Le liste collegate

Rappresentazione di liste

È molto comune dover rappresentare **sequenze di elementi** tutti dello stesso tipo e fare operazioni su di esse.

Esempi: sequenza di interi $(23\ 46\ 5\ 28\ 3)$
sequenza di caratteri `('x' 'r' 'f')`
sequenza di persone con nome e data di nascita

1. Rappresentazione sequenziale: tramite **array** (statici o dinamici)

Vantaggi:

- accesso agli elementi è diretto (tramite indice) ed efficiente
- l'ordine degli elementi è quello in memoria $\Longrightarrow$ non servono strutture dati addizionali che occupano memoria
- è semplice manipolare l'intera struttura (copia, ordinamento, ...)

Svantaggi:

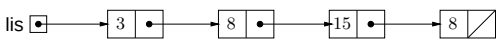
- inserire o eliminare elementi al centro è complicato ed inefficiente (bisogna spostare gli elementi che vengono dopo)

2. Rappresentazione collegata

- La sequenza di elementi viene rappresentata da una struttura di dati collegata, realizzata tramite **strutture e puntatori**.
- Ogni elemento è rappresentato con un una struttura C:
  - un campo per l'elemento (ad es. `int`)
  - un campo puntatore alla stuttura che rappresenta l'elemento successivo (ovviamente, tale struttura ha tipo indentico a quello della struttura corrente)
- L'ultimo elemento non ha un elemento successivo. $\Longrightarrow$ Il campo puntatore ha valore `NULL`.
- L'intera sequenza è **rappresentata** da un **puntatore al suo primo elemento** (vogliamo un'unica variabile per accedere a tutti gli elementi della sequenza).

  N.B. Il puntatore **non è** la sequenza, ma la rappresenta soltanto.

Graficamente:



Dichiarazioni di tipo

Esempio: Sequenze di interi.

```
struct nodoLista {
  int info;
  struct nodoLista *next;
};
typedef struct nodoLista NodoLista;
typedef NodoLista *TipoLista;
```

Esempio: Creazione di una lista di tre interi fissati: $(3, 8, 15)$

Implementazione: file `puntator/listeman.c`

```
TipoLista lis;                  /* puntatore al primo elemento della lista */

lis = malloc(sizeof(NodoLista));              /* allocazione primo elemento */
lis->info = 3;
lis->next = malloc(sizeof(NodoLista));       /* allocazione secondo elemento */
lis->next->info = 8;
lis->next->next = malloc(sizeof(NodoLista));        /* alloc. terzo elemento */
lis->next->next->info = 15;
lis->next->next->next = NULL;
```


Osservazioni:

- `lis` è di tipo `TipoLista`, quindi è un puntatore e **non** una struttura
- la zona di memoria per ogni elemento della lista (**non** per ogni variabile di tipo `TipoLista`) deve essere allocata esplicitamente con `malloc`

Esiste un modo più semplice di creare la lista di 3 elementi?

Esempio: Creazione di una lista di tre interi fissati, cominciando dall'ultimo: $(3, 8, 15)$

```
TipoLista aux, lis2 = NULL;

aux = malloc(sizeof(NodoLista));          /* allocazione dell'ultimo elemento */
aux->info = 15;    aux->next = lis2;
lis2 = aux;

aux = malloc(sizeof(NodoLista));          /* allocazione dell'ultimo elemento */
aux->info = 8;     aux->next = lis2;
lis2 = aux;

aux = malloc(sizeof(NodoLista));          /* allocazione dell'ultimo elemento */
aux->info = 3;     aux->next = lis2;
lis2 = aux;
```


Operazioni sulle liste

Supponiamo di aver creato una lista in memoria, con il puntatore iniziale in una variabile:

