U.D. 9.2

Liste

- Concetti generali - (Es. matrici sparse, tabelle)
- Liste - (statiche e dinamiche)
- Un esempio completo (liste dinamiche)

Fonti:

Antola Dispense del Corso
A. Ciampolini – UNI_BO – Lucidi del corso

Liste

Sequenze (*multi-insiemi* finiti e ordinati) di elementi di un determinato tipo.

Notazione: (*parentetica*)

L = ['a', 'b', 'c']

denota la *lista L dei caratteri 'a', 'b', 'c'*

[5,8,5,21,8]

denota una *lista di interi*

- E` un ***multi-insieme*:** ci possono essere ripetizioni del medesimo elemento.

Rappresentazione concreta di liste semplici

Rappresentazione *statica*:

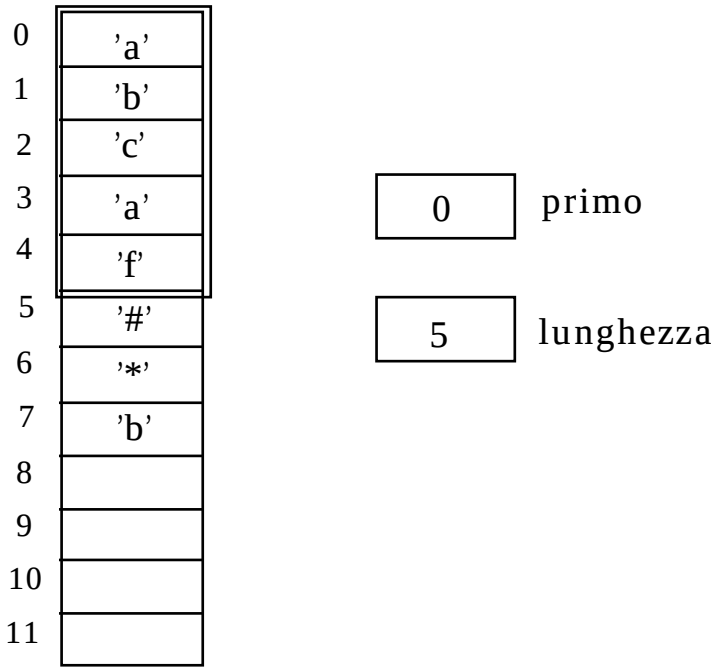
La piu` "banale" utilizza un **vettore mono-dimensionale** in cui sono inseriti gli elementi della lista in modo sequenziale (***rappresentazione sequenziale***).

- La variabile ***primo*** memorizza l'indice del vettore in cui e` inserito il primo elemento.

- La variabile ***lunghezza*** indica da quanti elementi e` composta la lista.

Esempio:

[‘a’,’b’,’c’,’a’,’f’]



Le componenti del vettore con indice:

i >= (*primo*+*lunghezza*)

non sono significative.

Problemi:

- Occupazione della memoria non ottimale.
- Costo elevato di operazioni di inserimento ed estrazione.

Liste: rappresentazione collegata

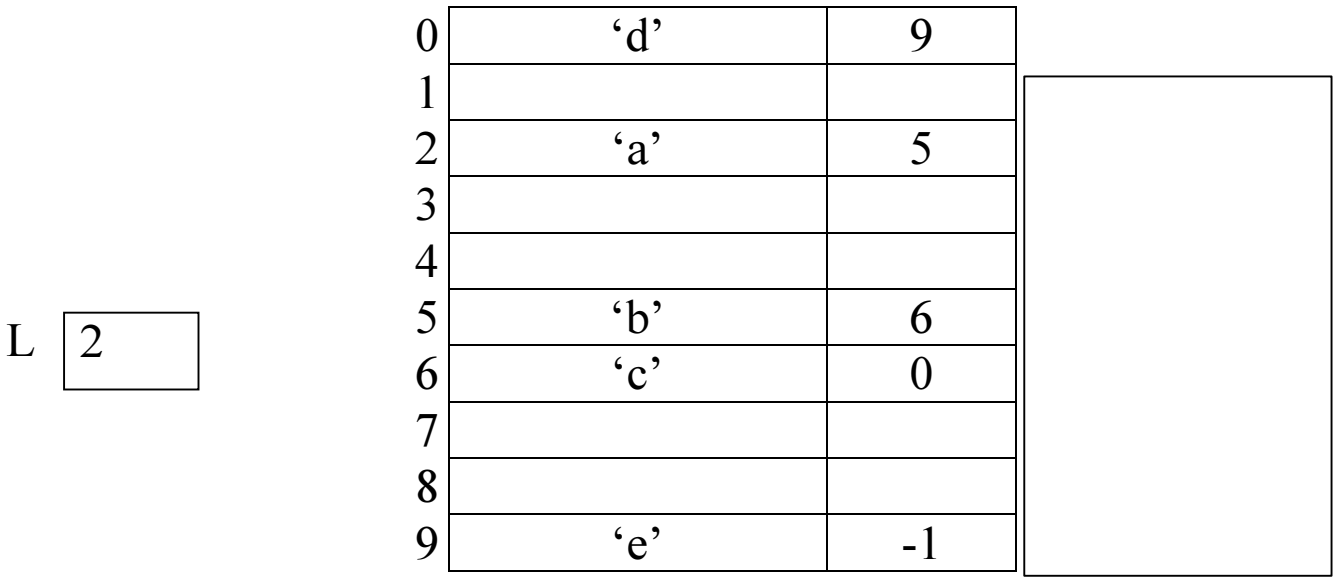
Si memorizzano gli elementi in locazioni di memoria distinte e non adiacenti associando ad ognuno di essi l'informazione (***riferimento***) che permette di individuare la locazione in cui e` inserito l'elemento successivo (**rappresentazione collegata**).

Rappresentazione mediante allocazione statica:

Si memorizzano gli elementi della lista in elementi di un vettore.

- La lista e` rappresentata dall'indice del suo primo elemento.
- La sequenzialita` degli elementi della lista non comporta l'adiacenza delle locazioni di memoria in cui sono memorizzati.

Esempio: **L=['a', 'b', 'c', 'd', 'e']**



Problemi:

- Occupazione della memoria non ottimale.

Liste: rappresentazione collegata

Rappresentazione mediante allocazione dinamica:

Si memorizzano gli elementi in variabili dinamiche distinte (nello ***heap***) associando ad ognuno di essi l'indirizzo della variabile dinamica in cui e` memorizzato l'elemento successivo.

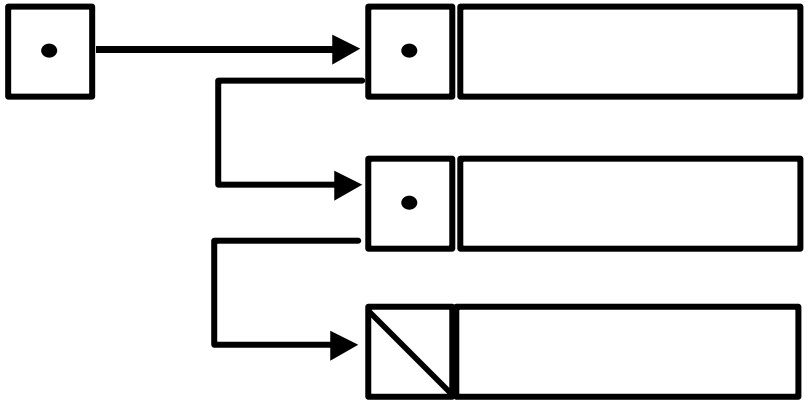
Esempio:

elementi della lista come **nodi (**allocati dinamicamente) e riferimenti come **archi**.

