

Appunti Algoritmi e Strutture Dati

Roberto Ingenito

20 settembre 2022

Indice

1	Introduzione	4
1.1	Algoritmo	4
1.2	Struttura dati	4
1.3	Modello computazionale	5
1.4	Modello RAM Random Access Machine	5
1.5	Esercizio: Conteggio delle coppie	6
2	Notazione asintotica	8
2.1	Proprietà delle notazioni	10
2.1.1	Riflessiva	10
2.1.2	Simmetrica	10
2.1.3	Transitiva	10
2.1.4	Altre proprietà	10
3	Divide Et Impera	11
3.1	Equazione di ricorrenza	11
4	Algoritmi di ordinamento	11
4.1	Insertion Sort	12
4.2	Merge Sort	14
4.2.1	Analisi	15
4.3	Selection Sort	16
4.3.1	Analisi	16
4.4	Heap Sort	17
4.4.1	Albero	17
4.4.2	Definizione di ALTEZZA di un albero binario	17
4.4.3	Proprietà di Heap	18
4.4.4	Algoritmo	19
4.4.5	Analisi	20
4.5	Quick Sort	21
4.5.1	Analisi	22
4.6	Limite degli ordinamenti per confronto	25
4.6.1	Albero di decisione	25
5	Strutture dati	26
5.1	Operazioni	26
5.2	Alberi binari	27
5.3	Casi d'uso degli scorrimenti di un albero	28
5.4	Alberi binari di ricerca (Binary Search Tree)	29
5.4.1	Ricerca	29
5.4.2	Inserimento	30
5.4.3	Cancellazione	31
5.4.4	Ricerca del successore	34
5.5	Albero perfettamente bilanciato	35
5.6	Alberi AVL	36
5.6.1	Alberi AVL minimi	36
5.6.2	Inserimento in un AVL	38
5.6.3	Cancellazione in un AVL	40
5.6.4	Algoritmi	42
5.7	Alberi Red Black	44
5.7.1	Altezza nera	44
5.7.2	Teorema	45
5.7.3	Inserimento in red-black	46
5.7.4	Cancellazione in red-black	47
5.7.5	Algoritmi di inserimento e cancellazione	49
6	Da ricorsivo a iterativo	51
6.1	Traduzione di un algoritmo	52

7	Grafi	55
7.1	Rappresentazione dei grafi	57
7.2	Esplorazione di un grafo	57
7.2.1	Visita in ampiezza	58
7.2.2	Calcolo delle distanze	60
7.2.3	Percorso minimo	61
7.2.4	Visita in profondità	62
7.3	Teorema della struttura a parentesi (DFS)	64
7.4	Teorema del percorso bianco	65
7.5	Aciclicità	65
7.6	Ordinamento topologico	66
7.6.1	Teorema	66
7.6.2	Algoritmo (ampiezza)	67
7.6.3	Algoritmo (profondità)	68
7.7	Calcolo delle componenti fortemente connesse	69

1 Introduzione

1.1 Algoritmo

Un algoritmo è una sequenza di istruzioni

- elementari istruzione **atomica, singola**
- finita
- non ambigua
- esplicita niente dev'essere sottinteso

che trasforma un valore in **input** in un valore in **output**, con lo scopo di risolvere uno specifico problema.

Un algoritmo dev'essere:

- **Efficiente**
sia in termini di tempo che di spazio
- **Generale**
Ossia deve risolvere il problema con qualsiasi input
Ad esempio, se l'algoritmo prende in input un array, deve poter prendere in input un array di qualsiasi dimensione
(e non per forza grande, ad esempio, 3)
- **Corretto**
Ossia, l'output è corretto, è quello per cui l'algoritmo è stato pensato.
 $\mathcal{R} \subseteq input \times output$
 \mathcal{R} è una relazione che associa un input ad un output

1.2 Struttura dati

Una struttura dati è un'entità che serve a:

- organizzare
- memorizzare
- accedere
- modificare

dei dati.

Distinguiamo diversi tipi di strutture dati in base al:

- Tipo dei dati gestiti
- Operazione sui dati
- Proprietà assunte sui dati (dati ordinati, hash, ...)
- Proprietà attese sui dati
Proprietà che ci aspettiamo che i dati abbiano
- Efficienza temporale delle operazioni
- Efficienza spaziale della struttura
- Persistenza vs Mutabilità
- Livello di astrazione dell'interfaccia
Interfaccia: insieme di operazioni concesse sui dati di una struttura dati

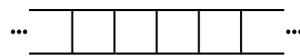
1.3 Modello computazionale

Macchina di Turing RAM: Random Access Machine

$M : \langle Q, q_i, q_f, \Sigma, \sqcup, \delta \rangle$

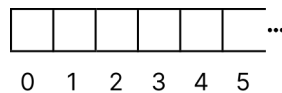
- M : Macchina
- Q : Insieme finito di stati
- $q_i \in Q$: Stato iniziale
- $q_f \in Q$: Stato finale
- Σ : Insieme dei simboli che la macchina elabora
ad esempio dei numeri $0, \dots, 9$
- $\sqcup \in \Sigma$: Blank, spazio vuoto. Dopo o prima di esso non c'è nulla.
- δ : Funzione di transizione. Descrive come la macchina evolve da stato a stato
 $\delta : Q \times \Sigma \longrightarrow Q \times \Sigma \times \{-1, 0, +1\}$ -1 : Left 0 : Null $+1$: Right

Possiamo rappresentare la macchina di Turing come un nastro infinito



1.4 Modello RAM Random Access Machine

Possiamo rappresentare una RAM come un nastro infinito, per semplicità lo poniamo finito a sinistra e infinito a destra.



