

Algoritmi e Strutture Dati

Informazioni sul corso

+

Un'introduzione informale
agli algoritmi

Fabio Patrizi



Informazioni Generali

Fabio Patrizi

Email: patrizi@dis.uniroma1.it

Web: <http://www.dis.uniroma1.it/~patrizi>

Libro di testo adottato:

C. Demetrescu, I. Finocchi, G. F. Italiano: Algoritmi e strutture dati, McGraw-Hill, seconda edizione, Gennaio 2008.

Pagina web del corso:

<http://www.dis.uniroma1.it/~patrizi/sections/teaching/asd-latina-16-17/index.html>

Ricevimento: prima/dopo lezione (inviare un'email il giorno precedente)



Obiettivi del corso

Obiettivi:

- Illustrare le tecniche algoritmiche e le strutture dati fondamentali per risolvere in modo efficiente problemi computazionali.
- Introdurre gli strumenti fondamentali per valutare la qualità degli algoritmi e delle strutture dati

Impareremo a:

- Progettare algoritmi e strutture dati per risolvere problemi classici
- Misurare l'efficienza delle strutture dati e degli algoritmi
- Scegliere algoritmi e strutture dati per risolvere efficientemente un problema

Definizione informale di algoritmo

Insieme di passi elementari che, eseguiti meccanicamente, producono un determinato risultato

- Esempio: algoritmo **preparaCaffè**

algoritmo preparaCaffè

1. Svita la caffettiera.
2. Riempি d'acqua il serbatoio della caffettiera.
3. Inserisci il filtro.
4. Riempি il filtro con la polvere di caffè.
5. Avvita la parte superiore della caffettiera.
6. Metti la caffettiera, così predisposta, su un fornello acceso.
7. Spegni il fornello quando il caffè è pronto.
8. Versa il caffè nella tazzina.

Perché studiare gli algoritmi?

- Gli algoritmi sono la base dei programmi:
forniscono il procedimento per giungere
alla soluzione di un problema

Pseudocodice

- Per mantenerci generali useremo lo pseudocodice:
 - ricorda linguaggi di programmazione reali come C, C++ o Java
 - può contenere alcune frasi in Italiano

Tuttavia, in molte esercitazioni, useremo il **linguaggio C** (di cui siete esperti!)

Correttezza ed efficienza

Vogliamo progettare algoritmi che:

- Producano correttamente il risultato desiderato
- Siano **efficienti** in termini di tempo di esecuzione ed occupazione di memoria

Perché analizzare algoritmi?

- L'analisi teorica sembra essere più affidabile di quella sperimentale: vale su **tutte le possibili istanze di dati su cui l'algoritmo opera**
- Ci aiuta a scegliere tra diverse soluzioni allo stesso problema
- Permette di predire le prestazioni di un programma prima ancora di iniziare l'implementazione

Un primo esempio: i numeri di Fibonacci

(un problema semplice, ma con molte soluzioni)

L'isola dei conigli

Leonardo da Pisa (anche noto come Fibonacci) si interessò di molte cose, tra cui il seguente problema di dinamica delle popolazioni:

Quanto velocemente si espanderebbe una popolazione di conigli sotto appropriate condizioni?

In particolare, partendo da una coppia di conigli in un'isola deserta, quante coppie si avrebbero nell'anno n?

Le regole di riproduzione

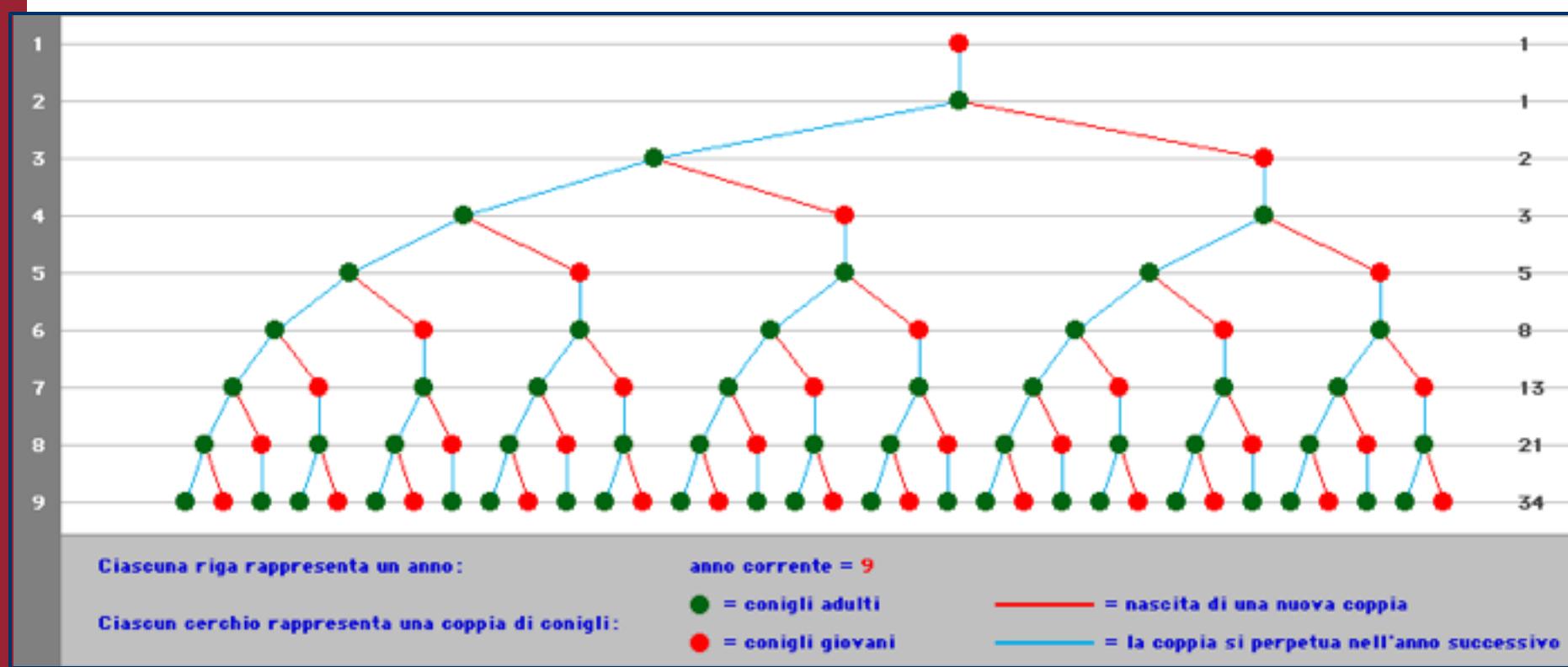
- Una coppia di conigli genera due coniglietti ogni anno
- I conigli cominciano a riprodursi soltanto dal secondo anno dopo la loro nascita
- I conigli sono immortali

Tasso di riproduzione dei conigli

- F_n : numero di coppie di conigli presenti nell'anno n
- $F_1 = 1$ (una sola coppia)
- $F_2 = 1$ (troppo giovani per riprodursi)
- $F_3 = 2$ (prima coppia di coniglietti)
- $F_4 = 3$ (seconda coppia di coniglietti)
- $F_5 = 5$ (prima coppia di nipotini)

L'albero dei conigli

La riproduzione dei conigli può essere descritta in un albero come segue:



La regola di espansione

- Nell'anno n , ci sono tutte le coppie dell'anno precedente, e una nuova coppia di conigli per ogni coppia presente due anni prima
- Indicando con F_n il numero di coppie dell'anno n , abbiamo la seguente **relazione di ricorrenza**:

$$F_n = \begin{cases} F_{n-1} + F_{n-2} & \text{se } n \geq 3 \\ 1 & \text{se } n=1,2 \end{cases}$$

Il problema

Dato un intero positivo n , quanto è F_n , ovvero il numero totale di conigli presenti all'anno n ?

Un approccio numerico

- Possiamo usare una funzione matematica che calcoli direttamente i numeri di Fibonacci.
- Si può dimostrare che:

$$F_n = \frac{1}{\sqrt{5}} (\phi^n - \hat{\phi}^n)$$

dove:

$$\begin{aligned}\phi &= \frac{1+\sqrt{5}}{2} \approx +1.618 \\ \hat{\phi} &= \frac{1-\sqrt{5}}{2} \approx -0.618\end{aligned}$$

Algoritmo fibonaccil

algoritmo fibonaccil(*intero n*) → *intero*
return $\frac{1}{\sqrt{5}} (\phi^n - \hat{\phi}^n)$

Correttezza?

- Quale deve essere l'accuratezza di ϕ e $\hat{\phi}$ per ottenere un risultato corretto?
- Ad esempio, con 3 cifre decimali:

$$\phi \approx 1.618 \text{ e } \hat{\phi} \approx -0.618$$

n	<code>fibonacci1(n)</code>	arrotondamento	F_n
3	1.99992	2	2
16	986.698	987	987
18	2583.1	2583	2584

Algoritmo fibonacci2

`fibonacci1` non è *corretto*!

Approccio alternativo: definizione ricorsiva

algoritmo fibonacci2(*intero n*) → *intero*
if (*n*≤2) **then return** 1
else return fibonacci2(*n*-1) +
 fibonacci2(*n*-2)

(OK: Opera solo con numeri interi)

Domande tipiche di questo corso

- Quanto *tempo* richiede fibonacci2?
- Come *misuriamo* il tempo?
 - In secondi (dipende dalla piattaforma)?
 - In numero di istruzioni macchina (dipende dal compilatore,...)?
- Prima approssimazione:
 - numero di linee di codice mandate in esecuzione (indipendente dalla piattaforma e compilatore)
- (La misura è funzione dell'input n)

Tempo di esecuzione

- Calcoliamo il **numero di linee di codice** mandate in esecuzione
 - misura indipendente dalla piattaforma utilizzata
- Se $n \leq 2$: una sola linea di codice
- Se $n = 3$: quattro linee di codice:
 - due per la chiamata **fibonacci2(3)**,
 - una per la chiamata **fibonacci2(2)** e
 - una per la chiamata **fibonacci2(1)**

Relazione di ricorrenza

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

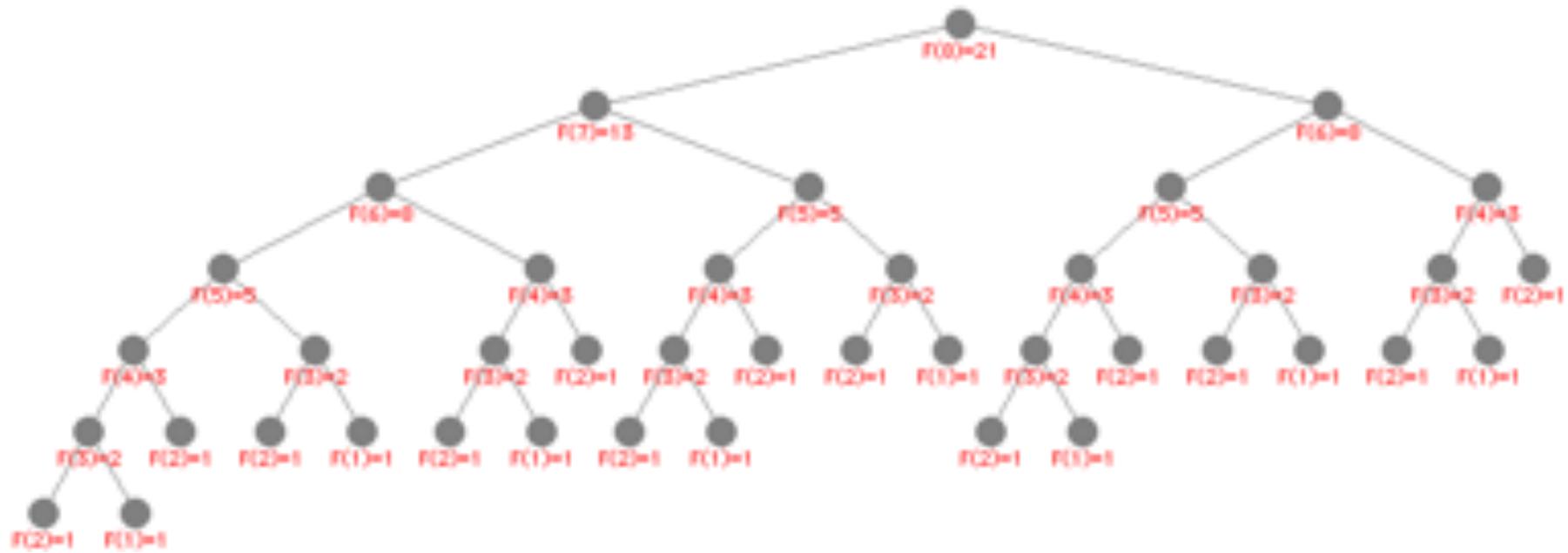
$$T(n) = 2 + T(n-1) + T(n-2)$$

In generale, il tempo richiesto da un algoritmo ricorsivo è pari al tempo speso all'interno della chiamata più il tempo speso nelle chiamate ricorsive

Albero della ricorsione

Utile per risolvere la relazione di ricorrenza:

- Nodi corrispondono a chiamate ricorsive
 - Figli corrispondono a sottochiamate

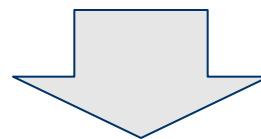


Calcolare $T(n)$

- Etichettiamo i nodi con # di linee di codice eseguite nella chiamata corrispondente:
 - Etichetta nodi interni: 2
 - Etichetta foglie: 1
- Per calcolare $T(n)$:
 - Contiamo il numero di foglie (come?)
 - Contiamo il numero di nodi interni (come?)

Calcolare T(n)

- Il numero di foglie dell'albero della ricorsione di **fibonacci2(n)** è pari a $F(n)$
- Il numero di nodi interni di un albero in cui ogni nodo ha due figli è pari al numero di foglie - 1



- In totale le linee di codice eseguite sono

$$F(n) + 2(F(n)-1) = 3F(n)-2$$

Osservazioni

`fibonacci2` è un algoritmo lento:

$$T(n) \approx F(n) \approx \phi^n$$

Crescita esponenziale in n

Possiamo fare di meglio?

Algoritmo fibonacci3

- fibonacci2 è lento perché calcola più volte la soluzione dello stesso sottoproblema.
- Soluzione: memorizzare in un array le soluzioni precedentemente calcolate

algoritmo fibonacci3(*intero n*) → *intero*
sia *Fib* un array di *n* interi
 $Fib[1] \leftarrow Fib[2] \leftarrow 1$
for {*i* = 3; *i* ≤ *n*; *i*++} **do**
 $Fib[i] \leftarrow Fib[i-1] + Fib[i-2]$
return *Fib[n]*

Calcolo del tempo di esecuzione

- L'algoritmo `fibonacci3` impiega tempo proporzionale a n invece che esponenziale in n come `fibonacci2`
- Tempo effettivo richiesto da implementazioni in C dei due algoritmi su piattaforme diverse:

	<code>fibonacci2(58)</code>	<code>fibonacci3(58)</code>
Pentium IV 1700MHz	15820 sec. (\simeq 4 ore)	0.7 milionesimi di secondo
Pentium III 450MHz	43518 sec. (\simeq 12 ore)	2.4 milionesimi di secondo
PowerPC G4 500MHz	58321 sec. (\simeq 16 ore)	2.8 milionesimi di secondo

Occupazione di memoria

- Il tempo di esecuzione non è la sola risorsa di calcolo. Anche la **quantità di memoria necessaria** può essere cruciale
- Un algoritmo lento impiega solo più tempo per calcolare il risultato
- Se un algoritmo richiede più spazio di quello a disposizione, il calcolo **non può** essere completato

Algoritmo fibonacci4

- fibonacci3 usa un array di dimensione n
- Non serve mantenere tutti i valori di F_n precedenti: bastano gli ultimi due!

algoritmo fibonacci4(*intero n*) \rightarrow *intero*

a \leftarrow *b* \leftarrow 1

for {*i* = 3; *i* \leq *n*; *i*++} **do**{

c \leftarrow *a*+*b*

a \leftarrow *b*

b \leftarrow *c*

}

return *b*

Notazione asintotica (1 di 4)

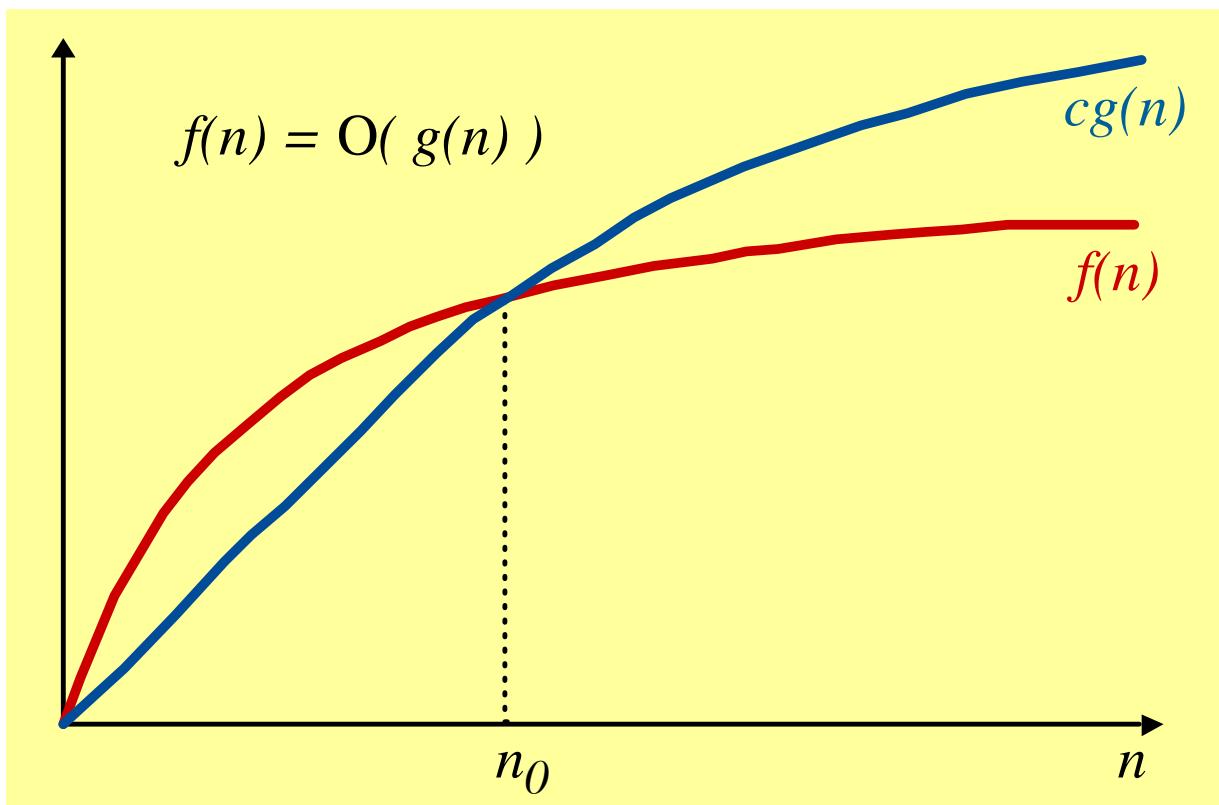
- Misurare $T(n)$ come il numero di linee di codice mandate in esecuzione è una misura molto approssimativa del tempo di esecuzione
- Se andiamo a capo più spesso, aumenteranno le linee di codice sorgente, ma certo non il tempo richiesto dall'esecuzione del programma!

Notazione asintotica (2 di 4)

- Per lo stesso programma impaginato diversamente potremmo concludere ad esempio che $T(n)=3n$ oppure $T(n)=5n$
- Vorremmo un modo per descrivere l'*ordine di grandezza* di $T(n)$ ignorando dettagli inessenziali
- Usiamo a questo scopo la notazione asintotica O ("O grande")

Notazione asintotica (3 di 4)

- Diremo che $f(n) = O(g(n))$ se $f(n) < c g(n)$ per qualche costante c , ed n abbastanza grande



Notazione asintotica (4 di 4)

La notazione O-grande ci permette di **valutare il limite dell'andamento di crescita di una funzione** (logaritmico, polinomiale, lineare, esponenziale, etc.)

Ad esempio:

- per $T(n)=3n + 4$, abbiamo $T(n)=O(n)$
- per $T(n)=100n + 50$, abbiamo $T(n)=O(n)$
- per $T(n)=2^n + 1$, abbiamo $T(n)=O(2^n)$
- per $T(n)=2^n + n^4 + 3$, abbiamo $T(n)=O(2^n)$

Un nuovo algoritmo

È possibile calcolare F_n in tempo inferiore a $O(n)$?

Potenze ricorsive

- `fibonacci4` non è il miglior algoritmo possibile
- E' possibile dimostrare per induzione la seguente proprietà di matrici:

$$\begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^n = \begin{bmatrix} F_{n+1} & F_n \\ F_n & F_{n-1} \end{bmatrix}$$

- Useremo questa proprietà per progettare un algoritmo più efficiente

Algoritmo fibonacci5

algoritmo fibonacci5(*interno n*) → *intero*

1. $M \leftarrow \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$
2. **for** $i = 1$ **to** $n - 1$ **do**
3. $M \leftarrow M \cdot \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}$
4. **return** $M[0][0]$

- Il tempo di esecuzione è ancora $O(n)$
- Cosa abbiamo guadagnato?

Calcolo di potenze

- Possiamo calcolare la n-esima potenza di un numero moltiplicando questo numero per se stesso n volte.

$$3^8 = 3 \cdot 3 = 6561$$

- Oppure ricorrere a quadrati ripetuti:

$$3^2 = 9, 9^2 = 3^4 = 81, 81^2 = 3^8 = 6561$$

ovvero: $3 \cdot 3$ poi $9 \cdot 9$ poi $81 \cdot 81$

Se n è dispari eseguo un'ulteriore moltiplicazione

Algoritmo fibonacci6

algoritmo fibonacci6(*intero n*) → *intero*

1. $M \leftarrow \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$
2. **potenzaDiMatrice**($M, n - 1$)
3. **return** $M[0][0]$

procedura potenzaDiMatrice(*matrice M, intero n*)

4. **if** ($n > 1$) **then**
5. **potenzaDiMatrice**($M, n/2$)
6. $M \leftarrow M \cdot M$
7. **if** (n è dispari) **then** $M \leftarrow M \cdot \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}$

Tempo di esecuzione

- Tutto il tempo è speso nella procedura `potenzaDiMatrice`
 - All'interno della procedura si spende tempo costante
 - Si esegue una chiamata ricorsiva con input $n/2$
- La relazione di ricorrenza è pertanto:

$$T(n) = \begin{cases} O(1) + T(n/2) & \text{se } n > 1 \\ O(1) & \text{se } n \leq 1 \end{cases}$$

- Come risolverla?

Metodo dell'iterazione

$$T(n) \leq c + T(n/2) \leq c + c + T(n/4) = 2c + T(n/2^2)$$

In generale:

$$T(n) \leq kc + T(n/2^k)$$

Per $k=\log_2 n$ si ottiene

$$T(n) \leq c \log_2 n + T(1) = O(\log_2 n)$$

fibonacci6 è quindi esponenzialmente più veloce di **fibonacci3**!

Riepilogo

	Tempo di esecuzione	Occupazione di memoria
fibonacci2	$O(2^n)$	$O(n)$
fibonacci3	$O(n)$	$O(n)$
fibonacci4	$O(n)$	$O(1)$
fibonacci5	$O(n)$	$O(1)$
fibonacci6	$O(\log n)$	$O(\log n)$

Algoritmi e Strutture Dati

Modelli di calcolo e
metodologie di analisi

Fabio Patrizi

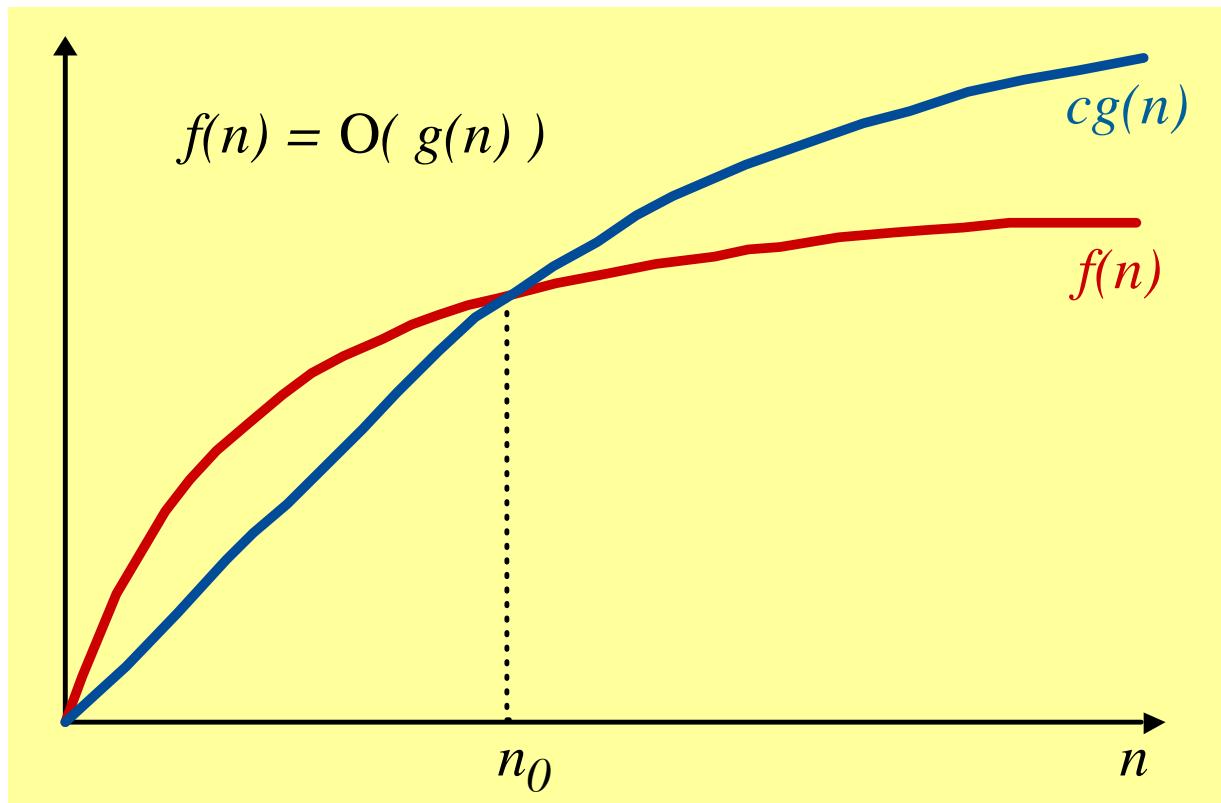
Notazione asintotica

- $f(n)$ = 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 di $f(n)$ ignorando i dettagli non influenti, come costanti moltiplicative e termini di ordine inferiore

Notazione asintotica O

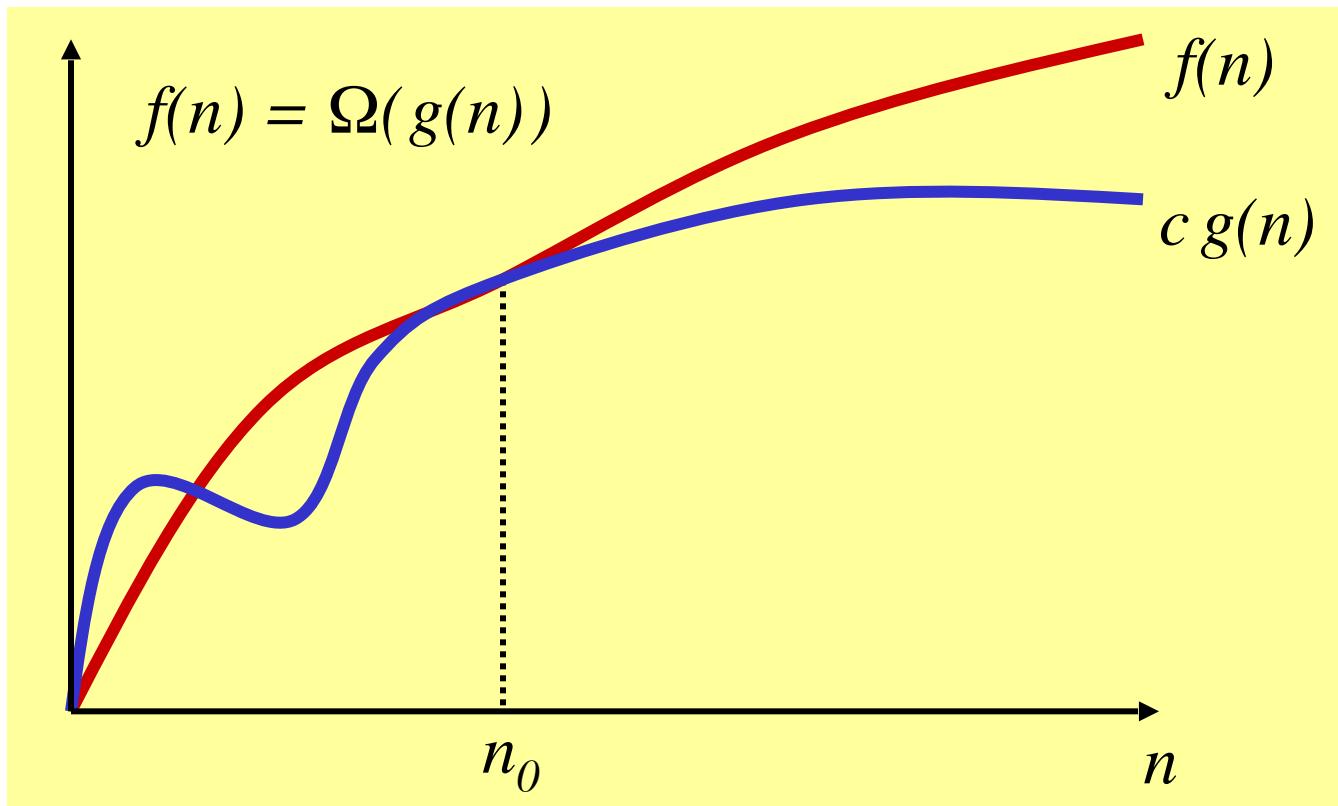
$f(n) = O(g(n))$ se \exists due costanti $c > 0$ e $n_0 \geq 0$ t.c.

$$f(n) \leq c g(n) \text{ per ogni } n \geq n_0$$



Notazione asintotica Ω

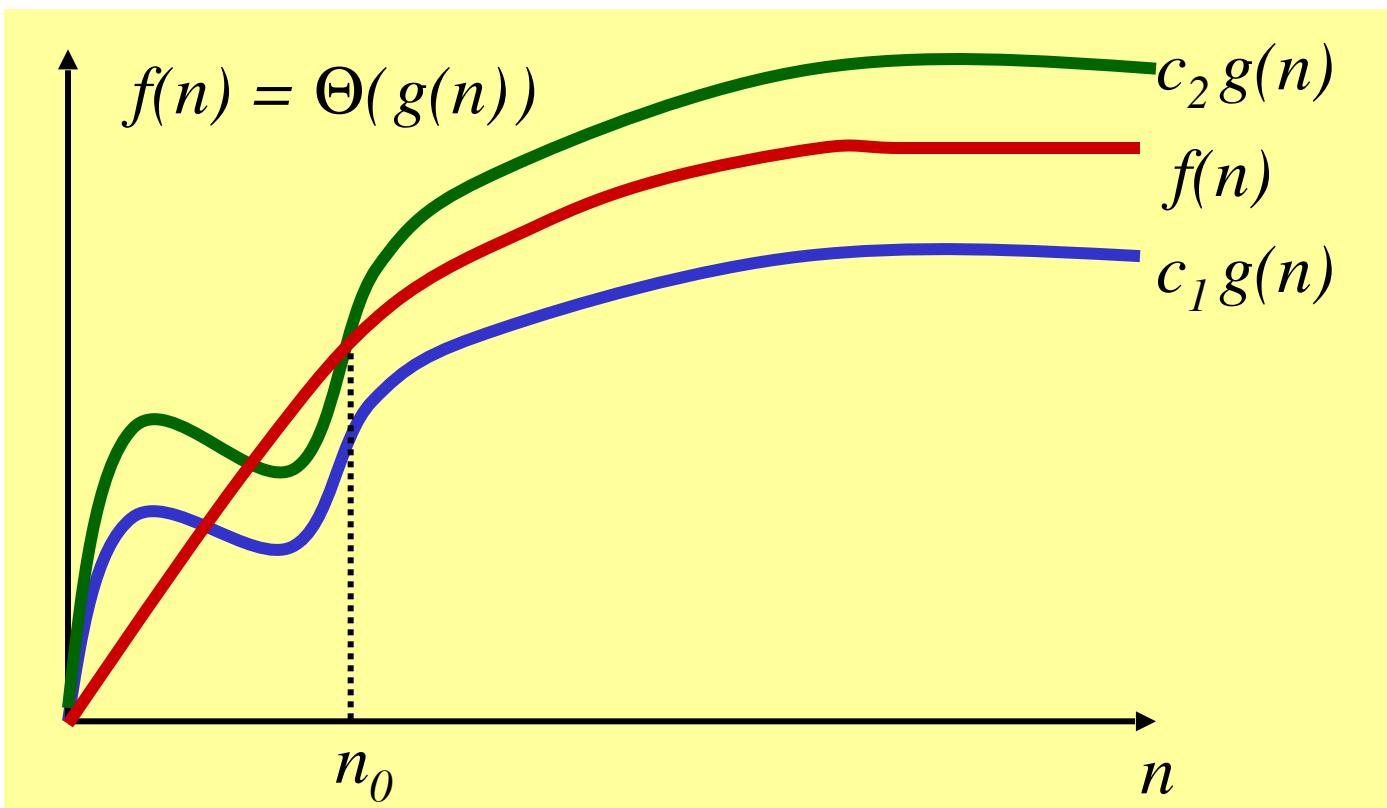
$f(n) = \Omega(g(n))$ se \exists due costanti $c > 0$ e $n_0 \geq 0$ t.c.
 $f(n) \geq c g(n)$ per ogni $n \geq n_0$



Notazione asintotica Θ

$f(n) = \Theta(g(n))$ se \exists tre costanti $c_1, c_2 > 0$ e $n_0 \geq 0$ t.c.

$c_1 g(n) \leq f(n) \leq c_2 g(n)$ per ogni $n \geq n_0$



Notazione asintotica Θ

Proprietà:

Date due funzioni $f(n)$ e $g(n)$, abbiamo che

$f(n) = \Theta(g(n))$ se e solo se

- $f(n) = O(g(n))$
- $f(n) = \Omega(g(n))$

Notazione asintotica: esempi

Consideriamo $g(n)=3n^2+10$

- $g(n)=O(n^2)$: scegliere $c=4$ e $n_0=10$
- $g(n)=\Omega(n^2)$: scegliere $c=1$ e $n_0=0$
- $g(n)=\Theta(n^2)$ (da quanto sopra)
- $g(n)=O(n^3)$ ma $g(n)\neq\Theta(n^3)$, in quanto:
 $g(n)\neq\Omega(n^3)$

Metodi di analisi

Ricerca Sequenziale

Ricerca di un elemento x in una lista L non ordinata

algoritmo ricercaSequenziale(*lista L, elem x*) \rightarrow booleano
1. **for each** ($y \in L$) **do**
2. **if** ($y = x$) **then return** trovato
3. **return** non trovato

Quanti confronti dobbiamo fare per trovare x in L ?

- Dipende da dove si trova x (all'inizio? Alla fine? E se non c'è?)
- Vorremmo una risposta che non sia «dipende».

