

Lezione 01

Università degli Studi di Padova

Scuola di Ingegneria

Corso di Laurea in

Ingegneria Informatica

Dati e Algoritmi (DA)

(ex Dati e Algoritmi 1)

Canale 1 o A (matr. 0-4)

Prof. Marcello Dalpasso

Sito Web del corso

- Tutte le **informazioni** relative a questo corso si trovano nel sito di Ateneo «Macroarea STEM»

<https://stem.elearning.unipd.it>

- Portale Web sviluppato con uno strumento open-source
- Attenzione: è **diverso** dal cosiddetto "**sito esami**" (<https://esami.elearning.unipd.it>)



- Sul sito del corso si troverà anche, giorno per giorno, **tutto il materiale didattico**

- copia di (quasi) tutte le presentazioni, in PDF
- esercizi proposti e risolti
- registrazione audio/video di (quasi) tutte le lezioni/esercitazioni
- ...

Studenti di questo canale

- Iscritti al Corso di Laurea in
Ingegneria Informatica con ultima
cifra del numero di matricola che
termina con **0, 1, 2, 3 o 4**
- **Gli altri fanno parte del canale 2 tenuto
dal Prof. Pellegrina e dal Prof. Vandin**
 - Il loro corso è già iniziato

Me ne voglio andare!



Posso cambiare canale?

- Per ottenere il cambio di canale, lo studente deve fare richiesta **motivata** al Presidente del Consiglio di Corso di Studi attraverso la Segreteria Didattica del DEI
 - **Link nel sito del corso**
 - **Chiedere ai docenti non serve a nulla!**
 - L'attribuzione del nuovo canale va poi comunicata al docente del canale “ricevente”
 - Solitamente ha senso solo se fatta per tutti gli insegnamenti del semestre
(altrimenti gli orari non sono compatibili)

Seguire le lezioni

- Quest'anno il corso di laurea torna alla didattica esclusivamente in aula
 - **Le lezioni si possono seguire in diretta SOLTANTO in aula**
 - In questo corso è iscritto uno studente che seguirà le lezioni remotamente, perché così autorizzato
 - **Facoltativamente** (a scelta del docente), le lezioni vengono anche **videoregistrate** e messe a disposizione nel sito del corso
 - **Io ho deciso di farlo**, salvo casi eccezionali: per motivi tecnici, alcune lezioni registrate potrebbero mancare (ci sono, comunque, le registrazioni dello scorso anno...)
- Metto anche a disposizione il materiale didattico di due anni precedenti (con video)

Come fare domande

- L'esperienza degli anni scorsi mi conferma che ci sono ben poche domande durante le mie lezioni
- Vorrei evitare di includere la vostra voce nella registrazione (per eventuali problemi di "diritti"), quindi **metterò in pausa la registrazione mentre fate la domanda**, che poi io ripeterò dopo aver ripreso la registrazione
 - Chi è in aula **alza la mano** e attende un mio cenno per fare la domanda
 - Chi è collegato remotamente, **chiede in chat** di poter intervenire (o pone direttamente la domanda in chat, se è breve)
- Spesso i dubbi si chiariscono con la slide successiva...
- A fine lezione (e quando mi pare opportuno), dopo aver messo in pausa la registrazione, chiederò se ci sono domande "remote" o in aula: in tali situazioni intervenite direttamente

Come fare domande

- La maggior parte dei dubbi che sorgono a lezione sono più facilmente oggetto di **domande/risposte in posta elettronica, dopo la lezione**
- Molto spesso si tratta di quesiti di una certa complessità, ai quali difficilmente si può rispondere "al volo" in modo didatticamente efficace
 - La domanda tipica riguarda algoritmi e/o codice...
 - L'elaborazione di una risposta sensata richiede tempo, che vi dedico volentieri al di fuori delle ore di lezione
- Sembra strano, ma vedrete che è così... **la maggior parte delle interazioni, per questo corso, avviene in posta elettronica**
(con eventuali integrazioni audio/video, mie o vostre)
 - È la natura della materia di insegnamento che spinge a questo

Docente (www.dalpasso.net)

□ (Prof. Ing.) Marcello **Dalpasso**

- Professore Associato nel settore dei Sistemi per l'Elaborazione dell'Informazione

□ DEI - Dip. di Ingegneria dell'Informazione

- [Via Gradenigo 6/A](#) (Padova), edificio principale (DEI/G)

□ Ricevimento studenti (anche via ZOOM) **solo su appuntamento**

- molto più efficiente e flessibile di un orario fisso
- appuntamento va richiesto tramite posta elettronica

□ Il metodo più efficace di interazione diretta con il docente è la *posta elettronica*

marcello.dalpasso@unipd.it

Attenzione: il mio **cognome** è (stranamente) **una parola unica** (importante nei motori di ricerca)

Richiesta di colloquio

- Illustrate **bene** il problema in **posta elettronica**, eventualmente allegando algoritmo o codice, anche in PDF o con immagini/foto/video
 - Ma **se è codice, allegate IL FILE SORGENTE**
 - **Il punto precedente è MOLTO importante: se avete scritto del codice, non ha senso che mi mandiate una foto dello schermo con il codice... perché poi lo devo ricopiare!**
 - Dovete **sempre** scrivere usando il vostro indirizzo ufficiale di ateneo (...[@studenti.unipd.it\)](mailto:@studenti.unipd.it)
 - Non sempre è necessario un colloquio, spesso riesco a rispondere adeguatamente in posta elettronica, eventualmente con registrazioni audio/video
- Si può contattare il docente anche per problemi non strettamente riguardanti il corso o la didattica



Interazioni con il docente

- **Durante il corso, normalmente rispondo alla posta elettronica entro 24 ore** (eventualmente anche soltanto per dirvi "ho ricevuto la domanda e risponderò al più presto")
 - Quindi, dopo avermi scritto, NON scrivetemi di nuovo prima che siano trascorse 24 ore, pensando "forse non ha ricevuto il mio messaggio precedente..."
 - A volte i messaggi di posta elettronica non vengono consegnati correttamente senza che il mittente riceva notifiche (chi si occupa di reti di calcolatori direbbe che si tratta, tecnicamente, di un servizio "non affidabile", come il servizio SMS dei cellulari), ma DOVETE aspettare 24 ore
- Terminato il corso e la prima sessione d'esami, il tempo di risposta può aumentare, fino a "qualche giorno", al limite una settimana (come se fosse il ricevimento settimanale), situazione che tipicamente segnalerò nel sito del corso

Curriculum vitæ et studiorum

- Nato il 19 ottobre 1965 a Ferrara (vivo a Ferrara)
- Maturità scientifica
- Laurea in Ingegneria Elettronica a Bologna
- Esame di Stato per l'abilitazione alla professione di Ingegnere
- Dottorato di Ricerca in Ingegneria Elettronica e Informatica a Bologna
- Ricercatore Universitario a Padova dal 1998
- Professore Associato a Padova dal 2004

Attività didattica

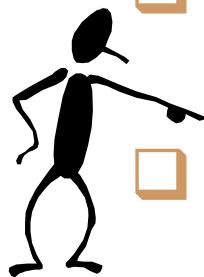
- Da molti anni inseguo Dati e Algoritmi
- Da cinque anni inseguo
Elementi di Informatica e Programmazione
- In precedenza, ho insegnato
 - Fondamenti di Informatica (fino a cinque anni fa)
 - Circuiti e Sistemi Logici
 - ora inserito in Architettura degli Elaboratori
 - Reti di Calcolatori
- Ho tradotto dall'inglese molti testi per la didattica universitaria, ricoprendo anche il ruolo di curatore dell'edizione italiana

Iscrizione in Moodle

□ Dopo aver fatto login, **bisogna iscriversi agli insegnamenti** presenti nel sistema (uno per uno...)

- Chiedere informazioni ai singoli docenti

□ **Per l'iscrizione a questo corso serve la password «X»**



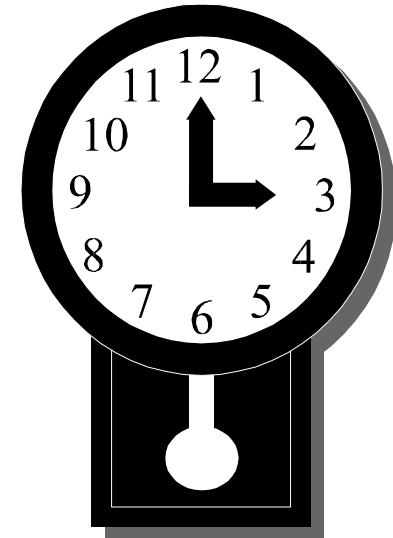
□ L'iscrizione al corso in Moodle è **NECESSARIA** per

- Accedere al materiale didattico e alle videoregistrazioni
- Accedere agli esercizi proposti
- Essere aggiornati in merito a eventuali modifiche al calendario delle lezioni o alle modalità d'esame

□ In pratica, è necessaria per (sperare di) superare l'esame ☺

Orario

- Lezioni in aula Ae (ma consultate l'orario)
 - Lunedì 10.30-12.30
 - Martedì 10.30-12.30
 - Giovedì 14.30-16.30
- Un'ora di didattica universitaria a UniPD ha una durata compresa tra 45 e 60 minuti, in relazione alle esigenze didattiche della lezione (cioè "decide il docente"...☺)
 - Decisione del Senato Accademico del 09/11/2015
- Quindi **2 ore di lezione durano 90-120 minuti**
 - **Io arriverò al minuto 30** (ad esempio, 10.30) e inizierò dopo qualche minuto (tempo tecnico di "setup"), facendo solitamente lezione SENZA pausa intermedia per **90/110 minuti**
 - Quando il tempo "sta finendo" **NON rumoreggiate** in aula, chi deve/vuole andare via, vada via in silenzio
- Il corso **NON** prevede esercitazioni in laboratorio



Calendario delle lezioni



□ Lezioni: 28 settembre 2022 – 20 gennaio 2023

- La durata del corso è di **72 ore**,
può darsi che finisca prima del 20 gennaio!

□ Vacanze di Natale

- 23 dicembre 2022 (compreso) – 8 gennaio 2023 (compreso)

□ Altre festività:

- **31 ottobre 2022, chiusura dell'ateneo (lunedì)**
- **1 novembre 2022, festa nazionale (martedì)**
- **8 dicembre 2022, festa nazionale (giovedì)**
- 9 dicembre 2022, chiusura dell'ateneo (venerdì)

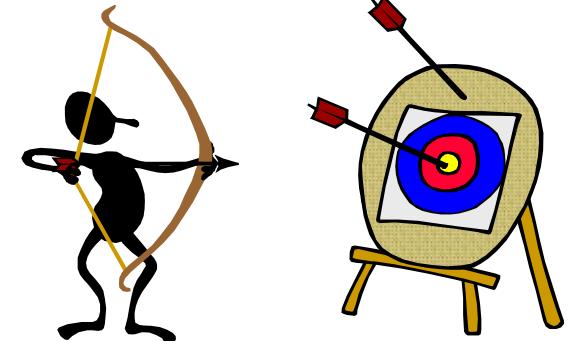
□ Può succedere che, con il massimo preavviso possibile, alcune lezioni vengano annullate (e, poi, recuperate)

- **Consultate sempre il sito e la posta elettronica!!!**

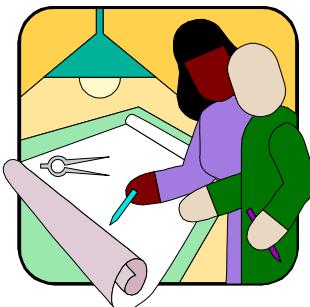
Studio "di gruppo"

- Nella preparazione degli esami di progettazione, come questo, è molto utile lavorare in gruppo (non durante l'esame ☺), in particolare per la parte di esercizi
 - Gli ingegneri dovranno certamente lavorare in "team", tanto vale abituarsi da subito
- **Non fate gli esercizi insieme!!**
 - Scegliete un esercizio, risolvetelo **in autonomia**, poi confrontate e discutete insieme le soluzioni trovate, per decidere quale sia la migliore e quali errori ci siano in altre
- Fate gruppi di piccole dimensioni (4 o 5), possibilmente "ben assortiti": quelli meno bravi trarranno vantaggio da quelli più bravi, che, a loro volta, avranno la soddisfazione di aiutare gli altri e faranno l'utilissimo esercizio di individuare gli errori

Obiettivi formativi



- Il corso ha l'obiettivo di
 - Presentare in forma sistematica le metodologie di **analisi** e di **progetto** efficiente di **algoritmi e strutture dati**
 - Delineare alcune strategie di massima per realizzare tali algoritmi e strutture dati nell'ambito della **programmazione orientata agli oggetti** (in Java)
- **Non sono previste attività di laboratorio**



Algoritmo = Tecnologia

- Spesso si dimentica che
gli algoritmi sono una tecnologia
 - Per ottenere un efficiente sistema di elaborazione dei dati, occorre utilizzare **hardware efficiente** e **algoritmi efficienti** (che usano **strutture dati opportune**, cioè efficienti per quegli algoritmi)
 - **Hardware e algoritmi sono due aspetti tecnologici**
 - Quindi, per entrambi, serve un **ingegnere!** ☺
 - La progettazione di algoritmi rientra nella più generale categoria delle strategie di ***problem solving***

Programma dettagliato

- Programmazione orientata agli oggetti in Java (richiami): classi, interfacce, ereditarietà, polimorfismo statico e dinamico, programmazione generica, documentazione del software.
Tecniche di programmazione orientata agli oggetti: adattatore, iteratore.
- Specifica di algoritmi: modello di calcolo, problema computazionale, algoritmo, strategia "divide et impera".
- Analisi di algoritmi: elementi di calcolo combinatorio e asintotico, ricorrenze.
- Alberi: definizioni, proprietà, algoritmi. Alberi binari.
- Code con priorità e heap.
- Mappe e tavole hash. Dizionari.
- Alberi di ricerca: definizioni e casi particolari.
- Algoritmi di ordinamento e selezione.
Limite inferiore al problema dell'ordinamento basato su confronti.
- Pattern matching tra stringhe.
- Grafi: definizioni, proprietà, algoritmi.



Programma sintetico

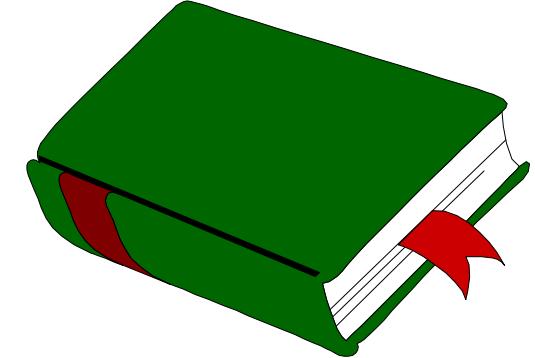
- Analisi di prestazioni
- Liste
- Alberi
- Code prioritarie
- Mappe e dizionari
- Alberi di ricerca
- Algoritmi di ordinamento
- (Elaborazione di stringhe)
- Grafi



In pratica,
è l'indice
del libro 

Libro “di testo”

Goodrich, Tamassia, Goldwasser



“*Data Structures and Algorithms in Java*”,

6th ed. Ed. John Wiley & Sons, 2014

Traduzione italiana

“*Algoritmi e strutture dati in Java*”,

Ed. Maggioli / Apogeo, 2015

Non è “obbligatorio”,

ma **NON chiedete se altri libri vanno bene...**

Attenzione: siete all’Università...

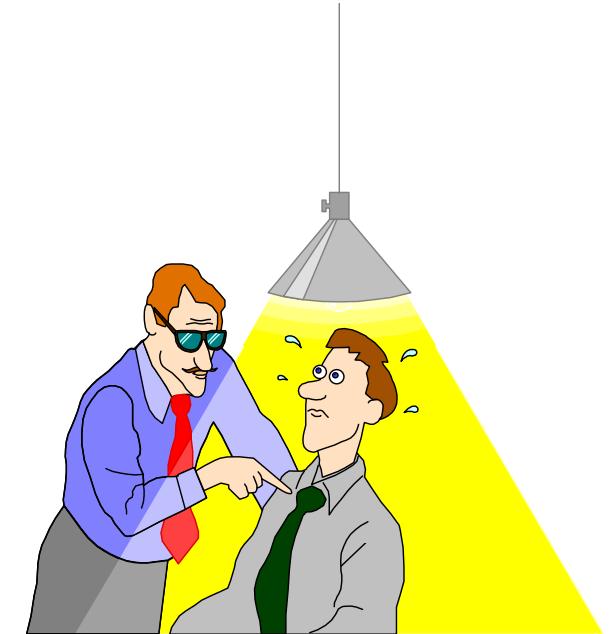
il “libro di testo” **NON** verrà seguito passo-passo e **NON** contiene tutto (e solo) il materiale presentato a lezione

Modalità d'esame

- Due appelli a gennaio/febbraio,
un appello a giugno/luglio e
un appello a settembre

Già fissati, vedere il sito del corso

- I dettagli sulle modalità d'esame si possono (si devono!) consultare sul sito
 - Una prova scritta in ogni appello
 - L'esame è soltanto scritto!!



"Compitini" ?

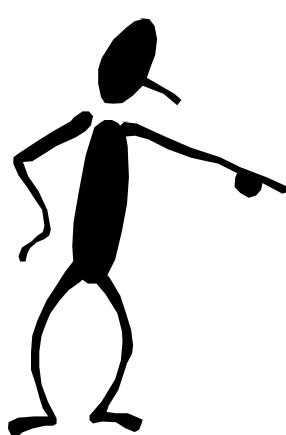
- Quest'anno, come l'anno scorso,
**NON ci sarà la settimana di pausa
didattica a metà corso**

- Quest'anno, come l'anno scorso,
**NON ci saranno i compitini per
questo corso**

Qui si progetta!

- **ATTENZIONE: questo è un corso di PROGETTAZIONE**
- Conoscere la teoria (cioè "avere studiato") è soltanto una condizione NECESSARIA per superare l'esame, non certo sufficiente
 - È (anche) per questo motivo che non c'è l'esame orale
- **Bisogna dimostrare di saper affrontare e risolvere problemi mai affrontati durante il corso (anche se ovviamente simili ad altri già visti e risolti)**

Frequenza alle lezioni

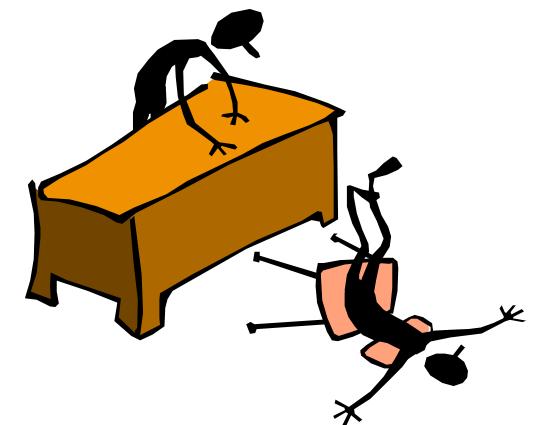


- La frequenza alle lezioni **NON** è obbligatoria però
 - Tutte le informazioni di carattere organizzativo che vengono fornite **A LEZIONE O SUL SITO** devono essere note
- La frequenza alle lezioni è comunque
 - **VIVAMENTE CONSIGLIATA**

Quanto bisogna studiare?

- Risposte banali
 - **MOLTO, ABBASTANZA, POCO...**
- Risposta da ingegnere per (aspiranti) ingegneri
 - dipende dai crediti
- Questo è un esame da **9 crediti**, quindi facciamo qualche calcolo...

Quanto bisogna studiare?



- Un credito = 25 ore di lavoro
 - totale corso di 9 crediti = $25 \times 9 = 225$ ore di lavoro
 - lezioni = 72 ore
 - studio finale prima dell'esame (ipotesi) = 33 ore
 - **studio settimanale (per 12 settimane) = 10 ore**
- Attenzione: vale per uno “studente medio”
 - **Non è possibile che siate TUTTI migliori dello studente medio... studiate statistica!!!**
- Attenzione: vale per “superare l'esame”, non per prendere 30 !!
- Per uno studente non frequentante
 - studio finale prima dell'esame = 33 ore
 - **studio settimanale (per 12 settimane) = 16 ore**

Conviene rifiutare un voto?

□ **Quanto influisce il voto V di un singolo esame sul voto di laurea F ?**

- Dipende da
 - Numero di crediti C del corso
 - Media M (pesata per crediti) degli altri esami
 - k = eventuali bonus

□ $F = k + (110/30) * [(180 - C) * M + C * V] / 180$

□ $F_{V1} - F_{V2} = (110/30) * C * (V1 - V2) / 180$

- Non dipende da M degli altri esami...
- $\Delta F = \Delta V * C * 0.02037$
- $\Delta F = \Delta V * 0.1833$ (per C = 9)

Conviene rifiutare un voto?

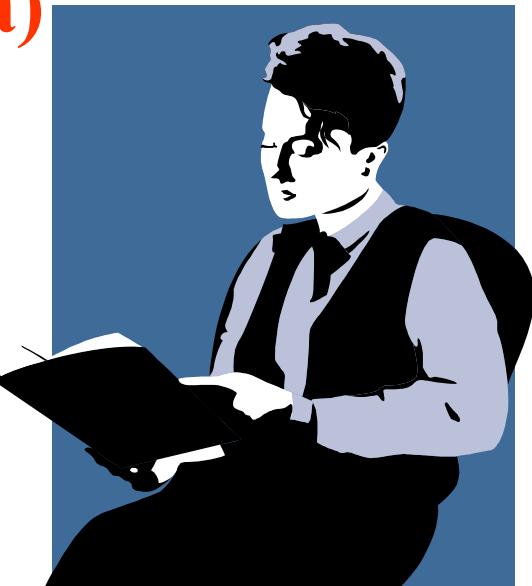
- Passando **da 18 a 30** nel voto di questo corso, il voto finale di laurea **aumenta di 2.2**
 - Ma di solito l'alternativa non è tra 18 e 30...
- Il voto finale di laurea aumenta di **0.1833** per ogni punto in più nel voto di un esame da **9 crediti**
 - Rifiutando 22 e prendendo poi 25 si aumenta il voto finale di laurea di poco più di mezzo punto
 - con gli arrotondamenti, si aumenta o di un punto o di... niente
 - Ma rifiutando 22 e prendendo poi 19...

Prerequisiti e Propedeuticità

- Nel vecchio ordinamento questo corso si chiamava **Fondamenti di Informatica 2** e, in effetti
 - costituisce la naturale prosecuzione del corso di **Fondamenti di Informatica** (che era Fondamenti di Informatica **1**)
 - è un corso “di fondamenti”,
non si occupa di ricerca avanzata
- Il corso di **Fondamenti di Informatica** **NON** è (più) **propedeutico**, cioè non è obbligatorio averlo superato per poter sostenere l'esame di Dati e Algoritmi (ma lo era...)
 - **È, però, un prerequisito per questo corso**

Prerequisito: Fondamenti di Informatica

- Oltre ad alcune nozioni di matematica e probabilità elementare, il contenuto del corso di Fondamenti di Informatica verrà dato per **acquisito all'inizio di questo corso** (faremo soltanto **pochi e mirati richiami**)
 - I **prerequisiti** del corso sono sostanzialmente **i primi 7 capitoli e l'appendice del libro (che contiene elementi di matematica)**
- **Compito a casa**
 - **Ripassare (o studiare...) i prerequisiti del corso**



Come organizzare il ripasso

- Il materiale che servirà **più urgentemente** sapere per seguire le lezioni è contenuto nei
 - **Capitoli 3, 4, 5, 6 e 7
oltre all'appendice**
- Il **ripasso di Java (capitoli 1 e 2)** può avvenire “in sottofondo” nelle prime settimane...
- I **Capitoli 4, 5, 6 e 7 verranno (rapidamente!)** ripassati in aula durante le prime lezioni
- Ho già pubblicato materiale che potrà agevolare il ripasso



Forum di aiuto reciproco nel sito

- Nel sito del corso ho aperto un forum nel quale tutti/e possono porre quesiti/problemi/dubbi, sia di carattere teorico sia in relazione ad esercizi, sperando nella risposta da parte di altre/i
 - **Usatelo soltanto per argomenti trattati nel corso**
- Ogni tanto il docente potrà intervenire per rispondere a quesiti che ritiene interessanti e che non hanno ottenuto una risposta corretta ed esauriente: questo intervento non è garantito
- Potrebbe essere un'iniziativa interessante e utile: dipende da voi
- NON verrà utilizzato in alcun modo per la valutazione ai fini dell'esame
- Il forum è sperimentale: **in caso di abusi verrà chiuso**

Modalità didattica per gli esercizi

- In questo corso, la presentazione degli esercizi risolti (essenziale per il superamento dell'esame, che è costituito prevalentemente da esercizi) avviene in questo modo
 - **I nuovi esercizi da risolvere vengono sempre presentati (o semplicemente annunciati) a lezione**, con alcuni commenti esplicativi rispetto al testo, qualche suggerimento, qualche spunto per iniziare... dipende dall'esercizio
 - A fine lezione, trovate il testo dell'esercizio nel sito del corso, **SENZA soluzione**
 - Naturalmente molto spesso potreste trovare la soluzione nel materiale dello scorso anno, ma **NON FATELO...**
 - **Dopo qualche giorno**, la soluzione dell'esercizio
 - Viene presentata durante una lezione, **oppure**
 - Viene annunciata come "pubblicata nel sito"
 - **In ogni caso, la soluzione viene pubblicata nel sito** (in PDF con eventuale video)

Esercizio
proposto per una
soluzione
autonoma

Esercizio proposto

□ Progettare un algoritmo che trovi il valore **massimo** e il valore **minimo** in un insieme di **n** numeri (reali) usando (SEMPRE) un **numero di confronti** (tra elementi dell'insieme) **minore di $3n/2$**

- Strategia didattica: a lezione pro porrò esercizi (poi pubblicati in Moodle), commentandoli ma senza risolverli, in modo che possiate provare a risolverli da soli
 - Dopo qualche giorno li risolverò in aula e/o pubblicherò la soluzione in Moodle (solo PDF o anche audio/video)
- Chi vuole seguire il corso in modo efficace **DEVE** provare a fare gli esercizi **PRIMA** di vedere la soluzione
 - In questo corso, **leggere una raccolta di esercizi risolti è sostanzialmente inutile!**

Esercizio proposto

- Progettare un algoritmo che trovi il valore **massimo** e il valore **minimo** in un insieme di n numeri (reali) usando (SEMPRE) un **numero di confronti** (tra elementi dell'insieme) **minore di $3n/2$**
 - Prima osservazione: possiamo usare un algoritmo “standard”?
 - L’algoritmo di ricerca del minimo in un insieme di n elementi richiede $n - 1$ confronti (scelgo il primo elemento come “attuale minimo”, cioè **“il minimo tra i valori ispezionati fino a questo momento”**, poi lo confronto con un altro elemento e decido quale sarà il nuovo “attuale minimo”, e via così)
 - Analogamente, la ricerca del massimo richiede $n - 1$ confronti
 - In totale, $2(n - 1) = 2n - 2 > 3n/2$ confronti ($\forall n > 4$, ma a noi interessa che rispetti i vincoli *sempre...*), non va bene, devo riuscire a risparmiare circa il 25% di confronti...
 - Osservazione: in questo esercizio, gli eventuali confronti tra “indici” usati per accedere al contenitore dell’insieme non contano

Esercizio proposto

- Progettare un algoritmo che trovi il valore **massimo** e il valore **minimo** in un insieme di **n** numeri (reali) usando (SEMPRE) un **numero di confronti** (tra elementi dell'insieme) **minore di $3n/2$**
 - Seconda osservazione: "sempre" significa...
 - Per qualsiasi valore (ammesso) di n
 - In questo caso, il minimo valore ammesso per n è 1, perché in un insieme di zero elementi non è definito né il valore massimo né il valore minimo
 - Questo significa che, per $n = 1$, dobbiamo risolvere il problema facendo un numero di confronti minore di $3/2 = 1.5$; siccome il numero di confronti è un numero intero, significa che possiamo fare, al massimo, un solo confronto (in realtà, **quando $n = 1$ bastano zero confronti...**)
 - » Questa analisi è utile perché se si riesce a **dimostrare** che anche in un solo caso NON è POSSIBILE individuare un algoritmo che rispetti i vincoli imposti, si può concludere l'esercizio
 - Per qualsiasi insieme di n valori
 - Per qualsiasi... (dipende dal problema)
 - Nei nostri esercizi, come vedrete, "sempre" è solitamente sottinteso, nel senso che una soluzione che risolva un problema "in alcuni casi" non è, normalmente, una soluzione

Esercizio proposto

- Progettare un algoritmo che trovi il valore **massimo** e il valore **minimo** in un insieme di n numeri (reali) usando (SEMPRE) un **numero di confronti** (tra elementi dell'insieme) **minore di $3n/2$**
 - Nel progettare un algoritmo risolutivo, immaginate che l'insieme di valori sia memorizzato in un array, **senza alcun ordinamento particolare tra i valori stessi**
 - È ovvio che, se l'array fosse ordinato, si risolverebbe il problema facendo zero confronti
 - **MOLTO IMPORTANTE:**
Non si possono aggiungere ipotesi al testo degli esercizi... (troppo comodo ☺)

Lezione 02

Algoritmi e prestazioni

Importanza delle prestazioni: ordinamento

Dimensione array	QuickSort $\Theta(n \log n)$	SelectionSort $\Theta(n^2)$
10^5	7 ms	5 s
10^6	84 ms	9 m
$3 \cdot 10^6$	270 ms	1 h
$5 \cdot 10^6$	470 ms	3 h
$10 \cdot 10^6$	1 s	15 h
$20 \cdot 10^6$	2 s	2 d
$50 \cdot 10^6$	5 s	15 d
$100 \cdot 10^6$	11 s	62 d
$200 \cdot 10^6$	23 s	250 d
$500 \cdot 10^6$	1 m	4 y
10^9	2 m	17 y

Problema computazionale

I calcolatori si occupano di risolvere
problemi **computazionali**
(cioè problemi "di calcolo")



Alcune di queste definizioni
non sono sul libro, c'è una
"dispensa" in PDF del
Prof. Pietracaprina
disponibile nel sito del corso

Problema computazionale

- (Ricordiamo che) Una **relazione tra due insiemi** è un sottoinsieme del loro prodotto cartesiano (es. una funzione è una relazione tra il suo dominio e il suo codominio)
- Un **problema computazionale P** è
una relazione tra un insieme I di esemplari
(o istanze) del problema stesso
e un insieme S di soluzioni

$$P \subseteq I \times S$$

con il vincolo che

$$\forall i \in I \exists s \in S | (i, s) \in P$$

Un esemplare del problema può avere più soluzioni

- Se $(i, s) \in P$, diciamo che s è una soluzione dell'esemplare i del problema P (può essere unica, ma non è detto che lo sia)

Problema computazionale: Esempio (1)

- Sia Z l'insieme dei numeri interi
- Il calcolo della somma di due numeri interi può essere formalizzato come problema computazionale in questo modo

$$I_1 = Z \times Z$$

$$S_1 = Z$$

$$P_1 = \{ ((a, b), c) \mid a, b, c \in Z \text{ e } c = a + b \}$$

$$\subseteq I_1 \times S_1 = (Z \times Z) \times Z$$

ATTENZIONE: il prodotto cartesiano non gode della proprietà associativa...

- In questo caso $P_1 \subset I_1 \times S_1$ **Perché?**

- In questo problema

$$\forall i \in I_1 \exists ! s \in S_1 \mid (i, s) \in P_1$$

Perché?

Problema computazionale: Esempio (1)

$$P_1 = \{ ((a, b), c) \mid a, b, c \in Z \text{ e } c = a + b \} \subseteq I_1 \times S_1 = (Z \times Z) \times Z$$

- In questo caso $\forall i \in I_1 \ \exists ! s \in S_1 \mid (i, s) \in P_1$ **Perché?**
- Perché il risultato di un'operazione di addizione esiste sempre ed **è unico**

Problema computazionale: Esempio (1)

$$P_1 = \{ ((a, b), c) \mid a, b, c \in \mathbb{Z} \text{ e } c = a + b \} \subseteq I_1 \times S_1 = (\mathbb{Z} \times \mathbb{Z}) \times \mathbb{Z}$$

- In questo caso $P_1 \subset I_1 \times S_1$ **Perché?**
- In generale, invece, cosa significa $P = I \times S$?
 - Significa che $\forall i \in I, \forall s \in S, (i, s) \in P$
 - Cioè: scelgo un esemplare i del problema P e scelgo una **qualsiasi** soluzione s del problema P (**l'insieme S contiene le possibili soluzioni di tutti gli esemplari del problema...**), **senza basarmi su i** , e scopro che s è una soluzione di i !!!!
 - Di solito problemi di questo tipo sono "banali" e poco interessanti dal punto di vista pratico
 - Esempio: calcolare il prodotto di un numero intero qualsiasi moltiplicato per zero (*slide successiva*)

Problema computazionale: Esempio(2)

- Esempio: calcolare il prodotto di un numero intero qualsiasi moltiplicato per zero

$$I_2 = \mathbb{Z}$$

$$S_2 = \{0\}$$

$$P_2 = \{ (a, 0) \mid a \in \mathbb{Z} \} = \mathbb{Z} \times \{0\} = I_2 \times S_2$$

- La coppia $(a, 0) \in P_2$ rappresenta il problema di calcolare a moltiplicato per zero, che ha come soluzione zero
 - lo zero presente nella coppia è la soluzione, cioè il risultato della moltiplicazione, non il secondo fattore della moltiplicazione..)
 - l'istanza è quindi $a \in I_2$, che rappresenta la moltiplicazione di a per zero
- È manifestamente un problema computazionale poco interessante ☺

Problema computazionale: Esempio (3)

- Problema: dato un numero intero n , trovare due numeri interi la cui somma sia uguale a n

$$I_3 = \mathbb{Z}$$

$$S_3 = \mathbb{Z} \times \mathbb{Z}$$

$$P_3 = \{ (a, (b, c)) \mid a, b, c \in \mathbb{Z} \text{ e } a = b + c \}$$

$$\subseteq I_3 \times S_3 = \mathbb{Z} \times (\mathbb{Z} \times \mathbb{Z})$$

- Ciascun esemplare di questo problema ammette infinite soluzioni. **Perché?**

- Ancora, $P_3 \subsetneq I_3 \times S_3$

- Confrontare con il problema P_1 : analogie e differenze...

Problema computazionale: Esempio (4)

- Ordinare in senso non decrescente una sequenza di n numeri reali
 - In una **sequenza** la posizione **relativa** degli elementi è significativa; la rappresentiamo tra parentesi tonde

$$I_4 = \{(a_1, a_2, \dots, a_n) \mid n \in \mathbb{Z}^+, a_i \in R \ \forall i \in Z_n\}, Z_n = [1, n] \cap \mathbb{Z}$$

$$S_4 = \{(b_1, b_2, \dots, b_n) \mid n \in \mathbb{Z}^+, b_i \in Z_n \ \forall i \in Z_n, \\ b_i \neq b_j \ \forall i \in Z_n, \forall j \in Z_n \setminus \{i\}\}$$

$$P_4 = \{ (i, s) \mid i \in I_4, s \in S_4, a_{b_k} \leq a_{b_{k+1}} \ \forall k \in Z_{n-1} \}$$

[ESEMPIO NELLA SLIDE SUCCESSIVA]

In pratica, S_4 contiene tutte le possibili permutazioni dei primi n numeri interi e (almeno) una di queste permutazioni consente di ordinare una determinata sequenza $i \in I_4$.

Se (e solo se) i contiene elementi duplicati, ci sono più soluzioni (il problema ha, ovviamente, sempre soluzione).

Problema computazionale: Esempio (4)

□ Esempio

□ $i = (3, 2, 5, 4)$

□ Quindi $a_1 = 3, a_2 = 2, a_3 = 5, a_4 = 4$

□ $s = (2, 1, 4, 3)$

□ Cioè $b_1 = 2, b_2 = 1, b_3 = 4, b_4 = 3$

□ Perché il **primo** elemento della sequenza ordinata che contiene gli stessi valori di i (che è, ovviamente, **2, 3, 4, 5**) è 2, che si trova in posizione **2** nella sequenza i , cioè è $a_{\textcolor{blue}{2}}$, quindi $b_1 = \textcolor{red}{2}$

□ Il **secondo** elemento della sequenza ordinata che contiene gli stessi valori di i è 3, che si trova in posizione **1** nella sequenza i , cioè è a_1 , quindi $b_2 = \textcolor{red}{1}$

□ Il **terzo** elemento della sequenza ordinata che contiene gli stessi valori di i è 4, che si trova in posizione **4** nella sequenza i , cioè è a_4 , quindi $b_3 = \textcolor{red}{4}$

□ Il **quarto** elemento della sequenza ordinata che contiene gli stessi valori di i è 5, che si trova in posizione **3** nella sequenza i , cioè è a_3 , quindi $b_4 = \textcolor{blue}{3}$

Problema computazionale: Esempio (5)

- Problema: data una stringa di cifre binarie, trovare la lunghezza della più lunga sottostringa composta da sole cifre 1

$I_5 = \text{insieme di tutte le possibili stringhe di cifre binarie}$

$S_5 = \mathbb{Z}^+ \cup \{0\}$ [lunghezza 0 se non ci sono cifre 1]

$P_5 = \{ (i, s) \mid i \in I_5, s = \text{uni}(i) \}$

dove la funzione $\text{uni}(i)$ calcola la lunghezza della più lunga sottostringa di i composta da sole cifre 1

- Evidentemente $\forall i \in I_5 \exists ! s \mid (i, s) \in P_5$

- Se il problema fosse stato "trovare la **posizione in cui inizia** la più lunga sottostringa composta da sole cifre 1", in generale la soluzione non sarebbe unica (perché?)

Problema computazionale

- L'informatica teorica ci dice che
 - **Se un problema è risolubile al calcolatore, allora è descrivibile sotto forma di problema computazionale**
 - Il fatto che un problema sia descrivibile come problema computazionale è anche condizione sufficiente perché il problema sia risolubile al calcolatore?
 - Vedremo che la risposta è no...
 - Entra in gioco il concetto di algoritmo!
 - È una condizione soltanto **necessaria**

Algoritmo

- Un **algoritmo** A che risolve un problema computazionale $P \subseteq I \times S$ è una **procedura** che, dato un esemplare $i \in I$, individua tramite una **sequenza finita e non ambigua di passi computazionali elementari** (almeno) una soluzione $s \in S$ tale che $(i, s) \in P$
 - Si dice anche che A *trasforma* l'ingresso i nell'uscita s
- Bisogna definire cosa si intende per "passi computazionali elementari"
 - Di solito lo si fa in relazione a un **modello computazionale**, cioè un'astrazione di un dispositivo di calcolo, caratterizzato da un ben determinato insieme di "istruzioni" o operazioni eseguibili
 - Un modello computazionale molto utilizzato è il calcolatore **RAM** (Random Access Machine), in pratica una semplice CPU con memoria ad accesso casuale

Modello computazionale RAM



- Il modello RAM (**Random Access Machine**) ha un insieme di istruzioni elementari che comprende:
 - Operazioni logiche e aritmetiche *elementari*
 - Trasferimento di dati tra CPU e memoria nelle due possibili direzioni
 - "scrittura" in memoria (assegnamento di un valore a una variabile)
 - "lettura" dalla memoria (utilizzo di una variabile in una espressione)
 - Indicizzazione in array in un tempo indipendente dall'indice
 - **Cioè "accesso casuale" agli elementi contenuti in un array**
 - Esecuzione condizionale per effetto di confronti tra dati
 - Quindi anche iterazioni
 - Invocazione di un metodo con eventuale passaggio di parametri ed eventuale restituzione di un valore (rendendo così possibile la ricorsione)

Lezione 03

Esercizio 01

Massimo e minimo

- Progettare un algoritmo che trovi il valore **massimo** e il valore **minimo** in un insieme di n numeri (reali) usando (SEMPRE) un **numero di confronti** (tra elementi dell'insieme) **minore di $3n/2$**
 - Prima osservazione: possiamo usare un algoritmo “standard”?
 - L’algoritmo di ricerca del minimo in un insieme di n elementi richiede $n - 1$ confronti (scelgo il primo elemento come “candidato minimo”, poi lo confronto con un altro elemento e decido quale sarà il nuovo “candidato minimo”, e via così)
 - Analogamente, la ricerca del massimo richiede $n - 1$ confronti
 - In totale, $2(n - 1) = 2n - 2 \geq 3n/2$ confronti ($\forall n \geq 4$, ma a noi interessa n grande e, in ogni caso, qualsiasi valore di $n\dots$), non va bene, devo riuscire a risparmiare circa il 25% di confronti...

Massimo e minimo

```
// insieme memorizzato in array
public static double[] findMinMax(double[] a)
{  if (a == null || a.length == 0)
   throw new IllegalArgumentException();
double min = a[0];
for (int i = 1; i < a.length; i++) // cerco minimo
  if (a[i] < min)
    min = a[i];
double max = a[0];
for (int i = 1; i < a.length; i++) // cerco massimo
  if (a[i] > max)
    max = a[i];
// ricordare:
// un modo semplice per restituire due valori...
return new double[] {min, max};
}
```

- Approccio "banale": richiede troppi confronti ($2n - 2$)

Digressione Java

```
// un modo semplice per restituire due valori...
return new double[] {min, max};
```

- Equivale a:

```
double[] d = new double[] {min, max};
return d;
```

- Quando una variabile viene dichiarata e inizializzata, per poi essere utilizzata (cioè "letta") in un solo punto del codice, si può SEMPRE evitare di definirla, copiando in quel punto il codice che la inizializzerebbe
 - Eccezione: si può usare comunque una variabile temporanea quando l'espressione risultante sarebbe troppo complessa o lunga
 - **Scrivere codice facilmente leggibile è sempre preferibile**

Digressione Java

```
double[] d = new double[] {min, max};
```

- Ricordiamo che questa sintassi costruisce un array e lo inizializza ordinatamente con i valori indicati; è equivalente a questa:

```
double[] d = new double[2];  
d[0] = min;  
d[1] = max;
```

- La **dimensione** dell'array viene calcolata dal compilatore sulla base del numero di valori inseriti tra le parentesi graffe (viene sempre creato un array "tutto pieno" e non "riempito solo nella sua parte iniziale")
- Si potrebbe scrivere anche così, in una forma ancora più implicita, ma personalmente preferisco la forma più esplicita

```
double[] d = {min, max};
```

Osserviamo che l'esercizio
non pone vincoli alle
strutture dati utilizzabili

Massimo e minimo

- Progettare un algoritmo che trovi il valore massimo e il valore minimo in un insieme di n numeri usando un numero di confronti minore di $3n/2$
 - Non possiamo calcolare separatamente il minimo e il massimo, perché questo richiede troppi confronti, facendo **due** scansioni dell'insieme
 - Proviamo a fare **un'unica scansione dell'insieme** tenendo traccia contemporaneamente del “candidato minimo” e del “candidato massimo”, e aggiornandoli opportunamente
 - Prendo un elemento dell'insieme (tipicamente il primo), che diventa sia il “candidato minimo” sia il “candidato massimo”
 - Per ognuno dei rimanenti $n - 1$ elementi dell'insieme...

Massimo e minimo

```
public static double[] findMinMax(double[] a)
{ if (a == null || a.length == 0)
    throw new IllegalArgumentException();
double min = a[0];
double max = a[0];
for (int i = 1; i < a.length; i++) // n-1 iterazioni
{ if (a[i] < min)
    min = a[i];
if (a[i] > max)
    max = a[i];
} // 2 confronti per ogni iterazione
return new double[] {min, max};
}
```

- Ora fa una scansione unica dell'array, ma il numero di confronti non è diminuito, è ancora **$2(n - 1)$**
- Risparmia **confronti tra indici**, ma non ci interessa

Massimo e minimo

□ Un'altra idea...

- Prendo un elemento dell'insieme, che diventa sia il “candidato minimo” sia il “candidato massimo”
- Per ognuno dei rimanenti $n - 1$ elementi dell'insieme
 - Se è minore dell'attuale “candidato minimo”, diventa il nuovo “candidato minimo”
 - **Altrimenti**, se è maggiore dell'attuale “candidato massimo”, diventa il nuovo “candidato massimo”
 - Perché "altrimenti" ?
 - L'elemento in esame non può essere sia minore dell'attuale minimo, sia maggiore dell'attuale massimo! Perché, ovviamente, l'attuale minimo è sempre minore (o al massimo uguale) all'attuale massimo.

```
if (a[i] < min)
    min = a[i];
if (a[i] > max)
    max = a[i];
```

Massimo e minimo

```
public static double[] findMinMax(double[] a)
{ if (a == null || a.length == 0)
    throw new IllegalArgumentException();
double min = a[0];
double max = a[0];
for (int i = 1; i < a.length; i++)
{ if (a[i] < min)
    min = a[i];
else if (a[i] > max)
    max = a[i];
}
return new double[] {min, max};
}
```

```
if (a[i] < min)
    min = a[i];
if (a[i] > max)
    max = a[i];
```

- **Nel caso pessimo** (commento nel seguito), servono 2 confronti per ognuno dei rimanenti $n - 1$ elementi dell'insieme, cioè $2(n - 1)$ confronti, come prima... (prima era "sempre", adesso solo nel caso pessimo)

Massimo e minimo

- Attenzione al caso pessimo dell’algoritmo precedente
 - Per ognuno dei rimanenti $n - 1$ elementi dell’insieme
 - Se è minore dell’attuale “candidato minimo”, diventa il nuovo “candidato minimo”
 - Altrimenti, se è maggiore dell’attuale “candidato massimo”, diventa il nuovo “candidato massimo”
 - Nel caso pessimo, per un elemento possono servire 2 confronti (quando l’elemento è maggiore dell’attuale “candidato minimo”)
 - Possiamo dire (come abbiamo fatto) che il caso pessimo che si può verificare per un elemento, si può verificare anche per tutti gli elementi?
 - Attenzione perché in generale non è vero!
In questo caso sì! Infatti...

Massimo e minimo

- Attenzione al caso pessimo dell’algoritmo precedente
 - Per ognuno dei rimanenti $n - 1$ elementi dell’insieme
 - ...
 - Nel caso pessimo, per un elemento possono servire 2 confronti (quando l’elemento è maggiore dell’attuale “candidato minimo”)
 - Possiamo dire (come abbiamo fatto) che il caso pessimo che si può verificare per un elemento, si può verificare anche per tutti gli elementi?
 - **Sì, questa situazione si verifica se (e solo se) il “candidato minimo” iniziale è effettivamente il minimo dell’intero insieme**
 - TUTTI i successivi confronti con `min` saranno falsi, quindi verranno eseguiti TUTTI i confronti con `max`
 - **BISOGNA SEMPRE RAGIONARE...**

Massimo e minimo

- Come possiamo risparmiare confronti?
 - Bisogna che ciascun confronto eseguito ci fornisca una "quantità di informazione" maggiore
- Osservazione dalle situazioni precedenti
 - Ogni volta che faccio un confronto tra due elementi, individuo un elemento (il minore dei due) che potrebbe essere il minimo e un elemento (il maggiore dei due) che potrebbe essere il massimo: certamente non può essere vero il contrario
 - Per qualsiasi coppia di elementi, x e y , se $x < y$, allora x potrebbe essere il minimo e y potrebbe essere il massimo, ma certamente x non è il massimo e y non è il minimo

Massimo e minimo

- Per semplicità, **supponiamo n pari** (ma dobbiamo poi...)
 - **Senza fare alcun confronto**, disponiamo gli elementi in $n/2$ coppie
 - All'interno di ogni coppia, facciamo **un confronto** tra il primo e il secondo elemento: se il primo è minore del secondo, allora il primo viene inserito nell'**insieme dei “candidati a essere minimo”** e l'altro viene inserito nell'**insieme dei “candidati a essere massimo”**, altrimenti viceversa; questo richiede $n/2$ confronti e ciascuno dei due insiemi, alla fine, avrà $n/2$ elementi [osservare che l'algoritmo è descritto bene anche nel caso in cui i due elementi di una coppia siano uguali, eventualità non esclusa dal testo del problema]

Osserviamo che l'esercizio non pone vincoli alle strutture dati utilizzabili,
quindi **possiamo usare insiemi aggiuntivi**

Massimo e minimo

- Progettare un algoritmo che trovi il valore massimo e il valore minimo in un insieme di n numeri usando un numero di confronti minore di $3n/2$
 - Facendo **$n/2$** confronti, costruiamo l'insieme dei candidati a essere minimo e l'insieme dei candidati a essere massimo
 - Nell'insieme dei “candidati a essere minimo”, che ha cardinalità $n/2$, cerchiamo il minimo (mediante semplice scansione, con l'algoritmo "banale"), facendo quindi **$n/2 - 1$** confronti
 - Nell'insieme dei “candidati a essere massimo”, che ha cardinalità $n/2$, cerchiamo il massimo (mediante semplice scansione, con l'algoritmo "banale"), facendo quindi **$n/2 - 1$** confronti
 - In totale, si fanno **$n/2 + 2(n/2 - 1)$** = $3n/2 - 2 < 3n/2$ confronti
 - Osservo che il limite sarebbe accettabile anche se fosse $3n/2 - 1$

Caso limite: $n = 2$

- Verifichiamo con cura che l'algoritmo funzioni anche nel caso limite $n = 2$, che è il più piccolo valore di n pari e positivo
- Progettare un algoritmo che trovi il valore massimo e il valore minimo in un insieme di $n = 2$ numeri usando un numero di confronti minore di $3n/2 = 3$
 - Facendo **$n/2 = 1$** confronti, costruiamo l'insieme dei candidati a essere minimo e l'insieme dei candidati a essere massimo
 - Nell'insieme dei “candidati a essere minimo”, che ha cardinalità $n/2 = 1$, cerchiamo il minimo (con l'algoritmo "banale"), facendo quindi **$n/2 - 1 = 0$** confronti
 - In un insieme di cardinalità 1, l'algoritmo banale non fa confronti
 - Nell'insieme dei “candidati a essere massimo”, che ha cardinalità $n/2 = 1$, cerchiamo il massimo (con l'algoritmo "banale"), facendo quindi **$n/2 - 1 = 0$** confronti
 - In totale si fa un solo confronto e ovviamente $1 < 3$: OK
 - Anche in questo caso il limite potrebbe essere $3n/2 - 1 (= 2)$

Massimo e minimo

- Si potrebbe pensare di ripetere "il trucco", sperando di risparmiare ulteriori confronti [ipotizziamo che n sia un multiplo di 4], anche se non richiesto per risolvere questo esercizio
- Facendo $n/2$ confronti, abbiamo ottenuto
 - Un insieme MIN di $n/2$ candidati minimi
 - Un insieme MAX di $n/2$ candidati massimi
- Prendo gli elementi di MIN e compongo $n/4$ coppie di numeri (senza fare confronti), poi faccio un confronto in ogni coppia (cioè $n/4$ confronti) e individuo
 - Un insieme $MINMIN$ di $n/4$ candidati minimi
 - Un insieme $MINMAX$ di $n/4$ elementi che non vengono più esaminati
- Analogamente, operando su MAX , con $n/4$ confronti individuo
 - Un insieme $MAXMIN$ di $n/4$ elementi che non vengono più esaminati
 - Un insieme $MAXMAX$ di $n/4$ candidati massimi

Massimo e minimo

- Con $n/2 + 2(n/4)$ confronti, ho ottenuto
 - Un insieme *MINMIN* di $n/4$ candidati minimi
 - Un insieme *MINMAX* di $n/4$ elementi che non vengono più esaminati
 - Un insieme *MAXMIN* di $n/4$ elementi che non vengono più esaminati
 - Un insieme *MAXMAX* di $n/4$ candidati massimi
- A questo punto, con $n/4 - 1$ confronti individuo il minimo assoluto (all'interno di *MINMIN*) e con $n/4 - 1$ confronti individuo il massimo assoluto (all'interno di *MAXMAX*), con gli algoritmi "banali"
- In totale, ho fatto $n/2 + 2(n/4) + 2(n/4 - 1) = 3n/2 - 2$ confronti, esattamente come nel caso precedente
 - Dividere ulteriormente non serve a nulla
- **Progettiamo l'algoritmo per n dispari**

□ Progettare un algoritmo che trovi il valore massimo e il valore minimo in un insieme di n numeri usando un numero di confronti minore di $3n/2$, nell'ipotesi che n sia **dispari**

- n dispari e positivo $\Rightarrow \exists k = (n - 1)/2 \in \mathbb{Z} \mid n = 2k + 1$
- Disponiamo $2k = n - 1$ elementi in k coppie (senza fare alcun confronto); sia x l'elemento non inserito in alcuna coppia
- I confronti richiesti sono k , uno per ciascuna coppia, e ciascuno dei due insiemi ("candidati minimi" e "candidati massimi"), alla fine, avrà k elementi
- **Inseriamo x in ciascuno dei due insiemi,**
che avranno così $k + 1$ elementi
 - Senza fare confronti; x è sia candidato minimo sia candidato massimo
- Nell'insieme dei "candidati a essere minimo", che ha cardinalità $k + 1$, cerchiamo il minimo, facendo k confronti (con l'algoritmo "banale")
- Nell'insieme dei "candidati a essere massimo", che ha cardinalità $k + 1$, cerchiamo il massimo, facendo k confronti (con l'algoritmo "banale")
- In totale, si fanno $3k = 3(n - 1)/2 = 3n/2 - 3/2 < 3n/2$ confronti
- Anche in questo caso il limite potrebbe essere $3n/2 - 1$

Caso limite: $n = 1$

Attenzione: nelle verifiche dei casi limite,
bisogna usare ESATTAMENTE le stesse
frasi usate nella descrizione generale

- Verifichiamo con cura che l'algoritmo funzioni anche nel caso limite $n = 1$, che è il più piccolo valore di n dispari
 - $n = 1 \Rightarrow k = (n - 1)/2 = 0$
 - Disponiamo $0 = 2k = n - 1$ elementi in $k = 0$ coppie (senza fare alcun confronto); sia x l'elemento non inserito in alcuna coppia
 - Essendoci 0 coppie, non dobbiamo fare alcun confronto per creare i due insiemi di “candidati a essere minimo” e “candidati a essere massimo”
 - Inseriamo x in ciascuno dei due insiemi (vuoti), che avranno così $k + 1 = 1$ elementi
 - Nell'insieme dei “candidati a essere minimo”, che ha cardinalità $k + 1 = 1$, cerchiamo il minimo, facendo $k = 0$ confronti (con l'algoritmo "banale")
 - Nell'insieme dei “candidati a essere massimo”, che ha cardinalità $k + 1 = 1$, cerchiamo il massimo, facendo $k = 0$ confronti (con l'algoritmo "banale")
 - In totale, si fanno $0 < 3n/2 = 1.5$ confronti
 - Anche in questo caso il limite potrebbe essere $3n/2 - 1$ ($= 0.5$)

Massimo e minimo

- Progettare un algoritmo che trovi il valore **massimo** e il valore **minimo** in un insieme di n numeri usando (SEMPRE) un numero di confronti minore di $3n/2$
- Abbiamo risolto il problema e abbiamo anche scoperto che il limite poteva essere posto a $3n/2 - 1$
- Dopo aver capito come si possa risolvere il problema, si può anche osservare che:
 - non serve "disporre" effettivamente gli elementi a coppie: è sufficiente effettuare una scansione dell'array **considerando gli elementi a coppie** (ad esempio, una coppia può essere costituita da due elementi consecutivi, oppure da due elementi equidistanti dai due estremi dell'array oppure... come vogliamo)

```
public static double[] findMinMax(double[] a)
{  if (a == null || a.length == 0)
   throw new IllegalArgumentException();
 // dimensioni corrette per n pari o dispari
double[] mins = new double[(a.length+1)/2];
double[] maxs = new double[(a.length+1)/2];
for (int i = 0; i < a.length - 1; i += 2)
  if (a[i] < a[i+1])
  {  mins[i/2] = a[i];
     maxs[i/2] = a[i+1];
  } else
  {  mins[i/2] = a[i+1];
     maxs[i/2] = a[i];
  }
if (a.length % 2 == 1) // se lunghezza dispari...
  mins[mins.length-1]
    = maxs[maxs.length-1] = a[a.length-1];
double min = mins[0];
double max = maxs[0];
for (int i = 1; i < mins.length; i++)
{  if (mins[i] < min) min = mins[i];
   if (maxs[i] > max) max = maxs[i];
}
return new double[] {min, max}; }
```

Massimo e minimo

- Dopo aver capito come si possa risolvere il problema, si può anche osservare che:
 - **non serve inserire gli elementi effettivamente in due ulteriori insiemi** di "candidati massimi" e "candidati minimi": è sufficiente avere due variabili che memorizzino il "miglior candidato minimo" e il "miglior candidato massimo" osservato fino a un dato passo dell'algoritmo
 - In pratica, la fase finale di ricerca del massimo/minimo viene fusa con quella di suddivisione...
- Scriviamo questa versione dell'algoritmo, che **opera "sul posto"**, cioè non copia gli elementi in un altro contenitore, e **non modifica il contenitore originario**, due proprietà spesso desiderabili (se non addirittura richieste)

```
public static double[] findMinMax(double[] a)
{  if (a == null || a.length == 0)
   throw new IllegalArgumentException();
double min = a[0];
double max = a[0];
// esamina elementi consecutivi a coppie disgiunte
// ma attenzione: il primo elemento è già considerato
for (int i = 1; i < a.length - 1; i += 2)
  if (a[i] < a[i+1])
    {  if (a[i] < min) min = a[i];
       if (a[i+1] > max) max = a[i+1];
    } else
    {  if (a[i+1] < min) min = a[i+1];
       if (a[i] > max) max = a[i];
    }
  if (a.length % 2 == 0) // esamina ultimo elemento
  {  if (a[a.length-1] < min) min = a[a.length-1];
     else if (a[a.length-1] > max) max= a[a.length-1];
  }
return new double[] {min, max};
}
// con array di lunghezza PARI fa un confronto in più
// rispetto alla versione con gli insiemi esplicativi,
// però rimane entro il limite imposto
```

Lezione 04

Pseudocodice

- L'insieme di istruzioni del modello RAM è ciò che solitamente usiamo nello "**pseudocodice**"
 - **Un linguaggio di programmazione destinato alla lettura "umana", per descrivere algoritmi prima della codifica in un linguaggio di programmazione vero e proprio**
 - Come tutti i linguaggi "naturali" usati dall'uomo, **non** ha una sintassi rigida
 - Diversamente dai linguaggi di programmazione, che sono veri e propri "codici", cioè hanno una sintassi rigida e sono linguaggi artificiali
 - Per questo si chiama "pseudo"codice... perché non è proprio un codice, ma quasi...

Alcune regole per lo pseudocodice (1)

- Per descrivere blocchi di codice, usare solo il rientro verso destra ("indentazione") e non le parentesi graffe o altro
 - Non è un errore usare le parentesi graffe (**non altre**)... però è indispensabile usare ANCHE l'indentazione
- Specificare il tipo delle variabili **solo quando è rilevante**
- Se non indispensabile, **non usare variabili globali**
 - In questo corso, l'uso di variabili globali o "di classe" (**static**, in Java) è **proibito**, a meno che non sia esplicitamente consentito in uno specifico esercizio, perché, in generale, rende il codice molto meno modulare e riutilizzabile
 - Sono ovviamente ammesse **costanti globali** (cioè **static final**)
- Per l'assegnazione, scegliere un simbolo e usare sempre quello
 - Esempi = := <- ←
- Nel confronto per uguaglianza scegliere un simbolo e usare sempre quello, indifferentemente = oppure ==
 - Non si può usare = se lo si usa per l'assegnazione

Alcune regole per lo pseudocodice (2)

- Come "operatore indice" negli array usare parentesi **quadre**
 - La lunghezza dell'array A può essere soltanto ispezionata (e non modificata), con la sintassi A.length (come in Java) oppure size(A) o length(A) o altre notazioni (come |A|)
- Per i cicli, usare una sintassi simile a quella di Java/C/C++
 - Per scandire array e liste, si può usare il costrutto **for each** (o **foreach**), con una sintassi simile al "for generico" in Java
- Racchiudere i parametri di metodi/funzioni tra parentesi **tonde**
 - Per restituire un valore, usare la parola chiave **return**
- Se si usano oggetti, si accede ai loro metodi e campi usando l'operatore "punto" come in Java, oppure l'operatore ->
 - Per creare un oggetto, usare l'operatore **new** come in Java
- Commenti con // o altro (es. #), tanto si capisce...

**Tipo di dato definito
perché è importante**

Un esempio di algoritmo descritto con pseudocodice

levelOrderTraversal(Tree T): // algoritmo che studieremo...

if T.isEmpty() return // inutile andare a capo con return...

q = new Queue()

↑
q.enqueue(T.root())

while !q.isEmpty() // oppure, forse meglio: while **not** q.isEmpty()

v = q.dequeue()

visit(v)

for each c ∈ T.children(v) // *children* restituisce una lista

 q.enqueue(c) // di qualche tipo... un array o altro

 // il ciclo indica una scansione completa in ordine indifferente

**Tipo di dato non definito,
sarà Queue (o qualcosa di simile...)**

Algoritmo

- Si dice anche che un **algoritmo A risolve un problema $P \subseteq I \times S$** se e solo se A calcola una funzione f_A tale che
$$\forall i \in I, (i, f_A(i)) \in P$$
 - Era già evidente dalla definizione, **ma lo ribadiamo**
 - **N.B. Non importa** cosa calcola $f_A(i)$ quando $i \notin I$
 - **Violazione delle pre-condizioni del problema...**
 - **Quando possibile**, un algoritmo verifica il rispetto delle proprie pre-condizioni (cioè verifica se $i \in I$), ma **a volte questo è troppo oneroso** per le prestazioni
 - Es. l'algoritmo di ricerca binaria non può verificare se l'array su cui opera è ordinato [perché? vedremo...]
 - In generale, quando $i \notin I$, non importa come agisce l'algoritmo, ma, se possibile, è meglio che rilevi l'errore

Algoritmo

- Molto importante! **Possono esistere diversi algoritmi che risolvono uno stesso problema**, eventualmente con prestazioni differenti in relazione a diverse metriche: tempo impiegato, spazio occupato in memoria, energia consumata...
- Viceversa, ma meno importante: **un algoritmo può risolvere più problemi diversi** (come esempio banale, due problemi, uno dei quali sia sottoinsieme dell'altro: l'algoritmo che trova il valore massimo in un insieme di numeri reali risolve anche il problema di trovare il valore massimo in un insieme di numeri interi)
- È proprio **questo** che fa assumere importanza al calcolo e alla **stima delle prestazioni degli algoritmi, per poter scegliere l'algoritmo migliore per la soluzione di un problema**
- **Ce ne occuperemo spesso**

Algoritmi e strutture dati

- Spesso gli algoritmi utilizzano **strutture dati**
 - Una **struttura dati** (o, meglio, *struttura per la memorizzazione e l'accesso ai dati*), come sapete, è definita a **due diversi livelli di astrazione**
 - **Logico/Funzionale**
ad esempio, in Java, una **interfaccia**, cioè un ADT (*abstract data type*, tipo di dato astratto)
 - **Fisico/Concreto**
ad esempio, in Java, una **classe**
 - Non è un caso che questo corso si chiami **(strutture) dati e algoritmi...**

Problema computazionale

- Quindi, l'informatica teorica ci dice che
 - Se un problema è risolubile al calcolatore, allora è descrivibile sotto forma di problema computazionale
- Viceversa, un problema computazionale è risolubile al calcolatore **se si riesce a individuare un algoritmo che lo risolva!**
- In pratica
 - **Un problema è risolubile al calcolatore se e solo se esiste (almeno) un algoritmo che lo risolve**

Correttezza di un algoritmo

Correttezza di un algoritmo

- Nel corso di Fondamenti di Informatica (solitamente) non ci si occupa di dimostrare la correttezza degli algoritmi
 - Individuato un algoritmo candidato a risolvere un problema, lo si verifica a mano "in alcuni casi", poi lo si implementa in un programma e si collauda il programma "in alcuni casi"...
 - L'insieme I che caratterizza molti problemi computazionali ha solitamente una cardinalità molto elevata
 - Inevitabilmente, la **copertura** del collaudo (cioè il sottoinsieme di I usato nel collaudo) è abbastanza bassa
- Questo, ovviamente, non dà alcuna garanzia sulla effettiva correttezza dell'algoritmo, quando questo viene utilizzato per risolvere l'intero problema computazionale per il quale è stato progettato, cioè quando viene applicato a un elemento qualsiasi di I

Correttezza di un algoritmo

- Dato un problema computazionale P e un algoritmo A candidato a risolverlo, spesso non è semplice **dimostrare** che A risolva effettivamente P
 - Occorre, cioè, **dimostrare** che l'algoritmo A sia **corretto**
- In generale, le dimostrazioni di correttezza usano diverse strategie, del tutto analoghe a quelle utilizzate in matematica, tra le quali, principalmente, combinazioni di:
 - Un **esempio** (o controesempio)
 - Una **deduzione per assurdo**, che cada in contraddizione
 - Una **induzione**
 - Spesso usata per algoritmi ricorsivi
 - Una **condizione invariante in un ciclo**
 - Spesso usata per algoritmi iterativi
 - ...

Lezione 05

Dimostrazione con un esempio

- La dimostrazione mediante un esempio si può utilizzare quando la tesi (cioè l'affermazione che va dimostrata) contiene **il quantificatore "uno/una"**
 - Es. Dimostrare che \exists (**almeno**) **un** elemento $x \in X$ che ha la proprietà Q ["almeno" è spesso sottinteso]
- In questo caso è sufficiente trovare un elemento dell'insieme X che goda della proprietà Q (cioè *un esempio* di Q) e, così, si dimostra la tesi
 - Difficilmente questo è sufficiente per dimostrare la correttezza di un algoritmo, ma può costituire una porzione della dimostrazione
 - **Ben diverso** è dimostrare che $\exists ! x \in X$ che gode di Q

Dimostrazione con un controesempio

- La dimostrazione mediante controesempio si può utilizzare quando la tesi contiene
la negazione di un quantificatore "per ogni"
- $\forall i \in Z, i > 2 \Rightarrow 2^i - 1$ è un numero primo.
Dimostrare che questa affermazione è **falsa** In alcuni casi è vera, ad esempio per $i = 3$ o 5
- Dato che l'affermazione contiene un quantificatore \forall , per dimostrarne la **falsità** è sufficiente individuare **un esempio che NON soddisfa** la proprietà (cioè un **controesempio**)
 - $i = 4$ è un controesempio, perché
 $4 \in Z, 4 > 2, 2^4 - 1 = 15 = 5 \cdot 3$, quindi $2^4 - 1$ non è primo
 - Difficilmente questo è sufficiente per dimostrare la correttezza di un algoritmo, ma può costituire una porzione della dimostrazione

Dimostrazione per assurdo

- Nella dimostrazione per assurdo si ipotizza "per assurdo" che la tesi sia falsa e si cerca di dedurre una contraddizione con le ipotesi, dimostrando così che non esiste una situazione coerente con il fatto che la tesi sia falsa e, quindi, la tesi è vera
- Nel "costruire" la negazione della tesi bisogna fare molta attenzione (di solito, è il punto più critico della dimostrazione)
 - Se la tesi contiene le congiunzioni "e" e/o "o" (cioè operatori *and* e *or* nell'algebra delle proposizioni), spesso si utilizzano le **Leggi di De Morgan** (fare attenzione!)

$$\text{not } (\text{A and B}) = (\text{not A}) \text{ or } (\text{not B})$$

$$\text{not } (\text{A or B}) = (\text{not A}) \text{ and } (\text{not B})$$

Dimostrazione per assurdo

- Nel "costruire" l'ipotesi "per assurdo", occorre **negare la tesi**
- Siano a e b numeri interi positivi. Dimostrare che, se il prodotto ab è dispari, allora ("tesi") è **vero** che a è dispari **e** b è dispari.
 - In italiano discorsivo, forse avrei detto "allora a e b sono dispari"
- *Per assurdo*, ipotizziamo che, con **ab dispari**,
sia **falso** che a è dispari **e** b è dispari [**not((a dispari) and (b dispari))**]
(cioè, per De Morgan, sia **vero** che a è pari **oppure** b è pari).

Senza perdere generalità, vista la simmetria del problema,
supponiamo a pari, per cui esiste un intero i tale che $a = 2i$.

Quindi $ab = (2i)b = 2(ib)$ \Rightarrow **ab è pari**, essendo uguale al doppio di un
numero intero (cioè al doppio del numero intero ib).

Ma il numero ab non può essere contemporaneamente pari e dispari
(ed è dispari per ipotesi), quindi: **supponendo che la tesi sia falsa,**
siamo caduti in contraddizione, per cui la tesi è vera.

Dimostrazione per induzione

- Il Principio di Induzione Matematica (PIM) è spesso utile per dimostrare la correttezza di un algoritmo **ricorsivo**
 - Non è un teorema... è un principio (o assioma/postulato) !
 - Non va dimostrato, è (intuitivamente) vero
- Sia S_1, S_2, S_3, \dots una sequenza di enunciati, messi in corrispondenza con l'insieme dei numeri interi positivi

L'enunciato S_n è vero $\forall n \in \mathbb{Z}^+$

se valgono le due seguenti proprietà

1) L'enunciato S_1 è vero

(il cosiddetto "caso base")

**2) $\forall n \in \mathbb{Z}^+,$ il fatto che l'enunciato S_n sia vero,
implica che l'enunciato S_{n+1} è vero**

(il cosiddetto "caso induttivo")

Attenzione: non dice "**se e solo se**"...

Il fatto di **NON** riuscire a fare una dimostrazione per induzione, **NON** implica che la sequenza di enunciati sia falsa!

Esempio di induzione

- Usando il Principio di Induzione Matematica, dimostrare che $\forall n \in \mathbb{Z}^+, \sum_{i=1}^n i = n(n + 1)/2$ (sommatoria di Gauss)
- $\forall n \in \mathbb{Z}^+$, l'enunciato S_n è, quindi: $\sum_{i=1}^n i = n(n + 1)/2$
- Dimostrazione del caso base: L'enunciato S_1 è vero
 - Infatti, per $n = 1$: $\sum_{i=1}^1 i = 1 = 1(1+1)/2$
- Dimostrazione del caso induttivo $\forall n \in \mathbb{Z}^+$

Supponendo che **l'enunciato S_n sia vero**, si riesce a dimostrare che **l'enunciato S_{n+1} è vero** ?

 - L'enunciato S_{n+1} è: $\sum_{i=1}^{n+1} i = (n + 1)(n + 2)/2$
 - Per dimostrare che un'uguaglianza è vera, **ad esempio** si può partire dal primo membro, cercando di renderlo uguale al secondo
 - $\sum_{i=1}^{n+1} i = (n + 1) + \sum_{i=1}^n i = (n + 1) + n(n + 1)/2$
 $= (n + 1)(n + 2)/2$

In rosso la cosiddetta "ipotesi induttiva"

Dimostrazione per induzione

- A volte si usa una enunciazione un po' diversa,
che in realtà è **assolutamente identica**...
- Sia S_1, S_2, S_3, \dots una sequenza di enunciati
L'enunciato S_n è vero $\forall n \in \mathbf{Z}^+$ se valgono le due
seguenti proprietà
 - 1) L'enunciato S_1 è vero
 - 2) $\forall n \in \mathbf{Z}^+ \setminus \{1\}$, il fatto che l'enunciato S_{n-1} sia vero,
implica che l'enunciato S_n è vero

Induzione Forte

- A volte è utile una forma ancora diversa
 - Sembra "più potente", in realtà (si dimostra che) è equivalente
- Sia S_1, S_2, S_3, \dots una sequenza di enunciati
- **Principio di Induzione Matematica "Forte"**
 - L'enunciato S_n è vero $\forall n \in \mathbb{Z}^+$ se valgono le due seguenti proprietà
 - 1) L'enunciato S_1 è vero
 - 2) $\forall n \in \mathbb{Z}^+$, il fatto che
gli enunciati S_1, S_2, \dots, S_n siano veri, implica che l'enunciato S_{n+1} è vero

Induzione Generale

- Più in generale (ma di utilizzo meno frequente), l'induzione matematica si può esprimere in questo modo
- Sia $S_K, S_{K+1}, S_{K+2}, \dots$ una sequenza di enunciati e siano K e L due numeri interi, con $K \leq L$

□ Principio di Induzione Matematica "Generale"

- L'enunciato S_n è vero $\forall n \in \mathbb{Z}$, $n \geq K$ se valgono le due seguenti proprietà

1) Gli enunciati S_K, S_{K+1}, \dots, S_L sono veri
("casi base")

2) $\forall n \in \mathbb{Z}$, $n \geq L$, il fatto che

Per $K = L = 1$,
è la forma forte

gli enunciati S_K, \dots, S_n siano veri,
implica che l'enunciato S_{n+1} è vero

Esercizio (di matematica... 😊)

□ Sia $fib(n)$ l' n -esimo numero della sequenza di Fibonacci:

$$fib(1) = 1, fib(2) = 1, \quad \forall n > 2 \quad fib(n) = fib(n - 1) + fib(n - 2)$$

□ Dimostrare, mediante induzione matematica, che

□ $\forall n \geq 1, fib(n) < 2^n$

□ Dimostrare, mediante induzione matematica, che

□ $\forall n \geq 3, fib(n) > (6/5)^n$

□ Dimostrare, mediante induzione matematica, che

□ $\forall n \geq 1, \quad fib(n) = \frac{1}{\sqrt{5}} \left(\left(\frac{1 + \sqrt{5}}{2} \right)^n - \left(\frac{1 - \sqrt{5}}{2} \right)^n \right)$

Quest'ultima dimostrazione equivale a dimostrare la **correttezza dell'algoritmo** che calcola $fib(n)$ usando la formula stessa!

Suggerimento: **NON servono i coefficienti binomiali**

Esercizio (induzione per Fibonacci)

- Sia $fib(n)$ l' n -esimo numero della sequenza di Fibonacci:

$$fib(1) = 1, fib(2) = 1, \forall n > 2 fib(n) = fib(n - 1) + fib(n - 2)$$

- Dimostrare, mediante induzione matematica, che

- $\forall n \geq 1, fib(n) < 2^n$

- Passo induttivo, $\forall n \geq 2$ dimostrare che,

- se $fib(n) < 2^n$ e $fib(n - 1) < 2^{n-1}$, allora $fib(n + 1) < 2^{n+1}$

- Non si può dimostrare il passo induttivo $\forall n \geq 1$, perché per esprimere $fib(n + 1)$ servono $fib(n)$ e $fib(n - 1)$: quest'ultima quantità non è definita per $n = 1$

- Bisogna, quindi, usare l'induzione generale, con $K = 1$ e $L = 2$, usando **due casi base** (per $n = 1$ e $n = 2$)

- I casi base sono evidentemente verificati, direttamente dalla definizione

Esercizio (induzione per Fibonacci)

- Sia $fib(n)$ l' n -esimo numero della sequenza di Fibonacci:
 - $fib(1) = 1, fib(2) = 1, \forall n > 2 fib(n) = fib(n - 1) + fib(n - 2)$
- Dimostrare, mediante induzione matematica, che
 - $\forall n \geq 1, fib(n) < 2^n$
- Passo induttivo, $\forall n \geq 2$ dimostrare che,
se $fib(n) < 2^n$ e $fib(n - 1) < 2^{n-1}$, allora $fib(n + 1) < 2^{n+1}$
 - **Ipotesi induttiva:** $\forall n \geq 2, fib(n) < 2^n$ e $fib(n - 1) < 2^{n-1}$
 - **Tesi:** $fib(n + 1) < 2^{n+1}$
 - Quindi:
$$fib(n + 1) = fib(n) + fib(n - 1) < 2^n + 2^{n-1} < 2^n + 2^n = 2 \cdot 2^n = 2^{n+1}$$

FARE LE ALTRE DIMOSTRAZIONI

Induzione

- Usando il principio di induzione, dimostrare che $\forall n \in \mathbb{Z}^+$ il numero $a(n) = n^3 + 5n$ è divisibile per 6

Induzione

- Si consideri la sequenza così definita:

$$p(1) = 10, p(2) = 10, p(3) = 10$$

$$p(n) = 6n + p(n - 3) \quad \forall n > 3$$

- Usando l'induzione, dimostrare che $p(n) \geq n^2 \quad \forall n \in \mathbb{Z}^+$

Invariante di ciclo

Dimostrazione con invariante di ciclo

- Questo tipo di dimostrazione è spesso utile per provare la correttezza di un algoritmo che contenga un ciclo, di cui si eseguono k iterazioni
- **Struttura generale di un algoritmo iterativo**
 - Inizializzazione (parte dell'algoritmo eseguita prima del ciclo)
 - Prima iterazione del ciclo
 - Seconda iterazione del ciclo
 - ...
 - Generica (i -esima) iterazione del ciclo
 - ...
 - Ultima (k -esima) iterazione del ciclo
 - Conclusione (parte dell'algoritmo eseguita dopo il ciclo)

ciclo

Dimostrazione con invariante di ciclo

- Un "invariante di ciclo" (o, meglio, una "condizione invariante durante l'esecuzione di un ciclo") è un enunciato (S_i) che è vero all'inizio della generica iterazione i -esima del ciclo stesso, $\forall i$
 - Solitamente l'enunciato S_i dipende da i , ma questo non è necessario
- Dopo aver individuato S_i bisogna **dimostrare** che:
 - **Se l'enunciato S_i è vero all'inizio della i -esima iterazione, allora è vero anche all'inizio dell'iterazione successiva (anche se essa non sarà eseguita)**
 - Dimostrato questo, ovviamente ne consegue che, **se** l'enunciato S_i è vero all'inizio della prima iterazione, **allora** è vero alla fine dell'ultima iterazione
 - Cioè la validità dell'enunciato S_i viene **preservata** non soltanto dalla singola iterazione del ciclo, ma **dall'esecuzione dell'intero ciclo**: è invariante!
- A questo punto, bisogna dimostrare ancora due cose:
 - **L'enunciato S_i è vero prima dell'inizio del ciclo**
 - **Se l'enunciato S_i è vero al termine del ciclo, allora l'algoritmo è corretto**
- La concatenazione delle **3 dimostrazioni** prova la correttezza

Dimostrazione con invariante di ciclo

Esempio: **dimostrare la correttezza dell'algoritmo che trova l'elemento massimo in un array** (qui descritto con pseudocodice)

```
findMax(int[ ] a, int n) // array a di dimensione n > 0
    ...
    m = a[0] // candidato massimo; esiste perché n > 0
    for (i = 1; i < n; i++)
        if (a[i] > m)
            m = a[i] // nuovo candidato massimo
    return m
```

Enunciato S_i : all'inizio della i -esima iterazione, m contiene il valore massimo presente nelle prime i posizioni dell'array a , cioè nelle celle $a[0], \dots, a[i - 1]$ (alla prima iterazione, $i = 1$, quindi c'è solo $a[0]$)

Per "inizio dell'iterazione" si intende **questo punto**, prima della verifica della condizione di terminazione del ciclo

Dimostrazione con invariante di ciclo

Enunciato S_i : all'inizio della i -esima iterazione, m contiene il valore massimo presente nelle prime i posizioni dell'array a , cioè nelle celle $a[0], \dots, a[i - 1]$

Devo dimostrare che: se S_i è vero all'inizio dell'iterazione i -esima, allora è vero anche all'inizio dell'iterazione successiva, la $(i+1)$ -esima.

Dimostrazione.

All'inizio della $(i+1)$ -esima iterazione, m deve essere uguale al massimo tra i primi $i+1$ elementi di a , da $a[0]$ a $a[i]$ compreso.

Se tale valore massimo si trova in $a[i]$, questo valore viene assegnato a m proprio durante la i -esima iterazione, quindi la tesi è dimostrata.

Se, invece, tale valore massimo non si trova in $a[i]$, l'esecuzione della i -esima iterazione non modifica m , che **per ipotesi** era uguale al valore massimo dei primi i elementi di a e sarà, quindi, anche uguale al valore massimo dei primi $i+1$ elementi di a (perché il massimo non è in $a[i]$).

Dimostrazione con invariante di ciclo

Enunciato S_i : all'inizio della i -esima iterazione, m contiene il valore massimo presente nelle prime i posizioni dell'array a , cioè nelle celle $a[0], \dots, a[i - 1]$

Tale enunciato è vero prima del ciclo, quando $i = 1$?

Cioè, è vero che, prima del ciclo, m contiene il valore massimo presente nelle prime i posizioni (con $i = 1$) dell'array?

Sì, per effetto dell'assegnazione iniziale, $m = a[0]$.

Se tale enunciato è vero alla fine del ciclo, l'algoritmo è corretto?

Alla fine del ciclo (cioè all'inizio dell'iterazione successiva all'ultima, che non verrà eseguita), $i = n$, quindi l'enunciato afferma che m contiene il valore massimo presente nelle prime n posizioni dell'array, cioè il valore massimo presente nell'intero array. L'enunciato afferma, quindi, in modo diretto che l'algoritmo è corretto, non c'è bisogno di dimostrare niente di più.

Esercizio

FARE A CASA

```
static long factorial(int n)
{   long f = 1;
    for (int i = 2; i <= n; i++)
        f = f * i;
    return f;
}
```

In questo ciclo una possibile condizione invariante è

All'inizio dell'iterazione di indice i , f è uguale al prodotto dei numeri interi positivi minori di i

Dimostrare che tale condizione è invariante e utilizzarla per dimostrare la correttezza di questo algoritmo di calcolo del fattoriale

Spesso in un ciclo ci sono molte condizioni invarianti, ma alcune sono inutili (in questo esempio, $i > 0$ è un invariante... inutile)

Esercizio

FARE A CASA

Progettare un metodo Java che:

- riceve un array a di numeri interi ordinato "in senso crescente"
 - cioè al crescere dell'indice crescono i valori presenti in a
- riceve un numero intero x
- restituisce il valore booleano **true** se e solo se x è presente nell'array a
- implementa l'algoritmo di ricerca binaria **in forma iterativa** (cioè senza utilizzare la ricorsione)

Poi, **individuare una condizione invariante nel ciclo e utilizzarla per dimostrare la correttezza del metodo.**

Suggerimento: **pensare bene a quale sia una condizione invariante utile per la dimostrazione di correttezza!**

Attenzione: **il metodo deve essere corretto sia quando risponde *true* sia quando risponde *false***

Lezione 06

Prestazioni di un algoritmo

Prestazioni di un algoritmo

- Dato che possono esistere diversi algoritmi che risolvono uno stesso problema computazionale, spesso di un algoritmo interessano le **prestazioni**, in termini di
 - **Tempo di esecuzione**
 - **Occupazione di memoria**
 - "temporanea", cioè in aggiunta a quella necessaria per memorizzare i dati dell'esemplare del problema e della soluzione di esso (perché queste due ultime quantità non dipendono dall'algoritmo)
 - Consumo di energia e altro...
 - In generale, le prestazioni (temporali e spaziali) di un algoritmo dipendono dalla **dimensione** o **taglia (size)** dell'esemplare che viene risolto

Dimensione di un problema

- La **dimensione** di un problema computazionale è **una funzione** che assegna a ciascun esemplare del problema **uno o più numeri (solitamente interi)** che forniscono una ***misura ragionevole*** dell'esemplare stesso. **Cosa vuol dire?**
- Una dimensione è una "misura ragionevole" degli esemplari di un problema se le **prestazioni** degli algoritmi che risolvono il problema **dipendono** (con buona approssimazione) **soltanto dalla dimensione stessa**
- Una dimensione induce una **partizione** nell'insieme I degli esemplari del problema, ottenuta raggruppando gli esemplari aventi la stessa dimensione
 - Quindi, per esemplari del problema aventi la stessa dimensione, un algoritmo opera (circa) nello stesso tempo, usando (circa) la stessa quantità di memoria

Dimensione di un problema

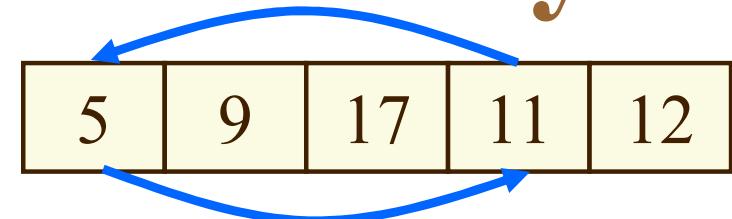
- Non c'è una regola generale per individuare la dimensione di un problema, ci si basa **sull'esperienza**
- Nel problema che calcola la somma di due numeri interi, una possibile dimensione è il numero di cifre necessarie a esprimere il risultato in una qualche base
 - In particolare, questa dimensione è ragionevole per l'algoritmo che calcola la somma "in colonna"
- **Solitamente**, nei problemi che elaborano strutture lineari (array, pile, code, liste, ecc.), la dimensione è **il numero di dati contenuti nella struttura** (che può essere inferiore alla dimensione della struttura stessa)
- In **strutture bidimensionali** (matrici, alberi, ecc.) a volte la dimensione è data da **due** numeri interi (in una matrice, ad esempio, la dimensione può essere il numero di righe e il numero di colonne, OPPURE il loro prodotto...)

Analisi delle prestazioni di algoritmi e ripasso di algoritmi operanti su array

Parte di questo materiale è contrassegnato
con la scritta "**RIPASSO AUTONOMO**"

RIPASSO AUTONOMO

SelectionSort per un array



- Per prima cosa, bisogna trovare *la posizione dell'elemento minore presente nell'intero array*
 - in questo caso è il numero 5 in posizione $a[3]$
- Essendo l'elemento minore, la sua **posizione finale corretta** nell'array ordinato è $a[0]$
 - in $a[0]$ è però memorizzato il numero 11, da spostare
 - non sappiamo quale sarà la posizione finale di 11
 - lo spostiamo temporaneamente in $a[3]$
 - quindi, *scambiamo $a[3]$ con $a[0]$* usando una sola variabile temporanea come spazio di memorizzazione aggiuntivo

RIPASSO AUTONOMO

Ordinamento per selezione

5	9	17	11	12
---	---	----	----	----

a[0] a[1] a[2] a[3] a[4]

la parte scura dell'array è già ordinata

□ La parte sinistra dell'array è già ordinata e non sarà più considerata, dobbiamo ordinare la parte destra

□ Ordiniamo la parte destra *con lo stesso algoritmo*

- cerchiamo l'elemento minore nella porzione non colorata, che è 9 in posizione a[1]

- dato che *è già nella prima posizione a[1] della parte da ordinare, non c'è bisogno di fare scambi*

- **la porzione ordinata cresce...**

5	9	17	11	12
---	---	----	----	----

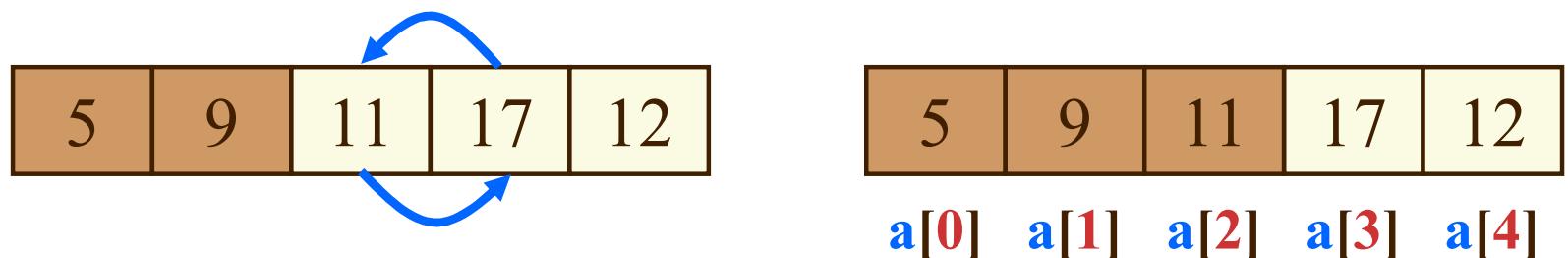
a[0] a[1] a[2] a[3] a[4]

RIPASSO AUTONOMO

Ordinamento per selezione

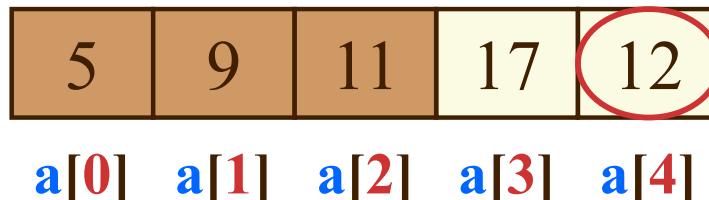


- Proseguiamo per ordinare la parte di array che contiene gli elementi **a[2]**, **a[3]** e **a[4]**
 - l'elemento minore è il numero **11** in posizione **a[3]**
 - scambiamo **a[3]** con **a[2]**



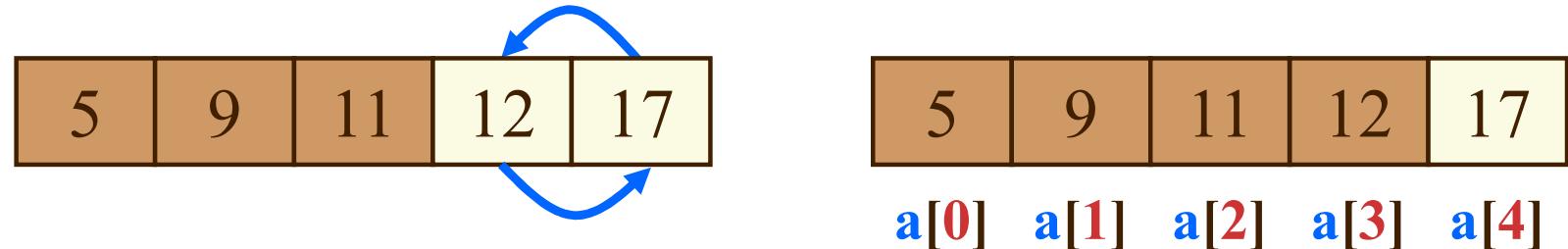
RIPASSO AUTONOMO

Ordinamento per selezione



□ Ora l'array da ordinare contiene **a[3]** e **a[4]**

- l'elemento minore è il numero **12** in posizione **a[4]**
- scambiamo **a[4]** con **a[3]**



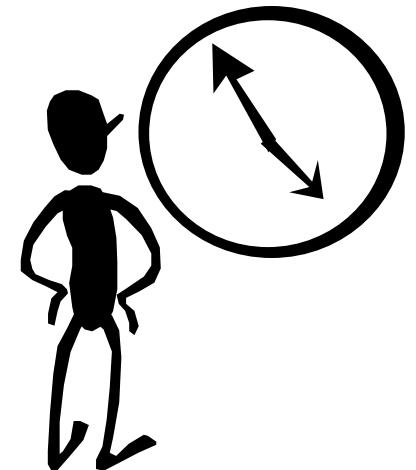
□ A questo punto *la parte da ordinare contiene un solo elemento*, quindi *è ovviamente ordinata*

Rilevazione delle prestazioni

RIPASSO AUTONOMO

Rilevazione delle prestazioni

- Per valutare l'efficienza temporale di un algoritmo si misura il tempo in cui viene eseguito su insiemi di dati di dimensioni via via maggiori
- Il tempo *non va misurato con un cronometro*, perché alcune componenti del tempo reale di esecuzione non dipendono dall'algoritmo stesso
 - caricamento della JVM
 - caricamento delle classi del programma
 - lettura dei dati dallo standard input
 - visualizzazione dei risultati
- Tali componenti sono, poi, anche variabili nel tempo, da un'esecuzione all'altra (bisognerebbe prendere un tempo di esecuzione medio)



RIPASSO AUTONOMO

Rilevazione delle prestazioni

- Il tempo di esecuzione di un algoritmo va misurato all'interno del programma
- Si può usare, ad esempio, il metodo statico **System.currentTimeMillis()** che, a ogni invocazione, restituisce un valore di tipo long che rappresenta:
 - il numero di millisecondi trascorsi da un evento di riferimento (*la mezzanotte del 1 gennaio 1970*)
- Ciò che interessa è la **differenza** tra due valori
 - si invoca **System.currentTimeMillis()** *prima e dopo* l'esecuzione dell'algoritmo (escludendo le operazioni di input/output dei dati)

RIPASSO AUTONOMO

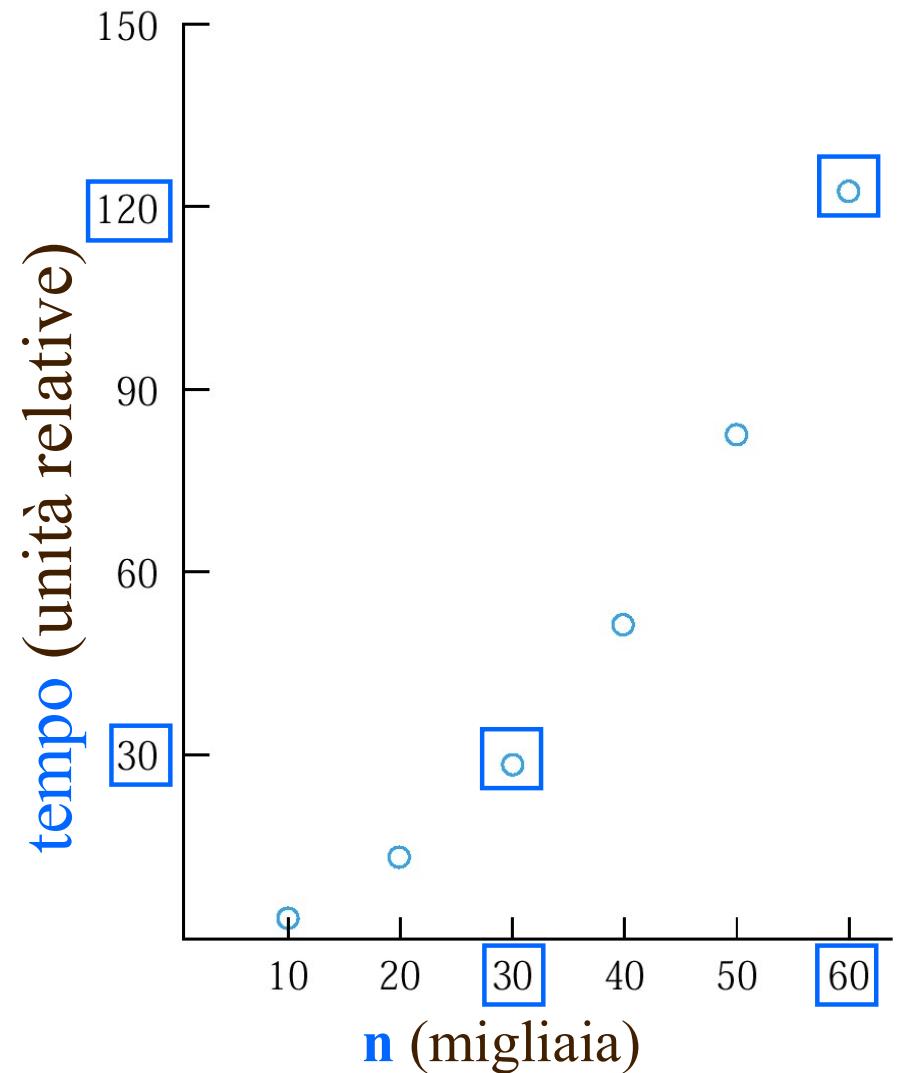
Rilevazione delle prestazioni

```
public static void main(String[] args)
{ int[] a = ...;
  long initTime = System.currentTimeMillis();
  iterSelectionSort(a);
  long finalTime = System.currentTimeMillis();
  long elapsedTime = (          // tempo trascorso
                        Math.round((finalTime - initTime) / 1000.0));
  System.out.print("Array ordinato in ");
  System.out.print(elapsedTime);
  System.out.println(" secondi");
}
```

RIPASSO AUTONOMO

Rilevazione delle prestazioni

- Usando la selezione, eseguiamo l'ordinamento di array di diverse dimensioni (**n**), contenenti numeri interi casuali
- Osserviamo l'andamento del tempo di esecuzione
 - *se n diventa il doppio ($30 \rightarrow 60$), il tempo diventa circa il quadruplo ($30 \rightarrow 120$) !!*



RIPASSO AUTONOMO

Rilevazione delle prestazioni

- Le prestazioni dell'algoritmo di ordinamento per selezione hanno quindi un *andamento quadratico* (o *parabolico*) in funzione delle dimensioni dell'array
- Le prossime domande che ci poniamo sono
 - *è possibile valutare le prestazioni di un algoritmo dal punto di vista teorico?*
 - cioè **senza** realizzare un programma e **senza** eseguirlo molte volte al solo fine di rilevarne i tempi di esecuzione e osservarne l'andamento?
 - *esiste un algoritmo più efficiente per l'ordinamento?*

Analisi teorica delle prestazioni

Analisi teorica delle prestazioni

- Il tempo d'esecuzione di un algoritmo dipende dal numero e dal tipo di istruzioni in *linguaggio macchina* eseguite dal processore
- Per fare un'analisi teorica
 - senza realizzare un algoritmo, compilarlo e tradurlo in linguaggio macchina!
 - dobbiamo fare delle *semplificazioni drastiche*
 - *durante l'esecuzione del codice dell'algoritmo, contiamo soltanto gli accessi in lettura o scrittura a singoli elementi dell'array*, ipotizzando che tali accessi siano le operazioni elementari più lente durante l'esecuzione del programma e che tutte le altre attività richiedano un tempo **trascurabile**

Analisi teorica delle prestazioni

- *Durante l'esecuzione del codice dell'algoritmo, contiamo soltanto gli accessi in lettura o scrittura a singoli elementi dell'array, ipotizzando che tali accessi siano le operazioni elementari più lente durante l'esecuzione del programma e che tutte le altre attività richiedano un tempo **trascutibile***
 - Ipotesi ragionevole se, come spesso accade, le variabili locali (e parametro) di tipi primitivi vengono conservate nei registri della CPU, con accesso estremamente più veloce di quello verso la memoria principale, dove si trovano invece gli oggetti (e, quindi, anche gli array)
 - **Più in generale, si contano le operazioni più onerose dal punto di vista del tempo di esecuzione**
 - Esempio: **in una struttura concatenata**, come una lista concatenata, l'operazione più onerosa è il passaggio da un oggetto all'altro seguendo un collegamento di concatenazione

RIPASSO AUTONOMO

Analisi teorica delle prestazioni

- Ordiniamo con la selezione un array di **n** elementi
 - per trovare la posizione dell'elemento minore si fanno **n** accessi
 - bisogna, ovviamente, leggere una (e una sola) volta tutti gli elementi dell'array
 - per scambiare due elementi si fanno **quattro** accessi
 - `temp = a[i];`
 - `a[i] = a[j];`
 - `a[j] = temp;`
 - trascuriamo il caso in cui non serva lo scambio, non ci interessano i casi “fortunati”... valutiamo le prestazioni “**nel caso peggiore**”
 - in totale, al primo passo servono (**n+4**) accessi

RIPASSO AUTONOMO

Analisi teorica delle prestazioni

- Ordiniamo con la selezione un array di **n** elementi
 - in totale, al primo passo servono (**n+4**) accessi
 - A questo punto dobbiamo ordinare la parte rimanente, cioè un array di (**n-1**) elementi
 - serviranno quindi (**(n-1) + 4**) accessi
- e via così fino al passo con **n=2**, incluso, che richiede (**2+4**) accessi

RIPASSO AUTONOMO

Analisi teorica delle prestazioni

- Il conteggio totale $T(n)$ degli accessi in lettura è quindi

$$\begin{aligned} T(n) &= (n+4) + ((n-1)+4) + \dots + (3+4) + (2+4) \\ &= n + (n-1) + \dots + 3 + 2 + (n-1) * 4 \\ &\stackrel{\rightarrow}{=} n * (n+1) / 2 - 1 + (n-1) * 4 \\ &= 1/2 * n^2 + 9/2 * n - 5 \end{aligned}$$

- Si ottiene *un'equazione di secondo grado* in n , che giustifica l'andamento *parabolico* dei tempi rilevati sperimentalmente (anche se in realtà questa formula non calcola il tempo, ma il numero di accessi)

Formula di Gauss

$$n + (n-1) + \dots + 3 + 2 + 1 = n * (n+1) / 2$$

RIPASSO AUTONOMO

Andamento asintotico delle prestazioni

$$1/2*n^2 + 9/2*n - 5$$

- Facciamo un'ulteriore osservazione, tenendo presente che ci interessano le prestazioni degli algoritmi per *valori elevati di n* (*andamento asintotico*)
- Se **n** vale 1000
 - $1/2*n^2$ vale 500000
 - $9/2*n - 5$ vale 4495, circa 1% del totale, assolutamente trascurabile
(viste le semplificazioni drastiche che abbiamo già fatto)

quindi diciamo che l'andamento asintotico delle prestazioni temporali dell'algoritmo in funzione di **n** è circa $1/2*n^2$

Andamento asintotico delle prestazioni

- In realtà, in generale quello che interessa **per confrontare le prestazioni temporali di due algoritmi** è una misura ancora più grossolana (ma rilevante), relativa soprattutto a valori di **n** molto elevati (quindi, al limite, per **n** che tende all'infinito)
 - Ci interessa sapere, più o meno, *come* cresce la funzione, cioè ***quale andamento qualitativo ha: è un retta? è una parabola?***
 - Confronteremo algoritmi diversi sulla base del loro andamento asintotico qualitativo, riservando ulteriori eventuali approfondimenti quantitativi soltanto ai casi di andamento asintotico equivalente (es. due algoritmi parabolici)

Andamento asintotico delle prestazioni

- Ci chiediamo ora: esiste una formulazione matematica del concetto “*quale andamento* ha una funzione di n , per n che tende all’infinito” ?
 - **Risposta: sì!**

Notazione O-grande

- La formulazione matematica dice che

Si legge: “ $f(n)$
è **O-grande**
di $g(n)$ ”

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

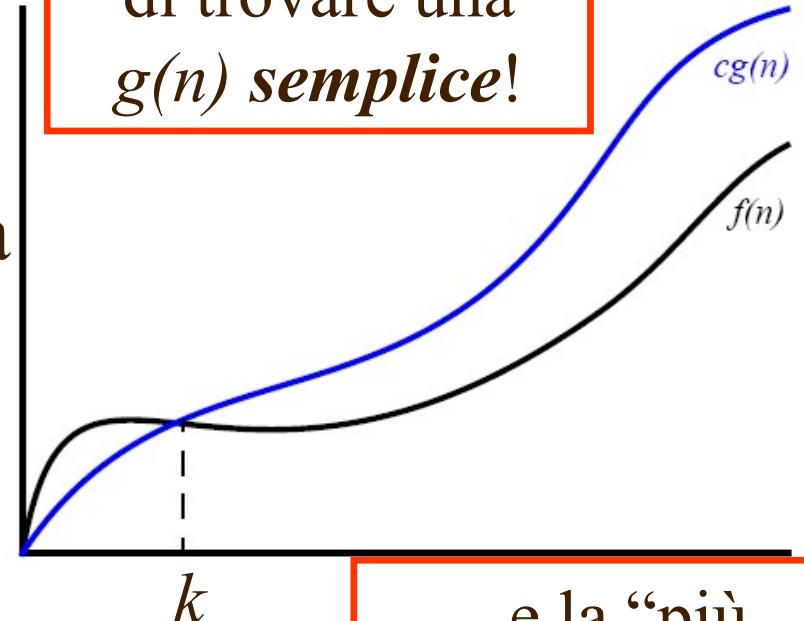
se $\exists c > 0$ e $k > 0$ tali che

$$f(n) \leq c g(n) \quad \forall n \geq k$$

- Cioè, al crescere di n , **“prima o poi”** (dipende da $k\dots$)
 $g(n)$ è sempre maggiore di $f(n)$,
a meno di una costante moltiplicativa ($c\dots$)
- Si dice anche che $g(n)$ è un **limite superiore asintotico** per $f(n)$

Si usa il segno \in perché $O(g(n))$ è un simbolo che rappresenta l’insieme di tutte le funzioni $f(n)$ che rispondono alla definizione per una determinata funzione $g(n)$

Si ha l’obiettivo
di trovare una
 $g(n)$ *semplice*!



... e la “più
bassa” $g(n)$!

Esercizio

□ Dimostrare che $(n + 1)^5 \in O(n^5)$

□ Dobbiamo trovare due costanti positive, c e k , tali che

$$(n + 1)^5 \leq c (n)^5 \quad \forall n \geq k$$

□ Osserviamo che

$$(n + 1)^5 = n^5 + 5n^4 + 10n^3 + 10n^2 + 5n + 1$$

$$n \geq 1 \Rightarrow 1 \leq n^5, n \leq n^5, n^2 \leq n^5, n^3 \leq n^5, n^4 \leq n^5$$

Quindi, ad esempio, $5n^2 \leq 5n^5$, per cui:

$$(n + 1)^5 \leq (1 + 5 + 10 + 10 + 5 + 1) n^5 = 32 n^5$$

Quindi con $c = 32$ e $k = 1$ la definizione di O-grande è verificata

Esercizio (continua)

- Attenzione se il polinomio ha qualche monomio con coefficiente negativo, perché, ad esempio:

$$n^4 \leq n^5 \Rightarrow -5n^4 \geq -5n^5$$

- E allora? È, però, vero che $-5n^4 < 0$. Quindi

$$(n - 1)^5 = n^5 - 5n^4 + 10n^3 - 10n^2 + 5n - 1$$

$$< n^5 + 10n^3 + 5n \leq (1 + 10 + 5) n^5 = 16 n^5$$

Quindi con $c = 16$ e $k = 1$ si dimostra che
 $(n - 1)^5 \in O(n^5)$

Nota: è inutile fare sforzi per cercare k e/o c “piccoli”...
ne basta una coppia!

Esercizio (continua)

- Più in generale, in modo assolutamente analogo si dimostra che **un polinomio è O-grande del suo monomio di grado massimo**
- Attenzione: cosa succede se il monomio **di grado massimo** del polinomio $P(n)$ ha coefficiente **negativo**?
 - Al crescere di n , prima o poi $P(n)$ diventa negativo
- **Non ha senso che il tempo di esecuzione di un algoritmo sia negativo !!!**
 - In un caso come questo, è stato fatto un errore nel calcolo della funzione che costituisce un modello del tempo di esecuzione

RIPASSO AUTONOMO

Notazione “O grande”

- Proprietà: Una funzione polinomiale è un O-grande del suo monomio di grado massimo, con coefficiente moltiplicativo arbitrario
 - Quindi, per semplicità, tale coefficiente viene assunto uguale a 1
- Nel caso peggiore dell'ordinamento per selezione, quindi:

$$f(n) = 1/2 * n^2 + 9/2 * n - 5 \in O(n^2)$$

- Usando la notazione O-grande per rappresentare le funzioni non abbiamo più bisogno di osservare che i monomi di grado non massimo all'interno di una funzione polinomiale non danno contributo: questo viene “eliminato” usando un valore di k opportunamente grande
- Analogamente, non c’è alcun contributo dall’eventuale coefficiente moltiplicativo del monomio di grado massimo, anche se maggiore di uno: questo viene “eliminato” usando un valore di c opportunamente grande

RIPASSO AUTONOMO

Notazione “O grande”

- **Proprietà: Una funzione polinomiale è un O-grande del suo monomio di grado massimo, con coefficiente pari a uno**
- Più in generale, per funzioni non polinomiali non è sempre facile individuare una funzione che ne caratterizzi l'andamento. Alcune regole:
 - Se la funzione è somma di funzioni, si possono eliminare tutti gli addendi che crescono più lentamente di un altro addendo presente
 - Si possono eliminare tutti i coefficienti (costanti) moltiplicativi
- Esempio

$$3n + \log_{10}(n^{22}) + 200 \in O(n)$$

perché la funzione logaritmo (in qualunque base) cresce più lentamente della funzione lineare e, ricordiamo che

$$\log_{10}(n^{22}) = 22 \log_{10}n$$

RIPASSO AUTONOMO

Notazione “O grande”

- Vediamo un altro esempio

$$3n + 7n^2 * \log_{10}(n^{22}) \in O(n^2 * \log_{10}n)$$

- Ricordiamo, però, che

$$\log_a n = (1/\log_b a) \log_b n$$

- Cioè il cambio di base in un logaritmo equivale all'applicazione di un **coefficiente moltiplicativo**, quindi in un'espressione O-grande si può omettere l'indicazione della base del logaritmo, così come si può omettere l'esponente a cui sia eventualmente elevato l'argomento del logaritmo stesso

$$\forall a, b, x: \log_a n^b \in O(\log_x n) \equiv O(\log n)$$

RIPASSO AUTONOMO

O-grande più stringente

- La definizione matematica implica che
 - se $f(n) \in O(g(n))$, allora è anche $f(n) \in O(h(n))$ per ogni funzione **h(n)** che cresca più velocemente di **g(n)**
- Ovviamente, la caratterizzazione che interessa di più è quella più precisa, cioè *più stringente* o “nei minimi termini”
- Però se, ad esempio, **T(n) ∈ O(n²)**, non è sbagliato (ma è poco utile...) dire che

T(n) ∈ O(n² log n) oppure **T(n) ∈ O(n³)** perché
T(n) ∈ O(n²) ⊂ O(n² log n) ⊂ O(n³) ⊂ ...

Attenzione però nei questionari:
è vero che l'ordinamento per selezione è **O(n³)**? Risposta: sì

O-grande più stringente

- Quand'è che un O-grande è quello "**più stringente**" per la funzione $f(n)$?
- Ricordiamo la definizione di **o-piccolo** e...
- Se $f(n) \in \mathbf{O}(g(n))$ e $f(n) \notin \mathbf{o}(g(n))$
allora $\mathbf{O}(g(n))$ è l'O-grande più stringente per $f(n)$
- Ovviamente $g(n)$ non è unica, ma tutte le possibili $g(n)$ che soddisfano la definizione precedente appartengono allo stesso O-grande, cioè sono "asintoticamente equivalenti"

RIPASSO AUTONOMO

Notazioni improprie ma accettate

Spesso, anziché

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

si scrive, impropriamente

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

- La notazione è accettata in questo corso e, in generale, in ambito informatico, ma bisogna sempre ricordare che
 - $O(g(n))$ è un simbolo che rappresenta **l'insieme** di tutte le funzioni $f(n)$ che rispondono alla definizione di O-grande per una determinata funzione $g(n)$

RIPASSO AUTONOMO

Notazioni improprie ma accettate

Spesso, anziché

$$T(n) = O(f(n)) \subseteq O(g(n))$$

si scrive, impropriamente

$$T(n) = O(f(n)) = O(g(n))$$

Evitiamo...

- Esempio: $5n + 2 = O(n) = O(n^2)$
- Accettabile, ma molto pericoloso... perché si tratta di una relazione che **NON** gode della proprietà **riflessiva!!** Mentre siamo abituati a usare il segno = per relazioni riflessive e ciò può trarre in inganno...

$O(n) \subseteq O(n^2)$ non implica che $O(n^2) \subseteq O(n)$

mentre **SEMBRA** che...

$O(n) = O(n^2) \Rightarrow O(n^2) = O(n)$

Esercizio

- Dimostrare che $2^{n+1} \in O(2^n)$
- Dobbiamo trovare due costanti, c e k , tali che

$$2^{n+1} \leq c (2^n) \quad \forall n \geq k$$

- Osserviamo che

$$\forall n \text{ (e, quindi, } \forall n \geq 1) \Rightarrow 2^{n+1} = 2 \cdot 2^n \leq 2 \cdot 2^n$$

Quindi con $c = 2$ e $k = 1$ la definizione di O-grande è verificata

Esercizio

- Dimostrare che $2^{2n} \notin O(2^n)$
- Per assurdo, suppongo che $2^{2n} \in O(2^n)$
 - Allora $\exists c > 0, k > 0$ tali che $2^{2n} \leq c 2^n \forall n \geq k$
Ma $2^{2n} \leq c 2^n \Leftrightarrow 2^n 2^n \leq c 2^n \Leftrightarrow 2^n \leq c \forall n \geq k$
E questo è assurdo perché $2^n > c \forall n > \log_2 c$
 - Quindi NON è vero che $2^{2n} \in O(2^n)$, cioè $2^{2n} \notin O(2^n)$

Lezione 07

Ordinamento per selezione

- Facendo ipotesi semplificative e contando soltanto gli accessi a singole celle dell'array
 - Abbiamo calcolato che le prestazioni temporali asintotiche dell'algoritmo di ordinamento per selezione, applicato a un array, sono **$O(n^2)$** **nel caso peggiore**
- Abbiamo verificato sperimentalmente, mediante misurazioni temporali di esecuzioni di prova, che il modello (quadratico) è realistico
 - Come si dice, abbiamo “**validato**” il modello
- Cosa succede nel caso migliore e medio?

Selezione nel “caso migliore”

- Ordiniamo con la selezione un array di **n** elementi
 - per trovare la posizione dell’elemento minore si fanno **n** accessi
 - nel “**caso migliore**” (cioè “il più fortunato”) non è mai necessario fare scambi
 - ogni minimo che troviamo si trova già nella sua posizione finale corretta: **l’array era già ordinato!**
- In totale, **n** accessi per trovare il primo minimo
- A questo punto dobbiamo ordinare la parte rimanente, cioè un array di **(n-1)** elementi
 - Il passo successivo richiede, nel caso migliore, **(n-1)** accessie via così fino al passo con **n=2**, incluso, che richiede **2** accessi

Selezione nel “caso migliore”

- Il conteggio totale degli accessi in lettura è quindi

$$\begin{aligned} T(n) &= n + (n-1) + \dots + 3 + 2 \\ &= n * (n+1) / 2 - 1 && \text{Usando la} \\ &= 1/2 * n^2 + 1/2 * n - 1 && \text{formula di Gauss} \\ &\in O(n^2) \end{aligned}$$

- L'ordinamento per selezione è, quindi,
quadratico anche nel caso migliore,
oltre che nel caso peggiore

- Non trae (asintoticamente) alcun vantaggio dal fatto che l'array da ordinare sia già ordinato!

Selezione nel “caso medio”

- Di solito le prestazioni nel “caso migliore” hanno poco interesse
 - Interessano molto di più le prestazioni nel “caso peggiore” e nel “caso medio”
- **Non è semplice definire le condizioni di “caso medio”**
 - bisogna fare la **media** del numero di accessi necessario per ordinare **tutte le possibili permutazioni dei dati iniziali**
 - cioè, più in generale, **per risolvere tutte le possibili istanze del problema** risolubile dall'algoritmo
 - in pratica, spesso si arriva a una definizione adeguata di "caso medio" in modo più semplice, con l'esperienza e la matematica...
 - in questo caso, **dato che l'ordinamento per selezione è quadratico sia nel caso migliore sia nel caso peggiore, non può che essere quadratico anche nel caso medio!**

Esercizio sulle dimensioni

- Un algoritmo, `find2D`, che cerca un elemento x all'interno di un array bidimensionale A , di dimensioni $n \times n$ (una matrice quadrata), opera in questo modo
 - Per ogni riga di A , invoca un algoritmo di ricerca lineare (detta anche ricerca sequenziale) sull'array unidimensionale di dimensione n che contiene la riga, finché trova x oppure ha cercato in tutte le n righe di A (senza trovare x)
- Quali sono le prestazioni temporali asintotiche dell'algoritmo, in funzione di n ?
- Quali sono le prestazioni temporali asintotiche dell'algoritmo, in funzione di $E = n \times n$, cioè del numero di elementi?
- È corretto dire che si tratta di un **algoritmo lineare**?
 - Perché sì o perché no?

Esercizio (continua)

- Nel caso peggiore (ad esempio, quando $x \notin A$) ognuna delle n invocazioni della ricerca lineare risulta essere $O(n)$
 - Quindi $\text{find2D}(n) \in O(n^2)$
- Il numero di elementi di A è, però, $E = n \times n = n^2$
 - Quindi $\text{find2D}(n) \in O(E)$
- **L'algoritmo è quadratico in n e lineare in E**
- Dipende da cosa si vuole sapere... cioè da quale quantità viene definita "dimensione" del problema
 - In questo caso non c'è una dimensione migliore dell'altra, vanno bene entrambe, **basta specificare bene**

Notazioni Omega e Theta

Notazioni OMEGA e THETA

- Sappiamo che $5n + 2 \in O(n)$
 - Ma anche che $5n + 2 \in O(n^2)$, $5n + 2 \in O(n^3)$, ecc.
- Per ovviare all'indeterminazione "verso l'alto" insita nella notazione O-grande, si usano **altre due notazioni**
- Si dice che una funzione $f(n) \in \Omega(g(n))$
se $\exists c > 0$ e $k > 0$ tali che $f(n) \geq c g(n) \quad \forall n \geq k$
cioè se “prima o poi” $f(n)$ cresce **più** velocemente di $g(n)$
(a meno di una costante moltiplicativa) e, quindi, $g(n)$ è un limite **inferiore** asintotico per $f(n)$
- Esempio: $3n^2 + 4 \in \Omega(n^2)$
 - Ma anche $3n^2 + 4 \in \Omega(n)$

Analogia indeterminazione "verso il basso"...

La notazione si dovrebbe chiamare omega-grande, ma in informatica la notazione omega-piccolo non si usa... quindi spesso si dice soltanto OMEGA

Notazioni OMEGA e THETA

- Sappiamo che $5n + 2 \in O(n)$
 - Ma anche che $5n + 2 \in O(n^2)$, $5n + 2 \in O(n^3)$, ecc.
- Per ovviare all'indeterminazione "verso l'alto" insita nella notazione O-grande, si usano altre due notazioni
- Si dice che una funzione $f(n) \in \Omega(g(n))$
 - se $\exists c > 0$ e $k > 0$ tali che $f(n) \geq c g(n) \quad \forall n \geq k$
 - cioè se “prima o poi” $f(n)$ cresce più velocemente di $g(n)$
 - e, quindi, $g(n)$ è un limite **inferiore** asintotico per $f(n)$
- Si dice, ancora, che una funzione $f(n) \in \Theta(g(n))$
 - se $f(n) \in \Omega(g(n))$ **e** $f(n) \in O(g(n))$
 - cioè se “prima o poi” $f(n)$ e $g(n)$ crescono **alla stessa velocità**
- Esempi: $3n^2 + n + 4 \in \Theta(n^2)$, $5n + 2 \in \Theta(n)$

L'indeterminazione non c'è più! Una funzione non può essere Θ di due **monomi** diversi

Notazioni OMEGA e THETA

- Si dice che una funzione $f(n) \in \Theta(g(n))$ se $f(n) \in \Omega(g(n))$ e $f(n) \in O(g(n))$
cioè se “prima o poi” $f(n)$ e $g(n)$ crescono
alla stessa velocità
- Una funzione $f(n) \in \Theta(g(n))$ se
 $\exists c' > 0, c'' > 0$ e $k > 0$ tali che
 $c' g(n) \leq f(n) \leq c'' g(n) \quad \forall n \geq k$
- Ovviamente la caratterizzazione mediante Θ è quella più precisa e più utile, ma a volte è più difficile da calcolare
 - È sia l’O-grande più stringente sia l’Omega più stringente

La notazione si dovrebbe chiamare theta-grande, ma in informatica la notazione theta-piccolo non si usa... quindi spesso si dice soltanto THETA

Notazioni OMEGA e THETA

- Le funzioni che interessano in questo corso (che esprimono il **tempo di calcolo** o l'**occupazione di spazio di memoria** in funzione della dimensione n del problema) sono **sempre positive!!!**
 - Si ammette che siano negative per alcuni valori di n (tipicamente piccoli), per effetto di errori introdotti da semplificazioni del modello di stima
 - In ogni caso, **devono** essere funzioni $f(n)$ **asintoticamente positive**
 - $\exists k > 0 \mid f(n) > 0 \quad \forall n \geq k$

Notazioni OMEGA e THETA

1. Dimostrare che una funzione polinomiale è asintoticamente positiva **se e solo se** il suo monomio di grado massimo ha coefficiente positivo
 2. Dimostrare che una funzione polinomiale asintoticamente positiva è O -grande del suo monomio di grado massimo, con coefficiente arbitrario (assunto uguale a uno per comodità)
 3. Dimostrare che una funzione polinomiale asintoticamente positiva è Ω del suo monomio di grado massimo, con coefficiente arbitrario (assunto uguale a uno per comodità)
 - **Attenzione ai monomi di grado non massimo aventi coefficiente negativo**
- Quindi **una funzione polinomiale asintoticamente positiva è anche Θ del suo monomio di grado massimo**

Notazioni asintotiche per funzioni di più variabili

- A volte i problemi computazionali si caratterizzano mediante una dimensione costituita da due o più numeri (solitamente interi), anziché un solo numero
 - Es. altezza e numero di nodi di un albero, numero di righe e di colonne di una matrice, numero di vertici e di rami di un grafo
- Le funzioni che descrivono le prestazioni (temporali e/o spaziali) di algoritmi per tali problemi dipendono, quindi, da due (o più) variabili
- Le notazioni asintotiche (O , Ω e Θ) si estendono "in modo naturale" a tali casi
- Es. $f(n, m) \in \Omega(g(n, m))$ se
 - $\exists c > 0$ e $k > 0$ tali che
$$f(n, m) \geq c g(n, m) \quad \forall n \geq k \quad \forall m \geq k$$
- Es. $n + 2m^2 + m + 3 \in O(n + m^2)$ [monomi di grado massimo]

Pari e dispari...

- Nelle dimostrazioni che coinvolgono la divisione per 2 di numeri interi dispari (o di cui non è nota la divisibilità per 2) possono essere utili queste due notazioni, con le relative proprietà

$\lfloor n \rfloor$ = il massimo numero intero non maggiore di n

Math.floor(n)

$\lceil n \rceil$ = il minimo numero intero non minore di n

Math.ceil(n)

- Attenzione ai numeri negativi...

- Ricordiamo, ad esempio, che $\forall n \in \mathbb{Z}$ (quindi anche negativi...)

$$n - 1 < \lfloor n \rfloor \leq n \leq \lceil n \rceil < n + 1$$

$$\lfloor n/2 \rfloor + \lceil n/2 \rceil = n$$

Esercizio

□ Dimostrare che, $\forall a > 1$

$$\sum_{i=1}^n \log_a i \in \Theta(n \log n)$$

- Spesso è comodo fare due dimostrazioni distinte: O-grande e Ω
- Questa proprietà verrà utilizzata in seguito, bisogna memorizzarla (oltre che dimostrarla)

Ricerca sequenziale

RIPASSO AUTONOMO

Ricerca sequenziale

- Per individuare la posizione di un elemento che abbia un particolare valore all'interno di un array i cui elementi ***non siano ordinati*** (o per i quali, per meglio dire, non si abbiano informazioni relative all'eventuale ordinamento) bisogna esaminare tutti gli elementi, uno dopo l'altro, in sequenza
 - si parla di ***ricerca sequenziale***

```
public static int sequentialSearch(int[] a, int v)
{ // importante in Java: gestione casi degeneri!
    if (a == null) return -1; // NON TROVATO
    for (int i = 0; i < a.length; i++)
        if (a[i] == v)
            return i; // TROVATO
    return -1; // NON TROVATO
}
```

RIPASSO AUTONOMO

Ricerca sequenziale: prestazioni

- Valutiamo le prestazioni dell'algoritmo di ricerca sequenziale
 - se l'elemento cercato **non è presente** nell'array, sono **sempre** necessari **n** accessi
 - se l'elemento cercato **è presente** nell'array, il numero di accessi necessari per trovarlo dipende dalla sua posizione
 - **al massimo** servono **n** accessi
 - **in media** servono **$n/2$** accessi
- In ogni caso, quindi, le prestazioni dell'algoritmo sono sia nel caso peggiore sia nel caso medio $\Theta(n)$
- Si tratta, quindi, di un algoritmo con prestazioni **lineari** in funzione della dimensione del problema
 - Per questo motivo si parla anche di **ricerca lineare**

Ricerca per bisezione (o “binaria”)

RIPASSO AUTONOMO

Ricerca in un array ordinato

- Il problema di individuare la posizione di un elemento che contenga un particolare valore all'interno di un array può essere affrontato in modo ***più efficiente se l'array è ordinato***

- Cerchiamo il valore **12**

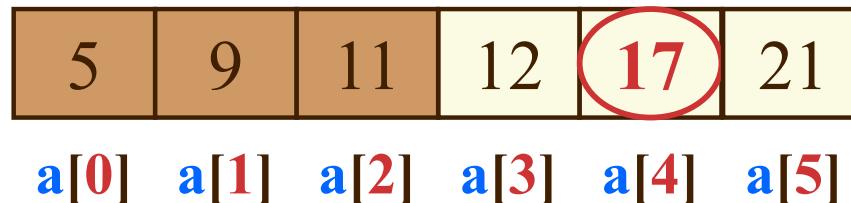
5	9	11	12	17	21
---	---	----	----	----	----

a[0] a[1] a[2] a[3] a[4] a[5]

- Confrontiamo **12** con il valore che si trova (circa) al centro dell'array, ad esempio **a[2]**, che è **11**
- Il valore che cerchiamo è maggiore di **11**
 - ***se è presente nell'array, sarà a destra di 11***

RIPASSO AUTONOMO

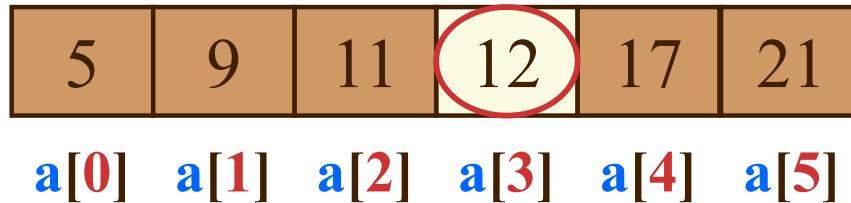
Ricerca in un array ordinato



- A questo punto dobbiamo cercare il valore **12** nel solo sotto-array che si trova a destra di **a[2]**
- Usiamo lo stesso algoritmo, confrontando **12** con il valore che si trova al centro, **a[4]**, che è **17**
- Il valore che cerchiamo è minore di **17**
 - *se è presente nell'array, sarà a sinistra di 17*

RIPASSO AUTONOMO

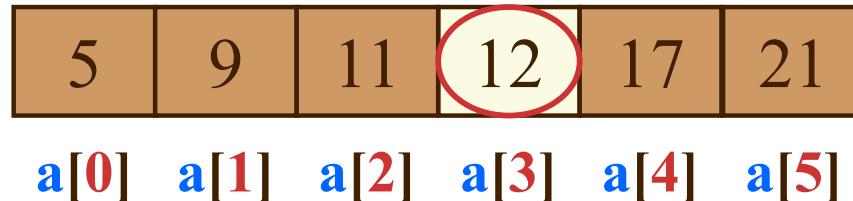
Ricerca in un array ordinato



- A questo punto dobbiamo cercare il valore **12** nel sotto-array composto dal solo elemento **a [3]**
- Usiamo lo stesso algoritmo, confrontando **12** con il valore che si trova al centro, **a [3]**, che è **12**
- Il valore che cerchiamo è uguale a **12**
 - *Il valore che cerchiamo è presente nell'array e si trova in posizione 3*

RIPASSO AUTONOMO

Ricerca in un array ordinato



- Se il valore da cercare fosse stato 13, il confronto con l'elemento **a[3]** avrebbe dato esito negativo e avremmo cercato nel sotto-array **vuoto** a destra, concludendo che il valore cercato non è presente nell'array
- Questo algoritmo si chiama **ricerca per bisezione** (o ricerca binaria), perché ad ogni passo si divide l'array **in due parti aventi circa le stesse dimensioni**
- **Funziona soltanto se l'array è ordinato!!!!!!**

RIPASSO AUTONOMO

Ricerca per bisezione

Ovviamente si può anche realizzare un algoritmo iterativo

```
public static int binarySearch(int[] a, int target)
{ if (a == null) return -1;
  return binSearch(a, 0, a.length, target);
} // come spesso si fa, un metodo ausiliario...
private static int binSearch(int[] a, int from,
                             int to, int target)
{ // from incluso, to escluso, come al solito,
  // e se sono uguali l'array ha lunghezza zero
  if (from >= to) return -1; // VUOTO, NON TROVATO
  int mid = (from + to) / 2; // CIRCA IN MEZZO
  int midVal = a[mid];
  if (midVal == target) return mid; // TROVATO
  else if (midVal < target) // CERCA A DESTRA
    return binSearch(a, mid + 1, to, target);
  else // (OPPURE!!) CERCA A SINISTRA
    return binSearch(a, from, mid, target);
} // algoritmo con ricorsione SEMPLICE in coda
```

Ricerca binaria: algoritmo

- Nella definizione degli algoritmi occorre essere MOLTO precisi (cioè "non ambigui")
- Tipica “definizione studentesca” dell’algoritmo di ricerca per bisezione
 - Guardo l’elemento centrale: se è maggiore cerco a sinistra, altrimenti cerco a destra, e così via...



