



UNIVERSITÀ
DEGLI STUDI
DI BERGAMO

Dipartimento
di Ingegneria Gestionale,
dell'Informazione e della Produzione



Introduzione alla progettazione di algoritmi

PROGETTAZIONE, ALGORITMI E
COMPUTABILITÀ
(38090-MOD1)

Corso di laurea
Magistrale in
Ingegneria
Informatica

RELATORE
Prof.ssa Patrizia
Scandurra

SEDE
DIGIP

Algoritmi e problemi computazionali

Gli algoritmi forniscono la **strategia risolutiva** per giungere alla soluzione di un dato problema computazionale (o di calcolo)

Classificazione dei problemi:

- **problem non decidibili**
 - es. l'*Halting program* [Turing, 1937]
// restituisce true se l'algoritmo A su input finito d termina, false altrimenti
boolean halts(A, d)
- **problem decidibili**
 - **problem trattabili** (costo polinomiale)
 - **problem intrattabili** (costo esponenziale)

Algoritmi come soluzioni di problemi computazionali

- Esempio 1 (**Ordinamento**): Dati i risultati di una prova di esame, disporre i nomi degli studenti in ordine decrescente di voto
- Esempio 2 (**Calendario**): Calcolare il giorno della settimana corrispondente ad una certa data specificata da giorno, mese e anno.
- Esempio 3 (**Duplicati**): Data una lista di prenotazioni agli esami, verificare se vi sono studenti che si sono prenotati più di una volta
- Esempio 4 (**Ricerca**): ricercare i dati di un dato studente da una base di dati e/o collezione di record studenti, data la matricola come chiave di ricerca

Definizione di algoritmo

Una sequenza **finita** di operazioni **non ambigue** ed **effettivamente calcolabili** che, una volta eseguite, **producono un risultato** in una **quantità finita di tempo**

- Esempio: **algoritmo preparaCaffè**



algoritmo preparaCaffè

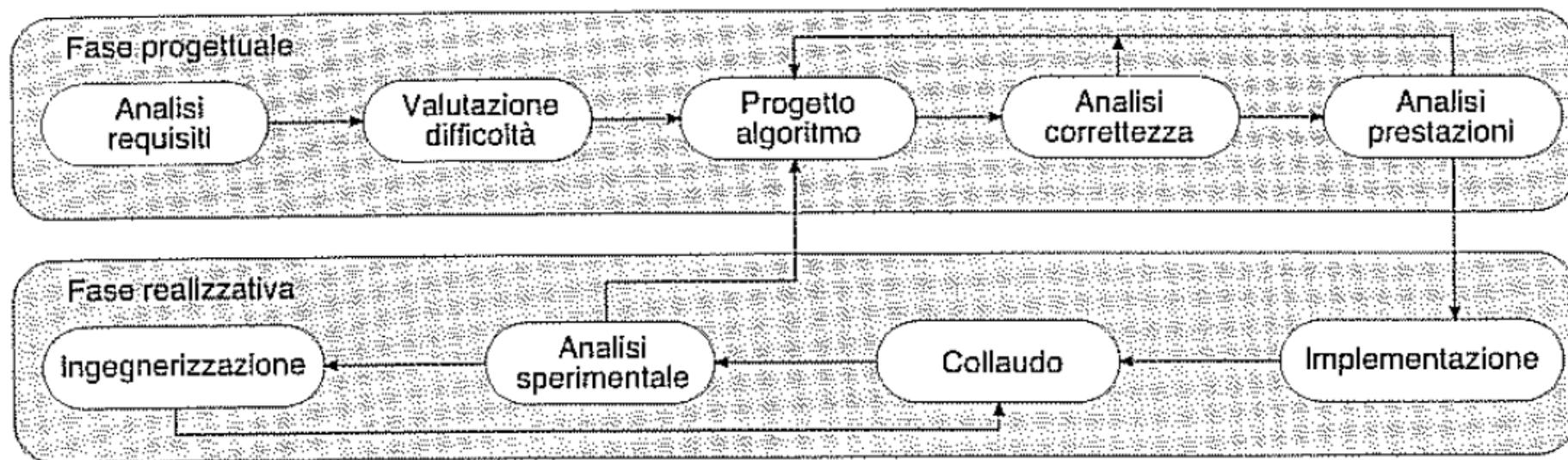
1. Svita la caffettiera.
2. Riempি d'acqua il serbatoio della caffettiera.
3. Inserisci il filtro.
4. Riempি il filtro con la polvere di caffè.
5. Avvita la parte superiore della caffettiera.
6. Metti la caffettiera, così predisposta, su un fornello acceso.
7. Spegni il fornello quando il caffè è pronto.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

Proprietà fondamentali di un algoritmo

- **Finitezza:** la sequenza di istruzioni deve essere finita (finitezza)
- **Non ambiguità:** le istruzioni devono essere espresse in modo non ambiguo
- **Realizzabilità:** le istruzioni devono essere eseguibili materialmente
- **Efficacia:** la sequenza di istruzioni deve portare ad un risultato
- **Efficienza:** È inoltre importante valutare le **risorse utilizzate** (tempo, memoria, ...) perché un consumo eccessivo delle stesse può pregiudicare la possibilità stessa di utilizzo di un algoritmo

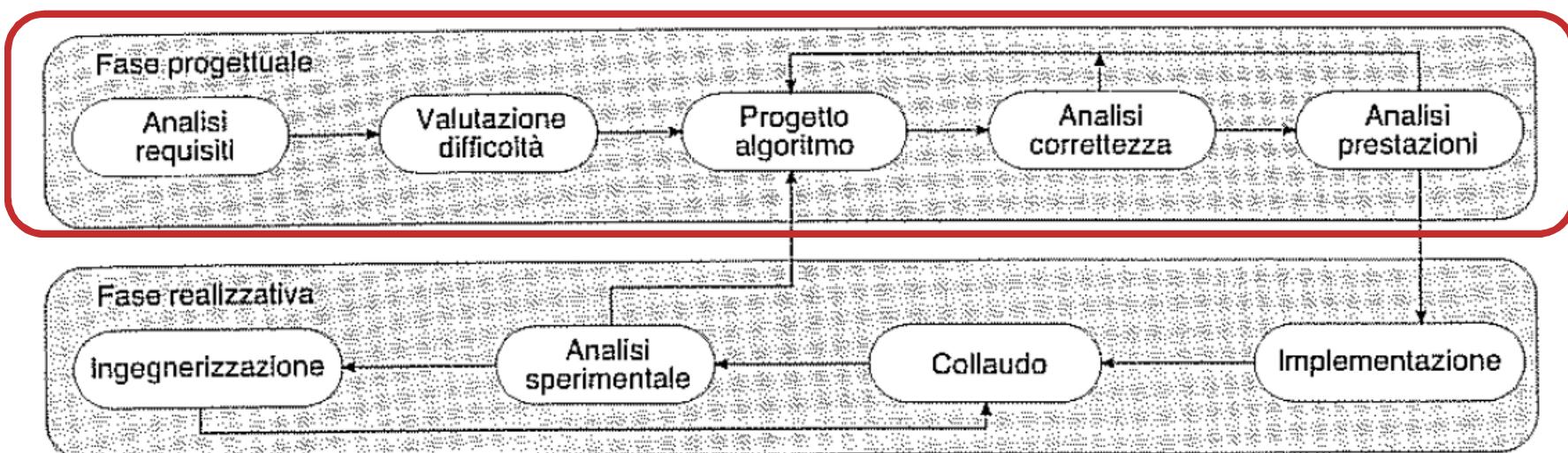
Ciclo di sviluppo di codice algoritmico

Modello ciclico a due fasi: **fase progettuale** e **fase realizzativa**

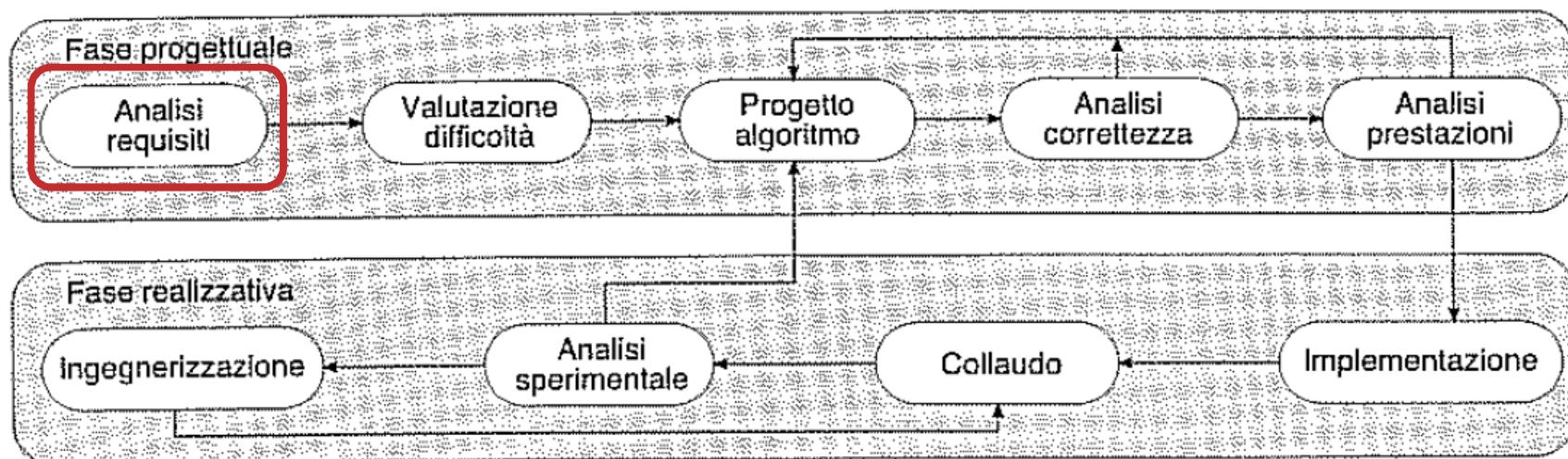


• Fase progettuale

design in piccolo!



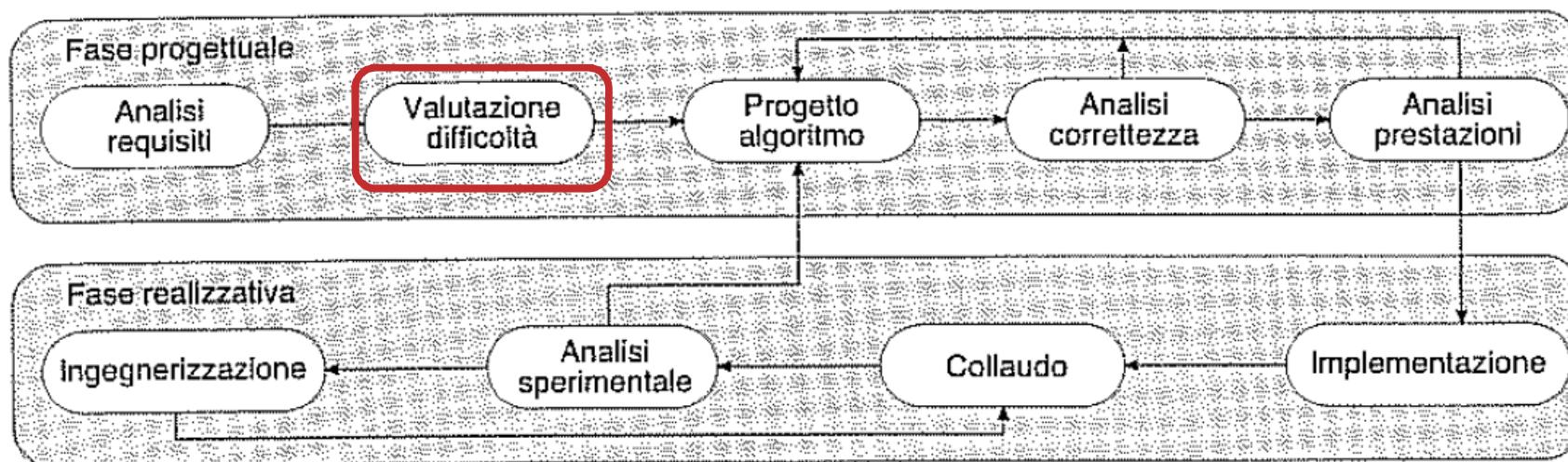
• Fase progettuale



Analisi dei requisiti

- Definire il **problema di calcolo** che si intende risolvere identificando:
 - i **requisiti dei dati in ingresso**
 - i **requisiti dei dati in uscita** prodotti dall'algoritmo
- Identificare una **metodologia di progettazione**:
 - Esempio: un problema complesso può essere scomposto in un certo numero di sotto-problemi risolvibili separatamente
 - L'approccio *divide-et-impera*

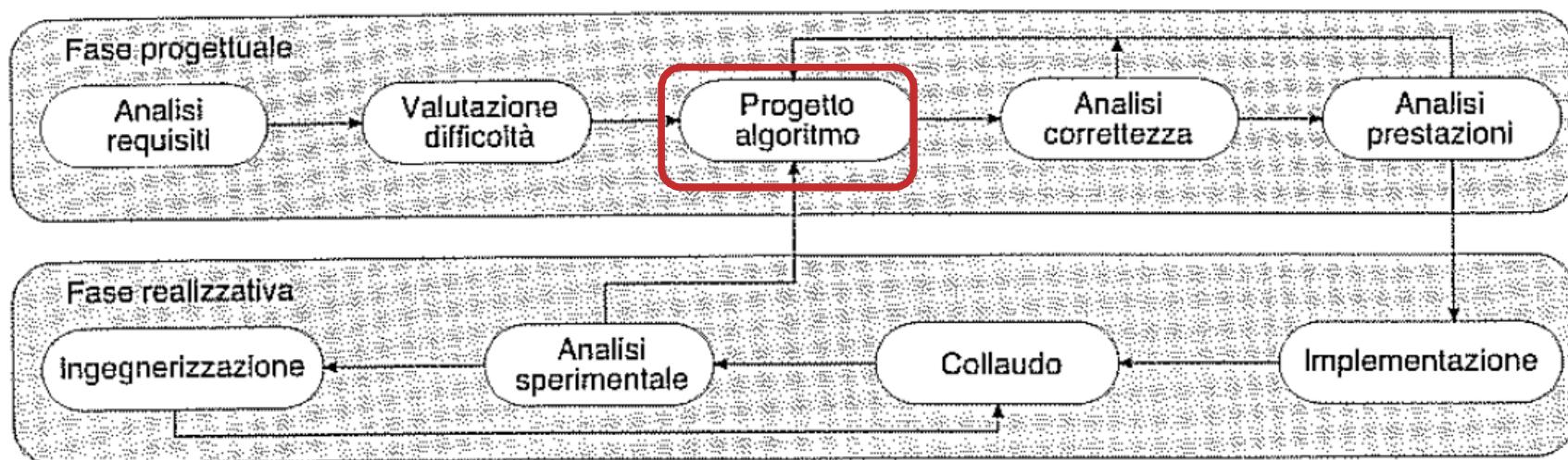
• Fase progettuale



Valutazione delle difficoltà

- Idealmente vorremmo algoritmi che minimizzino le risorse spazio/tempo utilizzate
 - **Complessità (spaziale o temporale) dell'algoritmo**
- Tuttavia, **per ogni problema computazionale esistono dei limiti inferiori** alle quantità di risorse di calcolo necessarie alla sua risoluzione
 - **Complessità del problema**
- Stimare tali limiti ci permette di capire se un dato algoritmo è genuino o se è sperabile poter fare meglio
 - *Esempi: ricerca di un elemento, ordinamento su confronto, duplicati*

• Fase progettuale



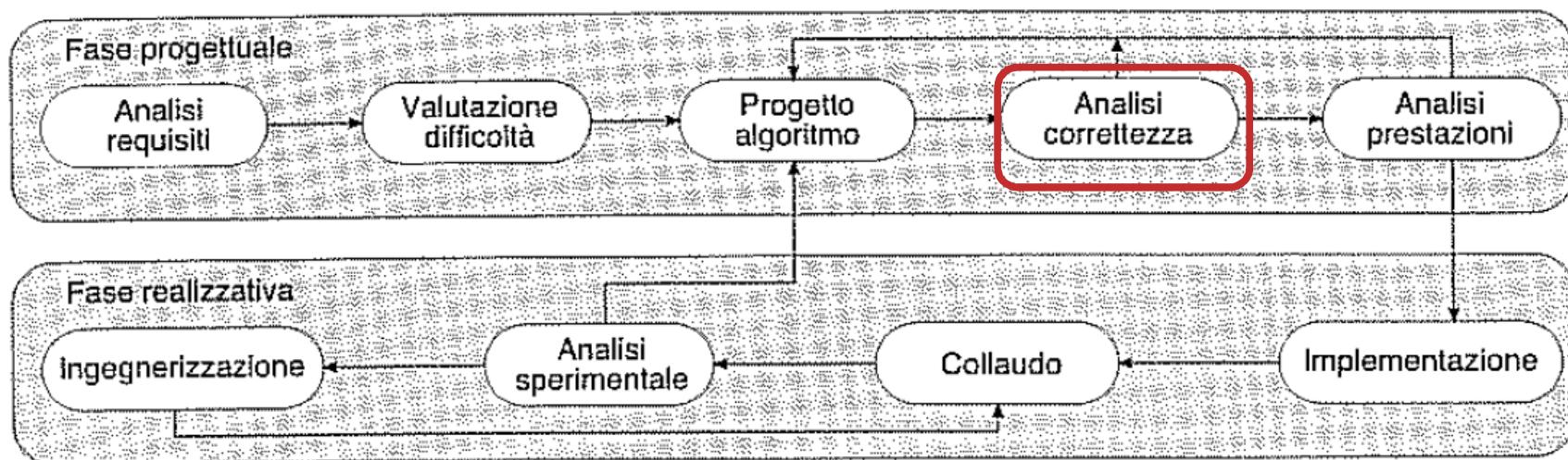
Progetto di un algoritmo risolutivo

- Richiede creatività ed esperienza
- Per mantenere il massimo grado di generalità, descriveremo gli algoritmi in **pseudocodice**:
 - ricorda linguaggi di programmazione reali come C, C++ o Java
 - può contenere alcune frasi in italiano



```
algoritmo verificaDup(sequenza S)
  for each elemento x della sequenza S do
    for each elemento y che segue x nella sequenza S do
      if x = y then return true
  return false
```

• Fase progettuale



Correttezza

Vogliamo progettare algoritmi che:

- Producano correttamente il risultato desiderato

Analisi di correttezza

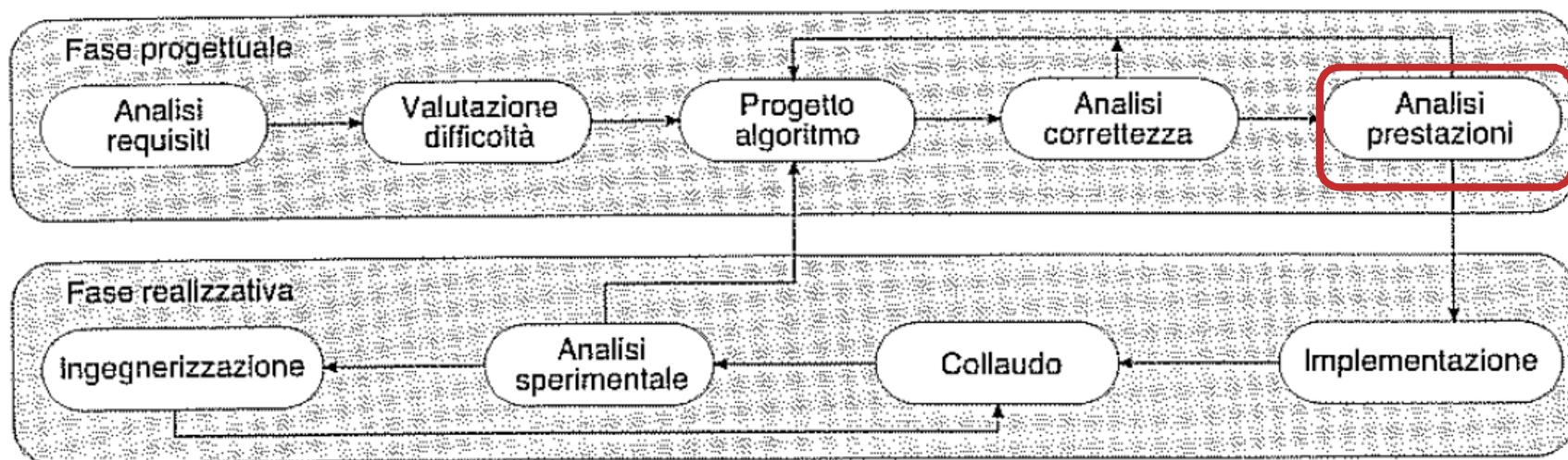
- **Verifica formale della correttezza**
 - prova che i requisiti sui dati prodotti dall'algoritmo (requisiti sull'output) siano soddisfatti
 - ad esempio, dimostrando che certe proprietà sui dati tenuti in memoria dall'algoritmo vengono mantenute ad ogni passo
- Ove l'analisi evidenzi discrepanze, è necessario ritornare alla fase di progettazione
- **Esempio (Duplicati):** Osserviamo che l'algoritmo confronta almeno una volta ogni coppia di elementi; quindi, se esiste un elemento che si ripete, verrà sicuramente trovato (true). Per lo stesso motivo, se l'algoritmo restituisce false, abbiamo invece la garanzia che tutte le coppie sono distinte.

Correttezza ed efficienza

Vogliamo progettare algoritmi che:

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

• Fase progettuale



Perché analizzare gli 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 software, prima ancora di scriverne le prime linee di codice

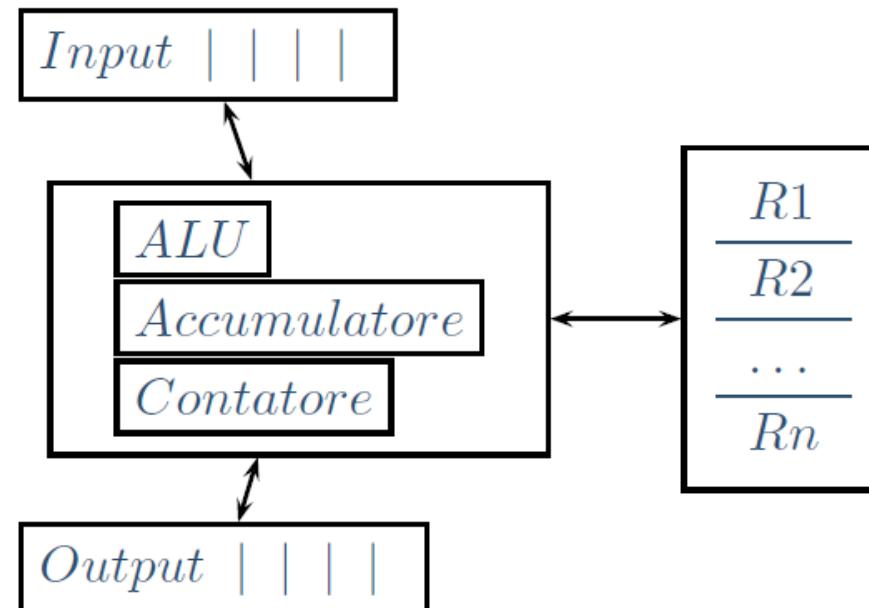
Analisi delle prestazioni

- Assumiamo di utilizzare *Mono-Processore + RAM (Random Access Memory)*: assenza di concorrenza e parallelismo
- **Analisi teorica** condotta fissando:
 1. **Modello di calcolo** e
 - tipicamente la **macchina a registri RAM (Random Access Machine)**
 - si ispira all'architettura di Von Neumann e alla macchina di Turing
 - l'accesso diretto o casuale alla memoria, e non sequenziale come nella macchina di Turing
 2. **Modello di costo teorico**
 - **modello a costo uniforme** (indipendente dalla dimensione degli operandi coinvolti) – come prima approssimazione
 - **modello a costo logaritmico** (dipende dalla dimensione degli operandi coinvolti) – un po' complicato

Macchine a registri

Sono caratterizzate da:

- **Nastri** di input e di output
- Un numero arbitrario di celle i di **memoria** (detti **registri**) contenenti un intero arbitrario: $R[i]$ (intero o reale di dimensione arbitraria)
- ALU e registri speciali:
 - **Contatore**: l'indirizzo istruzione successiva
 - **Accumulatore R0**: l'operando su cui agisce l'istruzione corrente
- **Passo di calcolo**:
 - Istruzione di I/O su nastro
 - Istruzione di I/O da/in memoria
 - Istruzione aritmetica o logica



Il modello assume che l'algoritmo sia sequenziale e deterministico!

Modello a costo uniforme

- **Ogni istruzione elementare** della macchina a registri costa una **unità di tempo**
- **Ciascuna locazione di memoria**, compreso l'accumulatore, richiede **una unità di spazio**
- **Pregio**: approccio molto semplice
- **Difetto**: analisi non realistica
 - Moltiplicare due interi di 32 bit costa quanto moltiplicare due interi di 1000 bit
 - Accedere ad una memoria di 1 kilobyte costa come accedere ad una memoria di 1 terabyte

Modello a costo logaritmico

- La complessità del **calcolo** è proporzionale al numero di bit dell'operando e l'**accesso** è proporzionale al numero di bit dell'indirizzo
 - l'elaborazione di un intero n ha costo $O(\log n)$
 - l'accesso ad un registro i ha costo $O(\log i)$
- Si applica la medesima filosofia per l'occupazione di spazio
- **Pregio:** la valutazione dei costi è più realistica dal punto di vista asintotico (ovvero per operandi molto grandi, o per memoria molto grande)
- **Difetto:** analisi più laboriosa

Il nostro modello di costo

- Limitiamo a $c * \log n$ bit la dimensione dei numeri interi rappresentabili
 - c è una costante
 - n è la dimensione dell'input del problema
- Non ammettiamo operazioni sui numeri in virgola mobile
- Quindi gli algoritmi che presentiamo d'ora in poi sono implementabili su una macchina a registri con **interi** di dimensione $\leq n^c$

Tempo di esecuzione

- Calcoliamo il numero di linee di codice mandate in esecuzione
 - misura indipendente dalla piattaforma utilizzata
 - ma approssimativa!
- Meglio considerare il numero di passi elementari eseguiti dall'algoritmo al variare della dimensione n dell'input: funzione $T(n)$
- Ci interessa soprattutto il comportamento di $T(n)$ per valori “grandi” di n (**comportamento asintotico**)

Occupazione di memoria

- Se abbiamo un algoritmo lento, dovremo solo attendere più a lungo per ottenere il risultato
- Ma se un algoritmo richiede più spazio di quello a disposizione, non otterremo mai la soluzione, indipendentemente da quanto attendiamo

Calcolabilità e complessità

- **Calcolabilità:** nozioni di algoritmo e di problema non decidibile
- **Complessità:** nozione di algoritmo efficiente e di problema intrattabile

La calcolabilità ha lo scopo di classificare i problemi in risolvibili e non risolvibili, mentre la complessità in facili e difficili

Calcolo della complessità

- **Complessità di un algoritmo:** misura del numero di passi elementari che si devono eseguire per risolvere il problema
- **Complessità di un problema:** complessità del **migliore** algoritmo che lo risolve

in funzione della **taglia** (dimensione dell'input) del problema

Taglia di un problema

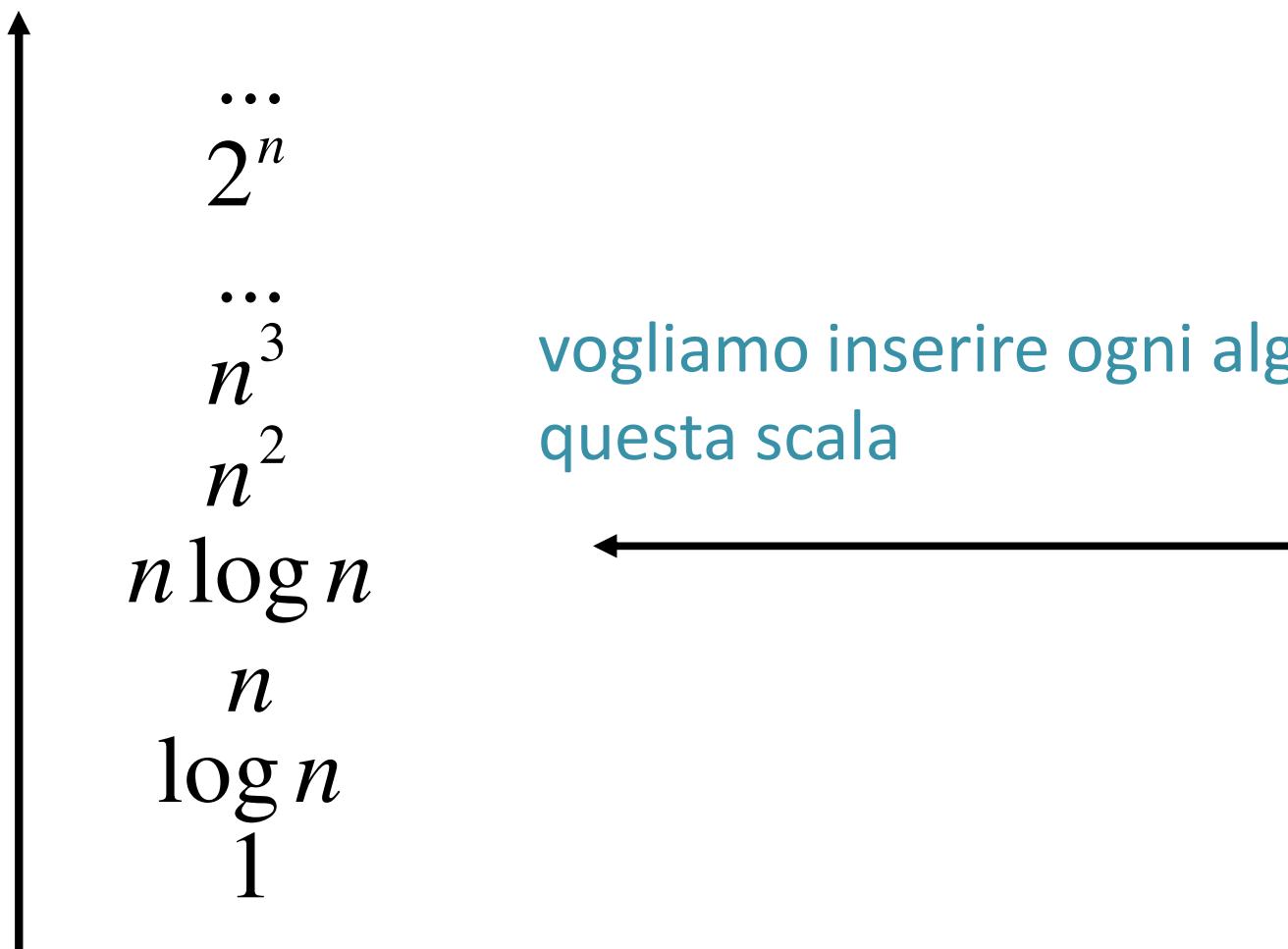
misura della dimensione dell'input (in bit, parole, ecc.)

- **ordinamento**: numero di oggetti da ordinare
- **gestione dati**: numero dei dati da gestire
- **algoritmi e problemi su grafi**: numero di archi e nodi del grafo

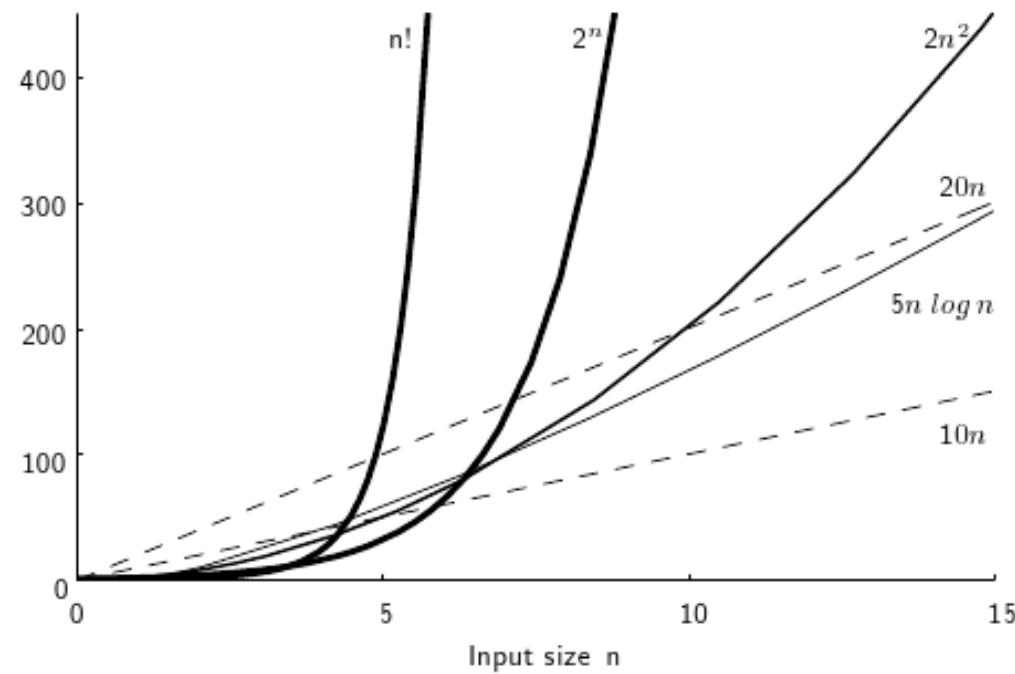
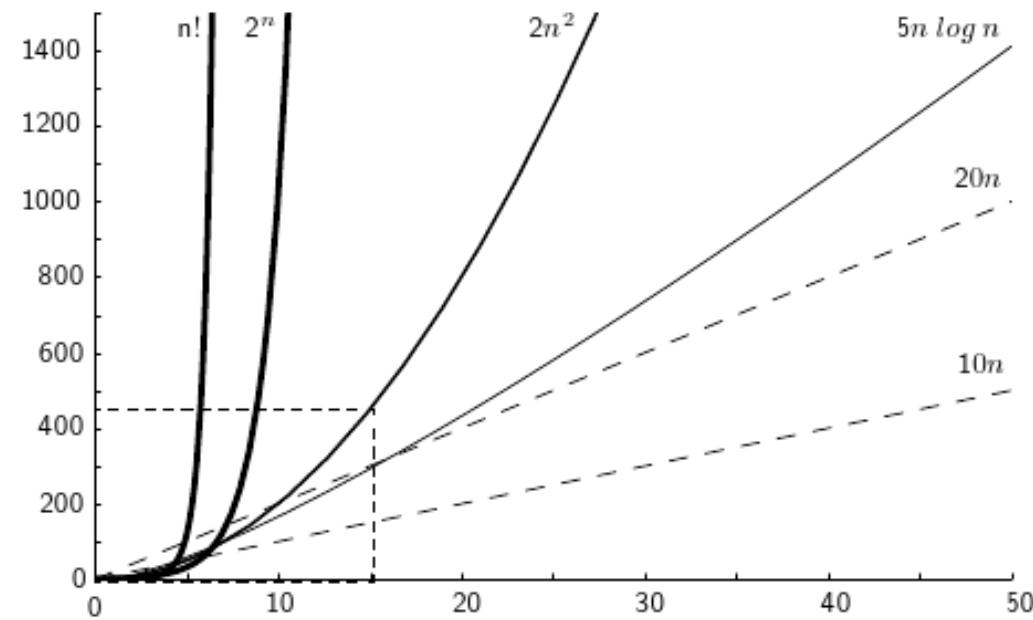
dipende dalla rappresentazione dei dati (struttura dati)

Paragonare tra loro algoritmi

Abbiamo una **scala di complessità**:



Growth Rate



Algoritmi esponenziali

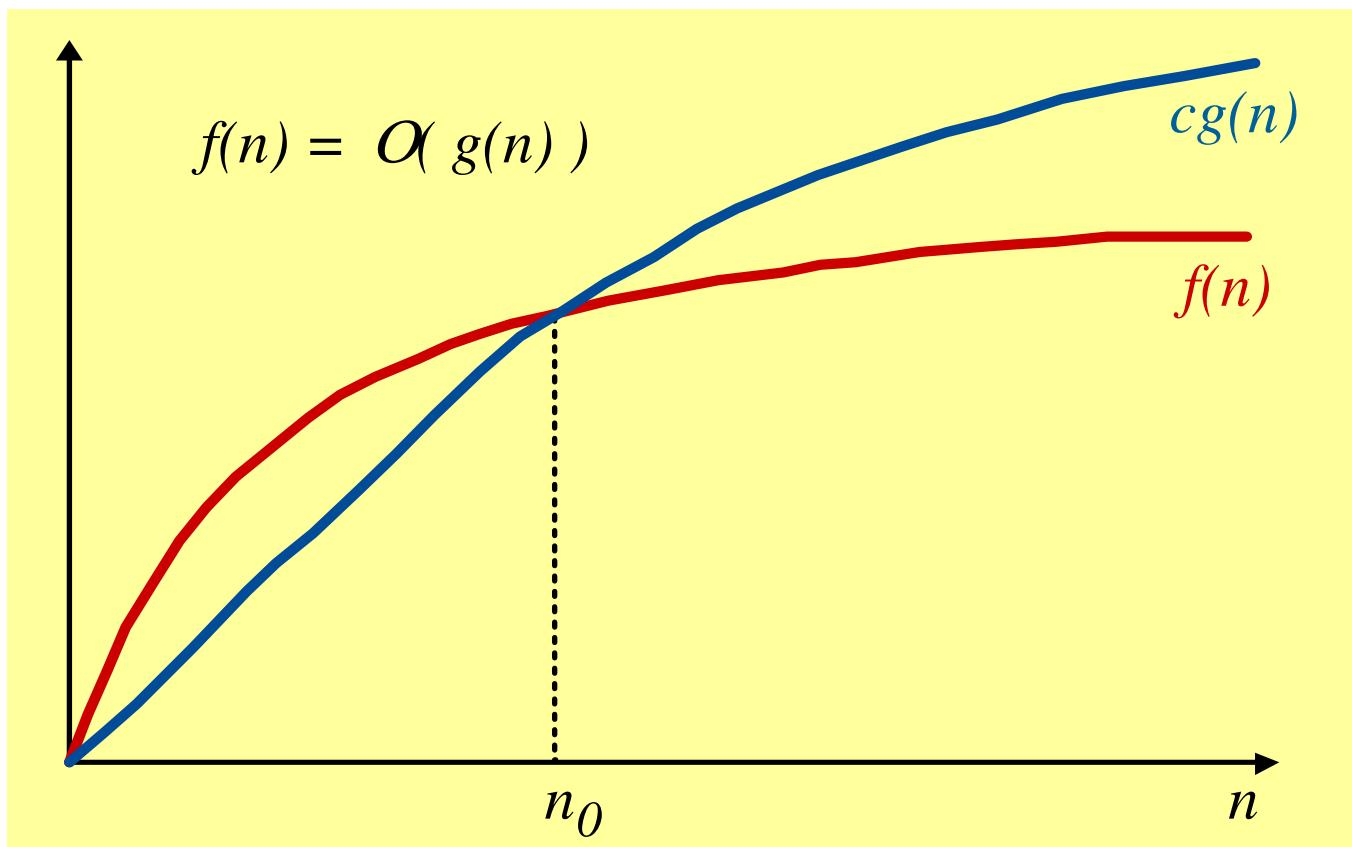
n	$\log_2(n)$	$100*n$	$10*n^2$	n^3	2^n
10	2.3 μ s	1ms	1ms	1ms	1024 μ s ~ 1ms
20	2.99 μ s	2ms	4ms	8ms	1.048sec
60	4.09 μ s	6ms	36ms	0.21sec	$2^{60} \mu$ s ~ 366 secoli

Notazione asintotica

- 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 come le costanti moltiplicative...
- Useremo a questo scopo la notazione asintotica O

Notazione asintotica O

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



Esempi di Growth Rate di T(n)

Grossolanamente....

Esempio1. al crescere di n, come cresce T(n)?

Istruzione di assegnamento $T(n) = c$ costante = $O(1)$

Esempio 2: $T(n) = c_1n + c_2$ steps = $O(n)$

```
// Return position of largest value in "A"
static int largest(int[] A) {
    int currlarge = 0; // Position of largest
    for (int i=1; i<A.length; i++)
        if (A[currlarge] < A[i])
            currlarge = i; // Remember pos
    return currlarge; // Return largest pos
}
```

Esempi (cont.)

Esempio 3: $T(n) = c_1 n^2 + c = O(n^2)$

```
sum = 0;
for (i=1; i<=n; i++)
    for (j=1; j<n; j++)
        sum++;
}
```

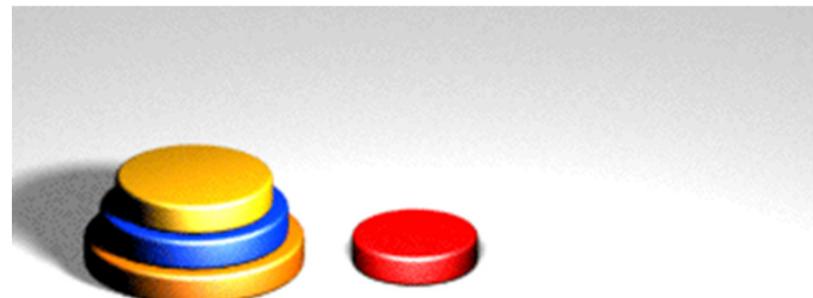
Esempi (cont.)

Esistono degli algoritmi che sono **intrinsecamente esponenziali**

- **Esempio:** un algoritmo che calcola le permutazioni di n elementi:

$$n! \sim n^n$$

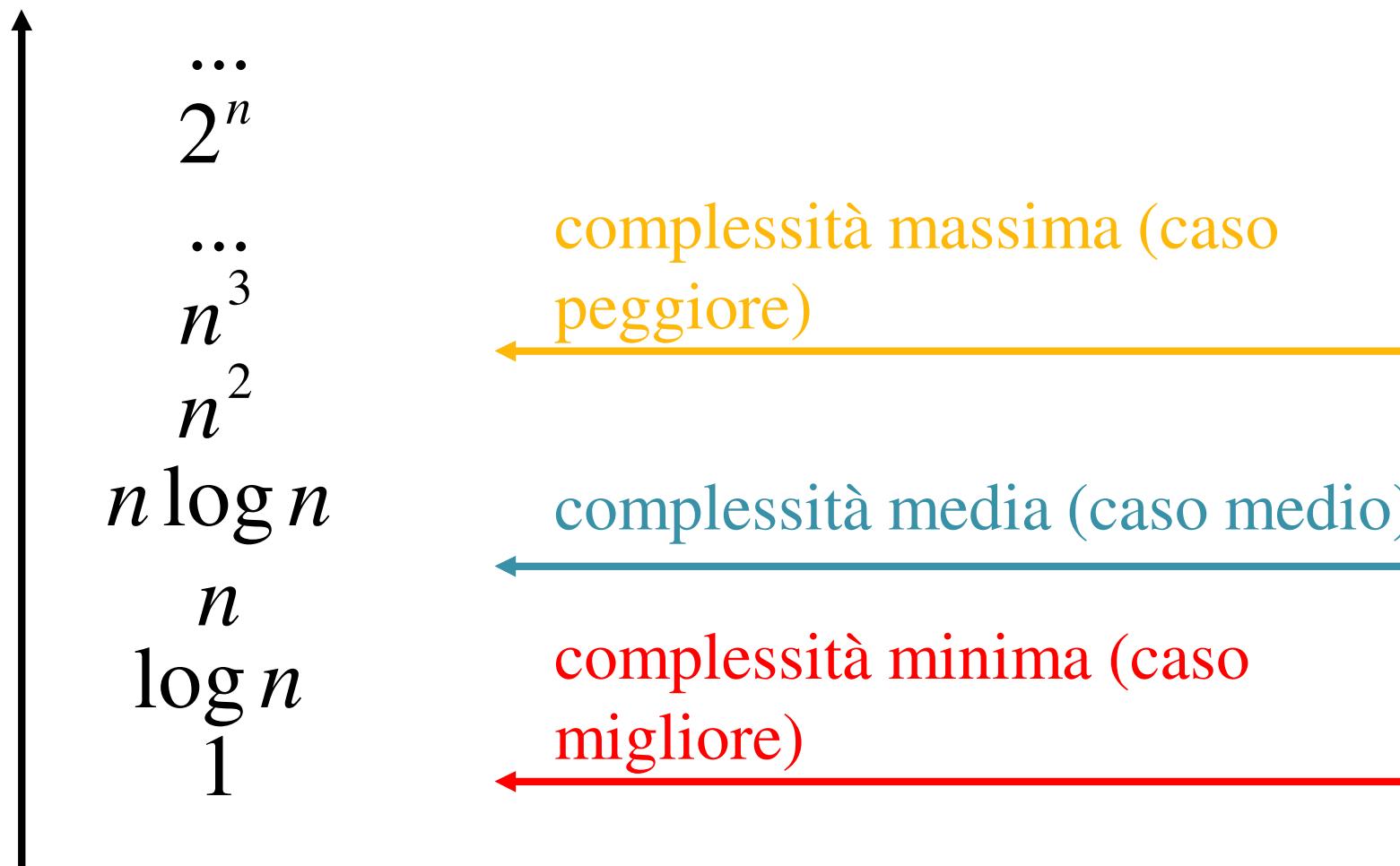
- **Esempio:** Torre di Hanoi: per spostare n dischi ci vogliono $2^n - 1$ mosse



Forma dell'input

Un algoritmo non ha, in generale, lo stesso comportamento **per tutti gli input di una data taglia**:

- *Esempio:* ordinare n oggetti può dipendere dal loro ordine iniziale (per esempio, potrebbero già essere ordinati!)



Progetto di algoritmi più efficienti

- Un'analisi più approfondita del problema permette di progettare algoritmi più efficienti
- **Esempio (Duplicati):** `verificaDup` ha tempo di esecuzione $O(1)$ nel caso migliore e $O(n^2)$ nel caso peggiore, per una sequenza di n elementi.



```
algoritmo verificaDup(sequenza S)
    for each elemento x della sequenza S do
        for each elemento y che segue x nella sequenza S do
            if x = y then return true
    return false
```

IDEA: Possiamo fare di meglio se la sequenza in input è ordinata (i duplicati sono consecutivi)!

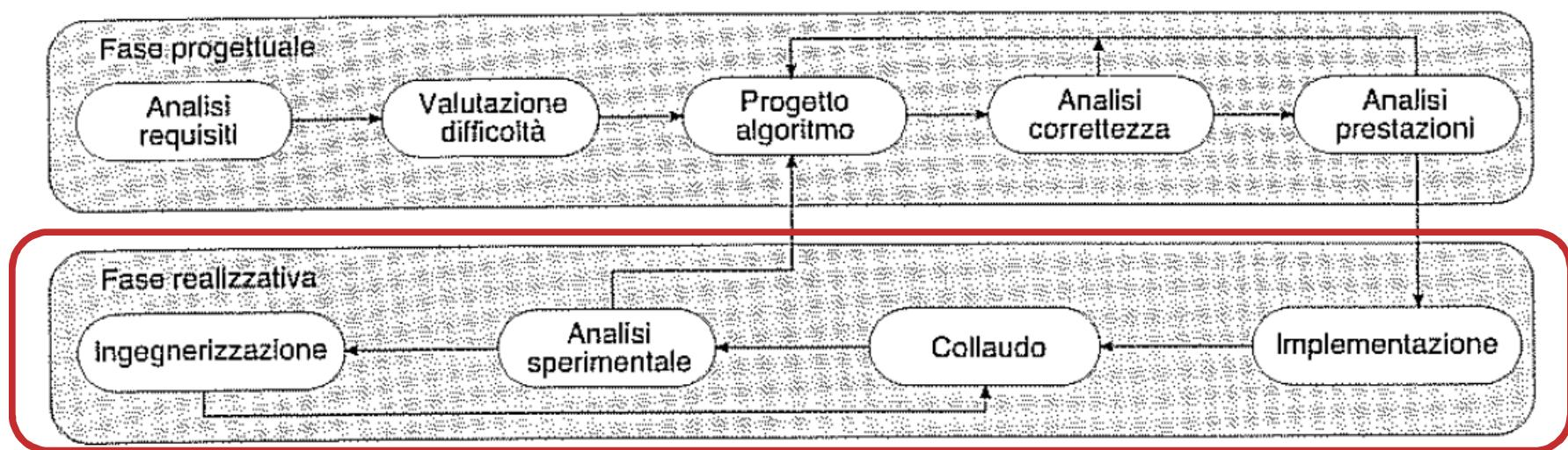
Progetto di algoritmi più efficienti

- **Esempio (Duplicati):** `verificaDupOrd` può essere implementato in modo da avere tempo di esecuzione $O(n \log n)$ nel caso peggiore:
 - Linea 1: Esistono algoritmi di ordinamento in grado di operare in tempo $O(n \log n)$ anche nel caso peggiore
 - Le linee 2-5 richiedono tempo $O(n)$ nel caso peggiore

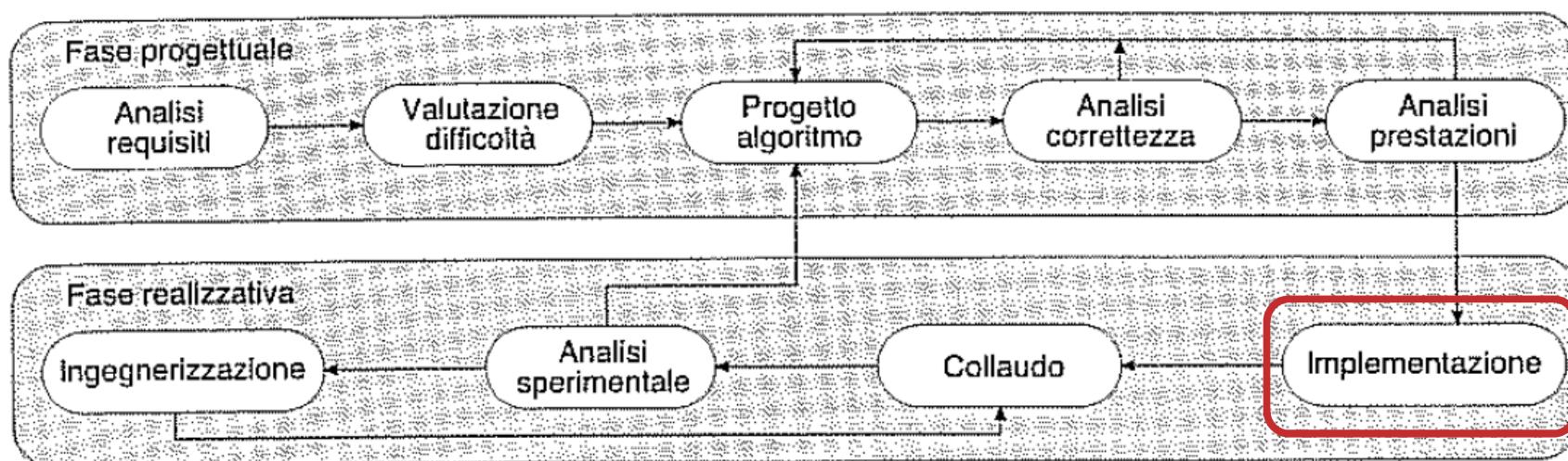


```
algoritmo verificaDupOrd(sequenza S)
    ordina la sequenza S in modo non-decrescente
    for each elemento x nella sequenza S riordinata, tranne l'ultimo do
        sia y l'elemento che segue x in S
        if x = y then return true
    return false
```

• Fase realizzativa



• Fase realizzativa



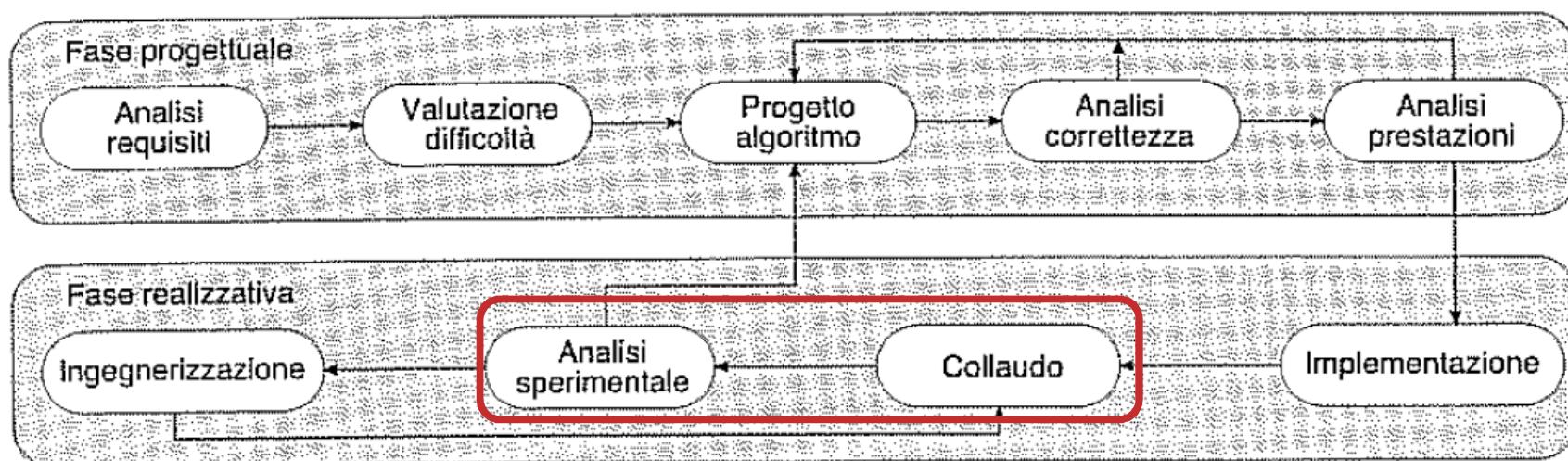
Implementazione in un linguaggio di programmazione

- Comporta molti dettagli implementativi non del tutto specificati nel pseudocodice
- *Esempio (Duplicati)*: assumiamo che la sequenza S sia rappresentata come una lista (in Java un oggetto di una classe che implementa l'interfaccia **java.util.List** del **Java Collection Framework**)



```
public static boolean verificaDupList(List S) {  
    for (int i = 0; i < S.size(); i++) {  
        Object x = S.get(i);  
        for (int j = i + 1; j < S.size(); j++) {  
            Object y = S.get(j);  
            if (x.equals(y)) return true;  
        }  
    }  
    return false;  
}
```

• Fase realizzativa

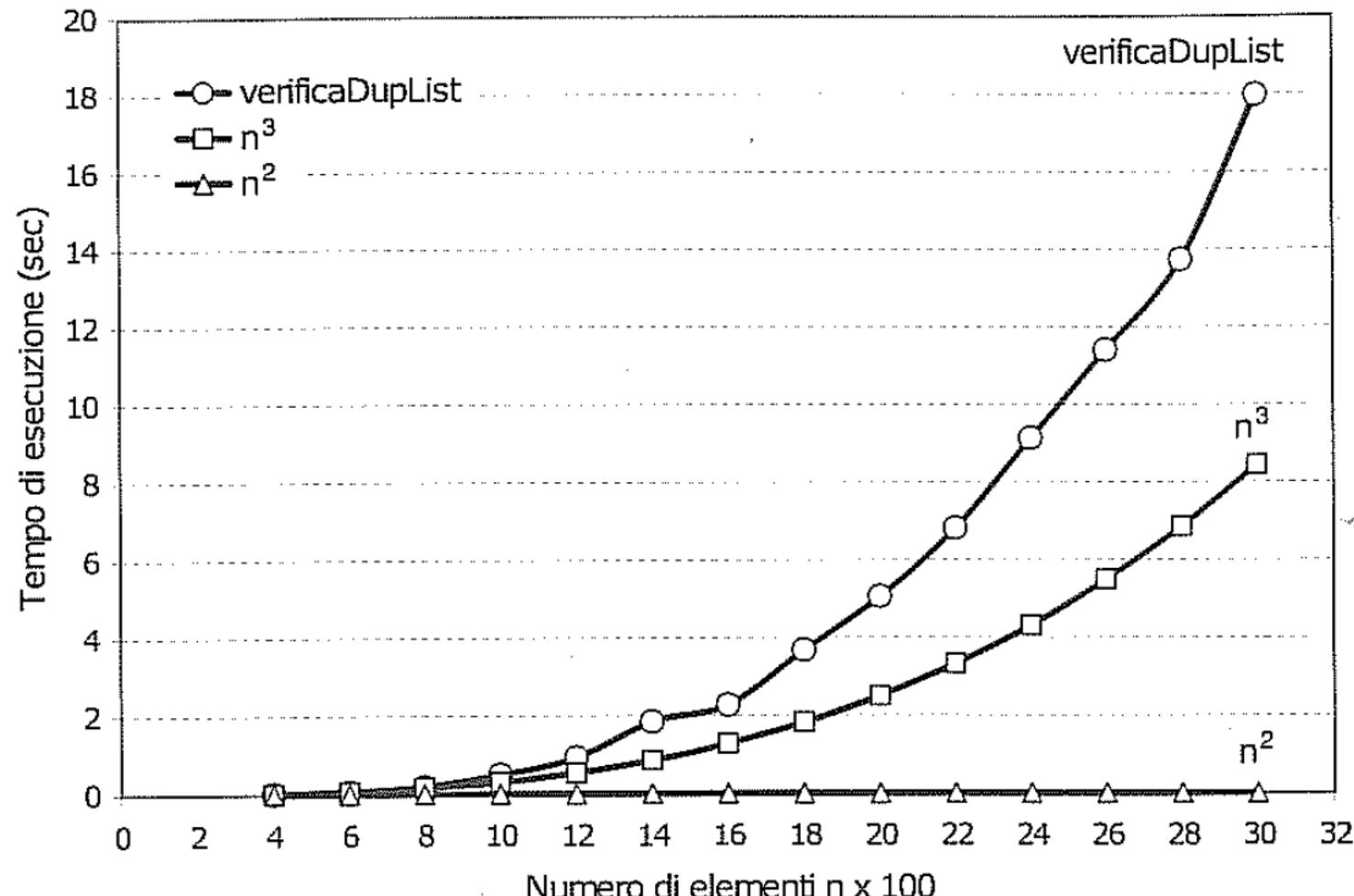


Collaudo e analisi sperimentale

- Una data implementazione va testata su data set reali per identificare eventuali errori implementativi
 - **Una data implementazione può risultare meno efficiente rispetto ai risultati dell'analisi teorica!**
 - In tal caso **occorre una messa a punto e re-ingegnerizzazione** opportuna che tenga conto delle caratteristiche del linguaggio e delle piattaforme di esecuzione
- Tale analisi sperimentale ci aiuta anche ad identificare le “**costanti nascoste**” (non considerate nell’analisi teorica) e a confrontare in modo più preciso algoritmi apparentemente simili

Collaudo e analisi sperimentale

- Esempio (Duplicati) La curva di `verificaDupList` non sembra somigliare alla funzione $c n^2$ ma alla funzione $c n^3$



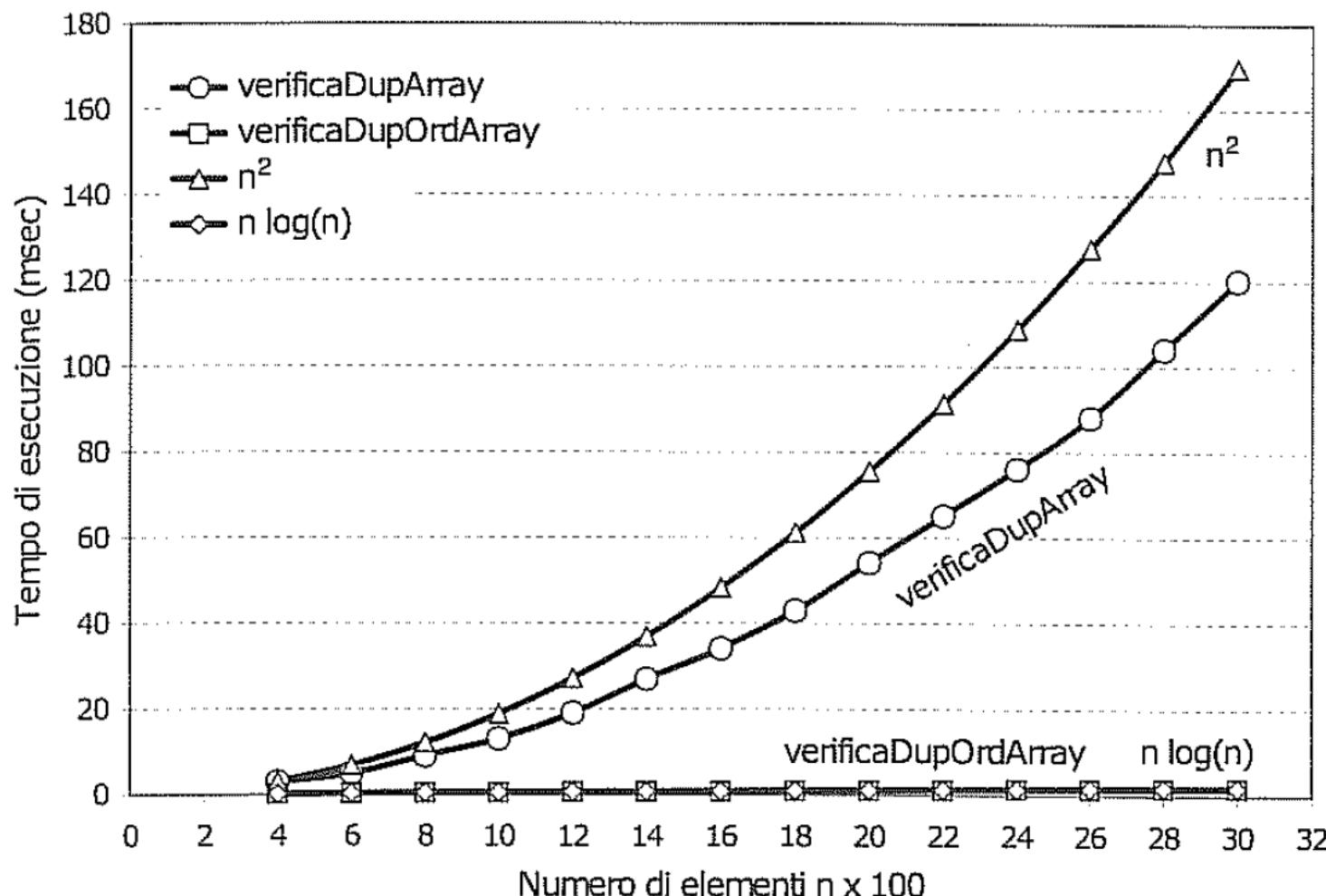
Collaudo e analisi sperimentale

- Esempio (Duplicati) L'inefficienza può essere eliminata implementando S come array.

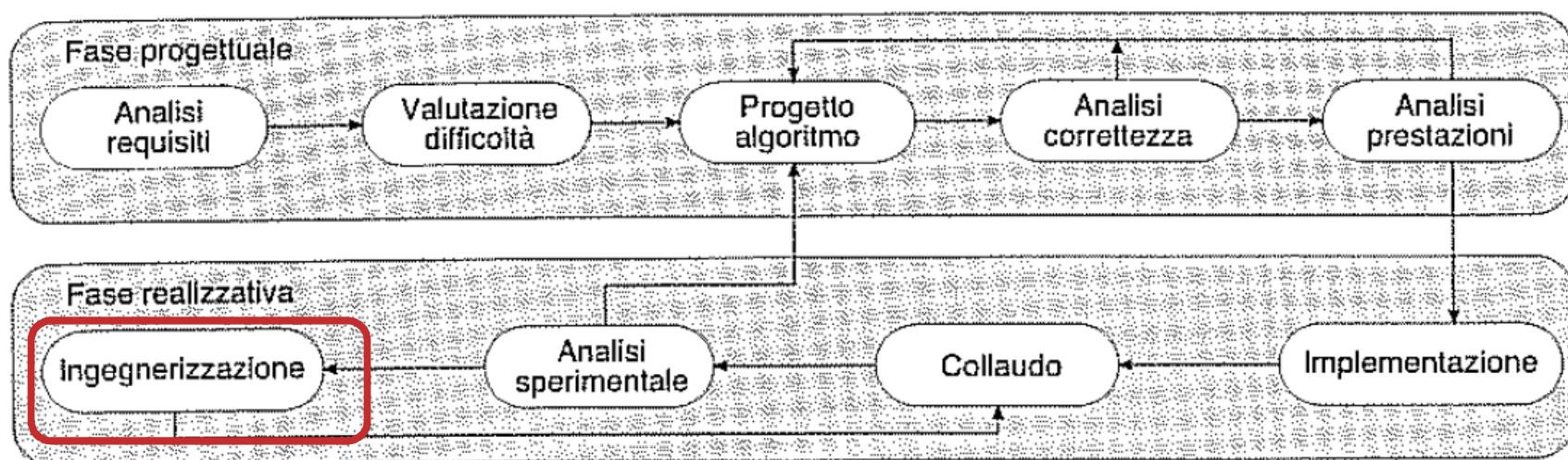
```
public static boolean verificaDupArray(List S) {  
    Object[] T = S.toArray();  
    for (int i = 0; i < T.length; i++) {  
        Object x = T[i];  
        for (int j = i + 1; j < T.length; j++) {  
            Object y = T[j];  
            if (x.equals(y)) return true;  
        }  
    }  
    return false;  
}
```

Collaudo e analisi sperimentale

- Esempio (Duplicati) Con `verificaDupArray` i risultati sono perfettamente allineati con la predizione teorica.



• Fase realizzativa



Ingegnerizzazione per ottimizzare/fare refactoring a livello di codice

Algoritmi e Strutture Dati

Capitolo 2
Modelli di calcolo e
metodologie di analisi

Camil Demetrescu, Irene Finocchi,
Giuseppe F. Italiano



Notazione asintotica



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



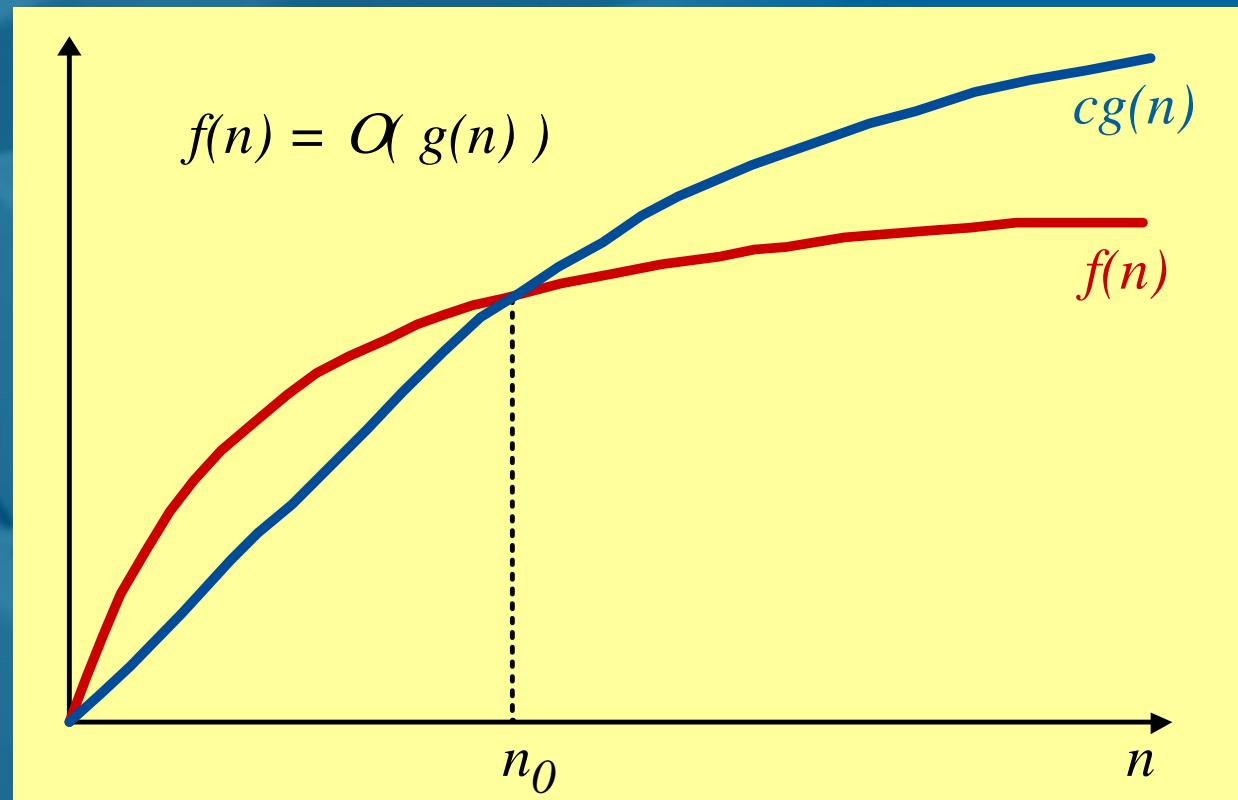
Tre notazioni: motivazioni e definizioni

O , Ω , Θ

- O e Ω forniscono un limite “lasco”, rispettivamente per i limiti superiore ed inferiore
- Θ fornisce un limite “stretto”
 - In alcuni contesti è difficile trovare un limite stretto per l’andamento delle funzioni, per cui ci si accontenta di un limite meno preciso

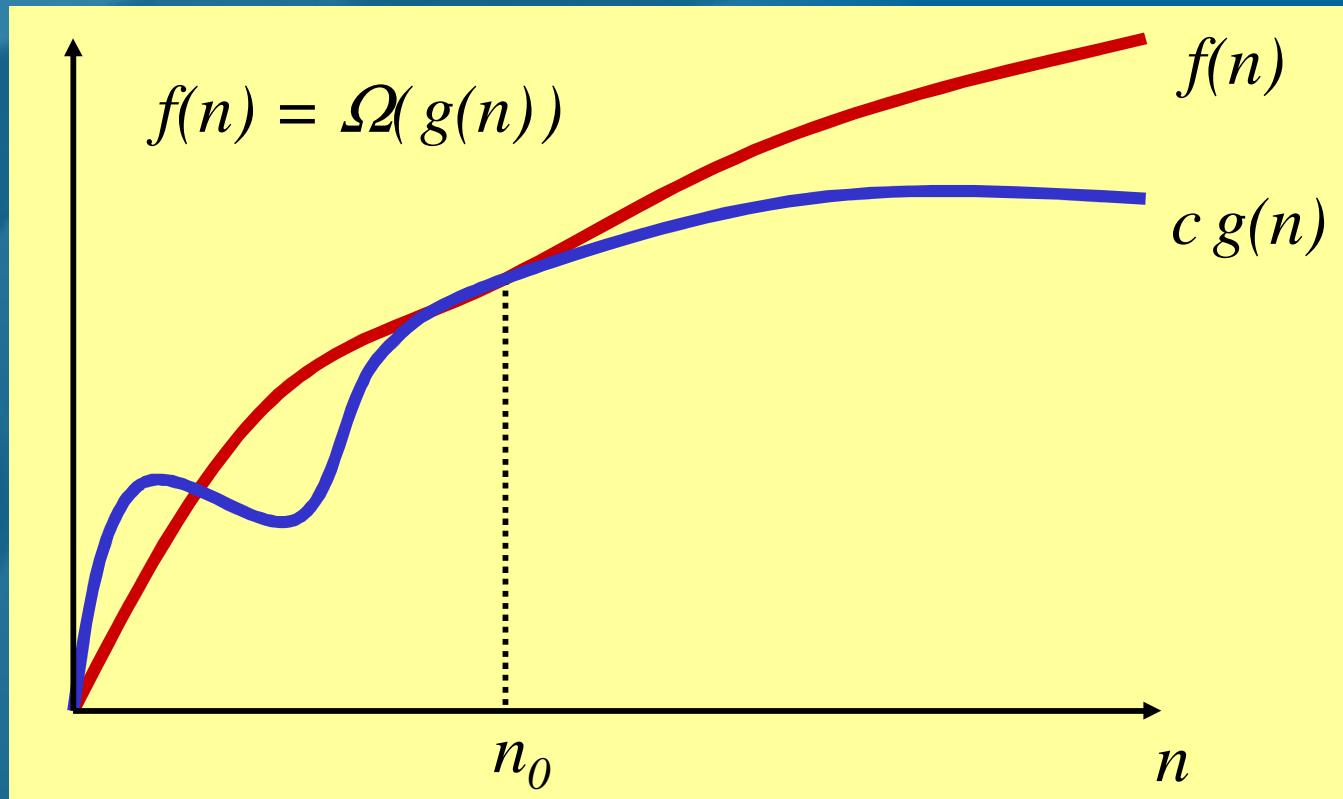
Notazione asintotica O

$f(n) = O(g(n))$ se \exists due costanti $c > 0$ e $n_0 \geq 0$ tali che $f(n) \leq c g(n)$ per ogni $n \geq n_0$



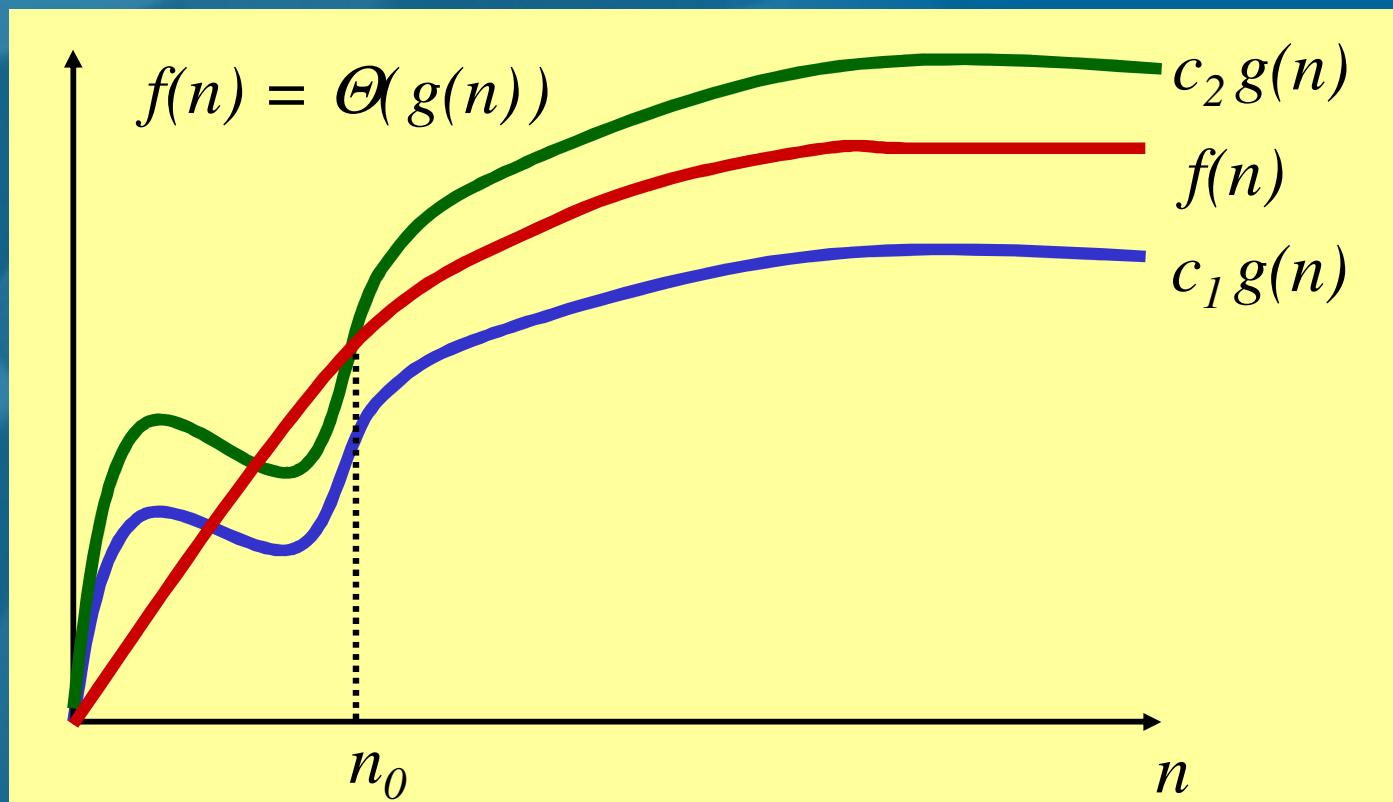
Notazione asintotica Ω

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



Notazione asintotica Θ

$f(n) = \Theta(g(n))$ se \exists tre costanti $c_1, c_2 > 0$ e $n_0 \geq 0$
tali che $c_1 g(n) \leq f(n) \leq c_2 g(n)$ per ogni $n \geq n_0$





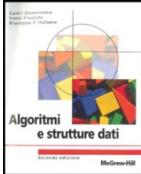
Abuso di notazione

Si noti che usiamo $f(n) = O(g(n))$ invece di $f(n) \in O(g(n))$

$f(n) = O(g(n))$ si legge $f(n)$ è “o grande” di $g(n)$

Se $f(n) = O(g(n))$ è il tempo calcolo richiesto da un algoritmo $\mathcal{A}lg$ diciamo che $O(g(n))$ è un **limite superiore asintotico** per la complessità tempo di $\mathcal{A}lg$

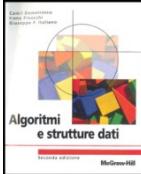
Similmente per le altre notazioni



Teorema

Dalle definizioni discende il seguente teorema:

Date due funzioni $f(n)$ e $g(n)$, $f(n) \in \Theta(g(n))$ se
e solo se $f(n) \in O(g(n))$ e $f(n) \in \Omega(g(n))$



Notazione asintotica: esempi

- Sia $g(n)=3n^2+10n$
- $g(n)=O(n^2)$: scegliere $c=4$ e $n_0=10$
- $g(n)=\Omega(n^2)$: scegliere $c=1$ e $n_0=0$
- $g(n)=\Theta(n^2)$: infatti $g(n)=\Theta(f(n))$ se e solo se $g(n)=O(f(n))$ e $g(n)=\Omega(f(n))$
- $g(n)=O(n^3)$ ma $g(n)\neq\Theta(n^3)$



Transitività

$$f(n) = \Theta(g(n)) \quad e \quad g(n) = \Theta(h(n)) \quad \Rightarrow \quad f(n) = \Theta(h(n))$$

$$f(n) = O(g(n)) \quad e \quad g(n) = O(h(n)) \quad \Rightarrow \quad f(n) = O(h(n))$$

$$f(n) = \Omega(g(n)) \quad e \quad g(n) = \Omega(h(n)) \quad \Rightarrow \quad f(n) = \Omega(h(n))$$

Legge transitiva per O

La relazione “è O grande di” soddisfa la proprietà transitiva, cioè se $f(n)$ è $O(g(n))$ e $g(n)$ è $O(h(n))$ abbiamo anche che $f(n)$ è $O(h(n))$.

$f(n)$ è $O(g(n)) \rightarrow \forall n \geq n_1 \ f(n) \leq c_1 g(n).$

$g(n)$ è $O(h(n)) \rightarrow \forall n \geq n_2 \ g(n) \leq c_2 h(n).$

Basta prendere come $n_0 = \max(n_1, n_2)$ e $c = c_1 \cdot c_2$.



Riflessività

$$f(n) = \Theta(f(n))$$

$$f(n) = O(f(n))$$

$$f(n) = \Omega(f(n))$$



Confronto tra funzioni : simmetria

$$f(n) = \Theta(g(n)) \Leftrightarrow g(n) = \Theta(f(n))$$

Confronto tra funzioni : simmetria transposta

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

Regole di semplificazione

1. Se $f(n) = O(k g(n))$ con k costante >0 , allora $f(n) = O(g(n))$
2. **Somma.** Se $f_1(n) = O(g_1(n))$ e $f_2(n) = O(g_2(n))$,
allora $f_1(n) + f_2(n) = O(\max(g_1(n), g_2(n)))$
(Blocchi sequenziali di istruzioni, rami if-then-else, switch)
3. **Prodotto.** Se $f_1(n) = O(g_1(n))$ e $f_2(n) = O(g_2(n))$,
allora $f_1(n)f_2(n) = O(g_1(n)g_2(n))$
(Cicli for, while, do-while)

Similmente per Ω e Θ

Esempio: Se $T(n) = 3n^4 + 5n^2$, allora $T(n) = ?$

Regole di semplificazione

1. Se $f(n) = O(k g(n))$ con k costante > 0 , allora $f(n) = O(g(n))$
2. **Somma.** Se $f_1(n) = O(g_1(n))$ e $f_2(n) = O(g_2(n))$,
allora $f_1(n) + f_2(n) = O(\max(g_1(n), g_2(n)))$
(Blocchi sequenziali di istruzioni, rami if-then-else, switch)
3. **Prodotto.** Se $f_1(n) = O(g_1(n))$ e $f_2(n) = O(g_2(n))$,
allora $f_1(n)f_2(n) = O(g_1(n)g_2(n))$
(Cicli for, while, do-while)

Similmente per Ω e Θ

Esempio: Se $T(n) = 3n^4 + 5n^2$, allora $T(n) = O(n^4)$



Regole di semplificazione: esercizi

Di che ordine è un algoritmo che ha come funzione $T(n)$:

- a. $8n^3 - 9n$
- b. $7 \log n + 20$
- c. $9 \log n + n$
- d. $n \log n + n^2$
- e. $\log(\log n) + 3 \log n + 4$



Regole di semplificazione: esercizio

Supponi che l'implementazione di un certo algoritmo in Java che processa un array di n elementi sia del tipo:

```
for (int pass = 1; pass <= n; pass++){
    for (int index = 0; index < n; index++)
        for (int count = 1; count < 10; count++)
            . . . //<istruzione non dipendente da n>
}
```

Qual è l'ordine della funzione $T(n)$ dell'algoritmo?



Notazione asintotica versus numeri naturali

$$f(n) = O(g(n)) \approx a \leq b$$

$$f(n) = \Omega(g(n)) \approx a \geq b$$

$$f(n) = \Theta(g(n)) \approx a = b$$

Proprietà di tricotomia

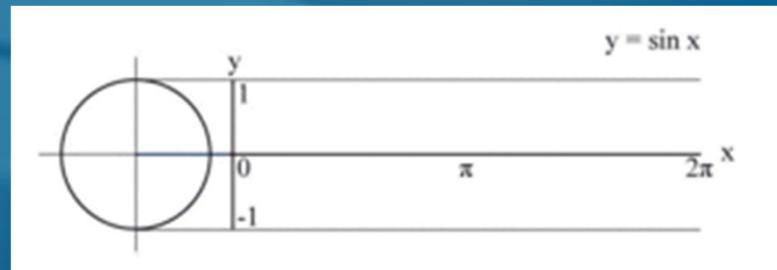
Per ogni coppia di numeri reali a e b , deve valere esattamente una delle seguenti espressioni:

$$a < b, \quad a = b, \quad a > b$$

Sebbene qualunque coppia di numeri possa essere confrontata, non tutte le funzioni sono asintoticamente confrontabili!

Esempio:

$$f(n) = n, \quad g(n) = n^{1+\sin n}$$



Teorema dei limiti (1)

Se $f(n)$ e $g(n)$ sono asintoticamente positive, dal limite del loro rapporto è possibile dedurre:

$$1. \lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = 0 \Rightarrow f(n) = O(g(n))$$

$$f(n) \neq \Theta(g(n))$$

$$f(n) \neq \Omega(g(n))$$

Notare che:

$$f(n) = O(g(n)) \not\Rightarrow \lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = 0$$

$$f(n) = O(g(n)) \Rightarrow \lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} \text{ (se esiste)} < \infty$$

Teorema dei limiti (2)

Se $f(n)$ e $g(n)$ sono asintoticamente positive, dal limite del loro rapporto è possibile dedurre:

$$2. \lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = \infty \Rightarrow f(n) = \Omega(g(n))$$

$$f(n) \neq O(g(n))$$

$$f(n) \neq \Theta(g(n))$$

Notare che:

$$f(n) = \Omega(g(n)) \not\Rightarrow \lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = \infty$$

$$f(n) = \Omega(g(n)) \Rightarrow \lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} \text{ (se esiste)} > 0$$

Teorema dei limiti (3)

Se $f(n)$ e $g(n)$ sono asintoticamente positive, dal limite del loro rapporto è possibile dedurre:

$$3. \lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} \in R^+ \Rightarrow f(n) = \Theta(g(n)) \\ g(n) = \Theta(f(n))$$

$$4. \lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} \text{ non esiste} \Rightarrow f(n) \text{ e } g(n) \text{ non sono asintoticamente confrontabili.}$$

Dimostrazione: Segue dalla teoria di Analisi Matematica.



Un pò di terminologia

Alcune **classi di complessità asintotica**:

O(1): costante, non dipende dalla dimensione dei dati

O($\lg n$): logaritmica (in generale: $\lg^k n$ $k \geq 1$)

O(n): lineare

O($n \lg n$): pseudolineare

O(n^2): quadratica (n^k polinomiale $k > 1$)

O(2^n): esponenziale (in generale: k^n $k > 1$)



Sommatorie

[vedi dispensa su Sommatorie]

Esempi – complessità tempo (1)

Esempio 1: $a = b;$

Tempo costante $\Theta(1)$

Esempio 2:

```
sum = 0;  
for (i=1; i<=n; i++)  
    sum += n;
```

$\Theta(n)$

Esempi –complessità tempo (2)

Example 3:

```
sum = 0;  
for (j=1; j<=n; j++) //Primo for (doppio)  
    for (i=1; i<=j; i++) serie aritmetica  
        sum++;  
    for (k=0; k<n; k++) //Secondo for  
        A[k] = k;
```

$\Theta(n^2)$

Esempi – complessità tempo(3)

Esempio 4:

```
sum1 = 0;  
for (i=1; i<=n; i++)  
    for (j=1; j<=n; j++)  
        sum1++;
```

```
sum2 = 0;  
for (i=1; i<=n; i++)  
    for (j=1; j<=i; j++)
```

serie aritmetica

Entrambi i loop $\Theta(n^2)$

Esempi – complessità tempo(3)

Esempio 5: Assumiamo n potenza del 2

```
sum1 = 0;  
for (k=1; k<=n; k*=2) // lg n volte  
    for (j=1; j<=n; j++) // n volte  
        sum1++;
```

$$T(n) = \sum_{i=0}^{\lg n} n = \Theta(n \lg n)$$

Esempi – complessità tempo(3)

Esempio 5: Assumiamo n potenza del 2

```
sum1 = 0;  
for (k=1; k<=n; k*=2) // lg n volte  
    for (j=1; j<=n; j++) // n volte  
        sum1++;  
  
sum2 = 0;  
for (k=1; k<=n; k*=2) // lg n volte  
    for (j=1; j<=k; j++) // k volte  
        sum2++;
```

$$T(n) = \sum_{i=0}^{\lg n} n = \Theta(n \lg n)$$

$$T(n) = \sum_{i=0}^{\lg n} 2^i = \Theta(n)$$

$\Theta(n \lg n)$

serie geometrica

Altre strutture di controllo

Altri loop: come il **for** loop

if-then-else: considera la complessità maggiore (max) delle due clausole **then/else**

switch: considera la complessità del caso più costoso (max)

Invocazione di una subroutine: complessità della subroutine



Complessità intrinseca di un problema

Complessità di un problema

Complessità intrinseca di un problema \neq
Complessità computazionale di un algoritmo

- Definizioni:

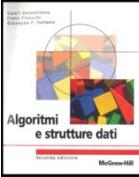
Un **problema computazionale** ha **complessità $O(f(n))$ (upper bound)** se esiste **un** algoritmo per la sua risoluzione con delimitazione superiore $O(f(n))$.

Un **problema computazionale** ha **complessità $\Omega(f(n))$ (lower bound)** se **tutti** gli algoritmi per la sua risoluzione hanno delimitazione inferiore $\Omega(f(n))$.

Algoritmi ottimali

Se dimostro che un problema ha **delimitazione inferiore** $\Omega(f(n))$ e trovo un **algoritmo** avente **delimitazione superiore** $O(f(n))$ allora, a meno di costanti, ho un **algoritmo ottimale** per risolvere il problema!!!

Esempio (Ricerca del minimo in un insieme non ordinato): Un algoritmo per la ricerca del minimo in un insieme non ordinato, avente complessità $O(n)$, è ottimo. Infatti, ogni algoritmo dovrà almeno leggere l'input, e quindi il problema avrà complessità inferiore $\Omega(n)$.



Metodi di analisi

Caso peggiore, migliore e medio

- Misureremo le risorse di calcolo usate da un algoritmo (tempo di esecuzione / occupazione di memoria) in funzione della dimensione n delle istanze di input
- Istanze diverse, a parità di dimensione, potrebbero però 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



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



Caso medio

- Sia $\mathcal{P}(I)$ la probabilità di avere in ingresso un’istanza I
- $T_{avg}(n) = \sum_{\text{istanze } I \text{ di dimensione } n} \{ \mathcal{P}(I) \text{ tempo}(I) \}$
- Intuitivamente, $T_{avg}(n)$ è il tempo di esecuzione nel caso medio, ovvero sulle istanze di ingresso “tipiche” per il problema
- Richiede conoscenza di una distribuzione di probabilità sulle istanze



Esempio 1

Ricerca di un elemento x in una lista \mathcal{L} non ordinata

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

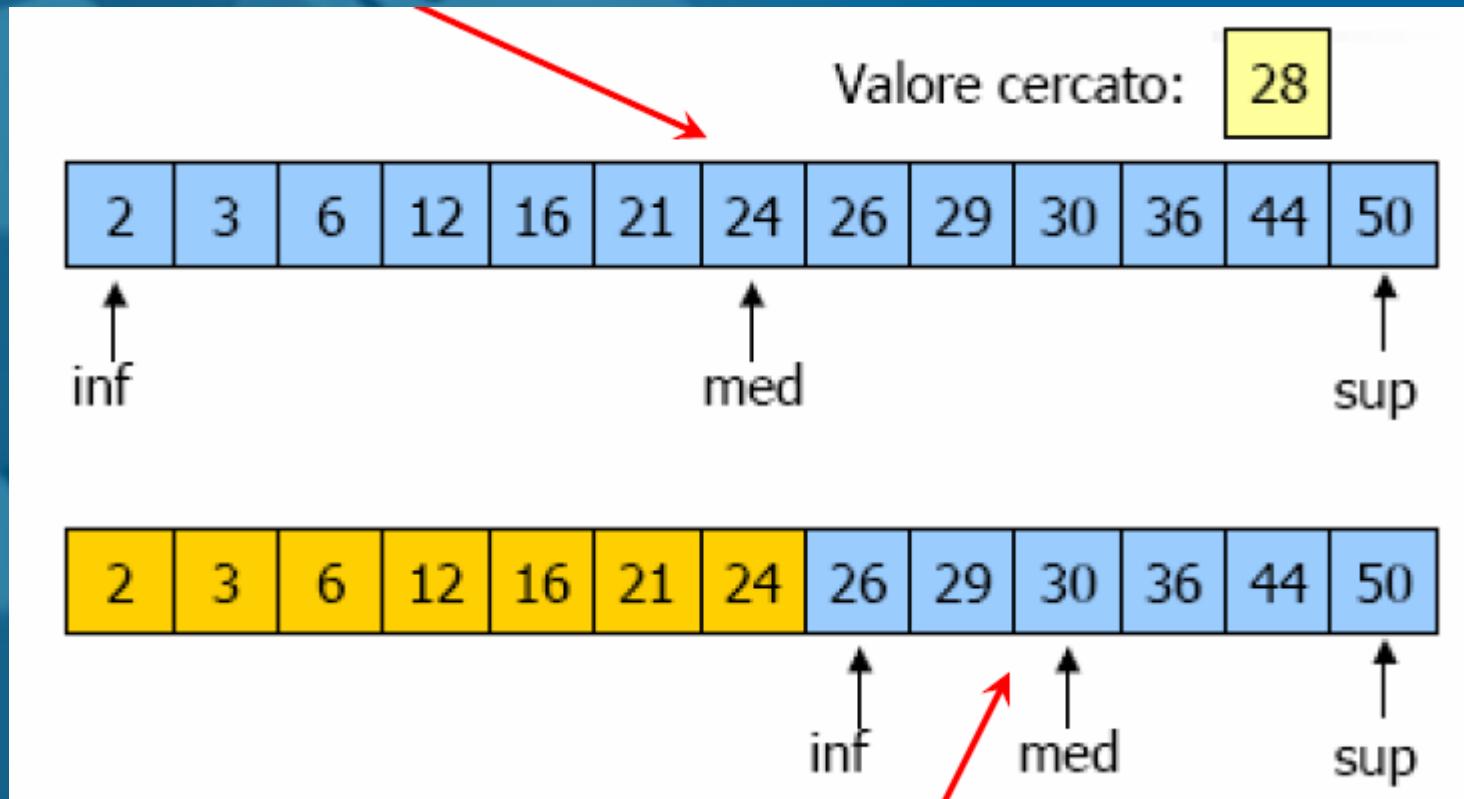
- ```

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

```

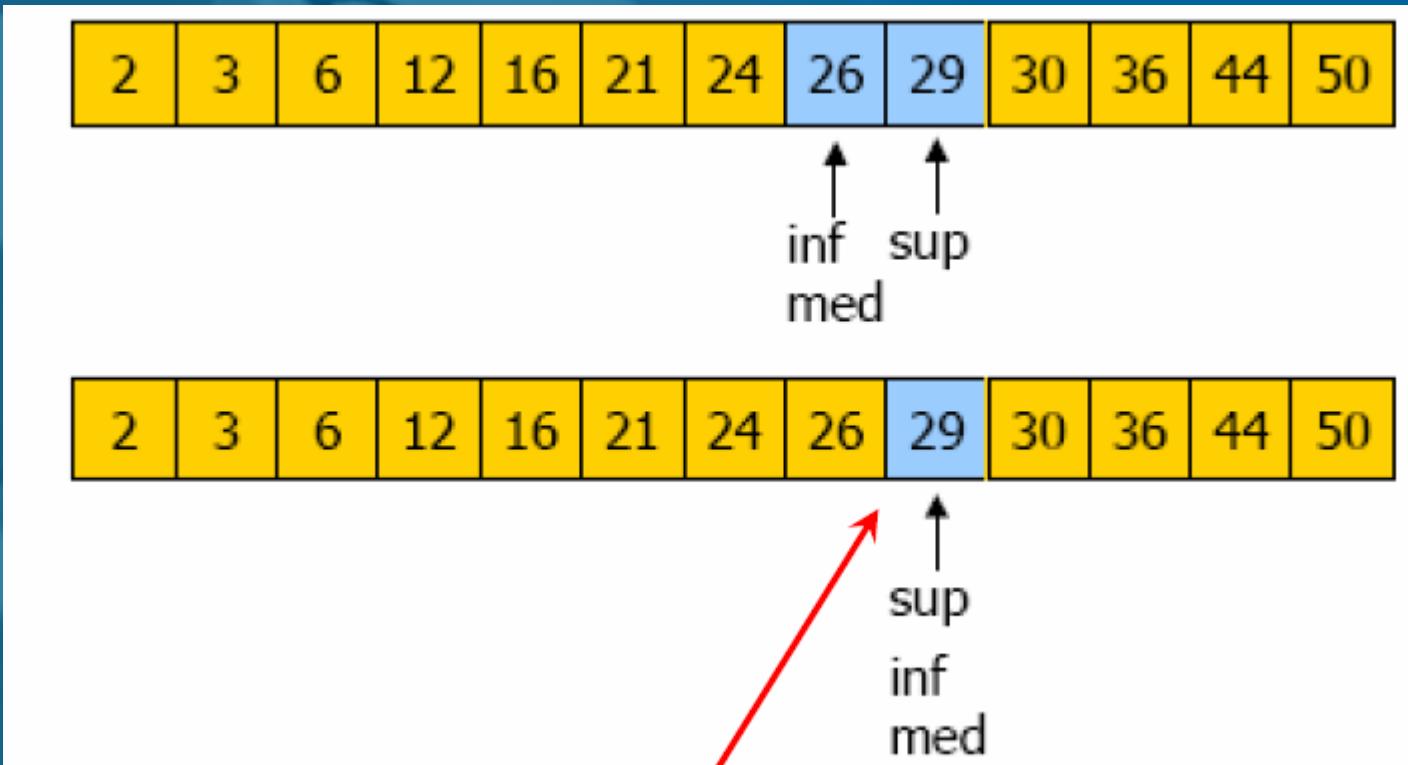
- $T_{\text{best}}(n) = 1$   $x \in \mathcal{L}$  è in prima posizione
  - $T_{\text{worst}}(n) = n$   $x \notin \mathcal{L}$  oppure è in ultima posizione
  - $T_{\text{avg}}(n) = (n+1)/2$  Assumendo che  $x \in \mathcal{L}$  e che le istanze siano equidistribuite:  
 $p\{\text{pos}(x)=i\} = 1/n$  per ogni  $i$  in  $[1, n]$

# Esempio 2: ricerca binaria (ricerca senza successo)

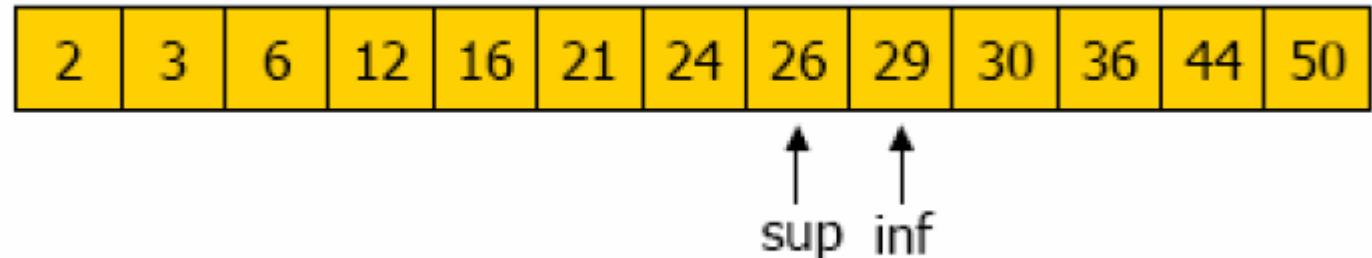


Assume che la sequenza di elementi sia ordinata

# Esempio 2: ricerca binaria



# Esempio 2: ricerca binaria



Elemento (28) non trovato!

# Esempio 2 (1/2)

## Ricerca binaria di un elemento in un array ordinato

```
algoritmo ricercaBinariaIter(array L, elem x) → booleano
1. a ← 1
2. b ← lunghezza di L
3. while (L[(a + b)/2] ≠ x) do ← Si assume
4. m ← (a + b)/2
5. if (L[m] > x) then b ← m - 1
6. else a ← m + 1
7. if (a > b) then return non trovato
8. return trovato
```

## Esempio 2 (2/2)

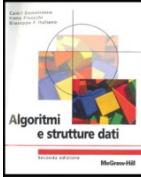
$T_{best}(n) = 1$  (costante) l'elemento centrale è uguale a x

$T_{worst}(n) = O(\lg n)$   $x \notin L$  oppure viene trovato all'ultimo confronto

Quante iterazioni nel ciclo while? Poiché la dimensione del sotto-array su cui si procede si dimezza dopo ogni confronto, dopo l'i-esimo confronto il sottoarray di interesse ha dimensione  $n/2^i$ .  
L'analisi assume n potenza del 2.

$T_{avg}(n) = \lg n - 1 + 1/n$  assumendo che le istanze siano equidistribuite

[Vedi Cap2 pag. 35 del libro Demetrescu et al.]



# 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 [Prossima lezione]

# **INFORMATICA III**

## **Parte B - Progettazione e Algoritmi**

### **Sommatorie**

- Proprietà
- Serie aritmetica
- Serie geometrica
- Serie armonica
- Serie telescopica

**Patrizia Scandurra** patrizia.scandurra@unibg.it

Università degli Studi di Bergamo

# Le funzioni floor e ceiling (parte intera e parte intera superiore)

- Per un numero reale  $x$ , la **parte intera di  $x$** , indicata con  $\lfloor x \rfloor$  è il più grande intero minore o uguale a  $x$ .
  - Per esempio  $\lfloor 2.9 \rfloor = 2$ ,  $\lfloor -2 \rfloor = -2$  e  $\lfloor -2.3 \rfloor = -3$
- Per un numero reale  $x$ , la **parte intera superiore di  $x$**  indicata con  $\lceil x \rceil$  è il più piccolo intero non minore di  $x$ .
  - Per esempio,  $\lceil 2,3 \rceil = 3$ ,  $\lceil 2 \rceil = 2$  e  $\lceil -2,3 \rceil = -2$

# Sommatorie

Quando un algoritmo contiene un costrutto di controllo iterativo come un ciclo while o for, il suo tempo di esecuzione può essere espresso come la somma dei tempi impiegati per ogni esecuzione del corpo del ciclo.

Data una sequenza di numeri  $a_1, a_2, \dots$  la **somma finita**  $a_1 + a_2 + \dots + a_n$  può essere scritta nel seguete modo:

$$\sum_{k=1}^n a_k$$

Se  $n=0$ , il valore della sommatoria è 0 per definizione.

Se  $n$  non è un intero si assume per definizione che il limite superiore sia  $\lfloor n \rfloor$

Se la somma comincia con  $k=x$ , dove  $x$  non è un intero, si assume che il valore iniziale sia  $\lfloor x \rfloor$

I termini della sommatoria possono essere sommati in qualsiasi ordine.

## Sommatorie

Data una sequenza di numeri  $a_1, a_2, \dots$  la **somma infinita**  $a_1 + a_2 + \dots$  può essere scritta nel seguete modo:

$$\sum_{k=1}^{\infty} a_k = \lim_{n \rightarrow \infty} \sum_{k=1}^n a_k$$

Se il limite non esiste la serie **diverge**, altrimenti **converge**.

## Proprietà di Linearità

Dato un qualunque numero reale  $c$  e due qualunque sequenze finite  $a_1, a_2, \dots, a_n$  e  $b_1, b_2, \dots, b_n$  allora

$$\sum_{k=1}^n (ca_k + b_k) = c \sum_{k=1}^n a_k + \sum_{k=1}^n b_k$$

La proprietà di linearità si applica anche a serie convergenti infinite e può inoltre essere impiegata per manipolare sommatorie contenenti termini di notazioni asintotiche.

$$\sum_{k=1}^n \Theta(f(k)) = \Theta\left(\sum_{k=1}^n f(k)\right)$$

## Serie aritmetica

```
Algo1(n)
 sum ← 0
 for i = 1 to n do
 for j = 1 to i do //i volte
 sum ++
```

La sommatoria che viene fuori dall'analisi di Algo1 è una serie aritmetica:

$$\sum_{i=1}^n i = 1 + 2 + \dots + n$$

$$\sum_{i=1}^n i = \frac{1}{2} n(n+1) = \Theta(n^2)$$

## Serie geometrica

Dato un reale  $x \neq 1$ , la sommatoria

$$\sum_{k=0}^n x^k = 1 + x + x^2 + \dots + x^n$$

è una *serie geometrica o esponenziale* ed ha come valore

$$\sum_{k=0}^n x^k = \frac{x^{n+1} - 1}{x - 1}$$

Se la serie è infinita e  $|x| < 1$ , si ha la *serie geometrica decrescente infinita*

$$\sum_{k=0}^{\infty} x^k = \frac{1}{1-x}$$

## Somma esponenziale

```
Algo2(n)
 key ← 1
 while key ≤ n do
 for i=1 to key do //key volte
 A[i] ++
 key ← key · 2
```

Due cicli annidati. Quante volte viene eseguito il ciclo più interno?

Passo 0 : key =  $2^0 \rightarrow$  1 volta

Passo 1 : key =  $2^1 \rightarrow$  2 volte

Passo 2 : key =  $2^2 \rightarrow$  4 volte

Passo  $k$  : key =  $2^k \rightarrow 2^k$  volte

Key assume i valori  $2^k$  fino a che  
key= $2^k = n$  (ciclo while esterno)

# Somma esponenziale

```
Algo2(n)
 key ← 1
 while key ≤ n do
 for i=1 to key do //key volte
 A[i] ++
 key ← key · 2
```

$$\begin{aligned} T(n) &= \sum_{k=0}^{\lg n} 2^k \\ &= \frac{2^{\lg n + 1} - 1}{2 - 1} \\ &= 2 \cdot 2^{\lg n} - 1 \\ &= 2n - 1 = \Theta(n) \end{aligned}$$

## Serie armonica

Dato l'intero positivo  $n$ , l' $n$ -esimo numero armonico è

$$H_n = 1 + \frac{1}{2} + \frac{1}{3} + \frac{1}{4} + \dots + \frac{1}{n} = \sum_{k=1}^n \frac{1}{k}$$

con valore

$$H_n = \sum_{k=1}^n \frac{1}{k} = \ln n + O(1)$$

## Serie telescopiche

Data una qualunque sequenza  $a_1, a_2, \dots, a_n$ , vale che

$$\sum_{k=1}^n (a_k - a_{k-1}) = a_n - a_1$$

Poiché ognuno dei termini della sotto-sequenza  $a_2, \dots, a_{n-1}$  è sia sommato che sottratto esattamente una volta.

Analogamente:

$$\sum_{k=1}^{n-1} (a_k - a_{k+1}) = a_0 - a_n$$

## Serie telescopiche

Come esempio di serie telescopica si consideri la sommatoria

$$\sum_{k=1}^{n-1} \frac{1}{k(k+1)}$$

Poiché ogni termine può essere scritto come

$$\frac{1}{k(k+1)} = \frac{1}{k} - \frac{1}{k+1}$$

si ha

$$\sum_{k=1}^{n-1} \frac{1}{k(k+1)} = \sum_{k=1}^{n-1} \left( \frac{1}{k} - \frac{1}{k+1} \right) = 1 - \frac{1}{n}$$

## Produttorie

Il prodotto finito di una sequenza di elementi  $a_1, a_2, \dots, a_n$ , può essere scritto come

$$\prod_{k=1}^n a_k$$

Se  $n=0$ , il valore del prodotto è 1 per definizione.

Si può convertire una formula contenente un prodotto in una formula contenente una sommatoria usando la seguente identità:

$$\lg\left(\prod_{k=1}^n a_k\right) = \sum_{k=1}^n \lg a_k$$

# Limitazioni sulle sommatorie

Molte tecniche disponibili per definire limiti sulle sommatorie che descrivono i tempi di esecuzione degli algoritmi.

Alcuni metodi usati più di frequente:

- Induzione matematica
- Limitazione dei termini
- Spezzare le sommatorie
- Approssimazione con integrali (*non trattato*)

# **Induzione matematica**

Il metodo base per calcolare il valore di una serie è di usare l'induzione matematica.

1. Si dimostra il passo base (per  $n=0$ , oppure  $n=1$ )
2. Si fa l'ipotesi induttiva che esso valga per  $n$
3. Si dimostra che vale per  $n+1$

# Induzione matematica

**Esempio:** dimostriamo che  $\sum_{k=1}^n k = \frac{1}{2}n(n+1)$

1. Passo base ( $n=1$ ):  $\sum_{k=1}^1 k = \frac{1}{2}1(1+1) = 1$

2. Supponiamo vero per  $n$ .

3. Dimostriamolo per  $n+1$ : 
$$\begin{aligned}\sum_{k=1}^{n+1} k &= \sum_{k=1}^n k + (n+1) \\ &= \frac{1}{2}n(n+1) + (n+1) \\ &= \frac{1}{2}(n+1)(n+2)\end{aligned}$$

# Induzione matematica

L'induzione può essere usata per **tentare un limite superiore**

**Esempio:** dimostriamo che  $\sum_{k=0}^n 3^k = O(3^n)$

o più precisamente dimostreremo che  $\sum_{k=0}^n 3^k \leq c3^n$

1. Passo base ( $n=0$ ):  $\sum_{k=0}^0 3^k = 1 = O(3^n)$

2. Supponiamo vero per  $n$ .

3. Dimostriamolo per  $n+1$ :  $\sum_{k=0}^{n+1} 3^k = \sum_{k=0}^n 3^k + 3^{n+1}$

$$\leq c3^n + 3^{n+1} \leq \left(\frac{1}{3} + \frac{1}{c}\right)c3^{n+1} \leq c3^{n+1}$$

## Limitazioni dei termini

Talvolta un buon limite superiore su una serie può essere ottenuto **maggiorando ogni termine della serie**, e spesso è sufficiente **usare il termine più grande** per limitare gli altri.

$$\sum_{k=1}^n k \leq \sum_{k=1}^n n = n^2$$

In generale:

$$a_{\max} = \max_{1 \leq k \leq n} a_k$$

$$\sum_{k=1}^n a_k \leq n a_{\max}$$

## Spezzare le sommatorie

Spesso si può spezzare (spezzando l'intervallo dell'indice) la sommatoria ottenuta dall'analisi di un algoritmo, **ignorando un numero costante di termini iniziali**. In generale si adotta questa tecnica **quando ogni termine  $a_k$  della sommatoria è indipendente da n**.

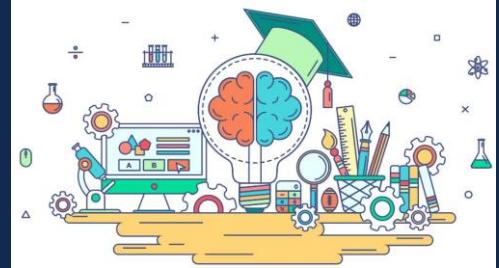
Per esempio, per qualunque  $k_0 > 0$  si può scrivere:

$$\begin{aligned} \sum_{k=1}^n a_k &= \sum_{k=1}^{k_0-1} a_k + \sum_{k=k_0}^n a_k \\ &= \Theta(1) + \sum_{k=k_0}^n a_k \end{aligned}$$



UNIVERSITÀ  
DEGLI STUDI  
DI BERGAMO

Dipartimento  
di Ingegneria Gestionale,  
dell'Informazione e della Produzione



# Implementazione di codice algoritmico in Java

PROGETTAZIONE, ALGORITMI E  
COMPUTABILITÀ  
(38090-MOD1)

Corso di laurea  
Magistrale in  
Ingegneria  
Informatica

RELATORE  
Prof.ssa Patrizia  
Scandurra

SEDE  
DIGIP

# Argomenti dell'esercitazione

- Implementare un algoritmo in Java
  - Interfacce, ereditarietà e polimorfismo
  - Tipi generici
  - Test di uguaglianza
  - Ordinamento naturale
  - Sequenze
    - (del *Java Collection Framework java.util*)
  - Iteratori, stream

# Interfaccia

- Stabilisce la struttura di un oggetto/componente o di astrazione di dato (*Abstract Data Type - ADT*), ma non un'implementazione!
- Un'interfaccia:
  - contiene i **prototipi dei metodi**, ma non i corpi degli stessi
  - può contenere **attributi**, ma queste sono implicitamente **static e final**
  - può essere dichiarata **public** (solo se definita in un file con lo stesso nome) o con visibilità di package

```
interface Instrument {
 // Compile-time constant:
 int i = 5; // static & final
 // Cannot have method definitions:
 void play();
 String what();
 void adjust();
}
```

# Interfacce, ereditarietà e polimorfismo

- Java fornisce solo **ereditarietà singola**
- In Java **una classe può** però ereditare da più interfacce -- Java supporta l'**ereditarietà multipla** tramite interfacce!
- Sintassi:

```
interface A { ... }
```

```
interface B { ... }
```

```
interface C { ... }
```

```
class MyClass implements A, B, C { ... }
```

- Si può ereditare da quante interfacce si vuole, ciascuna è un tipo indipendente verso il quale si può effettuare l'**upcasting (conversione larga)**

```
MyClass o = new MyClass(); A ao = new MyClass(); //upcasting
A ao = (A) o; //upcasting B bo = new MyClass(); //upcasting
B bo = (B) o; //upcasting
```

# Ereditarietà tra interfacce

Usando l'ereditarietà tra interfacce è possibile:

- aggiungere nuovi metodi alle interfacce
- combinare diverse interfacce fra loro in una nuova interfaccia

```
interface Monster { void menace(); }
```

```
interface DangerousMonster extends Monster {
 void destroy(); }
```

```
interface Lethal { void kill(); }
```

```
class DragonZilla implements DangerousMonster {
 public void menace() {...}
 public void destroy() {...} }
```

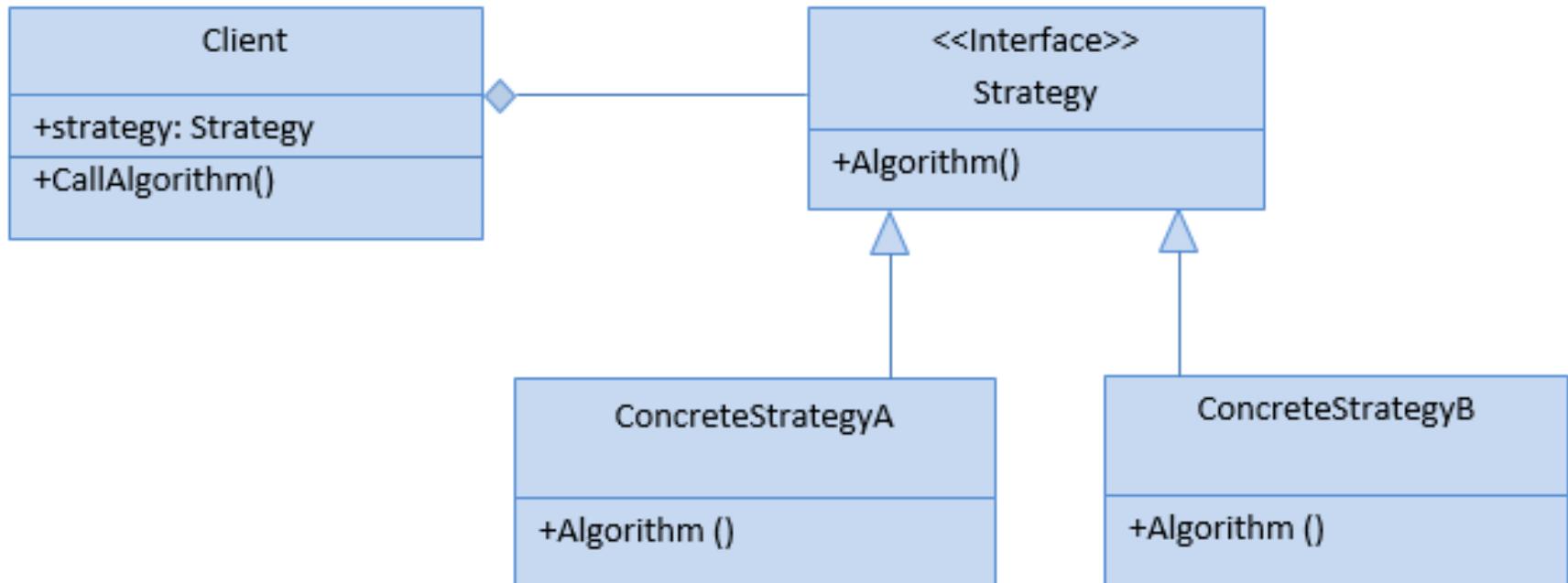
```
interface Vampire extends DangerousMonster,
Lethal {
 void drinkBlood(); }
```

```
public class HorrorShow {
 static void u(Monster b) { b.menace(); }
 static void v(DangerousMonster d) {
 d.menace();
 d.destroy(); }
 public static void main(String[] args) {
 Monster if2 = new DragonZilla();
 u(if2);
 v(if2); }
}
```

# Stessa interfaccia, diverse implementazioni

Ci aspettiamo che implementazioni algoritmi diversi per uno stesso problema computazionale condividano la medesima interfaccia.

*Soluzione: Design pattern **Strategy*** - le classi che implementano l'interfaccia rappresentano le *varianti* dell'algoritmo.