[5, 8, 21]

Linked Lists

Simple Linked List



NULL pointer means end of list

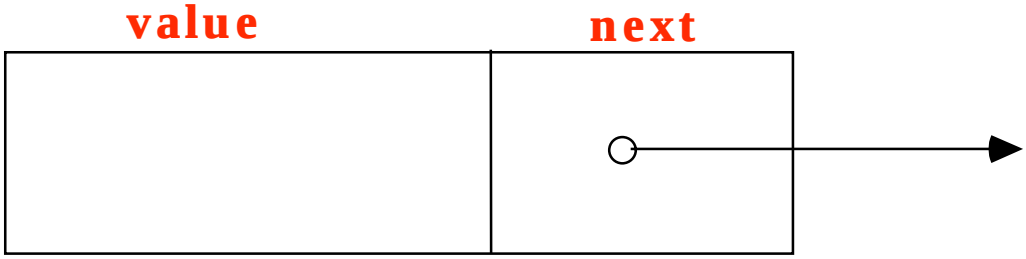
Rappresentazione collegata in memoria dinamica mediante puntatori

In C si utilizzano i ***puntatori*** per ovviare al limite di dimensione della lista dovuto alla dimensione del vettore: la memoria utilizzata viene gestita dinamicamente ed e` esattamente proporzionale al numero degli elementi della lista.

Ciascun elemento della lista e` un record di due campi:

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

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



Liste: Rappresentazione collegata

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

typedef   item* list;
```

- Nella definizione del tipo **struct** *list_element* si fa precedere l'identificatore del tipo l'identificatore alla collezione dei campi: in questo modo e` 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*.

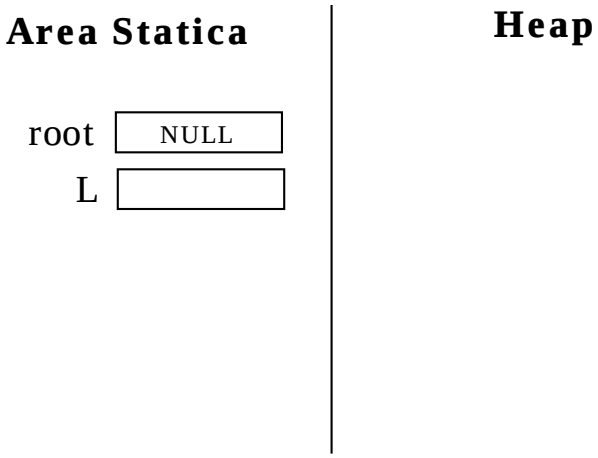
- Infine il tipo *list* e` un puntatore al tipo *item*.

In modo equivalente:

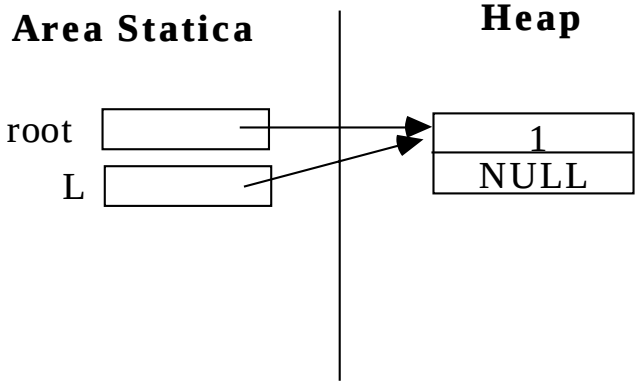
```
struct EL      {    int         value;
                    struct EL      *next;  };
typedef  struct EL      item;
typedef  item       *list;
```

Esempio:

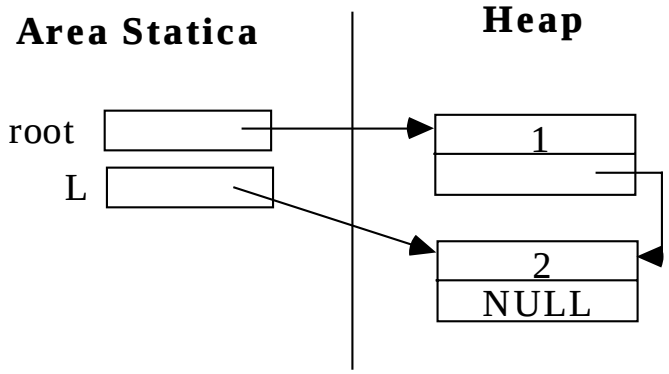
```
#include <stdlib.h>
typedef struct  list_element
  {int     value;
   struct  list_element  *next;
  } item;

typedef   item *list;

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



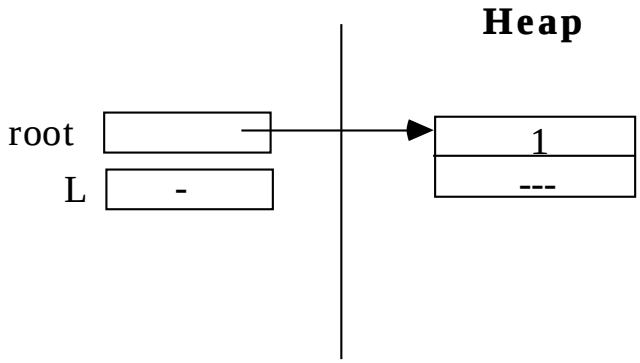

```
  root = (list) malloc(sizeof(item));
  root->value = 1;
  root->next = NULL;
  L = root;
```



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




```
free(L);
```




```
}
```

OPERAZIONI SU UNA LISTA

Per utilizzare una lista è necessario definire le operazioni che agiscono sulla lista.
Queste operazioni rappresentano gli «operatori elementari» che agiscono su variabili di tipo lista (Abstact Data Type). Le operazioni sono dei sottoprogrammi (funzioni o procedure, a seconda dell'operazione).

Operazioni tipiche sulle liste

- **Inizializzazione** modifica la lista
- **Inserimento in testa** modifica la lista
- **Inserimento in coda** modifica la lista
- **Inserimento all'interno** modifica la lista
- **Elimina un elemento** modifica la lista
- **Verifica se lista vuota** *non* modifica la lista
- **Ricerca elemento** *non* modifica la lista

DICHIARAZIONE DI UNA LISTA IN C

1. Dichiarazione del tipo dell'elemento

Il tipo dell'elemento di una lista può essere costruito come segue:

```
struct EL {
          TipoInfo       Info;
          struct  EL     * Next;
          };
```

TipoInfo: può essere di qualsiasi tipo semplice o strutturato, built-in o user-defined.

Con la **ridefinizione** di tipo

```
typedef   struct EL {
                   TipoInfo       Info;
                   struct  EL     * Next;
                   }ElemLista;
```

si definisce un nuovo identificatore di tipo (**ElemLista**) che rappresenta il tipo di un elemento della lista.

Ad esempio:

```
typedef   struct EL {
                   int            Info;
                   struct  EL     * Next;
                   }ElemLista;
```

definisce un tipo di elemento il cui campo **Info** è di tipo intero.

