ALGORITMI COMPLETO

Monday, 11 November 2024 09:38

APPUNTI 26/09

Introduzione alla materia: Gli algoritmi
Gli algoritmi sono il cuore della **computer science** e della **programmazione**. Comprenderli e saperli progettare è essenziale per risolvere problemi in modo efficiente e per ottimizzare il funzionamento di software e sistemi.
Importanza degli algoritmi
Gli algoritmi sono usati ovunque:

- Nella **navigazione online** (Google usa algoritmi di ricerca per trovare risultati rilevanti).
- Per calcolare il percorso più breve in una mappa (come fa Google Maps).
- Nei **motori di raccomandazione** (Netflix, YouTube, Spotify).
- Nell'**intelligenza artificiale**, per apprendere dai dati e prendere decisioni.

Obiettivi dello studio degli algoritmi

- **Comprendere come risolvere problemi** in modo strutturato e logico.
- **Valutare l'efficienza** degli algoritmi (tempo di esecuzione, uso della memoria).
- **Progettare nuove soluzioni** per problemi reali.
- **Ottimizzare risorse**, specialmente in applicazioni complesse.

Categorie di problemi risolvibili con algoritmi

- **Ricerca**: Trovare un elemento in un insieme di dati (ad esempio, ricerca binaria).
- **Ordinamento**: Disporre i dati in un certo ordine (es. QuickSort, MergeSort).
- **Ottimizzazione**: Trovare la soluzione migliore per un problema (es. problemi di minimo costo).
- **Grafi**: Calcolare percorsi, connettività, o trovare il ciclo minimo in un grafo.

Classificazione degli algoritmi
Gli algoritmi possono essere classificati in base al loro funzionamento:

- **Iterativi**: Basati su cicli (es. ricerca lineare).
- **Ricorsivi**: Risolvono il problema richiamando sé stessi su sotto-problemi (es. QuickSort).
- **Greedy**: Prendono decisioni ottimali localmente per arrivare alla soluzione globale (es. algoritmo di Kruskal).
- **Divide et Impera**: Dividono il problema in sottoproblemi più semplici (es. MergeSort).
- **Dinamici**: Risolvono problemi complessi memorizzando i risultati di sotto-problemi già risolti (es. algoritmo di Fibonacci ottimizzato).

Esempio semplice: Algoritmo per preparare un tè

1. Riempi un bollitore con acqua.
2. Accendi il bollitore.
3. Metti una bustina di tè in una tazza.
4. Quando l'acqua bolle, versala nella tazza.
5. Lascia in infusione per 3 minuti.
6. Rimuovi la bustina e servi.

Questo esempio mostra come un problema quotidiano possa essere suddiviso in una sequenza di passi chiari e finiti.

APPUNTI 27/09

Soluzione ricorsiva, ovvero una funzione che richiama sé stessa.
Esempio: il fattoriale è un algoritmo ricorsivo.

Facendo il return libera la pila e occupa meno spazio. Richiama quindi la stessa funzione ma ogni volta che avviene il return si libera lo spazio come se ripartisse dall'ultimo stato della funzione dimenticandosi il precedente.

ESEMPIO

```
N = 10
X={x1,..,xn}
X/2
X1={x1,…, xn/2}
X2={x(n/2),…,xn} // n dispari escludere l'ultimo valore, se pari includere

If peso(X1) > peso(X2) then
Return "la moneta falsa è nel primo gruppo"
X1 = sospetto
Else
Return "la moneta falsa è nel secondo gruppo"
X2 = sospetto

// Considerando X1

X1/2 // ora la lunghezza del campione è 5

X3 = {x1,…, xn/4}
X4 = {x(n/4+1),…,x(n/2-1)} // x(n/2) non incluso poiché dispari

If peso (X3) == peso (X4) then
Return xn/2

If peso(X3) > peso(X4) then
Return "la moneta falsa è nel primo gruppo"
Else "la moneta falsa è nel secondo gruppo"

//Considerando X3

X3/2 //ora il campione è grande 2
X5 = [x1]
X6 = [xn/4]

If peso(X5) > peso(X6) then
Return x1
Else
Return xn/4
```

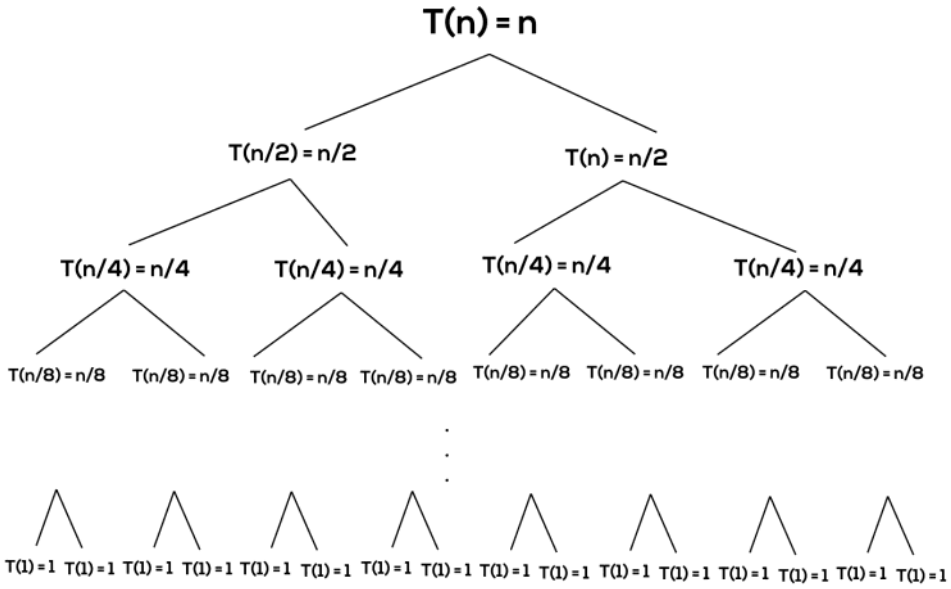
Idea corretta ma non ricorsiva, bisogna creare la funzione da richiamare

Soluzione professore

```
Alg3 //Creo una funzione, ovvero alg3, e la richiamo

1- if (|X| = 1) then //Caso base
             return unica moneta in X

2- dividi X in due gruppi di uguale dimensione k= |X|/2 e se dispari una ulteriore
moneta y

3- if peso(X1) = peso(X2) then
                           return y

4- if peso(X1) > peso(X2) then
                            return Alg3(X1)
                                        else return Alg3(X2)
```

//Creare sempre soluzione generica, sarà poi il programmatore a sviluppare il codice

L’algoritmo è corretto ed è efficiente perché nella migliore delle ipotesi effettua una pesata
Per sapere n nel caso peggiore bisogna disegnare l’albero di ricorsione ovvero un grafo non orientato.



L’albero ha alcune proprietà note:

- Grafo non diretto, connesso, aciclico
- Un albero binario è un albero dove ogni nodo ha al massimo due figli
- Le foglie dell’albero non hanno figli *(sono sempre alla fine dell’albero)*

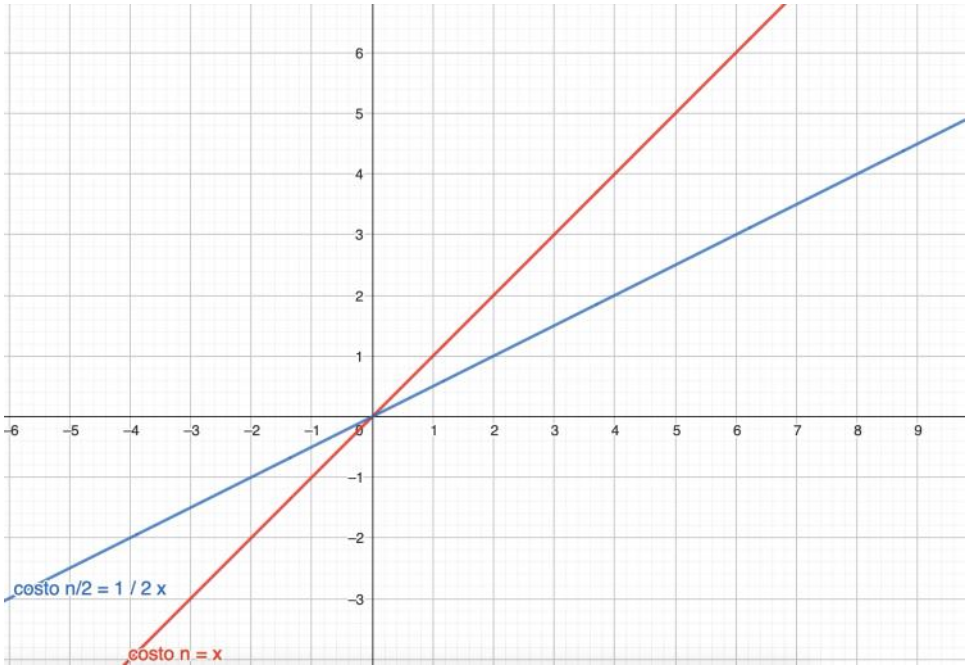
Profondità: distanza tra un nodo qualsiasi e la radice
Altezza: è la profondità massima che può raggiungere un nodo

Foglie k= 2^h dove h= altezza se l’albero è pieno di foglie. Altrimenti sarà un po’ meno di h.

Supponendo k=2^3=8 allora se volessi sapere il valore di h allora log8 in base 2 è uguale a h. (8= n foglie)

Ritornando all’esempio precedente, ogni chiamata ricorsiva rappresenta un nodo dell’albero. Per tanto, per sapere quante pesate effettueremmo nel caso peggiore. Considerando l’insieme X da 8 monete la mia radice avrà un figlio (dim 4) a sinistra X1 e uno a destra X2, a seconda di quale dei due risulta più pesante, quel nodo avrà a sua volta due figli (dim 2) a destra e a sinistra, ripeto nuovamente e anche questo nodo avrà due figli (dim 1) questa volta foglie. Dunque nel caso peggiore effettuerei 3 pesate (dopo h chiamate ricorsive, h= log in base 2 di n)

N solitamente coincide con la dimensione dell’istanza ma varia da algoritmo in algoritmo



Quanto è più veloce alg3 rispetto agli altri due algoritmi?

Assumendo che ogni pesata duri un minuto con alg1 se n= 100 impiego un'ora e quaranta minuti circa, cinquanta minuti con alg2 e sei minuti con alg3.

All'aumentare di n alg3 è sempre più veloce e dunque più efficiente rispetto ad alg1 e alg2.

ALG4

Dividendo in tre gruppi anziché due

```
Alg4(X)

If(|X|=1) then return unica moneta in X

Dividi X in tre gruppi di uguale dimensione siano X1 e X2, di dimensione bilanciata
X1, X2, X3

If peso(X1) = peso(X2) then
                            return Alg4(X3)

If peso(X1) > peso(X2) then
                            return Alg4(X1)
                     else
                            return Alg4(X2)
```

È corretto
Pesate nel caso peggiore -> log in base 3 di n (in base di 3 perché ogni nodo ha 3 figli avendo deciso di dividere in 3 gruppi)

È migliore rispetto ad Alg3

Se disegnassi le funzioni

f(x)= x/2
f(x)= log in base 3 di x
f(x)= log in base 2 di x

Vedrei come si comportano le tre funzioni (corrispondenti ad Alg2, Alg4 e Alg3) e posso dire chi è

la più efficiente. Ale 4 è l'algoritmo più efficiente per risolvere questo problema.

Esempio: **i numeri di Fibonacci**

La sequenza di Fibonacci nasce dall'idea di osservare la crescita di una popolazione di conigli sotto determinate condizioni

Partendo da una coppia di conigli su un'isola deserta, quanti conigli avrò nell'anno n?

- Ogni coppia produce una nuova coppia
- La gestazione dura un anno
- I conigli iniziano a riprodursi solo al secondo anno
- I conigli sono immortali

Dal secondo anno in poi, quindi, la coppia 0 inizia a riprodursi diventando matura, dunque solo al quarto anno avremo 2 coppie di conigli fertili.

Quante coppie ci sono nell'anno n?

Sommo:

- Tutte le coppie dell'anno precedente, F(n-1)
- Una nuova coppia per ogni coppia presente due anni prima F(n-2)

Escluso il caso base, ovvero la coppia 0.

Fn = F(n-1)+F(n-2) se n maggiore/uguale a 3, altrimenti 1 se n = 1, 2 (il primo anno i conigli non sono maturi, al secondo diventano maturi ed inizia la gestazione, al terzo nasce la nuova coppia)

Es: F18 = 2584

Ma come lo calcolo?

