POLITECNICO DI MILANO
Applicazione SchedaIncarico

Scheda Riassuntiva

Anno Accademico	2005/06
Facoltà	Facolta' di Ingegneria Industriale
Tipo Insegnamento	MONODISCIPLINARE
Codice Identificativo	060065
Denominazione Insegnamento	INFORMATICA C
Docente	MARTUCCI RENATO
CFU	5.0

Corsi di Studio cui l'insegnamento è offerto

Nome Corso di Laurea	Corso Unione	Indirizzo	DA	A
Ing.IV(1 liv.) - BV (100) INGEGNERIA AEROSPAZIALE	-	*	N	ZZZZ

- M9 parte b
  - Tipi Astratti di Dati (TAD)
    - Liste (statiche e dinamiche)
    - Un esempio completo

http://www.elet.polimi.it/upload/martucci/index.html

Rappresentazione dei Dati

- I componenti sono i dati semplici
  - `Char, int, long, float, double`
- Si strutturano dati omogenei (array) ed eterogenei (struct, union)
  - Array `[]` moltiplica per N lo spazio di un tipo di dato
  - `struct` aggrega in spazi adiacenti e con nome collettivo dati eterogenei
  - `union` aggrega in sovrapposizione con un nome collettivo dati eterogenei

Composizione dei dati

- Partendo da dati semplici, si può applicare più volte uno dei tre costruttori ([], struct, union)
- Per accedere al dato elementare si deve applicare la corretta sequenza di operatori di accesso ([] , . , ->)
  - Es. →

Esempio di composizione e accesso

```
typedef struct {
    char   intestatario[50];
    char   data_accensione[9];
    int    cod_filiale;
    double saldo;
    int tasso[10];
    double max_fido;
            } ContoCorrente ;

ContoCorrente FILIALE_1 [1000];

int Tasso_applicato , cliente# ,classe;

………….
Tasso_applicato = FILIALE_1[cliente#].tasso[classe];
```

Estensione dei dati

- Array si presta ad organizzare dati anche non sequenziali … all’interno di uno spazio fisso (liste puntate da indici)
- `malloc` e puntatori permettono
  - di allocare la memoria solo quando serve
  - creare strutture libere nella memoria centrale

Estensione (anche temporale)

- `file` è una struttura che estende la ampiezza di una struttura dati
  - Fuori dallo spazio della memoria centrale
  - Lungo un arco temporale indefinito

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*:

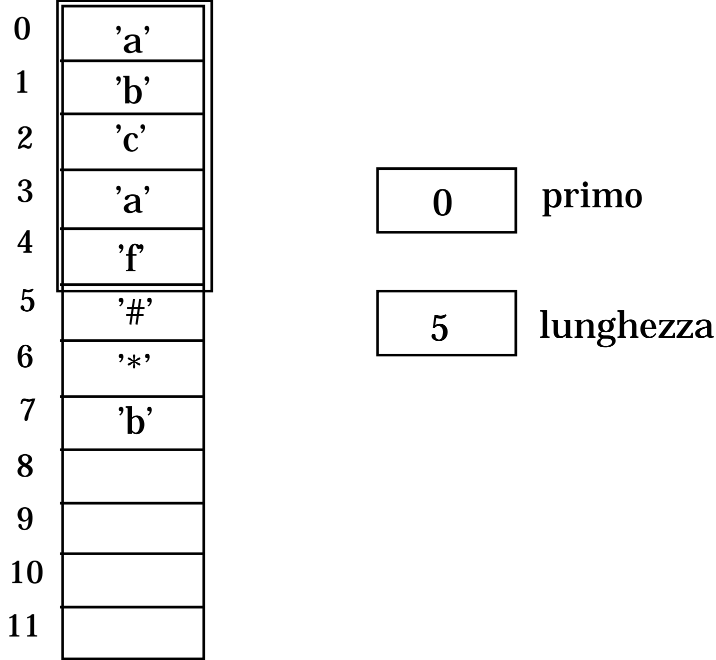
La piu` "banale" utilizza un **vettore mono-dimensionale** in cui sono inseriti gli elementi della lista in modo sequenziale (***rappresentazione sequenziale***).

- 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.

Tipi Astratti di Dati: operazioni

- Un TAD deve essere usato SOLO attraverso le operazioni definite
- Es.
  - Inserimento
  - Cancellazione
  - Appartenenza
  - Ricerca
  - Nullità
- La complessità e la fisicità del dato sono nascoste

Tavole

Spesso sono memorizzate su dispositivi di memoria di massa (***file***).

- In **C** si utilizzano ***file binari*** (lettura e scrittura di strutture).

Operazioni tipiche sulle tavole:

- **inserimento** di un elemento <Chiave, Attributi>

  inserisci: tavola × chiave × attributi → tavola

- **cancellazione** di un elemento (nota la chiave)

  cancella: tavola × chiave → tavola

- verifica di **appartenenza** di un elemento

  esiste: tavola × chiave → boolean

- **ricerca** di un elemento nella tavola

  ricerca: tavola × chiave → attributi

L'operazione di ***ricerca*** e\` la piu\` importante.

↣ Spesso la rappresentazione concreta viene scelta in modo da ottimizzare questa operazione.