Caso peggiore, migliore e medio

- Misureremo le risorse di calcolo (tempo di esecuzione / occupazione di memoria) usate da un algoritmo in funzione della *dimensione* n dell'istanza d'ingresso.
- A parità di dimensione, istanze diverse potrebbero richiedere risorse diverse
- Distinguiamo quindi ulteriormente tra analisi nel caso **peggiore, migliore e medio**

Caso peggiore

- Sia $\text{tempo}(I)$ il tempo di esecuzione di un algoritmo sull'istanza I

$$T_{\text{worst}}(n) = \max_{\text{istanze } I \text{ di dimensione } n} \{\text{tempo}(I)\}$$

- Intuitivamente, $T_{\text{worst}}(n)$ è il tempo di esecuzione sulle istanze di ingresso che comportano più lavoro per l'algoritmo
- Dà garanzia sulle prestazioni

Caso migliore

- Sia $\text{tempo}(I)$ il tempo di esecuzione di un algoritmo sull'istanza I

$$T_{\text{best}}(n) = \min_{\text{istanze } I \text{ di dimensione } n} \{\text{tempo}(I)\}$$

- Intuitivamente, $T_{\text{best}}(n)$ è il tempo di esecuzione sulle istanze di ingresso che comportano meno lavoro per l'algoritmo
- Non molto informativo

Caso medio

- Sia $P(I)$ la probabilità di avere in ingresso un'istanza I

$$T_{\text{avg}}(n) = \sum_{\text{istanze } I \text{ di dimensione } n} \{P(I) * \text{tempo}(I)\}$$

- Intuitivamente, $T_{\text{avg}}(n)$ è il tempo di esecuzione nel caso medio (ovvero sulle istanze di ingresso “tipiche” per il problema)
- Richiede conoscenza di una distribuzione statistica dell’input

Analisi della ricerca sequenziale (1/3)

algoritmo ricercaSequenziale(*lista L, elem x*) → booleano
1. **for each** ($y \in L$) **do**
2. **if** ($y = x$) **then return** trovato
3. **return** non trovato

Caso Migliore:

Nel caso migliore l'algoritmo esegue un solo confronto trovando immediatamente x in prima posizione.

Quindi: $T_{best}(n) = 1$

Analisi della ricerca sequenziale (2/3)

algoritmo ricercaSequenziale(*lista L, elem x*) → booleano

1. **for each** ($y \in L$) **do**
2. **if** ($y = x$) **then return** trovato
3. **return** non trovato

Caso Peggio:

Il caso peggiore ha luogo quando x si trova in ultima posizione oppure quando x non è in L.

Quindi: $T_{worst}(n) = n$

Analisi della ricerca sequenziale (3/3)

algoritmo ricercaSequenziale(*lista L, elem x*) → booleano
1. **for each** ($y \in L$) **do**
2. **if** ($y = x$) **then return** trovato
3. **return** non trovato

Caso Medio:

Indichiamo con **I_i** il caso in cui x si trova nella posizione **i** di L. Supponendo che x può occupare ogni posizione con la medesima probabilità, per ogni **i** abbiamo che:

$$P(I_i) = 1/n \text{ e } tempo(I_i) = i$$

Quindi:

$$T_{avg}(n) = \sum_{1 \leq i \leq n} \{P(I_i) * tempo(I_i)\} = \sum_{1 \leq i \leq n} (1/n) * i = (n + 1)/2$$



Ricerca Binaria

Ricerca di un elemento x in un array L ordinato

algoritmo ricercaBinariaIter(*array L, elem x*) \rightarrow booleano

1. $a \leftarrow 1$
2. $b \leftarrow$ lunghezza di L
3. **while** ($L[(a + b)/2] \neq x$) **do**
4. $i \leftarrow (a + b)/2$
5. **if** ($L[i] > x$) **then** $b \leftarrow i - 1$
6. **else** $a \leftarrow i + 1$
7. **if** ($a > b$) **then return** non trovato
8. **return** trovato

Confronta x con l'elemento centrale di L e prosegue nella metà sinistra o destra in base all'esito del confronto

Analisi ricerca binaria

$$T_{\text{best}}(n) = 1$$

l'elemento centrale è uguale a x

$$T_{\text{worst}}(n) = O(\log n)$$

$$T_{\text{avg}}(n) = \log n - 1 + (1/n)$$

Analisi di algoritmi ricorsivi

Ricerca Binaria – versione ricorsiva

L’algoritmo di ricerca binaria può essere riscritto ricorsivamente come:

algoritmo ricercaBinariaRic(*elemento x, array L*) \rightarrow booleano

1. $n \leftarrow$ lunghezza di L
2. **if** ($n = 0$) **then return** "elemento non trovato"
3. $i \leftarrow \lceil n/2 \rceil$
4. **if** ($L[i] = x$) **return** "trovato"
5. **else if** ($L[i] > x$) **then return** ricercaBinariaRic($x, L[1,i-1]$)
6. **else return** ricercaBinariaRic($x, L[i+1,n]$)

Come analizzarlo?

Relazioni di ricorrenza

Il tempo di esecuzione può essere descritto tramite una relazione di ricorrenza:

$$T(n) = \begin{cases} c + T(\lceil (n-1)/2 \rceil) & \text{se } n > 1 \\ 1 & \text{se } n = 1 \end{cases}$$

simile a fibonacci

Vari metodi per risolvere equazioni di ricorrenza:
iterazione, sostituzione, teorema Master

1. Metodo dell'iterazione

Idea:

Partendo dalla relazione di ricorrenza, “srotoliamo” la ricorsione, ottenendo una sommatoria dipendente solo dalla dimensione n del problema iniziale (e dal numero k di passi ricorsivi eseguiti, che vorremmo eliminare).

1. Metodo dell'iterazione - esempio

$$T(k) = \begin{cases} c + T(k/2) & \text{se } k > 1 \\ 1 & \text{se } k = 1 \end{cases}$$

$$T(n) = c + T(n/2) \quad T(n/2) = c + T(n/4) \quad T(n/4) = c + T(n/8)$$

Dopo i passi: $T(n) = c + T(n/2) = 2c + T(n/4) = \dots = i*c + T(n/2^i)$

Procedendo finché $n/2^i = 1$, cioè $i = \log_2 n$, otteniamo:

$$T(n) = c + T(n/2) = 2c + T(n/4) = \dots = i*c + T(n/2^i) = i*c + T(1)$$

Quindi, sostituendo $i = \log_2 n$: $T(n) = c * \log n + T(1) = \mathbf{O(\log n)}$

Analisi ricerca binaria

$$T_{\text{best}}(n) = 1$$

l'elemento centrale è uguale a x

$$T_{\text{worst}}(n) = O(\log n)$$

$x \notin L$ oppure viene trovato
all'ultimo confronto

$$T_{\text{avg}}(n) = \log(n - 1) + 1/n$$

assumendo che le istanze siano
equidistribuite

2. Metodo della sostituzione

Idea:

“Intuire” la soluzione di una relazione di ricorrenza ed utilizzare l’induzione matematica per dimostrare che la soluzione è effettivamente quella ipotizzata.

2. Metodo della sostituzione - esempio

Consideriamo la seguente relazione di ricorrenza:

$$T(n) = \begin{cases} T(\lceil n/2 \rceil) + n & \text{se } n > 1 \\ 1 & \text{se } n = 1 \end{cases}$$

Ipotesi: $T(n) \leq c n$ (per una opportuna $c > 0$) quindi

$$T(n) = O(n)$$

Passo base: $T(1) \leq c 1$ (banalmente verificato)

Passo induttivo: $T(n) = T(\lceil n/2 \rceil) + n \leq c \lceil n/2 \rceil + n$

$$\leq c(n/2) + n = (c/2 + 1) n$$

Quindi $T(n) \leq c n$ per $c \geq 2$

Divide et Impera (1/2)

Tecnica molto potente e generale per la progettazione di algoritmi per la risoluzione di problemi.

Idea:

Dividere i dati di ingresso in sottoinsiemi (*divide*), risolvere ricorsivamente il problema sui sottoinsiemi, e ricombinare la soluzione dei sottoproblemi per ottenere la soluzione globale (*impera*).

Divide et Impera (2/2)

- Analizziamo la tecnica:
 - Abbiamo un problema di dimensione n ;
 - Il problema viene suddiviso in a sottoproblemi di dimensione n/b ciascuno;
 - Suddividere il problema costa $f(n)$;
- La relazione di ricorrenza sarà quindi:

$$T(n) = aT\left(\frac{n}{b}\right) + f(n)$$

- l'algoritmo **ricorsivo della ricerca binaria** e l'algoritmo **fibonacci6** sono esempi di applicazione della tecnica *Divide et Impera*, con $a = 1$, $b = 2$ e $f(n) = O(1)$.

Teorema Master (1/5)

Permette di analizzare algoritmi basati sulla tecnica del *divide et impera*, quindi con relazione di ricorrenza del tipo:

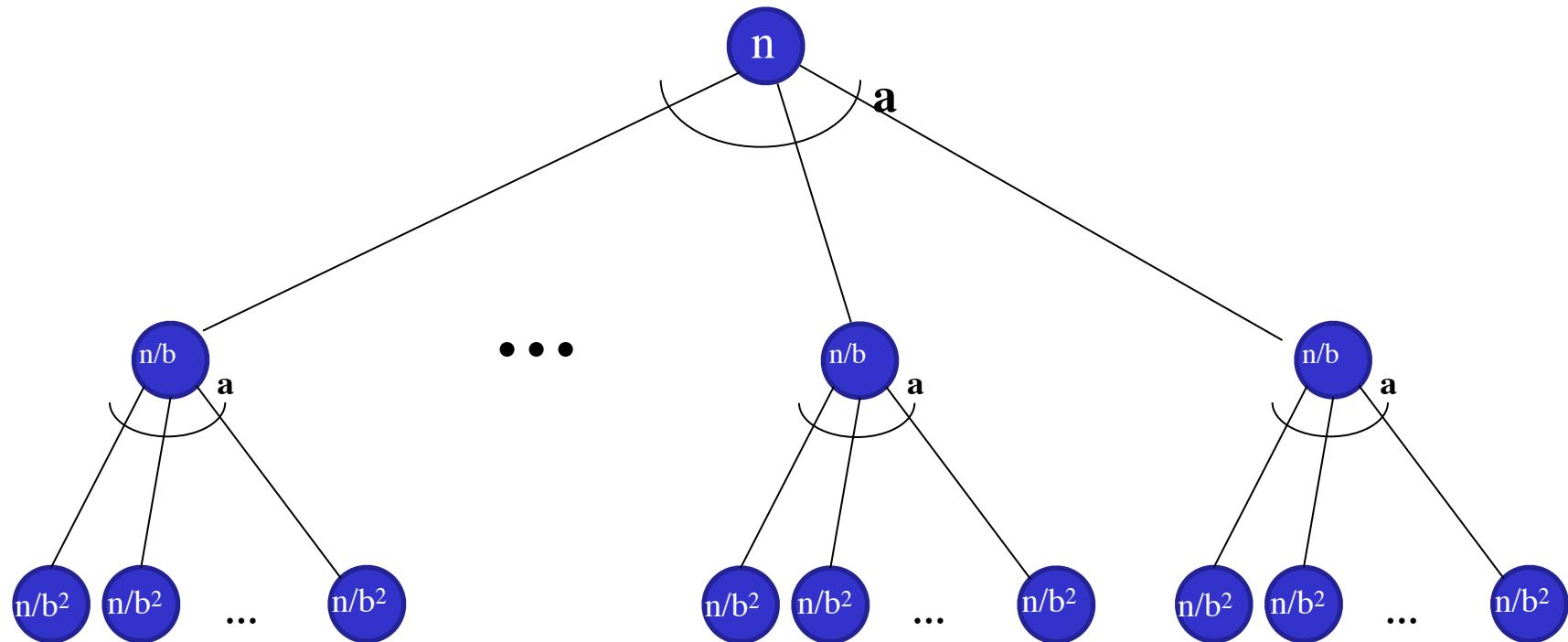
$$T(n) = \begin{cases} a T(n/b) + f(n) & \text{se } n > 1 \\ 1 & \text{se } n = 1 \end{cases}$$

Proviamo a risolvere la relazione...

Teorema Master (2/5)

Assumiamo per semplicità che n sia una potenza di b e che la ricorsione si fermi quando $n=1$

L'albero di ricorsione è:



Teorema Master (3/5)

$$T(n) = aT\left(\frac{n}{b}\right) + f(n)$$

Osservazione1: i sottoproblemi al livello **i** hanno dimensione **n/bⁱ**. Quindi (escluso il tempo per le chiamate ricorsive) il tempo speso al passo **i** è **f(n/bⁱ)**.

Osservazione2: ogni nodo interno ha **a** figli. Quindi sul livello **i** dell'albero di ricorsione avremo **aⁱ** nodi.

Teorema Master (4/5)

$$T(n) = aT\left(\frac{n}{b}\right) + f(n)$$

Osservazione3: all'ultimo livello abbiamo che l'input $n/b^i = 1$ quindi $n=b^i$ e quindi $i = \log_b n$
Segue che l'albero di ricorsione avrà al più altezza $\log_b n$

Quindi possiamo riscrivere la nostra relazione di ricorrenza come:

$$T(n) = \sum_{i=0}^{\log_b n} a^i f\left(\frac{n}{b^i}\right)$$

Teorema Master (5/5)

La soluzione alla precedente equazione è data dal *teorema fondamentale delle ricorrenze*, anche detto *teorema master*.

$$T(n) = \begin{cases} a T(n/b) + f(n) & \text{se } n > 1 \\ 1 & \text{se } n = 1 \end{cases}$$

ha soluzione:

1. $T(n) = \Theta(n^{\log_b a})$ se $f(n) = O(n^{\log_b a - \varepsilon})$ per $\varepsilon > 0$
2. $T(n) = \Theta(n^{\log_b a} \log n)$ se $f(n) = \Theta(n^{\log_b a})$
3. $T(n) = \Theta(f(n))$ se $f(n) = \Omega(n^{\log_b a + \varepsilon})$ per $\varepsilon > 0$ e $a * f(n/b) \leq c * f(n)$ per $c < 1$ e n sufficientemente grande

Teorema Master – Esempio 1

Consideriamo la relazione di ricorrenza della ricerca binaria:

$$T(n) = T(n/2) + O(1)$$

Abbiamo:

$$a = 1, b = 2 \text{ ed } f(n) = O(1)$$

Siccome $\Theta(n^{\log_b a}) = \Theta(1)$, siamo nel caso 2 del teorema e quindi:

$$T(n) = \Theta(n^{\log_b a} \log n) = \Theta(\log n).$$

Teorema Master – Esempio 2

Consideriamo la seguente relazione di ricorrenza

$$T(n) = 9T(n/3) + n$$

Abbiamo:

$$a = 9, b = 3 \text{ ed } f(n) = n$$

Segue che $n^{\log_b a} = n^{\log_3 9} = n^2$ e quindi, dato che $f(n) = n = O(n^{2-\varepsilon})$, ad esempio per $\varepsilon = 1$, siamo nel caso 1 del teorema. Da cui:

$$T(n) = \Theta(n^{\log_b a}) = \Theta(n^2).$$

Riepilogo

- Esprimiamo la quantità di una certa risorsa di calcolo (tempo, spazio) usata da un algoritmo **in funzione della dimensione n dell'istanza di ingresso**
- La **notazione asintotica** permette di esprimere la quantità di risorsa usata dall'algoritmo in modo sintetico, ignorando dettagli non influenti
- A parità di dimensione n, la quantità di risorsa usata può essere diversa, da cui la necessità di analizzare il **caso peggiore** o, se possibile, il **caso medio**
- La quantità di risorsa usata da algoritmi ricorsivi può essere espressa tramite **relazioni di ricorrenza**, risolvibili tramite vari metodi generali

Algoritmi e Strutture Dati

Strutture dati elementari

Fabio Patrizi

Gestione di collezioni di oggetti

Tipo di dato:

- Specifica delle operazioni di interesse su una collezione di oggetti (es. inserisci, cancella, cerca)

Struttura dati:

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

Il tipo di dato Dizionario

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 chiave k è presente in S restituisce l'elemento e ad essa associato,
e null altrimenti.

Il tipo di dato Pila

tipo Pila:

dati:

una sequenza S di n elementi.

operazioni:

`isEmpty() → result`

restituisce `true` se S è vuota, e `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).

- Disciplina di accesso *last-in first-out (LIFO)*

Il tipo di dato Coda

tipo Coda:

dati:

una sequenza S di n elementi.

operazioni:

`isEmpty() → result`

restituisce `true` se S è vuota, e `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).

- Disciplina di accesso *first-in first-out (FIFO)*

Tecniche di rappresentazione dei dati

Rappresentazioni indicizzate:

- I dati sono contenuti in array

Rappresentazioni collegate:

- I dati sono contenuti in record collegati fra loro mediante puntatori

Pro e contro

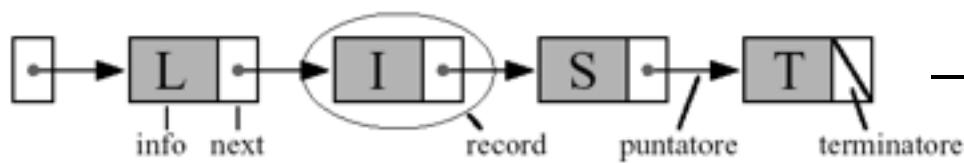
Rappresentazioni indicizzate:

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

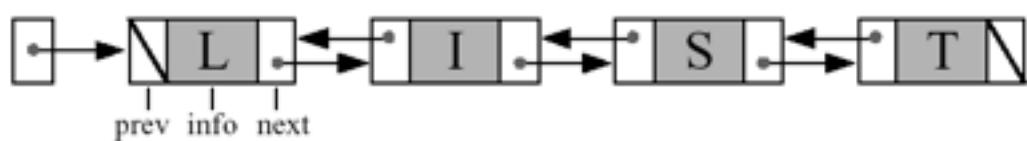
Rappresentazioni collegate:

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

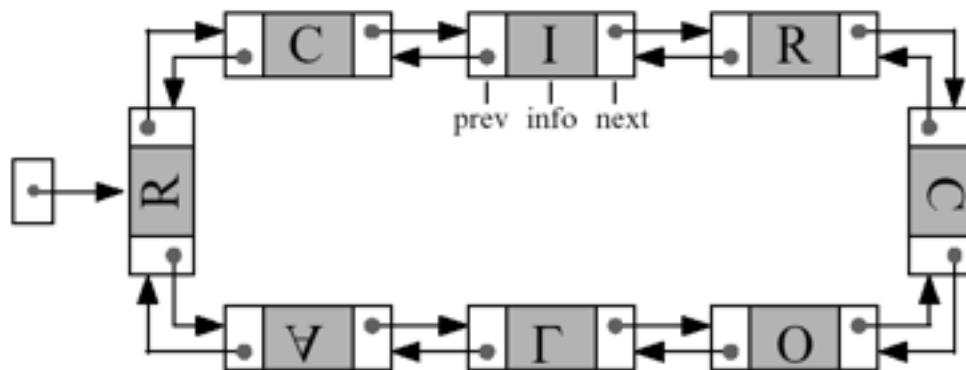
Esempi di strutture collegate



— Lista semplice

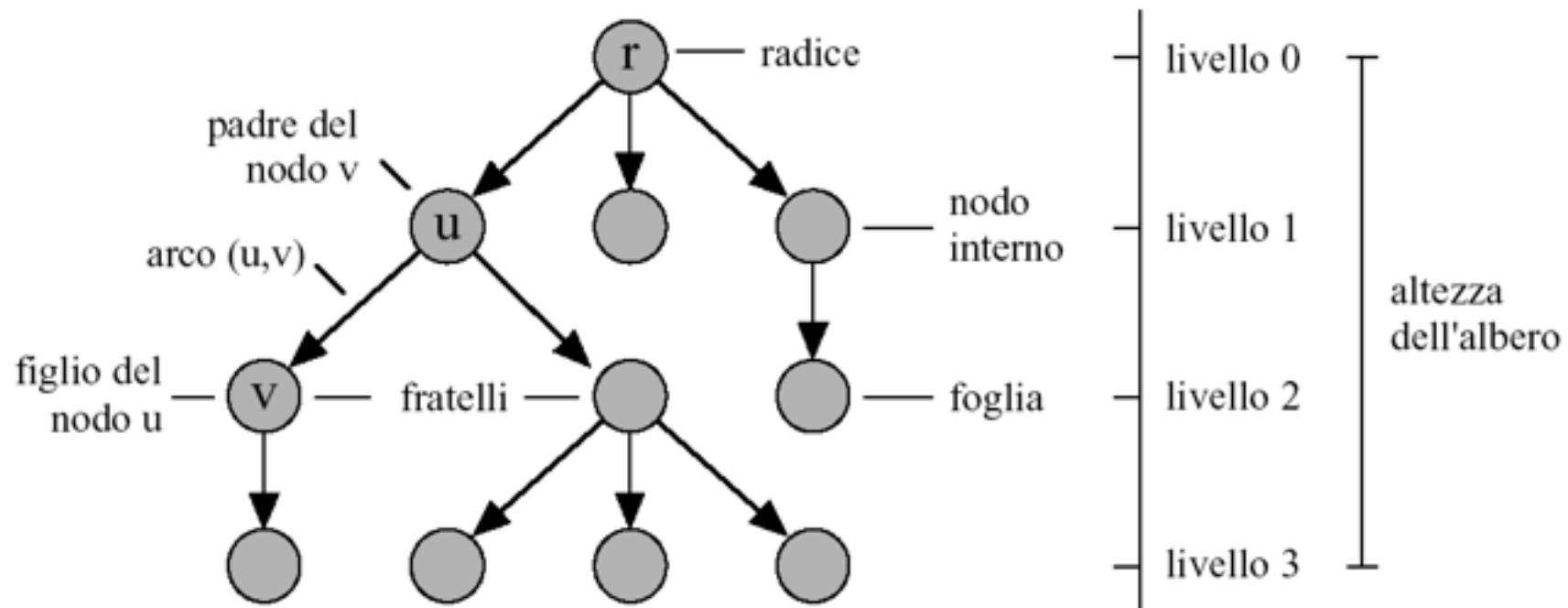


— Lista doppiamente collegata



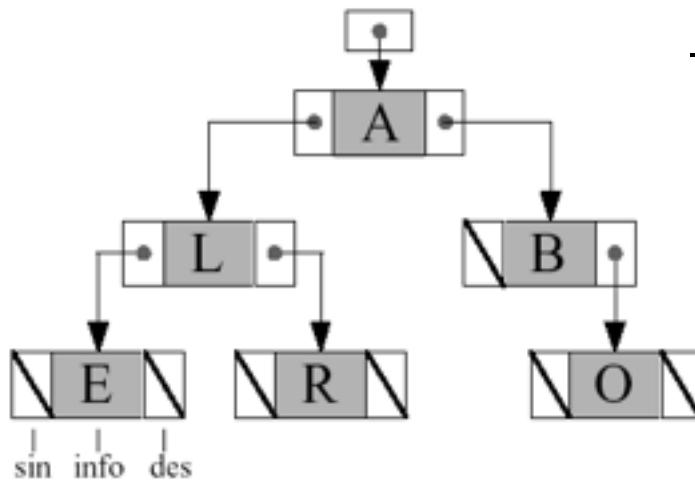
— Lista circolare doppiamente collegata

Organizzazione gerarchica dei dati



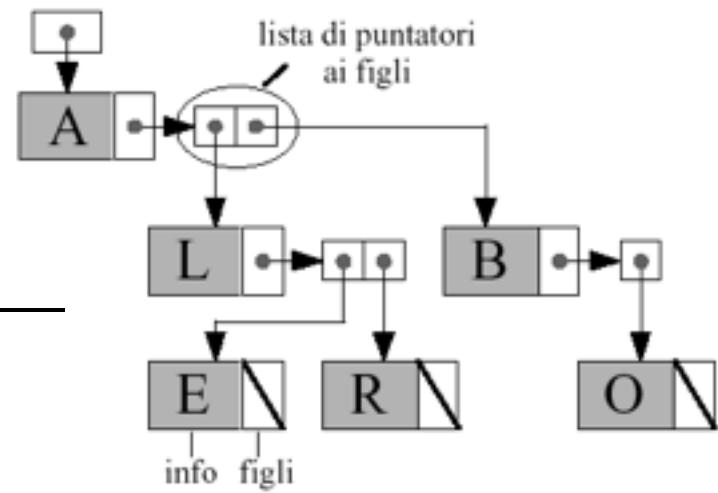
Dati contenuti nei nodi, relazioni gerarchiche definite dagli archi che li collegano

Rappresentazioni collegate di alberi

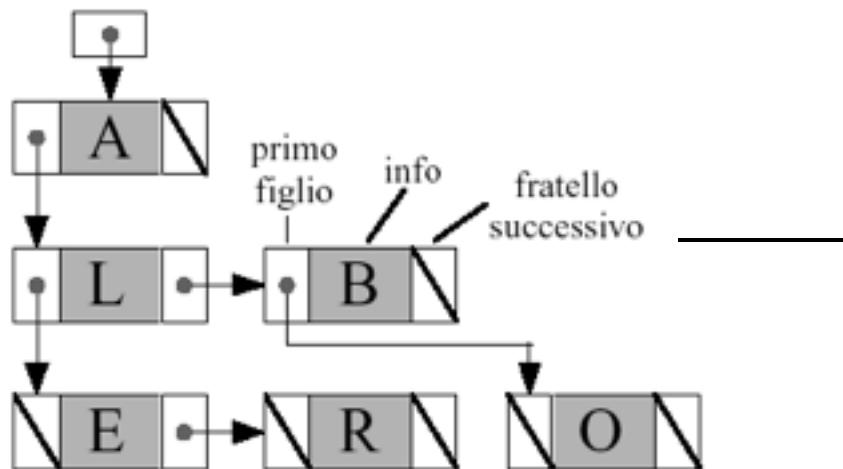


Rappresentazione
con puntatori ai figli
(nodi con numero
limitato di figli)

Rappresentazione
con liste di puntatori ai
figli (nodi con numero
arbitrario di figli)



Rappresentazioni collegate di alberi



Rappresentazione
di tipo primo figlio-fratello
successivo (nodi con numero
arbitrario di figli)

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

Algoritmo di visita generica

`visitaGenerica` visita il nodo r e tutti i suoi discendenti in un albero

```
algoritmo visitaGenerica(nodo r)
1.       $S \leftarrow \{r\}$ 
2.      while ( $S \neq \emptyset$ ) do
3.          estrai un nodo  $u$  da  $S$ 
4.          visita il nodo u
5.           $S \leftarrow S \cup \{ \text{figli di } u \}$ 
```

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

Algoritmo di visita in profondità

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



Algoritmo di visita in profondità

Versione iterativa (per alberi binari):

```
algoritmo visitaDFS(nodo r)
    Pila S
    S.push(r)
    while (not S.isEmpty()) do
        u  $\leftarrow$  S.pop()
        visita il nodo u
        S.push(figlio destro di u)
        S.push(figlio sinistro di u)
```

Algoritmo di visita in profondità

Versione ricorsiva (per alberi binari):

```
algoritmo visitaDFSRicorsiva(nodo r)
1.   if (r = null) then return
2.   visita il nodo r
3.   visitaDFSRicorsiva(figlio sinistro di r)
4.   visitaDFSRicorsiva(figlio destro di r)
```

Algoritmo di visita in ampiezza

L'algoritmo di visita in ampiezza (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.

Algoritmo di visita in ampiezza

Versione iterativa (per alberi binari):

```
algoritmo visitaBFS(nodo r)
    Coda C
    C.enqueue(r)
    while (not C.isEmpty()) do
        u  $\leftarrow$  C.dequeue()
        visita il nodo u
        C.enqueue(figlio sinistro di u)
        C.enqueue(figlio destro di u)
```

Riepilogo

- Nozione di tipo di dato come specifica delle operazioni su una collezione di oggetti
- Rappresentazioni indicizzate e collegate di collezioni di dati: pro e contro
- Organizzazione gerarchica dei dati mediante alberi
- Rappresentazioni collegate classiche di alberi
- Algoritmi di esplorazione sistematica dei nodi di un albero (algoritmi di visita)

Algoritmi e Strutture Dati

Ordinamento

Fabio Patrizi

Ordinamento

Problema: dato un insieme S di n oggetti presi da un dominio totalmente ordinato, ordinare S

Esempi: ordinare una lista di nomi alfabeticamente, o un insieme di numeri, o un insieme di compiti d'esame in base al cognome dello studente.

- Subroutine in molti problemi

E' possibile effettuare ricerche in array ordinati in tempo $O(\log n)$ (ricerca binaria)

Algoritmi e tempi tipici

- Numerosissimi algoritmi
- Tre tempi tipici: $O(n^2)$, $O(n \log n)$, $O(n)$

n	10	100	1000	10^6	10^9
$n \log_2 n$	~ 33	~ 665	$\sim 10^4$	$\sim 2 \cdot 10^7$	$\sim 3 \cdot 10^{10}$
n^2	100	10^4	10^6	10^{12}	10^{18}

Algoritmi di ordinamento

Approccio incrementale

Approccio incrementale:

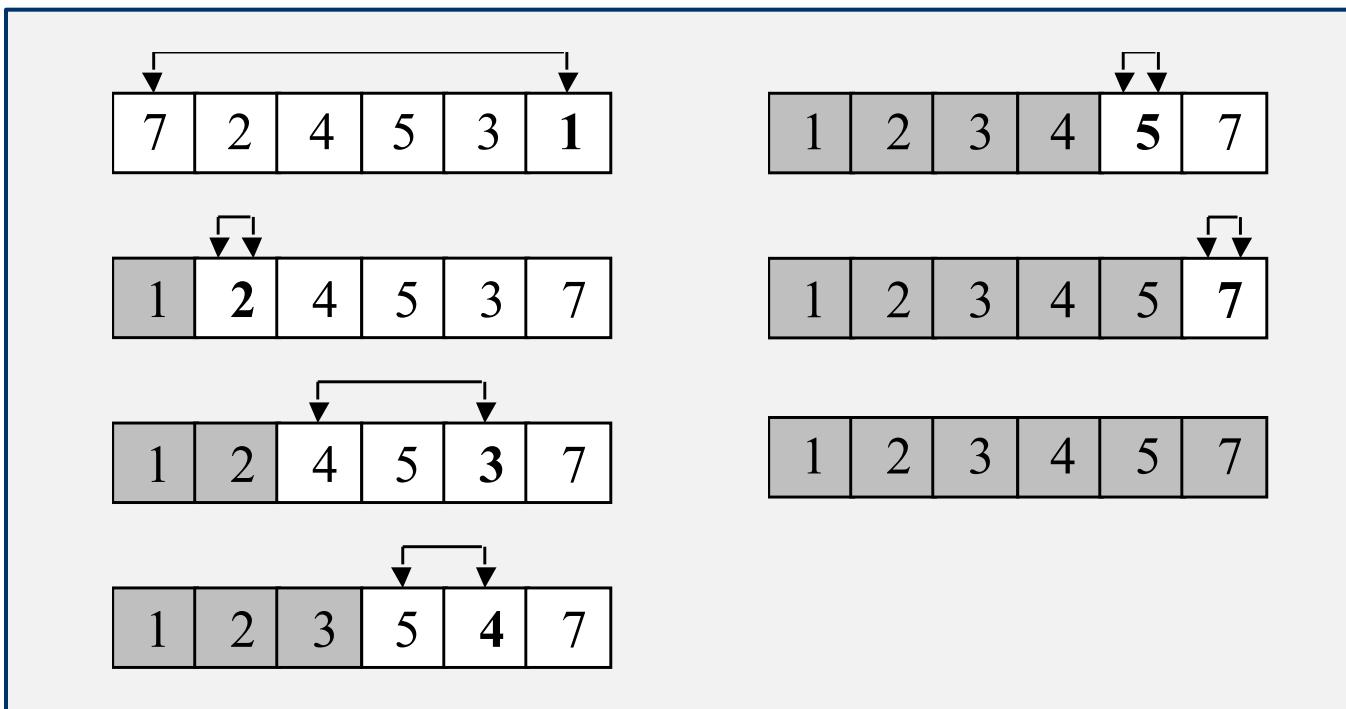
- Supponiamo che k elementi di un array di n siano già ordinati.
- Estendiamo l'ordinamento da k a $k+1$.

Come fare?

- **selectionSort**
- **insertionSort**

SelectionSort

Idea: estende l'ordinamento da k a $k+1$ elementi, scegliendo il minimo degli $n-k$ elementi non ancora ordinati e mettendolo in posizione $k+1$



SelectionSort: pseudocodice

```
Algoritmo: selectionSort(array S di dimensione n)

for(k = 0; k < n-1; k++) {
    min = k;
    for(i = k+1; i < n; i++) {
        if(S[i] < S[min]) then min = i;
    }
    scambia S[min] e S[k];
}
```

SelectionSort: analisi

- Il ciclo esterno viene eseguito $n-1$ volte ($k=0, \dots, n-2$)
- Il k -esimo ciclo interno viene eseguito $n-k+1$ volte

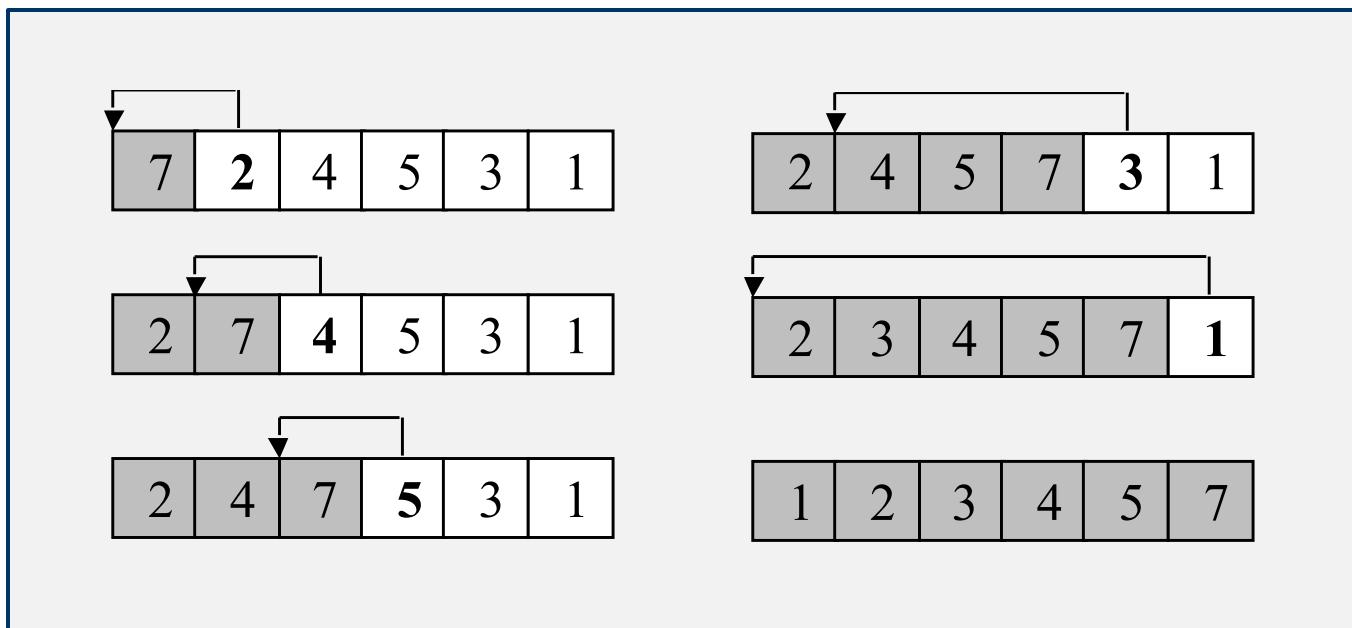
$$\sum_{k=0}^{n-2} (\mathcal{O}(1) + (n - k - 1)) = (\text{ponendo } i = k + 1) =$$

$$\sum_{i=1}^{n-1} (\mathcal{O}(1) + (n - i)) = \mathcal{O}(1)(n - 1) + \sum_{i=1}^{n-1} (n - i) =$$

$$\mathcal{O}(n) + \frac{n(n - 1)}{2} = \mathcal{O}(n) + \mathcal{O}(n^2) = \mathcal{O}(n^2)$$

InsertionSort

Idea: estende l'ordinamento da k a $k+1$ elementi, posizionando l'elemento $(k+1)$ -esimo nella posizione corretta rispetto ai primi k elementi



InsertionSort: pseudocodice

Algoritmo: insertionSort(array S di dimensione n)

```
for(k = 1; k < n; k++) {
    elem = S[k]; //elemento da riposizionare
    p=0; //p: destinazione
    while(p < k && S[p] < elem) {
        p++;
    }
    for(i = k; i > p; i--) {
        S[i] = S[i-1];
    }
    S[p] = elem;
}
```

InsertionSort: analisi

- Il ciclo for esterno viene eseguito $n-1$ volte
- Il k -esimo ciclo while viene eseguito p volte ($p \leq k$)
- Il k -esimo ciclo for interno viene eseguito $k-p$ volte

$$\mathcal{O}\left(\sum_{k=1}^{n-1} p + k - p\right) = \mathcal{O}\left(\sum_{k=1}^{n-1} k\right) = \mathcal{O}(n^2)$$

BubbleSort

Esegue una serie di scansioni dell'array:

- In ogni scansione confronta coppie di elementi adiacenti, scambiandoli se non sono nell'ordine corretto
- Dopo una scansione in cui non viene effettuato nessuno scambio l'array è ordinato
- Dopo la k -esima scansione, i k elementi più grandi sono correttamente ordinati ed occupano le k posizioni più a destra.