Cerchiamo l'approccio numerico, ovvero *una funzione matematica* che calcoli direttamente i numeri di Fibonacci

Cerchiamo la Sezione Aurea indicata con phi

$$\mathrm{Fn} \;=\; 1/\mathrm{rad}(5)(\mathrm{phi}^{\mathrm{n}} - \mathrm{phi}^{\mathrm{n}\hat{}})$$

Algoritmo fibonacci1(intero n) -> intero *//ovvero da un intero e restituisce un intero*

$$\mathrm{return\ Fn} \;=\; 1/\mathrm{sqrt}(5)(\mathrm{phi}^{\mathrm{n}} - \mathrm{phi}^{\mathrm{n}}\hat{})$$

Questo primo algoritmo non funziona al crescere di n, arrivati a F18 smette di funzionare

Dunque per avere un risultato corretto è ottimale lavorare direttamente sull'intero

APPUNTI 3/10

L'algoritmo per essere corretto ha dunque bisogno di una forma ricorsiva. Questo algoritmo altro non è che la traduzione in pseudocodice di quanto detto da Fibonacci.

Il mio caso base è $n = 1, 2;$

Se $n \geq 3$

Return nome algoritmo

```
Algoritmo fibonacci2(intero n) -> intero
     If(n ≤ 2) then return 1
          Else rerturn fibonacci2(n-1) + fibonacci2(n-2)
```

- È corretto? *Si*
- È efficiente?

Un modello di calcolo rudimentale: ogni linea di codice costa un'unità di tempo

Calcoliamo le linee di codice utilizzate

Se n ≤ 2 eseguo solo una linea di codice

Se n = 3
Fibonacci2(3): 2 linee
Fibonacci(2): 1 linea
Fibonacci2(1): 1 linea

Relazione di ricorrenza

T(n) = #linee di codice eseguite (nel caso peggiore) dall'algoritmo su input n

In ogni chiamata si esguono due linee di codice, oltre a quelle eseguite nelle chiamate ricorsive

$$T(n) = 2 + T(n-1) + T(n-2)$$

Ciascun nodo dell'albero può essere etichettato con il valore 2 (if e else, le righe di codice necessarie per ogni chiamata ricorsiva), *come mi torna in aiuto l'albero di ricorsione?*

Il numero di foglie è uguale al valore di F(n), ovvero il risultato del problema che ci eravamo posti all'inizio.

- Lemma 1.1 il numero di foglie di T_n è F_n
- Lemma 1.2 il numero di nodi interni di Tn è (F_n-1)

Ogni foglia dell'albero T(n) corrisponde al caso base

Per ogni foglia eseguo 1 linea di codice

Per ogni nodo interno eseguo due linee di codice

In conclusione le linee di codice eseguite sono $F_n + 2(F_n-1) = 3F_n-2$

Ma questo algoritmo è lento, *perché?*

Perché non solo i valori si ripetono, ma si ripetono pure in altri rami

```
Algoritmo fibonacci3(intero n) -> intero
          //Sia Fib un array di n interi
               Fib[1] <- Fib[2] <- 1

          For i = 3 to n do
               Fib[i] <- Fib[i-1] + Fib[i-2]
                    Return Fib[n]
```

Sostanzialmente il for trova il valore i-esimo nell'array e i valori F(n-1) e F(n-2), rifacendosi quindi alla formula matematica iniziale

L'algoritmo è corretto? Si
È efficiente?

Fib[1, 1, 2, 3, 5, 8], le corrispettive posizioni sono (1, 2, 3, 4, 5, 6)

i= 3 (il ciclo for inizia al valore 3)

$$T(n) = n - 1 + n - 2 + 3 = 2n, \qquad n > 1$$

È 38 milioni di volte più veloce, è una funzione lineare

Per quanto sia veloce, però, quanta memoria occupa?

COMPLESSITA DI UN ALGORITMO

La complessità temporale di un algoritmo cresce al crescere della dimensione dei dati in input, ovvero la quantità dei dati

Il tempo di esecuzione di un algoritmo è strettamente legato al funzionamento di un algoritmo.

Non si considerano gli aspetti hardware, né il compilatore o il linguaggio di programmazione utilizzato, ricordiamo che l'algoritmo va scritto in maniera generica, affinché sia leggibile ed efficiente indipendentemente da questi aspetti.

Il calcolo della complessità computazionale in informatica si chiama Analisi Asintotica, tuttavia questo fa riferimento ad aspetti della matematica, principalmente allo studio di funzioni, mentre il tempo di esecuzione non lo è.

Per effettuare l'analisi asintotica devo prima trasformare il tempo di esecuzione dell'algoritmo in una funzione T(n) in funzione dei dati in input, T(n) misura il numero di comandi eseguiti dall'algoritmo.

Data un'istanza di dimensione n, nel caso peggiore l'algoritmo ha una complessità temporale

$$O(f(n)) se\ T(n) = O(f(n))$$

Dove n è il numero delle righe eseguite (dimensione n dei dati) mentre f(n) è un limite superiore del tempo di esecuzione dell'algoritmo nell'ipotesi peggiore.

Per l'analisi computazionale si utilizza sempre il caso peggiore

Quando utilizzo un ciclo for o while devo sempre considerare n-1 perché comunque l'operazione verrà ripetuta n volte, ovvero O(n)

MODELLI DI CALCOLO E METODOLOGIE DI ANALISI
APPUNTI 4/10

- F(n) = tempo di esecuzione/occupazione di memoria di un algoritmo su input di dimensione n

- La notazione asintotica è un'astrazione utile per descrivere l'ordine di grandezza f(n) ignorando i dettagli non influenti, come costanti moltiplicative e termini di ordine inferiore

Cosa è O grande?

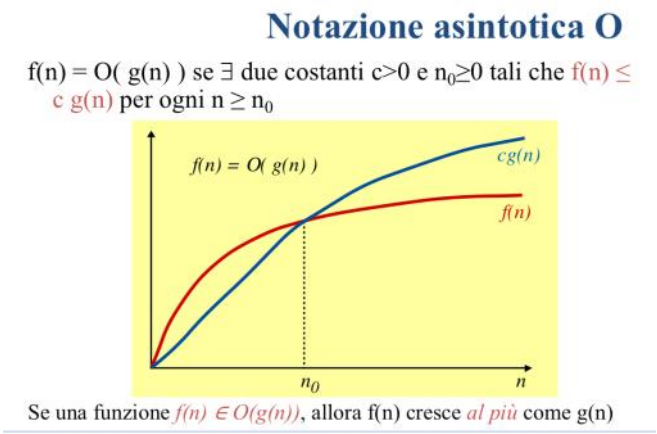
F(n) -> quella calcolata
G(n) -> quella che associamo in notazione asintotica

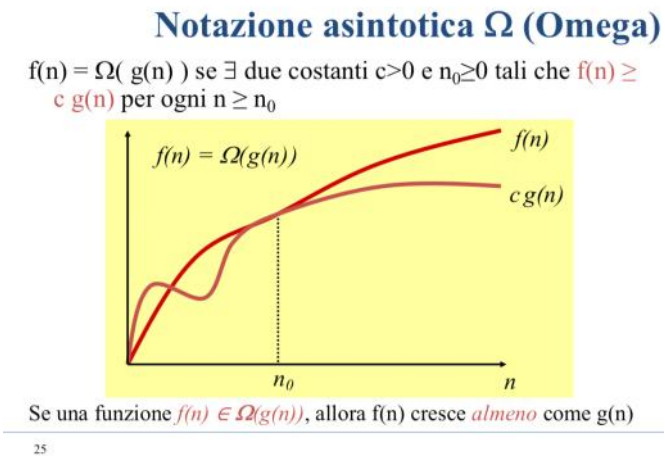
Quindi una funzione di cui sappiamo il costo, e una che troviamo dentro le parentesi di O grande.

C è una costante da me scelta proprio per vedere il comportamento.

C è maggiore di zero, n è positivo

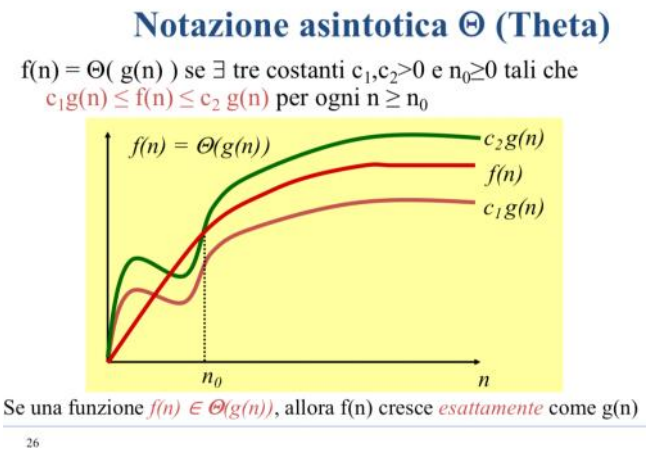
Cg(n) da un certo punto n0 sarà maggiore di f(n)





Se O grande è una stima per eccesso, Omega è una stima per difetto. Sono di fatto limiti superiori ed inferiori, O serve per stimare il caso peggiore, Omega invece indica il valore oltre il quale non possiamo andare perché f(n) cresce almeno come g(n).

Esiste un terzo caso, dove Omega e O si "fondono". Considerando due costanti c1 e c2, che possono anche essere uguali fra loro ma maggiori di zero e f(n) è compreso tra c1g(n) e c2g(n) per ogni n maggiore o uguale a n0 allora f(n) cresce esattamente come g(n).



Se omega e O non sono contemporaneamente verificate allora theta non può esistere

F(n) = theta(g(n))
Se e solo se f(n) = O(g(n)) e f(n) = omega(g(n))

Quindi theta è contenuto in entrambe le condizioni, banalmente:
Se f(n) si trova tra g(n) minore o uguale a 5 e g(n) maggiore o uguale a 5 allora f(n) è esattamente 5

Consideriamo $f(n) = 3n^2 + 10n$
Verifichiamo che $f(n) = O(n^2)$

Se scelgo c = 4 e n0 = 10

$$3n^2 + 10n \leq 4n^2$$

A me importa sapere cosa succede quando l'andamento delle curve si stabilizza

F(n)= O(n^3)

Dovrebbe essere una costante moltiplicata al cubo, sovrastimando sicuramente di molto.

Rivediamo l'esempio applicato a Omega

Consideriamo $f(n) = 3n^2 + 10n$
Verifichiamo che $f(n) = Omega(n^2)$

Analizzando i grafici delle funzioni determino che tipo di relazione si verifica ma se n fosse al cubo non si verificherebbe la funzione omega, non esiste valore al quadrato maggiore di un valore al cubo

Questo criterio è applicabile anche a theta, la condizione si verifica al quadrato ma al cubo sarà verificata solo per O grande ma non per omega.

MODELLO DI CALCOLO

- Il modello di calcolo definisce l'insieme di operazioni per risolvere un problema
- Ci serve per definire la complessità dell'algoritmo

Le linee di codice non impattano sulla complessità quanto più il loro contenuto.

Il modello di calcolo è una macchina astratta che definisce l'insieme delle operazioni ammissibili ed eseguibili durante una computazione e ne specifica i relativi costi (in termini di tempo e spazio)

Modello di calcolo: RAM (Random Access Machine) è un tipo particolare di macchina a registri (locazione di memoria ad accesso diretto)

- La RAM è definita da: un nastro di ingresso e uno di uscita, ove saranno scritti l'input e l'output
- Una memoria ad accesso diretto: strutturata come un array (di dimensione infinita) in cui ogni cella può contenere un valore intero arbitrariamente grande
- Un programma che consiste di istruzioni di input/output, operazioni aritmetiche, e di accesso o modifica del contenuto della propria memoria
- Un registro detto accumulatore
- Un registro detto contatore delle istruzioni

La RAM è un'astrazione dell'architettura di Von Neumann, ovvero l'architettura che regola il buon funzionamento di un calcolatore

RICERCA SEQUENZIALE 10/10

Un **algoritmo sequenziale** è un tipo di algoritmo in cui le operazioni vengono eseguite una dopo l'altra, in un ordine prestabilito, senza interruzioni o salti condizionali. Ogni passo dell'algoritmo deve essere completato prima che il successivo possa iniziare.
Ecco alcune caratteristiche principali di un algoritmo sequenziale:

1. **Ordine fisso**: Le istruzioni devono essere eseguite in sequenza, nell'ordine in cui sono scritte.
2. **Deterministico**: Il risultato finale dipende solo dai dati di input e dall'ordine delle istruzioni, senza variazioni basate su scelte o condizioni.
3. **Semplicità**: È il tipo di algoritmo più semplice e non richiede logica complessa come cicli o condizioni.

Ad esempio, un algoritmo che somma due numeri e poi moltiplica il risultato per un terzo numero sarebbe un algoritmo sequenziale.

