LISTE E ALGORITMI PER LA LORO GESTIONE

Una lista è una sequenza di elementi di un determinato tipo.

Per indicare strutture a lista useremo la seguente notazione (detta parentetica):
L = ['a', 'b', 'c'], denota la lista L dei caratteri 'a', 'b', 'c'

[5,8,5,21,8], lista di interi

-
- È un ***multi-insieme*****:** ci possono essere ripetizioni del medesimo elemento.
-
- È una ***sequenza*** (collezione ordinata): siamo in grado di individuare il primo elemento della sequenza, il secondo, etc.
-

Come realizzare una lista?

Spesso è necessario gestire e manipolare in un programma collezioni di dati le cui dimensioni sono soggette a variare dinamicamente.

La realizzazione come ***vettore*** (di N componenti) ***non è sufficiente*** se durante l'esecuzione di un programma ci si trovi a dover inserire l'(N+1)-esimo elemento.

Rappresentazione collegata, mediante ***puntatori*** e ***allocazione dinamica di memoria***!!!

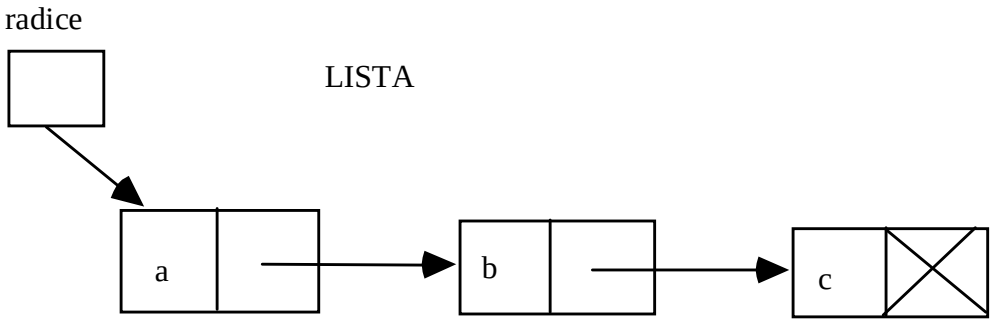
Liste: rappresentazione collegata

Si memorizzano gli elementi associando a ognuno di essi l'informazione (***riferimento***) che permette di individuare la locazione in cui è inserito l'elemento successivo (**rappresentazione collegata**).

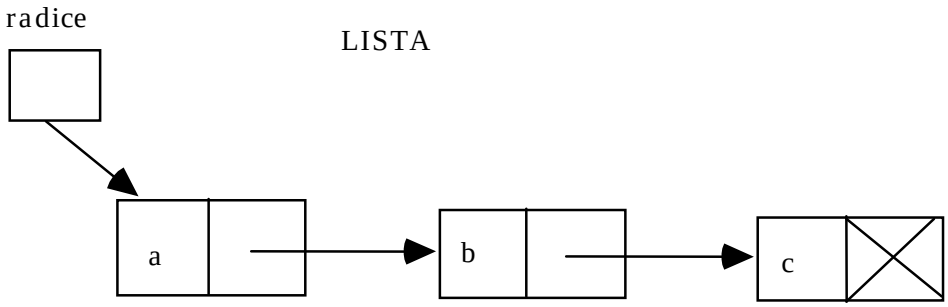
Notazione grafica:

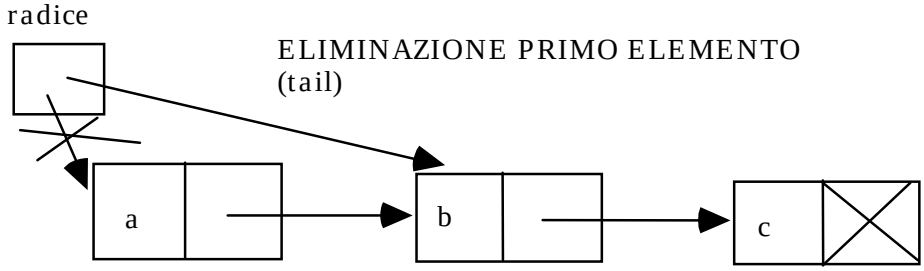
Elementi della lista come **nodi** e riferimenti come **archi**.

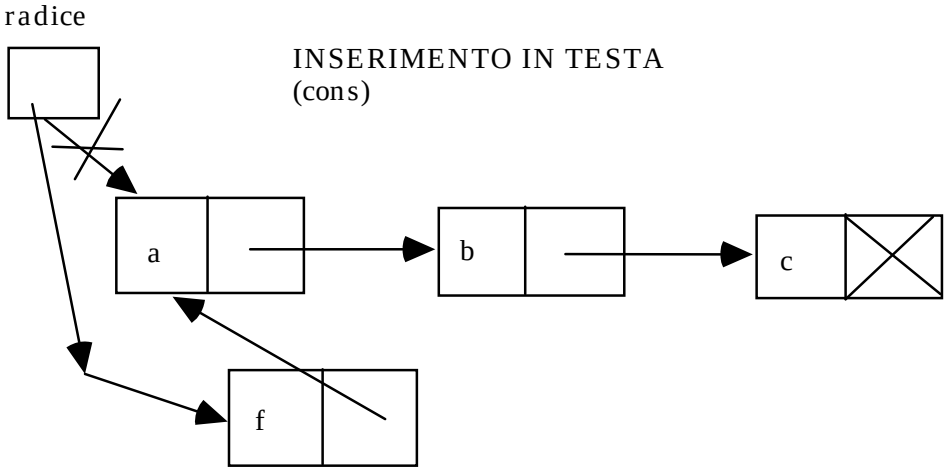
['a', 'b', 'c']

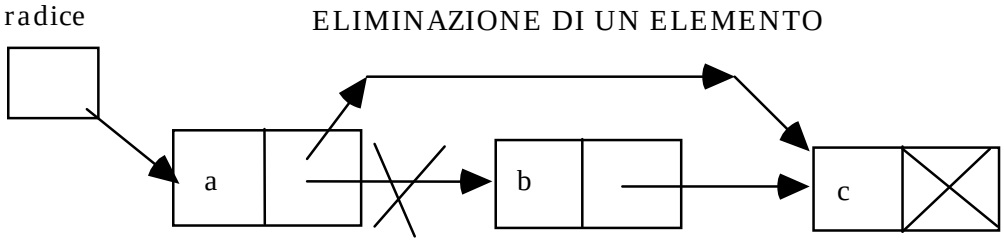


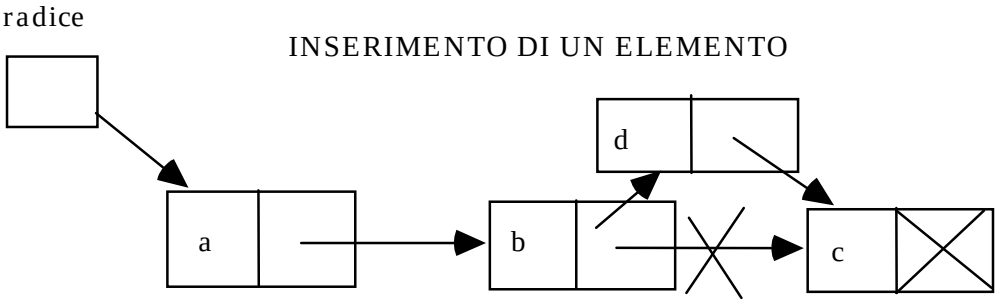
Si possono realizzare mediante ***strutture dati dinamiche***


radice
LISTA
a
b
c


radice
ELIMINAZIONE PRIMO ELEMENTO
(tail)
a
b
c


radice
INSERIMENTO IN TESTA
(cons)
a
b
c
f


radice
ELIMINAZIONE DI UN ELEMENTO
a
b
c

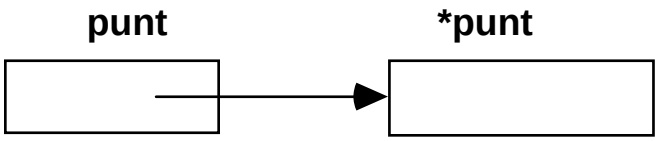

radice
INSERIMENTO DI UN ELEMENTO
d
a
b
c

Strutture dati dinamiche:

Il C (come altri linguaggi) consente di creare strutture dati dinamiche ***collegate*** attraverso puntatori in cui la memoria per ciascun elemento viene richiesta (e allocata) dinamicamente (tramite la funzione ***malloc***) prelevandola dall'***area heap***.

Allocazione:

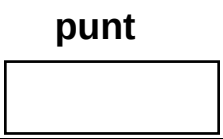
punt=(tipoelem *) malloc(sizeof(tipoelem));



- Con questa operazione viene allocato lo spazio per la variabile **`*punt`** (la memoria allocata per **`*punt`** è prelevata dall'insieme delle celle libere dello heap).

Deallocazione:

free (punt);



- Dopo questa operazione **`*punt`** non esiste piu` (la memoria allocata per **`*punt`** è di nuovo inserita nell'insieme delle celle libere dello heap).

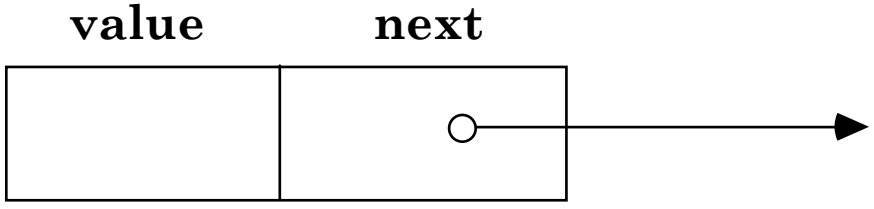
Rappresentazione collegata di liste mediante puntatori

In C si possono utilizzare i ***puntatori*** per ovviare alle limitazioni di dimensione della lista: la memoria utilizzata viene gestita dinamicamente ed è esattamente proporzionale al numero degli elementi della lista.

Ciascun elemento della lista è una struttura di due campi:

- un campo rappresenta il **valore** dell'elemento (char, nell'esempio)
- un campo è di tipo ***puntatore*** e punta all'**elemento successivo** nella lista (NULL, nel caso dell'ultimo elemento).

```
typedef char element_type;

typedef struct list_element
     { element_type value;
       struct list_element *next;
     } item;
```



Liste semplici: rappresentazione collegata me-diante puntatori

Esempio:

```
typedef char element_type;

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

typedef item *list;
```

Commenti:

- Nella definizione del tipo **`struct list_element`** si fa precedere l'identificatore del tipo alla collezione dei campi: in questo modo è possibile dichiarare in tale collezione un campo (**`next`**) di tipo puntatore a **`struct list_element`**.
-
- Inoltre, il tipo **`struct list_element`** viene rinominato come **`item`**.
-
- Infine il tipo **`list`** è un puntatore al tipo **`item`**.

0 Esercitazione 8.1: Creazione di una lista da ingresso

Sia data dal dispositivo di standard input una sequenza di caratteri, forniti in ordine casuale. La sequenza ha lunghezza non nota a priori ed è terminata da **EOF**.
Realizzare un programma che acquisisca la sequenza e la visualizzi sul dispositivo d'uscita standard.

Soluzione:

In questo caso, essendo la sequenza di lunghezza non nota a priori il problema può essere risolto attraverso l'uso di una *lista*.

