

10. ALLOCAZIONE DINAMICA DELLA MEMORIA

Per allocare lo spazio in memoria per le risorse locali e/o globali di un programma, si possono scegliere sostanzialmente **due strategie differenti**:

- ALLOCAZIONE STATICA** della MEMORIA: la memoria necessaria per il dato viene allocata prima dell'esecuzione del programma a tempo di compilazione (**compile-time**);
- ALLOCAZIONE DINAMICA** della MEMORIA: consente, durante l'esecuzione di un programma (quindi a **tempo di esecuzione** ossia a **run-time**), di eseguire le istruzioni per allocare lo spazio in memoria necessario al dato e di deallocare tale spazio al termine della sua esecuzione, in modo da renderlo disponibile ad altri usi/programmi.

STRATEGIE DI ALLOCAZIONE A CONFRONTO

Aspetti valutati	Allocazione statica	Allocazione dinamica
Occupazione di memoria dei dati	L'occupazione di memoria dei dati resta costante (ossia <u>inalterabile</u>) per tutta l'esecuzione del programma e non può essere in alcun modo modificata.	L'occupazione di memoria dei dati è variabile (ossia <u>alterabile</u>) nel corso dell'esecuzione del programma in quanto verranno allocati solo quando servono e della dimensione (modificabile) che verrà ritenuta necessaria.
Tempo di esecuzione del programma	Al tempo di esecuzione del programma determinato dalle sue istruzioni non viene aggiunto il tempo di allocazione dei dati in quanto l'allocazione viene fatta una volta sola prima dell'esecuzione del programma stesso. Quindi l'allocazione statica dei dati non appesantisce in alcun modo il tempo di esecuzione di un programma.	Al tempo di esecuzione del programma determinato dalle sue istruzioni deve essere aggiunto il tempo di allocazione (e successiva deallocazione) dei dati in quanto essa avverrà nel corso dell'esecuzione del programma stesso. Quindi l'allocazione dinamica dei dati appesantisce il tempo di esecuzione di un programma.
Tempo di esistenza dei dati (visibilità o scope)	I dati sono visibili (quindi esistono e <u>sono utilizzabili</u>) dall'inizio alla fine del programma in quanto fin dall'inizio dell'esecuzione del programma stesso ad ognuno di essi viene associata una zona di memoria di lunghezza costante che verrà rilasciata solo quando il programma terminerà.	I dati sono visibili (quindi esistono e <u>sono utilizzabili</u>) tra la loro allocazione e la successiva deallocazione che avvengono durante l'esecuzione del programma (ossia a run-time) con particolari chiamate a sottoprogrammi.

SPECIALE: L'esecuzione di un sottoprogramma

Per eseguire un sottoprogramma è necessario utilizzare una apposita **istruzione di chiamata di sottoprogramma** che è prevista da tutti i linguaggi di programmazione.

Meccanismo di funzionamento

Quando un programma non è in esecuzione risiede su una **MEMORIA DI MASSA** e subito dopo la compilazione ed il linkaggio, sarà costituito esclusivamente dal codice o istruzioni e dai dati ed occuperà un'area di memoria (la cui dimensione in BYTE dipende esclusivamente dalle istruzioni e dai dati utilizzati) che è possibile essere pensata come suddivisa in **due segmenti**:

- **il Segmento "CODICE"** o "ISTRUZIONI": area contenente le istruzioni del programma (codice) scritte in linguaggio macchina
- **il Segmento "DATI"**: area contenente variabili e costanti allocate staticamente

Quando un programma è in esecuzione (chiamato anche **TASK** o **processo**) viene allocato in memoria centrale (**RAM**) e gli viene assegnato, dal sistema operativo (un particolare programma chiamato **loader**), nella memoria di lavoro (**RAM**) anche una zona di memoria aggiuntiva rispetto a quella posseduta quando è "in quiete".

Un programma in esecuzione quindi occuperà una zona della memoria di lavoro che è possibile pensare ora suddivisa in **quattro segmenti**:

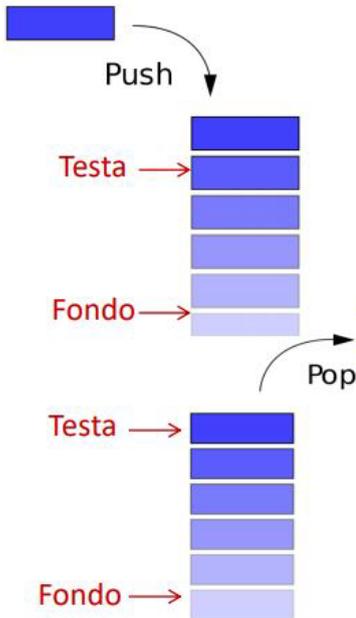
- **il Segmento "CODICE"** o "ISTRUZIONI": *vedi descrizione già data in precedenza*
- **il Segmento "DATI"**: *vedi descrizione già data in precedenza*
- **il Segmento "STACK"**: area destinata a gestire la "**PILA DELLE ATTIVAZIONI**"
- **il Segmento HEAP** (lett. mucchio) **di sistema**: area destinata a raccogliere i dati gestiti dinamicamente che come tali verranno allocati e deallocati dinamicamente (che verranno illustrati IN SEGUITO)

Una volta terminata l'esecuzione del programma questa zona di memoria allocata nella RAM verrà rilasciata liberando la memoria precedentemente occupata (**allocazione dinamica del codice da parte del sistema operativo**)



L'intero meccanismo della gestione delle chiamate a sottoprogrammi avviene, come già accennato in precedenza, tramite una particolare struttura dati detta "PILA DELLE ATTIVAZIONI".

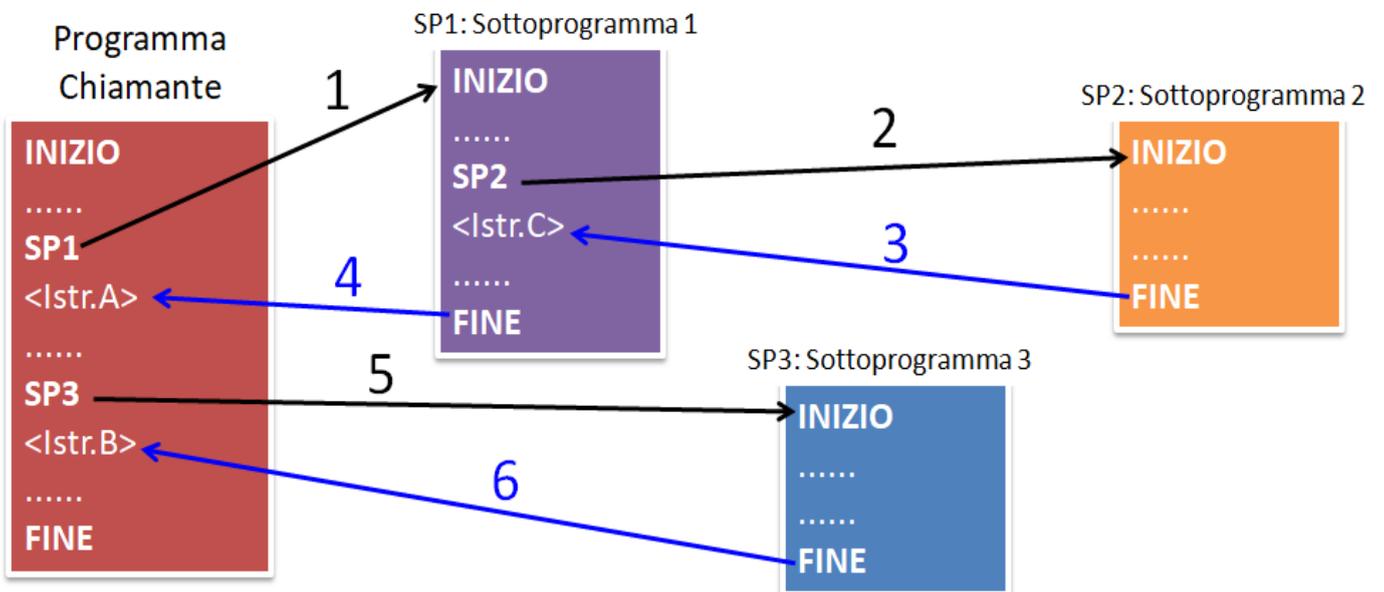
Per ricordare da quale istruzione va ripresa l'esecuzione dopo un sottoprogramma, la CPU si serve di una apposita STRUTTURA DATI detta **PILA delle attivazioni** o **STACK** dalla quale i dati possono essere inseriti o estratti solo da una estremità che viene detta **testa della pila**.



N.B. Questa struttura dati ha una struttura di tipo **LIFO o Last In First Out** nel senso che l'ultimo ad entrare è il primo ad uscire (esempio pila di piatti o di cd).

Quando la CPU esegue una istruzione di chiamata a sottoprogramma allora inserisce (**Push**) nella pila delle attivazioni in testa l'indirizzo della cella di memoria contenente l'istruzione che dovrà essere eseguita al rientro dal sottoprogramma.

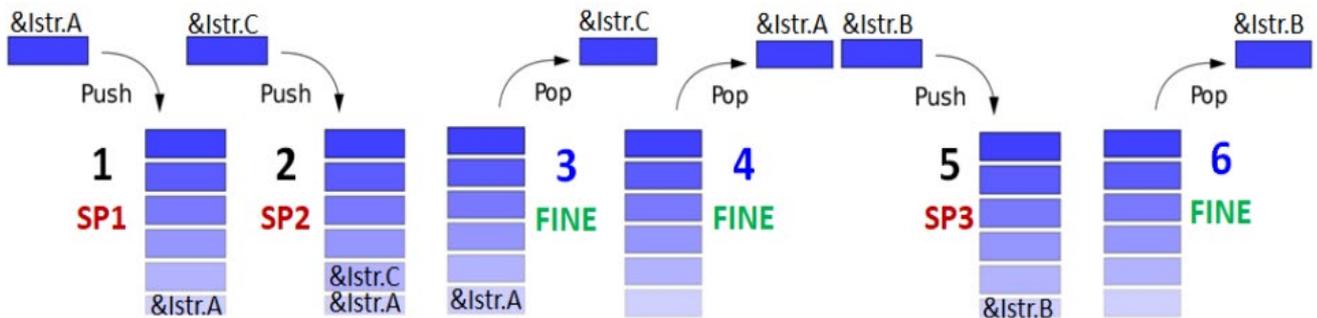
Quando la CPU esegue una istruzione di FINE allora utilizza la pila delle attivazioni per estrarre dalla testa (**Pop**) l'indirizzo di memoria in esso contenuto da dove riprendere l'esecuzione.



Tutto questo avviene utilizzando la “PILA DELLE ATTIVAZIONI” nel seguente modo:

- Quando la CPU esegue una istruzione di chiamata a sottoprogramma allora inserisce nella pila delle attivazioni, IN TESTA, l'indirizzo della cella di memoria contenente l'istruzione che dovrà essere eseguita al rientro dal sottoprogramma.
- Quando la CPU esegue una istruzione di FINE allora utilizza la pila delle attivazioni per estrarre, DALLA TESTA, l'indirizzo della cella di memoria in esso contenuto da dove riprendere l'esecuzione.

Nell'esempio specificato in precedenza l'utilizzo della pila delle attivazioni da parte del nostro programma in esecuzione (TASK) sarà il seguente:

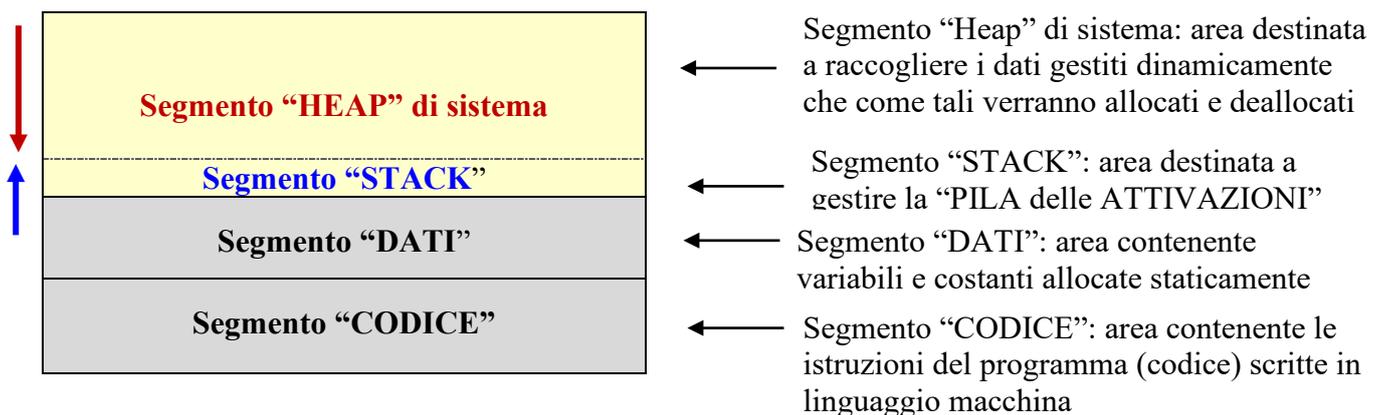


SCENARI POSSIBILI

Cosa può accadere ad un programma in esecuzione che oltre all'uso di sottoprogrammi, impiega l'allocazione dinamica per la gestione dei dati (anche solo in parte) ?

SCENARIO A)

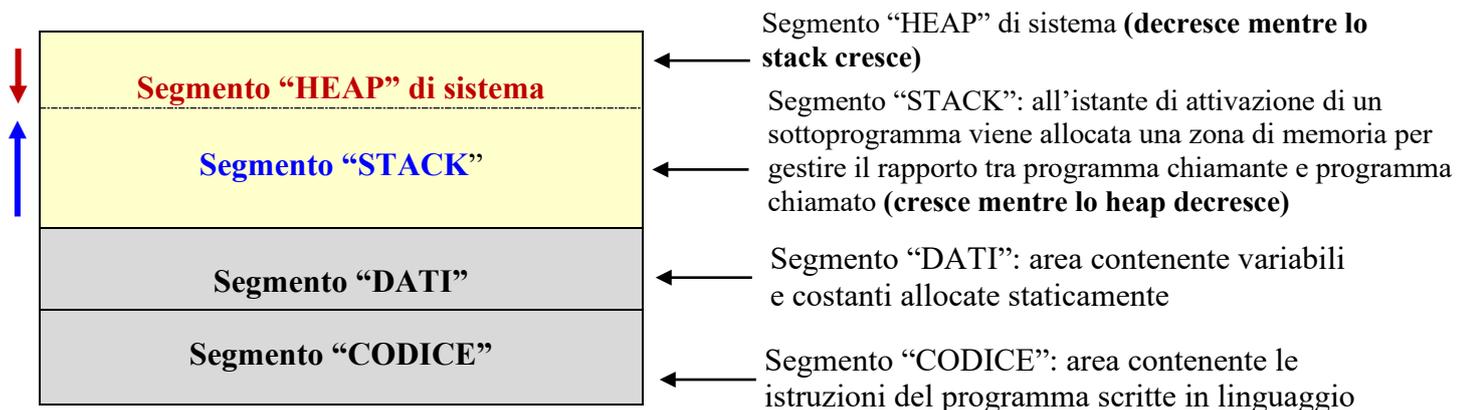
Se un programma non utilizza sottoprogrammi, impiega solamente risorse globali e locali ed occuperà sempre la stessa quantità di memoria, **non impiegando affatto il segmento stack** che permette di gestire le chiamate a funzioni e procedure tramite una struttura dati chiamata “pila delle attivazioni”.



SCENARIO B)

Se un programma utilizza sottoprogrammi impiega, oltre alle risorse globali e locali, **anche il segmento stack** inserendo (operazione **push**), ad ogni **chiamata** (o **attivazione**) di un sottoprogramma nella “pila delle attivazioni”, l’indirizzo dell’istruzione successiva alla chiamata stessa, che verrà poi richiamato alla fine dell’esecuzione del sottoprogramma stesso (operazione **pop**).

La “pila delle attivazioni” gestisce le informazioni fondamentali relative alle chiamate di procedure e funzioni utilizzando la tecnica **L.I.F.O.** (**Last In First Out** ossia “l’ultimo dato ad entrare è il primo ad uscire”).



N.B. L’allocazione all’interno dello heap di locazioni di memoria avverrà in entrambi i casi (SCENARIO A oppure SCENARIO B) solo se verranno utilizzate variabili allocate dinamicamente tramite l’utilizzo di un’apposita funzionalità - **la PROCEDURA *Alloca* ()** - che verrà spiegata più avanti.

E’ evidente che lo **heap** ossia l’area libera dove allocare dinamicamente i dati e lo **stack** ossia l’area di memoria dove vengono gestite le risorse dei sottoprogrammi, **si contendono l’area libera assegnata**, muovendosi verso direzioni convergenti ossia da direzioni opposte.

N.B. Una eventuale collisione provocherà di fatto l’esaurimento della dimensione totale di memoria messa a disposizione dei due segmenti (HEAP + STACK) e di conseguenza l’interruzione (abend o crash) del programma

Al contrario del segmento “stack”, il segmento “dati” ha una dimensione prefissata.

Questo è uno dei motivi per il quale si giustifica il ricorso all’allocazione dinamica delle variabili aggirando il problema che spesso le variabili necessarie al programma non riescono ad essere collocate tutte all’interno del segmento “dati”.

I PUNTATORI

Non tutti i linguaggi di programmazione permettono l'uso dell'allocazione dinamica della memoria da parte del programmatore ma è **indubbio che questa possibilità fornisca a tali linguaggi una marcia in più.**

Il linguaggio C (ed anche il linguaggio il C++) fortunatamente permettono al programmatore di gestire l'allocazione dinamica della memoria a disposizione un particolare tipo di dato chiamato **puntatore.**

Def: una variabile di TIPO PUNTATORE contiene un valore intero (espresso in esadecimale) che rappresenta l'indirizzo della locazione di memoria nella quale è memorizzato il dato cui si riferisce

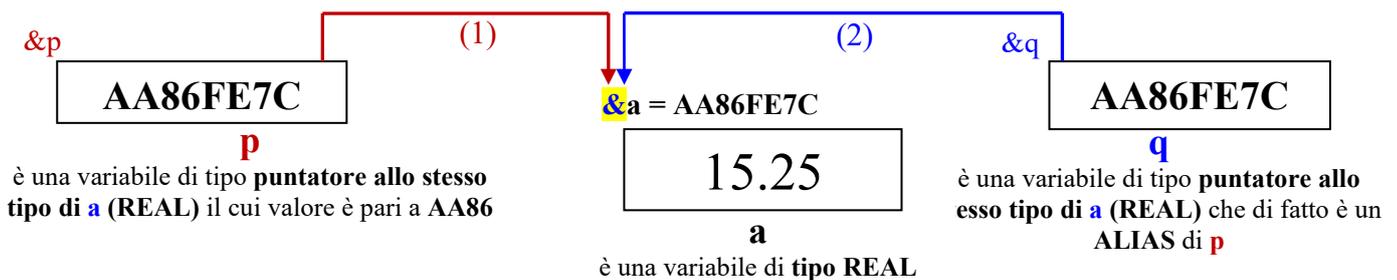
In altre parole una variabile di tipo puntatore fisicamente rappresenta una locazione di memoria che contiene l'indirizzo di un'altra locazione di memoria.

Con la PSEUDOCODIFICA la dichiarazione di una variabile di tipo puntatore sarà del tipo
 < variabile_puntatore > : **PUNTATORE A** < Tipo_dato_puntato >

ALGORITMO Esempio_0
PROCEDURA main()
p, q : PUNTATORE A REAL
a : REAL
INIZIO
 a ← 15.25
 p ← &a (1)
 q ← p (2)
 Scrivi (*p)
 Scrivi (*q)
FINE

Graficamente
 (ipotizzando indirizzi ad 8 BYTE)

```
// Utilizzando il linguaggio C - Esempio_0
#include <stdio.h>
#include <stdlib.h>
int main(int argc, char *argv[])
{
float* p, *q; //N.B. da notare * vicino a q, altrimenti float
float a;
a = 15.25;
p = &a; // (1)
q = p; // (2)
printf("%.2f", *p);
printf("\n%.2f", *q);
printf("\nIndirizzo di a = %x", &a); //aggiunte per spiegazione
printf("\nvalore di p = %x", p);
printf("\nvalore di q = %x", q);
return 0;
}
```



Osservazioni importanti

- Un puntatore quindi **non contiene direttamente il dato** come le altre variabili di altri tipi, ma contiene **un indirizzo di memoria dove reperire il dato.**
- Una variabile puntatore **occupa sempre la stessa quantità di memoria** indipendentemente dal tipo di dato puntato.
- Per riferirci al valore del dato puntato da un puntatore** sia in modalità STATICA che DINAMICA, DOBBIAMO utilizzare il simbolo ***** davanti al nome della variabile puntatore
- Per assegnare l'indirizzo di una cella di memoria ad un puntatore** in modalità STATICA DOBBIAMO utilizzare il simbolo **&** davanti al nome della variabile mentre in modalità DINAMICA DOBBIAMO utilizzare la chiamata ad un opportuno sottoprogramma.

NOTA BENE

UN PUNTATORE può essere usato anche SENZA UTILIZZARE L'ALLOCAZIONE DINAMICA.....