Ricerca binaria: algoritmo

- **Guardo l'elemento centrale: se è maggiore cerco a sinistra, altrimenti cerco a destra, e così via...**
- Molti aspetti non sono ben specificati. Ad esempio
 - Se l'array ha dimensione pari, qual è l'**elemento centrale**? E se l'array ha lunghezza zero?
 - Cosa significa “**e così via**” ?
 - Cosa succede se a destra/sinistra non c’è niente?
 - Come faccio a capire se ho avuto successo nella ricerca oppure no?

Torneremo più avanti
sulle relazioni d'ordine



Ricerca binaria: algoritmo ricorsivo

- Ricerca di x nell'array A di dimensione n , ordinato secondo il criterio C (usando, quindi, la relazione d'ordine $<_C$)
- Algoritmo: **Cerca** x in $A = A[0] \dots A[n-1]$
 - Se $n = 0$, $x \notin A$ e fine
 - Altrimenti, sia p la “posizione centrale” in A , $p = n / 2$, intesa come divisione intera
 - Se $x = A[p]$, $x \in A$ e fine
 - Altrimenti se $x <_C A[p]$
 $B = A[0] \dots A[p-1]$
 - Altrimenti
 $B = A[p+1] \dots A[n-1]$
 - **Cerca** x in B

Notazione:
"array" A

Verificare sempre **con cura** che l'algoritmo funzioni nei **casi limite** e nei **casi degeneri**

Come si scrive un algoritmo?

- Quando si scrive un algoritmo, occorre **precisione rigorosa e dettaglio sufficiente**
 - “Regola d’oro”: **Deve essere possibile “tradurre” l’algoritmo in un linguaggio di programmazione (come Java)**
 - Senza prendere decisioni
 - Senza aggiungere/togliere nulla
 - **Senza pensare!** ☺
 - Un algoritmo ben scritto, naturalmente, può ancora essere sbagliato... queste regole non hanno NULLA a che vedere con la correttezza

Descrizione di algoritmi ricorsivi

- In particolare, porre attenzione a
 - Definizione dei casi base
 - Semplificazione del problema prima dell'invocazione ricorsiva
 - Raggiungimento di un caso base con un numero finito di passi (sempre!!!)
 - Gestione dei casi limite (es: $n = 1$)
 - Gestione dei casi degeneri (es: $n = 0$)
 - Presenza di espressioni come “e così via” oppure “puntini, puntini...”
 - Non ci devono essere!!!

Lezione 08

Ricerca binaria: prestazioni

- Valutiamo le prestazioni dell'algoritmo (*ricorsivo*) di ricerca binaria in un array ordinato
 - Si dimostra che l'algoritmo iterativo ha le stesse prestazioni
- Per cercare in un array di dimensione **n** bisogna
 - effettuare un confronto (con l'elemento centrale)
 - nel caso migliore ho finito, ma il caso migliore non interessa
 - effettuare una ricerca in un array avente una dimensione che, nel caso peggiore, è uguale a **n/2 (dettagli in seguito)**
- Quindi
$$T(n) = 1 + T(n/2)$$
- È una funzione un po' strana, “implicita”
 - È una funzione implicita di variabile **interna**, e si dice “funzione alle ricorrenze” o “funzione ricorsiva”

RIPASSO AUTONOMO

Ricerca binaria: prestazioni

- Abbiamo appena visto che, per cercare in un array di dimensione **n**, bisogna (nel caso peggiore)
 - fare un confronto (con l'elemento centrale)
 - fare una ricerca in un array di dimensione **CIRCA** uguale a **n/2**
- Più precisamente, ricordando che la somma delle due porzioni è sempre uguale a **n-1** (perché l'elemento “centrale” non ne fa parte), si osserva che
 - se **n** è dispari, **n-1** è pari e, quindi, divisibile per due: le due porzioni (soltanto una delle quali andrà esaminata) hanno la stessa dimensione, uguale a **(n-1)/2**
 - se **n** è pari, **n-1** è dispari e le due porzioni hanno dimensioni diverse: **(n-2)/2** la più piccola, **n/2** la più grande
- **Nel caso peggiore**, quindi, la dimensione del sottoproblema da risolvere ricorsivamente è proprio **n/2**

Attenzione! In generale, $T(n/2) \neq T(n)/2$

Ricerca binaria: prestazioni

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

- La soluzione di questa equazione per ottenere $T(n)$ in funzione di n non è semplice

- È un problema di "matematica discreta"

- Si può però agevolmente **verificare**, per sostituzione, che questa sia **una** soluzione

$$T(n) = \log_2 n$$



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

IMPORTANTE:
si dimostra
(ma non è
facile...) che tali
prestazioni
valgono anche nel
caso medio

- Le prestazioni dell'algoritmo di ricerca binaria sono *assai migliori di quelle della ricerca lineare*

Ricerca binaria: prestazioni

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

Verifichiamo la soluzione $T(n) = \log_2 n$

Conseguentemente $T(n/2) = \log_2(n/2)$

Sostituendo nell'equazione, si ottiene la verifica:

$$\begin{aligned}\log_2 n &= \log_2(n/2) + 1 \\&= (\log_2 n - \log_2 2) + 1 \\&= \log_2 n - 1 + 1 \\&= \log_2 n\end{aligned}$$

Esistono altre soluzioni, con k reale: $T(n) = k + \log_2 n$

Ma è sempre: $T(n) \in \Theta(\log n)$

Perché la ricerca binaria è logaritmica?

- Il caso peggiore si ha quando l'elemento cercato (in un array di dimensione n) **non** è presente
- Ogni passo ricorsivo richiede l'ispezione di una cella e un confronto, quindi viene eseguito in un tempo che non dipende da n (cioè $\Theta(1)$): basta contare il numero di invocazioni ricorsive, m , e **il tempo sarà $T(n) \in O(m)$**
- Ad ogni invocazione la dimensione del problema si **dimezza** (in realtà, diventa un po' inferiore alla metà: diventa $\lceil (n - 1)/2 \rceil$)
 - Quindi, dopo m passi la dimensione del problema è superiormente limitata da $n / 2^m$
 - La ricorsione si ferma quando il problema ha dimensione zero o uno: $n / 2^m \leq 1 \Rightarrow 2^m \geq n \Rightarrow m \geq \log_2 n$
 - Il minimo valore di m che soddisfa questa relazione è $m = \lceil \log_2 n \rceil \Rightarrow m \in O(\log n) \Rightarrow T(n) \in O(\log n)$

Ricerca binaria: pre-condizioni

- Questo algoritmo ha la “scomoda” caratteristica di non poter verificare le proprie pre-condizioni
- **La pre-condizione per poter effettuare una ricerca per bisezione è che l'array sia ordinato**
- Verificare se un array è ordinato richiede un tempo lineare in funzione della dimensione dell'array
 - L'algoritmo di “verifica di ordinamento” è $\Theta(n)$
 - Devo almeno leggere tutti i valori...
- Se il metodo verificasse l'ordinamento dell'array prima di effettuarvi una ricerca, le sue prestazioni sarebbero $\Theta(n + \log n) \equiv \Theta(n)$ e l'algoritmo sarebbe equivalente alla ricerca lineare, diventando in realtà anche (un po') più lento di essa

RIPASSO AUTONOMO

Ancora su O-grande

- È ovvio che $\Theta(n + \log n) \equiv \Theta(n)$?
- Analizziamo $O(n + \log n) \equiv O(n)$, per Ω è analogo
- Errore tipico
 - Com'è possibile che sia $(n + \log n) \leq n$ per n abbastanza grande???
- Ma la definizione di O-grande
NON DICE QUESTO...
- Vediamo...

RIPASSO AUTONOMO

Aritmetica di O-grande

- $O(n)$ è l'insieme delle funzioni $f(n)$ per le quali
 $\exists c > 0, k > 0 : f(n) \leq cn \quad \forall n \geq k$
- $O(n + \log n)$ è l'insieme delle funzioni $f(n)$ per le quali $\exists c' > 0, k' > 0 : f(n) \leq c'(n + \log n) \quad \forall n \geq k'$
- I due insiemi coincidono, cioè **$O(n + \log n) \equiv O(n)$** , se (ad esempio) l'appartenenza a uno dei due implica l'appartenenza all'altro, e viceversa
- Il fatto che $f(n) \in O(n) \Rightarrow f(n) \in O(n + \log n)$ è “ovvio”, basta usare $c' = c$ e $k' = k$, perché il logaritmo è una funzione non negativa $\forall n (\geq 1)$

RIPASSO AUTONOMO

Aritmetica di O-grande

- Vediamo se $f(n) \in O(n + \log n) \Rightarrow f(n) \in O(n)$
Se $\exists c' > 0$ e $k' > 0$: $f(n) \leq c'(n + \log n)$ $\forall n \geq k'$
allora $\forall n \geq k'$

$$f(n) \leq c'(n + \log n) < c'(n + n) = c' 2n = c n$$
$$\Rightarrow f(n) \in O(n), \text{ con } k = k' \text{ e } c = 2c'$$

- In generale, O-grande gode di una proprietà di “assorbimento”... tutte le componenti che crescono più lentamente della componente dominante sono ininfluenti... analogamente a quanto visto per i polinomi

Stessa proprietà per Omega e Theta

NON è una definizione...

Aritmetica di O-grande

- A volte, invece di $O(n + \log n) \equiv O(n)$, si trova scritto

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

- Con la matematica ci prendiamo “qualche libertà”, ma dobbiamo essere sempre ben consapevoli di quello che stiamo facendo...

- Abbiamo detto che

$O(g(n))$ è un insieme di funzioni...

allora che significato ha la **somma di insiemi**?

Non ha significato... la nostra affermazione precedente andrebbe scritta così...

prosegue

Aritmetica di O-grande

- Il tempo di esecuzione dell'algoritmo è la somma di due componenti, $T(n) = F1(n) + F2(n)$
- $F1(n) \in O(n)$ è il tempo richiesto per la verifica di ordinamento dell'array
- $F2(n) \in O(\log n)$ è il tempo richiesto per la ricerca per bisezione nell'array ordinato
- Si dimostra agevolmente che
 $f(n) \in O(g(n))$ e $h(n) \in O(k(n))$
 $\Rightarrow f(n) + h(n) \in O(g(n) + k(n))$

prosegue

Aritmetica di O-grande

$f(n) \in O(g(n))$ e $h(n) \in O(k(n))$

$\Rightarrow f(n) + h(n) \in O(g(n) + k(n))$

□ Quindi, nel nostro caso

$T(n) = F1(n) + F2(n) \in \textcolor{red}{O(n + \log n)}$

□ Spesso scriveremo $T(n) = O(n) + O(\log n)$ per evidenziare il contributo delle due porzioni di un algoritmo, ma non ha molto senso....

□ **In pratica, a volte si usa O-grande come se fosse un operatore lineare!** Ma NON è un operatore, è semplicemente una notazione...

RIPASSO AUTONOMO

Ricerca iterativa per bisezione

```
public static int iterBinSearch(int[] a, int target)
{  if (a == null) return -1;
   int from = 0;
   int to = a.length;
   while (from < to)
   {  int mid = (from + to) / 2;
      int midVal = a[mid];
      if (midVal == target)
         return mid;
      else if (midVal < target)
         from = mid + 1;
      else
         to = mid;
   }
   return -1;
}
```

È difficile contare il numero di **accessi**, perché non è facile capire il numero di iterazioni eseguite dal **ciclo**, ma si può dimostrare che le prestazioni sono ancora logaritmiche

RIPASSO AUTONOMO

Prestazioni temporali
asintotiche di altri
algoritmi operanti su array

RIPASSO AUTONOMO

Clonazione di un array

- Per creare **una copia identica** di un array contenente **n** elementi, è necessario
 - creare un nuovo array di **n** elementi (che, in Java, vengono inizializzati al valore predefinito per quel tipo di dato, effettuando **n** accessi in scrittura)
 - leggere le **n** celle dell'array originario
 - scrivere nel nuovo array il contenuto di tali **n** celle
- Servono **3n** accessi
 - **2n** se la creazione dell'array non prevede la sua inizializzazione (dipende dal linguaggio di programmazione, cfr. Java e C)
- Tale operazione è, quindi, **$\Theta(n)$**

RIPASSO AUTONOMO

Ridimensionamento di un array

- Per *ridimensionare* un array contenente **n** elementi, **raddoppiandone** la dimensione, è necessario
 - creare un array di **2n** elementi, che vengono inizializzati al valore predefinito per quel tipo di dato effettuando **2n** accessi in scrittura
 - leggere le **n** celle dell'array originario
 - scrivere il contenuto di tali **n** celle nelle prime **n** celle del nuovo array
- Servono **$2n+2n=4n$** accessi
- Tale operazione è, quindi, ancora **$\Theta(n)$**

RIPASSO AUTONOMO

Ridimensionamento di un array

- Se, invece di raddoppiare la dimensione, la si moltiplica per un fattore **k**, occorrono $\mathbf{kn+2n= (k+2) n}$ accessi e l'operazione rimane $\Theta(n)$
 - cambia soltanto il fattore moltiplicativo del numero di accessi, che, però, sappiamo essere ininfluente

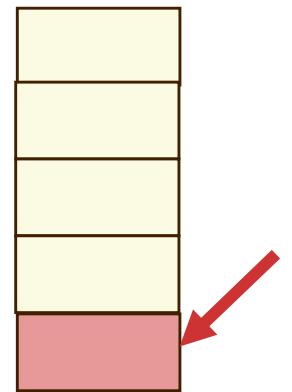
RIPASSO AUTONOMO

Ricerca di minimo/massimo

- Per effettuare la ricerca del valore minimo/massimo (o della sua posizione) all'interno di un array contenente **n** elementi
 - è *necessario e sufficiente* leggere una sola volta tutti gli elementi
- Il numero di accessi ad elementi dell'array è quindi uguale a **n** e questi algoritmi sono tutti $\Theta(n)$
- Questi due (quattro) algoritmi **non hanno caso migliore né caso peggiore**, perché devono sempre e comunque leggere tutti gli elementi dell'array, indipendentemente dai loro valori e/o posizioni

RIPASSO AUTONOMO

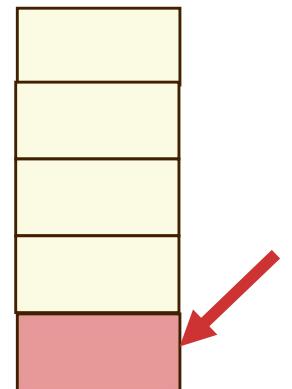
Inserimento di un elemento



- Se non interessa la posizione relativa tra gli elementi presenti nell'array e c'è spazio libero, l'inserimento di un elemento richiede un solo accesso in scrittura
 - $T(n) = 1$
 $\in O(1) \equiv \Theta(1)$
 - Ogni volta che il numero operazioni elementari NON dipende dalla dimensione del problema, si dice che l'algoritmo viene eseguito in **tempo costante** e l'algoritmo è **O(1)** (o $\Theta(1)$) indipendentemente dal valore della costante stessa
 - Qualunque algoritmo è $\Omega(1)$... quindi si può dire, indifferentemente, $O(1)$ oppure $\Theta(1)$
- Se non c'è spazio...

RIPASSO AUTONOMO

Inserimento di un elemento



- Se non interessa la posizione relativa tra gli elementi presenti nell'array **ma non c'è spazio**, l'inserimento di un elemento richiede il ridimensionamento dell'array, quindi l'operazione è

$$T(n) = \Theta(n) + \Theta(1) = \Theta(n)$$

- Se l'array è usato in modalità “**sempre pieno**” (cioè con dimensione fisica sempre uguale alla dimensione logica), il ridimensionamento avviene ad ogni inserimento, per cui l'operazione è sempre **$\Theta(n)$**
- Se l'array è usato in modalità “**riempito solo in parte**”, il ridimensionamento è necessario solo “ogni tanto”...
 - Quando serve il ridimensionamento (caso peggiore), l'inserimento è **$\Theta(n)$**
 - Quando non serve il ridimensionamento (caso migliore), l'inserimento è **$\Theta(1)$**
 - **E in media? Serve una analisi ammortizzata**

Analisi ammortizzata

- Dobbiamo calcolare il tempo **medio** speso per eseguire più operazioni di inserimento in fondo a un array riempito solo in parte
- Non è semplice identificare l'insieme di operazioni su cui fare la media: **usiamo un esempio**
 - Partiamo dalla situazione in cui l'array ha dimensione **n** ed è pieno
 - Facciamo **n** inserimenti consecutivi (senza rimozioni)
 - Il primo inserimento richiede un ridimensionamento, che porta la dimensione a **mn** (con $m \geq 2$) e crea spazio anche per (almeno) tutti gli **n-1** inserimenti successivi, che, quindi, non richiedono ridimensionamenti
 - **Calcoliamo il tempo medio richiesto per uno di questi n inserimenti**

Analisi ammortizzata

- Dobbiamo calcolare il tempo **medio** per eseguire **n** operazioni:
 - **n-1** operazioni, ciascuna delle quali richiede **1** accesso
 - **una** operazione che richiede **(m+2) n + 1** accessi
- $$\begin{aligned} T_{\text{medio}}(n) &= [(m+2)n + 1 + (n - 1)]/n \\ &= [(m+2)n + n]/n = (m+3)n/n = \\ &= m+3 \in \Theta(1) \end{aligned}$$
- *Distribuendo* in parti uguali tra tutte le operazioni di inserimento il tempo speso per l'unico ridimensionamento, si ottiene quindi **un tempo medio costante per la singola operazione, $\Theta(1)$** , che non dipende dalle dimensioni dell'array

- **Distribuendo** in parti uguali a tutte le operazioni di inserimento il tempo speso per l'unico ridimensionamento, si ottiene quindi un tempo medio costante, $\Theta(1)$, che non dipende dalle dimensioni dell'array
- Se tra le operazioni di inserimento vengono eseguite anche operazioni di rimozione la situazione **migliora**, perché le rimozioni riducono la necessità di ridimensionare: **abbiamo, quindi, fatto un'analisi media di caso peggiore**
 - ma un algoritmo non può avere prestazioni migliori di $\Theta(1)$... quindi siamo a posto
- Si può dimostrare che questo risultato è vero quando il numero di operazioni da eseguire è paragonabile alla dimensione dell'array oppure è più elevato
 - Se il numero di inserimenti è, invece, piccolo rispetto alla dimensione dell'array, si può dimostrare che il risultato è vero **statisticamente** ma, ovviamente, non può esserlo puntualmente (se faccio un solo inserimento e non ho fortuna...)

Analisi ammortizzata

- La stessa dimostrazione evidenzia che le prestazioni dell'inserimento con eventuale ridimensionamento rimangono, **in media, $\Theta(1)$** per qualsiasi costante **moltiplicativa** (maggiore di uno) usata per calcolare la nuova dimensione
- Se, invece, si usa una costante **additiva** (cioè la dimensione passa da **n** a **n+m**) si osserva che, su **n** operazioni di inserimento
 - quelle “lente” ($\Theta(n)$) sono **n/m** (perché ogni **m** inserimenti occorre un ridimensionamento)
 - quelle “veloci” ($\Theta(1)$) sono ovviamente **(n - n/m)**
- Facendo la media su n operazioni consecutive si ottiene un **tempo medio $\Theta(n)$** [fare i calcoli]
- **Proprio per questo motivo si usa SEMPRE un ridimensionamento con moltiplicazione della dimensione per un valore costante**

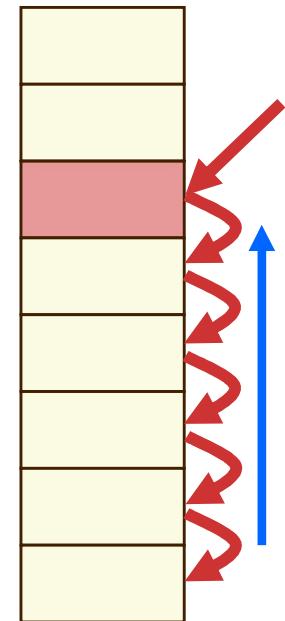
Analisi ammortizzata

- Ridimensionamento con costante *moltiplicativa*
⇒ inserimento **in media** $\Theta(1)$
- Ridimensionamento con costante *additiva*
⇒ inserimento **in media** $\Theta(n)$
- Il risultato era “intuibile”: aggiungere **k** celle, per **n** molto grande (al limite, tendente all’infinito...), equivale ad aggiungere una sola cella, cioè a ridimensionare ogni volta
 - Cioè, lo spazio aggiunto da un ridimensionamento con costante additiva è, in percentuale, sempre minore al crescere della dimensione dell’array: **al limite**, è come aggiungere una sola cella
 - Raddoppiando, invece, aggiungo sempre il 100% (oppure, se moltiplico per k , aggiungo il $(k - 1) \cdot 100\%$)

RIPASSO AUTONOMO

Inserimento di un elemento

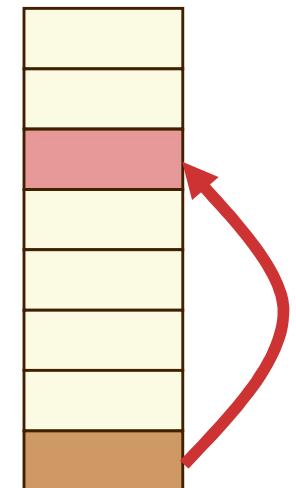
- Se, invece, la posizione relativa tra gli elementi presenti nell'array deve essere rispettata, il numero di elementi da spostare è, **in media**, $n/2$ (dipende da dove inserisco)
- Lo spostamento di un elemento richiede due accessi, per cui l'algoritmo è, in media, $\Theta(n)$
 - L'eventuale ridimensionamento, che sarebbe $\Theta(n)$, non modifica le prestazioni asintotiche dell'algoritmo
- Nel caso peggiore, il numero di elementi da spostare è proprio **n** e l'algoritmo rimane, evidentemente, $\Theta(n)$
- **Naturalmente esiste un “caso migliore” (a volte utile...) con prestazioni $\Theta(1)$: quando si inserisce in fondo a un array riempito solo in parte in cui ci sia spazio**
 - Se **questo** avviene ripetutamente, prima o poi si dovrà ridimensionare l'array e si ricade nel caso precedente: si possono avere inserimenti mediamente $\Theta(1)$, con analisi ammortizzata



RIPASSO AUTONOMO

Rimozione di un elemento

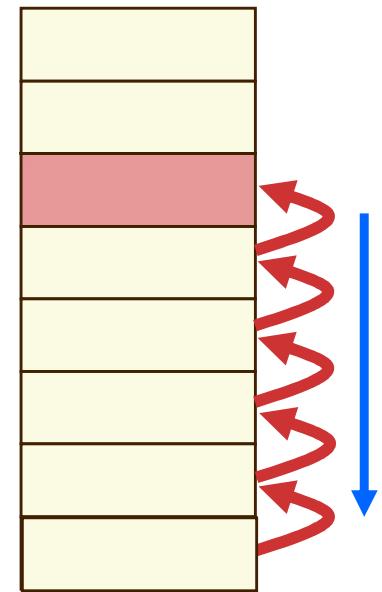
- Se non interessa la posizione relativa tra gli elementi presenti nell'array, la rimozione di un elemento richiede **due soli** accessi, uno in lettura e uno in scrittura
 - $T(n) = 2 \in \Theta(1)$
- Se l'array è usato in modalità “sempre pieno”, occorre poi ridimensionarlo (riducendone la dimensione di un'unità) e l'operazione diventa, nel suo complesso, **$\Theta(n)$**
 - Quest'ultimo problema non esiste per array riempiti solo in parte



RIPASSO AUTONOMO

Rimozione di un elemento

- Se, invece, la posizione relativa tra gli elementi presenti nell'array deve essere rispettata, il numero di elementi da spostare è, **in media, $n/2$** (dipende da quale rimuovo)
- Lo spostamento di un elemento richiede due accessi, per cui l'algoritmo è, in media, $\Theta(n)$
 - L'eventuale ridimensionamento, che sarebbe $\Theta(n)$, non modifica le prestazioni asintotiche dell'algoritmo
- Nel caso peggiore, il numero di elementi da spostare è **$n-1$** e l'algoritmo rimane, evidentemente, $\Theta(n)$
- Naturalmente esiste un “caso migliore” (a volte utile...) con prestazioni $\Theta(1)$: quando **si rimuove l'elemento che si trova in fondo a un array riempito solo in parte**



Lezione 09

Notazioni asintotiche: matematica vs. informatica

- Attenzione: il campo di utilizzo delle notazioni asintotiche è molto più ampio di quello che usiamo in informatica
- **Noi imponiamo "vincoli" sulla natura della funzione $g(n)$**
 - Ci interessano $g(n)$ "semplici" (monomi, logaritmi, esponenziali) perché vogliamo usare le notazioni asintotiche per **CONFRONTARE algoritmi sulla base dell'andamento delle loro prestazioni**
- Senza tali vincoli, alcune delle considerazioni che facciamo diventano poco interessanti (per noi...)
 - Es. **ogni funzione è O-grande, Omega e Theta di se stessa!**
 - Discende direttamente dalla definizione, con $c = 1$ e $k = 1$
 - Quindi una funzione è O-grande e Omega più stringente di se stessa... e questi coincidono con Theta...
 - E così via

O-grande più stringente e Theta

- $f(n) \in \Theta(g(n)) \Rightarrow O(g(n))$ è il più stringente O-grande di $f(n)$
[e $\Omega(g(n))$ è il più stringente Ω di $f(n)$]
 - Il viceversa non è vero: se $O(g(n))$ è il più stringente O-grande di $f(n)$, questo NON implica che $f(n) \in \Theta(g(n))$
- **Con i vincoli che abbiamo imposto su $g(n)$, non tutte le funzioni appartengono a un insieme Θ !!**
- Funzioni che non appartengono a nessun Θ sono un po' "strane"...
 - Sono, ad esempio, funzioni che, usando $g(n)$ "semplici", hanno **O-grande più stringente e Ω più stringente che non coincidono**
- Es. $f(n) = 1 + n (1 + \sin(n\pi/2))$
 - $f(n) \in O(n)$ stringente
 - $f(n) \notin \Omega(n)$ perché "ogni tanto" $f(n) = 1$ (quando $n = 3 + 4k$ con k intero), quindi $f(n) \in \Omega(1)$ stringente

Si potrebbe dire molto di più ma "ci accontentiamo"

RIPASSO AUTONOMO

Fattore di occupazione

- Quando si rimuovono ripetutamente elementi da un array riempito solo in parte senza ridurre la dimensione dell'array, diminuisce il suo **fattore di occupazione**, dando luogo a uno spreco di memoria
- Ovviamente la riduzione di dimensione ha un costo $\Theta(n)$, ma a volte può essere necessario cercare un compromesso tra prestazioni temporali ed efficienza nell'occupazione della memoria
- Si può dimostrare, analogamente a quanto fatto con l'aumento di dimensione, che la riduzione di dimensione per un fattore moltiplicativo porta a operazioni di rimozione $\Theta(1)$ in media
- **Si potrebbe pensare di dimezzare la dimensione ogni volta che il fattore di occupazione diventa minore del 50%**
- Bisogna, però, fare MOLTA attenzione a **coordinare il fattore moltiplicativo della riduzione di dimensione e il fattore moltiplicativo dell'aumento di dimensione**

RIPASSO AUTONOMO

Fattore di occupazione

- Bisogna **coordinare il fattore moltiplicativo della riduzione di dimensione e il fattore moltiplicativo dell'aumento di dimensione**
- Se raddoppiamo quando è pieno e dimezziamo quando è occupato al 50%, si può avere questa sequenza di eventi
 - Dimezzamento $\Theta(n)$ \Rightarrow Occupazione 100%
 - **Inserimento** \Rightarrow Necessario raddoppio $\Theta(n)$ \Rightarrow Occupazione poco più di 50%
 - **Rimozione** \Rightarrow Occupazione scende al 50% \Rightarrow Dimezzamento $\Theta(n)$ \Rightarrow Occupazione 100%
 - **Inserimento** \Rightarrow Necessario raddoppio $\Theta(n)$ \Rightarrow Occupazione poco più di 50%
 - **Rimozione** \Rightarrow Occupazione scende al 50% \Rightarrow Dimezzamento $\Theta(n)$ \Rightarrow Occupazione 100%
 - **E così via...**
- È una sequenza di operazioni $\Theta(n)$, quindi ciascuna operazione è **$\Theta(n)$ anche in media**

RIPASSO AUTONOMO

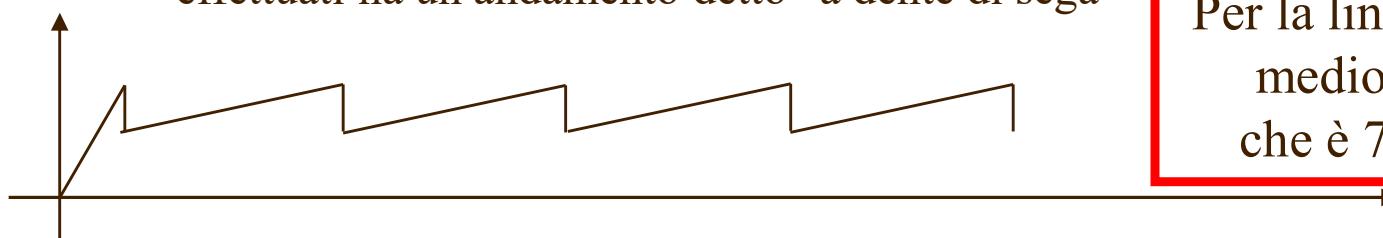
Fattore di occupazione

- Bisogna coordinare il fattore moltiplicativo della riduzione di dimensione e il fattore moltiplicativo dell'aumento di dimensione
 - Di solito si raddoppia quando è pieno e si divide la dimensione per tre quando è occupato per un terzo (o per quattro quando è occupato per un quarto)
- In questo modo, dopo un raddoppio di dimensione l'occupazione è, come prima, poco più del 50%
- Se avviene subito una rimozione, l'occupazione scende, ma non arriva subito a un terzo o a un quarto

RIPASSO AUTONOMO

Fattore di occupazione medio?

- Non è per nulla semplice calcolare il fattore di occupazione medio di un array riempito solo in parte a cui vengano applicati entrambi i criteri di ridimensionamento che abbiamo visto
 - Dipende dalle operazioni che si compiono
- Se si fanno **soltanto inserimenti**, con aumento di dimensione di un fattore moltiplicativo $k > 1$
 - Trascuriamo il transitorio iniziale, prima del primo aumento di dimensione, quando il fattore di occupazione varia linearmente da 0 a 1
 - Il fattore di occupazione minimo (subito dopo un ingrandimento) è $1/k$
 - Il fattore di occupazione massimo è 1
 - Il fattore di occupazione aumenta linearmente dal minimo al massimo, per poi tornare bruscamente al minimo e ripetere la sequenza
 - Il grafico del fattore di occupazione in funzione del numero di inserimenti effettuati ha un andamento detto “a dente di sega”



Per la linearità, il fattore medio è $(k + 1)/2k$ che è 75% per $k = 2$

RIPASSO AUTONOMO

Fattore di occupazione medio?

- Se si fanno **soltanto rimozioni**, con diminuzione di dimensione di un fattore moltiplicativo $k > 1$
 - Trascuriamo il transitorio iniziale e iniziamo l'analisi subito dopo una riduzione di dimensione, per cui il fattore di occupazione è massimo, uguale a 1
 - Il fattore di occupazione minimo (subito prima di una riduzione di dimensione) è $1 / k$
 - Il fattore di occupazione diminuisce linearmente dal massimo al minimo, per poi tornare bruscamente al massimo e ripetere la sequenza
 - Il grafico del fattore di occupazione in funzione del numero di rimozioni effettuate ha ancora un andamento “a dente di sega”

Per la linearità, il fattore medio è ancora
 $(k + 1)/2k$ che è 67% per $k = 3$

RIPASSO AUTONOMO

Fattore di occupazione medio?

- Se si fanno **inserimenti e rimozioni**, possiamo soltanto dire, in generale, che il fattore di occupazione varia (ovviamente) tra il suo valore massimo, che è 1, e il suo valore minimo (che è il minimo tra i due valori di $1/k$, solitamente quello di riduzione)
- È possibile che rimanga vicino al minimo
 - Sono al minimo (cioè se facessi una rimozione, dovrei ridurre la dimensione), ma inserisco, rimuovo, inserisco, rimuovo,, ...
- È possibile che rimanga vicino al massimo
 - Sono al 100% (cioè se facessi un inserimento, dovrei aumentare la dimensione), ma rimuovo, inserisco, rimuovo, inserisco, ...
- È possibile che... si comporti in qualsiasi modo

RIPASSO AUTONOMO

Osservazione: $\Theta(1)$ in media...

- Attenzione che le ricadute pratiche della scelta di un algoritmo **$\Theta(1)$ in media** rispetto a un algoritmo **$\Theta(1)$** possono essere **pesanti** se c'è interazione con un utente umano o, in generale, con un processo che richieda tempi certi in modo puntuale e non solo in media
 - Gestione della sicurezza in una centrale nucleare...
- Esempio. Sono in attesa all'ufficio postale e le operazioni richiedono mediamente qualche minuto, ma, quando è il mio turno, sono il milionesimo cliente e l'impiegato vince una settimana di ferie con decorrenza istantanea!
 - Alla fine dell'anno il tempo medio di servizio di un cliente non sarà modificato in modo sensibile dalla settimana di ferie (che viene “ammortizzata” tra tutti i clienti...)
 - Io però non sarò felice di dover aspettare il ritorno dalle Maldive...



Modello computazionale RAM

- Integriamo il modello che utilizziamo con due ipotesi semplificative:
 - Operazioni logiche e aritmetiche elementari **eseguite in un tempo $\Theta(1)$, indipendente dai valori elaborati**
 - È realistico finché i valori trovano posto in una singola cella di memoria, non è vero con valori appartenenti a un insieme infinito, ma... in informatica non esistono insiemi infiniti!
 - **Occupazione di memoria $\Theta(1)$ per qualsiasi valore numerico**
 - Come sopra...
 - Trasferimento di dati tra CPU e memoria nelle due possibili direzioni
 - "scrittura" in memoria (assegnamento a una variabile)
 - "lettura" dalla memoria (utilizzo in una espressione)
 - Indicizzazione in array in un tempo indipendente dall'indice
 - **Cioè "accesso casuale" agli elementi contenuti in un array**
 - Esecuzione condizionale per effetto di confronti tra dati
 - Quindi anche iterazioni
 - Invocazione di un metodo con eventuale restituzione di un valore (rendendo così possibile la ricorsione)

Molto importante!

Gli algoritmi ricorsivi e lo spazio

- Nel corso di Fondamenti di Informatica solitamente si trascura il fatto che **gli algoritmi ricorsivi**, per il proprio funzionamento, **necessitano di spazio aggiuntivo "implicito"**, che non appare evidente nella descrizione dell'algoritmo
- Ogni invocazione di un metodo, infatti, richiede spazio nel *runtime stack*, cioè nella zona di memoria dedicata al "congelamento" dei metodi invocanti, in attesa che termini l'esecuzione dei metodi invocati
 - È richiesta una quantità di memoria fissa per ogni invocazione, a cui si aggiunge la memoria richiesta per le variabili locali, di cui già teniamo sempre conto
 - In pratica, **bisogna valutare la massima "profondità di ricorsione"**, cioè quante invocazioni del metodo ricorsivo possono essere **contemporaneamente** presenti nel runtime stack, in funzione della dimensione del problema in esame
 - Esempio: **l'algoritmo ricorsivo di ricerca binaria in un array di dimensione n ha una profondità di ricorsione massima uguale a $\log_2 n$, quindi richiede uno spazio aggiuntivo $O(\log n)$.** Questo sarebbe vero anche se **NON usasse variabili locali** (c'è comunque l'indirizzo di ritorno e altri dettagli).

Algoritmi ottimi

Molto importante!

- Dato un problema computazionale e trovato un algoritmo che lo risolve, cosa significa dire che l'algoritmo è **ottimo** dal punto di vista delle prestazioni **asintotiche** (temporali e/o spaziali) ?
- **Ottimo** in italiano significa "**non si può far meglio**"...
- Per dimostrare che un algoritmo è (**asintoticamente**) **ottimo** occorre innanzitutto trovare il limite inferiore, $\mathbf{h}(n)$, per le prestazioni di **qualsiasi** algoritmo che possa risolvere il problema (**compresi eventuali algoritmi che ancora non siano noti...**), cioè bisogna dimostrare che qualsiasi algoritmo che risolve il problema è $\Omega(\mathbf{h}(n))$, **usando l'omega più stringente**
 - Esempio: calcolare il valore minimo in un insieme di n elementi è $\Omega(n)$, perché non si può calcolare il minimo senza esaminare almeno una volta ciascuno degli n elementi dell'insieme (si dimostra banalmente per assurdo)
- Se l'algoritmo in esame è $O(\mathbf{h}(n))$, allora è **asintoticamente ottimo**!
E, conseguentemente, tale algoritmo è $\Theta(h(n))$, senza doverne calcolare un Ω
- A volte trovare un limite inferiore per le prestazioni di **qualsiasi** algoritmo che possa risolvere un dato problema è "difficile", ma è importante perché consente di evitare ulteriori ricerche di algoritmi "migliori", se questi non possono esistere

Algoritmi deterministici e non

- **Un problema computazionale può ammettere più soluzioni per uno stesso esemplare del problema stesso**
- In generale, si chiede a un algoritmo di **calcolare soltanto una delle soluzioni possibili**, a meno che non sia esplicitamente specificato che vanno trovate tutte
 - Es. in un array di numeri, trovare LE POSIZIONI degli elementi che hanno valore massimo (in generale più d'una, in presenza di elementi duplicati nell'array)
 - In questo caso non parliamo di "più soluzioni" ma di un'unica soluzione che comprende tutte le posizioni descritte
- **In queste situazioni** esistono due categorie di algoritmi
 - Gli **algoritmi deterministic**i, chiamati ripetutamente a risolvere uno specifico esemplare del problema, calcolano **sempre la stessa soluzione** tra tutte quelle possibili
 - Gli **algoritmi non deterministic**i possono calcolare soluzioni diverse di volta in volta e, di solito, sfruttano questa "libertà" per migliorare le prestazioni
 - Sono anche detti algoritmi "randomizzati" (*randomized*), perché fanno scelte casuali
 - In questo corso non ci occupiamo di algoritmi non deterministic (con qualche eccezione), ma questi hanno un'importanza rilevante nella soluzione di problemi particolarmente onerosi dal punto di vista computazionale

RIPASSO AUTONOMO

Programmazione per tipi generici in Java: è sufficiente quello che trovate nelle slide iniziali di «ripasso autonomo» ma chi vuole approfondire ovviamente può farlo

Lezione 10

Esercizio 12

Esercizio: Numero mancante

- L'array A di dimensione $n - 1$ contiene numeri interi compresi nell'intervallo $[0, n - 1]$, senza ripetizioni. Nell'array, quindi, manca **uno e uno solo** dei numeri dell'intervallo: progettare un algoritmo $O(n)$ che lo identifichi, usando, oltre all'array, uno spazio di memorizzazione $O(1)$

Cos'è uno “spazio di memorizzazione **$O(1)$** ”?

È **uno spazio** di dimensioni **FISSE**
(cioè con **dimensione che NON dipende da n**)
che consenta di risolvere il problema per ogni n .

Possono essere anche 2000 variabili... ☺,
ma **NON** un numero di variabili uguale a $0.05n$...

Se avessimo tempo?

- L'array A di dimensione $n - 1$ contiene numeri interi compresi nell'intervallo $[0, n - 1]$, senza ripetizioni. Nell'array, quindi, manca uno e uno solo dei numeri dell'intervallo: progettare un algoritmo $O(n)$ che lo identifichi, usando, oltre all'array, uno spazio di memorizzazione $O(1)$.

- **Prima idea: non consideriamo il vincolo sul tempo**
e proviamo un approccio "brutale" (*brute-force approach*)

- `for (i = 0; i <= n - 1; i++)` // tutti i numeri possibili
 - Ricerca (sequenziale) di i in A
 - Se i non è in A , fine: il valore mancante è i
 - Nel caso peggiore e medio, l'algoritmo è $\Theta(n^2)$
 - Sarà brutale, ma intanto funziona...
spesso si comincia così, a volte può essere utile per capire dove si può migliorare...

Se avessimo spazio?

- L'array A di dimensione $n - 1$ contiene numeri interi compresi nell'intervallo $[0, n - 1]$, senza ripetizioni. Nell'array, quindi, manca uno e uno solo dei numeri dell'intervallo: progettare un algoritmo $O(n)$ che lo identifichi, usando, oltre all'array, uno spazio di memorizzazione $O(1)$.
- **Seconda idea: non consideriamo il vincolo sullo spazio. Se avessimo a disposizione uno spazio $\Theta(n)$, potremmo, in un tempo $\Theta(n)$:**

- Creare un array B di valori booleani di dimensione n , con tutte le celle inizializzate a **false**
- `for (i = 0; i < n - 1; i++)`
 - $B[A[i]] = \text{true}$
- `for (i = 0; i < n; i++)`
 - `if ($B[i] == \text{false}$) return i` // i è il numero mancante in A

Ignorare uno dei vincoli può essere utile per trovare un'idea da migliorare... anche se a volte, invece, non porta a niente

Un'altra idea

- L'array A di dimensione $n - 1$ contiene numeri interi compresi nell'intervallo $[0, n - 1]$, senza ripetizioni. Nell'array, quindi, manca uno e uno solo dei numeri dell'intervallo: progettare un algoritmo $O(n)$ che lo identifichi, usando, oltre all'array, uno spazio di memorizzazione $O(1)$.
 - Ordiniamo l'array
 - Esaminiamo ciascun numero e verifichiamo se è uguale all'indice attuale durante la scansione, oppure, più in generale, se il successivo è il consecutivo di quello in esame
 - Questa parte dell'algoritmo è $O(n)$
 - Purtroppo l'algoritmo è $\Theta(n \log n)$, per via dell'ordinamento
 - **Vedremo** che dati di questo tipo (cioè numeri interi) si possono anche ordinare in un tempo $\Theta(n)$, ma non “sul posto”, serve uno spazio aggiuntivo non $O(1)$ (cfr. bucket sort, vedremo)

Soluzione

- L’array A di dimensione $n - 1$ contiene numeri interi compresi nell’intervallo $[0, n - 1]$, senza ripetizioni. Nell’array, quindi, manca **uno e uno solo** dei numeri dell’intervallo: progettare un algoritmo $O(n)$ che lo identifichi, usando, oltre all’array, uno spazio di memorizzazione $O(1)$.
 1. Calcoliamo la somma S_1 dei numeri interi nell’intervallo $[0, n - 1]$: per la formula di Gauss, $S_1 = n(n - 1)/2$
 2. Calcoliamo la somma S_2 dei numeri contenuti nell’array A
 3. Il numero mancante in A è il risultato della sottrazione tra le due quantità precedenti, $i = S_1 - S_2$- L’algoritmo è $\Theta(n)$ (per il passo 2, gli altri passi sono $\Theta(1)$), quindi è anche $O(n)$, come richiesto, e usa uno spazio di memorizzazione $O(1)$

Soluzione

- Naturalmente quanto appena concluso è vero nell'**ipotesi semplificativa**, che solitamente riteniamo vera in questo corso (salvo esplicite indicazioni contrarie), che:
 - Ciascuna operazione aritmetica venga eseguita in un tempo costante, cioè indipendente dai valori dei suoi operandi e del risultato (anche se, ovviamente, operazioni diverse possono richiedere tempi diversi, ma comunque indipendenti dai valori degli operandi)
 - Qualsiasi valore numerico occupi uno spazio di memoria costante, cioè con una dimensione che sia indipendente dal valore stesso

Esercizio 13

Esercizio

- Dato un array L contenente n numeri interi, progettare un algoritmo che, in un tempo $O(n)$, sia in grado di generare un numero intero che NON sia il risultato della somma di due qualsiasi numeri presenti in L
- Attenzione: nei numeri interi ci sono anche quelli negativi! ☺
- Se necessario, si può ipotizzare di avere a disposizione un generatore di numeri interi casuali che agisca in un tempo $O(1)$
 - Ma non è necessario ☺

Esercizio

- Prima osservazione: ha senso cercare una soluzione che sia "meno che lineare"?
Oppure **il problema è $\Omega(n)$?**
- Non è difficile dimostrare (ad esempio per assurdo) che, per risolvere il problema, è necessario ispezionare almeno una volta TUTTI gli elementi di L
 - *Non è strettamente necessario fare questa dimostrazione né porsi questa domanda, ma, in generale, può essere utile conoscere il limite inferiore delle prestazioni dell'algoritmo che si deve progettare*
- Per assurdo: esiste un algoritmo che calcola una soluzione x dell'esemplare L del problema senza ispezionare l'elemento i -esimo di L , con i qualsiasi...

Esercizio

- **Per assurdo:** esiste un algoritmo che calcola una soluzione x dell'esemplare L del problema senza ispezionare l'elemento i -esimo di L , con i qualsiasi (chiamiamo $L[i]$ tale elemento)
- Tale soluzione x non può, quindi, dipendere dal valore di $L[i]$, ma l'algoritmo deve trovare una soluzione per qualsiasi esemplare del problema
 - Quindi, la stessa soluzione x deve valere per tutti gli array che condividono gli $n - 1$ elementi diversi da quello in posizione i , con $L[i]$ che può assumere qualsiasi valore, perché tali esemplari del problema sono indistinguibili
- **Di conseguenza**, esiste un esemplare del problema che ha $L[i] = x - L[j]$, con $j \neq i$, per il quale $\textcolor{red}{x = L[j] + L[i]}$.
Assurdo, perché allora x non è soluzione del problema.
- Il problema è, quindi, $\Omega(n)$ e stiamo cercando una soluzione ottima, $O(n)$

Esercizio

- **Per assurdo:** esiste un algoritmo che calcola una soluzione x dell'esemplare L del problema senza ispezionare l'elemento i -esimo di L , con i qualsiasi
- Questo schema di dimostrazione è molto utile e si usa in molti casi, ad esempio per dimostrare che sono $\Omega(n)$ questi problemi
 - Cercare l'elemento minimo
 - Cercare l'elemento massimo
 - Calcolare il valore medio
 - Rispondere negativamente alla domanda "è presente?"
 - Sotto-problema della ricerca di un elemento

Esercizio

□ Prima idea

- while (true)
 - $x = \text{generateRandomInteger}()$
 - if $\text{isSolution}(x)$
 - return x

□ Nel caso peggiore, isSolution è $\Omega(n^2)$ quindi non va bene

□ Attenzione, errore classico...

- Il metodo isSolution è **$O(n^2)$, quindi non va bene**
- L'affermazione è vera, ma **la deduzione è sbagliata**
 - Aver calcolato/dimostrato che un algoritmo è $\Omega(n^2)$ NON implica che tale algoritmo non possa essere anche $O(n)$, come richiesto
 - Se, invece, l'algoritmo è $\Omega(n^2)$, allora certamente non può essere $O(n)$, come correttamente dedotto

$\text{isSolution}(x)$

- for each $y \in L$
 - for each $z \in L$
 - if $x == y + z$
return false
- return true

Esercizio

- Attenzione: **il caso peggiore** per `isSolution` **DEVE** verificarsi perché l'algoritmo trovi una soluzione!! Il metodo `isSolution` restituisce `true` (e fa così terminare l'algoritmo) solo nel suo caso peggiore!
 - Per decidere che x sia una soluzione, è **NECESSARIO** impiegare un tempo $\Omega(n^2)$
- Cosa succede se x non è una soluzione?
 - L'algoritmo genera un nuovo numero casuale, che viene verificato
 - Ogni verifica è $O(n^2)$, ma se la verifica restituisce `true` (e fa terminare l'algoritmo) allora ha richiesto un tempo $\Omega(n^2)$
 - Quindi l'algoritmo è **sempre** $\Omega(n^2)$, non solo nel caso peggiore!
 - Qual è un O-grande per questo algoritmo? Dipende da quanti numeri vengono generati... ATTENZIONE: la generazione di numeri casuali non garantisce che si trovi una soluzione! Se non dopo infiniti tentativi...
 - Ops... non è un algoritmo ☺

Esercizio

- Abbiamo dimostrato che bisogna ispezionare tutti gli elementi di L .
- Cosa si può calcolare facendo un'ispezione completa di un array di numeri interi?
 - Ad esempio, il valore medio, il valore massimo, il valore minimo...
 - Il valore massimo sembra interessante! Se trovo **il valore massimo M** e osservo che **può anche essere ripetuto** all'interno di L (non c'è scritto che i numeri di L siano tutti diversi...), allora
il valore massimo della somma di due numeri presenti in L è $\leq 2M$
 - Quindi, **$x = 2M + 1$ è una possibile soluzione del problema** e l'algoritmo che calcola x è **$\Theta(n)$** , soddisfacendo i requisiti
 - È bene verificare che i valori negativi, eventualmente presenti in L , non rendano scorretto l'algoritmo
 - L'affermazione "**il valore massimo della somma di due numeri presenti in L è $\leq 2M$** " è valida anche in presenza di valori negativi

Esercizio

- Dimostriamo che "il valore massimo della somma di due numeri presenti in L è $\leq 2M$ " quando M è il valore massimo presente in L
- Per definizione di valore massimo: $\forall y \in L, y \leq M$
 - Anche se M è negativo... Anche se y è negativo...
- Ovviamente, anche $\forall z \in L, z \leq M$
- Quindi, sommando membro a membro le due disequazioni:
$$\forall y \in L, \forall z \in L, y + z \leq M + M = 2M < 2M + 1 = x \Rightarrow y + z \neq x$$
- Si osservi che non si fa l'ipotesi che sia $y \neq z$, quindi la dimostrazione è valida anche quando i due numeri sommati sono, in realtà, il medesimo numero

Esercizio

- Esistono molte altre soluzioni
 - Ad esempio,
 $1 + 2 * (\text{la somma dei valori assoluti di tutti i numeri})$
- Si potrebbe anche pensare
 - In Java, i numeri interi sono limitati all'intervallo
`[Integer.MIN_VALUE, Integer.MAX_VALUE]`
 - Senza cercare il valore massimo presente nell'array, produco in un tempo $\Theta(1)$ la soluzione `1 + 2 * Integer.MAX_VALUE`
 - Ma tale valore NON appartiene all'intervallo!
Non posso sfruttare i vantaggi dell'utilizzo di un intervallo limitato di numeri interi senza tener conto degli svantaggi conseguenti...

Esercizio 14

Matrice binaria

- Una matrice A , $n \times n$, contiene soltanto valori 0 o 1, disposti in modo che
 - In ogni riga, tutti i valori 1 *sono a sinistra* di tutti i valori 0 (cioè sono contenuti in celle aventi indice di colonna inferiore)
 - $\forall i = 0, 1, \dots, n - 2$, il numero di valori 1 contenuti nella riga i non è minore del numero di valori 1 contenuti nella riga $i + 1$
- Supponendo che la matrice sia già presente in memoria con contenuto valido, individuare un algoritmo $O(n)$ per contare il numero totale di valori 1 presenti in A .
- L'algoritmo può verificare le proprie pre-condizioni? Perché?

Matrice binaria

- L'algoritmo può verificare le proprie pre-condizioni?
Perché?
- La verifica delle pre-condizioni richiede l'ispezione di tutte le celle della matrice, il cui numero è n^2
 - si dimostra banalmente, ad esempio per assurdo: se anche una sola cella non viene visitata e proprio in quella cella è presente il "valore sbagliato"...
- Quindi la verifica delle pre-condizioni è sempre $\Omega(n^2)$
- Conseguentemente, qualunque algoritmo che la esegua diventa $\Omega(n^2)$ e, quindi, non può essere $O(n)$

Matrice binaria

- L'idea su cui si basa l'algoritmo è quella di partire dall'angolo "in alto a destra" della matrice
 - Si inizializza a zero il conteggio totale di valori 1
 - Si procede verso sinistra finché non si trova un valore 1, che sarà il valore 1 più a destra sulla riga (posso anche trovarlo subito, se la prima riga non contiene zeri)
 - Si somma al conteggio totale il numero di valori 1 così trovati sulla prima riga (uguale a $1 + \text{l'indice di colonna della cella in cui ho trovato il primo } 1$, cioè quello più a destra)
 - Si scende di una riga verso il basso, cominciando la ricerca del valore 1 più a destra su tale nuova riga a partire dalla colonna appena ispezionata nella riga precedente e procedendo verso sinistra, perché, per ipotesi, più a destra di tale colonna non ci possono essere valori 1
 - Dopo aver trovato una riga costituita da soli zeri, inutile proseguire: per ipotesi, le righe sottostanti conterranno solo zeri

Matrice binaria

```
CountOnes(A, n) // matrice quadrata, ma si può fare anche rettangolare
count = 0
j = n-1 // indice di colonna, procede da destra a sinistra
i = 0 // indice di riga, procede dall'alto in basso
while (i < n && j >= 0)
    if (A[i][j] == 1) // ho trovato l'1 più a destra sulla riga di indice i
        count += (j+1) // nella riga i ci sono (j+1) valori 1
        i++ // passo alla riga successiva
        // j non cambia, leggerò dalla stessa colonna
        // nella riga successiva
    else
        j-- // altrimenti mi sposto a sinistra sulla riga di indice i,
        // i non cambia
return count
```

// cerchiamo ora di valutare le prestazioni: sarà $O(n)$?
// non è semplice perché è un ciclo con due variabili di controllo (i e j)

Matrice binaria

// il funzionamento dell'algoritmo si può forse comprendere meglio
// osservando questo, con cicli annidati, assolutamente **equivalente**

```
CountOnes(A, n)
    count = 0
    j = n-1
    for (i = 0; i < n; i++) // per ogni riga, dall'alto in basso
        while (j >= 0) // vado da destra a sinistra a partire da j
            if (A[i][j] == 1) // appena trovo un valore 1 mi fermo
                count += (j+1) // interrompo il ciclo while
                break // questo farà incrementare i, così scendo
            else j-- // ho visto uno 0, procedo verso sinistra sulla riga
    // osservare che, cambiando riga, j NON viene modificata
    return count
```

// ma analizziamo il ciclo unico della prima versione, è più semplice

Matrice binaria

count = 0, j = n-1, i = 0

```
while (i < n && j >= 0) if (A[i][j] == 1) { count += (j+1); i++; } else j--  
return count
```

- Una tecnica spesso utile per valutare le prestazioni di cicli controllati da **più variabili** (o di cicli annidati che non siano banali) consiste nell'utilizzo di una **variabile ausiliaria**, dipendente da quelle di controllo del ciclo, di cui sia più facile valutare l'evoluzione, iterazione dopo iterazione
- Ciascuna iterazione di questo ciclo **incrementa i oppure decrementa j** di un'unità, ma non fa **mai entrambe le cose** (perché c'è un **if/else**)
 - Proviamo a definire una variabile che si incrementi sia quando i viene incrementata sia quando j viene decrementata
 - È facile: **k = i – j**

Matrice binaria

count = 0, j = n-1, i = 0

```
while (i < n && j >= 0) if (A[i][j] == 1) { count += (j+1); i++; } else j--  
return count
```

- Ciascuna iterazione di questo ciclo **incrementa i oppure decrementa j** di un'unità, ma non fa mai entrambe le cose, e ne fa sempre esattamente una
 - Se **k = i – j**, ogni iterazione del ciclo incrementa **k** di un'unità
- Ogni iterazione del ciclo viene eseguita in un tempo costante (fa un solo accesso a celle della matrice), quindi, per valutare le prestazioni dell'algoritmo nel suo caso peggiore, è sufficiente valutare il numero massimo di incrementi unitari che può subire la variabile **k** (naturalmente, in funzione di *n*)
 - La variabile **i** può essere incrementata al massimo *n* volte, dopodiché il ciclo termina
 - La variabile **j** può essere decrementata al massimo *n* volte, dopodiché il ciclo termina
 - Quindi, la variabile **k** può essere incrementata **al massimo *2n – 1* volte** e, conseguentemente, l'algoritmo è **O(n)**
 - Riflessione: perché **non** al massimo *2n* volte?

Si osservi che k parte da $1 - n$ e viene incrementata.

Matrice binaria Se si vuole che parta da 0 basta definire

$$k = i - j + (n - 1), \text{ ma non serve.}$$

count = 0, j = n-1, i = 0

```
while (i < n && j >= 0) if (A[i][j] == 1) { count += (j+1); i++; } else j--  
return count
```

- L'algoritmo è $O(n)$. Ma è anche $\Theta(n)$? Anche se non richiesto, ragioniamo...
- Il **caso migliore** si ha, ovviamente, quando il ciclo **termina** dopo aver fatto il **minimo** numero di iterazioni
 - Questo può accadere, facendo esattamente n iterazioni, quando j non viene mai modificata (quindi i viene modificata ad ogni iterazione e raggiunge il più rapidamente possibile il proprio valore massimo, n) oppure i non viene mai modificata (quindi j viene modificata ad ogni iterazione e raggiunge il più rapidamente possibile il proprio valore minimo, -1)
 - Questo succede in due soli casi:
 - La matrice **non contiene zeri** (quindi j non viene mai modificata), oppure
 - La matrice **non contiene uni** (quindi i non viene mai modificata, perché il ciclo termina dopo aver esaminato la sola prima riga e aver posto $j = -1$)
- Quindi, **nel caso migliore, l'algoritmo è $\Omega(n)$**
- Di conseguenza, **l'algoritmo è $\Theta(n)$**

Lezione 11

ADT di tipo Lista

**È necessario aver completato
il "ripasso autonomo"...**

List (o sequenza)

- Una *lista* (o *sequenza*) è una collezione di dati tra i quali esiste una **relazione posizionale lineare**, cioè di tipo **precedente/successivo**
 - Detto in altre parole: è una collezione S di n elementi tra i quali è definito l'elemento in **prima posizione**, quello in **seconda posizione**, e così via, fino a quello in posizione n -esima
- Non c'è NESSUN requisito di ordinamento dei dati in base al loro valore: non è nemmeno necessario che i dati appartengano a un insieme dotato di una relazione d'ordine
- C'è, storicamente, un po' di confusione (ormai ineliminabile) sulla nomenclatura relativa alle liste
 - Il termine “lista” è uno di quelli più ambigui

Lista (o sequenza)

- Una *lista* (o *sequenza*) è una collezione di dati tra i quali esiste una **relazione posizionale lineare**, cioè di tipo **precedente/successivo**
- L'array e la catena (semplicemente o doppiamente concatenata) sono esempi di strutture dati che realizzano concretamente una lista
 - Spesso la *catena* è chiamata *lista concatenata*
- La lista **NON** specifica come sia concretamente realizzata la relazione precedente/successivo
 - La lista è, quindi, un **ADT** (*abstract data type*, tipo di dato astratto)

Lista

- Ciascun dato presente in una lista può essere univocamente identificato in diversi modi, per cui definiremo, corrispondentemente, diversi ADT
 - Mediante un *indice* (*index*)
(un numero intero compreso tra **0** e **size () -1**)
 - **L'indice di un elemento è definito come il numero di elementi che lo precedono nella sequenza**
 - A precede B se e solo se l'indice di A è minore di quello di B
 - Meno frequentemente, mediante un *rango* (*rank*)
(un numero intero compreso tra **1** e **size ()**)
 - **Il rango di un elemento è definito come UNO più il numero di elementi che lo precedono nella sequenza**
 - Mediante una *rappresentazione astratta della sua posizione*

Lista con índice o con rango

Lista con indice (o con rango)

- Se la lista identifica i dati mediante un indice si definisce il corrispondente ADT **IndexList**

- Si chiama anche, tradizionalmente, **vettore**
- Se la lista usa il rango, si definisce un ADT **identico** denominato **RankedList** o **RankList**
- Tutti i metodi lanciano **IndexOutOfBoundsException**

```
public interface IndexList<T> extends Container
{
    T get(int index);
    T set(int index, T element);
    T remove(int index);
    void add(int index, T element);
}
```

ripasso

- Questa interfaccia **NON** è presente nella libreria standard Java ma c'è una classe (**ArrayList**) che ha questo comportamento

IndexList

- **get** restituisce il dato di indice $0 \leq \text{index} < \text{size}()$
- **set** sostituisce con **element** il dato di indice $0 \leq \text{index} < \text{size}()$ e restituisce il dato precedentemente presente in tale posizione
- **add** inserisce nel contenitore il dato con indice $0 \leq \text{index} \leq \text{size}()$
 - i dati presenti non vengono modificati ma cambia l'indice associato ai dati aventi indice maggiore o uguale a quello indicato
 - cioè “si fa spazio” al nuovo elemento
 - si noti che si può “aggiungere alla fine”, in una **nuova** posizione
 - l'esecuzione del metodo aumenta di uno il numero degli indici validi
- **remove** elimina (e restituisce) il dato di indice $0 \leq \text{index} < \text{size}()$
 - modifica l'indice associato ai dati aventi indice maggiore di quello indicato (cioè “chiude il buco”...) e decrementa il valore massimo degli indici validi
 - l'esecuzione del metodo diminuisce di uno il numero degli indici validi
- Poi ci sono i metodi **isEmpty()** e **size()**, come in tutti i **Container**

Lista con indice (o con rango)

- Si può realizzare una **IndexList** mediante una struttura concatenata, ma tutte le operazioni sono **$\Theta(n)$** nel caso peggiore e **medio**: la catena non consente l'accesso casuale alle singole posizioni
- È invece semplice ed efficiente realizzare un vettore mediante un array (riempito solo in parte), progettando la classe **ArrayList** (cioè una **IndexList** realizzata mediante un **Array**)
 - spesso viene semplicemente chiamata **ArrayList**, come nella libreria standard

```
public class ArrayList<T> implements IndexList<T>
{  private T[] v;
   private int vSize;
   ...
}
```

- le operazioni **get** e **set** sono **$\Theta(1)$** (accesso casuale)

Lista con indice (o con rango)

□ Le operazioni **add** e **remove** sono (in media e nel caso peggiore) $\Theta(n)$, perché devono spostare mediamente $n/2$ elementi (al massimo n)

- Sono, però, $\Theta(1)$ nel caso migliore, che è piuttosto importante: quando agiscono sull'**ultimo** elemento della lista, cioè quello di indice massimo
 - Per **add** bisogna usare **l'analisi ammortizzata** (anche per **remove**, nel caso in cui la struttura usi il ridimensionamento "in piccolo", cioè la riduzione di dimensione quando il fattore di occupazione scende sotto una certa soglia)

Array circolare:

strategia di
gestione di un
array vista nel
"ripasso
autonomo"
relativo alle code

- Usando una realizzazione ad “array circolare”, si possono rendere tali operazioni $\Theta(1)$ anche quando agiscono sul **primo** elemento della lista
 - Rimangono $\Theta(n)$ nel caso generale, quando agiscono su elementi intermedi all'interno della lista

Lista con posizione astratta

Lista con posizione astratta

- Usando una **struttura lineare concatenata** (catena o *linked list*), è facile realizzare una lista che rappresenti la posizione dei dati al proprio interno tramite un concetto astratto, che non sia un numero come un indice... ma un oggetto più complesso
 - Basta usare il **nodo**, che, nella catena, contiene un dato, come rappresentazione astratta della **posizione** del dato stesso
 - Esclusi i casi degeneri (primo e ultimo), ciascun nodo ha un nodo precedente e un nodo successivo, proprio come una posizione
 - Così facendo, però
 - si vincola la realizzazione dell'ADT a una particolare struttura interna, che usi un particolare tipo di nodo (ad esempio, a “singola” o “doppia” concatenazione?)
 - si “espone” eventualmente il nodo a manipolazioni dirette dall'esterno, che magari “spezzano” la catena...

Posizione in una lista

- Meglio definire un *concetto astratto di posizione* mediante un'apposita interfaccia

- Che comportamento deve avere una *posizione* all'interno di una lista? **Deve identificare univocamente un dato della lista, allo scopo di consentire (almeno) la sua ispezione**

```
public interface Position<T>
{ T element(); }
```

Sarebbe stato meglio chiamarlo
getElement () ...

- Una posizione **non** è un ADT di tipo contenitore

- In realtà è anch'esso un contenitore, ma può contenere un solo dato... quindi non appartiene alla generica categoria degli ADT di tipo contenitore
- Infatti non estende **Container**, non ha il metodo **isEmpty** né **size**... perché non avrebbero significato

Problemi di
nomenclatura...

Posizione in una lista

```
public interface Position<T>
{ T element(); }
```

- Una “lista con posizione astratta” o lista posizionale (**spesso detta “lista”**) è, quindi, un contenitore che memorizza ciascun dato in una diversa **posizione**, per ognuna delle quali è definita una posizione precedente e una posizione successiva
 - La prima posizione non ha precedente e l’ultima non ha ~~suecessiva~~

- Notiamo che **SLLNode** e **DLLNode**, i nodi delle catene, realizzano già questa interfaccia (perché hanno già il metodo **getElement**), basta che **aggiungiamo loro la relativa clausola implements e inseriamo un metodo stub**

```
public class SLLNode<T> implements Position<T>
{
    ...
    public T element() { return getElement(); } // stub
} // lo stub serve a "cambiare nome" a un metodo...
```

Visti nel
ripasso

Liste con posizione astratta

- A questo punto possiamo definire un ADT **PositionalList** che rappresenti il comportamento di una lista (cioè di una collezione di dati tra i quali esiste una relazione posizionale di tipo precedente/successivo) nella quale la posizione di ciascun dato sia rappresentata in modo astratto da un esemplare di una classe (ad esempio, **SLLNode** o **DLLNode**) che implementi l’interfaccia **Position**
 - A volte **PositionalList** viene chiamata **NodeList** (nome peggiore...)
 - Ha metodi che consentono di “navigare” da una posizione all’altra e metodi che consentono di ispezionare/modificare/inserire/rimuovere dati

Lista con posizione astratta

```
public interface PositionalList<T> extends Container
{ // metodi di "navigazione" tra le posizioni
    Position<T> first()
        throws EmptyListException;
    Position<T> last()
        throws EmptyListException;
    Position<T> prev(Position<T> p) // sta per "previous"
        throws InvalidPositionException,
            BoundaryViolationException;
    Position<T> next(Position<T> p)
        throws InvalidPositionException,
            BoundaryViolationException;
    // metodi di "modifica" del contenuto della lista
    ...
} // prosegue nella pagina successiva
```

Lezione 12

Lista con posizione astratta

```
public interface PositionalList<T> extends Container
{ // metodi di "navigazione" tra le posizioni
    ... // quelli appena visti
    // metodi di "modifica" del contenuto della lista
    void addFirst(T element);
    void addLast(T element);
    void addBefore(Position<T> p, T element)
        throws InvalidPositionException;
    void addAfter(Position<T> p, T element)
        throws InvalidPositionException;
    T set(Position<T> p, T element) // cambia il dato
        throws InvalidPositionException;
    T remove(Position<T> p)
        throws InvalidPositionException;
}
```

- Si osservi che nella definizione di questo ADT non c'è nessun riferimento **esplicito** al concetto di **nodo**
 - Si può realizzare concretamente anche usando un array (ma in tal caso non è per niente efficiente)

Lista con posizione astratta

- Non ha un metodo **get** per ispezionare i dati...
ma che parametro potremmo fornirgli? Sarebbe
 - T get (Position<T> p)**
throws InvalidPositionException;
- Per invocare **get** ci serve una posizione... ma, avendo una posizione, è inutile invocare **get**, basta invocare **element** sulla posizione stessa!
 - Per completezza e analogia con **set**, si può però aggiungere
- Quindi, per accedere a un elemento si invoca **first** o **last** e si naviga (**next/prev**) fino alla posizione che interessa, poi si invoca **element** sulla posizione

Lista con posizione astratta

- Come alternativa di progetto, si può aggiungere un metodo **setElement** all’interfaccia **Position**
 - Rimane il vantaggio di **impedire modifiche alla struttura** della catena dall’esterno e si può eliminare il metodo **set** nell’interfaccia **PositionalList**
- Oppure, alternativa: si può definire l’interfaccia **Position** priva di metodi!

public interface Position<T> { }

 - Bisogna conseguentemente definire il metodo **get** nell’interfaccia **PositionalList**
 - A cosa serve un’interfaccia priva di metodi? Come tutte le interfacce... a definire un tipo di dato astratto! Talmene astratto che non ha alcun comportamento... serve soltanto a introdurre **IL NOME** di un ADT, che magari possa essere condiviso (cioè "implementato") da più tipi di dati concreti, in questo caso più tipi di nodi... che hanno in comune soltanto il fatto di rappresentare il concetto astratto di posizione in una lista, senza fornire "servizi"

Lista con posizione astratta

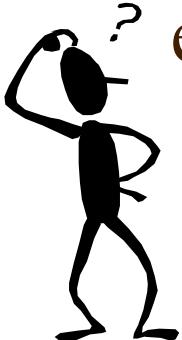
- I metodi **prev(p)** / **next(p)** lanciano
 - **BoundaryViolationException** se **p == first()** / **p == last()**
- Apparentemente il metodo **addFirst(e)** è ridondante
 - Si potrebbe svolgere la stessa funzione invocando **addBefore(first(), e)** ?
 - Sì, ma soltanto **in una lista NON vuota**, perché in una lista vuota l'invocazione di **first** provoca il lancio di un'eccezione: in tal caso è necessario invocare **addFirst**
- Analogamente, il metodo **addLast(e)** si può sostituire con **addAfter(last(), e)**, tranne quando la lista è vuota

Lista con posizione astratta

- Un esemplare di **Position** rappresenta **la posizione di uno specifico dato all'interno della lista**, non una particolare posizione (cioè non “la terza posizione”...)
- Questo significa che, dopo l'invocazione di **remove (p)**, la posizione **p** non è più valida, in quanto non esiste più nella lista un dato avente quella posizione (proprio perché il dato è stato rimosso)
 - Due invocazioni di **remove** con la stessa posizione provocano il lancio di **InvalidPositionException**
- **Al contrario**, in una lista con indice/rango, **ciascun indice/rango è associato a una posizione** (“la terza cella”) e non a un dato
 - Inserendo un dato con indice k , il dato che **aveva** indice k viene ad avere indice $k + 1$

Lista con posizione astratta

- Dal punto di vista dell'ingegneria del software, è **fondamentale** che le “posizioni” siano esemplari di una classe **PRIVATA, INTERNA** alla lista, altrimenti chi le riceve sotto forma di riferimento di tipo **Position** può fare un down-casting e ottenere l'oggetto (ad esempio, il nodo), per poi manipolarlo...
? e non va bene!



Realizzazione di PositionalList

- È abbastanza naturale realizzare una **PositionalList** mediante una struttura a nodi (doppiamente) concatenati, assai simile a una catena doppia

```
public class DLinkedListPositionalList<T>
    implements PositionalList<T>
{
    private class DLLNode<T> implements Position<T>
    {
        ...
    } // già vista nel ripasso autonomo
    private DLLNode<T> head, tail;
    private int size;
    public DLinkedListPositionalList() // crea lista vuota
    {
        size = 0;
        head = new DLLNode<T>();
        tail = new DLLNode<T>();
        head.setNext(tail);
        tail.setPrev(head);
    } // si può fare anche senza nodi header/trailer
      // ma è più complicato (pur con stesse prestazioni)
    ...
}
```

Realizzazione di PositionalList

```
public class DLinkedListPositionalList<T>
    implements PositionalList<T>
{
    ... // PROSEGUE
    public Position<T> first()
    { if (isEmpty()) throw new EmptyListException();
        return head.getNext(); /* restituisce un esemplare
                               di DLLNode, che è valido come Position */
    }
    public void addBefore(Position<T> p, T element)
        throws InvalidPositionException
    { DLLNode<T> n = checkPosition(p);
        ...
    }
    private DLLNode<T> checkPosition(Position<T> p)
        throws InvalidPositionException
    { /* è una posizione valida ?? */
        if (p == null || !(p instanceof DLLNode))
            // verifica incompleta, vedi in seguito...
            throw new InvalidPositionException();
        ...
        return (DLLNode<T>) p; // il cast servirà a tutti
    }                                // gli altri metodi
    ... // PROSEGUE
}
```

Lista con posizione astratta

- Tutti i metodi che ricevono una posizione **p** come argomento lanciano **InvalidPositionException** se **p** non è una posizione valida nella lista, cioè se
 - **p == null**
- oppure
 - **p** è una posizione **di un tipo diverso** dal tipo previsto per le posizioni della lista (si usa **instanceof**, d'altra parte vivamente consigliato per poter fare down-casting e accedere alle proprietà e funzioni del nodo)
- oppure
 - **p è una posizione che non appartiene alla lista**
 - i casi precedenti sono sotto-casi di questo, ma si tengono distinti perché sono più semplici da verificare

Lista con posizione astratta

- **Se p è una posizione che non appartiene più alla lista (ma vi apparteneva)**
 - **Non è difficile fare in modo di poter verificare questa precondizione**
 - Ad esempio, se le “posizioni” sono nodi doppiamente concatenati, questi sono “validi” soltanto se hanno almeno un collegamento non nullo verso un altro nodo (entrambi non nulli se si usano nodi header/trailer)
 - Ricordare che i nodi header/trailer non sono, però, posizioni valide, perché non contengono dati!! Ma tali nodi non vengono mai restituiti "all'esterno" da un metodo, quindi non possono "rientrare" come parametri di un altro metodo

Basta che il metodo **remove (p)** (l'unico che elimina posizioni dalla lista) renda “non valido” il nodo che ha ricevuto come argomento (sotto forma di posizione), rendendo nulli i suoi due collegamenti ad altri nodi (con nodi a concatenazione semplice si può usare un'apposita variabile di esemplare booleana) [oltre, ovviamente, a "staccare" il nodo dalla catena]

Lista con posizione astratta

- Se **p** è una posizione che **non appartiene più** alla lista (ma vi apparteneva)
 - Come visto, non è difficile fare in modo di poter verificare questa pre-condizione
- **Se, invece, p è una posizione che non è mai appartenuta alla lista (ma è del tipo corretto ed è, ad esempio, una posizione valida all'interno di un diverso esemplare della lista)**
 - **Non è facile verificare questa pre-condizione in un tempo $\Theta(1)$, ci vuole un “trucco”...**
 - In un tempo $O(n)$ è facile: si fa una ricerca sequenziale tra i nodi presenti nella lista... ma è “assurdo” !!
Molti dei metodi che hanno bisogno di verificare la validità della posizione ricevuta sono $\Theta(1)$...

Realizzazione di PositionalList

- Progettare in Java una struttura dati generica che implementi completamente l'interfaccia **PositionalList<T>** con **prestazioni ottime** per ogni metodo
 - Usare una struttura doppiamente concatenata, i cui nodi implementino l'interfaccia **Position<T>**
 - In particolare, tutti i metodi che ricevono "posizioni" come argomenti devono verificare, **in un tempo O(1)**, che siano posizioni valide all'interno della lista (definire un metodo privato ausiliario che svolga questo compito e venga invocato dagli altri metodi)
 - Suggerimento: serve uno spazio aggiuntivo $\Theta(n)$ (ma non una lista/catena/array/pila/coda, di nessun tipo... perché?)
 - Progettare alcune classi di collaudo per la struttura realizzata

Lezione 13

Prestazioni di PositionalList

- In una **PositionalList** realizzata mediante catena a doppia concatenazione, tutti i metodi hanno prestazioni temporali asintotiche $\Theta(1)$ [vedere esercizio...]
 - Se la si realizza, invece, mediante una catena a concatenazione semplice, i metodi che necessitano di riferimenti "all'indietro" hanno prestazioni $O(n)$
 - `prev`, `remove`, `addBefore`
- In Java è presente `java.util.LinkedList` che è proprio una catena a doppia concatenazione e presenta un'interfaccia abbastanza simile a **PositionalList** (ma un po' diversa)
 - Non vediamo i dettagli

IndexList o PositionalList ?

- La scelta del tipo di lista dipende dall'utilizzo che ne deve fare l'applicazione che si sta progettando
- Bisogna tenere presente le prestazioni temporali dei singoli metodi e/o l'occupazione di spazio
 - Se l'applicazione richiede l'utilizzo dell'accesso casuale ai singoli elementi, è probabilmente preferibile un'implementazione di **IndexList**
 - Es. ricerca per bisezione
 - Se l'applicazione effettua soltanto inserimenti, rimozioni e ispezioni alle due "estremità" della lista, è probabilmente preferibile un'implementazione di **PositionalList**
 - Es. una coda doppia (**Deque**)
 - ...

Iteratori

Iteratore per una lista

□ Operazione frequente su una lista (di ogni tipo)

- Ispezione di tutti i suoi elementi, dal primo all'ultimo (o fino a un certo punto), secondo la relazione posizionale indotta dalla lista
 - Esempi: ricerca sequenziale di un oggetto, ricerca del valore minimo/massimo (se definito, cioè se i dati della lista appartengono a un insieme totalmente ordinato...), calcolo del valore medio (se definito...)
- È comodo e utile definire un **iteratore**: un oggetto che *naviga* ordinatamente nella sequenza posizionale di dati contenuti nella lista, **indipendentemente dal tipo di lista...**
 - Al termine della presentazione vedremo motivi per cui è utile/comodo
- Un iteratore I è l'insieme di
 - una lista L (su cui opera)
 - un (riferimento a un) **elemento corrente** (o attuale) c in L detto *cursor*: l'elemento in cui è *posizionato* l'iteratore
 - una modalità per spostarsi all'**elemento successivo** di c in L , facendolo diventare il nuovo elemento corrente



Iteratore per una lista

- L'iteratore astrae e rappresenta, insieme, i concetti di “posizione” e di “successivo” all'interno di una lista
 - In pratica, anche se non sintatticamente, estende il concetto di “posizione”: una posizione è un iteratore che non si sposta! O, viceversa, un iteratore è una posizione che si sposta! ☺
- Definiamo un ADT che rappresenti un iteratore che agisce su una generica lista:
usiamo `java.util.Iterator`

```
public interface Iterator<T>
{ T next() throws NoSuchElementException;
  boolean hasNext(); // metodo non necessario
}                                // ma utile
```

Lista con iteratore

- Invece del metodo **element()** che troviamo in **Position<T>**, un iteratore ha il metodo **next()** che restituisce, ad ogni invocazione, il dato successivo a quello restituito in precedenza (oppure il primo dato, alla sua prima invocazione)
 - Se non esiste un dato successivo,
next() lancia **NoSuchElementException**
 - Scegliere di restituire **null** non è ragionevole, perché **null** è un dato che può legittimamente essere presente in una lista
(infatti i metodi di inserimento nelle liste NON rifiutano i valori **null**)
 - Alla prima invocazione,
next() restituisce il primo dato presente nella lista
 - Lancia **NoSuchElementException** se la lista è vuota
 - C'è anche un metodo **hasNext()** che restituisce **false** se e solo se un'invocazione di **next()** provocherebbe il lancio di **NoSuchElementException**

Interfaccia `java.lang.Iterable`

- Una lista che fornisce, a richiesta, un iteratore che agisca su di essa implementa l'interfaccia `java.lang.Iterable`

```
public interface Iterable<T>
{ java.util.Iterator<T> iterator();
} // sarebbe meglio getIterator()...
```

- **Ridefiniamo** le liste viste in precedenza, le classi concrete dovranno definire anche il metodo `iterator()`

```
public interface PositionalList<T> extends Container,
                                         Iterable<T>
{ ... // come prima ma con iterator() aggiunto
}
```

```
public interface IndexList<T> extends Container,
                                         Iterable<T>
{ ... // come prima ma con iterator() aggiunto
} // analogamente per RankedList<T>
```

Interfaccia `java.lang.Iterable`

- Una “lista iterabile” o “lista navigabile”, che fornisce, a richiesta, un iteratore che agisca su di essa, implementa, quindi, questa interfaccia

```
public interface Iterable<T> // in java.lang
{   java.util.Iterator<T> iterator();
}
```

- Il metodo **iterator()** implementato in una specifica lista restituirà un esemplare di una specifica classe (**generalmente interna privata**) che (ovviamente...) realizza l’interfaccia **Iterator<T>** e funge da iteratore per la lista stessa

Interfaccia `java.lang.Iterable`

- Implementare `java.lang.Iterable` è anche condizione sufficiente perché un oggetto possa essere utilizzato nell'enunciato “for generico” del linguaggio Java
 - Non è condizione necessaria:
gli array (“oggetti primitivi” in Java) funzionano nel “for generico” ma **non** implementano `java.lang.Iterable`

```
Iterable<T> list = new DLinkedListPositionalList<T>();  
...  
for (T obj : list)  
    System.out.println(obj);  
  
double[] a = new double[5];  
...  
for (double d : a)  
    System.out.println(d);
```

Iteratore per una lista

- La definizione di un iteratore su una lista **non** è condizione **necessaria** per effettuarne una scansione
 - Una lista con indice/rango si può scandire effettuando ispezioni dirette con **get**, usando valori crescenti di indice/rango, a partire dal primo (rispettivamente, 0 o 1)
 - Una lista con posizioni astratte si può scandire usando ripetutamente **next** sulla posizione ottenuta inizialmente invocando **first**
- Però con l'iteratore la scansione
 - **Si può fare in modo omogeneo per liste di tipo diverso!**

Uso di iteratori

- Operazioni omogenee per liste diverse

```
static <T> boolean find(T elem, Iterable<T> list)
{  if (list == null) return false;
   Iterator<T> iter = list.iterator();
   while (iter.hasNext())
      if (elem.equals(iter.next()))
         return true;
   return false;
}
// C'è un problema se elem è null... Vediamo dopo...
```

- Altrimenti dovremmo definire più metodi

```
static <T> boolean find(T e, IndexList<T> list) { ... }
static <T> boolean find(T e, RankedList<T> list) { ... }
static <T> boolean find(T e, PositionalList<T> list) { ... }
```

Uso di iteratori

□ È possibile che **elem** sia **null**?

- Sì, perché **in genere nei contenitori è ammessa la presenza di valori null** e, infatti, l'inserimento di valori **null** non viene generalmente impedito dai metodi di inserimento
- Quindi, può avere senso cercare un valore **null** all'interno di un contenitore: bisogna tenerne conto!

```
static <T> boolean find(T elem, Iterable<T> list)
{ if (list == null) return false;
  Iterator<T> iter = list.iterator();
  while (iter.hasNext())
    if (elem == null)
      { if (iter.next() == null) return true;
        // oppure if (elem == null) throw... se null
        // non ammesso, ma non è molto ragionevole...
        } else if (elem.equals(iter.next())) return true;
  return false;
}
```

Uso di iteratori

- Analogamente, se un metodo deve **creare e restituire** una collezione di elementi tra i quali esiste una relazione posizionale lineare, ma all'invocante non interessano le modalità di accesso alla struttura che ospita la collezione (ad esempio perché dovrà soltanto fare una scansione), il metodo può restituire un riferimento di tipo **Iterable<T>**
 - A volte è un po' scomodo non avere a disposizione il metodo **size()** che si avrebbe restituendo una lista, cioè è scomodo che né **Iterable** né **Iterator** abbiano **size()**
- All'interno del metodo, tale riferimento verrà inizializzato con un esemplare di una classe che realizza concretamente una lista qualsiasi (con indice, con posizioni astratte...)
- Tale riferimento verrà usato da chi lo riceve soltanto mediante un iteratore di tipo **Iterator<T>**, ottenuto invocando **iterator()**

Uso di iteratori

- Dovendo restituire genericamente una lista scansionabile, è preferibile restituire un riferimento **Iterable<T>** oppure un riferimento **Iterator<T>**? **Che differenza c'è?**

```
static <T> Iterator<T> returnIterator()
{   ArrayList<T> list = new ArrayList<T>();
    ... // inserisce dati nella lista
    return list.iterator(); }
static <T> Iterable<T> returnIterable()
{   ArrayList<T> list = new ArrayList<T>();
    ... // inserisce dati nella lista
    return list; }
static <T> myMethod()
{   Iterator<T> iterator = returnIterator();
    Iterable<T> iterable = returnIterable();
    ... }
```

- Con **iterator** posso fare un'unica scansione della lista, con **iterable** posso fare tutte le scansioni che voglio, invocando ripetutamente **iterable.iterator()**

Progetto di un iteratore

- Si può definire un iteratore che agisca su una lista **dall'esterno di essa**?
 - Sì, ma **in generale** è più efficiente (anche se spesso asintoticamente equivalente) che l'iteratore sia una classe **interna**, soprattutto se si tratta di una **PositionalList**
- Nelle liste con indice/rango è abbastanza indifferente che l'iteratore sia interno o esterno, perché la struttura consente l'accesso casuale anche da metodi esterni

Progetto di un iteratore: problemi

□ Nel progetto di un iteratore ci sono due strategie

- L'iteratore agisce su una “fotografia” dello stato della lista, “scattata” nel momento in cui esso viene creato
 - È necessario uno spazio aggiuntivo $\Theta(n)$ all'interno dell'iteratore
 - Le prestazioni del metodo **iterator()** sono $\Omega(n)$
 - Eventuali modifiche alla lista effettuate successivamente alla creazione dell'iteratore non vengono viste dall'iteratore stesso
(questo può essere un problema... oppure un requisito!)
- L'iteratore agisce direttamente sulla lista
 - È sufficiente uno spazio aggiuntivo $\Theta(1)$ nell'iteratore
 - Le prestazioni del metodo **iterator()** possono essere $\Theta(1)$
 - Eventuali modifiche alla lista effettuate successivamente alla creazione dell'iteratore vengono viste dall'iteratore stesso,
ma **possono farne fallire il funzionamento**

*snapshot
iterator*

*lazy
iterator*

Uso di iteratori

□ L'iteratore agisce direttamente sulla lista

- Eventuali modifiche alla lista effettuate successivamente alla creazione dell'iteratore vengono viste dall'iteratore stesso,
ma possono farne fallire il funzionamento

```
IndexList<Integer> list = ...;
list.add(list.size(), 5); // aggiungo in fondo
list.add(list.size(), 2); // aggiungo in fondo
Iterator<Integer> iter = list.iterator();
int x = iter.next();
if (iter.hasNext()) // garantisce che si può
    // invocare iter.next()
{   list.remove(list.size()-1); // elimino l'ultimo
    x = iter.next(); // lancia NoSuchElementException
}
```

□ Di solito, comunque, gli iteratori si progettano in questo modo, poi li si usa “con attenzione”

Uso di iteratori

- Un iteratore *lazy* può simulare il funzionamento di un iteratore *snapshot*: è sufficiente farlo operare su **un clone** della lista!
- In effetti, anziché mettere a disposizione iteratori *lazy* e *snapshot* su uno stesso tipo di lista, è più comodo progettare soltanto l'iteratore *lazy* e aggiungere alla lista un metodo di **clonazione** della lista stessa (metodo che, evidentemente, è $\Omega(n)$ e solitamente $\Theta(n)$ per una lista di dimensione n)
 - Chi ha bisogno di un iteratore *snapshot* chiede alla lista di generare un proprio clone, al quale chiederà un iteratore *lazy*
 - L'unica accortezza che l'utilizzatore deve avere è quella di non modificare il contenuto del clone mentre utilizza l'iteratore, ma può modificare il contenuto della lista originale

Progetto di iteratore lazy

```
// un esempio... come classe esterna (o interna statica)
class LazyIndexListIterator<T> implements Iterator<T>
{ IndexList<T> list;
  int cursor;
  public LazyIndexListIterator(IndexList<T> aList)
  { list = aList; // meglio eccezione se null
    cursor = 0;
  }
  public boolean hasNext()
  { return cursor < list.size(); }
  public T next()
  { if (!hasNext())
      throw new NoSuchElementException();
    return list.get(cursor++);
  }
}
public class IterableIndexList<T>
  implements IndexList<T>, Iterable<T>
{
  ...
  public Iterator<T> iterator()
  { return new LazyIndexListIterator(this);
  }
}
```

```
// un esempio... come classe interna (privata) NON statica
public class IterableIndexList<T>
    implements IndexList<T>, Iterable<T>
{ private T[] v; private int vSize;
...
public Iterator<T> iterator()
{ return new MyLazyIterator(); // nessun parametro
}
// classe interna NON statica, può accedere alle variabili di
// esemplare della classe che la contiene
private class MyLazyIterator<T> implements Iterator<T>
{ int cursor; // non serve riferimento alla lista
MyLazyIterator()
{ cursor = 0;
}
public boolean hasNext()
{ return cursor < vSize; // non serve invocare size()
}
public T next()
{ if (!hasNext())
    throw new NoSuchElementException();
return v[cursor++]; // non serve invocare get()
}
}
```

Pseudocodice per liste

- Quando il problema parla genericamente di "lista", è meglio usare una "lista con iteratore", indipendentemente dalla sua realizzazione, ipotizzando che
 - Un iteratore venga costruito con
`iter = list.iterator()` in un tempo $\Theta(1)$ e occupi uno spazio $\Theta(1)$ (cioè NON operi in "modalità fotografia")
 - Con l'iteratore si possano invocare i metodi
 - `iter.hasNext()` in un tempo $\Theta(1)$
 - `iter.next()` [oppure `iter.getNext()`] in un tempo $\Theta(1)$

```
iter = list.iterator()
while iter.hasNext()
    x = iter.getNext()
```

Iteratori bidirezionali

- A volte si definiscono anche iteratori bidirezionali

```
public interface TwoWayIterator<T>
{ T next() throws NoSuchElementException;
  boolean hasNext();
  T previous() throws NoSuchElementException;
  boolean hasPrevious();
}
```

- **Nella libreria standard di Java non c'è**
- C'è, però, un iteratore più complesso e articolato
 - Estende `java.util.Iterator<T>`
 - È **bidirezionale**
 - È “attivo”, perché **consente di apportare modifiche alla lista su cui naviga**

Interfaccia `java.util.ListIterator`

```
public interface ListIterator<T> extends Iterator<T>
{ T previous() throws NoSuchElementException;
  boolean hasPrevious();
  void add(T elem);
  void set(T elem);
  void remove();
}
```

- Ci sono parecchi dettagli complicati
- Le operazioni di modifica (`add`, `set` e `remove`) sono **OPZIONALI**
 - Si possono definire in modo che lancino **sempre** `UnsupportedOperationException`
 - Ricordare: **convenzione standard per implementare parzialmente un'interfaccia!**
 - In questo modo si ha un semplice iteratore bidirezionale

Il cursore di un iteratore

- La posizione di un iteratore all'interno di una lista (che determina quale sarà il successivo dato da restituire) viene rappresentata mediante il suo "**cursore**"
- Nelle raffigurazioni grafiche, il cursore viene rappresentato in vari modi, ad esempio
 - Una freccia posizionata tra due elementi: il prossimo dato restituito sarà quello a destra (o quello a sinistra) della freccia
 - **Una "finestra" posizionata su un elemento** (o una freccia che indica l'elemento stesso): il prossimo dato restituito sarà quello inquadrato dalla finestra, oppure quello alla sua sinistra o alla sua destra
- Sono tutte rappresentazioni equivalenti!
Basta usarle in modo **coerente**, tanto nel costruttore dell'iteratore quanto nei metodi **next/hasNext**

Il cursore di un iteratore

- Se l'iteratore opera su una catena, solitamente il suo cursore è un riferimento a un nodo, quindi è meglio usare una freccia che punta a un elemento (o una finestra)
- Se l'iteratore opera su una lista con indice/rango, solitamente il suo cursore è un indice/rango, quindi, di nuovo, è meglio usare una freccia che punta a un elemento (o a una finestra)
- La freccia che punta **tra due elementi** è un'astrazione non facilmente traducibile nel codice: di solito la si memorizza come *il riferimento alla posizione (nodo o indice) a destra della freccia*, ma, allora, tanto vale usare una freccia che punta all'elemento! (o una finestra)

Il cursore di un iteratore

- Bisogna poi **decidere come utilizzare il cursore**, che a questo punto immaginiamo essere una finestra
- Esempio. **next () restituisce il dato a cui punta il cursore**, quindi:
 - Il costruttore posiziona il cursore sul primo dato (o null/-1 se la lista è vuota)
 - Se hasNext () == true, allora next () restituisce il dato a cui punta il cursore e fa avanzare il cursore (facendolo eventualmente diventare uguale a null/size () dopo aver restituito l'ultimo dato)
 - hasNext () restituisce true se e solo se il cursore è diverso da null/-1/size ()

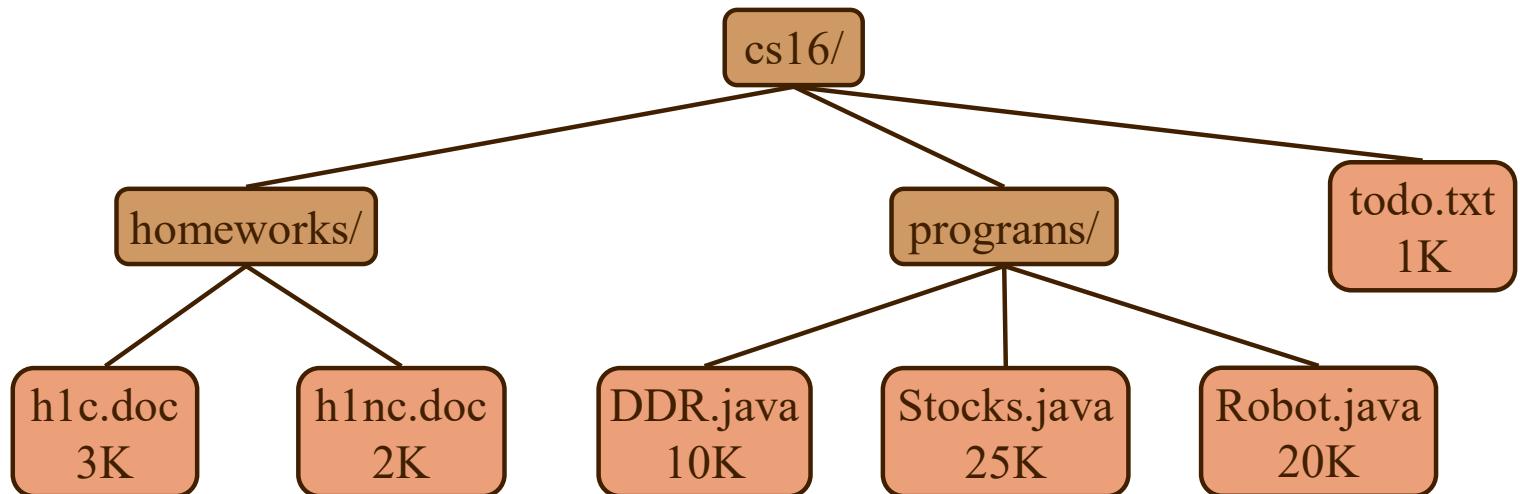
ADT sequenza (poco usato)

- È un ADT “misto”, funziona contemporaneamente come
 - **Deque**
 - **IndexList**
 - **PositionalList**
- Ogni dato ha **sia** una posizione astratta **sia** un indice
- Oltre all'unione dei metodi delle tre interfacce, ha due metodi di “traduzione” tra indici e posizioni

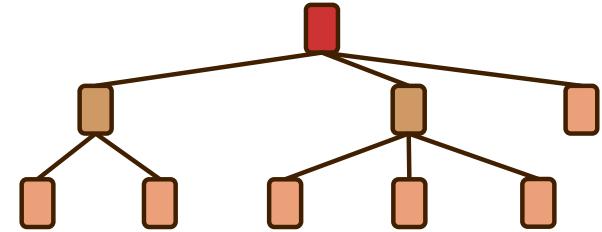
```
public interface Sequence<T>
    extends Deque<T>, IndexList<T>, PositionalList<T>
{
    Position<T> atIndex(int index)
        throws BoundaryViolationException;
    int indexOf(Position<T> position)
        throws InvalidPositionException;
} // non presente nella libreria standard di Java
// anche se è simile a java.util.List
```

Lezione 14

Alberi



Strutture non lineari



- Una *lista* o *sequenza* è un insieme di dati tra i quali esiste una **relazione posizionale lineare**, cioè di tipo precedente/successivo
 - Forti limiti alla rappresentazione dei dati
 - Esempio: Come rappresentare la **struttura gerarchica** di un'azienda?
- In **molti casi** è utile dare alla rappresentazione delle informazioni una **struttura non lineare**
 - Ad esempio una **struttura gerarchica**, come quella degli **alberi**
 - Altre strutture che vedremo sono **non lineari ma NON gerarchiche**, come, ad esempio, i grafi

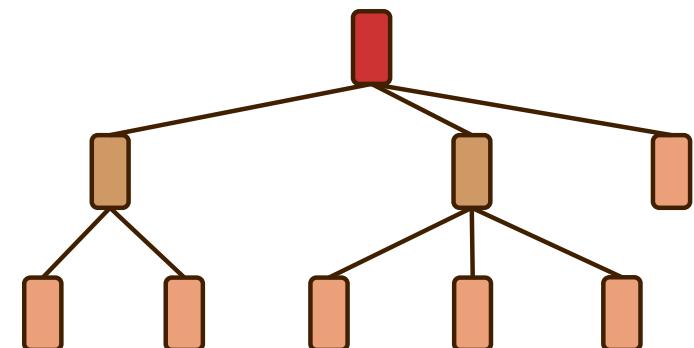
Strutture gerarchiche

□ Alcuni esempi di strutture gerarchiche

- Organizzazione dei file in un dispositivo di memoria di massa (*file system*)
- Organizzazione delle classi nel linguaggio Java

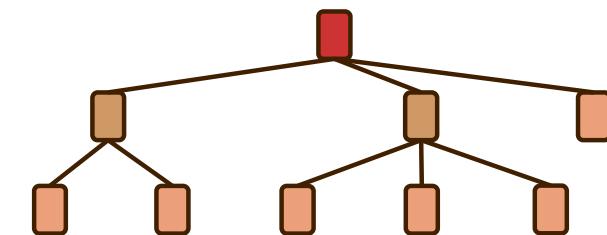
□ Nelle strutture lineari si parla di “prima/dopo” o
“precedente/successivo”

□ Nelle strutture gerarchiche si parla di “**genitore/figli**”
(*parent/children*, o, storicamente, “**padre/figli**”)



Alberi

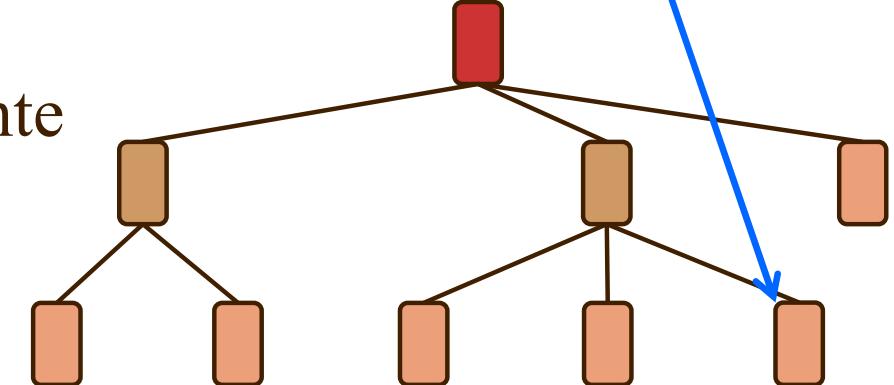
- Un albero (*tree*) è un modello astratto per rappresentare una struttura gerarchica
- Un albero T è costituito da un insieme (eventualmente vuoto) di nodi (o vertici o posizioni) tra i quali esiste una **relazione gerarchica di tipo genitore/figli** così definita:
 - Se T non è vuoto, **uno e uno solo dei suoi nodi non ha genitore** e viene chiamato **radice** (*root*) di T
 - Ogni nodo diverso dalla radice ha **un unico** nodo **genitore** (*parent*), p
 - I nodi aventi p come genitore sono **figli** (*children*) di p
 - ... (*definizione da completare*)
- Ciascun nodo dell'albero può contenere un dato



Nota: non c'è limite al numero di figli di un nodo
(non è necessario che siano soltanto due!)

Rappresentazione di alberi

- Per convenzione estremamente diffusa, gli alberi vengono rappresentati graficamente in questo modo
 - Ogni nodo è rappresentato da una forma geometrica (cerchio, ovale, rettangolo, ...)
 - La **radice** si trova **più in alto** di tutti gli altri nodi
 - Ogni nodo si trova più in basso del proprio genitore
 - Quindi ogni genitore è più in alto di **tutti** i suoi figli
 - Ogni nodo è collegato al proprio genitore mediante un segmento o arco (non orientato, detto **ramo**)



- In pratica, è un albero con la radice in alto e i “rami” che si estendono verso il basso

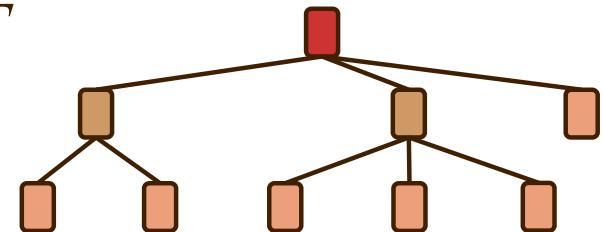


Alberi: UN ERRORE...

□ Integriamo la **definizione di albero** perché **il libro contiene un errore!!!**

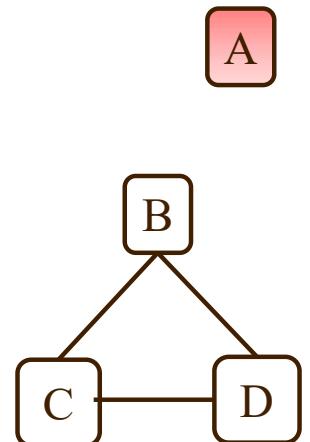
□ Un albero T è costituito da un insieme di **nodi** tra i quali esiste una **relazione gerarchica di tipo genitore/figli** rappresentata mediante rami è così definita:

- Se T non è vuoto, **uno e uno solo dei suoi nodi non ha genitore** e viene chiamato **radice (root)** di T
- **Ogni nodo diverso dalla radice ha un unico (nodo) genitore (parent), p**
 - I nodi aventi p come genitore sono **figli (children)** di p
- **L'insieme di nodi e rami, considerato come grafo, deve essere connesso!!!**
 - Senza utilizzare i grafi (che vedremo più avanti), il vincolo di connessione può essere espresso in questo modo: **Partendo da qualsiasi figlio e seguendo rami nella direzione che va da un figlio verso il suo genitore, si deve giungere alla radice.**



Alberi: UN ERRORE...

- **L'insieme di nodi e rami deve essere connesso!!!**
- Senza **questo** ulteriore vincolo, si potrebbe costruire un insieme di nodi e rami che rispetta la definizione del libro ma **NON** è un albero
 - L'insieme di nodi A, B, C e D, con i rami presenti in figura, sarebbe **ERRONEAMENTE** un albero
 - A è il nodo radice, l'unico privo di genitore
 - B ha C come genitore
 - C ha D come genitore
 - D ha B come genitore
 - Quindi tutti i nodi diversi dalla radice hanno uno e un solo genitore!
Ma questo non è un albero...

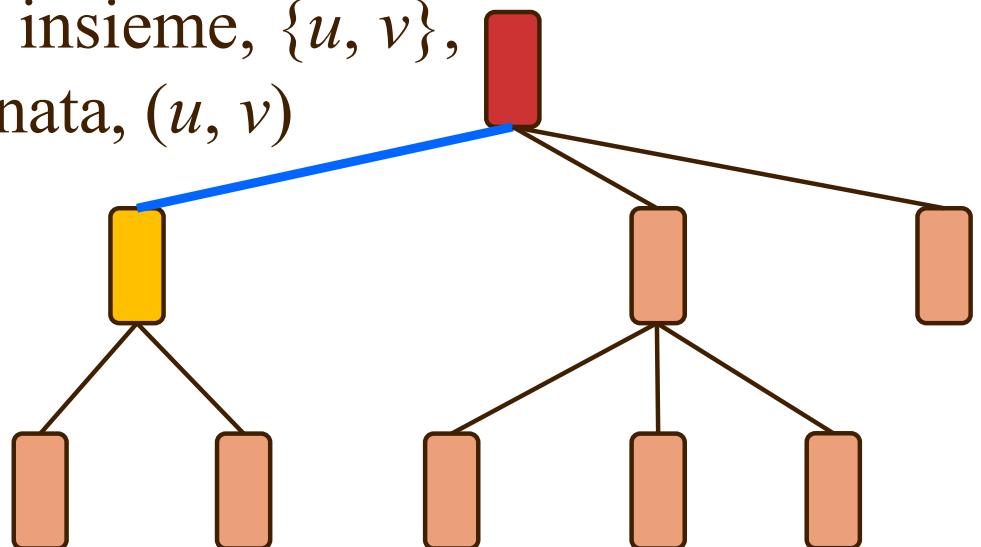


Una lista è un albero...

- La relazione precedente/successivo è **anche** una relazione di tipo genitore/figli, dove **ciascun genitore ha un unico figlio!**
- Le liste sono esempi ("degeneri") di alberi
- La prima posizione della lista è la radice dell'albero
- Si parla anche di "albero unario", per analogia con alberi binari, ternari, ecc., che vedremo

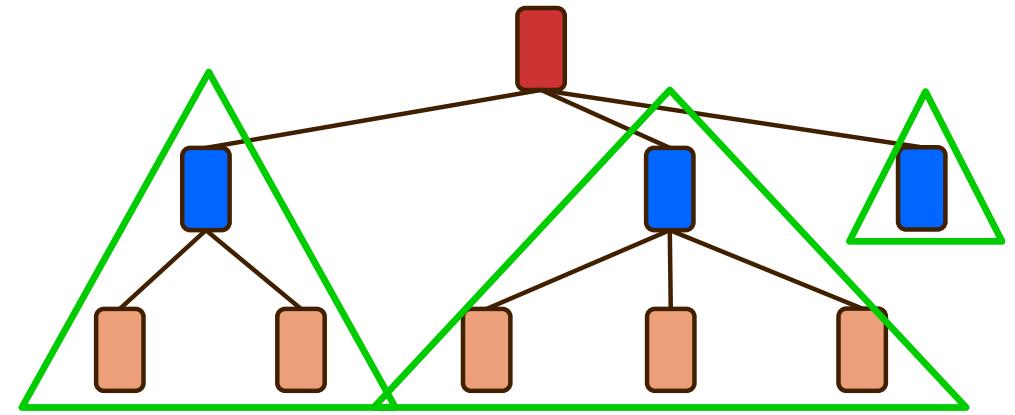
Ramo

- Un **ramo** (*edge*) dell'albero T è una coppia $\{u, v\}$ di nodi di T tale che u sia il genitore di v **o viceversa**
- **Dalla definizione** discende che un ramo è una coppia **non ordinata** di nodi tale che...
- Infatti usiamo il simbolo di insieme, $\{u, v\}$, e non quello di coppia ordinata, (u, v)



Alberi: definizione ricorsiva

- Un albero si può anche definire **ricorsivamente**
 - Un albero T
 - è vuoto **oppure**
 - è costituito da un nodo, r , detto **radice** di T , e da un insieme (eventualmente vuoto) di **alberi non vuoti**, le cui **radici** sono i figli di r
- A volte ci sarà utile la definizione ricorsiva, altre volte quella vista in precedenza
 - **(Si dimostra che)**
le due definizioni sono equivalenti



Ancora definizioni...

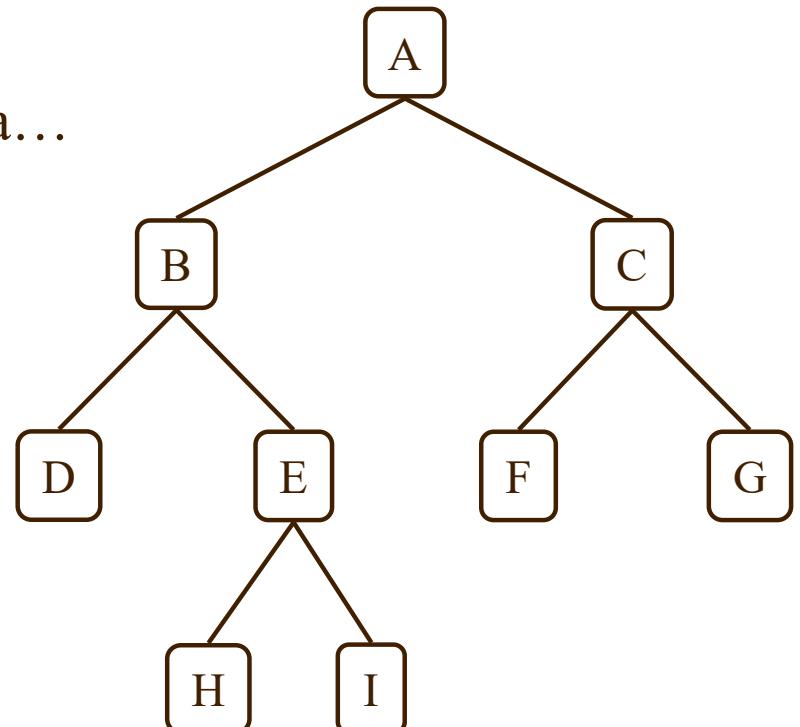
- Nodi figli dello stesso genitore si dicono **fratelli** (*sibling*)
- Un nodo avente almeno un figlio si dice **interno**
- Un nodo senza figli si dice **esterno** o **foglia** (*leaf*)

- Un albero non vuoto deve (ovviamente...) avere almeno una foglia

- Altrimenti sarebbe una struttura infinita... perché tutti i nodi avrebbero figli...

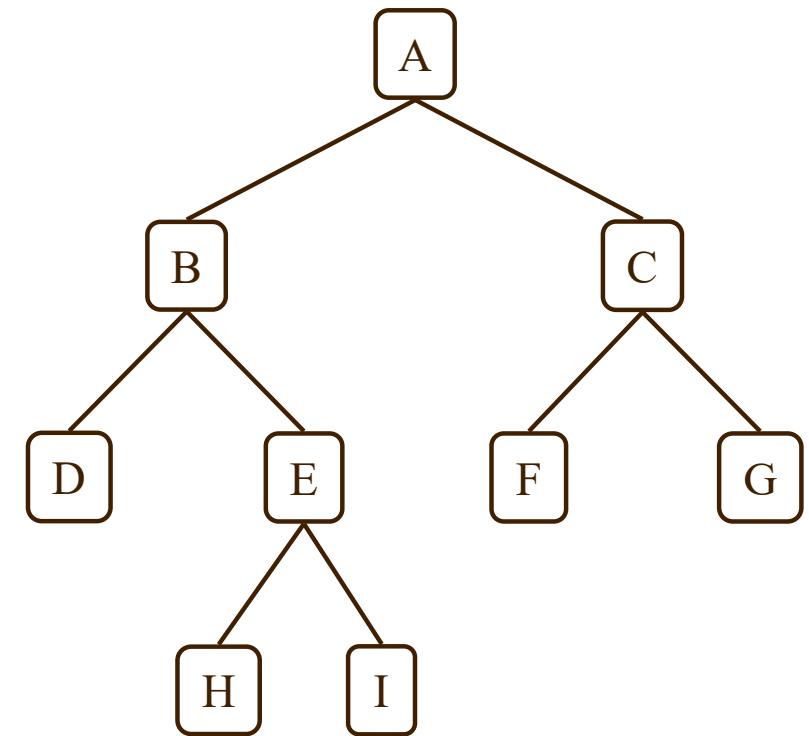
- Se un albero non vuoto non ha nodi interni, allora ha soltanto la radice (che, in tal caso, è una foglia)

- Se la radice di un albero è una foglia, allora non possono esserci altri nodi



Ancora definizioni...

- Un nodo a è un **antenato** (*ancestor*) del nodo n se e solo se
 - $a \equiv n$ oppure a è un antenato del genitore p di n
 - Quindi, dato che p (il genitore di n) è un antenato di se stesso, allora p è un antenato di n . E così via con il "nonno"...
- Un nodo d è un **discendente** (*descendant*) del nodo n se e solo se n è un antenato di d
 - Ogni nodo è antenato e discendente di se stesso!
Casi degeneri della definizione, ma comodi...

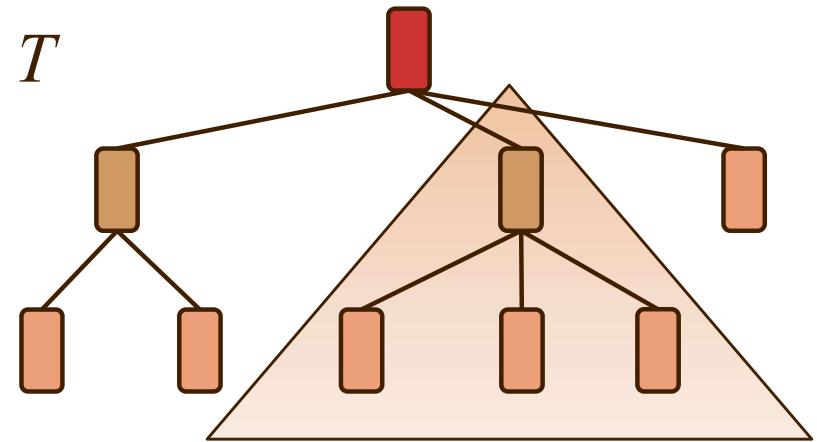


Chiamiamo "antenati propri" e "discendenti propri" gli antenati e i discendenti di un nodo, **con esclusione del nodo stesso**

Sottoalbero

□ Dato l'albero T , il suo **sottoalbero** (*subtree*) avente come radice $v \in T$ è l'albero costituito da **tutti e soli** i discendenti di v (tra i quali c'è, per definizione, v stesso)

- Un sottoalbero è, per costruzione, un **albero non vuoto** (dato che ha una radice...)
- Caso degenero: T è sottoalbero di T
- **Un albero T avente n nodi ha n sottoalberi diversi**
(perché ogni nodo di T è radice di un diverso sottoalbero)
- La radice di un sottoalbero non ha genitore all'interno del sottoalbero stesso, ma ce l'ha all'interno dell'albero (a meno che non sia l'unico sottoalbero che coincide con l'albero)



Albero ordinato

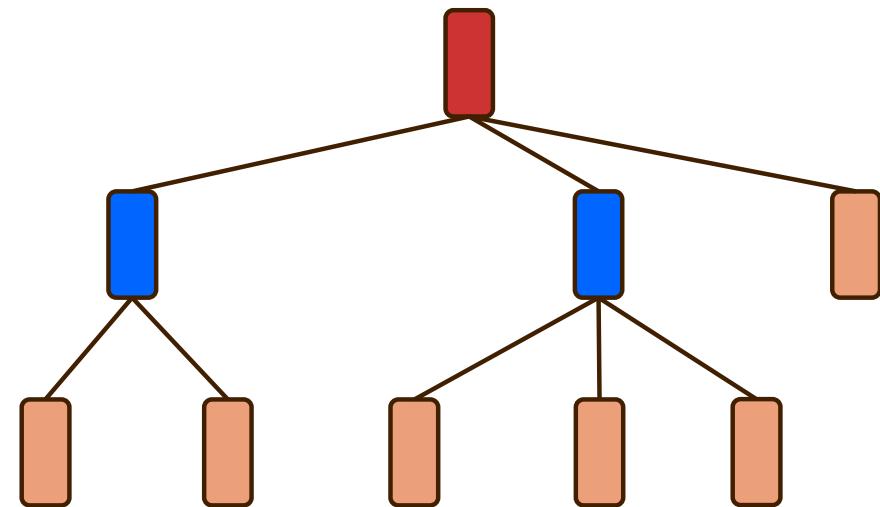
- Un albero è **ordinato** se, per ogni nodo, esiste una relazione posizionale tra i suoi figli, in modo che vi si possa identificare il primo, il secondo e così via
 - Nella rappresentazione grafica di un albero ordinato, i nodi figli di uno stesso genitore vengono disposti **da sinistra a destra** in base all'ordinamento
 - I figli di un nodo di un albero ordinato appartengono, quindi, a una lista
- Attenzione che questo "ordinato" è la traduzione di ***ordered***, **non di *sorted***... **è una proprietà topologica**, che non ha nulla a che vedere con i dati contenuti nei nodi (che possono anche non esserci...)

Grado

□ Il **grado** (*degree*) **di un nodo**

è il numero dei suoi figli

- Le foglie hanno grado zero

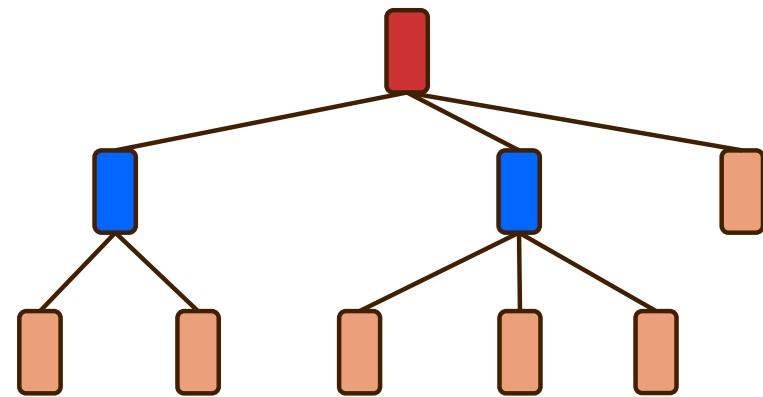


□ Il **grado di un albero** è il **massimo** grado dei suoi nodi

- Un albero di grado g si dice "albero n -ario" (es. albero unario, ternario, quaternario...)
 - Un **albero di grado due** si può anche chiamare **albero binario**, ma evitiamo per non fare confusione (come si capirà in seguito)
 - Ricordiamo: cos'è un albero unario (cioè di grado 1) ?
- Un albero di grado g **può** avere nodi interni con un numero di figli **minore** di g , basta che abbia almeno un nodo con g figli



Dimensione

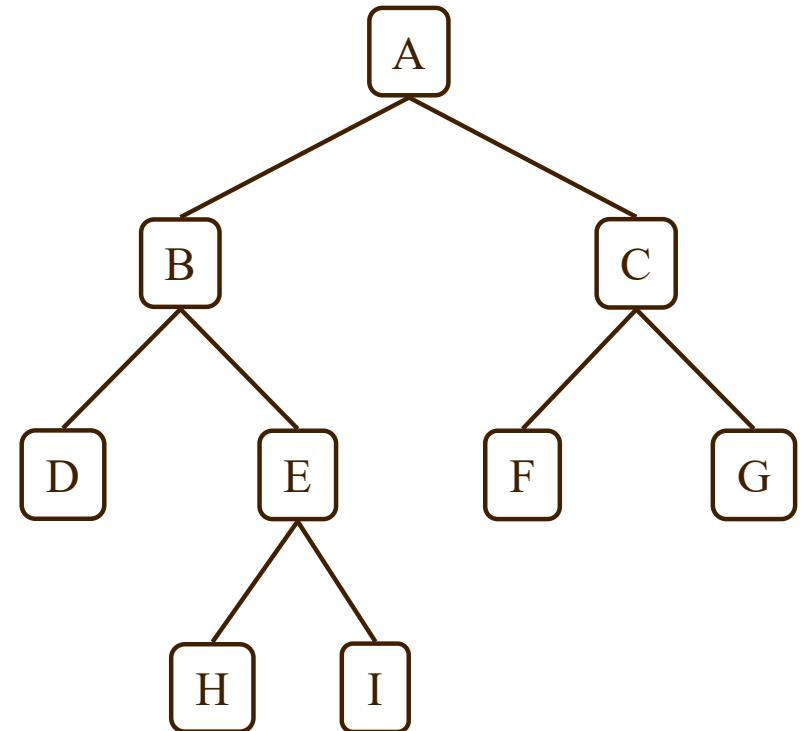


- La **dimensione** (*size*) di un albero è uguale al **numero dei suoi nodi**
- Definizione ricorsiva di dimensione di un albero
 - Se l'albero è vuoto, ha dimensione zero
 - Altrimenti l'albero ha dimensione uguale a uno più la somma delle dimensioni degli **(eventuali) sottoalberi che hanno come radice un figlio della radice**
 - Tali sottoalberi, d'ora in poi, verranno indicati come "**i sottoalberi della radice**", da non confondere con "il sottoalbero che ha come radice la radice stessa", cioè l'unico sottoalbero che coincide con l'albero

Percorso

- Un **percorso** (*path*) nell’albero T è una **sequenza posizionale** (cioè una lista!) di nodi di T tale che **ogni coppia** di nodi **consecutivi** della sequenza sia un ramo di T

- Es. (E, B, A) è un percorso, perché $\{E, B\}$ e $\{B, A\}$ sono rami
- Es. (A, B, F) **non** è un percorso, perché $\{B, F\}$ non è un ramo
- Es. (E, B, D, B, A) è un percorso, perché $\{E, B\}$, **{B, D}**, **{D, B}**, $\{B, A\}$ sono rami (**alcuni rami possono ripetersi!**)
- La **lunghezza di un percorso** è uguale al **numero di rami** che lo compongono
(ed è la lunghezza della lista di nodi – 1)
 - (E, B, A) ha lunghezza 2
 - (E, B, D, B, A) ha lunghezza **4**
- (ovviamente) **qualsiasi ramo è un percorso** di lunghezza unitaria e **qualsiasi nodo è un percorso** di lunghezza zero



Lezione 15

**Intermezzo
Pubblicitario!**

**Gli alberi sono
super!**

Ma a cosa servono gli alberi?

- Servono a rappresentare i dati in modo da **semplificare** la progettazione di **algoritmi più efficienti** di quelli che si potrebbero progettare usando strutture lineari, come le liste
- **Gli alberi non sono strutture necessarie!**
Come NON lo sono le pile, le code, le liste concatenate...
- (Si dimostra che) **Si può risolvere qualunque problema computazionale anche usando soltanto array**
 - D'altra parte, la memoria fisicamente disponibile nei calcolatori non è altro che un grande array, non ci sono memorie ad albero...
 - È **solo** una questione di **efficienza** (in termini di **tempo** e/o di **spazio**), ma spesso **decisamente rilevante**

**Anticipazione:
Un esempio di
alberi molto utili!**

Problema: Ricerca di un dato in un insieme

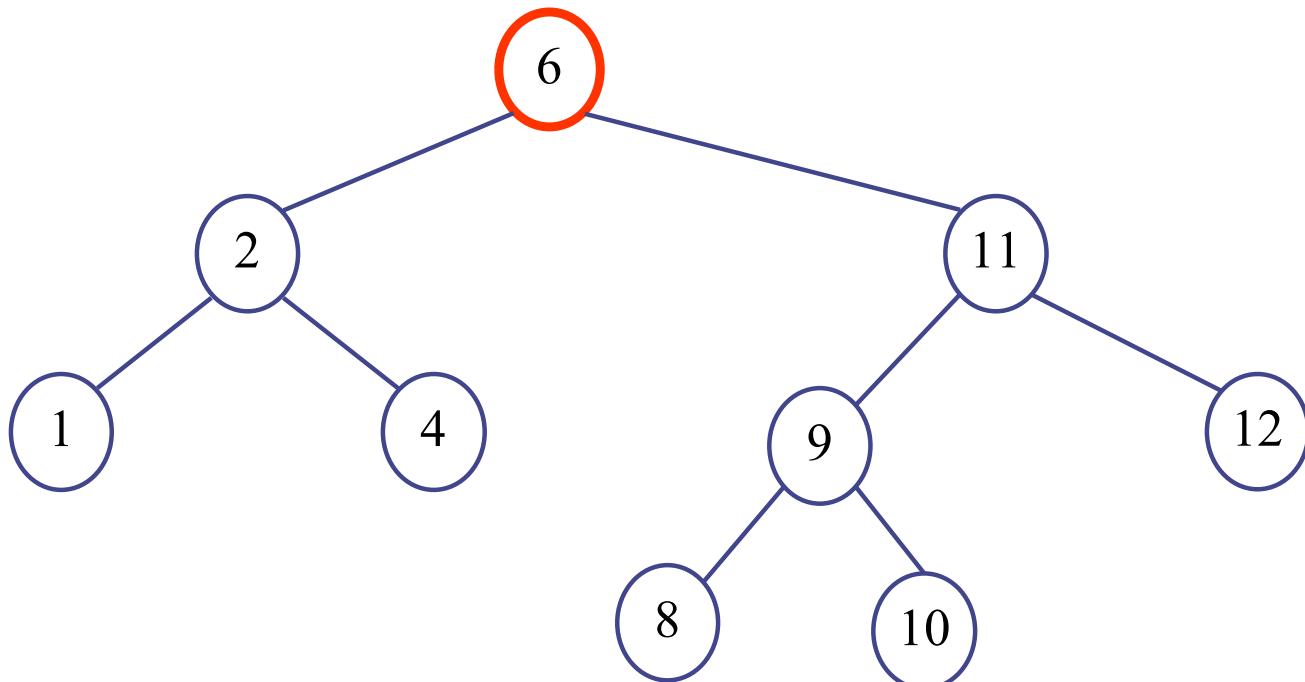
- **È un problema ESTREMAMENTE diffuso**
e può essere risolto in molti modi diversi
- **Catena non ordinata:** ricerca/rimozione $O(n)$, inserimento $O(1)$
- **Array non ordinato** (non ha senso, meglio la catena): ricerca/rimozione $O(n)$, inserimento $O(1)$ **in media**
- **Array ordinato:** **ricerca $O(\log n)$,**
inserimento/rimozione $O(n)$, particolarmente interessante se l'insieme non subisce modifiche
- **Catena ordinata** (non ha senso, meglio qualunque altra soluzione): ricerca/inserimento/rimozione $O(n)$

Problema: Ricerca in un insieme

- Quale soluzione è migliore?
Dipende dalla **frequenza delle diverse operazioni**
(ricerca, inserimento, rimozione)
- Se si fanno **soprattutto ricerche, meglio l'array ordinato** (ricerche $O(\log n)$, ma modifiche $O(n)$)
- Se si fanno **soprattutto inserimenti, meglio la catena non ordinata** (inserimenti $O(1)$, ma ricerche $O(n)$)
- Molte applicazioni, però, non hanno un'operazione molto più frequente delle altre
 - Sarebbe meglio avere prestazioni simili per le tre operazioni, **possibilmente $O(\log n)$... che è "quasi" $O(1)$!**

Problema: Ricerca in un insieme

□ L'**albero** binario di ricerca bilanciato (AVL), che vedremo, ha **prestazioni $O(\log n)$** per **tutte le operazioni** (ricerca, inserimento e rimozione), un risultato MOLTO importante



**Fine della
pubblicità** ☺

L'albero come ADT

L'albero come ADT (**Tree**)

- Il tipo di dato astratto “albero” (**Tree**, **non presente nella libreria Java**) memorizza i dati in “posizioni”, rappresentate da nodi e definite in modo da rispettare la relazione genitore/figlio che caratterizza l’albero stesso
 - Useremo spesso **indifferentemente** i termini “nodo” e “posizione”, ma ricordiamo che le posizioni sono (interfacce) astratte, mentre i nodi sono esemplari di classi, entità concrete
- Per rappresentare una posizione all’interno di un albero usiamo la stessa interfaccia usata per le liste
 - Ci serve soltanto **accedere** all’elemento memorizzato nella posizione; **la relazione tra posizioni all’interno dell’albero è gestita dall’albero**, non dalle singole posizioni
 - **Una posizione** **“non sa dove si trova”** **(ma un nodo lo sa...)**

Esattamente
come
avviene
nelle liste

```
public interface Position<T>
{   T element();
}
```

Interfaccia Tree

- Un’interfaccia “albero” si deve occupare di **tre aspetti**
 - **(1)** Gestione dell’albero come **contenitore**:
isEmpty() e **size()** (cioè numero di nodi)
 - Gestione della **relazione gerarchica** genitore/figlio rappresentata dall’albero (eventualmente ordinato)
 - **(2)** Gestione delle **ispezioni** alla relazione
 - Navigazione nell’albero e ispezione dei dati dei nodi
 - **(3)** Gestione delle **modifiche** alla relazione
 - Inserimento di nuovi nodi,
(eventuale) rimozione di nodi esistenti,
(eventuale) modifica delle relazioni “parentali” di un nodo
- Dato che gli alberi si usano per molte **diverse applicazioni**,
lasciamo a interfacce più specifiche la definizione di (3)

Interfaccia Tree

```
public interface Tree<T> extends Container
{ Position<T> root();
  Position<T> parent(Position<T> v);
  ...
}
```

□ Ispezioni alla struttura dell'albero

- **root()** fornisce il nodo radice
 - Lancia **EmptyTreeException** se l'albero è vuoto
 - Consente di ottenere una **Position** di partenza per la navigazione
 - Analogo al metodo **first()** di **PositionList**
- **parent(v)** fornisce il nodo genitore di **v**
 - Lancia **BoundaryViolationException** se **v** è la radice
 - Lancia **InvalidPositionException** se **v** non è valida

Interfaccia Tree

```
public interface Tree<T> extends Container
{   Iterable<Position<T>> children(Position<T> v) ;
    int numChildren(Position<T> v) ;
    ...
}
```

□ **children (v)** fornisce una lista (eventualmente vuota) contenente i figli di **v**

- Lancia **InvalidPositionException** se **v** non è valida
- Perché restituisce **Iterable<Position<T>>** e non **Iterable<T>**?
 - Se vogliamo **navigare nella struttura**, ci servono **LE POSIZIONI** dei figli di un nodo, **non i dati** contenuti in essi: **una lista di posizioni**
- Se l'albero è (topologicamente) **ordinato**, la lista restituita da **children** è (topologicamente) ordinata di conseguenza

Figli di un nodo

```
class TreeNode<T> implements Position<T>
{
    ...
    private Iterable<Position<T>> children;
}
```

□ Oppure **Iterable**<**TreeNode**<T>> **children**;

- Se uso una lista di **TreeNode**, il metodo **children** di **Tree** dovrà copiare la lista per poterla fornire all'esterno, perché all'esterno il tipo **TreeNode** non è noto (è una classe interna privata)
- Se uso una lista di **Position**, posso restituirla all'esterno senza doverla copiare (tanto non è modificabile, è solo “iterabile”), ma usarla all'interno è più scomodo
 - Ogni volta che uso un figlio devo fare down-casting per recuperare il nodo partendo da una posizione

Figli di un nodo

- Si potrebbe pensare
 - Per **children** uso una variabile di esemplare di tipo **Iterable<TreeNode<T>>** e la restituisco (senza copiarla) sotto forma di riferimento di tipo **Iterable<Position<T>>**
- Sembra ragionevole, perché un riferimento di tipo **TreeNode** può essere assegnato a una variabile di tipo **Position**, dato che **TreeNode implements Position**
 - Invece con le classi generiche non funziona

NO → **ArrayList<Position> x = new ArrayList<TreeNode>();**

- Funziona soltanto con gli array

SI → **Position[] x = new TreeNode[10];**

Interfaccia Tree

```
public interface Tree<T> extends Container
{ boolean isRoot(Position<T> v);
  boolean isInternal(Position<T> v);
  boolean isExternal(Position<T> v);
  ...
}
```

- Metodi non necessari ma utili come **condizioni per navigare nell'albero**
- Lanciano **InvalidPositionException** se **v** non è una posizione valida

```
public interface Tree<T> extends Container
{ // int size();
  // boolean isEmpty();
  ...
}
```

Interfaccia Tree

```
public interface Tree<T> extends Container
{   Iterator<T> iterator();
    Iterable<Position<T>> positions();
    ...
}
```

- **iterator()** fornisce un iteratore che opera su una lista (eventualmente vuota, se l'albero è vuoto) contenente **i dati** presenti nell'albero, in ordine non specificato
- **positions()** fornisce una lista (eventualmente vuota, se l'albero è vuoto) contenente **le posizioni** dei nodi dell'albero, in ordine non specificato
- Solitamente un albero NON conserva al proprio interno **una lista** delle proprie posizioni, né dei relativi dati, quindi entrambi questi metodi **operano in “modalità fotografia”**... cioè il costruttore dell'iteratore crea una lista che poi verrà scandita dall'iteratore
(quindi attenzione al tempo di costruzione... e allo spazio occupato...)