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

typedef char element_type;
typedef struct list_element
     { element_type value;
       struct list_element *next;
     } item;
typedef item *list;
```

Il tipo **list** è realizzato attraverso una struttura che ha due campi, il primo di tipo **element_type** (che è una ridefinizione del tipo **char**) e il secondo di tipo puntatore a una struttura di tipo **list**.
Si noti la natura ricorsiva di tale struttura dati.

```
/* prototipi */
void showlist (list l);
void showr_l (list l);
void showr_list (list l);
```

Nel codice, e nella funzione **main** in particolare, il riferimento alla testa della lista è mantenuto dalla variabile **root**, che ha inizialmente il valore **NULL**.

```
void main(void)
{list root=NULL, aux;
 char n;
 while ((n=getchar()) != EOF)
 {aux=(list)malloc(
         sizeof(struct list_element));
    aux->value=n;
    aux->next=root;
    root=aux;}
 showlist(root);
 showr_l(root);
}
```

Commenti:

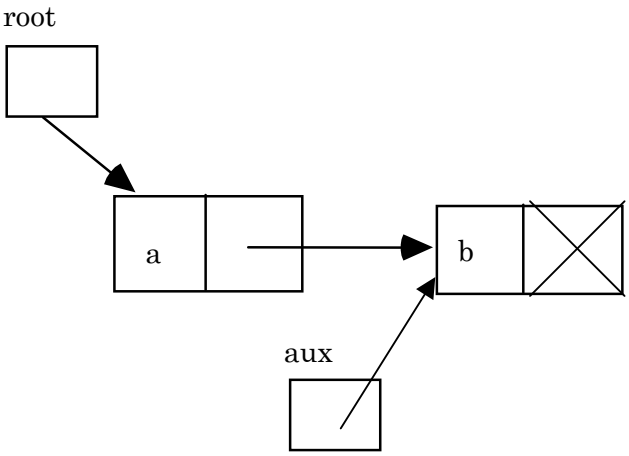
Il ciclo **`while`** legge i valori dallo standard input, fino a incontrare il carattere di fine file (**`EOF`**); per ogni valore letto viene allocato un nuovo elemento (con l'operazione **`malloc`**) il cui indirizzo è assegnato ad **`aux`**. Al campo **`aux->value`** viene assegnato il valore **`n`**, mentre al campo **`aux->next`** è il valore di **`root`**.

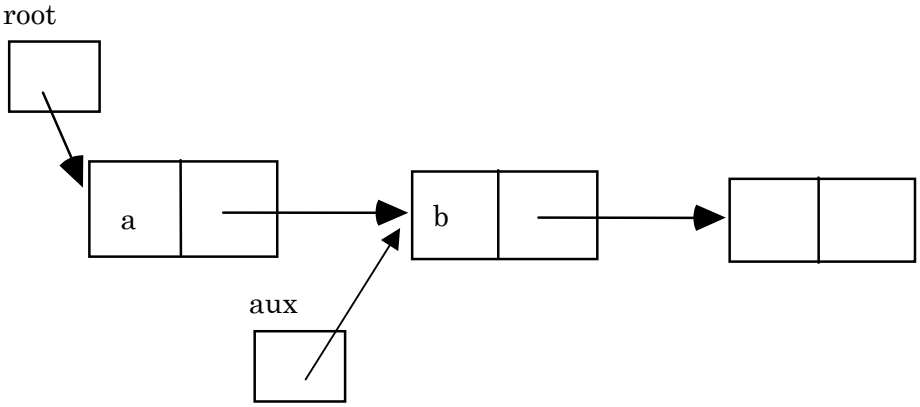
Il nuovo elemento è collocato ***in testa alla lista*** e risulta essere il primo della sequenza. Per questo motivo a **`root`** è assegnato l'indirizzo del nuovo elemento che è contenuto in **`aux`**.

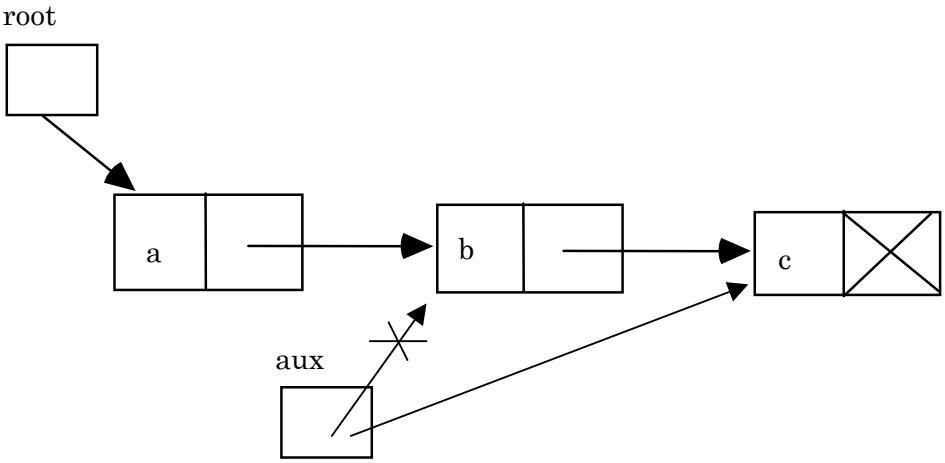
Per il particolare modo di creazione, il campo puntatore dell'ultimo elemento della lista è sempre **`NULL`** (importante per verificare la fine della lista)

Quindi la lista è creata in ordine inverso rispetto alla sequenza data in ingresso.

E se volessimo memorizzare i valori letti in modo tale da averli nello stesso ordine della sequenza di ingresso? Occorrerebbe inserire ogni nuovo elemento ***in fondo alla lista***.
Per ottenere questo tipo di inserimento serve un puntatore (per esempio, **`aux`**) aggiornato via via e che punti all'ultimo elemento della lista (oltre al puntatore **`root`**).


root
a
b
aux


root
a
b
aux


root
a
b
c
aux

```
void main(void)
{/* versione che inserisce in fondo */
list root=NULL, aux;
 char n;
 while (((n=getchar()) != EOF)
 { if (root==NULL)
        /*primo inserimento */
       { root=(list)malloc(
             sizeof(struct list_element));
         aux=root;
       }
   else
       { aux->next=(list)malloc(
             sizeof(struct list_element));
         aux=aux->next;
       }
   aux->value=n;
   aux->next=NULL;

}
 showlist(root);
 showr_l(root);
}
```

La stampa della sequenza è fornita in due versioni, iterativa (**showlist**) e ricorsiva (**showrlist**).

```
void showlist (list l)
/*versione iterativa */
{ printf("[");
  while (l!=NULL)
        {putchar(l->value);
        l=l->next;
        if (l!=NULL) printf (","); }
  printf("]\n");
}
```

La funzione **showlist** prevede una scansione iterativa della lista (realizzata mediante il costrutto **while**), a partire dal riferimento **root** che è passato alla chiamata della funzione e copiato nel parametro formale **l** della funzione stessa.

La lista è scandita a partire da **l** sino a che non si incontra, nel campo **l->next** dell'elemento corrente, il valore **NULL**.

La versione ricorsiva della stampa si avvale della funzione **showr_l** che stampa le parentesi quadre di inizio e fine lista, al fine di ottenere una stampa meglio presentata, e a sua volta invoca la funzione ricorsiva **showr_list**.

```
void showr_l (list l);
{printf("[");
 showr_list(l);
 printf("]\n");
}

void showr_list (list l)
/*versione ricorsiva */
{if (l!=NULL)
     {putchar(l->value);
      if (l->next!=NULL)
          {printf (",");
           showr_list(l->next);}
     }
}
```

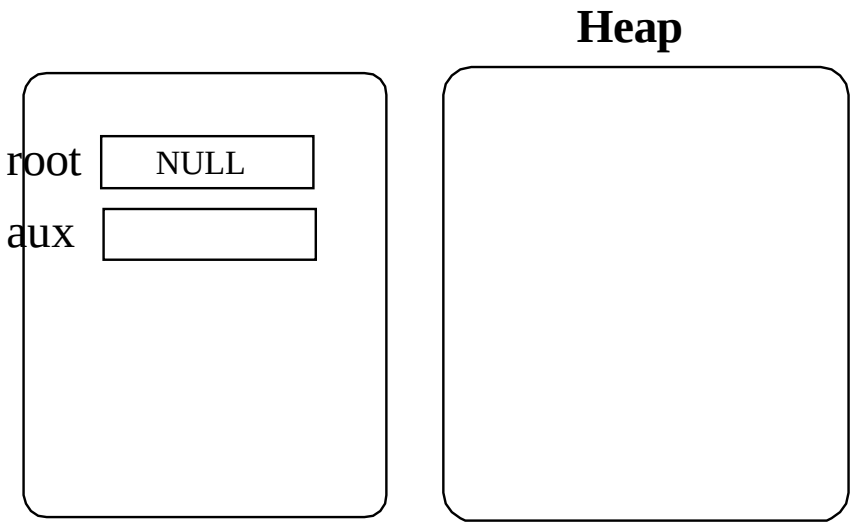
La funzione ricorsiva **showr_list** riferisce, attraverso il suo parametro formale **l**, un elemento nella lista (inizialmente quello riferito da **root**).
Nel caso in cui **l** non abbia il valore **NULL** (il valore **NULL** equivale alla condizione di terminazione della ricorsione), **showr_l** stampa il valore del campo **l->value** e si chiama ricorsivamente, passando il riferimento all'elemento successivo nella lista (**l->next**) purché anche questo non sia **NULL**.

Esempio di funzionamento:

Si supponga di ricevere da ingresso la sequenza:

```
a b EOF
```

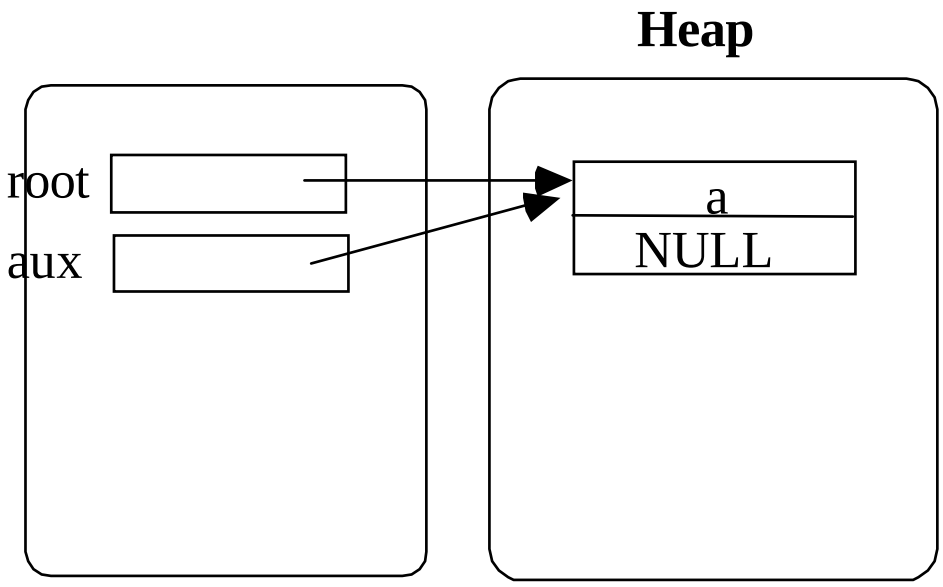
```
#include <stdio.h>
#include <stdlib.h>

typedef char element_type;
typedef struct list_element
     { element_type value;
       struct list_element *next;
     } item;
typedef item *list;
void main(void)
{list root=NULL, aux;
 char n;
```



