

# Lezione 1 18/09/19

## Analisi di algoritmi - Definizioni

Un **algoritmo** è una serie finita di passi che risolve un dato problema, nello specifico in ambito informatico si tratta di: una sequenza finita di passi computazionali che prende in ingresso un dato insieme di valori e produce un insieme di valori in uscita.

Un algoritmo è corretto se, per ogni istanza del problema, termina fornendo l'insieme corretto di valori in uscita. Un algoritmo incorretto potrebbe, in corrispondenza di alcune istanze del problema, non terminare o terminare in un insieme non corretto dei valori in uscita.

**Sintesi di un algoritmo** vuol dire: forniti i dati di un problema, come ricavo un algoritmo per risolvere tale problema.

**Analisi di un algoritmo** vuol dire: dato un algoritmo valuto quali sono le sue prestazioni.

**L'efficienza o complessità computazionale** di un algoritmo è misurata attraverso il numero di operazioni da effettuare per arrivare alla soluzione del problema.

Algoritmi di ordinamento sono una classe di algoritmi che risolvono lo stesso problema in diversi modi e questo ci permetterà di confrontare le diverse prestazioni d'ognuno.

**Una struttura dati** è una modalità con cui si organizzano i dati al fine di realizzare delle operazioni su di essi, quali: ricerca di un elemento, inserimento, cancellazione, etc....

Esistono diverse strutture dati, ognuna delle quali mantiene i dati secondo una propria modalità e che permette di eseguire quelle operazioni secondo particolari modalità. Una struttura dati è migliore rispetto ad altre sempre in relazione a quale problema risolvere.

**Il tempo d'esecuzione** è espresso in funzione della dimensione dei valori di ingresso. È dato dalla somma dei tempi di esecuzione di ciascuna linea dello pseudocodice, ciascuno moltiplicato per il numero di volte che la linea viene eseguita.

**IMPORTANTE:** algoritmo  $\neq$  programma. Un algoritmo può essere espresso anche in un linguaggio naturale, un programma è la traduzione di un algoritmo in uno specifico linguaggio di programmazione.

## Analisi di un algoritmo – Parametri

Per una corretta analisi, bisogna stimare le risorse in termini di:

- Occupazione di memoria, che può essere sul posto o no.
- Tempo di esecuzione o calcolo.
- Impegno di banda, quanti dati si cambiano i processi in parallelo.

Il modello usato per eseguire l'algoritmo consiste nel valutarlo con:

- Un singolo processore.
- Memoria RAM (ad accesso casuale).
- Assenza di gerarchie di memoria.

Si assume inoltre che le istruzioni richiedano un tempo costante.

## Insertion Sort

L'algoritmo di insertion sort è uno degli algoritmi più semplici ed intuitivi da comprendere, va prima di tutto detto che è un algoritmo che **ordina sul posto**. Ordinare sul posto vuol dire che dato il vettore, utilizzo esclusivamente la memoria occupata dal vettore per la risoluzione dell'algoritmo; viene considerato sul posto anche se utilizza della memoria aggiuntiva ma costante.

L'algoritmo di insertion sort è un algoritmo iterativo che punta a ordinare un elemento alla volta, partendo dal secondo all'ultimo in maniera iterativa, quindi considera un valore alla volta e lo inserisce nella corretta posizione tra i valori che lo precedono. Formalizzando diremo che: data una sequenza di  $n$  valori di ingresso  $(a_1, a_2, \dots, a_n)$ , determinare una permutazione  $(a'_1, a'_2, \dots, a'_n)$  della sequenza di ingresso tale che  $a'_1 \leq a'_2 \leq \dots \leq a'_n$ .

2	5	2	4	6	1	3
key	i	j				
4	2	5	4	6	1	3
6	2	4	5	6	1	3
1	2	4	5	6	1	3
3	1	2	4	5	6	3

La sequenza di passi iterativa è:

- Confronta l'elemento (partendo dal secondo) con i precedenti
- Se incontra un valore più grande lo sposta a destra e continua il ciclo
- Se incontra un valore più piccolo lo spostamento termina ed esce dal ciclo rimanendo nell'ultima posizione valida.

Siccome faremo scorrere gli elementi, sovrascrivendo la posizione, ci conviene mantenere da parte il valore dell'elemento da confrontare. Come detto precedentemente usiamo spazio aggiuntivo, ma essendo costante rimane un algoritmo sul posto. Seguendo l'esempio avremo che: 2, secondo elemento, confrontato con l'elemento precedente, 5 che è più grande, fa sì che 5 si sposi a destra e continui il ciclo. Il ciclo termina perché abbiamo confrontato i 2 elementi e li abbiamo ordinati, alla successiva iterazione verranno confrontati i primi tre elementi (2, 5, 4), nuovamente mettiamo il valore da confrontare, 4, da parte e lo confrontiamo con 5; scorre il 5 e continuiamo a confrontare con l'elemento precedente, 2 che è l'elemento più piccolo rispetto al valore che confrontiamo, quindi non spostò più ed esce dal ciclo. Continuiamo il procedimento fino a quando non terminiamo i confronti nel vettore. Di seguito lo pseudo codice

**N.B.** Negli pseudo-codici i cicli partono da 1 e sono svincolati da uno specifico linguaggio di programmazione, ed essendo uno pseudo linguaggio faremo alcune assunzioni esemplificative. Ad esempio, A è un oggetto con alcuni attributi, come ad esempio la funzione length .

## Note

### Insertion-Sort (A)

```
for j ← 2 to length[A] ←  $\cancel{m}$ 
  do key ← A[j] ←  $\cancel{m-1}$ 
    // Insert A[j] into A[1..j-1]
    i ← j-1 ←  $\cancel{m-1}$ 
    while i>0 and A[i]>key  $\sum_{i=1}^m T_j$ 
      do A[i+1] ← A[i] //,
        i ← i-1 //,
        A[i+1] ← key
           $\cancel{m-1}$ 
```

controllo della condizione finale

- “`<-`” è l'operatore di assegnazione.
- Il ciclo “for, while e do while” sono essenzialmente equivalenti, ma magari si preferisce la forma di uno rispetto all'altro. For se si conosce a priori il numero di iterazioni da effettuare, while e do while se non si conosce a priori quante operazioni vanno fatte. La differenza tra i due è che il “do while” richiede almeno un iterazione.
- Il “to” nella riga del for indica che bisogna crescere con l'indice, il “down to” indica il decrescere.
- Il ciclo for impiega  $n$  iterazioni perché è considerato anche il controllo della condizione finale

- Nello pseudo-codice non vengono usate le parentesi {} per delimitare i blocchi di codice associati a determinati costrutti, ma si usa l'indentazione.

- Nel ciclo **while**,  $i > 0$  deve per forza precedere  $A[i] > key$ , per evitare di accedere ad un indice negativo (o zero)
- Il ciclo **while** deve esplicitare la condizione di continuazione, cioè fin tanto che questa condizione è vera eseguiamo le istruzioni contenute nel ciclo. Spesso è più immediato scrivere la condizione di **terminazione**. Nel nostro caso termina o quando è finito oppure quando il termine è minore o uguale di quello da confrontare, usando il teorema di De Morgan neghiamo le nostre condizioni di terminazione e troviamo la nostra condizione di continuazione.
- I parametri sono passati **per valore alle procedure**.
- L'operatore [ ] indica sia l'accesso ai singoli elementi di un array che l'accesso ai campi di un oggetto.
- Se le parentesi quadre non sono complete, ovvero mancano della chiusura superiore è un'approssimazione per eccesso in caso contrario con una mancata chiusura inferiore è un'approssimazione per difetto.

## Lezione 2 25/09/19

### Analisi di correttezza

La **correttezza** si dimostra trovando **l'invariante** del ciclo, proprietà che deve essere vera all'inizio di ogni iterazione.

**Invariante dell'insertion sort:** tutti gli elementi da **1** a **j-1** corrispondono agli elementi che si trovano nelle prime **j-1** posizioni originariamente, ma ordinati in ordine crescente.

All'inizio di ciascuna iterazione del ciclo for, la sottosequenza  $A[1\dots j-1]$  consiste degli elementi originali di  $A[1\dots j-1]$  ma ordinati in senso crescente

Occorre dimostrare che:

- L'invariante è vero prima di iniziare il ciclo.
- Se l'invariante è vero prima di un'iterazione del ciclo, rimane vero pure prima della sua successiva iterazione, quindi mantiene la sottosequenza ordinata.
- Quando il ciclo termina  $j = n+1$  e tutta la sequenza da **1** a **n** è ordinata, l'invariante prova che l'algoritmo è corretto.

### Analisi di complessità

Il tempo di esecuzione di un algoritmo è solitamente espresso in funzione della dimensione dei valori in ingresso, per gli algoritmi **di ordinamento** è uguale alla lunghezza della sequenza da ordinare. Per gli algoritmi che **operano su grafi** è pari al numero di vertici e di archi.

Il tempo di esecuzione è dato dalla somma dei tempi di esecuzione di ciascuna linea dello pseudocodice (che assumiamo costanti), ciascuno moltiplicato per il numero di volte che la linea viene eseguita.

#### Note:

$T(n)$  è pari al tempo di esecuzione, mentre  $c_1, c_2 \dots c_n$  sono il costo nello specifico tempo.  $c_3$  è assente perché è una riga di commento,  $n$  invece è la lunghezza del vettore,  $t_j$  è il numero di volte che viene eseguito in test del ciclo while. All'interno dell'analisi di complessità si utilizzano queste notazioni :

$$T(n) = c_1 n + c_2 (n-1) + c_4 (n-1) + c_5 \sum_{j=2}^n t_j + c_6 \sum_{j=2}^n (t_j - 1) + c_7 \sum_{j=2}^n (t_j - 1) + c_8 (n - 1)$$

Il caso **migliore** si verifica quando il vettore è già ordinato ( $t_j = 1$ )

$$\begin{aligned} T(n) &= c_1 n + c_2 (n-1) + c_4 (n-1) + c_5 (n-1) + 0 + 0 + c_8 (n - 1) = \\ &= (c_1 + c_2 + c_4 + c_5 + c_8 )n + (-c_2 - c_4 - c_5 - c_8 ) = an + b \end{aligned}$$

il tempo di esecuzione è **funzione lineare** di  $n$ .

Il caso **peggiore** si verifica quando il vettore è ordinato al contrario ( $t_j = j$ )

$$\begin{aligned} T(n) &= c_1 n + c_2 (n-1) + c_4 \left( \frac{n(n-1)}{2} - 1 \right) + c_5 \left( \frac{n(n-1)}{2} \right) + c_6 \left( \frac{n(n-1)}{2} \right) + c_7 \left( \frac{n(n-1)}{2} \right) + c_8 (n - 1) = \\ &= an^2 + bn + c \end{aligned}$$

Il tempo di esecuzione è **funzione quadratica** di  $n$ . Tipicamente si considera il tempo di esecuzione nel caso peggiore e per indicare sinteticamente il tempo di esecuzione di un algoritmo, si tralasciano le costanti e si indica solo il termine dominante. Nel caso dell'insertion sort il tempo peggiore è  $\Theta(n^2)$  (da leggere come Theta di  $n$  quadro).

Da quanto visto possiamo capire che l'insertion sort adotta un approccio incrementale, un modo più efficiente per l'ordinamento è quello di usare il paradigma del **divide et impera** il quale richiede:

- Divisione del problema in sotto problemi più semplici.
- Risolvere ricorsivamente i sotto problemi.
- Combinare le soluzioni dei sotto problemi per ottenere la soluzione del problema di partenza.

## Merge Sort

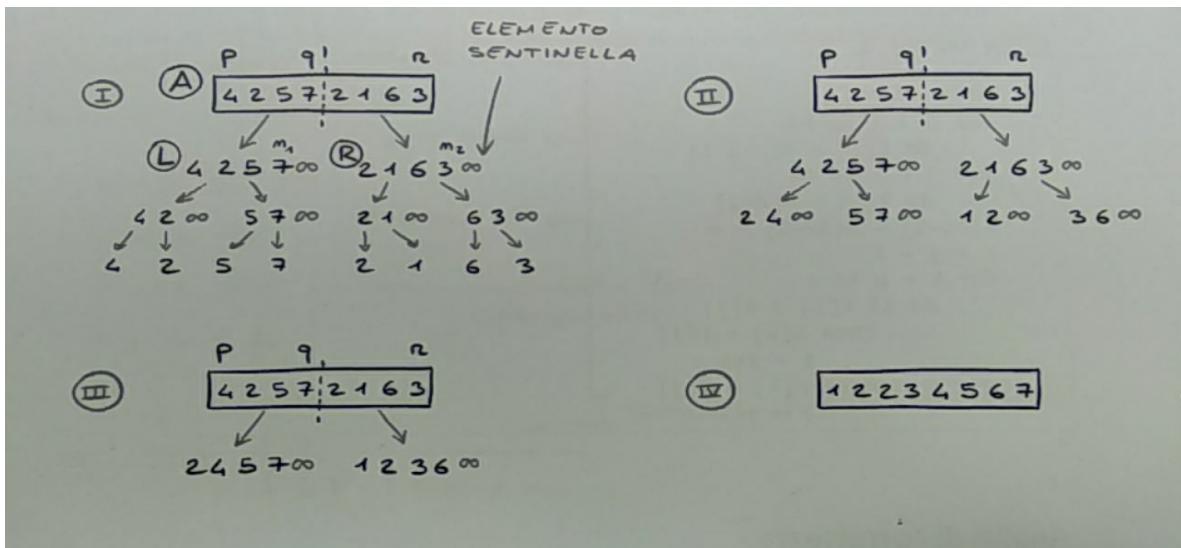
L'approccio usato per l'ordinamento del merge sort è il **divide et impera**, in particolare questo algoritmo divide la sequenza da ordinare in 2 sotto sequenze, che a loro volta vengono suddivise ricorsivamente fino ad ottenere sottosequenze di lunghezza 1. A questo punto le sottosequenze vengono combinate (merge) in un'unica sequenza ordinata, fino ad ottenere la sequenza principale ordinata.

L'algoritmo si basa su due funzioni:

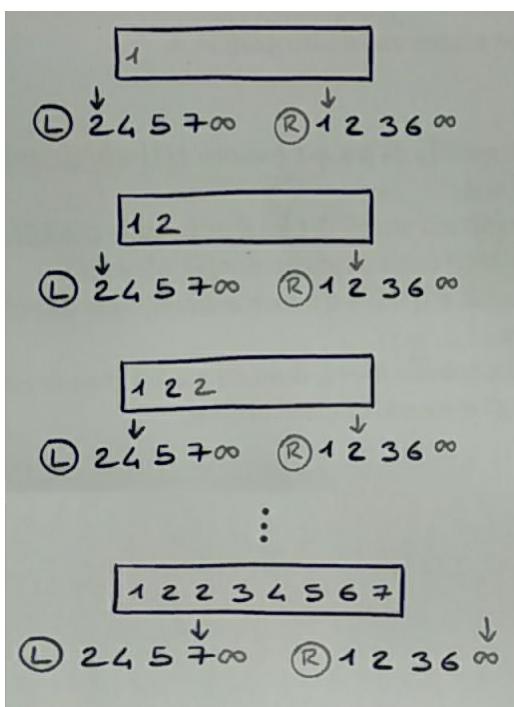
- **Merge**: Combina due sottosequenze ordinate in una sequenza ordinata.
- **Merge-Sort**: Divide la sequenza in 2 sottosequenze, le ordina (ricorsivamente) e le combina in un'unica sequenza ordinata.

## Lezione 3 26/09/19

Poiché sfrutta vettori esterni in quantità non costante, **non ordina sul posto**.



L'elemento sentinella è usato durante il confronto delle sottosequenze per avere un valore sicuramente maggiore degli altri. Vediamo più nello specifico come avviene l'operazione di Merge, prima di tutto gli elementi vengono confrontati due a due. Il valore più piccolo è scritto nel vettore originale e il puntatore è spostato verso destra. Considero quindi sempre la loro testa durante la comparazione, sposto poi l'indice di quello minore di  $j+1$  mentre l'altro rimane invariato e ripeto il ciclo. Il numero di iterazioni totali sarà  $i+j$ , indici dei due vettori.



durante la comparazione, sposto poi l'indice di quello minore di  $j+1$  mentre l'altro rimane invariato e ripeto il ciclo. Il numero di iterazioni totali sarà  $i+j$ , indici dei due vettori.

#### NOTA.

- Nell'algoritmo non ci preoccupiamo di allocare i vettori L left e R right. Ipotiziamo esistere già per questioni di velocità e scrittura dello pseudocodice.
- p, q ed r sono gli indici del vettore.
- n1 e n2 sono i due vettori che contengono le metà, con indici i e j.
- Nel primo for copiamo la prima metà degli elementi in n1 e la seconda in n2
- La posizione [p + i - 1] viene ricopiato in posizione 1 quando i è 1.

La complessità totale è  $\Theta(n)$ .

## Analisi di correttezza e della complessità

Verifichiamo la correttezza, ovvero che l'algoritmo risolva il nostro problema

**L'invariante del merge sort** risulta essere: All'inizio di ciascuna iterazione dell'ultimo ciclo for, la sottosequenza di A da  $p$  a  $k-1$  contiene i  $k-p$  elementi più piccoli, ordinati, di L ed R. L[i] ed R[j] sono i più piccoli elementi di L e R a non essere ancora ricopiatati in A.

#### Dimostriamo la verità dell'invariente e del passo induttivo.

- Prima della prima interazione  $k = p$ , quindi la sottosequenza da  $p$  a  $k-1$  è vuota. L[1] e R[1] sono i più piccoli elementi a non essere ancora ricopiatati in A.

```

Merge (A, p, q, r)
n1 ← q-p+1
n2 ← r-q
// create L[1..n1+1] and R[1..n2+1]
for i ← 1 to n1
    do L[i] ← A[p+i-1]
for j ← 1 to n2
    do R[j] ← A[q+j]
L[n1+1] ← R[n2+1] ← ∞
i ← j ← 1
for k ← p to r
    do if L[i] ≤ R[j]
        then A[k] ← L[i]
        i ← i+1
    else A[k] ← R[j]
        j ← j+1
    
```

T. Costante

$m = m_1 + m_2$

T. Cost.

$m$

Avremmo potuto usare l'insertion sort per ordinare i due vettori, ma come abbiamo visto nel caso peggiore il costo sarebbe stato quadratico, mentre con il merge avremo sempre tempi lineari.

Vediamo ora la procedura del **Merge-Sort** che si occupa dell'ordinamento dei sottovettori. Merge-sort è ricorsiva e serve a suddividere un vettore in due parti, indichiamo i due estremi con **p** ed **r** mentre **A** è il vettore. La procedura viene applicata fino a che il vettore non contiene l'elemento singolo.

```

Merge-Sort (A, p, r)
if p < r
    then q ← ⌊(p+r)/2⌋
    Merge-Sort (A, p, q)
    Merge-Sort (A, q+1, r)
    Merge (A, p, q, r)
    
```

L'algoritmo controlla prima se il vettore possiede due elementi, se non è verificata la condizione come detto precedentemente, si interrompe.

Procediamo adesso con l'analisi della complessità del Merge-Sort, ovvero il tempo di esecuzione dell'algoritmo .

Per gli approcci **divide et impera** consideriamo delle caratteristiche comuni:

- **$\Theta(1)$**  il tempo per risolvere il caso "banale".
- **D(n)** il tempo per suddividere un problema.
- **C(n)** il tempo per ricombinare le soluzioni dei sottoproblemi.
- **a** il numero di sottoproblemi in cui si divide un problema.
- **1/b** il fattore di riduzione della dimensione di un problema.

- Dato che per ogni interazione **A** ha da **p** a **k-1** elementi più piccoli di **L** ed **R** ordinati, e che **L[i]** e **R[j]** ci sono i più piccoli a non essere ricoperti, se **L[i] ≤ R[j]** viene ricoperto in **A[k]**, allora **A** è ordinato da **p** a **k** con gli elementi più piccoli di **L** ed **R**, e **L[i+1]** è il nuovo elemento più piccolo di **L**, quindi l'invariante rimane vero.

- L'algoritmo è quindi corretto, infatti il ciclo termina quando **k = r+1**, dove da **p** a **k+1** ci sono tutti gli elementi più piccoli di **L** e **R** ordinati.

Per il tempo di esecuzione ci si può rifare alle scritte segnate sull'algoritmo di Merge. Le prime due istruzioni sono delle assegnazioni e vengono fatte una sola volta, per questo il tempo è costante ed indipendente da **n**. I due cicli vengono ripetuti rispettivamente **n<sub>1</sub>** e **n<sub>2</sub>** volte, quindi complessivamente le sommiamo. Terzo blocco sono di nuovo delle assegnazioni, mentre nell'ultimo ciclo ci sono i confronti, costanti, di tutti gli elementi.

Il tempo di esecuzione di un algoritmo d.e.t basato su un problema di dimensione  $n$ , può essere scritto

$$\text{come: } T(n) = \begin{cases} \Theta(1) \text{ caso banale} \\ aT\left(\frac{n}{b}\right) + D(n) + C(n) \text{ tempo per dividere il problema} \end{cases}$$

La ricorsività dell'algoritmo si nota dalla presenza di  $T(n)$  all'interno della funzione stessa. Il caso banale è normalmente risolto in un tempo costante, infatti per costruzione suddividiamo i problemi che non siamo in grado di risolvere immediatamente in sottoproblemi che possono essere risolti in tempo costante.

Nel caso specifico del Merge-Sort sarà, assumendo  $n$  potenza di due:

$$\Theta(1) = c.$$

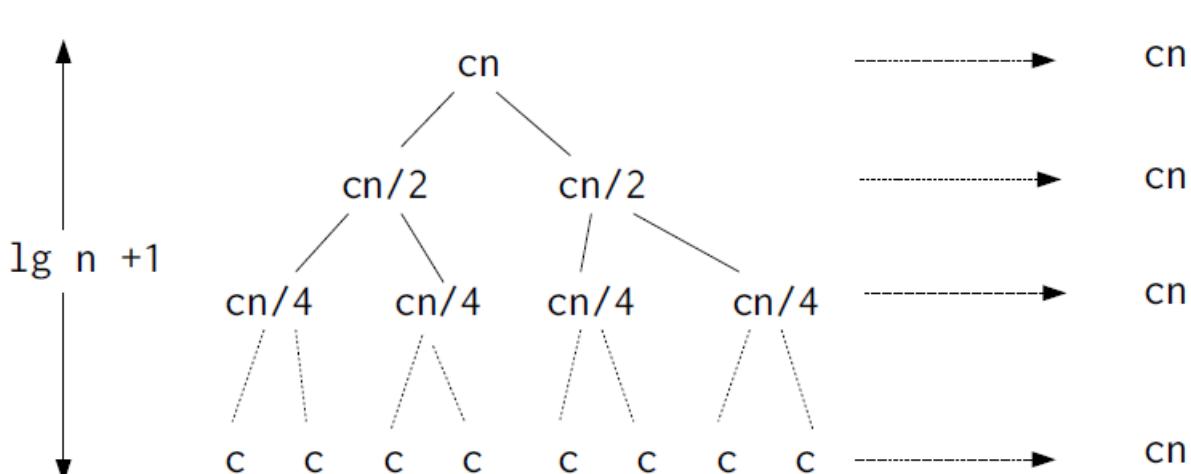
$$D(n) = \Theta(1) = c.$$

$C(n) = \Theta(n)$  calcolato dallo pseudocodice.

$$a = b = 2.$$

Avremo che  $T(n) = \begin{cases} c & \text{se } n = 1 \\ 2T\left(\frac{n}{2}\right) + C + Cn & \text{altrimenti} \end{cases}$  il  $C$  singolo nel secondo ramo è approssimabile a 0 per via di  $Cn$  che lo domina.

l'evoluzione del tempo è visionabile anche attraverso un albero che ha come caratteristica che i nodi sommati restituiscono sempre  $Cn$ . Ci resta da **determinare quanti livelli** ci sono nell'albero perché i suoi



livelli, moltiplicati per  $Cn$  ci risolve la ricorrenza indicata. L'albero è **un albero binario completo**, ovvero tutti i nodi hanno due figli tranne le foglie. La radice dell'albero corrisponde al termine noto, ossia tutto ciò che non è incluso nella parte ricorsiva. **Attenzione, n è il numero delle foglie dell'albero** in questo calcolo e non il numero degli elementi all'interno del vettore.

Il costo totale di merge-sort è quindi :

$$cn + cn + cn + \dots + cn = (1 + \log_2 n)cn = cn + cn \log_2 n = \theta(n \log_2 n)$$

Si può quindi dire che è leggermente peggiore del caso migliore dell'insertion sort, ma è di gran lunga migliore del caso peggiore di quest'ultimo.

## Lezione 4 02/10/19

# Crescita delle funzioni

Come abbiamo accennato in precedenza, valuteremo le prestazioni degli algoritmi osservando il tempo di esecuzione secondo un particolare modello: singolo processore e ad accesso casuale con memorie senza gerarchie(cache). L'efficienza di un algoritmo è calcolata mediante il suo tempo di esecuzione.

Studiare **l'efficienza asintotica** vuol dire considerare una grande mole di dati in ingresso al fine di rendere rilevante solo **l'ordine di crescita** del tempo di esecuzione.

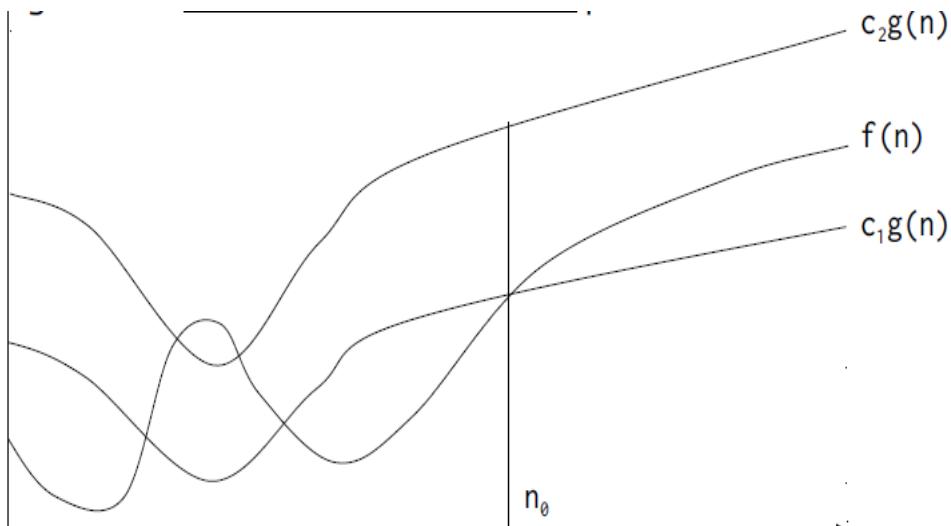
Una prima classe di funzioni che si introduce è la **notazione theta ( $\Theta$ )**.

## Notazione $\Theta$

Data la funzione  $g(n)$ ,  $\Theta(g(n))$  denota l'insieme di funzioni tali che = {  $f(n) : \exists c_1, c_2 \text{ e } n_0 \text{ costanti positive : } 0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n) \forall n > n_0$  }.

$g(n)$  è un **limite asintotico stretto** per  $f(n)$

Si usa impropriamente la scrittura  $f(n) = \Theta(g(n))$  anziché  $f(n) \in \Theta(g(n))$ .



La seguente notazione indica che la funzione  $f$  è limitata superiormente ed inferiormente da una funzione. Es:

$\Theta(n)$  limitato da funzioni lineari.

$\Theta(n^2)$  limitato da funzioni quadratiche.

Etc.

Tutti i termini di ordine inferiori e le costanti vengono ignorate.

Valutiamo un esempio più specifico :  $\frac{1}{2}n^2 - 3n = \Theta(n^2)$  per verificarlo bisogna trovare  $c_1$ ,  $c_2$ , e  $n_0$

tali che  $c_1 n^2 \leq \frac{1}{2}n^2 - 3n \leq c_2 n^2 \forall n > n_0$ .

Dividiamo tutto per  $n^2$  e otteniamo la seguente diseguaglianza :  $c_1 \leq \frac{1}{2} - \frac{3}{n} \leq c_2 \forall n > n_0$

La diseguaglianza di destra è valida per  $n \geq 1$  scegliendo  $c_2 \geq \frac{1}{2}$ , la diseguaglianza di sinistra è verificata

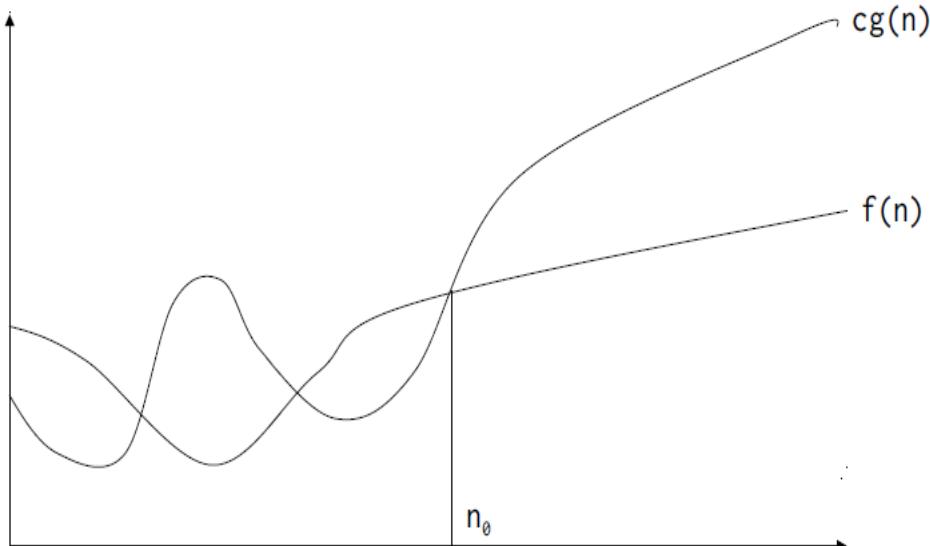
ma avremo che  $c_1$  risulterà un valore negativo andando quindi a non dispettare la sua definizione. Vediamo quando la funzione vale zero, la funzione vale 0 quando  $n = 6$  e la costante  $c_1$  ancora non rispetta la

definizione. Vediamo allora  $n \geq 7$ , facendo così  $c_1 = \frac{1}{14}$  e quindi tutte le definizioni vengono rispettate.

## Notazione O

Data la funzione  $g(n)$ ,  $O(g(n))$  denota l'insieme di funzioni:  $O(g(n)) = \{ f(n) : \exists c \text{ e } n_0 \text{ costanti positive : } 0 \leq f(n) \leq cg(n) \quad \forall n > n_0 \}$

$g(n)$  è il **limite superiore asintotico** per  $f(n)$ .



Esistono alcune relazioni tra la notazione O e la Θ:

- $f(n) = \Theta(g(n)) \Rightarrow f(n) = O(g(n))$
- $\Theta(g(n)) \subseteq O(g(n))$
- $an+b \neq \Theta(n^2)$  ma  $an+b = O(n^2)$  il **non uguale** significa **non appartenenza**.

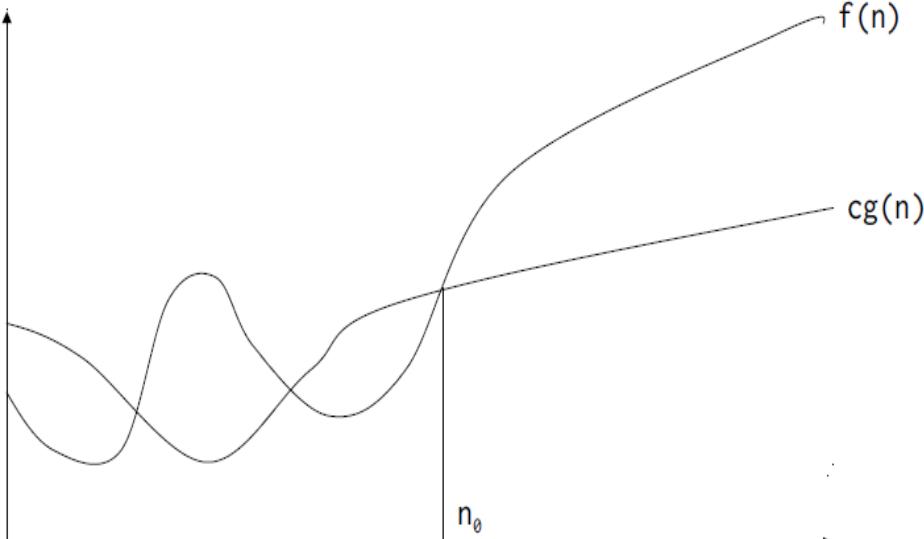
Il tempo di esecuzione di un algoritmo nel caso in cui il tempo peggiore è limitato superiormente da  $g(n)$ , per qualsiasi ingresso il tempo di esecuzione è sempre  $O(g(n))$ . Se il tempo di esecuzione peggiore invece è  $\Theta(g(n))$ , non è detto che il tempo di esecuzione per qualunque ingresso sia  $\Theta(g(n))$

Esempio: Insertion sort ha come caso peggiore  $\Theta(n^2)$ , sequenza ordinata  $\Theta(n)$ .

## Notazione Ω (omega)

Data la funzione  $g(n)$ ,  $\Omega(g(n))$  denota l'insieme di funzioni:  $\Omega(g(n)) = \{ f(n) : \exists c \text{ e } n_0 \text{ costanti positive : } 0 \leq cg(n) \leq f(n) \quad \forall n > n_0 \}$

$g(n)$  è un **limite inferiore asintotico** per  $f(n)$



Alcune proprietà della notazione Ω

- $f(n) = \Theta(g(n)) \Leftrightarrow f(n) = O(g(n)) \text{ e } f(n) = \Omega(g(n))$

Se il limite di esecuzione di un algoritmo nel caso migliore è  $\Omega(g(n))$ , il tempo di esecuzione per qualunque ingresso è sempre  $\Omega(g(n))$ .

## Proprietà delle notazioni asintotiche

Quando una notazione asintotica appare in una equazione va interpretata come una funzione ignota che non ci interessa specificare.

$2n^2 + 3n + 1 = 2n^2 + \Theta(n)$  equivale a  $2n^2 + 3n + 1 = 2n^2 + f(n)$ , dove  $f(n)$  è una qualunque funzione nell'insieme  $\Theta(n)$ . Valgono le seguenti proprietà:

1. Transitiva (valida per  $O$ ,  $\Theta$ ,  $\Omega$ )
  - a.  $f(n) = O(g(n))$  e  $g(n) = O(h(n)) \Rightarrow f(n) = O(h(n))$
2. Riflessiva (valida per  $O$ ,  $\Theta$ ,  $\Omega$ )
  - a.  $f(n) = O(f(n))$
3. Simmetrica
  - a.  $f(n) = \Theta(g(n)) \Leftrightarrow g(n) = \Theta(f(n))$
4. Anti-simmetrica
  - a.  $f(n) = O(g(n)) \Leftrightarrow g(n) = \Omega(f(n))$

Ricorrenze.

Una **ricorrenza** è un'equazione o disequazione che descrive una funzione in termini di sé stessa su ingressi di dimensione inferiore. Il **tempo di esecuzione** di algoritmi ricorsivi è, di solito, espresso mediante una ricorrenza. Un esempio già visto è quello degli algoritmi divide et impera:

- **$\Theta(1)$**  il tempo per risolvere il caso “banale”.
- **$D(n)$**  il tempo per suddividere un problema.
- **$C(n)$**  il tempo per ricombinare le soluzioni dei sottoproblemi.
- **$a$**  il numero di sottoproblemi in cui si divide un problema.
- **$1/b$**  il fattore di riduzione della dimensione di un problema.

$$T(n) = \begin{cases} \Theta(1) & \text{se } n = 1 \\ aT\left(\frac{n}{b}\right) + D(n) + C(n) & \text{altrimenti} \end{cases}$$

La **soluzione** di una ricorrenza è la classe di appartenenza della funzione  $T(n)$  in termini della sua complessità asintotica. Esistono diversi metodi per trovare la soluzione di una ricorrenza.

## Metodo di sostituzione

Non è un vero proprio metodo per risolvere una ricorrenza, ma semplicemente verifica che una soluzione candidata è valida. Questo perché non ci fornisce ne una soluzione, ne una soluzione ipotetica. Questo metodo consta di due passi:

- Ipotizzare una soluzione.
- Verificarla mediante il **principio di induzione**.

L'utilizzo di questo metodo può essere utile per derivare sia limiti superiori che inferiori su una ricorrenza.

Es:

$$T(n) = \begin{cases} T(1) = 1 \\ T(n) = 2T\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + n \end{cases}$$

Quello che vogliamo provare è che la soluzione  $T(n)$  è del tipo  $O(n \log n)$ . Questa soluzione può essere estratta a caso dalla funzione, cosa molto improbabile, oppure con un altro metodo di ricorrenza il quale però non ci garantisce la correttezza e quindi decidiamo di verificarla con questo metodo.

Vogliamo dimostrare che:  $\exists c : T(n) \leq c n \log n$  per un  $n$  sufficientemente grande.

**Passo induttivo** -> Assumiamo per vero che  $\left\lfloor \frac{n}{2} \right\rfloor$ , dimostriamo per  $n$

$$T(n) = 2T\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + n \leq 2c\left\lfloor \frac{n}{2} \right\rfloor \log_2\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + n \leq cn \log_2\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + n = (\text{il logaritmo è crescente e mantiene il verso della disegualanza})$$

$$= cn \log_2 n - cn \log_2 2 + n = cn \log_2 n - cn + n \leq cn \log_2 n \text{ Se } c \geq 1$$

**Passo base** -> Per  $n = 1$   $T(1) = 0$ , il tempo di esecuzione **non può essere**  $\leq 0$

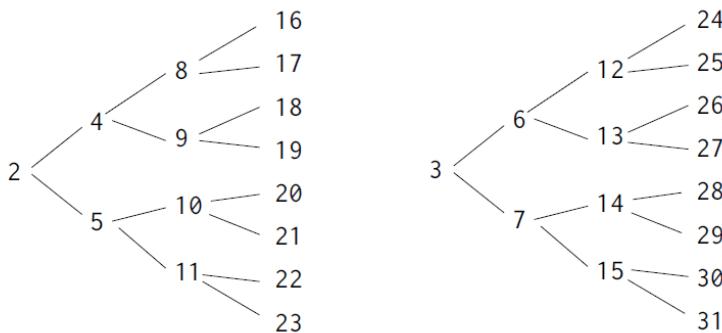
Dunque, proviamo che  $T(2)$  e  $T(3)$  siano minori di  $n \log n$ .

$$T(2) = 2 T(1) + 2 = 4 \leq 2 c \log 2 = 2c \rightarrow \text{Se } c \geq 2$$

$$T(3) = 2 T(1) + 3 = 5 \leq 3 c \log 3 = 3c \log 3 \rightarrow \text{Se } c \geq 1$$

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

I casi presi sono due in modo tale da coprire tutti i possibili casi.



## Lezione 5 03/10/19

### Metodo dell'albero di ricorrenza

Il metodo dell'albero di ricorrenza **consiste nel generare un albero la cui radice è il termine noto di  $T(n)$**  e i cui nodi rappresentano la chiamata ricorsiva su un sotto problema di dimensione inferiore. Questo metodo

può essere usato per generare delle buone soluzioni di tentativo, quest'ultime sono da verificare con il metodo di sostituzione se sono state fatte ipotesi semplificative per derivare le soluzioni.