Infatti li abbiamo già utilizzati:

a) PASSAGGIO DEI PARAMETRI PER RIFERIMENTO O INDIRIZZO

b) ASSOCIARLI A VARIABILI ALLOCATE STATICAMENTE (vedi anche esempio precedente)

ALGORITMO Esempio_1

PROCEDURA main()

a : INT

p : PUNTATORE A INT

INIZIO

p ← NULL //p non punta ad indirizzo valido

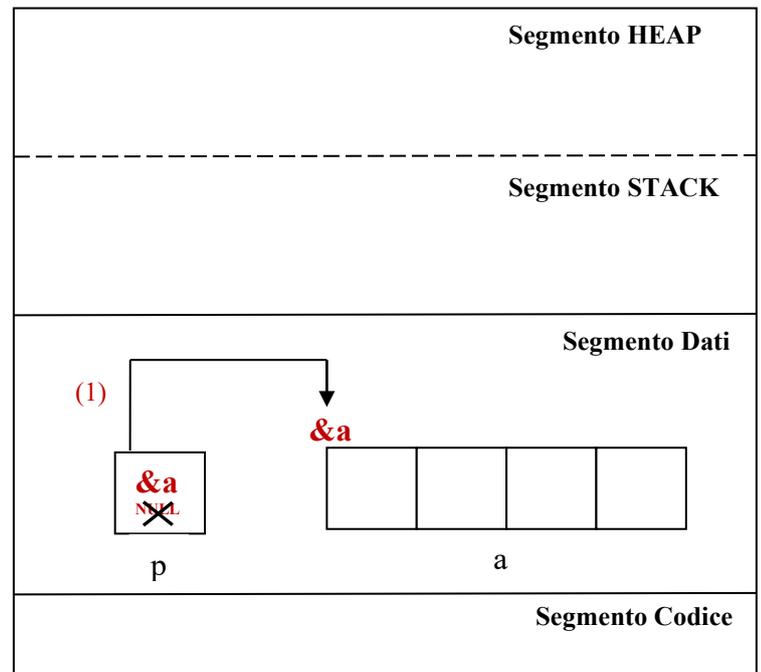
p ← &a // (1)

Leggi(*p) //invece di **Leggi(a)**

*p ← *p + 10 //invece di **a ← a + 10**

Scrivi(*p) //invece di **Scrivi(a)**

FINE



```
// Utilizzando il linguaggio C - Esempio_1
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[])
{
    int a;
    int* p;

    p = NULL; //p non punta ad indirizzo valido
    p = &a; // (1)
    fflush(stdin);
    scanf("%d", p); //invece di scanf("%d", &a);
    *p = *p + 10; //invece di a = a + 10;
    printf("%d", *p); //invece di printf("%d", a);
    return 0;
}
```

Il valore simbolico **NULL** viene usato per indicare che un puntatore non contiene al momento un indirizzo di memoria valido poiché il valore **0** invece è utilizzato come indirizzo in diverse applicazioni di sistema. Dato che l'indirizzo di memoria **NULL** (ma anche **0**) non è utilizzabile da nessun programma utente, qualsiasi tentativo di accesso a questa posizione, sia in scrittura che in lettura, produrrà un errore bloccante. Quindi, dopo aver posto **p ← NULL** oppure **p ← 0**, tentare di accedere a ***p** genererà un errore bloccante. Istruzioni successive a tale inizializzazioni quali ***p ← 4** oppure **a ← *p** produrranno anch'esse un errore bloccante. Nel **linguaggio C** la costante **NULL** è definita nella libreria standard **stdlib.h**

..... MA

L'ALLOCAZIONE DINAMICA non può esistere SENZA L'UTILIZZO DEI PUNTATORI

USO “DISEGNI” DI TRACCIA CON PUNTATORI USATI IN ALLOCAZIONE STATICA**ALGORITMO** Puntatori_1**PROCEDURA** main ()

p1, p2, p3 : PUNTATORE A INT

a, b, c : INT

INIZIO

Leggi (a)

Leggi (b)

Leggi (c)

p2 ← &c

p1 ← &b

p3 ← p1

a ← ((*p2) + (*p1)) DIV 3

*p1 ← a * (*p2) - 2*(*p3)

*p2 ← a + (*p1) - (*p2)

p3 ← p2

p2 ← &(*p1)

p1 ← &a

*p1 ← ((*p2) * 4) % (*p3)

*p2 ← (*p1) + (*p2) - 3*(*p3)

*p3 ← ((*p1) - (*p2)) DIV 5

Scrivi (a)

Scrivi (b)

Scrivi (c)

Scrivi (*p1)

Scrivi (*p2)

Scrivi (*p3)

Scrivi (p1)

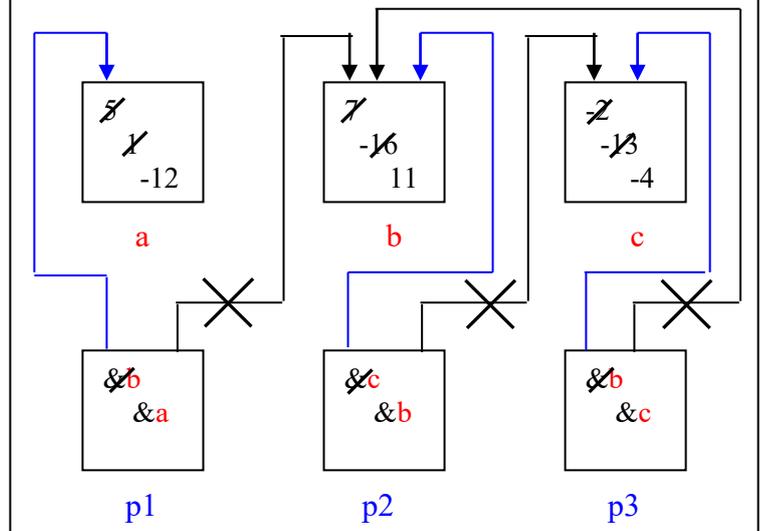
Scrivi (p2)

Scrivi (p3)

RITORNA**FINE**

Esercizio) Dire quale sarà il valore di **a, b, c, *p1, *p2, *p3, p1, p2, p3** dopo avere eseguito lo pseudocodice dell'algoritmo “Puntatori_1” seguente, illustrando il ragionamento eseguito per ottenere il risultato attraverso uno o più disegni esplicativi, nel caso l'utente immetta i seguenti valori iniziali:

a = 5, b = 7, c = -2

Segmento Dati**Soluzione**

a	b	c	*p1	*p2	*p3	p1	p2	p3
-12	11	-4	-12	11	-4	&a	&b	&c

Calcoli da eseguire per costruzione dello schema grafico (escluse le istruzioni di I/O)

p2 ← &c (p2 = &c) (ossia p2 punterà alla variabile c: occorre disegnare la relativa freccia)

p1 ← &b (p1 = &b) (ossia punterà alla variabile b: occorre disegnare la relativa freccia)

p3 ← p (p3 = p2 = &b) (ossia punterà alla medesima variabile puntata da p1 occorre disegnare la relativa freccia)

a ← ((*p2) + (*p1)) DIV 3 (a = ((-2) + 7) DIV 3 = 5 DIV 3 = 1)

*p1 ← a * (*p2) - 2*(*p3) (*p1 = b = 1 * (-2) - 2 * 7 = -2 - 14 = - 16)

*p2 ← a + (*p1) - (*p2) (*p2 = c = 1 + (-16) - (-2) = 1 - 16 + 2 = -13)

p3 ← p2 (p3 = p2) (ossia p3 punterà alla medesima variabile puntata da p2: disegnare la nuova freccia e cancellare precedente freccia)

p2 ← &(*p1) (p2 = p1) (ossia punterà p2 alla medesima variabile puntata da p1: disegnare la nuova freccia e cancellare precedente freccia)

p1 ← &a (p1 = &a) (ossia p1 punterà alla variabile a: disegnare la nuova freccia e cancellare precedente freccia)

*p1 ← ((*p2) * 4) % (*p3) (*p1 = a = ((-16) * 4) % (-13) = (-64) % (-13) = -12)

*p2 ← (*p1) + (*p2) - 3*(*p3) (*p2 = b = -12 + 8 - 16) - 3*(-13) = -28 + 39 = 11)

*p3 ← ((*p1) - (*p2)) DIV 5 (*p3 = c = (-12 - 11) DIV 5 = (-23) DIV 5 = -4)

COME ALLOCARE DINAMICAMENTE I DATI NELLO HEAP

FUNZIONE **DimensioneDi** (*VAL* <variabile> | <tipo dato>) : **INT**

E' un SOTTOPROGRAMMA (**FUNZIONE**) che restituisce la quantità di memoria espressa in byte (numero intero) che una variabile oppure un determinato tipo di dato occupa in memoria. E' di grande utilità in quanto evita al programmatore il calcolo della quantità di memoria da allocare nello heap per contenere un dato.

Può essere utilizzata indipendentemente dal tipo di allocazione che si intende eseguire.

ALGORITMO Esempio_2

PROCEDURA main()

a : **INT**

nbyte : **INT**

INIZIO

//N.B. Verrà restituito in entrambi i casi 4

nbyte ← **DimensioneDi** (a)

Scrivi(nbyte)

// oppure

nbyte ← **DimensioneDi** (**INT**)

Scrivi(nbyte)

FINE

```
// Utilizzando il linguaggio C -Esempio_2
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[])
{
    int a;
    int nbyte;

    nbyte = sizeof(a);
    printf("%d", nbyte);

    // oppure

    nbyte = sizeof(int);
    printf("\n%d", nbyte);

    return 0;
}
```

PROCEDURA **Alloca** (*REF* <nome puntatore> : **PUNTATORE A** <tipo dato puntato>, *VAL* <numero byte> | **DimensioneDi** (<variabile> | <tipo dato>))

E' un SOTTOPROGRAMMA (**PROCEDURA**) che assegna al puntatore <nome puntatore> l'indirizzo di un'area di memoria **ALLOCATA DINAMICAMENTE** situata nello heap con dimensione in byte pari al valore del secondo parametro, da utilizzare successivamente per memorizzare un dato.

Qualora tale allocazione non fosse possibile per ragioni tecniche o perché è terminata la memoria a disposizione tra heap e segmento stack, la procedura **Alloca** () assegnerà al puntatore il valore **NULL**.

Tale valore **NULL** (letteralmente "niente") abbiamo già visto essere un particolare valore **COSTANTE** che sta ad indicare che il puntatore non è associato a nessuna area di memoria correttamente allocata ossia che non punta a nessuna area di memoria, indipendentemente dalla modalità di allocazione.

PROCEDURA **Dealloca** (*VAL* <nome puntatore> : **PUNTATORE A** <tipo dato puntato>)

E' un SOTTOPROGRAMMA (**PROCEDURA**) che rilascia la memoria occupata nello heap da una precedente operazione di **Alloca**() rendendola disponibile di nuovo per eventuali successive allocazioni dinamiche.

Da notare la chiamata a tale procedura in pratica romperà il link o collegamento che si era creato tra il puntatore e la zona di memoria puntata nello heap ma **NON LO ANNULERÀ**' (ossia esso non verrà posto a **NULL**), in quanto conterrà ancora il valore che aveva ricevuto in precedenza ma che in ogni caso, non gli permetterà più il raggiungimento dell'area di memoria associata in precedenza.

Esempio:

ALGORITMO Esempio_3

PROCEDURA main()

p : **PUNTATORE A INT**

INIZIO

Alloca (p, DimensioneDi (INT)) (1)

SE (p ≠ NULL)

ALLORA

Scrivi("Allocazione OK!")

<B1> //Blocco di istruzioni da eseguire

//se allocazione ha avuto esito positivo

Dealloca (p) (2)

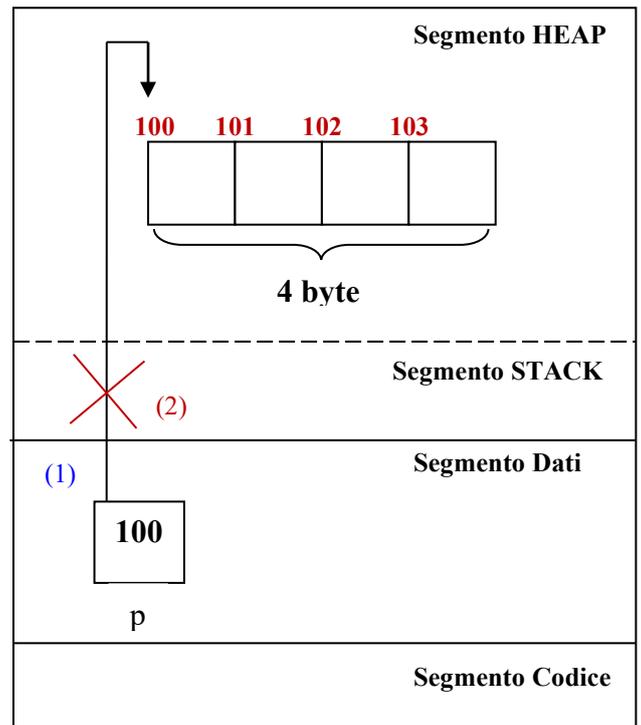
ALTRIMENTI

//se allocazione ha avuto esito negativo

Scrivi("Allocazione non riuscita!")

FINE SE

FINE



```
// Utilizzando il linguaggio C - Esempio_3
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[])
{
    int* p;
    //La funzione malloc restituisce un void* su cui va fatto un CAST a seconda del tipo
    p = (int*) malloc (sizeof(int)); // (1)
    if (p != NULL)
    {
        printf("Allocazione OK!");
        //<B1>
        free(p); // (2)
    }
    else
    {
        printf("Allocazione non riuscita!");
    }
    return 0;
}
```

NOTA BENE**ALGORITMO** Esempio_3_NOT_OK**PROCEDURA** main()

p : PUNTATORE A INT

INIZIO

Alloca (p, DimensioneDi (INT))

SE (p ≠ NULL)**ALLORA**

<B1> //Blocco di istruzioni da eseguire
 //se allocazione ha avuto esito positivo

ALTRIMENTI

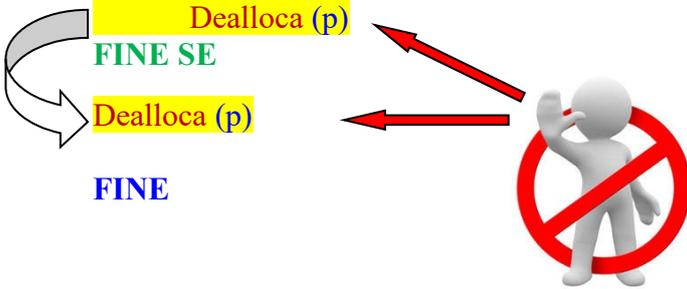
//se allocazione ha avuto esito negativo

Scrivi("Allocazione non riuscita!")

Dealloca (p)

FINE SE

Dealloca (p)

FINE

Il posizionamento della chiamata alla procedura **Dealloca()** all'interno del codice nella posizione indicata, può avere per l'esecuzione del programma esiti infausti ed imprevedibili (perfino un **abend** oppure un **crash**) in quanto, in caso di allocazione con esito negativo, si tenterà di deallocare una zona di memoria inesistente. Tale errore è reso ancora più subdolo dal fatto che non sarà possibile accorgersene fino a che l'allocazione dinamica in questione avrà esito positivo.

GARBAGE COLLECTION ed ALIASING: SIDE EFFECT E DANGLING REFERENCE

Uno dei problemi principali quando si utilizza l'allocazione dinamica è quello della deallocazione delle vecchie zone di memoria allocate (chiamata **garbage collection** lett. raccolta dei rifiuti).

Per recuperare il garbage (**garbage collection**) esistono due modalità :

(*) **modalità manuale** (a carico del programmatore) con opportune istruzioni del programma (ad esempio la procedura **Dealloca**());

(*) **modalità automatica** (a carico del sistema operativo) che periodicamente pulisce la memoria dai dati inutilizzata grazie ad un programma (**garbage collector**) che si occupa di recuperare locazioni oramai inaccessibili liberando porzioni di memoria dello heap.

Il fenomeno dell'**ALIASING** si ha quando un puntatore può essere creato come sostituto (**alias**) di un altro ossia quando entrambi puntano alla stessa locazione di memoria allocata sia staticamente sia dinamicamente (*n.b. basta assegnare ad un puntatore il valore dell'altro*)

Tale fenomeno, perfettamente lecito ed estremamente utile, può dare origine a gravi effetti collaterali (**SIDE EFFECT**) in grado di bloccare l'esecuzione del programma (abend o crash) causati da un programmatore poco "attento" .

Ad esempio è possibile avere zone di memoria allocate dinamicamente che diventano **inaccessibili** (ossia allocate in memoria ma mai deallocate) non solo per la mancata deallocazione del programmatore, ma per l'esecuzione di istruzioni particolari (vedi aliasing) che finiscono per avere un effetto collaterale (**side effect**) non previsto.

Esempio 4: Effetto collaterale ALIASING

ALGORITMO Esempio_4_Dangling_Reference

PROCEDURA main()

p, q : PUNTATORE A INT

INIZIO

Alloca (p, DimensioneDi (INT)) (1)

SE (p ≠ NULL)

ALLORA

q ← p (2) //ALIASING

.....

Dealloca (p) (3)

//Attenzione al terribile SIDE EFFECT

//ORA la zona puntata da q non è più accessibile

//ed il puntatore q è comunque "referenziato"

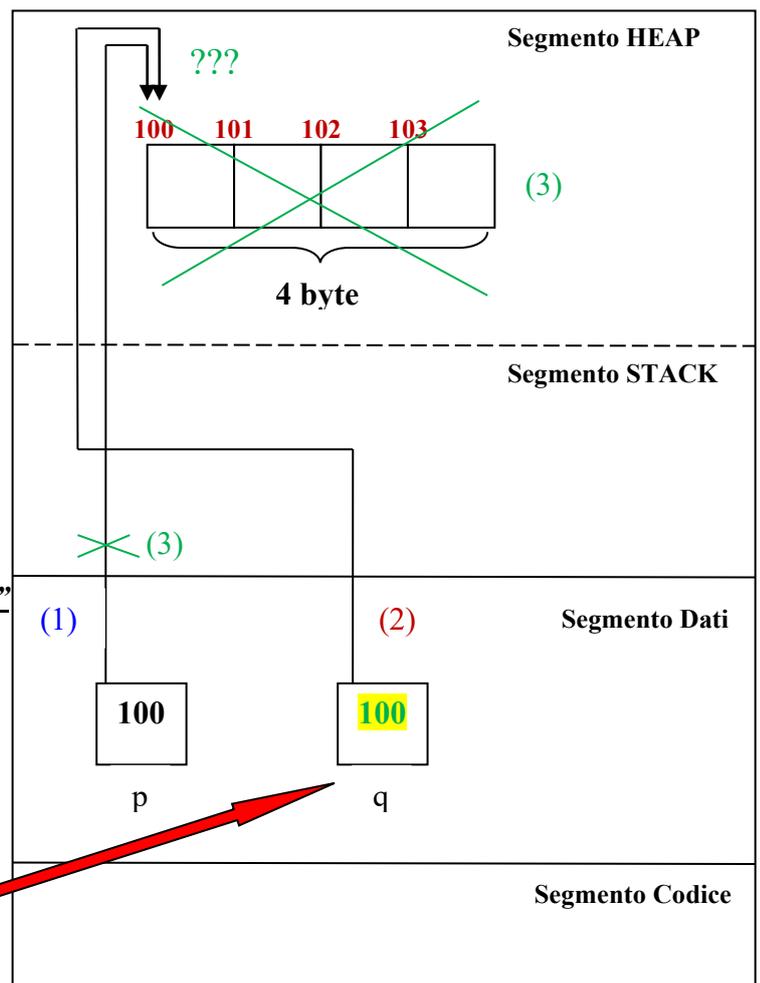
//DANGLING REFERENCE

ALTRIMENTI

Scrivi("Allocazione non riuscita!")

FINE SE

FINE





Esempio 5: Effetto collaterale ALIASING

ALGORITMO Esempio_5_Area_Zombie

PROCEDURA main()

p, q : PUNTATORE A INT

INIZIO

Alloca (p, DimensioneDi (INT)) (1)

SE (p ≠ NULL)

ALLORA

Alloca (q, DimensioneDi (INT)) (2)

SE (q ≠ NULL)

ALLORA

p ← q (3) //ALIASING

Dealloca (q)

//Attenzione al terribile SIDE EFFECT

//Ora c'è un area di memoria inaccessibile

//NON puntata più da alcun puntatore

// che non può essere più rilasciata

ALTRIMENTI

Scrivi("Allocazione di q non riuscita!")

FINE SE

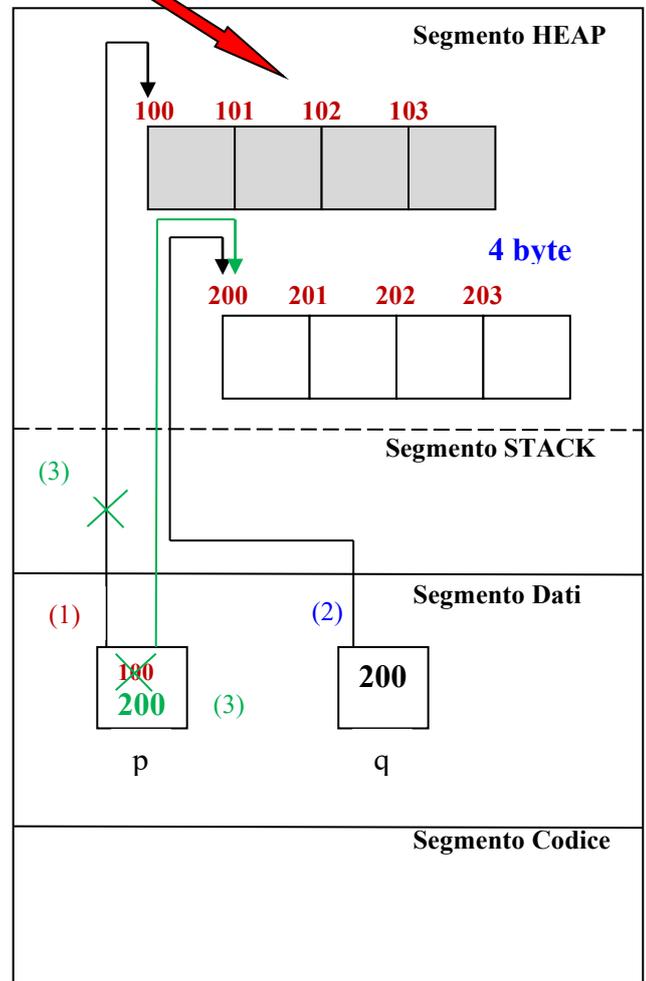
Dealloca (p)

ALTRIMENTI

Scrivi("Allocazione di p non riuscita!")

FINE SE

FINE



OPERAZIONI DI CONFRONTO SUI PUNTATORI

Le uniche **operazioni di confronto** consentite e che hanno senso tra puntatori che puntano a dati dello stesso tipo, sono quelle di **uguaglianza (=)** e **diversità (≠)**.