# Stessa interfaccia, diverse implementazioni

- **Definisci prima l'interfaccia:**

```
public interface AlgoDup {
 public boolean verificaDup(List S);
}
```

- **Per ogni possibile implementazione dell'algoritmo, definisci una classe che implementa l'interfaccia:**

```
public class verificaDupList implements AlgoDup {
 public boolean verificaDup(List S) {...}
}

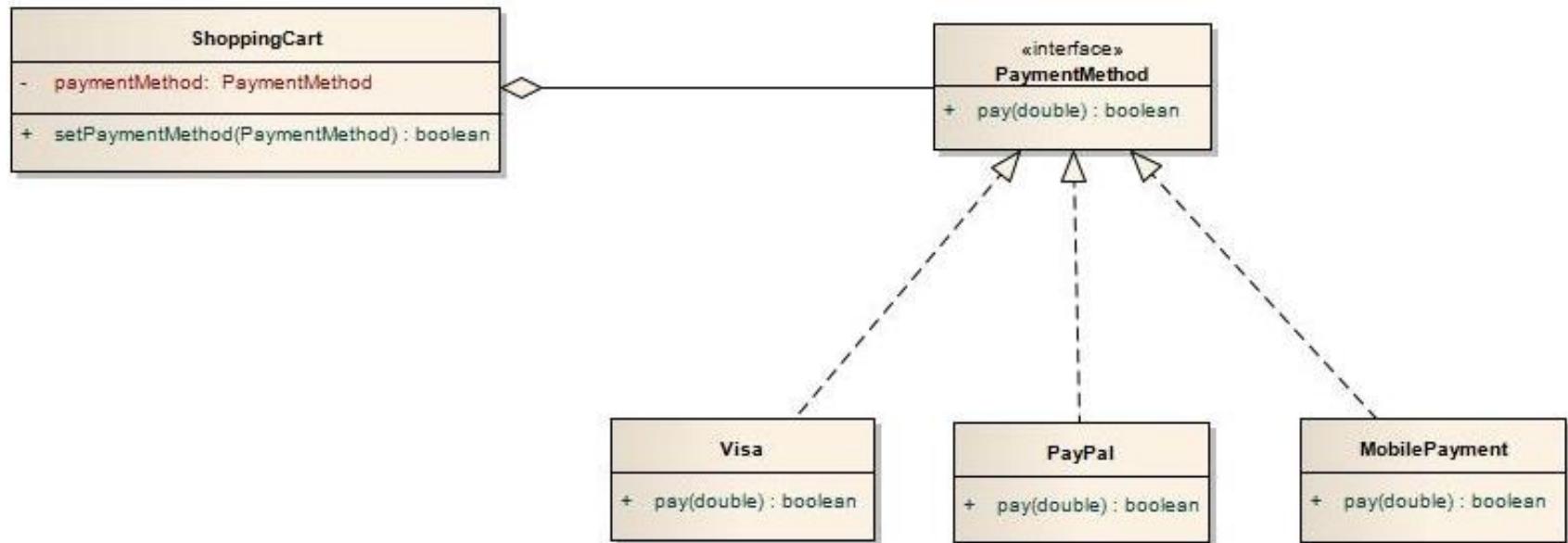
public class verificaDupOrdList implements AlgoDup {
 public boolean verificaDup(List S) {...}
}
```

- **Dichiara poi un oggetto usando come tipo l'interfaccia -- la scelta dell'algoritmo sarà così delegata all'operatore new (*upcasting*)**

```
AlgoDup myAlg = new VerificaDupOrdList();
boolean result = myAlg.verificaDup(S);
```

# Stessa interfaccia, diverse implementazioni

- Perché usiamo il pattern Strategy?
  - se cambia l'implementazione, non cambia l'interfaccia e quindi non cambiano le dipendenze della componente con il resto del programma (codice client)
  - consente di cambiare dinamicamente quale implementazione dell'algoritmo utilizzare a seconda delle diverse esigenze



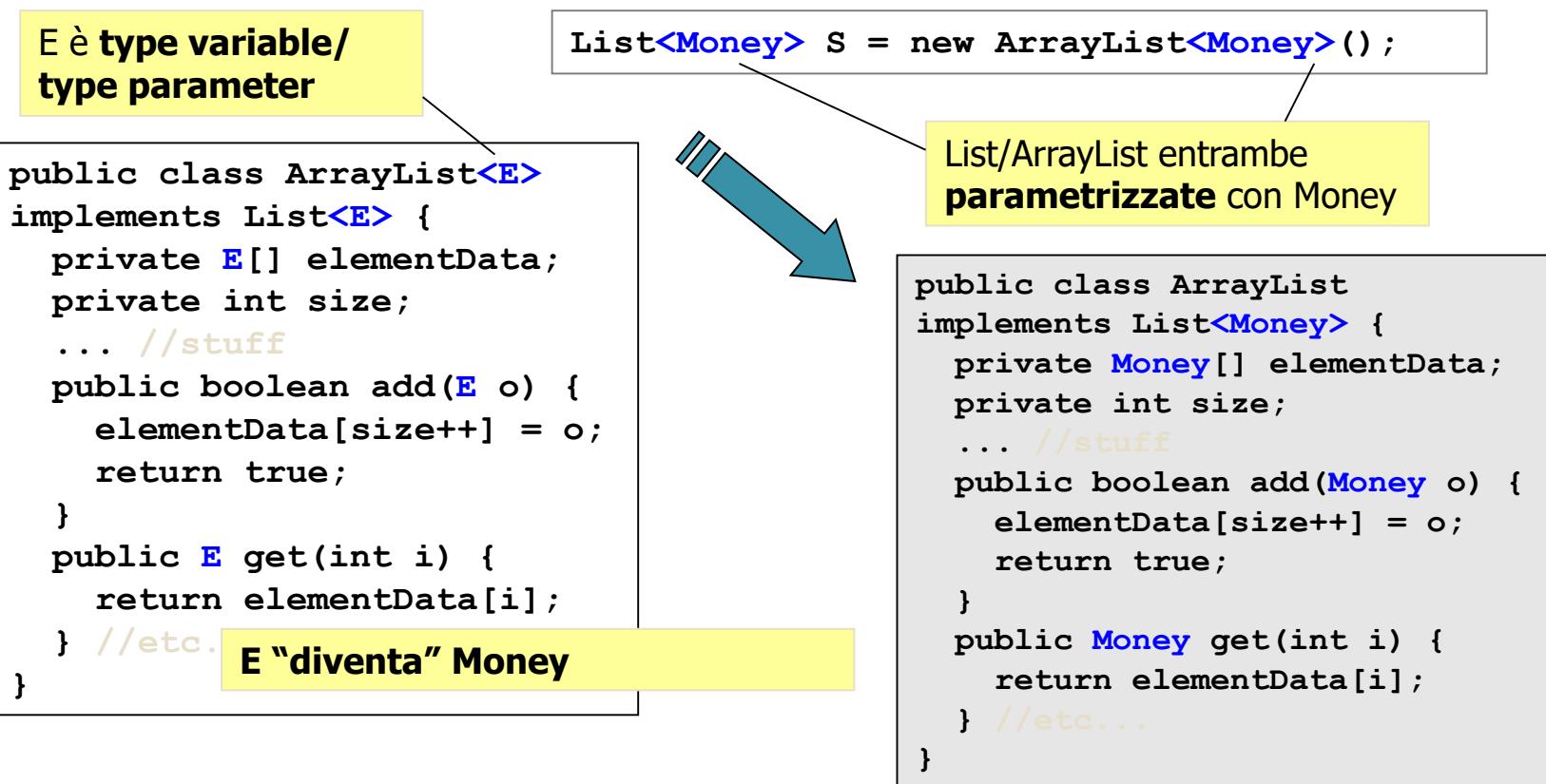
# Tipi generici e boxing

Java consente di definire tipi generici:

- Tramite la **classe Object**
  - Tipi primitivi inscatolati nelle **Classi wrapper** (Integer, Float, ..)
    - Gestione agevolata da Java 5 con l'**auto-boxing**:
      - Ad esempio, data una lista `List S = new LinkedList();` possiamo scrivere direttamente `S.add(5)` invece di `S.add(new Integer(5))` per aggiungere un elemento ad S
- Tramite i **tipi generici** (da Java 5):  
`List<Integer> myIntList = new LinkedList<Integer>();`  
equivale a livello di codice a dichiarare:  
`List myIntList = new LinkedList(); //Lista di Object`  
e ad eseguire implicitamente le conversioni Object->Integer e Integer->Object per leggere e scrivere gli elementi

# Recap sui tipi generici in Java

- `List<E>` è un **tipo generico** (ma anche una **interfaccia generica**)
- `ArrayList<E>` è una **classe generica** che **implements** `List<E>`



# Test di uguaglianza di oggetti

- Vogliamo il test dei “valori” non dei riferimenti!
  - **NON usare: equality (==) inequality (!=)**
- In Java, sfruttiamo il polimorfismo del **metodo equals()** ereditato dalla classe Object:

```
if (name.equals("Mickey Mouse")) ...
```

  - **va ridefinito in ogni (user) classe** (altrimenti, il comportamento di default è lo stesso di ==)
  - deve implementare una **relazione di equivalenza** tra riferimenti (non nulli) ad oggetti

# Test di uguaglianza di oggetti

```
public class Person {
 String title;
 String fullName;
 int age;
 public Person(String title, String fullName, int age) {
 this.title = title;
 this.fullName = fullName;
 this.age = age;
 }
 //Metodi accessori
 String getFullName() { return fullName; }
 ...
}
```

# Test di uguaglianza di oggetti

- **ATTENZIONE:** L'argomento del metodo equals deve essere Object (altrimenti sarebbe un overloading!)

```
public class Person {
 ...
 public boolean equals(Object obj) {
 if(this == obj) { return true; }
 if (!(obj instanceof Person)) { return false; }
 Person person = (Person)obj; //Cast
 return age == person.getAge() &&
 fullName.equals(person.getFullName())
 && title.equals(person.getTitle());
 }
 ...
}
```

# Ordinamento naturale con interfaccia Comparable

- **Confronto di tipi primitivi:** gli operatori relazionali < <= > = ==
- **Confronto di oggetti :** la classe dell'oggetto deve implementare l'interfaccia generica Comparable<T> o Comparable (non generica) e implementare il metodo **compareTo**

```
public interface Comparable<T>{
 int compareTo(T o);
}
//Uso con a e b oggetti dello stesso tipo:
a.compareTo(b); //return < 0, > 0, == 0
```

# Ordinamento naturale: esempio

Definire **compareTo** dell’interfaccia **Comparable<T>**

- Non servono cast
- Si noti l’uso “ricorsivo” di compareTo

```
class Person implements Comparable <Person> { ...
public int compareTo(Person another) {
 if (this.fullname.compareTo(another.getFullscreen())<0)
 return -1;
 if (this.fullname.compareTo(another.getFullscreen())>0)
 return 1;
 return this.age - another.getAge();
}
}
```

# Ordinamento naturale: esempio

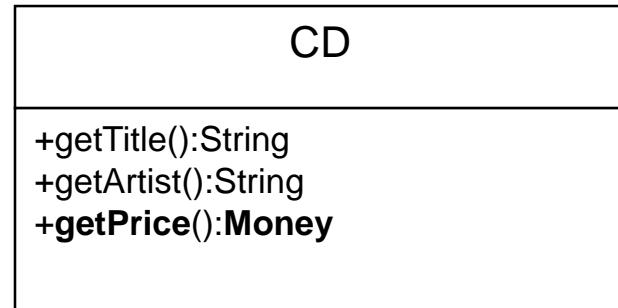
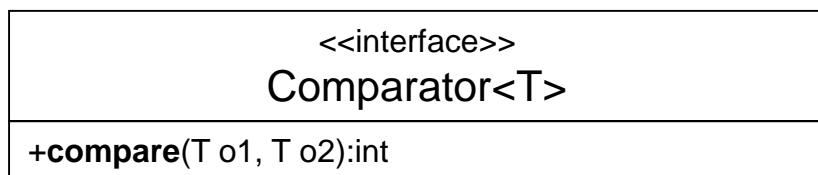
Definire il **compareTo** dell'interfaccia **Comparable**

- Occorre un cast!

```
class Person implements Comparable { ...
public int compareTo(Object another) throws
 ClassCastException {
 if (!(another instanceof Person)) throw new
 ClassCastException("A Person object expected.");
 Person anotherP = (Person) another; //cast
 if (this.fullname.compareTo(anotherP.getFullscreenname())<0)
 return -1;
 if (this.fullname.compareTo(anotherP.getFullscreenname())>0)
 return 1;
 return this.age - anotherP.getAge();
}
```

# Ordinamento naturale con interfaccia Comparator

- Ulteriore metodo: definire un oggetto che implementa l'interfaccia **java.util.Comparator<T>** per confrontare elementi di una certa natura



```
public class PriceComparator
implements Comparator<CD> {
 public int compare(CD c1, CD c2) {
 return c1.getPrice().compareTo(c2.getPrice());
 }
}
```

# Gestione degli errori

- Un algoritmo tipicamente assume che i dati in ingresso rispettino certe condizioni (*precondizioni*)
  - Ad es. un metodo che calcola la radice quadrata di un numero, assume che tale numero sia  $\geq 0$
- Se ciò non avviene, l'esecuzione del metodo non può essere (e non deve essere!) avviata e/o portata a termine
- **Java offre:**
  - Il meccanismo delle eccezioni per alterare il normale flusso del controllo di un programma
  - L'istruzione **assert expr;** permette di verificare se una data espressione è vera o falsa; se falsa viene sollevata un'eccezione di tipo *AssertionError*
    - Utile per implementare invarianti: *pre-condizioni, post-condizioni, invarianti interne, invarianti di classe*
    - Di default, sono disabilitate; vanno abilitate con il comando `java` che avvia l'applicazione

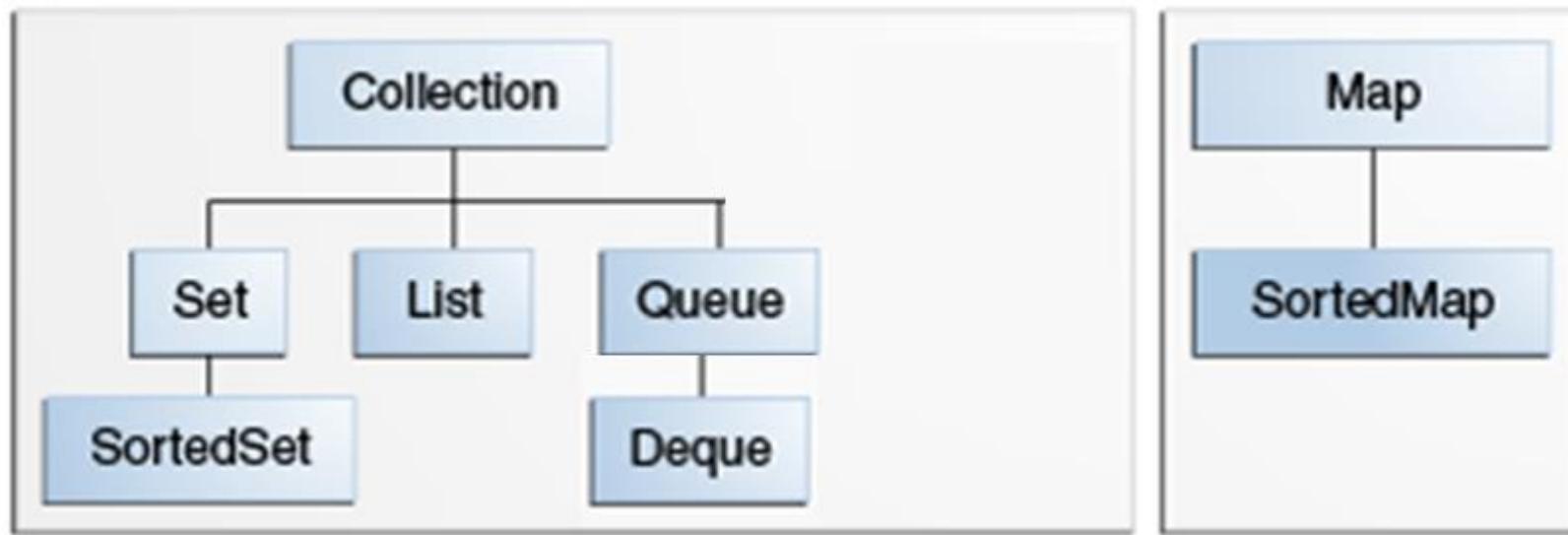
# Esercizio 1

- Definire una user-classe **Studente** che ridefinisce i metodi equals() e compareTo()

# Il Java Collection Framework (JCF)

- **Interfacce e classi del package java.util**
- Forniscono **ADT** e **strutture dati** per manipolare **collezioni di oggetti**
- E **algoritmi di base** (ad es. come ordinamento e ricerca)
- <http://docs.oracle.com/javase/tutorial/collections/index.html>

# JCF – le interfacce

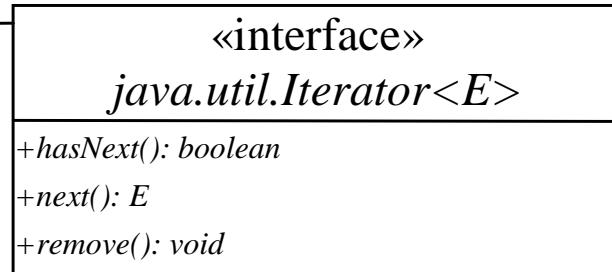
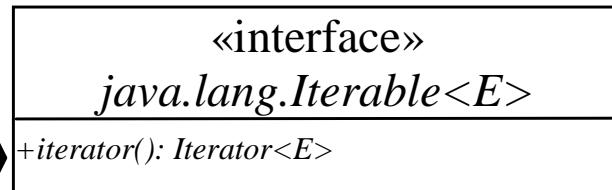


- Formano una gerarchia
- Tutte le **interfacce** sono **generiche**. Ad esempio la definizione dell'interfaccia Collection:  
**public interface Collection<E>...**

# L'interfaccia e classi predefinite per le sequenze

- **Collection:** nessuna ipotesi sul tipo di collezione; no implementazioni dirette
- **List:** introduce l'idea di sequenza (collezione ordinata, possibili duplicati)
- Classi per l'interfaccia **List**: **ArrayList**, **LinkedList**, **Vector**, **Stack**

# Interfaccia Collection



Returns an iterator for the elements in this collection.

Adds a new element o to this collection.

Adds all the elements in the collection c to this collection.

Removes all the elements from this collection.

Returns true if this collection contains the element o.

Returns true if this collection contains all the elements in c.

Returns true if this collection is equal to another collection o.

Returns the hash code for this collection.

Returns true if this collection contains no elements.

Removes the element o from this collection.

Removes all the elements in c from this collection.

Retains the elements that are both in c and in this collection.

Returns the number of elements in this collection.

Returns an array of Object for the elements in this collection.

Returns true if this iterator has more elements to traverse.

Returns the next element from this iterator.

Removes the last element obtained using the next method.

# Interfaccia List

«interface»  
*java.util.Collection<E>*



«interface»  
*java.util.List<E>*

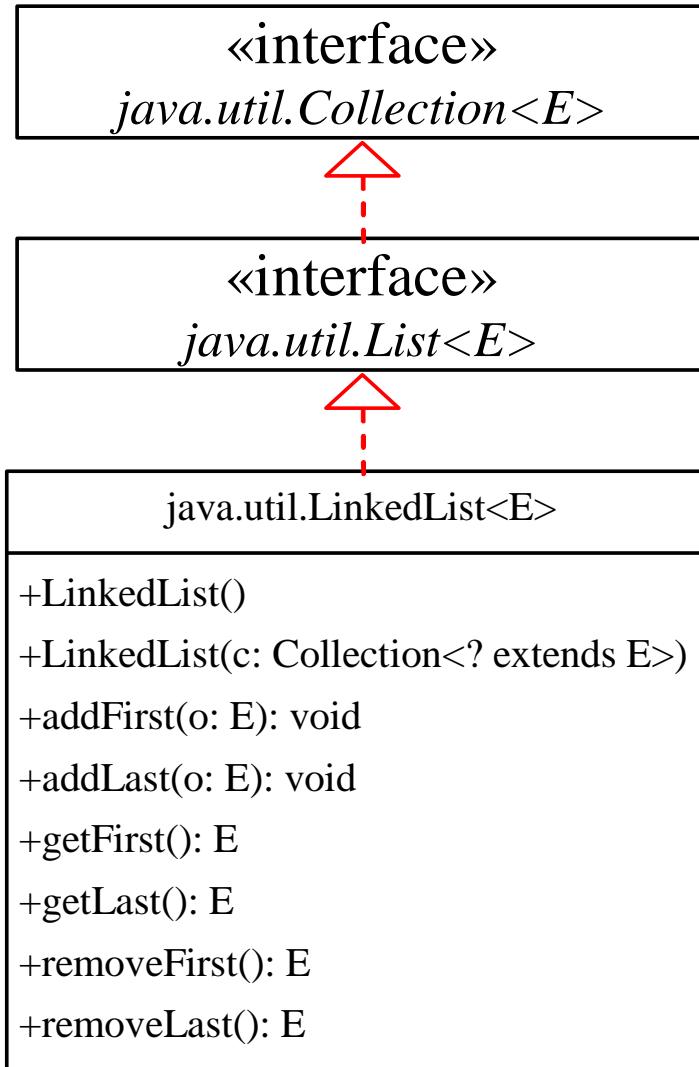
+*add(index: int, element:E): boolean*  
+*addAll(index: int, c: Collection<? extends E>): boolean*  
+*get(index: int): E*  
+*indexOf(element: Object): int*  
+*lastIndexOf(element: Object): int*  
+*listIterator(): ListIterator<E>*  
+*listIterator(startIndex: int): ListIterator<E>*  
+*remove(index: int): E*  
+*set(index: int, element: E): E*  
+*subList(fromIndex: int, toIndex: int): List<E>*

Sono presenti i metodi per  
**l'accesso posizionale!**

Adds a new element at the specified index.  
Adds all the elements in c to this list at the specified index.  
Returns the element in this list at the specified index.  
Returns the index of the first matching element.  
Returns the index of the last matching element.  
Returns the list iterator for the elements in this list.  
Returns the iterator for the elements from startIndex.  
Removes the element at the specified index.  
Sets the element at the specified index.  
Returns a sublist from fromIndex to toIndex.

# La classe `LinkedList`

- Rappresenta una lista **doppiamente concatenata**



Creates a default empty linked list.

Creates a linked list from an existing collection.

Adds the object to the head of this list.

Adds the object to the tail of this list.

Returns the first element from this list.

Returns the last element from this list.

Returns and removes the first element from this list.

Returns and removes the last element from this list.

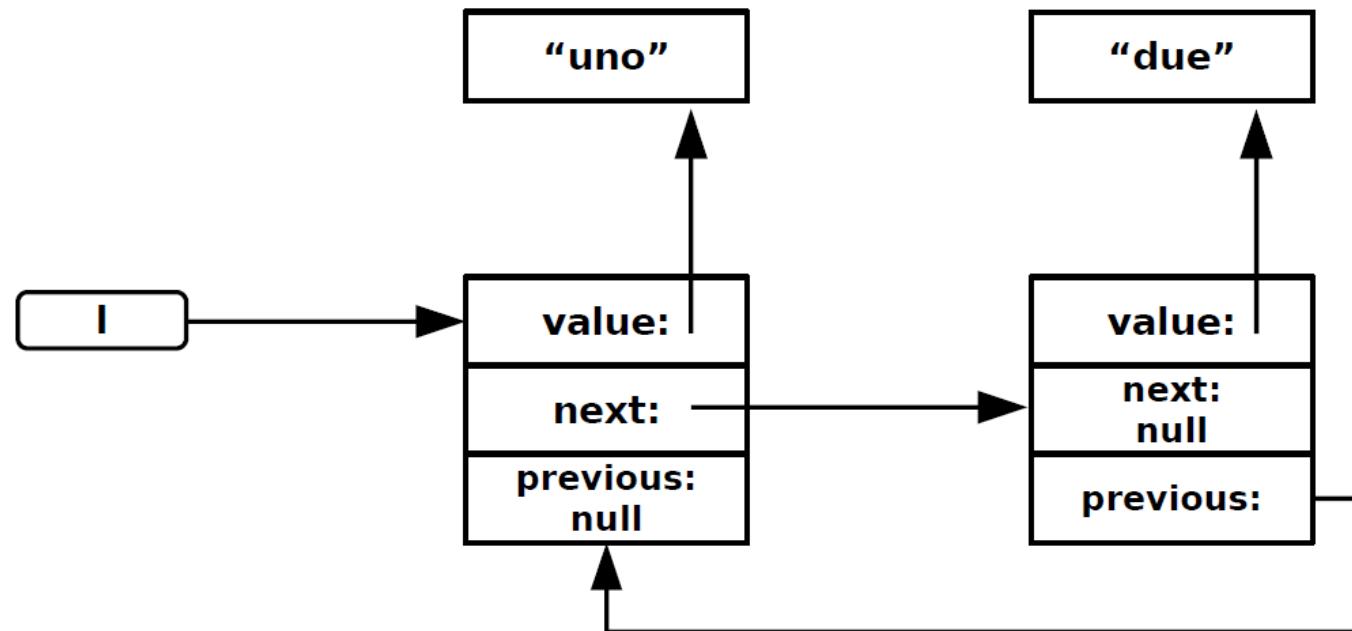
# Esempio LinkedList

```
LinkedList<String> l = new LinkedList<String>();
```

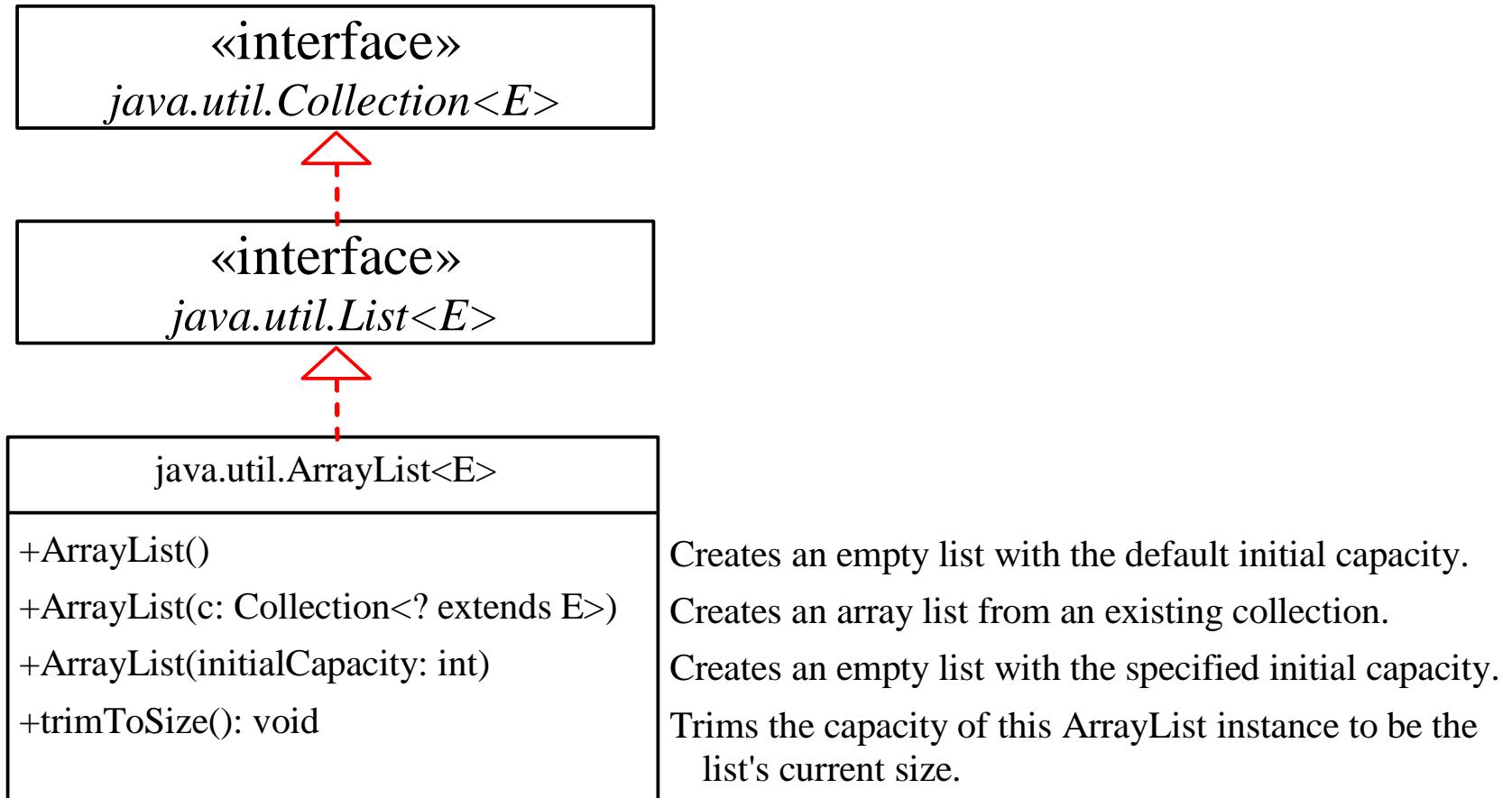
```
l.add("uno");
l.add("due");
```

---

**Memory layout:**



# La classe ArrayList

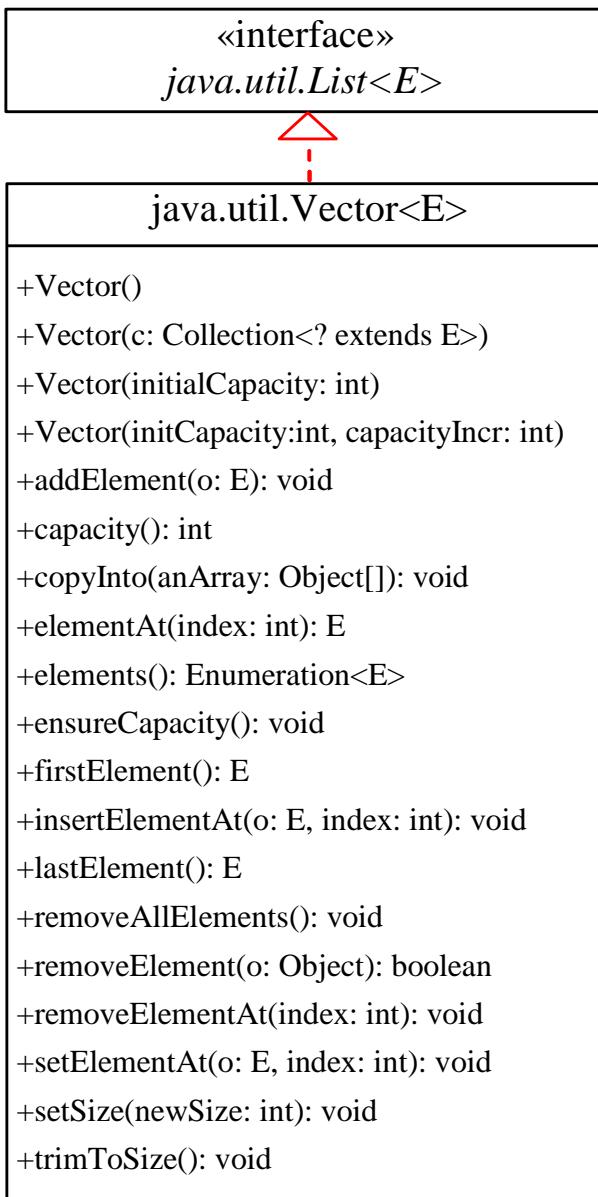


- `ArrayList` realizza `List` con un **array di dimensione dinamica**
- Il ridimensionamento avviene in modo che l'operazione di inserimento (`add`) abbia complessità *ammortizzata* costante

# Le liste e l'accesso posizionale

- In LinkedList, ciascuna operazione di accesso posizionale può richiedere un tempo proporzionale alla lunghezza della lista
  - per accedere all'elemento di posto  $i$  è necessario scorrere la lista, a partire dalla testa o dalla coda, fino a raggiungere la posizione desiderata in  $O(i)$  passi
- In ArrayList, ogni operazione di accesso posizionale richiede tempo costante –  $O(1)$  passi
- Pertanto, è **fortemente sconsigliato utilizzare l'accesso posizionale su LinkedList**

# La classe Vector



- Simile ad ArrayList ma Vector contiene la versione synchronized dei metodi per accedere e modificare l'array

|                                              |                                                                     |
|----------------------------------------------|---------------------------------------------------------------------|
| +Vector()                                    | Creates a default empty vector with initial capacity 10.            |
| +Vector(c: Collection<? extends E>)          | Creates a vector from an existing collection.                       |
| +Vector(initialCapacity: int)                | Creates a vector with the specified initial capacity.               |
| +Vector(initCapacity:int, capacityIncr: int) | Creates a vector with the specified initial capacity and increment. |
| +addElement(o: E): void                      | Appends the element to the end of this vector.                      |
| +capacity(): int                             | Returns the current capacity of this vector.                        |
| +copyInto(anArray: Object[]): void           | Copies the elements in this vector to the array.                    |
| +elementAt(index: int): E                    | Returns the object at the specified index.                          |
| +elements(): Enumeration<E>                  | Returns an enumeration of this vector.                              |
| +ensureCapacity(): void                      | Increases the capacity of this vector.                              |
| +firstElement(): E                           | Returns the first element in this vector.                           |
| +insertElementAt(o: E, index: int): void     | Inserts o to this vector at the specified index.                    |
| +lastElement(): E                            | Returns the last element in this vector.                            |
| +removeAllElements(): void                   | Removes all the elements in this vector.                            |
| +removeElement(o: Object): boolean           | Removes the first matching element in this vector.                  |
| +removeElementAt(index: int): void           | Removes the element at the specified index.                         |
| +setElementAt(o: E, index: int): void        | Sets a new element at the specified index.                          |
| +setSize(newSize: int): void                 | Sets a new size in this vector.                                     |
| +trimToSize(): void                          | Trims the capacity of this vector to its size.                      |

# Iterare una collezione

Esistono 3 modi:

- 1. Il costrutto foreach**
- 2. Iteratori**
- 3. Streaming API (da java 8)**

# Il ciclo for-each

- In generale, il ciclo for-each funziona su tutti gli oggetti che implementano l'interfaccia `Iterable<E>`
- Oltre che per gli array, il ciclo for-each funziona anche sulle collezioni

```
String[] array = {"uno", "due", "tre"}; for (Object o : collection)
for (String s: array) System.out.println(o);
 System.out.println(s);
```

`for (Object x : coll) { /* operazioni su x */ }`

equivale a:

```
for (Iterator i =coll.iterator(); i.hasNext();)
 {/* operazioni su x = i.next() */}
```

# Iteratori

- Oggetti (delle API JCF) associati ad una collezione che permettono di effettuare l'operazione di visita degli elementi della collezione in modo efficiente
- Per ottenere un iteratore per una data collezione, si invoca su essa l'apposito metodo **iterator()**
- Ogni iteratore offre:
  - un metodo **next()** che restituisce "il prossimo" elemento
  - un metodo **hasNext()** per sapere se ci sono altri elementi

```
public interface Iterator {
 boolean hasNext();
 Object next();
 void remove(); // operazione opzionale
}
```

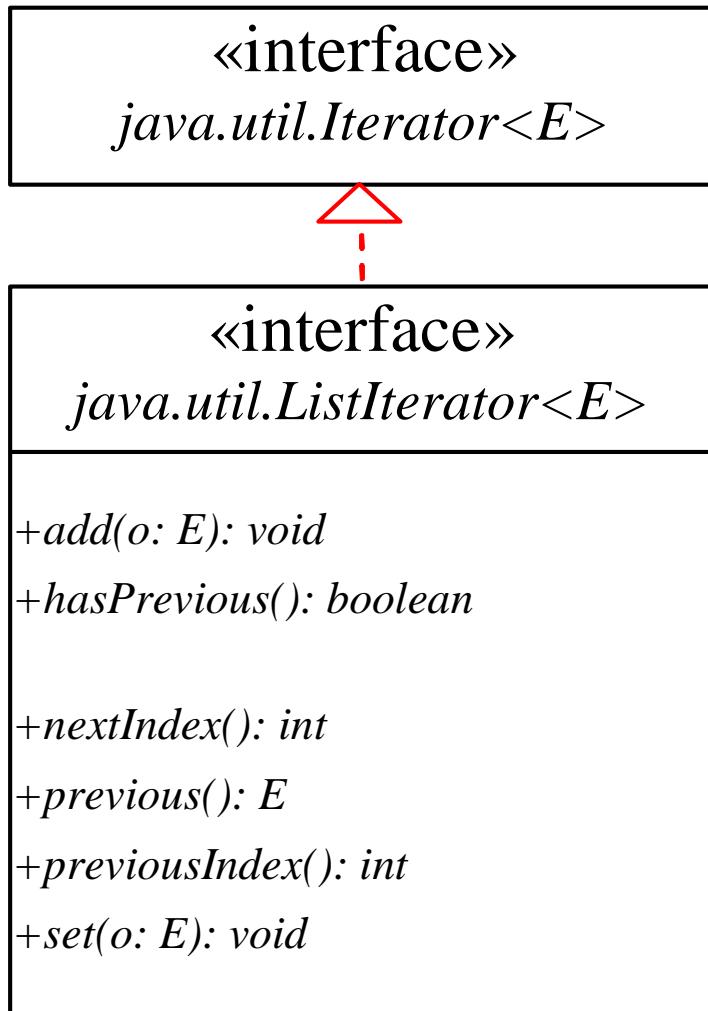
# Iteratori: esempio

```
Iterator i = S.iterator(); //S è una qualunque
 // collezione delle API JCF
while(i.hasNext()){ //se c'è ancora un elemento...
 //preleva l'elemento con i.next()
 Object e = i.next();
 //e lo utilizza
 System.out.println(e);
}
```

- Per usare un **iteratore generico <T>** (next() ha tipo T)

```
Iterator<String> i = S.iterator();
String e = i.next();
```

# ListIterator



<https://docs.oracle.com/en/java/javase/21/docs/api/java.base/java/util/List.html>

Adds the specified object to the list.

Returns true if this list iterator has more elements when traversing backward.

Returns the index of the next element.

Returns the previous element in this list iterator.

Returns the index of the previous element.

Replaces the last element returned by the previous or next method with the specified element.

# Streaming API

- Usano **IOC (*Inversion Of Control*)**: è il framework a controllare l'iterazione e non il programmatore
- L'interfaccia *Stream* è definita nel package *java.util.stream*
- Ampio uso di *espressioni lambda* e di funzioni di aggregazione definite su stream

```
myShapesCollection.stream()
 .filter(e -> e.getColor() == Color.RED)
 .forEach(e -> System.out.println(e.getName()));
```

```
myShapesCollection.parallelStream()
 .filter(e -> e.getColor() == Color.RED)
 .forEach(e ->
 System.out.println(e.getName()));
```

# Esercizio 2: scrittura di codice algoritmico

- Fornire un’implementazione dell’algoritmo **verificaDup** per il problema dei *Duplicati*
  - Scegliere se implementare l’iterazione usando oggetti Iterator o non
- Testare l’algoritmo su una collezione  $S$  di oggetti “studenti”

```
algoritmo verificaDup(sequenza S)
 for each elemento x della sequenza S do
 for each elemento y che segue x nella sequenza S do
 if $x = y$ then return true
 return false
```



# Esercizio 3

- Si consideri il tipo di dato astratto 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  $S$ ).

- Implementare il tipo di dato astratto Pila sfruttando una struttura dati di tipo *List* del JCF

—Due possibili modi:

**1. Per composizione:** la lista di elementi è un attributo della classe che implementa la Pila

**2. Per ereditarietà:** la classe che implementa la Pila estende una classe del JCF che implementa List (LinkedList, ArrayList, o Vector)

# Algoritmi e Strutture Dati

## Capitolo 3 Strutture dati elementari

Camil Demetrescu, Irene Finocchi,  
Giuseppe F. Italiano

# Gestione di collezioni di oggetti

## Tipo di dato (Abstract Data Type): cosa?

- Specifica delle operazioni di interesse su una collezione di oggetti (es. inserisci, cancella, cerca)
- **In Java:** un'interfaccia

## Struttura dati: come?

- Organizzazione dei dati che permette di supportare le operazioni di un tipo di dato usando meno risorse di calcolo possibile
- **In Java:** una classe che implementa l'interfaccia associata al tipo di dato



# 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 Dizionario – in Java

PRIMO modo: Object + Comparable

```
public interface Dizionario {
 public void insert(Object e, Comparable k);
 public void delete(Comparable k);
 public Object search(Comparable k);
};
```

# Il tipo di dato Dizionario – in Java

SECONDO modo: tipo generico + Comparable<T>

```
public interface Dizionario<E, K extends Comparable <? Super K>> {
 public void insert(E e, K k);
 public void delete(K k); // Null if none
 public E search(K k); // Null if none
};
```



# Tecniche di rappresentazione dei dati

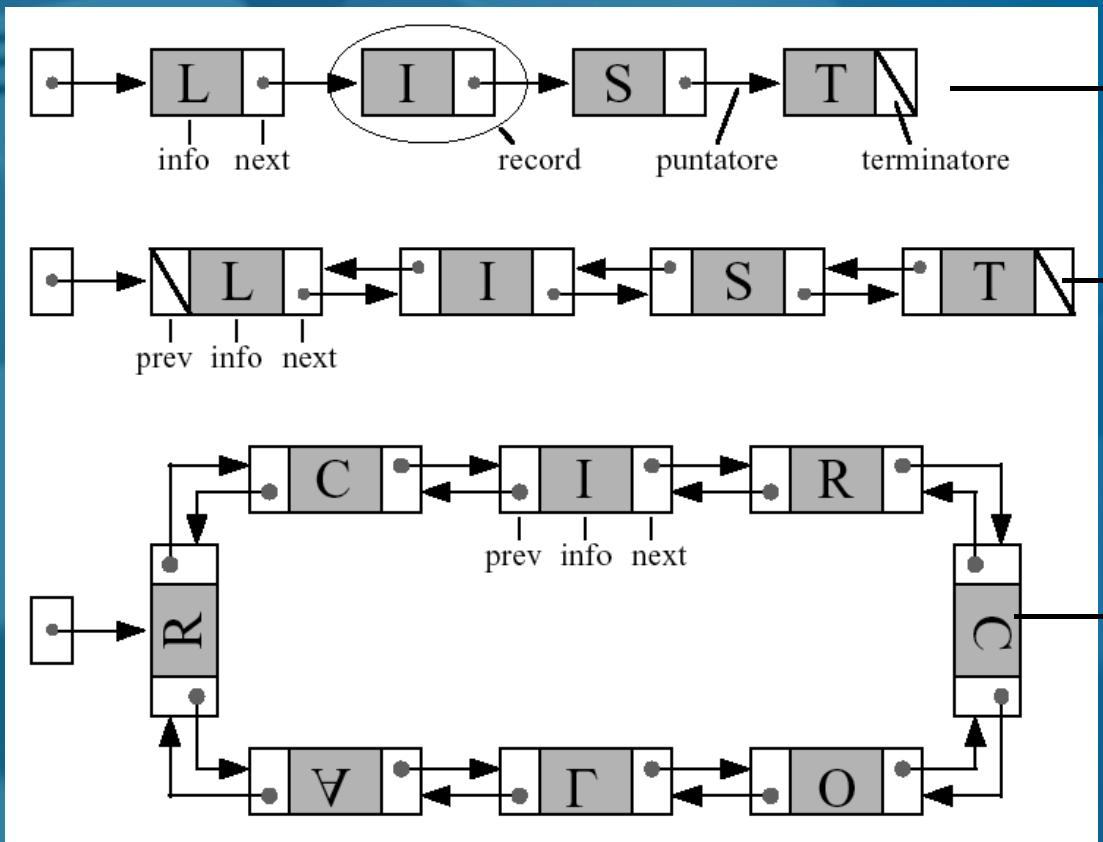
Rappresentazioni indicizzate:

- I dati sono contenuti in array

Rappresentazioni collegate:

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

# Esempi di strutture collegate



Lista semplice

Lista doppiamente  
collegata

Lista circolare  
doppiamente  
collegata



# 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

# Dizionario realizzato mediante array ordinato

classe `ArrayOrdinato` implementa `Dizionario`:

dati:

$$S(n) = \Theta(n)$$

un array  $S$  di dimensione  $n$  contenente coppie  $(elem, chiave)$ .

operazioni:

`insert(elem e, chiave k)`  $T(n) = O(n)$

rialloca l'array  $S$  aumentandone la dimensione  $n$  di uno; cerca il più piccolo indice  $i$  tale che  $k \leq S[i].chiave$  e pone  $S[j] \leftarrow S[j - 1]$  per ogni  $j$  da  $n - 1$  a  $i + 1$ ; infine, pone  $S[i] \leftarrow (e, k)$ .

`delete(chiave k)`  $T(n) = O(n)$

trova l'indice  $i$  della coppia con chiave  $k$  in  $S$  e pone  $S[j] \leftarrow S[j+1]$  per ogni  $j$  da  $i$  a  $n - 2$ ; infine, rialloca l'array  $S$  diminuendone la dimensione  $n$  di uno.

`search(chiave k) → elem`  $T(n) = O(\log n)$

esegue l'algoritmo di ricerca binaria su  $S$  per verificare se  $S$  contiene la chiave  $k$ . Se la ricerca ha successo restituisce l'elemento  $e$  associato alla chiave, altrimenti restituisce `null`.

# Dizionario mediante lista circolare

**classe StrutturaCollegata implementa Dizionario:**

**dati:**

$$S(n) = \Theta(n)$$

una collezione di  $n$  record contenenti ciascuno una quadrupla  $(elem, chiave, next, prev)$ , dove  $next$  e  $prev$  sono puntatori al successivo e precedente record nella collezione, rispettivamente. Manteniamo inoltre un puntatore  $list$  che contiene l'indirizzo di un record se la collezione non è vuota, e `null` altrimenti.

**operazioni:**

**insert(*elem e, chiave k*)**

$$T(n) = O(1)$$

viene creato un record  $p$  con elemento  $e$  e chiave  $k$ . Se  $list = null$ , si effettua  $p.next \leftarrow p$ ,  $p.prev \leftarrow p$  e  $list \leftarrow p$ . Altrimenti, si collega il record  $p$  tra  $list$  e  $list.next$  effettuando  $p.next \leftarrow list.next$ ,  $list.next.prev \leftarrow p$ ,  $p.prev \leftarrow list$  e  $list.next \leftarrow p$ .

**delete(*chiave k*)**

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

si trova il record  $p$  con chiave  $k$  come nella **search**; poi, si effettua  $p.prev.next \leftarrow p.next$  e  $p.next.prev \leftarrow p.prev$ ; infine, viene distrutto il record  $p$ .

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

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

se  $list = null$  si restituisce `null`. Altrimenti, si scandisce la struttura saltando di record in record con  $p \leftarrow p.next$  fino a quando non diventa  $p = list$ , verificando se qualche  $p$  ha chiave  $k$ . In caso positivo si restituisce l'elemento trovato, e `null` altrimenti.



# Il tipo di dato Pila

LIFO: Last In, First Out.

**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  $S$ ).

# Implementazione di una Pila mediante array

```
private int maxSize; //Max dimensione
private int top; //Indice elemento top
private Object [] listArray;
```

Questioni:

- Quale estremo dell'array è il top?
  - L'ultimo elemento
- Quali sono i costi delle operazioni?
  - O(1)

# Implementazione di una Pila mediante linked list

```
class LPila implements Pila {
 private Record top; //primo elemento della lista
 private int size;
 ...
}
```

## Questioni:

- Quali sono i costi delle operazioni?
  - $O(1)$  inserendo e cancellando in testa
- Quanto spazio richiede rispetto all'implementazione array-based?
  - Numero effettivo di elementi



## FIFO: First in, First Out

# 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  $S$ .

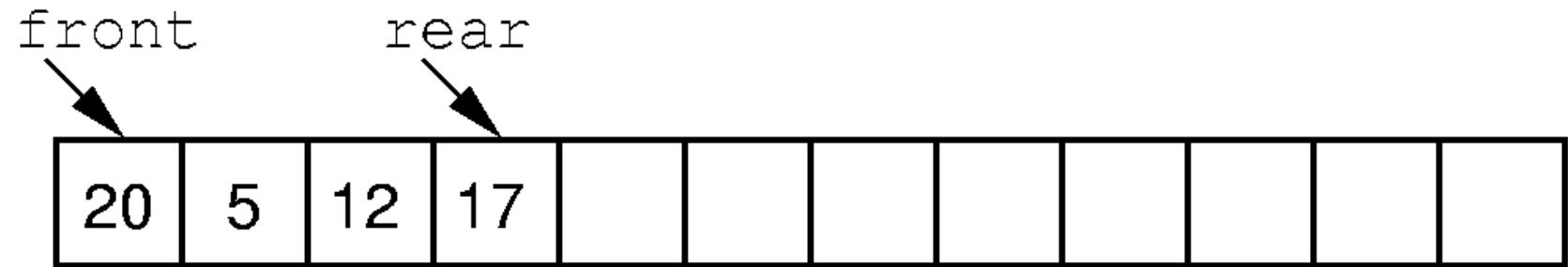
`dequeue() → elem`

toglie da  $S$  il primo elemento e lo restituisce.

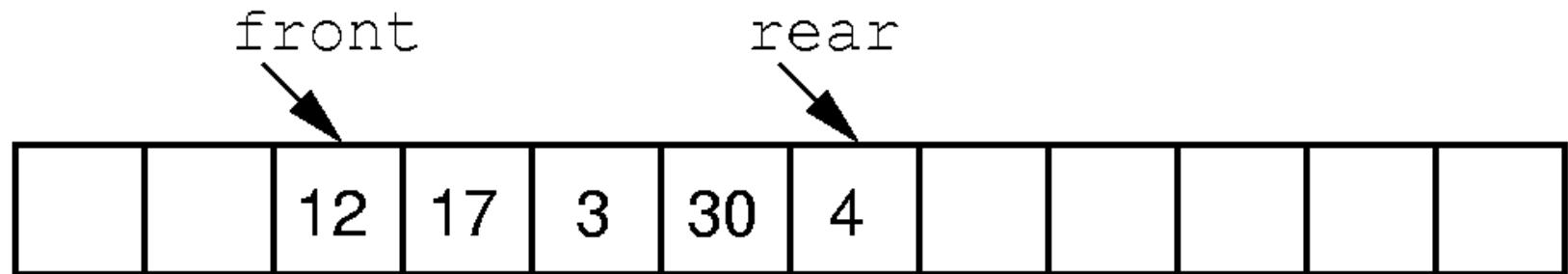
`first() → elem`

restituisce il primo elemento di  $S$  (senza toglierlo da  $S$ ).

# Implementazione di una coda mediante array(1)



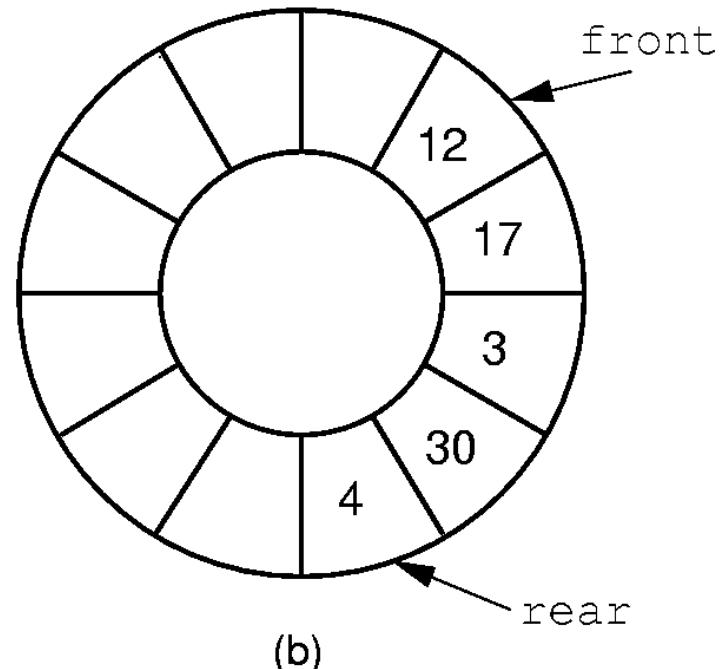
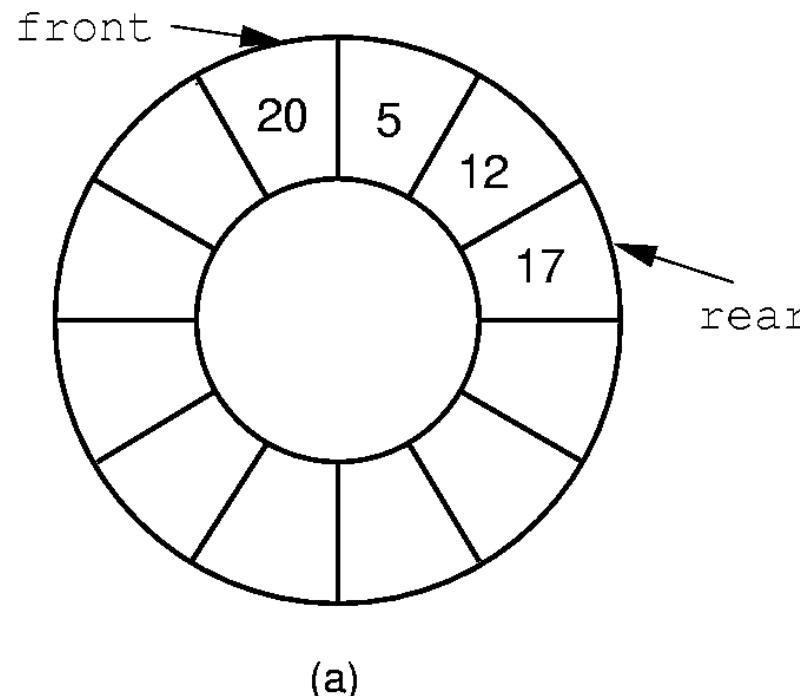
(a)



(b)

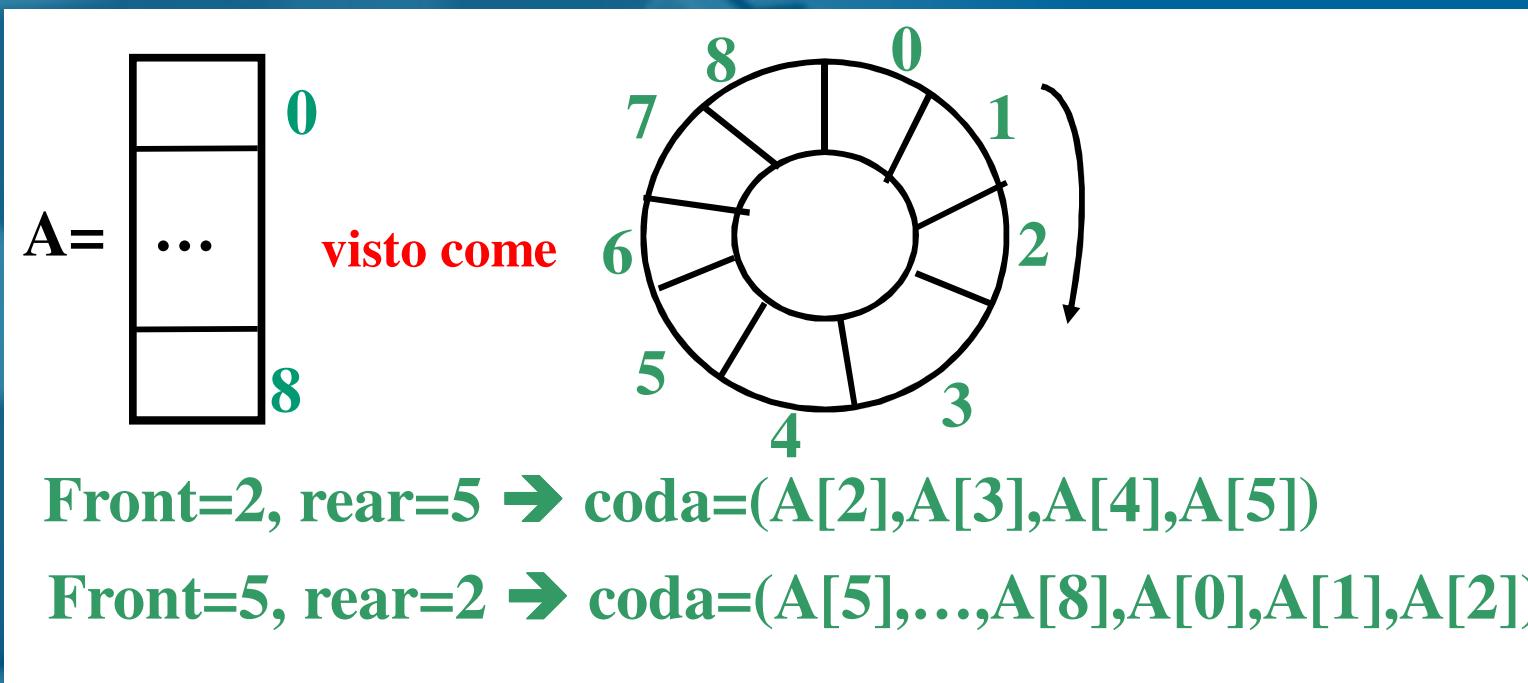
# Implementazione di una coda mediante array circolare (2)

- Dove puntano front e rear?
- Come distinguiamo tra una **coda piena** ed una **coda vuota**?



# Implementazione di una coda mediante array circolare (3)

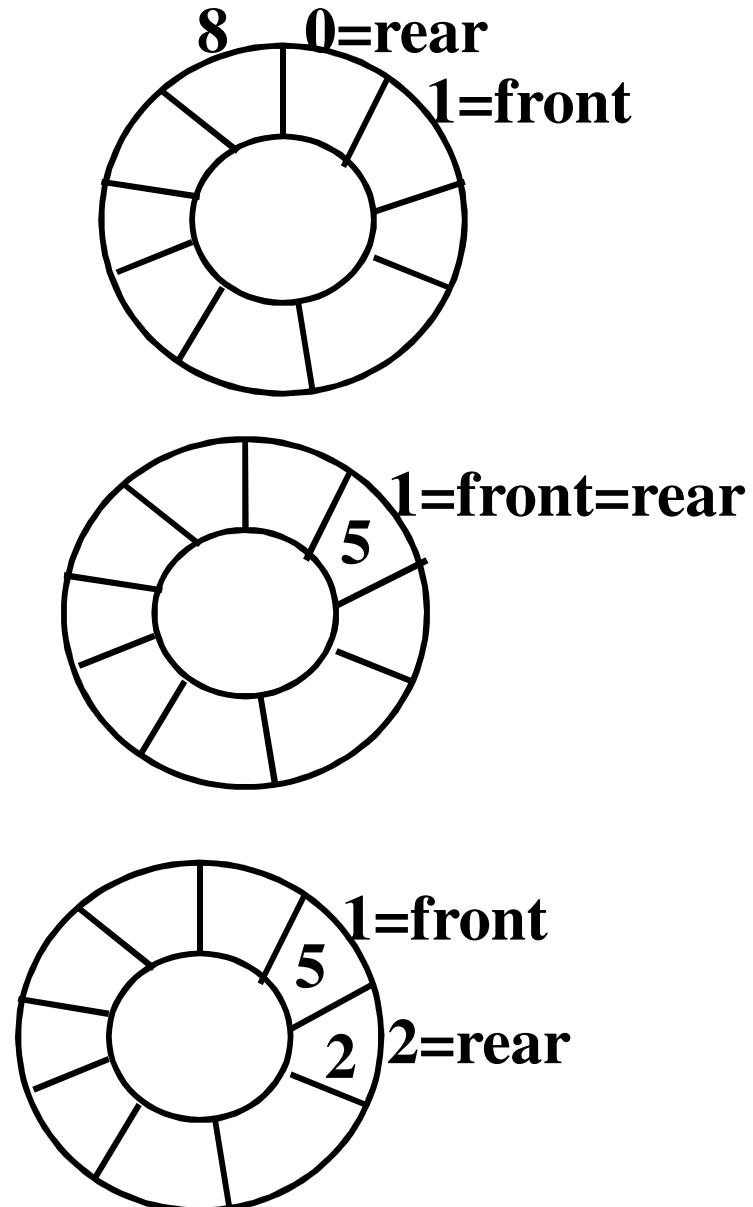
**Front, rear:** le posizioni occupate dalla coda vanno da front a rear, front precede rear in senso orario



**Coda circolare  
vuota di  
dimensione n=9:**

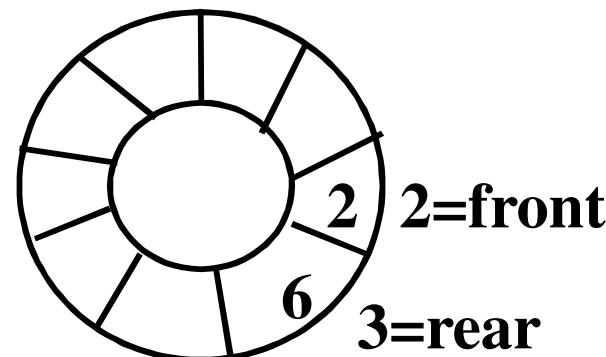
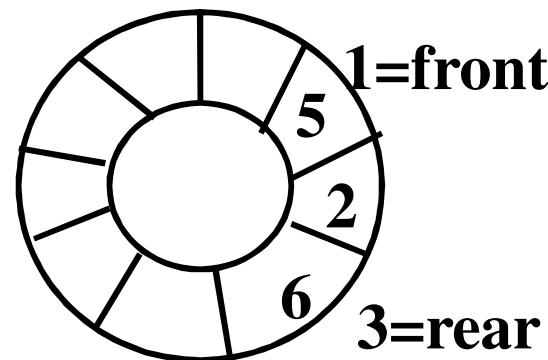
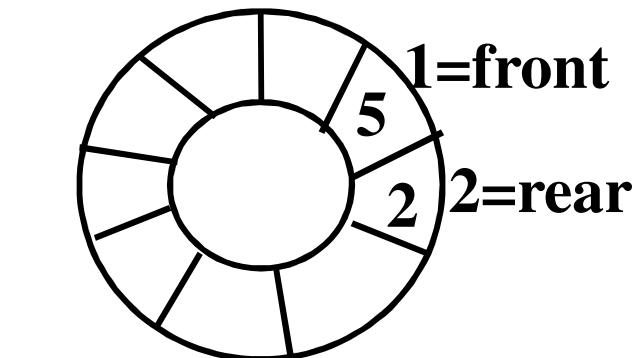
Inserire 5:

Inserire 2:

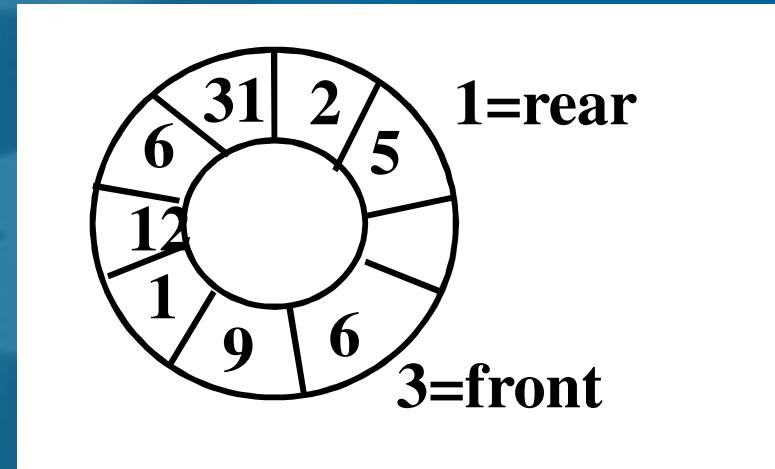


Inserire 6:

Cancellare:



**Coda piena:**





# Implementazione di una coda mediante array circolare (4)

**Coda vuota:** front in posizione immediatamente successiva al rear

es. front=3, rear=2.

**Coda piena:** contiene n-1 elementi (e non n)

front a 2 posizioni successive al rear

es. front=3, rear=1  $\rightarrow$  coda=(A[3],...,A[n-1],A[0],A[1])

**Nota:** se coda contenesse n elementi allora front in posizione immediatamente successiva al rear,

es. front=3, rear=2  $\rightarrow$

coda=(A[3],...,A[n-1], A[0], A[1], A[2]), si confonderebbe con coda vuota

# Implementazione di una coda mediante array circolare (5)

**Clear:** pon  $\text{front}=1$ ,  $\text{rear}=0$  (quindi  $\text{front}=\text{rear}+1 \% n$ )

**isEmpty:** controlla se  $\text{front}==(\text{rear}+1) \% n$

**isFull:** risultato del confronto ( $\text{front}==(\text{rear}+2) \% n$ )

**Enqueue(Q,x):** se coda è piena restituisce FALSE,  
altrimenti  $\text{rear}=(\text{rear}+1)\%n$ ;  $Q[\text{rear}]=x$ ; return TRUE

**Dequeue(Q,x):** se coda vuota restituisce null, altrimenti  
 $x=Q[\text{front}]$ ;  $\text{front}=(\text{front}+1)\%n$ ; return x



# Implementazione di una Coda mediante Linked list

## Esempio Coda

Vedi listati .java:

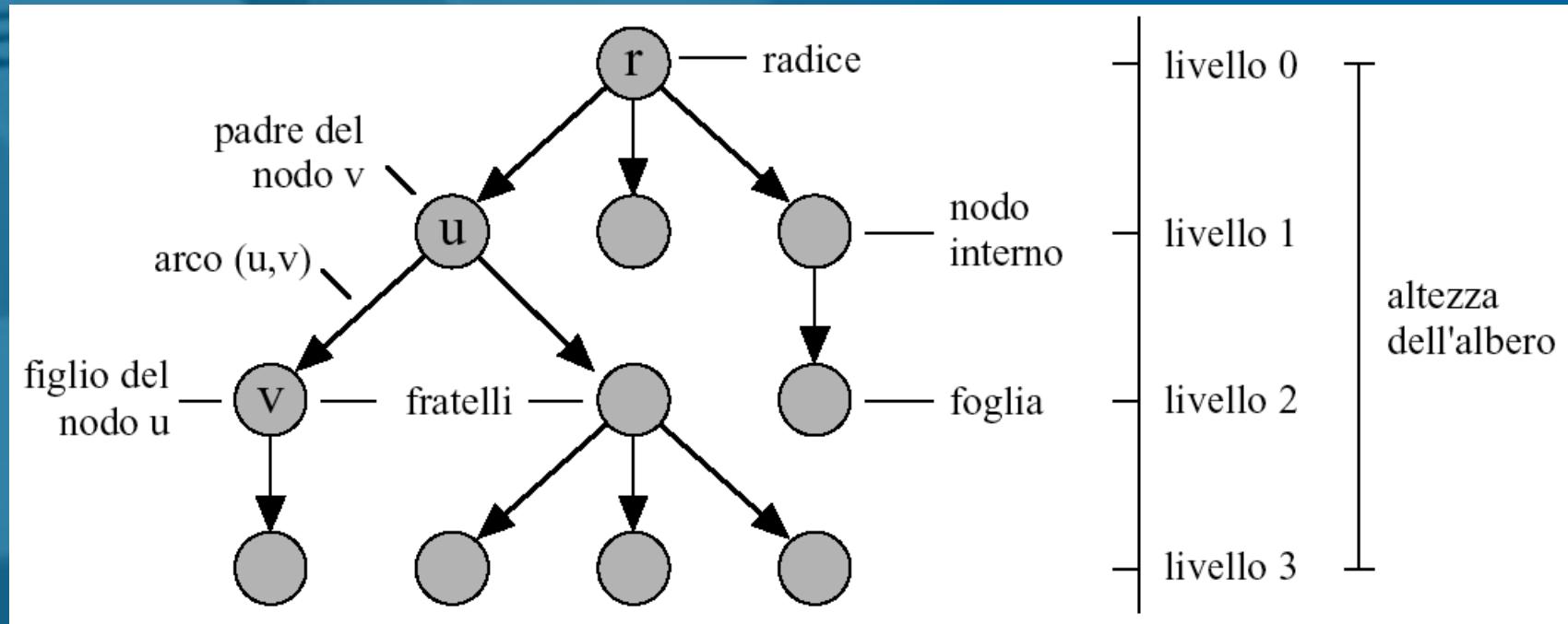
- Coda
- CodaCollegata
- EccezioneStrutturaVuota
- ProvaCoda (driver test)

Costi operazioni:  $O(1)$

Spazio occupato: numero effettivo di elementi

# Alberi

Un albero (radicato) è una **coppia**  $T = (N, A)$  costituita da un insieme  $N$  di **nodi** e da un insieme  $A$  di coppie di nodi, dette **archi**.



## Organizzazione gerarchica dei dati

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

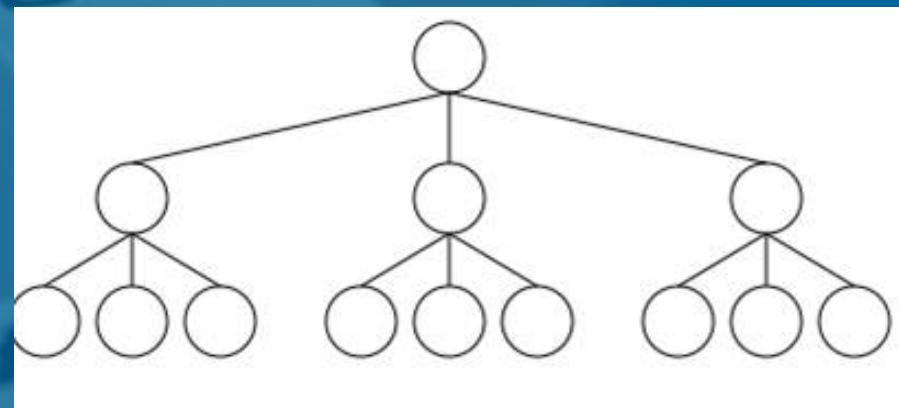
# Alberi: alcune definizioni

- **Grado di un nodo:** numero di figli del nodo
- **Cammino:** sequenza di nodi  $\langle n_0, n_1, \dots, n_k \rangle$  dove il nodo  $n_i$  è padre del nodo  $n_{i+1}$ , per  $0 \leq i < k$ 
  - La lunghezza del cammino è  $k$  (**numero di archi**)
  - Dato un nodo, esiste **un unico cammino** dalla radice dell’albero al nodo
- **Livello o profondità di un nodo:** lunghezza del cammino dalla radice al nodo
  - **Definizione ricorsiva:** il livello della radice è 0, il livello di un nodo non radice è  $1 +$  il livello del padre
- **Altezza dell’albero:** la lunghezza del più lungo cammino nell’albero (la max profondità a cui si trova una foglia)
  - Parte dalla radice e termina in una foglia

# Alberi k-ari completi

- Un **albero k-ario completo** è un albero k-ario in cui tutte le foglie hanno la stessa profondità e tutti i nodi interni hanno grado k

albero 3-ario completo





# Alberi k-ari completi

- Il numero di foglie di un albero k-ario completo è:
  - la radice ha k figli a profondità 1
  - ognuno dei figli ha k figli a profondità 2 per un totale di  $k^2$  foglie
  - a profondità h (altezza dell'albero) si hanno  $k^h$  foglie
- Il numero di nodi interni di un albero k-ario completo di altezza h è:  $1+k+k^2+\dots+k^{h-1} = \sum_{i=0..h-1} k^i = (k^h - 1)/(k-1)$
- Quindi un albero binario completo ha  $2^h - 1$  nodi interni e  $2^h$  foglie

**tipo Albero:**

**dati:**

un insieme di nodi (di tipo *nodo*) e un insieme di archi.

**operazioni:**

*numNodi() → intero*

restituisce il numero di nodi presenti nell'albero.

*grado(nodo v) → intero*

restituisce il numero di figli del nodo *v*.

*padre(nodo v) → nodo*

restituisce il padre del nodo *v* nell'albero, o null se *v* è la radice.

*figli(nodo v) → {nodo, nodo, ..., nodo}*

restituisce, uno dopo l'altro, i figli del nodo *v*.

*aggiungiNodo(nodo u) → nodo*

inserisce un nuovo nodo *v* come figlio di *u* nell'albero e lo restituisce.

Se *v* è il primo nodo ad essere inserito nell'albero, esso diventa la radice (e *u* viene ignorato).

*aggiungiSottoalbero(Albero a, nodo u)*

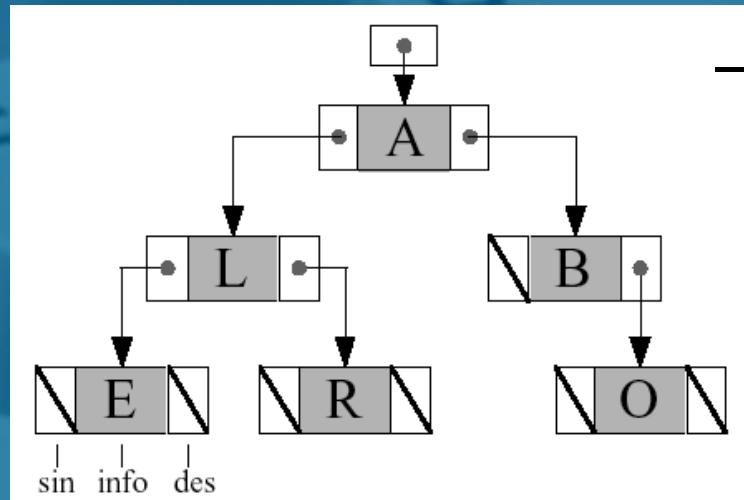
inserisce nell'albero il sottoalbero *a* in modo che la radice di *a* diventi figlia di *u*.

*rimuoviSottoalbero(nodo v) → Albero*

stacca e restituisce l'intero sottoalbero radicato in *v*. L'operazione cancella dall'albero il nodo *v* e tutti i suoi discendenti.

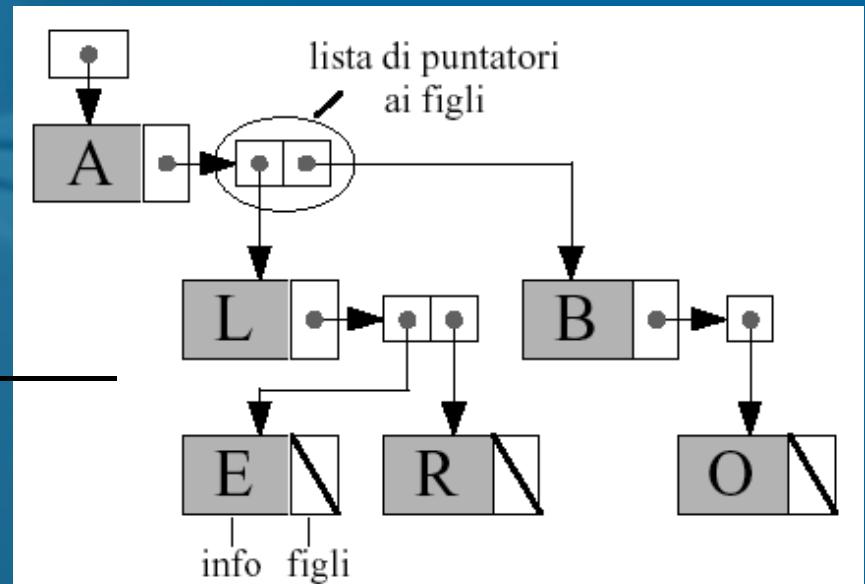
# Rappresentazioni collegate di alberi

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

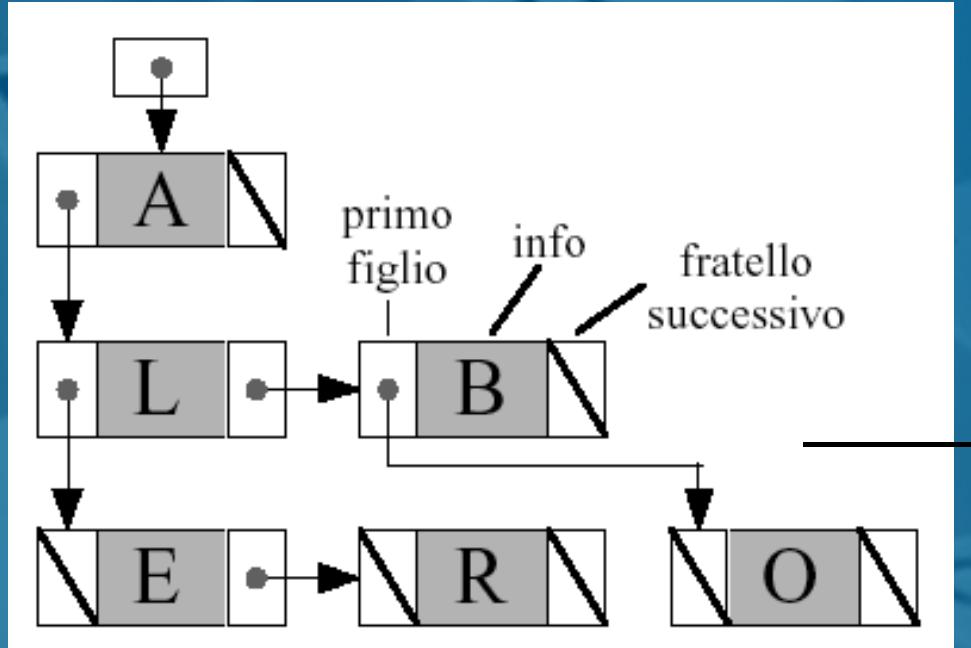


$S(n)=O(n)$   
Rappresentazione  
con liste di puntatori ai  
figli (nodi con numero  
arbitrario di figli)

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



# Rappresentazioni collegate di alberi



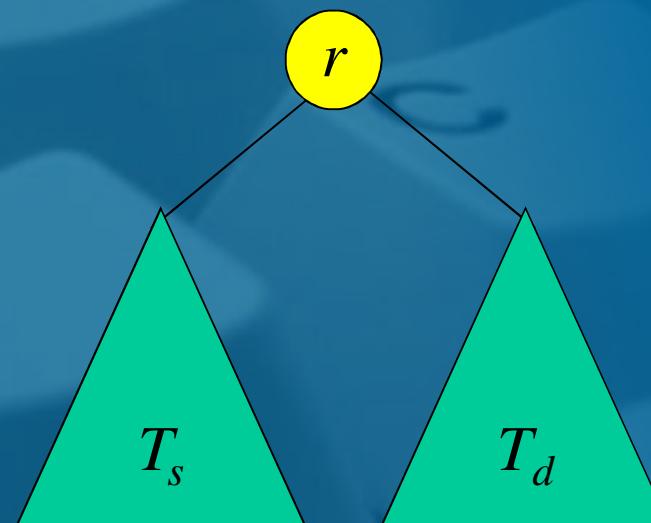
Rappresentazione  
di tipo **primo figlio-**  
**fratello successivo**  
(nodi con numero  
arbitrario di figli)

Utile per la rappresentazione binaria di alberi n-ari

# Alberi binari

Definizione ricorsiva di albero binario:

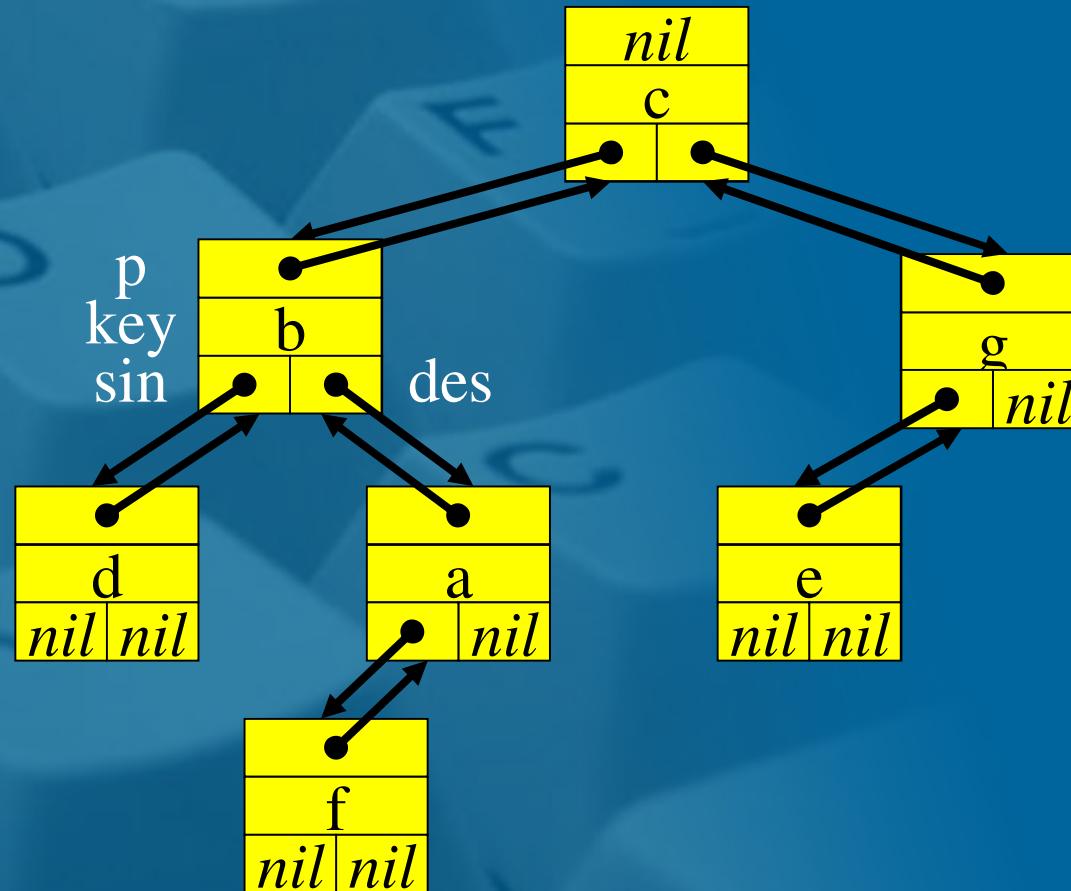
- a) l'insieme vuoto  $\emptyset$  è un albero binario ■
- b) se  $T_s$  e  $T_d$  sono alberi binari ed  $r$  è un nodo allora la terna ordinata  $(r, T_s, T_d)$  è un albero binario



In memoria l'albero binario:

$$T = (c, (b, (d, \phi, \phi), (a, (f, \phi, \phi), \phi)), (g, (e, \phi, \phi), \phi))$$

si rappresenta con due puntatori ai sotto-alberi, ed uno al padre:

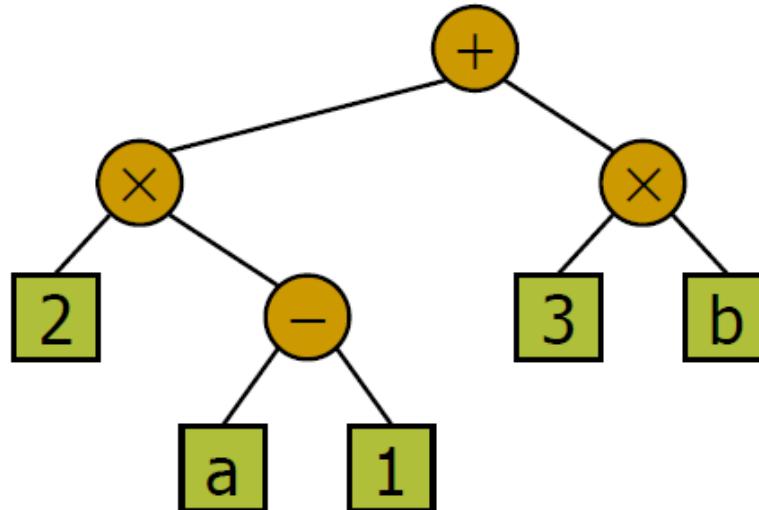


# Applicazioni di alberi binari: Espressioni aritmetiche, processi di decisione, ricerca, ecc..

Albero binario associato ad una espressione:

- Nodi interni: operatori
- Nodi esterni: operandi

Esempio:  $(2 \times (a - 1) + (3 \times b))$





# 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*)

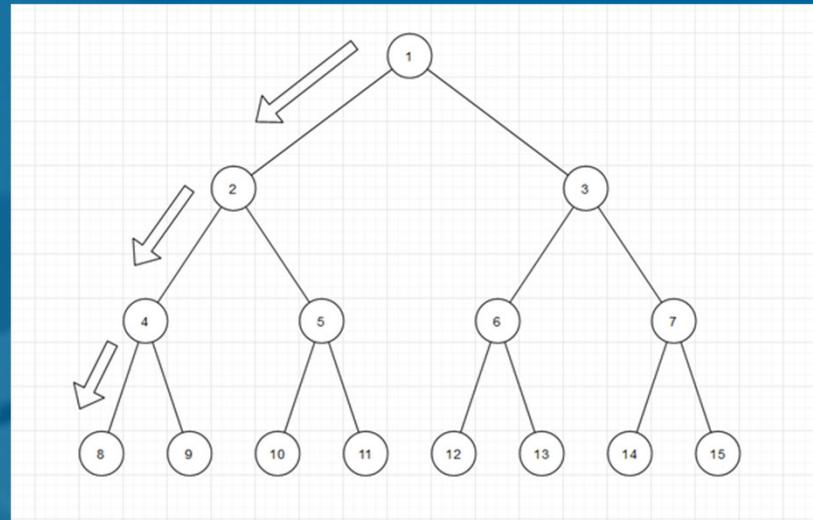
1.       $S \leftarrow \{r\}$
2.      **while** ( $S \neq \emptyset$ ) **do**
3.          estrai un nodo *u* da *S*
4.          *visita il nodo u*
5.           $S \leftarrow S \cup \{ \text{figli di } u \}$

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.

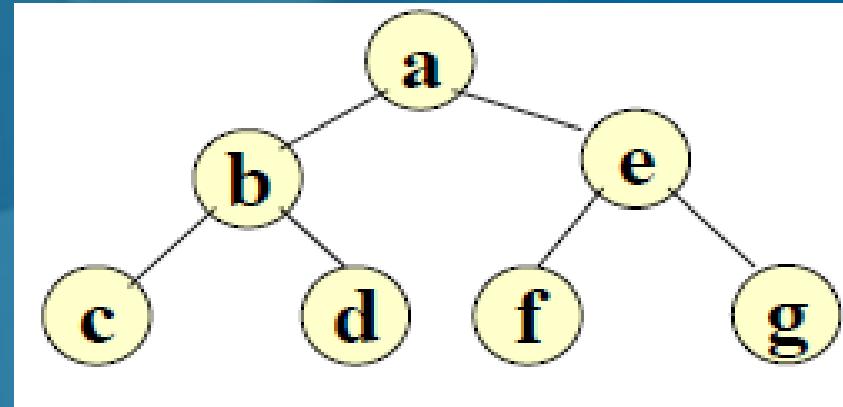


# Tre varianti dell'algoritmo di visita in profondità

- Con un albero binario sono possibili 3 strategie:
  - *preordine* o ordine anticipato: si visita prima il nodo e poi i sottoalberi sinistro e destro
  - *inordine* o ordine simmetrico: si visita prima il sottoalbero sinistro e poi il nodo e poi il sottoalbero destro
  - *postordine* o ordine posticipato: si visita prima il sottoalbero sinistro, poi quello destro e poi il nodo

# Tre varianti dell'algoritmo di visita in profondità

Esempio:



- Preordine:
- Inordine:
- Postordine:

a    b    c    d    e    f    g

c    b    d    a    f    e    g

c    d    b    f    g    e    a



# Algoritmo di visita in profondità

Versione iterativa (per alberi binari):  
visita in preordine

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



# Algoritmo di visita in profondità

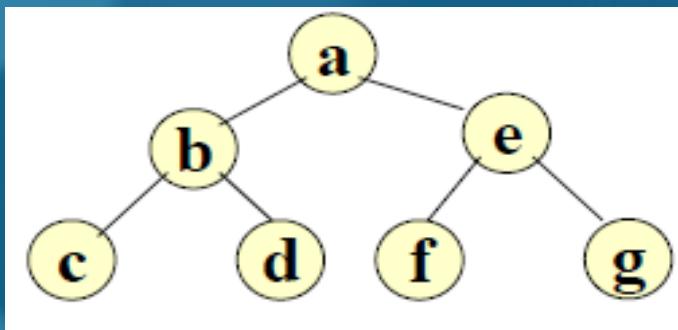
Versione ricorsiva (per alberi binari):  
visita in preordine

```
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.



a b e c d f g



# 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()
 if (u \neq null) then
 visita il nodo u
 C.enqueue(figlio sinistro di u)
 C.enqueue(figlio destro di u)
```



# Implementazione del tipo Albero Binario

Vedi codice sorgente:

- AlberoBinario.java (interface)
- AlberoBinarioImpl.java (classe albero binario)
- NodoBinario.java (classe nodo)
- AlberoBinarioDemo (classe driver test)



# 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

Capitolo 2  
Modelli di calcolo e  
metodologie di analisi

Camil Demetrescu, Irene Finocchi,  
Giuseppe F. Italiano



# Analisi di algoritmi ricorsivi

- La quantità di risorsa usata da algoritmi ricorsivi può essere espressa tramite **relazioni di ricorrenza**
- Risolvibili tramite vari metodi generali:  
iterazione, sostituzione, alberi di ricorsione,  
teorema Master...

# Esempio di algoritmo ricorsivo

L’algoritmo di ricerca binaria può essere riscritto in forma ricorsiva come

```
algoritmo ricercaBinariaRic(array L, elemento x) → booleano
1. n ← lunghezza di L
2. if (n = 0) then return non trovato
3. i ← $\lceil n/2 \rceil$
4. else if (L[i] = x) then 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])
```

(Si assume *L* ordinato e indicizzato da 1 a *n*)



# Esempio di algoritmo ricorsivo

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

1.  $n \leftarrow$  lunghezza di *L*
2. **if** ( $n = 0$ ) **then return** non trovato
3.  $i \leftarrow \lceil n/2 \rceil$
4. **else if** ( $L[i] = x$ ) **then 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?

# Equazioni di ricorrenza

Il tempo di esecuzione della ricerca binaria (versione ricorsiva) può essere descritto tramite l'equazione di ricorrenza:

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

O semplicemente (assumendo  $n$  potenza del 2):

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

# Metodo dell'iterazione

- Iterare la ricorsione, ottenendo una sommatoria dipendente solo dalla dimensione  $n$  e dalle condizioni iniziali.
- Le tecniche per limitare le sommatorie possono essere usate poi per fornire limiti alla soluzione.

**Esempio (Ricerca binaria) :**  $T(n) = c + T(n/2)$

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

→  $T(n) = c + T(n/2) = 2c + T(n/4) = 3c + T(n/8) = \dots$   
 $= (\sum_{j=1}^i c) + T(n/2^i) = i c + T(n/2^i)$

Raggiungiamo il passo base per  $n/2^i = 1$  e cioè per  $i = \lg n$ ,  
da cui:  $T(n) = c \lg n + T(1) = O(\lg n)$

# Metodo dell'iterazione

- Può richiedere l'uso di molta algebra e calcoli complessi
- Bisogna concentrarsi su due parametri:
  - il numero di volte che la ricorrenza deve essere iterata per raggiungere le condizioni iniziali (a contorno)
  - la somma dei termini originati da ogni livello del processo di iterazione
- Talvolta nel processo di iterazione si può indovinare la soluzione senza fare tutti i calcoli matematici. In tal caso è possibile passare al **metodo di sostituzione**.

# Metodo della sostituzione

Idea: “indovinare” una soluzione, ed usare induzione matematica per provare che la soluzione dell’equazione di ricorrenza è effettivamente quella intuita

Esempio:  $T(n) = n + T(n/2)$ ,  $T(1)=1$

Assumiamo che la soluzione sia  $T(n) \leq c n$  per una costante  $c$  opportuna

Passo base:  $T(1)=1 \leq c 1$  per ogni  $c \geq 1$

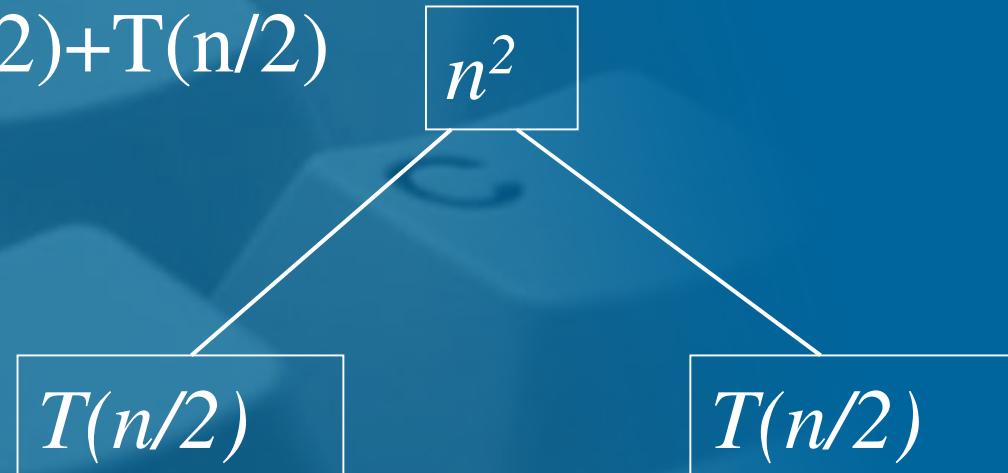
Passo induttivo:  $T(n) = n + T(n/2) \leq n + c(n/2) = (1+c/2)n$   
da cui  $T(n) \leq c n$  per  $1+c/2 \leq c$  ovvero per  $c \geq 2$

# Alberi di ricorsione

Un albero di ricorsione è un metodo molto comodo per visualizzare tutto ciò che accade quando si itera una ricorrenza.

Consideriamo la ricorrenza:

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



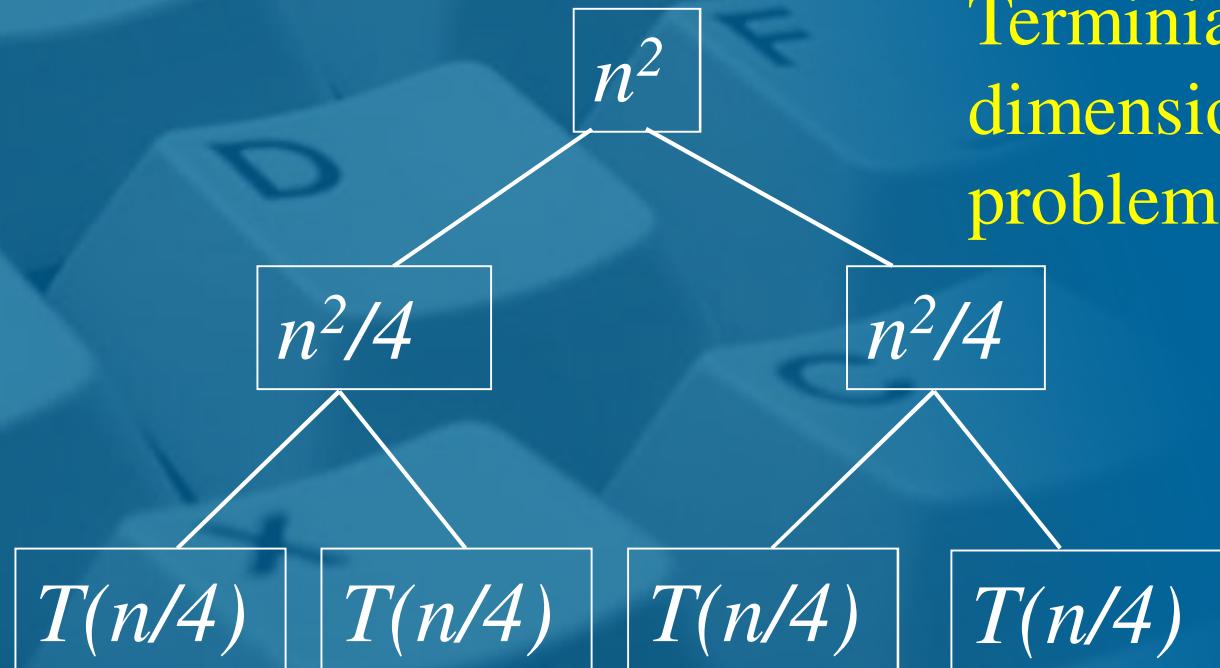
# Alberi di ricorsione

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

$$T(n/2) = n^2/4 + T(n/4) + T(n/4)$$

I nodi riportano le dimensioni dei sottoproblemi

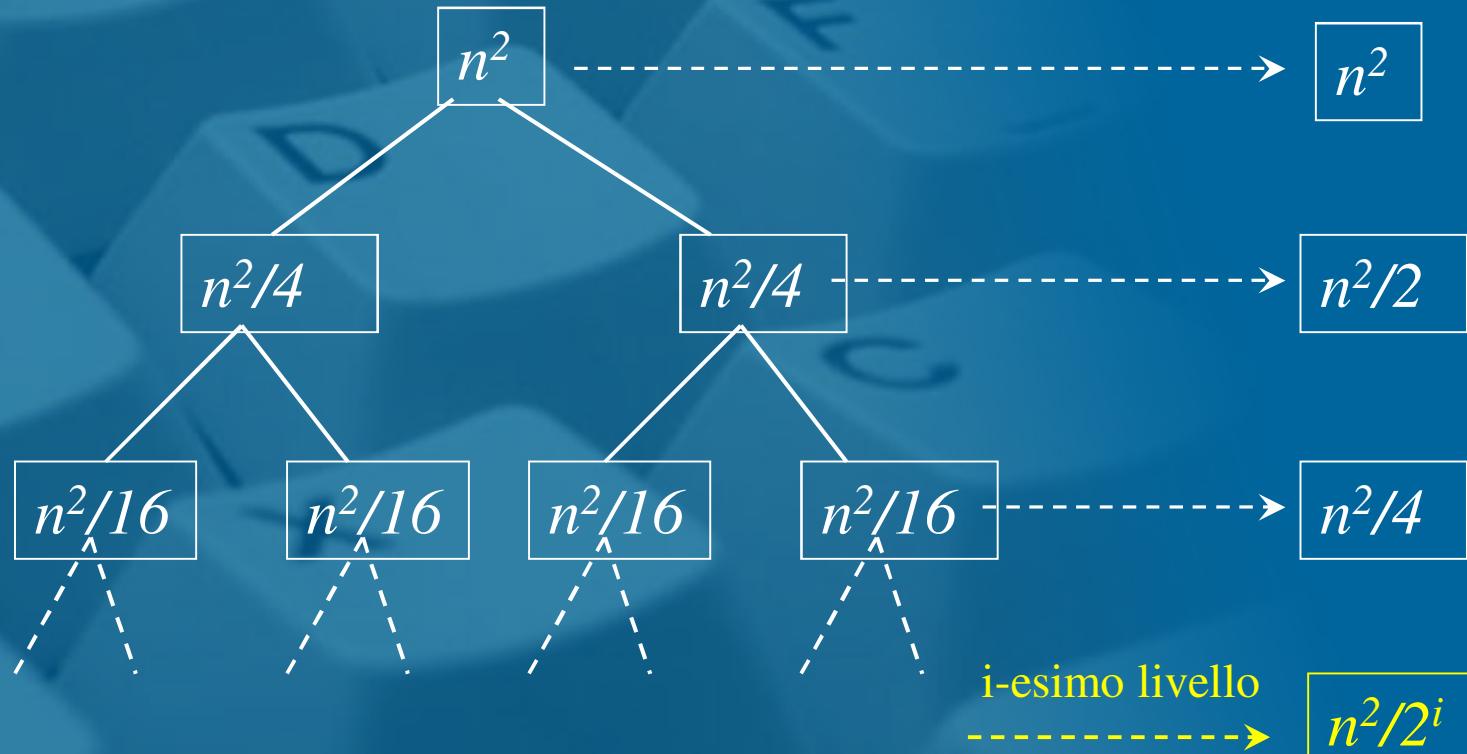
Terminiamo quando la dimensione del sottoproblema è 1:  $n/2^i = 1$



# Alberi di ricorsione

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

Si analizza il totale delle dimensioni per livelli



# Alberi di ricorsione

Poiché il valore decresce geometricamente, il valore totale differisce, al più di un fattore costante dal termine più grande, quindi la soluzione è  $\Theta(n^2)$

$$\frac{n}{2^i} = 1 \rightarrow i = \lg n$$

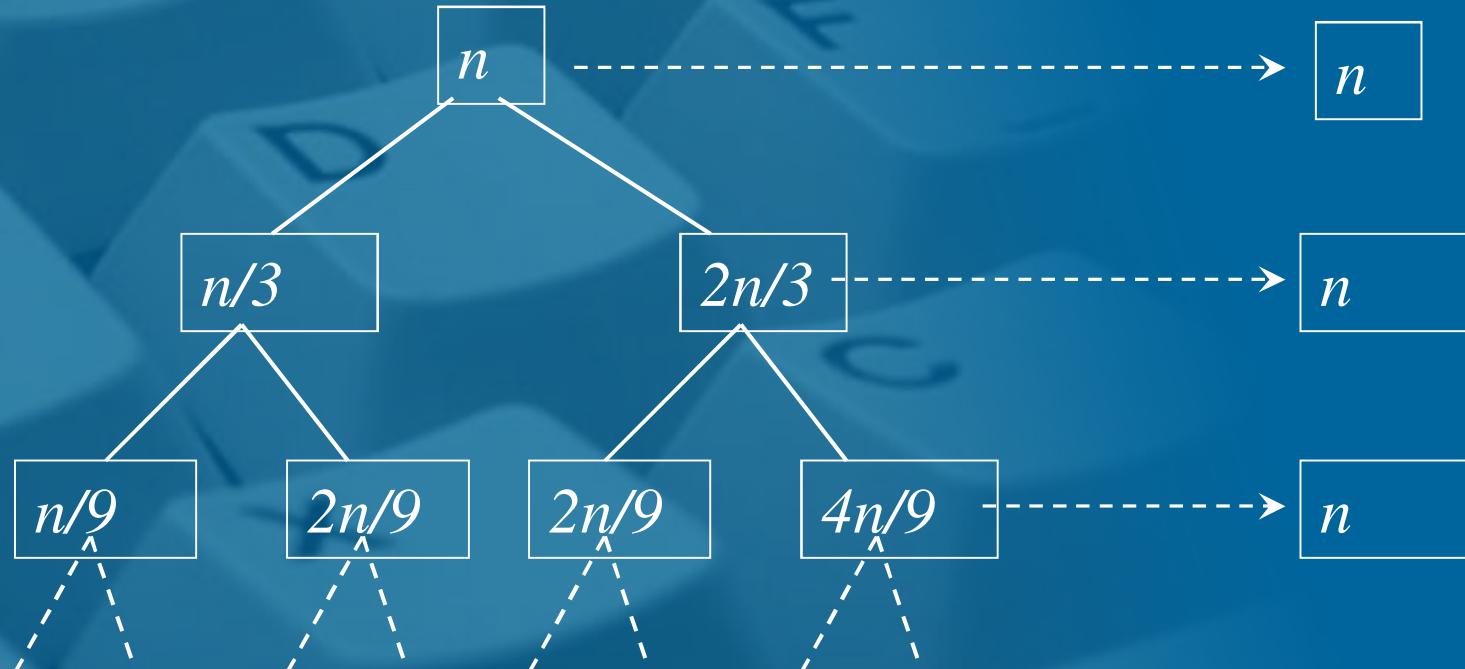
dimensione  
del sotto-problema  
pari alla dimensione minima 1

$$\begin{aligned} T(n) &= \sum_{i=0}^{\lg n} \frac{n^2}{2^i} \\ &= n^2 \sum_{i=0}^{\lg n} \left( \frac{1}{2^i} \right) = n^2 \frac{1 - (1/2)^{\lg n + 1}}{1/2} \\ &= n^2 \left( 2 - (1/2)^{\lg n} \right) = n^2 \left( 2 - n^{\lg(1/2)} \right) \\ &= 2n^2 - n \\ &= \Theta(n^2) \end{aligned}$$

# Alberi di ricorsione

Un altro esempio

$$T(n) = T(n/3) + T(2n/3) + n$$



# Alberi di ricorsione

Il ramo destro dell'albero è quello più lungo, quindi:

$$\left(\frac{2}{3}\right)^i n = 1 \quad \rightarrow \quad i = \log_{3/2} n$$

$$\begin{aligned} T(n) &\leq \sum_{i=0}^{\log_{3/2} n} n \\ &= n \log_{3/2} n \\ &= O(n \lg n) \end{aligned}$$



# Teorema Master

Permette di analizzare algoritmi basati sulla tecnica del *divide et impera*:

- dividi il problema (di dimensione  $n$ ) in  $a$  sottoproblemi di dimensione  $n/b$  con  $a \geq 1$  e  $b > 1$
- risolvi i sottoproblemi ricorsivamente
- ricombina le soluzioni

Sia  $f(n)$  il tempo per dividere e ricombinare istanze di dimensione  $n$ . La relazione di ricorrenza è data da:

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

# Teorema Master

La relazione di ricorrenza:

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

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



# Esempi

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

$a=2, b=2, f(n)=n=\Theta(n \log_2 2)$   $\Rightarrow T(n)=\Theta(n \log n)$   
(caso 2 del teorema master)

$$2) T(n) = c + 3T(n/9)$$

$a=3, b=9, f(n)=c=O(n \log_9 3 - \varepsilon)$   $\Rightarrow T(n)=\Theta(\sqrt{n})$   
(caso 1 del teorema master) per  $\varepsilon = 0,5$

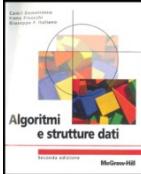
$$3) T(n) = n + 3T(n/9)$$

$a=3, b=9, f(n)=n=\Omega(n \log_9 3 + \varepsilon)$   
 $3(n/9) \leq c n$  per  $c=1/3$  e  $\varepsilon = 0,5$   $\Rightarrow T(n)=\Theta(n)$   
(caso 3 del teorema master)

# Esercizio (appello 31 Ottobre 2021)

- Calcolare la complessità tempo della seguente funzione ricorsiva, assumendo che l'algoritmo MergeSort ha complessità  $\Theta(n \log n)$

```
foo(n,A){ //n intero positivo e A array di dimensione n
 if (n < 10) return 1;
 else if (n < 1234){
 tmp = n;
 for (i = 1; i <= n; i = i*2)
 for (j = 1; j <= n; j = j+1)
 tmp = tmp + i * j;
 } return tmp;
 else { tmp = n; MergeSort(A,0,n-1); }
 return tmp + foo(n/4) + foo(n/4) + foo(n/4);
}
```



# 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



UNIVERSITÀ  
DEGLI STUDI  
DI BERGAMO

Dipartimento  
di Ingegneria Gestionale,  
dell'Informazione e della Produzione



# Pile e code nel JCF, alberi binari (esercitazione)

PROGETTAZIONE, ALGORITMI E  
COMPUTABILITÀ  
(38090-MOD1)

Corso di laurea  
Magistrale in  
Ingegneria  
Informatica

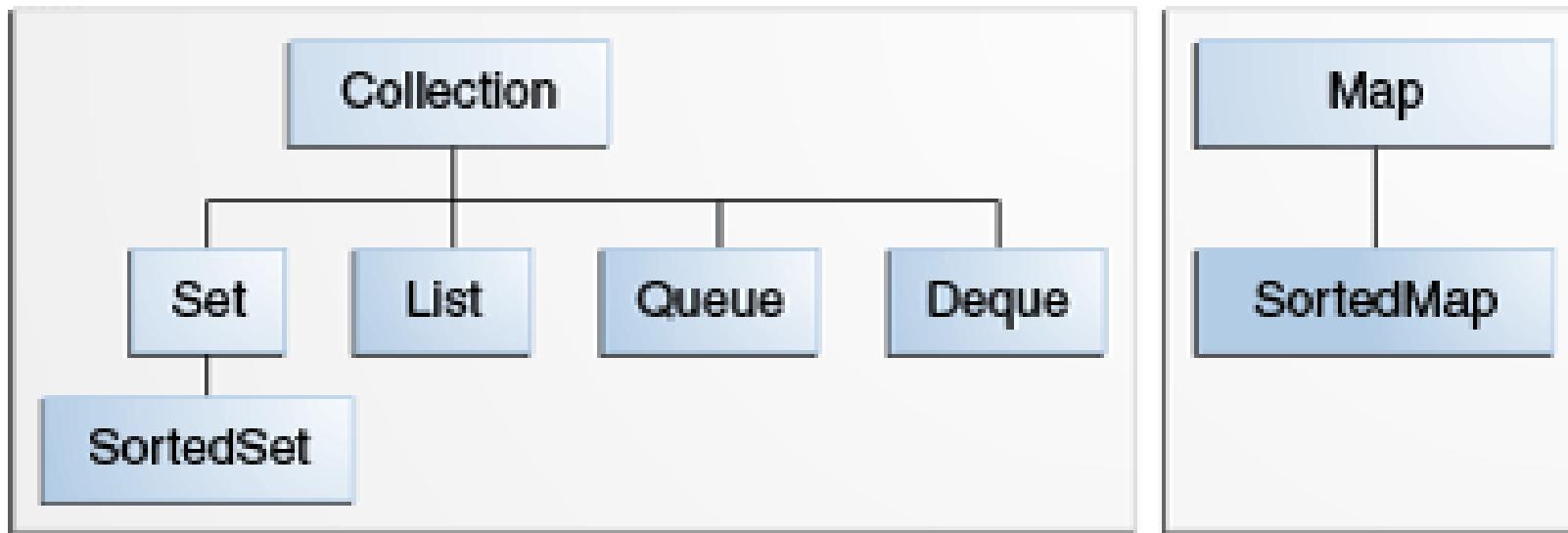
RELATORE  
Prof.ssa Patrizia  
Scandurra

SEDE  
DIGIP

# Il Java Collection Framework (JCF)

- **Interfacce e classi del package `java.util`**
- Forniscono:
  - **ADT e strutture dati** per manipolare **collezioni di oggetti**
  - **algoritmi di base** (ad es. come ordinamento e ricerca)
- <http://docs.oracle.com/javase/tutorial/collections/index.html>

# JCF – le interfacce

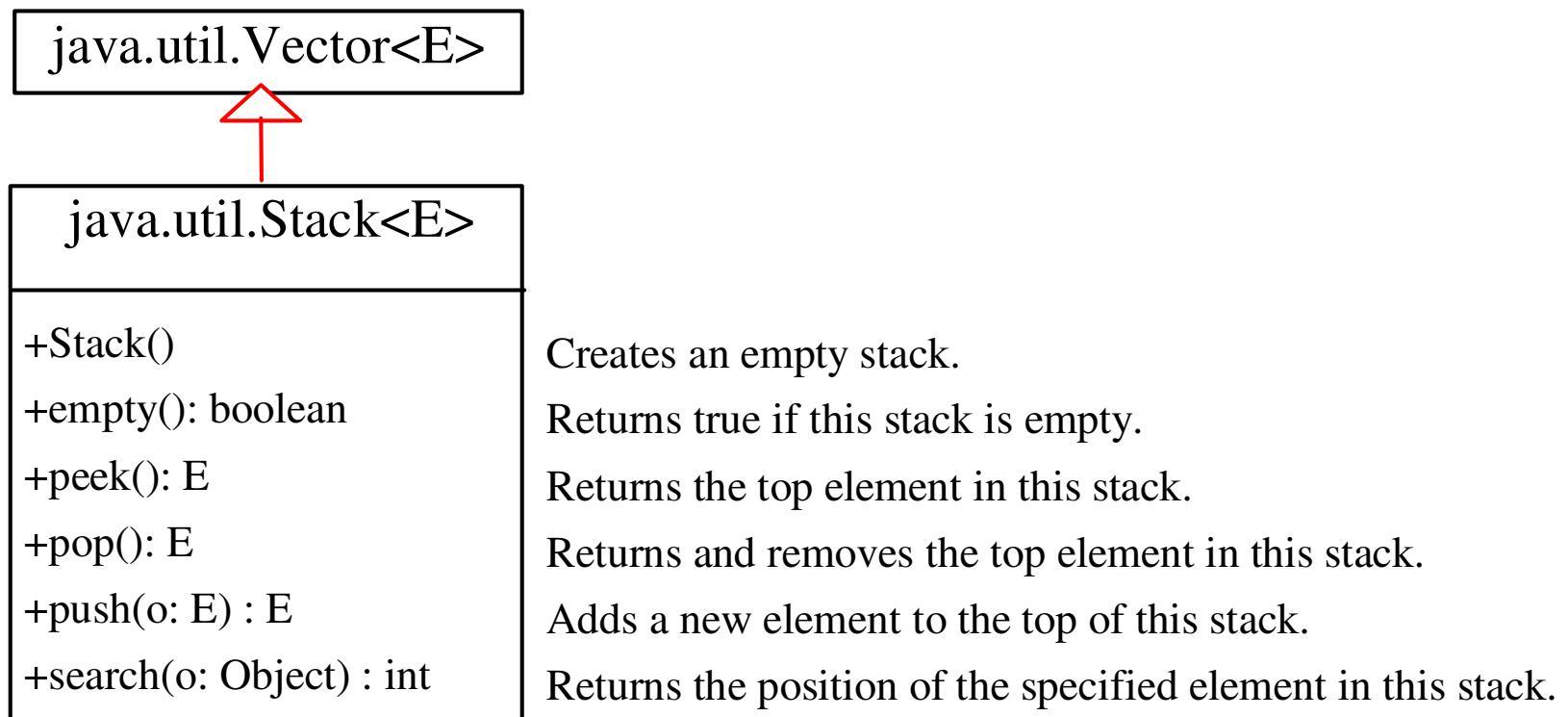


- Formano una gerarchia
- Tutte le interfacce sono generiche. Ad esempio la definizione dell'interfaccia Collection:  
`public interface Collection<E>...`

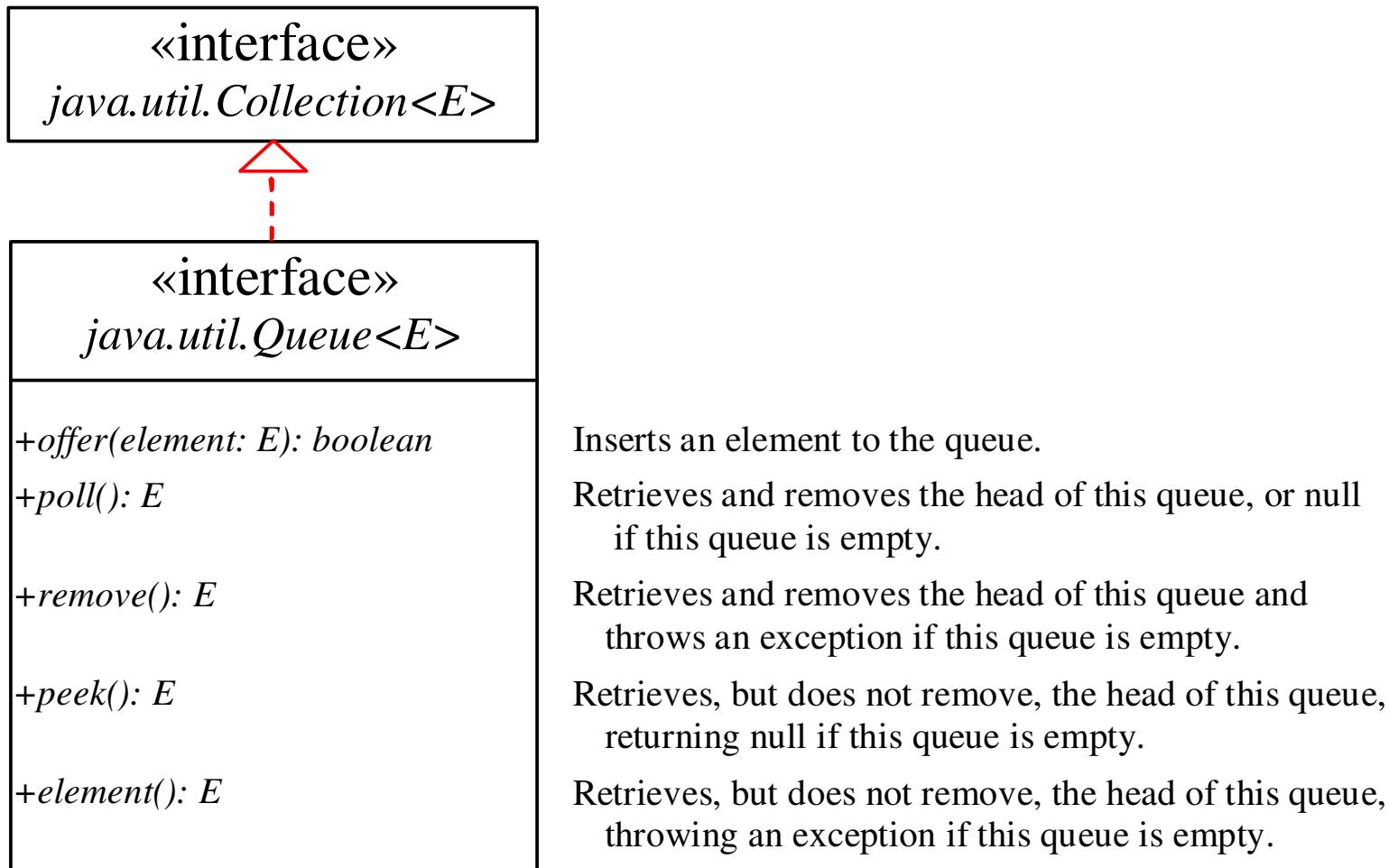
# ADT Pila e Coda in JCF

- La classe predefinita **Stack**:
  - public class Stack<E> extends Vector<E>
- Interfaccia **Queue**:
  - classi JCF che implementano Queue: AbstractQueue, ArrayBlockingQueue, ArrayDeque, ConcurrentLinkedQueue, DelayQueue, LinkedBlockingDeque, LinkedBlockingQueue, **LinkedList**, PriorityBlockingQueue, PriorityQueue, SynchronousQueue
- Interfaccia **Deque**: **usabile sia come pila che come coda**
  - public interface Deque<E> extends Queue<E>
  - classi JCF che implementano Deque: ArrayDeque, LinkedBlockingDeque, **LinkedList**

# La classe Stack



# Interfaccia Queue



# Interfaccia Dequeue

- <https://docs.oracle.com/javase/8/docs/api/java/util/Deque.html>

|                | <b>First Element (Head)</b> |                      | <b>Last Element (Tail)</b> |                      |
|----------------|-----------------------------|----------------------|----------------------------|----------------------|
|                | <i>Throws exception</i>     | <i>Special value</i> | <i>Throws exception</i>    | <i>Special value</i> |
| <b>Insert</b>  | addFirst(e)                 | offerFirst(e)        | addLast(e)                 | offerLast(e)         |
| <b>Remove</b>  | removeFirst()               | pollFirst()          | removeLast()               | pollLast()           |
| <b>Examine</b> | getFirst()                  | peekFirst()          | getLast()                  | peekLast()           |

# ADT PILA: Stack vs Dequeue

## Stack Method

push(e)

pop()

peek()

## Equivalent Deque Method

addFirst(e)

removeFirst()

peekFirst()

# ADT Coda: Queue vs Dequeue

## Queue Method

add(e)

offer(e)

remove()

poll()

element()

peek()

## Equivalent Deque Method

addLast(e)

offerLast(e)

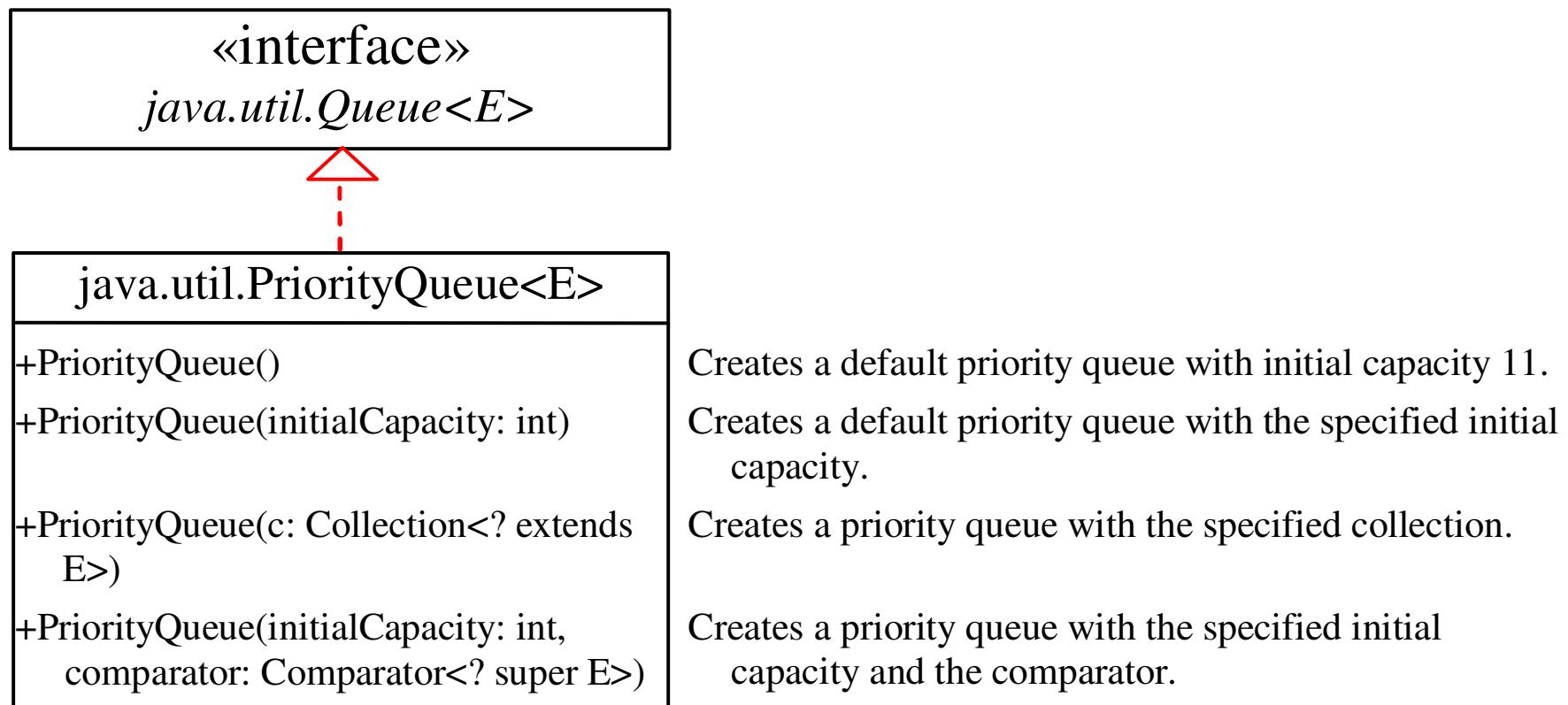
removeFirst()

pollFirst()

getFirst()

peekFirst()

# La classe PriorityQueue



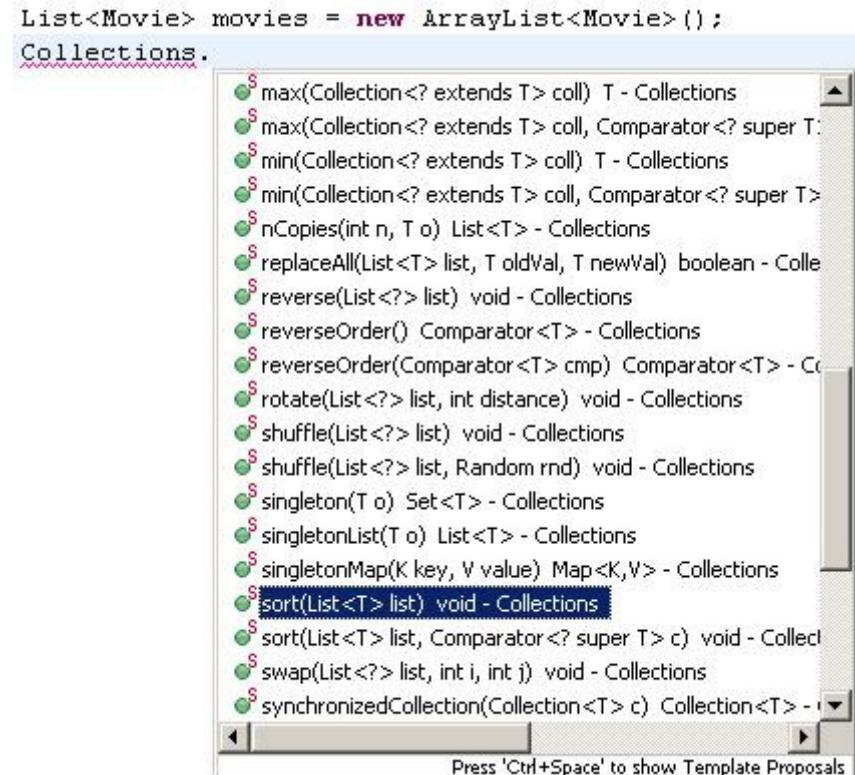
- L'elemento **a priorità max** viene **rimosso per primo**
- La **priorità** è stabilita esternamente mediante **un oggetto Comparator**

# JCF - Algoritmi

## java.util.Collections

### Algoritmi per diversi tipi di collezioni

- Sorting (e.g. sort)
- Shuffling (e.g. shuffle)
- Routine Data Manipulation  
(e.g. reverse, addAll)
- Searching (e.g. binarySearch)
- Composition (e.g. frequency)
- Finding Extreme Values  
(e.g. max)



# Synchronized Wrapper

- La classe **Collections** contiene dei metodi statici per costruire un **wrapper di sincronizzazione** per una collezione del JCF e renderla *thread-safe*

```
public static Collection synchronizedCollection(Collection c);
public static Set synchronizedSet(Set s);
public static List synchronizedList(List list);
public static <K,V> Map<K,V> synchronizedMap(Map<K,V> m);
public static SortedSet synchronizedSortedSet(SortedSet s);
public static <K,V> SortedMap<K,V> synchronizedSortedMap(SortedMap<K,V> m);
```

- In alternativa, usare le classi collezioni del package `java.util.concurrent`
  - `BlockingQueue`, `BlockingDeque`, `ConcurrentMap`, ecc..

