

LIBRERIE

#include <stdio.h>/<stdlib.h>/<string.h> / <math.h>

usa anche x includere "file.h"

FORMATTAZIONE

%c	char	→ 1 byte	{	%hd	short int	→ 2 byte
%d	int	→ 4 byte		%ld	long int	→ 8 byte
%f	float	→ 4 byte		%lf	double	→ 8 byte
%X	int esa			%p	puntatore	
%u	unsigned int	→ 4 byte		%s	stringa	

IF if (condiz) { ... ; **WHILE** while (condiz) { **DO** do { ... ;
} else { ... ; } ... ; } } **while** (condiz);

FOR for (inizializ; condiz; incremento) { **equivale a:**
... ; } **inizializzazione;**
SWITCH **while** (condiz) {
switch (espressione) {
case 1: istruz; break;
...
case n: istruz; break;
default: istruz;
}

inizializzazione;
while (condiz) {
istruz; incremento; }

case 1...n può voler dire: case 1 ... 100
case 3,7,15 ...
case 'A','B','C' ...

FUNZIONI tiporisultato nomefunzione (parametri) {
istruz; }

CAST PTR void * p ;
int * ptr = p ; // cast automatico
int * ptr = (int *) p ; // cast esplicito

ARRAY tipoelementi a[n] - { n₀, n₁, ..., n_n } ;
// accesso con a[i], i ∈ [0, n-1]

STRINGHE char s[n] ; s[ultimo] = s[n-1] = '\0' ;
printf ("%s \n", s);
puts (s); **con** char * S = "stringa\0" ;
scanf ("%s", s); **con** char s[256] **inizializzata**
gets (s);

MALLOC `int * a = (int *) malloc (10 * sizeof (int));`
alloca n byte contigui restituendo void*

FREE `free(a); // con a puntatore`

CALLOC `char * ris = (char *) calloc ((len+1), sizeof (char));`

FILE operazioni tramite puntatore a record FILE

`fopen : FILE * filep = fopen ("path/nome.txt", "w+");`
modalita' : r lettura
w scrittura (sovrascrive o crea)
a accodamento (accoda o crea)
r+ lettura e scrittura
w+ scrittura (sovrascrive o crea)
a+ accodamento (accoda o crea)

in caso di insuccesso : NULL

`fclose : fclose (filep);`
in caso di successo : 0
in caso di insuccesso : EOF

`output : fprintf (filep, "...");`
`fputs ("...", filep);`

`lettura : fscanf (filep, "%s", s);`
alla fine del file restituisce EOF

`fgets (char * line, int n, FILE * file);`
cioè `fgets (line, sizeof(line), filep);`
con line : `char line [256];`
// legge fino al carattere di fine linea

`fgetc (filep);`
// legge carattere per carattere

SCHEMA DI CICLO DI LETTURA DA FILE

`FILE * file = fopen (...);`

`while (!fine-file) { // leggi prossimo blocco (char/parola/riga) }`

`fclose (file);`

fscanf

fgetc ← fgets ←

MATRICI

collezione di elementi omogenei su ARRAY BIDIMENSIONALI

Dichiarazione + Inizializzazione:

```
const int r = 3;
```

```
const int c = 3;
```

```
int m[r][c] = {{1,2,3},{4,5,6}...}; /oppure
```

```
int m[r][c], m[0][0] = 1; m[0][1] = 2; ...
```

Numero byte tot: sizeof(m);

Numero elementi di m: sizeof(m)/sizeof(tipodidato);

Numero colonne: sizeof(m[0])/sizeof(tipodidato);

Numero righe: Numero elementi / Numero colonne;

NB sizeof() non valida se m passata per ptr

Ciclo per righe: for (int i=0; i<numrighe; i++) {
 for (int j=0; j<numcol; j++) {
 istruzioni; }}

Ciclo per colonne: for (int j=0; j<numcol; j++) {
 for (int i=0; i<numrighe; i++) {
 istruzioni; }}

Allocazione Dinamica (array di array):

```
int righe=10; int colonne=5;
```

```
int **m = (int **)calloc(righe,sizeof(int*));
```

```
for (int i=0; i<righe; i++) {
```

```
    m[i] = (int *)calloc(colonne,sizeof(int));
```

Accesso: m[i][j] che equivale a *(*(m+i)+j)

NB *(*(m+i)+j) È LA SEMANTICA DI m[i][j]

ammesso che m sia definita come array di array

NB Quando passi m come parametro devi passare anche le sue dimensioni

ENUMERATI

```
ttypedef enum {
```

```
    ALTO, BASSO, DX, SX } Direzione;
```

```
Direzione d; d = BASSO;
```

```
if (d == ALTO) { ... }; if (d == BASSO) { ... };
```

```
if (d == DX) { ... }; if (d == SX) { ... };
```

→ puo' essere tutto! LUN, MAR, MER, GIO...

CARATTERI MINUSCOLI E MAIUSCOLI

```
char maiuscolo se : ( c >= 'A' && c <= 'Z' )
```

```
char minuscolo se : ( c >= 'a' && c <= 'z' )
```

```
char da MAIUSC a MINUSC : c - 'A' + 'a' / C = C + 32
```

```
char da minusc a MAIUSC : c - 'a' + 'A' / C = C - 32
```

RECORD

• DEFINIZIONE DI RECORD CON STRUCT

```
struct { char * nome; char * cognome;  
short int eta; } persona1, persona2;
```

• DEFINIZIONE DI TAG DI STRUTTURA

```
struct persona { char * nome, * cognome;  
short int eta; };
```

```
struct persona persona1, persona2;
```

```
struct persona persona3;
```

• DEFINIZIONE DI RECORD CON TUPedef

```
tupedef struct { char * nome; char * cognome;  
short int eta; } Persona;
```

• DEFINIZIONE DI RECORD CON TUPedef + TAG

```
struct persona { char * nome, * cognome;  
short int eta; };
```

tupedef struct persona Persona; ↳ definisce Persona
tramite persona

ACCESSO E INIZIAZIONE

① Persona p1;

```
p1.nome = "Mario"; p1.cognome = "Rossi";
```

```
p1.eta = 35;
```

② Persona p1 = {"Mario", "Rossi", 35};

③ Persona p1 = {

```
.cognome = "Rossi";
```

```
.eta = 35;
```

```
.nome = "Mario";
```

```
};
```

- ACCESSO TRAMITE PUNTATORI

Persona p = {"Mario", "Rossi", 35};

Persona * punt = & p;

printf ("%s \n", (*punt). nome);

printf ("%s \n", punt -> nome);