Stampa degli elementi di una lista

Vogliamo che venga stampato:

```
5 8 4
```

Versione iterativa: file `puntator/listeman.c`

N.B. `lis = lis->next` fa puntare `lis` all'elemento successivo della lista

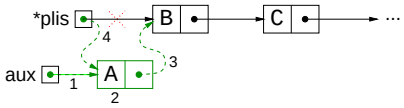
Versione ricorsiva

algoritmo stampa una lista
  if la lista non è vuota
  then stampa il dato del primo elemento
       stampa il resto della lista

Implementazione: file `puntator/listeman.c`

Inserimento di un nuovo elemento in testa

Esempio:



1. allochiamo una nuova struttura per l'elemento (`malloc`)
2. assegnamo il valore da inserire al campo `info` della struttura
3. concateniamo la nuova struttura con la vecchia lista
4. il puntatore iniziale della lista viene fatto puntare alla nuova struttura

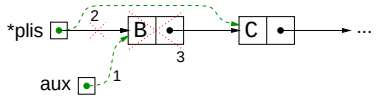
Implementazione: file `puntator/liste.c`, funzione `InserisciTestaLista`

- parametro di ingresso: elemento da inserire
- parametro di ingresso/uscita: lista $\Longrightarrow$ puntatore iniziale alla lista deve essere passato per indirizzo

Cancellazione del primo elemento

"Cancellare" significare deallocare la memoria occupata dall'elemento.

- se la lista è vuota non facciamo nulla
- altrimenti deallochiamo la struttura del primo elemento (con `free`) $\Longrightarrow$ la lista deve essere passata per indirizzo



Implementazione: file `puntator/liste.c`, funzione `CancellaPrimoLista`

```
void CancellaPrimoLista(TipoLista *plis)
{
  TipoLista aux;
  if (*plis != NULL) {
    aux = *plis;               /* 1 */
    *plis = (*plis)->next;     /* 2 */
    free(aux);                 /* 3 */
  }
}
```


Cancellazione di tutta la lista

Versione iterativa

Per **esercizio**: file `puntator/liste.c`, funzione `CancellaLista`

Versione ricorsiva: file `puntator/listeric.c`, funzione `CancellaLista`

```
void CancellaLista(TipoLista *plis)
{
  TipoLista aux;
  if (*plis != NULL) {
    aux = (*plis)->next;   /* memorizza il puntatore all'elemento successivo */
    free(*plis);                              /* dealloca il primo elemento */
    CancellaLista(&aux);                   /* cancella il resto della lista */
    *plis = NULL;
  }
}
```

Attenzione: una volta fatto `free`, non possiamo più far riferimento al campo `next` della struttura.

In che ordine vengono cancellati gli elementi della lista?

Verifica se un elemento compare in una lista

Versione iterativa

Scandiamo la lista continuando fino a che:
- l'elemento non è stato ancora trovato, e
- non siamo giunti alla fine della lista (usiamo una variabile booleana `trovato`)

Implementazione: file `puntator/listecon.c`, funzione `EsisteInLista`

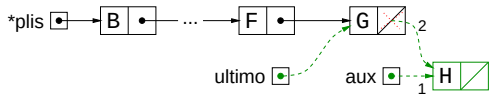
Versione ricorsiva

algoritmo verifica se *elem* compare in *lista*
  if *lista* è vuota
  then restituisci falso
  else if *elem* è il primo elemento di *lista*
    then restituisci vero
    else restituisci il risultato della verifica se *elem* compare nel resto di *lista*

Implementazione per **esercizio**: file `puntator/listeric.c`, funzione `EsisteInLista`

Inserimento di un elemento in coda — *Versione iterativa*

algoritmo inserisci *elem* in coda a *lista*
  alloca una struttura per il nuovo elemento
  if la *lista* è vuota
  then restituisci la *lista* costituita dal solo elemento
  else scandisci la *lista* fermandoti quando il puntatore corrente punta all'ultimo elem.
    concatena il nuovo elemento con l'ultimo



Nota: la lista è passata per indirizzo $\Longrightarrow$ dobbiamo usare un puntatore ausiliario per la scansione