Esempio: Confronto tra due puntatori

ALGORITMO Esempio_6_Confronto_Ptr

PROCEDURA main()

p, q : **PUNTATORE A INT**

a, b: **INT**

esito : **BOOL**

INIZIO

a ← 5

b ← 5

p ← &a // (1)

q ← &b // (2)

SE (p ≠ q) //N.b. 100 ≠ 200 **VERO**

ALLORA

esito = **VERO**

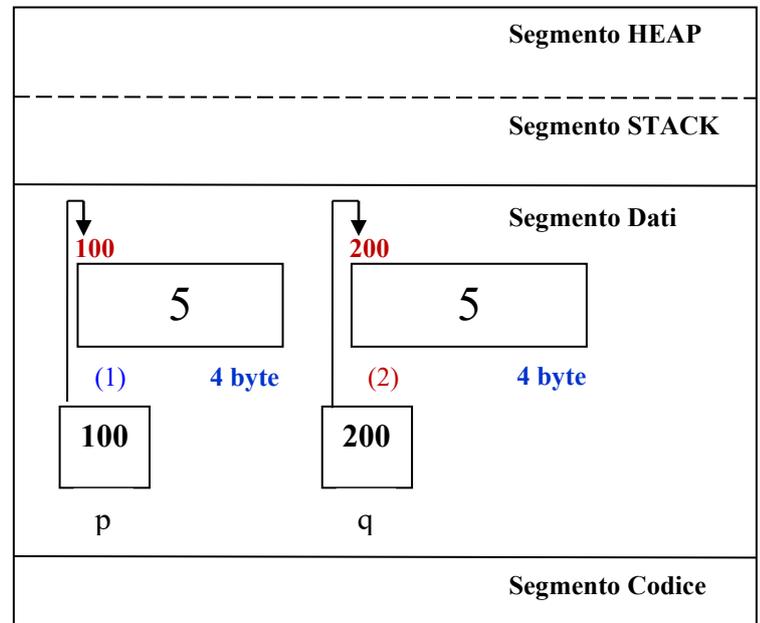
ALTRIMENTI

esito = **FALSO**

FINE SE

Scrivi (esito)

FINE



ALGORITMO Esempio_7_Confronto_Ptr

PROCEDURA main()

p, q : **PUNTATORE A INT**

esito : **BOOL**

INIZIO

Alloca (p, DimensioneDi (INT)) (1)

SE (p ≠ NULL)

ALLORA

q ← p (2) //ASSEGNAZIONE tra puntatori

SE (p = q)

ALLORA

esito = **VERO**

ALTRIMENTI

esito = **FALSO**

FINE SE

Scrivi (esito)

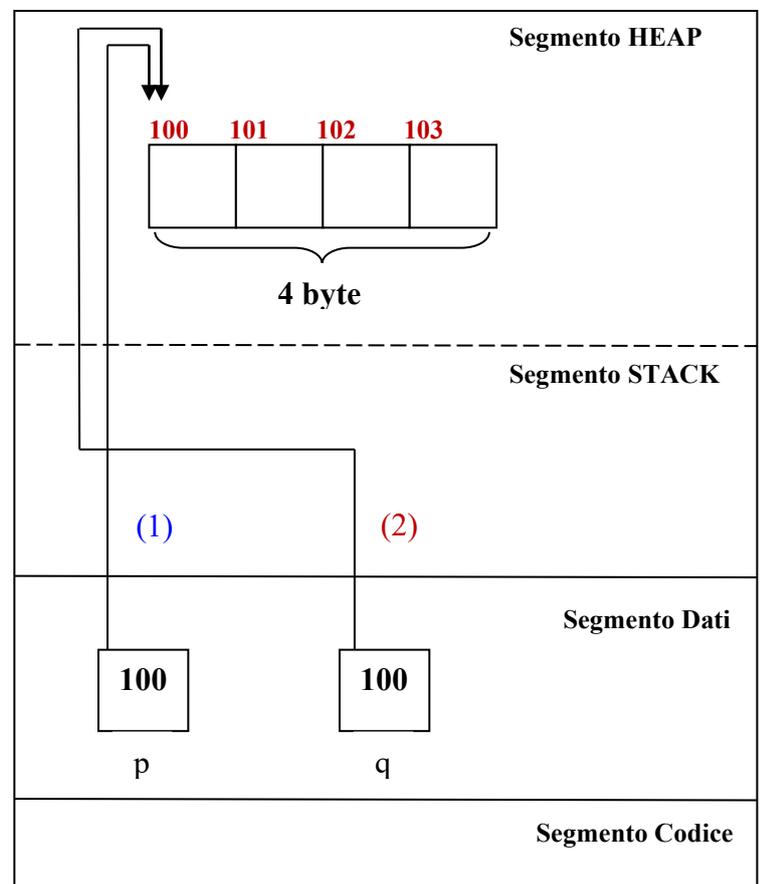
Dealloca (p)

ALTRIMENTI

Scrivi("Allocazione non riuscita!")

FINE SE

FINE



```
// Utilizzando il linguaggio C - Esempio_6_Confronto_Ptr
#include <stdio.h>
#include <stdlib.h>
#define FALSO 0
#define VERO 1
int main(int argc, char *argv[])
{
    int *p, *q;
    int a, b;
    int esito;

    a = 5;
    b = 5;
    p = &a;
    q = &b;
    //Confronto---> se puntatori DIVERSI ossia se puntano ad aree di memoria con indirizzi differenti
    if (p != q)
    {
        esito = VERO;
    }
    else
    {
        esito = FALSO;
    }
    printf("Esito = %d", esito);
    return 0;
}
```

```
// Utilizzando il linguaggio C - Esempio_7_Confronto_Ptr
#include <stdio.h>
#include <stdlib.h>
#define FALSO 0
#define VERO 1
int main(int argc, char *argv[])
{
    int *p, *q;
    int esito;
    p = (int*) malloc (sizeof(int));
    if (p != NULL)
    {
        //ALIASING
        q = p;
        //Confronto ---> se puntatori UGUALI ossia puntano alla stessa area di memoria
        if (p == q)
        {
            esito = VERO;
        }
        else
        {
            esito = FALSO;
        }
        printf("Esito = %d", esito);
        free(p);
    }
    else
    {
        printf("Allocazione non riuscita!");
    }
    return 0;
}
```

ARITMETICA DEI PUNTATORI

L'espressione **aritmetica dei puntatori** si riferisce a un insieme di operazioni aritmetiche applicabili sui valori di tipo puntatore.

Tali operazioni hanno lo scopo di consentire un alto livello di flessibilità nell'accesso a strutture di dati conservati in posizioni contigue della memoria (per esempio **array** ma anche **record**).

L'aritmetica dei puntatori è tipica del linguaggio C ed è stata mantenuta in alcuni linguaggi derivati

Operatore di somma di un puntatore e un intero

Definizione: L'operatore di somma di puntatore e intero richiede un operando di tipo puntatore e un operando di tipo intero.

Il **risultato** di questa **somma** è l'indirizzo dato dal puntatore **incrementato** del risultato della moltiplicazione dell'intero specificato per la dimensione del tipo base del puntatore espressa in byte.

Esempio:

.....

p : PUNTATORE A INT

INIZIO

Alloca (p, 5 * **DimensioneDi**(INT)) //L'area di memoria allocata nello heap può essere vista
 // come un'array monodimensionale formato da 5 interi

FINE

Supponiamo che, dopo la chiamata alla funzione **Alloca()** il puntatore **p** assuma valore 1000 e che la dimensione in byte per rappresentare un **INT** sia pari 4 allora si avrà che:

p vale **1000** (quindi **p** punterà alla locazione di memoria di indirizzo $1000 + 0 * \text{DimensioneDi}(\text{INT}) = 1000$)

p + 1 vale **1004** (quindi **p** punterà alla locazione di memoria di indirizzo $1000 + 1 * \text{DimensioneDi}(\text{INT}) = 1004$)

p + 2 vale **1008** (quindi **p** punterà alla locazione di memoria di indirizzo $1000 + 2 * \text{DimensioneDi}(\text{INT}) = 1008$)

p + 3 vale **1012** (quindi **p** punterà alla locazione di memoria di indirizzo $1000 + 3 * \text{DimensioneDi}(\text{INT}) = 1012$)

p + 4 vale **1016** (quindi **p** punterà alla locazione di memoria di indirizzo $1000 + 4 * \text{DimensioneDi}(\text{INT}) = 1016$).

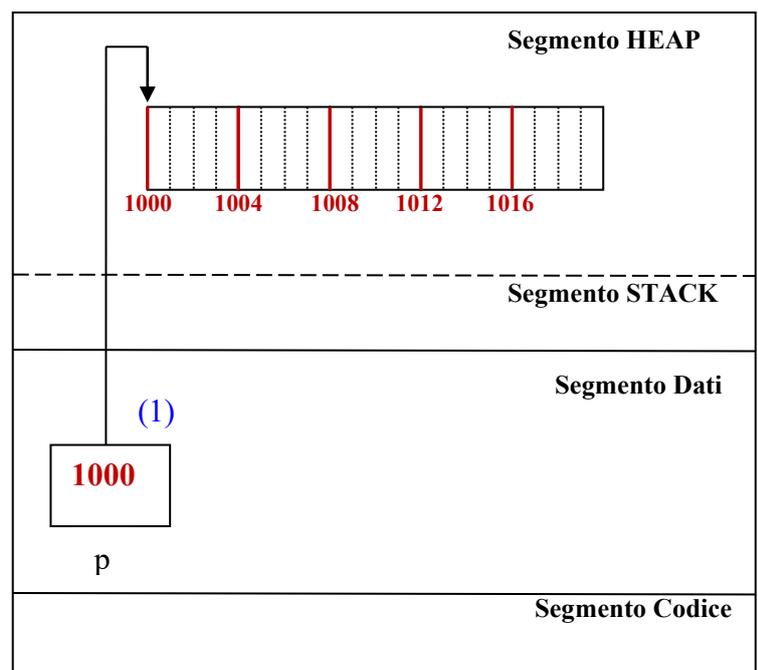
Significato dell'operazione

L'operazione di somma fra puntatore e intero è significativa nel caso in cui il **puntatore** **contenga l'indirizzo di una cella di un array di dati del tipo base del puntatore**.

Infatti, se ad esempio **p** (puntatore a intero) contiene l'indirizzo della prima cella di un **array di interi**, **p+1** produce l'indirizzo della seconda cella dell'array, **p+2** l'indirizzo della terza cella dell'array, e via dicendo fino a **p+(n-1)** che conterrà l'indirizzo della ennesima cella dell'array.

L'operazione di **somma di un intero e di un puntatore** consente di ricavare l'indirizzo di una cella di un array *successiva* a quella puntata dal puntatore su cui viene applicata l'addizione.

L'aritmetica dei puntatori quindi introduce una sintassi alternativa rispetto a quella tradizionale (basata sull'indice) per accedere agli elementi di un array.



ALGORITMO Array_Dinamico

PROCEDURA main ()

p : PUNTATORE A INT

n, i : INT

INIZIO

/ Check sul numero di elementi possibili dell'array dinamico
VERA DINAMICITA': unico limite la quantità di memoria
assegnata al TASK*/*

RIPETI

Leggi (n)

FINCHE' (n ≥ 1)

/ Allocazione area di memoria dinamica */*

Alloca (p, n * DimensioneDi (INT)) (1)

SE (p ≠ NULL)

ALLORA

/ Ciclo di caricamento array dinamico */*

PER i ← 0 A (n - 1) **ESEGUI**

Leggi (*(p + i)) (2)

i ← i + 1

FINE PER

/ Ciclo di visualizzazione array dinamico */*

PER i ← 0 A (n - 1) **ESEGUI**

Scrivi (*(p + i)) (2)

i ← i + 1

FINE PER

/ Deallocazione area di memoria dinamica */*

Dealloca (p) (3)

ALTRIMENTI

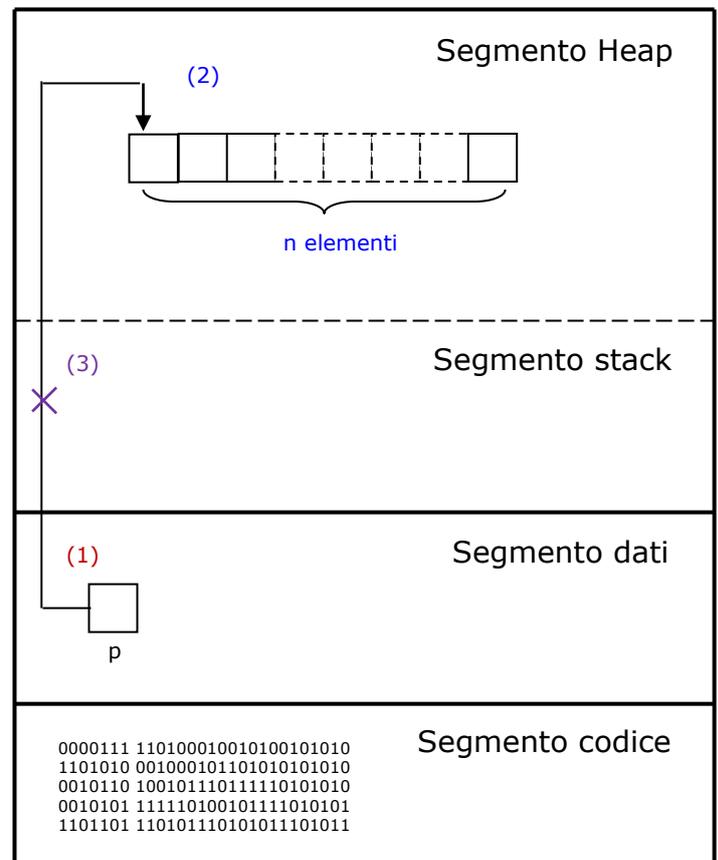
Scrivi ("Errore nell'allocazione")

FINE SE

RITORNA

FINE

Grazie alla somma di un puntatore e di un intero, possiamo finalmente scrivere la **pseudocodifica** di un algoritmo che esegue il **caricamento** e la visualizzazione di un **vettore** o **array monodimensionale** di **n elementi interi ALLOCATO DINAMICAMENTE**



(1) La funzione **Alloca** (...), se terminata con esito positivo, collegherà il puntatore **p** ad un'area di memoria allocata nello heap contenente **n** elementi aventi una lunghezza in byte tale da contenere tutti i dati del tipo previsto dalla funzione **DimensioneDi** (...).

Nel nostro caso quindi **p** punta alla prima locazione di memoria (delle **n** previste) in grado di contenere valori interi

N.B. Come abbiamo già detto la funzione **Alloca** (...) non inizializza in alcun modo i valori contenuti nelle locazioni di memoria fornite nello heap.

(2) Il puntatore **p**, una volta che la funzione **Alloca** (...) ha avuto esito positivo, punterà alla prima locazione di memoria dell'area complessiva assegnata nello heap.

Per poter accedere agli altri elementi è possibile utilizzare **l'aritmetica dei puntatori**.

In particolare, per quanto riguarda sia il caricamento sia la visualizzazione degli elementi dell'array dinamico, sarà possibile accedere ai vari elementi tenendo presente l'operazione somma di un puntatore ed un intero (in caso di iterazioni con indice crescente) oppure l'operazione differenza di un puntatore ed un intero (in caso di iterazioni con indice decrescente)

(3) La funzione **Dealloca** (...) scollegherà il puntatore **p** dall'area di memoria allocata precedentemente nello heap dalla funzione **Alloca** (...) mettendola a disposizione per eventuali altre allocazioni dinamiche (garbage collection)

N.B. Ovviamente la funzione **Dealloca** (...) non ripulisce in alcun modo i valori precedentemente assegnati alle locazioni di memoria fornite nello heap.

Nota Bene

Nel linguaggio C, il NOME di un vettore è un PUNTATORE AL SUO PRIMO ELEMENTO, e quindi, grazie all'aritmetica dei puntatori, se ne ricava che:

$v[i]$ equivale a scrivere $*(v+i)$

INDIPENDENTEMENTE DAL TIPO DI ALLOCAZIONE SCELTA PER L' ARRAY.

Ecco spiegato finalmente il motivo per cui l'indice del PRIMO ELEMENTO di un vettore nel linguaggio C (ed altri linguaggi "fratelli") parte da 0.

```
// Utilizzando il linguaggio C - Array_Dinamico
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[])
{
    int* p;
    int n,i;

    /* Check sul numero di elementi possibili dell'array dinamico */
    /* VERA DINAMICITA': unico limite la quantità di memoria assegnata al TASK*/
    do
    {
        printf("\nInserisci la dimensione dl vettore: ");
        fflush(stdin);
        scanf("%d",&n);
    }
    while (n <= 0);

    /* ALLOCAZIONE DINAMICA dell'array nello heap */
    p = (int*) malloc(n*sizeof(int));
    if(p != NULL)
    {
        printf("\nALLOCAZIONE puntatore OK!\n\n");
        /* Ciclo di caricamento array dinamico */
        printf("CARICAMENTO Vettore dinamico grazie al PUNTATORE AL SUO PRIMO ELEMENTO\n");
        for(i=0; i<n; i++)
        {
            printf("Inserire l'elemento del vettore con indice %d: ", i);
            fflush(stdin);
            scanf("%d", p+i);
        }
        /* Ciclo di visualizzazione array dinamico */
        printf("\nVISUALIZZAZIONE Vettore dinamico grazie al PUNTATORE AL SUO PRIMO ELEMENTO");
        for(i=0; i<n; i++)
        {
            printf("\nIl valore di *(p+%d) che equivale a p[%d] e': %d ", i, i, *(p+i));
        }
        /* Deallocazione area di memoria dinamica */
        printf("\n\nDEALLOCAZIONE puntatore OK!");
        free(p);
    }
    else
    {
        printf("\n*** ERRORE: ALLOCAZIONE puntatore KO!\n\n");
    }

    return 0;
}
```

Operatore di differenza tra un puntatore ed un intero

Definizione: L'operazione di sottrazione di un intero da un puntatore prevede un operando sinistro di tipo puntatore e un operando destro di tipo intero.

Il **risultato** (analogamente al caso della somma) di tale operazione è l'**indirizzo** dato dal puntatore **decrementato** del risultato della moltiplicazione dell'intero specificato per la dimensione del tipo base del puntatore espressa in byte.

Esempio: relativamente allo schema posto a pagina 11, se q è un puntatore al tipo intero INT (q : PUNTATORE A INT) e lo poniamo al valore 1016 e se supponiamo la dimensione di un INT pari quattro byte allora:

q vale **1016** (quindi p punterà alla locazione di memoria di indirizzo $1016 - 0 * DimensioneDi(INT) = 1016$)

$q - 1$ vale **1012** (quindi p punterà alla locazione di memoria di indirizzo $1016 - 1 * DimensioneDi(INT) = 1012$)

$q - 2$ vale **1008** (quindi p punterà alla locazione di memoria di indirizzo $1016 - 2 * DimensioneDi(INT) = 1008$)

$q - 3$ vale **1004** (quindi p punterà alla locazione di memoria di indirizzo $1016 - 3 * DimensioneDi(INT) = 1004$)

$q - 4$ vale **1000** (quindi p punterà alla locazione di memoria di indirizzo $1016 - 4 * DimensioneDi(INT) = 1000$).

Significato dell'operazione: Nelle stesse condizioni descritte sopra per la somma, l'operazione di sottrazione di un intero da un puntatore consente di ricavare l'indirizzo di una cella di un array precedente a quella puntata dal puntatore su cui viene applicata la sottrazione.

N.B. Ovviamente in questo caso il puntatore q deve essere inizializzato con il valore dell'indirizzo dell'ultimo elemento.

ALGORITMO Array_Dinamico_Back

PROCEDURA main ()

p, q : PUNTATORE A INT

n, i : INT

INIZIO

/* VERA DINAMICITA' */

RIPETI

Leggi (n)

FINCHE' ($n \geq 1$)

/* Allocazione area di memoria dinamica */

Alloca ($p, n * DimensioneDi(INT)$) (1)

SE ($p \neq NULL$)

ALLORA

/* q punta all'ultimo elemento dell'array dinamico */

$q \leftarrow p + (n-1)$ (2)

/* Ciclo di caricamento array dinamico con p */

PER $i \leftarrow 0$ A ($n - 1$) ESEGUI

Leggi ($*(p + i)$) (3)

$i \leftarrow i + 1$

FINE PER

/* Ciclo di visualizzazione array dinamico con q */

PER $i \leftarrow 0$ A ($n - 1$) ESEGUI

Scrivi ($*(q - i)$) (4)

$i \leftarrow i + 1$

FINE PER

/* Deallocazione area di memoria dinamica */

Dealloca (p) (5)

$q \leftarrow NULL$

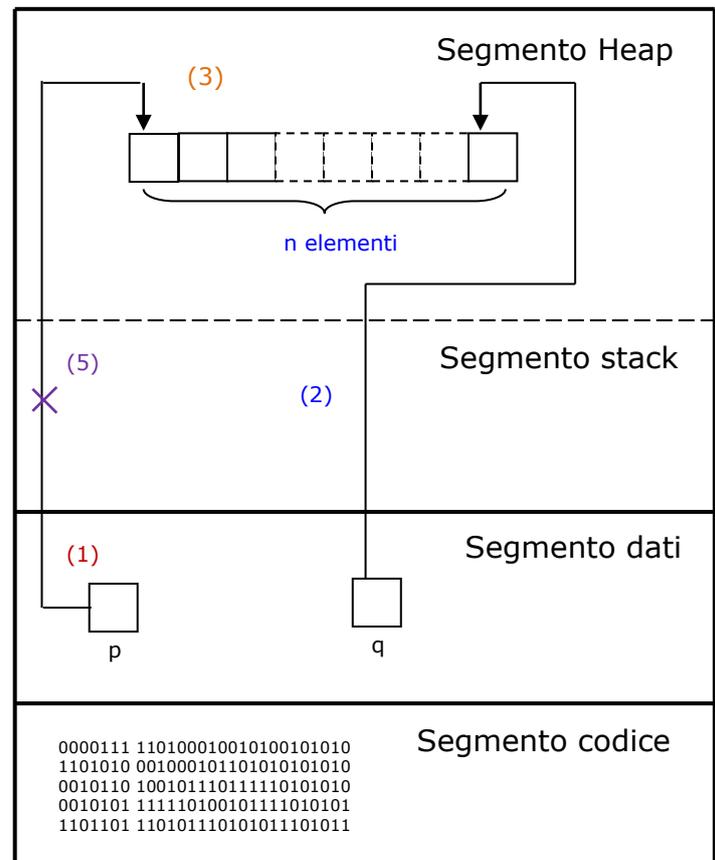
ALTRIMENTI

Scrivi ("Errore nell'allocazione!")

FINE SE

RITORNA

FINE



```

// Utilizzando il linguaggio C - Array_Dinamico_Back
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[])
{
int* p, *q;
int n,i;

/* Check sul numero di elementi possibili dell'array dinamico */
/* VERA DINAMICITA': unico limite la quantità di memoria assegnata al TASK*/
do
{
printf("\nInserisci la dimensione dl vettore: ");
fflush(stdin);
scanf("%d",&n);
}
while (n <= 0);
/* ALLOCAZIONE DINAMICA dell'array nello heap */
p = (int*) malloc(n*sizeof(int));
if(p != NULL)
{
printf("\nALLOCAZIONE puntatore OK!\n\n");

/* il puntatore q punta all'ultimo elemento dell'array dinamico */
q = p + (n-1);
/* Ciclo di caricamento array dinamico cou uso di p PUNTATORE 1° ELEMENTO */
printf("CARICAMENTO Vettore dinamico ATTRAVERSO IL PUNTATORE AL SUO PRIMO ELEMENTO\n");
for(i=0; i<n; i++)
{
printf("Elemento del vettore con indice %d ossia *(p+%d) che equivale a p[%d]: "
, i, i, i);
fflush(stdin);
scanf("%d", p+i);
}
/* Ciclo di visualizzazione array dinamico cou uso di q PUNTATORE ULTIMO ELEMENTO */
printf("\nVISUALIZZAZIONE Vettore dinamico ATTRAVERSO IL PUNTATORE AL SUO ULTIMO ELEMENTO");
for(i=0; i<n; i++)
{
printf("\nIl valore di *(q-%d) che equivale a q[%d] e': %d ",
(n-1)-i, (n-1) -i, *(q-i));
}
/* Deallocazione area di memoria dinamica */
printf("\n\nDEALLOCAZIONE puntatore a vettore OK!");
free(p);
/* Per evitare DANGLING REFERENCE */
q = NULL;
}
else
{
printf("\n*** ERRORE: ALLOCAZIONE puntatore a vettore KO!\n\n");
}
return 0;
}

```

Operatore di differenza fra due puntatori

Definizione: L'operatore di differenza fra puntatori richiede due operandi entrambi di tipo puntatore, aventi tipo base omogeneo (per esempio due puntatori a intero, due puntatori a carattere, e via dicendo).

Il **risultato** della differenza è la differenza aritmetica fra i due indirizzi specificati dai puntatori, divisa per la dimensione del tipo base.

Per esempio, se p contiene l'indirizzo 1008 e q contiene l'indirizzo 1000, ed entrambi sono puntatori al tipo intero e la dimensione di un INT è 4 byte, allora:

$q - p$ vale 2 ossia $(q - p) / DimensioneDi(INT) = (1008 - 1000) / 4 = 8 / 4 = 2$

Significato

L'operazione di differenza fra puntatori è significativa se i due operandi contengono gli indirizzi di due celle diverse del medesimo array, e se il tipo base dell'array coincide con quello dei due puntatori.