Data una lista non ordinata L, x appartiene a L?

Array: collezione di elementi deindicizzata, tutti elementi dello stesso tipo. Posso usare l'indice perché sono celle contigue. Spostarsi di una cella significa spostarsi di indirizzo di memoria

Lista: mentre l'array è una struttura statica, ovvero mi ritaglio 10 celle contigue, ad esempio, uno spazio di memoria dedicato. Le liste, come in Python, possono cambiare dimensione dinamicamente. Puoi aggiungere, rimuovere o modificare elementi in qualsiasi momento. I dati sono eterogenei.

Dato l'array abc trasformare l'array in ac

```
Ric SeQ (arrayL, int X) -> booleano

     for i = 1 to N
          if(L[i] == x)
```

```
            Return TRUE
```

Es:
L[1, 2, 3, 4], x=4
Appena il for arriva al 4 stampa true, se x fosse stato uguale a 6 sarebbe stato stampato false verificando la condizione dell'else

Il costo è n, in notazione asintotica O(n) ovvero il caso peggiore quindi x=4 o x=6, l'if o l'else si verificano comunque alla fine del ciclo for arrivando all'ultimo confronto.

X è in prima posizione T(n) best =1
X è in ultima posizione T(n) worst = n
T(n) average calcolo delle probabilità

A noi interessa solo il caso peggiore.

Ricerca binaria
Ricerca di un elemento x di un array ordinato

Array [1, 2, 3, 4, 5, 6, 7, 8, 9, 11, 13]
X = 12

Sicuramente in un array L ordinato cercherò dodici tra 9, 11 e 13. So già che 12 > 9 quindi tutto quello che è minore di 11 non mi interessa.

Questo algoritmo funziona solo sugli array poiché godono di deindicizzazione.

```
Algoritmo RicercaBinariaIterativa(array L, elemento x) -> booleano
a <- 1
b <- lunghezza di L
     while(L[a+b]/2]!= x) do
          m <- (a+b)/2 //elemento centrale

          if (L[m] > x then b <- m-1
               else a <- m+1
          if (a>b) then return non trovato

return trovato //se la condizione del ciclo while è verificata da subito
```

Nel caso peggiore di prosegue finché a = b, cioè $n/2^i = 1$, ovvero $i=\log_2 n$ quindi O(log n)

ANALISI DI ALGORITMI RICORSIVI

Lo stesso algoritmo si può scrivere in forma ricorsiva

```
RicercaBinariaRicorsiva (Array A, int x, a, b)
          if (A[m] > x) //elemento centrale  m <- (a+b)/2
               RicBinRic(A, x, a, b-1)
```

PSEUDOCODICE:

```
Algoritmo RicercaBinariaRicorsiva(array L, indice i, indice j, elemento x)
if(i>j) then return non trovato
m <- (i+j)/2 //elemento centrale
if(L[m]=x) then return trovato
if(L[m]>x) then return RicercaBinariaRicorsiva(L, i, m-1, x)
          else return RicercaBinariaRicorsiva(L, m+1, j, x) //rivedere sul libro è
          errato
```

VERSIONE LIBRO

```
Algoritmo RicercaBinariaRicorsiva(array L, elemento x) -> booleano
n <- lunghezza di L
if(n=0) then return non trovato      //caso base
i = (n/2)      //punto medio nell'array di dimensione n
```

if(L[i]= x) then **return** trovato *//lo trovo subito se fortunatamente dovessero coincidere*
else if(L[i] > x) then **return** RicercaBinariaRicorsiva(x, L[1; i-1]) *//riesegui l'operazione nella frazione inferiore dell'array*
else return RicercaBinariaRicorsiva(x, L[i+1; n]) *//altrimenti la riesegue nella frazione superiore dell'array*

RELAZIONI DI RICORRENZA

T(n) = 2+T(n/2)

LEZIONE 25/10

Strutture dati elementari

Gestione di collezione di oggetti

Tipo di dato:

- Specifica delle operazioni di interesse su una collezione di oggetti (es. inserisci, cancella, cerca)
- Specifica cosa un'operazione deve fare
- NON specifica come l'operazione deve essere realizzata
- NON specifica come gli oggetti della collezione debbano essere organizzati per garantire l'efficienza delle operazioni

```
tipo Dizionario:
Dati:
     Un insieme S di coppie(elem, chiave)
Operazioni:
     Insert(elem e, chiave k)
          Aggiunge a S una nuova coppia (e, k)
     Delete(chiave k)
          Cancella da S la coppia con chiave k
     Search(chiave k) -> elem
          Se la chiavae k è presente in S restituisce l'elemento e ad essa
          associato, e null altrimenti
```

Struttura dati

Organizzazione dei dati che permette di supportare le operazioni di un tipo di dato usando meno risorse di calcolo possibile

Per rappresentare una collezione di oggetti utilizziamo strutture indicizzate (array) e strutture collegate (record e puntatori)

Dizionario realizzato con un array ordinato:

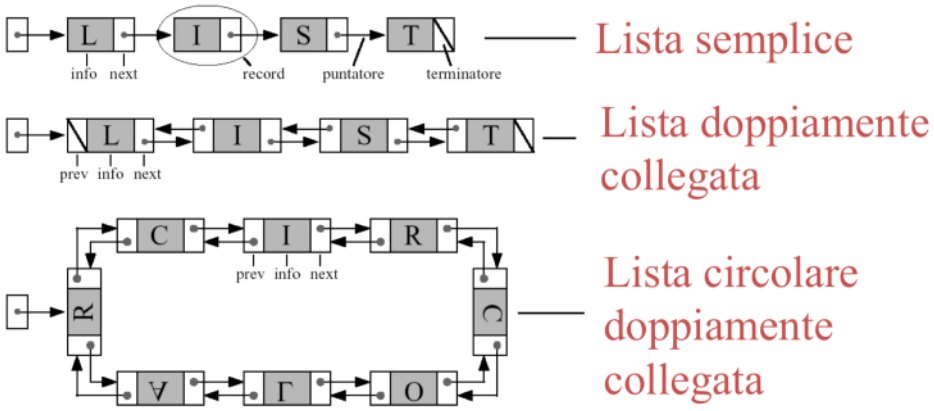
```
Classe ArrayOrdinato implementa Dizionario:
Dati:
     Un array S di dimensione n contenente coppie (elem, chiave).
Operazioni:
     insert(elem e, chiave k)
     Rialloca l'array S aumentandone la dimensione n di uno; cerca il più piccolo
     indice i tale che k <= S[i].chiave e pone S[j] <- S[j-1] per ogni j da n-1 a
     i+1; infine, pone S[i] <- (e, k)
     delete(chiave k)
     Trova l'indice i della coppia con chiave k in S e pone S[j] <- S[j+1] per ogni
     j da i a n-2; infine rialloca l'array S diminuendone la dimensione n di uno
     search(chiave k) -> elem
     Esegue l'algoritmo di ricerca binaria su S per verificare se S contiene la
     chiave k. Se la ricerca ha successo restituisce l'elemento e associato alla
     chiave, altrimenti restituisce null.
```

RECORD E PUNTATORI

- I costituenti di base di una struttura collegata sono i record
- Ogni record è numerato e contiene al suo interno un oggetto della collezione
- Ad ogni record è associato un indirizzo globale nell'ambito del programma (all'array è associato un indice locale)
- Gli indirizzi dei record non sono consecutivi (gli indici degli array si)
- Per mantenere una collezione di record, ciascuno dovrà contenere uno o più indirizzi di altri record della collezione
- I collegamenti tra record sono realizzati mediante puntatori

Proprietà 1: (forte) è possibile aggiungere o togliere record ad una struttura collegata

Proprietà 2: (debole) gli indirizzi dei record di una struttura collegata non sono necessariamente consecutivi



TECNICHE DI RAPPRESENTAZIONE DEI DATI

Rappresentazioni indicizzate:

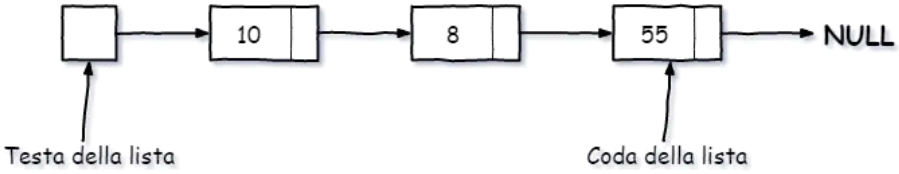
- Pro: accesso diretto ai dati mediante indici
- Contro: dimensione fissa (riallocazione array richiede tempo lineare)

Rappresentazioni collegate:

- Pro: dimensione variabile (aggiunta e rimozione record in tempo costante)
- Contro: accesso sequenziale ai dati

Tipo di dato Coda

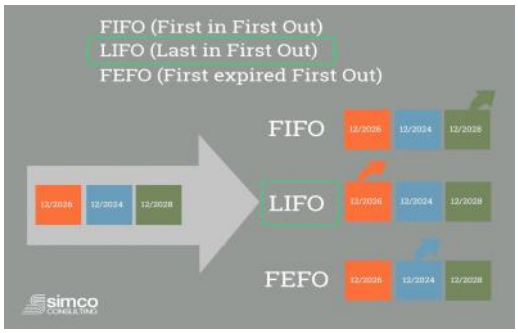
- Tipo di dato strutturato secondo una regola di accesso FIFO - First In First Out
- Gli inserimenti (ENQUEUE) aggiungono elementi alla fine delle sequenza
- Le cancellazioni (DEQUEUE) rimuovono sempre il primo elemento
- In una coda gli accessi riguardano entrambe le estremità




```
Tipo Coda:
dati:
       Una sequenza S di n elementi
operazioni:
       isEmpty() -> result
      Restituisce true se S è vuota, false altrimenti
       enqueue(elem e)
      Aggiunge e come ultimo elemento di S
       dequeue() -> elem
      Toglie da S il primo elemento e lo restituisce
       first() -> elem
      Restituisce il primo elemento di S (senza toglierlo da S)
```

Tipo di dato Pila

- Tipo di dato strutturato secondo una regola di accesso LIFO - Last In First Out
- Gli inserimenti (PUSH) aggiungono elementi alla fine delle sequenza
- Le cancellazioni (POP) rimuovono sempre l'ultimo elemento
- In una pila gli accessi riguardano solo l'ultimo elemento




```
Tipo Pila:
Dati:
     una sequenza S di n elementi
Operazioni:
     isEmpty() -> result (condizione booleana)
     Restituisce true se S è vuota, false altrimenti
     push(elem e)
     Aggiunge e come ultimo elemento di S
     pop() -> elem
     Toglie da S l'ultimo elemento e lo restituisce
     top() -> elem
     Restituisce l'ultimo elemento di S (senza toglierlo da S)
```

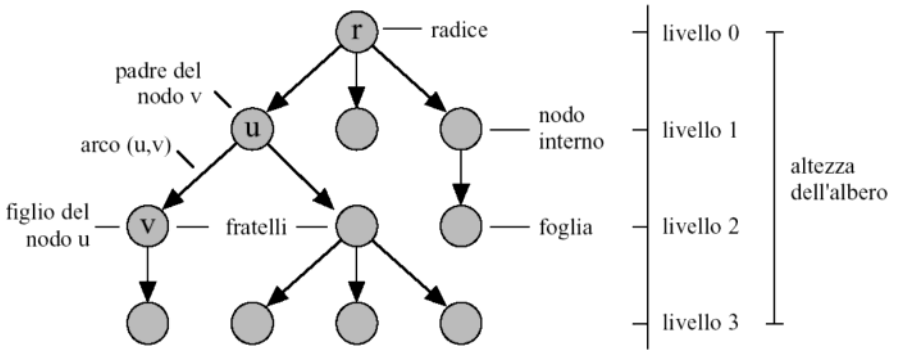
Pila: Si usa in algoritmi che richiedono un contesto “annidato” o che devono memorizzare l'ordine di ritorno.
Per esempio:

- **Backtracking**: come nella risoluzione di problemi di labirinti o percorsi dove è necessario tornare indietro sui passi.
- **Parsing di espressioni matematiche**: come nel valutare espressioni postfisse o infisse, o nel bilanciare parentesi.

Coda: È utilizzata in algoritmi che richiedono una sequenza di esecuzione lineare o di tipo "in prima linea".
Alcuni esempi:

- **Ricerca in ampiezza (BFS)**: un algoritmo di esplorazione di grafi che usa una coda per garantire che ogni nodo venga esplorato in ordine di distanza.
- **Sistemi di gestione delle richieste**: dove richieste e processi devono essere gestiti nell'ordine in cui arrivano, come nei server web.