Implementazione: file `puntator/liste.c`, funzione `InserisciCodaLista`

Inserimento di un elemento in coda — *Versione ricorsiva*

Caratterizzazione **induttiva** dell'inserimento in coda:

Sia *ris* la lista ottenuta inserendo *elem* in coda a *lis*. Allora:
1. se *lis* è la lista vuota, allora *ris* è costituita solo da *elem* (**caso base**)
2. altrimenti *ris* è ottenuta da *lis* facendo l'inserimento di *elem* in coda al resto di *lis* (**caso ricorsivo**)

Implementazione: file `puntator/listeric.c`, funzione `InserisciCodaLista`

```
void InserisciCodaLista(TipoLista *plis, TipoElemLista elem)
{
  if (*plis == NULL) {
    *plis = malloc(sizeof(NodoLista));
    (*plis)->info = elem;
    (*plis)->next = NULL;
  } else
    InserisciCodaLista(&(*plis)->next, elem);
}  /* InserisciCodaLista */
```

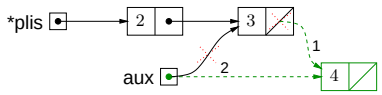
Vediamo l'evoluzione della pila dei RDA per l'inserimento di 3 in coda a $(1\ \ 2)$.

Creazione di una lista i cui elementi vengono letti da file

Vogliamo che gli elementi siano nell'ordine in cui compaiono nel file.
$\Longrightarrow$ Ogni elemento deve essere inserito in coda.

Per migliorare l'efficienza teniamo un puntatore all'ultimo elemento inserito in modo da non dover scandire tutta la lista per ogni elemento.

Esempio: contenuto del file: 2 3 4 5



Per evitare di dover trattare a parte il primo elemento usiamo la tecnica del **record generatore**:

- struct il cui campo `info` non contiene informazione significativa (record generatore viene detto anche **record fittizio**)
- viene creato e posto all'inizio della lista in modo che anche il primo elemento effettivo della lista abbia un elemento che lo precede
- prima di terminare la funzione bisogna rilasciare il record generatore

Implementazione: file `puntator/listeirw.c`, funzione `LeggiLista`

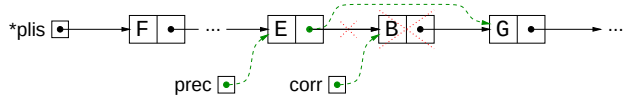
Cancellazione della prima occorrenza di un elemento

Versione iterativa

- si scandisce la lista alla ricerca dell'elemento
- se l'elemento non compare non si fa nulla
- altrimenti, a seconda di dove si trova l'elemento, si distinguono tre casi
  1. l'elemento è il primo della lista: si aggiorna il puntatore iniziale in modo che punti all'elemento successivo
  2. l'elemento non è ne il primo ne l'ultimo: si aggiorna il campo `next` dell'elemento che precede quello da cancellare in modo che punti all'elemento che segue
  3. l'elemento è l'ultimo: come (2), solo che il campo `next` dell'elemento precedente viene posto a `NULL`
- in tutti e tre i casi bisogna liberare la memoria occupata dall'elemento da cancellare

Osservazioni:

- per poter aggiornare il campo `next` dell'elemento precedente, bisogna **fermare la scansione sull'elemento precedente** (e non su quello da cancellare)



- per fermare la scansione dopo aver trovato e cancellato l'elemento, si utilizza una sentinella booleana
- per evitare di trattare a parte il caso del primo elemento, si può utilizzare di nuovo la tecnica del record generatore

Implementazione: file `puntator/listecon.c`, funzione `CancellaElementoLista`

Versione ricorsiva: per **esercizio**

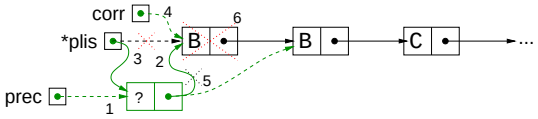
Implementazione: file `puntator/listeric.c`, funzione `CancellaElementoLista`