```
while (((n=getchar()) != EOF)
```

Prima iterazione:

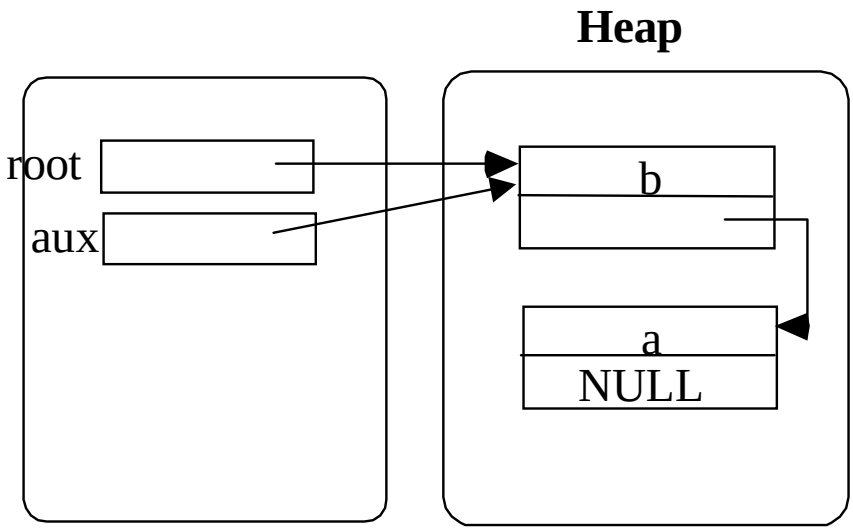
```
 {aux=(list)malloc(
          sizeof(struct list_element));
  aux->value=n;
  aux->next=root;
  root=aux;}
```



```
while (((n=getchar()) != EOF)
```

Seconda iterazione:

```
  {aux=(list)malloc(
            sizeof(struct list_element));
    aux->value=n;
    aux->next=root;
    root=aux;}
```

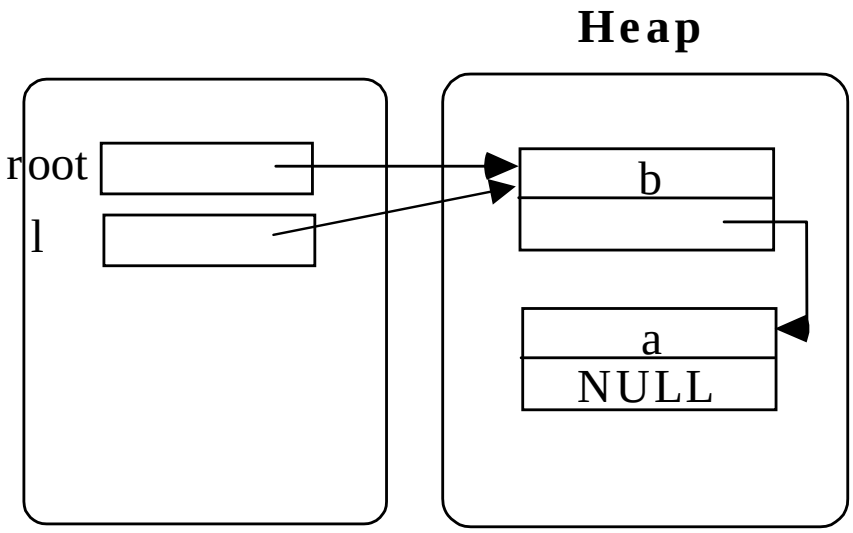


Uscita dal ciclo:

```
showlist(root);
```

Esecuzione della funzione:

```
void showlist (list l)
                /*versione iterativa */
{ printf("[");
 while (l!=NULL)
         {putchar(l->value);
         l=l->next;
         if (l!=NULL) printf (","); }
 printf("]\n");
}
```

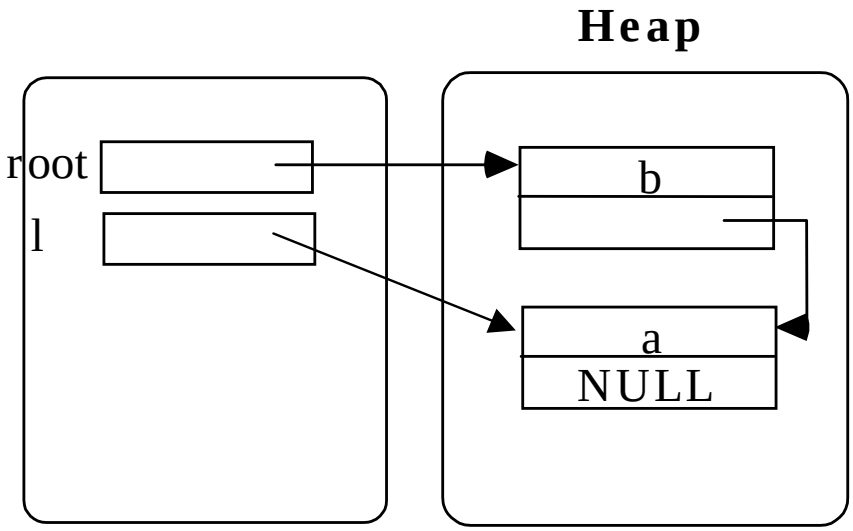


La prima iterazione stampa il carattere:

```
b
```

e aggiorna il parametro formale:

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



Alla seconda iterazione, si stampa il carattere:

```
a
```

e si aggiorna il parametro formale:

```
l=l->next;      /* ora l vale NULL */
```

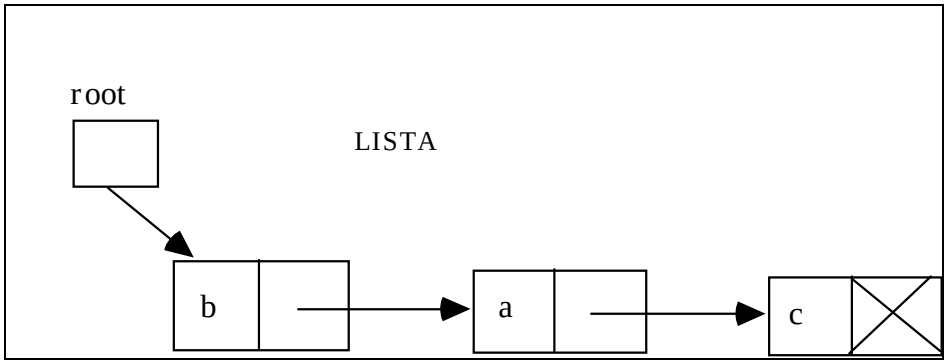
Il test:

```
while (l!=NULL)
```

è falso e si esce dal ciclo terminando poi la funzione.

Ricerca di un elemento in una lista:

L'operazione di ricerca di un elemento si realizza con la scansione della lista, a partire dalla radice:



verificando, per ciascun nodo della struttura, se il campo **`value`** del nodo corrente corrisponde al valore cercato.

Richiede nel caso peggiore (l'elemento cercato non esiste) la scansione di tutta la struttura dati (***complessità O(N)***, dove N è il numero di elementi della lista).

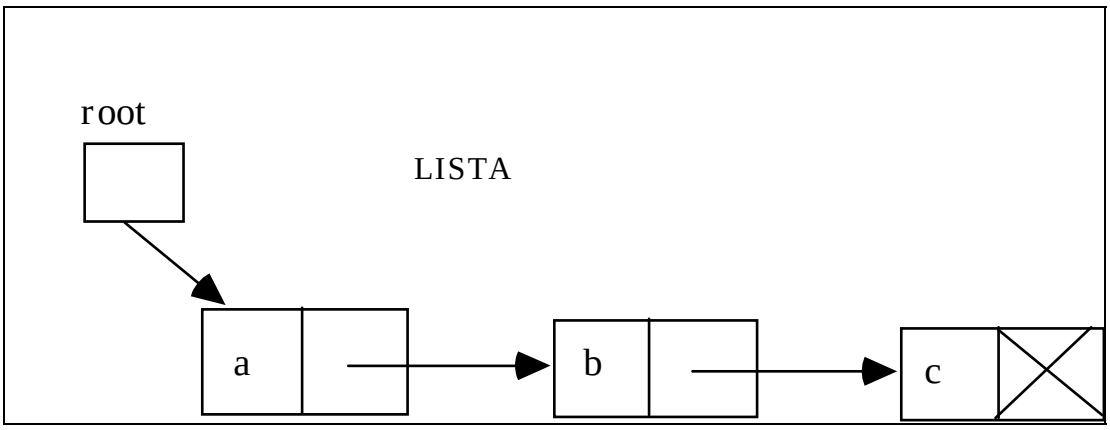
```
int member_it(char el,list l)
/* appartenenza di un elemento -
   versione iterativa  */
{ while (l!=NULL)
      {if(el==l->value) return(1);
      else l=l->next;}
return(0);  }
```

Attraverso il parametro formale **`l`**, aggiornato ad ogni iterazione, scandiamo la lista e verifichiamo se l'elemento corrente (**`l->value`**) è quello cercato, sino a che si raggiunge la fine della lista (**`NULL`**) o si trova l'elemento cercato.

La definizione ricorsiva della stessa operazione è data dalla funzione che segue:

```
int member(char el, list l)
/* appartenenza di un elemento -
   versione ricorsiva  */
{ if (l==NULL) return(0);
  else
   { if (el==l->value) return(1);
     else return(member(el, l->next)); }
}
```

La ricerca potrebbe essere migliorata se la lista fosse costruita mantenendo un ordine tra i valori della sequenza:



Si deve prevedere però una modalità con cui inserire in modo ordinato gli elementi nella sequenza man mano che li riceviamo da ingresso.

Creazione di una lista ordinata:

Modifichiamo l'esercitazione 8.1, creando la lista ordinata:

```
void main(void)
{list root=NULL, aux, prec, current;
 char n;  int trovato=0;
 while (((n=getchar()) != EOF)
 {aux=(list)malloc(sizeof(item));
  aux->value=n;   aux->next=NULL;
  if (root==NULL) /* primo elemento */
       root=aux;
  else
   {current=root; trovato=0;
    while((current!=NULL)&&(!trovato))
      { if (current->value<n)
         { prec=current;
           current=current->next;}
        else trovato=1;}
    if (current==NULL)
         {prec->next=aux;} /*in fondo */
    else
       if (current==root)
             {aux->next=root;
              root=aux;}  /* in testa */
       else {prec->next=aux;/*in mezzo */
              aux->next=current;}
       }
 }
 showlist(root);
}
```

Commenti:

Il ciclo **`while`** più esterno della funzione per ogni carattere letto alloca un nuovo elemento il cui indirizzo è assegnato ad **`aux`**.

Nel caso in cui la lista sia vuota (condizione **`root==NULL`**), si aggiunge l'elemento allocato (riferito dal puntatore **`aux`**) facendolo riferire anche dal puntatore alla radice della lista, **`root`**.

Se la lista non è vuota, inizia una scansione della lista attraverso un secondo ciclo iterativo **`while`**. Per ciascun passo, si permane nel ciclo se l'elemento corrente della lista (riferito dal puntatore **`current`**, inizialmente posto uguale a **`root`**) ha un valore (campo **`current-> value`**) minore di quello appena letto (**`n`**) e se non si è raggiunta la fine della lista (**`current!=NULL`**).

A ogni iterazione, si procede quindi alla scansione, aggiornando il puntatore **`current`** (**`current = current->next`**) e mantenendo nel puntatore **`prec`** un riferimento all'elemento precedente.

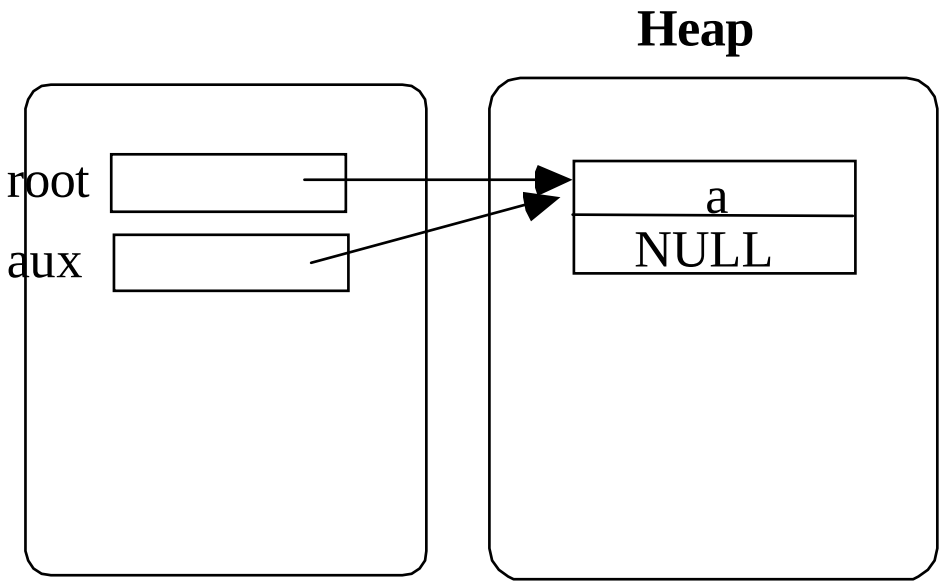
Si esce dall'iterazione, quindi, in due casi:
si è raggiunta la fine della lista (**`current`** vale **`NULL`** e in questo caso l'elemento allocato e riferito da **`aux`** è attaccato in fondo alla lista); oppure l'elemento corrente ha un valore più grande di quello letto (**`n`** e in questo caso l'elemento allocato e riferito da **`aux`** deve precederlo nella lista).

Esempio di funzionamento:

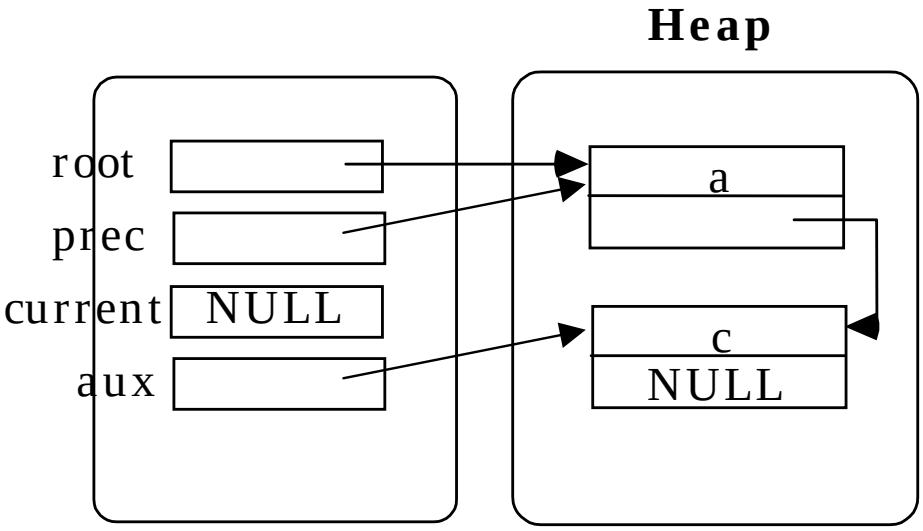
Si supponga di ricevere da ingresso la sequenza:

```
a c b EOF
```

Primo inserimento: (a partire dalla lista vuota, **`root=NULL`**)

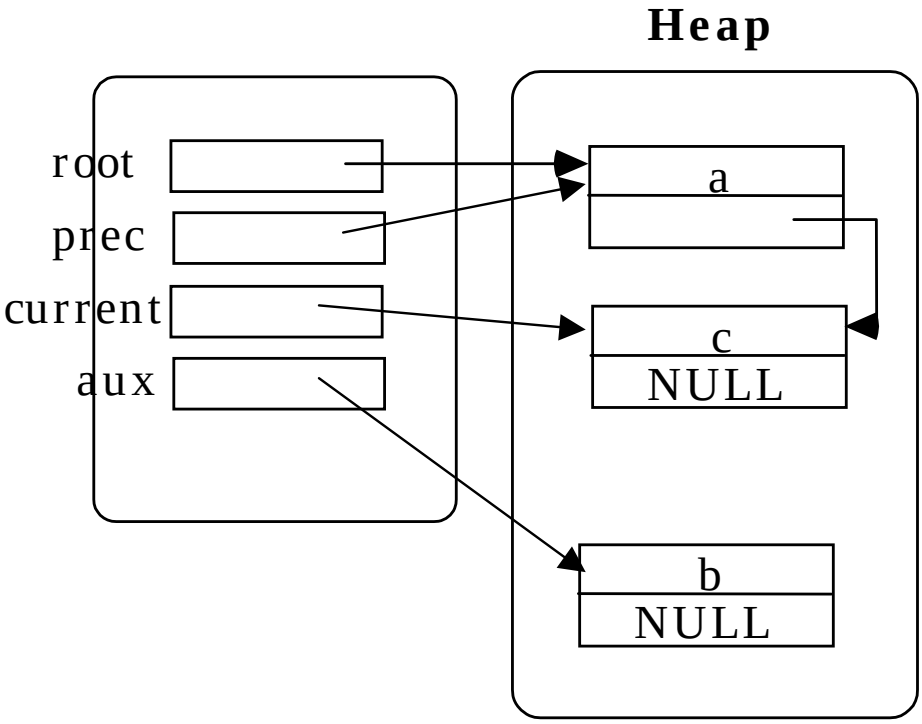


Secondo inserimento: (inserisce in fondo, **`c`**)

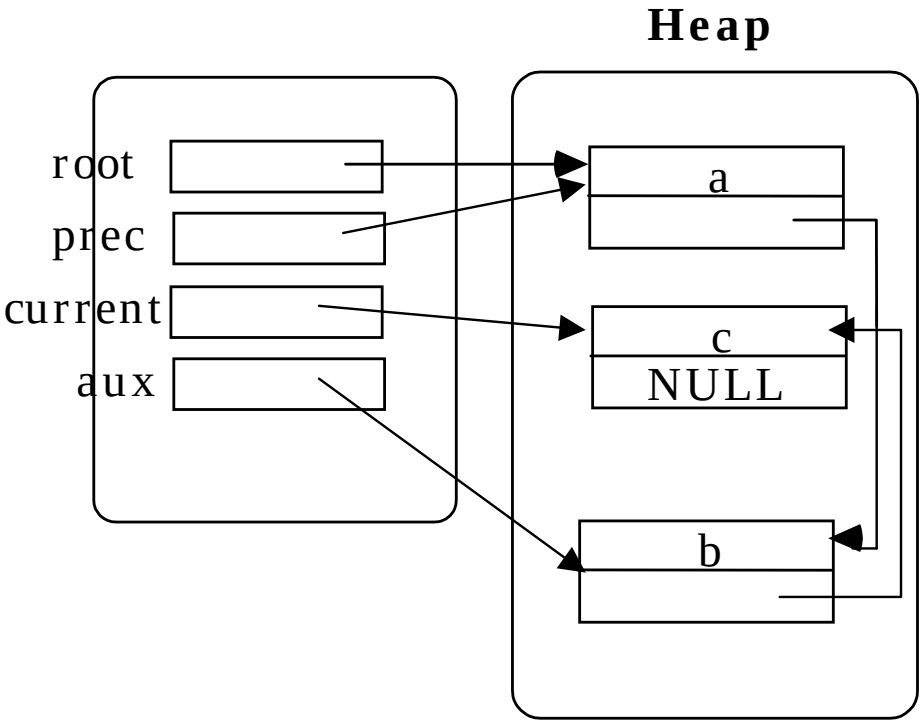


Terzo inserimento: (inserisce in mezzo, **`b`**)

A questo punto:



dove **`current->value`** non è minore del valore **`n`** da inserire, si effettua l'inserimento modificando i valori di **`prec->next`** e **`aux->next`**:



Esercizio 8.1.0a

Aggiungere al programma 8.1 una funzione:

```
list delete(element_type el, list l)
```

che elimina un elemento el dalla lista l. Realizzare tale funzione in modo non ricorsivo, operando sui puntatori che collegano i nodi della struttura dati.

Esercizio 8.1.0b

Aggiungere al programma 8.1 tre funzioni che realizzano l'ordinamento in testa, in fondo e ordinato di un elemento nella lista:

```
list cons(element_type el, list l);
list cons_tail(element_type el, list l);
list ordins(element_type el, list l);
```

e costruire tre liste, L1, L2 e L3 con queste tre modalità chiamandole dal main program.

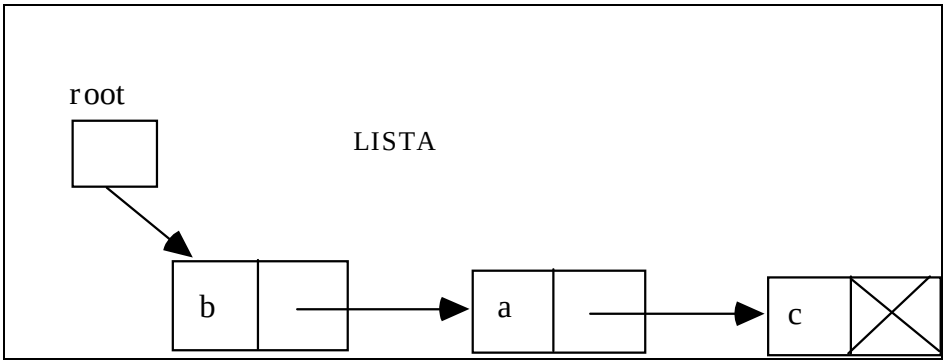
```
/*esercizio 8.1.0b */
#include <stdio.h>
#include <stdlib.h>

typedef char element_type;
typedef struct list_element
     { element_type value;
       struct list_element *next;
     } item;
typedef item *list;

void main(void)
{list L1=NULL, L2=NULL, L3=NULL, aux;
 char n;
 while ((n=getchar()) != EOF)
   { L1=cons(n,L1);
     L2=cons_tail(n,L2);
     L3=cons(n,L3);   }
 printf("Con inserimento in testa: \n");
 showlist(L1);
 printf("Con inserimento in fondo: \n");
 showlist(L2);
 printf("Con inserimento ordinato: \n");
 showlist(L3);
}

list  cons(element_type e, list l)
{ list t;    /* aggiunge in testa */
  t=(list)malloc(sizeof(item));
  t->value=e;
  t->next=l;
  return(t);  }
```