- ALLOCAZIONE DINAMICA

Persona * p1 = (Persona *) malloc (sizeof(Persona));

- ARRAY DI RECORD

Persona famiglia [3];

famiglia[0]. nome = "Mario";

famiglia[1]. nome = "Giulia";

SCL

```
typedef tipo TipoInfoScl;  
struct ELEMscl {  
    TipoInfoScl info;  
    struct ELEMscl * next;  
};
```

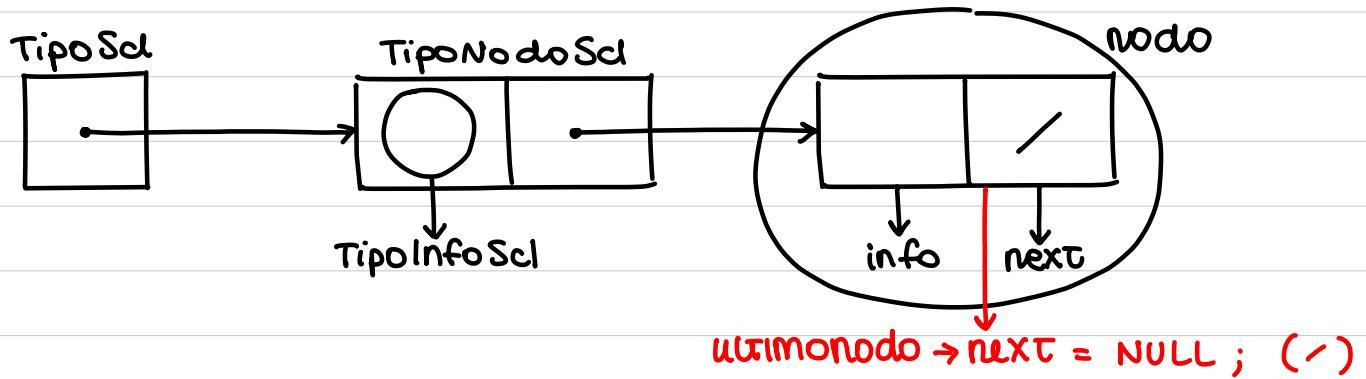
```
typedef struct ELEMscl TipoNodoScl;
```

```
typedef TipoNodoScl * TipoScl;
```

CON: `TipoInfoScl` : tipo di dati contenuti in info

`TipoNodoScl`: nome del tipo del nodo
(record info + next)

`TipoScl`: puntatore alla struttura del nodo



- SCL vuota : `tipoScl scl = NULL;`
- creazione di un nodo :
`tipoScl scl = (TipoNodoScl*) malloc(sizeof(TipoNodoScl));`
`scl -> info = e1 ; scl -> next = NULL;`
- creazione collegamento nodo :
`tipoScl Temp = (TipoNodoScl*) malloc(sizeof(TipoNodoScl));`
`Temp -> info = e2 ;`
`Temp -> next = NULL ;`
`scl -> next = Temp ;`

il collegamento è sempre del tipo `p -> next = q ;`

• inserimento nodo in prima posizione:

```
void addScl (TipoScl *scl, Tipoinfosc1 e) {  
    TipoScl temp = *scl;  
    *scl = (TiponodoScl *) malloc (sizeof(TiponodoScl));  
    (*scl) -> info = e;  
    (*scl) -> next = temp;  
}
```

• eliminazione nodo in prima posizione:

```
void delScl (TipoScl *scl) {  
    TipoScl temp = *scl;  
    *scl = (*scl) -> next;  
    free (temp);  
}
```

FUNZIONI SU SCL CHE MODIFICANO CONTENUTO

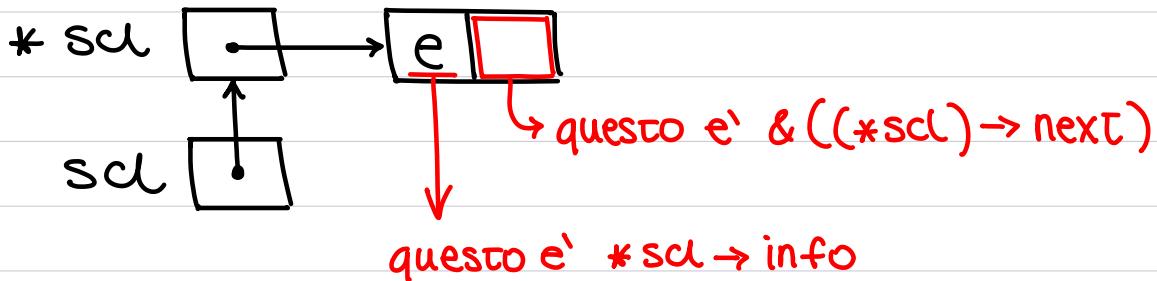
HANNO UN PARAMETRO DI TIPO TipoScl

```
if (!emptyScl(scl)) { ...; } // se non vuota  
scl → next; // prossimo nodo
```

CHE MODIFICANO STRUTTURA

HANNO UN PARAMETRO DI TIPO TipoScl *

APPROFONDIMENTO PUNTATORI A SCL



TIPO ASTRATTO (LISTE - CODE - PILE)

le SCL servono per implementare questi tipi di DATO ASTRATTO costituiti da:

- ① DOMINIO DI INTERESSE : contiene i valori ammissibili del tipo
- ② COSTANTI : valori del dominio utili per la rappresentazione degli elementi
- ③ FUNZIONI : che operano sul dominio e producono risultati sugli elementi

con i tipi astratti si possono definire tutti i tipi di dato

↳ LINEARI : LISTE , PILE , CODE , INSIEMI

NON LINEARI : ALBERI BINARI , N-ARI , GRAFI

DEFINIZIONE SEMI-FORMALE

Tipo Astratto T

Domini : D_1 : descrizione
...
 D_m : descrizione

Costanti : C_1 : descrizione
...
 C_k : descrizione

Funzioni : F_1 : descrizione con pre e post condizioni
...
 F_n : descrizione con pre e post condizioni

Fine Tipo Astratto

IMPLEMENTAZIONE

- ① Domini : usando tipi concreti del linguaggio
- ② Costanti : usando costrutti del linguaggio
- ③ Operazioni : usando funzioni del linguaggio

SCHEMI REALIZZATIVI

- con side-effect se modificano i dati che vengono passati come parametri della funz.
- in modalità funzionale se si crea un nuovo dato da restituire come output
- senza condivisione di memoria se le funzioni assicurano che non ci siano interferenze
- con condivisione di memoria