Interfaccia Tree

```
public interface Tree<T> extends Container
{  T replace(Position<T> v, T element);
   ...
}
```

- **replace** consente di modificare il dato contenuto in un nodo, ovviando alla limitazione dell’interfaccia **Position**, che consente soltanto l’ispezione
 - È come **set** di **PositionList**
 - Lancia **InvalidPositionException** se **v** non è una posizione valida
- Non è una semplice “ispezione”, però non altera la struttura delle relazioni gerarchiche rappresentate dall’albero: accettiamo il metodo in questa categoria

Interfaccia Tree

```
public interface Tree<T> extends Container  
{   Iterator<T> iterator();  
    ...  
}
```

- La presenza di questo metodo consente di definire **Tree** in questo modo

```
public interface Tree<T> extends Container,  
    Iterable<T>
```

quindi lo facciamo

- Aggiungendo tale definizione, un metodo che, ad esempio, calcola il valore minimo/medio/massimo in una lista “iterabile” di numeri oppure fa una ricerca sequenziale, può operare anche su un albero!! Molto comodo!!

```
// RIASSUMENDO... ma non è ancora utilizzabile
// perché non ci sono metodi di inserimento!!
public interface Tree<T> extends Container, Iterable<T>
{
    Position<T> root()
        throws EmptyTreeException;
    Position<T> parent(Position<T> v)
        throws InvalidPositionException,
               BoundaryViolationException;
    Iterable<Position<T>> children(Position<T> v)
        throws InvalidPositionException;
    int numChildren(Position<T> v)
        throws InvalidPositionException;
    boolean isRoot(Position<T> v)
        throws InvalidPositionException;
    boolean isInternal(Position<T> v)
        throws InvalidPositionException;
    boolean isExternal(Position<T> v)
        throws InvalidPositionException;
    Iterator<T> iterator();
    Iterable<Position<T>> positions();
    T replace(Position<T> v, T element)
        throws InvalidPositionException;
    int size();
    boolean isEmpty(); }
```

Useremo
queste
primitive
anche nello
pseudocodice

Il problema delle
posizioni valide va
risolto **come nelle liste**

Lezione 16

Progettazione di alberi

Progettazione di alberi

- Una **struttura concatenata** è forse la realizzazione di **albero** più naturale (ma non unica! vedremo...)

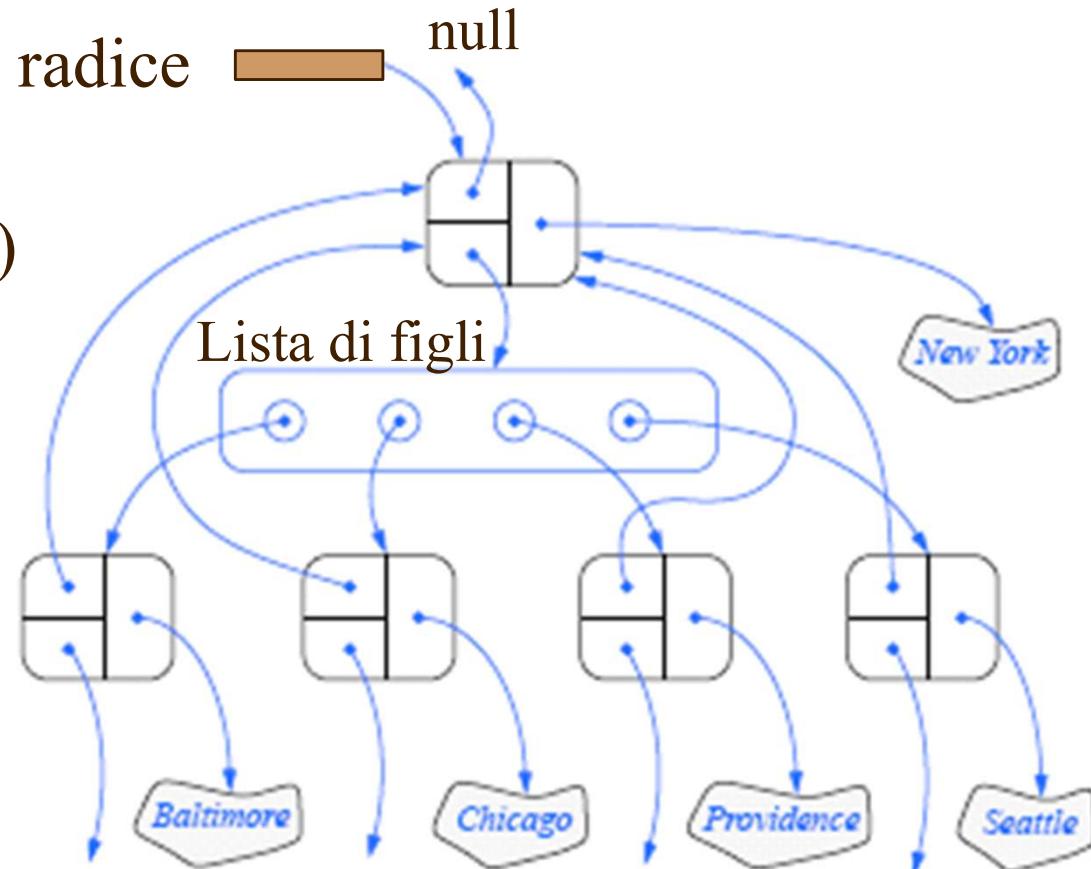
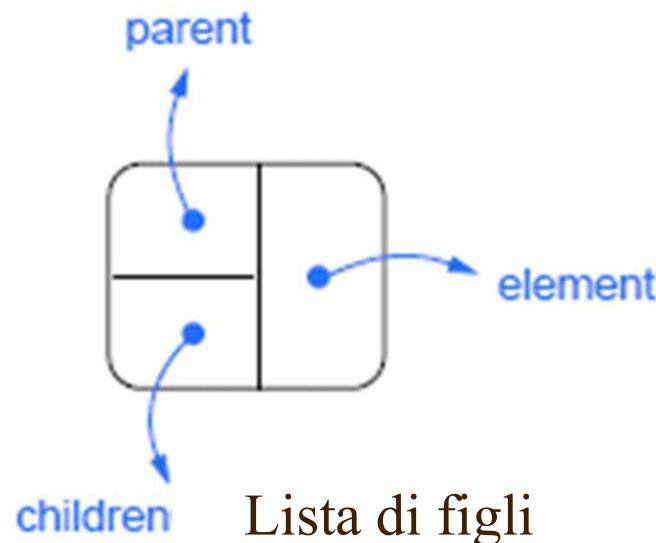
- `class LinkedTree<T> implements Tree<T>`

- Variabili di esemplare dell'albero

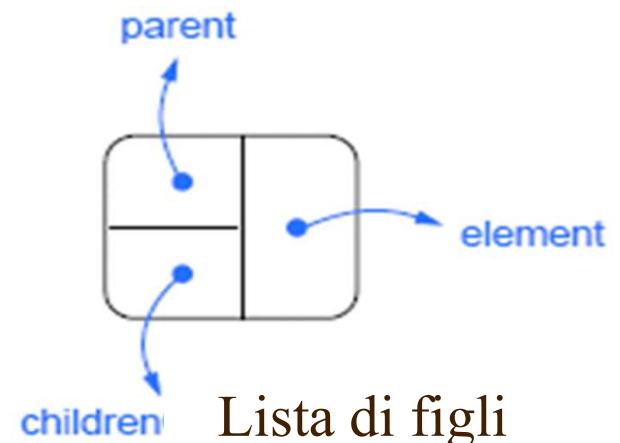
- Riferimento al nodo radice



- **Numero di nodi**, così **size()** è $\Theta(1)$



Nodo di un albero



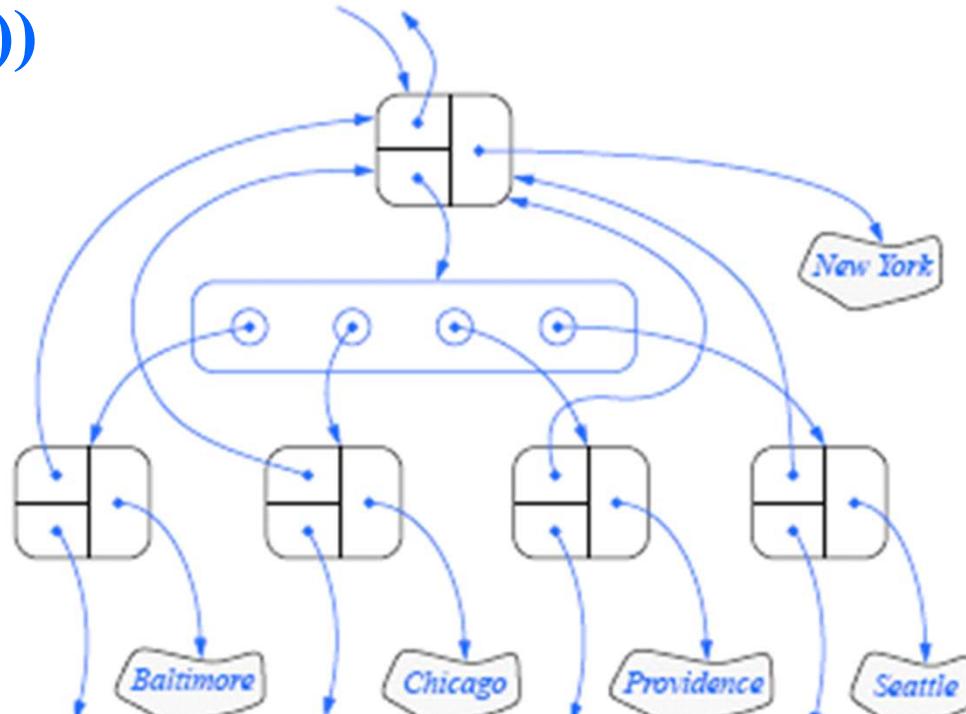
```
// classe tipicamente interna all'albero e privata
class TreeNode<T> implements Position<T>
{ private T element;
  private TreeNode<T> parent; // null nella radice
  private LinkedList<TreeNode<T>> children;
    // oppure ArrayList o altro...
  public T element() { return element; }
  // costruttore...

  // get/set parent...

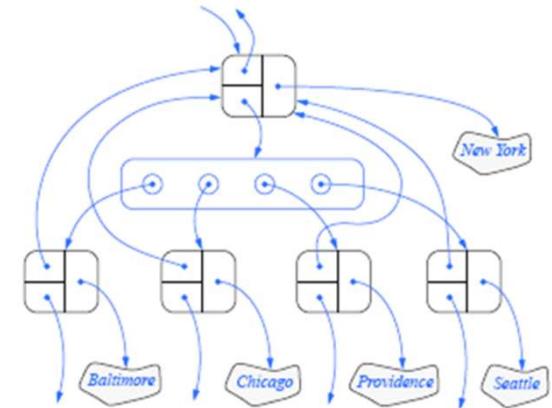
  // setElement...

  // metodi di gestione della lista di figli...
}
```

Progettazione di alberi

- Con questa realizzazione concatenata, tutte le operazioni di **Tree** sono **$\Theta(1)$, tranne**
 - **positions** e **iterator** che sono **$\Theta(n)$** , dove n è il numero di nodi dell'albero (**$\Theta(n)$ da dimostrare... però è ovvio che siano $\Omega(n)$**)
 - **children(v)** che è **$\Theta(g_v)$** , dove g_v è il grado di v ($\Theta(1)$ se restituiamo un riferimento diretto alla lista dei figli anziché un suo clone)
- Vedremo più avanti come si possano realizzare **questi metodi**, usando un **attraversamento**
- 

Progettazione di alberi



- L'occupazione di spazio di un albero contenente n dati e realizzato in questo modo è $\Theta(n)$

- Un nodo per ogni dato (quindi n nodi), più
- Due variabili di esemplare dell'albero, più
- Le liste dei figli...
la cui dimensione **totale** è uguale a $n - 1$

- Infatti, **quanti sono i figli presenti in un albero di dimensione n ?**

Ogni nodo dell'albero, **tranne la radice**, è figlio di un altro nodo, quindi i figli sono “tutti i nodi tranne uno”, cioè $n - 1$

- Quindi, proprietà da ricordare

$$\sum_{v \in T} g_v = \dim(T) - 1$$

Profondità
di un nodo

Profondità di un nodo

□ La **profondità** (*depth*) d_v di un nodo $v \in T$ è il numero di **antenati propri di v**

- Quindi, la radice ha profondità 0

□ Alternativa:

Definizione ricorsiva della profondità del nodo $v \in T$

- If $T.\text{isRoot}(v)$

- $d_v = 0$

- Else

- $d_v = 1 + d_{T.\text{parent}(v)}$

Si può dimostrare
che le due
definizioni sono
equivalenti

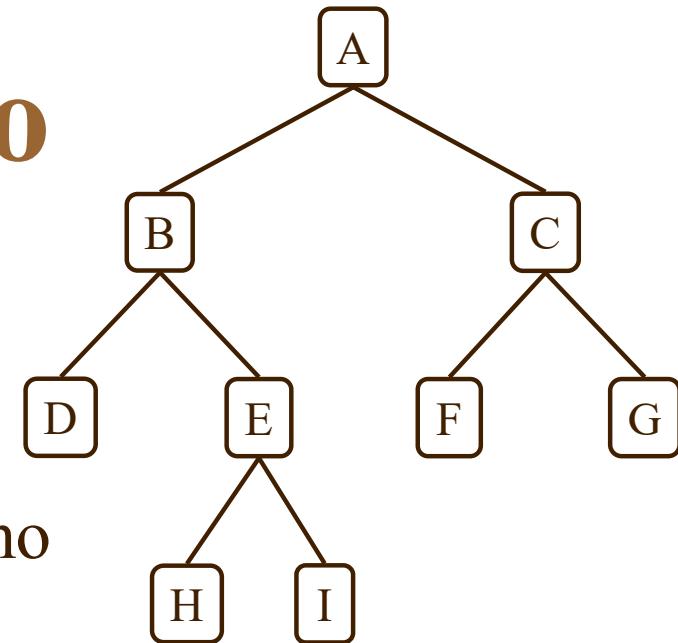
Profondità di un nodo

(H, E, B, A) è **il** percorso minimo
che collega H alla radice

(H, E, **B**, **D**, **B**, A) è **un** percorso **non** minimo
che collega H alla radice

□ Si può dimostrare che il percorso costituito da tutti e soli gli antenati di un nodo è il più breve percorso che collega il nodo alla radice (ed è unico)

- La lunghezza di tale percorso si dice **distanza dalla radice**
 - Quindi, **la profondità di un nodo è la sua distanza dalla radice** (distanza che, infatti, è zero se il nodo è la radice)
- In generale, la **distanza tra due nodi** è la lunghezza del più breve percorso che li collega.
 - Attenzione: anche se in alcuni casi lo è, **non sempre la distanza tra due nodi è la somma delle loro profondità!** es. (D, B, E, H))

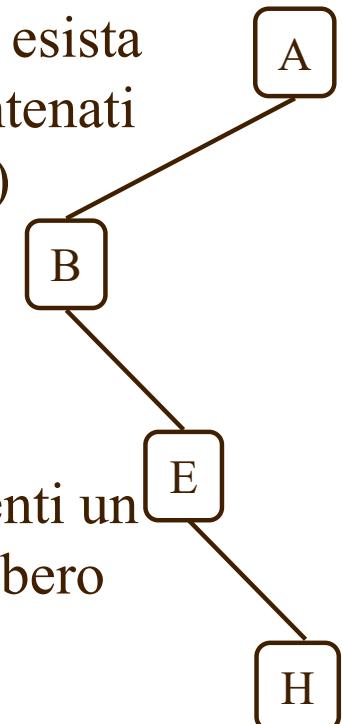


Profondità di un nodo

□ **Proprietà:** In un albero non vuoto di dimensione n , la profondità massima di un nodo è minore di n

- Dimostrazione: Ipotizzo per assurdo che, in un albero con n nodi, esista un nodo con profondità $n + k$ (con $k \geq 0$), che, quindi, ha $n + k$ antenati propri. Quindi, l'albero ha (almeno) $n + k + 1 > n$ nodi (**assurdo**)
- In un albero di grado uno, la sua unica foglia ha profondità $n - 1$
 - Dimostrare che alberi di grado maggiore di uno non possono avere nodi con profondità $n - 1$
- Quindi, $\forall n > 0$ esistono alberi di dimensione n (e grado uno) aventi un nodo con profondità $n - 1$, mentre non è possibile che esista un albero di dimensione n avente un nodo con profondità maggiore di $n - 1$
- Abbiamo dimostrato che

$$\max_{v \in T \neq \emptyset} d_v < n$$



Naturalmente, in uno specifico albero la profondità massima dei nodi può essere inferiore a $n - 1$: infatti, lo sarà in ogni albero di grado maggiore di uno

Ricordare: metodo generico in una classe non generica

Grazie ai metodi di navigazione, questo metodo può essere realizzato FUORI dalla classe “albero”

Profondità di un nodo

```
public static <T> int depth(Tree<T> t, Position<T> v)
{   if (t == null) throw new InvalidArgumentException();
    if (t.isRoot(v)) // se v non è valida...
        return 0;
    else // clausola else inutile... sopra c'è return...
        return 1 + depth(t, t.parent(v)); // ricorsivo
}
```

□ Che prestazioni ha il metodo a ricorsione semplice **depth**?

L’operazione elementare di costo unitario per gli alberi è l’attraversamento di un ramo (in questo caso, **t.parent(v)**).

Esegue un’invocazione ricorsiva per ogni ramo presente nel percorso minimo che va dal nodo in esame alla radice, quindi, se il nodo v ha profondità d_v , il metodo ha prestazioni $\Theta(d_v)$ (e richiede uno spazio $\Theta(d_v)$, pari alla profondità di ricorsione)

□ Nel caso peggiore il metodo è **$O(n)$** perché (ricordiamo che) $\max_{v \in T \neq \emptyset} d_v < n$

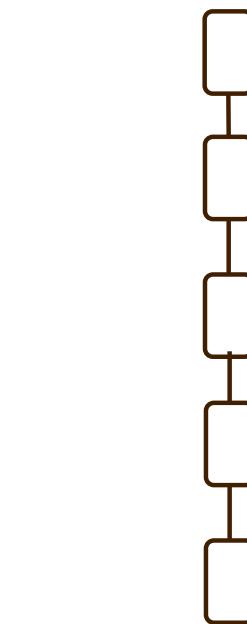
Ovviamente si può fare anche iterativo

Per la progettazione di algoritmi

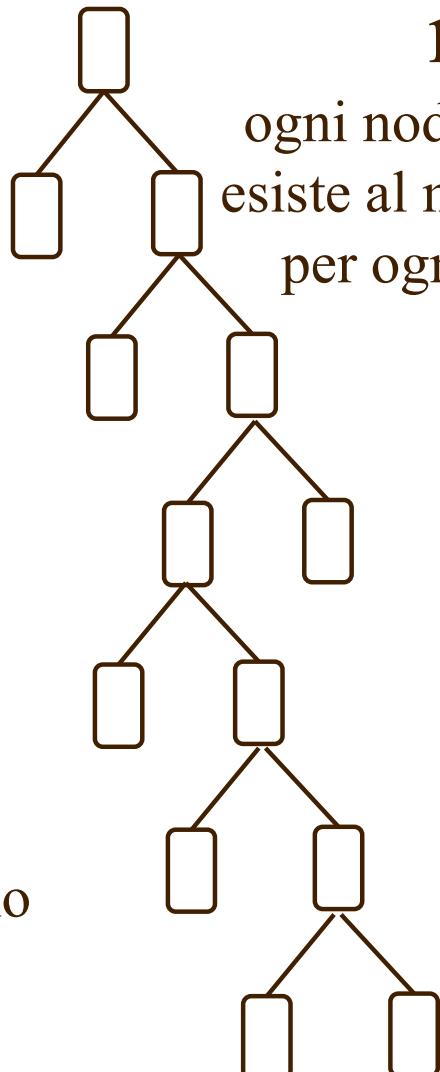
- Attenzione: **in generale, non è consentito che un algoritmo memorizzi informazioni nei nodi dell'albero** (che contengono già i dati per i quali l'albero viene utilizzato...), tranne quando questa azione sia esplicitamente consentita
- **Negli algoritmi si possono utilizzare soltanto i metodi dell'interfaccia Tree vista a lezione**
(a meno che l'esercizio non consenta l'utilizzo di metodi ulteriori) e altri metodi che elaborano alberi e che sono stati presentati a lezione, come **depth**, specificando con chiarezza quale versione si utilizza e che prestazioni ha

Per dimostrazioni/algoritmi su alberi

- Spesso è utile considerare alcuni alberi "strani"...

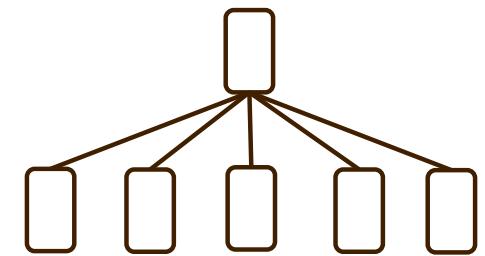


catena
albero di grado uno



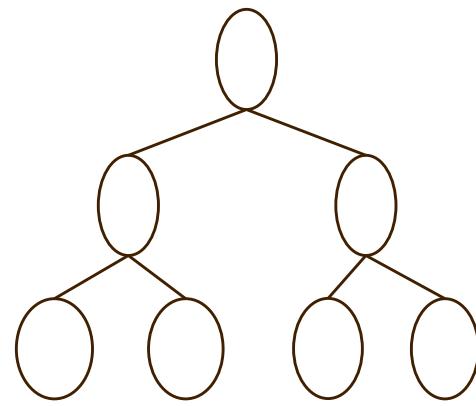
millepiedi

ogni nodo interno ha grado 2 ed
esiste al massimo un nodo interno
per ogni valore di profondità



pettine

albero con profondità
massima uguale a uno

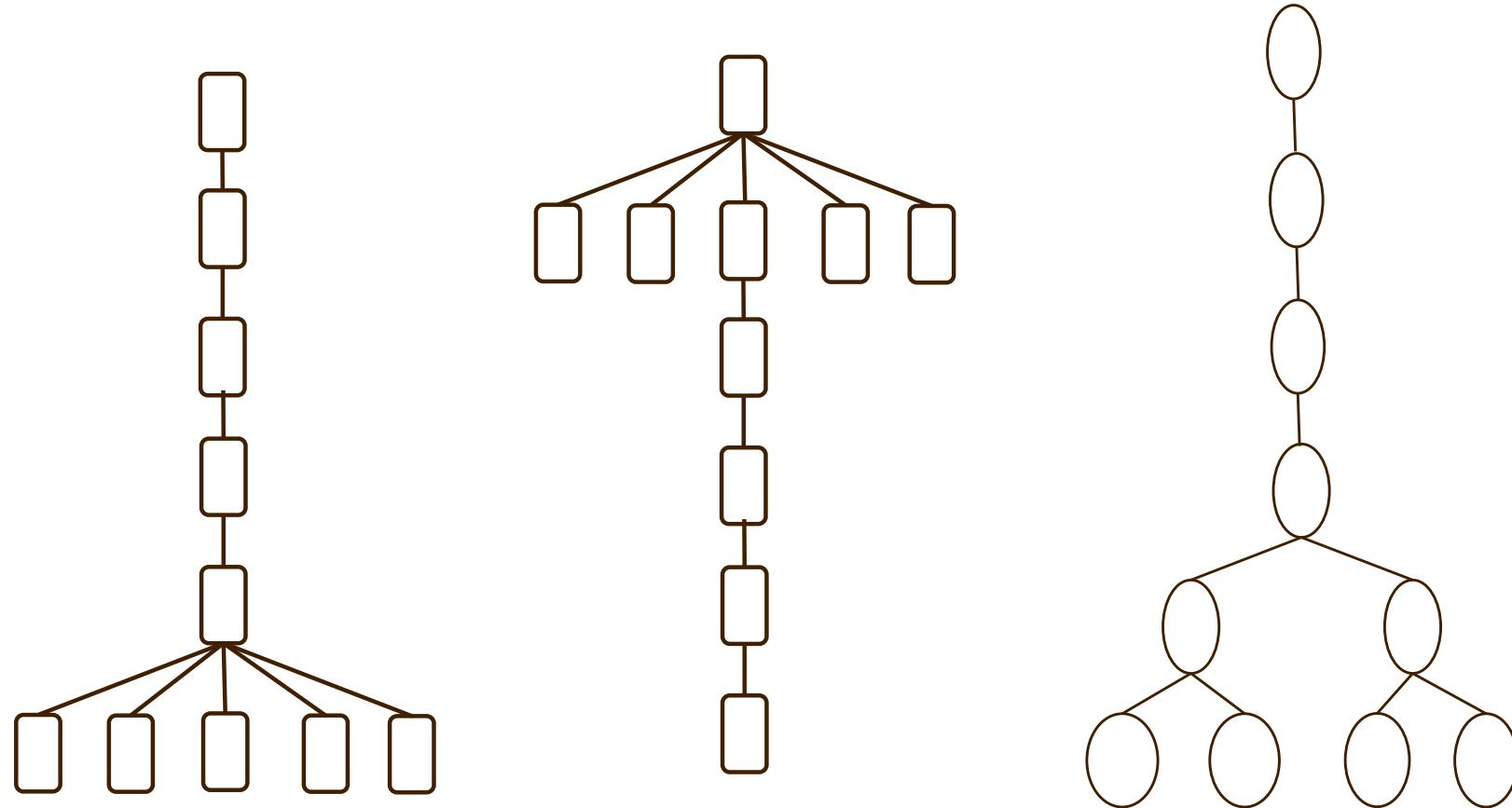


albero triangolare

ogni nodo interno ha grado uguale al grado
dell'albero (anche diverso da due) e tutte
le foglie hanno la stessa profondità
(catena e pettine sono anche
triangolari... di grado 1 e $n - 1$)

Per dimostrazioni/algoritmi su alberi

□ Oppure loro combinazioni...



Dubbio...

- Potrebbe essere comodo inserire una "dimensione" in ogni nodo, che contenga la dimensione del sotto-albero di cui quel nodo è radice?
- Certamente potrebbe essere comodo, ma quando si aggiungono informazioni di stato (cioè "di esemplare") bisogna sempre valutare con attenzione quanto "costa" **tenerle aggiornate...** [oltre allo spazio occupato]
- In questo caso, quando inserisco un nuovo nodo oppure elimino un nodo, dovrei poi aggiornare la "dimensione" di tutti gli antenati... un'operazione $O(n)$
- **È un ragionamento da fare per ogni informazione aggiuntiva che si può ipotizzare di memorizzare in ciascun nodo**

Lezione 17

Altezza
di un nodo
e di un albero

Altezza di un nodo e di un albero

- L'**altezza** (*height*) h_v **di un nodo**

$v \in T$ è così definita (ricorsivamente)

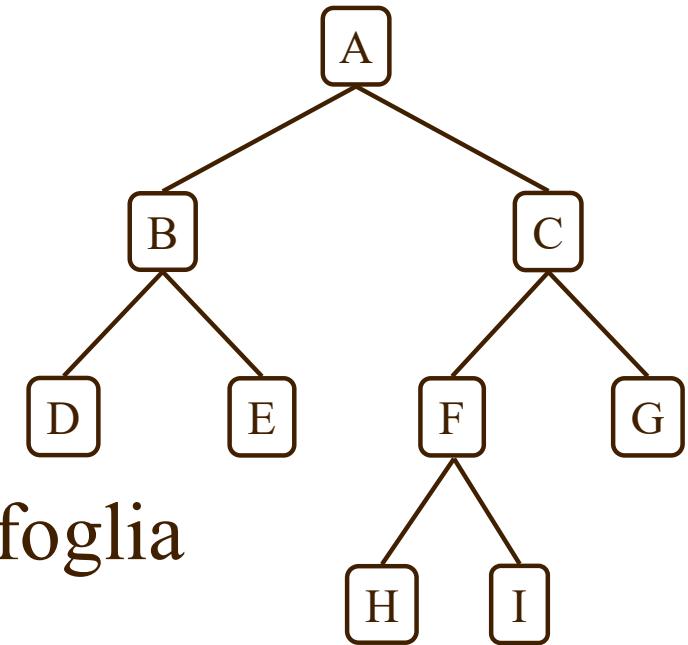
if $T.\text{isExternal}(v)$ $h_v = 0$ // v è una foglia

else $h_v = 1 + \max_{c \in T.\text{children}(v)} h_c$

Osserviamo che dalla definizione discende la proprietà: **$h_v = 0$ se e solo se v è una foglia**

- Si definisce **altezza** h_T **di un albero**
 T non vuoto l'altezza della sua radice,
mentre l'altezza di un albero vuoto si definisce
uguale a zero (come quella di un albero che abbia
soltanto la radice, che in tal caso è una foglia)

- La radice ha sempre altezza massima e profondità minima (= 0)



ATTENZIONE:

nodi aventi la stessa
profondità possono avere
altezze diverse (es. B C)
e, viceversa, nodi aventi
la stessa altezza possono
avere profondità diverse
(es. B F)

Altezza di un nodo e di un albero

- Prestazioni del metodo **height(*v*)** che calcola l'altezza di un nodo seguendo direttamente la definizione:

```
if T.isExternal(v) return 0  
return 1 + maxc ∈ T.children(v) height(c)
```

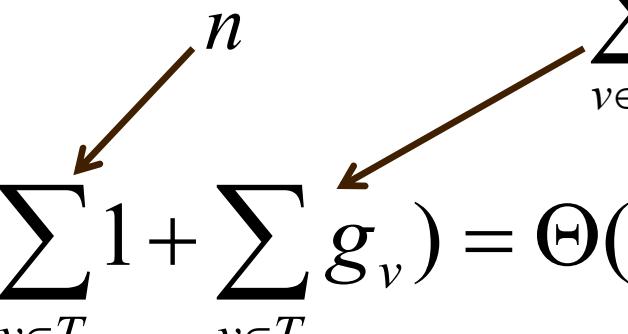
- In conseguenza della struttura dell'albero, l'invocazione ricorsiva avviene **al massimo una volta per ciascun nodo**, infatti:
 - il metodo viene invocato per un figlio soltanto in seguito all'invocazione per il suo genitore, che è unico
 - l'algoritmo “scende” sempre verso figli, non “risale” mai (infatti non c’è nessuna invocazione di **parent**)
- Quindi il **caso pessimo** è il calcolo dell'altezza della radice (cioè dell'altezza dell'albero), che invoca ricorsivamente l'algoritmo (una e una sola volta) **per ogni nodo**
 - Calcoliamo, ora, il tempo speso nell'esecuzione dell'algoritmo **per un nodo**, escludendo le conseguenti invocazioni ricorsive

Altezza di un nodo e di un albero

- if $T.\text{isExternal}(v)$ return 0 else return $1 + \max_{c \in T.\text{children}(v)} \text{height}(c)$
- Per ogni nodo v , l'algoritmo **height**(v) richiede un tempo d'esecuzione suddiviso in due porzioni (oltre alle invocazioni ricorsive)
 - Una porzione di tempo costante, $\Theta(1)$
 - Una porzione di tempo proporzionale al numero g_v di figli di v , cioè un tempo $\Theta(g_v)$, per l'invocazione di **children**(v) e la scansione della lista così ottenuta (anche se **children** fosse $\Theta(1)$)
- Il tempo totale richiesto per l'esecuzione del metodo **height** sulla **radice** di un albero di dimensione n (che provoca l'invocazione di **height** su tutti i nodi) è, quindi:

$$\Theta\left(\sum_{v \in T} (1 + g_v)\right) = \Theta\left(\sum_{v \in T} 1 + \sum_{v \in T} g_v\right) = \Theta(n + (n - 1)) = \Theta(n)$$

Ricordiamo che...
$$\sum_{v \in T} g_v = n - 1$$



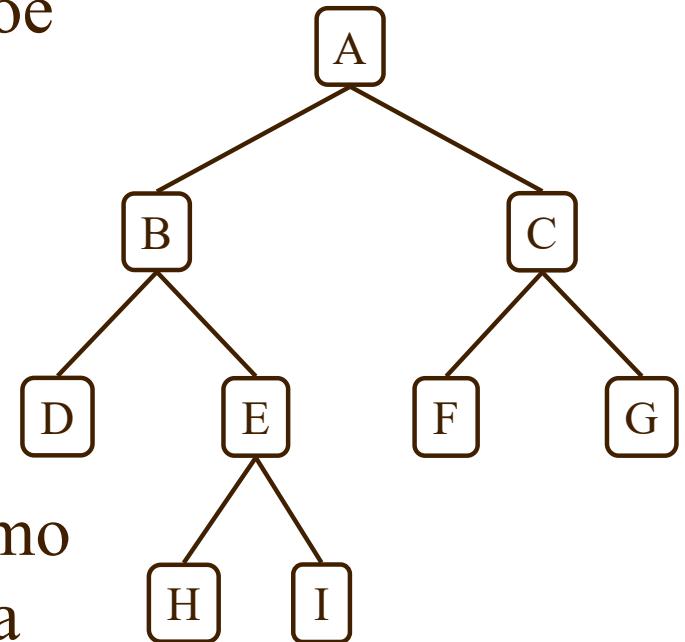
Altezza di un nodo e di un albero

- Abbiamo visto, quindi, che il tempo totale per l'esecuzione del metodo **height** sulla radice di un albero di dimensione n (che provoca l'invocazione di height su tutti i nodi) è $\Theta(n)$
- Più in generale, il tempo necessario per l'esecuzione del metodo **height** applicato a un nodo v è **$\Theta(\dim(S(v)))$** , dove $S(v)$ è il sottoalbero avente radice v , perché **il calcolo dell'altezza di un nodo coinvolge soltanto i discendenti di quel nodo**, che appartengono al sottoalbero di cui è radice

Proprietà

- Si può dimostrare (ad esempio **per induzione**... anche se non è semplice) che **l'altezza di un albero non vuoto è uguale alla massima profondità dei suoi nodi esterni**, cioè

- $\forall T \neq \emptyset, h_T = \max_{v \in E} d_v$
con $E = \{ v \in T \mid T.\text{isExternal}(v) \}$



- Che prestazioni ha, nel caso pessimo, l'algoritmo che calcola l'altezza di un albero usando questa proprietà, anziché usare la definizione ricorsiva vista in precedenza?
- **Si dimostra** (non banalmente) che questo algoritmo per il calcolo dell'altezza di un albero è, nel caso pessimo, $\Theta(n^2)$

Esercizio

Foglie e figli

- Usando il principio d'induzione, dimostrare che, se **ogni nodo interno ha almeno due figli**, un albero non vuoto avente n_I nodi interni ha almeno $n_I + 1$ foglie
- Fare la dimostrazione in due modi diversi, prima usando come variabile induttiva l'altezza dell'albero, poi usando come variabile induttiva il numero di nodi interni dell'albero

□ Quali sono le "frasi" di cui si vuole dimostrare la verità usando il principio di induzione?

□ Usando l'altezza h :

- La proprietà vale quando $h = 0$
- La proprietà vale quando $h = 1$
- La proprietà vale quando $h = 2$
- ...
- Se sono vere tutte queste frasi, allora la proprietà è vera per ogni albero, perché qualsiasi albero ha $h \geq 0$

Se ogni nodo interno ha almeno due figli, un albero non vuoto avente n_I nodi interni ha almeno $n_I + 1$ foglie

□ Usando n_I :

- La proprietà vale quando $n_I = 0$
- La proprietà vale quando $n_I = 1$
- La proprietà vale quando $n_I = 2$
- ...
- Se sono vere tutte queste frasi, allora la proprietà è vera per ogni albero, perché qualsiasi albero ha $n_I \geq 0$

Se ogni nodo interno ha almeno due figli, un albero non vuoto avente n_I nodi interni ha almeno $n_I + 1$ foglie

Foglie e figli

- La dimostrazione può essere condotta per induzione in vari modi, cioè usando diverse variabili induttive
 - Indichiamo con n_E il numero di foglie (o nodi esterni)
- Usando h :
 - Caso base: dimostrare la proprietà per alberi non vuoti di altezza zero
 - Caso induttivo: dimostrare la proprietà per alberi di altezza $h > 0$ supponendo che la proprietà sia vera per alberi di altezza $h' = h - 1$ (induzione semplice) oppure per alberi di altezza $h' < h$ (induzione forte)
- Usando n_I :
 - Caso base: dimostrare la proprietà per alberi non vuoti aventi $n_I = 0$ (cioè privi di nodi interni)
 - Caso induttivo: dimostrare la proprietà per alberi aventi $n_I > 0$, supponendo che la proprietà sia vera per alberi aventi $n_I' = n_I - 1$ nodi interni (induzione semplice) oppure per alberi aventi $n_I' < n_I$ nodi interni (induzione forte)

□ Induzione usando l'altezza:

- Caso base: dimostrare la proprietà per alberi non vuoti di altezza zero
 - Se un albero non vuoto ha altezza zero, ha solo la radice e questa è una foglia: in tale albero $n_I = 0$ e $n_E = 1 = n_I + 1 \geq n_I + 1$, quindi il caso base è vero
- Caso induttivo: dimostrare la proprietà per alberi di altezza $h > 0$ supponendo che la proprietà sia vera per alberi di altezza $h' = h - 1$ (induzione semplice) oppure per alberi di altezza $h' < h$ (induzione forte)
 - **Problema: trovare uno o più alberi per i quali valga l'ipotesi induttiva e che aiutino a dimostrare la proprietà...**
 - **Se l'albero ha altezza $h > 0$ significa che la radice è un nodo interno** (e ha almeno due figli, per ipotesi), **quindi prendiamo in esame i suoi sottoalberi: è facile osservare che ciascun sottoalbero della radice ha altezza minore di h ma non necessariamente $h - 1$, quindi conviene usare l'induzione forte e supporre valida la proprietà per ciascun sottoalbero della radice**
 - **In pratica, si usa (quasi) sempre questa scomposizione...**

□ Induzione usando l'altezza:

- Caso induttivo: dimostrare la proprietà per alberi di altezza $h > 0$ supponendo che la proprietà sia vera per alberi di altezza $h' < h$ **(induzione forte)**
 - ...
 - $\forall c \in C = T.\text{children}(r)$ (cioè per ogni figlio c della radice r), indichiamo con n_{Ic} e n_{Ec} , rispettivamente, il numero di nodi interni e di nodi esterni del sottoalbero avente radice c
 - Per ipotesi induttiva, $|C| \geq 2$ e $\forall c \in C, n_{Ec} \geq n_{Ic} + 1$
 - Inoltre, (in qualunque albero, se C sono i figli della radice)
 - $$n_E = \sum_{c \in C} n_{Ec}$$
 - $$n_I = 1 + \sum_{c \in C} n_{Ic}$$

- $$n_E = \sum_{c \in C} n_{Ec} \geq \sum_{c \in C} (n_{Ic} + 1) = |C| + \sum_{c \in C} n_{Ic} = |C| + n_I - 1 = (|C| - 2) + 1 + n_I \geq 1 + n_I$$
, che è la tesi

□ Induzione usando la variabile n_I :

- Caso base: dimostrare la proprietà per alberi non vuoti con $n_I = 0$
 - Se un albero non vuoto non ha nodi interni, ha solo la radice e questa è una foglia: in tale albero $n_I = 0$ e
$$n_E = 1 = n_I + 1 \geq n_I + 1,$$
 quindi il caso base è vero
- Caso induttivo: dimostrare la proprietà per alberi aventi $n_I > 0$ nodi interni, supponendo che la proprietà sia vera per alberi aventi $n_I' = n_I - 1$ nodi interni (induzione semplice) oppure per alberi aventi $n_I' < n_I$ nodi interni (induzione forte)
 - Problema: trovare uno o più alberi per i quali valga l'ipotesi induttiva e che aiutino a dimostrare la proprietà...
 - **Possiamo usare la stessa scomposizione del caso precedente!**
 - Certamente il numero di nodi interni di ciascun sottoalbero della radice è minore di n_I , perché almeno la radice (che è un nodo interno) non fa parte di nessuno dei sottoalberi.
 - Quindi, **usando l'induzione forte, si può considerare valida la proprietà per ciascuno dei sottoalberi della radice**, dopodiché la dimostrazione si conduce esattamente come nel caso precedente
 - Ci chiediamo: **sarebbe possibile usare l'induzione "semplice"?**

□ Induzione usando la variabile n_I :

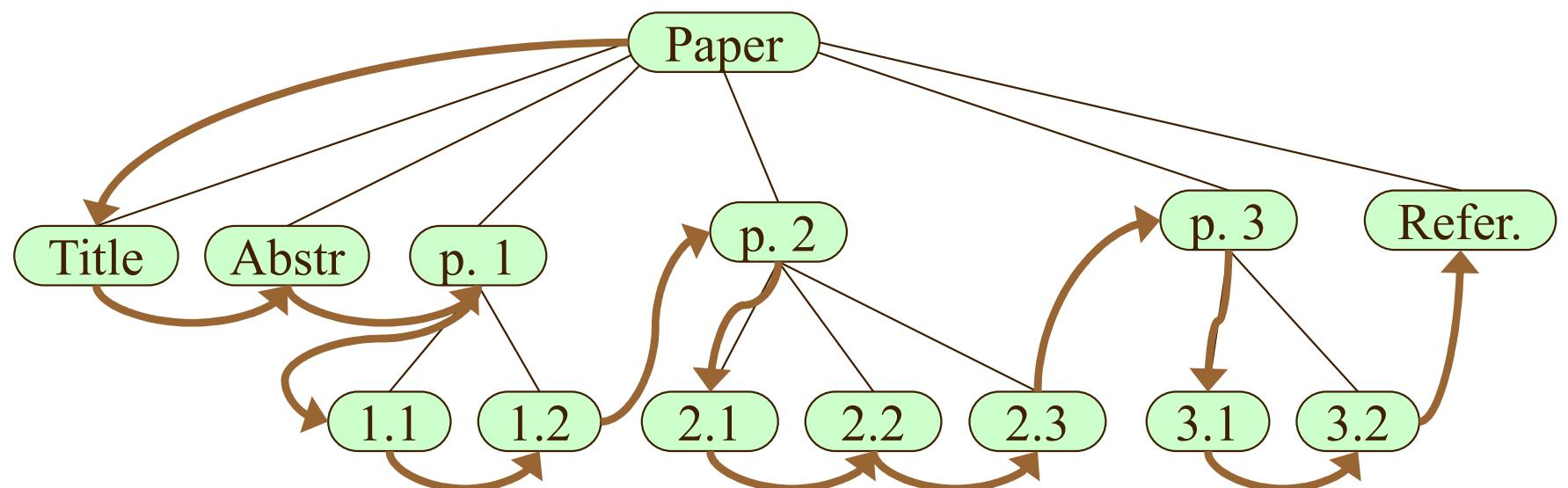
- Caso base: dimostrare la proprietà per alberi non vuoti con $n_I = 0$
 - Se un albero non vuoto non ha nodi interni, ha solo la radice e questa è una foglia: in tale albero $n_I = 0$ e $n_E = 1 = n_I + 1 \geq n_I + 1$, quindi il caso base è vero
- Caso induttivo: dimostrare la proprietà per alberi aventi $n_I > 0$ nodi interni, supponendo che la proprietà sia vera per alberi aventi $n_I' = n_I - 1$ nodi interni (induzione semplice) oppure per alberi aventi $n_I' < n_I$ nodi interni (induzione forte)
 - Problema: trovare uno o più alberi per i quali valga l'ipotesi induttiva e che aiutino a dimostrare la proprietà...
 - Ci chiediamo: **sarebbe possibile usare l'induzione "semplice"?**
 - Dobbiamo cercare **una diversa scomposizione dell'albero T** , perché con la scomposizione nei sottoalberi della radice l'induzione semplice, in generale, non è sufficiente
 - Qui non c'è una strategia generale, bisogna ragionare caso per caso...
 - Se l'albero T ha $n_I > 0$ nodi interni, sia **N uno dei suoi nodi interni avente come figli soltanto foglie**
 - N esiste perché se tutti i nodi interni avessero almeno un nodo interno tra i propri figli, l'albero sarebbe una struttura infinita

□ Induzione usando la variabile n_I :

- Caso induttivo: dimostrare la proprietà per alberi aventi $n_I > 0$ nodi interni, supponendo che la proprietà sia vera per alberi aventi $n_I' = n_I - 1$ nodi interni (induzione semplice) oppure **per alberi aventi $n_I' < n_I$ nodi interni (induzione forte)**
 - Se l'albero T ha $n_I > 0$ nodi interni, sia N uno dei suoi nodi interni avente come figli soltanto foglie
 - Se dall'albero T tolgo le f foglie (con $f \geq 2$) che sono figlie di N , ottengo un albero T' che ha $n_I' = n_I - 1$ nodi interni (perché N non è più un nodo interno) e $n_E' = n_E - f + 1$ foglie (il + 1 deriva dal fatto che N è diventato una foglia)
 - L'albero T' è **non vuoto** (ha almeno il nodo N) e **ogni suo nodo interno ha almeno 2 figli** (perché l'unica modifica rispetto a T è la rimozione dei figli di N , che erano foglie) quindi **posso usare l'induzione semplice** e supporre valida la proprietà per l'albero T'
 - $n_E' \geq n_I' + 1$
 $\Rightarrow n_E - f + 1 \geq n_I - 1 + 1$
 $\Rightarrow n_E \geq n_I + 1 + (f - 2) \geq n_I + 1$, che è la tesi

Lezione 18

Attraversamenti di alberi



Attraversamento di un albero

- Si dice **attraversamento** (*traversal*) di un albero una **strategia di ispezione** che visita **tutti** i nodi dell'albero **una sola volta**
 - È un'azione analoga alla **scansione** di una lista effettuata mediante un iteratore
- Si possono definire diversi attraversamenti, i più importanti dei quali sono
 - Attraversamento in pre-ordine o in ordine anticipato
 - Attraversamento in post-ordine o in ordine posticipato
 - Attraversamento "per livelli" o "in ampiezza" (*breadth-first*)
- I diversi attraversamenti differiscono tra loro per **l'ordine** in cui visitano i nodi dell'albero
- Per particolari tipi di alberi esistono ulteriori specifici attraversamenti
 - Ad esempio, per gli alberi binari definiremo l'**attraversamento in ordine simmetrico** o, semplicemente, "in ordine"

Visita di un nodo

- Si dice **attraversamento** (*traversal*) di un albero una **strategia di ispezione** che **visiti tutti** i nodi dell’albero **una sola volta**
- Cosa si intende per “**visita**” di un nodo?
 - **Una qualsiasi** azione che lo riguardi
 - Ad esempio: la visita visualizza il contenuto del nodo visitato, così l’attraversamento genera una visualizzazione del contenuto dell’intero albero, in una sequenza ben definita
 - Altro esempio: l’azione può essere semplicemente l’incremento di un contatore gestito durante l’intero attraversamento, che diventa, così, un modo per contare i nodi dell’albero (cioè di calcolarne la dimensione); oppure può contare soltanto le foglie o soltanto i nodi interni (pur visitando tutti i nodi)
 - L’attraversamento è soltanto un **algoritmo di navigazione completa** dell’albero, non compie elaborazioni
 - **L’elaborazione dipende dalla singola azione di visita**

Attraversamento in pre-ordine (*preorder traversal*)

Dimostreremo che è un attraversamento!

□ Algoritmo (ricorsivo)

- Per **attraversare in pre-ordine** $T \neq \emptyset$ avente radice v
 - **Si visita** v
 - **Per ogni** figlio w di v // in ordine, se l'albero è ordinato
 - Si **attraversa in pre-ordine** il sottoalbero di T avente radice w

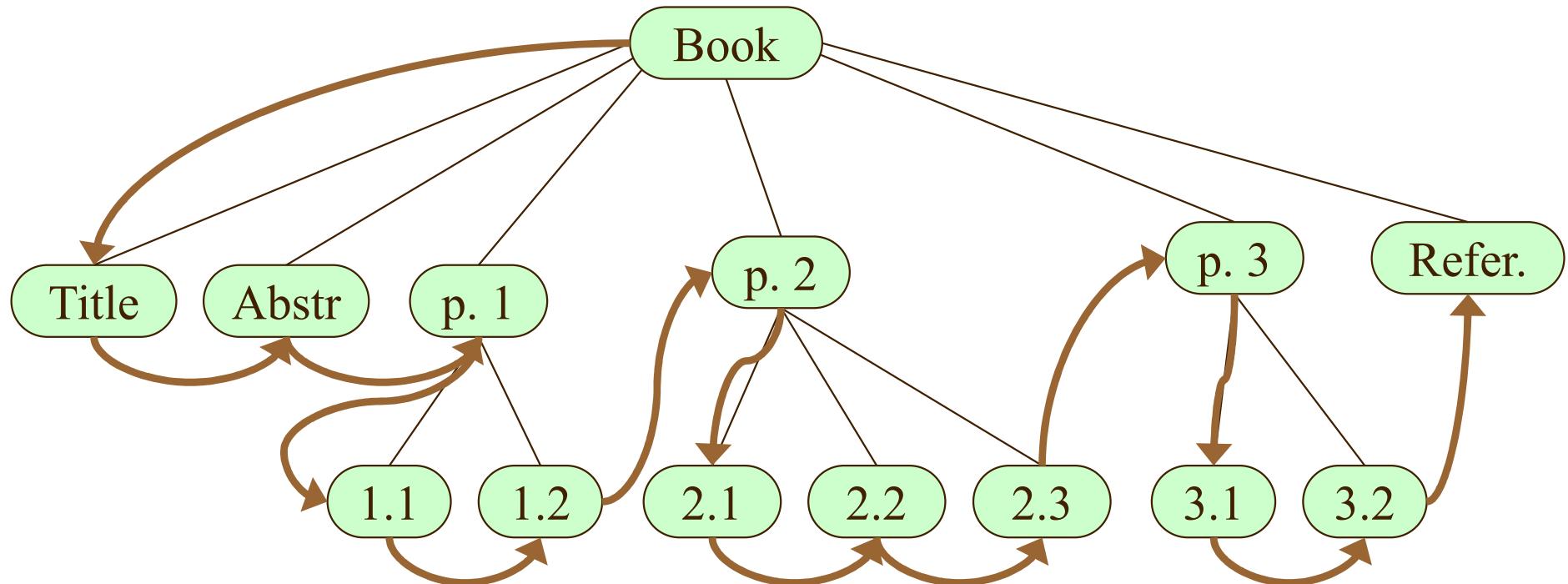
□ L'attraversamento in pre-ordine produce una sequenza posizionale lineare (una lista!) di visite nella quale

- **Ciascun nodo dell'albero viene visitato prima di tutti i suoi figli**

A volte si dice che un attraversamento "linearizza" l'albero, perché genera **implicitamente** una lista contenente le stesse posizioni

Attraversamento in pre-ordine

- Esempio: avendo un albero **ordinato** che rappresenta la struttura di un libro suddiviso in capitoli/paragrafi, usando **una visita che visualizzi il contenuto del nodo visitato**, un suo attraversamento in pre-ordine genera l'indice del libro



L'attraversamento in pre-ordine avviene secondo la sequenza indicata dalle frecce, quindi, **in generale, non segue rami dell'albero**

Preorder traversal: pseudocodice

- Descriviamo l'attraversamento in pre-ordine con pseudocodice

preOrderTraversal(T) // metodo non ricorsivo, innesca la ricorsione

if !T.isEmpty() // se è vuoto non c'è nulla da attraversare...

 preOrderTraversal(T, T.root())

preOrderTraversal(T, v) // metodo ricorsivo, attraversa S_v

visit(v) // azione di visita, metodo "esterno" all'algoritmo

 // molto probabilmente visit deve ricevere anche T

C = T.children(v) // lista ordinata, se l'albero è ordinato

for each w ∈ C

preOrderTraversal(T, w)

 // è una ricorsione **multipla**, con molteplicità uguale al grado dell'albero

- Al metodo ricorsivo serve T per invocare il metodo **chidren**

Preorder traversal: pseudocodice

- Senza il metodo non ricorsivo, l'utente dell'attraversamento deve invocare direttamente il metodo ricorsivo, passando come argomenti l'albero e la sua radice: assolutamente innaturale (perché chi riceve l'albero potrebbe ispezionarne la radice autonomamente)
- Per evitare questa forzatura, normalmente si mette a disposizione dell'utente un metodo che riceva i parametri più "naturali": questo tipo di metodi, che in pratica non fanno elaborazione ma predispongono i parametri per un altro metodo, si chiamano *stub* ("adattatore", perché "adattano" i parametri)
- È una tecnica che si usa spesso con i metodi ricorsivi
- Altro vantaggio: così il metodo ricorsivo può diventare "privato" (in Java) e non è più necessario che verifichi la validità dei propri parametri (come vedremo negli esempi di codice), risparmiando tempo significativo, anche se non asintotico (perché, anche se la verifica è $\Theta(1)$, il metodo viene invocato molte volte)

Utilizzo degli attraversamenti

- Gli algoritmi di attraversamento, **completati da un'opportuna azione di visita**, seguono generalmente uno di questi schemi
 - L'azione di visita modifica in qualche modo il nodo oggetto della visita stessa, senza generare un risultato globale per effetto dell'attraversamento dell'albero (cioè la visita è un'azione "locale")
 - Es. in ogni nodo viene memorizzata la propria profondità
 - L'azione di visita modifica una variabile globale (in Java, una variabile statica), producendo un risultato globale per effetto dell'attraversamento dell'albero **[nel nostro corso, in generale questo è proibito]**
 - Es. l'attraversamento conta le foglie dell'albero
 - L'attraversamento restituisce una variabile che rappresenta un risultato globale, elaborato componendo il risultato prodotto dall'azione di visita e dall'attraversamento dei sottoalberi del nodo visitato (**preferibile rispetto all'uso di una variabile globale**, visto al punto precedente)
 - Es. l'attraversamento conta le foglie dell'albero

Scrivere la profondità in ciascun nodo

- L'azione di visita modifica lo stato del nodo oggetto della visita stessa, senza generare un risultato globale per effetto dell'attraversamento dell'albero

```
public static void setDepth(Tree<Integer> t)
{   if (t != null && !t.isEmpty()) setDepth(t, t.root());
} // else eventuale eccezione, ma non è necessario
private static void setDepth(Tree<Integer> t,
                           Position<Integer> v)
{   int depthV = 0; // da calcolare
    Position<Integer> current = v;
    while (!t.isRoot(current))
    {   depthV++;
        current = t.parent(current);
    } // se v è la radice, depthV rimane zero: ok
    t.replace(v, depthV);
    // la visita è tutta la parte blu
    for (Position<Integer> x : t.children(v)) // for each
        setDepth(t, x);
} // privato: posso evitare il controllo dei parametri
```

- L'azione di visita calcola anche la profondità del nodo... lentamente!
Richiede un ciclo che sappiamo avere prestazioni $\Theta(d_v)$.

Scrivere la profondità in ciascun nodo

- Osserviamo che quando invochiamo il metodo ricorsivo con una posizione, di cui dovrà essere calcolata la profondità, in realtà CONOSCIAMO tale profondità! Non c'è bisogno di calcolarla, la forniamo come parametro.

```
public static void setDepth(Tree<Integer> t)
{   if (t != null && !t.isEmpty())
    setDepth(t, t.root(), 0);
} // assegna profondità uguale a zero alla radice

private static void setDepth(Tree<Integer> t,
                           Position<Integer> v,
                           int depthV) // nuovo parametro
{   t.replace(v, depthV); // questa è la visita
   for (Position<Integer> x : t.children(v))
       setDepth(t, x, depthV+1);
}
```

- Abbiamo aggiunto un parametro al metodo ricorsivo per rendere più efficiente il calcolo della profondità
- L'azione di visita è ora $\Theta(1)$

Scrivere la profondità in ciascun nodo

- ❑ Ancora meglio:

Uso le informazioni generate e memorizzate in precedenza!

```
public static void setDepth(Tree<Integer> t)
{   if (t != null && !t.isEmpty()) setDepth(t, t.root());
}
private static void setDepth(Tree<Integer> t,
                           Position<Integer> v)
{   if (t.isRoot(v)) t.replace(v, 0);
    else t.replace(v, 1 + t.parent(v).element());
    for (Position<Integer> x : t.children(v))
        setDepth(t, x);
}
```

- ❑ Cioè ricordo che, per ogni nodo diverso dalla radice, la profondità è uguale a uno più la profondità del genitore, e la vado a leggere, perché so che è già stata scritta (per le proprietà del pre-ordine...)
- ❑ Questa soluzione è asintoticamente veloce quanto la precedente, perché l'azione di visita rimane $\Theta(1)$, pur senza usare il parametro aggiuntivo **depthV**

Contare le foglie

- L'azione di visita modifica una variabile globale (in Java, una variabile statica), producendo un risultato globale per effetto dell'attraversamento dell'albero (**pessimo stile di programmazione...**)

```
public static int count;

public static <T> void worstCountLeaves(Tree<T> t)
{   count = 0;
    if (t != null && !t.isEmpty())
        worstCountLeaves(t, t.root());
} // il risultato sarà accessibile tramite count
private static <T> void worstCountLeaves(Tree<T> t,
                                         Position<T> v)
{   if (t.isExternal(v)) count++; // azione di visita
    for (Position<T> x : t.children(v))
        worstCountLeaves(t, x);
}
```

Contare le foglie

- Molto meglio: l'attraversamento **restituisce** un valore che rappresenta un risultato globale, elaborato componendo il risultato prodotto dall'azione di visita e dall'attraversamento dei sottoalberi del nodo visitato

```
public static <T> int countLeaves(Tree<T> t)
{ if (t == null || t.isEmpty()) return 0;
  return countLeaves(t, t.root());
}

private static <T> int countLeaves(Tree<T> t,
                                    Position<T> v)
{ if (t.isExternal(v)) // azione di visita di una foglia
  return 1;
// azione di visita di un nodo interno
int count = 0;
for (Position<T> x : T.children(v))
  count += countLeaves(t, x);
return count; // foglie nel sottoalbero con radice v
}
```

Contare i nodi interni

- Analogamente si possono contare i nodi interni di un albero

```
public static <T> int countInternals(Tree<T> t)
{   if (t == null || t.isEmpty()) return 0;
    return countInternals(t, t.root());
}
private static <T> int countInternals(Tree<T> t,
                                      Position<T> v)
{   if (t.isExternal(v)) // azione di visita di una foglia
    return 0;
    // azione di visita di un nodo interno
    int count = 1; // conto la radice del sottoalbero
    for (Position<T> x : T.children(v))
        count += countInternals(t, x);
    return count;
}
```

Contare i nodi

- ... o tutti i nodi di un albero (anche se è inutile, perché solitamente un albero ha una variabile di esemplare che ne memorizza il numero di nodi)

```
public static <T> int countNodes(Tree<T> t)
{ if (t == null || t.isEmpty()) return 0;
  return countNodes(t, t.root());
}
private static <T> int countNodes(Tree<T> t,
                                  Position<T> v)
{ if (t.isExternal(v))
  return 1;
  int count = 1;
  for (Position<T> x : T.children(v) )
    count += countNodes(t, x);
  return count;
}
```

Un'applicazione

- Come si può realizzare il metodo **positions ()** dell'interfaccia **Tree**?
 - Costruisce e restituisce una lista contenente le posizioni presenti nell'albero
 - **Ogni posizione deve essere presente nella lista una e una sola volta**
- Il metodo si può realizzare usando un **attraversamento** (qualsiasi, ad esempio in pre-ordine)
 - Come “azione di visita” del nodo si usa **“aggiungi la posizione alla lista” (inizialmente vuota)**
- Il metodo **iterator ()** è analogo: crea una lista di dati anziché di posizioni

- Come si può realizzare il metodo **positions()** dell’interfaccia **Tree**? Ad esempio, con un *preorder traversal*

```
import java.util.ArrayList;
public class XyzTree<T> implements Tree<T>
{
    ...
    public Iterable<Position<T>> positions()
    { // oppure un altro tipo di lista...
        ArrayList<Position<T>> list
            = new ArrayList<Position<T>>(); // vuota
        if (isEmpty()) return list; // lista vuota
        preorderAppendPosition(list, root());
        list.trimToSize(); // evito spreco di spazio
        return list;
    } // potrebbe anche essere un metodo "esterno"
    private void preorderAppendPosition(
        ArrayList<Position<T>> list, Position<T> v)
    { list.add(v); // azione di visita  $\Theta(1)$  in media
        for (Position<T> c : children(v)) // OK anche
            // senza figli
            preorderAppendPosition(list, c);
    } // metodo privato, non controllo i parametri
}
```

Preorder traversal: correttezza

- Cosa significa "dimostrare che l'algoritmo di attraversamento in pre-ordine è corretto"?
 - Significa dimostrare che **visita una e una sola volta ciascun nodo dell'albero**
- Più in dettaglio, dimostreremo che
 - **Ogni nodo dell'albero viene visitato** (almeno una volta)
 - **Nessun nodo viene visitato più di una volta**

Preorder traversal: ricordiamo lo pseudocodice

```
preOrderTraversal(T)
  if !T.isEmpty()
    preOrderTraversal(T, T.root())
```

preOrderTraversal(T, v)

visit(v)

$C = T.\text{children}(v)$

for each $w \in C$

preOrderTraversal(T, w)

Osservo che **l'azione di visita per v** avviene **incondizionatamente (e una volta sola)** in seguito a ogni **invocazione di preOrderTraversal** avente come **parametro il nodo v** . Quindi, "contare" le azioni di visita per v **equivale** a contare le invocazioni di preOrderTraversal per v

Lezione 19

Attenzione: l'ipotesi "per assurdo" deve (sempre) essere l'esatta negazione della tesi che si vuole dimostrare!

Preorder traversal: correttezza

- Dimostrazione: **ogni nodo dell'albero viene visitato**
- Poniamo $H \equiv \{y \in T \mid y \text{ non viene visitato da } \text{preOrderTraversal}(T, T.\text{root}())\}$
- **Tesi:** $H = \emptyset$
- **Per assurdo:** $H \neq \emptyset$. Allora $\exists v \in H \mid \text{depth}(v) \leq \text{depth}(z) \forall z \in H$
- *Primo caso:* v è la radice di T .
 - Ma il primo enunciato di $\text{preOrderTraversal}(T, T.\text{root}())$ visita proprio la radice di $T \Rightarrow v \notin H$. **Assurdo.**
- *Secondo caso:* v non è la radice di T . Allora, $\exists p = T.\text{parent}(v)$.
 - $v \in H \Rightarrow p \in H$ (altrimenti, se p venisse visitato, anche v verrebbe visitato, per effetto dell'esecuzione del "for each" di p)
 - Ma $p = T.\text{parent}(v) \Rightarrow \text{depth}(p) = \text{depth}(v) - 1 < \text{depth}(v)$ con $p \in H$.
Quindi, non è vero che $\text{depth}(v) \leq \text{depth}(z) \forall z \in H$. **Assurdo.**

Preorder traversal: correttezza

- Dimostrazione: **ogni** nodo dell’albero viene visitato
- Attenzione a un **errore frequente**: si pone un’ipotesi “per assurdo” che **non** è l’esatta negazione della tesi
- Per assurdo, suppongo che $\exists ! v \in T \mid v$ non viene visitato
 - ...
 - Dimostro l’assurdo, sfruttando (implicitamente o esplicitamente) il fatto che v sia **unico**
 - Da questo deduco la tesi.
 - Ma è sbagliato! Non ho escluso che $\exists v \in T \text{ e } w \in T \mid v \neq w$ non vengono visitati!
 - Cioè potrebbe esistere una coppia di nodi che non vengono visitati dall’algoritmo, pur avendo escluso che l’algoritmo possa visitare “tutti i nodi tranne uno”.

Preorder traversal: correttezza

- Dimostrazione: nessun nodo viene visitato più di una volta
- Poniamo $K \equiv \{y \in T \mid y \text{ viene visitato più di una volta da } \text{preOrderTraversal}(T, T.\text{root()})\}$
- Tesi: $K = \emptyset$. Per assurdo: $K \neq \emptyset$. Allora $\exists v \in K \mid \text{depth}(v) \leq \text{depth}(z) \forall z \in K$
- Primo caso: v è la radice di T .
 - La radice di T viene visitata una volta durante l'esecuzione di $\text{preOrderTraversal}(T, T.\text{root()})$ e questo è l'unico caso in cui un nodo viene visitato senza che l'invocazione di preOrderTraversal sia stata eseguita durante la scansione di una lista di figli. Per essere visitata una seconda volta, la radice deve comparire nella lista dei figli di un nodo. Ma la radice non ha genitore. **Assurdo.**
- Secondo caso: v non è la radice di T . Allora, $\exists p = T.\text{parent}(v)$.
 - $v \in K \Rightarrow p \in K$: infatti, ogni volta che v viene visitato, significa che è stato invocato $\text{preOrderTraversal}(T, v)$ e questo avviene una sola volta durante la scansione di $T.\text{children}(p)$ all'interno dell'esecuzione di $\text{preOrderTraversal}(T, p)$, che, quindi, è stato invocato due volte e ognuna di queste invocazioni visita p
 - Ma $p = T.\text{parent}(v) \Rightarrow \text{depth}(p) = \text{depth}(v) - 1 < \text{depth}(v)$ con $p \in K$
Quindi, non è vero che $\text{depth}(v) \leq \text{depth}(z) \forall z \in K$. **Assurdo.**

Attraversamento in pre-ordine

- Dal punto di vista **temporale** è un algoritmo **ottimo**: se la visita è un’azione $\Theta(1)$, l’attraversamento in pre-ordine visita tutti i nodi di un albero di dimensione n in un tempo $\Theta(n)$
 - Tale operazione è necessariamente $\Omega(n)$, quindi è ottimo!
- **Dimostrazione:** L’analisi è simile a quella fatta per l’algoritmo che calcola l’altezza di un albero
 - Per ogni nodo v , l’algoritmo richiede un tempo d’esecuzione suddiviso in due porzioni
 - Una porzione di tempo per la visita del nodo v , $\Theta(1)$
 - Una porzione di tempo proporzionale al numero g_v di figli di v , cioè un tempo $\Theta(g_v)$ [per la scansione della lista di figli]
 - Il tempo totale è, quindi:

Quindi il metodo **positions()** realizzato con questo attraversamento è $\Theta(n)$

$$\Theta\left(\sum_{v \in T} (1 + g_v)\right) = \Theta(n)$$

perché

$$\sum_{v \in T} g_v = n - 1$$

Attraversamento in pre-ordine

- Molto spesso l’azione di visita è un’operazione **locale**, che coinvolge soltanto il nodo visitato (e/o il dato in esso contenuto) e/o il suo genitore: in tutti questi casi, l’operazione di visita è $\Theta(1)$
- Altre volte la visita, pur essendo locale, richiede un’ispezione dei figli del nodo visitato: in tal caso l’azione di visita del nodo v è $O(g_v)$, ma, in ogni caso, l’attraversamento risulta essere $\Theta(n)$, perché

$$\Theta\left(\sum_{v \in T} (1 + g_v + g_v)\right) = \Theta(n)$$

Prestazioni di un attraversamento

- Quindi, se ogni azione di visita è $\Theta(1)$ o, più in generale, $O(g_v)$, l'attraversamento in pre-ordine è $\Theta(n)$
- Se, invece, ogni azione di visita fosse, ad esempio, $\Theta(\log n)$, il tempo totale richiesto per tutte le n azioni di visita sarebbe $\Theta(n \log n)$, da "sommare" al tempo richiesto per l'attraversamento effettuato con visite tempo-costanti, ottenendo un attraversamento complessivamente $\Theta(n + n \log n) \equiv \Theta(n \log n)$
- In generale, se ogni visita è $\Theta(f(n, h))$, l'attraversamento è $\Theta(n f(n, h))$
 - Es. Visita $\Theta(n) \Rightarrow$ attraversamento $\Theta(n^2)$
 - Es. Visita $\Theta(h) \Rightarrow$ attraversamento $\Theta(nh)$, che è anche $O(n^2)$ perché, ovviamente, h è $O(n)$

Prestazioni di un attraversamento: memoria

- La profondità di ricorsione dell'algoritmo di attraversamento in pre-ordine è, evidentemente, uguale all'altezza h dell'albero, quindi **necessita di uno spazio aggiuntivo di dimensioni $\Theta(h)$**
- Bisogna poi sommare lo spazio richiesto per le liste restituite dalle invocazioni di **children** nei metodi ricorsivi attivi sul runtime stack e per i relativi iteratori
 - Il metodo **children** potrebbe restituire semplicemente un riferimento alla lista presente nel nodo ma a volte fa una copia di tale lista
 - Quindi serve uno spazio $O(g_v)$ per ogni nodo v presente nel percorso che va dalla radice al punto in cui si trova l'algoritmo... nel caso pessimo la somma di questi spazi, occupati contemporaneamente, è $\Theta(n)$, in generale è $O(n)$ [pensare a un esempio in cui è $\Theta(n)$]
- Infine, a questo spazio va sommato quello eventualmente necessario per **una** azione di visita, che però spesso è $\Theta(1)$ (anche se può essere, ad esempio, $O(h)$ o $O(n)$)
- In conclusione, lo spazio aggiuntivo richiesto dall'attraversamento è **$\Omega(h)$ e $O(n)$ più quello richiesto dalle azioni di visita**

Postorder traversal: pseudocodice

□ Descriviamo con pseudocodice
postOrderTraversal(T)

if !T.isEmpty()

 postOrderTraversal(T, T.root())

postOrderTraversal(T, v)

 C = T.children(v)

for each w ∈ C

 postOrderTraversal(T, w)

visit(T, v)

Ciascun nodo
dell'albero viene
visitato **dopo**
tutti i suoi figli

Con dimostrazione analoga a quella vista per
l'attraversamento in pre-ordine, è corretto ed è $\Theta(n)$

Attraversamento in post-ordine

- L'attraversamento in post-ordine è particolarmente utile quando, visitando un nodo, si vuole effettuare un'elaborazione che dipende da quella analoga **già svolta sui figli** del nodo
- Esempio: progettare un algoritmo che scriva in ciascun nodo di un albero la propria altezza
 - L'algoritmo sarà simile a quello, già visto, che scrive in ciascun nodo la propria profondità, ma **l'altezza di un nodo dipende dalle altezze dei suoi figli**, mentre la sua profondità dipende da quella del suo genitore
 - Quindi, serve un attraversamento in post-ordine, non in pre-ordine!

Attraversamento in post-ordine

```
public static void setHeight(Tree<Integer> t)
{ if (t != null && !t.isEmpty())
    setHeight(t, t.root());
}
private static int setHeight(Tree<Integer> t,
                           Position<Integer> v)
{ int height = 0; // così è OK se v è una foglia
  for (Position<Integer> x : t.children(v) )
  { int h = setHeight(t, x);
    // h è l'altezza del sotto-albero avente radice x
    if (h + 1 > height)
      height = h + 1;
  }
  t.replace(v, height); // questa è la visita
  return height;
}
```

Attraversamento in post-ordine

```
public static void setHeight(Tree<Integer> t)
{ if (t != null && !t.isEmpty())
    setHeight(t, t.root());
}
// invece di restituire un valore lo può ispezionare...
private static void setHeight(Tree<Integer> t,
                           Position<Integer> v)
{ int height = 0; // così è OK se v è una foglia
  for (Position<Integer> x : t.children(v))
  { setHeight(t, x);
    int h = x.element(); // ispeziono perché adesso c'è
    // h è l'altezza del sotto-albero avente radice x
    if (h + 1 > height)
      height = h + 1;
  }
  t.replace(v, height); // questa è la visita
}
```

Attraversamenti

- Nell'esempio appena visto era possibile evitare che il metodo ricorsivo dovesse restituire un valore, perché il valore restituito viene anche scritto nel nodo e quindi è ispezionabile successivamente
- **In generale, però, per fornire un valore "verso l'alto" durante l'attraversamento si usa il valore restituito dal metodo ricorsivo**
 - Se servono più valori, si può usare un oggetto contenitore (un array, una lista o un oggetto ad hoc)
- **Analogamente, per fornire valori "verso il basso" durante l'attraversamento si usano i parametri del metodo ricorsivo**

Proprietà

- Ricordiamo che nell'attraversamento in **pre-ordine**
 - Ciascun nodo dell'albero viene visitato **prima** di tutti i suoi **figli**
 - Ma anche i suoi figli vengono visitati **prima** di tutti i loro **figli**...
- Applicando questa proprietà ricorsivamente, otteniamo che
 - Ciascun nodo dell'albero viene visitato dall'attraversamento in **pre-ordine** **prima** di tutti i suoi **discendenti propri** e, quindi, **dopo** tutti i suoi **antenati propri**, perché di questi ultimi è un discendente
 - Quindi, la **radice** dell'albero è il **primo** nodo visitato, perché tutti gli altri nodi dell'albero sono suoi discendenti propri
- Attenzione, però: in generale, l'albero **NON** si partiziona in antenati e discendenti di un nodo (ci sono zii, fratelli, cugini...)

Proprietà

- Ricordiamo che nell'attraversamento in **post-ordine**
 - Ciascun nodo dell'albero viene visitato **dopo** tutti i suoi **figli**
 - Ma anche i suoi figli vengono visitati **dopo** tutti i loro **figli**...
- Applicando questa proprietà ricorsivamente, otteniamo che
 - Ciascun nodo dell'albero viene visitato dall'attraversamento in **post-ordine** **dopo** tutti i suoi **descendenti propri** e, quindi, **prima** di tutti i suoi **antenati propri**, perché di questi ultimi è un discendente
 - Quindi, la **radice** dell'albero è **l'ultimo** nodo visitato, perché tutti gli altri nodi dell'albero sono suoi discendenti propri

Attraversamenti non ricorsivi

- Tutti gli algoritmi di attraversamento possono (**ovviamente!**) essere realizzati anche con schemi iterativi (cioè non ricorsivi)
- Solitamente gli attraversamenti iterativi sono progetti più complessi
 - Li vedremo soltanto nel caso di alberi binari

Postorder traversal

postOrderTraversal(T)

 if !T.isEmpty()

 postOrderTraversal(T, T.root())

postOrderTraversal(T, v)

 C = T.children(v)

 for each w ∈ C

postOrderTraversal(T, w)

visit(T, v)

Lo stesso dubbio può sorgere
nell'attraversamento in pre-ordine

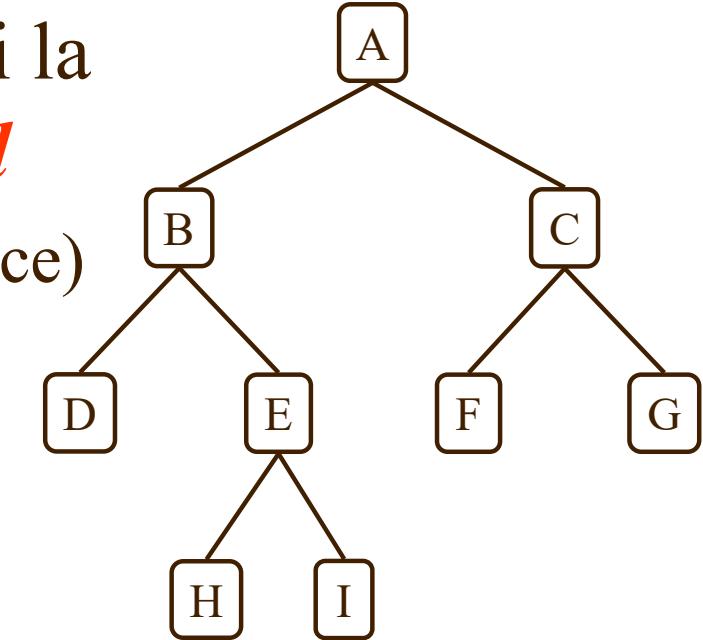
Attenzione a non confondere l'azione di VISITA con il fatto che l'algoritmo "transita" attraverso un nodo per scendere verso i suoi figli... oppure poi per risalire verso il genitore

Un nodo viene visitato SOLTANTO nel momento in cui viene invocato il metodo visit() su di esso.

Attraversamento “per livelli” o “in ampiezza” (breadth-first traversal)

- L’insieme dei nodi dell’albero T aventi la stessa profondità d si chiama **livello d**

- Il livello 0 ha al massimo un nodo (la radice)
- Nota: **nodi aventi la stessa profondità possono avere altezze diverse tra loro**

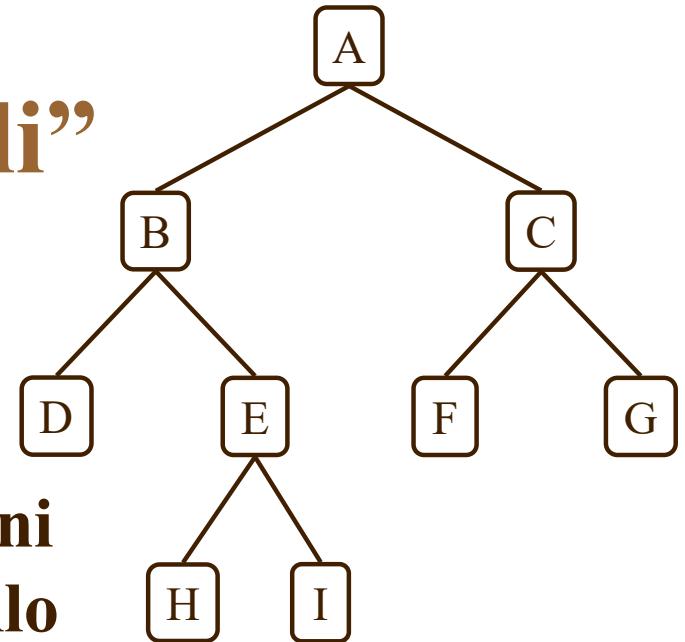


- Attraversamento “per livelli”

- Per ogni valore di profondità, d , visita tutti i nodi di profondità d prima di qualsiasi nodo di profondità maggiore di d
 - All’interno di un livello può procedere in modo arbitrario, rispettando l’ordinamento tra fratelli solo se l’albero è ordinato
- Numerando progressivamente i nodi secondo l’ordine di visita, si ottiene una **numerazione dell’albero “per livelli”**
 - I nodi più profondi sono associati a indici più elevati

Attraversamento “per livelli”

- **Idea:** mentre visitiamo i nodi di un livello, **prepariamo** le informazioni necessarie all’attraversamento del livello immediatamente successivo
- Immaginiamo che “qualcuno” voglia predisporre le informazioni per attraversare il terzo livello (cioè il livello 2). Quali sono le informazioni che ci servono?
 - Ci basta avere una lista dei nodi del livello (in questo caso, D, E, F e G, in qualsiasi ordine): effettuiamo una scansione della lista e visitiamo i nodi
 - Se l’albero è ordinato, la lista deve essere ordinata di conseguenza
 - **Non serve avere accesso casuale alla lista**

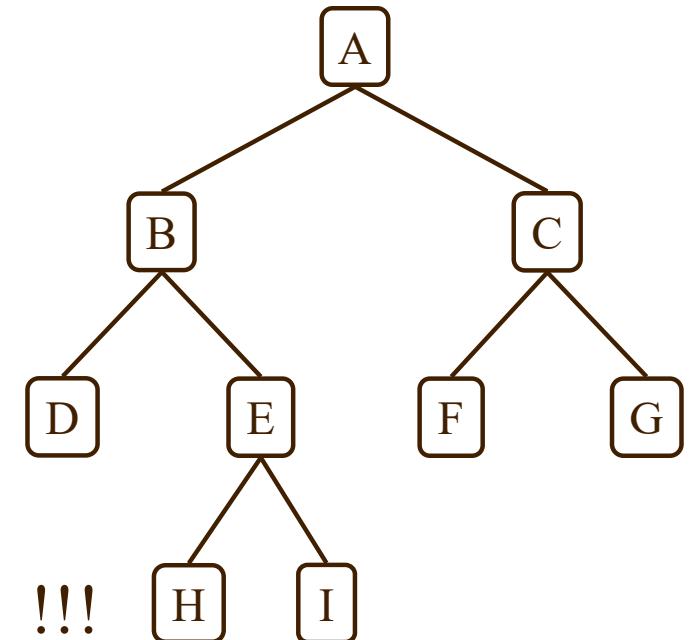


Attraversamento “per livelli”

- Immaginiamo che “qualcuno” abbia predisposto le informazioni per attraversare il livello i

- Come facciamo a costruire la lista dei nodi da visitare al livello $i + 1$?

La costruiamo mentre visitiamo il livello i !!!



- Visitando i nodi del livello i , chiediamo a ciascun nodo quali siano i suoi figli e li inseriamo ordinatamente **in una coda**: questi saranno (tutti e soli) i nodi di livello $i + 1$
 - Un nodo di profondità $i + 1$ ha il genitore avente profondità i
 - Da questa coda, poi, estraiamo i nodi ordinatamente (sono quelli di livello $i + 1$) e li visitiamo
 - Mentre facciamo questo, inseriamo **nella stessa coda** i figli di tali nodi, e così via... Tali figli, che appartengono al livello $i + 2$, andranno in coda DOPO tutti i nodi di livello $i + 1$, che sono già in coda
 - Si parte inserendo la radice in una coda vuota

Attraversamento “per livelli”

levelOrderTraversal(Tree T):

```
if  $T.isEmpty()$ 
    return
q = new Queue()
q.enqueue( $T.root()$ )
while !q.isEmpty()
    v = q.dequeue()
    visit( $T, v$ )
    for each  $f \in T.children(v)$ 
        q.enqueue( $f$ )
```

Correttezza

(dimostrazione non formale)

I nodi **inseriti** in coda sono la radice e i figli di ogni nodo estratto dalla coda (che, quindi, vi era stato inserito). Dato che tutti i nodi di un albero sono figli o sono la radice, tutti i nodi vengono inseriti in coda (una sola volta).

La coda viene svuotata, quindi tutti i nodi vengono **estratti** e, poi, **visitati**. Essendo stati inseriti una sola volta, vengono estratti e visitati una sola volta.

Il fatto che questo avvenga “per livelli” è già stato “dimostrato” in precedenza.

Attraversamento “per livelli”

levelOrderTraversal(Tree T):

```
if  $T.isEmpty()$ 
    return
    q = new Queue()
    q.enqueue( $T.root()$ )
    while !q.isEmpty()
        v = q.dequeue()
        visit( $T$ , v)
        for each  $f \in T.children(v)$ 
            q.enqueue( $f$ )
```

Al massimo la coda contiene i nodi di due livelli consecutivi: **lo spazio massimo occupato**, quindi, è non superiore al doppio della dimensione del livello più popolato (che non è necessariamente l’ultimo...).

Ricordiamo, poi, che **l’algoritmo non è ricorsivo**, quindi non occupa altro spazio

Prestazioni temporali

Ciascun nodo viene inserito in coda una e una sola volta, perché non può essere figlio di più nodi: $\Theta(n)$ per tutti gli inserimenti in coda.

Ciascun nodo viene, quindi, estratto dalla coda (che viene svuotata) una e una sola volta: $\Theta(n)$ per tutte le estrazioni dalla coda.

Per ciascun nodo viene creata e scandita la lista dei figli, in un tempo totale $\Theta(n)$ [solita sommatoria dei gradi...]

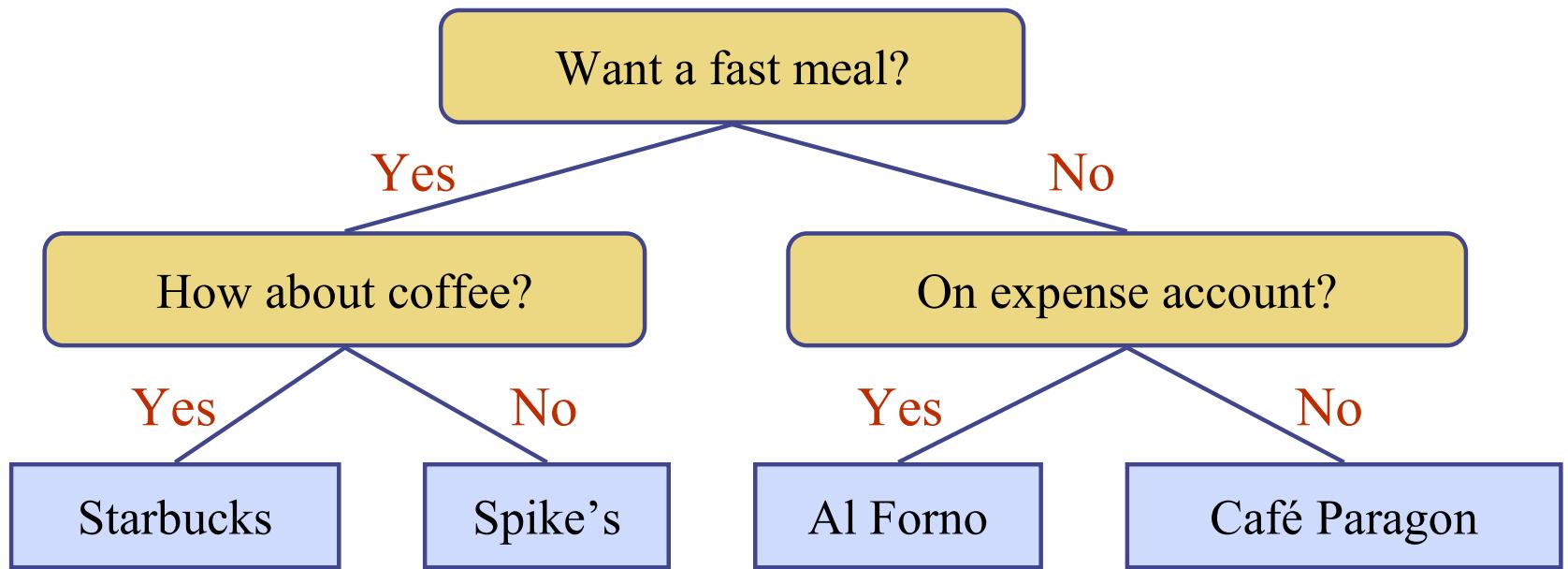
Se l’azione di visita è $\Theta(1)$, l’attraversamento è $\Theta(n)$.

Attraversamento “per livelli”

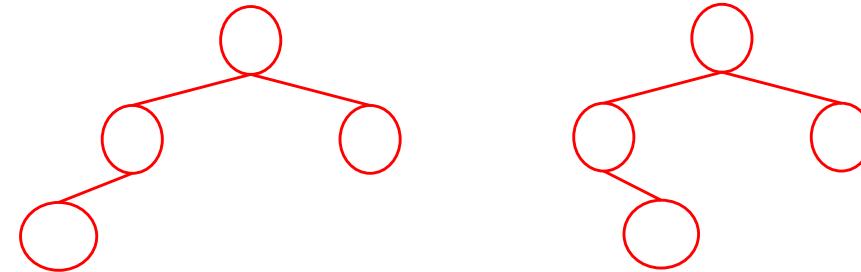
- Invece di una coda si potrebbe ovviamente usare una lista, in cui i nuovi elementi vengono aggiunti in fondo e gli elementi da visitare, anziché "estratti" dall'inizio vengono semplicemente ottenuti da una scansione progressiva della lista dall'inizio alla fine (e durante tale scansione vengono aggiunti in fondo alla lista i figli degli elementi visitati)
- **In questo modo, però, lo spazio aggiuntivo richiesto è $\Theta(n)$, mentre con la coda è soltanto $O(n)$ e tipicamente molto inferiore**
- Si osservi che "sovrapporre" una tale lista concatenata ai nodi dell'albero equivale all'aggiunta di un collegamento che colleghi ciascun nodo al nodo che lo segue nell'attraversamento per livelli, però questa struttura aggiuntiva è più flessibile, in quanto non richiede di modificare i nodi dell'albero

Lezione 20

Alberi binari



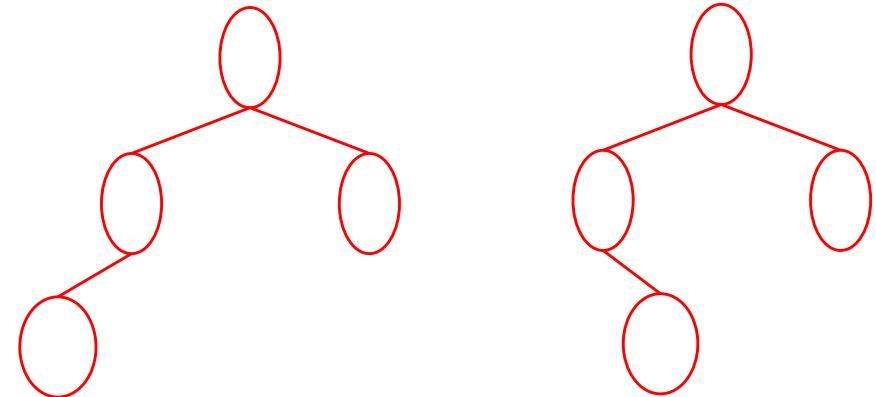
Alberi binari



- Un albero binario (*BT, binary tree*) è un ADT che risponde alla definizione di “albero ordinato di grado due”,
con le seguenti proprietà aggiuntive
 - Un figlio può chiamarsi **sinistro (left)** o **destro (right)**
 - In un nodo avente due figli, **il figlio sinistro precede il figlio destro** nell’ordinamento posizionale dei figli
- Dalla definizione: un nodo può avere anche il solo figlio sinistro, oppure **il solo figlio destro**
 - Quindi, è **sbagliato** dire che, in un nodo, il **primo** figlio è il figlio sinistro: non si applica il concetto di primo e secondo figlio, si parla di figlio sinistro e figlio destro
- **Non** è un semplice albero ordinato di grado 2

Alberi binari

- Un **albero binario non** è un semplice **albero ordinato di grado 2**
- Questi due alberi ordinati di grado 2 sono identici, mentre gli stessi due alberi, se considerati alberi binari, sono diversi!
 - In quello di sinistra il figlio sinistro della radice ha il solo figlio sinistro, mentre in quello di destra ha il solo figlio destro
- Problema di nomenclatura
 - In questo corso usiamo il termine “albero binario” per i soli “alberi ordinati di grado 2 che distinguono i figli come sinistro e destro”, chiamando gli altri “alberi ordinati di grado 2”
 - In alcuni testi (ma **non in questo corso**), si definiscono gli alberi binari in modo che ogni nodo interno abbia esattamente due figli, uno dei quali eventualmente “vuoto”, cioè **privo di dati**, mentre i **“nostri” alberi hanno tanti nodi quanti dati** [tranne quando diversamente specificato]



Alberi binari propri e impropri

- Se **non** esiste un nodo avente **un solo** figlio, l’albero binario si dice **proprio** (*proper*) o pieno (*full*)
 - Tutti i suoi nodi interni hanno esattamente due figli e si dicono, a loro volta, *propri*
 - **Non** usiamo l’aggettivo “completo”, che è riferito a un’altra proprietà (che vedremo)
- Un albero binario che non sia proprio si dice **improprio** (*improper*) o **non proprio**

Alberi binari e sottoalberi

- Il sottoalbero avente radice nel figlio sinistro/destro di un nodo interno v si chiama **sottoalbero sinistro/destro di v**
- Ricordiamo la definizione di sottoalbero in un albero generico
 - **Dato l'albero T , il suo sottoalbero (*subtree*) avente come radice $v \in T$ è l'albero costituito da tutti e soli i discendenti di v (tra i quali c'è, per definizione, v stesso)**
 - **Quindi, un sottoalbero è, per definizione, un albero non vuoto (dato che ha la radice...)**
- In un albero binario, ciascun nodo v può avere al massimo due sottoalberi, il sottoalbero sinistro (che ha radice nel figlio sinistro di v , **se esiste**) e il sottoalbero destro (che ha radice nel figlio destro di v , **se esiste**)
- A volte, nell'enunciare proprietà e nella loro dimostrazione, può far comodo ipotizzare che, in un albero binario, **ciascun nodo abbia SEMPRE entrambi i suoi possibili sottoalberi**, sinistro e destro, con la convenzione di ritenere **vuoto** (e, quindi, di altezza zero), un sottoalbero "**mancante**"
 - È un'eccezione che useremo anche in questo corso

Albero binario: attraversamento in ordine (simmetrico)

□ **Algoritmo (ricorsivo)**

□ **inorderTraversal($T \neq \emptyset, v \in T$) // con il solito bootstrap**

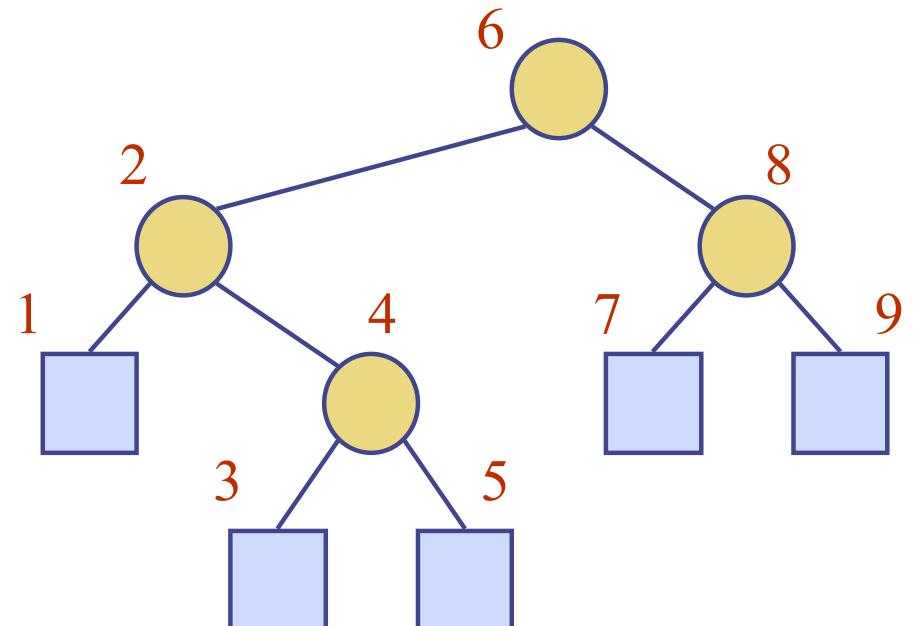
- if v ha il figlio sinistro $u \in T$
 - inorderTraversal(T, u)
- **visit(T, v)**
- if v ha il figlio destro $w \in T$
 - inorderTraversal(T, w)

*Inorder traversal:
attraversamento*

**“in ordine simmetrico” o,
semplicemente “in ordine”**

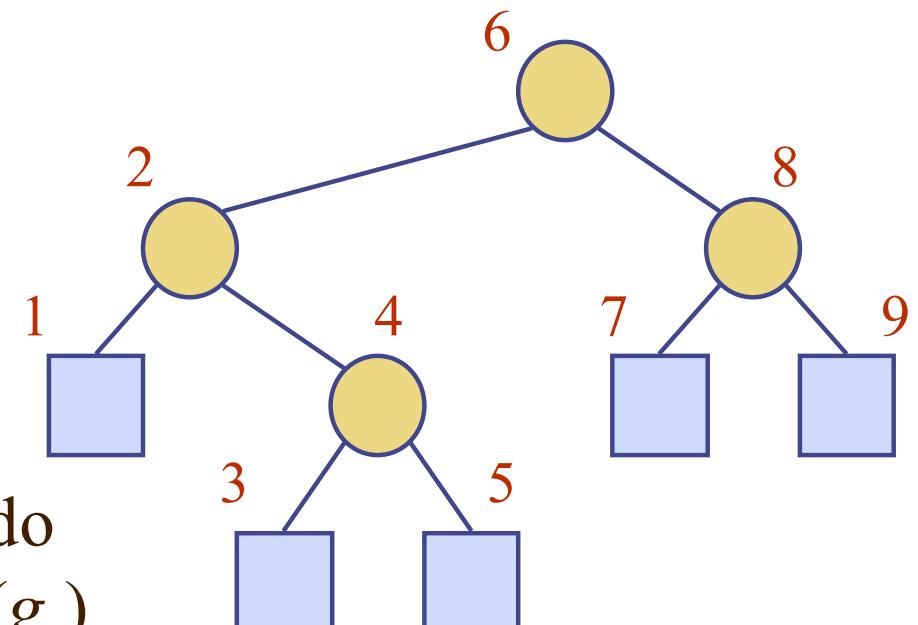
Si dimostra che anche questo è un attraversamento, perché ogni nodo viene visitato una e una sola volta

Questo attraversamento è definito SOLO per alberi binari !!



Attraversamento in ordine simmetrico

- Se alla consueta rappresentazione grafica degli alberi aggiungiamo il vincolo che **il sottoalbero sinistro/destro di ciascun nodo sia disegnato completamente a sinistra/destra del nodo stesso** (oltre che più in basso)
 - L'attraversamento in ordine simmetrico visita i nodi **“da sinistra a destra”**, indipendentemente dalla loro profondità
- Con dimostrazione analoga a quella vista per pre-ordine e post-ordine, anche l'attraversamento in ordine simmetrico viene **eseguito in un tempo $\Theta(n)$** , quando l'azione di visita è $\Theta(1)$ oppure $O(g_v)$



Alberi binari: Alcune proprietà

□ In un albero binario (e in un albero di grado due)

- Il livello 0 ha al massimo un nodo (la radice, se l'albero non è vuoto)
- Il livello 1 ha al massimo 2 nodi (i figli della radice)
- Il livello 2 ha al massimo 4 nodi (i figli dei due nodi di livello 1)
- Il livello **3** ha al massimo **8** nodi (i figli dei quattro nodi di livello 2)
- Il livello **4** ha al massimo **16** nodi (i figli degli otto nodi di livello 3)
- ...
- **Il livello d ha**, quindi, **al massimo 2^d nodi**
- Da questa semplice proprietà derivano molte relazioni utili per un albero binario non vuoto T

□ In modo analogo si osserva che, **in un albero di grado m , il livello d ha al massimo m^d nodi**

Alberi binari: Alcune proprietà

□ Usiamo questi simboli

- n_E , numero di nodi esterni (foglie) dell'albero
- n_I , numero di nodi interni dell'albero
- n , dimensione dell'albero, ovviamente $n = n_E + n_I$
- h , altezza dell'albero
- $h(v)$ oppure h_v , altezza del nodo v
- $S_L(v)$, sottoalbero sinistro (eventualmente vuoto) di v
 - Per brevità, se v è la radice r potremo scrivere semplicemente S_L anziché $S_L(r)$
- $S_R(v)$, sottoalbero destro (eventualmente vuoto) di v
 - Per brevità, se v è la radice r potremo scrivere semplicemente S_R anziché $S_R(r)$
- $h_L(v)$, altezza di $S_L(v)$ [h_L , altezza di $S_L = S_L(r)$]
- $h_R(v)$, altezza di $S_R(v)$ [h_R , altezza di $S_R = S_R(r)$]

Alberi binari: Alcune proprietà

□ **Si può dimostrare** che in un albero binario
NON VUOTO

- $1 \leq n_E \leq 2^h$
- $h \leq n_I \leq 2^h - 1$
- $h + 1 \leq n \leq 2^{h+1} - 1$
- $\log_2 (n + 1) - 1 \leq h \leq n - 1$

Esercizio:
Dimostrare
queste proprietà

Alberi binari: Alcune proprietà

- **Dimostrazione:** In un albero binario **NON VUOTO**, $1 \leq n_E \leq 2^h$
- Usiamo l'induzione forte su h
 - Caso base: $h = 0 \Rightarrow 1 \leq n_E \leq 2^0 = 2^0 = 1 \Leftrightarrow n_E = 1$
 - Quindi, per $h = 0$ la tesi si riduce a $n_E = 1$
 - Dimostrazione del caso base:
 - Se l'albero ha altezza zero, allora la sua radice ha altezza zero (per definizione dell'altezza di un albero, che è uguale all'altezza della sua radice)
 - Se un nodo (in questo caso la radice) ha altezza zero, allora è una foglia (per definizione di altezza di un nodo), quindi la radice è una foglia
 - Se la radice di un albero è una foglia, non ha discendenti propri: dato che tutti i nodi di un albero sono discendenti della radice, l'albero ha soltanto la radice, quindi ($n =$) $n_E = 1$, come si voleva dimostrare

- **Dimostrazione:** In un albero binario **NON VUOTO**, $1 \leq n_E \leq 2^h$
- Usiamo l'induzione forte su h e dimostriamo che: **se la proprietà è vera per alberi di altezza minore di h , allora è vera per alberi di altezza $h > 0$**
 - Se l'albero T ha altezza $h > 0$, la sua radice r , avendo altezza $h_r = h > 0$, non è una foglia, quindi, per definizione, $h = h_r = 1 + \max \{h_L, h_R\}$
[uno dei due sottoalberi, S_L o S_R , potrebbe non esistere]
 - Definiamo $E_L = \{v \in S_L \mid T.\text{isExternal}(v)\}$ e $E_R = \{v \in S_R \mid T.\text{isExternal}(v)\}$
 - Qui è comodo ritenere vuoto l'eventuale sottoalbero non esistente...
 - Siano $n_{EL} = |E_L|$ e $n_{ER} = |E_R|$ le cardinalità dei due insiemi, E_L e E_R
 - Se $S_L \neq \emptyset$, h_L è, per definizione, uguale all'altezza della radice di S_L , la quale è figlia di r , quindi $h_L < h = h_r$ (la definizione di $h_r \Rightarrow h_r \geq 1 + h_L > h_L$)
 - Quindi, per $S_L \neq \emptyset$, vale (per ipotesi induttiva) la proprietà $1 \leq n_{EL} \leq 2^{hL}$; se, invece, $S_L = \emptyset$ (cioè non esiste), allora certamente non ha foglie, cioè $n_{EL} = 0$
 - Analogamente per S_R
 - Ricordiamo che almeno uno dei due sottoalberi della radice deve esistere (altrimenti sarebbe $h = 0$), cioè $S_L \cup S_R \neq \emptyset$

[continua]

- **Dimostrazione:** In un albero binario **NON VUOTO**, $1 \leq n_E \leq 2^h$
- Usiamo l'induzione forte su h e dimostriamo che: **se la proprietà è vera per alberi di altezza minore di h , allora è vera per alberi di altezza $h > 0$**

[continua]

- L'insieme dei nodi esterni di T è l'unione degli insiemi dei nodi esterni dei sottoalberi della radice r (perché r , che è l'unico nodo di T che non appartiene ai sottoalberi di r , non è un nodo esterno, altrimenti sarebbe $h = 0$), quindi $n_E = n_{EL} + n_{ER}$ [la proprietà "essere un nodo esterno" non dipende dall'albero a cui si appartiene, dipende soltanto dal fatto di non avere figli]
- Qual è il valore minimo di n_E ? Se i due sottoalberi della radice esistono, allora $n_E = n_{EL} + n_{ER} \geq 2$ (perché $n_{EL} \geq 1$ e $n_{ER} \geq 1$), ma se uno dei due non esiste (e l'altro necessariamente esiste) allora $n_E \geq 1$. Quindi, come minimo, $\textcolor{blue}{n_E \geq 1}$.
- Qual è il valore massimo di n_E ? Quando entrambi i sottoalberi della radice esistono: $\textcolor{blue}{n_E = n_{EL} + n_{ER} \leq 2^{hL} + 2^{hR} \leq 2 \cdot 2^{\max\{hL, hR\}} = 2^{1+\max\{hL, hR\}} = 2^h}$ [se uno dei due non esiste, si ottiene un limite superiore più basso, quindi inutile]
- Quindi, riassumendo, si ha la tesi: $1 \leq \textcolor{blue}{n_E} \leq 2^h$

Alberi binari: Alcune proprietà

- **Dimostrazione:** In un albero binario **NON VUOTO**, $h \leq n_I \leq 2^h - 1$
- Usiamo l'induzione forte su h
 - Caso base: $h = 0 \Rightarrow 0 = h \leq n_I \leq 2^h - 1 = 2^0 - 1 = 0 \Leftrightarrow n_I = 0$
 - Dimostrazione del caso base:
 - Se l'albero ha altezza zero, allora la sua radice ha altezza zero (per definizione dell'altezza di un albero, che è uguale all'altezza della sua radice)
 - Se un nodo (in questo caso la radice) ha altezza zero, allora è una foglia (per definizione di altezza di un nodo), quindi la radice è una foglia
 - Se la radice di un albero è una foglia, non ha discendenti propri: dato che tutti i nodi di un albero sono discendenti della radice, l'albero ha soltanto la radice (che **non** è un nodo interno), quindi $n_I = 0$, come si voleva dimostrare

- **Dimostrazione:** In un albero binario **NON VUOTO**, $h \leq n_I \leq 2^h - 1$
- Usiamo l'induzione forte su h e dimostriamo che: **se la proprietà è vera per alberi di altezza minore di h , allora è vera per alberi di altezza $h > 0$**
 - Se l'albero T ha altezza $h > 0$, la sua radice r , avendo altezza $h_r = h > 0$, non è una foglia, quindi, per definizione, $h = h_r = 1 + \max \{h_L, h_R\}$
[uno dei due sottoalberi, S_L o S_R , potrebbe non esistere]
 - Definiamo $I_L = \{v \in S_L \mid T.\text{isInternal}(v)\}$ e $I_R = \{v \in S_R \mid T.\text{isInternal}(v)\}$
 - Siano $n_{IL} = |I_L|$ e $n_{IR} = |I_R|$ le cardinalità dei due insiemi, I_L e I_R
 - Se $S_L \neq \emptyset$, h_L è, per definizione, uguale all'altezza della radice di S_L , la quale è figlia di r , quindi $h_L < h = h_r$ (la definizione di $h_r \Rightarrow h_r \geq 1 + h_L > h_L$)
 - Quindi, per $S_L \neq \emptyset$, vale (per ipotesi induttiva) la proprietà $h_L \leq n_{IL} \leq 2^{hL} - 1$; se, invece, $S_L = \emptyset$ (cioè non esiste), allora non ha nodi interni, $n_{IL} = 0$
 - Analogamente per S_R
 - Ricordiamo che almeno uno dei due sottoalberi della radice deve esistere (altrimenti sarebbe $h = 0$), cioè $S_L \cup S_R \neq \emptyset$

[continua]

- **Dimostrazione:** In un albero binario **NON VUOTO**, $h \leq n_I \leq 2^h - 1$
- Usiamo l'induzione forte su h e dimostriamo che: **se la proprietà è vera per alberi di altezza minore di h , allora è vera per alberi di altezza $h > 0$**

[continua]

- L'insieme dei nodi interni di T è l'unione degli insiemi dei nodi interni dei sottoalberi della radice r , **a cui va aggiunta la radice stessa**, quindi
 $n_I = n_{IL} + n_{IR} + 1$
- Qual è il valore minimo di n_I ? Se esiste un solo sottoalbero della radice, supponiamo, senza ledere la generalità, che questo sia S_L : allora
 $n_I = n_{IL} + n_{IR} + 1 \geq h_L + 1 = h$ (perché $n_{IL} \geq h_L$ e $n_{IR} = 0$).
[se esistono entrambi, si ottiene un limite inferiore più alto, quindi inutile]
- Qual è il valore massimo di n_I ? Quando entrambi i sottoalberi della radice esistono: $n_I = n_{IL} + n_{IR} + 1 \leq (2^{hL} - 1) + (2^{hR} - 1) + 1 \leq 2 \cdot 2^{\max\{hL, hR\}} - 1 = 2^{1+\max\{hL, hR\}} - 1 = 2^h - 1$
[se uno dei due non esiste, si ottiene un limite superiore più basso, quindi inutile]
- Quindi, riassumendo, si ha la tesi: $h \leq n_I \leq 2^h - 1$

Alberi binari: Alcune proprietà

□ Si può dimostrare che in un albero binario **NON VUOTO**

- $1 \leq n_E \leq 2^h$ [dimostrato]
- $h \leq n_I \leq 2^h - 1$ [dimostrato]
- $h + 1 \leq n \leq 2^{h+1} - 1$
- $\log_2 (n + 1) - 1 \leq h \leq n - 1$

□ Non sono tutte da dimostrare... dopo aver dimostrato le prime due, si possono sommare membro a membro

$$1 + h \leq n_E + n_I \leq 2^h + 2^h - 1 = 2 \cdot 2^h - 1 = 2^{h+1} - 1$$

$$\Rightarrow h + 1 \leq n \leq 2^{h+1} - 1 \text{ perché } n = n_E + n_I$$

Alberi binari: Alcune proprietà

□ Si può dimostrare che in un albero binario **NON VUOTO**

- $1 \leq n_E \leq 2^h$
- $h \leq n_I \leq 2^h - 1$
- $\textcolor{red}{h + 1} \leq n \leq \textcolor{blue}{2^{h+1} - 1}$
- $\log_2 (n + 1) - 1 \leq h \leq n - 1$

□ Avendo dimostrato la terza proprietà, si può ora agevolmente dimostrare la quarta

$$\textcolor{red}{h + 1} \leq n \Rightarrow h \leq n - 1$$

$$n \leq \textcolor{blue}{2^{h+1} - 1} \Rightarrow n + 1 \leq 2^{h+1}$$

$$\Rightarrow \log_2 (n + 1) \leq \log_2 (2^{h+1}) = (h+1)\log_2 2 = h + 1$$

$$\Rightarrow \log_2 (n + 1) - 1 \leq h$$

Quindi, componendo i due risultati: $\log_2 (n + 1) - 1 \leq h \leq n - 1$

Alberi binari: Alcune proprietà

□ **Abbiamo dimostrato** che in un albero binario

NON VUOTO

- $1 \leq n_E \leq 2^h$
- $h \leq n_I \leq 2^h - 1$
- $h + 1 \leq n \leq 2^{h+1} - 1$
- $\log_2 (n + 1) - 1 \leq h \leq n - 1$

Riusciamo a ricordarle meglio se
le comprendiamo...
E le comprendiamo meglio se le
visualizziamo...

□ **Che forma hanno gli alberi che costituiscono i casi limite per ciascuna disequazione?**

- Inoltre: i casi limite (dimostrati) sono raggiungibili?

□ Cominciamo da questa: **$h + 1 = n$**

Alberi binari: Alcune proprietà

- Com'è fatto un albero binario in cui $n = h + 1$?

□ Se $h = 0$ e $n = 1$ \longrightarrow A

\Rightarrow L'albero ha soltanto la radice

□ Se $h = 1$ e $n = 2$

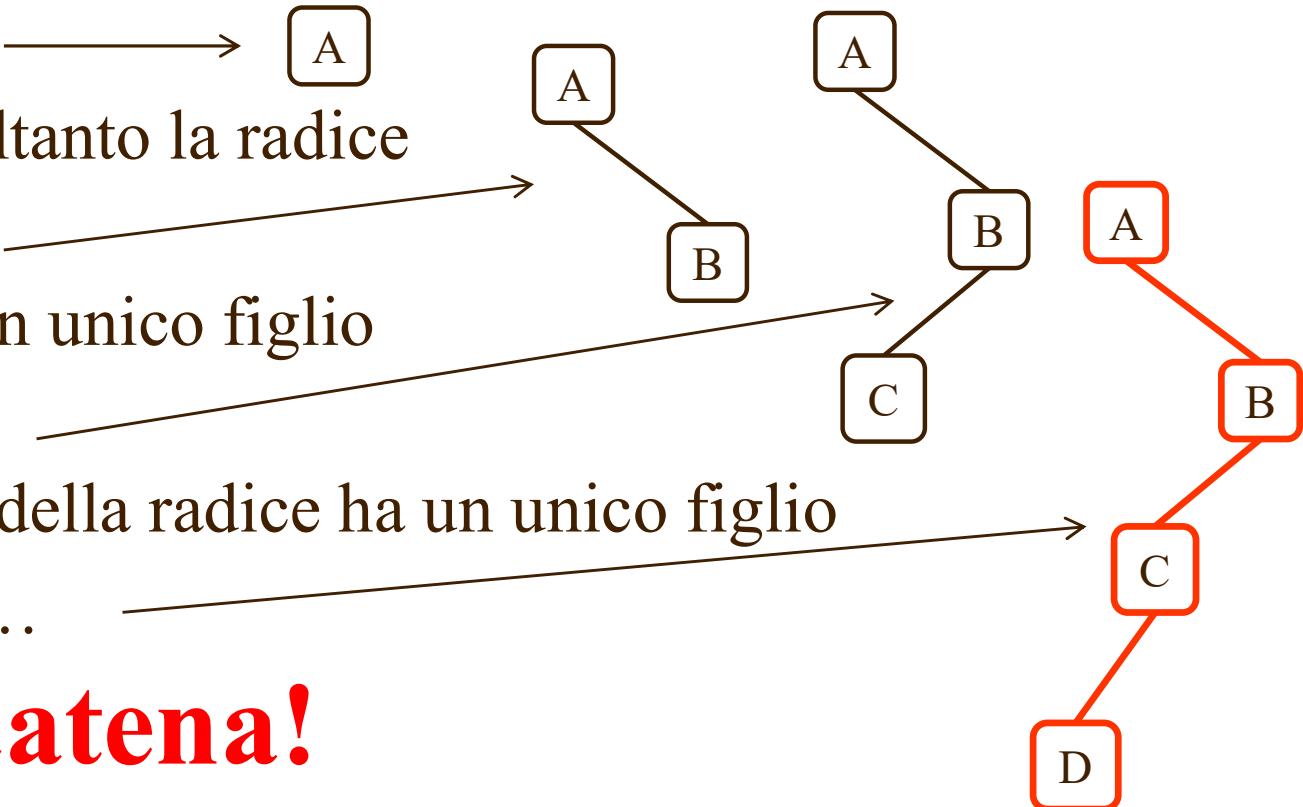
\Rightarrow La radice ha un unico figlio

□ Se $h = 2$ e $n = 3$

\Rightarrow L'unico figlio della radice ha un unico figlio

□ Se $h = 3$ e $n = 4$...

\Rightarrow è una catena!



- Come è facile verificare, lo stesso *albero-limite* vale per i casi limite delle altre disequazioni: $n_I = h$, $n_E = 1$ e $h = n - 1$

Alberi binari: Alcune proprietà

□ Com'è fatto un albero binario in cui $n = 2^{h+1} - 1$?

□ Se $h = 0$ e $n = 1$ \longrightarrow A

\Rightarrow L'albero ha soltanto la radice

□ Se $h = 1$ e $n = 3$

\Rightarrow Oltre alla radice, l'albero ha 2 nodi al livello 1

□ Se $h = 2$ e $n = 7$

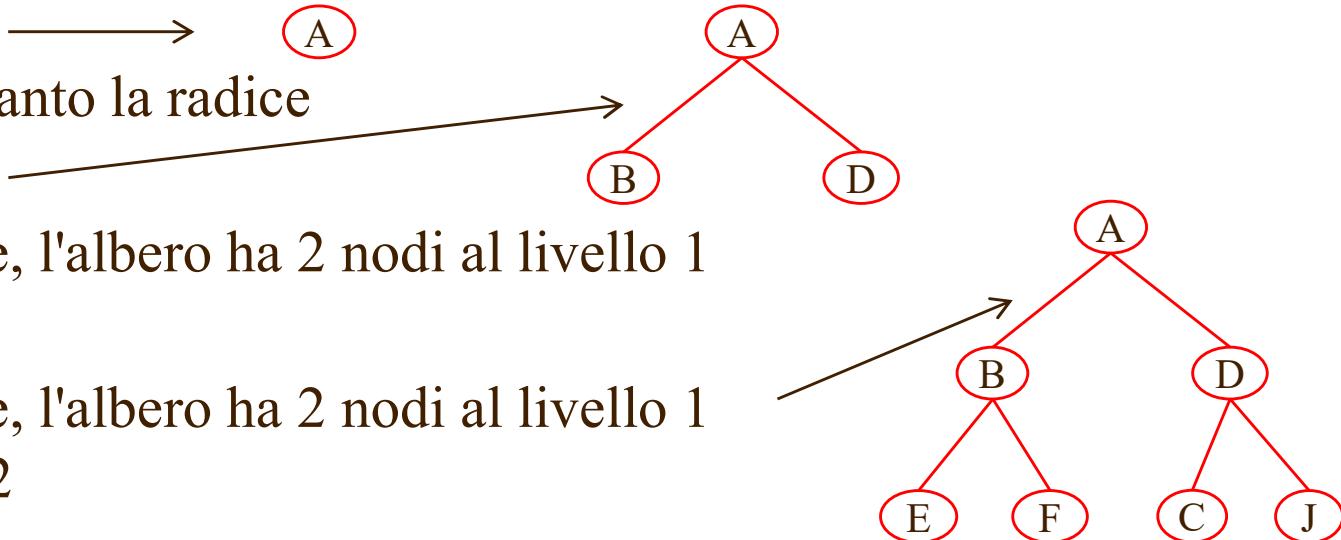
\Rightarrow Oltre alla radice, l'albero ha 2 nodi al livello 1
e 4 nodi al livello 2

□ Se $h = 3$ e $n = 15$

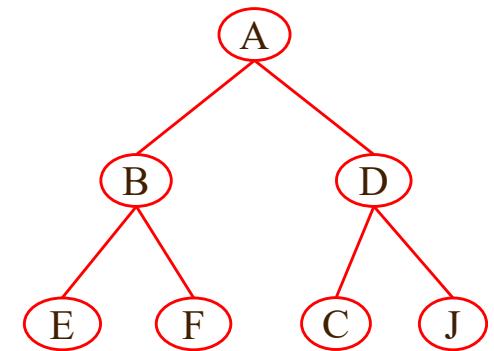
\Rightarrow Oltre alla radice (cioè 2^0 nodi al livello 0), l'albero ha
 2^1 nodi al livello 1, 2^2 nodi al livello 2 e 2^3 nodi al livello 3

□ In generale, un albero binario di altezza h che abbia
 $n = 2^{h+1} - 1$ nodi, ha 2^i nodi al livello $i \forall i = 0, \dots, h$, cioè
ha il massimo numero di nodi in ciascuno dei suoi livelli

□ Lo stesso albero-limite vale per i casi limite delle altre
disequazioni: $n_I = 2^h - 1$, $n_E = 2^h$ e $h = \log_2 (n + 1) - 1$



Alberi binari triangolari



- Un albero binario che abbia il massimo numero di nodi in ciascuno dei suoi livelli si dice **triangolare** (per evidente analogia geometrica) e gode di alcune utili proprietà (facilmente dimostrabili, alcune già dimostrate...)

- Se ha altezza h , allora ha 2^h foglie, che si trovano tutte allo stesso livello, il livello h
- Tra gli alberi di altezza h , quello triangolare ha il massimo numero di foglie (2^h) e il massimo numero di nodi interni ($2^h - 1$)
- Non può avere dimensione qualsiasi: ha un numero di nodi, n , uguale a una potenza di 2 diminuita di 1
 - Tra tutti gli alberi aventi tale dimensione n , quello triangolare ha altezza minima, $h = \log_2(n + 1) - 1$
- Tutti i suoi nodi interni hanno due figli, cioè è **un albero proprio**
- Come vedremo, è anche un caso particolare di albero **completo**

Alberi binari PROPRI: Alcune proprietà

- Se l'albero è **proprio**, si può dimostrare che **alcuni vincoli** diventano più stringenti
 - $\mathbf{h + 1} \leq n_E \leq 2^h}$ (le altre modifiche sono conseguenti...)
 - $h \leq n_I \leq 2^h - 1$
 - $\mathbf{2h + 1} \leq n \leq 2^{h+1} - 1$
 - $\log_2 (n + 1) - 1 \leq h \leq \mathbf{(n - 1)/2}$
 - Inoltre: $\mathbf{n_E = n_I + 1}$
(quindi $n = n_E + n_I = 2n_I + 1$ è dispari)

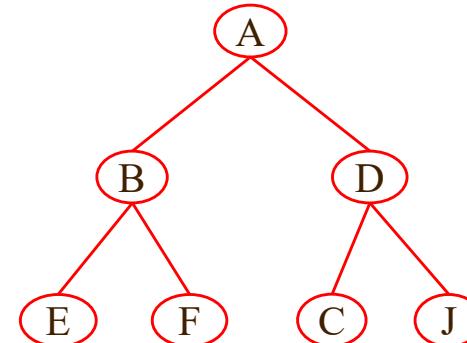
Riusciamo a ricordarle meglio se
le comprendiamo...

E le comprendiamo meglio se le
visualizziamo...

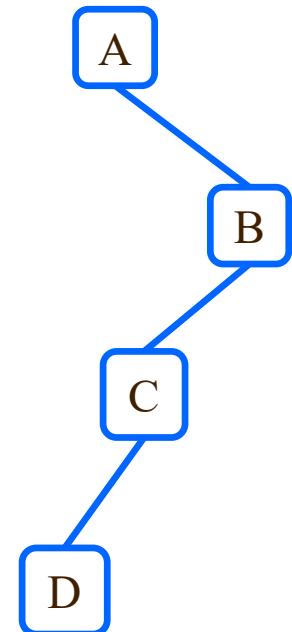
Alberi binari propri: Alcune proprietà

□ Alcune disequazioni rimangono valide

- $h + 1 \leq n_E \leq 2^h$
- $h \leq n_I \leq 2^h - 1$
- $2h + 1 \leq n \leq 2^{h+1} - 1$
- $\log_2 (n + 1) - 1 \leq h \leq (n - 1)/2$



□ Si riferiscono al caso limite di **albero triangolare**, che è anche un albero proprio, quindi è "ovvio" che rimangano valide



□ L'altro caso limite era costituito da un albero che "degenera" in **una catena**, che, però, NON è un albero proprio e, quindi, non può essere di nuovo il caso limite

Alberi binari propri: Alcune proprietà

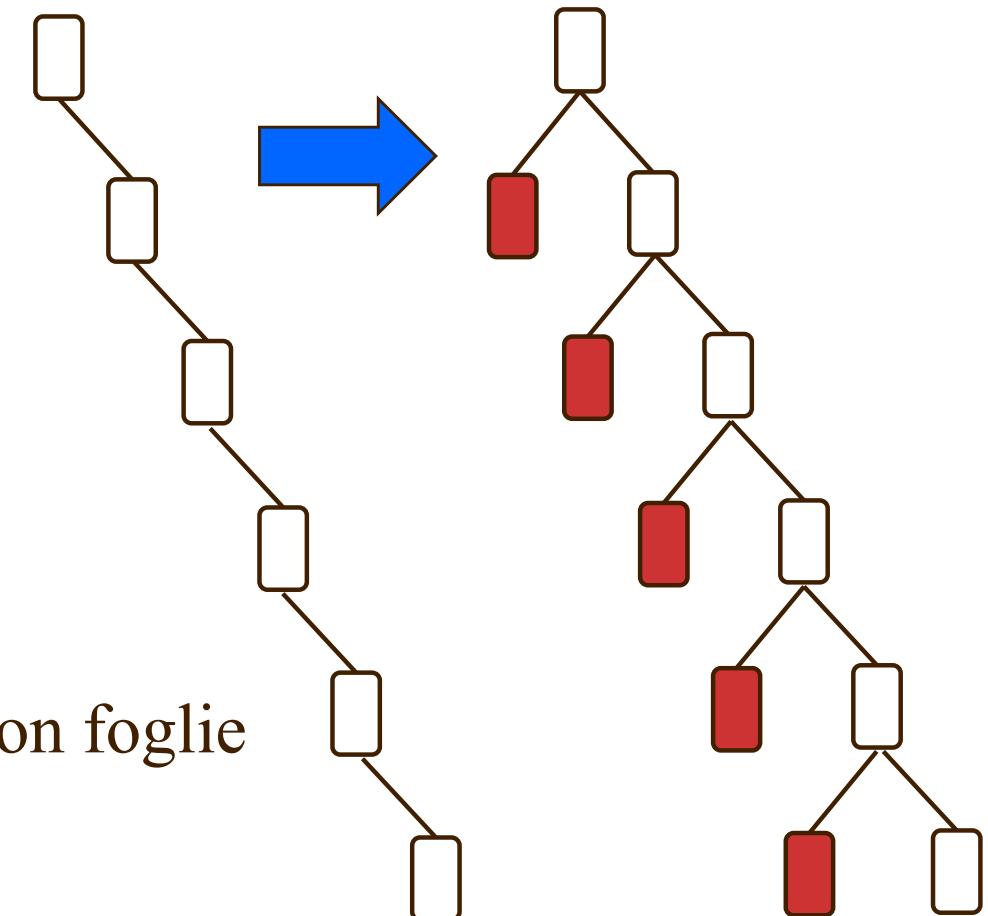
□ Quale sarà l'albero limite per le altre disequazioni?

- $h + 1 \leq n_E \leq 2^h$
- $h \leq n_I \leq 2^h - 1$
- $2h + 1 \leq n \leq 2^{h+1} - 1$
- $\log_2 (n + 1) - 1 \leq h \leq (n - 1)/2$

□ Deve ancora essere un albero
molto alto ma con pochi nodi...

