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**:

a) **allocazione statica** della memoria: la memoria necessaria viene allocata prima dell'esecuzione del programma a tempo di compilazione (**compile-time**);
b) **allocazione dinamica** della memoria: consente, durante l'esecuzione di un programma (**run-time**), di eseguire le istruzioni per allocare lo spazio in memoria necessario e di deallocarlo al termine della sua esecuzione, in modo da renderlo disponibile ad altri usi.

STRATEGIE DI ALLOCAZIONE A CONFRONTO

	Allocazione statica	**Allocazione dinamica**
Occupazione di memoria	Costante per tutta l'esecuzione del programma	Variabile nel corso dell'esecuzione in quanto le variabili sono allocate solo quando servono
Tempo di esecuzione	L'allocazione viene fatta una volta sola prima dell'esecuzione del programma, non appesantendo il suo tempo di esecuzione	L'allocazione e la deallocazione avvengono durante l'esecuzione del programma, appesantendo il suo tempo di esecuzione
Tempo di esistenza delle variabili	Sin dall'inizio dell'esecuzione del programma a tutte le variabili viene associata una zona di memoria permanente che verrà rilasciata solo quando il programma terminerà.	Le variabili vengono allocate durante l'esecuzione del programma solo quando servono e la memoria ad esse assegnata viene rilasciata con particolari istruzioni quando non servono più

Quando un programma non è in esecuzione - ossia quando risiede su una memoria di massa – subito dopo la compilazione ed il linkaggio, è costituito esclusivamente da codice o istruzioni e dai dati ed occupa un'area di memoria (la cui dimensione dipende esclusivamente dalle istruzioni e dai dati utilizzati) che è possibile essere pensata come suddivisa in:

Segmento "dati"
Segmento "codice"

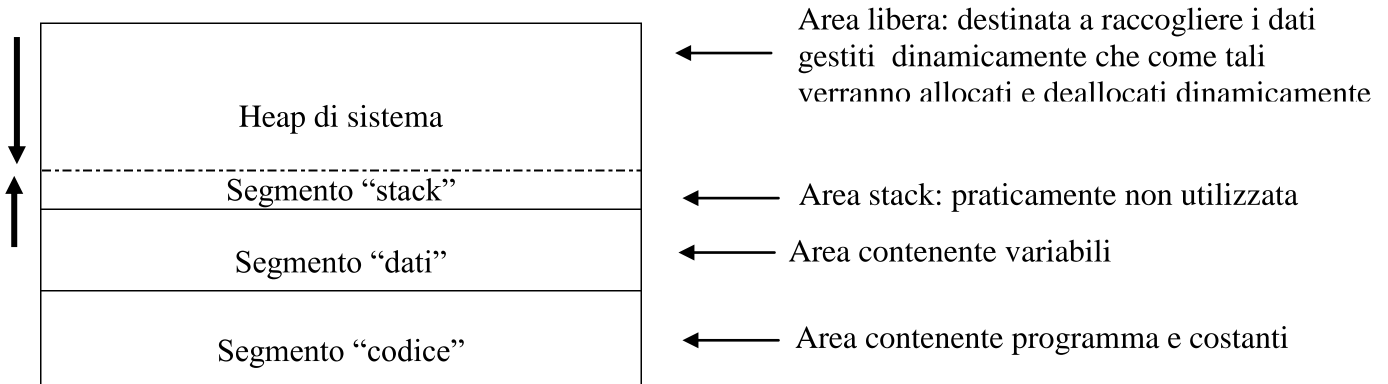
Per comprendere la tecnica dell'allocazione dinamica della memoria dobbiamo partire dall'illustrazione e dalla comprensione del **metodo** con cui un programma in esecuzione viene allocato in memoria centrale.

Generalmente quando un programma è in esecuzione (detto anche **TASK** o **processo**) gli viene assegnato dal sistema operativo nella memoria di lavoro (RAM) di un PC anche una zona di memoria aggiuntiva rispetto a quella posseduta quando è "in quiete".
Complessivamente possiamo pensare che esso occupi una zona che è suddivisa in 4 parti o segmenti:

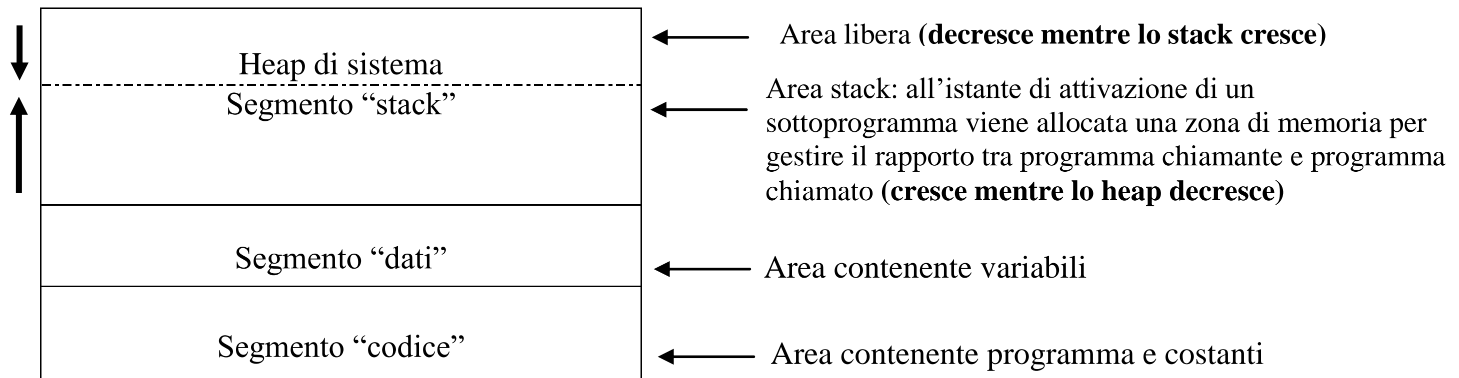
- **il Segmento "codice"** o "istruzioni"**;**
- **il Segmento "dati";**
- **il Segmento "stack";**
- **l'Heap** (lett. mucchio) **di sistema**

Heap di sistema
Segmento "stack"
Segmento "dati"
Segmento "codice"

CASO A)
Se un programma non utilizza sottoprogrammi impiega solamente risorse globali e pertanto occuperà sempre la stessa posizione di memoria non impiegando il segmento stack che gestisce le chiamate a funzioni e procedure con tecnica LIFO (pila o stack delle attivazioni).



CASO B)
Se un programma utilizza sottoprogrammi impiega, oltre alle risorse globali, anche il segmento stack allocando, ad ogni attivazione di un sottoprogramma nella pila o stack delle attivazioni, un'area di memoria (contenente l'indirizzo dell'istruzione successiva alla chiamata) che verrà poi liberata alla fine dell'esecuzione del sottoprogramma stesso.



N.B. L'allocazione all'interno dello heap di locazioni di meoria avverrà in entrambi i casi solo se verranno utilizzate variabili allocate dinamicamente tramite la pseudo-istrusìzione *Alloca* ( ) spiegata più avanti.

Il **segmento "stack**" come suggerisce la linea tratteggiata non ha una dimensione prefissata ma può crescere sino ad occupare l'intero heap del programma.
E' evidente che lo **heap** ossia l'area libera dove vengono allocate le variabili dinamiche e lo **stack** ossia l'area di memoria dove vengono allocate le risorse dei sottoprogrammi **si contendono l'area libera** muovendosi verso direzioni convergenti ossia da direzioni opposte.
Una eventuale collisione provoca l'interruzione 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

La maggior parte dei linguaggi di programmazione favorisce l'uso di allocazione dinamica della memoria mettendo 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 un dato

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

Un puntatore quindi non contiene direttamente dati come le altre variabili di altri tipi, ma contiene un indirizzo di memoria dove reperire i dati.

PSEUDOCODIFICA <Variabile puntatore> **= PUNTATORE A** <Tipo dato>

N.B. Per riferirci al valore del dato puntato da un puntatore useremo il simbolo * davanti al nome della variabile puntatore

N.B. Per assegnare l'indirizzo di una cella di memoria ad un puntatore useremo il simbolo & davanti al nome della variabile

N.B.: Una variabile puntatore occupa sempre la stessa quantità di memoria indipendentemente dal tipo di dato puntato.

Per indicare il dato puntato da un puntatore utilizzeremo l'operatore *.
Nel caso dell'esempio sopra riportato sarà

***p = q**

COME ALLOCARE I DATI NELLO HEAP

Al tipo puntatore possono essere applicate le procedure e le funzioni che diremo (intese in PSEUDOCODIFICA) di seguito messe a disposizione da quasi tutti i linguaggi di programmazione.

Alloca (<nome puntatore>, <numero byte>|**DimensioneDi (**<variabile> | <tipo dato>**))**

è una procedura che assegna al puntatore <nome puntatore> l'indirizzo di un'area disponibile nello heap di dimensione in byte pari al secondo parametro eventualmente restituita dalla funzione DimensioneDi( )) per generare una variabile dinamica.

Dealloca (<nome puntatore>)

è una procedura che rilascia la memoria occupata nello heap da una precedente operazione di *Alloca* rendendola disponibile di nuovo per eventuali allocazioni dinamiche. In pratica rompe il link o collegamento che si era creato tra la variabile di tipo puntatore e la zona di memoria puntata.

DimensioneDi (<variabile> | <tipo dato>)

è una funzione che restituisce la quantità di memoria espressa in byte che una variabile oppure un determinato tipo di dato occupa in memoria. E' una funzione di grande utilità in quanto evita al programmatore il calcolo della quantità di memoria da allocare nello heap per contenere un dato.