# Synchronized Wrapper

- Esempio

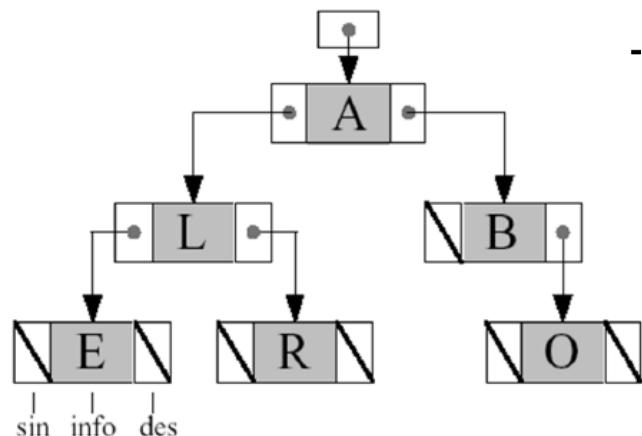
```
Collection c = Collections.synchronizedCollection(myCollection);
synchronized(c) {
 Iterator i = c.iterator(); // Must be in a synchronized block!
 while (i.hasNext())
 foo(i.next());
}
```

# Esercizio

- Invertire l'ordine degli elementi di una pila S usando una coda Q  
(Sfruttare gli ADT Stack, Queue, o Dequeue del JCF)

# Alberi binari -- *Descrizione ricorsiva*

- Molti algoritmi su alberi binari possono essere descritti in modo naturale sfruttando la definizione ricorsiva (e quindi la ***strategia divide et impera***):
  - **Caso base** - albero vuoto o formato dalla sola radice;
  - **Clausola ricorsiva** - due chiamate ricorsive, una per ogni sotto-albero
- Esempio: Vedi i metodi ***numNodi()*** e ***numNodi(NodoBinario r)*** nell'implementazione in Java del tipo albero binario
- Nota che la rappresentazione collegata a cui stiamo facendo riferimento è con puntatori diretti ai figli

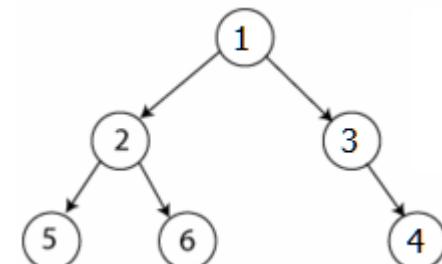


# Esercizi

Dato il codice sorgente su Moodle nella cartella *Codice sorgente/src\_alberi*, arricchire l'interfaccia **AlberoBinario** e la classe **AlberoBinarioImpl** con le seguenti operazioni usando (dove possibile) la ricorsione:

1. **public int level(NodoBinario u)**: restituisce la profondità (o livello) di un nodo.  
Ricorda che la profondità della radice è 0 e quella di un nodo diverso dalla radice è quella del padre del nodo, incrementata di 1.
2. **public int altezza()**: restituisce l'altezza dell'albero binario.
3. **public int numFoglie()**: restituisce il numero di foglie dell'albero binario.
4. **public int numNodiInterni()**: restituisce il numero di nodi *interni* dell' albero binario. Ricorda che un nodo interno è un nodo che NON è foglia.
5. **public boolean equals (Object anotherTree)**: restituisce true se l'albero (*this*) e *anotherTree* (eventualmente vuoti) sono uguali (stessa struttura e medesimi contenuti); false, altrimenti.

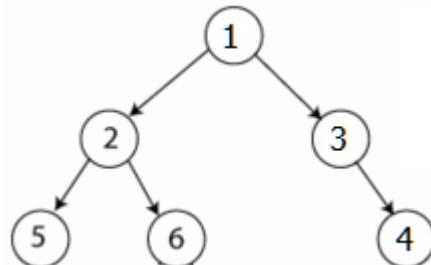
Albero di test costruito da **AlberoBinarioDemo**:



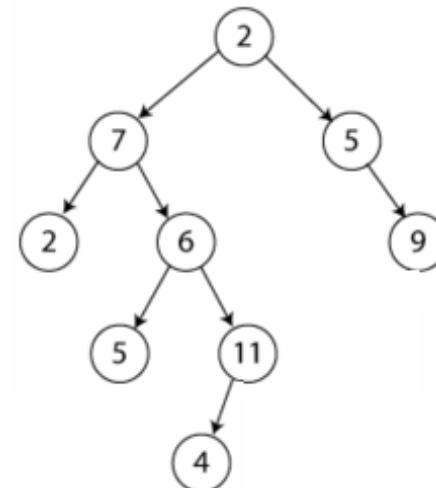
# Esercizi

6. **public void eliminaFoglieUguali()**: modifica l'albero eliminando tutte le foglie che hanno un valore uguale al valore del fratello.
7. **public boolean search (Object elem)**: stabilisce se un elemento appartiene o meno all'albero binario, attraverso una *visita esaustiva* ricorsiva. Il caso base è banale: se l'albero è vuoto, si restituisce false (zero). La visita viene interrotta non appena si trova l'elemento cercato (la prima occorrenza).
8. **public List nodiCardine ()**: Definire un algoritmo ricorsivo che restituisce i nodi cardine di T. In un albero binario T, un nodo  $u$  è un nodo cardine se e solo se  $p_u = h_u$  dove  $p_u$  è la profondità di  $u$  e  $h_u$  è l'altezza dell'albero radicato in  $u$ .

(Appello 18 Giugno 2018)



Nodi cardine: 2, 3



Nodi cardine: 5, 6

# Algoritmi e Strutture Dati

## Capitolo 6 Alberi di ricerca

Camil Demetrescu, Irene Finocchi,  
Giuseppe F. Italiano



# 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)`

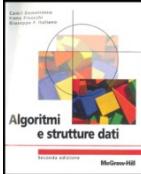
aggiunge a  $S$  una nuova coppia  $(e, k)$ .

`delete(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.



# Alberi binari di ricerca (BST = binary search tree)

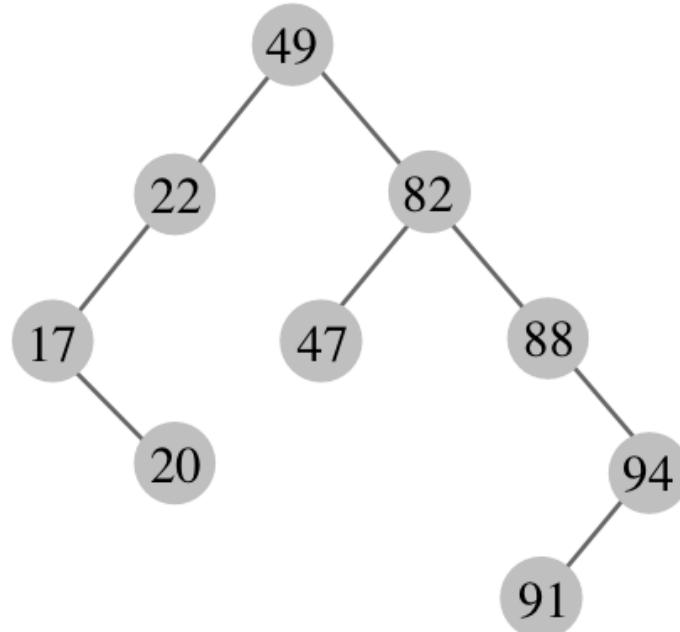
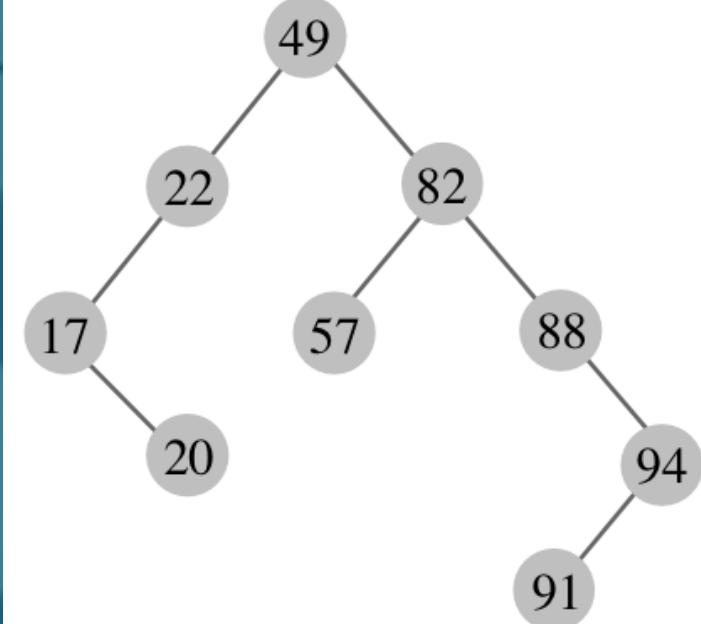


# Definizione

Albero binario che soddisfa le seguenti proprietà

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

# Esempi

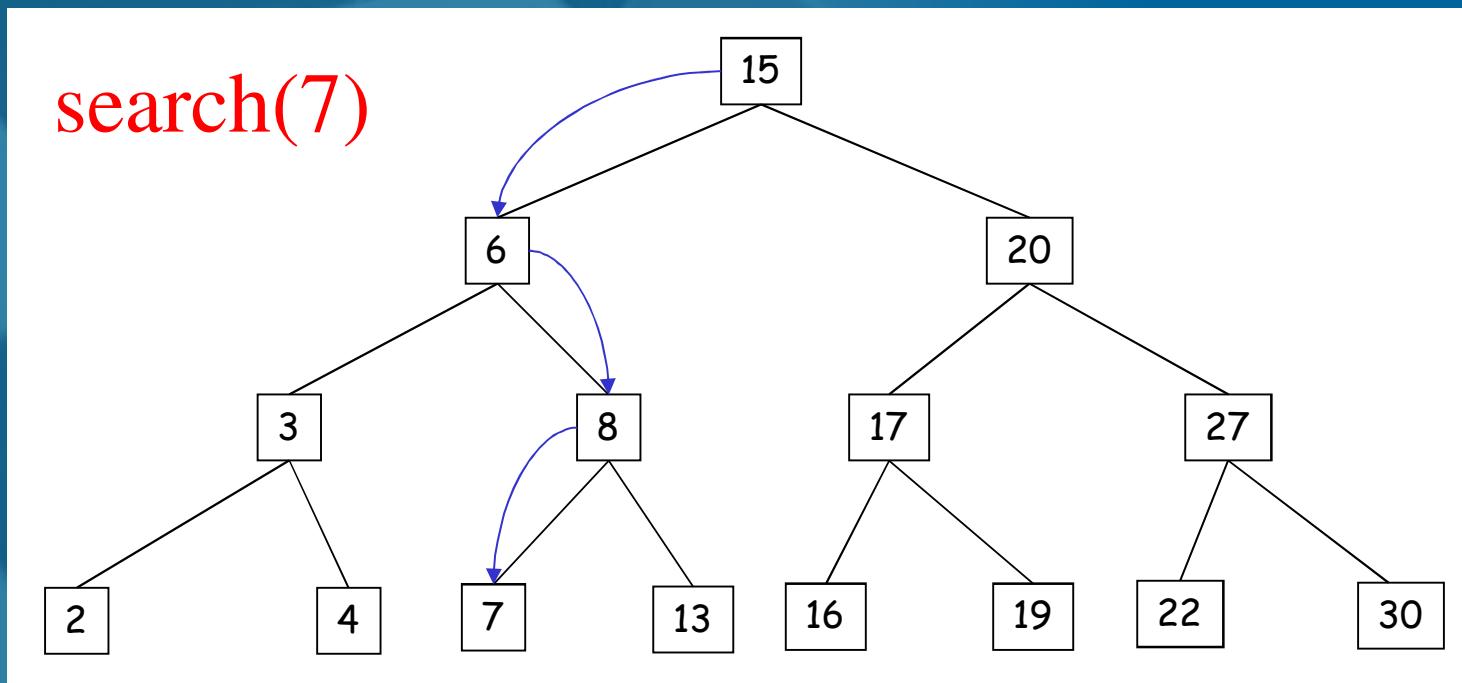


Albero binario  
di ricerca

Albero binario  
non di ricerca

# search(chiave k) -> elem

- Costo della ricerca su un albero non di ricerca?
- Sfruttando la proprietà di ricerca, partendo dalla radice, su ogni nodo decidiamo se proseguire nel sottoalbero sinistro o destro



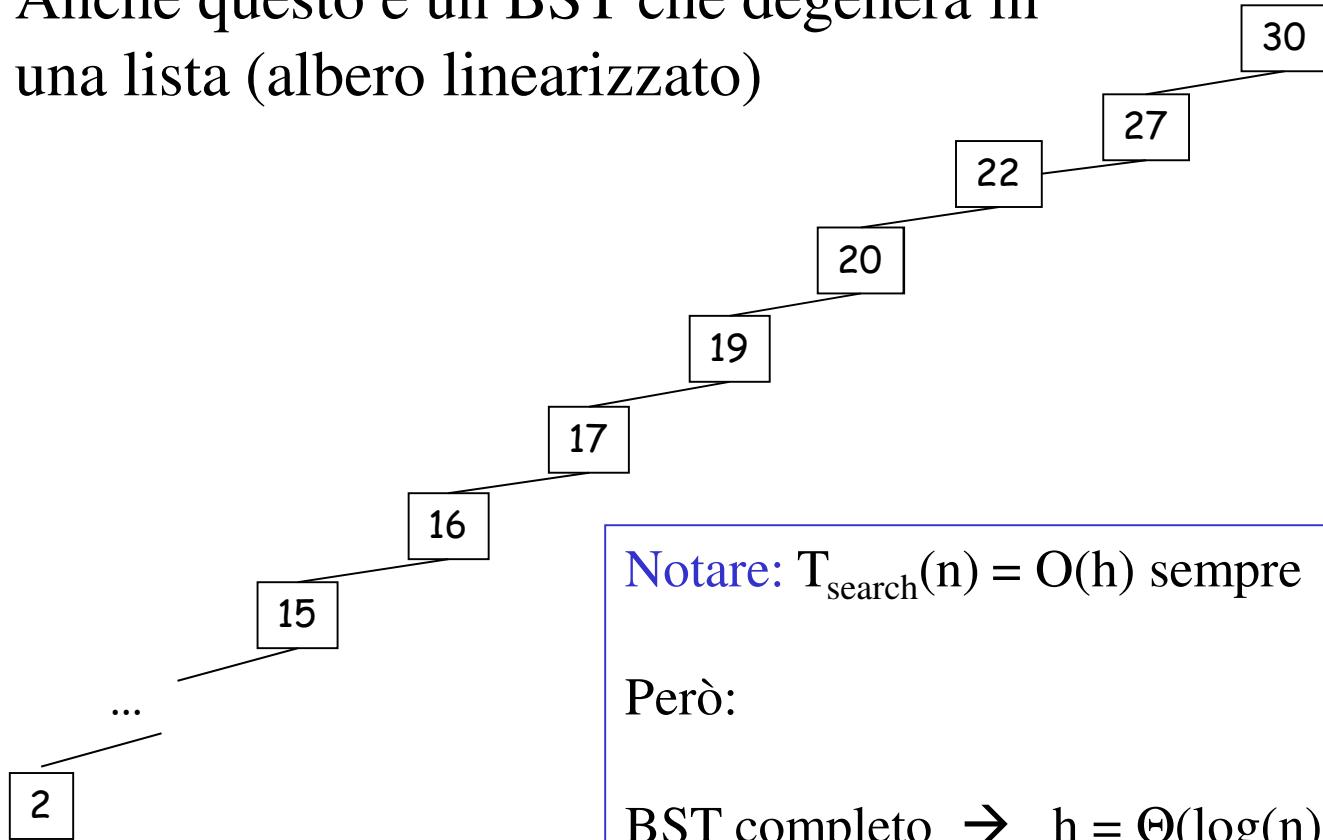
# search(chiave k) -> elem

- Traccia un cammino nell'albero partendo dalla radice:  
su ogni nodo, **usa la proprietà di ricerca** per decidere se proseguire nel sottoalbero sinistro o destro
- Costo **O(h)**

**algoritmo** search(*chiave k*) → *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*

Anche questo è un BST che degenera in una lista (albero linearizzato)



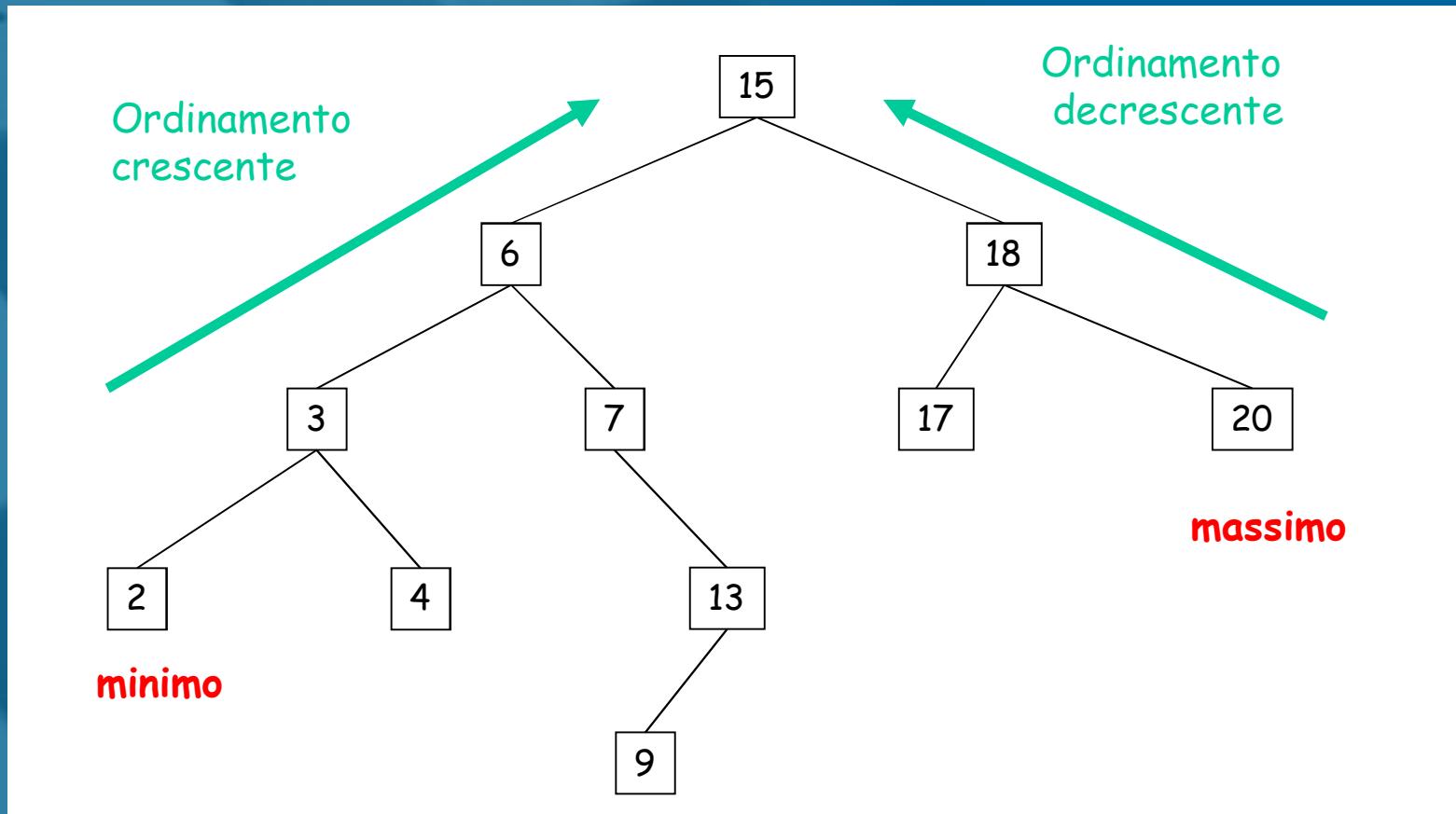
Notare:  $T_{\text{search}}(n) = O(h)$  sempre

Però:

BST completo  $\rightarrow h = \Theta(\log(n))$   
BST “linearizzato”  $\rightarrow h = \Theta(n)$

# Ricerca del massimo e del minimo

- Dove si trovano gli elementi di chiave minima e massima, rispettivamente?





# Ricerca del massimo

**algoritmo** max(*nodo u*) → *nodo*

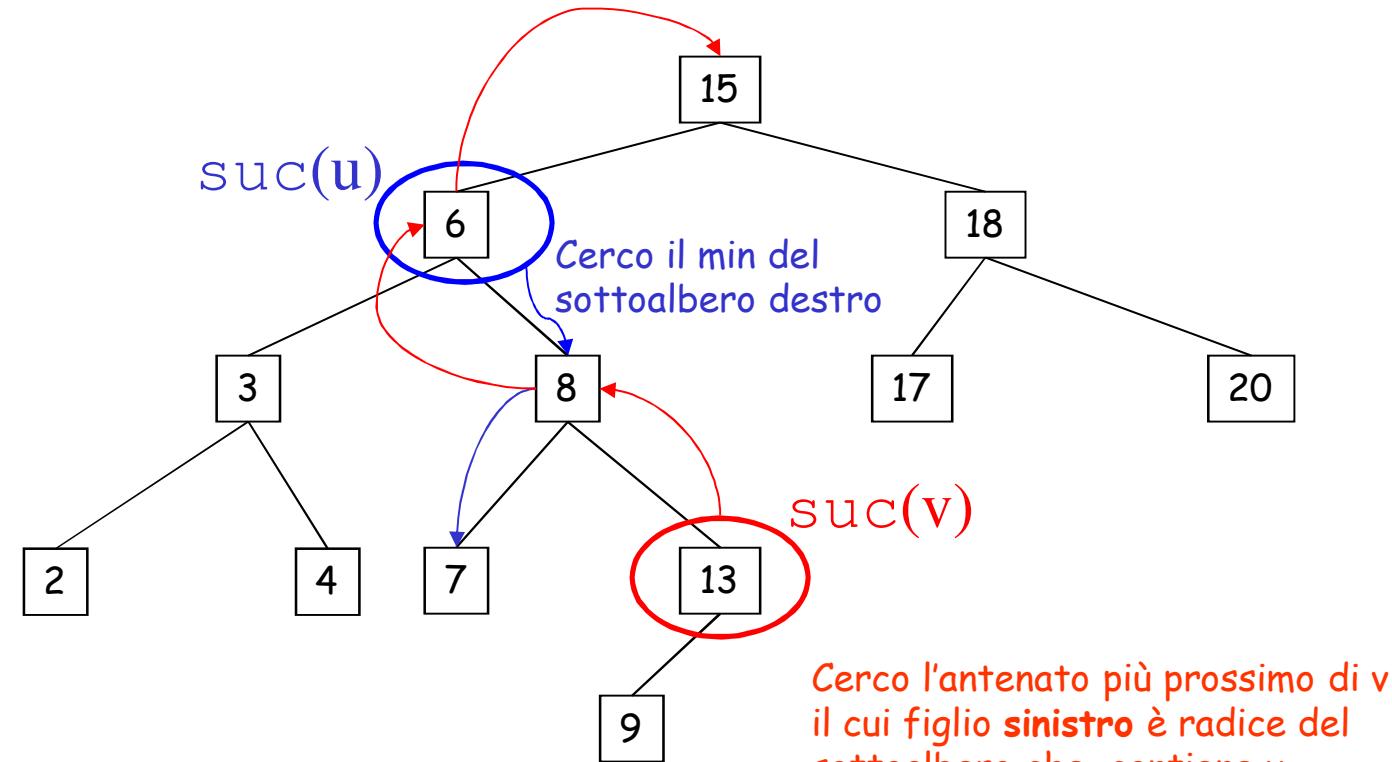
1.       $v \leftarrow u$
2.      **while** ( figlio destro di  $v \neq \text{null}$  ) **do**
3.             $v \leftarrow \text{figlio destro di } v$
4.      **return**  $v$

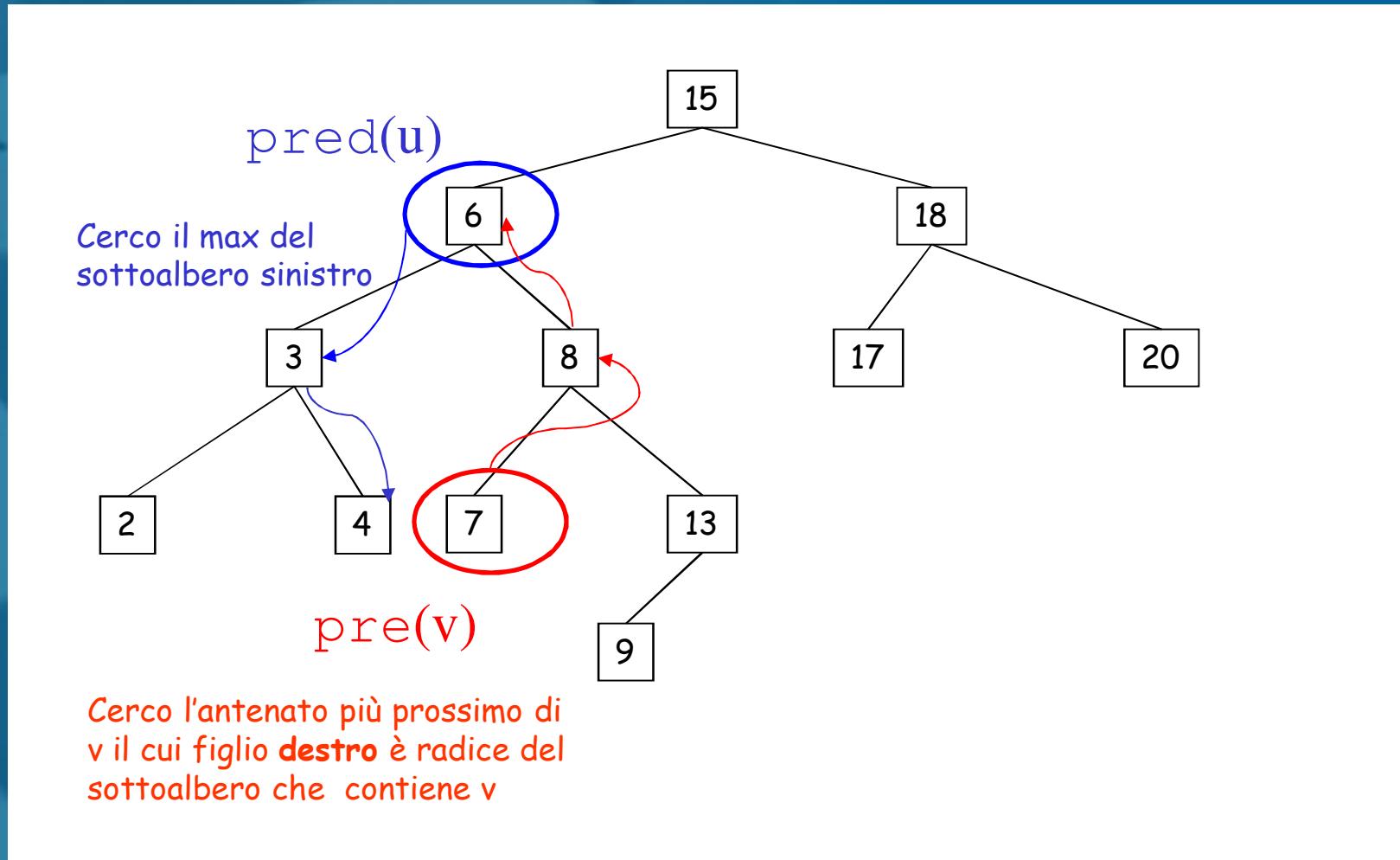
**Nota:** è possibile definire una procedura min(*nodo u*) in maniera del tutto analoga



# Predecessore e successore

- Il **predecessore** di un nodo  $u$  in un BST è il nodo  $v$  nell'albero di **chiave massima**  $\leq \text{chiave}(u)$
- Il **successore** di un nodo  $u$  in un BST è il nodo  $v$  nell'albero di **chiave minima**  $\geq \text{chiave}(u)$
- Come trovo il predecessore/successore di un nodo in un BST?

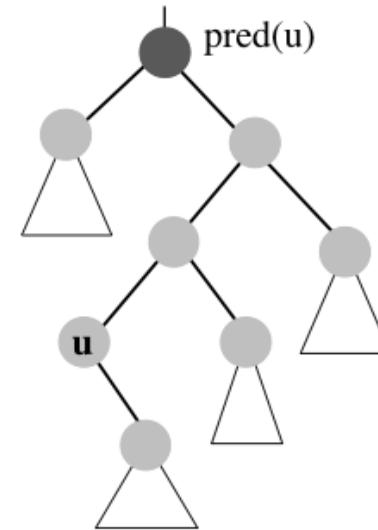
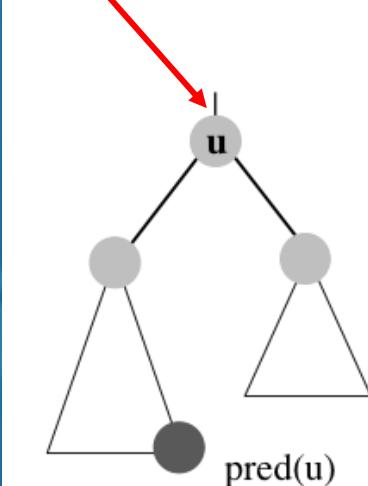




# Ricerca del predecessore

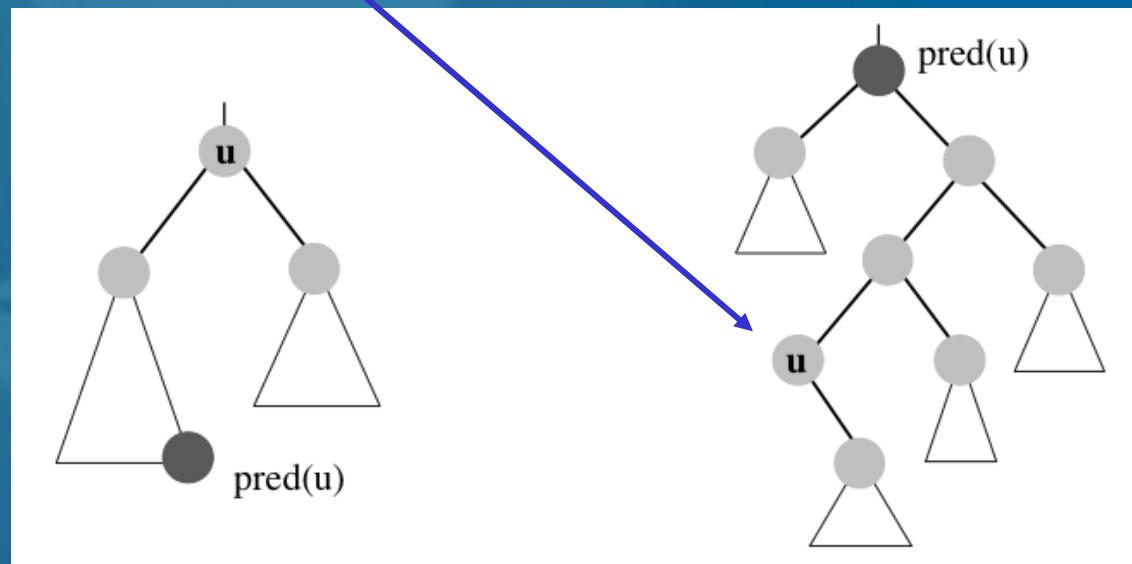
**algoritmo pred(*nodo u*) → *nodo***

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



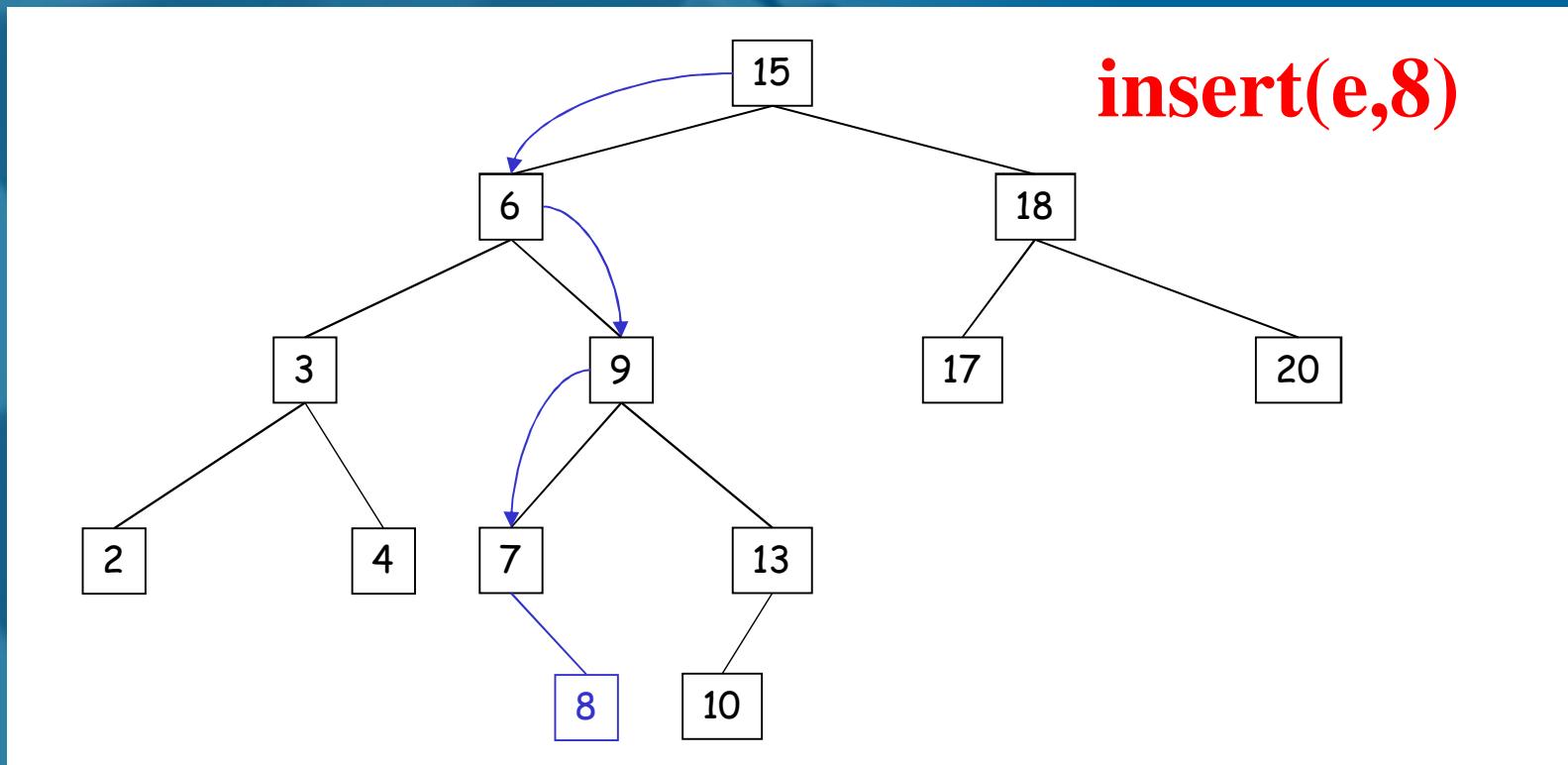
# Ricerca del predecessore

```
algoritmo pred(nodo u) → nodo
1. if (u ha figlio sinistro sin(u)) then
2. return max(sin(u))
3. while (parent(u) ≠ null e u è figlio sinistro di suo padre) do
4. u ← parent(u)
5. return parent(u)
```



# insert(elem e, chiave k)

**Idea:** aggiunge la nuova chiave **come nodo foglia** simulando una ricerca con la chiave da inserire per individuare la corretta posizione





# insert(elem e, chiave k)

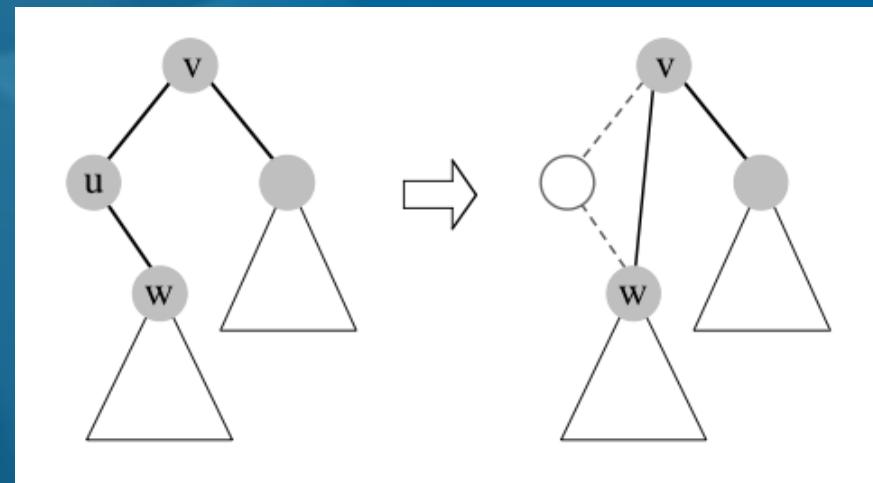
1. Crea un nuovo nodo u con elem=e e chiave=k
  2. Cerca la chiave k nell'albero, identificando così il nodo v che diventerà padre di u
  3. Appendi u come figlio sinistro/destro di v in modo che sia mantenuta la proprietà di ricerca
- Vedi implementazione in Java (src\_BST):
    - AlberoBR.java
    - Dizionario.java
    - ecc..

# delete(chiave k)

Sia u il nodo contenente l'elemento da cancellare:

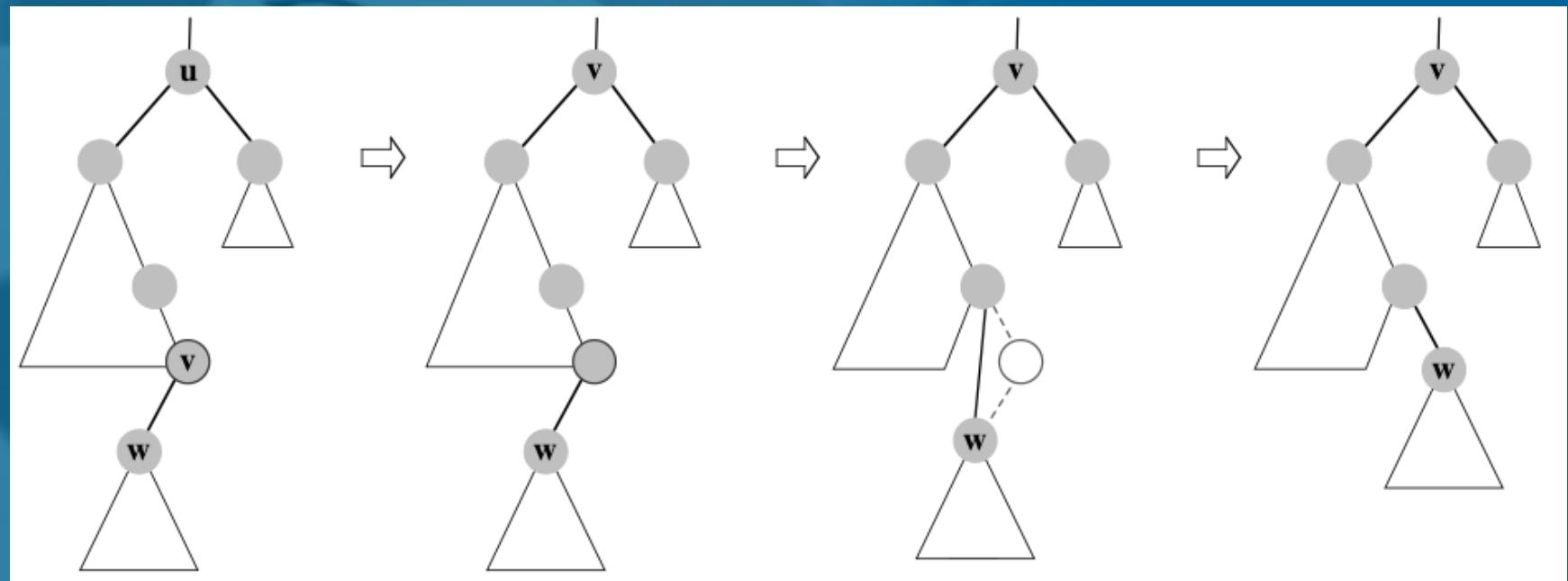
1) u è una foglia: rimuovila

2) u ha un solo figlio:

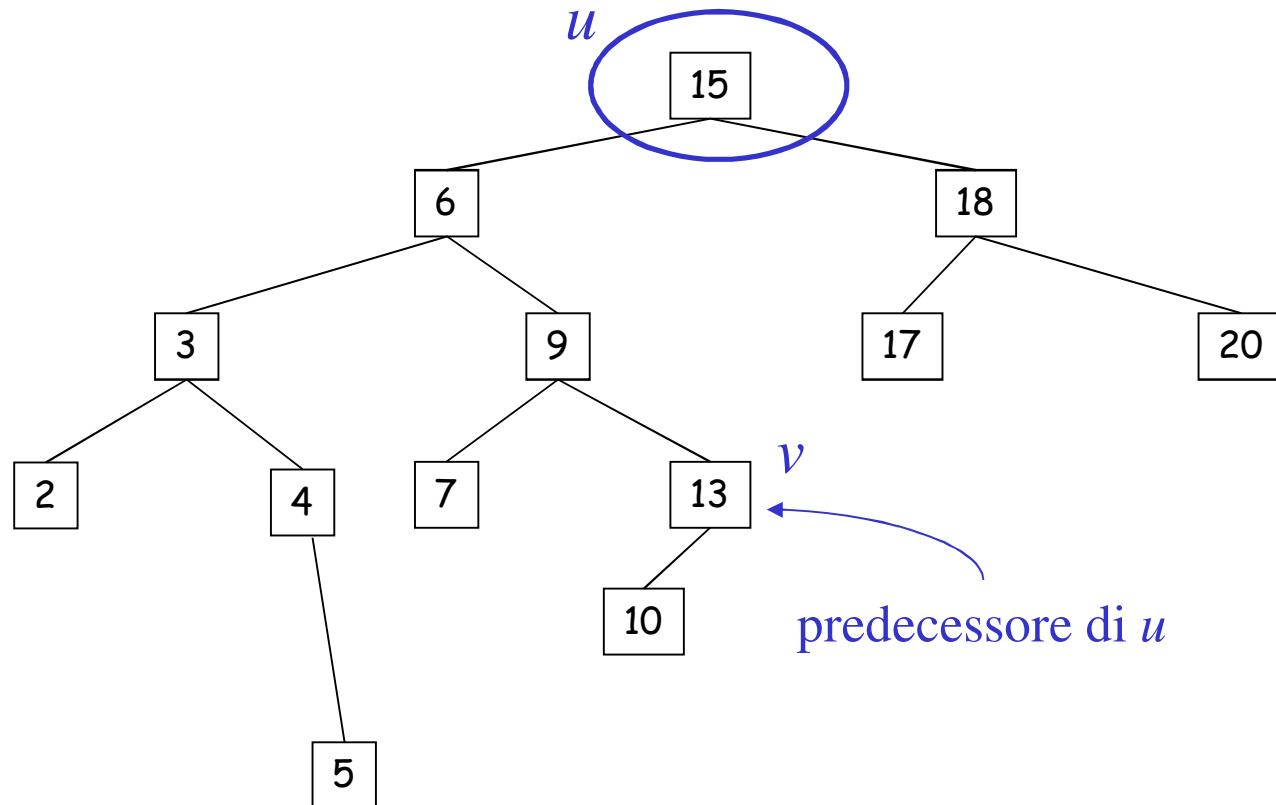


## delete(chiave k)

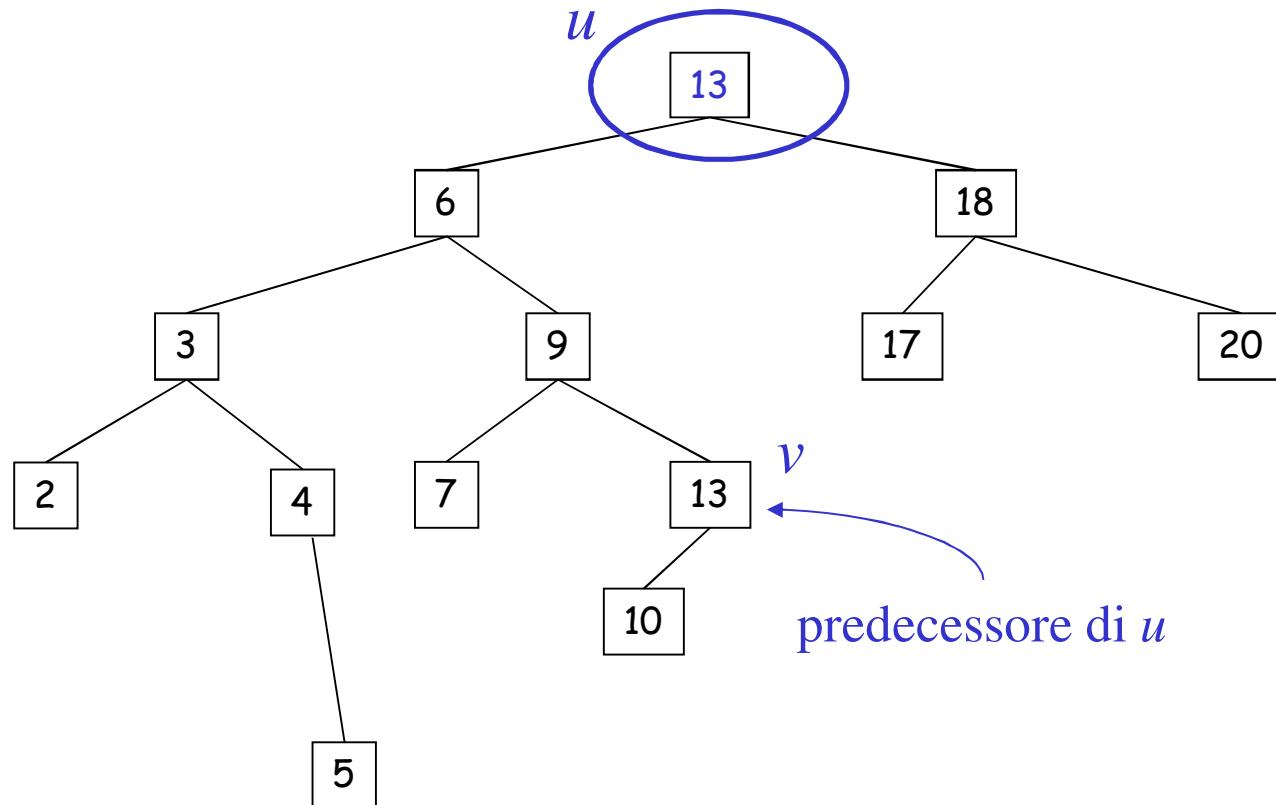
- 3) u ha due figli: sostituiscilo con il predecessore (v), e rimuovi fisicamente il predecessore
- Il predecessore ha un solo figlio al più, casi 1) o 2)
  - In alternativa, si può usare il successore di u.



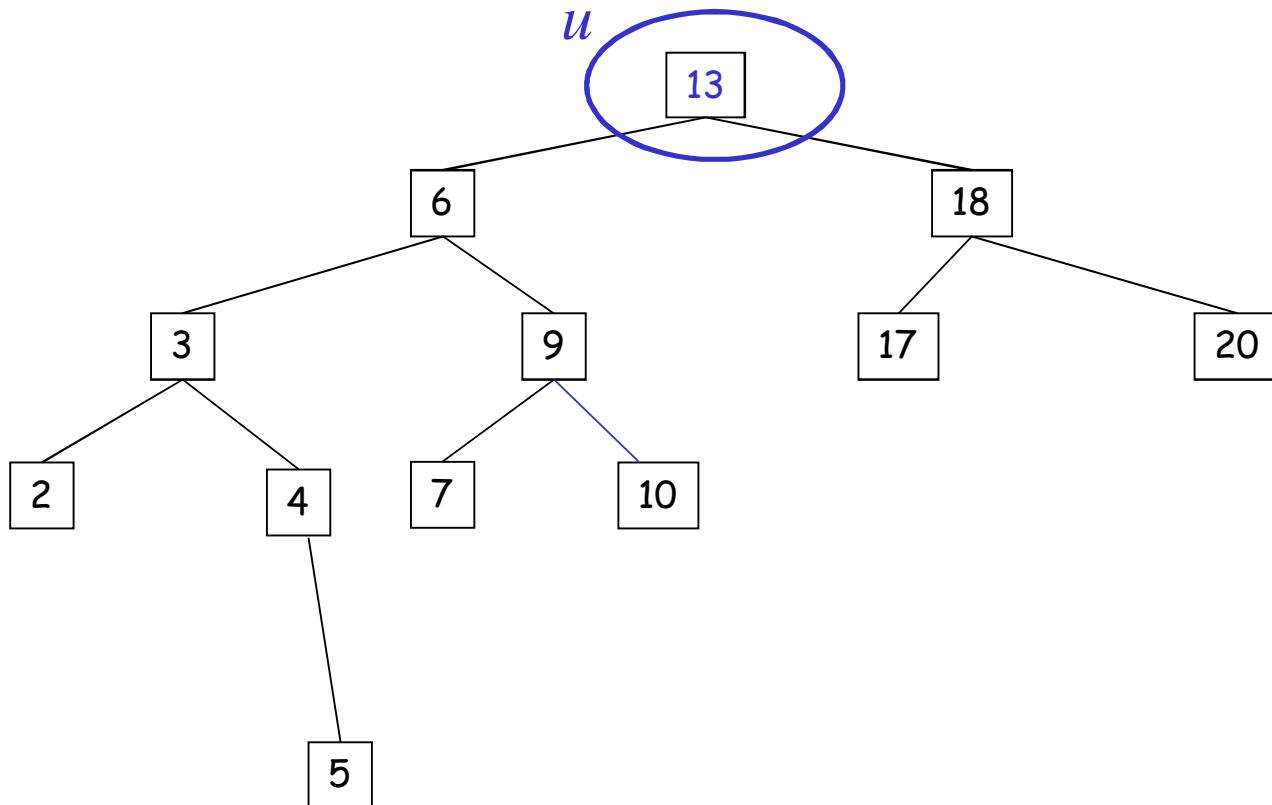
# delete (15)



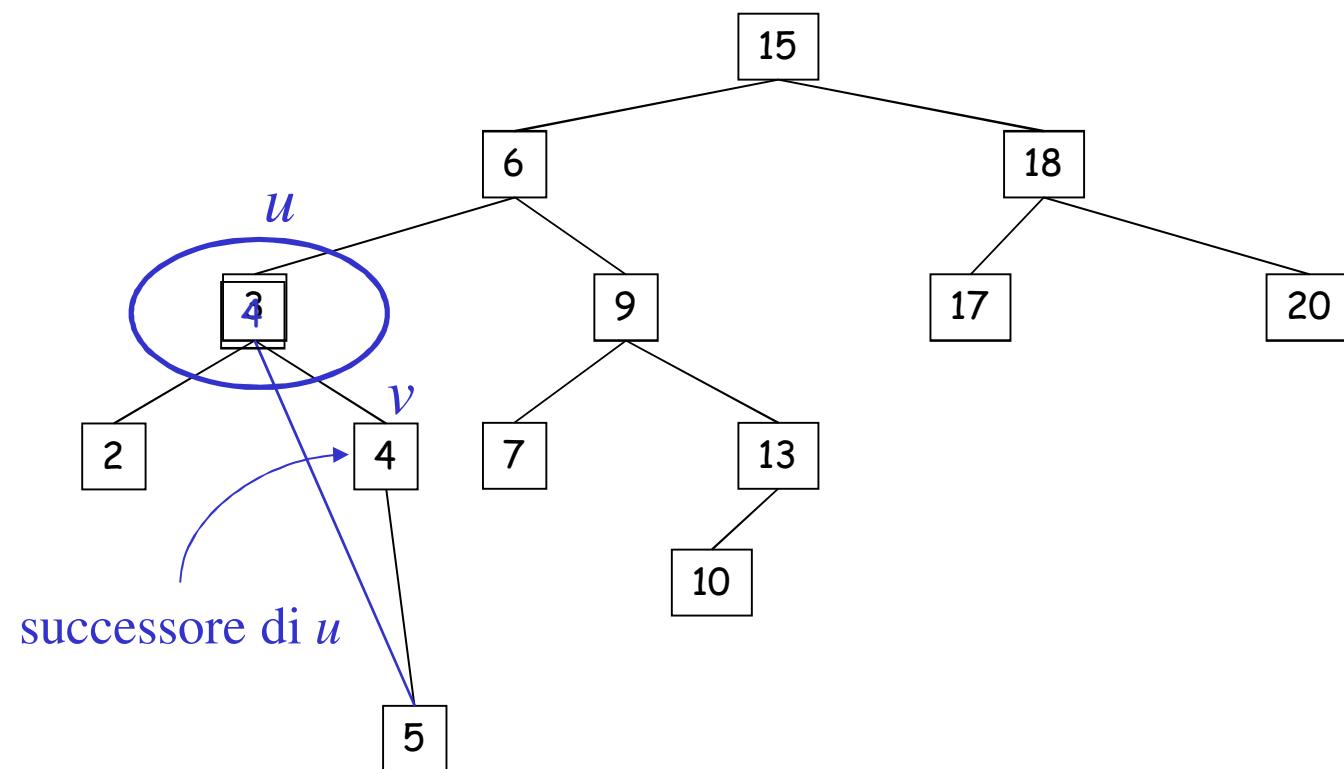
# delete (15)



# delete (15)



## delete (3)





# delete(chiave k)

- No pseudocodice
- Vedi direttamente l'implementazione in Java nei file:
  - AlberoBR.java
  - Dizionario.java
  - ecc..della cartella <src\_BST>



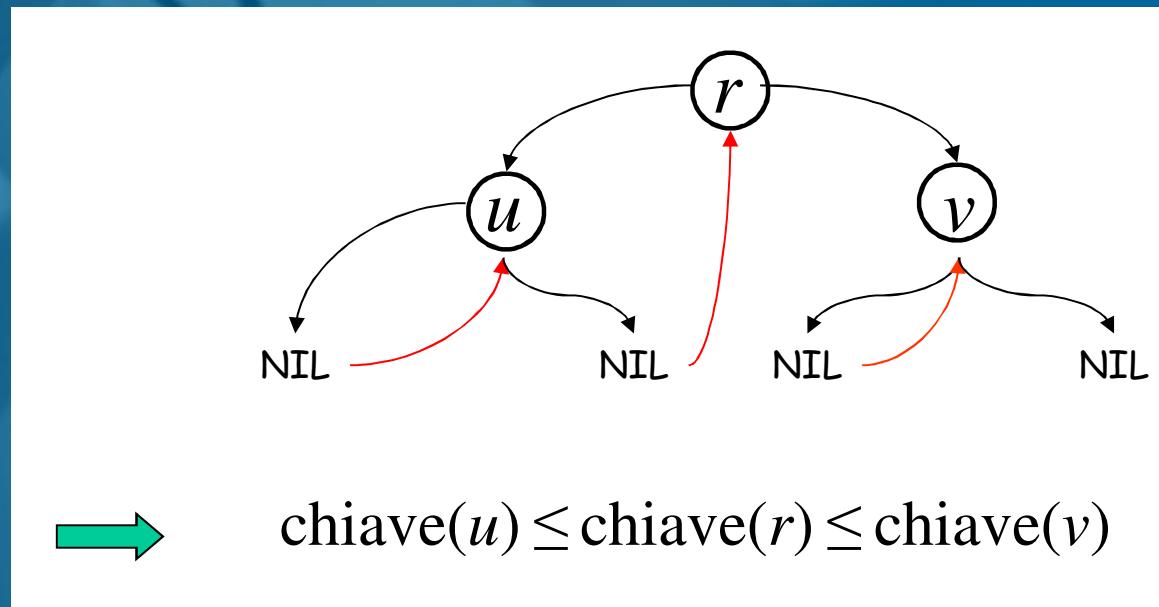
# Visita simmetrica di un BST

- Visita in ordine simmetrico – dato un nodo x, elenco prima il sotto-albero sinistro di x (in ordine simmetrico), poi il nodo x, poi il sotto-albero destro (in ordine simmetrico)
- Una visita simmetrica di un BST equivale a visitare i nodi del BST in ordine crescente rispetto alla chiave!

**Verifica di correttezza –** Supponiamo, per semplicità, che l'albero sia completo. Indichiamo con  $h$  l'altezza dell'albero.

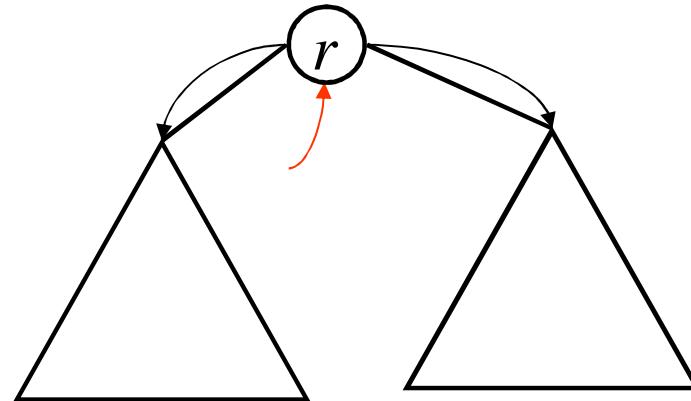
Vogliamo mostrare che la visita in ordine simmetrico restituisce la sequenza ordinata

Per induzione sull'altezza:  $h=1$  (caso base)



## Verifica correttezza (continua ...)

$h = \text{generico}$  (ipotizzo che la procedura sia corretta per  $h-1$ )



Albero di altezza  $h-1$ .  
Tutti i suoi elementi sono  
minori o uguali della  
radice

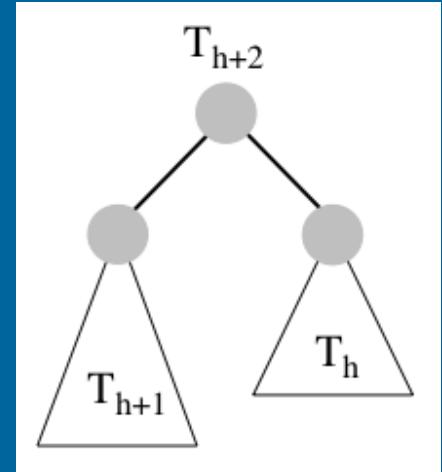
Albero di altezza  $h-1$ .  
Tutti i suoi elementi sono  
maggiori o uguali della  
radice

# Costo delle operazioni dipende da h

- Tutte le operazioni hanno costo  $O(h)$  nel caso peggiore, dove  $h$  è l'altezza dell'albero
  - $O(n)$  nel caso peggiore, e cioè per alberi molto sbilanciati e profondi
    - Ad es. un albero che “degenera” in una lista ha  $h=n-1$
  - $O(\log n)$  nel caso peggiore, per un albero molto bilanciato
    - Ad es. un albero binario completo con  $n$  nodi
      - Sommando i nodi per livelli: 
$$n = \sum_{i=0}^h 2^i = 2^{h+1} - 1$$
      - Passando al logaritmo: 
$$h = \log(n + 1) - 1 = \Theta(\log n)$$

# Alberi binari bilanciati - definizioni

- Fattore di bilanciamento di un nodo v:  
 $\beta(v) = | h(\text{sinistro}(v)) - h(\text{destro}(v)) |$



- Un albero binario si dice **bilanciato in altezza** se ogni nodo v ha:  $\beta(v) \leq 1$ 
  - Un **albero binario completo** ha  $\beta(v) = 0$  su ogni nodo v
  - Un albero binario che **degenera in una lista** ha  $\beta$  pari ad h

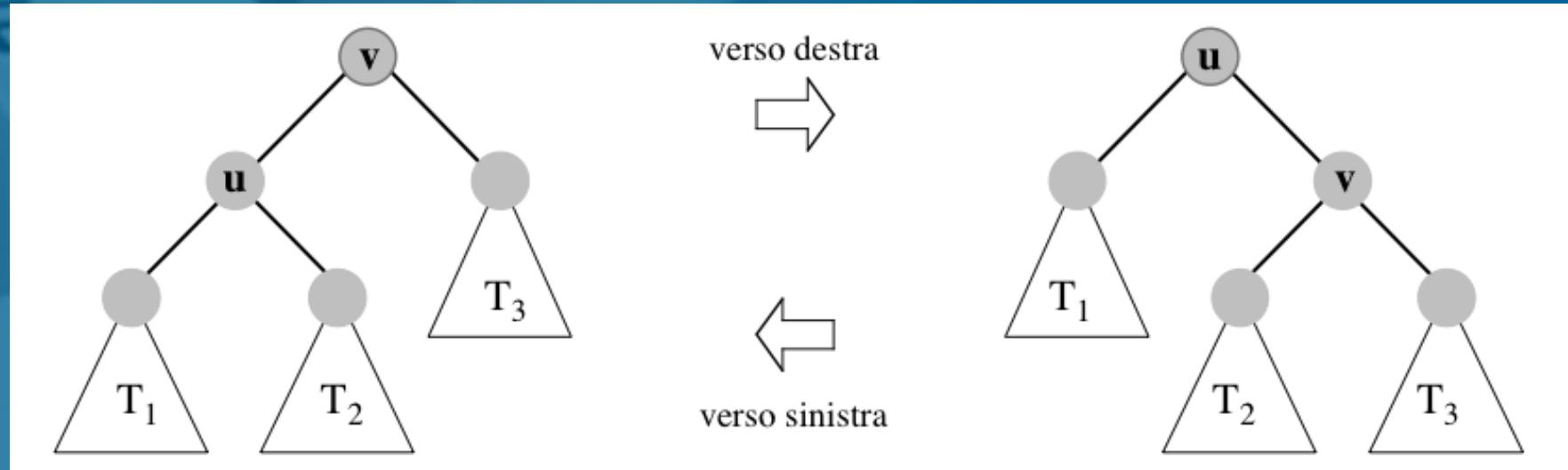
Alberi **AVL** = alberi binari di ricerca bilanciati in altezza dai noti ideatori (Adel'sonVel'skiǐ e Landis)

*Un albero AVL con n nodi ha altezza  $h = O(\log n)$  [dimostrabile]*

# Mantenere il bilanciamento

- Mantenere il bilanciamento sembra cruciale per ottenere buone prestazioni
  - L'operazione search non crea problemi
  - Ma inserimenti e cancellazioni potrebbero sbilanciare l'albero
- Esistono vari approcci per mantenere il bilanciamento attraverso **trasformazioni dell'albero**:
  - Tramite **rotazioni** anche in cascata (alberi AVL, alberi SPLAY o auto-aggiustanti)
  - Tramite **separazioni** o fusioni di nodi (alberi 2-3, alberi **rossoneri**, B-alberi) con grado variabile
- In tutti questi casi si ottengono tempi di esecuzione logaritmici nel caso peggiore

# Rotazione di base



- Mantiene la proprietà di ricerca
- Richiede tempo  $O(1)$

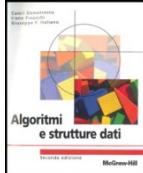
# insert(elem e, chiave k)

1. Crea un nuovo nodo u con elem=e e chiave=k
2. Inserisci u come in un BST
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  $\pm 2$  (nodo critico)
  4. Esegui una rotazione opportuna su v
    - Oss.: 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 (che potrebbe essere il predecessore del nodo contenente e)
3. Ripercorrendo il cammino dal basso verso l'alto, esegui l'opportuna rotazione semplice o doppia sui nodi sbilanciati
  - Oss.: potrebbero essere necessarie  $O(\log n)$  rotazioni



# 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**  $T(n) = O(\log n)$   
ereditata.

**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.



# Riepilogo

- Mantenere il **bilanciamento** sembra cruciale per ottenere buone prestazioni
- Esistono vari approcci per mantenere il bilanciamento:
  - Tramite **rotazioni**
  - Tramite **fusioni o separazioni** di nodi (alberi 2-3, B-alberi )
- In tutti questi casi si ottengono tempi di esecuzione logaritmici nel caso peggiore
- E' anche possibile non mantenere in modo esplicito alcuna condizione di bilanciamento, ed ottenere tempi logaritmici ammortizzati su una intera sequenza di operazioni (alberi auto-aggiustanti)



# B-alberi

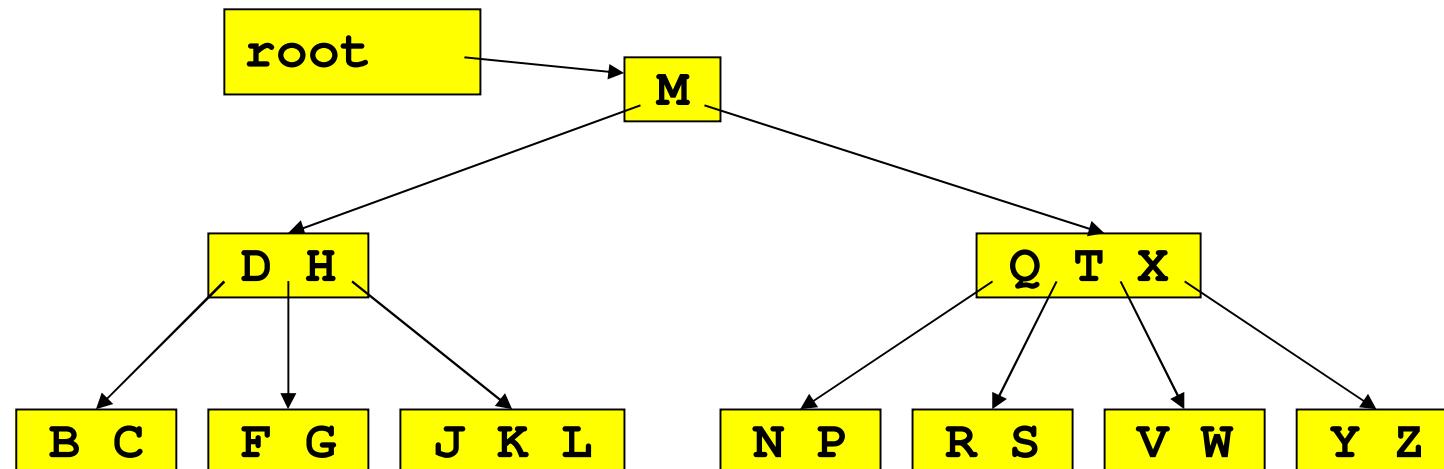
I **B**-alberi sono **alberi bilanciati** particolarmente adatti per memorizzare grandi quantità di **dati in memoria secondaria** (disco)

Sono progettati in modo tale da **minimizzare il numero di accessi a disco**

Le operazioni fondamentali sui **B**-alberi sono ***Insert*, *Delete* e *Search*** alle quali si aggiungono ***Split* e *Join*** per mantenere il bilanciamento

A differenza dei nodi degli alberi binari di ricerca che contengono una sola chiave ed hanno due figli, i nodi dei *B*-alberi possono contenere un numero  $n$  di chiavi ed avere  $n+1$  figli con  $n \geq 1$

*Esempio di B-albero* (per le lettere dell’alfabeto):





# Costi operazioni

Nel valutare la complessità delle operazioni, terremo separate le due componenti:

**1. tempo di lettura-scrrittura su disco** (che assumiamo proporzionale al numero di blocchi letti-scritti su disco)

- Ordine dei millisecondi (es. 10-20 msec)

**2. tempo di CPU** (tempo di calcolo in memoria centrale)

- Ordine del microsecondo (es. 1/5-1/50  $\mu$ sec)



L'operazione di lettura da disco è:

***DiskRead(x)***

che, dato il riferimento  $x$  ad un oggetto, legge l'oggetto da disco in memoria centrale.

Assumiamo che sia possibile accedere al valore di un oggetto soltanto dopo che esso sia stato letto in memoria centrale.

Assumiamo inoltre che, se l'oggetto da leggere si trova già in memoria centrale, l'operazione ***DiskRead(x)*** non abbia alcun effetto.



L'operazione di scrittura su disco è :

**DiskWrite( $x$ )**

che, dato il riferimento  $x$  ad un oggetto presente in memoria centrale, scrive tale oggetto su disco.

Tipicamente **l'elaborazione di un oggetto residente su disco** segue lo schema:

**DiskRead( $x$ )**

.....

▷ **elaborazioni che accedono/modificano  $x$ .**

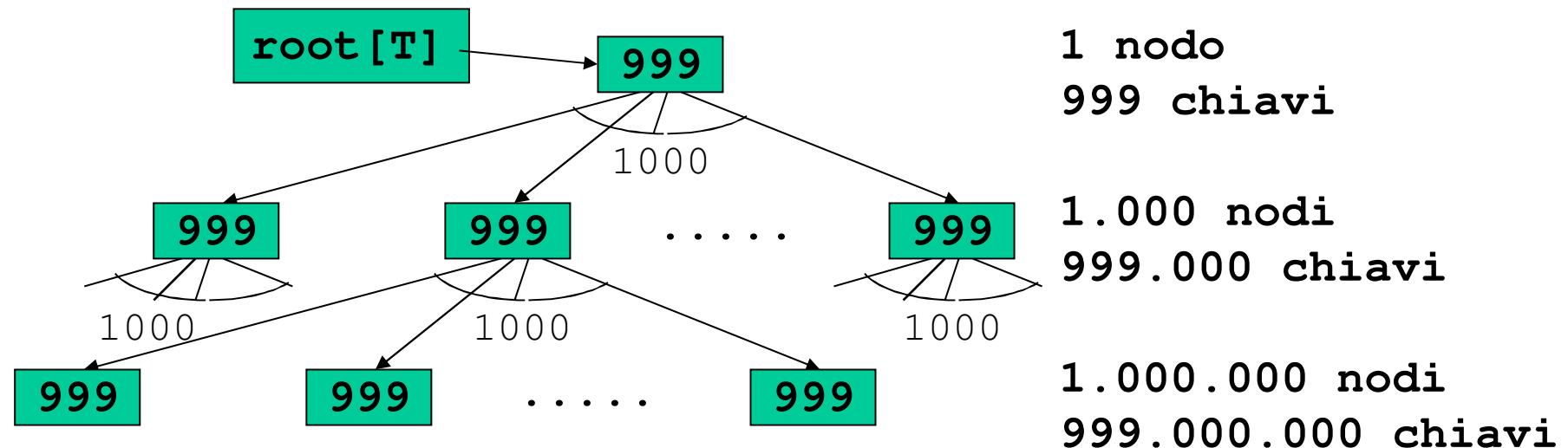
**DiskWrite( $x$ )** ▷ **non necessaria se  $x$  resta invariato**

.....

▷ **elaborazioni che non modificano  $x$ .**

**Un elevato grado di diramazione riduce in modo drastico sia l'altezza dell'albero che il numero di letture da disco necessarie per cercare una chiave.**

La figura seguente mostra che un *B*-albero di altezza 2 con un grado di diramazione 1000 può contenere  $10^9 - 1$  chiavi (un miliardo).





## Definizione di *B*-albero

Un *B*-albero  $T$  è un albero di radice  $\text{root}[T]$  tale che:

1. Ogni nodo  $x$  contiene i seguenti campi:
  - a)  $n[x]$  : il numero di chiavi presenti nel nodo  
 $n[x] + 1$  : il numero di figli del nodo
  - b)  $key_1[x] \leq key_2[x] \leq \dots \leq key_{n[x]}[x]$  : le  $n[x]$  chiavi in ordine non decrescente
  - c)  $leaf[x]$  : valore booleano che è *true* se il nodo  $x$  è foglia, *false* altrimenti;



2. se il nodo non è una foglia contiene anche
  - d)  $c_1[x], c_2[x], \dots, c_{n[x]+1}[x]$  : gli  $n[x]+1$  puntatori ai figli;
3. le  $n[x]$  chiavi  $key_1[x], key_2[x], \dots, key_{n[x]}[x]$  di un nodo interno separano gli **intervalli contenenti le chiavi dei sottoalberi**.  
In altre parole, se  $k_1, k_2, \dots, k_{n[x]+1}$  sono chiavi appartenenti rispettivamente ai sottoalberi di radici  $c_1[x], c_2[x], \dots, c_{n[x]+1}[x]$  allora:  
$$k_1 \leq key_1[x] \leq k_2 \leq key_2[x] \leq k_3 \leq \dots \leq k_{n[x]} \leq key_{n[x]}[x] \leq k_{n[x]+1}.$$



4. Le foglie sono tutte allo stesso livello  $h$  detto altezza dell'albero.
5. Vi è un *limite superiore* ed un *limite inferiore* al numero di chiavi contenuto in un nodo e tali limiti dipendono da una **costante  $t$**  detta **grado minimo** del  $B$ -albero. Precisamente:
  - a) ogni nodo, eccetto la radice, ha **almeno  $t - 1$**  chiavi e **almeno  $t$**  figli (se non è una foglia):
$$n[x] \geq t - 1$$
  - b) se l'albero non è vuoto, **la radice contiene almeno una chiave** e **se la radice non è foglia ha almeno due figli**.



- c) ogni nodo ha **al più  $2t - 1$**  chiavi e **al più  $2t$**  figli (se non è foglia):  $n[x] \leq 2t - 1$   
Un nodo con  $2t - 1$  chiavi si dice *pieno*.

Ad ogni chiave sono generalmente associate delle informazioni ausiliarie. Assumeremo implicitamente che quando viene copiata una chiave vengano copiate anche tali informazioni.

I  **$B$ -alberi** più semplici sono quelli con grado minimo  $t = 2$ . Ogni nodo interno ha **2, 3 o 4** figli. In questo caso si dicono anche **2-3-4-alberi**.



## Altezza di un *B*-albero

### Proprietà.

Ogni *B*-albero di grado minimo  $t$  contenente  $N$  chiavi ha altezza  $h \leq \log_t \frac{N+1}{2}$  (assumendo per convenzione che l'albero vuoto abbia altezza -1).

### Dimostrazione.

Invece della diseguaglianza  $h \leq \log_t \frac{N+1}{2}$

dimostriamo quella equivalente  $N \geq 2t^h - 1$ .



Se l'albero è vuoto  $h = -1$  ed  $N = 0 \geq 2t^{-1} - 1$ .

Supponiamo quindi che l'albero non sia vuoto e sia  $r$  la sua radice ed  $h \geq 0$  la sua altezza.

Sia  $m_i$  il numero di nodi a livello  $i$ .

Allora:  $m_0 = 1$ ,

$m_1 = n[\text{root}] + 1 \geq 2$  (la radice ha almeno una chiave)

$m_i \geq t m_{i-1}$  per  $i > 1$  (ogni nodo ha almeno  $t$  figli)  
 $\geq t^{i-1} m_1 \geq 2 t^{i-1}$

e quindi:  $m_i \geq 2t^{i-1}$  per ogni  $i \geq 1$



Quindi per le chiavi:

$$\begin{aligned}N &= \sum_x n[x] \\&= n[root] + \sum_{i=1}^h \sum_{x \text{ di livello } i} n[x] \\&\geq 1 + \sum_{i=1}^h 2t^{i-1}(t-1) \\&= 1 + 2(t-1) \frac{t^h - 1}{t-1} \\&\geq 1 + 2(t^h - 1) = 2t^h - 1 \quad \text{c.v.d.}\end{aligned}$$

L'altezza di un ***B***-albero è  $O(\log_t N)$ , dello stesso ordine  $O(\log_2 N)$  degli alberi binari bilanciati, ma la costante nascosta nella  $O$  è inferiore di un fattore  $\log_2 t$  che, per  $50 \leq t \leq 2000$ , è compreso tra 5 e 11.



# Operazioni elementari

Adottiamo le seguenti convenzioni per le operazioni sui *B*-alberi:

1. La radice del *B*-albero è sempre in memoria.
2. I nodi passati come parametri alle funzioni sono stati preventivamente letti in memoria.



Defineremo le seguenti operazioni:

- *BTree* costruttore di un albero vuoto;
- *Search* che cerca una chiave nell’albero;
- *Insert* che aggiunge una chiave;
- *Delete* che toglie una chiave.

ci serviranno anche tre procedure ausiliarie:

- *SearchSubtree*
- *SplitChild*
- *InsertNonfull*



## *BTree*

Un *B*-albero vuoto si costruisce con la procedura:

```
BTree(T)
 root[T] \leftarrow nil
```

la cui complessità è  $O(1)$ .

## *Search*

La procedura di ricerca in un *B*-albero è:

```
Search(T, k)
 if root[T] = nil then return nil
 else return SearchSubtree(root[T], k)
```

che si limita a richiamare la funzione ausiliaria  
*SearchSubtree*.



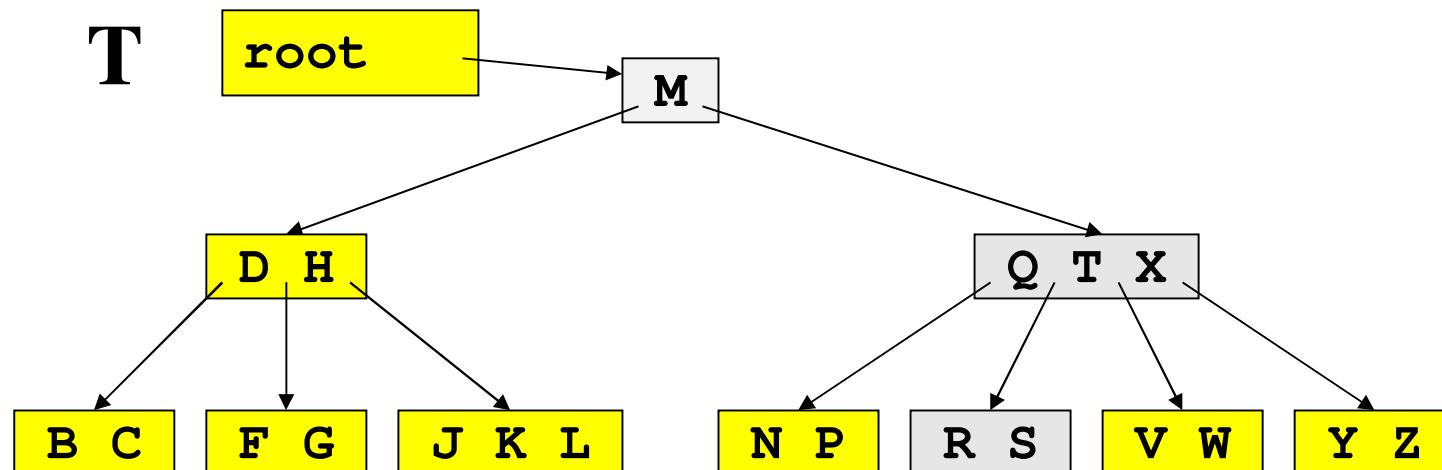
La procedura ausiliaria ***SearchSubtree*** restituisce  $(y, i)$  con  $y$  nodo ed  $i$  il più piccolo indice t.c.  $y.key_i = k$  se  $k$  si trova nel B-albero, nil altrimenti.

### ***SearchSubtree(x, k)***

1.  $i \leftarrow 1$
2. while  $i \leq n[x]$  and  $k > key_i[x]$  do
  3.      $i \leftarrow i + 1$  ▷ Ricerca dell'indice  $i$  t.c.:  $k \leq key_i[x]$   
▷ Invariante:  $key_{1..i-1}[x] < k \leq key_{i..n[x]}[x]$
  4.     if  $i \leq n[x]$  and  $k = key_i[x]$  return  $x, i$  ▷ Ric. successo
  5.     else if  $leaf[x]$  return nil ▷ Ric. senza successo
  6.     else ▷ Ric. ricorsiva nel sotto-albero  $c_i[x]$
  7.         DiskRead( $c_i[x]$ )
  8.         return ***SearchSubtree(c\_i[x], k)***

# Esempio di B-albero (per le lettere dell’alfabeto)

Search(T,R)





Il numero di **DiskRead** è al più uguale all'altezza  $h$  dell'albero ed è quindi  $O(h) = O(\log_t N)$  con  $N$  numero di chiavi nel B-albero .

Il tempo **T** di CPU di **Search** è:

$$T = O(t h) = O(t \log_t N)$$

poiché  $n[x] \leq 2t - 1 < 2t$ , e quindi il tempo impiegato per il ciclo while (righe 2-3) per ogni nodo è  $O(t)$ .

Essendo le chiavi in un nodo ordinate, si può fare di meglio con una ricerca binaria nel nodo:

$$T = O(\log t h) = O(\log t \log_t N) = O(\log N)$$



La procedura ausiliaria *SearchSubtree* con ricerca binaria nel nodo è:

```
SearchSubtree(x, k)
 $i \leftarrow 1, j \leftarrow n[x] + 1$
 ▷ INVARIANTE: $key_{1..i-1}[x] < k \leq key_{j..n[x]}[x]$
 while $i < j$ do ▷ Ricerca binaria
 if $k \leq key_{\lfloor(i+j)/2\rfloor}[x]$ then $j \leftarrow \lfloor(i+j)/2\rfloor$
 else $i \leftarrow \lfloor(i+j)/2\rfloor + 1$
 ▷ $key_{1..i-1}[x] < k \leq key_{i..n[x]}[x]$
 if $i \leq n[x]$ and $k = key_i[x]$ return x, i
 if $leaf[x]$ return nil
 DiskRead($c_i[x]$)
 return SearchSubtree($c_i[x], k$)
```



## Insert

Non possiamo aggiungere una chiave ad un nodo interno perché dovremmo aggiungere anche un sottoalbero.

Quindi l'**aggiunta di una chiave in un *B*-albero può avvenire soltanto in una foglia.**

Questo si può fare soltanto **se la foglia non è piena.**

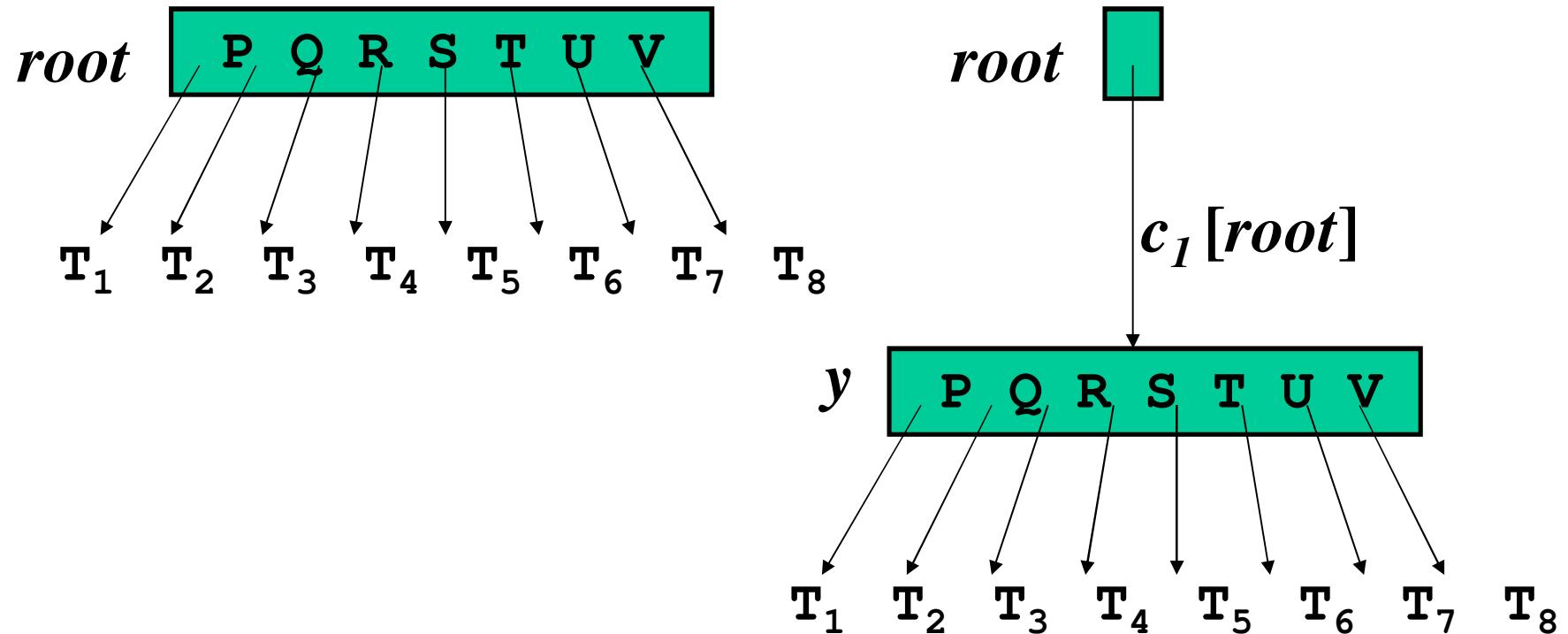


Possiamo assicurarci che la foglia a cui arriveremo non sia piena se durante la discesa dalla radice a tale foglia ci assicuriamo ad ogni passo che il figlio su cui scendiamo non sia pieno.

Nel caso in cui tale figlio sia pieno chiamiamo prima una particolare funzione *SplitChild* che lo divide in due parti.

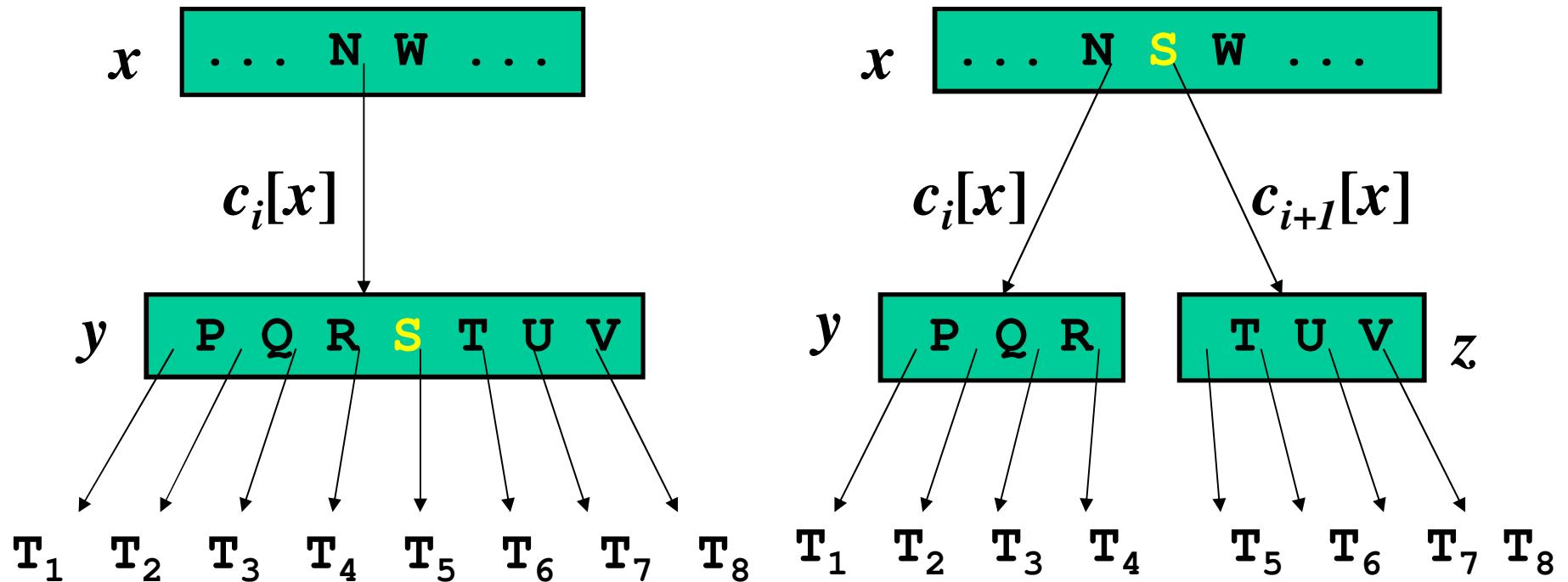
La stessa cosa dobbiamo fare all'inizio per la radice se essa è piena.

Ecco come funziona *Insert* se la radice è piena in un B-albero di grado minimo  $t = 4$ :



dopo di che richiama *SplitChild*.

Ecco come funziona *SplitChild* su un figlio pieno in un  $B$ -albero di grado minimo  $t = 4$ :



**Operazione “taglia ed incolla”:** il nodo  $z$  adotta i figli maggiori di  $y$  e diventa nuovo figlio di  $x$ , subito dopo  $y$ , e la chiave mediana si sposta da  $y$  ad  $x$  per separare  $y$  e  $z$



# La funzione ausiliaria *SplitChild* è:

*SplitChild(x, i, y)*

▷ PRECONDIZIONE:  $y$  è figlio  $i$ -esimo di  $x$  ed è pieno

$z \leftarrow \text{AllocateNode}()$

$\text{leaf}[z] \leftarrow \text{leaf}[y]$

▷ Sposta le ultime  $t - 1$  chiavi di  $y$  in  $z$

**for**  $j \leftarrow 1$  to  $t - 1$  **do**

$\text{key}_j[z] \leftarrow \text{key}_{j+t}[y]$

▷ Se  $y$  non è una foglia sposta gli ultimi  $t$  puntatori

▷ di  $y$  in  $z$

**if** not  $\text{leaf}[y]$  **then**

**for**  $j \leftarrow 1$  to  $t$  **do**

$c_j[z] \leftarrow c_{j+t}[y]$

$n[z] \leftarrow t - 1$

▷ Sposta la  $t$ -esima chiave di  $y$  in  $x$

for  $j \leftarrow n[x]$  down to  $i$  do

$key_{j+1}[x] \leftarrow key_j[x]$  ▷ fa spazio in  $x$  alla nuova chiave

for  $j \leftarrow n[x]+1$  down to  $i+1$  do

$c_{j+1}[x] \leftarrow c_j[x]$  ▷ fa spazio in  $x$  al nuovo figlio  $z$

$key_i[x] \leftarrow key_t[y]$  ▷ copia della  $t$ -esima chiave di  $y$  in  $x$

$c_{i+1}[x] \leftarrow z$  ▷ inserimento in  $x$  del nuovo figlio  $z$

$n[x] \leftarrow n[x] + 1$  ▷ ora  $x$  ha una chiave in più

▷ Rimuove le ultime  $t$  chiavi da  $y$

$n[y] \leftarrow t - 1$

▷ Scrive su disco i nodi modificati

$DiskWrite(x), DiskWrite(y), DiskWrite(z)$



La procedura di inserzione in un *B*-albero è:

*Insert*( $T, k$ )

if  $root[T] = \text{nil}$  then  $\triangleright$  l'albero è vuoto

$x \leftarrow \text{AllocateNode}()$

$n[x] \leftarrow 1, key_1[x] \leftarrow k, leaf[x] \leftarrow \text{true}$

$root[T] \leftarrow x$   $\triangleright$  Qui l'altezza cresce di 1

else  $\triangleright$  l'albero è non vuoto

if  $n[root[T]] = 2t - 1$  then  $\triangleright$  se la radice è piena

$y \leftarrow root[T]$

$x \leftarrow \text{AllocateNode}()$   $\triangleright$  x è la nuova radice

$n[x] \leftarrow 0, c_1[x] \leftarrow y, leaf[x] \leftarrow \text{false}$

$\text{SplitChild}(x, 1, y)$   $\triangleright$  Split della radice (nodo y)

$root[T] \leftarrow x$   $\triangleright$  Qui l'altezza cresce di 1

*InsertNonfull*( $root[T], k$ )



La funzione ausiliaria *InsertNonfull* è:

*InsertNonfull(x, k)*

▷ Precondizione:  $x$  non è pieno

1. if  $leaf[x]$  then      ▷ Inserisce la chiave  $k$  nel nodo  $x$
2.       $i \leftarrow n[x]+1$
3.      while  $i > 1$  and  $key_{i-1}[x] > k$  do
4.           $key_i[x] \leftarrow key_{i-1}[x], i \leftarrow i - 1$
5.           $key_i[x] \leftarrow k, n[x] \leftarrow n[x]+1$
6.      *DiskWrite(x)*

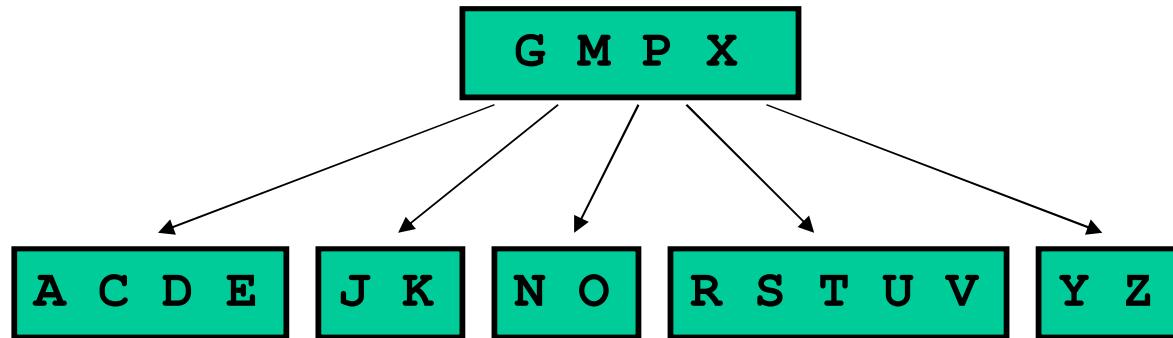
Se  $x$  è foglia, inseriamo  $k$  direttamente  
in  $x$  nella posizione appropriata



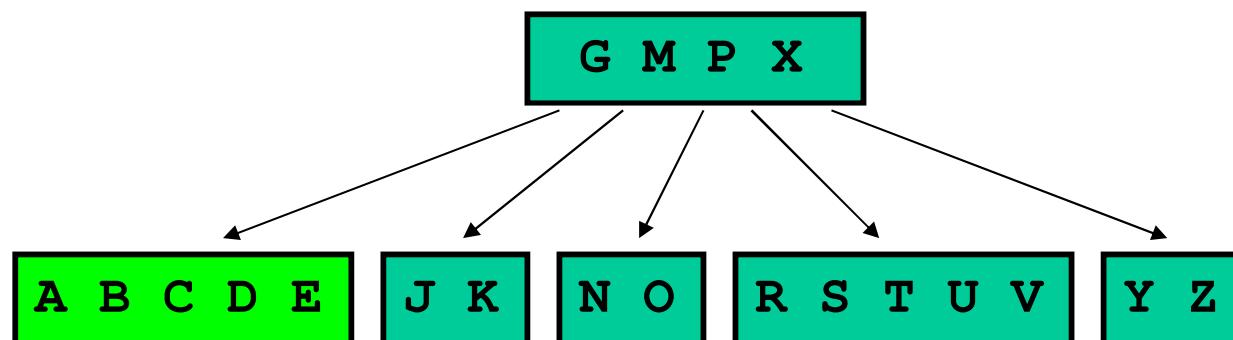
Se  $x$  non è foglia, determiniamo il figlio di  $x$  su cui far scendere la ricorsione (con eventuale split del figlio)

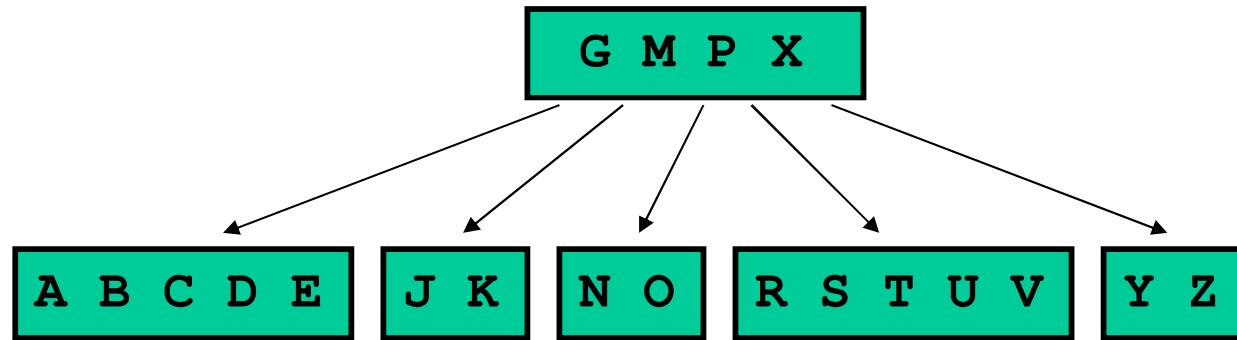


**Grado  $t = 3$**  al più 5 chiavi  
e al più 6 figli



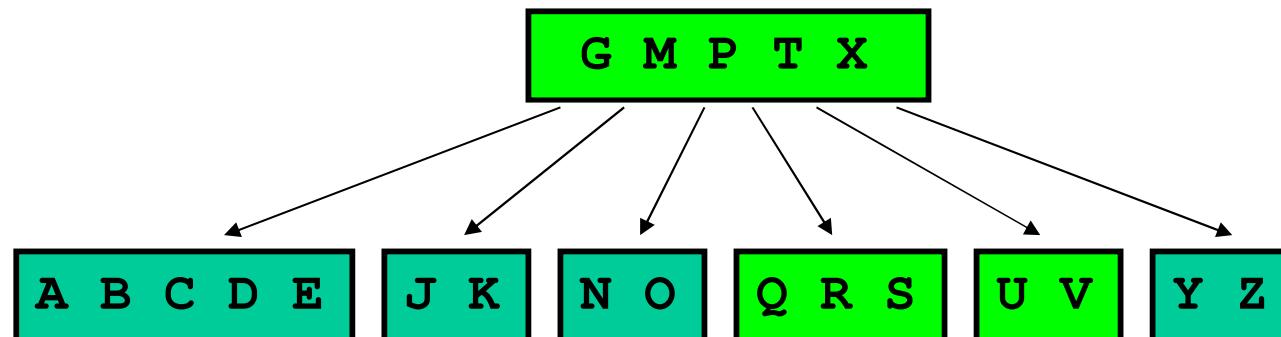
***Insert( $T, B$ )***

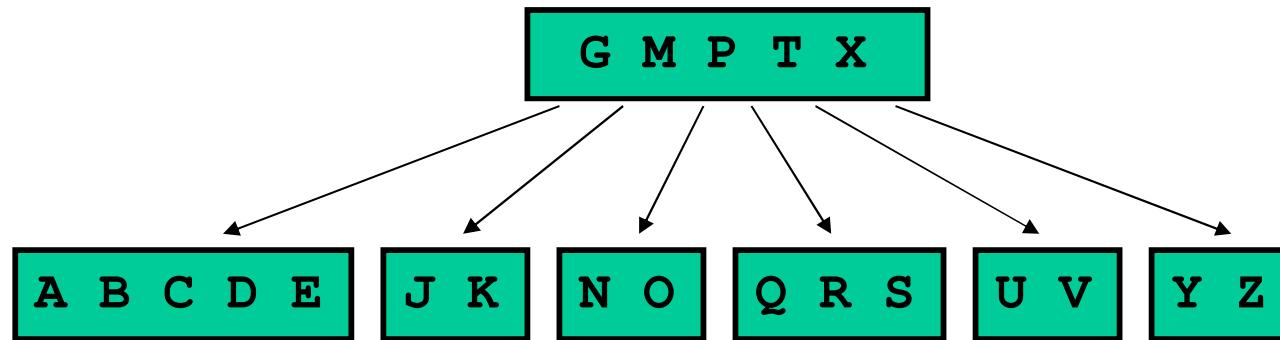




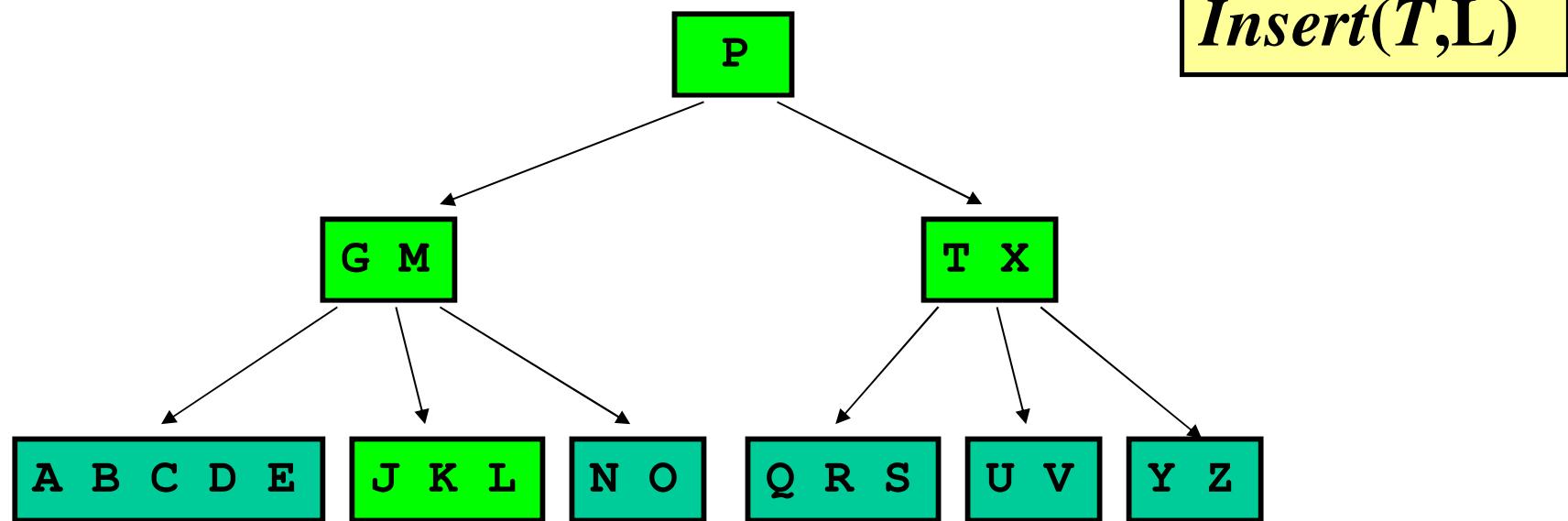
Split del figlio (R S T U V)  
Inserimento di Q nella metà a sx

*Insert(T,Q)*



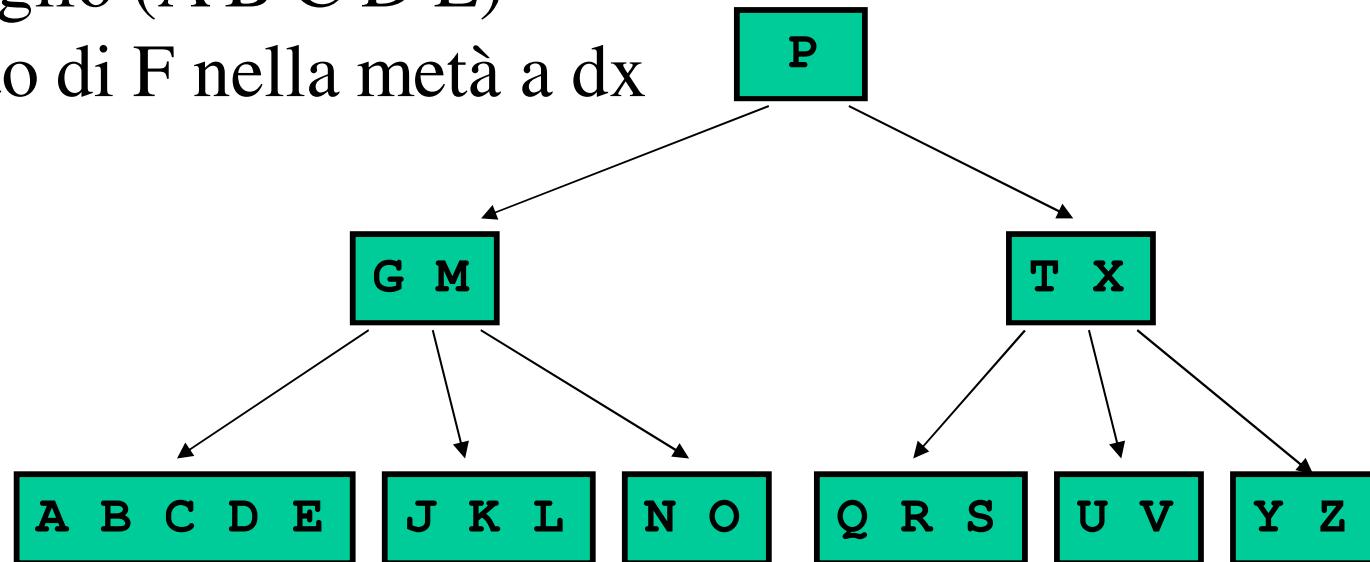


Split della radice che è piena  
Inserimento di L nella foglia (J K)

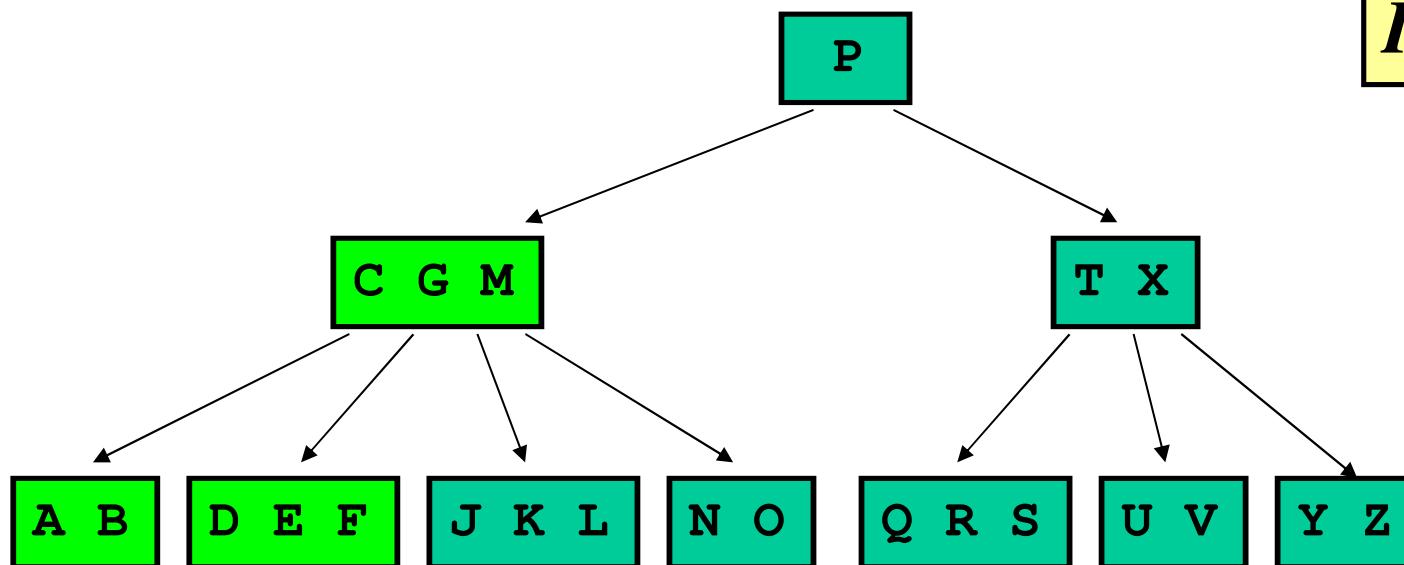


Split del figlio (A B C D E)

Inserimento di F nella metà a dx



*Insert(T,F)*





# Insert

Il numero di *DiskRead* e *DiskWrite* è  $O(h)$  perché sono eseguite  $O(1)$  operazioni di accesso al disco tra due chiamate consecutive di *InsertNonfull*

Il tempo  $T$  di CPU di è  $O(t h)$  perché *InsertNonfull* è ricorsiva in coda, in alternativa può essere iterativa con un ciclo while



## Delete

La procedura di rimozione di una chiave da un  $B$ -albero è:

***Delete*( $T, k$ )**

**if**  $root[T] \neq \text{nil}$  **then**

***DeleteNonmin*( $root[T], k$ )**

**if**  $n[root[T]] = 0$  **then**

$root[T] \leftarrow c_1[root[T]]$  ▷ **unico figlio**

essa si limita a richiamare la funzione ausiliaria ***DeleteNonmin*** sulla radice dell'albero dopo essersi assicurata che l'albero non sia vuoto.

Se al ritorno la radice non contiene alcuna chiave essa viene rimossa.



La procedura *DeleteNonmin* usa la procedura *AugmentChild* per assicurarsi di scendere sempre su di un nodo che non contiene il minimo numero di chiavi.

La procedura *AugmentChild* aumenta il numero di chiavi del figlio *i*-esimo prendendone una da uno dei fratelli adiacenti. Se i fratelli adiacenti hanno tutti il minimo numero di chiavi allora riunisce (fusione) il figlio *i*-esimo con uno dei fratelli adiacenti.

Tempi di Delete come Insert.



UNIVERSITÀ  
DEGLI STUDI  
DI BERGAMO

Dipartimento  
di Ingegneria Gestionale,  
dell'Informazione e della Produzione



# Esercizi su alberi binari di ricerca

PROGETTAZIONE, ALGORITMI E  
COMPUTABILITÀ  
(38090-MOD1)

Corso di laurea  
Magistrale in  
Ingegneria  
Informatica

RELATORE  
Prof.ssa Patrizia  
Scandurra

SEDE  
DIGIP

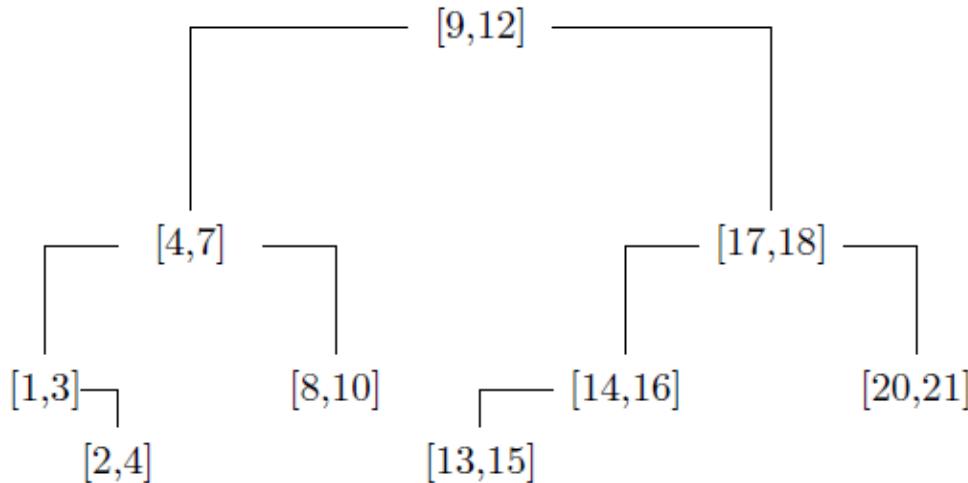
# Esercizi

Procurarsi la cartella src\_BST di codice fornita a lezione, e svolgere:

1. Arricchire la classe **AlberoBR** con i metodi per la ricerca del *predecessore* e del *successore* di un nodo (vedi pseudocodice fornito a lezione).
2. Arricchire la classe **AlberoBR** con un metodo che, dati due insiemi numerici rappresentati mediante alberi binari di ricerca (senza ripetizioni), crei l'albero rappresentante l'*intersezione dei due insiemi*. Qual è la complessità dell'algoritmo?
3. Si definisce **Interval Tree** un *albero binario di ricerca* in cui:
  - gli elementi sono intervalli  $[a, b]$ , con  $a$  e  $b$  sulla retta reale;
  - la chiave di un elemento  $[a, b]$  è l'estremo sinistro  $a$  dell'intervalllo;
  - non possono esistere intervalli annidati  $[a, b]$  e  $[c, d]$  con  $c <= a$  e  $b <= d$ .

(continua nella slide successiva)

- Ad esempio, l’Interval Tree costruito a partire dall’albero vuoto inserendo in ordine  $[9, 12]$ ,  $[4, 7]$ ,  $[17, 18]$ ,  $[1, 3]$ ,  $[8, 10]$ ,  $[14, 16]$ ,  $[20, 21]$ ,  $[2, 4]$ ,  $[13, 15]$  è:



- Definire lo pseudocodice di un algoritmo efficiente che, dato un intervallo  $[a,b]$  e un Interval Tree T, determina se  $[a, b]$  può essere inserito in T, ovvero se non esiste in T un intervallo che contiene interamente  $[a, b]$  o che è contenuto interamente in  $[a, b]$ .
- Qual è la complessità dell’algoritmo?

# Algoritmi e Strutture Dati

## Capitolo 7 Tabelle hash

Camil Demetrescu, Irene Finocchi,  
Giuseppe F. Italiano

# Dizionario: implementazioni

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**  $O(1)$

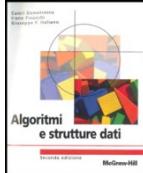
# Tabelle ad accesso diretto

Dizionari basati sulla rappresentazione indicizzata mediante array

Idea:

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

| Array V |            |
|---------|------------|
| Key     | Value      |
| 1       | New York   |
| 2       | Boston     |
| 3       | Mexico     |
| 4       | Kansas     |
| 5       | Detroit    |
| 6       | California |



# 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

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

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

$$n=100 \quad m=10^6 \quad \alpha = 0,0001 = 0,01\%$$

Grande spreco di memoria, se  $m \gg n$



# Pregi e difetti delle Tabelle ad accesso diretto

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!

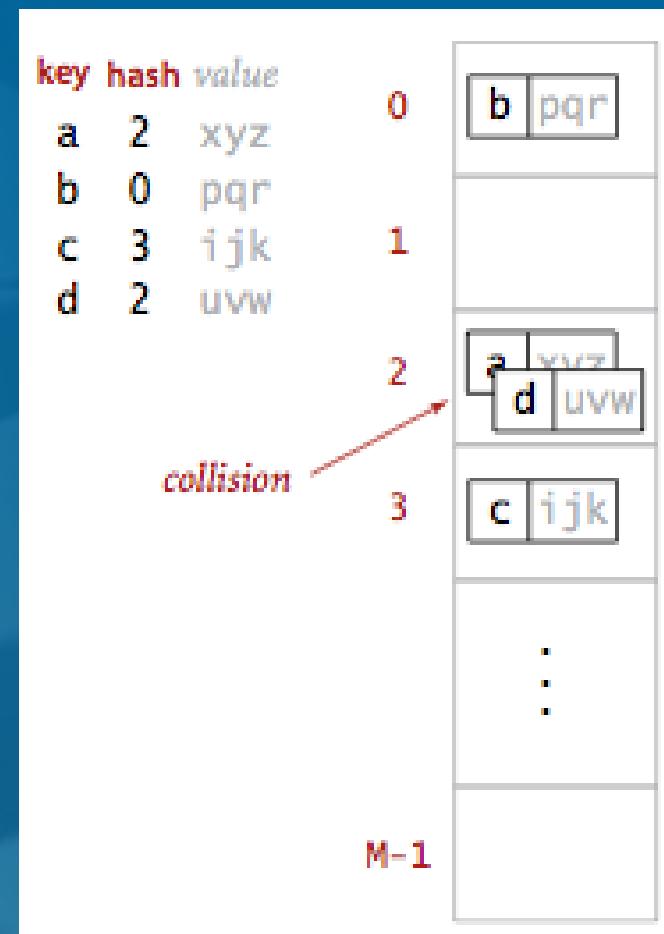
# Tabelle hash

Per ovviare agli inconvenienti delle tabelle 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:**  
 $h: U \rightarrow [0, m-1]$   
Elemento con chiave  $k$  in posizione  $v[h(k)]$

Le tabelle hash presentano il fenomeno delle **collisioni**





# 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  $v$ , con  $u \neq v$ , tale che  $h(u)=h(v)$ 
  - il nuovo elemento andrebbe a sovrascrivere il vecchio!

# Funzioni hash perfette

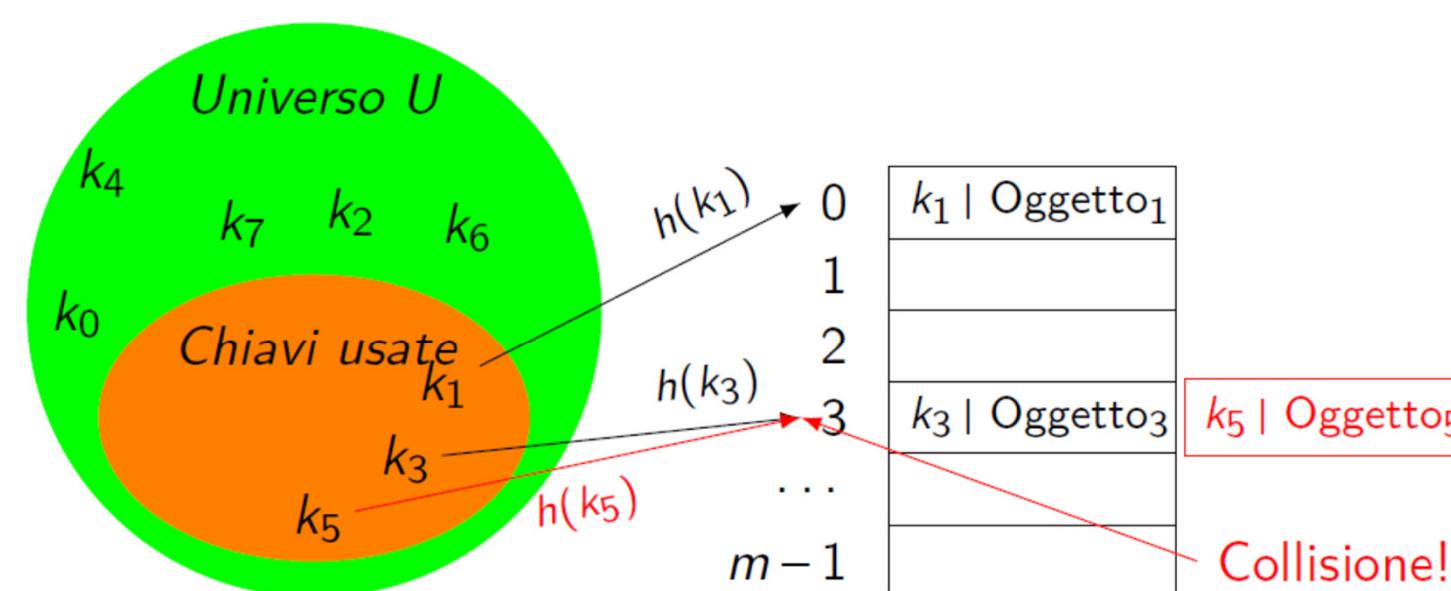
Un modo per evitare il fenomeno delle collisioni è usare **funzioni hash perfette**.