In questo caso, infatti, la differenza fra i due puntatori corrisponde al numero di celle (**distanza** o **offset**) dell'array che separano la cella puntata dal puntatore di valore minore da quella del puntatore di valore maggiore.

USO DEI PUNTATORI CON LA STRUTTURA DATI RECORD

Abbiamo già visto in precedenza come creare tipi di dato utente a partire da tipi di dato base grazie alla pseudoistruzione **TIPO** (analoga nel linguaggio C l'istruzione **typedef**).

Utilizziamo quanto appreso finora per realizzare un algoritmo che legga (controllandone i valori) e visualizzi poi a video un record allocato dinamicamente.

ALGORITMO Record_Singolo_Dinamico

MAXNUMCHAR 30

MINNUMMAGLIA 1

MAXNUMMAGLIA 99

TIPO **Calciatore** = **RECORD**

Cognome : **ARRAY**[**MAXNUMCHAR**] **DI CHAR**

Nome : **ARRAY**[**MAXNUMCHAR**] **DI CHAR**

Maglia : **INT**

FINE RECORD

PROCEDURA main()

p : **PUNTATORE A Calciatore**

INIZIO

/ Allocazione record DINAMICA */*

Alloca (**p**, **DimensioneDi** (**Calciatore**))

SE (**p** ≠ **NULL**)

ALLORA

/ Caricamento e controllo campi del record */*

RIPETI

Leggi(**(*p)**.Cognome)

FINCHE' (Lunghezza(**(*p)**.Cognome) ≠ 0) **AND** (Lunghezza(**(*p)**.Cognome) ≤ **MAXNUMCHAR**)

RIPETI

Leggi(**(*p)**.Nome)

FINCHE' (Lunghezza(**(*p)**.Nome) ≠ 0) **AND** (Lunghezza(**(*p)**.Nome) ≤ **MAXNUMCHAR**)

RIPETI

Leggi(**(*p)**.Maglia)

FINCHE' ((**(*p)**.Maglia) ≥ **MINNUMMAGLIA**) **AND** ((**(*p)**.Maglia) ≤ **MAXNUMMAGLIA**)

/ Visualizzazione campi del record */*

Scrivi (**(*p)**.Cognome)

Scrivi (**(*p)**. Nome)

Scrivi (**(*p)**. Maglia)

/ Deallocazione record allocato DINAMICAMENTE */*

Dealloca(**p**)

ALTRIMENTI

Scrivi ("Allocazione non riuscita!")

FINE SE

FINE

N.B. **ATTENZIONE** Del tutto analogo a:
(Lunghezza (**p** → Cognome) ≠ 0) **AND** (Lunghezza (**p** → Cognome) ≤ **MAXNUMCHAR**)



N.B. **ATTENZIONE** Del tutto analogo a:

Scrivi (**p** → Cognome)

Scrivi (**p** → Nome)

Scrivi (**p** → Maglia)

N.B. L'operatore **→** (da non confondere con quello di assegnazione **←**) può essere applicato **ESCLUSIVAMENTE** ai **PUNTATORI AD UN TIPO RECORD**

// Utilizzando il linguaggio C - Record_Singolo_Dinamico

```

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#define MAXNUMCHAR 30
#define MINNUMMAGLIA 1
#define MAXNUMMAGLIA 99
typedef struct
{
    char Cognome[MAXNUMCHAR + 1];
    char Nome[MAXNUMCHAR + 1];
    int Maglia;
} Calciatore;
int main(int argc, char *argv[])
{
    Calciatore* p;

    /* ALLOCAZIONE DINAMICA del record nello heap */
    p = (Calciatore*) malloc(sizeof(Calciatore));
    if(p != NULL)
    {
        /* Caricamento e controllo campi del record allocato dinamicamente*/
        do
        {
            printf("Cognome = ");
            gets((*p).Cognome);
        }
        while ( (strlen((*p).Cognome) == 0) || (strlen((*p).Cognome) > MAXNUMCHAR) );
        do
        {
            printf("Nome = ");
            gets((*p).Nome);
        }
        while ( (strlen((*p).Nome) == 0) || (strlen((*p).Nome) > MAXNUMCHAR) );
        do
        {
            printf("Maglia = ");
            fflush(stdin);
            scanf("%d",&((*p).Maglia) );
        }
        while ( ((*p).Maglia < MINNUMMAGLIA ) || ((*p).Maglia > MAXNUMMAGLIA) );
        /* Visualizzazione del record */
        printf("\nCognome immesso: ");
        puts((*p).Cognome);
        printf("Nome immesso: ");
        puts((*p).Nome);
        printf("Maglia immessa: ");
        printf("%d", (*p).Maglia);
        /* Deallocazione record allocato DINAMICAMENTE */
        free(p);
    }
    else
    {
        printf("\n*** ERRORE: ALLOCAZIONE puntatore a record KO!\n\n");
    }
    return 0;
}

```

Attenzione del tutto analogo a:
gets (p->Cognome);

Attenzione del tutto analogo a:
gets (p->Nome);

Attenzione del tutto analogo a:
scanf ("%d", &(p->Maglia));

Attenzione del tutto analogo a:
puts(p->Cognome);
....
puts (p->Nome);
....
printf ("%d", p->Maglia);

ALGORITMO Array_Record_Dinamico

MAXNUMCHAR 30

MINNUMMAGLIA 1

MAXNUMMAGLIA 99

TIPO **Calciatore** = **RECORD**

Cognome : **ARRAY**[**MAXNUMCHAR**] **DI CHAR**

Nome : **ARRAY**[**MAXNUMCHAR**] **DI CHAR**

Maglia : **INT**

FINE RECORD

PROCEDURA main()

p : **PUNTATORE A Calciatore**

i, n : **INT**;

INIZIO

/ Check sul numero di elementi possibili dell'array dinamico di RECORD */*

/ VERA DINAMICITA': unico limite la quantità di memoria assegnata al TASK*/*

RIPETI

 Leggi (n)

FINCHE' (n ≥ 1)

/ Allocazione DINAMICA ARRAY di record */*

Alloca (p, n * **DimensioneDi** (**Calciatore**))

SE (p ≠ **NULL**)

ALLORA

/ Caricamento e controllo campi dell'ARRAY di record dinamico */*

PER i ← 0 **A** (n-1) **ESEGUI**

RIPETI

 Leggi((p+i) → Cognome)

FINCHE' (Lunghezza((p+i) → Cognome) ≠ 0) **AND** (Lunghezza((p+i) → Cognome) ≤ **MAXNUMCHAR**)

RIPETI

 Leggi((p+i) → Nome)

FINCHE' (Lunghezza((p+i) → Nome) ≠ 0) **AND** (Lunghezza((p+i) → Nome) ≤ **MAXNUMCHAR**)

RIPETI

 Leggi((p+i) → Maglia)

FINCHE' ((p+i) → Maglia) ≥ **MINNUMMAGLIA**) **AND** ((p+i) → Maglia) ≤ **MAXNUMMAGLIA**)

 i ← i + 1

FINE PER

/ Visualizzazione campi dell'ARRAY di record allocato dinamico */*

PER i ← 0 **A** (n-1) **ESEGUI**

 Scrivi((p+i) → Cognome)

 Scrivi((p+i) → Nome)

 Scrivi((p+i) → Maglia)

 i ← i + 1

FINE PER

/ Deallocazione ARRAY di record allocato DINAMICAMENTE */*

Dealloca(p)

ALTRIMENTI

 Scrivi ("Allocazione non riuscita!")

FINE SE

FINE

```
// Utilizzando il linguaggio C - ARRAY_Record_Dinamico
```

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#define MAXNUMCHAR 30
#define MINNUMMAGLIA 99
#define MAXNUMMAGLIA 99
typedef struct
{
    char Cognome[MAXNUMCHAR + 1];
    char Nome[MAXNUMCHAR + 1];
    int Maglia;
} Calciatore;

int main(int argc, char *argv[])
{
    Calciatore* p;
    int i, n;
    /* Check sul numero di elementi possibili dell'array dinamico di RECORD */
    /* VERA DINAMICITA': unico limite la quantità di memoria assegnata al TASK*/
    do
    {
        printf("Inserire dimensione array record dinamico: ");
        fflush(stdin);
        scanf("%d", &n);
    }
    while (n <= 0);
    /* ALLOCAZIONE DINAMICA dell'ARRAY di record nello heap */
    p = (Calciatore*) malloc(n*sizeof(Calciatore));
    if(p != NULL)
    {
        /* Caricamento e controllo campi dell'array di record */
        for(i=0; i<n; i++)
        {
            do
            {
                printf("Cognome = ");
                fflush(stdin);
                gets((p+i)->Cognome);
            }
            while ( (strlen((p+i)->Cognome) == 0) || (strlen((p+i)->Cognome) > MAXNUMCHAR) );
            do
            {
                printf("Nome = ");
                fflush(stdin);
                gets((p+i)->Nome);
            }
            while ( (strlen((p+i)->Nome) == 0) || (strlen((p+i)->Nome) > MAXNUMCHAR) );
            do
            {
                printf("Maglia = ");
                fflush(stdin);
                scanf("%d",&((p+i)->Maglia) );
            }
            while ( ((p+i)->Maglia < MINNUMMAGLIA ) || ((p+i)->Maglia > MAXNUMMAGLIA) );
        }
    }
}
```

```
/* Visualizzazione dell'array di record */
for(i=0; i<n; i++)
{
    printf("\nCognome immesso: ");
    puts((p+i)->Cognome);
    printf("Nome immesso: ");
    puts((p+i)->Nome);
    printf("Maglia immessa: ");
    printf("%d", (p+i)->Maglia);
}