□ Proviamo a "**rendere proprio**"
l'albero-catena... completandolo con foglie

□ **Fare la dimostrazione**



L'albero binario come ADT

- Ricordiamo che **un albero binario è un albero!**

```
public interface BinaryTree<T>
    extends Tree<T>
{
    Position<T> left(Position<T> v)
        throws InvalidPositionException,
               BoundaryViolationException;
    Position<T> right(Position<T> v)
        throws InvalidPositionException ,
               BoundaryViolationException;
    boolean hasLeft(Position<T> v)
        throws InvalidPositionException;
    boolean hasRight(Position<T> v)
        throws InvalidPositionException;
} // c'è anche children, ovviamente
```

Ancora privo
di metodi di
inserimento,
quindi
inutilizzabile

Lezione 21

Il nodo di un albero binario

- Anche per realizzare alberi binari si usa **solitamente** una struttura a **nodi concatenati**, ma si usa un tipo di nodo diverso da quello generico, perché qui tutti i nodi hanno **grado due** e usare una lista di figli sarebbe uno spreco

```
class BTNode<T> implements Position<T>
{ private T element;
  private BTNode<T> parent;
  private BTNode<T> left; // non serve una lista
  private BTNode<T> right; // di figli!
  ... // metodi get/set per tutte le
      // variabili di esemplare (e metodo element())
}
```

- Se un figlio è assente, il riferimento corrispondente vale **null** (come accade per **parent** nella radice)

Realizzazione di albero binario

- Per realizzare mediante una struttura concatenata un albero binario (**LinkedBinaryTree**) che sia utilizzabile, aggiungiamo metodi ulteriori rispetto all'interfaccia **BinaryTree**

```
public class LinkedBinaryTree<T> implements BinaryTree<T>
{ private int size;
  private BTNode<T> root;
  private class BTNode<T> implements Position<T>
  { ... }
  public LinkedBinaryTree() { root = null; size = 0; }
  ... // metodi dell'interfaccia BinaryTree, più...
  public Position<T> addRoot(T elem)
    throws NonEmptyTreeException{...} // ha già la radice!
  public Position<T> addLeft(Position<T> v, T elem)
    throws InvalidPositionException{...}
    // aggiunge un nuovo nodo come figlio sinistro di v
  public Position<T> addRight(Position<T> v, T elem)
    throws InvalidPositionException{...}
    // aggiunge un nuovo nodo come figlio destro di v
}
```

Realizzazione di albero binario

```
public Position<T> addRoot(T elem)
    throws NonEmptyTreeException{...}
public Position<T> addLeft(Position<T> v, T elem)
    throws InvalidPositionException{...}
public Position<T> addRight(Position<T> v, T elem)
    throws InvalidPositionException{...}
```

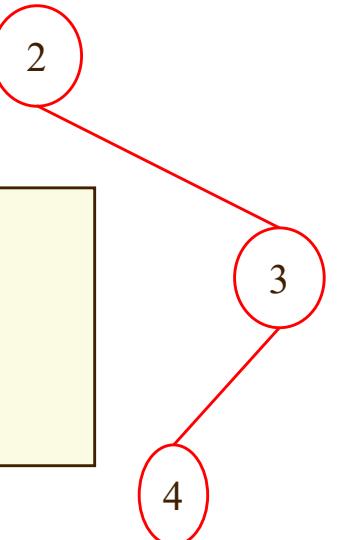
- Non è **necessario** che **addroot**, **addLeft** e **addRight** restituiscano **Position**, potrebbero essere **void**, ma così il loro utilizzo è più comodo, perché, ad esempio, li possiamo “concatenare”

```
LinkedBinaryTree<Integer> t =
    new LinkedBinaryTree<Integer>();
t.addLeft(t.addRight(t.addRoot(2), 3), 4);
// si crea l'albero qui raffigurato
```

2

3

4



Albero binario un po' più comodo

- Per agevolare l'utilizzo dell'albero, si possono aggiungere **ulteriori metodi**, non indispensabili

```
public class LinkedBinaryTree<T> implements BinaryTree<T>
{
    ...
    // ha senso perché in un albero binario (quindi di
    // grado 2) ogni nodo ha al massimo un solo fratello
    public Position<T> sibling(Position<T> v)
        throws InvalidPositionException,
               BoundaryViolationException{...} // se v non ha
                                         // fratello
    public T remove(Position<T> v) // seguono commenti...
        throws InvalidPositionException{...}
}
```

Albero binario: `remove`

- Il metodo `remove` può agire soltanto su un nodo v che **NON** abbia due figli
 - Altrimenti lancia **InvalidPositionException**
(anche se forse sarebbe meglio usare un'eccezione diversa...)
- Se v non ha figli
 - **Se v è la radice, l'albero diventa vuoto**, altrimenti bisogna decrementare la dimensione dell'albero e in **parent(v)** rendere **null** il riferimento **left** o **right** (dipende da quale è uguale a v)
- Se v ha un (solo) figlio
 - **Se v è la radice, il suo unico figlio diventa la nuova radice** (e si decrementa la dimensione dell'albero)
 - Altrimenti, bisogna fare in modo che l'unico figlio w di v diventi figlio di **parent(v)**, decrementando la dimensione dell'albero e aggiornando i due riferimenti coinvolti, in w e nel genitore di v
 - Se v era figlio sinistro/destro di **parent(v)**,
 w diventa figlio sinistro/destro di **parent(v)**

Albero binario: Prestazioni

- Per le prestazioni valgono le stesse considerazioni viste per gli alberi generici, **tranne**
 - Il metodo **children** è sempre $\Theta(1)$, perché i figli sono al massimo due
 - I metodi modificatori (**addRoot**, **addLeft**, **addRight**, **remove**) sono $\Theta(1)$, perché richiedono la modifica di un numero di collegamenti che NON dipende dalla dimensione dell’albero (“**operano localmente**”)
- Occupazione di memoria: anche in questo caso è $\Theta(n)$ per un albero di dimensione n , cioè è ottima

Realizzazione di albero binario

- Esempio di implementazione di alcuni metodi

```
public class LinkedBinaryTree<T> implements BinaryTree<T>
{ private int size = 0;
  private BTNode<T> root = null;
  // costruttore inutile...
  private class BTNode<T> implements Position<T> { ... }
  private BTNode<T> checkPosition(Position<T> v)
    throws InvalidPositionException
  { ... } // come nelle liste concatenate
  public boolean hasLeft(Position<T> v)
  { BTNode<T> n = checkPosition(v); // throws...
    return (n.left != null)
  }
  public Position<T> addRoot(T elem)
  { if (!isEmpty()) throw new NonEmptyTreeException();
    root = new BTNode<T>(elem);
    size++;
    return root; // BTNode implementa Position
  }
  ...
}
```

Realizzazione di albero binario

- Esempio di implementazione di alcuni metodi

```
public class LinkedBinaryTree<T> implements BinaryTree<T>
{
    ...
    public Position<T> addLeft(Position<T> v, T elem)
    {   if (hasLeft(v)) // controlla anche se v è valida
        throw new InvalidPositionException();
        BTNode<T> p = (BTNode<T>) v; // v è già valida,
                                         // inutile invocare
                                         // checkPosition
        BTNode<T> n = new BTNode<T>(elem);
        p.setLeft(n);
        n.setParent(p);
        size++;
        return n;
    }
    ...
}
```

Albero binario in un vettore!

- Per realizzare un albero binario si usa **solitamente** una struttura a nodi concatenati
- **Altrimenti?**
 - Si può usare un array (o, per meglio dire, **una lista con indice**)
- Si definisce una funzione $p(v)$ che **assegna un indice a ogni nodo** v di un albero binario T
 - Se v è la radice di T , allora $p(v) = 1$
 - Altrimenti, se v è il figlio sinistro di $u = T.\text{parent}(v)$, allora $p(v) = 2 p(u)$
 - Altrimenti, v è il figlio destro di $u = T.\text{parent}(v)$ e $p(v) = 2 p(u) + 1$

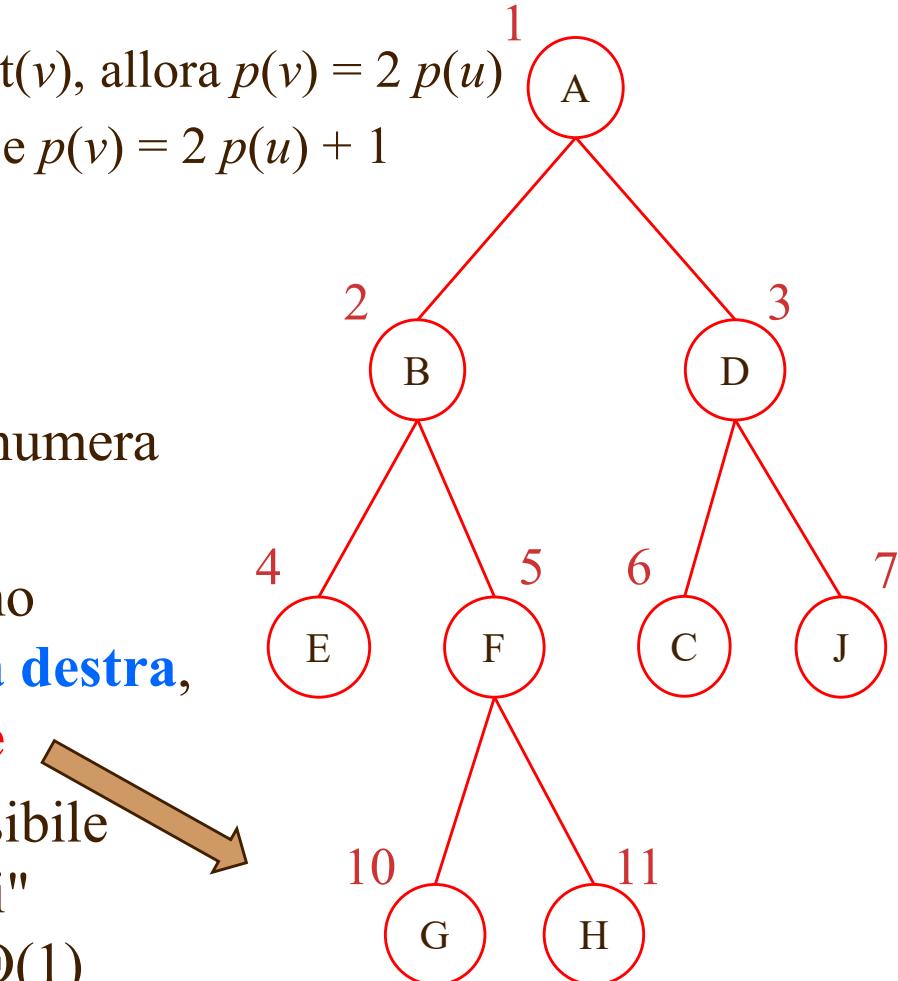
In effetti, è una lista con **rango**...

Il libro propone una numerazione che parte da zero: è un'alternativa

Albero binario in un vettore!

- Se v è la radice di T , allora $p(v) = 1$
- Altrimenti, se v è il figlio sinistro di $u = T.\text{parent}(v)$, allora $p(v) = 2 p(u)$
- Altrimenti, v è il figlio destro di $u = T.\text{parent}(v)$ e $p(v) = 2 p(u) + 1$
- Di conseguenza, $\forall v \neq T.\text{root}()$
$$p(T.\text{parent}(v)) = \lfloor p(v)/2 \rfloor$$

- **Proprietà:** La funzione di numerazione p numera i nodi in ordine crescente **per livelli**
- **Proprietà:** I nodi di ciascun livello vengono numerati in **ordine crescente da sinistra a destra**, anche se **alcuni numeri possono mancare**
- **Proprietà:** Dato l'indice di un nodo, è possibile calcolare l'indice dei suoi "parenti prossimi" (figli e genitore, se esistono) in un tempo $\Theta(1)$
 - Quindi, ogni passo elementare di "navigazione" nell'albero (cioè lo spostamento lungo un ramo) è $\Theta(1)$, come negli alberi concatenati



Albero binario in un vettore!

- Se v è la radice di T , allora $p(v) = 1$
 - Altrimenti, se v è il figlio sinistro di $u = T.\text{parent}(v)$, allora $p(v) = \mathbf{2} p(u)$
 - Altrimenti, v è il figlio destro di $u = T.\text{parent}(v)$ e $p(v) = \mathbf{2} p(u) + 1$
- $\forall v \neq T.\text{root}(), \quad p(T.\text{parent}(v)) = \lfloor p(v)/2 \rfloor$
- Si noti che le **operazioni di calcolo** degli indici necessari per navigare nell'albero sono **estremamente veloci**, se eseguite su numeri interi rappresentati in binario posizionale assoluto (o in complemento a due, che, per numeri positivi, è la stessa cosa)
 - Moltiplicare per 2 equivale a uno scorrimento (*shift left*) verso sinistra, introducendo uno zero da destra
 - Aggiungere 1 al risultato di una moltiplicazione per 2 (cioè aggiungere 1 a un numero pari) equivale a impostare a 1 il bit più a destra: non serve aggiungere 1, basta fare un OR bit per bit con una maschera composta da tutti zeri e un solo uno nel bit più a destra
 - Dividere per 2 (troncando all'intero inferiore) equivale a uno scorrimento verso destra (*shift right*)

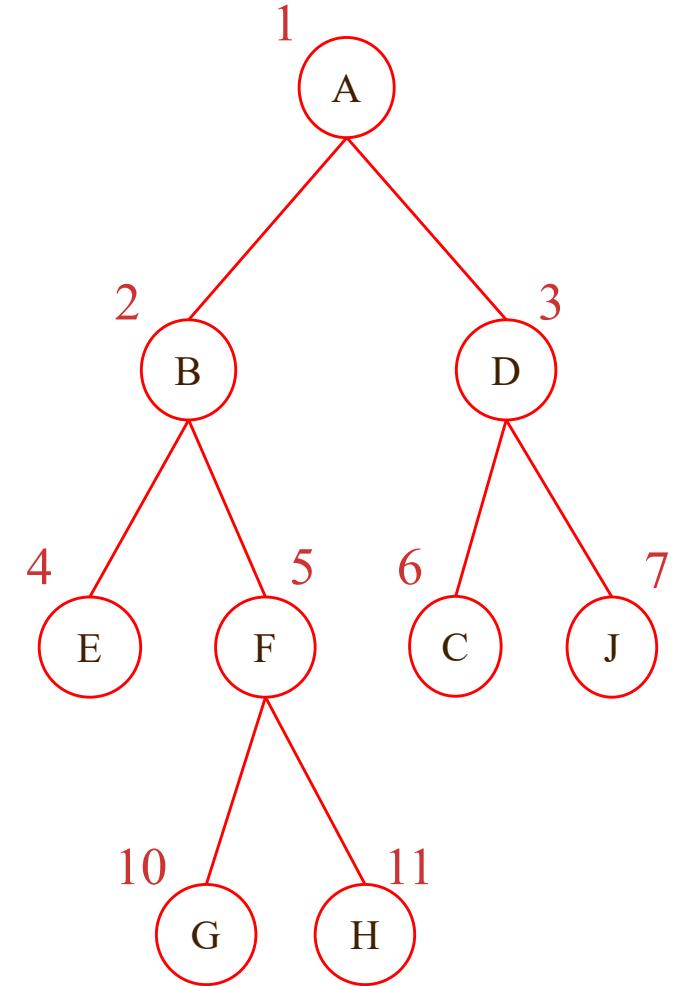
Albero binario in un vettore!

- Avendo assegnato un indice a ogni nodo dell'albero binario, possiamo associare a ciascuna posizione all'interno di un vettore V un (riferimento a un) dato contenuto nell'albero (attenzione: **è un vettore DI DATI, non DI NODI, i nodi NON CI SONO**)



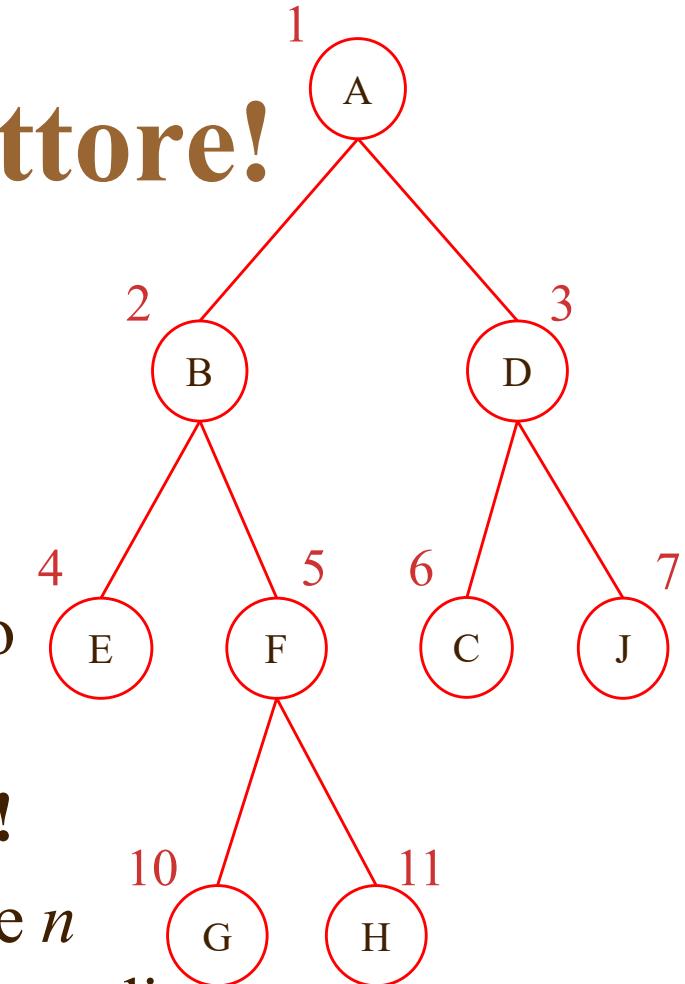
- Dato che gli indici assegnati iniziano, per definizione di p , dal valore 1, sarebbe più opportuno usare una **lista con rango** (ma non cambia molto, con il vettore si “spreca” una cella)

Naturalmente si può definire p in modo che i numeri inizino da 0, ma le formule si complicano un po' (inutilmente)



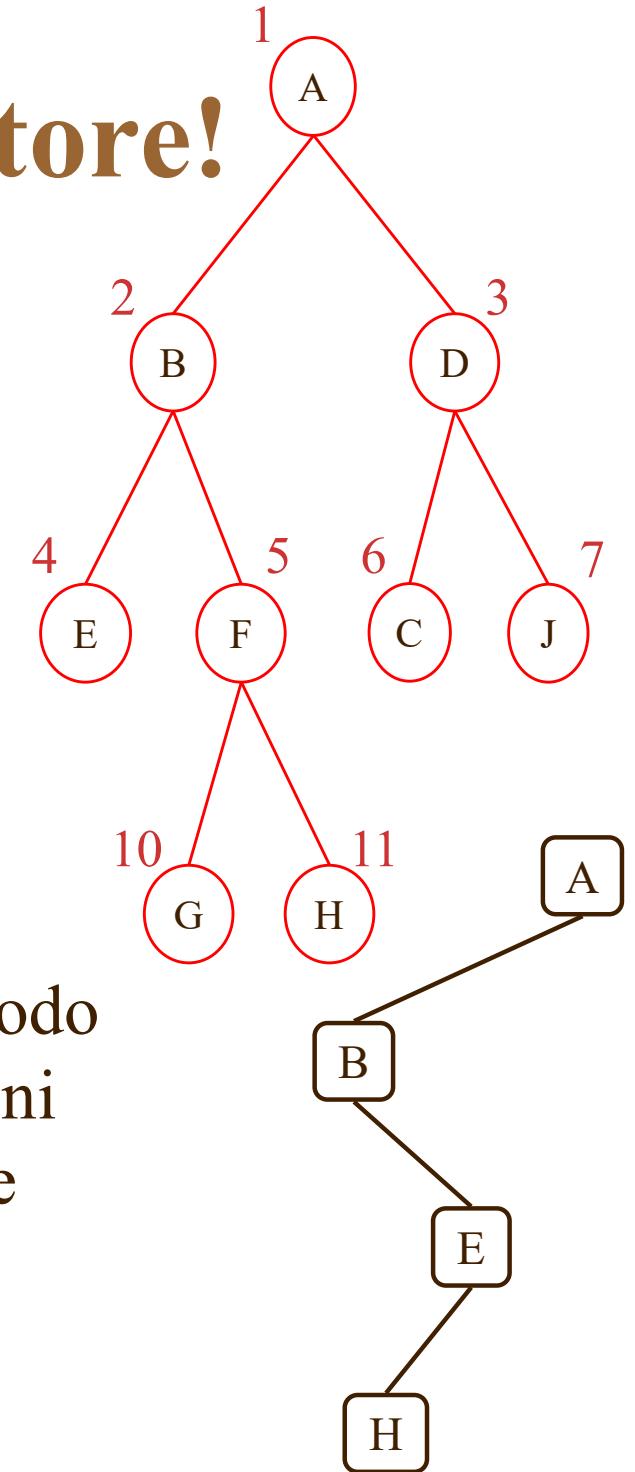
Albero binario in un vettore!

- Si può progettare una
`class ArrayListBinaryTree<T>`
`implements BinaryTree<T>`
i cui metodi abbiano **le stesse prestazioni**
temporali di quelli visti per l'albero binario
realizzato mediante struttura concatenata
- **Il problema è l'occupazione di memoria!**
- Un albero di dimensione n (cioè contenente n
nodi/dati) viene memorizzato in un esemplare di
LinkedBinaryTree usando uno spazio $\Theta(n)$
- Cosa possiamo dire per la sua memorizzazione in
un esemplare di **ArrayListBinaryTree** ?



Albero binario in un vettore!

- Che dimensione deve avere il vettore necessario per memorizzare un albero di dimensione n nel caso peggiore?
- Sappiamo che la funzione p “spreca” indici se l’albero “ha dei buchi”
 - Nell’esempio qui raffigurato, l’albero ha 9 nodi ma l’indice massimo vale 11
- Qual è il caso pessimo?
 - L’albero “pessimo” deve avere un solo nodo per ogni livello (massimo “spreco” ad ogni livello, dato che un livello non può essere vuoto...), cioè deve essere un albero “degenerato” in lista concatenata



Albero binario in un vettore!



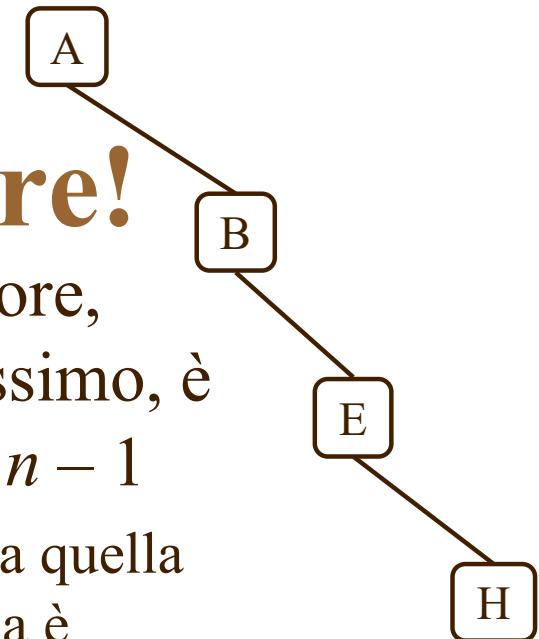
□ Qual è il caso pessimo?

- L'albero “pessimo” deve avere **un solo nodo per ogni livello** (massimo “spreco” ad ogni livello, dato che un livello non può essere vuoto...), cioè deve essere un albero “degenerato” in lista concatenata
- A ciascun livello dell’albero, si ha il massimo spreco di indici se l’unico nodo presente si trova nella **posizione più a destra** possibile
 - Perché questo avvenga, ogni nodo interno dell’albero deve avere il solo figlio destro
 - Procedendo dalla radice, che ha indice 1, ogni nodo ha indice uguale al doppio di quello del genitore + 1, quindi gli indici sono:

$$3, 7, 15, 31, 63, 127, \dots, (2^{h+1} - 1)$$

Albero binario in un vettore!

- Quindi, realizzando un albero binario con un vettore, l'occupazione di spazio è $O(2^h)$, che, nel caso pessimo, è ovviamente $O(2^n)$, dato che, come sappiamo, $h \leq n - 1$
- **C'è interesse per questa realizzazione**, che è equivalente a quella concatenata per quanto riguarda le prestazioni temporali, ma è **decisamente peggiore in merito all'occupazione di memoria?**
 - Sì, quando un'applicazione usa alberi binari che siano molto diversi dal caso pessimo! (Magari triangolari... o quasi...)
 - In alcune applicazioni, che **vedremo**, si usano alberi binari, chiamati **heap**, i cui nodi, con la funzione di numerazione p appena vista, hanno un indice massimo uguale a n per un albero di dimensione n , quindi $\Theta(n)$ [quando hanno questa proprietà si chiamano **completi**]
 - Ricordiamo che l'occupazione "specifica" (cioè per singolo dato) di una struttura concatenata è decisamente maggiore di quella di un array (**un nodo è un oggetto e occupa molta più memoria di una cella di array** contenente soltanto un riferimento, cioè 32 bit...)



Albero binario in un vettore!

- Come detto, per realizzare un albero binario è sufficiente un array contenente soltanto i dati (e non riferimenti a nodi), memorizzando il riferimento **null** in una cella avente indice corrispondente a una posizione non presente nell'albero (un "buco")
- Per fare questo, però, è necessario ridefinire l'interfaccia **BinaryTree** (e la sua super-interfaccia **Tree**) in modo che scambino con l'utente numeri interi (cioè indici) anziché riferimenti di tipo **Position**; inoltre, i dati non potranno avere il valore **null**
- Per ovviare a entrambi questi "inconvenienti" si può comunque definire un nodo semplificato (ad esempio, **ArrayBTNode**) che implementi **Position** e che contenga (invece dei quattro riferimenti **parent**, **left**, **right** e **element**) soltanto un indice (nell'array) e un riferimento (**element**) al dato corrispondente (che, a questo punto, può essere **null**)
 - Pensare ai problemi di prestazioni che sorgono nel metodo **remove...**
- Conseguentemente, l'array che contiene l'albero sarà un array di riferimenti a questi "nodi con indice", che occupano comunque meno spazio in memoria dei "nodi concatenati" (circa la metà di spazio per ogni nodo)

Lezione 22

Code Prioritarie

ADT che fanno confronti...

- Tutti i tipi di dati astratti che abbiamo visto finora gestiscono dati **senza fare confronti** tra loro
 - Pile, code, code doppie
 - Liste (con indice, con rango, con posizione astratta)
 - Alberi, alberi binari
- Gestiscono **relazioni posizionali** (lineari o gerarchiche), cioè **l'informazione rappresentata dalla struttura è costituita dalla posizione relativa dei dati e non dai loro valori**
 - Naturalmente "dall'esterno" della struttura possiamo fare confronti, ma questi non vengono fatti tramite primitive del tipo di dato astratto
- In molte altre applicazioni siamo, invece, interessati alla gestione di **relazioni d'ordine**

Relazione d'ordine

- All'interno di un insieme S si possono definire relazioni (o criteri) di ordinamento (o di confronto)
- Una **relazione d'ordine** definita nell'insieme S è una funzione $f: S \times S \rightarrow B \equiv \{\text{vero, falso}\}$ avente le proprietà:
 - Riflessiva: $\forall a \in S, a \leq a$
 - Transitiva: $\forall a, b, c \in S, a \leq b, b \leq c \Rightarrow a \leq c$
 - Antisimmetrica: $\forall a, b \in S, a \leq b, b \leq a \Rightarrow a = b$
 - La notazione $a \leq b$, con $a \in S$ e $b \in S$, significa che $f(a, b) = \text{vero}$
 - \leq è un simbolo come un altro, potremmo scrivere $a \# b$ oppure $a @ b$), usiamo \leq per analogia con la relazione d'ordine "naturale" esistente tra i numeri reali

La relazione “minore di” in campo reale (indicata con $<$) **non** è una relazione d'ordine, non essendo riflessiva: si dice che è una **relazione d'ordine stretto**

Relazione d'ordine totale

□ Una **relazione d'ordine** si dice **totale** per l'insieme S se vale anche questa ulteriore proprietà

- Totalità: $\forall a, b \in S, a \leq b$ oppure $b \leq a$ (oppure entrambe)
 - Cioè qualunque coppia di elementi dell'insieme dev'essere confrontabile
- Una relazione d'ordine non totale si dice anche **parziale**
- **Esempio:** Consideriamo l'insieme S costituito da tutti i sottoinsiemi dell'insieme dei numeri naturali (detto "l'insieme delle parti" di S)
 $S \equiv \{\emptyset, \{1\}, \{2\}, \{3\}, \dots, \{1, 2\}, \{1, 3\}, \dots \{2, 3\}, \dots \{1, 2, 3\}, \dots\}$
La relazione di "inclusione larga" tra insiemi (\subseteq) è una relazione d'ordine parziale (**e non totale**) per S .

Infatti, la proprietà di totalità non vale, perché, ad esempio:

non è vero che $\{1\} \subseteq \{2\}$ e non è vero nemmeno che $\{2\} \subseteq \{1\}$

Le proprietà della relazione d'ordine parziale sono, invece, valide (perché tale relazione è riflessiva, transitiva e antisimmetrica)

Relazione d'ordine totale

- Dato un insieme S totalmente ordinato
(secondo una determinata relazione d'ordine totale f)
 - Data una qualunque coppia di elementi dell'insieme S , è possibile determinare quale sia l'elemento minore (e quale sia l'elemento maggiore) secondo f
 - Gli elementi dell'insieme S si dicono **confrontabili** mediante f
 - In qualunque sottoinsieme finito e non vuoto dell'insieme S esiste l'elemento **minimo** e l'elemento **massimo** (secondo f)
 - **Gli elementi di S** possono essere disposti in una relazione **lineare** di tipo precedente/successivo (o minore/maggiore) secondo f
(cioè **possono essere "ordinati" in base a f**)

Relazione d'ordine totale

- Un insieme può essere totalmente ordinato secondo diverse relazioni d'ordine; es. stringhe secondo ordinamento lessicografico oppure secondo lunghezza
- Come si può definire una relazione d'ordine nell'insieme delle stringhe S tramite la loro lunghezza?
 - Es. (sbagliato):
$$\forall a, b \in S, f(a, b) = \text{true} \Leftrightarrow \text{length}(a) \leq \text{length}(b)$$
Questa non è una relazione d'ordine perché non gode della proprietà antisimmetrica, es. $a = \text{"PIPPO"}$, $b = \text{"PLUTO"}$, $f(a, b) = \text{true}$, $f(b, a) = \text{true}$, ma $a \neq b$
 - Es. (corretto):
$$\forall a, b \in S, f(a, b) = \text{true} \Leftrightarrow \text{length}(a) < \text{length}(b) \text{ oppure } \text{length}(a) = \text{length}(b) \text{ e } a \text{ "non segue" } b \text{ (cioè } a \text{ precede } b \text{ oppure è uguale a } b\text{) secondo l'ordinamento lessicografico}$$

Relazione d'ordine totale

- Dato un **insieme S totalmente ordinato**
(secondo una determinata relazione d'ordine totale f)
 - **Ripetiamo:** Data una qualunque coppia di elementi dell'insieme S , è possibile determinare quale sia l'elemento minore (e quale sia l'elemento maggiore) secondo f
 - In questo corso **supponiamo** (salvo esplicita indicazione contraria) **che, in una relazione d'ordine, ciascun confronto si possa fare in un tempo $\Theta(1)$ in funzione della dimensione dell'insieme S** , cioè in un tempo **che non dipende dalla dimensione dell'insieme S** (ma che può dipendere dalla dimensione dei singoli dati, come, ad esempio, nell'ordinamento lessicografico di stringhe)
- Si possono definire criteri di ordinamento “strani” per i quali il tempo necessario per un confronto non è $\Theta(1)$ rispetto alla dimensione dell'insieme, ma "di solito" non è così

Relazione d'ordine totale

- Si possono definire criteri di ordinamento “strani” per i quali il tempo necessario per un confronto non è $\Theta(1)$ **rispetto alla dimensione dell’insieme**
- Esempio
 - Sia dato un insieme di stringhe, S
 - $\forall s \in S$, sia $N(s) = |\{p \in S \mid p.length() < s.length()\}|$
 - Definiamo la relazione d’ordine totale f su S in modo che $\forall x \in S, \forall y \in S, f(x, y) = \text{true} \Leftrightarrow N(x) \leq N(y)$
 - In questo caso, ciascun confronto richiede un tempo $\Theta(|S|)$ per calcolare $N(x)$ e $N(y)$
 - In realtà il tempo richiesto è addirittura $\Theta(m|S|)$, dove m è la lunghezza media delle stringhe appartenenti a S , ma m può spesso essere considerata una costante

ADT coda prioritaria

- Una **coda prioritaria** (*priority queue*, PQ) è un contenitore che memorizza **elementi associati a un livello di priorità** o precedenza
 - Si possono inserire nuovi elementi nel contenitore solo se ne viene specificato il livello di priorità
 - Si può **estrarre** o **ispezionare** soltanto l'elemento avente la priorità **massima** (o **minima**) tra quelli presenti nel contenitore
 - Si parla, rispettivamente, di maxPQ e minPQ
 - Se non viene specificato, per PQ si intende una **minPQ**
 - I livelli di priorità **devono**, quindi, appartenere a un **insieme totalmente ordinato** e si dice che la PQ "usa" tale relazione d'ordine
 - spesso sono numeri interi non negativi o positivi
 - Gli elementi, invece, **non** devono **necessariamente** (ma possono) appartenere a un insieme totalmente ordinato

ADT coda prioritaria

□ Esempio

- La gestione dei processi in esecuzione in un calcolatore, dove esistono processi a priorità più elevata di altri, che vanno serviti dalla CPU in tempi più rapidi (la lettura/scrittura nel *file system* o il controllo della temperatura della CPU sono, ad esempio, azioni prioritarie rispetto al controllo periodico di nuova posta in arrivo)

□ Caso “degenero” ma ammesso e non raro:

Gli elementi coincidono con la propria priorità

- Esempio: una coda prioritaria di numeri interi, ciascuno dei quali rappresenta la propria priorità

Coda prioritaria: coda e pila

- Una **coda** è una **minPQ** in cui la priorità di ciascun elemento è il suo istante di inserimento in coda
 - La priorità viene assegnata dal contenitore stesso
 - La primitiva di inserimento ha, quindi, un vincolo ulteriore: ogni elemento, subito dopo il suo inserimento nella coda, ha priorità massima (perché il tempo aumenta sempre...)
 - Come abbiamo visto, consente implementazioni particolarmente efficienti, con tutte le primitive $\Theta(1)$
- Una **pila** è una **maxPQ** in cui la priorità di ciascun elemento è il suo istante di inserimento in coda
 - La priorità viene assegnata dal contenitore stesso
 - La primitiva di inserimento ha, quindi, un vincolo ulteriore: ogni elemento, subito dopo il suo inserimento nella pila, ha priorità massima (perché il tempo aumenta sempre...)
 - Come abbiamo visto, consente implementazioni particolarmente efficienti, con tutte le primitive $\Theta(1)$

ADT coda prioritaria

- Una **coda prioritaria** è un contenitore che memorizza elementi associati a un livello di priorità o precedenza
 - Gli elementi solitamente si chiamano **valori** (*value*)
 - La priorità associata a un valore si chiama **chiave** (*key*)
 - Chiavi e valori sono entità generiche (in Java, oggetti)
 - **Possono esistere più valori aventi la medesima chiave**, i casi di “pareggio” nella ricerca del minimo/massimo vanno risolti in modo **coerente** tra i metodi di ispezione e rimozione
 - Uno stesso valore può comparire più volte con chiavi diverse
 - **Le chiavi** vengono assegnate ai valori da una specifica applicazione, generalmente **non sono proprietà intrinseche** dei valori
 - Infatti, uno stesso valore può avere chiavi diverse in code prioritarie diverse

ADT coda prioritaria: chiavi

- Una **coda prioritaria** è un contenitore che memorizza **valori associati a una chiave**
 - Si possono inserire elementi, indicandone la chiave
 - Si può estrarre o ispezionare soltanto l'elemento avente la chiave **minima** tra quelli presenti nel contenitore
 - Perché una coda prioritaria possa funzionare, deve essere definito l'elemento minimo in qualunque sottoinsieme dell'insieme delle chiavi (in particolare, tra le chiavi presenti nel contenitore)
 - **È sufficiente che l'insieme delle chiavi sia totalmente ordinato**
 - Per definire una coda prioritaria, occorre definire
 - Il tipo di valori ammessi
 - Il tipo di chiavi associate ai valori
 - Una relazione d'ordine **totale** tra le chiavi

Interfaccia Entry

- Una coda prioritaria memorizza valori associati a una chiave (quindi **non può esistere un valore privo di chiave**)
 - Inseriamo **copie chiave/valore**, che chiamiamo **entry** (parola che significa genericamente “elemento”)
 - (Vedremo che) C’è qualcosa di simile in **java.util**

```
public interface Entry<K,V>
{ public K getKey();
  public V getValue();
  public V setValue(V newVal);
  // non mettiamo setKey perché in genere
  // cambiare la chiave di una coppia
  // è un'operazione non ammessa
}
```

Classe SimpleEntry

- Definiamo una classe
SimpleEntry

```
public interface Entry<K,V>
{   public K getKey();
    public V getValue();
    public V setValue(V newVal);
}
```

```
public class SimpleEntry<K,V> implements Entry<K,V>
{   private K key;    // può essere null
    private V value; // può essere null
    public SimpleEntry(K k, V v) { key = k; value = v; }
    public K getKey() { return key; }
    public V getValue() { return value; }
    public V setValue(V newVal)
    {   V old = value;
        value = newVal;
        return old;
    }
}
```

Forse avremmo potuto definire solo la classe **Entry**, **senza interfaccia**, dato che non ci sono molti modi diversi per realizzare un comportamento così semplice...

ADT coda prioritaria

```
public interface PriorityQueue<K,V> extends Container
{ Entry<K,V> insert(K key, V value)
    throws InvalidKeyException;
Entry<K,V> min() // ispezione, oppure getMin
    throws EmptyPriorityQueueException;
Entry<K,V> removeMin()
    throws EmptyPriorityQueueException;
} // size() restituisce il numero di Entry presenti
```

- Lanciando **InvalidKeyException**, **insert** rifiuta chiavi che non appartengono all'insieme totalmente ordinato delle chiavi definito per la PQ
 - Di solito rifiuta **null**
 - Altrimenti bisogna definire come si confronta **null** con le altre chiavi
 - La verifica da fare dipende dal tipo di chiavi e dal criterio di ordinamento totale utilizzato dalla PQ
 - Es. il tipo **K** potrebbe essere **String**, ma potrebbero essere ritenute valide soltanto le stringhe composte da sole lettere o cifre

ADT coda prioritaria

```
public interface PriorityQueue<K,V> extends Container
{ Entry<K,V> insert(K key, V value)
    throws InvalidKeyException;
Entry<K,V> min() // ispezione, oppure getMin
    throws EmptyPriorityQueueException;
Entry<K,V> removeMin()
    throws EmptyPriorityQueueException;
} // size() restituisce il numero di Entry presenti
```

- Nelle realizzazioni concrete di coda prioritaria (che implementeranno l'interfaccia **PriorityQueue**) ci dovrà essere un meccanismo per comunicare al contenitore quale sia la relazione d'ordine totale da utilizzare
- Solitamente questo avviene nel costruttore, quindi non lo menzioniamo nell'interfaccia
 - Vedremo diverse possibili soluzioni

Relazioni d'ordine in Java

ADT coda prioritaria: confronti

- Per definire una coda prioritaria, occorre definire
 - Una relazione d'ordine **totale** tra le chiavi
- Ci sono fondamentalmente due approcci
 - Usare **un ordinamento intrinseco** delle chiavi
 - Es. Interfaccia `java.lang.Comparable<T>`
 - **Non si può usare sempre**
 - Usare **un ordinamento esterno** alle chiavi
 - Es. Interfaccia `java.util.Comparator<T>`
 - **Approccio di applicabilità generale**

Interfaccia Comparable

- Se le chiavi sono esemplari di un tipo di dato che implementa l'interfaccia `java.lang.Comparable<T>`, la coda prioritaria può effettuare confronti tra chiavi invocando il metodo `int compareTo(T obj)`

- Esempio nella libreria standard

```
class String implements Comparable<String>
```

- Se **s1** e **s2** sono due stringhe, allora
s1.compareTo(s2) restituisce

- Zero se **s1.equals(s2)** vale **true**
 - Un valore negativo (non necessariamente **-1 !!**) se **s1** precede **s2** nell'ordinamento lessicografico
 - Un valore positivo (non necessariamente **+1 !!**) altrimenti

La semantica
di **compareTo**
deve essere
sempre questa

Interfaccia `java.lang.Comparable`

```
public interface Comparable<T>
{   int compareTo(T t);
}
```

□ Regole per implementare `Comparable`

- Il metodo deve rispettare la definizione di relazione d'ordine, quindi deve essere **riflessivo, transitivo e antisimmetrico, come l'operatore \leq**
- La funzione (booleana) che realizza la relazione d'ordine è
 - `a.compareTo(b) <= 0`

```
public interface Comparable<T>
{ int compareTo(T t); }
```

Interfaccia Comparable

CENNI

□ Regole per implementare Comparable

- È “fortemente raccomandato” che il metodo **compareTo** sia **coerente con equals**, eventualmente ridefinendo quest’ultimo
 - **a.compareTo(b) == 0 ⇔ a.equals(b) == true**
 - ma **a.compareTo(null)** deve lanciare **NullPointerException** (anche se, invece, **a.equals(null) == false**)
- Di solito (ma non necessariamente) esemplari di una classe sono confrontabili solo con altri esemplari della stessa classe (o di una classe derivata), quindi
 - **class XYZ implements Comparable<XYZ>**
 - Se si vogliono confrontare soltanto alcuni esemplari della classe, per consentire l’uso di un sottoinsieme di chiavi, si possono rifiutare gli altri lanciando **ClassCastException**

La sintassi per l'espressione di **vincoli di tipo** prevede l'uso di **extends** anche quando forse verrebbe da scrivere **implements...**

Interfaccia Comparable | CENNI

- Possiamo quindi realizzare una coda prioritaria con questa firma

```
class PQ1<K extends Comparable<K>,V>
    implements PriorityQueue<K,V>
{ . . . } // con oggetti di tipo K si può
           // così invocare compareTo
```

- I tipi generici si possono, infatti, “vincolare”
 - Mentre **V** può essere un tipo qualsiasi ,
K deve essere **un tipo che implementa Comparable**

Interfaccia Comparable

- C'è un problema: usando la proprietà **Comparable** delle chiavi, non riusciamo a usare **un criterio di ordinamento diverso da quello previsto dal progettista della classe**
 - Se le chiavi sono stringhe, l'unico ordine di priorità realizzabile è quello indotto dal criterio di ordinamento lessicografico
 - Invece, potremmo voler progettare una coda prioritaria che gestisce stringhe **in altro modo**: ad esempio, in base alla lunghezza... Con questo approccio non è possibile, se non modificando il codice della coda prioritaria
 - Inoltre, **se le chiavi sono esemplari di una classe che non implementa Comparable**, come facciamo?
 - **Abbiamo bisogno di un approccio più generale**

Interfaccia Comparator

- Dato un insieme di chiavi, vogliamo poter imporre **dall'esterno** un criterio di ordinamento
- Invece di lasciare che le chiavi “si confrontino da sole”, definiamo **un comparatore**
 - Un oggetto che abbia il compito di confrontare due chiavi, dotato di un unico metodo avente **la stessa semantica di compareTo**

```
package java.util;  
public interface Comparator<T>  
{  int compare(T obj1, T obj2);  
}
```

Osservare che si tratta di un’interfaccia a genericità **semplice**, non doppia: si confrontano oggetti dello stesso tipo (eventualmente derivato)

Interfaccia Comparator

- Per realizzare una coda prioritaria con stringhe come chiavi che usi un criterio di ordinamento diverso dall'ordinamento lessicografico possiamo usare un comparatore, definito come in questo esempio

```
class ByLengthStringComp
    implements java.util.Comparator<String>
{ public int compare(String s1, String s2)
{ if (s1 == null || s2 == null)
    // così è richiesto dalla documentazione
    // dell'interfaccia java.util.Comparator
    throw new ClassCastException();
    if (s1.length() == s2.length())
        // stessa lunghezza, usa ordine lessicografico
        return s1.compareTo(s2);
    return s1.length() - s2.length();
}
// spesso non ha variabili di esemplare
}
```

Lezione 23

**Svolgimento della
prima parte
dell'Esercizio 29
(nella cartella degli esercizi)**

Ancora
comparatori

```

import java.util.Comparator;
public class PQWithComparator<K,V>
    implements PriorityQueue<K,V>
{ private Comparator<K> comp;
  ... // altre variabili di esemplare, dipende da
       // come viene realizzata la coda prioritaria
  public PQWithComparator(Comparator<K> c)
  { comp = c; ...}
  /* i metodi della coda prioritaria confrontano
     chiavi usando comp.compare(..., ...) */
  ...
}

```

Usa chiavi di qualsiasi tipo, purché venga fornito un comparatore adeguato

```

PriorityQueue<String, Integer> pq
    = new PQWithComparator<String, Integer>(
        new ByLengthStringComp()
    );
pq.insert("AAA", 7);
pq.insert("BB", 8); // diventa la chiave "minima"!
System.out.println(pq.removeMin().getValue()); // 8
System.out.println(pq.removeMin().getValue()); // 7
System.out.println(pq.size()); // 0

```

```

class StandardStringComp
    implements java.util.Comparator<String>
{
    public int compare(String s1, String s2)
    {   if (s1 == null || s2 == null)
        throw new ClassCastException();
        return s1.compareTo(s2);
    }
}

```

- La stessa classe **PQWithComparator**, usata con il comparatore **StandardStringComp** anziché **ByLengthStringComp**, usa l'ordinamento lessicografico

```

PriorityQueue<String, Integer> pq
    = new PQWithComparator<String, Integer>(
        new StandardStringComp()
    );
pq.insert("AAA", 7); // rimane la chiave "minima"!
pq.insert("BB", 8);
System.out.println(pq.removeMin().getValue()); // 7
System.out.println(pq.removeMin().getValue()); // 8
System.out.println(pq.size()); // 0

```

```

import java.util.Comparator;
public class StringPQ<V> // le chiavi non sono
                           // generiche, ma String
    implements PriorityQueue<String, V>
{ private class StandardStringComp
    implements Comparator<String>
    { ... } // vista in precedenza
    private Comparator<String> comp;
    public StringPQ() {comp=new StandardStringComp(); }
    public StringPQ(Comparator<String> c) {comp=c; }
    ...
}

```

Questa è una PQ specifica, le chiavi
non sono più generiche, sono stringhe!

```

PriorityQueue<String, Integer> pq           // osservare...
    = new StringPQ<Integer>();
pq.insert("AAA", 7);
pq.insert("BBB", 8);
System.out.println(pq.removeMin().getValue()); // 7
System.out.println(pq.removeMin().getValue()); // 8
System.out.println(pq.size());                // 0

```

Coda Prioritaria
realizzata
con una lista

PQ realizzata con lista

- La realizzazione di coda prioritaria con una lista di *entry* prevede due diverse strategie
 - Lista non ordinata
 - Lista ordinata

Richiamo: lista concatenata

- In una lista semplicemente (o doppiamente) concatenata
 - L'inserimento di un nodo nella posizione successiva a una posizione p , quando si conosce già il riferimento p , è $\Theta(1)$, perché con p si ha accesso anche al nodo successivo (oltre che al nuovo nodo che si sta inserendo) e il numero di collegamenti da modificare è $\Theta(1)$, cioè è un'operazione **LOCALE**
 - Analogamente, la rimozione del nodo successivo al nodo p , quando si conosce già il riferimento p , è $\Theta(1)$

PQ con lista non ordinata

- Ad esempio con una lista concatenata (o un array)
- Per realizzare **insert**, usiamo **addLast** o **addFirst**: $\Theta(1)$
 - In un array, $\Theta(1)$ in media, con analisi ammortizzata
- Per realizzare **min**, dobbiamo ispezionare in sequenza **TUTTE** le coppie presenti (ad esempio con un iteratore), per trovare quella avente chiave minima (o una qualsiasi di quelle aventi chiave minima) : $\Theta(n)$, anche in un array
- Per realizzare **removeMin**, dobbiamo trovare la posizione della coppia che verrebbe restituita da **min**, per poi rimuoverla: la prima parte è $\Theta(n)$, la seconda è $\Theta(1)$ (in un array, scambiamo l'elemento da rimuovere con l'ultimo, $\Theta(1)$ in media se si ridimensiona "in piccolo"), quindi complessivamente la rimozione è $\Theta(n)$
- Osserviamo che **min** e **removeMin** sono $\Theta(n)$ **sempre**, non solo nel caso peggiore o medio! Per "colpa" di **min**

PQ con lista ordinata

- Ad esempio con una lista concatenata, con le coppie ordinate in modo che ogni coppia abbia chiave non inferiore a quella della coppia precedente (o un array)
 - Quindi, `min` e `removeMin` si realizzano, rispettivamente, con `getFirst` e `removeFirst`: sono $\Theta(1)$
[array: si ordina al contrario o si usa un array in modalità circolare]
- Per realizzare `insert` dobbiamo trovare la posizione giusta in cui inserire la nuova coppia, scandendo la lista con un iteratore:
 $O(n)$ per trovare la posizione [$O(\log n)$ in un array, per bisezione]
 $\Theta(1)$ per poi inserire [$O(n)$ in un array, con eventuale ridimensionamento $\Theta(n)$]
complessivamente [anche in un array] $O(n)$ nel caso medio,
 $\Theta(n)$ nel caso peggiore
- Se inserimenti e rimozioni hanno la stessa frequenza, questa soluzione è quindi (un po') migliore della precedente, perché una delle due operazioni non è $\Theta(n)$ nel caso medio
 - Ad esempio, se ogni volta inserisco una chiave che diventa quella minima... sono sempre nel caso migliore!

Riassunto: Prestazioni Priority Queue

	Listo non ordinata	Listo ordinata
insert	$\Theta(1)$	$O(n)$
min	$\Theta(n)$	$\Theta(1)$
removeMin	$\Theta(n)$	$\Theta(1)$

PQ con lista ordinata o no?

□ Con **lista ordinata** abbiamo **rimozioni veloci**,
con **lista non ordinata** abbiamo **inserimenti veloci**

□ Caso particolare (**non raro**)

- Prima si fanno (quasi) tutti gli inserimenti, poi (quasi) tutte le rimozioni, fino a vuotare la coda prioritaria, avente dimensione massima n
- In questo caso conviene
 - Fare (circa) n (veloci) inserimenti con lista gestita senza ordinamento
 - Le eventuali (poche) rimozioni saranno lente
 - Su comando dell’utente, ordinare la lista: $\Theta(n \log n)$
 - In seguito, fare le (circa) n (veloci) rimozioni sulla lista ordinata
 - Gli eventuali (pochi) inserimenti saranno lenti
 - Complessivamente: $\Theta(n \log n)$
 - Con gestione immutabile (sempre ordinata o sempre non ordinata) si ha complessivamente $O(n^2)$, nel caso peggiore $\Theta(n^2)$ [**molto peggio**]

Lezione 24

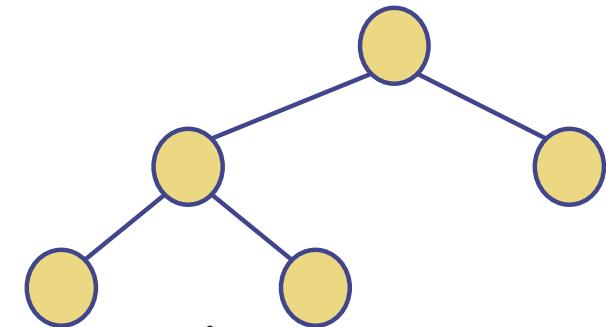
Heap

PQ con lista ordinata oppure no?

	Liste non ordinata	Liste ordinata
<code>insert</code>	$\Theta(1)$	$O(n)$
<code>min</code>	$\Theta(n)$	$\Theta(1)$
<code>removeMin</code>	$\Theta(n)$	$\Theta(1)$

- Con lista ordinata abbiamo rimozioni veloci,
con lista non ordinata abbiamo inserimenti veloci
- Possiamo cercare di “**bilanciare**” le prestazioni delle due azioni?
 - Usando un (particolare) albero binario, detto *heap* (“mucchio”), realizzeremo una coda prioritaria avente inserimenti e rimozioni $O(\log n)$, con ispezioni $\Theta(1)$
- **Ci servono alcune definizioni**

Essere a sinistra di...



□ **Definizione:** Dati due nodi v e w distinti e posti allo stesso livello l di un albero binario T , diciamo che v è a sinistra di w se v precede w nell'attraversamento in ordine simmetrico di T (altrimenti, v è a destra di w)

- Riflette la proprietà geometrica "essere a sinistra di" nella rappresentazione grafica standard degli alberi binari
 - Non è definita tra nodi che abbiano profondità diverse (anche se si potrebbe definire usando sempre l'ordine di attraversamento simmetrico)
- Definizione che (si dimostra essere) equivalente:
 v è a sinistra di $w \neq v$ (con $\text{depth}(v) = \text{depth}(w)$) se e solo se esiste un nodo $u \in T$, antenato comune di v e w , per cui
 - v appartiene al sottoalbero sinistro di u
 - e
 - w appartiene al sottoalbero destro di u

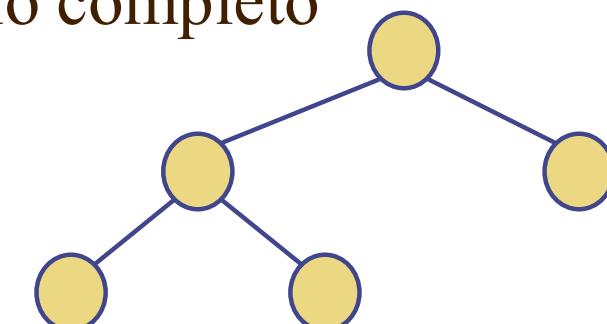
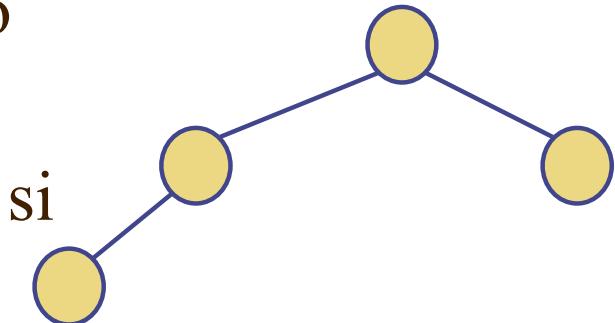
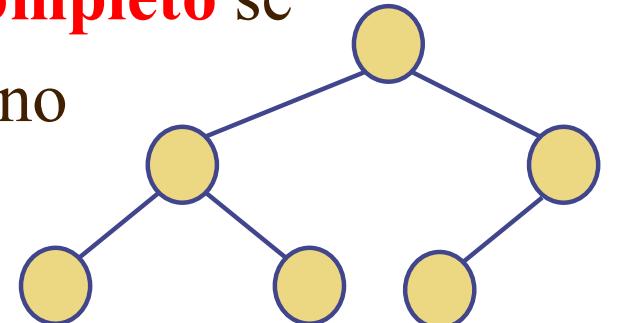
Analogamente si definisce "essere a destra di..."

Albero binario completo

Casi degeneri:

alberi vuoti o con la sola radice sono completi

- Un albero **binario** T di altezza $h > 0$ si dice **completo** se
 - Tutti i livelli di profondità minore di h hanno il massimo numero di nodi possibile (quindi il livello i ha 2^i nodi, $\forall i < h$)
 - Al livello $h - 1$ tutti i nodi interni si trovano a sinistra di tutti gli **eventuali** nodi esterni
 - Al livello $h - 1$ soltanto il nodo interno che si trova più a destra (tra i nodi interni) può avere un solo figlio, che deve essere il suo figlio sinistro
- Dalla definizione discende che un albero binario completo
 - è triangolare, oppure
 - è triangolare fino al livello $h - 1$, poi ha un ultimo livello “compatto a sinistra”



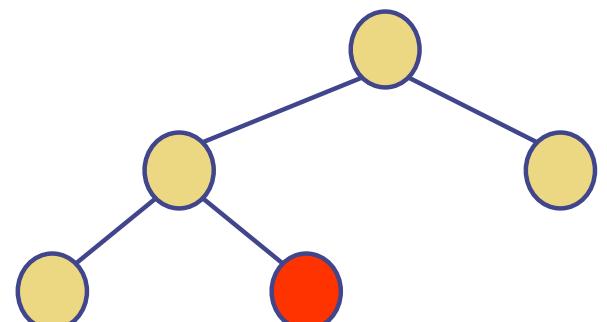
Albero binario completo

□ In un albero binario **completo** esiste un nodo che ha una posizione caratteristica, così definita

- L'**ultimo nodo** di un albero binario completo è la foglia più a destra tra quelle più profonde
 - Non è sufficiente dire "la foglia più a destra", come si vede in figura

□ Detto in altro modo

- L'**ultimo nodo** di un albero binario completo di altezza h è quel nodo di livello h che ha alla propria sinistra **tutti** gli altri nodi di livello h



Albero binario completo

□ In un albero binario completo T di dimensione $n > 0$

- $h = \lfloor \log_2 n \rfloor$ (quindi $h \in \Theta(\log n)$)

□ **Dimostrazione.** Se T ha altezza h ed è completo, allora

- La sua dimensione **minima** si ha quando è "quasi" triangolare, cioè ha un solo nodo nel livello h
 - Se non ha nodi nel livello h , vuol dire che non ha altezza h ☺

$$n_{\min} = 1 + 2 + 4 + \dots + 2^{h-1} + 1 = 2^h - 1 + 1 = 2^h$$

- La sua dimensione **massima** si ha quando è triangolare
 $n_{\max} = 1 + 2 + 4 + \dots + 2^{h-1} + 2^h = 2^{h+1} - 1$

□ Quindi: $2^h \leq n \leq 2^{h+1} - 1$

Albero binario completo

□ $2^h \leq n \leq 2^{h+1} - 1$

- $2^h \leq n \Rightarrow \log_2 2^h \leq \log_2 n \Rightarrow h \log_2 2 \leq \log_2 n$
 $\Rightarrow h \leq \log_2 n$
 - $n \leq 2^{h+1} - 1 \Rightarrow n + 1 \leq 2^{h+1} \Rightarrow \log_2 (n + 1) \leq \log_2 2^{h+1}$
 $\Rightarrow \log_2 (n + 1) \leq (h + 1) \log_2 2 \Rightarrow \log_2 (n + 1) \leq h + 1$
 $\Rightarrow \log_2 (n+1) - 1 \leq h$
- Quindi: $\log_2 (n+1) - 1 \leq h \leq \log_2 n$
- Quanti numeri interi appartengono a questo intervallo?
- Se questo intervallo contiene uno e un solo numero intero, allora $h = \lfloor \log_2 n \rfloor$, e anche $h = \lceil \log_2 (n+1) - 1 \rceil$
 - Dimostriamo...

Albero binario completo

□ Tesi: $\forall n > 0$, $\log_2 n$ e $\log_2 (n+1) - 1$ differiscono per meno di un'unità

$$\log_2 n - [\log_2 (n+1) - 1] < 1$$

$$\log_2 n - \log_2 (n+1) + 1 < 1$$

$$\log_2 n - \log_2 (n+1) < 0$$

$$\log_2 [n / (n+1)] < 0$$

$$n / (n+1) < 2^0$$

$$n / (n+1) < 1$$

$$n < n+1 \text{ vero } \forall n$$

Quindi, nell'intervallo $[\log_2 (n+1) - 1, \log_2 n]$ può esistere UN SOLO numero intero, altrimenti la differenza tra gli estremi dell'intervallo non sarebbe minore di uno

Albero binario completo

- Abbiamo dimostrato che tra $\log_2(n+1) - 1$ e $\log_2 n$ non possono esistere DUE numeri interi diversi
- Rimarrebbe da dimostrare, però, che tra $\log_2(n+1) - 1$ e $\log_2 n$ esiste certamente UN numero intero, che è l'altezza dell'albero

- Non serve dimostrarlo:
un albero binario
ha certamente un'altezza!!
Che abbiamo dimostrato essere
all'interno del suddetto
intervallo... il quale, quindi,
contiene certamente un numero
intero! Che è $h = \lfloor \log_2 n \rfloor = \lceil \log_2(n+1) - 1 \rceil$

Risultato molto importante:
in un albero binario completo
di dimensione $n > 0$
l'altezza è $h = \lfloor \log_2 n \rfloor$

Algoritmi che abbiano prestazioni proporzionali all'altezza sono logaritmici in funzione del numero di nodi, se operano su un albero binario completo

ADT: Albero binario completo

- Integriamo l'interfaccia **BinaryTree** con metodi che la rendano utilizzabile, **con il vincolo di completezza**
- Ricordiamo che l'interfaccia **BinaryTree** aggiunge ai metodi dell'interfaccia **Tree** soltanto i metodi **left**, **right**, **hasLeft** e **hasRight**
- Un albero che implementi soltanto l'interfaccia **BinaryTree** non è utilizzabile, perché non ci sono metodi per aggiungervi nodi
 - I metodi **addRoot**, **addLeft**, **addRight**, **sibling** e **remove** erano stati inseriti solamente in un esempio di implementazione dell'interfaccia **BinaryTree**, ma NON ne fanno parte e non ci servono in questo caso, perché, in generale, non preservano la completezza dell'albero

ADT: Albero binario completo

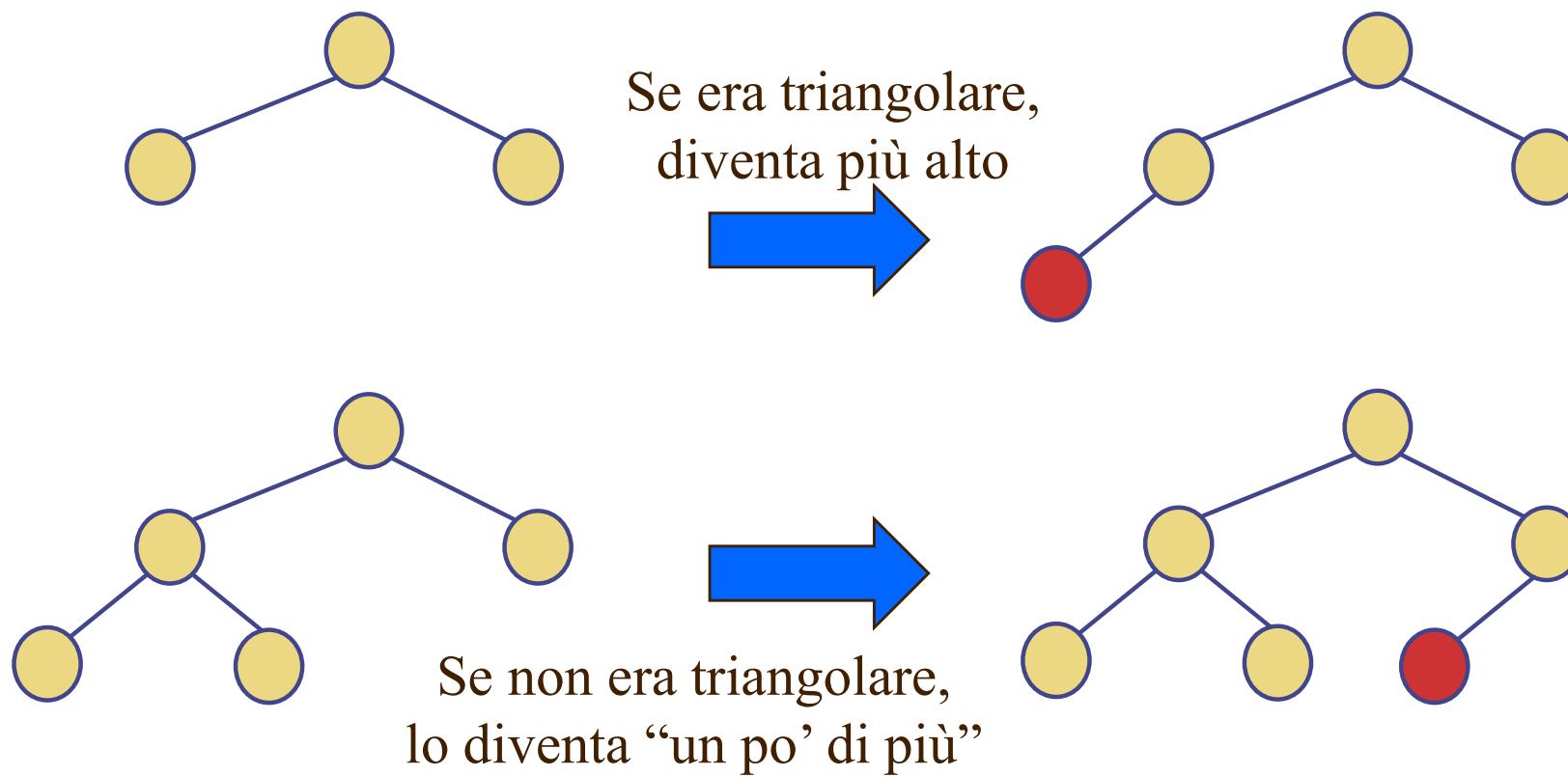
- Integriamo l'interfaccia **BinaryTree** con metodi che la rendano utilizzabile, **con il vincolo di completezza**

```
public interface CompleteBinaryTree<T>
    extends BinaryTree<T>
{
    void add(T element);
    T remove() throws EmptyTreeException;
}
```

- Il metodo **add** inserisce **element** in un nuovo nodo che diventa il nuovo **l'ultimo nodo** dell'albero (è l'unico nodo che si può inserire mantenendo la completezza dell'albero); non fallisce mai; può agire su un albero vuoto
- Il metodo **remove** elimina **l'ultimo nodo** dell'albero (l'unico che si può eliminare mantenendo la completezza) e restituisce l'elemento che vi era contenuto; fallisce se l'albero è vuoto
- Entrambe le operazioni, per come sono definite, preservano la completezza dell'albero: non essendoci altri metodi di modifica, l'albero è sempre completo (perché l'albero vuoto è completo)

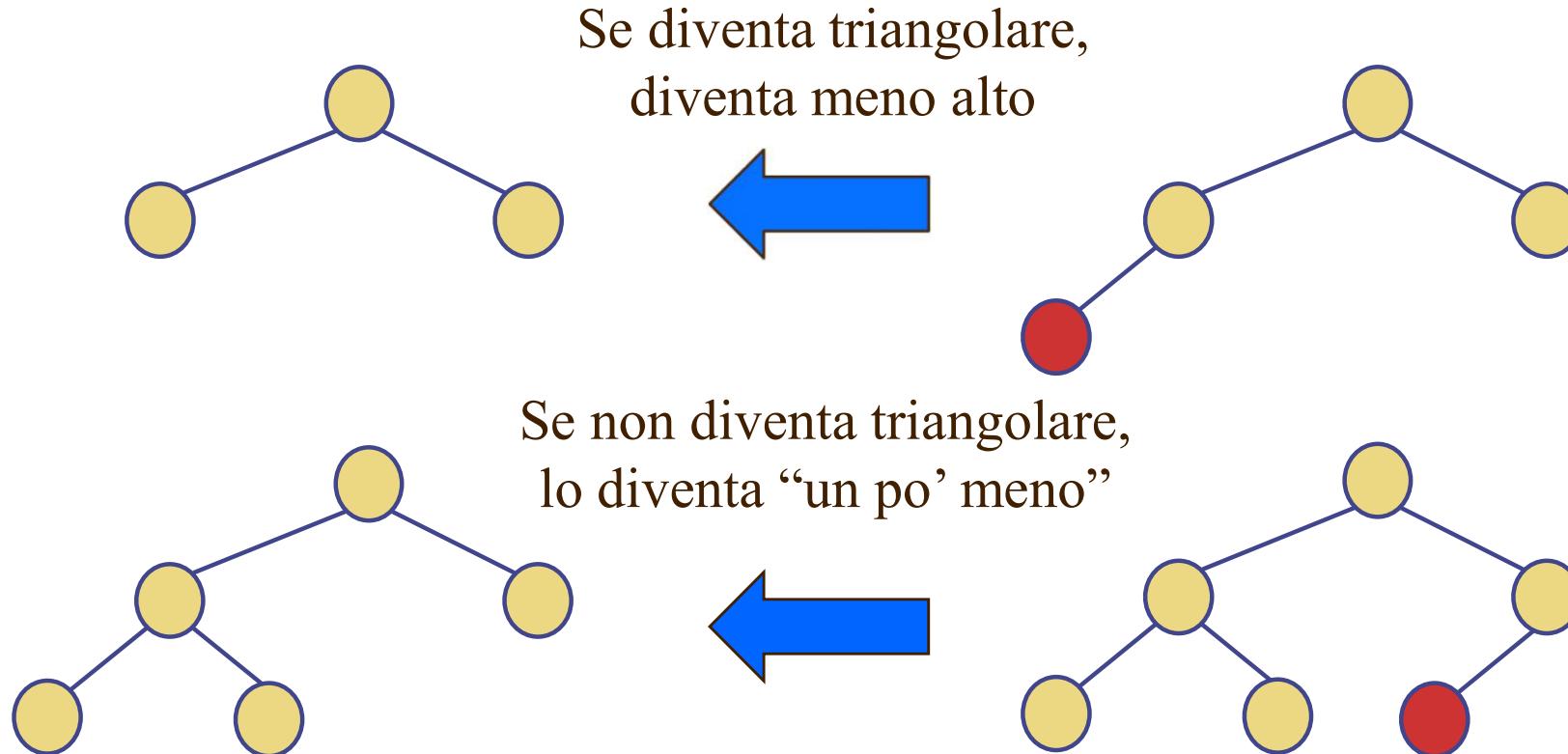
add in CompleteBinaryTree

- Il metodo **add** inserisce **element** in un nuovo nodo che diventa l'ultimo nodo dell'albero; non fallisce mai
- Ci sono due casi: l'albero può **essere triangolare** oppure no



remove in CompleteBinaryTree

- Il metodo **remove** elimina l'ultimo nodo dell'albero e restituisce l'elemento che vi era contenuto; fallisce se l'albero è vuoto
- Ci sono due casi: l'albero può **diventare triangolare** oppure no



Realizzazione di CompleteBinaryTree

- Realizziamo concretamente l'interfaccia mediante una struttura concatenata?
 - Meglio di no, è uno di quei casi in cui la memorizzazione dell'albero binario **in un vettore** ha prestazioni ottime per quanto riguarda l'occupazione di memoria, classe **ArrayCompleteBT**
- Ricordiamo la definizione della funzione di numerazione $p(v)$ che assegna un indice a ogni nodo v di un albero binario T di dimensione n
 - Se v è la radice di T , allora $p(v) = 1$
 - Altrimenti, se v è il figlio sinistro di u , $p(v) = 2 p(u)$
 - Altrimenti (v è il figlio destro di u e) $p(v) = 2 p(u) + 1$
- **Se (e solo se) T è completo, i suoi n nodi hanno indici consecutivi nell'intervallo $[1, n]$**
 - **L'ultimo nodo ha sempre indice n**

Classe **ArrayListCompleteBT**

- Nella realizzazione di un albero binario completo T di dimensione n mediante un array, **i nodi di T hanno indici consecutivi nell'intervallo $[1, n]$ e l'ultimo nodo ha sempre indice n**
- Occupazione di memoria: $\Theta(n)$
 - Se si può prevedere la dimensione n dell'albero (caso abbastanza frequente nelle applicazioni, usando un parametro del costruttore), l'array ha dimensione $n + 1$, altrimenti, ridimensionando per un fattore moltiplicativo sia in crescita che in calo, ha dimensioni $\Theta(n)$
- Prestazioni temporali del metodo **add**
 - Se si può prevedere la dimensione massima dell'albero e, quindi, l'array non deve mai essere ridimensionato, il metodo **add** ha prestazioni **$\Theta(1)$** , altrimenti ha prestazioni **$\Theta(1)$ in media**, con analisi ammortizzata
- Prestazioni temporali del metodo **remove**
 - Se non si effettua il ridimensionamento "in calo", il metodo **remove** ha prestazioni **$\Theta(1)$** , altrimenti ha prestazioni **$\Theta(1)$ in media**, con analisi ammortizzata

Classe `LinkedCompleteBT`

- Naturalmente un albero binario completo T di dimensione n si può realizzare anche mediante una **struttura concatenata**
- Occupazione di memoria: come al solito, $\Theta(n)$
- Prestazioni temporali del metodo **add**
 - Si può dimostrare che sono **$O(\log n)$**
- Prestazioni temporali del metodo **remove**
 - Si può dimostrare che sono **$O(\log n)$**
- Il tempo serve a cercare l'ultimo nodo...
- **Se un algoritmo utilizza alberi binari completi, conviene usare una realizzazione mediante array**

È peggiore
dell'array!

Esercizio - Difficile

- Progettare un algoritmo che, dato un albero binario **completo** di dimensione n **realizzato con una struttura a nodi concatenati**, restituisca la posizione del suo **ultimo nodo** in un tempo $O(\log n)$
 - Si può così realizzare il metodo **remove** con prestazioni $O(\log n)$
- L'algoritmo può usare soltanto le primitive dell'interfaccia **BinaryTree** e non può modificare la struttura interna dei nodi, né memorizzarvi alcuna informazione
- **Suggerimento:** riflettere sulla **numerazione dei nodi** utilizzata dalla realizzazione di albero binario completo all'interno di un array, ma riflettere... in binario! E l'ultimo nodo ha indice $n\dots$
- Modificare l'algoritmo in modo che, con le stesse prestazioni e con gli stessi vincoli, trovi la posizione del **genitore di un "nuovo ultimo nodo"** (utile per poi realizzare il metodo **add** con prestazioni $O(\log n)$)

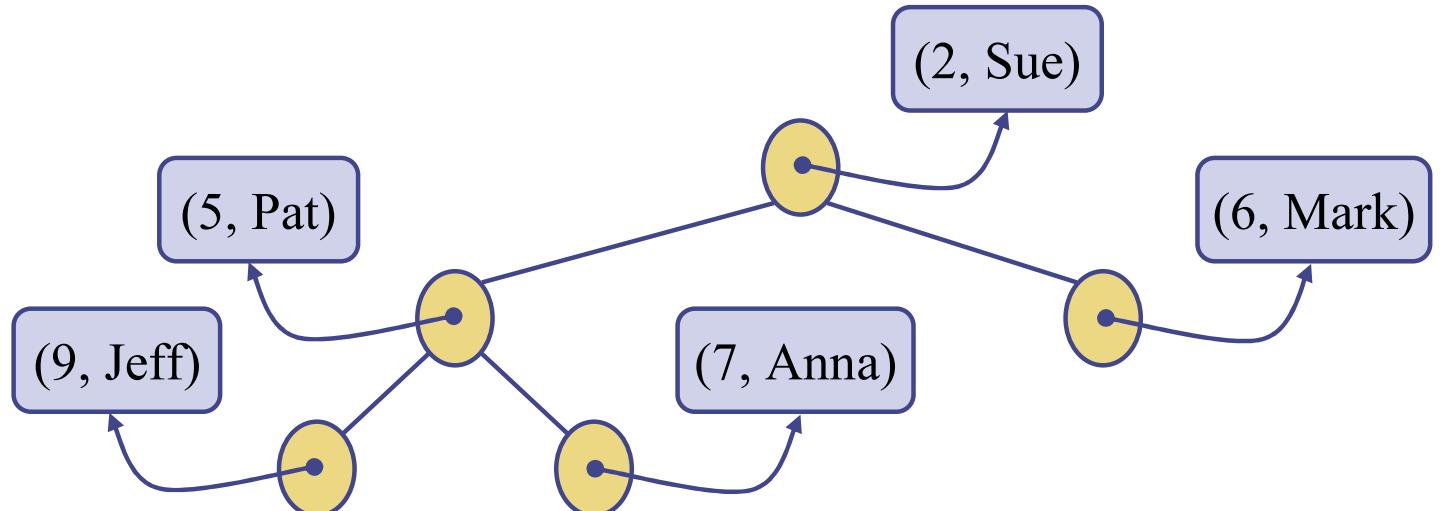
Heap

- Uno **heap** è un **albero binario completo** T che
 - Memorizza in ciascun nodo $v \in T$ una *entry* $e(v) = (k, val)$, composta da una chiave, $k = \text{key}(v) = e.\text{getKey}()$, appartenente a un insieme totalmente ordinabile e da un valore, $val = \text{value}(v) = e.\text{getValue}()$

- Soddisfa la seguente **ulteriore** proprietà:

Questa viene chiamata “proprietà di heap”, **heap property**

- $\forall v \in T, v \neq T.\text{root}() \Rightarrow \text{key}(\text{parent}(v)) \leq \text{key}(v)$



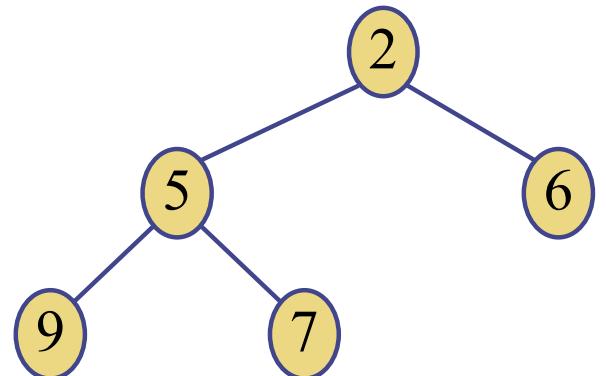
La completezza è, invece, una proprietà soltanto topologica, non dipende dai dati

$$\forall v \in T, v \neq T.\text{root}() \Rightarrow \text{key}(\text{parent}(v)) \leq \text{key}(v)$$

Proprietà di uno Heap

- Dalla definizione, si dimostra (agevolmente) che
 - Le chiavi che si incontrano lungo qualsiasi percorso minimo che vada dalla radice a una foglia sono in ordine **non decrescente** (e viceversa, "risalendo"...)
 - **Nella radice è presente la chiave minima** (o una delle chiavi minime): è la chiave “in cima al mucchio”
 - Invertendo la **“direzione” della relazione** presente nella proprietà di heap, si ottiene uno heap avente la chiave massima (anziché minima) in cima al mucchio
 - Si chiamano minHeap e maxHeap
(quando si dice semplicemente heap si intende minHeap)

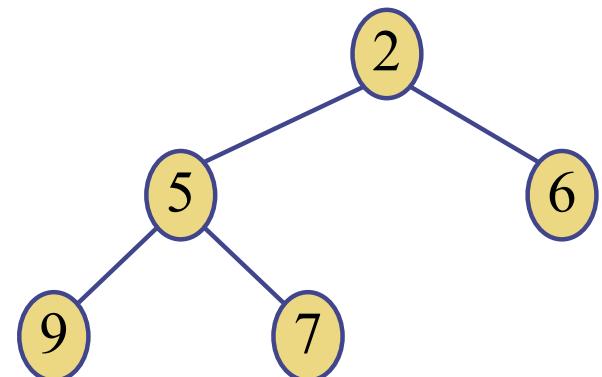
Attenzione:
non confondere lo heap con la coda prioritaria!!
Lo heap servirà a realizzare una coda prioritaria



Realizziamo una PQ con uno Heap

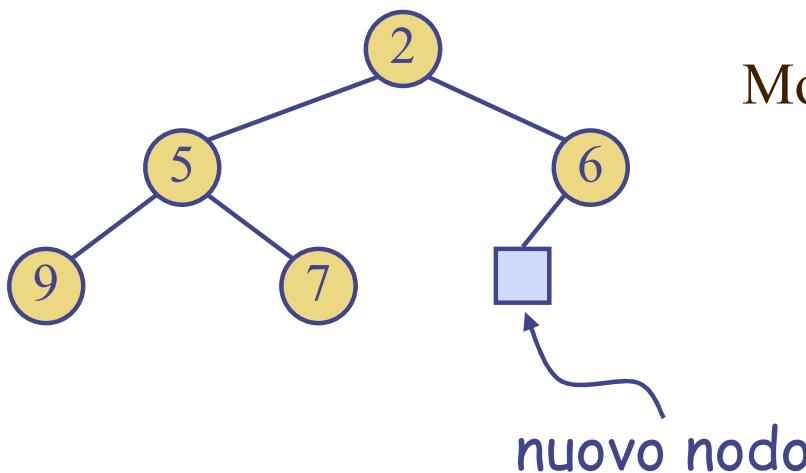
- Usiamo uno heap memorizzato in un vettore
 - Classe `ArrayHeapPQ`
- Il metodo **size** restituisce la dimensione (logica) del vettore **diminuita di uno**, in un tempo $\Theta(1)$
 - Il metodo **isEmpty** è, conseguentemente, $\Theta(1)$
- Il metodo **min** restituisce semplicemente il contenuto della radice (cioè della cella di indice 1), in un tempo $\Theta(1)$
- I metodi **insert** e **removeMin** sono più complessi, ma **dimostreremo** che sono entrambi $O(\log n)$ in generale e $\Theta(\log n)$ nel caso pessimo

È **MOLTO** migliore della PQ realizzata con una lista!!



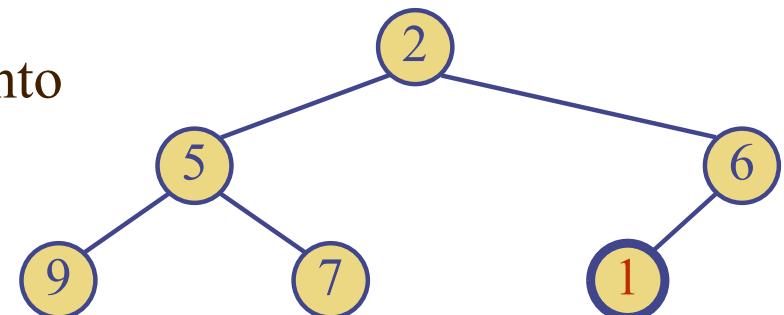
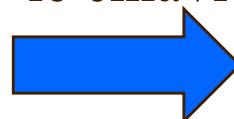
Metodo `insert` in `ArrayHeapPQ`

- Per prima cosa inseriamo nell'albero completo un nuovo nodo contenente la nuova *entry*
 - L'albero binario completo accetta di inserire un nuovo nodo **soltanto** come suo **ultimo nodo z** (operazione $\Theta(1)$)
 - Dopo questo inserimento, l'albero sarà certamente ancora completo, ma, in generale, non rispetterà più la **proprietà di heap**
 - Se la PQ era vuota, lo heap non necessita di aggiustamenti, perché ha un solo nodo
 - Altrimenti, il nodo inserito z è certamente diverso dalla radice



Mostriamo soltanto

le chiavi

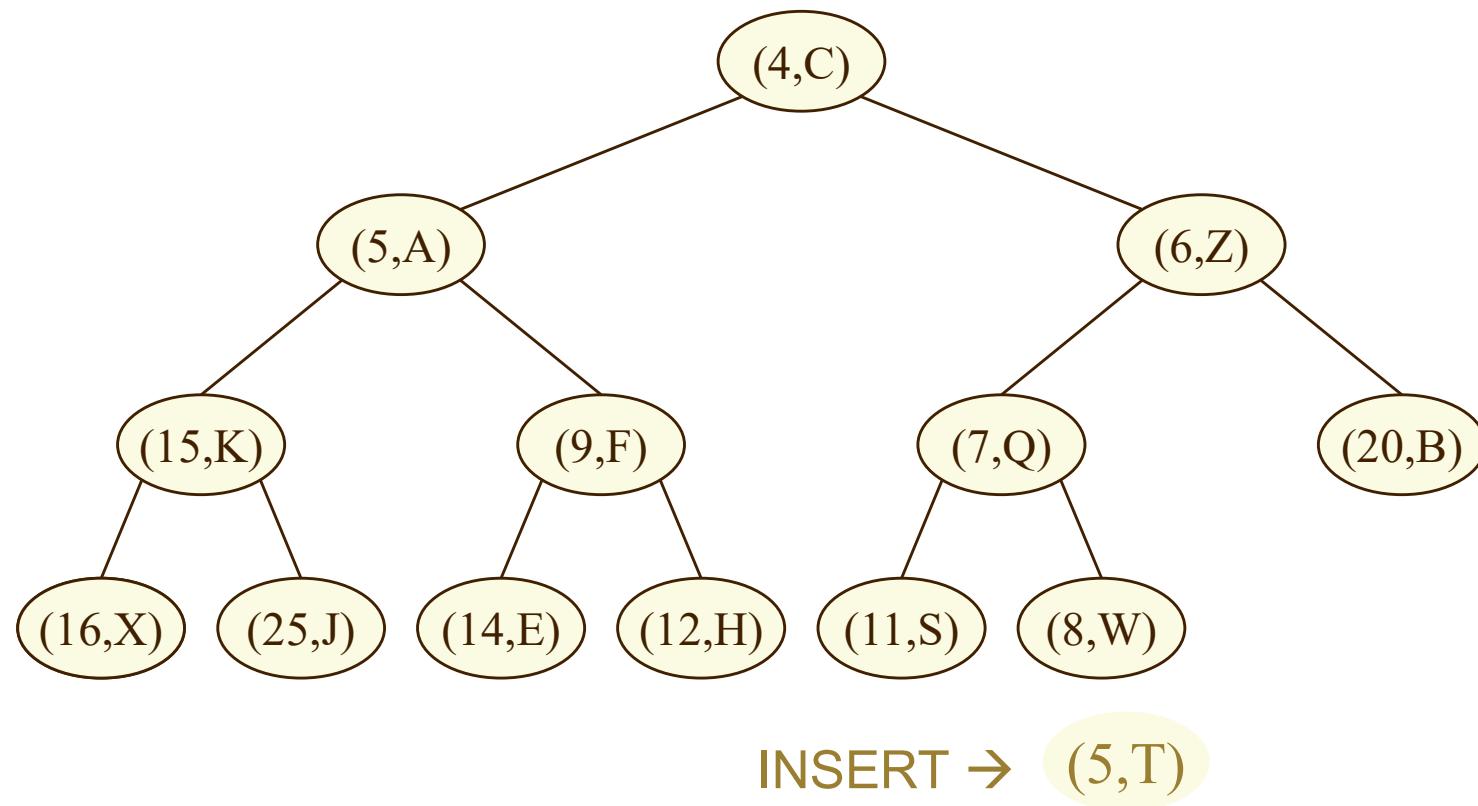


Metodo `insert` in `ArrayHeapPQ`

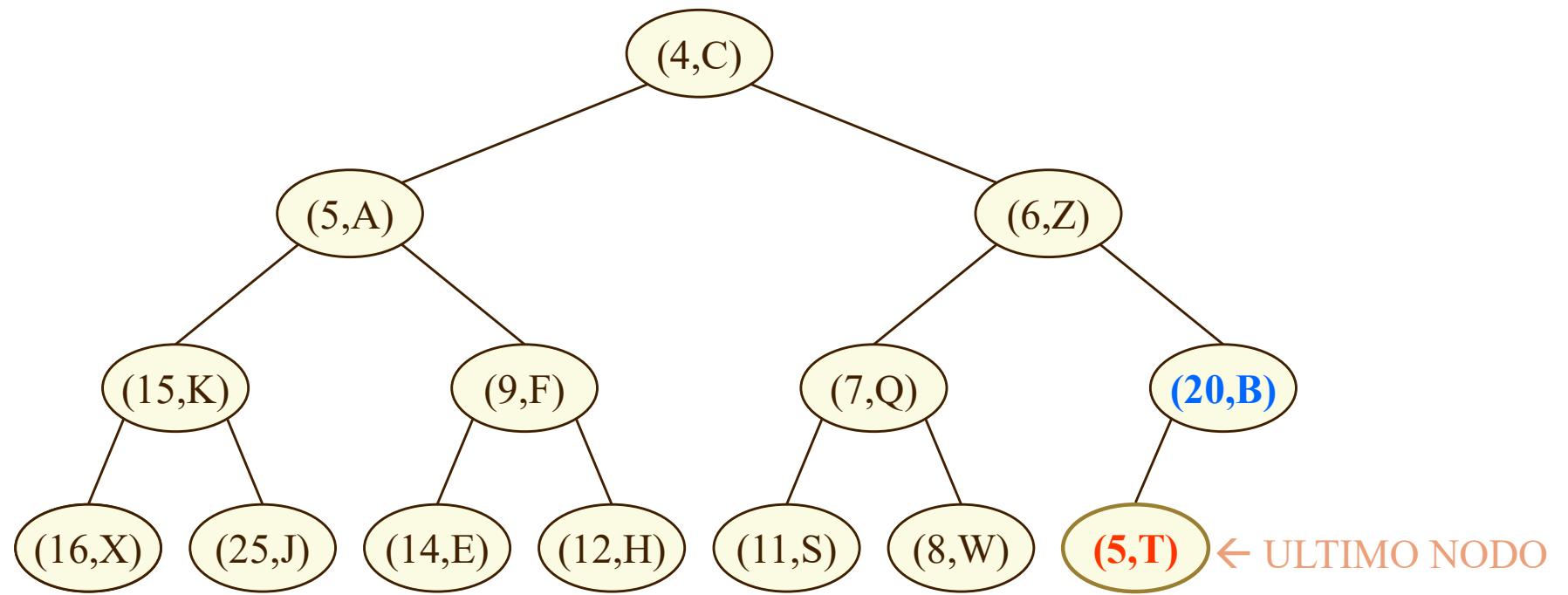
- Dopo l'inserimento così effettuato, dobbiamo “sistemare” l'albero T in modo che rispetti la proprietà di heap
 - Si potrebbe pensare di ristrutturare completamente il contenuto dell'albero, ma vogliamo stare molto attenti alle prestazioni...
 - **Si può fare in vari modi:** un modo ottimo (dal punto di vista delle prestazioni) consiste nel far “risalire” il nuovo nodo z lungo la sua catena di antenati, finché la proprietà di heap non risulta soddisfatta di nuovo in tutto l'albero
 - In realtà ciò che si fa risalire è il dato presente nel nuovo nodo
- Algoritmo **up-heap bubbling** relativo al nodo z , l'unico che non rispetta la proprietà di heap

```
while ( $z \neq T.\text{root}()$  &&  $\text{key}(z) < \text{key}(\text{parent}(z))$ )
    scambia  $e(z)$  con  $e(\text{parent}(z))$  // metodo replace di Tree
                                // si scambiano i dati, non i nodi!
     $z = \text{parent}(z)$  // riferimenti, nessuna modifica all'albero
```
- **Adesso è tutto a posto... anche se è meglio dimostrarlo** ☺

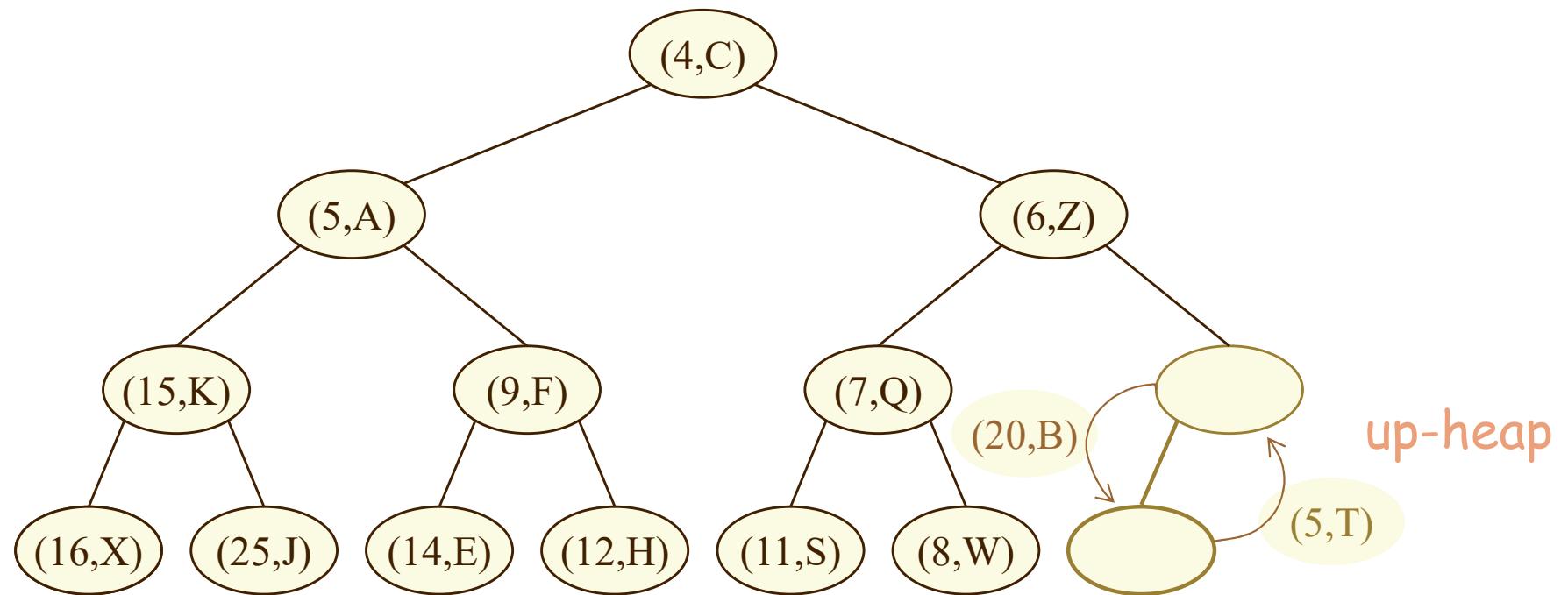
insert



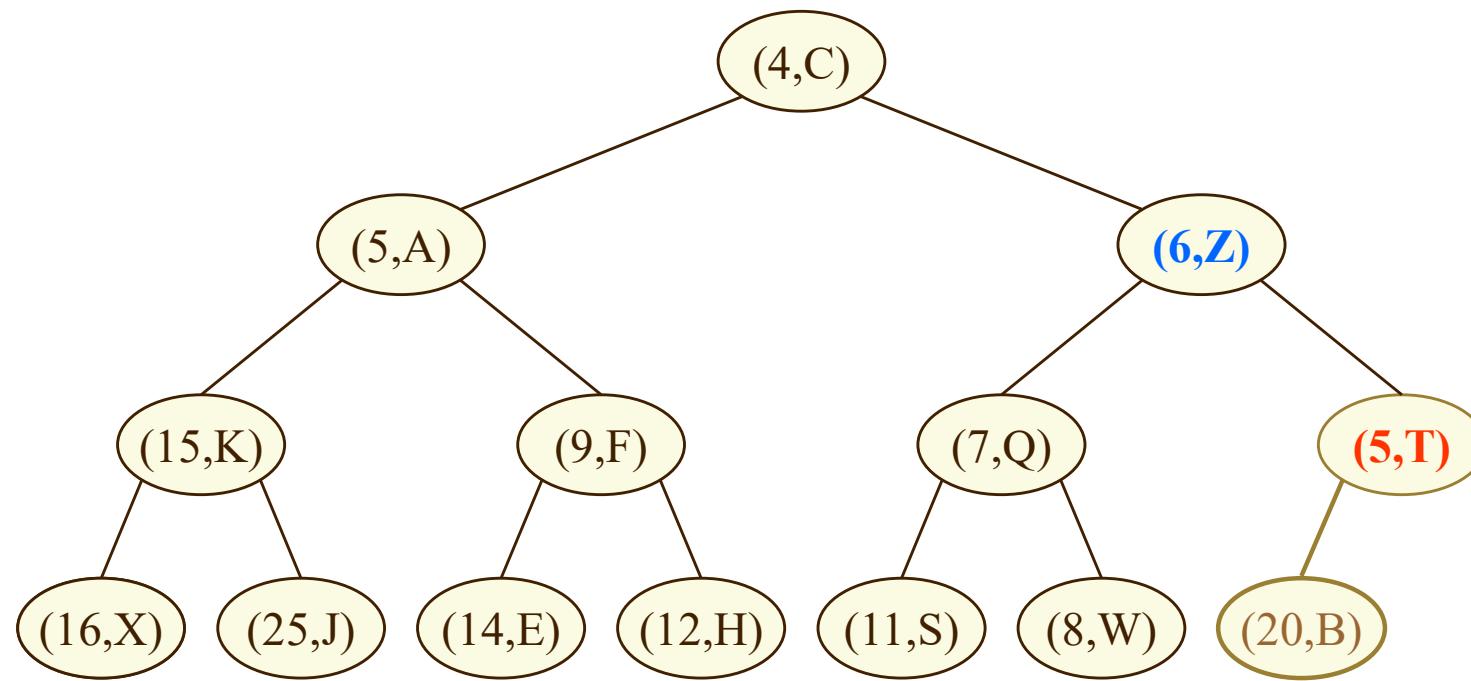
insert



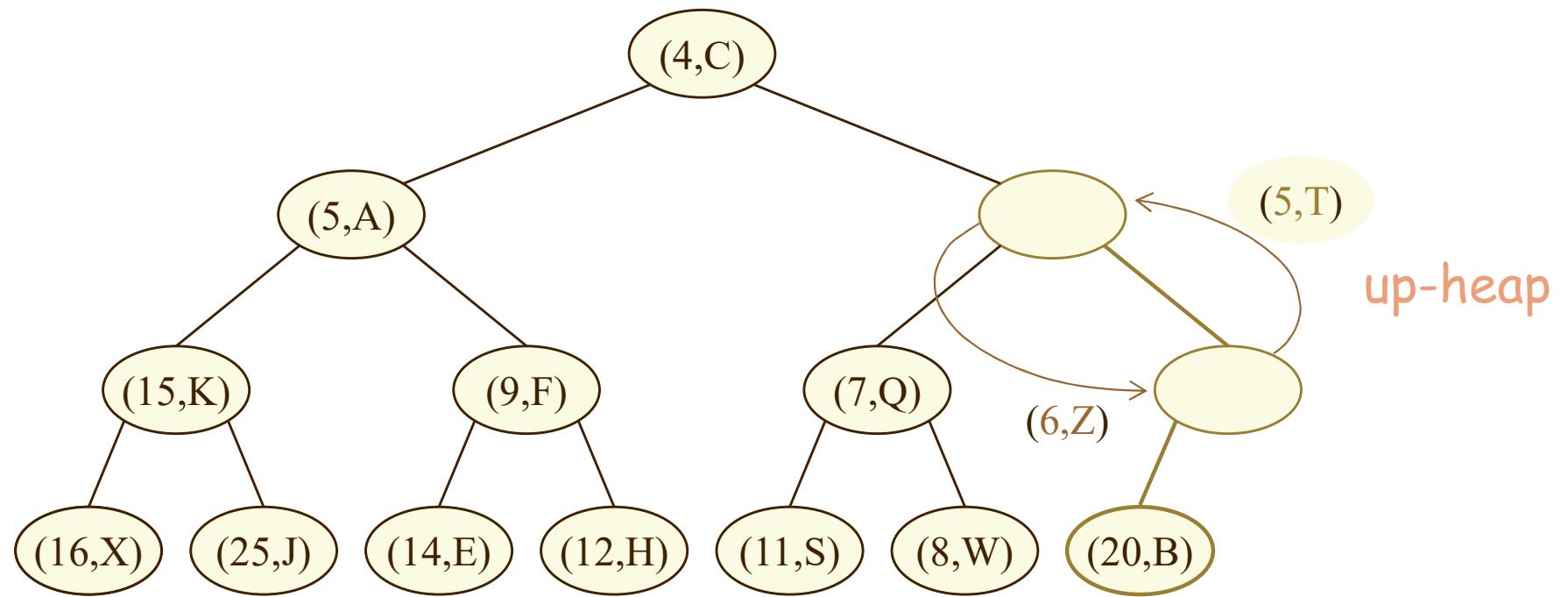
insert



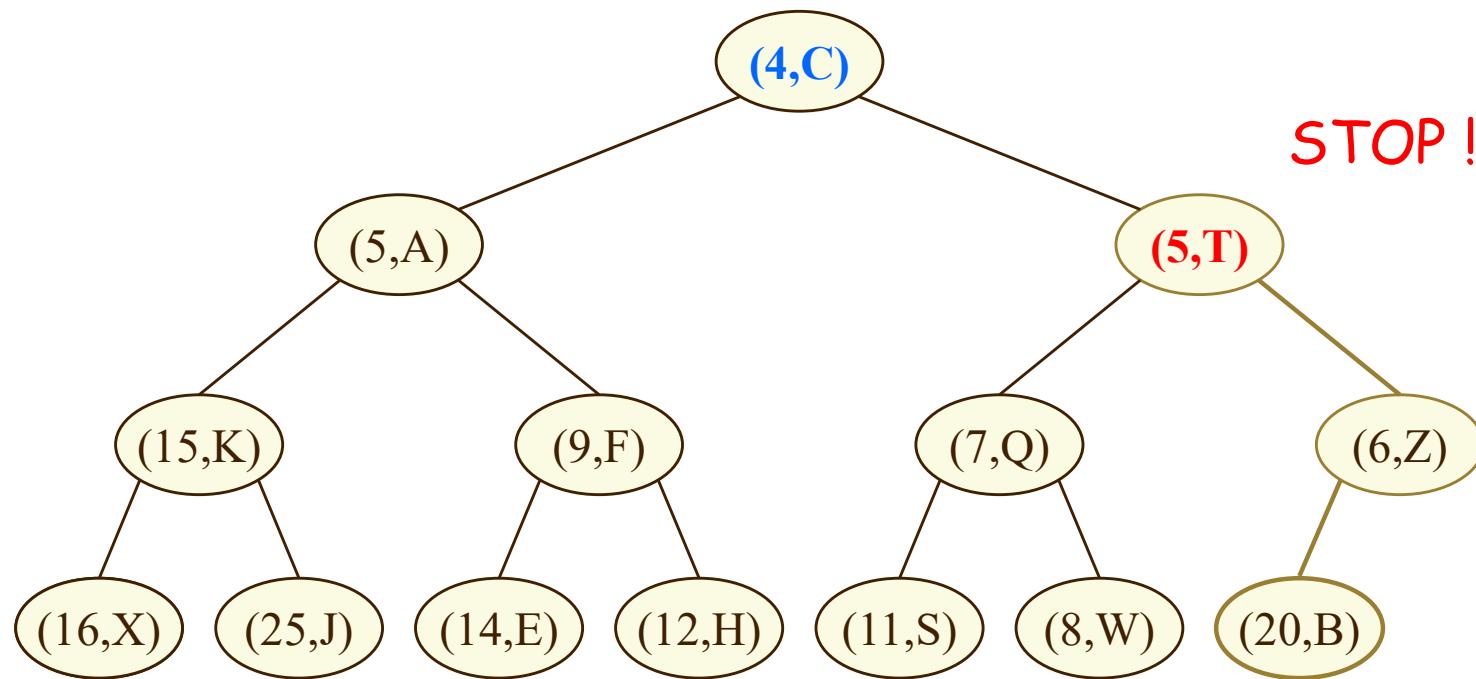
insert



insert



insert: finito



Metodo `insert` in `ArrayHeapPQ`

```
while ( $z \neq T.\text{root}()$  &&  $\text{key}(z) < \text{key}(\text{parent}(z))$ )
    scambia  $e(z)$  con  $e(\text{parent}(z))$  // metodo replace di Tree
     $z = \text{parent}(z)$  // riferimenti, nessuna modifica all'albero
```

- Se z è inizialmente la radice, ho inserito in una PQ vuota, ho finito
- Altrimenti, z non è la radice, quindi esiste $\text{parent}(z)$; se $\text{key}(z) \geq \text{key}(\text{parent}(z))$, la proprietà di heap è rispettata localmente, ma z non ha figli e nel resto dell'albero era già rispettata (una nuova foglia deve confrontarsi soltanto con il proprio genitore), quindi l'algoritmo correttamente termina
- Altrimenti, $\text{key}(z) < \text{key}(\text{parent}(z))$ e la nuova foglia è fuori posto
- Ricordiamo che **la proprietà di heap è LOCALE**: **ogni nodo deve confrontarsi soltanto con il proprio genitore**, non con antenati/descendenti/fratelli...

Metodo `insert` in `ArrayHeapPQ`

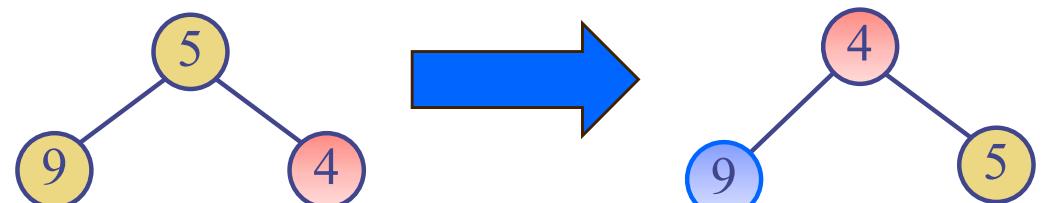
```
while (z ≠ T.root() && key(z) < key(parent(z)))
    scambia e(z) con e(parent(z)) // metodo replace di Tree
    z = parent(z) // riferimenti, nessuna modifica all'albero
```

- Se $\text{key}(z) < \text{key}(\text{parent}(z))$ il nodo z è fuori posto
 - Usando (due volte) il metodo `replace` dell'interfaccia `Tree`, **scambio il contenuto** del nodo fuori posto con quello del suo genitore
 - **Non c'è alcuna modifica della struttura**, ma è come se il nodo z si scambiasse di posto con il suo genitore: risale!
 - Il secondo enunciato del corpo del ciclo `while` fa in modo che **il contenuto** del prossimo nodo che verrà controllato sia... quello di prima, che è però risalito!
 - Cos'è successo localmente in seguito allo scambio?

Metodo `insert` in `ArrayHeapPQ`

```
while ( $z \neq T.\text{root}()$  &&  $\text{key}(z) < \text{key}(\text{parent}(z))$ )
    scambia  $e(z)$  con  $e(\text{parent}(z))$  // metodo replace di Tree
     $z = \text{parent}(z)$  // riferimenti, nessuna modifica all'albero
```

- Se $\text{key}(z) < \text{key}(\text{parent}(z))$, scambia $e(z)$ con $e(\text{parent}(z))$
 - **Cos'è successo localmente in seguito allo scambio?**
 - Il genitore (5), che è diventato figlio (e foglia), rispetto al suo nuovo genitore (4) è (certamente) a posto, per la condizione del ciclo while
 - L'ex ultimo nodo (4), che è diventato genitore, è (analogamente) a posto rispetto al suo vecchio genitore (5), che ora è diventato suo figlio
 - Ma cosa si può dire di (4) rispetto all'eventuale **altro figlio (9)** del vecchio genitore (5)? Ora (9) è diventato figlio di (4)...
 - Il nuovo genitore (4) di tale figlio (9) è minore del precedente genitore (5): dato che il precedente era a posto rispetto al figlio (9), questo (4) lo sarà a maggior ragione



Metodo `insert` in `ArrayHeapPQ`

```
while ( $z \neq T.\text{root}()$  &&  $\text{key}(z) < \text{key}(\text{parent}(z))$ )
    scambia  $e(z)$  con  $e(\text{parent}(z))$  // metodo replace di Tree
     $z = \text{parent}(z)$  // riferimenti, nessuna modifica all'albero
```

- Quindi l'eventuale fratello di un nodo fuori posto non è mai coinvolto: era a posto con il proprio genitore e, se avviene uno scambio, rimane a posto (ancor di più) rispetto al nuovo genitore
 - Analogamente, un nodo che “scende” perché scambiato è sempre a posto con il proprio nuovo genitore e con i propri eventuali nuovi figli, perché questi erano già suoi discendenti
- Il nodo che risale si ferma quando è a posto rispetto al proprio genitore, oppure è diventato la radice!
- Quindi, se l'albero ha altezza h , al massimo si fanno h scambi. Ma sappiamo che un albero binario completo ha altezza $h = \lfloor \log_2 n \rfloor$, quindi il metodo è $O(\log n)$
[$\Theta(\log n)$ nel caso peggiore]

Lezione 25

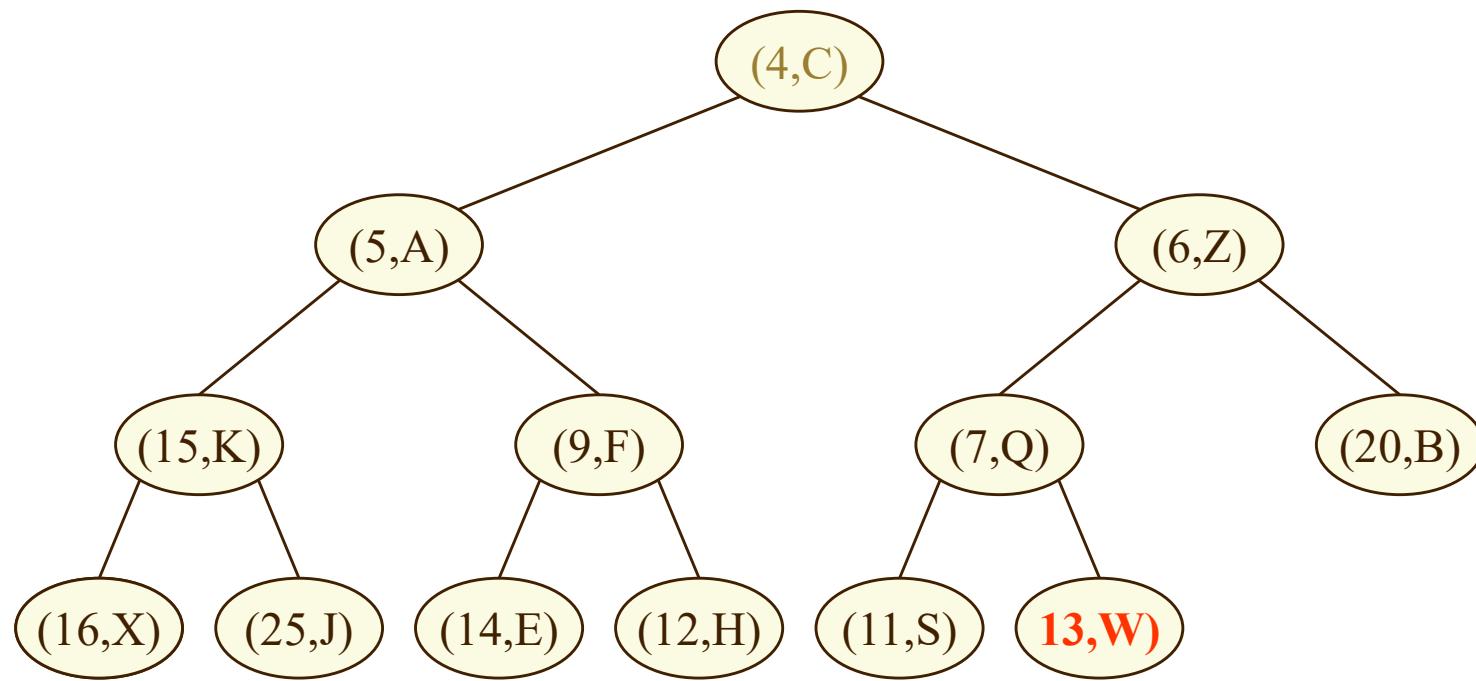
Metodo `removeMin` in `ArrayHeapPQ`

- Sappiamo che nella radice c'è la chiave minima (o una delle chiavi minime), ma non possiamo eliminare la radice dell'albero
 - L'albero binario completo accetta di rimuovere soltanto il suo **ultimo nodo z** (operazione $\Theta(1)$)
 - **Scambiamo il contenuto della radice con il contenuto dell'ultimo nodo, poi eliminiamo tale ultimo nodo**
 - A questo punto, l'albero contiene i dati corretti, ma nella radice, in generale, non sarà rispettata la proprietà di heap
 - Casi degeneri
 - Aveva due soli nodi, è rimasto con la sola radice, non necessita di aggiustamenti
 - Aveva un solo nodo, è rimasto vuoto, non necessita di aggiustamenti

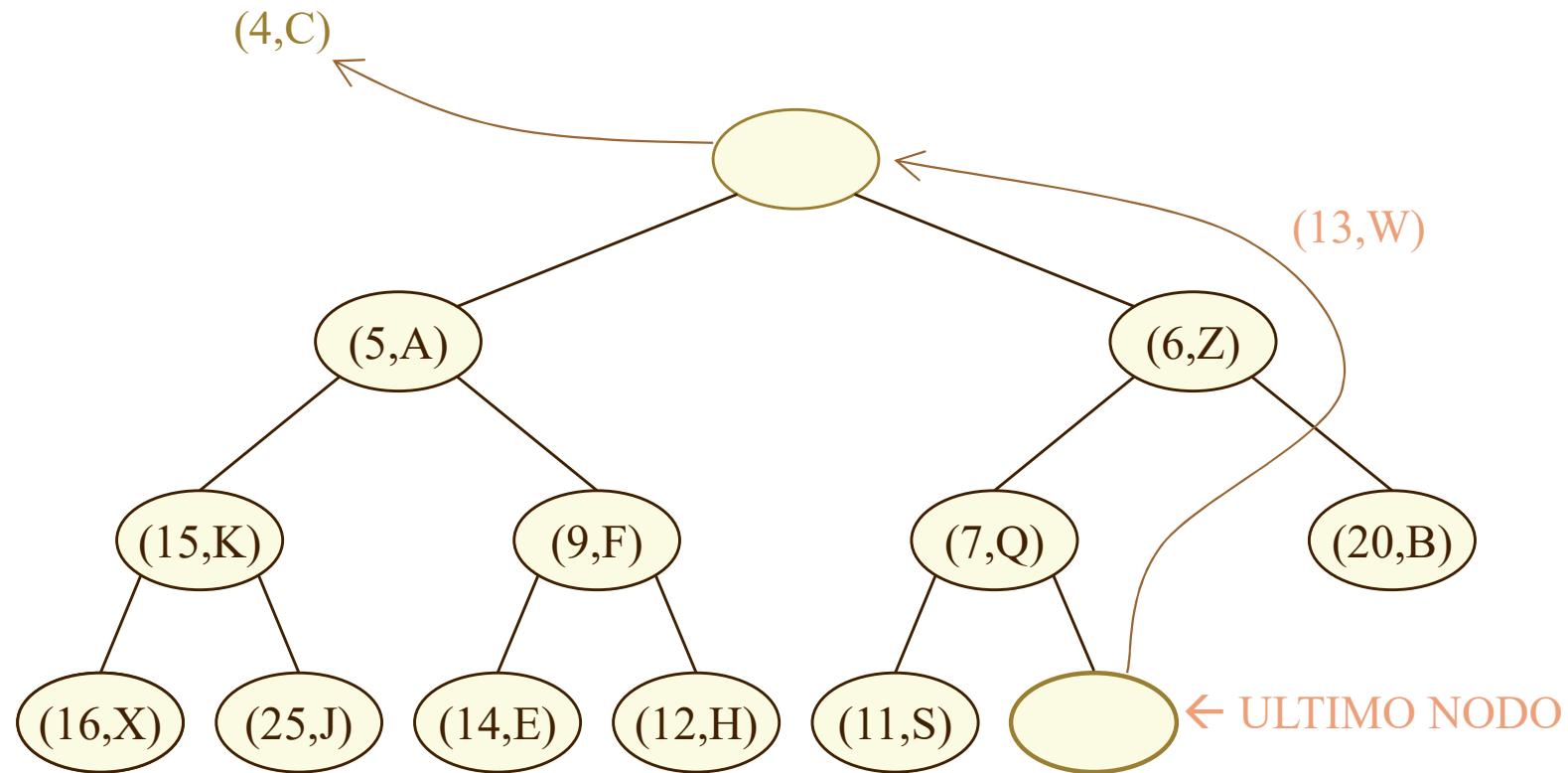
Metodo `removeMin` in `ArrayHeapPQ`

- Nella radice, in generale, non sarà rispettata la proprietà di heap
- Eseguiamo un algoritmo di **down-heap bubbling**
 1. Il primo nodo v da esaminare è la radice
 2. Se v non ha figli, ho finito
 3. Identifico **il figlio w di v avente chiave minima** (v può anche avere un solo figlio, che sarà il figlio sinistro, perché in un albero completo nessun nodo può avere il solo figlio destro)
 4. Se w e v non rispettano la proprietà di heap, li scambio (cioè ne scambio il contenuto), altrimenti ho finito
 5. Il nuovo nodo v da esaminare è quello che è appena sceso (se nessun nodo è sceso, avevo finito al passo 4)
 6. Ripeto dal punto 2

removeMin

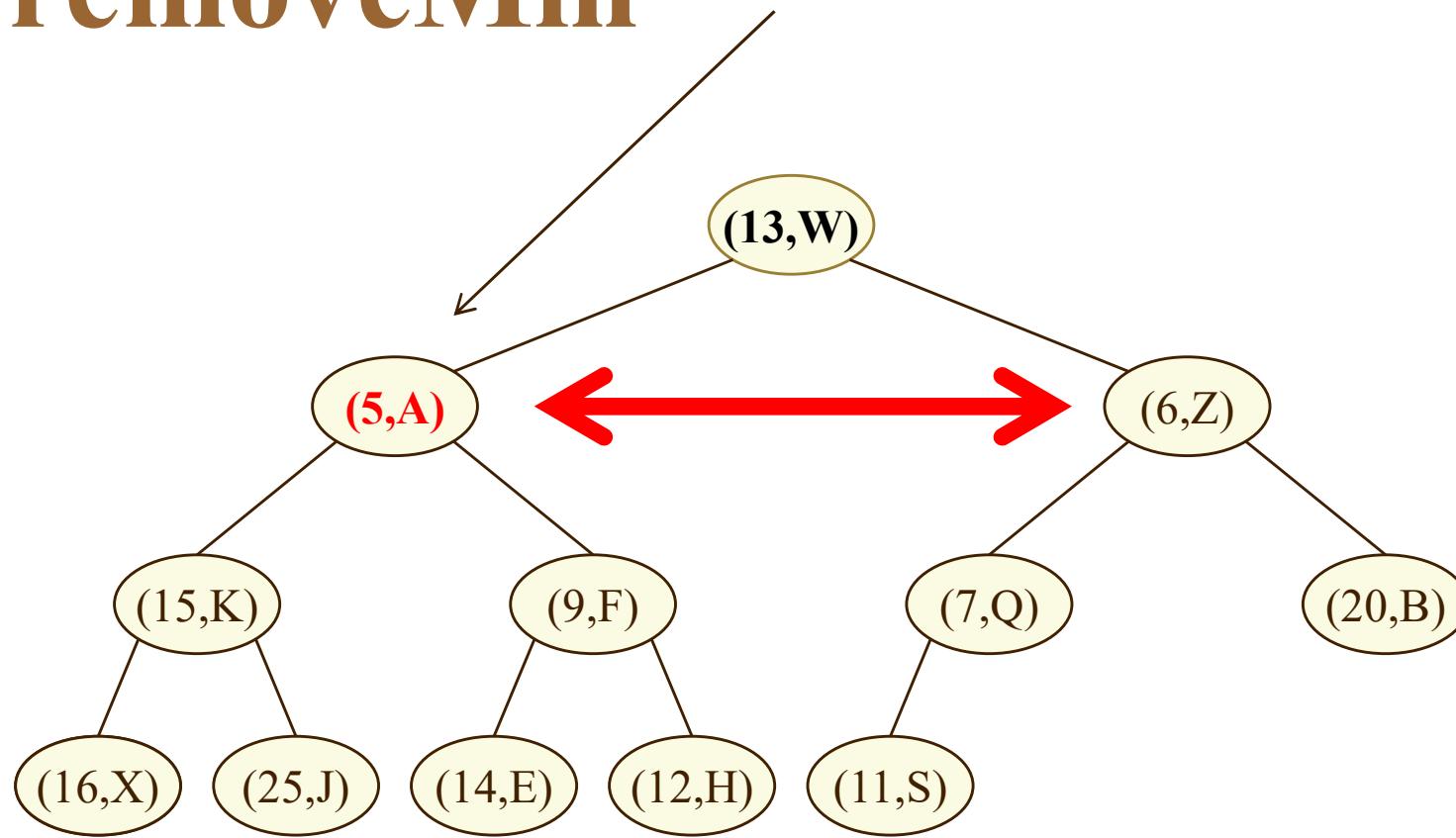


removeMin

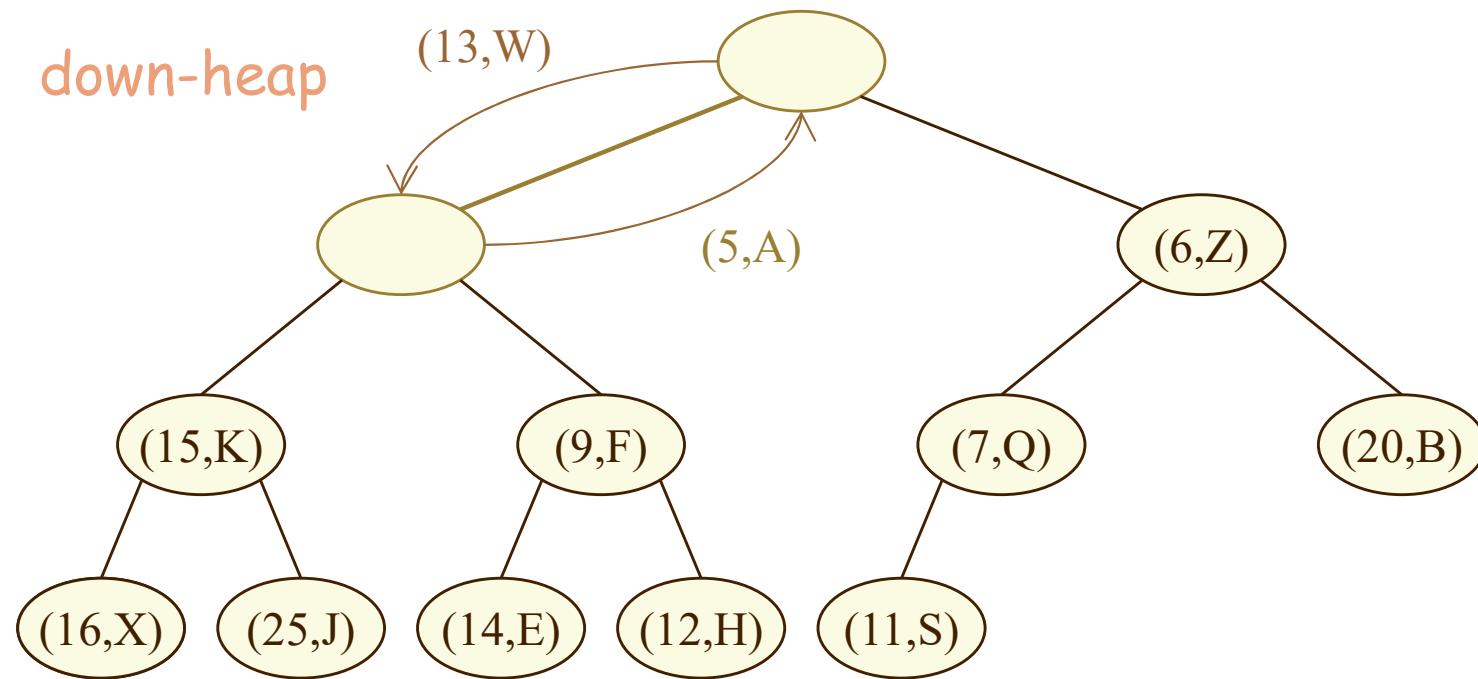


removeMin

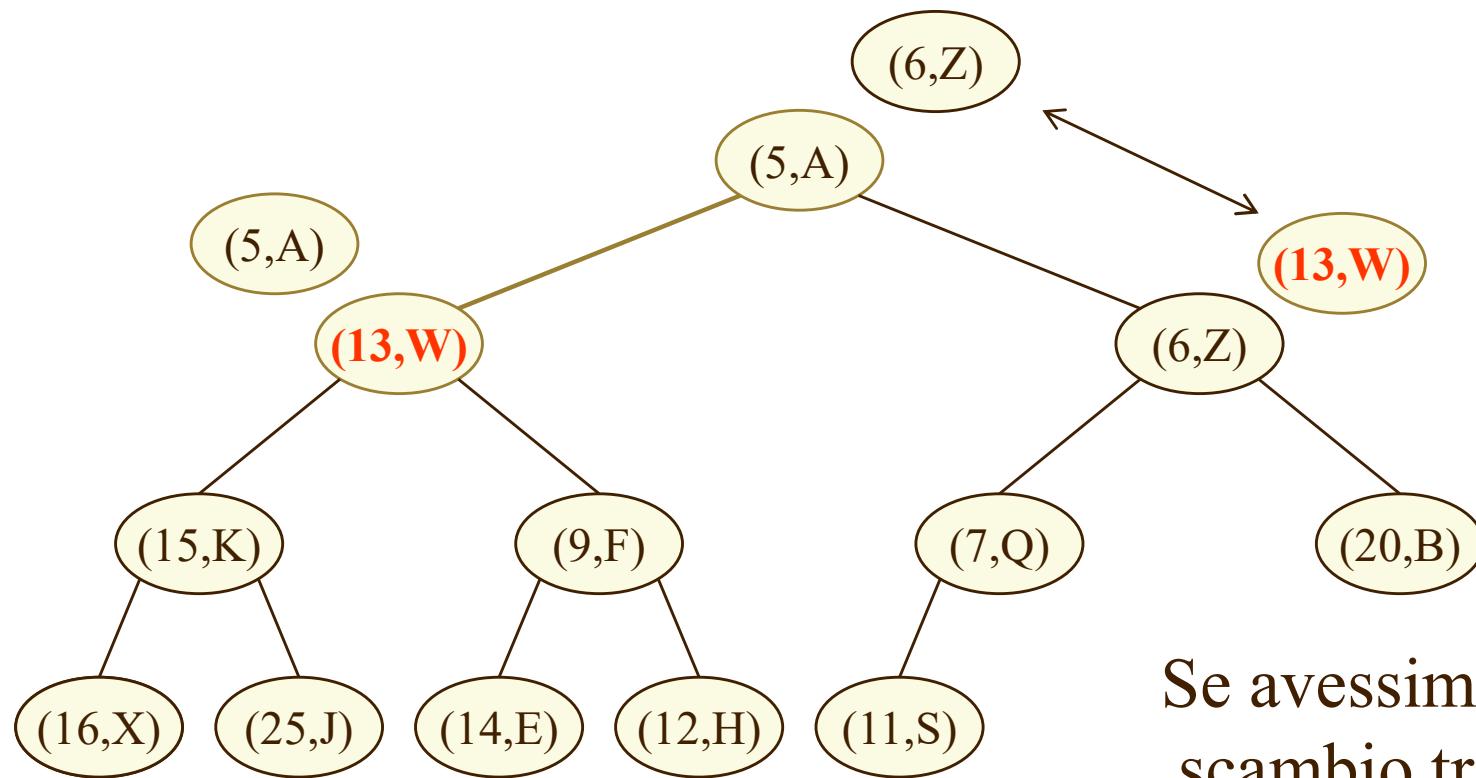
chiave **minima** tra i due figli



removeMin



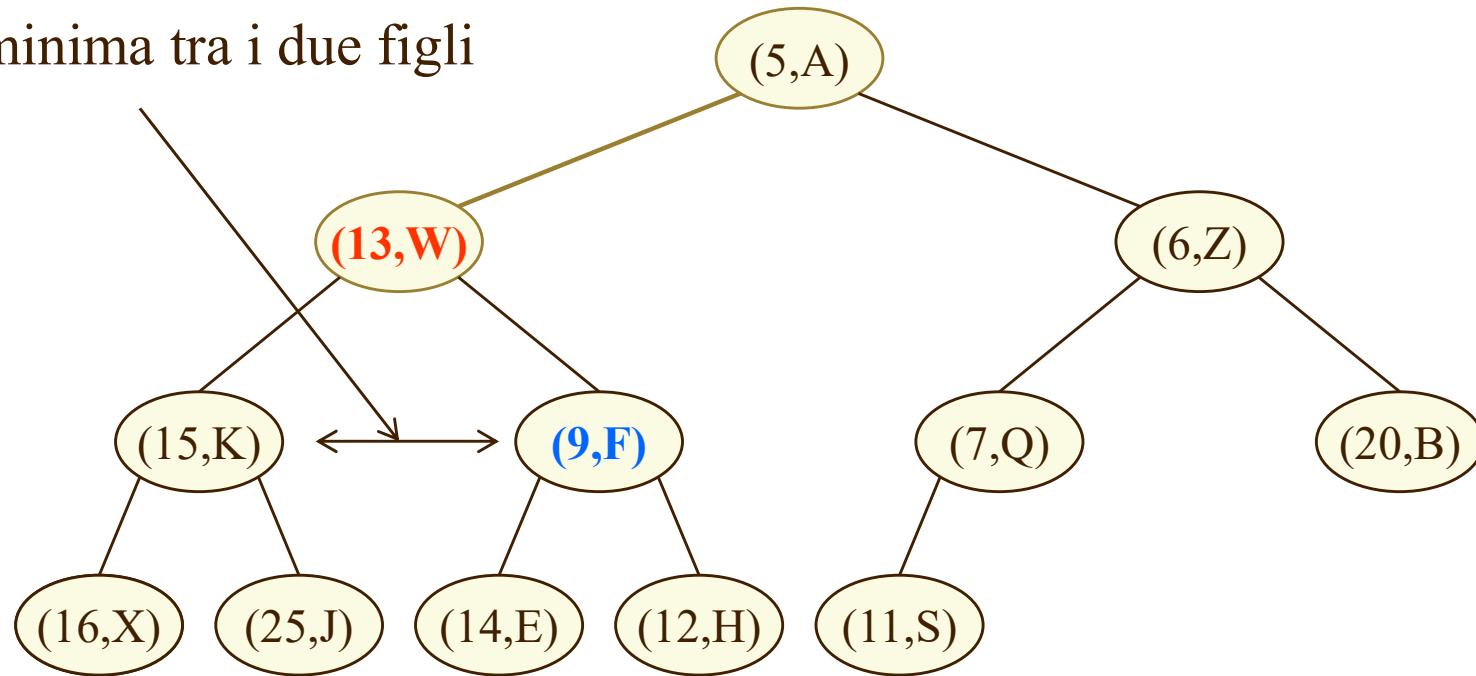
removeMin



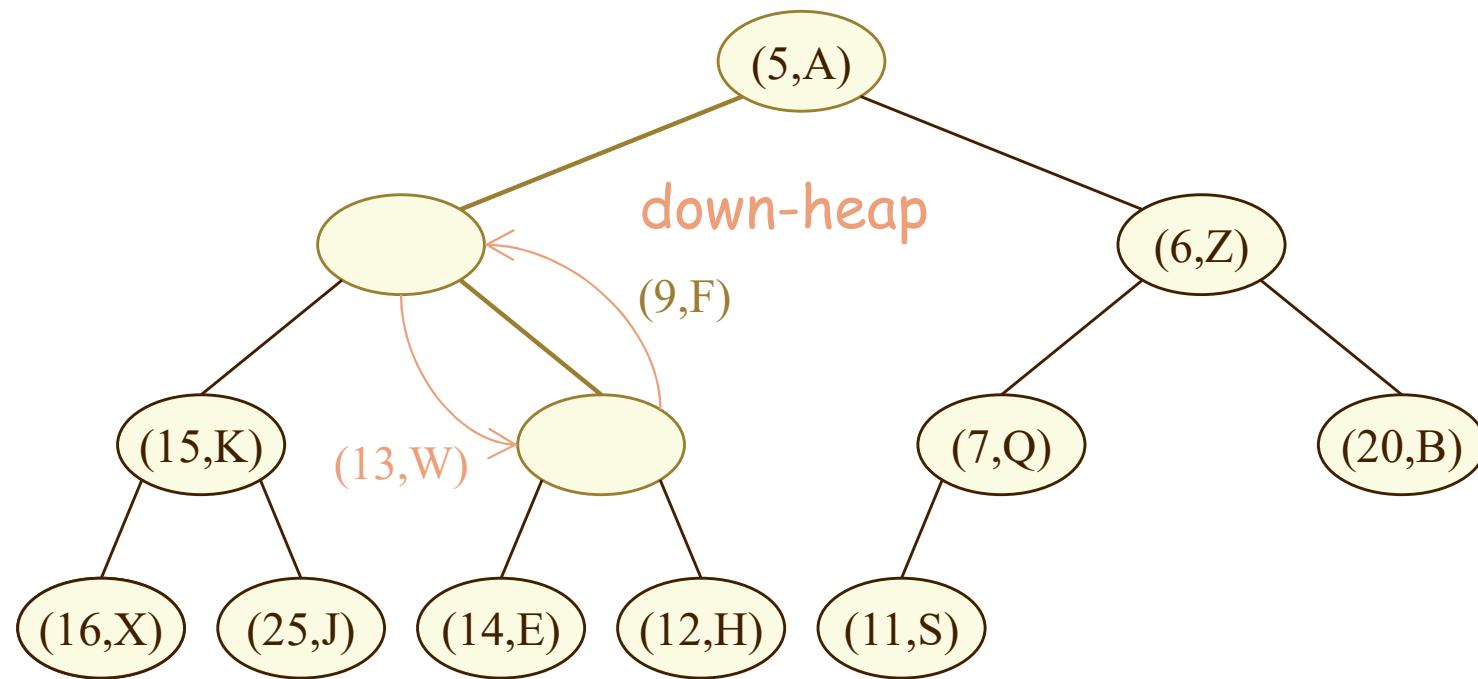
Se avessimo fatto lo scambio tra 13 e 6, anziché tra 13 e 5, ora 6 sarebbe nella radice e violerebbe la proprietà di heap nei confronti del suo (nuovo) figlio sinistro, 5

removeMin

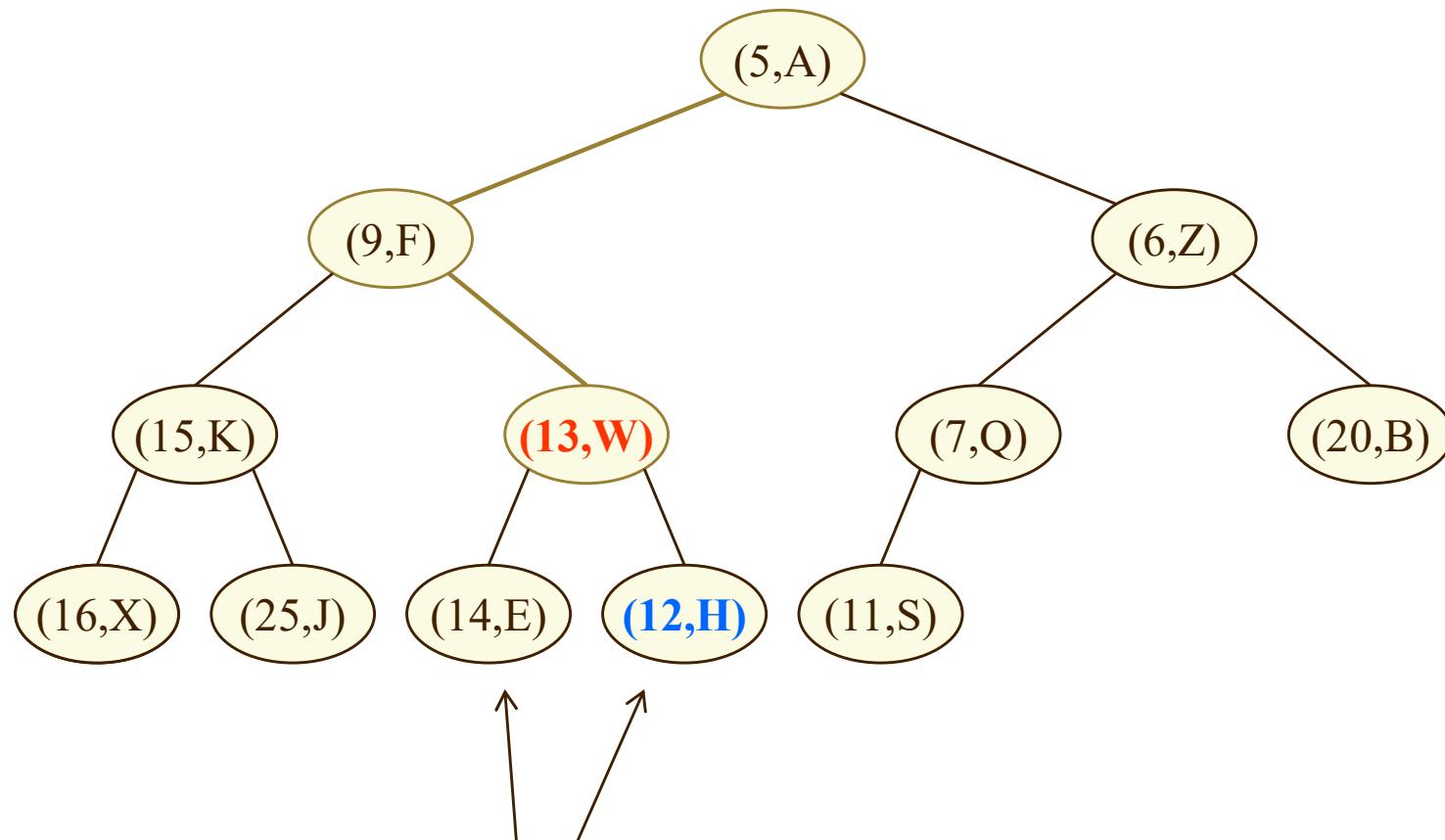
chiave minima tra i due figli



removeMin

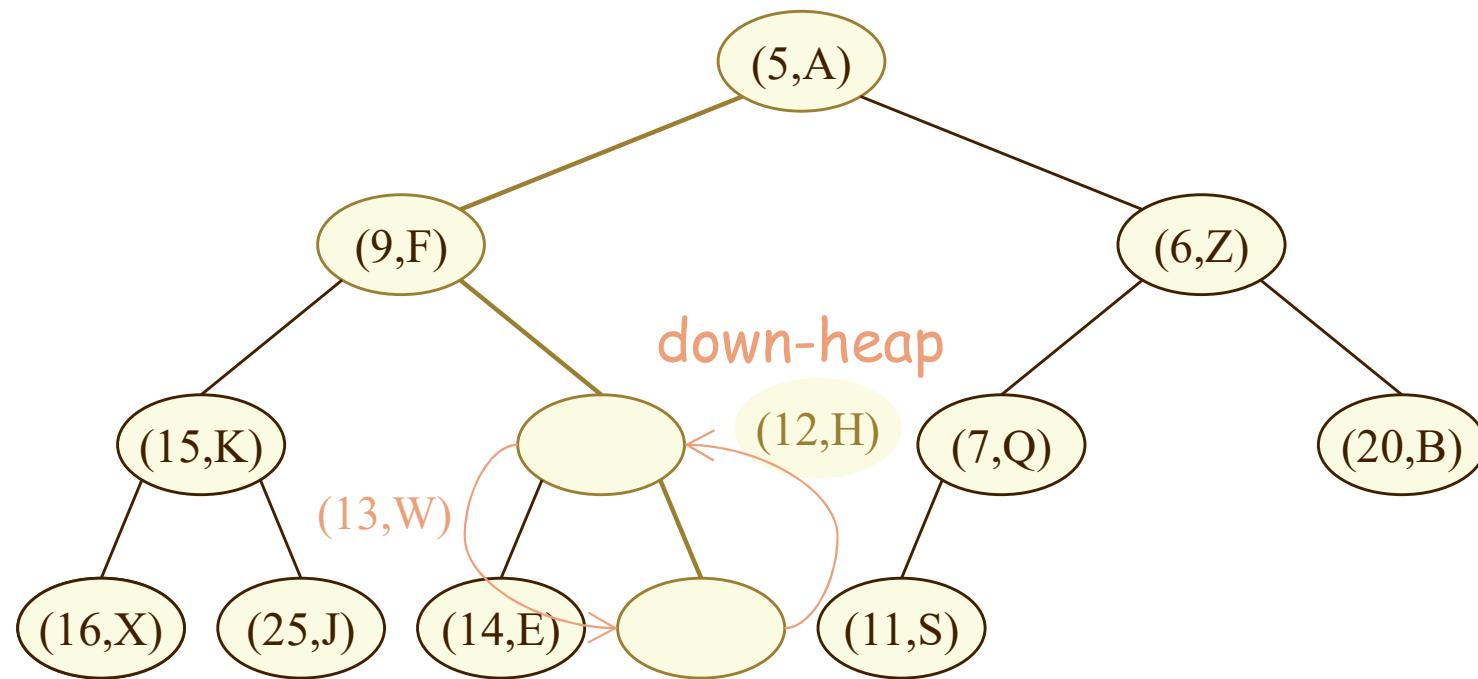


removeMin

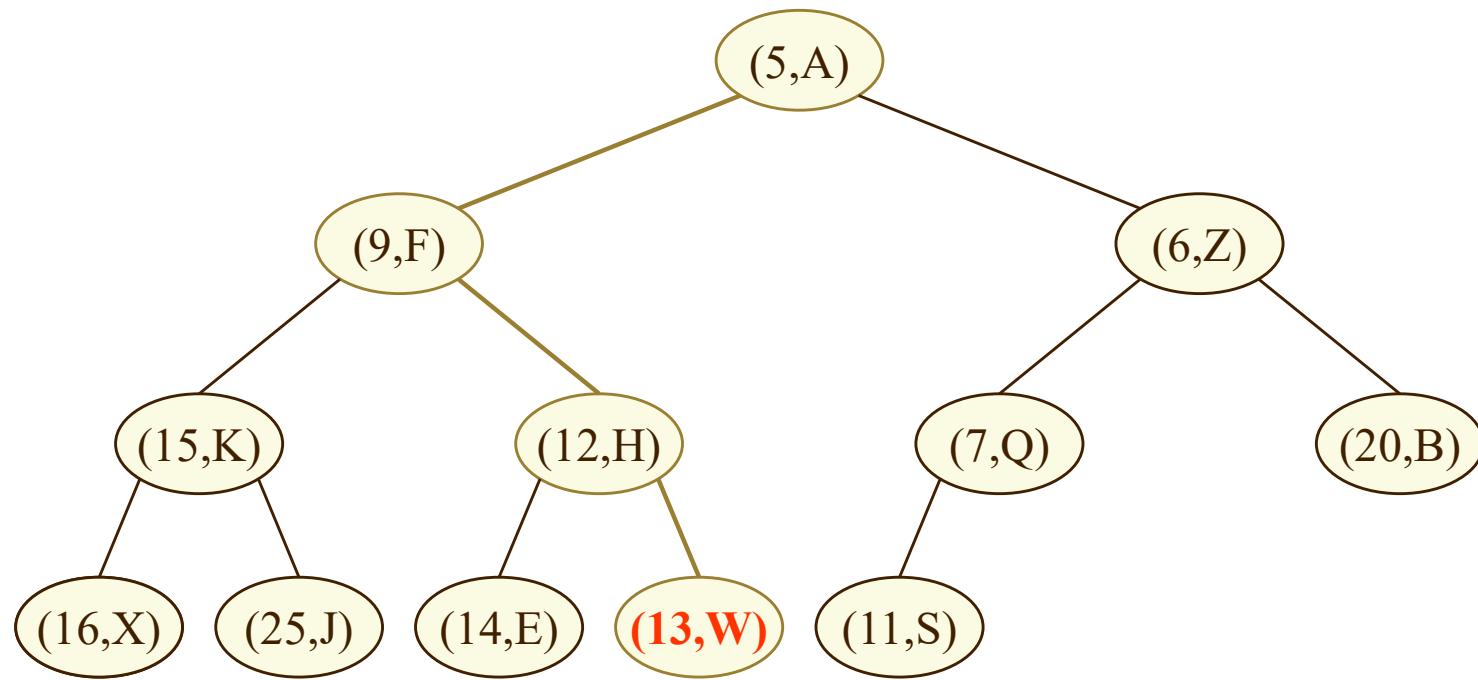


chiave minima tra i due figli

removeMin



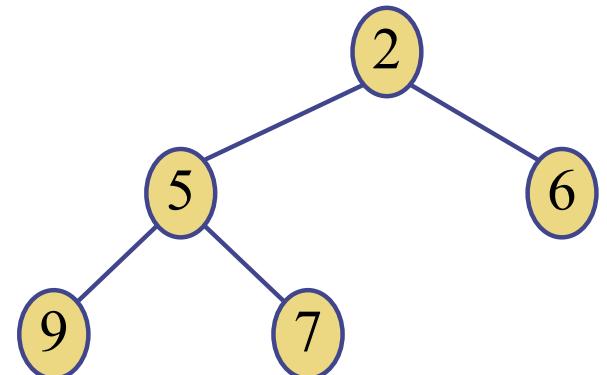
removeMin: finito



Realizziamo una PQ con uno Heap

- Usiamo uno heap memorizzato in un vettore
 - Classe `ArrayHeapPQ`
- Il metodo **size** restituisce la dimensione (logica) del vettore diminuita di uno, in un tempo $\Theta(1)$
 - Il metodo **isEmpty** è, conseguentemente, $\Theta(1)$
- Il metodo **min** restituisce semplicemente il contenuto della radice (cioè della cella di indice 1), in un tempo $\Theta(1)$
- I metodi **insert** e **removeMin** sono più complessi, ma abbiamo dimostrato che sono $O(\log n)$ in generale e $\Theta(\log n)$ nel caso pessimo

È **MOLTO** migliore della PQ realizzata con una lista!!