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

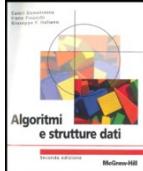
$$u \neq v \Rightarrow h(u) \neq h(v)$$

Deve essere  $|U| \leq m$

# Funzioni hash perfette



- $m \ll |U|$
- Funzione hash  $h: U \rightarrow \{0, \dots, m-1\}$ 
  - disperdere in modo uniforme le chiavi;
  - suriettiva;  $\Leftarrow$  Possibili collisioni!
  - se è biunivoca quando si considerano solo le chiavi usate, allora è una funzione hash perfetta.



# 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 però 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 per  $m$  primo):  $h(k) = \text{ascii}(k) \bmod m$

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

$$h('C') = 67 \bmod 11 = h('N') = 78 \bmod 11 = 1$$

$\Rightarrow$  se volessimo inserire sia ‘C’ and ‘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  $\{0, \dots, m-1\}$**

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  $h$  gode **dell'uniformità semplice** se per ogni indice  $i$  scelto a caso in  $\{0, \dots, m-1\}$ :

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



# Esempio

Se  $U$  è l'insieme dei numeri reali in  $[0,1]$  e ogni chiave ha la stessa probabilità di essere scelta, allora si può dimostrare che la funzione hash:

$$h(k) = \lfloor km \rfloor$$

soddisfa la proprietà di uniformità semplice

# Altri esempi: funzioni crittografiche di hash

- Mappano dati di lunghezza arbitraria (**messaggio**) in una stringa binaria di dimensione fissa chiamata valore di hash (o **message digest**)
- Progettate per essere unidirezionale (**one-way**), ovvero una funzione difficile da invertire
- Esempi:
  - MD5 (128 bit come lunghezza del digest)
  - SHA-1 (128 bit come lunghezza del digest)  
(Ormai superati)!
  - SHA-2 (256 bit come lunghezza del digest)

# Sicurezza di funzioni hash crittografiche

- *Resistenza alla collisione*
  - Dato un valore di hash  $x$ , deve essere difficile risalire ad un messaggio  $m$  con  $h(m) = x$
  - Deriva dal concetto di unidirezionalità
- *Resistenza alla seconda preimmagine*
  - Dato un input  $m_1$ , deve essere difficile trovare un secondo input  $m_2$  tale che  $h(m_1) = h(m_2)$
  - La resistenza alla collisione implica una resistenza alla seconda preimmagine

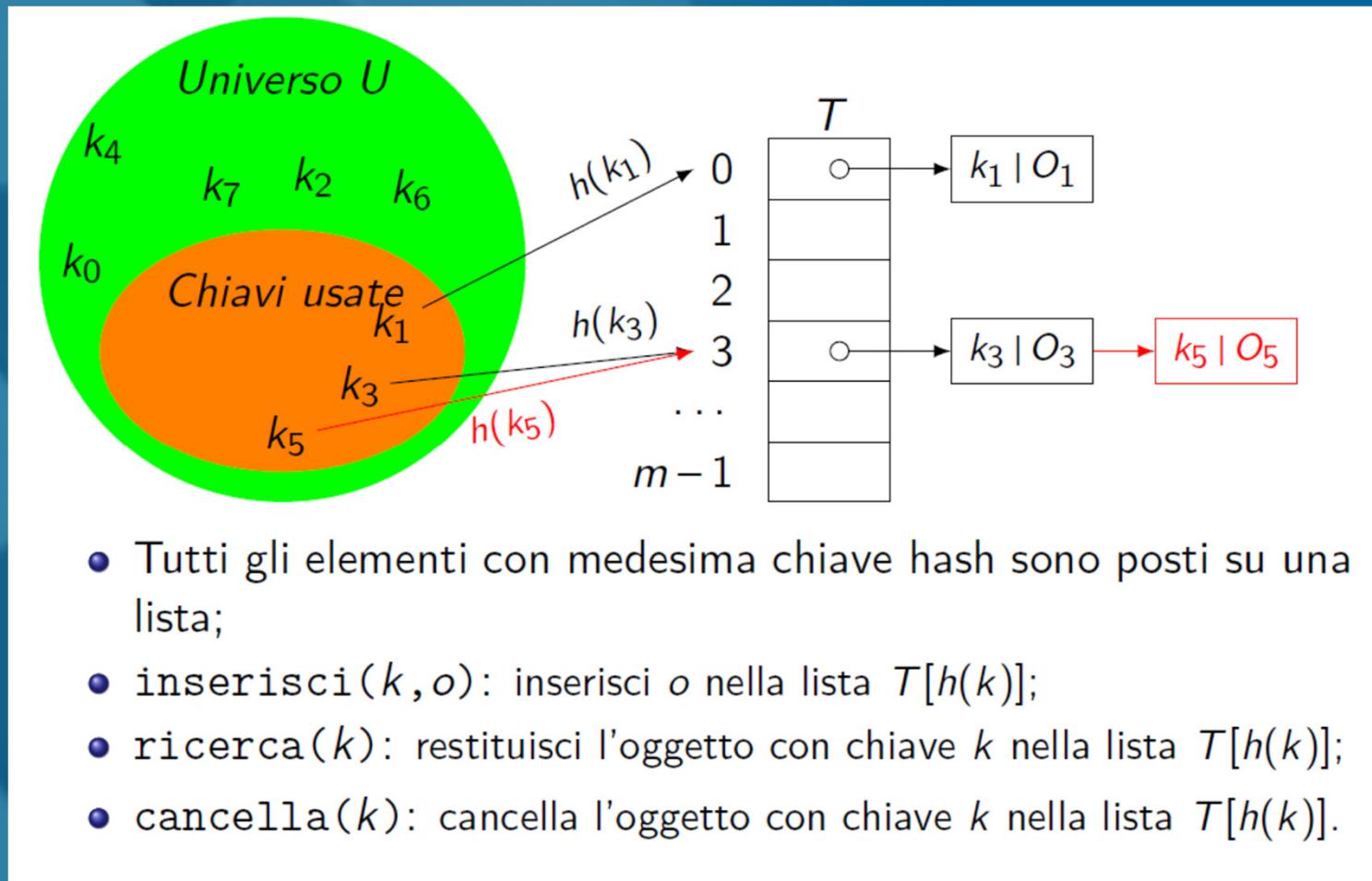
# Risoluzione delle collisioni

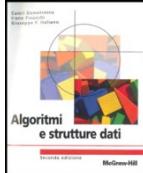
Nel caso in cui non si possano evitare le collisioni, dobbiamo trovare un modo per risolverle.

Due metodi classici sono i seguenti:

1. **Liste di collisione (chaining).** 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 (chaining)





# 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  $O(1)$ .

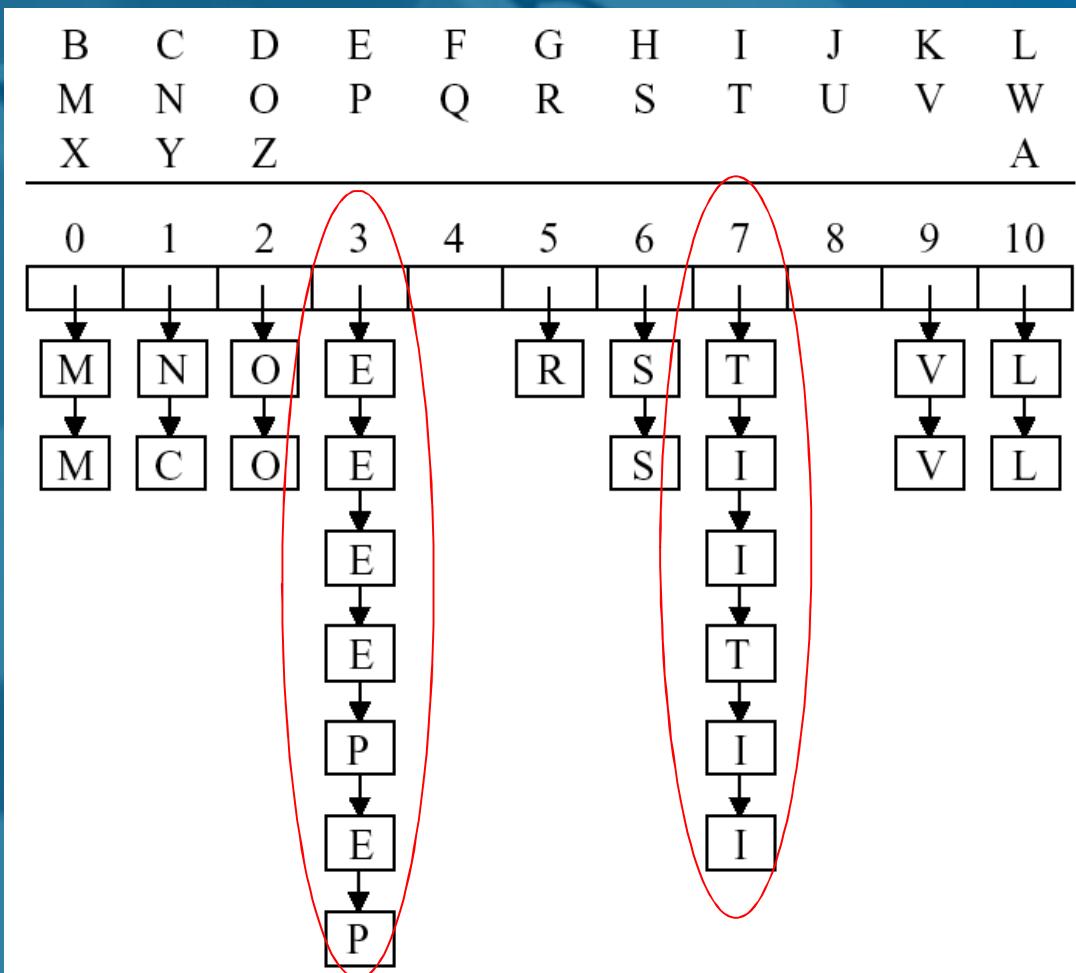
**operazioni:**

**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.

# Liste di collisione



$$h(k) = \text{ascii}(k) \bmod 11$$
$$\alpha = 26/11 = 2,36\dots$$

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

**PRECIPITEVOLIS  
SIMEVOLMENTE**



# Indirizzamento aperto

- Supponiamo di voler inserire un elemento con chiave  $k$  e la sua posizione “naturale”  $h(k)$  sia già occupata (collisione)
- Idea dell’indirizzamento aperto: in caso di collisione, si occupa un’altra cella della stessa tabella
- La selezione della nuova cella avviene tramite una sequenza pre-stabilita di indici (**sequenza di scansione degli indici**)

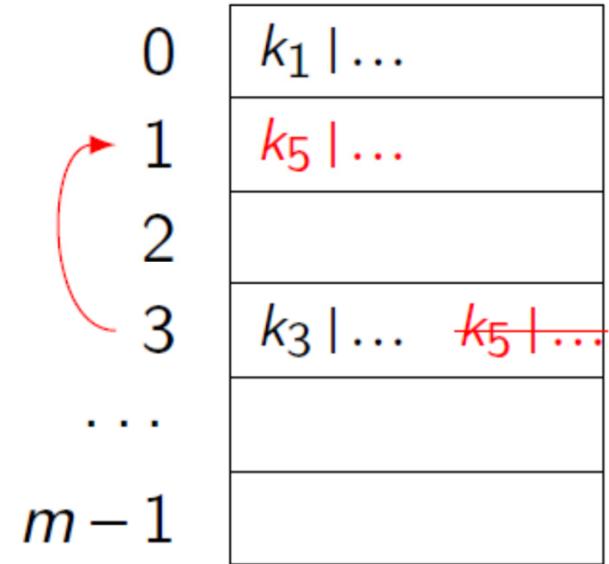
$c(k,0), c(k,1), c(k,2), \dots c(k,m-1)$

dove tipicamente  $c(k,0)=h(k)$

# Indirizzamento aperto: esempio

$$h(k_1) = 0$$

$$h(k_3) = h(k_5) = 3$$



$$c(k_5, 0) = 3, c(k_5, 1) = 3, \dots, c(k_5, m-1) = m-1$$

# Implementazione 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



# Sequenze di scansione

- Il metodo di selezione delle celle di scansione determina il tipo di reindirizzamento:
  - Lineare
  - Hashing doppio

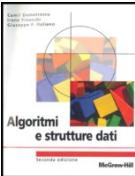


# Metodi di scansione: scansione lineare

Scansione lineare:

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

per  $0 \leq i < m$



$h(k) = \text{ascii}(k) \bmod m$  con  $m=31$

# Esempio

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

# PRECIPITE VOLISSI ME VOLMENTE

- 124 scansioni totali  
(celle in grigio!)
  - $124/26=4,8$  celle  
scandite in media per  
ogni inserimento

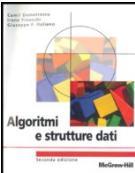
# Metodi di scansione: hashing doppio

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 e  $m$  e  $h_2(k)$  primi fra loro (MCD = 1)



|   | 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 |   |   |   |   |   |   |   |   |   |    |    |    |    |    |    |    |    |    |    | P  | R  |    |    |    |    |    |    |    |    |    |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
| E |   |   |   |   |   |   |   |   |   |    |    |    |    |    |    |    |    |    |    | P  | R  |    |    |    |    |    |    |    |    |    |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
| C |   |   |   |   |   |   |   |   |   |    |    |    |    |    |    |    |    |    |    | P  | R  |    |    |    |    |    |    |    |    |    |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
| I |   |   |   |   |   |   |   |   |   |    |    |    |    |    |    |    |    |    |    | P  | R  |    |    |    |    |    |    |    |    |    |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
| P |   |   |   |   |   |   |   |   |   |    |    |    |    |    |    |    |    |    |    | P  | R  |    |    |    |    |    |    |    |    |    |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
| I |   |   |   |   |   |   |   |   |   |    |    |    |    |    |    |    |    |    |    | P  | R  | I  |    |    |    |    |    |    |    |    |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
| T |   |   |   |   |   |   |   |   |   |    |    |    |    |    |    |    |    |    |    | P  | R  | T  | I  |    |    |    |    |    |    |    |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
| E |   |   |   |   |   |   |   |   |   |    |    |    |    |    |    |    |    |    |    | E  | P  | R  | T  | I  |    |    |    |    |    |    |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
| V |   |   |   |   |   |   |   |   |   |    |    |    |    |    |    |    |    |    |    | E  | P  | R  | T  | I  | V  |    |    |    |    |    |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
| O |   |   |   |   |   |   |   |   |   |    |    |    |    |    |    |    |    |    |    | O  | P  | R  | T  | I  | V  |    |    |    |    |    |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
| L |   |   |   |   |   |   |   |   |   |    |    |    |    |    |    |    |    |    |    | L  | E  | O  | P  | R  | T  | I  | V  |    |    |    |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
| I |   |   |   |   |   |   |   |   |   |    |    |    |    |    |    |    |    |    |    | I  | L  | E  | O  | P  | R  | T  | I  | V  |    |    |   |   |   |   |   |   |   |   |   |   |   |   |   |   |
| S |   |   |   |   |   |   |   |   |   |    |    |    |    |    |    |    |    |    |    | I  | C  | P  | E  | I  | L  | E  | O  | P  | R  | S  | T | I | V |   |   |   |   |   |   |   |   |   |   |   |
| S |   |   |   |   |   |   |   |   |   |    |    |    |    |    |    |    |    |    |    | I  | C  | P  | E  | I  | S  | L  | E  | O  | P  | R  | S | T | I | V |   |   |   |   |   |   |   |   |   |   |
| I |   |   |   |   |   |   |   |   |   |    |    |    |    |    |    |    |    |    |    | I  | C  | P  | E  | I  | S  | L  | E  | I  | O  | P  | R | S | T | I | V |   |   |   |   |   |   |   |   |   |
| M |   |   |   |   |   |   |   |   |   |    |    |    |    |    |    |    |    |    |    | M  | I  | C  | P  | E  | I  | S  | L  | E  | I  | O  | P | R | S | T | I | V |   |   |   |   |   |   |   |   |
| E |   |   |   |   |   |   |   |   |   |    |    |    |    |    |    |    |    |    |    | M  | I  | C  | P  | E  | E  | I  | S  | L  | E  | I  | O | P | R | S | T | I | V |   |   |   |   |   |   |   |
| V |   |   |   |   |   |   |   |   |   |    |    |    |    |    |    |    |    |    |    | V  | M  | I  | C  | P  | E  | E  | I  | S  | L  | E  | I | O | P | R | S | T | I | V |   |   |   |   |   |   |
| O |   |   |   |   |   |   |   |   |   |    |    |    |    |    |    |    |    |    |    | O  | M  | I  | C  | P  | E  | E  | O  | I  | S  | L  | E | I | O | P | R | S | T | I | V |   |   |   |   |   |
| L |   |   |   |   |   |   |   |   |   |    |    |    |    |    |    |    |    |    |    | L  | M  | I  | C  | P  | E  | E  | O  | I  | S  | L  | E | I | O | P | R | S | T | I | V |   |   |   |   |   |
| M |   |   |   |   |   |   |   |   |   |    |    |    |    |    |    |    |    |    |    | M  | M  | I  | C  | P  | E  | E  | O  | I  | S  | L  | E | I | O | P | R | S | T | I | V |   |   |   |   |   |
| E |   |   |   |   |   |   |   |   |   |    |    |    |    |    |    |    |    |    |    | E  | M  | M  | I  | C  | P  | E  | E  | O  | I  | S  | L | E | I | O | P | R | S | T | I | V |   |   |   |   |
| N |   |   |   |   |   |   |   |   |   |    |    |    |    |    |    |    |    |    |    | N  | M  | M  | E  | I  | C  | P  | E  | E  | O  | I  | S | L | E | I | O | P | N | R | S | T | I | V |   |   |
| T |   |   |   |   |   |   |   |   |   |    |    |    |    |    |    |    |    |    |    | T  | M  | M  | E  | I  | C  | P  | E  | E  | O  | I  | S | T | L | E | I | O | P | N | R | S | T | I | V |   |
| E |   |   |   |   |   |   |   |   |   |    |    |    |    |    |    |    |    |    |    | E  | M  | M  | E  | I  | C  | P  | E  | E  | O  | E  | I | S | T | L | E | I | O | P | N | R | S | T | I | V |

# Esempio

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

# PRECIPITEVOLIS SIMEVOLMENTE

82 scansioni,  $82/26=3,1$  celle scandite in media per inserimento

# Analisi del costo di scansione per la ricerca di una chiave

Assumendo che le chiavi siano prese con probabilità uniforme da U:

Tempo richiesto nel caso peggiore:  $O(n)$

Tempo richiesto nel caso medio:

| <i>esito ricerca</i> | <i>sc. lineare</i>                      | <i>hashing doppio</i>                  |
|----------------------|-----------------------------------------|----------------------------------------|
| chiave trovata       | $\frac{1}{2} + \frac{1}{2(1-\alpha)}$   | $-\frac{1}{\alpha} \log_e(1 - \alpha)$ |
| chiave non trovata   | $\frac{1}{2} + \frac{1}{2(1-\alpha)^2}$ | $\frac{1}{1-\alpha}$                   |

dove  $\alpha = n/m < 1$  (fattore di carico)



# 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 verificarsi le **collisioni**
- Tecniche classiche per risolvere le collisioni sono **liste di collisione** e **indirizzamento aperto**

# Algoritmi e Strutture Dati

## Capitolo 4 Ordinamento per “confronto”

Camil Demetrescu, Irene Finocchi,  
Giuseppe F. Italiano



# Ordinamento

Il problema dell'ordinamento:

*Input:* sequenza  $a_1, a_2, \dots, a_n$  di elementi su cui è definita una relazione d'ordine totale  $\leq$ .

*Output:*  $a'_1, a'_2, \dots, a'_n$  permutazione di  $a_1, a_2, \dots, a_n$  tale che  $a'_1 \leq a'_2 \leq \dots \leq a'_n$ .



# Complessità temporale

- Come misurarla?
  - Numero confronti
  - Numero operazioni (confronti + spostamenti)
  - Numero passi elementari (darà lo stesso risultato del precedente)

# Algoritmi e tempi tipici

- Numerosissimi algoritmi
- Tre tempi tipici:  $O(n^2)$ ,  $O(n \lg n)$ ,  $O(n)$  (non basati su confronto)
- $O(n^2)$  è l'upper bound del problema perché corrisponde ad eseguire tutti i confronti

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

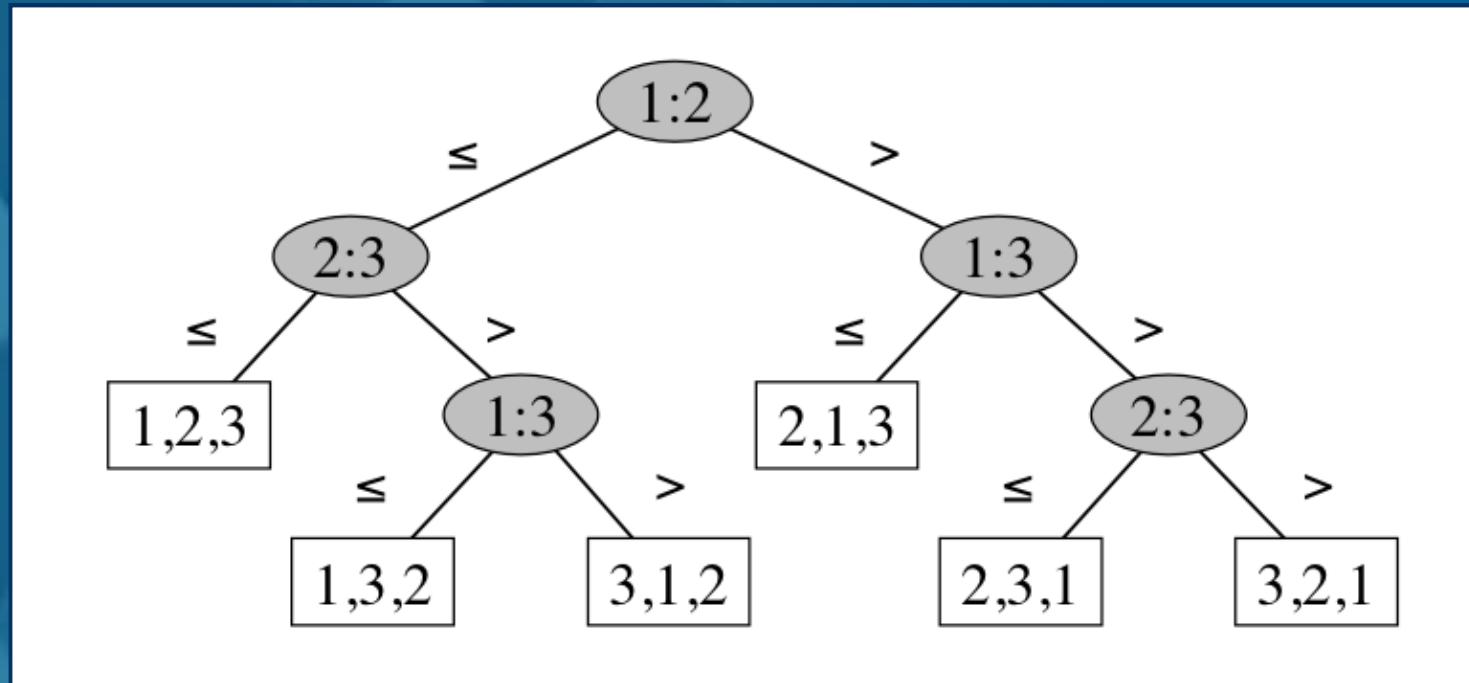


# Lower bound

- Delimitazione inferiore alla quantità di una certa risorsa di calcolo **necessaria** per risolvere un problema
- $\Omega(n \log n)$  è un **lower bound** al numero di confronti richiesti per ordinare  $n$  oggetti
- Consideriamo un generico algoritmo  $\mathcal{A}$ , che ordina eseguendo solo confronti: dimostreremo che  $\mathcal{A}$  esegue  $\Omega(n \log n)$  confronti

# Alberi di decisione

- Descrive le diverse sequenze di confronti che  $\mathcal{A}$  potrebbe fare su istanze di lunghezza n





# Proprietà

- Per una particolare istanza, i confronti eseguiti da  $\mathcal{A}$  su quella istanza rappresentano un cammino radice - foglia
- Il numero di confronti nel caso peggiore è pari all'altezza dell'albero di decisione
- Un albero di decisione per l'ordinamento di  $n$  elementi contiene almeno  $n!$  foglie  
(una per ogni possibile permutazione degli  $n$  oggetti)

# Altezza in funzione delle foglie

- Un albero binario con  $k$  foglie ha altezza almeno  $\log_2 k$ 
  - Un albero binario completo ha  $2^h$  foglie. Quindi, un albero binario ha al più  $2^h$  foglie:  $k \leq 2^h$  da cui:  $h \geq \log_2 k$  cioè  $h = \Omega(\lg k)$
- In alternativa: dimostrazione per induzione su  $k$ 
  - Passo base:  $0 \geq \log_2 1$
  - $h(k) \geq 1 + h(k/2)$  poiché uno dei due sottoalberi ha almeno metà delle foglie
  - $h(k/2) \geq \log_2 (k/2)$  per ipotesi induttiva
  - $h(k) \geq 1 + \log_2 (k/2) = \log_2 k$

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

**Teorema.**  $\Omega(n \log n)$  è un limite inferiore per la complessità del problema dell’ordinamento basato su “confronti”

- *Dimostrazione.*
  - Siccome le permutazioni di  $1, 2, \dots, n$  sono  $n!$  l’albero di decisione deve avere almeno  $k=n!$  foglie
  - L’altezza dell’albero di decisione è almeno  $\lg k$  e cioè  $h \geq \lg(n!)$  (vedi dimostrazione slide precedente)
  - Dunque nel caso peggiore il generico algoritmo  $\mathcal{A}$  deve eseguire almeno  $\lg(n!)$  confronti

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

*Dimostrazione (cont.).*

– Formula di Stirling:  $n! \sim (2\pi n)^{1/2} \cdot (n/e)^n$

da cui:

$$\begin{aligned} h &\geq \lg (n!) > \lg (n/e)^n \\ &= n \lg n - n \lg e \\ h &= \Omega(n \log n) \end{aligned}$$

e quindi

$$T_{worst}^A(n) = \Omega(n \log n)$$

per ogni algoritmo generale  $\mathcal{A}$  di ordinamento.

Possiamo concludere che  $\Omega(n \log n)$  è un limite inferiore per la complessità del problema dell'ordinamento.



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

- Altro modo (senza formula di Stirling):

$$\begin{aligned}\log_2(n!) &\geq \log_2(n/2)^{n/2} \\ &= (n/2) \log_2(n/2) \\ &= \Omega(n \log n)\end{aligned}$$

L’algoritmo di ordinamento *MergeSort* risolve il problema dell’ordinamento con complessità

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

Dunque  $O(n \log n)$  è anche un limite superiore per la complessità del problema di ordinamento.

Siccome limite superiore e inferiore coincidono  $\Theta(n \log n)$  è un limite stretto per il problema dell’ordinamento.



# Ordinamenti quadratici

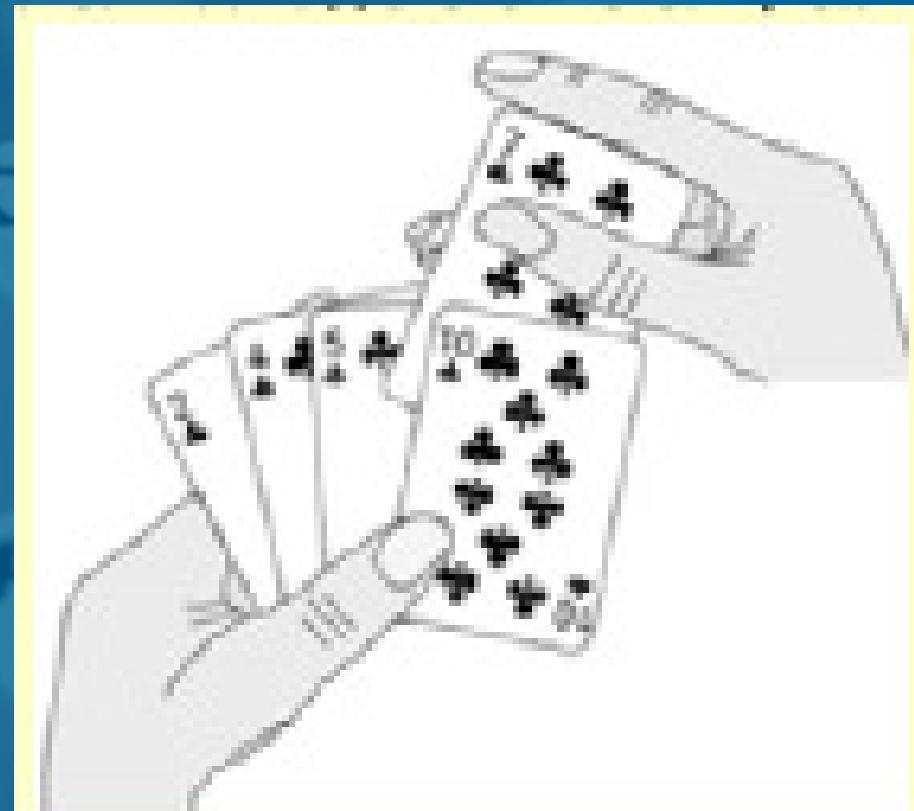
( $O(n^2)$  confronti nel caso peggiore)

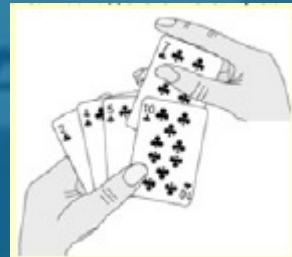
- InsertionSort
- SelectionSort
- BubbleSort

} Approccio incrementale:  
estendono l'ordinamento  
da  $k$  a  $k+1$  elementi,  
iterando  $n$  volte  
( $n$  numeri elementi da ordinare)

# InsertionSort

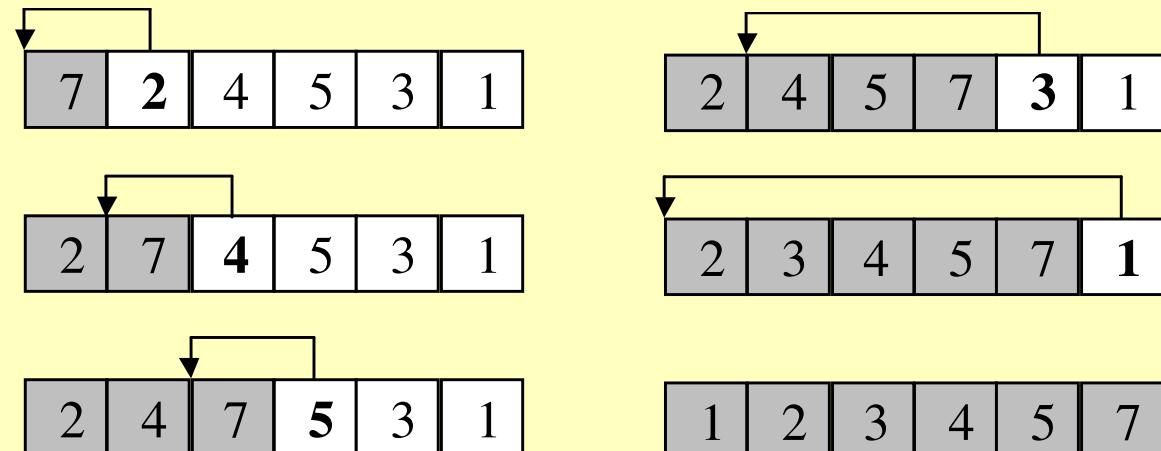
Approccio incrementale: estende l'ordinamento da  $j-1$  a  $j$  elementi, posizionando l'elemento  $j$ -esimo nella posizione corretta rispetto ai primi  $j-1$  elementi





# InsertionSort

**Approccio incrementale:** estende l'ordinamento da  $j-1$  a  $j$  elementi, posizionando l'elemento  $j$ -esimo nella posizione corretta rispetto ai primi  $j-1$  elementi



## InsertionSort (A)

```
1. for j=2 to n do
2. x = A[j] //elemento da inserire nella sotto-sequenza ordinata A[1..j-1]
3. i= j-1 //indice di scansione da destra verso sinistra (da j-1 ad 1)
4. while i>0 and A[i] > x do
5. A[i+1]= A[i] //sposta di una posizione tutti gli elementi A[i] > x
6. i= i-1
7. A[i+1]=x //Inserisce x nella locazione che gli compete
```

- $n = A.length$  e si assume che  $A$  sia numerato da 1 ad  $n$ :  $A[1..n]$
- Al generico passo  $j$ ,  $A[1..j-1]$  già ordinata
- Elemento  $x = A[j]$  inserito nella posizione che gli compete
- Righe 3-4-5-6: individuano la posizione ( $i+1$ ) in cui va messo  $x$ , facendo nel contempo spazio per  $x$  slittando gli elementi maggiori di  $x$  a destra



# InsertionSort: correttezza

- Si dimostra facendo vedere che dopo il generico passo  $j$ , i primi  $j$  elementi sono ordinati
- Induzione su  $j$ :
  - $j=1$ : banale. All'inizio il primo elemento è ordinato
  - $j>1$ . All'inizio del passo  $j$  i primi  $j-1$  elementi sono ordinati (ipotesi induttiva). Allora la tesi segue banalmente dalla struttura dell'algoritmo.

# InsertionSort: analisi del caso peggiore

Array ordinato in ordine decrescente     $T(n) = \Theta(n^2)$

Bisogna confrontare ogni elemento  $A[j]$  per  $j=2..n$  con ogni elemento del sotto-array  $A[1..j-1]$ , ovvero le linee 4-6 richiedono  $j-1$  confronti nel caso peggiore

Il tempo di esecuzione (confronti) è nel caso peggiore:

$$T_{\text{worst}}(n) = \sum_{j=2}^n (j-1) = \Theta(n^2)$$

Num. spostamenti per ogni  $j=2..n$ : 2 (per caricare/scaricare  $x$ ) +  $j-1 = \Theta(n^2)$

# InsertionSort: analisi del caso migliore

Array ordinato in ordine crescente

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

Per ogni elemento  $A[j]$  per  $j=2..n$  si trova che  $A[i] \leq x$  quando  $i=j-1$  (cioè al primo confronto), ovvero le linee 4-6 richiedono 1 solo confronto, per un totale di  $n-1$

In totale, il tempo di esecuzione è nel caso migliore:

$$T_{\text{best}}(n) = \sum_{j=2}^n 1 = n-1 = \Theta(n)$$

Num. spostamenti per ogni  $j=2..n$ : 2 (per caricare/scaricare  $x$ ) =  $\Theta(n)$



# InsertionSort: analisi del caso medio

Array in ordine sparso

$$T_{avg}(n) = \Theta(n^2)$$

Per ogni elemento  $A[j]$  per  $j=2..n$ , mediamente metà elementi in  $A[1..j-1]$  sono più piccoli di  $A[j]$  e quindi si effettuano  $(j-1)/2$  confronti

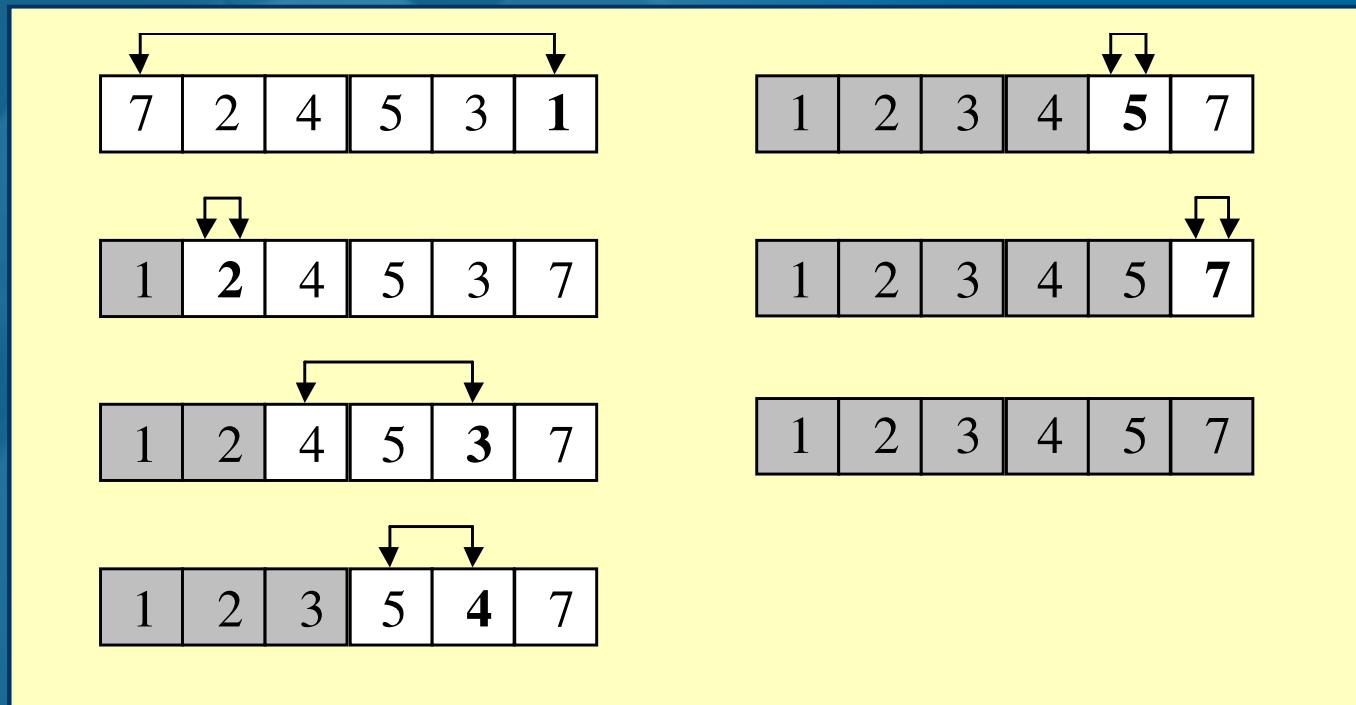
In totale, il tempo di esecuzione è nel caso medio:

$$T_{avg}(n) = \sum_{j=2}^n (j-1)/2 = \Theta(n^2)$$

Num. spostamenti per ogni  $j=2..n$ : 2 (per caricare/scaricare  $x$ ) +  $(j-1)/2 = \Theta(n^2)$

# SelectionSort

**Approccio incrementale:** estende l'ordinamento da k a k+1 elementi, scegliendo il **minimo** degli n-k elementi **non ancora ordinati** (caselle bianche) e mettendolo in posizione k+1





# SelectionSort

## SelectionSort (A)

1. **for**  $k=1$  **to**  $n-1$  **do**
2.      $m = k$  //indice del minimo elemento (inizializzato a  $k$ )
3.     **for**  $j=k+1$  **to**  $n$  **do** //ricerca del minimo in  $A[k..n]$
4.         **if**  $A[j] < A[m]$  **then**  $m = j$
5.         *scambia*  $A[m]$  con  $A[k]$  se  $m$  è diverso da  $k$

- $n = A.length$  e  $A[1..n]$
- Al generico passo  $k$ , si cerca il minimo (righe 2-4) in  $A[k..n]$  non ancora ordinato e lo si pone alla posizione  $k$

Minimizza il numero di spostamenti

# SelectionSort: correttezza

- Si dimostra facendo vedere che dopo il generico passo  $k$  ( $k=1,\dots,n-1$ ) si ha: (i) i primi  $k$  elementi sono ordinati e (ii) contengono i  $k$  elementi più piccoli dell'array
- Induzione su  $k$ :
  - $k=1$ : banale. All'inizio il primo elemento è il minimo. (i) e (ii) banalmente verificate.
  - $k>1$ . All'inizio del passo  $k$  i primi  $k-1$  elementi sono ordinati e sono i  $k-1$  elementi più piccoli nell'array (ipotesi induttiva). Allora la tesi segue dal fatto che l'algoritmo seleziona il minimo dai restanti  $n-k+1$  elementi e lo mette in posizione  $k$ . Infatti:
    - (ii) i  $k$  elementi restano i minimi nell'array
    - (i) l'elemento in posizione  $k$  non è mai più piccolo dei primi  $k-1$

# SelectionSort: analisi (in tutti i casi)

La k-esima estrazione di minimo richiede n-k confronti

Il ciclo più esterno è eseguito n-1 volte

In totale, il tempo di esecuzione è pertanto:

$$\sum_{k=1}^{n-1} (n-k) = \sum_{i=1}^{n-1} i = \Theta(n^2)$$

con il cambiamento di variabile  $i = n-k$  e la serie aritmetica.

# SelectionSort: analisi (spostamenti)

CASO PEGGIORE: l'elemento max è sempre nella prima posizione, ed i rimanenti sono in ordine

- Se in ogni k-esima iterazione (per  $k=1..n-1$ ) il max è sempre nella k-esima posizione, allora occorrono  $n-1$  scambi (sostamenti) e quindi il numero di sostamenti è lineare:  $\Theta(n)$

CASO MIGLIORE: array ordinato, nessun sostamento:

$\Theta(1)$

CASO MEDIO: lineare  $\Theta(n)$  [dimostrabile]

# 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 → *correttezza* per induzione (vedi Lemma 4.3 del libro Demetrescu et al.)

# 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 |
|---|---|---|---|---|---|

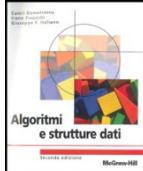
2 3  
1 3

(5) 

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

1 2  

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



# BubbleSort

## BubbleSort (A)

1. **for**  $i=1$  **to**  $n-1$  **do** //i conta le scansioni
2.     *scambiAvvenuti=false*
3.     **for**  $j=2$  **to**  $n-i+1$  **do** //Il max di  $A[1..n-i+1]$  è portato in  $n-i+1$
4.         **if**  $A[j-1] > A[j]$
5.             **then** *scambia A[j] con A[j-1]; scambiAvvenuti=true;*
6.         **if not** *scambiAvvenuti* **then break**

- $n = A.length$  e  $A[1..n]$
- Nella i-esima scansione confronta le prime ( $n-i$ ) coppie adiacenti
- Dopo una scansione in cui non viene effettuato nessuno scambio (not *scambiAvvenuti*) l'array è ordinato e l'algoritmo termina



# BubbleSort: analisi nel caso peggiore e medio

Array ordinato in ordine decrescente  $T_{\text{worst}}(n) = \Theta(n^2)$

Nella i-esima scansione vengono eseguiti  $n-i$  confronti

In totale, il tempo di esecuzione è pertanto:

$$\sum_{i=1}^{n-1} (n-i) = \sum_{j=1}^{n-1} j = \Theta(n^2)$$

usando il cambiamento di variabile  $j = n-i$  e la serie aritmetica

Similmente nel caso medio (array in ordine sparso):  $T_{\text{avg}}(n) = \Theta(n^2)$   
[dimostrabile]

Sia nel caso peggiore che nel caso medio, numero di scambi = numero di confronti =  $\Theta(n^2)$

# BubbleSort: analisi del caso migliore

Array ordinato in ordine crescente

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

Il ciclo più esterno viene eseguito una volta: infatti poiché il vettore è ordinato durante il ciclo interno non vi sono scambi e la variabile *scambiAvvenuti* alla fine è uguale a false. L'algoritmo termina con l'istruzione break e i=1.

Il numero di confronti è quindi uguale a  $n-i=n-1$ .

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

Spostamenti: numero di scambi = 0 =  $\Theta(1)$



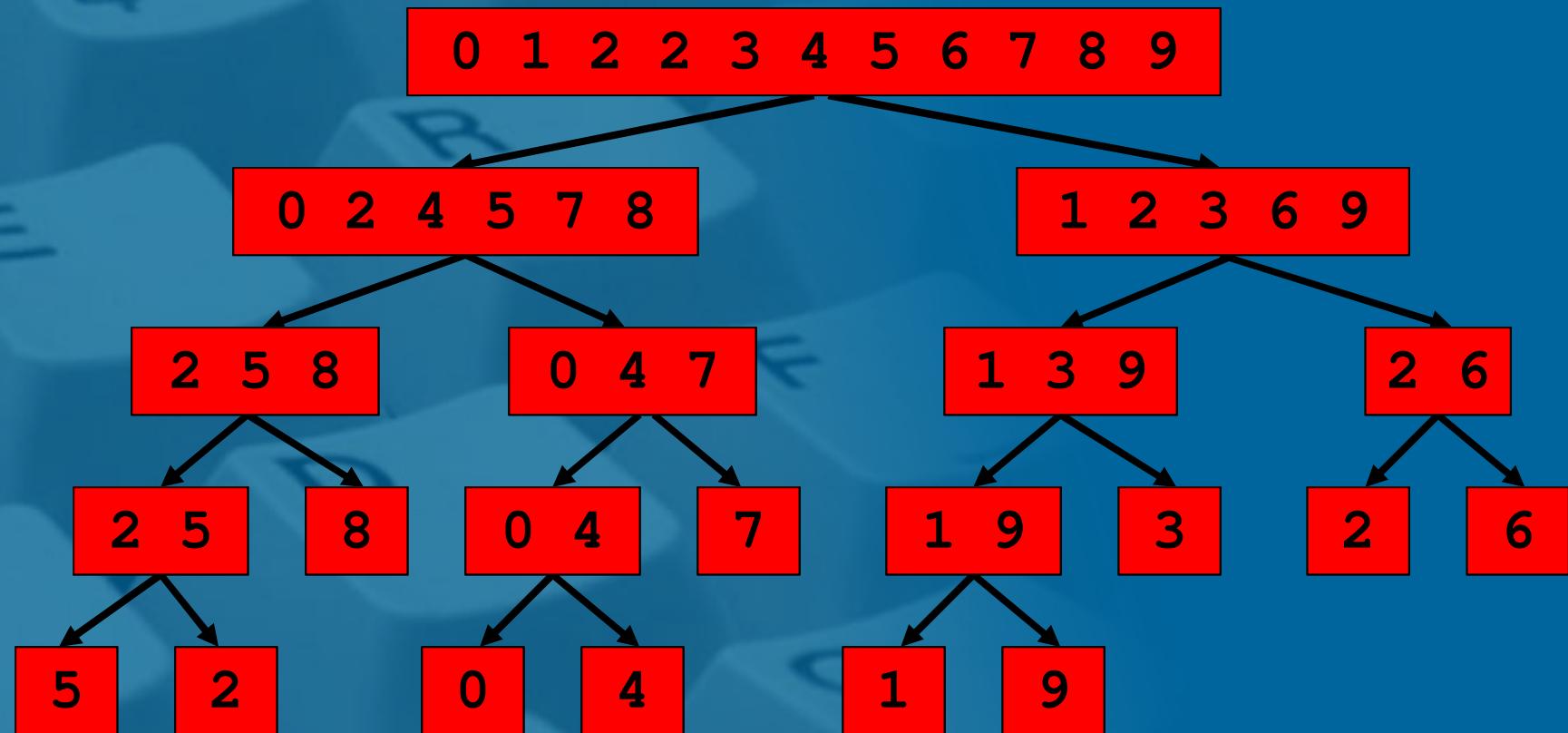
# Ordinamenti ottimi (nel modello basato su confronti)



# MergeSort

- Usa la tecnica del **divide et impera**:
  - 1 **Divide**: dividi l'array a metà
  - 2 Risolvi il sottoproblema ricorsivamente
  - 3 **Impera**: fondi le due sottosequenze ordinate

# Un esempio di esecuzione



L'albero delle chiamate ricorsive è sempre bilanciato!



# Algoritmo MergeSort

A array di dimensione n

MergeSort (A,p,r)

if  $p < r$  then

$q \leftarrow \lfloor (p+r)/2 \rfloor$

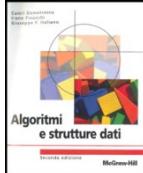
MergeSort (A,p,q) //  $\lceil n/2 \rceil$  elementi

MergeSort (A,q+1,r) //  $\lfloor n/2 \rfloor$  elementi

Merge(A,p,q,r)

# Procedura Merge

- Due array ordinati A e B possono essere fusi rapidamente:
  - estrai ripetutamente il minimo di A e B e copialo nell’array di output C, finché A oppure B non diventa vuoto
  - copia gli elementi rimasti nell’array non vuoto alla fine dell’array di output C
- Non opera in “loco” perché richiede un ulteriore array C di output
- Tempo esecuzione del Merge:  $\Theta(n)$ , dove n è il numero totale di elementi ( $n=r-p+1$ )
- Non opera in loco



# Algoritmo Merge

Merge (A,p,q,r)

$$n_1 \leftarrow q - p + 1$$

$$n_2 \leftarrow r - q$$

for  $i \leftarrow 1$  to  $n_1$  do

$$L[i] \leftarrow A[p + i - 1]$$

for  $j \leftarrow 1$  to  $n_2$  do

$$R[j] \leftarrow A[q + j]$$

$$L[n_1 + 1] \leftarrow R[n_2 + 1] \leftarrow \infty$$



```
i ← j ← 1
for k ← p to r do
 if L[i] ≤ R[j] then
 A[k] ← L[i]
 i ← i + 1
 else
 A[k] ← R[j]
 j ← j + 1
```

# Tempo di esecuzione del MergeSort

- Il numero di confronti del MergeSort è descritto dalla seguente relazione di ricorrenza per  $n > 1$  ( $T(1) = 1$ ):

$$T(n) = d(n) + 2*T(n/2) + c(n)$$

- $d(n) \rightarrow$  tempo necessario a dividere  $\rightarrow \Theta(1)$
- $c(n) \rightarrow$  tempo necessario per combinare 2 sequenze ordinate di  $n/2$  elementi (Merge())  $\rightarrow \Theta(n)$

- Si ha:  $T(n) = 2 * T(n/2) + f(n)$  con  
 $f(n) = d(n) + c(n) = \Theta(n)$
- Usando il Teorema Master (caso 2. con  $a=b=2$ ) si ottiene:

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

# QuickSort

- Usa la tecnica del **divide et impera**:
  - 1 **Divide**: scegli un elemento  $x$  della sequenza (*pivot o perno*) e partiziona la sequenza in elementi  $\leq x$  ed elementi  $>x$
  - 2 Risolvi i due sottoproblemi ricorsivamente
  - 3 **Impera**: restituisci la concatenazione delle due sottosequenze ordinate

# QuickSort (non in loco)

## QuickSort ( $A$ )

1. scegli un elemento  $x$  (il *pivot*) in  $A$
2. partiziona  $A$  rispetto a  $x$  calcolando:
  3.  $A_1 = \{y \in A : y \leq x\}$
  4.  $A_2 = \{y \in A : y > x\}$
5. **if** ( $|A_1| > 1$ ) **then** QuickSort( $A_1$ )
6. **if** ( $|A_2| > 1$ ) **then** QuickSort( $A_2$ )
7. copia la concatenazione di  $A_1$  e  $A_2$  in  $A$

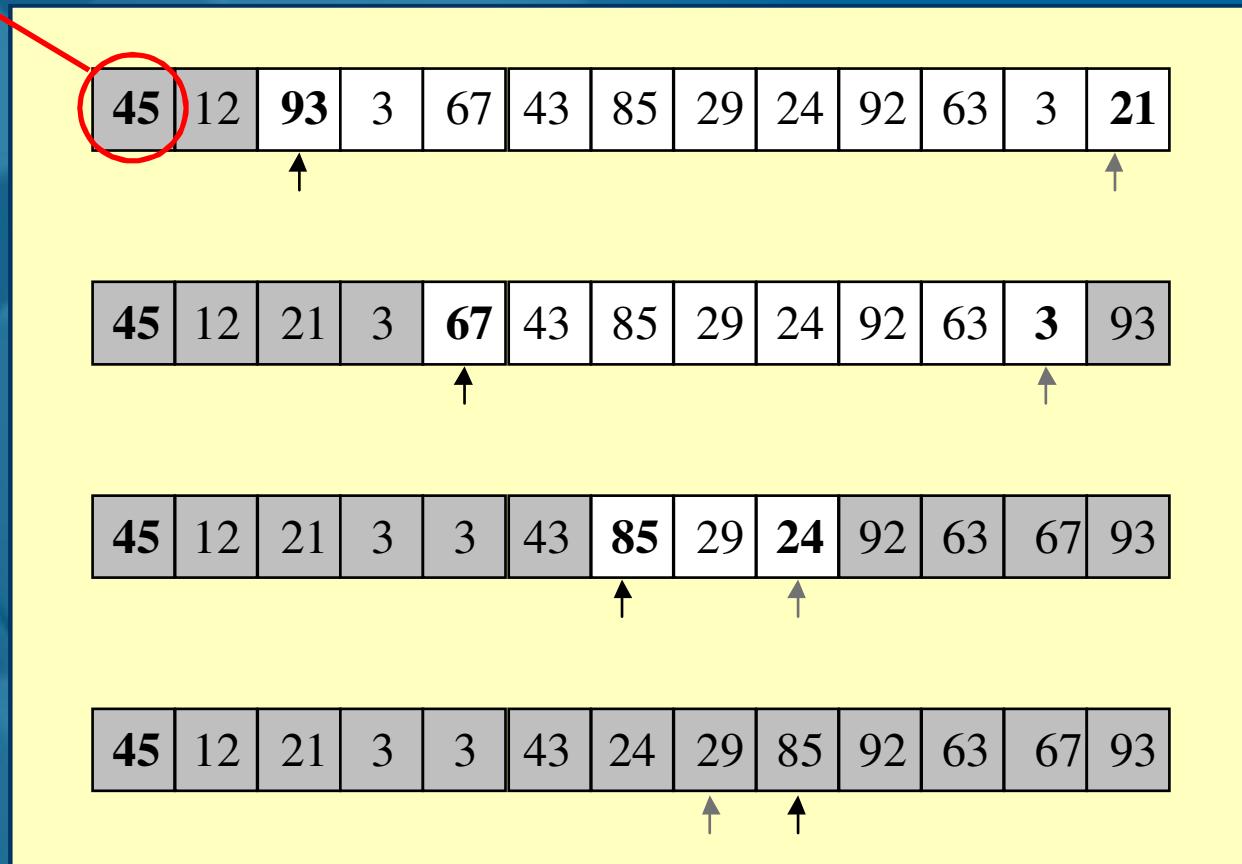
# Partizione in loco

- Scorri l'array “in parallelo” da sinistra verso destra e da destra verso sinistra
  - da sinistra verso destra, ci si ferma su un elemento maggiore del perno
  - da destra verso sinistra, ci si ferma su un elemento minore del perno
- Scambia gli elementi e riprendi la scansione fino a quando gli indici non si incrociano

Tempo di esecuzione (num. confronti):  $\Theta(n)$   
ogni elemento è confrontato con il pivot una sola volta, quindi si hanno  $n-1$  confronti

# Partizione in loco: un esempio

pivot





# QuickSort (in “loco”)

## Partition (A, i, f )

```
1. x=A[i] //Partiziona A[i..f] intorno al pivot A[i]
2. inf =i e sup= f + 1
3. while (true) do
4. do (inf=inf + 1) while (inf <= f and A[inf] ≤ x)
5. do (sup=sup-1) while (A[sup] > x)
6. if (inf < sup) then scambia A[inf] e A[sup]
7. else break
8. scambia A[i] e A[sup]
9. return sup
```

## Proprietà:

In ogni istante, gli elementi  $A[i], \dots, A[sup-1]$  sono  $\leq$  del perno,  
mentre gli elementi  $A[sup+1], \dots, A[f]$  sono  $>$  del perno

Tempo di esecuzione:  
 $\Theta(n)$

mette il pivot  
“al centro”  
restituisce  
l’indice del pivot

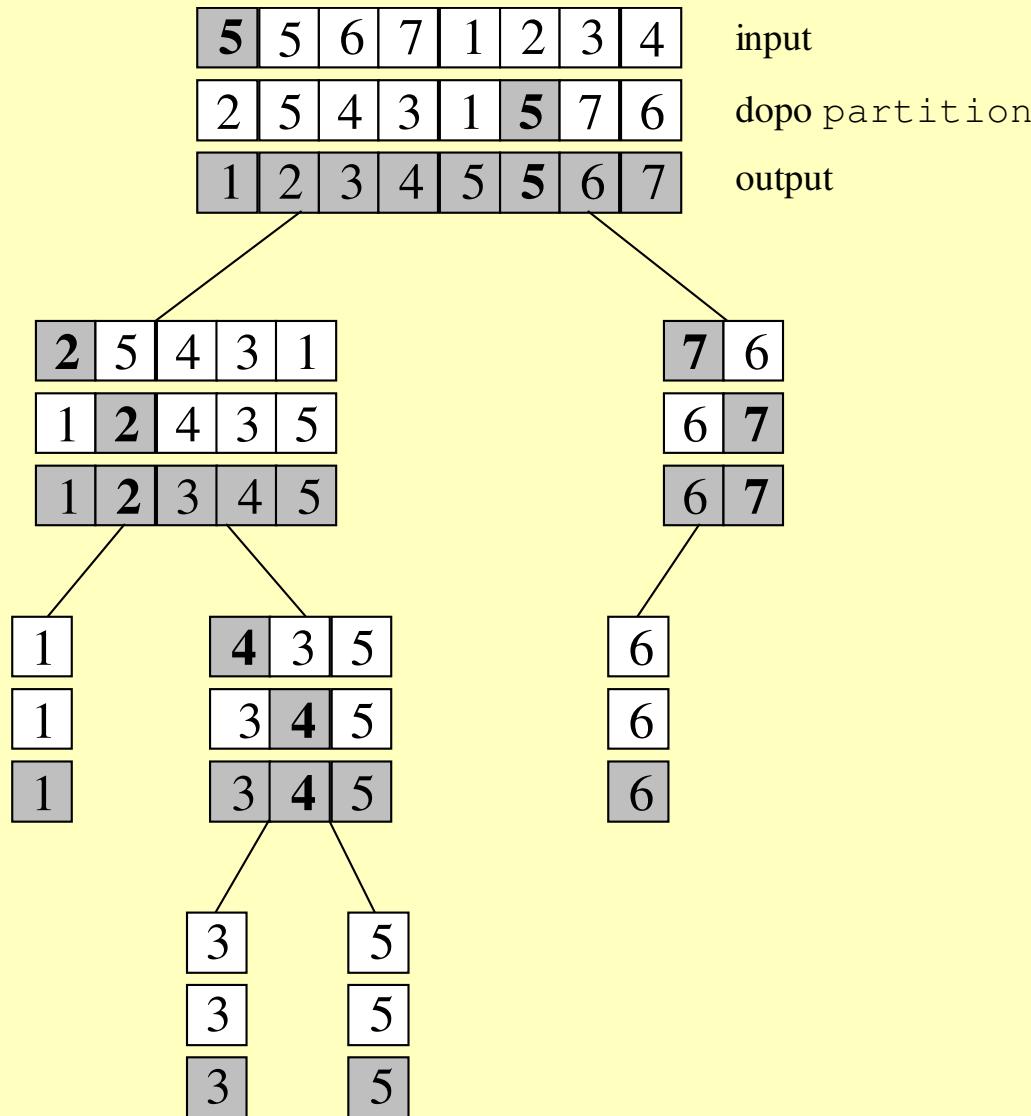
# QuickSort (in “loco”)

**QuickSort (A, i, f )**

1. **if** ( $i \geq f$ ) **then return**
2.  $m = Partition(A, i, f)$
3. QuickSort( $A, i, m - 1$ )
4. QuickSort( $A, m + 1, f$ )

- $m$ = indice del pivot ( $A[m]$  è il pivot)
- La chiamata iniziale è QuickSort( $A, 1, n$ )
- Dopo ogni partizione vale l'**invariante**:
  - Ogni elemento di  $A[i..m-1]$  è  $\leq$  del pivot  $A[m]$  e ogni elmento di  $A[m+1..f]$  è  $>$  del pivot  $A[m]$

# Esempio di esecuzione

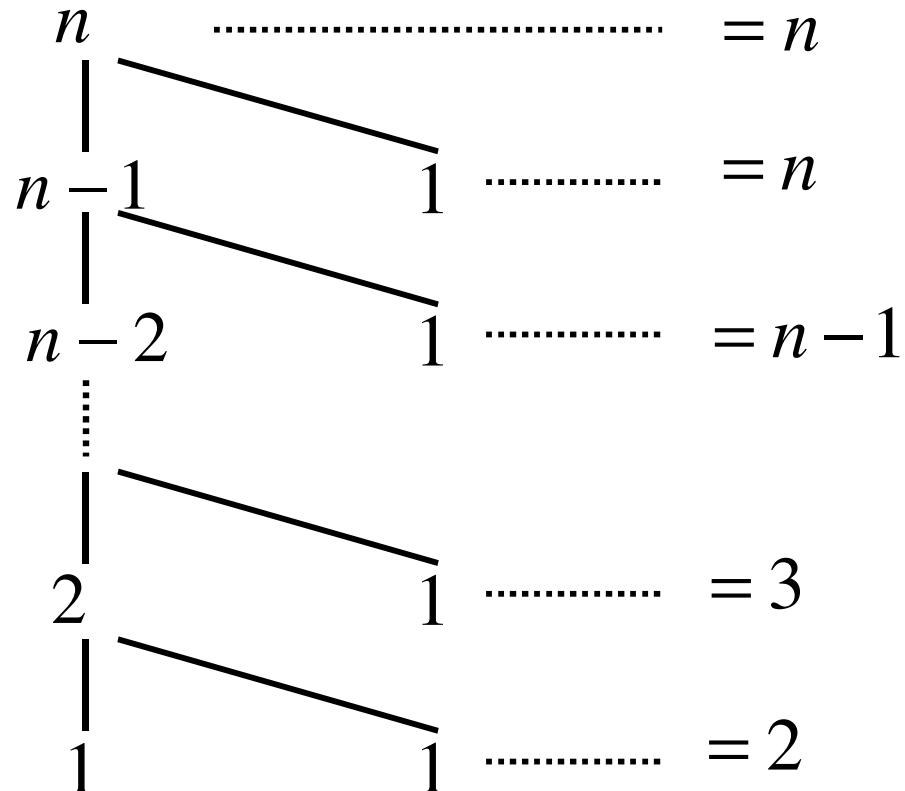


L'albero delle chiamate ricorsive può essere sbilanciato!

# QuickSort: analisi nel caso peggiore

- Nel caso peggiore, **il perno scelto ad ogni passo è il minimo o il massimo** degli elementi nell'array (ovvero quando l'array di partenza è ordinato o ordinato in senso inverso!)
- Il numero di confronti è pertanto:  
$$T(n) = \Theta(n) + T(1) + T(n-1) = \Theta(n) + T(n-1)$$
- Svolgendo per iterazione si ottiene:  
$$T(n) = \Theta(n^2)$$

# QuickSort: analisi nel caso peggiore



- Albero di ricorsione dei costi (confronti)

$$T(n) = \sum_{i=2}^n i + n = \Theta(n^2)$$

# QuickSort: analisi nel caso migliore

- Il partizionamento produce un albero di ricorsione perfettamente bilanciato
  - due sottoalberi di dimensione non maggiore di  $n/2$  ciascuno
- Si ha la ricorrenza del MergeSort:

$$T(n) \leq 2 T(n/2) + \Theta(n)$$

che per il caso 2 del teorema Master:

$$T(n) = \Theta(n \lg n)$$

# Randomizzazione

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

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

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  $C(n) \leq 2 n \log n$  quindi:  $T(n) = O(n \lg n)$

*Dimostrazione per sostituzione*

Assumiamo per ipotesi induttiva che  $C(a) \leq 2 a \log a$

$$\rightarrow C(n) \leq n-1 + \sum_{a=0}^{n-1} \frac{2}{n} 2 a \log a \leq n-1 + \frac{2}{n} \int_2^n x \log x dx$$

Integrando per parti si dimostra che  $C(n) \leq 2 n \log n$



# Riepilogo

- **Algoritmi elementari**  
(InsertionSort, SelectionSort, BubbleSort, ecc..):
  - quadratici nel caso peggiore
  - ordinano “in loco”
- **Algoritmi ottimi ( $O(n \lg n)$  confronti):**
  - MergeSort (ma non ordina in loco), HeapSort
  - QuickSort (ordina in loco):  $O(n \lg n)$  solo nel caso medio, quadratico nel caso peggiore; altamente “configurabile” e per questo il più veloce e il più usato nella pratica

# Tabella riassuntiva

| Nome           | Migliore      | Medio         | Peggiore      | Stabile | In loco | Caratteristiche dell'ordinamento                                                   |
|----------------|---------------|---------------|---------------|---------|---------|------------------------------------------------------------------------------------|
| Bubble sort    | $O(n)$        | $O(n^2)$      | $O(n^2)$      | Sì      | Sì      | Algoritmo per confronto tramite scambio di elementi                                |
| Insertion sort | $O(n)$        | $O(n^2)$      | $O(n^2)$      | Sì      | Sì      | A. per confronto tramite inserimento                                               |
| Merge sort     | $O(n \log n)$ | $O(n \log n)$ | $O(n \log n)$ | Sì      | No      | A. per confronto tramite unione di componenti, ottimale e facile da parallelizzare |
| Quicksort      | $O(n \log n)$ | $O(n \log n)$ | $O(n^2)$      | No      | Sì      | A. per confronto tramite partizionamento. Le sue varianti possono: essere stabili  |
| Selection sort | $O(n^2)$      | $O(n^2)$      | $O(n^2)$      | No      | Sì      | A. per confronto tramite selezione di elementi                                     |

# Algoritmi e Strutture Dati

## Capitolo 4 Ordinamento lineare

Camil Demetrescu, Irene Finocchi,  
Giuseppe F. Italiano



# Ordinamenti lineari

## (per dati in **input** con proprietà particolari)

- IntegerSort  
(o CountingSort)
- BucketSort
- RadixSort

# IntegerSort: fase 1

- Ordina un array X di n interi con valori in [1,k] per k costante intera non troppo grande
- Mantiene un array Y di k contatori tale che  $Y[i] = \text{numero di volte che il valore } i \text{ compare nell'array di input } X$

|   |   |   |   |   |   |   |   |   |  |
|---|---|---|---|---|---|---|---|---|--|
| X | 5 | 1 | 6 | 8 | 6 |   |   |   |  |
| Y | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 |  |
|   | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |  |

|   |   |   |   |   |   |   |   |   |  |
|---|---|---|---|---|---|---|---|---|--|
| 5 | 1 | 6 | 8 | 6 |   |   |   |   |  |
| 1 | 0 | 0 | 0 | 0 | 1 | 0 | 0 | 0 |  |
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |   |  |

|   |   |   |   |   |   |   |   |   |  |
|---|---|---|---|---|---|---|---|---|--|
| 5 | 1 | 6 | 8 | 6 |   |   |   |   |  |
| 1 | 0 | 0 | 0 | 0 | 1 | 1 | 0 | 0 |  |
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |   |  |

|   |   |   |   |   |   |   |   |   |  |
|---|---|---|---|---|---|---|---|---|--|
| X | 5 | 1 | 6 | 8 | 6 |   |   |   |  |
| Y | 1 | 0 | 0 | 0 | 1 | 1 | 0 | 1 |  |
|   | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |  |

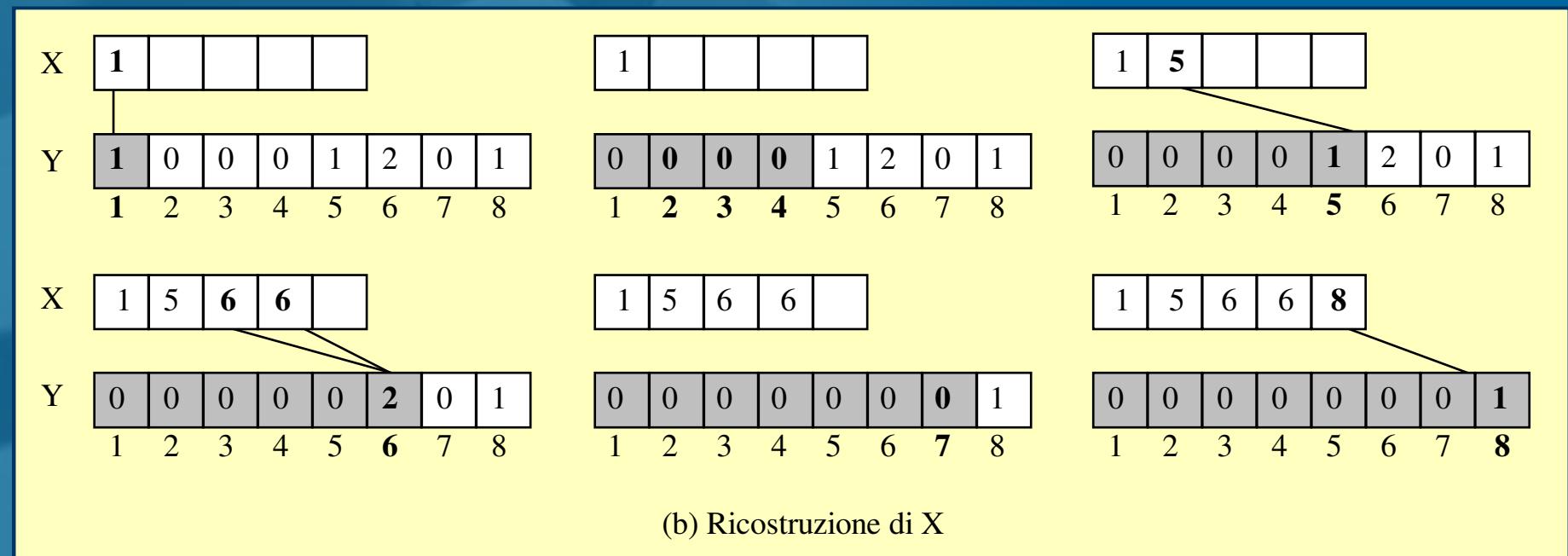
|   |   |   |   |   |   |   |   |   |  |
|---|---|---|---|---|---|---|---|---|--|
| 5 | 1 | 6 | 8 | 6 |   |   |   |   |  |
| 1 | 0 | 0 | 0 | 0 | 1 | 2 | 0 | 1 |  |
| 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |   |  |

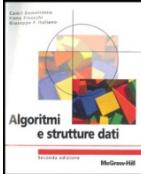
(a) Calcolo di Y

# IntegerSort: fase 2

**Ricostruzione di X:** L'indice i di Y è il valore da ricopiare in X, mentre l'elemento  $y[i]$  è il numero di copie di i

Scorre Y da sinistra verso destra e, se  $Y[i]=c$ , scrive in X il valore i per c volte





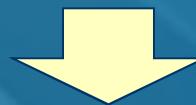
# IntegerSort: pseudocodice

**IntegerSort (X, k)**

1. Sia Y un array di dimensione k //Fase 1: calcolo di Y
2. **for**  $i=1$  **to**  $k$  **do**  $Y[i]=0$  //inizializzazione di Y a 0
3. **for**  $j=1$  **to**  $n$  **do**  $Y[X[j]]++$   
// Y[i] ora contiene il num. di elementi in X uguali a i,  $i=1..k$
4.  $j=1$  //indice per scandire X
5. **for**  $i=1$  **to**  $k$  **do** //Fase 2: ricostruzione di X a partire da Y
6.     **while** ( $Y[i] > 0$ ) **do** //considera solo i valori Y[i] non nulli
7.          $X[j]=i$
8.          $j++$
9.          $Y[i] = Y[i] - 1$

# IntegerSort: analisi

- Tempo  $O(k)$  per inizializzare  $Y$  a 0
- Tempo  $O(n)$  per calcolare i valori dei contatori
- Tempo  $O(n+k)$  per ricostruire  $X$

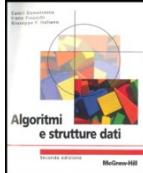


$O(n+k)$

Tempo lineare se  $k=O(n)$

Contraddice il lower bound di  $\Omega(n \log n)$ ?

No, perché l'Integer Sort non è un algoritmo basato su confronti!



# BucketSort

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

- Esempio: ordinare  $n$  record con campi:
  - nome, cognome, anno di nascita, matricola,...
- si potrebbe voler ordinare per matricola o per anno di nascita

Input del problema:

- $n$  record mantenuti in un array
- ogni elemento dell'array è un record con
  - campo chiave (rispetto al quale ordinare)
  - altri campi associati alla chiave (informazione satellite)



# BucketSort

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  $x$
- Concatenare poi le liste

# Esempio

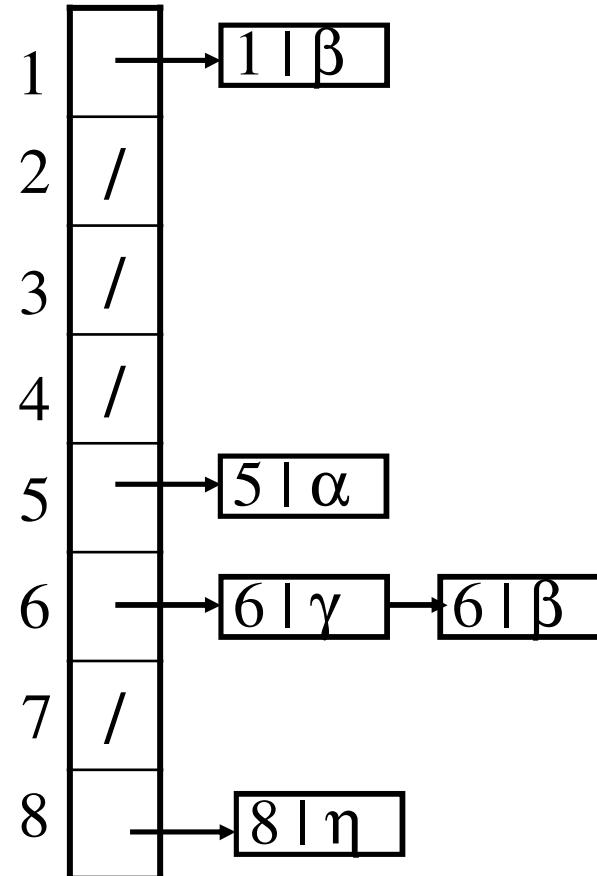
Y è indicizzato con i valori delle chiavi dei record in X

X

*chiave info satellite*

|   |   |          |
|---|---|----------|
| 1 | 5 | $\alpha$ |
| 2 | 1 | $\beta$  |
| 3 | 6 | $\gamma$ |
| 4 | 8 | $\eta$   |
| 5 | 6 | $\beta$  |

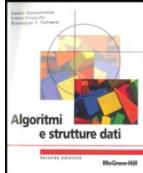
Y



X (ordinato)

*chiave info satellite*

|   |   |          |
|---|---|----------|
| 1 | 1 | $\beta$  |
| 2 | 5 | $\alpha$ |
| 3 | 6 | $\gamma$ |
| 4 | 6 | $\beta$  |
| 5 | 8 | $\eta$   |



# BucketSort: pseudocodice

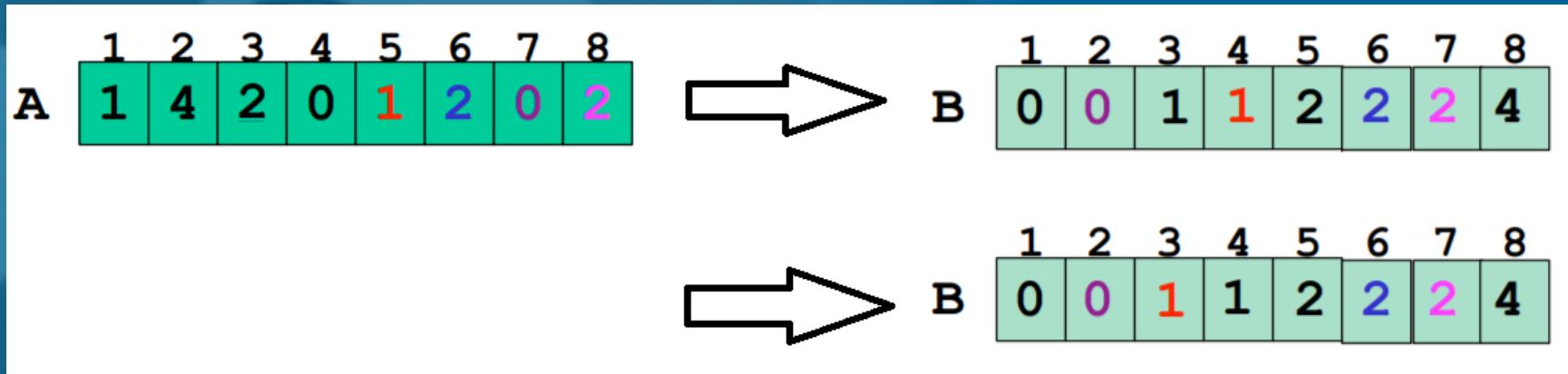
**BucketSort (X, k)**

1. Sia Y un array di dimensione k
2. **for**  $i=1$  **to**  $k$  **do**  $Y[i]=$ lista vuota //*Inizializzazione*
3. **for**  $j=1$  **to**  $n$  **do**
4.     **if** ( $\text{key}(X[j]) \notin [1,k]$  ) **then errore**
5.     **else** appendi il record  $X[j]$  alla lista  $Y[\text{key}(X[j])]$
6. **for**  $i=1$  **to**  $k$  **do** //*Ordinamento classico a livello di bucket!*
7.     copia “ordinatamente” in X gli elementi della lista  $Y[i]$

In media, tempo  $O(n+k)$ , ovvero  $O(n)$  se  $k = O(n)$

# Stabilità

- Un algoritmo è **stabile** se preserva l'ordine iniziale tra elementi con la stessa chiave



# Esempio di applicazione

| <b>voli</b> | <b>ordinati per orario</b> | <b>ordinati per destinazioni (non stabile)</b> | <b>ordinati per destinazioni (stabile)</b> |
|-------------|----------------------------|------------------------------------------------|--------------------------------------------|
|             | <b>Londra 09:05:00</b>     | <b>Berlino 09:25:00</b>                        | <b>Berlino 09:25:00</b>                    |
|             | <b>Londra 09:20:00</b>     | <b>Berlino 10:00:00</b>                        | <b>Berlino 10:00:00</b>                    |
|             | <b>Berlino 09:25:00</b>    | <b>Londra 09:30:00</b>                         | <b>Londra 09:05:00</b>                     |
|             | <b>Londra 09:30:00</b>     | <b>Londra 09:05:00</b>                         | <b>Londra 09:20:00</b>                     |
|             | <b>Parigi 09:30:00</b>     | <b>Londra 09:20:00</b>                         | <b>Londra 09:30:00</b>                     |
|             | <b>Parigi 09:40:00</b>     | <b>Parigi 10:00:00</b>                         | <b>Parigi 09:30:00</b>                     |
|             | <b>Parigi 09:50:00</b>     | <b>Parigi 09:50:00</b>                         | <b>Parigi 09:50:00</b>                     |
|             | <b>Berlino 10:00:00</b>    | <b>Parigi 09:30:00</b>                         | <b>Parigi 10:00:00</b>                     |



# Stabilità

- Il BucketSort è reso stabile appendendo gli elementi di X **in coda** alla opportuna lista Y[i]
- **Domanda:** gli altri algoritmi di ordinamento che abbiamo visto finora sono stabili?
  - CountingSort?
  - Mergesort?
  - Quicksort?

# RadixSort

- Rappresentiamo i valori in una certa base  $b$ , ed eseguiamo una serie di **BucketSort** sulle cifre in modo *bottom-up*: dalla cifra meno significativa verso quella più significativa

**Radixsort(A,t)= for  $i=0$  to  $t-1$  do**

**<usa un ordinamento stabile lineare per ordinare gli elementi di A sulla cifra  $i$ >**

Esempio per interi a  $t=4$  cifre in base  $b=10$



# Correttezza

- Se  $x$  e  $y$  hanno una diversa  $i$ -esima cifra, la  $i$ -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



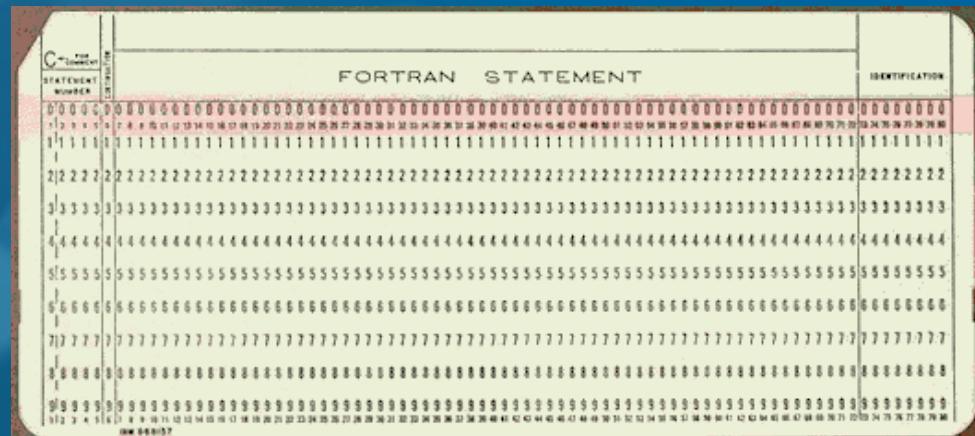
Dopo la  $t$ -esima passata di BucketSort, i numeri sono correttamente ordinati rispetto alle  $t$  cifre meno significative

# RadixSort: esempio di applicazione

IBM 802 sorter



12X80



[https://en.wikipedia.org/wiki/IBM\\_card\\_sorter](https://en.wikipedia.org/wiki/IBM_card_sorter)



# Altre applicazioni

- Ordinamento tra stringhe
- Indicizzazione di testi
- Controllo copiature e plagio
- Biologia molecolare computazionale
  - Data una stringa di N caratteri, trovare la sottostringa ripetuta più lunga

a a c a a g t t t a c a a g c

# RadixSort: pseudocodice

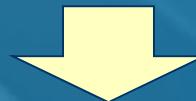
```
procedura bucketSort(array A di n interi, interi b e t)
1. sia Y un array di dimensione b
2. for i = 1 to b do Y[i] ← lista vuota
3. for i = 1 to n do
4. c ← t-esima cifra di A[i] nella rappresentazione in base b
5. appendi A[i] alla lista Y[c + 1]
6. for i = 1 to b do
7. copia ordinatamente in A gli elementi della lista Y[i]
```

```
algoritmo radixSort(array A di n interi)
8. t ← 0
9. while (esiste un numero la cui t-esima cifra è ≠ 0)
10. bucketSort(A, 10, t)
11. t ← t + 1
```

**Figura 4.23** L'algoritmo radixSort, utilizzando come base per il bucketSort il valore 10. La  $t$ -esima chiamata al bucketSort considera come chiave la  $t$ -esima cifra meno significativa della rappresentazione decimale dei numeri.

# Tempo di esecuzione

- $O(\log_b k)$  passate di bucketsort per  $n$  interi in  $[1, k]$
- Ciascuna passata richiede tempo  $O(n+b)$



$$\log_2 k = \log_n k \log_2 n$$

$$O((n+b) \log_b k)$$

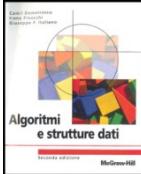
Se  $b = \Theta(n)$ , si ha  $O(n \log_n k) = O\left(n \frac{\lg k}{\lg n}\right)$

→ Tempo lineare se  $k=O(n^c)$ ,  $c$  costante



# Esempio

- Si supponga di voler ordinare  $n=10^6$  numeri da 32 bit
- Come scelgo la base  $b$ ?
- $n=10^6$  è compreso fra  $2^{19}$  e  $2^{20}$
- Scegliendo  $b=2^{16}=\Theta(n)$  si ha:
  - $k = 2^{32}-1 = O(n^c)$  con  $c=2$
  - sono sufficienti 2 passate di bucketSort
  - ogni passata richiede tempo lineare  $O(n)$



# Riepilogo

- Bucket sort e Radix sort sono generalizzazioni del counting sort.
  - Se ogni bucket ha dimensione 1 si ottiene il Counting sort
- Il **Counting Sort** assume che gli elementi siano in  $[1,k]$ ; utile se  $k$  è piccolo ( $k=O(n)$ )
- Il **Bucket sort** presuppone la conoscenza sulla distribuzione dell'input, caso peggiore  $\Theta(n^2)$ , caso medio  $\Theta(n)$ .
- Il **Radix sort** assume che gli interi siano di  $t$  cifre, con ogni cifra in base  $b$ . Utile quando i numeri da ordinare sono molti,  $t$  è costante e piccolo.
- Nessuno ordina in loco!

# Esercizio: Oracolo

Definire un algoritmo che, dato in input un array X di  $n$  interi in  $[1, k]$ , processa X in modo da poter poi rispondere a domande del tipo:

“*Quanti interi di X cadono nell’intervallo  $[a, b]$ ?*”, per  $a$  e  $b$  qualsiasi (anche non appartenenti a X).

- L’algoritmo deve richiedere tempo di **pre-processamento  $O(n + k)$**  per costruire l’oracolo (un array Y)
  - *costrisciOracolo* (X, k): Y
- Mentre l’**oracolo** deve poter poi rispondere in **tempo  $O(1)$**  alla domanda
  - *interrogaOracolo* (Y, k, a, b): c



UNIVERSITÀ  
DEGLI STUDI  
DI BERGAMO

Dipartimento  
di Ingegneria Gestionale,  
dell'Informazione e della Produzione



# Alberi **rosso**-neri nel JCF

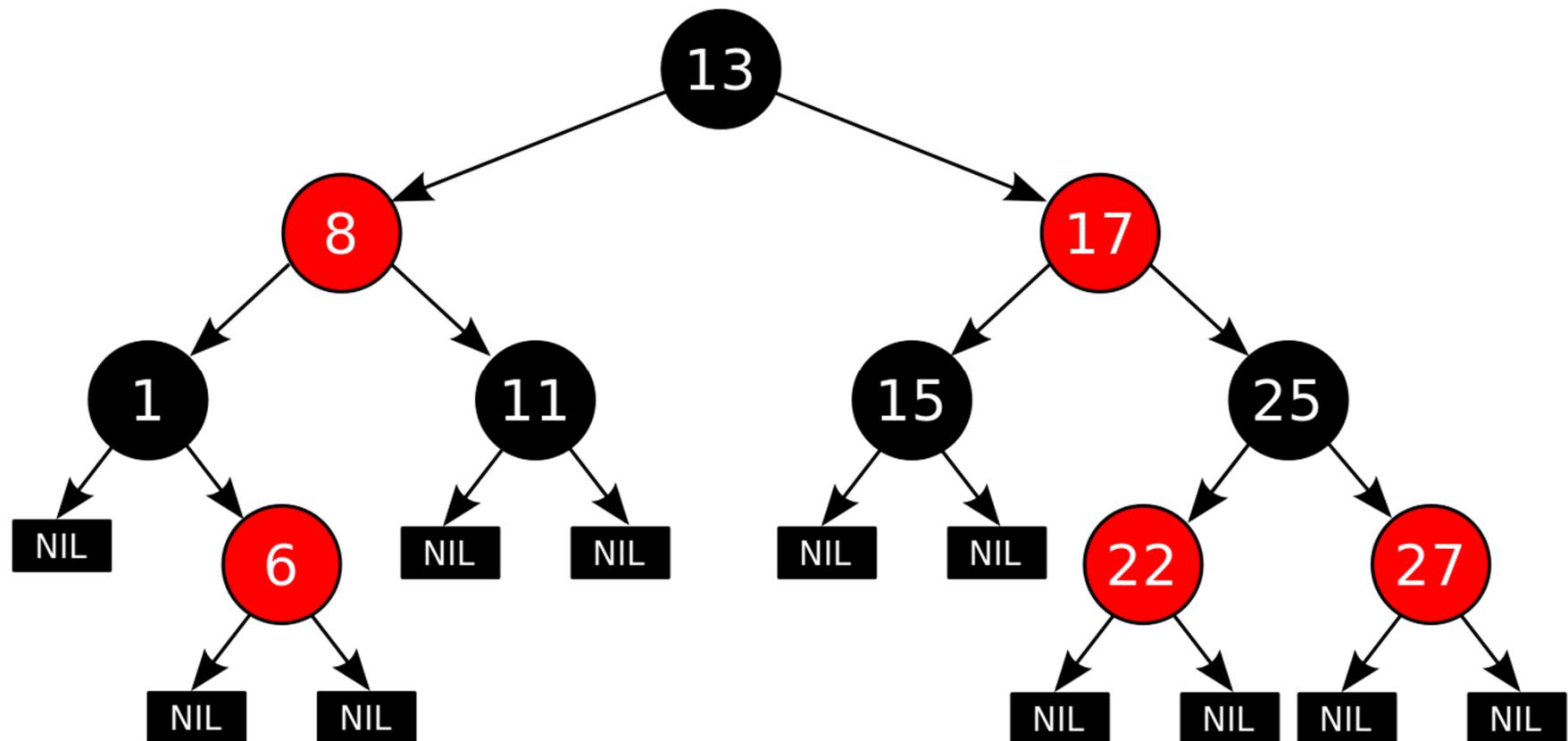
Corso di laurea  
**Magistrale in**  
**Ingegneria**  
**Informatica**

PROGETTAZIONE, ALGORITMI E  
COMPUTABILITÀ  
(38090-MOD1)

RELATORE  
Prof.ssa Patrizia  
Scandurra

SEDE  
DIGIP

# Esempio di albero rosso-nero



Alberi binari di ricerca colorati

Foglie *NIL* o *sentinella*  
(elemento nullo)

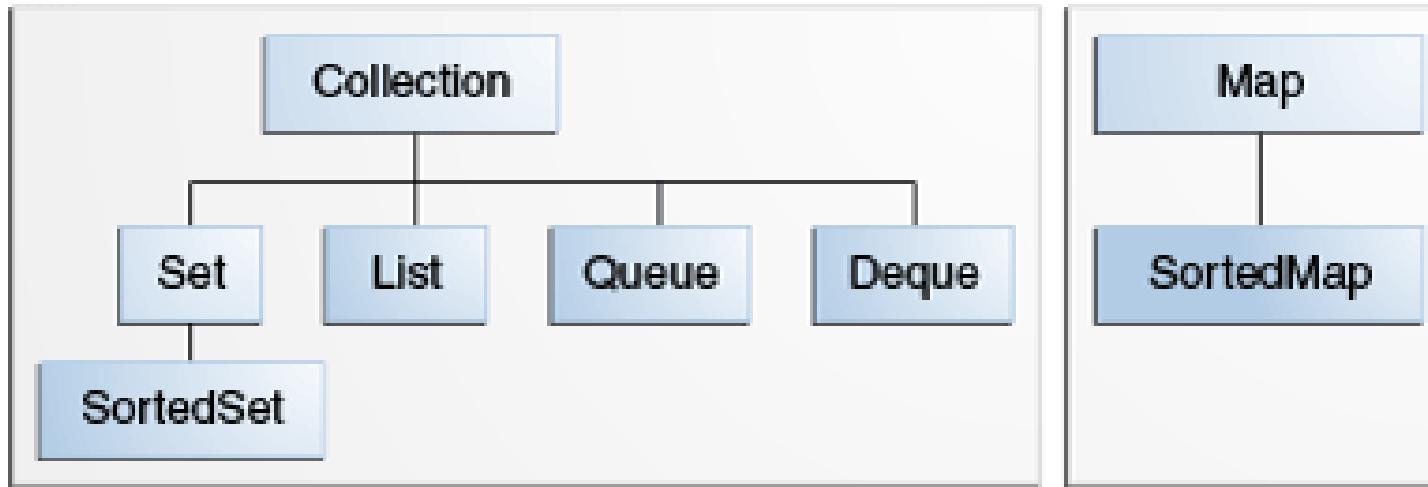
# Bilanciamento con albero rosso-nero

- Albero binario di ricerca
- Proprietà sul colore (vincoli per il bilanciamento):
  1. ogni nodo è colorato o di **nero** o di **rosso**
  2. la radice e le *foglie NIL* sono **nere**
  3. i nodi **rossi** possono avere solo figli **neri**
  4. Ogni cammino da un nodo ad una foglia contiene lo stesso numero di nodi **neri**
- Algoritmi di bilanciamento:
  - I nuovi nodi vengono inseriti come nodi **red**
  - A seguito di aggiornamenti che violano la proprietà 3, una serie di trasformazioni (rotazioni e ri-colorazione) vengono eseguite per ripristinarla
    - Costo nel caso peggiore: visita discendente dell'albero in  $\lg n$  passi

# Perché l'albero rosso-nero funziona?

- I vincoli rafforzano una proprietà importante degli alberi rosso-neri
- **Proprietà:** il cammino più lungo dalla radice a una foglia è al massimo lungo il doppio del cammino più breve.
  - Nessun cammino può avere due nodi **rossi** in fila, per la proprietà 3
  - Il cammino più breve possibile ha tutti nodi neri, e il più lungo alterna nodi **rossi** e nodi neri
  - Poiché tutti i cammini massimi hanno lo stesso numero di nodi neri, per la proprietà 3, ciò dimostra che nessun cammino è lungo più del doppio di qualsiasi altro cammino.
- Le operazioni di bilanciamento non impattano sulla complessità in modo eccessivo
  - best case: no fixup è necessario
  - worst case: fixup su tutto l'albero, con costo  $\log n$

# Recap JCF - interfacce



- L'interfaccia **Map** è l'interfaccia *Dizionario* che rappresenta una collezione di coppie (*key, value*):
  - *key* univoca
  - *value* (non necessariamente univoca)

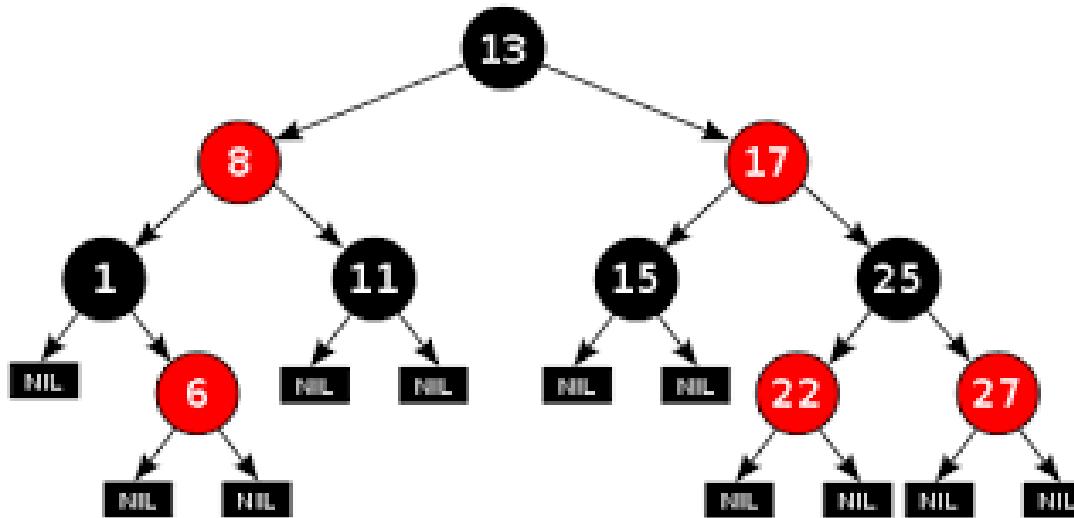
|       |
|-------|
| key   |
| value |

# L'interfaccia Map

- Map definisce i metodi:
  - `put(k,v)` – inserire una coppia (k,v)
  - `remove(k)` – rimuovere l'elemento di chiave k
  - `containsKey(k)` – ricerca in base alla chiave k
  - `containsValue(v)` – ricerca in base al valore v
    - Ci può essere più di una coppia con chiave distinta
  - `size()`, `equals()`, `clear()`

# Gli alberi nel JCF

- Due implementazioni:
  - `TreeMap`
  - `TreeSet` (degenerate case of `TreeMap`)
- Basate sugli alberi **rosso-neri**: alberi binari di ricerca auto-bilancianti



# Caratteristiche delle API

- Ordinamento
  - Default: ordinamento naturale
  - E' possibile cambiare ordinamento passando un oggetto **Comparator** al costruttore
- Thread safety
  - **TreeSet** e **TreeMap** non sono **thread safe**
  - Occore usare i wrapper di sincronizzazione della classe **Collections**
    - **unmodifiableSortedSet**, **unmodifiableSortedMap**
    - **synchronizedSortedSet**, **synchronizedSortedMap**

# TreeMap : implementazione di un BST nel JCF

- TreeMap memorizza il dizionario di coppie in un albero **rosso-nero**, ordinato in base alle chiavi k

```
public class TreeMap<K, V>
 implements SortedMap<K, V>
 extends AbstractMap<K, V>
```

- Esempio:

```
TreeMap<Integer, Student> students;
//Integer: unique student ID number
//Student: student object
```

# TreeMap: metodi

- Implementa l'interfaccia Map
  - Non ha il metodo `iterator()`
- E altri metodi per gli alberi BST

| Operation       | Method                                                                             |
|-----------------|------------------------------------------------------------------------------------|
| Visita ordinata | <code>Set keySet();</code>                                                         |
| Inserimento     | <code>put(Object key, Object value);</code><br><code>putAll(Map map);</code>       |
| Ricerca         | <code>containsKey(Object key);</code><br><code>containsValue(Object value);</code> |
| Cancellazione   | <code>remove(Object key);</code>                                                   |
| Successore      | <code>SortedMap tailMap(Object key);</code>                                        |
| Predecessore    | <code>SortedMap headMap(Object key);</code>                                        |

# TreeMap: costruttori

```
public TreeMap()
{
 //comparator = null;
 // Comparable interface
} // default constructor

public TreeMap(Comparator<? super K> c)
{
 comparator = c; // c implementa l'interfaccia Comparator
} // one-parameter constructor
```

# **TreeSet: una collezione ordinata senza duplicati**

- Implementa tutti i metodi dell'interfaccia **Collection**
  - add, remove, size, contains, ...
  - `toString` (ereditato da `AbstractCollection`)

```
public class TreeSet<E>
 extends AbstractSet<E>
 implements SortedSet<E>, Cloneable,
 java.io.Serializable
{ ... }
```

# TreeSet: API

| Operazione      | Metodo                                                            |
|-----------------|-------------------------------------------------------------------|
| Visita ordinata | <code>Iterator iterator();</code>                                 |
| Inserimento     | <code>add(Object o);</code><br><code>addAll(Collection c);</code> |
| Ricerca         | <code>contains(Object o);</code>                                  |
| Cancellazione   | <code>remove(Object o);</code>                                    |
| Successore      | <code>SortedSet tailSet(Object from);</code>                      |
| Predecessore    | <code>SortedSet headSet(Object to);</code>                        |

# TreeSet: costruttori

```
public TreeSet()
 // assumes elements ordered by
 // Comparable interface

public TreeSet(Comparator<? super E> c)
 // assumes elements ordered
 // by Comparator c

public TreeSet(Collection<? extends E> c)
 // copy constructor; assumes elements ordered
 // by Comparable interface
```

# La classe TreeSet è implementata però sulla base della classe TreeMap

- Gli elementi del TreeSet sono le **chiavi** di una TreeMap
- Un dummy object rappresenta il campo **valore** del TreeMap
- Tutte le operazioni del TreeSet sono implementate attraverso le operazioni del TreeMap

```
/**
 * Initializes this TreeSet object to be empty.
public TreeSet()
{
 this (new TreeMap<E, Object>());
} // default constructor
```

# Esercizio

- Implementare una componente software in Java (interfaccia e classe!) che gestisce una struttura dati albero binario **rosso-nero** per un insieme di stringhe. La relazione di ordinamento totale tra le stringhe è quella lessicografica.
- In particolare la componente software deve consentire di:
  1. Inserire una nuova stringa nell'albero
  2. Cancellare una stringa
  3. Verificare se una stringa è stata inserita in precedenza
  4. Restituire una stringa ottenuta concatenando in ordine alfabetico tutte le stringhe memorizzate nell'albero.

# Algoritmi e Strutture Dati

## Capitolo 10 Programmazione dinamica

Camil Demetrescu, Irene Finocchi,  
Giuseppe F. Italiano



# Divide et impera

- Tecnica top-down:
  - 1 Dividi l'istanza del problema in due o più sottoistanze
  - 2 Risolvi ricorsivamente il problema sulle sottoistanze
  - 3 Ricombina la soluzione dei sottoproblemi allo scopo di ottenere la soluzione globale
- Esempi: ricerca binaria, mergesort, quicksort

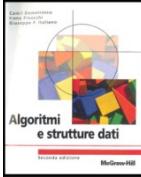
# Altre tecniche algoritmiche

**Divide et impera** applicabile solo se:

- il numero di sotto-problemi da risolvere ricorsivamente è **POLINOMIALE** nella dimensione dell'input
- i sotto-problemi sono **INDIPENDENTI**
  - non devo risolvere più volte la stessa istanza di sotto-problema

Se ciò non è garantito si usano altre strategie, come:

- **Programmazione dinamica**
- **Golosa (greedy)**
- Tipicamente per problemi di ottimizzazione



# Programmazione dinamica

- Tecnica **bottom-up**:
  1. Identifica dei sottoproblemi del problema originario, procedendo logicamente **dai problemi più piccoli verso quelli più grandi**
  2. Utilizza una **tabella delle soluzioni parziali** per **memorizzare le soluzioni** dei sottoproblemi
  3. Si usa quando **i sottoproblemi non sono indipendenti**
    - Quando si ripresenta uno stesso sottoproblema è sufficiente recuperare la sua soluzione dalla tabella
  4. Può risultare onerosa perché risolve tutti i sottoproblemi in modo esaustivo



# Un esempio: *Numeri di Fibonacci*

## Definizione ricorsiva (o induttiva)

- $F(1) = F(0) = 1$
- $F(n) = F(n-1) + F(n-2)$

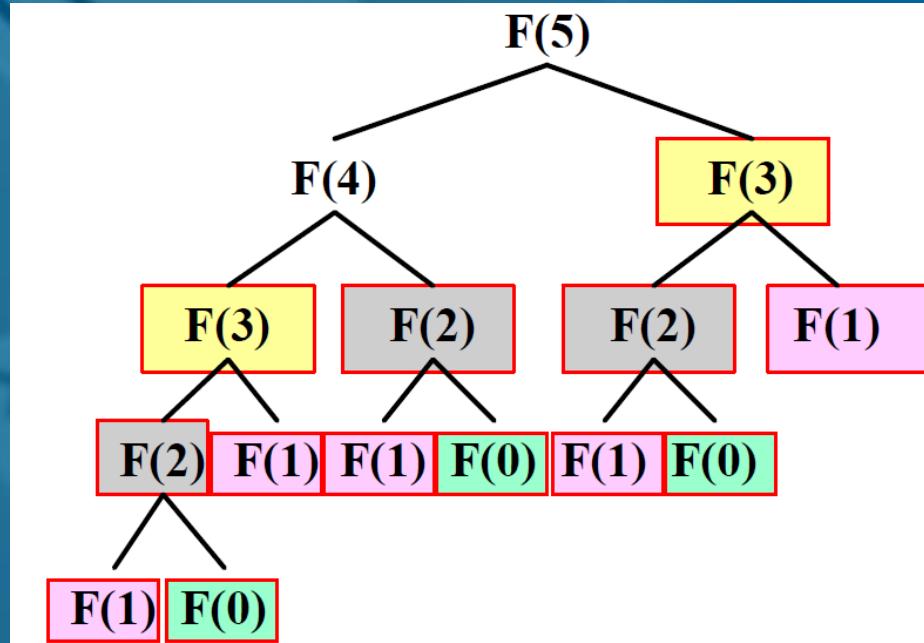
## Algoritmo ricorsivo (Divide et impera)

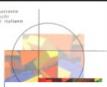
```
algoritmo fibonaccil(intero n) → intero
if n = 0 or n = 1 then return 1
else return fibonaccil (n-1) + fibonaccil (n-2)
```

# Un esempio: *Numeri di Fibonacci*

Tempo di esecuzione finonacci1:  $T(n)=O(2^n)$

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

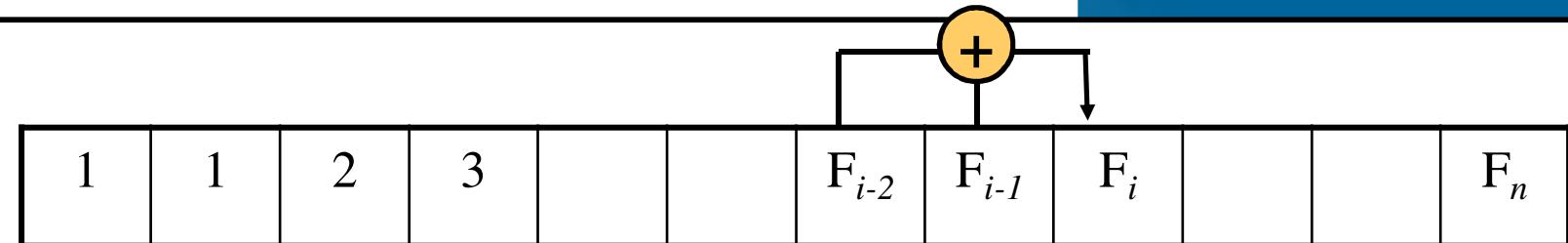




**algoritmo** fibonacci2(*intero n*) → *intero*  
sia *Fib* un array di *n* interi (indicizzato 1..*n*)  
 $Fib[1] \leftarrow Fib[2] \leftarrow 1$   
**for** *i* = 3 **to** *n* **do**  
    *Fib[i]*  $\leftarrow Fib[i-1] + Fib[i-2]$   
**return** *Fib[n]*

## Esempio Fibonacci

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



1. Identifichiamo i **sottoproblemi**: l'*i*-esimo problema consiste nel calcolo dell'*i*-esimo numero di fibonacci. Soluzioni in una tabella Fib.
2. Calcoliamo le **soluzioni di alcuni sottoproblemi “facili”**:  $F[1]=1$  e  $F[2]=1$
3. All'*i*-esimo passo, **avanziamo sulla tabella** e calcoliamo soluzione all'*i*-esimo sottoproblema in base alle soluzioni precedentemente calcolate:  
 $F[i] \leftarrow F[i-1] + F[i-2]$
4. Restituiamo il valore memorizzato in un **particolare elemento** della tabella, ovvero  $F[n]$



# Distanza fra due stringhe

- Come definire la distanza?
- Intuitivamente:
  - gaber e haber sono stringhe molto simili (vicine)
  - zolle e abracadabra sono molto diverse (lontane)
- edit distance:
  - numero *minimo* di “modifiche elementari” per trasformare una stringa in un’altra
  - usata nei correttori ortografici (*spell checker*):
    - se una parola digitata non è nel dizionario sostituiscila con la più vicina presente nel dizionario

# Distanza tra due stringhe

Siano X e Y due stringhe di lunghezza m ed n:

$$X = x_1 \cdot x_2 \cdot \dots \cdot x_m$$

$$Y = y_1 \cdot y_2 \cdot \dots \cdot y_n$$

Calcoliamo la “distanza” tra X e Y come **minimo** numero di operazioni elementari che trasformano X in Y scelte tra:

**inserisci**(*a*):

Inserisci il carattere *a* nella posizione corrente della stringa.

**cancella**(*a*):

Cancella il carattere *a* dalla posizione corrente della stringa.

**sostituisci**(*a, b*):

Sostituisci il carattere *a* con il carattere *b* nella posizione corrente della stringa.



# Esempio

- Come posso trasformare la stringa **risotto** in **presto**?
- Con 13 operazioni è semplice:
  - 7 cancellazioni
  - 6 inserimenti
- è 13 la distanza?
- ...NO! C'è un numero minore di operazioni che trasforma la prima stringa nella seconda (e viceversa)

# Esempio

| Azione              | Costo | Stringa ottenuta |
|---------------------|-------|------------------|
| Inserisco P         | 1     | P RISOTTO        |
| Mantengo R          | 0     | PR ISOTTO        |
| Sostituisco I con E | 1     | PRE SOTTO        |
| Mantengo S          | 0     | PRES OTTO        |
| Cancello O          | 1     | PRES TTO         |
| Mantengo T          | 0     | PREST TO         |
| Cancello T          | 1     | PREST O          |
| Mantengo O          | 0     | PRESTO           |

la distanza è 4

# Approccio

- Denotiamo con  $\delta(X, Y)$  la distanza tra  $X$  e  $Y$
- Definiamo  $X_i$  il prefisso di  $X$  ( $i$ -esimo carattere incluso) per  $0 \leq i \leq m$ :

$X_0$  è la stringa vuota

$$X_i = x_1 \cdot x_2 \cdot \dots \cdot x_i \text{ se } i \geq 1$$

- Riduciamo il problema di calcolare  $\delta(X, Y)$  al calcolo di  $\delta(X_i, Y_j)$  per ogni  $i, j$  tali che  $0 \leq i \leq m$  e  $0 \leq j \leq n$
- Manteniamo le soluzioni parziali in una tabella  $D$  di dimensione  $(m+1) \times (n+1)$

# Inizializzazione della tabella

- Alcuni sottoproblemi sono molto semplici
- $\delta(X_0, Y_j) = j$  partendo dalla stringa vuota  $X_0$ , basta inserire uno ad uno i j caratteri di  $Y_j$
- $\delta(X_i, Y_0) = i$  partendo da  $X_i$ , basta rimuovere uno ad uno gli i caratteri per ottenere  $Y_0$
- Queste soluzioni sono memorizzate rispettivamente nella prima riga e prima colonna della tabella D



# Esempio

|   |   | P | R | E | S | T | O |
|---|---|---|---|---|---|---|---|
|   | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
| R | 1 |   |   |   |   |   |   |
| I | 2 |   |   |   |   |   |   |
| S | 3 |   |   |   |   |   |   |
| O | 4 |   |   |   |   |   |   |
| T | 5 |   |   |   |   |   |   |
| T | 6 |   |   |   |   |   |   |
| O | 7 |   |   |   |   |   |   |

La tabella D  
inizializzata  
dall'algoritmo

# Avanzamento nella tabella (1/3)

$$\delta(X_i, Y_j) = ? \quad \text{per } i \geq 1 \ j \geq 1$$

- Se  $x_i = y_j$  : il minimo costo per trasformare  $X_i$  in  $Y_j$  è uguale al minimo costo per trasformare  $X_{i-1}$  in  $Y_{j-1}$   
$$D[i, j] = D[i-1, j-1]$$
- Se  $x_i \neq y_j$  : distinguiamo in base all'ultima operazione usata per trasformare  $X_i$  in  $Y_j$  in una sequenza ottima di operazioni  
(vedi slide successiva)

# Avanzamento nella tabella (2/3)

il minimo costo per trasformare  $X_i$  in  $Y_j$

**inserisci**( $y_j$ ):

è uguale al minimo costo per trasformare  $X_i$  in  $Y_{j-1}$  più 1 per inserire il carattere  $y_j$

$$\rightarrow D[i,j] = 1 + D[i, j-1]$$

il minimo costo per trasformare  $X_i$  in  $Y_j$  è

**cancella**( $x_i$ ):

uguale al minimo costo per trasformare  $X_{i-1}$  in  $Y_j$  più 1 per la cancellazione del carattere  $x_i$

$$\rightarrow D[i,j] = 1 + D[i-1, j]$$

# Avanzamento nella tabella (3/3)

**sostituisci**( $x_i, y_j$ ):

il minimo costo per trasformare  $X_i$  in  $Y_j$  è uguale al minimo costo per trasformare  $X_{i-1}$  in  $Y_{j-1}$  più 1 per sostituire il carattere  $x_i$  per  $y_j$

$$\rightarrow D[i,j] = 1 + D[i-1, j-1]$$

In conclusione, per  $i \geq 1$   $j \geq 1$ :

$$D[i,j] = \begin{cases} D[i-1, j-1] & \text{se } x_i = y_j \\ 1 + \min\{D[i, j-1], D[i-1, j], D[i-1, j-1]\} & \text{se } x_i \neq y_j \end{cases}$$



# Pseudocodice

```
algoritmo distanzaStringhe(stringa X, stringa Y) → intero
 matrice D di $(m + 1) \times (n + 1)$ interi
 for i = 0 to m do D[i, 0] ← i
 for j = 1 to n do D[0, j] ← j
 for i = 1 to m do
 for j = 1 to n do
 if (xi ≠ yj) then
 D[i, j] ← 1 + min{D[i, j - 1], D[i - 1, j], D[i - 1, j - 1]}
 else D[i, j] ← D[i - 1, j - 1]
 return D[m, n]
```

Tempo di esecuzione ed occupazione di memoria:  $\Theta(m n)$

**NOTA:** Se tengo traccia della cella della tabella che utilizzo per decidere il valore di  $D[i,j]$ , ho un metodo costruttivo per ricostruire la sequenza di operazioni di editing ottima.

# Esempio

|   |   | P | R | E        | S        | T        | O        |
|---|---|---|---|----------|----------|----------|----------|
|   | 0 | 1 | 2 | 3        | 4        | 5        | 6        |
| R | 1 | 1 | 1 | 2        | 3        | 4        | 5        |
| I | 2 | 2 | 2 | <b>2</b> | 3        | 4        | 5        |
| S | 3 | 3 | 3 | 3        | <b>2</b> | 3        | 4        |
| O | 4 | 4 | 4 | 4        | <b>3</b> | 3        | 3        |
| T | 5 | 5 | 5 | 5        | <b>4</b> | <b>3</b> | 4        |
| T | 6 | 6 | 6 | 6        | 5        | <b>4</b> | 4        |
| O | 7 | 7 | 7 | 7        | 6        | 5        | <b>4</b> |

La tabella D costruita dall'algoritmo.

In grassetto sono indicate due sequenze di operazioni che permettono di ottenere la distanza tra le stringhe

# Esercizio – distributore

- Un distributore di bibite contiene al suo interno  $n$  monete i cui valori (interi positivi) sono rispettivamente  $c[1], c[2], \dots, c[n]$ . Si consideri il problema di decidere se è possibile erogare, in qualsiasi modo, un resto *esattamente uguale a  $R$*  (un intero positivo) utilizzando un opportuno sottoinsieme delle  $n$  monete a disposizione.
1. Descrivere un algoritmo efficiente per decidere se il problema ammette una soluzione oppure no.
  2. Determinare il costo computazionale dell'algoritmo descritto al punto 1.
  3. Modificare l'algoritmo di cui al punto 1 per determinare anche quali sono le monete da erogare per produrre il resto  $R$ .
    - Si noti che *non è necessario erogare il resto con il numero minimo di monete*: è sufficiente erogarlo in modo qualsiasi.

# Svolgimento 1. e 2.

- Definiamo la matrice booleana  $M[1..n, 0..R]$  tale che  $M[i, r] = true$  se e solo se esiste un sottoinsieme delle prime  $i$  monete di valore complessivo uguale a  $r$ .
- Casi base: se  $i=1$  possiamo solo scegliere se usare la prima moneta oppure no. Quindi possiamo erogare solamente un resto pari a zero (non usando la moneta) oppure pari al valore della moneta,  $c[1]$ . Quindi per ogni  $r=0, \dots, R$  abbiamo:

$$M[1, r] = \begin{cases} true & \text{se } r=0 \text{ oppure } r=c[1] \\ false & \text{altrimenti} \end{cases}$$

# Svolgimento 1. e 2.

- Caso generale:  $i > 1$  monete a disposizione per erogare un resto  $r$ . Ci sono due possibilità:
  - Se  $r \geq c[i]$  allora possiamo usare o meno la  $i$ -esima moneta. Se la usiamo, possiamo erogare  $r$  se e solo se possiamo erogare  $r - c[i]$  usando un sottoinsieme delle rimanenti  $i-1$  monete, cioè se  $M[i-1, r-c[i]]$  è *true*. Se decidiamo di non usare la moneta  $i$ -esima,  $r$  è erogabile se e solo se  $r$  è erogabile con un sottoinsieme delle restanti  $i-1$  monete, cioè  $M[i-1, r]$  è *true*.
  - Se  $r < c[i]$ , non possiamo usare la moneta  $i$ -esima perché il suo valore è superiore alla somma da erogare. Quindi il resto è erogabile se e solo se lo è utilizzando le rimanenti  $i-1$  monete.
- Quindi possiamo definire  $M[i, r]$ , per  $i=2, \dots, n$ ,  $r=0, \dots, R$  come:

$$M[i, r] = \begin{cases} M[i-1, r] \vee M[i-1, r - c[i]] & \text{se } r \geq c[i] \\ M[i-1, r] & \text{altrimenti} \end{cases}$$

- Il nostro problema di partenza ammette soluzione se e solo se  $M[n, R] = \text{true}$ .
- Il calcolo della matrice  $M[i, r]$  è effettuato in tempo  $\Theta(nR)$



# Svolgimento 1. e 2.

- L'algoritmo può essere descritto nel modo seguente:

```
algoritmo RESTO(array c[1..n] di int, int R) \rightarrow bool
 array M[1..n, 0..R] di bool;
 // inizializza M[1, r]
 for r:=0 to R do
 if (r == 0 || r == c[1]) then
 M[1, r] := true;
 else
 M[1, r] := false;
 endif
 endfor
 // calcola i restanti elementi della tabella
 for i := 2 to n do
 for r := 0 to R do
 if (r \geq c[i]) then
 M[i, r] := M[i-1, r] || M[i-1, r-c[i]];
 else
 M[i, r] := M[i-1, r];
 endif
 endfor
 endfor
 return M[n, R];
```



# Svolgimento 3.

- Per determinare le monete da usare, calcoliamo una ulteriore matrice booleana  $U[i, r]$  della stessa dimensione di  $M$ :

**$U[i, r] == \text{true}$  se e solo se usiamo la moneta  $i$ -esima per erogare il resto  $r$**

```
algoritmo RESTOMAXMONETE(array c[1..n] di int, int R) → bool
 array M[1..n, 0..R] di bool;
 array U[1..n, 0..R] di bool;
 // inizializza M[1, r] e U[1, r]
 for r:=0 to R do
 if (r == c[1]) then
 M[1, r] := true;
 U[1, r] := true;
 elseif (r == 0) then
 M[1, r] := true;
 U[1, r] := false;
 else
 M[1, r] := false;
 U[1, r] := false;
 endif
 endfor
 //((continua nella slide successiva))
```



# Svolgimento 3.

```
// calcola i restanti elementi delle tabelle
for i:=2 to n do
 for r := 0 to R do
 if (r ≥ c[i]) then
 M[i, r] := M[i-1, r] || M[i-1, r-c[i]];
 U[i, r] := M[i-1, r-c[i]];
 else
 M[i, r] := M[i-1, r];
 U[i, r] := false; // non usiamo la moneta i-esima
 endif
 endfor
endfor
//costruzione della soluzione (monete selezionate)
if (M[n, R] == true) then
 int i := n; int r := R;
 while (r > 0) do
 if (U[i, r] == true) then
 print "uso la moneta" i;
 r := r - c[i];
 endif
 i := i - 1;
 endwhile
else print "nessuna soluzione";
endif
return M[n, R];
```

# Esercizio - palinsesto

- Una emittente televisiva deve organizzare il palinsesto di una giornata di 1440 minuti (24 ore). L'emittente dispone di una lista di  $n \geq 1$  programmi le cui durate in minuti (interi) sono rispettivamente  $d[1], \dots, d[n]$ .
- 1. Scrivere un algoritmo di *programmazione dinamica* che restituisce il true se e solo se esiste un opportuno sottoinsieme degli  $n$  programmi la cui durata complessiva sia esattamente di 1440 minuti.
- 2. Determinare il costo computazionale dell'algoritmo di cui al punto 1.
- 3. Raffinare l'algoritmo 1. per individuare anche i programmi che faranno parte della soluzione.