Quando si usano variabili di tipo puntatore è buona norma, come sempre, inizializzarle sempre con valori definiti, al fine di evitare inutili inconvenienti.
Per le variabili puntatore si utilizza un particolare valore per indicare che esso non è associato a nessuna variabile dinamica ossia che non punta a nessuna area di memoria.
Tale valore costante (è quindi un valore a tutti gli effetti) prende il nome di **NULL** (niente)

MemoraLibera ( )

è una funzione che restituisce la dimensione espressa in byte del più grande blocco di memoria disponibile per allocazioni dinamiche. Ci permette quindi di verificare che nello heap esista spazio sufficiente per allocare dinamicamente nuove variabili.

OPERAZIONI DI CONFRONTO SUI PUNTATORI

Le uniche operazioni di confronto consentite sui puntatori sono quelle di assegnazione e di confronto (solo = e ≠) tra puntatori dello stesso tipo.

ARITMETICA DEI PUNTATORI

Nei linguaggi di programmazione, 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 collezioni di dati omogenei conservati in posizioni contigue della memoria (per esempio array).
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** della somma è l'indirizzo dato dal puntatore **incrementato** del risultato della moltiplicazione dell'intero specificato per la dimensione (DimensioneDi) del tipo base del puntatore espressa in byte.

*Per esempio, se p è un puntatore al tipo intero INT (p: PUNTATORE A INT) di valore 1000 (p=1000 quindi p punterà alla locazione di memoria di indirizzo 1000), e se la dimensione di un INT è due byte, p+1 vale 1002 (quindi p punterà alla locazione di memoria di indirizzo 1002), p+2 vale 1004 (quindi p punterà alla locazione di memoria di indirizzo 1004) e in generale p+n vale 1000+n*2 (quindi p punterà alla locazione di memoria di indirizzo 1000 + n*2),.*

Significato

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

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.

Operatore di sottrazione di un intero da un puntatore

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) è l'indirizzo dato dal puntatore **decrementato** del risultato della moltiplicazione dell'intero specificato per la dimensione (DimensioneDi) del tipo base del puntatore espressa in byte.

*Per esempio, se p è un puntatore al tipo intero INT (p: PUNTATORE A INT) di valore 1000 (p=1000 quindi p punterà alla locazione di memoria di indirizzo 1000), e se la dimensione di un INT è due byte, p-1 vale 998 (quindi p punterà alla locazione di memoria di indirizzo 998), p-2 vale 996 (quindi p punterà alla locazione di memoria di indirizzo 996), , e in generale p-n vale 1000-n*2 (quindi p punterà alla locazione di memoria di indirizzo 1000 - n*2),.*

Significato

Sotto le 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.

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 1002 e q contiene l'indirizzo 1000, ed entrambi sono puntatori al tipo intero INT e la dimensione di un INT è 2 byte, allora q-p vale 1 ossia (q – p) / DimensioneDi(INT)

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 dell'array che separano la cella puntata dal puntatore di valore minore da quella del puntatore di valore maggiore.

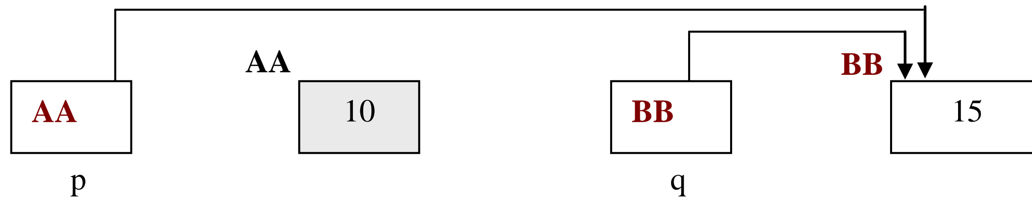
GARBAGE, ALIASING, SIDE EFFECT E DANGLING REFERENCE

Uno dei problemi principali quando si utilizza l'allocazione dinamica è quello della deallocazione delle vecchie variabili allocate (**garbage collection**).

Infatti a volte è possibile avere variabili dinamiche **inaccessibili** ovvero allocate in memoria e mai deallocate non solo per la mancata deallocazione del programmatore, ma per l'esecuzione di istruzioni particolari che hanno un effetto collaterale.

Esempio: Siano dati i due puntatori p e q ad intero **allocati dinamicamente** che puntano rispettivamente a due aree di memoria distinte di indirizzo rispettivamente AA e BB (esadecimale) valorizzate una con il valore 10 ed una con il valore 15:

Con l'istruzione p ← q viene creato il fenomeno dell'**ALIASING** ossia un puntatore può essere creato come sostituto (alias) dell'altro.



Questo fenomeno dell'alias si ha solo con i puntatori perché il compilatore per ogni variabile allocata staticamente associa una cella di memoria distinta.

La cella prima puntata da l puntatore p è ora <u>inaccessibile</u> e diventa **garbage**.
Il ripetersi di tali situazioni durante l'esecuzione del programma può portare al collasso della memoria heap dovuto al proliferare di locazioni inaccessibili.

L'aliasing usato male porta a rischi di effetti secondari indesiderati ossia può avere **SIDE EFFECT**

Esempio: si consideri il seguente frammento di pseudocodice
*p ← 3
*q ← 5
p ← q
*q ← 7
Dopo l'ultima istruzione anche ***p vale 7**

Un altro fenomeno che frequentemente si verifica con i puntatori strettamente legato all'aliasing è quello del **DANGLING REFERENCE** (ossia il riferimento "penzoloni")

Esempio: si consideri il seguente frammento di pseudocodice
p ← q
Dealloca (q)
Scrivi (*p)
ERRORE: <u>l'area a cui puntava p è stata deallocata in precedenza</u>

Per recuperare il garbage (**garbage collection**) evitando così anche il dangling reference esistono due modalità :

(*) **modalità manuale** (<u>a carico del programmatore)</u> con opportune istruzioni del programma (ad esempio la pseudo istruzione Dealloca( ));

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

ESERCIZI SULL'UTILIZZO DEI PUNTATORI

Esercizio 1) 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

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

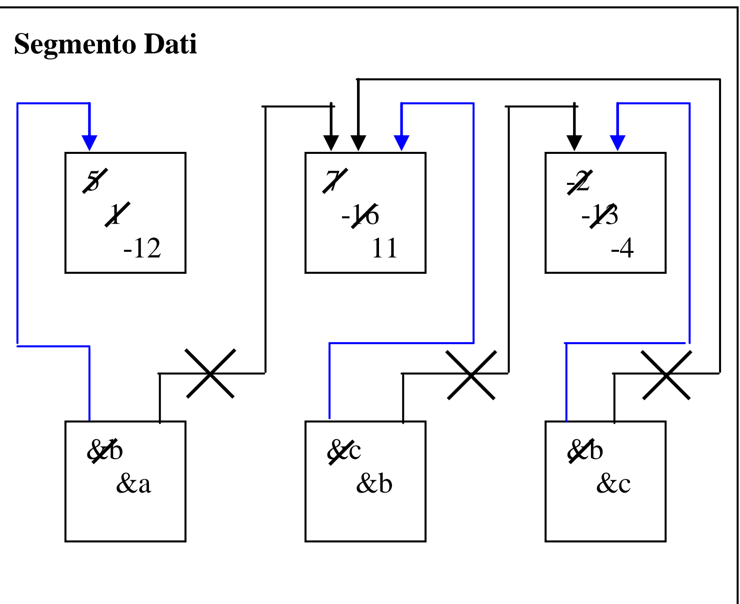
Scrivi (a)
Scrivi (b)
Scrivi (c)

Scrivi (*p1)
Scrivi (*p2)
Scrivi (*p3)

Scrivi (p1)
Scrivi (p2)
Scrivi (p3)

RITORNA
FINE

Segmento Dati



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

p2 ← &c (p2 punterà alla variabile c: occorre disegnare la relativa freccia)
p1 ← &b (p1 punterà alla variabile b: occorre disegnare la relativa freccia)
p3 ← p (p3 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)
p3 ← a + (*p1) – (*p2) (*p2 = c = 1 + (-16) – (-2) = 1 – 16 + 2 = -13)

p3 ← p2 (p3 punterà alla medesima variabile puntata da p2: disegnare la nuova freccia e cancellare precedente freccia)
p2 ← &(*p1) (p2 punterà alla medesima variabile puntata da p1 disegnare la nuova freccia e cancellare precedente freccia)
p1 ← &a (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)

Soluzione

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

Esercizio 2) Scrivere la pseudocodifica di un algoritmo che esegue il caricamento e la visualizzazione di un vettore o array monodimensionale allocato dinamicamente di n elementi interi

ALGORITMO Array_Dinamico_1

PROCEDURA main ( )

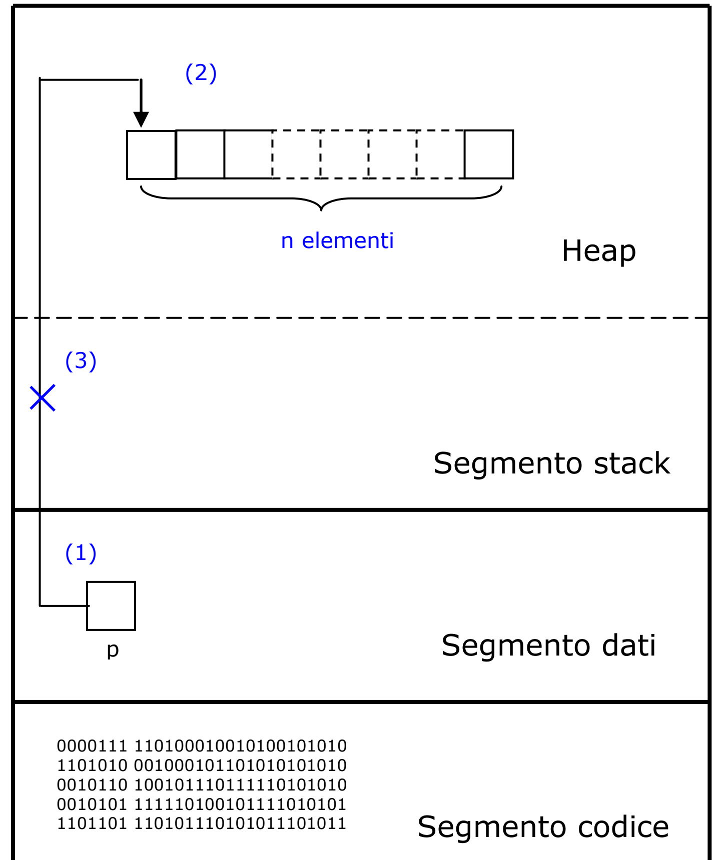
p : **PUNTATORE A INT**
n, i : **INT**

INIZIO

/* Check sul numero di elementi possibili dell'array */
/* dinamico: VERA DINAMICITA' */
RIPETI
  Leggi (n)
FINCHE' (n ≥ 1)