SCHEMI PIÙ INTERESSANTI

- CON SIDE EFFECT & SENZA CONDIVISIONE
(per realizzare dati mutabili)
- SCHEMA FUNZIONALE & CON CONDIVISIONE
(per realizzare oggetti immutabili)

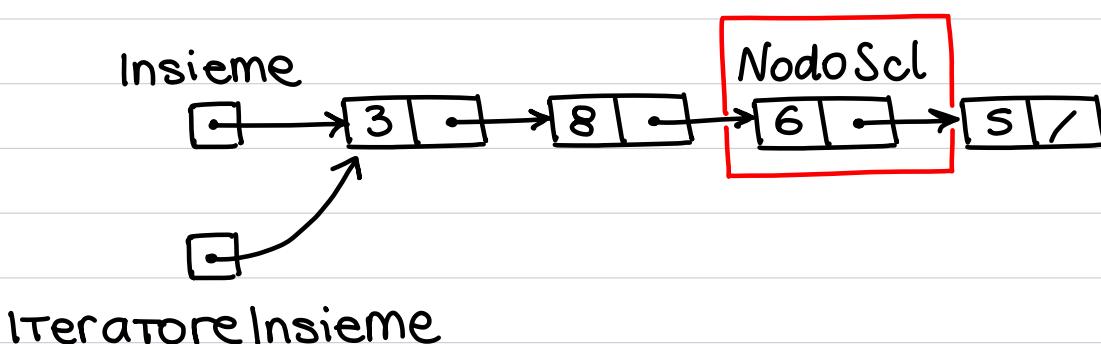
TIPO INSIEME

```
typedef int T;  
#define TERRORVALUE -9999999 ; → tipo T per l'iteratore  
struct Nodoscl {  
    T info;  
    struct Nodoscl * next;  
};  
typedef Nodoscl * Insieme; → L'insieme è un puntatore  
typedef struct {  
    Nodoscl * ptr  
} IteratoreInsieme;
```

valore di errore del

tipo T per l'iteratore

→ Insieme → info →



ANALISI DEI COSTI DI UN PROGRAMMA

- si analizza il caso PEGGIORE
- si analizza il comportamento ASINTOTICO
- n è la DIMENSIONE dell' input
- Definizione O-grande:

una funzione $f(n)$ è $O(g(n))$ (cioè O-grande di g)

se e solo se esistono 2 costanti positive c ed n_0 tali che:

$$|f(n)| \leq c g(n)$$

per tutti i valori di $n \geq n_0$

FUNZIONI DI COSTO

$O(1)$	COSTO COSTANTE
$O(\log n)$	COSTO LOGARITMICO
$O(n)$	COSTO LINEARE
$O(n \log n)$	COSTO QUASI-LINEARE
$O(n^2)$	COSTO QUADRATICO
$O(n^k)$	COSTO POLINOMIALE (k COSTANTE)
$O(k^n)$	COSTO ESPOENZIALE (k COSTANTE > 1)

OPERAZIONI DOMINANTI

sia A un programma di costo $O(f_A(n))$. Una sua operazione si dice DOMINANTE se, nel caso peggiore, viene ripetuta un numero di volte $g(n) = O(f_A(n))$.

Se un programma A ha un'operazione dominante che viene eseguita un numero di volte $O(g(n))$, allora il COSTO del programma è $O(g(n))$.

le operazioni dominanti sono solitamente nei cicli più interni, o sono quelle attivate nelle chiamate ricorsive.

COSTO DI ALGORITMI NOTI ALGORITMI DI RICERCA

- sequenziale : $O(n)$
- binaria : $O(\log(n))$

ALGORITMI DI ORDINAMENTO

- per selezione (Selection Sort) : $O(n^2)$
- a bolle (Bubble Sort) : $O(n^2)$
- per fusione (Merge Sort) : $O(n \log(n))$
- veloce (Quick Sort) : $O(n^2)$

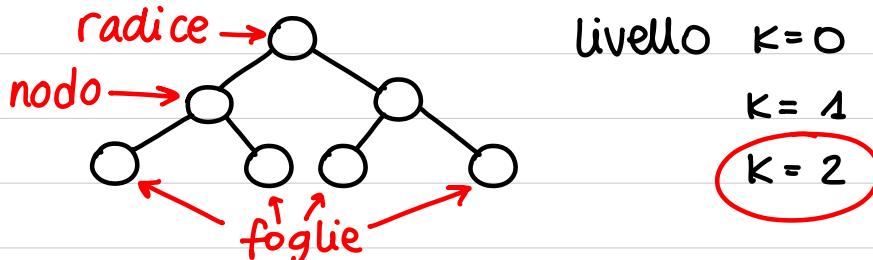
ALBERI

ALBERO BINARIO

- ① l'albero vuoto è un albero binario
- ② se Bd e Bs sono alberi binari, allora l'albero che ha un nodo radice e Bd e Bs come sottoalberi è binario
- ③ nient'altro è un albero binario

ALBERO BINARIO COMPLETO

un albero binario è completo sse ogni nodo che non sia una foglia ha 2 figli e se le foglie sono tutte allo stesso livello