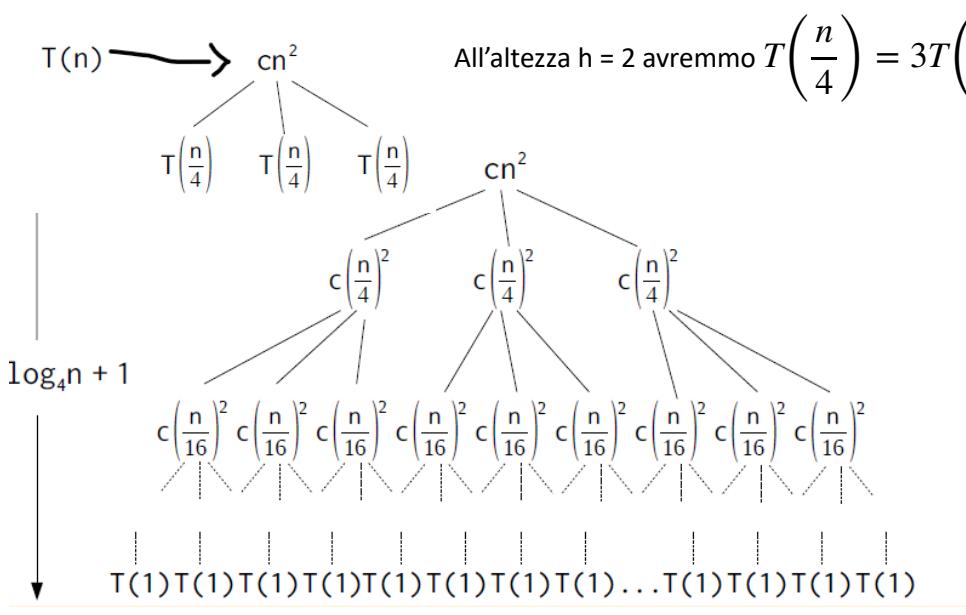
Ogni nodo riporta il costo per risolvere il suo sotto problema, sommando il costo di ogni livello dell'albero si ottiene una stima della complessità asintotica di  $T(N)$ .

Es:

Consideriamo  $T(n)$  dell'algoritmo di Merge sort, con alcune semplificazioni,

$$T(n) = 3T\left(\left\lfloor \frac{n}{4} \right\rfloor\right) + \Theta(n^2) \text{ approssimiamo considerando } \frac{n}{4} \text{ stesso e non approssimato per difetto,}$$

e invece di una generica funzione di  $\Theta(n^2)$  consideriamo  $cn^2$  con l'esplicitazione di  $c$  costante. Ed infine come ulteriore semplificazione consideriamo  $n$  potenza di 4, che ci è necessario per ottenere un albero completo.



$$\text{All'altezza } h = 2 \text{ avremmo } T\left(\frac{n}{4}\right) = 3T\left(\frac{n}{16}\right) + c\frac{n^2}{16}$$

Partiamo dalla radice dell'albero che ha complessità  $T(n)$ , il **livello della radice è zero**. Per questo albero è vero che la somma di tutti i nodi ci dà come complessità  $T(n)$ , i tre figli generati dalla radice hanno tutti costo pari a  $\frac{n}{4}$ . Genericamente la

dimensione del sottoproblema diventa  $\frac{n}{4^i}$ , continuando con questa suddivisione arriveremo ) al livello

$\log_4 n + 1$  ad avere le foglie con tempo di esecuzione pari a  $T(1)$ . Il nostro obiettivo è quello di determinare il costo e il numero dei livelli dell'albero, dunque al livello  $i$  il **costo di un singolo nodo è**

$c \left(\frac{n}{4^i}\right)^2$  e ogni livello ha un numero di nodi pari a  $3^i$  per un totale del costo pari a  $3^i c \left(\frac{n}{4^i}\right)^2$ , quindi

ogni sottoproblema è 4 volte più piccolo del precedente. Scriviamo dunque la sommatoria per  $T(n)$  considerando tutti i livelli, dal livello zero all'ultimo:

$$T(n) = \sum_{i=0}^{\log_4 n - 1} 3^i c \left(\frac{n}{4^i}\right)^2 + 3^{\log_4 n} T(1) = \sum_{i=0}^{\log_4 n - 1} \left(\frac{3}{16}\right)^i c n^2 + \theta(n^{\log_4 3}) \text{ applicando il}$$

cambiamento di base di logaritmi e andando ora a **maggiorare con una somma infinita**

$$< \sum_{i=0}^{\infty} \left(\frac{3}{16}\right)^i c n^2 + \theta(n^{\log_4 3}) = \frac{1}{1 - (\frac{3}{16})} c n^2 + \theta(n^{\log_4 3}) \rightarrow T(n) = O(n^2) \text{ non ho usato theta}$$

perché abbiamo maggiorato con la somma infinita.

Con questo esempio abbiamo visto che la complessità asintotica dovrebbe essere  $O(n^2)$ . Bisogna ora però verificarlo applicando il metodo di sostituzione.

Dim.

$T(n) = O(n^2) \leftrightarrow \exists d : T(n) \leq dn^2$  per ogni  $n \geq n_0$  troviamo un  $c$  costante positiva

- Passo induttivo ( Assumiamo vero per  $\left\lfloor \frac{n}{4} \right\rfloor$ , dimostriamo per  $n$ )

$T(n) = 3T\left(\left\lfloor \frac{n}{4} \right\rfloor\right) + \theta(n^2) \leq 3T\left(\left\lfloor \frac{n}{4} \right\rfloor\right) + cn^2$  ora applicheremo l'ipotesi induttiva e quindi

$$\leq 3d \left\lfloor \frac{n}{4} \right\rfloor^2 + cn^2 \leq \frac{3}{16}dn^2 + cn^2 \text{ se } d \geq \left(\frac{16}{13}\right)c$$

- Passo base

$$T(1) \leq d \cdot 1^2 = d \rightarrow d \geq T(1) \text{ e quindi } T(n) \rightarrow O(n^2).$$

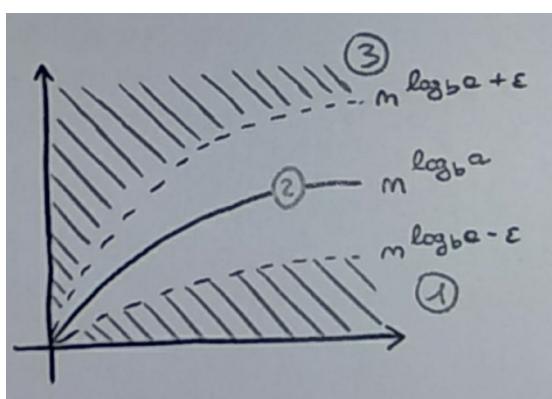
In realtà vale anche  $T(n) = \Omega(n^2)$  perché il theta di  $n$  quadro è limitata sia superiormente che inferiormente dal termine quadrato, avremmo quindi potuto considerare una minorazione rispetto ad una maggiorazione. Quindi  $T(n) = \Theta(n^2)$ .

## Metodo dell'esperto

Il metodo dell'esperto si basa sul **teorema dell'esperto**, il quale afferma che:  $T(n) = aT\left(\frac{n}{b}\right) + f(n)$  e siano  $a \geq 1$  e  $b > 1$  costanti,  $f(n)$  una funzione e  $\left(\frac{n}{b}\right)$  può essere considerato sia approssimato per difetto che per eccesso.

Allora possono applicarsi tre situazioni, dove andiamo a confrontare l'ordine di crescita di due termini:

1.  $f(n) = O(n^{\log_b a - \epsilon})$  per un dato  $\epsilon > 0 \rightarrow T(n) = \theta(n^{\log_b a})$ .
2.  $f(n) = \theta(n^{\log_b a}) \rightarrow T(n) = \theta(n^{\log_b a} \lg n)$ .
3.  $f(n) = O(n^{\log_b a + \epsilon})$  per un dato  $\epsilon > 0$  e  $af\left(\frac{n}{b}\right) \leq cf(n)$  con  $c < 1 \rightarrow T(n) = \theta(f(n))$ .



Si nota che in ciascuno dei 3 casi "vince" la funzione **più grande tra  $f(n)$  e  $n^{\log_b a}$** , c'è poi un gap tra il 1° ed in 2° caso e tra il 2° ed il 3° poiché i casi in cui  $f(n)$  è più piccola(o grande) di  $n^{\log_b a}$ , ma non è polinomiale, non possiamo usare il teorema dell'esperto per risolvere il problema.

Es:

- $T(n) = 9T\left(\frac{n}{3}\right) + n$  abbiamo dunque  $a=9$ ,  $b=3$  e  $f(n) = n$ ,  $n^{\log_3 9} = n^2$

Rientriamo nelle ipotesi del teorema, nello specifico nel primo caso e quindi la soluzione è un theta della funzione che cresce più velocemente. Infatti, vediamo che:

$$n = f(n) = O(n^{\log_3 9 - \epsilon}) \text{ con } \epsilon = 1 \rightarrow T(n) = \theta(n^2)$$

- $T(n) = T\left(\frac{2n}{3}\right) + 1$  abbiamo dunque  $a=1$ ,  $b=\frac{3}{2}$  e  $f(n) = 1$ ,  $n^{\log_{\frac{3}{2}} 1} = n^0 = 1$

Si applica il secondo caso, infatti abbiamo  $1 = f(n) = \theta(n^{\log_b a}) = \theta(1) \rightarrow T(n) = \theta(\lg n)$

- $T(n) = 3T\left(\frac{n}{4}\right) + n \lg n$  abbiamo dunque  $a=3$ ,  $b=4$  e  $f(n) = n \lg n$ ,  $n^{\log_4 3} = n^{0.8}$

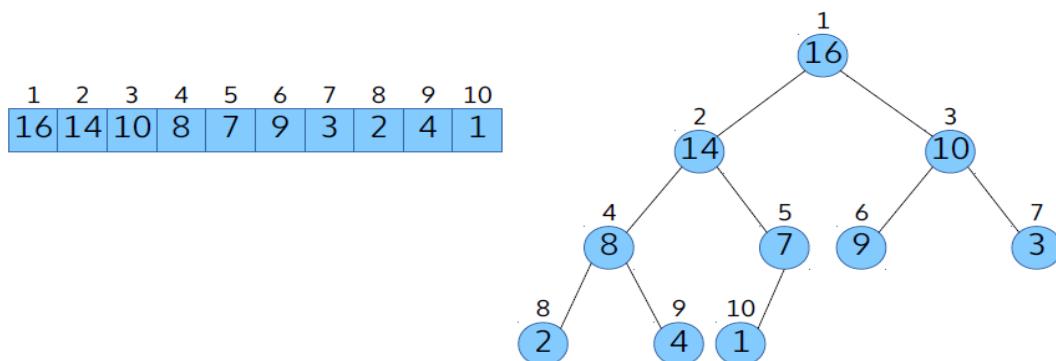
Stiamo dunque confrontando  $f(n)$  che cresce più velocemente di  $n^{0.8}$ .

Quindi  $n \lg n = f(n) = \Omega(n^{\log_4 3 + \epsilon})$  con  $\epsilon \simeq 0.2(\log_4 3)$   $\rightarrow T(n) = \theta(n \lg n)$

- Questo è il caso particolare che rientra tra i tre casi.  $T(n) = 2T\left(\frac{n}{2}\right) + n \lg n$  abbiamo  $a=2$ ,  $b=2$  e  $f(n) = n \lg n$ ,  $n^{\log_2 2} = n$ . Quindi  $f(n)$  non è polinomialmente più grande di  $n$  e  $\frac{f(n)}{n} = \lg n$  è asintoticamente più piccolo di  $n^\epsilon$ . Si sceglie quindi un altro metodo per analizzare il seguente algoritmo.

## Heapsort

L'**heapsort** è un algoritmo di ordinamento basato sulla struttura dati **heap**, ossia un array che soddisfa determinate proprietà ovvero un array che utilizza locazioni consecutive di memoria, **ordina sul posto e ha tempo di esecuzione pari a  $O(n \lg n)$** . Un heap array può essere visualizzato come un albero binario quasi completo, in cui solo l'ultimo livello può non essere completo.



Disponendo in base a come leggiamo dal vettore abbiamo: 1° elemento diventa radice, poi riempiendo da sinistra verso destra 2° e 3° nodo pari a 14 e 10, etc...

Per costruzione abbiamo quindi che dato un indice  $i$

- Padre  $\rightarrow \left\lfloor \frac{i}{2} \right\rfloor$
- Left  $\rightarrow 2i$
- Right  $\rightarrow 2i + 1$
- Heap-size[A]  $\leq \text{lenght}[A]$

Queste sono operazioni implementate in maniera molto efficiente, dal momento che corrispondono a dei semplici shift dei bit di una casella a destra o a sinistra. L'**altezza** di un nodo è il più semplice e lungo percorso (numero di archi) verso una foglia, l'altezza dell'albero corrisponde dunque all'altezza della radice.

L'heap utilizzato dall'algoritmo di heap-sort è detto **max-heap**, ossia un array il cui albero ha il nodo più piccolo di suo padre, per cui la radice ha il valore massimo, non vale il viceversa. Un max-heap soddisfa

quindi la seguente proprietà:  $A[i] \leq A \left[ \left\lfloor \frac{i}{2} \right\rfloor \right] \forall i$ . Mentre per **min-heap** la radice è il valore minimo e vale

$$A[i] \geq A \left[ \left\lfloor \frac{i}{2} \right\rfloor \right] \forall i.$$

L'array però non è ordinato in modo decrescente, ma vale la proprietà del max-heap appena descritta.

## Lezione 6 09/10/19 (solo un'ora di lezione, piccole gioie della vita)

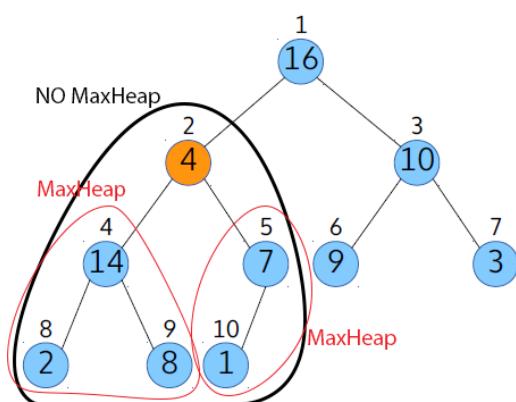
Utilizziamo il max-heap per realizzare l'algoritmo di **Max-heapify** o per realizzare code a priorità, che a differenza delle code FIFO, che estraggono in base alla priorità dell'elemento. Un esempio di uso di heap è l'algoritmo di Dijkstra che trova il percorso minimo del nodo sorgente al nodo d'arrivo in un grafo, il quale realizza un vettore contenente i nodi non ancora marcati contenente la distanza dalla sorgente.

### Max-Heapify

La funzione **Max-heapify** ha il compito di trasformare in max-heap un sottoalbero i cui figli (left e right) della radice sono entrambi già radici di max-heap.

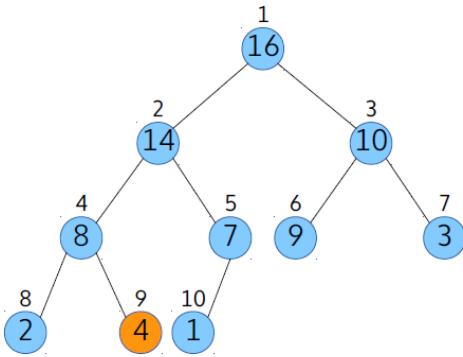
È importante notare che la max-heapify può essere applicata solo se i figli sono max-heap, condizione che sarà impostata da un'altra funzione che espliceremo dopo.

Es.



Il sottoalbero che ha radice nel nodo 2 non è un max-heap, ma entrambi i suoi figli sono dei max-heap. Quindi possiamo usare la max heapify per trasformarlo in un max-heap.

Confrontiamo dunque la radice con i suoi figli e scambiamo il suo posto con quello maggiore. Questo processo viene effettuato ricorsivamente finché la radice non diventa maggiore di entrambi i figli o finché non raggiunge le foglie.



## Pseudocodice

Supponiamo che il primo elemento sia il massimo e poi con un ciclo confrontiamo il valore per vedere se effettivamente questo sia il massimo assoluto. Partendo dall'indice iniziale i salviamo il valore sia nella variabile Left che Right, e salviamo l'indice più che il valore. Prima di confrontare il valore di sinistra(destra) devo controllare che esista e che abbia un figlio(stessa cosa per right), quindi controllo con l'attributo heap-size; se i fosse maggiore della dimensione, vorrebbe dire che calcolando il valore del figlio di sinistra siamo usciti fuori dall'array. La parola **exchange** indica un generico algoritmo per scambio di valori, nello pseudocodice non ci interessa il modo implementativo. La dimensione del sottoproblema è pari poi al numero dei nodi del sottoalbero, quindi non può essere fissa e dipende dalle varie dimensioni ma

potremmo dire che è pari a  $O\left(\frac{n}{2}\right)$  mentre per il resto è **costante** quindi indipendente dal problema.

```
Max-Heapify (A, i)
1 ← Left(i)
r ← Right(i)
largest ← i
if 1≤heap-size[A] and A[1]>A[i]
    then largest ← 1
if r≤heap-size[A] and A[r]>A[largest]
    then largest ← r
if largest≠i
    then exchange A[i]↔A[largest]
Max-Heapify (A, largest)
```

Valutando meglio la dimensione del sottoproblema consideriamo che:

- n è la dimensione del problema originario
- m è la dimensione del sottoalbero con più elementi
- h è l'altezza del nodo oggetto del problema originario

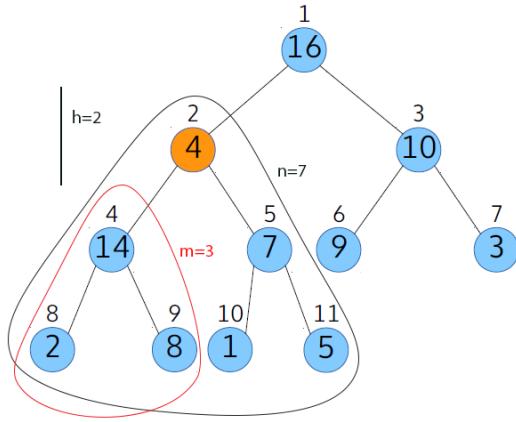
Possiamo dire che si presentano due casi:

Caso 1: l'ultimo livello del problema è pieno.

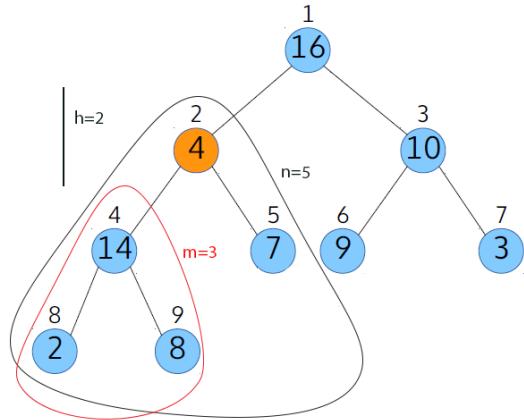
$$n = \sum_{i=0}^h 2^i = 2^{h+1} - 1$$

$$m = \sum_{i=0}^{h-1} 2^i = 2^h - 1 \rightarrow \frac{m}{n} = \frac{2^h - 1}{2^{h+1} - 1} = \frac{2^h - 1}{2 * 2^h - 1} \xrightarrow{h \rightarrow \infty} \frac{1}{2}$$

vale quindi  $O\left(\frac{n}{2}\right)$  e dunque il sottoproblema è due volte più piccolo del problema di partenza e non è il caso peggiore.



Caso 2: l'ultimo livello è pieno a metà.



$$n = \sum_{i=0}^{h-1} 2^i + \sum_{i=0}^{h-2} 2^i + 1 = (2^h - 1) + (2^{h-1} - 1) + 1 = 3 * 2^{h-1} - 1$$

La prima sommatoria riguarda il sottoalbero pieno, il secondo è il sottoalbero senza l'ultimo livello ed infine il +1 riguarda la radice.

$$m = 2^h - 1 \rightarrow \frac{m}{n} = \frac{2}{3} * \frac{2^{h-1} - 1}{2^{h-1} - 1} = \frac{2^h - 1}{2 * 2^{h-1} - 1} \xrightarrow{h \rightarrow \infty} \frac{2}{3}$$

vale quindi  $O\left(\frac{2}{3}n\right)$  e dunque la dimensione del sottoproblema ed è ridotto di  $\frac{2}{3}$  di quello di partenza e **risulta il caso peggiore.**

Abbiamo quindi come **tempo di esecuzione:**  $T(n) \leq T\left(\frac{2}{3}n\right) + \theta(1)$  che rientra nel 2° caso del

teorema dell'esperto ( $f(n) = \theta(1)$  e  $n^{(\log_b a)} = n^{\frac{\log_3 1}{2}} = 1$  e quindi  $T(n) = O(\lg n)$ )

**La complessità quindi della Max-Heapify è  $O(\lg n)$ , dove  $n$  è il numero di nodi del sottoalbero che stiamo trasformando in max-heap, non il numero totale dei nodi di tutto l'albero.** Inoltre, indicando con  $h$  l'altezza del sottoalbero al quale applichiamo il max-heap:  $h = \lfloor \lg n \rfloor \rightarrow T(n) = O(h)$

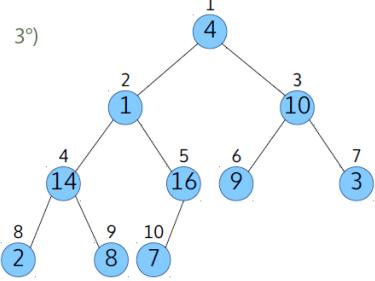
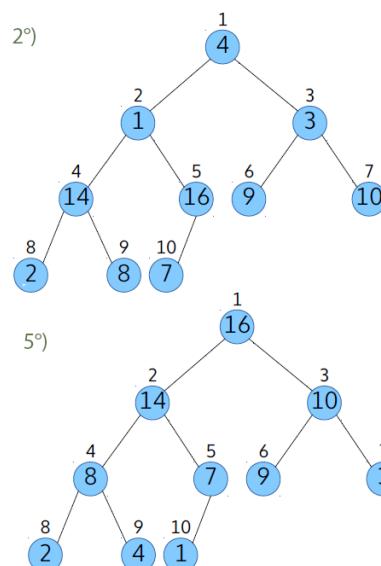
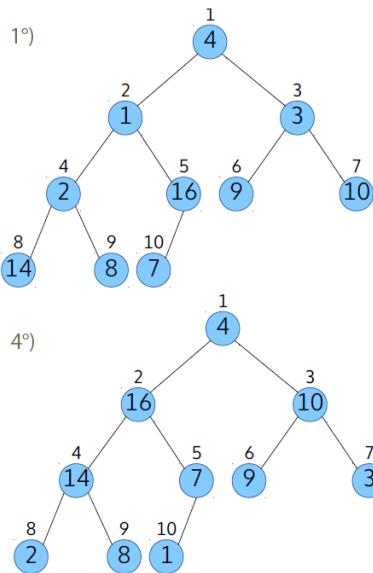
## Lezione 7 10/10/19

### HeapSort: Build-Max-Heap

La funzione Build-Max-Heap invoca la Max-Heapify dal basso verso l'alto, ad eccezione dei nodi foglia, che in un certo senso sono già dei max-heap di un solo elemento. Invocando la Max-Heapify dal basso si fa in modo da soddisfare le sue condizioni(cioè entrambi i figli devono già essere dei max-heap).

ES)

1	2	3	4	5	6	7	8	9	10
4	1	3	2	16	9	10	14	8	7



Dalla prima invocazione cambiamo di posto, all'interno dell'array il posto di 2 e 14, successivamente scambiamo 10 e 3 e così via fino al 4 che scala fino ad essere posizionato sulla 9° posizione.

Notiamo che, avendo saltato le foglie, il primo nodo su cui è invocata la Max-Heapify coincide con l'ultimo nodo non foglia. Il suo indice si può calcolare a partire dal numero di foglie:

Foglie =

$$n - (2^h - 1) + 2^{h-1} - \left\lceil \frac{n - (2^h - 1)}{2} \right\rceil = n - 2^{h-1} + 1 - \left\lceil \frac{n+1}{2} \right\rceil + \frac{2^h}{2} = n + 1 - \left\lceil \frac{n+1}{2} \right\rceil$$

Infatti se **n** è il numero totale di nodi, e  $2^h - 1$  nodi senza l'ultimo livello abbiamo che  $n - (2^h - 1)$  è il numero di foglie nell'ultimo livello. Mentre se consideriamo  $2^{h-1}$  abbiamo il numero di nodi del penultimo

livello,  $\left\lceil \frac{n - (2^h - 1)}{2} \right\rceil$  il numero di padri delle foglie dell'ultimo livello.

$$\text{Per calcolare l'indice dell'ultimo nodo non foglia} = n - \text{foglie} = \left\lceil \frac{n+1}{2} \right\rceil - 1 = \left\lceil \frac{n}{2} \right\rceil$$

Dunque il primo nodo su cui è invocata la Max-Heap + quello di indice  $\left\lceil \frac{n}{2} \right\rceil$ , dopodiché è infovata a ritroso su ogni nodo precedente.

**Nota:**  $n$  e  $h$  sono rispettivamente il numero di nodi e l'altezza di tutto l'albero, mentre nella Max-Heapify abbiamo indicato con  $n$  e  $h$  i nodi e l'altezza del sottoalbero.

## Pseudocodice

```
Build-Max-Heap (A)
heap-size[A] ← length[A]
for i ← ⌊length[A]/2⌋ down to 1 →  $O(n)$ 
    do Max-Heapify (A, i) →  $O(\lg n)$ 
```

## Analisi di correttezza

**Invariante** dell'Heap-Sort:

All'inizio di ciascuna interazione del ciclo *for*, ogni nodo  $i+1, i+2, \dots, n$  è la radice di un max-heap.

Dimostriamo che l'invariante è vera per la proprietà induttiva:

- Prima della prima interazione, tutti i nodi successivi a  $i = \left\lfloor \frac{n}{2} \right\rfloor$  sono foglie, i nodi  $i+1$  fino ad  $n$ , quindi max-heap.
- L'invariante si conserva attraverso le interazioni, infatti in ogni interazione i figli sono max-heap (per ipotesi) e applicando Max-Heapify, il nodo corrente diventa a sua volta un max-heap. Dunque nelle successive interazioni i figli saranno o le foglie o i nodi che sono stati trasformati in max-heap nelle iterazioni precedenti. Quindi si conserva la veridicità dell'invariante.
- L'algoritmo è corretto, il ciclo termina quando tutti i nodi sono radici di max-heap, cioè l'albero è un max-heap.

## Analisi di complessità

Possiamo dire che:

- Il ciclo *for* è eseguito un numero pari a  $\left\lfloor \frac{n}{2} \right\rfloor$  volte, quindi la complessità è  $O(n)$
- La Max-Heapify è  $O(\lg n)$
- Il totale sarà di :  $O(n \lg n)$

Vi è un problema però. Non c'è un limite stretto, nel senso che abbiamo supposto che la Max-Heapify abbia complessità  $O(\lg n)$  ogni volta che viene invocata. Questo accade perché abbiamo supposto che  $n$  che si trova all'interno del logaritmo sia il numero totale di nodi dell'albero, mentre sappiamo che dovrebbe indicare il numero di nodi del sottoalbero corrente.

Infatti, sappiamo che la Max-Heapify è chiamata dal basso verso l'alto quindi l'altezza del sottoalbero non è più fissa, ma parte da  $h = 1$  ed aumenta fino a diventare  $h = \lfloor \lg n \rfloor$ . Possiamo dunque scrivere:

$$T(n) = \sum_{h=0}^{\lfloor \lg n \rfloor} \left\lceil \frac{n}{2^{h+1}} \right\rceil O(h) = \text{quello tra gli arrotondamenti indica il numero di nodi di altezza } h.$$

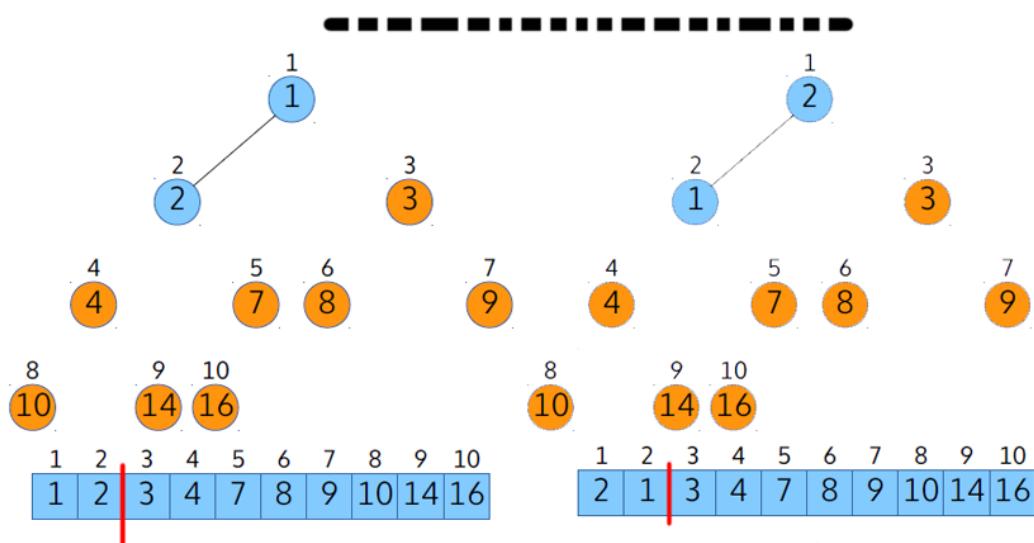
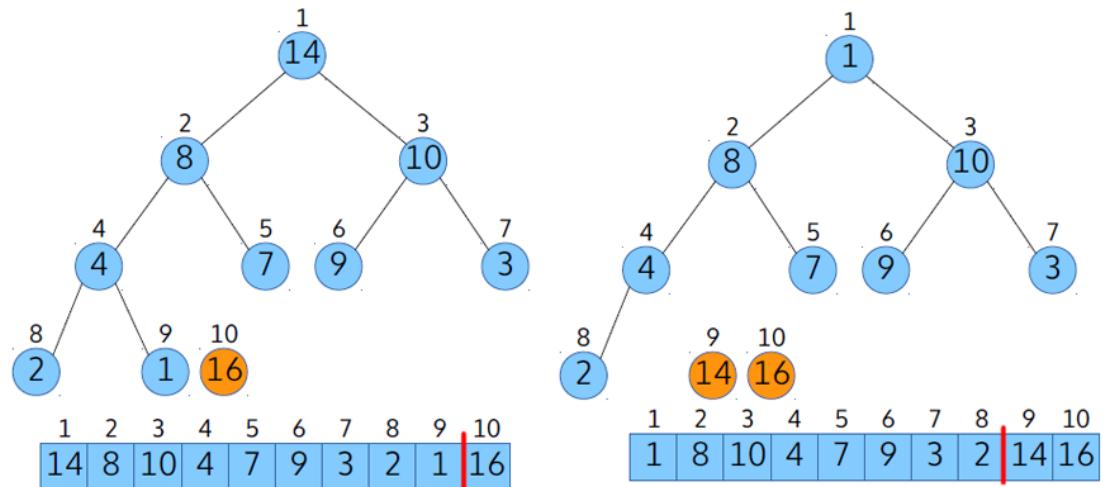
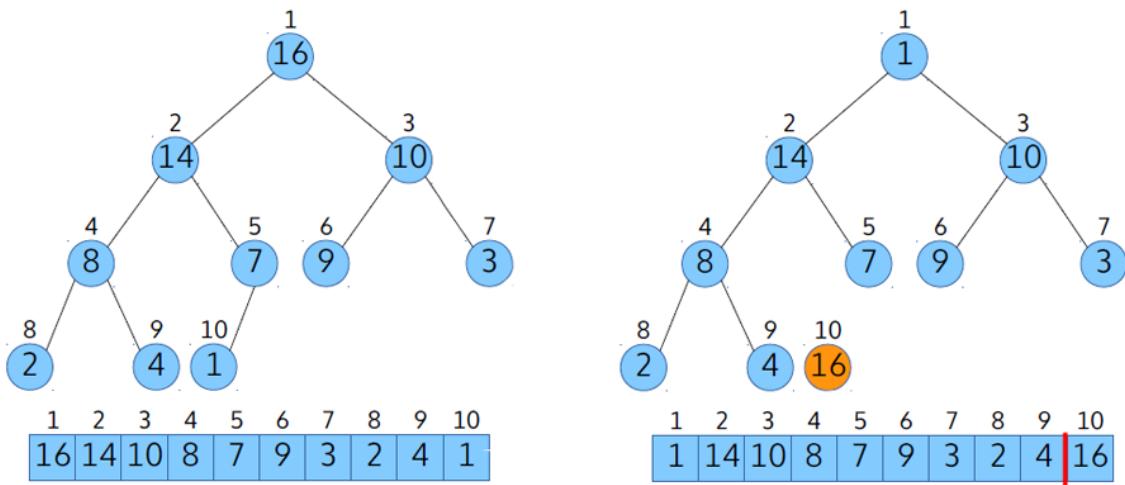
$$= O\left(\frac{n}{2} \sum_{h=0}^{\lfloor \lg n \rfloor} \frac{h}{2^h}\right) = O\left(\frac{n}{2} \sum_{h=0}^{\lfloor \lg n \rfloor} \frac{h}{2^h}\right) = O(2n) = O(n)$$

nell'ultima uguaglianza abbiamo semplificato il due perché i valori costanti nel calcolo asintotico si possono trascurare.

## HeapSort: Sort

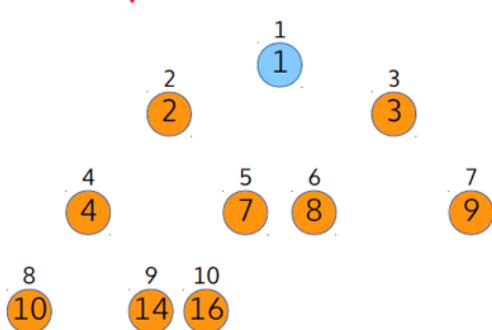
Ora che abbiamo reso un vettore un max-heap usando un tempo lineare, dobbiamo ordinarlo in modo crescente. Esistono tanti modi per farlo e noi sceglieremo questo:

- Scambio il primo elemento( la radice che è l'elemento più grande) con l'ultimo, in modo da avere il valore più grande alla fine.
- Decremento la dimensione dell'heap (come se eliminassimo l'ultimo elemento dal vettore)
- Invoco la Max-Heapify sulla radice per trovare il più grande elemento in cima
- Ripeto il ciclo.



Vettore ordinato

1	2	3	4	5	6	7	8	9	10
1	2	3	4	7	8	9	10	14	16



## Pseudocodice

### Heap-Sort (A)

```
Build-Max-Heap(A) → O(n)
for i ← length[A] down to 2 → O(n)
    do exchange A[1]↔A[i]
    heap-size[A] ← heap-size[A]-1
    Max-Heapify (A,1) → O(log n)
```

## Analisi di correttezza

**Il professore non lo definisce, ipotizziamo quindi una cosa del genere:** Tutti gli elementi da  $i+1$  ad  $n$ , sono gli  $n-i$  elementi più grandi ordinati, inoltre il primo elemento di  $A$  è il più grande tra gli elementi da 1 ad  $i$ .

## Analisi di complessità

Poiché la Build-Max-Heap è  $O(n)$ , e la Max-Heapify è  $O(\lg n)$  ed è chiamata  $O(n)$  volte, la complessità dell'HeapSort è  $O(n \lg n)$ .

$$T(n) = O(n) + O(n)*O(\lg n) = O(n \lg n)$$

## Lezione 8 16/10/19

### Code a priorità

Le code a priorità sono **strutture dati basate sull'heap**, vengono utilizzate per estrarre il valore a più alta o bassa priorità usando le chiavi. Si parla di struttura dati quando si parla di un particolare modo di organizzare gli elementi al fine di effettuare in modo efficiente un certo insieme di operazioni. Si considerano In particolare:

- Le code a massima priorità usano Max-Heap, dunque hanno nella radice l'elemento maggiore e si estraie l'elemento più importante.
- Le code a minima priorità usano Min-Heap, quindi hanno nella radice l'elemento minore e si estraie l'elemento a priorità più bassa.

Un algoritmo che utilizza le code a massima priorità è quello di scheduling dei processi all'interno del processore, mentre le code a priorità minima sono usate per l'algoritmo di Dijkstra.

Una cosa a massima priorità **supporta le seguenti operazioni:**

- Insert( A, key ) = Inserisce l'elemento key in A.
- Maximum(A) = Restituisce l'elemento più grande.
- Extract-Max(A) = Rimuove l'elemento più grande.
- Increase-Key(A,i,key) = L'elemento key incrementa di un valore i, ed è funzionale all'inserimento.

Se non usassimo la struttura basata sull'heap ma un'altra struttura avremmo che le operazioni di inserimento e di estrazione, avrebbero rispettivamente tempo di esecuzione pari a  $\theta(1)$  e  $O(n)$  perché non ordinati.

## Heap-Maximum

La Heap-Maximum semplicemente **restituisce il più grande elemento di A**, ottenendo così anche la chiave più grande, senza rimuoverlo.

```
Heap-Maximum (A)  
    return A[1]
```

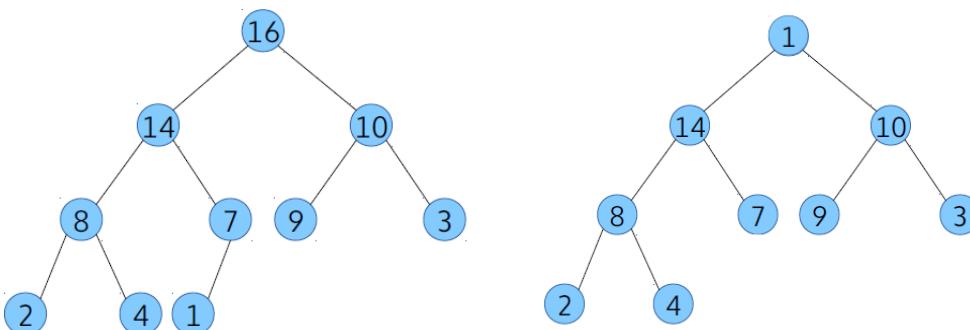
Con tempo di esecuzione pari a  $\theta(1)$ . Ricordiamo che essendo il vettore un Max-Heap, l'elemento in prima posizione è quello di valore massimo.