/* Allocazione area di memoria dimamica */
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 dimamica */
      Dealloca (p) (3)
  ALTRIMENTI
    Scrivi ("Errore nell'allocazione")

RITORNA
FINE



(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 punterà al prima locazione di memoria (di n previsti) in grado di contenere valori interi

N.B. 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 acccedere 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)

In particolare per iterazioni con indice crescente
p ptr alla prima locazione di memoria (ossia **p + 0**) *p (ossia * (p+0)) primo elemento dell'array
p+1 ptr alla seconda locazione di memoria *(p+1) secondo elemento dell'array
p+2 ptr alla terza locazione di memoria *(p+2) terzo elemento dell'array
p+(n-1) ptr alla n-esima locazione di memoria *(p+(n-1)) n-esimo elemento dell'array

(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. La funzione **Dealloca** (…) non ripulisce in alcun modo i valori precedentemente assegnati

STRUTTURE DATI ASTRATTE LINEARI

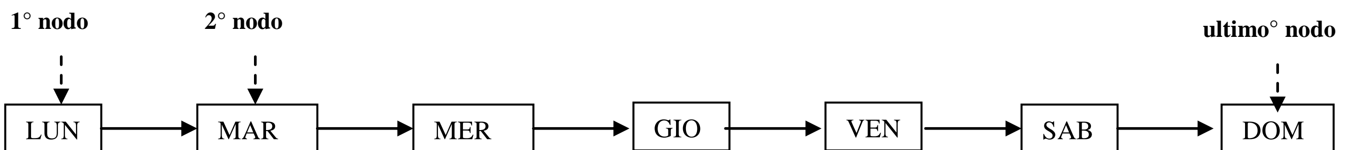
Definiamo **nodo** l'unità di informazione relativa ala struttura dati che stiamo analizzando

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** $P_1, P_2, \ldots P_n$ 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.

Vediamo ora **SECONDO LE SPECIFICHE DELL'ADT SEQUENZA** quali sono le principali operazioni possibili su di una sequenza o lista di nodi:

Una premessa rotazionale : indichiamo
con **[ ]** una sequenza o lista vuota (con **'['** che indica la **testa** e con **']'** che indica il **fondo**)
con $[P_1, P_2, \ldots P_n]$ una sequenza formata dai nodi $P_1, P_2, \ldots P_n$ 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 ∅ l'insieme vuoto
con **B** l'insieme contenente i valori booleani VERO e FALSO

Le operazioni possibili su tale struttura dati astratta vengono definite in questo caso come **funzioni matematiche** che calcolano valori a fronte di altri valori e sono:

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

$$Crea: \varnothing \rightarrow S$$

che provvede a creare la struttura o lista vuota [ ];

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

- **in testa** utilizzeremo la funzione ***InsTesta( )***

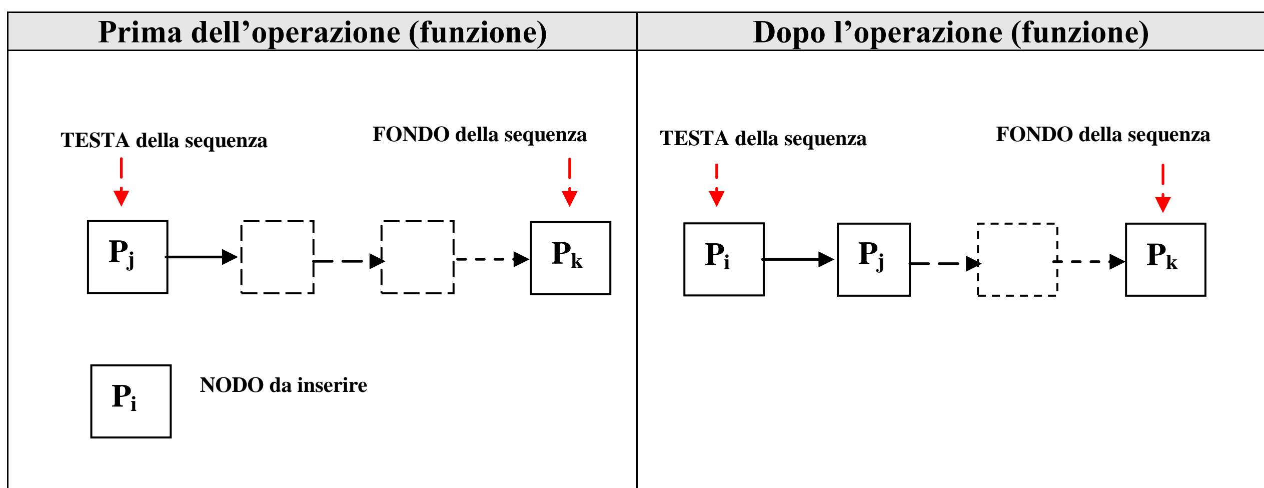
$$InsTesta: S \times N \rightarrow S$$

che necessita di due parametri in ingresso: uno identifica la sequenza che stiamo considerando $[P_j, .., P_k]$ 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
$[P_i, P_k, \ldots, P_k]$.

Possiamo scrivere in breve

$$InsTesta\ ([P_j, .., P_k] , P_i) = [P_i, P_k, \ldots, P_k].$$

Graficamente



- **in fondo** utilizzeremo la funzione ***InsFondo( )***

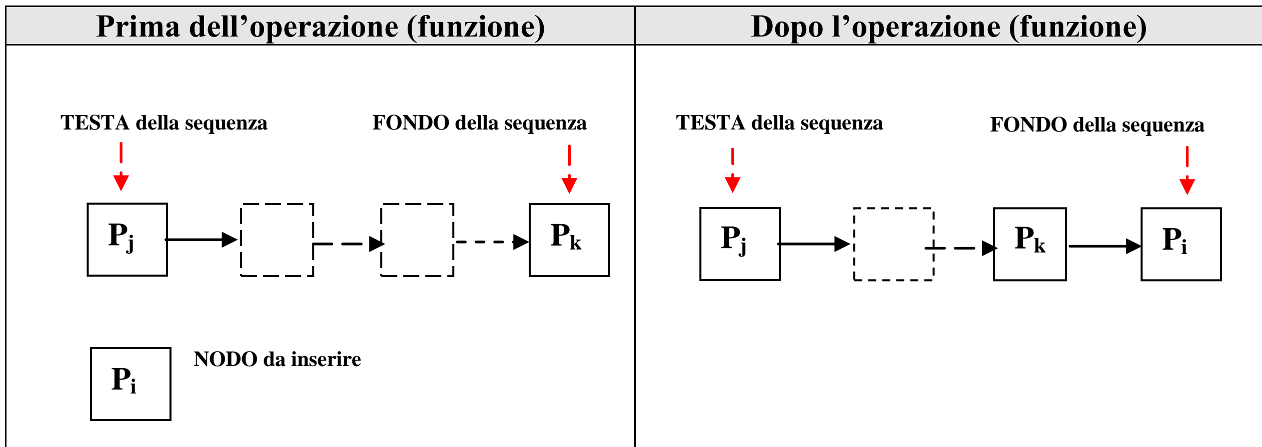
$$InsFondo: S \times N \rightarrow S$$

che necessita di due parametri in ingresso: uno identifica la sequenza che stiamo considerando $[P_j, .., P_k]$ 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 $[P_j, \ldots, P_k, P_i]$.

Possiamo scrivere in breve

$$InsFondo\ ([P_j, .., P_k] , P_i) = [P_j, \ldots, P_k, P_i]$$

Graficamente



- **in un punto qualsiasi** utilizzeremo la funzione ***InsPos( )***

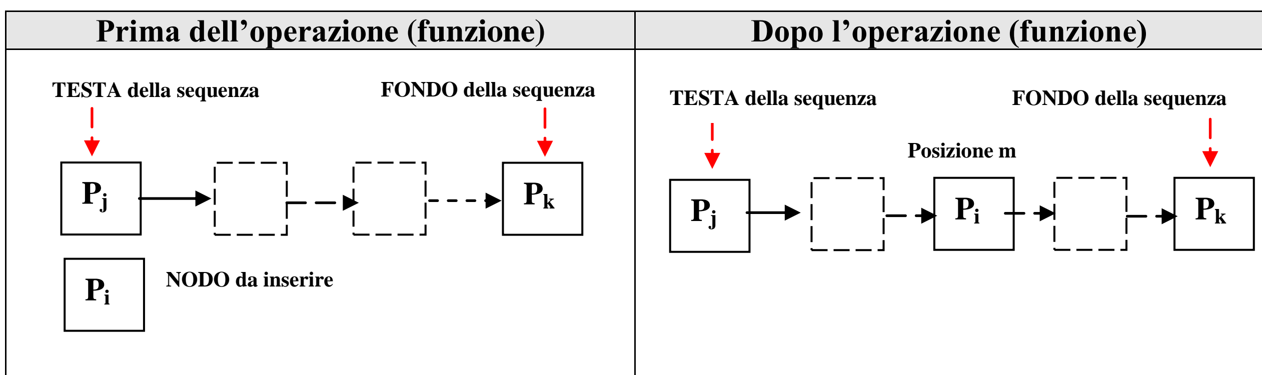
$$InsPos: S \times N \times Z^{+} \rightarrow S$$

che necessita di tre parametri in ingresso: uno identifica la sequenza che stiamo considerando **[P_j,..,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,..,P_i,.., P_k].**