# Svolgimento 1. e 2.

- Definiamo una matrice  $B[1..n, 0..1440]$ , tale che  $B[i, j] = \text{true}$  se e solo se esiste un qualche sottoinsieme dei primi  $i$  programmi  $\{1, \dots, i\}$  la cui durata complessiva sia esattamente  $j$ .
- Se  $i = 1$  abbiamo che  $B[i, j] = \text{true}$  se e solo se  $j = d[1]$  oppure  $j = 0$ .
- Per ogni  $i = 2, \dots, n, j = 0, \dots, 1440$ , gli altri elementi della matrice sono definiti in questo modo:

$$B[i, j] = \begin{cases} B[i-1, j] \vee B[i-1, j-d[i]] & \text{se } j \geq d[i] \\ B[i-1, j] & \text{altrimenti} \end{cases}$$

# Svolgimento 1. e 2.

- L'algoritmo calcola tutti i valori  $B[i, j]$ , e restituisce  $B[n, 1440]$
- Il costo dell'algoritmo è  $\Theta(1440 n) = \Theta(n)$

**Algoritmo PALINSESTO(array d[1..n] di int ) → bool**

```
array B[1..n, 0..1440] di bool;
// Inizializziamo B[1, j]
for j:=0 to 1440 do
 if (j == 0 || j == d[1]) then
 B[1, j] := true;
 else
 B[1, j] := false;
 endif
endfor
// Riempiamo la matrice B
for i:=2 to n do
 for j:=0 to 1440 do
 if (j ≥ d[i]) then
 B[i, j] := B[i-1, j] || B[i-1, j-d[i]];
 else
 B[i, j] := B[i-1, j];
 endif
 endfor
endfor
return B[n, 1440];
```



# Esercizio - ascensore

- Un gruppo di  $n > 0$  persone deve salire su un ascensore che può sostenere un peso massimo di  $C$  kg. Indichiamo con  $p[1], \dots, p[n]$  i pesi (in kg) delle  $n$  persone. Il vettore non è ordinato e i pesi sono numeri interi.
- 1. Scrivere un algoritmo di *programmazione dinamica* che restituisce il **numero max di persone** che possono salire contemporaneamente senza superare la capacità  $C$  dell'ascensore.
- 2. Determinare il costo computazionale dell'algoritmo di cui al punto 1.
- 3. Raffinare l'algoritmo 1. per individuare anche le persone che faranno parte del sottogruppo che sale.

# Svolgimento 1. e 2.

- L'approccio basato sulla programmazione dinamica si basa su una matrice i cui elementi  $N[i,c]$  indicano il massimo numero di persone, scelte tra le prime  $i$ , che è possibile “stipare” in un ascensore avente portata massima di  $c$  Kg per  $i=1, \dots, n$ ,  $c=0, \dots, C$
- Consideriamo il **caso  $i=1$**  (una sola persona):
  - Si ha che  $N[1,c] = 1$  se e solo se  $c \geq p[1]$ , quindi:

$$N[1,c] = \begin{cases} 1 & \text{se } c \geq p[1] \\ 0 & \text{altrimenti} \end{cases}$$

- Nel **caso generico** ( $i>1$ ):

$$N[i,c] = \begin{cases} \max\{N[i-1,c], N[i-1,c - p[i]] + 1\} & \text{se } c \geq p[i] \\ N[i-1,c] & \text{altrimenti} \end{cases}$$

- Dopo aver compilato la tabella di programmazione dinamica in tempo  $\Theta(nC)$ , il risultato si leggerà nella cella  $N[n,C]$ .

# Algoritmi e Strutture Dati

## Capitolo 10 Strategia Greedy

Camil Demetrescu, Irene Finocchi,  
Giuseppe F. Italiano

# Altre tecniche algoritmiche

- Divide et impera

**Applicabili solo se il numero di sotto-problemi da risolvere ricorsivamente è POLINOMIALE nella dimensione dell'input e i sotto-problemi sono INDIPENDENTI**

- cioè non mi devo trovare nella situazione di risolvere più volte la stessa istanza di problema

**Quando ciò non è garantito si usano altre strategie, come:**

- Programmazione dinamica
- Golosa (greedy)

**Tipicamente per problemi di ottimizzazione**



# Tecnica golosa (o greedy)

Per alcuni problemi di ottimizzazione la strategia dinamica può risultare onerosa perché risolve tutti i sottoproblemi in modo esaustivo.

Può in alcuni casi essere conveniente optare per una strategia greedy.



# Algoritmi golosi

Strategia “euristica” top-down a problemi di ottimizzazione

- Idea: per trovare una soluzione globalmente ottima, scegli ripetutamente soluzioni ottime localmente
- In questo modo per alcuni problemi si ottiene una soluzione globalmente ottima



# Proprietà della strategia golosa

Sottostruttura ottima: Ogni soluzione ottima non elementare si compone di soluzioni ottime di sottoproblemi

Proprietà della scelta golosa: La scelta ottima localmente (golosa) non pregiudica la possibilità di arrivare ad una soluzione globalmente ottima

# Elementi della strategia golosa (1/2)

- La tecnica greedy è usata per risolvere problemi di ottimizzazione
  - Vogliamo trovare la **migliore soluzione** a un problema (e.g., il cammino più corto per andare da Napoli a Torino)

## INGREDIENTI:

1. Insieme di **candidati** (e.g., città)
2. Insieme dei candidati già esaminati
3. **Funzione obiettivo** da minimizzare o massimizzare (e.g., lunghezza del cammino da Napoli a Torino)

# Elementi della strategia golosa (2/2)

4. Funzione **ammissibile**: verifica se un insieme di candidati rappresenta **una soluzione**, anche non ottima (un insieme di città è un cammino da Napoli a Torino?)
5. Funzione **ottimo**: verifica se un insieme di candidati rappresenta **una soluzione ottima** (un insieme di città è il più breve cammino da Napoli a Torino?)
6. Funzione **seleziona**: indica quale dei candidati non ancora esaminati è al momento il più promettente (**scelta greedy**)

# Pseudo codice generico per la tecnica golosa

**algoritmo** **paradigmaGreedy**(insieme di candidati  $C$ )  $\rightarrow$  soluzione

$S \leftarrow \emptyset$

**while** ( (not ottimo( $S$ )) **and** ( $C \neq \emptyset$ ) ) **do**

$x \leftarrow \text{seleziona}(C)$

$C \leftarrow C - \{x\}$

**if** ( ammissibile( $S \cup \{x\}$ ) ) **then**  $S \leftarrow S \cup \{x\}$

**if** ( ottimo( $S$ ) ) **then return**  $S$

**else errore** non ho trovato soluzioni

Perché goloso? Perché sceglie sempre il candidato più promettente!

# Un problema di sequenziamento

- Un server (e.g., una CPU, o un impiegato dell’ufficio postale) deve servire **n clienti** ( $n$  è fissato!)
- Il servizio richiesto dall’ $i$ -esimo cliente richiede  $t_i$  secondi per essere eseguito (**tempo di servizio**)
- Chiamiamo  $T(i)$  il tempo di attesa del cliente  $i$
- Vogliamo **minimizzare il tempo di attesa medio**, i.e.:

$$T_{avg} = \frac{T}{n} = \frac{1}{n} \sum_{i=1}^n T(i) \quad \text{e quindi}$$

$$T = \sum_{i=1}^n T(i)$$

# Esempio

$$t_1 = 50 \text{ msec}, \quad t_2 = 100 \text{ msec}, \quad t_3 = 3 \text{ msec}$$

Dei sei possibili ordinamenti, il migliore è evidenziato in arancione

| <i>Ordine</i> | <i>T</i>                            |      |   |          |
|---------------|-------------------------------------|------|---|----------|
| 1 2 3         | $50 + (50 + 100) + (50 + 100 + 3)$  | msec | = | 353 msec |
| 1 3 2         | $50 + (50 + 3) + (50 + 3 + 100)$    | msec | = | 256 msec |
| 2 1 3         | $100 + (100 + 50) + (100 + 50 + 3)$ | msec | = | 403 msec |
| 2 3 1         | $100 + (100 + 3) + (100 + 3 + 50)$  | msec | = | 356 msec |
| 3 1 2         | $3 + (3 + 50) + (3 + 50 + 100)$     | msec | = | 209 msec |
| 3 2 1         | $3 + (3 + 100) + (3 + 100 + 50)$    | msec | = | 259 msec |

# Un algoritmo goloso

- Il seguente algoritmo genera l'ordine di servizio in maniera incrementale secondo una strategia greedy
- Supponiamo di aver deciso di sequenziare i clienti  $i_1, i_2, \dots, i_m$ . Se adesso decidiamo di servire il cliente  $j$ , il tempo totale di servizio diventa:

$$t_{i_1} + t_{i_2} + \dots + t_{i_m} + t_j$$

- **Scelta greedy:** al generico passo  $j$  l'algoritmo goloso serve la richiesta più breve tra quelle rimanenti

# Pseudocodice

**algoritmo** sequenziamento(array C)->soluzione

S={ }

C = tempi (durata)  $t_j$  per  $j=1..n$  dei servizi richiesti

Ordina C in modo non decrescente

**for**  $j=1$  **to** n **do**

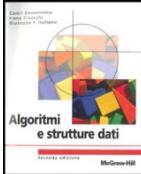
    S = S U {j}

**return** S

- Tempo di esecuzione:  $O(n \log n)$ , per ordinare le richieste

# Il problema del distributore automatico di resto

- Una strategia greedy non sempre garantisce l'ottimalità della soluzione prodotta
- Come esempio, consideriamo il problema dei distributori automatici di restituire un certo resto  $R$  usando il minor numero di monete
- Supponiamo di avere un certo numero di monete da 1, 5, 10, 20 e 50 centesimi di euro



# Il problema del distributore automatico di resto

## Formulazione del problema con strategia greedy:

- Insieme  $C$  dei candidati: insieme finito di monete da 1,5,10,20 e 50 centesimi di euro nel serbatoio
- Funzione **obiettivo**: numero di monete della soluzione
- Funzione **ammissibile**: true se il valore delle monete scelte non è superiore al resto  $R$  da restituire
- Funzione **seleziona**: la **moneta di valore più grande** tra quelle rimaste nel serbatoio  $C$
- Funzione **ottimo**: true se il valore delle monete scelte è esattamente uguale al resto

# Il problema del distributore automatico di resto

## Pseudocodice:

---

**algoritmo** distribuisceResto(*resto R*)  $\rightarrow$  soluzione

1.  $C \leftarrow$  monete contenute nel serbatoio del distributore
  2.  $S \leftarrow \emptyset$
  3. **while** (( valore( $S$ )  $\neq R$ ) and ( $C \neq \emptyset$ )) **do**
  4.      $x \leftarrow$  moneta di valore più elevato in  $C$
  5.      $C \leftarrow C - \{x\}$
  6.     **if** ( valore( $S \cup \{x\}$ )  $\leq R$  ) **then**  $S \leftarrow S \cup \{x\}$
  7.     **if** ( valore( $S$ ) =  $R$  ) **then return**  $S$  come resto esatto
  8.     **else return**  $S$  come resto parziale
-

# Il problema del distributore automatico di resto

- Non sempre l'algoritmo **distribuisciResto** è in grado di restituire il resto esatto.
- Esempio positivo:
  - $C=\{50,50,20\}$  ed  $R=70$ :  $S=\{50,20\}$   $\text{valore}(S)=70=R$
- Esempio negativo:
  - $C=\{50,20,20,20,5\}$  ed  $R=65$ :  $S=\{50,5\}$   $\text{valore}(S)=55$   
L'errore sta al primo passo: viene fatta la scelta sbagliata  $x=50$  che non potrà mai essere disfatta; non utilizzando invece la moneta da 50, potremmo restituire il resto esatto

# Esercizio - ascensore

- Un gruppo di  $n > 0$  persone deve salire su un ascensore che può sostenere un peso massimo di  $C$  kg. Indichiamo con  $p[1], \dots, p[n]$  i pesi (in kg) delle  $n$  persone. Il vettore non è ordinato e i pesi sono numeri interi.
- 1. Scrivere un algoritmo *greedy* che restituisce il **numero max di persone** che possono salire contemporaneamente senza superare la capacità  $C$  dell'ascensore.
- 2. Determinare il costo computazionale dell'algoritmo di cui al punto 1.
- 3. Raffinare l'algoritmo 1. per individuare anche le persone che faranno parte del sottogruppo che sale.

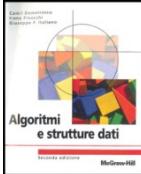


# Svolgimento 1. e 2.

- Selezione golosa: prima le persone che pesano meno e non superano la portata C dell'ascensore

```
Algoritmo ASCENSOREGREEDY(int C, array p[1..n] di int) → int
 ORDINACRESCENTE(p);
 int i := 1;
 while(i≤n && p[i] ≤ C) do
 C := C - p[i];
 i := i+1;
 endwhile
 return i-1;
```

- Il costo dell'algoritmo è dominato dal costo dell'operazione di ordinamento, cioè  $O(n \log n)$  usando un algoritmo ottimale basato su confronti
  - è possibile ordinare in tempo lineare usando un algoritmo che sfrutta il fatto che i pesi sono interi



# Riepilogo

Tre utili tecniche algoritmiche:

- **Divide et impera**: altre applicazioni nella ricerca binaria, mergesort, quicksort, moltiplicazione di matrici
- **Programmazione dinamica**: altre applicazioni nel calcolo dei numeri di Fibonacci, associatività del prodotto tra matrici, cammini minimi tra tutte le coppie di nodi (algoritmo di Floyd e Warshall)
- **Tecnica greedy**: altre applicazioni nel distributore automatico di resto, nel calcolo del minimo albero ricoprente (algoritmi di Prim, Kruskal, Boruvka) e dei cammini minimi a sorgente singola (algoritmo di Dijkstra)



UNIVERSITÀ  
DEGLI STUDI  
DI BERGAMO

Dipartimento  
di Ingegneria Gestionale,  
dell'Informazione e della Produzione



# Tabelle Hash nel JCF

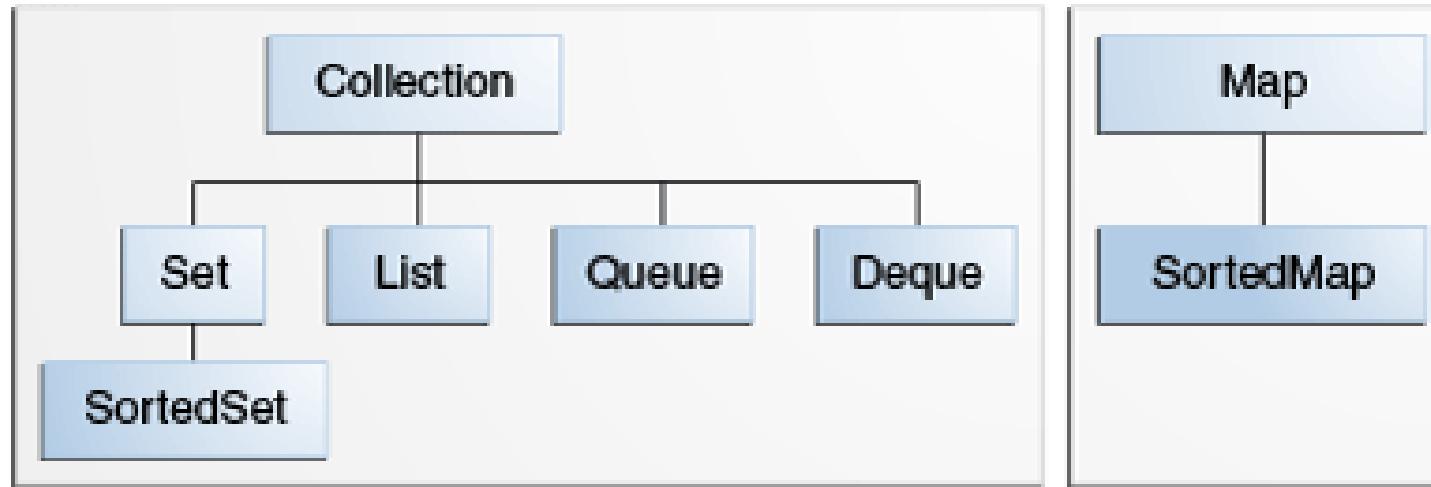
Corso di laurea  
**Magistrale in  
Ingegneria  
Informatica**

PROGETTAZIONE, ALGORITMI E  
COMPUTABILITÀ  
(38090-MOD1)

RELATORE  
Prof.ssa Patrizia  
Scandurra

SEDE  
DIGIP

# Recap JCF - interfacce



- L'interfaccia **Map** è l'interfaccia *Dizionario* che rappresenta una collezione di coppie (*key, value*):
  - *key* univoca
  - *value* (non necessariamente univoca)



# L'interfaccia Map

Map definisce i metodi:

- `put(k,v)` – inserire una coppia (k,v)
- `remove(k)` – rimuovere l'elemento di chiave k
- `containsKey(k)` – ricerca in base alla chiave k
- `containsValue(v)` – ricercare in base al valore v
  - Ci può essere più di una coppia con chiave distinta
- `size()`, `equals()`, `clear()`



UNIVERSITÀ  
DEGLI STUDI  
DI BERGAMO

Dipartimento  
di Ingegneria Gestionale,  
dell'Informazione e della Produzione

# **Hash tables in the JCF:**

## **Hashtable**

## **HashMap**

## **HashSet**

# Hashtable in JCF

- **Example:** a hashtable of numbers, using the names of the numbers as keys:
- ```
Hashtable<String, Integer> numbers
        = new Hashtable<String, Integer>();
numbers.put("one", 1);
numbers.put("two", 2);
numbers.put("three", 3);
Integer n = numbers.get("two");
if (n != null) { System.out.println("two = " + n); }
```

Hashtable (1/2)

- An instance of **Hashtable** has two parameters that affect its performance: initial capacity and *load factor*.
- Generally, the **default load factor (.75)** offers a good tradeoff between time and space costs.
- Higher values decrease the space overhead but increase the time cost to look up an entry.
- Constructors:
 - `Hashtable()`: Constructs a new, empty hashtable with a **default initial capacity (11)** and **load factor (0.75)**.
 - `Hashtable(int initialCapacity)`: Constructs a new, empty hashtable with the specified initial capacity and default load factor (0.75).
 - `Hashtable(int initialCapacity, float loadFactor)`: Constructs a new, empty hashtable with the specified initial capacity and the specified load factor.⁶

Hashtable (2/2)

- Hashtable basically retains values of key-value pair.
- **It is synchronized**. So it comes with its cost. Only one thread can access in one time
- **It didn't allow null for both key and value**. You will get NullPointerException if you add null value.

```
Hashtable<Integer,String> cityTable =  
    new Hashtable<Integer,String>();  
cityTable.put(1, "Lahore");  
cityTable.put(2, "Karachi");  
cityTable.put(3, null); /* NullPointerException at runtime */  
System.out.println(cityTable.get(1));  
System.out.println(cityTable.get(2));  
System.out.println(cityTable.get(3));
```

HashMap

- The default **initial capacity is 16** and the default **load factor is 0.75**
- It **allows null for both key and value**
- It is **unsynchronized**. So come up with better performance. If required, it must be synchronized externally.

```
HashMap<Integer,String> productMap = new  
    HashMap<Integer,String>();  
  
productMap.put(1, "Keys");  
productMap.put(2, null);  
//At creation time, to prevent accidental  
//unsynchronized access to the map  
  
Map m = Collections.synchronizedMap(new  
    HashMap(...));8
```

HashSet

- HashSet **does not allow duplicate values**
 - can be used where you want to maintain a unique list
- Default **initial capacity is 16** and default **load factor is 0.75**.
- It provides **add method** rather put method (it **implements Set<E>**)
- You also use its **contains method** to check whether the object is already available in the HashSet.
- **It is unsynchronized**

```
HashSet<String> stateSet = new HashSet<String>();  
stateSet.add ("CA"); stateSet.add ("WI"); stateSet.add ("NY");  
if (stateSet.contains("PB")) /* if already inside, we do not add it but  
shows following message*/  
    System.out.println("Already found");  
else    stateSet.add("PB");  
// At creation time, to prevent accidental unsynchronized access  
Set s = Collections.synchronizedSet(new HashSet(...));  
9
```

Esercizio

- **Conta frequenza.**

Implementare un algoritmo che dato in input una frase (ad esempio l'insieme delle parole digitate sulla linea di comando), restituisce le occorrenze di ciascuna parola.

- Due possibili implementazioni a seconda se ordiniamo o no l'output:

```
>java ContaFrequenza cane gatto cane gatto gatto cane pesce  
3 parole distinte: {cane=3, pesce=1, gatto=3}
```

```
>java ContaFrequenzaOrd cane gatto cane gatto gatto cane pesce  
3 parole distinte: {cane=3, gatto=3, pesce=1}
```

elenco ordinato!

Soluzione esercizio

Si può scegliere:

- **HashMap**: tabella non ordinato, tempo d'accesso costante
- **TreeMap**: tabella ordinato, tempo di accesso non costante

Output con **HashMap**:

```
>java HashMapFreq cane gatto cane gatto gatto cane pesce  
3 parole distinte: {cane=3, pesce=1, gatto=3}
```

Output con **TreeMap** (*elenco ordinato*):

```
>java TreeMapFreq cane gatto cane gatto gatto cane pesce  
3 parole distinte: {cane=3, gatto=3, pesce=1}
```

Soluzioni esercizio

```
import java.util.*;
public class ContaFrequenzaOrd {
    public static void main(String args[]) {
        SortedMap m = new _____;
        for (int i=0; i<args.length; i++) {
            Integer freq = (Integer) m.get(args[i]);
            m.put(args[i], (freq==null ? new Integer(1) :
                           new Integer(freq.intValue() + 1)));
        }
        System.out.println(m.size()+" parole distinte:");
        System.out.println(m);
    }
}
```

```
>java ContaFrequenza cane gatto cane gatto gatto cane pesce
3 parole distinte: {cane=3, pesce=1, gatto=3}
```

```
>java ContaFrequenzaOrd cane gatto cane gatto gatto cane pesce
3 parole distinte: {cane=3, gatto=3, pesce=1}
```

elenco ordinato!

Equals() and hashCode()

The **java.lang.Object** has two very important methods:

- **public boolean equals(Object obj)**: returns true if and only if x and y refer to the same object (`x==y` has the value true)
- **public int hashCode()**: returns the hash code value for the object on which this method is invoked. The hash code value is an integer supported for the benefit of hashing based collection classes such as `Hashtable`, `HashMap`, e `HashSet`.

These two methods are interrelated and they should be overridden correctly:

- **override the hashCode method whenever the equals method is overridden,**
- so as to maintain the general contract for the hashCode method, which states that **equal objects must have equal hash codes**

Correct Implementation Example

```
1.  public class Test
2.  {
3.      private int num;
4.      private String data;
5.
6.      public boolean equals(Object obj)
7.      {
8.          if(this == obj)
9.              return true;
10.         if((obj == null) || (obj.getClass() != this.getClass()))
11.             return false;
12.         // object must be Test at this point
13.         Test test = (Test)obj;
14.         return num == test.num &&
15.             (data == test.data || (data != null && data.equals(test.data)));
16.     }
17.
18.     public int hashCode()
19.     {
20.         int hash = 7;
21.         hash = 31 * hash + num;
22.         hash = 31 * hash + (null == data ? 0 : data.hashCode());
23.         return hash;
24.     }
25.
26.     // other methods
27. }
```

This implementation will ensure
that equal objects will have equal
hash codes.

Guidelines for implementing hashCode()

- Store an arbitrary non-zero constant integer value (say 7) in an int variable **hash**.
- Compute an hash code **int var_code** for each variable **var** of your object as follows:
 - If the variable(var) is byte, char, short or int: **var_code = (int)var;**
 - If the variable(var) is long: **var_code = (int)(var ^ (var >>> 32));**
 - If the variable(var) is float: **var_code = Float.floatToIntBits(var);**
 - If the variable(var) is double:
long bits = Double.doubleToLongBits(var);
var_code = (int)(bits ^ (bits >>> 32));
 - If the variable(var) is boolean: **var_code = var ? 1 : 0;**
 - If the variable(var) is an object reference:
var_code = (null == var ? 0 : var.hashCode());
- Combine this individual variable hash code **var_code** in the original hash code **hash** as follows: **hash = 31 * hash + var_code;**
- Follow these steps for all the significant variables and in the end return the resulting integer hash.
- Lastly, review your hashCode method and check if it is returning equal hash codes for equal objects.
- Also, verify that the hash codes returned for the object are consistently the same for multiple invocations during the same execution.

Ad es. $78 \ggg 2$ sarebbe:
1001110 >>> 2 = 0010011
ovvero 19 in decimale

ESERCIZIO

Definire e testare un componente in Java **FinancialHistory**:

- **Attributi (privati):**

- float cashOnHand = la cifra presente sul conto

- Hashtable incomes = le entrate relative a certe voci (stringhe) di spesa

- Hashtable expenditures = le uscite relative a certe voci (stringhe) di spesa

- Nello stesso file definire due **classi per le eccezioni**:

- NegAmountException** e **NegCashException**.

- **Costruttore della classe FinancialHistory:**

- FinancialHistory(float amount)** -- prende come argomento la cifra iniziale da mettere sul conto, che deve essere ≥ 0 altrimenti solleva eccezione negAmountException

- **Implementa un'interfaccia FinancialHistoryIF con i metodi:**

- **float cashOnHand()** -- ritorna la cifra presente sul conto
 - **float receivedFrom(String s)** -- se la stringa corrisponde alla descrizione di un'entrata, restituisce la cifra entrata
 - **float spentFor(String s)** -- se la stringa corrisponde alla descrizione di un'uscita, restituisce la cifra uscita

- **void receiveFrom(float amount, String s)** -- registra una nuova entrata con cifra amount e stringa di descrizione s, modifica sia la tabella hash incomes che la cifra corrente sul conto (cashOnHand) aggiungendo amount
- **void spendFor(float amount, String s)** -- registra una nuova uscita con cifra amount e stringa di descrizione s, modifica sia la tabella hash expenditures che la cifra corrente sul conto (cashOnHand) sottraendo amount

Nota: entrambe le funzioni vogliono amount ≥ 0 altrimenti sollevano eccezione NegAmountException; la funzione spendFor controlla anche che dopo la spesa il conto non vada in rosso, e nel caso solleva eccezione NegCashException.

- **String printIncomes()** -- genera stringa con scritte tutte le entrate
- **String printExpenditures()** -- genera stringa con scritte tutte le uscite

Algoritmi e Strutture Dati

Capitolo 12 Grafi e visite di grafi

Camil Demetrescu, Irene Finocchi,
Giuseppe F. Italiano

Definizione

Struttura dati non lineare e non di tipo gerarchico

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

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

Esempio 1: $V=\{\text{persone che vivono in Italia}\}$,
 $E=\{\text{coppie di persone che si sono strette la mano}\}$

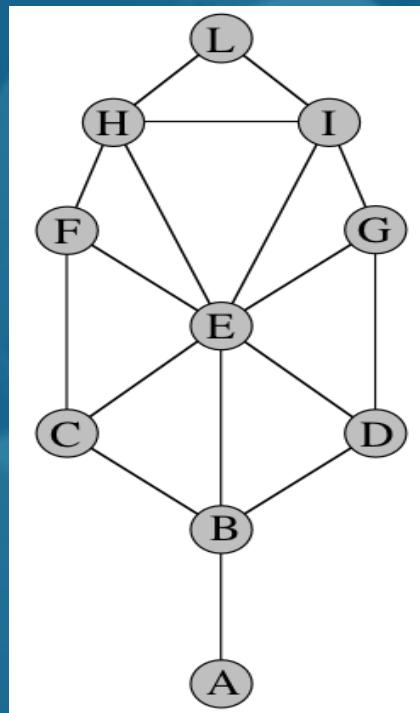
Esempio 2: $V=\{\text{persone che vivono in Italia}\}$,
 $E=\{(x,y) \text{ tale che } x \text{ ha inviato una mail a } y\}$

Tipologia di grafi

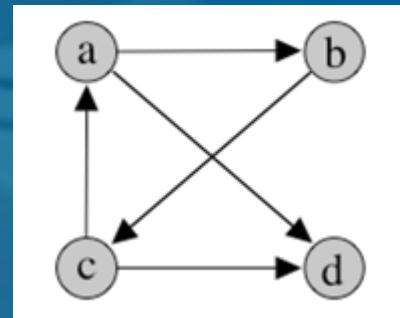
Esempio 1: E relazione simmetrica → grafo non orientato

Esempio 2: E relazione non simmetrica → grafo orientato

Grafo non orientato



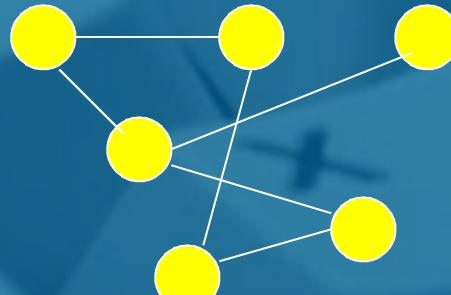
Grafo orientato



Un cappio è un arco i cui estremi coincidono.

Un grafo non orientato è semplice se **non ha cappi** e **non ci sono due archi con gli stessi estremi**. In caso contrario si parla di multigrafo.

grafo semplice



multigrafo

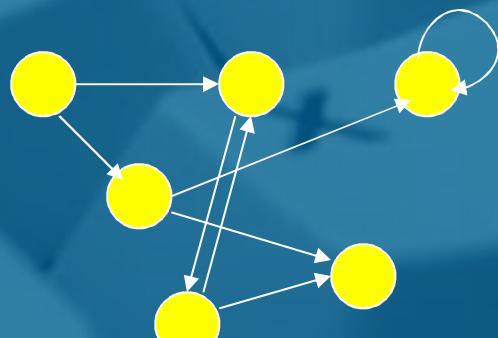


non
orientato

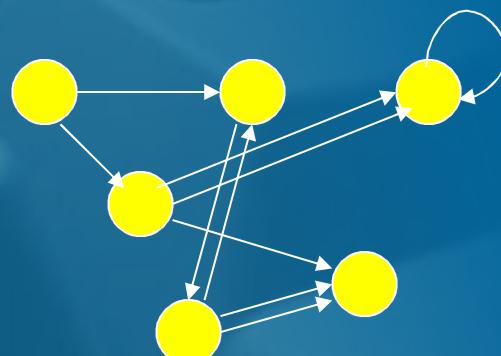
Un grafo orientato è semplice se non ci sono due archi con gli stessi estremi. In caso contrario si parla di multigrafo.

Salvo indicazione contraria noi assumeremo sempre che un grafo sia semplice.

grafo semplice



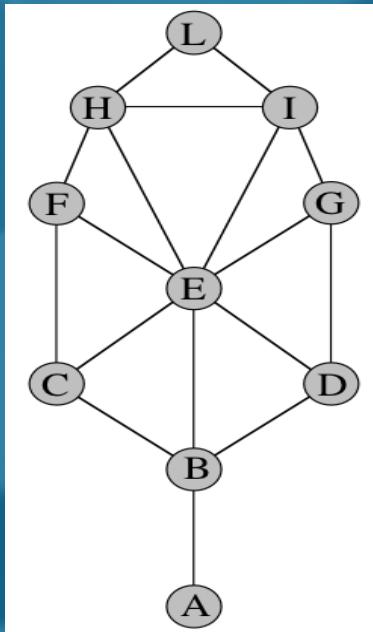
multigrafo



orientato

Terminologia: adiacenza, incidenza e grado

Grafo non orientato



L ed I sono **adiacenti** (o vicini)

(L,I) è **incidente** su L e su I

I ha **grado** δ (num. archi incidenti) 4: $\delta(I)=4$
 $n = |V|$ numero di vertici

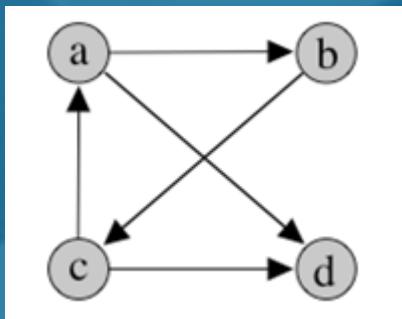
$m = |E|$ numero di archi, varia da 0 a $|V|^2 - |V|$

Sommmando i gradi di tutti i vertici contiamo ogni arco esattamente due volte, quindi:

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

Terminologia: adiacenza, incidenza e grado

Grafo orientato



b è **adiacente** (o vicino) ad a

(a,b) **esce** dal vertice a ed **entra** in b

a ha **grado** (num. archi entranti ed uscenti) 3:

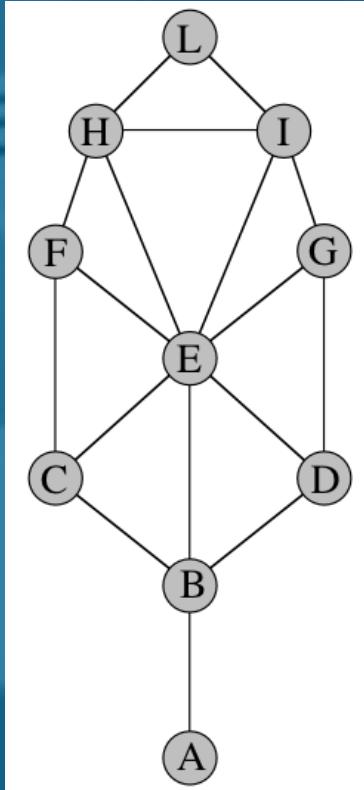
$$\delta(a) = \delta_{\text{in}}(a) + \delta_{\text{out}}(a) = 3$$

Ogni arco (u,v) è entrante per v ed uscente per u ,
quindi:

$$\sum_{v \in V} \delta_{\text{in}}(v) = \sum_{v \in V} \delta_{\text{out}}(v) = m$$

$$\sum_{v \in V} \delta(v) = \sum_{v \in V} \delta_{\text{in}}(v) + \sum_{v \in V} \delta_{\text{out}}(v) = 2m$$

Terminologia: cammini e cicli



$\langle L, I, E, C, B, A \rangle$ è un **cammino semplice** (tutti i vertici sono distinti) nel grafo di lunghezza (num. archi) 5

Non è il più corto cammino tra L ed A

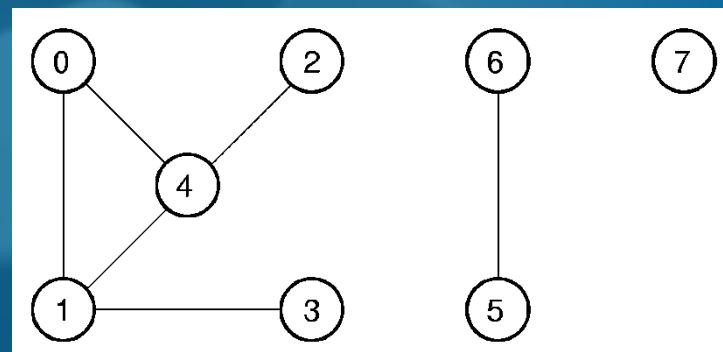
La lunghezza del più corto cammino tra due vertici si dice **distanza**: L ed A hanno distanza 4

$\langle L, H, I, L \rangle$ è un **ciclo semplice** di lunghezza 3
Un cappio (u, u) è un ciclo di lunghezza 1

Componenti connesse di un grafo non orientato

Una **componente连通子图** di G è un insieme massimale di vertici $U \subseteq V$ t.c. per ogni coppia di vertici in U esiste in G un cammino che li collega

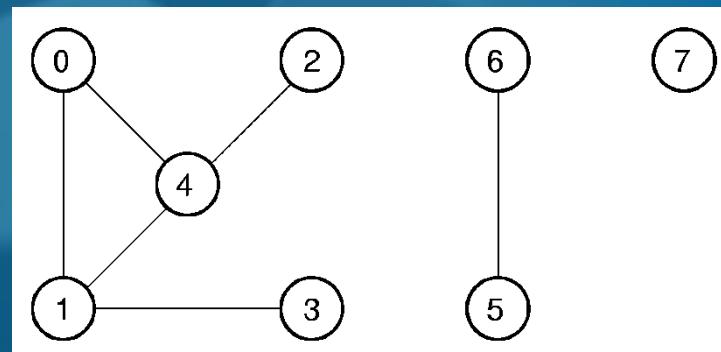
Un grafo $G=(V,E)$ non orientato è **连通** se esiste almeno un cammino tra ogni coppia di vertici; o anche se esiste un'unica componente连通子图



Componenti connesse di un grafo non orientato

E' facile verificare che la **relazione tra vertici "essere raggiungibile da"** è di equivalenza (gode della proprietà riflessiva, simmetrica e transitiva)

Ne segue che: Le componenti connesse di un grafo non orientato sono le classi di equivalenza dei suoi vertici rispetto alla relazione di raggiungibilità.

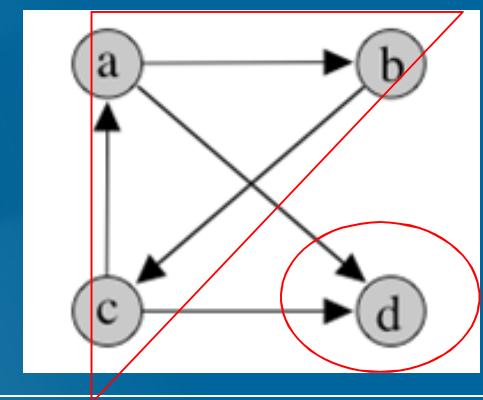


Componenti fortemente connesse di un grafo orientato

Una **componente fortemente connessa** di G è un insieme massimale di vertici $U \subseteq V$ t.c. per ogni coppia di vertici u e v in U esiste in G un cammino orientato da u a v e da v ad u che li collega

Un grafo $G=(V,E)$ orientato è **fortemente连通** se esiste almeno un cammino (orientato) tra ogni coppia di vertici

Le componenti fortemente connesse sono le classi di equivalenza dei vertici rispetto alla relazione di equivalenza «di connettività forte»





Grafi, alberi e DAG

- Gli alberi sono casi particolari di grafi
- Un albero è un **grafo non orientato, connesso e aciclico** (senza cicli)
 - Con esattamente $|V|-1$ archi [dimostrabile]
- Un **DAG** (direct acyclic graph) è un grafo orientato ed aciclico



Strutture dati per rappresentare grafi

Tipo grafo in Java

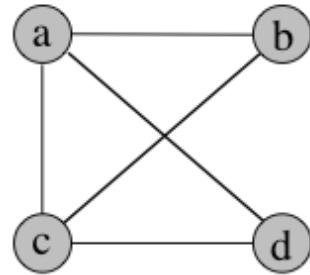
```
interface Grafo {                                // Graph ADT
    public int n();                            // num. vertici
    public int m();                            // num. di archi
    public int grado(Vertice v);
    public Arco[] archiIncidenti(Vertice v);
    public Arco sonoAdiacenti(Vertice x,
                             Vertice y);
    public void aggiungiArco(Vertice x, Vertice
                           y, int weight);
    public void aggiungiVertice(Vertice v);
    public void cancellaArco(Arco e);
    public void cancellaVertice(Vertice v);
    //ecc...
}
```



Librerie Java per grafi

- JGraphT <https://jgrapht.org/>
 - Per gli algoritmi!
 - Usa Jgraph come UI
- JDSL (Data Structures Library in Java)
<http://128.148.32.110/cgc/jdsl/>
- Annas <http://code.google.com/p/annas/>
- JGraphX - Java Swing graph visualization library
<https://github.com/jgraph/jgraphx>
 - Utile come debugging dei vostri algoritmi
- Altre API alternative: Jung, yworks, prefuse.org e BFG

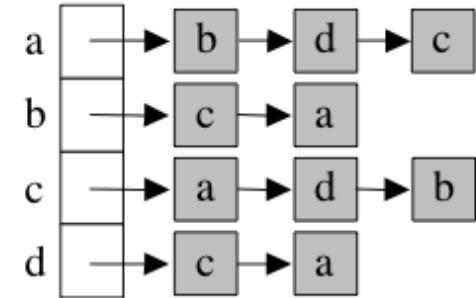
Grafi non orientati



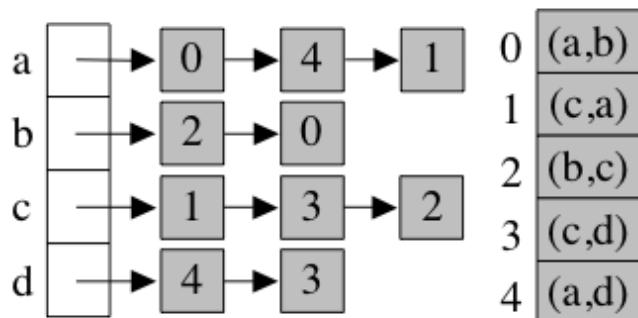
(a) Grafo non orientato G

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

(b) Lista di archi di G
 $O(m+n)$



(c) Liste di adiacenza di G
 $O(m+n)$



(d) Liste di incidenza di G
 $O(m+n)$

Limiti di spazio

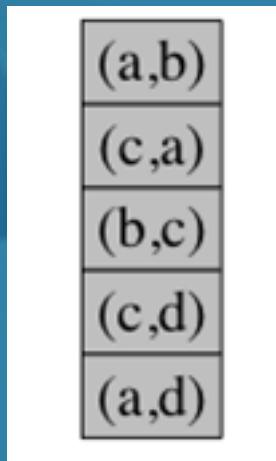
a	b	c	d
0	(a,b)		
1	(c,a)		
2	(b,c)		
3	(c,d)		
4	(a,d)		

(e) Matrice di adiacenza di G
 $O(n^2)$

	(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

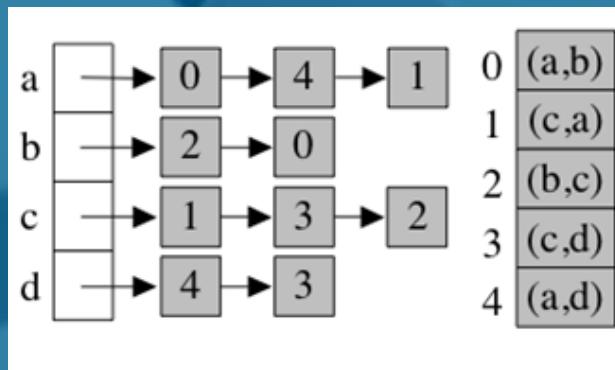
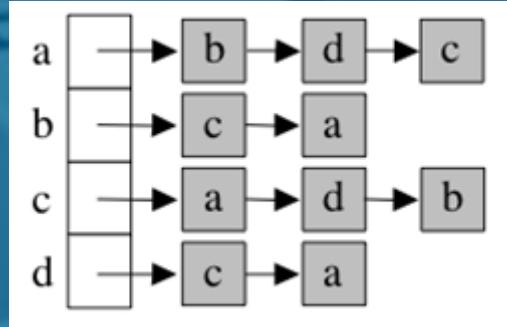
(f) Matrice di incidenza di G
 $O(n \cdot m) = O(n^3)$

Prestazioni della lista di archi



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

Prestazioni delle liste di adiacenza e delle liste di incidenza



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>	$O(\delta(x) + \delta(y))$

Prestazioni della matrice di adiacenza

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

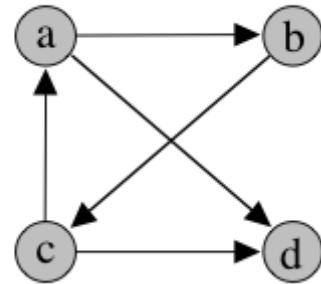
Operazione	Tempo di esecuzione
$\text{grado}(v)$	$O(n)$
$\text{archiIncidenti}(v)$	$O(n)$
$\text{sonoAdiacenti}(x, y)$	$O(1)$
$\text{aggiungiVertice}(v)$	$O(n^2)$
$\text{aggiungiArco}(x, y)$	$O(1)$
$\text{rimuoviVertice}(v)$	$O(n^2)$
$\text{rimuoviArco}(e)$	$O(1)$

Prestazioni della matrice 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

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

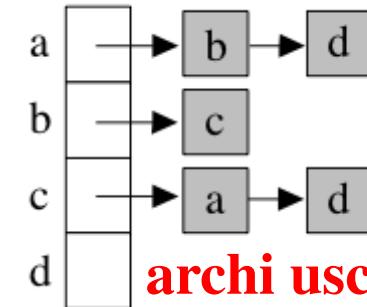
Grafi orientati



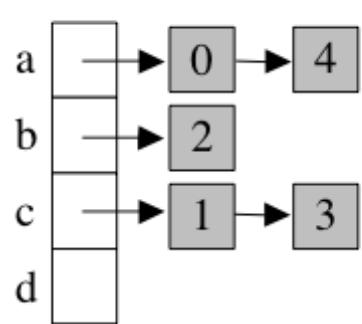
(a) Grafo orientato G

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

(e) Matrice di adiacenza di G

a	(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** a partire da un vertice sorgente s
- Problema di base in molte applicazioni
- Esistono vari tipi di visite con diverse proprietà: in particolare, **visita in ampiezza** (BFS=breadth first search) e **visita in profondità** (DFS=depth first search)



Osservazioni

- Un vertice viene **marcato** quando viene incontrato per la prima volta: la marcatura può essere mantenuta tramite un vettore di bit di marcatura
- La visita genera un **albero di copertura** T del grafo radicato in s
- Un insieme di vertici $F \subseteq T$ mantiene la **frangia** di T :
 - $v \in F$: v è **aperto**, esistono archi incidenti su v non ancora esaminati
 - $v \in T - F$: v è **chiuso**, tutti gli archi incidenti su v sono stati esaminati



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)

Costo della visita

Il tempo di esecuzione dipende dalla struttura dati usata:

- Lista di archi: $O(mn)$
 - Invochiamo l'operazione **archiIncidenti(v)**, che richiede l'esame di tutti gli archi del grafo, per ogni vertice v
- Liste di adiacenza o di incidenza: $O(m+n)$
 - La somma su tutti i vertici delle lunghezze delle liste di adiacenza è $2m$
- Matrice di adiacenza: $O(n^2)$
 - L'operazione **archiIncidenti(v)** richiede la scansione di una intera riga e quindi un tempo $O(n)$ per ogni vertice v
- Matrice di incidenza: $O(mn)$
 - L'operazione **archiIncidenti(v)** richiede $O(m)$ per ogni vertice v



Visita in ampiezza



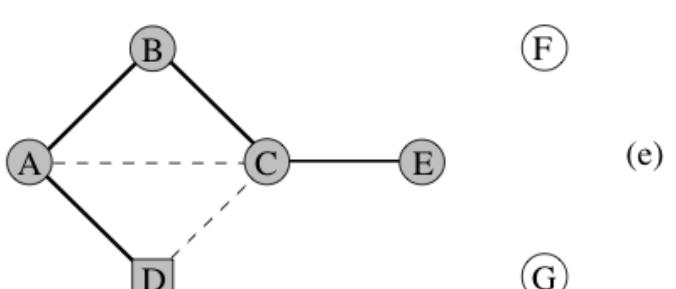
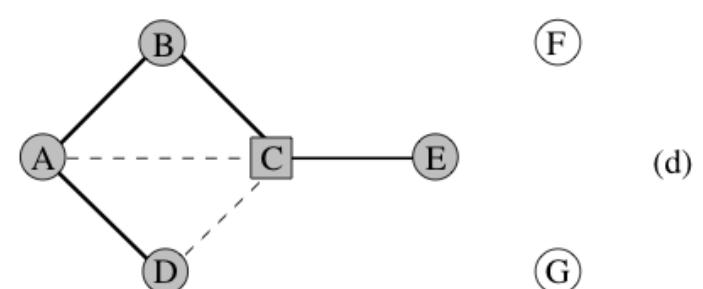
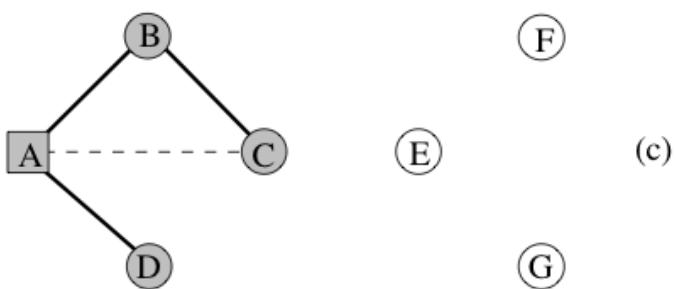
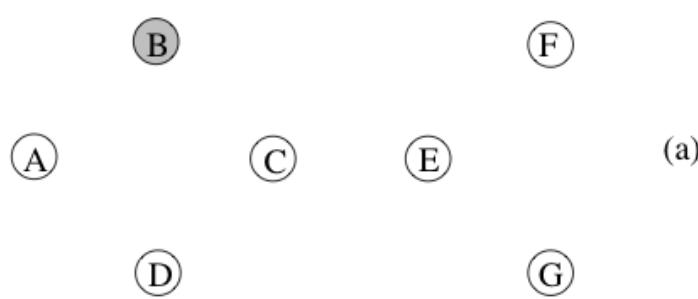
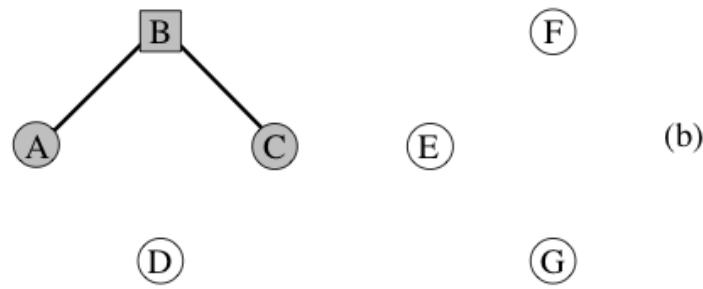
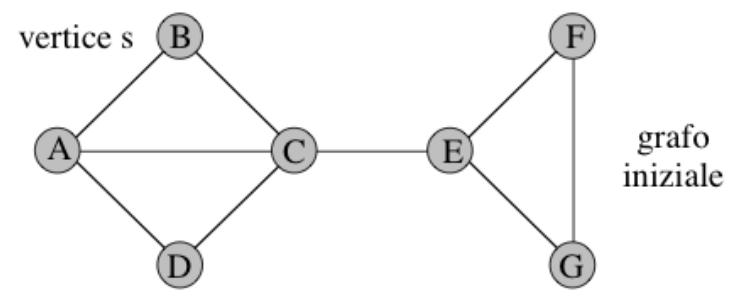
Visita in ampiezza

algoritmo visitaBFS(*vertice s*) → *albero*

1. rendi tutti i vertici non marcati
2. $T \leftarrow$ albero formato da un solo nodo s
3. **Coda F**
4. marca il vertice s
5. **F.enqueue(*s*)**
6. **while (not F.isEmpty()) do**
 7. $u \leftarrow F.dequeue()$
 8. **for each (arco (u, v) in G) do**
 9. **if (v non è ancora marcato) then**
 10. **F.enqueue(*v*)**
 11. marca il vertice v
 12. rendi u padre di v in T
 13. **return T**

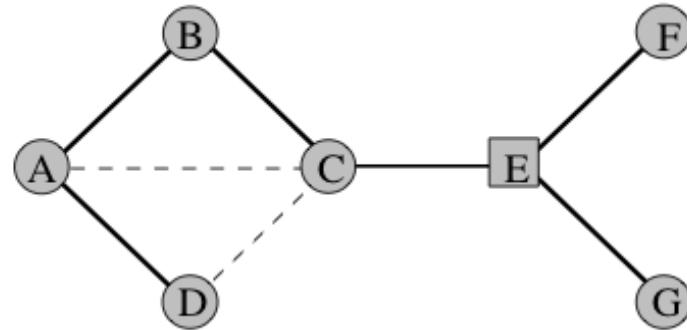
- Gli archi incidenti in v possono essere esaminati in qualsiasi ordine
- Nell'albero BFS, ogni vertice si trova il più vicino possibile alla radice s .

Esempio: grafo non orientato (1/2)

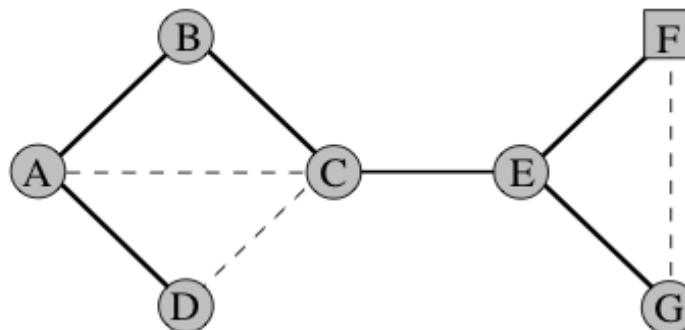


Vertici adiacenti presi secondo ordine lessicografico

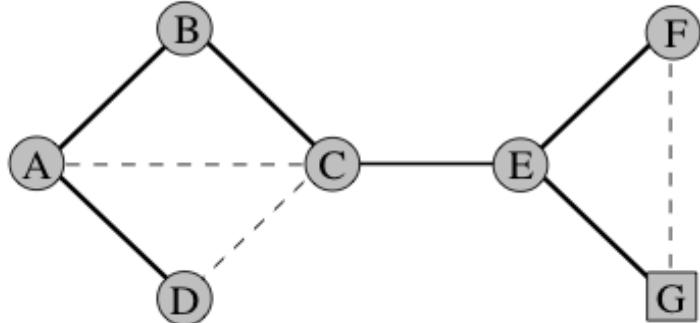
Esempio: grafo non orientato (2/2)



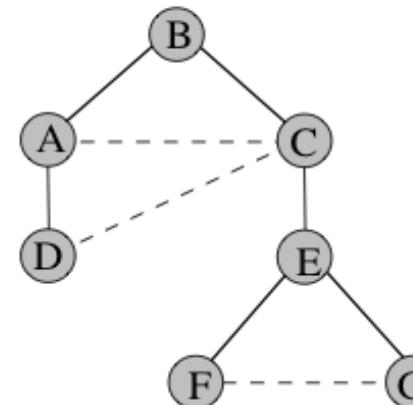
(f)



(g)



(h)



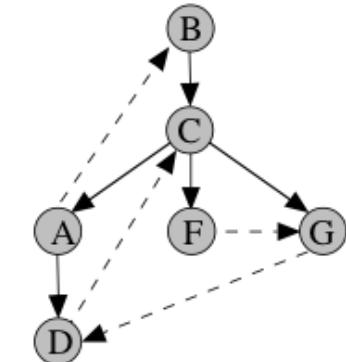
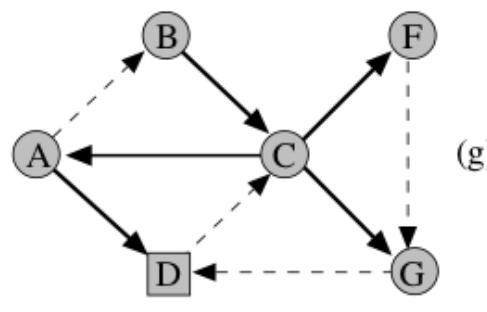
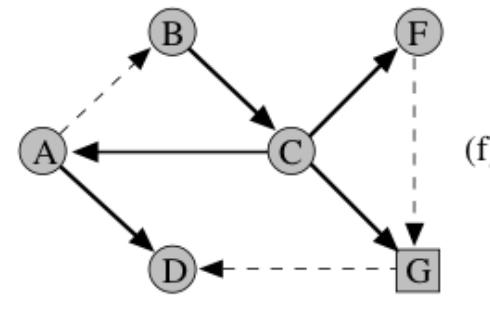
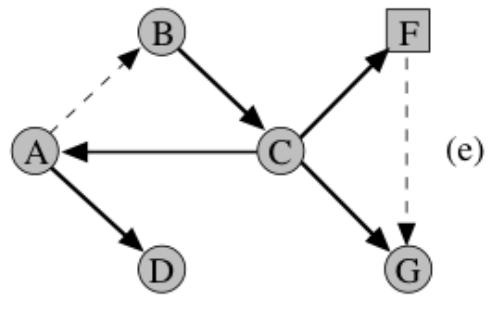
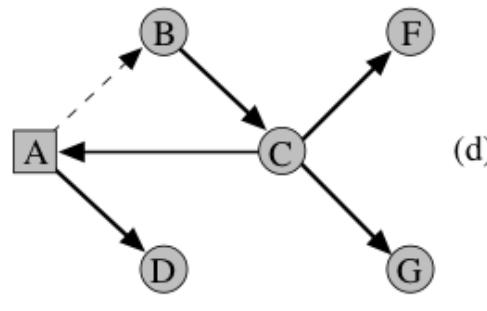
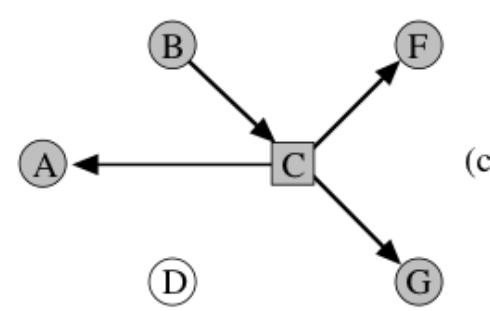
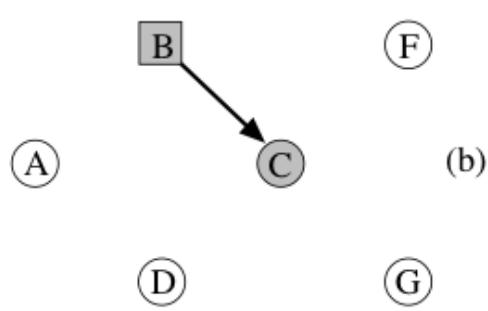
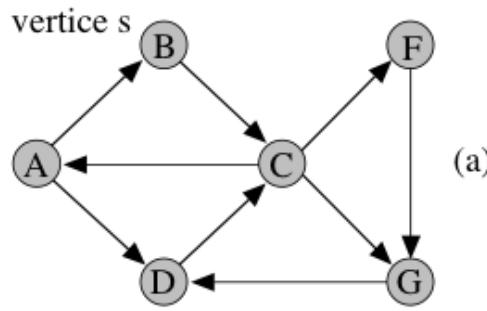
albero
BFS

Archi **dell'albero**: ad es. (B,A)

Archi tra vertici dello **stesso livello**: ad es. (A,C)

Archi **tra livelli consecutivi**: (C,D)

Esempio: grafo orientato



archi dell'albero: (B,C)

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)



Proprietà

- Per ogni nodo v , il **livello** di v nell'albero BFS è pari alla **distanza** di v dalla sorgente s
- Per ogni arco (u,v) di un **grafo non orientato**, gli estremi u e v appartengono allo stesso livello o a livelli consecutivi dell'albero BFS
- Se il **grafo è orientato**, possono esistere archi (u,v) che attraversano all'indietro più di un livello



Visita in profondità

Visita in profondità

procedura visitaDFSRicorsiva(vertice v , albero T)

1. *marca e visita il vertice v*
2. **for each** (arco (v, w)) **do**
3. **if** (w non è marcato) **then**
4. aggiungi l'arco (v, w) all'albero T
5. visitaDFSRicorsiva(w, T)

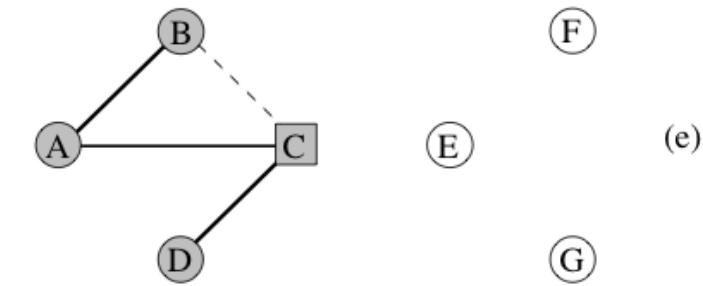
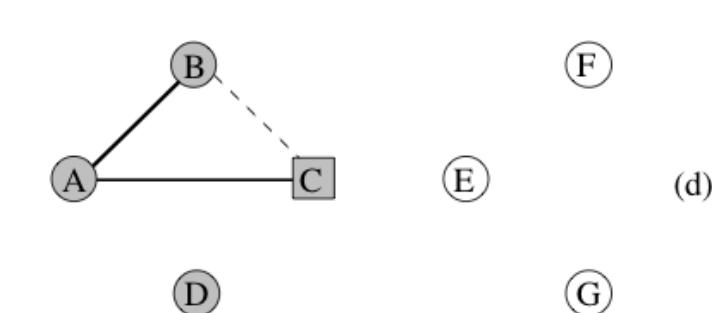
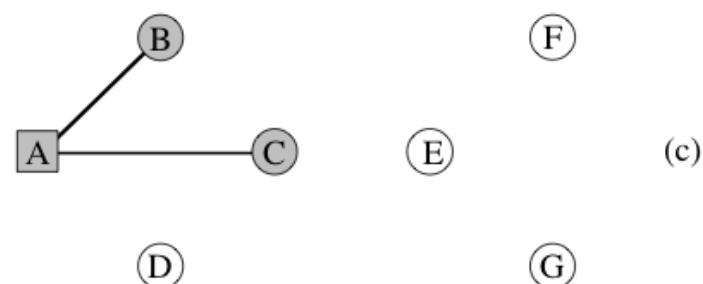
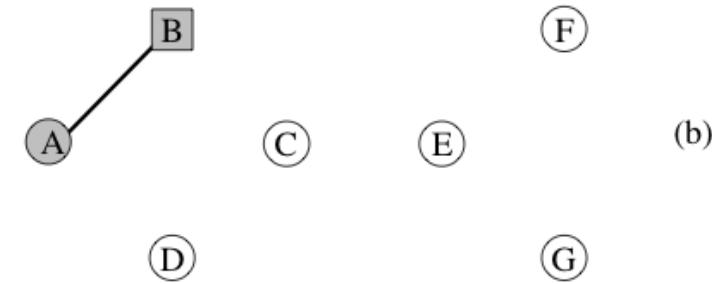
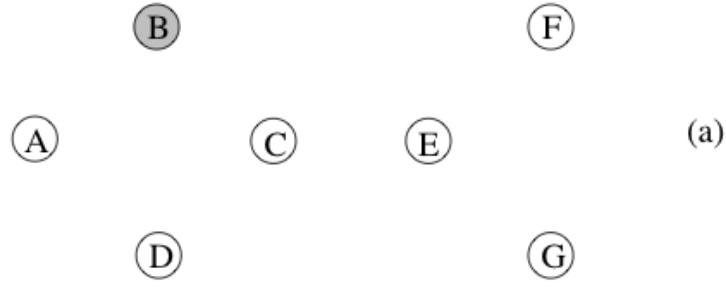
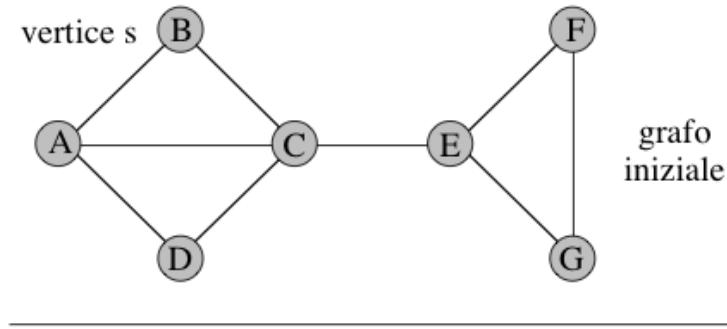
algoritmo visitaDFS(vertice s) → albero

6. $T \leftarrow$ albero vuoto
7. visitaDFSRicorsiva(s, T)
8. **return** T

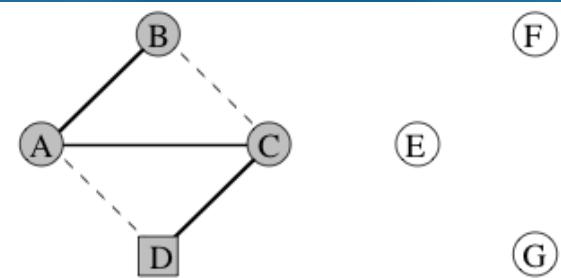
Complessità $O(m+n)$ con liste di adiacenza

- visitaDFSRicorsiva è chiamata esattamente una volta per ogni vertice ed il ciclo 2-5 viene eseguito $|adj(v)|$ volte, per cui sommando su tutti i vertici: $O(m)$
- Inizializzazione sui vertici: $O(m+n)$.

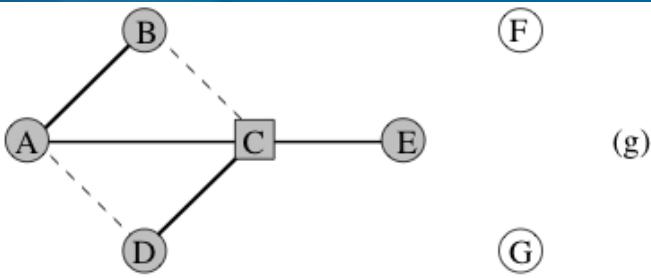
Esempio: grafo non orientato (1/2)



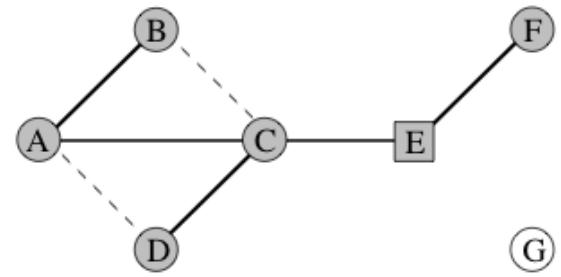
Esempio: grafo non orientato (2/2)



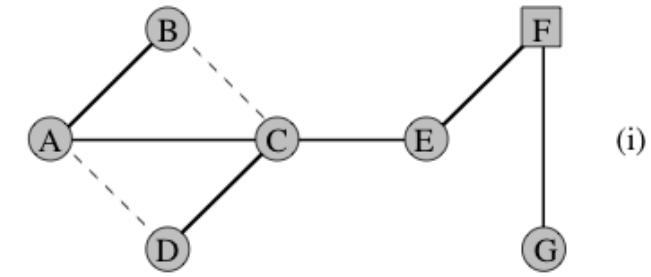
(f)



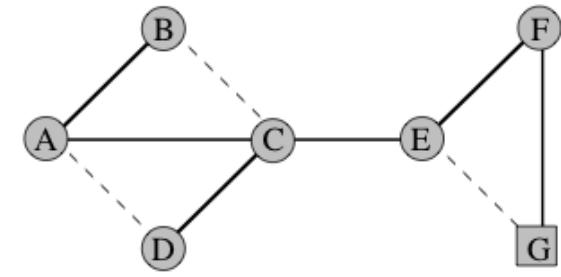
(g)



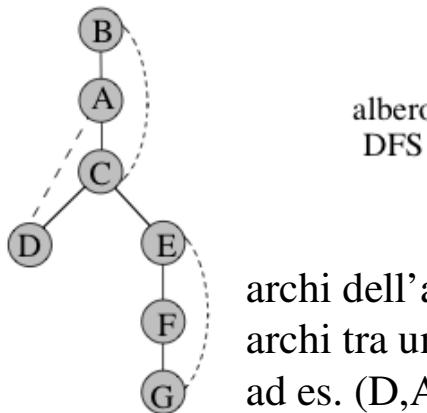
(h)



(i)



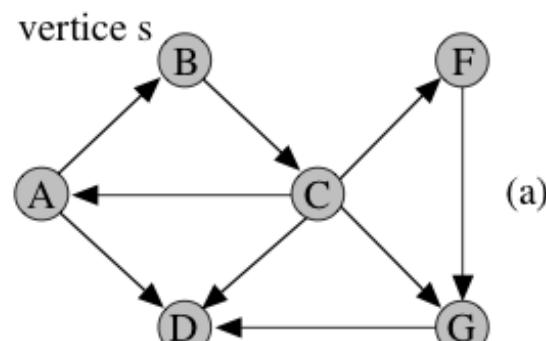
(j)



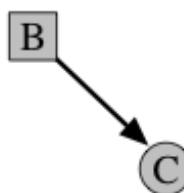
albero
DFS

archi dell'albero: ad es. (B,A)
archi tra un antenato ed un discendente:
ad es. (D,A)

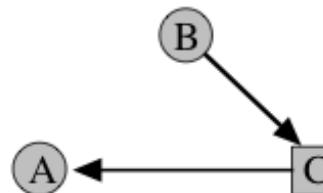
Esempio: grafo orientato (1/2)



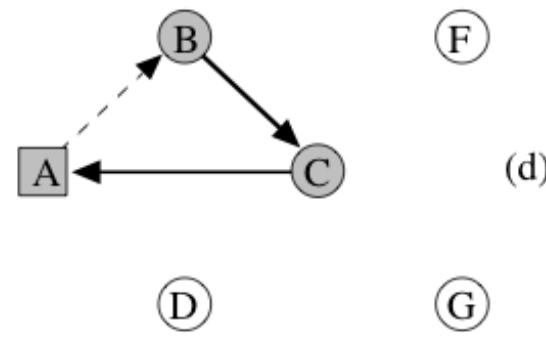
(a)



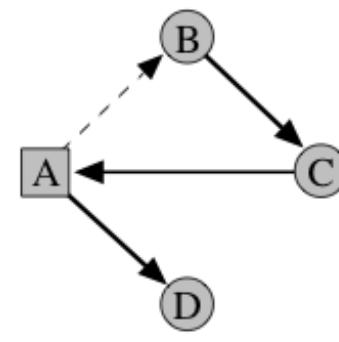
(b)



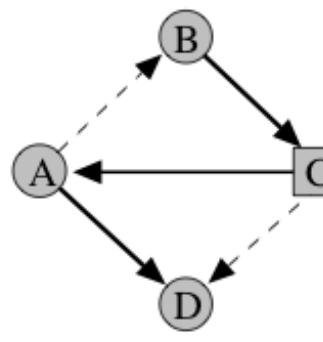
(c)



(d)

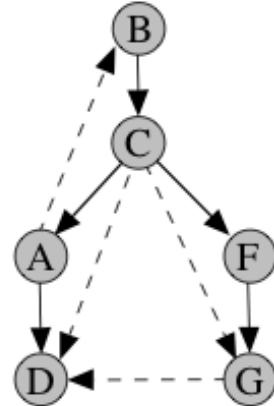
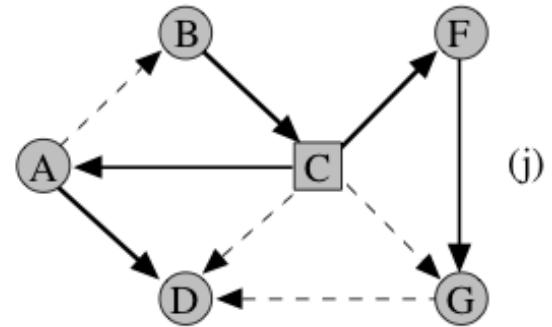
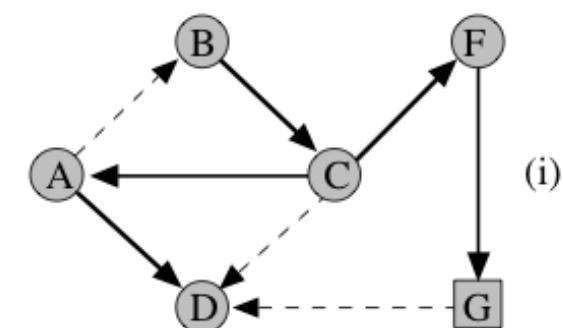
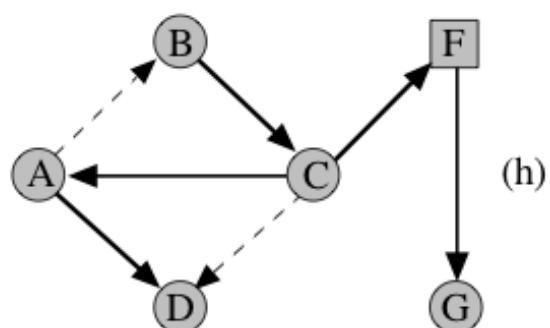
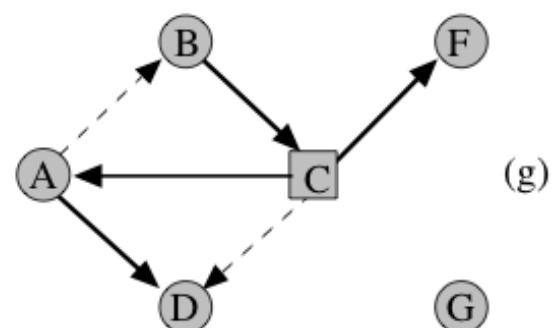


(e)



(f)

Esempio: grafo orientato (2/2)



archi dell'albero: (B,C)
archi in avanti: (C,D) e (C,G)
archi all'indietro: (A,B)
archi trasversali a sinistra: (G,D)



Proprietà

- Sia (u,v) un arco di **un grafo non orientato**. Allora:
 - (u,v) è un **arco dell'albero** DFS, oppure
 - i nodi u e v sono l'uno discendente/antenato dell'altro
oppure: archi in avanti e archi all'indietro
- Sia (u,v) un arco di **un grafo orientato**. Allora:
 - (u,v) è un **arco dell'albero** DFS, oppure
 - i nodi u e v sono l'uno discendente/antenato dell'altro,
oppure: archi in avanti e archi all'indietro
 - (u,v) è un arco **trasversale a sinistra**, ovvero il vertice v è in un sottoalbero visitato precedentemente ad u

Visita in profondità

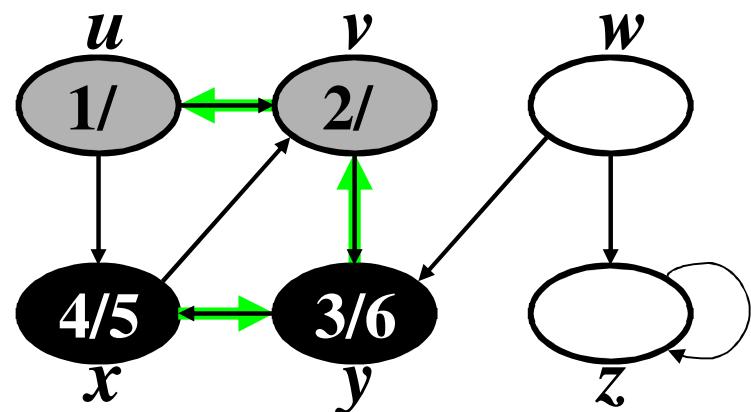
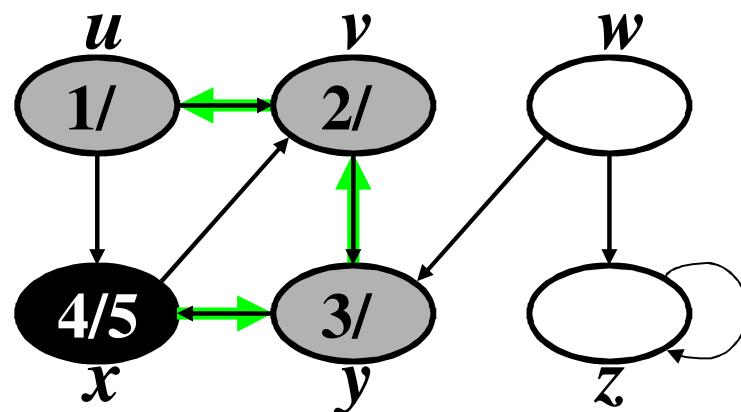
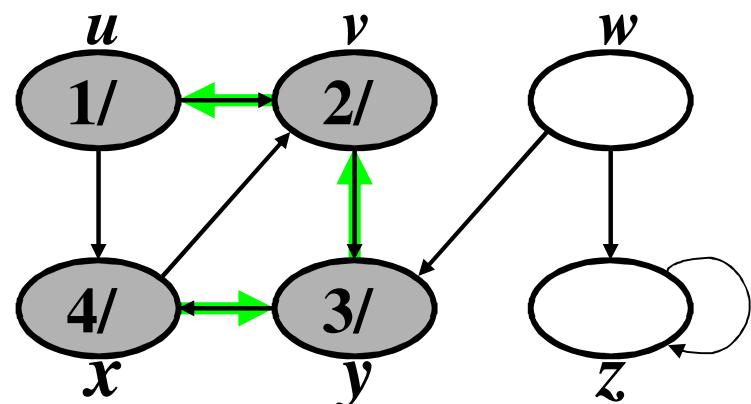
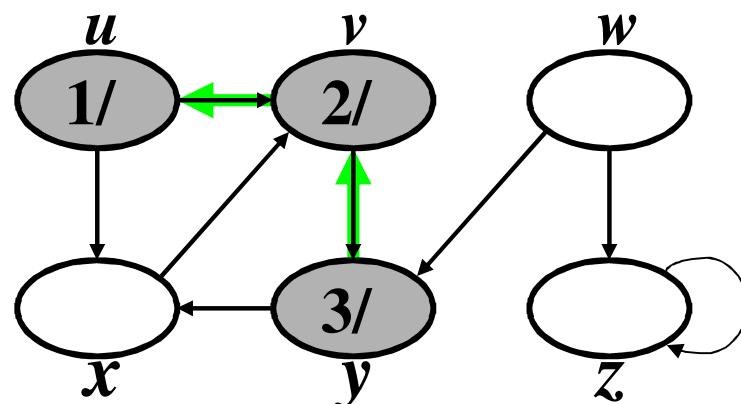
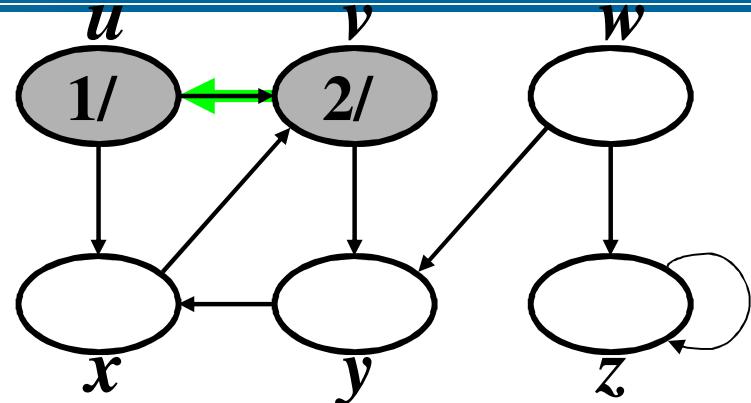
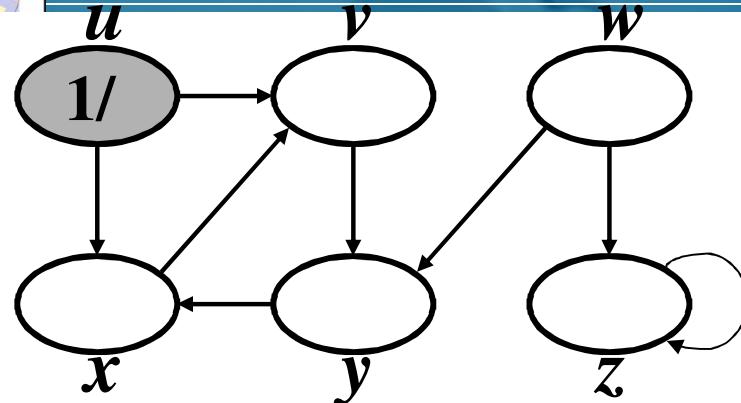
La variabile t utile per segnare i **marcatempi** inizio e fine visita

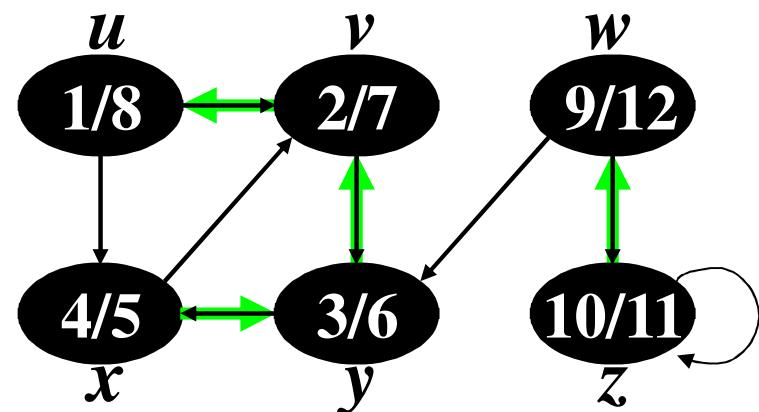
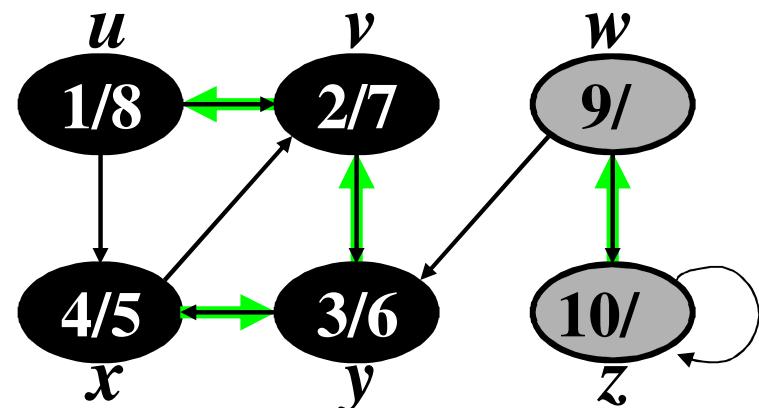
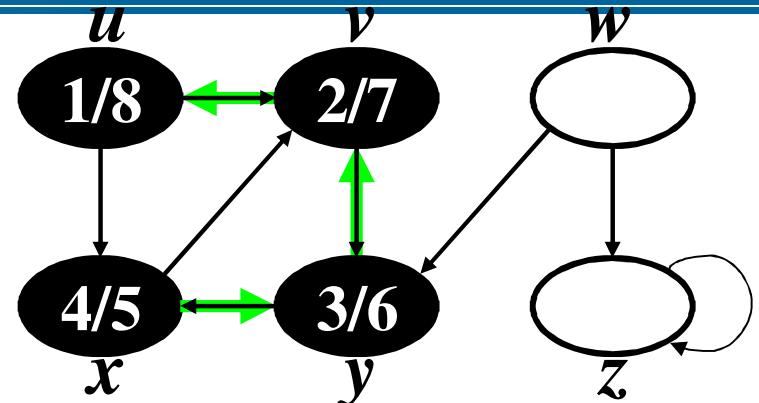
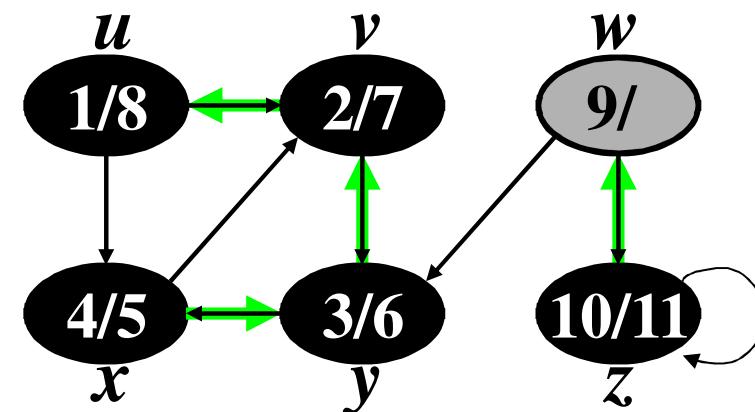
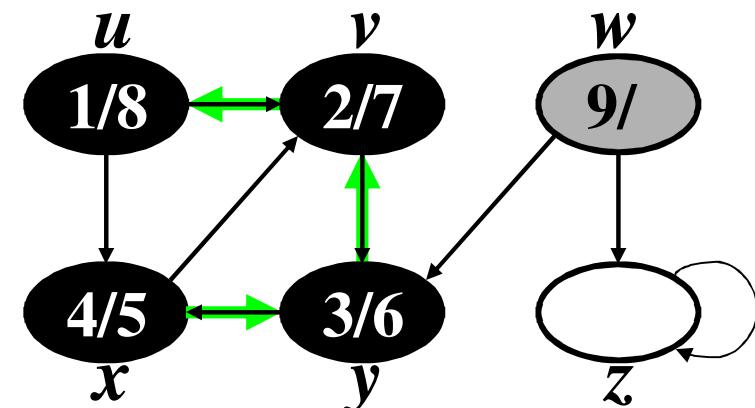
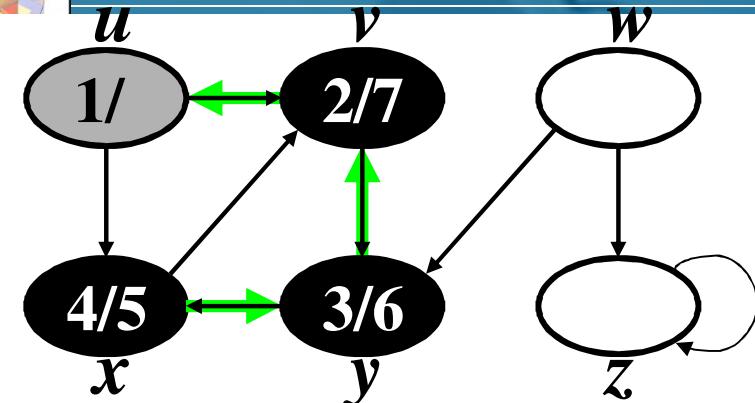
procedura visitaDFSRicorsiva(*vertice v, albero T*)

1. *marca e visita il vertice v* $\text{inizio}[v] \leftarrow t++$
2. **for each** (arco (v, w)) **do**
3. **if** (w non è marcato) **then**
4. aggiungi l'arco (v, w) all'albero T
5. visitaDFSRicorsiva(w, T)
6. $\text{fine}[v] \leftarrow t++$

algoritmo visitaDFS(*vertice s*) \rightarrow *albero*

6. $T \leftarrow$ albero vuoto $t \leftarrow 1$
7. visitaDFSRicorsiva(s, T)
8. **return** T





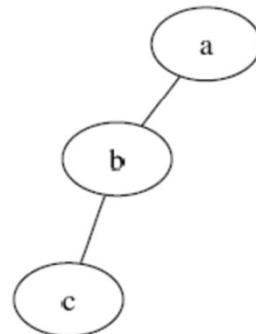


Riepilogo

- 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)
- Algoritmi di visita: visita in ampiezza e visita in profondità

Esercizio 1 (Appello Settembre 2016)

Sia $G = (V, E)$ un grafo *non orientato connesso* e sia s un nodo in V . Si ricorda che la distanza da s ad un nodo v di V è pari al numero minimo di archi che collega s a v . Scrivere un algoritmo che ritorni la *distanza media* di s da tutti gli altri nodi del grafo (escluso s). Nell'esempio in figura, la distanza media di b è 1, mentre la distanza media di a e c è 1.5.



Determinare, inoltre, il costo computazionale dell'algoritmo proposto.

Soluzione 1

- Visita BFS di costo $O(m+n)$
- Si calcola la distanza di ogni nodo (perché G è connesso) da s
 - Viene aggiunta ad una variabile tot che poi viene divisa per $n-1$

integer averageDistance(GRAPH G , NODE r)

```
QUEUE  $S \leftarrow$  Queue()
 $S.enqueue(r)$ 
integer[]  $dist \leftarrow$  new integer[1 ...  $G.n$ ]
integer  $tot \leftarrow 0$ 
foreach  $u \in G.V() - \{r\}$  do  $dist[u] \leftarrow -1$ 

 $dist[r] \leftarrow 0$ 
while not  $S.isEmpty()$  do
    NODE  $u \leftarrow S.dequeue()$ 
    foreach  $v \in G.adj(u)$  do
        if  $dist[v] < 0$  then
             $dist[v] \leftarrow dist[u] + 1$ 
             $tot \leftarrow tot + dist[v]$ 
             $S.enqueue(v)$ 

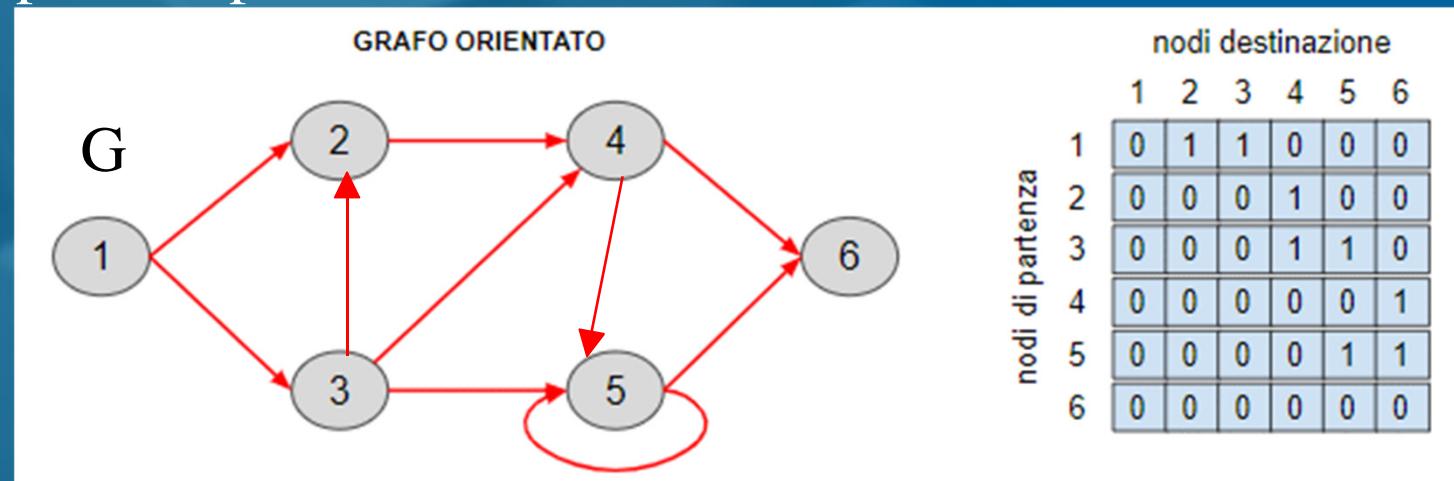

---

return  $tot/(G.n - 1)$ 
```

Esercizio 2 (Appello Giugno 2016)

Il *quadrato* di un grafo orientato $G = (V, E)$ è il grafo $G^2 = (V, E^2)$ tale che l'arco (x, y) appartiene a E^2 se e solo se esiste un vertice u tale che (x, u) appartiene ad E e (u, y) appartiene ad E . In altre parole, se esiste un percorso di due archi fra i nodi x e y . Scrivere un algoritmo che, dato un grafo G rappresentato con matrice di adiacenza, restituisce il grafo G^2 .

Esempio di input:



Soluzione 2

- Si considera la rappresentazione mediante matrice di adiacenza
- Il costo computazionale è $O(n^3)$

```
for i = 1 to n do      Iteriamo con i,j la matrice di adiacenza
    for j = 1 to n do
```

```
        k ← 1
        found ← false
        while k ≤ n ∧ not found do
            found ← A[i, k] = 1 ∧ A[k, j] = 1 ∧ i ≠ k ∧ j ≠ k
            k ← k + 1
        if found then
            A2[i, j] ← 1
        else
            A2[i, j] ← 0
```



Cammini minimi

Sia $G = (V,E)$ un **grafo orientato** ai cui archi (u,v) è associato un *costo* o *peso* $W(u,v)$ (ad es. un numero reale).

Il *costo di un cammino* $p = (v_0, v_1, \dots, v_k)$ è la somma dei costi degli archi che lo costituiscono.

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



Il costo di cammino minimo da un vertice u ad un vertice v è definito nel seguente modo:

$$\delta(u, v) = \begin{cases} \min\{W(p)\} & \text{se esistono cammini } p \text{ da } u \text{ a } v \\ \infty & \text{altrimenti} \end{cases}$$

Un cammino minimo da u a v è un cammino p da u a v di costo $W(p) = \delta(u, v)$.

Nel problema dei cammini minimi viene appunto richiesto di calcolare i cammini minimi.

Vi sono quattro versioni del problema:



- 1) Cammini minimi da un'unica sorgente a tutti gli altri vertici.
- 2) Cammini minimi da ogni vertice ad un'unica destinazione.
- 3) Cammini minimi da un'unica sorgente ad un'unica destinazione.
- 4) Cammini minimi tra tutte le coppie: da ogni vertice ad ogni altro vertice.



Noi risolveremo la **prima variante**.

- La seconda variante si risolve simmetricamente
- La terza si può risolvere usando la soluzione della prima
 - non si conosce alcun algoritmo asintoticamente migliore
- La quarta si può risolvere usando la soluzione della prima per ogni vertice del grafo, ma in genere si può fare di meglio



Archi di costo negativo. In alcuni casi il costo degli archi può anche essere negativo.

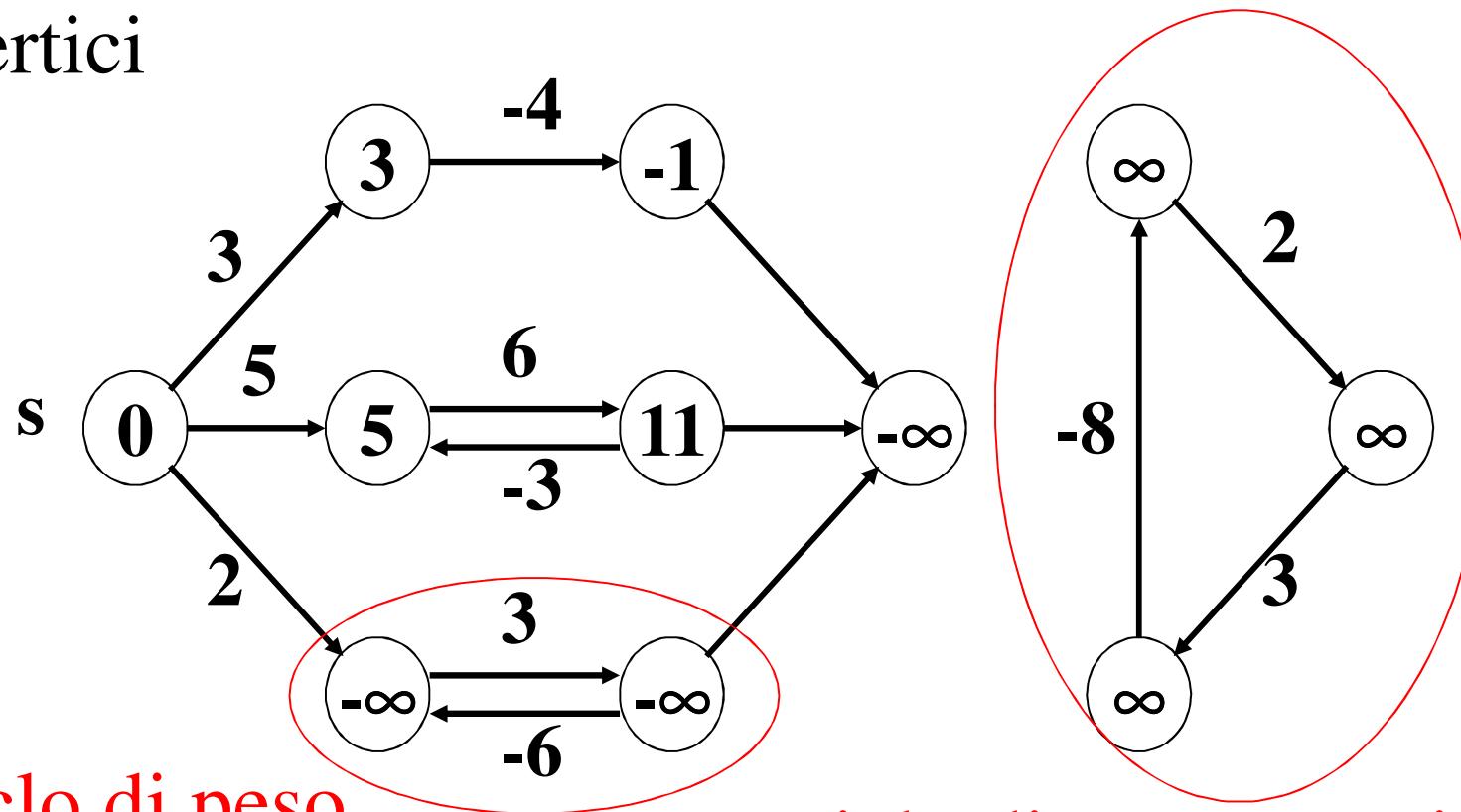
Questo non crea problemi nella ricerca dei cammini minimi da una sorgente s a meno che vi siano cicli di costo negativo raggiungibili da s .

Se u è un vertice raggiungibile da s con un cammino p passante per un vertice v di un ciclo negativo allora esistono cammini da s a u di costi sempre minori e il costo di cammino minimo $\delta(s,u)$ non è definito.

In questo caso poniamo $\delta(s,u) = -\infty$.

Esempio:

I costi minimi da s sono riportati all'interno dei vertici



ciclo di peso negativo -3

ciclo di peso negativo -3,
ma i vertici non sono
raggiungibili da s



Rappresentazione dei cammini minimi.

In genere ci interessa calcolare non solo i costi dei cammini minimi dalla sorgente s ad ogni vertice del grafo ma anche i cammini minimi stessi.

Siccome i cammini minimi hanno sottostruttura ottima possiamo rappresentarli aumentando ogni vertice **con un puntatore $p[v]$ che punta al vertice precedente in un cammino minimo da s a v .**



Teorema

Sottostruttura ottima dei cammini minimi. Se il cammino $p = (v_0, v_1, \dots, v_k)$ è minimo allora sono minimi anche tutti i sottocammini $p_{ij} = (v_i, \dots, v_j)$ per $0 \leq i \leq j \leq k$.

Dimostrazione. Per assurdo, se esistesse un cammino q da v_i a v_j di costo minore di p_{ij} allora sostituendo nel cammino p il sottocammino p_{ij} con il cammino q si otterrebbe un cammino da v_0 a v_k di costo minore di p . Impossibile se p è minimo.



Corollario.

Scomposizione dei costi di cammino minimo. Se p è un **cammino minimo** da s ad un vertice v diverso da s ed u è il vertice che precede v nel cammino allora

$$\delta(s, v) = \delta(s, u) + W(u, v).$$

Dimostrazione. Conseguenza della sottostruttura ottima: $\delta(s, v) = W(p) = \delta(s, u) + W(u, v)$.



Lemma: Limite superiore per i costi di cammino minimo.

Per ogni arco (u,v) vale la diseguaglianza:

$$\delta(s,v) \leq \delta(s,u) + W(u,v).$$

Dimostrazione.

- Se u non è raggiungibile da s allora: $\delta(s,u) = \infty$ e $\delta(s,v) \leq \infty + W(u,v)$ banalmente.
- Se u è raggiungibile da s allora $\delta(s,u) + W(u,v)$ è il costo di un cammino da s a v ed è quindi maggiore o uguale di $\delta(s,v)$.



Tecnica del rilassamento

Gli algoritmi che studieremo per il problema dei cammini minimi usano la tecnica del rilassamento.

Aggiungiamo ad ogni vertice v del grafo un campo $d[v]$ che rappresenta una *stima di cammino minimo*: durante tutta l'esecuzione dell'algoritmo è un limite superiore per $\delta(s, v)$ mentre alla fine è proprio uguale a $\delta(s, v)$.



Tecnica del rilassamento (continua)

L'inizializzazione dei campi $p[v]$ e $d[v]$ è la stessa per tutti gli algoritmi.

```
Inizializza( $G, s, d, p$ )
    for ogni  $v \in V[G]$  do
         $p[v] \leftarrow \text{nil}$ 
         $d[v] \leftarrow \infty$ 
     $d[s] \leftarrow 0$ 
```

G grafo pesato sugli archi



Tecnica del rilassamento (continua)

Il **rilassamento di un arco** (u, v) consiste nel controllare se è possibile migliorare il cammino finora trovato per v (e quindi la stima $d[v]$) allungando il cammino trovato per u con l'arco (u, v) .

```
Rilassa( $G, u, v, d, p, W$ )
  if  $d[v] > d[u] + W(u,v)$  then
     $d[v] \leftarrow d[u] + W(u,v)$ 
     $p[v] \leftarrow u$ 
```



Proprietà del rilassamento

Lemma1. *Effetto del rilassamento*. Dopo aver eseguito $Rilassa(G, u, v)$ vale la diseguaglianza

$$d[v] \leq d[u] + W(u, v)$$

Ovvero le *stime* $d[v]$ sono monotone non crescenti.

Dimostrazione

Se $d[v] > d[u] + W(u, v)$ prima del rilassamento, viene posto $d[v] = d[u] + W(u, v)$.

Se $d[v] \leq d[u] + W(u, v)$ prima del rilassamento, non viene fatto nulla e quindi è vero anche dopo.



Proprietà del rilassamento

Lemma2. Invariante del rilassamento.

Dopo l'inizializzazione per ogni vertice v vale la diseguaglianza $d[v] \geq \delta(s, v)$ che rimane vera anche dopo un numero qualsiasi di rilassamenti.

Inoltre se a un certo punto $d[v] = \delta(s, v)$ il suo valore non può più cambiare.

Dimostrazione.

Dopo l'inizializzazione:

- $d[s] = 0 \geq \delta(s, s)$ (in quali casi vale il $>$ stretto?)
- e per ogni altro vertice $d[v] = \infty \geq \delta(s, v)$



Se $d[v]$ non viene modificata durante l'esecuzione di $Rilassa(G,u,v)$ la disegualanza resta ancora vera; se $d[v]$ viene invece modificata allora $d[v] = d[u] + W(u,v)$.

Siccome $d[u]$ non è stata modificata vale la disegualanza $d[u] \geq \delta(s,u)$ e quindi per il limite superiore dei costi di cammino minimo:

$$d[v] \geq \delta(s,u) + W(u,v) \geq \delta(s,v)$$

Infine siccome il valore di $d[v]$ può soltanto diminuire e $d[v] \geq \delta(s,v)$ se $d[v] = \delta(s,v)$ allora il suo valore non può più cambiare.



Lemma3. Correttezza di $d[v]$ per vertici non raggiungibili.

Dopo l'inizializzazione per ogni vertice v non raggiungibile da s vale $d[v] = \delta(s, v)$ e tale uguaglianza rimane vera anche dopo un numero qualsiasi di rilassamenti.

Dimostrazione. Dopo l'inizializzazione $d[v] = \infty$ per ogni vertice diverso da s .

Se v non è raggiungibile da s allora $\delta(s, v) = \infty = d[v]$ e per l'invariante del rilassamento $d[v]$ non può più cambiare.



Lemma4. Estensione della correttezza di $d[v]$ per vertici raggiungibili. Se (u,v) è l'ultimo arco di un **cammino minimo** da s a v e $d[u] = \delta(s,u)$ prima di eseguire il rilassamento dell'arco (u,v) , allora dopo il rilassamento $d[v] = \delta(s,v)$.

Dimostrazione.

Dopo il rilassamento $d[v] \leq \delta(s,u) + w(u,v)$. Siccome (u,v) è l'ultimo arco di un cammino minimo: $\delta(s,v) = \delta(s,u) + w(u,v)$ e quindi $d[v] \leq \delta(s,v)$.

Per l'invariante del rilassamento $\delta(s,v)$ è anche un limite inferiore di $d[v]$, per cui vale necessariamente l'uguaglianza: $d[v] = \delta(s,v)$.



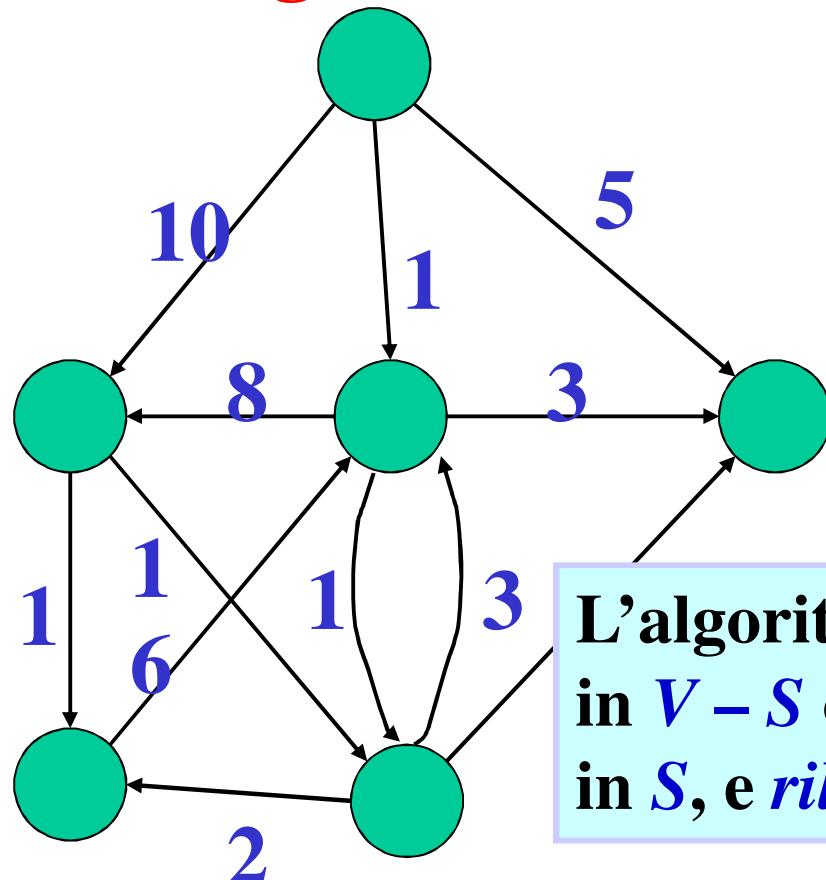
Possiamo concludere che qualsiasi algoritmo che esegua l'inizializzazione ed una sequenza di rilassamenti per cui alla fine $d[v] = \delta(s, v)$ per ogni vertice v calcola correttamente i cammini minimi.

Vi sono due algoritmi classici di questo tipo, uno dovuto a *Dijkstra* ed uno dovuto a *Bellman e Ford*.

L'algoritmo di Dijkstra richiede che i pesi degli archi non siano negativi mentre quello di Bellman-Ford funziona anche nel caso generale.

Algoritmo di Dijkstra

Algoritmo greedy che risolve il problema dei cammini minimi da singola sorgente **per pesi non negativi**



Utilizza un insieme S di vertici i cui pesi dei percorsi minimi sono già stati determinati.

L'algoritmo seleziona a turno il vertice u in $V - S$ col **minimo valore $d[u]$** , inserisce u in S , e **rilassa** tutti gli archi uscenti da u .



Pseudocodice - algoritmo di Dijkstra

```
Dijkstra( $G, s, d, p, W$ )
```

```
    Inizializza( $G, s, d, p$ )
```

```
     $S = \emptyset$ 
```

```
     $Q = V(G)$ 
```

Coda (di priorità)

```
    while ( $Q \neq \emptyset$ )
```

```
         $u = \text{extract\_Min}(Q)$  // scelta greeedy
```

```
         $S = S \cup \{u\}$ 
```

```
        for each vertice  $v$  adiacente a  $u$ 
```

```
            relax( $u, v, d, p, W$ )
```

```
relax( $u, v, d, p, W$ )
```

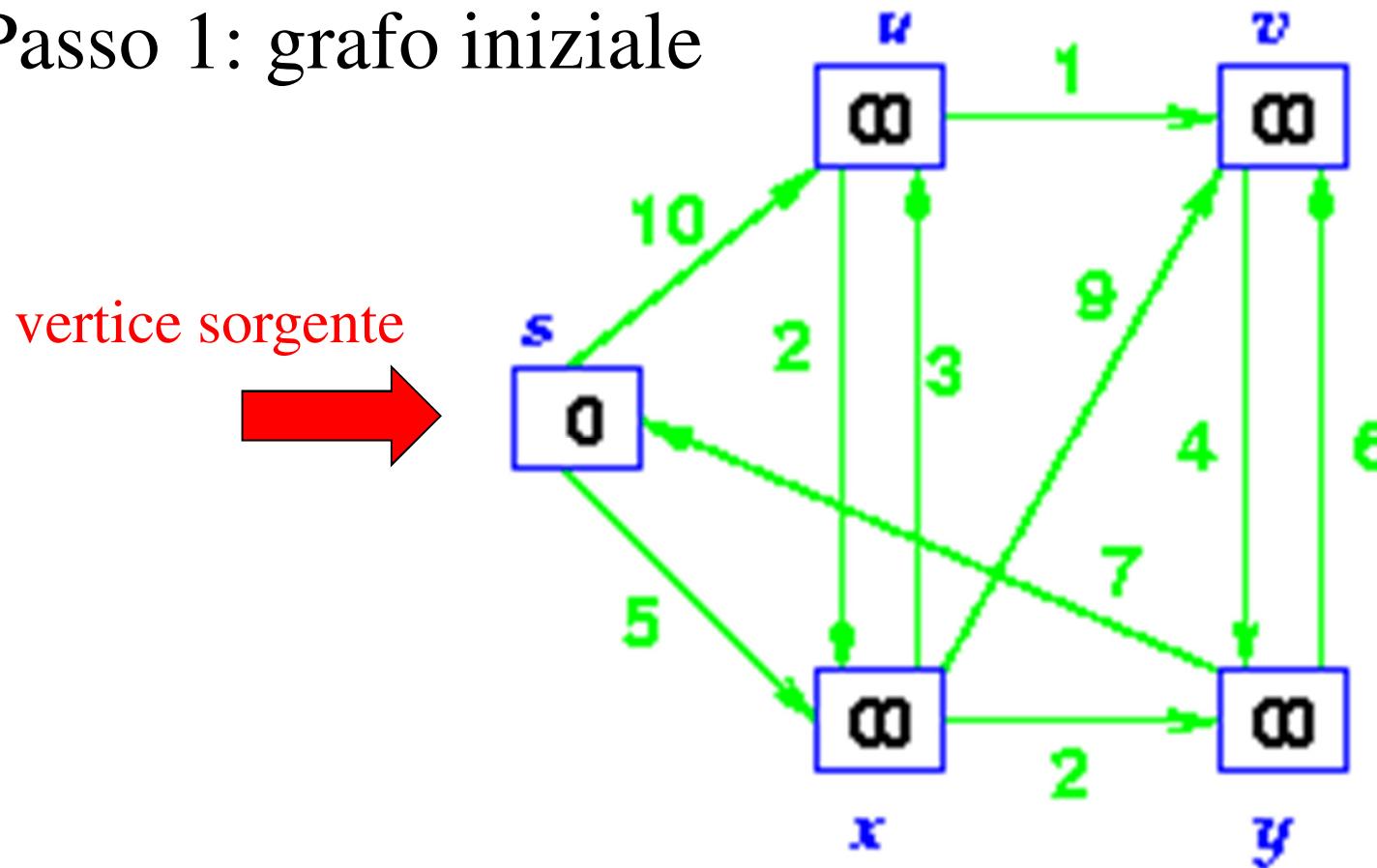
```
    if  $d[v] > d[u] + w(u, v)$ 
```

```
    then decrease_key( $d[v]$ ,  $d[u] + w(u, v)$ )
```

```
     $p[v] = u$ 
```

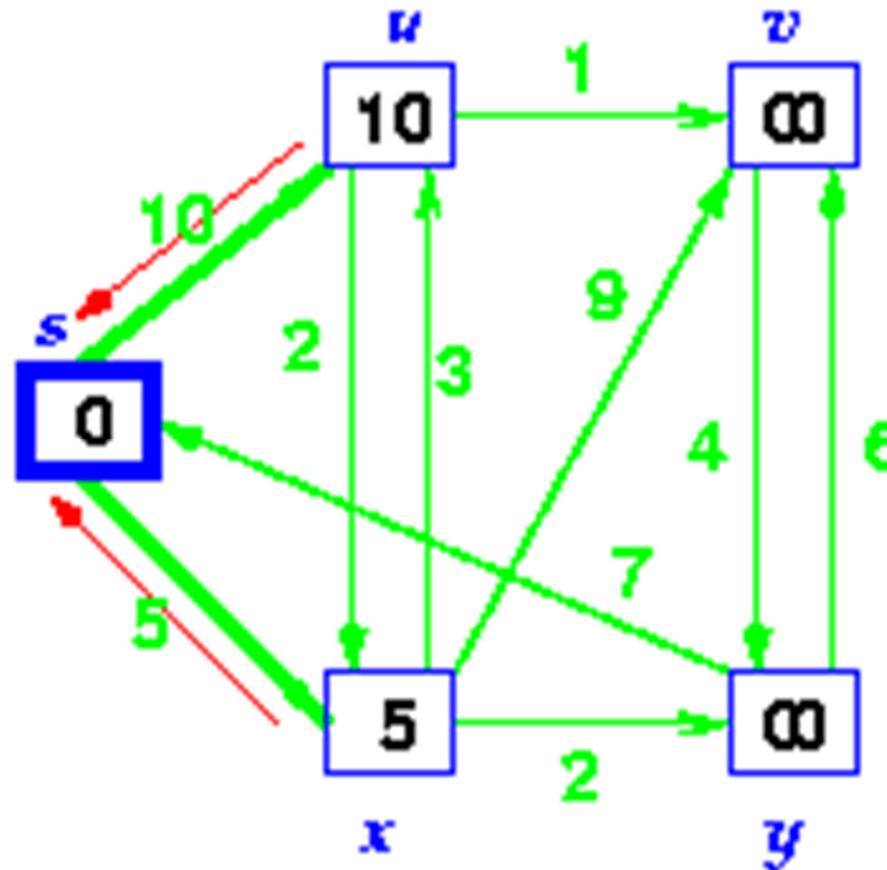
Esempio di esecuzione: Dijkstra

Passo 1: grafo iniziale



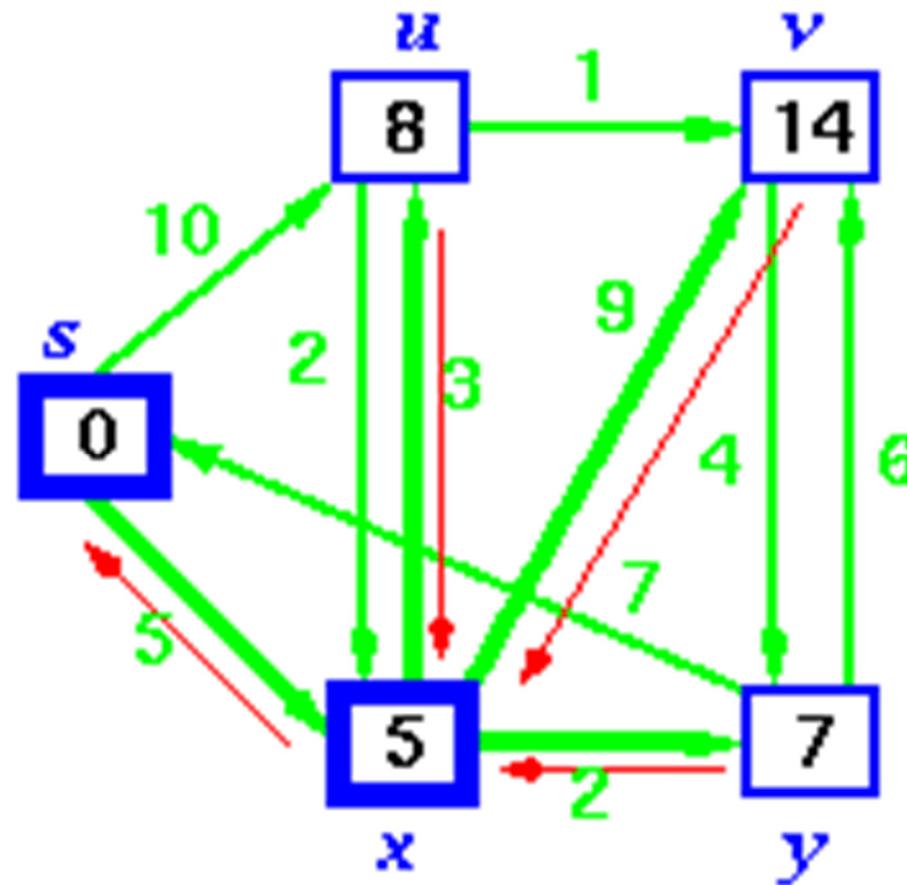
Esempio di esecuzione: Dijkstra

Passo 2: s viene estratto da Q ed i vertici adiacenti x ed u vengono «rilassati» (le **frecce rosse** indicano i predecessori nel cammino minimo)



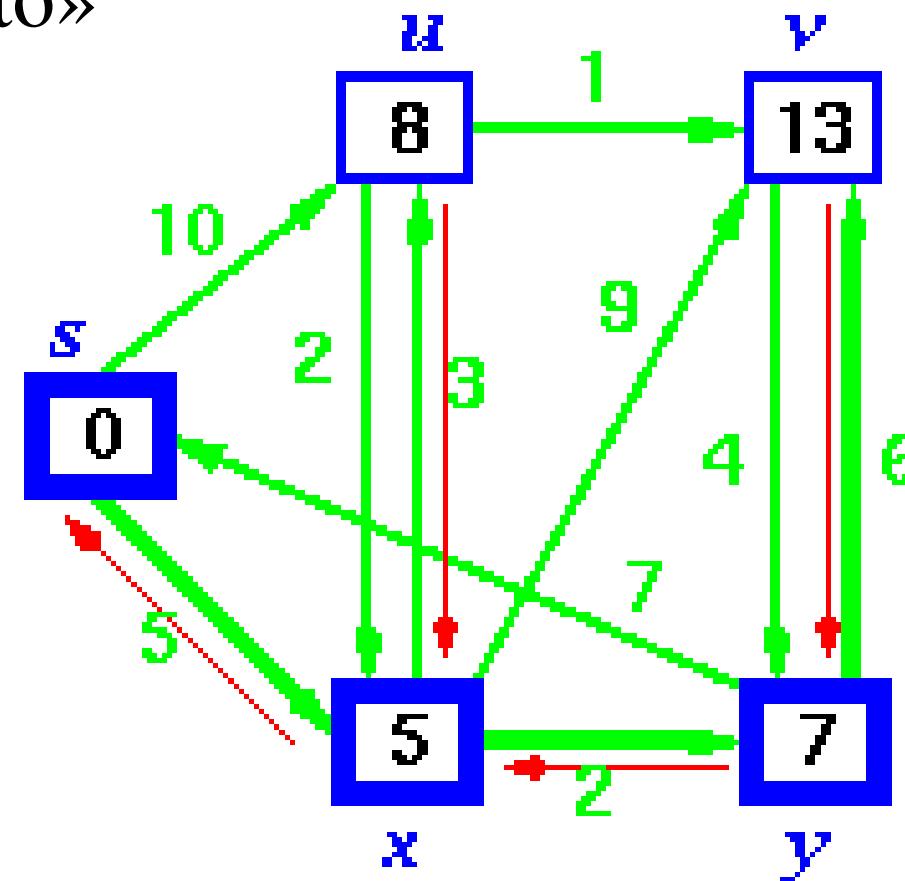
Esempio di esecuzione: Dijkstra

Passo 3: x viene estratto da Q e i vertici adiacenti u,v e y vengono «rilassati»



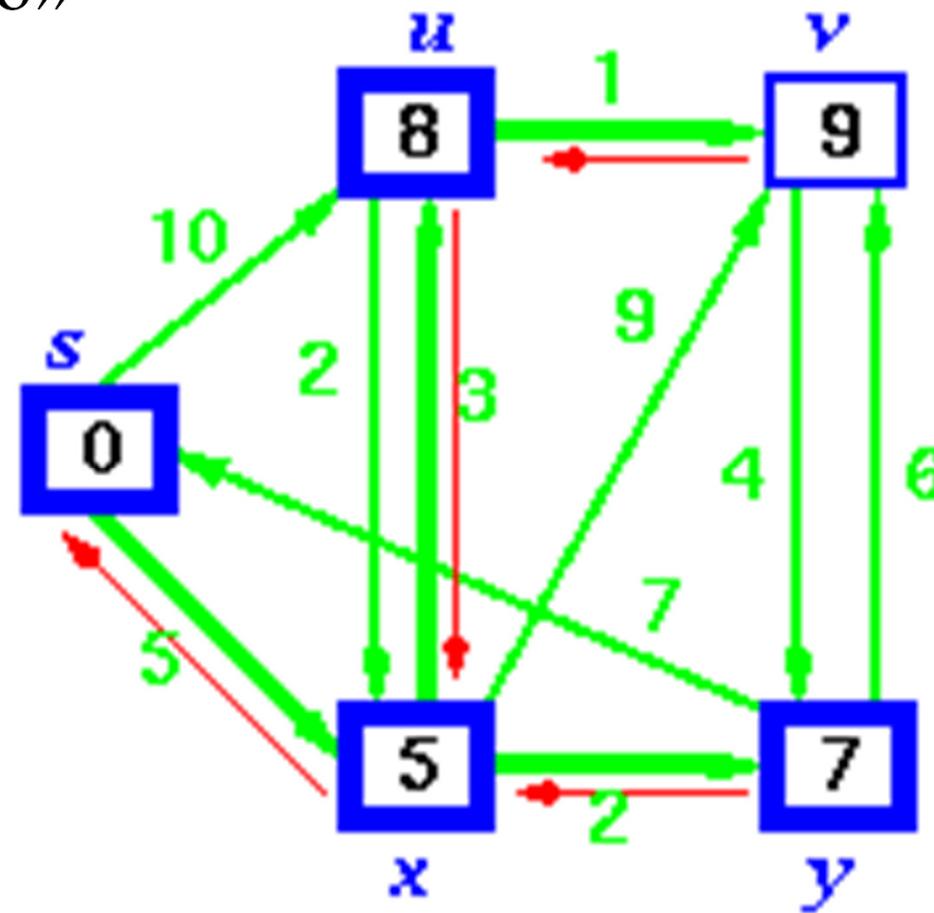
Esempio di esecuzione: Dijkstra

Passo 4: y viene estratto da Q ed il vertice adiacente v viene «rilassato»



Esempio di esecuzione: Dijkstra

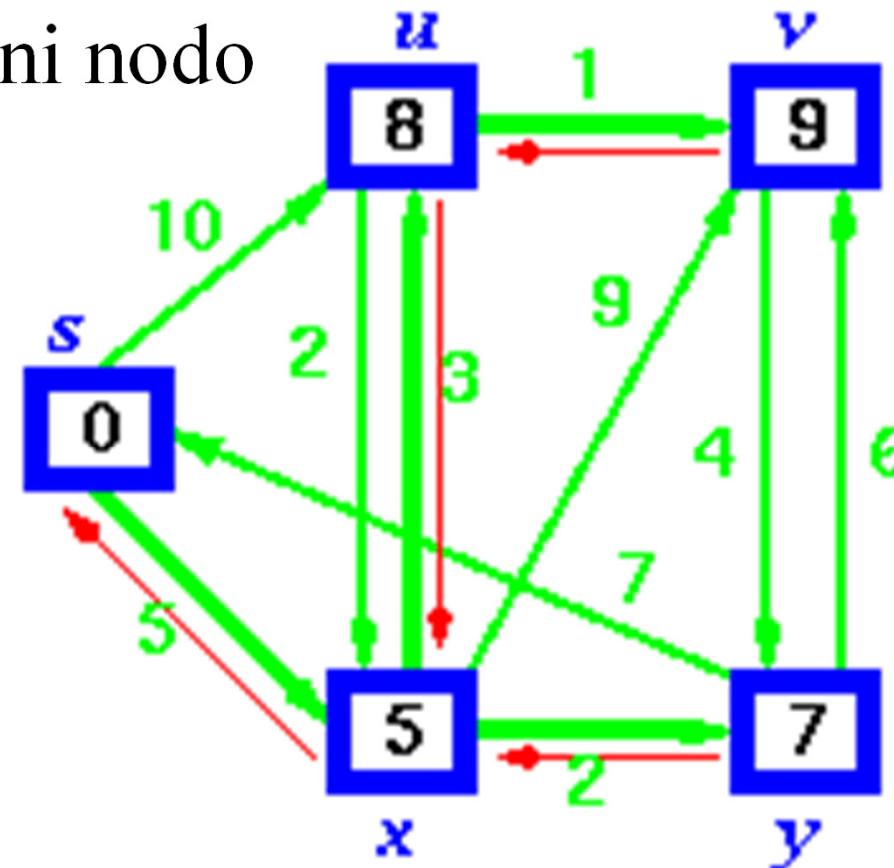
Passo 5: u viene estratto da Q ed il vertice adiacente v viene «rilassato»



Esempio di esecuzione: Dijkstra

Passo 6: infine v viene estratto da Q .