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']

L: 2

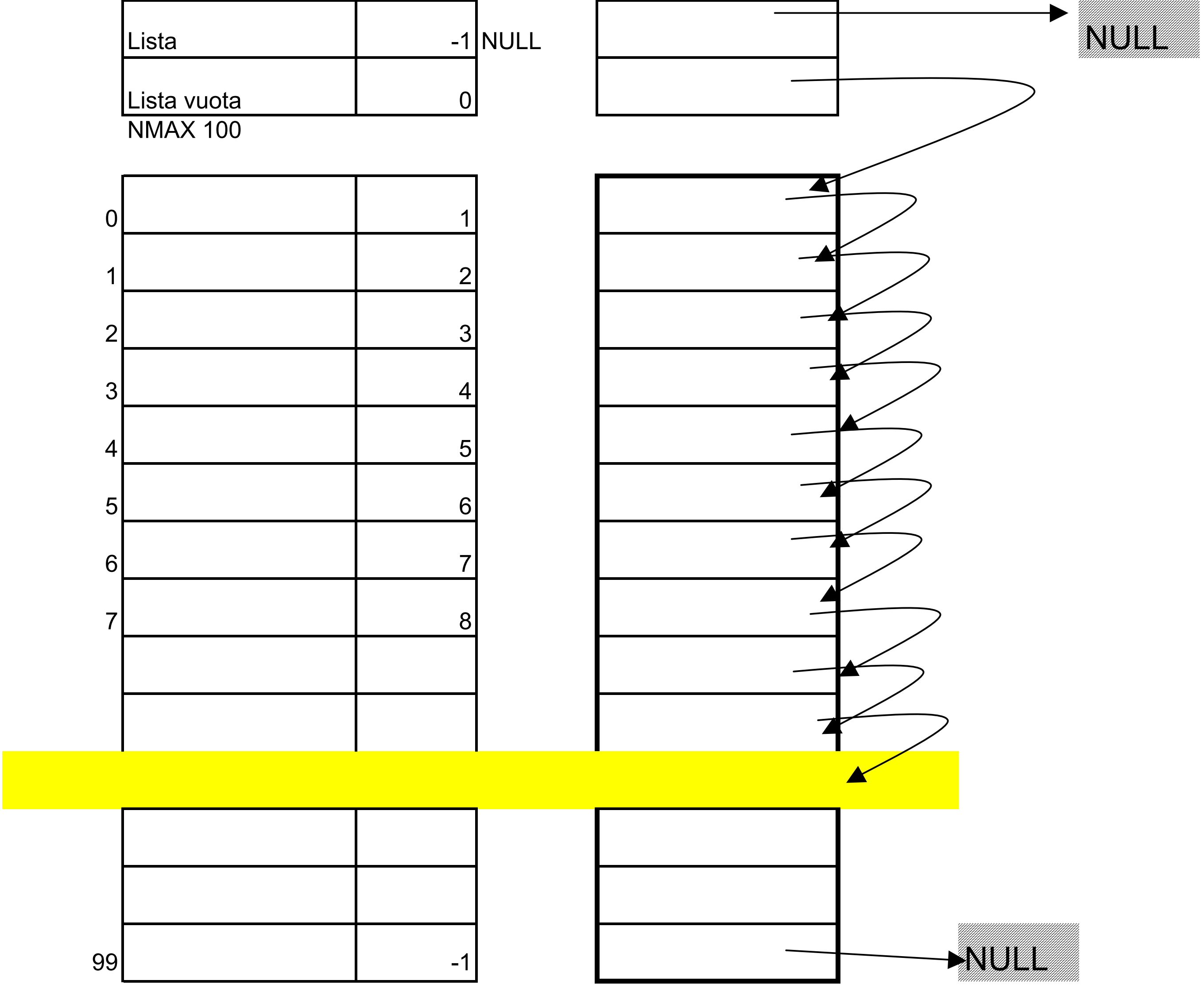
0	'd'	9
1		
2	'a'	5
3		
4		
5	'b'	6
6	'c'	0
7		
8		
9	'e'	-1