ALBERI



Alberi binari

Un nodo u può avere zero o più figli v, e il loro numero viene chiamato grado del nodo

Un albero d-ario è un albero in cui tutti i nodi tranne le foglie hanno grado d

Per d = 2 diremo che l'albero è binario

Un albero d-ario in cui tutte le foglie sono sullo stesso livello è completo

```
Tipo Albero:
dati:
    un insieme di nodi di (di tipo nodo) e un insieme di archi.
operazioni:
    numNodi() -> intero
     Restituisce il numero di nodi presenti nell'albero
    grado(nodo v) -> intero
     Restituisce il numero di figli del nodo v
    padre(nodo v) -> nodo
     Restituisce il padre del nodo v nell'albero, o null se v è radice
    figli(nodo v) -> {nodo, nodo, … , nodo}
     Restituisce uno dopo l'altro i figli del nodo v
    aggiungiNodo(nodo u) -> nodo
     Inserisce un nuovo nodo v come figlio di u nell'albero e lo restituisce. Se v
     è il primo nodo ad essere inserito nell'albero esso diventa la radice (e u
     viene ignorato)
     aggiungiSottoalbero(Albero a, nodo u)
     Inserisce nell'albero il sottoalbero a in modo che la radice di a diventi
     figlia di u
     rimuoviSottoalbero(nodo v) -> Albero
     Stacca e restituisce l'intero sottoalbero radicato in v. L'operazione cancella
     dall'albero il nodo v e tutti i suoi discendenti
```

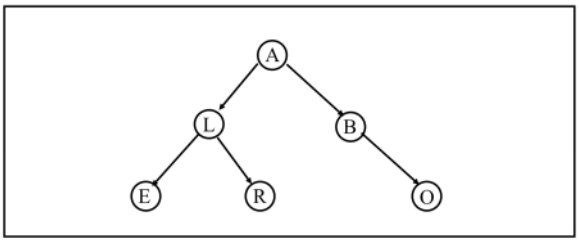
Rappresentazioni indicizzate di alberi

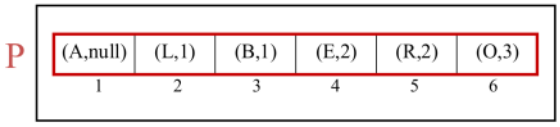
Sia T = (N, A) un albero con n nodi numerati da 0 a (n-1)

Un vettore padri è un array P di dimensione n le cui celle contengono coppie (info, parent)

Per ogni indice v appartenente [0, n-1]:

- P[v].info è il contenuto informativo del nodo v
- P[v].parent = u se e solo se vi è un arco (u, v) appartenente a A, altrimenti se v è radice P[v].parent = null





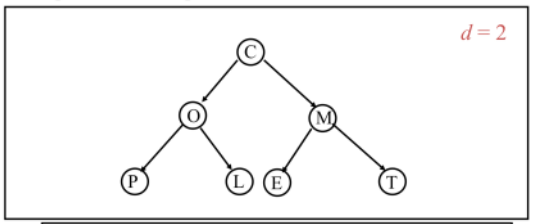
P[v].*info*: contenuto informativo nodo
P[v].*parent*: indice del nodo padre

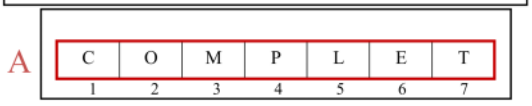
Sia T = (N, A) un albero d-ario (d >= 2) con n nodi numerati da 1 a n

Un *vettore posizionale* è un array P di dimensione n tale che

- P[v] contiene l'informazione associata al nodo v
- L'informazione associata all'i-esimo figlio di v è in posizione P[d*v+i] con (0 <= i <= d-1)

Per alberi *d*-ari completi
– *P[v] informazione nodo v*
- *P[d•v+i] informazione i-esimo figlio di v*
– *Il padre di i è in posizione* $\lfloor v/d \rfloor$





Vettore padri Vs Vettori posizionali

Usando un vettore padri, da ogni nodo è possibile risalire in tempo O(1) al proprio padre
Trovare un figlio richiede una scansione dell'array in O(n)

Usando un vettore posizionale, da ogni nodo v è possibile risalire in tempo O(1) sia al proprio padre che a uno qualunque dei figli

- Padre(v) richiede tempo costante
- Figli(v) richiede tempo proporzionale al grado di v

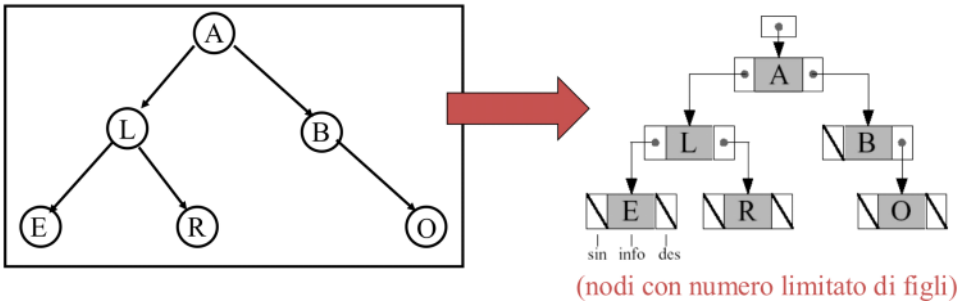
Rappresentazioni collegate di alberi

Puntatori ai figli
Se un albero ha grado al più d, si mantiene in ogni nodo un puntatore a ciascuno dei d figli
Lo spazio richiesto sarà O(n*d), che per d costante è O(n)

Puntatori ai figli

Se un albero ha grado al più *d*, si mantiene in ogni nodo un puntatore a ciascuno dei *d* figli

Lo spazio richiesto sarà $O(n \cdot d)$, che per *d* costante è $O(n)$

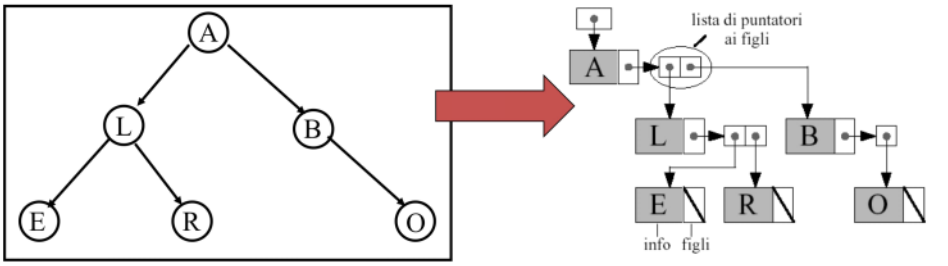


(nodi con numero limitato di figli)

Liste di puntatori ai figli

Ad ogni nodo è associata una lista di puntatori ai figli

Lo spazio richiesto sarà $O(n)$ indipendentemente dal numero di figli di un nodo

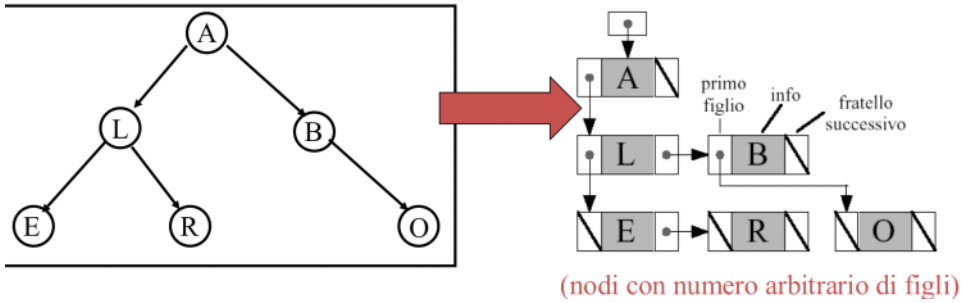


(nodi con numero arbitrario di figli)

Primo figlio-fratello successivo

Per ogni nodo manteniamo un puntatore al primo figlio ed uno al fratello successivo

Lo spazio richiesto sarà $O(n)$



(nodi con numero arbitrario di figli)

"Immaginiamo l'albero come una struttura con un nodo principale (la radice) e vari "figli" sotto di esso. Ogni nodo ha collegamenti (o "puntatori") ai propri figli, che possono essere al massimo d. Se un albero ha n nodi e ciascun nodo ha al massimo d figli, allora ogni nodo ha bisogno di d puntatori per connettersi ai suoi figli.
In termini di spazio, questo si traduce in n × d puntatori in totale. Tuttavia, se d è un numero fisso, possiamo dire che lo spazio richiesto cresce essenzialmente solo con n. Così, il requisito di spazio complessivo si esprime come O(n), il che significa che la memoria necessaria dipende principalmente dal numero di nodi, non dal numero massimo di figli."

- *ChatGPT*

Visite di alberi

Algoritmi che consentono l'accesso sistematico ai nodi e agli archi di un albero

Gli algoritmi di visita si distinguono in base al particolare ordine di accesso ai nodi

```
visitaGenerica visita il nodo r e tutti i suoi discendenti in un albero
     Algoritmo visitaGenerica(nodo r)
          S <- {r}
          While (S != 0) do
               Estrai un nodo u da S
               Visita il nodo u
               S <- S U {figli di u}
```

Richiede tempo O(n) per visitare un albero con n nodi a partire dalla radice

La "visita di alberi" è un concetto usato per descrivere il processo di esplorazione di un albero, cioè un modo per "passare in rassegna" tutti i nodi dell'albero, uno dopo l'altro, seguendo un ordine specifico. Questi algoritmi permettono di visitare i nodi e di accedere alle informazioni in essi contenute, assicurandosi che ogni nodo sia "visitato" in modo sistematico.

Algoritmi di visita di un albero

Gli algoritmi di visita permettono di attraversare tutti i nodi di un albero e si differenziano per il tipo di ordine con cui visitano questi nodi. Uno dei modi più comuni per fare una visita generica di un albero è l'algoritmo "visitaGenerica".

Spiegazione dell'Algoritmo "visitaGenerica"

Vediamo come funziona l'algoritmo che hai descritto:

1. *Inizializzazione: si parte dal nodo radice (r) e lo si aggiunge a un insieme S. Questo insieme S tiene traccia dei nodi che devono ancora essere visitati.*
2. *Ciclo di visita:*
    - *L'algoritmo entra in un ciclo While (finché S non è vuoto).*
    - *Ad ogni ciclo, estrae un nodo u da S.*
    - *Visita il nodo u: questo significa che viene eseguita un'operazione su u (come stamparlo o elaborare i suoi dati).*
    - *Poi, aggiunge tutti i figli di u nell'insieme S, cioè quelli direttamente collegati a u nel livello inferiore dell'albero.*
3. *Fine della visita: il ciclo si interrompe quando S è vuoto, quindi quando non ci sono più nodi da visitare. A questo punto, tutti i nodi dell'albero sono stati visitati.*

Esempio di utilizzo

Immagina un albero come un'organizzazione gerarchica:

- *Si parte dal direttore (nodo radice) e si visitano tutti i suoi collaboratori.*
- *Poi, si passa ai collaboratori dei collaboratori, e così via fino a coprire tutta la struttura.*

Utilità dell'algoritmo

La visita generica permette di esplorare interamente la struttura dell'albero, nodo per nodo, utile in situazioni come:

- *Ricerca di un particolare elemento o nodo nell'albero.*
- *Elaborazione di informazioni in ogni nodo, come calcoli o raccolta di dati.*
- *Verifica della struttura dell'albero per controllare se soddisfa determinate proprietà.*

In questo tipo di algoritmo, il modo in cui viene estratto u da S (ad esempio come primo o ultimo elemento) determina l'ordine esatto della visita e può cambiare il comportamento dell'algoritmo stesso, adattandolo a diverse necessità.