Possiamo scrivere in breve

InsPos (**[P_j, ..,P_k] , P_i , m**) = **[P_j,.., P_i,.., P_k]** (con P_i in posizione m)

Graficamente



N.B. Ovviamente tale funzione sarà possibile se la lista ha almeno n nodi con n ≥ 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( )***

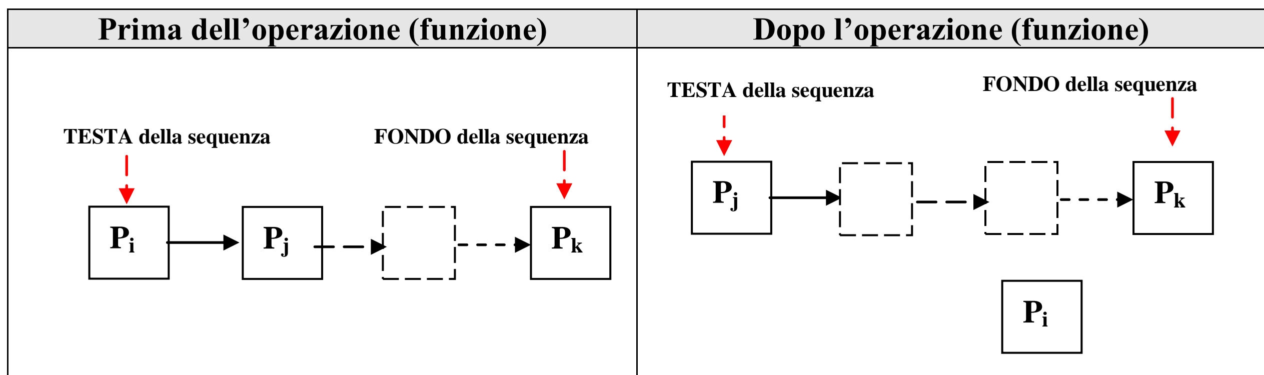
  CancTesta: $S \rightarrow S \times N$

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

  Possiamo scrivere in breve

  CancTesta ($[P_i, P_j,.., P_k]$) = $[P_j, .., P_k]$, P_i

Graficamente



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

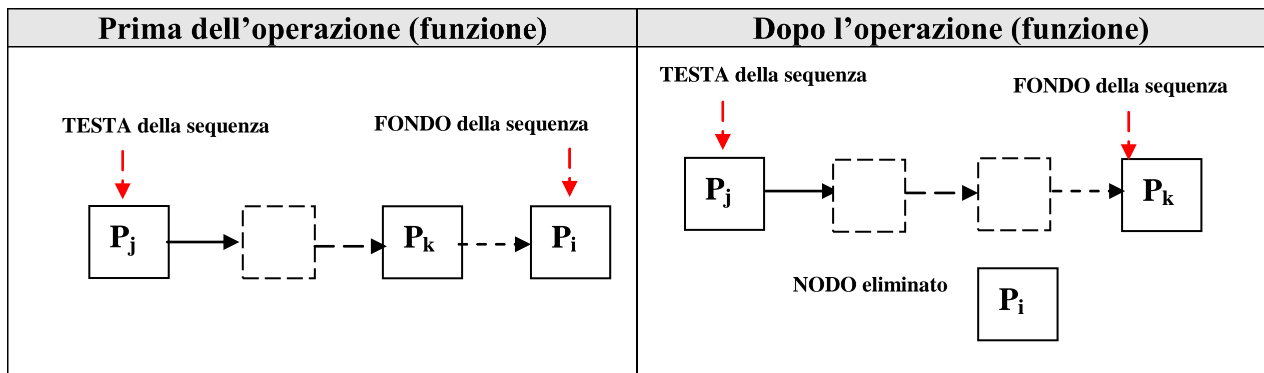
  CancFondo: $S \rightarrow S \times N$

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

  Possiamo scrivere in breve

  CancFondo($[P_j, ..,P_k, P_i]$) = $[P_j,.., P_k]$., P_i

Graficamente



- **di un qualsiasi nodo** utilizzeremo la funzione ***CancPos( )***

  CancPos: S x $Z^+ \rightarrow$ S x N

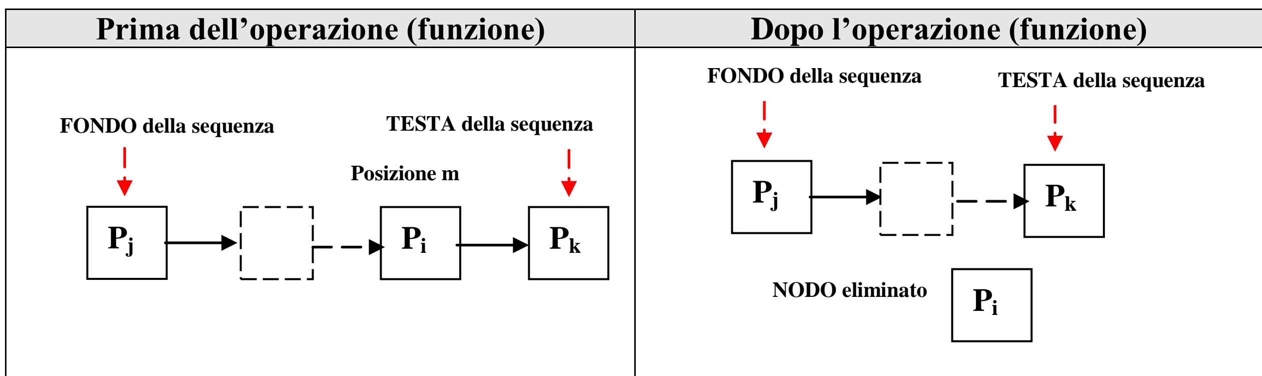
  che necessita di due parametri in ingresso: uno contenente la sequenza che stiamo considerando **[P_j,..,P_i,.., P_k]** ed il secondo contenente la posizione del nodo da eliminare. La funzione restituirà la nuova sequenza ottenuta eliminando il nodo **[P_j,.., P_k]**, assieme al nodo **P_i** eliminato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

  .

  Possiamo scrivere in breve

  CancPos (**[P_j,.., P_i,.., P_k] , m) = [P_j,..,P_k]., P_i**

Graficamente



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

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

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

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**.

Ad esempio

TestVuota (**[P_j,.., P_k]**) = **FALSO**
TestVuota ([ ]) = **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( )***

Ricerca: $S \times N \rightarrow B$

che necessita di due parametri in ingresso: uno contenente la sequenza che stiamo considerando **[P_j,.., 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( )***

Lunghezza: $S \rightarrow Z^+$

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

Ordina: $S \times M \rightarrow S$

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

Fondi: $S \times S \rightarrow S$

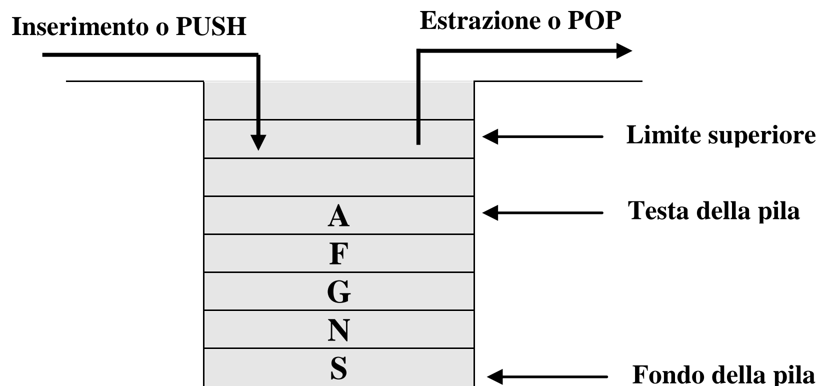
PILA O STACK

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

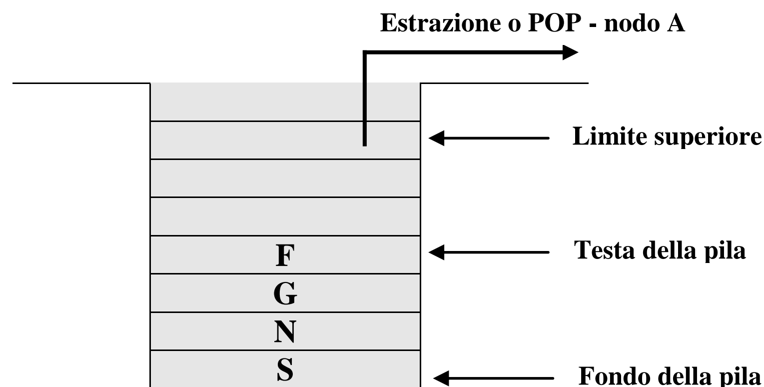
Questo implica che il primo nodo estraibile o prelevabile dalla testa della pila sia sempre l'ultimo nodo inserito (struttura dati di tipo **LIFO** ossia Last In First Out)

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

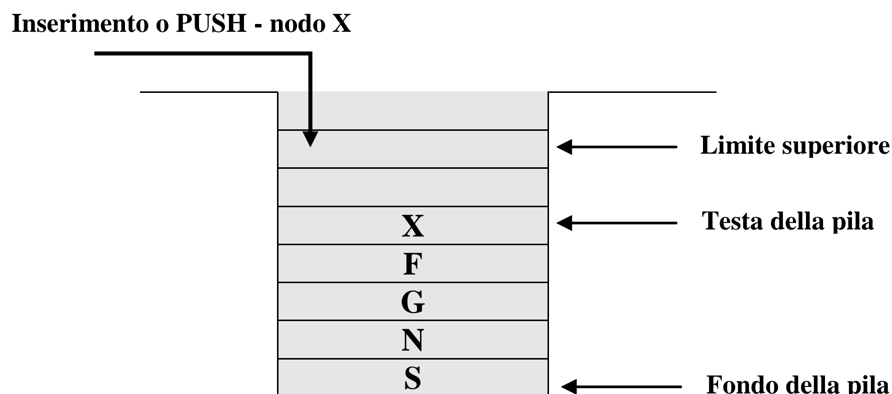
a) **estrazione o prelevamento** del nodo dalla testa (operazione detta **pop**);
b) **inserimento** di un nuovo nodo in testa (operazione detta **push**).