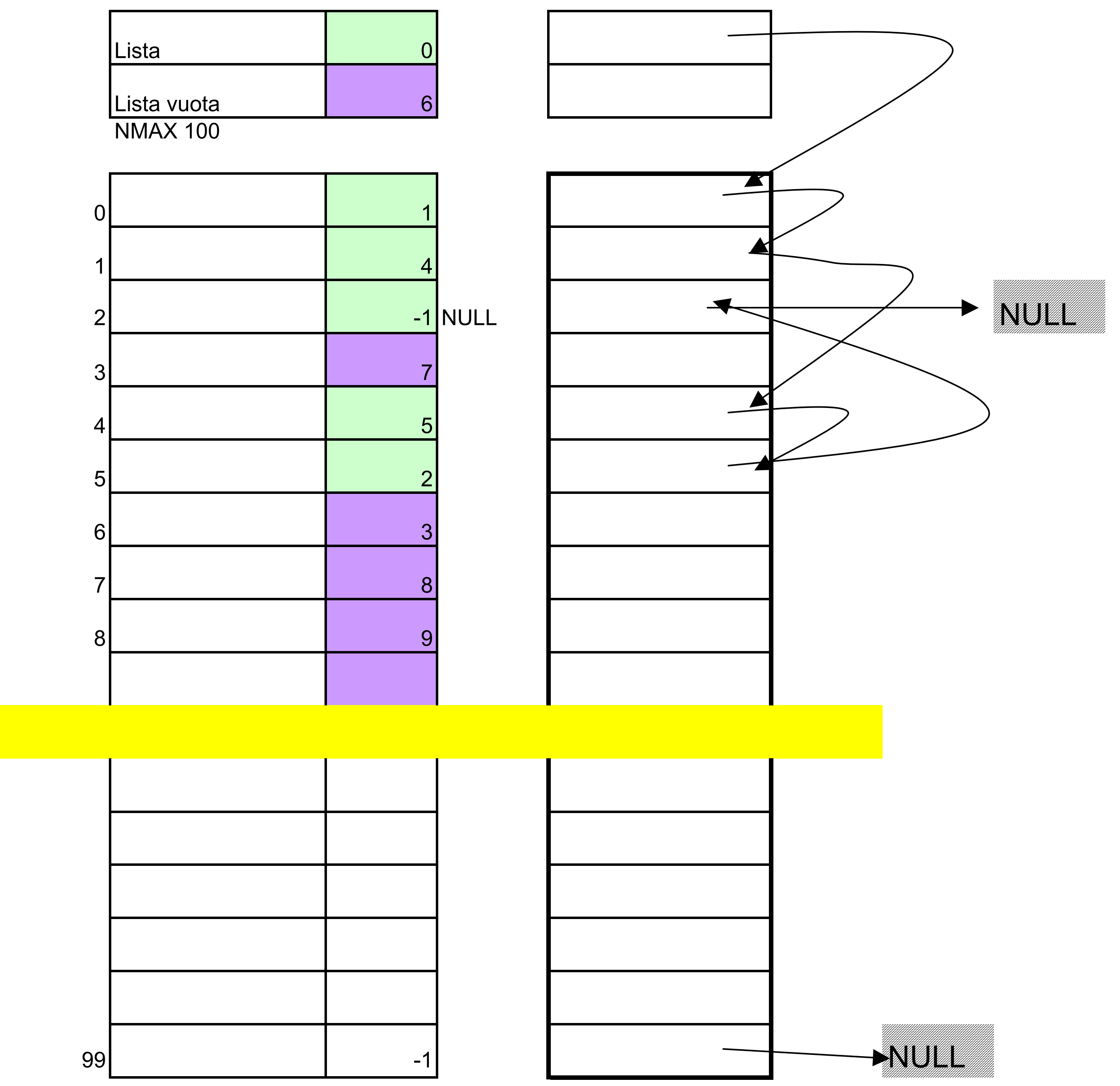
Problemi:

- Occupazione della memoria non ottimale.

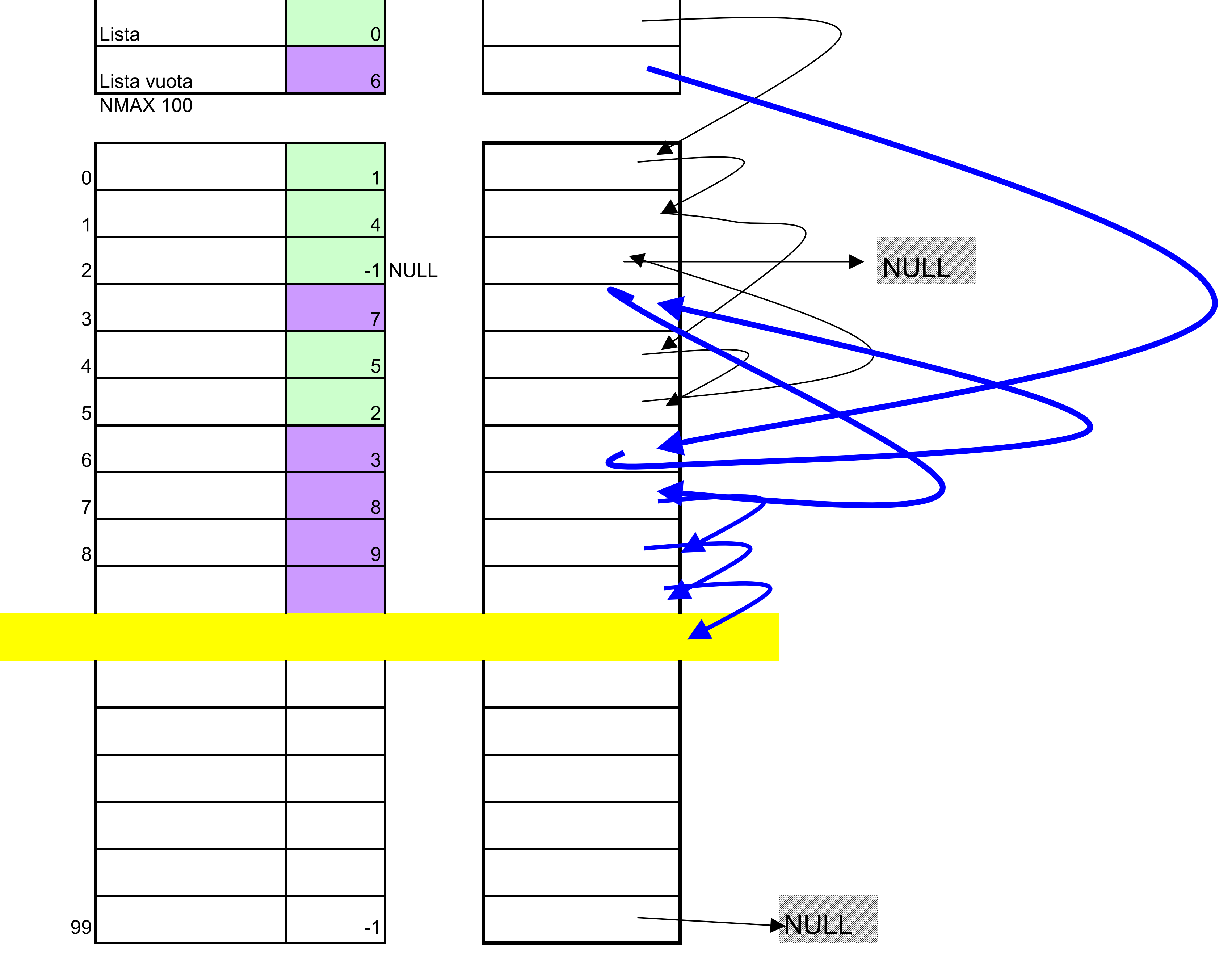
Caratteristiche della lista collegata su un vettore