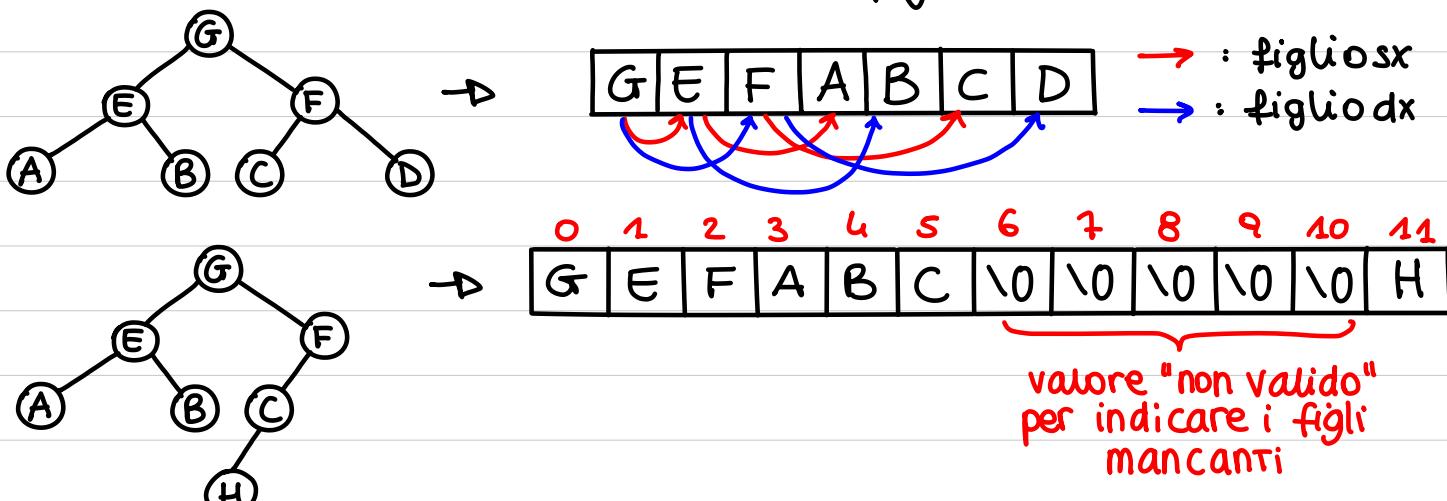
NODI

un albero binario completo non vuoto di profondità K ha $2^{k+1} - 1$ nodi

RAPPRESENTAZIONE CON ARRAY

Radice : posizione 0

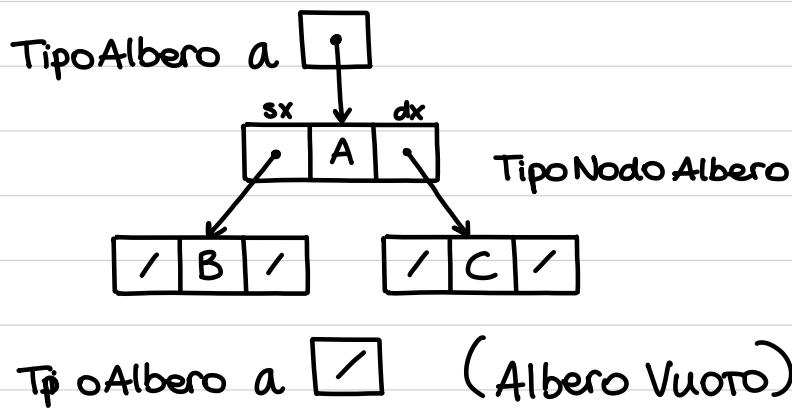
Dato un nodo in posizione i : figlio sinistro : $2i + 1$
figlio destro : $2i + 2$



Albero di n elementi \Rightarrow Dimensione della sua rappresentazione con array : $O(2^n)$

RAPPRESENTAZIONE CON SCNL

```
STRUCT Struct Albero {  
    TipolInfoAlbero info ;  
    STRUCT Struct Albero * sinistro , * destro ;  
};  
typedef STRUCT Struct Albero Tipo Nodo Albero ;  
typedef Tipo Nodo Albero * TipoAlbero ;
```

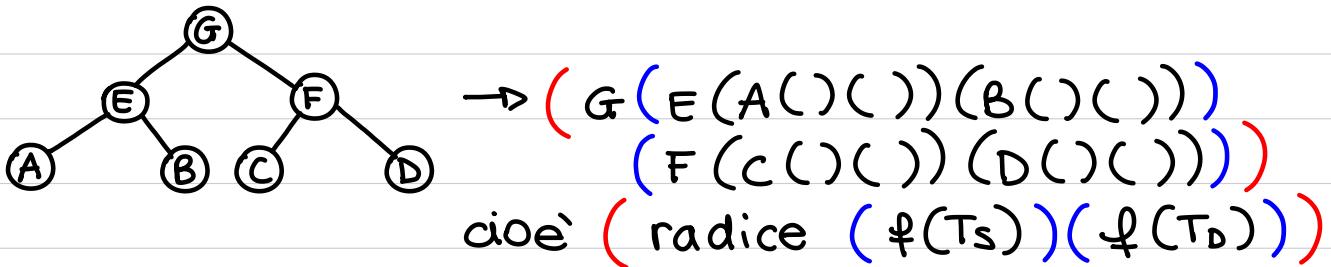


RAPPRESENTAZIONE PARENTETICA

Albero binario $T \rightarrow$ stringa $f(T)$;

Albero vuoto $\rightarrow f(\text{albero vuoto}) = ()$;

Albero binario con radice x e sottoalberi T_s e $T_d \rightarrow f(T) = (x + f(T_s) + f(T_d))$;



NB sia la rappresentazione con SCNL che quella parentetica si basano su definizioni INDUTTIVE!

VISITE DEGLI ALBERI

IN PROFONDITÀ: l'ordine di visita dipende dai collegamenti padre-figlio (cioè 2 nodi visitati consecutivamente hanno sempre questa relazione)

E' impossibile implementare questa visita con al dalo w file di costo lineare $O(1)$

E' necessaria una struttura dati aggiuntiva: la PILA o STACK
(LIFO: Last In First Out)

più efficiente se usata con RICORSIONE

IN AMPIEZZA: l'ordine di visita dipende dai livelli (cioè visita tutti i nodi di un livello, poi passa al livello successivo)

E' necessaria una struttura dati aggiuntiva: la CODA
(FIFO: First In First Out)

più efficiente se usata con ITERAZIONE

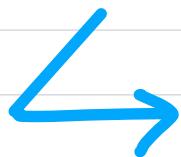
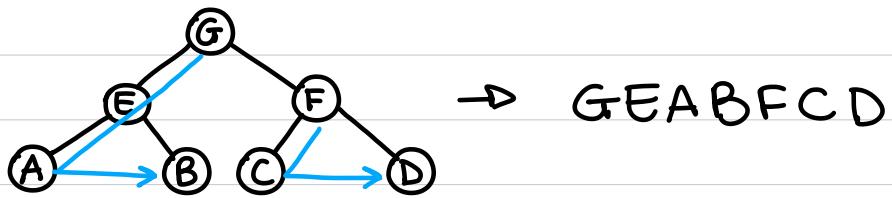
TIPOLOGIE DI VISITE IN PROFONDITÀ

IN PREORDINE: radice, sotto alb. sx, sotto alb. dx

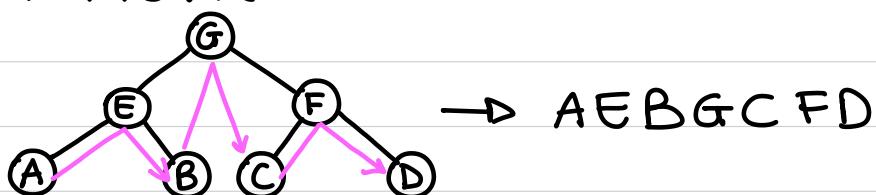
SIMMETRICA: sotto alb. sx, radice, sotto alb. dx