L'importante differenza rispetto alla macchina di Turing, è che ogni cella è identificata da un indice **univoco**. Possiamo quindi accedere a qualsiasi cella in tempo costante

Costrutti utilizzati in una RAM:

1. Assegnamento / Lettura / Scrittura
2. Sequenze (blocchi di codice o interi programmi)
Esempio: PRG1, PRG2, ...
3. Selezione (costrutto condizionale)
IF *cond* PRG1 ELSE PRG2
4. Iterazione
WHILE *cond* DO PRG1

Tempo di esecuzione:

1. $T(\text{Assegnamento}) : O(1)$ tempo costante
2. $T(\text{PRG}_1, \text{PRG}_2) = T(\text{PRG}_1) + T(\text{PRG}_2)$
3. $T(\text{IF}) = T(\text{cond}) + \max\{T(\text{PRG}_1), T(\text{PRG}_2)\}$
4. Dipende dalla condizione, non è possibile stabilirlo a priori

1.5 Esercizio: Conteggio delle coppie

Algoritmo 1

Input: $n \in \mathbb{N}$

output: $\{(i, j) \in \mathbb{N} \times \mathbb{N} \mid i \leq j \leq n\}$

```

1. counter ← 0           t1
2. for i=0 to n do       t2
3.   for j=0 to n do     t3
4.     if i ≤ j then     t4
5.       counter++       t5
6. return counter        t6

```

Tempo di esecuzione:

t_1 : dell'assegnazione

t_2 : di un ciclo for (non del suo corpo, ma di tutto il resto. Controllo della condizione, assegnamento, incremento, ecc.)

t_3 : stessa cosa

t_4 : dell'*if* (anche qui, non del corpo ma dell'istruzione *if* in se)

t_5 : di un incremento

t_6 : del *return*

Il secondo *for* viene eseguito $n + 1$ volte, quindi (incluso il corpo) impiega $(n + 1)[t_3 + t_4 + t_5]$

Stesso discorso per il primo *for*, quindi impiega $(n + 1)[t_2 + (n + 1)[t_3 + t_4 + t_5]]$

L'intero algoritmo impiega $t_1 + (n + 1)[t_2 + (n + 1)[t_3 + t_4 + t_5]] + t_6$

$$= t_1 + (n + 1)t_2 + (n + 1)^2[t_3 + t_4 + t_5] + t_6 \quad (n + 1)^2 = n^2 + 2n + 1$$

$$= t_1 + n t_2 + t_2 + (n + 1)^2[t_3 + t_4 + t_5] + t_6 =$$

$$= t_1 + n t_2 + t_2 + n^2(t_3 + t_4 + t_5) + n[2(t_3 + t_4 + t_5)] + (t_3 + t_4 + t_5) + t_6 =$$

$$= \underbrace{n^2(t_3 + t_4 + t_5)}_{c_1} + \underbrace{n[2(t_3 + t_4 + t_5)]}_{c_2} + \underbrace{(t_1 + t_2 + t_3 + t_4 + t_5 + t_6)}_{c_3} =$$

$$= n^2 c_1 + n c_2 + c_3$$

l'algoritmo impiega dunque un tempo quadratico.

Algoritmo 2

```

1. counter ← 0           t1
2. for i=0 to n do       t2
3.   for j=i to n do     t3
4.     counter++         t5
5. return counter        t6

```

$$t_1 + \sum_{i=0}^n [t_2 + (n - i + 1)(t_3 + t_5)] + t_6 =$$

$$= t_1 + \sum_{i=0}^n [t_2 + (n + 1)(t_3 + t_5) - i(t_3 + t_5)] + t_6 =$$

$$= t_1 + \sum_{i=0}^n [t_2 + (n + 1)(t_3 + t_5)] - \sum_{i=0}^n [i(t_3 + t_5)] + t_6 =$$

$$= t_1 + t_6 + (n + 1)[t_2 + (n + 1)(t_3 + t_5)] - (t_3 + t_5) \sum_{i=0}^n (i) = \quad \sum_{i=0}^n (i) \text{ somma dei primi } n \text{ numeri naturali } \frac{n(n + 1)}{2}$$

$$= t_1 + t_6 + (n + 1)t_2 + (n + 1)^2(t_3 + t_5) - (t_3 + t_5) \frac{n(n + 1)}{2} =$$

$$= (t_3 + t_5)(n + 1) \left[(n + 1) - \frac{n}{2} \right] + n t_2 + (t_1 + t_2 + t_6) \quad \left[(n + 1) - \frac{n}{2} \right] = \frac{n}{2} + 1$$

$$\begin{aligned}
&= (t_3 + t_5) \left[\frac{n^2}{2} + n + \frac{n}{2} + 1 \right] + n t_2 + (t_1 + t_2 + t_6) = \\
&= n^2 \underbrace{\frac{t_3 + t_5}{2}}_{c_1} + n \underbrace{\left[\frac{3}{2}(t_3 + t_5) + t_2 \right]}_{c_2} + \underbrace{(t_1 + t_2 + t_3 + t_5 + t_6)}_{c_3} = \\
&= n^2 c_1 + n c_2 + c_3
\end{aligned}$$

Questo secondo algoritmo è migliore solo per le costanti moltiplicative, ma ha comunque un andamento quadratico.

Algoritmo 3

```

1. counter ← 0           t1
2. for i=0 to n do       t2
4.   counter ← counter + (n-i+1) t7
5. return counter       t6

```

$$\begin{aligned}
&(n+1)(t_2 + t_7) + t_1 + t_6 = \\
&= n(t_2 + t_7) + (t_1 + t_2 + t_6 + t_7) = \\
&= n c_1 + c_2
\end{aligned}$$

Si passa da un andamento quadratico ad un andamento lineare

Algoritmo 4

```

1. return  $\frac{n^2 + 3n + 2}{2}$    t1

```

Infine, è possibile avere il risultato in tempo costante

dall'algoritmo precedente

```

2. for i=0 to n do
4.   counter ← counter + (n-i+1)

```

Questo non è altro che $\sum_{i=0}^n (n - i + 1)$

$$\begin{aligned}
&= \sum_{i=0}^n (n+1) - \sum_{i=0}^n (i) = (n+1) \sum_{i=0}^n (1) - \sum_{i=0}^n (i) = \sum_{i=0}^n (1) = n+1 \quad \sum_{i=0}^n (i) = \frac{n(n+1)}{2} \\
&= (n+1)^2 - \frac{n(n+1)}{2} = \frac{2(n+1)^2 - n(n+1)}{2} = \frac{2n^2 + 4n + 2 - n^2 + n}{2} = \frac{n^2 + 3n + 2}{2}
\end{aligned}$$

2 Notazione asintotica

Nel calcolo del tempo di esecuzione di un algoritmo, quando l'input è sufficientemente grande, le costanti moltiplicative e i termini di ordine inferiore diventano trascurabili.

Ci interessa quindi sapere come il tempo di esecuzione aumenta all'aumentare dell'input, **al limite**.