BubbleSort: esempio di esecuzione

(1)

7	2	4	5	3	1
---	---	---	---	---	---

2 7

4 7

5 7

3 7

1 7

(2)

2	4	5	3	1	7
---	---	---	---	---	---

2 4

4 5

3 5

1 5

2	4	3	1	5	7
---	---	---	---	---	---

(3)

2	4	3	1	5	7
---	---	---	---	---	---

2 4

3 4

1 4

(4)

2	3	1	4	5	7
---	---	---	---	---	---

2 3

1 3

(5)

2	1	3	4	5	7
---	---	---	---	---	---

1 2

1	2	3	4	5	7
---	---	---	---	---	---

BubbleSort: pseudocodice

```
Algoritmo: bubbleSort(array S di dimensione n)
scambiato = true;
k = 0;
while(k < n-1 && scambiato) {
    scambiato = false;
    for(j = 1; j < n-k; j++) {
        if(S[j-1] > S[j]) then {
            scambia S[j-1] e S[j];
            scambiato = true;
        }
    }
    k++;
}
```

BubbleSort: analisi

- Dopo la k-esima scansione, i k elementi più grandi sono ordinati. Nel caso peggiore: (n-1) iterazioni del while
- La k-esima esecuzione del ciclo for esegue n-k iterazioni

$$\mathcal{O}\left(\sum_{k=1}^{n-1} n - k\right) = \mathcal{O}\left(\sum_{i=1}^{n-1} i\right) = \mathcal{O}(n^2)$$

$O(n^2)$ significa eseguire tutti i confronti possibili!

Esistono soluzioni più efficienti?

HeapSort

- Stesso approccio di **selectionSort**, ma numero di confronti ridotto tramite l'uso di un'opportuna struttura dati **Heap**
- **Heap** associato ad un insieme ordinato S:
 - **Albero binario** in cui:
 1. *Completo* fino al penultimo livello (*completo*: tutte le foglie sono allo stesso livello)
 2. Ciascun nodo **v** contiene un solo elemento di S, indicato con **chiave(v)** e nodi distinti contengono elementi distinti
 3. Per ogni nodo **v** e per ogni suo figlio **w**, **chiave(v) ≥ chiave(w)**
- **NOTA:** Ordineremo a partire dall'elemento più grande

HeapSort: proprietà di ordinamento

Dato un nodo v :

- $\text{sin}(v)$ denota il figlio sinistro di v
- $\text{des}(v)$ denota il figlio destro di nodo v

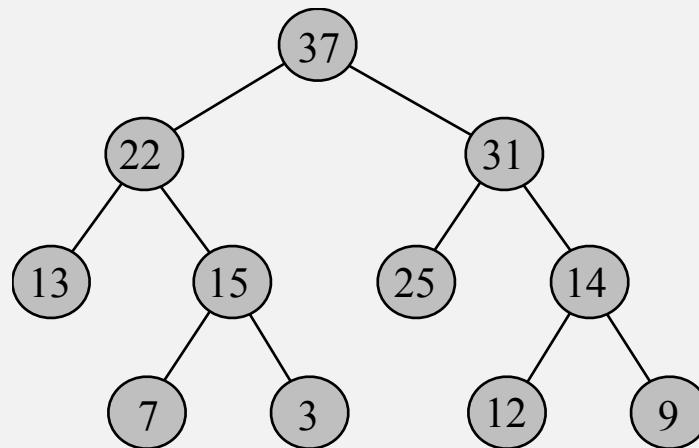
- Proprietà 3):

$$\mathbf{chiave}(v) \geq \mathbf{chiave}(\text{sin}(v))$$

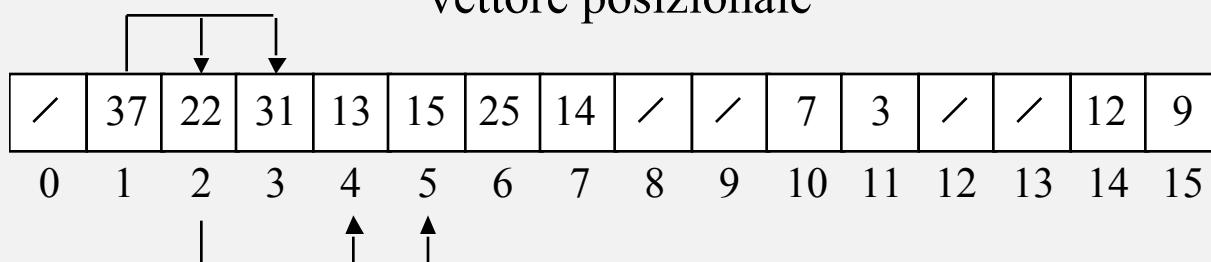
$$\mathbf{chiave}(v) \geq \mathbf{chiave}(\text{des}(v))$$

Heap

Rappresentazione ad albero e con vettore posizionale



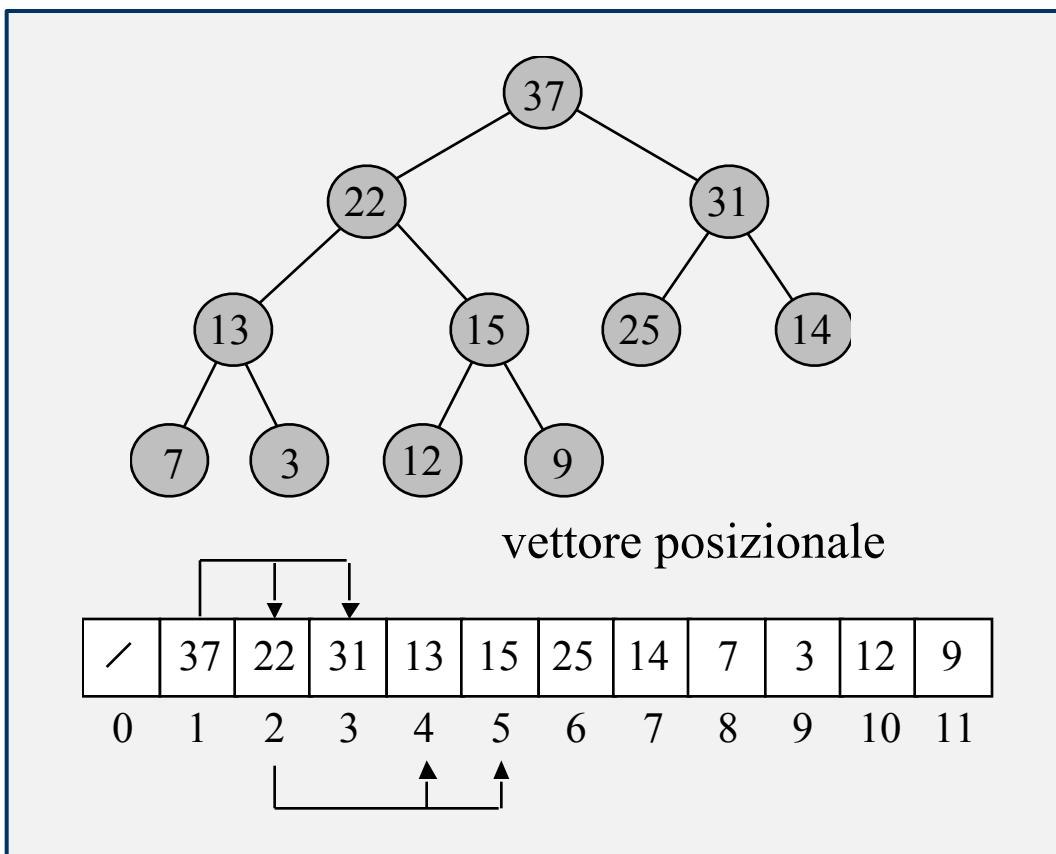
vettore posizionale



$2i \rightarrow \text{sin}(i)$
 $2i+1 \rightarrow \text{des}(i)$

Heap con struttura rafforzata

Le foglie nell'ultimo livello sono compattate a sinistra



Il vettore
posizionale ha
esattamente
dimensione n
(memorizzabile
in loco)

Heap: Proprietà notevoli

- 1) Massimo contenuto nella radice
- 2) Altezza: $O(\log n)$
- 3) Heap con struttura rafforzata rappresentabile in array di dimensione **n** (permette di **non** usare memoria aggiuntiva rispetto all'input — memorizzazione *in loco*)

La procedura fixHeap

Se tutti i nodi di H tranne v soddisfano la proprietà di ordinamento a heap, possiamo ripristinarla come segue:

```
Algoritmo fixHeap(nodo v, heap H)
    if (v è una foglia) then return
    else
        sia u il figlio di v con chiave massima
        if (chiave(v) < chiave(u)) then
            scambia chiave(v) e chiave(u)
            fixHeap(u,H)
```

Tempo di esecuzione: **O(log n)**

Estrazione del massimo

Dato un heap H vogliamo estrarne il massimo valore (contenuto nella radice)

- Come riorganizzare l'albero rimanente preservando le proprietà di Heap?

getMax(heap H) → elem e

1. elem e \leftarrow chiave(radice di H);
2. copia nella radice la chiave contenuta nella foglia più a destra dell'*ultimo livello*;
3. rimuovi la foglia;
4. ripristina la proprietà di ordinamento dell'heap richiamando **fixHeap** sulla radice;
5. **return e**;

(La scelta della foglia preserva completezza e rafforzamento)

Tempo di esecuzione: **O(log n)** (profondità di H)

Costruzione dell'heap

Costruiamo un albero binario con gli elementi di S e poi lo ordiniamo usando il seguente algoritmo (*divide et impera*)

```
Algoritmo heapify(albero H)
    if (H è vuoto) then return
    else
        heapify(sottoalbero sinistro di H)
        heapify(sottoalbero destro di H)
        fixHeap(radice di H, H)
```

Tempo di esecuzione: $T(n) = 2T(n/2) + O(\log n)$

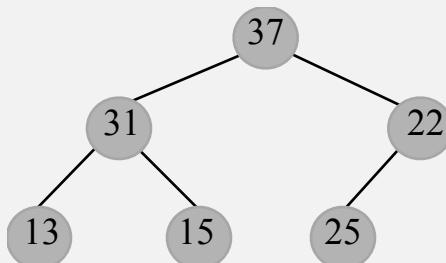
→ $T(n) = O(n)$ dal *Teorema Master (caso 1)*

L'algoritmo HeapSort

```
Algoritmo heapSort(array S di dimensione n)
if (S è vuoto) then return S;
else {
    - costruisci un albero binario H con gli elementi di S;
    - ordinalo usando heapyfy(H); //costa O(n)
    - crea un coda C vuota
    - while(H non è vuoto){           //cicla n volte
        C.enqueue(getMax(H));         //getMax ha costo O(log n)
    - copia gli elementi di C in S nel giusto ordine
    - return S
}
```

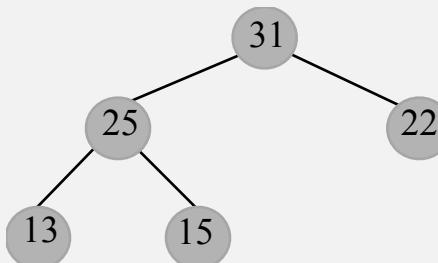
Ordina in tempo **O(n log n)**

HeapSort: esempio di esecuzione



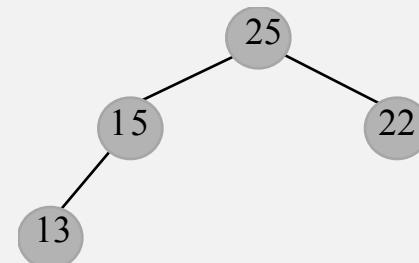
37	31	22	13	15	25
----	----	----	----	----	----

(1)



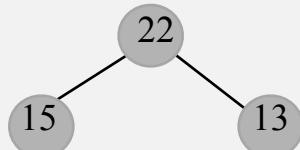
31	25	22	13	15	37
----	----	----	----	----	----

(2)



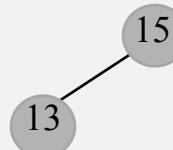
25	15	22	13	31	37
----	----	----	----	----	----

(3)



22	15	13	25	31	37
----	----	----	----	----	----

(4)



15	13	22	25	31	37
----	----	----	----	----	----

(5)



13	15	22	25	31	37
----	----	----	----	----	----

(6)

MergeSort (1/2)

- Approccio *Divide et Impera*
- Passo ***Impera***: dati due array **ordinati** A e B fondiamoli in un nuovo array **ordinato**:

merge(array ordinato A, array ordinato B) → array ordinato X

1. Sia X un array vuoto di dimensione $|A| + |B|$;
2. estrai ripetutamente il minimo tra gli elementi di A e B e copialo in X, finché A o B non sono vuoti;
3. copia i restanti elementi dell'array non vuoto alla fine di X;
4. Restituisci X;

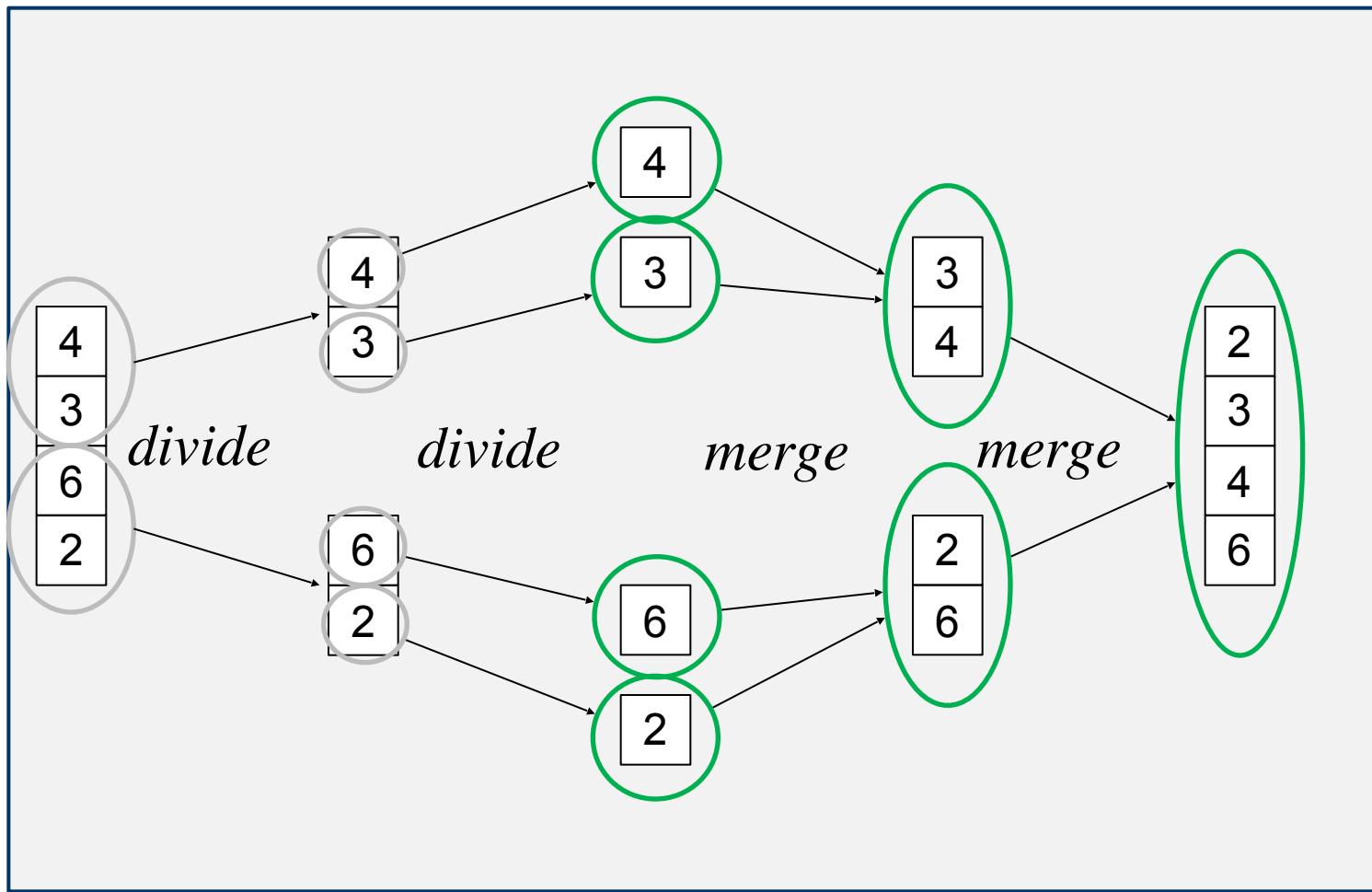
La procedura **merge(A,B)** esegue al più $|A| + |B| - 1$ confronti

MergeSort (2/2)

Passo **Divide**: dividiamo l'array in due e ordiniamo ricorsivamente gli array risultanti (poi li fondiamo)

```
Algoritmo mergeSort(array S) → array
    if (|S| <= 1) then return S;
    S1 ← mergeSort(metà sinistra di S)
    S2 ← mergeSort(metà destra di S)
    return merge(S1, S2)
}
```

MergeSort: esempio di esecuzione



Tempo di esecuzione

- Il numero di confronti del MergeSort è descritto dalla seguente relazione di ricorrenza:

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

- Usando il *Teorema Master (caso 2)* si ottiene:

$$T(n) = \mathbf{O(n \log n)}$$

(stesso costo dell'HeapSort!)

QuickSort

- *Divide et impera*
- **Divide:**
 - scegli un elemento x di A (detto *perno* o *pivot*)
 - partiziona gli elementi di A in due sotto-array:
 - uno con gli elementi di A più piccoli o uguali ad x
 - uno con gli elementi di A strettamente maggiori di x
 - Risolvi i due sottoproblemi ricorsivamente
- **Impera:**
 - restituisci la concatenazione dei due sotto-array ordinati

QuickSort: pseudocodice

```
Algoritmo quickSort(array A)
scegli un elemento x di A;
Partiziona A rispetto ad x calcolando i due array:
    A1 = { y ∈ A : y ≤ x }
    A2 = { y ∈ A : y > x }
if (|A1| > 1) then quickSort(A1);
if (|A2| > 1) then quickSort(A2);
copia la concatenazione di A1 e A2 in A
```

Calcolare le due partizioni richiede **O(n)** confronti

QuickSort: analisi del caso peggiore

- Nel caso peggiore, il perno scelto ad ogni passo è il minimo o il massimo degli elementi nell'array
- Costo:
 - $n-1$ confronti per la partizione
 - ordinamento dell'array con $n>1$ elementi: $C(n-1)$
- Quindi: $C(n) = n-1 + C(n-1)$
- Svolgendo per iterazione: $C(n) = O(n^2)$

QuickSort: randomizzazione

- Possiamo evitare il caso peggiore scegliendo come perno un elemento dell'array A "a caso".
- Poiché ogni elemento ha la stessa probabilità, pari a $1/n$, di essere scelto come perno, il numero di confronti nel caso atteso è:

$$C(n) = \sum_{a=0}^{n-1} \frac{1}{n} \left[n-1 + C(a) + C(n-a-1) \right] = n-1 + \sum_{a=0}^{n-1} \frac{2}{n} C(a)$$

dove a e $(n-a-1)$ sono le dimensioni dei sottoproblemi risolti ricorsivamente

QuickSort: analisi nel caso medio

La relazione di ricorrenza: $C(n) = n - 1 + \sum_{a=0}^{n-1} \frac{2}{n} C(a)$

ha soluzione $C(n) \leq 2 n \log n$, pertanto il numero di confronti atteso è **$O(n \log n)$**

Quindi il *quickSort* è meno efficiente nel caso peggiore del *mergeSort* e del *heapSort* e per evitare il caso peggiore lo rendiamo "*randomizzato*".

Il *quickSort* è però molto utilizzato in quanto esistono sue varianti ottimizzate estremamente veloci.

Utilizzando algoritmi come il *mergeSort* e l'*heapSort* possiamo ordinare un insieme di elementi con un numero di confronti pari a **O(n log n)**.

Possiamo fare di meglio?

Lower bound

- Dimostrazione (matematica) che non possiamo andare più veloci di una certa quantità.
- Più precisamente, ogni algoritmo all'interno di un certo **modello di calcolo** deve avere tempo di esecuzione almeno pari a quella quantità.
- Non significa necessariamente che non è possibile fare di meglio, infatti potrebbe essere possibile fare di meglio al di fuori di quel modello di calcolo.

Lower bound per l'ordinamento

- Delimitazione inferiore: quantità di risorsa necessaria per risolvere un determinato problema.
- Dimostrare che $\Omega(n \log n)$ è **lower bound** al numero di confronti richiesti per ordinare n oggetti.

Come?

Dobbiamo dimostrare che tutti gli algoritmi richiedono almeno $\Omega(n \log n)$ confronti!

Modello basato sui confronti

- In questo modello, per ordinare è possibile usare solo confronti tra oggetti
- Primitive quali operazioni aritmetiche (somme o prodotti), logiche (and e or), o altro (shift) sono proibite.
- Sufficientemente generale per catturare le proprietà degli algoritmi più noti

Come dimostrare il lower bound

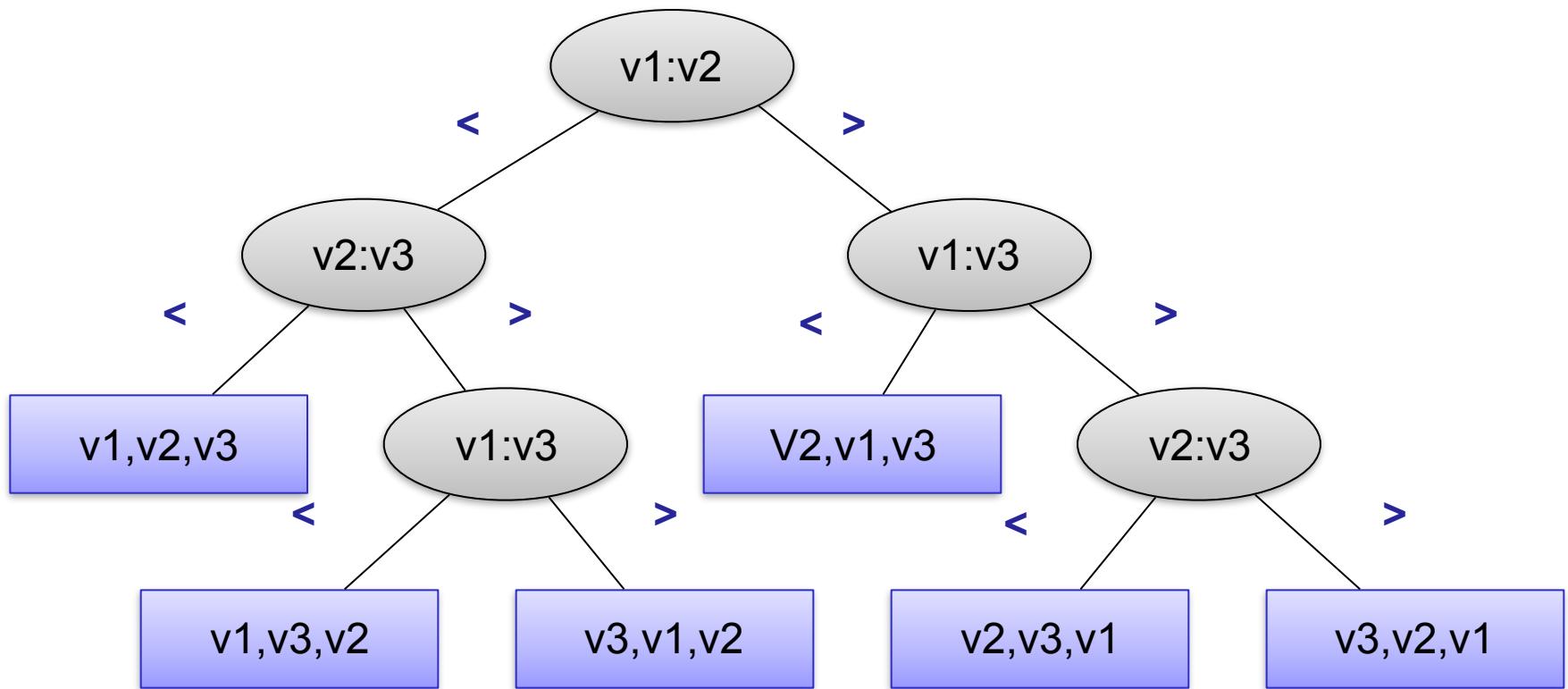
- Consideriamo un qualsiasi algoritmo **A**, che ordina solo mediante confronti.
- Non abbiamo assolutamente idea di come funziona **A**!
- Ciò nonostante, dimostreremo che **A** deve eseguire almeno $\Omega(n \log n)$ confronti per ordinare un insieme di elementi con cardinalità **n**.
- Dato che l'unica ipotesi su **A** è che ordini mediante confronti, il **lower bound** sarà valido per tutti gli algoritmi basati su confronti!

Come dimostrare il lower bound

- Osservazione fondamentale: tutti gli algoritmi devono confrontare elementi
- Dati due valori v_1 e v_2 , i casi possibili sono tre:
 $v_1 < v_2$, $v_2 > v_1$, oppure $v_1 = v_2$.
- Per semplicità assumeremo che tutti gli elementi sono tra loro diversi
- Si assume dunque che tutti i confronti abbiano la forma $v_1 < v_2$, e il risultato del confronto sia *vero* o *falso*

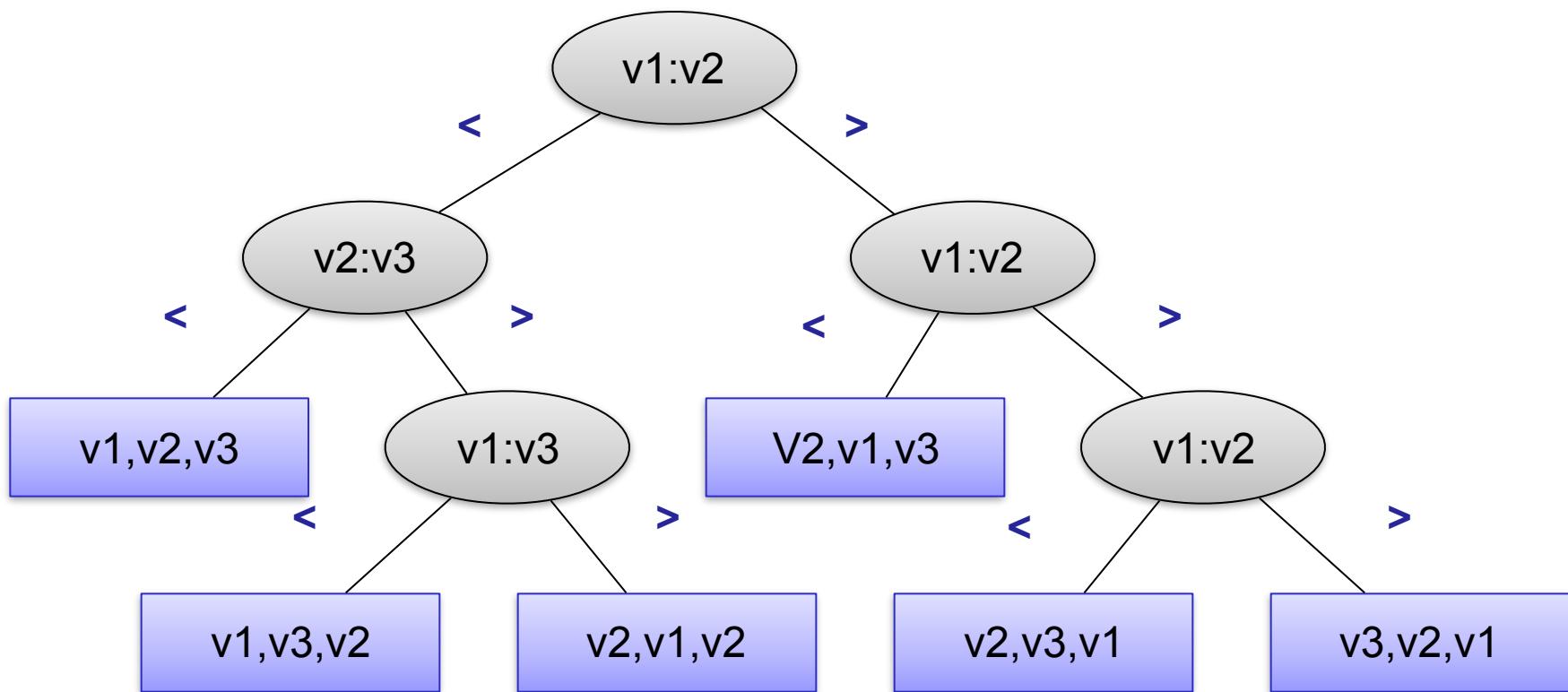
Alberi di decisione (1/3)

- Un albero di decisione rappresenta i confronti eseguiti da un algoritmo su un dato input
- Il seguente esempio indica l'albero di decisione sull'insieme $\{v1, v2, v3\}$



Alberi di decisione (2/3)

- Dato un insieme di **n** valori, sono possibili **n!** permutazioni (possibili ordinamenti) → l'albero di decisione deve quindi avere **n!** foglie.
- L'esecuzione di un algoritmo **A** corrisponde ad un cammino sull'albero di decisione corrispondente all'input dato.



Alberi di decisione (2/4)

- Riassumendo:
 - Albero binario;
 - Deve contenere $n!$ foglie
 - Il cammino da una radice ad una foglia rappresenta l'esecuzione dell'algoritmo su un'istanza.
- Il più **lungo** cammino dalla radice ad una foglia rappresenta il numero di confronti che l'algoritmo deve eseguire nel **caso peggiore**.

Il lower bound $\Omega(n \log n)$

- **Teorema:** qualunque albero di decisione che ordina n elementi ha altezza $\Omega(n \log n)$
- **Corollario:** nessun algoritmo di ordinamento basato sui confronti ha complessità migliore di $\Omega(n \log n)$

Dimostrazione del teorema

- Un albero di decisione è binario;
- Un albero binario di altezza h non ha più di 2^{h-1} foglie
- Il nostro albero deve avere almeno $n!$ foglie, quindi:
 - $2^{h-1} \geq n! \rightarrow h-1 \geq \log(n!)$
 - $n! \geq (n/e)^n$ (approssimazione di Stirling)
 - $h-1 \geq \log(n!) \geq \log(n/e)^n = n \log(n) - n \log(e) = \Omega(n \log n)$

Quindi?

- Un importante corollario del teorema che fissa $\Omega(n \log n)$ come lower bound per il problema dell'ordinamento nel modello basato sui confronti è che gli algoritmi **heapSort** e **mergeSort** sono algoritmi **ottimi** (con complessità asintotica ottima).

Ordinamenti lineari

(per dati di input con proprietà particolari)

Ordinamento di n interi in $[1,n]$

Abbiano n interi distinti con valore compreso nell'intervallo $[1,n]$

In quanto tempo possiamo ordinarli?

- $O(1) \rightarrow$ l'unico ordinamento possibile è $\{1,2,3,4,5,6,\dots,n\}$
- Contraddice il lower bound?
- No! Perché?

Ordinamento di n interi in $[1,n]$

Abbiano n interi distinti con valore compreso nell'intervallo $[1,n]$

In quanto tempo possiamo ordinarli?

- $O(1) \rightarrow$ l'unico ordinamento possibile è $\{1,2,3,4,5,6,\dots,n\}$
- Contraddice il lower bound?
- No! Perché? \rightarrow **Siamo fuori dal modello basato sui confronti**

IntegerSort: fase 1

Nuovo problema, meno banale:

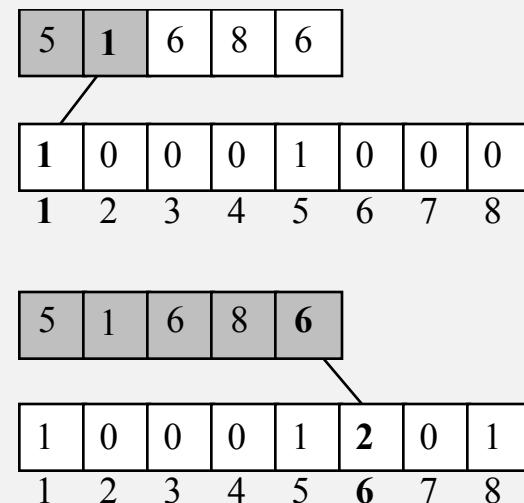
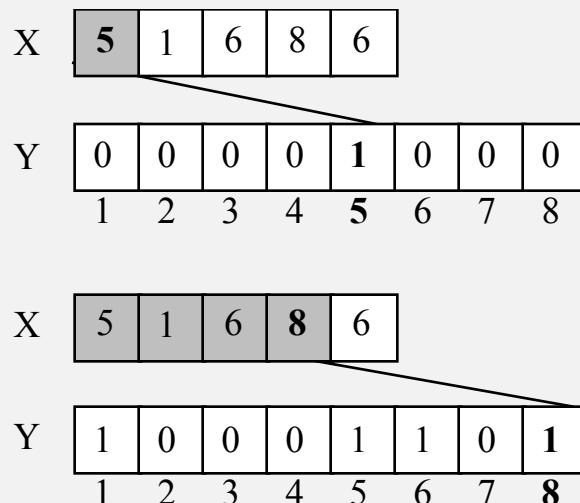
"ordinare n interi con valori in [1,k]"

Strategia: mantiene un array Y di k contatori tale che
 $Y[x] = \text{numero di volte che il valore } x \text{ compare nell'array}$
di input X

IntegerSort: fase 1

- ESEMPIO:

array X di 5 interi compresi tra 1 e 8



(a) Calcolo di Y

IntegerSort: fase 2

Dopo aver contato quanto volte si ripete ogni elemento in X, scorriamo Y da sinistra verso destra e, se $Y[x]=k$, scriviamo in X il valore x per k volte

IntegerSort: pseudocodice

```
Algoritmo integerSort(intero k, array X di dimensione n)
Sia Y un array di dimensione k;
//inizializzo tutte le celle di Y a 0;
for(int i < 0; i < k; i++) { Y[i] < 0; }
//scandisco X ed ogni volta che X contiene un valore p //
incremento di uno il valore in Y[p]
for(int h < 0; h < n; h++) { Y[X[h]] < Y[X[h]] + 1; }
j < 1;
//scandisco Y, se Y[t]=k con k > 0, scrivo in X
//il valore t per k volte
for( t < 0; t < k; t++ ) {
    while( Y[t] > 0 ){
        X[j] < t; j++; Y[t] < Y[t]-1;
    }
}
```

IntegerSort: analisi

L'algoritmo IntegerSort richiede:

- Tempo **O(k)** per inizializzare **Y** (creo **Y** e inizializzo ogni sua cella a 0)
- Tempo **O(n)** per scandire **X** (contare i suoi valori e incrementare i contatori in **Y**)
- Tempo **O(n+k)** per ricostruire **X** (scandire **Y** e riscrivere gli **n** valori in **X**)

Quindi *integerSort* richiede un tempo pari a **O(n+k)**. Questo significa che, finché **k** è proporzionale con **n** ($k = O(n)$), il costo di **integerSort** è lineare con **n** (**O(n)**). Questa proprietà di perde se si possono avere valori di **k** molto grandi.