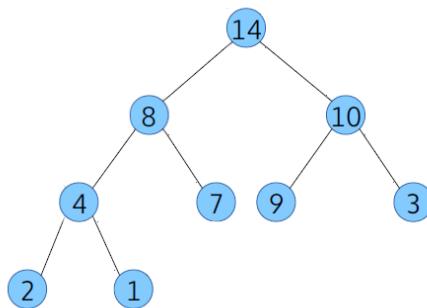
## Heap-Extract-Max

La Heap-Extract-Max restituisce l'elemento più grande(quello nella radice) e lo estrae riducendo la dimensione dell'heap. Prima di ridurre la dimensione, l'ultimo elemento è inserito nella radice, altrimenti andrebbe perso. Ovviamente occorre invocare poi una **Max-Heapify per ripristinare il max-heap**.

```
Heap-Extract-Max (A)  
if heap-size[A]<1  
    then error "heap empty"  
max ← A[1]  
A[1] ← A[heap-size[A]]  
heap-size[A] ← heap-size[A] - 1  
Max-Heapify (A,1)  
return max →  $O(\lg n)$ 
```

Il controllo iniziale serve per verificare che il vettore non sia nullo, altrimenti dovrebbe lanciare un'eccezione. Il tempo di esecuzione è  $O(\lg n)$  e invoco una sola volta Max-Heapify.





## Heap-Increase-Key

La Heap-Increase-Key sostituisce un elemento con una chiave key, con un nuovo elemento con una chiave **più grande**. Quando incrementiamo la chiave di un elemento però possiamo violare la proprietà dei figli minori o uguali al padre, **facendolo risalire fino a quando non incontra un elemento più grande di se stesso manteniamo inalterata questa proprietà**. È tipicamente usata quando viene inserito un nuovo elemento, per mantenere le proprietà del max-heap.

### Heap-Increase-Key ( $A, i, \text{key}$ )

```

if key < A[i]
    then error "new key is smaller"
  
```

```

A[i] ← key
  
```

```

while i > 1 and A[Parent(i)] < A[i]
    do exchange A[i] ↔ A[Parent(i)]
    i ← Parent(i)
  
```

Se ad esempio vogliamo cambiare la chiave di 4 in 15, esso salirà dalla posizione del 4 fino a salire fino al primo livello nel nodo di sinistra. Tutto il ciclo *while* ha tempo di esecuzione pari a  $O(\lg n)$ . Il tempo totale è pari a  $O(\lg n)$  perché il nuovo elemento è fatto risalire al massimo fino alla radice, quindi fino ad altezza  $h = \lg n$ .

## Max-Heap-Insert

La Max-Heap-Insert inserisce un nuovo elemento nel posto giusto. Per farlo:

- Aumenta la dimensione dell'heap.
- L'elemento vale inizialmente  $-\infty$ .
- All'elemento viene assegnato un valore key e viene spostato al posto giusto, per preservare le proprietà del max-heap.

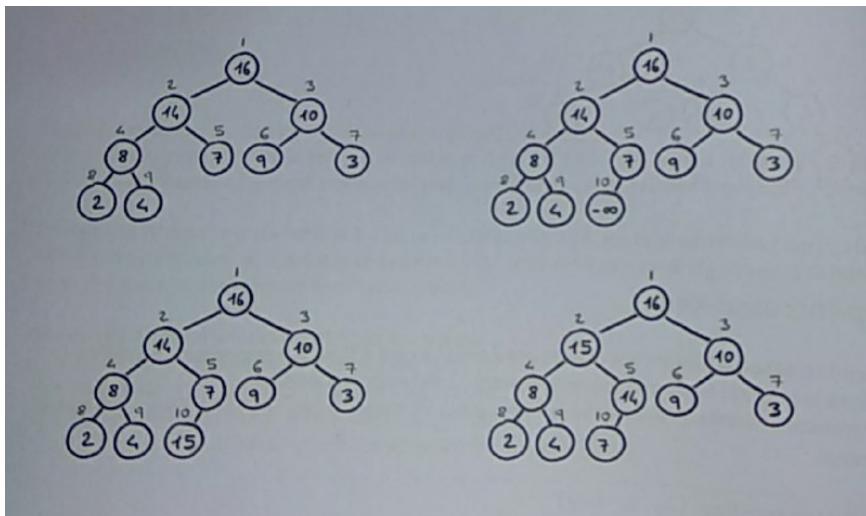
Impostare il nuovo nodo a  $-\infty$  è necessario, perché nella Heap-Increase-Key la chiave è confrontata con l'elemento in quella posizione. Quindi, se non ci fosse, il confronto avverrebbe con un elemento indefinito.

Max-Heap-Insert (A, key)

heap-size[A]  $\leftarrow$  heap-size[A] + 1

A[heap-size[A]]  $\leftarrow -\infty$

Heap-Increase-Key (A, heap-size[A], key)  $\rightarrow O(\log m)$



## Quick Sort

Il **quick sort** è un algoritmo di ordinamento **ricorsivo** non richiedente nessun requisito in particolare, è considerato uno dei più efficienti. **Ordina sul posto** ed utilizza un approccio **divide et impera**, nel caso peggiore ha un tempo d'esecuzione pari a  $\theta(n^2)$  e in quello medio  $\theta(n \lg n)$ . Nella pratica, risulta essere tra i più efficienti, grazie al fatto che i fattori "nascosti" in  $\Theta(n \lg n)$  sono piuttosto piccoli.

Esso consiste nel:

- Scegliere un elemento detto **pivot**.
- Partizionare il vettore in 2 sotto array che contengono rispettivamente i valori più piccoli e quelli più grandi del pivot.
- Il pivot è posizionato tra i 2 vettori.
- Continuare con questa modalità per ordinare i 2 sotto array.

## Pseudocodice

Quicksort (A, p, r)

```
if p < r  
    then q  $\leftarrow$  Partition (A, p, r)  
    Quicksort (A, p, q-1)  
    Quicksort (A, q+1, r)
```

Il quicksort esegue solo se il caso è non banale, partizioniamo il problema in due sottoproblemi, e l'algoritmo di partition ci restituisce l'indice del pivot(q). Le due successive invocazioni del quicksort saranno basate sulle posizioni del pivot. Per ordinare l'intero array si invoca Quicksort (A, 1, length[A]). Il quicksort

come visto è l'opposto al merge sort, a livello di tempo di esecuzione infatti la suddivisione dei sottoproblemi contribuisce maggiormente al tempo dell'algoritmo mentre la ricostruzione è banale.

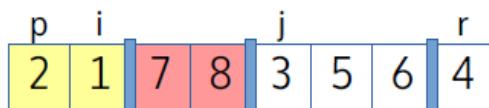
## Lezione 9 17/10/19

### Partition

La funzione Partition utilizza come pivot l'ultimo elemento dell'array, dopodichè crea i 2 sotto array confrontando ogni elemento con il pivot. Infine il pivot è spostato al centro dei 2 sotto-array, scambiandosi di posto con il primo elemento del secondo sotto-array.

Per un certo tempo, il vettore è quindi suddiviso in 4 aree:

- Gli elementi  $\leq$  del pivot [p,i].
- Gli elementi  $>$  del pivot [i+1 , j-1].
- Gli elementi non ancora confrontati [j, r-1].
- Il pivot [r].



Ad ogni iterazione, l'elemento **j** è confrontato col pivot **r**:

- Se è maggiore:
  - Incremento **j**
- Se è minore:
  - Incremento **i**
  - Scambio  $A[i]$  e  $A[j]$
  - Incremento **j**

Avremo così un vettore con a sinistra del pivot tutti i valori più piccoli di esso, e a destra tutti i valori più grandi del pivot. È possibile vedere sia un algoritmo che non ordina sul posto, che uno che lo fa ma ovviamente si preferisce quello che ordina sul posto poiché consuma meno memoria.

### Pseudocodice

```

Partition (A,p,r)
x ← A[r]
i ← p-1
for j ← p to r-1
    do if A[j] ≤ x
        then i ← i+1
        exchange A[i]↔A[j]
exchange A[i+1] ↔ A[r]
return i+1

```

$\Theta(n)$

#### Nota:

**j** = è il primo elemento non analizzato che deve essere confrontato con il pivot.

**k** = è il valore della cella di memoria

**x** = pivot

All'interno dei linguaggi di programmazione, in questo caso bisogna fare attenzione alla definizione di **i** perché potremmo uscire dall'area di memoria.

Es:

Inizialmente, due barriere (quella tra 1a e 2a area e tra 2a e 3a area) coincidono. Si confronta il primo elemento con il pivot se è minore o uguale:

- Si incrementa i (si sposta barriera tra 1a e 2a area)
- Si scambia A[i] con A[j]
- Si incrementa j (si sposta barriera tra 2a e 3a area)

Il secondo elemento è maggiore del pivot, si incrementa j (si sposta barriera tra 2a e 3a area). Il terzo elemento è maggiore del pivot, si incrementa j (si sposta barriera tra 2a e 3a area). Il quarto elemento è minore o uguale del pivot,

Il quinto elemento è minore o uguale del pivot, il sesto elemento è maggiore del pivot, il settimo elemento è maggiore del pivot, il ciclo finisce scambiando A[i+1] con A[r]. Finito il ciclo, risituiamo la posizione del pivot e poi lavoriamo sui due sottogruppi separatamente con il quicksort. Il tempo di esecuzione è  $\theta(n)$  con  $n = r-p+1$ , le n iterazioni impiegano un tempo costante.

**L'invariante** dell'algoritmo di partitioning è che all'inizio di ogni iterazione del ciclo *for* avremo che:

- Se  $p \leq k \leq i$  allora  $A[k] \leq x$
- Se  $i+1 \leq k \leq j-1$  allora  $A[k] > x$
- Se  $k = r$  allora  $A[k] = x$

Dimostriamo che l'invariante è vero. Alla prima iterazione ( $i=p-1$  e  $j=p$ ) sia l'area degli elementi minori che quella degli elementi maggiori sono vuote. Dato che per ogni iterazione l'elemento in  $j$  è spostato in una delle due aree, si conserva la veridicità dell'invariante. L'algoritmo è corretto, infatti il ciclo termina quando  $j=r$  cioè l'array è suddiviso in 3 insiemi che rispettano le proprietà dell'invariante.

i	p,j	2	8	7	1	3	5	6	r
		2	8	7	1	3	5	6	4

p,j,i	2	8	7	1	3	5	6	r
	2	8	7	1	3	5	6	4

p,i	j	2	8	7	1	3	5	6	r
	2	8	7	1	3	5	6	4	

p,i	j	2	8	7	1	3	5	6	r
	2	8	7	1	3	5	6	4	

p,i	j	2	8	7	1	3	5	6	r
	2	8	7	1	3	5	6	4	

p,i	j	2	8	7	1	3	5	6	r
	2	8	7	1	3	5	6	4	

p,i	j	2	1	7	8	3	5	6	r
	2	1	7	8	3	5	6	4	

p,i	j	2	1	7	8	3	5	6	r
	2	1	7	8	3	5	6	4	

p,i	j	2	1	3	8	7	5	6	r
	2	1	3	8	7	5	6	4	

p,i	j	2	1	3	8	7	5	6	r
	2	1	3	8	7	5	6	4	

p,i	j	2	1	3	8	7	5	6	r
	2	1	3	8	7	5	6	4	

p,i	j	2	1	3	8	7	5	6	r
	2	1	3	8	7	5	6	4	

p,i	j	2	1	3	8	7	5	6	r
	2	1	3	8	7	5	6	4	

p,i	j	2	1	3	8	7	5	6	r
	2	1	3	8	7	5	6	4	

p,i	j	2	1	3	8	7	5	6	r
	2	1	3	8	7	5	6	4	

p,i	j	2	1	3	8	7	5	6	r
	2	1	3	8	7	5	6	4	

### Analisi di complessità

Valutiamo adesso quicksort complessivamente, considerando anche l'algoritmo di partitioning. Studiamo 3 casi:

#### Caso peggiore:

Il caso peggiore si presenta quando in ogni iterazione i due sottoarray sono totalmente sbilanciati, ossia uno ha dimensione **0** e l'altro ha dimensione **n-1**.

$$T(n) = T(n-1) + T(0) + \theta(n) = T(n-1) + \theta(n)$$

avremo che ricorsivamente

$$\theta(1) + \theta(n) \rightarrow \sum_{i=1}^n \frac{n(n+1)}{2} \cong n^2$$

quindi la soluzione

$$\text{risulta } T(n) = \theta(n^2).$$

Non siamo sicuri se questo sia il caso peggiore, ovvero  $\theta(n^2)$ , però possiamo sicuramente dire che il caso peggiore è  $T(n) = \Omega(n^2)$ , questo perché esiste un caso in cui vale  $\theta(n^2)$ .

Dimostriamolo:

$$T(n) = \max[T(q) + T(n-q+1)] + \theta(n) \leq \text{dove}$$

$T(q)$  è l'area degli elementi minori, mentre l'altra è l'area degli elementi maggiori, il max è con  $0 \leq q \leq n-1$ . Lo scriviamo in questa forma perché è un algoritmo che utilizza il divide et impera. Priviamo a risolvere l'equazione con il metodo di sostituzione con  $T(n) = O(n^2)$ .

$$\leq [cq^2 + c(n-q-1)^2] + \theta(n) = c * \max[q^2 + (n-q-1)^2] + \theta(n) \leq c$$

può essere portato fuori perché una costante positiva, se fosse stata negativa avremmo invertito il max con il minimo.

Ora abbiamo che l'espressione tra parentesi quadre è una parabola rivolta verso l'alto e quindi il max è negli estremi ovvero  $q = 0 \rightarrow (n-1)^2$  e vale anche che  $q = n-1 \rightarrow (n-1)^2$ .

Quindi

$$c * \max[q^2 + (n - q - 1)^2] + \theta(n) \leq c * (n - 1)^2 + \theta(n) = cn^2 - c(2n - 1) + \theta(n) \leq cn^2$$

ma è valido se  $c(2n - 1) \geq d^*n$ .

Il tutto ci porta a dire che  $T(n) = O(n^2)$ , quindi otteniamo che valgono in contemporanea:

$$\begin{cases} T(n) = \Omega(n^2) \\ T(n) = O(n^2) \end{cases} \rightarrow T(n) = \theta(n^2)$$

### Caso migliore:

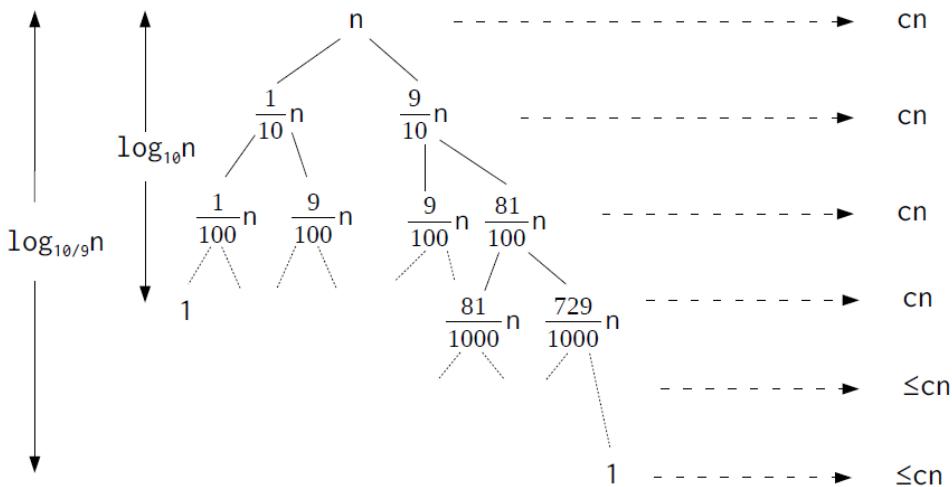
Il caso migliore si ha quando ad ogni iterazione i 2 sottoarray sono perfettamente bilanciati, ossia con dimensione  $\left\lfloor \frac{n}{2} \right\rfloor$  e  $\left\lceil \frac{n}{2} \right\rceil - 1$ . In questo caso si ha:

$T(n) \leq 2T\left(\frac{n}{2}\right) + \theta(n)$  possiamo maggiorarlo perché abbiamo delle approssimazioni per difetto. Si soddisfano i requisiti per poter usare il **teorema dell'esperto (caso 2)** e abbiamo che la soluzione dell'equazione ricorrente è:  $T(n) = O(n \lg n)$  e non è theta perché vi è il  $\leq$

### Caso bilanciato:

Consideriamo il caso in cui abbiamo un bilanciamento fisso di 9:1 ad ogni iterazione, in questo caso si ha:

$T(n) \leq T\left(\frac{9n}{10}\right) + T\left(\frac{n}{10}\right) + cn$ . Il tempo medio di esecuzione di Quicksort è più vicino al caso migliore, per vederlo utilizziamo il metodo dell'albero di ricorrenza; il quale riporta le dimensioni dei

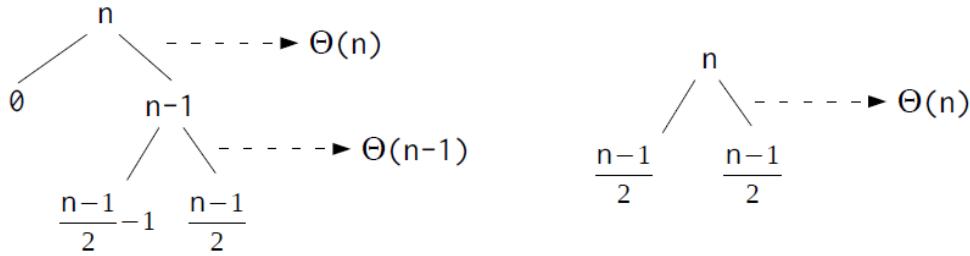


sottoproblemi.

Questo risultato vale per qualsiasi rapporto, anche molto sbilanciato come 99:1. Si nota come è più vicino al caso migliore che al caso peggiore.  $T(n) \leq n \log_{\frac{10}{9}} n = n \lg n * \log_{\frac{10}{9}} 2 = O(n \lg n)$ .

### Caso medio:

Nel caso medio si alternano partizioni “buone” e “cattive”. Intuitivamente, se si alternano i best e i worst case, i worst case vengono annullati dai best case.



Possiamo notare che la partizione negativa ha aggiunto un termine  $n$ , che però non influisce sul totale perché asintoticamente si ha sempre  $\Theta(n)$  in quel punto. Quindi è come se non avessimo avuto la partizione cattiva, e il totale rimane  $\Theta(n \lg n)$ .

## Quicksort randomizzato

Una tecnica per aumentare la probabilità di ottenere un partizionamento ben bilanciato consiste nello scegliere il pivot casualmente ad ogni iterazione, piuttosto che prendere sempre l'ultimo elemento. Dunque l'unica differenza nel codice sta nel fatto che prima di effettuare la partizione scambiamo l'ultimo elemento con uno a caso all'interno del vettore.

```
Randomized-Partition (A, p, r)
i ← Random (p, r)
exchange A[r] ↔ A[i]
return Partition (A, p, r)
```

```
Randomized-Quicksort (A, p, r)
if p < r
    then q ← Randomized-Partition (A, p, r)
        Randomized-Quicksort (A, p, q-1)
        Randomized-Quicksort (A, q+1, r)
```

La versione randomizzata di Quicksort è la **migliore quando si ordinano array di grosse dimensioni**. L'uso della funzione random non influisce sulla complessità poiché la si reputa di tempo costante.

## Lezione 10 23/10/19

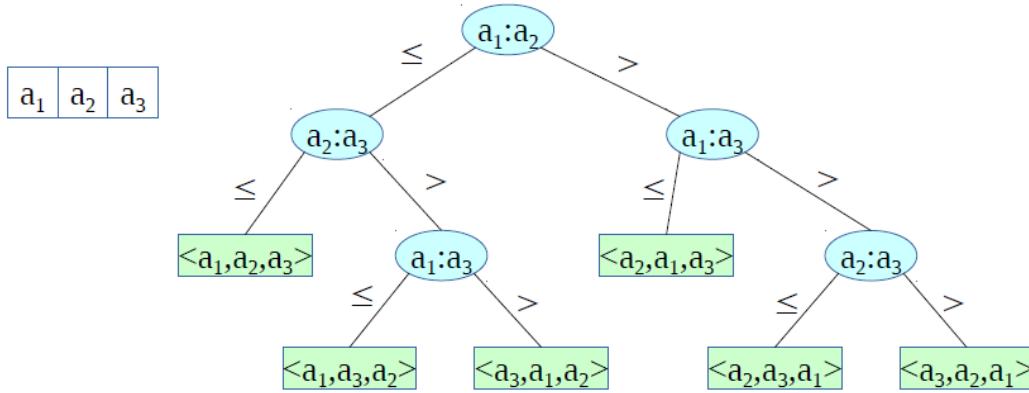
### Algoritmi lineari

#### Algoritmi basati sul confronto

Gli algoritmi di ordinamento visti fino ad ora sono **basati sul confronto (comparison sort)**. Possiamo ora dimostrare che per questo tipo di algoritmi nel caso peggiore sono necessari almeno  $\Omega(n \lg n)$  confronti. Quindi qualsiasi algoritmo basato sul confronto richiede un tempo **non lineare**.

Le operazioni di un algoritmo basato su confronto corrispondono a uno specifico percorso della radice ad una foglia. Ovviamente c'è una foglia per ogni possibile soluzione, quindi le foglie sono almeno  $n!$ , che corrisponde al numero di permutazioni degli  $n$  elementi.

Consideriamo ad esempio l'albero di Insertion Sort per un vettore di dimensione 3.



Quando andiamo a considerare un'istanza precisa del problema, arriviamo alla soluzione(foglia) attraverso un determinato cammino che riprende tutti i confronti fatti. Il tempo d'esecuzione del caso peggiore è legato al numero di confronti che vengono effettuati. Questo numero è dato dalla lunghezza del percorso più lungo che porta alla soluzione, che coincide con l'altezza dell'albero.

Il limite inferiore  **$\Omega$  per il caso peggiore** è la minima altezza di tutti gli alberi di decisione dei vari algoritmi.

**Teorema:**

Qualunque algoritmo di ordinamento basato su confronti richiede  **$\Omega(n \lg n)$**  confronti nel caso peggiore.

$n! \leq 2^h$  dove  $n!$  è il numero minimo di foglie mentre l'altro è il numero massimo di foglie

S e

$$h \geq \lg(n!) = \lg(n(n-1)(n-2)\dots) = \lg((n^2-n)(n-2)\dots) = \lg((n^3-3n^2+2n)\dots) = \lg(n^n + \dots) = \theta(\lg n^n) = \lg \theta(n \lg n)$$

altrimenti  $h = \Omega(n \lg n)$

Questo significa che per ottenere un tempo **lineare** occorre sfruttare meccanismi diversi dal confronto di elementi.

## Counting Sort

Il Counting Sort è un algoritmo di ordinamento per vettori di interi compresi tra **0** e **k**. Consiste nel contare, per ogni elemento *i* del vettore:

- Quanti elementi uguali ad *i* ci sono.
- Quanti elementi  $\leq i$  ci sono.

Queste informazioni ci indicano in che posizione deve stare ogni elemento.

Tuttavia come detto in precedenza **non** stiamo affrontando un algoritmo **basato sul confronto**, quindi non andiamo a confrontare direttamente *i* con gli altri elementi. Infatti, per contare quanti elementi minori o uguali ad *i* ci sono nel vettore A, consideriamo un **vettore C** in cui l'elemento **C[i]** è il numero di occorrenze dei numeri  $\leq i$  presenti in A.

È un algoritmo che **non ordina sul posto**, poiché sfrutta un vettore ausiliario **C** (che indica le occorrenze di ciascun valore compreso tra 0 e k) ed uno **B** (che mantiene i valori ordinati) con dimensione variabile.

Esempio: Valori interi tra 0 e k = 5.

I passi del Counting sort sono:

- Inizializzare a 0 un vettore C di  $k+1$  elementi.

1)	A	<table border="1" style="display: inline-table; vertical-align: middle;"><tr><td>2</td><td>5</td><td>3</td><td>0</td><td>2</td><td>3</td><td>0</td><td>3</td></tr></table>	2	5	3	0	2	3	0	3	C	<table border="1" style="display: inline-table; vertical-align: middle;"><tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td></tr><tr><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td><td>0</td></tr></table>	0	1	2	3	4	5	0	0	0	0	0	0
2	5	3	0	2	3	0	3																	
0	1	2	3	4	5																			
0	0	0	0	0	0																			

2)	A	<table border="1" style="display: inline-table; vertical-align: middle;"><tr><td>2</td><td>5</td><td>3</td><td>0</td><td>2</td><td>3</td><td>0</td><td>3</td></tr></table>	2	5	3	0	2	3	0	3	C	<table border="1" style="display: inline-table; vertical-align: middle;"><tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td></tr><tr><td>0</td><td>0</td><td>1</td><td>0</td><td>0</td><td>0</td></tr></table>	0	1	2	3	4	5	0	0	1	0	0	0
2	5	3	0	2	3	0	3																	
0	1	2	3	4	5																			
0	0	1	0	0	0																			

A	<table border="1" style="display: inline-table; vertical-align: middle;"><tr><td>2</td><td>5</td><td>3</td><td>0</td><td>2</td><td>3</td><td>0</td><td>3</td></tr></table>	2	5	3	0	2	3	0	3	C	<table border="1" style="display: inline-table; vertical-align: middle;"><tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td></tr><tr><td>0</td><td>0</td><td>1</td><td>0</td><td>0</td><td>1</td></tr></table>	0	1	2	3	4	5	0	0	1	0	0	1
2	5	3	0	2	3	0	3																
0	1	2	3	4	5																		
0	0	1	0	0	1																		

C	<table border="1" style="display: inline-table; vertical-align: middle;"><tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td></tr><tr><td>0</td><td>0</td><td>1</td><td>1</td><td>0</td><td>0</td></tr></table>	0	1	2	3	4	5	0	0	1	1	0	0
0	1	2	3	4	5								
0	0	1	1	0	0								

3)	A	<table border="1" style="display: inline-table; vertical-align: middle;"><tr><td>2</td><td>5</td><td>3</td><td>0</td><td>2</td><td>3</td><td>0</td><td>3</td></tr></table>	2	5	3	0	2	3	0	3	C	<table border="1" style="display: inline-table; vertical-align: middle;"><tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td></tr><tr><td>2</td><td>0</td><td>2</td><td>3</td><td>0</td><td>1</td></tr></table>	0	1	2	3	4	5	2	0	2	3	0	1
2	5	3	0	2	3	0	3																	
0	1	2	3	4	5																			
2	0	2	3	0	1																			

C	<table border="1" style="display: inline-table; vertical-align: middle;"><tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td></tr><tr><td>2</td><td>2</td><td>2</td><td>3</td><td>0</td><td>1</td></tr></table>	0	1	2	3	4	5	2	2	2	3	0	1
0	1	2	3	4	5								
2	2	2	3	0	1								

C	<table border="1" style="display: inline-table; vertical-align: middle;"><tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td></tr><tr><td>2</td><td>2</td><td>4</td><td>3</td><td>0</td><td>1</td></tr></table>	0	1	2	3	4	5	2	2	4	3	0	1
0	1	2	3	4	5								
2	2	4	3	0	1								

C	<table border="1" style="display: inline-table; vertical-align: middle;"><tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td></tr><tr><td>2</td><td>2</td><td>4</td><td>7</td><td>0</td><td>1</td></tr></table>	0	1	2	3	4	5	2	2	4	7	0	1
0	1	2	3	4	5								
2	2	4	7	0	1								

C	<table border="1" style="display: inline-table; vertical-align: middle;"><tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td></tr><tr><td>2</td><td>2</td><td>4</td><td>7</td><td>7</td><td>1</td></tr></table>	0	1	2	3	4	5	2	2	4	7	7	1
0	1	2	3	4	5								
2	2	4	7	7	1								

C	<table border="1" style="display: inline-table; vertical-align: middle;"><tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td></tr><tr><td>2</td><td>2</td><td>4</td><td>7</td><td>7</td><td>8</td></tr></table>	0	1	2	3	4	5	2	2	4	7	7	8
0	1	2	3	4	5								
2	2	4	7	7	8								

4)	A	<table border="1" style="display: inline-table; vertical-align: middle;"><tr><td>2</td><td>5</td><td>3</td><td>0</td><td>2</td><td>3</td><td>0</td><td>3</td></tr></table>	2	5	3	0	2	3	0	3	C	<table border="1" style="display: inline-table; vertical-align: middle;"><tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td></tr><tr><td>2</td><td>2</td><td>4</td><td>7</td><td>7</td><td>8</td></tr></table>	0	1	2	3	4	5	2	2	4	7	7	8
2	5	3	0	2	3	0	3																	
0	1	2	3	4	5																			
2	2	4	7	7	8																			

B	<table border="1" style="display: inline-table; vertical-align: middle;"><tr><td></td><td></td><td></td><td></td><td></td><td></td><td>3</td></tr></table>							3
						3		

A	<table border="1" style="display: inline-table; vertical-align: middle;"><tr><td>2</td><td>5</td><td>3</td><td>0</td><td>2</td><td>3</td><td>0</td><td>3</td></tr></table>	2	5	3	0	2	3	0	3	C	<table border="1" style="display: inline-table; vertical-align: middle;"><tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td></tr><tr><td>2</td><td>2</td><td>4</td><td>6</td><td>7</td><td>8</td></tr></table>	0	1	2	3	4	5	2	2	4	6	7	8
2	5	3	0	2	3	0	3																
0	1	2	3	4	5																		
2	2	4	6	7	8																		

B	<table border="1" style="display: inline-table; vertical-align: middle;"><tr><td>0</td><td>0</td><td>2</td><td>2</td><td>3</td><td>3</td><td>3</td><td>5</td></tr></table>	0	0	2	2	3	3	3	5
0	0	2	2	3	3	3	5		

A	<table border="1" style="display: inline-table; vertical-align: middle;"><tr><td>2</td><td>5</td><td>3</td><td>0</td><td>2</td><td>3</td><td>0</td><td>3</td></tr></table>	2	5	3	0	2	3	0	3	C	<table border="1" style="display: inline-table; vertical-align: middle;"><tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td></tr><tr><td>0</td><td>2</td><td>2</td><td>4</td><td>7</td><td>7</td></tr></table>	0	1	2	3	4	5	0	2	2	4	7	7
2	5	3	0	2	3	0	3																
0	1	2	3	4	5																		
0	2	2	4	7	7																		

B	<table border="1" style="display: inline-table; vertical-align: middle;"><tr><td>0</td><td>0</td><td>2</td><td>2</td><td>3</td><td>3</td><td>3</td><td>5</td></tr></table>	0	0	2	2	3	3	3	5
0	0	2	2	3	3	3	5		

• Incrementare ogni elemento di C in base a quante volte il suo indice compare in A.

• Aggiungere ad ogni elemento di C la somma dei precedenti.

• Inserire ogni elemento x di A in un vettore B nella posizione C[x], devrementando poi C[x].

NOTA: Nel vettore C il valore 7 è arancione perché vuol dire che esistono 7 elementi minori o uguali di 3, quindi ne mettiamo uno proprio in 7° posizione. Nel posizionamento dei valori nel vettore B scorriamo A al contrario.

### Pseudocodice

Si può notare che l'ultimo *for* scorrere A al contrario, questo serve a garantire la proprietà di **stabilità**, ossia che gli elementi uguali sono messi nel vettore B nello stesso ordine in cui compaiono nel vettore A. Inoltre abbiamo supposto che gli indici di C partono da **0**.

```

Counting-Sort (A,B,k)
for i ← 0 to k    ] → Θ(k)
    do C[i] ← 0
for j ← 1 to length[A]    ] → Θ(n)
    do C[A[j]] ← C[A[j]]+1
// C[i] è il numero di occorrenze di i
for i ← 1 to k    ] → Θ(k)
    do C[i] ← C[i]+C[i-1]
// C[i] è il numero di elementi ≤ i
for j ← length[A] downto 1    ] → Θ(n)
    do B[C[A[j]]] ← A[j]
    C[A[j]] ← C[A[j]]-1

```

### Analisi di complessità

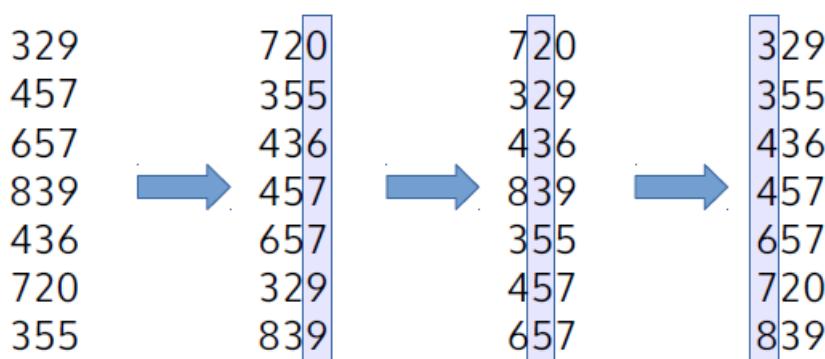
La somma delle complessità è  $\theta(k + n)$  ma counting sort è in genere usato quando  $k = O(n)$ , quindi il tempo d'esecuzione è pari a  $\theta(n)$ .

$$T(n) = 2\theta(k) + 2\theta(n) = \theta(n)$$

Quindi counting sort è efficiente se  $k = O(n)$ , cioè se  $k \leq cn$ . Inoltre, conviene quando  $k$  è relativamente piccolo, se ad esempio gli elementi del vettore sono interi a 32 bit,  $k$  può essere anche circa 4 miliardi. In questo caso non conviene usare il counting sort, poiché difficilmente avremo un vettore con  $n$  più grande di 4 miliardi. Se  $k > cn$  perderemmo tempo a contare le occorrenze, invece che ad ordinare e quindi potrebbe diventare più lento.

### Radix Sort

Il **radix sort** è un algoritmo di ordinamento che assume di lavorare con vettori i cui elementi hanno **d** cifre. Gli elementi sono ordinati a partire dalla cifra meno significativa, usando un algoritmo **stabile** (ad esempio usando il counting sort). La stabilità permette di perservare l'ordine dei valori.



Se invece di cominciare dall'ultima cifra, avessimo incominciato dalla prima il vettore sarebbe ovviamente ordinato solo rispetto all'ultima cifra. Si può notare che la proprietà di **stabilità**, in questi casi, è essenziale perché in questo modo due numeri che iniziano con la stessa cifra rimarranno ordinati in base alla seconda

cifra. Se non ci fosse la stabilità tutti gli ordinamenti, tranne l'ultimo, sarebbero inutili perché "sovrascritti" dall'ultimo.

## Pseudocodice

```
Radix-Sort (A, d)
for i ← 1 to d
    do use a stable sort to sort array A on digit i
```

La cifra di posto 1 è quella meno significativa

Il *for* va da **1** a **d**, dove 1 è la cifra meno significativa e d quella più significativa.

## Analisi di correttezza

L'invariante del radix sort è:

All'inizio di ogni iterazione gli elementi sono ordinati considerando solo le cifre da **1** a **i-1**.

Dimostriamo la verità dell'invariante:

- Prima della prima iterazione gli elementi da **1** a **i-1** sono in realtà zero.
- Ogni iterazione mantiene l'ordinamento corretto per via della stabilità, quindi conserva la veridicità dell'invariante.
- L'algoritmo quindi è corretto, infatti quando usciamo dal ciclo abbiamo ordinati tutti gli elementi considerando le cifre da **1** a **d**, ovvero tutte le cifre.

## Analisi di complessità

Poiché le cifre vanno da 0 a 9, possiamo utilizzare efficientemente il Counting Sort con **k=9**.

$$T(n) = d \theta(n + k) = \theta(d(n + k)) = \theta(n) \text{ sia se } n \gg 9 \text{ e } n \gg d.$$

Un caso in cui compaiono numeri con lo stesso numero di cifre è quello dei registri di un calcolatore, per cui si può pensare il radix sort non sui valori degli elementi, ma sulle loro rappresentazioni. Il tempo è lineare se **d** è costante e **k = O(n)**.

In realtà ordinare sulla base decimale non è una grande idea se si lavora su un calcolatore, poiché per calcolare l'ultima cifra decimale dobbiamo effettuare il resto del modulo 10 che non ha un algoritmo veloce e quindi appesantire il tempo di esecuzione, anche se è sempre lineare. Si sceglie quindi di valutare la rappresentazione dei bit

Es.

Valutiamo il numero 5866 in bit sarà = 0001011011101010 avremo quindi **k = 1** e **d = 16 = b**.

$$T(n) = \theta(16(n + 1)) = \theta(n) \text{ però posso migliorare quel 16.}$$

Il fattore che moltiplica **n** in questo caso è **b**, cioè il numero di bit. Questo può notevolmente migliorato se considero i bit in **r** gruppi invece che singolarmente.

$$0001|0110|1110|1010 \text{ avremo } k = 2^r - 1 = 15 \text{ e } d \left\lceil \frac{e}{r} \right\rceil = \frac{16}{4} = b.$$

$$T(n) = \theta\left(\frac{b}{r}(n + 2^r)\right)$$

Per minimizzare il tempo d'esecuzione basta scegliere un  $r$  abbastanza grande da rendere  $\frac{b}{r}$  molto vicino a 1 (ma non può essere più piccolo di 1 per la frazione), ma non troppo grande da dominare  $n$ . In particolare:

- Se  $b < \lg n \rightarrow 2^b < n$  conviene  $r$  più grande possibile  $\rightarrow \theta(n + 2^b) = \theta(n)$ . Scegliere  $r = d$  mi riduce il tempo a counting sort. Quindi il numero degli elementi da ordinare è maggiore del numero degli elementi rappresentabili.
- Se  $b \geq \lg n \rightarrow 2^b \geq n$  conviene scegliere  $r = \lg n$  per minimizzare l'espressione. Quindi  $\theta\left(\frac{b}{\lg n}(n + n)\right) = \theta\left(\frac{bn}{\lg n}\right) \cong \theta(n)$  avendo il rapporto di  $b$  e  $\lg n$  maggiore di 1. Al crescere di  $r$  diventa più veloce il radix sort rispetto al counting sort

**Facendo delle considerazioni finali** a prima vista Radix Sort, ha tempo pari a  $\theta(n)$ , può sembrare migliore di Quick Sort, che è  $\theta(n \lg n)$ . In realtà però, dipende dalla situazione, poiché richiede meno passaggi del Quick Sort ma ogni passaggio potrebbe impiegare un tempo molto maggiore.

Inoltre se la memoria è limitata, non conviene usare il Radix Sort che sfrutta il counting sort, poiché questo non ordina sul posto.

## Bucket Sort

Il **bucket sort**, rispetto ai precedenti algoritmi, non richiede che i numeri siano interi o che siano rappresentabili su cifre decimali finiti. Il bucket sort quindi è usato per ordinare vettori i cui elementi sono distribuiti uniformemente tra  $[0, 1]$ . La distribuzione uniforme serve per dimostrare il tempo lineare.

Bucket sort consiste in:

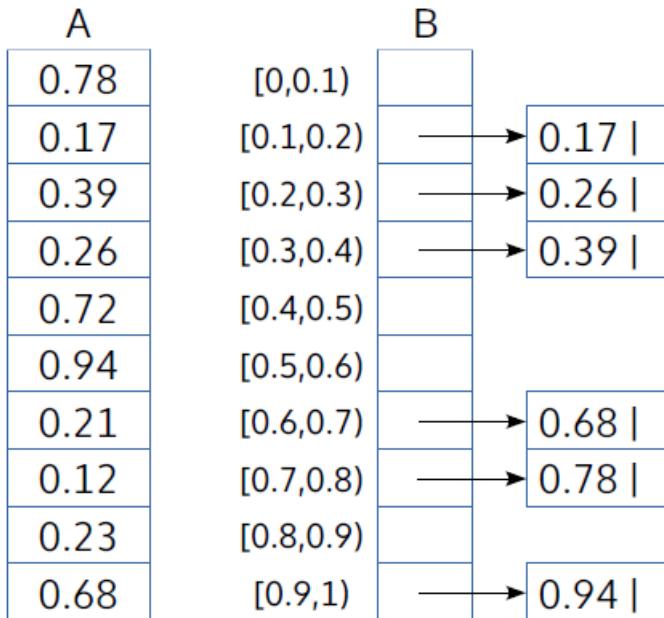
- Dividere l'intervallo  $[0, 1]$  in  $n$  sottointervalli uguali, detti **bucket**.
- Distribuire gli elementi nei rispettivi bucket
- Ordinare i bucket con **Insertion sort**
- Concatenare i bucket in un'unica lista.