Realizziamo una PQ con uno Heap

- Se usiamo **uno heap memorizzato in una struttura concatenata**, anziché in un array, cosa succede alle prestazioni asintotiche della coda prioritaria? **Peggiorano?**
 - Sappiamo che i metodi **add** e **remove** di un albero binario **completo** diventano $\Theta(\log n)$, anziché $\Theta(1)$ in media
 - $\Theta(\log n)$ è il tempo necessario per trovare l'ultimo nodo
- Ma cosa accade ai metodi **insert** e **removeMin** della coda prioritaria?
 - Niente, rimangono $O(\log n)$!! Perché, anche quando si usa un array, ci sono up/down heap bubbling che sono $O(\log n)$!!!
 - In realtà mediamente peggiorano un po', perché diventano $\Theta(\log n)$
- **Un albero binario completo realizzato con array è migliore di quello concatenato, ma ciò è indifferente (nel caso pessimo) quando lo si usa (come heap) per realizzare una PQ [rimane però un risparmio di memoria, perché non ci sono i nodi]**

Realizzare una PQ con uno Heap

- Usa un albero binario completo che memorizza *entry* chiave/valore ed è dotato di un comparatore di chiavi

```
import java.util.Comparator;

public class ArrayHeapPQ<K,V>
    implements PriorityQueue<K,V>
{
    private Comparator<K> comp;
    private ArrayCompleteBT<Entry<K,V>> heap;
    public ArrayHeapPQ(Comparator<K> c)
    {
        comp = c;
        heap = new ArrayCompleteBT<Entry<K,V>>();
    }
    public int size() { return heap.size(); }
    ...
}
```

Riassunto: Prestazioni Priority Queue

	Listা non ordinata	Listা ordinata	Heap
insert	$\Theta(1)$	$O(n)$	$O(\log n)$
min	$\Theta(n)$	$\Theta(1)$	$\Theta(1)$
removeMin	$\Theta(n)$	$\Theta(1)$	$O(\log n)$

□ Attenzione a non fare confusione tra

- **Albero binario completo**
 - Ha soltanto proprietà topologiche! Ammette inserimenti e rimozioni soltanto nell'ultimo nodo.
- **Heap**
 - È un albero binario completo con dati che soddisfano la proprietà di heap.
- **Coda prioritaria realizzata mediante uno heap**
 - È un tipo di dato astratto (coda prioritaria) realizzato mediante una struttura dati concreta (heap).
 - Deve realizzare le proprie primitive rispettando i vincoli topologici dell'albero binario completo e i vincoli relazionali della proprietà di heap.
 - Non è "navigabile" dall'esterno!!
L'albero è "nascosto"... cioè encapsulato

Code prioritarie
in `java.util`

`java.util.PriorityQueue`

- In `java.util` non esiste l'interfaccia **PriorityQueue**, bensì una **classe** che ha tale comportamento
 - Errore di progetto della libreria standard... come **Stack**
- Usa la semantica “i valori coincidono con le chiavi”, per cui si inseriscono soltanto chiavi
 - Ma nulla vieta di inserire coppie, triplette, etc, basta che agiscano anche da chiavi di loro stesse! Cioè che siano entità appartenenti a un insieme totalmente ordinato, coppie **confrontabili**...
- Accetta chiavi che implementano **Comparable**, oppure si può fornire un **Comparator** nel costruttore
- Oltre a **size** e **isEmpty**, i metodi si chiamano **add**, **remove** (rimozione del minimo) e **peek** (ispezione del minimo)
 - Se serve una MaxPQ basta fornire un comparatore "inverso"
- È realizzata mediante heap, per cui **add** e **remove** hanno prestazioni logaritmiche nel caso peggiore

java.util.Map.Entry

- Il pacchetto `java.util` definisce anche un’interfaccia adeguata per rappresentare il comportamento di *entry*, che, nel pacchetto, vengono usate soprattutto dalle mappe (che vedremo...)
- È definita l’interfaccia `java.util.Map.Entry`
 - È una *interfaccia interna statica pubblica* di `java.util.Map` ma questo è un dettaglio “tecnico” che non ci interessa

```
package java.util;
public interface Map<K,V> {
    ...
    public static interface Entry<K,V>
    { K getKey();
        V getValue();
        V setValue(V newVal); // opzionale
    }
}
```

java.util.Map.Entry

```
interface java.util.Map.Entry<K,V>
{ K getKey();
  V getValue();
  V setValue(V newVal); // opzionale
}
```

- Il metodo **setValue** restituisce il vecchio valore ed è “opzionale”: deve essere realizzato, ma può semplicemente lanciare **UnsupportedOperationException**
- Possiamo definire **un nome più comodo** per l’interfaccia
 - Così il codice che avevamo scritto usando la “nostra” **Entry** non cambia!

```
public interface Entry<K,V>
    extends java.util.Map.Entry<K,V>
{
    // intenzionalmente vuoto, serve solo
    // a creare un "sinonimo"
}
```

java.util.AbstractMap.SimpleEntry

- Non abbiamo bisogno di progettare una classe che implementi `Entry<K, V>`, possiamo usare questa

```
package java.util;
public abstract class AbstractMap<K,V>
{
    ...
    public static class SimpleEntry<K,V>
        implements Map.Entry<K,V>
    {
        ... } // classe interna statica pubblica
    }           // ha setValue funzionante
```

- Usiamo, però, un nome più comodo...

```
public class SimpleEntry<K,V>
    extends java.util.AbstractMap.SimpleEntry<K,V>
    implements Entry<K,V> // la "nostra" interfaccia
{
    public SimpleEntry(K key, V value)
    { super(key, value); } // non serve altro
}
```

- Con queste definizioni, possiamo riutilizzare tutto il codice che avevamo scritto, senza usare un nostro progetto di **Entry** (per quanto fosse semplice...)

```
public interface Entry<K,V>
    extends java.util.Map.Entry<K,V>
{ // intenzionalmente vuoto
}
```

```
public class SimpleEntry<K,V>
    extends java.util.AbstractMap.SimpleEntry<K,V>
    implements Entry<K,V> // la "nostra" interfaccia
{ public SimpleEntry(K key, V value)
    { super(key, value); } // non serve altro
}
```

Priority Queue Sort

Applicazione di coda prioritaria

- Dato un insieme di n elementi in cui sia definita una relazione d'ordine totale f e memorizzato in una struttura posizionale lineare S , lo si vuole trasferire in un'altra struttura posizionale lineare T in modo che la nuova relazione posizionale sia quella indotta dalla relazione d'ordine f

- **Cioè vogliamo ordinare una lista!** ☺
- Svuotiamo (o usiamo un iteratore su) S , inserendo i suoi n elementi in una coda prioritaria P (inizialmente vuota) che usi f
 - I valori coincidono con le chiavi (solitamente si usano solo le chiavi, senza usare esplicitamente le coppie chiave/valore)
- Svuotiamo P (nell'unico modo possibile), inserendo ordinatamente gli n elementi in T , inizialmente vuota
 - Se S era stata svuotata, T può coincidere con S
- Si parla di **Priority Queue Sort**:
Le prestazioni dipendono da quelle della coda prioritaria
 - Coda prioritaria realizzata con lista non ordinata oppure lista ordinata: prestazioni $O(n^2)$, ma non analizziamo
 - **Coda prioritaria realizzata con heap**

PQ Sort = Heap Sort

- **Priority Queue Sort:** Per ordinare n elementi di una lista S , le prestazioni dipendono da quelle della coda prioritaria
 1. Estraiamo (in qualunque ordine) gli n elementi da S e li inseriamo in una coda prioritaria P inizialmente vuota (gli elementi coincidono con la propria priorità)
 2. Svuotiamo P (nell'unico modo possibile), reinserendo ordinatamente gli n elementi nella lista S , rimasta vuota
- Se usiamo una coda prioritaria realizzata con heap
 - La fase 1 ha prestazioni $O(n \log n)$
 - Ogni inserimento ha prestazioni $O(\log k)$, dove k è la dimensione (via via crescente) della coda prioritaria al momento dell'inserimento, quindi le prestazioni complessive sono $O(\log 1 + \log 2 + \dots + \log (n - 1) + \log n)$, cioè $O(n \log n)$
 - La fase 2 ha prestazioni $O(n \log n)$, facendo lo stesso calcolo
 - L'intero ordinamento (**Heap Sort**) ha prestazioni **$O(n \log n)$**

Vedremo che in realtà si può fare sul posto

Esercizio

Facile

- Si potrebbe pensare: magari **non è possibile che**, durante gli n inserimenti nella PQ, **ci si trovi ogni volta nel caso pessimo...** quindi le prestazioni complessive dell'inserimento, pur essendo certamente $O(n \log n)$, sono anche, ad esempio, $O(n)$...
- **Descrivere una sequenza di n inserimenti in uno heap che richieda un tempo $\Omega(n \log n)$**
- **Soluzione:** Perché ciò accada, ogni inserimento (o almeno "la maggior parte" degli inserimenti) deve essere "un caso pessimo"
 - Un inserimento è un caso pessimo se il valore inserito nell'ultimo nodo dello heap deve risalire fino alla radice
 - Questo accade se e solo se il valore inserito è il valore minimo presente nello heap, perché nella radice ci deve sempre essere il valore minimo
 - Questo accade per ogni inserimento se e solo se gli n elementi vengono inseriti nello heap in ordine decrescente, cosa certamente possibile
 - **Quindi, nel caso pessimo, Heap Sort è $\Theta(n \log n)$**

Heap Sort “sul posto”

- Se la lista S da ordinare è una lista con indice, si può evitare di usare esplicitamente uno heap, usando invece direttamente la porzione iniziale (di dimensione crescente) di tale lista per realizzare lo heap che serve alla coda prioritaria
- Le prestazioni asintotiche non cambiano
 - Le prestazioni migliorano per un fattore moltiplicativo, ovviamente ininfluente sulle prestazioni asintotiche ma comunque importante nei casi pratici
 - Si ha anche un risparmio di memoria, sempre per un fattore moltiplicativo (si usa circa metà memoria)

Heap Sort “sul posto”

- Si sceglie una diversa funzione di numerazione p per l'implementazione dello heap nell'array: serve una funzione che usi 0 come primo indice valido, ad esempio

$$p(T.\text{root}()) = 0$$

$$p(T.\text{left}(u)) = 2p(u) + 1$$

$$p(T.\text{right}(u)) = 2p(u) + 2$$

$$p(T.\text{parent}(u)) = \lfloor ((p(u) - 1)/2) \rfloor$$

- Si parte con l'array S da ordinare e lo si suddivide, dal punto di vista logico, in due parti
 - La parte iniziale, con indici da 0 a i , contiene lo heap in fase di costruzione
 - La parte finale, con indici da $i + 1$ a $n - 1$, contiene la lista in fase di svuotamento

Heap Sort “sul posto”

- Si parte con l'array S da ordinare e lo si suddivide, dal punto di vista logico, in due parti
 - La parte iniziale, con indici da 0 a i , contiene lo heap in fase di costruzione
 - La parte finale, con indici da $i + 1$ a $n - 1$, contiene la lista in fase di svuotamento
- Per $i = 0$, il primo elemento viene "estratto" dalla lista e viene memorizzato come radice dello heap
- Per $i = 1$, il secondo elemento viene "estratto" dalla lista e viene memorizzato (senza muoversi!) nella cella di indice 1, quindi diventa il figlio sinistro della radice dello heap
 - A questo punto si esegue up-heap bubbling per tale elemento, nello heap di dimensione $i + 1$ (cioè 2) memorizzato nella parte iniziale dell'array
- Per $i = 2$, il terzo elemento viene "estratto" dalla lista e viene memorizzato (senza muoversi!) nella cella di indice 2, quindi diventa il figlio destro della radice dello heap
 - A questo punto si esegue up-heap bubbling per tale elemento, nello heap di dimensione $i + 1$ (cioè 3) memorizzato nella parte iniziale dell'array
- E via così, fino a $i = n - 1$

Heap Sort “sul posto”

- L’intera procedura di n inserimenti effettuata sul posto "trasforma" a tutti gli effetti l’array in uno heap (memorizzato in un array)
 - Tale procedura prende spesso il nome di **heapify** (cioè, appunto, "trasforma in uno heap")
- Al termine della procedura di "inserimento nello heap" o di "costruzione dello heap" l’array che conteneva la lista S contiene lo heap
- Ora "estraiamo" uno dopo l’altro tutti gli elementi dallo heap, che ad ogni passo diminuisce di dimensione, e usiamo la parte finale dell’array (che aumenta corrispondentemente di dimensione) per memorizzare la lista ordinata
 - Naturalmente ad ogni "estrazione" si scambia il contenuto della radice (che va a finire nella lista ordinata) con il contenuto dell’ultimo nodo dello heap, eseguendo poi down-heap bubbling a partire dalla radice
- Unico inconveniente: la lista risulta ordinata "al contrario", con l’elemento minimo in fondo
 - Basta invertirne l’ordine alla fine (in un tempo $\Theta(n)$), oppure usare una coda prioritaria che tenga l’elemento massimo nella radice

Lezione 26

Mappe

ADT mappa

- Una **mappa** (*map*) è un contenitore che memorizza **valori** associati a una **chiave** (o etichetta o indice)
 - La funzione principale di una mappa è quella di consentire la **ricerca di un valore presente nel contenitore, fornendone la chiave**
 - **Sembra** simile alla coda prioritaria, invece qui la chiave ha lo scopo di identificare **univocamente** un valore all'interno del contenitore
 - Es. studenti identificati mediante numero di matricola, persone identificate mediante codice fiscale
 - Quindi, una mappa memorizza coppie chiave/valore (*entry*), ma
 - **Le chiavi devono essere distinte: nella mappa non possono esistere due *entry* aventi la stessa chiave**

ADT mappa

A volte viene chiamata "dizionario", ma non usiamo questo termine che è un po' (più) ambiguo in letteratura

- Una **mappa** (*map*) è un contenitore che memorizza **valori** associati a una **chiave** (o etichetta o indice)
- Diversamente da quanto visto per la coda prioritaria, **non è necessario che le chiavi appartengano a un insieme totalmente ordinato**, possono essere oggetti di qualsiasi tipo
 - È soltanto necessario che siano tra loro **confrontabili per uguaglianza**, ma questo è vero per qualsiasi entità
 - In Java: metodo **equals** di **Object**...
- Come nelle code prioritarie, le chiavi:
 - possono essere proprietà intrinseche dei valori, oppure
 - possono essere associate ai valori da una particolare applicazione

ADT mappa

- Una **mappa** (*map*) è un contenitore che memorizza **valori** associati a una **chiave** (o etichetta o indice)
- La funzione principale di una mappa è quella di consentire la **ricerca di un valore presente nel contenitore, fornendone la chiave**
 - Se la mappa contiene una coppia (e necessariamente una sola) avente la chiave richiesta, i metodi di ispezione e rimozione restituiscono il valore presente nella coppia
 - Altrimenti, restituiscono un valore “sentinella”
 - In Java, il riferimento **null**
 - Alternativa di progetto: tali metodi **lanciano un’eccezione**, ma
 - **Non ha molto senso semanticamente**: se cerco qualcosa in un contenitore, il fatto di **non** trovarlo **non** è un evento eccezionale!
 - L’esecuzione controllata in un blocco **try/catch** rallenta l’esecuzione

ADT mappa

- Se la mappa **non** contiene una coppia avente la chiave richiesta, i metodi di ispezione e rimozione restituiscono un valore “sentinella”
 - In Java, il riferimento **null**
- Conseguentemente, i **valori** inseriti in una mappa non possono essere **null**!
Altrimenti **null** non sarebbe più un valore sentinella
 - Alternativa: i metodi restituiscono LA COPPIA anziché il valore, così può esistere il valore **null**: complicazione inutile
 - Nota: nella coda prioritaria era utile restituire la coppia, perché l’utente non conosceva la **chiave minima**, qui invece è l’utente che **fornisce** la chiave! Quindi non ha bisogno di conoscerla... gli interessa il valore.
- Nulla vieta, invece, che esista una coppia con **chiave uguale a null**, ma si complica il codice che verifica se due chiavi sono uguali, quindi si ritiene tale **caso non valido**

ADT mappa

Per il confronto tra chiavi si usa il metodo **equals** che, essendo definito in **Object**, è invocabile con qualsiasi tipo, anche generico, come, in questo caso, **K**

- Usiamo metodi *simili* a quelli definiti in **java.util.Map**

```
public interface Map<K,V> extends Container
{ V put(K key, V val); // oppure insert
  V get(K key); // oppure search o find
  V remove(K key);
  Iterable<K> keys();
  Iterable<V> values();
} // size() è il numero di coppie
```

Se **key** o **val** sono **null**,
si lancia
un'eccezione

- La semantica di **put** è Si potrebbe chiamare **insertOrReplace**

- Se nella mappa esiste già una coppia avente chiave uguale a **key**, inserisce nella coppia il nuovo valore, **val**, e restituisce il vecchio valore
- Altrimenti, inserisce nella mappa la nuova coppia e restituisce **null**

keys e **values** restituiscono liste in cui le posizioni delle chiavi e dei valori sono arbitrarie, ma **corrispondenti**, cioè chiave e valore di una stessa coppia si trovano nella stessa posizione nelle due liste; in **java.util.Map** definizione e comportamento di questi metodi differiscono in modo sostanziale ma forse inutilmente complesso

Mappa in una lista

- Possiamo realizzare una mappa **memorizzando le coppie in una lista, ad esempio un array**
 - Non dobbiamo decidere se conservarle in ordine oppure no, perché, in generale, **non è detto che le coppie siano ordinabili! Né le chiavi!**
- **I metodi `get`, `put` e `remove` sono $O(n)$**
 - **`get` e `remove`** devono fare una ricerca sequenziale, tempo $O(n)$
 - Poi, se ha trovato la coppia cercata, **`remove`** richiede un tempo $\Theta(1)$ per eseguire effettivamente la rimozione (**scambio con l'ultima coppia...**)
 - **`put`** potrebbe semplicemente inserire alla fine della lista, in un tempo $\Theta(1)$ (in media, per l'eventuale ridimensionamento dell'array), ma **prima deve verificare se esiste già una coppia con la stessa chiave**, quindi è $O(n)$ [è $\Theta(n)$ se deve effettivamente inserire, perché per decidere di inserire deve aver visitato tutte le coppie]
 - **Usando una lista concatenata le prestazioni non cambiano**
- I metodi che restituiscono liste (**`keys` e `values`**) sono naturalmente $\Theta(n)$

ADT “mappa ordinata”

- Se le **chiavi** sono **ordinabili**, si può definire il tipo di dato astratto “mappa ordinata”
 - **Interfaccia derivata** dalla “mappa” (come **java.util.SortedMap**)
 - Il metodo **keys** restituisce una lista contenente le **chiavi in ordine** e il posizionamento dei valori nella lista restituita da **values** è (come in **Map**) conseguente (quindi i **valori** non sono, in generale, ordinati, né peraltro ordinabili...)
 - Questo fatto non è specificabile nell’interfaccia, perché non esiste il tipo “lista ordinata” da restituire... va scritto in un commento
 - Classi che implementano questa interfaccia vorranno ricevere un **Comparator**, oppure chiavi **Comparable**

```
public interface SortedMap<K,V> extends Map<K,V>
{ // il metodo keys restituisce una lista
  // ordinata
}
```

SortedMap in un array ordinato

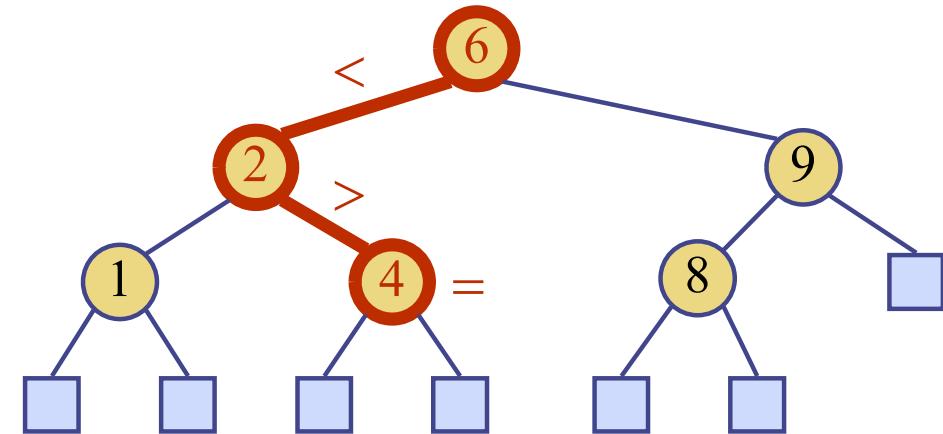
- Il metodo **get** diventa **$O(\log n)$** , usando la **ricerca binaria**
- Il metodo **put** deve fare una ricerca binaria, **$O(\log n)$** , poi (se la coppia è nuova) deve fare l'inserimento nell'array ordinato, un'operazione **$O(n)$** per "farsi largo", quindi il metodo è complessivamente **$O(n)$**
 - Nel caso di "aggiornamento" (cioè se la chiave è già presente) il metodo è $O(\log n)$
- Il metodo **remove** deve fare una ricerca binaria, **$O(\log n)$** , poi (se la coppia esiste) deve fare la rimozione dall'array ordinato ("chiudendo il buco", **$O(n)$**), quindi il metodo è complessivamente **$O(n)$**
 - Nel caso di "rimozione di coppia che non esiste", il metodo è $O(\log n)$
- **Migliorano soltanto le prestazioni del metodo get**
 - **Comunque importante, perché di solito è quello usato più spesso: usiamo una mappa proprio quando vogliamo fare ricerche!**
- Usando una lista concatenata, invece, l'ordinamento non porta alcun miglioramento, perché non si riesce a fare la ricerca binaria in un tempo logaritmico: la lista concatenata non consente accesso casuale

Prestazioni: riassunto

	get	put	remove	keys / values
Mappa in una lista (non ordinata)	$O(n)$	$O(n)$	$O(n)$	$\Theta(n)$
Mappa ordinata in un array ordinato	$O(\log n)$	$O(n)$	$O(n)$	$\Theta(n)$

Si può far meglio per la mappa ordinata?
Certo, vedremo gli **alberi binari di ricerca**

Alberi binari di ricerca



Mappa ordinata (SortedMap)

- La nostra migliore realizzazione di mappa ordinata usa un array ordinato, che, nel caso pessimo e medio, ha prestazioni
 - $O(n)$ per **put** e **remove**, $O(\log n)$ per **get**

Esiste una realizzazione che migliori le prestazioni di **put** e **remove**, senza peggiorare quelle di **get** ?

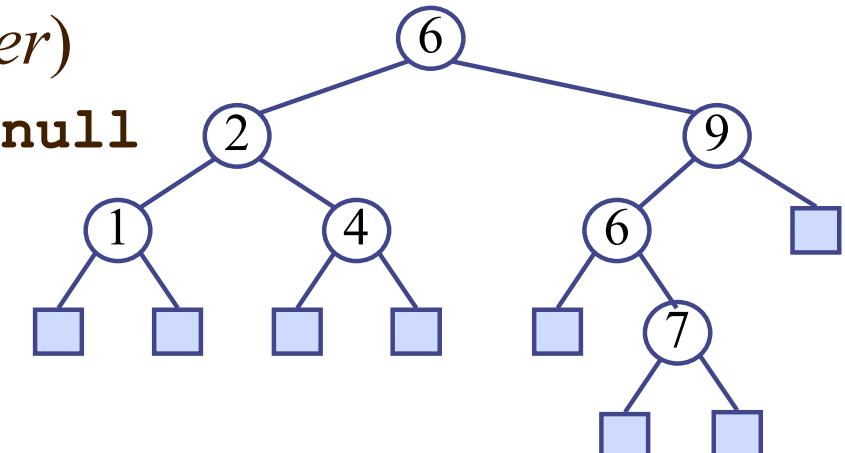
Sì!!

Albero binario di ricerca

- Un albero binario di ricerca (*Binary Search Tree, BST*) è un **albero binario proprio, non vuoto**, con queste ulteriori caratteristiche

- Ogni nodo **interno** v contiene un elemento, $x(v)$, appartenente a un **insieme totalmente ordinato**
- I nodi esterni non contengono alcun elemento (sono nodi “segnaposto”, *placeholder*)
 - In Java, possiamo dire che contengono **null**
- **Per ogni nodo interno v**
 - Gli elementi memorizzati nei nodi interni del sottoalbero sinistro di v sono “non maggiori” di $x(v)$
 - Gli elementi memorizzati nei nodi interni del sottoalbero destro di v sono “non minori” di $x(v)$

Per un paio di slide,
"dimentichiamo"
la mappa...



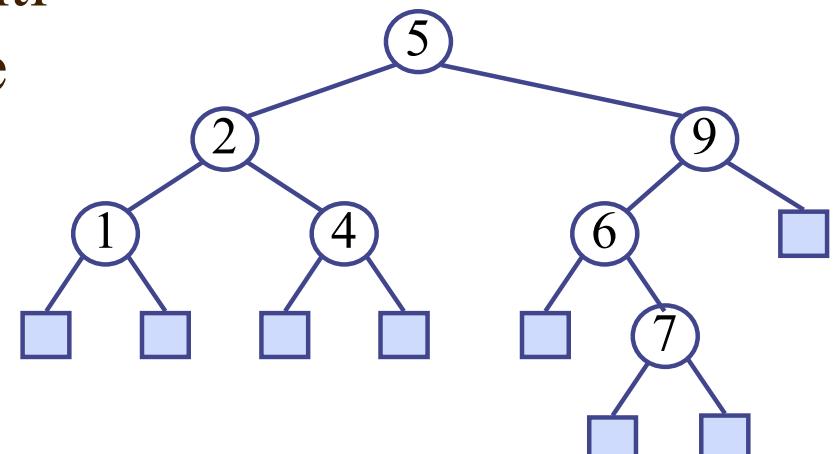
Albero binario di ricerca

□ Alcune osservazioni

- Si può anche definire un albero binario di ricerca che **non** usi i nodi **placeholder**, in modo che l’albero non sia necessariamente proprio, ma gli algoritmi si complicano un po’, inutilmente (ma hanno le stesse prestazioni asintotiche)
 - È lo stesso discorso che abbiamo fatto per i nodi intestazione (*header* e *trailer*) nelle liste concatenate
- **Un BST con i placeholder non può essere vuoto:** come minimo deve avere la sola radice, contenente **null** (in tal caso il BST non contiene elementi, ma ha nodi, quindi **non** è un albero vuoto)
- **Sono ammessi elementi duplicati nell’albero**
 - Attenzione: non stiamo (ancora) parlando di mappa...
 - Per questo diciamo “non maggiori” a sinistra e “non minori” a destra: se non ci fossero duplicati diremmo “minori” a sinistra e “maggiori” a destra

SortedMap in BST

- Proviamo a usare un BST per memorizzare una **mappa ordinata**
- Gli *elementi* usati nella definizione del BST sono, in questo caso, le *coppie* della mappa
 - Gli elementi del BST, cioè le coppie chiave/valore della mappa ordinata, appartengono a un insieme totalmente ordinato se li si confronta usando le chiavi che contengono (le quali, in una mappa ordinata, appartengono a un insieme totalmente ordinato)
 - Nel BST sarebbero ammessi elementi duplicati, ma la mappa non ammette chiavi duplicate, quindi **nei BST usati come mappa non ci saranno elementi duplicati**



SortedMap in BST: memoria

- Che prestazioni ha questa mappa in riferimento all'occupazione di memoria?
 - C'è un nodo interno per ogni coppia della mappa
 - Quanti sono i nodi esterni? È un albero binario **proprio**, quindi ricordiamo che $n_E = n_I + 1$
 - Se la mappa ha n coppie, **l'albero binario di ricerca** che la contiene ha n nodi interni e $n + 1$ nodi esterni, quindi **ha dimensione** $2n + 1 \in \Theta(n)$
- Quindi, **la dimensione della mappa e quella del BST** che la realizza sono legate da proporzionalità lineare: nella valutazione di prestazioni **asintotiche useremo n per indicare indifferentemente una dimensione o l'altra** (distinguendole esplicitamente soltanto quando sarà necessario)

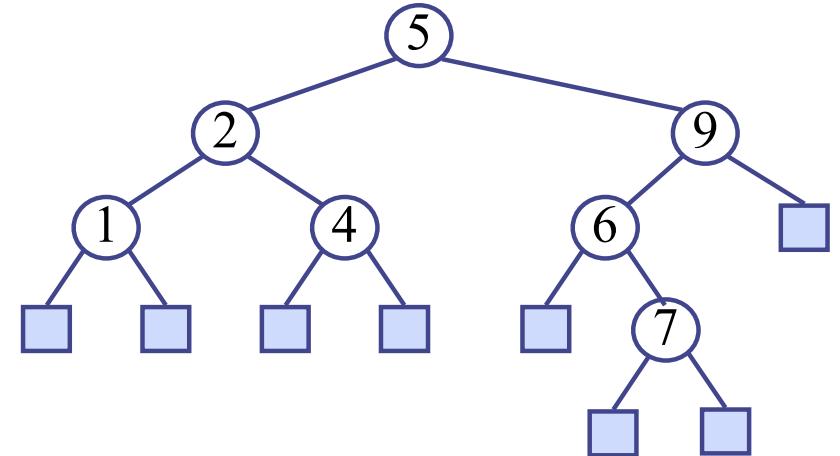
Albero binario di ricerca

□ (Si dimostra una) Importante proprietà

L’attraversamento in ordine **simmetrico** visita i nodi interni di un BST in ordine di **contenuto non decrescente** (ossia crescente, in mancanza di duplicati)

- Es. 1 2 4 5 6 7 9

□ Quindi i metodi **keys** e **values** della mappa ordinata saranno $\Theta(n)$, perché basta fare un attraversamento in ordine simmetrico, usando come azione di visita l’inserimento in fondo a una lista (rispettivamente delle chiavi e dei valori)



È una mappa: quindi **non ci sono chiavi duplicate!**

SortedMap in BST

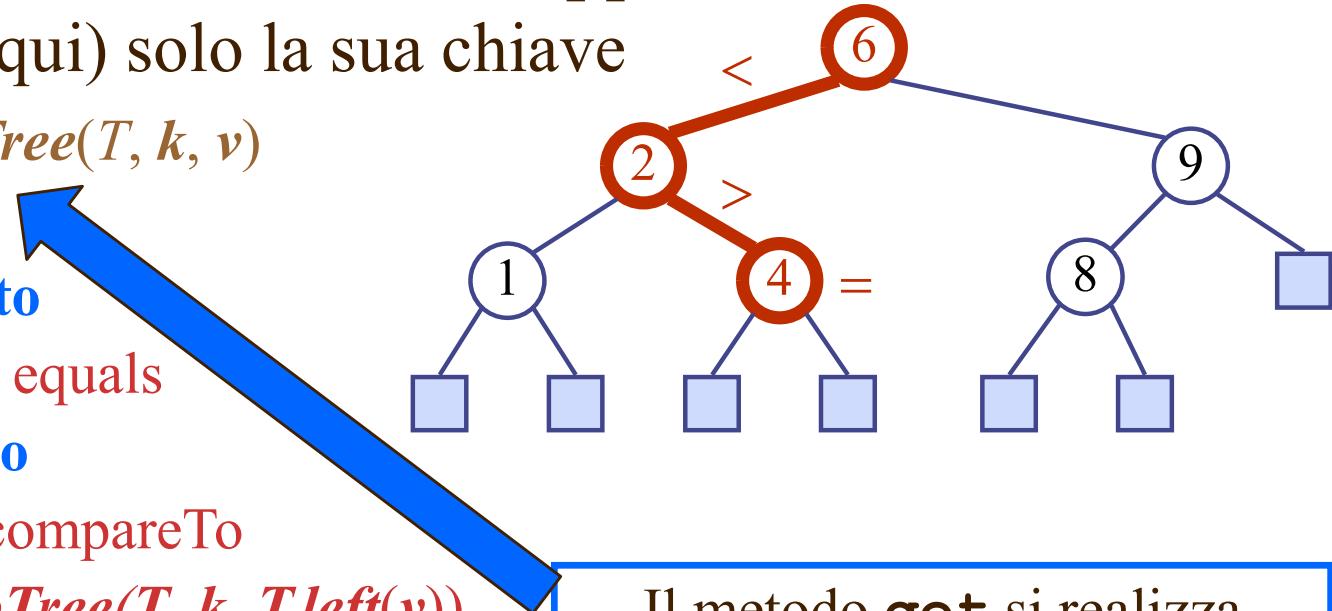
Anche se nel BST sarebbero ammesse

- Usiamo un BST per realizzare **SortedMap**

- In ogni nodo memorizziamo **una coppia**, ma usiamo (e raffiguriamo qui) solo la sua chiave

Algoritmo *SearchInBSTsubTree(T, k, v)*

```
if T.isExternal (v)
    return null // non trovato
(else) if k == key(v) // usare equals
    return value(v) // trovato
(else) if k < key(v) // usare compareTo
    return SearchInBSTsubTree(T, k, T.left(v))
(else) // k > key(v)
    return SearchInBSTsubTree(T, k, T.right(v))
```



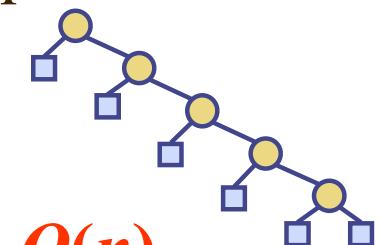
Il metodo **get** si realizza invocando l'algoritmo con $v = T.root()$, che esiste sempre

- Questo algoritmo di ricerca è **simile alla ricerca per bisezione in un array ordinato**, anche se (purtroppo) **non è detto che la dimensione del problema si dimezzi a ogni passo**, dipende (come vedremo) dalla struttura dell'albero (si pensi a un albero "degenere", come un millepiedi...)

SortedMap in BST: prestazioni

□ Che prestazioni ha il metodo **get** di questa mappa?

- La decisione presa in ogni nodo visitato è $\Theta(1)$ (perché il tempo necessario a prendere la decisione non dipende dalle dimensioni dell’albero, anche se può dipendere dalla dimensione dei dati confrontati)
 - Quindi, il tempo speso per una ricerca dipende linearmente dal numero di nodi visitati, che, nel caso peggiore, è pari all’altezza dell’albero, h , perché **a ogni passo si scende** verso un figlio
 - Il metodo **get** è $O(h)$ e, quindi, nel caso pessimo è $O(n)$, perché l’altezza massima di un albero binario proprio di dimensione n è $(n - 1)/2$ (degenera in un “millepiedi” ☺)
 - **Non sembra essere molto promettente**, sappiamo già far meglio: usando un array ordinato, si ottiene una mappa ordinata con **get** che è $O(\log n)$ nel caso pessimo



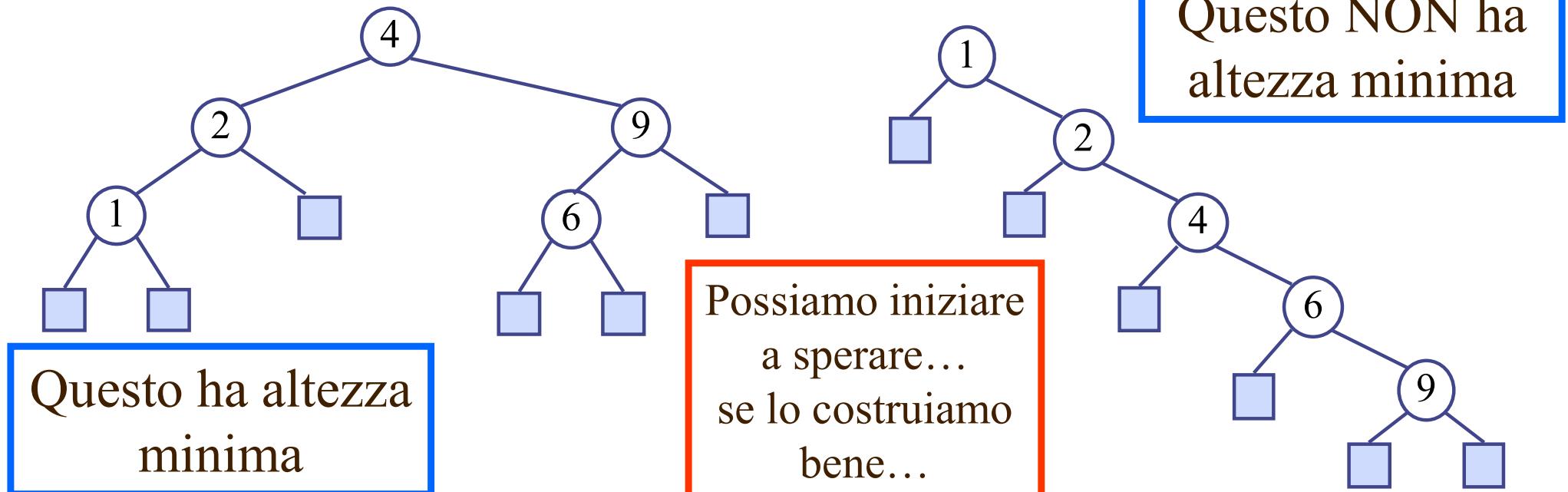
SortedMap in BST: prestazioni

□ Che prestazioni ha il metodo `get` di questa mappa?

- Il metodo `get` è $O(h)$ e, quindi, nel caso pessimo è $O(n)$
 - Non sembra essere molto promettente, sappiamo già far meglio: usando un array ordinato, si ottiene una mappa ordinata con `get` che è $O(\log n)$ nel caso pessimo
- Ma sappiamo che l'altezza **minima** di un albero binario proprio di dimensione n è $\lceil \log_2 (n + 1) - 1 \rceil$, quindi, nel caso di albero con la struttura migliore (es. completo, ma non solo), le prestazioni di `get` sono, nel caso pessimo, $O(\log n)$
 - Facciamo attenzione e teniamo distinti due concetti
 - La ricerca ha un suo **caso pessimo**, indipendentemente dalla struttura dell'albero, che si ha quando deve visitare il massimo numero di nodi, h [solo se l'elemento cercato non è presente]
 - L'altezza dell'albero ha un suo **caso pessimo** (che, cioè, ne massimizza il valore), quando $h = (n - 1)/2$
 - Nel “**caso pessimo**” i due effetti si combinano, ma non è detto che ciò accada! La “combinazione pessima” potrebbe essere evitata...

SortedMap in BST: prestazioni

- C'è speranza di memorizzare una mappa ordinata in un BST in modo che questo abbia **SEMPRE altezza minima**?
 - Il metodo **get** sarebbe **$O(\log n)$ nel caso pessimo**
- Innanzitutto, osserviamo che, dato un insieme di chiavi, $\{1, 2, 4, 6, 9\}$, la loro distribuzione in un BST non è unica



Lezione 27

SortedMap in BST: prestazioni

- **Tesi:** Dato un insieme di chiavi di dimensione n , esiste un BST che le contiene e che rende la ricerca $O(\log n)$ nel caso pessimo
- Facciamo una **dimostrazione costruttiva**
 - Disponiamo le chiavi in un array A e lo ordiniamo
 - **Se A è vuoto (unico caso base), inseriamo nel BST la sola radice/foglia, senza contenuto: FINE**
 - Individuiamo l'elemento "centrale" di A , lo chiamiamo v e lo mettiamo nella radice del BST
 - Gli elementi del semi-array “sinistro” di A , contenente gli elementi minori di v , verranno memorizzati nel sottoalbero sinistro della radice
 - Gli elementi del semi-array “destro” di A , contenente gli elementi maggiori di v , verranno memorizzati nel sottoalbero destro della radice
 - Ripetiamo ricorsivamente la procedura per i due sottoalberi

SortedMap in BST: prestazioni

- **Tesi:** Dato un insieme di chiavi di dimensione n , esiste un BST che le contiene e che rende la ricerca $O(\log n)$ nel caso pessimo
- Vediamo come prosegue la dimostrazione costruttiva
 - Non è difficile dimostrare che **l'albero così costruito è un BST** e che **la ricerca al suo interno esegue gli stessi passi della ricerca per bisezione** nell'array A , quindi **ha prestazioni $O(\log n)$ nel caso pessimo**, come richiesto
 - Sapendo già che la ricerca in un BST è $O(h)$, ne consegue che il BST così costruito ha $h \in O(\log n)$
 - Inoltre, sappiamo che, in qualsiasi albero binario, $h \in \Omega(\log n)$, perché $h \geq \log_2(n + 1) - 1$
 - Quindi, nel BST così costruito $h \in \Theta(\log n)$

Possiamo sperare!

SortedMap in BST: prestazioni

□ Che prestazioni ha la costruzione di questo BST ottimo?

- È un algoritmo a ricorsione doppia
- Ogni passo richiede un tempo $\Theta(1)$, per fare tre cose: identificazione dell'elemento “centrale” di un array, inserimento della radice in un nuovo (sotto)albero, **delimitazione** dei due semi-array
 - Non serve costruire i semi-array: **dato che il contenuto dell'array non viene mai modificato**, le invocazioni ricorsive possono usare come parametri i limiti sinistro e destro dell'array su cui operare, una porzione dell'array originario
- Quante invocazioni ricorsive servono?
 - È facile, ogni invocazione inserisce una nuova coppia nell'albero: dovendo inserire n coppie, servono n invocazioni (a cui si aggiungono le $n + 1$ invocazioni che operano su array vuoti e inseriscono le foglie, prive di contenuto)
- **La costruzione richiede, quindi, un tempo $\Theta(n)$ se si dispone già dell'array ordinato, altrimenti $\Omega(n)$ e $O(n \log n)$ per ordinare l'array**

SortedMap in BST: prestazioni

- Proviamo a realizzare i metodi **put** e **remove** della mappa ordinata,
facendo in modo che l'altezza dell'albero rimanga sempre minima
 - Il metodo **get** è **$O(\log n)$**
 - Per inserire una coppia (**put**), **$O(n)$**
 - Controllo se esiste, $O(\log n)$; se esiste, ne aggiorno il valore, $\Theta(1)$
 - Se non esiste, creo l'array ordinato dall'albero (attraversamento simmetrico, $\Theta(n)$), aggiungo la nuova coppia nell'array mantenendone l'ordinamento ($O(n)$), poi **ricostruisco l'albero**, $\Theta(n)$, che ha nuovamente altezza minima: tempo totale $\Theta(n)$
 - Per rimuovere una coppia (**remove**), **$O(n)$**
 - Controllo se esiste, $O(\log n)$; se non esiste, ho finito
 - Se esiste, creo l'array ordinato dall'albero (attraversamento simmetrico, $\Theta(n)$), elimino la coppia dall'array mantenendone l'ordinamento ($O(n)$), poi **ricostruisco l'albero**, $\Theta(n)$, che ha nuovamente altezza minima : tempo totale $\Theta(n)$

Prestazioni: riassunto

	get	put	remove	keys / values	memoria
Mappa in una lista (non ordinata)	$O(n)$	$O(n)$	$O(n)$	$\Theta(n)$	$\Theta(n)$
Mappa ordinata in un array ordinato	$O(\log n)$	$O(n)$	$O(n)$	$\Theta(n)$	$\Theta(n)$
Mappa ordinata in un BST di altezza minima	$O(\log n)$	$O(n)$	$O(n)$	$\Theta(n)$	$\Theta(n)$

- Ancora non è un granché: **stesse prestazioni asintotiche (in tempo e spazio) della mappa ordinata realizzata con array ordinato**, con prestazioni effettive certamente peggiori

SortedMap in BST: prestazioni

□ Prima idea: riusciamo a realizzare **put** e **remove** in un tempo che, nel caso pessimo, sia migliore di $O(n)$?

- **Un primo obiettivo: cerchiamo di farli $O(h)$**
 - Ovviamente, se la struttura del BST sarà tale da far assumere all'altezza il suo valore pessimo, i metodi saranno nuovamente $O(n)$, ma, in generale, potranno essere migliori
 - **Non richiediamo che l'altezza rimanga minima, quindi anche get è $O(h)$, cioè, in generale, può peggiorare rispetto a $O(\log n)$**

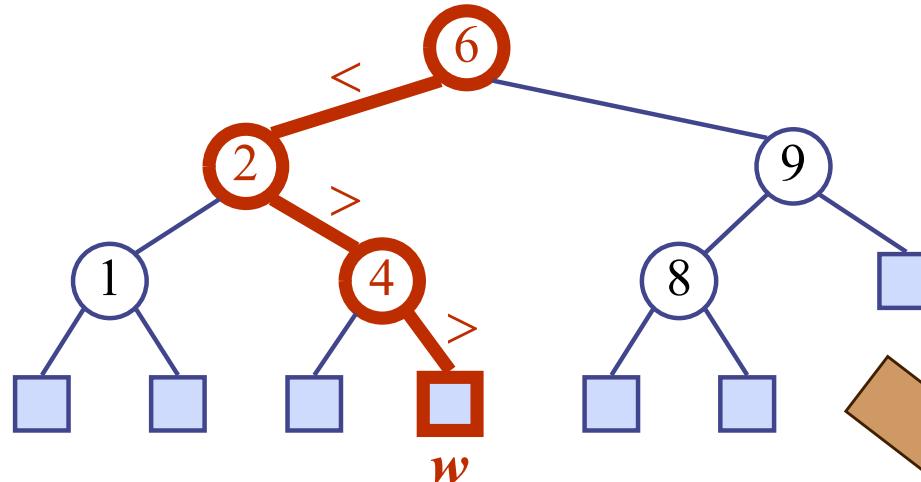
SortedMap in BST: inserimento

- Strategia di **inserimento** di (k, v) :
Cerco la chiave k , usando l'algoritmo del metodo **get**
 - Se (in un tempo $O(h)$) la trovo, sostituisco il vecchio valore associato a k con il nuovo, v , e restituisco il vecchio: **ho finito, tempo totale $O(h)$**
 - Altrimenti, il metodo **get** è arrivato (in un tempo $O(h)$) in una foglia, dove avrebbe trovato la chiave se questa ci fosse stata
 - Tale foglia è, quindi, una posizione valida in cui inserire la nuova coppia **senza fare altre modifiche alla struttura dell'albero**
 - Trasformo la foglia in un nodo interno contenente la nuova coppia e inserisco due nuove foglie (ovviamente “vuote”) come suoi figli: operazioni complessivamente $\Theta(1)$
 - **Ho finito, tempo totale $O(h)$**

Il BST
rimane
un BST

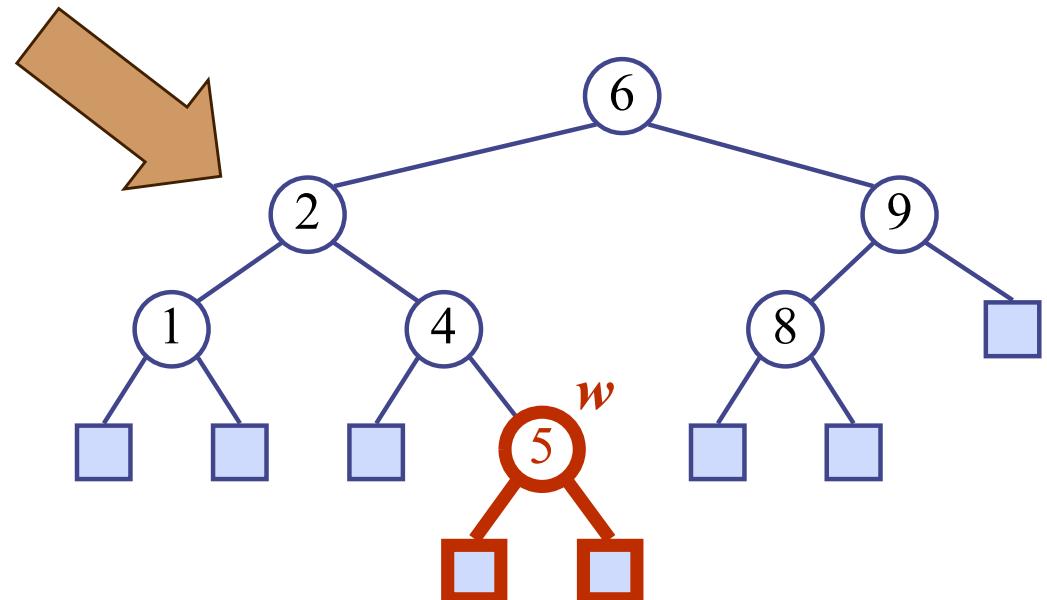
SortedMap in BST: inserimento

- Esempio di **inserimento** di $(5, v)$



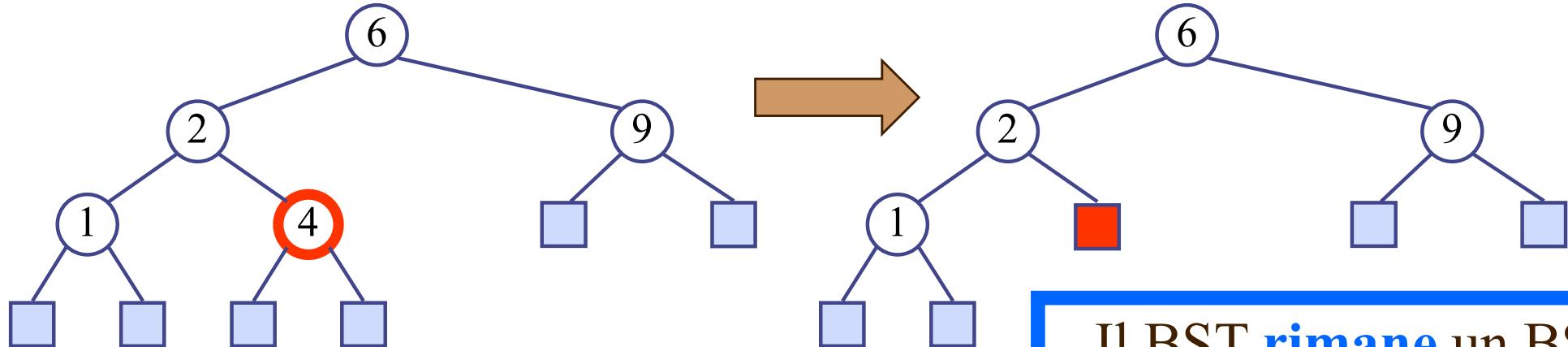
La dimensione della mappa aumenta **di uno**, mentre **la dimensione del BST aumenta di due!**

L'altezza del BST può aumentare (di uno) oppure no (se la nuova chiave è uguale a 10...) In generale, come in questo caso, **la nuova altezza può non essere minima**, anche se prima lo era



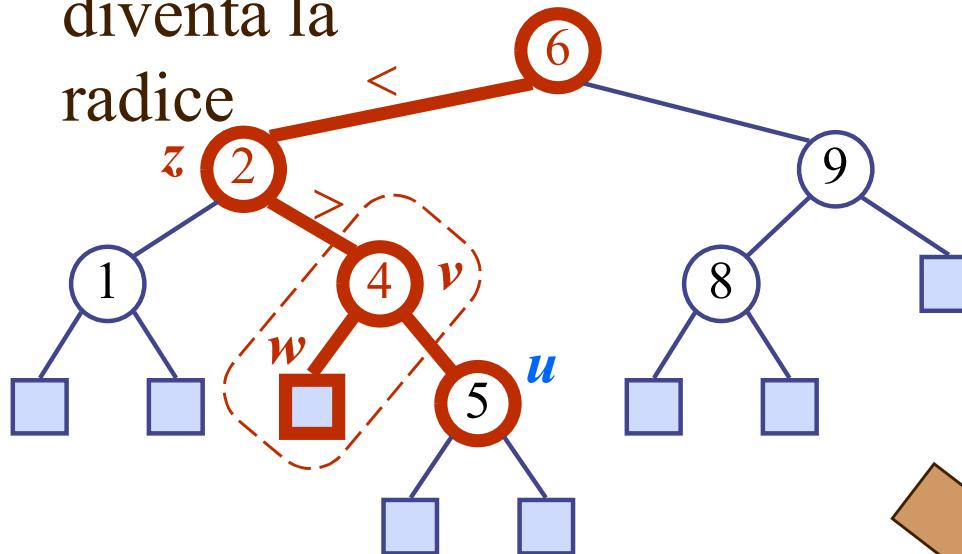
SortedMap in BST: rimozione

- Strategia di **rimozione** di (k, v) , nota la chiave k :
Cerco la chiave k , usando l'algoritmo del metodo **get**
 - Se (in un tempo $O(h)$) non la trovo, restituisco **null**:
ho finito, tempo totale $O(h)$
 - Altrimenti, il metodo **get** è arrivato (in un tempo $O(h)$) nel nodo v contenente la chiave cercata: devo eliminare il nodo v
 - Se v ha **come figli due foglie**, in un tempo $\Theta(1)$ le elimino e trasformo v in una foglia



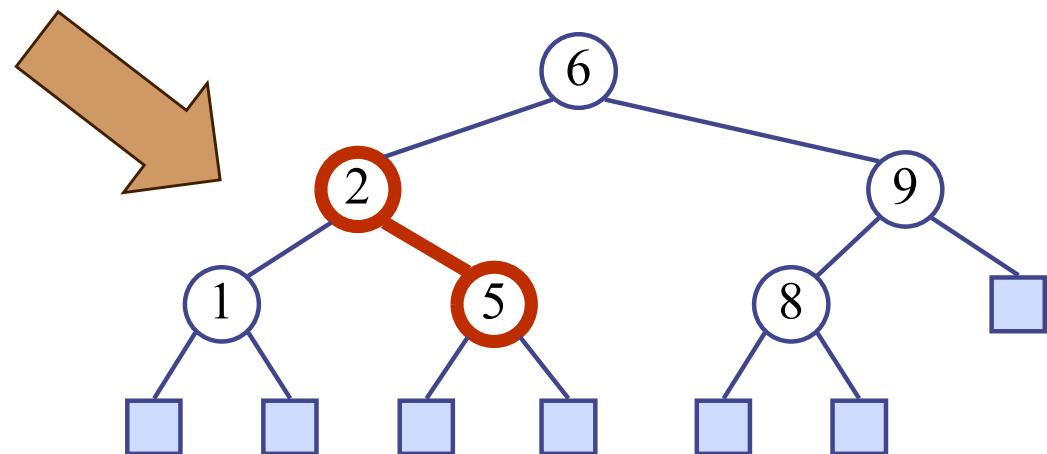
SortedMap in BST: rimozione

- Se v ha **come figli una foglia**, w , e **un nodo interno**, u , in un tempo $\Theta(1)$ elimino w e v , poi u prende il posto di v come figlio del genitore (z) di v , oppure diventa la radice



In generale, **la nuova altezza può non essere minima**, anche se prima lo era: ad esempio, se nel BST sottostante elimino 8, poi elimino 9, poi elimino 1, rimane una mappa con tre valori (2, 5, e 6) in un BST di altezza 3, mentre lo si potrebbe fare di altezza 2 (con 5 nella radice)

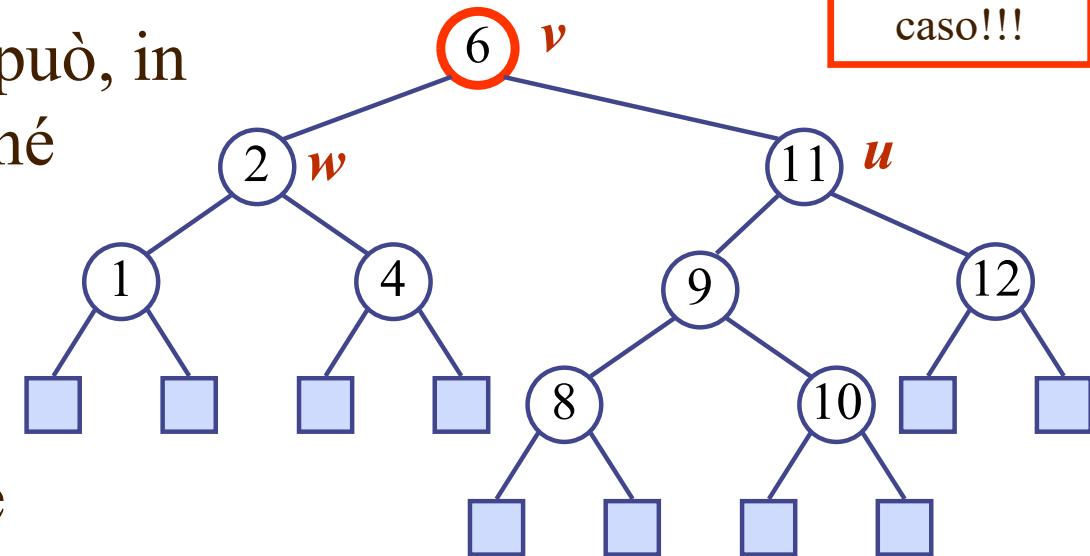
Il BST **rimane** un BST, perché u si trova, rispetto al genitore z di v , nel medesimo sotto-albero



SortedMap in BST: rimozione

- Se v ha **come figli due nodi interni**, w e u , il problema si complica: immaginiamo di voler rimuovere **6**!

- Nessuno dei due figli di v può, in generale, sostituire v , perché lascerebbe orfano uno dei propri sottoalberi!
- Inoltre, in generale, le chiavi contenute nei due figli di v non sono adatte a sostituire la chiave contenuta in v , rispettando la proprietà del BST
 - Infatti, in questo caso, né 2 né 11 possono andare nella radice al posto di 6, lasciando inalterati **i contenuti** dei due sottoalberi della radice (necessariamente ristrutturandoli), perché quello sinistro contiene $4 > 2$ e quello destro contiene $8 < 11$

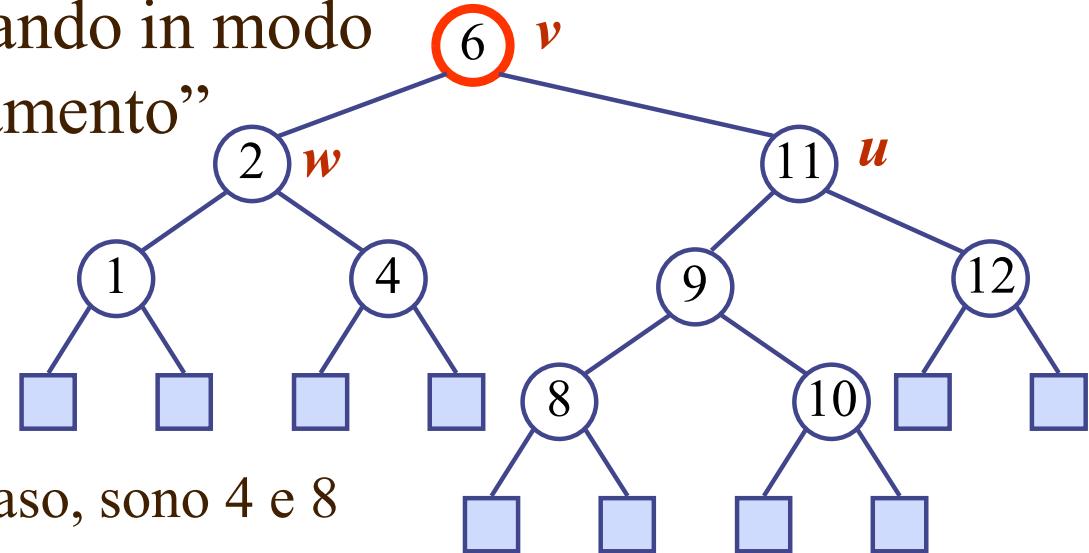


Il fatto che **6** sia nella radice è un caso!!!

SortedMap in BST: rimozione

- Invece di procedere per tentativi... **ragioniamo!** Quali sono i candidati a sostituire 6, modificando in modo minimale la “proprietà di ordinamento” del BST ?

- Sono i valori “più vicini” a 6 nella sequenza ordinata delle chiavi memorizzate nel BST! In questo caso, sono 4 e 8

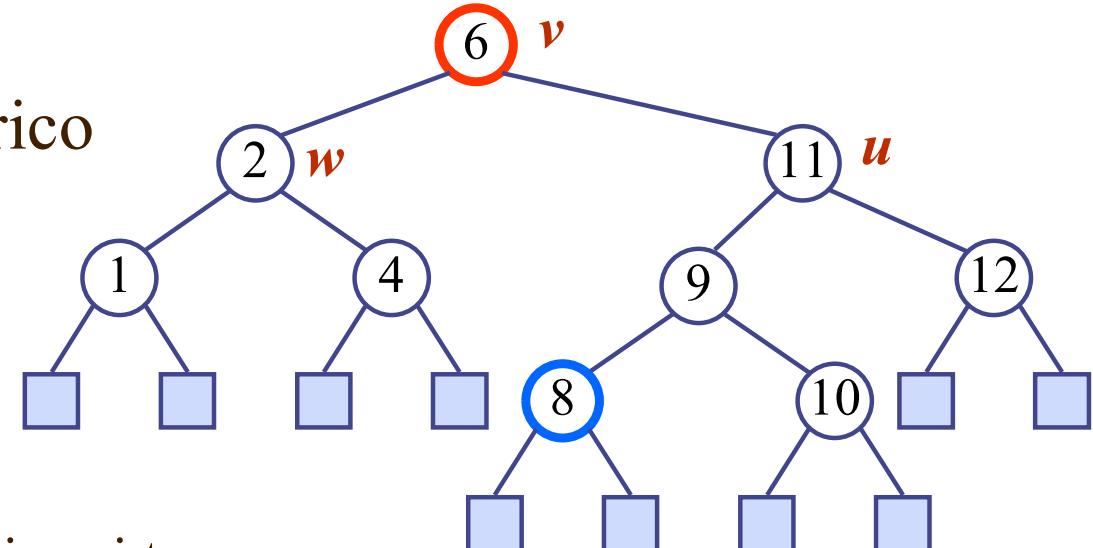


- Cioè il valore **precedente o successivo** a 6 nell’ordine indotto dall’attraversamento in ordine simmetrico del BST
 - Dato che v ha come figli due nodi interni, certamente esiste sia un valore precedente sia un valore successivo
- Poniamo la nostra attenzione, ad esempio, sul **successivo** (ma si può analogamente usare il precedente)

SortedMap in BST: rimozione

□ Continuiamo la rimozione di **6** ...

- Dove si trova il **successivo** di un particolare nodo v nell'attraversamento simmetrico di un BST ?
- **È il nodo più a sinistra nel sottoalbero destro di v !**



- Infatti, tutti i nodi che seguono v nell'attraversamento simmetrico si trovano nel suo sottoalbero destro (che sappiamo **non** essere costituito da una sola foglia "vuota", altrimenti non avremmo dovuto porci il problema...), quindi lì sarà anche il successivo di v
- In ogni sottoalbero di un BST (che è, a sua volta, un BST), scendendo verso sinistra trovo valori via via decrescenti, scendendo verso destra trovo valori crescenti: il valore minimo in un sottoalbero è, quindi, contenuto nel suo nodo più a sinistra, quello che si raggiunge **scendendo sempre a sinistra**

SortedMap in BST: rimozione

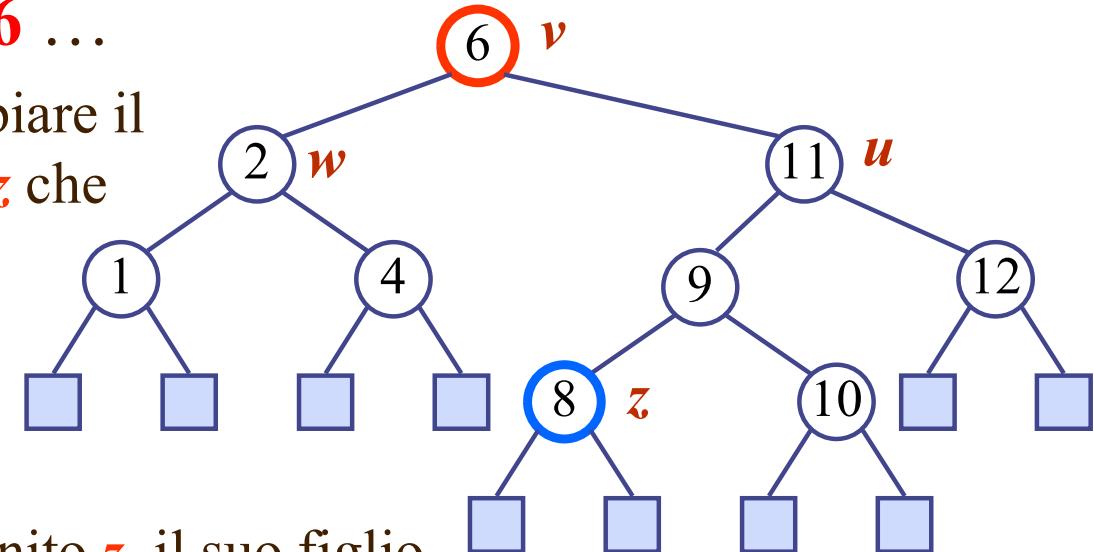
Continuiamo la rimozione di 6 ...

- Ci poniamo l'obiettivo di scambiare il nodo da rimuovere con il nodo z che si trova più a sinistra nel suo sottoalbero destro (8)

- Ma z non avrà come figli due nodi interni??

Certamente no! Per come è definito z , il suo figlio sinistro è certamente un nodo esterno, altrimenti saremmo scesi a sinistra di un altro passo (può, però, avere un nodo interno come figlio destro, ma avrà chiave maggiore, quindi non sarà il minimo del sottoalbero)

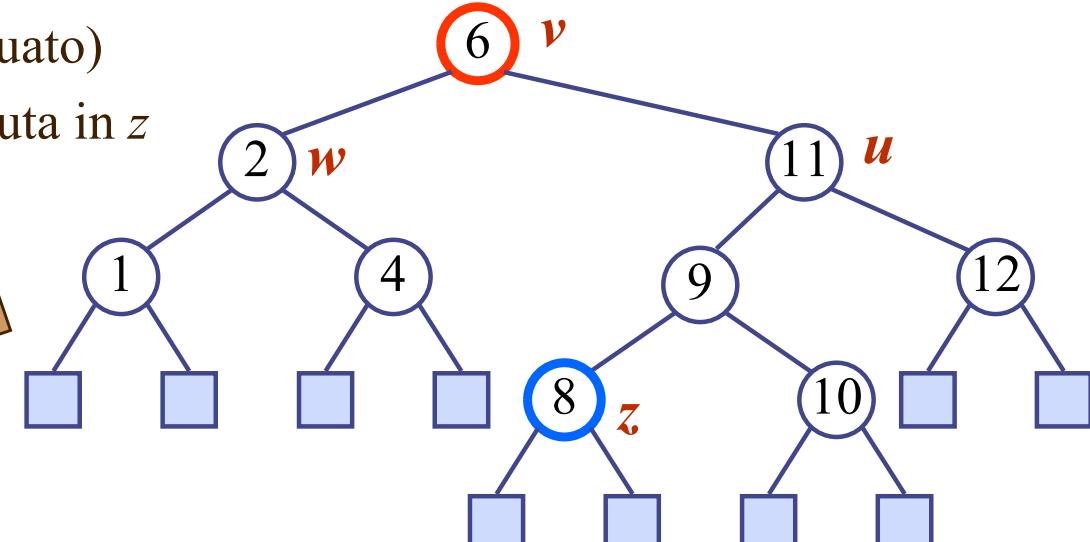
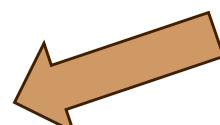
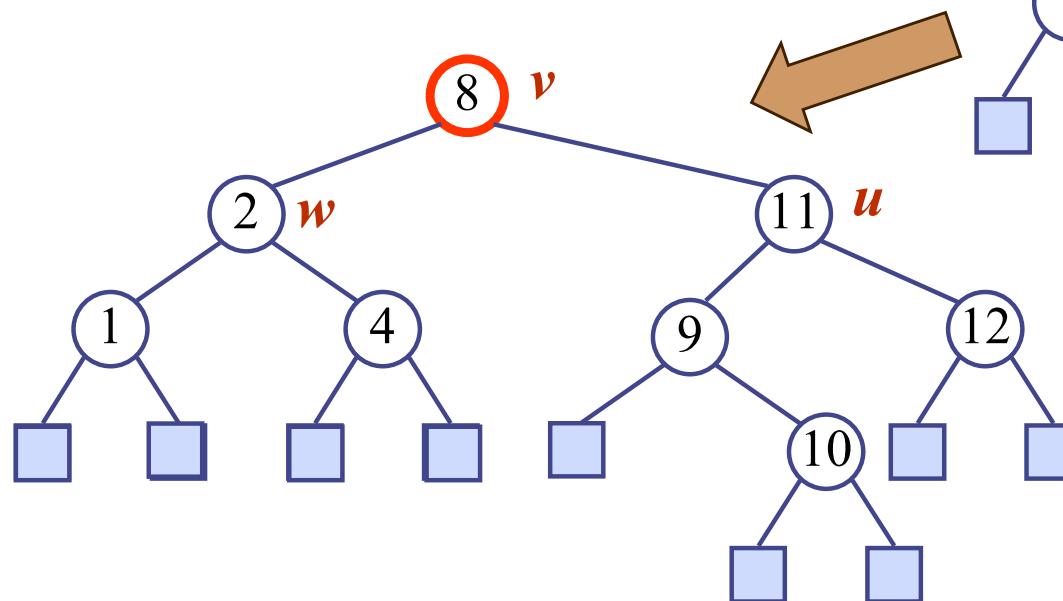
- Allora
 - Rimuovo z usando l'algoritmo visto nei due casi precedenti
 - Rimozione di un nodo avente uno o due nodi esterni come figli
 - Copio in v la coppia che era contenuta in z e ho finito!



SortedMap in BST: rimozione

- Se v ha **come figli due nodi interni**, w e u

- Sia z il nodo più a sinistra del sottoalbero destro di v
- Rimuovo z (usando algoritmo adeguato)
- Copio in v la coppia che era contenuta in z
- Ho finito!



Il BST rimane un BST

Dopo aver trovato v in un tempo $O(h)$, si fanno operazioni che sono tutte $\Theta(1)$, **tranne la ricerca di z**

SortedMap in BST: rimozione

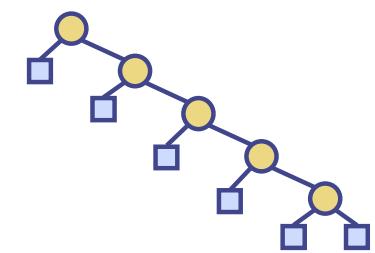
- Qual è la complessità temporale asintotica della **ricerca dell'elemento minimo in un (sotto)albero binario di ricerca?**
- Basta partire dalla radice del (sotto)albero e scendere sempre verso sinistra, fin quando è possibile (cioè fin quando il figlio sinistro non è una foglia)
- Al massimo, si scende $h_s - 1$ volte, quindi la ricerca dell'elemento minimo in un BST è $O(h_s)$, essendo h_s l'altezza del (sotto)albero; ovviamente $O(h_s) \subseteq O(h)$
- Riassumendo
 - L'eliminazione da un BST di una coppia avente una determinata chiave, usando l'algoritmo appena visto, è $O(h)$ nel caso pessimo, essendo h l'altezza dell'albero

SortedMap in BST: prestazioni

- Abbiamo, quindi, individuato algoritmi che ci consentono di realizzare i metodi **get**, **put** e **remove** di una mappa ordinata usando un BST e garantendo prestazioni **$O(h)$** nel caso pessimo, **con h , in generale, non minima**
 - A questo punto, si potrebbe (**molto faticosamente**) **dimostrare** che, operando su una tale mappa con una serie **casuale** di inserimenti e rimozioni, **l'altezza media** del BST è **$\Theta(\log n)$** , essendo n la dimensione della mappa
 - La dimostrazione è "facoltativa" anche sul Cormen... ☺
 - Questa realizzazione di mappa ordinata ha, quindi, **tutti** i metodi che sono $O(h)$, cioè:
 - **$O(\log n)$** in media **[BUONO]** **[non abbiamo dimostrato]**
 - **$O(n)$** nel caso pessimo **[PURTROPPO]**
 - **Caso medio:** **meglio di un array ordinato** (per inserimento e rimozione)
 - **Caso pessimo:** **peggio di un array ordinato** (per la ricerca)

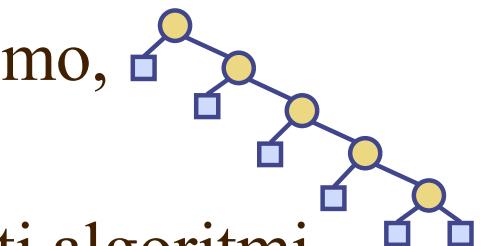
SortedMap in BST: prestazioni

- Abbiamo, quindi, individuato algoritmi che ci consentono di realizzare i metodi **get**, **put** e **remove** di una mappa ordinata usando un BST e garantendo prestazioni **$O(h)$** nel caso pessimo, **con h , in generale, non minima**
 - A questo punto, si potrebbe (molto faticosamente) **dimostrare** che, operando su una tale mappa con una serie **casuale** di inserimenti e rimozioni, **l'altezza media** del BST ha un andamento **$\Theta(\log n)$** , essendo n la dimensione della mappa
 - Non facciamo la dimostrazione perché...
 - **Useremo algoritmi migliori! Tutti i metodi saranno $O(\log n)$ nel caso pessimo**
 - Attenzione: **è facile cadere nel caso pessimo** per l'altezza del BST
 - Es: Inserendo chiavi in ordine crescente (o decrescente)... si ottiene un "millepiedi"



SortedMap in BST: prestazioni

- Abbiamo visto due soluzioni diverse
 - BST di **altezza minima** (è come un array ordinato)
 - Metodo **get** $O(\log n)$ nel caso medio e pessimo
 - Metodi **put** e **remove** $O(n)$ nel caso medio e pessimo
(ricostruiscono l'intero albero!!)
 - BST di **altezza non minima**, $h \in O(n)$, $h \in \Omega(\log n)$
 - Metodi **get**, **put** e **remove** $O(h)$ nel caso pessimo, quindi $O(n)$ (es. inserendo in ordine crescente)
 - Si può (**faticosamente**) dimostrare che, con questi algoritmi di inserimento e rimozione, $h \in \Theta(\log n)$ **nel caso medio**
 - Cerchiamo un BST in cui tutti i metodi siano $O(\log n)$ nel caso pessimo!
- Useremo un BST di altezza "quasi minima"...**



Alberi AVL

Albero AVL

- Un nodo interno v di un albero binario **proprio** si dice **bilanciato in altezza** (*height-balanced*, diremo semplicemente "bilanciato") se le altezze dei suoi (due) figli differiscono tra loro al massimo di un'unità (altrimenti si dice **sbilanciato**)
 - Facendo "degenerare coerentemente" la proprietà, Ha due sottoalberi "quasi" alti uguali possiamo definire bilanciati anche tutti i nodi esterni (non hanno figli...)
- Un albero binario con **tutti i nodi** (interni) **bilanciati** si dice "bilanciato" (è una proprietà strutturale, non dipende dai valori)
 - Ogni nodo interno ha due sottoalberi che sono "quasi" alti uguali
- Un **BST bilanciato** si dice "**albero AVL**" o semplicemente **AVL**
 - Dal nome dei suoi inventori (1962, 60 anni fa!!!!), G.M. **Adel'son-Vel'sky** e E.M. **Landis**
 - Pronunciandolo un po' in fretta sembra "**evil**" ☺ infatti è diabolico...

SortedMap in albero AVL

□ Proprietà (ovvia)

- Un albero binario proprio T è un BST se e solo se tutti i suoi sottoalberi sono BST
 - Dimostrazione "se": Tra i sottoalberi di T c'è T stesso, quindi se tutti i sottoalberi di T sono BST, allora T è un BST
 - Dimostrazione "solo se": Sia S un sottoalbero di T la cui radice r non soddisfa la proprietà dei BST (e, quindi, S non è un BST). In tal caso nemmeno T è un BST, perché r appartiene anche a T

□ Proprietà (analogia)

- Un albero binario proprio è bilanciato se e solo se tutti i suoi sottoalberi sono bilanciati

□ Proprietà (deriva dalle due precedenti)

- Un albero binario proprio è un AVL se e solo se tutti i suoi sottoalberi sono AVL

SortedMap in albero AVL

□ **Proprietà** – L'altezza h di un AVL di dimensione n è $O(\log n)$

- **Segue dimostrazione**
- **Attenzione: non è detto che l'altezza sia minima, ma "quasi"...**
potrebbe ad esempio essere $h = 2 \lfloor \log_2 n \rfloor$ oppure $h = 3 \lceil \log_2 n \rceil + 1$...
- Ovviamente ricordiamo che in un albero binario proprio $\mathbf{h \leq (n - 1)/2}$, ma questo non ci aiuta... il risultato sarà più stringente per h , perché non tutti gli alberi binari propri sono bilanciati (il "millepiedi", che corrisponde al limite $h = (n - 1)/2$, NON è bilanciato quando $n > 5$)

□ **Conseguenza** – Se si realizza una mappa ordinata con un AVL e **si riescono a progettare operazioni `put` e `remove` che mantengano l'albero BILANCIATO**, si ottiene **get** con prestazioni $O(\log n)$ nel caso pessimo

- Se poi nel caso pessimo anche **put** e **remove** fossero $O(h)$ (cioè $O(\log n)$) ...

Proprietà di un AVL

AVL: BST bilanciato

□ **Tesi** – L'altezza h di un AVL di dimensione n è $O(\log n)$

- Lavoriamo sul problema "inverso": cerchiamo un limite inferiore al numero di nodi, $n(h)$, in un AVL di altezza h
- Qual è il numero **minimo**, $n_{\min}(h)$, di nodi in un AVL di altezza h ? [ricordiamo che un BST è proprio e non vuoto]
- $h = 0 \Rightarrow n_{\min}(0) = 1$ [$=n(0)$, non ci sono altri alberi possibili]
 - ha solo la radice
- $h = 1 \Rightarrow n_{\min}(1) = 3$ [$=n(1)$, non ci sono altri alberi possibili]
 - $h = 1 \Rightarrow$ la radice deve avere figli; essendo un albero proprio, li deve avere entrambi
- $h = 2 \Rightarrow n_{\min}(2) = 5$ [$n_{\max}(2) = 7$, con $n(2) \in \{5, 7\}$]
 - la radice deve avere almeno un figlio interno, altrimenti l'altezza non può essere 2; tale figlio interno avrà due figli (foglie)

CONTINUA

Proprietà di un AVL

□ **Tesi** – L’altezza h di un AVL di dimensione n è $O(\log n)$

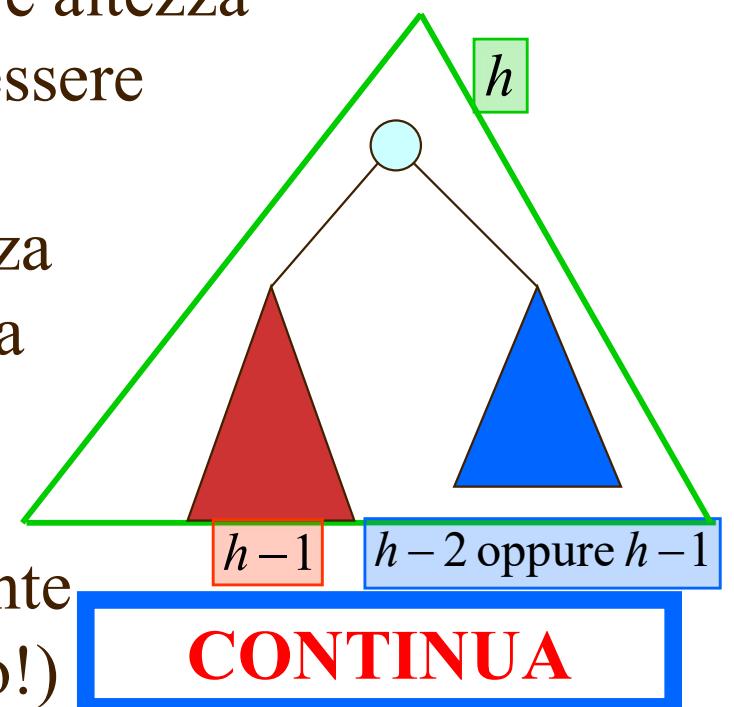
- Dimostrato: $n_{\min}(0) = 1$, $n_{\min}(1) = 3$, $n_{\min}(2) = 5$
- $h \geq 3 \Rightarrow$ l’AVL di altezza h con n minimo ha, come sottoalberi della radice, **due AVL di dimensione minima (relativamente all’altezza che hanno)**. Infatti:
 - Se uno dei sottoalberi della radice non è un AVL, l’albero non può essere un AVL (proprietà già dimostrata)
 - Se uno dei sottoalberi della radice non ha dimensione minima (relativamente all’altezza che ha), l’albero non ha dimensione minima, perché ne esiste un altro avente la stessa altezza ma con sottoalbero di dimensioni inferiori, quindi con dimensione totale dell’albero inferiore

CONTINUA

Proprietà di un AVL

□ **Tesi** – L'altezza h di un AVL di dimensione n è $O(\log n)$

- $h \geq 3 \Rightarrow$ l'AVL di altezza h con $n_{\min}(h)$ nodi ha, come sottoalberi della radice, due AVL di dimensione minima
- Almeno **uno dei due sottoalberi della radice** deve avere altezza $h - 1$, altrimenti l'intero albero non potrebbe avere altezza h , quindi tale sottoalbero ha **$n_{\min}(h - 1)$** nodi
- **L'altro sottoalbero della radice** può avere altezza inferiore a $h - 1$, ma tale altezza non può essere inferiore a $h - 2$, altrimenti la radice non sarebbe bilanciata: quindi, può avere altezza $h - 1$ oppure $h - 2$ e, conseguentemente, ha un numero di nodi uguale a **$\min \{n_{\min}(h - 1), n_{\min}(h - 2)\}$** (si può intuire che $n_{\min}(h)$ sia una funzione crescente ma è meglio dimostrarlo... lo dimostriamo!)

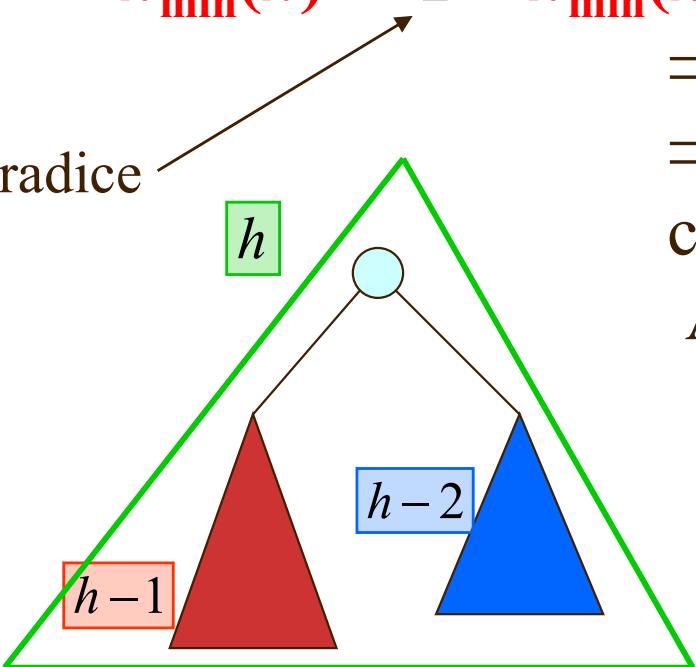


Proprietà di un AVL

- **Tesi** – L’altezza h di un AVL di dimensione n è $O(\log n)$
- $h \geq 3 \Rightarrow$ l’AVL di altezza h con $n_{\min}(h)$ nodi ha, come sottoalberi della radice, due AVL di dimensione minima
 - Uno dei due sottoalberi della radice ha $n_{\min}(h - 1)$ nodi
 - L’altro sottoalbero ha $\min \{n_{\min}(h - 1), n_{\min}(h - 2)\}$ nodi
 - Quindi, $\forall h \geq 3$,
$$n_{\min}(h) = 1 + n_{\min}(h - 1) + \boxed{\min \{n_{\min}(h - 1), n_{\min}(h - 2)\}}$$

quantità > 0

radice


$$\Rightarrow n_{\min}(h) > n_{\min}(h - 1)$$
$$\Rightarrow n_{\min}(h) \text{ è una funzione crescente di } h,$$

come si poteva facilmente intuire, per cui un AVL di altezza $h \geq 3$ avente **dimensione minima** ha un sottoalbero di altezza $h - 1$ e uno di altezza $h - 2$, entrambi di dimensione minima

CONTINUA

Proprietà di un AVL

□ **Tesi** – L'altezza h di un AVL di dimensione n è $O(\log n)$

- $\forall h \geq 3, n_{\min}(h) = 1 + n_{\min}(h - 1) + n_{\min}(h - 2)$
- $n_{\min}(h)$ è una funzione crescente, quindi $n_{\min}(h - 1) > n_{\min}(h - 2)$
 - $n_{\min}(h) = 1 + n_{\min}(h - 1) + n_{\min}(h - 2)$
 $> 2 n_{\min}(h - 2)$
 - $n_{\min}(h) > 2 n_{\min}(h - 2)$
 $> 4 n_{\min}(h - 4)$
 $> 8 n_{\min}(h - 6)$
...
 $> 2^i n_{\min}(h - 2i) \quad \forall i \in \mathbb{N} \mid h - 2i \geq 0$

n_{\min} “più che raddoppia” ogni volta che h aumenta di 2, quindi anche

$$n_{\min}(h - 2) > 2 n_{\min}(h - 4)$$
$$n_{\min}(h - 4) > 2 n_{\min}(h - 6)$$

e così via

- Ricordiamo che $n_{\min}(1) = 3$ e che $n_{\min}(2) = 5$

CONTINUA

Proprietà di un AVL

□ **Tesi** – L'altezza h di un AVL di dimensione n è $O(\log n)$

- $n_{\min}(1) = 3, n_{\min}(2) = 5, n_{\min}(h) > 2^i n_{\min}(h - 2i) \quad \forall h \geq 3$
- Se h è dispari ($h \geq 3$),
 $\exists i = (h - 1)/2 \in \mathbb{N} \mid h - 2i = 1 \Rightarrow n_{\min}(h) > 2^{(h-1)/2} n_{\min}(1)$
 $\Rightarrow n_{\min}(h) > 3 \cdot 2^{(h-1)/2} \Rightarrow \log_2(n_{\min}(h)/3) > (h-1)/2$
 $\Rightarrow h < 1 + 2 \log_2(n_{\min}(h)/3) \leq 1 + 2 \log_2(n(h)/3)$
 $\Rightarrow h < 1 + 2 \log_2 n(h) \Rightarrow h \in O(\log n)$
- Se h è pari ($h > 3$)
 $\exists i = h/2 - 1 \in \mathbb{N} \mid h - 2i = 2 \Rightarrow n_{\min}(h) > 2^{h/2-1} n_{\min}(2)$
 $\Rightarrow n_{\min}(h) > 5 \cdot 2^{h/2-1} \Rightarrow \log_2(n_{\min}(h)/5) > h/2 - 1$
 $\Rightarrow h < 2 + 2 \log_2(n_{\min}(h)/5) \leq 2 + 2 \log_2(n(h)/5)$
 $\Rightarrow h < 2 + 2 \log_2 n(h) \Rightarrow h \in O(\log n)$

Il numero minimo, $n_{\min}(h)$, di nodi di un AVL di altezza h è certamente non maggiore della dimensione, $n(h)$, di qualsiasi AVL di altezza h

Proprietà di un AVL

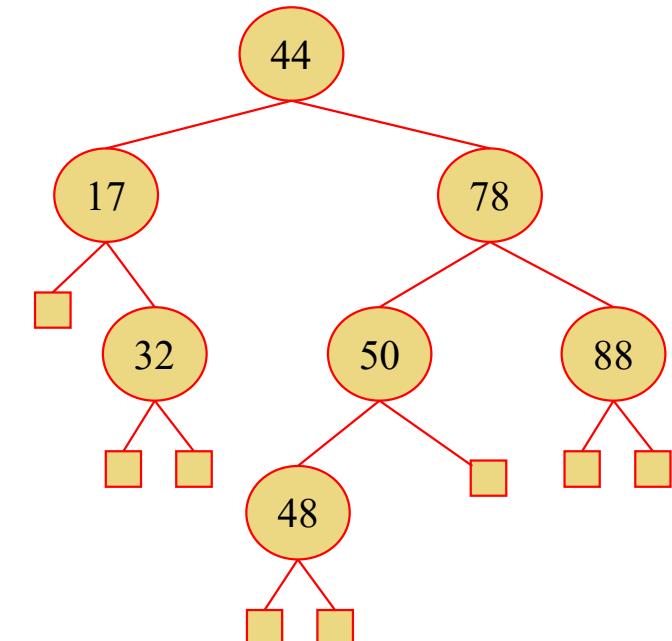
□ Proprietà (appena dimostrata)

L'altezza h di un AVL di dimensione n è $O(\log n)$, in particolare abbiamo anche dimostrato che $h < 2 + 2 \log_2(n) \forall h \geq 0$

- In realtà, abbiamo anche dimostrato un limite superiore più stringente (anche se non asintoticamente): $h < 2 + 2 \log_2(n/3) \forall h \geq 1$

□ Attenzione NON è come dire “l'altezza di un AVL è minima”

- L'altezza minima di un albero binario proprio è $h = \lfloor \log_2 n \rfloor$
- Infatti, questo è un AVL ma **NON** ha altezza minima
 - Però è “**QUASI MINIMA**”... non può differire “molto” da quella minima... al massimo è (circa) il **doppio** di quella minima



Lezione 28

SortedMap in albero AVL

- Abbiamo, quindi, dimostrato che se si realizza una mappa ordinata con un AVL e **si riescono a progettare operazioni `put` e `remove` che mantengano l'albero BILANCIATO**, si ottiene `get` con prestazioni $O(h) \equiv O(\log n)$ nel caso pessimo

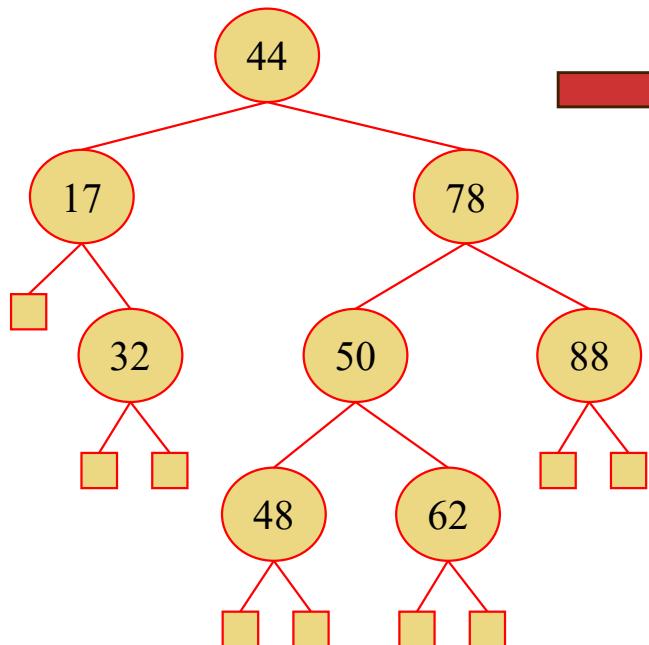
- Molto migliore di BST generico (dove `get` è $O(n)$ nel caso pessimo), ma ancora uguale a un array ordinato
- Se, però, anche `put` e `remove` fossero $O(h) \equiv O(\log n)$ nel caso pessimo, l'AVL sarebbe migliore di un array ordinato!
Dove tali metodi sono $O(n)$ nel caso pessimo
- Dedichiamoci ai metodi `put` e `remove`
Vogliamo progettarli in modo che
 - Mantengano l'albero **BILANCIATO**
 - Siano $O(h) \equiv O(\log n)$ nel caso peggiore

Perché non usiamo un albero di altezza **minima**, ad esempio completo?
Perché non si riescono a progettare metodi `put` e `remove` logaritmici!

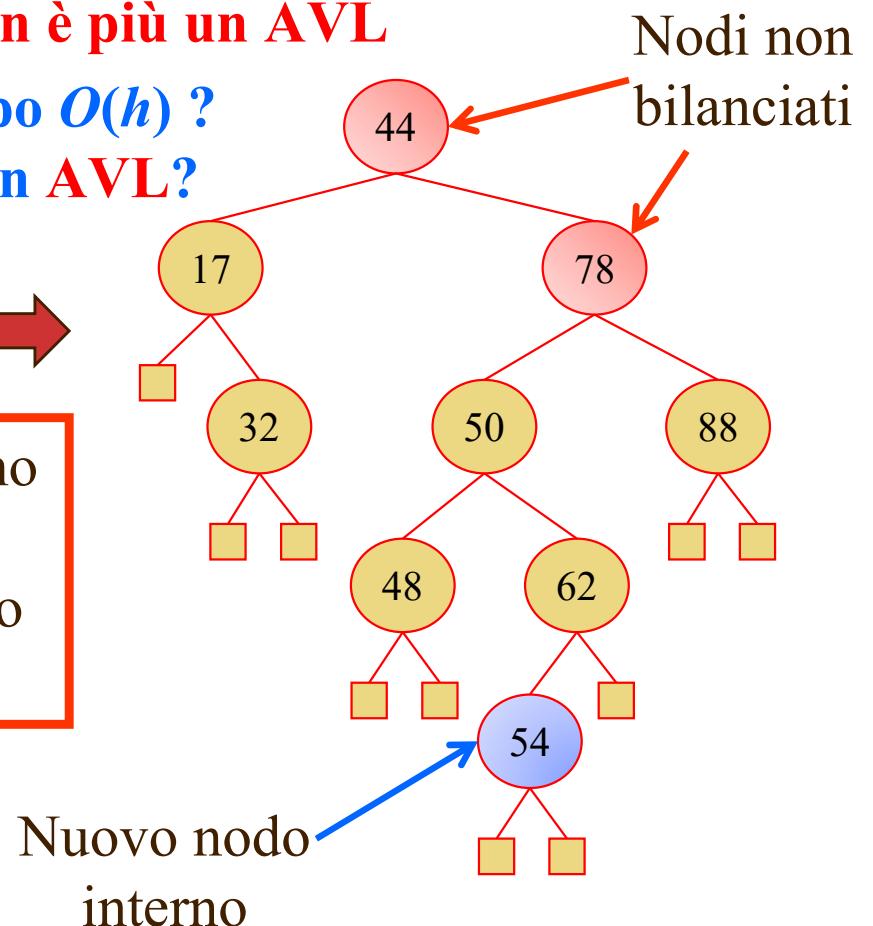
Non è semplice

Inserimento in AVL

- Il metodo **put** che inserisce una nuova coppia in una mappa realizzata con AVL esegue, per prima cosa, lo stesso algoritmo visto per la mappa realizzata con BST "normale"
 - Sappiamo che questa operazione è $O(h)$, cioè, essendo bilanciato, $O(\log n)$
 - In generale, però, **il BST risultante non è più un AVL**
 - Riusciamo a "sistemarlo" in un tempo $O(h)$?**
Cioè a farlo diventare nuovamente un AVL?



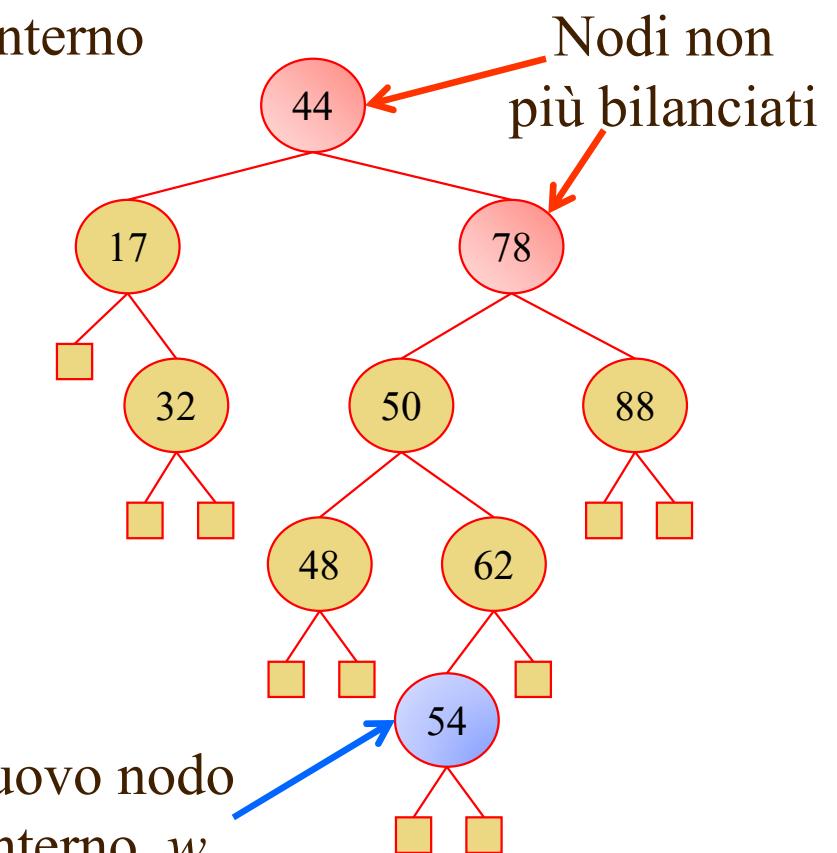
Non sempre ci sono problemi, ad esempio inserendo **15 e/o 80...**



Inserimento in AVL

- Per prima cosa vediamo se in un tempo $O(h)$ riusciamo almeno a **verificare** se l'albero AVL è rimasto un AVL dopo l'inserimento oppure no!
- Ricordiamo che la proprietà di bilanciamento di un nodo dipende SOLTANTO dalle altezze dei suoi figli: se queste non sono cambiate, il nodo (che era bilanciato per ipotesi) non può essersi sbilanciato
- L'altezza di un nodo dipende soltanto dal sottoalbero che contiene i suoi discendenti, quindi **gli unici nodi che possono cambiare altezza sono gli antenati della foglia w trasformata** in nodo interno
- Di conseguenza, **gli unici nodi che possono sbilanciarsi sono**, di nuovo, **gli antenati di w**
- Il numero massimo di tali nodi è $O(h)$, quindi perché l'intera operazione di verifica sia $O(h)$, occorre che la verifica di bilanciamento di ciascun nodo sia $\Theta(1)$

Perturbazione iniziale delle
altezze... l'altezza di w
passa da 0 a 1



Inserimento in AVL

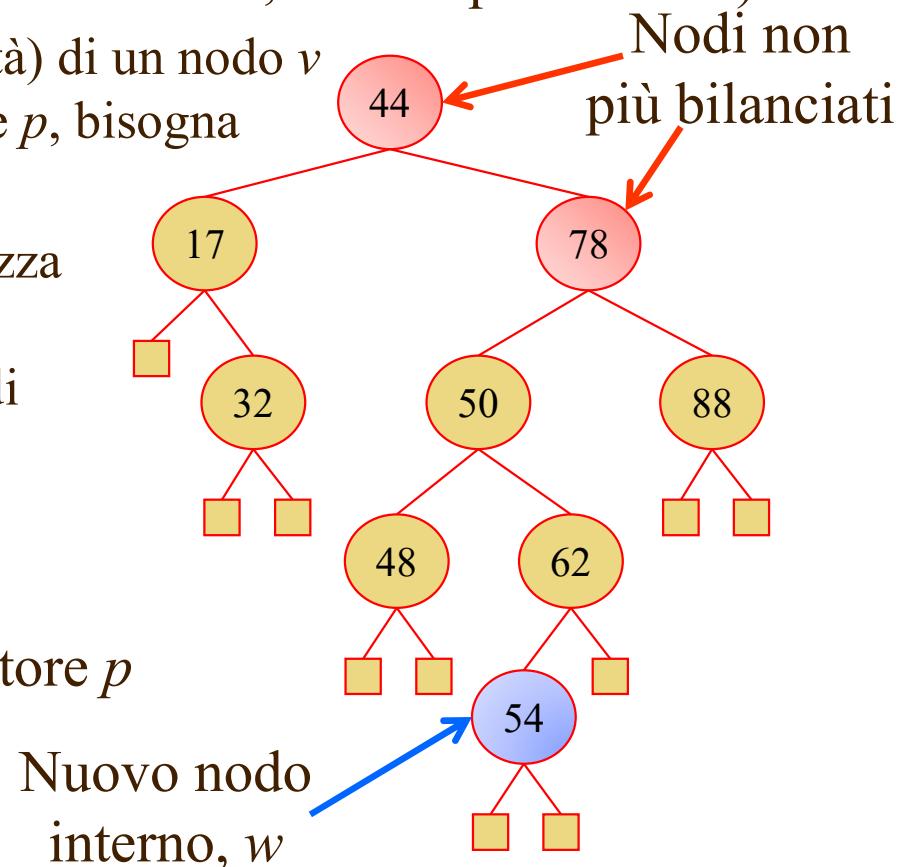
- Per prima cosa vediamo se in un tempo $O(h)$ riusciamo a verificare se l'albero AVL è rimasto un AVL dopo l'inserimento oppure no!
- **Dobbiamo verificare il bilanciamento di ogni antenato di w**

- In realtà **è sufficiente interrompere la verifica non appena si trova un nodo sbilanciato**, perché questo rende sbilanciato l'albero senza bisogno di ulteriori verifiche

- La procedura di verifica risale da w lungo il percorso minimo verso la radice propagando la variazione di altezza (l'altezza di w è ora 1, mentre prima era 0)

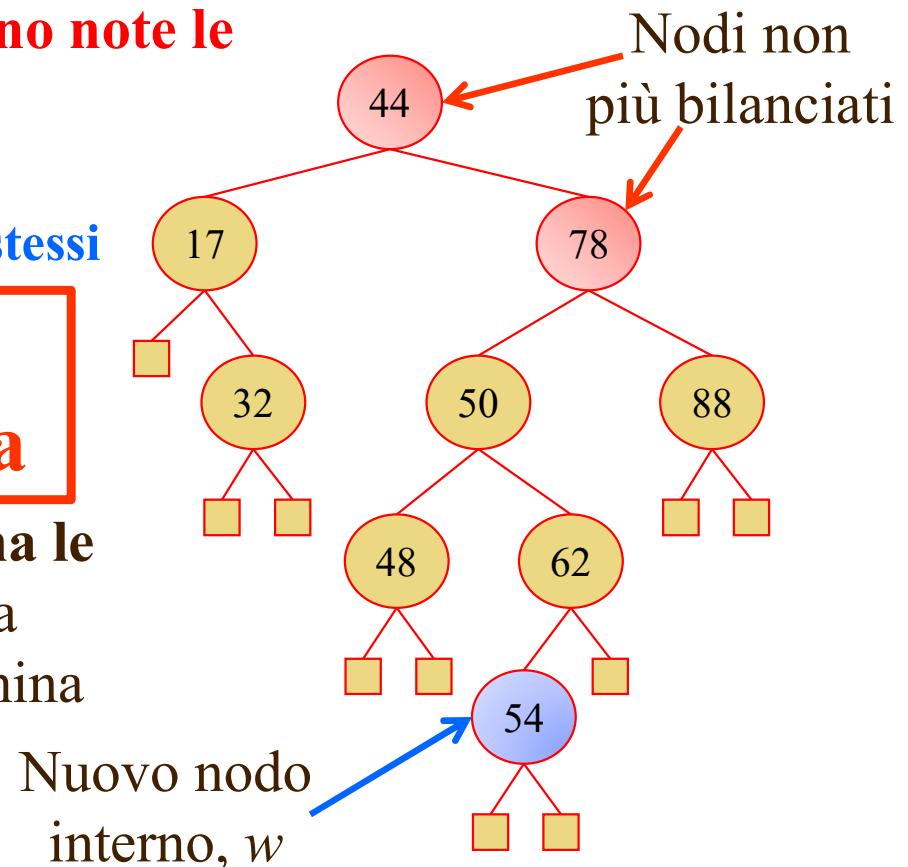
- Per decidere se l'aumento di altezza (di un'unità) di un nodo v provoca l'aumento dell'altezza del suo genitore p , bisogna conoscere l'altezza del fratello di v
- Non appena si osserva che la variazione di altezza non si propaga, la verifica si può interrompere: l'albero è rimasto bilanciato (nessun antenato di profondità inferiore potrà cambiare altezza, né sbilanciarsi)

- Ogni volta che un nodo v aumenta la propria altezza, bisogna poi controllare che il suo genitore p non si sbilanci: per farlo, occorre conoscere l'altezza del fratello di v



Inserimento in AVL

- Per prima cosa vediamo se in un tempo $O(h)$ riusciamo a verificare se l'albero AVL è rimasto un AVL dopo l'inserimento oppure no!
- Dobbiamo verificare il bilanciamento di ogni antenato di w
 - In realtà è sufficiente interrompere la verifica non appena si trova un nodo sbilanciato, perché questo rende sbilanciato l'albero senza bisogno di ulteriori verifiche
- Quindi, durante la propagazione verso l'alto della variazione dell'altezza degli antenati di w , e le conseguenti verifiche di bilanciamento, le operazioni richieste per ogni nodo richiedono un tempo $\Theta(1)$ **se sono note le altezze dei fratelli dei nodi in esame**
 - Il calcolo di queste altezze è troppo oneroso, quindi devono essere memorizzate nei nodi stessi
 - **In un AVL, ogni nodo memorizza la propria altezza**
- Inoltre, questa procedura di verifica aggiorna le altezze di tutti gli antenati di w , fin quando la propagazione dell'aumento di altezza non termina
- **Tutto questo richiede un tempo $O(h) \equiv O(\log n)$**



Se il nodo z non esiste, l'albero è rimasto bilanciato (per la definizione di z) e siamo a posto

Inserimento in AVL

□ Facciamo un'analisi per localizzare i potenziali problemi

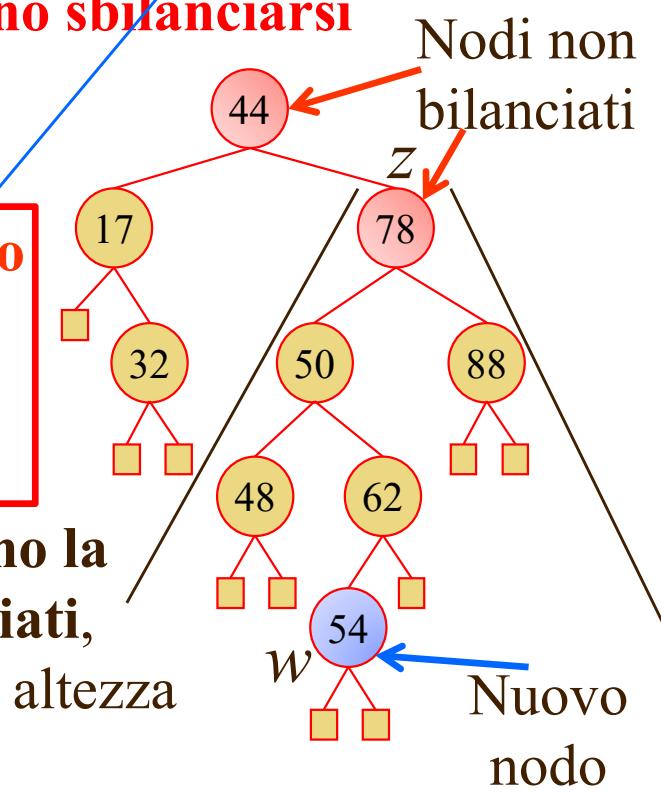
- **Riassunto:** Per effetto dell'inserimento in T , una foglia, w , diventa nodo interno e un sottoinsieme del percorso minimo che collega w alla radice aumenta la propria altezza di un'unità: tale sottoinsieme comprende almeno w e un certo numero di suoi antenati disposti **consecutivamente** lungo il percorso (al limite fino a comprendere anche la radice)

- Cambiando le altezze, **alcuni antenati di w possono sbilanciarsi**

→ Identifichiamo **il nodo z** come il primo, risalendo verso la radice, che si è sbilanciato (**SE ESISTE**)

- **Strategia:** facciamo **modifiche nel sottoalbero avente z come radice** in modo che tale sottoalbero **torni a essere bilanciato e torni ad avere la sua altezza precedente!**

- Di conseguenza, tutti gli antenati di z avranno la propria altezza precedente e saranno bilanciati, perché i loro sottoalberi non avranno cambiato altezza



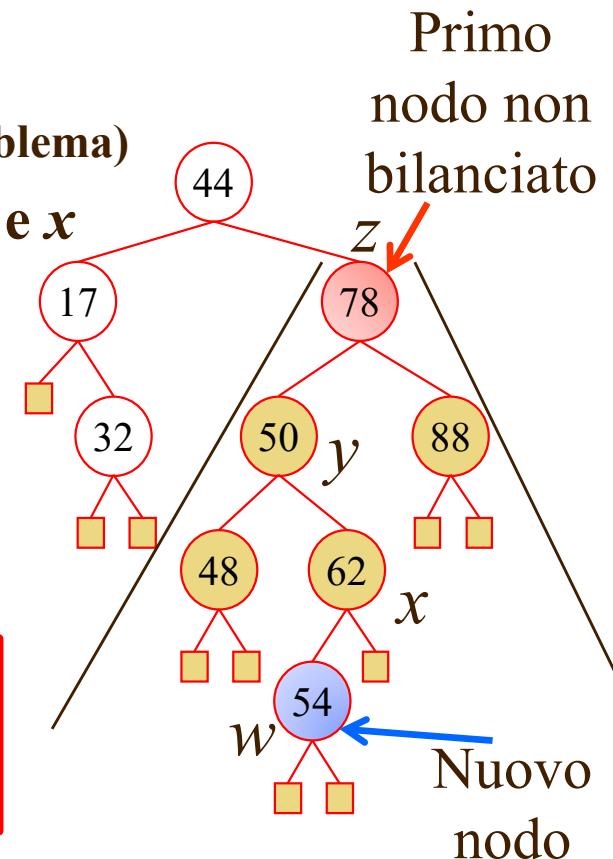
Inserimento in AVL

z può anche essere la radice dell'albero, quindi il "sottoalbero" può anche essere l'intero albero

- Abbiamo ricondotto il problema a quello del bilanciamento di un sottoalbero in cui l'unico nodo sbilanciato è la radice, *z*

- Identifichiamo altri due nodi che ci saranno “utili”
- Chiamiamo *y* il figlio di *z* avente altezza maggiore:
si dimostra che è, tra i figli di *z*, l'antenato di *w*
- Chiamiamo *x* il figlio di *y* avente altezza maggiore:
si dimostra che è, tra i figli di *y*, l'antenato di *w*
 - ATTENZIONE: *x* può coincidere con *w* (ma non è un problema)
- Si dimostra che, se *z* esiste, allora esistono anche *y* e *x*
 - In pratica, *z* non può essere "troppo vicino" a *w*: non può essere il genitore di *w*, deve essere "almeno" il nonno di *w* (quando *x* coincide con *w*) o un suo antenato più lontano
 - Se *z* non esiste, non c'è sbilanciamento ed è ancora un AVL: non servono "ristrutturazioni" dell'albero, sono sufficienti gli aggiornamenti delle altezze di (alcuni) antenati di *w*

Nel seguito le dimostrazioni qui citate,
solo per persone curiose

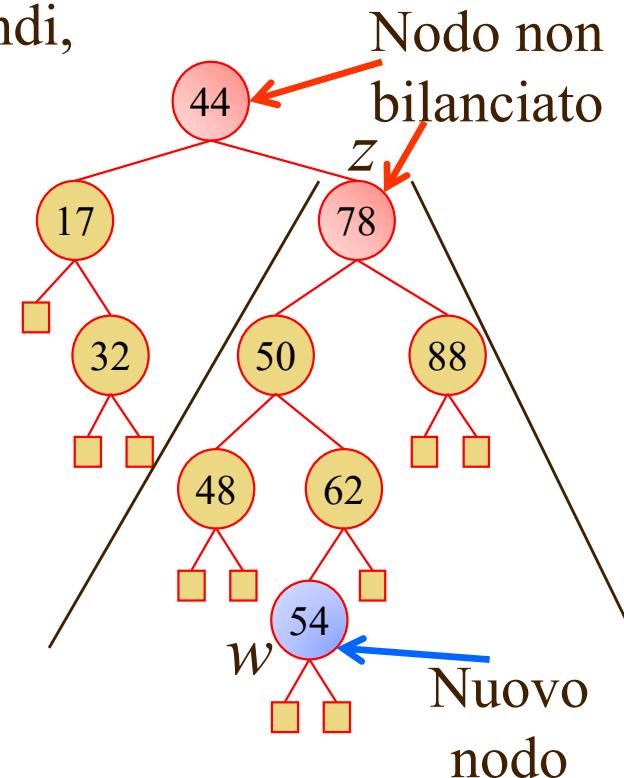


Inserimento in AVL

Dimostrazioni solo per persone curiose

□ Abbiamo ricondotto il problema a quello del bilanciamento di un sottoalbero in cui l'unico nodo sbilanciato è la radice, z

- Sappiamo anche che tale sottoalbero è “**minimamente sbilanciato**”, cioè i due sottoalberi di z hanno altezze che differiscono di 2
 - Non possono avere differenza maggiore perché prima dell'inserimento z era bilanciato e si è sbilanciato perché uno dei suoi figli ha aumentato la propria altezza **di una sola unità**
 - Proprietà (ovvia): **L'insieme degli antenati di w che hanno aumentato la propria altezza contiene z** (e, quindi, contiene tutti gli antenati di w che sono discendenti di z). Infatti:
 - z era bilanciato e non lo è più (per definizione)
 - il suo sottoalbero che **non** contiene w non ha modificato la propria altezza, quindi deve averlo fatto l'altro, che può averla solo aumentata (al massimo di uno)
 - per provocare lo sbilanciamento di z tale aumento si deve essere aggiunto a un'altezza già superiore a quella dell'altro sottoalbero, quindi aumenta l'altezza di z
 - Quindi, **non solo z si è sbilanciato, ma ha anche aumentato (di un'unità) la propria altezza**



Inserimento in AVL

Dimostrazioni solo per persone curiose

- Abbiamo ricondotto il problema a quello del bilanciamento di un sottoalbero in cui l'unico nodo sbilanciato è la radice, z , che ha anche aumentato la propria altezza

- Identifichiamo altri due nodi che ci saranno “utili”

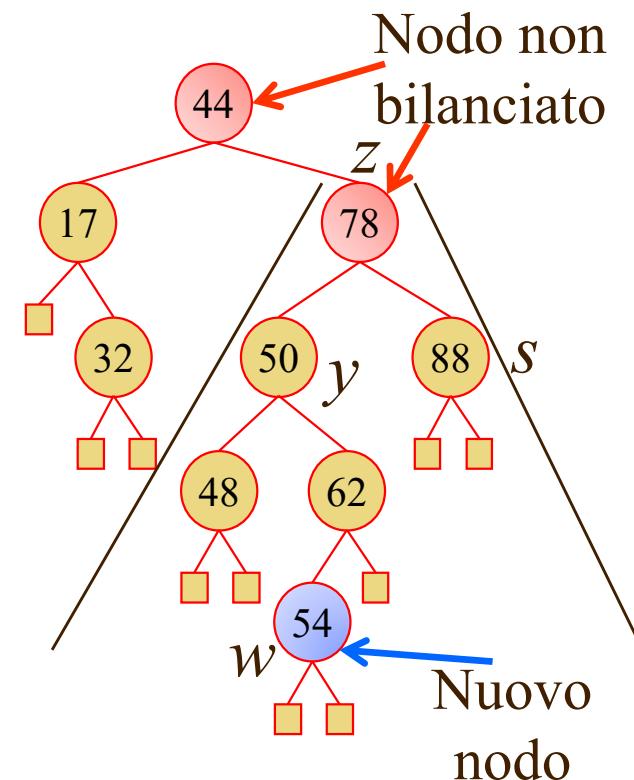
→ Chiamiamo y il figlio di z avente altezza maggiore:

è, tra i figli di z , l'antenato di w (vedi dimostrazione seguente)

- Definiamo $s = T.\text{ sibling}(y)$, cioè il fratello di y
- Da quanto detto: $h(y) = 2 + h(s)$

□ Dimostrazione (per assurdo)

- Ipotesi: $h(y) \leq h(s)$
- Se $h(y) < h(s)$, dato che l'altezza di y è aumentata (e quella di s no), la loro differenza (che ora è almeno 2 perché z è sbilanciato) ERA almeno 3, quindi z ERA sbilanciato: **assurdo**
- Se $h(y) = h(s)$, allora z è bilanciato: **assurdo**



Dimostrazioni solo per persone curiose

Inserimento in AVL

- Abbiamo ricondotto il problema a quello del bilanciamento di un sottoalbero in cui l'unico nodo sbilanciato è la radice, z , che ha anche aumentato la propria altezza

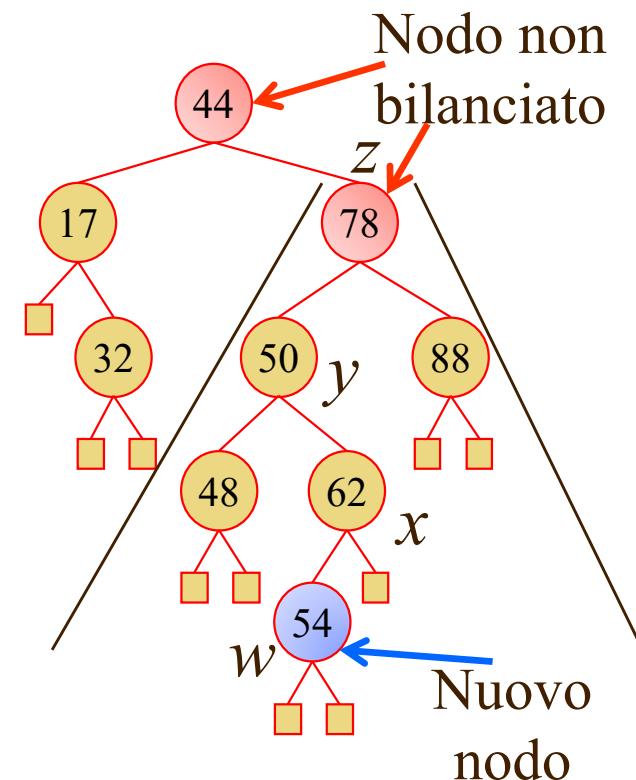
- Identifichiamo altri due nodi che ci saranno “utili”

- Chiamiamo y il figlio di z avente altezza maggiore:
è, tra i figli di z , l'antenato di w

Chiamiamo x

**il figlio di y avente altezza maggiore
(è, tra i figli di y , l'antenato di w)**

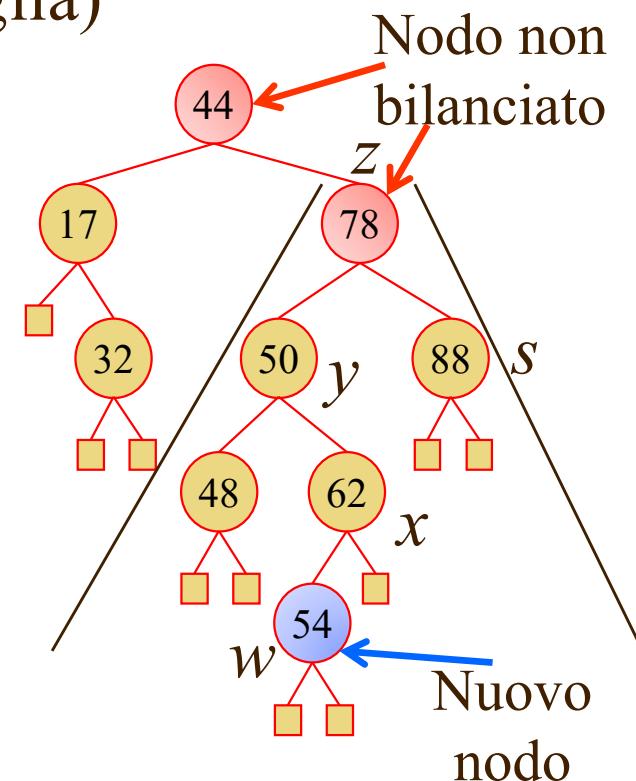
- Dimostrazione analoga a quella di y
- z è, quindi, il “nonno” di x
- ATTENZIONE: x può coincidere con w (ma non è un problema)



Inserimento in AVL

Dimostrazioni solo per persone curiose

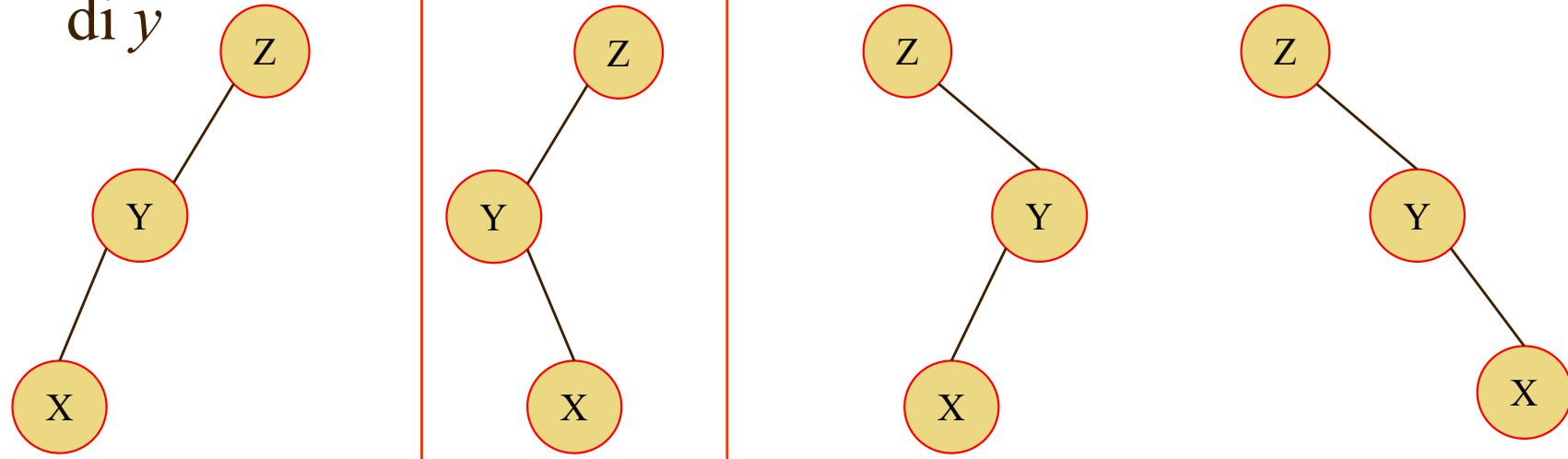
- **Tesi:** se z esiste, allora esistono anche y e x
- Se z esiste, significa che, per definizione, è sbilanciato
- Quindi l'altezza dei suoi due sottoalberi differisce almeno di 2
- Abbiamo già dimostrato che y ha altezza maggiore di suo fratello, s
- L'altezza minima di s è 0 (può essere una foglia)
- L'altezza minima di y è, quindi, 2
- Quindi y ha almeno un figlio interno, che, se è l'unico figlio interno, sarà x (se i 2 figli di y fossero foglie, y avrebbe altezza 1)
 - Se l'altezza di y è proprio uguale a 2, allora x coincide con w (ma non è un problema)



Inserimento in AVL

Il secondo caso è quello relativo all'esempio precedente

- Esistono **soltanto quattro** configurazioni possibili per i nodi x , y e z
 - y può essere il figlio sinistro o destro di z e, in ciascuno di questi due casi, x può essere il figlio sinistro o destro di y



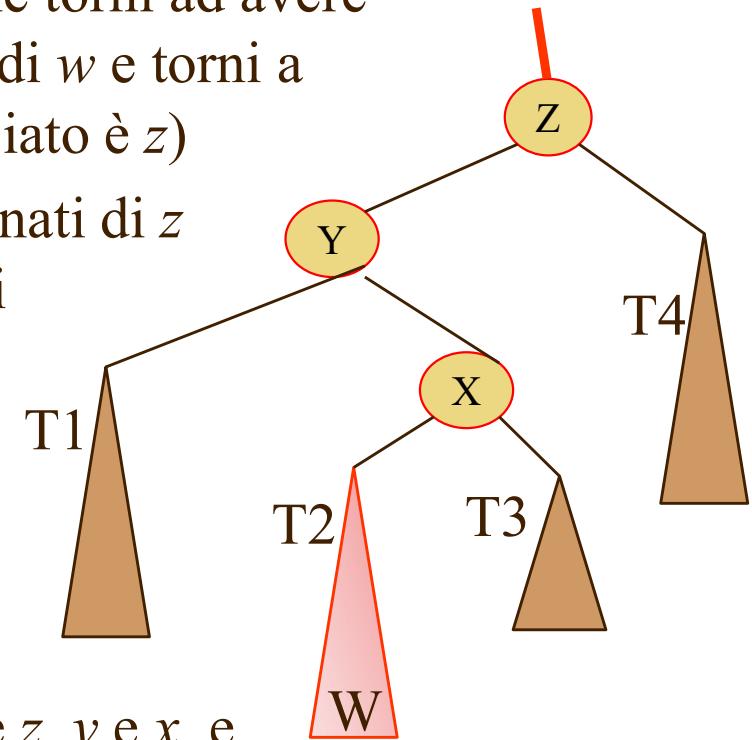
- Analizziamo in dettaglio il secondo caso, relativo all'esempio precedente, **completando l'albero con i necessari sottoalberi** (è un albero proprio), eventualmente contenenti soltanto una foglia

Inserimento in AVL

- Il nuovo nodo interno, w , appartiene a uno dei due sottoalberi di x (o coincide con x): nel nostro esempio w appartiene al sottoalbero sinistro di x
 - ATTENZIONE: i sottoalberi sono qui rappresentati da triangoli, ma non sono necessariamente triangolari!