IN POSTORDINE: sotto alb. sx, sotto alb. dx, radice

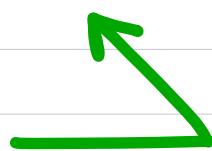
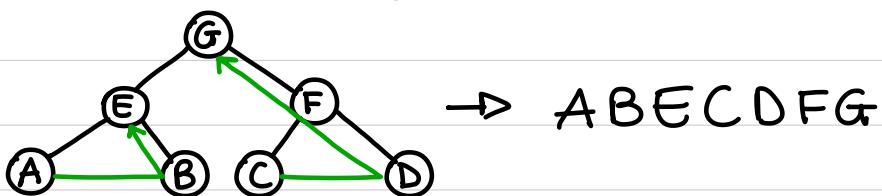
IN PREORDINE :



SIMMETRICA :

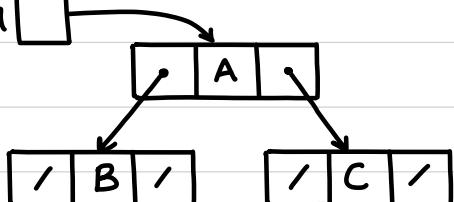


IN POSTORDINE :



NB

Dato a



* a è l'albero

(* a) → sinistro è info

(* a) → destro è info

&(* a) → sinistro è ptr

&(* a) → destro è ptr

&((* a) → sinistro)

uso * accedere alla struttura

poi uso & per avere l'indirizzo

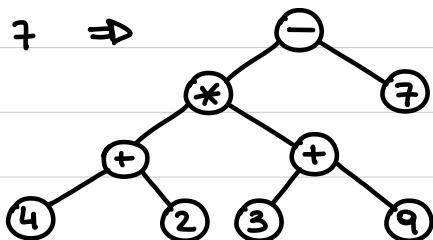
ESPRESSIONI ARITMETICHE CON ALBERI

NODI INTERMEDI: operatori ($>$ profondità \Rightarrow $>$ priorità)

FOGLIE: valori

$$(4+2) * (3+9) - 7 \Rightarrow$$

\Rightarrow si esegue una visita in POSTORDINE



```
int ValutaEspressione (TipoAlbero a) {  
    if (estVuoto(a)) { errore; }
```

```
    if (estVuoto(a->sx) && estVuoto(a->dx)){  
        return a->info; }
```

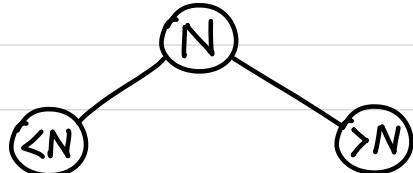
```
    int s=ValutaEspressione(a->sx);  
    int d=ValutaEspressione(a->dx);  
    if (operandoRadice == -) { return s-d; }  
    if (operandoRadice == +) { return s+d; }  
    if (operandoRadice == *) { return s*d; }  
    if (operandoRadice == /) { return s/d; }  
}
```

NB il NodoAlbero può contenere sia valori che operatori
 \Rightarrow può non bastare un solo campo info



ALBERO BINARIO DI RICERCA BINARY SEARCH TREE (BST)

è un albero tale che, per ogni nodo, tutti gli elementi del sottoalbero sx sono < del nodo, e tutti gli elementi del sottoalbero dx sono > del nodo

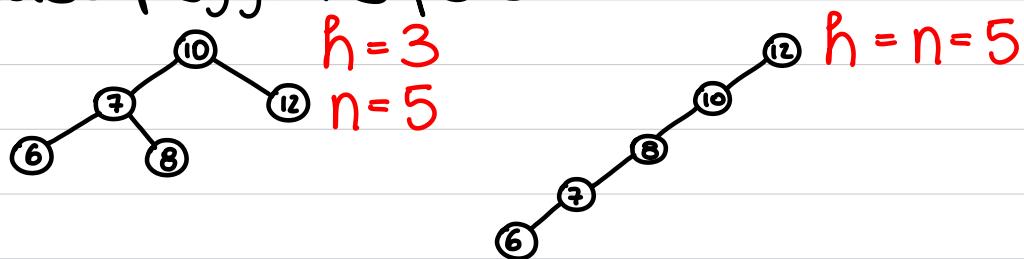


essendo questa definizione valida per OGNI nodo, dato un nodo N di un BST i suoi sottoalberi dx e sx sono a loro volta BST

QUANTO COSTA LA RICERCA SU UN BST ?

Il costo della ricerca di un valore in un albero binario qualsiasi di n elementi è $O(n)$. Nei BST ad ogni passo scendiamo di livello (non vediamo mai più nodi su uno stesso livello), quindi il costo sarà proporzionale ad h (altezza).

Nel caso peggiore però $h = n$:



E' possibile ridurre questo costo?

FATTORE DI BILANCIAMENTO

Dato un nodo, e' la differenza tra l'altezza del sottoalbero sx e l'altezza del sottoalbero dx presa in modulo

ALBERO BILANCIATO

un albero binario è bilanciato se tutti i suoi nodi hanno fattore di bilanciamento ≤ 1

SE UN BST È BILANCIATO ALLORA LA SUA H SARÀ AL PIÙ PARI A $\log n$

$$h \leq \log_2 n + 1$$

QUINDI IL COSTO DI RICERCA PER UN BST BILANCIATO SARÀ

$$O(h) = O(\log n)$$

NOTE SUGLI ALBERI

- è ST Vuoto (a) equivale ad a == NULL solo se l'albero è strutturato con SCNL (altrimenti la scrittura a == NULL non è più valida) per questo è preferibile usare funzioni valide per tutti i tipi astratti
- se vuoi cambiare la radice passa il parametro tramite puntatore TipoAlbero * a

VISITA CON L'USO DI PILA

```
InfoPila {
```

```
    TipoAlbero alb;
```

```
}
```

```
NodoPilaSCL {
```

```
    InfoPila info;
```