Cancellazione di tutte le occorrenze di un elemento — *Versione iterativa*

- analoga alla cancellazione della prima occorrenza di un elemento
- però, dopo aver trovato e cancellato l'elemento, bisogna continuare la scansione
- ci si ferma solo quando si è arrivati alla fine della lista

Osservazioni:

- in questo caso è decisamente meglio utilizzare il record generatore



- non serve la sentinella booleana per fermare la scansione

Implementazione: file `puntator/listecon.c`, funzione `CancellaTuttiLista`

Cancellazione di tutte le occorrenze di un elemento — *Versione ricorsiva*

Caratterizzazione **induttiva** della cancellazione di tutte le occorrenze:

Sia *ris* la lista ottenuta cancellando tutte le occorrenze di *elem* da *lis*. Allora:

1. se *lis* è la lista vuota, allora *ris* è la lista vuota (**caso base**)
2. altrimenti, se il primo elemento di *lis* è pari ad *elem*, allora *ris* è ottenuta da *lis* cancellando il primo elemento e tutte le occorrenze di *elem* dal resto di *lis* (**caso ricorsivo**)
3. altrimenti *ris* è ottenuta da *lis* cancellando tutte le occorrenze di *elem* dal resto di *lis* (**caso ricorsivo**)

Implementazione: file `puntator/listeric.c`, funzione `CancellaTuttiLista`

N.B. **Non ha alcun senso** applicare la tecnica del record generatore quando si implementa una funzione sulle liste in modo ricorsivo.

Inserimento di un elemento in una lista ordinata

Data una lista (ad es. di interi) già ordinata (in ordine crescente), si vuole inserire un nuovo elemento, mantenendo l'ordinamento.

Versione iterativa: per **esercizio**

Implementazione: file `puntator/listeord.c`, funzione `InserisciInListaOrdinata`

Versione ricorsiva

Caratterizzazione **induttiva** dell'inserimento in lista ordinata crescente:

Sia *ris* la lista ottenuta inserendo l'elemento *elem* nella lista ordinata *lis*. Allora

1. se *lis* è la lista vuota, allora *ris* è costituita solo da *elem* (**caso base**)
2. se il primo elemento di *lis* è maggiore o uguale a *elem*, allora *ris* è ottenuta da *lis* inserendovi *elem* in testa (**caso base**)
3. altrimenti *ris* è ottenuta da *lis* inserendo *elem* nel resto di *lis* (**caso ricorsivo**)

Implementazione: file `puntator/listeric.c`, funzione `InserisciInListaOrdinata`

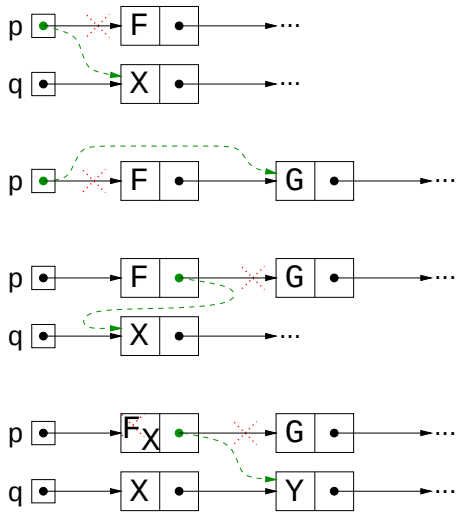
Sommario delle operazioni sulle liste

Funzione	Iterativa	Ricorsiva
`InserisciTestaLista`	lezione	
`CancellaPrimoLista`	lezione	
`StampaLista`	lezione	lezione
`EsisteInLista`	lezione	esercizio
`CancellaLista`	esercizio	lezione
`InserisciCodaLista`	lezione	lezione
`LeggiLista`	lezione (con rec. gen.)	esercizio
`CancellaElementoLista`	lezione (con rec. gen.)	esercizio
`CancellaTuttiLista`	lezione (con rec. gen.)	lezione
`InserisciInListaOrdinata`	esercizio	lezione
`CopiaLista`	esercizio	esercizio
`InvertiLista`	esercizio	esercizio

Implementazione: file `liste.c`, `listeric.c`, `listecon.c`, `listeord.c`, `listeirw.c`, `listerw.c` (tutti nella directory `puntator/`)

Tutte le funzioni sono discusse ed implementate nel libro di esercizi.