E.s. per $k = n^5$ il costo di *integerSort* sarà **O(n + n⁵) = O(n⁵)**.

BucketSort

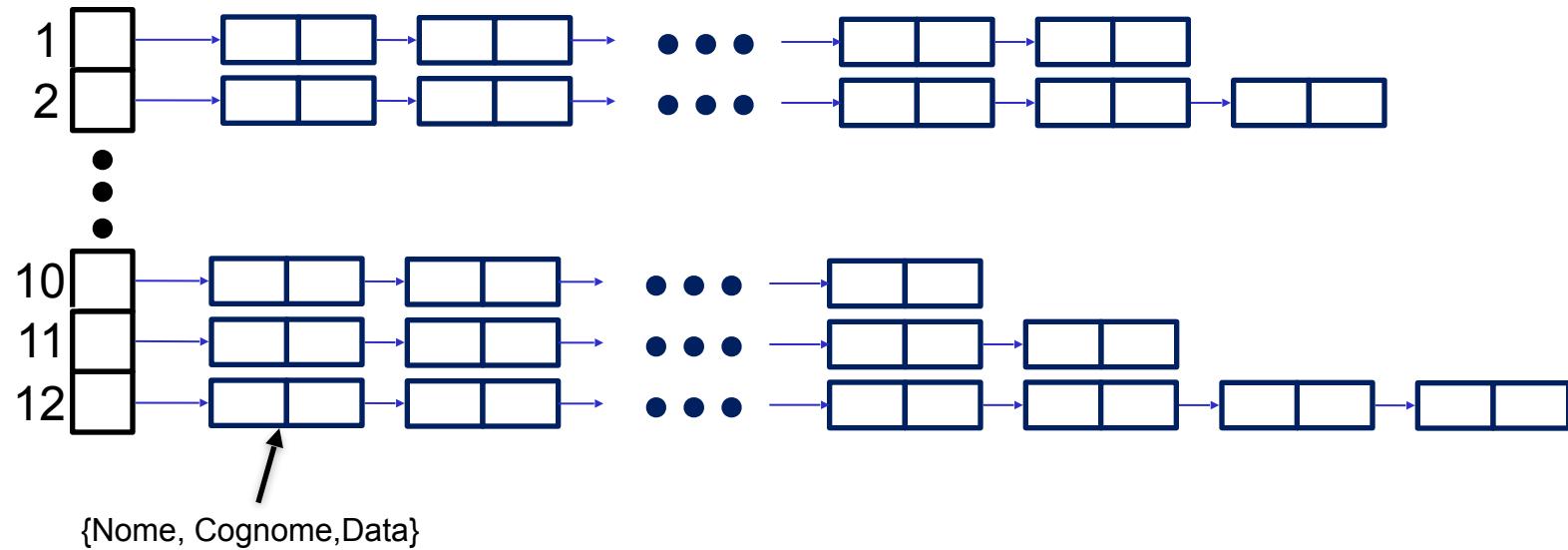
E se invece di interi vogliamo ordinare altri tipi di elementi (esempio le persone in una lista in base al loro mese di nascita)?

Per ordinare **n** record con chiavi intere in **[1,k]**

- Basta mantenere un **array di liste**, anziché di contatori, ed operare come per IntegerSort
- La lista **Y[x]** conterrà gli elementi con chiave uguale a **x**
- Per ottenere una lista ordinata vado poi a concatenare le liste in **Y**
- Tempo **O(n+k)** come per IntegerSort

BucketSort: esempio

- Ho una lista di 1000 record contenenti le seguenti informazioni sulle persone:
 - Nome; Cognome; Data di nascita (gg/mm/aaaa)
- Ordinare i record sulla base del mese di nascita delle persone
- Ci sono solo 12 mesi quindi userò un array di liste con 12 caselle



Stabilità

- Un algoritmo è stabile se preserva l'ordine iniziale tra elementi con la stessa chiave
- Il *BucketSort* può essere reso stabile appendendo gli elementi di X in coda alla opportuna lista Y[i]

RadixSort

Come abbiamo visto l'*BucketSort* funziona bene quando k è dello stesso ordine di n . Che fare quando k è molto grande?

- Rappresentiamo gli elementi in base b (binario, decimale, esadecimale, ecc.), ed eseguiamo una serie di *BucketSort*
- Partiamo ordinando i numeri usando come chiave la cifra meno significativa e poi ci spostiamo verso quella più significativa.



RadixSort: correttezza

- Se x e y hanno una diversa t -esima cifra, la t -esima passata di *BucketSort* li ordina
- Se x e y hanno la stessa t -esima cifra, la proprietà di stabilità del *BucketSort* li mantiene ordinati correttamente



Dopo la t -esima passata di *BucketSort*, i numeri sono correttamente ordinati rispetto alle t cifre meno significative

RadixSort: tempo di esecuzione

Se usiamo come base per il *bucketSort* un valore **b = Θ(n)**, l'algoritmo *radixSort* ordina **n** numeri interi compresi tra [1,k], in tempo:

$$T(n) = O(n \left(\frac{\log(k)}{\log(n)} \right))$$

Algoritmi e Strutture Dati

Alberi di ricerca

Fabio Patrizi

Dizionari

Gli **alberi di ricerca** sono usati per realizzare in modo efficiente il **tipo di dato Dizionario**

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($elem e$)

cancella da S l'elemento e

search($chiave k \rightarrow elem$)

se la chiave k è presente in S restituisce un elemento e ad essa associato,
e **null** altrimenti

Alberi binari di ricerca

(BST = binary search tree)

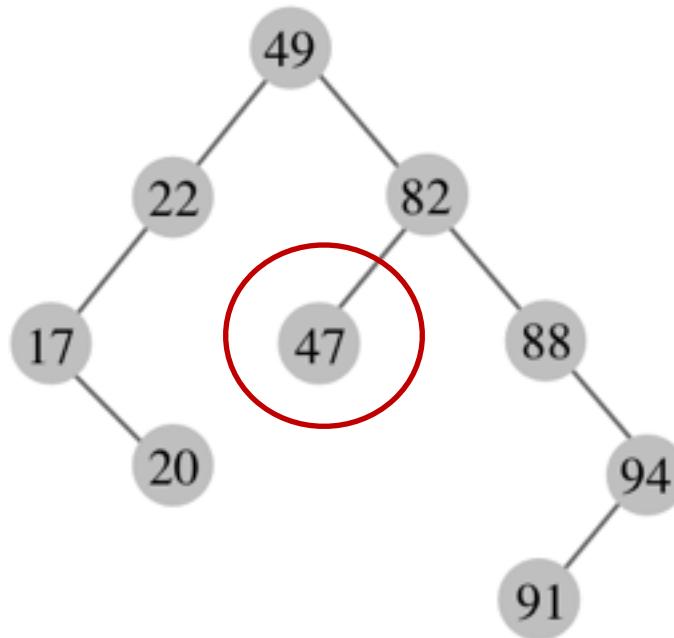
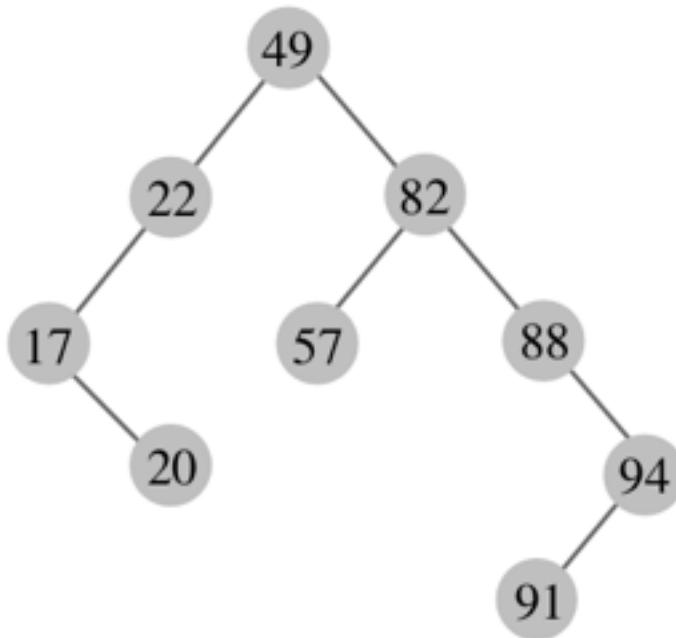
Definizione

Un BST è albero binario che soddisfa le seguenti proprietà:

1. ogni nodo v contiene un elemento $\text{elem}(v)$ cui è associata una chiave $\text{chiave}(v)$ presa da un dominio **totalmente ordinato**
2. le **chiavi** nel sottoalbero sinistro di v sono \leq $\text{chiave}(v)$
3. le **chiavi** nel sottoalbero destro di v sono \geq $\text{chiave}(v)$

Le proprietà 2 e 3 sono dette "**proprietà di ricerca**".

Esempi



Albero binario di ricerca

Albero binario non di ricerca

Search

Idea: Segui un cammino nell'albero partendo dalla radice: su ogni nodo sfruttiamo le proprietà di ricerca per decidere se proseguire nel sottoalbero sinistro o destro

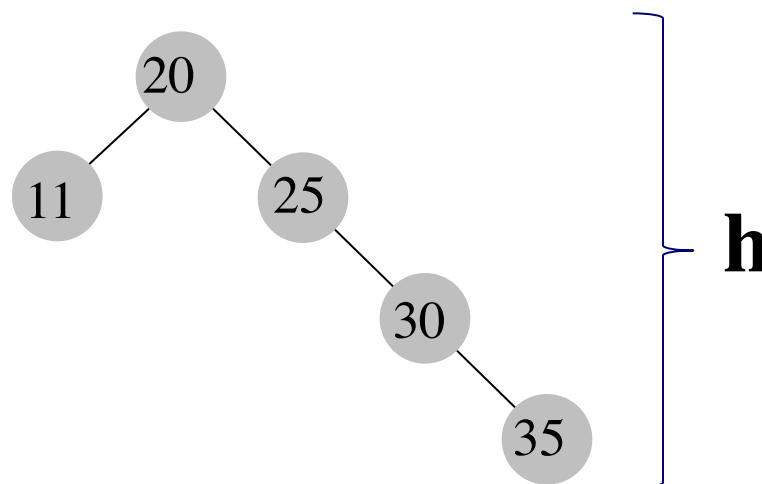
algoritmo **search**(*chiave k*) \rightarrow *elem*

1. $v \leftarrow$ radice di T
2. **while** ($v \neq \text{null}$) **do**
 3. **if** ($k = \text{chiave}(v)$) **then return** *elem(v)*
 4. **else if** ($k < \text{chiave}(v)$) **then** $v \leftarrow$ figlio sinistro di v
 5. **else** $v \leftarrow$ figlio destro di v
6. **return** **null**

Search: analisi

L'approccio adottato nella funzione *search* ricorda molto da vicino quello della ricerca binaria. Il suo costo però è **O(h)**, dove **h** è l'altezza dell'albero, e non **O(log n)**.

Questa differenza è dovuta al fatto che un BST potrebbe non essere bilanciato.



Insert

Un nuovo nodo viene sempre aggiunto come foglia.
L'inserimento può quindi essere implementato come segue:

- 1.crea un nuovo nodo **u** con **elem(u)=e** e **chiave(u)=k**
- 2.cerca la chiave **k** nell'albero, identificando così il nodo **v** che diventerà padre di **u**
- 3.appendi **u** come figlio sinistro/destro di **v** in modo che siano mantenute le proprietà di ricerca

Insert: analisi

- I passi 1. e 3. richiedono un numero di passi costante e posso quindi essere eseguiti in tempo **O(1)**.
- Il passo 2. richiede di effettuare una ricerca nell'albero che, come abbiamo visto, richiede tempo **O(h)**.
- In totale, anche il costo per l'inserimento è quindi **O(h)**

Delete

- Come avviene in molte strutture dati, cancellare è più difficile che inserire.
- Per implementare l'operazione di *delete* introduciamo due nuove operazioni:
 - $\max(\text{nodo } u) \rightarrow \text{nodo}$
 - $\text{pred}(\text{nodo } u) \rightarrow \text{nodo}$

- Questa operazione permette di individuare il valore massimo in un albero radicato in **u**.
- Grazie alle proprietà di ricerca, per trovare il massimo è sufficiente, partendo da **u**, scendere verso destra nell'albero finché è possibile.
- Il tempo di esecuzione è chiaramente **O(h)**.

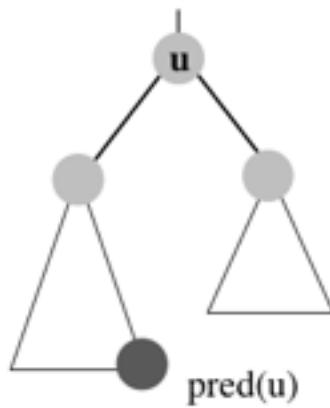
algoritmo **max**(*nodo u*) \rightarrow *nodo*

1. $v \leftarrow u$
2. **while** (figlio destro di $v \neq \text{null}$) **do**
3. $v \leftarrow$ figlio destro di v
4. **return** v

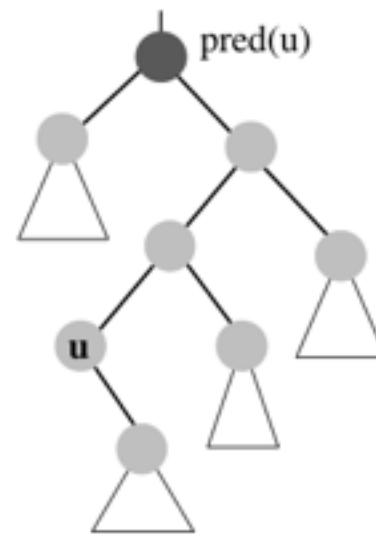
Ricerca del predecessore (1/3)

- Il predecessore di un nodo u è il nodo v dell'albero avente **massima chiave minore o uguale** alla chiave di u .
- Per trovare il predecessore di u possiamo distinguere due casi:
 1. u ha un figlio sinistro: in tal caso $\text{pred}(u)$ è il massimo del sottoalbero sinistro di u .
 2. u non ha un figlio sinistro: $\text{pred}(u)$, se esiste, è il più basso antenato di u (ovvero, l'antenato di u con massima profondità nell'albero) il cui figlio destro è anch'esso antenato di u . Per trovarlo, risaliamo da u verso la radice fino ad incontrare la prima "svolta a sinistra"

Ricerca del predecessore (2/3)



Caso 1: u ha un figlio
sinistro



Caso 2: u non ha un figlio
sinistro

Ricerca del predecessore (3/3)

algoritmo $\text{pred}(\text{nodo } u) \rightarrow \text{nodo}$

1. **if** (u ha figlio sinistro $\text{sin}(u)$) **then**
2. **return** $\text{max}(\text{sin}(u))$
3. **while** ($\text{parent}(u) \neq \text{null}$ e u è figlio sinistro di suo padre) **do**
4. $u \leftarrow \text{parent}(u)$
5. **return** $\text{parent}(u)$

Anche la ricerca del predecessore di un nodo **u** ha costo **O(h)**, dove **h** è l'altezza dell'albero

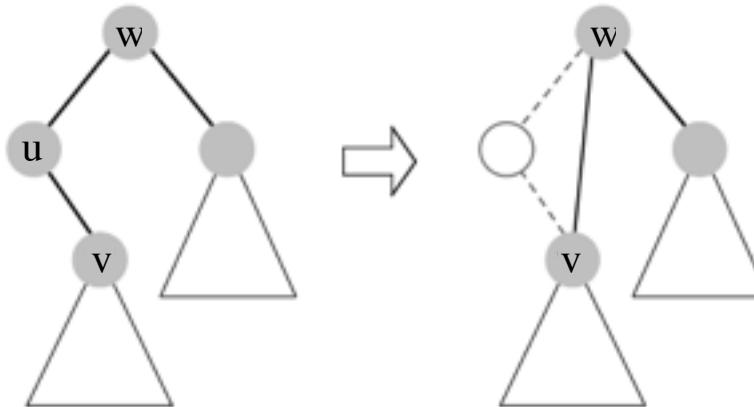
Delete – caso 1

Sia **u** il nodo contenente l'elemento **e** da cancellare.
Distinguiamo tre casi:

Caso 1: **u** è una foglia: in tal caso è sufficiente
eliminarla

Delete – caso 2

Caso 2: **u** ha un unico figlio: sia **v** l'unico figlio di **u**
Se **u** è la radice dell'albero, **v** diviene la
nuova radice. Altrimenti, dopo aver
individuato il genitore **w** di **u**, l'arco **(w, u)**,
viene sostituito dall'arco **(w, v)**



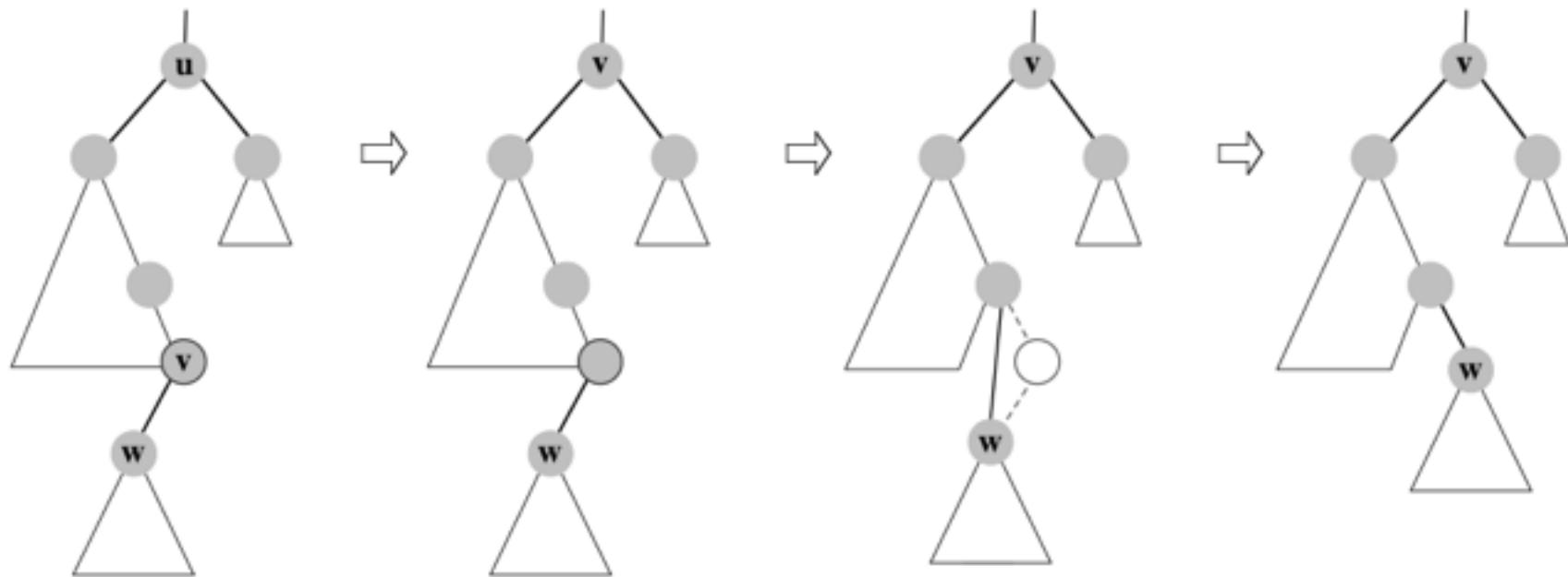
Delete – caso 3

Caso 3: u ha due figli: ci riconduciamo ad uno dei casi precedenti nel seguente modo.

Individuiamo il *predecessore* v di u . Tale predecessore è sicuramente il massimo del sottoalbero sinistro di u poiché u ha due figli. Copiamo v in u ed eliminiamo il vecchio nodo v applicando uno dei casi precedenti.

Delete – caso 3

Caso 3: u ha due figli: lo sostituisco con il predecessore (v) e rimuovo fisicamente il predecessore (che ha un solo figlio)



Delete: analisi

- Si può verificare che le proprietà di ricerca sono mantenute.
- La cancellazione di un nodo interno richiede l'individuazione del nodo da cancellare nonché del suo predecessore (caso 3). Quindi il costo della cancellazione è **O(h)**

BST – riassumendo

- Tutte le operazioni hanno costo $O(h)$ dove h è l'altezza dell'albero
- $O(n)$ nel caso peggiore (alberi molto sbilanciati e profondi)

Alberi AVL

(Adel'son-Vel'skii e Landis)

Definizioni

Fattore di bilanciamento di un nodo v =
| altezza del sottoalbero sinistro di v –
altezza del sottoalbero destro di v |

Un albero si dice **bilanciato in altezza** se ogni nodo v ha fattore di bilanciamento ≤ 1

Alberi AVL = alberi binari di ricerca bilanciati in altezza

Altezza di alberi AVL

Si può dimostrare che un albero AVL con n nodi ha altezza **$O(\log n)$**

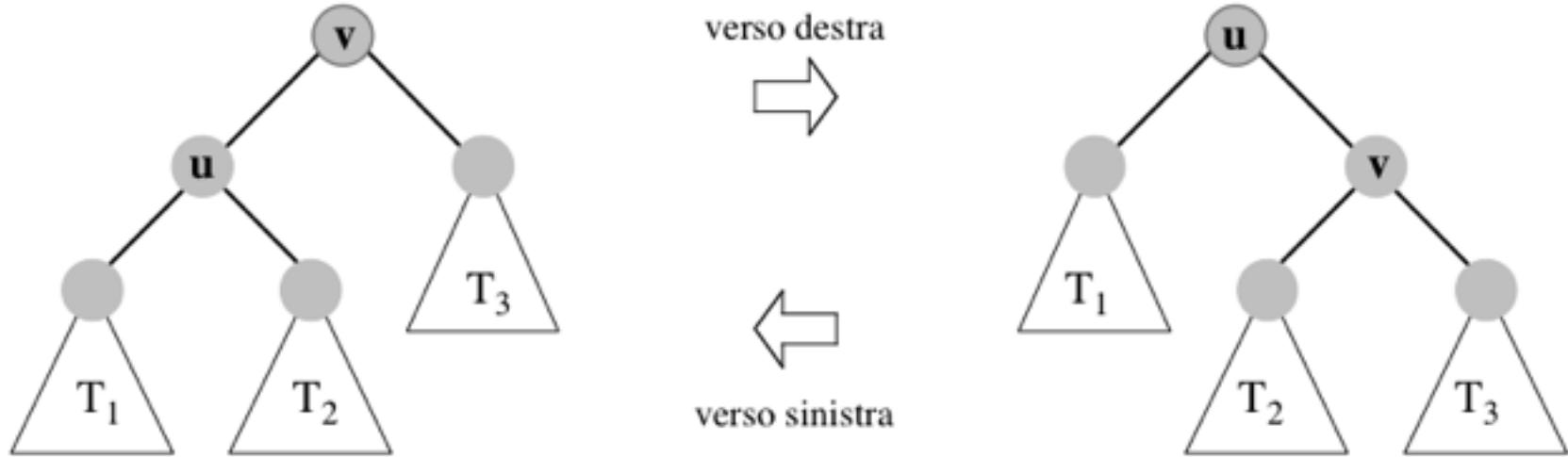
Segue che, utilizzando l'algoritmo *search* introdotto per i BST, possiamo implementare la ricerca in un AVL con costo **$O(\log n)$**

Implementazione delle operazioni

- L'operazione *search* procede come in un BST
- Ma inserimenti e cancellazioni potrebbero sbilanciare l'albero
 - Dobbiamo mantenere il bilanciamento
 (utilizziamo le **rotazioni**...)

La rotazione di base

- Nella *rotazione di base* (anche detta *rotazione semplice*), un nodo **perno** viene fatto ruotare verso **destra** o verso **sinistra** (i due casi sono simmetrici).
- La rotazione mantiene la proprietà di ricerca
- Richiede tempo **O(1)**



Ribilanciamento tramite rotazioni

- Le rotazioni sono effettuate su nodi sbilanciati
- Sia v un nodo con fattore di bilanciamento ≥ 2
- Esiste almeno un sottoalbero T di v che lo sbilancia
- A seconda della posizione di T si hanno 4 casi:

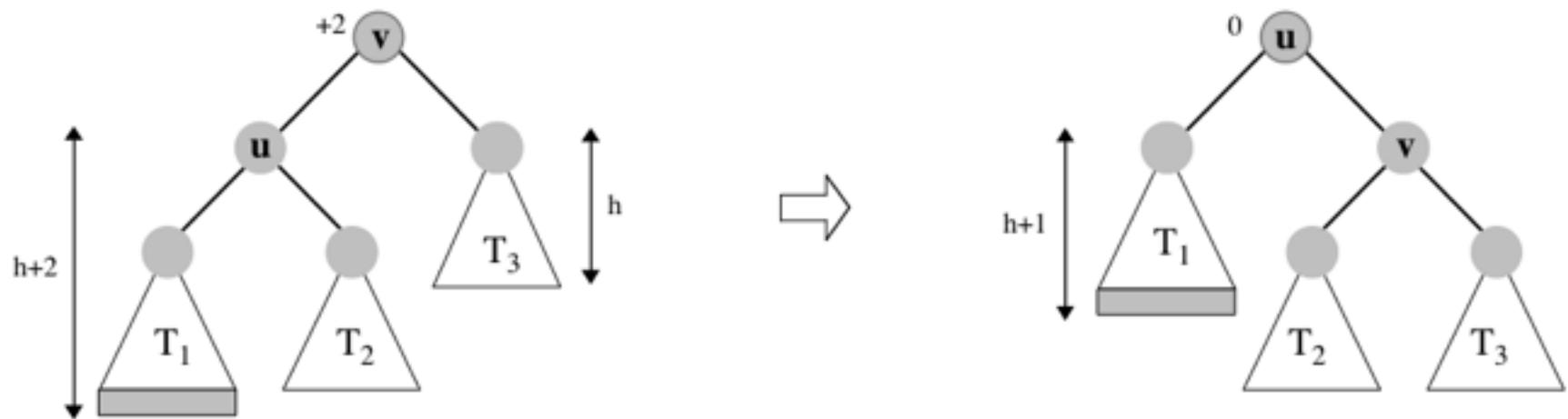
Se esiste un solo sottoalbero T che causa lo sbilanciamento:

Sinistra - sinistra	(SS)	T è il sottoalbero sinistro del figlio sinistro di v
Destra - destra	(DD)	T è il sottoalbero destro del figlio destro di v
Sinistra - destra	(SD)	T è il sottoalbero destro del figlio sinistro di v
Destra - sinistra	(DS)	T è il sottoalbero sinistro del figlio destro di v

- I quattro casi sono simmetrici a coppie (SS con DD e SD con DS).

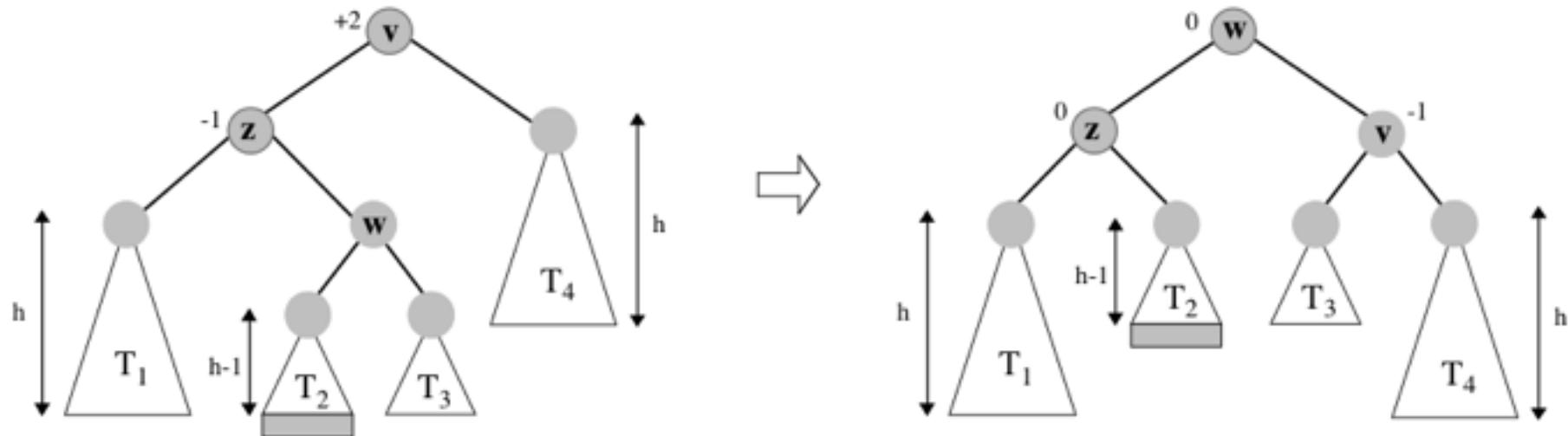
Rotazione SS

- Applicare una *rotazione semplice* verso **destra** su **v**
- L'altezza dell'albero coinvolto nella rotazione passa da **h+2** a **h+1**



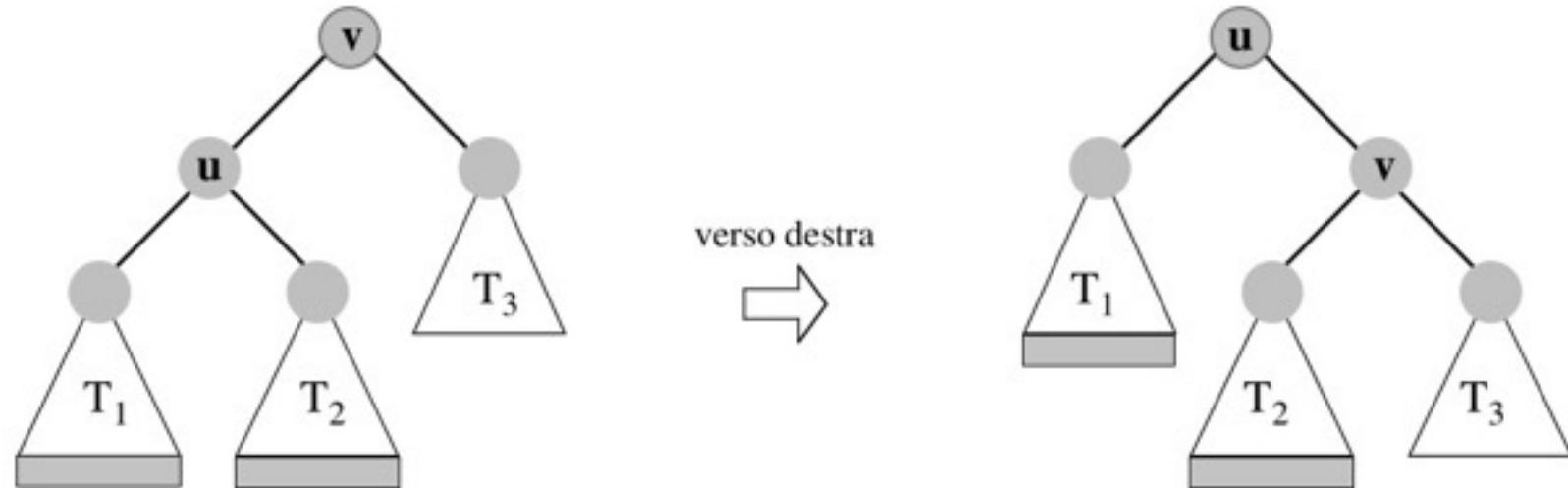
Rotazione SD

- Richiede di applicare due rotazioni. Sia **z** il figlio sinistro di **v**: l'albero che sbilancia **v** è il sottoalbero destro di **z**. Effettuiamo una prima rotazione semplice di **z** verso **sinistra**, seguita da una rotazione semplice di **v** verso **destra**.