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



Vediamo ora **SECONDO LE SPECIFICHE DELL'ADT PILA** quali sono le principali operazioni possibili su di una pila o stack:

Una premessa rotazionale : indichiamo

con **[ )**	una pila vuota (con **')'** che indica la **testa** e con **'['** che indica il **fondo** fisso)
con **[P_1, P_2 ,…. P_n)**	una pila qualsiasi formata dai nodi **P_1, P_2 ,….** **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 $\varnothing$	l'insieme vuoto
con **B**	l'insieme contenente i valori booleani VERO e FALSO

Anche in questo caso le operazioni possibili su tale struttura dati astratta vengono definite in questo caso come **funzioni matematiche** che calcolano valori a fronte di altri valori e sono:

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

$$Crea: \varnothing \rightarrow P$$

che provvede a creare la pila vuota [ );

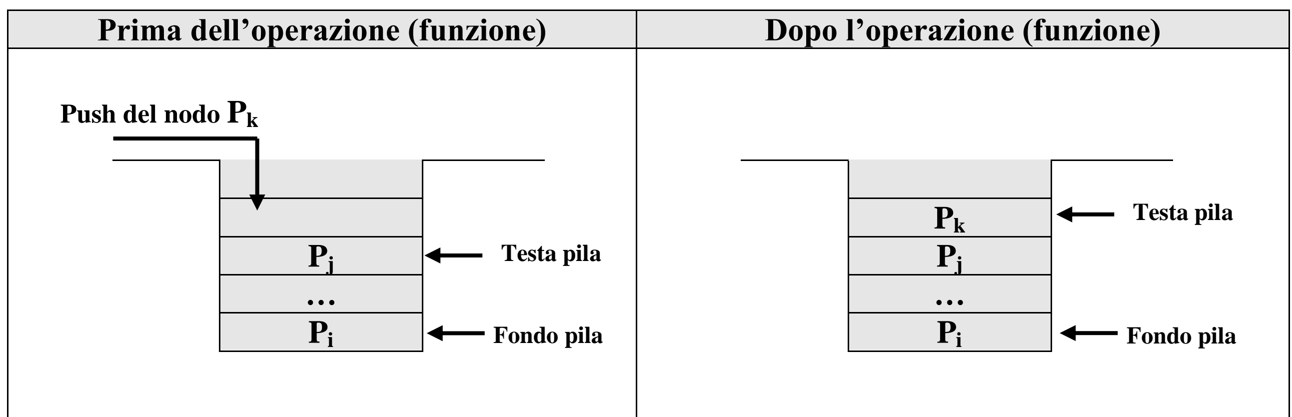
b) l' **inserimento (PUSH)** di un nodo che può avvenire **esclusivamente in testa** per il quale utilizzeremo la funzione ***Push( )***

$$Push: P \times N \rightarrow P$$

che necessita di due parametri in ingresso: uno identifica la pila che stiamo considerando **[P_i,..,P_j)** e l'altro il nodo **P_k** che vogliamo aggiungere in testa. La funzione restituirà la nuova pila ottenuta **[P_i,.., P_j, P_k).**
Possiamo scrivere in breve

Push (**[P_i,..,P_j) , P_k) = [P_i,..,P_j, P_k).**

Graficamente



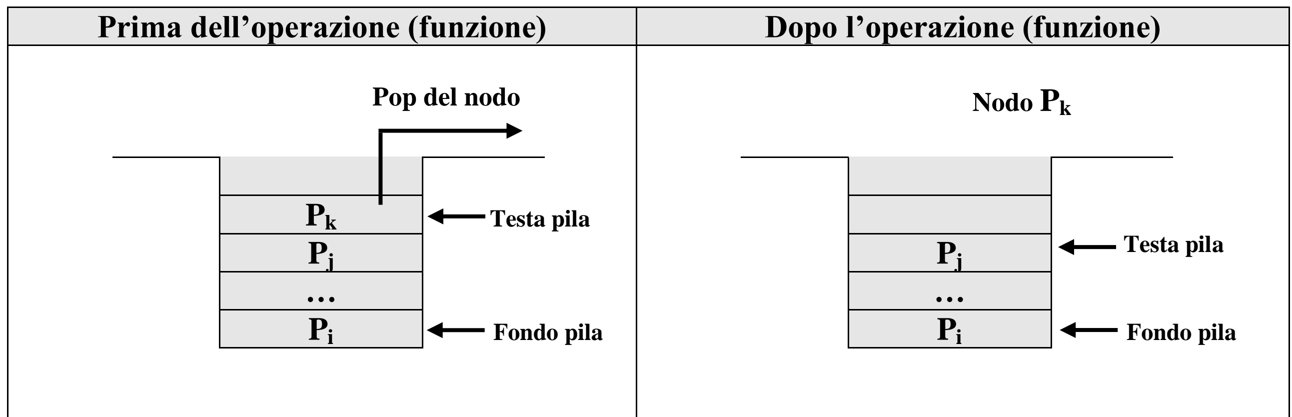
c) l' **estrazione (POP)** di un nodo che può avvenire **esclusivamente dalla testa** per la quale utilizzeremo la funzione ***Pop( )***

Pop: $P \rightarrow P \times N$

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

Possiamo scrivere in breve
Pop ($[P_i,..,P_j ,P_k)$) = $[P_i,..,P_j)$ più il nodo estratto dalla testa P_k

<u>Graficamente</u>



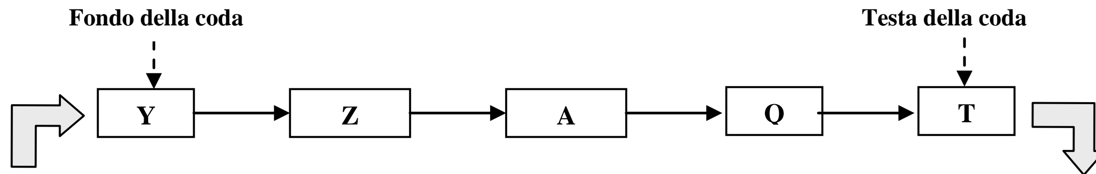
d) il **test di pila vuota** per il quale utilizzeremo la funzione ***TestVuota( )***
TestVuota: $P \rightarrow B$

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**.

Ad esempio
TestVuota ($[P_i,..P_j)$) = **FALSO**
TestVuota (**[ )**) = **VERO**

CODA O QUEUE

La **coda 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 coda** e gli inserimenti avvengono sempre sull'altro estremo detto **fondo della coda**



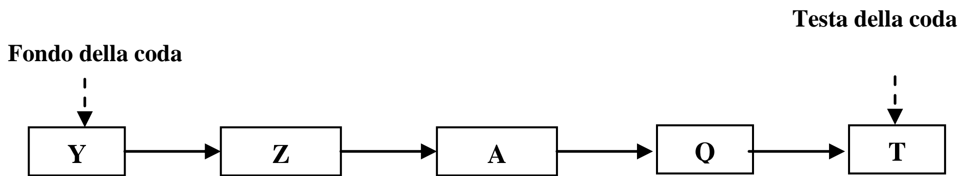
Questo implica che il nodo che può essere estratto da una coda sia sempre il primo nodo inserito nella stessa (struttura dati di tipo **FIFO** ossia First In First Out)

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

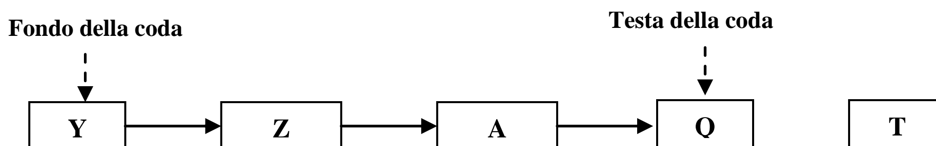
a) **estrazione o prelevamento** di un nodo **dalla testa** della coda;
b) **inserimento** di un nuovo nodo **in fondo** alla coda.