Bucket sort **non ordina sul posto**, poiché usa un vettore ausiliario  $B[0, \dots, n-1]$  di liste collegate. Inoltre l'ipotesi di distribuzione uniforme garantisce la linearità del tempo di esecuzione, poiché indica che ogni bucket abbia più o meno lo stesso numero di elementi, per cui l'insertion sort lavora su pochissimi elementi.

Es:

Dividiamo il vettore B nelle varie frazioni di 1. Poi quando troviamo dei valori simili, come 0.78 e 0.72 il secondo viene inserito in **testa**, poiché l'inserimento in testa è molto più veloce di quello in coda. Se la lista è doppiamente concatenata allora non vi è differenza per l'inserimento tra testa e coda.

Infine concateniamo tutte le liste insieme.



Lezione 11 24/10/19

Pseudocodice

### Bucket-Sort (A)

```

n ← length[A]
for i ← 1 to n
    do insert A[i] into list B[ $\lfloor n \cdot A[i] \rfloor$ ] }  $\Theta(n)$ 
for i ← 0 to n-1
    do sort list B[i] with insertion sort }  $\Theta(n) \cdot \Theta(\text{?})$ 
Concatenate the lists B[0], ..., B[n-1] in order →  $\Theta(n)$ 

```

Notiamo che per inserire un elemento nel corrispettivo sottointervallo, basta moltiplicarlo per  $n$  (numero di sottointervalli) e arrotondarlo per difetto, ottenendo così l'indice del sottointervallo:

$$j = \lfloor n \cdot A[i] \rfloor$$

Utilizziamo poi per ordinare l'algoritmo di insertion sort poiché quest'ultimo lavora bene su vettori di piccole dimensioni. Una volta ordinate le liste le concateniamo dalla prima all'ultima, terminando così l'algoritmo. Il vettore delle liste **B** ha gli indici vanno da **0** a **n-1**.

Analisi di correttezza

Se  $A[i] \leq A[j]$ , anche  $\lfloor n \cdot A[i] \rfloor \leq \lfloor n \cdot A[j] \rfloor$ , quindi  $A[i]$  è inserito o nella stessa lista di  $A[j]$  o in quella precedente. Nel primo caso, il secondo ciclo *for* lo mette nel giusto ordine, nel secondo caso la concatenazione delle liste lo mette nel giusto ordine. Abbiamo così dimostrato la correttezza dell'algoritmo.

Analisi di complessità

Il primo ciclo *for* e la concatenazione sono lineari, ma nel secondo ciclo *for* c'è l'insertion sort che sappiamo essere  $O(n^2)$ , o meglio  $O(n_i^2)$  dove  $n_i$  è il numero di elementi in ogni lista  $B[i]$ .

$$T(n) = \theta(n) + \sum_{i=0}^{n-1} O(n_i^2).$$

Valuteremo il tempo atteso  $E[T(n)]$  poiché valuteremo tutte le possibili distribuzioni del vettore da ordinare. Consideriamo ora la variabile aleatoria indicatore  $X_{ij}$  che denota il numero di elementi nella lista  $B[i]$ . Vale 1 se l'elemento  $j$  cade nell'intervallo della lista di  $i$ , vale 0 altrimenti.

$$X_{ij} = \begin{cases} 1 & \text{con probabilità } \frac{1}{n} \\ 0 & \text{con probabilità } 1 - \frac{1}{n} \end{cases}$$

Semplifichiamo l'espressione usando la variabile aleatoria al posto di  $n_i = \sum_{j=1}^n X_{ij}$ , questo perché sommare le  $X_{ij}$  vuol dire andare a prendere le variabili indicatorie che mi dicono se tutti gli elementi del vettore partenza, da 1 fino ad  $n$ , finiscono nella lista **iesima**:

$$E[n_i^2] = E\left[\left(\sum_{j=1}^n X_{ij}\right)^2\right] = E\left[\sum_{j=1}^n X_{ij} \sum_{k=1}^n X_{ik}\right] = E\left[\sum_{j=1}^n \sum_{k=1}^n X_{ij} X_{ik}\right] = E\left[\sum_{j=1}^n X_{ij}^2 + \sum_{j=1}^n \sum_{k=1}^n X_{ij} X_{ik}\right] =$$

La media di una variabile aleatoria discreta è pari alla sommatoria dei valori per la probabilità di assumere quei valori. L'ultima sommatoria è quando consideriamo  $k \neq j$ . Dove si considera  $X_{ij}$  l'elemento jesimo finisce nella lista iesima, mentre  $X_{ik}$  è l'elemento kesimo che finisce nella lista iesima. Possiamo assumere anche le due variabili aleatorie siano indipendenti.

$$= \sum_{j=1}^n E[X_{ij}^2] + \sum_{j=1}^n \sum_{k=1}^n E[X_{ij} X_{ik}] = \text{la prima sommatoria è pari a: } 1 * \frac{1}{n} + 0 \left(1 - \frac{1}{n}\right) = \frac{1}{n}$$

$$\text{mentre le altre sommatorie sono pari a } E[X_{ij}] E[X_{ik}] = \frac{1}{n} * \frac{1}{n} = \frac{1}{n^2}$$

E quindi dalle sommatorie precedenti arriviamo a:

$$\sum_{j=1}^n \frac{1}{n} + \sum_{\substack{k=1 \\ k \neq j}}^n \frac{1}{n^2} = n * \frac{1}{n} + n(n-1) * \frac{1}{n^2} = 1 + \frac{n-1}{n} = 2 - \frac{1}{n}$$

Questo che abbiamo trovato è quindi il valor medio di  $E[n_i^2]$  quando le distribuzioni sono uniformi, ora lo sostituiamo nel calcolo della complessità:

$$E[T(n)] = E\left[\theta(n) + \sum_{i=0}^{n-1} O(n_i^2)\right] = \theta(n) + \sum_{i=0}^{n-1} O(E[n_i^2]) = \text{La notazione O è limitata}$$

superiormente da  $2 - \frac{1}{n}$  ma asintoticamente tende a 0. Quindi come visto altre volte ha un tempo costante. Riscriviamo il tutto come:  $\theta(n) + O(2n - 1) = \theta(n) + O(n) = \theta(n)$ .

Quindi Bucket Sort ha tempo di esecuzione  $\Theta(n)$ , cioè **lineare**. In realtà si vede che la linearità si può ottenere anche senza una distribuzione uniforme, purché la somma dei quadrati delle dimensioni dei bucket sia lineare con il numero totale degli elementi.

## Algoritmi di selezione

### Statistiche d'ordine $i$

In un insieme di  $n$  elementi, l'**iesimo** elemento più piccolo è detto **statistica d'ordine  $i$** . Dunque:

- La statistica di ordine primo è il minimo
- La statistica di ordine  $n$  è il massimo

La mediana informalmente è definita come il valore di mezzo. Formalmente il valore per cui metà degli elementi è maggiore e l'altra metà è minore è detto **mediana**:

- $n$  è dispari  $\rightarrow$  ordine  $\frac{n+1}{2}$
- $n$  è pari  $\rightarrow$  ordine  $\frac{n+1}{2}$
- **mediana inferiore**  $\rightarrow$  ordine  $\left\lfloor \frac{n+1}{2} \right\rfloor$
- **mediana superiore**  $\rightarrow$  ordine  $\left\lceil \frac{n+1}{2} \right\rceil$

Se il numero di elementi è pari, le due mediane saranno distinte, altrimenti coincidono. Per selezionare la statistica di ordine  $i$  da un insieme di elementi distinti si potrebbe usare un algoritmo di ordinamento e prendendo poi l'elemento al posto  $i$ . È possibile però farlo anche con algoritmi lineari. Il vantaggio degli **algoritmi di selezione** della statistica di ordine  $i$  è che non fanno ipotesi sulle caratteristiche dei valori del vettore in ingresso.

### Selezione di Minimo e Massimo

Vediamo ad esempio come trovare minimo e massimo:

```
MinMax (A)
min ← max ← A[1]
for i ← 2 to length[A]
    do if A[i] < min
        then min ← A[i]
    if A[i] > max
        then max ← A[i]
return min, max
```

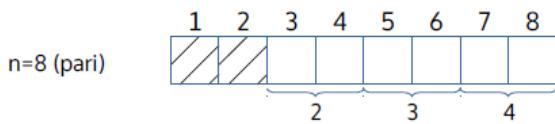
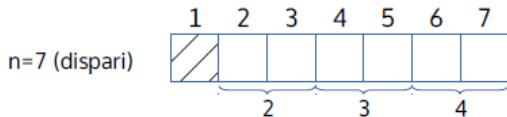
Questo non è il modo più veloce per trovare il minimo e il massimo, perché per ordinare una sequenza di  $n$  elementi generici, non possiamo sperare di avere un tempo di esecuzione migliore di  $n \lg n$ . Questo codice effettua due confronti alla volta, per un totale di  $2(n-1)$ , poiché tra  $n$  elementi sono necessari  $n-1$  confronti. Il **tempo asintotico di questo algoritmo è sempre  $\Theta(n)$**  ma consideriamo anche i fattori moltiplicativi che sono nascoste dalla notazione asintotica.

È possibile effettuare anche tre confronti alla volta, ovvero confrontando:

- 2 elementi tra loro.
- Il più piccolo dei 2 con il massimo.

- Il più grande dei 2 con il minimo.

In questo caso si effettueranno  $3 \left\lfloor \frac{n}{2} \right\rfloor$  confronti in totale.



Scriveremo quindi un algoritmo più efficiente e veloce.

```
QuickMinMax (A)
n ← length[A]
if n dispari
    then min ← max ← A[1]
else
    if A[1] < A[2]
        then min ← A[1], max ← A[2]
        else min ← A[2], max ← A[1]
for i ← 2 to  $\lfloor (n+1)/2 \rfloor$ 
    if A[2*i-1-n%2] < A[2*i-n%2]
        then low ← A[2*i-1-n%2], high ← A[2*i-n%2]
        else low ← A[2*i-n%2], high ← A[2*i-1-n%2]
    if low < min
        then min ← low
    if high > max
        then max ← high
return min, max
```

## Lezione 12 30/10/19

### Statistiche d'ordine $i$

Vediamo un algoritmo che adotta un approccio analogo al Quick Sort, ma che risolve ricorsivamente solo uno dei due sottoproblemi e utilizza una versione randomizzata.

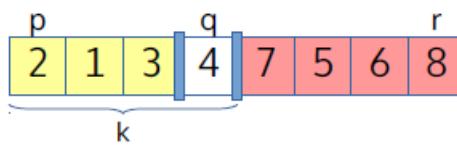
```
Partition (A,p,r)
x ← A[r]
i ← p-1
for j ← p to r-1
    do if A[j] ≤ x
        then i ← i+1
        exchange A[i]↔A[j]
```

$\Theta(n)$

```
exchange A[i+1] ↔ A[r]
return i+1
```

Adesso non ci interessa sapere se gli elementi sono ordinati ma soltanto selezionare uno specifico posto per ottenerne la statistica. Usiamo l'algoritmo di partition proprio grazie alla presenza del pivot, che sappiamo trovarsi proprio nel mezzo.

```
Randomized-Partition (A,p,r)
i ← Random (p,r)
exchange A[r] ↔ A[i]
return Partition (A,p,r)
```



```

RandomizedSelect (A, p, r, i)
if p=r
    then return A[p]
q ← RandomizedPartition (A, p, r)
k ← q-p+1
if i=k
    then return A[q]
elseif i<k
    then return RandomizedSelect (A, p, q-1, i)
else return RandomizedSelect (A, q+1, r, i-k)

```

La **RandomizedSelect** si basa sempre sul paradigma del *divide et impera* e opea confrontando l'ordine **i** con il numero **k** di elementi  $\leq$  del pivot, salva il pivot nella variabile **q** e si presenta tre casi:

- Se **i = k** vuol dire che la statistica di ordine **i** è **A[q]**.
  - Se **i < k** bisogna controllare nel sotto-array di sinistra
  - Se **i > k** bisogna controllare nel sotto-array di destra
- Anche questo algoritmo, come Quick sort non si occupa di riordinare il vettore.

Nel peggiore dei casi (quando il pivot è sempre il massimo) il tempo di esecuzione è  $\Theta(n^2)$ , dato dal tempo di esecuzione dei vari algoritmi di partition.

## Lezione 13 31/10/19

Calcoliamo ora un limite superiore per il **tempo atteso(medio)** di esecuzione. Consideriamo una variabile aleatoria indicatrice, cioè che assume solo valori 1 e 0 (quella di Bernulli). In particolare, consideriamo una variabile  $X_k$  che valga 1 se da **p** a **q** ci sono esattamente **k** elementi  $\leq$  del pivot e 0 altrimenti.

$$X_k = I\{A[p..q] \text{ ha } k \text{ elementi } \leq \text{ del pivot}\} \quad 1 \leq k \leq n$$

Poiché il pivot è scelto casualmente, le probabilità di avere **k** elementi  $\leq$  del pivot è uguale per tutti.

$$P\{X_i = 1\} = P\{X_j = 1\} \quad \forall i, j \rightarrow E[X_k] = \frac{1}{n}$$

Es:

Dato un vettore di 5 elementi così disposti: 1 3 3 3 6, qual è la probabilità di avere 4 elementi  $\leq$  di 3 e un elemento maggiore di 3? La probabilità sarà come visto  $\frac{1}{5}$  perché solo lo specifico 3 antecedente al 6 rispetta questa proprietà.

Per trovare un limite superiore, consideriamo il caso in cui bisogna controllare sempre il sotto-array più grande, cioè il **max( k-1, n-k )**. Chiaramente questo valore dipende da dove **k** si trova rispetto all metà.

$$\max(k-1, n-k) = \begin{cases} k-1 & \text{se più piccoli del pivot } k > \left\lceil \frac{n}{2} \right\rceil \\ n-k & \text{se più grandi del pivot } k \leq \left\lceil \frac{n}{2} \right\rceil \end{cases}$$

Questo è quindi il tempo per risolvere la partizione di dimensione maggiore, quando il pivot viene scelto come il kesimo elemento più piccolo

$$T(n) \leq \sum_{k=1}^n X_k [T(\max(k-1, n-k)) + O(n)] = \sum_{k=1}^n X_k T(\max(k-1, n-k)) + O(n)$$

Abbiamo scritto tale sommatoria perché così delle  $n$  variabili aleatorie di  $k$ , solo una sarà pari a 1, tutte le altre saranno 0. Ovviamente quella diversa sarà il pivot. Semplifichiamo l'espressione considerando che  $O(n)$  non dipende da  $k$  e lo consideriamo così una sola volta. Prendiamo quindi ora la **media** del limite superiore.

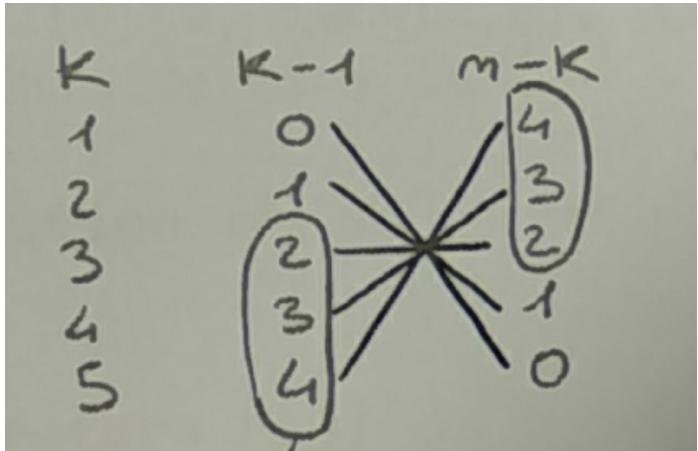
$$E[T(n)] \leq \sum_1^n E[X_k T(\max(k-1, n-k))] + O(n) = \sum_1^n \frac{E}{n} [X_k] * E[T(\max(k-1, n-k))] + O(n) =$$

e riconosciamo che  $X_k$  e  $T$  sono variabili aleatorie indipendenti perché nel primo caso dipende dalla scelta del pivot fatta durante il partizionamento, mentre per la seconda dipende dalle successive scelte del pivot.

$$= \sum_1^n \frac{1}{n} * E\left[T\left(\max(k-1, n-k)\right)\right] + O(n) = \text{facciamo altre osservazioni su questa quantità, in particolare sul massimo:}$$

La sommatoria di  $n$  termini con  $k$  che va da 1 a  $n$ , in ogni termine confrontiamo  $k-1$  con  $n-k$ . Vogliamo riscriverlo in maniera diversa (considerando un esempio).

Es:  $k = 5$ .



Leggendo orizzontalmente la riga avremmo che il massimo tra 0 e 4 è 4, tra 1 e 3 è 3 e così via, cerchiando il massimo tra gli elementi. Nel caso di  $n$  pari, abbiamo una sommatoria che magari si estende su 6 termini, ma solo tre sono diversi come in questo caso dove  $n$  è dispari. Possiamo quindi in generale dire che la sommatoria è pari:

$$\frac{2}{n} \sum_{\left[\frac{n}{2}\right]}^{n-1} E[T(k)] \text{ il termine } \left[\frac{n}{2}\right] \text{ vale solo se } n$$

**è dispari** poiché preferiamo includere due volte il numero condiviso piuttosto che escluderlo. Se

$n$  è pari consideriamo invece  $\left[\frac{n}{2}\right]$ , il 2 fuori dalla sommatoria è dovuto al fatto che ogni termine di  $T(k-1)$

compare due volte. Riprendiamo dunque la valutazione sul tempo di esecuzione.

$$= \frac{2}{n} \sum_{\left[\frac{n}{2}\right]}^{n-1} E[T(k)] + O(n) \leq \text{metodo di sostituzione (ovvero cerchiamo una soluzione candidata)}$$

$$\leq \frac{2}{n} \sum_{\left[\frac{n}{2}\right]}^{n-1} ck + an = \frac{2c}{n} \left( \sum_1^{n-1} k - \sum_1^{\left[\frac{n}{2}\right]-1} k \right) + an = \frac{2c}{n} \left( \frac{(n-1)n}{2} - \frac{\left(\left[\frac{n}{2}\right]-1\right) * \left[\frac{n}{2}\right]}{2} \right) + an \leq$$

$$\text{dimostriamo con il passo induttivo la nostra ipotesi .} \leq \frac{2c}{n} \left( \frac{(n-1)n}{2} - \frac{\left(\frac{n}{2}-2\right)\left(\frac{n}{2}-1\right)}{2} \right) + an =$$

$$= \frac{2c}{n} \left( \frac{n^2 - n}{2} - \frac{\frac{n^2}{4} - \frac{3n}{2} + 2}{2} \right) + an = \frac{c}{n} \left( \frac{3n^2}{4} + \frac{n}{2} - 2 \right) + ac = c \left( \frac{3n}{4} + \frac{1}{2} - \frac{2}{n} \right) + an \leq cn - \left( \frac{cn}{4} - \frac{c}{2} - an \right) \leq cn$$

Dobbiamo mostrare che per  $n$  sufficientemente grande:

$$cn - \left( \frac{cn}{4} - \frac{c}{2} - an \right) \leq cn \rightarrow n \left( \frac{c}{4} - a \right) \geq \frac{c}{2} \text{ se scegliamo } c \text{ tale che} \\ \frac{c}{4} - a > 0 \text{ avremo che } n \geq \frac{\frac{c}{2}}{\frac{c}{4} - a} = \frac{2c}{c - 4a}$$

Se assumiamo che  $T(n) = O(1)$  per  $n < \frac{2c}{c - 4a}$  abbiamo che  $E[T(n)] = O(n)$ . Nonostante il tempo di esecuzione **atteso sia lineare**, ciò non implica che l'algoritmo non possa impiegare tempo maggiore, infatti abbiamo visto che nel caso peggiore è  $\Theta(n^2)$ .

## Lezione 14 06/11/19

### Selezione in tempo lineare

#### Selezione in tempo lineare

Sebbene abbiam visto che il tempo d'esecuzione atteso sia lineare, ciò non implica che l'algoritmo non possa impiegare un tempo maggiore, infatti abbiamo visto che nel caso peggiore è  $\Theta(n^2)$ . Consideriamo dunque un algoritmo che sia lineare anche nel caso peggiore (Sarà anche l'ultimo algoritmo del corso, successivamente si passerà alla seconda parte del corso, le strutture dati) :

```
Partition (A,p,r)
x ← A[r]
i ← p-1
for j ← p to r-1
    do if A[j] ≤ x
        then i ← i+1
        exchange A[i]↔A[j]
exchange A[i+1] ↔ A[r]
return i+1
```

$\Theta(n)$

Dato un vettore per il quale dobbiamo individuare la statistica dell'ordine *i*esimo individuiamo il pivot che ci permette di trovare il sottoproblema sul quale reinvocare la funzione. Idealmente vorremmo un pivot che divida esattamente a metà il vettore, per questo usiamo la funzione Select, il valore che ci fornisce è quello chiamato **mediana**.

Paradossalmente ottenere la mediana significa proprio risolvere il problema che ci siamo posti, quindi non è pensabile il risolvere un problema usando il problema stesso. Decidiamo quindi di ottenere una mediana approssimata.

```
M-Partition (A,p,r,m)
exchange A[r] ↔ A[m]
return Partition (A,p,r)
```

Quindi la prima differenza sta nel fatto che il pivot non è scelto a caso, ma in modo da avere un partizionamento bilanciato. Per scegliere un **m** che bilanci partizionamento basta scegliere la

**mediana(inferiore)**, ovvero la statistica di ordine  $\left\lfloor \left( \frac{n+1}{2} \right) \right\rfloor$ .

In realtà, per avere un costo minore, non scegliamo la mediana del vettore intero, ma quella di un **vettore ristretto**. Suddividiamo quindi il vettore in gruppi da **5 elementi**, dove 5 è scelto perché dispari e abbastanza piccolo, e ordiniamo i gruppi con Insertion Sort. La mediana di ogni gruppo è l'elemento centrale (cioè in posizione 3).

```

Select (A,p,r,i)
Raggruppa gli n=r-p+1 elementi in gruppi da 5
Ordina con Insertion Sort ciascun gruppo
Se c'è un solo gruppo restituisci A[i]
L'elemento centrale di ogni gruppo è la mediana
Sia B il vettore delle mediane dei gruppi
x ← Select (B,1,length[B],|(length[B]+1)/2|)
Sia m l'indice di x in A
q ← M-Partition (A,p,r,m)
k ← q-p+1
if i=k
    then return A[q]
elseif i<k
    then return Select (A,p,q-1,i)
else return Select (A,q+1,r,i-k)

```

Caso base

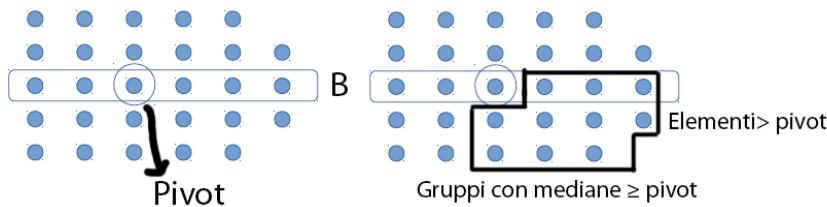
(Uguale a partition)

A questo punto, consideriamo come **vettore ristretto** il vettore delle mediane dei gruppi, che ha dimensione  $\left\lceil \frac{n}{5} \right\rceil$ , e assegniamo ad **m** la mediana di questo vettore.

Dunque, **riassumendo**:

- Divido **A** in gruppi da 5.
- Ordino ogni gruppo con Insertion Sort
- Trovo il vettore delle mediane **B**.
- Calcolo ricorsivamente la mediana delle mediane.

Es: Un vettore di 28 elementi, dividiamolo in gruppi di al più 5 elementi per ogni gruppo calcoliamo la mediana. Troviamo ora la mediana delle mediane che utilizzeremo come pivot.



Possiamo notare che gli elementi sicuramente **maggiori** del pivot, sono quelli maggiori delle mediane **maggiori o uguali** del pivot. Quindi i gruppi che hanno 3 elementi > del pivot sono **almeno la metà** degli  $\left\lceil \frac{n}{5} \right\rceil$  gruppi, tranne 2:

- Il gruppo contenente il pivot.
- Il gruppo finale, che potrebbe avere meno di 5 elementi.

Quindi il numero minimo degli elementi > del pivot è:

$$3 \left( \left\lceil \frac{1}{2} \left\lceil \frac{n}{5} \right\rceil \right\rceil - 2 \right) \geq \frac{3}{10}n - 6 \text{ da questo perveniamo che il sottoproblema da implementare ha dimensione pari a : } n - \left( \frac{3}{10}n - 6 \right) = \frac{7}{10}n + 6.$$

Dunque, avremo due sottoproblemi di dimensione:  $\frac{3}{10}n - 6$   
 $\frac{7}{10}n + 6$  caso peggiore

Perveniamo quindi al tempo di esecuzione del Select come:

$$T(n) \leq \begin{cases} O(1) & \text{se } n < 140 \\ T\left(\left\lceil \frac{n}{5} \right\rceil\right) + T\left(\frac{7}{10}n + 6\right) + O(n) & \text{se } n \geq 140 \end{cases}$$

Nel secondo termine  $T\left(\left\lceil \frac{n}{5} \right\rceil\right)$  è il tempo per individuare la mediana delle mediane, più il tempo per reinvocare l'algoritmo sulla partizione più grande ed infine il tempo della partition.

### Applichiamo il metodo di sostituzione

$$\begin{aligned} T(n) &\leq c \left\lceil \frac{n}{5} \right\rceil + c\left(\frac{7}{10}n + 6\right) + an \leq c \left(\frac{n}{5} + 1\right) + c\left(\frac{7}{10}n + 6\right) + an = \frac{9}{10}cn + 7c + an = \\ &= cn + \left(-\frac{cn}{10} + 7c + an\right) \leq cn \end{aligned}$$

$$\text{Se } -\frac{cn}{10} + 7c + an \leq 0 \rightarrow \frac{cn - 70c}{10} \geq an \rightarrow c \geq 10 * \frac{an}{n - 70} \text{ per } n > 70.$$

$T(n) = O(n)$  in particolare abbiamo la condizione  $n \geq 140 \rightarrow c \geq 20a$ . La scelta di 140 è puramente casuale e al solo fine d'esempio.

Gli algoritmi di selezione non sono soggetti al vincolo sul tempo d'esecuzione nel caso peggiore  $O(n \lg n)$  perché non effettuano confronti. A differenza degli algoritmi di ordinamento "lineari" gli algoritmi di selezione non fanno ipotesi sulle **caratteristiche dei valori di ingresso**.

## Strutture dati

### Strutture dati-Definizione

Una **struttura dati** è una modalità di organizzazione e memorizzazione dei dati. Le operazioni sui dati vengono effettuate in un modo dipendente dalla specifica struttura dati scelta, in base al tipo di operazioni che dobbiamo fare sceglieremo l'opportuna struttura dati. Tipicamente una struttura dati ha a che fare con **insiemi dinamici** di dati, ossia gli elementi cambiano nel tempo.

Si dice **dizionario** un insieme dinamico che supporta le operazioni di:

- Inserimento di un elemento.
- Eliminazione di un elemento.
- Verifica dell'appartenenza di un elemento all'insieme.

Gli elementi dinamici in questione sono in genere costituiti da dati strutturati, che possono essere suddivisi in:

- Elemento **chiave**.
- Dati **satellite**.

In alcuni casi, si presuppone l'esistenza di una relazione d'ordine totale tra le chiavi degli elementi.

Alcune operazioni tipiche sono:

<b>Search(S, k)</b>	Restituisce un puntatore all'elemento di chiave k
<b>Minimum(S)</b>	Restituisce un puntatore all'elemento con chiave più piccola
<b>Maximum(S)</b>	Restituisce un puntatore all'elemento con chiave più grande
<b>Successor(S, x)</b>	Restituisce un puntatore all'elemento con la chiave immediatamente maggiore
<b>Predecessor(S, x)</b>	Restituisce un puntatore all'elemento con la chiave immediatamente minore
<b>Insert(S, x)</b>	Inserisce l'elemento puntato da x
<b>Delete(S, x)</b>	Rimuove l'elemento puntato da x

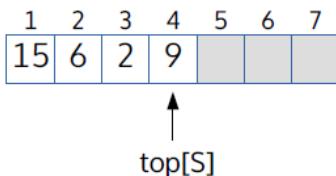
Le prime 5 operazioni sono di query, mentre le ultime due sono operazioni che modificano l'insieme.

## Stack

Uno **stack** o (**pila**) è una struttura dati che adotta una strategia **LIFO** (Last In Last Out), ovvero:

- La *insert* (detta **push**) inserisce in testa.
- La *delete* (detta **pop**) elimina in testa.

Uno stack di **n** elementi può essere implementato tramite un array lungo **n** con un attributo **top** che indica l'ultimo elemento inserito (0 se vuoto).



```
Stack-Empty (S)
if top[S]=0
    then return TRUE
    else return FALSE
```

```
Push (S,x)
top[S] ← top[S] + 1
S[top[S]] ← x
```

```
Pop (S)
if Stack-Empty(S)
    then error "underflow"
    else top[S] ← top[S]-1
        return S[top[S]+1]
```

Sono tutte operazioni che richiedono un tempo d'esecuzione pari a O(1). Questo significa che nell'ipotetico caso dovessimo gestire dei dati in maniera LIFO o come vedremo FIFO, non ha nessun senso usare alberi o altre strutture più complicate quando è possibile usare array che implementano questo tipo di strutture.

## Code

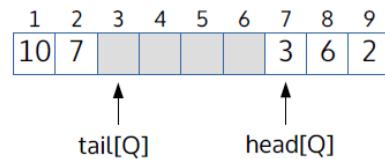
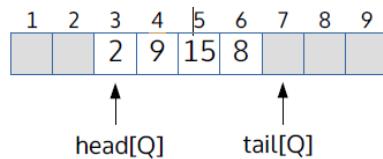
Uno **coda** adotta una strategia **FIFO** (First In First Out) , ovvero:

- La *insert* (detta **enqueue**) inserisce in coda.
- La *delete* (detta **dequeue**) elimina in testa.

Una coda di **n** elementi può essere implementato tramite un array lungo **n-1** con:

- Un attributo **head** che indica la posizione della testa (l'elemento meno recente).
- Un attributo **tail** che indica la posizione di coda o la posizione in cui inserire il nuovo elemento.

- Coda vuota se  $\text{head} = \text{tail}$ , piena se  $\text{head} = \text{tail} + 1$ .



Enqueue ( $Q, x$ )

```
Q[tail[Q]] ← x
if tail[Q] = length[Q]
    then tail[Q] ← 1
else tail[Q] ← tail[Q]+1
```

Dequeue ( $Q$ )

```
x ← Q[head[Q]]
if head[Q] = length[Q]
    then head[Q] ← 1
else head[Q] ← head[Q]+1
return x
```

Sono tutte operazioni O(1). Abbiamo trascurato la verifica dell'overflow e dell'underflow (ovvero aggiungere un nuovo elemento se la pila è già piena). Se volessimo inserire anche l'**nesimo** elemento la condizione di coda piena sarebbe uguale a quella di coda buona, quindi non si distinguerebbero.

## Liste concatenate

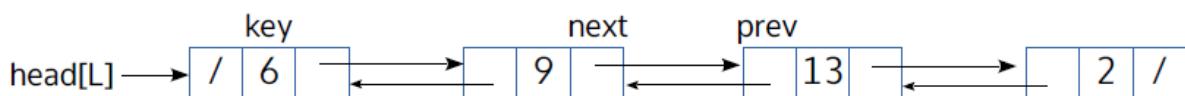
Una **lista concatenata** è una struttura dati in cui gli elementi sono disposti secondo un ordine lineare determinato da dei **puntatori**, non sono dunque quindi ordinati per forza consecutivamente contrariamente agli array. Supportano tutte le operazioni indicate per gli insiemi dinamici, anche se non in maniera efficiente. Una lista può essere:

- Singolarmente concatenata.
- Doppialmente concatenata.
- Ordinata.
- Circolare.

Le liste usano:

- Un attributo **head** che punta al primo nodo.
- Un puntatore **next** che punta al nodo successivo.
- Un puntatore **prev** che punta al nodo precedente.

Sono quindi un singolo valore e due puntatori. Se la coda **non è circolare** allora il puntatore prev al primo nodo e il puntatore next all'ultimo nodo puntano entrambi a NIL.



## Lezione 15 07/11/19

La **ricerca** avviene confrontando la chiave di ogni nodo con una chiave richiesta  $k$ . Nel caso in cui l'elemento con chiave  $k$  non sia presente, restituisce NIL.

```

List-Search (L,k)
x ← head[L]
while x≠NIL and key[x]≠k
    do x ← next[x]
return x

```

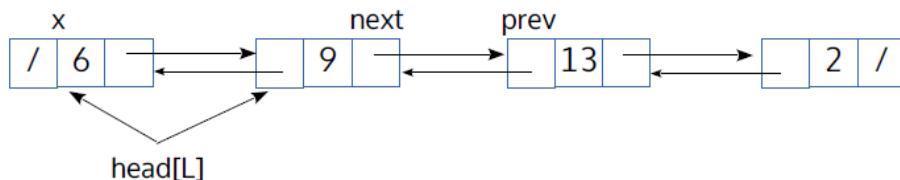
In una lista concatenata il nostro unico riferimento è il primo elemento, se dobbiamo accedere al successivo useremo il campo **next** che punta al prossimo indirizzo di memoria. Nel ciclo **while** viene sfruttata la condizione di **corto circuito**, ovvero si mettono le condizioni in modo tale che se la prima è falsa si esce senza accedere alla seconda che può provocare enormi errori. Nel nostro caso se si fosse acceduto al valore NIL vi sarebbe stato un errore di segmentation fault con conseguente crash del sistema. Poiché nel caso peggiore l'algoritmo effettua **n** confronti, ha complessità  $\Theta(n)$ . Mentre genericamente ha una complessità  $O(n)$

**L'inserimento** avviene tipicamente in testa alla lista, assegnando al campo **prev** della testa il puntatore al nuovo nodo e al campo **next** del nuovo nodo il puntatore alla testa. Mentre il campo **prev** del nuovo valore **x** punta a NIL, e il campo **head** ora punta a **x**. È importante rispettare l'ordine delle operazioni, poiché potremmo perdere i riferimenti alla lista. Richiede un tempo costante **O(1)**.

```

List-Insert (L,x)
next[x] ← head[L]
if head[L]≠NIL
    then prev[head[L]] ← x
head[L] ← x
prev[x] ← NIL

```



Come è possibile vedere le liste concatenate rendono molto semplici le operazioni di inserimento ed eliminazione di elementi, questo perché è possibile lavorare direttamente sui puntatori.

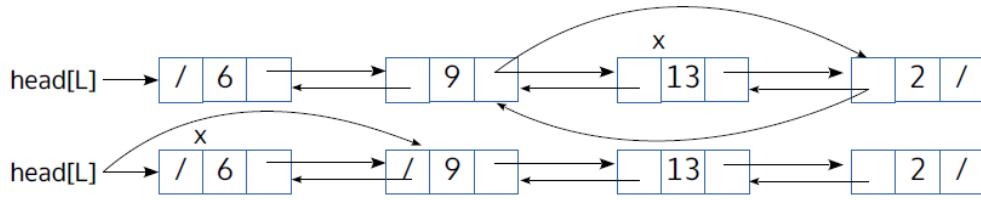
**L'eliminazione** di un elemento è tipicamente preceduta da una ricerca la quale restituisce il puntatore all'elemento da eliminare. L'elemento eliminato è in realtà bypassato, assegnando il suo **next** al precedente e il suo **prev** al successivo.

Richiede un tempo costante **O(1)**, soltanto se è una lista doppiamente concatenata, nel caso peggiore **O(n)**.

```

List-Delete (L,x)
if prev[x]≠NIL
    then next[prev[x]] ← next[x]
    else head[L] ← next[x]
if next[x]≠NIL
    then prev[next[x]] ← prev[x]

```



Rispetto allo pseudocodice ci dovrebbe essere prima di tutto un controllo sulla effettiva presenza del valore  $x$ . L'immagine va letta in questo modo: Il primo passaggio è spostare next di 9 al prev di 2, pooi il campo prev di 2 deve puntare a next di 9. Facendo così le informazioni a 13 vanno perdute e lui è eliminato

## Lista concatenata con sentinella

Il codice può essere semplificato aggiungendo un oggetto vuoto, detto **sentinella**, i cui campi:

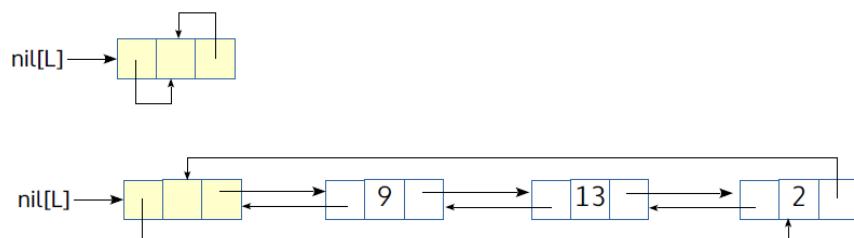
- **next** punta alla testa.
- **prev** punta alla coda.

Aggiungendo questo oggetto la lista diventa circolare. Viene inoltre aggiunto un attributo **nil**, che punta alla sentinella.

Quindi:

- **nil** sostituisce **NIL**.
- **next[nil]** sostituisce **head**.

Rendendo il codice più leggibile.