```
list cons_tail(element_type e, list l)
{ /* funzione iterativa che inserisce un
   elemento e in fondo a una lista l */

  list prec, aux;
  list root=l;

  aux=(list)malloc(sizeof(item));
  aux->value=e;
  aux-next=NULL;
  if (l==NULL) return aux;
        /* inserimento in lista vuota */
  else
    {while (!(l==NULL)) /*lista non vuota*/
        { prec=l ;
          l=l->next; }
     prec->next=aux; /*inserisce in fondo*/
     return root; /* restituisce radice*/
    }
}
```

```
list   ordins(element_type el, list root)
{/* inserimento ordinato - iterativo */
 element_type  e;
 int trovato=0;
 list aux, prec, current;
          /* inserimento ordinato con
             possibili duplicazioni */
  aux=(list)malloc(sizeof(item));
  aux->value = el;
  aux->next = NULL;

  if (root==NULL)    /* primo elemento */
       root=aux;
  else
   {current=root;
    while((current!=NULL)&& !trovato)
       {if (current->value < el)
          { prec=current;
            current=current->next; }
        else trovato=1; }

    if (current==NULL)
         {prec->next=aux;}   /*in fondo */
    else
         if (current==root)
             {aux->next=root;
              root=aux;}    /* in testa */
         else
             {prec->next=aux;
                   /* inserisce in mezzo */
              aux->next=current;}
       }
   }
 return root;
}
```

Calcolo della lunghezza di una lista:

L'operazione di calcolo della lunghezza di una lista si realizza con la scansione della lista, a partire dalla radice:




```
int length_it(list l)
/*  versione iterativa  */
{ int sum=0;
  while (l!=NULL)
       {sum++;
        l=l->next;}
return sum;  }
```

Attraverso il parametro formale **`l`**, aggiornato ad ogni iterazione, scandiamo la lista e per ogni nodo incrementiamo la variabile **`sum`**, sino a che si raggiunge la fine della lista (**`NULL`**).

La definizione (non tail) ricorsiva della stessa operazione è data dalla funzione che segue:

```
int length(list l)
/*  versione ricorsiva, non tail  */
{ if (l==NULL) return(0);
  else
   { return(1 + length(l->next)); }
}
```

Esercizio 8.1.1

Scriverne una versione tail ricorsiva della funzione che calcola la lunghezza di una lista:

```
int length_tr(list l)
/*  versione tail ricorsiva */
{ int sum=0;
  return l_tr_aux(l, sum) ; }

int l_tr_aux(list l, int s)
/*  versione tail ricorsiva */
{ if (l==NULL) return(s);
  else
    { return l_tr_aux(l->next,s+1); }
}
```

Il tipo di dato astratto lista semplice:

Piuttosto che realizzare le singole operazioni a seconda delle necessità, costruiamo un ***tipo di dato astratto*** lista semplice.

Il tipo di dato astratto ***lista*** è definito a partire da tipi più semplici ed è caratterizzato da operazioni di ***costruzione*** e ***selezione*** sul tipo lista.

Una lista semplice è un tipo di dato astratto <S,Op,C> dove:

- S= (list, element_type, boolean)
  list è il dominio di interesse, element_type il dominio degli elementi che formano la lista.

- Op = (cons, head, tail, empty)
  cons: elem × list → list (costruttore)
  head: list → elem (o testa)
  tail: list → list (o coda)
  empty: list → boolean (o vuota)

- C={emptylist}
  costante che denota la lista senza elementi.

Pochi linguaggi possiedono il tipo lista (LISP, Prolog).

Per gli altri si costruisce a partire da altre strutture dati: ***vettori*** o ***puntatori*** in C o in Pascal, vettori e matrici in FORTRAN.

Esempio:

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

head : lista -> elemento

Restituisce il primo elemento della lista data: head(L) = 'a'

tail : lista -> lista

Restituisce la coda della lista data: tail(L) = ['b','c']

cons : elemento, lista -> lista

Restituisce la lista data con in testa l'elemento dato:
cons('d',L) = ['d', 'a', 'b', 'c']

empty : lista -> boolean

Restituisce true se la lista data e' vuota, false altrimenti
empty(L) = false

emptylist : -> lista

Restituisce una lista vuota.
cons('a', emptylist) = ['a']

Definizione induttiva del dominio list:

Ogni valore del tipo lista semplice o è una sequenza vuota di elementi oppure è una sequenza formata da un elemento del dominio *elem*, seguito a sua volta da un valore del tipo lista semplice.

Definizione utile per esprimere semplici algoritmi ricorsivi su strutture a lista (***operazioni derivate***).

Operazioni derivate come:

inserimento (ordinato) di elementi,
concatenamento di liste,
stampa degli elementi di un lista,
ribaltamento di una lista,
ricerca di un elemnto in una lista etc.
si possono definire usando le primitive precedenti.

Come realizzare l'ADT lista semplice?

Realizzando un "componente" software (**`list.h`** e **`list.c`**) che esporta l'identificatore di tipo **`list`** e le operazioni (primitive e derivate) associate.

Esercitazione 8.2: Creazione di un modulo lista

Definiamo un componente (costituito dai file **`list.h`** e **`list.c`**) per la realizzazione del tipo **`list`** come lista di interi.

In particolare, nel modulo sono definite le operazioni primitive di:
creazione della lista vuota (**`emptylist`**),
verifica se la lista è vuota (**`empty`**),
aggiunta di un elemento in testa alla lista (**`cons`**),
lettura dell'elemento in testa alla lista (**`head`**),
estrazione della coda della lista (**`tail`**)

e le operazioni derivate che consentono di:
stampare una lista, in versione iterativa (**`showlist`**);
calcolare la lunghezza di una lista, in versione sia ricorsiva (**`length`**) sia iterativa (**`length_it`**);
verificare se un elemento appartiene a una lista, in versione sia ricorsiva (**`member`**) sia iterativa (**`member_it`**);
effettuare la copia di una lista (**`copy`**);
cancellare un elemento da una lista (**`delete`**);
concatenare due liste per produrne una terza (**`append`**);
effettuare l'inversione di una lista, sia in versione ricorsiva (**`reverse`**) sia tail ricorsiva (**`nreverse`**).

L'interfaccia del modulo `list`

```
/* LIST INTERFACE  - file list.h*/
typedef  int element_type;
typedef struct list_element
     { element_type  value;
       struct list_element  *next;
     } item;
typedef  item* list;
typedef  int boolean;

/*  PROTOTIPI DI FUNZIONE */
list emptylist();
boolean empty(list);
list cons(element_type, list);
element_type head(list);
list tail(list);

void showlist(list);
int   length(list l);
int   length_it(list l);
boolean member(element_type, list);
boolean member_it(element_type, list);
list copy (list l);
list delete(element_type el, list l);
list  append (list l1, list l2);
list reverse (list l);
list nreverse(list root);
```

Esercitazione 8.1 (cont): Creazione di una lista da ingresso usando un modulo `list`

Riscriviamo il main che acquisisce una sequenza di caratteri terminata da EOF (nell'ipotesi che invece **`element_type`** in **`list.h`** sia definito come sinonimo del tipo **`char`**):

```
/* MAIN  - file mainlist.c */

#include <stdio.h>
#include "list.h"
/* include  l'interfaccia del
   componente list */

void main(void)
{list root=emptylist(), aux;
 element_type n;

 while (((n=getchar()) != EOF)
     root=cons(n,root);

 showlist(root);
 showr_l(root);
}
```

Esercitazione 8.2: Creazione di un modulo lista

L'utilizzo del modulo `list`

Realizzare un programma che acquisisca da ingresso una sequenza di interi e

```
/* MAIN  - file mainlist.c */
#include <stdio.h>
#include "list.h" /* include interfaccia */

void main(void)
{/* dich. variabile di tipo list */
 list L1,L2;
 int i;
 /* inizializzazione della lista */
 L1=emptylist();
```

crei una lista (**L1**) a partire da tale sequenza, aggiungendo gli elementi in testa alla lista;

```
   do
     {printf("\n Valore intero?:\t");
      scanf("%d",&i);
      L1 = cons(i,L1);
     } while (i!=0);
```

stampi la lista **L1**;

```
   showlist(L1);  /* stampa */
```

calcoli e stampi la lunghezza della lista **L1**;

```
   printf("Lunghezza: %d\n",length(L1));
```

copi la lista **L1** in una seconda lista **L2** e ne determini la lista inversa;

```
   L2=copy(L1);
   L2=nreverse(L2);
   printf("Lista invertita:\n");
   showlist(L2);
```

costruisca una nuova lista costituita dal concatenamento della prima e della sua inversa (lista palindroma).

```
/* creazione palindroma */
   L2=append(L1,L2);
   printf("Lista palindroma:\n");
   showlist(L2);
```

Infine, letto un valore a terminale, il programma deve determinare se l'elemento appartiene alla lista e, in questo caso, cancellarlo dalla lista.

```
/* ricerca elemento */
   printf("\n Valore da cercare?:\t");
   scanf("%d", &i);
   if (member(i,L1))
     {printf("trovato; nuova lista:\n");
            L1=delete(i,L1);}
   showlist(L1);
}
```

La realizzazione del modulo list

```
/* LIST IMPLEMENTATION - list.c */
#include <stdio.h>
#include <stdlib.h>
#include "list.h"

/* OPERAZIONI PRIMITIVE */
list  emptylist()
/* costruttore lista vuota */
{ return NULL; };

boolean  empty(list l)
/* verifica se lista vuota */
{ return (l==NULL); };

list  cons(element_type e, list l)
{ list t;
/* aggiunge in testa alla lista */
  t=(list)malloc(sizeof(item));
  t->value=e;
  t->next=l;
  return(t);  }

element_type  head(list l)
/* selettore testa lista */
{ if (empty(l))
          { abort(); return(0); }
  else    { return (l->value); }
}
```

```
list  tail(list l)
/* selettore coda lista */
{ if (empty(l))
           { abort(); return(0); }
  else     { return (l->next); }
}

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

Operazioni derivate:

L = [1,2,3] L1 = [4]

length : lista -> int

Restituisce la lunghezza di una lista
Es. length(L) = 3

length(L): 0, se empty(L)
1+length(tail(L)), altrimenti

```
int   length_it (list l)
/* Lunghezza lista (iterativa) */
{ int N=0;
  while (!empty(l))
    { N++;  l=tail(l); }
  return(N);}

int   length(list l)
/* Lunghezza lista (ricorsiva) */
{ if (empty(l)) return(0);
  else { return(1+length(tail(l))); }
}
```

È una funzione ***tail ricorsiva***? No, perché dopo la chiamata ricorsiva viene eseguita l'operazione di somma.
Scriverne una versione tail ricorsiva (Esercizio 8.2.1, idem Es. 8.1.1)

Operazioni derivate (cont.):

L = [1,2,3] L1 = [4]

member : elemento, lista -> boolean

Verifica se un elemento appartiene ad una lista
Es. member(2,L) = true

member(el,L): false, se empty(L)
true, se el==head(L)
altrimenti member(el, tail(L))