Più sottoalberi che sbilanciano

- Nel caso in cui il sottoalbero che crea lo sbilanciamento non è unico (T_1 e T_2 nella figura) è sufficiente effettuare una rotazione nel verso opportuno.



insert(elem e, chiave k)

1. Crea un nuovo nodo **u** con:

$$\text{elem}(u) = e$$

$$\text{chiave}(u) = k$$

2. Inserisci **u** come in un BST (quindi come foglia)
3. Ricalcola i fattori di bilanciamento dei nodi nel cammino dalla radice a **u**: sia **v** il più profondo nodo con fattore di bilanciamento pari a ± 2 (nodo critico)
4. Esegui una rotazione opportuna su **v**

Osservazione: una sola rotazione è sufficiente, poiché l'altezza dell'albero coinvolto diminuisce di 1

delete(elem e)

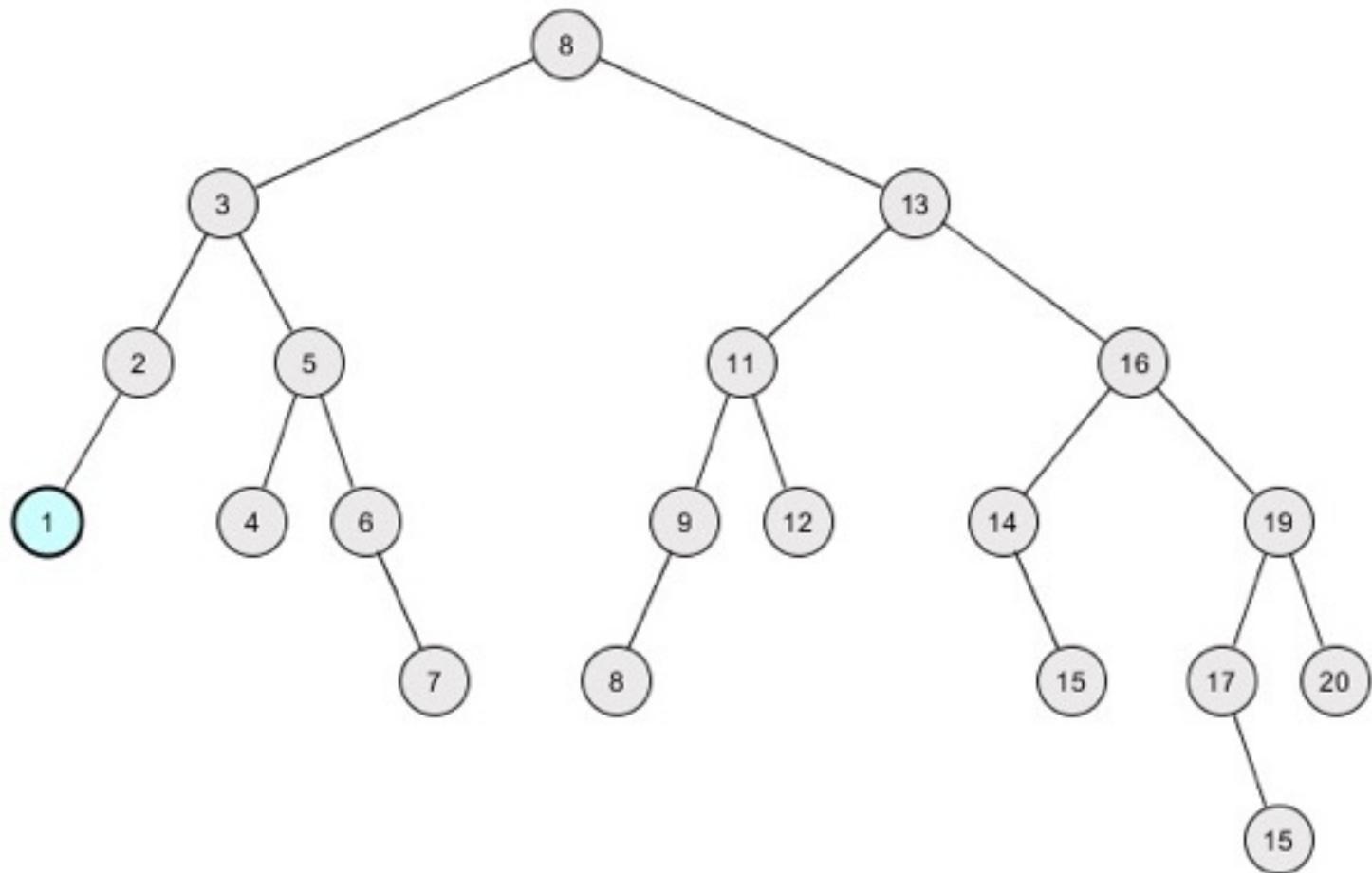
1. Cancella il nodo come in un BST
2. Ricalcola i fattori di bilanciamento dei nodi nel cammino dalla radice al padre del nodo eliminato fisicamente
3. Ripercorrendo il cammino dal basso verso l'alto, esegui l'opportuna rotazione semplice o doppia sui nodi sbilanciati

Osservazione: potrebbero essere necessarie **$O(\log n)$** rotazioni

Cancellazione – esempio (1/7)

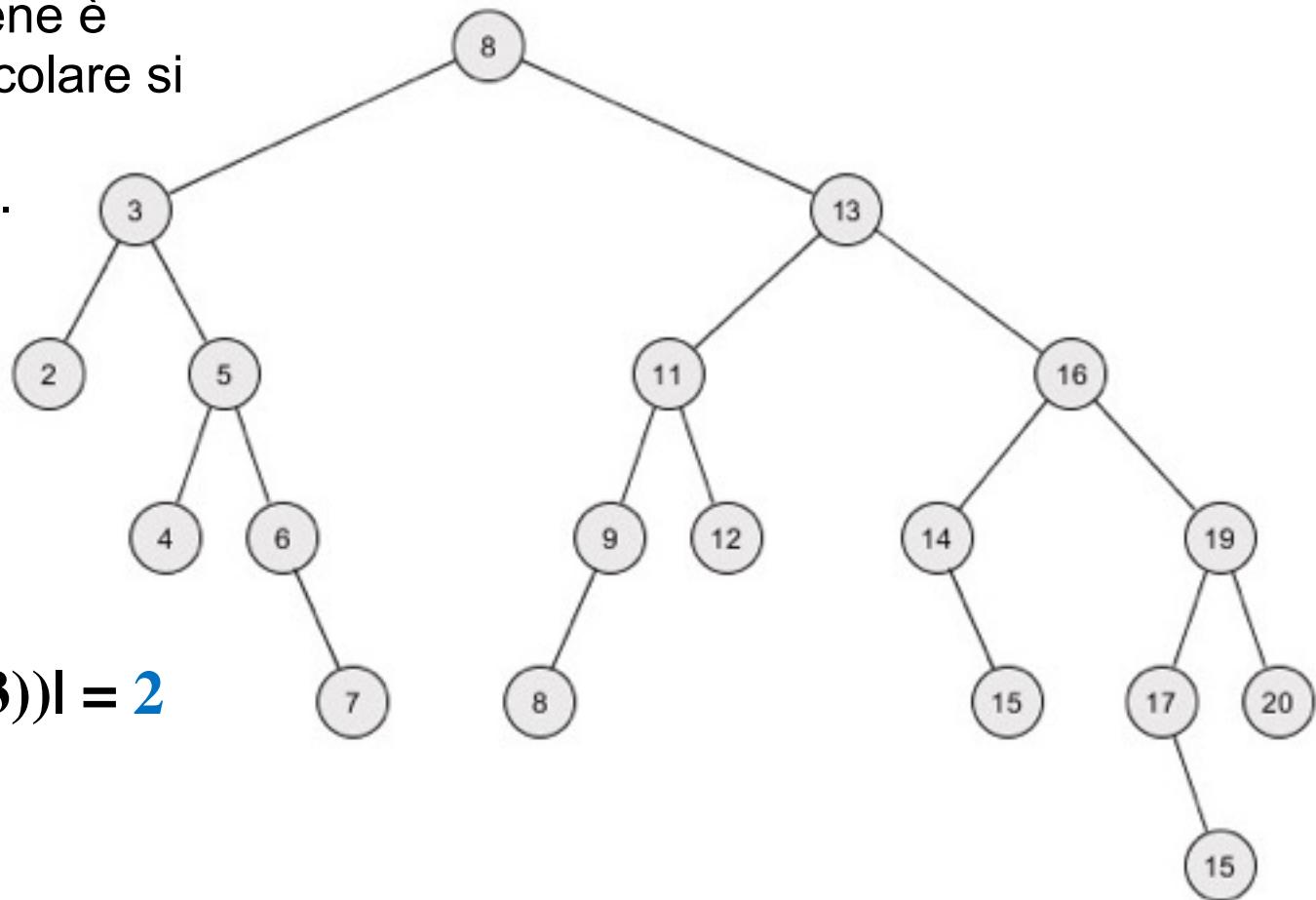
Supponiamo di voler cancellare il nodo

1



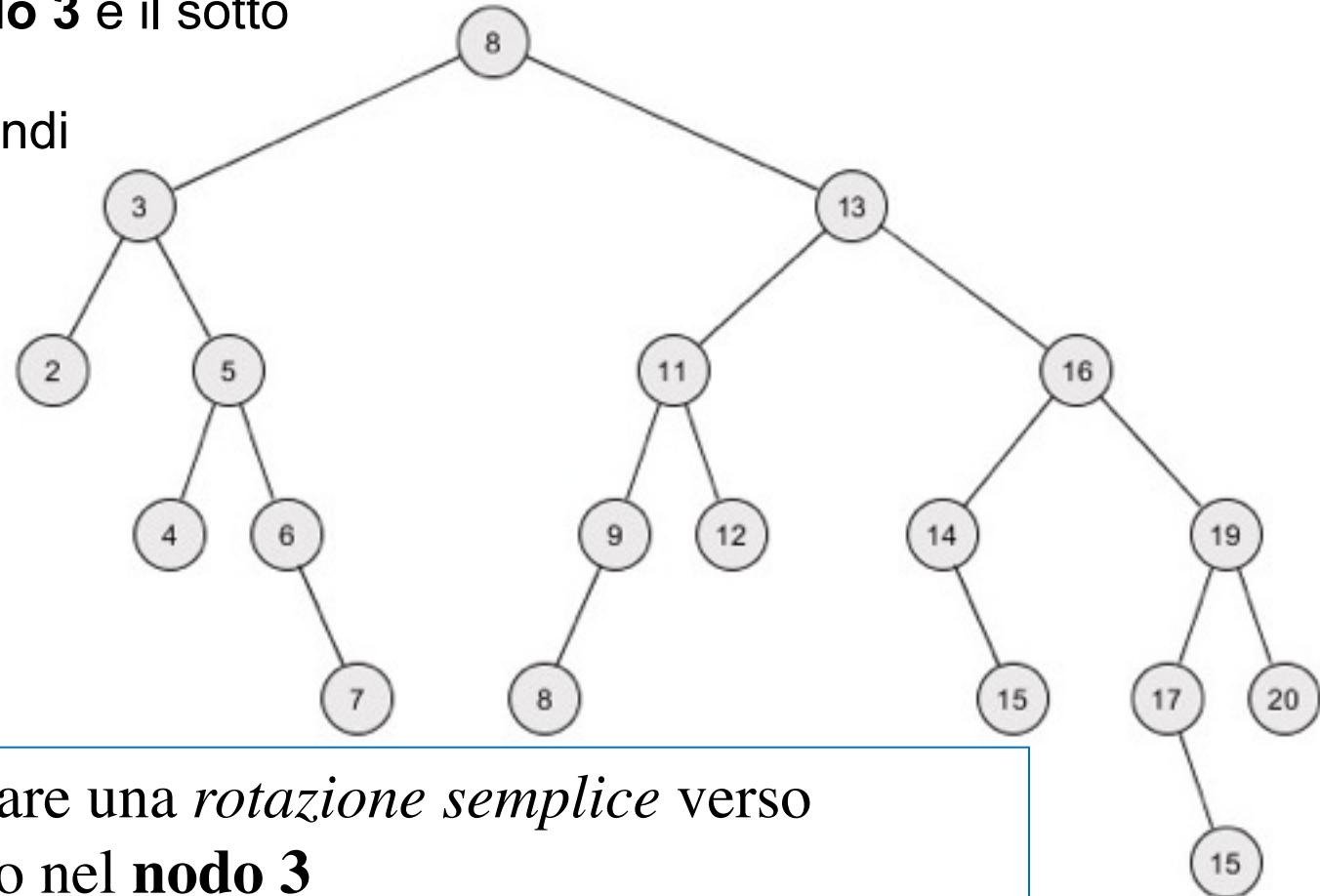
Cancellazione – esempio (2/7)

L'albero che si ottiene è sbilanciato, in particolare si individua il **nodo 3** come **nodo critico**.



Cancellazione – esempio (3/7)

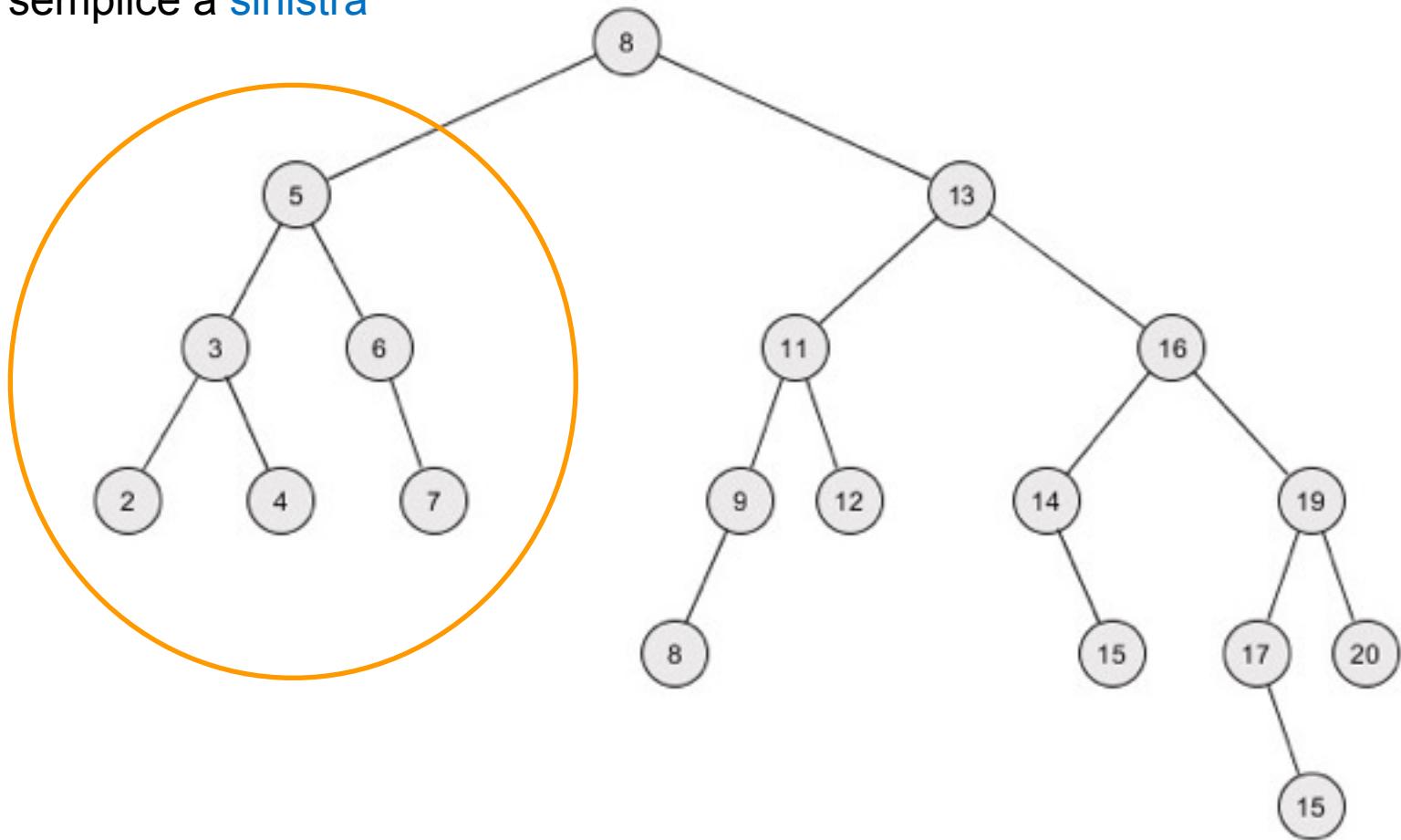
A sbilanciare il **nodo 3** è il sotto
albero dx del suo
figlio dx. Siamo quindi
nel caso DD



Dobbiamo applicare una *rotazione semplice* verso
sinistra con perno nel **nodo 3**

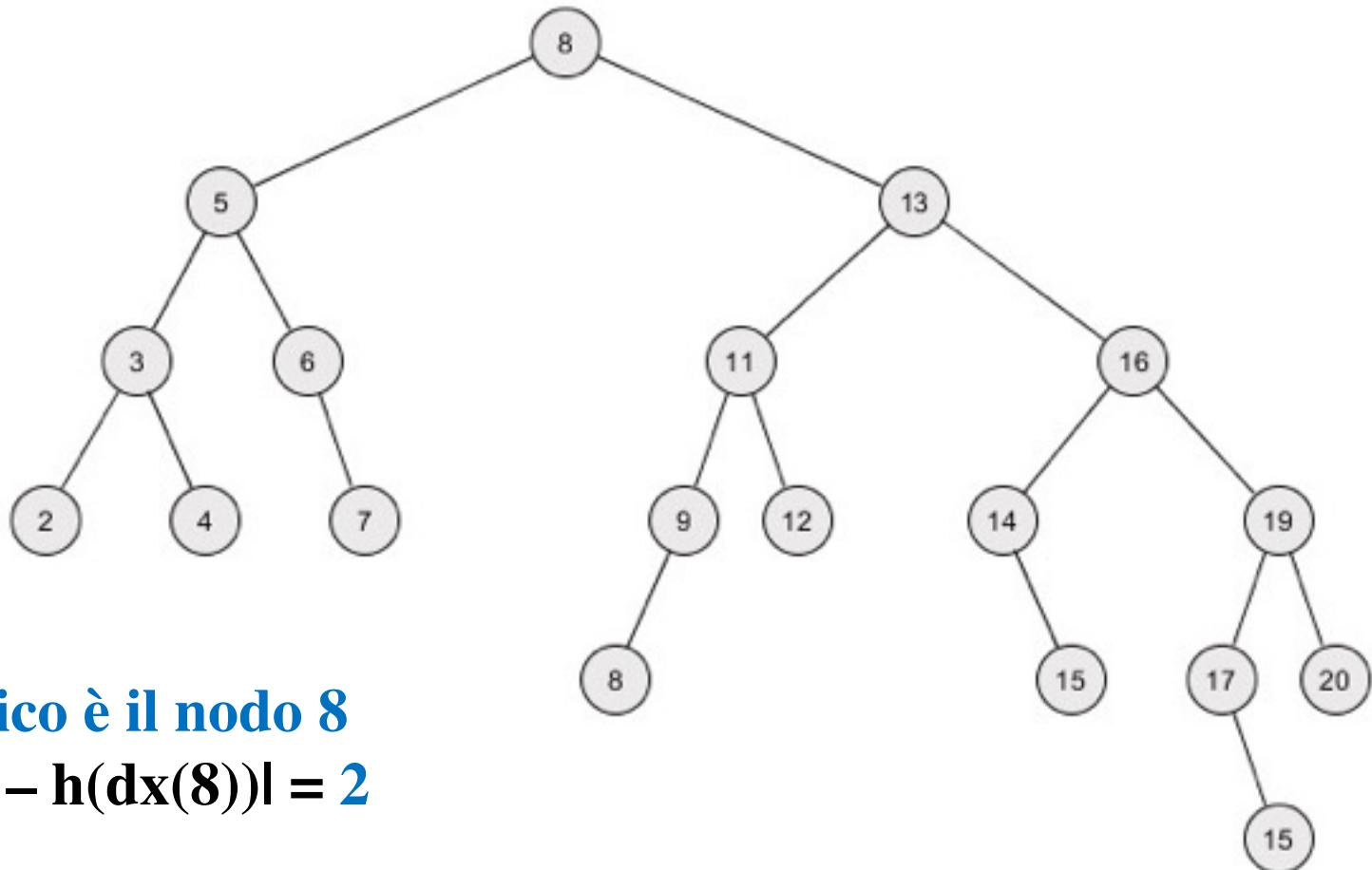
Cancellazione – esempio (4/7)

Rotazione semplice a sinistra



Cancellazione – esempio (5/7)

L'albero è ancora sbilanciato!

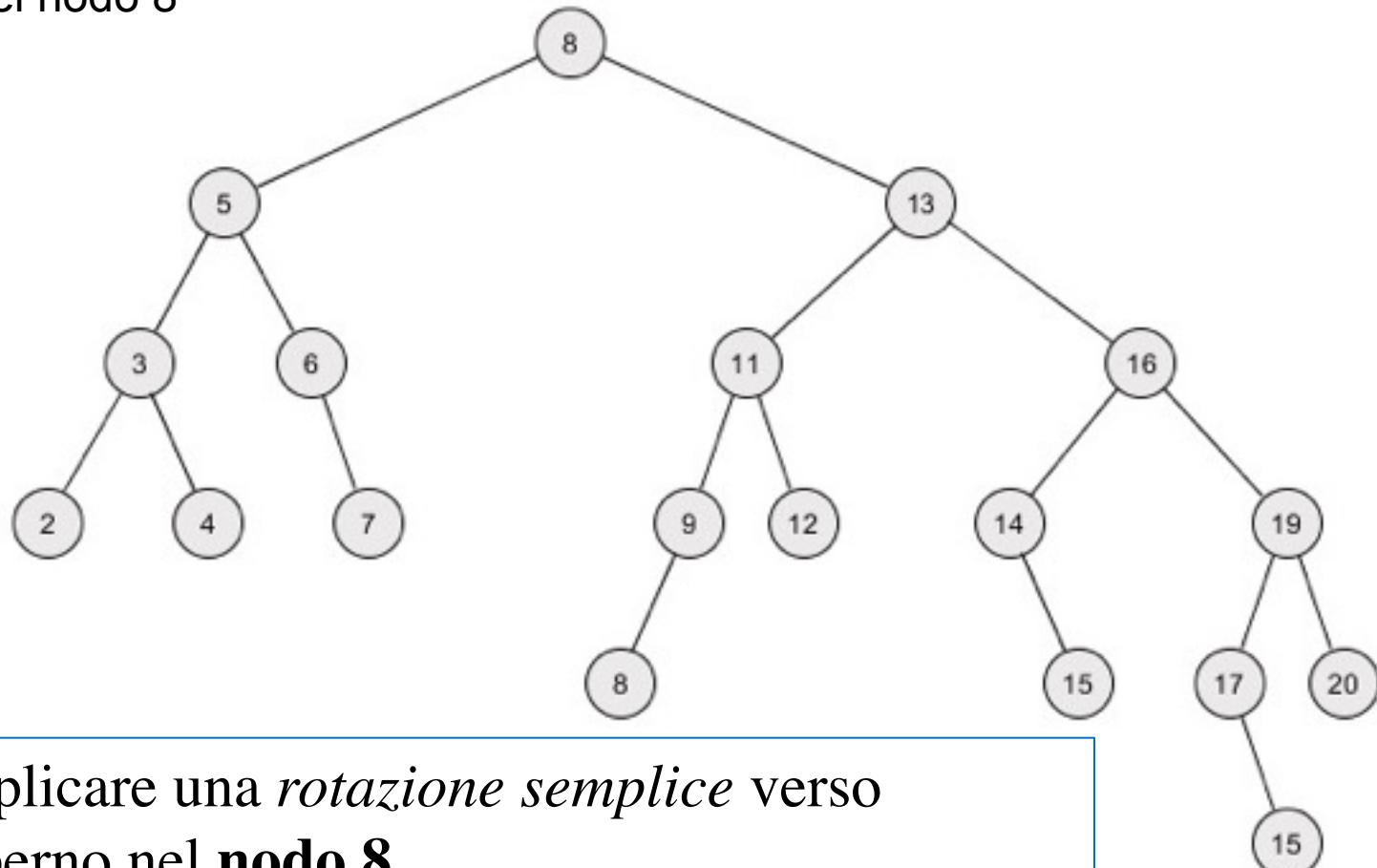


Il nodo critico è il nodo 8

$$\rightarrow |h(\text{sx}(8)) - h(\text{dx}(8))| = 2$$

Cancellazione – esempio (6/7)

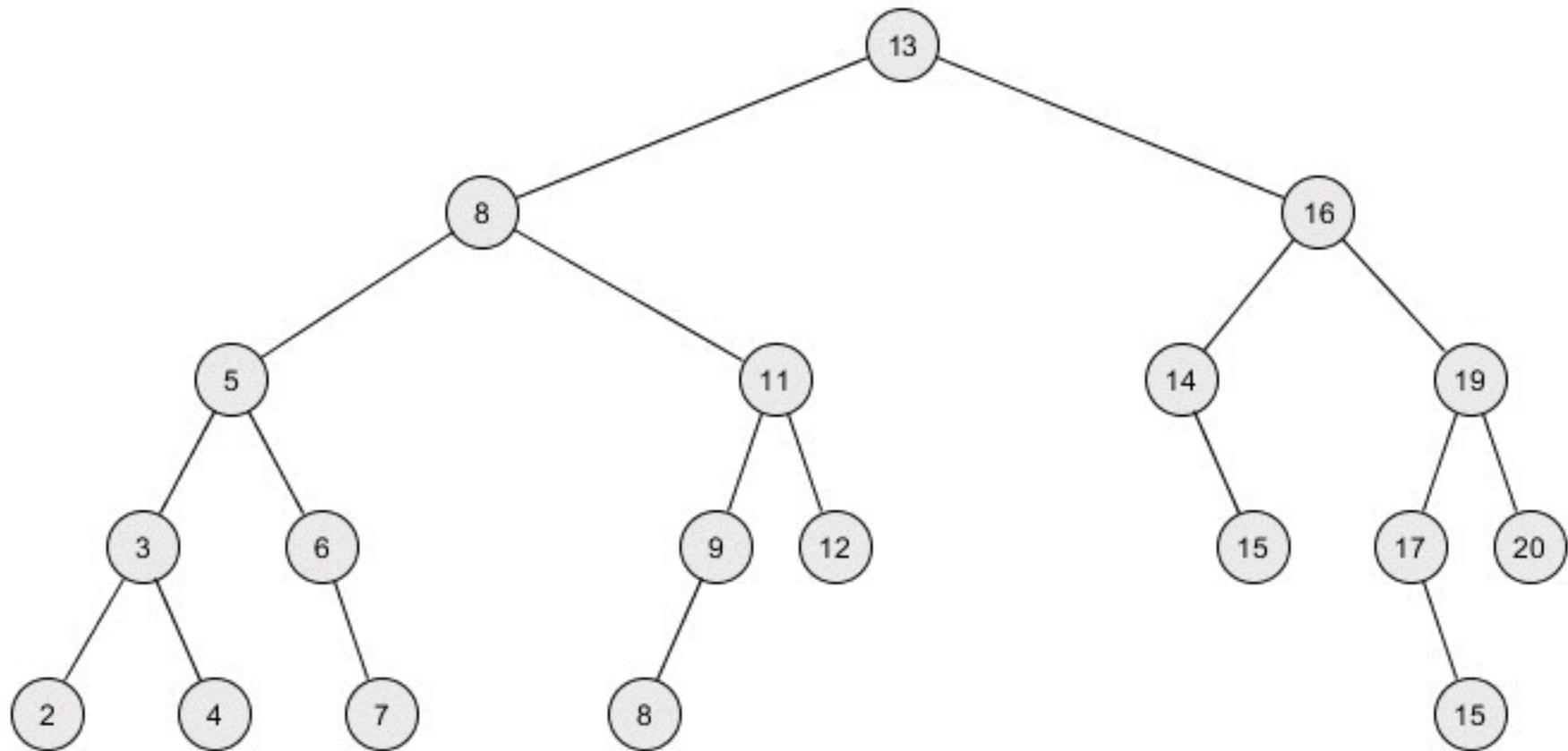
Siamo ancora nel caso **DD**: a sbilanciare
l'albero radicato nel nodo 8 è il sottoalbero
dx del figlio dx del nodo 8



Dobbiamo applicare una *rotazione semplice* verso
sinistra con perno nel **nodo 8**

Cancellazione – esempio (8/8)

Dopo la *rotazione semplice* verso **sinistra** con perno nel **nodo 8** otteniamo finalmente un albero bilanciato



Classe AlberoAVL

classe AlberoAVL estende AlberoBinarioDiRicerca:

dati:

$$S(n) = O(n)$$

albero binario di ricerca T ereditato, più il fattore di bilanciamento di ogni nodo.

operazioni:

search(chiave k) → elem
ereditata.

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

insert(elem e, chiave k)

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

chiama **insert()** ereditata, poi ricalcola i fattori di bilanciamento ed eventualmente ribilancia tramite $O(1)$ rotazioni.

delete(elem e)

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

chiama **delete()** ereditata, poi ricalcola i fattori di bilanciamento ed eventualmente ribilancia tramite $O(\log n)$ rotazioni.

Costo delle operazioni

- Tutte le operazioni hanno costo **$O(\log n)$** poiché l'altezza dell'albero è **$O(\log n)$** e ciascuna rotazione richiede solo tempo costante

Algoritmi e Strutture Dati

Tabelle di hash

Fabio Patrizi

Implementazioni Dizionario

Tempo richiesto dall'operazione più costosa:

- Liste $O(n)$
- Alberi di ricerca non bilanciati $O(n)$
- Alberi di ricerca bilanciati $O(\log n)$
- Tabelle hash** $O(1)$

Tabelle ad accesso diretto

Sono dizionari basati sulla proprietà di accesso diretto alle celle di un array

Idea:

- dizionario memorizzato in array v di m celle
- a ciascun elemento è associata una chiave intera nell'intervallo $[0, m-1]$
- elemento con chiave k contenuto in $v[k]$
- al più $n \leq m$ elementi nel dizionario

Implementazione

classe TavolaAccessoDiretto **implementa** Dizionario:

dati:

$$S(m) = \Theta(m)$$

un array v di dimensione $m \geq n$ in cui $v[k] = elem$ se c'è un elemento $elem$ con chiave k nel dizionario, e $v[k] = \text{null}$ altrimenti. Le chiavi k devono essere interi nell'intervallo $[0, m - 1]$.

operazioni:

insert(*elem e, chiave k*) $T(n) = O(1)$
 $v[k] \leftarrow e$

delete(*chiave k*) $T(n) = O(1)$
 $v[k] \leftarrow \text{null}$

search(*chiave k*) $\rightarrow elem$ $T(n) = O(1)$
return $v[k]$

Fattore di carico

Misuriamo il grado di riempimento di una tabella usando il **fattore di carico**

$$\alpha = \frac{n}{m}$$

Dove **n** è il numero di elementi in essa memorizzati e **m** è la sua dimensione.

Esempio: tabella con nomi di studenti indicizzati da numeri di matricola a 6 cifre

$$n=100 \text{ e } m=10^6 \rightarrow \alpha = 0,0001 = 0,01\%$$

Grande spreco di memoria!

Pregi e difetti

Pregi:

- Tutte le operazioni richiedono tempo $O(1)$

Difetti:

- Le chiavi devono essere necessariamente interi in $[0, m-1]$
- Lo spazio utilizzato è proporzionale ad **m**, non al numero **n** di elementi: può esserci grande spreco di memoria!

Tabelle hash

Per ovviare agli inconvenienti delle tavelle ad accesso diretto ne consideriamo un'estensione: le **tabelle hash**

Idea:

- Chiavi prese da un universo totalmente ordinato **U** (possono non essere numeri)
- Funzione hash: $h: U \rightarrow [0, m-1]$
(funzione che trasforma chiavi in indici)
- Elemento con chiave **k** in posizione **v[h(k)]**

Collisioni

Le tabelle hash possono soffrire del fenomeno delle **collisioni**:

Si ha una collisione quando si deve inserire nella tabella hash un elemento con chiave **u**, e nella tabella esiste già un elemento con chiave **v** tale che $h(u)=h(v)$ → **il nuovo elemento andrebbe a sovrascrivere il vecchio!**

Funzioni hash perfette

Un modo per evitare il fenomeno delle collisioni è usare **funzioni hash perfette**.

Una funzione hash si dice **perfetta** se è iniettiva, cioè per ogni $u, v \in U$:

$$u \neq v \Rightarrow h(u) \neq h(v)$$

Deve essere $|U| \leq m$

Implementazione

classe TavolaHashPerfetta **implementa** Dizionario:

dati:

$$S(m) = \Theta(m)$$

un array v di dimensione $m \geq n$ in cui $v[h(k)] = e$ se c'è un elemento e con chiave $k \in U$ nel dizionario, e $v[h(k)] = \text{null}$ altrimenti. La funzione $h : U \rightarrow \{0, \dots, m - 1\}$ è una funzione hash perfetta calcolabile in tempo $O(1)$.

operazioni:

insert(*elem e, chiave k*) $T(n) = O(1)$
 $v[h(k)] \leftarrow e$

delete(*chiave k*) $T(n) = O(1)$
 $v[h(k)] \leftarrow \text{null}$

search(*chiave k*) \rightarrow *elem* $T(n) = O(1)$
return $v[h(k)]$

Esempio

Tabella hash con nomi di studenti aventi come chiavi numeri di matricola nell'insieme $U=[234717, 235717]$

Funzione hash perfetta: $h(k) = k - 234717$

$n=100$ $m=1000$ $\alpha = 0,1 = 10\%$

L'assunzione $|U| \leq m$ necessaria per avere una funzione hash perfetta è raramente conveniente (o possibile)...

Esempio

Tabella hash con elementi aventi come chiavi lettere dell'alfabeto $U=\{A,B,C,\dots\}$

Funzione hash non perfetta (ma buona in pratica con m numero primo):

$$h(k) = \text{ascii}(k) \bmod m$$

Ad esempio, per $m=11$: $h('C') = h('N')$

⇒ se volessimo inserire sia 'C' che 'N' nel dizionario avremmo una collisione!

Uniformità delle funzioni hash

Per ridurre la probabilità di collisioni, una buona funzione hash dovrebbe essere in grado di distribuire in modo uniforme le chiavi nello spazio degli indici della tabella

Questo si ha ad esempio se la funzione hash gode della proprietà di **uniformità semplice**

Uniformità semplice

Sia $P(k)$ la probabilità che la chiave k sia presente nel dizionario e sia:

$$Q(i) = \sum_{k:h(k)=i} P(k)$$

la probabilità che la cella i sia occupata.

Una funzione hash \mathbf{h} gode **dell'uniformità semplice** se per ogni \mathbf{i} si ha:

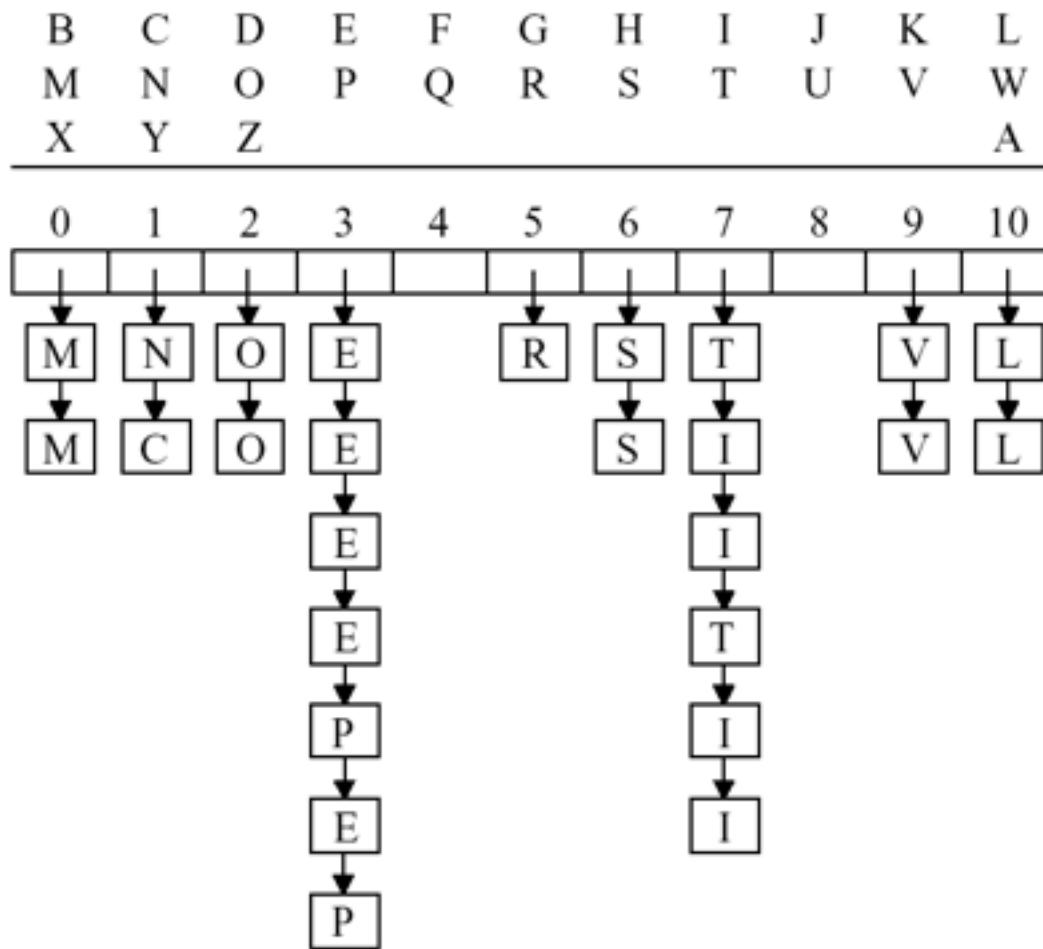
$$Q(i) = \frac{1}{m}$$

Risoluzione delle collisioni

Nel caso in cui non si possano evitare le collisioni, dobbiamo trovare un modo per risolverle. Due metodi classici sono i seguenti:

1. **Liste di collisione.** Gli elementi sono contenuti in liste esterne alla tabella: $v[i]$ punta alla lista degli elementi tali che $h(k)=i$
2. **Indirizzamento aperto.** Tutti gli elementi sono contenuti nella tabella: se una cella è occupata, se ne cerca un'altra libera

Liste di collisione



Esempio di tabella hash basata su liste di collisione contenente le lettere della parola:

**PRECIPITEVOLISS
IMEVOLMENTE**

Liste di collisione - costo

- La lunghezza media di una lista di collisione sarà pari al fattore di carico $\alpha = n/m$. Quindi, assumendo di usare una funzione di hash che gode della uniformità semplice, il tempo medio per la ricerca di un elemento sarà: $T_{avg}(n, m) = O(1 + n/m)$
- A differenza delle tabelle ad accesso diretto e delle tabelle di hash con funzione hash perfetta, usando liste di collisione possiamo avere **fattori di carico $\alpha > 1$**

Implementazione

classe TavolaHashListeColl implementa Dizionario:

dati:

$$S(m, n) = \Theta(m + n)$$

un array v di dimensione m in cui ogni cella contiene un puntatore a una lista di coppie $(elem, chiave)$. Un elemento e con chiave $k \in U$ è nel dizionario se e solo se (e, k) è nella lista puntata da $v[h(k)]$, con $h : U \rightarrow \{0, \dots, m-1\}$ funzione hash con uniformità semplice calcolabile in tempo $O(1)$.

operazioni:

insert(*elem e, chiave k*) $T(n) = O(1)$
aggiungi la coppia (e, k) alla lista puntata da $v[h(k)]$.

delete(*chiave k*) $T_{avg}(n) = O(1 + n/m)$
rimuovi la coppia (e, k) nella lista puntata da $v[h(k)]$.

search(*chiave k*) $\rightarrow elem$ $T_{avg}(n) = O(1 + n/m)$
se (e, k) è nella lista puntata da $v[h(k)]$, allora restituisci e , altrimenti
restituisci null.

trade-off spazio/tempo

Le tabelle di hash con liste di collisione forniscono un ottimo esempio di *bilanciamento spazio-tempo*:

- Per $m = 1$ (minimo spazio) tutte le n chiavi sono in una sola lista e la tabella diventa una struttura a ricerca sequenziale ($T(n,m) = O(1 + n/m) = O(n)$) con spazio $S(n) = O(n)$
- Se invece siamo disposti ad utilizzare molto spazio, allora possiamo usare una funzione di hash perfetta ottenendo tempo per la ricerca $T(n) = O(1)$ con spazio $O(|U|)$ dove U è l'universo delle chiavi associabili agli elementi del dizionario.