Nel dettaglio vediamo l'operazione di **estrazione** di un nodo <u>dalla testa della coda</u>

PRIMA DELL'ESTRAZIONE

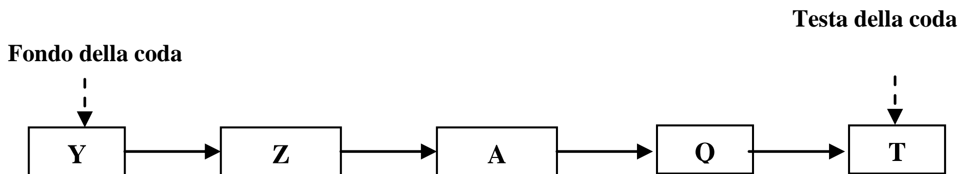


DOPO DELL'ESTRAZIONE

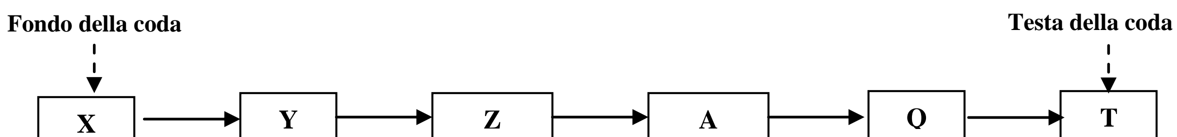


Nel dettaglio vediamo l'operazione di **inserimento** di un nuovo nodo X <u>in fondo alla coda</u>

PRIMA DELL'INSERIMENTO DEL NODO X



DOPO DELL'INSERIMENTO DEL NODO X



Vediamo ora **SECONDO LE SPECIFICHE DELL'ADT CODA** quali sono le principali operazioni possibili su di una coda o queue:

Una premessa rotazionale : indichiamo

con {)	una coda vuota (con **'{'** che indica il **fondo** e con **')'** che indica la **testa**)
con **{P_1, P_2 ,…. P_n)**	una coda qualsiasi formata dai nodi **P_1, P_2 ,…. 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 ∅	l'insieme vuoto
con **B**	l'insieme contenente i valori booleani VERO e FALSO

Anche in questo caso le operazioni possibili su tale struttura dati astratta vengono definite in questo caso come **funzioni matematiche** che calcolano valori a fronte di altri valori e sono:

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

$$Crea: \varnothing \rightarrow C$$

che provvede a creare la coda vuota {);

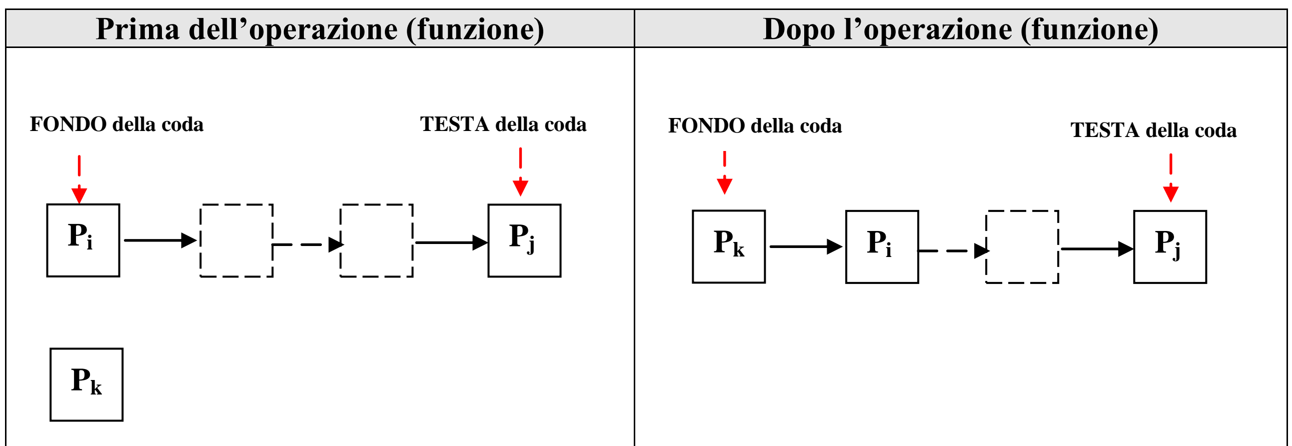
b) l' **inserimento** di un nodo che può avvenire **esclusivamente in fondo alla coda** e per il quale utilizzeremo la funzione ***Inserisci( )***

$$Inserisci: C \times N \rightarrow C$$

che necessita di due parametri in ingresso: uno identifica la coda che stiamo considerando **{P_i,..,P_j)** e l'altro il nodo **P_k** che vogliamo aggiungere in fondo alla coda. La funzione restituirà la nuova coda ottenuta **{P_k, P_i,.., P_j).**
Possiamo scrivere in breve

Inserisci (**{P_i,..,P_j) , P_k**) = **{P_k, P_i,.., P_j).**

Graficamente



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

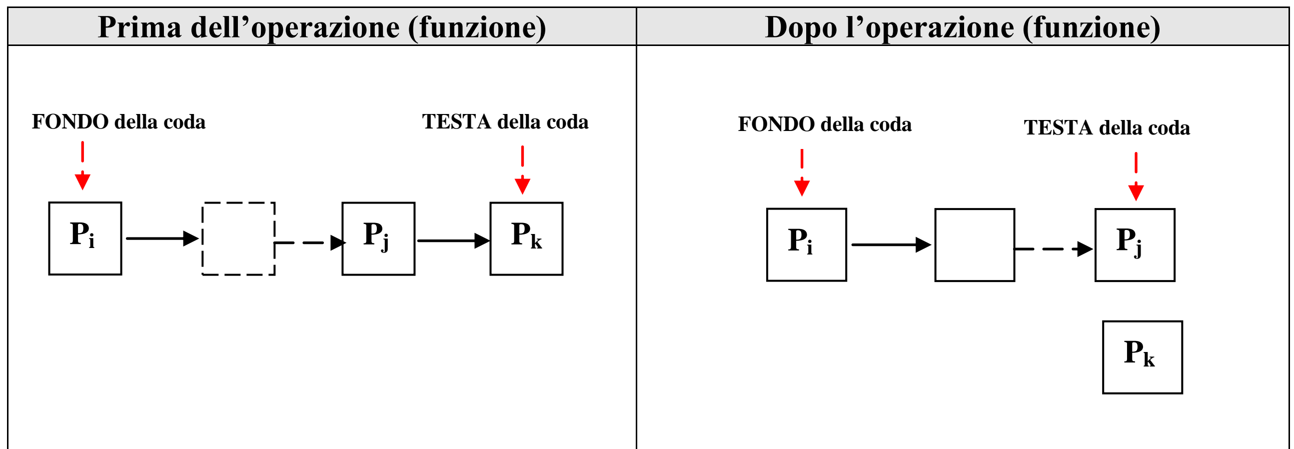
Estrai: C $\rightarrow$ C x N

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

Possiamo scrivere in breve

Estrai (**{P_i,..,P_j, P_k)**)= **{P_i,..,P_j)** più il nodo estratto dalla testa **P_k**

Graficamente



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

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**.

Ad esempio

TestVuota (**{P_i,..,P_j)**) = **FALSO**
TestVuota (**{)**) = **VERO**

IMPLEMENTAZIONE DELLE STRUTTURE DATI ASTRATTE LINEARI

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

- attraverso strutture dati **ad allocazione sequenziale e statica** del linguagio di programmazione scelto;

- attraverso strutture dati **ad allocazione non sequenziale e dinamica** del linguagio di programmazione scelto;

Limiti dell'allocazione statica della memoria

L'allocazione statica della memoria ha molti vantaggi tra i quali segnaliamo:
1) **semplicità degli algoritmi** che devono gestire la struttura dati così allocata;
2) **accesso diretto ai nodi** (poiché vengono utilizzati i vettori).

Nonostante ciò tale metodo di memorizzazione è poco flessibile per quanto riguarda:
a) **occupazione di memoria**: gli array in genere sono sovradimensionati non potendo stimare a priori la dimensione del problema;
b) **velocità in fase di esecuzione**: gli array sono strutture rigide e la loro gestione richiede a volte tempi che non sono accettabili;
c) **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.

L'utilizzo di strutture dati **concatenate e dinamiche** (al posto di quelle **sequenziali e statiche**) 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.

N.B. In una struttura dati concatenata la successione logica dei nodi è arbitraria e non corrisponde in alcun modo a quella fisica.

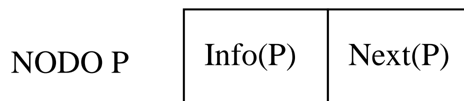
Infatti in queste strutture non è assolutamente necessario che la locazione (fisica) di memoria in cui si trova il nodo $\mathbf{P_k}$ generico debba essere successiva a quella dove si trova il nodo $\mathbf{P_{k-1}}$ oppure debba essere precedente a quella dove si trova il nodo $\mathbf{P_{k+1}}$.
Il nodo $\mathbf{P_k}$ può essere memorizzato dovunque: l'importante è che il nodo $\mathbf{P_{k-1}}$ contenga tra le sue informazioni l'indirizzo della locazione di memoria dove si trova il nodo $\mathbf{P_k}$ (ossia "punti al nodo" $\mathbf{P_k}$) e che quest'ultimo a sua volta contenga l'indirizzo della locazione di memoria dove si trova il nodo $\mathbf{P_{k+1}}$ (ossia "punti al nodo" $\mathbf{P_{k+1}}$)
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.

L'indirizzo di memoria del primo nodo è infatti una informazione necessaria ma esterna alla struttura dati concatenata. Esso deve essere memorizzato in un apposita variabile che chiameremeo **puntatore alla testa** della struttura dati.
Inoltre con le struttura concatenate non è 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.

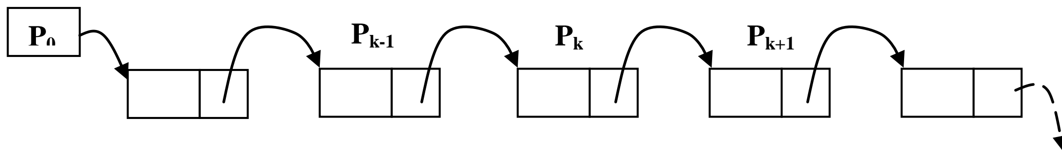
LA LISTA LINKATA

DEF. La lista concatenata o linkata è la struttura dati dinamica di base.
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** (quindi tutti i nodi hanno lo stesso formato) è strutturato in due campi:
- un campo **informativo** (che chiameremo *Info*) che può contenere qualsiasi tipo di informazione;
- un campo **puntatore** (che chiameremo *Next*) che contiene l'indirizzo di memoria in cui è possibile reperire il nodo successivo.



Graficamente abbiamo:



P_0 come abbiamo detto è il puntatore al primo nodo ed è *"esterno alla lista",* mentre l'ultimo elemento che non ha successori non punta a nessuno (ossia *"punta a terra"* per convenzione il valore NULL).

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**.