/* Deallocazione ARRAY di record allocato DINAMICAMENTE */
free(p);
}
else
{
    printf("\n*** ERRORE: ALLOCAZIONE puntatore ad ARRAY di record KO!\n\n");
}
return 0;
}
```

STRUTTURE DATI ASTRATTE LINEARI

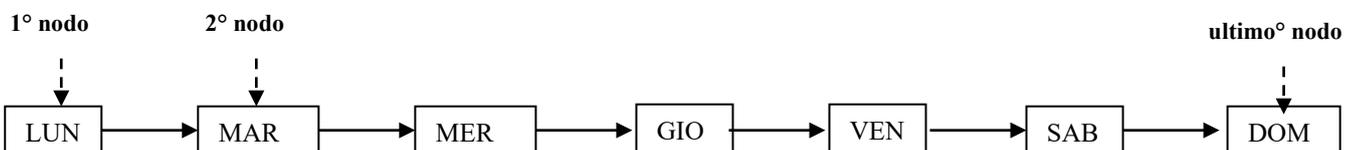
Definiamo **nodo** l'unità di informazione relativa alla struttura dati che stiamo analizzando

SEQUENZA O LISTA

La **sequenza o lista** è uno dei tipi più semplici di struttura dati *astratta* informatica. E' composta da una collezione di nodi tutti dello stesso tipo (*s.d. omogenea*) che sono caratterizzati dal fatto di avere, ad eccezione del primo e dell'ultimo nodo della sequenza, **un unico predecessore ed un unico successore**

(N. B. PER QUESTO MODO SI USA PER TALI STRUTTURE ASTRATTE L'AGGETTIVO LINEARE)

Esempio: consideriamo come nodo il giorno della settimana possiamo rappresentare graficamente la settimana come la sequenza seguente



Dal punto di vista formale **una sequenza o lista** può essere vista come un insieme di **n nodi** P_1, P_2, \dots, P_n dove P_1 è il primo nodo, P_n è l'ultimo nodo ed il generico nodo P_k è preceduto dal nodo P_{k-1} ed è seguito dal nodo P_{k+1}

Il **parametro n** definisce la **lunghezza** della sequenza o lista. Se **n = 0** la sequenza o lista è **vuota**.

Una sequenza o lista può essere:

(-) **a lunghezza fissa**: ossia il numero di nodi che la costituisce non può variare e quindi in tal caso la struttura dati è statica;

(-) **a lunghezza variabile**: ossia il numero di nodi che la costituisce può variare e quindi in tal caso la struttura dati è dinamica.

Sulla struttura dati sequenza o lista **non è consentito** l'accesso diretto come avviene per l'array, ma solo l'accesso sequenziale ossia per accedere ad un determinato nodo della lista occorre accedere prima necessariamente a tutti i nodi che lo precedono.

Per questo motivo su tale tipo di struttura dati sono consentite ricerche solo di tipo sequenziale.

Una premessa rotazionale : indichiamo

con $[]$ una sequenza o lista vuota (con '[' che indica la **testa** e con ']' che indica il **fondo**)

con $[P_1, P_2, \dots, P_n]$ una sequenza formata dai nodi P_1, P_2, \dots, P_n con P_1 in **testa** e P_n in **fondo**

con N l'insieme dei possibili nodi di una sequenza

con S l'insieme di tutte le possibili sequenze di nodi

con \emptyset l'insieme vuoto

con B l'insieme contenente i valori booleani **VERO** e **FALSO**

Nota Bene: In accordo con l'ADT le **operazioni** possibili su questa struttura dati astratta vengono definite come **funzioni matematiche** che a fronte di uno o più valori in ingresso calcolano i relativi valori in uscita.

UNA DOVEROSA PRECISAZIONE.....

In matematica per **funzione** si intende una legge (o relazione) che fa corrispondere ad un valore di un **insieme A di partenza** (detto **DOMINIO**) uno ed un sol valore di un **insieme di B arrivo** (detto **CODOMINIO**) eventualmente coincidente con A

Per rappresentare una funzione in matematica si utilizza principalmente la **NOTAZIONE COMPATTA** ossia **$y = f(x)$** dove:

f è la generica legge di corrispondenza

y è la variabile **dipendente**

$$y = f(x)$$

x la variabile **indipendente**

Esiste anche la **NOTAZIONE ESTESA** di una funzione scritta nel seguente modo:

x la variabile **indipendente**

y è la variabile **dipendente**

$$f : x \in A \longrightarrow y \in B$$

con **$y = f(x)$**

f è la generica legge di corrispondenza

$y = f(x)$ è la **CHIAMATA** della funzione **f**

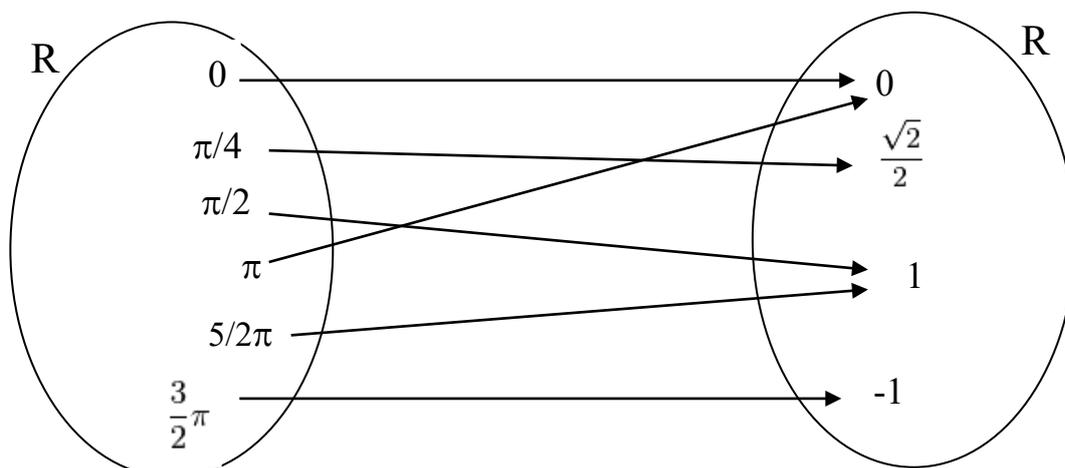
Esempio: prendiamo in esempio al funzione che ad ogni valore **x** reale associi il valore **y** anch'esso reale pari al **seno** di **x** ossia, in altri termini, consideriamo la funzione **$y = \sin(x)$** espressa in **NOTAZIONE COMPATTA**.

Esposta in **NOTAZIONE ESTESA** invece è possibile scrivere:

$$\sin : x \in \mathbb{R} \longrightarrow y \in \mathbb{R}$$

con **$y = \sin(x)$**

Esposta in **NOTAZIONE GRAFICA** o tramite diagramma di **EULERO-VENN** è possibile scrivere per questa funzione:



Secondo l'ADT le principali operazioni per la struttura dati astratta SEQUENZA o LISTA sono:

a) la **creazione** di una nuova sequenza o lista vuota per la quale utilizzeremo la **funzione Crea**

$$\text{Crea} : \emptyset \rightarrow S$$

che provvede a creare la struttura o **lista vuota** [] attraverso la seguente **chiamata**:

$$\text{Crea} () = []$$

b) l' **inserimento** di un nodo:

➤ **in testa** utilizzeremo la **funzione InsTesta**

$$\text{InsTesta} : S \times N \rightarrow S$$

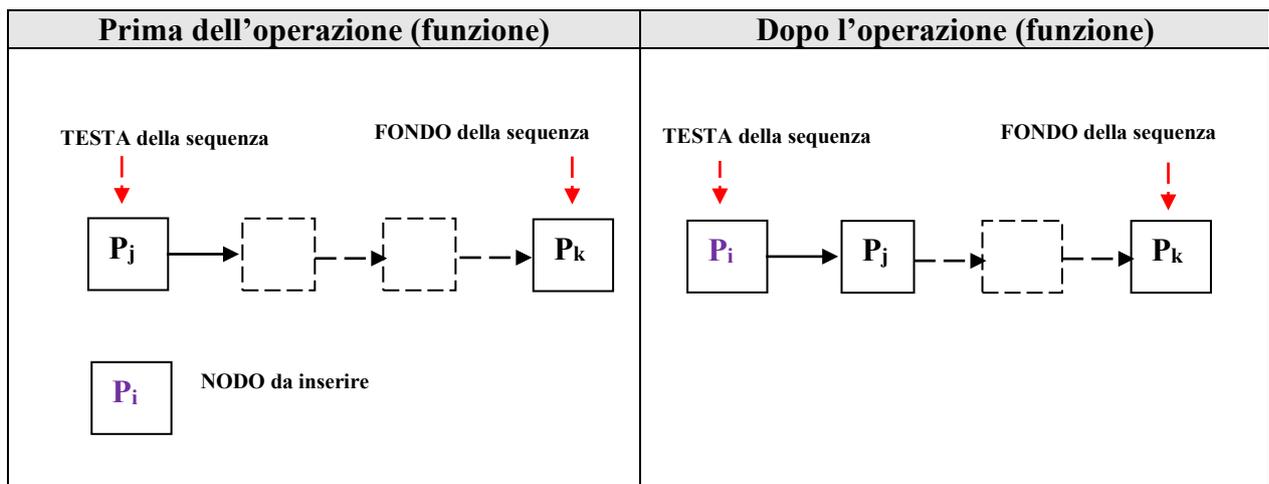
che necessita di due parametri in ingresso: uno identifica la sequenza che stiamo considerando $[P_j, \dots, P_k]$ e l'altro il nodo P_i che vogliamo aggiungere in testa alla sequenza (ossia in prima posizione). La funzione restituirà la nuova sequenza ottenuta

$$[P_i, P_j, \dots, P_k].$$

La generica **chiamata** alla **funzione** sarà:

$$\text{InsTesta} ([P_j, \dots, P_k], P_i) = [P_i, P_j, \dots, P_k].$$

Graficamente



➤ **in fondo** utilizzeremo la **funzione InsFondo**

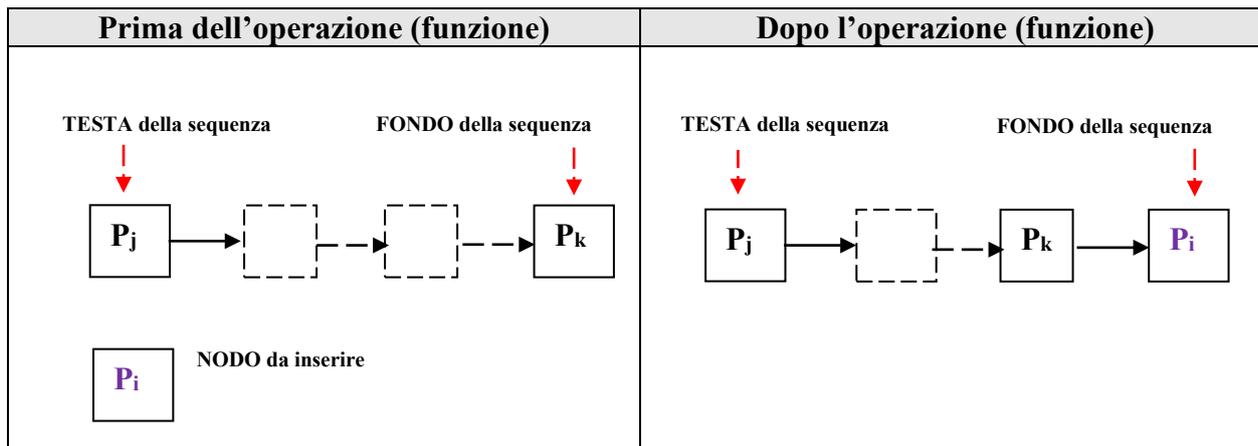
$$\text{InsFondo} : S \times N \rightarrow S$$

che necessita di due parametri in ingresso: uno identifica la sequenza che stiamo considerando $[P_j, \dots, P_k]$ e l'altro il nodo P_i che vogliamo aggiungere in fondo alla sequenza (ossia in ultima posizione). La funzione restituirà la nuova sequenza ottenuta $[P_j, \dots, P_k, P_i]$.

La generica **chiamata** alla **funzione** sarà:

$$\text{InsFondo} ([P_j, \dots, P_k], P_i) = [P_j, \dots, P_k, P_i]$$

Graficamente



➤ in un punto qualsiasi utilizzeremo la funzione *InsPos*

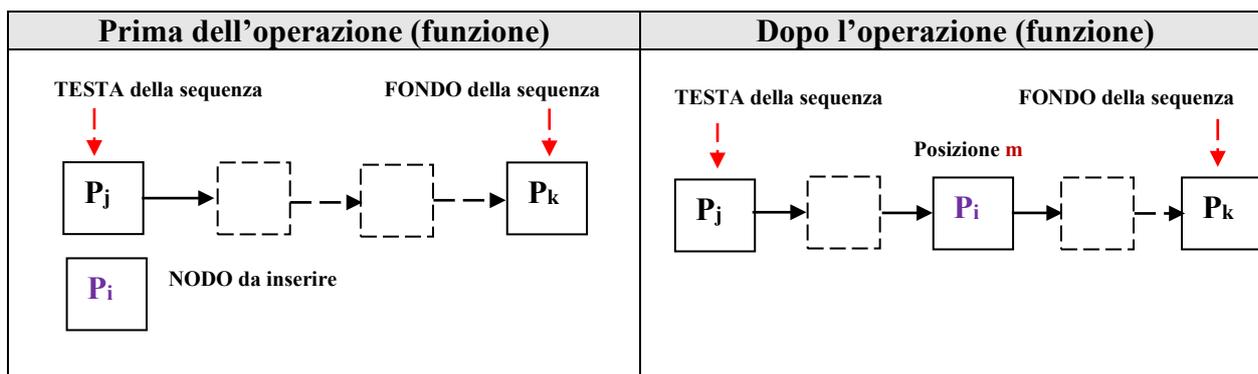
$$InsPos : S \times N \times Z^+ \rightarrow S$$

che necessita di tre parametri in ingresso: uno identifica la sequenza che stiamo considerando $[P_j, \dots, P_k]$, l'altro il nodo P_i che vogliamo aggiungere ed il terzo la posizione m nella quale lo vogliamo inserire (con m diversa dalla prima e dall'ultima per le quali esistono le operazioni apposite). La funzione restituirà la nuova sequenza ottenuta $[P_j, \dots, P_i, \dots, P_k]$.

La generica **chiamata** alla **funzione** sarà:

$$InsPos ([P_j, \dots, P_k] , P_i , m) = [P_j, \dots, P_i, \dots, P_k] \quad (\text{con } P_i \text{ in posizione } m)$$

Graficamente



N.B. Ovviamente tale funzione sarà possibile se la lista ha almeno n nodi con $n \geq 3$ a partire dalla posizione 2 fino alla posizione $n-1$.

Per inserire un nodo nelle posizioni di TESTA e FONDO occorrerà utilizzare le funzioni specifiche introdotte.

c) la **cancellazione** di un nodo:

- **del primo nodo** utilizzeremo la **funzione CancTesta**

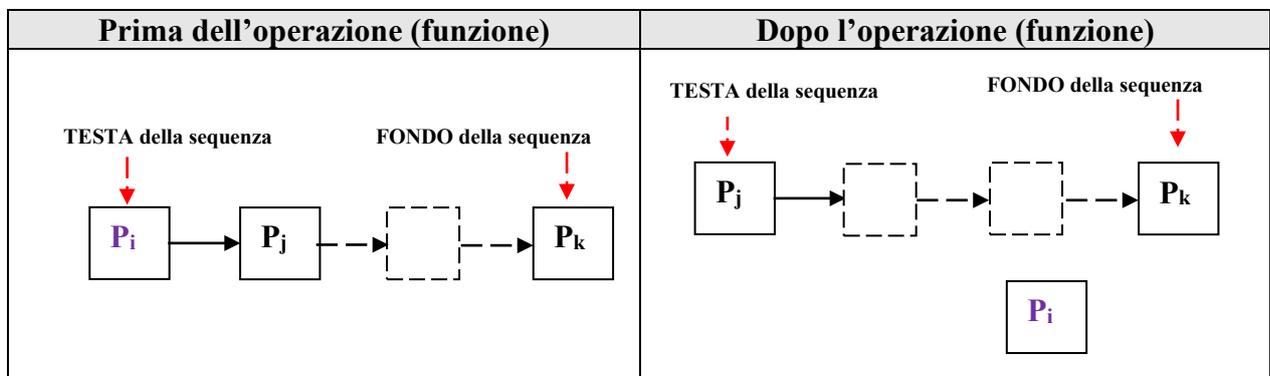
$$\text{CancTesta} : S \rightarrow S \times N$$

che necessita di un solo parametro in ingresso ossia la sequenza che stiamo considerando $[P_i, P_j, \dots, P_k]$. La funzione restituirà la nuova sequenza ottenuta eliminando il nodo ossia $[P_j, \dots, P_k]$ assieme al nodo P_i eliminato dalla testa della sequenza (ossia in prima posizione).

La generica **chiamata** alla **funzione** sarà:

$$\text{CancTesta} ([P_i, P_j, \dots, P_k]) = [P_j, \dots, P_k] , P_i$$

Graficamente



- **dell'ultimo nodo** utilizzeremo la **funzione CancFondo**

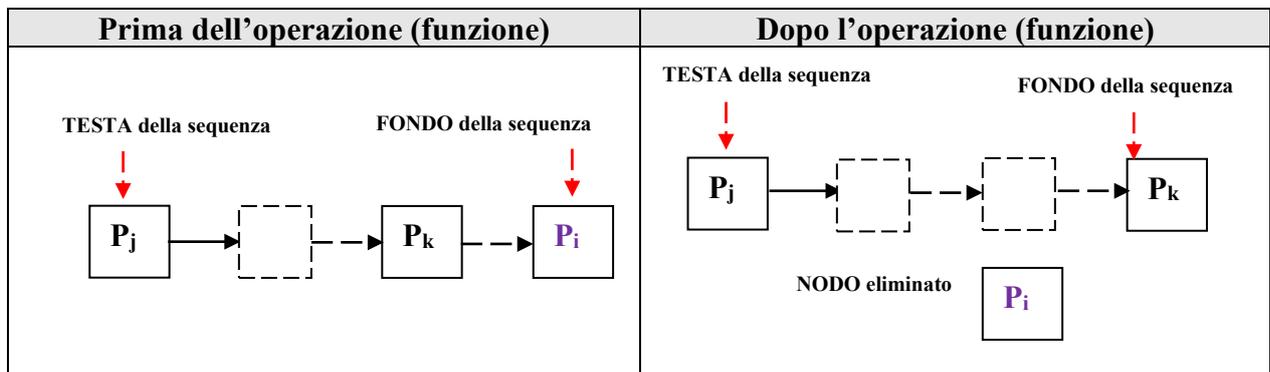
$$\text{CancFondo} : S \rightarrow S \times N$$

che necessita di un solo parametro in ingresso ossia la sequenza che stiamo considerando $[P_j, \dots, P_k, P_i]$. La funzione restituirà la nuova sequenza ottenuta eliminando il nodo ossia $[P_j, \dots, P_k]$, assieme al nodo P_i eliminato dal fondo della sequenza (ossia in ultima posizione).

La generica **chiamata** alla **funzione** sarà:

$$\text{CancFondo} ([P_j, \dots, P_k, P_i]) = [P_j, \dots, P_k] , P_i$$

Graficamente



- di un qualsiasi nodo utilizzeremo la funzione *CancPos*

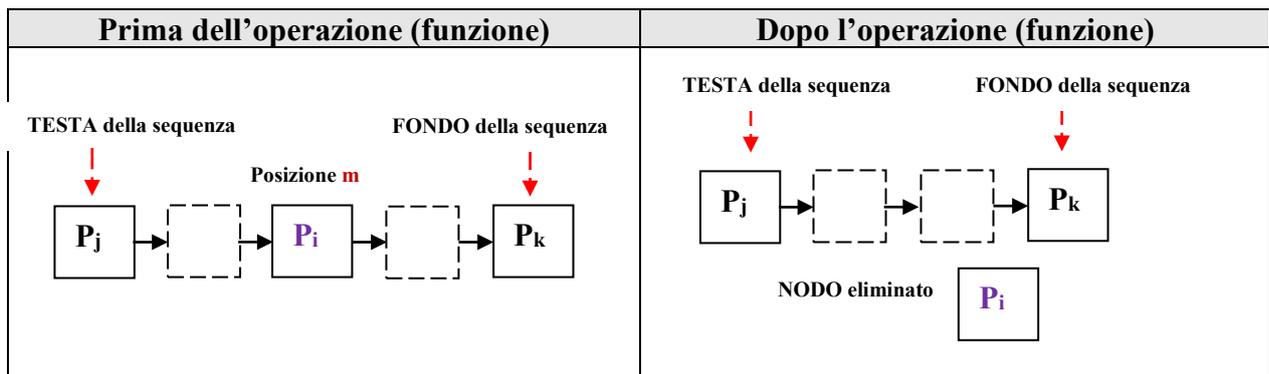
$$\mathbf{CancPos} : S \times Z^+ \rightarrow S \times N$$

che necessita di due parametri in ingresso: uno contenente la sequenza che stiamo considerando $[P_j, \dots, P_i, \dots, P_k]$ ed il secondo contenente la posizione del nodo da eliminare. La funzione restituirà la nuova sequenza ottenuta eliminando il nodo $[P_j, \dots, P_k]$, assieme al nodo P_i eliminato nella posizione m specificata (con m diversa dalla prima e dall'ultima posizione)

La generica **chiamata** alla **funzione** sarà:

$$\mathbf{CancPos} ([P_j, \dots, P_i, \dots, P_k], m) = [P_j, \dots, P_k]_{\cdot, P_i}$$

Graficamente



N.B. Ovviamente tale funzione sarà possibile se la lista ha almeno n nodi con $n \geq 3$ a partire dalla posizione 2 fino alla posizione $n-1$.

N.B. Per cancellare un nodo nelle posizioni di TESTA e FONDO occorrerà utilizzare le funzioni specifiche introdotte e MAI PER NESSUN MOTIVO la funzione *CancPos* ()

- d) il test di sequenza vuota: per il quale utilizzeremo la funzione *TestVuota*

$$\mathbf{TestVuota} : S \rightarrow B$$

che necessita di un solo parametro in ingresso ossia la sequenza che vogliamo controllare essere vuota oppure no. La funzione restituirà:

- il valore booleano **VERO** se la sequenza considerata è **VUOTA**
- il valore booleano **FALSO** se la sequenza considerata è **PIENA**.

La generica **chiamata** alla **funzione** sarà:

$$\mathbf{TestVuota} ([P_j, \dots, P_k]) = \mathbf{FALSO}$$

$$\mathbf{TestVuota} ([]) = \mathbf{VERO}$$

Altre possibili operazioni sulla struttura dati astratta sequenza o lista sono

e) la **ricerca** di un nodo: per la quale utilizzeremo la **funzione *Ricerca***

$$\mathbf{Ricerca} : S \times N \rightarrow B$$

che necessita di due parametri in ingresso: uno contenente la sequenza che stiamo considerando $[P_j, \dots, P_k]$ ed il secondo contenente il nodo P_i da ricercare. La funzione restituirà il valore booleano **VERO** se P_i appartiene alla sequenza considerata oppure il valore booleano **FALSO** se P_i **NON** appartiene alla sequenza considerata.

f) la **lunghezza di una sequenza** per la quale utilizzeremo la **funzione *Lunghezza***

$$\mathbf{Lunghezza} : S \rightarrow Z^+$$

g) l'**ordinamento** dei nodi secondo un certo criterio M per il quale utilizzeremo la **funzione *Ordina***

$$\mathbf{Ordina} : S \times M \rightarrow S$$

h) la **fusione** di 2 sequenza concatenandole per la quale utilizzeremo la **funzione *Fondi***

$$\mathbf{Fondi} : S \times S \rightarrow S$$

Esercizi riepilogativi:

<https://www.riochierego.it/mobile/docs/quarta/lab/STRUTTURE-DATI-ASTRATTE-LINEARI-Esercizi.pdf>

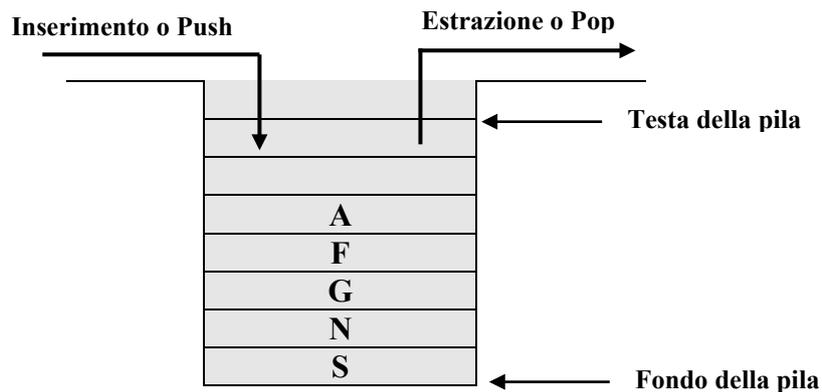
PILA O STACK

La **pila o stack** è una collezione di nodi tutti dello stesso tipo (struttura dati omogenea) dove gli inserimenti (**Push**) e le estrazioni (**Pop**) avvengono sempre a partire da uno stesso estremo detto **testa** (o **top**) della pila.

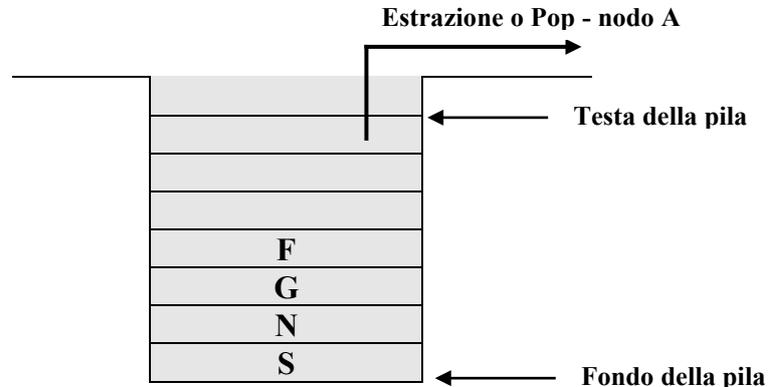
Questo implica che il primo nodo estraibile o prelevabile dalla testa della pila, sia sempre l'ultimo nodo inserito (struttura dati di tipo **L.I.F.O.** ossia **Last In First Out**).

Le principali operazioni possibili su di una pila di nodi sono due:

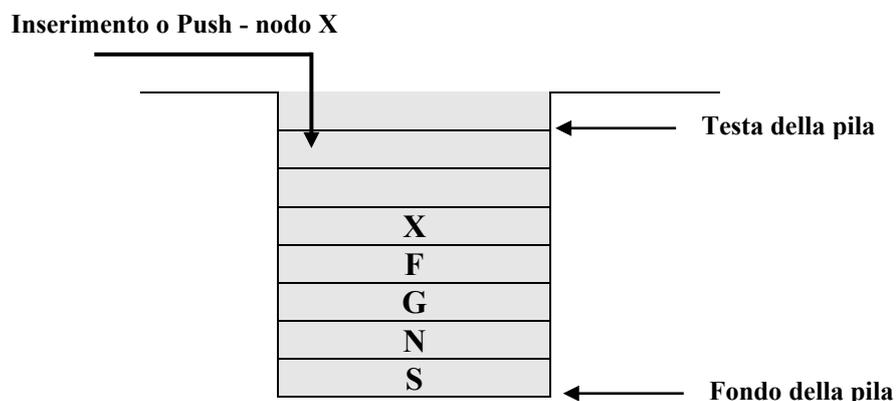
- a) **estrazione o prelevamento** del nodo dalla testa (operazione detta **Pop**);
- b) **inserimento** di un nuovo nodo in testa (operazione detta **Push**).



Nel dettaglio vediamo l'operazione di **Pop** di un nodo dalla testa della pila



Nel dettaglio vediamo l'operazione di **Push** di un nodo **X** in testa alla pila



Una premessa rotazionale : indichiamo

- con $[]$ una pila vuota (con '[' che indica il **fondo** fisso e con ')' che indica la **testa**)
 con $[P_1, P_2, \dots, P_n]$ una pila qualsiasi formata dai nodi P_1, P_2, \dots, P_n con P_1 in fondo e P_n in testa
 con N l'insieme dei possibili nodi di una pila
 con P l'insieme di tutte le possibili pile di nodi
 con \emptyset l'insieme vuoto
 con B l'insieme contenente i valori booleani **VERO** e **FALSO**

In accordo con l'ADT le **operazioni** possibili anche su questa struttura dati astratta vengono definite come **funzioni matematiche** che a fronte di uno o più valori in ingresso calcolano i relativi valori in uscita.

Secondo l'ADT le principali operazioni per la struttura dati astratta **PILA** o **STACK** sono:

a) la **creazione** di una nuova pila vuota per la quale utilizzeremo la **funzione Crea** così definita:

$$\text{Crea} : \emptyset \rightarrow P$$

che provvede a creare la pila vuota $[]$ attraverso la seguente **chiamata**:

$$\text{Crea} () = []$$

b) l'**inserimento (Push)** di un nodo che può avvenire **esclusivamente in testa** per il quale utilizzeremo la **funzione Push**

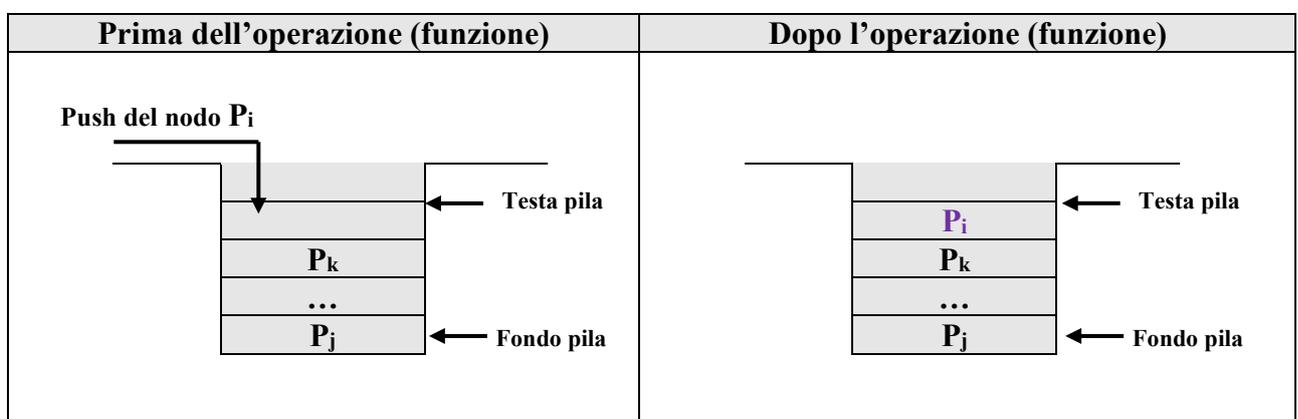
$$\text{Push} : P \times N \rightarrow P$$

che necessita di due parametri in ingresso: uno identifica la pila che stiamo considerando $[P_j, \dots, P_k]$ e l'altro il nodo P_i che vogliamo aggiungere in testa. La funzione restituirà la nuova pila ottenuta $[P_j, \dots, P_k, P_i]$.

La generica **chiamata** alla **funzione** sarà:

$$\text{Push} ([P_j, \dots, P_k], P_i) = [P_j, \dots, P_k, P_i].$$

Graficamente



c) l'estrazione (**Pop**) di un nodo che può avvenire **esclusivamente dalla testa** per la quale utilizzeremo la **funzione Pop**

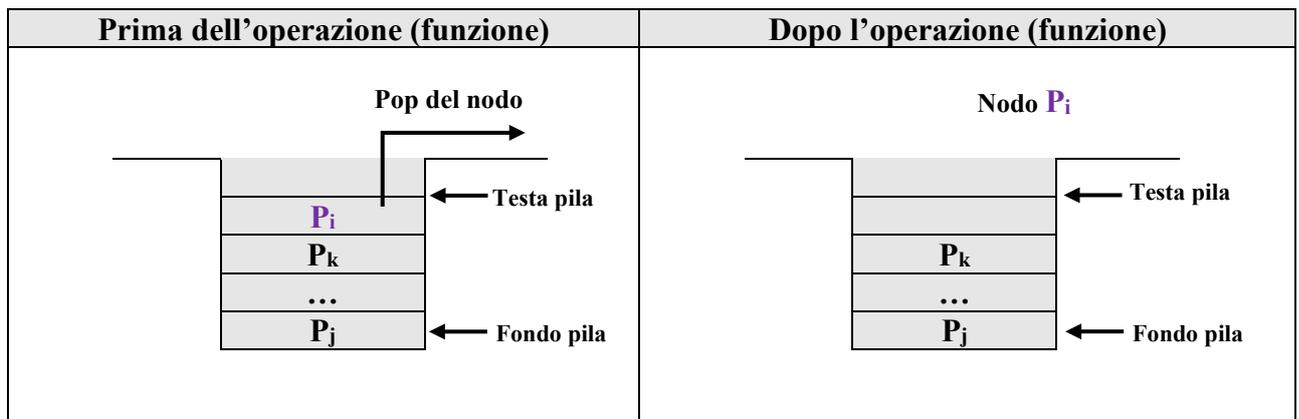
$$\mathit{Pop} : P \rightarrow P \times N$$

che necessita di un solo parametro in ingresso ossia la pila che stiamo considerando $[P_j, \dots, P_k, P_i]$. La funzione restituirà la nuova pila ottenuta $[P_j, \dots, P_k]$ assieme al nodo P_i prelevato dalla testa.

La generica **chiamata** alla **funzione** sarà:

$$\mathit{Pop} ([P_j, \dots, P_k, P_i]) = [P_j, \dots, P_k], P_i \text{ nodo estratto dalla testa}$$

Graficamente



d) il **test di pila vuota** per il quale utilizzeremo la **funzione TestVuota**

$$\mathit{TestVuota} : P \rightarrow B$$

che necessita di un solo parametro in ingresso ossia la pila che vogliamo controllare essere vuota oppure no. La funzione restituirà:

- il valore booleano **VERO** se la **pila considerata è VUOTA**
- il valore booleano **FALSO** se la **pila considerata è PIENA**.

La generica **chiamata** alla **funzione** sarà:

$$\mathit{TestVuota} ([P_i, \dots, P_j]) = \text{FALSO}$$

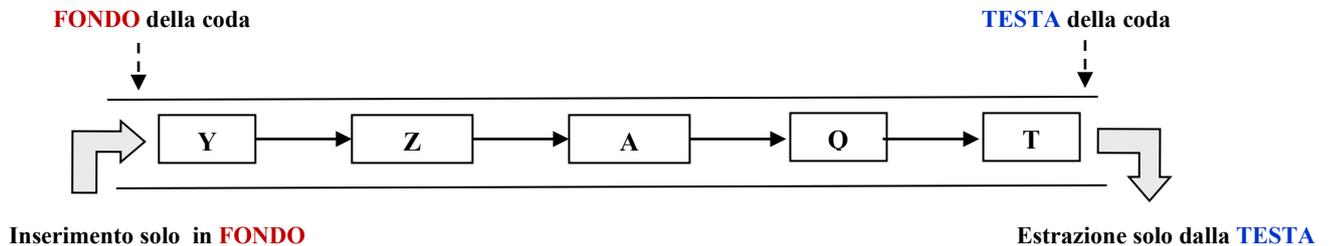
$$\mathit{TestVuota} ([]) = \text{VERO}$$

Esercizi riepilogativi:

<https://www.riochierego.it/mobile/docs/quarta/lab/STRUTTURE-DATI-ASTRATTE-LINEARI-Esercizi.pdf>

CODA O QUEUE

La **codà o queue** è una collezione di nodi tutti dello stesso tipo (struttura dati omogenea) dove le estrazioni avvengono sempre da uno stesso estremo detto **testa della codà** e gli inserimenti avvengono sempre sull'altro estremo detto **fondo della codà**



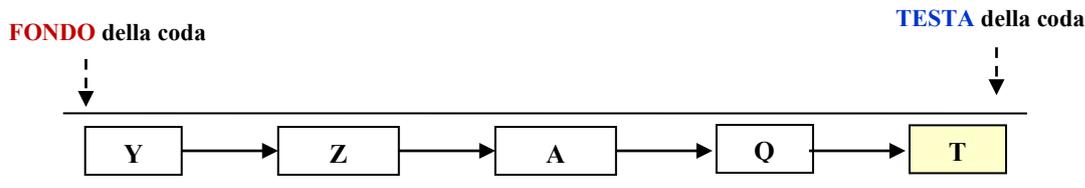
Questo implica che il nodo che può essere estratto da una codà sia sempre il primo nodo inserito nella stessa (struttura dati di tipo **F.I.F.O.**, ossia **F**irst **I**n **F**irst **O**ut).

Le principali operazioni possibili su di una pila di nodi sono due:

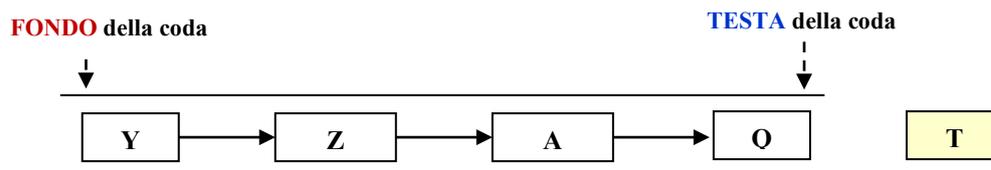
- a) **estrazione o prelevamento** di un nodo **dalla testa** della codà;
- b) **inserimento** di un nuovo nodo **in fondo** alla codà.

Nel dettaglio vediamo l'operazione di **estrazione** di un nodo dalla **TESTA** della codà

PRIMA DELL'ESTRAZIONE

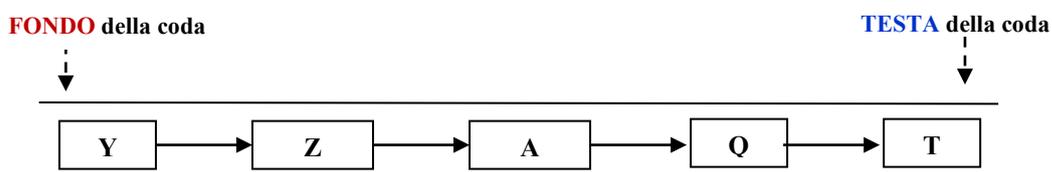


DOPO DELL'ESTRAZIONE

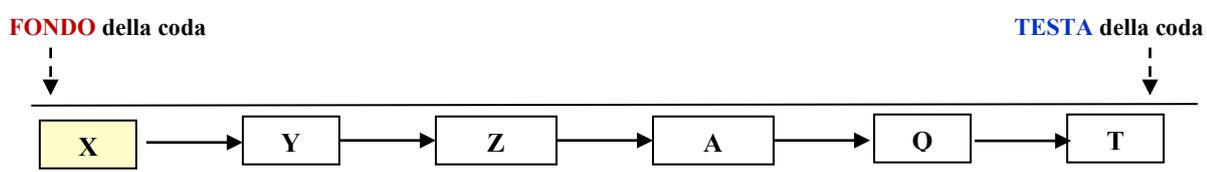


Nel dettaglio vediamo l'operazione di **inserimento** di un nuovo nodo X in **FONDO** alla codà

PRIMA DELL'INSERIMENTO DEL NODO X



DOPO DELL'INSERIMENTO DEL NODO X



Una premessa rotazionale : indichiamo

- con $\{ \}$ una coda vuota (con ‘{’ che indica il **fondo** e con ‘)’ che indica la **testa**)
 con $\{P_1, P_2, \dots, P_n\}$ una coda qualsiasi formata dai nodi P_1, P_2, \dots, P_n con P_1 in fondo e P_n in testa
 con N l’insieme dei possibili nodi di una coda
 con C l’insieme di tutte le possibili code di nodi
 con ϕ l’insieme vuoto
 con B l’insieme contenente i valori booleani **VERO** e **FALSO**

In accordo con l'ADT le **operazioni** possibili anche su questa struttura dati astratta vengono definite come **funzioni matematiche** che a fronte di uno o più valori in ingresso calcolano i relativi valori in uscita.

Secondo l'ADT le principali operazioni per la struttura dati astratta **PILA** o **STACK** sono:

a) la **creazione** di una nuova coda vuota per la quale utilizzeremo la **funzione Crea**

$$\text{Crea} : \phi \rightarrow C$$

che provvede a creare la coda vuota $\{ \}$ attraverso la seguente **chiamata**:

$$\text{Crea} () = \{ \}$$

b) l’**inserimento** di un nodo che può avvenire **esclusivamente in fondo alla coda** e per il quale utilizzeremo la **funzione Inserisci**

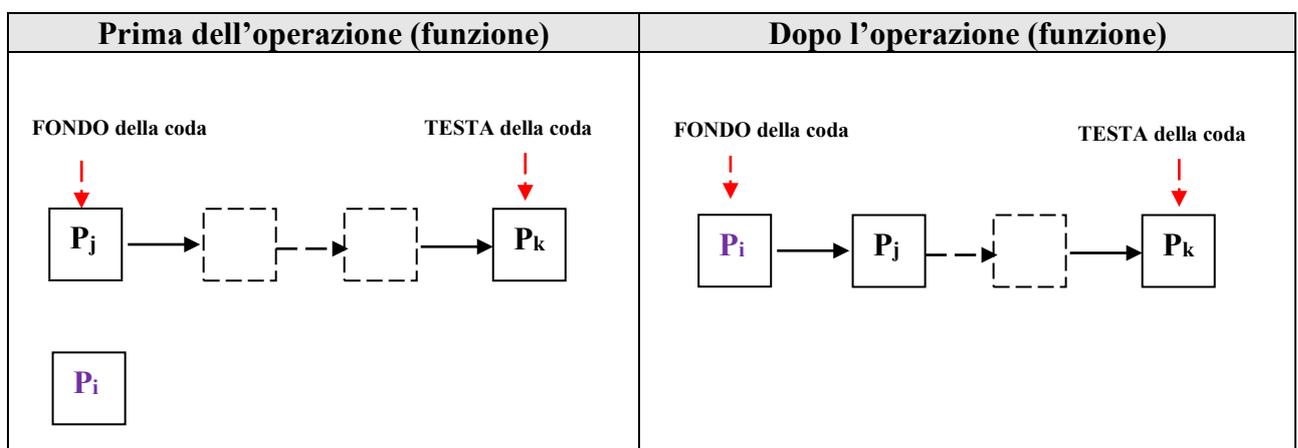
$$\text{Inserisci} : C \times N \rightarrow C$$

che necessita di due parametri in ingresso: uno identifica la coda che stiamo considerando $\{P_j, \dots, P_k\}$ e l’altro il nodo P_i che vogliamo aggiungere in fondo alla coda. La funzione restituirà la nuova coda ottenuta $\{P_i, P_j, \dots, P_k\}$.

La generica **chiamata** alla **funzione** sarà:

$$\text{Inserisci} (\{P_j, \dots, P_k\}, P_i) = \{P_i, P_j, \dots, P_k\}.$$

Graficamente



c) l'**estrazione** di un nodo che può avvenire **esclusivamente in dalla testa della coda** e per il quale utilizzeremo la **funzione Estrai**

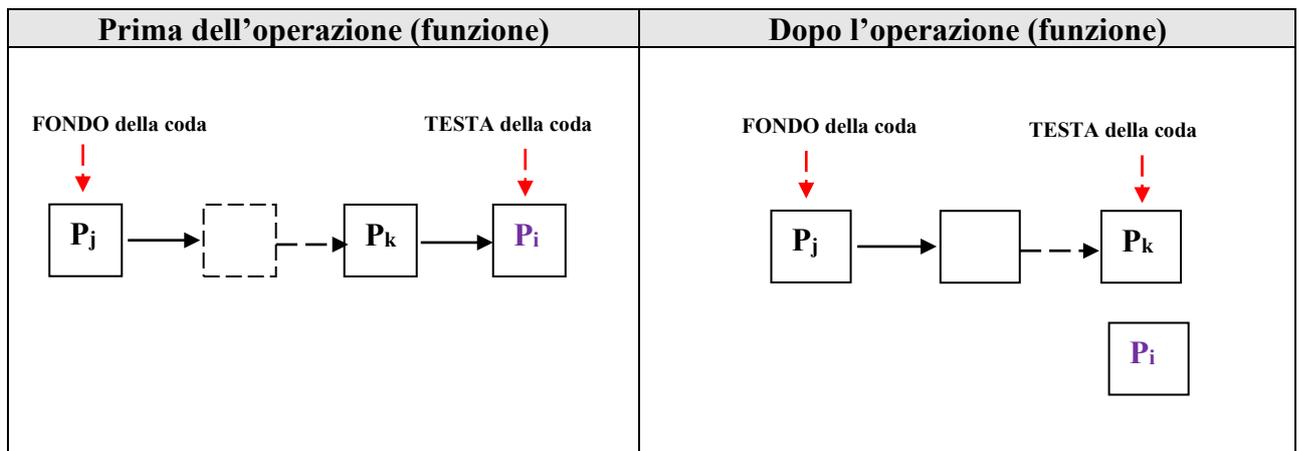
$$\mathbf{Estrai} : C \rightarrow C \times N$$

che necessita di un solo parametro in ingresso ossia la coda che stiamo considerando $\{P_j, \dots, P_k, P_i\}$. La funzione restituirà la nuova coda ottenuta $\{P_j, \dots, P_k\}$ assieme al nodo P_i prelevato dalla testa della coda

La generica **chiamata** alla **funzione** sarà:

$$\mathbf{Estrai} (\{P_j, \dots, P_k, P_i\}) = \{P_j, \dots, P_k\}, P_i \quad \text{nodo estratto dalla testa}$$

Graficamente



d) il **test di coda vuota** per il quale utilizzeremo la **funzione TestVuota**

$$\mathbf{TestVuota} : C \rightarrow B$$

che necessita di un solo parametro in ingresso ossia la coda che vogliamo controllare essere vuota oppure no. La funzione restituirà:

- il valore booleano **VERO** se la **coda considerata è VUOTA**
- il valore booleano **FALSO** se la **coda considerata è PIENA**.

La generica **chiamata** alla **funzione** sarà:

$$\mathbf{TestVuota} (\{P_i, \dots, P_j\}) = \mathbf{FALSO}$$

$$\mathbf{TestVuota} (\{ \}) = \mathbf{VERO}$$

Esercizi riepilogativi:

<https://www.riochierego.it/mobile/docs/quarta/lab/STRUTTURE-DATI-ASTRATTE-LINEARI-Esercizi.pdf>

IMPLEMENTAZIONE DELLE STRUTTURE DATI ASTRATTE LINEARI

A livello di codifica sono possibili due strategie implementative per le strutture dati astratte (ADT) descritte precedentemente:

- attraverso strutture dati **SEQUENZIALI, AD ALLOCAZIONE STATICA ED AD ACCESSO DIRETTO** del linguaggio di programmazione scelto (ossia attraverso **array** o **vettore**);
- attraverso strutture dati **NON SEQUENZIALI, AD ALLOCAZIONE DINAMICA ED AD ACCESSO SEQUENZIALE** del linguaggio di programmazione scelto (ossia attraverso le cosiddette liste linkate);

Limiti dell'utilizzo di strutture dati SEQUENZIALI, AD ALLOCAZIONE STATICA ED AD ACCESSO DIRETTO

L'allocazione statica della memoria presenta senza dubbio molti **VANTAGGI** tra i quali segnaliamo:

- semplicità degli algoritmi** che devono gestire la struttura dati così allocata;
- accesso diretto ai nodi** (poiché vengono utilizzati gli array o vettori).

Nonostante ciò tale metodo di memorizzazione possiede degli **innegabili SVANTAGGI** in quanto è poco flessibile per quanto riguarda:

- occupazione di memoria:** gli array in genere sono sovradimensionati non potendo stimare a priori la dimensione del problema;
- velocità in fase di esecuzione:** gli array sono strutture rigide e la loro gestione richiede a volte tempi che non sono accettabili;
- linearità della soluzione:** talvolta la soluzione offerta dall'allocazione statica della memoria per questo tipo di strutture dati fornisce un risultato negativo sia in relazione alla bontà dell'algoritmo, sia dal punto di vista del rispetto dei criteri generali della programmazione.

VANTAGGI nell'utilizzo di strutture dati NON SEQUENZIALI, AD ALLOCAZIONE DINAMICA ED AD ACCESSO SEQUENZIALE

L'utilizzo di strutture dati **concatenate e dinamiche** dette **LISTE LINKATE** (al posto di quelle **sequenziali, statiche ad accesso diretto**) permettono di gestire con molta più semplicità numerose operazioni di inserimento e di cancellazione.

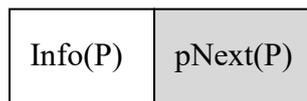
Grazie al loro utilizzo non è più necessario neanche definire a priori la dimensione massima occupata in memoria dalla struttura dati in quanto esse hanno la caratteristica di essere *dinamiche*, ossia di potere modificare la dimensione occupata in memoria nel corso dell'esecuzione del programma che le utilizza.

LA LISTA LINKATA

DEF. La lista (semplicemente) concatenata o linkata è la struttura dati di base ad allocazione dinamica. Essa è formata da una successione di nodi che occupano in memoria posizioni qualsiasi. Ciascun nodo è legato o collegato o linkato al suo successivo mediante un puntatore.

In una lista linkata **ogni nodo P** (quindi tutti i nodi hanno lo stesso formato) possiamo immaginarlo formato da un tipo di dati strutturato (RECORD) costituito da due campi:

- un campo **informativo** (che chiameremo *Info*) che può contenere qualsiasi tipo di dato (semplice o strutturato a sua volta);
- un campo **puntatore** (che chiameremo *pNext*) che contiene l'indirizzo di memoria in cui è possibile reperire il nodo successivo.

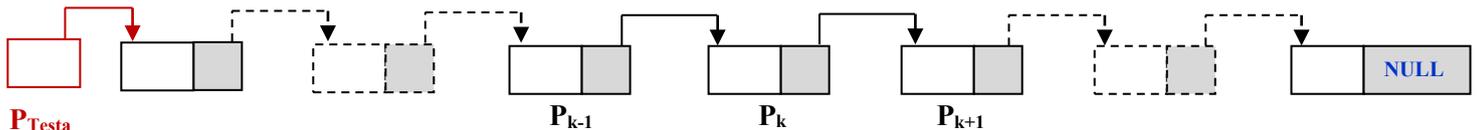
Generico **Nodo P****TIPO Nodo = RECORD**

Info : <Tipo_dato>

pNext : PUNTATORE A **Nodo****FINE RECORD**

con <Tipo_dato> semplice o strutturato a sua volta
 possibile anche se apparentemente sembra un riferimento circolare

Graficamente avremo:



N.B. In una struttura dati concatenata (o LISTA LINKATA) la successione LOGICA dei nodi è arbitraria e non corrisponde in alcun modo a quella FISICA.

Infatti in queste strutture non è assolutamente necessario che la locazione (fisica) di memoria in cui si trova il nodo P_k generico debba essere successiva a quella dove si trova il nodo P_{k-1} oppure debba essere precedente a quella dove si trova il nodo P_{k+1} (come invece accadrebbe in un array).

Il nodo P_k può essere memorizzato dovunque: l'importante è che il nodo P_{k-1} contenga tra le sue informazioni l'indirizzo della locazione di memoria dove si trova il nodo P_k (ossia "punti al nodo" P_k) e che quest'ultimo a sua volta contenga l'indirizzo della locazione di memoria dove si trova il nodo P_{k+1} (ossia "punti al nodo" P_{k+1}).

Questa regola basilare va mantenuta per tutti i nodi della struttura concatenata ad eccezione dell'ultimo nodo della struttura che non deve puntare a nessun nodo in quanto non ha successori.

Con questa regola abbiamo visto che ogni nodo punta al successivo (il primo al secondo, il secondo al terzo...il penultimo all'ultimo) ma **nessuno di essi è in grado di puntare al primo nodo** della struttura.

N.B. L'indirizzo di memoria del primo nodo è infatti una informazione necessaria ma esterna alla struttura dati concatenata.

Esso dovrà essere memorizzato in un apposita variabile che chiameremo **puntatore alla testa** (P_{Testa}) della struttura dati che conterrà il riferimento (ossia il puntatore) al primo nodo ed è "esterno alla lista linkata", mentre l'ultimo nodo, che non ha successori, non dovrà fare riferimento ad alcunchè (ossia "punta a terra" per convenzione avrà il valore **NULL**).

Inoltre con le STRUTTURE DATI CONCATENATE non sarà possibile l'accesso diretto ma quello sequenziale che prevede la scansione di tutta la struttura a partire dal primo nodo fino ad arrivare a quello desiderato.

NOTA BENE

Le LISTE A PUNTATORI sono le strutture dati messe a disposizione dai linguaggi C e C++ per poter implementare le LISTE LINKATE appena descritte, le cui funzionalità devono, al contrario di altri linguaggi di programmazione in cui sono già previste o built-in (vedi PYTHON), essere interamente sviluppate dal programmatore attraverso le apposite funzionalità per la gestione dinamica della memoria (malloc(), calloc(), realloc(), etc.)

STRUTTURE DATI ASTRATTE NON LINEARI

Le strutture dati astratte lineari viste finora (lista, pila e coda) sono caratterizzate dal fatto che **ogni nodo** ha **un solo successore** (tranne l'ultimo nodo) ed **un solo predecessore** (tranne il primo).
Da qui deriva l'aggettivo **LINEARE**.

Vogliamo estendere questi concetti a strutture dati non lineari chiamate *grafi* ed in particolare agli *alberi* che costituiscono le più importanti strutture della nuova classe di strutture da esaminare.