-ChatGPT

```
Algoritmo visitaGenerica(nodo r)
     S <- {r}
     While (S != 0) do
          Estrai un nodo u da S
          Visita il nodo u
          S <- S U {figli di u}
```

Algoritmo di visita in profondità

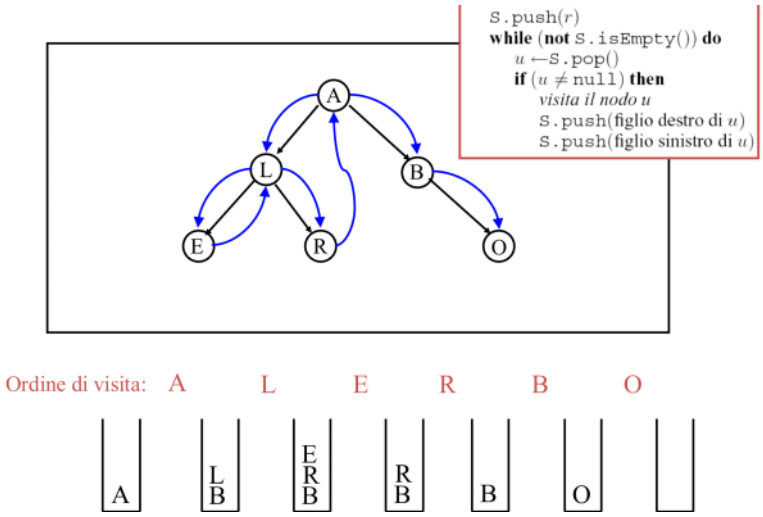
A partire dall'algoritmo di visita generica, si rappresenta l'insieme dei nodi aperti S mediante il tipo di dato Pila

L'algoritmo di visita in profondità (DFS Depth-First Search) parte da r e procedere visitando nodi di figlio in figlio fino a raggiungere una foglia. Retrocedere poi al primo antenato che ha ancora figli non visitati (se esiste) e ripete il procedimento a partire da uno di quei figli

Versione iterativa (per alberi binari)

```
     Algoritmo visitaDFS(nodo r)
```

```
Pila S
S.push( r )
While(not S.isEmpty()) do
     u <- S.pop()
     If(u != null) then
          Visita il nodo u
          S.push(figlio destro di u)
          S.push(figlio sinistro di u)
```



Per **l'algoritmo di visita in ampiezza** si usa il dato Coda

L'algoritmo di visita in ampiezza (Breadth-First Search BFS) parte da r e procede visitando nodi per livelli successivi. Un nodo sul livello i può essere visitato solo se tutti i nodi sul livello i-1 sono stati visitati

Versione iterativa (per alberi binari)

```
AlgoritmoBFS(nodo r)
     Coda C
     C.enqueue( r )
     While(not C.isEmpty()) do //"continua a eseguire il ciclo finché ci sono
     nodi da visitare"
          u <- c.dequeue()
          If(u != null) then
               Visita il nodo u
               C.enqueue(figlio sinistro di u)
               C.enqueue(figlioo destro di u)
```

L'algoritmo aggiunge alla fine, senza alterarne i precedenti ordini, r. Finché ci sono nodi da visitare: estraiamo il nodo u (e di conseguenza i suoi figli), verifichiamo u sia valido e se ha dei figli li riaggiungiamo alla coda per poter effettuare il controllo in maniera sistematica, livello per livello.

Versione ricorsiva in profondità

Visita in preordine: radice, sottoalbero sin, sottoalbero destro

```
Algoritmo visitaDFSRicorsiva(nodo r)
     If(r = null) then return
     Else visita il nodo r
     visitaDFSRicorsiva(figlio sinistro di r)
     visitaDFSRicorsiva(figlio destro di r)
```

Visita simmetrica: sottoalbero sin, radice, sottoalbero destro
(scambia riga 2 con 3)

```
Algoritmo visitaDFSRicorsiva(nodo r)
     If(r = null) then return
     Else visitaDFSRicorsiva(figlio sinistro di r)
     visita il nodo r
     visitaDFSRicorsiva(figlio destro di r)
```

Visita in postordine: sottoalbero sin, sottoalbero destro, radice
(sposta riga 2 dopo 4)

```
Algoritmo visitaDFSRicorsiva(nodo r)
     If(r = null) then return
     Else visitaDFSRicorsiva(figlio sinistro di r)
     visitaDFSRicorsiva(figlio destro di r)
     visita il nodo r
```

ALGORITMI DI ORDINAMENTO

Es: motori di ricerca

Dato un insieme S di n oggetti che appartiene a un dominio ordinato può essere ordinato.

Cosa è un dominio ordinato?

È un insieme di elementi per il quale è possibile ordinare delle relazioni reciproche.
Es: lettere, colori ed intensità riferiti in valori numerici…

L'ordinamento è una subroutine di molti problemi

Come misurare i tempi?

- Modello basato su confronti: supporremo che l'algoritmo ordina operando solo confronti tra oggetti, senza far uso di primitive (operazioni aritmetiche, logiche, ecc…). Il modello dei confronti è sufficientemente generale per catturare le proprietà degli algoritmi più noti.
- Unità di misura: numero di confronti

Algoritmi e tempi tipici

- Numerosissimi algoritmi
- Due tempi tipici: $O(n^2)$, O(nlogn) <- *sono le sue complessità tipiche*

Lower bound

Il suo lower bound è tipicamente noto, proprio perché si tratta di algoritmi molto usati:

Omega(nlogn)

Posso trovare un O peggiore ma Omega, ovvero il limite inferiore, è sempre Omega(nlogn)
Se l'algoritmo ha complessità (nlogn) allora è ottimo, se oscilla tra (nlogn) e n^2 allora è buono/efficiente ma non il migliore, per n^2 è il caso peggiore invece.

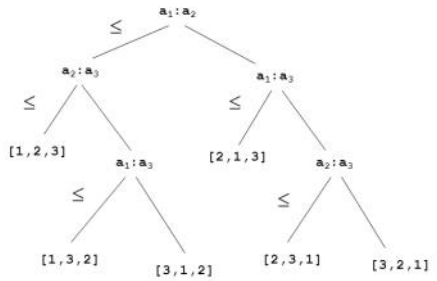
L'algoritmo di ordinamento opera facendo dei confronti, nello specifico coppie di elementi:

$a_i < a_j$; $a_i <= a_j$; $a_i = a_j$; ecc…

Alberi di decisione

Si chiama di decisione perché gli archi seguono una sequenza specifica

Albero di decisione per Ordinamento



Nota: 3 elementi, 3! combinazioni, 6 foglie

Ciascun nodo modella il confronto fra due elementi

Algoritmi di ordinamento con tempo quadratico:

SelectionSort, InsertionSort, BubbleSort sono tre algoritmi che eseguono $O(n^2)$ confronti nel caso peggiore, non sono algoritmi ottimi in base al lower bound

Sono algoritmi che operano in loco, ovvero hanno solo bisogno della memoria necessaria per mantenere gli elementi da ordinare (array) e uno spazio addizionale costante.

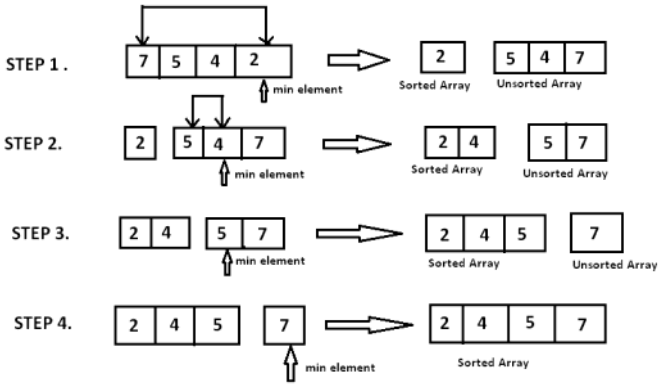
SelectionSort e InsertionSort sono algoritmi di ordinamento incrementale.

Supponiamo che i primi k elementi dell'array siano già ordinati per 0 < k < n, estremi inclusi, vogliamo estendere l'ordinamento a k+1 elementi.

- **SelectionSort** sceglie il minimo degli n-k elementi non ancora ordinati e lo mette in posizione k+1

- L'algoritmo di **InsertionSort** prende l'elemento successivo, quello in posizione k+1, e lo inserisce nel punto corretto rispetto alla porzione già ordinata a sinistra. In altre parole, scorre all'indietro finché non trova il posto giusto per posizionare il nuovo elemento, assicurandosi che tutto rimanga ordinato fino a quel punto.

SELECTION SORT



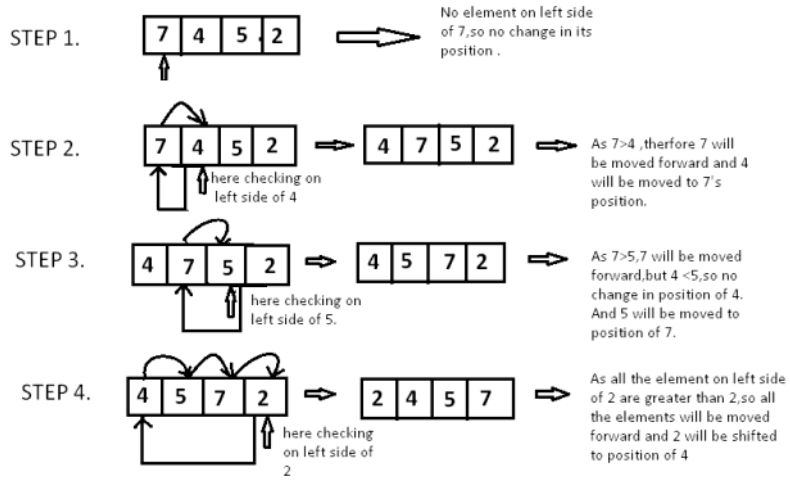
Pseudocodice

```
Algoritmo SelectionSort
for k = 0 to n-2 do //se k fosse partito da 1 avrebbe avuto senso n-1 perché a me
non importa solo     l'ultimo elemento
      m = k+1
      for j = k+2 to n do
            if(A[j] < A[m]) then m = j
```

```
scambia A[m] con A[k+1]
```

Il costo sarà $O(n^2)$

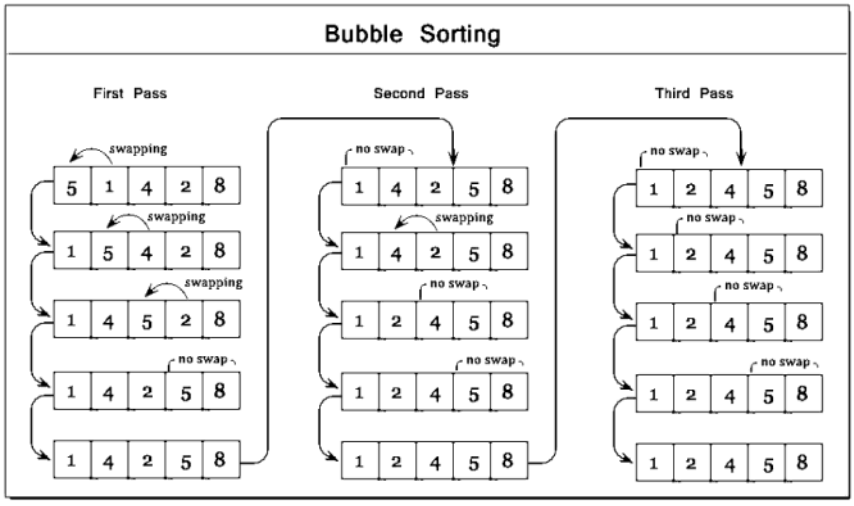
INSERTION SORT



Pseudocodice

```
Algoritmo insertionSort(array A)
    for k = 1 to n-1 do
      X <- A[k+1]
      For j = k downto 0 do
          if(j > 0 and A[j] <= x) then breal
      If (j < k) then
          for t = k downto j+1 do A[t+1] <- A[t]
          A[j+1] <- x
```

BubbleSort



Pseudocodice

```
BubbleSort(A)

for i = 1 to n-1 do
      scambi = false //scambi è una variabile booleana
      for j = 2 to n-i+1 do
            if(A[j-1] > A[j]) then scambia A[j-1] e A[j]; scambi = true
      if (scambi = false) then break
```

Il bubble sort iterativamente spinge l'ultimo elemento alla fine

Da un punto di vista spaziale tutti e tre operano in loco, anche in termini di notazione asintotica

non cambia, sono tutti $O(n^2)$

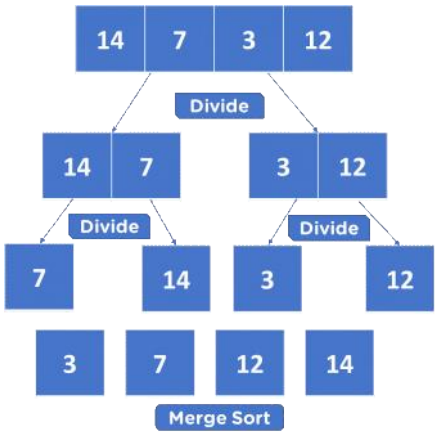
Ordinamenti ottimi

MergeSort

- Usa la tecnica del divide et impera:

1. Divide: dividi l'array a metà
2. Risolvi i due sottoproblemi ricorsivamente
3. Impera: fondi le due sottosequenze ordinate

```
Merge(A, i, f)

if(i >= f) then return
m = [(i+f)/2]
MergeSort(A, i, m) // in ordine richiama l'array A da i fino a m. PRIMA META
(divide)
MergeSort(A, m+1, f) // in ordine richiama l'array A da m+1 fino a f. SECONDA META
(divide)
Merge(A, i, m, f) // fonde le due sottosequenze richiamando l'array A per intero e riordinandolo (merge)
```



Nel merge le sequenze vengono ripetute al contrario e le riordina partendo dalle piccole sotto sequenze ottenute del divide.