La lista dei predecessori ora definisce il cammino minimo da s per ogni nodo





Tempo di esecuzione: Dijkstra

Dipende dall'implementazione della coda di priorità:

- Differenti implementazioni danno differenti costi per le operazioni sulla coda.

extract_Min quante volte viene eseguita?

- $O(|V|)$ volte

relax (decrease_key) quante volte viene eseguita?

- $O(|E|)$ volte

```
Dijkstra(G, s, d, p, W)
    Inizializza(G, s, d, p)
    S = ∅
    Q = V(G)
    while ( Q ≠ ∅ )
        u = extract_Min(Q) //scelta greeedy
        S = S ∪ {u}
        for each vertice v adiacente a u
            relax(u, v, d, p, W)
```



Tempo di esecuzione: Dijkstra

- **Extract_Min** viene eseguita $O(|V|)$ volte.
- **Decrease_key** viene eseguita $O(|E|)$ volte.
- **Tempo totale** = $|V| T_{\text{extract-Min}} + |E| T_{\text{decrease_key}}$

Coda a priorità	$T_{\text{extract-Min}}$	$T_{\text{decrease_key}}$	Tempo totale
• Array	$O(V)$	$O(1)$	$O(V ^2)$
• Heap binario	$O(\lg V)$	$O(\lg V)$	$O((V + E) \lg V)$

Se G è denso è preferibile l'array



Algoritmo di Dijkstra: correttezza

Teorema: Se eseguiamo l'*algoritmo di Dijkstra* su un grafo orientato e pesato $G = (V, E)$, con funzione peso $w: E \rightarrow \mathbb{R}$ a valori reali *non-negativi* e un vertice sorgente s , allora, al termine $d[u] = \delta(s, u)$ per ogni vertice u in V .



Algoritmo di Dijkstra: correttezza

Dimostrazione: Dimostriamo che per ogni u di V , quando u viene inserito in S , vale $d[u] = \delta(s,u)$.

Supponiamo, per assurdo, che u sia il primo vertice per cui $d[u] \neq \delta(s,u)$ quando viene inserito in S .

Consideriamo la situazione all'inizio del *while* quando u viene inserito in S per ottenere la contraddizione $d[u] = \delta(s,u)$, esaminando un *percorso minimo* da s a u .

Sappiamo che:

1. $u \neq s$ poiché $d[s] = 0 = \delta(s,s)$ all'inizio del loop.
2. Ma allora deve pure valere anche $S \neq \emptyset$ prima che u sia inserito.
3. Ci deve essere un percorso da s a u altrimenti $d[u] = \infty = \delta(s,u)$, e quindi contraddizione.
4. Se c'è un percorso, c'è anche un *percorso minimo* p .

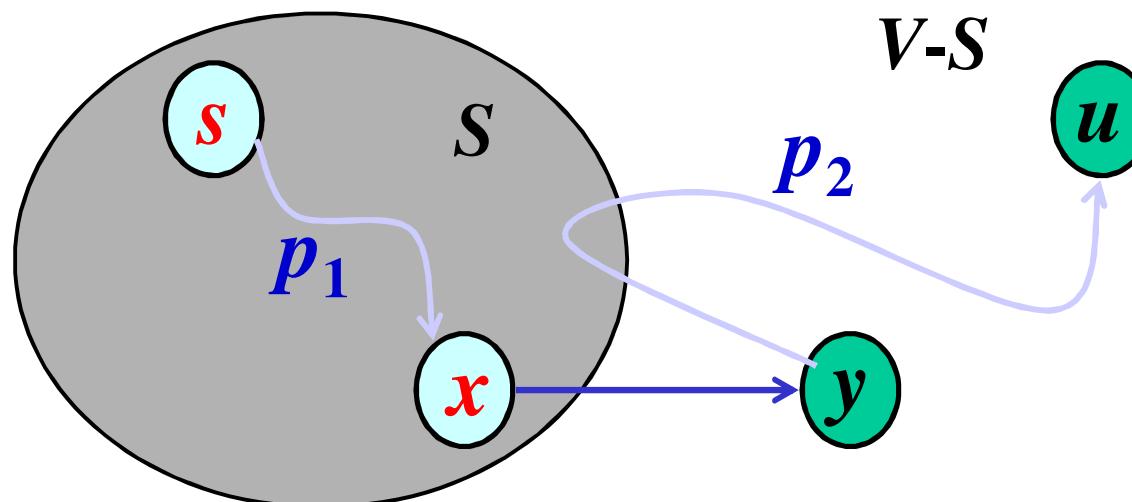
Algoritmo di Dijkstra: correttezza

Dimostrazione (cont.):

Il percorso minimo p connette un nodo (s) in S con un nodo (u) in $V-S$.

Sia y il primo in $V-S$ di p e x il suo predecessore.

p può essere scomposto in $p_1+(x,y)+p_2$ come in figura:



Algoritmo di Dijkstra: correttezza

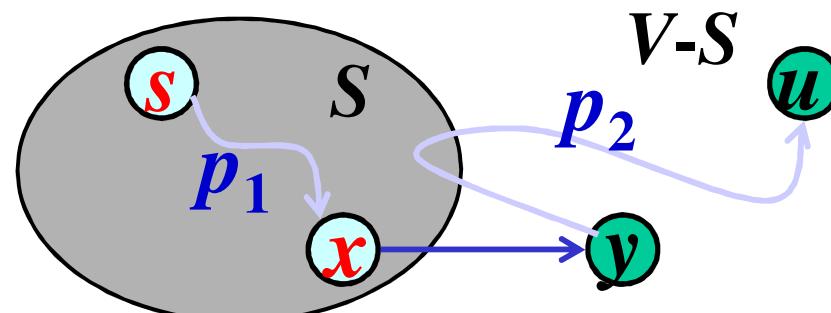
Dimostrazione (cont.):

Mostriamo ora che quando u è inserito in S , vale:

$$d[y] = \delta(s,y)$$

Infatti sappiamo che:

- u è il primo nodo per cui $d[u] \neq \delta(s,u)$, quando viene inserito in S
- x appartiene ad S , quindi avevamo $d[x] = \delta(s,x)$ quando è stato inserito in S
- Ma proprio in quel momento l'algoritmo rilassa l'arco (x,y) , e dal **Lemma4**, dopo la chiamata *relax* su (x,y) deve essere $d[y] = \delta(s,y)$



Algoritmo di Dijkstra: correttezza

Dimostrazione (cont.): Possiamo ora ottenere la nostra contraddizione. Poiché y compare nel *percorso minimo* tra s e u e i *pesi* (ed in particolare quelli in p_2) sono tutti *non-negativi*, si ha:

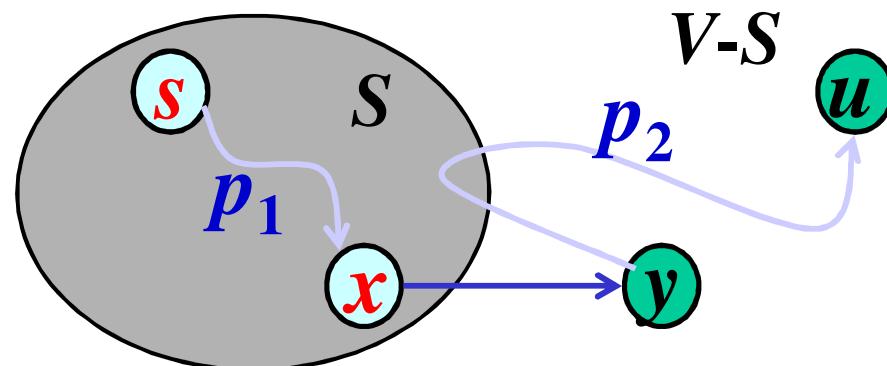
$$\delta(s,y) \leq \delta(s,u)$$

e inoltre $d[y] = \delta(s,y) \leq \delta(s,u) \leq d[u]$ (*Lemma 2*)

Ma poiché sia u che y sono in $V-S$ quando u viene estratto dalla coda, vale anche $d[u] \leq d[y]$.

Cioè $d[y] = d[u]$, e quindi

$d[y] = \delta(s,y) = \delta(s,u) = d[u]$ (*contraddizione!*)





In letteratura

- Algoritmi classici per il calcolo di distanze (e quindi di cammini minimi), basati sulla tecnica del rilassamento:
 - **Dijkstra**: cammini minimi a sorgente singola, grafi diretti senza pesi negativi, tempo $O((m+n) \log n)$
 - **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 $O(n+m)$
 - **Floyd e Warshall**: cammini minimi tra tutte le coppie in tempo $O(n^3)$

dove $n=|V|$ e $m=|E|$



UNIVERSITÀ
DEGLI STUDI
DI BERGAMO

Dipartimento
di Ingegneria Gestionale,
dell'Informazione e della Produzione



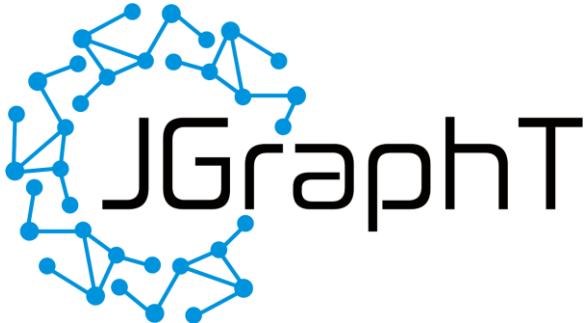
Esercizi su grafi con JGraphT

Corso di laurea
Magistrale in
Ingegneria
Informatica

PROGETTAZIONE, ALGORITMI E
COMPUTABILITÀ
(38090-MOD1)

RELATORE
Prof.ssa Patrizia
Scandurra

SEDE
DIGIP



- Sito web:

<https://jgrapht.org/>

- Documentazione:

<https://jgrapht.org/guide/UserOverview>

- Istruzioni operative per importare la libreria nel tuo IDE (ad esempio Eclipse)

<https://github.com/jgrapht/jgrapht/wiki/Users:-How-to-use-JGraphT-as-a-dependency-in-your-projects>

- JavaDoc online:

<https://jgrapht.org/javadoc/org.jgrapht.core/org/jgrapht/Graphs.html>

JGraphT: tipi di grafi

- **Interfaccia Graph:**
<https://jgrapht.org/javadoc/org.jgrapht.core/org/jgrapht/Graph.html>
- Classi concrete che la implementano:

Class Name	Edges	Self-loops	Multiple edges	Weighted
SimpleGraph	undirected	no	no	no
Multigraph	undirected	no	yes	no
Pseudograph	undirected	yes	yes	no
DefaultUndirectedGraph	undirected	yes	no	no
SimpleWeightedGraph	undirected	no	no	yes
WeightedMultigraph	undirected	no	yes	yes
WeightedPseudograph	undirected	yes	yes	yes
DefaultUndirectedWeightedGraph	undirected	yes	no	yes
SimpleDirectedGraph	directed	no	no	no
DirectedMultigraph	directed	no	yes	no
DirectedPseudograph	directed	yes	yes	no
DefaultDirectedGraph	directed	yes	no	no
SimpleDirectedWeightedGraph	directed	no	no	yes
DirectedWeightedMultigraph	directed	no	yes	yes
DirectedWeightedPseudograph	directed	yes	yes	yes
DefaultDirectedWeightedGraph	directed	yes	no	yes

Esempi

- Importare ed eseguire gli esempi forniti dal docente nell'archivio *<esempi con JGraphT>*

Esercizio 1

- Creare un grafo non orientato con JGraphT per rappresentare le amicizie come in fb. I nodi sono i nickname (di tipo String) delle persone.
- Esplorare l’interfaccia Graph o anche la classe statica Graphs per stampare a video gli “amici diretti” di nodi dati (la lista di adiacenza)
 - <https://jgrapht.org/javadoc/org.jgrapht.core/org/jgrapht/Graph.html>
 - <https://jgrapht.org/javadoc/org.jgrapht.core/org/jgrapht/Graphs.html>

Esercizio 2 (più complesso)

- Avete a disposizione il grafo delle amicizie di Facebook. Volete calcolare la persona che ha il maggior numero di “amici e amici di amici”, ovvero nodi a distanza 1 o 2 da esso.
- Definire un algoritmo che ritorna questo valore e calcolare la complessità.
- Implementare l’algoritmo usando JGraphT.

Soluzione da implementare con JGraphT

- Possiamo fare n visite BFS, in cui però limitiamo a due il numero massimo di passi
- l'algoritmo è $O(n^3)$, ma evita i duplicati

```
int facebook(GRAPH G)
```

```
    int max = 0
    foreach v ∈ G.V() do
        int t = count(GRAPH G, NODE v)
        if t > max then
            max = t
    return max
```

```
int count(GRAPH G, NODE r)
```

```
    boolean[] visited = new boolean[1...G.n] = {false}
    visited[r] = true
    int f = 0
    foreach u ∈ G.adj(r) do
        visited[u] = true
        f = f + 1
        foreach v ∈ G.adj(u) do
            if not visited[v] then
                visited[v] = true
                f = f + 1
    return f
```



UNIVERSITÀ
DEGLI STUDI
DI BERGAMO

Dipartimento
di Ingegneria Gestionale,
dell'Informazione e della Produzione



Software architecture validation

Corso di laurea
**Magistrale in
Ingegneria
Informatica**

PROGETTAZIONE, ALGORITMI E
COMPUTABILITÀ
(38090-MOD1)

RELATORE
Prof.ssa Patrizia
Scandurra

SEDE
DIGIP

DATA
04-10-2021

Outline

- Software architecture validation and methods
- Software quality and tactics
- Mathematical computation of reliability and availability of a software architecture
 - Examples



UNIVERSITÀ
DEGLI STUDI
DI BERGAMO

Dipartimento
di Ingegneria Gestionale,
dell'Informazione e della Produzione

Software architecture validation

- **Architecture validation** assess whether architecture meets certain quality goals
- **SA validation/evaluation methods** can be used to compare and identify strengths and weaknesses in different architecture alternatives during the early design stages
- Several evaluation methods exist to be used independently or combined



UNIVERSITÀ
DEGLI STUDI
DI BERGAMO

Dipartimento
di Ingegneria Gestionale,
dell'Informazione e della Produzione

Architecture evaluation methods

1. **Experience-based evaluations** are based on the previous experience and domain knowledge to assess if a software architecture will be good enough (*qualitative assessment*)
2. **Mathematical modeling** uses mathematical methods for evaluating in a quantitative manner (by *metrics*) quality requirements (e.g., performance and reliability)
3. **Simulation-based evaluations** rely on a high level implementation of some or all of the components in the software architecture; the simulation can then be used to evaluate quality requirements and correctness of the architecture.
 - Examples: Layered Queuing Network (LQN), event-based methods such as RAPIDE, PALLADIO (ADL UML-based)
4. **Scenario-based architecture evaluation** tries to evaluate a particular quality attribute by creating a profile of scenarios used to step through the software architecture and the consequences of the scenario are documented
 - The SEI *Architecture Tradeoff Analysis Method* (ATAM) is the leading method <https://www.sei.cmu.edu/our-work/software-architecture/>
 - Other methods: Software Architecture Analysis Method (SAAM), Architecture Level Modifiability Analysis (ALMA)



Software architecture and quality

- Some qualities are observable via execution:
 - Performance
 - Localise critical operations and minimise communications
 - Security
 - Use a layered architecture with critical assets in the inner layers
 - Safety
 - Localise safety-critical features in a small number of sub-systems
 - Availability
 - Include redundant components and mechanisms for fault tolerance
 - etc.
- And some are not observable via execution:
 - Maintainability
 - Use fine-grain, replaceable components
 - Modifiability, portability, reusability, integrability, testability, etc.



Quality properties may originate architectural conflicts

Examples:

- Using large-grain components improves performance but reduces maintainability
- Introducing redundant data improves availability but makes security more difficult

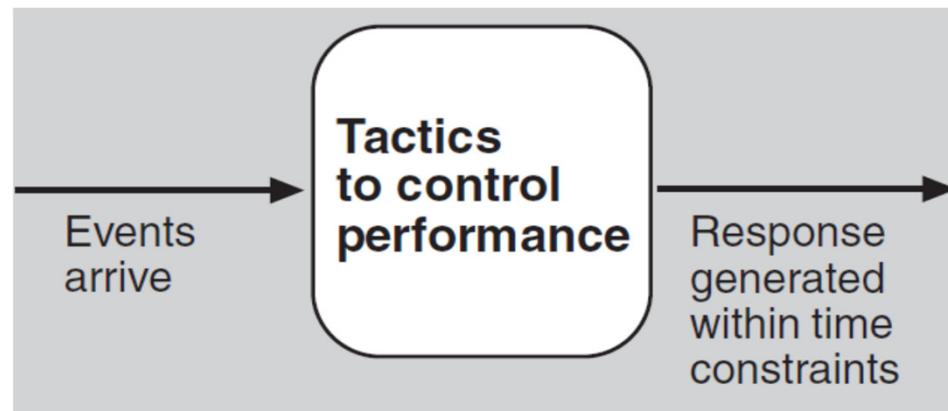


UNIVERSITÀ
DEGLI STUDI
DI BERGAMO

Dipartimento
di Ingegneria Gestionale,
dell'Informazione e della Produzione

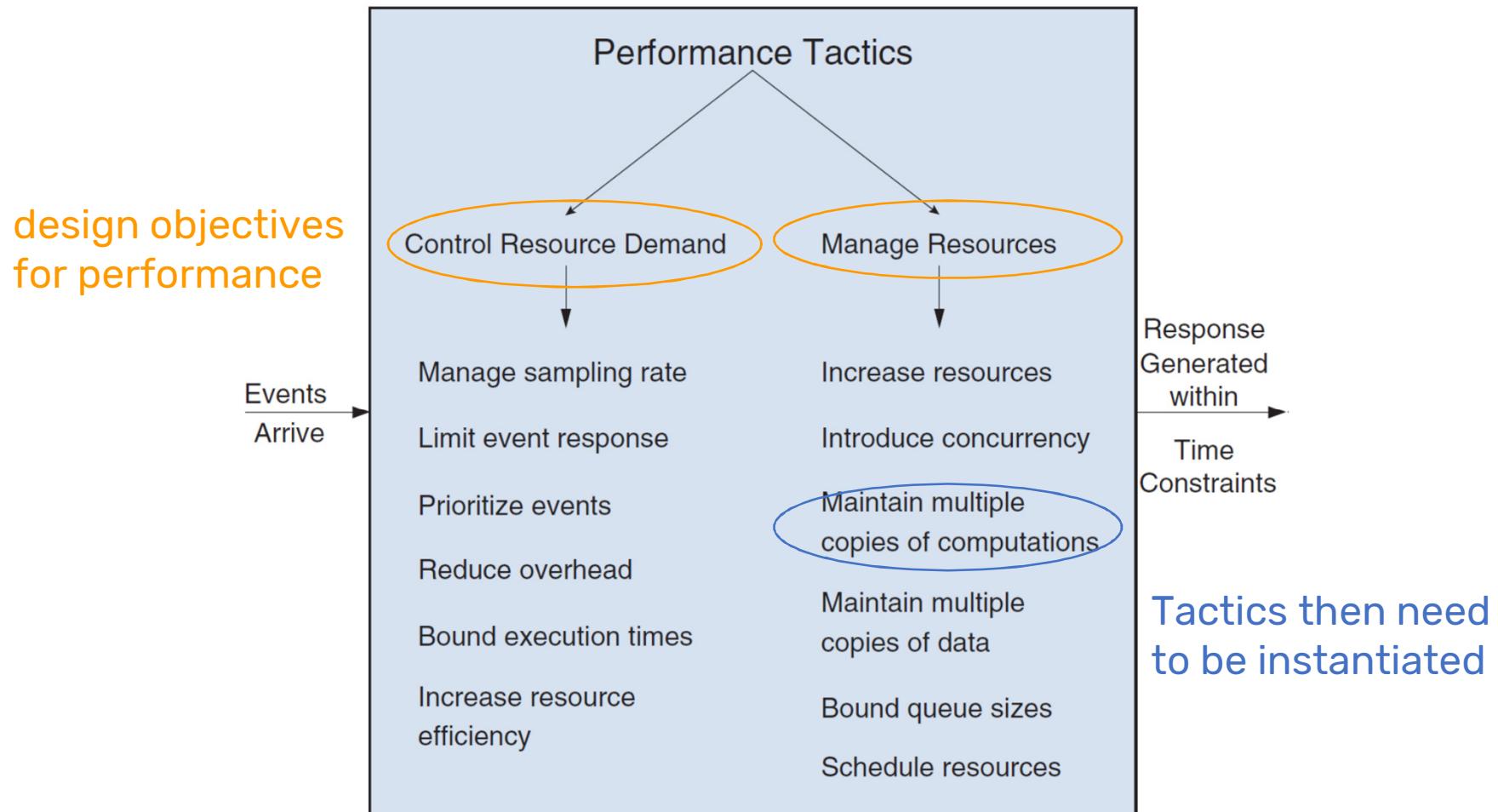
Tactics

- Architectural design primitives and decisions **to achieve a response for a particular quality attribute**
- Tactics are both simpler and more primitive than patterns
- Patterns, in contrast, typically focus on resolving and balancing multiple quality attribute goals
- *Example:* performance tactics to have low latency or high throughput



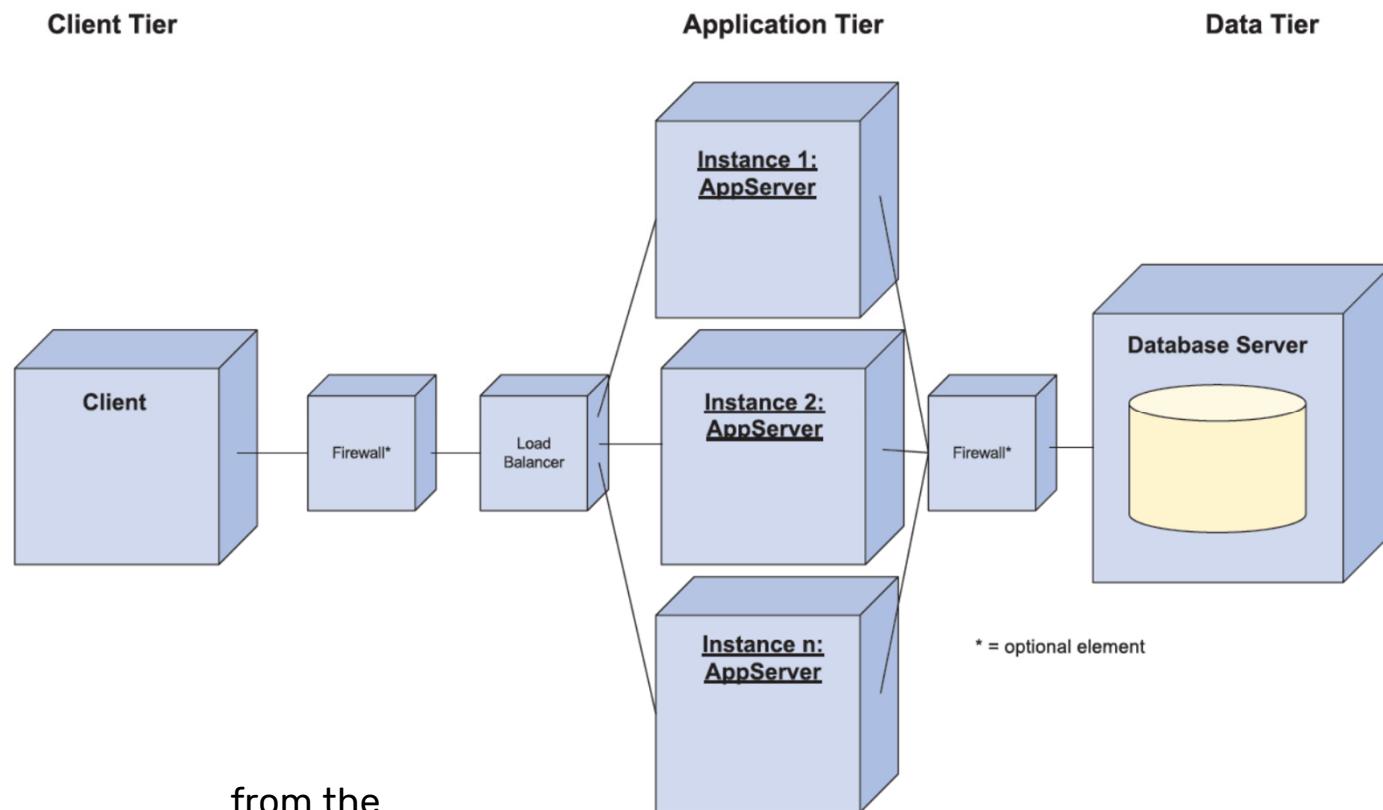
Performance tactics

Tactics provide **a top-down way of thinking** about design objectives related **to the achievement of a quality attribute**



Performance tactics: example of instantiation

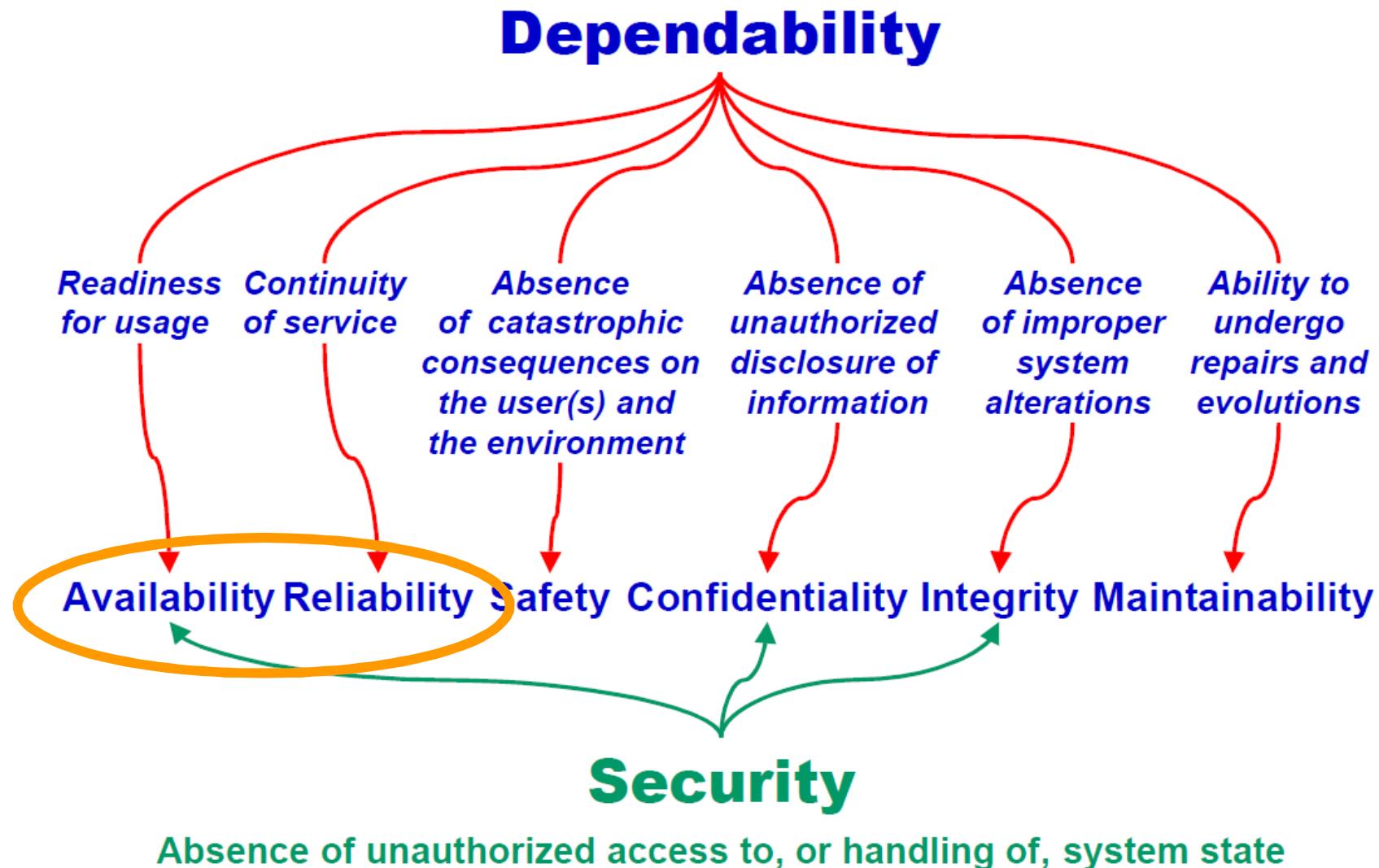
As an example, an architect might choose the *Load-Balanced Cluster deployment* pattern to maintain multiple copies of computations.



from the
Microsoft Application Architecture Guide (Key: UML)



Quality attributes in focus



Reliability and availability as measures

- Reliability tells information about the *failure-free interval*
- Definition:** $R(t)$ of a component/system is the probability that the component/system (working at time 0) is still working at time t
- **uptime**
- Availability tells information about how you use time
 - availability builds upon reliability **by adding the notion of recovery**—that is, when the system breaks, it repairs itself
 - Both are **described in % values or as number of “nines”**

Example:

$R(1000) = 90\%$ means 90% chance of not failing in 1000 hours of operations



UNIVERSITÀ
DEGLI STUDI
DI BERGAMO

Dipartimento
di Ingegneria Gestionale,
dell'Informazione e della Produzione

Reliability

Number of “nine” is important!

Uptime (%)	Downtime (%)	Downtime per year	Downtime per week
90%	10%	36.5 days	16:48 hours
99%	1%	3.65 days	1:41 hours
99.9%	0.1%	8:45 hours	10:05 minutes
99.99%	0.01%	52:30 minutes	1 minute
99.999%	0.001%	5:15 minutes	6 seconds
99.9999%	0.0001%	31.5 seconds	0.6 seconds



More formally: Reliability theory

X : instant of failure of a component

$F(t)$: its cumulative distribution

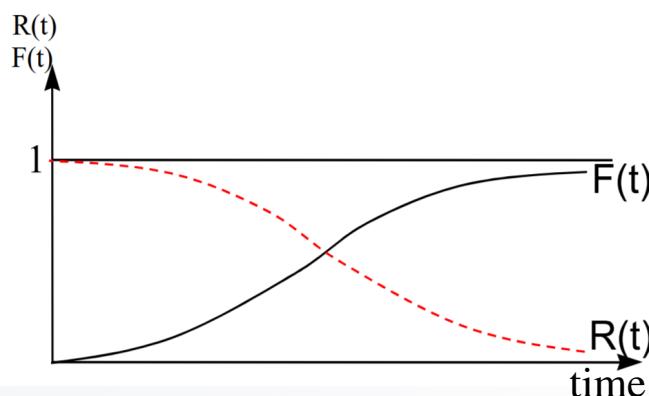
$R(t)$: probability that the component (working at time 0) is still working at time t

$$R(t) = P(X > t) = 1 - F(t)$$

$f(t)dt$ is the probability of failure in $(t, t+dt)$

$$F(t) = \int_0^t f(t)dt$$

$$R(t) = 1 - F(t) = 1 - \int_0^t f(t)dt = \int_t^\infty f(t)dt$$



UNIVERSITÀ
DEGLI STUDI
DI BERGAMO

Dipartimento
di Ingegneria Gestionale,
dell'Informazione e della Produzione

Reliability for exponential distribution

In the case $F(t)$ is exponential an approximation is available:

$$F(t) = P(X \leq t) = 1 - e^{-\frac{t}{MTTF}}$$
$$\approx \frac{t}{MTTF} \quad \left(\text{for } \frac{t}{MTTF} \ll 1 \right)$$

$$R(t) = 1 - F(t) = e^{-t/MTTF}$$

about $1 - t/MTTF$

where:

- MTTF (Mean Time To Failure): mean time before next failure
- t is smaller than MTTF



UNIVERSITÀ
DEGLI STUDI
DI BERGAMO

Dipartimento
di Ingegneria Gestionale,
dell'Informazione e della Produzione

Exercise 1: failure probability

Compute the probability that a disk with MTTF = 100000 hours fails at least once every 3 years:

- $t = 3 \times 365 \times 24 = 26280$ hours

$$P(X \leq t) = 1 - e^{-\frac{26280}{100000}} \quad \text{That is } \approx 23\%$$

- considering the 3 years period *small* with respect to the MTTF, probability can be approximated with: $26280/100000 \approx 26\%$
- If instead we have 2 disks, the probability that at least one of them fails is:

$$P(X \leq t) = 1 - e^{-\frac{26280}{100000} \times 2} \quad \text{That is } \approx 41\%$$



How to increase reliability

$$R(t) = 1 - F(t) = e^{-t/MTTF}$$

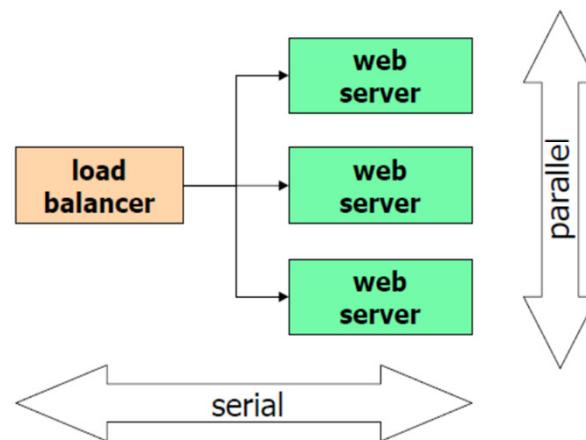
- A system that comprises redundant elements can tolerate a number of failed components
- To increase reliability:
 - Use multiple redundant components
 - Use element internally highly reliable (high *MTTF*)
 - Have *spare* components at disposal
 - Reduce *MTTR* to a minimum

System reliability computation

Each component is independent from the others (concerning the failure probability) and the system is not operational (*down*) if not able to provide a given service level

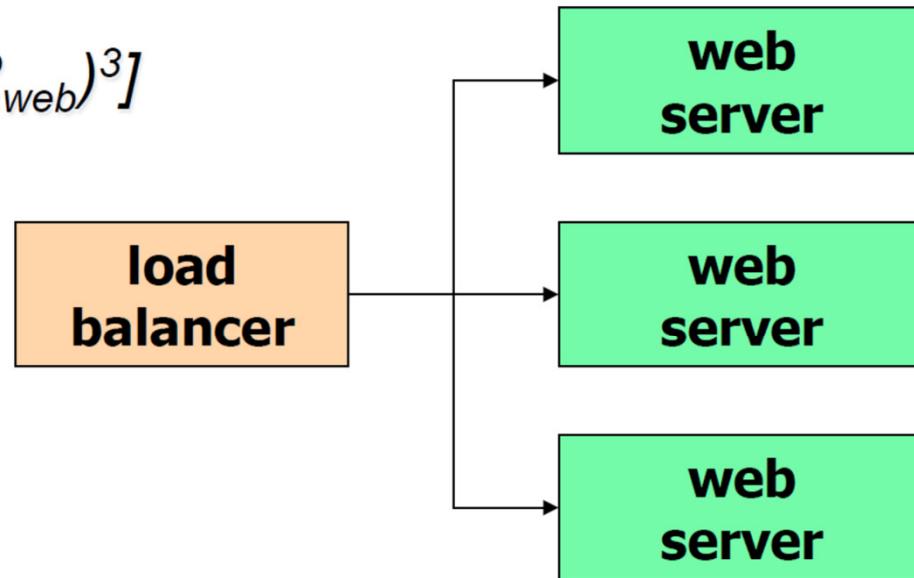
The working state of the whole system depends on how many and which components are operative – its behavior can be represented using reliability diagrams **RBD** (Reliability Block Diagram)

- **serial** connections between components requires that *all* must be operative
- **parallel** connections requires that *at least one* (or some, if differently specified) must be operative



Reliability Block diagram

- $R = R_{lb} [1 - (1 - R_{web})^3]$

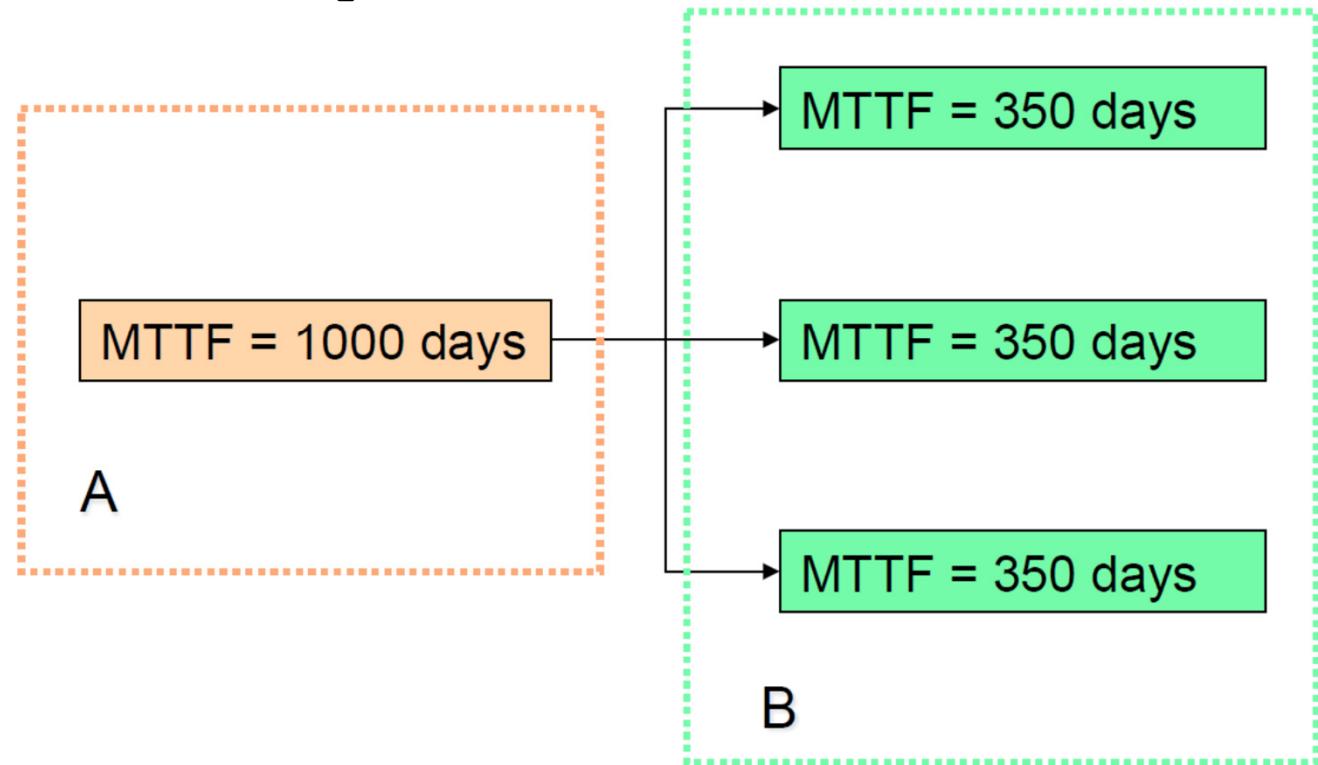


- Components in serial = *non redundant*
- Components in parallel = *redundant*
- Reliability of components in *serial*
 - $R = \prod R_i$
- Reliability of components in *parallel*
 - $R = 1 - \prod (1 - R_i)$



Exercise 2: computation of

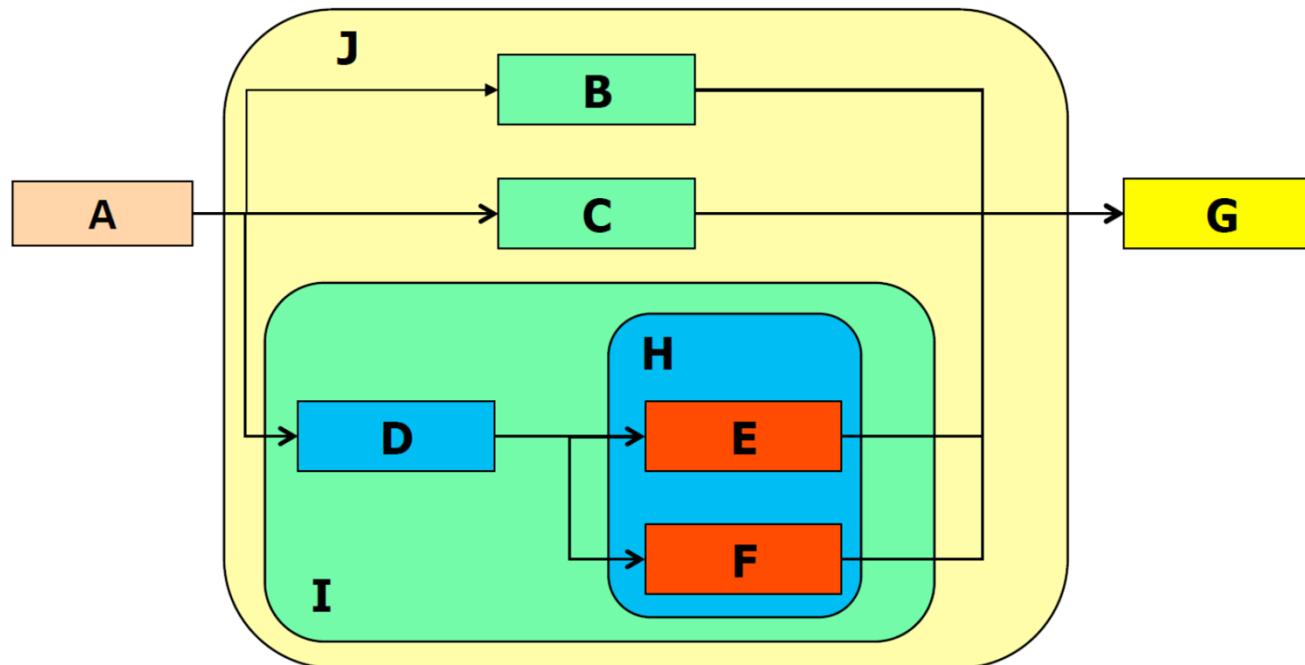
- $R(7)$: probability of no failure in 7days
 $\approx 1 - 7 / \text{MTTF}$



- $R_A(7) = 1 - 7/1000 = 0.993$
- $R_B(7) = 1 - (7/350)^3 = 0.999992$ (*parallel*)
- $R_{A+B}(7) = R_A(7) \times R_B(7) = 0.992992$ (*serial*)



Reliability Block diagram with hierarchical design



Bottom-up approach

$$1. H = E // F$$

$$2. I = H + D$$

$$3. J = B // C // I$$

$$4. S = A + J + G$$



Quantitative system availability parameters : Mean Time To Repair (MTTR)

Mean time required to repair a failed component

- mean time during which a component is not working
- it includes the time required to
 - Discover the failure
 - Detect the failure
 - Remove the faulty component
 - Repair the component
 - Reset the component
 - Software operations needed to reset the system



System availability formula

- the *availability* represents the *probability* (stationary mean) that at a given instant of time the system is *working*. It can be computed as the fraction of time during which the system works correctly.

$$A = \frac{\text{MTTF}}{\text{MTTF} + \text{MTTR}}$$

- Typical notation is that of *nines*
 - Availability at 3-nine corresponds to 99.9%
 - Availability at 5-nine corresponds to 99.999%



UNIVERSITÀ
DEGLI STUDI
DI BERGAMO

Dipartimento
di Ingegneria Gestionale,
dell'Informazione e della Produzione

Availability computation

Failure conditions

$$A = \prod_{i=1}^n A_i$$

- *One component (serial)*

$$A = 1 - \prod_{i=1}^n (1 - A_i)$$

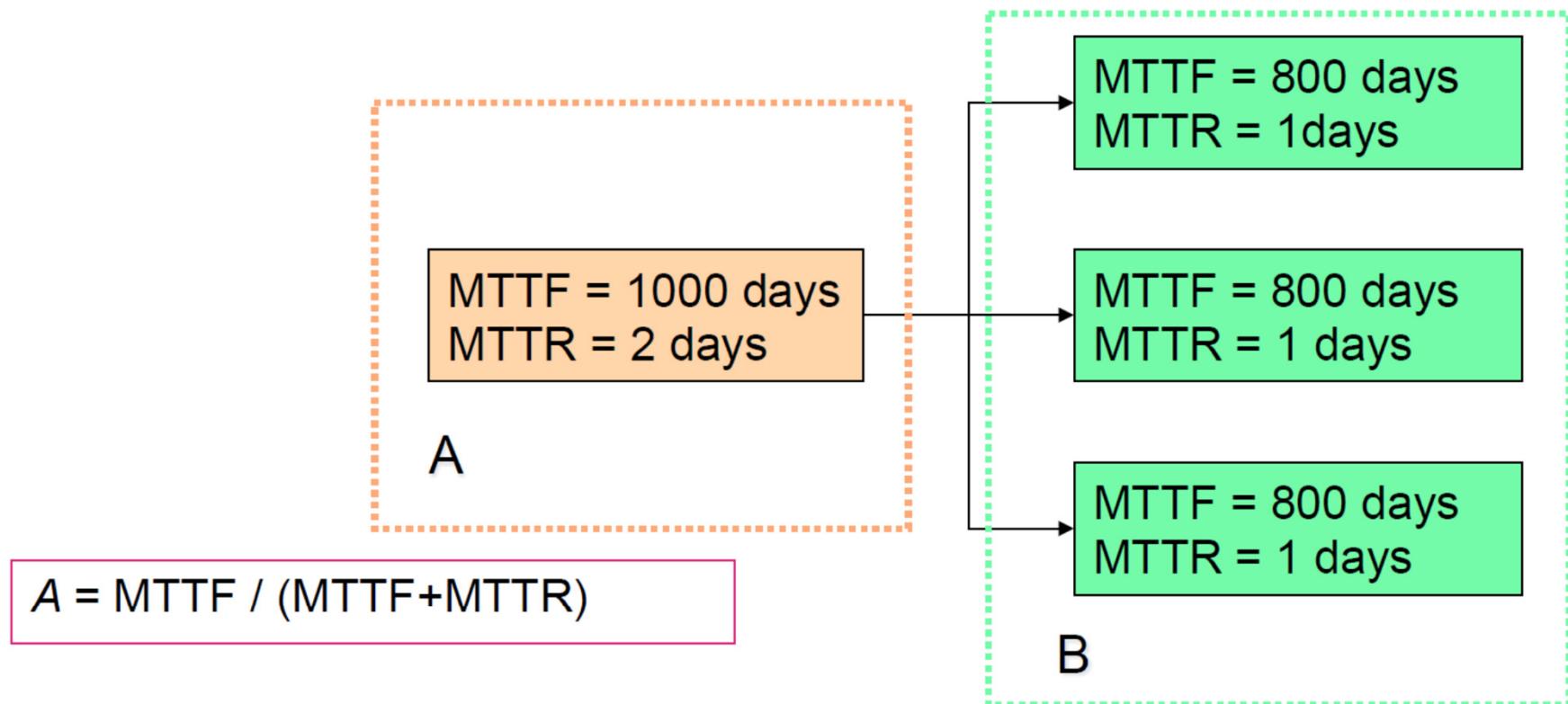
- *All components (parallel)*



UNIVERSITÀ
DEGLI STUDI
DI BERGAMO

Dipartimento
di Ingegneria Gestionale,
dell'Informazione e della Produzione

Esercise 3: availability A



$$A_A = 1000 / (1000 + 2) = 0.998$$

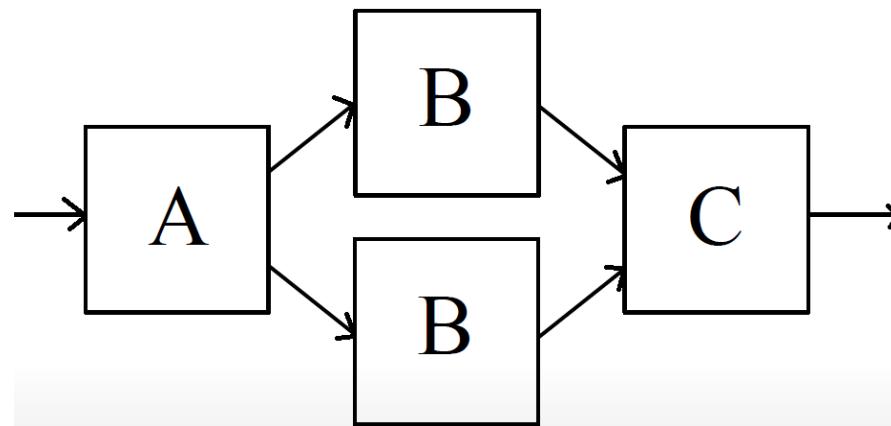
$$A_B = 1 - (1 - 800 / (800+1))^3 = 1 - 0.00125^3 = 0.999999998 \quad (\text{parallel})$$

$$A_{A+B} = A_A \cdot A_B = 0.998 \quad (\text{serial})$$



Exercise 4

Compute the system reliability R_{sys} assuming as reliability values for components A, B and C the following: $R_A = 60\%$, $R_B = 85\%$ e $R_C = 80\%$.



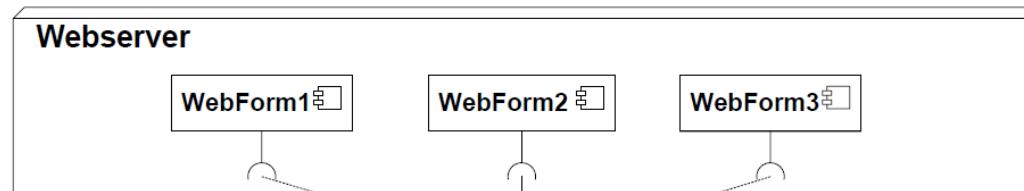
Solution:

$$R_{Sys} = R_A \times (1 - (1 - R_B)^2) \times R_C = \\ 0,6 \times (1 - 0,0225) \times 0,8 = 0,6 \times 0,98 \times 0,8 = 0,47 = 47\%$$

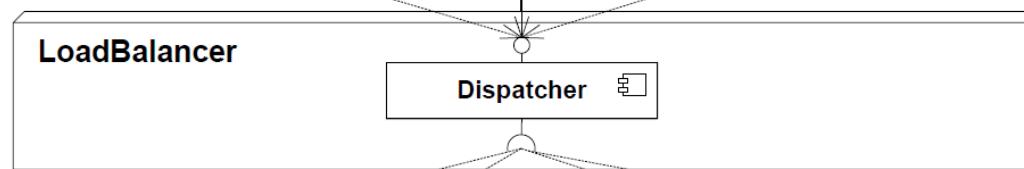


Exercise 5

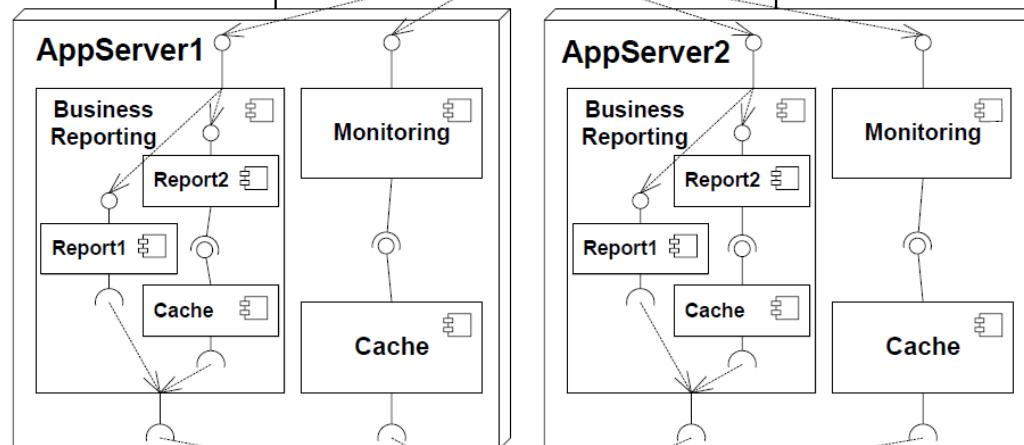
Formulate the system reliability R_{sys}



$$R_W = 1 - (1 - R_{WF})^3$$

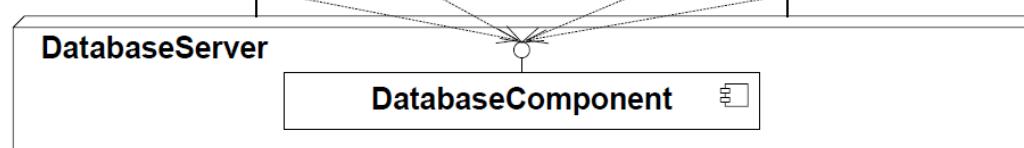


$$R_{LB} = R_D$$



$$R_{AS} = 1 - (1 - R_{BR}) \times (1 - R_M * R_C)$$

$$R_{BR} = 1 - (1 - R_{R1}) \times (1 - R_{R2} * R_C)$$



$$R_{DS} = R_{DC}$$

$$R_{sys} = R_W \times R_{LB} \times (1 - (1 - R_{AS})^2) \times R_{DS}$$





UNIVERSITÀ
DEGLI STUDI
DI BERGAMO

Dipartimento
di Ingegneria Gestionale,
dell'Informazione e della Produzione



Stili architetturali di base

Corso di laurea
Magistrale in
Ingegneria
Informatica

PROGETTAZIONE, ALGORITMI E
COMPUTABILITÀ
(38090-MOD1)

RELATORE
Prof.ssa Patrizia
Scandurra

SEDE
DIGIP

Outline

- Basic concepts
- Basic architectural styles and examples
 - Pipe-and-filter
 - Multi-layer
 - Client-server and variants
 - Broker
 - MVC
 - Message-oriented architecture
 - Service-oriented architecture
 - Hexagonal architecture (or ports and adapters architecture)

Architectural styles

- The architecture of a system may conform to a generic architectural model or style
- An awareness of these styles can simplify the problem of defining system architectures
- However, most large systems are heterogeneous and do not follow a single architectural style

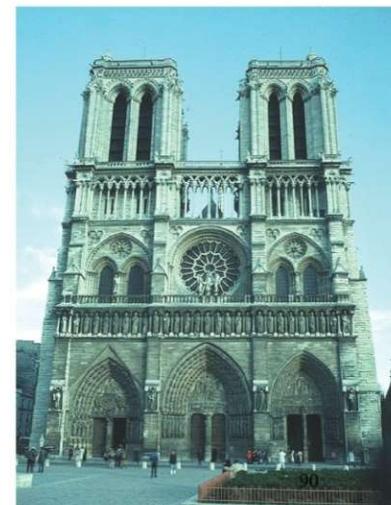
The Classical Style



The Californian Style



The Gothic Style

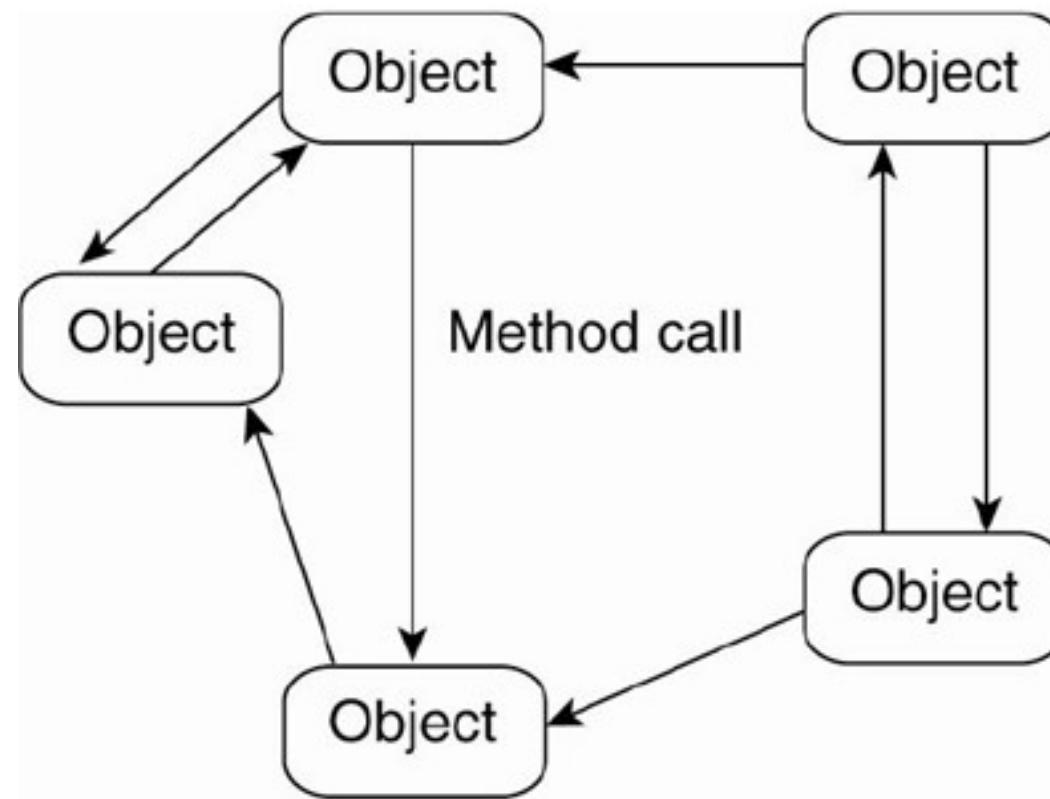


Definition

- An **architectural style** is a *named collection of architectural design decisions* that:
 - are applicable in a given development context
 - constrain architectural design decisions to a particular system within that context
 - elicit beneficial qualities in each resulting system
- Recurring organizational patterns & idioms
 - Established, shared understanding of common design forms
 - Mark of mature engineering field
 - Shaw & Garlan
- Abstraction of recurring composition & interaction characteristics in a set of architectures
 - Medvidovic & Taylor

A well-known architectural style

- Architetture based on **objects and method calls**
 - local or
 - remote (via an IPC mechanism – e.g., RPC and RMI)
- *Synchronous interaction style*



The two dimensions of a inter-process communication mechanism

- First dimension (*space dimension*):
 - the interaction is **one-to-one** or **one-to-many**?
 - how interacting parts get to know each other?
- Second dimension (*time dimension*):
 - the interaction is **synchronous** or **asynchronous**?
- The two dimensions combined determined **different interaction styles and architecture styles!**

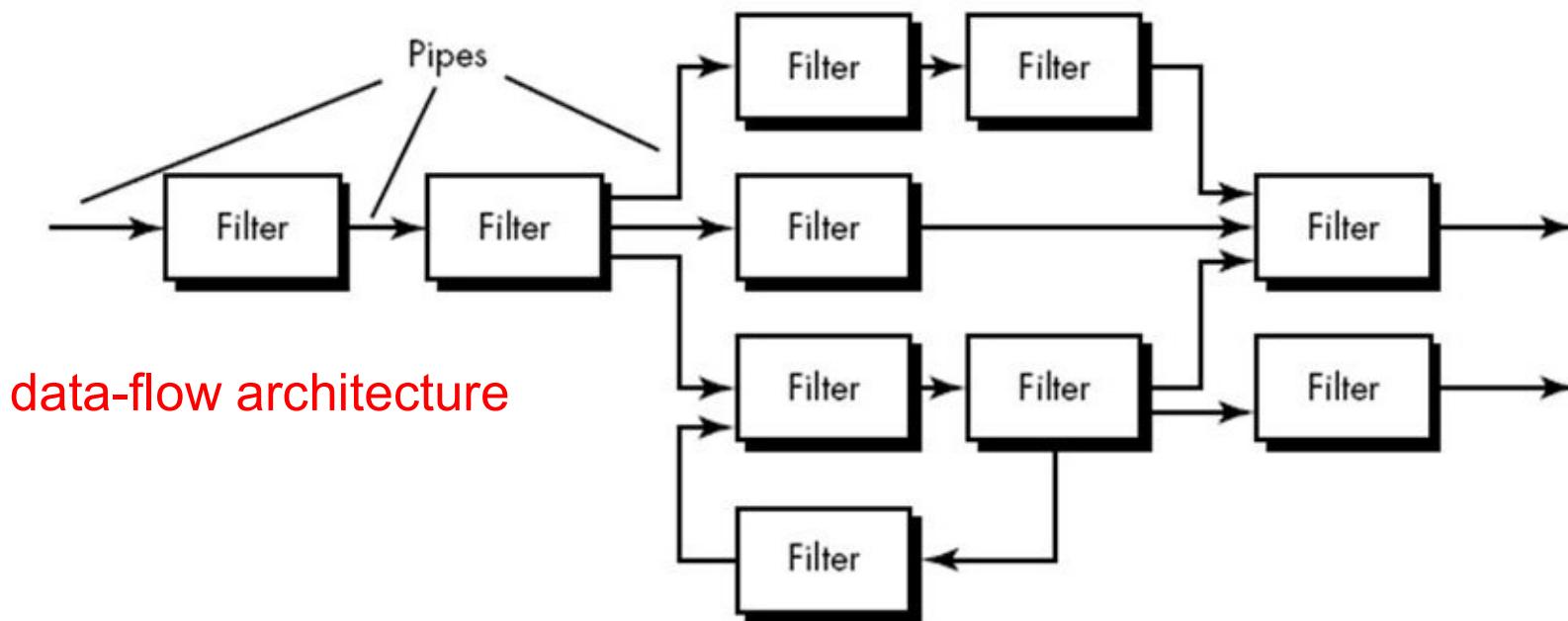
	One-to-One	One-to-Many
Synchronous	Request/response	—
Asynchronous	Notification	Publish/subscribe
	Request/async response	Publish/async responses

Some basic architectural styles

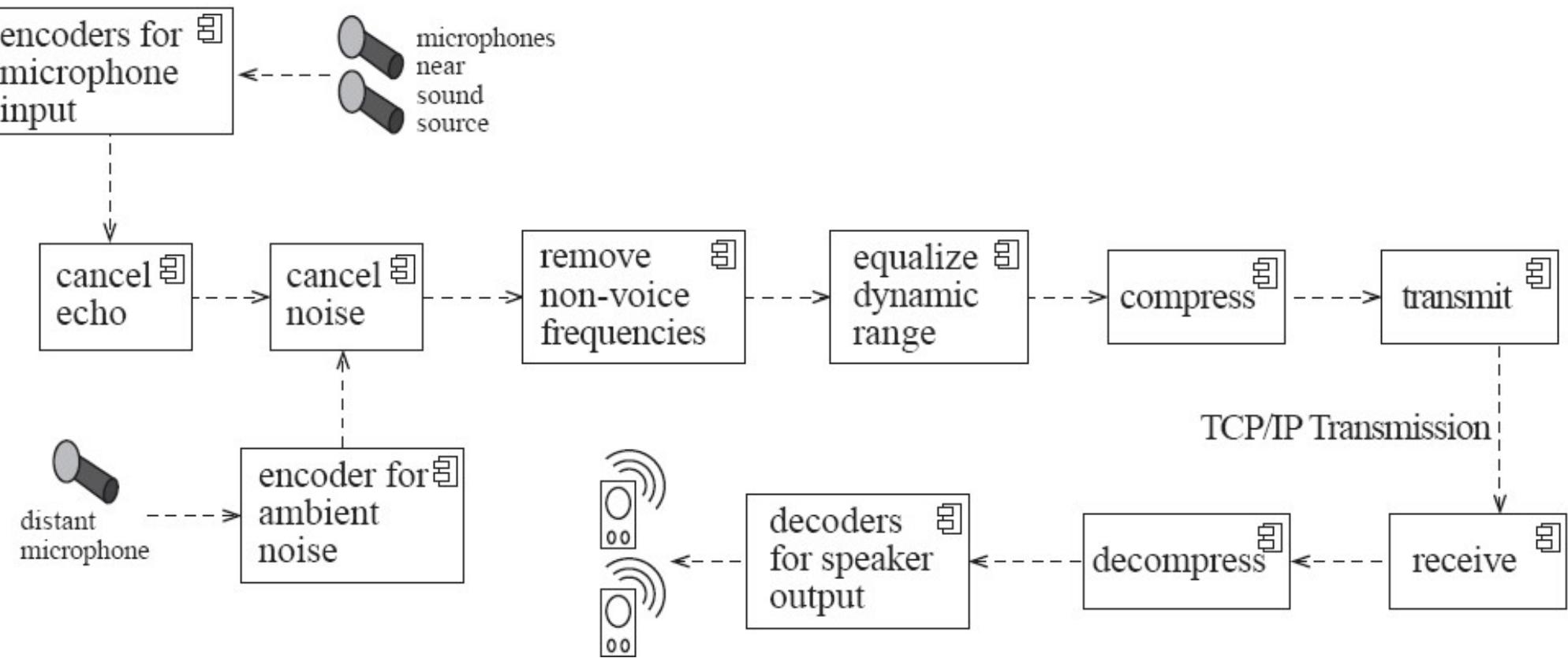
- Pipe-and-filter
- Multy-layer
- Client-server and variants
- Broker
- MVC
- Message-oriented architecture
- Service-oriented architecture
- Hexagonal architecture (or ports and adapters architecture) – used for *micro-services!*

The Pipe-and-Filter architecture

- A stream of data, in a relatively simple format, is passed through a series of processes
 - Each of which transforms it in some way
 - Data is constantly fed into the pipeline
 - The processes work concurrently
 - The architecture is very flexible
 - Almost all the components could be removed/replaced, new components could be inserted



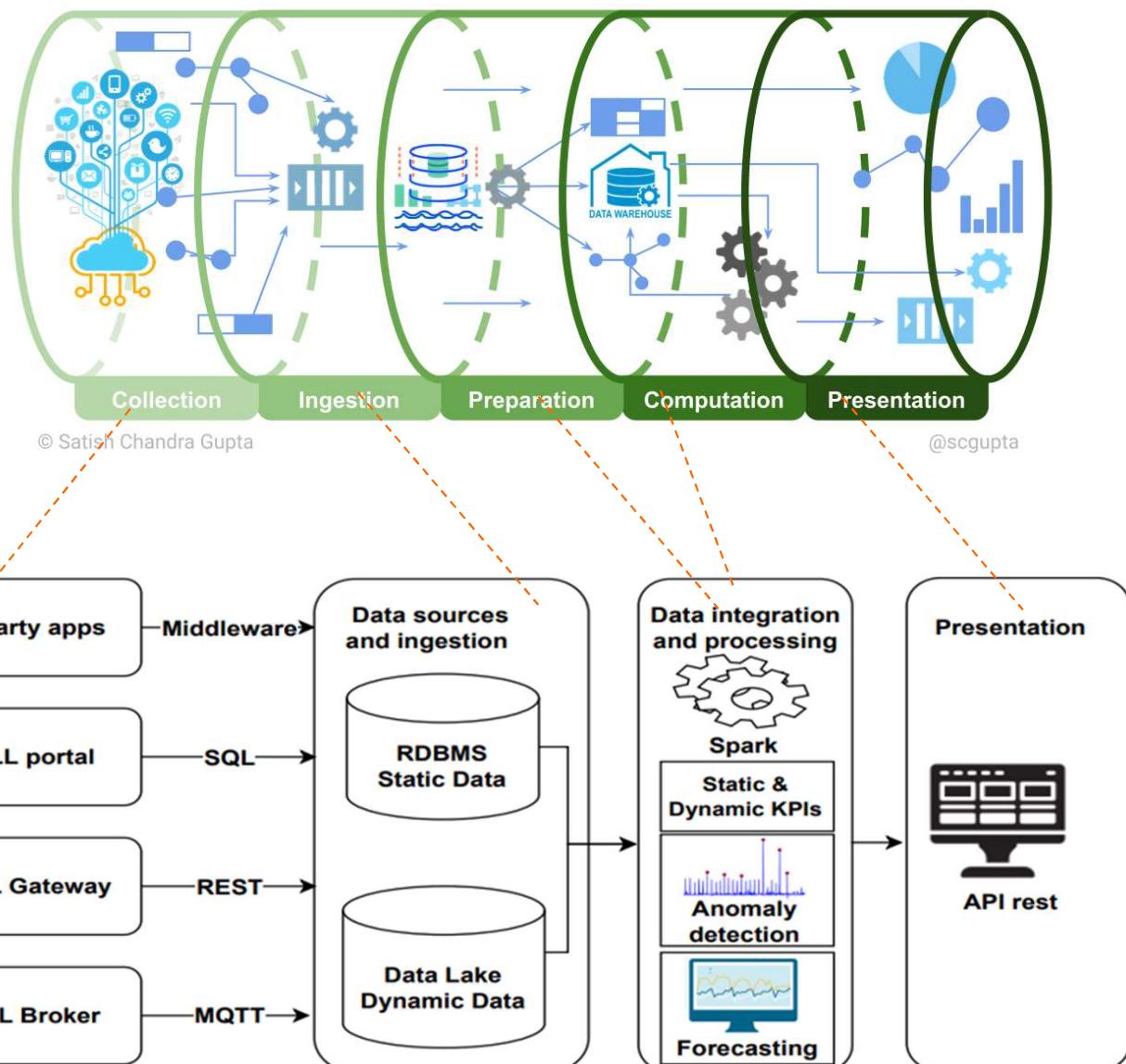
Example of a pipe-and-filter system



Big Data Pipeline on Cloud

- Pattern *Big data pipeline* has five stages
 - a *data lake* contains all data in blobs or streams
 - BLOBs (Binary Large Objects) or
 - CLOBs (Character Large Objects) or
 - files (e.g. JSON files)
- Example
 - Open source technologies around Apache Spark ecosystem

from data lake to data warehouse to analytics



ENEA PELL smart city platform for public street lighting

The pipe-and-filter architecture and design principles

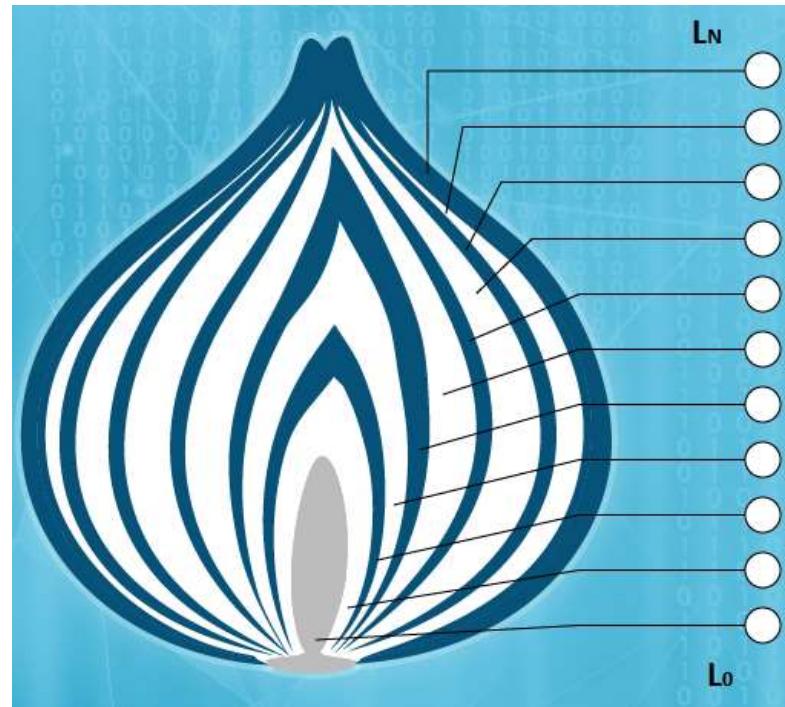
1. *Divide and conquer:* The separate **processes** can be **independently designed**
2. *Increase cohesion:* The processes have functional cohesion
3. *Reduce coupling:* The processes have only one input and one output
4. *Increase abstraction:* The pipeline components are often good abstractions, hiding their internal details
5. *Increase reusability:* The processes can often be used in many different contexts
6. *Increase reuse:* It is often possible to find reusable components to insert into a pipeline

The pipe-and-filter architecture and design principles

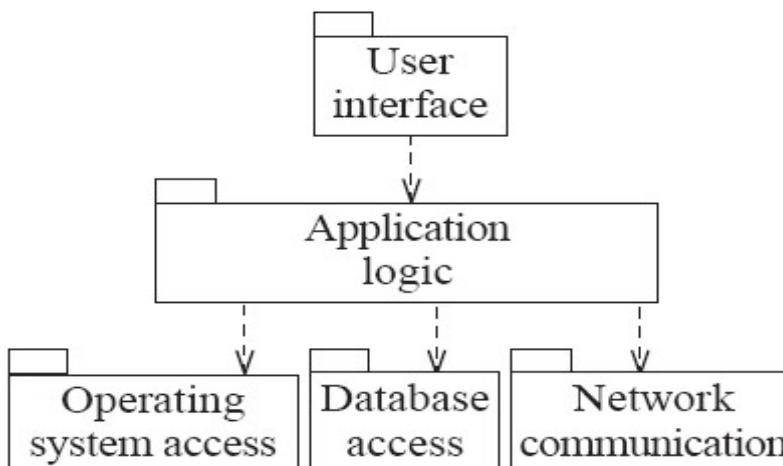
- 7. *Design for flexibility*: There are several ways in which the system is flexible
- 10. *Design for testability*: It is normally easy to test the individual processes
- 11. *Design defensively*: You rigorously check the inputs of each component, or else you can use design by contract

The Multi-Layer architecture pattern

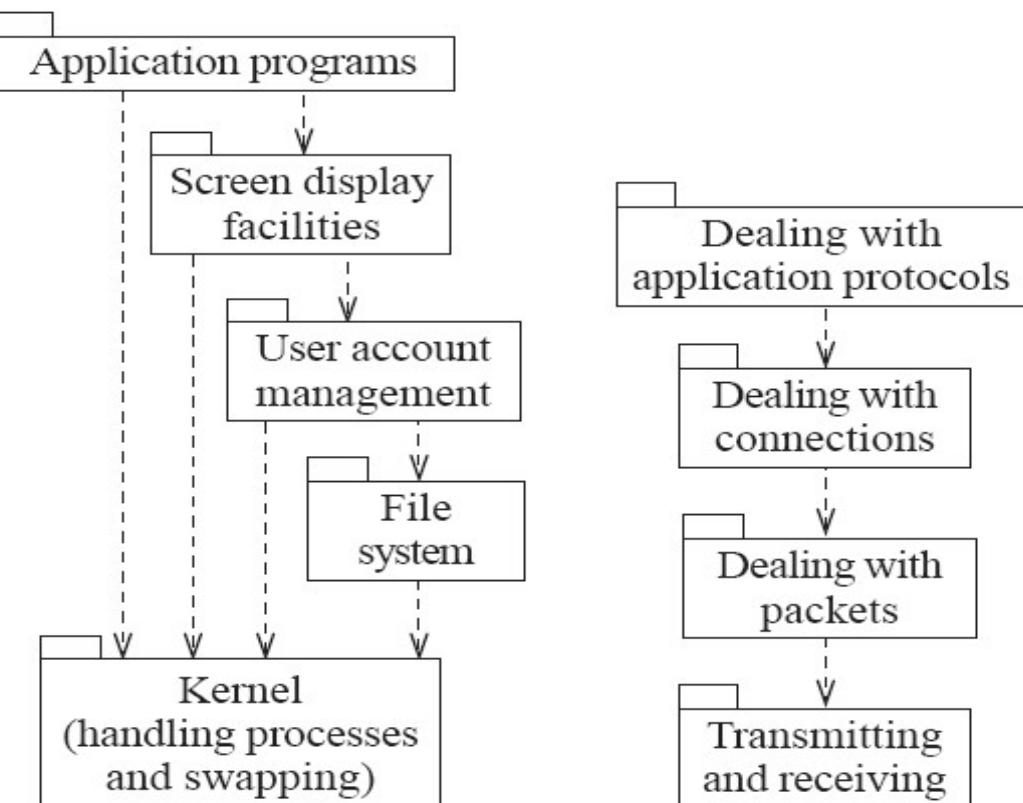
- A complex system can be built by **superposing layers at increasing levels of abstraction**
 - Each layer communicates only with the layer immediately below it
 - Each layer provides a well-defined interface (a set of services) to the layer immediately above



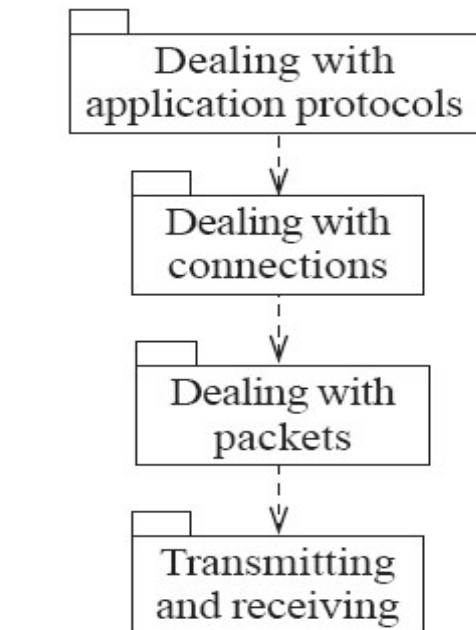
Examples of multi-layer systems



(a) Typical layers in an application program



(b) Typical layers in an operating system



(c) Simplified view of layers in a communication system

The multi-layer architecture and design principles

1. *Divide and conquer*: The **layers** can be **independently designed**
2. *Increase cohesion*: Well-designed layers have **layer cohesion**
3. *Reduce coupling*: Well-designed lower layers do not know about the higher layers and **the only connection between layers is through the API**
4. *Increase abstraction*: you do **not** need to know the **details of** how the **lower layers** are implemented
5. *Increase reusability*: The **lower layers** can often be **designed generically**

The multi-layer architecture and design principles

6. *Increase reuse*: You can often **reuse layers built by others** that provide the services you need
7. *Increase flexibility*: you can **add** new facilities built on lower-level services, or **replace higher-level layers**
8. *Anticipate obsolescence*: By isolating components in separate layers, the system becomes more resistant to obsolescence
9. *Design for portability*: All the **dependent facilities** can be **isolated in one of the lower layers**
10. *Design for testability*: **Layers can be tested independently**
11. *Design defensively*: The **APIs** of layers are natural places to build in rigorous **assertion-checking**

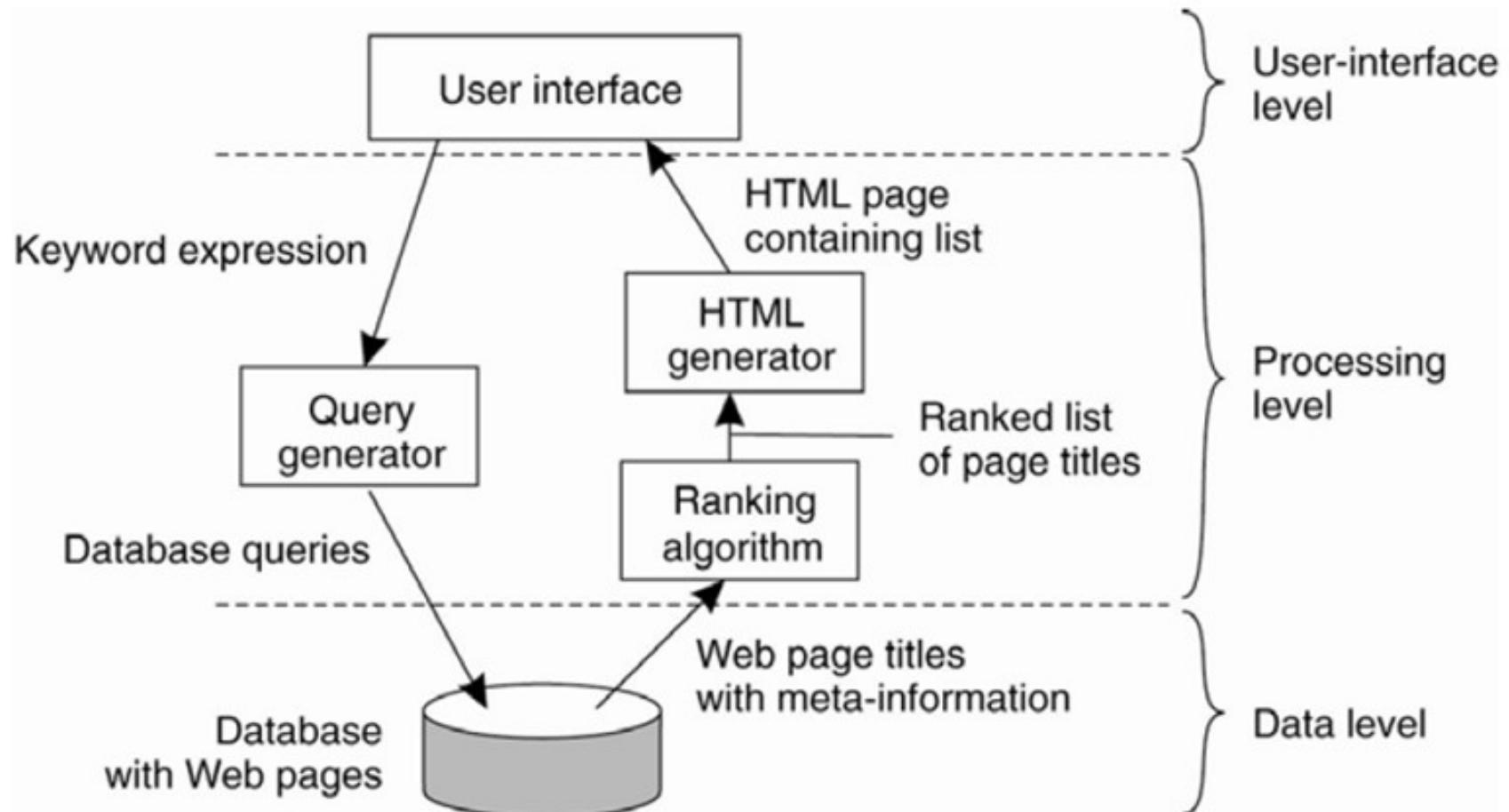
The Client-Server and variants (distributed architecture)

- There is at least one component that has the role of *server*, waiting for and then handling connections
- There is at least one component that has the role of *client*, initiating connections in order to obtain some service
- Servers do not know number or identities of clients
- Clients know server's identity
- A further extension is the ***Peer-to-Peer pattern***
 - A system composed of various software components that are distributed over several hosts
 - client-server decentralized

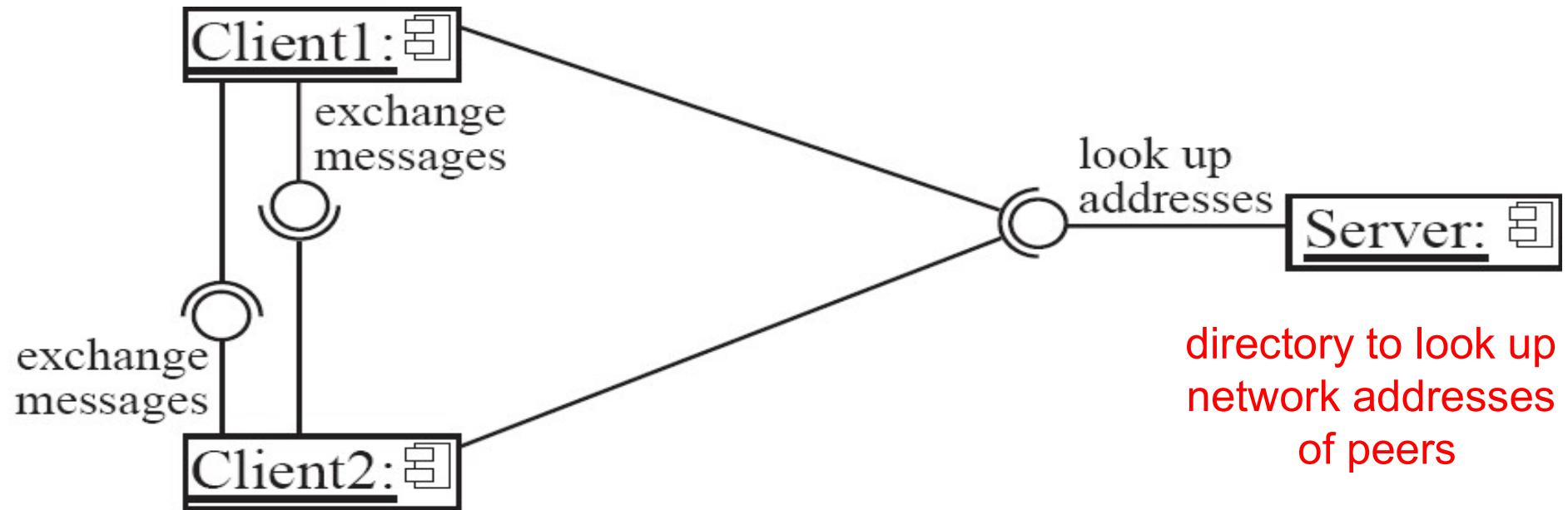
Example: web search engine

Patterns?

- layer + pipe/filter + client-server

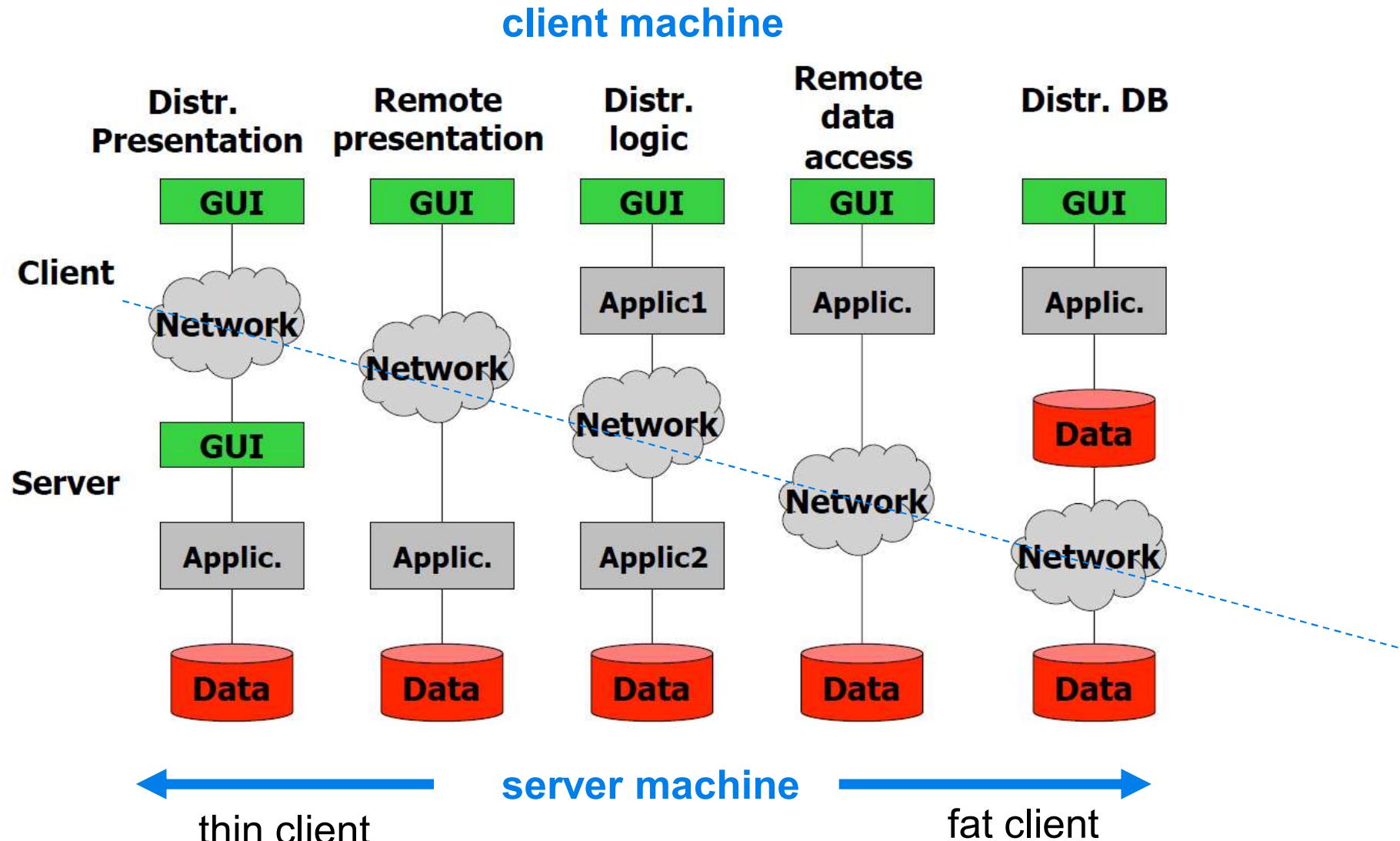


Hybrid peer-to-peer style



Two-tier architecture

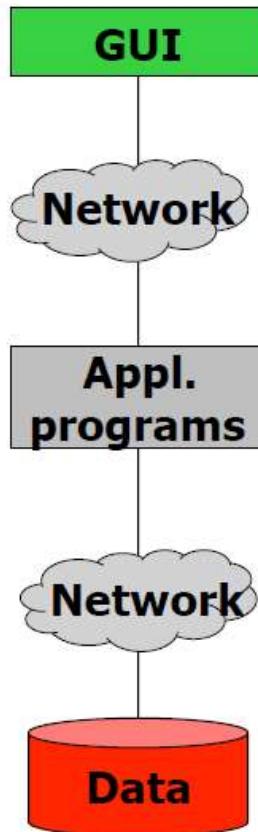
- Two tier (*physical levels*): client/server on separate machines
- Three layers (*logical levels*): GUI, Application processing, Data
- Different types of two-tier styles



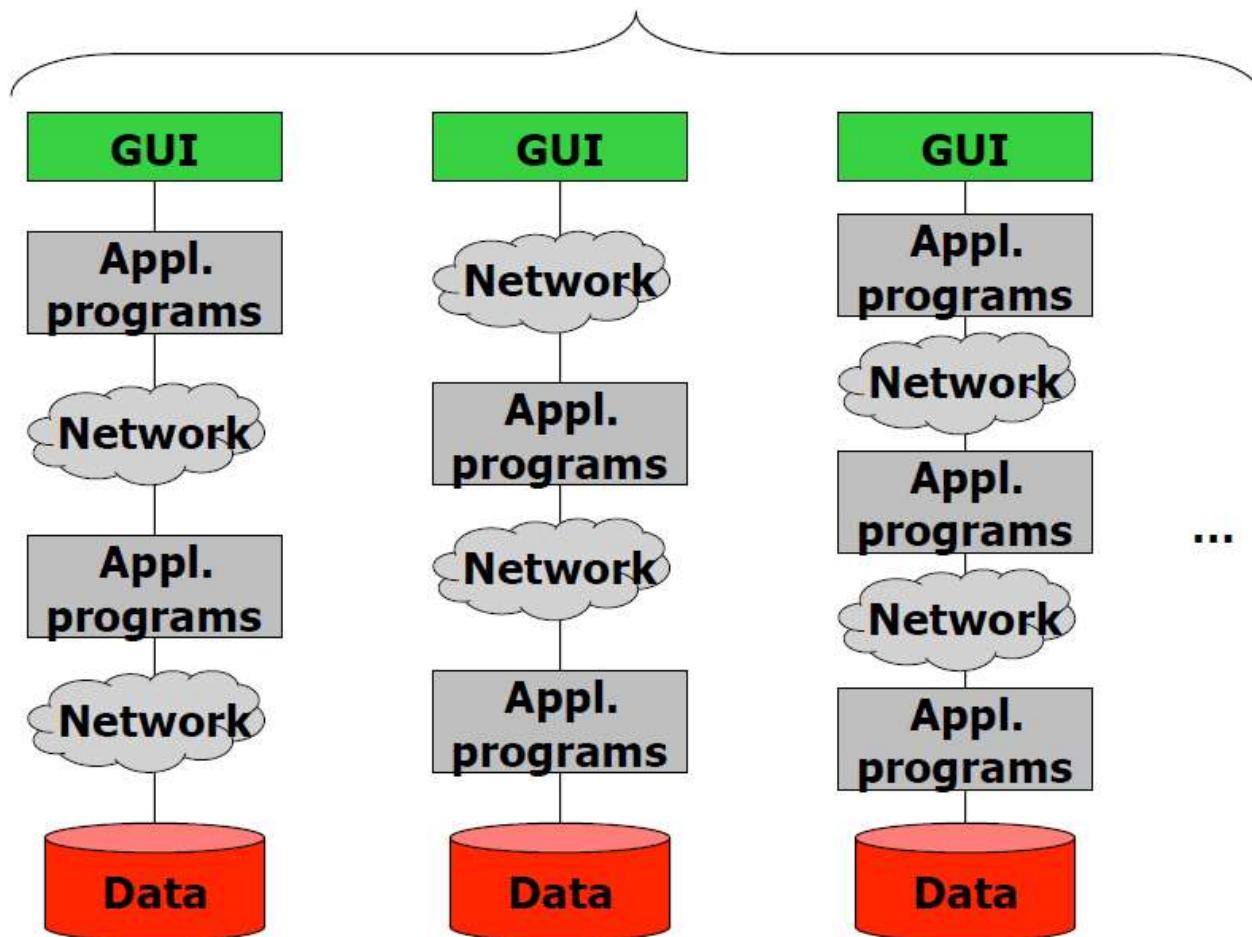
Three-tier architecture (and N-tier architecture)

- Each layer (Presentation, Application and Data) on a separate machine

Typical scheme



Alternative schemes



The distributed architecture and design principles

1. *Divide and conquer:* Dividing the system into **client and server processes** is a strong way to divide the system
 - Each can be **separately developed**
2. *Increase cohesion:* The server can provide a **cohesive service to clients**
3. *Reduce coupling:* There is usually only one communication channel **exchanging simple messages**
4. *Increase abstraction:* Separate distributed components are often good abstractions
6. *Increase reuse:* It is often possible to find **suitable frameworks** on which to build good distributed systems

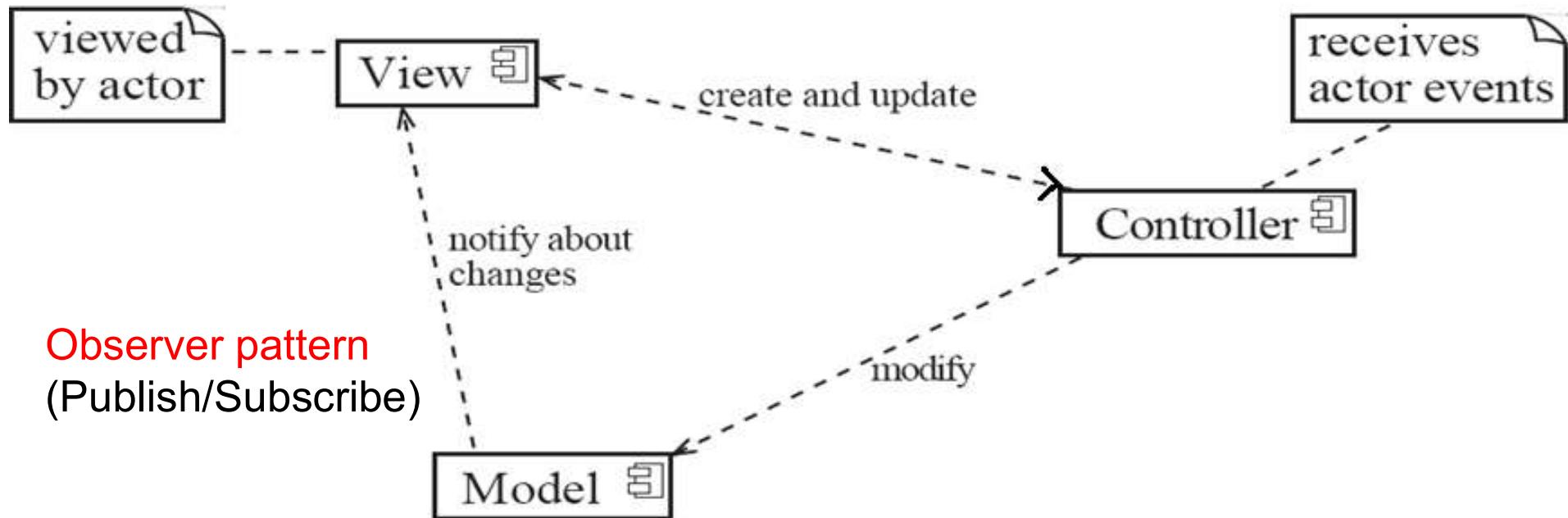
The distributed architecture and design principles

- 7. *Design for flexibility*: Distributed systems can often be easily reconfigured by adding extra servers or clients
- 9. *Design for portability*: You can write clients for new platforms without having to port the server
- 10 *Design for testability*: You can test clients and servers independently
- 11. *Design defensively*: You can put rigorous checks in the message handling code

The Model-View-Controller (MVC) architecture

- An architectural pattern used to help **separate the user interface layer from other parts of the system**
 - The ***model*** contains the underlying classes whose instances are to be viewed and manipulated
 - The ***view*** contains objects used to render the appearance of the data from the model in the user interface
 - The ***controller*** contains the objects that control and handle the user's interaction with the view and the model
 - The ***Observable design pattern*** is normally used to separate the **model from the view**

Example of the MVC architecture for the UI



- **Triangular interactions**
- Passive model (“Pull”)
 - the View layer “pull” results from the (multiple) controller
- Active model (“Push”)
 - Model reports state changes to view(s) through an Observer pattern (*Publish / Subscribe*)

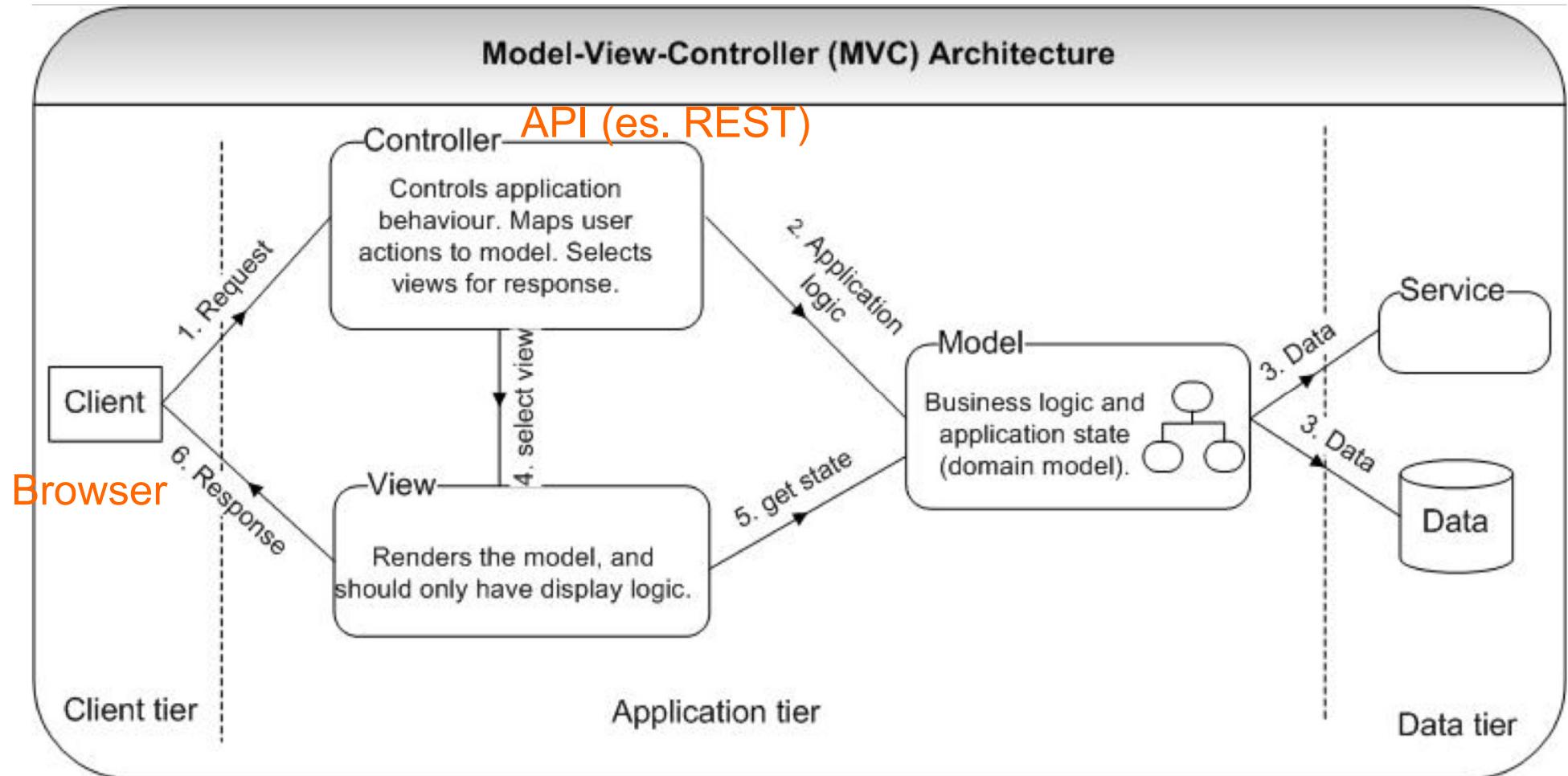
The MVC architecture and design principles

1. *Divide and conquer:* The three **components** can be somewhat **independently designed**
2. *Increase cohesion:* The components have stronger layer cohesion than if the view and controller were together in a single UI layer
3. *Reduce coupling:* The communication channels between the three components are minimal
6. *Increase reuse:* The view and controller normally make extensive use of reusable components for various kinds of UI controls
7. *Design for flexibility:* It is usually quite easy to **change the UI by changing the view, the controller, or both**
10. *Design for testability:* You can test the application separately from the UI

Comparison with the three-tier architecture

- Conceptually the **three-tier architecture is linear**:
 - A fundamental rule in a three tier architecture is the client tier never communicates directly with the data tier
 - in a three-tier model all communication must pass through the middle tier
- The **MVC architecture is triangular**:
 - the view sends updates to the controller
 - the controller updates the model, and
 - the view gets updates directly from the model (*push*) or from the controller (*pull*)

Typical web application style: MVC on a three-tier architecture



The Broker architectural pattern

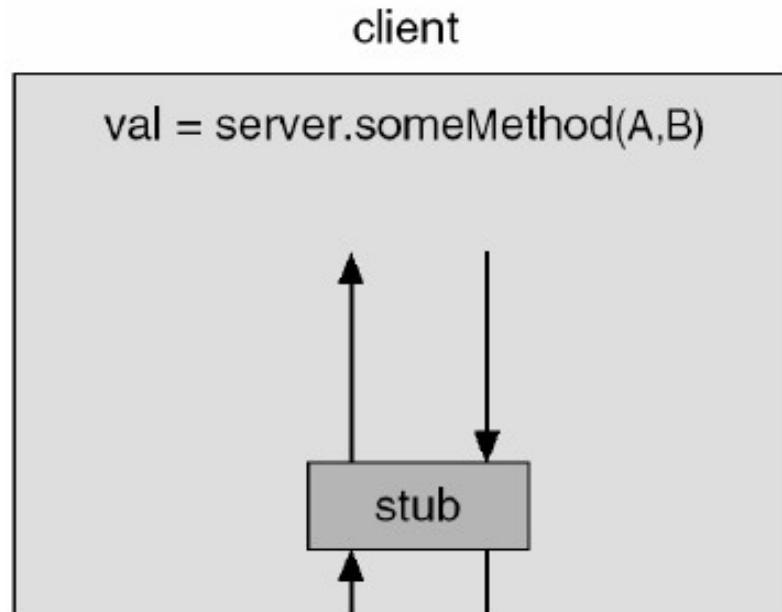
A broker to decouple clients and servers!



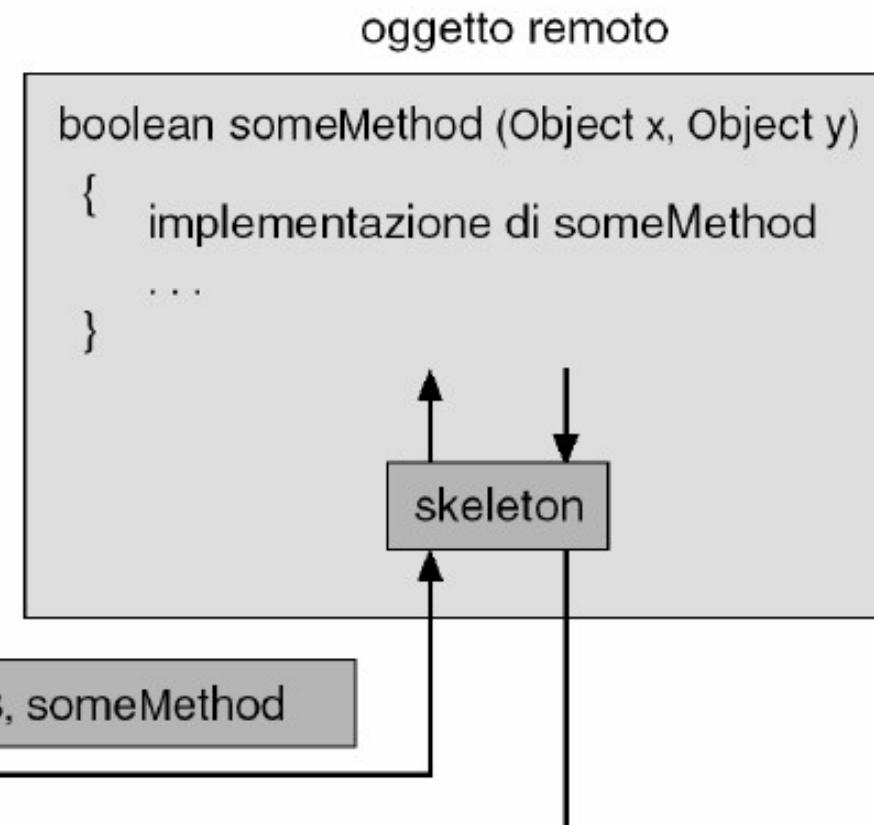
- It is used to structure distributed systems with decoupled components
 - Servers publish their capabilities (services and characteristics) to a broker
 - Clients request a service from the broker, and the broker then redirects the client to a suitable service from its registry.
- Based on **connectors of remote invocation**: an object can call methods of another object without knowing that this object is remotely located
 - Examples of implementation technology:
 - OMG open standard CORBA (Common Object Request Broker Architecture)
 - RPC (Remote Procedure Call), Java RMI (Remote Method Invocation)
 - gRPC for remote service invocations **NEW!**
 - (*Services not objects, Messages not References*)

Example: Remote Method Invocation (RMI)

JVM –client side



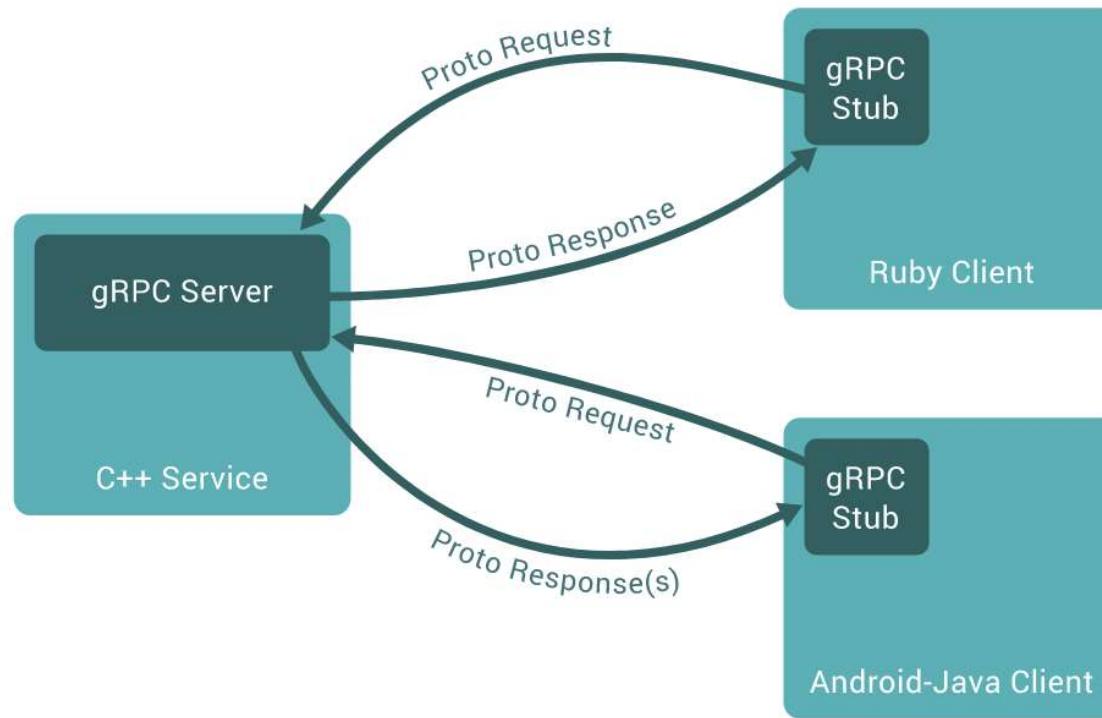
JVM –server side (plays the broker role!)



valore booleano di ritorno

Another example: gRPC (gRPC Remote Procedure Calls)

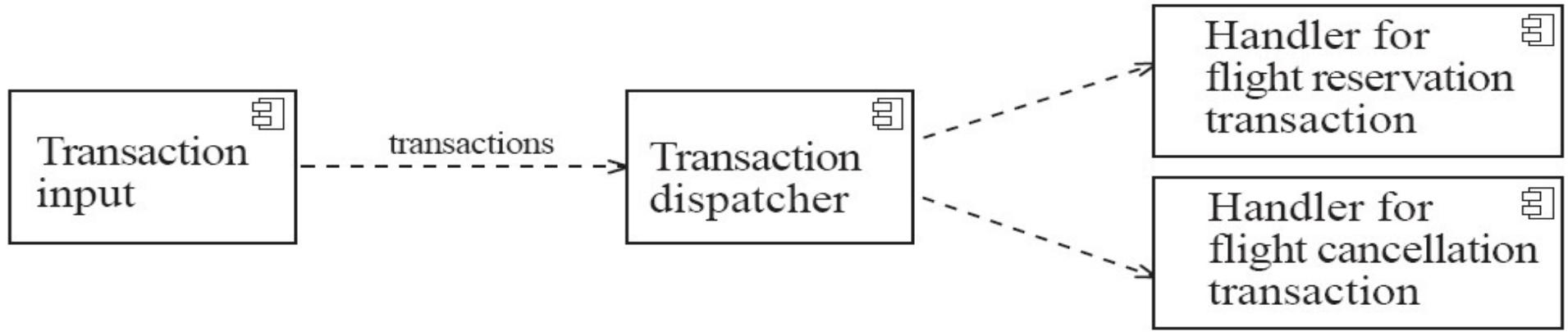
- Open and general-purpose **RPC infrastructure by Google** for connecting services
- It uses HTTP/2 for transport, *Protocol Buffers* as IDL
- It provides features such as authentication, bidirectional streaming and flow control, blocking or non-blocking bindings, cancellation and timeouts



The broker architecture and design principles

1. *Divide and conquer:* The **remote objects can be independently designed**
5. *Increase reusability:* It is often possible to design the remote objects so that other systems can use them too
6. *Increase reuse:* You may be able to reuse remote objects that others have created
7. *Design for flexibility:* **Brokers can be updated as required, or the proxy can communicate with different remote objects**
9. *Design for portability:* You can write clients for new platforms while still accessing brokers and remote objects on other platforms
11. *Design defensively:* You can provide careful assertion checking in the remote objects

The Transaction-Processing architectural pattern



- A process reads a series of inputs one by one
 - Each input describes a *transaction* – a command that typically some change to the data stored by the system
 - There is a **transaction dispatcher component that decides what to do with each transaction**
 - This dispatches a procedure call or message to one of a series of component that will *handle* the transaction

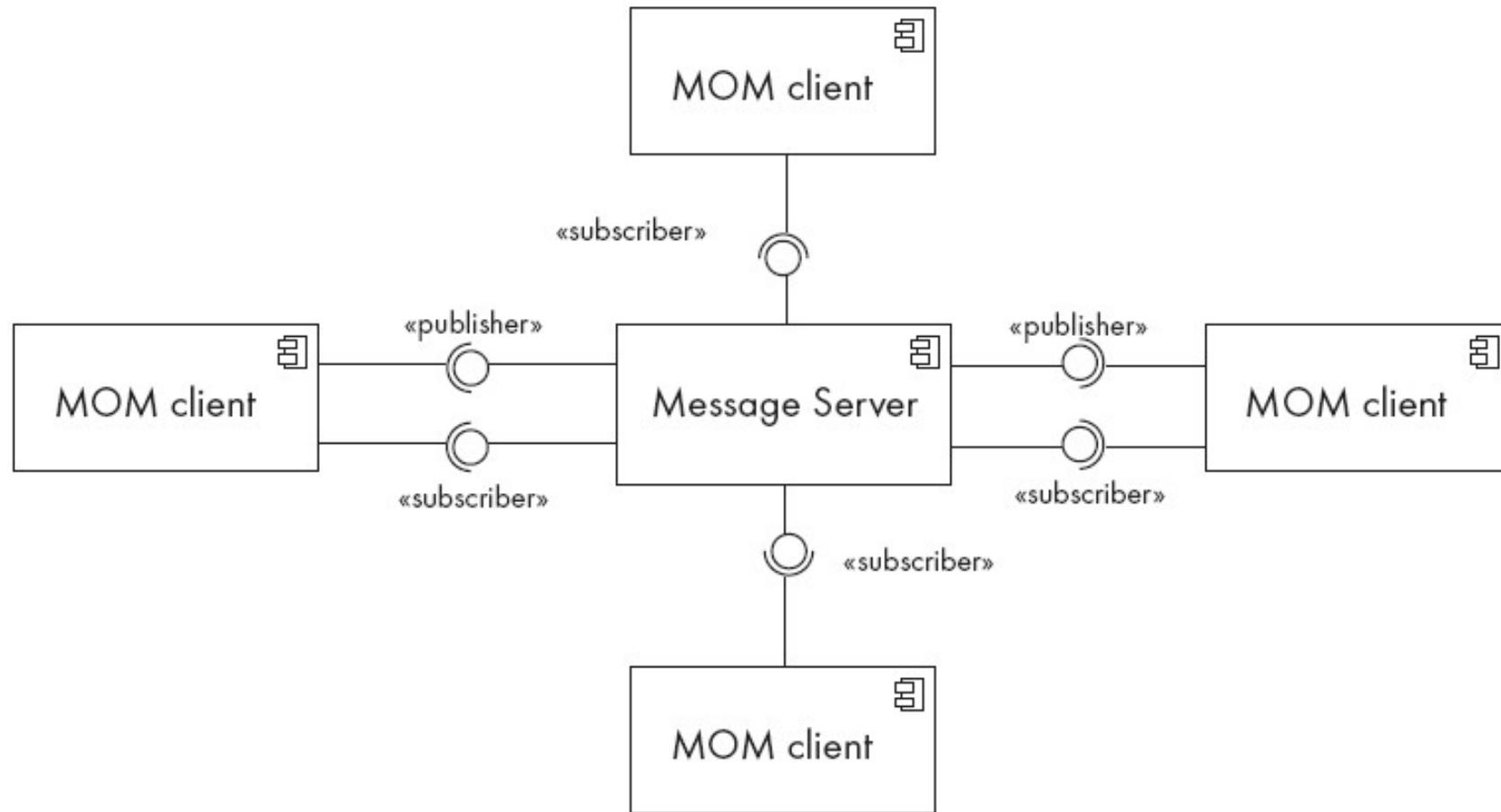
The transaction-processing architecture and design principles

1. *Divide and conquer:* The transaction handlers are suitable system divisions that you can give to separate software engineers
2. *Increase cohesion:* Transaction handlers are naturally cohesive units
3. *Reduce coupling:* **Separating the dispatcher from the handlers** tends to reduce coupling
7. *Design for flexibility:* You can readily add new transaction handlers
11. *Design defensively:* You can add assertion checking in each transaction handler and/or in the dispatcher

The Message-oriented architectural pattern

- Different sub-systems communicate and collaborate only by exchanging messages even when the destination is not available
 - Also known as **Message-oriented Middleware (MOM)**
 - Senders and receivers need only to know what are the message formats
 - In addition, the communicating applications do not have to be available at the same time (*asynchronous model, publish/subscribe*), messages can be made persistent
- Technology examples:
 - Java Message Service (o JMS) of Java EE
 - allow Java applications to exchange messages
 - Google Cloud Messaging (GCM), superseded by Google Firebase Cloud Messaging (FCM)
 - Roscore for Robot Operating System (ROS)-based systems
 - Standard ISO MQTT (MQ Telemetry Transport or Message Queue Telemetry Transport) on TCP/IP

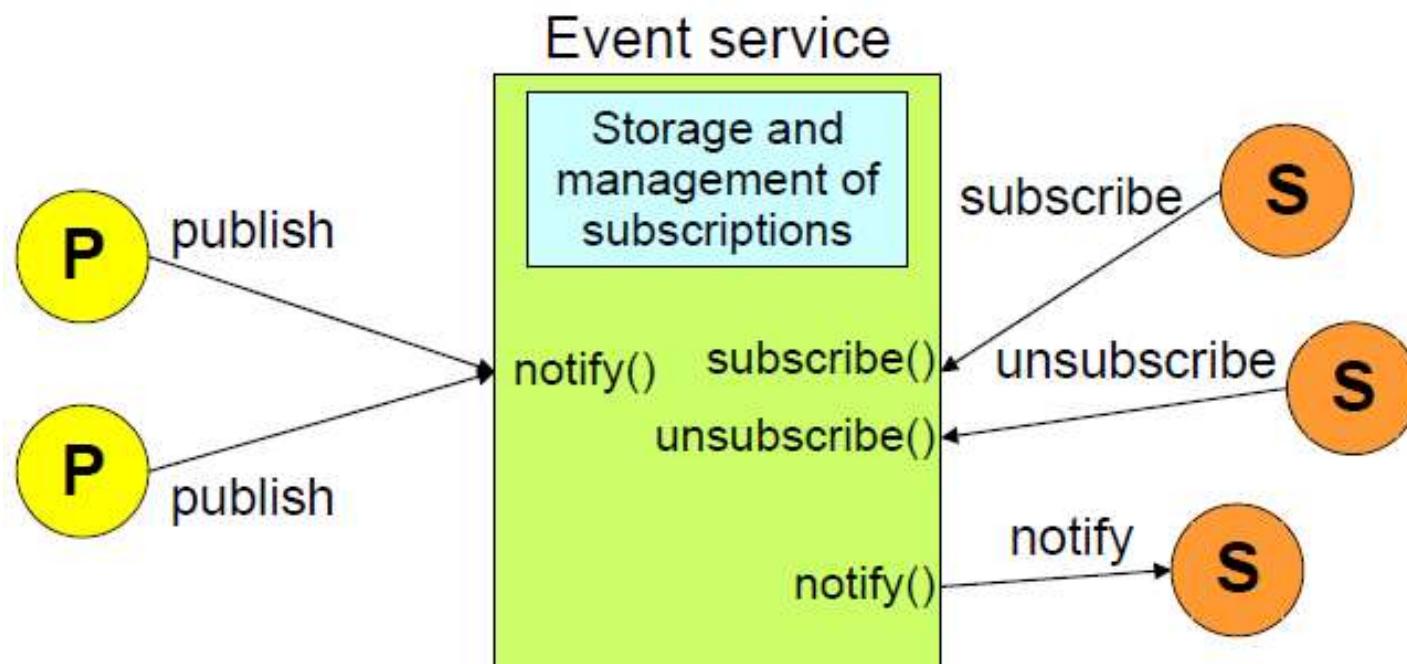
Message-oriented architecture



Publish-subscribe (or event) interaction style

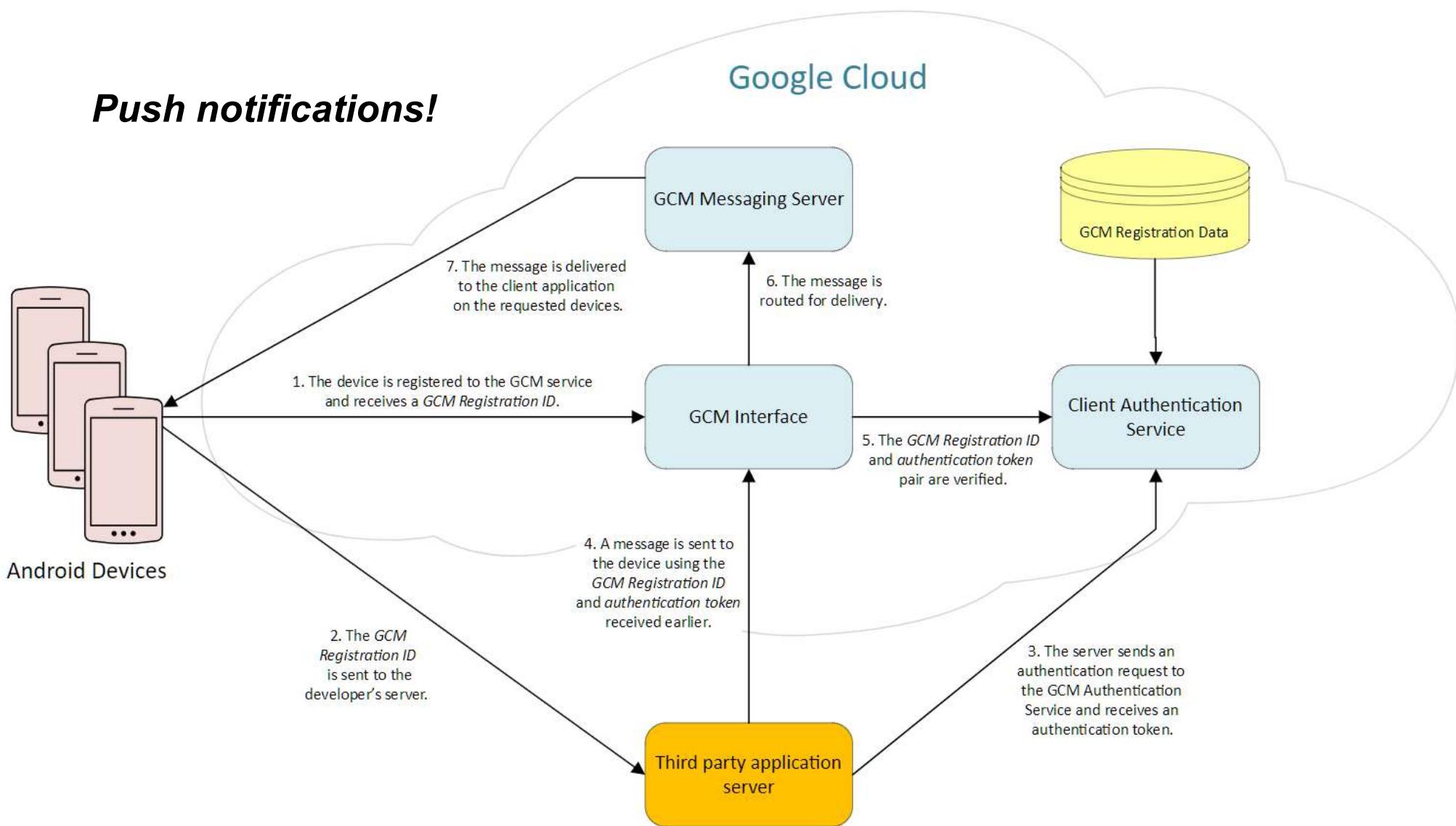
- Messages are sent by one component (the *publisher*) through virtual channels (*topics*) to which other interested software components can subscribe (*subscribers*)

Riferimento: P.T. Eugster *et al*, "The many faces of publish/subscribe"
ACM Comput. Surv. 35(2):114-131, June 2003.