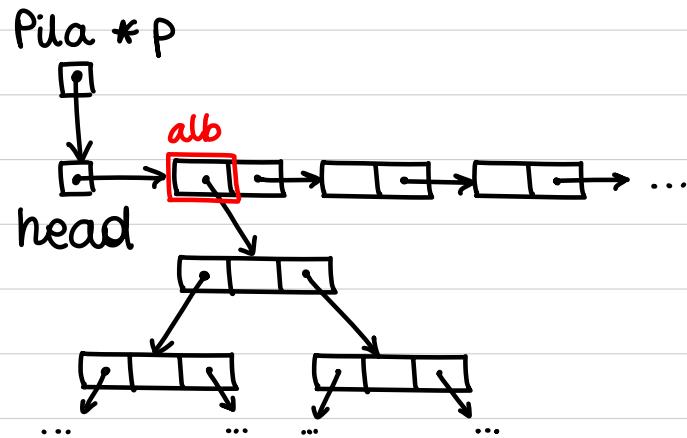
```
    NodoPilaSCL * next;
```

```
}
```

```
Pila {
```

```
    NodoPilaSCL * head;
```

```
}
```



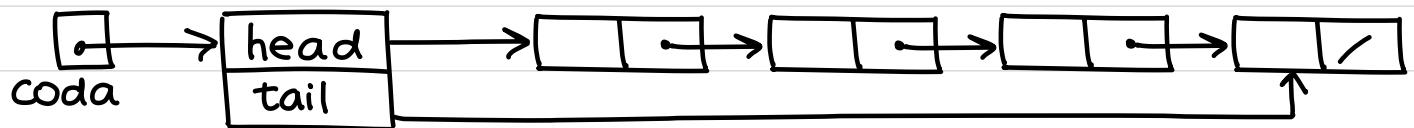
PSEUDO CODICE :

```
Pila * p = pilavuota();
push(p, a) // con a = albero VISITATO
while (!estVuota(p)) {
    e = Top(p)
    if (e != NULL) {
        pop(p);
        if (e-> destro != NULL) {
            push(p, e-> destro);
        }
        if (e-> sinistro != NULL) {
            push(p, e-> sinistro);
        }
    }
}
```

COSÌ VISITA
PRIMA i.e.
SOTTO alb.
SINISTRO !

NB • con le liste e i cicli NON basta
modificare l'ordine delle chiamate per
modificare il TIPO di VISITA
• usa le pile per le visite in PROFONDITA'

VISITA CON L'USO DI CODA



visita (TipoAlbero a) {

 Coda* p = codaVuota ();

 inCoda (p,a);

 while (!estVuota(p)) {

 e = primo (p);

 if (e != NULL) {

 outCoda (p);

 inCoda (p, e->sinistro);

 inCoda (p, e->destro);

 }

} VISITA prima
il SINISTRO

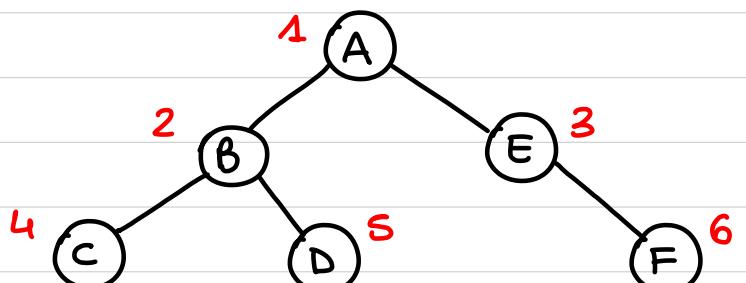
NB usa le code per le visite in AMPIEZZA, infatti



figlio sx di E



figli di C figli di D



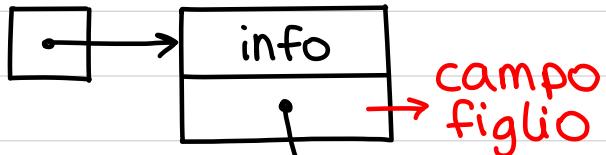
ALBERI N-ARI

Sono alberi con un numero arbitrario di figli.

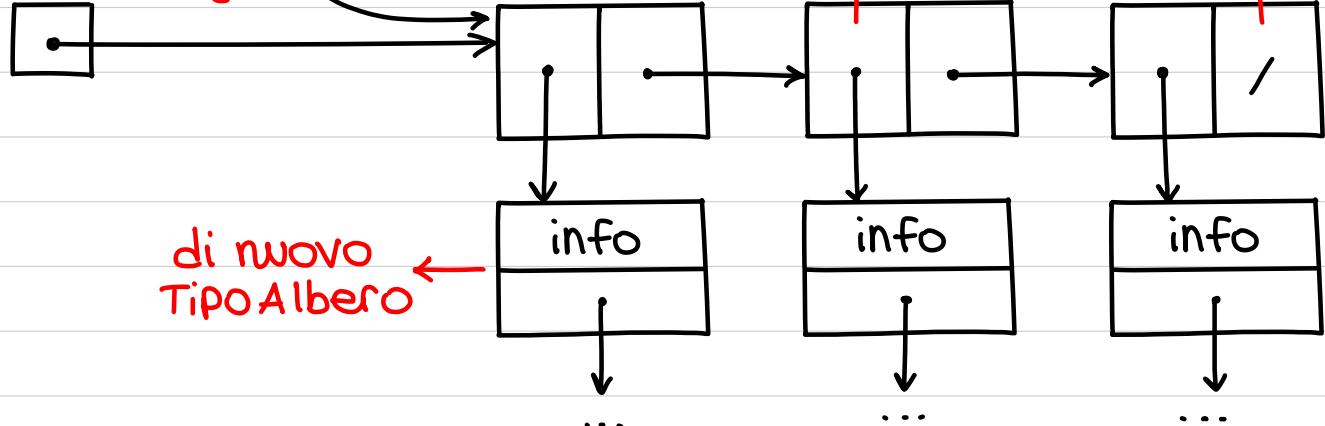
Non bastano più i soliti campi destro e sinistro