Esercizio: liste di collisione

Sia dato l'insieme di chiavi $\mathbf{K} = \{ 35, 83, 57, 26, 15, 63, 97, 46 \}$ e sia $\mathbf{m} = 11$.

1. Calcolare per ogni chiave k di \mathbf{K} la funzione di hash
$$h(\mathbf{k}) = \mathbf{k} \bmod \mathbf{m}$$
2. Inserire le chiavi dell'insieme K in una tabella hash (inizialmente vuota) di dimensione \mathbf{m} usando le **liste di collisione**.

Soluzione: liste di collisione / 1

$\mathbf{K} = \{ 35, 83, 57, 26, 15, 63, 97, 46 \}; \mathbf{m} = 11.$

1. Calcolare per ogni chiave k di \mathbf{K} la funzione di hash

$$h(\mathbf{k}) = \mathbf{k} \bmod \mathbf{m}$$

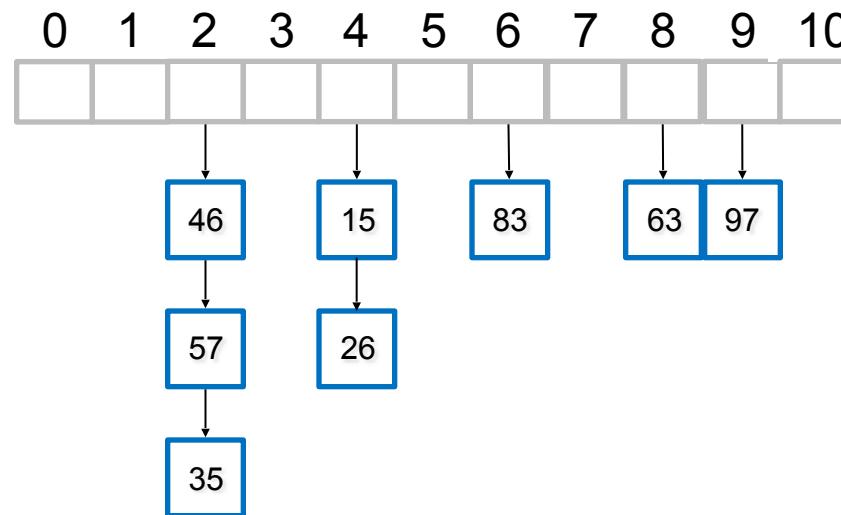
- $h(35) = 2$
- $h(83) = 6$
- $h(57) = 2$
- $h(26) = 4$
- $h(15) = 4$
- $h(63) = 8$
- $h(97) = 9$
- $h(46) = 2$

Soluzione: liste di collisione / 2

$\mathbf{K} = \{ 35, 83, 57, 26, 15, 63, 97, 46 \}; \mathbf{m} = 11.$

2. Inserire le chiavi dell'insieme K in una tabella hash (inizialmente vuota) di dimensione \mathbf{m} usando le **liste di collisione**.

- $h(35) = 2$
- $h(83) = 6$
- $h(57) = 2$
- $h(26) = 4$
- $h(15) = 4$
- $h(63) = 8$
- $h(97) = 9$
- $h(46) = 2$



Indirizzamento aperto

Supponiamo di voler inserire un elemento con chiave k e che la sua posizione “naturale” $h(k)$ sia già occupata

L’indirizzamento aperto consiste nell’occupare un’altra cella, anche se naturalmente associata ad un’altra chiave

Cerchiamo la cella vuota (se c’è) scandendo le celle secondo una sequenza di indici:

$$c(k,0), c(k,1), c(k,2), \dots c(k,m-1)$$

Implementazione

classe TavolaHashAperta **implementa** Dizionario:

dati:

$$S(m) = \Theta(m)$$

un array v di dimensione m in cui ogni cella contiene una coppia $(elem, chiave)$.

operazioni:

insert(*elem e, chiave k*)

1. **for** $i = 0$ **to** $m - 1$ **do**
2. **if** ($v[c(k, i)].elem = null$) **then**
3. $v[c(k, i)] \leftarrow (e, k)$
4. **return**
5. **errore** tavola piena

delete(*chiave k*)

errore operazione non supportata

search(*chiave k*) \rightarrow *elem*

1. **for** $i = 0$ **to** $m - 1$ **do**
2. **if** ($v[c(k, i)].elem = null$) **then**
3. **return** null
4. **if** ($v[c(k, i)].chiave = k$) **then**
5. **return** $v[c(k, i)].elem$
6. **return** null

Metodi di scansione: scansione lineare

Scansione lineare:

$$c(k,i) = (h(k) + i) \bmod m$$

per $0 \leq i < m$

Esempio

	C	E	I	L	M	N	O	P	R	S	T	V																		
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30
P																														
R																														
E																														
C																														
I																														
P																														
I																														
T																														
E																														
V																														
O																														
L																														
I																														
S																														
S																														
I																														
M																														
E																														
V																														
O																														
L																														
M																														
E																														
N																														
T																														
E																														
E																														

Inserimenti in tabella hash basata su indirizzamento aperto con scansione lineare delle lettere della parola:

**PRECIPITEVOLISS
IMEVOLMENTE**

4,8 celle scandite in media per inserimento

Metodi di scansione: hashing doppio

La scansione lineare provoca effetti di agglomerazione, cioè lunghi gruppi di celle consecutive occupate che rallentano la scansione

L'hashing doppio riduce il problema:

$$c(k,i) = \lfloor h_1(k) + i \cdot h_2(k) \rfloor \bmod m$$

per $0 \leq i < m$, h_1 e h_2 funzioni hash

Esempio

	C	E	I	L	M	N	O	P	R	S	T	V																		
0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30
P																														
R																														
E																														
C																														
I																														
P																														
I																														
T																														
E																														
V																														
O																														
L																														
I																														
S																														
S																														
I																														
M																														
M																														
V																														
O																														
L																														
M																														
M																														
E																														
N																														
M																														
T																														
E																														

Inserimenti in tabella hash basata su indirizzamento aperto con hashing doppio delle lettere della parola:

**PRECIPITEVOLISS
IMEVOLMENTE**

3,1 celle scandite in media per inserimento

Analisi del costo di scansione

Usando l'indirizzamento aperto la ricerca di un elmento o di una cella vuota può richiedere tempo $O(n)$.

Tempo richiesto in media da un'operazione di ricerca di una chiave, assumendo che le chiavi siano prese con probabilità uniforme da U dipenderà dal fattore di carico e dalla particolare funzione $c(k, i)$ utilizzata.

Il tempo medio richiesto per le operazioni di search, insert e delete è:

- $O\left(\frac{m^2}{(m-n)^2}\right)$ usando la *scansione lineare*
- $O\left(\frac{m}{m-n}\right)$ usando la *scansione quadratica* o l'*hashing doppio*

Esercizio: indirizzamento aperto

Sia dato l'insieme di chiavi $\mathbf{K} = \{ 35, 83, 57, 26, 15, 63, 97, 46 \}$ e sia $\mathbf{m} = 11$.

1. Calcolare per ogni chiave k di \mathbf{K} la funzione di hash

$$h(k) = k \bmod m$$

2. Inserire le chiavi dell'insieme K in una tabella hash (inizialmente vuota) di dimensione \mathbf{m} usando l'indirizzamento aperto e con scansione lineare data da:

$$c(k, i) = (h(k) + i) \bmod 11$$

Soluzione: indirizzamento aperto

$$h(35) = 2$$

$$h(83) = 6$$

$$h(57) = 2$$

$$h(26) = 4$$

$$h(15) = 4$$

$$h(63) = 8$$

$$h(97) = 9$$

$$h(46) = 2$$

$$c(35,0) = (2 + 0) \bmod 11 = 2$$

$$c(83,0) = (6 + 0) \bmod 11 = 6$$

$$c(57,0) = (2 + 0) \bmod 11 = 2$$

$$c(26,1) = (2 + 1) \bmod 11 = 3$$

$$c(15,0) = (4 + 0) \bmod 11 = 4$$

$$c(15,1) = (4 + 1) \bmod 11 = 5$$

$$c(63,0) = (8 + 0) \bmod 11 = 8$$

$$c(97,0) = (9 + 0) \bmod 11 = 9$$

$$c(46,0) = (2 + 0) \bmod 11 = 2$$

$$c(46,1) = (2 + 1) \bmod 11 = 3$$

$$c(46,2) = (2 + 2) \bmod 11 = 4$$

$$c(46,3) = (2 + 3) \bmod 11 = 5$$

$$c(46,4) = (2 + 4) \bmod 11 = 6$$

$$c(46,5) = (2 + 5) \bmod 11 = 7$$

0 1 2 3 4 5 6 7 8 9 10



Cancellazione elementi con indir. aperto

- Se per cancellare un elemento sostituisco il valore della cella che lo contiene con *null* l'implementazione del metodo di ricerca non funzionerebbe più!
- **Idea:** utilizzo un valore speciale *canc* invece di *null*.



Cancellazione elementi con indir. aperto

classe TavolaHashApertaBis implementa Dizionario:
dati:

$$S(m) = \Theta(m)$$

un array v di dimensione m in cui ogni cella contiene una coppia $(elem, chiave)$.

operazioni:

insert(*elem e, chiave k*)

1. **for** $i = 0$ **to** $m - 1$ **do**
2. **if** ($v[c(k, i)].elem = \text{null}$ **or** $v[c(k, i)].elem = \text{canc}$) **then**
3. $v[c(k, i)] \leftarrow (e, k)$
4. **return**
5. **errore** tavola piena

delete(*chiave k*)

1. **for** $i = 0$ **to** $m - 1$ **do**
2. **if** ($v[c(k, i)].elem = \text{null}$) **then**
3. **errore** chiave non in dizionario
4. **if** ($v[c(k, i)].chiave = k$ **and** $v[c(k, i)].elem \neq \text{canc}$) **then**
5. $v[c(k, i)].elem \leftarrow \text{canc}$
6. **errore** chiave non in dizionario

search(*chiave k*) $\rightarrow elem$

1. **for** $i = 0$ **to** $m - 1$ **do**
2. **if** ($v[c(k, i)].elem = \text{null}$) **then**
3. **return** null
4. **if** ($v[c(k, i)].chiave = k$ **and** $v[c(k, i)].elem \neq \text{canc}$) **then**
5. **return** $v[c(k, i)].elem$
6. **return** null

Riepilogo

- La proprietà di accesso diretto alle celle di un array consente di realizzare dizionari con operazioni in tempo $O(1)$ indicizzando gli elementi usando le loro stesse chiavi (purché siano intere)
- L'array può essere molto grande se lo spazio delle chiavi è grande
- Per ridurre questo problema si possono usare funzioni hash che trasformano chiavi (anche non numeriche) in indici
- Usando funzioni hash possono avversi collisioni
- Tecniche classiche per risolvere le collisioni sono liste di collisione e indirizzamento aperto

Algoritmi e Strutture Dati

Grafi e visite di grafi

Fabio Patrizi

Grafo: definizione

Un grafo $G=(V,E)$ consiste in:

- un insieme V di vertici (o nodi)
- un insieme E di coppie di vertici, detti **archi** (o *spigoli*): ogni arco connette due vertici

Grafo orientato

Grafo orientato (o diretto): ogni arco è *orientato* e rappresenta relazioni *orientate* tra coppie di oggetti

Esempio:

$V = \{\text{persone che vivono in Italia}\},$
 $E = \{(x,y) \text{ tale che } x \text{ ha inviato una mail a } y\}$

Esempio:

$V = \{\text{persone che vivono in Italia}\},$
 $E = \{(x,y) \text{ tale che } x \text{ è padre di } y\}$

Grafo non orientato

Grafo non orientato (o non diretto): gli archi non hanno un orientazione (relazioni simmetriche)

Esempio:

$V = \{\text{persone che vivono in Italia}\}$,
 $E = \{\text{coppie di persone che si sono strette la mano}\}$

Esempio:

$V = \{\text{persone che vivono in Italia}\}$,
 $E = \{\text{coppie di fratelli}\}$

Terminologia

relazione **simmetrica** → grafo **non orientato**

relazione **non simmetrica** → grafo **orientato**

Sia **G** un grafo. Denotiamo con:

- **n** il numero di vertici di **G**
- **m** il numero di archi di **G**
- **V(G)** l'insieme dei nodi di **G**
- **E(G)** l'insieme degli archi di **G**
- Quindi: **|V(G)| = n** e **|E(G)| = m**

Adiacenza ed Incidenza

- Sia (x,y) un arco del grafo \mathbf{G} , diciamo che (x,y) è **incidente** sui nodi x e y ;
- Se \mathbf{G} è un grafo *orientato*, allora diciamo che (x,y) **esce** da x ed **entra** in y ;
- Se \mathbf{G} è un grafo *non orientato* allora diremo che x è **adiacente** ad y e che y è **adiacente a** x , ovvero x ed y sono **adiacenti**;
- Dato un nodo v chiameremo i nodi adiacenti a v i **vicini** di v

Grado di un vertice (1/2)

- Il **grado** di un vertice **v** in un grafo è dato dal numero di archi **incidenti** su **v**.
- Denotiamo con $\delta(v)$ il grado del vertice **v**.
- Se sommiamo il grado di tutti i nodi di un grafo **G** *non orientato*, andiamo a contare ogni arco **2** volte (es: (x,y) una volta per x ed una volta per y).
- Quindi:

$$\sum_{v \in V} \delta(v) = 2m$$

Grado di un vertice (2/2)

Se il grafo G è *orientato*, parleremo di:

- **Grado in uscita** di v, intendendo con questo il numero di archi che escono da v e lo indicheremo con $\delta_{\text{out}}(v)$
- **Grado in entrata** di v, intendendo con questo il numero di archi che entrano in v e lo indicheremo con $\delta_{\text{in}}(v)$
- Il **grado** di un vertice sarà dato dalla somma dei due:

$$\delta(v) = \delta_{\text{in}}(v) + \delta_{\text{out}}(v)$$

Cammini

Un **cammino** in un grafo G da un vertice x ad un vertice y è dato da una sequenza di vertici $[v_0, v_1, \dots, v_k]$, con $v_0 = x$ e $v_k = y$, tale che per ogni $1 \leq i \leq k$, la coppia (v_{i-1}, v_i) appartiene a $E(G)$

In tal caso diremo che il cammino è di **lunghezza** k .

La lunghezza del più **corto cammino** tra due vertici è la **distanza** tra i vertici

Diremo che un cammino è **semplice** se tutti i suoi vertici sono distinti.

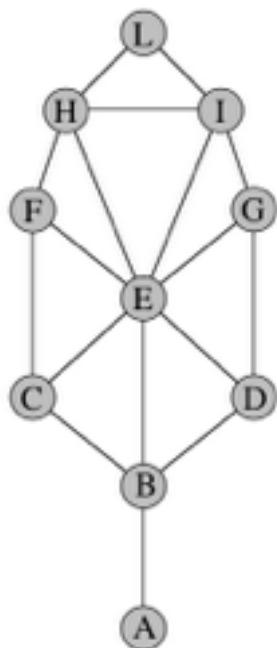
Indichiamo il cammino da x a y con $x \rightarrow y$

Se esiste un cammino da x ad y diremo che y è **raggiungibile** da x

Cicli

- Un cammino $[v_0, v_1, \dots, v_k]$ tale che $v_0 = v_k$, con $k \geq 1$, è detto **ciclo**
- Il ciclo è detto **semplice** se tutti i nodi v_1, \dots, v_{k-1} sono distinti.
- Un grafo è **aciclico** se **non** contiene cicli.

Esempio



Il grafo in figura è un grafo **non** orientato

Il vertice **H** ha grado 4: $\delta(v) = 4$

Esiste un cammino semplice di lunghezza 4 tra il vertice **A** ed il vertice **L**, contenente i vertici **A, B, E, I, L** e gli archi **(A,B), (B,E), (E,I),** ed **(I,L)**

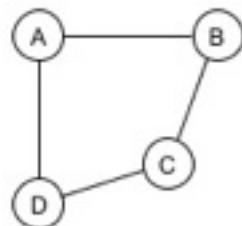
Dato che 4 è anche la lunghezza del cammino più corto tra **A** ed **L**, allora **A** ed **L** sono a *distanza 4*.

Connettività e connettività forte

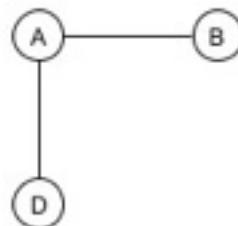
- Un grafo *non orientato* $\mathbf{G} = (\mathbf{V}, \mathbf{E})$ si dice **connesso** se esiste un cammino tra ogni coppia di vertici in \mathbf{G} .
- Un grafo *orientato* $\mathbf{G} = (\mathbf{V}, \mathbf{E})$ si dice **fortemente connesso** se esiste un cammino (orientato) tra ogni coppia di vertici in \mathbf{G} .
- Diremo che un vertice x è **fortemente connesso** ad un vertice y se esiste un cammino (orientato) da x ad y ed un cammino (orientato) da y a x .

Sottografi

- Diciamo che un grafo $\mathbf{G}' = (\mathbf{V}', \mathbf{E}')$ è un *sottografo* di un grafo $\mathbf{G} = (\mathbf{V}, \mathbf{E})$ se $\mathbf{V}' \subseteq \mathbf{V}$ e $\mathbf{E}' \subseteq \mathbf{E}$.
In questo caso useremo la notazione $\mathbf{G}' \subseteq \mathbf{G}$.
- Dato un grafo $\mathbf{G} = (\mathbf{V}, \mathbf{E})$ ed un insieme di vertici $\mathbf{V}' \subseteq \mathbf{V}$, il *sottografo* di \mathbf{G} *indotto* da \mathbf{V}' è definito come il grafo $\mathbf{G}' = (\mathbf{V}', \mathbf{E}')$ dove $\mathbf{E}' = \{ (x,y) \in \mathbf{E} \text{ tale che } x,y \in \mathbf{V}' \}$.



\mathbf{G}



\mathbf{G}'

\mathbf{G}' è il sottografo di \mathbf{G}
indotto da $\mathbf{V}' = \{A, B, D\}$

Strutture dati per rappresentare grafi

Operazioni sui grafi

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.

grado(vertice v) → intero

restituisce il numero di archi incidenti sul vertice *v*.

archiIncidenti(vertice v) → {arco, arco, ..., arco}

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

restituisce true se esiste l'arco (*x, y*), e false altrimenti

aggiungiVertice(vertice v)

inserisce un nuovo vertice *v*.

aggiungiArco(vertice x, vertice y)

inserisce un nuovo arco tra i vertici *x* e *y*.

rimuoviVertice(vertice v)

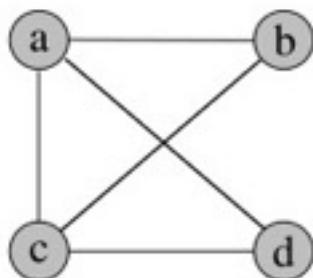
cancella il vertice *v* e tutti gli archi ad esso incidenti.

rimuoviArco(arco e)

cancella l'arco *e*.

A titolo di esempio consideriamo le seguenti operazioni su grafi non orientati

Lista di archi



I vertici del grafo sono rappresentati come record in una lista. Ogni record contiene le informazioni che si vogliono associare al nodo (Es. etichetta)

Gli archi del grafo sono rappresentati come record di una lista. Ogni record contiene i puntatori ai vertici che sono gli estremi dell'arco

(a,b)
(c,a)
(b,c)
(c,d)
(a,d)

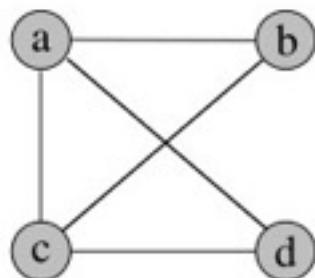
Lo spazio utilizzato sarà $O(n+m)$

Rappresentazione poco efficiente → molte operazioni richiedono di scandire l'intera lista degli archi.

Prestazioni della lista di archi

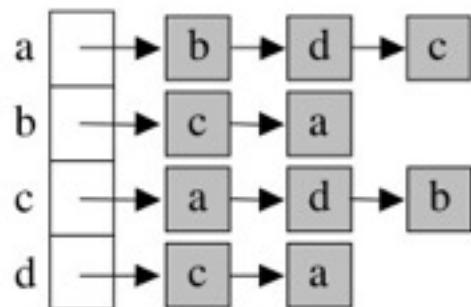
Operazione	Tempo di esecuzione
<code>grado(v)</code>	$O(m)$
<code>archiIncidenti(v)</code>	$O(m)$
<code>sonoAdiacenti(x, y)</code>	$O(m)$
<code>aggiungiVertice(v)</code>	$O(1)$
<code>aggiungiArco(x, y)</code>	$O(1)$
<code>rimuoviVertice(v)</code>	$O(m)$
<code>rimuoviArco(e)</code>	$O(m)$

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) .

Vi saranno **n** liste, una per ogni vertice.
Lo spazio utilizzato sarà $O(n+m)$



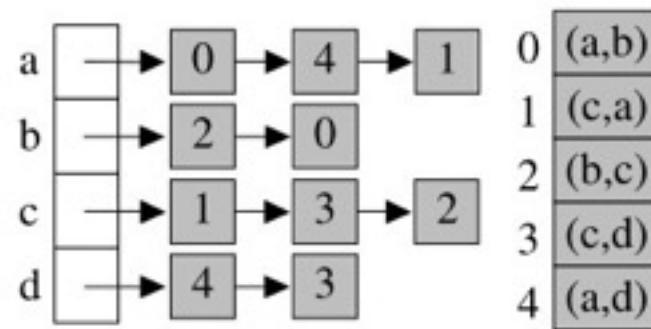
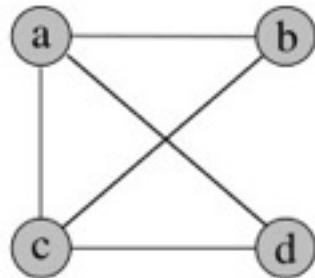
Efficiente per quelle operazioni in cui si chiede di visitare un grafo esaminando gli archi incidenti sui vertici.

Contro: per sapere se due nodi sono adiacenti devo scandire tutta la lista di adiacenza.

Prestazioni delle liste di adiacenza

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

Liste di incidenza



Combinazione tra la rappresentazione con liste di archi e liste di adiacenza: in aggiunta alla rappresentazione con lista di archi, memorizziamo per ogni vertice v una lista di puntatori agli archi incidenti a v .

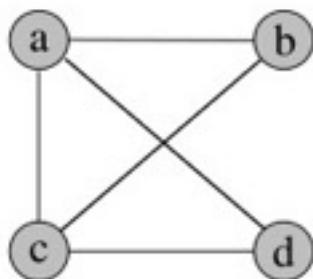
Lo spazio utilizzato sarà leggermente maggiore, ma comunque nell'ordine di $O(n+m)$

Prestazioni della lista di incidenza

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



Matrici di adiacenza



Assumiamo che i vertici del grafo corrispondano a numeri interi da 1 ad n.

Sia M una matrice $n \times n$ le cui righe e colonne sono "indicizzate" dai vertici del grafo. Avremo:

$$M[u,v] = \begin{cases} 1 & \text{se } (u,v) \text{ è un arco di } G \\ 0 & \text{altrimenti} \end{cases}$$

	a	b	c	d
a	0	1	1	1
b	1	0	1	0
c	1	1	0	1
d	1	0	1	0

Utilizziamo molto spazio (matrice $n \times n$) però possiamo verificare la presenza di un arco (u,v) in tempo costante.

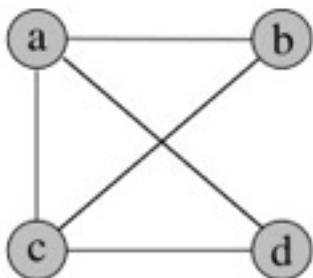
Utile se si voglio adottare tecniche basate su calcoli algebrici (ES: prodotto tra matrici)

Prestazioni della matrice di adiacenza

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



Matrici di incidenza



	(a,b)	(c,a)	(b,c)	(c,d)	(a,d)
a	1	1	0	0	1
b	1	0	1	0	0
c	0	1	1	1	0
d	0	0	0	1	1

La matrice di incidenza è una matrice **$n \times m$** le cui righe sono "indicizzate" dai vertici e le colonne sono "indicizzate" dagli archi. La posizione $M[v,e]$ sarà uguale ad 1 se l'arco e incide sul vertice v .

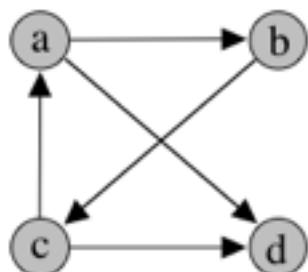
- ogni colonna ha esattamente due 1: la colonna corrispondente all'arco (x,y) avrà un 1 nella riga corrispondente ad x ed un 1 nella riga corrispondente ad y

Prestazioni della matrice di incidenza

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



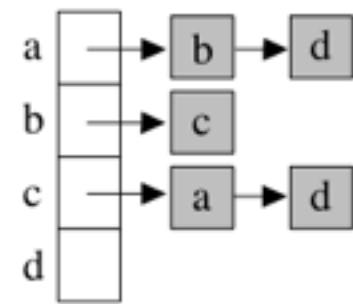
Rappresentazione di grafi orientati



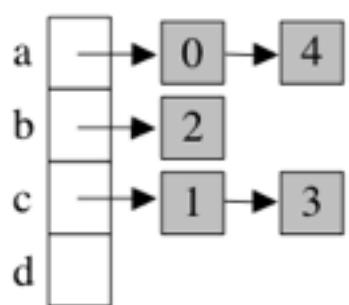
(a) Grafo orientato G

(a,b)
(c,a)
(b,c)
(c,d)
(a,d)

(b) Lista di archi di G



(c) Liste di adiacenza di G



0	(a,b)
1	(c,a)
2	(b,c)
3	(c,d)
4	(a,d)

(d) Liste di incidenza di G

	a	b	c	d
a	0	1	0	1
b	0	0	1	0
c	1	0	0	1
d	0	0	0	0

(e) Matrice di adiacenza di G

	(a,b)	(c,a)	(b,c)	(c,d)	(a,d)
a	1	-1	0	0	1
b	-1	0	1	0	0
c	0	1	-1	1	0
d	0	0	0	-1	-1

(f) Matrice di incidenza di G

Visite di grafi

Scopo e tipi di visita

- Una visita (o attraversamento) di un grafo G permette di esaminare i nodi e gli archi di G in modo sistematico
- Problema di base in molte applicazioni
- Esistono vari tipi di visite con diverse proprietà: in particolare:
 - **visita in ampiezza** (BFS=breadth first search)
 - **visita in profondità** (DFS=depth first search)
- Definiamo prima un algoritmo di visita "generico".

Algoritmo di visita generica

Assumiamo che ogni nodo di \mathbf{G} sia raggiungibile da s

algoritmo visitaGenerica (*vertice s*)

M, F : insiemi vuoti di vertici

 aggiungi s ad F e M

while ($F \neq \emptyset$) **do**

 estrai un vertice u da F

 visita u

for each (arco (u, v) in G) **do**

if ($v \notin M$) **then**

 aggiungi v ad F e M

Algoritmo di visita generica

1. L'insieme F è detto **frangia** (o frontiera)
2. I nodi di M sono detti **marcati**: sono i nodi già inseriti nella frangia
3. Il sistema di marcatura garantisce che nessun nodo venga inserito nella frangia (e quindi visitato) più di una volta

Costo della visita

Il tempo di esecuzione dipende dalla struttura dati usata per rappresentare il grafo.

In particolare, la visita di un grafo $G = (V, E)$ con **n** vertici e **m** archi costerà:

- **O(mn)** se il grafo è rappresentato come una **lista di archi**
- **O($m+n$)** se il grafo è rappresentato come una **lista di adiacenza o di incidenza**
- **O(n^2)** se il grafo è rappresentato come una **matrice di adiacenza**

La rappresentazione mediante lista di adiacenza (o incidenza) è quindi la più efficiente rispetto alla visita di una grafo

Visite particolari

- Se la frangia F è implementata come **coda** si ha la visita in ampiezza (BFS)
- Se la frangia F è implementata come **pila** si ha la visita in profondità (DFS)

Visita in ampiezza

Visita in ampiezza

La visita procede in ampiezza sul grafo a partire dai vertici incontrati

La frangia è implementata come una **coda** (politica di accesso FIFO)

Visita in ampiezza: algoritmo

algoritmo visitaBFS (*vertice s*)

M: insieme vuoto di vertici

F: coda vuota di vertici

F.enqueue(s)

aggiungi *s* ad *M*

while (*F* ≠ \emptyset) **do**

u = *F.dequeue()*

 visit *u*

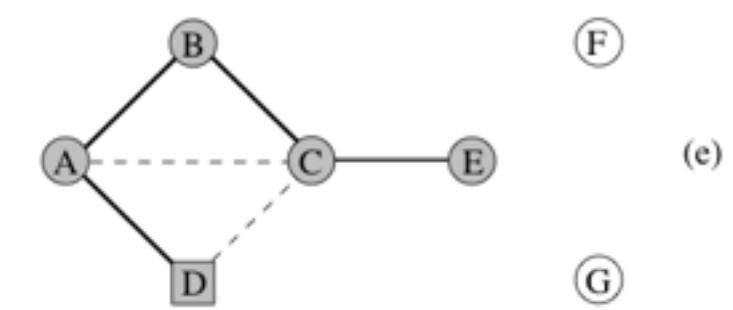
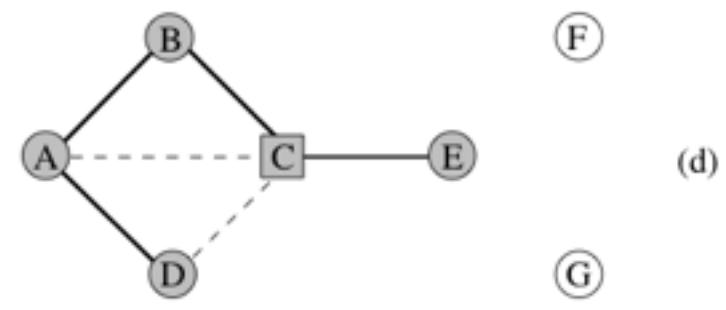
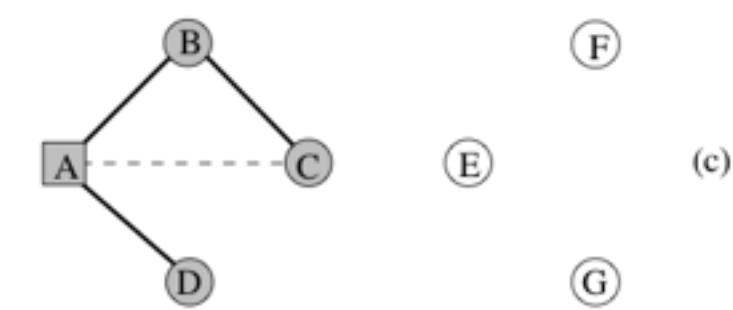
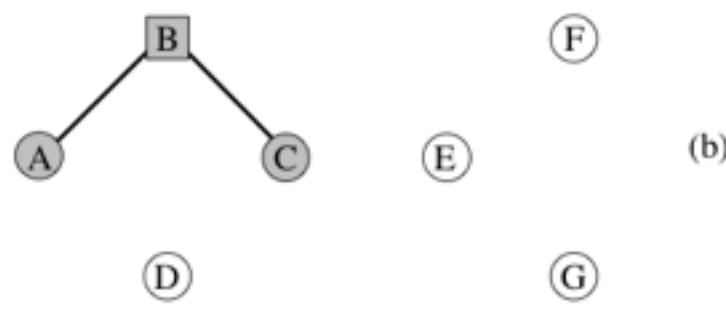
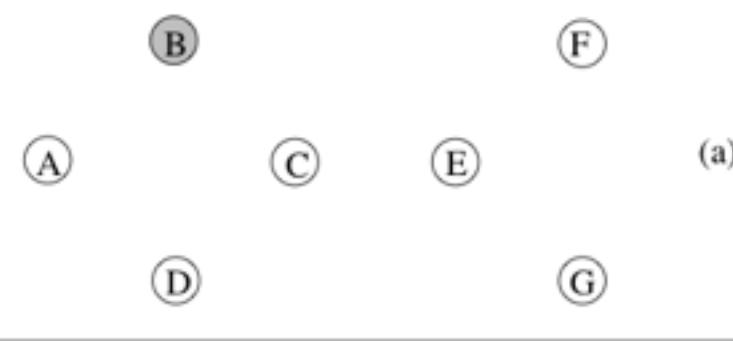
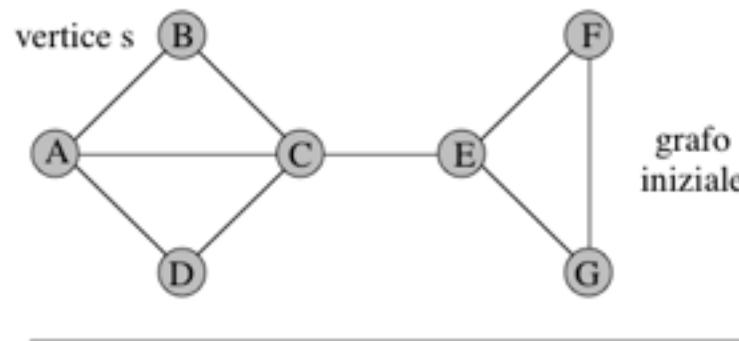
for each (arco (*u,v*) in *G*) **do**

if (*v* ∉ *M*) **then**

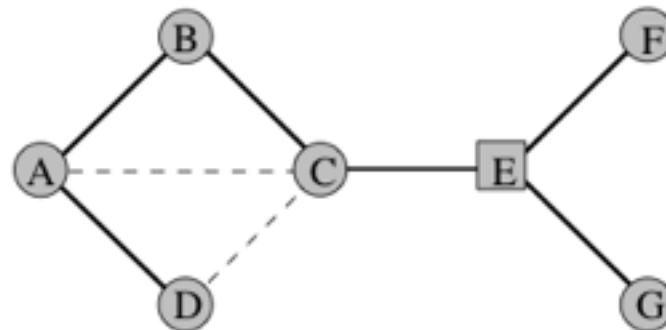
F.enqueue(v)

 aggiungi *v* ad *M*

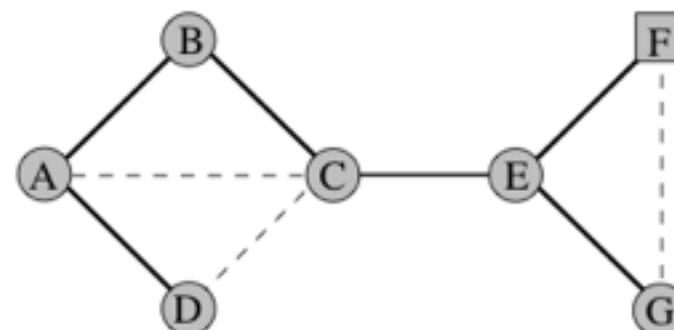
Esempio: grafo non orientato (1/2)



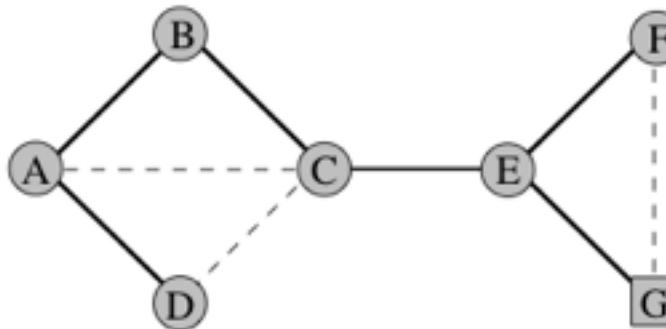
Esempio: grafo non orientato (2/2)



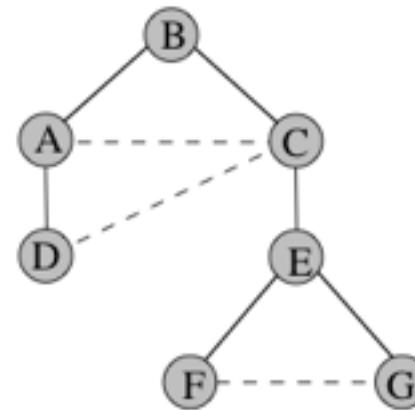
(f)



(g)



(h)



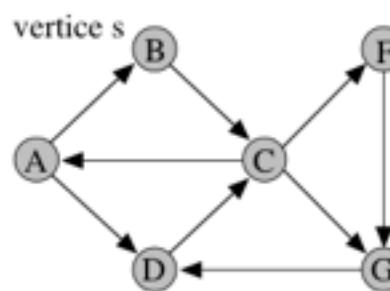
albero
BFS

Visita in ampiezza: grafi orientati

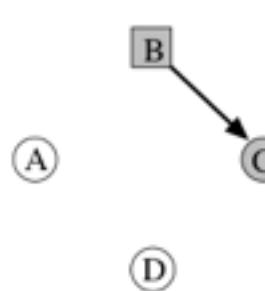
La visita in ampiezza può essere applicata anche ai grafi orientati.