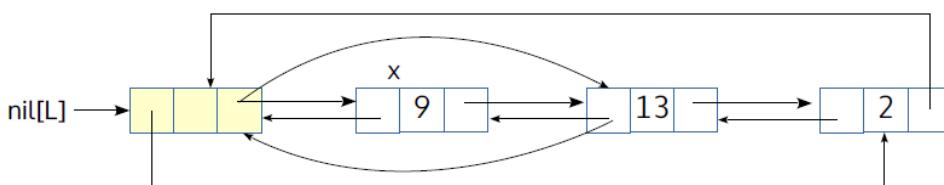


La lista, come detto, diventa quindi **circolare** dunque esiste sempre un nodo precedente e uno successivo e il codice è semplificato nel seguente modo:

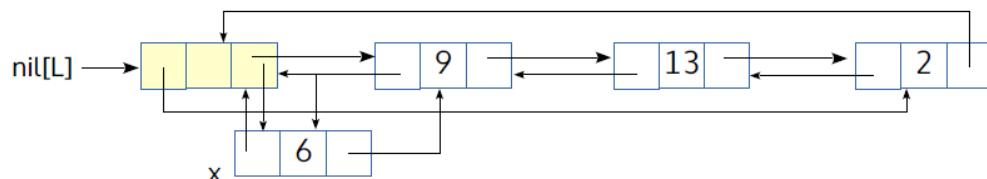
Senza sentinella	Con sentinella
<u>List-Search</u> ( $L, k$ ) $x \leftarrow \text{head}[L]$ $\text{while } x \neq \text{NIL} \text{ and } \text{key}[x] \neq k$ $\quad \text{do } x \leftarrow \text{next}[x]$ $\text{return } x$	<u>List-Search'</u> ( $L, k$ ) $x \leftarrow \text{next}[nil[L]]$ $\text{while } x \neq nil[L] \text{ and } \text{key}[x] \neq k$ $\quad \text{do } x \leftarrow \text{next}[x]$ $\text{return } x$

<pre><u>List-Insert</u> (L,x) next[x] ← head[L] if head[L]≠NIL     then prev[head[L]] ← x head[L] ← x prev[x] ← NIL</pre>	<pre><u>List-Insert'</u> (L,x) next[x] ← next[nil[L]] prev[next[nil[[L]]]] ← x next[nil[[L]]] ← x prev[x] ← nil[L]</pre>
<pre><u>List-Delete</u> (L,x) if prev[x]≠NIL     then next[prev[x]] ← next[x]     else head[L] ← next[x] if next[x]≠NIL     then prev[next[x]] ← prev[x]</pre>	<pre><u>List-Delete'</u> (L,x) next[prev[x]] ← next[x] prev[next[x]] ← prev[x]</pre>

Esempio di cancellazione con sentinella:



Esempio di inserimento con sentinella



Con l'aggiunta di un solo nodo la lettura del codice migliora di molto, quindi il tradeoff è più che giustificabile. Ora mostriremo un piccolo confronto tra **liste concatenate ed array**.

I vettori assumono che i dati vengano salvati in zone **contigue di memoria** e il loro accesso è diretto e quindi hanno un tempo  $O(1)$ , questo perché si calcola subito dove si trova l'elemento. Mentre nelle liste concatenate questo accesso diretto non è possibile poiché sono costretti a scorrerla fino all'elemento desiderato.

Valutiamo le varie operazioni effettuabili dunque, considerando però che gli elementi nelle due diverse strutture non sono ordinati:

- Ricerca in un array  $O(n)$ . Ricerca in una lista  $O(n)$ .
- Inserimento in una posizione generica in un array, a causa dello scorrimento  $O(n)$ . Inserimento in una posizione generica in una lista  $O(1)$ .
- Cancellazione di un determinato elemento in un array  $O(n)$ , analogamente a prima. Cancellazione in una lista  $O(1)$ .

Ovviamente questa valutazione è molto generica, dipende ovviamente dal tipo di struttura dati che utilizziamo, come abbiamo potuto vedere nella **pila il tempo era O(1)**.

## Tabelle Hash

### Tabelle ad indirizzamento diretto

Le **tabelle ad indirizzamento diretto** hanno come obiettivo quello di mantenere un insieme dinamico di dati, sono strutture dati che possono essere utilizzate quando le chiavi sono distinte e possono assumere come valore l'insieme  $\{0, 1, \dots, m-1\}$ , con  $m$  non troppo grande. Viene quindi usato un array lungo  $m$  in cui nella posizione di indice  $k$  è memorizzata la chiave  $k$ , o **NIL** se la chiave non è presente. Ovviamente questa struttura dati è possibile solo se le chiavi **sono tutte distinte**, conviene poi usare la seguente struttura quando la cardinalità dell'insieme dei valori non è molto grande rispetto al numero di elementi, che dobbiamo tipicamente mantenere nella nostra struttura dati.

Direct-Address-Search ( $T, k$ )

```
return  $T[k]$ 
```

Direct-Address-Insert ( $T, x$ )

```
 $T[key[x]] \leftarrow x$ 
```

Direct-Address-Delete ( $T, x$ )

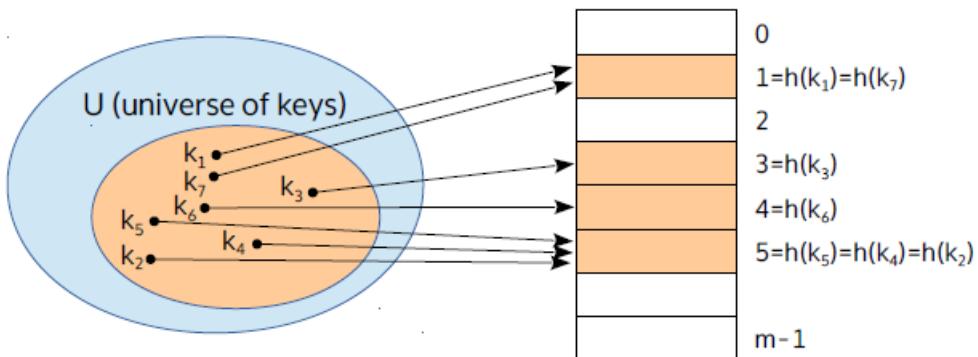
```
 $T[key[x]] \leftarrow NIL$ 
```

Il vantaggio di queste tabelle è che ogni operazione è **O(1)**. Lo svantaggio è che se  $m$  è grande rispetto al numero di elementi da memorizzare, si ha un notevole spreco di memoria. Ipotizziamo che  **$m$  sia minore della cardinalità dell'insieme di elementi da mantenere**. La funzione **non sarà iniettiva** perchè capiterà che coppie di elementi con chiavi distinte verranno mappate su uno stesso indice. Il vettore in posizione  $i$  presenterà inizi di liste concatenate. Ciò migliora il consumo di memoria perchè nonostante la cardinalità dell'insieme di elementi che voglio memorizzare sia alta, spesso il numero effettivo di valori che contemporaneamente voglio mantenere non è altrettanto alto, e con le tabelle ad indirizzamento diretto allocherei molta più memoria di quanta viene poi usata.

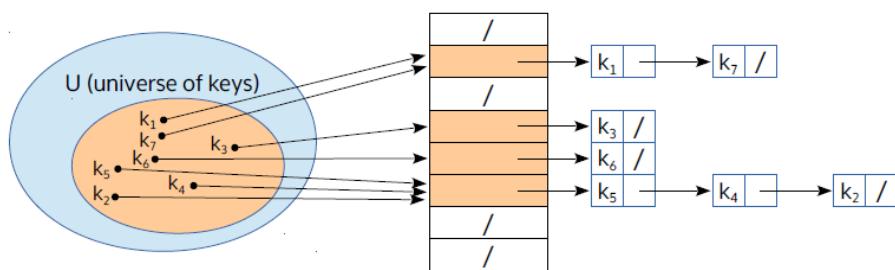
### Tabelle hash

Le **tabelle hash** permettono di considerare un numero di chiavi da memorizzare anche molto inferiore rispetto alla cardinalità dell'insieme delle possibili chiavi. Le tabelle di hash mappano le chiavi usando **non una semplice funzione di identità**, ma con una funzione iniettiva detta appunto di hash, indicata con  $h$ .

$$h: U \rightarrow \{0, 1, \dots, m - 1\}$$



Se la funzione hash produce lo stesso risultato per chiavi diverse, si parla di **collisione**. Nonostante sia impossibile evitare le collisioni (quando ad esempio il numero di chiavi è maggiore della cardinalità di  $U$ ), è però possibile ridurne la probabilità. Per gestire le collisioni si possono usare delle liste concatenate in cui vengono memorizzate le chiavi che collidono in una stessa casella.



Ogni casella è una lista concatenata che mantiene tutti gli elementi fatti corrispondere ad una determinata chiave dalla funzione hash. La **tecnica di concatenazione** (chaining) prevede di memorizzare tutti gli elementi che collidono in una lista concatenata.

## Lezione 16 13/11/19

Chained-Hash-Insert ( $T, k$ )  
insert  $x$  at the head of list  $T[h(key[x])]$

Chained-Hash-Search ( $T, k$ )  
search for an element with key  $k$  in list  $T[h(k)]$

Chained-Hash-Delete ( $T, x$ )  
delete  $x$  from the list  $T[h(key[x])]$

L'**inserimento** richiede di calcolare il tempo della funzione hash sulla chiave dell'elemento da inserire, ha un tempo di  **$O(1)$** , anche l'operazione di **eliminazione** richiede un tempo di  **$O(1)$**  ma solo se sono liste doppiamente concatenate. Infatti, se usassimo liste singolarmente concatenate, anche sapendo dove si trova l'elemento da eliminare grazie al puntatore  $x$ , dovremmo comunque scorrere la lista per accedere all'elemento precedente al fine di modificare il suo campo **next**. In questo caso l'eliminazione richiederebbe un tempo  $\theta(n_j)$  nel caso peggiore, dove  $n_j$  è il numero di elementi di una singola lista. Entrambe le due operazioni richiedono prima una ricerca, per questo la valuteremo separatamente.

### Tempo di esecuzione della ricerca

La **ricerca** richiede  $\Theta(n)$  nel caso peggiore, ossia il caso in cui tutti gli  $n$  elementi siano collocati nelle stessa lista. In generale, le prestazioni dipendono da come la funzione hash distribuisce gli elementi tra le liste. Supponiamo che ogni elemento abbia la stessa probabilità di capitare in ogni lista (*simple uniform hashing*).

Se la lista **jesima** lunga  $\mathbf{n}$ ,  $\mathbf{n} = n_0 + n_1 + \dots + n_{m+1}$  è la lunghezza totale ed  $\mathbf{m}$  il numero di liste consideriamo la probabilità del **fattore di carico**, ovvero quanti elementi vi sono mediamente nella lista linkata.

$$E[n_j] = \frac{n}{m} = \alpha \leftarrow \text{Lunghezza media.}$$

Calcoliamo dunque il numero atteso di elementi esaminati dalla funzione ricerca:

- Caso 1: La ricerca di un elemento con chiave  $k$  non ha successo

In questo caso il tempo impiegato è pari alla dimensione media  $\alpha$  della lista  $\mathbf{h(k)}$  più il tempo della funzione hash. Quindi  $\Theta(1 + \alpha)$ . Una chiave non presente nella tabella ha una uguale probabilità di essere associata ad una delle  $m$  liste, l'uno serve per considerare il tempo di calcolo dell'hash.

- Caso 2: La ricerca ha successo

In questo caso il numero di elementi esaminati è almeno 1 più il numero di elementi che precedono  $x_i$  che precedono  $x_i$  nella lista. Questi elementi  $x_j$ :

- Hanno  $i + 1 \leq j \leq n$  perché gli  $x_j$  sono stati inseriti successivamente.
- Hanno  $h(k_j) = h(k_i)$  perché sono sulla stessa lista concatenata, questo accade la funzione hash restituisce lo stesso valore.

Possiamo quindi considerare una variabile aleatoria indicatrice che vale 1 quando l'elemento  $x_j$  è nella lista  $x_i$ .

$$X_{ij} = I\left\{h(k_i) = h(k_j)\right\} \rightarrow E[x_{ij}] = \frac{m}{m^2} = \frac{1}{m} \text{ probabilità che i valori vengano mappati sulla stessa lista.}$$

Il numero atteso di elementi esaminati è:

$$\begin{aligned} E\left[\frac{1}{n} \sum_{i=1}^n \left(i + \sum_{j=i+1}^n x_{ij}\right)\right] &= \frac{1}{n} \sum_{i=1}^n \left(1 + \sum_{j=i+1}^n E[x_{ij}]\right) = \frac{1}{n} \sum_{i=1}^n \left(1 + \sum_{j=i+1}^n \frac{1}{m}\right) = \frac{1}{n} \sum_{i=1}^n 1 + \frac{1}{nm} \sum_{i=1}^n \sum_{j=i+1}^n 1 \\ 1 + \frac{1}{nm} \sum_{i=1}^n (n - i) &= 1 + \frac{1}{nm} \left(n \sum_{i=1}^n 1 - \sum_{i=1}^n i\right) = 1 + \frac{1}{nm} \left(n^2 - \frac{n(n-1)}{2}\right) = 1 + \frac{n}{m} * \frac{1}{2} - \frac{1}{2m} = \\ 1 + \frac{\alpha}{2} - \frac{\alpha}{2n} &= \theta(1 + \alpha) \end{aligned}$$

Quindi anche in questo caso si ha  $\Theta(1 + \alpha)$ .

Notiamo però che se  $\mathbf{m}$  è proporzionale ad  $\mathbf{n}$ , cioè se al crescere del numero di elementi cresce anche il numero di liste in maniera proporzionale, la lunghezza media delle liste diventa fissa. Quindi, in questo caso,

bisogna ricercare un elemento in liste la cui dimensione è fissa, a prescindere da quanto sia grande  $n$ .  
Analiticamente:

$$m = O(n) \rightarrow \theta(1 + \alpha) = \theta\left(1 + \frac{n}{m}\right) = \theta\left(1 + \frac{n}{O(n)}\right) = O(1).$$

Si ha quindi un tempo costante nel caso di *simple uniform hashing*. In realtà, tale proprietà di distribuzione uniforme è difficile da ottenere, poiché in generale la distribuzione di probabilità delle chiavi non è nota a priori. Si cerca comunque di definire delle funzioni hash che abbiano buone prestazioni anche quando tale distribuzione non è nota.

## Funzioni hash

Vogliamo quindi adesso fare in modo di essere indipendenti dalla probabilità di distribuzione, questo perché non sappiamo sempre la distribuzione. Le **funzioni hash** lavorano con i numeri naturali, nel caso in cui le chiavi non siano numeri naturali, possono comunque essere convertite in numeri naturali. Ad esempio la stringa "pt" può essere espressa come: p = 112 e t = 116 nel codice ASCII 7-bit, possiamo vedere i numeri in base 128, per arrivare a  $112*128^0+116*128^1 = 14452$ .

## Metodo della divisione

Valutiamo questo tipo di funzione hash  $h$ , dove prende in ingresso un intero positivo e restituisce il valore in modulo  $m$  di  $k$ .

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

Restituisce valori in  $[0, \dots, m-1]$ . Questo metodo è molto veloce, poiché richiede solo una semplice divisione. Si cerca, però, di evitare alcuni valori di  $m$ , ad esempio le potenze di 2. In questo caso, infatti il modulo per  $2^p$  dipende solo dai  $p$  bit meno significativi della chiave  $k$ , mentre in genere è meglio se  $h(k)$  ha i valori sparpagliati e che dipendono da tutti i bit.

Un buon valore di  $m$  è un numero primo abbastanza vicino al numero di slot desiderato, ma non troppo vicino ad una potenza di 2.

Alcune funzioni hash vedremo, e tutte devono restituire una chiave compreso tra 0 e  $n-1$  cardinalità della tabella.

## Metodo della moltiplicazione

Definisce la funzione differentemente da quella precedente prendendo  $0 < A < 1$  costante e si fa:

$$h(k) = \lfloor m * (kA - \lfloor kA \rfloor) \rfloor$$

La parte  $kA - \lfloor kA \rfloor$  toglie da  $kA$  la sua parte intera cioè preleva la sua parte decimale. Dunque stiamo prendendo la parte intera del prodotto di  $m$  per la parte decimale di  $kA$ , che è un valore compreso tra 0 e 1. Quindi anche in questo caso abbiamo valori in  $[0, \dots, m-1]$ .

Una implementazione efficiente si ottiene con:

$$m = 2^p$$

$$A = \frac{s}{2^w} \text{ con } 2^w > s \in N \text{ w è il numero di bit di una word(in cui entra una chiave)}$$

Es.

$$w=4 \text{ bits}, k=6, p=3, s=9 \rightarrow A = \frac{9}{10} = 0.5625$$

	0110    k=6
	* 1001    s=9
k*s	110110
k*A=(k*s)/2^w	11,0110
Parte decimale	0,0110
Molt. per m=2^p	11,0
Parte intera	11    h(6)=3

Questo metodo è più efficiente anche del metodo della divisione se messi nelle stesse ipotesi. Il risultato è poi espresso in **p** bit.

## Hashing universale

Nel caso di funzioni deterministiche, definiamo una famiglia di funzioni di hash che come i metodi di divisione e moltiplicazione, viene usato per evitare a priori i casi di collisioni. Quindi possono esserci delle specifiche sequenze di chiavi che generano tutte collisioni, quindi ricadono nel caso peggiore. Per il funzionamento dell'hashing universale abbiamo già trattato un qualcosa si simile quando abbiamo parlato di randomizzare gli algoritmi (ad esempio il random quicksort), questo perché rende non individuabile a priori una sequenza per la quale l'algoritmo finisce nel tempo di caso peggiore.

Possiamo allora pensare di introdurre l'aleatorietà nella scelta della funzione hash (solo nella scelta, non nella funzione stessa, altrimenti non potremmo ritrovare un elemento inserito).

La tecnica dell'**hashing universale** sceglie casualmente una funzione hash da una classe di funzioni (ma solo all'inizio dell'esecuzione), in questo modo, nessuna sequenza di chiavi può cadere sempre nel caso peggiore.

Se **H** è la classe di funzioni **h**, tale classe è *universale* se per ogni coppia di chiavi distinte, il numero di

funzioni che collidono è al massimo  $\frac{|H|}{m}$  (cioè se solo la frazione  $\frac{1}{m}$  delle funzioni della classe genera collisione).

$$h: U \rightarrow \{0, \dots, m-1\}$$

$H = \{h_i\}$   $H$  è **universale** se  $\forall k, l \in U, k \neq l$ , chiavi distinte,  $h(k) = h(l)$  al massimo  $\frac{|H|}{m}$  volte dove il modulo indica la cardinalità.

In questo caso, la probabilità di collisione è  $\frac{1}{m}$ .

Es:

Sia  $p$  un numero primo sufficientemente grande da assicurare che ogni chiave  $k$  sia all'interno dell'intervallo  $[0, p-1]$ .

$$k \in z_p \quad z_p = \{0, \dots, p-1\}, \quad z_p^* = z_p - \{0\}.$$

$$h_{a,b}(k) = ((aK + b) \bmod p) \bmod m \text{ con } a \in z_p^*, b \in z_p$$

Ci sono **p(p-1)** funzioni di questa classe ed **m** non deve essere necessariamente essere un numero primo.

## Tabelle ad indirizzamento aperto

Le **tabelle ad indirizzamento aperto** utilizzano anch'esse un array di **m** elementi. A differenza delle tabelle di hash tradizionali, non usano liste concatenate, quindi possono memorizzare al più **m** elementi, come le tabelle ad indirizzamento diretto.

Per evitare le collisioni, ogni chiave può essere inserita in un insieme di posizioni possibili, in modo tale che se la posizione risulta occupata viene effettuato un altro tentativo in una seconda cella. Si utilizza quindi una funzione hash che è funzione sia della chiave **k** che del tentativo **i**.

$$h(k, i) : U * \{0, 1, \dots, m - 1\} \rightarrow \{0, 1, \dots, m - 1\}$$

La sequenza di ricerca di una posizione libera è detta **sequenza di probing**:

$$< h(k, 0), \dots, h(k, m - 1) >$$

E, al fine di coprire tutta la tabella, **deve essere una permutazione di**  $<0, \dots, m - 1> \forall k$ . Ho quindi  $m!$  permutazioni possibili ma me ne servono solo  $m$ . Questa tabella può essere usata quando però abbiamo una stima degli elementi che utilizzeremo, perché se superiamo il limite non ne potremmo inserire più.

## Linear probing

È una tecnica per gestire una tabella di indirizzamento aperto,  $h(k, i) = (h'(k) + i) \bmod m$ .

Consideriamo una funzione hash  $h'(k)$ , se la posizione è occupata, cerca la prima posizione libera successiva, il **mod m** serve per rendere la ricerca circolare, cioè se anche l'ultima casella è occupata cerca a partire dalla casella **0**. **i** indica i tentativi, che sono al massimo  $m$ . Le posizioni che vengono provate sono  $h'(k)$ ,  $h'(k)+1$ ,  $h'(k)+2, \dots, m-1, 0, 1, 2, \dots, h'(k)-1$ .

Es:

0
1 79
2
3
4 69
5 98
43
6
7 72
8
9
10
11 50
12

$$h'(k) = k \bmod 13$$

$$k=43 = h'(43) = 43 \bmod 13 = 4$$

$$(4+0) \bmod 13 = 4 \text{ occupato}$$

$$(4+1) \bmod 13 = 5 \text{ occupato}$$

$$(4+2) \bmod 13 = 6 \text{ libero}$$

Viene fatta la divisione di 43 per 13 e si recupera il resto, 4. La prima posizione quindi sarà 4+0 diviso 13 ha sempre resto 4 che risulta occupato e così via.

Una volta individuato il primo tentativo, le altre saranno fatte ricorsivamente.

## Lezione 17 14/11/19

## Quadratic probing

$$h(k, i) = (h'(k) + c_1 i + c_2 i^2) \bmod m$$

Le successive posizioni sono separate da un offset che cresce quadraticamente con il tentativo. Nel linear probing i successivi tentativi, l'offset è pari a uno, in questo caso invece aumenta seguendo una legge quadratica e la scelta di **c<sub>1</sub>** e **c<sub>2</sub>** è fatta in modo tale da coprire tutta la tabella durante la ricerca. Questa tecnica permette di migliorare l'efficienza nel caso in cui la funzione hash tende a mappare le chiavi in locazioni consecutive (in questo caso il linear probing è inefficiente). Le sequenze viste con questo metodo

iniziano da zero, poi uno, etc ma purtroppo anche in questo caso rimane il problema delle **m** sequenze distinte. Quindi, se due chiavi sono destinate a collidere all'inizio, lo saranno anche per le successive interazioni.

## Double hashing

Il **double hashing** fa uso non di una, ma di ben due funzioni ausiliari:

$$h(k, i) = (h_1(k) + i * h_2(k)) \bmod m \text{ considerando sempre}$$

$$h_1, h_2: U \rightarrow \{0, 1, \dots, m - 1\} \text{ e con } i = 0, 1, \dots, m - 1.$$

Le posizioni non sono individuate solo dalla posizione iniziale, che è sempre uguale, ma dall'offset che varia in base a  $h_2(k)$ . Quindi al primo tentativo le due chiavi collidono, se vengono mappate nella stessa posizione, ma nel secondo tentativo probabilmente riusciremmo ad evitare la collisione.

Es. definiamo  $h_1$  e  $h_2$ , 13 è il numero **m** di caselle disponibili, Sceglieremo una chiave k=14.

	0	
	1	79
	2	
	3	
	4	69
14	5	98
	6	
	7	72
	8	
	9	
	10	
	11	50
	12	

$h_1(k) = k \bmod 13$   
 $h_2(k) = 1 + (k \bmod 11)$   
  
 $k=14 = h_1(14) = 14 \bmod 13 = 1$   
 $h_2(14) = 1 + (14 \bmod 11) = 4$   
  
 $(1+0*4) \bmod 13 = 1$  occupato  
 $(1+1*4) \bmod 13 = 5$  occupato  
 $(1+2*4) \bmod 13 = 9$  libero

La scelta di  $k \bmod 11$  nella seconda funzione di hash è casuale e non ha niente a che vedere con la prima funzione di hash.

Anche in questo caso ( come per **c<sub>1</sub>** e **c<sub>2</sub> del quadratic probing**)  $h_2$  è scelto in modo da coprire tutta la tabella. In particolare,  $h_2$  non può avere fattori in comune con **m**, poiché genererebbe dei loop:

Es:  $h_1(k)=7$  e  $h_2(k)=4$ ,  $m=10$  ed abbiamo un fattore in comune tra  $m$  e la seconda funzione!.

$$h \rightarrow 7, 1, 5, 9, 3, 7, 1, 5, 9, 3, \dots$$

Due possibili modi per garantire che non abbiano fattori comuni sono:

- **m** è una potenza di 2,  $h_2(k)$  è dispari.
- **m** è primo,  $h_2(k)$  è minore di **m**.

Il double hashing ha prestazioni migliori, poiché questa volta si ha un numero proporzionale a **m<sup>2</sup>** tentativi (per ogni **m** di  $h_1$ , ci sono gli **m** di  $h_2$ ).

## Pseudocodice

Per inserire un elemento si provano tutte le posizioni fino a quando non se ne trova una libera. Per poter fare ciò si usa il *do while* che è scritto nello pseudocodice come *repeat until*.

```

Hash-Insert (T,k)
i ← 0
repeat
    j ← h(k,i)
    if T[j] = NIL
        then T[j] ← k
        return j
    else i ← i+1
until i = m
error "hash table overflow"

```

Per **la ricerca**, si ripercorre di nuovo tutta la sequenza fino a quando l'elemento non è ritrovato e ci restituisce la posizione. O si restituisce NIL se l'elemento non è presente in tabella.

```

Hash-Search (T,k)
i ← 0
repeat
    j ← h(k,i)
    if T[j] = k
        then return j
    i ← i+1
until T[j] = NIL or i = m
return NIL

```

Il caso peggiore per la search è purtroppo **O(n)** perché scorriamo tutto il vettore, ma **c'è un modo per scorrere tutto il vettore dal primo valore. Si cerca** prima in base ai vettori acceduti durante **l'inserimento**, una ricerca fatta in questo modo, nonostante nel caso peggiore abbia sempre un tempo **O(n)** ha più probabilità di restituire un valore più basso.

Per **l'eliminazione**, non possiamo sostituire l'elemento eliminato con NIL, poiché minerebbe l'algoritmo di ricerca (che prosegue fino a NIL).

Si può però **inserire** un valore speciale detto DELETED e modificare l'inserimento:

```

Hash-Insert (T,k)
i ← 0
repeat
    j ← h(k,i)
    if T[j]=NIL or T[j]=DELETED
        then T[j] ← k
        return j
    else i ← i+1
until i = m
error "hash table overflow"

```

Il problema del valore DELETED è che il tempo della ricerca non dipende più da  $\alpha = \frac{n}{m}$ , per cui in questo caso conviene usare la tabella hash con **concatenazione**.

LE SLIDE SULL'ANALISI DELL'INDIRIZZAMENTO APERTO SONO SALTATE N°26/27.

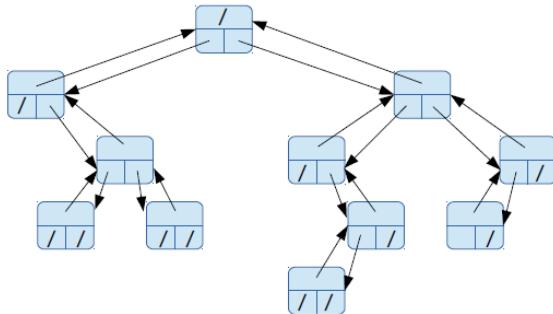
Alberi binari di ricerca

## Alberi Binari e n-ari

Un **albero binario** è un modo di rappresentare una struttura dati concatenata, ed è costituita da nodi che hanno:

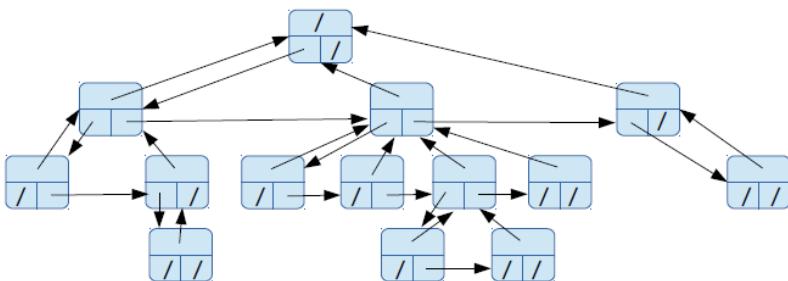
- Un puntatore al padre (**p**).
- Un puntatore al figlio di sinistra (**left**).
- Un puntatore al figlio di destra (**right**).

Un attributo **root** punta alla radice dell'albero.



In un **albero n-ario** ogni nodo può avere più figli, ma i puntatori sono ancora 3 e può essere rappresentato attraverso una struttura dati concatenata:

- Un puntatore al padre (**p**).
- Un puntatore al primo figlio (**left-child**).
- Un puntatore al fratello di destra (**right-sibling**).



## Alberi binari di ricerca

Gli **alberi di ricerca** sono strutture dati che supportano molte delle operazioni definite su insiemi dinamici, le operazioni basate su un albero binario richiedono un tempo proporzionale alla sua altezza. Le operazioni possibili sono:

- Search.
- Minimum.
- Maximum.
- Predecessor.
- Successor.
- Insert.

- Delete.

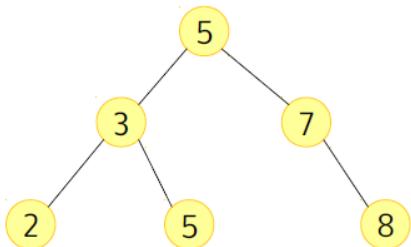
Ci sono due casi limite per il tempo d'esecuzione:

- Albero completo (bilanciato e con altezza  $\lg n$ ) =  **$\lg n$** .
- Lista concatenata (caso degenere sbilanciato):  **$n$** .

Un albero binario di ricerca presenta dei vincoli, per poter essere chiamato così e sono:

- Se **y** è nel sottoalbero di sinistra del nodo **x**, allora **key[y] ≤ key[x]**. In altre parole i nodi del sottoalbero di sinistra contengono un valore minore del nodo di partenza.
- Se **y** è nel sottoalbero di destra del nodo **x**, allora **key[y] ≥ key[x]**. In altre parole i nodi del sottoalbero di destra contengono un valore maggiore del nodo di partenza.

Purtroppo non è sempre possibile costruire un albero bilanciato, alcuni tipi particolari di alberi presentano nel caso peggiore comunque un tempo di esecuzione accettabile.



## Visita in ordine

La visita in ordine consiste nello stampare tutti i nodi, una sola volta, in **ordine crescente**. Basta invocare ricorsivamente la funzione in modo tale che per ogni nodo stampi prima il figlio di sinistra, poi sé stesso, poi quello di destra.

```

Inorder-Tree-Walk (x)
if x≠NIL
  then Inorder-Tree-Walk (left[x])
  print key[x]
  Inorder-Tree-Walk (right[x])
  
```

Il tempo di esecuzione, intuitivamente è  **$\Theta(n)$** , poiché visitiamo una alla volta tutti gli **n** nodi.

$T(n) = T(k) + T(n - k - 1) + d$  dove  $T(k)$  è il numero di nodi del sottoalbero di sinistra, l'altro è il numero di nodi del sottoalbero di destra.

Usiamo il **metodo di sostituzione**

$$T(k) = (c + d)k + c \rightarrow T(n) = (c + d)k + c + (c + d)(n - k - 1) + c + d = (c + d)n + c$$

Se l'elaborazione viene fatta dopo l'ordine è chiamata **postordine** oppure nel caso contrario ,quello visto, è **visita in preordine**.

## Ricerca

La ricerca viene effettuata a sinistra o a destra, a seconda se l'elemento da cercare è rispettivamente minore o maggiore del nodo di partenza. Ovviamente nel caso in cui la chiave si trova nel nodo corrente o non viene trovata, la ricerca termina.

```

Tree-Search (x,k)
if x=NIL or key[x]=k
    then return x
if k < key[x]
    then return Tree-Search (left[x],k)
else return Tree-Search (right[x],k)

```

Il tempo di esecuzione è **O(h)**, con **h** altezza dell'albero, poiché ad ogni step si scende di un gradino.

In realtà è più efficiente la sua versione **iterativa**, poiché la ricorsione consuma ad ogni iterazione una porzione di memoria per lo stack di attivazione:

```

Iterative-Tree-Search (x,k)
while x≠NIL and key[x]≠k
    do if k < key[x]
        then x ← left[x]
        else x ← right[x]
return x

```

Ricordandoci che questo è possibile proprio perché il nodo è un puntatore.

## Minimo e massimo

Questo tipo di algoritmo ci consente di trovare il **minimo**, sempre usando un'iterazione, infatti scendendo subito sulla sinistra sappiamo che troviamo sempre un valore più piccolo del padre; ma se ci restituisse NIL allora tornando al padre sapremmo che è il più piccolo. Provando a scendere a destra, controlleremo se ci sono valori più piccoli del padre, anche se per costruzione dovrebbero essere più grandi. Per questo richiamiamo ricorsivamente la funzione.

```

Tree-Minimum (x)      Tree-Maximum (x)
while left[x]≠NIL      while right[x]≠NIL
    do x ← left[x]      do x ← right[x]
return x                  return x

```

Il tempo di esecuzione è ancora **O(h)**, poiché si scende ad ogni iterazione, al massimo fino alla foglia. In ogni caso è sempre da preferire un **algoritmo iterativo** che uno **ricorsivo**, data dalla semplicità implementativa del primo.

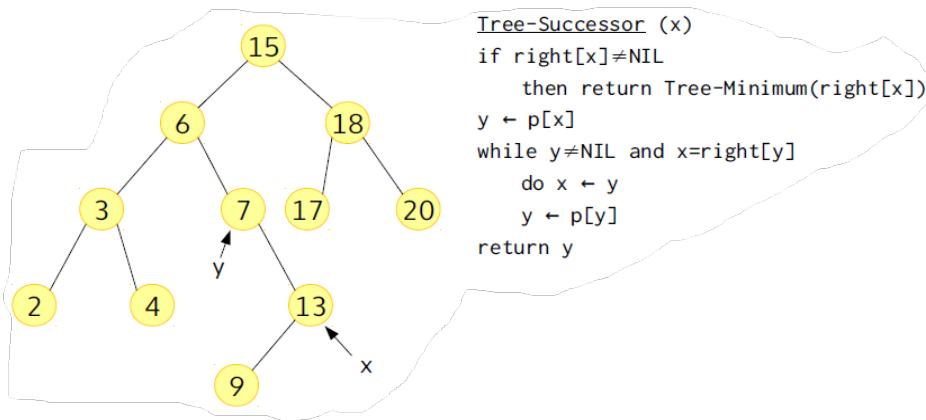
## C

### Successore

Il **successore** è il più piccolo numero maggiore della chiave, cioè il successivo in visita in ordine.

Distinguiamo due casi:

- Se il nodo ha un figlio di destra, il successore è il minimo del sottoalbero di destra (cioè quello più a sinistra di quelli a destra). Es successore di 6 è 7.
- Se non ha un figlio di destra, il successore è il primo antenato (risalendo l'albero) il cui figlio di sinistra è antenato del nodo di partenza, cioè il primo antenato a cui arriviamo da destra. Es il successore di 13 è 15.



All'interno dell'algoritmo **p[x]** indica il padre di x. Nel momento in cui x diventa il figlio di sinistra di y, ci fermiamo. Il ciclo termina quando il nodo x è un figlio di sinistra di y (o se non esiste un successore). Anche in questo caso il tempo di esecuzione è **O(h)**, poiché o si scende sempre, o si sale sempre.

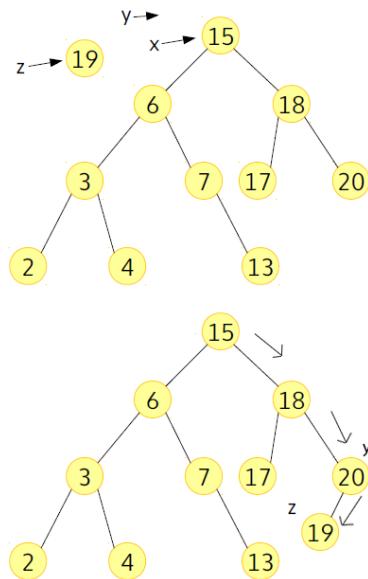
## Inserimento

Quando si inserisce un elemento all'interno di un albero di ricerca, bisogna rispettarne le sue proprietà. Viene dunque fatto scendere a sinistra o a destra, ad ogni iterazione, a seconda se è minore o maggiore del nodo con cui si sta confrontando facendo sì che l'elemento attraversi l'albero verso il basso partendo dalla radice.

```

Tree-Insert (T,z)
y ← NIL
x ← root[T]
while x≠NIL
    do y ← x
    if key[z] < key[x]
        then x ← left[x]
        else x ← right[x]
p[z] ← y
if y = NIL
    then root[T] ← z // albero vuoto
    else if key[z] < key[y]
        then left[y] ← z
        else right[y] ← z

```



Sia **z** il nuovo nodo, occorre dunque modificare sia il puntatore al suo padre, sia il puntatore al nuovo figlio **z**. In questo algoritmo c'è la possibilità che **y** sia nullo e che quindi differenziamo un albero vuoto, creando problemi di segmentation fault, risolviamo questo problema con la seconda parte del codice. Assegnamo quindi l'attributo radice a **z** che diventerà la radice dell'albero.

Il tempo di esecuzione è sempre **O(h)**.

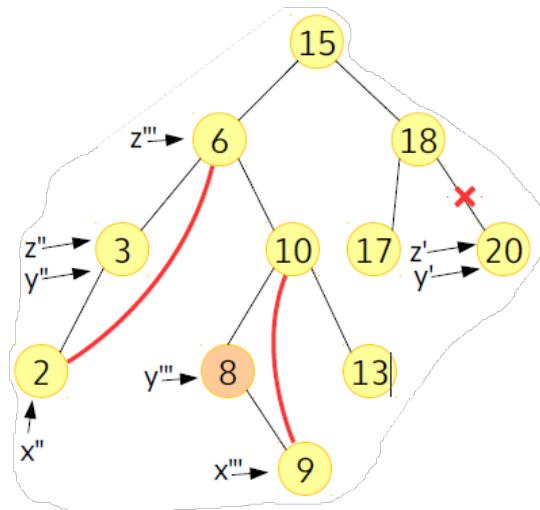
## Eliminazione

L'operazione di **eliminazione** è piuttosto complicata, poiché per mantenere le proprietà dell'albero di ricerca occorre modificare le posizioni di alcuni nodi.

Distinguiamo 3 casi diversi:

- Il nodo non ha figli.
  - Il padre punterà a NIL.
- Il nodo ha un solo figlio.
  - Il padre punterà al figlio del nodo eliminato (e viceversa).
- Il nodo ha due figli
  - È sostituito con il suo successore.
  - Il successore (che ha un solo figlio) viene eliminato come nel secondo caso.

```
Tree-Delete (T,z)
if left[z]=NIL or right[z]=NIL
  then y ← z
  else y ← Tree-Successor (z)
if left[y] ≠ NIL
  then x ← left[y]
  else x ← right[y]
if x ≠ NIL      // y non e' foglia
  then p[x] ← p[y]
if p[y] = NIL    // y è radice
  then root[T] ← x
  else if y = left[p[y]]
    then left[p[y]] ← x
    else right[p[y]] ← x
if y ≠ z // terzo caso
  then key[z] ← key[y]
  Copy y's satellite data into z
return y
```



Dall'esempio possiamo vedere di come nel primo caso  $z'$  e  $y'$  indicano l'assenza di figli ed infatti è il caso più semplice, basta eliminare il nodo ed è fatta. Nel secondo caso  $z''$  e  $y''$  indicano la situazione nella quale il nodo ha un solo figlio. Questo caso si risolve tagliando fuori il nodo 3, il figlio del padre di 6 diventa 2 e significa aggiornare due puntatori. Il terzo caso invece è il più complicato, eliminiamo 6, dobbiamo individuare il successore di esso e quindi di sicuro è maggiore di tutti i nodi di sinistra e più piccolo dei nodi di destra.

Si nota di come il caso tre viene implementato usando il caso due, ma applicato al successore. Infatti, la prima condizione booleana, è quella di controllare in uno dei primi due casi oppure nel tre.