serve una struttura per memorizzare tutti i figli

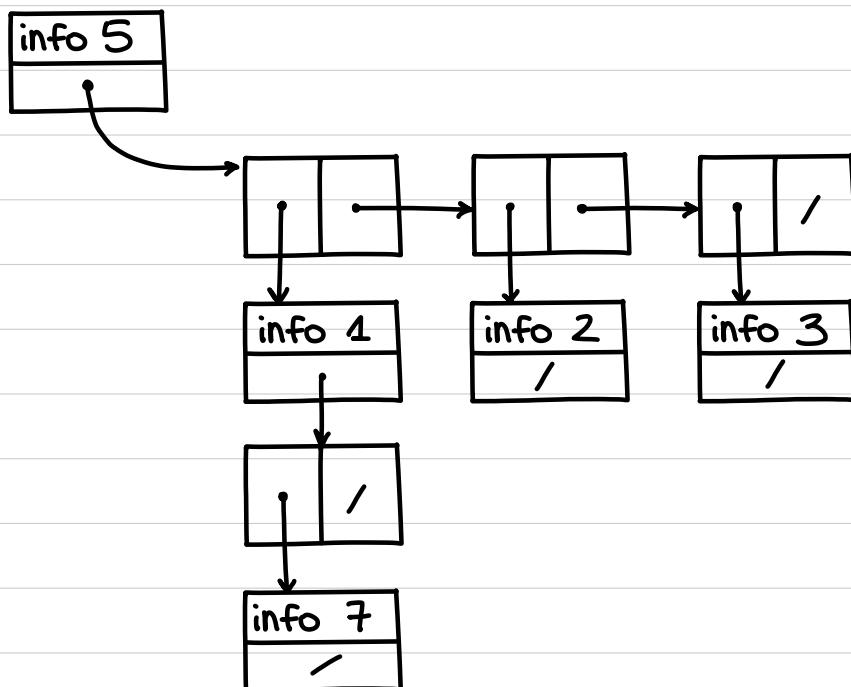
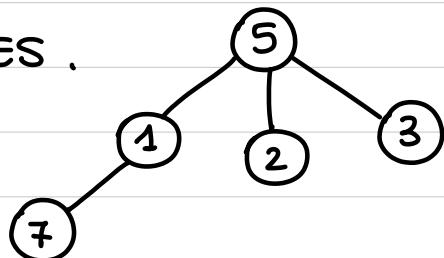
TipoAlbero



TipoNodoFiglio



ES.



TipoAlbero e TipoNodoFiglio sono strutture che hanno implementazioni reciprocamente ricorsive, cioè:

```
struct B; → ricorda di dichiararla per usarla nell'implementazione di A
struct A {
```

```
    ... B... ;
}
```

```
struct B {
    ... A... ;
}
```

L'albero n-ario vuoto è: TipoAlbero a == NULL;

VISITE DEGLI ALBERI N-ARI

IN PREORDINE: radice → tutti i figli

IN POSTORDINE: tutti i figli → radice

SIMMETRICA: non utile

Serve un procedimento che iteri su tutti gli elementi della lista (tutti i sottoalberi). Si può fare con cicli o con ricorsione doppia.

```
ƒ(Albero a) {
    if ( ) { → passo base
        operazione A ;
    }
```

```
} ƒFigli (a → figli); funzione che scansiona  
i figli  
implementata nella  
prossima pagina
```

VISITA ITERATIVA

fFigli (a) {

TipoFiglio p = a → figli \rightarrow punta al primo figlio
while (p != NULL) {
 operazione (\neq (p → albero));
 p = p → next; \rightarrow scompona tutti i campi
}

VISITA RICORSIVA

fFigli (a) {

if (a → figli == NULL) { ; }

else {

\neq (a → figli → albero); \downarrow

fFigli (a → figli → next); \downarrow

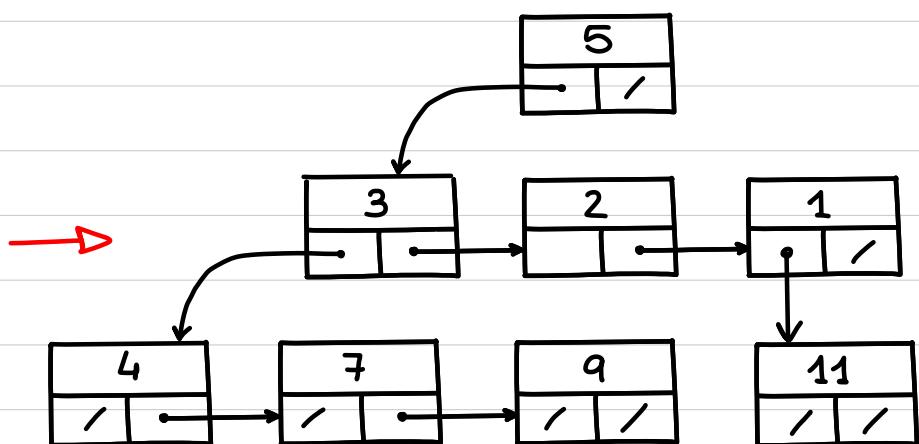
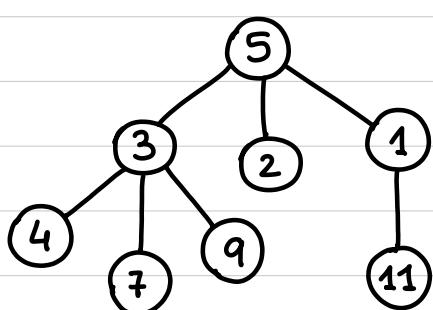
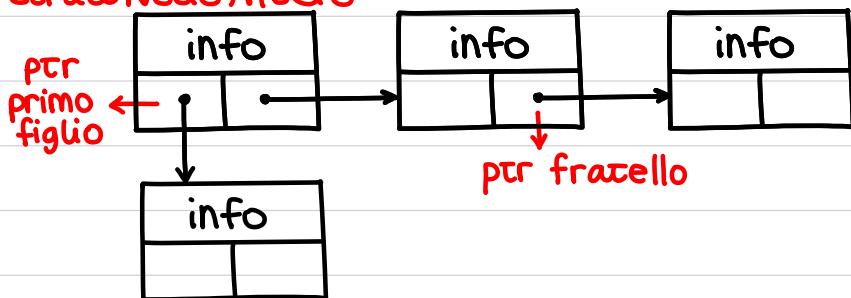
}

chiamata ricorsiva
sul primo figlio

chiamata ricorsiva
per scomporre
tutti gli altri figli

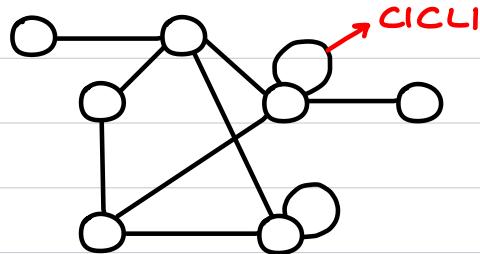
STRUTTURA ALTERNATIVA ALBERI N-ARI

struct NodoAlbero

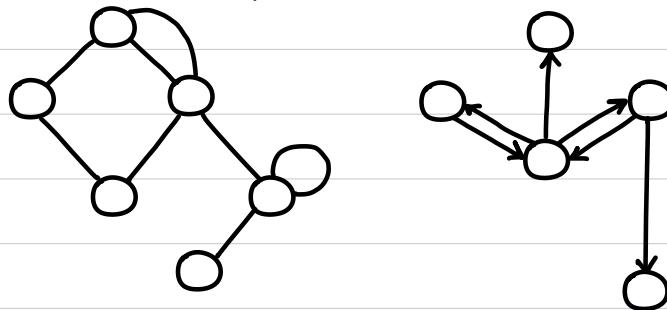


GRAFI

sono SCNL in cui non c'è gerarchia (cioè un nodo può avere più padri, più figli, e collegamenti con nodi di livelli non adiacenti).