Example: Google Cloud Messaging (GCM)

– superseded by Google's Firebase Cloud Messaging (FCM)



The Message-oriented architecture and design principles

1. *Divide and conquer*: The application is made of isolated software components
3. *Reduce coupling*: The **components are loosely coupled** since they **share only messages** using data interchange formats
4. *Increase abstraction*: The prescribed format of the messages are generally simple to manipulate, all the application details being hidden behind the messaging system
5. *Increase reusability*: A component will be reusable if the message formats are flexible enough
6. *Increase reuse*: The components can be reused as long as the new system adhere to the proposed message formats

The Message-oriented architecture and design principles

- 7. *Design for flexibility*: The functionality of a message-oriented system can be easily updated or enhanced by adding or replacing components in the system
- 10. *Design for testability*: Each component can be tested independently
- 11. *Design defensively*: Defensive design consists simply of validating all received messages before processing them.

The Service-oriented architectural pattern

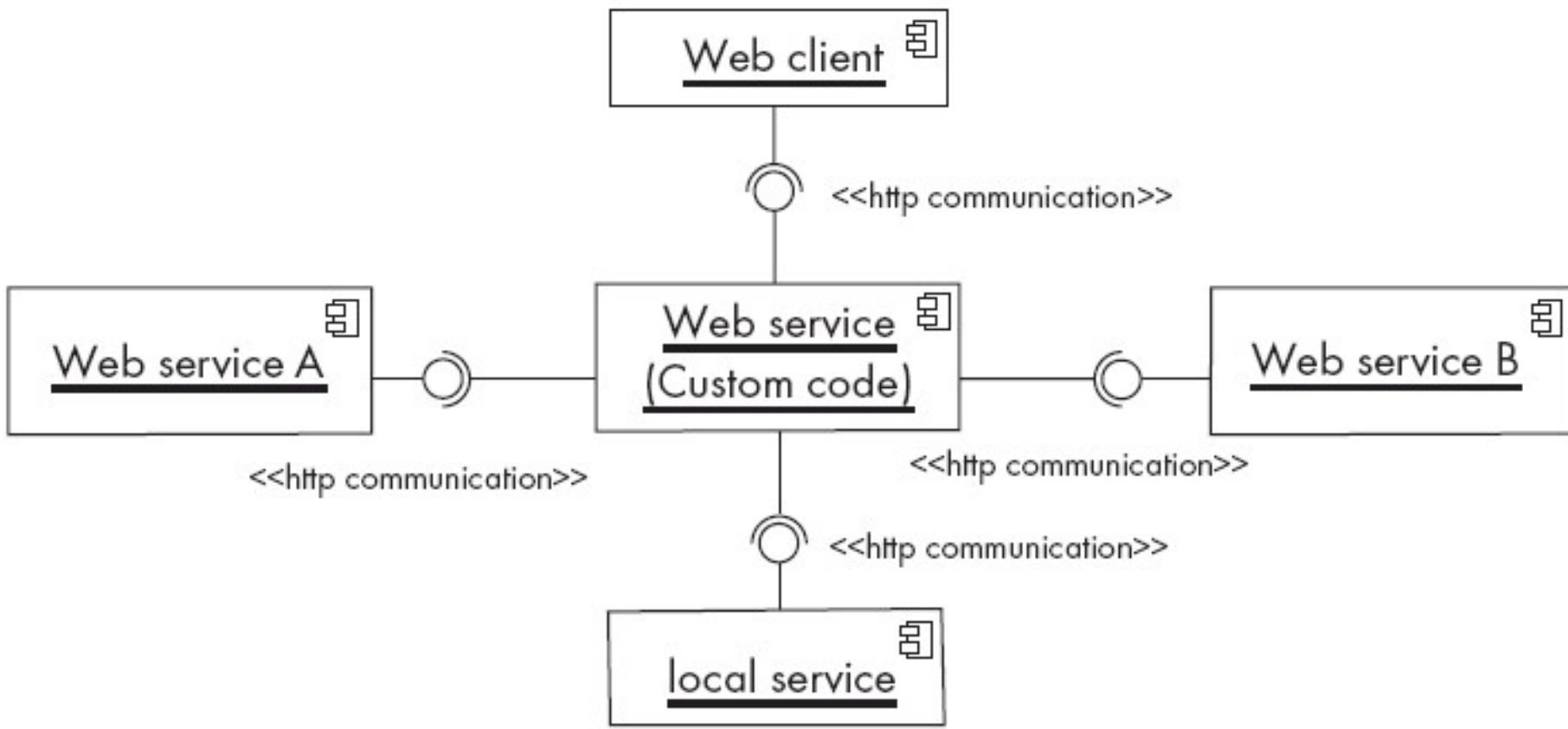
- An application as **an assembly of services**
- Services are **coarse-grained components** that fulfill a specific *business requirement with well-defined quality of service (QoS)*
 - *loosely-coupled* and available across a network
 - communicate using well-defined interfaces and open standards
 - such as XML or JSON for message interchange format and
 - message-based communication protocols over http (SOAP, REST, etc.)
 - not tied to any operating system or programming language!
 - may be composed dynamically by an *enterprise service bus (ESB)* and/or a *workflow engine* (such as BPEL or a BPMN workflow tool)
 - service *orchestration/choreography*

The Service-oriented architectural pattern

- In the context of Internet, services are called *Web services*
 - A web service is a service accessible through the Internet or private (intranet) networks
- Web services are self-contained, modular, distributed, dynamic applications that
 - can be described, published, located, or invoked over the Internet to
 - create products, processes, and supply chains
- Web services are built on top of open standards such as TCP/IP, HTTP, Java, HTML, and XML

Example of SA of a web service application

end-to-end *request-response* interaction model



The Service-oriented architecture and design principles

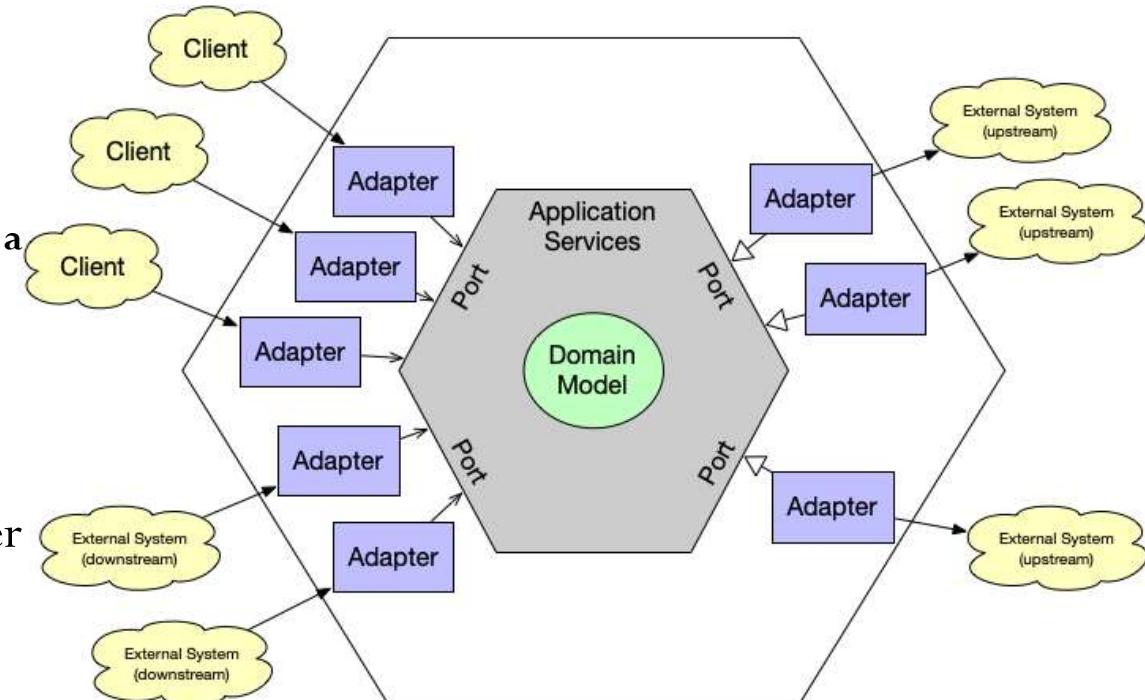
1. *Divide and conquer*: The application is made of independently designed services
2. *Increase cohesion*: The Web services are structured as layers and generally have good functional cohesion
3. *Reduce coupling*: Web-based applications are loosely coupled built by binding together distributed components
5. *Increase reusability*: A Web service is a highly reusable component
6. *Increase reuse*: Web-based applications are built by reusing existing Web services
8. *Anticipate obsolescence*: Obsolete services can be replaced by new implementation without impacting the applications that use them

The Service-oriented architecture and design principles

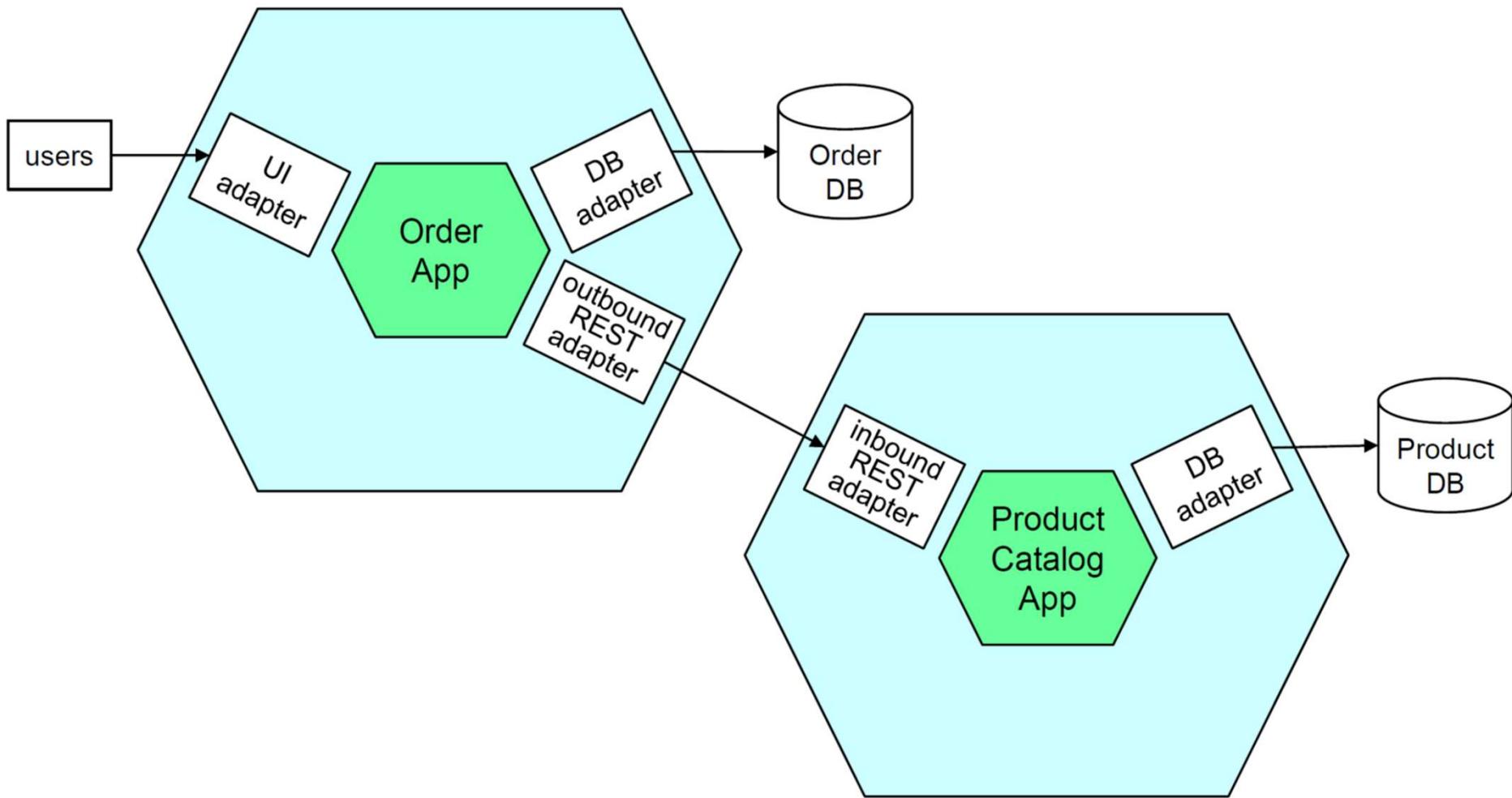
9. *Design for portability*: A service can be implemented on any platform that supports the required standards
10. *Design for testability*: Each service can be tested independently
11. *Design defensively*: Web services enforce defensive design since different applications can access the service

The hexagonal architecture (ports and adapters architecture)

- Alternative to the layered architecture; at the **origin of the microservices architecture**
- The system is divided into **several loosely-coupled components (hexagons!)**
- Components are connected through
 - **ports (*abstract API*)**
 - **adapters**
 - the glue between ports and the outside world
 - **allow interaction through a port using a particular communication technology/connector**
- There can be **several adapters for one single port**
 - Example: data can be provided by a user through a GUI or a command-line interface or an API-controller



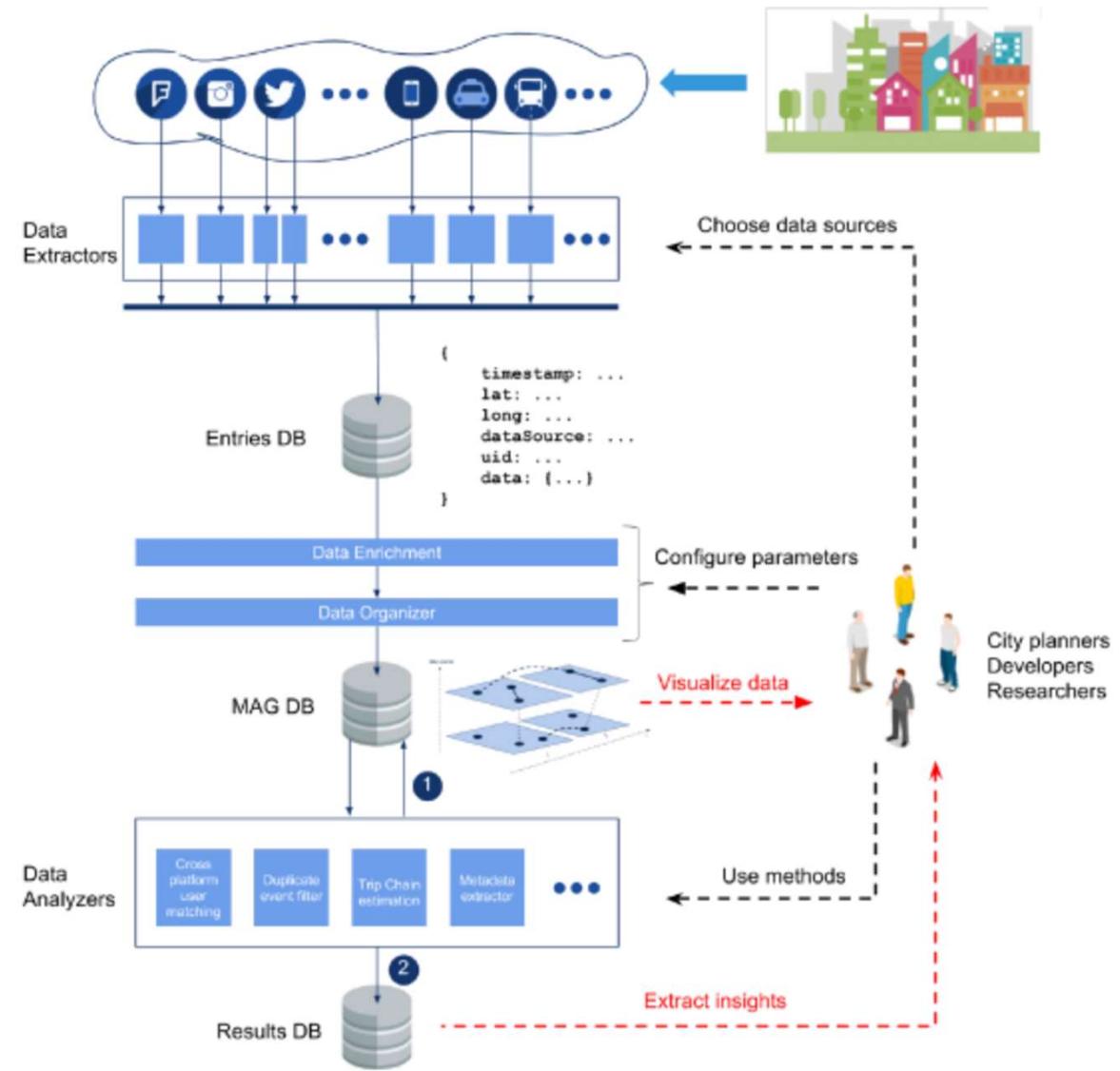
Example: two (hexagonal) apps communicating over REST



Exercise 1

- Which architectural design patterns/styles do you recognize?

SMAFramework: Urban Data Integration Framework for Mobility Analysis in Smart Cities





UNIVERSITÀ
DEGLI STUDI
DI BERGAMO

Dipartimento
di Ingegneria Gestionale,
dell'Informazione e della Produzione



Sviluppo di architetture a (micro-)servizi e API- led

PROGETTAZIONE, ALGORITMI E
COMPUTABILITÀ
(38090-MOD1)

Corso di laurea
Magistrale in
Ingegneria
Informatica

RELATORE
Prof.ssa Patrizia
Scandurra

SEDE
DIGIP

Argomenti

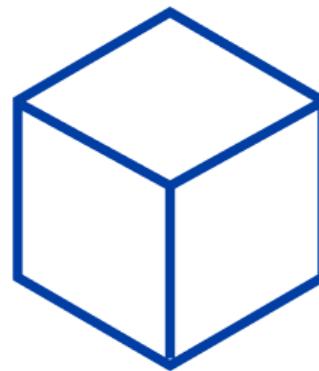
- Architettura a microservizi
 - Definizioni
 - Design pattern architetturali per micro-servizi
 - Implementazione di architetture a (micro-)servizi mediante REST/http e il framework Java Spring
- Pattern architetturali API-led (*in Inglese!*)

Riferimenti aggiuntivi

- Walls, C. **Spring in Action**, fifth edition. Manning, 2019.
 - Chapter 6, Creating REST services
 - Chapter 7, Consuming REST services
- JSON

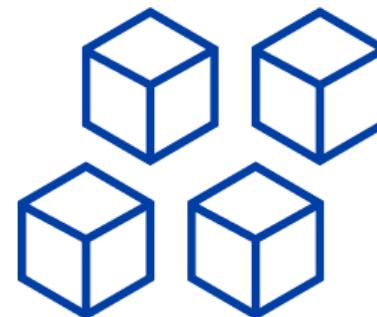
<https://www.json.org/>

Architetture a (micro-)servizi



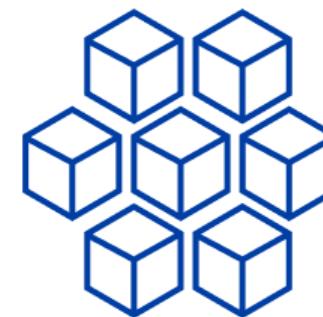
MONOLITHIC

Single unit



SOA

Coarse-grained



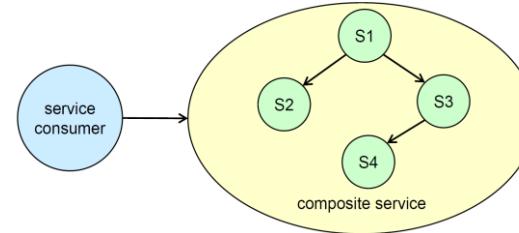
MICROSERVICES

Fine-grained

Caratteristiche dei micro-servizi

I microservizi sono servizi a grana “fine”

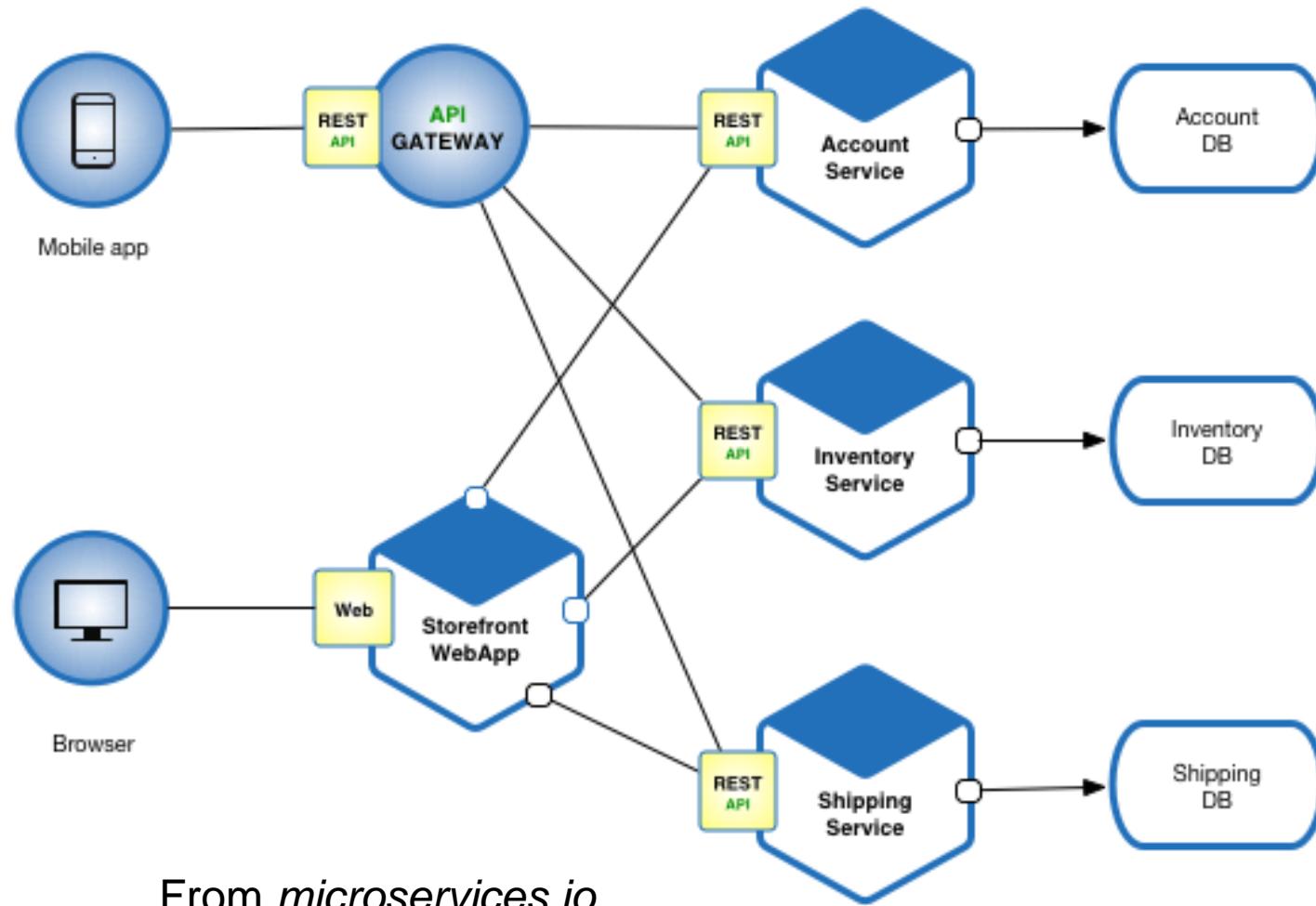
- Come servizi: contratto e astrazione, accoppiamento debole, scopribilità (*service discovery*), componibilità e riusabilità, ecc..
- In aggiunta:



Caratteristica	Spiegazione
Autocontenuti	Non hanno dipendenze esterne; gestiscono i loro dati e implementano la loro interfaccia utente
Leggeri	Comunicano attraverso protocolli leggeri, in modo da ridurre l’overhead della comunicazione
Indipendenti dall’implementazione	Possono essere implementati usando linguaggi di programmazione diversi e possono usare tecnologie diverse (ad es. diversi tipi di DB)
Rilasciabili in modo indipendente	Eseguibili in un proprio processo e rilasciabili in modo indipendente mediante <i>container</i> e sistemi automatici di <i>deployment / delivery continuo</i>
Orientati al business	Rappresentano una singola e specifica capacità/responsabilità di business (ad es. in un e-commerce: gestione degli ordini, la gestione dell’inventario e le spedizioni)

Esempio di architettura a microservizi

- Il principale vantaggio dell'architettura dei microservizi è il ridimensionamento dei requisiti di business di un servizio

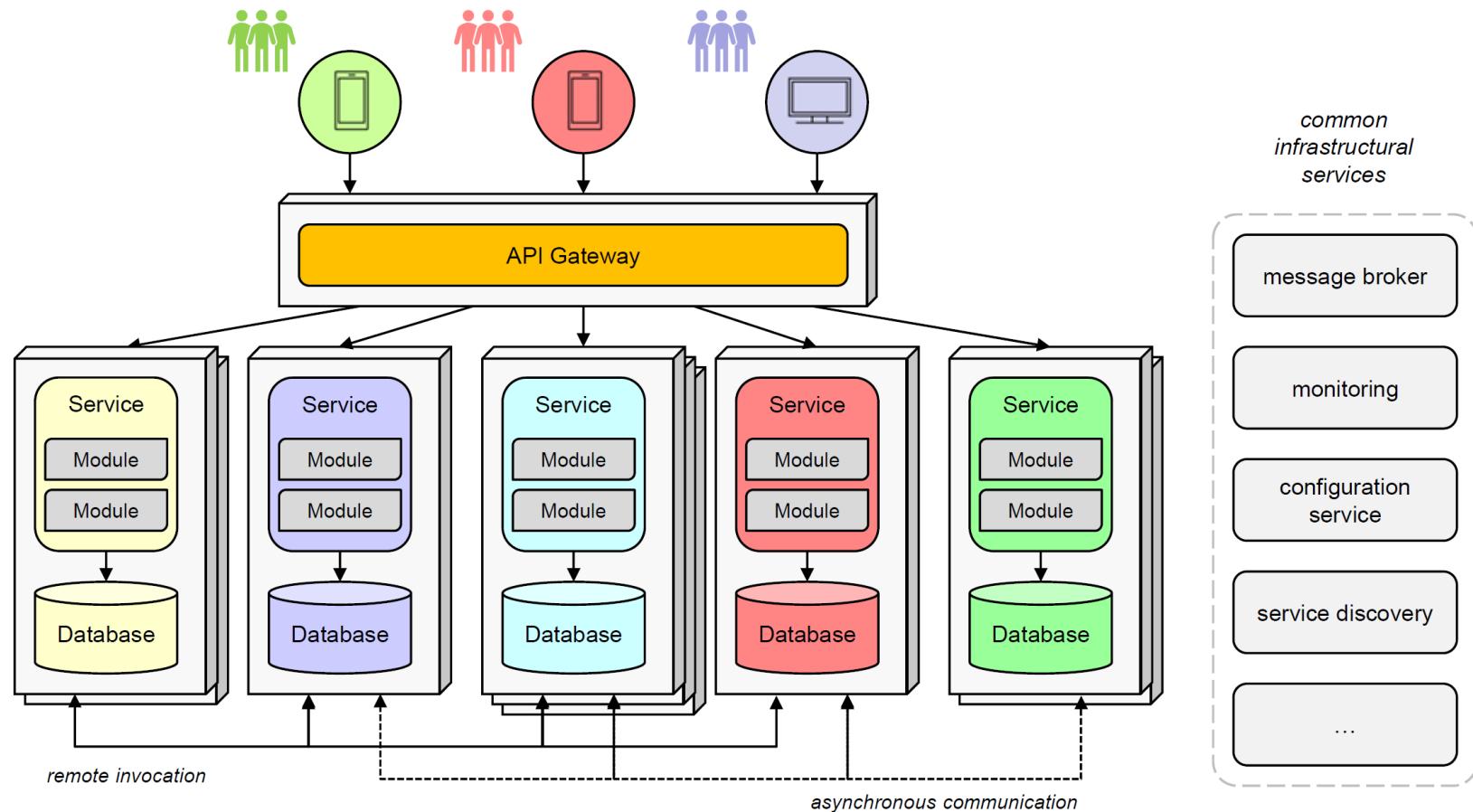


Principali vantaggi dei micro-servizi

- **Ridimensionamento:** ogni servizio può essere ridimensionato individualmente e indipendentemente senza riguardare gli altri requisiti di business
- **Isolamento dei guasti e alta manutenibilità** i micro-servizi sono lasciamente accoppiati e di conseguenza anche i bug/guasti
- **Indipendenza dallo stack tecnologico:** ogni micro-servizio può essere sviluppato in qualsiasi linguaggio di programmazione, indipendentemente dagli altri microservizi con cui comunica via API
- **Sicurezza:** i microservizi encapsulano dati sensibili al loro interno, e il loro accesso avviene in modo protetto con API progettate ad-hoc

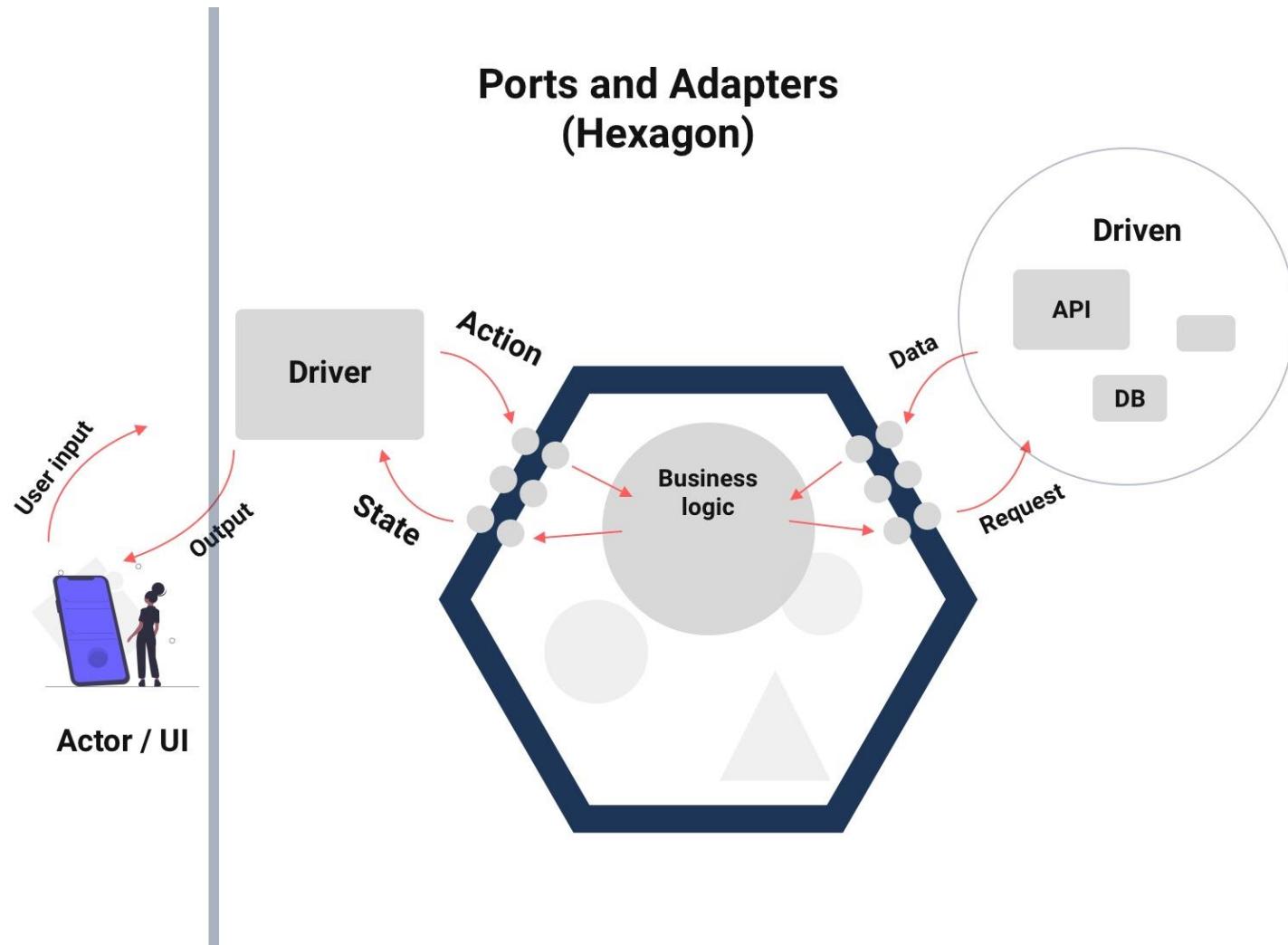
Modello concettuale

- Un API gateway è un servizio (lato server) che fornisce un punto di accesso unificato ai micro-servizi



Design pattern per (micro-)servizi

- Design pattern **esagonale** (*a porte-adattatori*)
 - Un micro-servizio come un esagono!



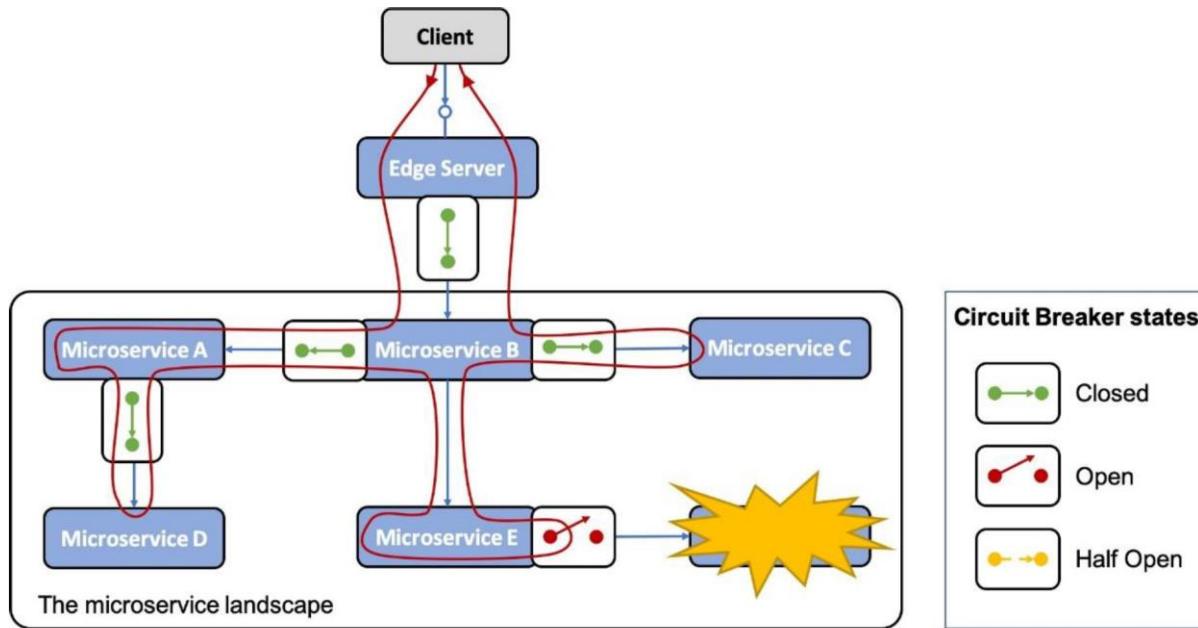
Altri design pattern per micro-servizi

<https://microservices.io/patterns/microservices.html>

- Esempio: pattern *circuit breaker* (*fault tolerance* nei microservizi)

Problem: “How to prevent a network or service failure from cascading to other services?”

Solution: “A service client should invoke a remote service via a proxy that functions in a similar fashion to an electrical circuit breaker”



Risorse e rappresentazioni

- Una *risorsa* (*resource*) è ogni elemento informativo di interesse a cui può essere attribuito:
 - un nome
 - ad es., il “corso di PAC a UniBG”
 - un *identificatore URI* (**Uniform Resource Identifier**)
 - Esempio: l’URI <https://www.unibg.it/ugov/degreecourse/77426>
 - una *rappresentazione* è un gruppo di dati (in formato XML, HTML, JSON, TEXT) che è lo stato (o valore) attuale della risorsa
- L’uso delle rappresentazioni consente di disaccoppiare **il modo in cui i servizi** memorizzano internamente le risorse dal modo con cui le **condividono esternamente**

JSON (*JavaScript Object Notation*)

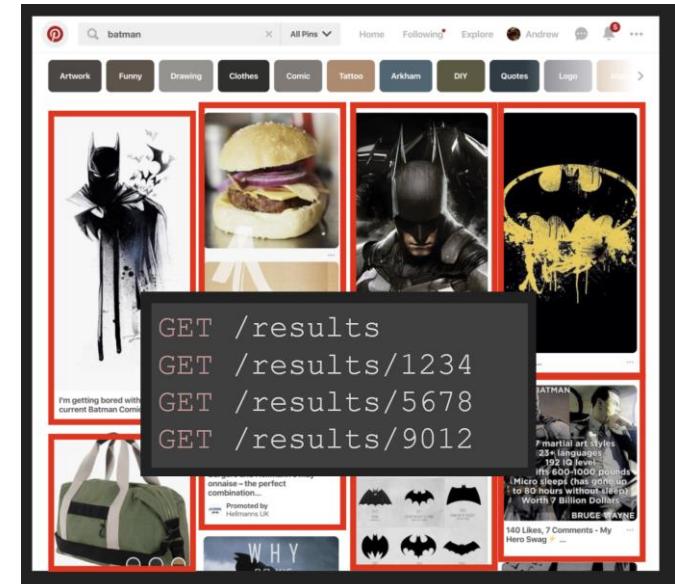
- Formato di interscambio di dati, testuale e leggero - facile da comprendere per l'uomo e per il software da interpretare e da generare (in una varietà di linguaggi di programmazione)
- Due costrutti principali
 - un *oggetto* (un record) è un gruppo di coppie *nome / valore*
 - un *array* (una lista) è una sequenza ordinata di valori
- I nomi sono delle stringhe; i valori possono essere stringhe, numeri, valori booleani, null oppure (ricorsivamente) anche oggetti o array

```
{  
    "employeeId" : 42,  
    "firstName" : "John",  
    "lastName" : "Doe"  
}  
{  
    "employees" : [  
        { "firstName" : "John", "lastName" : "Doe" },  
        { "firstName" : "Anna", "lastName" : "Smith" }  
    ]  
}
```

REST/http (*REpresentational State Transfer*)

REST consente di accedere ad una risorsa con interazione client/server:

- (lato client) effettuare l'invocazione remota di un'operazione del server – tramite una richiesta HTTP all'URI associato ad una risorsa
- (lato server) associare un'operazione del server (il servizio!) con un'operazione HTTP relativa a un certo URI di una risorsa
 - per scambiare **rappresentazioni di dati** (parametri I/O delle API) su HTTP nei formati opportuni (ad es. JSON)
 - una API call **per collezioni omogenee**
 - una API call per i dettagli di ciascun elemento della collezione



Servizi REST

- In genere, un *servizio REST* consente di gestire **collezioni omogenee di risorse**
 - Ad es. un insieme di corsi e un insieme di docenti
- Il servizio definisce, per ciascuna collezione, un URI di base – chiamato ***collection URI***
 - ad es., <https://www.unibg.it/ugov/degreecourse/>
- Ciascuna istanza di risorsa ha poi un proprio URI – ***element URI***
 - ad es., <https://www.unibg.it/ugov/degreecourse/77426>
- Le operazioni offerte dal servizio sono messe in corrispondenza con le **operazioni HTTP** – come GET, PUT, POST e DELETE
 - La semantica di HTTP/REST è *at-most once* (maybe)
- Un'operazione può terminare con **un'eccezione che restituisce lo stato delle risposte HTTP**
 - ad es., *HTTP 404 Not Found* oppure *HTTP 500 Internal Server Error*

REST: verbi http

- L'astrazione principale in REST è la **risorsa**
 - Evitare quindi azioni (verbi); usare **nomi** per definire le API
- REST utilizza **verbi di HTTP** per codificare le **operazioni** sulle risorse
- **POST: Crea** una risorsa (**CREATE**)
- **GET: Leggi** una risorsa (**READ**)
- **PUT: Aggiorna** una risorsa (**UPDATE**)
- **DELETE: Elimina** una risorsa (**DELETE**)

Operazioni REST

- **Operazioni riferite a un collection URI**
 - GET – restituisce tutti gli elementi della collezione
 - PUT – sostituisce la collezione con un’altra collezione
 - POST – crea un nuovo elemento della collezione, e gli assegna un nuovo URI
 - DELETE – cancella l’intera collezione
- **Operazioni riferite a un elemento URI**
 - GET – restituisce uno specifico elemento della collezione
 - PUT – crea un nuovo elemento della collezione, oppure lo aggiorna
 - POST – considera l’elemento della collezione come un’altra collezione, e ne aggiunge un elemento
 - DELETE – cancella l’elemento della collezione

Java frameworks for REST (micro-)service-based API

- Restlets <https://restlet.talend.com/>
- Spring <https://spring.io/guides/gs/rest-service/>
- Swagger <https://swagger.io/tools/open-source/>
 - Approccio dichiarativo e generativo per sviluppare REST API conformi allo standard ***OpenAPI Specification (OAS)***

The screenshot shows the Swagger UI interface for a Petstore API. At the top, there is a dropdown menu for 'Schemes' set to 'HTTP' and a 'Authorize' button with a lock icon. Below this, the 'pet' resource is expanded, showing the following operations:

- POST /pet**: Add a new pet to the store. (Green background)
- PUT /pet**: Update an existing pet. (Orange background)
- GET /pet/findByStatus**: Finds Pets by status. (Blue background)
- GET /pet/findByTags**: Finds Pets by tags. (Light gray background)
- GET /pet/{petId}**: Find pet by ID. (Blue background)
- POST /pet/{petId}**: Updates a pet in the store with form data. (Green background)
- DELETE /pet/{petId}**: Deletes a pet. (Red background)
- POST /pet/{petId}/uploadImage**: uploads an image. (Green background)

Below the 'pet' section, the 'store' resource is partially visible, showing 'Access to Petstore orders'. The interface uses color coding to distinguish between different operations and resources.

Passi di sviluppo di un (micro-)servizio REST

L'uso di REST per implementare un'API richiede:

1. **Definizione dell'interfaccia del servizio** da esporre remotamente – in genere in modo informale o mediante un IDL
2. **Implementazione del servizio**: una componente lato server (un *controller* REST), scritto con riferimento a un client HTTP generico
3. **Codificare i dati scambiati** tra server e client
 - Definire un opportuno *Presentation Model* – un insieme di classi “rappresentazione”
 - I dati vengono scambiati su HTTP mediante JSON oppure XML
 - Spring e altri framework supportano in genere una conversione automatica

Definizione dell'interfaccia del servizio

- In REST, non è necessario definire formalmente con un IDL l'interfaccia del servizio
 - OpenAPI Specification (OAS) non è ancora molto diffuso
- L'interfaccia del servizio va comunque stabilita.
- Si consideri un semplice servizio per generare dei saluti

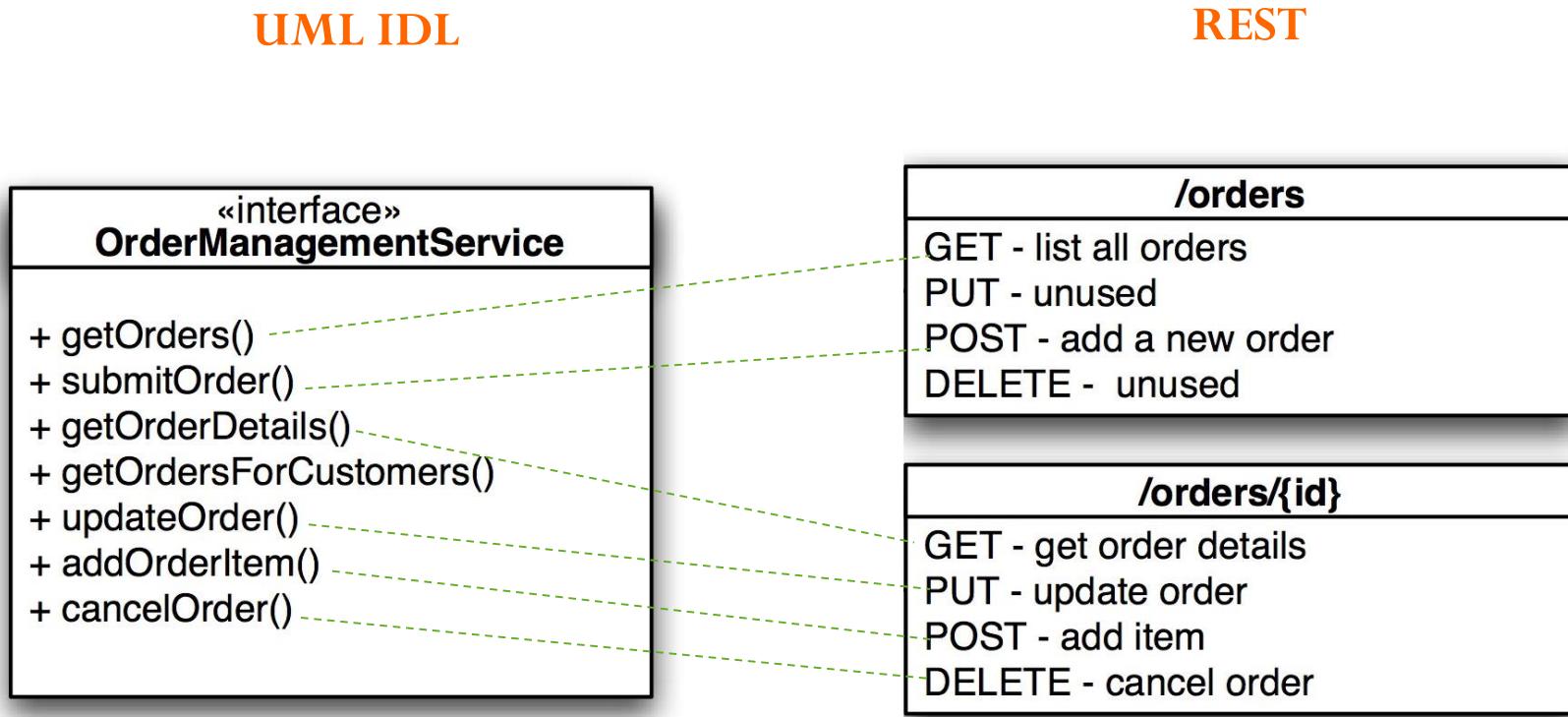


Definizione dell'interfaccia del servizio



- Per ogni operazione del servizio bisogna definire almeno
 - l'operazione HTTP e il percorso (l'*URL name*)
 - i parametri, con i loro tipi – nel percorso o nel corpo
 - i risultati, con i loro tipi
- Operazione *sayHello*
 - **GET /hello/{name}** .../hello?name=Pippo
 - il parametro *name* è una stringa, nel percorso
 - il risultato è una stringa

Definizione dell'interfaccia del servizio: ancora un esempio



<http://example.com/orders>

<http://example.com/orders/776654>

Il framework Spring

<https://spring.io/projects>

- Framework leader Java usato fin dal 2002 per semplificare la creazione di applicazioni web (altrimenti basata su Java EE)
- *Annotation driven* e supporta il paradigma *Inversion of Control* (IoC – il nostro codice viene richiamato dalla libreria) tramite *Dependency Injection* (DI)
 - tutte le dipendenze di un oggetto (ovvero altri oggetti di cui l'oggetto ha bisogno) all'interno della nostra applicazione vengono *iniettate* dall'esterno, tramite costruttore o metodi setter
- Incorpora molti progetti:
 - Spring boot: semplifica la creazione e configurazione dei progetti con vari tool (es. Spring Initializr; standalone JAR che incorporano un web server Tomcat nel JAR)
 - Spring MVC: per realizzare applicazioni web basate sul modello MVC
 - Spring Cloud: supporto ai pattern distribuiti (service discovery, circuit breaker, ecc.)
 - Spring data: per la persistenza
 - Spring security
 - ...

Creazione di un progetto Spring in un IDE

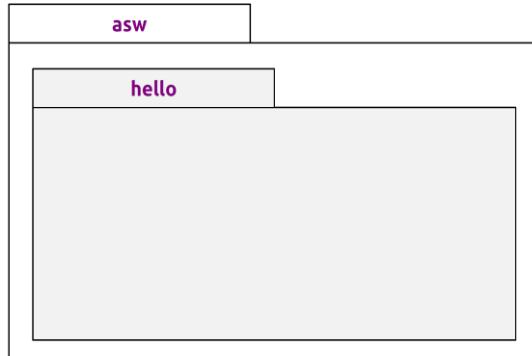
- Approccio dichiarativo: uso **Spring Initializr** + import del progetto nell'IDE preferito
 - <https://start.spring.io/>
- Approccio operativo: download e import di Spring nell'IDE preferito e creazione di un progetto
 - [Vedi lezione di tutorato con IDE Eclipse]

Implementare un'applicazione a micro-servizi in Spring (lato server)

- Bisogna creare la struttura di package e una classe con annotazione `@SpringBootApplication` che fa da start point

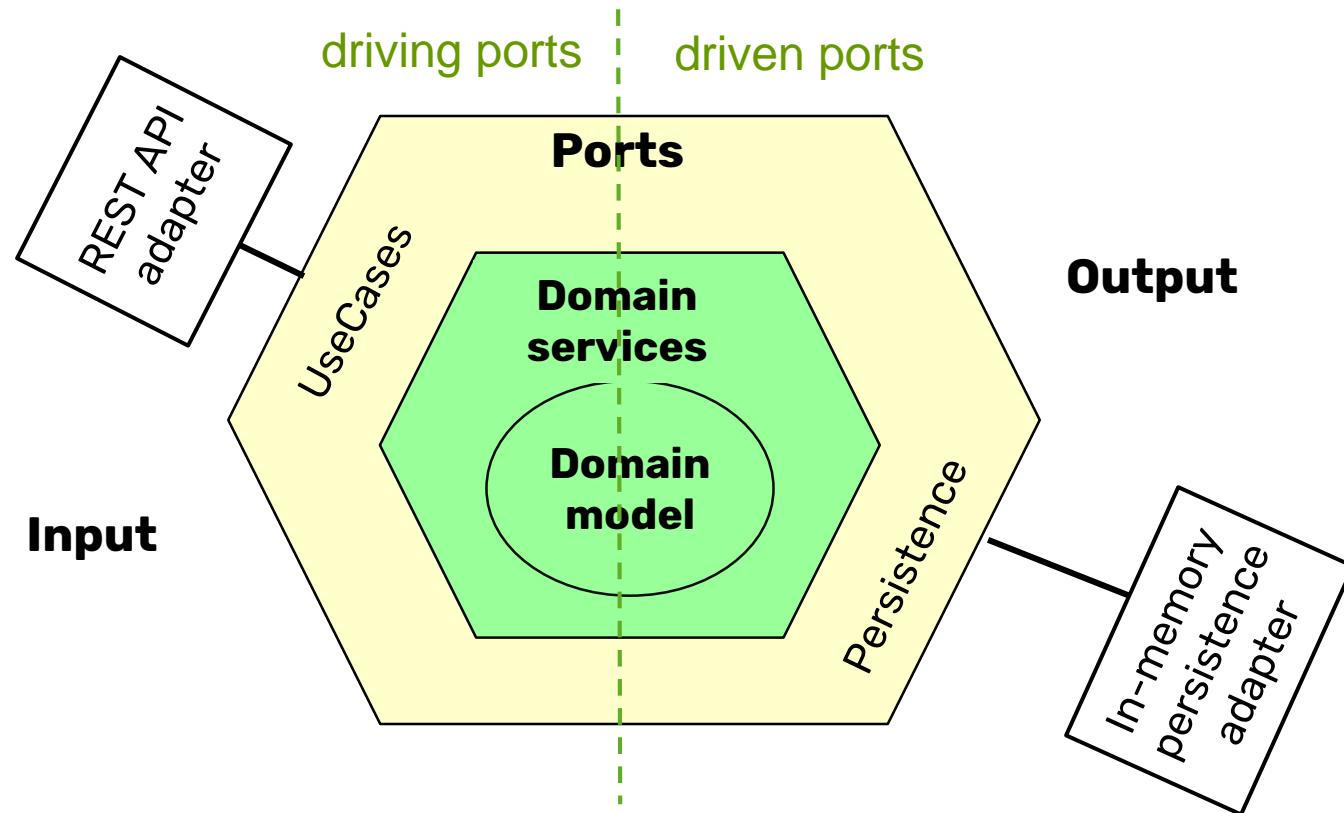
```
package asw.hello;  
import org.springframework.boot.SpringApplication;  
import org.springframework.boot.autoconfigure.SpringBootApplication;
```

```
@SpringBootApplication  
public class HelloServiceApplication {  
    public static void main(String[] args) {  
        SpringApplication.run(HelloServiceApplication.class, args);}}
```



Implementazione di un (micro-) servizio in Spring

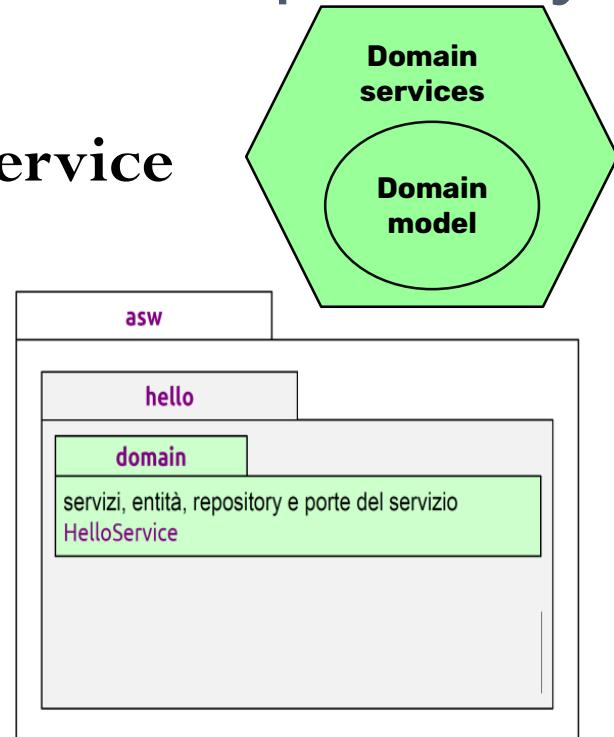
- Lato server, bisogna definire un **server REST** per il servizio
- Modello architetturale *esagonale*:



Implementazione di un (micro-) servizio in Spring: porte di servizio, entità e repository

- In Spring i servizi sono classi annotate **@Service**

```
package asw.rest.hello.domain;  
import org.springframework.stereotype.Service;  
@Service  
public class HelloService {  
    public String sayHello(String name) {  
        return "Hello, " + name + "!";  
    }  
}
```



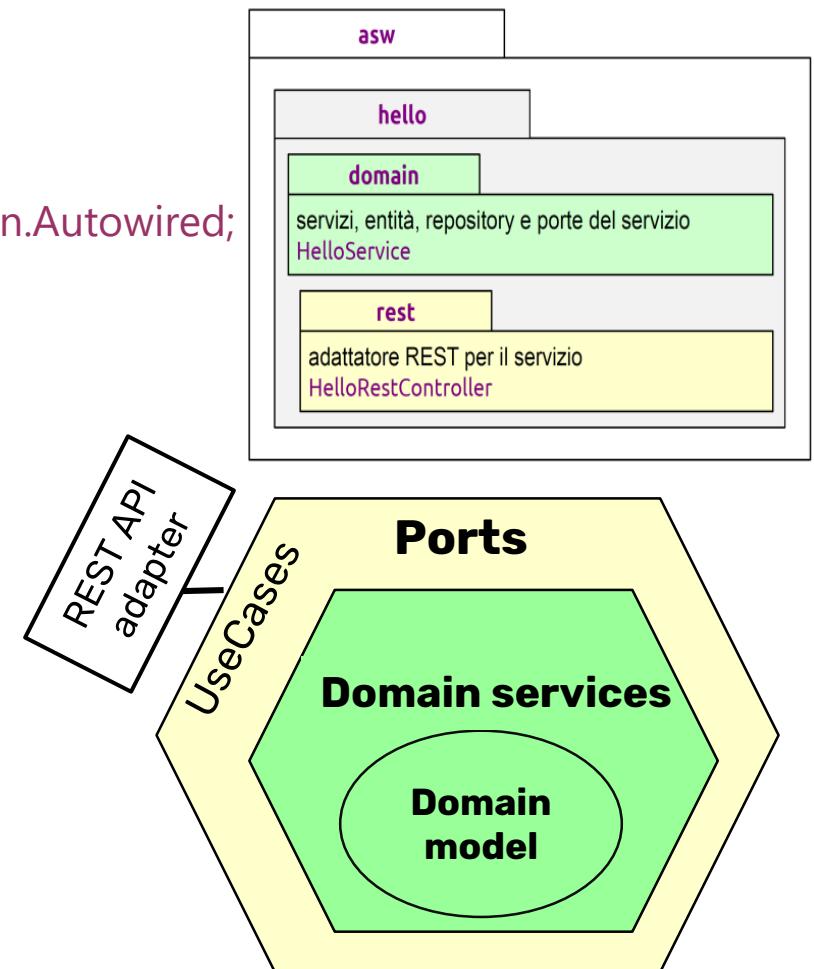
- Classi annotate con **@Entity** e **@Repository** (non presenti nell'esempio) indicano il *presentation model/domain model*:
 - oggetti per rappresentare i dati da scambiare (in JSON o XML) e memorizzare in modo persistente con *Spring data*

Implementazione di un (micro-) servizio in Spring: controller

Definiamo nel package **asw.rest.hello.rest** una classe **HelloRestController** che è il REST controller (o adaptor) per il servizio *HelloService*

```
package asw.rest.hello.rest;
import asw.rest.hello.domain.HelloService;
import org.springframework.web.bind.annotation.*;
import org.springframework.beans.factory.annotation.Autowired;
@RestController
public class HelloRestController {
@.Autowired
private HelloService helloService;

//Operazione acceduta come GET /hello/{name}
@GetMapping("/hello/{name}")
public String sayHello(@PathVariable String name) {
    return helloService.sayHello(name);
}
```



Consumo del servizio

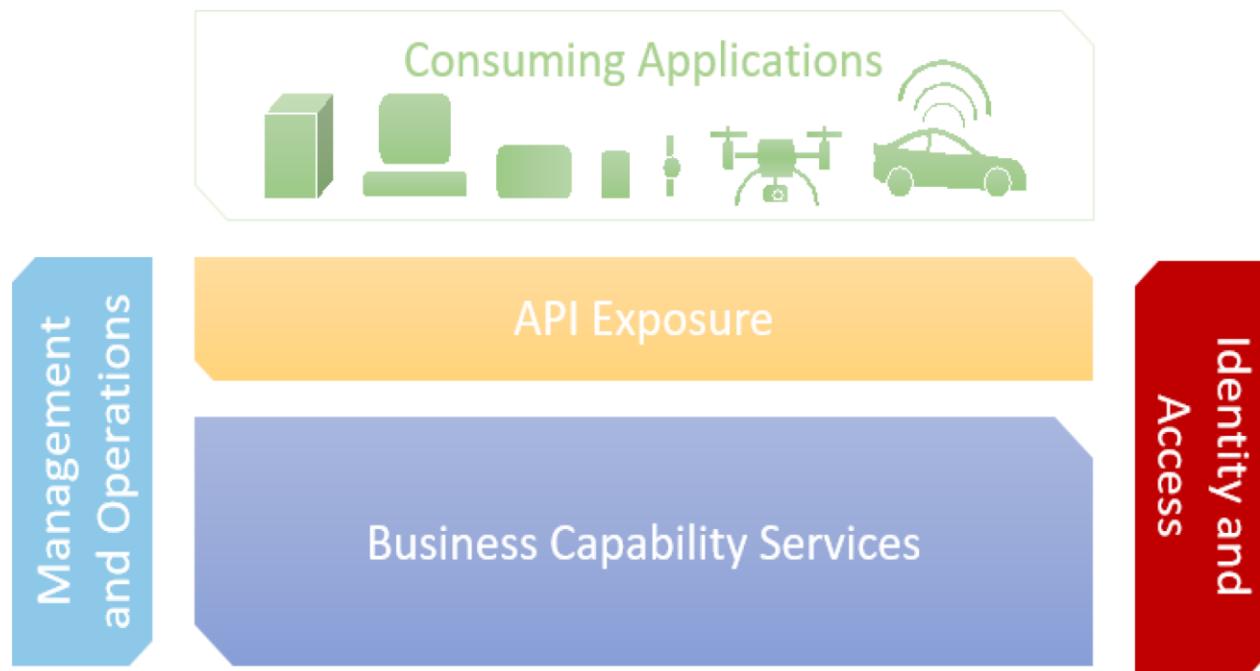
Client REST

- Come client per questo servizio è possibile usare **un qualunque client HTTP**
 - Ad esempio: un browser web, **curl** (Client URL) a linea di comando oppure il tool **Postman**
- *Swagger UI* (<https://swagger.io/tools/swagger-ui/>) consente di visualizzare e interagire con un servizio REST mediante un'interfaccia web generata automaticamente a partire dal servizio REST
- E' possibile realizzare una **componente lato client** mediante l'utilizzo di una delle tante **librerie per la realizzazione di client HTTP**
 - Ad es., in Spring: **RestTemplate** (ma in via di deprecazione) o **WebClient**

API-led software architectures

Conceptual model of API-led architectures

- Horizontal blocks represent core runtime capabilities
 - Services expose their functionality (*Business Capability*) via API endpoints that are not accessed directly but mediated via the *API exposure layer* (*gateways!*)
- Vertical blocks represent important supporting capabilities geared towards life cycle support, management, operations, and analytics

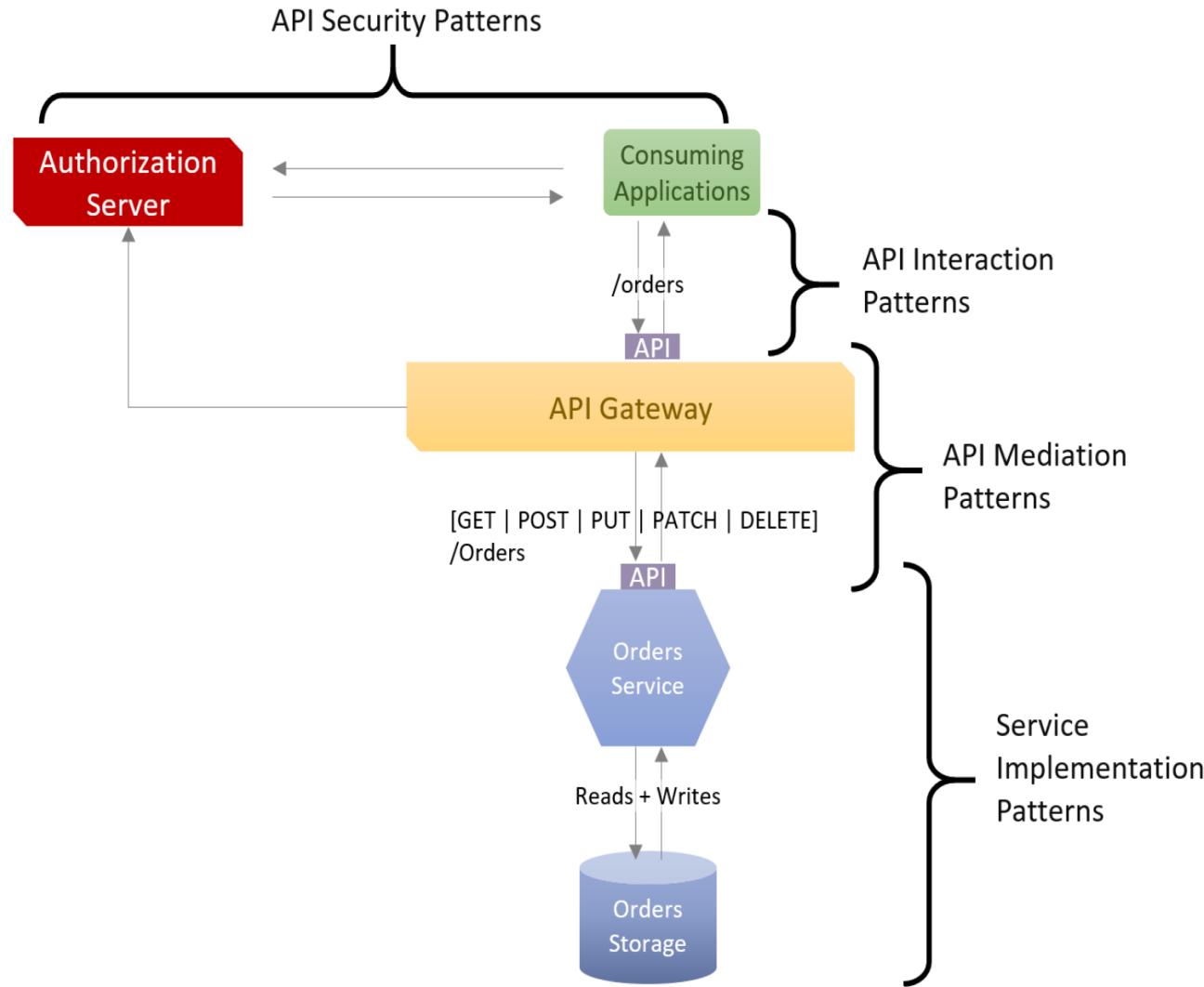


API-led architecture patterns

- Address common requirements/challenges that arise when adopting API-led architectures
- Enable API-led connectivity
- Main pattern types:
 - Mediation: e.g., *API resource routing*
 - Implementation: e.g. *API aggregator*
 - Interaction: e.g., *API webhook*
 - Security: e.g., *API basic authentication*

API-led pattern categories

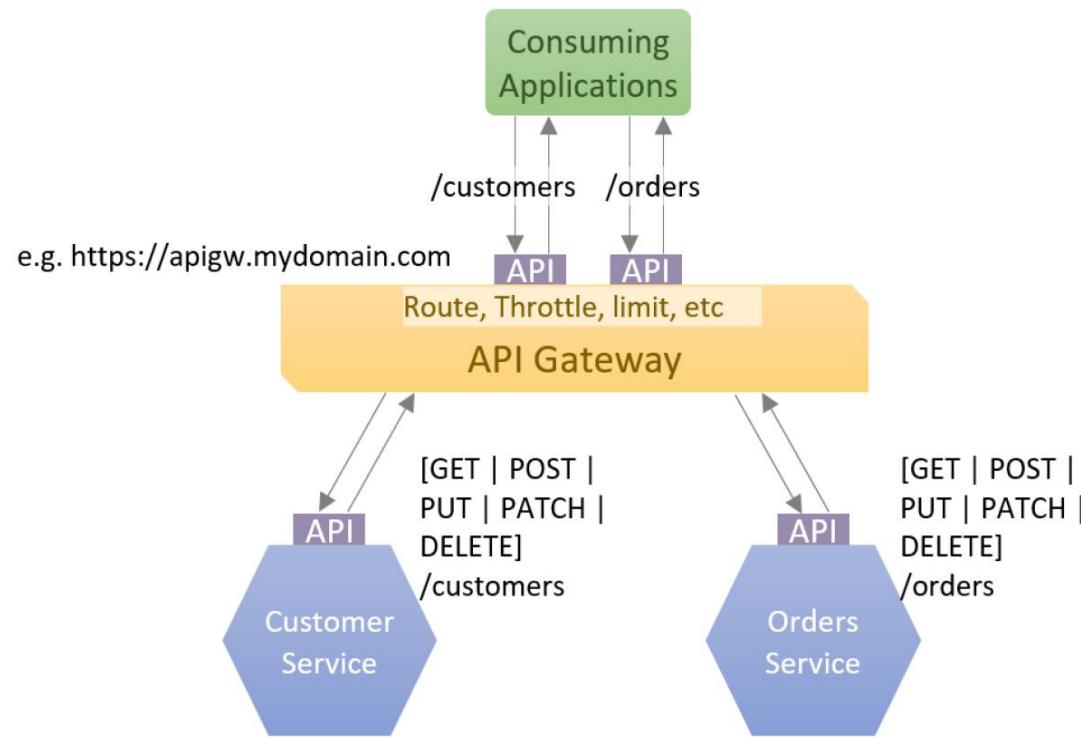
- Illustration done using REST



API Resource routing

Problem statement: As the number of services increases, so does the number of HTTP endpoints exposed to access them.

Solution: Uses an API gateway to route calls and enforce policies based on unique resource identifiers (URIs).

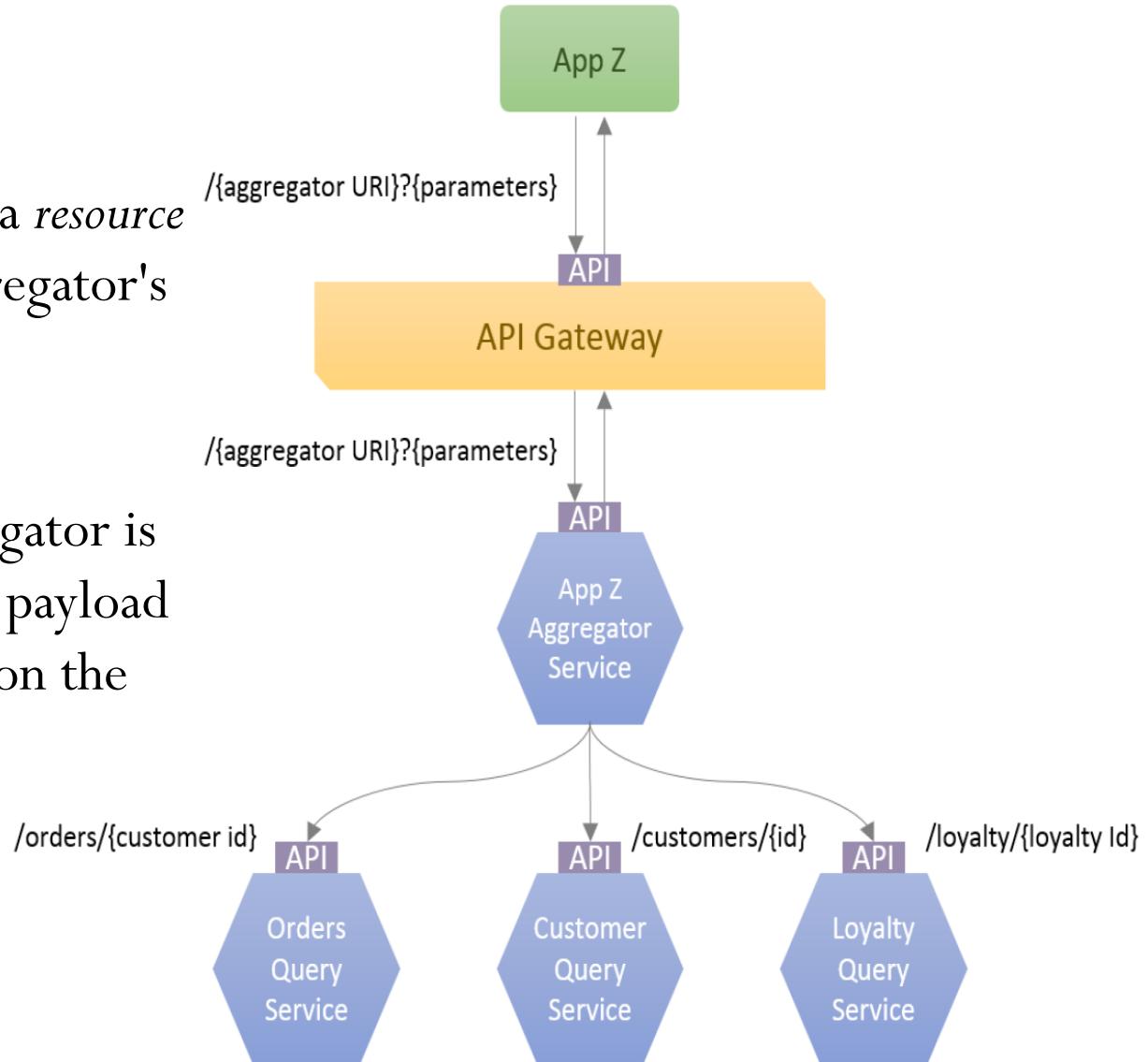


API aggregator pattern

- *Problem statement:* A consuming application (for example, a mobile or JavaScript browser app) has to collect data from multiple calls to different APIs, and this can result in inefficiencies
 - increased complexity in the client-side code
 - over-utilization of network resources
 - poor user experience as the application is more exposed to latency issues
- *Solution:* Instead of having a client application making several calls to multiple APIs, an API aggregator does this on behalf of the consumer on the server-side

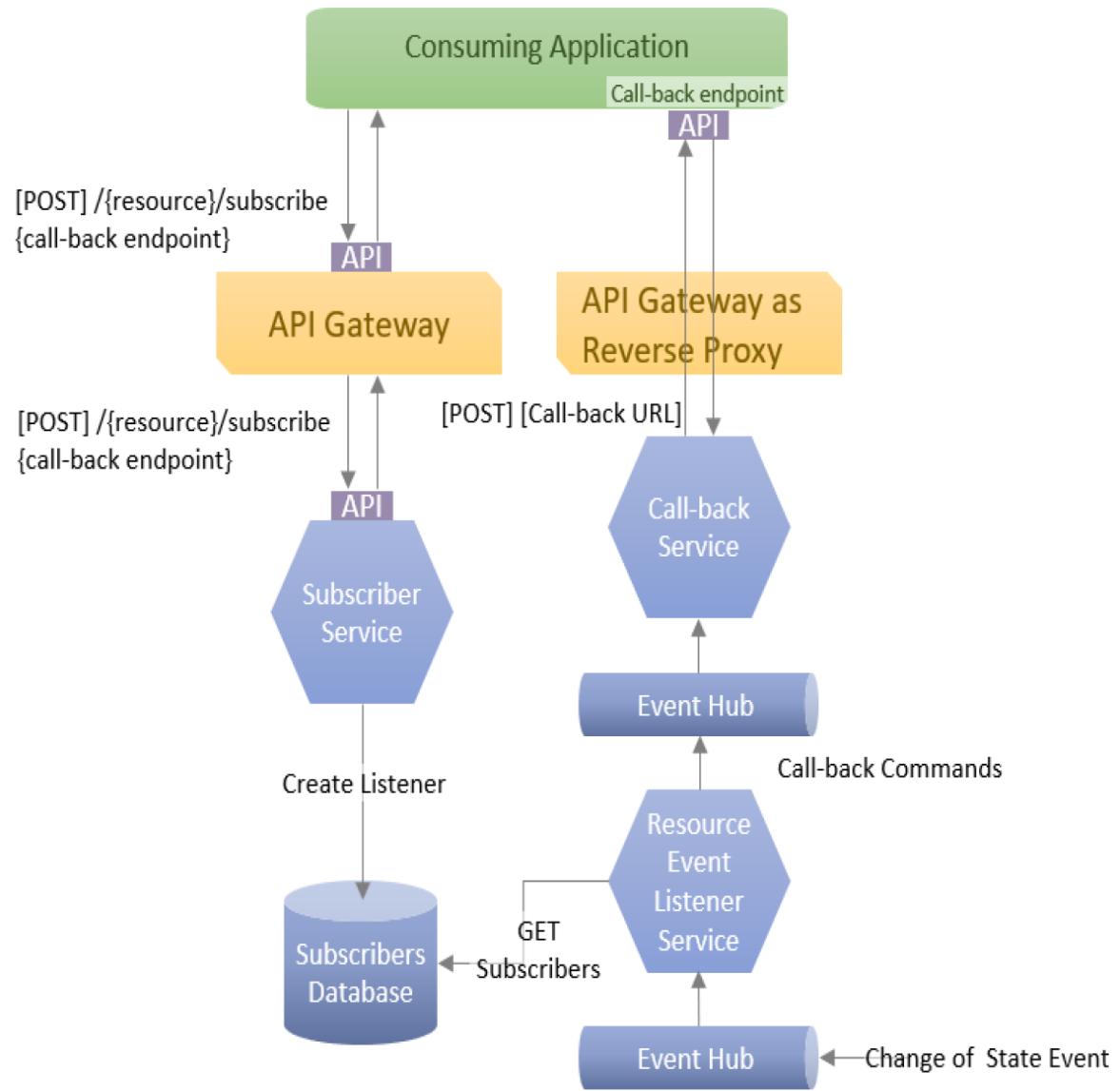
API aggregator pattern

- An API gateway acts as a *resource router* based on the aggregator's unique URI
- The input for the aggregator is either a parameter or a payload in the body depending on the HTTP verb



API Webhook

- *Problem statement:* A consuming application desires to be informed of any change of state on a specific record or records
 - Updates in customers in a CRM SaaS, new posts on a user's blog, etc.
- *Solution:* Asynchronous APIs that allow API consumers to be notified (*called back*) when a change of state takes place in a given record or set of records

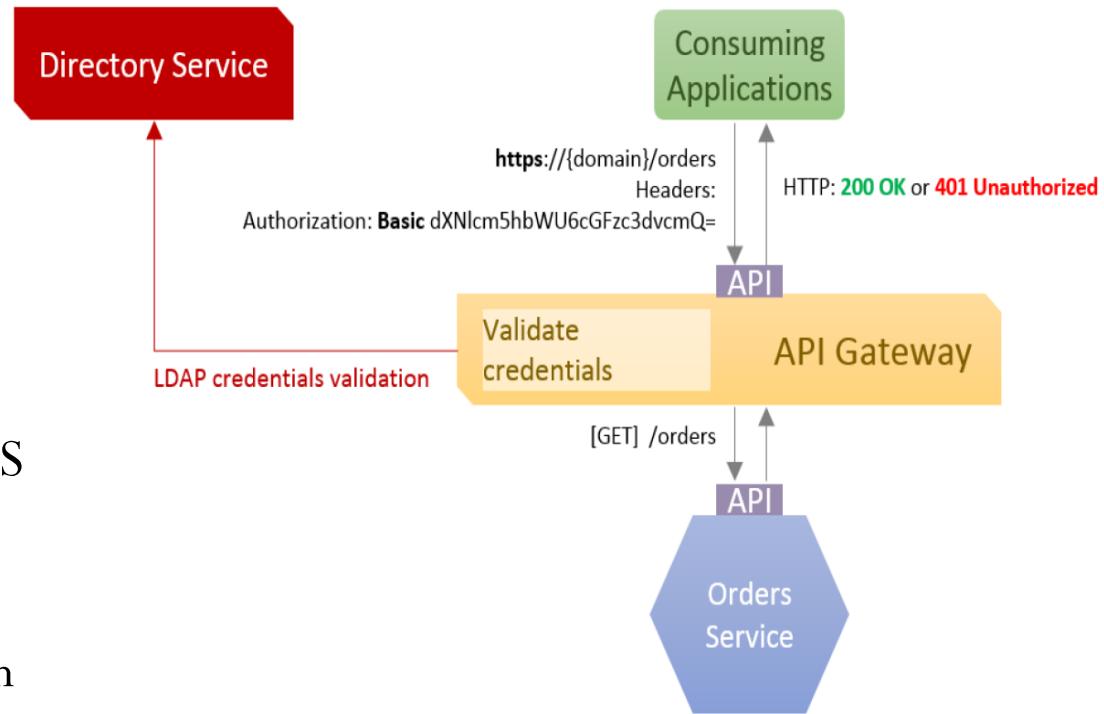


API basic authentication

- *Problem statement:* verify that the application users exist and have valid credentials in the corporate directory (e.g., MS Active Directory and/or any other LDAP server)

- *Solution:* HTTP basic authentication at the API gateway level, assuming HTTPS for transport encryption

- a consuming application must include the user's credentials in the HTTP header Basic `<base64(username:password)>`





UNIVERSITÀ
DEGLI STUDI
DI BERGAMO

Dipartimento
di Ingegneria Gestionale,
dell'Informazione e della Produzione



Modern architectural styles and computing paradigms

PROGETTAZIONE, ALGORITMI E
COMPUTABILITÀ
(38090-MOD1)

Corso di laurea
Magistrale in
Ingegneria
Informatica

RELATORE
Prof.ssa Patrizia
Scandurra

SEDE
DIGIP

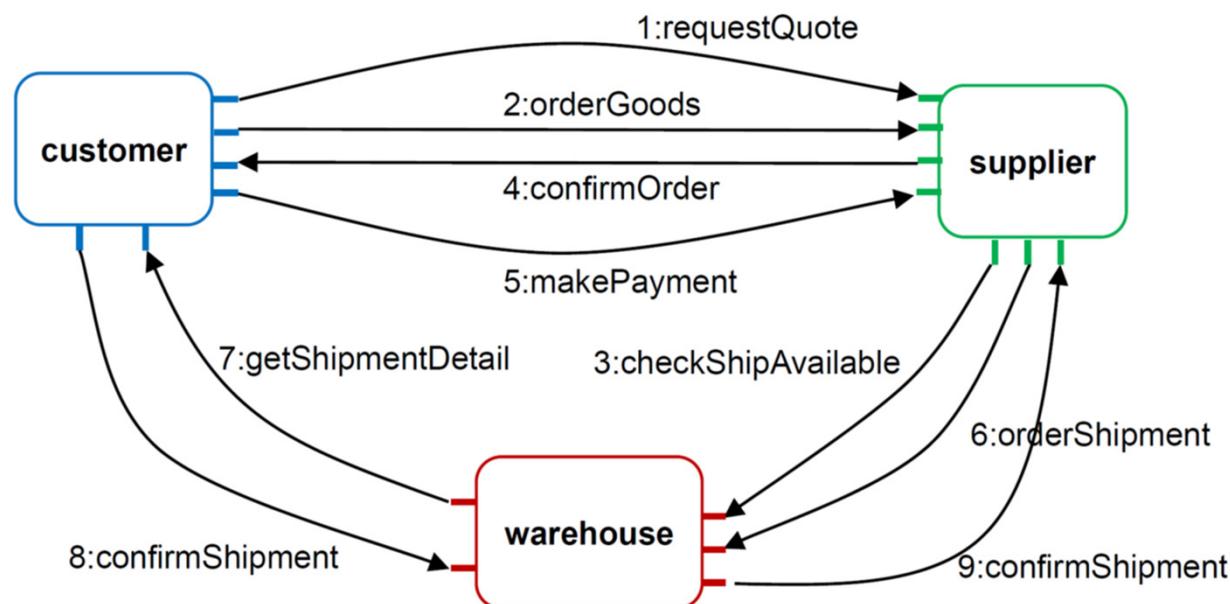
Outline

- Modern software architectural styles and computing paradigms
 - service systems (*service computing*)
 - *Service-oriented architecture (SOA)*
 - *Service-based architecture*
 - *Microservice architecture*
 - cloud-based systems (*cloud computing, IoT-Edge-Cloud computing*)
 - What's next?
 - *Microservices and container-as-a-service*
 - *Functions-as-a-service (serverless computing)*
 - Self-adaptive and AI-based systems or autonomous systems (*autonomic computing*)

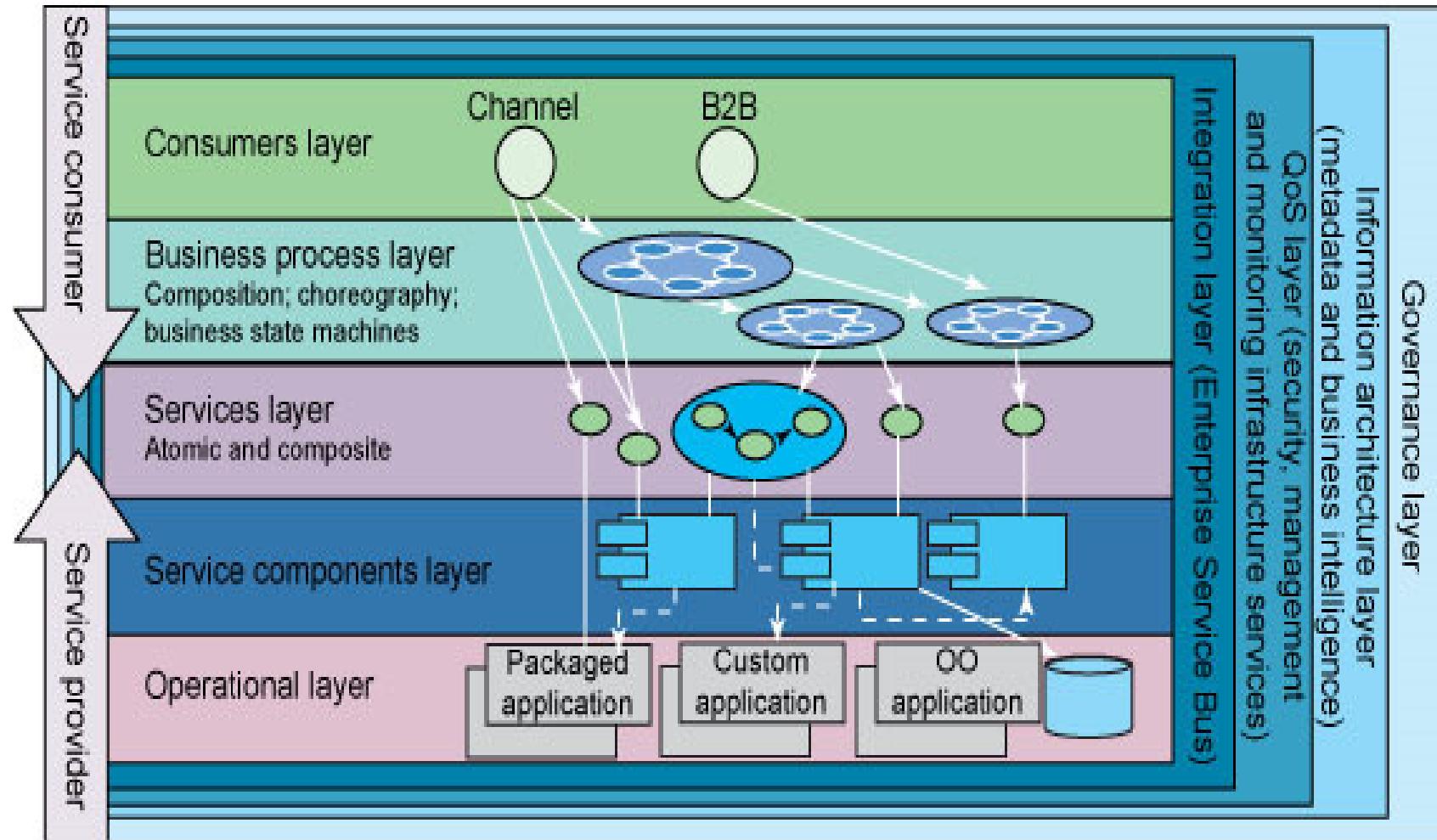
Service systems & Service Computing

Service systems

- A service is a software component that is intended to implement a feature or service of business of an organization
- It can be discovered and invoked by its consumers through an open interface and using standard (web) technologies (*API end points*)
- Services are composed based on the cooordination and exchange of messages (*service request*)

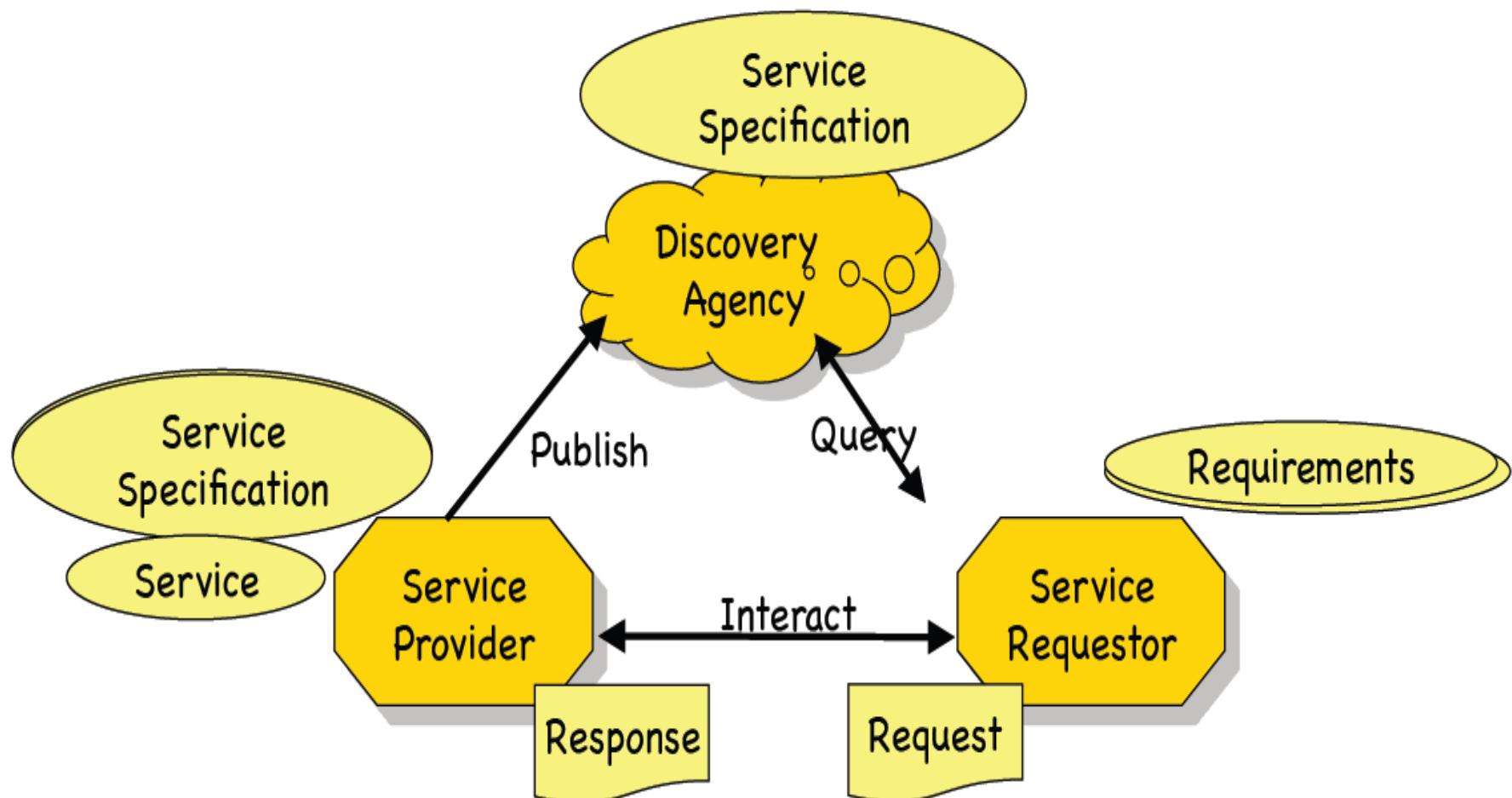


Reference architecture: Service-Oriented Architecture (SOA)



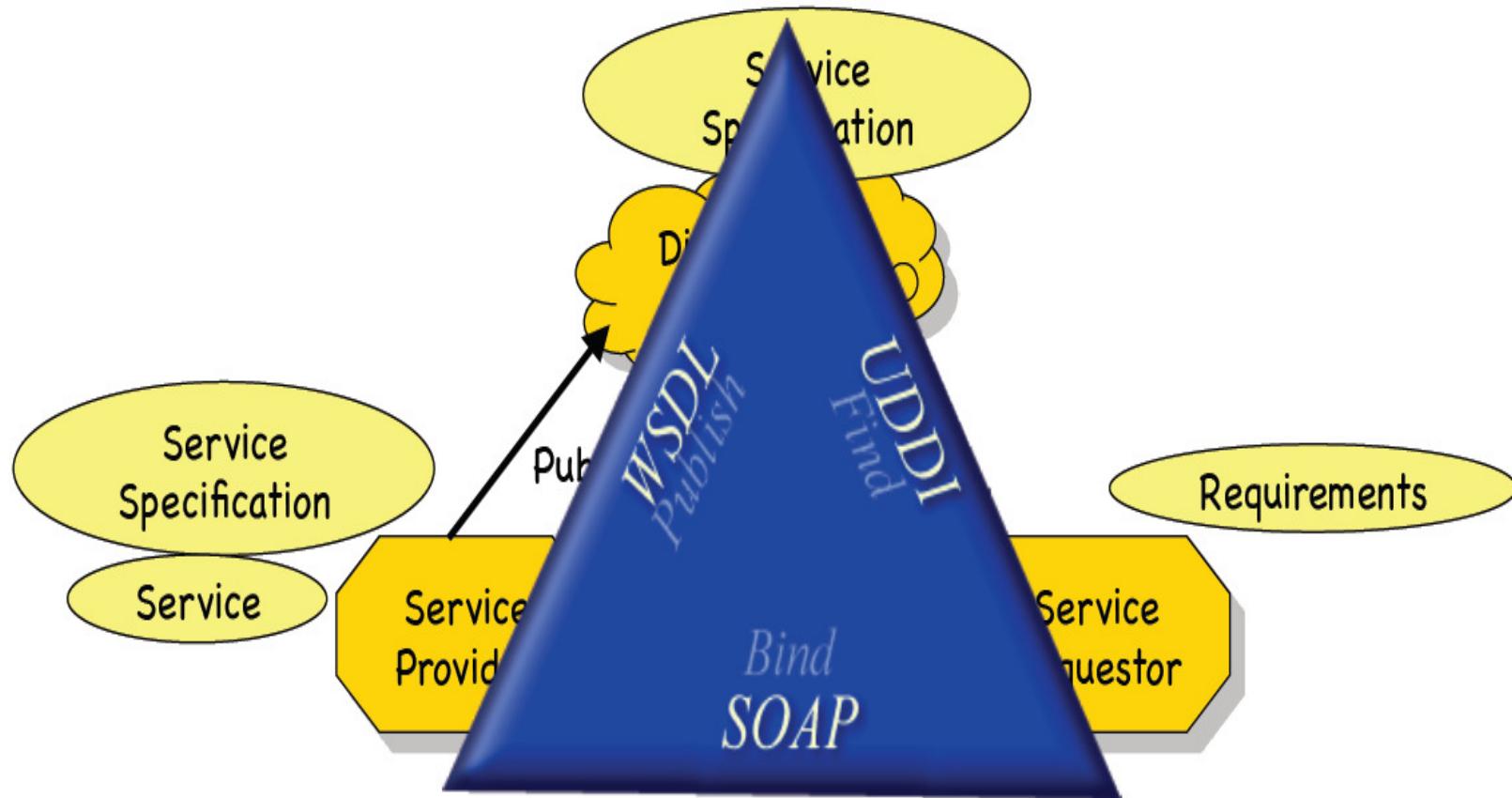
<https://www.ibm.com/developerworks/library/ar-archtemp/>

The SOA triangle interaction model



The SOA triangle interaction model

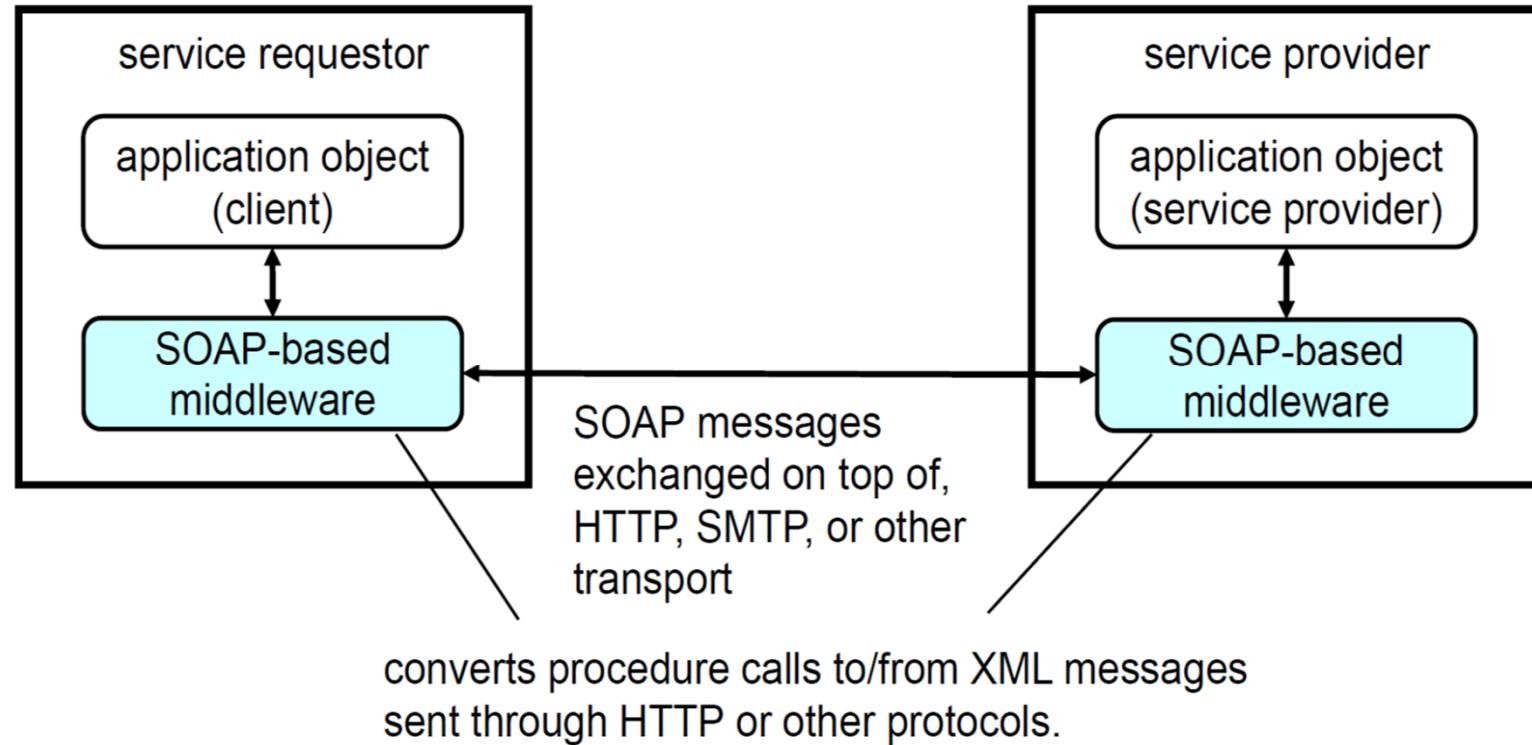
- As realized through the XML-based standards defined by W3C



The SOA standards by W3C

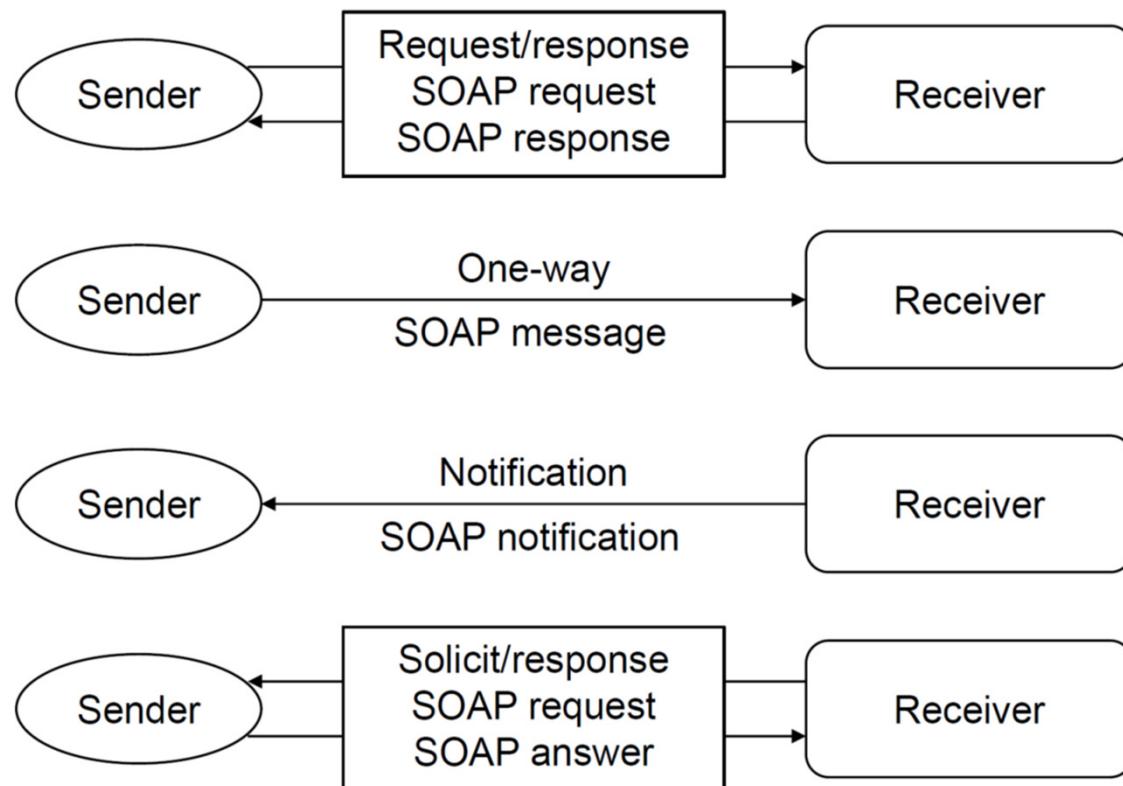
- **WSDL (Web Service Description Language)**
 - WSDL is a language for describing web services and how to access them
- **SOAP (Simple Object Access Protocol)**
 - is a protocol specification for exchanging structured information in the implementation of web services in computer networks
 - relies on other application layer protocols, most notably Hypertext Transfer Protocol (HTTP) or Simple Mail Transfer Protocol (SMTP), for message negotiation and transmission
- **UDDI (Universal Description Discovery and Integration)**
 - a directory service where businesses can register and search for Web services

SOAP interactions



SOAP interaction patterns

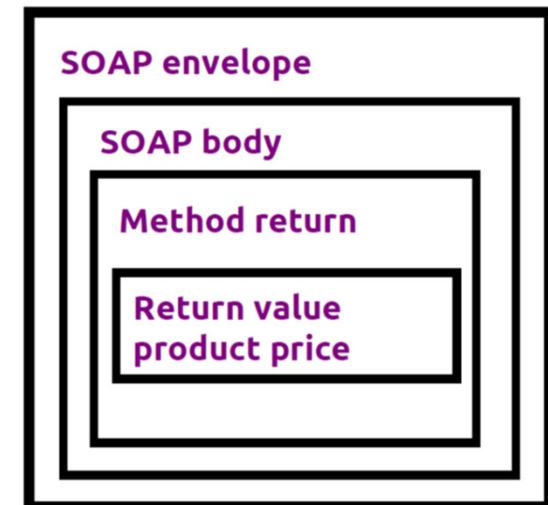
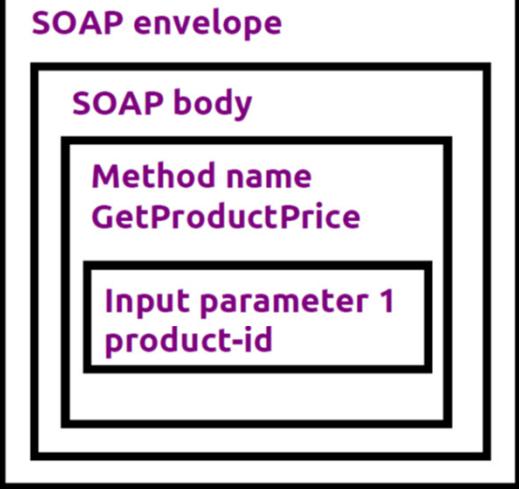
- In WSDL, each operation can have only one input message and/or a single output message
- Four possible interaction patterns:



SOAP messages

```
<env:Envelope  
    xmlns:SOAP="http://www.w3.org/2003/05/soap-envelope"  
    xmlns:m="http://www.plastics_supply.com/product-prices">  
    <env:Header>  
        <tx:Transaction-id  
            xmlns:t="http://www.transaction.com/transactions"  
            env:mustUnderstand='1'>  
            512  
        </tx:Transaction-id>  
    </env:Header>  
    <env:Body>  
        <m:GetProductPrice>  
            <product-id> 450R60P </product-id>  
        </m:GetProductPrice >  
    </env:Body>  
</env:Envelope>
```

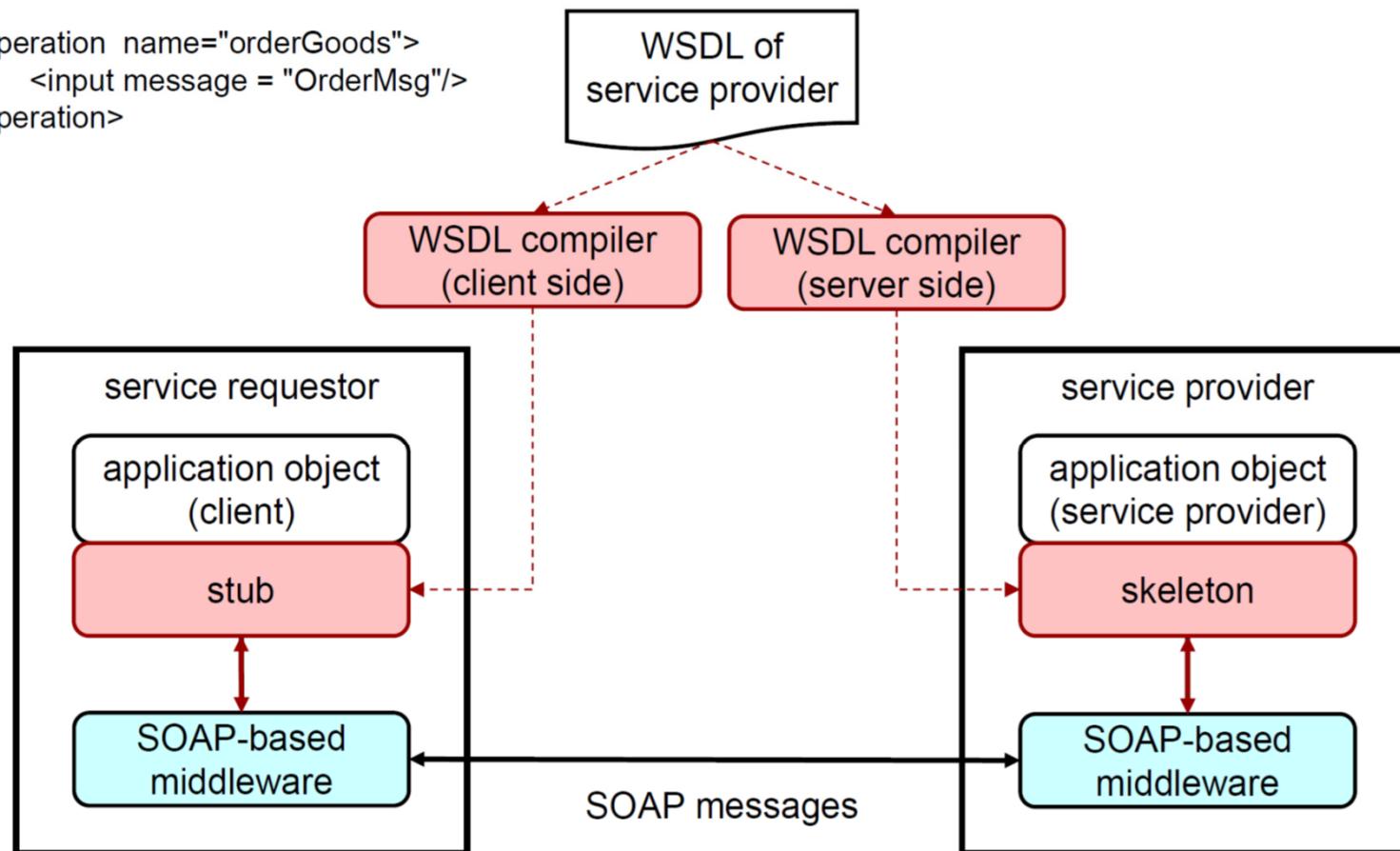
```
<env:Envelope  
    xmlns:SOAP="http://www.w3.org/2003/05/soap-envelope"  
    xmlns:m="http://www.plastics_supply.com/product-prices">  
    <env:Header>  
        <!--! - Optional context information -->  
    </env:Header>  
    <env:Body>  
        <m:GetProductPriceResponse>  
            <product-price> 134.32 </product-price>  
        </m:GetProductPriceResponse>  
    </env:Body>  
</env:Envelope>
```



WSDL compilers

- Top-down generation

```
<operation name="orderGoods">  
    <input message = "OrderMsg"/>  
</operation>
```



Example of WSDL document (simplified)

- The <portType> element defines a web service, the operations that can be performed, and the messages that are involved
- The *request-response* type is the most common operation type

```
<message name="getTermRequest">
    <part name="term" type="xs:string"/>
</message>

<message name="getTermResponse">
    <part name="value" type="xs:string"/>
</message>

<portType name="glossaryTerms">
    <operation name="getTerm">
        <input message="getTermRequest"/>
        <output message="getTermResponse"/>
    </operation>
</portType>
```

Web service technologies in Java

- Lots of **SDK and frameworks help Java developers build web SOAP APIs**
 - Several libraries and frameworks as part of *Java EE*
 - JAX-WS (Java API for XML Web Services)
 - <https://docs.oracle.com/javaee/6/tutorial/doc/bnayl.html>
 - Apache Axis <http://axis.apache.org/axis/>
 - etc.
 - The SDKs for WS can use WSDL and SOAP for the automatic construction of **proxy objects** - both service side and client side

Service modeling languages and standards

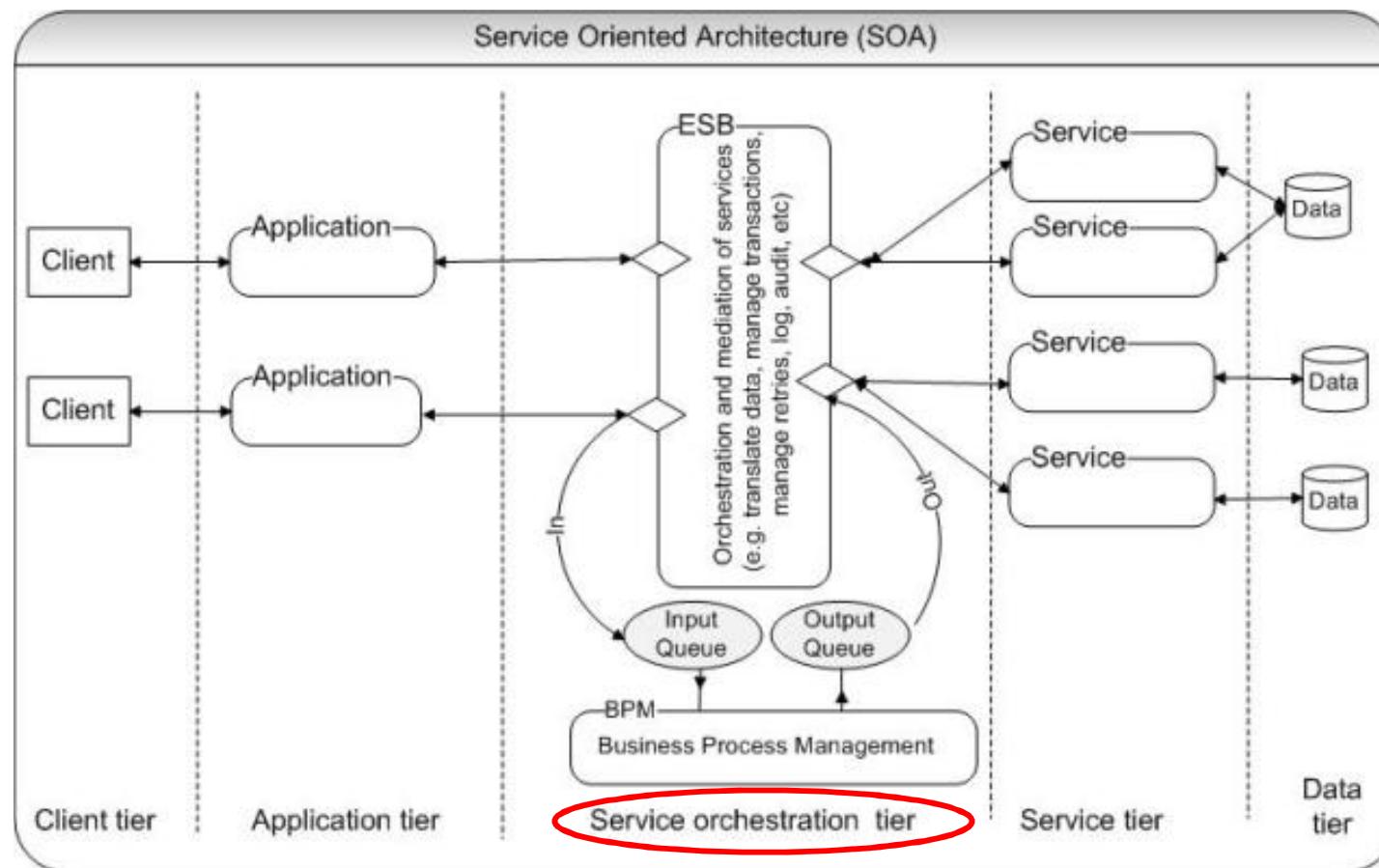
ADLs for service architecture modeling:

- SoaML (UML profile)
 - <http://www.omg.org/spec/SoaML/1.0.1/>
- SOMA-SOMF
 - <http://www.sparxsystems.com/somf>
- OASIS's standard Service Component Architecture (SCA)
<http://oasis-opencsa.org/sca>
- etc..