2. Dichiarazione della lista

Una lista è rappresentata dalla sua Testa_di_Lista, cioè da un puntatore ad un tipo ElemLista.

```
ElemLista      *Lista;
```

FUNZIONI CHE REALIZZANO LE OPERAZIONI SULLE LISTE

Alcune operazioni modificano la lista: è quindi necessario che l'effetto del sottoprogramma modifichi lo stato di esecuzione del chiamante:

- la lista è dichiarata come variabile globale e quindi visibile sia al chiamante che al chiamato, oppure
- la lista è passata per indirizzo al chiamato

INIZIALIZZAZIONE

USO DI VARIABILE GLOBALE

```
/*  nella parte dichiarativa globale  */

typedef  struct EL {
                   TipoInfo      Info;
                   struct  EL    * Next;
                   }ElemLista;

ElemLista          *Lista;
```

Chiamata

```
Inizializza( );
```

Definizione della funzione:

```
void Inizializza (void)
{
  Lista=NULL;
}
```

INIZIALIZZAZIONE

PASSAGGIO PER INDIRIZZO

```
/*  nella parte dichiarativa globale  */

typedef  struct EL {
                    TipoInfo      Info;
                    struct  EL    * Next;
                    }ElemLista;


/*  nella parte dichiarativa locale del chiamante*/

ElemLista          *Lista1, *Lista2;
```

Chiamata

```
Inizializza(&Lista1);
Inizializza(&Lista2);
```

Definizione della funzione:

```
void Inizializza (ElemLista **Lista)
{
  *Lista=NULL;
}
```

INSERISCI IN TESTA UN ELEMENTO

Lista non vuota

crea un nuovo elemento	PNuovo=malloc(sizeof(ElemLIsta);
assegna al campo Info del nuovo elemento il valore	PNuovo→ Info=Elem;
assegna al campo Next del nuovo elemento il valore della testa della lista	PNuovo→ Next=*valore testa_lista*;
assegna alla testa della lista l'indirizzo del nuovo elemento	*testa_lista*=Pnuovo;

Vai al Modulo

Lista vuota

INSERISCI IN TESTA UN ELEMENTO: PASSAGGIO PER INDIRIZZO

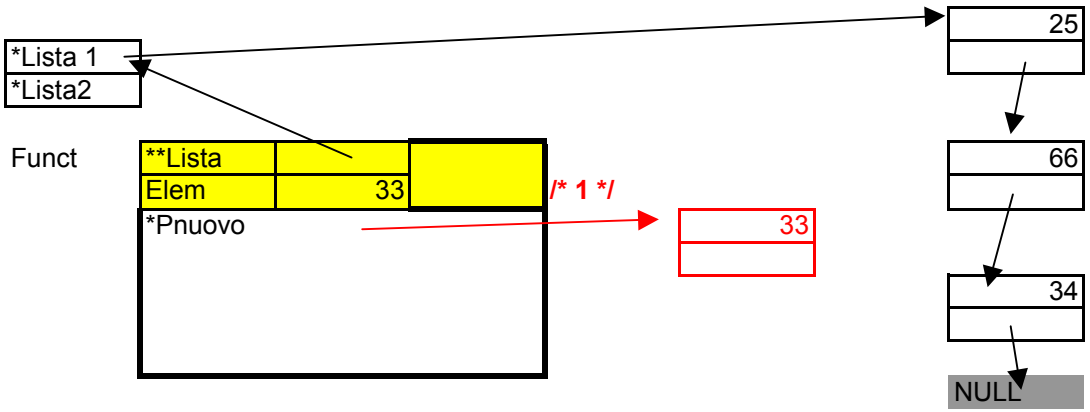
```
/*  nella parte dichiarativa globale  */

typedef  struct EL {
                  TipoInfo      Info;
                  struct  EL    * Next;
                  }ElemLista;

typedef  .......   TipoInfo;

/*  nella parte dichiarativa locale del chiamante*/

ElemLista          *Lista1(, *Lista2);
TipoInfo      Dato1(,Dato2);
……….
scanf(«…..», &Dato1);
…….
```

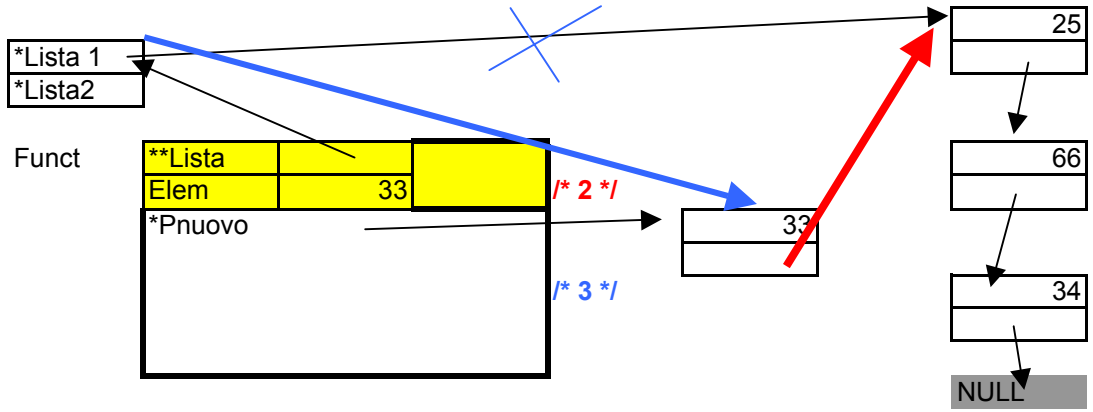
Chiamata

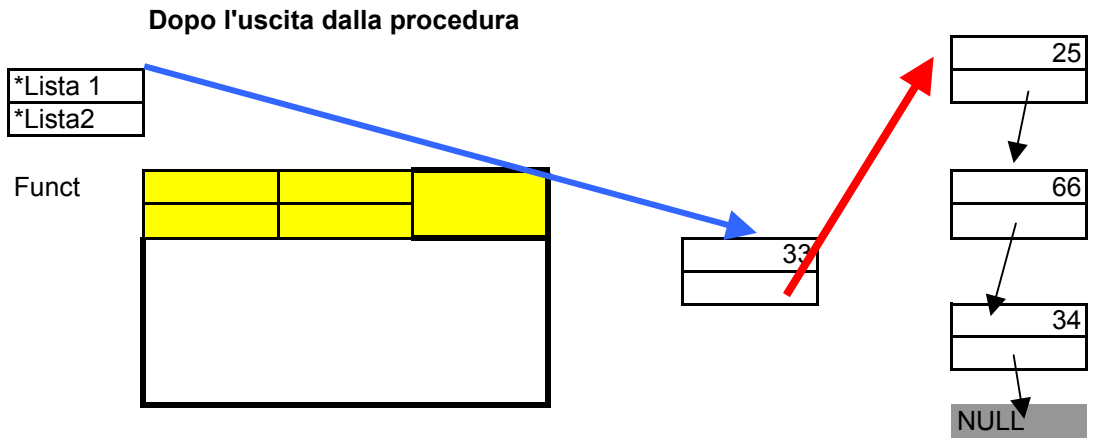
```
Inserisci_in_Testa(&Lista1,Dato1);
```

Definizione della funzione:

```
void   Inserisci_in_Testa(ElemLista   **Lista,   TipoInfo
Elem)
{
  ElemLista   *Pnuovo;

  PNuovo=malloc(sizeof(ElemLista)); /* 1 */
  PNuovo->Info=Elem;
  PNuovo->Next=*Lista; /* 2 */
  *Lista=PNuovo; /* 3 */
}
```


*Lista 1
*Lista2
Funct
**Lista
Elem
33
*Pnuovo
/* 1 */
33
25
66
34
NULL


*Lista 1
*Lista2
Funct
**Lista
Elem
33
*Pnuovo
/* 2 */
/* 3 */
33
25
66
34
NULL


Dopo l'uscita dalla procedura
*Lista 1
*Lista2
Funct
33
25
66
34
NULL

```
#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)
/* inserimento in testa */
{list t;
  t=(list)malloc(sizeof(item));
  t->value=e;
  t->next=l;
  return t ;
}
void main(void)
{list root=NULL, l;
  int i;
  do
    {printf("\n Introdurre valore:\t");
   scanf("%d", &i);
   root = insert(i, root);
    } while (i!=0);
  l=root;        /* stampa */
  while (l!=NULL)
  {printf("\nValore estratto:
         \t%d", l->value);
   l=l->next;
  }
}
```

ELIMINA ELEMENTO CON CAMPO INFO A VALORE PREFISSATO.