□ Ricordiamo l'obiettivo

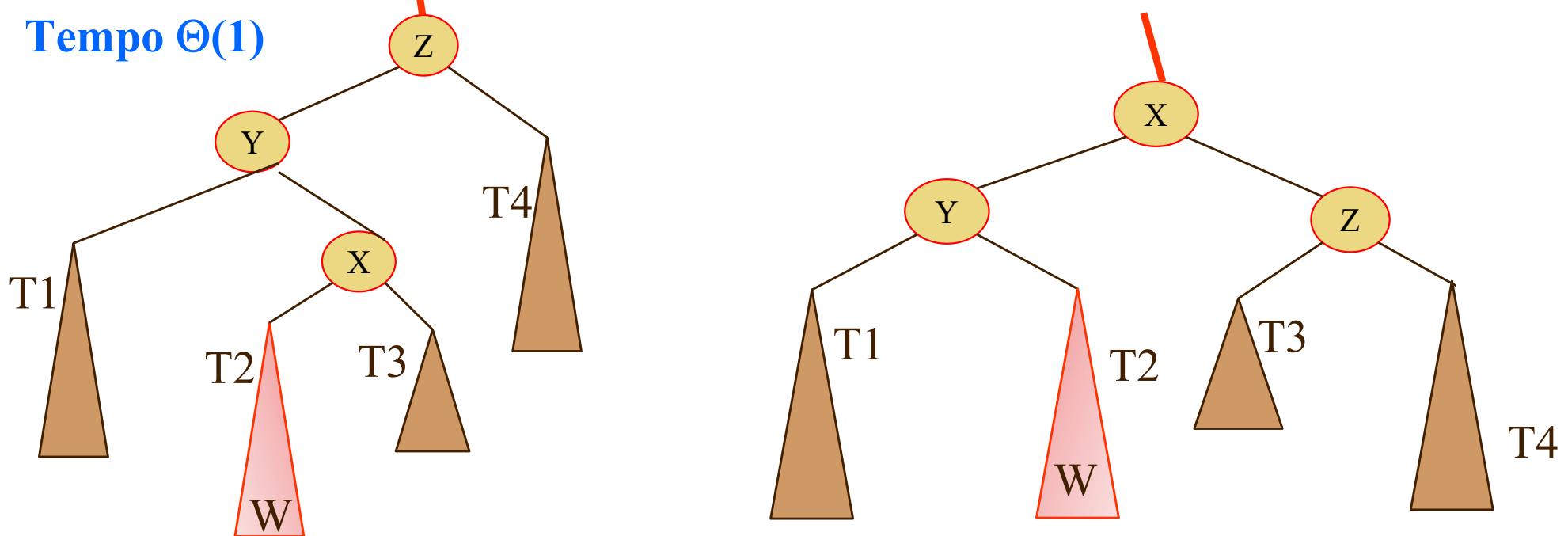
- Ristrutturare questo sottoalbero in modo che torni ad avere l'altezza che aveva prima dell'inserimento di w e torni a essere bilanciato (il suo unico nodo sbilanciato è z)
- Se riusciamo a farlo, tutti gli eventuali antenati di z non cambieranno la propria altezza e non si sbilanceranno, quindi la "perturbazione" introdotta da w si "spegnerà" all'interno di questo sottoalbero
- Questa ristrutturazione fa parte della procedura di inserimento, che ha richiesto finora un tempo $O(h)$ per trovare w , trovare z , y e x , e aggiornare le altezze tra w e z : se riusciamo a fare anche la ristrutturazione finale in un tempo $O(h)$, siamo a posto!



La ristrutturazione AVL

- Si prende, **tra i nodi x , y , e z , quello che ha l'elemento intermedio tra i tre** (può essere x o y , mai z) e lo si fa diventare radice del sottoalbero che aveva radice z , sistemandolo poi gli altri due nodi di conseguenza, come figli sinistro e destro della nuova radice (nell'unico modo possibile, in relazione al valore dei loro elementi)
- I quattro sottoalberi T^* si “attaccano” di conseguenza, nei rami liberi, rispettando il loro ordinamento relativo precedente (perché era e deve rimanere un BST) e si “correggono” coerentemente le altezze dei nodi x , y e z (conoscendo le altezze delle radici dei T^*), le uniche che sono eventualmente cambiate, oltre a quelle all'interno di T_2 , che sono già state aggiornate durante la ricerca di z

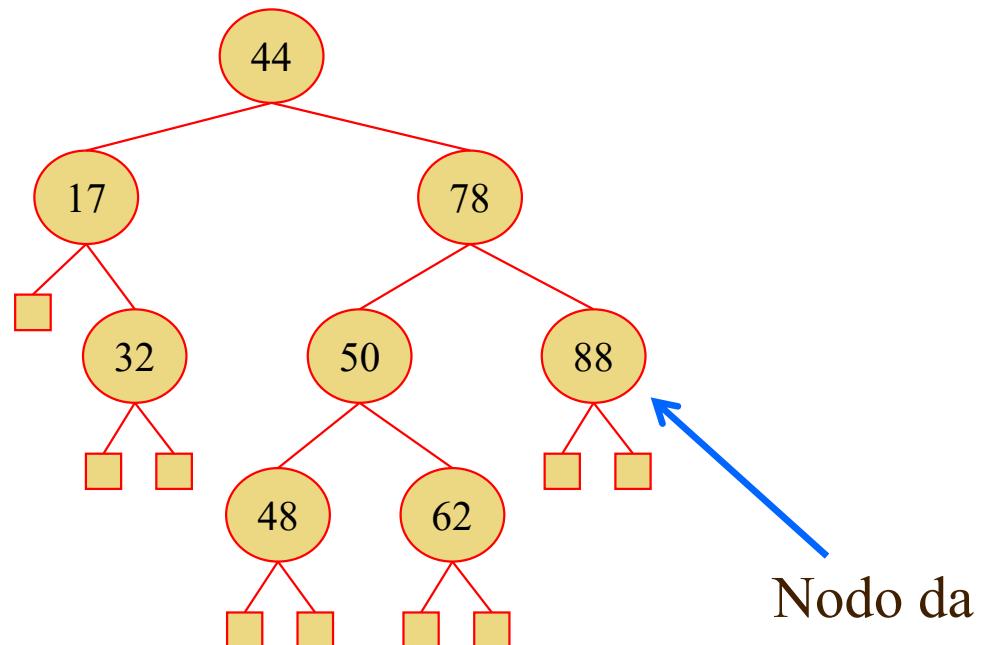
□ **Tempo $\Theta(1)$**



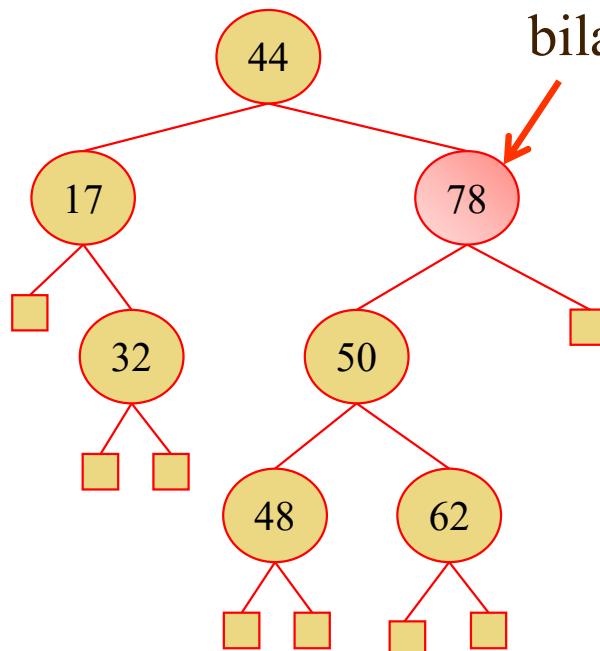
Rimozione da AVL

- Il metodo **remove** che elimina una coppia da una mappa realizzata con AVL esegue, per prima cosa, lo stesso algoritmo di rimozione visto per la mappa realizzata con BST normale
 - Con i suoi tre casi... in relazione al numero di figli del nodo rimosso

- In generale, **il BST risultante non è più un AVL**



Nodo da
eliminare



Nodo non più
bilanciato

Non sempre ci sono problemi, ad esempio eliminando **48** e/o **62**...

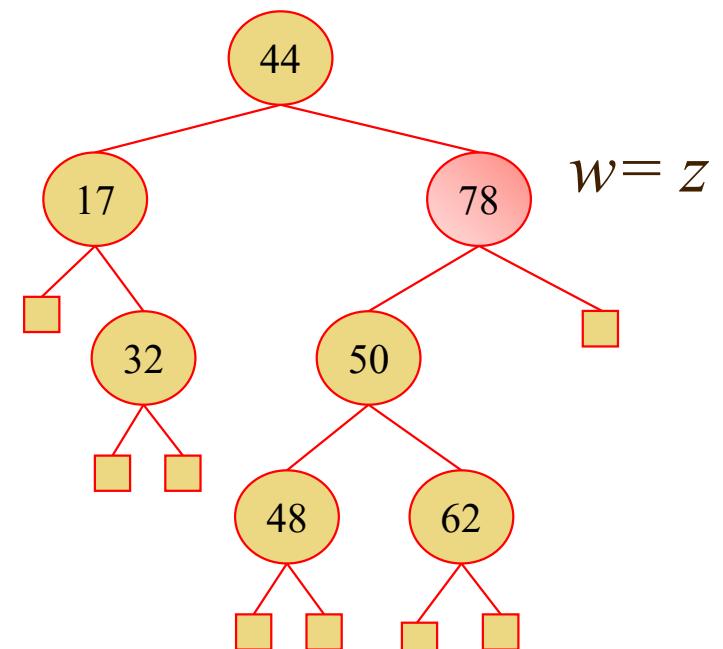
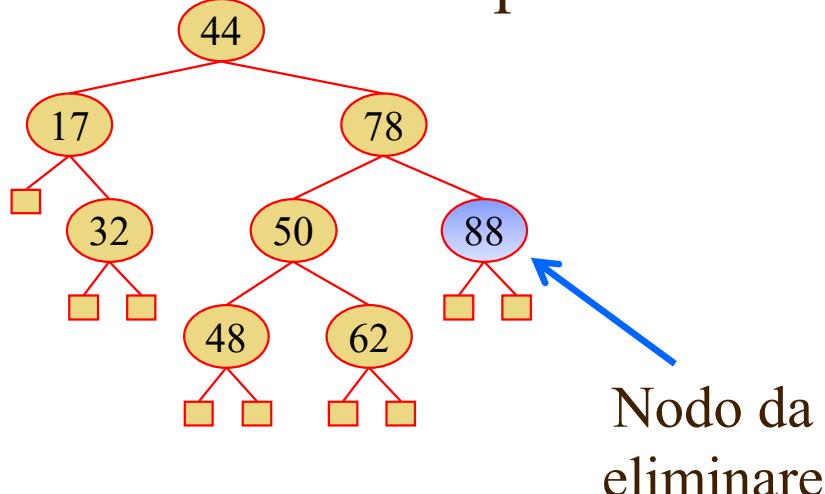
Rimozione da AVL

□ Facciamo **un'analisi per localizzare i potenziali problemi**

- L'analisi è simile a quella vista per l'inserimento, evidenziamo le differenze
- Sia w **il genitore** del nodo rimosso (nell'inserimento, w era il nodo inserito, non il suo genitore)
- Se w non esiste, significa che il nodo eliminato era la radice dell'albero e ci possono essere due casi
 - Se l'albero rimasto è una mappa vuota (cioè ha una foglia come radice), è bilanciato e non sono necessarie operazioni di sistemazione (la radice aveva due foglie come figli)
 - Altrimenti, la radice aveva come figli una foglia e un nodo interno (che è diventato la nuova radice): il fatto che fosse bilanciato implica che sia ora bilanciato (era un sottoalbero di un AVL...)
 - (se la radice ha due figli interni, non può essere eliminata, ricordare l'algoritmo)

Rimozione da AVL

- Sia w il genitore del nodo rimosso
 - Alcuni antenati (consecutivi) di w (compreso w stesso) possono diminuire la propria altezza di un'unità e alcuni di essi possono sbilanciarsi
- Sia (come nell'inserimento) z il primo nodo sbilanciato risalendo da w verso la radice
 - Se z non esiste, allora non ci sono nodi sbilanciati e l'albero non va bilanciato
 - In questo caso può essere $z = w$, come nell'esempio in discussione



Rimozione da AVL

□ Sia z il primo nodo sbilanciato risalendo da w verso la radice (può essere $z = w$)

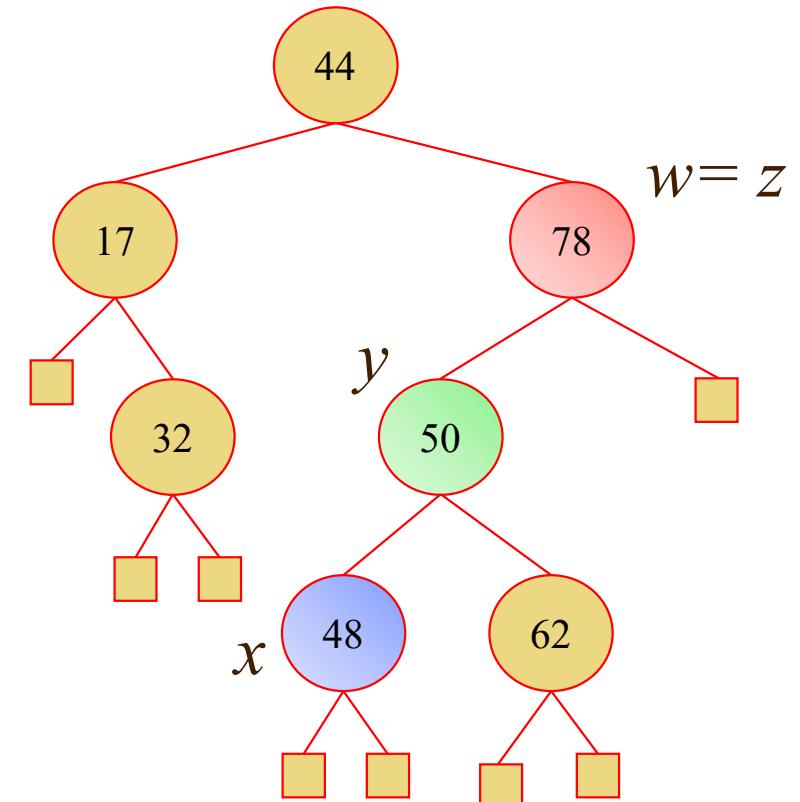
□ **Sia (ancora) y il figlio di z avente altezza massima**

- Si dimostra che, ora, y è il figlio di z che **non** è un antenato di w

□ **Sia x il figlio di y così definito**

- Quello di altezza massima, se i due figli di y hanno altezze diverse
- Altrimenti, x è il figlio di y “che si trova dalla stessa parte di y ”, cioè x è il figlio destro di y se e solo se y è il figlio destro di z
- (Ovviamente) anche x **non** è un antenato di w

□ Si dimostra che, se z esiste, allora esistono anche y e x
Inoltre, si dimostra che x non è una foglia



Rimozione da AVL

□ Si dimostra che sottponendo il sottoalbero avente radice z a una ristrutturazione uguale a quella vista per l'inserimento, in un tempo $\Theta(1)$ si ottiene

- Un sottoalbero **bilanciato** ma, in generale, **non avente la stessa altezza che aveva in precedenza**: può diminuire di un'unità
 - Non si può far meglio
- Di conseguenza, un antenato della radice di tale sottoalbero, dopo il suo bilanciamento per ristrutturazione, si può sbilanciare: si ripete la procedura usando quel nodo come z !
E così via, fino a usare, al limite, la radice dell'albero come z
 - Alla fine, quindi, l'albero torna ad essere un AVL, eventualmente diminuendo la propria altezza di un'unità
 - Il numero massimo di ristrutturazioni è $O(h)$, una per ogni antenato di w , ciascuna delle quali è $\Theta(1)$, per cui complessivamente anche la rimozione è $O(h)$ nel caso pessimo, anche se **possono rendersi necessarie più ristrutturazioni (mentre nell'inserimento ne basta sempre una)**

Prestazioni di AVL

□ Riassumiamo

- Un AVL (cioè un BST bilanciato) ha altezza $h \in O(\log n)$
- **La ricerca in un AVL** è identica alla ricerca in qualunque BST, che è $O(h)$: quindi, in un AVL la ricerca **$O(\log n)$**
- **I nodi di un AVL devono memorizzare la propria altezza**
- **L'inserimento in un AVL** si fa con l'algoritmo di inserimento nel BST seguito dall'aggiornamento delle altezze (di alcuni) degli antenati del nodo inserito, con controllo dello sbilanciamento (operazione $O(h)$), e da un'eventuale ristrutturazione locale, $\Theta(1)$, che **produce di nuovo un AVL**, avente ancora $h \in O(\log n)$: quindi, **l'inserimento in un AVL è $O(\log n)$**
- **La rimozione in un AVL** si fa con l'algoritmo di rimozione nel BST seguito dall'aggiornamento delle altezze (di alcuni) degli antenati del nodo inserito, con controllo dello sbilanciamento (operazione $O(h)$), e da eventuali $O(h)$ ristrutturazioni locali, ciascuna $\Theta(1)$, **producendo di nuovo un AVL**, avente ancora $h \in O(\log n)$: quindi, **la rimozione in un AVL è $O(\log n)$**
- **Un AVL realizza una mappa ordinata con operazioni $O(\log n)$**

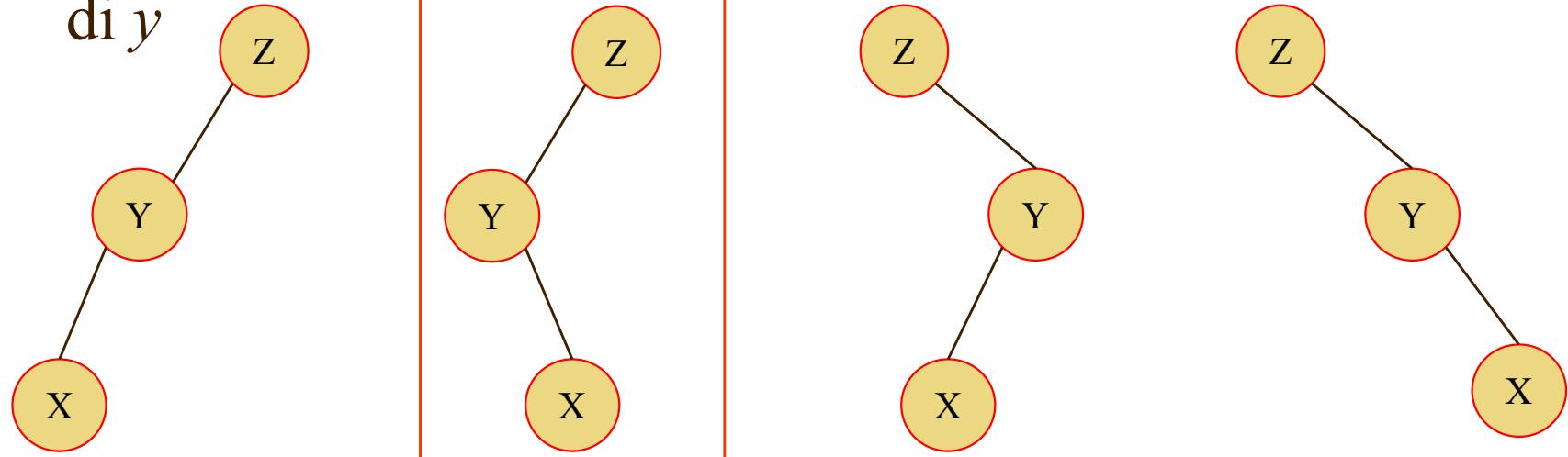
A-V & L ci
sono riusciti!!

**Dimostrazione di
correttezza per la
procedura di
inserimento in AVL
(solo per le persone
più curiose)**

Inserimento in AVL

Il secondo caso è quello relativo all'esempio precedente

- Esistono **soltanto quattro** configurazioni possibili per i nodi x , y e z
 - y può essere il figlio sinistro o destro di z e, in ciascuno di questi due casi, x può essere il figlio sinistro o destro di y

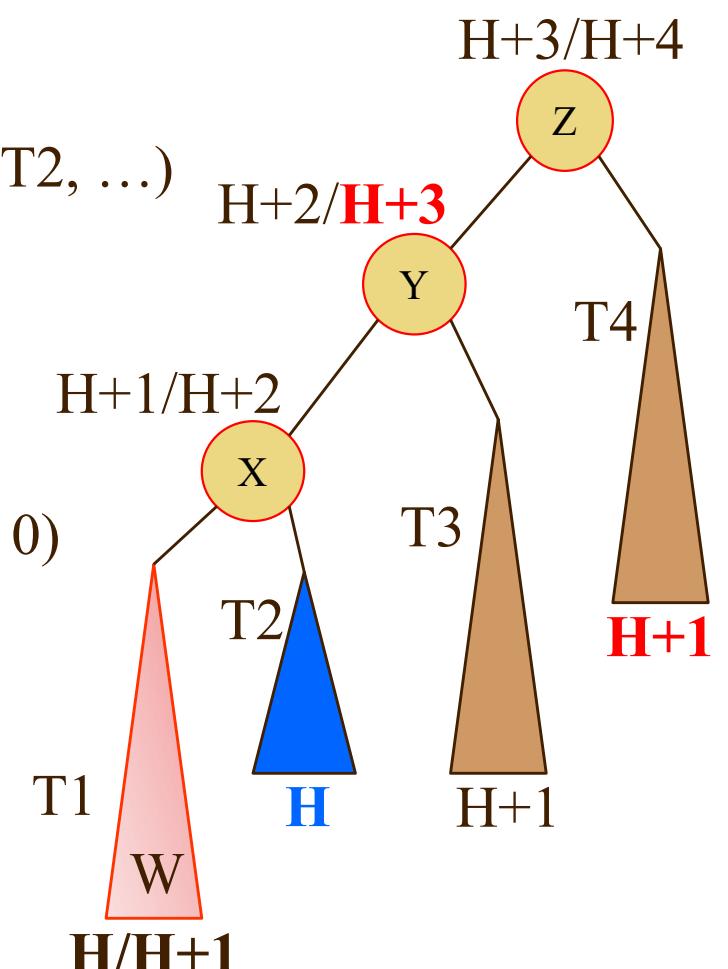


- Analizziamo in dettaglio il caso di sinistra,
completando l'albero con i necessari sottoalberi
(è un albero proprio), eventualmente contenenti soltanto
una foglia

Inserimento in AVL

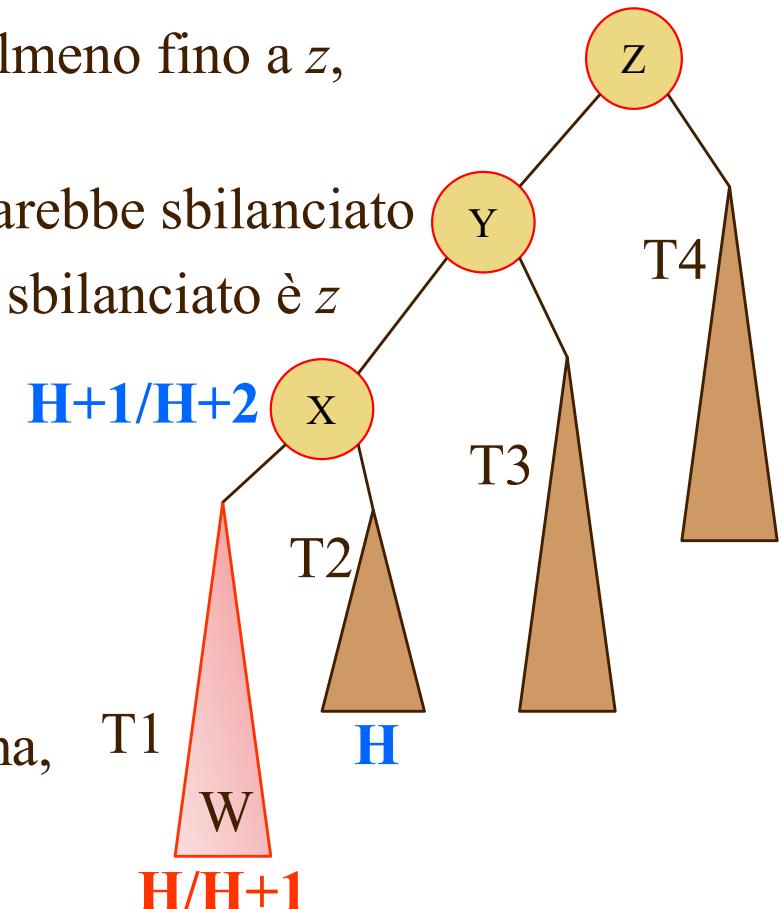
ATTENZIONE: i sottoalberi sono qui rappresentati da triangoli, ma non sono necessariamente triangolari!

- w appartiene a uno dei due sottoalberi di x (o coincide con x):
supponiamo che w appartenga al sottoalbero sinistro di x
 - Ma ricordiamoci poi di verificare il caso degenere $x = w$ e il caso in cui w appartiene al sottoalbero destro di x
- Identifichiamo con “prima” e “dopo” gli istanti precedente e successivo all’inserimento di w
- **Sotto ogni sottoalbero** (cui diamo un nome, T_1, T_2, \dots) **indichiamo la sua altezza** nel formato **prima/dopo** (un solo valore se non è cambiata)
 - Indichiamo con H l’altezza (prima e dopo) del sottoalbero destro di x (ovviamente $H \geq 0$)
- **Al di sopra di ogni nodo indichiamo la sua altezza** nel formato **prima/dopo**
- Segue discussione per giustificare i valori indicati in figura!



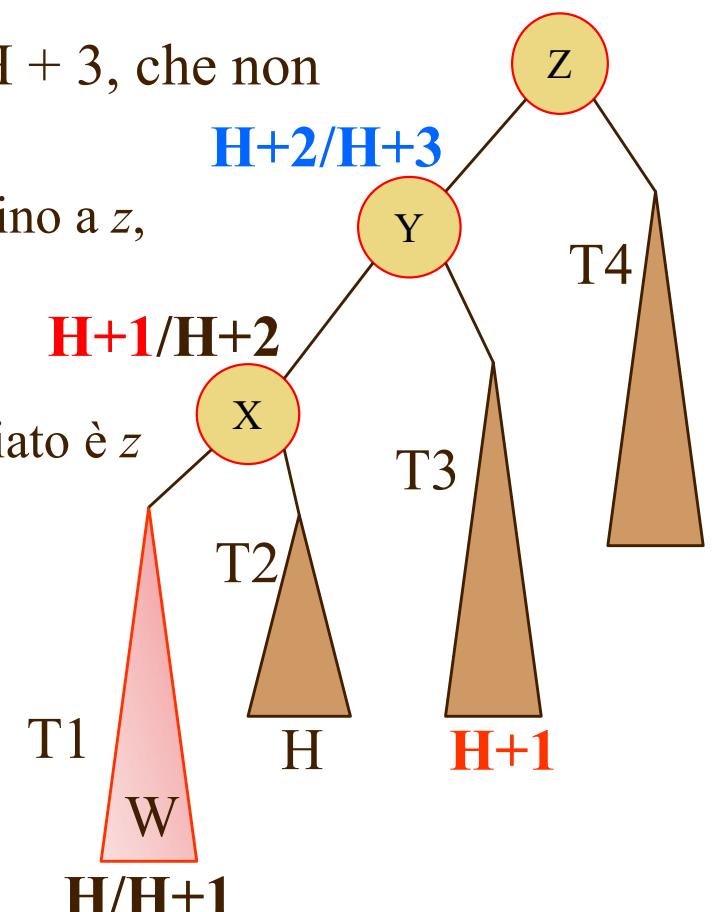
Inserimento in AVL

- Prima dell'inserimento di w , **T2 aveva altezza H per ipotesi**
- Era un AVL, l'altezza di T1 poteva essere $H - 1$, **H** o $H + 1$ perché x era bilanciato (come tutti gli altri nodi) e ora è sicuramente aumentata (altrimenti non esisterebbe z)
 - Se fosse stata $H - 1$, ora sarebbe H e l'altezza di x non sarebbe aumentata
 - Ma sappiamo che, negli antenati di w almeno fino a z , l'altezza è aumentata
 - Se fosse stata $H+1$, ora sarebbe $H + 2$ e x sarebbe sbilanciato
 - Ma sappiamo che il più profondo nodo sbilanciato è z
 - Quindi, T1 aveva altezza **H** e, conseguentemente, ora ha altezza **H + 1**
 - Quindi x aveva altezza **H + 1** (perché aveva due sottoalberi di altezza H) e ora ha altezza **H + 2** (perché il suo sottoalbero di altezza massima, T1, ha ora altezza H + 1)



Inserimento in AVL

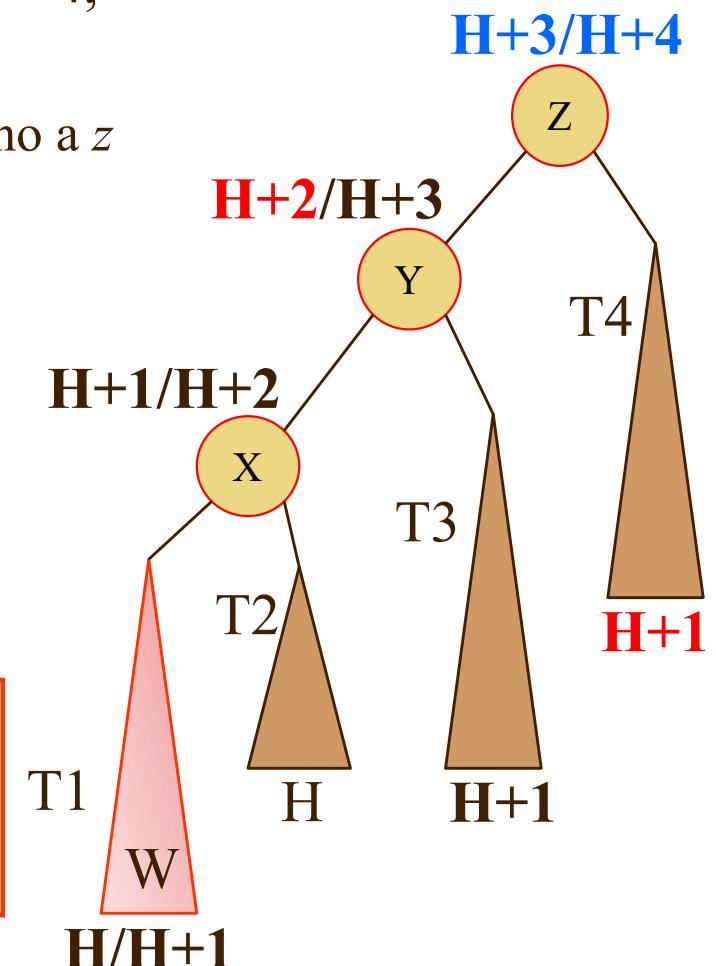
- Abbiamo dimostrato che x aveva altezza $H + 1$ e ha altezza $H + 2$
- Era un AVL, quindi y era bilanciato e l'altezza di T_3 poteva essere H , $H + 1$ o $H + 2$ (e ricordiamo che non è cambiata), perché x aveva altezza $H + 1$
 - Se fosse stata $H + 2$, y avrebbe avuto altezza $H + 3$, che non aumenterebbe
 - Ma sappiamo che, negli antenati di w almeno fino a z , l'altezza è aumentata
 - Se fosse stata H , ora y sarebbe sbilanciato
 - Ma sappiamo che il più profondo nodo sbilanciato è z
 - Quindi, T_3 aveva (e ha) altezza **$H + 1$**
 - Quindi y aveva altezza **$H + 2$** e ora ha altezza **$H + 3$**



Inserimento in AVL

- Abbiamo dimostrato che y aveva altezza $H + 2$ e ha altezza $H + 3$
- Era un AVL, quindi z era bilanciato e l'altezza di T_4 poteva essere $H + 1$, $H + 2$ o $H + 3$ (e ricordiamo che non è cambiata)
 - Se fosse stata $H + 3$, z avrebbe avuto altezza $H + 4$, che non aumenterebbe
 - Ma sappiamo che, negli antenati di w almeno fino a z (compreso), l'altezza è aumentata
 - Se fosse stata $H + 2$, ora z sarebbe **bilanciato**
 - Ma sappiamo che è sbilanciato per definizione
 - Quindi, T_4 aveva (e ha) altezza **$H + 1$**
 - Quindi z aveva altezza **$H + 3$** e ora ha altezza **$H + 4$**

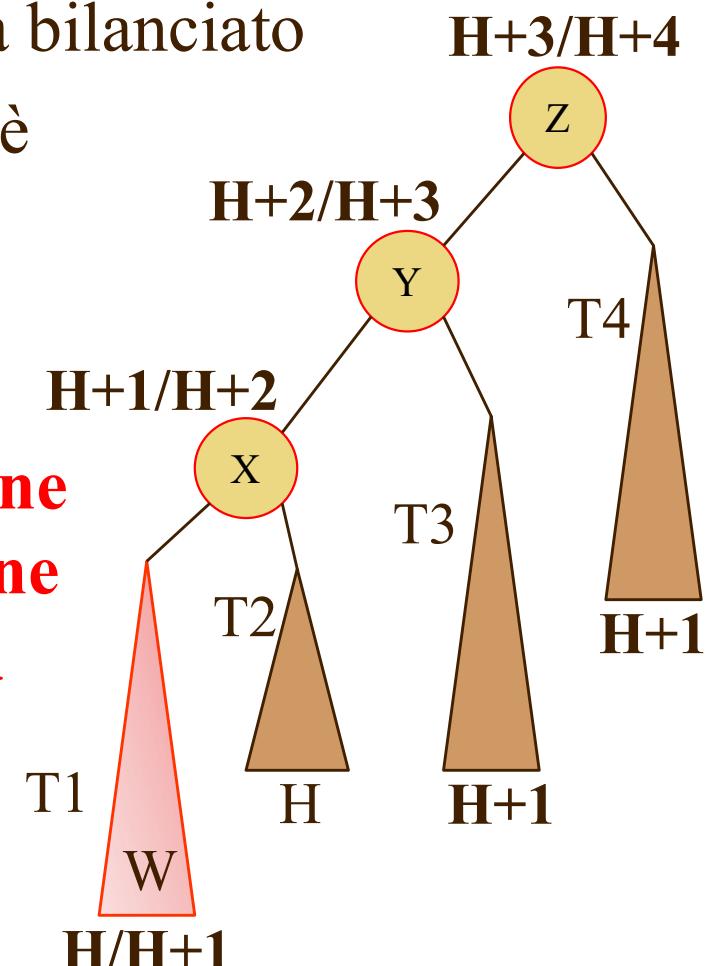
Osserviamo che, nel disegno schematico, le altezze dei sottoalberi rispecchiano le loro dimensioni, cioè T_2 è più basso degli altri, che sono tra loro uguali



Vogliamo agire senza modificare i sottoalberi T1, T2, T3 e T4

Inserimento in AVL

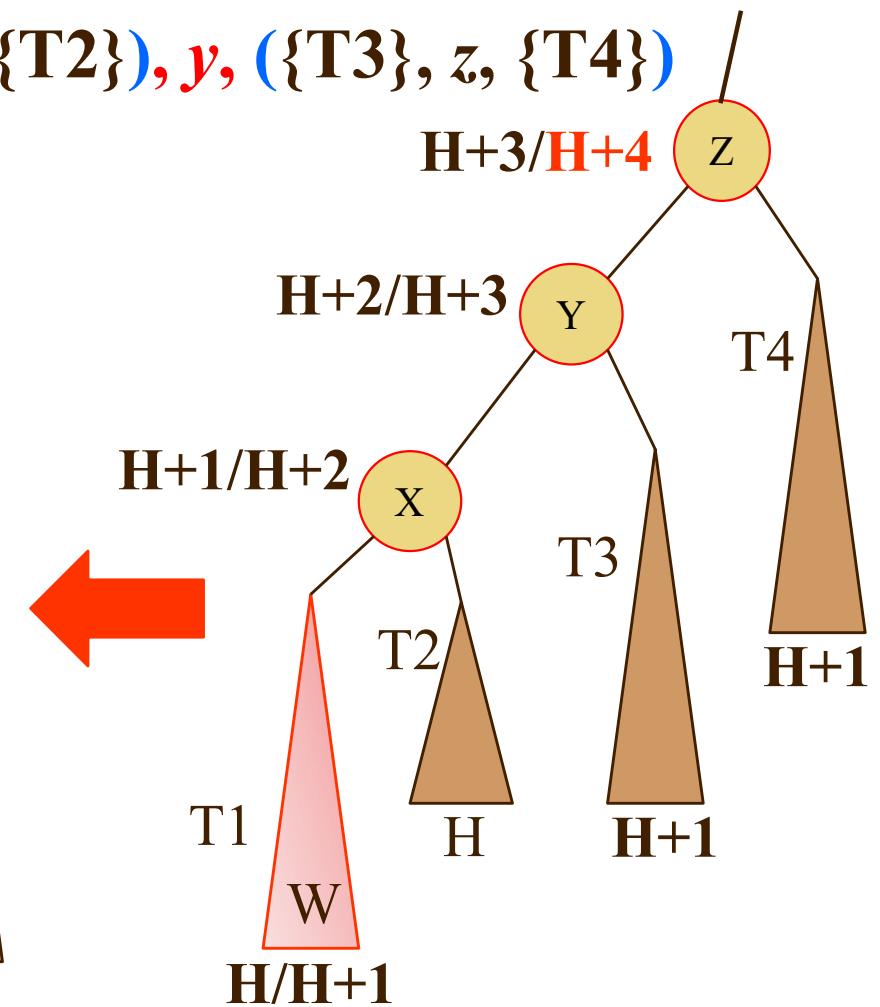
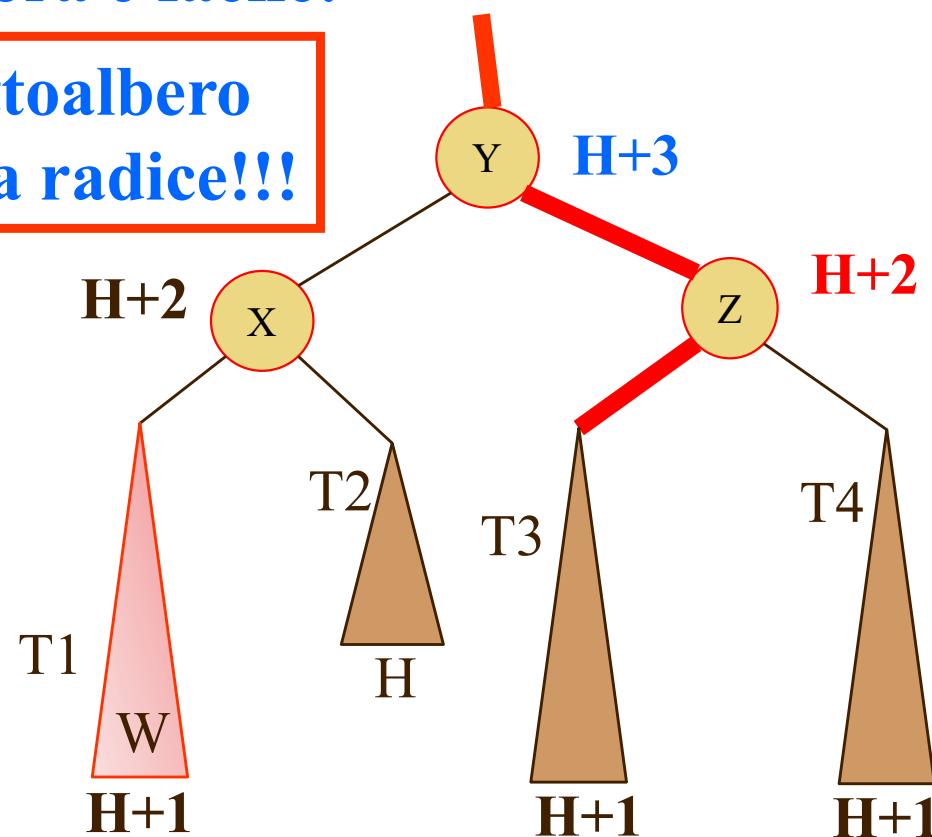
- Quindi, abbiamo dimostrato che, in seguito alla definizione dei nodi x , y e z , la situazione **era/è** quella in figura
 - Vogliamo **ristrutturare questo sottoalbero** di radice z in modo che, come prima, abbia altezza $H + 3$ e sia bilanciato
 - Notiamo che T_1 è “troppo in basso” e T_4 è “troppo in alto”: alziamo il primo e abbassiamo il secondo!
 - Dobbiamo però fare attenzione al fatto che è un BST: **L'attraversamento in ordine simmetrico deve fornire i valori in ordine**
 - L'ordine è: $\{T_1\}, x, \{T_2\}, y, \{T_3\}, z, \{T_4\}$
 - **L'ordine deve restare uguale!**
(dentro T_1 ora c'è w , nel posto giusto)
- $\{T_1\}$ è il contenuto di T_1 , etc.



Inserimento in AVL

- Vogliamo ristrutturare questo sottoalbero di radice z in modo che abbia di nuovo altezza $H + 3$ e sia bilanciato
- È un BST, quindi l'ordine è: $\{T1\}, x, \{T2\}, y, \{T3\}, z, \{T4\}$
 - L'ordine deve restare uguale!
- Proviamo a scriverlo così: $(\{T1\}, x, \{T2\}), y, (\{T3\}, z, \{T4\})$
- Allora è facile!

Il sottoalbero
cambia radice!!!



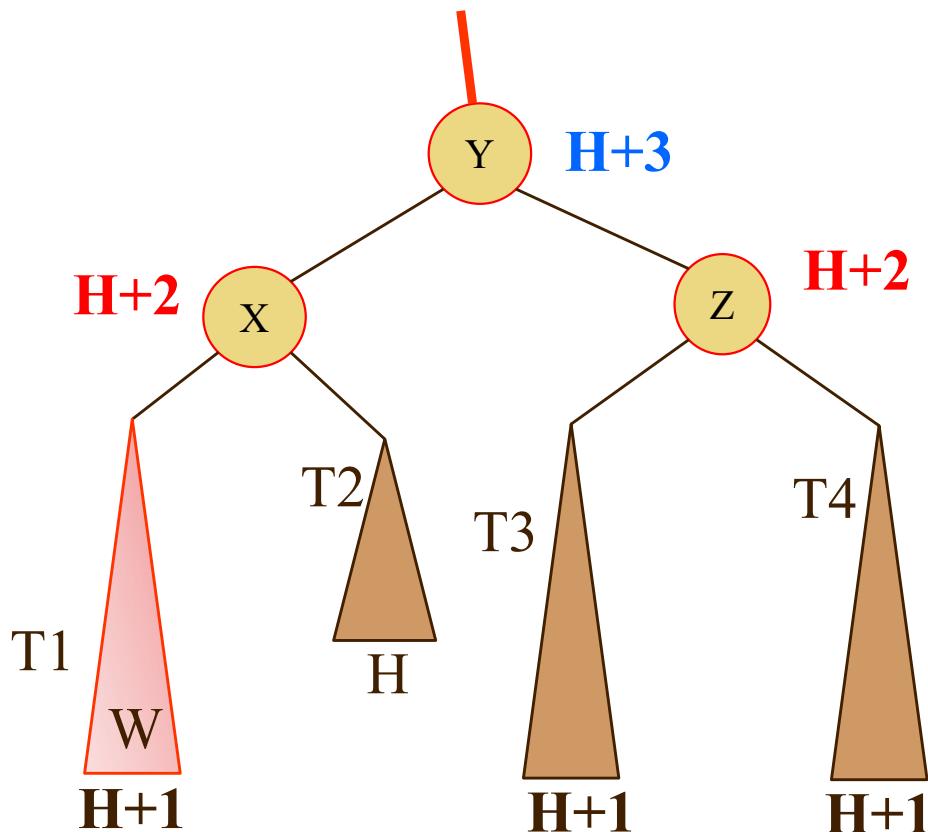
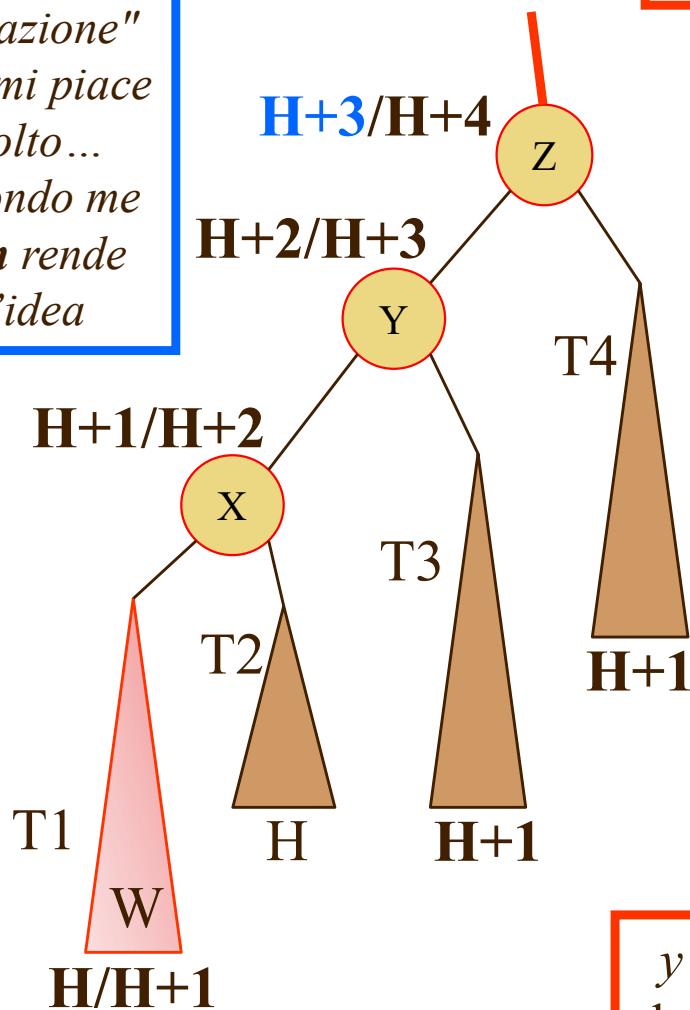
“Rotazione” in un AVL

- Dal punto di vista topologico, questa **ristrutturazione** venne chiamata dagli autori “**rotazione**” (di y attorno a z): è un’operazione **locale**, $\Theta(1)$, che ha effetti **globali**

A-V & L ci
sono riusciti!!

Il nome
"rotazione"
non mi piace
molto...
secondo me
non rende
l’idea

Gli UNICI nodi in cui bisogna aggiornare l’altezza
sono x , y e z , oltre agli antenati di w all’interno di T_1

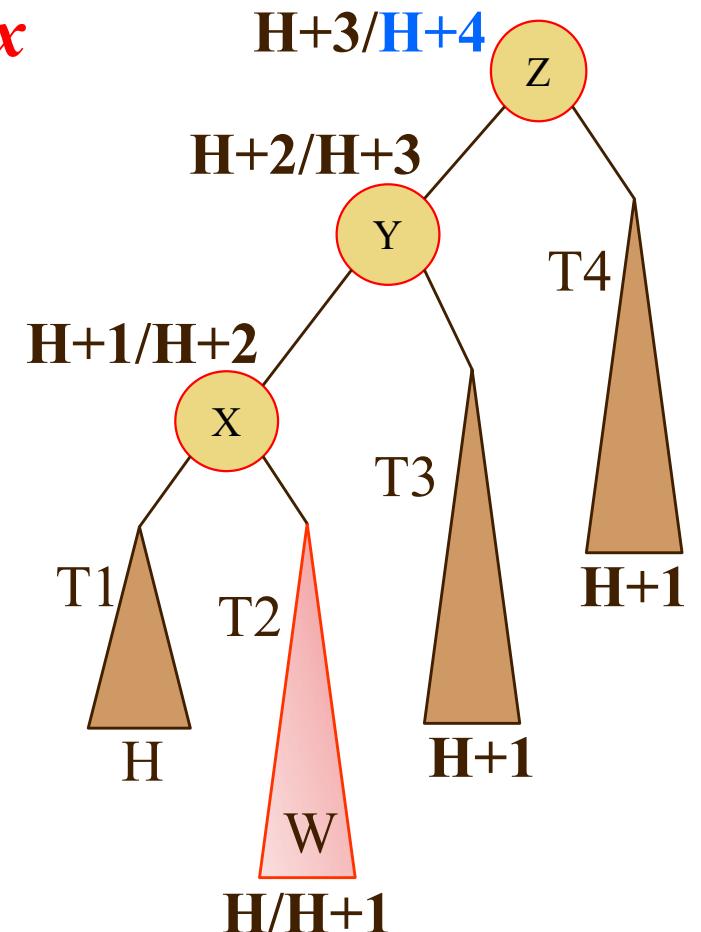
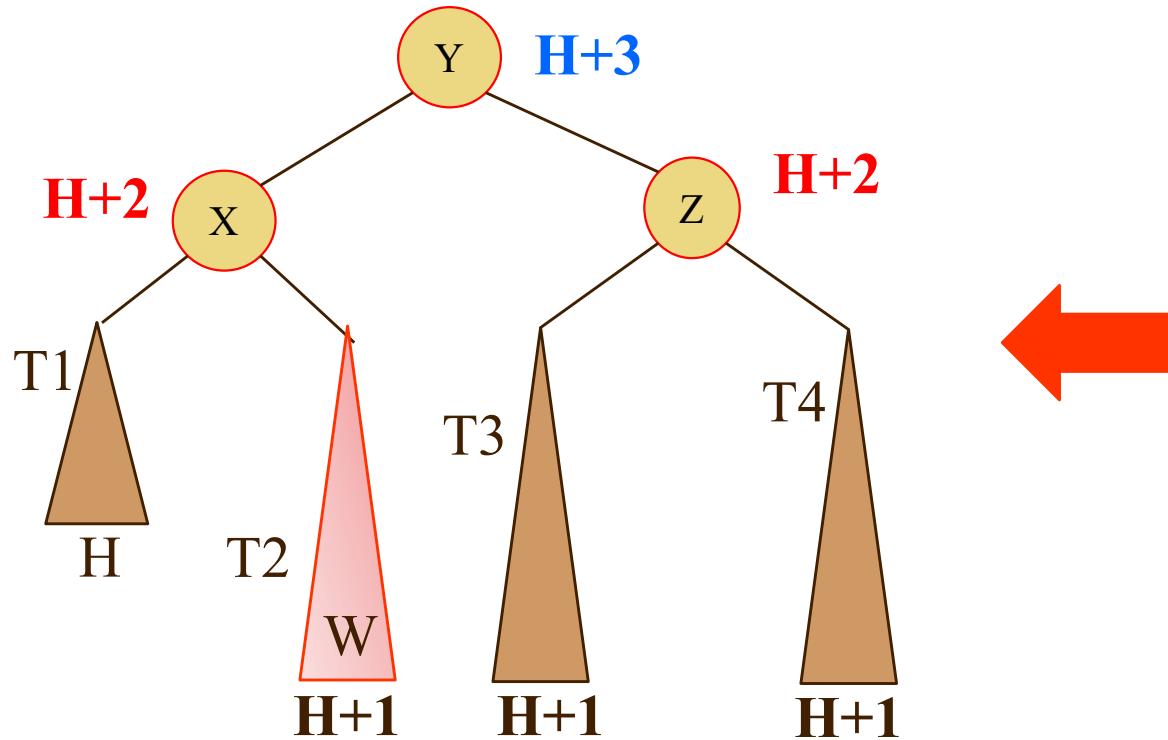


y diventa radice del sottoalbero in esame: come richiesto,
la “nuova” altezza di y è uguale alla “vecchia” altezza di z

Inserimento in AVL

□ Abbiamo alcune cose da verificare

- Caso degenero $w = x$
- Le altre tre configurazioni di x, y e z
- **Caso con w nel sottoalbero destro di x anziché sinistro (non cambia niente)**

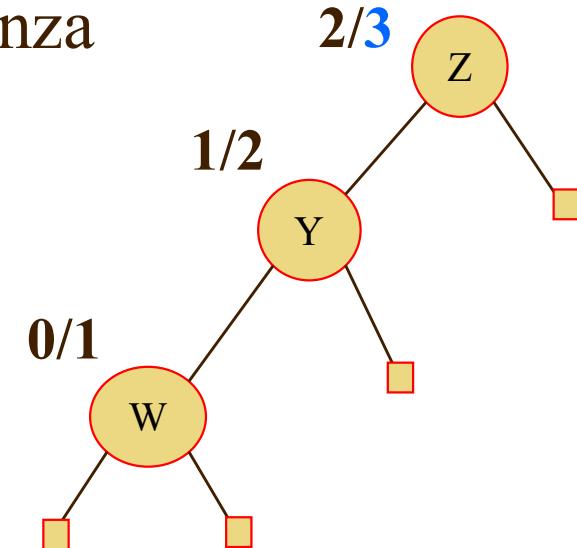
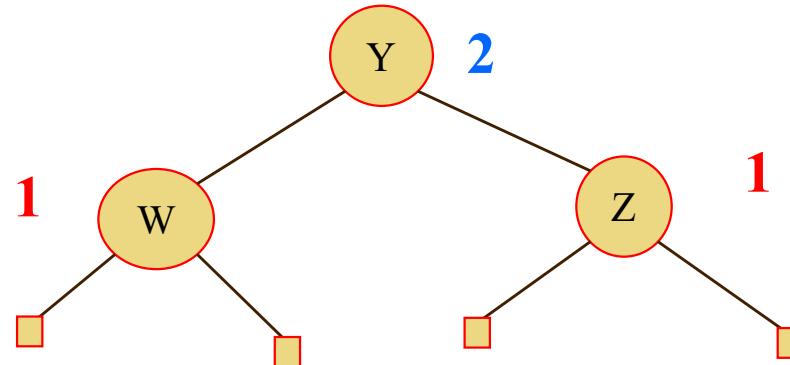


Inserimento in AVL

Provare a fare la dimostrazione

□ Caso degenere $w = x$

- Ne consegue (da dimostrare...) che i 4 sottoalberi sono foglie, ma la dimostrazione procede come in precedenza

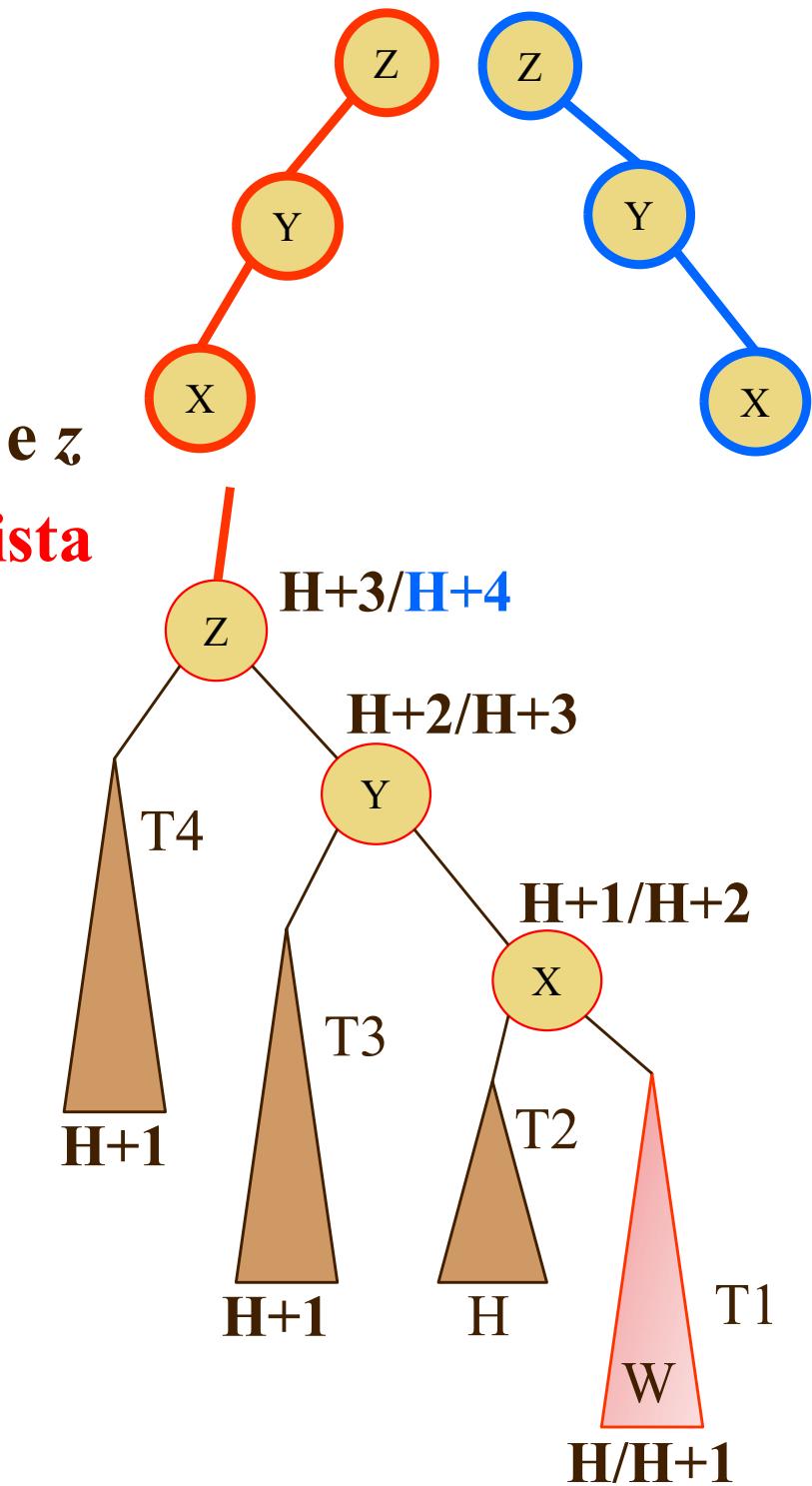
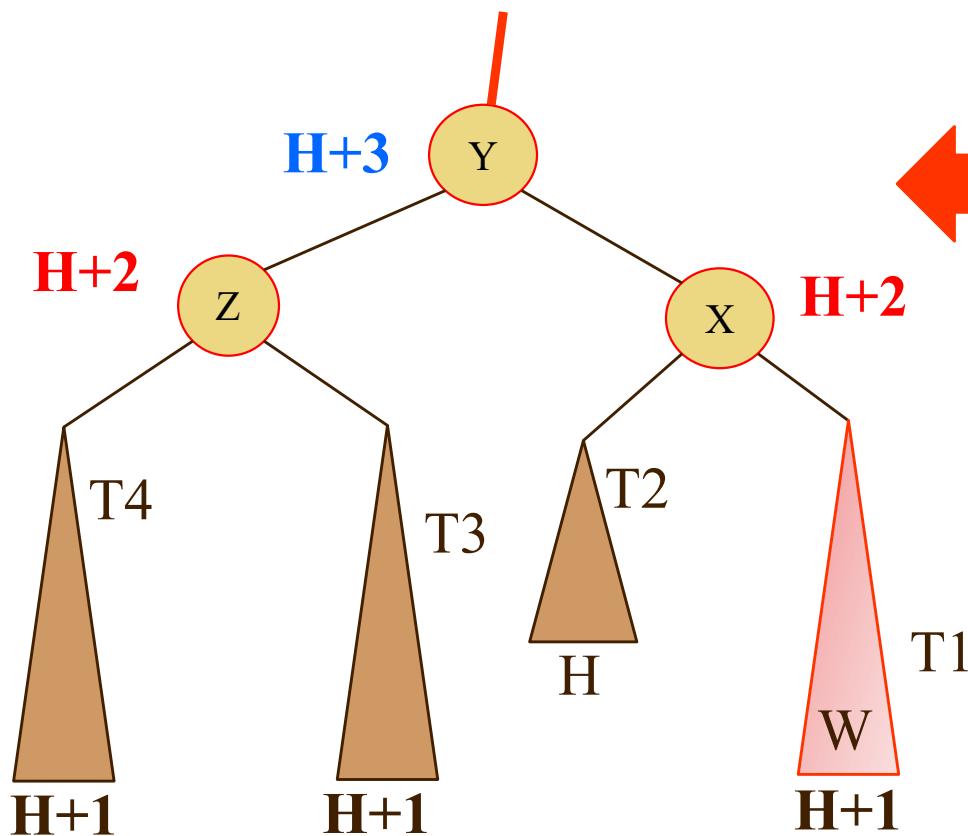


- Esempio: partendo da una struttura vuota, inseriamo 30, poi 20, infine 10 \Rightarrow si ottiene una topologia come quella di destra, che viene ristrutturata ottenendo la topologia di sinistra, con 20 nella radice

Inserimento in AVL

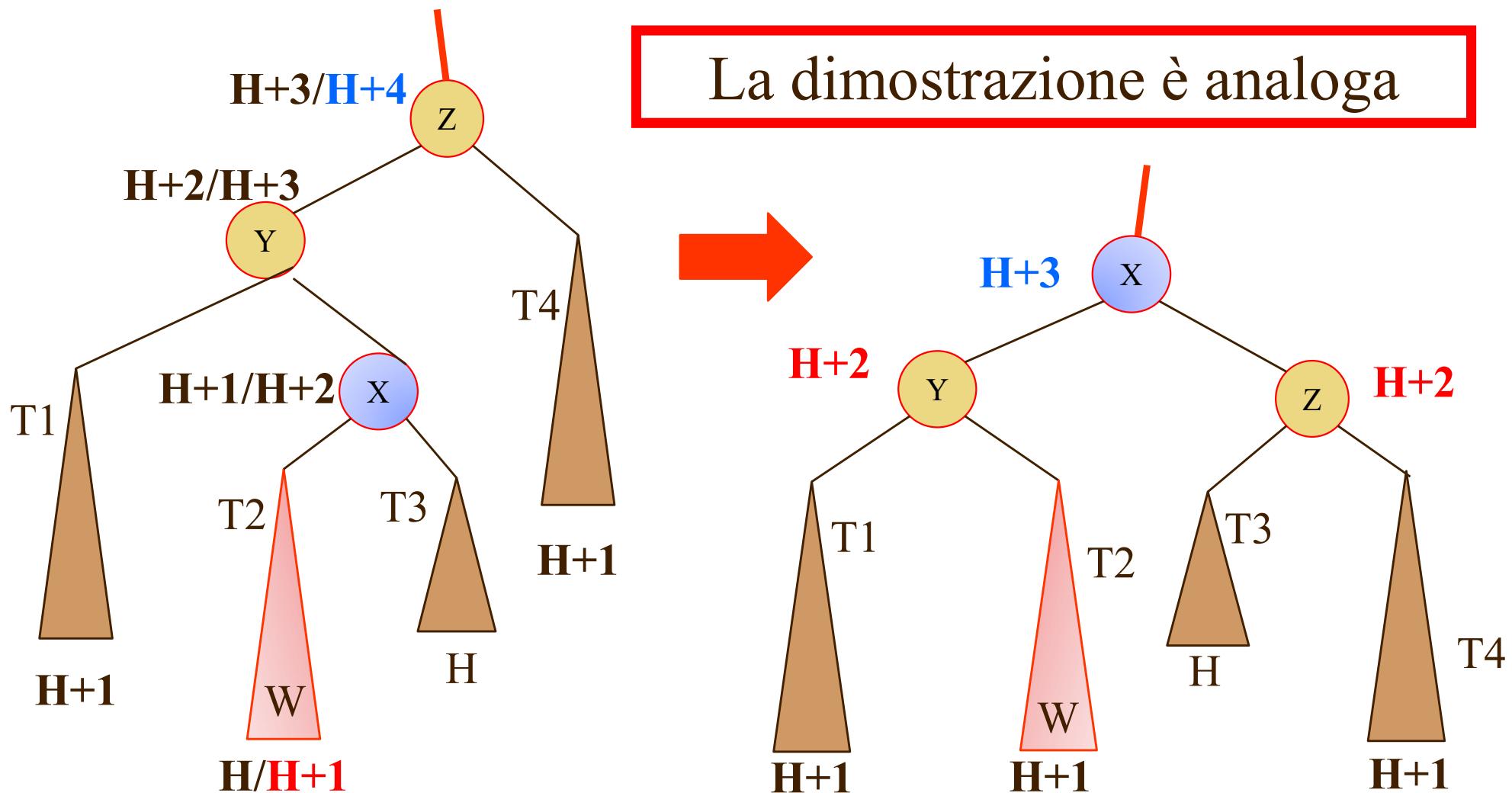
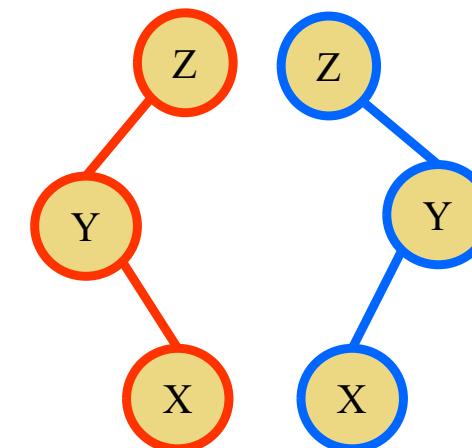
□ Abbiamo alcune cose da verificare

- Le altre tre configurazioni di x , y e z
- Ce n'è una simmetrica a quella vista



Inserimento in AVL

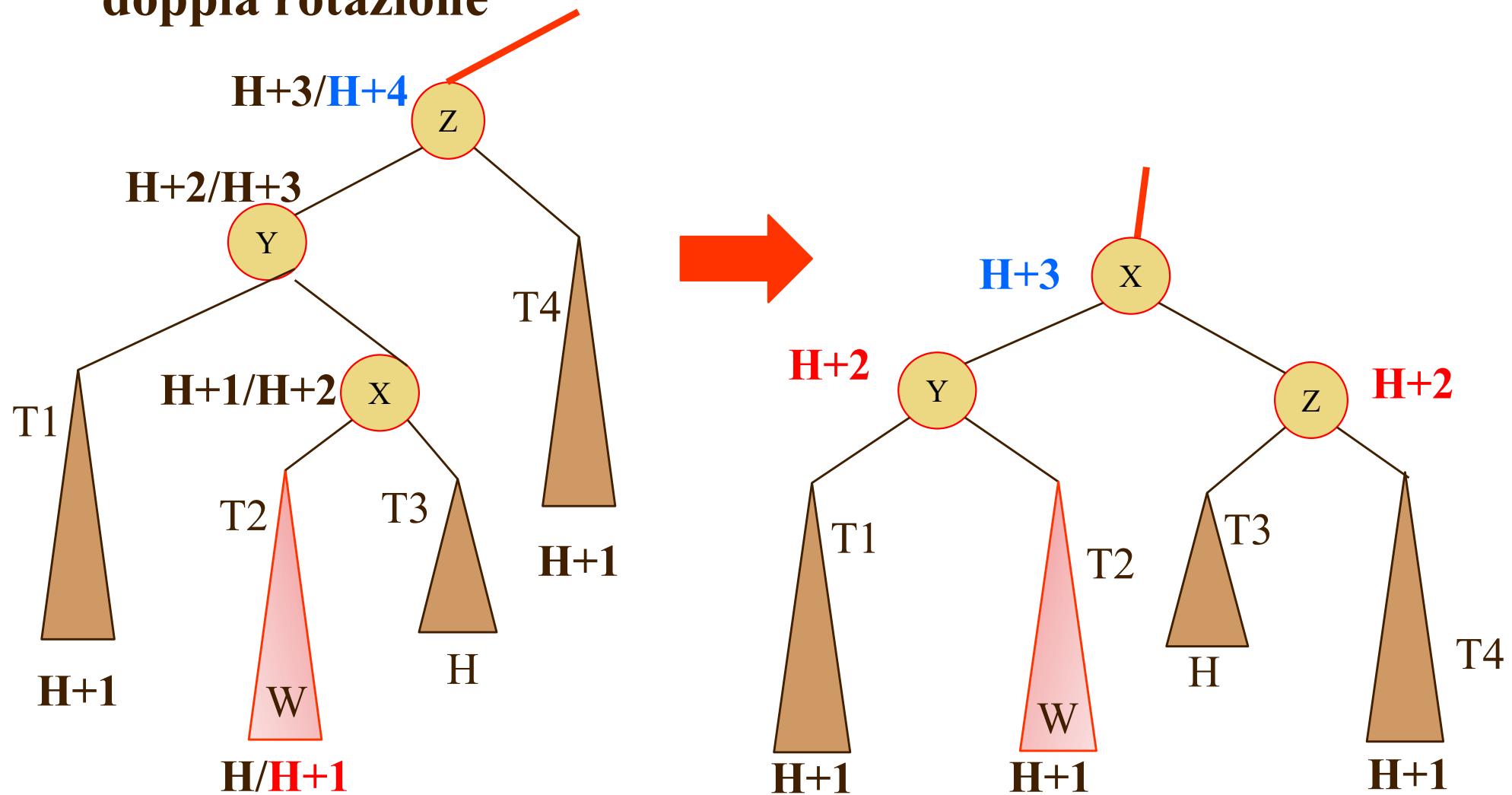
- Le altre due configurazioni sono tra loro simmetriche, ne vediamo **soltanto una**



Rotazione “doppia”

*Il nome "rotazione" non mi piace molto...
secondo me non rende l'idea*

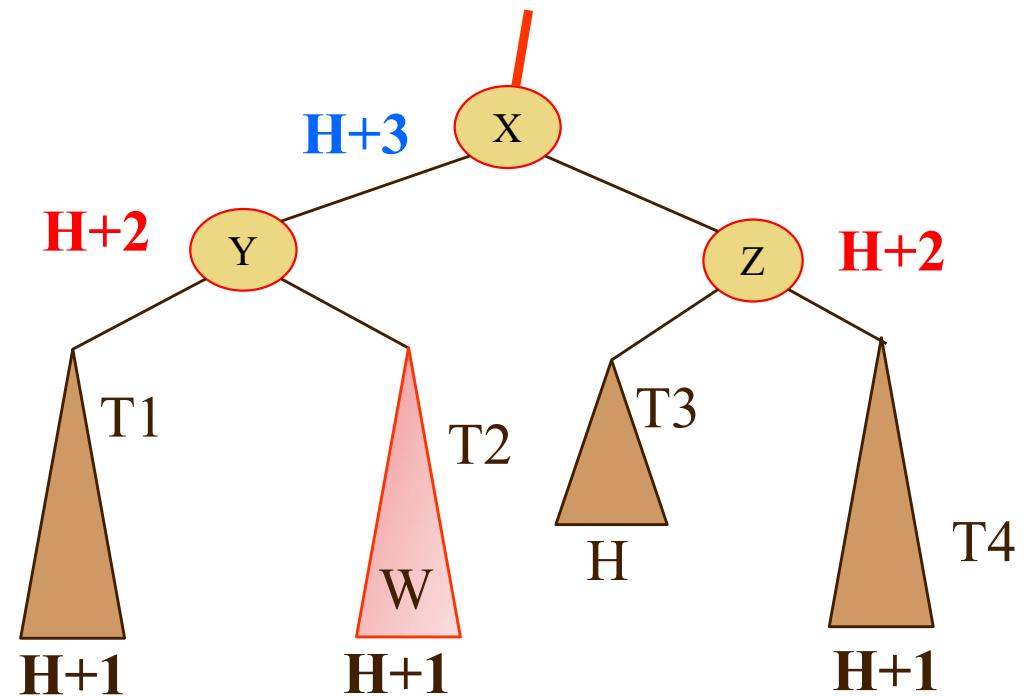
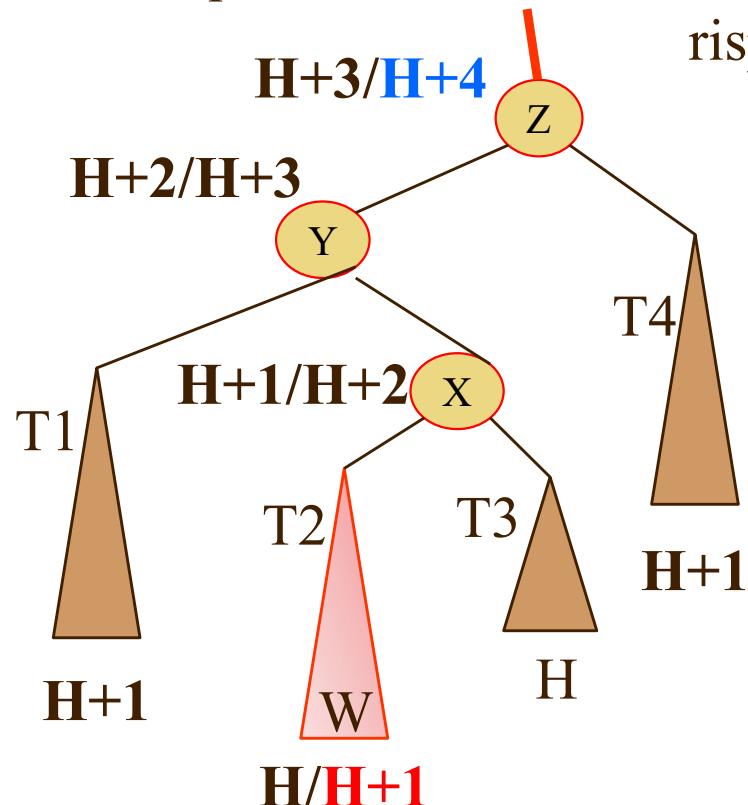
- Parlando di rotazioni, si dice che questa ristrutturazione si esegue facendo ruotare prima “ x attorno a y ”, poi “ x attorno a z ”, una **doppia rotazione**



Una regola...

Questa regola è molto più semplice
e funziona in tutti i casi, senza
parlare di “rotazioni”

- Osserviamo, piuttosto, come si agisce, **guardando soltanto i nodi x , y , e z** : in ognuna delle quattro situazioni, si deve “compattare” un albero di altezza 2
- Si prende **il valore intermedio tra i tre** e lo si fa diventare radice!! Sistemando poi gli altri di conseguenza, come figli sinistro e destro. Proprio come nella procedura di costruzione di un BST di altezza minima....
- I quattro sottoalberi si “attaccano” di conseguenza, nei rami liberi, rispettando il loro ordinamento relativo precedente



**Fine delle
dimostrazioni della
procedura di
inserimento in AVL
(per la procedura di
rimozione, si possono fare
dimostrazioni analoghe)**

Altri alberi di ricerca?

- Negli anni sono state progettate **molte** altre strutture ad albero per realizzare mappe ordinate
 - Alcune sono alberi binari
 - Es. **Splay Tree** (1985) ha prestazioni $O(\log n)$ **in media** (con analisi ammortizzata) per ricerca, inserimento e rimozione, quindi è peggiore di un albero AVL; per di più è $O(n)$ nel caso pessimo
 - Vantaggio: non ha bisogno di nodi "speciali" (negli AVL hanno l'altezza)
 - Vantaggio per alcune applicazioni: **tende a rendere più veloci le ricerche di chiavi cercate di recente**
 - Es. **Red-Black Tree** (albero rosso-nero, 1978) stesse prestazioni degli alberi AVL, ma “ristrutturazioni” più veloci (anche se non asintoticamente), soprattutto la rimozione
 - Il pacchetto **java.util** contiene **TreeMap**, una mappa ordinata realizzata in questo modo
 - Altre sono alberi **non binari**
 - Es. **(2,4) Tree**, stesse prestazioni degli alberi AVL e rosso-nero

Sempre alberi di ricerca?

- Nell'implementazione di una mappa ordinata, non sempre è necessario o preferibile utilizzare un AVL (o un'altra struttura con le stesse prestazioni)
- Rispetto a una semplice implementazione mediante array ordinato, l'AVL migliora le prestazioni dell'inserimento e della rimozione
 - Le prestazioni della ricerca sono asintoticamente identiche, ma quelle dell'array ordinato sono migliori (o, meglio, non peggiori) perché sempre equivalenti a quelle di un BST di altezza minima (il problema viene sempre suddiviso a metà...), mentre un AVL, in generale, non ha altezza minima (quindi, in generale, non suddivide il problema a metà ad ogni passo)
 - Inoltre, c'è un risparmio di memoria (non servono i nodi...)
- **Si usa un AVL (o simile) soltanto quando l'applicazione richiede una mappa ordinata dal contenuto molto dinamico, con frequenti operazioni di inserimento e/o rimozione**

Prestazioni: riassunto

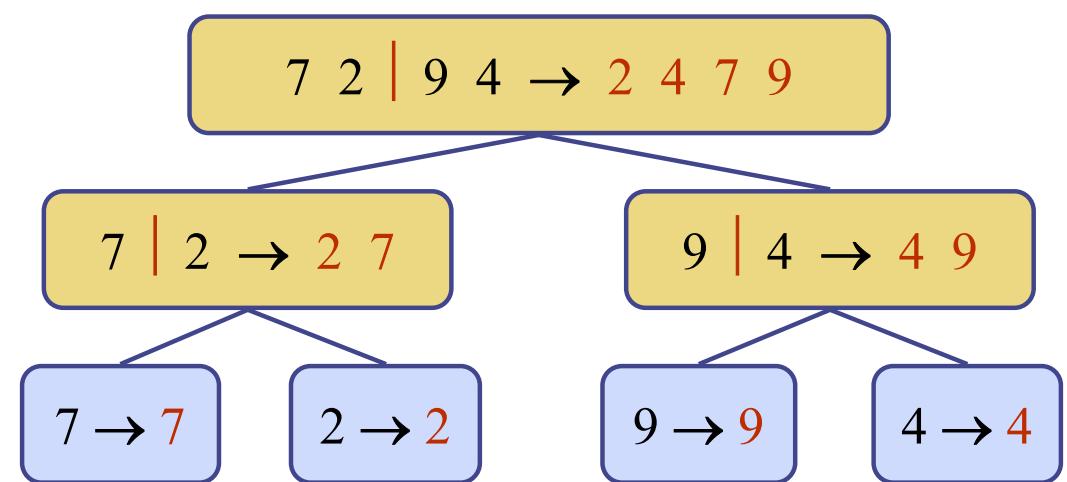
	get	put	remove	keys / values
Mappa in lista (non ordinata)	$O(n)$	$O(n)$	$O(n)$	$\Theta(n)$
Mappa ordinata in array ordinato	$O(\log n)$	$O(n)$	$O(n)$	$\Theta(n)$
Mappa ordinata in AVL	$O(\log n)$	$O(\log n)$	$O(\log n)$	$\Theta(n)$

E le mappe "non ordinate"?

- Se serve una semplice mappa **ma le chiavi sono ordinabili**, meglio usare comunque un AVL (o un array ordinato, se l'applicazione deve fare poche modifiche alla mappa)
 - Anche se non ci serve che il metodo **keys** restituisca le chiavi in ordine
 - La mappa in array non ordinato ha prestazioni, per tutti i metodi, $O(n)$ nel caso pessimo e **nel caso medio**

Lezione 29

Ordinamento



Si può far meglio di $O(n \log n)$?

- I due algoritmi di ordinamento migliori che conosciamo (MergeSort e HeapSort) sono **entrambi $O(n \log n)$** nel caso pessimo
 - È un caso? Si può far meglio? **Magari no?**
- SI DIMOSTRA che
 - Qualunque algoritmo di ordinamento che traggia informazioni da **confronti** tra gli elementi da ordinare è, **nel caso pessimo**,
 $\Omega(n \log n)$
 - **Non si può far meglio!**
 - Di conseguenza. **nell'insieme dei possibili algoritmi di ordinamento per confronto, MergeSort e HeapSort sono, nel caso pessimo, asintoticamente ottimi**
 - Naturalmente possono esistere algoritmi asintoticamente più veloci nel caso ottimo o in altri casi (es. insertionSort nel caso ottimo è lineare)

Si potrà ordinare **senza fare confronti???** Vedremo...

**Dimostrazione del
limite inferiore
asintotico per le
prestazioni di algoritmi
di ordinamento
(solo per le persone
più curiose)**

Si può far meglio di $O(n \log n)$?

- Faremo alcune ipotesi semplificative che non tolgono significatività al risultato, dato che cerchiamo un **limite inferiore asintotico** al tempo di esecuzione **nel caso pessimo**
 - Le ipotesi che faremo ci porteranno a individuare **un limite inferiore eventualmente sottostimato** (ma corretto), che, quindi, potrebbe non essere il limite inferiore più stringente
 - Però **sappiamo già che esistono algoritmi $O(n \log n)$ nel caso pessimo**, quindi un limite inferiore asintotico nel caso pessimo che sia valido per qualsiasi algoritmo non può essere asintoticamente superiore a $\Omega(n \log n)$
 - Altrimenti gli algoritmi citati violerebbero tale limite inferiore!
 - Quindi se, facendo una eventuale sottostima di tale limite inferiore, ottengo $\Omega(n \log n)$, significa che NON ho ottenuto una sottostima, ma il vero limite inferiore asintotico, perché non ne può esistere uno superiore a $\Omega(n \log n)$
 - Devo, però, fare attenzione: le ipotesi semplificative devono garantire di non portare mai a una **sovraffidata** del limite inferiore asintotico, altrimenti perde di significato, perché potrebbe non essere più un limite inferiore asintotico

Si può far meglio di $O(n \log n)$?

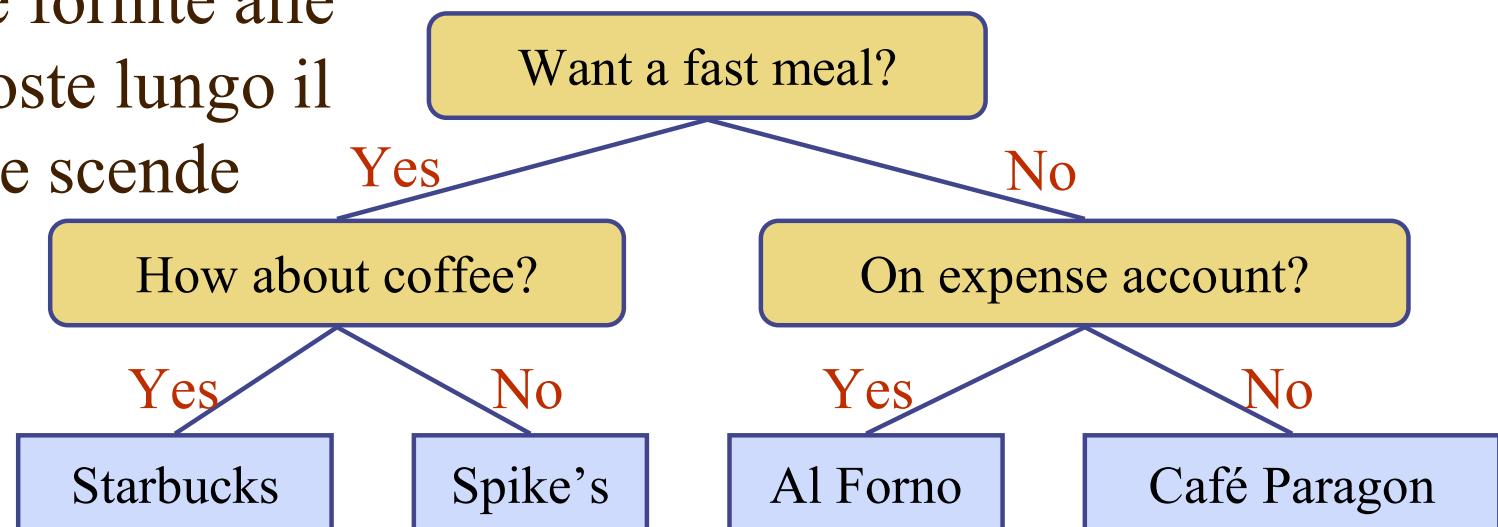
- Ipotesi semplificative che possono portare a sottostimare il limite inferiore che cerchiamo ma non tolgono significatività al risultato, dato che cerchiamo un **limite inferiore asintotico** al tempo di esecuzione **nel caso pessimo**
 - Le uniche operazioni che richiedono tempo sono i **confronti tra gli elementi** da ordinare e **ciascun confronto è $\Theta(1)$**
 - Se ci sono altre operazioni “costose”, il tempo sarà **maggiore**
 - Se il singolo confronto non è $\Theta(1)$, il tempo sarà **maggiore**
 - Vogliamo ordinare una **sequenza** $S = (a_1, a_2, \dots, a_n)$
 - Non importa se è un array/vettore o una lista posizionale, tanto non valutiamo il tempo necessario a fare scambi o ad accedere ai valori, ci interessa **solo il tempo necessario a fare i confronti**, che ovviamente non dipende dalle caratteristiche della struttura di memorizzazione
 - Gli elementi della sequenza sono distinti
 - La gestione degli eventuali elementi duplicati richiederà eventualmente un po’ di tempo in più... non ci interessa, ammettiamo una sottostima
 - Se, invece, la presenza di elementi duplicati fa risparmiare tempo... non ci interessa ugualmente: vuol dire che non è il caso pessimo!

Si può far meglio di $O(n \log n)$?

- Usiamo il simbolo \leq per la relazione d'ordine totale su cui si basa l'algoritmo di ordinamento (che qui chiamiamo genericamente **sort**, indifferentemente iterativo o ricorsivo)
- Ogni volta che il generico **sort** confronta due elementi della sequenza, a_i e a_j :
 - La risposta alla domanda “ $a_i \leq a_j$?” può essere (soltanto) **sì** o **no** (per la proprietà di totalità)
 - Sulla base della risposta ottenuta, **sort** effettuerà accessi alla sequenza e/o la modificherà
 - Come detto, trascuriamo il tempo così impiegato
 - Se non ha finito di ordinare la sequenza, **sort** confronterà altri due valori

Albero di decisione: ci è utile

- Esempio di **albero binario proprio**: **albero di decisione** (*decision tree*, o anche **BDD**, *binary decision diagram*)
 - Ogni nodo interno contiene una domanda
 - Il suo figlio sinistro contiene l'effetto della risposta “Sì”
 - Il suo figlio destro contiene l'effetto della risposta “No”
 - Oppure viceversa... ma coerentemente in tutti i nodi
 - Ogni foglia contiene una “**decisione finale**”, conseguente alle risposte fornite alle domande poste lungo il percorso che scende dalla radice



Si può far meglio di $O(n \log n)$?

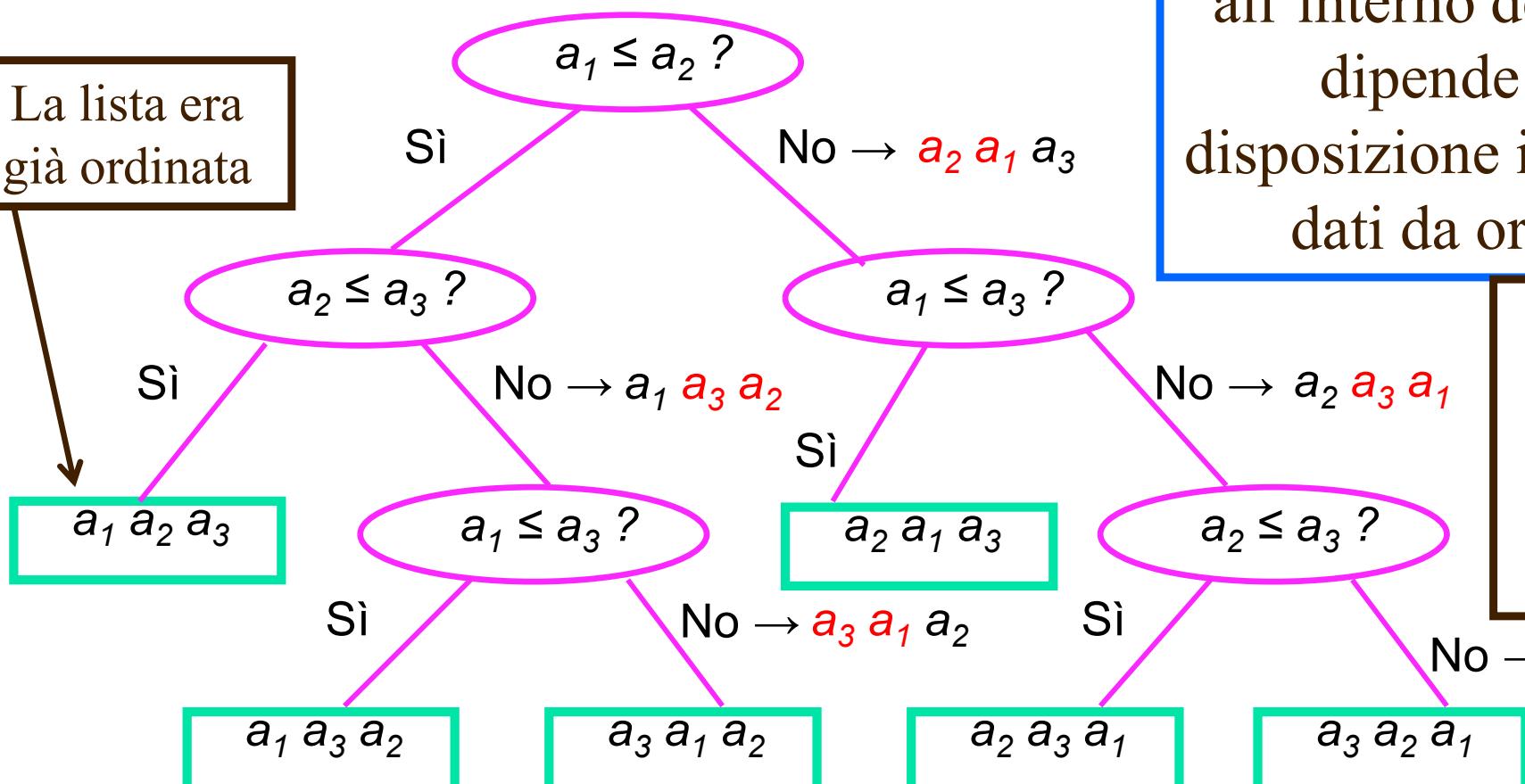
- Il processo decisionale di un algoritmo di ordinamento che operi mediante confronti si può, quindi, rappresentare con un **albero di decisione** (che è un albero binario proprio)
 - Ogni **nodo interno** contiene una domanda del tipo “ $a_i \leq a_j ?$ ”
 - Cosa contengono le **foglie**? Una “decisione finale”... Come rappresentiamo la “decisione finale” presa da un algoritmo di ordinamento al termine dei confronti che gli sono necessari?
 - Ad esempio, con la sequenza degli indici dei valori nell’array originario (o la sequenza delle loro posizioni nella lista) che genera la sequenza ordinata

Esempio: Insertion Sort con $n = 3$

In generale, **ogni algoritmo utilizzerà**
(eventualmente in modo implicito) **un
diverso albero di decisione**

Il cammino percorso
dall'algoritmo
all'interno dell'albero
dipende dalla
disposizione iniziale dei
dati da ordinare

La lista era
già ordinata



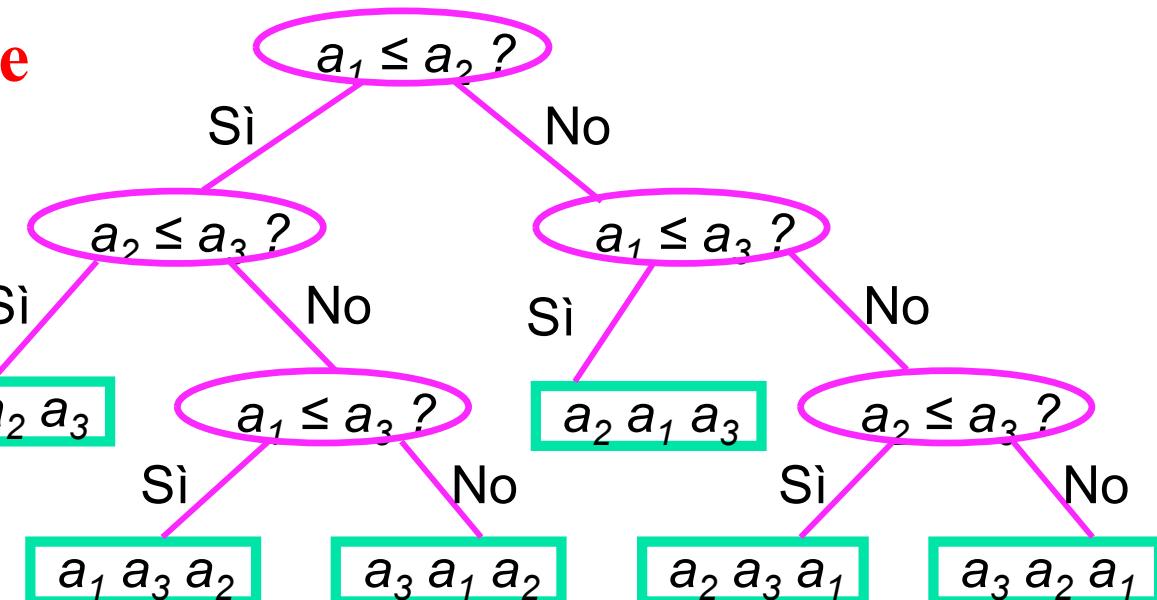
In questo
esempio,
ogni “No”
implica uno
scambio

Questa è la sequenza ordinata, il contenuto della foglia sarà 1, 3, 2

Si può far meglio di $O(n \log n)$?

- Il tempo impiegato dall'algoritmo a “prendere una decisione finale”, cioè a ordinare una sequenza, dipende dalla profondità della foglia che rappresenta la sequenza ordinata da individuare, in relazione ai valori a_i
 - Nel caso pessimo**, il tempo impiegato è, quindi, linearmente dipendente dall'**altezza** dell'albero di decisione, perché l'algoritmo non va "su e giù" nell'albero di decisione
- Stiamo cercando di capire quali siano le prestazioni di **caso pessimo** dell'algoritmo **migliore** che si possa progettare, perché cerchiamo il limite **inferiore** alle prestazioni di caso pessimo

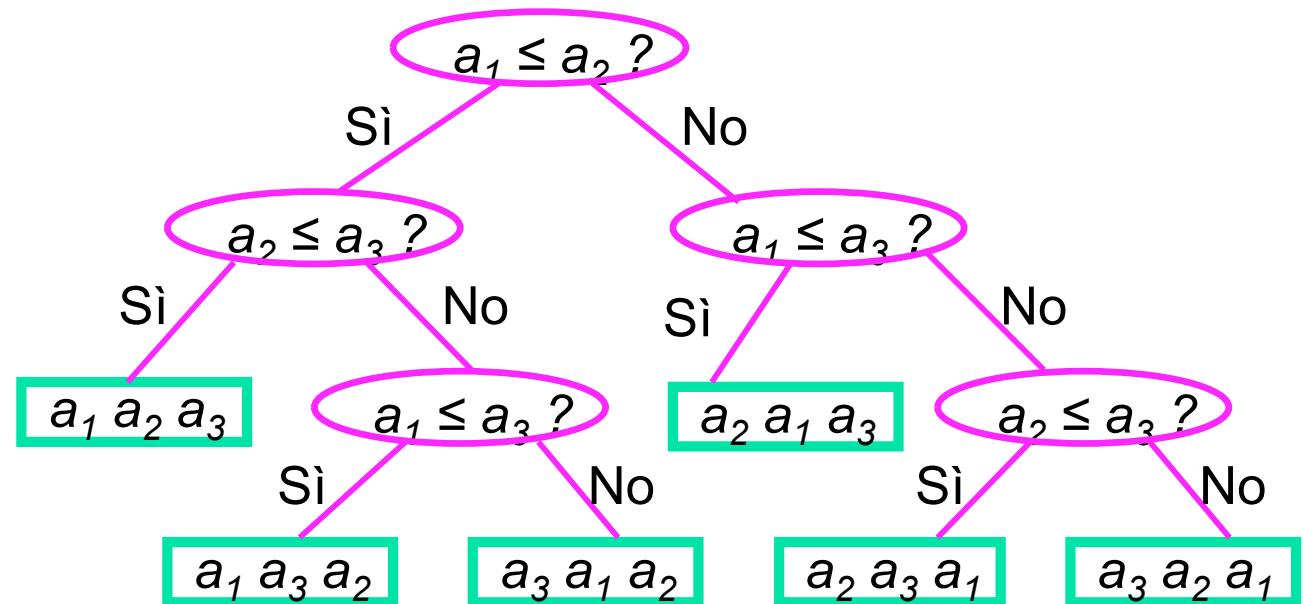
- L'algoritmo migliore sarà quello avente l'albero di decisione con altezza **minima**!



Si può far meglio di $O(n \log n)$?

- **Che altezza ha l'albero di altezza minima** tra gli alberi di decisione che rappresentano tutti i possibili algoritmi di ordinamento che operano mediante confronti?
- Ricordiamo che in un albero binario proprio (quale è l'albero di decisione) si ha $n_E \leq 2^h$, cioè $h \geq \log_2 n_E$ quindi $h_{\min} = \lceil \log_2 n_E \rceil$
- Qual è, quindi, il numero **minimo** di foglie, n_E , che può avere un tale albero di decisione? In corrispondenza, avremo l'albero di altezza minima

La dimostrazione **NON**
è costruttiva: non
sapremo quale sarà
l'algoritmo avente
albero d'esecuzione di
altezza minima



- **Proprietà:** Tutti gli alberi di decisione che rappresentano un algoritmo di ordinamento che opera mediante confronti su una sequenza contenente n elementi **distinti** hanno **almeno $n!$** foglie
- Ricordiamo dalla matematica combinatoria che esistono $n!$ (diverse) permutazioni di n elementi distinti
 - Quindi esistono **$n!$ diverse possibili sequenze iniziali** contenenti gli stessi n elementi distinti (cioè $n!$ diversi possibili esemplari del problema di ordinamento degli n elementi di un insieme)
 - Per ognuna di queste deve esistere **almeno una foglia** che contiene la soluzione del problema, perché **il problema dell'ordinamento ha** (ovviamente) **sempre soluzione**
 - Inoltre, **una foglia non può contenere la soluzione di due problemi diversi** (come si dimostra agevolmente ragionando per assurdo, vedi nel seguito)
 - Quindi esistono **almeno $n!$ foglie**

- Dimostriamo che una foglia non può contenere la soluzione di due problemi diversi
- Ipotizziamo, per assurdo, che esistano due problemi di ordinamento diversi, P_1 e P_2 , la cui soluzione sia presente nella stessa foglia f
- Perché i due problemi (che contengono due permutazioni degli stessi valori) siano diversi, è necessario che esista almeno una coppia di indici, a e b , tali che $x_{1a} < x_{1b}$ in P_1 , mentre $x_{2b} < x_{2a}$ in P_2 (x_{1a} è il valore avente indice a in P_1 e così via)
- La soluzione presente in f indica se il valore associato all'indice a deve precedere il valore associato all'indice b (o viceversa) ma questo significa che la sequenza ordinata corrispondente alla situazione iniziale P_1 oppure alla situazione iniziale P_2 contiene necessariamente un errore
- Assurdo, perché in tal caso l'algoritmo di ordinamento non sarebbe corretto

Si può far meglio di $O(n \log n)$?

- Abbiamo quindi dimostrato che il **miglior** algoritmo che ordini mediante confronti una sequenza di n elementi è rappresentato da un albero di decisione avente altezza minima $h = \lceil \log_2 (n!) \rceil$ e prestazioni temporali asintotiche, nel caso **pessimo**, $\Omega(h) \equiv \Omega(\log (n!))$
- Non sappiamo quale sia questo algoritmo (né se sia unico) perché **la dimostrazione non è costruttiva**
- Ora, abbiamo già dimostrato in un esercizio che $\log_a (n!) \in \Theta(n \log n) \forall a > 1$, da cui la tesi

Si può far meglio di $O(n \log n)$?

- Abbiamo quindi dimostrato che il **miglior** algoritmo che ordini **mediante confronti** una sequenza di n elementi ha prestazioni temporali asintotiche **$\Omega(n \log n)$** nel caso **pessimo**
 - Questo ovviamente non impedisce l'esistenza di algoritmi più veloci nel caso ottimo o in alcuni casi (es. insertion sort è lineare nel caso ottimo e quando c'è un solo elemento fuori posto)
- Nella dimostrazione abbiamo usato una minorazione, quindi potrebbe rimanere un dubbio
 - Magari esiste un limite inferiore asintotico “più stretto”, cioè asintoticamente maggiore di **$\Omega(n \log n)$**
 - Abbiamo già osservato che ciò non è possibile, perché conosciamo già (almeno due) algoritmi che, nel caso pessimo, sono **$O(n \log n)$** , quindi non possono essere inferiormente limitati da una funzione superiore al loro limite superiore! Dato che cerchiamo un limite inferiore valido per **QUALSIASI** algoritmo... è quello!
 - Questo non esclude che **alcuni** possano essere **$\Omega(n^2)$** (es. Selection Sort)

Fine della
dimostrazione

Bucket Sort

- Per ordinare una lista di **numeri interi appartenenti a un intervallo di tipo $[0, N - 1]$** esiste un algoritmo di ordinamento (**Bucket Sort**) che **non fa confronti**

- Supponiamo che i numeri siano memorizzati in una lista S di dimensione n
 - Possono esserci duplicati, quindi non è vero che sia $n \leq N$, non c'è relazione tra n e N
- Usiamo un array ausiliario A di N numeri interi, tutti inizializzati a zero: questa fase è $\Theta(N)$ [Se N non è noto può essere calcolato in un tempo $\Theta(n)$, cercando il valore massimo, ma occorre fare confronti... un po' una contraddizione...]
- Estraiamo gli n valori da S in modo che ogni estrazione sia $\Theta(1)$ (possibile per ogni tipo di lista perché non interessa l'ordine in cui avvengono le estrazioni)
 - Se il valore estratto è k , incrementiamo di un'unità la cella $A[k]$
 - L'accesso alla cella di A e il suo incremento è un'operazione $\Theta(1)$, perché l'array ausiliario dispone di accesso casuale, quindi tutta la procedura è $\Theta(n)$
 - In pratica, la cella di indice i dell'array ausiliario è un contatore dei valori uguali a i presenti nella lista (se non ci fossero elementi duplicati nella lista, sarebbe sufficiente un array di valori di tipo booleano)
- Esaminiamo le N celle di A **in ordine di indice i crescente**
 - Inseriamo in S (rimasta vuota) un numero $A[i]$ di valori uguali a i
 - Questa fase è $\Theta(n + N)$ [$\Theta(N)$ per leggere A e $\Theta(n)$ per scrivere S]
- **Tempo totale: $\Theta(n + N)$, Spazio aggiuntivo $\Theta(N)$**

Bucket Sort

Bucket Sort

- Più in generale, *Bucket Sort* si può usare per **ordinare coppie aventi numeri interi non negativi come chiave**
- Ciascuna cella dell'array, invece di essere un contatore, è un riferimento a una lista di coppie, inizialmente vuota
 - Queste liste vengono solitamente chiamate *bucket*, da cui il nome dell'algoritmo
- Ogni coppia viene inserita nella lista corrispondente alla cella avente indice uguale alla propria chiave (l'ordine in ciascun bucket non è importante, basta aggiungere in fondo o all'inizio)
- Al termine, si esaminano le celle in ordine di indice crescente e ciascuna lista viene aggiunta alla fine della lista complessiva
- Le prestazioni temporali non cambiano, ma lo spazio aggiuntivo richiesto diventa $\Theta(n + N)$ [$\Theta(n)$ per tutti i bucket]

Considerazioni su bucket-sort

- Ordina sequenze di n coppie chiave/valore con chiavi intere non negative in un tempo $\Theta(n + N)$ e spazio aggiuntivo $\Theta(n + N)$, essendo N la chiave massima
 - Se $N \in O(n)$, allora bucket-sort è $\Theta(n)$
 - Ma **cosa significa $N \in O(n)$?** Significa che, nel dominio di una specifica applicazione, al variare della dimensione della sequenza da ordinare varia proporzionalmente il valore della chiave massima
 - Non è una situazione molto frequente, assai più **N è fisso o, comunque, non dipende da n :** quindi, nella notazione asintotica, N è una costante additiva e si potrebbe eliminare, ma **nella pratica può essere molto influente**

Considerazioni su bucket-sort

- **Spesso N è fisso, non dipende da n :** quindi, nella notazione asintotica, N è una costante additiva e si può eliminare, ma **nella pratica può essere molto influente e va gestita “con accortezza”**
- Esempio: dobbiamo ordinare un array di n valori positivi di tipo **short**, il cui valore massimo è **Short.MAX_VALUE = 32767**
 - Se, nei casi di pratico utilizzo previsti da una particolare applicazione, n è significativamente minore di $N = 32767$, il peso di N , ancorché costante, domina il tempo di esecuzione (e l’occupazione di memoria!)
 - Può essere utile fare una preventiva ricerca del valore massimo **effettivamente** presente nell’array (in un tempo $\Theta(n)$) e usare quello come valore di N : in generale, sarà minore di 32767 e si risparmierà tempo (e spazio di memoria)
 - Naturalmente se n è molto grande tutto ciò non ha senso, infatti asintoticamente N , se non dipende da n , è ininfluente sul tempo di esecuzione

Bucket-sort: Esempio pratico

- Dato un insieme di studenti universitari iscritti a un corso, li si vuole ordinare in base al numero di matricola, M
 - Il numero di matricola è un numero intero: uso bucket-sort
 - Gli studenti di un corso sono un centinaio, diciamo $n = 200$
 - I numeri di matricola di UniPD sono di 7 cifre, quindi $N = 9999999$
 - N non dipende in alcun modo da n , né n dipende da N
 - Nelle prestazioni di tipo $\Theta(n + N)$, N è decisamente dominante
- Quindi... sarebbe meglio ridurre N ! Le prestazioni migliorerebbero, sia per il tempo di esecuzione sia per l'occupazione di memoria
 - In un tempo $\Theta(n)$, calcoliamo il valore minimo, M_{\min} , e massimo, M_{\max} , dei numeri di matricola degli studenti iscritti al corso
 - Invece di usare il numero di matricola M come indice, usiamo una sua trasformazione lineare monotona (che mantiene, quindi, l'ordinamento), $K = M - M_{\min}$
 - Per questo corso, si ha $N = K_{\max} \approx 173000$ (K_{\min} è ovviamente zero, per definizione):
L'ordinamento degli stessi studenti è circa 57 volte più veloce!

Paradigma ricorsivo
«divide and conquer»

Paradigma "divide and conquer"

- A volte si possono risolvere problemi complessi usando il **paradigma ricorsivo *divide and conquer*** (in latino, *divide et impera*), che segue questo schema generale
 - **Caso base:** se la dimensione del problema è inferiore a una determinata soglia, risolvi il problema direttamente, usando qualche strategia (solitamente elementare), e **termina**
 - **Divisione:** suddividi i dati del problema in **due o più insiemi disgiunti** (cioè individua una **partizione** dei dati del problema)
 - **Ricorsione:** risolvi ricorsivamente i sotto-problemi **omogenei** associati ai sottoinsiemi creati dal passo di divisione
 - Si usa una ricorsione multipla, quasi sempre doppia (in ogni caso, la stessa molteplicità della partizione creata nel passo di divisione)
 - **Conquista:** usa le soluzioni dei sotto-problemi per generare (mediante una “fusione”) la soluzione del problema originario

Merge-Sort

Merge-Sort

John von Neumann, 1945

- **Merge-Sort** ordina un array (o una catena, perché **NON necessita dell'accesso casuale**) usando il paradigma *divide and conquer*
- Algoritmo per ordinare una sequenza (array o catena) S di n oggetti (ovviamente appartenenti a un insieme totalmente ordinato)
 - **Caso base:** se S ha meno di 2 elementi, termina (S è ordinata)
 - **Divisione:** estrai tutti gli elementi da S , inserendoli in due sequenze, S_1 e S_2
 - S_1 contiene i primi $\lceil n/2 \rceil$ elementi di S
 - S_2 contiene i rimanenti elementi di S (che sono $\lfloor n/2 \rfloor$)
 - **Ricorsione:** ordina ricorsivamente (e separatamente) S_1 e S_2
 - **Conquista:** “fondi” S_1 e S_2 in una sequenza ordinata, re-inserendo gli elementi in S (che era rimasta vuota)

Serve uno spazio aggiuntivo $\Theta(n)$, vedere nel seguito

Merge-Sort: la “conquista”

- Sappiamo già come funziona la “fusione” di due sequenze ordinate per generare una sequenza ordinata (cfr. corso di Fondamenti di Informatica)
- Rivediamo l’algoritmo se le sequenze A e B sono liste

$S \leftarrow$ sequenza vuota (in merge-sort, è già vuota)

```
while !S1.isEmpty() && !S2.isEmpty()
    if S1.getFirst().element() < S2.getFirst().element()
        S.addLast(S1.removeFirst())
    else
```

Per fondere due **array** ordinati, l’algoritmo è assai simile, con le stesse prestazioni, **usando indici**

```
        S.addLast(S2.removeFirst())
```

```
while !S1.isEmpty() S.addLast(S1.removeFirst()) // uno e soltanto uno di questi
```

```
while !S2.isEmpty() S.addLast(S2.removeFirst()) // due cicli fa qualcosa
```

```
return S
```

□ Prestazioni di questa **fusione**

- Ci sono tre cicli, **durante ogni iterazione di un ciclo viene aggiunto uno e un solo elemento a S** , facendo al massimo un confronto e rimuovendo un elemento da $S1$ o $S2$
- Se, come nelle liste e negli array, tutte le operazioni coinvolte sono $\Theta(1)$, **la fusione che genera una sequenza di dimensione n è $\Theta(n)$** per qualsiasi dimensione di $S1$ e $S2$

Merge-Sort: prestazioni

- Facciamo l'analisi delle prestazioni, usando la consueta strategia, tipica degli algoritmi ricorsivi
- Indichiamo con $T(n)$ il tempo necessario a ordinare con merge-sort una sequenza di dimensione n
 - Ipotizziamo che $n \geq 2$ sia una potenza di due (ma si dimostra in ogni caso)
 - Quindi, $n/2$ è un numero intero e, inoltre: $\lceil n/2 \rceil = \lfloor n/2 \rfloor = n/2$
- Per ordinare n elementi bisogna
 - Costruire le due sottosequenze, in un tempo $\Theta(n)$ [e spazio aggiuntivo $\Theta(n)$]
 - Invocare ricorsivamente merge-sort su due problemi di dimensione $n/2$, in un tempo $2 T(n/2)$
 - Effettuare la fusione, in un tempo $\Theta(n)$
- Quindi

$$T(n) = \begin{cases} b & n < 2 \\ 2T(n/2) + cn & n \geq 2 \end{cases}$$

Prima e terza fase insieme,
usiamo cn per rappresentare
una funzione $\Theta(n)$ qualsiasi

È una funzione in forma
implicita, si dice
“equazione ricorrente” o
“equazione alle ricorrenze”

Vorremmo una funzione
in forma esplicita

$$T(n) = \begin{cases} b & n < 2 \\ 2T(n/2) + cn & n \geq 2 \end{cases}$$

Merge-Sort: prestazioni

- Applichiamo di nuovo l'equazione ricorrente

$$T(n/2) = 2T[(n/2)/2] + c(n/2)$$

$$\begin{aligned} T(n) &= 2T(n/2) + cn \\ &= 2[2T((n/2)/2) + c(n/2)] + cn \\ &= 4T(n/4) + 2c(n/2) + cn \\ &= 2^2 T(n/2^2) + 2cn \end{aligned}$$

- Applicandola k volte, si ottiene quindi $T(n) = 2^k T(n/2^k) + kcn$
 - Ricordiamo che, se k è abbastanza piccolo, **$n/2^k$ è un numero intero**, perché n è una potenza di 2, quindi l'espressione è valida (altrimenti non lo sarebbe, T è definita solo per interi)

- Ora, quando **$n/2^k = 1$** , sappiamo che $T(n/2^k) = b$
- Ovviamente questo avviene quando **$k = \log_2 n$**

Merge-Sort: prestazioni

- Sostituiamo $k = \log_2 n$ in $T(n) = 2^k T(n/2^k) + kcn$

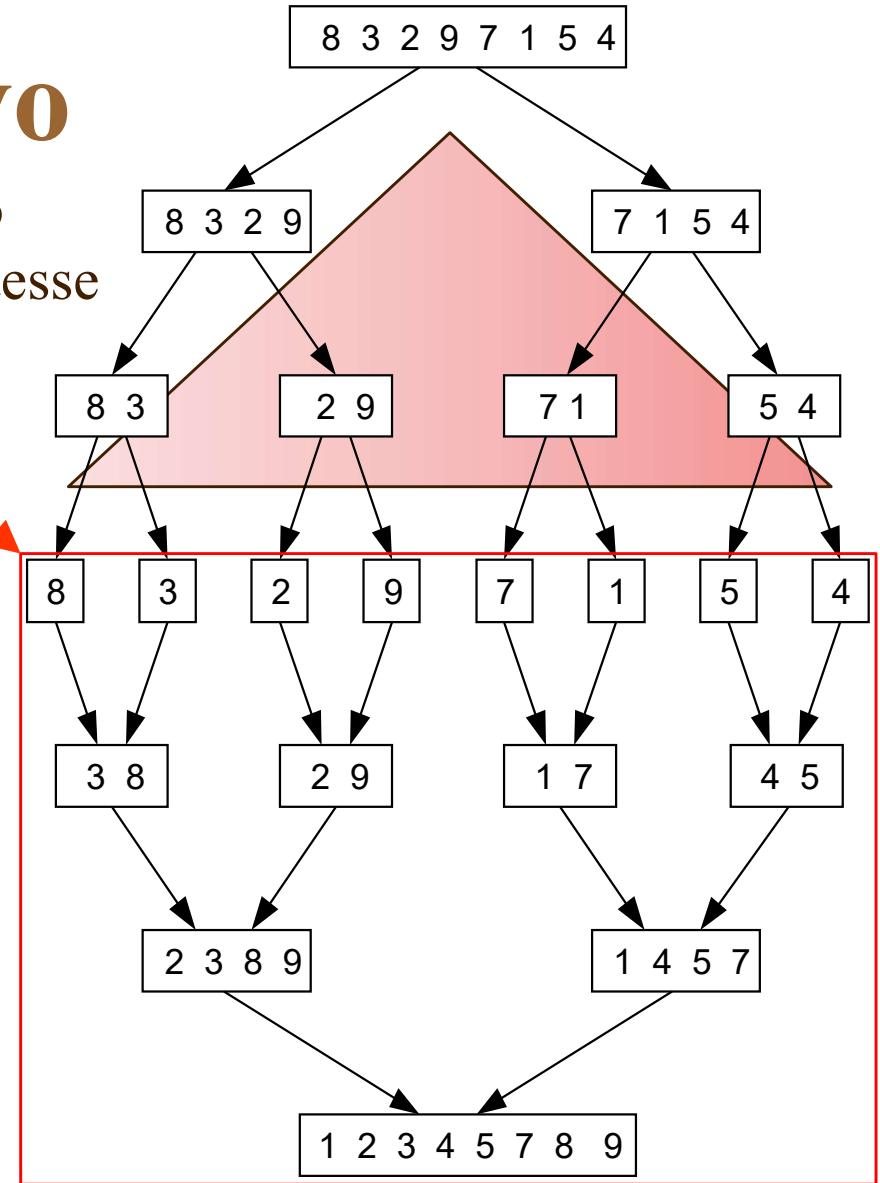
$$\begin{aligned} T(n) &= 2^{\log_2 n} T(n / 2^{\log_2 n}) + (\log_2 n) cn \\ &= nT(n/n) + cn \log_2 n \\ &= nT(1) + cn \log_2 n \\ &= bn + cn \log_2 n \in \Theta(n \log n) \end{aligned}$$

- Se n non è una potenza di 2, bisogna usare in modo opportuno le notazioni $\lfloor \dots \rfloor$ e/o $\lceil \dots \rceil$

Merge-Sort iterativo

- L'algoritmo merge-sort per un array si può realizzare anche in modo **iterativo**, con le stesse prestazioni (un po' più veloce, ma non asintoticamente). **Idea:**

- Ogni elemento dell'array è un array (ordinato) di lunghezza 1
- Ciascuna coppia di elementi consecutivi viene "fusa" in un array ordinato di lunghezza 2 (evitando sovrapposizioni)
- Ciascuna coppia di array di lunghezza 2 consecutivi (ora ordinati) viene fusa in un array ordinato di lunghezza 4 (evitando sovrapposizioni)
- E così via... (con qualche attenzione in fondo all'array se la sua dimensione non è una potenza di 2)

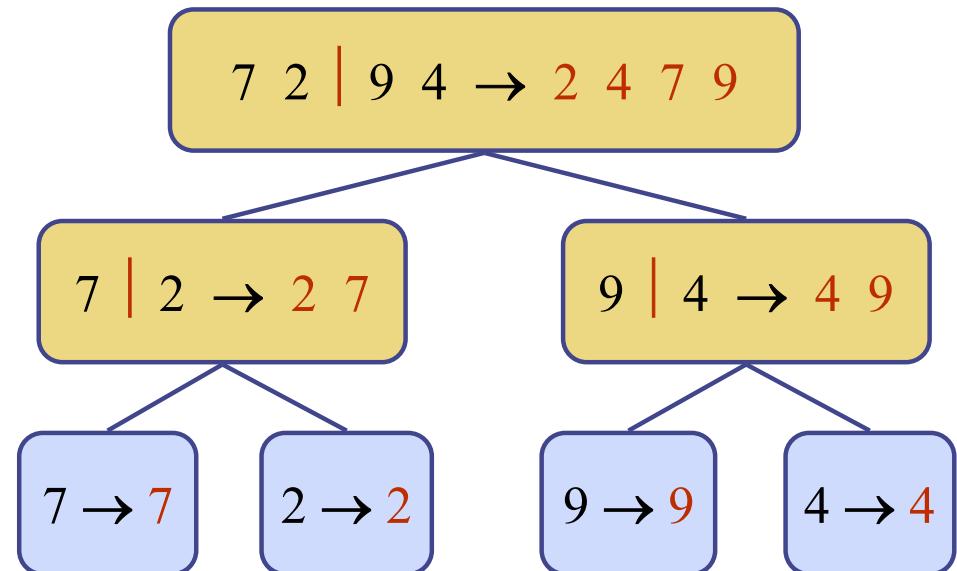


Serve comunque uno spazio ausiliario $\Theta(n)$, perché **le fusioni non si possono fare "sul posto"** in un tempo lineare

Lezione 30

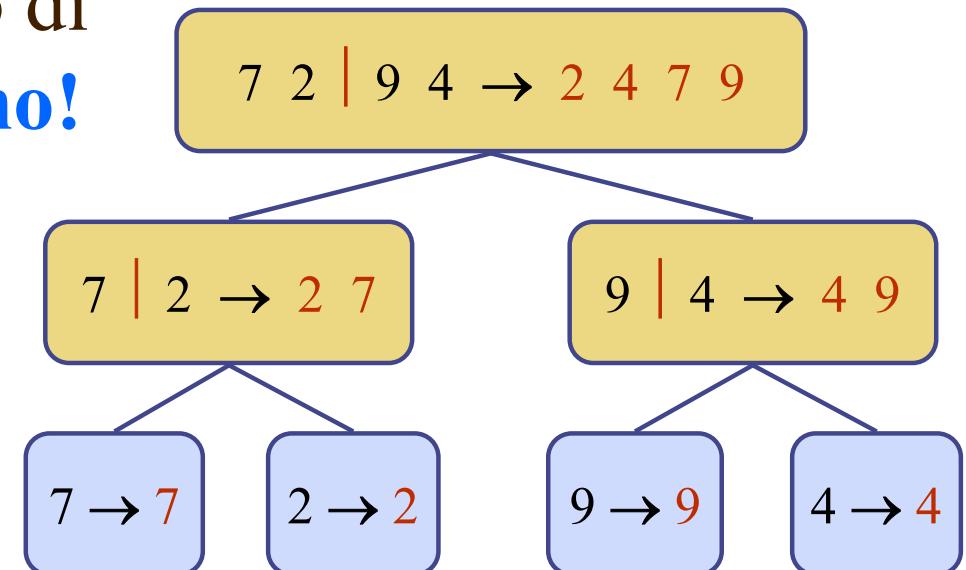
Merge-Sort: albero di esecuzione

- Il funzionamento di merge-sort può essere visualizzato mediante un “**albero d'esecuzione**”, utile anche per altri algoritmi
 - Ogni nodo rappresenta una delle invocazioni ricorsive e contiene sia il problema da risolvere (in cui evidenziamo la suddivisione) sia il problema risolto (la sequenza ordinata)
 - È un albero binario: ogni invocazione ricorsiva scende verso uno dei figli e dopo ogni fusione si risale nel genitore
 - **Attenzione: non** è una struttura usata dall'algoritmo, serve solo a visualizzarne il flusso d'esecuzione
 - **I nodi NON sono tutti contemporaneamente attivi**, le invocazioni attive sono sempre **un solo percorso di nodi di livelli consecutivi**, contenente la radice, e i loro fratelli, quindi uno spazio $\Theta(n)$ [ragionare autonomamente...]
 - Attenzione: n è la dimensione della sequenza da ordinare, non dell'albero...