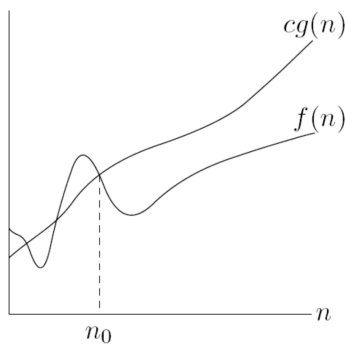
Infatti, un algoritmo asintoticamente più efficiente sarà il migliore con tutti gli input tranne con quelli molto piccoli.

Le notazioni utilizzate per descrivere il tempo di esecuzione asintotico di un algoritmo, sono funzioni:

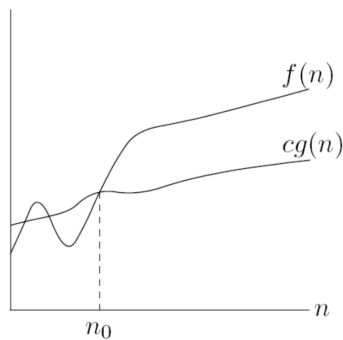
$$f : \mathbb{N} \longrightarrow \mathbb{R}$$

In particolare f è

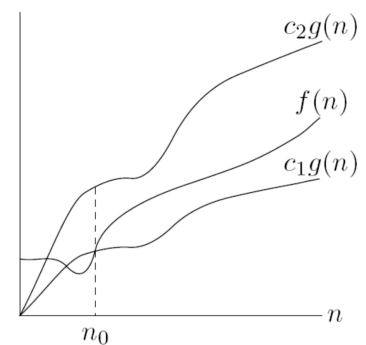
- **positiva**
 $\exists n_0 \in \mathbb{N} : \forall n \geq n_0 \quad f(n) \geq 0$
- **crescente**
 $\exists n_0 \in \mathbb{N} : \forall n \geq n_0 \quad f(n) \leq f(n+1)$



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



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



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

Notazione O grande

Indichiamo con $O(g(n))$ l'insieme delle funzioni:

$$O(g(n)) = \left\{ f(n) : \begin{array}{l} \exists c \in \mathbb{R}^+ \\ \exists n_0 \in \mathbb{N} \end{array} \text{ tale che } \forall n \geq n_0, \quad f(n) \leq c g(n) \right\}$$

$O(g(n))$ limita superiormente $f(n)$

Notazione Ω grande

Indichiamo con $\Omega(g(n))$ l'insieme delle funzioni:

$$\Omega(g(n)) = \left\{ f(n) : \begin{array}{l} \exists c \in \mathbb{R}^+ \\ \exists n_0 \in \mathbb{N} \end{array} \text{ tale che } \forall n \geq n_0, \quad f(n) \geq c g(n) \right\}$$

$\Omega(g(n))$ limita inferiormente $f(n)$

Notazione Θ

Indichiamo con $\Theta(g(n))$ l'insieme delle funzioni:

$$\Theta(g(n)) = \left\{ f(n) : \begin{array}{l} \exists c_1, c_2 \in \mathbb{R}^+ \\ \exists n_0 \in \mathbb{N} \end{array} \text{ tale che } \forall n \geq n_0, \quad c_1 g(n) \leq f(n) \leq c_2 g(n) \right\}$$

Infatti il $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = L$

$$\forall \varepsilon \in \mathbb{R}^+ : \quad \forall n \geq n_0, \quad L - \varepsilon \leq \frac{f(n)}{g(n)} \leq L + \varepsilon$$

$$(L - \varepsilon) g(n) \leq f(n) \leq (L + \varepsilon) g(n) \quad (L - \varepsilon) = c_1 \quad (L + \varepsilon) = c_2$$

Notazione o piccolo

Indichiamo con $o(g(n))$ l'insieme delle funzioni:

$$o(g(n)) = \left\{ f(n) : \begin{array}{l} \forall c \in \mathbb{R}^+ \\ \exists n_0 \in \mathbb{N} \end{array} \text{ tale che } \forall n \geq n_0, \quad f(n) \leq c g(n) \right\}$$

Nella O grande è sufficiente che ci sia **ALMENO** un $c > 0$ tale che $f(n) \leq c g(n) \quad \forall n > n_0$.

Nella o piccola, invece, $f(n) \leq c g(n)$ **PER OGNI** $c > 0$

Quindi se $f(n)$ è o piccolo di $g(n)$, è anche O grande di $g(n)$

$o(g(n))$ limita superiormente $f(n)$

Infatti il $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = 0$ perché, al limite, $g(n)$ cresce più velocemente

Notazione ω piccolo

Indichiamo con $\omega(g(n))$ l'insieme delle funzioni:

$$\omega(g(n)) = \left\{ f(n) : \begin{array}{l} \forall c \in \mathbb{R}^+ \\ \exists n_0 \in \mathbb{N} \end{array} \text{ tale che } \forall n \geq n_0, \quad f(n) \geq c g(n) \right\}$$

Nella Ω grande è sufficiente che ci sia **ALMENO** un $c > 0$ tale che $f(n) \geq c g(n) \quad \forall n > n_0$.

Nella ω piccola, invece, $f(n) \geq c g(n)$ **PER OGNI** $c > 0$

Quindi se $f(n)$ è ω piccolo di $g(n)$, è anche Ω grande di $g(n)$

$\omega(g(n))$ limita inferiormente $f(n)$

Infatti il $\lim_{n \rightarrow \infty} \frac{f(n)}{g(n)} = \infty$ perché, al limite, $g(n)$ cresce più lentamente

2.1 Proprietà delle notazioni

2.1.1 Riflessiva

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

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

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

2.1.2 Simmetrica

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

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

$$\begin{array}{l} \exists c_1, c_2 \in \mathbb{R}^+ \\ \exists n_0 \in \mathbb{N} \end{array} : \forall n \geq n_0, \quad c_1 g(n) \leq f(n) \leq c_2 g(n)$$

$$c_1 g(n) \leq f(n) \Rightarrow g(n) \leq \frac{1}{c_1} f(n)$$

$$f(n) \leq c_2 g(n) \Rightarrow \frac{1}{c_2} f(n) \leq g(n)$$

$$\text{Scelgo } c'_1 = \frac{1}{c_2} \text{ e } c'_2 = \frac{1}{c_1}$$

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

$$\begin{array}{l} \exists c'_1, c'_2 \in \mathbb{R}^+ \\ \exists n_0 \in \mathbb{N} \end{array} : \forall n \geq n_0, \quad c'_1 f(n) \leq g(n) \leq c'_2 f(n)$$