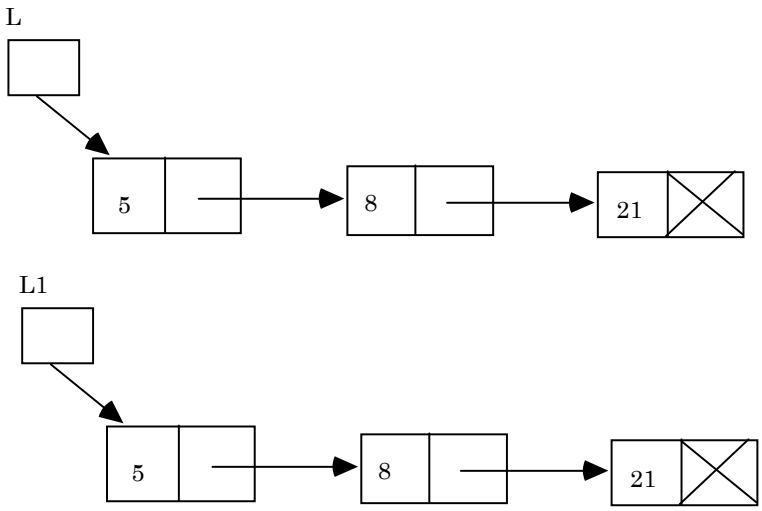
```
boolean member_it (element_type el,
                                   list l)
/* ricerca - iterativa  */
{ while (!empty(l))
     { if (el==head(l)) return(1);
            else  l= tail(l); }
  return(0);  }

boolean member(element_type el, list l)
/* appartenenza - ricorsiva  */
{ if (empty(l)) return(0);
  else { if (el==head(l)) return(1);
         else return(member(el,tail(l))); }
}
```

Operazioni derivate (cont.):

copy : lista -> lista

Crea una copia di una lista
(non c'è condivisione di struttura)

L1=copy(L)



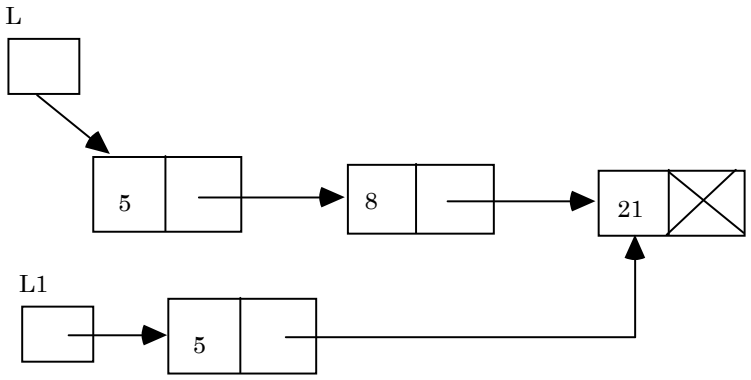

```
list copy (list l)
/* crea una copia della lista l */
{ if (empty(l)) return l;
  else return(cons(head(l),copy(tail(l))
));
}
```

Scriverne una versione iterativa (Esercizio 8.2.2)

Operazioni derivate (cont.):

delete : elemento, lista -> lista

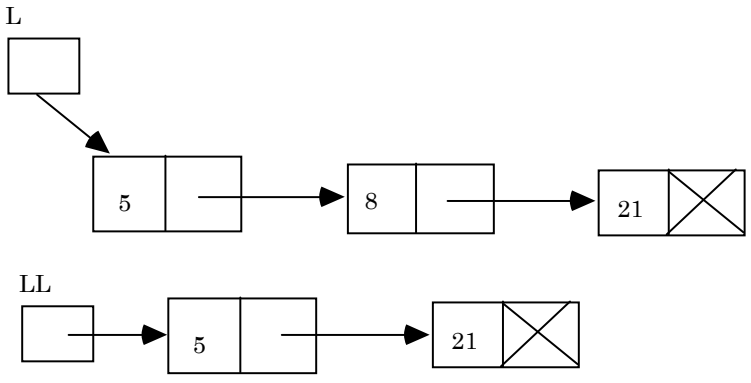
Restituisce una lista che differisce da l per l'elemento e, non altera l L1= delete(8,L)



(c'è condivisione di struttura)

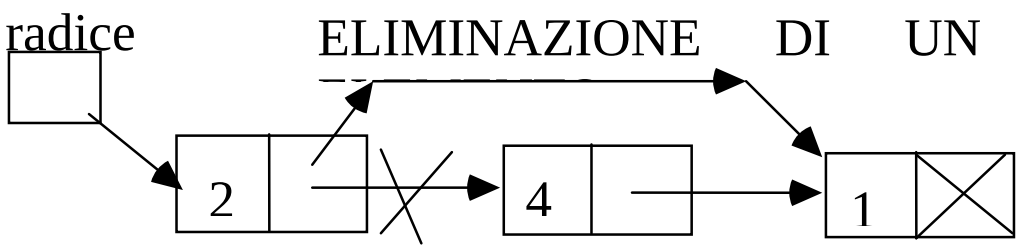
```
list  delete(element_type el, list l)
{ if (empty(l)) return(emptylist());
  else
   {if (el==head(l))return(tail(l));
    else return(cons(head(l),
                     delete(el,tail(l))) );}
}
```

Per non avere condivisione di parte della struttura dati L:
LL= delete(8,copy(L))

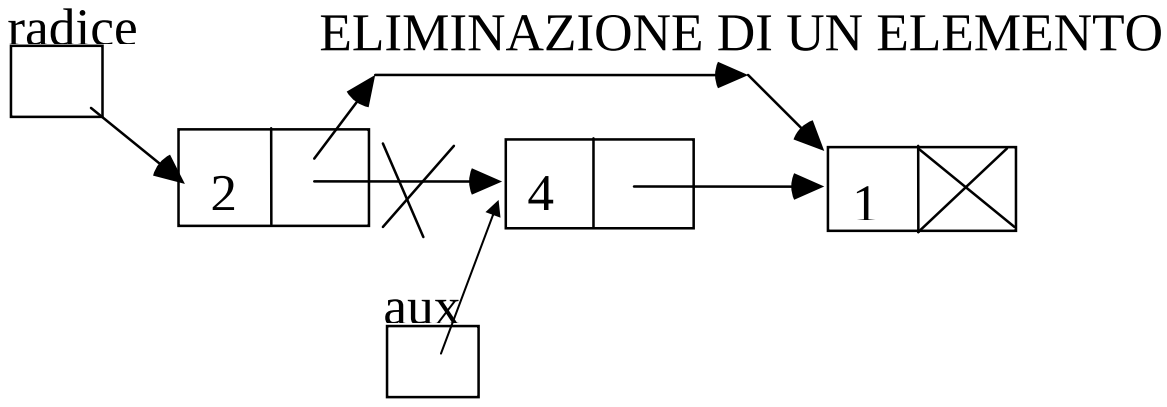


Esercizio 8.2.3:
Scrivere una versione iterativa della procedura delete.

```
list  delete(element_type el, list l)
{ list prec, current, aux;
  if (empty(l)) return(emptylist());
  else
   {current=l;
    while(!(head(current)==el)
            &&(!empty(current)))
       { prec=current;
         current=tail(current);}
    if (!empty(current))
         if (current==l)
             {return tail(l);
              /* elimina il primo */
             }
         else
             {prec->next=current->next;
              return l;
                /* elimina in mezzo */
             }
    else return l;
}
```



```
/* con rilascio di memoria */

list  delete(element_type el, list l)
{ list prec, current, aux;
  if (empty(l)) return(emptylist());
  else
   {current=l;
    while(!(head(current)==el)
           &&(!empty(current)))
       { prec=current;
         current=tail(current);}
    if (!empty(current))
         if (current==l)
             {l=tail(l);
              free(current);
              return l;
              /* elimina il primo */
             }
        else
             {
              prec->next=current->next;
              free(current);
              return l;
                /* elimina in mezzo */
             }
   else return l;
}
```

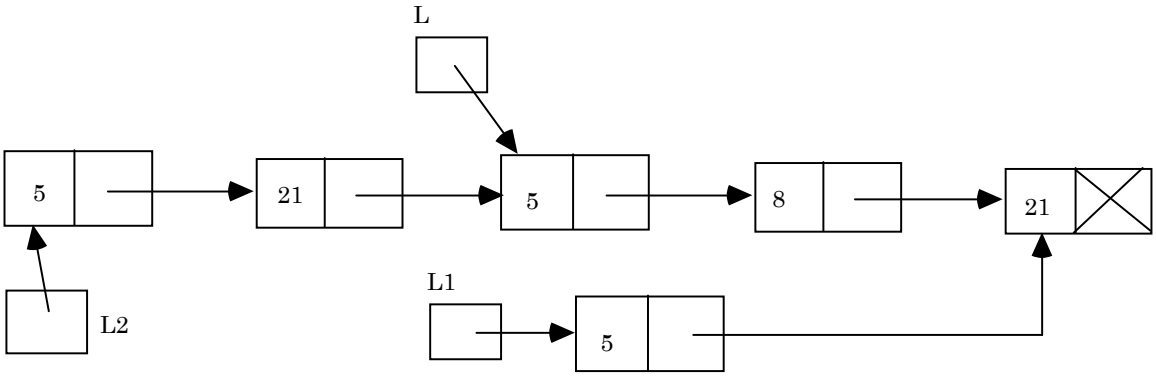


Operazioni derivate (cont.):

append : lista, lista -> lista

Concatena due liste
append(L1,L2):
L2, se empty(L1)
cons(head(L1), append(tail(L1), L2), altrimenti

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



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

Con operazioni definite in questo modo due puntatori a lista possono condividere parte della struttura dati (***structure sharing***).

```
list  append (list l1, list l2)
{ /* concatenamento - ricorsiva    */
 if (empty(l1)) return(l2);
              /* return(copy(l2)) */
  else return(
cons(head(l1),append(tail(l1),l2)) );
}
```

Scriverne una versione iterativa (Esercizio 8.2.4)

Operazioni derivate (cont.):

reverse : lista -> lista

Ribalta una lista (lista inversa)
Es. reverse(L) = [3,2,1]

reverse(L): /* lista inversa */
emptylist, se empty(L)
append(reverse(tail(L)), cons(head(L),emptylist)),
altrimenti

L1=reverse(L)

```
list reverse (list l)
{ /* inversione lista - ricorsiva    */
  if (empty(l)) return(emptylist());
  else return(append(reverse(tail(l)),
         cons(head(l),emptylist())) );
}
```

Una versione iterativa si ottiene scandendo la lista **l** e riallocandone man mano gli elementi in testa a una lista ausiliaria (inizialmente vuota):

```
list reverse_it (list l)
{ /* inversione lista - iterativa    */

  list aux=emptylist();

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

  return aux;

}
```

Scriverne una versione tail ricorsiva della reverse (Esercizio 8.2.5)

Nota Bene:

Quando per le liste si utilizza la rappresentazione collegata realizzata con **strutture** e **puntatori** è importante rilasciare memoria man mano che questa non viene più utilizzata.

Il recupero di spazio di memoria non più utilizzato da liste prende il nome di ***garbage collection***.

Se fatto “metodicamente” con la procedura **free**, si dice “on the fly”.

Altrimenti, periodicamente si esegue un algortimo detto “mark and sweep”: si marcano le celle dello heap che contengono variabili dinamiche usate correntemente dal programma (***dereferencing***), si recuperano nella lista libera tutte le celle dello heap non marcate, si toglie la marcatura alle altre.