Impera: procedura Merge

Due array ordinati A e B possono essere fusi rapidamente:

- Estrai ripetutamente il minimo di A e B e copialo nell'array di output, finché A oppure B non diventa vuoto
- Copia gli elementi dell'array non vuoto alla fine dell'array di output

Pseudocodice del Merge

```
Merge(A, i, m, f)
sia X un array ausiliario di lunghezza m-i+1
i = 1
i2 = f +1
while (i <= f e i2 <= m) do
   if(A[i] <= A[i2]
     then X[i] = A[i]
   else X[i] = A[i2]
if(i <= f) then copia A[i;f] alla fine di X
else copia A[i2;m] alla fine di X
copia X in A[i;m]
```

Visualizzazione pratica:

Algoritmo fornito:

1. Si assume che A sia diviso in due sotto-array ordinati, uno che va da i a f e l'altro che va da i2 = f + 1 a m.
2. Si utilizza un array ausiliario X per aiutare nel processo di fusione.
3. I passi seguenti si basano sul confronto e sulla copia degli elementi dei due sotto-array in X, quindi il contenuto di X viene copiato di nuovo in A alla fine.

Esempio pratico

Supponiamo di avere l'array A = [3, 27, 38, 43, 9, 10, 82], dove:

- La prima metà ordinata va da i = 0 a f = 3, ovvero [3, 27, 38, 43]
- La seconda metà ordinata va da i2 = f + 1 = 4 a m = 6, ovvero [9, 10, 82]

Vogliamo fondere questi due sotto-array ordinati usando Merge(A, i, m, f).

Passaggi con lo Pseudocodice

Inizializza l'array ausiliario X di lunghezza m - i + 1 = 7.

Inizializza gli indici: i1 = i (inizio del primo sotto-array) e i2 = f + 1 (inizio del secondo sotto-array).

Esegui il ciclo while per riempire X confrontando elementi da entrambi i sotto-array:

X inizia vuoto [].

Iterazione 1:
Confronta A[i1] = 3 con A[i2] = 9
3 <= 9, quindi X diventa [3], e incrementiamo i1.

Iterazione 2:
Confronta A[i1] = 27 con A[i2] = 9
9 < 27, quindi X diventa [3, 9], e incrementiamo i2.

Iterazione 3:
Confronta A[i1] = 27 con A[i2] = 10
10 < 27, quindi X diventa [3, 9, 10], e incrementiamo i2.

Iterazione 4:
Confronta A[i1] = 27 con A[i2] = 82
27 <= 82, quindi X diventa [3, 9, 10, 27], e incrementiamo i1.

Iterazione 5:
Confronta A[i1] = 38 con A[i2] = 82
38 <= 82, quindi X diventa [3, 9, 10, 27, 38], e incrementiamo i1.

Iterazione 6:
Confronta A[i1] = 43 con A[i2] = 82
43 <= 82, quindi X diventa [3, 9, 10, 27, 38, 43], e incrementiamo i1.

Copia gli elementi rimanenti:

Il primo sotto-array è stato completamente elaborato (i1 ha superato f).
Copia il restante elemento del secondo sotto-array (82) alla fine di X.
Ora X è [3, 9, 10, 27, 38, 43, 82].

Copia X in A[i : m]:

Ora aggiorniamo A con i valori di X nell'intervallo originale.
A diventa [3, 9, 10, 27, 38, 43, 82], che è ordinato.

Risultato Finale
L'array A dopo il merge sarà [3, 9, 10, 27, 38, 43, 82].

Il ciclo while terminerà quando uno dei due sotto-array si svuota. In questo caso, verranno effettuate **7 iterazioni** del ciclo while (4 dal primo sotto-array e 3 dal secondo).
Dopo queste 7 iterazioni, uno dei sotto-array sarà vuoto, e verrà copiato il rimanente dall'altro (se presente) in X.

LEZIONE 14/11

Il ***QuickSort*** è un altro algoritmo di ordinamento che usa la tecnica del divide et impera.

L'idea è di partizionare la sequenza in modo tale che gli elementi di una sottosequenza siano tutti più piccoli degli elementi dell'altra

Dunque il **Divide** è complesso ma l'**Impera** è banalmente una concatenazione.

- **Divide**: scegli un elemento x della sequenza (perno) e partiziona la sequenza in elementi $\leq$ x e > x
- Risolvi i due sottoproblemi ricorsivamente
- **Impera**: restituisci la concatenazione delle due sottosequenze ordinate

QuickSort(A)
1. Scegli elemento x in A
2. Partiziona A rispetto a x calcolando
   a. A1 = {y appartenente A : y $\leq$ x} #divide
   b. A2 = {y appartenente A : y > x} #divide
3. if(|A1| > 1) **then** QuickSort(A1)
4. if(|A2| > 1) **then** QuickSort(A2)
5. Copia la concatenazione di A1 e A2 in A #impera

Creando le due sottosequenze A1 e A2 sto di fatto utilizzando memoria ausiliaria e non sto ordinando in loco poiché non opero modificando direttamente l'Array A ma creando sue copie

Partizione in loco

Scorri l'array "in parallelo" da sinistra verso destra e da destra verso sinistra

- Da sinistra verso destra ci si ferma su un elemento maggiore del nostro perno x (indice sup)
- Da destra verso sinistra ci si ferma su un elemento minore del nostro perno x (indice inf)

Scambia gli elementi e riprendi la scansione

```
QuickSort(A, i, f)
if(i >= f) then return
m = Partition(A, i, f)
QuickSort(A, i, m-1)
QuickSort(A, m+1, f)

Partition(A, i, f)
x = A[i]
inf = i
sup = f+1
while (true) do
     do(inf = inf+1) while (inf <= f e A[inf] <= x)
     do(sup = sup-1) while (A[sup]> x)
     if(inf < sup) then scambia A[inf] e A[sup]
     else break
scambia A[i] e A[sup]
return sup
```

Il QuickSort si occupa di riordinare le sottosequenze individuato il perno che inizializziamo come m = Partition(A, i, f). Partition è la funzione che si occupa di trovare il nostro perno dati l'array A e

gli elementi i e f dell'array

Esempio pratico

```
A = [8, 3, 6, 1, 7, 5, 2]
Partition(A, 0, 6)          #considerati come A[0] e A[6]
```

Considerando quindi i = 0 il mio perno, la mia x, è 8.
Sup = 6+1 quindi 7 che è la lunghezza dell'array
Entriamo ora nel ciclo While quindi:

Incrementiamo inf finché A[inf] <= perno:

1. inf = 1 (A[1] = 3 <= 8), quindi inf si incrementa.
2. inf = 2 (A[2] = 6 <= 8), quindi inf si incrementa.
3. inf = 3 (A[3] = 1 <= 8), quindi inf si incrementa.
4. inf = 4 (A[4] = 7 <= 8), quindi inf si incrementa.
5. inf = 5 (A[5] = 5 <= 8), quindi inf si incrementa.
6. inf = 6 (A[6] = 2 <= 8), quindi inf si incrementa.
7. inf = 7 (fuori dall'intervallo, si ferma qui).

Stato attuale:
inf = 7, sup = 7
A = [8, 3, 6, 1, 7, 5, 2]
Decrementiamo sup finché A[sup] > pivot:

sup = 6 (A[6] = 2 <= 8), quindi sup si ferma qui.

Stato attuale:
inf = 7, sup = 6
A = [8, 3, 6, 1, 7, 5, 2]
Si interrompe anche il while true perché inf >= sup (7 >= 6)

Si scambiano A[0] (ovvero 8) e A[6] (ovvero 2)

[2, 3, 6, 1, 7, 5, 8]

Il perno 8 è stato posizionato correttamente all'indice 6.

Tutti gli elementi **a sinistra** del perno sono **minori o uguali a 8**.
Tutti gli elementi **a destra** del perno (in questo caso nessuno, perché è l'ultimo elemento) sono **maggiori di 8**.

Quando l'indice inf supera l'indice sup si ha la partizione desiderata

L'albero delle chiamate ricorsive può essere sbilanciato, cosa comporta in termini di calcolo del costo?

Calcolare la partizione intorno al perno richiede al peggio n-1 confronti

Nel caso peggiore: $T(n) = Theta(n^2)$
Nel caso migliore: $T(n) = Theta(n\log_2 n)$

Per rendere il QuickSort più efficiente possiamo parlare di randomizzazione

- Un algoritmo si dice randomizzato se usa numeri casuali
- Supponiamo di scegliere il perno x come il k-esimo elemento di A, scegliendo k in modo casuale
- Per calcolare il numero atteso di confronti dobbiamo calcolare la somma dei tempi di

esecuzione pensandoli in base alla probabilità

Il QuickSort randomizzato non è uguale al caso medio

- Complessità temporale non dipende dall'ordine dell'input
- Nessuna assunzione sulla distribuzione di probabilità delle istanze
- Nessun input specifico per il quale si verifica il caso peggiore
- Il caso peggiore determinato solo dal generatore di numeri casuali

Teorema
L'algoritmo quickSort randomizzato ordina in loco un array
di lunghezza *n* in tempo O(*n*^2) nel caso peggiore e
O(*n* log *n*) tempo atteso

LEZIONE 15/11

Alberi di ricerca

Sono degli alberi e mantengono le proprietà degli alberi ma servono per modellare problemi di ricerca.
Disponiamo di un insieme S di elementi cui sono associate chiavi prese da un dominio totalmente ordinato.

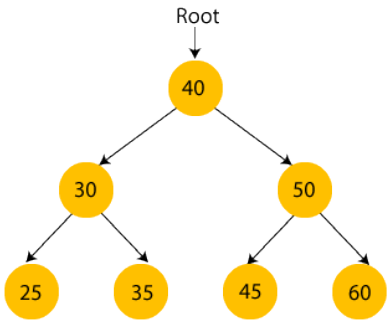
L'insieme S è dinamico è dunque possibile:
- Inserire elementi (insert)
- Cancellare elementi (delete)

Vogliamo poter cercare elementi in S (search).

BST - binary search tree

Ricerca binaria: confronto l'elemento da cercare all'interno dell'array ordinato X con X[n/2] ed effettuo chiamate ricorsive su n/4, n/8, etc

Associamo all'array ordinato X un albero T partendo dall'elemento centrale come radice



Ogni nodo ha a sinistra un nodo più piccolo e a destra un nodo più grande, la stessa relazione ricorsiva si ripete anche per il padre. Quindi tutti i nodi più piccoli si troveranno nel sottoalbero a sinistra, al contrario i più grandi si troveranno nel sottoalbero di destra.
Corrispondo tutti a un cammino dalla radice al nodo 40 > 30 > 25; 40 < 50 < 60. La ricerca prevede nel caso peggiore di dover percorrere l'albero in un cammino che va dalla radice alla foglia *(ovvero* $log_2 n$*)*.