2.1.3 Transitiva

$$f(n) = \Theta(g(n)) \wedge g(n) = \Theta(h(n)) \Rightarrow f(n) = \Theta(h(n))$$

Questa proprietà si applica a tutte le notazioni. In questo caso vediamo solo il Θ

$$\begin{array}{l} \exists c_1, c_2 \in \mathbb{R}^+ \\ \exists n_0 \in \mathbb{N} \end{array} : \forall n \geq n_0, \quad c_1 g(n) \leq f(n) \leq c_2 g(n) \quad f(n) = \Theta(g(n))$$

$$\begin{array}{l} \exists c'_1, c'_2 \in \mathbb{R}^+ \\ \exists n_0 \in \mathbb{N} \end{array} : \forall n \geq n_0, \quad c'_1 h(n) \leq g(n) \leq c'_2 h(n) \quad g(n) = \Theta(h(n))$$

$$\text{Moltiplico } c'_1 h(n) \leq g(n) \text{ per } c_1$$

$$c_1 c'_1 h(n) \leq c_1 g(n)$$

$$\text{Moltiplico } g(n) \leq c'_2 h(n) \text{ per } c_2$$

$$c_2 g(n) \leq c_2 c'_2 h(n)$$

$$c_1 c'_1 h(n) \leq c_1 g(n) \leq f(n) \leq c_2 g(n) \leq c_2 c'_2 h(n)$$

$$\text{Scelgo } c''_1 = c_1 c'_1 \text{ e } c''_2 = c_2 c'_2 \quad \text{quindi}$$

$$c''_1 h(n) \leq f(n) \leq c''_2 h(n) \quad f(n) = \Theta(h(n))$$

2.1.4 Altre proprietà

- $O/\Omega/\Theta f(k \cdot n) = O/\Omega/\Theta f(n)$
posso rimuovere la costante moltiplicativa
- $O(f(n) + g(n)) = O(\max\{f(n), g(n)\})$
- $\Omega(f(n) + g(n)) = \Omega(\min\{f(n), g(n)\})$
- $\left. \begin{array}{l} f_1(n) = O/\Omega/\Theta(g_1(n)) \\ f_2(n) = O/\Omega/\Theta(g_2(n)) \end{array} \right\} \Rightarrow f_1(n) \cdot f_2(n) = O/\Omega/\Theta(g_1(n) \cdot g_2(n))$
- $p(n) = \Theta(n^d)$
 \uparrow polinomio di grado d
- $f(n) = \Theta(n^k) \quad g(n) = \Theta(h(n))$
 $f(g(n)) = \Theta((h(n))^k)$

3 Divide Et Impera

Il paradigma *divide et impera* prevede tre passi ad ogni livello di ricorsione:

- **Divide**
Il problema viene diviso in sottoproblemi (figli) che sono istanze più piccole del problema (padre)
- **Impera**
I problemi vengono risolti in maniera ricorsiva.
Quando si è arrivati ad una dimensione sufficientemente piccola, il problema viene risolto direttamente.
- **Combina**
Le soluzioni dei sottoproblemi vengono combinate per risolvere il problema iniziale.
Ovvero, l'ultimo figlio risolve il problema direttamente per poi passare il risultato al padre che a sua volta può risolvere il problema per passare il risultato al padre, e così via.

3.1 Equazione di ricorrenza

Una **ricorrenza** è un'equazione o disequazione che descrive una funzione in termini del suo valore, con input via via più piccoli.

Permette di descrivere i tempi di esecuzione degli algoritmi divide et impera.

Esempio merge sort

$$T(n) = \begin{cases} \Theta(1) & \text{se } n \leq 1 \\ 2 \Theta\left(\frac{n}{2}\right) + \Theta(n) & \text{altrimenti} \end{cases}$$

4 Algoritmi di ordinamento

Un algoritmo di ordinamento, prende in input una sequenza arbitraria di dati e la ordina secondo una certa relazione d'ordine. Si indica con:

(D, ρ) ρ è la relazione secondo la quale saranno ordinati gli elementi di D

ρ è anche definita **relazione d'ordine**

Ad esempio $(\{1, 4, 2, 3, 1\}, \leq)$ produrrà $\{1, 1, 2, 3, 4\}$

Input:

- array $A \in D^*$ sequenza arbitraria di dati
- $n = |A| \in \mathbb{N}$
lunghezza di A

Output:

- $A' \in D^*$ $|A'| = n$ vettore dello stesso tipo e lunghezza
- $\exists f : [0, n) \rightarrow [0, n)$ biettiva: $\forall i \in [0, n), A'[i] = A[f(i)]$
Ossia, A' contiene gli stessi elementi di A
- $\forall i \in [0, n - 2], A'[i] \leq A'[i + 1]$

Un algoritmo di ordinamento restituisce una permutazione di A che rispetta la relazione d'ordine

4.1 Insertion Sort

Algoritmo Ricorsivo

```

1) InsertionSort(A, n)
2)   if n > 1 then
3)     InsertionSort(A, n - 1)
4)     Insert(A, n - 1)

```

Algoritmo Iterativo

```

1) InsertionSort(A, n)
2)   for i ← 1 to n - 1 do
3)     Insert(A, i)

```

```

1) Insert(A, i)
2)   x ← A[i]
3)   j ← i - 1
4)   while ( j ≥ 0 ∧ A[j] > x ) do
5)     A[j + 1] ← A[j]
6)     j ← j - 1
7)   A[j + 1] ← x

```

$$k_i^A = | \{ (i, j) \in [0, n) : i < j \wedge A[i] > A[j] \} |$$

$$k_i^A \leq i$$

k_i^A è il numero di volte che viene eseguito il ciclo while

Analisi dell'algoritmo iterativo

$$\sum_{i=1}^n T_{insert}(A, i) = \sum_{i=1}^n \Theta(k_i^A) = \Theta\left(\sum_{i=1}^n k_i^A\right)$$

Se $k_i^A = i$ caso peggiore

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

Esempio: 10 9 8 7 6 5 4 3 2 1 0

Se $k_i^A = 1$ caso migliore

$$\sum_{i=1}^n 1 = n \quad \Theta(n)$$

Esempio: 0 1 2 3 4 5 6 7 8 9 10

Analisi dell'algoritmo ricorsivo

In termini di tempo, impiega anch'esso $\Theta(n^2)$

Cambia però in termini di spazio, a causa dello stack di sistema che viene riempito dalle chiamate ricorsive.