In particolare anticipiamo che nei **grafi** per **ogni nodo** è possibile avere **più nodi predecessori** e **più nodi successori** mentre per gli **alberi** per **ogni nodo** è possibile avere **più successori** ma **un solo predecessore** (tranne il primo).

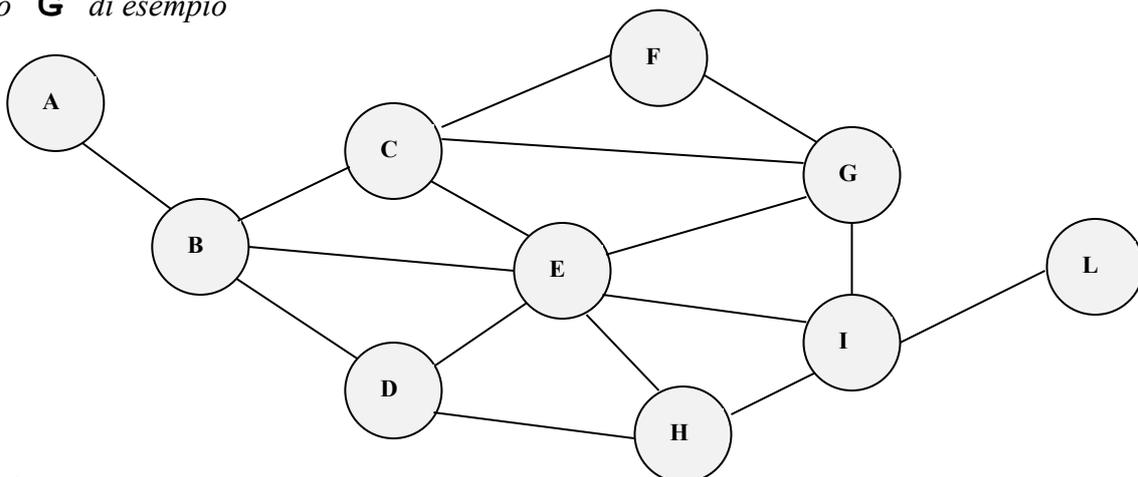
Da qui deriva l'aggettivo **NON LINEARE**.

I GRAFI

DEF. Un **grafo G** è definito da :

- un **insieme** finito e non vuoto **N** di **nodi** detti anche “vertici” o “punti”;
- un **insieme** **A** di **archi** detti anche “segmenti” o “spigoli” o “lati”;
- una **funzione** **F** che descrive le **connessioni** tra le coppie di nodi ossia la “motivazione” per la quale ad ogni arco fa corrispondere una coppia di nodi.

Grafo G di esempio



Notazione:

nodi del grafo G :

A, B, C, D, E, F, G, H, I, L

arco che congiunge i nodi A e B:

(A, B) oppure A, B

DEF. Un nodo di un **grafo G** si dice **DI ORDINE PARI** se è collegato da un numero pari di archi, altrimenti si dice di **ORDINE DISPARI**.

Ad esempio nel nostro grafo G :

i nodi B, I, G sono di ORDINE PARI (esattamente ordine 4) mentre

i nodi A, L sono di ORDINE DISPARI (esattamente ordine 1)

DEF. Due nodi di un **grafo G** si dicono **ADIACENTI** se esiste un arco che li congiunge (altrimenti si dicono **non adiacenti**).

Ad esempio nel nostro grafo G :

I nodi A e B sono ADIACENTI mentre i nodi A e C NON SONO ADIACENTI

DEF. In un grafo G si definisce **CAMMINO** una successione di nodi adiacenti che **non** contiene due volte lo stesso arco.

N.B. Un cammino può invece contenere due volte lo stesso nodo

Ad esempio nel nostro grafo G :

la successione di nodi B, C, E, H, I, E, D è un **cammino** tra i nodi B e D

la successione di nodi B, C, E, G, I, E, D è un altro **cammino** tra i nodi B e D

MA FATE MOLTA ATTENZIONE...è facile farsi ingannare.....

la successione di nodi B, C, E, G, E, D **NON** è un **cammino** tra i nodi B ed D

DEF. In un grafo G si definisce **CAMMINO SEMPLICE** un cammino costituito da tutti nodi distinti ad eccezione eventualmente del primo e dell'ultimo (che possono eventualmente coincidere). Nel caso di cammino semplice con nodo iniziale uguale al nodo finale si parla di **CAMMINO SEMPLICE CICLICO** o più semplicemente **CICLO**.

Ad esempio nel nostro grafo G :

la successione di nodi B, C, G, E, D è un **cammino semplice** tra i nodi B e D

la successione di nodi B, C, F, G, E, D, B è un **cammino semplice ciclico** ossia un **ciclo** di B

N.B. In generale possono esistere diversi cammini semplici che uniscono due stessi nodi

la successione di nodi B, E, D è un **cammino semplice** che unisce i nodi B ed D

la successione di nodi B, E, H, D è un **cammino semplice** che unisce i nodi B ed D

la successione di nodi B, C, E, H, D è un **cammino semplice** che unisce i nodi B ed D

la successione di nodi B, C, E, I, H, D è un **cammino semplice** che unisce i nodi B ed D

DEF. Un grafo G si definisce **connesso** se per ogni coppia di nodi considerata esiste sempre almeno un cammino che li congiunge.

In modo equivalente un grafo G si definisce **connesso** se, considerati due suoi nodi arbitrari, esiste sempre almeno un cammino che li congiunge

DEF. Un grafo G si definisce **orientato** se ogni arco è dotato di orientamento ossia di un verso di percorrenza

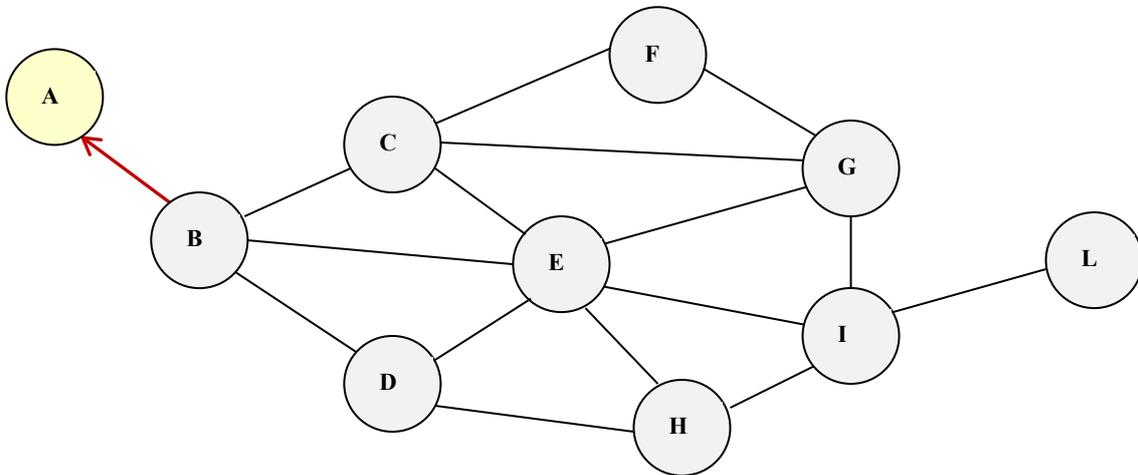
Tale orientamento indica la relazione logica che esiste tra i dati contenuti nei due nodi congiunti dall'arco.

N.B. In tal caso al posto dei segmenti semplici per rappresentare gli archi vengono utilizzate le frecce con uno o due punte considerando in questo tipo di grafo un eventuale segmento semplice del tutto equivalente alla freccia con le due punte.

N.B. Per quanto finora affermato, il grafo G del nostro esempio appare CONNESSO

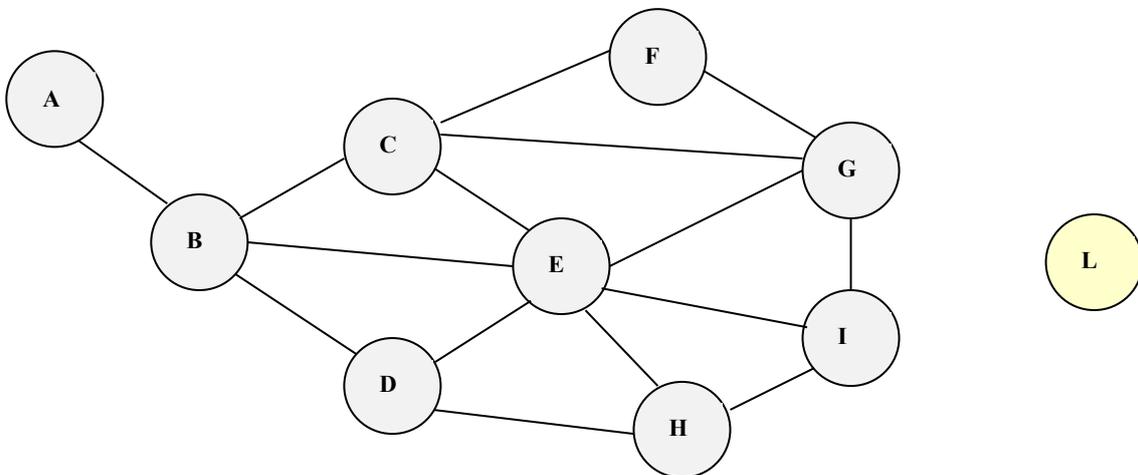
DOMANDA: In che modo è possibile allora rendere NON CONNESSO ossia DISCONNETTERE il nostro grafo G di esempio?

1) Orientando opportunamente uno o più archi del grafo



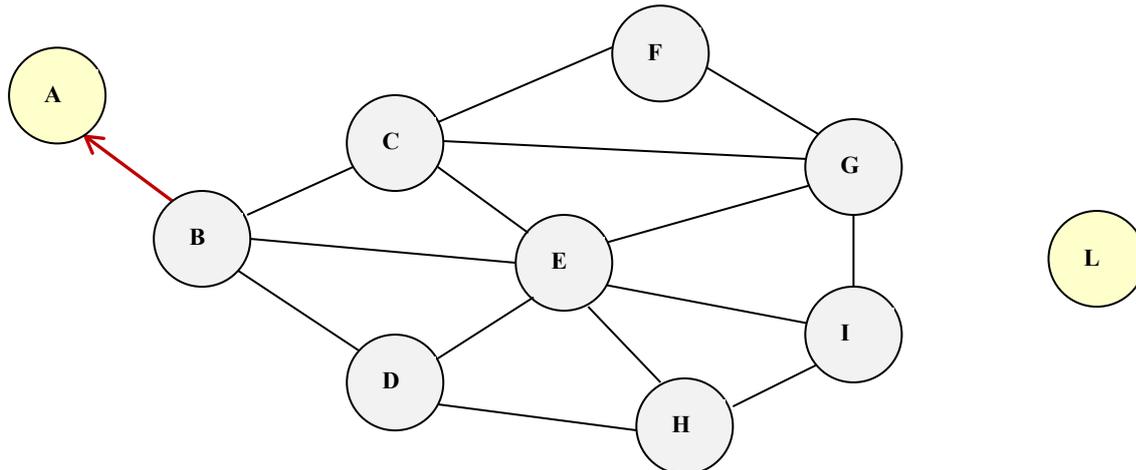
In questo caso poiché dal nodo A non può partire alcun cammino verso uno qualsiasi dei rimanenti nodi del grafo, dovendo rispettare il nuovo orientamento fornito, esisterà ameno una coppia di nodi che non è congiunta da alcun cammino (ad esempio la coppia di nodi A e E).

2) Eliminando opportunamente uno o più archi del grafo



In questo caso poiché il nodo A appare ora completamente isolato non potrà essere raggiunto in alcun modo da nessun cammino, entrante o uscente

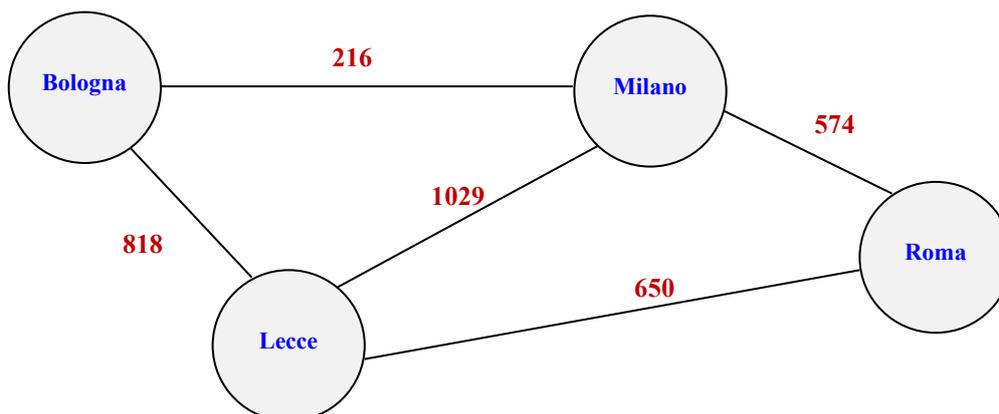
3) Orientando ed eliminando opportunamente uno o più archi del grafo



In questo caso è stato sia orientato opportunamente l'arco che congiunge i nodi B ed A, sia eliminato l'arco che unisce i nodi L ed I.

DEF. Un grafo G si definisce **pesato** se ad suo arco viene associato un valore detto *peso*

Esempio di grafo G connesso e pesato



DOMANDA: A che servono i grafi?

I grafi possono essere visti come modelli in grado di rendere possibile l'astrazione su alcuni aspetti del mondo reale ossia sono in grado di rappresentare un sistema "semplificato" eventualmente presente nella realtà.

*Esempio: Per rappresentare un sistema di trasporti ci si può servire di un grafo G pesato con le località da servire (che possono essere rappresentate come **nod**i), le linee di comunicazione tra le località (che possono essere rappresentate come **archi**) e la distanza chilometrica tra due località (che può essere vista come **peso**).*

Esiste in informatica una classe di problemi legati al **percorso minimo** che unisce tutti i nodi di un grafo G ossia **il cammino semplice con peso minimo** che congiunge tutti i nodi di un grafo G .

Un grafo è particolarmente adatto a rappresentare anche **automi**, **reti di trasporto**, **reti elettriche**, **reti dati**, etc.

APPROFONDIMENTO: I ponti di Königsberg

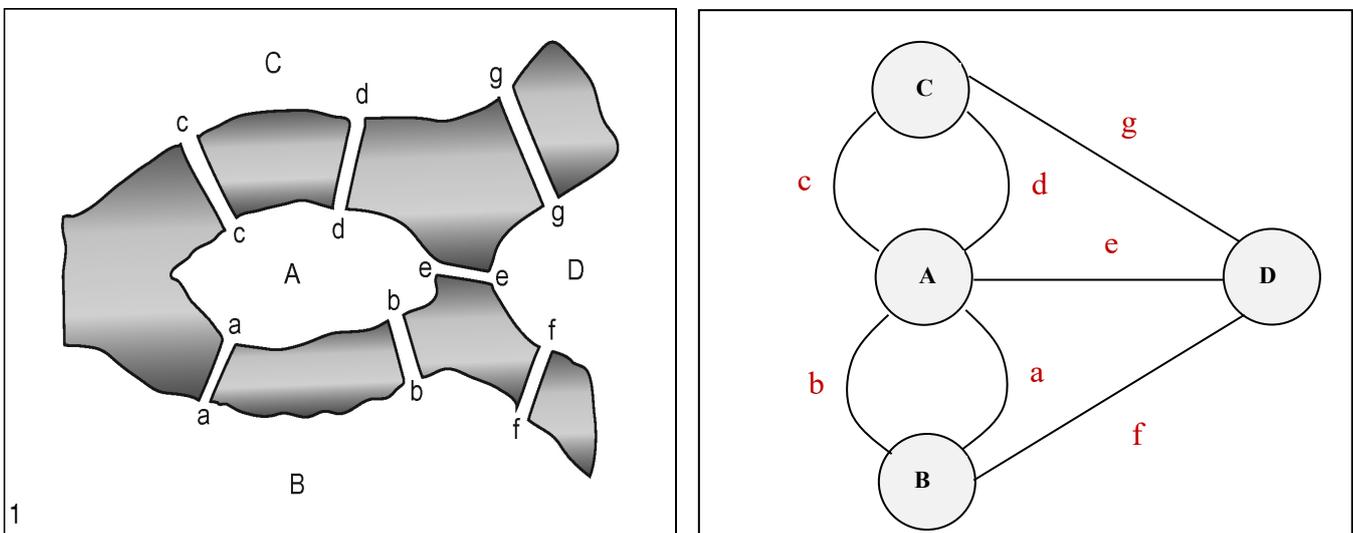
È un problema che la tradizione vuole legato alla città di **Königsberg** della Prussia orientale (oggi **Kaliningrad**, in Russia), nota per aver dato i natali a Immanuel Kant (1724-1804).

Il problema riguarda la possibilità di compiere un percorso (cammino semplice) attraversando tutti i ponti della città e passando per ognuno di essi una (e una sola) volta.

Alla semplicità dell'enunciato, nei termini in cui la tradizione vuole fosse posto dagli abitanti del luogo, non corrisponde una soluzione matematica della questione altrettanto semplice.

In termini moderni, il problema si risolve attraverso le proprietà di percorribilità di un grafo, ma utilizzando prove empiriche, la maggior parte delle persone di quel tempo sembra propendesse per una risposta negativa.

Analizzando il problema dei **ponti di Königsberg** con il formalismo dei grafi, il percorso può essere rappresentato con un grafo con 4 nodi A, B, C, D collegati da 7 archi e aventi tutti ordine dispari, rispettivamente ordine 5, 3, 3, 3.



Fu **Eulero** a fornire la dimostrazione generale del problema nel trattato *Solutio problematis ad geometriam situs pertinentis* del 1741; nella sua soluzione si individua oggi l'origine della moderna teoria dei grafi.

Egli infatti stabilì quindi le seguenti generalizzazioni:

- soltanto un grafo composto da nodi di ordine pari, può essere percorso toccando una e una sola volta tutti i nodi in modo da ritornare infine al punto di partenza; in pratica si effettua un percorso che è un cammino semplice ciclico o ciclo;
- se il grafo contiene tutti nodi di ordine pari ma soltanto due ordine dispari esso è ancora percorribile, ma occorre partire da uno dei due nodi dispari ed arrivare all'altro nodo dispari; in questo caso non è quindi possibile giungere alla fine del percorso allo stesso nodo di partenza;
- se il grafo contiene più di due nodi di ordine dispari risulterà impossibile percorrerlo senza dover attraversare archi già toccati in precedenza.

Per queste ragioni e grazie ad Eulero, il problema dei **ponti di Königsberg** quindi, **non ammette soluzione**.

GLI ALBERI

DEF: Un **ALBERO** \mathcal{A} è un particolare tipo di **GRAFO CONNESSO E SENZA CICLI**

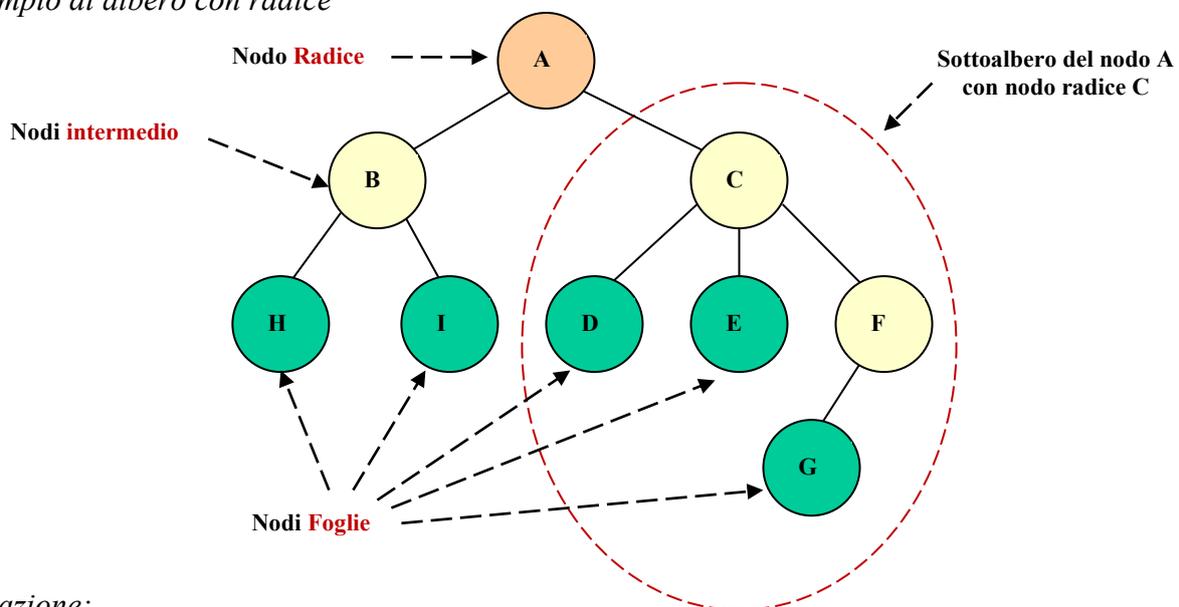
Un **albero** \mathcal{A} gode **SEMPRE** delle seguenti **3 PROPRIETA'**:

1. se un albero possiede **n nodi** allora conterrà **n-1 archi** (N.B. **non è vero il viceversa**);
2. due nodi qualsiasi in un albero sono sempre connessi da un **unico cammino semplice**;
3. **rimuovendo un qualsiasi arco** dell'albero, la struttura dati risultante
 - **non è più connessa**
 - **rimane divisa** in due strutture dati astratte non lineari che risultano essere anch'esse alberi

I tipi più utili di alberi usati in informatica sono gli **alberi con radice** ossia quegli alberi ai quali ad un nodo detto “**radice**” viene attribuito un significato speciale.

In un albero **ogni nodo** può essere considerato **radice del sottoalbero** che da esso trae origine. I nodi dai quali non vengono originati sottoalberi, vengono chiamate **foglie**.

Esempio di albero con radice



Notazione:

radice dell'albero:

A

foglie dell'albero:

H, I, D, E, G

nodi intermedi dell'albero:

B, C, F

Gli alberi che considereremo saranno **alberi ordinati** (nei quali si possono distinguere gli elementi primo, secondo, ... n-esimo secondo un determinato criterio) alberi per i quali l'ordinamento dei sottoalberi è importante.

DEF. In un **albero** \mathcal{A} si dice **grado di un nodo** il numero dei sottoalberi di quel nodo.

DEF. In un **albero** \mathcal{A} si dice **livello di un nodo** il numero di nodi attraversati da un cammino semplice dalla radice al nodo. Se il nodo è **la radice** dell'albero, il livello è uguale **ad 1**.

DEF. Si dice **altezza o profondità h** di un **albero** \mathcal{A} la lunghezza (ossia il numero di nodi) del cammino semplice più lungo esistente tra nodo radice e nodi foglie, escludendo dal conteggio il nodo radice.

N.B. Esiste dunque una relazione matematica tra numero di livelli ed altezza di un albero

$$\text{Altezza di un albero} = \text{Numero di livelli} - 1$$

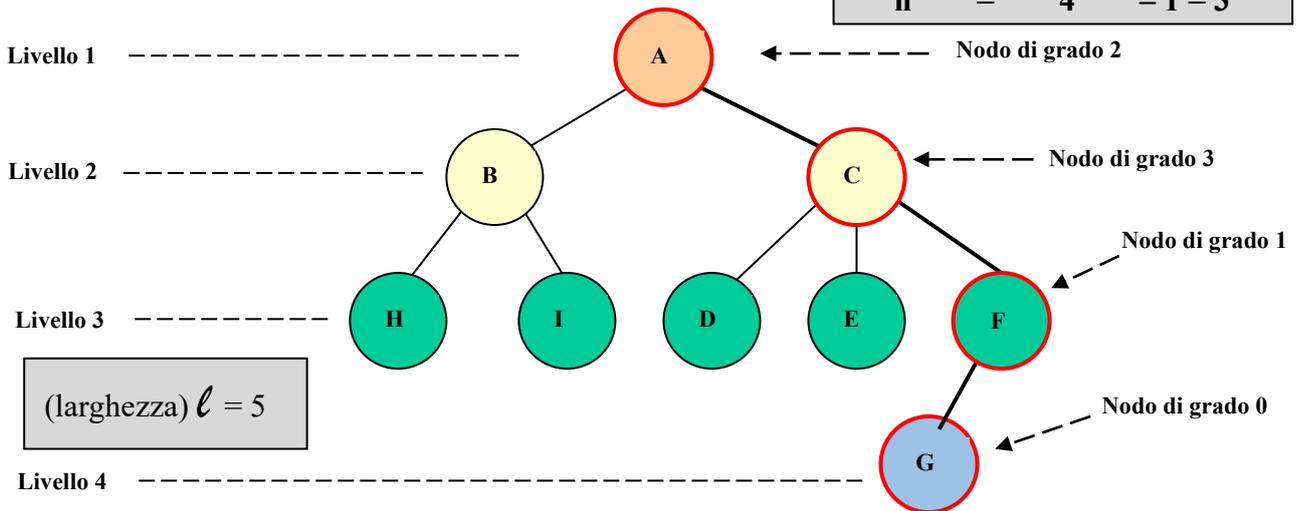
ossia

$$\text{Numero di livelli} = \text{Altezza di un albero} + 1$$

DEF. Si dice **larghezza o ampiezza** ℓ di un albero \mathcal{A} il massimo numero di nodi dell'albero che si trovano allo stesso livello

Applichiamo quanto detto ad un albero con radice di esempio:

$$\begin{aligned} (\text{altezza}) &= (\text{numero livelli}) - 1 \\ h &= 4 - 1 = 3 \end{aligned}$$



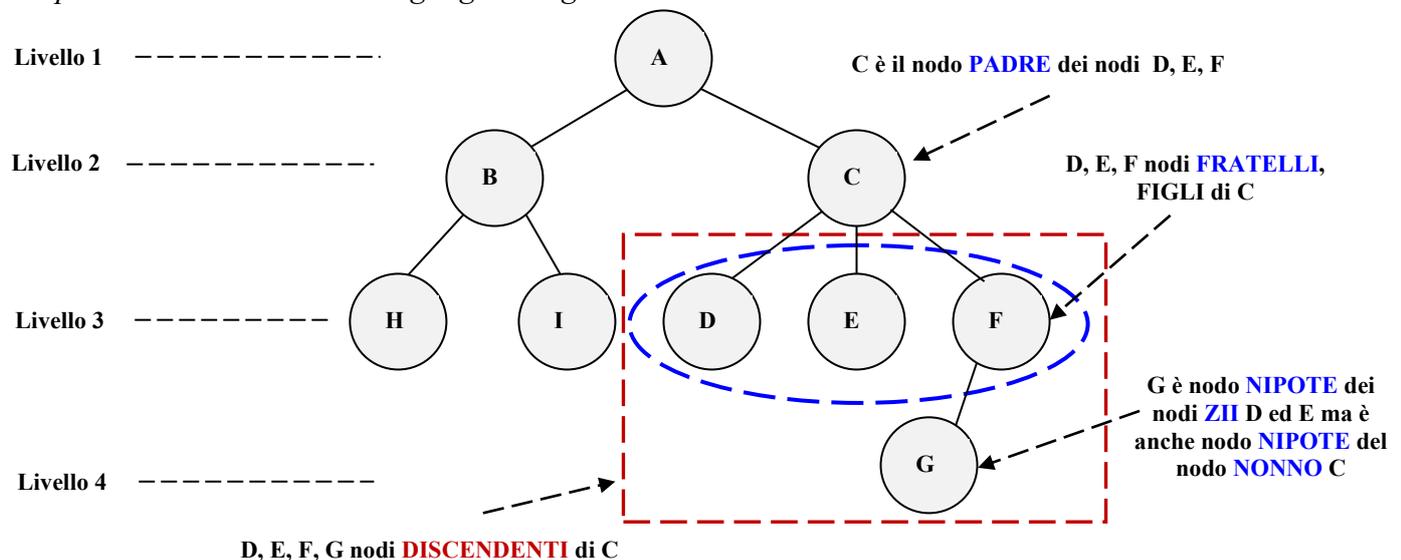
N.B. E' possibile parlare di alberi in termini "genealogici".

Ciascuna radice è detta **padre** delle radici dei suoi sottoalberi che a loro volta sono **figli** del padre. Le radici dei sottoalberi dello stesso padre si dicono **fratelli**.

Questa terminologia potrebbe essere anche estesa a **nonno**, **zio**, **cugino** (ma anche a madre-figlia-sorella e nonna-zia-cugina)

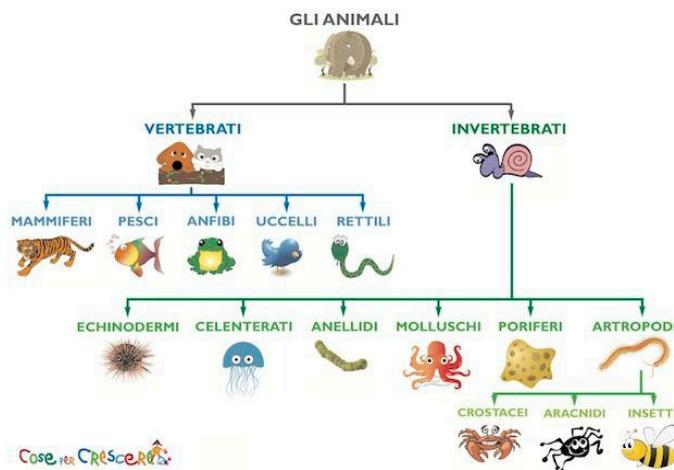
DEF. In un albero si dicono **discendenti di un nodo** quei nodi che appartengono ad un sottoalbero che ha quel nodo come radice.

Esempio di albero con terminologia genealogica

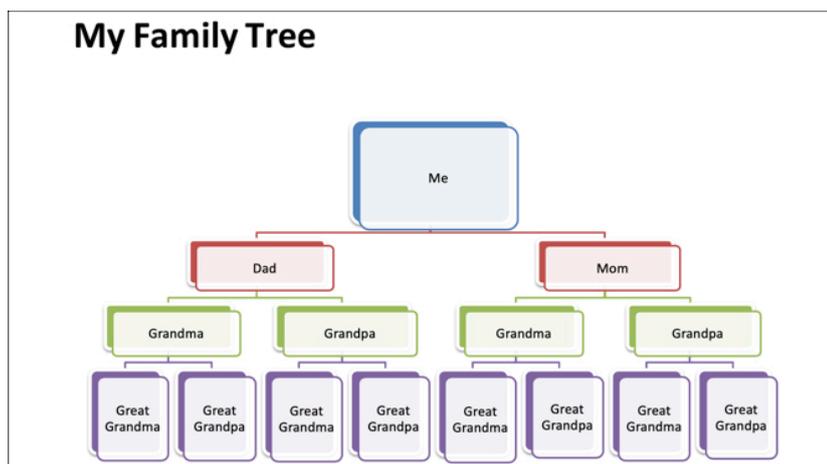


Da quanto detto finora è evidente che un albero è una struttura particolarmente adatta alla descrizione delle informazioni tra le quali è possibile stabilire una **gerarchia o una classificazione**.

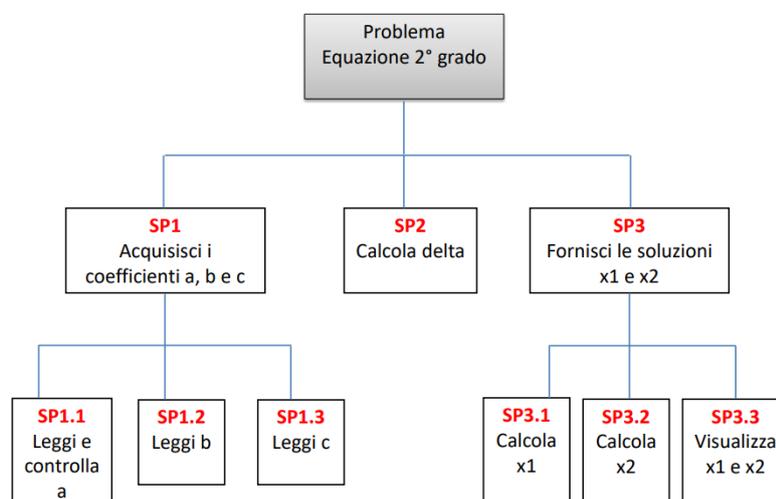
Esempio 1 - classificazione di animali e piante:



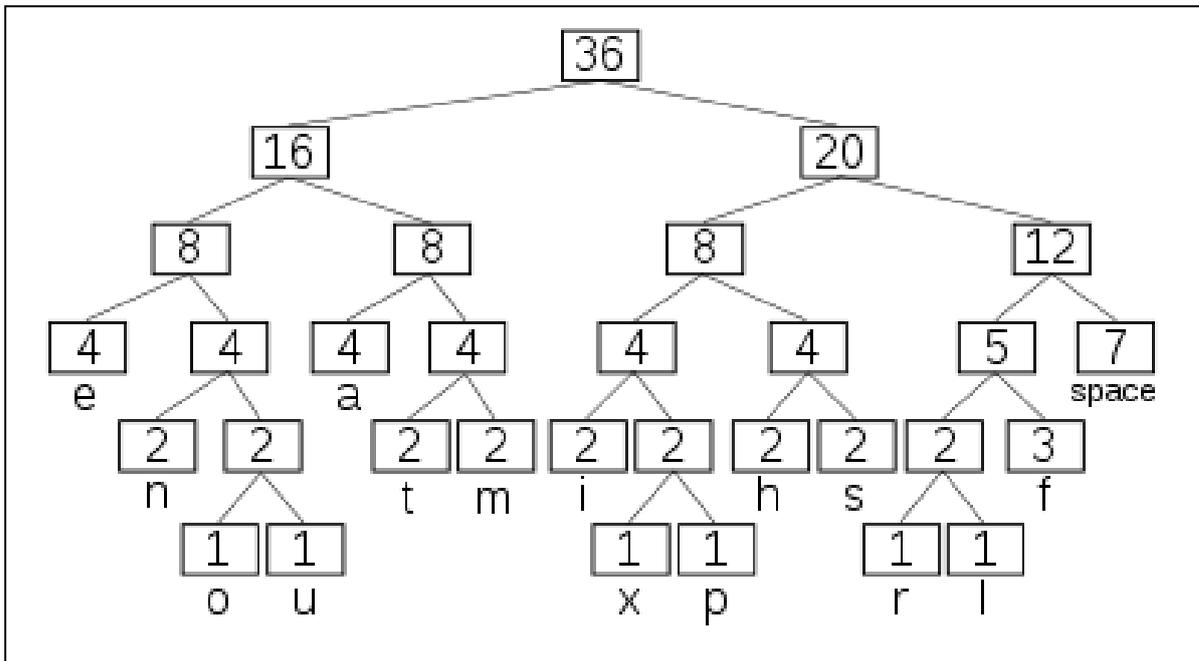
Esempio 2 - albero genealogico familiare



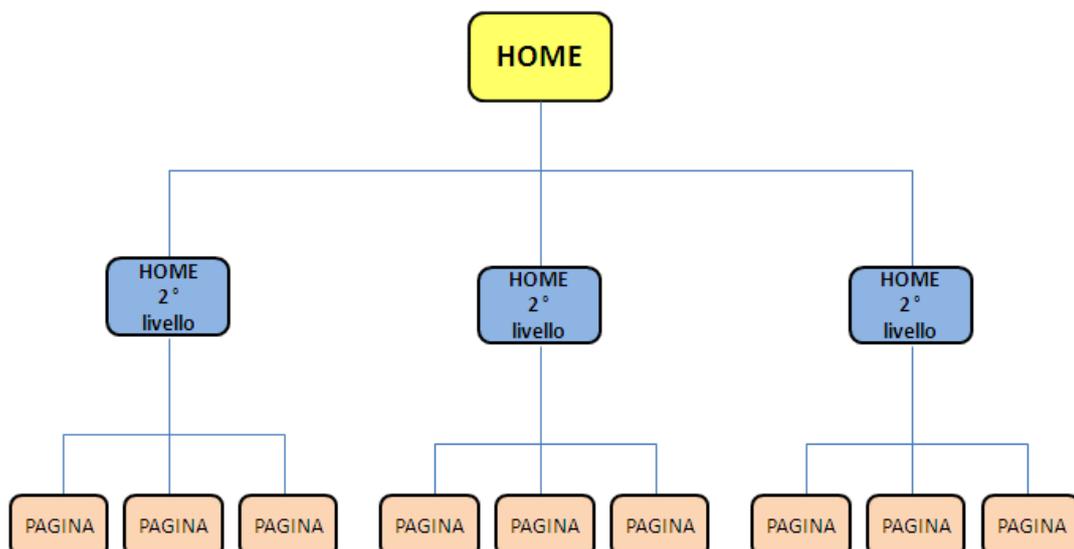
Esempio 3 - meccanismo di funzionamento della metodologia top-down che porta alla determinazione della soluzione di un problema complesso attraverso la scomposizione in sottoproblemi via via sempre più semplici



Esempio 4 - Codifica di Huffman si intende un algoritmo di codifica dei simboli usato per la compressione di dati, basato sul principio di trovare il sistema ottimale per codificare stringhe basato sulla frequenza relativa di ciascun carattere



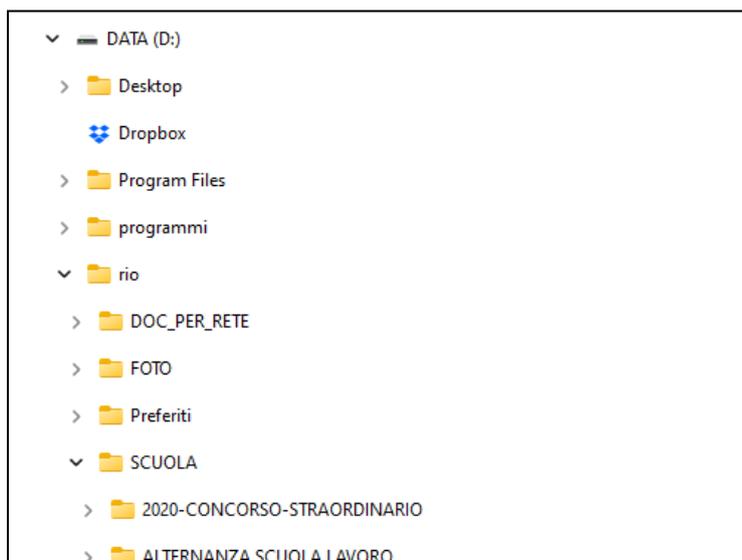
Esempio 5 - struttura di un sito web



Esempio 6 – file system di un sistema operativo (qui esito del comando "tree" di UNIX)