Si suppone che la lista contenga al più un elemento con il valore prefissato.

Elemento da eliminare è il primo della lista

Elemento da eliminare può essere in posizione qualsiasi

ELIMINA UN ELEMENTO: PASSAGGIO PER INDIRIZZO

```
/*  nella parte dichiarativa globale  */

typedef  struct EL {
                    TipoInfo      Info;
                    struct  EL    * Next;
                    }ElemLista;

typedef  .......   TipoInfo;

/*  nella parte dichiarativa locale del chiamante*/

ElemLista          *Lista1(, *Lista2);
TipoInfo     Dato1(,Dato2);
```

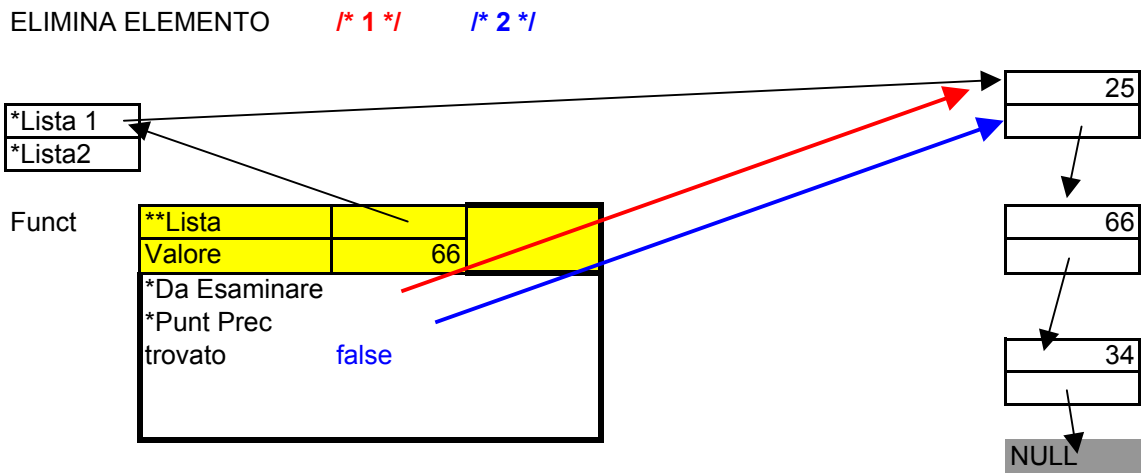
Chiamata

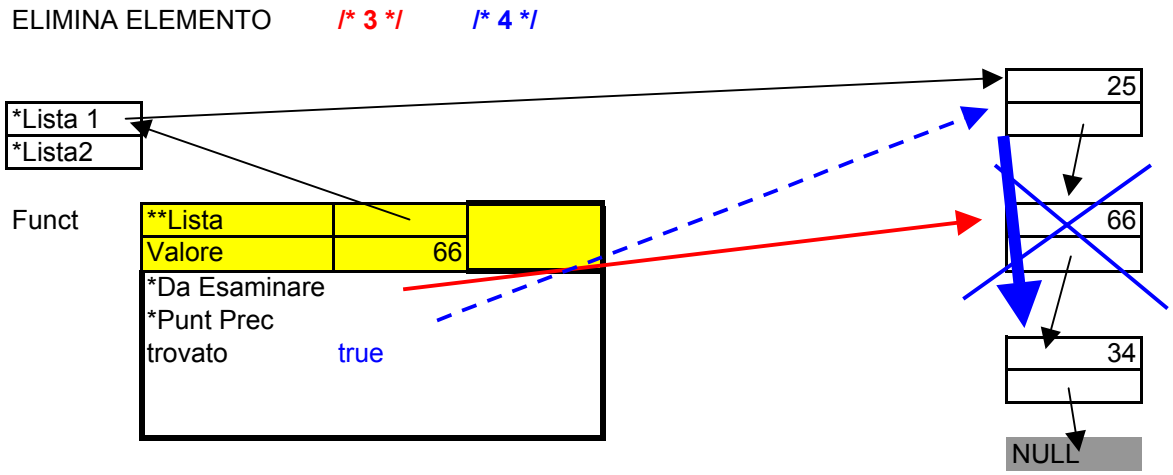
```
Elimina_Elem(&Lista1,Dato1);
```

ELIMINA UN ELEMENTO: PASSAGGIO PER INDIRIZZO

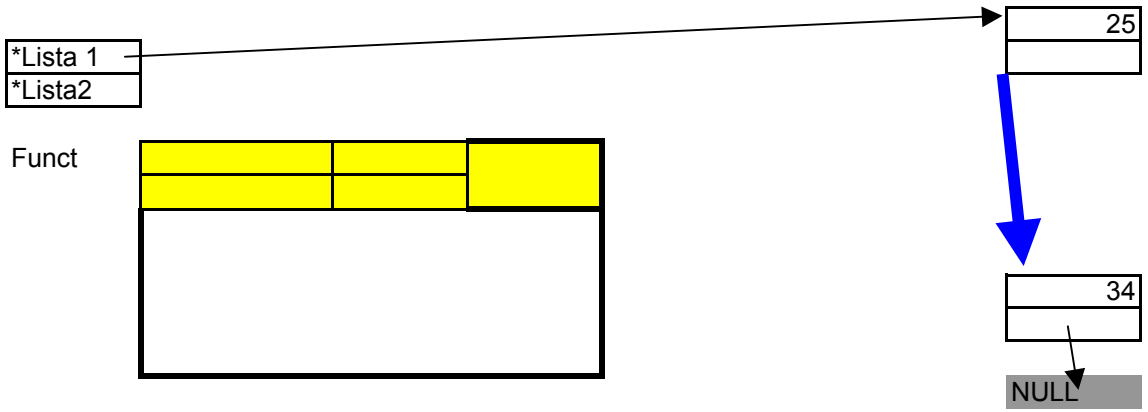
Definizione della funzione:

```
void Elimina_Elem(ElemLista **Lista, TipoInfo Valore)
{
  ElemLista  *Da_Esam;
  ElemLista  *PuntPrec;
  int        trovato;

  if((*Lista)!=NULL)
  {          /* lista non vuota */
   Da_Esam=*Lista; /* 1 */
   if((Da_Esam->Info)==Valore)
     {   /* e' il primo elemento */
      *Lista=Da_Esam->Next;
      free(Da_Esam);
     }
   else /*l'elemento non e' il primo della lista */
     {
      PuntPrec=*Lista;
      trovato=FALSE;   /* 2 */
      while((PuntPrec->Next != NULL) && (!trovato))
        {
         Da_Esam=PuntPrec->Next;  /* 3 */
         if((Da_Esam->Info)==Valore)
           {PuntPrec->Next=Da_Esam->Next;
            free(Da_Esam);
            trovato = TRUE;  /* 4 */
           }
         else /* elemento esaminato non è da eliminare */
           PuntPrec=Da_Esam;
        } /* fine ciclo while */
     } /* fine ricerca all'interno della lista */
  } /* fine lista non vuota */
} /* fine procedura */
```


ELIMINA ELEMENTO
/* 1 */
/* 2 */
*Lista 1
*Lista2
Funct
**Lista
Valore
66
*Da Esaminare
*Punt Prec
trovato
false
25
66
34
NULL


ELIMINA ELEMENTO
/* 3 */
/* 4 */
*Lista 1
*Lista2
Funct
**Lista
Valore
66
*Da Esaminare
*Punt Prec
trovato
true
25
66
34
NULL

ELIMINA ELEMENTO **uscita della procedura**



Il tipo di dato astratto lista semplice

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

- S= (list, elem, boolean)
  dove **list** e` il dominio di interesse, **elem** il dominio degli elementi che formano la lista.