Riassunto delle operazioni sui puntatori



Esercizio di esame sulle liste

La Società Sportiva Olimpia dispone di un archivio in cui memorizza i farmaci che vengono assunti da ciascun atleta. L'archivio, realizzato attraverso strutture di dati C in memoria centrale, consiste di un vettore di 22 componenti, ciascuna delle quali è costituita da una struttura in cui sono memorizzati:

- la matricola dell'atleta (un intero) ed
- il puntatore ad una lista, rappresentata mediante strutture e puntatori, i cui elementi sono costittuiti da:
  - la sigla del farmaco (una stringa di al più 12 caratteri);
  - la data della prescrizione della terapia.

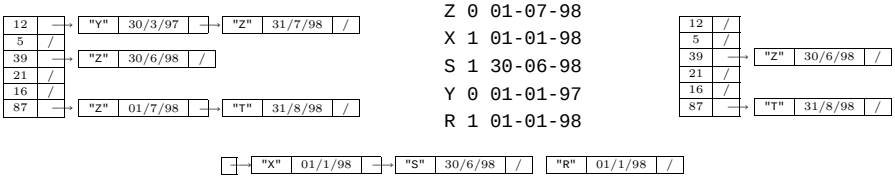
La Commissione Nazionale Antidoping ha aggiornato l'elenco delle sostanze vietate. In particolare, ha vietato alcune sostanze che precedentemente erano ammesse, ed ha ammesso alcune sostanze che erano vietate. La Commissione ha prodotto, e distribuito su floppy disk, un file in cui sono memorizzate le nuove informazioni. In particolare, ogni riga del file è costituita da tre campi, separati da uno spazio, che rappresentano rispettivamente:

- la sigla del farmaco (una striga di al più 12 caratteri);
- una cifra intera `0` o `1`, ad indicare se il farmaco è vietato (cifra `0`) o ammesso (cifra `1`);
- la data a partire dalla quale il farmaco è vietato o ammesso, a seconda dei casi, rappresentata nel formato `GG-MM-AA`.

Si richiede di:

1. Definire i tipi di dato C adeguati a risolvere i problemi di cui ai successivi punti (2) e (3).
2. Scrivere una funzione C che, dati come parametri l'archivio degli atleti della Società Olimpia ed il nome del file contenente le nuove informazioni sui farmaci vietati o ammessi, elimini dall'archivio tutte le prescrizioni dei farmaci vietati avvenute **non anteriormente** alla data del divieto.
3. Scrivere una funzione C che, dato come parametro il nome del file con le nuove informazioni sui farmaci, costruisca e fornisca in uscita la lista, rappresentata mediante strutture e puntatori, di tutti i farmaci ammessi, unitamente alle date di ammissione.

Esempio: Si consideri nella figura a sinistra l'archivio di atleti e al centro il file con le nuove informazioni sui farmaci vietati o ammessi. A destra è mostrato l'archivio dopo l'esecuzione della funzione richiesta al punto (2), ed in basso la lista fornita dalla funzione di cui al punto (3).