- Occupazione di spazio “max”
- Necessità di mantenere due liste: una reale e una dei vuoti
- Cancellazione migliore, si deve aggiungere l’elemento ad una lista vuota
- Inserimento migliore, si inserisce l’elemento in un elemento della lista vuota
- Scansione totale per cercare


Lista
-1
NULL
Lista vuota
0
NMAX 100
NULL
0
1
1
2
2
3
3
4
4
5
5
6
6
7
7
8
99
-1
NULL



Lista
0
Lista vuota
6
NMAX 100
0
1
1
4
2
-1
NULL
3
7
4
5
5
2
6
3
7
8
8
9
99
-1
NULL
NULL



Lista
0
Lista vuota
6
NMAX 100
0
1
1
4
2
-1
NULL
3
7
4
5
5
2
6
3
7
8
8
9
99
-1
NULL
NULL

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:

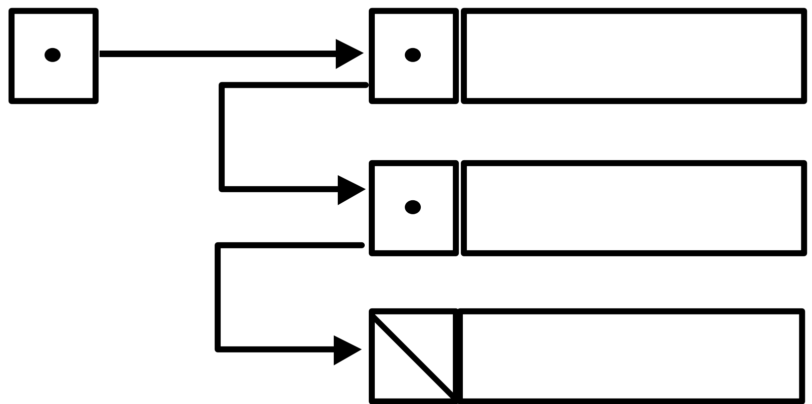
elementi della lista come **nodi** **(**allocati dinamicamente) e riferimenti come **archi**.

[5, 8, 21]

©M.L.Mick 1994

Linked Lists

Simple Linked List



NULL pointer means end of list

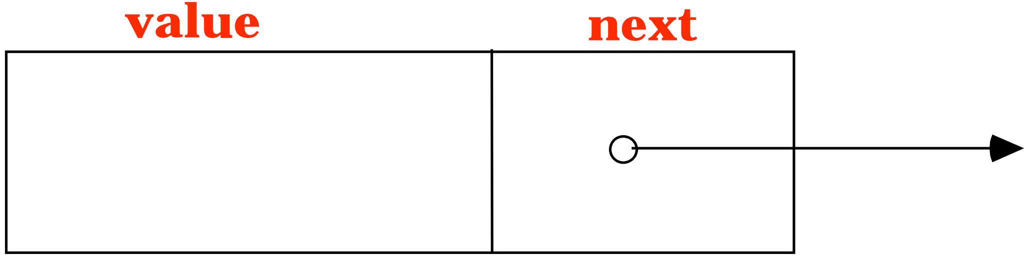
Rappresentazione collegata in memoria dinamica mediante puntatori

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:

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

typedef    item *list;

void main(void)
{list   root=NULL, L;    /*   2   liste    */
```

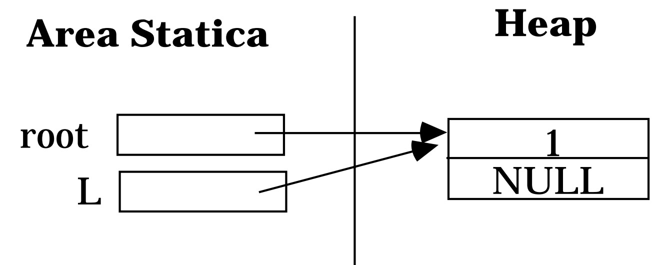
Area Statica

Heap

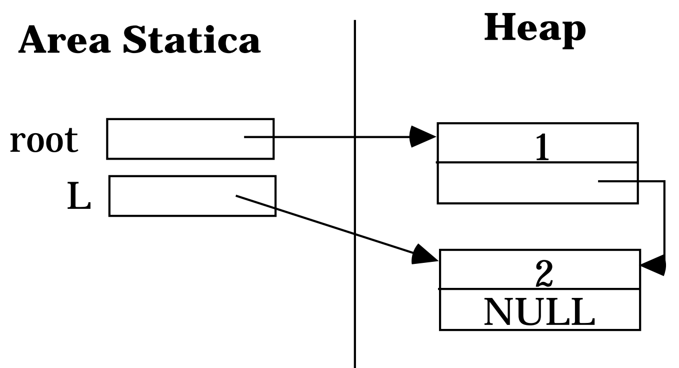
root NULL

L

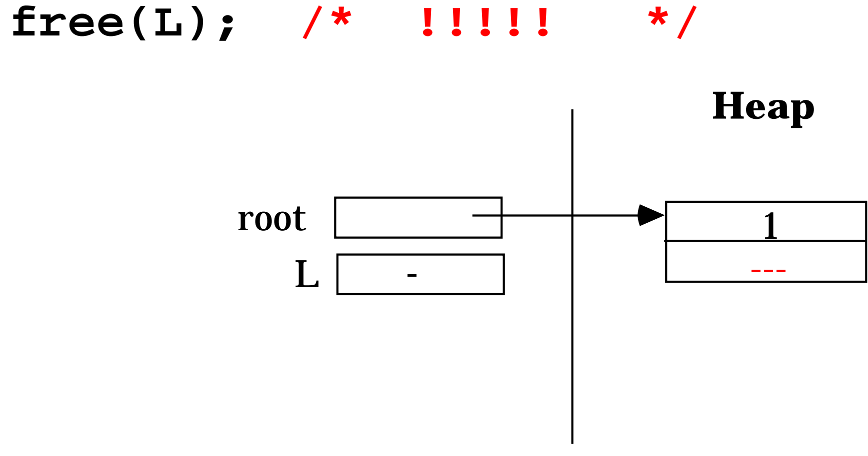
```c
L = (list) malloc(sizeof(item));
L->value = 1;
L->next = NULL;
root = L;
```



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




```
free(L);  /*  !!!!!    */
```




```
}
```

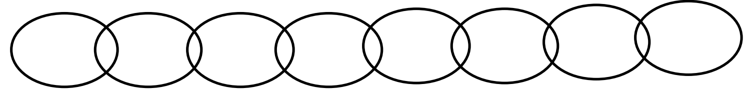
Altri esempi

Creating a Node

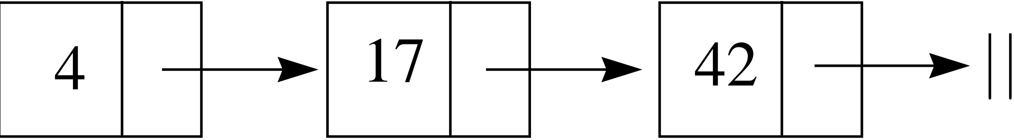
We can create records upon demand, as many as needed:

```
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 };
```

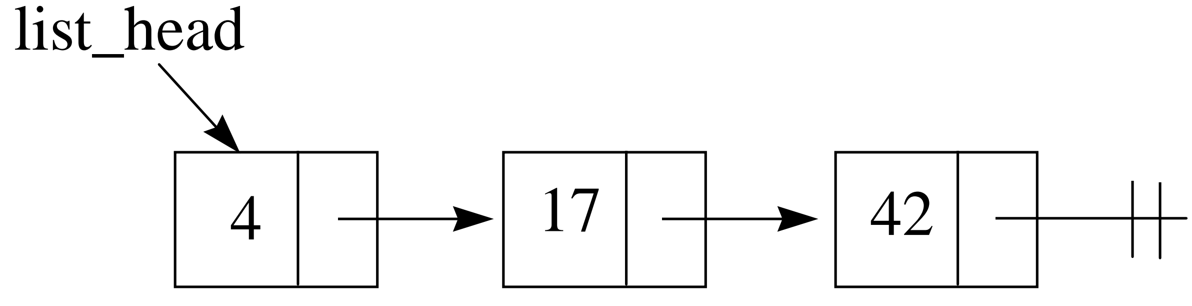
Creating a Linked List

```
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
 . . .
.....
```

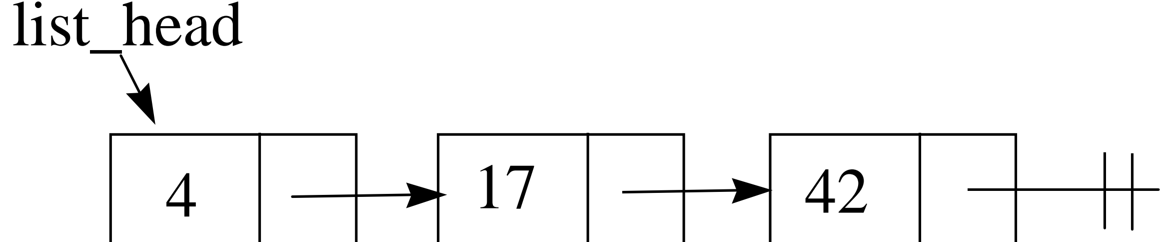
Traversing a Linked List

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
```

