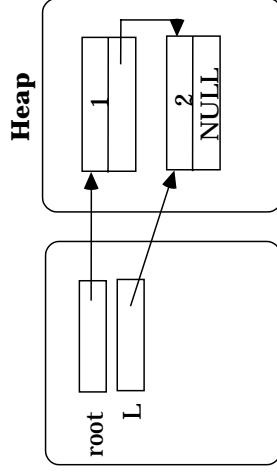
```
root = (list) malloc(sizeof(item));
root->value = 1;
root->next = NULL;
L = root;
```



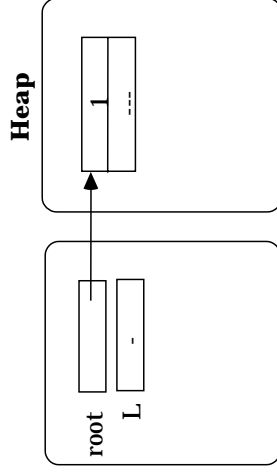
ESERCIZIO 1 (segue)

```
/* segue main */
```

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



```
free(L);
```



```
}
```

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

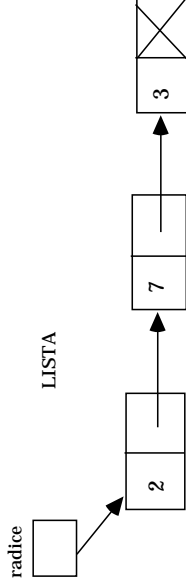
ESERCIZIO 2

Creazione di una lista di interi

```
#include <stdio.h>  
#include <stdlib.h>  
  
typedef struct list_element {  
    int value;  
    struct list_element *next;  
} item;  
  
typedef item* list;  
  
list insert(int e, list l) {  
    /* 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("\n Valore estratto:\t%d", l->value);  
        l=l->next;  
    }  
}
```

ESERCIZIO 2

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;
}
```

È una scansione sequenziale

→ nel caso peggiore, occorre scandire l'intera lista

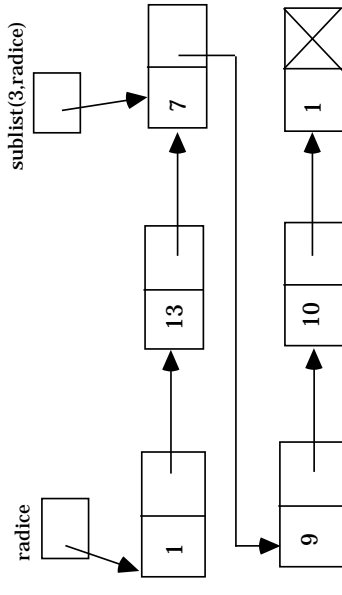
→ nel caso migliore, è il primo elemento

Esercizio: progettare e implementare una soluzione 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**:

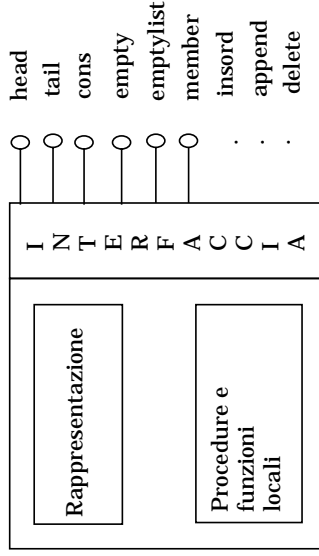
```
list sublist(int n, list l) {
    int i=1;
    while ((l!=NULL)&&(i<n)) { l = l->next; i++; }
    return l;
}
```

Versione **ricorsiva**:

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


COSTRUZIONE ADT LISTA

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**



il funzionamento di una lista **non dipende dal tipo degli elementi di cui è composta** -> cerchiamo di impostare una **soluzione generale**

LINEE GUIDA:

- definire un tipo **element** per rappresentare il generico tipo di elemento (con le sue proprietà)
- realizzare 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)

ADT LISTA

SCelta DI PROGETTO:

uso di `free()` in presenza di *structure sharing* è pericoloso
-> **NO DEALLOCAZIONE** memoria

- **inefficiente in termini di spazio occupato** a meno che non ci sia un garbage collector (Java, Prolog, Lisp)
- **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);
...
```

ADT LISTA

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;
}
```

ADT LISTA: FILE IMPLEMENTAZIONE (list.c)

```
#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");
}
```

Nota: 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);
}
```

ADT LISTA

ALTRE OPERAZIONI (NON PRIMITIVE)

operazione	descrizione
member : $D \times list \rightarrow boolean$	Restituisce vero o falso a seconda se l'elemento dato è presente nella lista data
length : $list \rightarrow int$	Calcola il numero di elementi della lista data
append: $list \times list \rightarrow list$	Restituisce una lista che è concatenamento delle due liste date
reverse: $list \rightarrow list$	Restituisce una lista che è l'inverso della lista data
copy: $list \rightarrow list$	Restituisce una lista che è copia della lista data

ADT LISTA

Il predicato member

<code>member(el,L) =</code>	<code>false</code>	se <code>empty(L)</code>
	<code>vero</code>	se <code>el==head(L)</code>
	<code>member(el, tail(L))</code>	<i>altrimenti</i>

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 molto simile alla specifica di alto livello -> codifica di **estrema semplicità**

Esercizio: scriverne una versione con **maggiore efficienza**

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, perché la somma viene eseguita **dopo** la chiamata ricorsiva

Esercizio: scriverne una versione con **maggior efficienza**

ADT LISTA

LA FUNZIONE append

append (come anche *copy* e *reverse*) non è solo un'operazione di **analisi** del contenuto o della struttura della lista data (come le precedenti), 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 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



l'unico modo per ottenere una lista diversa è
costruirne una nuova

Dunque, per aggiungere in coda a L1 la lista L2 occorre **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