Merge-Sort: albero di esecuzione

- Usiamo l'albero per “visualizzare” le prestazioni
 - Attenzione: n è la dimensione della sequenza da ordinare, non dell'albero...
 - **Ad ogni livello**, occorre un tempo $\Theta(n)$ per le suddivisioni e le fusioni, perché la somma del numero di elementi presenti nei nodi di ciascun livello è sempre n
 - Si può dimostrare che **questo albero ha altezza $\Theta(\log n)$**
 - Ogni volta si divide per due... come in binarySearch...
 - Ma non abbiamo bisogno di dimostrarlo, **lo deduciamo!**
Perché abbiamo già dimostrato che le prestazioni sono **$\Theta(n \log n)$!!**



Quick-Sort

Quick-Sort

- Algoritmo che ordina una sequenza (array o catena) S di n oggetti (di un insieme totalmente ordinato) usando ***divide and conquer***
 - **Caso base:** se S ha meno di 2 elementi, termina (S è ordinata)
 - **Divisione:** scegli un elemento $x \in S$ (detto *pivot*) ed estrai tutti gli elementi da S , inserendoli in tre sequenze, che (per come sono qui definite) risultano essere **una partizione** di S :
 - L (*less than*) contiene gli elementi minori di x
 - E (*equals to*) contiene gli elementi uguali a x
(quindi solo x se gli elementi di S sono tutti distinti)
 - G (*greater than*) contiene gli elementi maggiori di x
 - **Ricorsione:** ordina ricorsivamente L e G (E è già ordinata)
 - **Conquista:** re-inserisci gli elementi in S (che era rimasta vuota), prima quelli di L , poi quelli di E , infine quelli di G

La fase di divisione sembra richiedere uno spazio aggiuntivo lineare, ma vedremo che si può fare anche “sul posto” (*in place*), cioè con spazio aggiuntivo $\Theta(1)$

Quick-Sort: scelta del pivot

- Come pivot si può usare, ad esempio, l’elemento che si trova in una posizione prefissata, a cui si possa accedere in un tempo $\Theta(1)$: solitamente **l’ultimo elemento** della sequenza (array o lista)
- Ciascuna fase di “divisione” e di “conquista” è (ovviamente) lineare
 - Indipendentemente dalla scelta del pivot
- L’analisi delle prestazioni si può fare usando l’albero di esecuzione
- **Nel caso pessimo** l’altezza dell’albero di esecuzione è $n - 1$
 - Infatti, se la sequenza era già ordinata, **ad ogni passo** tutti gli elementi di S , tranne il pivot, vanno a finire in L , quindi **la dimensione del problema non viene dimezzata, bensì ridotta di un’unità**
- **Le prestazioni nel caso pessimo sono, quindi, $\Theta(n^2)$**
 - Inoltre, si dimostra che, **se si preleva il pivot da una posizione prefissata della sequenza** (prima, ultima, seconda, centrale, ecc.), è sempre possibile progettare una sequenza “pessima”, che renda quick-sort quadratico

Quick-Sort: scelta del pivot ottimo

- Che prestazioni si ottengono scegliendo **il pivot ottimo?**
 - Quick-sort opera mediante confronti, quindi è $\Omega(n \log n)$.
Facendo la scelta ottima, riuscirà a essere $O(n \log n)$?
- Ciascuna fase di divisione/conquista per una lista di lunghezza k , come già detto, è $\Theta(k)$
- Usando l'albero di esecuzione (la cui struttura dipende dalla scelta del pivot), l'analisi delle prestazioni è uguale a quella vista per merge-sort, dove il tempo richiesto era **$\Theta(n)$ per ogni livello**
 - Il caso ottimo si ha quando l'albero di esecuzione ha **altezza minima**
 - Tale altezza è $\Omega(\log n)$, non può essere più bassa, altrimenti l'algoritmo non sarebbe $\Omega(n \log n)$ (mentre sappiamo che lo deve essere)
- Se **ad ogni suddivisione** le due sotto-sequenze hanno dimensioni che differiscono tra loro al massimo di un'unità (**come in merge-sort**), l'albero di esecuzione ha altezza minima, $O(\log n)$

Quick-Sort: scelta del pivot ottimo

□ Quindi, cosa deve fare il pivot ottimo?

- Deve, ad ogni passo, rendere minima la differenza tra le dimensioni di L e G , cioè deve essere l'elemento **mediano**
 - **Attenzione, in generale non è l'elemento di “valore medio”...**
- **L'elemento mediano di una sequenza è quello che, se la sequenza fosse ordinata, occuperebbe “la posizione centrale”**
 - Per evitare ambiguità nel caso in cui la dimensione sia un numero pari, **definiamo come elemento mediano** di una sequenza ordinata di dimensione n **l'elemento che si trova nella posizione di indice $\lfloor n/2 \rfloor$**
- La **selezione del pivot** fa parte della fase di “divisione”, quindi, perché quick-sort sia complessivamente $\Theta(n \log n)$, ciascuna ricerca dell'elemento mediano tra k elementi deve essere $O(k)$, non è necessario che sia $\Theta(1)$, tanto la divisione ha già una fase $\Theta(k)$

Quick-Sort: scelta del pivot ottimo

- Si riesce a **individuare l'elemento mediano di una sequenza non ordinata** di dimensione n in un tempo $O(n)$?
- Si tratta di un problema classico che, nella sua forma più generale, prende il nome di **selezione**
 - La **selezione dell'elemento di ordine k** (detto anche "elemento k -esimo") di una sequenza S non ordinata di dimensione n è l'individuazione dell'elemento di S che avrebbe **rango k** nella sequenza S ordinata
 - La ricerca dell'elemento mediano equivale, per definizione, alla selezione dell'elemento di rango $\lfloor n/2 \rfloor + 1$
- **Esiste un algoritmo**, piuttosto complesso (che non vediamo), **che risolve il problema della selezione in un tempo $O(n)$ nel caso pessimo** (per qualsiasi valore di k)

Quick-Sort: scelta del pivot ottimo

- Quindi, usando a ogni passo **l'elemento mediano come pivot**, quick-sort ha prestazioni $\Theta(n \log n)$, come merge-sort
- Purtroppo, **l'algoritmo di selezione citato ha** prestazioni lineari, ma con **una costante moltiplicativa molto elevata**
 - Quick-sort risulta essere più lento di merge-sort!!!
 - **Allora, a cosa serve quick-sort?**
 - È utile perché esiste **un algoritmo probabilistico di scelta del pivot** che rende quick-sort **$O(n \log n)$ nel caso medio, con una costante moltiplicativa migliore** di quella di merge-sort
 - Ma è *quadratico* nel caso pessimo

Quick-Sort: scelta **casuale** del pivot

- Ad ogni divisione, scegliamo il pivot tra tutti gli elementi di una sequenza di dimensione k **in modo casuale, con distribuzione di probabilità uniforme**
 - **Ogni elemento ha la stessa probabilità** (uguale a $1/k$) di essere scelto
- Obiettivo: **dimostrare che**, con questo algoritmo di scelta del pivot, **quick-sort ha prestazioni medie $O(n \log n)$** , anche se quadratiche nel caso pessimo
 - Non affrontiamo compiutamente la teoria delle probabilità, ma, avendo le variabili in gioco **distribuzione di probabilità uniforme**, ne usiamo il “valore medio” anziché il “valore atteso”: è ragionevole
- Utilità: **la costante di tempo dell'algoritmo risulta essere migliore** di quella di merge-sort
 - Per applicazioni in cui il limite nel caso pessimo sia importante, è comunque preferibile usare merge-sort

Indichiamo con $|X|$ la lunghezza della sequenza X

CENNI

Dimostrazione: scelta casuale del pivot

- Scegliamo il pivot tra tutti gli elementi di una sequenza di dimensione k **in modo casuale, con distribuzione di probabilità uniforme**
- Sappiamo che i pivot migliori sono quelli che generano sequenze L e G di dimensioni simili

- Scelto un pivot a caso, lo definiamo “**buono**” se genera L in modo che $k/4 \leq |L| < 3k/4$, cioè **|L| "vicino" a $k/2$**

Ipotesi:
elementi
distinti,
ma si
dimostra
anche con
duplicati

- Ovviamente anche la dimensione di G sarà compresa negli stessi limiti, perché $|L| + |G|$ è uguale a k (in realtà $k - 1$, ma non importa)
- Essendo la scelta del pivot casuale, i valori $|L| = 0, |L| = 1, \dots, |L| = k - 1$ sono tra loro equiprobabili
 - Dato che l'intervallo di valori “buoni” di $|L|$ contiene (circa) **la metà dei valori possibili ed equiprobabili di $|L|$** , la probabilità di scegliere casualmente un pivot “buono” è (circa) **il 50%, cioè 1/2**

Dimostrazione: scelta casuale del pivot

- La probabilità di scegliere casualmente un pivot “**buono**” è **il 50%, cioè 1/2**
- Consideriamo un nodo v dell’albero di esecuzione di quick-sort che abbia profondità d : è frutto di d fasi di “divisione”, quindi la dimensione della sequenza S_v in esso contenuta è uguale a
 - $|S_v| = f_1 f_2 \dots f_d n$

dove f_i è il rapporto tra la dimensione della sotto-sequenza dell’antenato di v che si trova al livello i e la dimensione della sotto-sequenza del genitore di quell’antenato (il “rapporto di divisione”, compreso tra zero e uno)

- Se f_i è compreso tra $1/4$ e $3/4$, è frutto della scelta di un pivot “buono” nella i -esima fase di divisione: **f_i è “buono” con probabilità 1/2**
- Quindi, per le leggi della probabilità, ci saranno **mediamente** metà (cioè $d/2$) rapporti di divisione “buoni” e metà “cattivi”

- $|S_v|_m \leq (3/4)^{d/2} 1^{d/2} n = (3/4)^{d/2} n$
- $|S_v|_m \equiv$ Dimensione media di S_v

I $d/2$ coefficienti “buoni” vengono maggiorati da $3/4$, quelli “cattivi” da 1

Dimostrazione: scelta casuale del pivot

- Dimensione media di $S_v \equiv |S_v|_m \leq (3/4)^{d/2} n$
- Ci interessa, **nel caso medio**, l'altezza h dell'albero di esecuzione: è la profondità dei suoi nodi più profondi
 - I nodi più profondi di un albero sono necessariamente foglie
 - Se v è una foglia, allora $|S_v| = 1$ (caso base di quick-sort)
 - Se v è una foglia di profondità massima, allora $d = h$
- Quindi, **nel caso medio**, scegliendo ogni volta il pivot a caso
 $1 \leq (3/4)^{h/2} n \Rightarrow n \geq (4/3)^{h/2} \Rightarrow \log_{4/3} n \geq \log_{4/3} (4/3)^{h/2}$
 $\Rightarrow \log_{4/3} n \geq h/2 \Rightarrow h \leq 2 \log_{4/3} n \Rightarrow h \in O(\log n)$
- Come volevasi dimostrare, questa versione di quick-sort è **$O(n \log n)$** , come merge-sort, **nel caso medio**
- Nel caso pessimo tutte le scelte casuali saranno "cattive", facendo diminuire la dimensione di una sola unità: $\Theta(n^2)$

Randomized Quick-Sort

- Abbiamo “dimostrato” che l’algoritmo “quick-sort con scelta casuale del pivot” (detto **randomized quick-sort**) è
 - **$O(n \log n)$ nel caso medio**, come merge-sort e heap-sort, ma (vedi in seguito) **con una costante moltiplicativa minore, quindi migliore**
 - **$\Theta(n^2)$ nel caso pessimo**, mentre merge-sort e heap-sort sono sempre log-lineari, cioè $O(n \log n)$
- A favore di **randomized quick-sort** si dimostra anche (**molto faticosamente!** vedi Cormen *et al.*) che le prestazioni sono **"molto probabilmente"** $O(n \log n)$, in particolare lo sono con probabilità non inferiore a $1 - 1/n$
 - Quindi, **al crescere di n , è sempre più probabile che randomized quick-sort **NON** si trovi nel caso pessimo:** buona cosa, perché la differenza di prestazioni tra un algoritmo $\Theta(n^2)$ e uno $O(n \log n)$ è più significativa quando n è grande

Quick-sort: divisione “sul posto”

Algoritmo
interessante anche
"al di fuori" di
quick-sort

- In quick-sort, la fase di “divisione” si può effettuare “sul posto”, in un tempo $\Theta(n)$, lasciando le sottosequenze L , E e G (con L e G da ordinare) all’interno della sequenza originaria S , **se è un array**
- In questo modo
 - Serve uno spazio aggiuntivo $\Theta(1)$ e non $\Theta(n)$
 - La fase di “conquista” non deve fare niente!!
 - Le prestazioni asintotiche non cambiano, ma **la costante di tempo migliora**
- Sia S l’array esaminato da uno dei passi di quick-sort
 - Scegliamo il pivot e lo scambiamo con l’ultimo elemento
- Obiettivo: la parte iniziale dell’array conterrà L , seguita da G , seguita da E
 - Alla fine della divisione la sequenza E verrà inserita tra L e G (basta scambiare gli elementi di E con i primi elementi di G , tanto G non è ordinata)

Normalmente quick-sort
usa la divisione sul posto!

Quick-sort: divisione “sul posto”

- Ipotesi semplificativa: **gli elementi dell’array sono tutti distinti**, quindi E contiene soltanto il pivot
- Usiamo due indici
 - un indice, $left$, che parte dal primo elemento e **aumenta**
 - un indice, $right$, che parte dal penultimo elemento e **diminuisce** (nell’ultimo c’è il pivot)
- Spostiamo $left$ finché gli elementi esaminati sono minori del pivot e spostiamo $right$ finché gli elementi esaminati sono maggiori del pivot
 - Quando ci fermiamo, se $left < right$ allora scambiamo tra loro i due elementi che ci hanno fatto fermare: a questo punto, certamente sia $left$ sia $right$ potranno fare almeno un altro passo ciascuno e ripetiamo la procedura; altrimenti abbiamo terminato la suddivisione e $left$ è la lunghezza di L
 - Ogni elemento viene letto una sola volta ed eventualmente scritto una volta (se è coinvolto in uno scambio)
 - Cosa succede se ci sono, invece, elementi uguali al pivot? Provare...

Quick-sort: divisione “sul posto”

- L’algoritmo (lineare) di divisione sul posto di un array "attorno a un pivot" è interessante anche come algoritmo a sé stante (come l’algoritmo di fusione di due liste ordinate...)
- Si noti che l’algoritmo di divisione sul posto funziona correttamente anche se il pivot NON appartiene all’array...
 - semplicemente la porzione E sarà vuota
 - Da tenere presente per alcune applicazioni...
- In effetti, anche nella formulazione di Quick-Sort non sarebbe richiesto che il pivot appartenga all’array: la porzione E sarà vuota, ma le prestazioni asintotiche non cambiano
 - Anche se, in effetti, non c’è necessità di scegliere un pivot che non appartenga all’array
 - Però può avere senso se si riesce a "stimare" il valore mediano di un array... non importa usare il valore esatto!

Quick-sort: divisione “sul posto” in una catena

- Se la sequenza da ordinare è una catena (e non un array), l’algoritmo di divisione sul posto si può usare ugualmente, con le stesse prestazioni asintotiche, SOLTANTO se si tratta di una catena bidirezionale (cioè "a doppia concatenazione")
- Gli indici *left* e *right* diventano "posizioni", così come la posizione del *pivot*

Quick-sort vs. Merge-sort

- Senza fare una dimostrazione rigorosa, cerchiamo di capire perché quick-sort è più veloce di merge-sort (anche se non asintoticamente), contando soltanto gli accessi a singoli elementi dell'array
- Usiamo l'albero di esecuzione, **nell'ipotesi che quick-sort usi sempre il pivot ottimo**: in questo caso, l'albero di esecuzione è sostanzialmente identico a quello di merge-sort (a parte il fatto che, ad ogni suddivisione, quick-sort elimina il pivot, che non fa parte di nessuno dei due sotto-array L e G)
- Facciamo l'ipotesi semplificativa che n (dimensione dell'array da ordinare) sia una potenza di 2 e che, quindi, l'albero di esecuzione di merge-sort abbia altezza $h = \log_2 n$ (quello di quick-sort avrà "circa" la stessa altezza, trascuriamo il fatto che, in realtà, il pivot viene via via escluso dall'insieme degli array da ordinare e, quindi, in realtà l'altezza sarà "un po'" inferiore se n è abbastanza grande)

Quick-sort vs. Merge-sort

- Per **merge-sort**, a ogni livello dell'albero di esecuzione
 - La somma delle lunghezze di tutti gli array presenti è n
 - Per fare le divisioni servono $2n$ accessi (ignoriamo il fatto che, in Java, c'è l'inutile inizializzazione implicita di tutti i nuovi array)
 - Per fare le fusioni servono $2n$ accessi per trasferire i dati, più gli accessi necessari per fare i confronti
 - Durante la fusione di due array ordinati di dimensione $k/2$ per riempire un array di dimensione k serve un numero di confronti compreso tra $k/2$ (caso ottimo, quando uno dei due array va a riempire la prima metà dell'array di destinazione) e $k - 1$ (caso pessimo, quando il ciclo principale della fusione termina lasciando un solo elemento in uno dei due array, per semplicità diciamo che i confronti siano k)
 - Quindi, per fare tutti i confronti necessari alle fusioni effettuate a un livello dell'albero di decisione serve un numero di accessi variabile tra n e $2n$ (due accessi per ogni confronto)
 - Accessi A per un livello: $5n \leq A \leq 6n$
- Per l'intero ordinamento, **$5n \log_2 n \leq A \leq 6n \log_2 n$** (caso ottimo: array ordinato)

Quick-sort vs. Merge-sort

□ Per **quick-sort**, a ogni livello dell'albero di esecuzione

- La somma delle lunghezze di tutti gli array presenti è n (in realtà diminuisce almeno di un'unità per ogni livello, ma ignoriamo questo aspetto)
- **Utilizzando l'algoritmo che opera sul posto**, durante la divisione di un array di dimensione k serve un numero di confronti uguale a $k - 1$ (per semplicità diciamo che i confronti siano k): ogni elemento viene confrontato con il pivot (che può stare in una variabile, quindi servono $k - 1$ accessi per i confronti)
- Utilizzando l'algoritmo che opera sul posto, per fare tutte le divisioni a un livello dell'albero di decisione serve
 - Un numero di accessi uguale a n per fare n confronti (in realtà, un po' meno...)
 - Un numero di accessi per fare gli scambi compreso tra 0 (caso ottimo, non serve nessuno scambio) e $2n$ (caso pessimo, ogni elemento è coinvolto in uno scambio, quindi ci sono $n/2$ scambi, ognuno dei quali richiede 4 accessi)
- Le fasi di conquista sono "vuote" !
- Accessi A per un livello: $n \leq A \leq 3n$

merge-sort

$$5n \log_2 n \leq A \leq 6n \log_2 n$$

□ Per l'intero ordinamento, **$n \log_2 n \leq A \leq 3n \log_2 n$** (caso ottimo: array ordinato)

□ **Quick-sort fa un numero di accessi compreso tra la metà e un quinto degli accessi necessari a merge-sort!!**

Quick-sort vs. Merge-sort

- Può essere utile confrontare i due algoritmi anche in base al solo numero di confronti necessari per l'ordinamento
 - Importante quando fare un confronto richiede un tempo (molto) maggiore di quello richiesto per un accesso all'array: ad esempio, quando si ordina un array di stringhe, fare un confronto significa confrontare due stringhe... quando invece si ordinano array di tipi primitivi, fare un confronto richiede un tempo decisamente inferiore a quello richiesto per un accesso (dopo aver fatto gli accessi necessari a portare i valori da confrontare all'interno della CPU)
- Come abbiamo appena visto usando l'albero di esecuzione, il numero di confronti, C , è
 - Quick-sort: $C = n \log_2 n$
 - Merge-sort: $n/2 \log_2 n \leq C \leq n \log_2 n$
 - Caso pessimo: stessi confronti di quick-sort
 - Caso ottimo: metà confronti rispetto a quick-sort

Ordinamento e Java

- La libreria standard di Java usa l'algoritmo **merge-sort** in alcuni metodi (statici):
 - **java.util.Arrays.sort(...)** per array di
 - Tipi di dati che realizzano **Comparable**
 - Qualsiasi cosa, fornendo un **java.util.Comparator**
 - Per tipi di dati primitivi (cioè "non oggetti") il metodo usa invece **quick-sort** perché è più veloce di merge-sort (ma effettua un maggior numero di confronti, quindi è peggiore quando fare i confronti può essere "oneroso", vedi sopra...)
- **java.util.Collections.sort(...)** per strutture che realizzano **java.util.List** senza poter sfruttare l'eventuale accesso casuale
 - **java.util.ArrayList**
 - **java.util.LinkedList**
 - ...

Lezione 31

Hashing

Mappa: un caso speciale!

- Se (caso non così raro!) una mappa usa come chiavi **numeri interi compresi nell'intervallo $[0, N - 1]$,** **con N noto a priori**, si può usare una struttura speciale
- Un array A di dimensione N (un array di tipo “valore”) che memorizza nella cella $A[k]$ il **valore** (non la coppia) corrispondente alla chiave k
 - $A[k]$ contiene **null** se e solo se non esiste nella mappa un valore associato alla chiave k (quando la mappa è vuota, tutte le celle dell’array contengono **null**)

chiave = 0	valore0
null	
chiave = 2	valore2
chiave = 3	valore3
null	
null	

Tabella

□ Una mappa con chiavi numeriche intere comprese in un intervallo $[0, N - 1]$ viene detta *tabella* o tavola (*table* o, anche, *search table*)

□ In una tabella (caso speciale di mappa), **tutte le operazioni, get, put e remove, sono $\Theta(1)$**

- La chiave è direttamente l'indice nell'array!
- Non occorre mai fare ricerche, né lineari né per bisezione!

Svantaggio:

i metodi **keys** e **values** sono $\Theta(N)$ anziché $\Theta(n)$, perché devono scandire tutta la tabella, anche le celle vuote (in realtà sono $O(N)$ e $\Omega(n)$ se, come probabile, la tabella memorizza n , per avere **size()** in tempo costante: interrompo la scansione dell'array quando ho trovato n valori)

chiave = 0	valore0
	null
chiave = 2	valore2
chiave = 3	valore3
	null
	null

Tabella: Un problema

- La tabella non utilizza la memoria in modo efficiente
 - **L'occupazione di memoria** richiesta per contenere **n** coppie non dipende da **n**
 - come invece avviene per molti altri ADT, che hanno un'occupazione di memoria lineare in funzione di **n**ma **dipende linearmente dal valore della chiave massima, $N - 1$** , che **non ha**, in generale, **alcuna relazione con il numero, n , di coppie**
 - **Occupazione di memoria: $\Theta(N)$**
- Si definisce **fattore di riempimento** (**load factor**, λ) della tabella il numero di coppie contenute nella tabella diviso per la dimensione della tabella stessa (cioè per il valore della chiave massima aumentato di uno), **n/N**
 - Purtroppo, una tabella può avere un fattore di riempimento molto basso (al massimo, ovviamente, $\lambda = 1$)

Tabella ridimensionabile

- Se il valore della chiave massima, $N - 1$, non è noto a priori, la tabella può avere ***dimensione variabile***, cioè può utilizzare un ***array di dimensione crescente*** quando sia necessario
 - L'operazione di ***inserimento*** richiede un tempo $\Theta(N)$ ogni volta che è necessario un ridimensionamento
 - In questo caso, però, ***non si può utilizzare l'analisi ammortizzata*** perché non si può prevedere quali siano le chiavi fornite dall'utente negli inserimenti
 - ***Non è più vero che il ridimensionamento avviene “una volta ogni tanto”, può avvenire anche TUTTE LE VOLTE***
 - Es. vengono inserite le chiavi in ordine crescente, ad esempio 10, 100, 1000, 10000
 - In una tabella ridimensionabile, le prestazioni dell'inserimento sono ***O(N)*** anche in media

Tabella

- Abbiamo visto che, se una mappa usa **come chiavi numeri interi compresi nell'intervallo $[0, N - 1]$** , con N noto a priori, si può usare una **tabella**
 - Le prestazioni sono ottime per i metodi **get, put e remove**
 - Problema: occupazione di memoria non lineare in n ma $\Theta(N)$
 - Problema: metodi **keys** e **values** non lineari in n ma $O(N)$
 - Riusciamo a estendere il campo di utilizzo di questa struttura?
Ci sono **due problemi**
 - **Non sempre le chiavi sono numeri interi**
 - Se sono numeri interi ma non appartengono a un intervallo di tipo **$[0, N - 1]$** , bensì **$[a, b]$** , basta fare una traslazione, sottraendo **a** a tutte le chiavi e usando una cosiddetta "chiave ridotta", cioè una chiave che è stata "costretta" ad appartenere all'intervallo $[0, N - 1]$
 - Vorremmo poter **controllare l'occupazione di memoria, possibilmente rendendola lineare** in funzione della dimensione della mappa (cioè n)

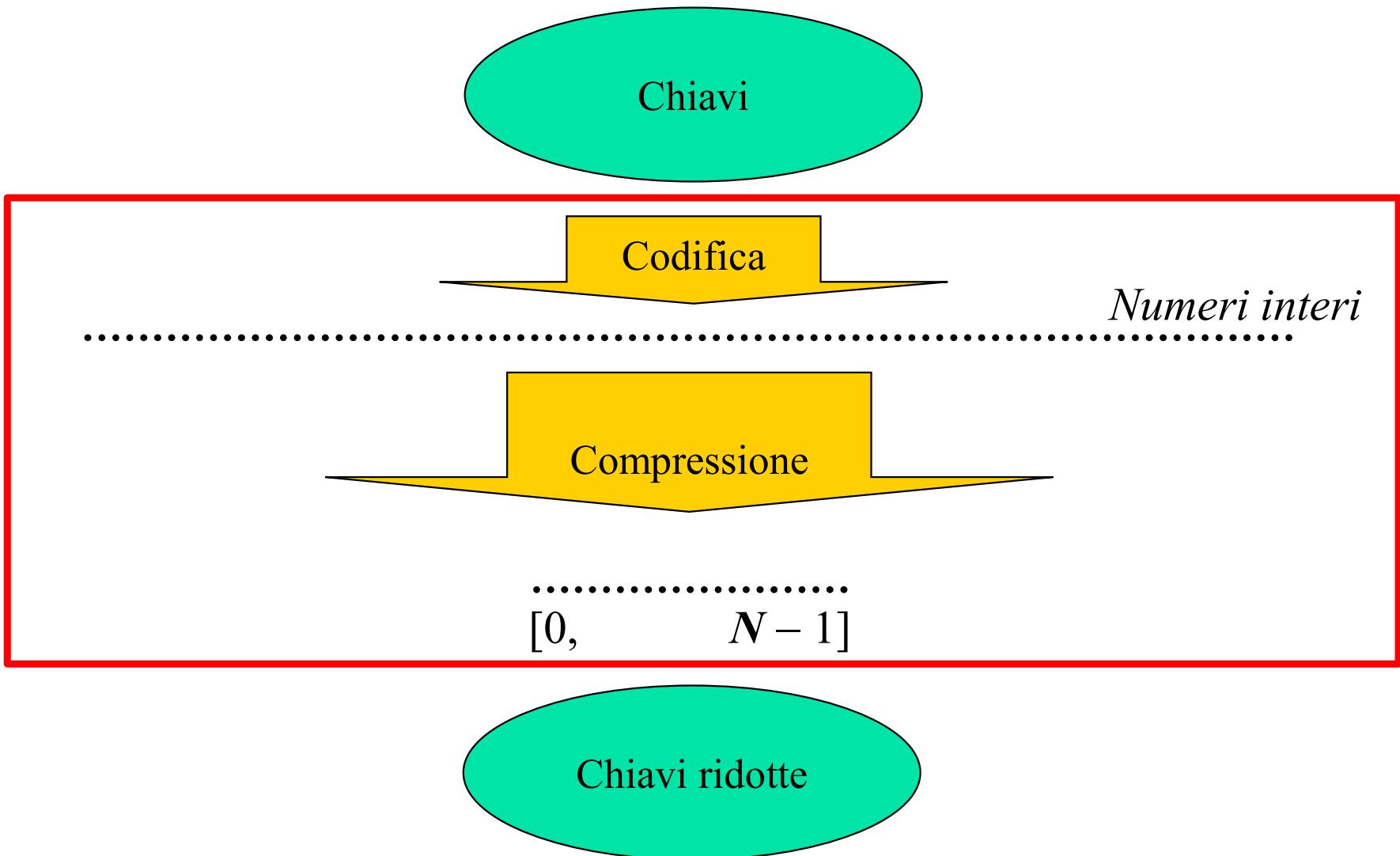
Tabella con chiavi non intere?

- Se una mappa usa **chiavi che non sono numeri interi**, definiamo una funzione di trasformazione che abbia
 - come **dominio** l'insieme delle chiavi della mappa
 - come **codominio** l'insieme degli indici (cioè numeri interi) validi per accedere a una tabella di dimensione N
- Una simile funzione prende genericamente il nome di **funzione di hash**
 - *to hash* significa “pasticciare”, “mescolare”, “fare il polpettone” ! ☺
 - Il valore prodotto dall'applicazione della funzione di hash a una chiave si chiama **chiave ridotta**
- **Con un'opportuna funzione di hash, qualunque mappa può essere realizzata mediante tabella**

Funzione di hash

- Il compito di una funzione di hash può essere decomposto in due funzioni, applicate una dopo l'altra
- Chiamiamo *funzione di codifica (di hash)* la prima: genera un *codice (di hash)* a partire da una chiave e ha
 - come **dominio** l'insieme delle chiavi della mappa
 - come **codomnio** l'insieme dei numeri interi (che qui approssimeremo con l'insieme dei valori rappresentati dal tipo di dato **int** in Java, perché in Java gli indici in un array devono essere di tipo **int**)
- Chiamiamo *funzione di compressione* la seconda: genera una *chiave ridotta* a partire da un codice di hash e ha quindi
 - come **dominio** l'insieme dei codici di hash (che, come appena detto, sono numeri interi) generati dalla funzione di codifica (applicata a un particolare insieme di chiavi)
 - come **codomnio** l'insieme degli indici validi per accedere a una tabella di dimensione N

Funzione di hash



Funzione di codifica

- Se l’insieme delle chiavi è di “piccole” dimensioni (cardinalità C) e i suoi valori sono noti a priori, la funzione di codifica può essere semplicemente una struttura di tipo **if...else if...**
- Attenzione, però, perché in questo modo le prestazioni del calcolo della funzione di hash per una data chiave sono $O(C)$
 - Quindi hanno prestazioni $O(C)$ tutti i metodi della mappa, ciascuno dei quali richiede il calcolo della funzione di hash della chiave in esame

Funzione di codifica biunivoca?

- Prima di vedere alcune possibili funzioni di codifica più generali, ci poniamo un problema
 - In Java, gli indici negli array sono rappresentati con il tipo `int` e devono essere numeri non negativi quindi la quantità massima di indici diversi disponibili è
 - `1 + Integer.MAX_VALUE`
 - Se l'insieme delle chiavi ha cardinalità maggiore di questo valore, il problema non è risolvibile
 - Anche se usiamo `long`, ci sarà un limite
 - **In generale la funzione di codifica non è biunivoca: chiavi diverse possono avere lo stesso codice di hash**
 - Il fatto di NON porre la biunivocità della funzione di codifica come requisito/vincolo ci sarà utile anche per altri motivi

Funzione di codifica: collisioni

- In generale, **la funzione di codifica non è biunivoca: chiavi diverse possono avere lo stesso codice di hash**
- Quando ciò accade, si parla di “collisione” nella tabella
 - Se due chiavi diverse hanno lo stesso codice di hash, necessariamente avranno anche la stessa chiave ridotta
 - La funzione di compressione “non può fare miracoli”... ☺
 - Come può allora funzionare la tabella? Cosa succede se cerco di inserire due coppie aventi proprio quelle due chiavi?
Sono coppie con chiavi diverse, devono trovar posto entrambe nella mappa, ma non possono stare nella stessa cella dell’array che realizza la tabella
 - Vedremo come **risolvere le collisioni** (ad esempio **con un array di liste, chiamate *bucket***...), per il momento cerchiamo **“buone”** funzioni di codifica che **minimizzino il numero di collisioni!**
 - Una funzione di codifica che non genera collisioni si dice **ottima** (aggettivo che non riguarda le prestazioni spazio/tempo)

Funzioni di codifica ottime

- Se le chiavi originarie appartengono a un sottoinsieme dell'insieme rappresentato dal tipo `int`, basta usare la funzione identità!
 - Se il tipo di dato è solo "teoricamente" diverso (es. `byte`, `short`, `char`), basta fare un cast (implicito)
 - Se il tipo di dato è `float`, si può ancora usare una strategia simile, usando la sua rappresentazione in memoria (che è costituita da 32 bit) come se fosse un `int` (che ha 32 bit)
 - Esiste addirittura il metodo:
`static int Float.floatToIntBits(float x)`
- Sono (ovviamente?) funzioni ottime
 - Non manipolano i bit: valori diversi hanno rappresentazioni binarie diverse, quindi codici di hash diversi

Funzioni di codifica: somma di componenti

- Se le chiavi originarie sono di tipo **double**
 - Per prima cosa le convertiamo in **long** in modo ottimo:
`static long Double.doubleToLongBits(double x)`
- Se le chiavi sono di tipo **long** (o **double**), ovviamente non esistono funzioni ottime, cioè biunivoche
(il dominio ha cardinalità maggiore del codominio!)
 - Facendo semplicemente un cast a **int**, ignoriamo completamente le informazioni contenute nei 32 bit più a sinistra: funziona, ma probabilmente non è molto “buona”
 - Si vede **sperimentalmente** che è meglio “mescolare” le due porzioni da 32 bit, ad esempio con una funzione di questo tipo

```
static int hashCode(long i)
{   return (int)((i >> 32) + (int)i);
}
```

Funzioni di codifica: somma di componenti

- In generale, si può estendere il concetto visto per le variabili di tipo **long** (e **double**) a chiavi di qualsiasi dimensione, scomponendo la relativa rappresentazione binaria in blocchi di 32 bit e sommandoli
 - **Funzione di codifica “a somma di componenti”**
- Un caso frequente: le chiavi sono stringhe
 - Le stringhe **sembrano** un caso di studio interessante per applicare questa funzione di codifica: sono già “separate” in blocchi (da 16 bit), i loro singoli caratteri!
 - “Sommiamo” tra loro i singoli caratteri e otteniamo un codice di hash per la stringa

Funzioni di codifica: somma di componenti

- “Sommiamo” tra loro i singoli caratteri di una stringa e otteniamo un codice di hash per la stringa stessa
 - Sfortunatamente, l’evidenza sperimentale mostra che, anche per effetto della proprietà commutativa dell’addizione, **questa funzione di codifica tende a generare molte collisioni**, perché genera (ad esempio) **lo stesso codice di hash per tutti gli anagrammi** di una data stringa!
 - Es. "temp01" e "temp10" collidono
 - Es. "stop", "tops", "pots", "spot" collidono
(e sono tutte parole di senso compiuto in inglese...)
- È meglio usare una funzione che tenga conto della posizione dei caratteri nella stringa
 - Ad esempio analoga alla rappresentazione posizionale usata nei sistemi numerici

Sono problemi **difficili da studiare analiticamente**,
si usano soprattutto verifiche sperimentali

Funzioni di codifica polinomiali

- È meglio usare una funzione che tenga conto della posizione dei caratteri nella stringa, ad esempio analoga alla rappresentazione posizionale usata nei sistemi numerici
- Si usa proprio un polinomio
 - Si sceglie una base, si usano ordinatamente i valori Unicode dei caratteri come coefficienti del polinomio e se ne calcola il valore (**ignorando situazioni di overflow**)
 - Usando (almeno) 65536 come base e facendo i calcoli con `java.math.BigInteger` non si avrebbero mai collisioni, ma, per stringhe di lunghezza maggiore di 2, il codominio non sarebbe più l'insieme dei valori di tipo `int`
 - Si osserva **sperimentalmente** che questo riduce drasticamente il numero di collisioni, *soprattutto se si usa una base che abbia pochi zeri nei suoi bit meno significativi*
 - Con un insieme di 50000 parole inglesi e usando 33, 37, 39 o 41 come base, si sono osservate meno di 7 collisioni!

Il metodo **hashCode** di **Object**

- Ci sono **molte** altre funzioni di codifica, progettate per specifici insiemi di chiavi
- La classe **Object** ha una funzione di codifica predefinita, **hashCode**, per oggetti di tipo qualsiasi
 - Dovendo funzionare per ogni oggetto, non può basarsi sulle proprietà specifiche dell'oggetto... quindi come fa?
 - Si basa su una delle proprietà che caratterizza ogni oggetto Java e che lo rende diverso da tutti gli altri oggetti:
l'indirizzo di memoria in cui l'oggetto è posizionato
 - Analogamente a **equals**, che, in **Object**, è definito così

```
public boolean equals(Object obj)
{   return (this == obj);
}
```

Il metodo **hashCode** di **Object**

```
public boolean equals(Object obj)
{ return (this == obj);
}
public int hashCode()
{ return ...; }
```

Dato che qualsiasi oggetto eredita **hashCode** da **Object**, il problema della codifica sembrerebbe risolto!

- In pratica, il metodo **hashCode** potrebbe semplicemente restituire l'indirizzo in memoria dell'oggetto (che ha 32 bit), "convertito" in **int**, ma questo in Java non si può fare!
- In Java, gli indirizzi si possono confrontare tra loro soltanto per uguaglianza (come fa **equals**) e non secondo una relazione d'ordine; inoltre, non se ne può conoscere il valore
 - Per questo, il metodo **hashCode** figura nella libreria con la qualifica **native**, cioè è realizzato in linguaggio macchina o in altro linguaggio di programmazione (forse in C ?)

Dato che qualsiasi oggetto eredita **hashCode** da **Object**,
il problema della codifica sembrerebbe risolto! **Invece no...**

CENNI

Il metodo **hashCode** di **Object**

- Un metodo **hashCode** che restituisca **un codice di hash basato soltanto sulla posizione dell'oggetto in memoria**
NON è adatto per tutte le situazioni
 - In particolare, non è adatto a classi come **String** o come le classi involucro dei tipi di dato primitivi (**Integer**, **Double**, ecc.)
 - Infatti, questo codice non funzionerebbe...

```
HashMap<String, Qualcosa> m = ...;
String s = "topolino";
Qualcosa q = ...;
m.put(s, q);
String t = "topo" + "lino";
// t != s ma t.equals(s) == true !!!!
if (m.get(t) == null)
    System.out.println("Ops...");
```

(ma funziona con le stringhe della libreria standard)

Il metodo **hashCode** di **Object**

□ Esistono molte situazioni in cui si vuole che

- Due esemplari **distinti** di una classe (cioè due oggetti **non coincidenti**, perché posizionati in zone **diverse** della memoria) siano **considerati “uguali”** se contengono le medesime informazioni di stato, cioè, ad esempio, se hanno variabili di esemplare identiche (tutte o un sottoinsieme appositamente definito)
 - Essere **uguali/diversi** dipende dal **contenuto** degli oggetti
 - Essere **distinti/coincidenti** dipende dalla **posizione** degli oggetti in memoria
- Perché una tabella possa funzionare, bisogna che due oggetti **uguali, anche se distinti** in memoria, abbiano lo stesso codice di hash (e, conseguentemente, la stessa chiave ridotta)
 - Perché **la ricerca avviene solitamente con una chiave “uguale” a quella usata per l’inserimento, ma non necessariamente con “la stessa” chiave, cioè con il medesimo oggetto** (es. chiedo all’utente quale parola vuole cercare...)

Il metodo **hashCode** di **Object**

- Molte classi, come **String**, sovrascrivono **equals** in modo che usi un criterio di confronto basato sul valore delle variabili di esemplare degli oggetti e non sull'indirizzo degli oggetti stessi
- Solitamente questo implica la necessità di sovrascrivere anche **hashCode**, perché bisogna rispettare il “contratto” che sta alla base delle funzioni di codifica di hash
 - Proprietà di coerenza tra **equals** e **hashCode**: Due oggetti uguali secondo **equals** **devono** avere lo stesso **hashCode**; due oggetti con **hashCode** diversi **devono** essere diversi secondo **equals**.
 - Inoltre, due oggetti diversi secondo **equals** **dovrebbero** avere codici di hash diversi secondo **hashCode**
 - Se anche questo è sempre vero, la funzione di codifica è **ottima**, non ci sono mai collisioni
- Il metodo **hashCode** definito in **Object** è **coerente** con il metodo **equals** definito in **Object** ed è anche **ottimo** ma se si sovrascrive soltanto **equals** questo non è più vero

Funzione di compressione

- L’obiettivo di una **funzione di compressione** è quello di fare in modo che il codice di hash (un numero intero) associato a una chiave “rientri” in un intervallo di numeri interi di tipo $[0, N - 1]$
 - Nel farlo, dovrebbe **minimizzare la probabilità di collisione**
 - In generale, **la probabilità di collisione dipende**, oltre che dalla funzione di compressione, **anche dal valore di N**
 - Esempio: se N è minore della cardinalità dell’insieme di chiavi, la probabilità di collisione è necessariamente diversa da zero
 - Caso limite: se $N = 1$, la probabilità di collisione è 1 !! ☺
 - **Una funzione di compressione è ottima se non aumenta la probabilità di collisione rispetto a quanto fatto dalla funzione di codifica**
 - Se la funzione di codifica ha prodotto collisioni nei codici di hash, queste (ovviamente...) non possono essere eliminate dalla funzione di compressione, perché i codici di hash di due chiavi collidenti sono indistinguibili

Funzione di compressione

- Una funzione di compressione dovrebbe **minimizzare la probabilità di collisione**, cioè la probabilità che una qualsiasi coppia di codici di hash diversi dia luogo a una collisione nel dominio delle chiavi ridotte
 - Assegnato un valore di N , qual è il valore minimo teorico della probabilità di collisione?
 - Al contrario, il caso pessimo qual è? Come possiamo ottenere il numero **massimo** di collisioni? Basta usare **una “funzione di compressione” costante!**
 - Se la funzione di compressione restituisce **sempre la stessa chiave ridotta** indipendentemente dal codice di hash, la probabilità di collisione tra due codici di hash diversi è 1, cioè la collisione avviene sempre. Questa è la funzione di compressione **pessima!**

Funzione di compressione

- La **migliore** funzione di compressione (che **minimizza** la probabilità di collisione) deve agire “al contrario” di una funzione costante
 - Deve distribuire **uniformemente** le chiavi ridotte nel proprio codominio, indipendentemente dalla distribuzione dei codici di hash nel dominio (così “**sfrutta al meglio**” lo spazio disponibile nel codominio)
 - Ciascun codice di hash ha, in tal caso, probabilità $1/N$ di avere una determinata chiave ridotta
 - La probabilità che due qualsiasi codici di hash diversi collidano è, quindi, **$1/N$** , decrescente all'aumentare di N
 - Fare attenzione al calcolo delle probabilità: questa sembrerebbe $1/N^2$, ma sarebbe tale solo se si chiedesse una collisione in una **specifica** chiave ridotta, cioè: qual è la probabilità che due chiavi collidano in un determinato valore $x \in [0, N - 1]$? È $1/N^2$

Compressione per divisione

- Una semplice (e abbastanza efficace) funzione di compressione prevede di assegnare al codice di hash h la seguente chiave ridotta
 - $q = |h| \bmod N$, essendo **mod** l'operatore che calcola il resto della divisione intera (in Java, $q = |h| \% N$)
- Quando si ha collisione?
 - Ciascun codice di hash può essere scritto nella forma $h = pN + q$, dove q è la sua chiave ridotta (p e q interi)
 - Due codici di hash diversi collidono se hanno lo stesso valore di q , pur avendo valori di p diversi (essendo i codici diversi)
 - Esempio, con $N = 2$: tutti i codici di hash pari collidono nella chiave ridotta 0, tutti i codici di hash dispari collidono nella chiave ridotta 1
 - In generale
 - Tutti i multipli di N collidono fra loro (hanno $q = 0$)
 - Tutti i numeri del tipo $(q + \text{un multiplo di } N)$ collidono tra loro (e hanno chiave ridotta = q)

Compressione per divisione

- Una semplice (e abbastanza efficace) funzione di compressione prevede di assegnare al codice di hash h la seguente chiave ridotta
 - $q = |h| \bmod N$
- Si verifica **sperimentalmente** che questa funzione di compressione è "solitamente" migliore se si sceglie un valore di N che sia **un numero primo "abbastanza distante" da una potenza di due**
 - Dipende, comunque, dall'insieme dei codici di hash: scelto un valore di N , si riesce sempre a trovare un controesempio, cioè un insieme di codici che renda pessima questa funzione...

È molto difficile fare una trattazione analitica

Funzione di compressione

- Si utilizzano molte altre funzioni di compressione, che si rivelano sperimentalmente adatte a specifici insiemi di dati, ma i miglioramenti rispetto alla semplice funzione di "compressione per divisione" non sono generalizzabili
- Solo "per curiosità":
 - Metodo **MAD** (*multiply, add and divide*)
 - Metodo di moltiplicazione
(cfr. Cormen, Leiserson, Rivest, Stein)
 - Hashing “universale”
(cfr. Cormen, Leiserson, Rivest, Stein)
nota: il nome è fuorviante...
- **L'analisi teorica della probabilità di collisione è molto complessa**

Funzione di compressione

□ In teoria, N può essere scelto liberamente

- Non è vincolato ad essere uguale all'estensione dell'intervallo di numeri interi che contiene i codici di hash ottenuti applicando la funzione di codifica a un determinato insieme di chiavi (come dovrebbe, invece, essere in assenza di funzione di compressione)
- Occorre però tener presente che il valore di N ha, in generale, effetti sulla probabilità di collisione

□ In particolare, quindi, è possibile scegliere $N = O(n)$, essendo n la cardinalità dell'insieme di chiavi

- L'occupazione di memoria della tabella può così essere resa linearmente proporzionale alla dimensione della mappa che si realizza mediante la tabella stessa
- Questo vuol anche dire che, aumentando n , deve aumentare N ... vedremo che servirà un *rehashing*

Tabella: Gestione delle collisioni

- Come visto, la progettazione di un'opportuna funzione di hash **ottima** (cioè che non dia luogo a collisioni), consente di realizzare, mediante una tabella, una mappa avente **prestazioni $\Theta(1)$** per i metodi **get**, **put** e **remove**
 - Si può scegliere il valore di N desiderato per la realizzazione della funzione di compressione, in modo che sia, ad esempio, linearmente proporzionale alla dimensione n della mappa
- **Purtroppo, non esistono funzioni di hash che siano ottime per qualunque insieme di chiavi**
 - In generale, dobbiamo porci il **problema di risolvere le collisioni**, sperando di ottenere comunque buone prestazioni

Tabella: Gestione delle collisioni

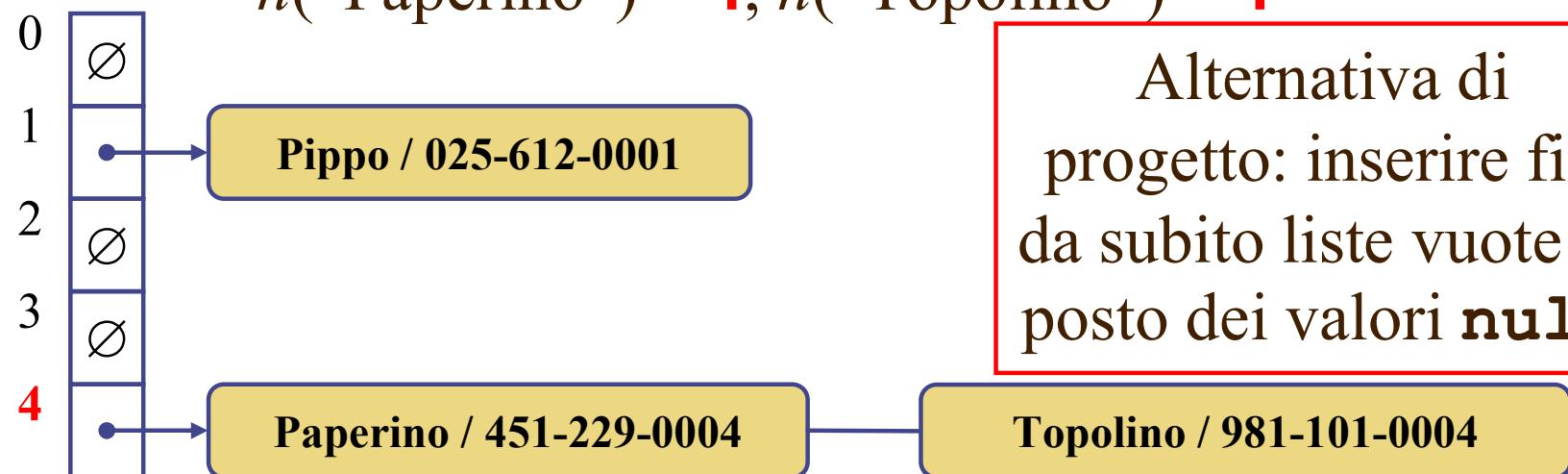
- Data la funzione di hash h e l'array A
 - In assenza di collisioni, abbiamo visto che la coppia (k, v) viene memorizzata con l'istruzione $A[h(k)] = v$, dopo aver inizializzato l'array con $A[m] = \text{null } \forall m$
 - In presenza di collisioni, la stessa cella $A[h]$ dovrebbe contenere tutte le coppie aventi chiavi (**anche diverse**) cui corrisponde la stessa chiave ridotta h : **usiamo un contenitore di coppie!**
 - Invece di un array di valori, usiamo un array di liste (dette *bucket*), contenenti coppie
 - Questa gestione delle collisioni si chiama **separate chaining** (“concatenazione separata”)

I metodi
keys e
values
sono
 $\Theta(n + N)$

Collisioni: Separate Chaining

□ Esempio: coppie nome/telefono

- Ipotesi: $h(\text{"Pippo"}) = 1$,
 $h(\text{"Paperino"}) = 4$, $h(\text{"Topolino"}) = 4$



□ Attenzione: non è sufficiente memorizzare **nei bucket** i soli valori, **servono le coppie!!**

- In un bucket ci sono, in generale, **coppie contenenti chiavi diverse** (aventi la stessa chiave ridotta): inserendo **solo i valori, non potrei più fare get e remove !**

Collisioni: *Separate Chaining*

- Tutte le operazioni (**get**, **put** e **remove**) sulla tabella sono realizzate seguendo queste fasi
 - **Calcolo della chiave ridotta** corrispondente alla chiave fornita come parametro, mediante la funzione di hash utilizzata dalla tabella (codifica + compressione)
 - **Accesso al bucket** corrispondente alla chiave ridotta
 - **Ricerca/inserimento/rimozione nel bucket**
 - Ricordiamo che in una mappa anche l'inserimento richiede una ricerca, così come la richiede la rimozione
- Le prime due fasi sono $\Theta(1)$
- Le prestazioni dei metodi sono, **nel caso pessimo**, $O(n)$, perché nel caso pessimo tutte le chiavi hanno la stessa chiave ridotta e la tabella usa un'unica lista (non ordinata)!
 - **Ma nel caso medio?**

Collisioni: *Separate Chaining*

- **Nel caso medio**, le prestazioni dei metodi della mappa dipendono linearmente dalla dimensione media dei bucket
- **Se la funzione di compressione è “buona”**, le chiavi ridotte si distribuiscono uniformemente nell’intervallo $[0, N - 1]$, quindi la dimensione media delle liste è n/N , che abbiamo chiamato *fattore di riempimento*, λ
 - Quindi, **in tale ipotesi**, le prestazioni medie dei tre metodi sono $O(\lambda) = O(n/N)$
 - **Se progettiamo N in modo che sia $N = O(n)$, allora i metodi sono mediamente $O(1)$!!**

	get	put	remove	keys / values
Mappa in lista (non ordinata)	$O(n)$	$O(n)$	$O(n)$	$\Theta(n)$
Mappa ordinata in array ordinato	$O(\log n)$	$O(n)$	$O(n)$	$\Theta(n)$
Mappa ordinata in AVL	$O(\log n)$	$O(\log n)$	$O(\log n)$	$\Theta(n)$
Mappa in una tabella hash: caso medio e caso pessimo	$O(\lambda)$ $O(n)$	$O(\lambda)$ $O(n)$	$O(\lambda)$ $O(n)$	$\Theta(n+N)$

Lezione 32

Tabella con collisioni: REHASHING

- Si vede sperimentalmente che le prestazioni di una tabella tendono a peggiorare quando λ si avvicina al valore 1
 - La probabilità di collisione si avvicina conseguentemente a 1 e ogni operazione richiede la scansione di una lista, per quanto di piccole dimensioni (mediamente, λ)
- Soluzione: quando λ supera una soglia, si può decidere di effettuare il **rehashing**
 - Si crea una tabella di dimensioni maggiori, ad esempio il doppio: così si aumenta N a parità di n e, quindi, si riduce λ
 - Si trasferiscono le coppie dalla vecchia tabella alla nuova tabella, calcolando le **nuove chiavi ridotte** (dato che, in generale, dipendono da N)
 - Si usa la nuova tabella al posto della vecchia
 - L'operazione è **$\Theta(N + n)$ ma (solitamente) vale l'analisi ammortizzata** perché λ dipende linearmente da n , quindi “non si fa spesso”

Mappe in `java.util`

- Oltre che da **TreeMap** (con albero rosso-nero), l’interfaccia **Map** è realizzata (anche) da **HashMap**
 - Usa una tabella hash con *separate chaining* il cui costruttore accetta
 - Capacità iniziale (cioè N), con default = 16
 - Soglia del fattore di riempimento per il *rehashing*, con default = 0.75
 - usando una soglia molto elevata si può, di fatto, inibire il rehashing
 - Funzione di codifica: metodo **hashCode** delle chiavi
 - Funzione di compressione: resto della divisione intera del codice di hash per N

Bucket Sort con hash?

- Con le funzioni di hash siamo riusciti a utilizzare un array di bucket con chiavi di qualsiasi tipo
- Quando queste sono ordinabili si potrebbe pensare di poter sempre usare Bucket Sort, con le chiavi ridotte anziché le chiavi “vere”
- Purtroppo, in generale, **le funzioni di hash non sono monotone** (in particolare, non lo è la funzione di compressione), quindi non trasferiscono sulle chiavi l’ordinamento delle coppie indotto dalle chiavi ridotte
 - Bucket Sort ordinerebbe le coppie in base all’ordinamento tra le chiavi ridotte che, in generale, non ha alcuna relazione con l’ordinamento presente tra le chiavi, né tale ordinamento può essere ricostruito a partire da quello delle chiavi ridotte, dal momento che la funzione di hash non è invertibile
 - **Non si può usare**

Mappa ordinata (SortedMap)

- In generale, le funzioni di hash non sono monotone, quindi non trasferiscono sulle chiavi ridotte la relazione d'ordine eventualmente presente tra le chiavi
 - La strategia di hashing non è adatta alla realizzazione di mappe ordinate
 - I metodi **keys** e **values** sono $O(n \log n)$
 - L'unica implementazione ragionevole di mappa ordinata usa un array ordinato oppure un **albero binario di ricerca!**
 - Se le chiavi sono ordinabili ma **non vengono usati i metodi specifici della mappa ordinata** (che necessitano dell'ordinamento, come **keys**), **si può usare comunque una tabella hash**, che ha **prestazioni medie migliori!**
 - Metodi **get**, **put** e **remove**
 - **$\Theta(\lambda)$ nel caso medio, $O(n)$ nel caso pessimo**
 - Attenzione: se le prestazioni di caso pessimo sono vincolanti (es. elaborazione in tempo reale), per le ricerche rimane migliore l'array ordinato (metodo **get** $O(\log n)$) o AVL

Collisioni: altre soluzioni

- La tecnica di *separate chaining* richiede, oltre all'array di *bucket*, **uno spazio totale $\Theta(n)$ per le liste**
- Per risparmiare spazio, a volte si usano tecniche diverse, che usano **solo un array di coppie (non di liste)** e si chiamano “a indirizzamento aperto” (*open addressing*)
 - Usando un solo array, deve essere $\lambda \leq 1$
- Tecnica principale: *linear probing*
 - **Inserimento** di (k, v) : // manca il caso «update»
$$h_{\text{probe}} = h(k) \quad // \text{tentativo iniziale}$$
$$\text{while } A[h_{\text{probe}}] \neq \text{null} \quad // \text{cerco cella "vuota"}$$
$$h_{\text{probe}} = (h_{\text{probe}} + 1) \bmod N \quad // \text{vado avanti!}$$

// aggiungere controllo "fatto giro completo?"...
$$A[h_{\text{probe}}] = (k, v) \quad // \text{ho trovato null : inserisco}$$

Collisioni: Open addressing

- Tecnica principale: *linear probing*

- **Ricerca** di k :

```
 $h_{\text{probe}} = h(k)$  // tentativo iniziale  
while  $A[h_{\text{probe}}] \neq \text{null}$  // cerco cella "vuota"  
    if  $\text{key}(A[h_{\text{probe}}]) == k$   
        return  $\text{value}(A[h_{\text{probe}}])$  // trovata!!  
 $h_{\text{probe}} = (h_{\text{probe}} + 1) \bmod N$  // vado avanti!  
    // aggiungere controllo "fatto giro completo?"  
return null // chiave non trovata
```

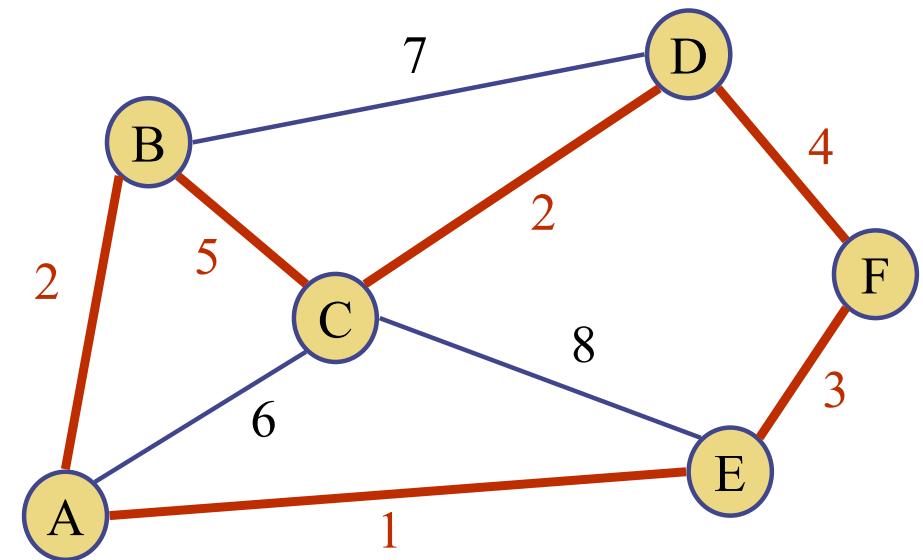
- **Rimozione** di k : se trovo la coppia da rimuovere, c'è il **problema di non lasciare buchi**... ci sono diverse soluzioni, tutte piuttosto complicate

Collisioni: Open addressing

□ Tecnica principale: *linear probing*

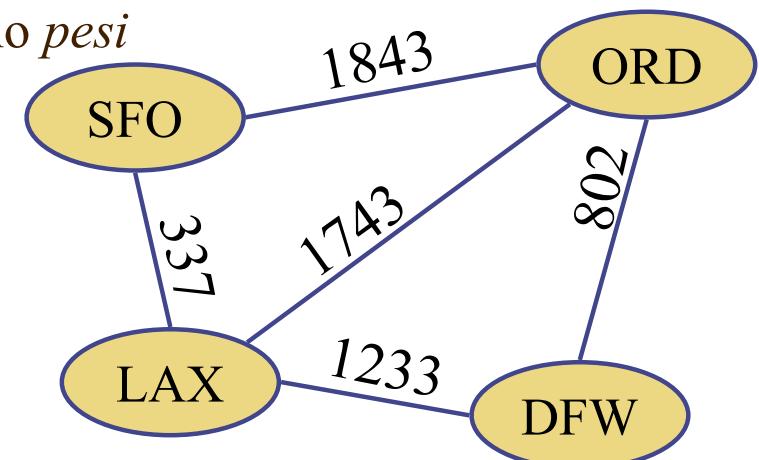
- Problema: fenomeno di *clustering*, tendono a formarsi gruppi di celle consecutive occupate e le prestazioni peggiorano, soprattutto se λ supera 0.5
 - Varie altre strategie di *open addressing* tentano di risolvere il problema del *clustering*
 - Quadratic probing
 - Double hashing
 - ...
- In generale: **le prestazioni non sono migliori del *separate chaining***, quindi se non ci sono seri problemi di spazio...

Grafi



I grafi

- Il **grafo** (*graph*) è un tipo di dato astratto che rappresenta **relazioni esistenti tra coppie di oggetti**
 - Da non confondere con il **grafico**... ☺
- Un grafo $G = (V, E)$ è costituito da
 - Un insieme V di oggetti, detti *vertici* o *nodi*
 - Solitamente in ogni vertice è memorizzato un dato
 - Un insieme E di “**connessioni**” tra **coppie di vertici**, dette *rami* (*edge*, che vorrebbe dire “spigolo”) o *archi*
 - Ad ogni ramo **può** essere associato un dato: in tal caso il grafo si dice “**pesato**” (*weighted*) e i dati si dicono *pesi*
- Il grafo è un tipo di dato astratto piuttosto complesso ma molto “potente”, utilizzato in molte applicazioni



Esempi di grafi e problemi tipici

- Rappresentare un insieme di persone e le relazioni di amicizia che legano alcune coppie di esse (il "problema di Facebook")
 - Chi sono gli amici di A ? Chi sono gli amici degli amici di B ?
 A e B sono “collegati” da una catena di amicizie? Con quale lunghezza minima?
- Rappresentare un insieme di aeroporti e i voli che connettono alcune coppie di essi
 - È possibile volare senza scalo da BLQ a LAX ?
Se non è possibile, qual è il minimo numero di scali necessari?
- Rappresentare calcolatori e le connessioni di rete che collegano direttamente alcune coppie di essi (es. la rete Internet)
 - Qual è il percorso minimo, attraverso la rete, per accedere dal mio calcolatore al sito Web **www.unipd.it**?
(si possono fare esperimenti con il comando traceroute in Linux o tracert in MS Windows)

Grafi e alberi

- Come vedremo, il grafo è (*in un certo senso*) un'estensione dell'albero
 - Anche l'albero rappresenta relazioni esistenti tra coppie di oggetti (un nodo e il suo genitore), ma ha più vincoli
 - Ad esempio, in un albero due fratelli o due "cugini" non possono essere legati da una relazione diretta
 - Non bisogna, però, fare confusione
 - Una delle differenze sostanziali è che l'albero ha un nodo "privilegiato", la radice
 - "Purtroppo" vedremo che esistono i "grafi ad albero"
- **Torneremo su questo punto**

Grafi orientati e non orientati

- Per rappresentare un ramo $e \in E$, usiamo la coppia di vertici connessi dal ramo stesso e scriviamo

$$e = (u, v) \in E \Rightarrow u \in V, v \in V$$

- Se l'ordine tra i vertici di un ramo è importante, **il ramo è orientato** (*directed*), altrimenti **il ramo è non orientato** (*undirected*)
 - Se tutti i rami di un grafo sono orientati, **il grafo è orientato** e si dice anche **digrafo** (*directed graph*)
 - Se tutti i rami di un grafo sono **non** orientati, **il grafo è non orientato**
 - Altrimenti, il grafo è “**misto**”
- I **grafi non orientati** rappresentano, in generale, **relazioni simmetriche** tra i vertici
- Un grafo non orientato o misto può sempre essere rappresentato da un digrafo avente gli stessi vertici e gli stessi rami orientati (se è misto), sostituendo ogni ramo non orientato del grafo originario con due rami, orientati nelle due direzioni opposte

Esempi di grafi orientati e non orientati

- Un grafo che rappresenta le relazioni di amicizia all'interno di un gruppo di persone è **non orientato**, perché la relazione di amicizia è simmetrica: se A è amico di B , allora B è amico di A
- In Java, un grafo che abbia classi e interfacce come vertici e relazioni **extends** e **implements** come rami è un grafo **orientato**, perché le relazioni di ereditarietà (**extends**) tra classi o tra interfacce e la relazione **implements** non sono simmetriche
- La carta stradale di una città può essere rappresentata da un grafo i cui vertici sono gli incroci e i cui rami sono le strade, ciascuna delle quali collega due incroci: è un grafo **misto**, perché le strade possono essere a doppio senso oppure a senso unico (come detto, può essere trasformato in un grafo orientato)

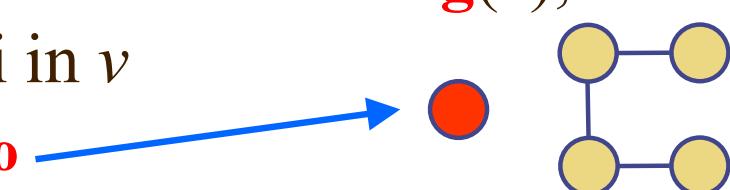
Rami e vertici

A volte per semplicità
si scrive $v \in G$

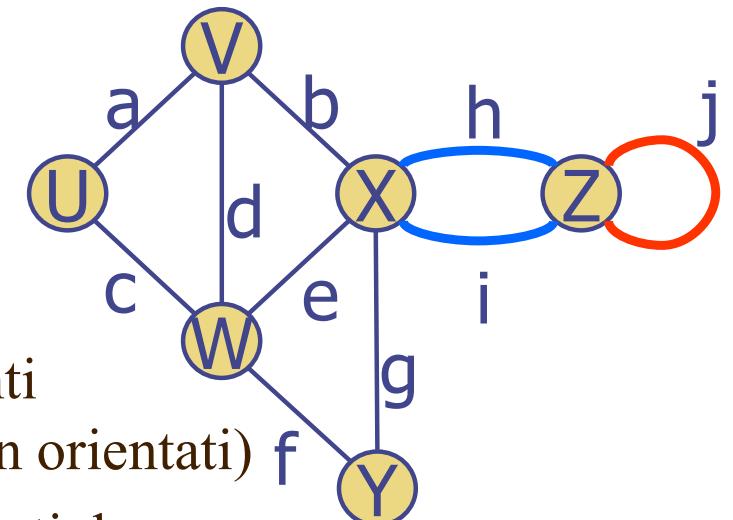
- Due vertici si dicono **adiacenti** se esiste un ramo che li collega
- I due vertici collegati da un ramo sono i suoi **estremi** o **terminali** (*endpoint*, e sono, naturalmente, adiacenti)
 - In un ramo orientato, gli estremi si distinguono in **origine** e **destinazione** del ramo
- Un ramo si dice **incidente** nei suoi due vertici e il **grado** (*degree*) di un vertice $v \in V \in G$, indicato con **deg**(v), è il numero dei rami che sono incidenti in v
 - Un vertice di grado zero si dice **isolato**
 - Tra i rami **orientati** incidenti in un vertice v , si distinguono:

Attenzione:
diverso dal
grado di un
nodo in un
albero

- i rami **uscenti** da v (*outgoing*), che hanno v come origine e contribuiscono al conteggio di **outdeg**(v)
- i rami **entranti** in v (*incoming*), che hanno v come destinazione e contribuiscono al conteggio di **indeg**(v)
- Naturalmente, in un grafo orientato, $\forall v$, **deg**(v) = **indeg**(v) + **outdeg**(v)



Rami “strani”

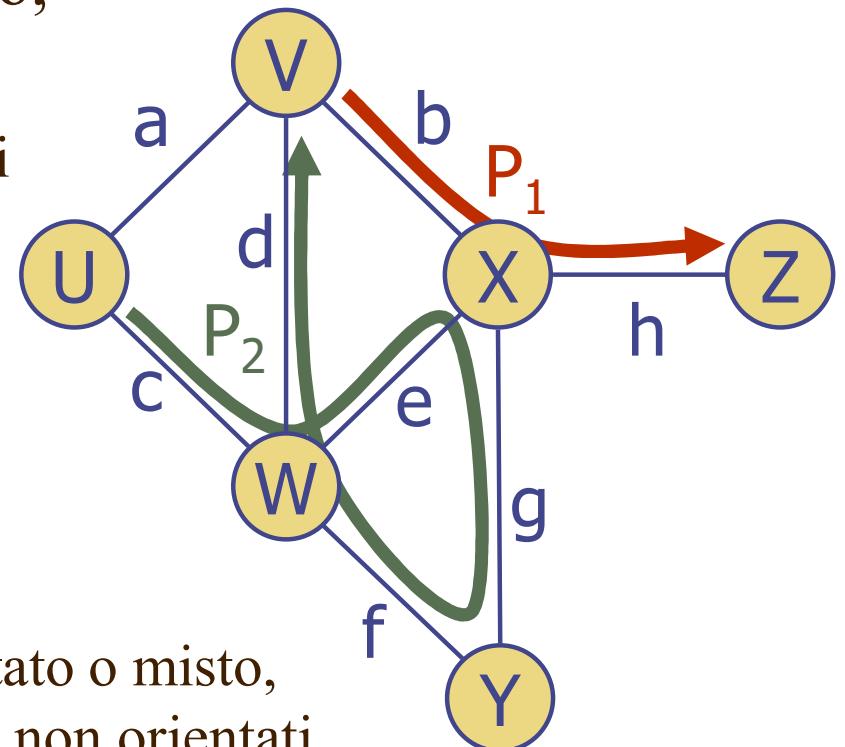


- In alcune applicazioni, può far comodo collegare una stessa coppia di vertici tramite più rami, aventi la stessa direzione se orientati (in figura, *h* e *i*, non orientati)
 - Esempio: due aeroporti possono essere collegati da più voli, ciascuno con un proprio costo, durata, compagnia, ecc.
 - Si parla di rami **paralleli** o **multipli**
 - Attenzione: due rami orientati aventi gli stessi estremi ma orientazione diversa **NON sono paralleli**
- In alcune applicazioni, può far comodo ammettere l'esistenza di **auto-anelli** (*self-loop*), cioè di rami aventi un solo vertice che ne costituisce entrambi gli estremi (in figura, *j*)
 - Esempio: in una mappa stradale, una rotonda consente (anche) di tornare al punto di partenza...
- Se un grafo non ha rami paralleli né auto-anelli, si dice **semplice**
 - In questo corso tratteremo esclusivamente **grafi semplici**

Percorsi in un grafo

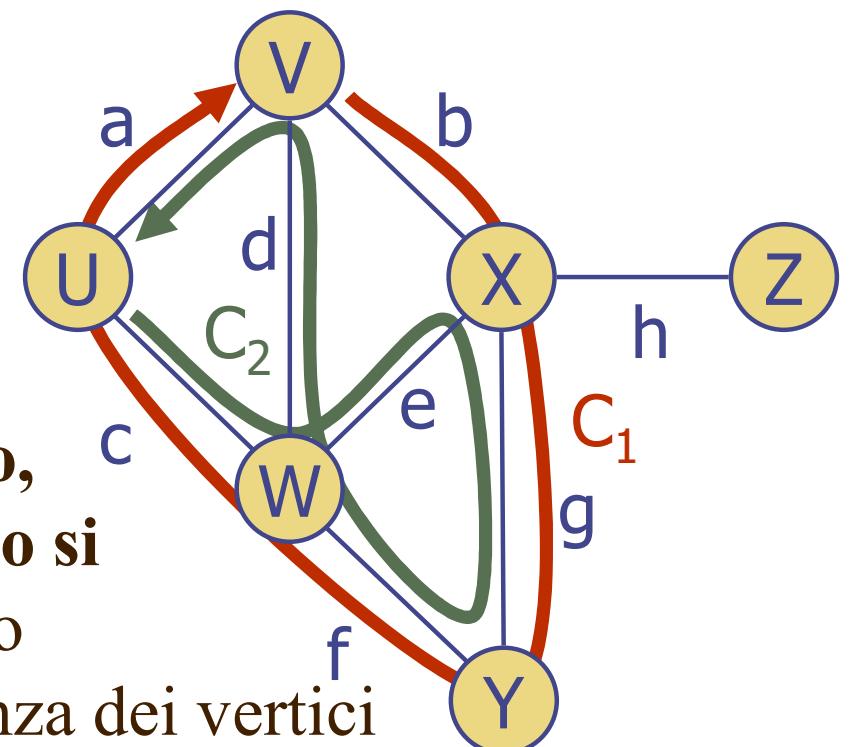
Un singolo vertice
è un percorso
(privo di rami)

- In un grafo, un **percorso** o cammino (*path*) è una sequenza di vertici e rami alternati tra loro in modo che
 - La sequenza inizi con un vertice e termini con un vertice
 - Ciascun ramo sia incidente nel vertice che lo precede e nel vertice che lo segue nella sequenza
 - **Nessun ramo compaia in due posizioni consecutive**
(diversamente dal percorso in un albero, non posso fare "avanti e indietro"...)
- Esempio: $P_1=(V,b,X,h,Z)$ è un percorso,
così come $P_2=(U,c,W,e,X,g,Y,f,W,d,V)$
- Un percorso è **semplice** se tutti i suoi vertici sono distinti (cioè se, percorrendolo, non si passa mai per un punto in cui si è già passati): P_1 è semplice, P_2 non lo è, perché il vertice **W** compare due volte
- Un percorso è **orientato** se tutti i suoi rami sono orientati e vengono attraversati secondo la loro direzione; in un grafo orientato o misto, quindi, esistono percorsi orientati e percorsi non orientati



Cicli in un grafo

- Un **ciclo** (*cycle* o *loop*) è un percorso (avente almeno due rami) i cui vertici iniziale e finale coincidono
- Se i due vertici, iniziale e finale, del percorso sono gli unici a coincidere, il ciclo si dice **semplice**
 - $C_1 = (V, b, X, g, Y, f, W, c, U, a, V)$ è un ciclo semplice
 - $C_2 = (U, c, W, e, X, g, Y, f, W, d, V, a, U)$ è un ciclo ma non è semplice
- Un ciclo è **orientato** se è un percorso orientato
- In un grafo semplice e non orientato, per descrivere un percorso o un ciclo si possono omettere i rami, perché sono univocamente individuati dalla sequenza dei vertici



Il tipo di dato astratto Graph

Il tipo di dato astratto Graph

- Come tipo di dato astratto, un grafo è costituito da vertici e rami, ognuno dei quali rappresenta una *posizione* nel grafo (e può contenere un dato)
- Possiamo, quindi, definire le interfacce generiche **Vertex<T>** e **Edge<T>**, ad esempio estendendo **Position<T>**
- Solitamente l'interfaccia **Graph<V, E>** **non** realizza **Container**, perché non è ben definita la sua “dimensione” (il numero di vertici? di rami?): in pratica ha due dimensioni
 - È un’interfaccia doppiamente generica perché, in generale, il tipo dei dati contenuti nei vertici, **V**, è diverso dal tipo dei dati contenuti nei rami, **E**
- Tutti i metodi che ricevono vertici e/o rami come parametri, possono lanciare **InvalidPositionException** (stesso problema di "appartenenza" visto nelle liste concatenate e negli alberi radicati) o, ancora meglio, **InvalidVertexException** e **InvalidEdgeException**, rispettivamente

```

public interface Edge<T> extends Position<T> { }
public interface Vertex<T> extends Position<T> { }
public interface Graph<V,E> // non orientato
{
    // ispezione
    int numEdges();
    int numVertices();
    Iterable<Edge<E>> edges();
    Iterable<Vertex<V>> vertices();
    Iterable<Edge<E>> incidentEdges(Vertex<V> v);
    Vertex<V> opposite(Vertex<V> v, Edge<E> e);
    Vertex<V>[] endVertices(Edge<E> e)
    boolean areAdjacent(Vertex<V> u, Vertex<V> v)
    // modifica
    V replace(Vertex<V> v, V o); // il dato
    E replace(Edge<E> e, E o); // il dato
    Vertex<V> insertVertex(V o); // inserito isolato
    V removeVertex(Vertex<V> v); // di solito rimuove
        // anche i rami incidenti, altrimenti
        // lancia eccezione se ha rami incidenti
    Edge<E> insertEdge(Vertex<V> u, Vertex<V> v, E o);
    E removeEdge(Edge<E> e);
}

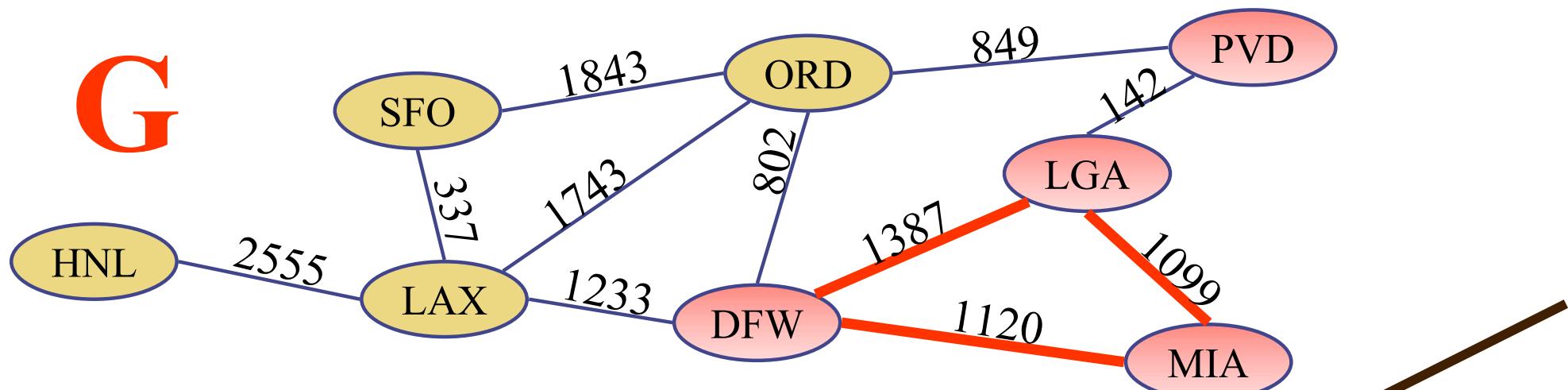
```

**Si può realizzare in
diversi modi: Vedremo**

Lezione 33

Sottografi

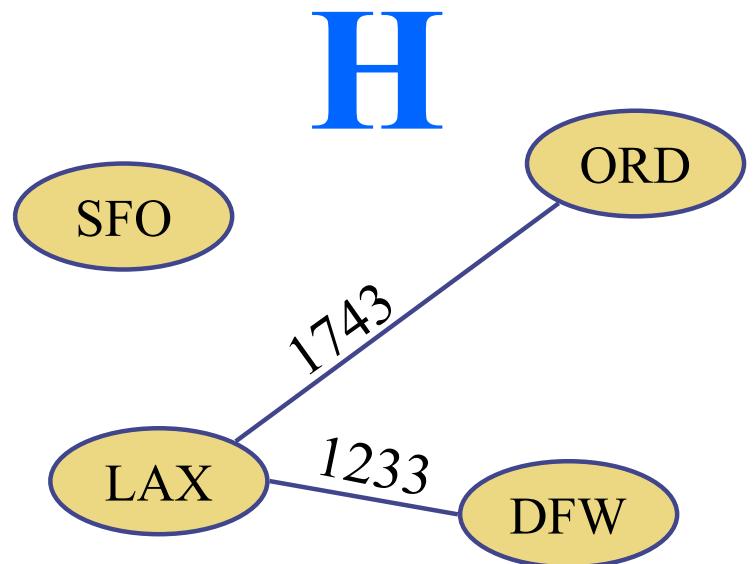
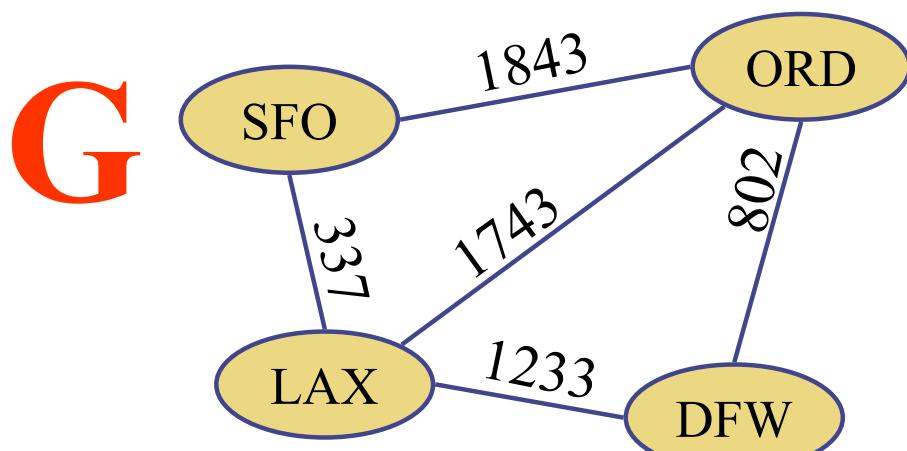
- Dato un grafo $G = \{V_G, E_G\}$, il grafo $H = \{V_H, E_H\}$ è un **sottografo** di G se $V_H \subseteq V_G$ e $E_H \subseteq E_G$



- Naturalmente, tutti i vertici che sono terminali di un ramo appartenente a E_H devono appartenere a V_H
 - In un grafo non possono esistere rami a cui manca uno dei vertici terminali!

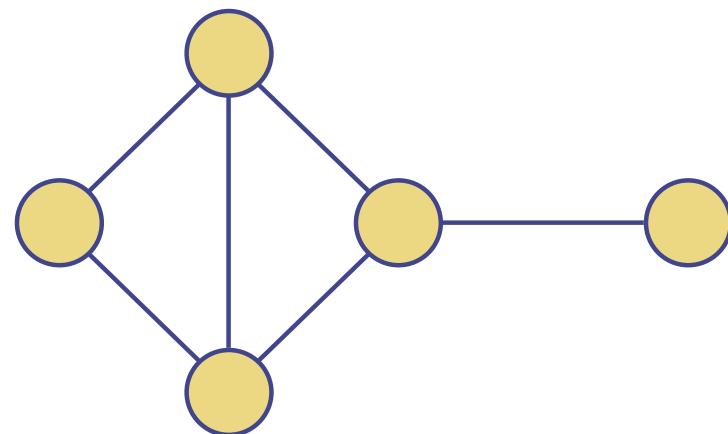
Spanning subgraph

- Se H è un sottografo di G contenente gli stessi vertici (ed eventualmente qualche ramo in meno), si dice che H è uno *spanning subgraph* di G (sottografo “ricoprente” o sottografo di copertura)



Grafo connesso

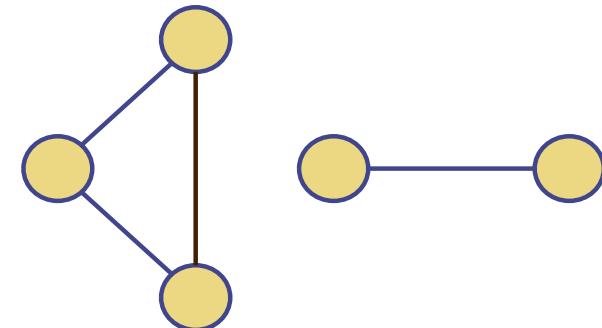
- Un grafo G è **connesso** se esiste un percorso tra due qualsiasi vertici distinti
 - Attenzione: anche se il grafo è orientato, è sufficiente un percorso, non necessariamente un percorso orientato



Grafo non connesso

- Se G non è connesso, i suoi **sottografi connessi di dimensioni massime** si dicono **componenti connessi** di G (a volte chiamati "isole")

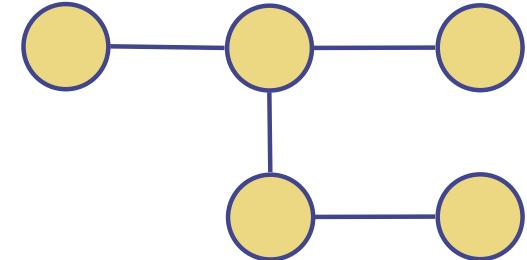
Questo è un
grafo con due
componenti
connessi



- Definiamo meglio i **sottografi connessi di dimensioni massime** di un grafo G
 - Sono i **sottografi connessi di G che NON sono sottografi di altri sottografi connessi di G** ma solo di se stessi
 - Come caso "degenere", per omogeneità si considera che un grafo connesso abbia un unico componente connesso, che coincide con il grafo stesso

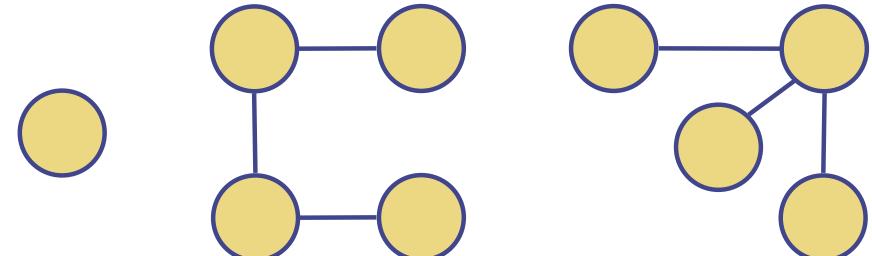
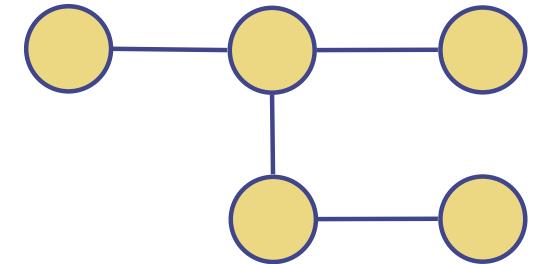
Grafo connesso e grafo ad albero

- Un grafo **connesso privo di cicli** è un **albero** (*tree*)
 - È una struttura diversa dagli alberi visti finora, perché in questo caso **non è definita una radice**
 - In caso di ambiguità, in quel caso parliamo di *albero con radice* o *albero radicato* (**rooted tree**), mentre in questo caso parliamo di *albero libero* o *grafo ad albero* (**free tree** o **tree graph**)
 - Proprietà: Un albero libero diventa un albero radicato (non ordinato) scegliendo un suo vertice **qualsiasi** e facendolo diventare radice
 - Ovviamente si ottiene UN DIVERSO ALBERO radicato per ogni nodo che viene scelto come radice, a partire da un grafo ad albero
 - In questo corso usiamo il termine “albero” per indicare gli alberi con radice e il termine “albero libero” o “grafo ad albero” per indicare, appunto, i grafî ad albero [nei casi in cui possano esserci dubbi]



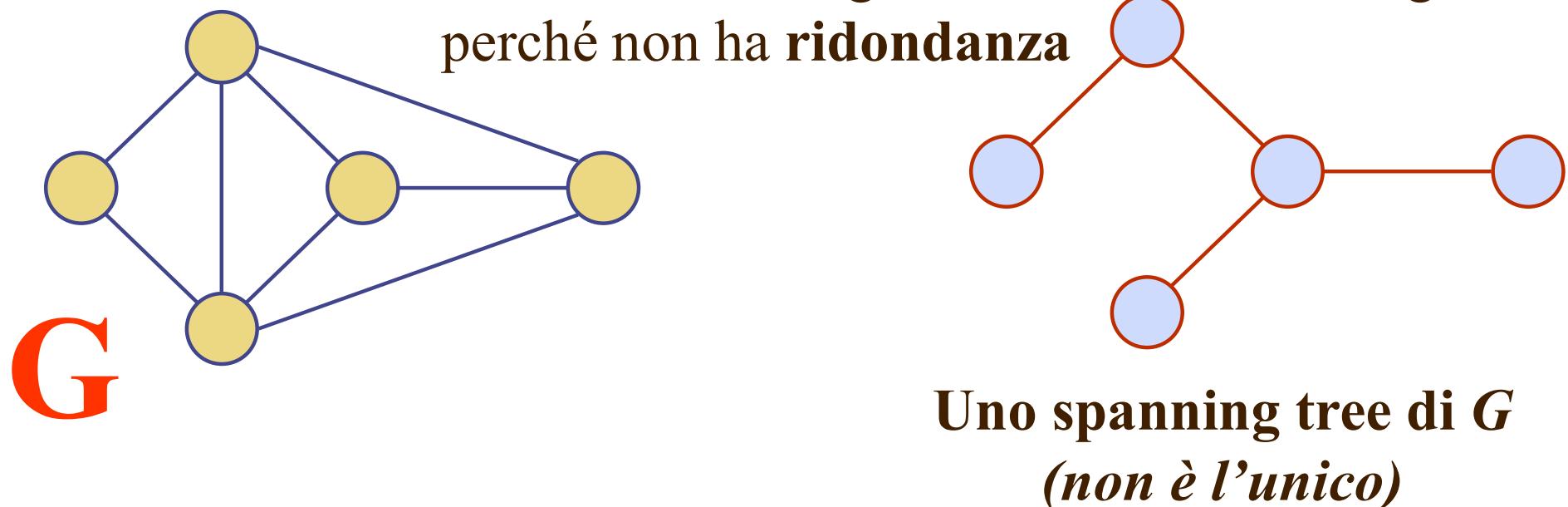
Alberi e foreste

- I grafi ad albero hanno una interessante proprietà (che si può dimostrare)
 - Un albero contiene il **minimo** numero di rami che rende connesso un grafo avente quell'insieme di vertici
 - Quindi, eliminando **un ramo qualsiasi** da un albero (che è un grafo connesso), si ottiene un grafo non connesso
- Un grafo **privo di cicli** (indipendentemente dal fatto che sia connesso oppure no) è una **foresta** (*forest*) e (ovviamente) i suoi componenti connessi sono alberi
 - Un albero è, evidentemente, anche una foresta... e una foresta connessa è un albero



Spanning tree

- Se uno *spanning subgraph* di un grafo G è un albero, lo si chiama ***spanning tree*** di G
- Dato un grafo, il calcolo di un suo spanning tree è un problema interessante: individua **un** (sotto)insieme **minimo** di rami che rendono connesso il grafo (in generale, **non è unico**)
 - Utile se, ad esempio, costruire rami ha un costo... C'è un rovescio della medaglia: è una struttura “fragile”...



Alcune proprietà

- Se il grafo G ha m rami (cioè $|E \in G| = m$), allora

$$\sum_{v \in G} \deg(v) = 2m$$

- Infatti, ogni ramo contribuisce per un'unità al grado di ciascuno dei suoi due estremi, quindi contribuisce con due unità alla sommatoria dei gradi di tutti i vertici
- Se il grafo **orientato** G ha m rami, allora

$$\sum_{v \in G} \text{indeg}(v) = \sum_{v \in G} \text{outdeg}(v) = m$$

- Infatti, ogni ramo orientato contribuisce per un'unità al grado di uscita della sua origine e per un'unità al grado di ingresso della sua destinazione

- Sia G un grafo **semplice** con n vertici e m rami
 - **Se G è non orientato, allora $m \leq n(n - 1)/2$**
 - **Se G è orientato, allora $m \leq n(n - 1)$**
- In un grafo **semplice non orientato**, non essendoci rami paralleli né auto-anelli, **ciascuno** degli n vertici può al massimo essere adiacente **a tutti gli altri** (che sono $n - 1$), quindi il suo grado massimo è $n - 1$ e, conseguentemente, la somma dei gradi di tutti i vertici del grafo (che sappiamo essere $2m$) è al massimo $n(n - 1)$. Quindi:
 - $2m \leq n(n - 1) \Rightarrow m \leq n(n - 1)/2$
 - Quando $m = n(n - 1)/2$, il grafo si dice **completo**
- Se il grafo **semplice** è **orientato**, lo stesso ragionamento è valido per il grado di ingresso (e di uscita), che, in ciascuno degli n vertici, può essere al massimo $n - 1$. Sappiamo che la somma di tutti i gradi di ingresso è m , quindi:
 - **$m \leq n(n - 1)$**
- Quindi, in ogni caso, in un grafo semplice **$m \in O(n^2)$**
 - Se il grafo non è semplice, non ci sono limiti superiori per m

Altre proprietà

- Sia G un grafo **semplice, non orientato**, con n vertici e m rami
- **Si può dimostrare** che
 - Se G è连通的, **allora** $m \geq n - 1$
 - Se G è una foresta (cioè è privo di cicli), **allora** $m \leq n - 1$
 - Se G è un albero (cioè è连通的 ed è privo di cicli), **allora**, dovendo valere entrambe le proprietà precedenti, $m = n - 1$
- **Attenzione:**
NON sono delle condizioni "se e solo se"...
(individuare dei contro-esempi), quindi:
 - Se $m < n - 1$, il grafo non è连通的; altrimenti... non si sa
 - Se $m > n - 1$, il grafo non è privo di cicli; altrimenti... non si sa
 - Se $m \neq n - 1$, il grafo non è un albero; altrimenti... non si sa

Attraversamento di Grafi

Attraversamento di grafi

- Un **attraversamento** (*traversal*) di un grafo è una procedura sistematica per **visitare** tutti i suoi n vertici e tutti i suoi m rami, una e una sola volta
 - Ci occupiamo soltanto di grafi **non orientati**
- L'azione di “visita” è variabile (con due azioni diverse, per vertici e per rami) e, come per gli alberi con radice, **l'attraversamento di un grafo G** è una delle operazioni fondamentali che si compiono su questa struttura e **costituisce la base di molti utili algoritmi**, ad esempio:
 - Trovare, se esiste, un percorso tra una coppia di vertici di G
 - Trovare un ciclo in G , oppure segnalare che G è una foresta
 - Verificare se G è connesso
 - Se G è connesso, calcolare un suo spanning tree
 - Altrimenti, identificare i suoi componenti connessi e uno spanning tree per ciascun componente (cioè una *spanning forest* per G)

Attraversamento di grafi

- **Osservazione:** l'interfaccia **Graph** contiene due metodi, **vertex()** e **edges()**, che restituiscono, rispettivamente, la lista dei vertici e la lista dei rami
 - E, come vedremo, normalmente le implementazioni di tale interfaccia hanno, come variabili di esemplare, proprio **una lista di vertici e una lista di rami**
 - Mentre in un albero radicato, come sappiamo, ciò non è vero: **in un albero realizzato come struttura concatenata non esiste una lista dei nodi**
- Quindi, un semplicissimo attraversamento si può effettuare invocando l'azione di visita per ogni elemento delle liste restituite dai metodi **vertex()** e **edges()**
- Questo attraversamento, però, non gode di nessuna delle proprietà topologiche peculiari e utili che vedremo essere, invece, caratteristiche dei "veri e propri" algoritmi di attraversamento di un grafo
 - È utile per fare operazioni "integrali", che non riguardino la struttura topologica del grafo: sommare i pesi contenuti nei rami, cercare uno specifico valore in un vertice, ecc.

```
SimpleTraverse(G)
  for each u ∈ G.vertices()
    visit(u)
  for each e ∈ G.edges()
    visit(e)
```

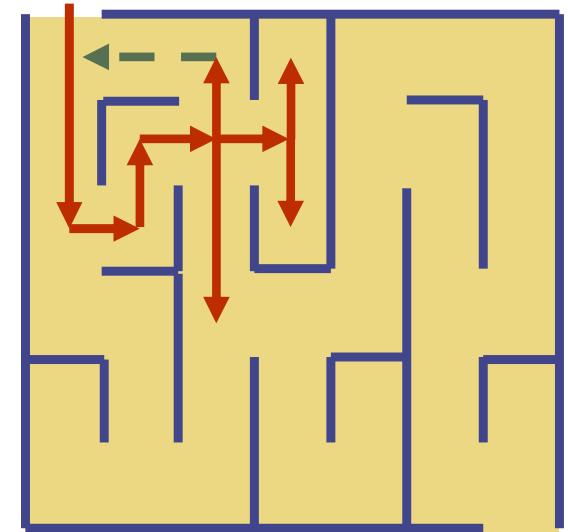
Attraversamento di grafi

- Gli algoritmi di *attraversamento topologico* si spostano da un vertice a un altro seguendo un ramo
- Esistono due fondamentali algoritmi di attraversamento topologico di un grafo, che agiscono con “filosofie” radicalmente diverse
 - Attraversamento “in profondità” (**depth-first traversal**, solitamente viene chiamato *depth-first search*, **DFS**)
 - Partendo da un vertice, cerca di andare “il più lontano possibile”, poi “torna sui suoi passi” (con un’azione di *backtracking*) e percorre altre strade
 - Attraversamento “in ampiezza” (**breadth-first traversal**, solitamente viene chiamato *breadth-first search*, **BFS**)
 - Partendo da un vertice, esplora prima tutti i vertici adiacenti, poi si espande “a macchia d’olio”, per “cerchi concentrici”

Attraversamento DFS

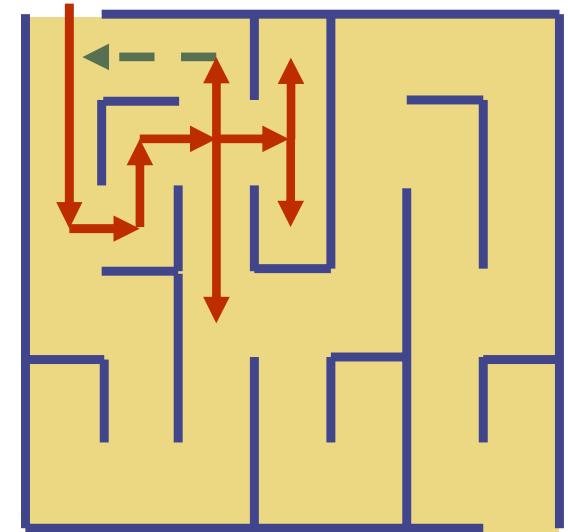
DFS – Depth-First Search

- L'idea di DFS corrisponde a un classico metodo di uscita da un labirinto, che usa un filo (cfr. il filo di Arianna!) per tenere traccia del percorso fatto (e poter tornare sui propri passi quando arriva in un vicolo cieco) e qualcosa per contrassegnare le zone (corridoi e incroci) già visitate
- Possiamo rappresentare un labirinto con un grafo
 - Un ramo per ogni corridoio e un vertice per ogni incrocio (non è altro che una "mappa stradale"...)
 - Aggiungiamo un vertice per l'ingresso e uno per l'uscita
 - Uscire dal labirinto equivale a risolvere un problema classico della teoria dei grafi: trovare un **percorso** tra due specifici vertici di un grafo
 - È il problema dei navigatori stradali! che hanno un problema in più, trovare il percorso **minimo** (secondo una certa metrica: tempo, lunghezza, spesa, ecc. cfr. **algoritmo di Dijkstra**, che vedremo)
 - Naturalmente dovremo modificare un po' l'algoritmo di Arianna, perché non vogliamo **solo** trovare **un** percorso, bensì visitare **tutto** il grafo!

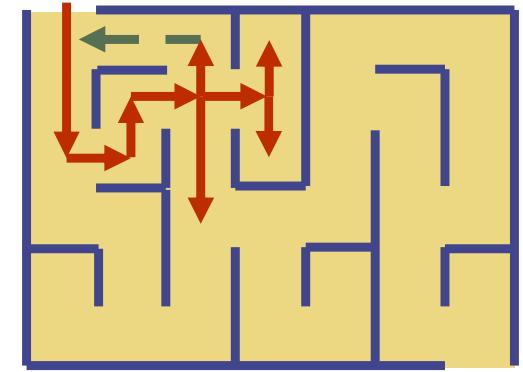


DFS – Depth-First Search

- Abbiamo bisogno di assegnare etichette “temporanee” a vertici e rami, per cui estendiamo le loro interfacce con i metodi **setLabel** e **getLabel**
(usiamo, cioè, **posizioni “decorabili”**)
- Decoriamo tutti i rami e tutti i vertici come “non visitati”
 $\forall v \in G, v.\text{setLabel}(\text{UNEXPLORED})$
 $\forall e \in G, e.\text{setLabel}(\text{UNEXPLORED})$
- Scegliamo a caso un vertice da cui partire, $s \in G$
 - Nel caso del labirinto, il vertice di partenza dovrà essere il punto di ingresso
 - Attacchiamo il filo al punto di partenza (saldamente! ☺)
- Usiamo un vertice u per rappresentare la posizione corrente nel grafo durante l'attraversamento
 - Iniziamo ponendo $u = s$
 - Passiamo da un incrocio (cioè un vertice) a un altro usando un corridoio (cioè un ramo)



DFS – Depth-First Search



1. Decoriamo u come visitato:

$u.\text{setLabel}(\text{VISITED})$

2. Cerchiamo di allontanarci da u : dobbiamo farlo percorrendo un ramo, quindi chiediamo al grafo la lista L dei rami incidenti in u

$L = G.\text{incidentEdges}(u)$

3. $\forall e \in L, \text{if } (e.\text{getLabel}() == \text{UNEXPLORED})$

- a) Il ramo e non è stato ancora visitato, quindi lo visitiamo

$e.\text{setLabel}(\text{VISITED})$

- b) Cosa c'è alla fine del corridoio? C'è $v = G.\text{opposite}(u, e)$

$\text{i. if } (v.\text{getLabel}() == \text{UNEXPLORED})$

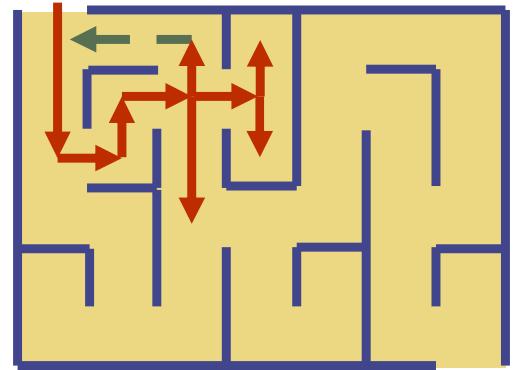
$u = v$ (ci spostiamo...) e **break**, torniamo al punto 1

ii. //else si torna al punto 3, scegliendo il ramo successivo

4. Se tutti i rami incidenti in u sono stati esplorati, siamo arrivati in un vicolo cieco...

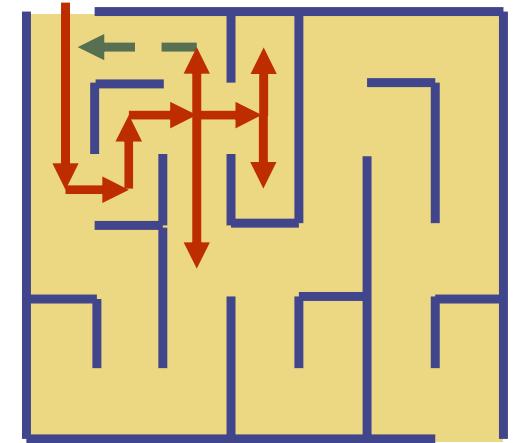
Tentativo di descrizione,
ancora incompleta...

DFS – Depth-First Search



- Se tutti i rami incidenti in u sono stati esplorati, siamo arrivati in un “vicolo cieco”
- Bisogna tornare indietro di un passo, riavvolgendo il filo, e provare un corridoio diverso
 - In pratica, il filo serve a realizzare uno stack dei vertici visitati che abbiano rami incidenti ancora da visitare
 - Ci consente di “tornare indietro”, realizzando un paradigma di programmazione che si chiama **backtracking** e che caratterizza molti algoritmi che procedono “per tentativi”, molto utilizzati nella "teoria dei giochi", un settore di "dati e algoritmi" che usa molto i grafi

DFS – Depth-First Search



- Per uscire dal labirinto, basta terminare l’algoritmo nel momento in cui u diventa uguale al vertice di uscita
- Se, invece, continuiamo a visitare vertici e rami, quando il backtracking ci riporta in s e abbiamo esplorato tutti i rami incidenti in s , l’algoritmo termina e (**si dimostra che**) ha visitato tutti i vertici e tutti i rami del componente连通的 C a cui appartiene s (quindi dell’intero grafo, se è connesso)
- È più semplice descrivere l’algoritmo in forma ricorsiva, così non serve lo stack, che viene realizzato implicitamente dalla ricorsione

DFS – Depth-First Search

- È più semplice descrivere l’algoritmo in forma ricorsiva
 - Per ogni vertice e per ogni ramo, l’azione `setLabel(VISITED)` avviene **una volta sola**, subito dopo aver verificato che `getLabel()` restituisca `UNEXPLORED`: ad essa possiamo quindi associare la vera e propria “azione di visita”
 - Cioè: ogni volta che invochiamo `setLabel(VISITED)` invochiamo anche `visit()` per il ramo o per il nodo, senza scriverlo qui esplicitamente, per brevità
- DFS(G, s)***

s.setLabel(VISITED) // visita s

for each *e* **in** *G.incidentEdges(s)*

if *e.getLabel() == UNEXPLORED*

e.setLabel(VISITED) // visita e

w \leftarrow *G.opposite(s,e)*

if *w.getLabel() == UNEXPLORED*

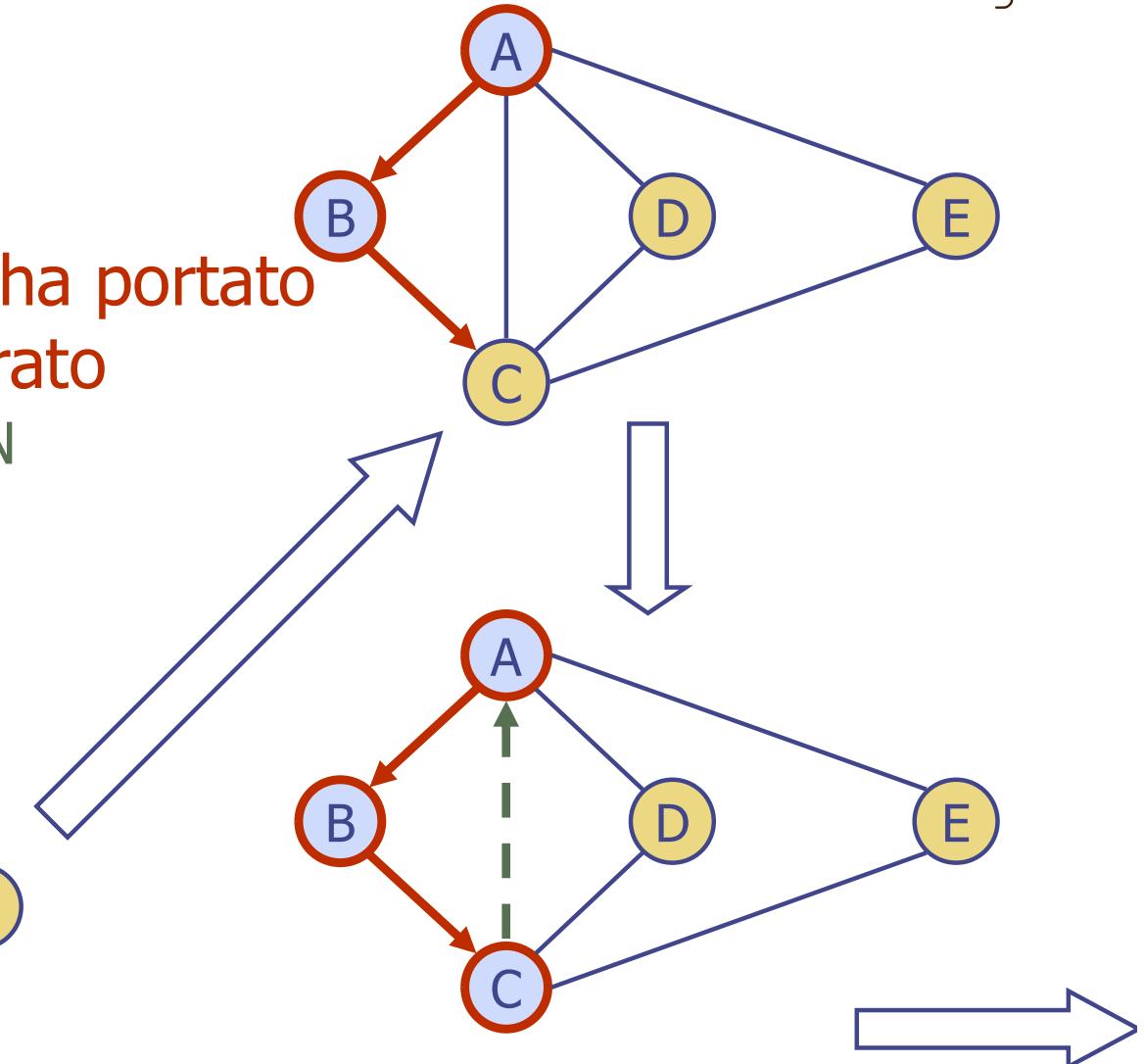
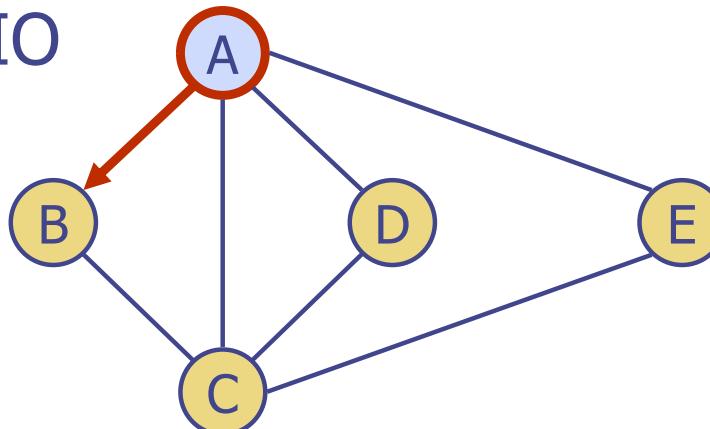
DFS(G, w) // RICORSIONE

Esempio di esecuzione di DFS

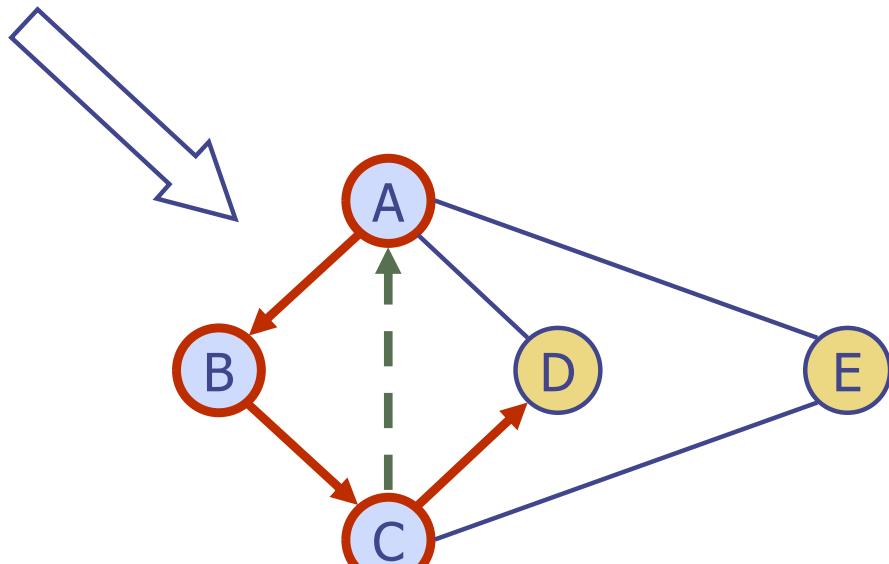
L'esecuzione
dipende dall'ordine
in cui i rami sono
elencati da
incidentEdges

- Vertice inesplorato
- Vertice visitato
- Ramo inesplorato
- Ramo esplorato che ha portato
in un vertice inesplorato
- Ramo esplorato che NON
ha portato verso un
vertice inesplorato

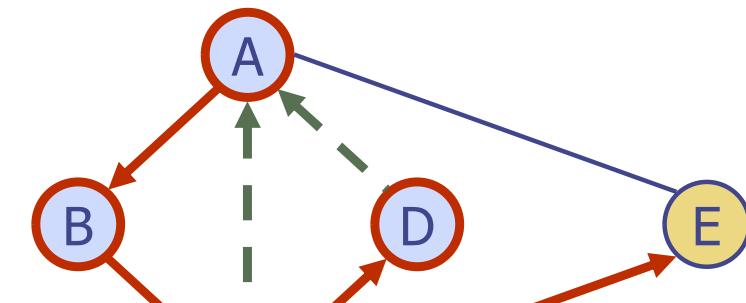
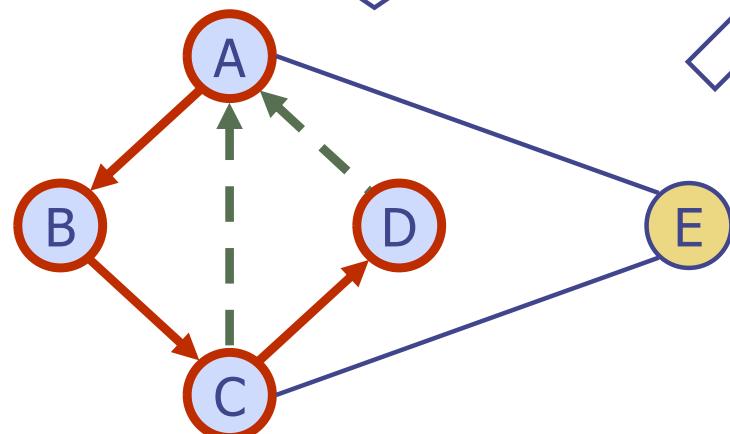
INIZIO



Esempio di esecuzione di DFS



Backtracking
da D a C



Backtracking da E a
C, poi da C a B, poi
da B ad A e FINE

Lezione 34

DFS

Tesi: L'algoritmo DFS effettua un attraversamento del componente连通的 C a cui appartiene il vertice di partenza s (quindi dell'intero grafo, se è connesso)

DFS(G, s)

```
s.setLabel(VISITED) // visita s
for each  $e \in G.incidentEdges(s)$ 
    if  $e.getLabel() == UNEXPLORED$ 
        e.setLabel(VISITED) // visita e
         $w \leftarrow G.opposite(s,e)$ 
        if  $w.getLabel() == UNEXPLORED$ 
            DFS( $G, w$ ) // RICORSIONE
```

□ Per assurdo: esiste $v \in C, v \neq s$, che non è stato visitato.

Dato che s e v appartengono allo stesso componente connesso, C , per definizione esiste un percorso P che va da s a v : sia w il vertice di P più vicino a s tra quelli che non sono stati visitati (al limite, $w = v$).

Dato che $w \neq s$ (perché s è stato visitato), esiste un vertice u che precede w in P (cioè che, lungo P , è più vicino a s di quanto lo sia w ; al limite, $u = s$) e u è stato visitato, perché w è, per definizione, il più vicino a s tra quelli che non sono stati visitati.

Sia e il ramo che ha u e w come estremi ($e \in P$)

Dato che u è stato visitato, l'algoritmo (per come è definito) ha certamente visitato anche il ramo e che incide su di esso, e, conseguentemente, è stato visitato l'altro estremo di e , cioè w .

Assurdo.

□ Analogamente, si dimostra che tutti i rami di C sono stati visitati.