Occupava spazio lineare, rispetto a quello costante dell'algoritmo iterativo.

$$T_{insert_ric}(A, n) = \begin{cases} \Theta(1) & \text{se } n \leq 1 \\ T_{insert_ric}(A, n-1) + \Theta(k_{n-1}^A) & \text{altrimenti} \end{cases} =$$

$$= \begin{cases} '' & \\ T_{insert_ric}(A, n-1) + c_1 k_{n-1}^A & \end{cases} \leq \circ \leq \begin{cases} '' & \\ T_{insert_ric}(A, n-1) + c_2 k_{n-1}^A & \end{cases}$$

Semplificando le notazioni: $T^A(n) = \begin{cases} a \text{ (una certa costante)} & \text{se } n \leq 1 \\ T^A(A, n-1) + c k_{n-1}^A & \text{altrimenti} \end{cases}$

$$\begin{cases} T^A(n) = T^A(n-1) + c k_{n-1}^A \\ T^A(n-1) = T^A(n-2) + c k_{n-2}^A \\ \vdots \\ T^A(2) = T^A(1) + c k_1^A \\ T^A(1) = a \end{cases}$$

$$\sum_{i=0}^{n-1} (T^A(n-i)) =$$

$$\sum_{i=1}^{n-1} [T^A(n-i) + c k_{n-i}^A] + a = \quad \text{attenzione, ora } i \text{ parte da } 1$$

$$\sum_{i=1}^{n-1} (i) \text{ e } \sum_{i=1}^{n-1} (n-i) \text{ sono la stessa cosa,}$$

con la differenza che il primo scorre "dal basso verso l'alto" e il secondo "dall'alto verso il basso"
per comodità riscriviamo il tutto come la prima forma

$$\sum_{i=1}^{n-1} [T^A(i) + c k_i^A] + a =$$

$$\sum_{i=1}^{n-1} [T^A(i)] + c \sum_{i=1}^{n-1} k_i^A + a =$$

$$T^A(n) = \sum_{i=1}^n T^A(i) = \sum_{i=1}^{n-1} [T^A(i)] + c \sum_{i=1}^{n-1} k_i^A + a$$

$$T^A(n) = \sum_{i=1}^n T^A(i) - \sum_{i=1}^{n-1} [T^A(i)] = c \sum_{i=1}^{n-1} k_i^A + a$$

$$T^A(n) = c \sum_{i=1}^{n-1} k_i^A + a$$

Ci basta dunque sapere il valore di k_i^A

Come abbiamo visto prima:

- Se $k_i^A = 1$ è il caso migliore

$$\sum_{i=1}^{n-1} 1 = n-1$$

Abbiamo quindi $c(n-1) + a = \Theta(n)$

- Se $k_i^A = i$ è il caso peggiore

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

Abbiamo quindi $c \frac{n(n-1)}{2} + a = \Theta(n^2)$

4.2 Merge Sort

Nel merge sort, il *divid et impera* agisce in questo modo:

- **Divide**
Divide la sequenza di n elementi da ordinare in due sottosequenze di $n/2$ elementi ciascuna
- **Impera**
Vengono effettuate le chiamate ricorsive
- **Combina**
crea una sequenza ordinata unendo le due sottosequenze (già ordinate)

La ricorsione "tocca il fondo" quando la sequenza ha lunghezza 1, dato che una sequenza di un elemento è già ordinata.

L'operazione chiave è l'algoritmo di **merge**.

Questo assume che i sottoarray siano già ordinati e li fonde per creare un unico array ordinato.

```
1) Algorithm MergeSort( $A, n$ )
2)   MergeSort( $A, 0, n - 1$ )
```

```
1) MergeSort( $A, p, r$ )
2)   if  $p < r$  then
3)      $q \leftarrow \frac{p+r}{2}$ 
4)     MergeSort( $A, p, q$ )
5)     MergeSort( $A, q+1, r$ )
6)   Merge( $A, p, q, r$ )
```

Le due chiamate ricorsive, dividono la sequenza in due metà.

Il caso base è $p < r$, dove p è l'indice dell'inizio del vettore, r è quello della fine.

Se $p \geq r$ allora il vettore non è più divisibile

```
1) Merge( $A, p, q, r$ )
2)    $L \leftarrow \text{Copy}(A, p, q)$  // Copia nell'array L, l'array A nell'intervallo [p,q]
3)    $R \leftarrow \text{Copy}(A, q+1, r)$  // Copia nell'array R, l'array A nell'intervallo [q+1,r]
4)    $i, j \leftarrow 0$ 
5)   for  $k \leftarrow p$  to  $r$  do
6)     if  $(i \leq q - p) \wedge ((j \leq r - q) \vee (L[i] \leq R[j]))$  then
7)        $A[k] \leftarrow L[i]$ 
8)        $i++$ 
9)     else
10)       $A[k] \leftarrow R[j]$ 
11)      $j++$ 
```

$i \leq q - p$ controlla se l'array L è finito.

Infatti la dimensione di L è proprio $q - p$

Nel caso in cui $i > q - p$, l'*if* sarà falso, dunque entrerà sempre nell'*else*; ovvero inserisce i rimanenti elementi di R .