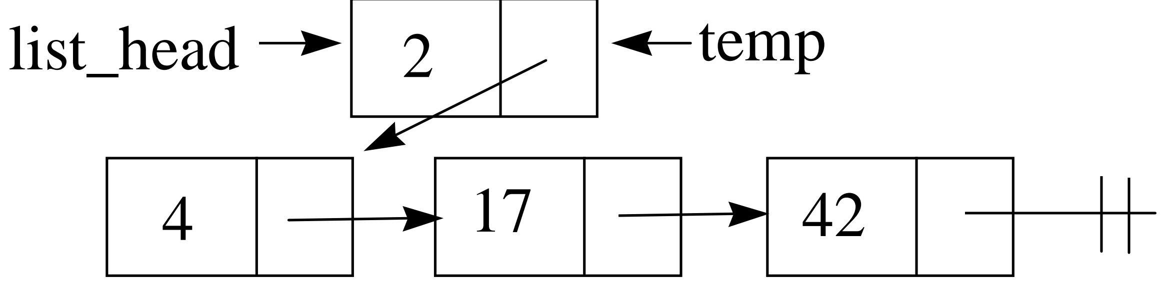


Inserting into a Linked List

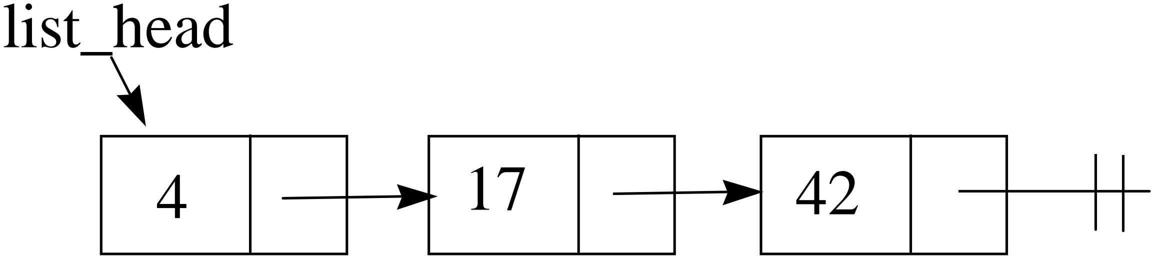


To add to the FRONT of the list:

```
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

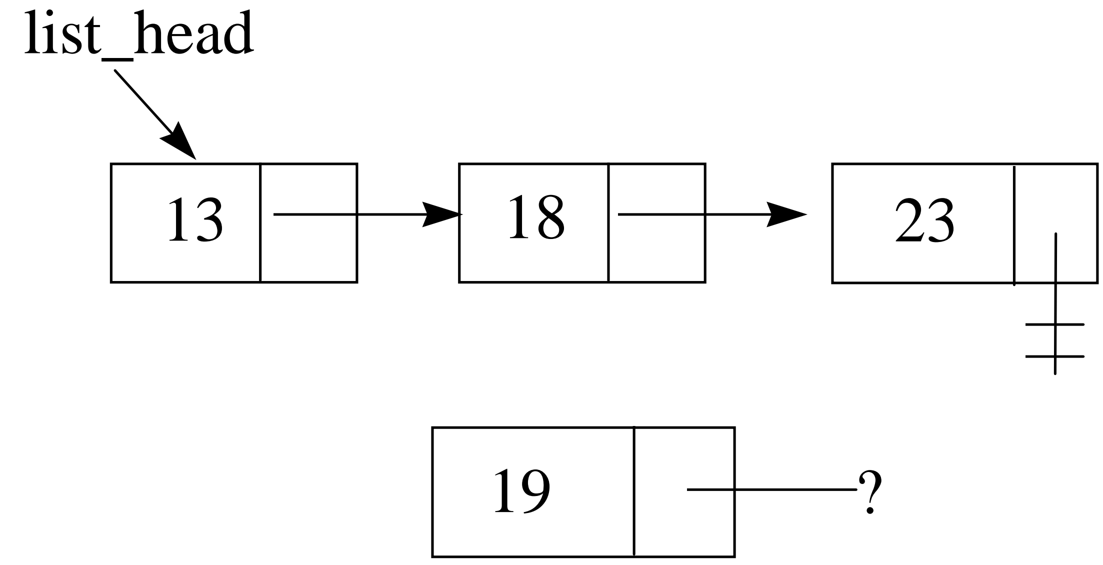


To add to the END of a linked list:

```
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
```

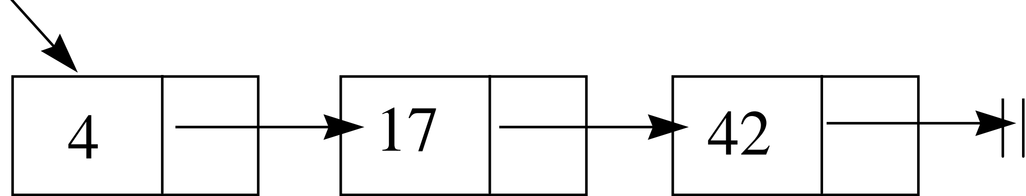
Deleting from a Linked List

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

list_head



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
```

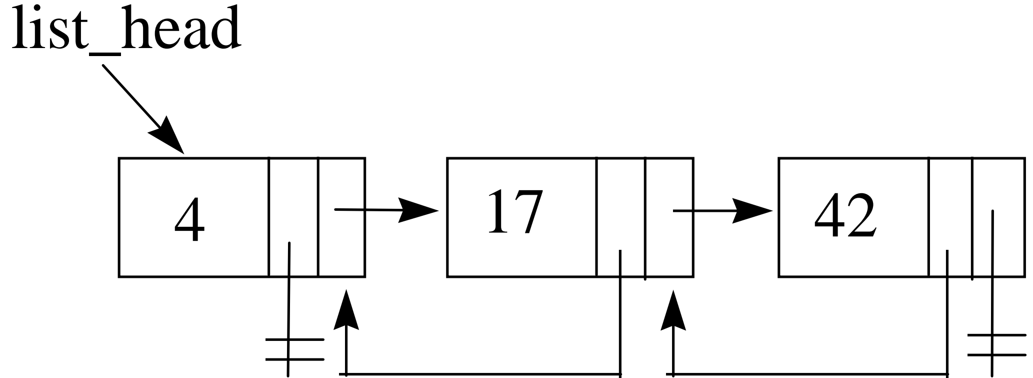
- The pointer is a *in* parameter, cannot change the original *pointer value*.... BUT...
- It is *used* to change the *state of a node*... and that change WILL last....
- Giving access to a list *means* giving ability to make changes to it; cannot protect

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;
```

Doubly Linked Lists

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

nodo

new_node

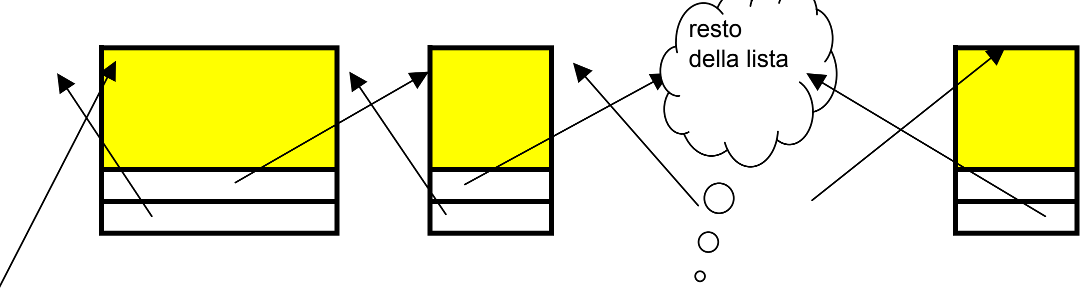
dato

next

prev

new_node = (…) malloc (….)