```
[tecmint@centos8 opt]$ sudo tree
.
├── apache-tomcat-9.0.26
│   ├── bin
│   │   ├── bootstrap.jar
│   │   ├── catalina.bat
│   │   ├── catalina.sh
│   │   ├── catalina-tasks.xml
│   │   ├── ciphers.bat
│   │   ├── ciphers.sh
│   │   ├── commons-daemon.jar
│   │   ├── commons-daemon-native.tar.gz
│   │   ├── configtest.bat
│   │   ├── configtest.sh
│   │   ├── daemon.sh
│   │   ├── digest.bat
│   │   ├── digest.sh
│   │   ├── makebase.bat
│   │   ├── makebase.sh
│   │   ├── setclasspath.bat
│   │   ├── setclasspath.sh
│   │   ├── shutdown.bat
│   │   ├── shutdown.sh
│   │   ├── startup.bat
│   │   ├── startup.sh
│   │   ├── tomcat-juli.jar
│   │   ├── tomcat-native.tar.gz
│   │   ├── tool-wrapper.bat
│   │   ├── tool-wrapper.sh
│   │   ├── version.bat
│   │   └── version.sh
│   ├── BUILDING.txt
│   ├── conf
│   │   ├── catalina.policy
│   │   ├── catalina.properties
│   │   ├── context.xml
│   │   ├── jaspic-providers.xml
│   │   ├── jaspic-providers.xsd
│   │   ├── logging.properties
│   │   ├── server.xml
│   │   ├── tomcat-users.xml
│   │   ├── tomcat-users.xsd
│   │   └── web.xml
│   ├── CONTRIBUTING.md
│   └── lib
│       ├── annotations-api.jar
│       ├── catalina-ant.jar
│       └── catalina-ha.jar
```

Qui attivazione risorse del computer (WINDOWS)



Dopo avere introdotto tutti i concetti relativi ad un albero possiamo darne, comprendendone il significato, la seguente definizione ricorsiva:

DEF. 2 (RICORSIVA) Un **albero** è un insieme non vuoto **T** di nodi dove:

- a) esiste un solo nodo distinto detto **radice**;
- b) i rimanenti nodi sono ripartiti in **n insiemi tutti disgiunti** T_1, T_2, \dots, T_n con $n \geq 0$ ciascuno dei quali è a sua volta un albero

Se l'insieme **T** è vuoto si parla di **albero vuoto**

PROBLEMA DELL'ATTRAVERSAMENTO DI UN ALBERO

In informatica è fondamentale trovare dei buoni metodi o algoritmi di **attraversamento** o **visita** di un albero poiché tali operazioni sono fortemente richieste dalle applicazioni che li utilizzano.

Notazione:

esame di un nodo: significa accedere al nodo ed estrarre le informazioni contenute in esso;

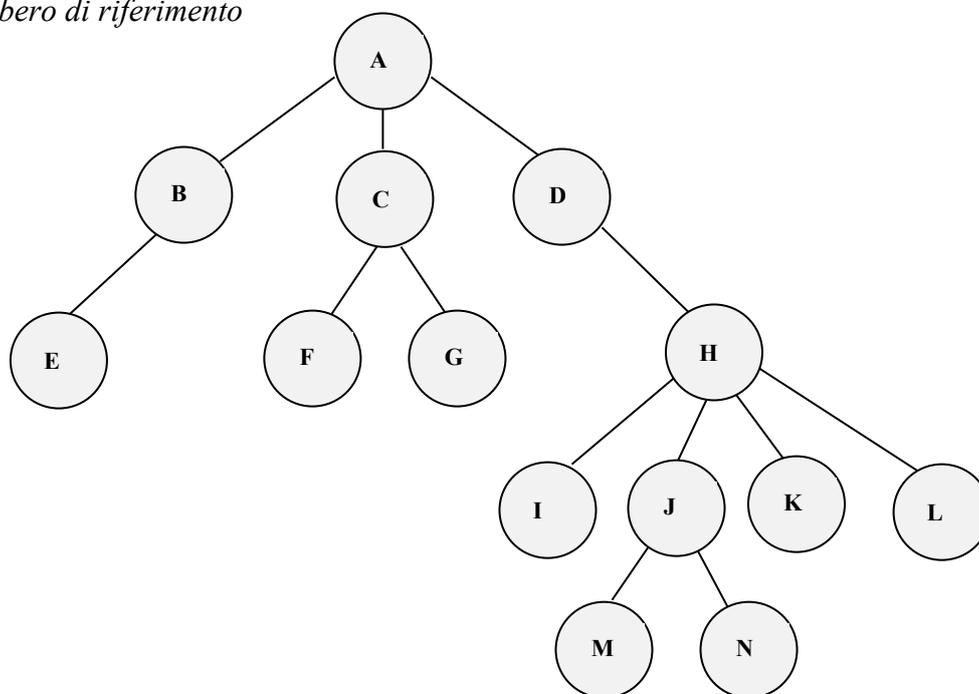
attraversare o visitare un albero: significa esaminare sistematicamente in un ordine appropriato tutti i nodi di un albero in modo che ciascun nodo venga visitato una volta sola;

attraversamento o visita di un albero: significa trovare un qualsiasi algoritmo che ci permetta di visitare un albero

I due principali algoritmi di attraversamento di un albero sono:

- 1) **attraversamento o visita in ordine ANTICIPATO (PRE-ORDER);**
- 2) **attraversamento o visita in ordine POSTICIPATO (POST ORDER);**

Esempio: albero di riferimento



1) VISITA anticipata di un albero (PRE-ORDER)

Il processo risolutivo RICORSIVO di questo algoritmo si può riassumere come segue:

Esamina la radice;

SE il numero n dei sottoalberi della radice è MAGGIORE DI zero

ALLORA

Attraversa il **primo** sottoalbero (in ordine anticipato o pre-order);

Attraversa il **secondo** sottoalbero (in ordine anticipato o pre-order);

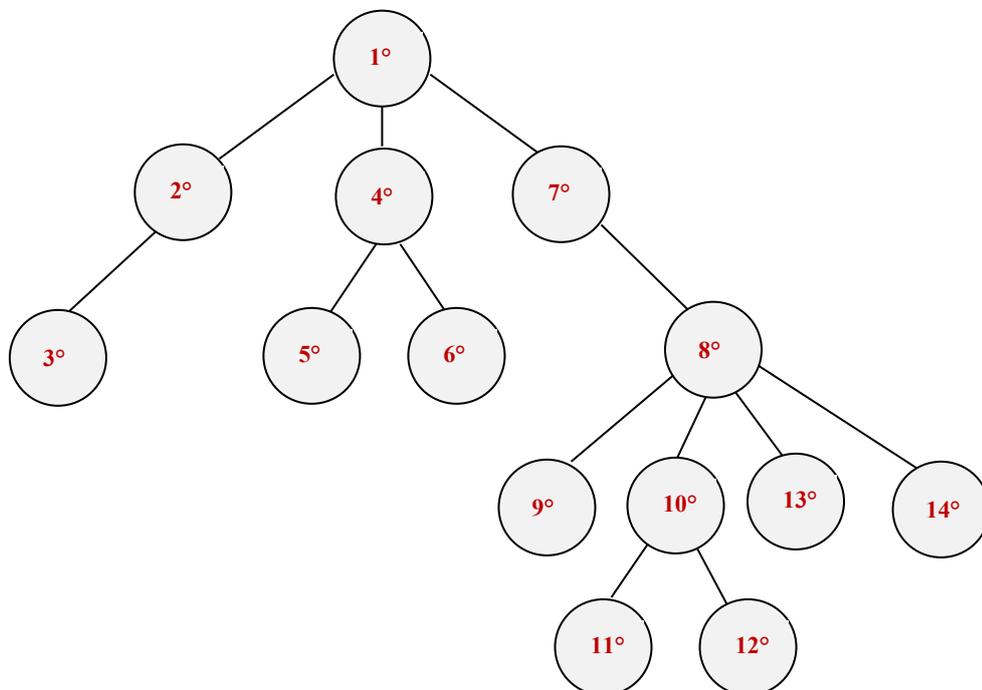
Attraversa l' **n -esimo** sottoalbero (in ordine anticipato o pre-order).

ALTRIMENTI

Ferma la ricorsione

FINE SE

Schematizzazione ordine di visita dei nodi con questo algoritmo:



Quindi l'attraversamento in ordine anticipato sull'albero di esempio darà la seguente sequenza di nodi:

A, B, E, C, F, G, D, H, I, J, M, N, K, L

2) VISITA posticipata o differita di un albero (POST-ORDER)

Il processo risolutivo RICORSIVO di questo algoritmo si può riassumere come segue:

SE il numero n dei sottoalberi della radice è **MAGGIORE DI zero**

ALLORA

Attraversa il **primo** sottoalbero (in ordine posticipato o post-order);

Attraversa il **secondo** sottoalbero (in ordine posticipato o post-order);

Attraversa l'**n-esimo** sottoalbero (in ordine posticipato o post-order);

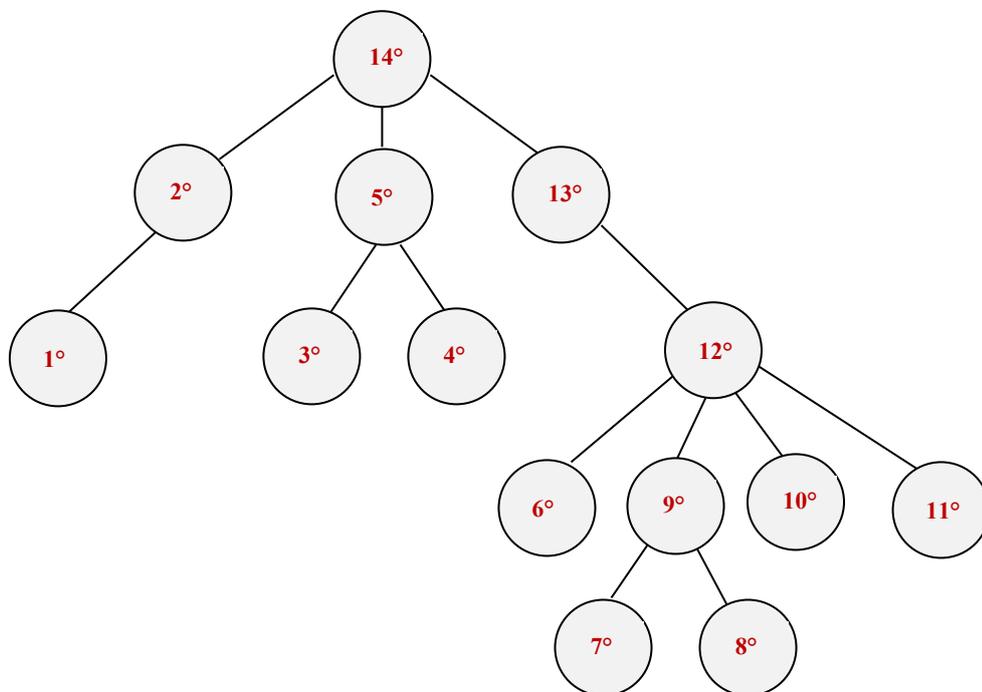
ALTRIMENTI

Esamina la radice

FINE SE

Ferma la ricorsione

Schematizzazione ordine di visita dei nodi con questo algoritmo:

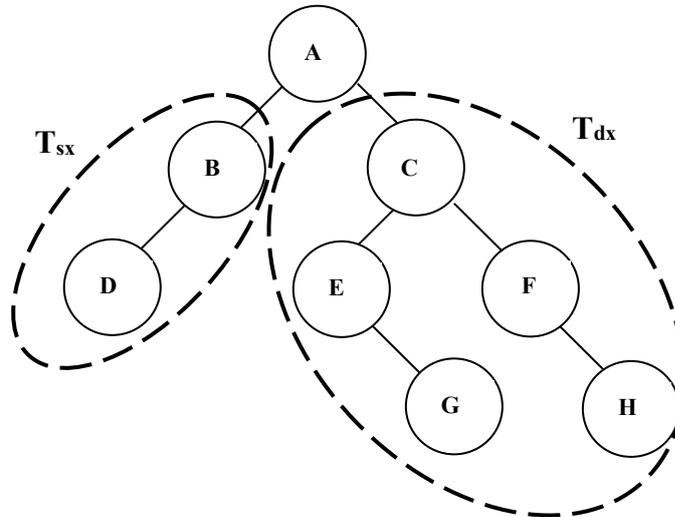


Quindi l'attraversamento in ordine posticipato sull'albero di esempio darà la seguente sequenza di nodi:

E, B, F, G, C, I, M, N, J, K, L, H, D, A

GLI ALBERI BINARI

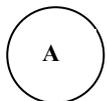
DEF 1. Un **albero binario** è un albero in cui ciascun nodo ha al massimo due figli



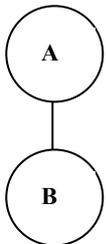
DEF 2 (RICORSIVA). Un **albero** si dice **binario** se:

- ha solo la radice (allora si dice **vuoto**);
- la radice ha al più due sottoalberi binari (rispettivamente *sottoalbero sinistro* e *sottoalbero destro*).

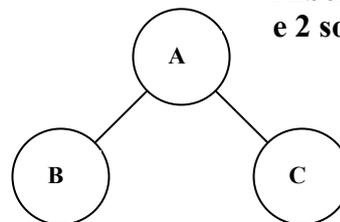
Secondo la definizione appena data sono alberi binari:



Albero binario vuoto (costituito dalla sola radice)



Albero binario con radice ed 1 sol sottoalbero



Albero binario con radice e 2 sottoalberi

APPLICAZIONI CON GLI ALBERI

Le strutture dati astratte **alberi** trovano applicazione nella risoluzione di problemi in cui i dati hanno una struttura logica appropriata.

Essi in genere vengono utilizzati **nella teoria dei giochi** per rappresentare le possibili partite di un gioco di abilità.

Inoltre i **motori di molti database** si basano sugli alberi per velocizzare le operazioni di ricerca delle informazioni memorizzate.