Si tratta dunque di un albero binario che gode di alcune **proprietà**:

1. Ogni nodo v contiene un elemento elem(v) cui è associata una chiave key(v) presa da un dominio totalmente ordinato.
2. Le chiavi nel sottoalbero sinistro di v sono <= key(v)
3. Le chiavi nel sottoalbero destro di v sono >= key(v)

Albero binario di ricerca:

- Alla sinistra del nodo devo trovare un nodo più piccolo
- Alla destra del nodo devo trovare un nodo più piccolo
- Tutti i nodi a sinistra devono essere minori della radice
- Tutti i nodi a destra devono essere maggiori della radice

Se queste condizioni sono soddisfatte allora l'albero è di ricerca

NB: si tratta di una visita SIMMETRICA (sottoalbero sinistro, radice, sottoalbero destro)

Se avessi trovato un nodo minore della radice nel sottoalbero di destra allora non ho un albero di ricerca

```
classe AlberoBinarioDiRicerca implementa Dizionario:
dati:                                                    S(n) = O(n)
Un albero binario di ricerca T di altezza h e con n nodi ciascuno contenente coppie
(elem e, key k) in cui le chiavi sono prese da un universo totalmente ordinato
operazioni:

Search(key k) -> elem    costo proporzionale all'altezza dell'albero T(h) = O(h)

Partendo dalla radice, traccia un cammino nell'albero per cercare un elemento con
chiave k. Su ogni nodo, usa la proprietà di ricerca per decidere se proseguire nel
sottoalbero sinistro o destro.

Insert(elem e, key k)                                    T(h) = O(h)

Crea un nuovo nodo v contenente la coppia (e, k) e lo aggiunge all'albero come
foglia nella posizione opportuna, in modo da mantenere la proprietà di ricerca.

delete(elem e)                                  T(h) = O(h)

Se il nodo v contenente l'elemento e ha al più un figlio, elimina v collegando il
figlio all'eventuale padre. Altrimenti scambia il nodo v con il suo predecessore ed
elimina il predecessore
```

Pseudocodice

```
Algoritmo search(key k) -> elem
v <- radice di T
while (v != null) do
    if (k = key(v)) then return elem(v)
    else if (k < key(v)) then return v <- figlio sinistro di v
    else v <- figlio destro di v
return null
```

L'algoritmo effettua una ricerca finché v è diverso da null
Quando v è uguale a null allora restituisce null e finisce la ricerca, non lo trova

Il costo è T(h) = O(h) proporzionale all'altezza dell'albero

Nella ricerca binaria, per forza, il costo è O(logn) perché l'albero è bilanciato (ovvero l'altezza del sottoalbero di destra è uguale all'altezza del sottoalbero di sinistra)

In questo caso, invece, l'albero viene costruito secondo l'ordinamento delle proprietà di ricerca, dunque non è bilanciato dunque il costo è O(h)

Un nuovo nodo u con elem = e, chiave = k viene sempre inserito come foglia. Il costo dell'inserimento è O(h)

L'operazione **insert** prevede due passi:

1. Cerca il nodo v che diventerà genitore di u
2. Appendi u come figlio

```
Algoritmo max(nodo v) -> nodo
v <- u
while(figlio destro di v != null) do
     v <- figlio destro di v
return v
```

Ricerca del predecessore

È la chiave più grande che risulta più piccola del nodo che sto cercando, la prima più piccola

Per trovare il predecessore distinguiamo due casi:

- u ha un figlio sinistro: pred(u) è il massimo del sottoalbero sinistro di u, ovvero il più a destra del sottoalbero sinistro
- u non ha un figlio sinistro: pred(u), se esiste, è l'antenato di u con massima profondità nell'albero, il cui figlio destro è anch'esso antenato di u. Per trovarlo risaliamo l'albero da u verso la radice fino ad incontrare la prima "svolta a sinistra" (sostanzialmente leggo l'albero al contrario)

```
Algoritmo pred(nodo u) -> nodo
if (u ha un figlio sinistro sin(u)) then
     return max(sin(u))
while (parent(u) != null e u è figlio sinistro di suo padre) do
     u <- parent(u)
return parent(u)
```

Sia u il nodo contenente l'elemento e da cancellare; ci sono tre possibilità:

1. u è una foglia, basta rimuoverla
2. u ha un solo figlio w:
    - Se u è radice, w diventa la nuova radice
    - Altrimenti individuiamo il genitore v di u e sostituiamo l'arco (u, w) con (v, w)
3. u ha due figli: ci si riconduce ad uno dei casi precedenti:
    - Si individua il predecessore v di u (v è il max del sottoalbero sx di u)
    - Si copia chiave(v) in chiave(u)
    - Si cancella v dall'albero secondo uno dei due casi precedenti

Recap:

Alberi binari di ricerca di altezza h sono in grado di supportare operazioni search, insert e delete in tempo O(h)

Per alberi molto sbilanciati h = O(h)

Quanto vale però h? quanto l'altezza dell'albero

Per alberi molto bilanciati però il costo è h = O(logn)

Alberi bilanciati garantiscono un tempo di ricerca
logaritmico

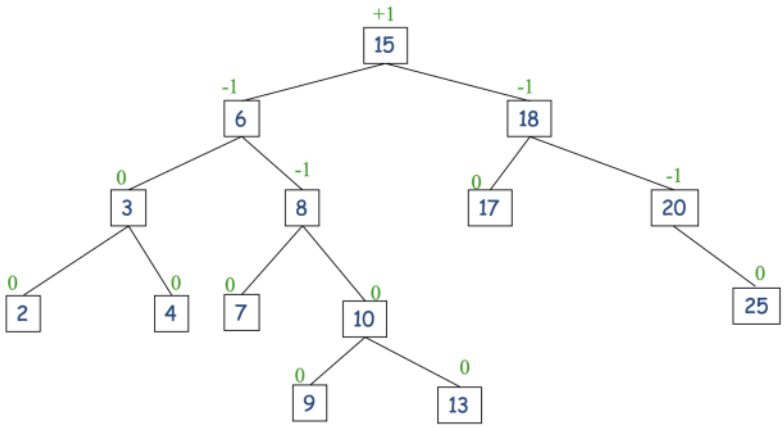
• Pur partendo da un albero bilanciato, inserimenti e cancellazioni lo sbilanciano peggiorando le prestazioni
• Obiettivo: mantenere il bilanciamento

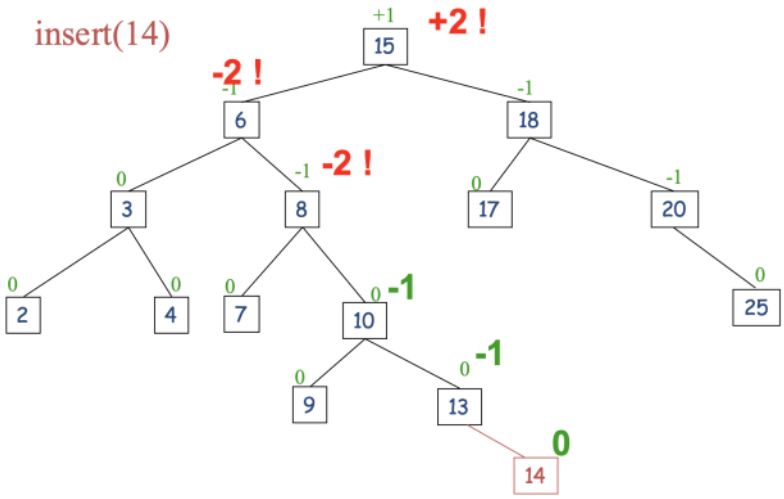
Bilanciamento in altezza: un albero è bilanciato in altezza se le altezze dei sottoalberi sinistro e destro differiscono al più di uno
Gli alberi bilanciati in altezza sono detti AVL
In un albero AVL, oltre all'elemento e alla chiave, ciascun nodo mantiene l'informazione relativa al fattore di bilanciamento
Il fattore di bilanciamento (beta) di un nodo v è la differenza tra l'altezza del sottoalbero sinistro e quella del sottoalbero destro di v

$$\beta = altezza(sin(v)) - altezza(des(v))$$

In un albero binario completo il fattore di bilanciamento è zero
Un albero binario di ricerca è un albero AVL se su ogni nodo v si ha β(v) <= 1

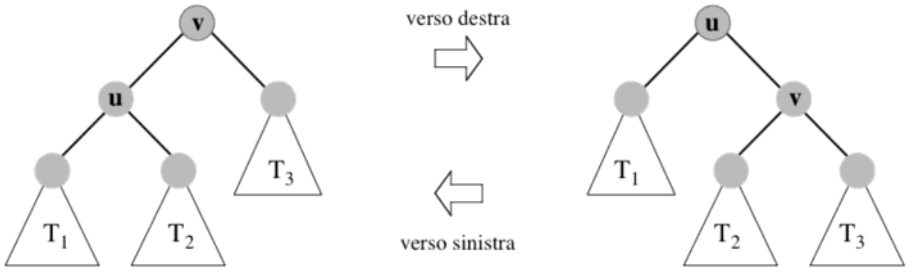
Gli inserimenti possono sbilanciare l'AVL





Anche le cancellazioni possono sbilanciare l'AVL ma esistono opportune operazioni dette rotazioni che permettono di mantenere il bilanciamento.

ROTAZIONE DI BASE



Si sposta un nodo (perno) verso destra o verso sinistra
Mantiene la proprietà di ordinamento totale: l'ordine relativo delle chiavi in T1, T2, T3 e nei nodi u,v rimane invariato
Viene effettuata sui nodi sbilanciati β >= 2
Si implementa aggiornando i puntatori dei nodi interessati

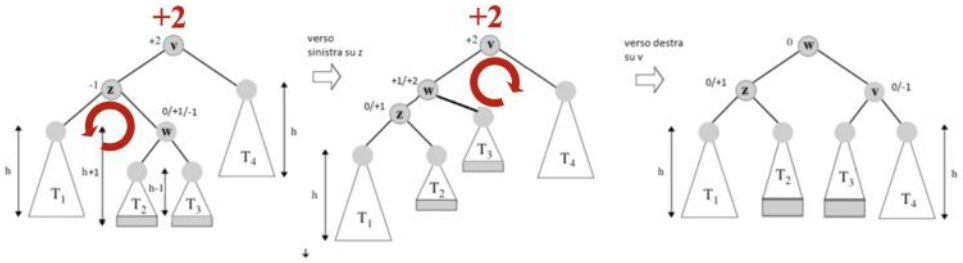
RIBILANCIAMENTO TRAMITE ROTAZIONI

Sia v un nodo con fattore di bilanciamento β = +- 2

- Il sottoalbero sinistro o destro di v sbilancia v (vale a dire che ha un'altezza eccessiva)

Sia T il sottoalbero che sbilancia v, allora a seconda della posizione di T si hanno 4 casi:

- (SS) lo sbilanciamento è dovuto al sottoalbero **S**inistro del figlio **S**inistro di v
- (DD) lo sbilanciamento è dovuto al sottoalbero **D**estro del figlio **D**estro di v
- (SD) lo sbilanciamento è dovuto al sottoalbero **S**inistro del figlio **D**estro di v
- (DS) lo sbilanciamento è dovuto al sottoalbero **D**estro del figlio **S**inistro di v



Si applicano due rotazioni semplici

- Una verso sinistra sul figlio sinistro del nodo critico (nodo z)
- L'altra verso destra sul nodo critico (nodo z)

```
classe AlberoAVL estende AlberoBinarioDiRicerca
dati:
      albero binario di ricerca T ereditato, più il fattore di bilanciamento di ogni nodo S(n) =
      O(n)
operazioni:
search(chiave k) -> elem                        T(n) = O(logn)
      ereditata
insert(elem e, chiave k)                          T(n) = O(logn)
      chiama insert() ereditata, poi ricalcola i fattori di bilanciamento ed eventualmente
      ribilancia tramite O(1) rotazioni
delete(elem e)                                          T(n) = O(logn)
      chiama delete() ereditata, poi ricalcola i fattori di bilanciamento ed eventualmente
      ribilancia tramite O(logn) rotazioni
```

Grafi e visite di grafi

Un grafo è una notazione matematica astratta usata per rappresentare l'idea di connessione o relazione tra coppie di oggetti. Un grafo G = (V, E) consiste in un:

- Un insieme di vertici V = {v1, …, vn} o nodi
- Un insieme E = {(vi, vj) | vi,vj appartenenti a V} di coppie non ordinate di vertici, detti archi.

Esempio:

a. V = {persone che vivono in Italia}, E = {coppie di persone che si sono strette la mano} relazione simmetrica => grafo non orientato
b. V = {persone che vivono in Italia}, E = {(x, y) tale che x ha inviato una email a y} relazione non simmetrica => grafo orientato (diretto)

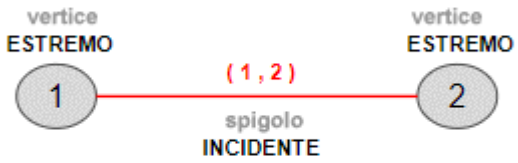
Un grafo diretto D = (V, A) consiste in:

- Un insieme di vertici V = {v1, …, vn} o nodi
- Un insieme E = {(vi, vj) | vi,vj appartenenti a V} di coppie ordinate di vertici, detti archi

Esempio: Disegnare il grafo diretto che ha come vertici i primi 6 numeri interi, e ha un arco diretto da x verso y se x≠y e x è un multiplo di y

V={1,…,6}, A={(2,1), (3,1), (4,1), (5,1), (6,1), (4,2), (6,2),(6,3)}in

Incidenza

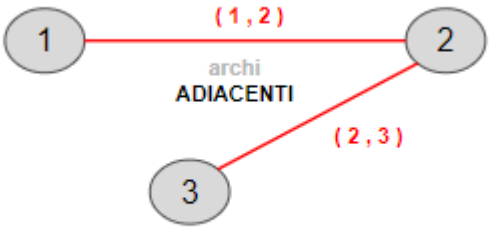


WWW.ANDREAMININI.COM

- Un vertice è incidente ad un arco se il vertice è uno degli estremi dell'arco
- Un arco è incidente ad un vertice se il vertice è uno degli estremi dell'arco