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

- C=(emptylist), costante che denota la lista senza elementi (NULL).

Esempio:

Date tre liste:

[], [52],[6,7,11,21,3,6]

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

Realizzazione del tipo di dato astratto lista

Pochi linguaggi possiedono il tipo concreto lista (LISP, Prolog); per gli altri si costruisce a partire da altre strutture dati.

Ad esempio:

- **puntatori** in C
- vettori e matrici in FORTRAN.

Realizzazione:

rappresentazione delle operazioni associate al tipo nel linguaggio scelto.

Operazioni primitive sulle liste

L=[‘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 (il resto della lista, tolto il primo elemento):

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

Realizza la costante ”lista vuota”.

Liste

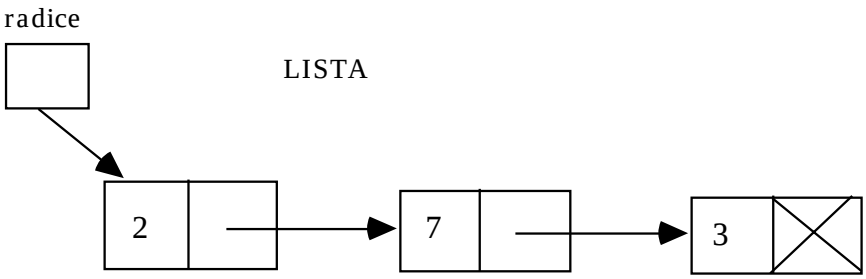
Definizione induttiva del dominio list:

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

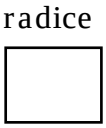
natura **ricorsiva** del tipo ***list***:

- Questa definizione e` utile per esprimere semplici algoritmi ricorsivi su strutture a lista (***operazioni derivate***).

Liste semplici



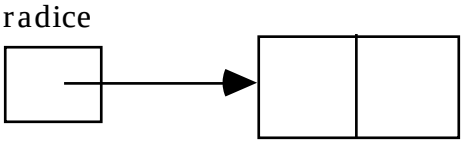
list radice;



radice= NULL;

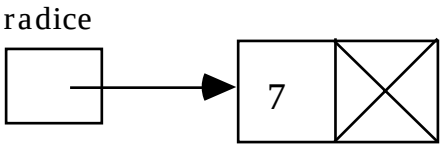
Inizializzazione lista vuota (***emptylist***).

radice = (list) malloc(sizeof(item));



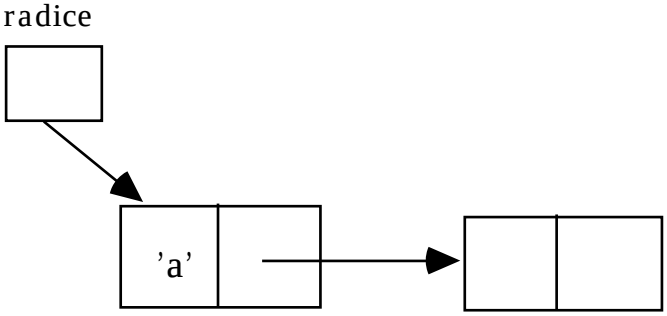
radice*	variabile puntata
radice->value	componente *value* della variabile puntata (testa della lista)
radice->next	puntatore all'elemento successivo (coda o resto della lista)

radice->value=7;
radice->next=NULL;



cons(7,radice)

radice->next=(list)malloc(sizeof(item));



Aggiunge un elemento in seconda posizione.

Realizzazione modulare di Liste

Suddivido la realizzazione del tipo di dato astratto lista nei "moduli" **list** e **el**, rappresentati dai seguenti file:

Modulo list:

realizza genericamente la lista (indipendentemente del tipo dell'elemento):

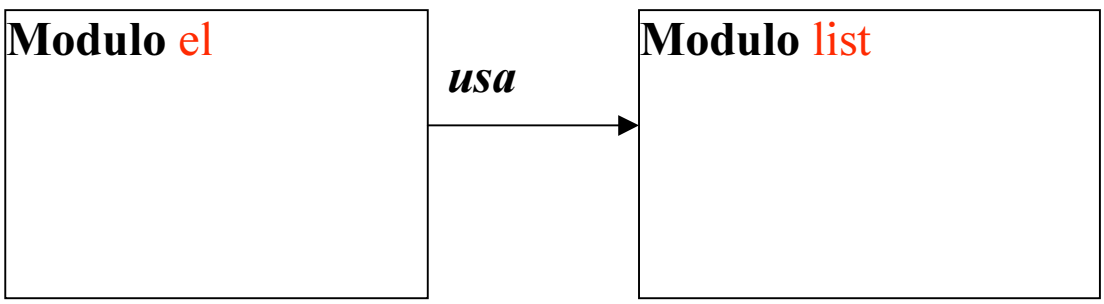
list.c realizzazione del tipo di dato astratto lista **list.h interfaccia del modulo**

Modulo el:

realizza l'elemento:

el.c funzioni di utilita` dipendenti dalla rappresentazione dell'elemento (definizione) **el.h interfaccia del modulo (rappresentazione dell'elemento)**

Relazioni:



Modulo list: interfaccia

File list.h:

```
/* LIST INTERFACE  - file list.h*/
#include "el.h"

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

typedef  item* list;

/*  PROTOTIPI DI FUNZIONE (extern) */
list emptylist();
int empty(list);
element_type head(list);
list tail(list);
list cons(element_type, list);
void showlist(list);
```

Modulo list: implementazione

File list.c:

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

list  emptylist()
{ return NULL; }

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

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

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

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

```
void showlist(list l)
{
while (!empty(l))
  {showel(l->value);
   l=tail(l);
  }
}
```

- Il modulo list necessita della dichiarazione del tipo **element_type** (che caratterizza il campo **value** dell'elemento della lista) e di alcune funzioni (ed es. showel) dipendenti dal tipo **element_type**. Queste entita` sono importate dal modulo el.