```
// 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;
```



current_node

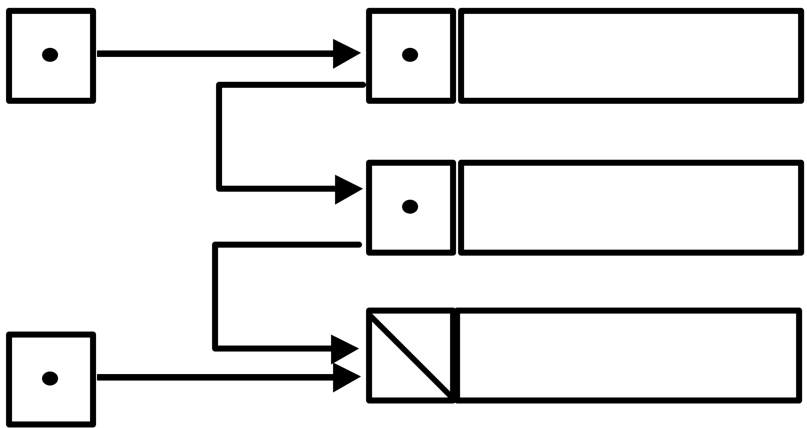
©M.L.Mick 1994

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 <u>Enqueue</u>
- Nodes deleted via <u>Dequeue</u>
- 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

©M.L.Mick 1994

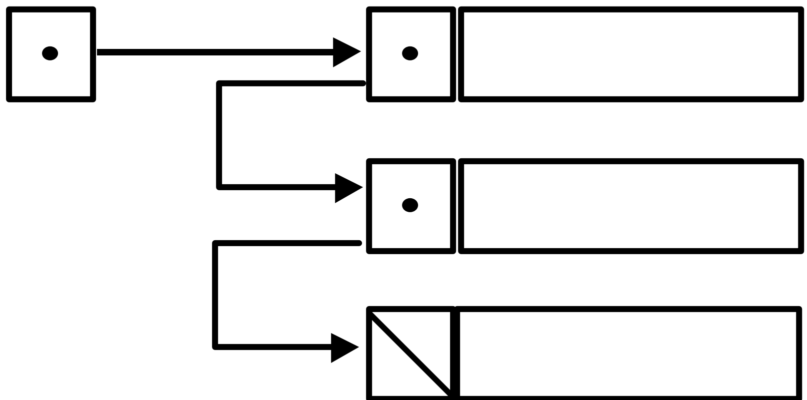
Stacks

- 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

©M.L.Mick 1994

Stacks

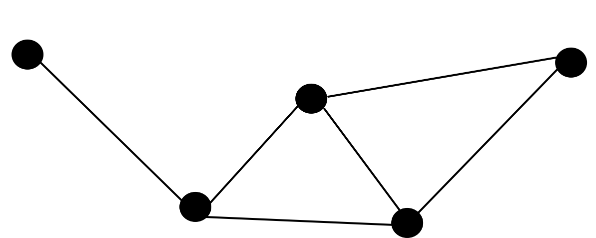
Stack - New Nodes added at Top

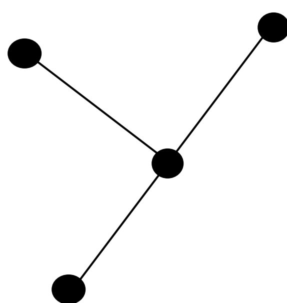


NULL pointer means end of list

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

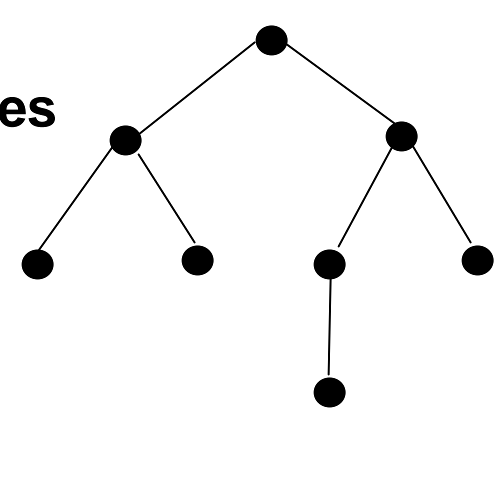


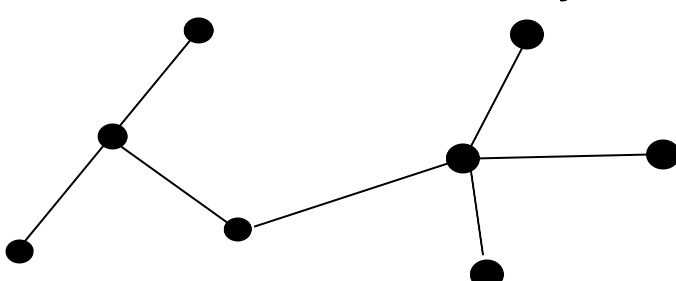


©M.L.Mick 1994

Graphs and Trees

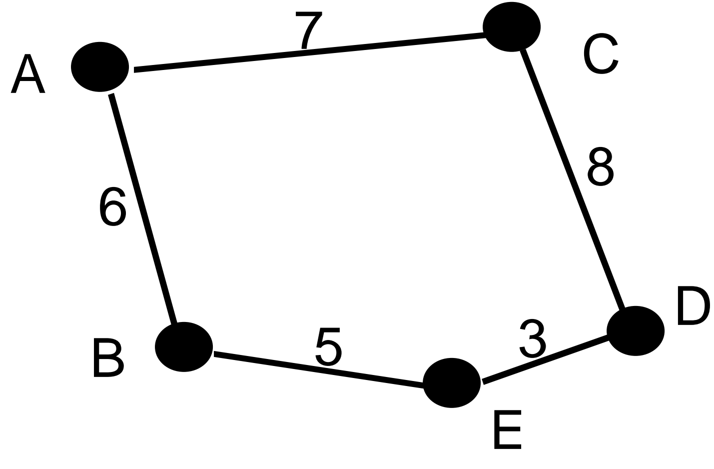
- A graph is a tree if there is a unique, simple path from any one node to another
- A rooted tree is a tree with a particular node designated as the root; each lower node is a child
- a binary tree is a special kind of rooted tree
  - each node has at most two children; left and right
  - child nodes 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

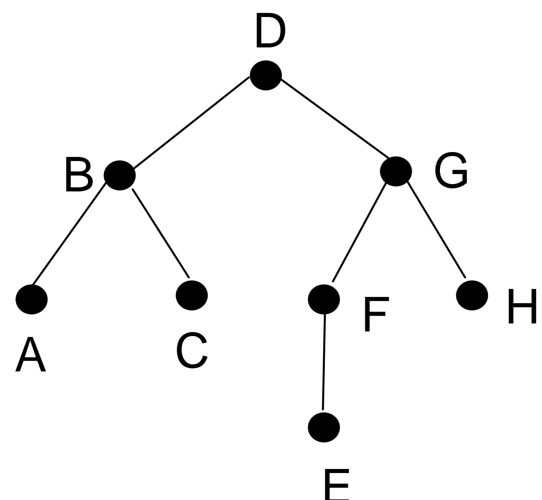




©M.L.Mick 1994

Examples





Prefix (NLR):

Infix (LNR):

Postfix (LRN):