Esercizi d'esame:

ESERCIZIO su LISTE

Un file sequenziale di tipo testo (ESPR.TXT) contiene stringhe (una per linea) costituite dai caratteri {a,b,c,*,+}. Si realizzi un programma C che:

a) riconosca le stringhe del file uguali a "a*b+c" oppure a "a+b*c".

b) inserisca i numeri di linea delle stringhe accettate in una lista a puntatori, L1;

c) produca, a partire dalla lista L1 generata precedentemente, un file (UNICHE.TXT) di tipo testo in cui ogni stringa accettata compare una sola volta ed è seguita, sulla stessa linea, dalle sue posizioni originali nel file ESPR.TXT.

Esempio:

ESPR.TXT:

```
a*b+c
a**b+
a+b*c
a*b+c
a+b*c
```

UNICHE:

```
a*b+c   4   1
a+b*c   5   3
```

Strutture dati:

Le strutture dati risultano del tipo:

```
typedef  struct
            {char stringa[20];
            int  pos;} el_type;
typedef struct nodo
          {el_type  value;
           struct nodo *next;  } NODO;
typedef NODO* list;
```

Struttura del programma:

La lista L1 viene inizializzata a lista vuota. La variabile contatore di linea vale 0.

- Domanda a) e b):

Deve essere eseguito un ciclo di lettura (**while (!eof(f))**) delle stringhe dal file ESPR.TXT (assegnato alla variabile **f** ed aperto in lettura).

Per ciascuna stringa letta (memorizzata in una variabile **s**):

- si incrementa il valore della variabile contatore di linea (che mantiene il numero di riga),
- si controlla se la stringa letta è uguale ad una delle due date.

In questo caso (Domanda b), si inserisce la coppia <s, numero di linea> nella lista L1.

- Domanda c):

Al termine del ciclo di lettura, si produce dalla lista L1 il file UNICHE.TXT.

Esercizio 8.2.6: Inserire gli elementi nella lista ordinati secondo l'ordinamento lessicografico.

File utilizzati:

main.c	programma principale
el.h	file header del tipo di dato astratto el_type
el.c	implementazione
list.h	file header del tipo di dato astratto lista
list.c	implementazione della libreria liste

Nel ***project file*** vanno menzionati tutti i file con estensione ***.c*** utilizzati.

```
/* MAIN FILE */
#include "lists.h"
#include <stdio.h>
#include <string.h>
void  main ()
{  list  Laux,  L1 = emptylist();
   el_type EL;
   char  s[30];
   int  pos,i;
   FILE  *f, *g;

   f = fopen("ESPR.TXT", "rt");

   /* DOMANDA a)
      CICLO DI SCANSIONE DEL FILE */

   /* ricerco la prima stringa */
   pos=0;
   while (!feof(f))
    { if (fgets(s,i,f)!=0)
     {pos++;
      if (strcmp(s,"a*b+c\n")==0)

         { EL = crea_el(s,pos);

         /* DOMANDA  b)
        INSERIMENTO NELLA LISTA */
         L1 = cons(EL,L1);
        /* L1 = ordins(EL,L1); /*   }
      }  }
```

```
/* ricerco la seconda stringa */

   rewind(f);
   pos=0;
   while (!feof(f))
    { if (fgets(s,i,f)!=0)
     {pos++;
      if (strcmp(s,"a+b*c\n")==0)
         { EL = crea_el(s,pos);

         /* DOMANDA  b)
             INSERIMENTO NELLA LISTA */
         L1 = cons(EL,L1);   }
      }
   }

   fclose(f);
```

```
/* DOMANDA  c)  */

   g = fopen("UNICHE.TXT", "wt");

   Laux=L1;
   while (!empty(L1))
   {EL=head(L1);
    for(i=0;i<strlen(EL.stringa)-1;i++)
         fprintf(g,"%c",EL.stringa[i]);

    fprintf(g,"\t%d", EL.pos);
    Laux=tail(L1);
    while (!empty(Laux)  &&
           (strcmp(Laux->value.stringa,
                 L1->value.stringa)==0))
     {
      fprintf(g,"\t%d",Laux->value.pos);
      Laux=tail(Laux);
      }
    fprintf(g,"\n");
    L1=Laux;
   }
fclose(g);
}
```

```
/* ELEMENT TYPE - file el.h*/

typedef  struct
         {char stringa[20];
         int  pos;}  el_type;

typedef  enum {false,true}  bool;

el_type  crea_el (char s[], int);
```

```
/* ELEMENT TYPE - file el.c*/
#include "el.h"
#include <stdio.h>
#include <string.h>

el_type  crea_el (char s[], int  p)
{el_type  EL;
  strcpy(EL.stringa, s);
  EL.pos = p;  return(EL);  }
```

```
/* LIST  INTERFACE  - file list.h*/
#include "el.h"
typedef struct nodo
          {el_type  value;
           struct nodo *next;  } NODO;
typedef NODO* list;

/* operatori esportabili */
list  emptylist();
bool  empty(list l);
el_type  head(list l);
list  tail(list l);
list  cons(el_type e, list l);
```

```
/* LIST IMPLEMENTATION - file lists.c */
#include <stdio.h>
#include <stdlib.h>
#include "lists.h"

/* OPERAZIONI PRIMITIVE */
list  emptylist()
{ return NULL; };

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

el_type  head(list l)
{ if (empty(l)) { abort(); }
  else  return(l->value); }

list  tail(list l)
{ if (empty(l)) { abort(); return(0); }
  else          { return (l->next); }
}

list  cons(el_type e, list l)
{list t;
 t=(list)malloc(sizeof(NODO));
 t->value=e;
 t->next=l;
 return(t);
}
```

III Esercitazione in Laboratorio

append() : "funzione (iterativa) che effettua il concatenamento di due liste"; si richiede di implementare tale funzione in modo tale da concatenare l1 ad l3 (l3=[l3|l1]). Per osservare il corretto funzionamento della append() stampare la nuova versione di l3;
deleteValue() iterativa: "funzione che elimina dalla lista (passata come parametro) l'elemento il cui valore è specificato a tale funzione come parametro di input" ; *utilizzare tale funzione per eliminare dalla lista l3 l'(eventuale) elemento n=length(l1), con length() che è una funzione tail-ricorsiva (da implementare);*
deleteOrd() iterativa : "funzione che effettua la cancellazione dell'elemento della lista che presenta il campo *value* uguale a quello che viene passato a questa funzione come parametro (considerando che si ha una lista ordinata in base al campo *value* stesso)" ; *per provare questa funzione creare un nuovo main() nel quale si leggano da tastiera un certo numero n di interi (terminati dal valore 0) memorizzandoli su una lista l1 con inserimento ordinato (insOrd()) e successivamente invocare la deleteOrd() per verificare il suo corretto funzionamento*;

Leggere da terminale un valore intero (*value*) e verificare se tale valore intero compare in L e quante volte compare. Stampare poi a video il risultato della funzione chiamata. è nella ricerca occorre considerare che la lista è ordinata rispetto al campo chiave *value*.
[DA FARE AUTONOMAMENTE] Data la lista di valori interi ordinata creare una nuova lista contenente tutti gli elementi della lista considerata aventi campo *value* <= (oppure >=) di un dato valore intero acquisito da input.
Dove la append e la funzione 5. devono essere realizzate **senza condivisione di struttura.**

Scritto del 21 Giugno 2007

Esercizio n. 2

Due file di tipo binario (UNO.DAT e DUE.DAT) contengono due insiemi di record, ciascuno costituito da un nome, un cognome (stringhe di 16 caratteri), un codice numerico (intero) e un email (stringa di 20 caratteri).
Non ci ripetizioni dello stesso nome e cognome all'interno di ciascun file, ma lo stesso nome e cognome può comparire in entrambi i file, eventualmente con codice numerico diverso.
Si scriva un programma C **strutturato in (almeno) tre funzioni** dedicate rispettivamente:
creare una lista L a partire dal contenuto dei due file: in L si inserisce ogni coppia nome e cognome presente in entrambi i file con lo stesso codice, il codice e le due email. Si crei la lista L in modo che sia ordinata lessicograficamente in base al cognome e al nome;
leggere a terminale un cognome e, accedendo a L, verificare se

tale cognome compare in L e quante volte compare, stampando

poi a video il risultato della funzione chiamata;

accedendo a L, mediante una funzione ricorsiva, si conti il

numero record contenuti in L e lo si stampi poi a video.

È possibile utilizzare ***librerie C*** (ad esempio per le stringhe).
Nel caso si strutturi a moduli l'applicazione qualunque ***libreria utente*** va riportata nello svolgimento.

Esercizi proposti a lezione per Lunedì 2 Febbraio 2009:

Esercizio 8.2.1: Calcolo della lunghezza di una lista, in versione tail ricorsiva:

```
int   length(list l)
/* Lunghezza lista (ricorsiva) */
{ if (empty(l)) return(0);
  else { return(1+length(tail(l))); }
}

int   length_tr(list l)
/* Lunghezza lista (tail ricorsiva) */
{ return length_tr_aux(0,l))); }
}

int   length_tr_aux(int n,list l)
/* Lunghezza lista (ricorsiva) */
{ if (empty(l)) return n;
  else { return
           length_tr_aux(n+1,tail(l)); }
}
```

Esercizio 8.2.2: Scrivere una versione iterativa della funzione copy (qui si utilizza una funzione auliaria, iterativa, che inserisce in fondo a una lista)

```
list copy_it(list l)
{  /* versione iterativa che
   crea una copia della lista l */

 list aux=emptylist();

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

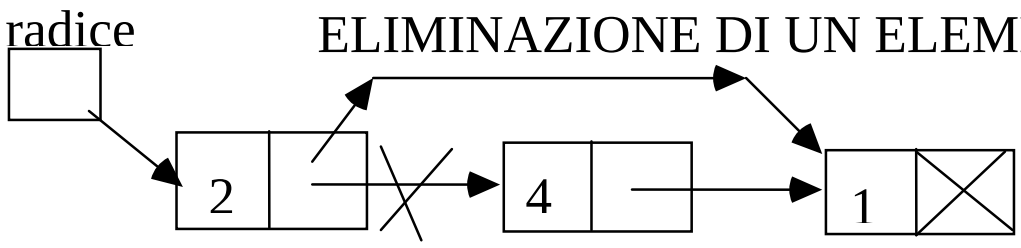
```
list cons_tail(element_type e, list l)
{ /* funzione iterativa che inserisce un
   elemento e in fondo a una lista l */

  list prec, aux;
  list root=l;

  aux=(list)malloc(sizeof(item));
  aux->value=e;
  aux-next=NULL;
  if (l==NULL) return aux;
        /* inserimento in lista vuota */
  else
    { while (!(empty(l)))
        { prec=l ;
          l=tail(l); }
       prec->next=aux;
                  /*inserisce in fondo e*/
       return root; /* restituisce radice*/
    }
}
```

Una versione iterativa della funzione copy si può anche ottenere come segue, inserendo ogni elemento di l incontrato nel ciclo while che scandisce i nodi di l stessa, in una lista aux inizialmente vuota e poi invertendo la lista aux costruita in questo modo:

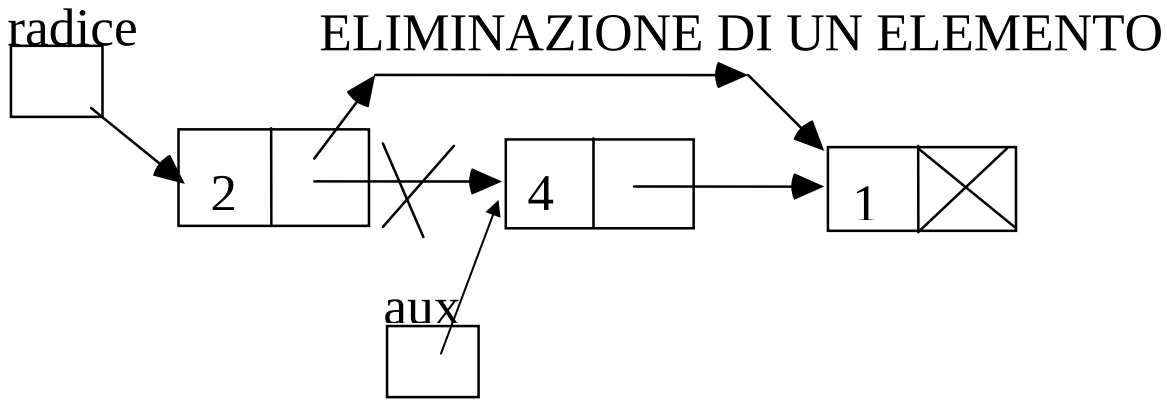
```
list copy_it(list l)
{ /* versione iterativa che
     crea una copia della lista l */

 list p, aux=emptylist();

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

  l=reverse_it(aux);

/* qui occorre rilasciare tutto lo
     spazio allocato per la lista inversa
     referenziata da aux:  */

  p=aux;
  while (!(empty(p)))
      {p=tail(p);
       free(aux);
       aux=p;}
  return l;
}
```

Esercizio 8.2.3:
Scrivere una versione iterativa della procedura delete.