Modulo el

Interfaccia: File el.h (ad esempio, per liste di interi)

```
/* LIST ELEMENT TYPE - file el.h*/
typedef int element_type;


int isequal(element_type, element_type);
int isless(element_type, element_type);
void showel (element_type);
```

Implementazione: File el.c (ad esempio, per liste di interi)

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

int isequal(element_type e1, element_type
e2)
{
return  (e1==e2); }

int isless(element_type e1, element_type
e2)
{return (e1<e2);}

void showel (element_type e)
{ printf("%d\n",e);}
```

Esempio:

Un esempio di programma che utilizza questo modulo (file separato, ad esempio prova.c):

```
#include <stdio.h>
#include "list.h"
/* importa il tipo list e le operazioni*/
void main(void)
{list root;  /* dich. dato */
 int i;
  root=emptylist();
  do
   {printf("\n Introdurre valore:\t");
   scanf("%d", &i);
   root = cons(i, root);
    }
  while (i!=0);
  showlist(root);        /* stampa */
}
```

Altri esempi

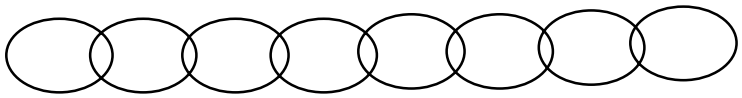
Le slides che seguono sono approfondimenti
ed argomento di esercitazione

Creating a Node

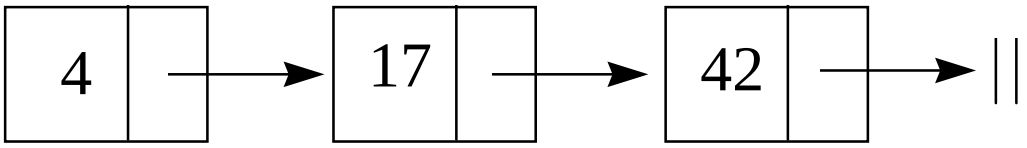
We can create records upon demand, as many as needed:

```
Pnew = (struct node *)
malloc(sizeof(struct node));
```

Linked Lists



With pointers, we can form a “chain” of data structures:




```
struct node {
     int data ;
     struct node *next };
```

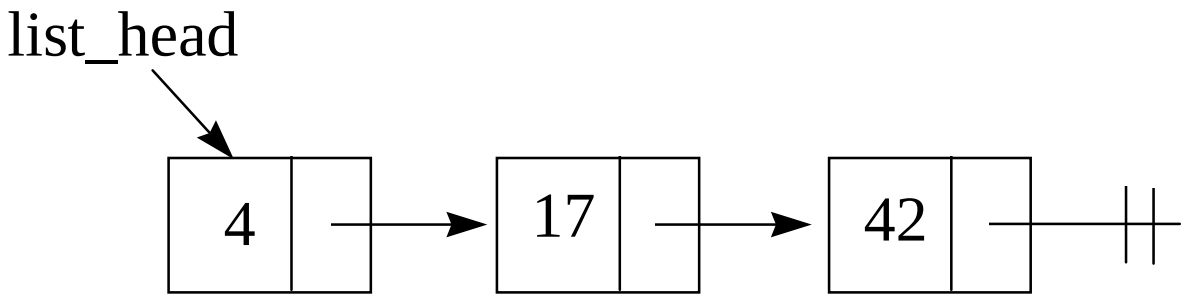
Creating a Linked List

```
struct node *list_head,*temp ;
list_head=(struct node *)

malloc(sizeof(struct node));
list_head->data = 2
list_head->next = NULL
temp = list_head

list_head = ...malloc ....
list_head->data = 4
list_head->next = temp
. . .
.....
```

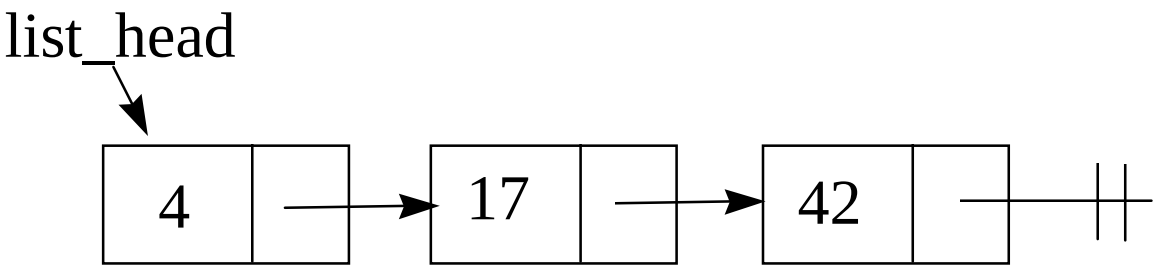
Traversing a Linked List

All of the nodes in a linked list
can be visited recursively:

```
void Print_List(struct node *node_ptr)
       {
/*prints out all data in list */
    if (node_ptr != NULL) {
        print(node_ptr->data)
        Print_List(node_ptr->next) }

       }/*Print_List
```

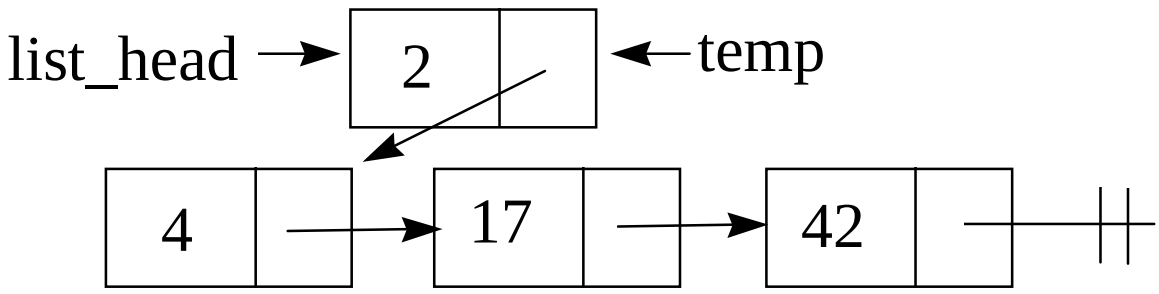


Inserting into a Linked List

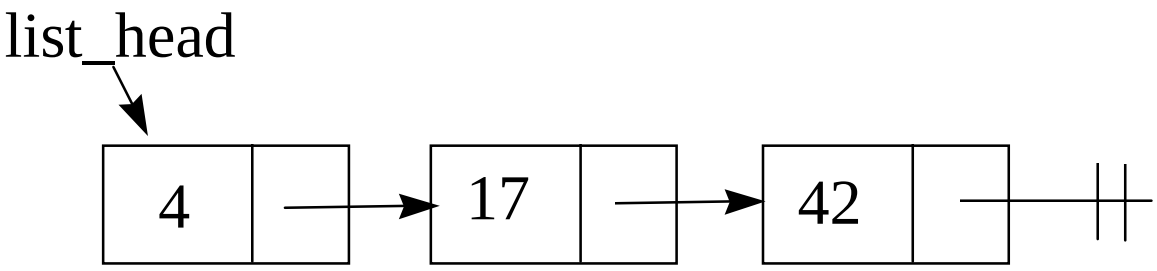


To add to the FRONT of the list:

```
struct node *list_head,*temp ;
  temp =(struct node *)
  malloc(sizeof(struct node));


    temp->data = 2
    temp->next = list_head
    list_head = temp
 ...
```



Inserting into a Linked List

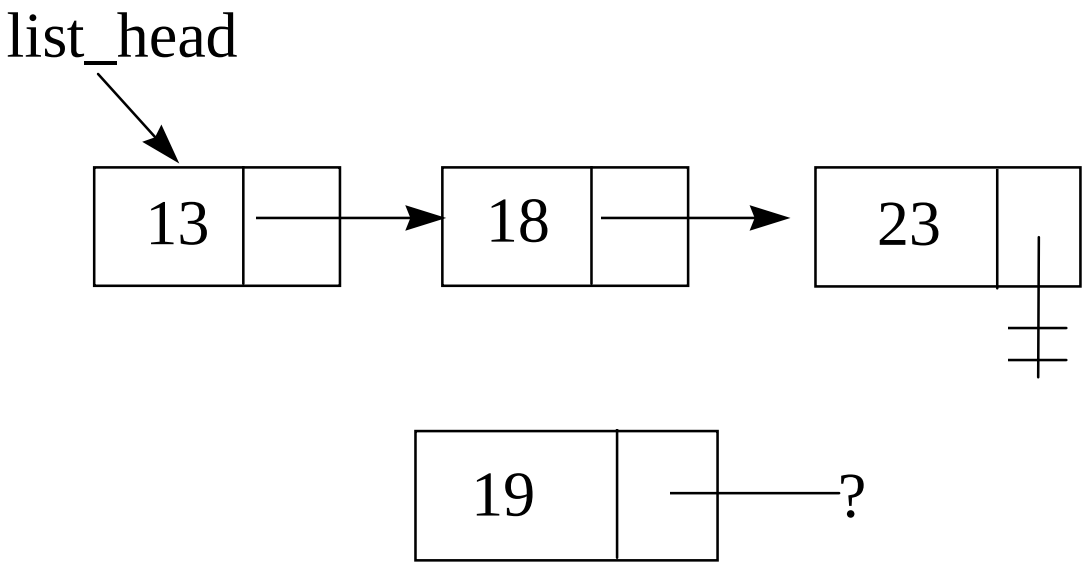


To add to the END of a linked list:

```
struct node * Add_To_End(
struct node *head,int new_data )

/* Add new node to end of list
/* Precond: head points to front of
/*    NULL-terminated list
/* Postcond: new list has one more
/*    element than old list
  { if( head == NULL )    {
      head = ...malloc ....;
      head->data = new_data;
      head->next = NULL;  }
   else
head=Add_To_End( head->next, new_data);
return head;   }
```

Inserting in the Middle of a Linked List

Linked lists can maintain a list of items in sorted order



Inserting in the Middle

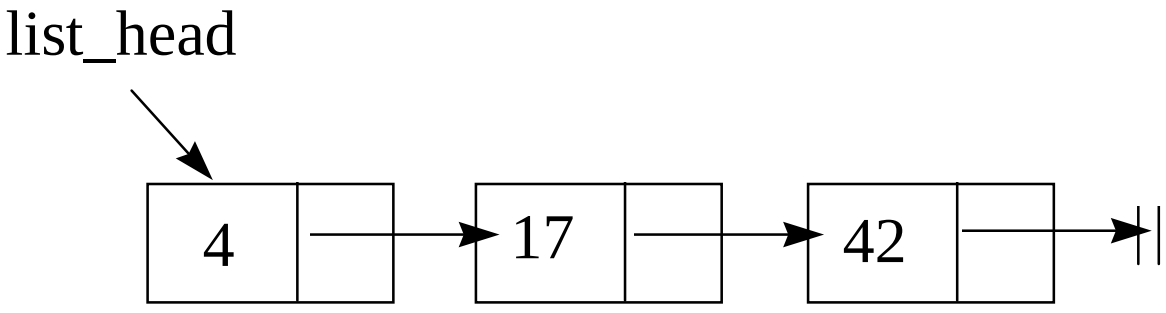
```
struct node * Insert_In_Order
   (struct node *head, int  new_data )
     {
 /* Inserts a Num into a list in order
 /* Precond: head points to a list in
 /*    increasing order
 /* Postcond: List contains one new
 /*    element in right spot
     struct node *temp;

   if ((head == NULL) ||
      (head->data > new_data)) {
     /*get new node, store data in it
     temp = ...malloc ....;
     temp->data = new_data;
     /*connect new node in the list
     temp->next =head
     head = temp}
   else
 head= Insert_In_Order(head->next,
                       new_data)
   return head}
/*end of Insert_In_Order
```

Deleting from a Linked List

To remove a Record from the list:

{ Remove the first element }
list_head = list_head->next

{ Remove the second element }
list_head->next = list_head->next->next



Scope of Linked data

The nodes of a linked structure are not limited in scope

```
void Do_Stuff(struct node *node_ptr)
    {
   /*Assign value to first element
   node_ptr->data = 42;
     } /*Do_Stuff
```

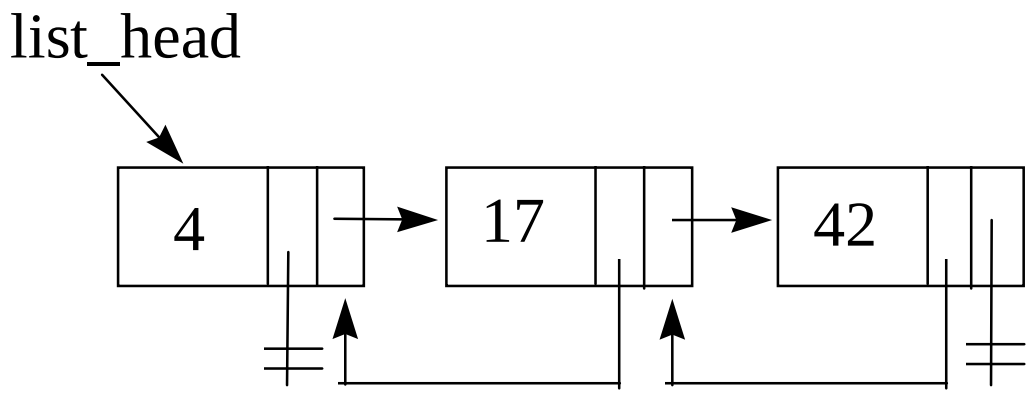
- The pointer is a *in* parameter, cannot change the original *pointer value*.... BUT...
- It is *used* to change the *state of a node*... and that change WILL last....
- Giving access to a list *means* giving ability to make changes to it; cannot protect

Altri Esempi di strutture semplici

Le slides che seguono sono approfondimenti utili per la preparazione

Doubly Linked Lists

Doubly linked lists allow us to traverse in either direction:



Doubly Linked Lists

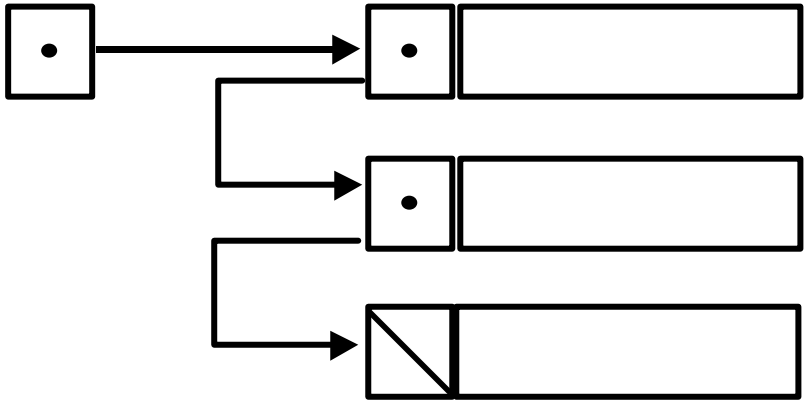
```
struct node {
   int data ;
   struct node *next ;
   struct node *prev ;
            }