Strutture di dati

- per ogni atleta: – matricola
  – lista di prescrizioni: per ogni prescrizione sigla e data

  $\Longrightarrow$ vettore di strutture: – matricola
  – puntatore inziale alla lista di prescrizioni

- file: ogni riga: – sigla
  – se vietato (`0`) o ammesso (`1`)
  – data nel formato `GG-MM-AA`

- lista di farmaci ammessi: per ogni farmaco: – sigla
  – data di ammissione

  $\Longrightarrow$ stessi tipi della lista di prescrizioni

Implementazione: file `esami/doping.c`

Costruzione della lista di farmaci ammessi

Funzione che prende in ingresso il nome di un file e restituisce la lista dei farmaci ammessi:

- un parametro di ingresso (per valore), di tipo `char*`
- la lista viene restituita come valore di ritorno

algoritmo costruisci la lista di farmaci ammessi
  inizializza la lista alla lista vuota
  apri il file in lettura
  while non si è giunti alla fine del file
  do leggi un farmaco dal file
    if il farmaco è ammesso
    then inseriscilo (con sigla e data) nella lista
  chiudi il file

N.B. Non è richiesto un ordine specifico per gli elementi della lista.
$\Longrightarrow$ Adottiamo il modo più semplice per costruirla, cioè inserendo in testa.

Implementazione: file `esami/doping.c`

Cancellazione delle prescrizioni di farmaci vietati

Funzione che: prende in ingresso: – nome di un file – vettore di liste
restituisce: vettore di liste dalle quali sono stati tolti i farmaci non presenti prima del divieto

algoritmo cancella farmaci prescritti non prima del divieto
for ogni elemento del file
do if rappresenta un divieto
then cancella le prescrizioni opportune dalle liste

Raffinamento successivo:

algoritmo cancella farmaci prescritti non prima del divieto
apri il file in lettura
while non si è giunti alla fine del file
do leggi un elemento dal file
if l'elemento rappresenta un divieto
then for ogni atleta nel vettore di liste
do cancella dalla lista di farmaci dell'atleta quelli richiesti dal divieto
chiudi il file

Algoritmo ausiliario:

algoritmo cancella da una lista di farmaci quelli richiesti dal divieto
if la lista non è vuota
then cancella dal resto della lista i farmaci richiesti dal divieto
if il primo elemento
– ha una sigla che coincide con quella del divieto **e**
– ha una data di prescrizione non anteriore a quella del divieto
then eliminalo

Per confrontare due date usiamo una funzione ausiliaria (booleana).

Implementazione: file `esami/doping.c`

Esercizio: Gestione di un archivio di esami memorizzato su un file.

Le **informazioni** associate ad ogni esame sono:

- materia
- data in cui è stato sostenuto (giorno, mese, anno)
- voto

Le **operazioni di interesse** sono:

`A` ... inserimento di un nuovo esame
`B` ... cancellazione di un esame (per correggere eventuali errori)
`C` ... stampa della lista di esami su schermo
`D` ... calcolo media esami
`E` ... stampa esami con voto superiore/inferiore alla media
`F` ... stampa esami ordinati per data (difficile)
`G` ... stampa esami ordinati per voto (difficile)
`H` ... calcolo trimestre in cui sono stati sostenuti più/meno esami
...
`R` ... lettura dei dati da un file
`S` ... salvataggio dei dati su un file
`X` ... uscita dal programma, con eventuale richiesta di salvataggio dei dati

Interfaccia utente

- All'utente deve essere presentato un semplice menu delle operazioni, dal quale può scegliere l'operazione da effettuare digitando un carattere.
- Il menu viene presentato fino a quando l'utente non decide di uscire dal programma (operazione X).

Suggerimenti

- costruire (in memoria centrale) una lista collegata contenente i dati sugli esami letti da file (operazione R)
- effettuare le operazioni modificando la rappresentazione in memoria
- salvare su file i dati nella lista a richiesta dell'utente (operazione S)
- salvare su file i dati nella lista anche al momento dell'uscita dal programma (solo se sono state fatte modifiche dall'ultimo salvataggio), in modo da evitare di perdere eventuali modifiche fatte
- realizzare ciascuna operazione attraverso una funzione con opportuni parametri