Altrimenti controlla lo stesso per il sotto-vettore R con $j \leq r - q$

4.2.1 Analisi

La funzione *merge* impiega $\Theta(r - p)$, visto che le copie dei vettori (riga 1,2) sommate impiegano $\Theta(r - p)$ e il ciclo *for* impiega $\Theta(r - p)$

$$T^A(p, r) = \begin{cases} a & \text{se } p \geq r \\ T^A\left(p, \frac{p+r}{2}\right) + T^A\left(\frac{p+r}{2}, r\right) + T_{\text{merge}}^A(p, r) & \text{altrimenti} \end{cases}$$

$$n = r - p + 1$$

$$T^A(n) = \begin{cases} a & \text{se } n \leq 1 \\ T^A\left(\frac{n}{2}\right) + T^A\left(\frac{n}{2}\right) + \Theta(n) & \text{altrimenti} \end{cases}$$

$$T^A(n) = \begin{cases} a & \text{se } n \leq 1 \\ T^A\left(\frac{n}{2}\right) + T^A\left(\frac{n}{2}\right) + cn & \text{altrimenti} \end{cases}$$

$$\begin{cases} T(n) = 2T\left(\frac{n}{2}\right) + cn \\ T\left(\frac{n}{2}\right) = 2T\left(\frac{n}{4}\right) + c\frac{n}{2} \\ T\left(\frac{n}{4}\right) = 2T\left(\frac{n}{8}\right) + c\frac{n}{4} \\ \vdots \\ T(1) = a \end{cases} \quad \begin{array}{l} \text{multiplico per } 2^k \text{ per potere semplificare} \\ k \text{ rappresenta il numero di passaggi che} \\ \text{occorrono nella ricorsione.} \\ \text{Il primo livello parte da } k = 0 \end{array} \quad \begin{cases} T(n) = 2T\left(\frac{n}{2}\right) + cn \\ 2T\left(\frac{n}{2}\right) = 4T\left(\frac{n}{4}\right) + cn \\ 4T\left(\frac{n}{4}\right) = 8T\left(\frac{n}{8}\right) + cn \\ \vdots \\ 2^k T(1) = 2^k a \end{cases}$$

Semplificando ottengo:

$$T(n) = kn + 2^k a$$

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

Sommatoria del lato sinistro
della catena di uguaglianze

Sommatoria del lato destro della catena di uguaglianze

Escludo il caso k dato che l'ultima uguaglianza ha una forma diversa

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

$$\sum_{i=0}^{k-1} [cn] = cn \sum_{i=0}^{k-1} [1] = cnk$$

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

$$\text{faccio partire la sommatoria da } i = 1 \rightarrow \sum_{i=1}^k \left[2^i T\left(\frac{n}{2^i}\right) n \right]$$

estraggo il primo elemento di $\sum_{i=0}^k \left[2^i T\left(\frac{n}{2^i}\right) \right]$ ovvero $T(n)$ e faccio partire la sommatoria da $i = 1$

$$T(n) + \sum_{i=1}^k \left[2^i T\left(\frac{n}{2^i}\right) \right] = \sum_{i=1}^k \left[2^i T\left(\frac{n}{2^i}\right) n \right] + cnk + 2^k a$$

$$T(n) = cnk + 2^k a$$

Quanto vale k ?

k agisce in funzione di n

$$\text{quindi } T\left(\frac{n}{2^k}\right)$$

Il caso base lo ottengo quando $\left(\frac{n}{2^k}\right) \leq 1$

$$\frac{n}{2^k} \leq 1 \Rightarrow n \leq 2^k \Rightarrow \log(n) \leq k$$

$$T(n) = cnk + 2^k a = cn \log(n) + na = \Theta(n \log(n))$$

4.3 Selection Sort

Il selection sort è un algoritmo di ordinamento che opera in place.

Consiste di selezionare di volta in volta il minimo valore presente nella sequenza $A[i..n]$ e lo si sposta alla i -esima posizione.

Così facendo si viene a creare una sottosequenza ordinata da 0 ad $i - 1$, trovato il valore minimo lo si sposta all' i -esima posizione e così via.

```
1) SelectionSort( $A, n$ )
2)   for  $i \leftarrow n - 1$  down to 1 do
3)      $m \leftarrow \text{Max}(A, i)$ 
4)     Swap( $A, i, m$ )
```

```
1) Max( $A, i$ )
2)    $m \leftarrow 0$ 
3)   for  $j \leftarrow 1$  to  $i$  do
4)     if  $A[m] < A[j]$  then
5)        $m \leftarrow j$ 
```

4.3.1 Analisi

La funzione Max impiega $\Theta(i)$, dato che le istruzioni all'interno del *for* sono tutte costanti, si ottiene quindi:

$$\sum_{j=1}^i 1 = i$$

La funzione Max viene eseguita $n - 1$ volte, nel *for* della funzione SelectionSort .

Il tutto impiega $\sum_{i=1}^{n-1} \Theta(i) = \sum_{i=1}^{n-1} i = \frac{n(n-1)}{2} = \Theta(n^2)$

4.4 Heap Sort

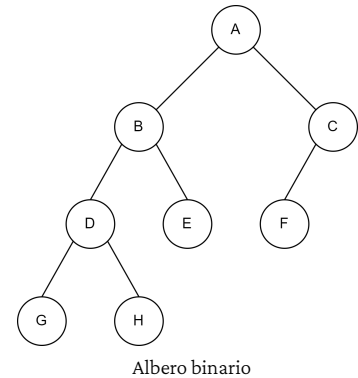
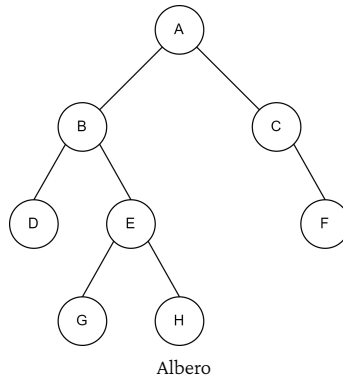
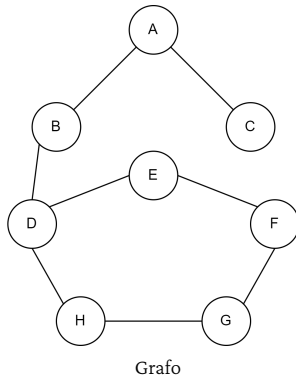
4.4.1 Albero

Un albero è un grafo composto da nodi, che contengono il dato, i quali sono suddivisi in:

- Nodo padre
- Nodo figlio

Rispetto ai grafi, i nodi di un albero possono avere al massimo un genitore.