// insert the new_node in the list
// AFTER current_ptr

new_node->next = current_ptr->next;
new_node->prev = current_ptr;
current_ptr->next = new_node;
new_node->next->prev = new_node;
```

Stacks

- Specialized linked list
- Nodes added/deleted only from top
- LIFO data structure (Last-in First-out)
- Nodes added via Push operation
- Nodes deleted via Pop operation
- Referenced via pointer to top of stack

Stacks

Stack - New Nodes added at Top



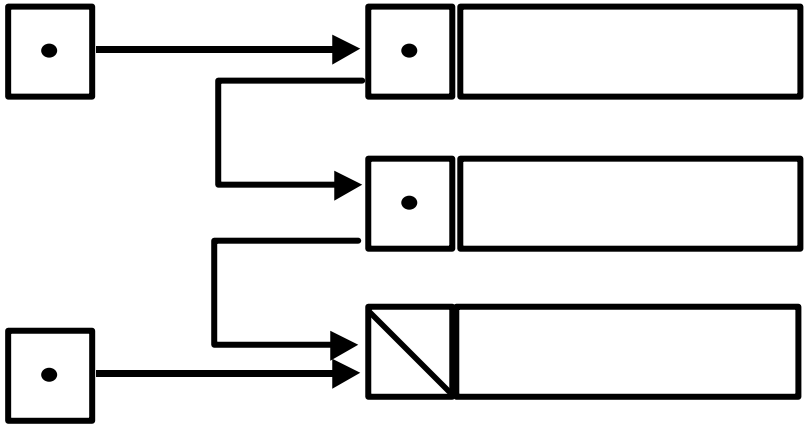
NULL pointer means end of list

Queues

- Specialized linked list
- Nodes added only at bottom, or *tail*
- Nodes removed only from top, or *head*
- FIFO data structure (First-in Last-out)
- Nodes added via Enqueue
- Nodes deleted via Dequeue
- Requires two pointers, one for each end

Queues

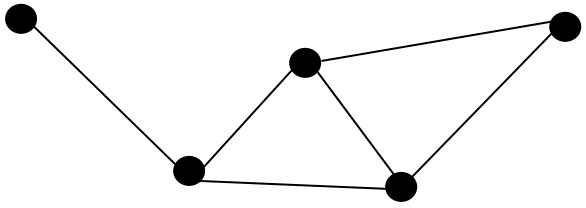
Queue - Nodes removed at Top

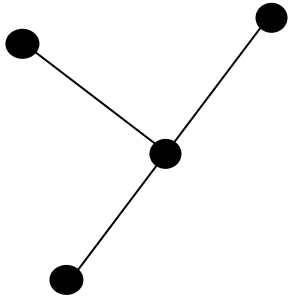


Nodes added at Bottom
NULL pointer means end of list; could also compare to Bottom pointer

Graphs

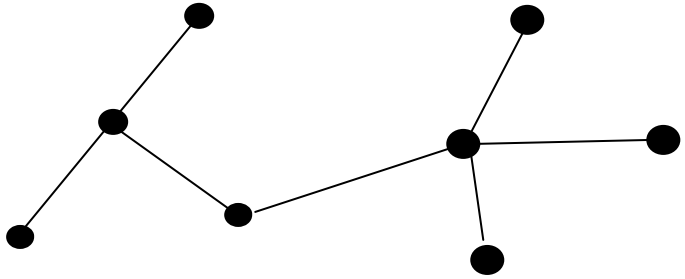
- Nonlinear data structure
- Set of nodes and edges, or paths
- Each edge associated with a pair of nodes
- 1 edge for each pair of nodes
- Edge is incident on the paths; nodes are adjacent
- Path is sum of edges from node A to node B
- Simple path has no repeated nodes
- Weighted graph, or network, assigns values to edges
- Length of path is sum of weights

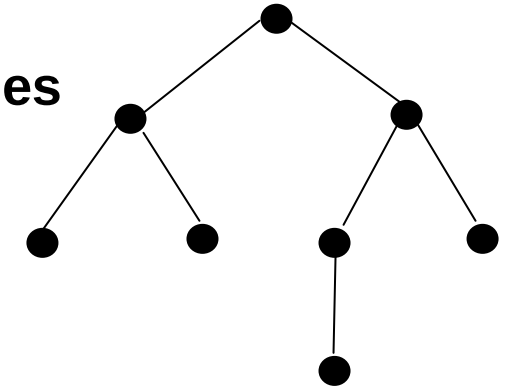




Graphs and Trees

- A graph is a tree if there is a unique, simple path from any one node to another
- A rooted tree is a tree with a particular node designated as the root; each lower node is a child
- a binary tree is a special kind of rooted tree
  - each node has at most two children; left and right
  - child nodes are siblings
  - a node without a child is a leaf node
- Binary search tree - ordered node values
  - left child < parent < right child
  - new node inserted only as leaf

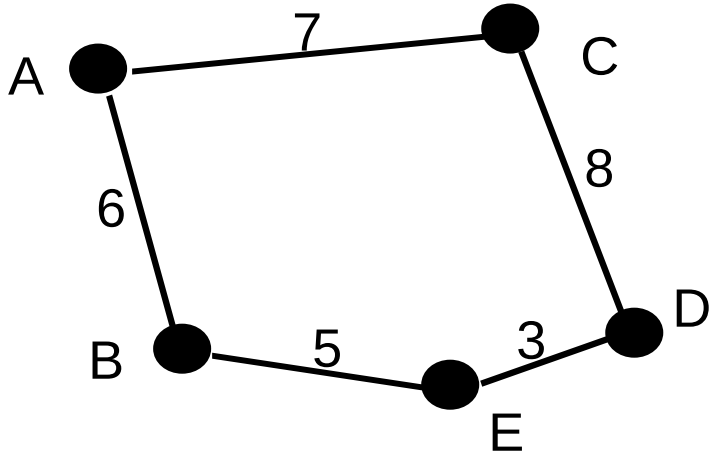


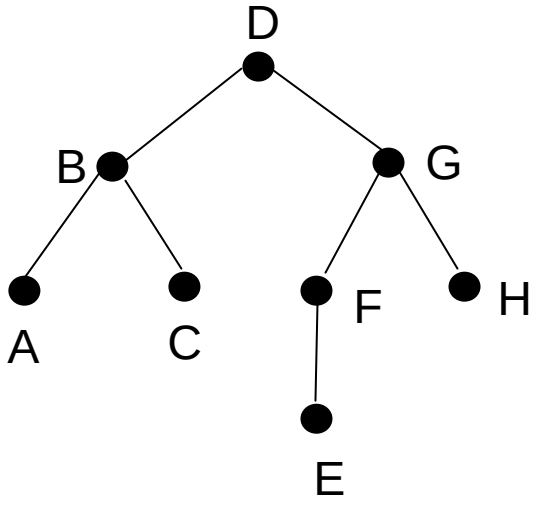


Binary Tree Traversal

- **Tree Traversal**
  - to visit each node in the tree
  - 3 algorithms: preorder, inorder, postorder
  - a.k.a. prefix, infix, postfix
- **Preorder (Prefix) - NLR**
  - visit NODE, left subtree, right subtree; all preorder
- **Inorder (Infix) - LNR**
  - visit Left subtree, NODE, Right subtree; all inorder
- **Postorder (Postfix) - LRN**
  - visit Left subtree, Right subtree, NODE; all postorder

Examples





Prefix (NLR):

Infix (LNR):

Postfix (LRN):

Inserimento e estrazione dalla pila (1)

```
struct EL      /* tipo dichiarato globalmente */
    {
      int   info;
      struct EL    *next;
    };



/* PUSH: inserimento in testa. Procedura:
riceve la pila e il dato da inserire*/



void push (struct EL ** stack, int dato)
{
 struct EL *p;

 p= (struct EL *)malloc(sizeof(struct EL));
 p->info = dato;
 p->next = (*stack);
 *stack  = p;
}
```

Inserimento e estrazione dalla pila (2)

```
/* POP: estrazione dalla testa. Funzione:
riceve la pila, restituisce come parametro il dato
estratto, restituisce come valore se l'operazione e'
andata a buon fine (-1 = stack vuoto)*/


int pop (struct EL ** stack, int *dato)
{
 struct EL *p;

 if (*stack == NULL)          /* pila vuota */
    return –1;

  else          /* pila non vuota */
    { *dato = (*stack)->info;
      p = *stack;
      *stack = (*stack)->next;
      free(p);
      return 0;
    }
}
```