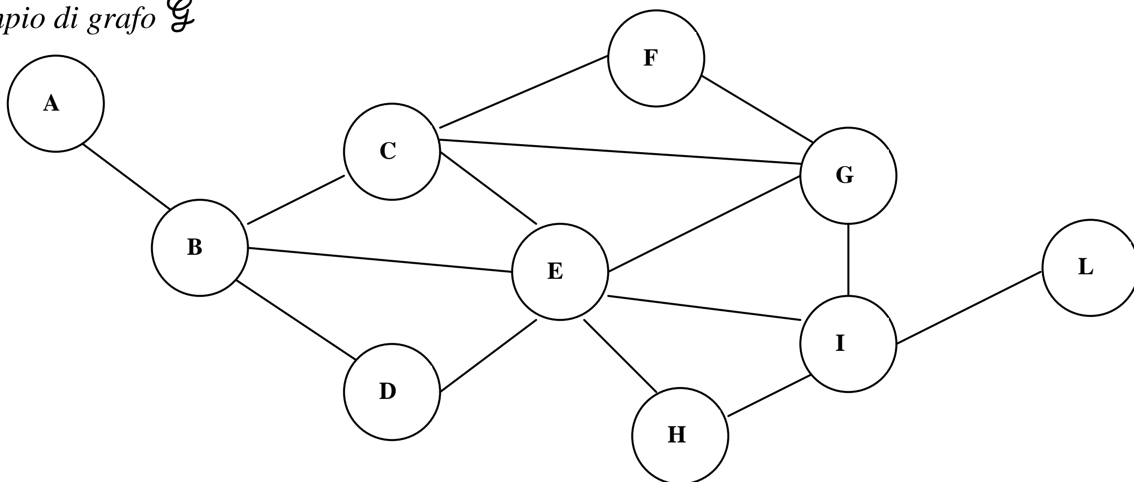
Vogliamo estendere questi concetti a strutture dati non lineari chiamate *grafi* ed in particolare agli *alberi* che costituiscono le più importanti strutture della nuova classe di strutture da esaminare.
In particolare anticipiamo che nei **grafi** per **ogni nodo** è possibile avare **più nodi predecessori** 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** $\mathcal{G}$ è definito da :

- un **insieme** finito e non vuoto **N** di **nodi o vertici** chiamati comunemente "**punti**";
- un **insieme A** di **segmenti o archi** detti anche "**spigoli**" o "**lati**";
- una **funzione F** che descrive le **connessioni** tra le coppie di punti (ossia ad ogni arco fa corrispondere una coppia di nodi).

Esempio di grafo $\mathcal{G}$



Notazione:

nodi del grafo $\mathcal{G}$: *A, B, C, D, E, F, G, H, I, L*

arco che congiunge i nodi A e B: *(A, B)*

DEF. Due nodi di un **grafo** $\mathcal{G}$ si dicono **adiacenti** se esiste un arco che li congiunge (altrimenti si dicono **non adiacenti**).

DEF. In un **grafo** $\mathcal{G}$ si definisce **cammino** una successione di nodi adiacenti che **non** contiene due volte lo stesso arco.

DEF. In un **grafo** $\mathcal{G}$ si definisce **cammino semplice** un cammino comprendente tutti nodi distinti ad eccezione eventualmente del primo e dell'ultimo (che possono coincidere).
Nel caso essi coincidano il cammino semplice prende il nome di **ciclo**.

Nell'esempio sopra riportato:
la successione di nodi B, C, E, H, I, E, D *è un **cammino(non semplice)** tra i nodi **B** e **D***
la successione di nodi A, B, E, G, I, L *è un **cammino semplice** tra i nodi **A** e **L***
la successione di nodi C, E, H, I, G, F, C *è un **cammino semplice ciclico** ossia un **ciclo***
la successione di nodi B, C, E, D, E,H ***NON** è un **cammino***

N.B. In un grafo 𝒢 possono esistere diversi cammini semplici che uniscono gli stessi due nodi

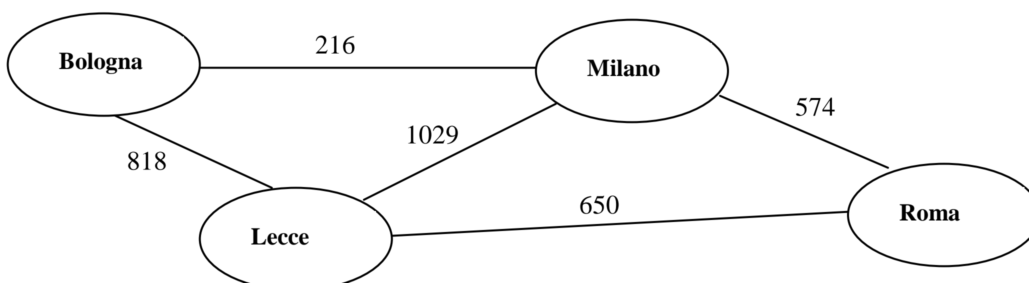
la successione di nodi A, B, C, E *è un **cammino semplice** che unisce i nodi A ed E*
la successione di nodi A, B, D, E *è un **cammino semplice** che unisce i nodi A ed E*
la successione di nodi A, B, E *è un **cammino semplice** che unisce i nodi A ed E*

DEF. Un **grafo 𝒢** si definisce **connesso** se per ogni coppia di nodi considerata esiste sempre almeno un cammino che li congiunge.
In altri termini se consideriamo due suoi nodi arbitrari è sempre possibile congiungerli mediante almeno un cammino.

DEF. Se ad ogni arco di un **grafo 𝒢** associamo un valore o *peso* il grafo prende il nome di **grafo pesato.**

DEF. Un **grafo 𝒢** si definisce **orientato** se ogni arco è dotato di orientamento.
Tale orientamento indica la relazione logica che esiste tra i dati contenuti nei due nodi congiunti dall'arco. In tal caso al posto dei segmenti semplici per rappresentare gli archi vengono utilizzate le frecce con uno o due punte

Esempio di grafo 𝒢 connesso e pesato



DOMANDA: <u>A che servono i grafi?</u>

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 $\mathcal{G}$ pesato con le località da servire (che possono essere rappresentate come **nodi**), 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 $\mathcal{G}$ ossia **il cammino semplice con peso minimo** che congiunge tutti i nodi di un grafo $\mathcal{G}$.
Un grafo è particolarmente adatto a rappresentare anche **automi**, **reti di trasporto**, **reti elettriche**, reti dati, etc.

GLI ALBERI

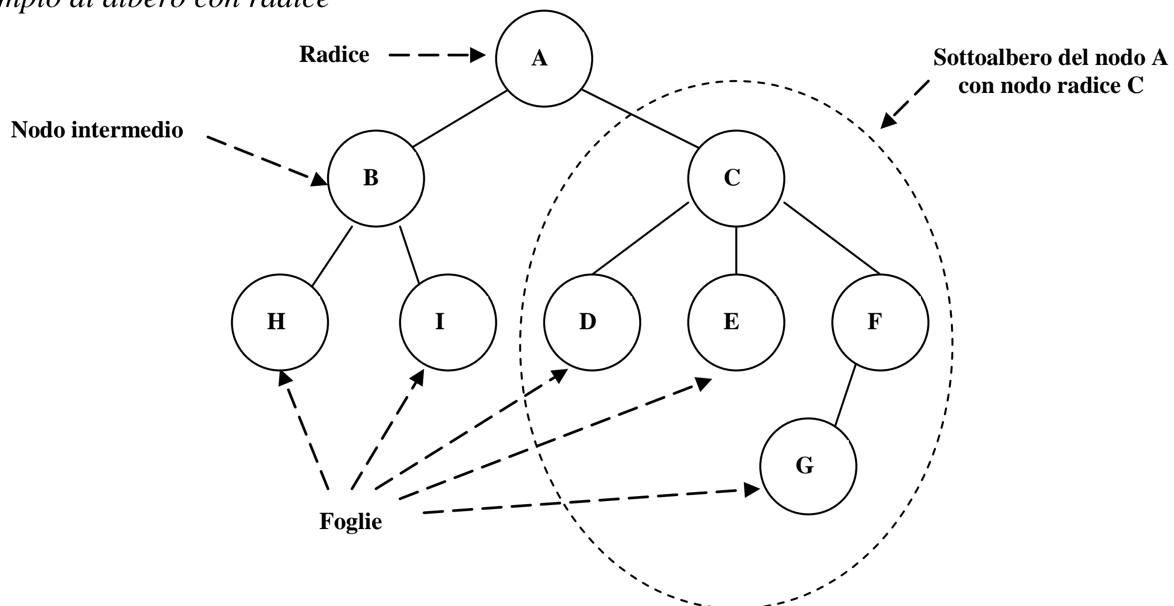
DEF: Un **albero** è un particolare tipo di grafo **connesso e senza cicli**

Una struttura dati astratta non lineare se è un **albero** gode delle seguenti tre proprietà:

1. se un albero possiede **n nodi** allora conterrà **n-1 archi**
2. due nodi qualsiasi in un albero sono sempreconnessi da **un unico cammino semplice**;
3. **rimuovendo un qualsiasi arco** dell'albero, **la struttura dati risultante non è più connessa** e 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



<u>*Notazione:*</u>

radice dell'albero:	*A*
foglie dell'albero:	*H, I, D, E, G*
nodi intermedi dell'albero:	*B, C, D, E, F*