L'unica operazione non costante è quella per trovare il successore, quindi anche in questo caso si ha  $O(h)$ .

## Lezione 19 21/11/19

### Alberi rosso-neri

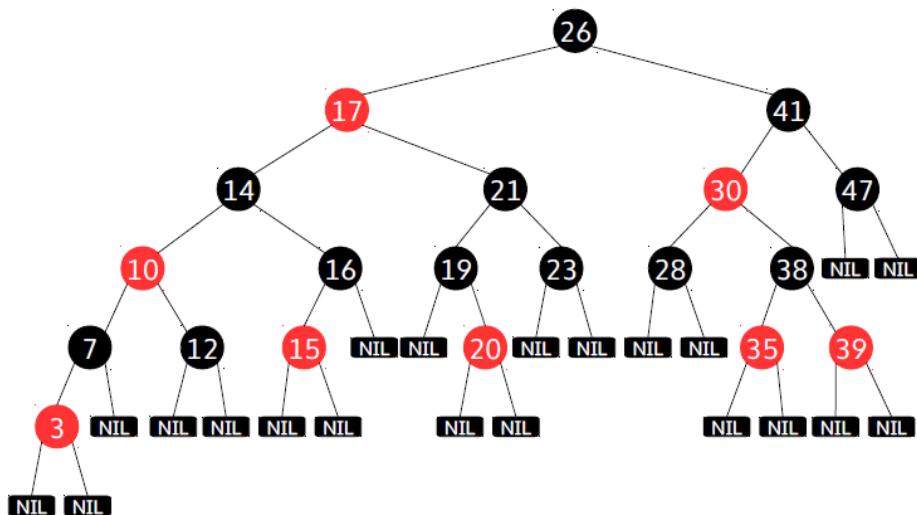
#### Alberi rosso-neri

Una limitazione degli alberi di ricerca generici è il non avere garanzia che l'albero sia bilanciato. Infatti, quando un nuovo nodo viene inserito esso è posto a destra, se è maggiore, o a sinistra se è minore della radice. Inserendo ad esempio, solo valori crescenti l'albero crescerebbe solo a destra. Esistono dunque diverse soluzioni per far sì che l'albero sia bilanciato e quindi sia possibile sfruttare le complessità

computazionali precedentemente descritte; le soluzioni più adottate sono gli alberi rosso-neri e gli alberi auto-aggiustanti.

Gli **Alberi rosso-neri** sono particolari alberi binari di ricerca in cui ogni nodo ha un attributo addizionale **color**, che può valere rosso o nero e garantiscono che nessun percorso dalla radice ad una foglia sia più lungo del doppio di un altro percorso. Le foglie esterne sono puntatori a NIL, in più devono essere soddisfatte le seguenti proprietà:

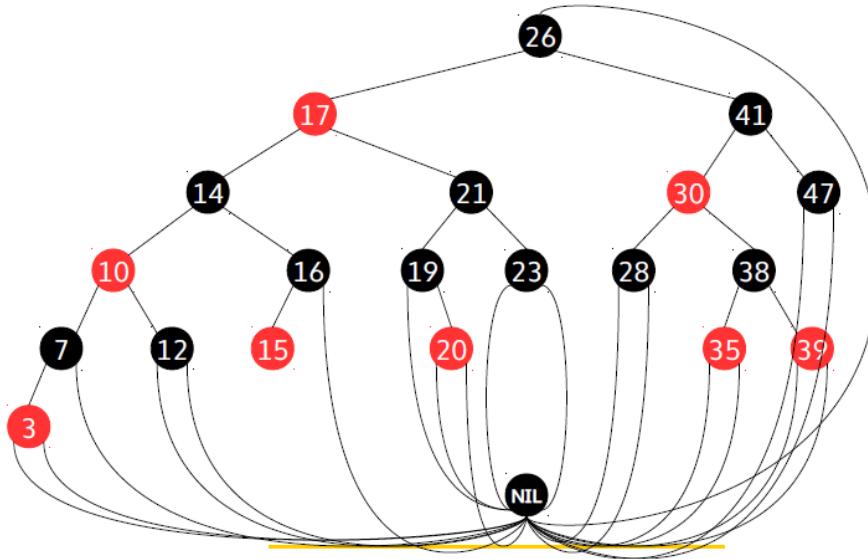
- Ogni nodo è rosso o nero.
- La radice deve essere nera.
- Ogni nodo foglia esterno (NIL) è nero.
- Se un nodo è rosso, entrambi i figli sono neri.
- Tutti i percorsi da un nodo alle foglie hanno lo stesso numero di nodi neri.



Non dimentichiamoci però che è sempre un nodo binario, quindi a sinistra ci sono gli elementi più piccoli e a destra quelli più grandi.

Possiamo notare che, in virtù delle ultime due proprietà, il percorso più breve possibile è quello senza nodi rossi e quello più lungo ha rossi e neri che si alternano. Di conseguenza, il percorso più lungo è al massimo il doppio di quello più corto. Ogni nodo interno poi ha sempre due figli, uno o entrambi possono essere NIL. Un nodo rosso o ha due figli NIL oppure sono entrambi interni (proprietà 5 violata in caso contrario).

Da un punto di vista implementati il valore NIL non saranno diverse istanze dei nodi, ma saranno un unico oggetto NIL al quale tutti i nodi esterni punteranno.



### Altezza e black-height (bh)

Oltre al normale concetto di altezza(height), è introdotto il concetto di **black-height** che consiste nel numero di nodi neri lungo un quale percorso arriva ad una foglia(escluso il nodo stesso, incluso il nodo foglia NIL).

La black-height di un albero rosso-nero è la black-height della radice. Dimostriamo che un albero rosso-nero con  $n$  nodi interni ha altezza al più pari a  $2\ln(n+1)$ , ricordandoci che l'altezza di un albero completo è  $\log(n)$  quindi è proporzionale. Il primo passo da dimostrare è che se prendiamo un qualsiasi nodo  $x$  dell'albero rosso-nero il sottoalbero con radice in  $x$  ha almeno  $2^{bh(x)} - 1$  nodi interni, dove  $bh(x)$  è la black-height di  $x$ . Dimostriamolo con il principio di induzione (sull'altezza di un nodo):

- Caso base:  $h = 0$  (*foglia esterna*)  $\rightarrow 2^{bh(x)} - 1 = 2^0 - 1 = 0$  nodi interni. In effetti è vero che un sottoalbero di una radice esterna ha zero nodi interni.
- Passo induttivo:  $\forall$  figlio di  $x$ :  $bh = \begin{cases} bh(x) & \text{se } x \text{ è rosso} \\ bh(x) - 1 & \text{se } x \text{ è nero} \end{cases}$  ogni singolo figlio di  $x$  ha almeno  $2^{bh(x)} - 1$  nodi interni.

$x$  ha almeno  $2 * (2^{bh(x)} - 1) + 1 = 2^{bh(x)} - 1$  nodi interni, la moltiplicazione del 2 fuori parentesi indica la presenza di 2 figli perché l'altezza  $h > 0$  e la somma unitaria è per se stesso.

Es: Qual è la black-height del nodo 10(rosso)? 2 mentre per i figli (7 e 12 è pari 1 ovvero  $x-1$ )

Dimostriamo ora che l'altezza è al massimo  $2\lg(n+1)$ , dove  $n$  è il numero di nodi interni.

Poiché per ogni nodo rosso c'è almeno un nodo nero, nel percorso più lungo rossi e neri si alternano, quindi almeno metà dei nodi di un percorso sono neri:

$$\rightarrow bh(\text{root}) \geq \frac{h}{2} \text{ abbiamo applicato il risultato precedente alla radice}$$

$$\rightarrow n \geq 2^{\frac{h}{2}} - 1 \text{ } n \text{ numero di nodi interni}$$

$$\rightarrow \log(n+1) \geq \lg\left(2^{\frac{h}{2}}\right) = \frac{h}{2} \rightarrow h \leq 2\lg(n+1)$$

Questo risultato cosa ci dice? Per quanto riguarda le operazioni di ricerca si eseguono tutte identicamente a come accadeva sugli alberi binari, quindi con un tempo  $O(h)$ ; ma  $h$  negli alberi rosso-neri è limitato superiormente da  $2\lg(n+1)$ , quindi questo vuol dire che tutte le operazioni di ricerca si conducono in un tempo pari a  $O(\log(n))$ . Ricordiamo che negli alberi  $h = \lg n$  solo nel caso in cui l'albero risulti bilanciato. Purtroppo inserimento ed eliminazione devono essere modificate per preservare le proprietà degli alberi rosso-neri.

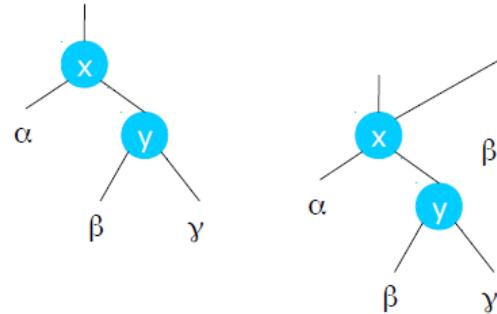
## Rotazione in alberi rosso-neri

L'operazione di rotazione si esegue sugli alberi di ricerca generici, quindi non è una esclusiva dei rosso-neri, ma in cosa consiste? Esistono rotazioni sia a sinistra che a destra (uno il duale dell'altro).

La rotazione verso sinistra è una rotazione in senso anti-orario rispetto al nodo  $x$  (ovvero il nodo sinistro deve avere un figlio sulla destra), vediamo come  $y$  prenda il posto di  $x$ :

1. Spostiamo  $\beta$ :

- a.  $\beta$  diventa il figlio di destra di  $x$ .
- b.  $x$  diventa il padre di  $\beta$ .



2. Spostiamo  $y$

- a. Il padre di  $x$  diventa il padre di  $y$ .  
(Salendo  $y$  si porta il suo sottoalbero)
- b.  $y$  diventa figlio del padre di  $x$ .

3. Spostiamo  $x$

- a.  $x$  diventa figlio di sinistra di  $y$ . (Quindi dobbiamo spostare il sottoalbero di sinistra di  $y$ )
- b.  $y$  diventa padre di  $x$ .

Queste operazioni vanno eseguite nel seguente ordine per evitare di perdere pezzi di informazione.

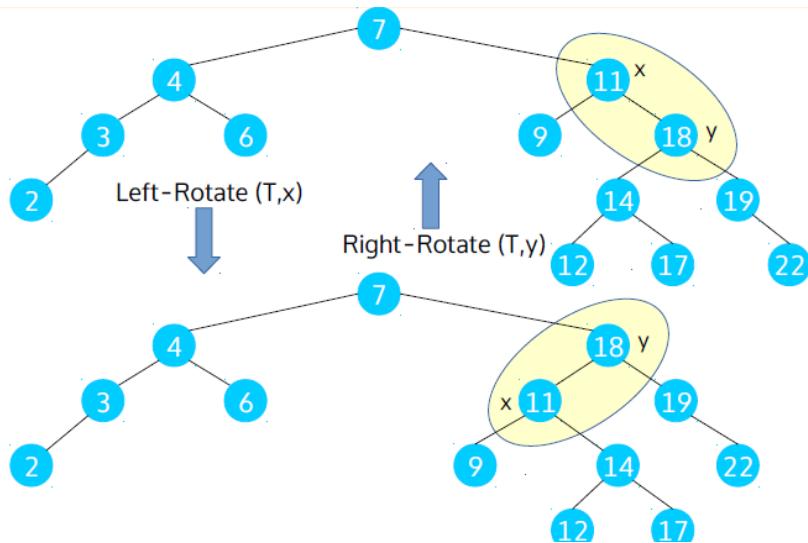
Il tempo di esecuzione è di  $O(1)$ , poiché non ci sono interazioni.

Left-Rotate ( $T, x$ )

```

y ← right[x]
// operazione 1.
right[x] ← left[y]
if left[y] ≠ nil[T]
    then p[left[y]] ← x
// operazione 2.
p[y] ← p[x]
if p[x] = nil[T]
    then root[T] ← y
    else if x = left[p[x]]
        then left[p[x]] ← y
        else right[p[x]] ← y
// operazione 3.
left[y] ← x
p[x] ← y

```



La cosa importante di questo pseudocodice è l'ordine con il quale viene eseguito, poiché come detto precedentemente potremmo perdere i riferimenti a dei sottoalberi. Per chiascuna operazione bisogna prima aggiornare il puntatore, indifferentemente dall'ordine (alto o basso). La rotazione inversa (quella destra) richiede che esista un figlio sulla sinistra.

## Inserimento

Partendo dall'algoritmo di inserimento in un albero di ricerca generico, le uniche variazioni sono:

- NIL diventa un nodo **nil[T]**.
- Il nodo da inserire avrà 2 figli **nil[T]**.
- Il nodo è colorato di rosso.
- Si invoca una funzione ausiliaria(RB-Insert-Fixup) per ripristinare le proprietà degli alberi rosso-neri.

Albero Genrico	Albero rosso-nero
<pre><u>Tree-Insert</u> (T,z) y ← NIL x ← root[T] while x≠NIL     do y ← x     if key[z] &lt; key[x]         then x ← left[x]         else x ← right[x] p[z] ← y if y = NIL     then root[T] ← z // albero vuoto else if key[z] &lt; key[y]     then left[y] ← z     else right[y] ← z</pre>	<pre><u>RB-Insert</u> (T,z) y ← <b>nil[T]</b> x ← root[T] while x≠<b>nil[T]</b>     do y ← x     if key[z] &lt; key[x]         then x ← left[x]         else x ← right[x] p[z] ← y if y = <b>nil[T]</b>     then root[T] ← z // albero vuoto else if key[z] &lt; key[y]     then left[y] ← z     else right[y] ← z <b>left[z] ← nil[T]</b> <b>right[z] ← nil[T]</b> <b>color[z] ← RED</b> <u>RB-Insert-Fixup</u> (T,z)</pre>

La parte del while nel codice dell'albero rosso-nero serve nel caso in cui il figlio di sinistra sia assente. Ovviamente dovremmo cercare di far sì che il codice abbia come tempo di esecuzione **O(lg n)**.

Vediamo ora se sono rispettate le proprietà:

1	Ogni nodo è rosso o nero	SI
2	La radice è nera	NO(se l'albero era vuoto)
3	Le foglie esterne(NIL) sono nere	SI
4	I nodi rossi hanno figli neri	NO(se il padre è rosso)
5	Tutti i percorsi da un nodo alle foglie hanno lo stesso numero di nodi neri	SI

Le uniche proprietà che possono essere violate sono la **2** e la **4**, considerando di procedere secondo la maniera precendentemente scritta. La seconda proprietà può essere violata quando inseriamo un nodo all'interno di un albero vuoto, perché la radice risulterebbe rossa; per la quarta proprietà il nuovo nodo potrebbe risultare che un nodo rosso ha un figlio rosso.

## Algoritmo di Fixup

### Invariante

La RB-Insert-Fixup è costituita da un ciclo while che viene eseguito fintanto che il padre di  $z$  resta rosso, ed è costituita in modo da mantenere il seguente **invariante**:

Ad ogni iterazione del ciclo valgono le seguenti proprietà:

1. Il nodo  $z$  è rosso, attenzione però che durante le interazioni  $z$  cambierà e punterà a nodi diversi.
2. Se  $p[z]$ , il padre di  $z$ , è la radice, allora  $p[z]$  è nero
3. In tutto l'albero è violata al massimo una tra le proprietà 2 e 4
  - a. Se è violata la 2, si presena perché  $z$  è la radice ed è rosso.
  - b. Se è violata la 4, si presenta perché sia  $z$  che  $p[z]$  sono rossi.

Prima della prima iterazione (quindi alla fine di *Insert* e prima di evocare la *Fixup*), abbiamo che ( $z$  è il nodo che è stato inserito):

1. Il nuovo nodo è rosso
2. La *Insert* non ha modificato  $p[z]$ , quindi se era la radice rimane nero.
3. Le proprietà 1,3,5 non sono violate da RB-Insert
  - a. Se c'è una violazione di tipo 2, è perché l'albero era vuoto e il nodo inserito è la nuova radice. Non ci sono violazioni di tipo 4 (foglie nere)
  - b. Se c'è una violazione di tipo 4, è perché  $z$  viene inserito come figlio di un nodo rosso. Non ci sono violazioni di tipo 2

### Costruzione della Fixup

Il ciclo termina quando  $p[z]$  diventa nero, quindi non può esserci una violazione della proprietà 4 grazie all'invariante. Può essere però ancora violata la 2, ma basta colorare il nodo di nero. Vediamo come implementare il corpo del ciclo while in modo da preservare l'invariante.

Se entriamo nel ciclo *while* abbiamo che l'invariante è vero e che:

- $p[z]$  è rosso
  - Perché il ciclo termina se è nero.
- $p[p[z]]$  è interno ed è nero (il nonno di  $z$ )
  - Perché  $p[z]$  è rosso, quindi non è radicem quindi suo padre non è un nodo **nil[T]**.
  - Essendo  $p[z]$  rosso, suo padre è nero (o la proprietà 4 sarebbe violata due volte)
- Non è violata la proprietà 2

- Perché  $z$  sarebbe la radice e quindi  $p[z]$  sarebbe nero.

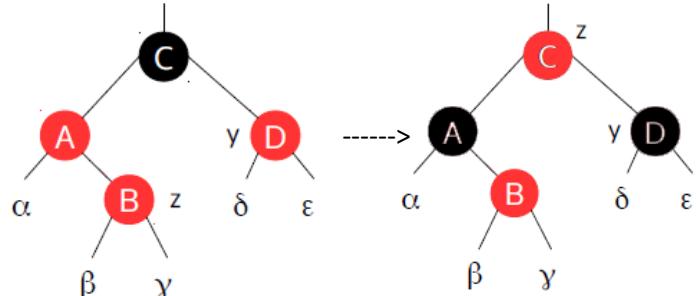
Ora sapendo che  $z$  e il padre sono rossi, mentre il nonno è nero, distinguiamo 3 casi in base al colore dello zio. Assumiamo che  $p[z]$  sia il figlio di sinistra (per il caso opposto basta invertire left e right).

## Lezione 20 27/11/19

### Caso 1: Lo zio di $z$ è rosso (lo indichiamo con $y$ )

- Invertiamo colori del padre, zio, nonno
- Il nuovo  $z$  è il nonno

L'invarianza è preservata:



1. Il nuovo  $z$  è rosso.
2. Suo padre non è stato modificato, quindi

Se era la radice è ancora nero.

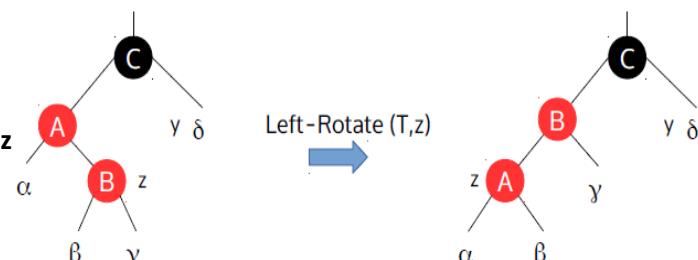
3. È violata al massimo una proprietà:

- a. La 2 è violata se il nuovo  $z$  è la radice.
- b. La 4 è violata se suo padre è rosso.
- c. La 5 NON è violata, perché al posto del nodo C nero abbiamo A e D nodi neri su due percorsi diversi.

### Caso 2: Lo zio di $z$ è nero, $z$ è a destra

- Il nuovo  $z$  è il padre
- Ruotiamo a sinistra intorno al nuovo  $z$

Ci riconduciamo poi al caso 3.



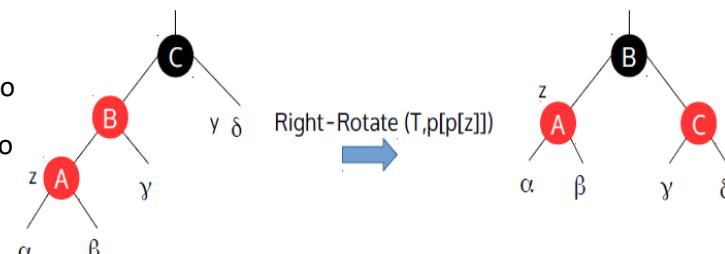
L'invarianza è preservata:

1. Il nuovo  $z$  è rosso.
2. Suo padre non è la radice.
3. È violata al massimo una proprietà:
  - a. La 2 non è violata perché il nuovo  $z$  **non** è la radice.
  - b. La 4 è violata.
  - c. La 5 NON è violata, perché i nodi neri non sono stati toccati.

### Caso 3: Lo zio di $z$ è nero, $z$ è a sinistra

- Invertiamo i colori di padre e nonno
- Ruotiamo a destra intorno al nonno

Il ciclo è terminato perché  $p[z]$  è nero.



L'invariante è preservato:

1. Il nuovo  $z$  è rosso.

2. Suo padre è colorato di nero in ogni caso.
3. È violata al massimo una proprietà (in realtà nemmeno una):
  - a. La 2 non è violata perché il nuovo **z** **non** è la radice, ma in ogni caso suo padre risulta nero.
  - b. La 4 **NON** è violata.
  - c. La 5 **NON** è violata, perché i nodi neri non sono stati toccati.

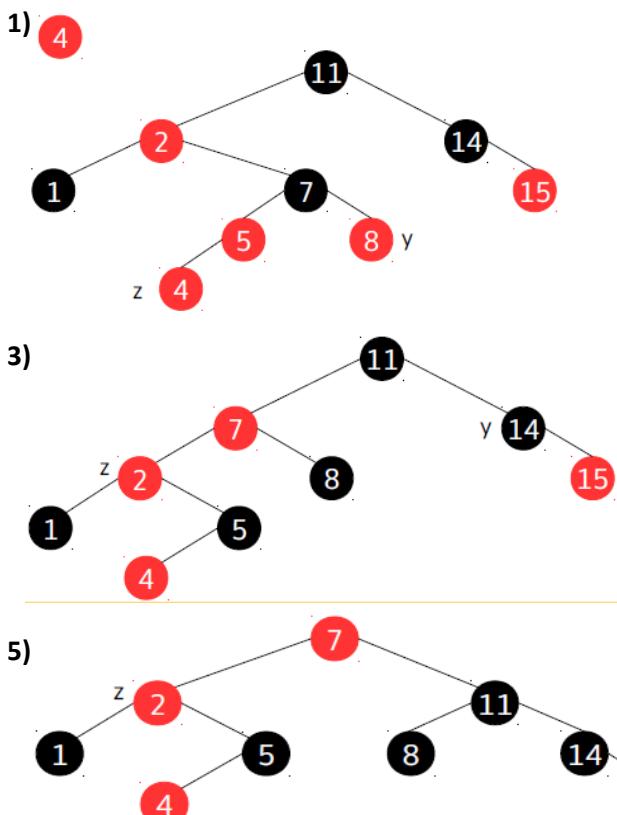
#### RB-Insert-Fixup (T,z)

```

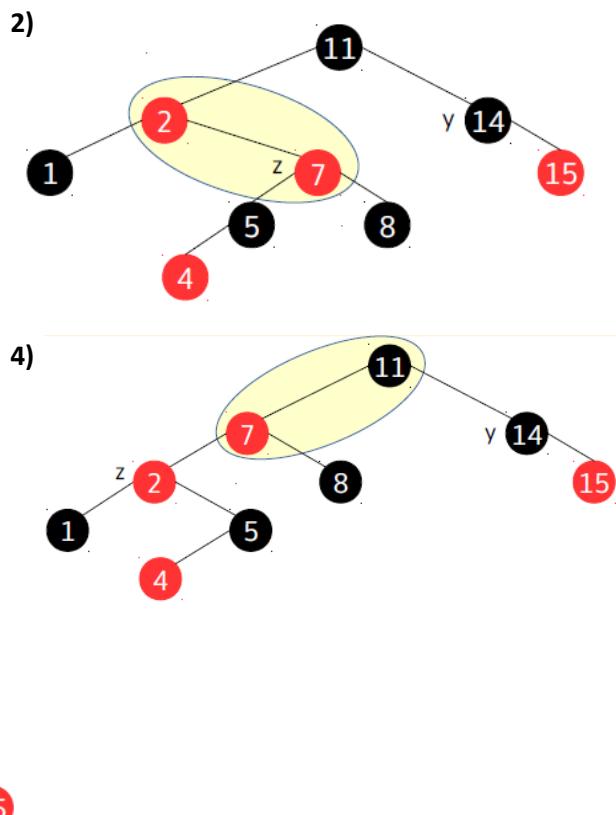
while color[p[z]] = RED
  do if p[z] = left[p[p[z]]]
    //Il padre è figlio di Sx
    then y ← right[p[p[z]]] // zio
      if color[y] = RED
        //Caso 1: zio rosso
        then color[p[z]]←BLACK
          color[y] ← BLACK
          color[p[p[z]]]←RED
          z ← p[p[z]]
        else if z = right[p[p[z]]]
          //Caso 2: zio nero, z a Dx
          then z ← p[z]
            Left-Rotate(T,z)
        //Caso 3: zio nero, z a Sx
          color[p[z]]←BLACK
          color[p[p[z]]] ← RED
          Right-Rotate(T,p[p[z]])
    //Il padre è figlio di Dx
    else (stesso codice di then con right e left scambiati)

color[root[T]] ← BLACK
  
```

Il tempo di esecuzione della *RB-Insert-Fixup* è **O(lg n)** poiché: il ciclo si ripete solo se siamo nel caso 1 e sale ogni volta verso l'alto; il ciclo termina quando entriamo nel caso 2 o 3 e quindi al più due rotazioni.



Esempio: inserimento della chiave 4



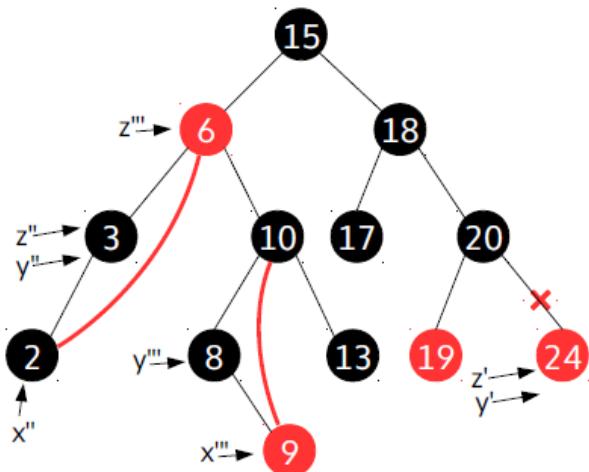
## Eliminazione di un elemento

Albero Genrico	Albero rosso-nero
a	<pre> <u>RB-Delete</u> (T, z)   <b>if</b> left[z] = nil[T] <b>or</b> right[z] =nil[T]     <b>then</b> y ← z     <b>else</b> y ← RB-Successor(z)   <b>if</b> left[y] ≠ nil[T]     <b>then</b> x ← left[y]     <b>else</b> x ← right[y]   p[x] ← py[y]   <b>if</b> p[] = nil[T]     <b>then</b> root[T] ← x     <b>else</b> <b>if</b> y = left[p[y]]       <b>then</b> left[p[y]] ← x       <b>else</b> right[p[y]] ← x   <b>if</b> y ≠ z     <b>then</b> key[z] ← key[y]     <i>copy y's satellite data into z</i>   <b>if</b> color(y) = BLACK     RB-Delete-Fixup(T, x)  <b>return</b> y </pre>

La *Fixup* (di cui non vediamo il codice) è invocata **solo se il nodo da tagliare fuori è nero**. Infatti l'unico caso in cui viene tagliato fuori un nodo rosso, è quando è una foglia interna, poiché negli altri casi ha 2 figli (neri) altirimenti viene violata la proprietà 5 (quindi viene tagliato fuori il successore).

Rimuovere una foglia rossa non viola le proprietà dell'albero, quindi non necessita di un fixup.

Anche l'eliminazione è **O(lg n)**.



## Lezione 21 28/11/19

### Analisi ammortizzata

#### Analisi ammortizzata

Esistono diversi modi di analizzare le prestazioni degli algoritmi, con **l'analisi ammortizzata** si va a calcolare il tempo di esecuzione di una sequenza di **n** operazioni nel caso peggiore (quindi si valuta il limite superiore), piuttosto che analizzare le singole operazioni separatamente è quindi un'analisi delle prestazioni medie nel caso peggiore. È detta "ammortizzata" perché tiene conto del fatto che il costo di alcune operazioni che finiscono nel caso peggiore può essere ammortizzato da altre operazioni che non finiscono nel caso peggiore.

Fino ad adesso avevamo valutato solo il costo della singola operazione, sulle strutture dati invece ha più senso valutare una sequenza di operazioni quali: ricerca, eliminazione etc. Nella sequenza il limite superiore può essere individuato come la somma dei limiti superiori delle varie sequenze.

Eseguire l'analisi ammortizzata può fornire degli spunti per ottimizzare una struttura e ha senso quando i costi ammortizzati alle varie operazioni sono tali che, qualunque sequenza di **n** operazioni andiamo a considerare, la somma dei costi ammortizzati delle **n** operazioni effettuate è un limite superiore per il costo reale. Vediamo 3 metodi:

- Metodo dell'aggregazione
- Metododegli accantonamenti
- Metodo del potenziale

#### Metodo dell'aggregazione

Una sequenza di **n** operazioni impiega un totale, nel caso peggiore, **T(n)**, per ogni **n**.

Il **costo ammortizzato** di ogni operazione è **T(n) / n**. In questo caso, quindi, il costo ammortizzato rappresenta il costo medio che viene assegnato a ciascuna operazione.

#### Esempio 1: Pila con operazione Multipop

La *multipop* consente di estrarre k elementi alla volta.

```
Multipop (S, k)
while not Stack-Empty(S) and k ≠ 0
    do Pop(S)
    k ← k-1
```

Il suo costo è minimo tra **s** e **k**, dove **s** è il numero di elementi dello stack. Consideriamo ora una sequenza di **n** operazioni *push* (*costo 1*), *pop* (*costo 1*), *multipop* su una pila vuota.

Il caso peggiore è quando abbiamo **n** multipop, che è pari a **O(n)** nel caso peggiore, indicando con **n** anche il numero massimo di elementi che può contenere lo stack. Si ha quindi **O(n<sup>2</sup>)**. Questa è l'analisi tradizionale mentre possiamo trovare un limite più stretto con **l'analisi aggregata**.

Ogni elemento inserito può essere estratto una sola volta. Di conseguenza, possono essere inseriti ed estratti massimo **n** elementi. Nel caso peggiore si ha quindi **O(n)** e il costo ammortizzato di ciascuna operazione è **O(n) / n = O(1)**.

#### Esempio 2: Contatore binario (a crescere) su **k** bit. Contatore modulo $2^k$ .

Interpretiamo il costo come il numero di bit che vengono invertiti nel contatore:

cou nter	A[7]	A[6]	A[5]	A[4]	A[3]	A[2]	A[1]	A[0]	cost
0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	1	1
2	0	0	0	0	0	0	1	0	3
3	0	0	0	0	0	0	1	1	4
4	0	0	0	0	0	1	0	0	7
5	0	0	0	0	0	1	0	1	8
6	0	0	0	0	0	1	1	0	10
7	0	0	0	0	0	1	1	1	11
8	0	0	0	0	1	0	0	0	15
9	0	0	0	0	1	0	0	1	16
10	0	0	0	0	1	0	1	0	18
11	0	0	0	0	1	0	1	1	19
12	0	0	0	0	1	1	0	0	22
13	0	0	0	0	1	1	0	1	23
14	0	0	0	0	1	1	1	0	25
15	0	0	0	0	0	1	1	1	26
16	0	0	0	1	0	0	0	0	31

### Increment (A)

```
i ← 0
while i<length[A] and A[i]=1
    do A[i] ← 0
    i ← i+1
if i<length[A]
    then A[i] ← 1
```

Notiamo che:

- Se l'ultima cifra è 0 il costo aumenta di 1
  - Se l'ultima cifra è 1 il costo aumenta di 1+ il numero di 1 meno significativi.
- Quindi se ad esempio doppiamo contare da 7 a 8, abbiamo che 7 ha tre 1 dopo lo zero quindi sono i meno significativi ed ha costo 11; al conteggio successivo sarà con costo  $11+1+1+1+1 = 15$ . Mentre da 8 a 9 bisogna solo aumentare di 1 il costo.

Il costo della *increment* nel caso peggiore si ha quando vengono invertiti tutti i bit, quindi è **O(k)**. Se consideriamo una sequenza di **n** *Increment*, abbiamo **O(nk)**. Otteniamo un bound più stretto osservando che:

- A[0] è invertito ogni volta
- A[1] è invertito ogni 2 volte (quindi metà delle volte)
- A[2] è invertito 4 volte (quindi un quarto delle volte)
- ...

Quindi il numero di bit invertiti in una sequenza di **n** *Increment* è pari a:  $\sum_{i=1}^{k-1} \left\lfloor \frac{n}{2^i} \right\rfloor < n \sum_{i=1}^{k-1} \frac{1}{2^i} = 2n$ .

Quindi in realtà il caso peggiore è **O(n)** perché è limitato da un **n** proporzionale, e il costo ammortizzato di ciascuna operazione è **O(n)/n = O(1)**.

## Metodo dell'accantonamento

In questo caso ad ogni operazione è assegnato un costo ammortizzato diverso, che può differire dal costo reale dell'operazione. Questo metodo è basato sul concetto di **credito**, quando un costo ammortizzato supera quello reale, la differenza è mantenuta come **credito**, che viene consumato quando è invece inferiore a quello reale.

Un **vincolo** da rispettare è che, data una sequenza di **n** operazioni, il costo ammortizzato totale deve essere un limite superiore del costo reale totale, cioè il credito non deve essere mai negativo. Applichiamo questo metodo ai due esempi precedenti.

### Esempio 1: Pila (inizialmente vuota)

	<b>push</b>	<b>pop</b>	<b>multipop</b>
<b>Costo reale</b>	1	1	$\min(s,k)$
<b>Costo ammortizzato</b>	2	0	0

Abbiamo assegnato un costo ammortizzato maggiore alla push, così che il credito non sia mai negativo.

Possiamo interpretarla come una pila di piatti, in cui il credito è rappresentato da monete:

- Con la *push* "riceviamo" due monete, una la usiamo per pagare il costo reale di push, l'altra la lasciamo all'interno.
- Con la *pop* bisogna consumare una moneta di credito, quindi diamo via il piatto e consumiamo la moneta che avevamo conservato.
- Analogamente per la *multipop*.

Quindi il credito che avanza con la *push* serve a "ripagare" il costo di *pop* e *multipop*.

Il credito **non è mai negativo**, quindi il costo ammortizzato è **(2n)** un limite superiore del costo reale, che è quindi **O(n)**.

**Esempio 2:** Contatore binario su **k** bit (inizialmente nullo)

	<b>Set</b>	<b>Reset</b>	<b>Increment</b>
<b>Costo reale</b>	1	1	n° di bit invertiti
<b>Costo ammortizzato</b>	2	0	2

Abbiamo che:

- Il set genera un credito pari a 1.
- Il reset consuma un credito pari a 1.

Il costo ammortizzato dell'*increment* è 2, perché ad ogni incremento c'è al massimo un set (che costa 2) e dei reset, che costano 0. Dunque una sequenza di **n** incrementi costa **2n**.

Il credito ricordiamo non è mai negativo, perché ogni bit che è stato portato in alto ha una moneta associata che usa quando viene resettato. Quindi il costo reale totale non supera quello ammortizzato totale, che è **2n**, quindi **O(n)**.

## Metodo del potenziale

Invece di utilizzare un **credito associato** ai singoli elementi si utilizza il concetto di **potenziale** associato alla struttura dati nel suo stato corrente. La **funzione potenziale**  $\Phi$  associa a  $D_i$  un numero  $\Phi(D_i)$ , dove  $D_i$  è la struttura dati dopo l'operazione i-esima ( $D_0$  è la struttura dati allo stato iniziale).

Il costo ammortizzato della i-esima operazione è:

$$\tilde{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1}) \text{ dove } c_i \text{ è il costo reale mentre } \Phi(D_i) - \Phi(D_{i-1}) \text{ incremento di potenziale}$$

Il costo ammortizzato totale delle **n operazioni** è dato da:

$\sum_{i=1}^n \hat{C}_i = \sum_{i=1}^n (c_i + \Phi(D_i) - \Phi(D_{i-1})) = \sum_{i=1}^n c_i + \Phi(D_n) - \Phi(D_0)$  dove l'ultima sommatoria di  $c_i$  è il costo reale totale.

Il tutto risulta poi  $\geq \sum_{i=1}^n c_i$  se vale che  $\Phi(D_n) \geq \Phi(D_0)$  dove  $D_0$  è lo stato iniziale e  $D_n$  è quello a valle delle operazioni.

Quindi il costo ammortizzato totale è un limite superiore per il costo reale totale. Se  $n$  non è noto, si suppone che  $\Phi(D_n) \geq \Phi(D_0) \forall i$

### Esempio 1: Pila (inizialmente vuota)

Definiamo:  $\Phi(D_i) = \text{n° di elementi}$

$$\rightarrow \begin{cases} \Phi(D_0) \geq 0 \\ \Phi(D_i) \geq 0 \end{cases} \rightarrow \Phi(D_n) \geq \Phi(D_0) \quad \forall i$$

Il numero di elementi non è mai negativo, valutiamo ora il costo ammortizzato delle varie operazioni:

$$\text{PUSH: } \tilde{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1}) = 1 + (s+1) - s = 2$$

$$\text{POP: } \tilde{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1}) = 1 + (s-1) - s = 0$$

$$\text{MULTIPOP: } \tilde{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1}) = \min(s, k) + (s - \min(s, k)) - s = 0$$

Quindi otteniamo:

	push	pop	multipop
<b>Costo reale</b>	1	1	$\min(s, k)$
<b>Costo ammortizzato</b>	2	0	0

Sulla base del metodo del potenziale non abbiamo bisogno di fare ulteriori verifiche, perché ci è bastato dimostrare che la funzione potenziale a valle di  $n$  operazioni è sempre maggiore o uguale del potenziale iniziale. Si ha ancora una volta che il costo ammortizzato di ogni operazione è  $O(1)$  e quello totale è  $O(n)$ .