```
list  delete(element_type el, list l)
{ list prec, current, aux;
  if (empty(l)) return(emptylist());
  else   /* l non vuota */
   {current=l;
    while(!(head(current)==el)
           &&(!empty(current))) /*(NOTA)*/
       { prec=current;
         current=tail(current);}
    if (!empty(current))
         if (current==l)
             {return tail(l);
              /* elimina il primo */
             }
         else
             {prec->next=current->next;
              return l;
                /* elimina in mezzo */
             }
    else return l;
}
```



```
/* con rilascio di memoria */

list  delete(element_type el, list l)
{ list prec, current, aux;
  if (empty(l)) return(emptylist());
  else
   {current=l;
    while(!(head(current)==el)
           &&(!empty(current)))
       { prec=current;
         current=tail(current);}
    if (!empty(current))
         if (current==l)
             {l=tail(l);
              free(current);
              return l;
              /* elimina il primo */
             }
        else
             {
              prec->next=current->next;
              free(current);
              return l;
                /* elimina in mezzo */
             }
   else return l;
}
```



Si noti che nella valutazione del codice:

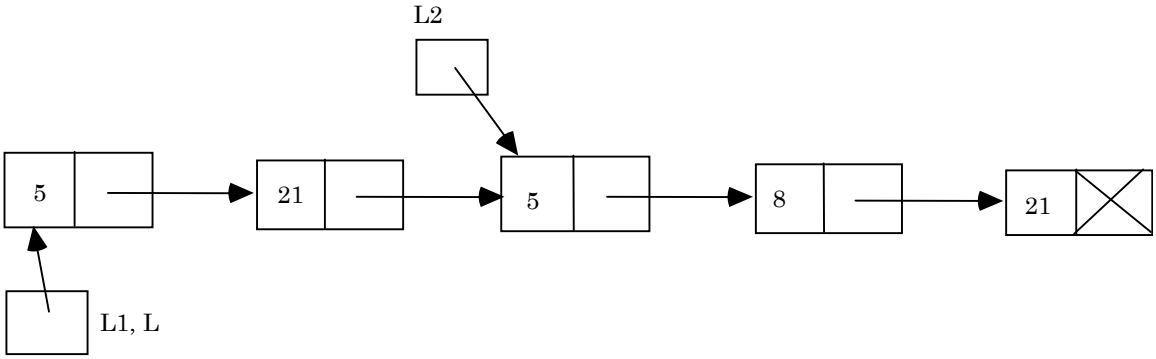
```
while(!(head(current)==el)
            &&(!empty(current))) /*(NOTA)*/
```

indicato con la NOTA commentata nel testo, il test del ciclo while potrebbe valutare head(current) quando current è NULL.

Esercizio 8.2.4: Scrivere una versione iterativa della funzione append.

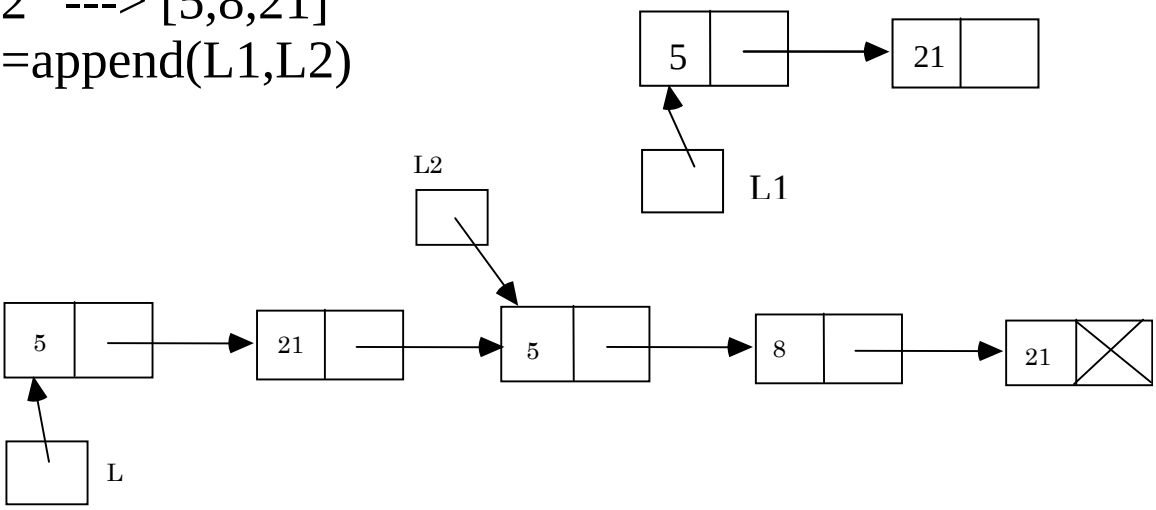
```
list  append_it (list l1, list l2)
{ /* concatenamento - iterativa    */

 list prec=emptylist(),current=l;

 while (!(empty(current)))
    {prec=current;
     current=tail(current);
    }

 if (!(empty(prec)))
      {prec->next=l2;
       return l1;}
 else return l2;

}
```

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



```
list  append_it (list l1, list l2)
{ /* concatenamento - iterativa    */

 list root1,prec=emptylist(),current;

 root1=copy_it(l1);
 current=root1;
 while (!(empty(current)))
    {prec=current;
     current=tail(current);
    }

 if (!(empty(prec)))
      {prec->next=l2;  /* =copy(l2)*/
       return root1;}
 else return l2;       /* =copy(l2)*/

}
```

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



Esercizio 8.2.5: Scrivere una versione tail ricorsiva della reverse

```
list reverse (list l)
{ /* inversione lista - ricorsiva    */
  if (empty(l)) return(emptylist());
  else return(append(reverse(tail(l)),
         cons(head(l),emptylist())) );
}
```

```
list reverse_it (list l)
{ /* inversione lista - iterativa    */

  list aux=emptylist();

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

  return aux;
}
```

```
list nreverse (list l)
{/* inversione lista – tail ricorsiva    */
  nrev_aux(emptylist(),l);
}


list nrev_aux (list aux, list l)
{ if (empty(l)) return(aux);
  else
    return nrev_aux(cons(head(l),aux),
                    tail(l) ) ;
}


  list aux=emptylist();

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

  return aux;

}
```

Esercizio 8.2.6

Inserimento ordinato in una lista, versione ricorsiva nello stile del modulo list :

```
/* ELEMENT TYPE - file el.h*/

typedef  struct
         {char stringa[20];
         int  pos;}  el_type;

typedef  enum {false,true}  bool;

el_type  crea_el (char s[], int);

bool isless(el_type e1, el_type e2);
```

```
/* ELEMENT TYPE - file el.c*/
#include "el.h"
#include <stdio.h>
#include <string.h>

el_type  crea_el (char s[], int  p)
{el_type  EL;
  strcpy(EL.stringa, s);
  EL.pos = p;  return(EL);  }

bool isless(el_type e1, el_type e2)
{return (strcmp(e1.stringa, e2.stringa)<0);
}
```

```
/*inserimento ordinato – ricorsivo */
list  ordins(el_type el, list l)
{         /* inserimento ordinato con
             possibili duplicazioni */
 if (empty(l)) return(cons(el,l));
  else
     {
        if (isless(el,head(l))
              return(cons(el,l));
        else return(cons(head(l),
                 ordins(el,tail(l))) );
     }
}
```

```
list   ordins(el_type el, list root)
{/* inserimento ordinato - iterativo */
 el_type  e;
 int trovato=0;
 list aux, prec, current;
          /* inserimento ordinato con
             possibili duplicazioni */
  aux=(list)malloc(sizeof(NODO));
  aux->value = el;
  aux->next = NULL;

  if (root==NULL)    /* primo elemento */
       root=aux;
  else
   {current=root;
    while((current!=NULL)&& !trovato)
       {if (isless(current->value,el))
          { prec=current;
            current=current->next; }
        else trovato=1; }

    if (current==NULL)
         {prec->next=aux;}   /*in fondo */
    else
         if (current==root)
             {aux->next=root;
              root=aux;}    /* in testa */
         else
             {prec->next=aux;
                   /* inserisce in mezzo */
              aux->next=current;}
       }
   }
 return root;
}
```

```
/* IL MAIN FILE RIVISTO*/
#include "lists.h"
#include <stdio.h>
#include <string.h>
void  main ()
{  list  Laux,  L1 = emptylist();
   el_type EL;
   char  s[30];
   int  pos,i;
   FILE  *f, *g;

   f = fopen("ESPR.TXT", "rt");

   /* DOMANDA a)
      CICLO DI SCANSIONE DEL FILE */

   /* ricerco una delle due stringhe */
   pos=0;
   while (!feof(f))
    { if (fgets(s,i,f)!=0)
     {pos++;
      if ( (strcmp(s,"a*b+c ")==0) ||
           (strcmp(s,"a+b*c ")==0)   )

         { EL = crea_el(s,pos);

         /* DOMANDA  b) CON
        INSERIMENTO ORDINATO NELLA LISTA */
         L1 = ordins(EL,L1);
         }
     }
    }
```

```
    fclose(f);

    /* DOMANDA  c) (identica a prima) */

    g = fopen("UNICHE.TXT", "wt");

    Laux=L1;
    while (!empty(L1))
    {EL=head(L1);
     for(i=0;i<strlen(EL.stringa)-1;i++)
          fprintf(g,"%c",EL.stringa[i]);

     fprintf(g,"\t%d", EL.pos);
     Laux=tail(L1);
     while (!empty(Laux)  &&
            (strcmp(Laux->value.stringa,
                  L1->value.stringa)==0))
       {
        fprintf(g,"\t%d",Laux->value.pos);
        Laux=tail(Laux);
        }
     fprintf(g,"\n");
     L1=Laux;
    }
fclose(g);
}
```