ogni nodo puo' essere collegato con un qualsiasi sottoinsieme di nodi, compreso se' stesso



GRAFI NON ORIENTATI

la relazione di collegamento è simmetrica (se A è collegato a B, B è collegato ad A)

GRAFI ORIENTATI

le connessioni sono indicate con frecce orientate e valgono solo nel verso della freccia
($A \rightarrow B$ A è collegato a B, ma non vale i \leftrightarrow v \leftarrow)

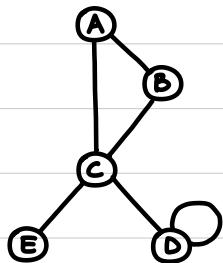
GRAFI ETICHETTATI / PESATI

Grafi che contengono info negli archi

RAPPRESENTAZIONI GRAFI NON PESATI

STATICHE

se si conosce a priori il numero di nodi



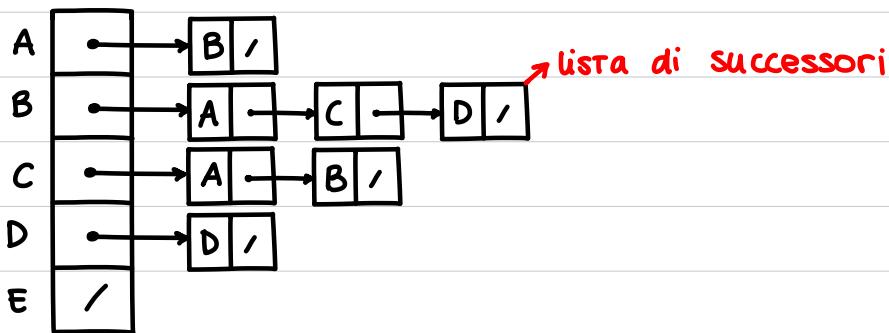
	A	B	C	D	E
A	0	1	1	0	0
B	1	0	1	0	0
C	1	1	0	1	1
D	0	0	1	1	0
E	0	0	1	0	0

→ MATRICE DI ADIACENZA

è ovviamente simmetrica in questo caso (non lo sarà) nel caso di grafo orientato

MISTE

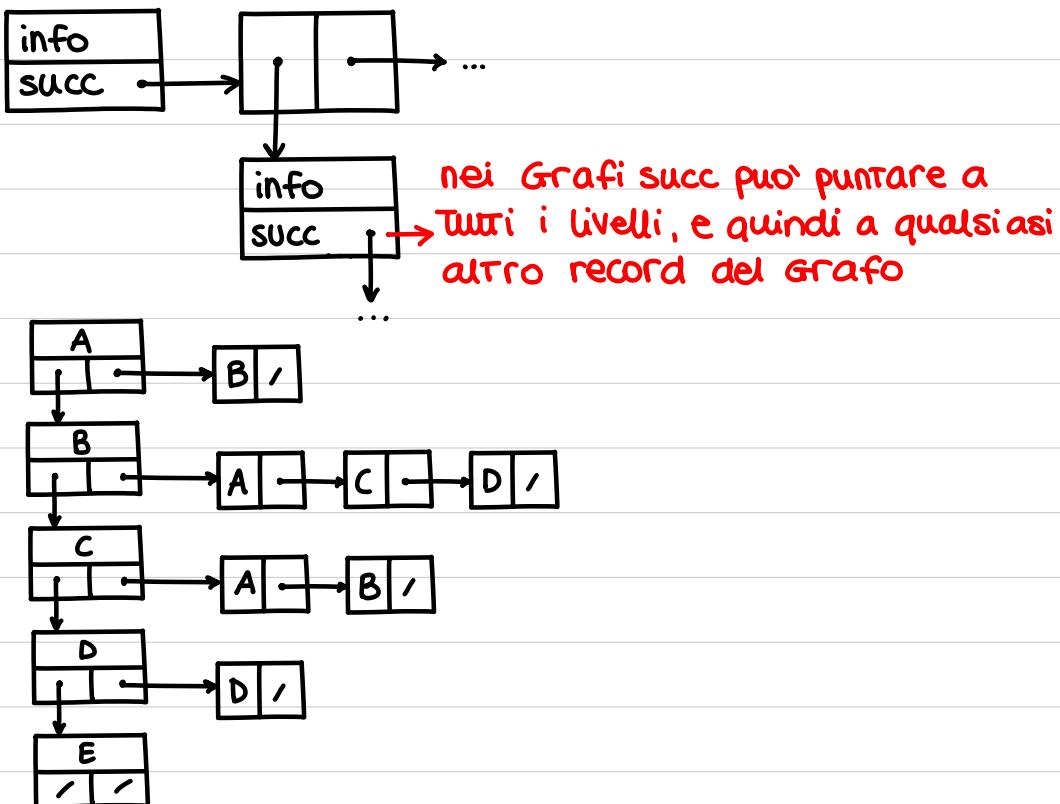
vettore (statico) + lista (dinamica)



DINAMICHE

se NON si conosce a priori il numero di nodi

StructNodo StructNodosucc



```

#define ERRORE_InfoGrafo;
typedef int TipoInfoGrafo;
struct StructNodo;
struct StructNodo {
    TipoInfoGrafo info;
    struct StructNodoSucc * succ ;
};
struct StructNodoSucc {
    struct StructNodo * nodo ;
    // int peso; se vuoi grafo pesato
    struct StructNodoSucc * next ;
};
typedef struct StructNodo * TipoGrafo;
typedef struct StructNodoSucc * TipoSuccessori;

```

VISITE DEI GRAFI

la presenza di loop ci costringe ad usare una struttura che ci permetta di riconoscere i cicli già visitati (un insieme dei nodi)

```

insieme = {};
if () → passo base
(A T1 ... Tn) → operazione su A
    Ti if (Ti → info ∈ insieme) {
        ∈(Ti) };

```

se non si usasse questa struttura dati ausiliaria, in caso di cicli l'esecuzione non terminerebbe.