### Esempio 2: Contatore binario su $k$ bit (inizialmente nullo)

Definiamo:  $\Phi(D_i) = \text{n° di bit alti} = b_i$

$$\rightarrow \begin{cases} \Phi(D_0) \geq 0 \\ \Phi(D_i) \geq 0 \end{cases} \rightarrow \Phi(D_n) \geq \Phi(D_0) \quad \forall i$$

- $c_i \leq (t_i + 1)$  questo è il costo reale del generico incremento del contatore con  $t_i$  numero di bit resettati, mentre il  $+1$  è l'eventuale bit settato.
  - $\begin{cases} b_i = 0 \rightarrow b_{i-1} = t_i = k \\ b_i > 0 \rightarrow b_i = b_{i-1} - t_i + 1 \end{cases}$   $b_i$  è il numero di bit alti dopo l'incremento, mentre  $b_{i-1}$  il numero di bit alti prima dell'incremento
- $$b_i \leq b_{i-1} - t_i + 1$$
- $\tilde{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1}) = c_i + b_i - b_{i-1} \leq (t_i + 1) - t_i + 1 = 2$

Si ha ancora una volta che il costo ammortizzato di ogni operazione è **O(1)** e quello totale è **O(n)**. Esiste anche un altro tipo di analisi che valuta il contatore che non è inizialmente nullo, ma il costo totale rimane di **O(n)**:

## Lezione 22 04/12/19

### Alberi auto-aggiustanti

#### Splay trees

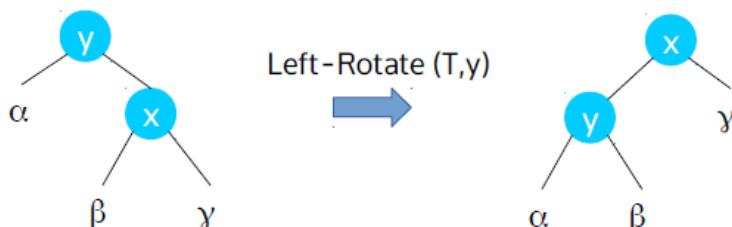
Gli **alberi autoaggiustanti (splay trees)** sono alberi lineari, ma a differenza degli alberi rossoneri, non impongono ulteriori vincoli sulla disposizione degli elementi. Questo chiaramente rende impossibile prevedere l'altezza dell'albero. Ogni qualvolta si accede ad un elemento nell'albero, esso viene spostato nella radice attraverso rotazioni. In questo modo gli elementi a cui si accede più spesso sono più vicini alla radice rispetto a gli altri, avendo così un enorme vantaggio pratico.

Sebbene l'altezza non sia  $\lg n$ , il tempo di esecuzione ammortizzato su una sequenza di  $n$  operazioni è comunque logaritmico.

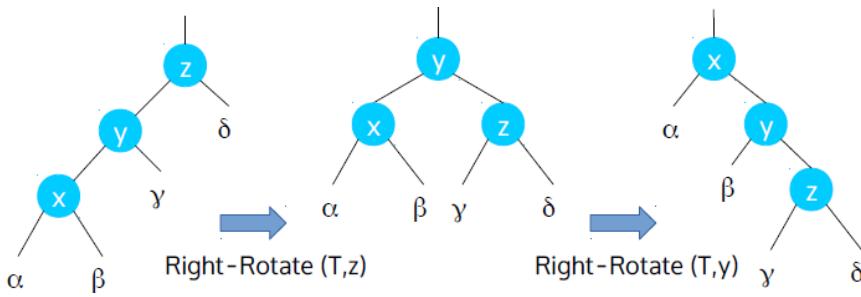
#### Splay

L'operazione di *splay* effettua una serie di rotazioni che spostano il nodo nella radice, così facendo tendiamo a compattare l'albero diminuendo il numero di livelli. Si distinguono 3 casi:

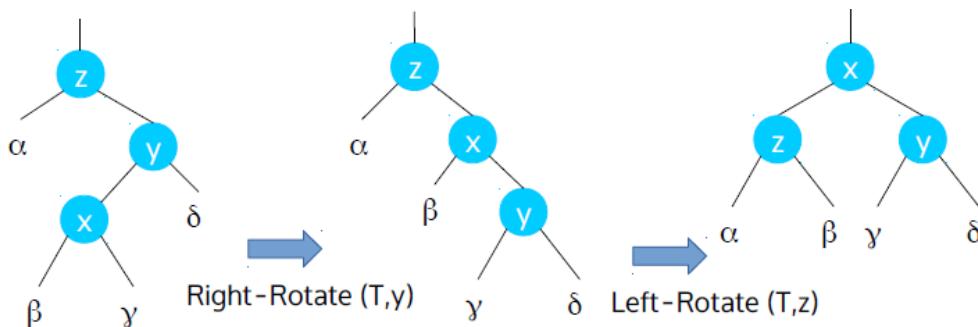
- **Caso 1:** x è figlio della radice
  - Ruotiamo sul padre (verso sinistra se il nodo è a destra, verso destra se il nodo è a sinistra).



- **Caso 2:** x e suo padre sono entrambi figli nello stesso lato (entrambi di sx o entrambi di dx)
  - Ruotiamo sul nonno
  - Ruotiamo sul padre

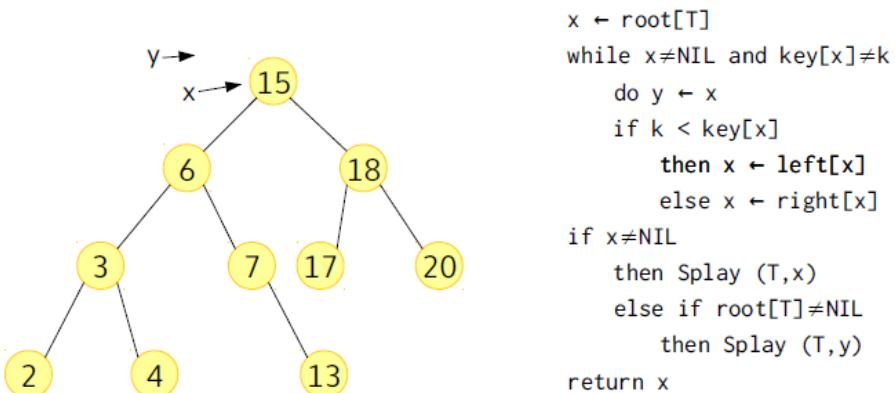


- **Caso 3:**  $x$  e suo padre sono figli di lati opposti (uno a sinistra e uno a destra) (uno a sinistra e uno a destra)
  - Ruotiamo sul padre
  - Ruotiamo sul nuovo padre (l'ex nonno)



## Ricerca

Dopo aver effettuato la ricerca, è invocata la splay sul nodo in cui la ricerca ha dato esito positivo (che è una foglia se l'elemento non è stato trovato). Quindi se l'albero è vuoto non si fa nulla.



## Inserimento ed eliminazione

Dopo l'inserimento è invocata la splay sul nuovo nodo. Dopo l'eliminazione di un nodo  $z$ , la splay è invocata sul padre di  $z$ , se il padre di  $z$  è NIL non si fa nulla.

Inserimento	Eliminazione
<u>Splay-Tree-Insert</u> ( $T, z$ ) Tree-Insert ( $z$ ) Splay ( $T, z$ )	<u>Splay-Tree-Delete</u> ( $T, z$ ) Tree-Delete ( $z$ ) if $p[z] \neq \text{NIL}$ then Splay ( $T, p[z]$ )

## Analisi ammortizzata

Utilizziamo il metodo del potenziale per calcolare il costo ammortizzato di una operazione di splay (che di fatto è una sequenza di  $k$  operazioni). Consideriamo la seguente funzione potenziale:

$$\phi(T) = \sum_{x \in T} r(x) \text{ con } r(x) \text{ rango del nodo } x, \text{ ed è pari al logaritmo del numero di discendenti di } x(x \text{ incluso}): r(x) = \lg d(x).$$

In questo modo, più l'albero è bilanciato più è basso il potenziale. Inoltre, un albero con un solo nodo ( $T_0$ ) ha potenziale 0, perché ha un solo discendente ( $\log 1 = 0$ ). Quindi vale:

$$\begin{aligned} \Phi(T_0) &\geq 0 \\ \Phi(T_i) &\geq 0 \end{aligned} \rightarrow \Phi(T_i) \geq \Phi(T_0) \quad \forall i \text{ dato che non si esegue lo splay su un albero vuoto}$$

Ricordiamoci e teniamo a mente la seguente proprietà:

1.  $r(x) > 1 + \min\{r(y), r(z)\}$  dove  $y$  e  $z$  sono figli di  $x$ .

**Dimostrazione:**

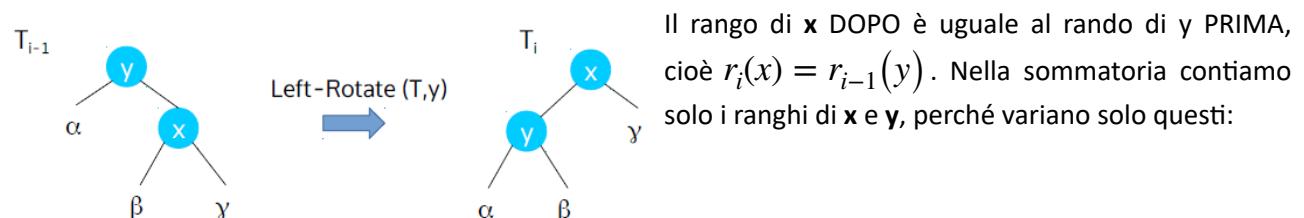
- $d(x) = d(y) + d(z) + 1 \geq 2\min\{d(y), d(z)\} + 1 > 2\min\{d(y), d(z)\}$
- $r(x) = \lg d(x) > \lg 2 + \lg \min\{d(y), d(z)\} = 1 + \min\{\lg d(y), \lg d(z)\} = 1 + \min\{r(y), r(z)\}$

Ora troviamo un limite superiore per la **differenza di potenziale** ( $\Delta\phi_i = \phi(T_i) - \phi(T_{i-1})$ ) tra uno step della splay ed il successivo:

2. Se  $i$  è un passo di un'operazione splay sul nodo  $x$  consideriamo le seguenti 3 proprietà:

- $r_i(x) \geq r_{i-1}(x)$  questa è vera perché la splay opera dal basso verso l'alto, quindi il numero di discendenti cresce ad ogni step.
- Se  $p[x]$  è la radice  $\rightarrow \Delta\phi_i < r_i(x) - r_{i-1}(x)$

**Dimostrazione:**



$$\Delta\phi_i = [r_i(x) - r_{i-1}(x)] + [r_i(y) - r_{i-1}(y)] = r_i(y) - r_{i-1}(x)$$

Nello step successivo  $x$  è padre di  $y$ , quindi ha rango maggiore  $r_i(y) < r_i(x) \rightarrow \Delta\phi_i < r_i(x) - r_{i-1}(x)$

- $p[x]$  non è la radice  $\rightarrow \Delta\phi_i < 3(r_i(x) - r_{i-1}(x)) - 1$  solo se  $x, y$  e  $z$  cambiano di rango.

A questo punto possiamo dire che: *Il costo ammortizzato di una operazione splay in un albero auto-aggiustante con n nodi è  $O(\lg n)$ .*

Il singolo passo è una semplice rotazione  $c_i = O(1)$  quindi il costo ammortizzato di un passo dell'operazione **splay** su **x** è :

$$\hat{c}_i = c_i + \phi(T_i) - \phi(T_{i-1}) = O(1) + \Delta\phi_i < \begin{cases} p[x] \text{ è radice } O(1) + r_i(x) - r_{i-1}(x) \\ \text{altrimenti } O(1) + 3[r_i(x) - r_{i-1}(x)] - 1 \end{cases} < O(1) + 3[r_i(x) - r_{i-1}(x)]$$

Se consideriamo tutti i **k** passi otteniamo il costo ammortizzato totale:

$$\sum_{i=1}^k \hat{c}_i < O(1) + 3 \sum_{i=1}^k [r_i(x) - r_{i-1}(x)] = O(1) + 3[r_k(x) - r_0(x)] = \text{dove } r_k(x) \text{ vale } \lg n \text{ mentre } r_0(x) \text{ vale } 0 \text{ nel caso peggiore}$$

$$= O(1) + 3\lg n = O(\lg n) \text{ In quest'ultimo passaggio abbiamo considerato che:}$$

- Alla fine della splay (dopo k passi) **x** è la radice, quindi ha **n** discendenti, quindi il rango è **lg n**
- Nel caso peggiore si parte dalla foglia, che ha rango **0**

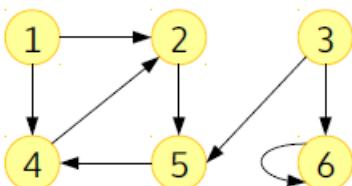
Abbiamo quindi dimostrato che il costo ammortizzato è  **$O(\lg n)$** .

Si può inoltre dimostrare che il costo ammortizzato di **m** operazioni di insert, search e delete è  **$O(m \lg n)$** , con **n** numero massimo dei nodi durante le operazioni. Quindi il costo ammortizzato di una singola operazione è  **$O(\lg n)$**

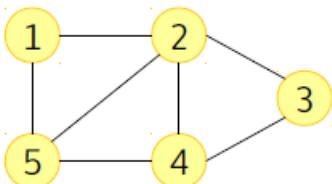
## Grafi

Un **grafo G** consiste in un insieme di nodi **V** connessi mediante un insieme di archi **E**. I grafi possono essere:

- Direzionali (gli archi hanno un unico verso).



- Non direzionali.



Esistono poi due principali modi di rappresentare i grafi, indifferentemente dal fatto che sono o no direzionali:

- Liste di adiacenza (tipicamente preferita per grafi sparsi)

- Matrice di adiacenza (tipicamente preferita per sapere se due nodi sono connessi)

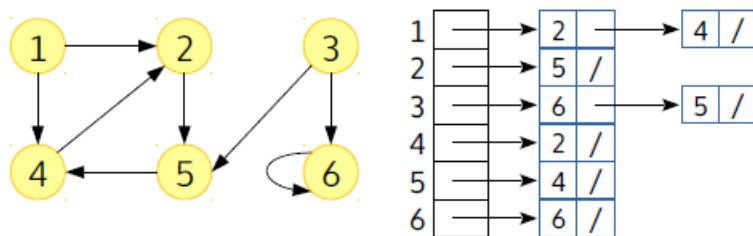
## Liste di adiacenza

La rappresentazione mediante liste di adiacenza, come detto precedentemente, è preferita per grafi sparsi ovvero con pochi archi  $|E| \ll |V|^2$  ed è vantaggioso per le memorie poiché crea nodi solo per quello che gli è necessario.

Consiste in un array **Adj** (adjacency) di  $|V|$  liste, una per ogni nodo **u**:  
 $\forall u \text{ Adj}[u] = \{ \text{nodi } v : (u, v) \in E \}$

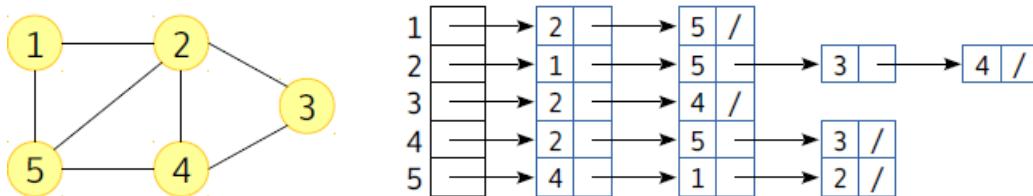
Nel caso **direzionale**:

- Ogni lista **Adj[u]** contiene i nodi puntati da **u**
- Il numero di elementi nelle liste è pari al numero di archi:  $|E|$



Nel caso **non direzionale**:

- Ogni lista **Adj[u]** contiene i nodi vicini di **u**
- Se **v** è nella lista di **u**, **u** è nella lista di **v**
- Il numero di elementi nelle liste è pari al doppio del numero di archi:  $2|E|$



In ogni caso, la memoria richiesta è proporzionale, come detto precedentemente, al numero di nodi il quale influisce sul numero delle liste, e al numero degli archi (che influisce sulla lunghezza delle liste). Quindi la quantità di memoria richiesta è  $\theta(V + E)$ .

Le liste di adiacenza possono anche essere usate in **grafi pesati**, in cui ad ogni arco è associato un peso. In questo caso, il peso **w** dell'arco **(u,v)** è memorizzato insieme a **v** nella lista di **u**.

**Vantaggio:** è economico in termini di memoria

**Svantaggio:** per vedere se un arco **(u,v)** è presente, bisogna effettuare una ricerca in una lista di adiacenza **u**.

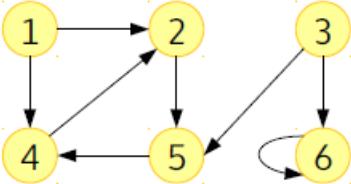
## Matrice di adiacenza

La rappresentazione mediante matrice di adiacenza è preferibile per grafi densi e di piccole dimensioni, o quando serve sapere velocemente se due nodi sono connessi.

Questa rappresentazione consiste in una matrice  $\mathbf{A}$  di dimensione  $|V| \times |V|$  in cui le righe e le colonne rappresentano i nodi. Ogni elemento della matrice vale 1 se l'arco è presente:

$$a_{ij} = \begin{cases} 1 & \text{se } (i,j) \in E \\ 0 & \text{altrimenti} \end{cases}$$

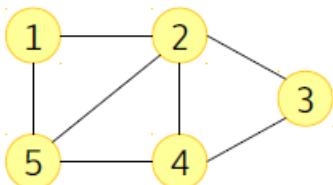
Nel caso **direzionale**:



	1	2	3	4	5	6
1	0	1	0	1	0	0
2	0	0	0	0	1	0
3	0	0	0	0	1	1
4	0	1	0	0	0	0
5	0	0	0	1	0	0
6	0	0	0	0	0	1

Nel caso **non direzionale**:

- La matrice è simmetrica rispetto alla diagonale  $\rightarrow A = A^T$ .
- Dunque si può dimezzare il consumo di memoria



	1	2	3	4	5
1	0	1	0	0	1
2	1	0	1	1	1
3	0	1	0	1	0
4	0	1	1	0	1
5	1	1	0	1	0

Anche la matrice di adiacenza può essere usata per grafi pesati, indicando nella matrice il peso dell'arco piuttosto che il simbolo 1. Tuttavia, è più usata per i grafi non pesati, poiché ogni elemento è memorizzato su un solo bit. Possiamo avere al massimo  $n(n-1)$  archi, oppure  $n(n)$  se si considera anche il cappio (ripiegamento su se stesso).

**Vantaggio:** La verifica della presenza di un arco è immediata

**Svantaggio:** La quantità di memoria richiesta è sempre  $\Theta(V^2)$ , per cui si usa su grafi di piccole dimensioni, indipendentemente dal numero di archi.

## Visita in ampiezza

La **visita in ampiezza** del grafo (**BFS** - Breadth First Search) è un algoritmo usato per visitare tutti i nodi di un grafo raggiungibili a partire da uno specifico nodo  $s$ .

BFS visita tutti i nodi adiacenti ad  $s$ , poi passa ai nodi adiacenti ad essi e così via, con il vincolo che tutti i nodi distanti  $k$  dalla sorgente sono visitati prima di quelli che distano  $k+1$ . Il risultato è un albero la cui radice è  $s$  e che contiene tutti i nodi raggiungibili da  $s$ . (Gli alberi sono particolari grafi privi di cicli)

Un importante proprietà è che il percorso da  $s$  ad un nodo  $v$  dell'albero corrisponde anche al più corto tragitto nel grafo tra  $s$  e  $v$ .

Questo approccio è adottato da alcuni algoritmi, come:

- Algoritmo di Dijkstra per lo shortest path
- Algoritmo di Prim per il minimum-spanning-tree (algoritmo per il networking ed evita i cicli di broadcast)

## Lezione 23 05/12/19

Durante la BFS, ogni nodo può essere colorato di:

- **Bianco** (non visitato)
- **Nero** (visitato, vicini visitati)
- **Grigio** (visitato, vicini non visitati)

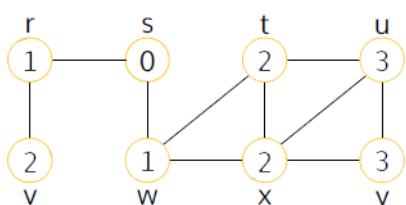
Inoltre, se il nodo **v** è visitato perché adiacente a **u**, si dice che **u** è **il predecessore** di **v**:

- Il predecessore della sorgente sarà sempre NIL
- I nodi non ancora scoperti hanno predecessore NIL

Consideriamo anche un attributo **distanza**, che indica la distanza dalla sorgente:

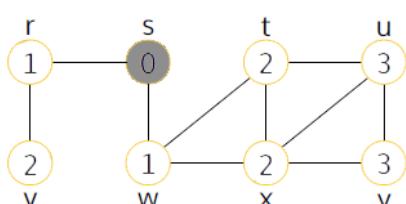
- La sorgente ha distanza 0
- I nodi non ancora scoperti hanno distanza  $\infty$
- I nodi scoperti hanno distanza pari alla distanza del predecessore +1

Es: consideriamo il seguente grafo (**nell'immagine i numeri nei nodi ne rappresentano la distanza, inizialmente dovrebbero avere tutti valore  $\infty$  !!!!**)

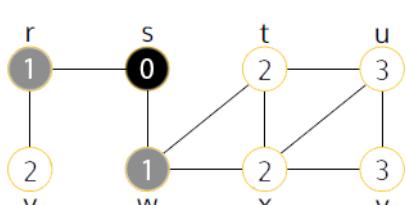


L'albero che si ottiene dipende dall'ordine con cui si visitano i vicini, ma la lunghezza dei percorsi minimi non cambia.

È necessario mantenere in una coda i nodi che sono in attesa di esplorazione, partendo dall'esempio iniziamo a colorare di grigio il nodo **s**.

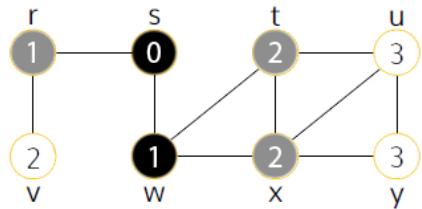


Ad ogni step noi preleviamo il nodo in testa alla coda e vi aggiungiamo i vicini bianchi: **w** ed **r** in questo caso.

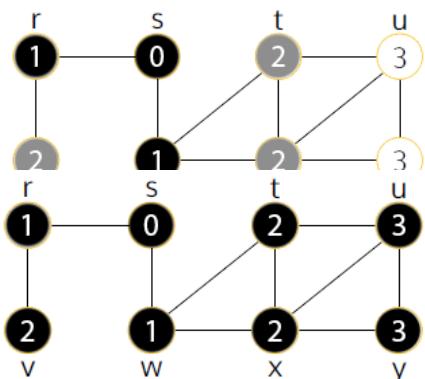


I nodi in coda vengono colorati di grigio mentre **s** diventa nero, poiché lo abbiamo visitato e abbiamo visitato tutti i vicini. A questo punto estraiamo uno dei due nodi dalla coda, non cambia nulla se avessimo preso prima **r** rispetto a **w** cambiava solo il come veniva rappresentato

l'albero ma l'algoritmo risultava invariato.



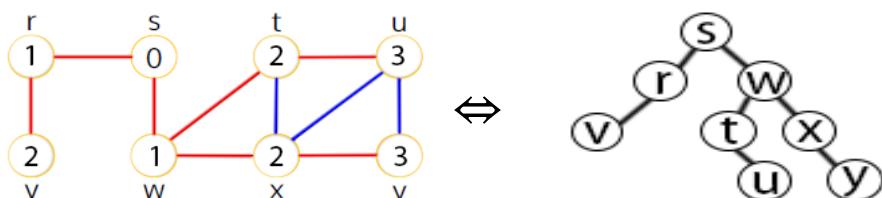
L'ordine di come vengono esplorati i vicini viene dettato da come viene memorizzato il grafo, se ad esempio è tramite liste di adiacenza, converrà esplorare in base all'ordine della lista.



Prendiamo il prossimo nella coda, ovvero **r**, quindi esploriamo i vicini di **r** e inseriamo all'interno il nodo **v**, ma ovviamente dopo **t** ed **x**.

Si arriverà infine, alla situazione dove tutti i nodi sono neri e quindi la lista sarà vuota. Gli attributi di distanza e predecessore andranno settati ovviamente, solo quelli che esploriamo, dunque se al termine della BFS un nodo ha ancora distanza infinita ed il predecessore NIL allora questo nodo probabilmente non è connesso oppure non esiste un percorso direzionale.

Notiamo come la coda contenga sempre nodi grigi, e l'ordine di visita dipende dall'ordine in cui sono inseriti nella lista. Gli archi rossi collegano ogni nodo al suo predecessore e costituiscono il sottografo (che è un albero) che contiene i percorsi minimi, mentre quelli blu sono dei predecessori "secondari" ovvero se avessimo esplorato prima l'altro nodo:



## Pseudocodice

Consideriamo la rappresentazione a liste di adiacenza. In dichiamo il predecessore con  $\pi$  (che risulta null se il nodo è la sorgente oppure se non è stata aggiunta ancora) e la distanza con  $d$ .

Enqueue indica l'inserire in una cosa, mentre Dequeue è l'estrazione. L'invariante è che all'inizio di ogni iterazione del ciclo while, tutti i nodi di  $Q$  sono grigi:

- È vero prima della prima iterazione
- Ogni iterazione conserva tale proprietà

La fase di inizializzazione è  $O(V)$ , perché viene effettuata per ogni nodo. Usiamo la  $O$  piuttosto che la  $\Theta$  perché alcuni nodi potrebbero non essere scoperti.

```

BFS (G, s)
for each vertex  $u \in V[G] - \{s\}$ 
    do  $\text{color}[u] \leftarrow \text{WHITE}$ 
     $d[u] \leftarrow \infty$ 
     $\pi[u] \leftarrow \text{NIL}$ 
 $\text{color}[s] \leftarrow \text{GRAY}$ 
 $d[s] \leftarrow 0$ 
 $\pi[s] \leftarrow \text{NIL}$ 
 $Q \leftarrow \emptyset$ 
Enqueue ( $Q, s$ )
while  $Q \neq \emptyset$ 
    do  $u \leftarrow \text{Dequeue } (Q)$ 
    for each  $v \in \text{Adj}[u]$ 
        do if  $\text{color}[v] = \text{WHITE}$ 
            then  $\text{color}[v] \leftarrow \text{GRAY}$ 
             $d[v] \leftarrow d[u] + 1$ 
             $\pi[v] \leftarrow u$ 
            Enqueue( $Q, v$ )
    color[u]  $\leftarrow \text{BLACK}$ 

```

Il tempo legato all'analisi delle liste di adiacenza è  $O(E)$ , questo perché è fatto per tutte le iterazione del while e quindi è pari alla somma delle lunghezze di tutte le liste.

$O(V)$  Se usiamo un'analisi aggregata giungiamo che il tempo di esecuzione di BFS è  $O(V+E)$ .

## Proprietà

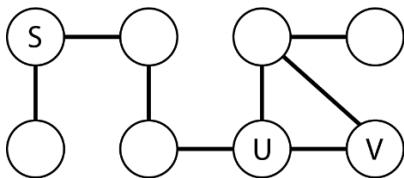
Indichiamo la distanza minima (ovvero uno *shortest-path*) con  $\delta(s,v)$  e rappresenta il numero minimo di archi di qualunque percorso tra  $s$  e  $v$

*Lemma 1:*  $\forall (u,v) \in E$  si ha  $\delta(s,v) \leq \delta(s,u) + 1$

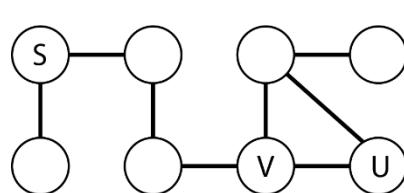
Per dimostrarlo dobbiamo distinguere due casi:

### Dimostrazione:

- Se  $u$  è raggiungibile da  $s$ , allora lo è anche da  $v$ . Poiché c'è un arco che collega  $u$  e  $v$ , nel peggior dei casi (in cui  $v$  è più lontano di  $u$ )  $v$  può essere raggiunto passando per  $u$  più quell'arco.

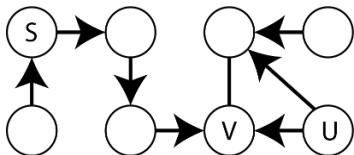


$$\delta(s,v) = \delta(s,u) + 1$$



$$\delta(s,v) < \delta(s,u) + 1$$

- Se  $u$  non è raggiungibile da  $s$ , la sua distanza è  $\infty$ , quindi  $v$  è sicuramente più vicino.



*Lemma 2:* Dopo la **BFS** ( $G,s$ )  $\forall v \in V \delta(s,v) \leq d[v]$

Siccome stiamo esprimendo una proprietà dell'attributo distanza, ci ricordiamo che la distanza si imposta quando estraiamo un nodo dalla coda e visitiamo i suoi vicini bianchi, il predecessore e la distanza del nodo che andiamo a visitare vengono settati.

### Dimostrazione:

Procediamo per induzione sul numero di Enqueue.

Caso base:  $d[s] = 0 = \delta(s,s)$

- accodiamo  $s$ ,  $d[v] = \infty \geq \delta(s,v)$

- Passo induttivo: Consideriamo un vertice viano  $v$  scoperto durante la ricerca dei vicini di  $u$ .  $d[v] = d[u]$  (che è il predecessore) + 1  $\geq \delta(s, u) + 1$  supponendo che sia vero per  $u \geq \delta(s, v)$  per il **lemma 1**. Il vertice  $v$  viene colorato di grigio e quindi non viene più riaccodato, per cui la sua distanza non cambia più

**Lemma 3:** Dopo la **BFS**, se  $Q$  contiene i vertici testa  $v_1$  e coda  $v_r: <v_1, v_2, \dots, v_r>$

$$\rightarrow \forall i = 1, \dots, r. \quad d[v_i] \leq d[v_{i+1}] \text{ è crescente e che } d[v_r] \leq d[v_i] + 1$$

#### Dimostrazione:

Procediamo per induzione sul numero di operazioni sulla coda

- Caso base: inizialmente la coda contiene  $s$  e la tesi è valida  $<v_s>$
- Passo induttivo: 1) Dequeue ( $v_2$  è la nuova testa):

$$\text{Supp. vero per } v_1 \rightarrow d[v_1] \leq d[v_2] \leq \dots$$

Dim per  $v_2$ :

$$d[v_r] \leq d[v_1] + 1 \leq d[v_2] + 1$$

2) Enqueue ( $v_r$  è la nuova coda,  $u$  è il predecessore scartato prima di aggiungere  $v_{r+1}$ )

$$\text{Supp. vero per } u \rightarrow d[u] \leq d[v_1] \leq \dots \text{ e per } d[v_r] \leq d[u] + 1$$

Dim per  $v_{r+1}$ :

$$d[v_{r+1}] = d[u] + 1 \leq d[v_1] + 1$$

$$d[v_r] \leq d[u] + 1 = d[v_{r+1}], \text{ le altre relazioni restano immutate.}$$

**Corollario 4:** Se il vertice  $v_i$  è accodato prima di  $v_j$ , allora  $d[v_i] \leq d[v_j]$

#### Dimostrazione:

Discende dal lemma 3 e dal fatto che le distanze vengono assegnate una sola volta e sono maggiori di quelle del predecessore, che è accodato prima nella coda.

**Teorema 5:** BFS visita tutti i nodi raggiungibili da  $s$  se al termine  $d[v] = \delta(s, v) \quad \forall v$ . Inoltre, per ogni  $v$  raggiungibile da  $s$ , uno shortest path da  $s$  a  $v$  è uno shortest path da  $s$  al predecessore  $\pi[v]$  più l'arco  $(\pi[v], v)$ .

In altre parole, vogliamo dimostrare che l'albero che si ottiene con la BFS contiene effettivamente i percorsi minimi (cioè il lemma 2, ma solo con l'uguaglianza) verso tutti i nodi raggiungibili.

#### Dimostrazione PER ASSURDO:

Assumiamo per assurdo che qualche vertice riceva una distanza diversa da quella minima. Tra questi, consideriamo il nodo  $v$  con la distanza minima più piccola (che non può essere  $s$ , che ha distanza 0)

Per il lemma 2:  $\delta(s, v) \leq d[v] \rightarrow d[v] > \delta(s, v)$   $v$  è raggiungibile da  $s$ , altrimenti si avrebbe  $d[v] > \infty$ .

Se  $u$  precede  $v$  lungo uno shortest path, si ha:

- $d[u] = \delta(s, v)$
- $\delta(s, v) = \delta(s, u) + 1 = d[v] + 1 \rightarrow d[v] > d[u] + 1$

Ora quando viene estratto  $u$  dalla coda,  $v$  può essere:

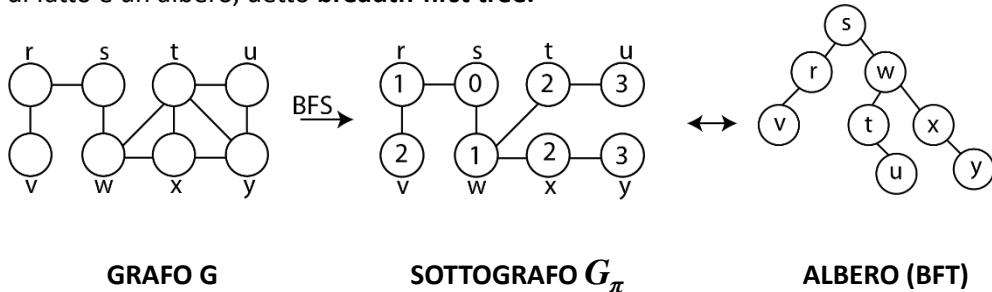
- Bianco, ma poiché c'è un link tra  $u$  e  $v$ , cioè implica che  $v$  viene scoperto da  $u$  e quindi si contraddice  $d[v] = d[u] + 1$
- Nero, cioè è stato estratto prima di  $u$  e per il corollario 4 si ha:  $d[v] \leq d[u]$  ma è contraddittorio sempre per  $d[v] = d[u] + 1$
- Grigio, vuol dire che era stato già scoperto da un altro nodo  $w$  estratto prima di  $u$ , quindi  $d[w] \leq d[u]$  e:  $d[v] = d[w] + 1 \leq d[u] + 1$  contraddittorio.

Quindi è dimostrato per assurdo che  $d[v] = \delta(s, v) \forall v$ . Questo dimostra anche che tutti i nodi raggiungibili sono visitati, o avrebbero distanza infinita, che è diversa da quella minimia

Infine, se  $\pi[v] = u$ , allora il BFS ha settato  $d[v] = d[u] + 1$ . Essendo la minima distanza, stiamo considerando lo shortest path da  $s$  a  $u$  si può ottenere prendendo uno shortest path da  $s$  a  $\pi[v]$  e aggiungendo l'arco  $(\pi[v], v)$ .

### Predecessor subgraph

Come già accennato, la BFS genera un sottografo dei percorsi minimi, detto **predecessor subgraph** ( $G_\pi$ ), che di fatto è un albero, detto **breadth-first tree**.



$$G_\pi = (V_\pi, E_\pi) \text{ dove } V_\pi = \{v \in V : \pi[v] \neq \text{NIL}\} \cup \{s\} \text{ e } E_\pi = \{(\pi[v], v) : v \in V_\pi - \{s\}\}$$

$G_\pi$  è un albero perché soddisfa la condizione necessaria e sufficiente per un albero:

$$|E| = |V| - 1$$

In realtà  $G_\pi$  è un breadth-first tree se:

- $V\pi$  consiste dei vertici raggiungibili da  $s$
- Per ogni nodo  $v \in V\pi$ , il suo percorso  $G_\pi$  è anche lo shortest path  $G$ .

*Lemma 6:* Il predecessor subgraph restituito da BFS è un breadth-first tree.

**Dimostrazione:**

- BFS setta  $\pi[v]=u$  se e solo se  $(u,v) \in E$  e  $v$  è raggiungibile da  $s$ , per definizione di  $V\pi$  ogni  $v$  ha predecessore nullo.
- Applicando il teorema 5 in maniera induttiva, si conclude che ogni percorso è uno shortest path

## Stampa del percorso

Dopo la BFS, può essere stampato il path da  $s$  a  $v$ .

```
Print-Path (G,s,v)
if v=s
    then print s
else if π[v] = NIL
    then print "no path from s to v exists"
    else Print-Path (G,s,π[v])
        print v
```

Poiché  $v$  è stampato dopo la chiamata ricorsiva, il percorso è stampato in ordine da  $s$  a  $v$ . Il tempo di esecuzione è lineare nel numero di vertici che costituiscono il percorso.

## Lezione 24 11/12/19

### Visita in profondità

La **visita in profondità del grafo** nota come (**DFS** – Depth First Search) è l'algoritmo usato per visitare **tutti i nodi** del grafo, non solo quelli raggiungibili dalla sorgente.

DFS visita i nodi adiacenti al nodo scoperto più recentemente che ha ancora vicini inesplorati. Se rimangono nodi non esplorati, la ricerca riparte da quelli. Per questo motivo il predecessor subgraph (il quale potrebbe avere più alberi distinti) può essere composto da più alberi (foresta ovviamente). Poiché tutti i vertici vengono esplorati, non c'è un ndo sorgente ed il predecessor subgraph è definito in manira diversa rispetto a prima:

$$G_\pi = (V, E_\pi) \text{ e } E_\pi = \{( \pi(v), v ) : v \in V \text{ and } \pi(v) \neq \text{NIL}\}$$

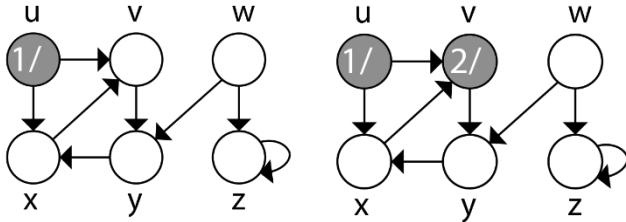
Anche durante la DFS ad ogni nodo è associato un colore, identico a quelli della BFS.

Il concetto di **distanza però cambia** e viene sostituito da **timestamp**:

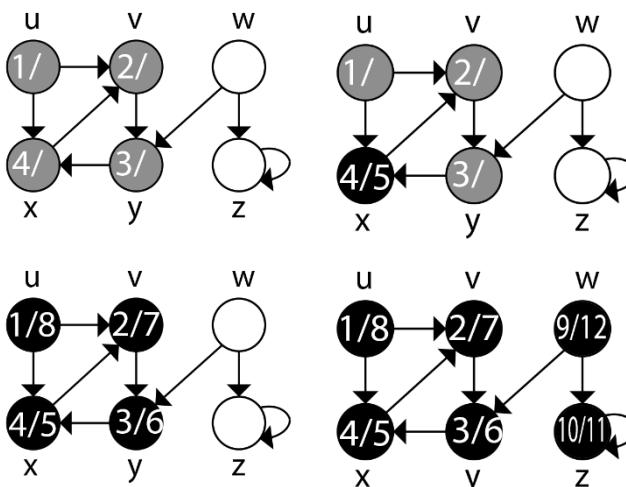
- **d[v]**, che marca il tempo in cui è scoperto (quando diventa **grigio**)
- **f[v]**, che marca il tempo nel quale termina la ricerca dei vicini (quando diventa **nero**).

Poiché per ogni nodo ci sono 2 timestamp, che sono valori interi, il timestamp parte da **1** e arriva a  **$2|V|$** . Ovviamente si ha che  $d[v] < f[v] \forall v$

Vediamo un esempio del DFS usando un grafo direzionale partendo casualmente da **u** con tempo di scoperta  **$d[u]=1$**  e lo coloriamo di grigio, procediamo poi indifferentemente dall'ordine come per il BFS.



Esploriamo **v** con un tempo di scoperta pari a **2** e lo coloriamo di grigio, procediamo così fino a scoprire **x**.



Abbiamo scoperto **x** al tempo **4**, esploriamo i suoi vicini i quali però sono tutti già stati esplorati, questo significa che segniamo il tempo di fine visita  **$f[x] = 5$** . Ora torniamo al predecessore **y** il quale non ha altri vicini da esplorare e quindi il suo  **$f[y] = 6$**  e continuiamo a ritroso.