Un albero si dice **binario** se ogni nodo ha al massimo 2 figli



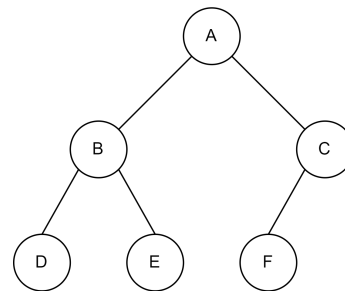
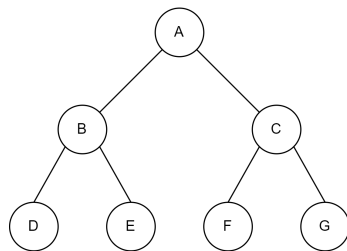
Albero binario completo e quasi completo

Un albero binario si dice completo se:

- è pieno
ossia, tutti i nodi hanno entrambi i figli
- ogni sua foglia ha la stessa profondità
un nodo si dice foglia quando non ha figli

Un albero binario si dice **quasi** completo se:

- è pieno **fino al penultimo livello**
- i nodi dell'ultimo livello sono inseriti da sinistra a destra



4.4.2 Definizione di ALTEZZA di un albero binario

L'altezza misura la massima distanza di una foglia dalla radice dell'albero, in termini di numero di archi attraversati.

Prendiamo ad esempio l'albero binario completo nella figura sopra, questo ha altezza 2 a partire dal nodo A.

La stessa definizione però si applica ai sottoalberi. Quindi i figli della radice, B e C, avranno altezza 1, e così via fino ad arrivare alle foglie che avranno altezza 0.

4.4.3 Proprietà di Heap

Un albero binario **completo o quasi completo**, possiede la proprietà di **heap** se il valore del nodo padre è ordinato rispetto ai valori dei figli.

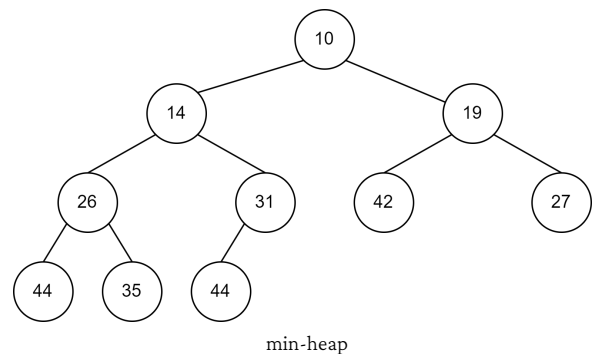
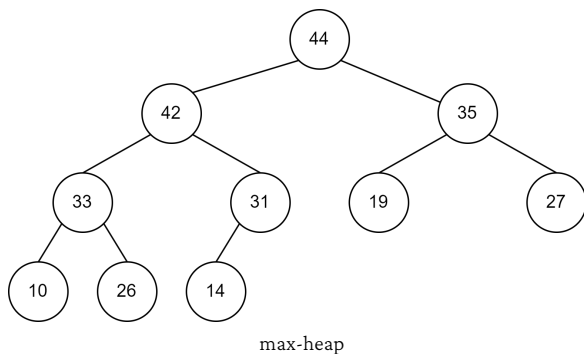
Gli heap possono essere suddivisi in

- **max-heap**

In un max heap, i valori di ogni nodo sono sempre maggiori o uguali di quelle dei figli, e il valore massimo appartiene alla radice.

- **min-heap**

In un min heap, i valori di ogni nodo sono sempre minori o uguali di quelle dei figli, e il valore minimo appartiene alla radice.



In questo modo si garantisce che compiendo un qualsiasi percorso che parte da un nodo dell'albero e scendendo verso le foglie, si attraversano nodi con valore sempre maggiore (o minore) dell'ultimo nodo visitato.

Numero di nodi in un albero binario completo

$$n = 2^{h+1} - 1$$

Ogni nodo ha due figli, quindi all' i -esimo livello ci saranno 2^i figli.

Quindi l'intero albero avrà: $n = \sum_{i=0}^h 2^i$ nodi

Questa è la somma dei primi h termini di una serie geometrica di ragione 2

una serie geometrica di ragione $q \geq 1$ diverge positivamente $\sum_{k=0}^n q^k = \frac{1 - q^{n+1}}{1 - q}$

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

Altezza di un albero binario completo di n nodi

$$h \geq \lceil \log_2(n + 1) - 1 \rceil$$

oppure

$$h \geq \lfloor \log_2(n) \rfloor$$

$$2^{h+1} - 1 = n \quad \Rightarrow$$

$$2^{h+1} = n + 1 \quad \Rightarrow$$

$$\log_2(2^{h+1}) = \log_2(n + 1) \quad \Rightarrow$$

$$h + 1 = \log_2(n + 1) \quad \Rightarrow$$

$$h = \log_2(n + 1) - 1$$

Se l'albero è quasi completo, allora: $h \geq \lceil \log_2(n + 1) - 1 \rceil$

4.4.4 Algoritmo

```

1) HeapSort( $A, n$ )
2)   BuildHeap( $A, n$ )
3)   for  $i \leftarrow n - 1$  to 1 do
4)     Swap( $A, 0, i$ ) // il max sta in  $A[0]$  perché è un heap
5)     Heapify( $A, i, 0$ )

```

```

1) BuildHeap( $A, n$ )
2)   for  $i \leftarrow ((n/2) - 1)$  down to 0 do
3)     Heapify( $A, n, i$ )

```

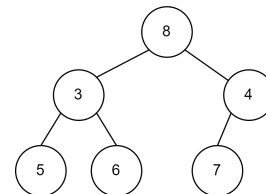
```

1) Heapify( $A, n, i$ )
2)    $max \leftarrow i$ 
3)    $l \leftarrow L(i)$  //  $L(i) = 2i + 1$  indice del figlio sinistro
4)    $r \leftarrow R(i)$  //  $R(i) = 2i + 1$  indice del figlio destro
5)   if ( $l < n$ ) AND ( $A[max] < A[l]$ ) then
6)      $max \leftarrow l$ 
7)   if ( $r < n$ ) AND ( $A[max] < A[r]$ ) then
8)      $max \leftarrow r$ 
9)   if ( $max \neq i$ ) then
10)    Swap( $A, max, i$ )
11)    Heapify( $A, n, max$ )

```

Un array è già rappresentato in memoria come un albero, infatti è possibile passare da una struttura all'altra.

8	3	4	5	6	7
---	---	---	---	---	---



Come prima cosa viene eseguito *BuildHeap*, che trasforma la sequenza in input in un max-heap.

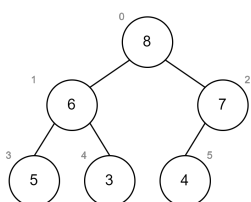
BuildHeap esegue la funzione *Heapify* a partire dall'indice $i = (n/2) - 1$, ovvero l'indice dell'ultimo nodo non foglia, che nell'esempio sopra è il nodo con il valore 4.