DFS – Depth-First Search

- E se volessimo **attraversare completamente un grafo non connesso?**
- Se, dopo aver scelto un vertice iniziale (ad esempio a caso) e aver eseguito DFS partendo da quel vertice, si hanno ancora vertici inesplorati (cosa che si può facilmente verificare scandendo la lista dei vertici e ispezionando ciascuna etichetta), possiamo intanto rispondere a una domanda: **Il grafo è connesso?**
Risposta: No, altrimenti, come abbiamo (parzialmente) dimostrato, DFS l'avrebbe visitato interamente
- A questo punto, basta **scegliere un nuovo vertice iniziale tra quelli rimasti inesplorati** e ripetere DFS a partire da esso
 - Ripetendo la procedura finché esistono vertici inesplorati, si esegue (ricorsivamente) **un DFS per ogni componente connesso del grafo**
 - Alla fine, si ottiene effettivamente un attraversamento del grafo, anche se non è connesso (attraversamento topologico "per quanto possibile")

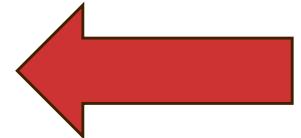
DFS – Depth-First Search

- Riassumendo, come si usa DFS per attraversare un intero grafo G , anche se non è connesso?

```
DFS(G) // inizializzazione e main
  for each  $u \in G.vertices()$ 
     $u.setLabel(UNEXPLORED)$ 
  for each  $e \in G.edges()$ 
     $e.setLabel(UNEXPLORED)$ 
  connectedComponents = 0
  for each  $v \in G.vertices()$ 
    if  $v.getLabel() == UNEXPLORED$ 
      connectedComponents++
    DFS(G, v)
```

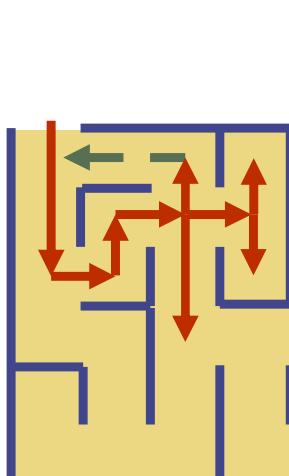
```
DFS(G, s) // ricorsivo, attraversa il
// componente连通的 a cui appartiene s
   $s.setLabel(VISITED) // visita s$ 
  for each  $e \in G.incidentEdges(s)$ 
    if  $e.getLabel() == UNEXPLORED$ 
       $e.setLabel(VISITED) // visita e$ 
       $w \leftarrow G.opposite(s,e)$ 
      if  $w.getLabel() == UNEXPLORED$ 
        DFS(G, w) // ricorsione
```

- Al termine, G è connesso se e solo se **connectedComponents == 1**
- Questa informazione, ad esempio, non si può ottenere facendo l'attraversamento "banale" visto inizialmente



DFS – Ricerca di spanning tree

- Come si può usare DFS per individuare uno spanning tree di un grafo (o di un suo componente) connesso?
 - Uno spanning tree ha gli stessi vertici del grafo (o componente connesso), quindi il problema coincide con l'**identificazione di un sottoinsieme di rami** (eventualmente improprio, se il grafo è un albero e, quindi, è lo spanning tree di se stesso)
 - **Bisogna migliorare un po' la “decorazione” dei rami, non basta contrassegnarli come “visitati” o “inesplorati”**
 - Quando esaminiamo un ramo inesplorato, se questo porta verso un vertice inesplorato lo contrassegniamo come ***DISCOVERY edge***, altrimenti come ***BACK edge*** (perché "torna indietro")
 - Solo nel primo caso "srotoliamo effettivamente un po' di filo"… nel secondo caso andiamo un po' avanti (fino al vertice, che scopriamo essere già stato visitato) poi torniamo immediatamente indietro
 - **Si dimostra che** al termine dell’attraversamento di un grafo (o di un suo componente) connesso, l’insieme dei soli rami etichettati come ***DISCOVERY*** definisce un suo spanning tree (in generale non unico)

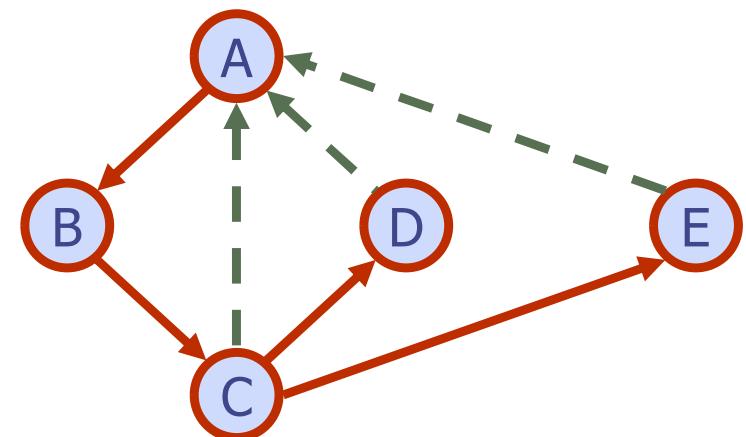
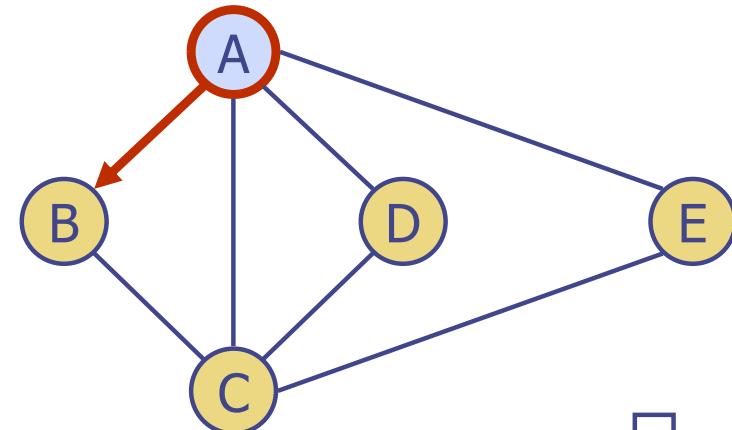


DFS individua uno spanning tree

- A Vertice inesplorato
- A Vertice visitato
- Ramo inesplorato
- Ramo DISCOVERY
- → Ramo BACK

Stesso esempio analizzato in precedenza, ora con etichette DISCOVERY e BACK

In generale, esistono più spanning tree di un grafo ciclico



DFS individua una spanning forest

- Riassumendo, come si usa DFS per individuare una spanning forest di G ?

```
DFS(G) // inizializzazione e main
for each  $u \in G.vertices()$ 
     $u.setLabel(UNEXPLORED)$ 
for each  $e \in G.edges()$ 
     $e.setLabel(UNEXPLORED)$ 
connectedComponents = 0
for each  $v \in G.vertices()$ 
    if  $v.getLabel() == UNEXPLORED$ 
        connectedComponents++
        DFS(G, v)
```

$DFS(G, s)$ // ricorsivo

```
s.setLabel(VISITED) // visita s
for each  $e \in G.incidentEdges(s)$ 
    if  $e.getLabel() == UNEXPLORED$ 
        e.setLabel(VISITED) // visita e
         $w \leftarrow G.opposite(s,e)$ 
        if  $w.getLabel() == UNEXPLORED$ 
            DFS(G, w) // ricorsione
```

modificato

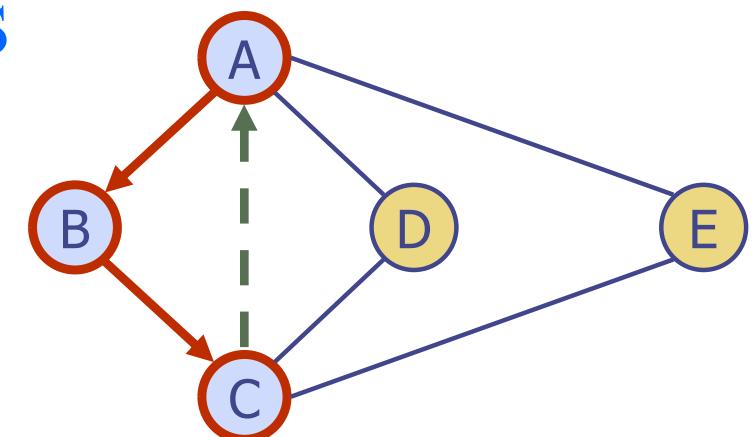
$DFS(G, s)$ // ricorsivo

```
s.setLabel(VISITED)
for each  $e \in G.incidentEdges(s)$ 
    if  $e.getLabel() == UNEXPLORED$ 
        // qui c'era e.setLabel(VISITED)
         $w \leftarrow G.opposite(s,e)$ 
        if  $w.getLabel() == UNEXPLORED$ 
            e.setLabel(DISCOVERY)
            DFS(G, w)
        else
            e.setLabel(BACK)
```

- I rami DISCOVERY individuano una spanning forest di G (che è uno spanning tree se G è connesso)

DFS: individuare la presenza di cicli

- Come si può usare DFS per **individuare un ciclo** in un grafo e, **conseguentemente**, decidere che il grafo **NON** è una foresta?
 - Si usa lo stesso algoritmo con etichette DISCOVERY/BACK per i rami
 - Cosa significa assegnare un'etichetta **BACK** a un ramo?
Significa che, percorrendo quel ramo, si arriva in un nodo già visitato: dato che l'algoritmo non percorre mai due volte uno stesso ramo, ciò può avvenire soltanto se il grafo ha un ciclo (si può dimostrare in modo più formale)
- Quindi, **si può dimostrare che un grafo è una foresta se e solo se un suo attraversamento DFS non ha rami di tipo BACK** e
(si dimostra che) tale risultato NON dipende dal vertice scelto per iniziare il DFS

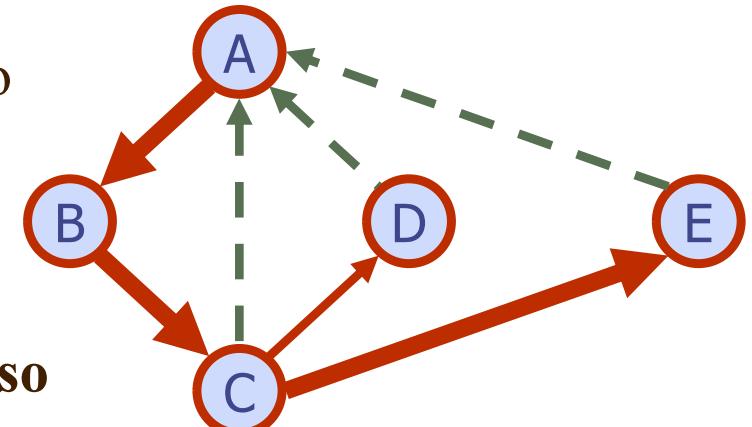


DFS: trovare un percorso tra due vertici

- Come si può usare DFS per trovare (se esiste) un percorso tra due vertici qualsiasi, v e z ?
 - È il problema dell'uscita da un labirinto...
 - **Si inizia DFS da uno dei due vertici**, diciamo v
 - Si esegue normalmente DFS, con un'unica modifica:
nel momento in cui si visita un vertice, assegnandogli quindi l'etichetta VISITED, **si verifica se si tratta di z** , nel qual caso l'algoritmo termina
 - **Rimane un problema da risolvere: terminato l'algoritmo con successo, sappiamo che esiste un percorso tra v e z , ma qual è?**
 - Se DFS termina **senza** aver trovato z , allora non esiste un percorso tra v e z , cioè v e z appartengono a componenti connessi diversi

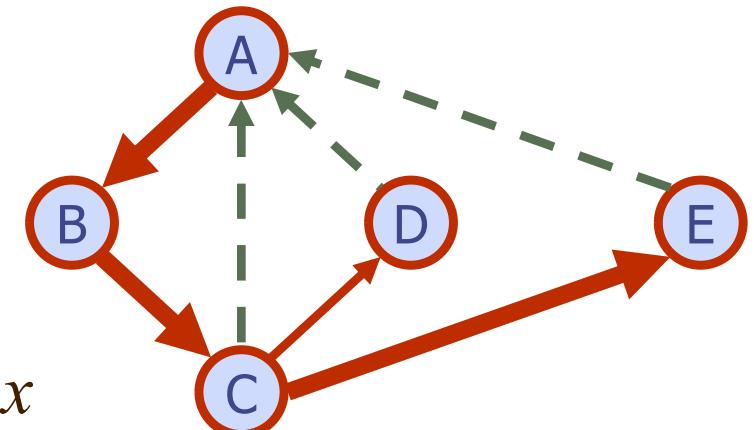
DFS: trovare un percorso tra due vertici

- Rimane un problema da risolvere: sappiamo che esiste un percorso tra v e z , ma qual è?
- Durante l'esecuzione di DFS, bisogna tenere traccia del percorso “netto” tra v e il vertice corrente, depurandolo dei tentativi che hanno portato a un vicolo cieco
 - Certamente i rami BACK non fanno parte del percorso
 - Ma anche alcuni rami DISCOVERY devono essere esclusi dal percorso...
 - Ad esempio, nella ricerca di un percorso tra A ed E, il ramo C-D, pur essendo di tipo DISCOVERY, non fa parte del percorso utile, che è A-B-C-E
 - Si osservi che A-B-C-E non è il percorso più breve, ma questo è un altro problema...



DFS: trovare un percorso tra due vertici

- Rimane un problema da risolvere: sappiamo che esiste un percorso tra v e z , ma qual è?
- Durante l'esecuzione di DFS, bisogna tenere traccia del percorso “netto” tra v e il vertice corrente, **depurandolo** dei tentativi che hanno portato a un vicolo cieco
 - Usiamo, ad esempio, uno stack per memorizzare il percorso utile e iniziamo impilando il vertice di partenza (lo stack conterrà, alternativamente, vertici e rami: infatti, memorizza un percorso)
 - Quando visitiamo un ramo, lo impiliamo solo se è DISCOVERY
 - Quando visitiamo un vertice, lo impiliamo
 - Quando **terminiamo** la visita di un vertice x (senza aver trovato z), lo togliamo dalla pila e, poi, facciamo un altro **pop**, che toglie il ramo DISCOVERY che ci aveva portato in x



DFS: trovare un percorso tra due vertici

Inizializzazione

```
for each  $u \in G.vertices()$ 
     $u.setLabel(UNEXPLORED)$ 
for each  $e \in G.edges()$ 
     $e.setLabel(UNEXPLORED)$ 
 $S = \text{new Stack}()$ 
pathDFS( $G, v, z, S$ )
//  $S$  contiene il percorso trovato;
// se  $S$  è vuoto significa che
// non c'è un percorso tra  $v$  e  $z$ 
```

In questo caso, le etichette DISCOVERY e BACK non servono, basterebbe VISITED per tutti i rami

```
pathDFS( $G, v, z, S$ )
 $v.setLabel(VISITED)$ 
 $S.push(v)$ 
if  $v == z$ 
    return true // caso base
for each  $e \in G.incidentEdges(v)$ 
    if  $e.getLabel() == UNEXPLORED$ 
         $w \leftarrow \text{opposite}(v, e)$ 
        if  $w.getLabel() == UNEXPLORED$ 
             $e.setLabel(DISCOVERY)$ 
             $S.push(e)$ 
            if pathDFS( $G, w, z, S$ )
                return true
             $S.pop(e)$ 
        else
             $e.setLabel(BACK)$ 
     $S.pop(v)$ 
return false
```

DFS – Prestazioni?

- Per discutere le prestazioni temporali di DFS bisogna conoscere i dettagli della implementazione di grafo con cui si opera
 - I tipi di dati **astratti** non hanno prestazioni...
 - **Ovviamente qualsiasi attraversamento è $\Omega(n + m)$, non si può immaginare di visitare tutti i vertici e tutti i rami senza... visitarli tutti!**
- Ma quali metodi usa DFS ?
 - I metodi di “decorazione” sono ragionevolmente $\Theta(1)$ [vedere più avanti]
 - Nell’inizializzazione usa
 - **vertices**, sarà $O(n)$? Se anche fosse $\Theta(1)$, restituendo il riferimento a una lista interna, il ciclo di inizializzazione sarebbe comunque $\Omega(n)$
 - **edges**, sarà $O(m)$? Se anche fosse $\Theta(1)$, restituendo il riferimento a una lista interna, il ciclo di inizializzazione sarebbe comunque $\Omega(m)$
 - Nella parte ricorsiva usa
 - **incidentEdges (v)** , sarà $O(\deg(v))$?
Se anche fosse $\Theta(1)$, la scansione della lista sarebbe $\Omega(\deg(v))$...
 - **opposite (e, v)** : sarà $\Theta(1)$?

DFS – Prestazioni?

Bisognerebbe contare anche le operazioni **getLabel** e **setLabel**, nella slide successiva (influente...)

- Che prestazioni ha DFS se tutti i metodi sono ottimi, cioè se tutti gli Ω sono anche Θ ?
- La fase di inizializzazione è $\Theta(n + m)$
- Nella parte ricorsiva, complessivamente
 - Il metodo **incidentEdges** viene invocato una e una sola volta per ogni vertice del grafo (subito dopo la “visita” del vertice)
 - Se **incidentEdges** (v) = $\Theta(\deg(v))$, allora il contributo di tutte le invocazioni di **incidentEdges** è $\Theta(m)$, perché la somma dei gradi di tutti i vertici di un grafo è $2m$
 - Il metodo **opposite** viene invocato una e una sola volta per ogni ramo del grafo (subito prima della “visita” del ramo), quindi il contributo di tutte le invocazioni di **opposite** è $\Theta(m)$
- Quindi, se tutti i metodi sono ottimi, **DFS** è $\Theta(n + m)$, cioè è ottimo (perché gli attraversamenti sono $\Omega(n + m)$)

```
s.setLabel(VISITED)
for each e ∈ G.incidentEdges(s)
```

```
if e.getLabel() == UNEXPLORED
    w ← G.opposite(s,e)
    if w.getLabel() == UNEXPLORED
        e.setLabel(DISCOVERY)
        DFS(G, w)
    else e.setLabel(BACK)
```

DFS – Prestazioni?

- Conteggio di **getLabel** / **setLabel**
 - Un **setLabel(UNEXPLORED)** per ogni vertice e per ogni ramo nell'inizializzazione: $\Theta(n + m)$
 - **setLabel(VISITED)** per ogni vertice (una e una sola volta): $\Theta(n)$
 - **setLabel(DISCOVERY o CROSS)** per ogni ramo (una e una sola volta): $\Theta(m)$
 - **getLabel** per vertici: uno per ciascuno nel ciclo che segue l'inizializzazione, $\Theta(n)$
 - **getLabel** per vertici: uno per ogni invocazione di **opposite**, quindi $\Theta(m)$
 - **getLabel** per rami: uno per ogni elemento di una lista restituita da **incidentEdges**, quindi $\Theta(m)$
- Totale: $\Theta(n + m)$

Implementazioni di grafi

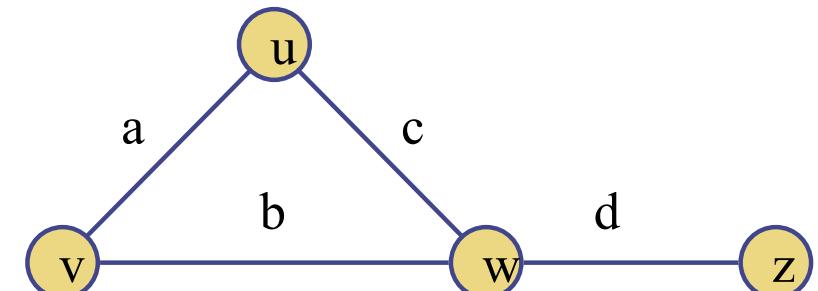
Implementazioni di Graph

- **Grafi semplici e non orientati**
- Esistono molte diverse realizzazioni dell'interfaccia **Graph**, ma, in pratica, sono tutte (piccole) variazioni sul tema di una delle tre seguenti
 - Edge list (lista dei rami)
 - Adjacency list (lista delle adiacenze)
 - Adjacency matrix (matrice delle adiacenze)
- Anticipiamo le **prestazioni comuni** per un grafo con n vertici e m rami, mettendo poi in evidenza soltanto le differenze
 - **numEdges** e **numVertices** sono $\Theta(1)$
 - **opposite** e **endVertices** sono $\Theta(1)$
 - i due metodi **replace** sono $\Theta(1)$
 - **insertEdge** e **removeEdge** sono $\Theta(1)$
 - Quindi questi metodi sono tutti **ottimi** in ogni realizzazione

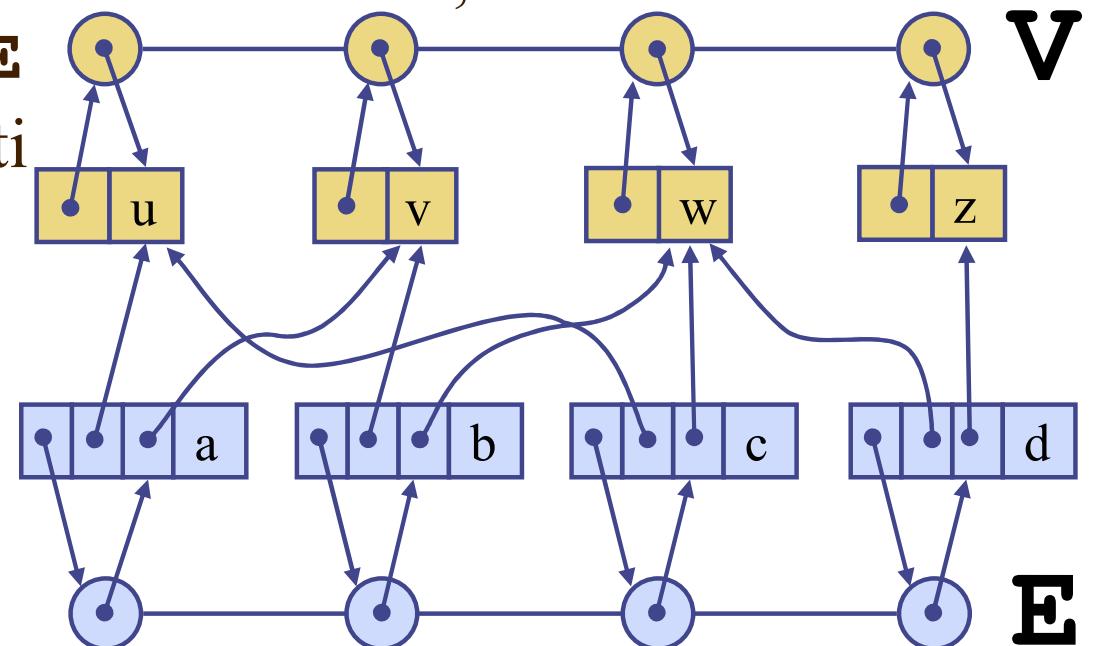
Differiscono per occupazione di **spazio** e per i metodi
incidentEdges
areAdjacent
insertVertex
removeVertex

Edge list

- È l'implementazione più semplice
- Usa una lista di vertici **V** e una lista di rami **E**
- Ipotizziamo che tutte le liste siano **doppiamente concatenate**, per semplicità; potrebbero anche essere array o liste semplicemente concatenate, ma alcune procedure si complicherebbero (ad esempio, gli array vanno ridimensionati...)
- Ogni vertice è un oggetto che contiene un dato e un riferimento alla propria posizione nella lista **V** (si dice che è **location-aware**...)
- Ogni ramo è un oggetto che contiene un dato, un riferimento alla propria posizione nella lista **E** (**location-aware**) e riferimenti ai propri vertici terminali
 - Il riferimento alla propria posizione in **E** rende $\Theta(1)$ il metodo **removeEdge**
 - Anche **insertEdge** e **insertVertex** sono $\Theta(1)$



esempio



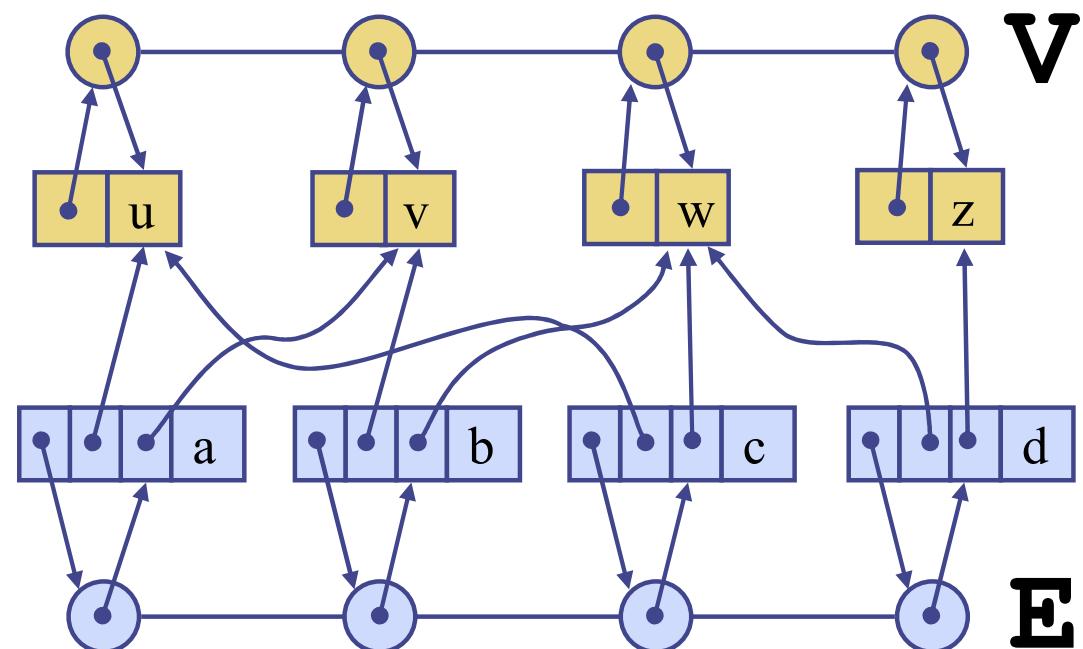
V

E

Edge list

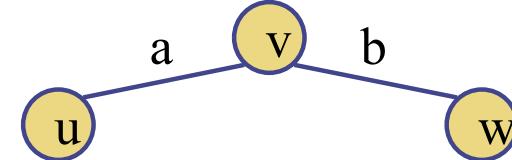
L'occupazione di memoria è $\Theta(n + m)$, perché le dimensioni di vertici e rami sono costanti

- In questa implementazione è agevole trovare vertici a partire da rami, infatti **opposite** e **endVertices** sono $\Theta(1)$
- La struttura è, però, poco efficiente nell'accesso a rami partendo da vertici
 - Il metodo **incidentEdges** è $\Theta(m)$, perché, per trovare i rami incidenti in un particolare vertice, bisogna scandire l'intera lista **E**
 - Si può rendere $O(m)$ aggiungendo ai vertici un campo che contenga il loro grado
 - Analogamente, il metodo **areAdjacent** è $O(m)$: nel caso pessimo, bisogna scandire l'intera lista **E**, cercando un ramo che abbia come terminali i due vertici in esame
- Il fatto che i vertici siano location-aware in **V** velocizza un po' il metodo **removeVertex** che, però, è $\Theta(m)$. Infatti, deve prima eliminare da **E** i rami incidenti nel vertice rimosso (oppure verificare che sia isolato) e usa **incidentEdges**



Lezione 35

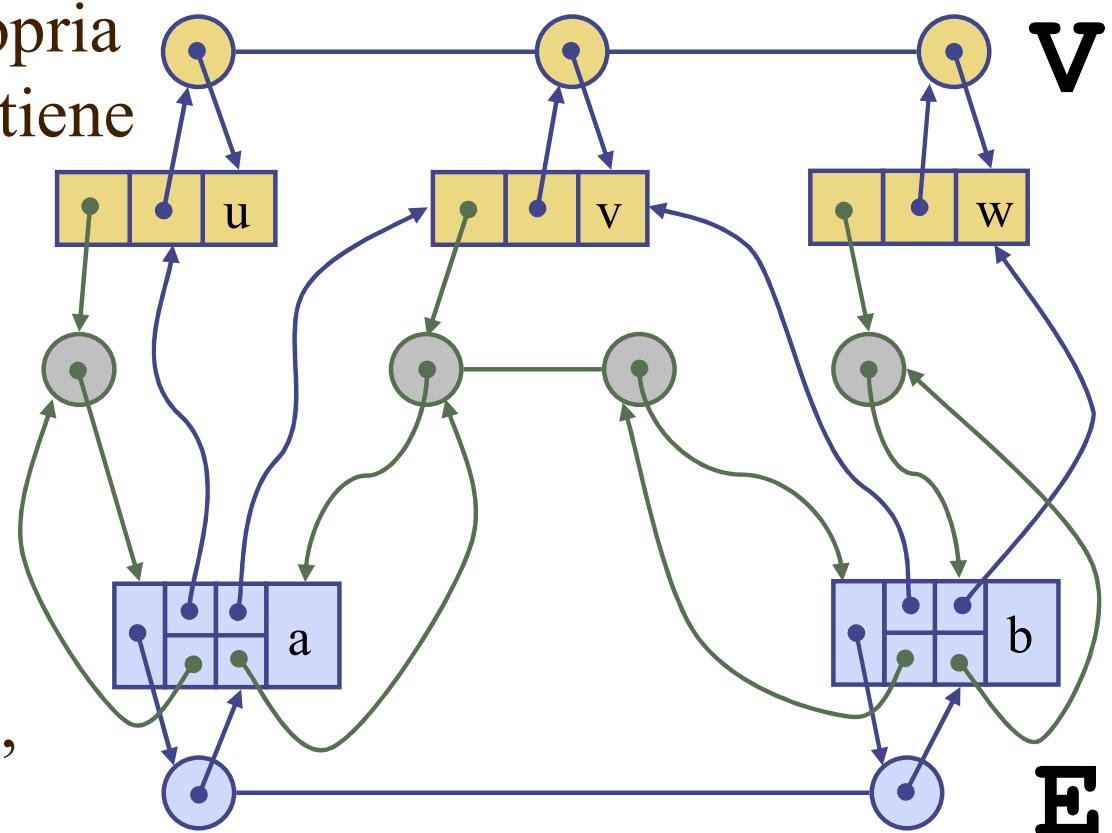
Adjacency list



- La lista di adiacenze modifica la *edge list* **migliorando** le prestazioni asintotiche dei metodi `incidentEdges`, `areAdjacent` e `removeVertex` (erano tutti $O(m)$), **senza peggiorare nulla** (quindi rende inutile la *edge list*, tranne quando va minimizzata l'occupazione di memoria non asintotica)

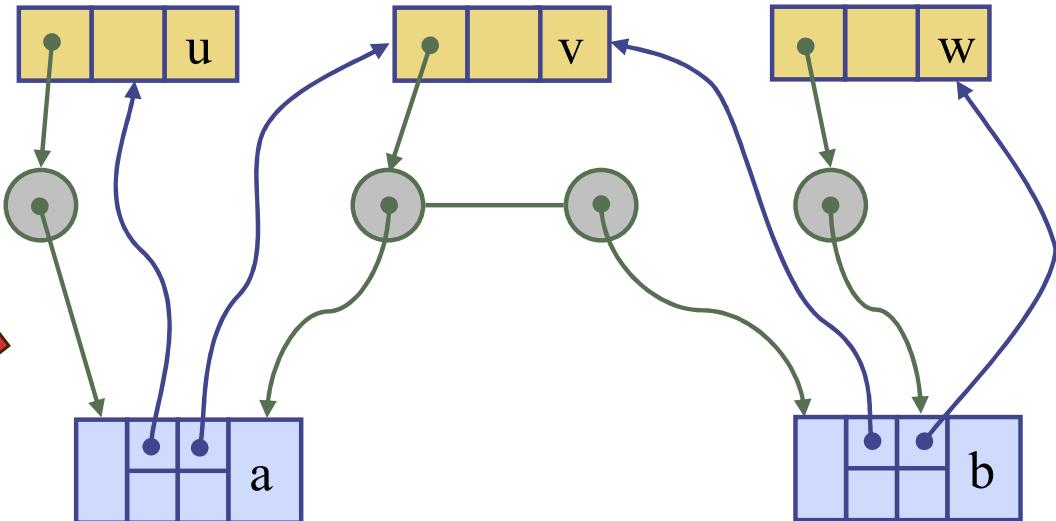
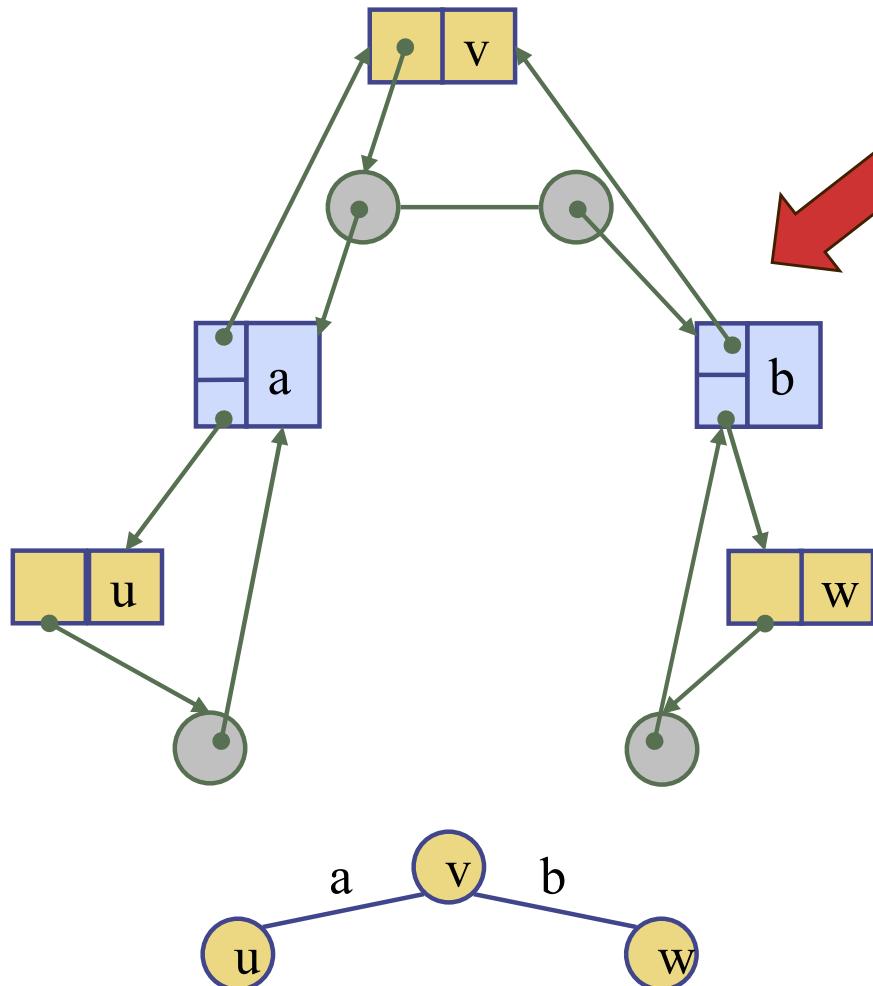
- Ogni vertice punta a una propria “lista di incidenze”, che contiene riferimenti ai rami incidenti nel nodo

- Ogni ramo punta anche alla propria posizione nelle liste di incidenze relative ai suoi due vertici terminali (location-aware in entrambe, oltre che in E...)



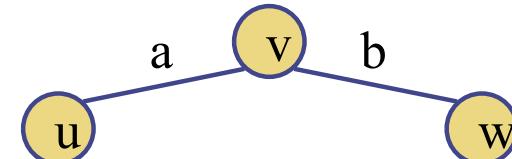
Adjacency list

- Proviamo a togliere le liste **V** ed **E** e i riferimenti di tipo location-aware
- **Sistemiamo un po'...**



- È la classica struttura a nodi concatenati usata per gli alberi, con l'aggiunta dei rami come oggetti
- Nella lista dei rami che portano ai "figli" di un nodo ora è inserito anche il ramo che porta al "genitore", perché nei grafi questa distinzione di ruoli non c'è: sono tutti rami incidenti

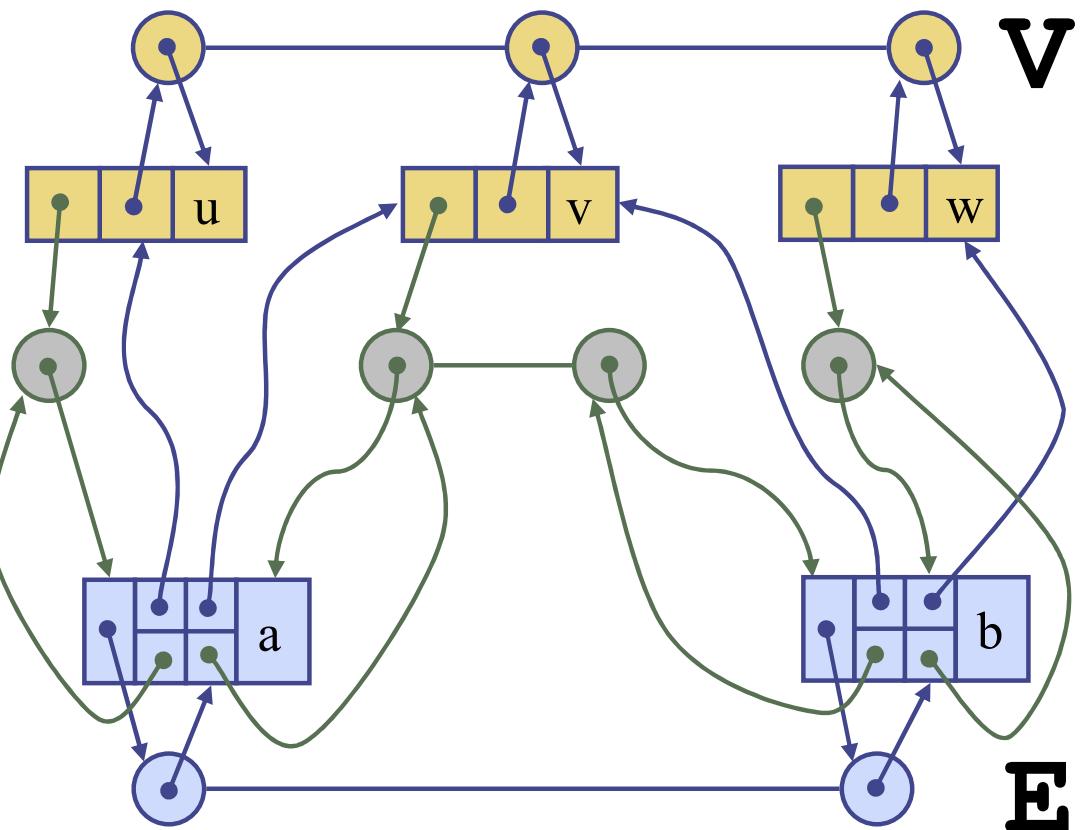
Adjacency list



- Ogni vertice punta a una propria “lista di incidenze”, che contiene riferimenti ai rami incidenti nel nodo
- Questo rende ***O(deg(v))*** i metodi **incidentEdges** e **removeVertex** relativi al vertice v

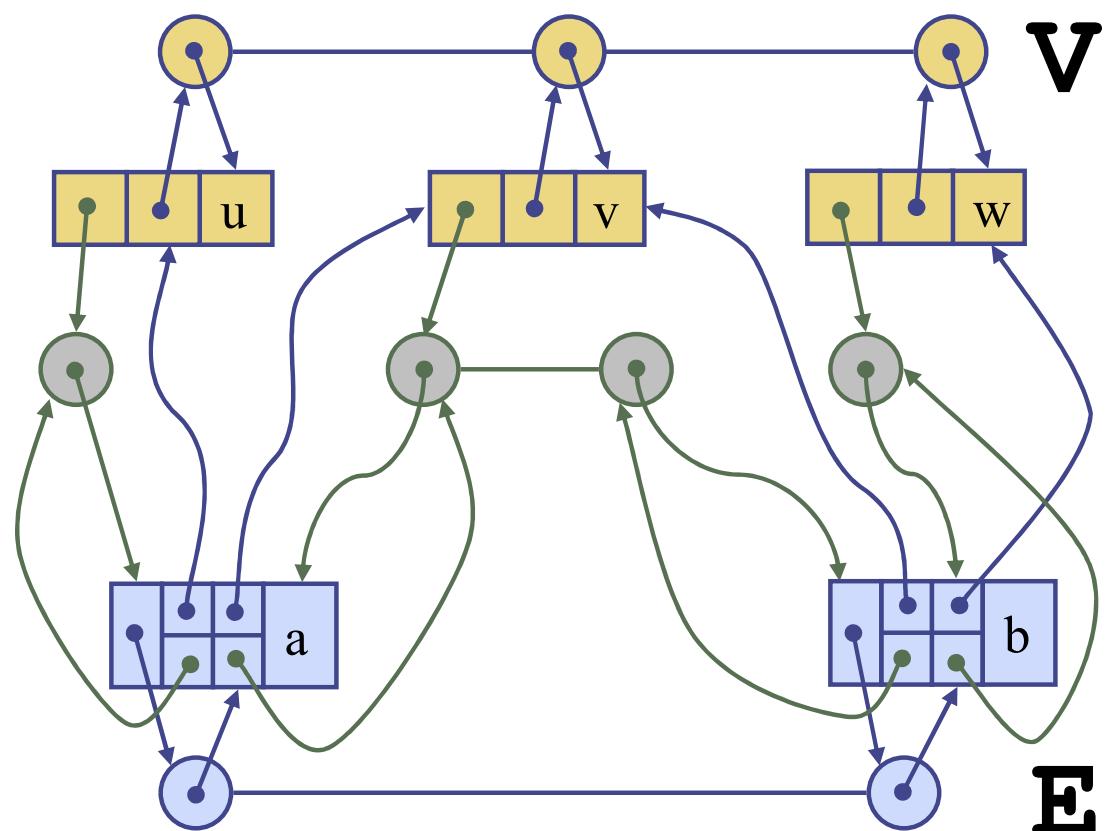
- Ogni ramo punta alla propria posizione nelle liste di incidenze relative ai suoi due vertici terminali (location-aware...)

- Questo serve a fare in modo che **removeEdge** rimanga $\Theta(1)$



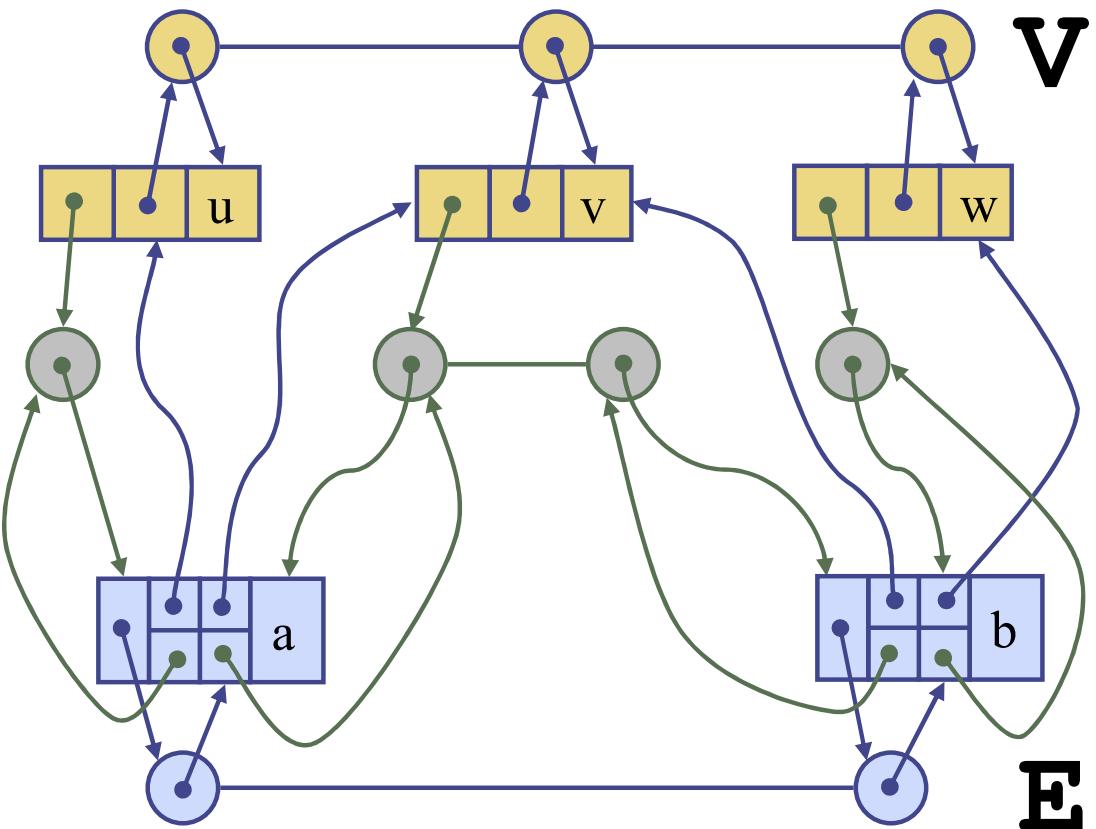
Adjacency list

- Per realizzare il metodo **areAdjacent**(u , v) è ora sufficiente:
 - Dai due vertici, u e v , accedere alle relative liste di incidenza, $I(u)$ e $I(v)$
 - Selezionare quella di dimensione minore: supponiamo sia $I(u)$
 - Verificare se uno dei rami a cui si accede da $I(u)$ ha v tra i suoi vertici terminali (l'altro è u)
 - Ogni verifica è $\Theta(1)$
- Le prestazioni del metodo **areAdjacent** sono $O(\min(\deg(u), \deg(v)))$
- E l'occupazione di memoria?



Adjacency list

- Oltre alle liste **V** ed **E**, con occupazione complessiva $\Theta(n + m)$, ora servono le liste di incidenza
 - Ciascun nodo e ciascun vertice ha dimensione maggiore di prima, ma, in ogni caso, le loro dimensioni sono costanti, non dipendono né da n né da m
- La lista di incidenza del vertice v ha dimensione $\deg(v)$, quindi lo spazio totale per tali liste è linearmente proporzionale alla somma dei gradi di tutti i vertici, che è $\Theta(m)$
- Quindi, l'occupazione di memoria è ancora $\Theta(n + m)$, anche se la costante moltiplicativa è più grande rispetto a *edge list*

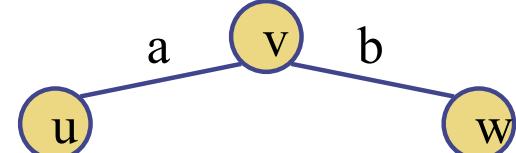


Edge list vs. Adjacency list

<ul style="list-style-type: none">▪ n vertici, m rami▪ grafo semplice	Edge List	Adjacency List
Memoria	$n + m$	$n + m$
incidentEdges(v)	m	deg(v)
areAdjacent (v, w)	m	min(deg(v), deg(w))
removeVertex(v)	m	deg(v)

Osserviamo che un grafo realizzato mediante **adjacency list** **garantisce l'esecuzione di DFS con prestazioni ottime**, $\Theta(n + m)$, diversamente da quanto accade con grafi realizzati mediante edge list, nei quali DFS, per “colpa” di **incidentEdges**, ha prestazioni $\Theta(nm)$

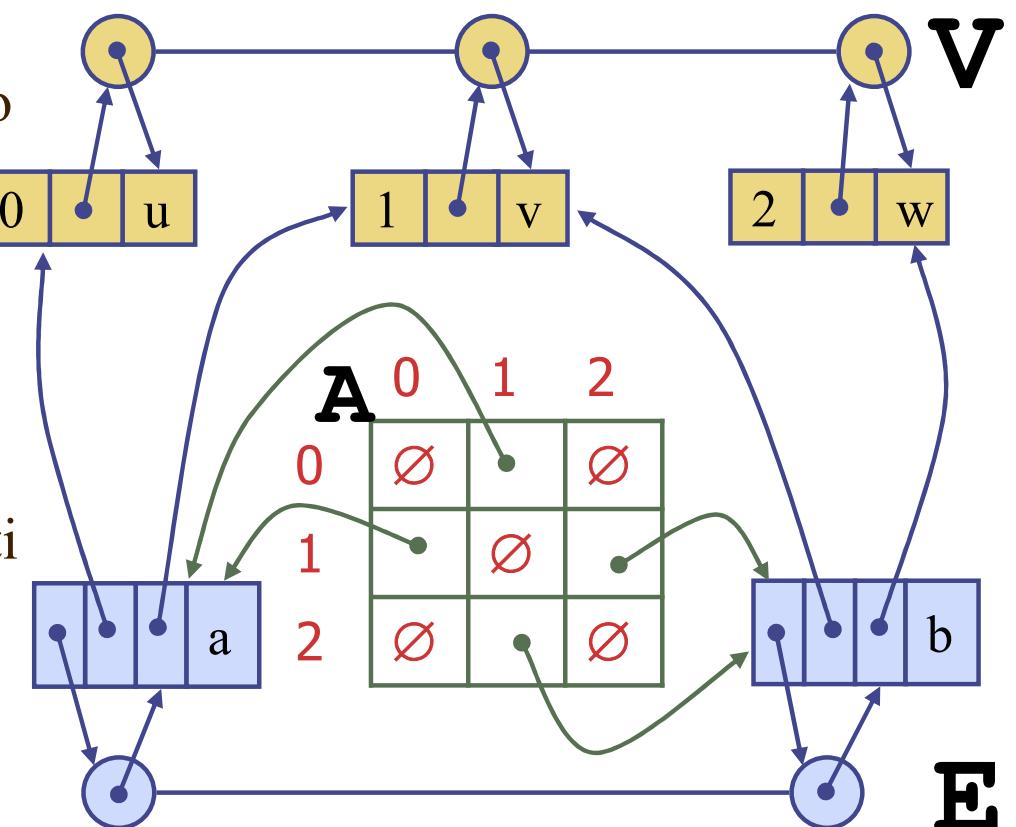
Adjacency matrix



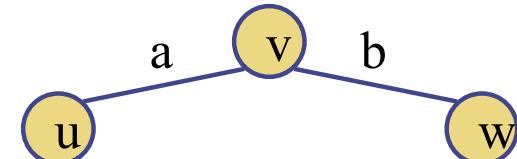
- La realizzazione mediante matrice di adiacenze è specifica per quei problemi che traggono vantaggio dalle prestazioni del metodo **areAdjacent**, perché lo rende **tempo costante**
 - Arricchisce la edge list con una matrice **A** che rappresenta, in ogni casella, coppie di vertici adiacenti

- Ad ogni vertice viene aggiunto un indice intero tra 0 e $n - 1$, usato come indice di riga e di colonna nella matrice \mathbf{A}

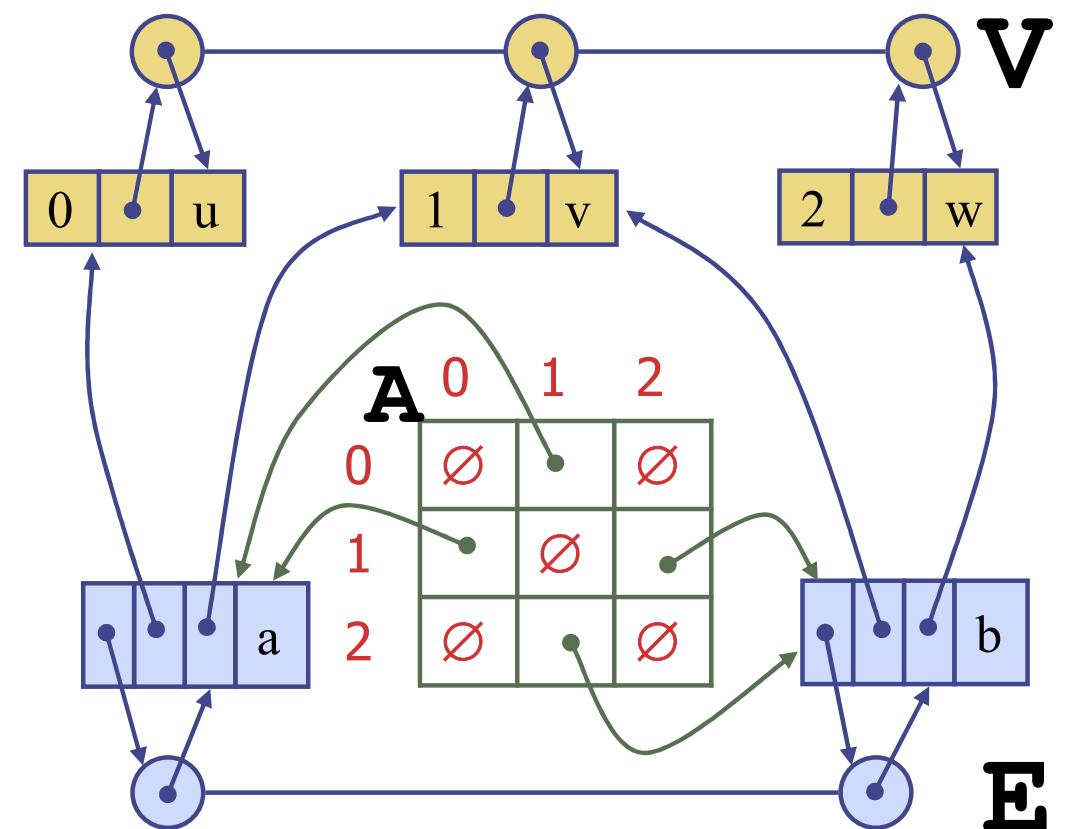
- Ogni cella di **A** contiene un riferimento al ramo che collega i due vertici corrispondenti ai suoi indici, se sono adiacenti, altrimenti contiene **NULL**



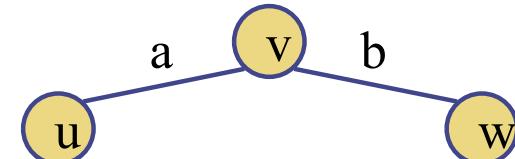
Adjacency matrix



- Per ogni ramo orientato e dal vertice di indice x al vertice di indice y , la cella avente x come indice di riga e y come indice di colonna (o viceversa, ma deciso una volta per tutte) contiene il riferimento al ramo e
- Per ogni ramo non orientato e tra il vertice di indice x e il vertice di indice y , la cella avente x come indice di riga e y come indice di colonna (e viceversa!) contiene il riferimento al ramo e
- Se il grafo è non orientato, la matrice \mathbf{A} è (quindi) simmetrica rispetto alla diagonale principale



Adjacency matrix



- Con questa soluzione, **areAdjacent** è $\Theta(1)$: si ispezionano i due vertici, si leggono i relativi indici, si ispeziona in **A** la cella corrispondente agli indici e si vede se contiene **NULL**

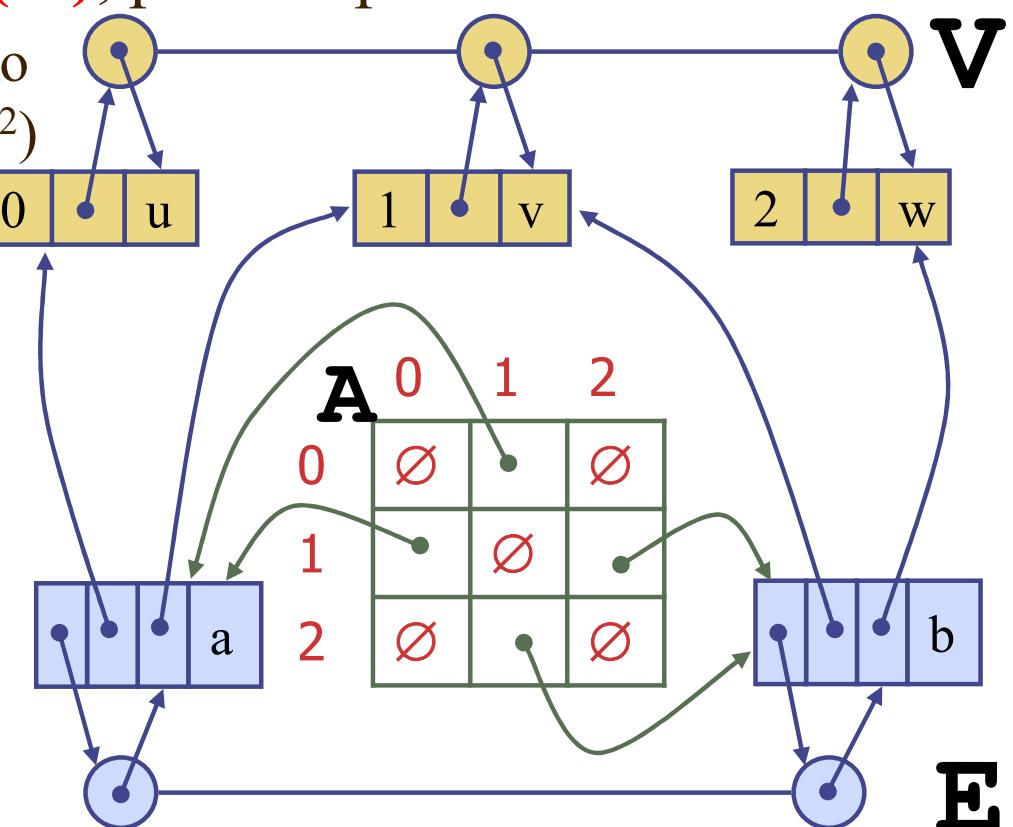
- Parecchie altre prestazioni, però, peggiorano

- L'occupazione di memoria è $\Theta(n^2)$** , per “colpa” della matrice

- Sarebbe $\Theta(m + n + n^2)$, ma sappiamo che in ogni grafo semplice $m \in O(n^2)$

- Il metodo `incidentEdges`** è $\Theta(n)$, perché, trovato l’indice del vertice in esame, bisogna esaminare un’intera riga (o colonna) di **A** (così DFS è $\Theta(n^2)$)

- Inserimento e rimozione di vertici sono $\Theta(n^2)$** : bisogna rinumerare i vertici e ricostruire la matrice



Adjacency list vs. Adjacency matrix

<ul style="list-style-type: none">▪ n vertici, m rami▪ grafo semplice	Edge List	Adjacency List	Adjacency Matrix
Memoria	$n + m$	$n + m$	n^2
<code>incidentEdges(v)</code>	m	$\deg(v)$	n
<code>areAdjacent (v, w)</code>	m	$\min(\deg(v), \deg(w))$	1
<code>insertVertex(o)</code>	1	1	n^2
<code>removeVertex(v)</code>	m	$\deg(v)$	n^2

Se il grafo non subisce frequenti modifiche, la matrice di adiacenza può essere vincente, in particolare se serve `areAdjacent` e non serve `incidentEdges`

Attraversamento BFS

Attraversamento BFS

- L'attraversamento **BFS** (*breadth-first search*) lavora “in ampiezza” anziché “in profondità”
 - **DFS** tenta di allontanarsi dal vertice iniziale quanto più possibile e si ferma (e torna un po’ indietro) soltanto quando trova un ciclo o arriva in un vicolo cieco
 - **BFS** visita prima i vertici adiacenti al vertice iniziale, poi si allontana “a macchia d’olio”, per “cerchi concentrici”: è meno avventuroso di DFS !
 - BFS, durante il proprio funzionamento, oltre a “decorare” vertici e rami visitati, assegna un “**livello**” (un numero intero non negativo) a ciascun vertice, ma è un’assegnazione “virtuale”, non c’è bisogno di memorizzarla nei vertici per poi ispezionarla (ma può essere molto utile per risolvere alcuni problemi)

BFS – Breadth-First Search

- BFS è, come DFS, una strategia generale per realizzare algoritmi su grafi e, con piccole modifiche, consente di risolvere gli stessi problemi risolubili con DFS, con le stesse prestazioni asintotiche
 - Ad esempio, verificare la connettività di un grafo, identificarne i componenti connessi e trovare uno spanning tree per ciascuno di essi, identificare cicli e **trovare un percorso tra due vertici qualsiasi**
 - In particolare, come vedremo, BFS risolve quest'ultimo problema in modo **ottimale**, perché trova **un percorso di lunghezza minima**, cioè con il minimo numero di rami (**DFS non ha questa proprietà**)

BFS – Breadth-First Search

- Dopo aver inizializzato a UNEXPLORED tutti i vertici e tutti i rami (come in DFS) scegliamo un vertice s di partenza tra quelli non visitati, **lo visitiamo e lo poniamo al “livello 0”**
- Esaminiamo tutti i rami incidenti in s e visitiamo tutti quelli che risultano non visitati (per il primo vertice, ovviamente, tutti)
 - I rami che portano a un vertice non visitato vengono etichettati come DISCOVERY, gli altri come **CROSS (?)**
 - Ai vertici non visitati che si raggiungono tramite rami DISCOVERY assegniamo livello 1, cioè “attuale” + 1, e li visitiamo
- Per ogni vertice di livello 1... ripetiamo!
- In realtà, il livello serve soltanto a identificare quali saranno i vertici analizzati al passo successivo: **anziché decorare ulteriormente i vertici, si usa una lista temporanea**

Simile all'attraversamento per livelli visto per gli alberi radicati

BFS – Breadth-First Search

```
BFS(G) // inizializzazione e main  
    // identico a DFS  
for each u ∈ G.vertices()  
    u.setLabel(UNEXLORED)  
for each e ∈ G.edges()  
    e.setLabel(UNEXLORED)  
connComps = 0  
for each v ∈ G.vertices()  
    if v.getLabel() == UNEXLORED  
        connComps++  
        BFS(G, v)
```

- Al termine, G è connesso se e solo se **connComps==1**

BFS(G, s) // non ricorsivo

```
 $L_0 \leftarrow$  new empty sequence  
 $L_0.addLast(s)$   
s.setLabel(VISITED)  
i ← 0  
while  $L_i.isEmpty()$   
 $L_{i+1} \leftarrow$  new empty sequence  
for each v ∈  $L_i$   
    for each e ∈ G.incidentEdges(v)  
        if e.getLabel() == UNEXLORED  
            w ← G.opposite(v,e)  
            if w.getLabel() == UNEXLORED  
                e.setLabel(DISCOVERY)  
                w.setLabel(VISITED)  
                 $L_{i+1}.addLast(w)$   
            else  
                e.setLabel(CROSS)  
i ← i + 1
```

Invece di una sequenza di liste, si può usare una (unica) coda, anche se si perdono le informazioni relative ai "livelli" (che comunque non servono per l'attraversamento)

BFS – Breadth-First Search

- Come in DFS, associamo l’azione unica di “visita” di un vertice o di un ramo all’invocazione di **setLabel**, che avviene solo se **getLabel == UNEXPLORED**, quindi non può avvenire due volte

BFS(G, s) // non ricorsivo

$L_0 \leftarrow$ new empty sequence

$L_0.addLast(s)$

$s.setLabel(VISITED)$

$i \leftarrow 0$

while $L_i.isEmpty()$

$L_{i+1} \leftarrow$ new empty sequence

for each $v \in L_i$

for each $e \in G.incidentEdges(v)$

if $e.getLabel() == UNEXPLORED$

$w \leftarrow G.opposite(v,e)$

if $w.getLabel() == UNEXPLORED$

$e.setLabel(DISCOVERY)$

$w.setLabel(VISITED)$

$L_{i+1}.addLast(w)$

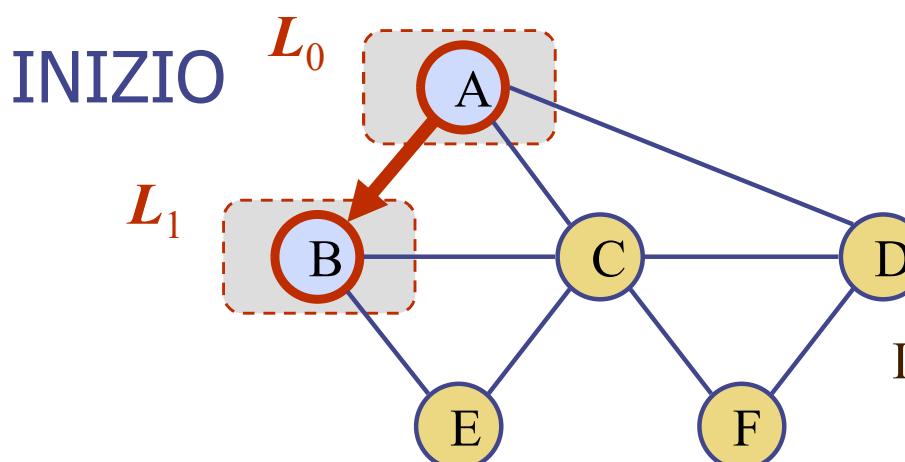
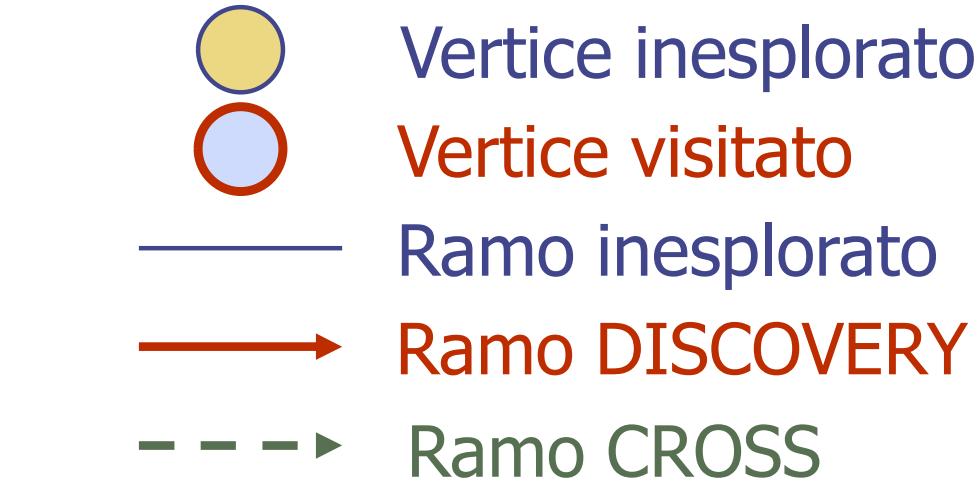
else

$e.setLabel(CROSS)$

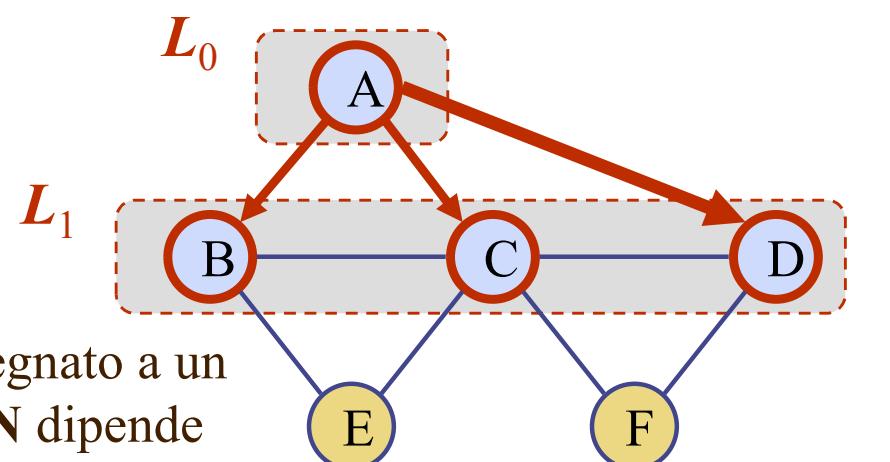
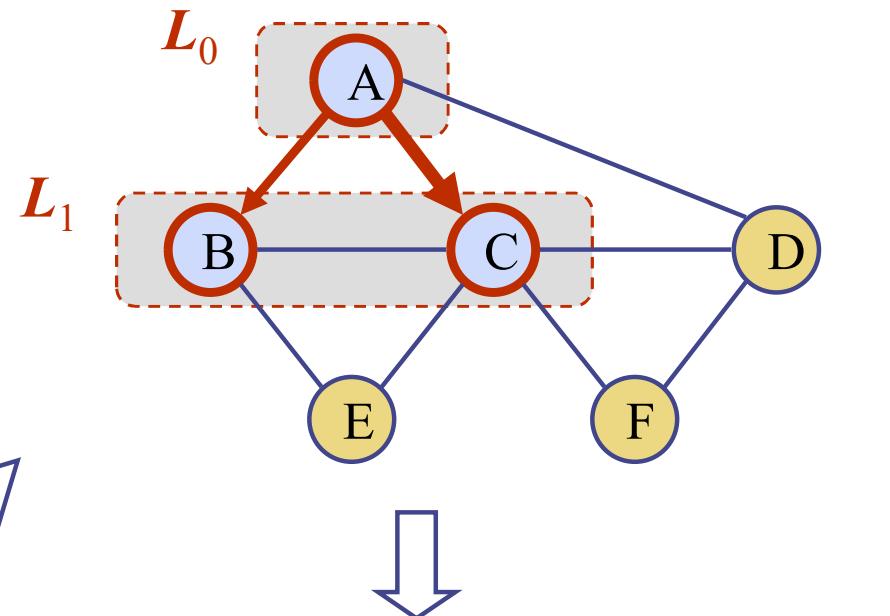
$i \leftarrow i + 1$

Esempio di esecuzione di BFS

L'esecuzione dipende dall'ordine in cui i rami sono elencati da `incidentEdges`

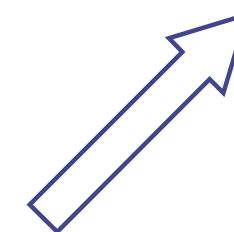
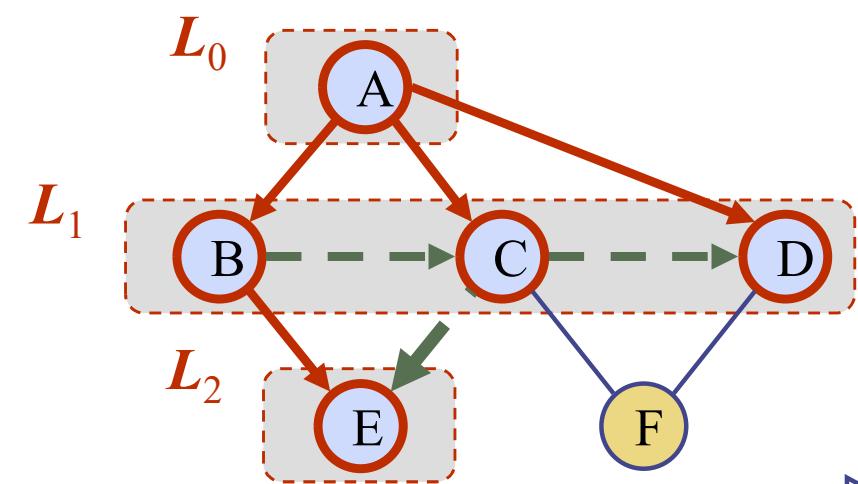
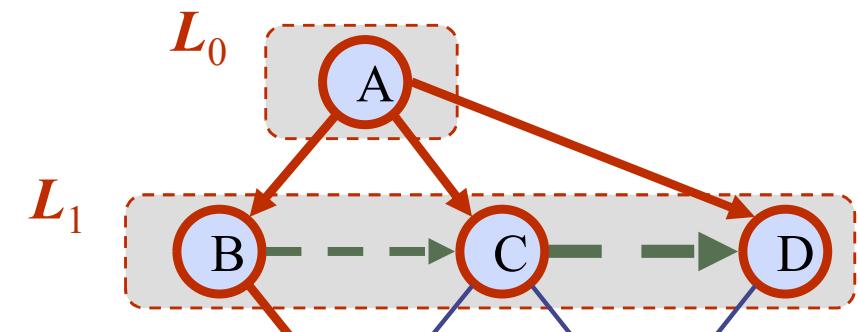
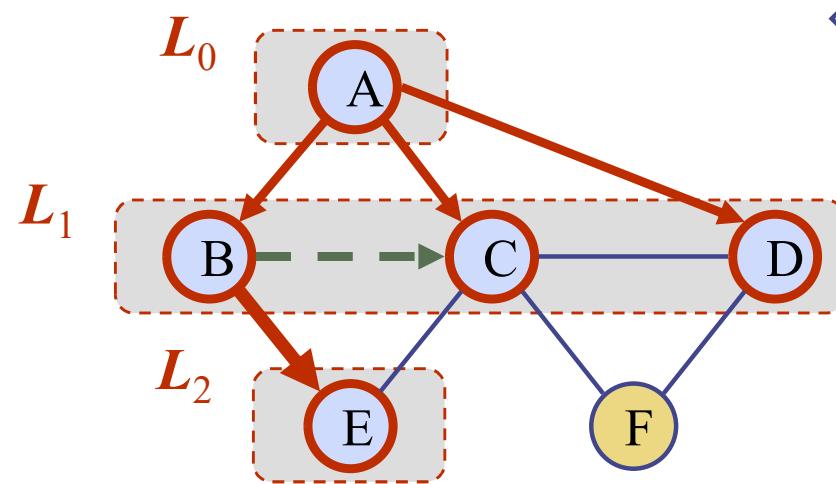
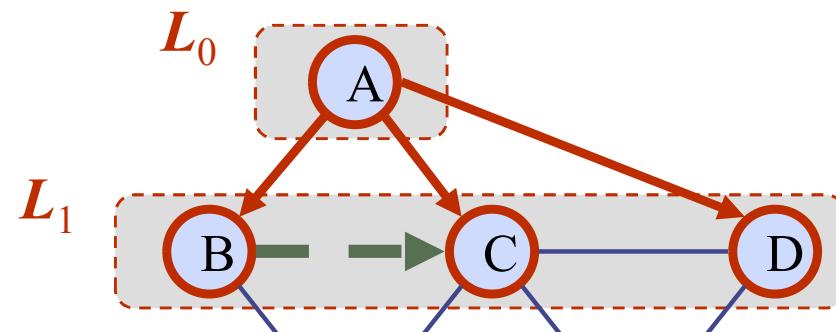


Il livello assegnato a un vertice **NON** dipende dall'ordine in cui i rami sono elencati da `incidentEdges`

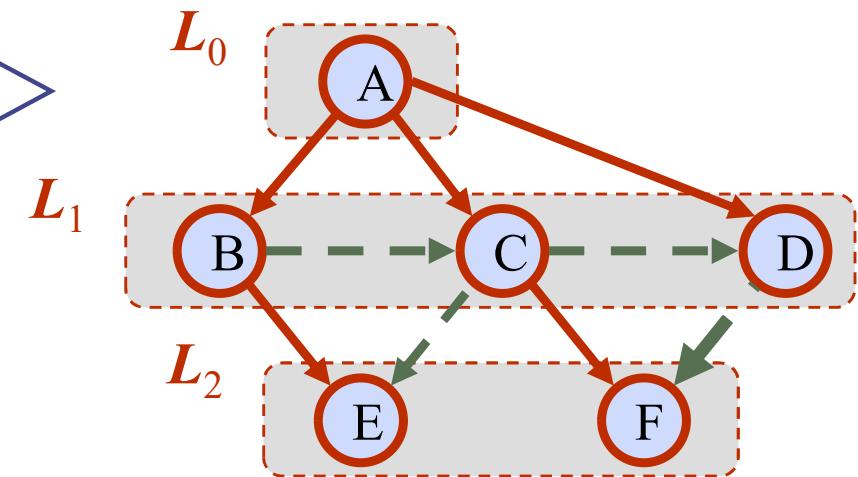
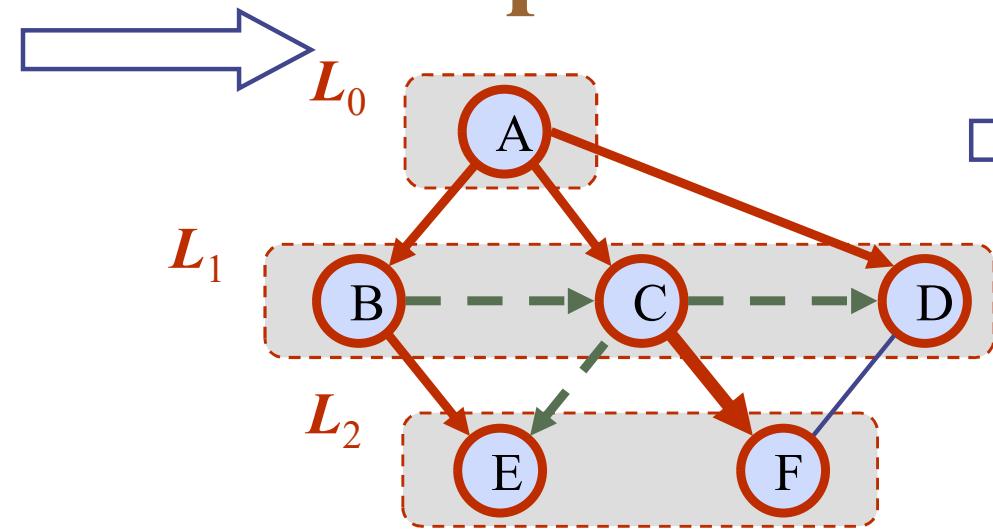


sono elencati da
incidentEdges

Esempio di esecuzione di BFS

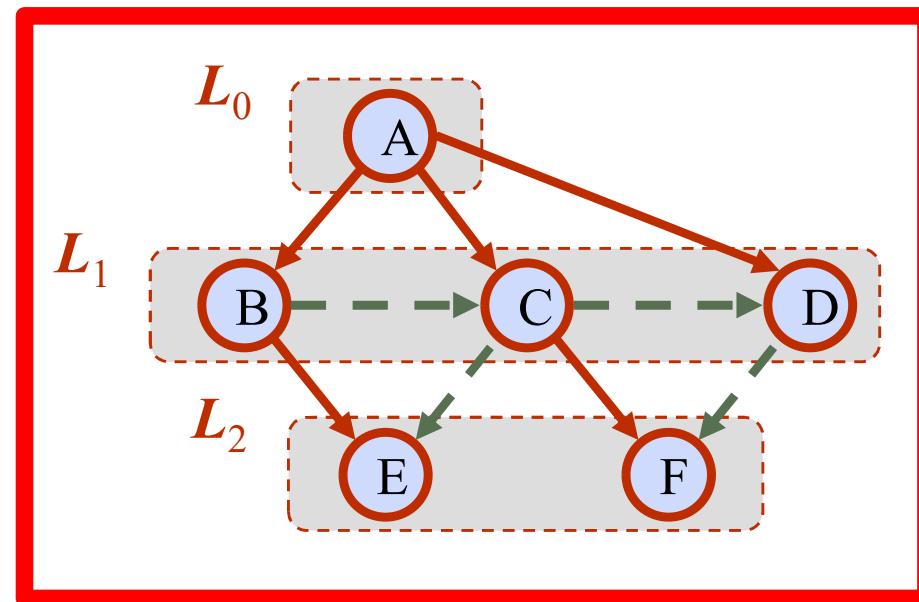


Esempio di esecuzione di BFS



FINE

Grafo con i livelli in evidenza, così come sono stati (implicitamente) assegnati ai nodi durante l'esecuzione di BFS

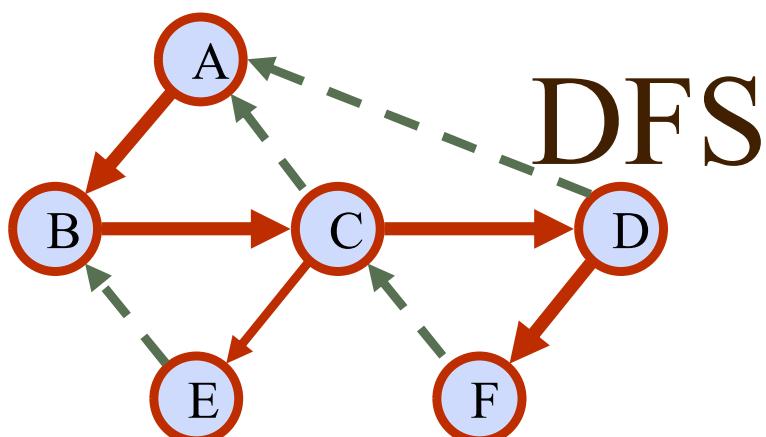


BFS – Breadth-First Search

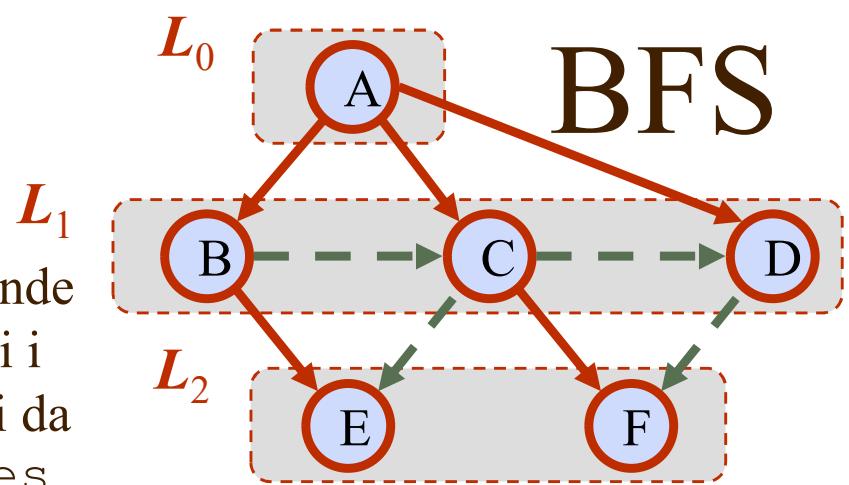
- Si può dimostrare che
 - $\text{BFS}(G, s)$ attraversa il componente连通的 a cui appartiene il vertice iniziale s
 - $\text{BFS}(G)$ attraversa il grafo G
 - Se G è rappresentato mediante ***adjacency list***, $\text{BFS}(G)$ ha prestazioni ottime, $\Theta(n + m)$
 - Con edge list è $\Theta(nm)$
 - Con adjacency matrix è $\Theta(n^2)$
 - BFS ha, quindi, le stesse prestazioni di DFS per ciascuna modalità di rappresentazione del grafo che abbiamo esaminato

BFS – Breadth-First Search

- Si può dimostrare che, se esiste un percorso tra i vertici u e v , BFS iniziato in u (opportunamente modificato per tenere traccia del percorso) trova un percorso **minimo** di rami DISCOVERY che collega u e v
- **Si può dimostrare che, se il vertice v viene posto al livello k dal BFS iniziato in u , allora il percorso minimo tra u e v ha lunghezza k**
- Nell'esempio, seguendo percorsi DISCOVERY da A, in BFS ogni vertice è raggiungibile con un percorso di lunghezza 2, mentre in DFS si arriva anche a 4 (ad esempio, per A-F)



L'esecuzione dipende
dall'ordine in cui i
rami sono elencati da
incidentEdges

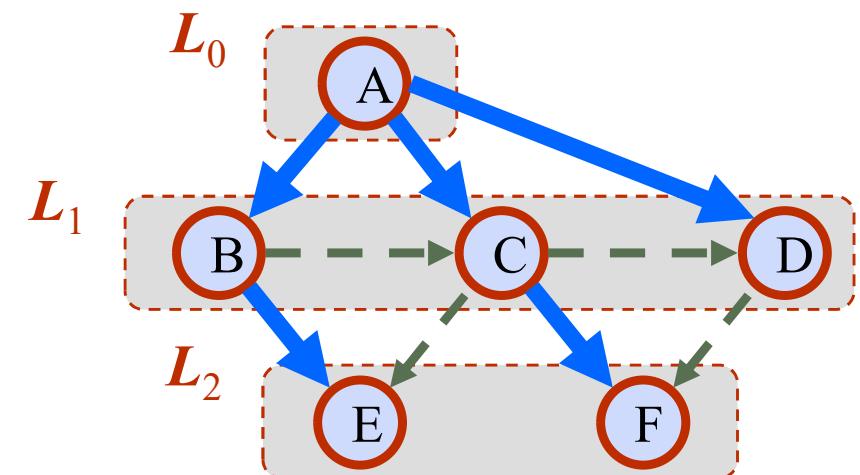


BFS

BFS – Breadth-First Search

- Si può dimostrare che i rami DISCOVERY di BFS costituiscono una **spanning forest** di G (come per DFS)
 - Ciascuno spanning tree (relativo a un componente连通) viene detto “**albero BFS**”
 - Si può dimostrare che l’albero BFS contiene, tra i suoi rami e vertici, **un percorso minimo** che collega s e u , essendo s il vertice di partenza di BFS e u un vertice qualsiasi appartenente al componente connesso a cui appartiene s
- **Vediamo come trovarlo!**

Per il momento, BFS ci dice che tale percorso minimo ha una lunghezza pari al livello di u



BFS – Breadth-First Search

- Si può dimostrare che, se esiste un percorso tra i vertici u e v , BFS (**opportunamente modificato per tenere traccia del percorso**) trova un percorso **minimo** di rami DISCOVERY che li collega
- Basta modificare BFS in modo che “decori” ogni vertice con una ulteriore etichetta, che rappresenta **il ramo che lo collega al suo genitore** nell’albero BFS che si sta costruendo
- Questa tecnica può essere usata anche in DFS (ma il percorso non è minimo)

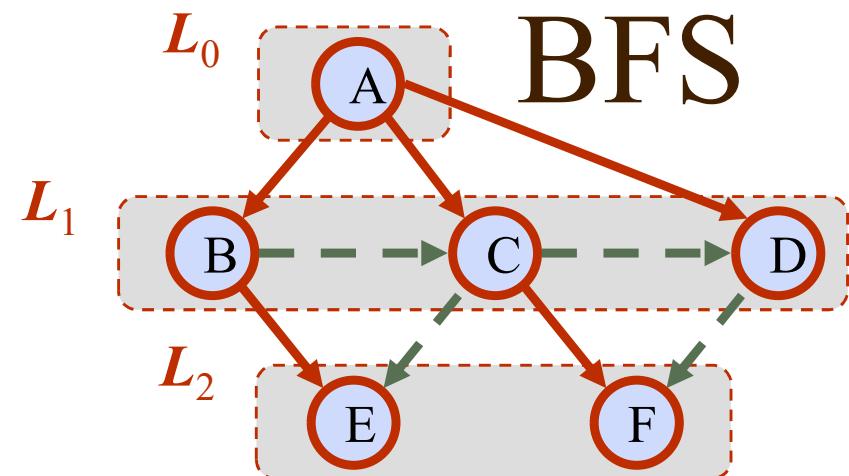
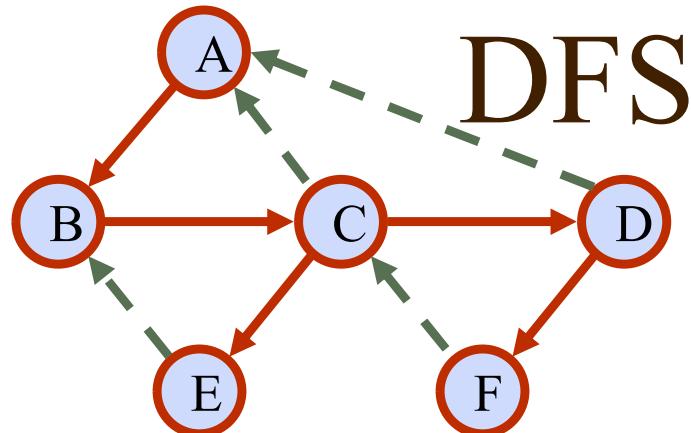
$BFS(G, s)$

```
 $L_0 \leftarrow$  new empty sequence  
 $L_0.addLast(s)$   
 $s.setLabel(VISITED)$   
 $s.setParent(NULL)$  // radice albero BFS  
 $i \leftarrow 0$   
while  $!L_i.isEmpty()$   
   $L_{i+1} \leftarrow$  new empty sequence  
  for each  $v \in L_i$   
    for each  $e \in G.incidentEdges(v)$   
      if  $e.getLabel() == UNEXPLORED$   
       $w \leftarrow G.opposite(v,e)$   
      if  $w.getLabel() == UNEXPLORED$   
         $e.setLabel(DISCOVERY)$   
         $w.setLabel(VISITED)$   
         $w.setParent(e)$   
         $L_{i+1}.addLast(w)$   
      else  
         $e.setLabel(CROSS)$   
 $i \leftarrow i + 1$ 
```

BFS vs. DFS

□ Si può dimostrare che

- In **DFS**(G, s), se (u, v) è un ramo di tipo **BACK**, allora **u è un antenato o un discendente di v** nello spanning tree costituito dai rami **DISCOVERY** e avente radice in s (per questo si chiama BACK, perché torna verso un antenato)
- In **BFS**(G, s), se (u, v) è un ramo di tipo **CROSS**, allora **u non è un antenato né un discendente di v** nello spanning tree costituito dai rami **DISCOVERY** e avente radice in s (per questo si chiama CROSS)
 - Inoltre, **i livelli di u e v differiscono al massimo per un'unità**



BFS e DFS per alberi

- Come si comportano gli algoritmi BFS e DFS applicati a un albero?
 - Ricordiamo che un albero (libero) può diventare radicato usando come radice un suo vertice qualsiasi
- Come si comportano DFS e BFS in un albero radicato nel vertice scelto come vertice iniziale?
 - **BFS** visita i vertici nello stesso ordine in cui li visiterebbe un **attraversamento per livelli** (oltre, naturalmente, a visitare anche i rami)
 - **DFS** visita i vertici nello stesso ordine in cui li visiterebbe l'**attraversamento in pre-ordine** (oltre, naturalmente, a visitare anche i rami)

BFS vs. DFS

- In conclusione, BFS e DFS hanno le stesse prestazioni (a parità di implementazione del grafo) e risolvono i medesimi problemi **tra quelli che abbiamo analizzato**, ma BFS ne risolve uno in modo migliore (ricerca di un percorso tra due vertici)
- **Allora, quando ha senso utilizzare DFS?**
- Molte applicazioni che usano i grafi hanno bisogno di fare "**attraversamenti parziali**", interrompendosi dopo aver raggiunto un obiettivo prefissato
 - ad esempio, nella **teoria dei giochi**, spesso c'è un limite al **tempo** disponibile per l'esplorazione del grafo delle possibili mosse da eseguire e delle conseguenti situazioni di gioco (cfr. gioco degli scacchi)
 - e la diversa strategia dei due attraversamenti può portare a risultati drasticamente diversi, perché visitano rami e vertici **IN UN ORDINE DIVERSO**: per alcune applicazioni l'ordine risultante da DFS può essere preferibile

BFS vs. DFS

- Molte applicazioni che usano i grafi hanno bisogno di fare "**attraversamenti parziali**", interrompendosi dopo aver raggiunto un obiettivo prefissato
- Esempio: progettare un algoritmo che, dato un albero, trovi una sua foglia (qualsiasi)
 - Nel caso pessimo DFS e BFS si equivalgono, ma **nel caso ottimo (e medio) DFS trova una foglia più rapidamente**, perché BFS "scende di livello lentamente", il più lentamente possibile... BFS scende al livello i se e solo se ha visitato TUTTI i nodi che si trovano a un livello $k < i$

Lezione 36

Ricerca di
percorsi minimi

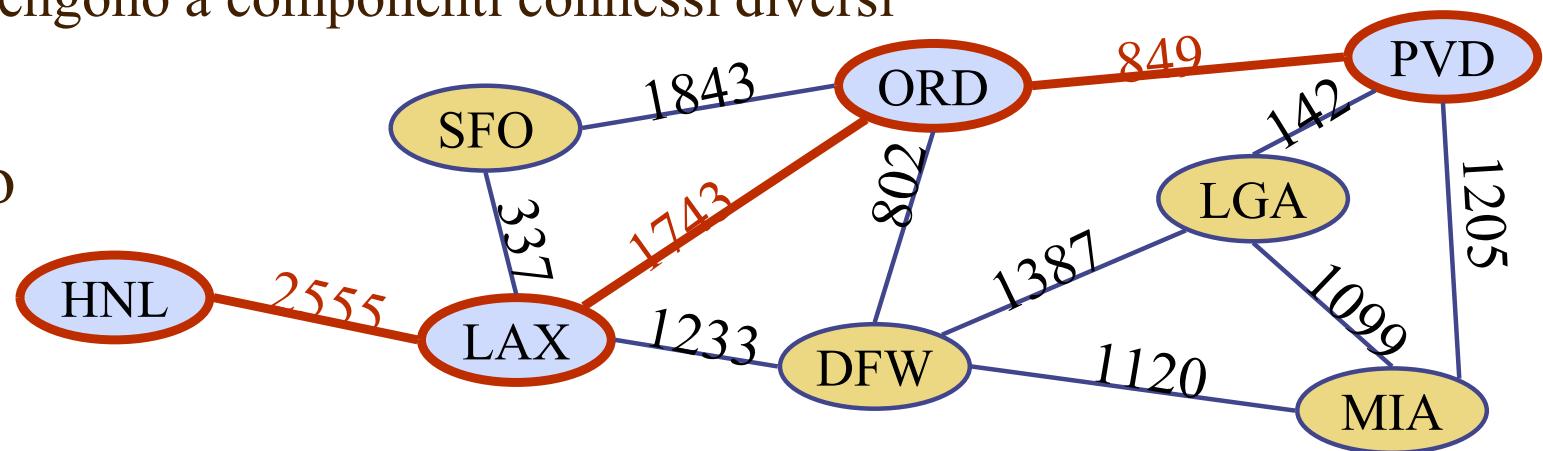
Ricerca di un percorso minimo

- Come detto, una variante di BFS consente di trovare un percorso **minimo** tra due vertici appartenenti allo stesso componente连通, usando come metrica (per la definizione del minimo) il **numero di rami** del percorso
- **Questa metrica** è ragionevole per alcuni problemi
 - es. trovare un percorso tra due aeroporti che abbia il minimo numero di scali, cioè (analogamente) il minimo numero di tratte di volo
 - ma lo è meno per altri
 - es. trovare il percorso più breve, in termini di chilometri o di tempo di volo, tra due aeroporti
 - Per i navigatori stradali non va bene!

Grafo pesato: lunghezza e distanza

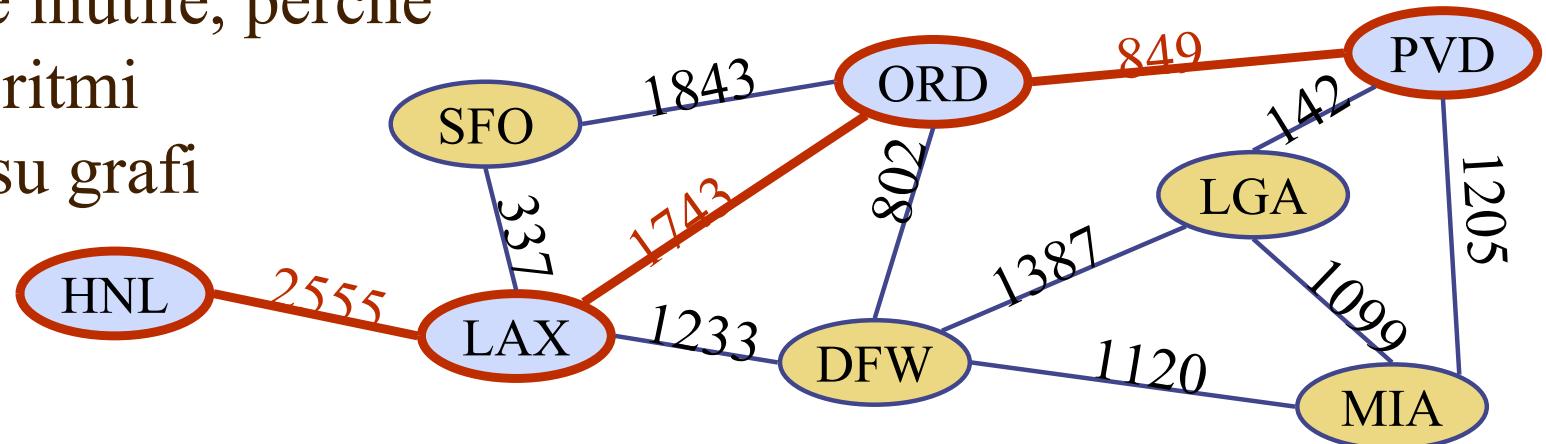
- In generale, per risolvere problemi con metrica generica, servono **grafi pesati**, in cui i rami siano etichettati in modo coerente con la metrica desiderata (es. chilometri di volo)
- Dato un grafo pesato, si definisce
 - **Lunghezza di un percorso** la somma dei pesi dei suoi rami
 - **Distanza tra due vertici** appartenenti a uno stesso componente连通的 la lunghezza di un percorso di lunghezza minima (detto anche “percorso più breve”, *shortest path*) che li collega
 - Spesso si definisce, per convenzione, **infinita** la distanza tra due vertici che appartengono a componenti connessi diversi

Percorso minimo
tra Providence
e Honolulu



Grafo pesato: necessario?

- Se i pesi sono numeri interi, la ricerca di percorsi minimi in un grafo pesato può essere ricondotta alla ricerca di percorsi minimi in un grafo non pesato, risolubile con BFS
- Basta sostituire ciascun ramo di peso w con una **catena** di rami e vertici costituita da w rami e $w - 1$ vertici, ciascuno dei quali abbia grado 2 (cioè un percorso privo di alternative!)
- La dimensione del grafo, in termini di rami e vertici, aumenta in modo considerevole... in generale non è una buona strategia
- Se i pesi non sono numeri interi... si potrebbe ricorrere a qualche “trucco”, ma è inutile, perché sono noti algoritmi che agiscono su grafi pesati



Rami con pesi negativi

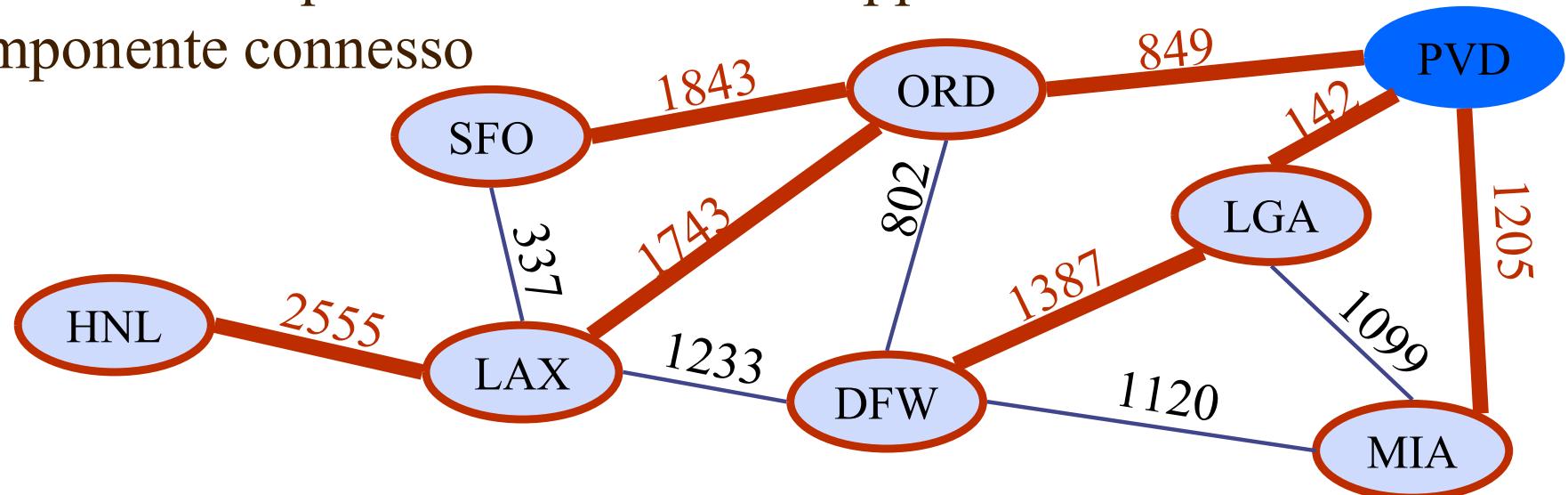
- Abbiamo definito **distanza tra due vertici** appartenenti a uno stesso componente连通的 la lunghezza di un percorso di lunghezza minima (detto anche “percorso più breve”, *shortest path*) che li collega
 - I pesi, in generale, possono anche essere **negativi**
 - In tal caso, la distanza tra due vertici è definita soltanto se nel componente连通的 a cui appartengono i due vertici **non è presente un ciclo i cui rami abbiano pesi con somma negativa**
 - Se esiste un tale ciclo, qualunque percorso lo può includere più volte, raggiungendo in tal modo una lunghezza negativa con valore assoluto grande a piacere, per cui non è possibile definirne un valore minimo

Ricerca di percorsi minimi

- Ipotesi semplificativa:
Grafo pesato semplice, connesso e non orientato
- In generale, interessano diversi problemi
 - Percorsi minimi **da singola sorgente (*single-source*)** verso tutti i vertici del grafo (**algoritmo di Dijkstra** per pesi non negativi, algoritmo di Bellman-Ford per pesi anche negativi)
 - Viceversa, da tutti i vertici verso **singola destinazione**: è equivalente, perché il grafo è non orientato
 - Percorso minimo **tra una coppia** di vertici (***single-pair***)
 - Si usano gli algoritmi precedenti, con **uno dei vertici della coppia che funge da sorgente**: non sono noti algoritmi che risolvano questo problema specifico in un tempo asintoticamente più veloce nel caso pessimo
 - Percorsi minimi **tra tutte le coppie** di vertici (***all-pair***)
 - Esistono algoritmi specifici (Floyd-Warshall, Johnson) più veloci rispetto all'applicazione ripetuta dei precedenti con ogni vertice iniziale

Proprietà di un percorso minimo

- Qualunque **sottopercorso** (cioè porzione di percorso) di un percorso minimo è, a sua volta, un percorso minimo tra i suoi due estremi
 - Se, per assurdo, un sottopercorso non fosse minimo...
- (Si dimostra che, come in BFS) Dato **un vertice**, esiste **un albero ricoprente** i cui rami e vertici contengono un percorso minimo verso qualsiasi altro vertice appartenente allo stesso componente连通



Algoritmo di Dijkstra

□ Problema: *single-source shortest paths*

Ricerca di percorsi minimi tra un vertice assegnato s e qualsiasi altro vertice del grafo

- Se il grafo non è pesato (o se, equivalentemente, i pesi sono tutti uguali...) si può usare BFS

□ Primo obiettivo

- Eseguire l'**algoritmo di Dijkstra**, che assegna a ogni vertice un'etichetta: la sua **distanza da s**

Olandese.
Pronuncia:
Deikstra

□ Secondo obiettivo

- Modificare l'algoritmo in modo che consenta, al termine, di costruire un percorso minimo tra il vertice s e qualsiasi altro vertice del grafo, cioè di **costruire l'albero dei percorsi minimi da s**

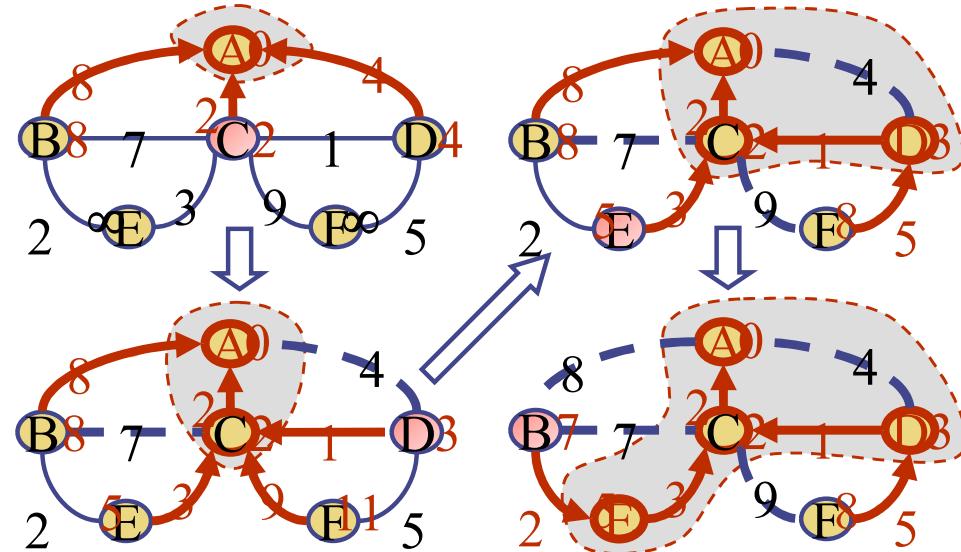
- Vedremo che **la soluzione è simile a quella vista per BFS**: basta aggiungere a ogni vertice il riferimento al "genitore" all'interno dell'albero dei percorsi minimi e, diversamente da BFS, **aggiornare tale riferimento finché non diventa definitivo**

Algoritmo di Dijkstra

- Obiettivo: assegnare a ogni vertice un’etichetta, che sia la sua distanza da un vertice di partenza, s

□ Idea

- Si fa crescere attorno a s una “nuvola” (*cloud*) di vertici che, passo dopo passo, includerà tutti i vertici del grafo
- La nuvola è inizialmente vuota e un vertice v ne entra a far parte se e solo se, tra i vertici esterni alla nuvola, è quello avente distanza **minima** da s
 - Quando un vertice entra nella nuvola, viene etichettato con la sua distanza da s : tale insieme di etichette è il risultato prodotto dall’algoritmo
 - Problema (risolto dall’algoritmo di Dijkstra): **come si calcolano le distanze da s dei vertici esterni alla nuvola?**



Algoritmo di Dijkstra

- Indichiamo con $d(u)$ la distanza da s del generico vertice u
- L'algoritmo adotta una strategia di “**approssimazioni successive**” delle informazioni che deve calcolare (che sono le distanze di tutti i vertici del grafo da s) mediante **stime per eccesso** dei valori veri
 - Da quali sovrastime possiamo partire, per essere **certi che siano stime per eccesso** (cioè sovrastime)?
 - $d(s) = 0, d(u) = +\infty \forall u \neq s$
 - **Significato di queste etichette in un certo istante:** distanze da s che sono **note fino a quel momento** perché si conosce un percorso (anche non minimo) tra s e il vertice
 - **(Da dimostrare)** Proprietà invariante (cioè sempre vera, in ogni istante): **un percorso minimo da s a u non può avere lunghezza maggiore dell'etichetta $d(u)$**

Algoritmo di Dijkstra

- L'algoritmo adotta una strategia di “approssimazioni successive” delle informazioni che deve calcolare (che sono le distanze dei vertici del grafo da s)
 - Di quali informazioni disponiamo inizialmente?
 - $d(s) = 0, d(u) = +\infty \forall u \neq s$
 - Proprietà invariante:
il percorso minimo da s a u non può avere lunghezza maggiore di $d(u)$
 - La proprietà è (ovviamente) inizialmente vera
 - La lunghezza del percorso minimo che collega s con s è 0, che non è maggiore del valore iniziale di $d(s)$, che è 0
 - Essendo il grafo连通的, la lunghezza del percorso minimo che collega s con qualsiasi vertice u sarà finita, "non maggiore di $+\infty$ "
 - Dovremo innanzitutto verificare che l'algoritmo **preservi** tale proprietà ogni volta che modifica un'etichetta
 - Inizializziamo l'algoritmo assegnando **queste etichette** ai vertici del grafo, creando anche la nuvola, C , vuota

Algoritmo di Dijkstra

- Algoritmo **iterativo**
- Finché la nuvola non contiene tutto il grafo
 - Scegli, tra i vertici del grafo **esterni** alla nuvola, quello avente **al momento** distanza minima da s , come indicato dalla sua etichetta: sia v il vertice così scelto
 - Inserisci v nella nuvola
 - Nota: la sua etichetta, $d(v)$, non verrà più modificata
 - In questo modo abbiamo acquisito una nuova informazione definitiva, $d(v)$, e aggiorniamo di conseguenza (**vedremo come...**) le etichette dei vertici **adiacenti a v** che siano **esterni** alla nuvola
 - Questo passo conferma che l'algoritmo non modifica mai le etichette dei vertici interni alla nuvola

Algoritmo di Dijkstra

- Prima iterazione: la nuvola è vuota
 - Tra tutti i vertici esterni alla nuvola (cioè, in questo momento, tra tutti i vertici del grafo), quello avente etichetta minima (zero) è **s** (gli altri vertici hanno ancora etichetta infinita)
 - Inserisci **s** nella nuvola
 - Osserviamo che l'etichetta del vertice, nel momento in cui entra nella nuvola, è proprio la sua **vera** distanza da s : **si dimostra** che **è sempre così**, e meno male ☺ perché le etichette, all'interno della nuvola, non vengono mai modificate
 - Aggiorna l'etichetta dei vertici adiacenti a s **esterni** alla nuvola (al momento tutti i vertici adiacenti a s sono esterni alla nuvola)
 - **In che modo si aggiornano le etichette?**
In base alle nuove informazioni acquisite:
la distanza da s del vertice appena entrato nella nuvola

Algoritmo di Dijkstra

□ In che modo si aggiornano le etichette?

In base alle **nuove informazioni acquisite**:
la distanza da s del vertice appena entrato nella nuvola

- Inizialmente non sapevamo nulla in merito alla distanza da s dei nodi del grafo diversi da s , per cui avevamo assegnato **cautelativamente** a tutti loro **distanza infinita da s**
 - **Era l'unica scelta che poteva, in quel momento, garantire la validità della proprietà invariante**
- In generale, inserendo un vertice v nella nuvola, **apprendiamo** che, tra s e v , esiste un percorso di lunghezza $d(v)$, quindi possiamo usare questa informazione

Algoritmo di Dijkstra

□ In che modo si aggiornano le etichette?

In base alle **nuove informazioni acquisite**:

la distanza da s del vertice appena entrato nella nuvola

□ In generale, inserendo v nella nuvola, **apprendiamo** che, tra s e v , esiste un percorso di lunghezza $d(v)$, quindi possiamo usare questa informazione

□ Se $v \in C$ e $u \notin C$ sono **adiacenti**

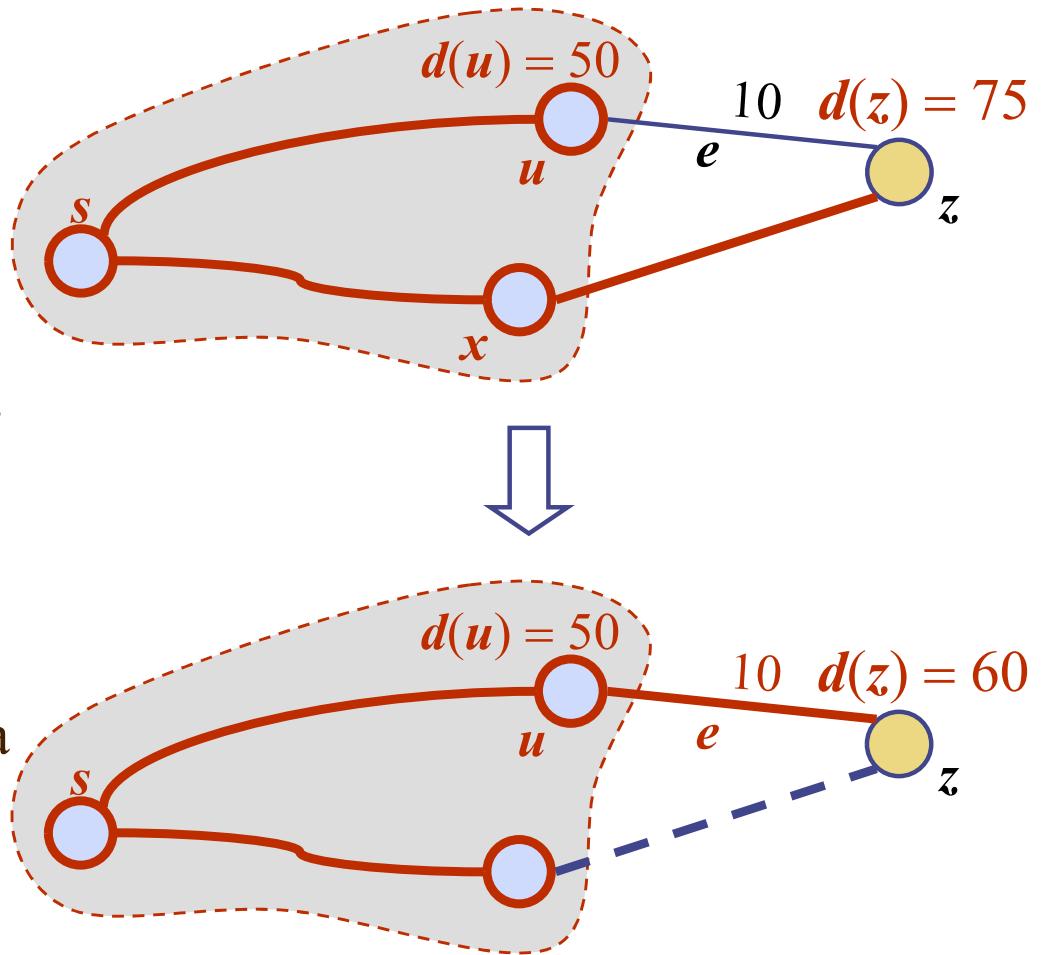
e w è il peso del ramo $e = (u, v)$, allora:

□ **se $d(v) + w < d(u)$** , apprendiamo che $d(v) + w$ è una **migliore approssimazione** della distanza di u da s , cioè il percorso $P(s, v) \cup \{e, u\}$, di cui apprendiamo l'esistenza, ha una lunghezza, $d(v) + w$, inferiore a quella, $d(u)$, del più breve percorso finora noto da s a u

□ Quindi, **se $d(v) + w < d(u)$** , poniamo $\textcolor{blue}{d}(u) = \textcolor{blue}{d}(v) + w$

Algoritmo di Dijkstra: Esempio

- Nell'esempio, il vertice u è appena entrato nella nuvola e $d(u) = 50$
- Il vertice z , con $d(z) = 75$, è esterno alla nuvola ma è adiacente a u , tramite il ramo e di peso $w = 10$
 - z aveva acquisito $d(z) = 75$ per effetto della sua adiacenza a x , quando x era entrato nella nuvola
 - Dato che $d(u) + w < d(z)$, si aggiorna l'etichetta di z , che diventa $d(z) = d(u) + w = 60$



Algoritmo di Dijkstra: Rilassamento

- In generale, inserendo v nella nuvola, “apprendiamo” che v ha distanza $d(v)$ da s , quindi possiamo usare questa informazione
 - Se $v \in C$ e $u \notin C$ sono adiacenti e w è il peso del ramo $e = (u, v)$, allora, se $d(v) + w < d(u)$, poniamo $d(u) = d(v) + w$
 - Questa procedura di aggiornamento delle informazioni, che si **avvicina al risultato corretto mediante approssimazioni successive**, è una strategia generica che si chiama **rilassamento**
 - Notiamo che **l'aggiornamento mediante rilassamento è monotono**: la distanza stimata può soltanto diminuire, avvicinandosi al valore effettivo (perché inizia come sovrastima)
 - Tale rilassamento preserva la proprietà delle etichette
 - Il percorso minimo non potrà avere lunghezza maggiore del valore dell'etichetta, perché questa è la lunghezza di un percorso **esistente**
 - Il percorso minimo potrebbe però avere lunghezza minore dell'etichetta finale: l'algoritmo non sarebbe corretto

Algoritmo di Dijkstra: Correttezza

- Si può dimostrare che, nel momento in cui un vertice v entra nella nuvola costruita attorno a s , non solo la sua etichetta è maggiore o uguale alla distanza di v da s , ma è proprio uguale
 - Si dimostra per assurdo ma è un po' complicato
- Questa proprietà rende corretto l'algoritmo di Dijkstra
- Osservazione: questa proprietà consente di ottimizzare (non nel caso pessimo) Dijkstra nella soluzione del problema *single-pair*, perché è sufficiente eseguirlo fino a quando il secondo vertice della coppia entra nella nuvola (il primo vertice della coppia sarà stato usato come sorgente)

Algoritmo di Dijkstra: Estensione

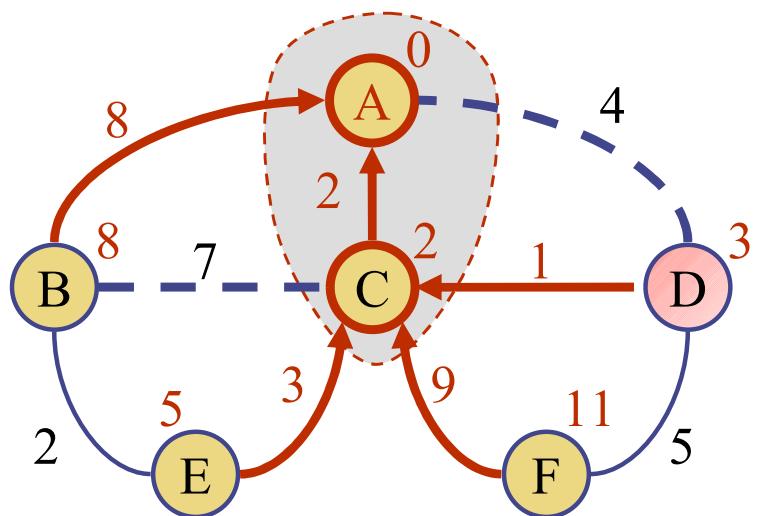
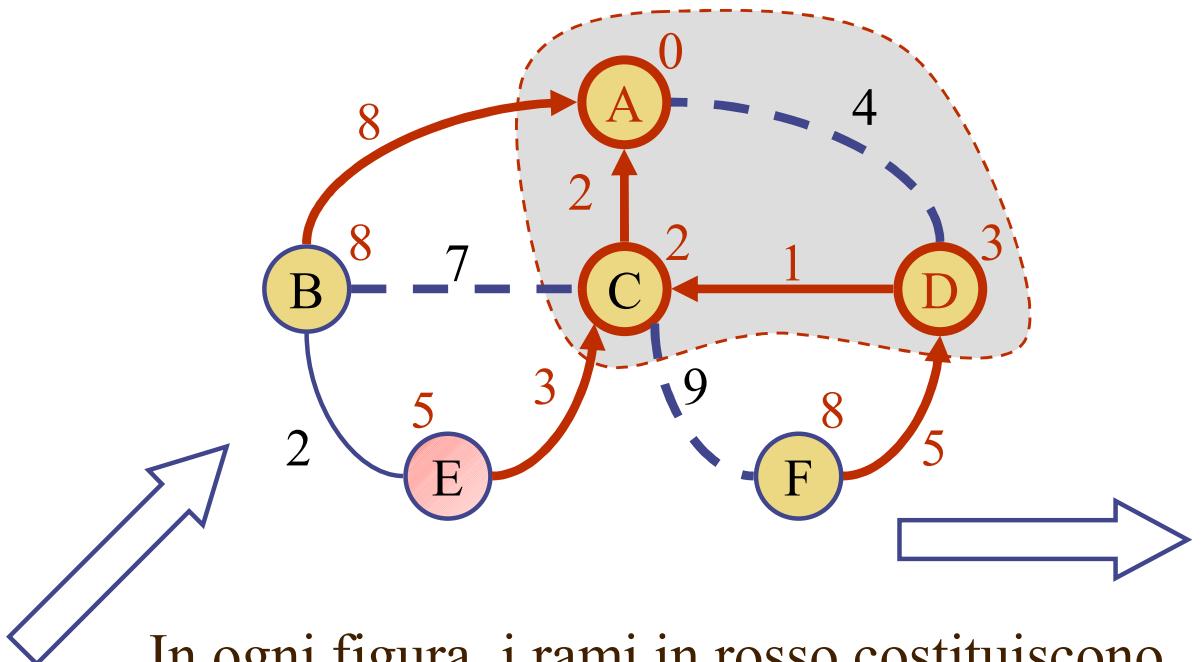
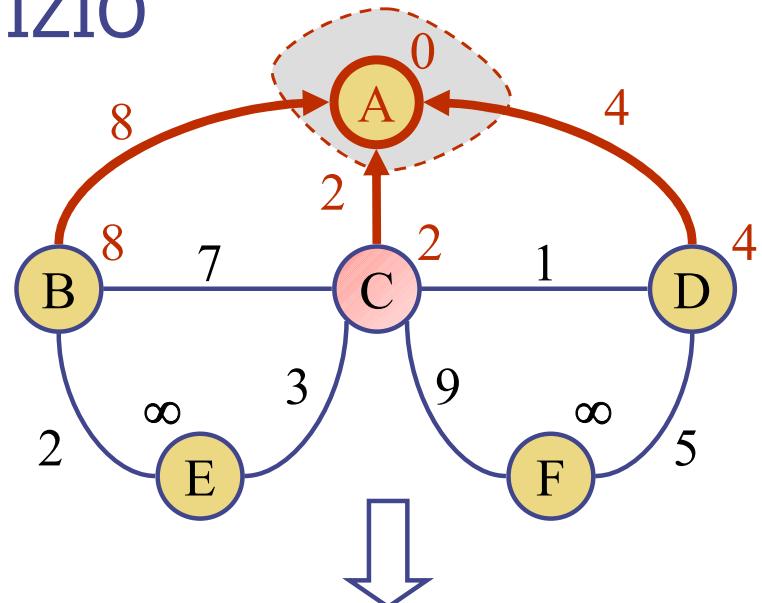
- Come si può modificare l'algoritmo perché, oltre a calcolare la distanza da s di tutti i vertici, trovi anche l'albero dei percorsi minimi avente radice in s e, conseguentemente, un percorso minimo tra s e un vertice qualsiasi?
- Si usa la stessa tecnica vista per BFS
 - Un'etichetta **parent** che, alla fine dell'esecuzione dell'algoritmo, in ogni vertice indica il ramo che lo collega al suo genitore nell'albero dei percorsi minimi
 - L'etichetta viene inizializzata a NULL per il vertice di partenza, s , che sarà ovviamente la radice dell'albero
 - Quando si effettua un rilassamento che coinvolge $u \notin C$ e $v \in C$, si individua un nuovo percorso minimo tra s e u che passa per v , quindi v è il nuovo genitore (ancora non definitivo) di u nell'albero dei percorsi minimi: $u.setParent(e)$, essendo e il ramo che rende u adiacente a v
 - Queste etichette "genitoriali" vanno, quindi, aggiornate a ogni rilassamento (in BFS erano già definitive nel momento in cui venivano assegnate)
 - Da quando un vertice entra nella nuvola, non subisce più rilassamenti, quindi la sua etichetta "genitore" diventa definitiva

Manca la figura iniziale,
con la nuvola vuota

**Non è un attraversamento, non si "colorano" i rami
e i vertici...** qui sono colorati soltanto per capire...

Algoritmo di Dijkstra: Esempio

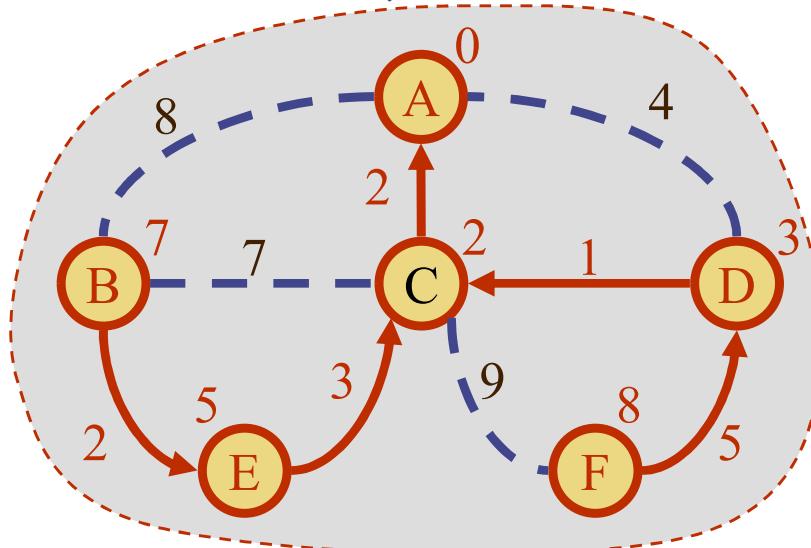
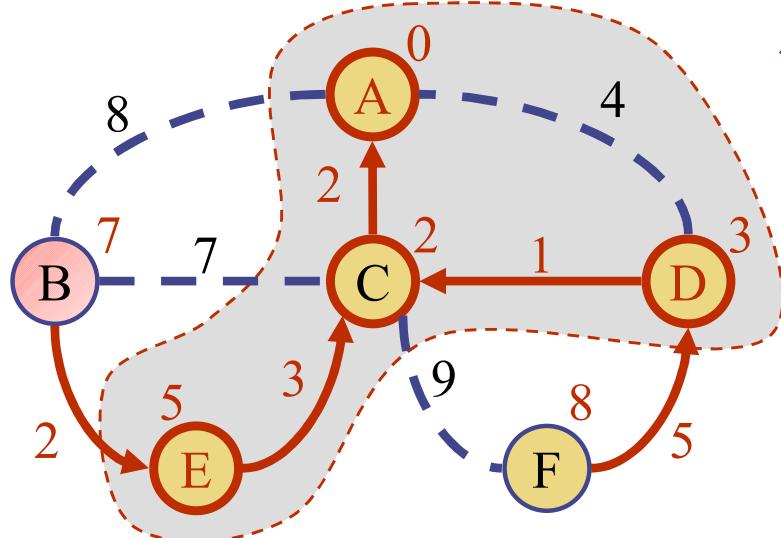
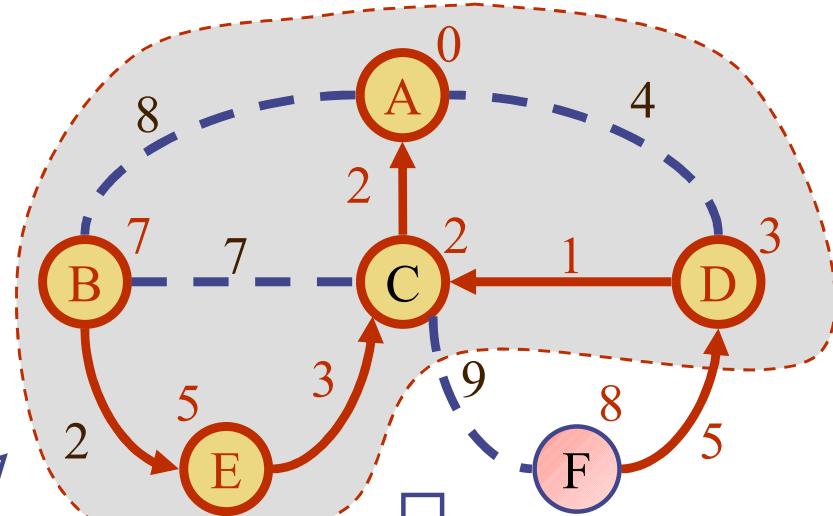
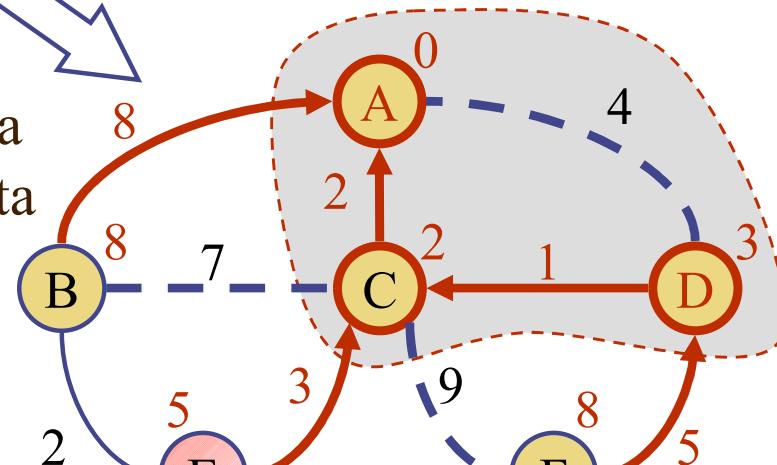
INIZIO



In ogni figura, i rami in rosso costituiscono l'albero **non definitivo** dei percorsi minimi dalla sorgente, determinato dalle informazioni note in quel momento (osservare, ad esempio, che il percorso minimo verso F cambia). Le frecce dei rami puntano da un vertice al suo "genitore" temporaneo nell'albero.

Algoritmo di Dijkstra: Esempio

Figura
ripetuta



Analizzare il risultato finale

FINE

Dijkstra per grafo non connesso

- L'algoritmo di Dijkstra funziona anche se il grafo non è connesso
 - Opera all'interno del componente连通的 che contiene il vertice iniziale
 - Termina quando, cercando all'esterno della nuvola il vertice avente etichetta minima, si osserva che **tutti i vertici rimasti hanno etichetta infinita**
 - **Non ha senso "far ripartire" l'algoritmo, non stiamo facendo un attraversamento**
 - Stiamo cercando un percorso minimo tra un vertice (il vertice “di partenza”) e qualsiasi altro vertice del grafo: semplicemente, l'algoritmo dichiarerà che, per i vertici rimasti all'esterno della nuvola, non esiste un percorso che li collega al vertice di partenza

Algoritmo di Dijkstra: Prestazioni

- L'algoritmo di Dijkstra
 - Usa **vertices**, nella **fase di inizializzazione**
 - Inserisce tutti i vertici nel contenitore “esterno alla nuvola” e assegna le distanze iniziali: $\Omega(n)$, dipende dal tipo di contenitore
 - **Estrae** ripetutamente dal contenitore “esterno alla nuvola” (fino a vuotarlo) **il vertice avente etichetta minima**
 - Per ogni vertice v estratto
 - Inserisce v nella nuvola
 - Invoca **incidentEdges** ed esegue il rilassamento per tutti i vertici adiacenti a v che appartengono al contenitore “esterno alla nuvola”
 - Osserviamo che, **in realtà, il contenitore “nuvola” non serve: non viene mai ispezionato in alcun modo. Non si usa**, è sufficiente il contenitore "esterno alla nuvola"

Algoritmo di Dijkstra: Prestazioni

- Per ogni vertice v inserito nella nuvola
 - Invoca **incidentEdges** ed esegue il rilassamento per tutti i vertici adiacenti a v **che appartengono al contenitore “esterno alla nuvola”**
 - Osserviamo che, **in realtà, non serve controllare se un vertice z adiacente a v appartenga o meno alla nuvola** (azione che richiederebbe tempo): è sufficiente controllare se il rilassamento è necessario per z , cioè se $d(v) + w < d(z)$, dove $w \geq 0$ è il peso del ramo (v, z)
 - Infatti, **se z appartiene alla nuvola** (e, quindi, vi è entrato prima di v , che è l'ultimo entrato), allora $d(v) \geq d(z)$, per cui $d(v) + w \geq d(z)$ e (correttamente) **il rilassamento non verrà effettuato**

Algoritmo di Dijkstra: Prestazioni

- In totale, quante operazioni di rilassamento si fanno?
- Al massimo, inserendo un vertice v nella nuvola, vengono sottoposti a rilassamento tutti i vertici ad esso adiacenti, cioè collegati a uno dei rami ad esso incidenti, che sono $\deg(v)$
 - In realtà, tranne per il primo vertice che entra nella nuvola, almeno uno dei vertici adiacenti a un vertice che entra deve già essere nella nuvola, ma ignoriamo questo "risparmio"
- Il numero totale delle operazioni di rilassamento è, quindi, al massimo uguale alla somma dei gradi di tutti i vertici, cioè $O(m)$

Algoritmo di Dijkstra: Prestazioni

- Le prestazioni dell'algoritmo di Dijkstra, quindi, dipendono dalle prestazioni:
 - del metodo `incidentEdges`, invocato una volta per ogni vertice (quando entra nella nuvola), in totale un tempo:
 - $O(m)$ per adj. list, $O(n^2)$ per adj. matrix,
 $O(nm)$ per edge list
 - del contenitore “esterno alla nuvola”, nel quale si fanno
 - n inserimenti (tutti nella fase di inizializzazione)
 - n **estrazioni dell'elemento minimo**
 - Usiamo **una coda prioritaria?**
 - $O(m)$ **aggiornamenti di etichette** (per rilassamento), che sono le chiavi delle coppie (etichetta, vertice)
 - È un' **operazione non consentita** in una coda prioritaria!!
Serve una **AdaptablePQ**

Algoritmo di Dijkstra: Prestazioni

- Proviamo a usare una AdaptablePQ realizzata mediante heap per rappresentare il contenitore “esterno alla nuvola”
 - n inserimenti = $O(n \log n)$
 - In realtà, si può usare una “bottom-up heap construction” (**che non abbiamo visto**) in un tempo $\Theta(n)$ perché i dati sono tutti disponibili fin dall’inizio e, tra l’altro, sono tutti uguali a $+\infty$, tranne lo zero che andrà nella radice [asintoticamente non si risparmia niente, vedi la fase successiva...]
 - n estrazioni dell’elemento minimo = $O(n \log n)$
 - $O(m)$ aggiornamenti di etichette (per rilassamento)
 - tempo complessivo per i rilassamenti: $O(m \log n)$
 - Totale: $O[(n + m) \log n]$

Algoritmo di Dijkstra: Prestazioni

- Riassumendo, usando una AdaptablePQ realizzata mediante heap, l'algoritmo di Dijkstra ha prestazioni:
 - Per le operazioni sul contenitore “esterno alla nuvola”, in totale $O[(n + m) \log n]$
 - Per le n esecuzioni del metodo **incidentEdges**
 - $O(m)$ per adj. list, $O(n^2)$ per adj. matrix
 - Quindi, in totale:
 - **Con adj. list: $O[(n + m) \log n]$**
Con adj. matrix: $O[n^2 + m \log n]$

Quasi lo stesso tempo
richiesto per un "semplice"
attraversamento, algoritmo
MOLTO efficiente

Algoritmo di Dijkstra: Pseudocodice

Dijkstra(G, s)

 q = HeapAdaptableMinPQ()
 s.setLocationAware(q.insert(0, s))

 s.setLabel(0)
 s.setParent(NULL) // per i percorsi, non “Dijkstra puro”
 for v ∈ G.vertices()

 if v != s

 v.setLocationAware(q.insert(+∞, v))
 v.setLabel(+∞) // inutile inizializzare parent, verrà sovrascritto

 while !q.isEmpty()

 entry = q.removeMin()

 label = entry.getKey()

 vertex = entry.getValue()

 if label == +∞

 break // G non è connesso, algoritmo terminato nel componente connesso di s

 for edge ∈ G.incidentEdges(vertex)

 z = G.opposite(vertex, edge)

 newLabel = label + edge.getLabel()

 if newLabel < z.getLabel() // rilassamento

 z.setLabel(newLabel)

 z.setParent(edge) // per i percorsi, non “Dijkstra puro”

 q.replaceKey(z.getLocationAware(), newLabel)