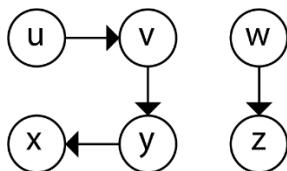
Non essendoci più vicini esplorati di recente parte un'altra volta la DFS da uno dei due nodi non ancora scoperti. Ripetiamo l'algoritmo e completiamo la visita.

Si nota che, a differenza della BFS, i vicini di un nodo non diventano grigi tutti in una volta, ma si sceglie solo un vicino e si continua da lì allo stesso modo.

Inoltre:

- La BFS visita **solo** i nodi raggiungibili (quindi si ha un albero)
- La DFS visita **tutti** i nodi (quindi si può avere una foresta)

Risulta quindi la foresta:



Pseudocodice

```

DFS ( $G$ )
for each vertex  $u \in V[G]$ 
    do  $\text{color}[u] \leftarrow \text{WHITE}$ 
         $\pi[u] \leftarrow \text{NIL}$ 
     $\text{time} \leftarrow 0$ 
for each vertex  $u \in V[G]$ 
    do if  $\text{color}[u] = \text{WHITE}$ 
        then DFS-Visit ( $u$ )

```

Visto che si ritorna a visitare un nodo, l'algoritmo presenterà della ricorsione, quindi scriviamo l'algoritmo in due parti che è la **DFS-Visit**.

Per essere certi che tutti i nodi vengano scoperti facciamo una *for* su tutti i nodi bianchi.

Per lo stesso ragionamento fatto nella visita in ampiezza, la complessità risulta essere  **$O(V+E)$**  con crescita lineare.

```

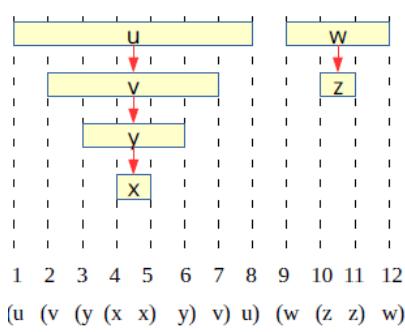
DFS-Visit (u)
color[u] ← GRAY
time ← time+1
d[u] ← time
for each v ∈ Adj[u]
    do if color[v] = WHITE
        then  $\pi[v]$  ← u
            DFS-Visit (v)
color[u] ← BLACK
f[u] ← time ← time+1

```

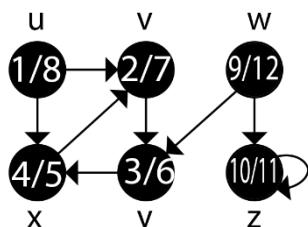
### Proprietà

Una proprietà della DFS è che i tempi di scoperta e di terminazione dei nodi (i **timestamp**) hanno una struttura a parentesi, nel senso che:

- $d[v]$  apre le parentesi
- $f[v]$  chiude le parentesi



Se scriviamo  $d[u]$  come  $u$  e  $f[u]$  come  $u$  otteniamo un'espressione corretta



**Teorema 7:** Per ogni coppia di nodi **u** e **v**, si può verificare solo una delle seguenti condizioni:

- Gli intervalli di visita di **u** e **v** sono disgiunti, nessuno dei due discende dall'altro.
- L'intervallo di visita di **u** è contenuto interamente in quello di **v**, **u** è discendente di **v**.
- L'intervallo di visita di **v** è contenuto interamente in quello di **u**, **v** è discendente di **u**.

In altri termini:

- Gli intervalli di visita sono totalmente contenuti l'uno nell'altro o sono disgiunti. Non può mai accadere che un intervallo sia solo parzialmente contenuto in un altro intervallo.
- Nel primo caso, il nodo "contenuto" è discendente del nodo "contenitore".

### Dimostrazione:

Supponiamo che **u** sia più vecchio di **v**:  $d[u] < d[v]$ .

Quando **v** è scoperto, **u** può essere o ancora grigio o già nero.

Caso 1:  $d[v] < f[u]$

$v$  è scoperto prima che  $u$  finisce, quindi prima che  $u$  sia nero, cioè quando è ancora grigio.

Di conseguenza,  $v$  è un discendente di  $u$ . Si ha quindi che  $v$  diventerà nero prima di  $u$ , cioè  $f[v] < f[u]$

Quindi abbiamo  $d[u] < d[v] < f[v] < f[u]$  cioè l'intervallo di visita di  $v$  è contenuto in quello di  $u$ .

### Caso 2: $d[v] > f[u]$

$v$  è scoperto dopo che  $u$  finisce, quindi quando  $u$  è nero, quindi nessuno discende dall'altro.

*Corollario 8:* Un vertice  $v$  è un discendente di  $u$  nella depth-first forest se e solo se  $d[u] < d[v] < f[v] < f[u]$

*Teorema 9:* In una depth-first forest, un vertice  $v$  è un discendente di  $u$ , se e solo se, al tempo  $d[u]$  (quando  $u$  è scoperto) il percorso da  $u$  a  $v$  contiene solo nodi

#### Dimostrazione:

$\Rightarrow$  )  $v$  è un discendente di  $u$ .

Qualsiasi nodo  $w$  tra  $u$  e  $v$  discende da  $u$ , quindi (corollario 8)  $d[u] < d[w]$ , cioè  $w$  diventa grigio dopo di  $u$  (il quale è ancora bianco quando  $u$  è scoperto).

$\Leftarrow$  ) Al tempo  $d[u]$  il percorso da  $u$  a  $v$  contiene solo nodi bianchi.

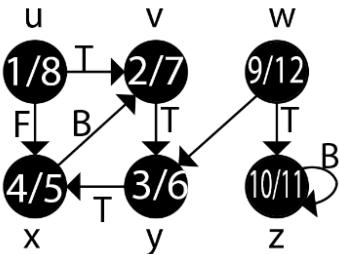
Lungo il percorso, i nodi da  $u$  a  $v$  (tranne  $v$ ) diventano discendenti di  $u$ .  $w$ , il predecessore di  $v$ , diventa discendente di  $u$  (o è  $u$  stesso), quindi per il corollario 8  $f[w] \leq f[u]$ .  $v$  è scoperto dopo  $w$  quindi  $d[u] \leq d[w] < d[v] < f[v] < f[w] \leq f[u]$ . Per il teorema 7, l'intervallo di visita di  $v$  non può essere contenuto solo parzialmente in quello di  $u$ , quindi è interamente contenuto. Dunque per il corollario 8  $v$  è discendente di  $u$ .

### Classificazione degli archi

La DFS partiziona gli archi di un grafo in 4 insiemi, in base a come procede la visità in profondità:

<b>Tree edges</b>	$(u,v)$ è un tree edge se $v$ è scoperto da $u$ (quindi se fa parte della foresta dopo la DFS)
<b>Back edges</b>	$(u,v)$ è un back edge se $v$ è un antenato di $u$ (sono inclusi i self-loop)
<b>Forward edges</b>	$(u,v)$ è un forward edge se $v$ è un discendente di $u$
<b>Cross edges</b>	Tutti gli altri

Per i grafi non direzionali, ci possono essere delle ambiguità tra  $(u,v)$  e  $(v,u)$ , che sono lo stesso arco.



La classificazione può essere fatta stessa dalla DFS man mano che incontra un nodo  $v$ , a seconda del suo colore:

- Bianco ( $u,v$ ) è un **Tree edge**
- Grigio ( $u,v$ ) è un **Back edge** (perché è stato già scoperto in precedenza)
- Nero:
  - ( $u,v$ ) è un **forward edge** se  $d[u] < d[v]$
  - ( $u,v$ ) è un **cross edge** se  $d[u] > d[v]$

Per i grafi non direzionali, ci possono essere solo tree edge e back edge.

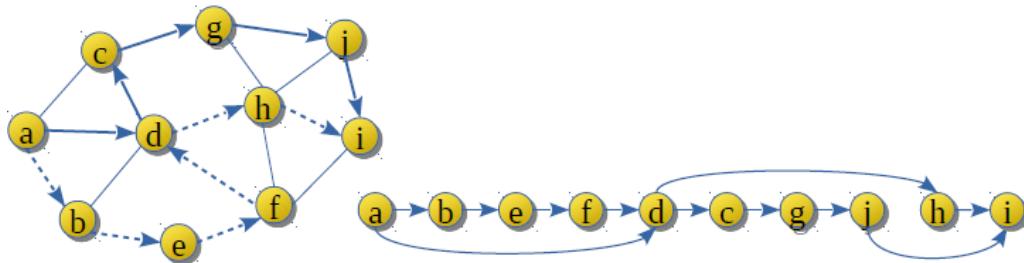
## Ordinamento topologico

Se un grado direzionale non presenta cicli (DAG – Directed Acyclic Graph), la DFS può essere usata per realizzare un **ordinamento topologico**.

Può ad esempio essere usato quando ogni nodo rappresenta un task che deve essere eseguito prima di un altro, quindi rappresenta una relazione di precedenza. In questo modo individuiamo come eseguire i task per rispettare le dipendenze.

L'ordine coincide con l'inverso dei tempi di fine visita, cioè ogni volta che un nodo diventa nero è inserito a "sinistra" nella lista.

Ad esempio partendo dal nodo  $a$ , un possibile ordinamento è:



### Topological-Sort (G)

```
Call DFS(G) to compute finishing times f[v] for each vertex v
As each vertex is finished, insert it onto the front of a linked list
return the linked list of vertices
```

Chiamiamo la  $\text{DFS}(G)$  per trovare i  $f[v]$  per ogni vertice  $v$ . Ogni volta che un vertice è completato, va inserito in testa alla lista. Ritorna la lista concatenata dei vertici.

La  $\text{DFS}$  richiede  $\theta(V + E)$ , l'inserimento di  $|V|$  nella lista richiede  $O(V)$ , dunque il tempo di esecuzione dell'ordinamento topologico è  $\theta(V + E)$ .

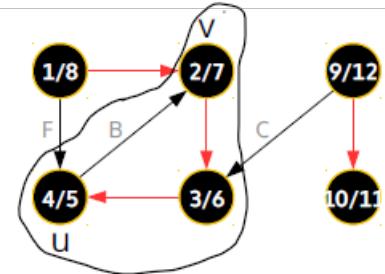
*Lemma 11:* Un grafo direzionale è aciclico se e solo se una DFS su G non risuitisce back edges.

**Dimostrazione:**

$\Rightarrow$  ) Il grafo è aciclico

Se c'è un back edge  $(x, v)$ , vuol dire che  $v$  è antenato di  $x$ , che implica che c'è un percorso da  $v$  a  $x$ . Il back edge collega  $x$  di nuovo a  $v$ , quindi c'è un ciclo: CONTRADDIZIONE.

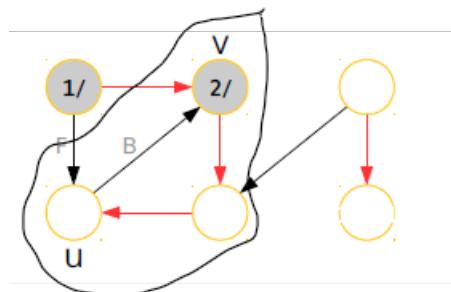
Quindi non possono esserci back edges.



$\Leftarrow$  ) Il grafo non ha back edges.

Se c'è un ciclo  $c$ , quando è scoperto per la prima volta un nodo del ciclo (supponiamo  $v$ ) gli altri nodi del ciclo sono bianchi. Quindi per il teorema 9 il nodo  $x$  che precede  $v$  nel ciclo sarà discendente di  $v$ , quindi  $(x, v)$  è un back edge: CONTRADDIZIONE

Quindi non ci sono cicli:



*Teorema 12: Topological-Sort (G) produce un ordinamento*

topologico di un directed acyclic graph G

**Dimostrazione:**

Consideriamo l'arco  $(u, v)$ , Occorre mostrare che, per ogni coppia di vertici  $u$  e  $v$ , allora  $f[v] < f[u]$

- $v$  non può essere grigio, altrimenti  $(u, v)$  sarebbe un back edge e introdurrebbe un ciclo
- Se  $v$  è bianco, diventa discendente di  $u$  e si ha  $f[v] < f[u]$
- Se  $v$  è nero, la sua visita è già finita e si ha ancora  $f[v] < f[u]$

## Lezione 25 12/12/19

### Crittografia RSA

### Teoria dei numeri

Nelle applicazioni della teoria dei numeri, vengono usati numeri molto grandi pari a centinaia di migliaia di bit per rappresentare gli input, piuttosto che il numero di input. Le operazioni anche più semplici devono essere effettuate attraverso specifici algoritmi in funzione del numero di bit.

Una complessità accettabile per gli algoritmi relativi alla teoria dei numeri è quindi  $\lg n$ , dove  $n$  è il valore in ingresso. Infatti  $\lg n$  rappresenta il numero di bit necessari per rappresentare l'ingresso  $n$ .

In particolare, la complessità è valutata in base al numero di operazioni su bit. Operazioni anche elementari che possono essere poco efficienti su numeri molto grandi (la moltiplicazione di due interi a  $k$  bit è  $\Theta(k^2)$ )

## Teoremi ed altre nozioni

**Simbolo | :**

il simbolo  $d|a$  indica “ $d$  divide  $a$ ” se  $a = kd$  (con  $k$  intero). Elenchiamo alcune sue proprietà:

- $a$  è multiplo di  $d$
- Se  $a > 0$ , allora  $|d| \leq |a|$
- Ogni intero divide 0
- $d|a \leftrightarrow -d|a$
- Se  $d \geq 0$ ,  $d$  è un divisore di  $a$
- Un divisore di un intero  $a$  non nullo è compreso tra 1 e  $|a|$
- Ogni intero  $a$  è sempre divisibile per 1 e  $a$  (divisori banali)
- I divisori non banali di  $a$  sono detti *fattori* di  $a$

**Interi primi e composti:**

Un intero  $a > 1$  che ha solo divisori banali è detto **primo**. I numeri che non sono primi sono detti **composti**. L'intero 1 non è né primo né composto, analogamente 0 e i numeri negativi non sono né primi né composti.

**Teorema 1:** Per ogni intero  $a$  e ogni intero positivo  $n$ , sono unici gli interi  $q$  e  $r$  tali che  $0 \leq r < n$  e  $a = qn + r$

$$\forall a, n \in \mathbb{Z} : n > 0 \exists !q, r : 0 \leq r < n \text{ e } a = qn + r$$

- $q = \lfloor a/n \rfloor$  è il **quoziente** della divisione
- $r = a \bmod n$  è il **resto** della divisione
- $n | a$  se e solo se il resto è zero ( $r = 0$ )

**Classi di equivalenza:**

Gli interi possono essere divisi in  $n$  classi di equivalenza  $[a]_n$  in base al resto modulo  $n$ .

$$[a]_n = \{a + kn : k \in \mathbb{Z}\}$$

- es  $[3]_7 = \{\dots, -11, -4, 3, 10, 17, \dots\}$  sono tutti numeri che hanno 3 come resto se dividiamo il valore per 7.

La notazione  $a \equiv b \pmod{n}$  si legge “ $a$  equivalente a  $b$  in resto modulo  $n$ ” è analoga a  $a \in [b]_n$  ed indica che il resto **mod n** di  $a$  e  $b$  sono uguali.

$$\text{-es } 10 \equiv 17 \pmod{7} \rightarrow [17]_7 \equiv [3]_7 = \{\dots, -11, -4, \dots, 10, \dots\}$$

L'insieme di tutte le **n** classi di equivalenza modulo **n** è indicato con:

$$Z_n = \{[a]_n : 0 \leq a < n\}$$

### Massimo comune divisore

Se **d** | **a** e **d** | **b**, allora **d** è un divisore comune di **a** e **b**. Il massimo comune divisore (gcd) di due interi **a** e **b** è il più grande tra i divisori comuni di **a** e **b**.

Casi particolari:

- $\text{gcd}(a, 0) = a$
- $\text{gcd}(0, 0) = 0$

### Interi relativamente primi

Due interi **a** e **b** sono detti relativamente primi se il loro unico comune divisore è 1, quindi  $\text{gcd}(a, b) = 1$

- es. 8 e 15 sono relativamente primi

Gli interi  $n_1, n_2, \dots, n_k$  sono relativamente primi a coppie se:  $\text{gcd}(n_i, n_j) = 1$  per  $i \neq j$ .

**Teorema 6:** Siano **a**, **b** e **p** tre interi tali che  $\text{gcd}(a, p) = 1$  e  $\text{gcd}(b, p) = 1$  allora  $\text{gcd}(ab, p) = 1$

$a, b, p \in Z : \text{gcd}(a, p) = 1 \text{ e } \text{gcd}(b, p) = 1 \text{ quindi } \rightarrow \text{gcd}(ab, p) = 1$

I due seguenti teoremi sfruttano il concetto di **Fattorizzazione**

**Teorema 7:** Sia **p** un numero intero. Se **p** | **ab**, allora **p** | **a** oppure **p** | **b** (o entrambe)

$a, b, p \in Z$  **p** primo :  $p \mid ab \rightarrow p \mid a \text{ e/o } p \mid b$

**Teorema 8:** Un intero composto **a** può essere scritto in un solo modo come prodotto della forma:

$a = p_1^{e_1} * p_2^{e_2} * \dots * p_r^{e_r}$  dove **p<sub>i</sub>** sono numeri primi ed **e<sub>i</sub>** interi positivi.

Questa vuol dire che dato un numero, esiste un'unica forma che permette di esprimere attraverso una fattorizzazione. Il problema della fattorizzazione è un problema **n-pi completo**, **NON può essere risolto in un tempo polinomiale rispetto al numero di bit sul quale è rappresentato un numero**.

### Massimo comune divisore tramite fattorizzazione

Il massimo comune divisore di **a** e **b** interi positivi si può calcolare tramite la loro fattorizzazione:

$a, b > 0 : a = p_1^{e_1} * p_2^{e_2} * \dots * p_r^{e_r}$  e  $b = p_1^{f_1} * p_2^{f_2} * \dots * p_r^{f_r} \rightarrow \text{gcd}(a, b) = p_1^{\min(e_1, f_1)} * \dots * p_r^{\min(e_r, f_r)}$ . Esponenti nulli per numeri primi non presenti nella fattorizzazione

**Teorema 9 (sul quale si basa l'algoritmo di Euclide):** Per ogni intero non negativo **a** e ogni intero positivo **b**,  $\text{gcd}(a, b) = \text{gcd}(b, a \bmod b)$

$a, b \in Z \quad \forall a \geq 0, b > 0 \quad \text{gcd}(a, b) = \text{gcd}(b, a \bmod b)$

```
Euclid (a, b)
if b = 0
    then return a
else return Euclid (b, a mod b)
```

Utile è la sua versione estesa, che restituisce anche due interi  $x$  e  $y$  tali che  $d = \gcd(a, b) = ax + by$ .

```
Extended-Euclid (a,b)
if b = 0
    then return (a,1,0)
(d',x',y') ← Extended-Euclid (b, a mod b)
(d,x,y) ← (d',y',x' - [a/b]y')
return (d,x,y)
```

La complessità è pari al logaritmo del termine maggiore, quindi polinomiale al numero di bit.

## Aritmetica modulare

Sia  $S$  un insieme e  $\oplus$  un'operazione binaria su  $S$ .  $(S, \oplus)$  è un **gruppo** se:

- $\oplus$  è un'operazione interna:

$$\forall a, b \in S \rightarrow a \oplus b \in S$$

- Esiste l'elemento neutro  $e \in S$

$$a \oplus e = e \oplus a = a \in S$$

- Vale la proprietà associativa:

$$a \oplus (b \oplus c) = (a \oplus b) \oplus c \quad \forall a, b, c \in S$$

- Esiste l'elemento inverso  $b \in S$

$$\forall a, b \in S, \exists b \in S : a \oplus b = b \oplus a = a$$

Se vale ancora la proprietà commutativa, allora il gruppo è detto **abeliano**.

$$\forall a, b \in S, a \oplus b = b \oplus a$$

Il gruppo è finito se la cardinalità di  $S$  è finita.

Per definire un gruppo su  $Z_n$  occorre definire operazioni binarie su  $Z_n$

$$\begin{aligned} a \equiv a' \pmod{n} &\rightarrow a + b \equiv a' + b' \pmod{n} \\ b \equiv b' \pmod{n} &\rightarrow ab \equiv a'b' \pmod{n} \end{aligned}$$

Definiamo:

- Addizione modulo  $n$ :  $[a]_n +_n [b]_n = [a + b]_n$
- Moltiplicazione modulo  $n$ :  $[a]_n *_n [b]_n = [ab]_n$

$$\text{-es } [2]_7 +_7 [4]_7 = [6]_7$$

Si dimostra che  $(Z_n, +_n)$  e  $(Z_n^*, *_n)$  sono gruppi abeliani finiti dove  $Z_n^*$  è l'insieme degli elementi di  $Z_n$  relativamente primi a  $n$ :

$$Z_n^* = \{[a]_n \in Z_n : \gcd(a, n) = 1\}$$

La cardinalità di  $Z_n$  è  $n$ . La cardinalità di  $Z_n^*$  è pari a : (funzione di Eulero)  $\phi(n) = n \prod_{p|n} (1 - 1/p)$

Dove  $p$  rappresenta un numero primo. Essendo  $p$  primo:  $Z_n^* = \{1, 2, \dots, p-1\}$  con  $\phi(p) = p-1$

L'inverso di  $[a]_n$  rispetto alla moltiplicazione  $(Z_n^*, \cdot)_n$  è:  $[x]_n : [a]_n \cdot_n [x]_n = [1]_n$  cioè  
 $ax \equiv 1 \pmod{n}$

Per trovare l'elemento inverso utilizzeremo dei corollari:

**Corollario 21:** L'equazione  $ax \equiv b \pmod{n}$  ammette soluzioni se e solo se  $\gcd(a, n) | b$

**Corollario 22:** L'equazione  $ax \equiv b \pmod{n}$  o ammette  $d = \gcd(a, n)$  distinte soluzioni modulo  $n$  o non ammette soluzione.

L'elemento inverso della moltiplicazione è tale che  $b = 1$ , quindi i due corollari si modificano e diventano:

**Corollario 26:** Per ogni  $n > 1$ , se  $\gcd(a, n) = 1$ , allora l'equazione  $ax \equiv 1 \pmod{n}$  ammette un'unica soluzione modulo  $n$ . In particolare, il valore di  $x$  è la  $x$  restituita dalla Extended-Euclid, poiché  $\gcd(a, n) = 1 = ax + ny$  implica che  $ax \equiv 1 \pmod{n}$ .

**Teorema cinese del resto (corollario 29):** Se  $n_1, n_2, \dots, n_k$  sono relativamente primi a coppie e  $n = n_1 * n_2 * \dots * n_k$  allora per tutti gli interi  $x$  e  $ax \equiv a \pmod{n_i}$  per  $i=1, 2, \dots, k$  se e solo se  $x \equiv a \pmod{n}$

$$\forall x, a \in Z \quad x \equiv a \pmod{n} \Leftrightarrow x \equiv a \pmod{n_i} \quad \text{per } i=1, 2, \dots, k$$

-es:

$$n_1 = 2, \quad n_2 = 15, \quad n_3 = 7 \rightarrow n = 210$$

$$10 \equiv 220 \pmod{210}$$

$$10 \equiv 220 \pmod{2} \text{ infatti } 10 \text{ e } 220 \text{ appartengono a } [0]_2$$

$$10 \equiv 220 \pmod{15} \text{ infatti } 10 \text{ e } 220 \text{ appartengono a } [10]_{15}$$

$$10 \equiv 220 \pmod{7} \text{ infatti } 10 \text{ e } 220 \text{ appartengono a } [3]_7$$

**Teorema di Fermat:** Si consideri la potenza modulo  $n$ , definita per estensione dalla moltiplicazione modulo  $n$  su  $Z_n^*$ .

$$p \text{ primo, } a \in Z_p^* \rightarrow a^{p-1} \equiv 1 \pmod{p}$$

### Crittografia a chiave pubblica

Un sistema di crittografia a chiave pubblica è usato per cifrare i messaggi scambiati tra 2 entità al fine di impedire la loro decodifica da terze parti. Consideriamo due entità **A** e **B** che si scambiano dei messaggi, ogni entità ha una chiave pubblica **P** e una chiave segreta **S**.

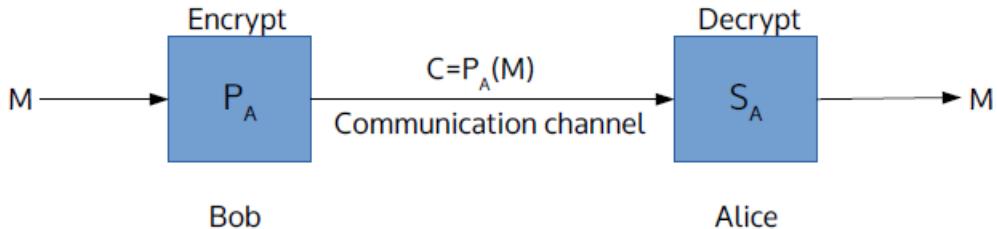
Per segretezza intendiamo che un determinato messaggio sia decifrabile solo dal ricevente. L'integrità e la correttezza sono invece proprietà che permettono al ricevente di verificare che il messaggio sia integra, senza alterazioni, e che sia stato effettivamente mandato dal mittente.

I messaggi sono codificati con una funzione pubblica **P<sub>A</sub>()** che ha come parametro la chiave **P<sub>A</sub>**, decodificati con una funzione segreta **S<sub>A</sub>()** che ha come parametro **S<sub>A</sub>**. Le due funzioni sono l'una l'inversa dell'altra ,

quindi  $P(S(M)) = S(P(M)) = M$ . Se B vuole mandare un messaggio ad A, lo codifica con la chiave pubblica di A, così A può decodificarlo con la sua chiave segreta.

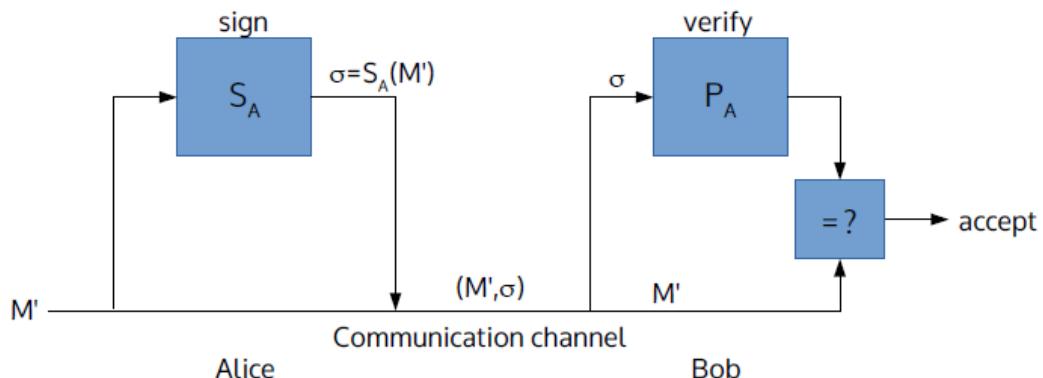
Dove sta la difficoltà di questa operazione? La difficoltà sta nel risalire nella chiave segreta dalla chiave pubblica.

La **segretezza** del messaggio è garantita se  $P_A()$  è difficilmente invertibile, quindi solo A deve riuscire a calcolare  $S_A()$  in un tempo ragionevole. Quindi Bob invia il messaggio cifrandolo con la chiave pubblica di Alice, in questo modo garantiamo che per risalire al messaggio M è necessario applicare a quello inviato sul canale la funzione chiave segreta corrispondente.



Operando in questo modo, garantiamo anche l'autenticità? No, perché se qualcuno attaccasse il messaggio o anche il canale stesso;

Per garantire l'**autenticità** del mittente e l'**integrità** del messaggio, il mittente invia sia il messaggio in chiaro che il messaggio cifrato con la propria chiave segreta (il messaggio cifrato è detto firma digitale  $\sigma$ ). In questo modo, il destinatario confronta il messaggio in chiaro con quello ottenuto usando la chiave pubblica del mittente sulla firma digitale.



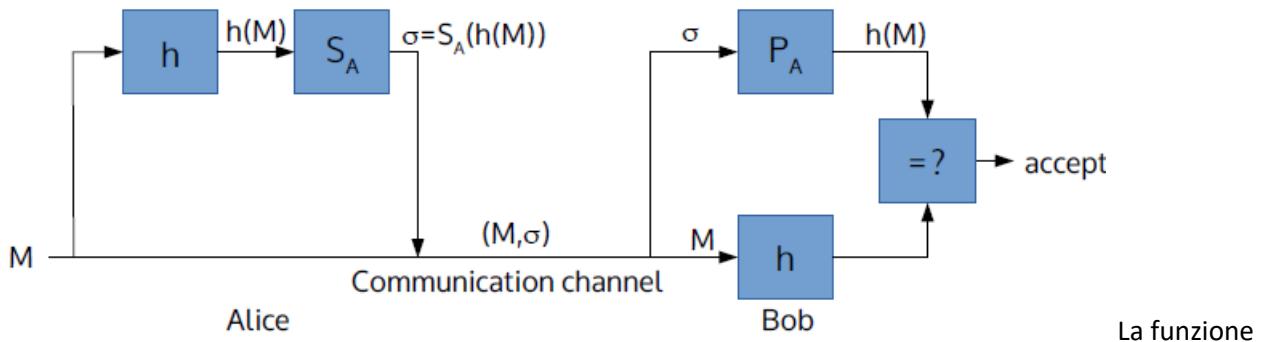
Combinando le due cose, si garantiscono entrambe. Quindi:

- Il mittente A
    - Calcola la firma digitale (con la propria chiave segreta  $S_A$ ) e la appende al messaggio
    - Calcola il messaggio e lo firma ( con la chiave pubblica  $P_B$  del destinatario.
  - Il ricevente B
    - Decifra firma e messaggio del mittente ( con la propria chiave privata  $S_B$ )
    - Verifica la firma del mittente (con la chiave pubblica  $P_A$  del mittente)

In realtà, non è agevole inviare una firma digitale della stessa dimensione del messaggio poiché raddoppiamo il numero di bit da trasmettere. Un'alternativa consiste nell'inviare una firma digitale basata, non sull'intero messaggio, sul risultato di una funzione **hash** che sia:

- Facile da calcolare

- Difficile che generi lo stesso risultato per due messaggi diversi
- Produce una stringa di bit di dimensione fissata (e minore di quella del messaggio)



hash è usata anche dal ricevente per verificare la firma digitale. In realtà si va a perdere un po' di sicurezza, perché è garantita l'autenticità di  $h$ , non di  $M$ . Tuttavia la scelta di comporta poi difficile tornare al messaggio. Purtroppo questo tipo di funzione non garantisce la **sicurezza**, ma garantisce la **proprietà di integrità e l'autenticità**.

## Crittografia RSA

Il sistema di crittografia a chiave pubblica **RSA** (Rivest,Shamir,Adleman) si basa sulla differenza di complessità tra:

- Trovare grandi numeri primi (semplice)
- Fattorizzare il prodotto di due primi grandi (complesso)

La coppia di chiavi si calcola nel seguente modo:

- Si scelgono due numeri primi distinti molto grandi  $p$  e  $q$ .
- Si calcola il prodotto  $n = pq$ .
- Si sceglie un piccolo intero dispari  $e$  relativamente primo con la funzione di Eulero  
 $\phi(n) = (p - 1)(q - 1)$ 
  - $e$  deve essere dispari perché  $\phi(n)$  è pari, se  $e$  fosse pari non potrebbe essere relativamente primo con  $\phi(n)$
- Si calcola l'inverso moltiplicativo  $d$  di  $e$  modulo  $\phi(n)$ , questo ci fa capire perché il vincolo precedente, usando la Extended-Euclid
- La **chiave pubblica** è la coppia  $P = (e, n)$
- La **chiave privata** è la coppia  $S = (d, n)$
- La funzione associata alla chiave pubblica è  $P(M) = M^e \pmod{n}$
- La funzione associata alla chiave privata è  $S(M) = M^d \pmod{n}$

L'insieme dei messaggi appartiene a  $Z_n$ .

Ricavare la chiave privata è complicato, perché trovare  $d$  è computazionalmente molto complesso. Queste funzioni possono essere calcolate con la procedura Modular-Exponentiation. Inoltre poiché  $e$  è più piccolo di  $d$ , applicare la chiave pubblica richiede meno tempo rispetto a quella privata. In ogni caso però la

complessità è polinomiale al numero di bit (stiamo supponendo che il numero di bit di  $e$  è  $O(1)$ , mentre quello di  $n$  e  $d$  è  $O(\beta)$ ).

Vogliamo ora dimostrare che queste funzioni sono l'una l'inversa dell'altra quindi  $P(S(M)) = S(P(M)) = M$ .

### Dimostrazione.

$$e^*d \equiv 1 \pmod{\phi(n)} \rightarrow e^*d = 1 + k(p-1)(q-1)$$

- Se  $M \neq 0 \pmod{p}$

$$\rightarrow M^{ed} \equiv M^*M^{k(p-1)(q-1)} \pmod{p} \equiv M(M^{(p-1)})^{k(q-1)} \pmod{p} \equiv \text{Teorema di Fermat}$$

$$M(1)^{k(q-1)} \pmod{p} \equiv M \pmod{p}$$

- Se  $M = 0 \pmod{p}$

$$\rightarrow M^{ed} \equiv M^* \pmod{q} \equiv \text{Teorema cinese del resto}$$

$$M^{ed} \equiv M \pmod{n}$$

Quindi le funzioni sono una l'inversa dell'altra perché alla fine si ottiene il messaggio originale.

## Programmazione dinamica e algoritmi Greedy

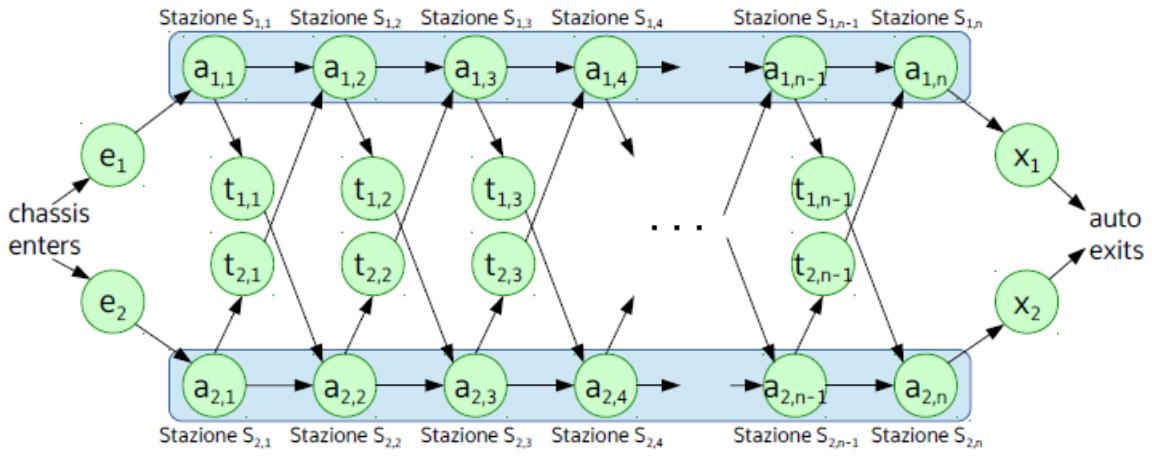
Gli algoritmi Greed e la programmazione dinamica vengono applicati ai problemi di **ottimizzazione**, ovvero a problemi che ammettono diverse soluzioni, ad ogni soluzione è associato un costo e il nostro obiettivo è quello di trovare la soluzione con costo ottimo. Per ottimo si intende in base a quale parametro si valuta.

Come operano questi metodi? Prima di tutto il problema da trattare deve avere una **sottostruttura ottima**, ovvero quando dobbiamo ricavare una soluzione ottima a più sottoproblemi. Partendo da un problema troviamo sempre dei sottoproblemi.

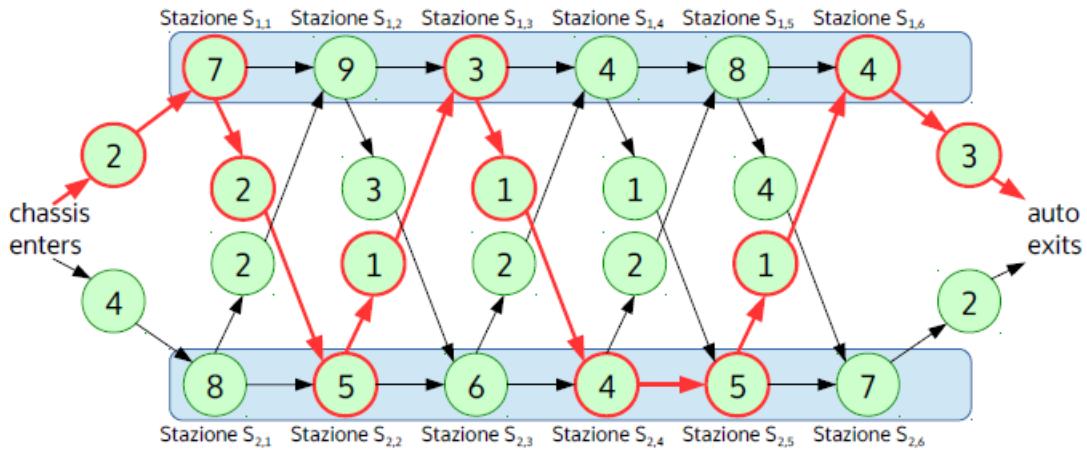
Un algoritmo di programmazione dinamica può essere sviluppato quindi seguendo quattro passi

- Caratterizzazione della struttura di una soluzione ottima
- Definizione ricorsiva del valore di una soluzione ottima
- Calcolo del valore di una soluzione ottima in modo bottom-up
- Costruzione di una soluzione ottima dalle informazioni calcolate
- L'ultimo passo non è necessario se è richiesto solo il valore ottimo e non una soluzione ottima

Un esempio è quello della catena di montaggio.



Valutiamo come completare uno chassis di un automobile, il problema presenta varie soluzioni e costi, noi vogliamo trovare la subottima con costo e tempo minori.



Arrivare nel tempo minimo nella terza stazione della seconda linea è un sottoproblema, ma anche arrivare all'uscita della quarta linea, etc. Quindi valutiamo il percorso più veloce come quello evidenziato. Dato il persorso, il calcolarne il tempo richiede  $\theta(n)$  ma esistono  $2^n$  possibili percorsi; quindi l'appuccio con forza bruta non è minimamente fattibile.

In ogni caso, la soluzione ottima di un problema contiene al suo interno la soluzione ottima di un sottoproblema. Si perverrà dunque ad una soluzione ricorsiva, è conveniente utilizzare la programmazione dinamica quando un algoritmo ricorsivo invocherebbe più volte uno stesso sottoproblema. Se ciò non accade, l'appuccio divide-et-impera è adeguato. La programmazione dinamica risolve ogni sottoproblema una sola volta e utilizza il valore calcolato quando si ripresenta lo stesso sottoproblema nell'appuccio bottom-up.

Per alcuni problemi di ottimizzazione, un appuccio greedy risulta più efficiente della programmazione dinamica. Gli algoritmi greedy procedono top-down scegliendo di volta in volta il sottoproblema che garantisce l'ottimo "locale". L'algoritmo di Dijkstra per il calcolo del percorso minimo utilizza un appuccio greedy