-> Serve una **chiamata ricorsiva** ad *append*

ADT LISTA

LA FUNZIONE append

La definizione di append può quindi essere data così:

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

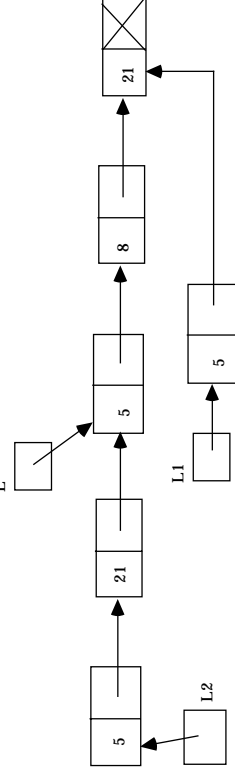
CODIFICA

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

NOTA: quando L1 diventa vuota, **append** restituisce direttamente L2, **non una sua copia**
→ L1 è duplicata, ma **L2 rimane condivisa**

structure sharing (parziale): ad esempio,

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



ADT LISTA

LA FUNZIONE 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**

Ma **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
```

CODIFICA

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

ADT LISTA

L'OPERAZIONE reverse

Esercizio: scrivere una versione *tail ricorsiva*

Per passare da una struttura ricorsiva a una *ricorsiva tail* occorre:

- predisporre una **funzione di interfaccia** che richiami la funzione tail-ricorsiva con i 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 copy

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

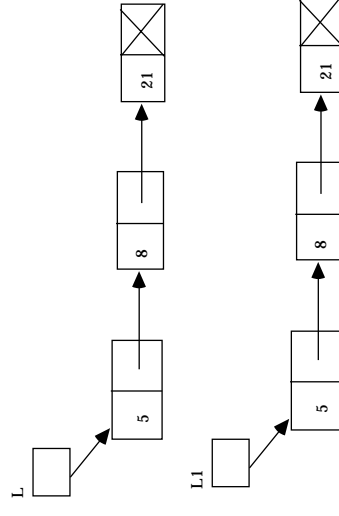
Si tratta quindi di impostare un **ciclo** (o una **funzione ricorsiva-tail**) che duplichi uno a uno tutti gli elementi

CODIFICA

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

ESEMPIO:

L1=copy(L)



ADT LISTA

LA FUNZIONE delete

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)

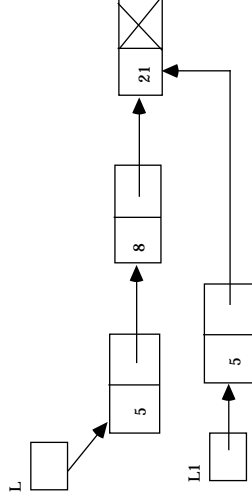
Occorre:

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

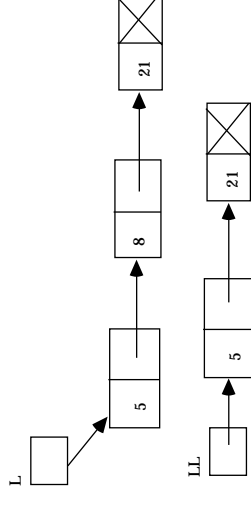


ADT LISTA

LA FUNZIONE delete

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 **free()** -> **uso inefficiente dello heap** in un linguaggio, come il C, privo di garbage collector (per evitare il rischio di riferimenti pendenti)

Realizzare **liste come valori (entità non modificabili)**: ogni modifica comporta la creazione di una nuova lista (per evitare il rischio di effetti collaterali indesiderati)

LISTE ORDINATE

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

NOTA: 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, etc

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

```
list inputordlist(int n) {
    element e;
    if (n<0) 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

LA FUNZIONE insord

Per inserire un elemento in modo ordinato in una lista che si *suppone ordinata*:

- 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
altrimenti

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 l'elemento el nella coda di l,
        e restituisci la lista così modificata >
}
```


LISTE ORDINATE

LA FUNZIONE insord

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

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

- come primo elemento (testa), **la testa della lista data**
- come coda, **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 **dupplicata**
- 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**

A questo fine, abbiamo introdotto ADT ausiliario **element**, realizzando poi ADT lista in termini di **element**

Osservazioni:

- **showList** dipendeva da una printf che svelava il tipo dell'elemento (e la rendeva **dipendente** da esso)
- **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 la **relazione d'ordine** fra due elementi
- verificare **l'uguaglianza** fra due elementi
- leggere da **input** un elemento
- scrivere su **output** un elemento

ADT element

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 dovrà essere **adattato** caso per caso al **tipo effettivo** dell'elemento considerato

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

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

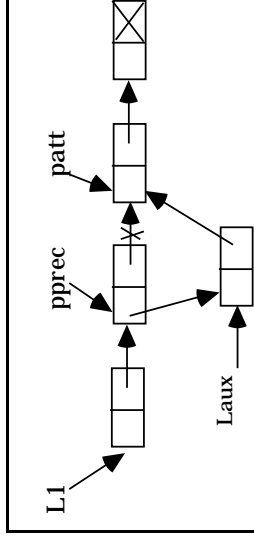
ESERCIZIO 4

Scrivere una versione primitiva della funzione `insord`

Ipotesi: la lista di partenza è ordinata

Algoritmo:

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



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:

Realizzare una **versione iterativa** di `mergeList` (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 da terminale alcuni nomi (al massimo di 30 caratteri) e li si inserisca in una lista ordinata alfabeticamente. Si stampi la lista ottenuta