Nell'operazione di scelta dei vicini di un vertice v (implementata nel passo 8 dell'algoritmo) si dovranno scegliere **solo** gli archi (v, w) *uscenti* da v

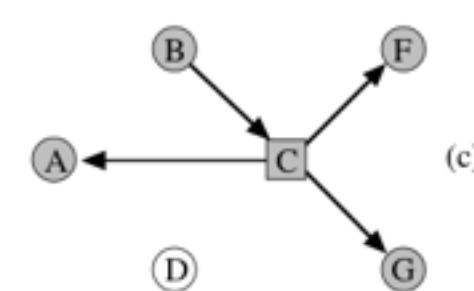
Esempio: grafo orientato



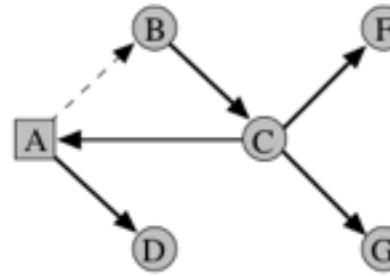
(a)



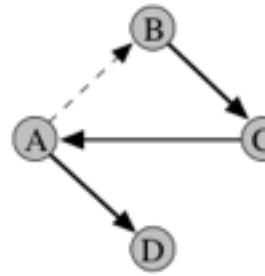
(b)



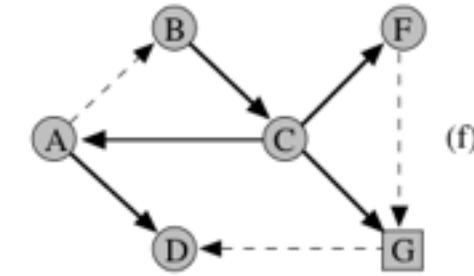
(c)



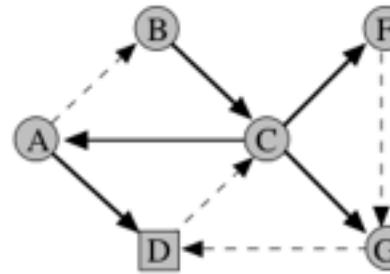
(d)



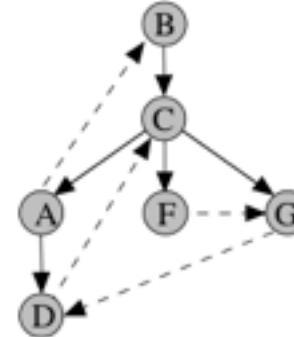
(e)



(f)



(g)



archi con estremi nello stesso livello: (F,G)

archi da un livello a quello immediatamente successivo: (G,D)

archi da un livello a uno precedente: (A,B)
e (D,C)

Visita in profondità

Visita in profondità

La visita procede in profondità sul grafo a partire dai vertici incontrati.

La frangia F è implementata come una **pila** (politica di accesso LIFO).

Visita in profondità

algoritmo visitaDFS (*vertice s*)

M: insieme vuoto di vertici

F: pila vuota di vertici

F.push(s)

aggiungi *s* ad *M*

while (*F* ≠ \emptyset) **do**

u = *F.pop()*

 visita *u*

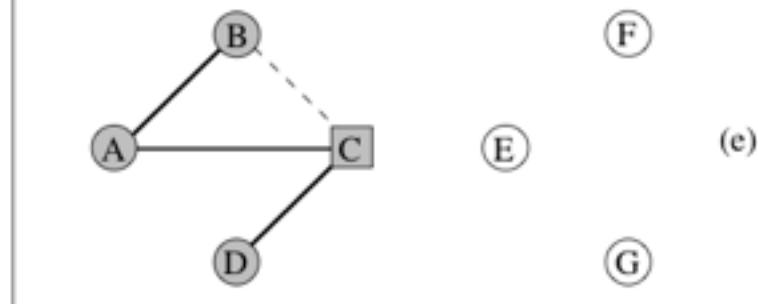
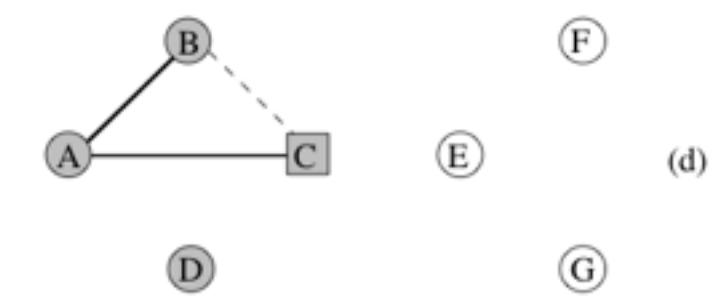
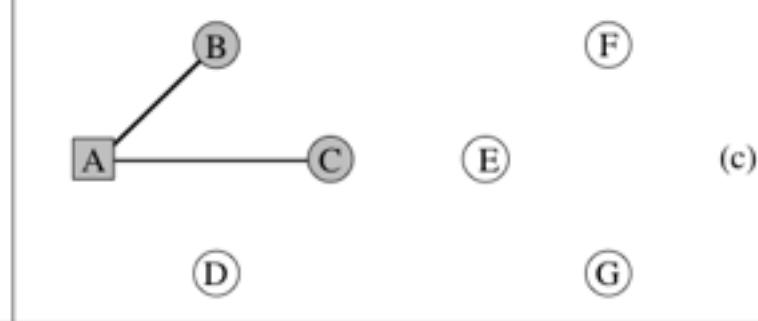
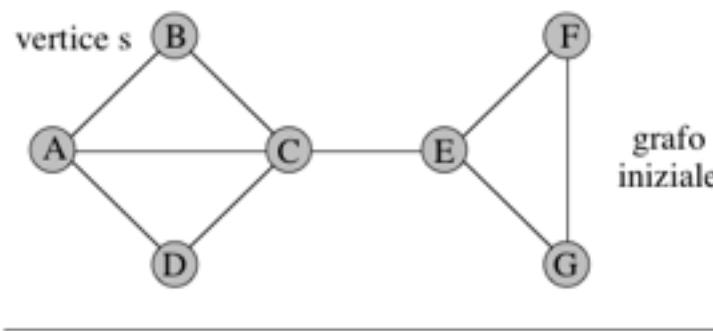
for each (arco (*u,v*) in *G*) **do**

if (*v* ∉ *M*) **then**

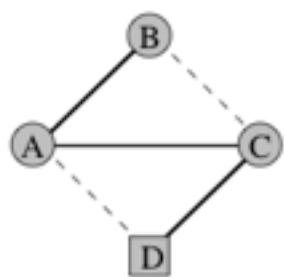
F.push(v)

 aggiungi *v* ad *M*

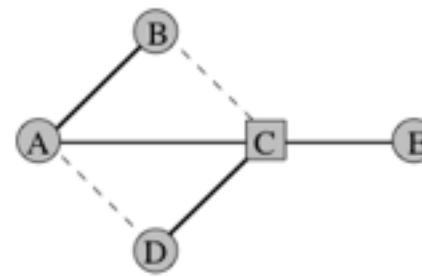
Esempio: grafo non orientato (1/2)



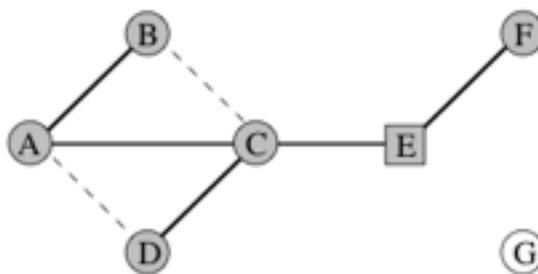
Esempio: grafo non orientato (2/2)



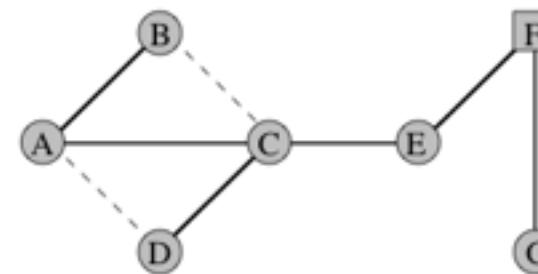
(f)



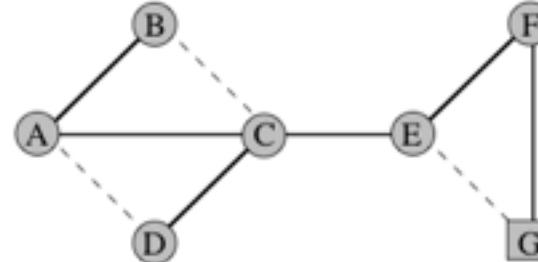
(g)



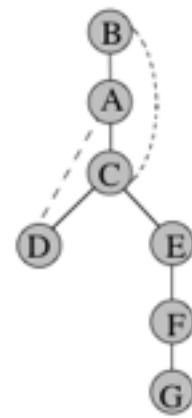
(h)



(i)



(j)



albero
DFS

Visita in profondità: versione ricorsiva

Quando effettuiamo la chiamata ricorsiva, dobbiamo passare l'insieme dei nodi marcati

Questo permette alla nuova chiamata di verificare se un nodo è già stato marcato

Visita in profondità: versione ricorsiva

procedura visitaDFSricorsiva (*vertice u, insieme di vertici M*)

visita *u*

aggiungi *u* ad *M*

for each (*arco (u,v) ∈ G*) **do**

if (*v* \notin *M*) **then**

visitaDFSricorsiva(*v,M*)

algoritmo visitaDFS(*vertice u*)

M: insieme vuoto di nodi

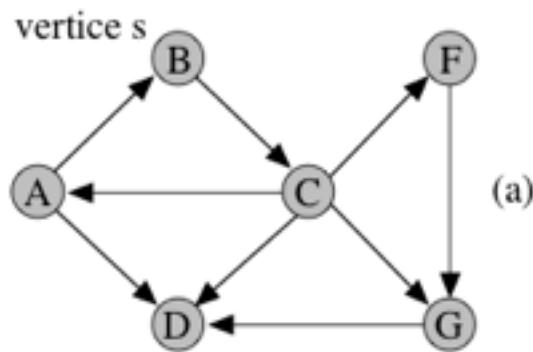
visitaDFSricorsiva(*u,M*)

Visita in profondità: grafi orientati

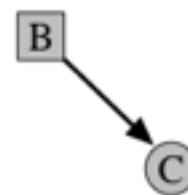
Anche la visita in profondità può essere applicata ai *grafi orientati*.

Anche in questo caso nell'operazione di scelta dei vicini di un vertice v si dovranno scegliere **solo** gli archi (v, w) *uscenti* da v

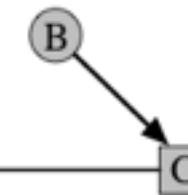
Esempio: grafo orientato (1/2)



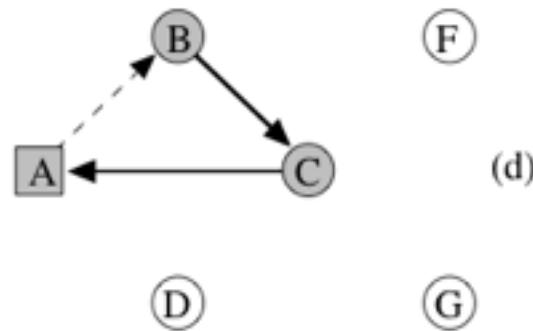
(a)



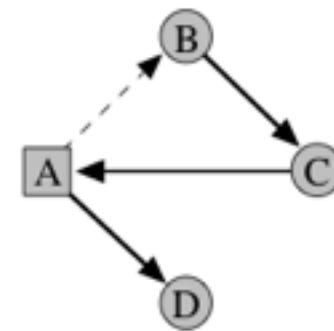
(b)



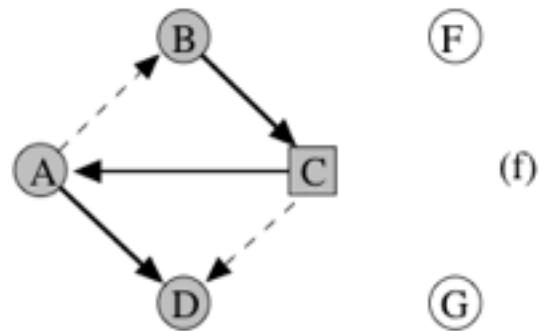
(c)



(d)

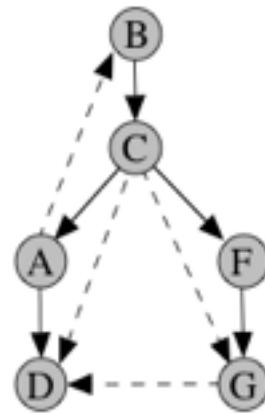
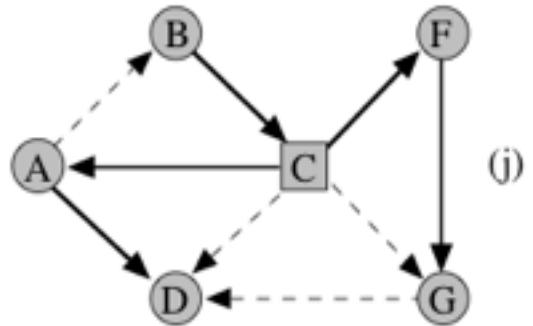
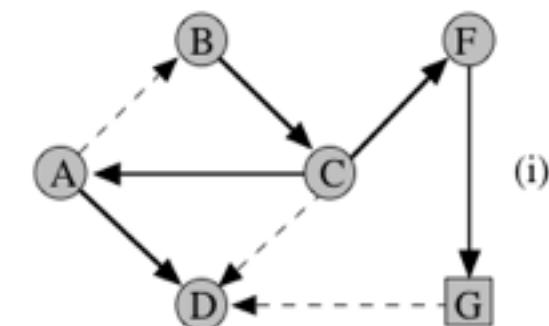
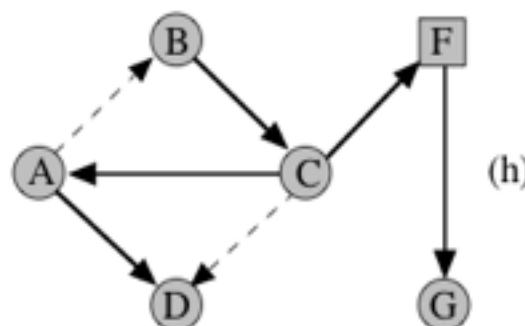
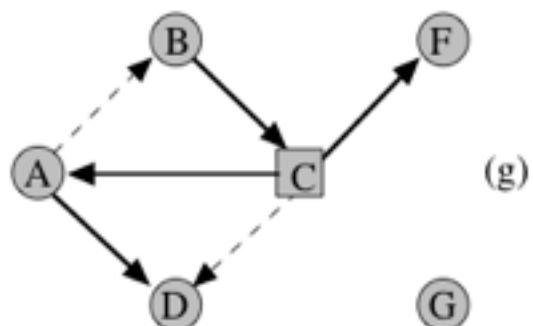


(e)



(f)

Esempio: grafo orientato (2/2)



La relazione "raggiungibile da"

Proprietà: La relazione di *raggiungibilità* tra vertici di un grafo orientato è una relazione di *equivalenza*.

- **Riflessività:** ogni nodo è *raggiungibile da* se stesso;
- **Simmetria:** se v è *raggiungibile da* u , allora u è *raggiungibile da* v
- **Transitività:** se v è *raggiungibile da* u e u è *raggiungibile da* w allora v è *raggiungibile da* w

Componenti connesse

Le **componenti connesse** di un grafo non orientato $G=(V,E)$ sono le *classi di equivalenza* dei suoi vertici rispetto alla relazione "raggiungibile da"

Ricordiamo che:

Un grafo *non orientato* $G = (V, E)$ si dice **connesso** se esiste un cammino tra ogni coppia di vertici in G .

Un grafo *non orientato* è **connesso** se e solo se ha una sola **componente connessa**.

Componenti connesse

Per verificare se un grafo **G** con **n** nodi è connesso, è sufficiente eseguirne una visita *memorizzando l'insieme M dei nodi marcati*

Se *M* contiene **n** nodi, allora *tutti i nodi sono stati raggiunti*, quindi **G** è connesso

Altrimenti, **G** non è connesso

Se un grafo non è connesso, per visitarne tutti i nodi è sufficiente eseguire la visita partendo prima da un nodo **s**, calcolando così *M*, poi da un nodo **s'**, non in *M*, calcolando un nuovo *M*, e così via

La relazione "fortemente connesso"

Un vertice x è **fortemente connesso** ad un vertice y se esiste un cammino (orientato) da x ad y ed un cammino (orientato) da y a x .

Proprietà: La relazione di *connettività forte* tra vertici in un grafo orientato è una relazione di *equivalenza (riflessiva, simmetrica e transitiva)*

Componenti fortemente connesse

Definizione: Sia $G = (V, E)$ un grafo orientato. Le componenti fortemente connesse di G sono le classi di equivalenza della relazione di connettività forte:

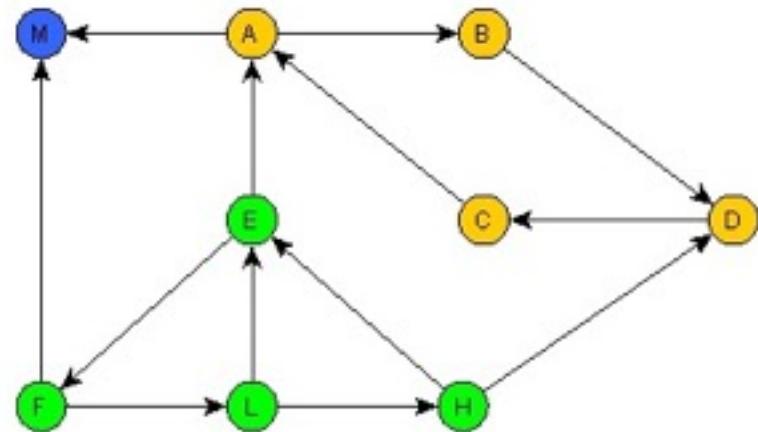
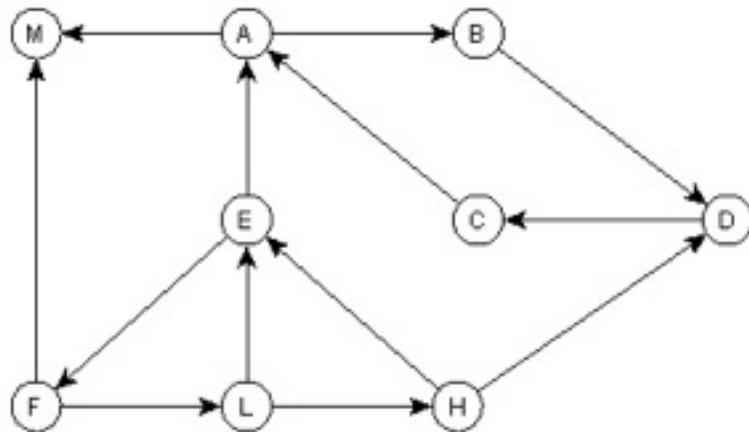
$[v] = \{ u \text{ tale che } u \text{ è un nodo di } G \text{ fortemente connesso a } v \}$

Dalla definizione di *componenti fortemente connesse* segue che un grafo G è **fortemente连通** se e solo se ha una sola *componente fortemente connessa*

Componenti fortemente connesse

Esempio:

notare che una componente fortemente connessa può essere formata anche da un solo nodo (nodo M)



Calcolare le componenti fortemente connesse

Per calcolare *la componente fortemente connessa* contenente il vertice x operiamo calcolando i seguenti due insiemi:

- I discendenti **D(x)**, ovvero tutti i nodi di G raggiungibili da x (per calcolarli eseguiamo una visita partendo da x)
- Gli antenati **A(x)**, ovvero tutti i nodi di G che raggiungono x (per calcolarli prima invertiamo la direzione di tutti gli archi in G e poi eseguiamo una visita partendo da x)

La *componente fortemente connessa* contenente x sarà data dai nodi che sono nell'intersezione tra **D(x)** e **A(x)**.

Per calcolare tutte le *componenti fortemente connesse* di G iteriamo il procedimento per tutti i nodi di G .

Riepilogo

- Concetto di grafo e terminologia
- Diverse strutture dati per rappresentare grafi nella memoria di un calcolatore
- L'utilizzo di una particolare rappresentazione può avere un impatto notevole sui tempi di esecuzione di un algoritmo su grafi (ad esempio, nella visita di un grafo)
- Algoritmo di visita generica e due casi particolari: visita in ampiezza e visita in profondità
- Componenti connesse e fortemente connesse di un grafo.

Algoritmi e Strutture Dati

Minimo albero ricoprente

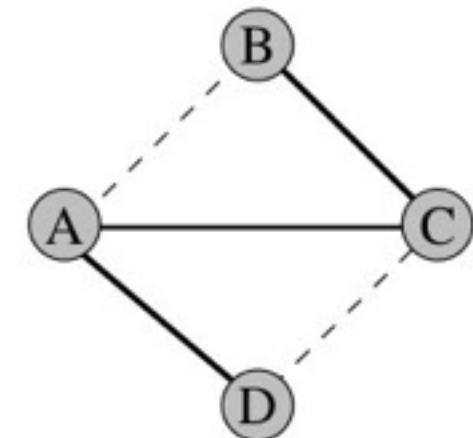
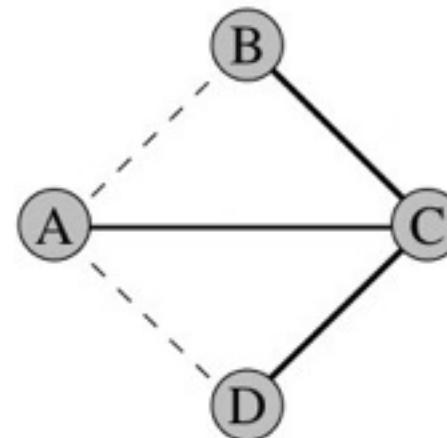
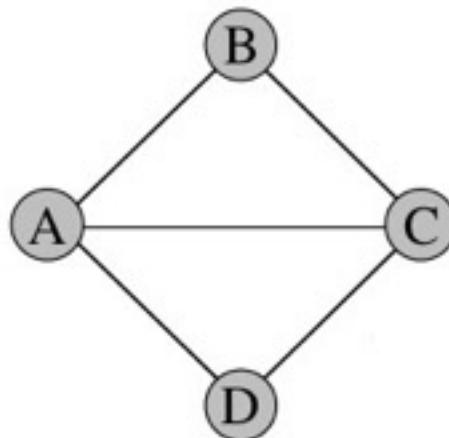
Fabio Patrizi

Albero ricoprente

Sia $G=(V,E)$ un grafo *non orientato* e *connesso*. Un **albero ricoprente** di G è un *sottografo* $T \subseteq G$ tale che:

- T è un **albero**;
- T contiene tutti i vertici di G .

Un grafo può avere più alberi ricoprenti.



Minimo albero ricoprente

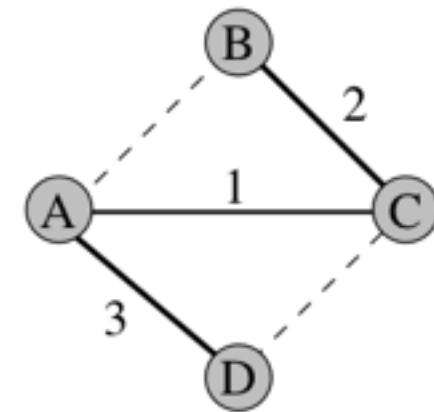
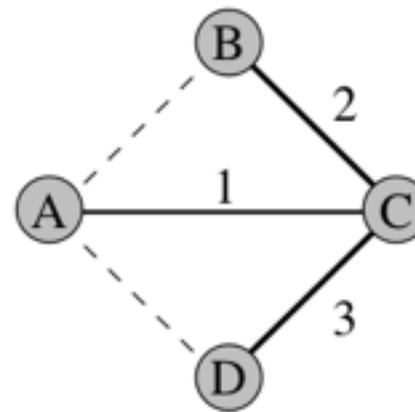
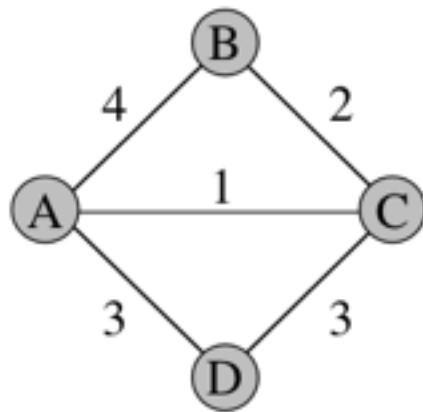
- Ad ogni arco $e \in E$ associamo un **costo** (o **peso**) $w(e)$.
- Definiamo il **costo di un albero ricoprente** T di un grafo G come la somma dei costi dei suoi archi:

$$w(T) = \sum_{e \in E} w(e)$$

Definizione: (minimo albero ricoprente) Sia $G=(V,E)$ un grafo *non orientato, connesso e pesato sugli archi*. Un **minimo albero ricoprente** di G è un albero ricoprente di G con costo *minimo*.

Esempi

Il minimo albero ricoprente non è necessariamente unico



Il problema

Il problema: Dato un grafo $G = (V, E)$ *non orientato, connesso e pesato sugli archi* trovare un **minimo albero ricoprente** di G

Problema di *ottimizzazione* molto studiato che si presenta in diversi contesti applicativi (esempio: *noleggio connessioni di rete*)

La tecnica algoritmica *Greedy*

- Adottiamo la tecnica algoritmica *greedy* (*golosa*)
- Tecnica spesso utilizzata per problemi di ottimizzazione
- Prevede la costruzione di una soluzione effettuando ad ogni passo la scelta che sembra più promettente, senza preoccuparsi dell'impatto sulle scelte future
- Scelte già effettuate non possono essere cambiate (*no backtracking*)

Un approccio *Greedy*

- Costruiamo il **minimo albero ricoprente** un arco alla volta, effettuando scelte *localmente* vantaggiose:
 - includere nella soluzione archi di **costo piccolo**
 - escludere dalla soluzione archi di **costo elevato**
- Processo di *colorazione*:
 - archi **blu**: inclusi nella soluzione
 - archi **rossi**: esclusi dalla soluzione

Tagli

Taglio:

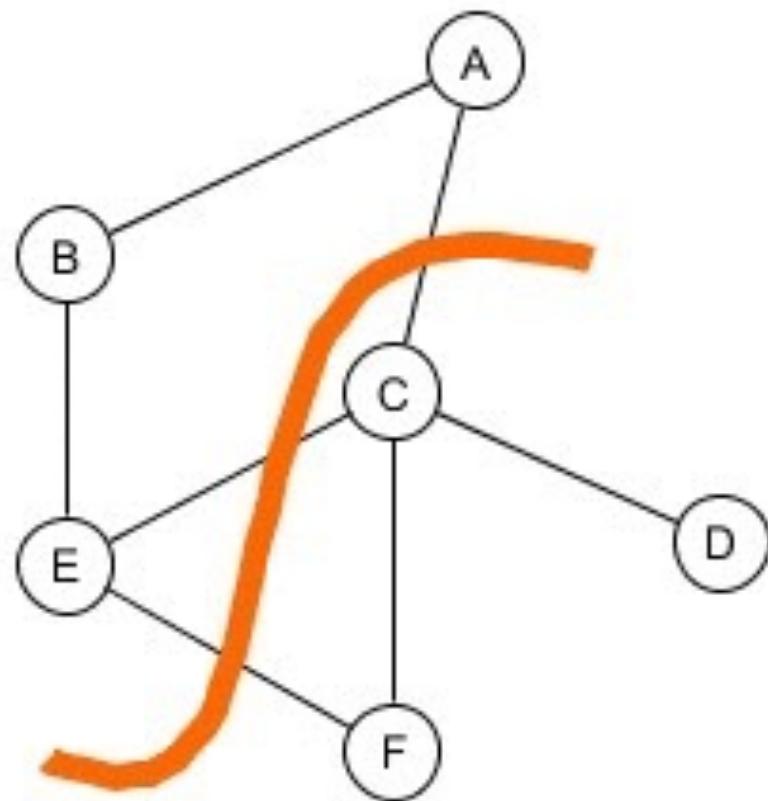
partizione dell'insieme di vertici V di G in due insiemi: X e $X' = V - X$.

Un arco $e = (u, v)$ attraversa il taglio (X, X') se $u \in X$ e $v \in X'$

Identifichiamo un **taglio** con *l'insieme di archi che lo attraversano*

Esempio: taglio

Taglio $\{(A,C), (E,C), (E,F)\}$



Regola del taglio (regola blu)

Scegli un taglio che non contiene archi blu. Tra tutti gli archi non colorati del taglio, sceglie uno di costo minimo e coloralo di blu

- Ogni albero ricoprente deve contenere almeno un arco del taglio
- E' conveniente includere quello di costo minimo

Regola del ciclo (regola rossa)

Scegli un ciclo che non contiene archi rossi. Tra tutti gli archi non colorati del ciclo, sceglie uno di costo massimo e coloralo di rosso

- Ogni albero ricoprente deve escludere almeno un arco del ciclo
- E' conveniente escludere quello di costo massimo

L'approccio greedy

- L'approccio greedy applica una delle due regole ad ogni passo, finché tutti gli archi sono colorati
- Si può dimostrare che, una volta applicato l'algoritmo, esiste un minimo albero ricoprente contenente tutti gli archi blu e nessun arco rosso
- Si può inoltre dimostrare che il metodo greedy colora tutti gli archi

Tempi di esecuzione

A seconda della scelta della regola da applicare e del taglio/ciclo usato ad ogni passo, si ottengono diversi algoritmi con diversi costi di esecuzione

Algoritmo di Kruskal

Strategia

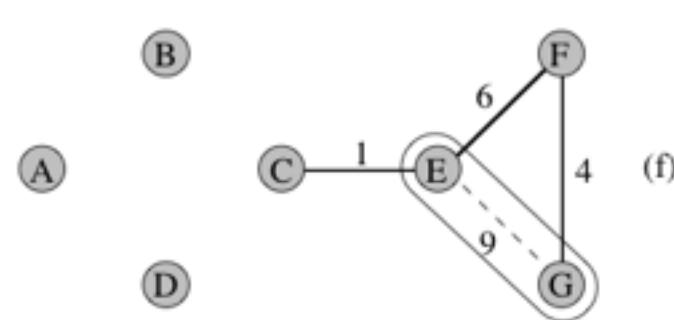
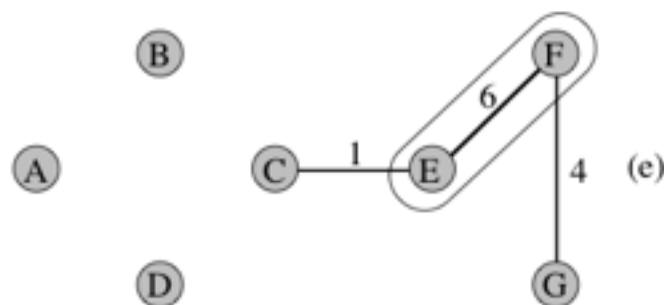
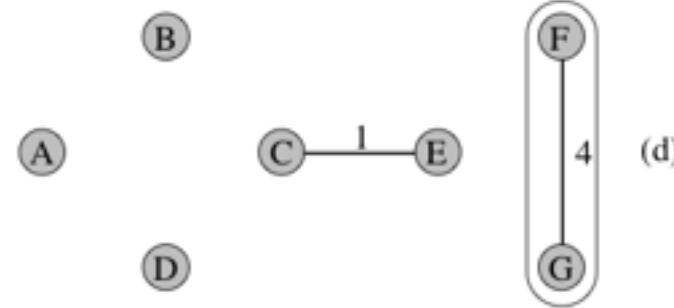
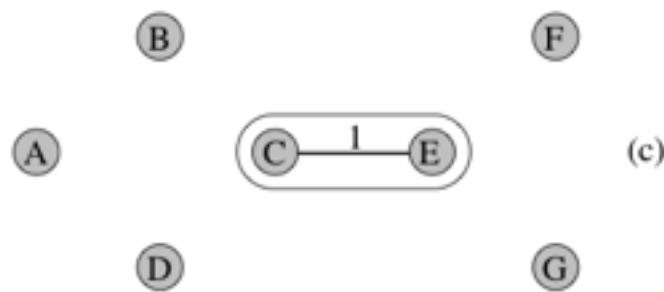
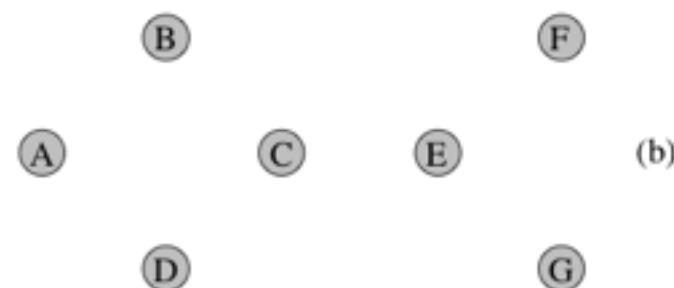
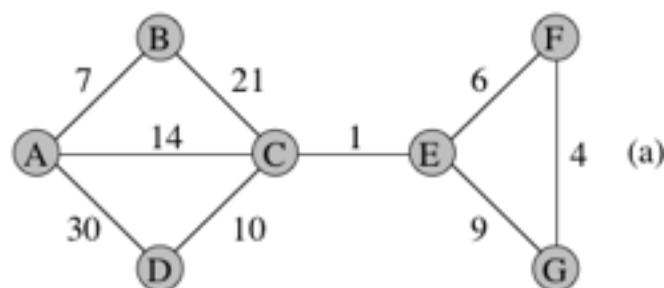
- Mantiene una foresta di alberi formati da soli archi blu (alberi blu), all'inizio tutti disgiunti.
- Per ogni arco, in ordine non decrescente di costo, applica il seguente passo: *se l'arco ha entrambi gli estremi nello stesso albero blu, applica la regola del ciclo e coloralo rosso, altrimenti applica la regola del taglio e coloralo blu*