Se il grafo è orientato, l'arco (x, y) esce da x e entra in y

Adiacenza

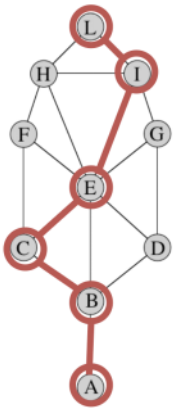


WWW.ANDREAMININI.COM

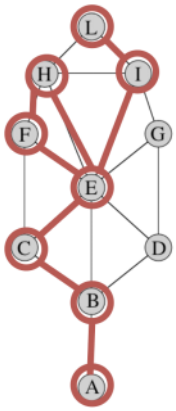
- Se G è un grafo non orientato diremo che x ed y sono adiacenti
- Se G è un grafo orientato diremo che y è adiacente ad x

Dato un vertice v, i vertici adiacenti a v sono vicini di v

Un cammino tra due vertici x,y in G è una sequenza di vertici <v0, v1, …, vk> con v0 = x e vk = y, tale che 1<=i<=k, l'arco (v(i-i), vi) appartenente a G. Se tutti i vertici sono distinti il cammino viene detto semplice



<L,I,E,C,B,A> è un cammino semplice di lunghezza 5 tra L ed A



<L,I,E,C,B,A> è un cammino semplice di lunghezza 5 tra L ed A
<A,B,C,E,F,H,E,I,L> non è un cammino semplice

- Se esiste un cammino da x a y, y è raggiungibile da x, x è antenato di y
- La lunghezza del più corto cammino tra i due vertici si dice distanza
- Un cammino tale che v0 = vk e k>= 1 è un ciclo
- Un grafo non orientato G = (V, E) si dice *connesso* se esiste un cammino tra ogni coppia di vertici in G
- Un grafo orientato G = (V, E) si dice *fortemente connesso* se esiste un cammino orientato tra ogni coppia di vertici in G

Il problema dei sette ponti

Un grafo G = (V, E) si dice percorribile (Euleriano) se e solo se contiene un cammino che passa una ed una sola volta su ciascun arco in E

Teorema di Eulero
Un Grafo G = (V, E) connesso è percorribile se e solo se ha tutti i nodi di grado pari, oppure se ha esattamente due nodi di grado dispari

Dimostrazione
• Un grafo con tutti i nodi di grado pari può essere percorso nel seguente modo: si parte da un qualsiasi nodo, e si percorrono arbitrariamente gli archi, eliminandoli una volta percorsi; così facendo si terminerà sul nodo di inziale
• Per percorrere un grafo avente due nodi di grado dispari e tutti gli altri di grado pari, è necessario partire da uno qualsiasi dei due nodi di grado dispari, e il percorso terminerà sull'altro nodo di grado dispari

Il problema dei sette ponti non ammette soluzione, in quanto i quattro nodi hanno tutti grado dispari, e quindi il grafo non è percorribile

STRUTTURE DATI PER RAPPRESENTARE GRAFI

Sono circa le medesime viste fino ad ora.

Si dovranno ovviamente rappresentare con strutture che abbiano un senso per il calcolatore.
Si consideranno quindi delle liste e più degli array delle matrici (ovvero array bidimensionale).

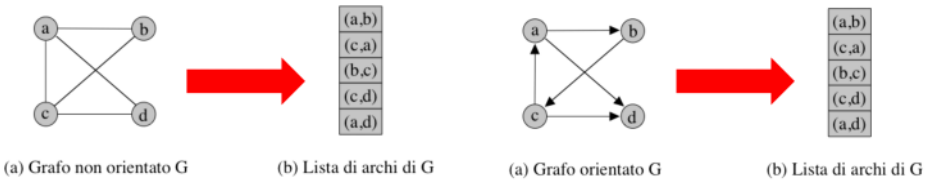
Le matrici sono array di array allocati in memoria in posizioni vicine.
Queste strutture che vedremo saranno o **strutture indicizzate** (array/matrici) o **strutture collegate** (record/puntatori).

Le operazioni sono definite in termini del loro significato

tipo Grafo:
dati:
un insieme di vertici (di tipo vertice) e un insieme di archi (di tipo arco)
operazioni:
numVertici()→ intero
restituisce il numero di vertici presenti nel grafo
numArchi() → intero
restituisce il numero di archi presenti nel grafo
archiIncidenti(vertice v) → (arco, arco, …, arco) ***O(m)***
restituisce uno dopo l'altro gli archi incidenti sul vertice v
estremi(arco e) → (vertice, vertice)
restituisce gli estremi x e y dell'arco e = (x,y)
opposto(vertice x, arco e) → vertice
restituisce y, l'estremo dell'arco e = (x,y) diverso da x
sonoAdiacenti(vertice x, vertice y) → booleano ***O(m)***
restituisce true se esiste l'arco (x,y), e false altrimenti
aggiungiVertice(vertice v) ***O(1)***
inserisce un nuovo vertice v
aggiungiArco(vertice x, vertice y) ***O(1)***
inserisce un nuovo arco tra i vertici x e y
rimuoviVertice(vertice v) ***O(m)***
cancella il vertice v e tutti gli
arco ad esso incidenti
rimuoviArco(arco e) ***O(m)***
cancella l'arco e

Il costo della ricerca del grado v è O(m)

LISTA DI ARCHI



(a) Grafo non orientato G (b) Lista di archi di G (a) Grafo orientato G (b) Lista di archi di G

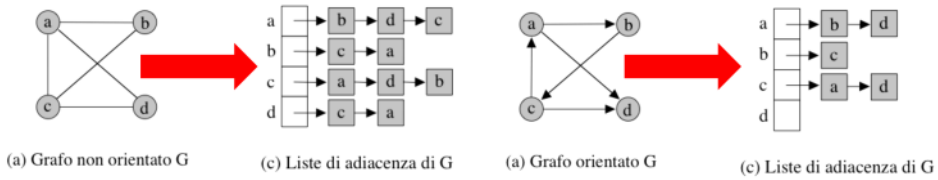
Rappresentazione tramite lista o array di strutture per memorizzare n vertici e m archi. La lista di archi è a tutti gli effetti una lista ed è quella più vicina alla definizione di grafo, ovvero *un insieme di vertici e di archi*.

Lo spazio usato è O(m+n)

LISTE DI ADIACENZA

- Ogni vertice v ha una lista contenente i suoi vertici adiacenti, ovvero tutti i vertici u per cui esiste un arco (v, u)

- Per trovare gli archi incidenti su un vertice basta scandire le n (una per ogni vertice) liste di adiacenza



Anche se si riferisce agli archi adiacenti mi fornisce informazioni sugli archi incidenti

Nel caso di grafo orientato le operazioni sono le medesime tenendo però conto della definizione di adiacenza (d non punta verso nessuno, a e c sono adiacenti a d *ma d non è adiacente ad a e c*)

I costi variano rispetto alla lista di archi

Operazione	Tempo di esecuzione
`grado`(v)	$O(\delta(v))$
`archiIncidenti`(v)	$O(\delta(v))$
`sonoAdiacenti`(x, y)	$O(\min\{\delta(x), \delta(y)\})$
`aggiungiVertice`(v)	$O(1)$
`aggiungiArco`(x, y)	$O(1)$
`rimuoviVertice`(v)	$O(m)$
`rimuoviArco`$(e = (x, y))$	$O(\delta(x) + \delta(y))$

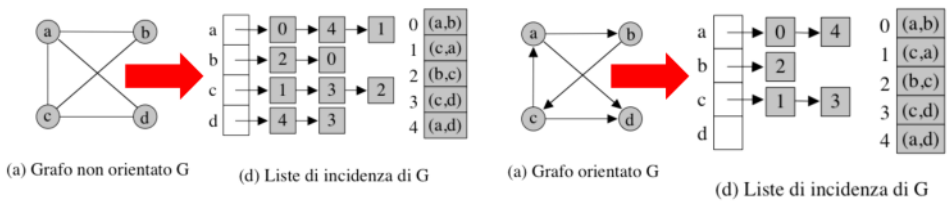
Il simbolo δ(v)δ(v) rappresenta il **grado del vertice vv** in un grafo.

- Nel caso di grafi non orientati, δ(v)δ(v) indica il numero di archi incidenti al vertice vv.
- Nel caso di grafi orientati, δ(v)δ(v) potrebbe riferirsi al grado entrante (deg−(v)deg−(v)) o uscente (deg+(v)deg+(v)) del vertice vv, ma spesso ci si riferisce al grado totale.

Quindi, nei contesti della tabella, δ(x)δ(x) e δ(y)δ(y) indicano rispettivamente il grado dei vertici xx e yy.

L'adiacenza è ridondante in questa rappresentazione (caso di grafi non orientati), l'informazione che a sia adiacente a b la trovo sia guardando la lista di a che la lista di b. Tra i due elementi è sempre utile trovare il minimo nella lunghezza della lista, se a ha 1 milione di elementi e b solo 5 se sono adiacenti mi basta verificare che a sia presente nella lista di b.

LISTE DI INCIDENZA



- Combina i vantaggi della lista di archi e della lista di adiacenza
- Memorizziamo per ogni vertice v una lista di puntatori degli archi incidenti su v

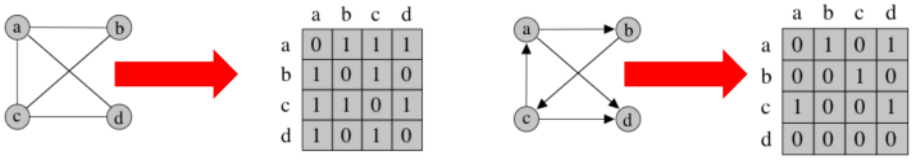
Se un arco (a, b) è un arco che collega i due nodi allora è incidente, nella lista di adiacenza avrei duplicato l'informazione, in questo caso invece lo scrivo una volta sola

Operazione	Tempo di esecuzione
`grado`(v)	$O(\delta(v))$
`archiIncidenti`(v)	$O(\delta(v))$
`sonoAdiacenti`(x, y)	$O(\min\{\delta(x), \delta(y)\})$
`aggiungiVertice`(v)	$O(1)$
`aggiungiArco`(x, y)	$O(1)$
`rimuoviVertice`(v)	$O(m)$
`rimuoviArco`$(e = (x, y))$	$O(\delta(x) + \delta(y))$

MATRICI DI ADIACENZA

- La matrice di adiacenza è una matrice di n righe x n colonne
- Lo spazio usato è O(n^2), tuttavia la verifica degli archi è più efficiente. Nasce per facilitare tutti quei tipi di operazioni finalizzate a verificare l'adiacenza fra due nodi

- M(u, v)= 1 se (u, v) è un arco di G
  (è una matrice binaria)

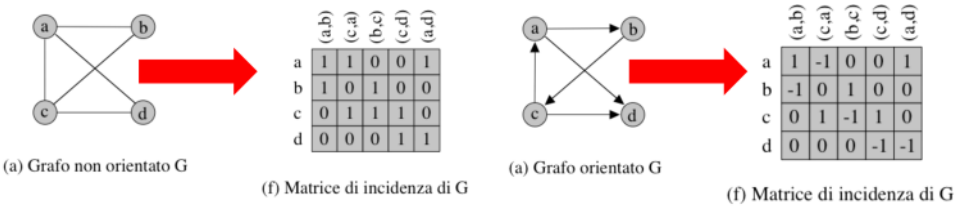


- Per i grafi non orientati M[x, y] = M[y, x], per grafi orientati no
- La verifica di un arco avviene in tempo costante accedendo all'elemento della matrice

Operazione	Tempo di esecuzione
`grado`(v)	$O(n)$
`archiIncidenti`(v)	$O(n)$
`sonoAdiacenti`(x, y)	$O(1)$
`aggiungiVertice`(v)	$O(n^2)$
`aggiungiArco`(x, y)	$O(1)$
`rimuoviVertice`(v)	$O(n^2)$
`rimuoviArco`(e)	$O(1)$

Efficiente per la verifica dell'adiacenza e la verifica degli archi, non troppo per le altre operazioni. Non sono quindi efficienti per sistemi molto dinamici

MATRICI DI INCIDENZA



(a) Grafo non orientato G (f) Matrice di incidenza di G (a) Grafo orientato G (f) Matrice di incidenza di G

- La cella avrà valore 1 se vertice ed arco corrispondente sono incidenti
- Lo spazio usato è O(mn), indicizza vertici (righe) ed archi (colonne)
- Ogni colonna (x, y) avrà due valori pari ad 1 in corrispondenza di x, y
- Per grafi orienta
- ti avremo valori +1, -1 per distinguere il vertice sorgente da quello di destinazione

Operazione	Tempo di esecuzione
`grado`(v)	$O(m)$
`archiIncidenti`(v)	$O(m)$
`sonoAdiacenti`(x, y)	$O(m)$
`aggiungiVertice`(v)	$O(nm)$
`aggiungiArco`(x, y)	$O(nm)$
`rimuoviVertice`(v)	$O(nm)$
`rimuoviArco`(e)	$O(n)$