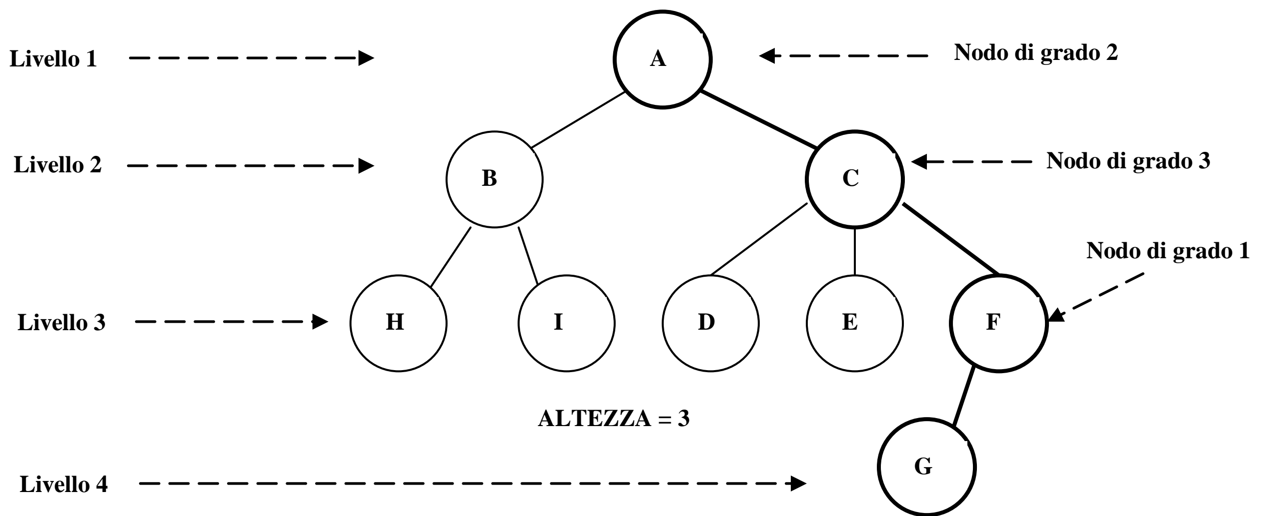
Gli alberi che considereremo saranno **alberi ordinati** (nei quali si possono distinguere gli elementi primo, secondo, … n-esimo secondo un determinato criterio) alberi per i quali <u>l'ordinamento dei sottoalberi</u> è importante.

DEF. In un albero si dice **grado di un nodo** il numero dei sottoalberi di quel nodo.

DEF. In un albero si dice **livello di un nodo** il numero di nodi attraversati da un cammino dalla radice al nodo. Il livello è uguale <u>ad 1</u> se il nodo è <u>la radice</u> dell'albero.

DEF. Si dice **altezza o profondità** di un albero la lunghezza del cammino più lungo esistente tra nodo radice e nodi foglie.

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

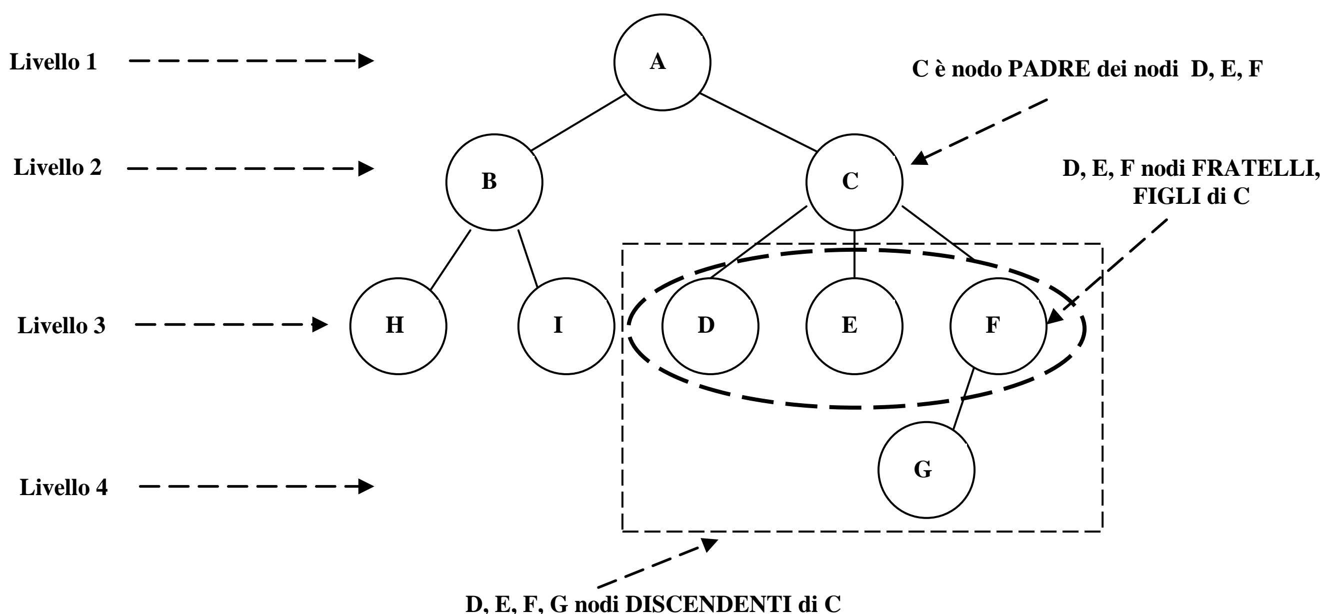


N.B. E' possibile parlare di alberi in termini più **"genealogici".**

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** (esempi: albero genealogico familiare e classificazione di animali e piante).

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:
- esiste un solo nodo distinto detto **radice**;
- i rimanenti nodi sono ripartiti in n insiemi tutti disgiunti $\mathbf{T_1, T_2, \ldots. T_n}$ con $\mathbf{n \geq 0}$ 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.

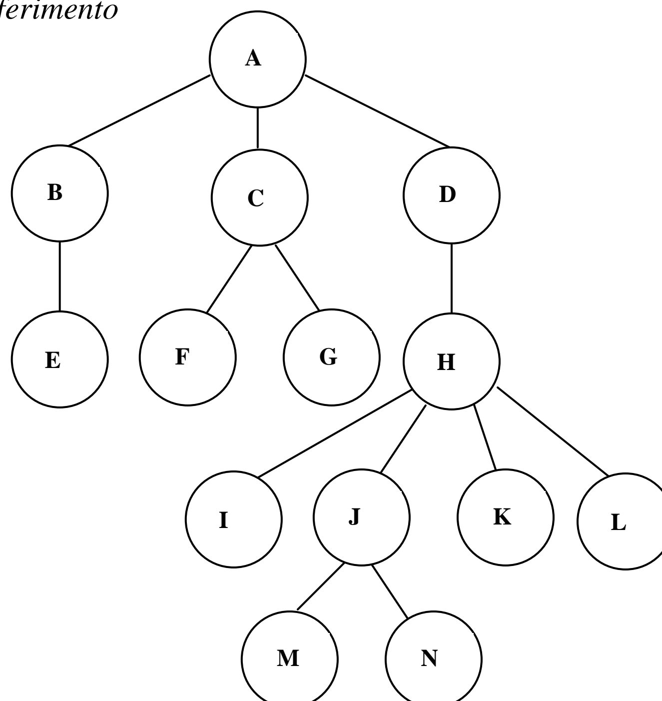
Notazione:
esame di un nodo:
significa accedere al nodo ed estrarre le informazioni contenute in esso;

attraversare o visitare un albero:
significa esaminare sistematicamente in un ordine appropriato tutti i nodi di un albero in modo che ciascun n odo 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 o differito (post-order);**

Esempio: albero di riferimento



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

Il processo risolutivo RICORSIVO di questo algoritmo si articola nei seguenti passi:

1) si esamina la radice;
2) se il numero n dei sottoalberi della radice è uguale a zero ci fermiamo, altrimenti si prosegue come al successivo punto 3;
3) attraversa il primo sottoalbero (in ordine anticipato o pre-order);
4) attraversa il secondo sottoalbero (in ordine anticipato o pre-order);
…………………………………………………
n+2) attraversa l'n-esimo sottoalbero (in ordine anticipato o pre-order).

Quindi l'attraversamento in ordine anticipato da 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 articola nei seguenti passi:

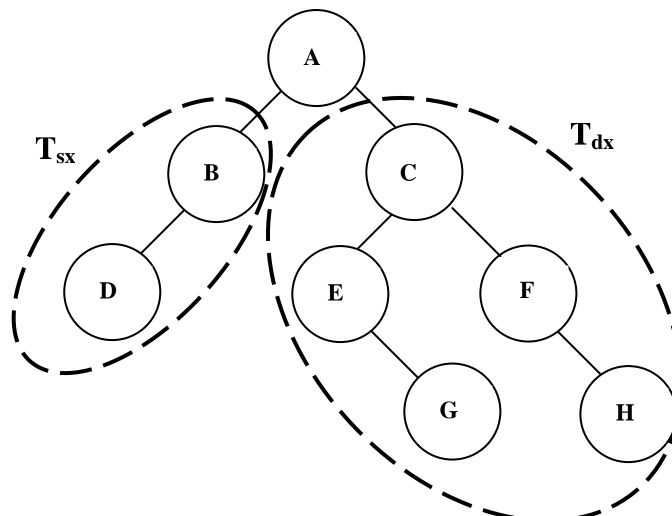
se il numero dei sottoalberi della radice n ≥ 0

1) attraversa il primo sottoalbero (in ordine posticipato o post-order);
2) attraversa il secondo sottoalbero (in ordine posticipato o post-order);
…………………………………………………
n) attraversa l'n-esimo sottoalbero (in ordine posticipato o post-order);
n+1) esamina la radice

Quindi l'attraversamento in ordine posticipato da la seguente sequenza di nodi:
E, B, F, G, C, I, M, N, J, K, L, H, D, A

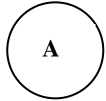
GLI ALBERI BINARI

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

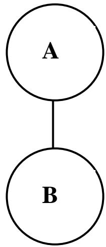


DEF 2 (RICORSIVA). Un **albero** si dice **binario** se:
- ha solo la radice (allora si dice **vuoto**);
- la radice ha al più due sottoalberi binari (rispettivamente *sottoalbero sinistro* e *sottoalbero destro*).

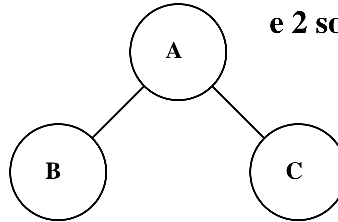
<u>Secondo la definizione appena data sono alberi binari:</u>



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.