Pseudocodice

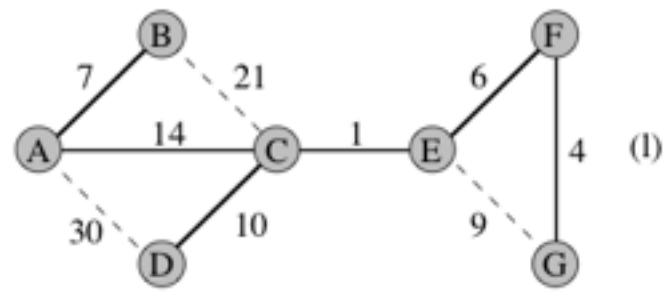
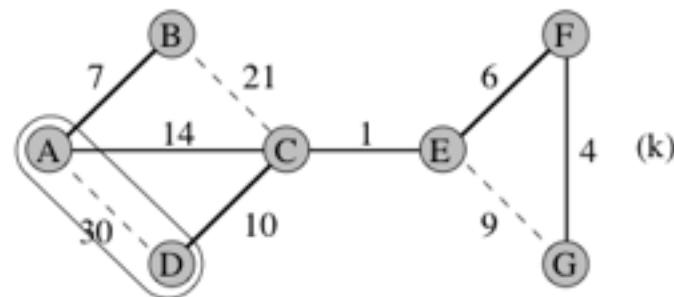
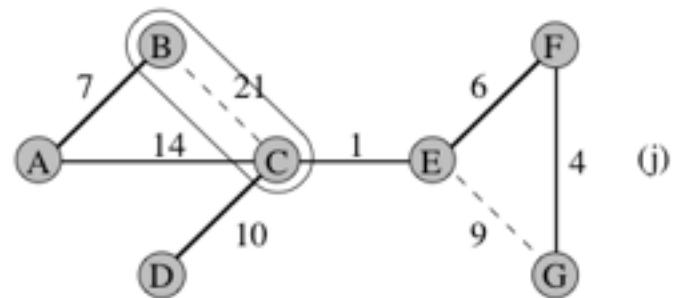
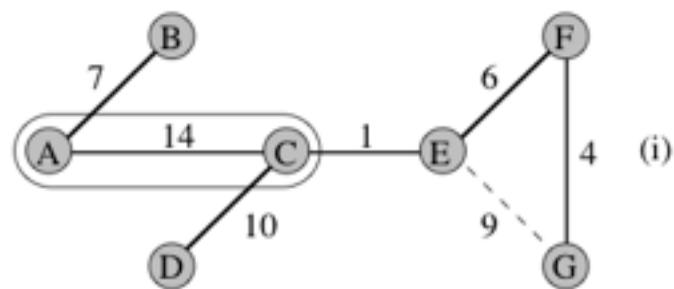
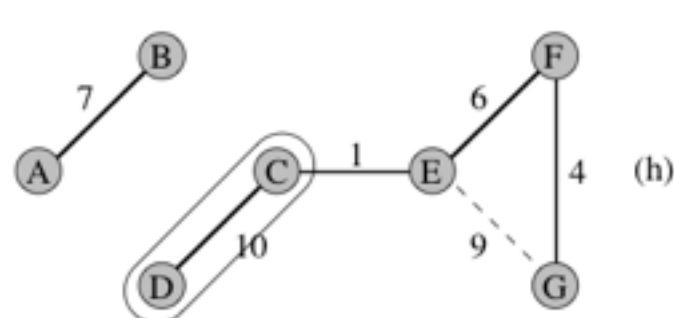
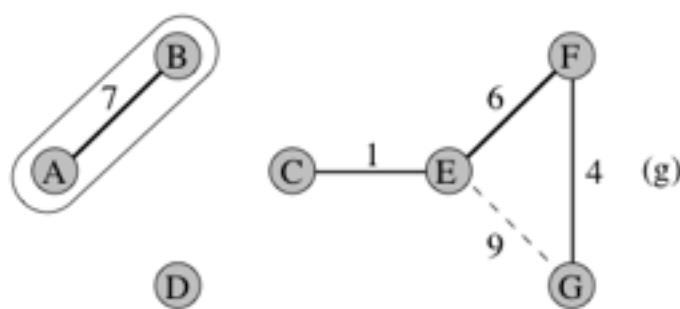
algoritmo Kruskal(grafo G) \rightarrow albero

1. ordina gli archi di G secondo costi non decrescenti;
2. $T \leftarrow$ albero vuoto;
3. **for each** (arco (x,y) di G in ordine non decrescente di costo) **do**
4. **if** (x e y non sono connessi in T) **then**
5. aggiungi l'arco (x,y) a T ;
6. **return** T

Esempio (1/2)



Esempio (2/2)



Il tempo di esecuzione dell'algoritmo di Kruskal è **$O(m \log n)$** nel caso peggiore

Dove **n** è il numero di nodi e **m** è il numero di archi

Algoritmo di Prim

Strategia

- Mantiene un unico albero blu T , che all'inizio consiste di un vertice arbitrario s .
- Applica per **n-1** volte il seguente passo:

scegli un arco di costo minimo incidente su uno dei nodi di T e coloralo di blu

Pseudocodice

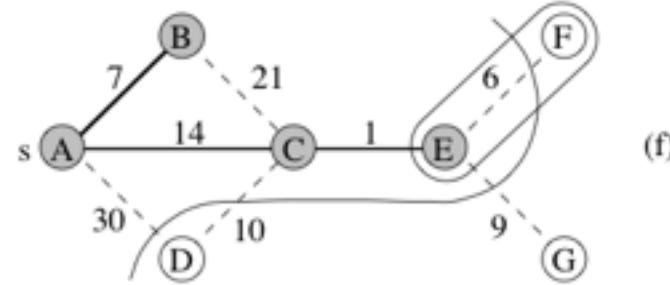
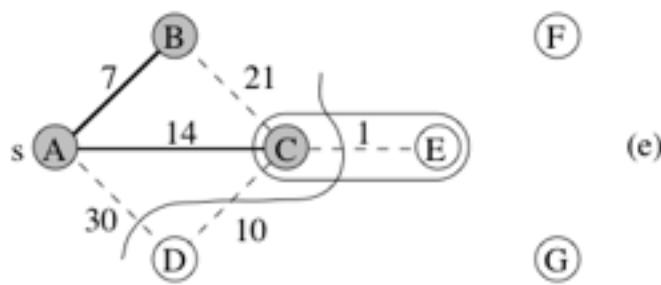
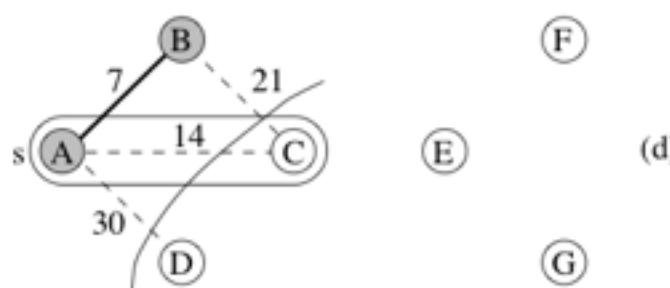
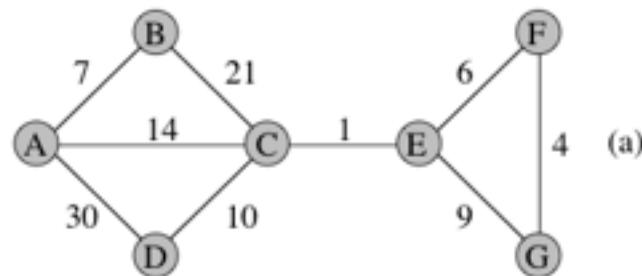
Definiamo **arco azzurro** un arco (x,y) tale che:

- $x \in T$
- $y \notin T$

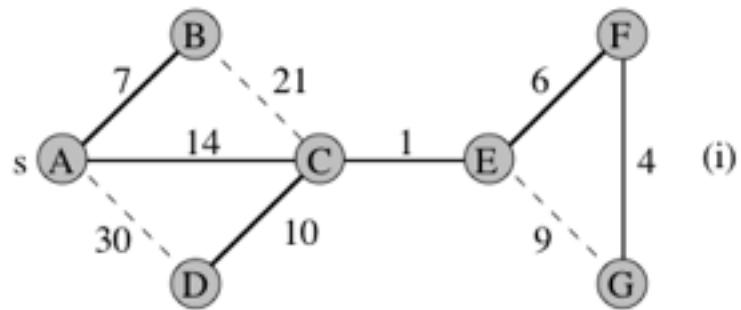
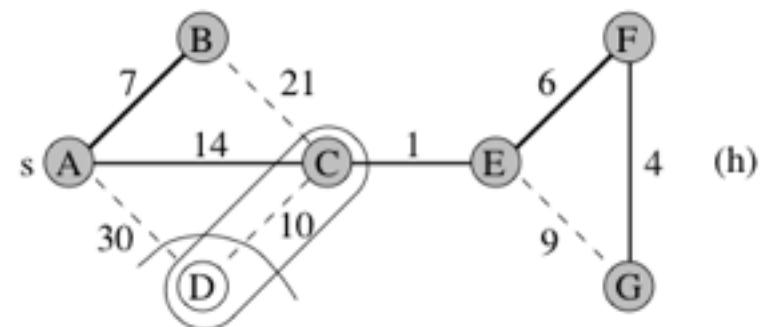
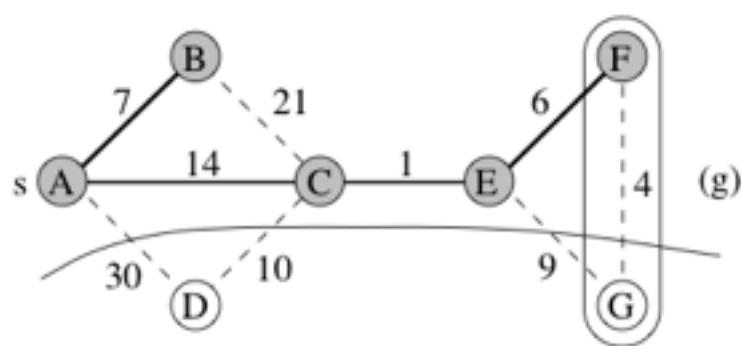
algoritmo Prim(grafo **G**) \rightarrow albero

1. scegli un nodo **s**;
2. $T \leftarrow$ albero formato dal solo nodo **s**;
3. **while** (numero di nodi di **T** < **n**) **do**
4. trova **l'arco azzurro** (x,y) con costo minimo;
5. aggiungi l'arco (x,y) a **T**;
6. **return** **T**

Esempio (1/2)



Esempio (2/2)



Il tempo di esecuzione dell'algoritmo di Prim è $O(m + n \log n)$ nel caso peggiore

Algoritmo di Borůvka

Strategia

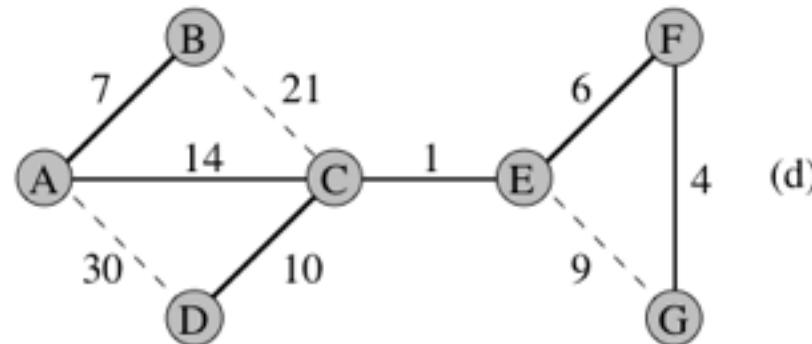
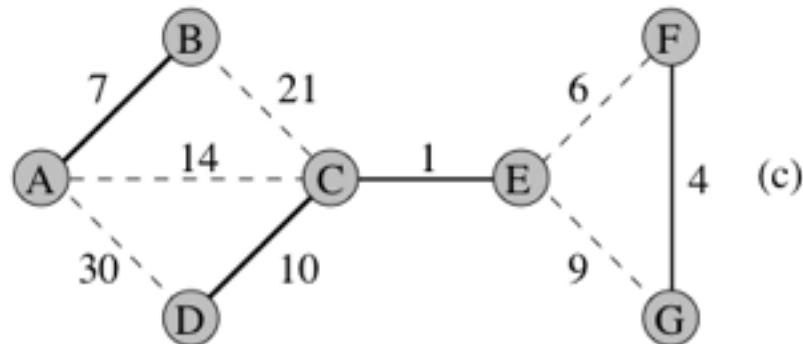
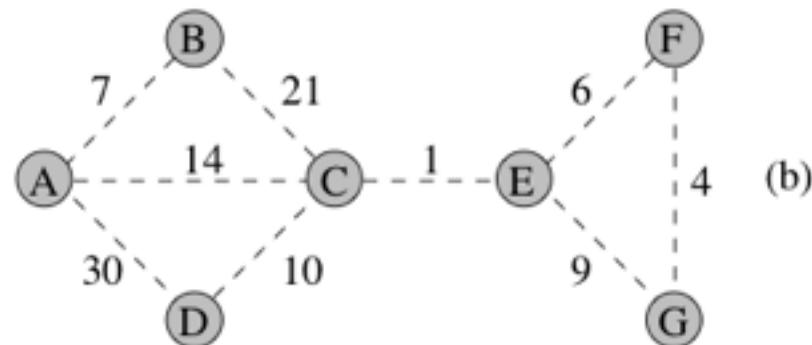
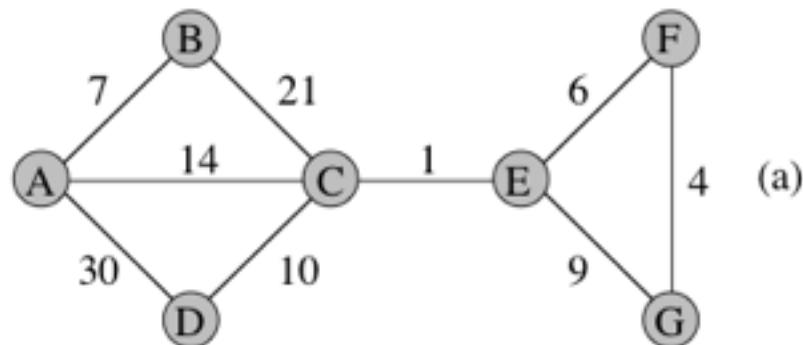
- Mantiene una foresta di alberi blu, all'inizio tutti disgiunti
- Ad ogni passo, applica la seguente regola:
"per ogni albero blu T nella foresta, scegli un arco di costo minimo incidente su T e coloralo blu (applica la regola del taglio)"
- (Assumiamo che i costi degli archi siano tutti distinti)

Pseudocodice

algoritmo Boruvka(grafo $G = (V,E)$) \rightarrow albero

1. $T \leftarrow$ albero vuoto;
3. **while** (T ha più di una componenti connessa) **do**
4. per ogni componente C di T scegli l'arco (u,v) di G
 con costo minimo tale che $u \in C$ e $v \notin C$;
5. aggiungi l'arco (u,v) a T ;
6. **return** T

Esempio



Analisi

Il tempo di esecuzione dell'algoritmo di Boruvka è $O(m \log n)$ nel caso peggiore

Riepilogo

- Paradigma generale per il calcolo di minimi alberi ricoprenti
- Applicazione della tecnica golosa
- Tre algoritmi specifici ottenuti dal paradigma generale: Prim, Kruskal e Boruvka

Algoritmi e Strutture Dati

Cammini minimi

Fabio Patrizi

Cammino minimo: definizione

Sia $G=(V,E)$ un grafo orientato con funzione di costo $w: E \rightarrow \text{Real}$ sugli archi. Il costo $w(\pi)$ di un cammino $\pi = \langle v_0, v_1, v_2, \dots, v_k \rangle$ è dato da:

$$w(\pi) = \sum_{i=1}^k w(v_{i-1}, v_i)$$

Un **cammino minimo** tra una coppia di vertici x e y è un cammino di costo minore o uguale a quello di ogni altro cammino tra gli stessi vertici.

Proprietà dei cammini minimi (1/3)

Sottostruttura ottima:

Sia $G=(V,E)$ un grafo orientato con funzione di costo $w: E \rightarrow \text{Real}$ sugli archi e sia π un cammino minimo in G . Allora ogni sottocammino di π è esso stesso un cammino minimo in G (tra i nodi di partenza e destinazione)

Proprietà dei cammini minimi (2/3)

Supponiamo che i costi degli archi possano essere negativi e che vi siano in un grafo cicli con costo minore di zero.

Proprietà:

In un grafo con cicli negativi, se due vertici x e y appartengono a un ciclo di costo negativo allora non esiste nessun cammino minimo finito tra di essi

Proprietà dei cammini minimi (3/3)

Se G non contiene cicli negativi, tra ogni coppia di vertici connessi in G esiste sempre un cammino minimo semplice

Distanza fra vertici

- La distanza d_{xy} tra due vertici x e y è il costo di **un cammino minimo** tra da x a y , o $+\infty$ se i due vertici non sono connessi:

$$d_{xy} = \begin{cases} w(\pi_{xy}), & \text{Se esiste cammino tra } x \text{ e } y \text{ in } G \\ +\infty, & \text{altrimenti} \end{cases}$$

Distanza fra vertici (proprietà)

- **Disegualanza triangolare**: per ogni x, y e z

$$d_{xz} \leq d_{xy} + d_{yz}$$

- **Condizione di Bellman**: per ogni arco (u,v) e per ogni vertice s

$$d_{su} + w(u, v) \geq d_{sv}$$

Cammini minimi e distanze

Un arco (u,v) appartiene ad un **cammino minimo** a partire da un vertice s se e solo se la condizione di Bellman vale con l'uguaglianza, ovvero se u è raggiungibile da s e $d_{su} + w(u,v) = d_{sv}$

Calcolare cammini minimi dalle distanze

algoritmo cammino(grafo G , distanze d , vertice x , vertice y)

1. $\pi \leftarrow \langle y \rangle$; $v \leftarrow y$
2. **while** ($v \neq x$) **do**
 3. **for each** (arco (u, v) in G) **do**
 4. **if** ($d_{xu} + w(u, v) = d_{xv}$) **then**
 5. aggiungi u come primo vertice in π
 6. $v \leftarrow u$
 7. **break**
8. **return** π

Albero dei cammini minimi

I cammini minimi da un vertice s a tutti gli altri vertici del grafo possono essere rappresentati tramite un albero radicato in s . Tale albero è detto **albero dei cammini minimi**

Varianti del problema dei cammini minimi

- Problema dei **cammini minimi fra tutte le coppie**: calcolo delle distanze tra ogni coppia di vertici nel grafo
- Problema dei **cammini minimi a sorgente singola**: calcolo delle distanze a partire da un dato vertice chiamato **sorgente**
- Problema dei **cammini minimi fra singola coppia**: calcolo della distanza tra due vertici prefissati

Tecnica del rilassamento

Gli algoritmi che vedremo per calcolare le distanze per identificare i cammini minimi partono da stime D_{xy} per eccesso delle distanze d_{xy} , ovvero: $D_{xy} \geq d_{xy}$ e le aggiornano progressivamente decrementandole fino a renderle esatte ($D_{xy} = d_{xy}$).

Aggiornamento delle stime basato sul seguente passo di rilassamento:

(RILASSAMENTO) **if** $(D_{xv} + w(\pi_{vy}) < D_{xy})$
 then $D_{xy} \leftarrow D_{xv} + w(\pi_{vy})$

Algoritmo generico per il calcolo delle distanze

algoritmo *distanzeGenerico*(grafo G) \rightarrow *distanze*

inizializza D in modo che $D_{xy} \geq d_{xy}$ per ogni coppia di interesse (x, y)

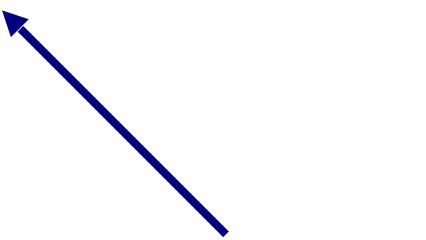
while (esiste (x, y) fra le coppie di interesse tale che $D_{xy} > d_{xy}$) **do**

 considera un vertice v e un cammino π_{vy}

if ($D_{xv} + w(\pi_{vy}) < D_{xy}$) **then** $D_{xy} \leftarrow D_{xv} + w(\pi_{vy})$

return D

Gli algoritmi per il calcolo delle distanze che vedremo differiscono principalmente su come scegliere il vertice v ed il cammino π_{vy}



Algoritmo di Bellman e Ford

(per cammini minimi a sorgente singola)

Ordine di rilassamento

Sia $\pi = \langle s, v_1, v_2, \dots, v_k \rangle$ un cammino minimo.

Se fossimo in grado di eseguire i passi di rilassamento nell'ordine seguente:

1. $D_{sv_1} \leftarrow D_{ss} + w(s, v_1)$
2. $D_{sv_2} \leftarrow D_{sv_1} + w(v_1, v_2)$
3. $D_{sv_3} \leftarrow D_{sv_2} + w(v_2, v_3)$
- ⋮ ⋮
- $k.$ $D_{sv_k} \leftarrow D_{sv_{k-1}} + w(v_{k-1}, v_k)$

in k passi avremmo la soluzione. Purtroppo non conosciamo l'ordine giusto, essendo π ignoto

Approccio di Bellman e Ford

- L'algoritmo di Bellman e Ford si basa sulla seguente idea:
 - Effettuando una prima passata "rilassando" tutti gli archi di G, otteniamo sicuramente $D_{sv1} = d_{sv1}$
 - Ripetendo la passata su tutti gli archi otterremo $D_{sv2} = d_{sv2}$
 - Con **n** passate arriveremo ad avere tutte le distanze

Pseudocodice

algoritmo BellmanFord(*grafo* G , *vertice* s) \rightarrow *distanze*
inizializza D tale che $D_{sv} = +\infty$ per $v \neq s$, e $D_{ss} = 0$
for $i = 1$ **to** n **do**
 for each ($(u, v) \in E$) **do**
 if ($D_{su} + w(u, v) < D_{sv}$) **then** $D_{sv} \leftarrow D_{su} + w(u, v)$
return D

Approccio di Bellman e Ford

- L'algoritmo cicla per **n** volte, dove n è il numero di vertici del grafo **G**.
- In ogni ciclo effettua il rilassamento di tutti gli **m** archi.



Tempo di esecuzione: **O(n m)**

- Dopo la j-esima passata, i primi j rilassamenti corretti sono stati certamente eseguiti.
- Esegue però molti rilassamenti inutili!

Algoritmo per grafi diretti aciclici (per cammini minimi a sorgente singola)

Algoritmo per grafi diretti aciclici

- L'algoritmo di Bellman e Ford esegue un grande numero di operazioni di rilassamento inutili.
- Eseguire i passi di rilassamento del giusto ordine è cruciale per evitare di fare operazioni inutili.
- Il prossimo algoritmo sfrutterà alcune proprietà del grafo in modo da rilassare ciascun arco esattamente una volta.

Ordinamento topologico (1/2)

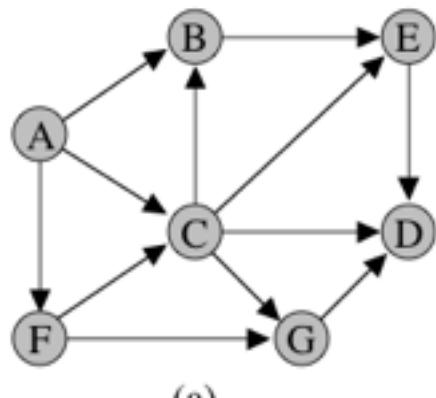
In un grafo aciclico, se esiste un cammino da **u** a **v** allora **non esiste** un cammino da **v** ad **u**. Questo ci permette di definire un ordinamento dei vertici, tale che **u** precede **v** nell'ordinamento se esiste un cammino da **u** a **v**.

Tale ordinamento è detto **ordinamento topologico**

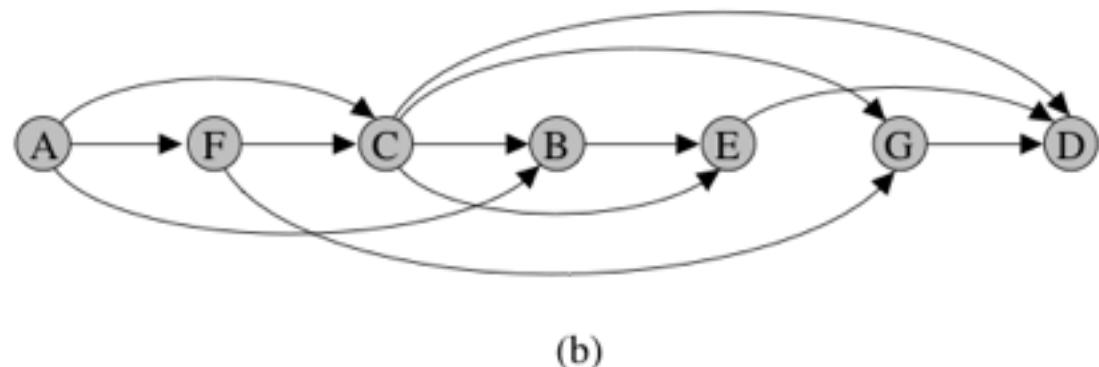
Ordinamento topologico (2/2)

Funzione $\sigma: V \rightarrow \{1, \dots n\}$ tale che $\sigma(u) < \sigma(v)$
se esiste un cammino da u a v in G

L'ordinamento topologico non è sempre unico.



(a)



(b)

(a) Esempio di grafo aciclico; (b) Ordinamento topologico del grafo (a).

Calcolo di un ordinamento topologico (1/2)

IDEA:

1. scelgo un nodo u senza archi entranti, lo cancello dal grafo e pongo $ord(u) = 1$.
2. Ripeto il ragionamento: scelgo un altro vertice v senza archi entranti, lo cancello dal grafo e pongo $ord(v) = 2$
3. Ripeto finché non ho eliminato tutti i nodi

Calcolo di un ordinamento topologico (2/2)

```
algoritmo ordinamentoTopologico(grafo G) → lista
     $\widehat{G} \leftarrow G$ 
    ord ← lista vuota di vertici
    while ( esiste un vertice u senza archi entranti in  $\widehat{G}$  ) do
        appendi u come ultimo elemento di ord
        rimuovi da  $\widehat{G}$  il vertice u e tutti i suoi archi uscenti
    if (  $\widehat{G}$  non è diventato vuoto ) then errore il grafo G non è aciclico
    return ord
```

Tempo di esecuzione: **O(n+m)**

Cammini minimi in grafi aciclici: idea

Il prossimo algoritmo è una variante dell'algoritmo di Bellman e Ford per soli grafi aciclici in cui le operazioni di rilassamento vengono effettuate secondo un ordinamento topologico:

Vengono prima rilassati tutti gli archi (u,v) con $ord(u) = 1$ nell'ordinamento topologico ord , poi quelli con $ord(u) = 2$ e così via.

In questo modo si evita di effettuare rilassamenti inutili: ogni arco viene rilassato esattamente una volta.

Cammini minimi in grafi aciclici: pseudocodice

algoritmo `distanzeAciclico`(grafo G , vertice s) \rightarrow distanze
inizializza D tale che $D_{sv} = +\infty$ per $v \neq s$, e $D_{ss} = 0$
 $ord \leftarrow$ ordinamentoTopologico(G)
for $i = 1$ **to** n **do**
 sia u l' i -esimo vertice nell'ordinamento topologico ord
 for each ($(u, v) \in E$) **do**
 if ($D_{su} + w(u, v) < D_{sv}$) **then** $D_{sv} \leftarrow D_{su} + w(u, v)$
return D

Tempo di esecuzione: **O(n+m)**

Algoritmo di Dijkstra

(per cammini minimi a sorgente singola in grafi con costi non negativi)

Estendere l'albero dei cammini minimi

Negli anni '50 E.W. Dijkstra scoprì la seguente proprietà:

Se T è un albero dei cammini minimi radicato in s che non include tutti i vertici raggiungibili da s , l'arco (u,v) tale che $u \in T$ e $v \notin T$ che minimizza la quantità $d_{su} + w(u,v)$ appartiene a un cammino minimo da s a v

Tale proprietà ci fornisce un semplice metodo per costruire un albero dei cammini minimi che include tutti i vertici raggiungibili da s

Approccio di Dijkstra

IDEA:

Applichiamo il seguente passo fino alla costruzione dell'albero T:

Scegli un arco (u,v) con $u \in T$ e $v \notin T$ che minimizza la quantità $D_{su} + w(u,v)$, effettua il passo di rilassamento $D_{sv} \leftarrow D_{su} + w(u,v)$, ed aggiungilo a T

Pseudocodice

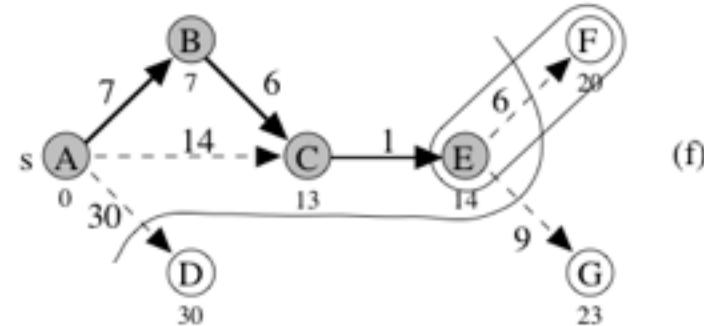
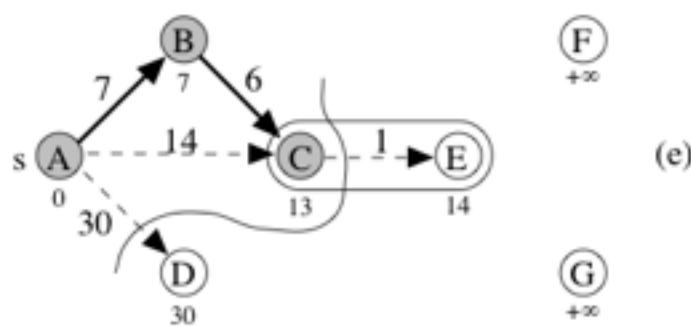
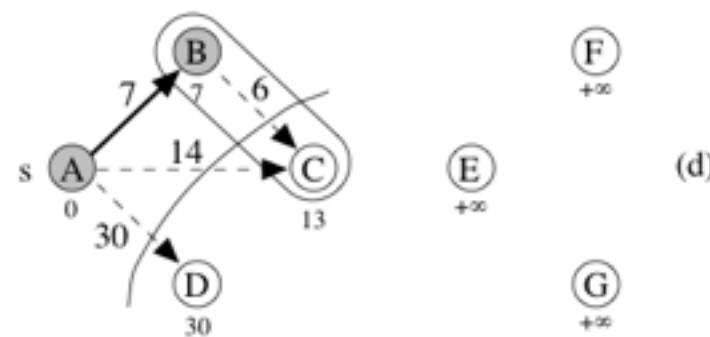
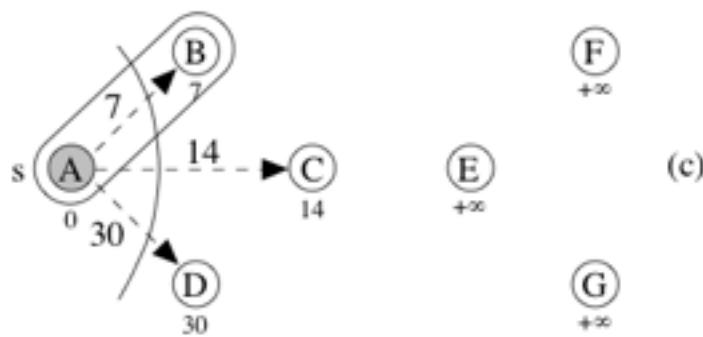
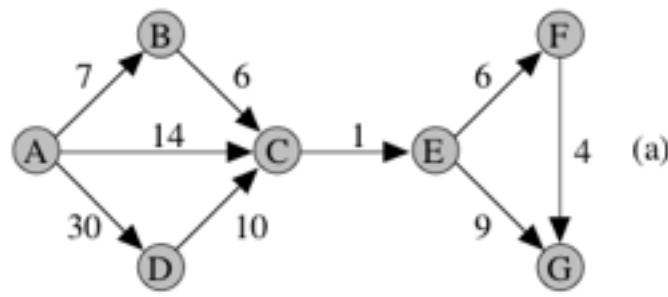
algoritmo DijkstraGenerico(*grafo G, vertice s*) \rightarrow *albero*

1. inizializza D tale che $D_{sv} = +\infty$ per $v \neq s$, e $D_{ss} = 0$
2. $T \leftarrow$ albero formato dal solo nodo s
3. **while** (T ha meno di n nodi) **do**
4. trova l'arco (u, v) incidente su T con $D_{su} + w(u, v)$ minimo
5. $D_{sv} \leftarrow D_{su} + w(u, v)$ {rilassamento}
6. rendi u padre di v in T
7. **return** T

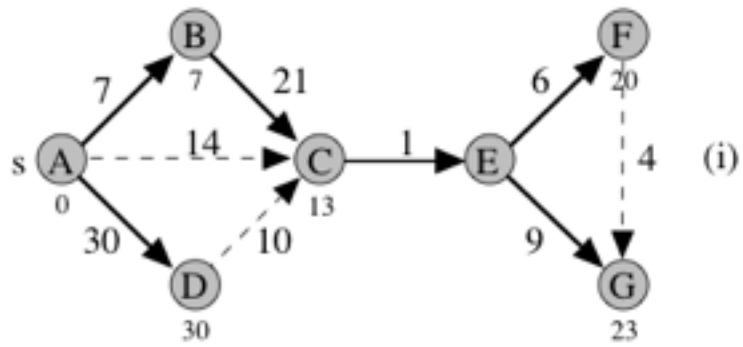
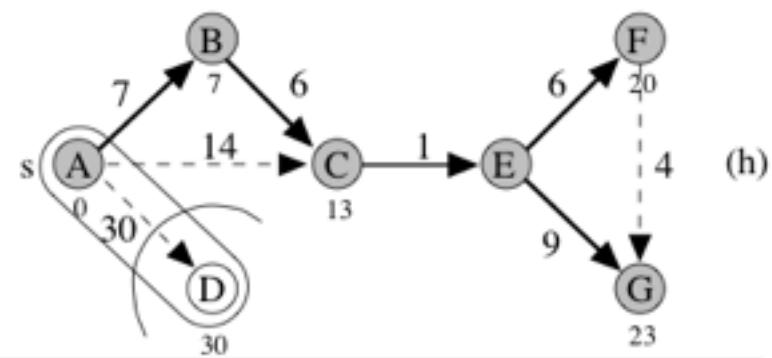
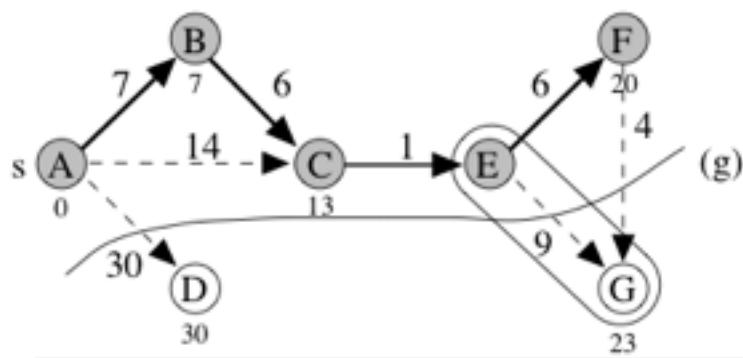
Si noti la somiglianza con l'algoritmo di Prim per il calcolo di un minimo albero ricoprente: come l'algoritmo di Prim anche l'algoritmo di Dijkstra si basa sulla tecnica *golosa*.

È possibile realizzare una versione dell'algoritmo che calcola T in $O(m + n \log n)$

Esempio (1/2)



Esempio (2/2)



Riepilogo

- Algoritmi classici per il calcolo di distanze (e quindi di cammini minimi), basati sulla tecnica del rilassamento:
 - Bellman e Ford: cammini minimi a sorgente singola, grafi diretti senza cicli negativi, tempo $O(nm)$
 - Grafi diretti aciclici: cammini minimi a sorgente singola in tempo $O(n+m)$
 - Dijkstra: cammini minimi a sorgente singola, grafi diretti senza pesi negativi, tempo $O(m+n \log n)$