Example of SA of a web service orchestration

Service *orchestration* interaction model

- coordination and arrangement of multiple services exposed as a single aggregate service



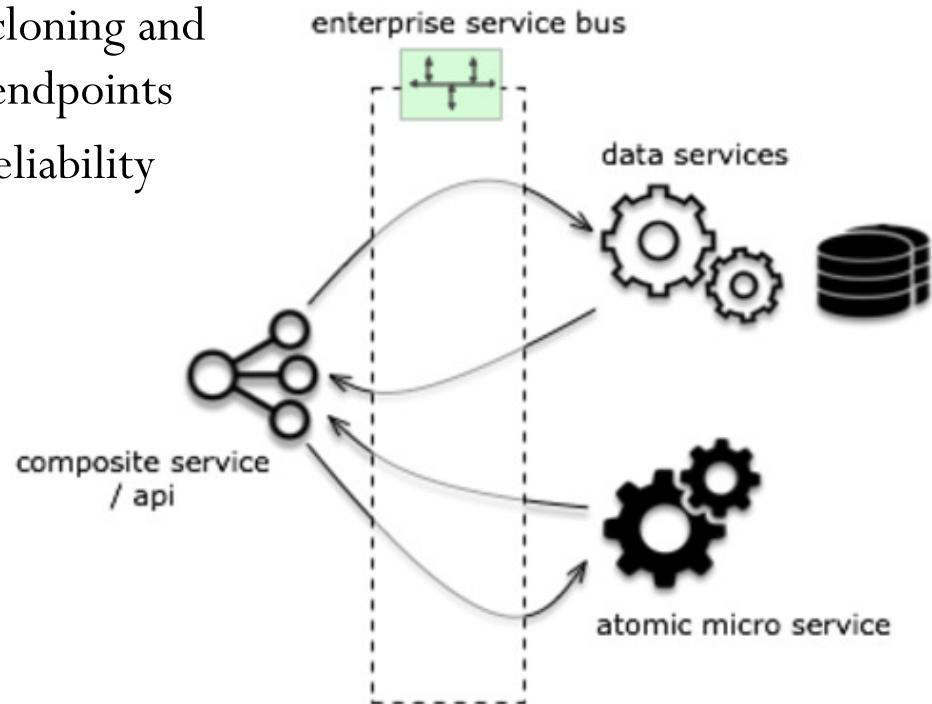
Service orchestration models

There are two distinct type of orchestrations:

- **Short-running stateless orchestrations**
 - Service invocations are more synchronous in nature and deals with transient data/sessions
- **Long-running stateful orchestrations**
 - Service invocations are asynchronous in nature and with persistent sessions

Service orchestration - When To Use An ESB versus a Workflow Engine

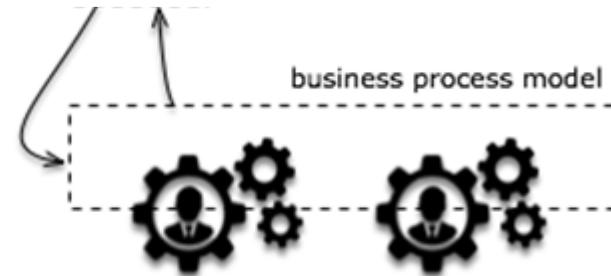
- **Short-running stateless orchestrations**
 - use **request/response interaction patterns (end-to-end)**
- The **enterprise service bus (ESB)** fits this description
 - It deals with multiple heterogeneous systems:
 - message splitting, transformation, cloning and aggregation from multiple service endpoints
 - manage security, transactions and reliability
 - supports the discovery of services



Service orchestration - When To Use An ESB versus a Workflow Engine

- **Long-running stateful orchestrations** may involve human approval or delegation activities (*human-in-the-loop*); resulting in a complex orchestration with human tasks and external links

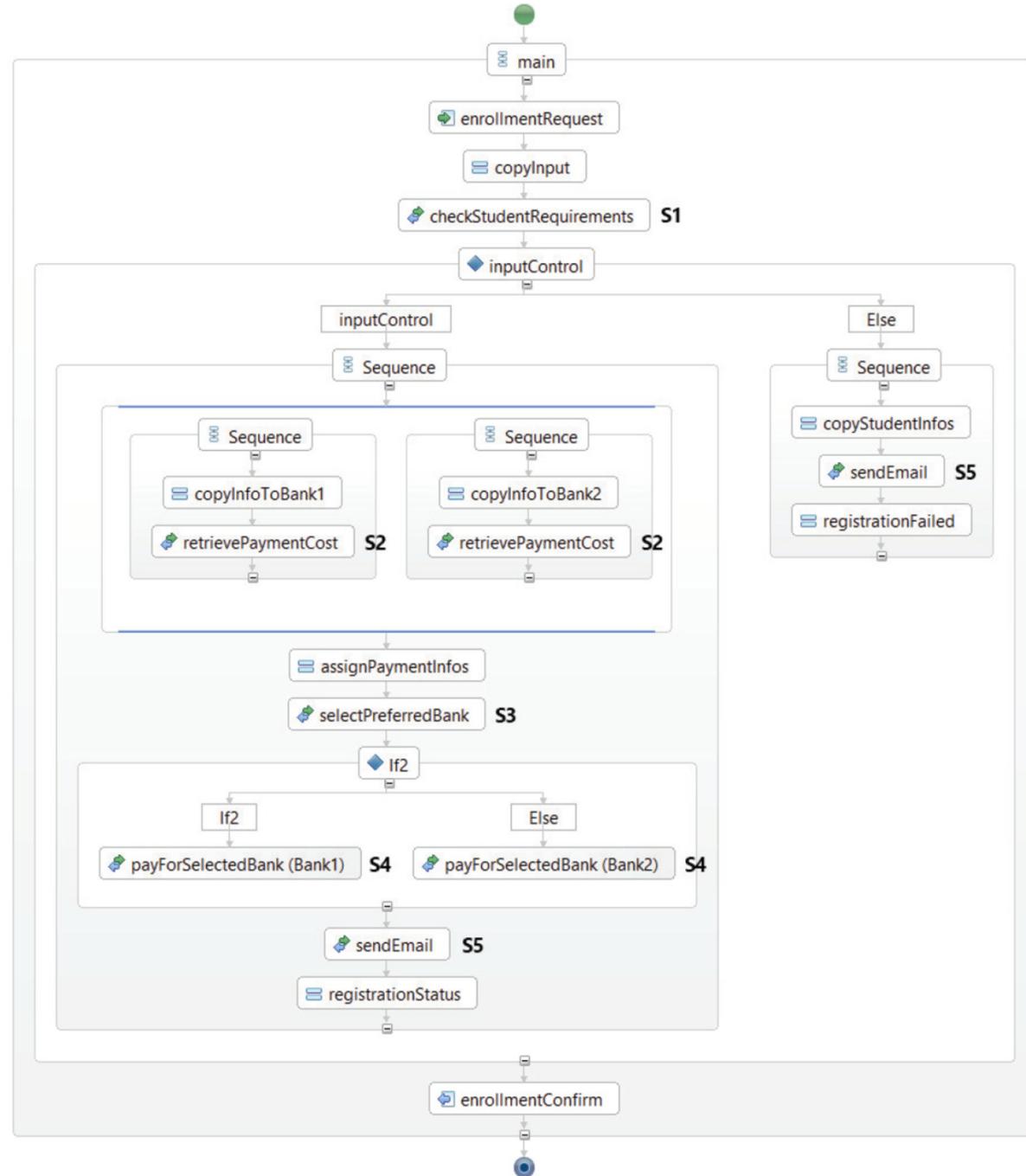
- Example: a *loan approval workflow*



- The traditional architectural belief is to model such a *business process* as a **workflow**
 - **Standards:** **BPEL** (Business Process Execution Language) and **BPMN** (Business Process Modeling Notation)
 - Involve the use of workflow modeling editors and engines (BPEL or BPMN engines)
 - https://en.wikipedia.org/wiki/List_of_BPEL_engines

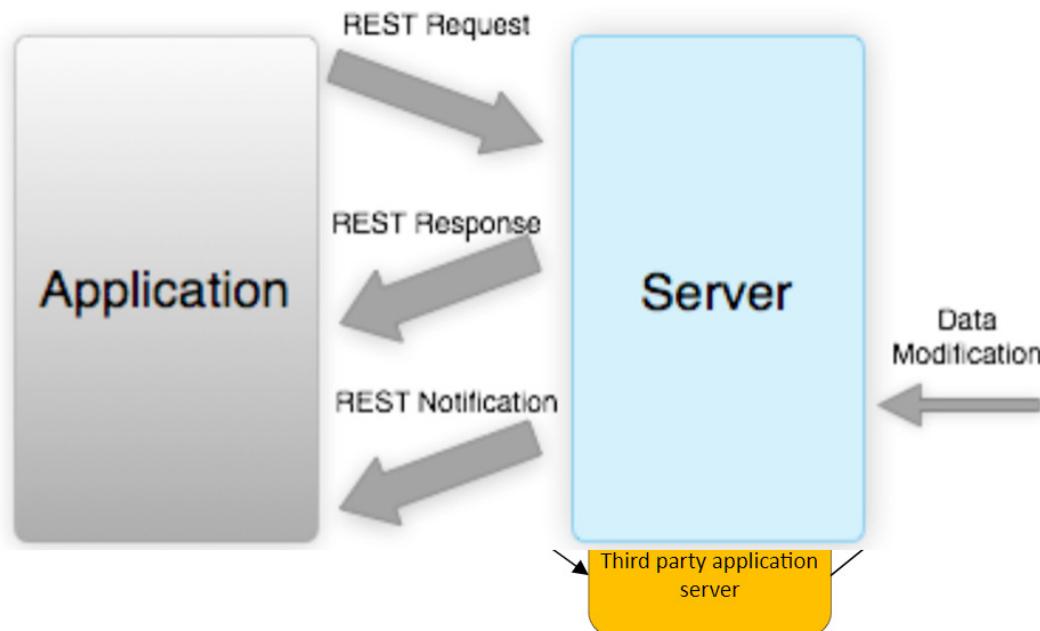
Example of BPEL workflow

- Student enrollment BPEL process



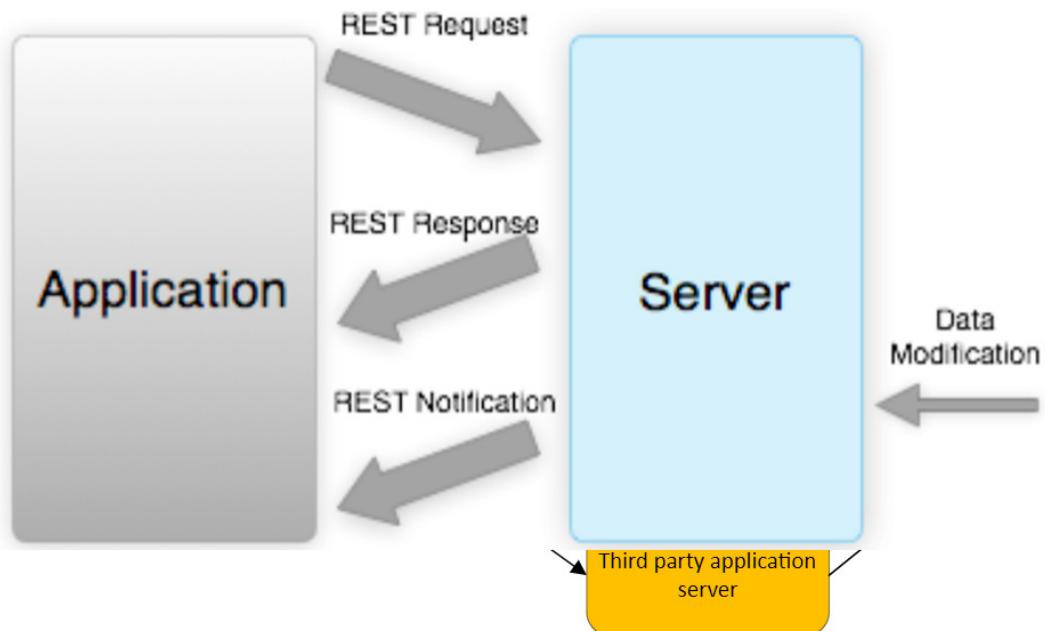
SOA with RESTful services

- In 2000: architectural style **Representational State Transfer (REST)** [*Roy Fielding's PhD thesis*]
- The goal is to simplify the architecture of *stateless* web services
- REST is one of the most popular technologies

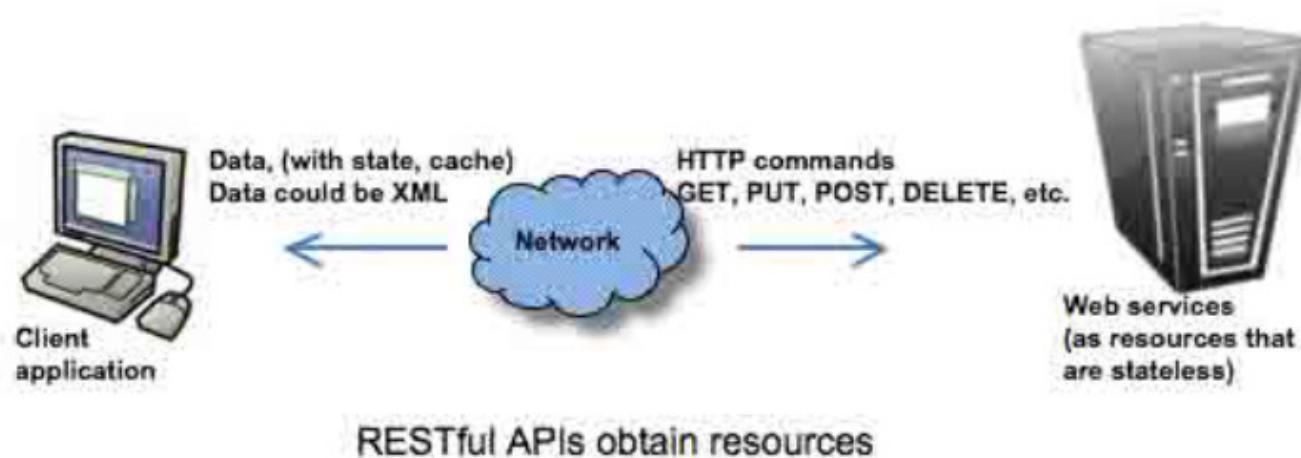
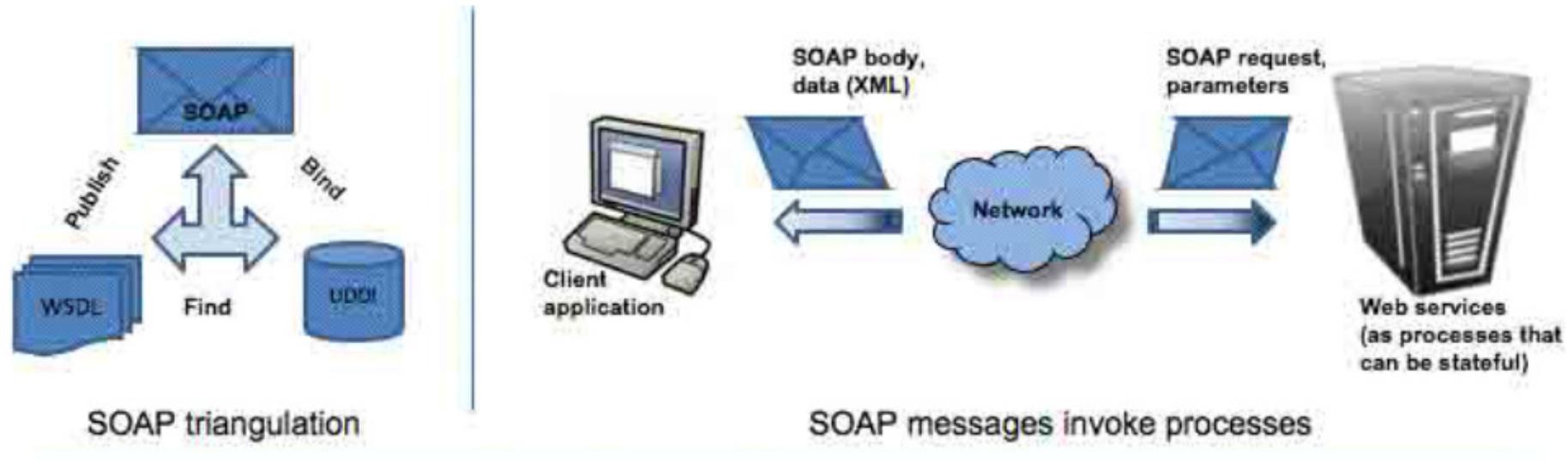


SOA with RESTful services

- The focus of a RESTful service is on resources and how to provide access to these resources
 - You can use any format for representing the resources: JSON or XML
 - The client and service interact *statelessly* and using *HTTP operations* explicitly: **POST (C), GET (R), PUT (U), DELETE (D)**
- but **there are principles** for designing RESTful Web services!



REST versus SOAP Web services





GraphQL (Graph Query Language)

- By Facebook around 2012; open-sourced in 2015
- JSON-based **query language for Web APIs**
 - Do not think of data as resource URLs or joint tables, but rather as a *graph of objects* that could be queried within application models
- GraphQL can bundle **lots of data into one query**

GraphQL: one query instead of many

The screenshot shows a GraphQL playground integrated into a Pinterest-like interface. The user has performed a search for "batman". The results page displays several pins, some of which are highlighted with red boxes. Below the results, a code editor window shows a GraphQL query:

```
POST /graphql
{
  pins(last: 10) {
    imgUrl
    name
    likes
  }
}
```

REST: One API Call for collection.
Then one call per item for details

The screenshot shows a REST API playground integrated into the same Pinterest-like interface. The results page displays the same pins as the GraphQL version, with some highlighted by red boxes. Below the results, a code editor window shows multiple separate GET requests:

```
GET /results
GET /results/1234
GET /results/5678
GET /results/9012
```

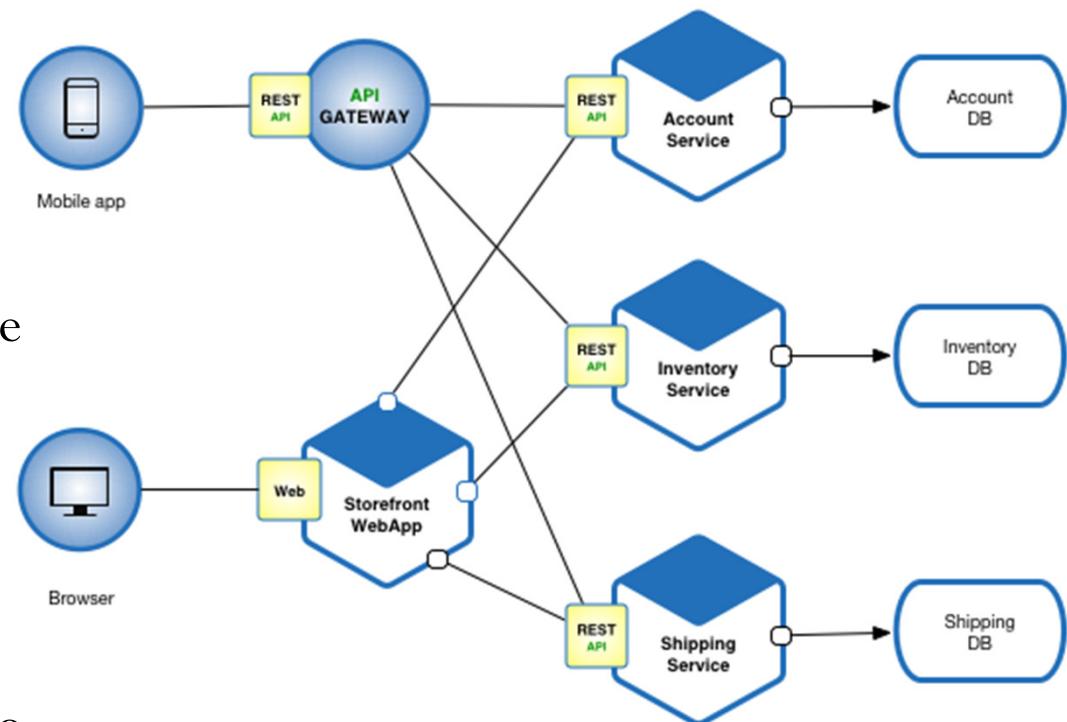
From services to microservices

- Microservices architecture **emerged (2010) from experiments** relating to changes in the service architecture
 - Many teams have tried to apply ideas in new ways of service architecture, combining them with agile practices and with the possibilities offered by virtualization and the cloud
 - Successful cases: Amazon, Ebay, Groupon, Netflix, Spotify, Uber, Zalando and Zoom
- It is a recent architectural style and its definition **cannot yet be considered consolidated**
- A common reason for adopting the microservices architecture is **make a monolithic application scalable**

Microservice Architecture

Microservices is **an architectural style** that structures an application as a collection of services that are

- Loosely coupled
- Organized around *business capabilities*
- Owned by a small team
- Highly maintainable and testable
- Independently deployable
- Enables the rapid, frequent, scalable and reliable delivery of large, complex applications
- It also enables an organization to evolve its *technology stack*

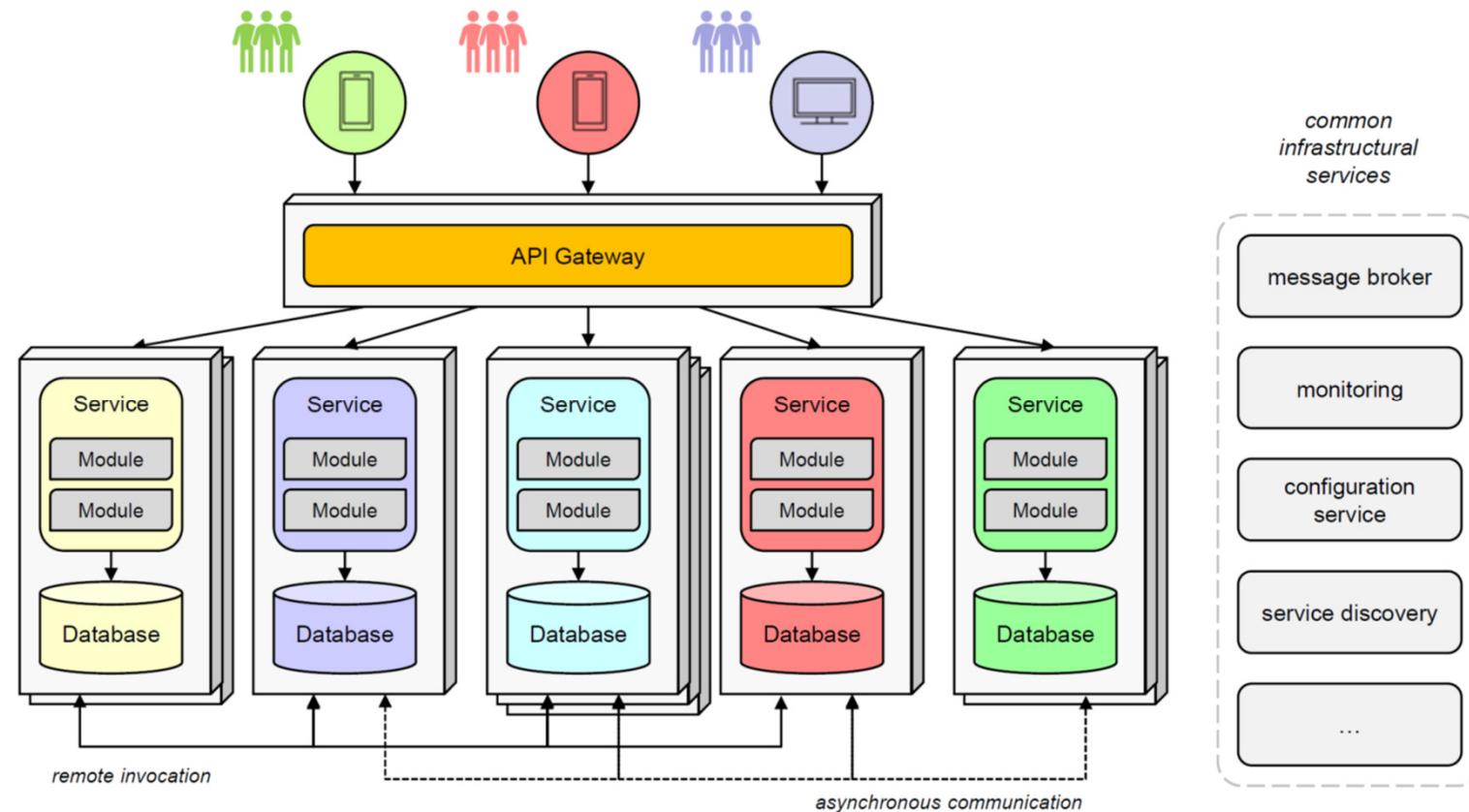


From *microservices.io*

Microservice architectural style

- The application clients usually interact with the microservices indirectly, through an *API gateway* (server-side unified access point to microservices)
- A minimum of centralized control and infrastructural support is required!

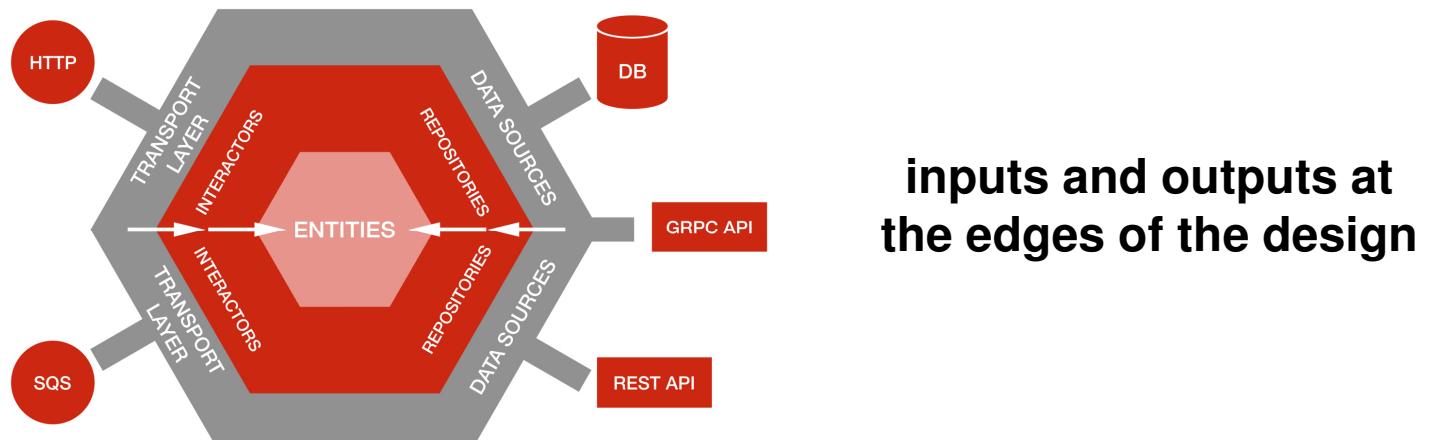
[see dedicated lesson!]



Microservices and hexagonal architecture

A micro-service as a "Hexagon"

- **Business logic** (the **core layer + ports**):
 - **Entities** are the **domain objects** (e.g., a movie)
 - **Repositories** are the **interfaces to getting entities from data sources** as well as creating and changing them
 - **Interactors** are classes that **orchestrate and perform business logic functions**
- **Adapters** (provide/require via open communication protocols http, REST, gRPC, etc.)
 - Data sources are adapter to different storage implementations
 - Transport layer controllers can trigger an interactor to perform business logic



From: Netflix Technology Blog Mar 10, 2020

Cloud-based architectures & Cloud Computing

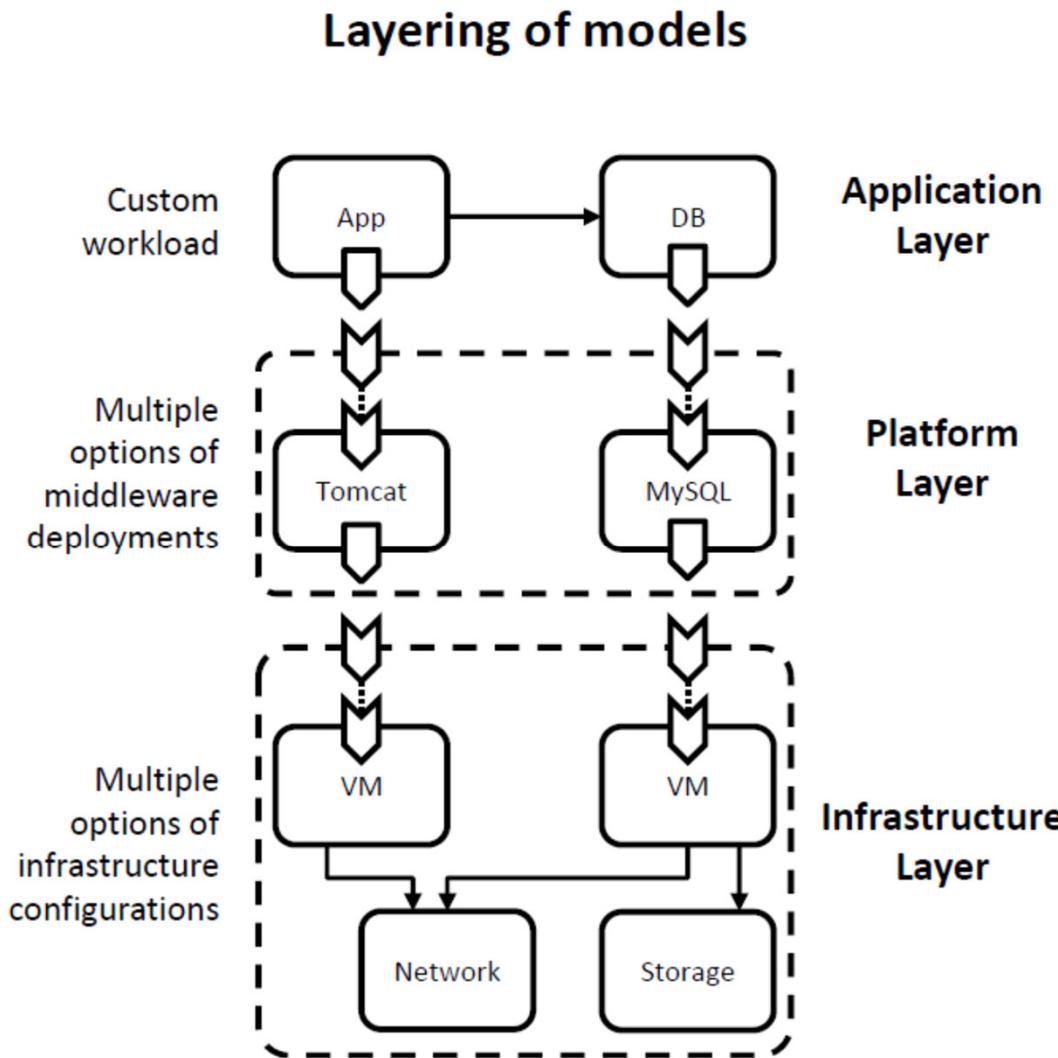
Cloud computing

- **A switch in the IT world:**
from in-house computing power (mainframes)
into utility-supplied computing resources delivered
over the Internet as Web services (Clouds)
- **Computing delivered as a utility:**
 - “on demand delivery of infrastructure, applications, and business processes in a security-rich, shared, scalable, and based computer environment over the Internet for a fee” [*]
 - Model pay-as-you-go
 - Elasticity and self-service: the illusion of infinite computing resources in any quantity at any time without intervention of human operators
 - Adoption of virtualization on server systems



[*] M. A. Rappa, The utility business model and the future of computing systems, IBM Systems Journal, 43(1):32-42, 2004.

Cloud architecture stack



Software as a Service (SaaS)

Offering Web services accessed by end users (e.g., word processing and spreadsheet)

- Example of SaaS provider: Salesforce.com

Platform as a Service offers platforms to build software, applications, programs and web tools

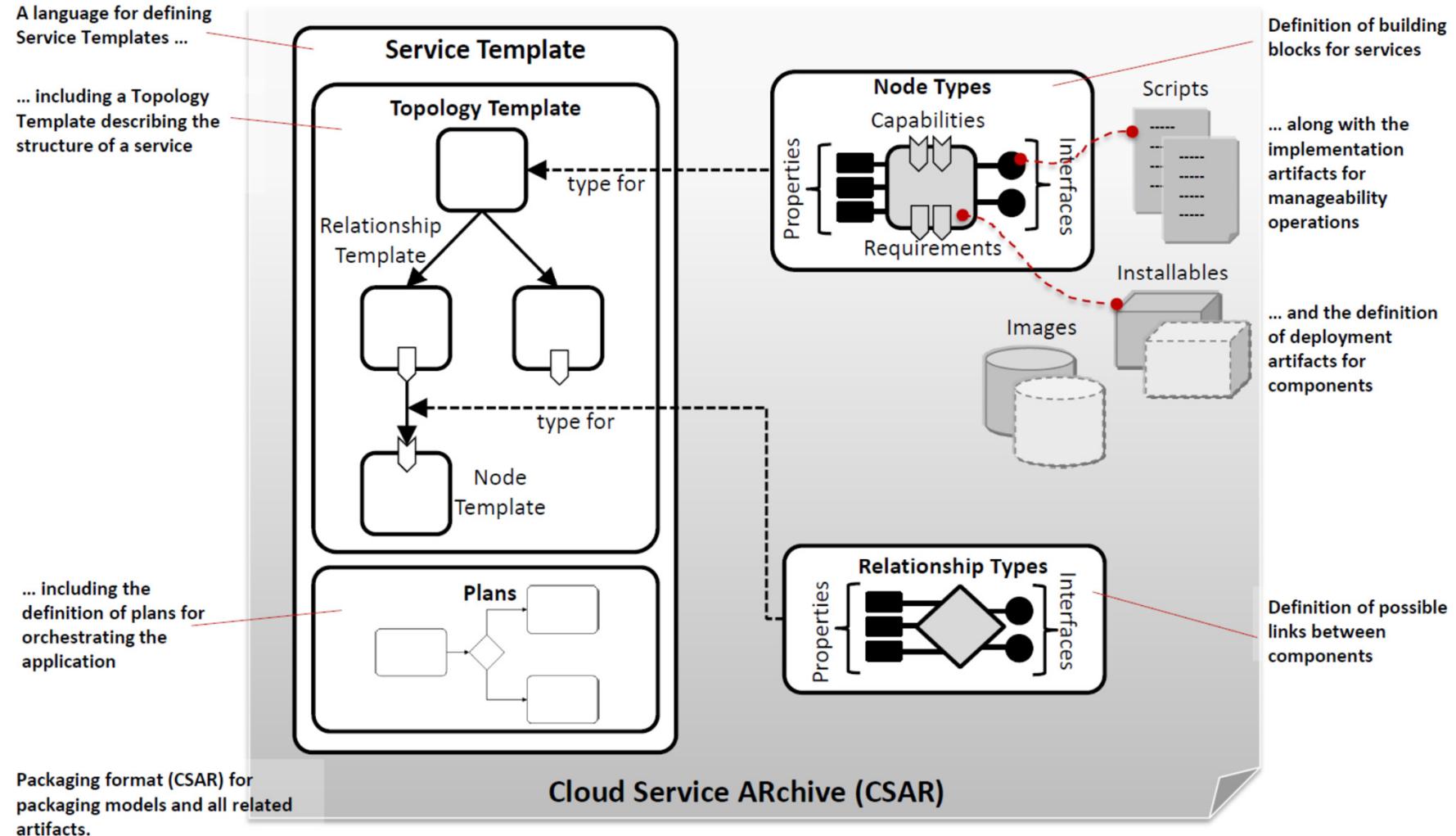
- Example of platform provider: Google AppEngine

Infrastructure as a Service (IaaS)

(IaaS) Offers virtualized resources (computation, storage, and communication) on demand

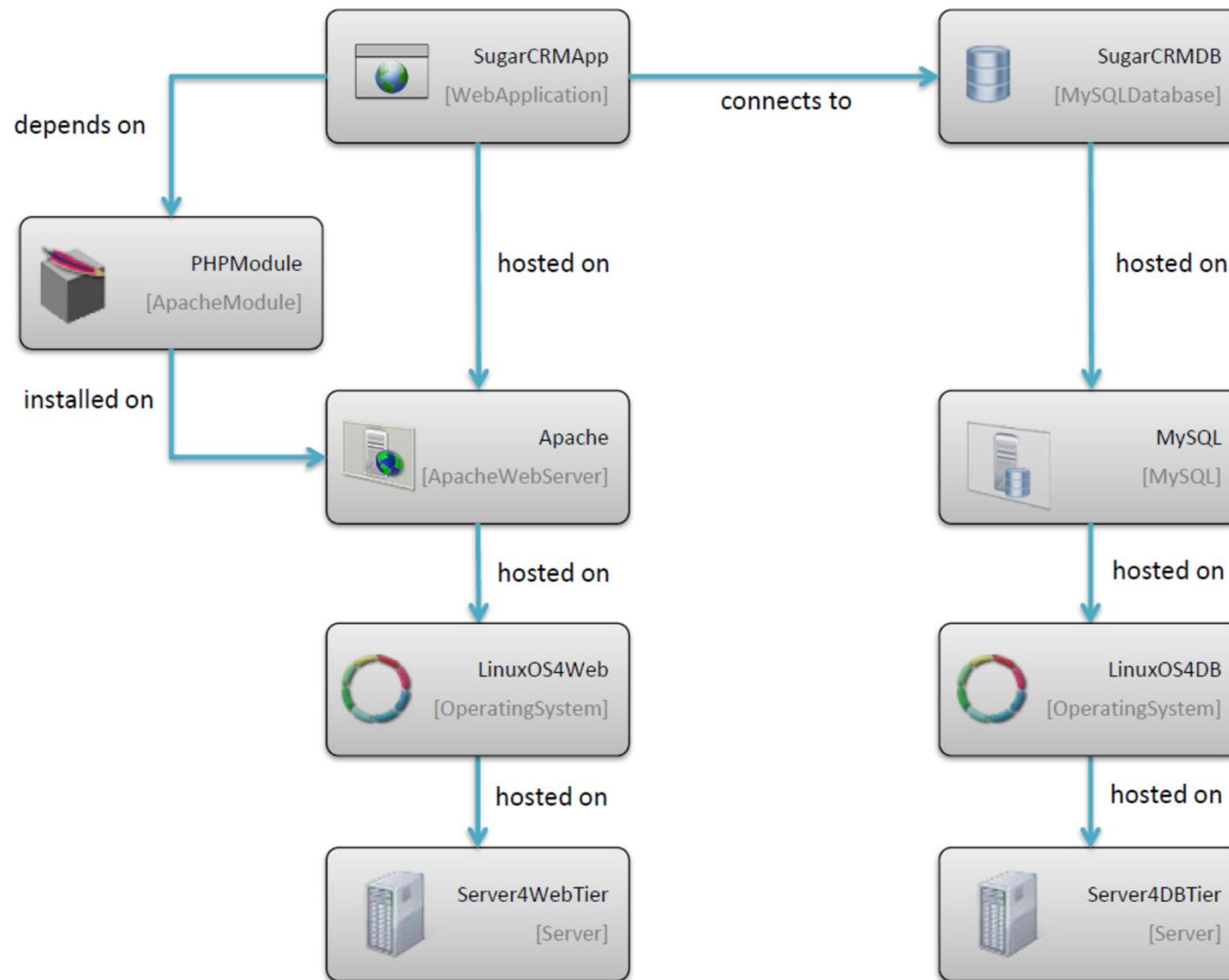
- Example of Infrastructure provider: Amazon EC2

OASIS's standard TOSCA (Topology and Orchestration Specification for Cloud Applications)



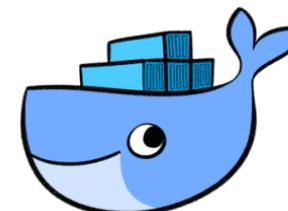
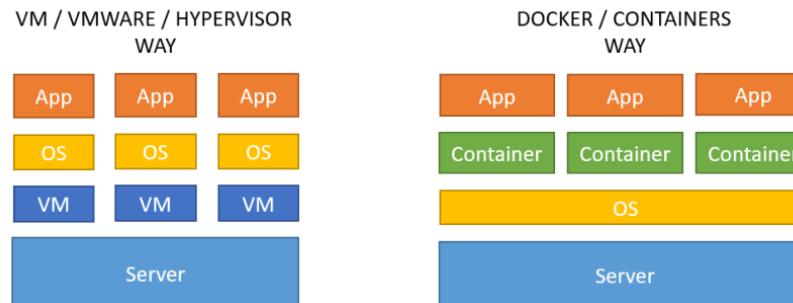
TOSCA service and topology concepts

Example: SugarCRM topology in TOSCA



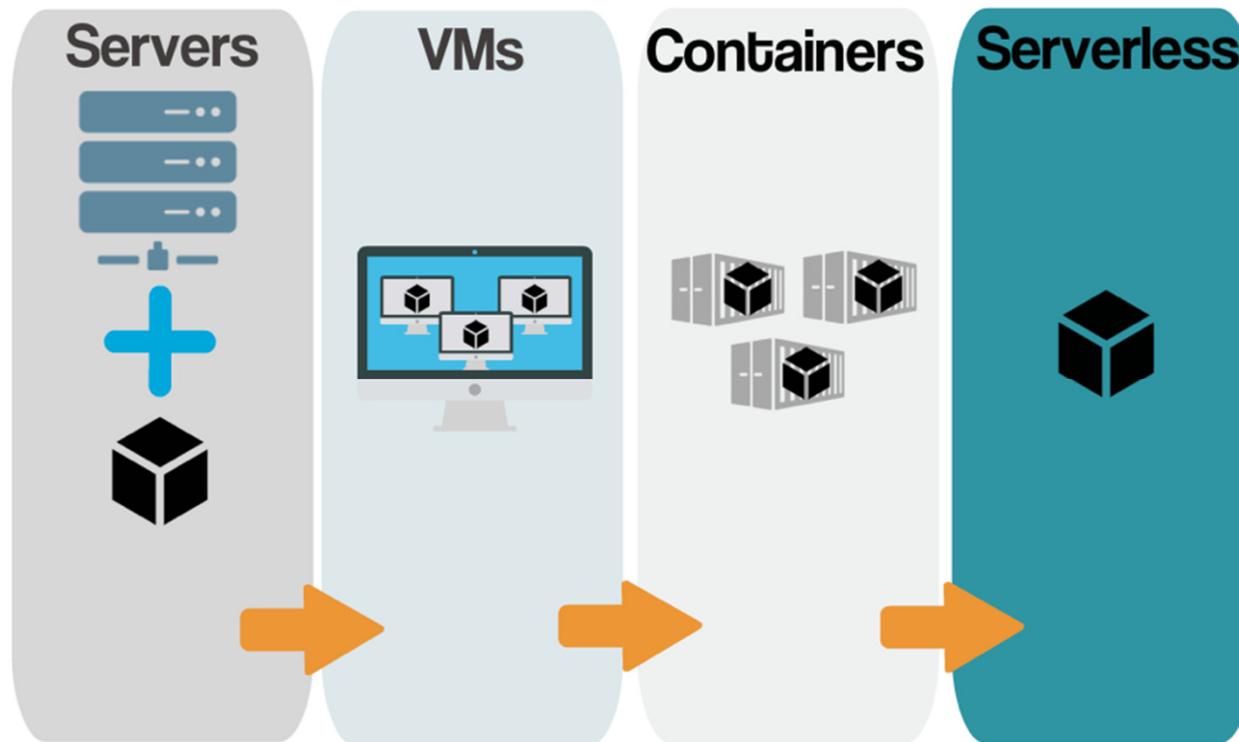
Cloud containerization and microservices

- “**Containerization**”: packaging of software code with just the OS libraries dependencies required to run the code.
 - More portable and resource-efficient than virtual machines
 - de facto *compute units* of modern *cloud-native applications*
- **Microservices** can be placed within **containers** that have the smallest libraries and executables needed by that service
- **Container-as-a-Service**: PaaS host and manage containers using OS-level virtualization; most popular ones are *Docker* and *Kubernetes*



Serverless computing: FaaS (Function as a Service)

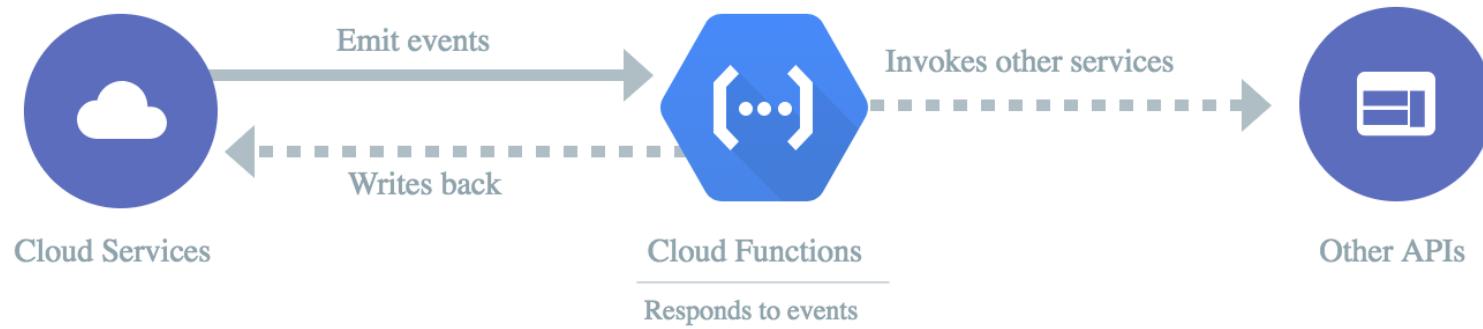
- What's next? *Serverless functions*



From: <https://www.spindox.it/it/blog/serverless-vs-docker/>

Serverless Compute Platform

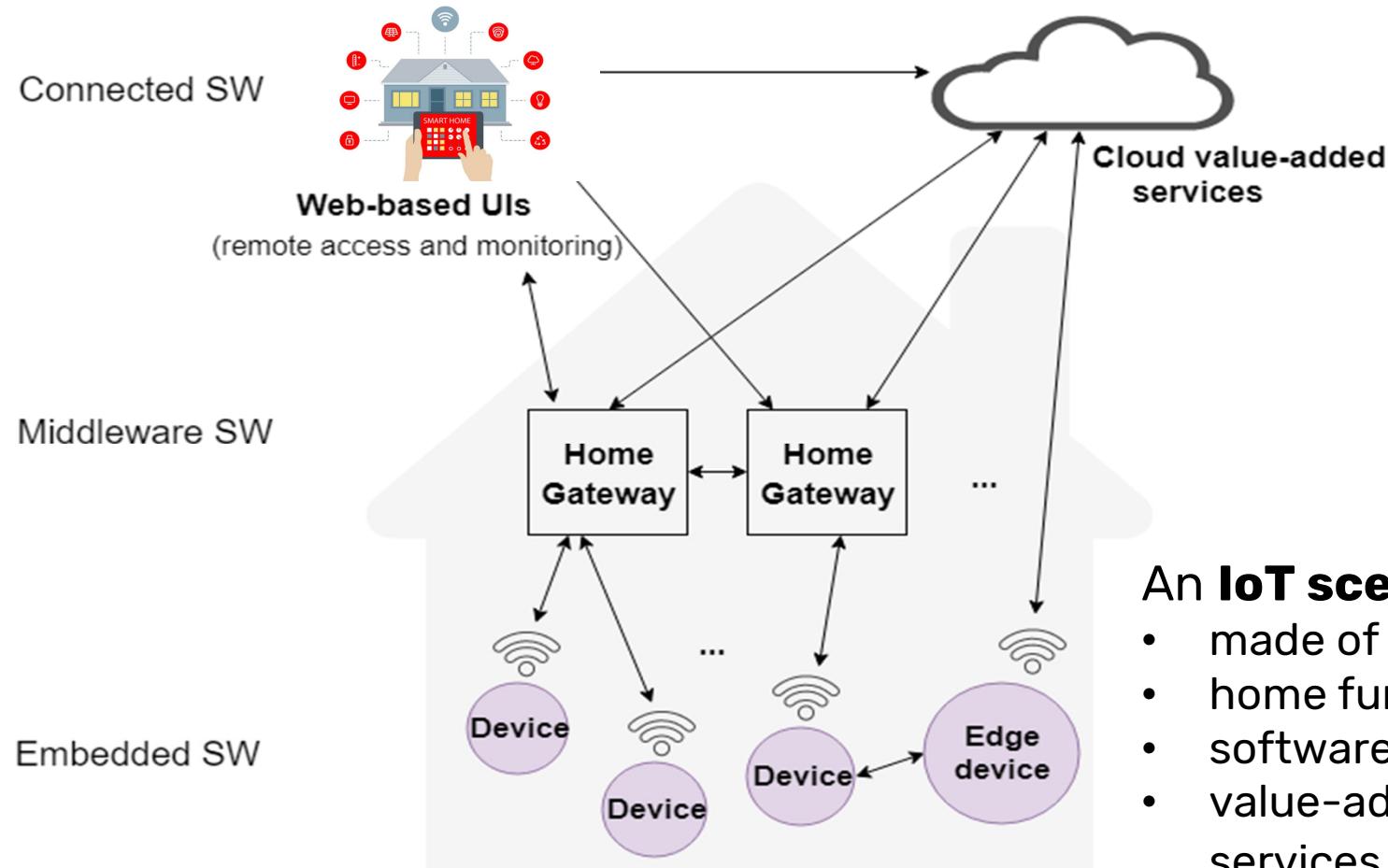
- *Cloud-native* architectures with *server-less microservices*
- Example:
 - **AWS Lambda**
 - **Google Cloud Functions**, an Event-driven Serverless Compute Platform
 - **Microsoft Azure Functions**



IoT-Cloud IoT-Edge-Cloud
computing

Gateway Integration Pattern, Cloud Integration Pattern

- Common architectural style for IoT-based applications
- **Example:** *smart home automation, homecare*
 - Centralized home automation control and connection to the Cloud via gateway(s) and edge devices

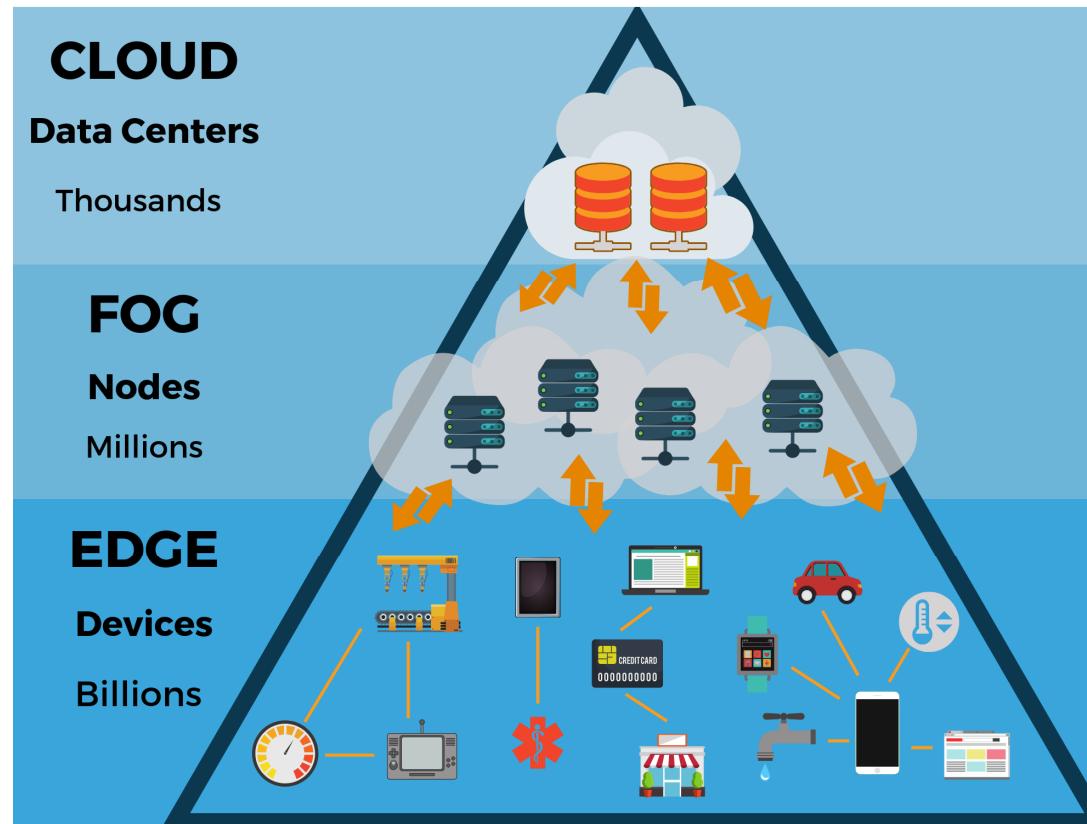


An IoT scenario

- made of home devices
- home functions
- software platforms
- value-added business services

Edge computing

- A distributed computing paradigm that brings computation and data storage closer to the sources of data
- Edge devices can be laptops, sensors, smartphones, gateways, etc.



Autonomic computing

Self-adaptive software systems

- The available knowledge at design time is not adequate to anticipate all the runtime operating conditions of a system
- Therefore, to achieve system objectives/requirements designers often prefer to deal with this uncertainty at runtime, when more knowledge is available to collect and reason about
- A self-adaptive system continuously monitors itself, gathers data, and analyzes them to decide if adaption is required to achieve system goals

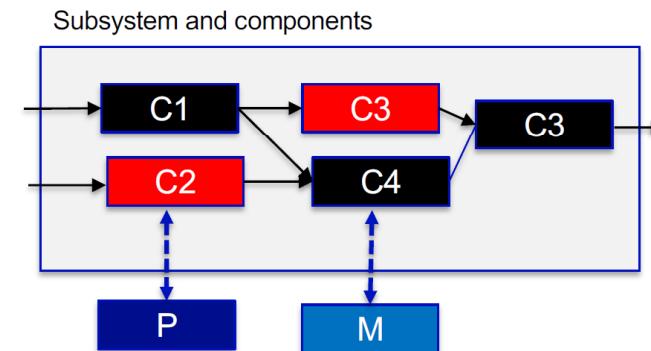
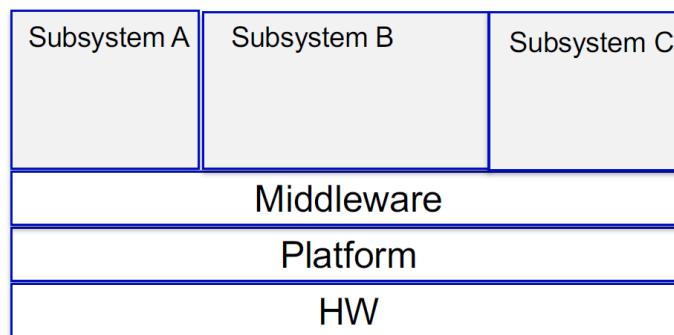
[see dedicated lesson!]



AI-based systems or autonomous systems (autonomic computing)

- ▶ Moving **from automation to self-adaptivity to autonomy**
- ▶ The **complexity of system design and analysis** is **exacerbated by** components encapsulating **AI-based functionality**
 - components with unpredictable behavior (e.g., machine learning)
 - the system's behavior is continually subject to and modified by the learning behavior
 - the system's behavior is unknown

AI-System - system architecture (example)



[from Ivica Crnkovic's keynote talk @ECSA2020, *AI engineering: New challenges in system and software architecting and managing lifecycle for AI-based systems*]



UNIVERSITÀ
DEGLI STUDI
DI BERGAMO

Dipartimento
di Ingegneria Gestionale,
dell'Informazione e della Produzione



Self-adaptive software systems

Corso di laurea
Magistrale in
Ingegneria
Informatica

PROGETTAZIONE, ALGORITMI E
COMPUTABILITÀ
(38090-MOD1)

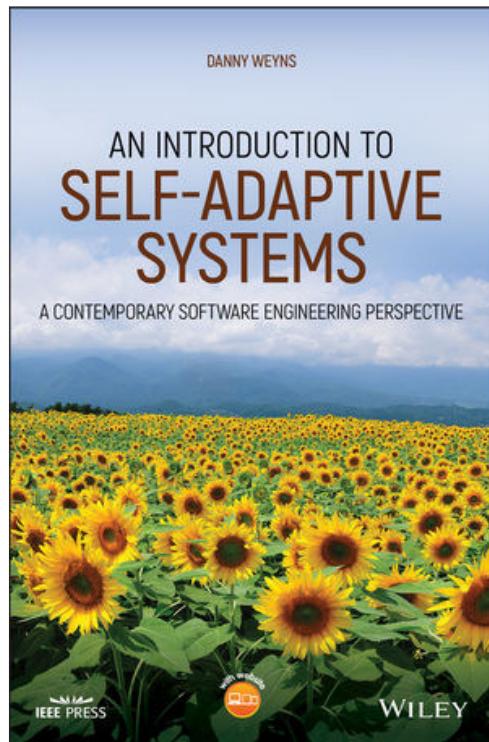
RELATORE
Prof.ssa Patrizia
Scandurra

Outline

- Foundations and engineering principles of *self-adaptation*
- Conceptual model
 - Why, What and How to engineer them?
- The MAPE-K feedback loop model: *architecture-based self-adaptation*
 - Architectural style
 - Examples
 - MAPE architectural patterns
- Other self-adaptation mechanisms

Main reference

- Danny Weyns. An Introduction to Self-adaptive Systems: A Contemporary Software Engineering Perspective. Wiley, Oct. 2020



- Webinar ICSE 2020 *Technical Briefing Self-Adaptive Systems* (D. Weyns)

<https://www.youtube.com/watch?v=f3LR7EBbd2I&feature=youtu.be>

Why self-adaptation?

Management of Computing Systems



Challenges

Complexity of systems
Uncertain operating conditions, 24/7

Implications

System qualities may degrade or worse
Manual approach: costly, prone to error, not timely

Problem

How to engineer systems such that they achieve their qualities and are trustworthy despite the uncertainties?

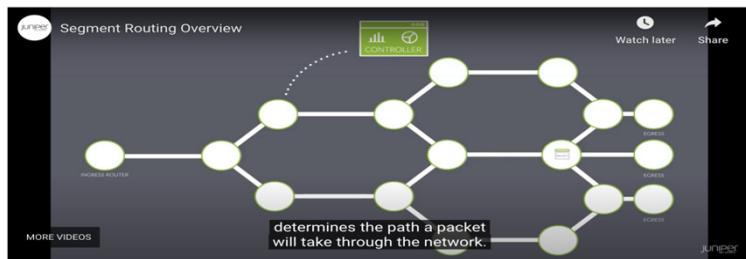
Answer:

the system adjusts itself autonomously in response to changes

- in the operational environment,
- in the system itself, and
- in its requirements



“Self-adaptation is everywhere”



From <https://www.juniper.net/>



From <http://www.jayatech.in/>



From <https://awsadvices.com/>



From <https://twitter.com/OracleDevs/>

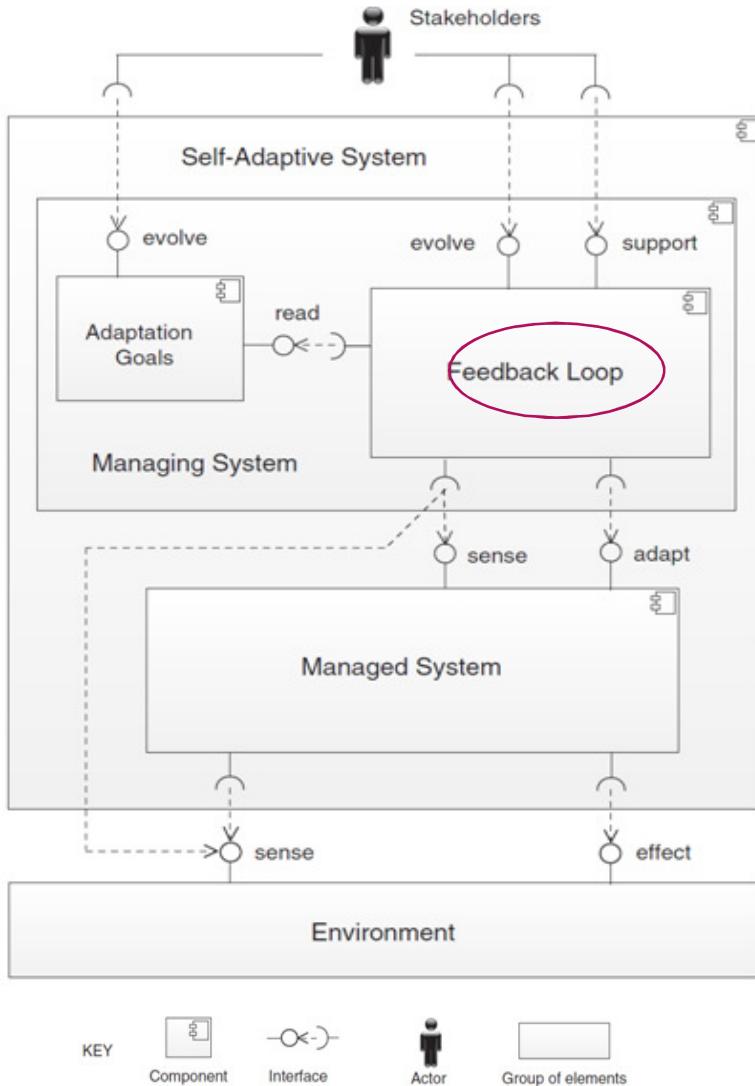
What is self-adaptation?

Basic principles

- There is no general agreement on a definition of the notion of *self-adaptation*
- However, there are *two complementary basic principles* that determine what a self-adaptive system is

1. ***External principle:*** A self-adaptive system is a system that can handle changes and uncertainties in its environment, the system itself, and its goals autonomously (i.e. without or with minimal required human intervention).
2. ***Internal principle:*** A self-adaptive system comprises two distinct parts: the first part interacts with the environment and is responsible for the domain concerns – i.e. the concerns of users for which the system is built; the second part consists of a feedback loop that interacts with the first part (and monitors its environment) and is responsible for the adaptation concerns – i.e. concerns about the domain concerns.

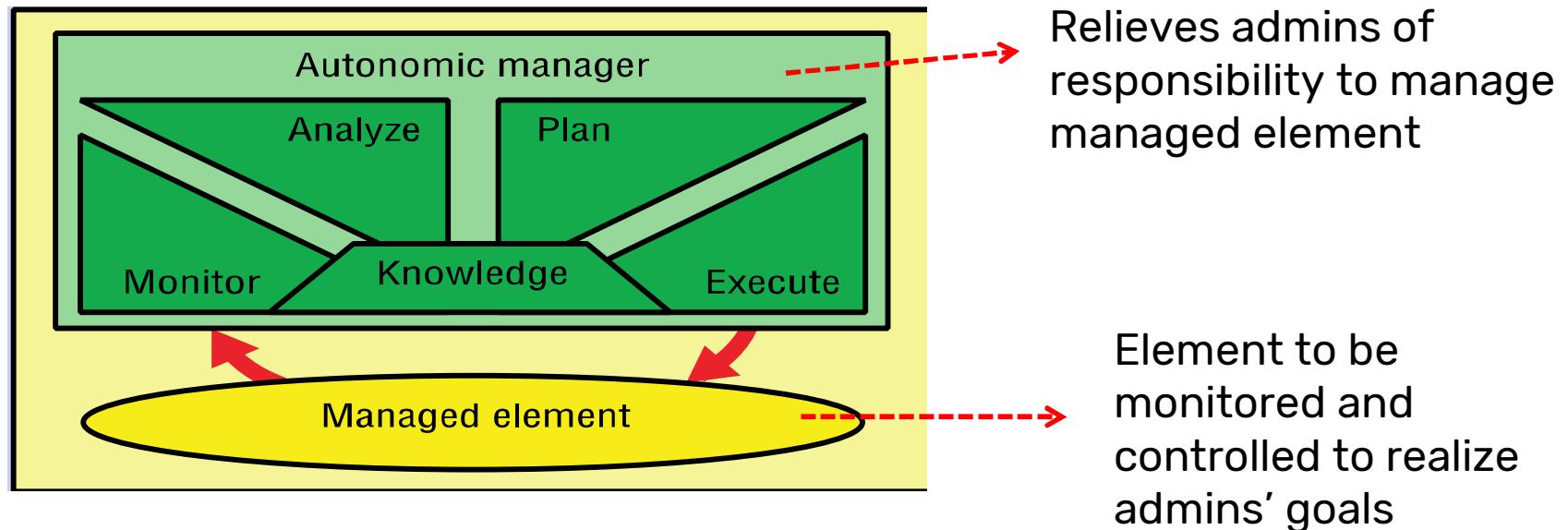
Conceptual Model of a Self-Adaptive System architecture



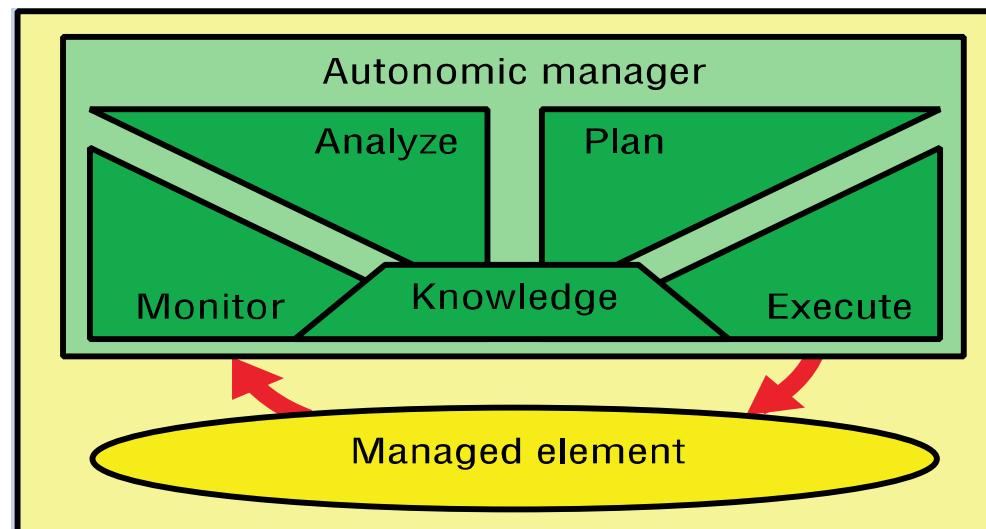
A **feedback loop realizes the adaptation goals** by monitoring and adapting the managed system

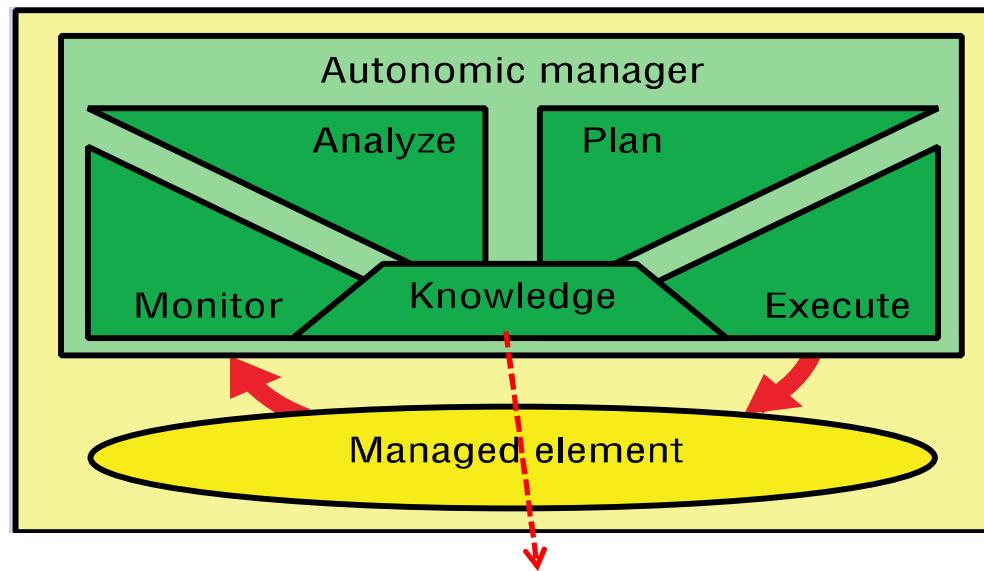
- **Reactive adaptation**: the feedback loop **responds to a violation** of the adaptation goals by adapting the managed system to a new configuration that complies with the adaptation goals
- **Proactive adaptation**: the feedback loop tracks the behavior of the managed system and adapts the system to **anticipate a possible violation** of the adaptation goals

IBM's MAPE-K reference model for autonomic systems



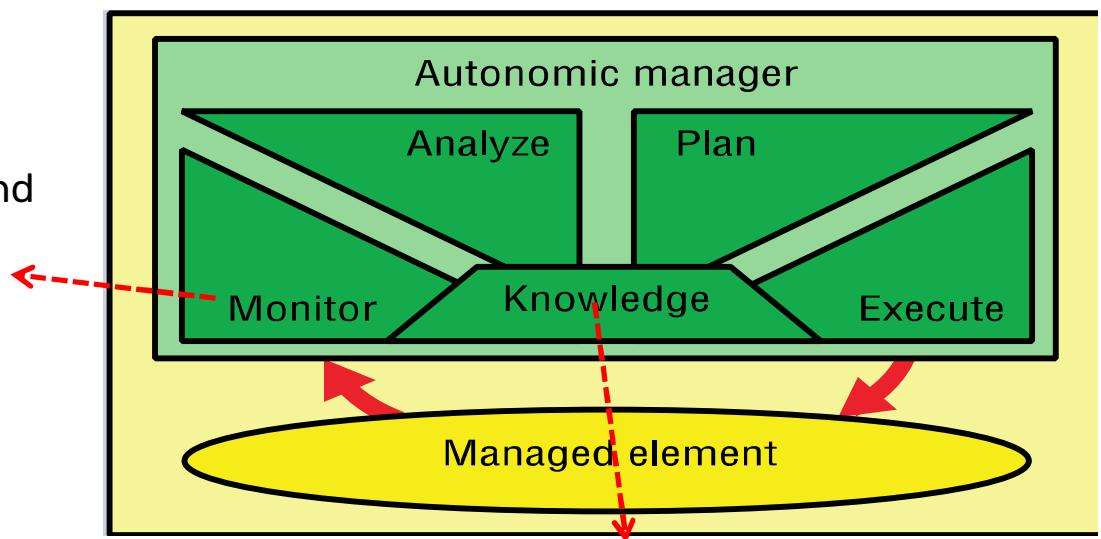
- Kephart, J.O., Chess, D.M.: The vision of autonomic computing. *IEEE Computer* 36(1), 41–50 (2003)
- IBM Corporation: An architectural blueprint for autonomic computing. White Paper, 4th edn., IBM Corporation



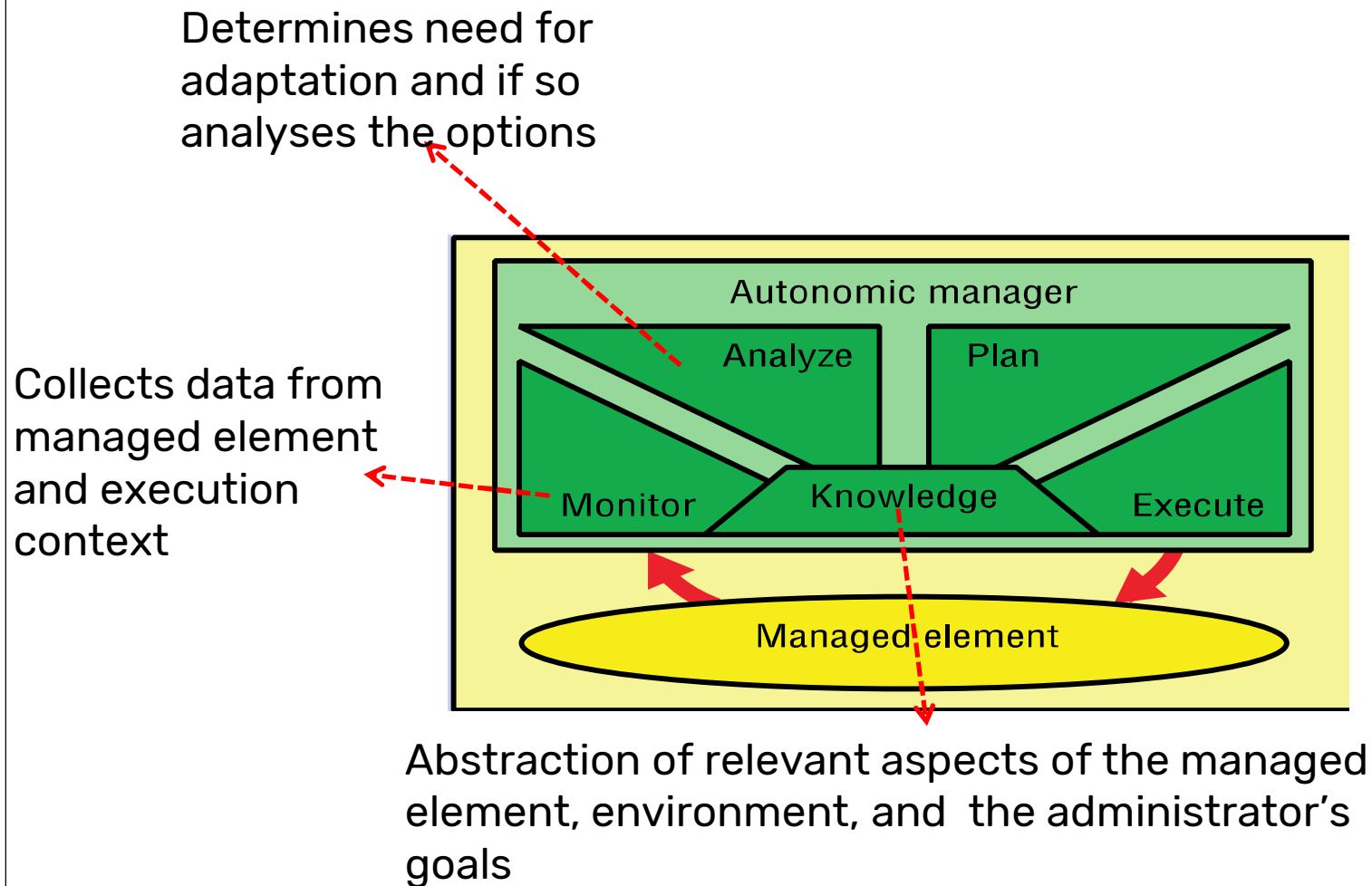


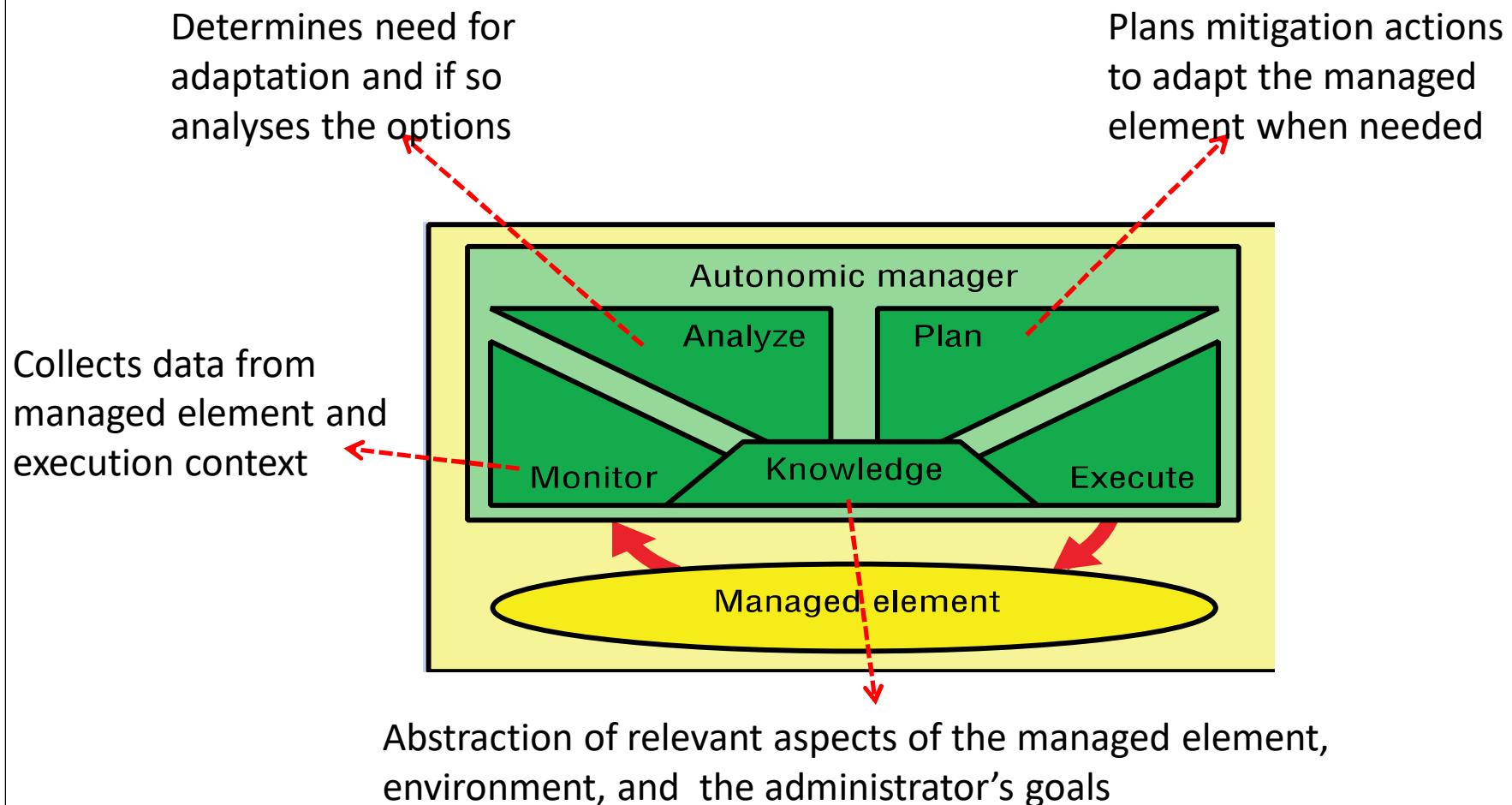
Abstraction of relevant aspects of the managed element,
environment, and the administrator's goals

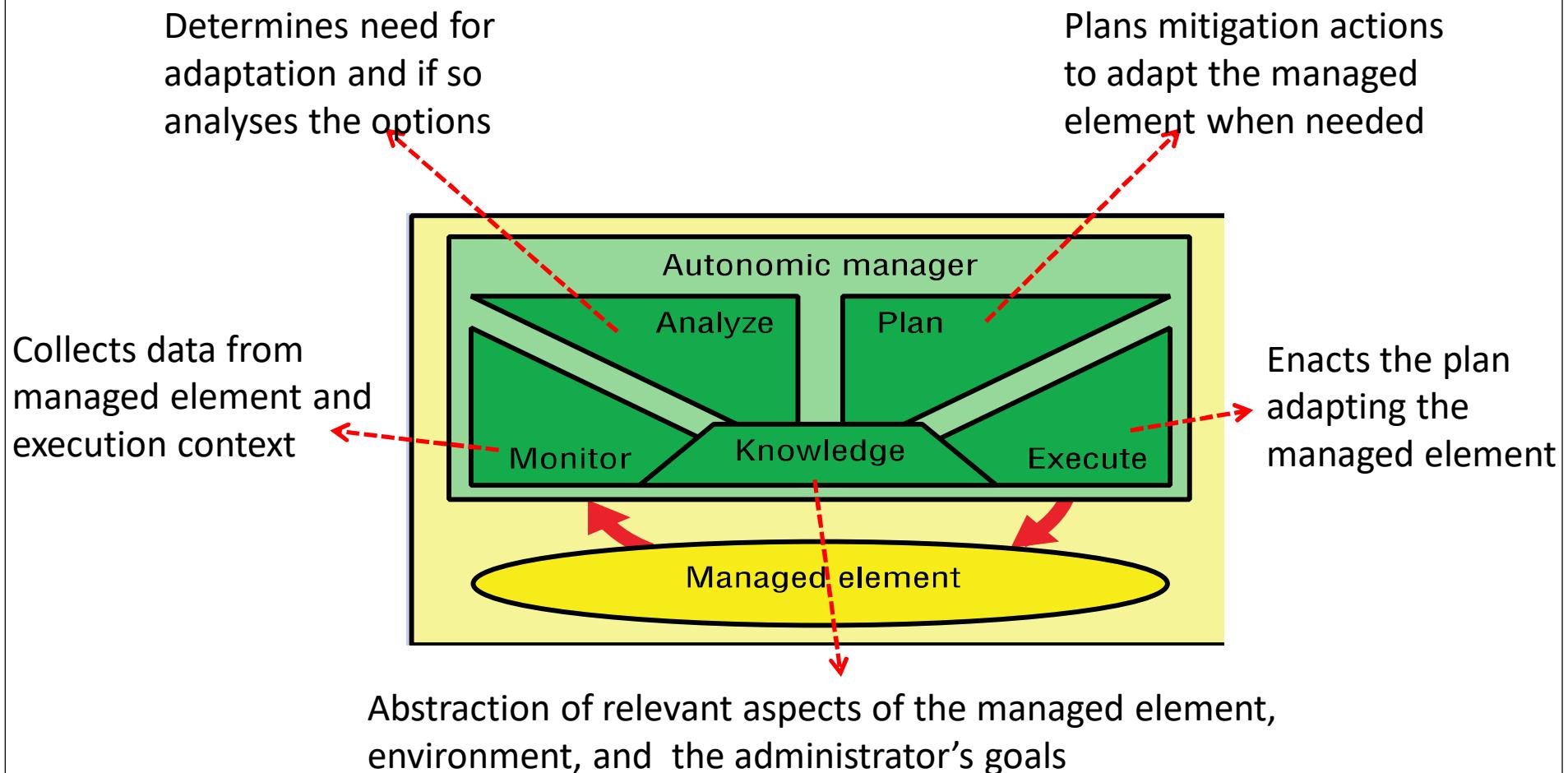
Collects data from managed element and execution context



Abstraction of relevant aspects of the managed element, environment, and the administrator's goals







Plan phase of a MAPE loop

- A variety of methodologies and analysis based on runtime analytical models can be used to plan the proper adaptation
 - Optimization theories
 - Heuristics
 - Machine learning
 - Queueing theory
 - Formal analysis (e.g., runtime verification)
 - ...
- Example: QoS-driven self-adaptation of service applications
 - Plan: select optimal set of concrete services (and their coordination) by means of linear programming optimization

Utility functions

- When an autonomic system needs **to make an adaptation decision**, there are usually multiple options to select from
- Utility functions are used to **compute the expected utility of choices** among different options
 - i.e., a preference ordering over the choice set
- The expected **utility for a configuration c** is defined as follows:

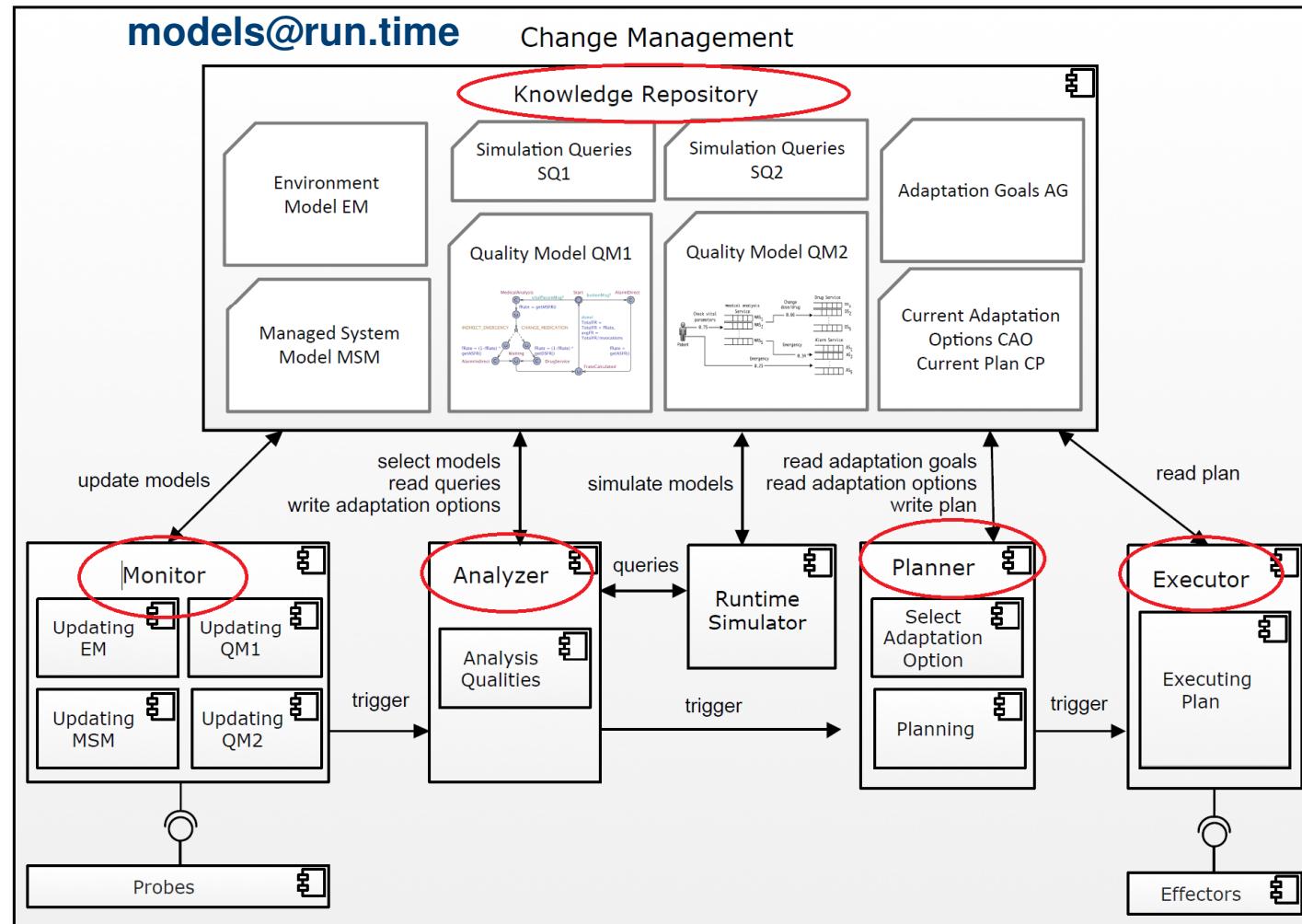
$$U_c = \sum_{i=1}^n w_i \cdot p_i$$

p_i utility **preference** of the stakeholders for property i of n properties

w_i relative **weight** for property i of n properties

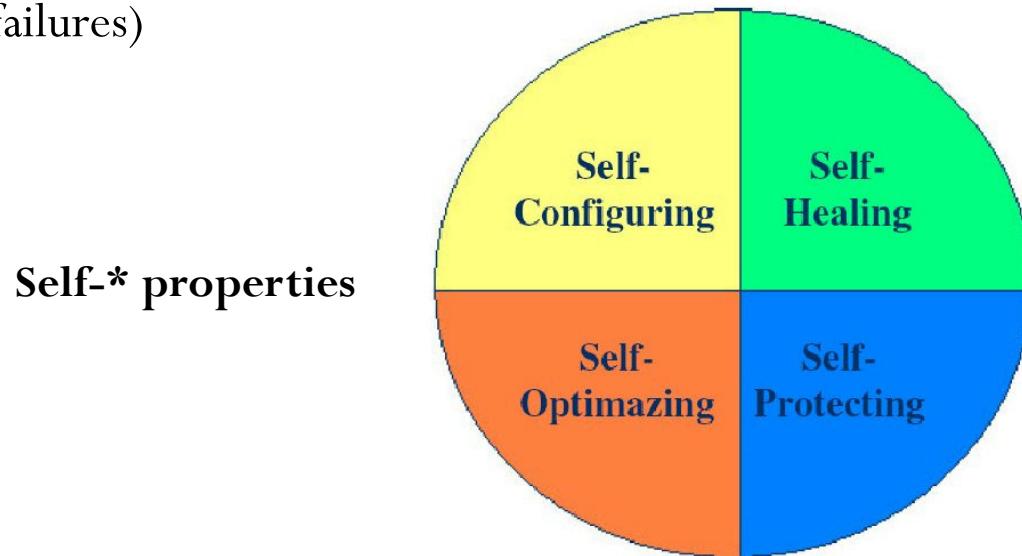
Within the knowledge of a MAPE-K architecture

Runtime models (of system/context/funct. and QoS requirements) serve as a knowledge-base and their analysis supports decision making about adaptation



Adaptation goals and autonomic system properties

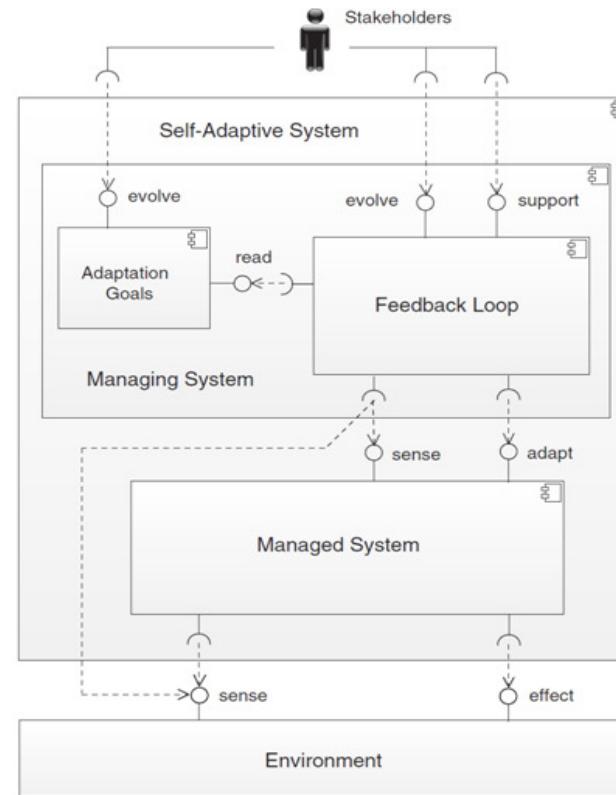
- Four principal types of high-level adaptation goals:
 1. **self-configuration** (i.e. systems that configure themselves automatically)
 2. **self-optimization** (systems that continually seek ways to improve their performance or reduce their cost)
 3. **self-healing** (systems that detect, diagnose, and repair problems resulting from bugs or failures)
 4. **self-protection** (systems that defend themselves from malicious attacks or cascading failures)



How the conceptual model apply?

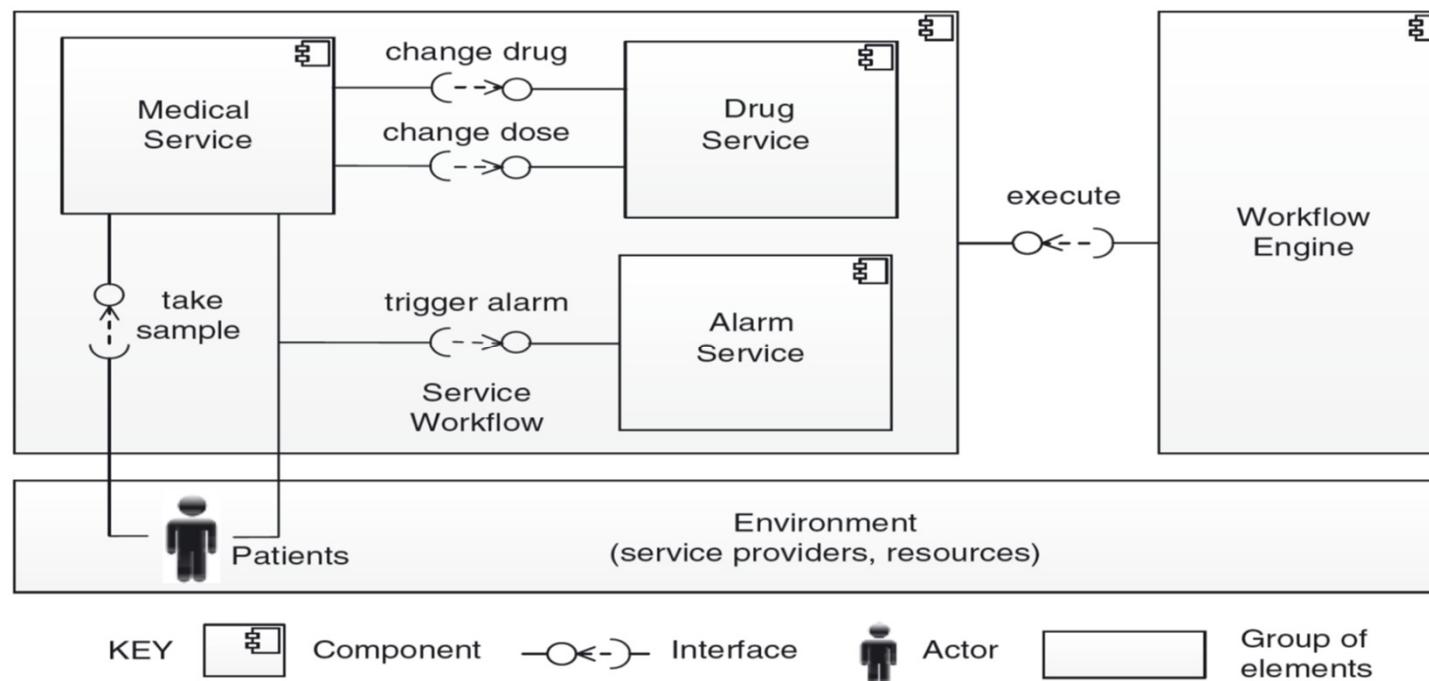
Two examples:

- Example 1 (**centralized adaptation**): **Tele Assistance System (TAS)**
- Examples 2 (**decentralized adaptation**): **Traffic Monitoring Application (TMA)**



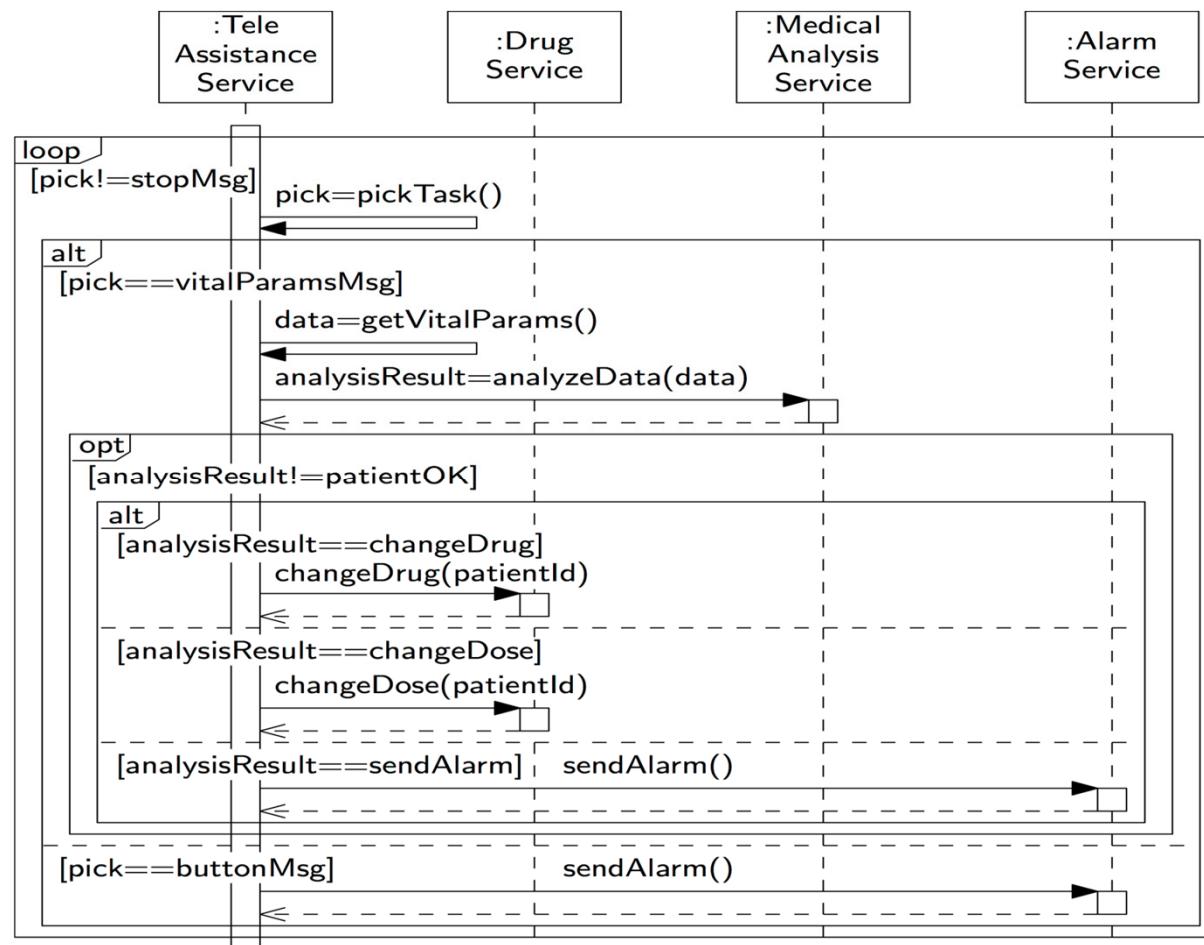
Example 1: TAS (without self-adaptation)

- Each **drug service type** can be realized by **one of multiple service instances** provided by **third-party service providers** and may have **different quality properties**, such as **failure rate** and **cost**.



Example1 : TAS (without self- adaptation)

- The result of an analysis may trigger:
 - a *pharmacy service* to deliver new medication to the patient or to change dose of medication, or
 - an *alarm service* that will send a medical assistance team to the patient.
- The alarm service can be invoked also through a *panic button* by the patient



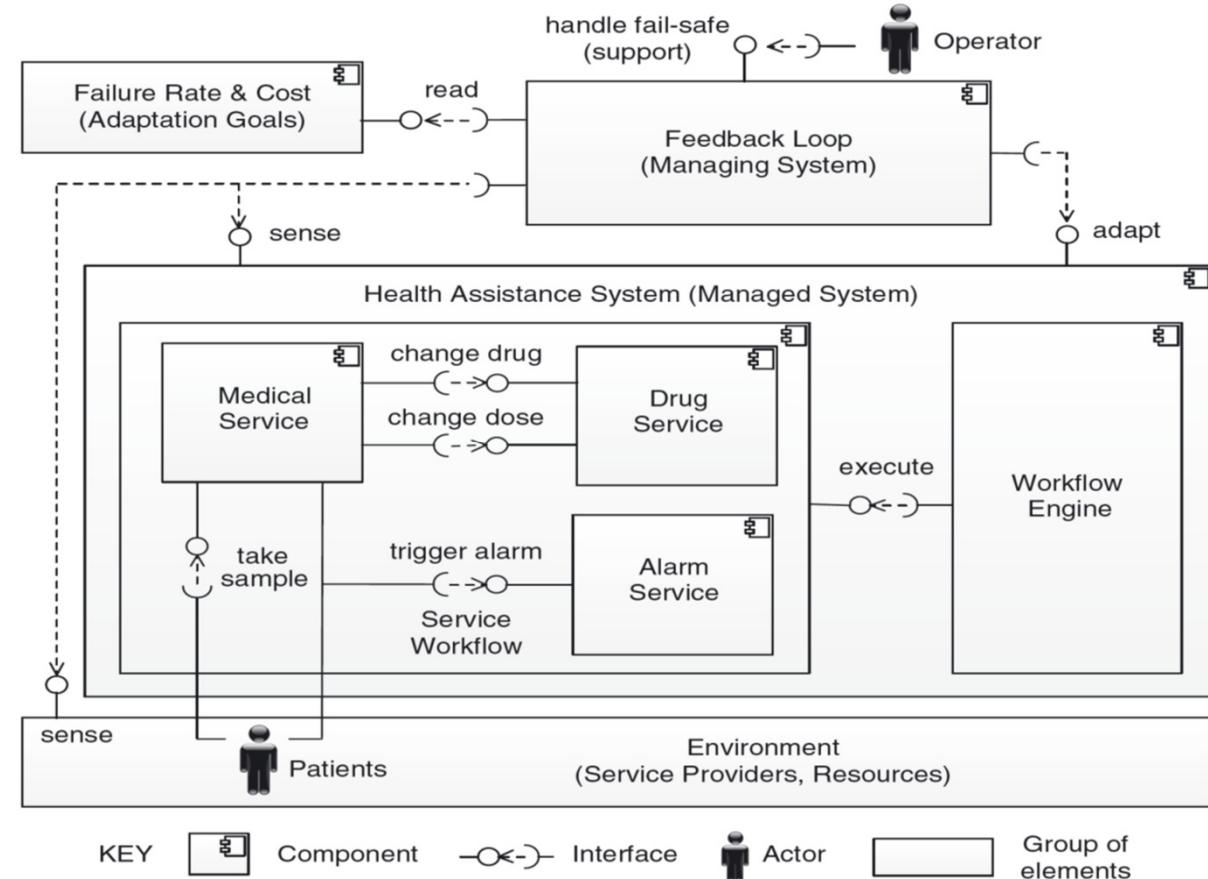
Example 1: TAS (with self-adaptation)

- **Variety of system' uncertainties:**
 - Services may fail, service response times may vary, or new services may become available
 - Users may even require new features or new services need to be integrated to the system, which were not anticipated upfront (e.g., in case of an alarm, relatives may need to be informed)
- **Types of adaptation actions** to deal with these uncertainties:
 - switch to equivalent service or
 - simultaneous invocation of several services for idempotent operations or
 - change the workflow architecture by removing failed services from the set of available services
- **Types of Adaptations goals:** minimize both the failure rate and the cost for using services, minimize also the time to adapt the system, fail-safe modes, etc.

Example 1: TAS (with self-adaptation)

See more details on the *Tele-Assistance System (TAS)* exemplar (in Java!)

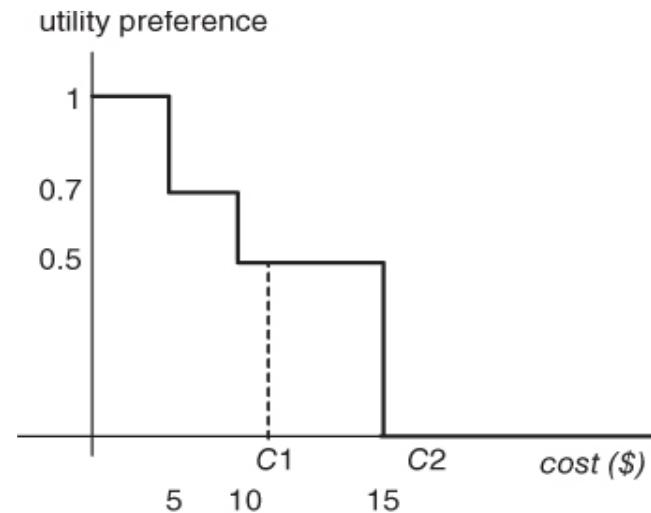
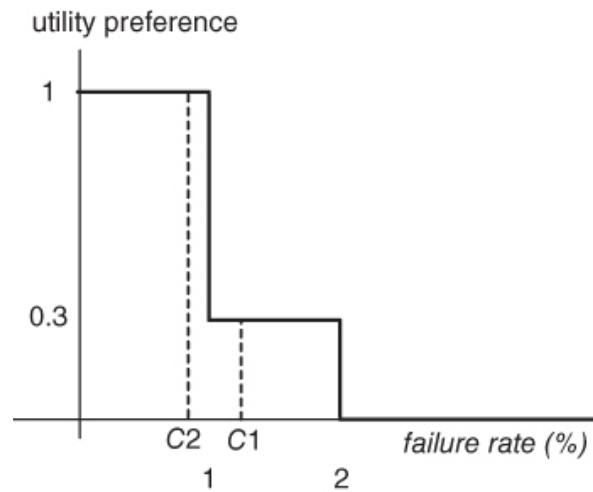
- <https://people.cs.kuleuven.be/~danny.weyns/software/TAS/>



Utility function – example

- Consider the **TAS exemplar**; two **quality properties** are the subjects of adaptation: **service failures** and **service cost**
- Suppose a new service configuration has to be selected from **two possible options**:
 - C_1 : failure rate of 1.2 % and cost of \$ 11
 - C_2 : failure rate of 0.9 % and cost of \$ 16

Assume the following utility **preferences**:



Utility function – example

- Furthermore, assume the **weights**: 0.7 for **failure rate**, 0.3 for **cost**
 - The expected **utility of service configurations** is:

$$U_c = w_{\text{failure_rate}} \cdot p_{\text{failure_rate}} + w_{\text{cost}} \cdot p_{\text{cost}}$$

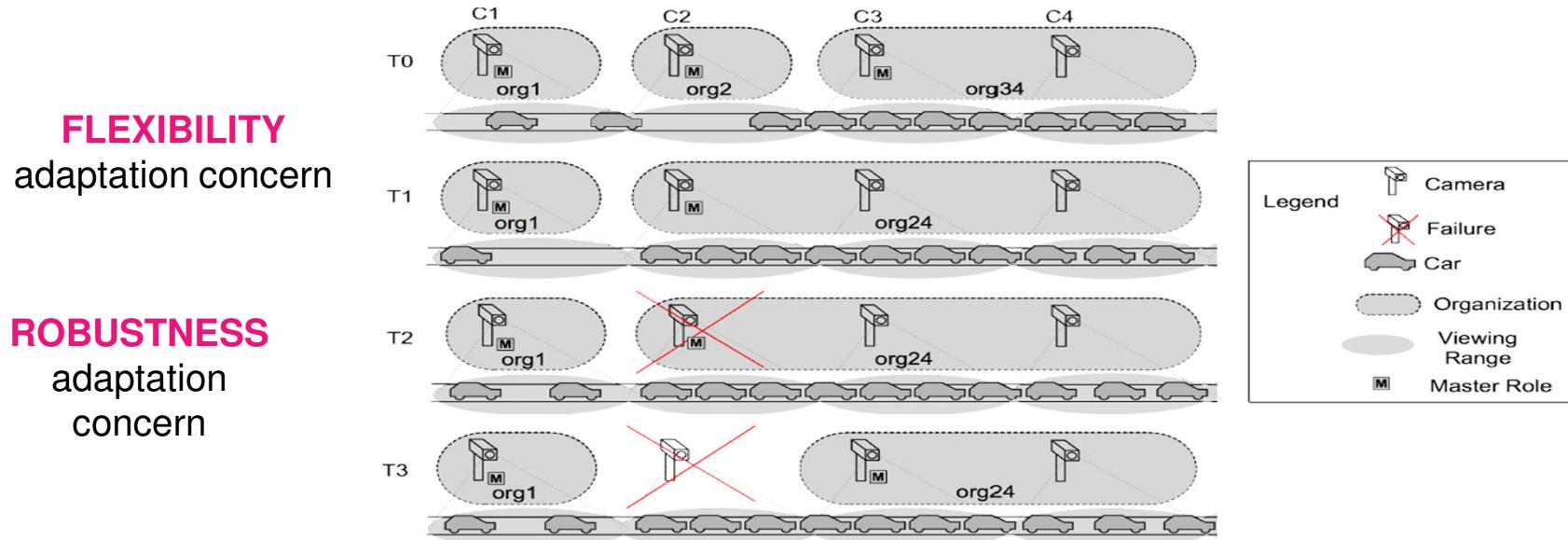
$$U_{C1} = 0.7 \cdot 0.3 + 0.3 \cdot 0.5 = 0.36$$

$$U_{C2} = 0.7 \cdot 1.0 + 0.3 \cdot 0.0 = 0.70$$

Highest expected utility ->
C₂ is selected for adaptation

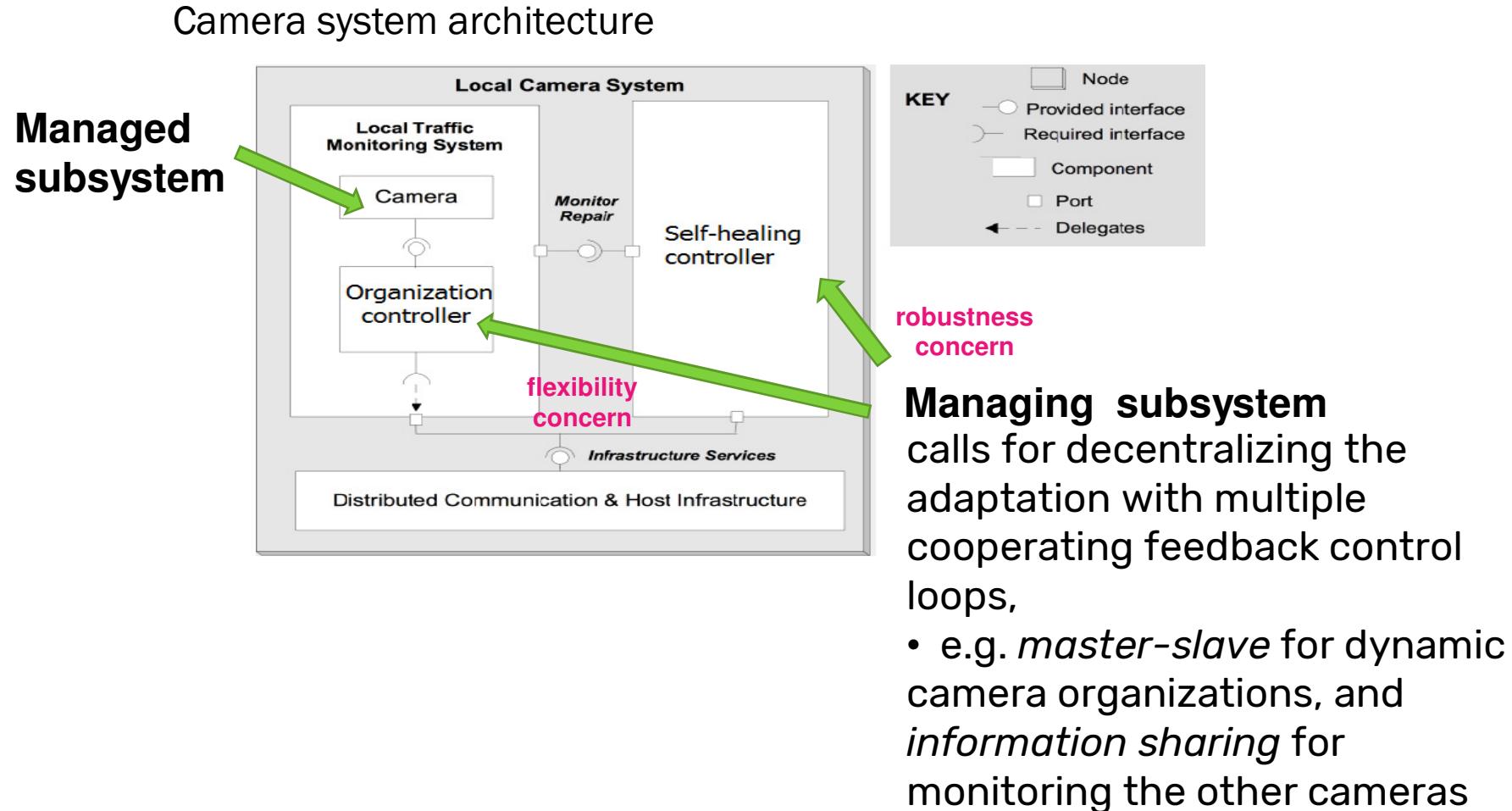
Example 2: TMA

- Intelligent cameras collaborate in dynamic *master/slaves organizations* to monitor and aggregate useful data whenever the traffic jam enters/leaves their viewing range
- The adaptation logic is to be conceived as **multiple interacting feedback loops**
 - Typically, **one loop per each adaptation concern** or goal



* M. U. Iftikhar and D. Weyns. A case study on formal verification of self-adaptive behaviors in a decentralized system. In FOCLASA 2012.

Example 2: TMA



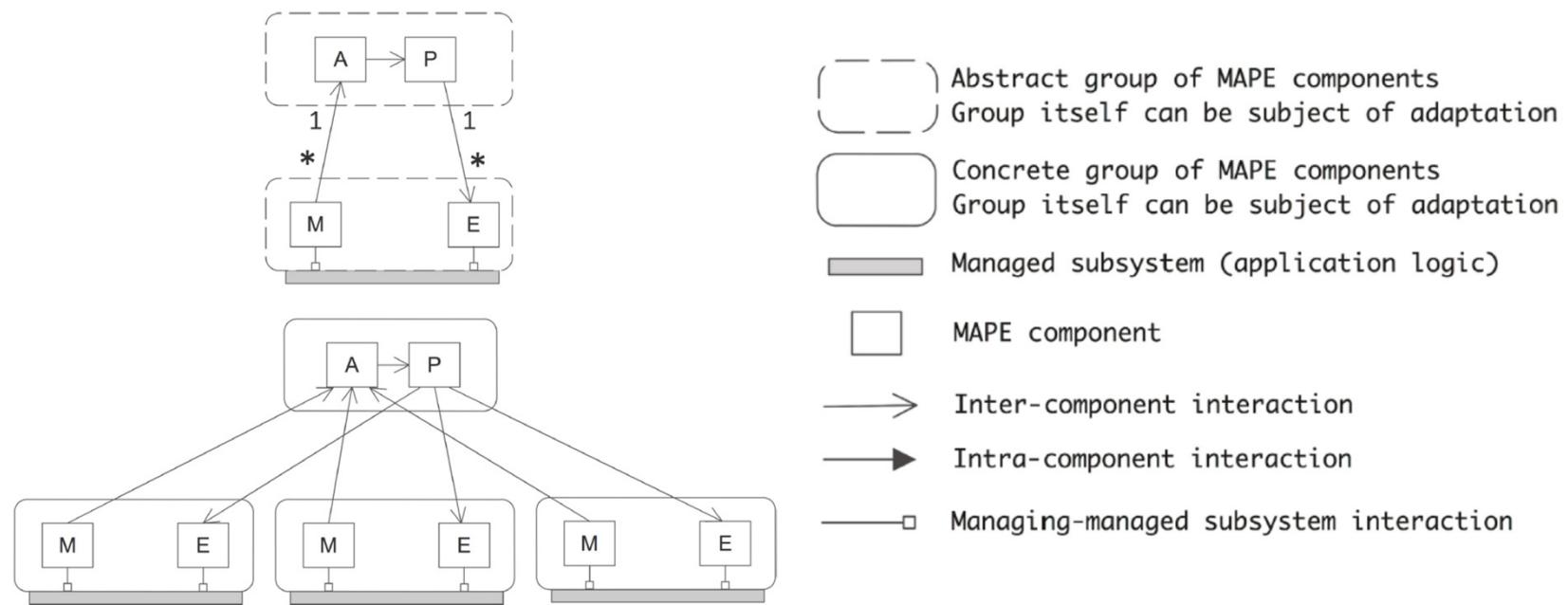
MAPE design patterns for decentralized self-adaptation

Design patterns for multiple interacting MAPE loops have been proposed [*] to decentralize the adaptation control

- e.g., aggregate, master-slave, information sharing, hierarchical, ecc.

[*] D. Weyns et al., Software Engineering for Self-Adaptive Systems II: Int. Dagstuhl Seminar, chapter "On Patterns for Decentralized Control in Self-Adaptive Systems", pages 76-107, Springer, 2013.

Example of MAPE pattern: Master-slaves MAPE loop



Other Adaptation Approaches



- *Autonomous systems* are built around the central idea to mimic human or animal behavior
- *Multi-agent systems*
 - architectures of **autonomous agents**, communication and coordination mechanisms, and
 - supporting infrastructure for the **knowledge representation** and its use to coordinate the autonomous agents
- *Self-organizing systems* emphasize decentralized control
 - simple reactive agents apply local rules to adapt their interactions with other agents in response to changing conditions
 - in order **to cooperatively realize the system goals**
- *Context-aware systems:*
 - Typically have a layered architecture, where a context manager/middleware is responsible for sensing and dealing with context changes