A questo punto avrò all'indice 0 il massimo della sequenza (garantito dalla proprietà di max-heap).

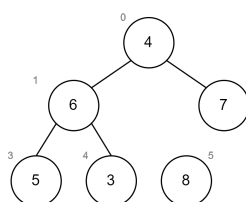
Quindi scambio l'elemento all'indice 0 con quello all'indice i , questo perché l'indice i sta scorrendo al contrario. L'elemento appena scambiato ora si troverà nella posizione corretta, viene quindi "scollegato" dall'albero. L'indice i scorre al contrario, quindi al ciclo successivo (con $i - 1$) quell'elemento non verrà più considerato.

Ora però non si possiede più la proprietà di max-heap, di conseguenza si esegue la funzione *Heapify* che riparerà il danno creato dalla funzione *Swap*.

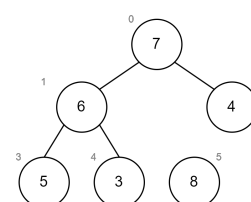
0	1	2	3	4	5
8	6	7	5	3	4

struttura dopo *BuildHeap*

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

struttura dopo lo *Swap*

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

struttura dopo *Heapify*

Funzione Heapify

La funzione *Heapify* rende heap l'albero di dimensione n a partire dalla radice i .

I primi due *if*, controllano quale dei due figli è maggiore della radice.

Nella variabile *max* verrà inserito l'indice del figlio di sinistra o di destra a seconda del caso.

Se nessuno dei due figli è più grande della radice, allora $max = i$ di conseguenza il terzo *if* non viene eseguito.

Altrimenti viene eseguito lo swap fra la radice e il figlio più grande.

Viene poi chiamato nuovamente la *Heapify* per rendere heap anche i figli.

Ad esempio, nel **BuildHeap** si rende heap prima il sotto albero che ha come radice l'ultimo nodo **non** foglia, poi il penultimo e così via risalendo l'albero; così da creare un max-heap.

Essendo *Heapify* ricorsiva, prima rende heap un albero, successivamente rende heap i figli (nel caso non fossero foglie), poi i figli dei figli e così via fino ad arrivare alle foglie.

Se i figli non sono heap, una volta resi heap dalla chiamata ricorsiva, non ci sarebbero problemi con il padre data la proprietà di heap (ovvero il padre è ordinato rispetto ai figli). Ossia: reso il padre un heap, il problema viene passato ai figli, e una volta resi heap i figli, il problema poi **non** viene passato al padre.

Una volta trasformata la sequenza in un max-heap, all'interno del *for* della funzione **HeapSort**, viene eseguito lo *Swap* tra l'elemento massimo (che si trova all'indice 0 perché è un max-heap) e l'elemento all'indice i (indice che sta scorrendo al contrario).

Così facendo si sposta il massimo alla fine del vettore, però c'è bisogno di eseguire *Heapify* perché all'indice 0 è stato inserito un valore potenzialmente scorretto; quindi bisogna rendere nuovamente max-heap l'intero albero ma escludendo di volta in volta gli elementi posizionati correttamente.

Viene chiamata *Heapify* con il parametro $n=i$, ovvero l'altezza dell'albero meno uno (quindi si escludono gli elementi posizionati correttamente).

Invece il parametro i (ovvero la radice dalla quale partire per costruire un heap) sarà sempre 0 perché si deve ricostruire l'intero albero.

4.4.5 Analisi

Heapify

Data la ricorsività, *Heapify* scorre l'albero in altezza, ovvero $\log(n)$. Il resto delle operazioni sono eseguite a tempo costante, quindi in totale il costo di *Heapify* è $\Theta(\log(n))$

Heapify all'interno del for di HeapSort

La chiamata viene fatta su un albero di dimensione i a partire dalla radice di indice 0.

Quindi scorre l'intero albero in altezza ad ogni chiamata, ovvero $\log(i)$

Heapify viene chiamata $n - 1$ volte nel *for*:

$$\begin{aligned} \sum_{i=1}^{n-1} \log(i) &= \log\left(\prod_{i=1}^{n-1} i\right) = \log((n-1)!) = \text{applichiamo l'approssimazione di Stirling} \\ &= \log\left(\sqrt{2\pi n} \left(\frac{n}{e}\right)^n\right) = \log\left(\sqrt{2\pi} \sqrt{n} \frac{n^n}{e^n}\right) = \log(\sqrt{2\pi}) + \frac{1}{2} \log(n) + n \log(n) - n \log(e) \end{aligned}$$

$n \log(n)$ è il termine che asintoticamente cresce più velocemente.

BuildHeap

Il costo di *Heapify* ad ogni iterazione nel *for* è di $\frac{n}{2^{i+1}} \cdot i$, avrò quindi:

$$\sum_{i=1}^k \left[\frac{n}{2^{i+1}} \cdot i \right] = \sum_{i=1}^k \left[\frac{n}{2^{i-1} 2^2} \cdot i \right] = \frac{n}{4} \sum_{i=1}^k \left[i \cdot \frac{1}{2^{i-1}} \right] = \frac{n}{4} \sum_{i=1}^k \left[i \cdot \left(\frac{1}{2}\right)^{i-1} \right] = \frac{n}{4} \sum_{i=1}^k [i \cdot x^{i-1}]_{x=\frac{1}{2}}$$

Possiamo riscrivere $i \cdot x^{i-1}$ come la derivata di x^i $\frac{d}{dx} x^i = i \cdot x^{i-1}$

$$= \left[\frac{n}{4} \sum_{i=1}^k \frac{d}{dx} x^i \right]_{x=\frac{1}{2}} = \left[\frac{n}{4} \frac{d}{dx} \sum_{i=1}^k x^i \right]_{x=\frac{1}{2}} \quad \text{approssimo alla serie geometrica}$$

$$= \left[\frac{n}{4} \frac{d}{dx} \sum_{i=1}^{\infty} x^i \right]_{x=\frac{1}{2}} = \left[\frac{n}{4} \frac{d}{dx} \frac{1}{1-x} \right]_{x=\frac{1}{2}} = \left[\frac{n}{4} \frac{1}{(1-x)^2} \right]_{x=\frac{1}{2}} = \frac{n}{4} \cdot 4 = n$$

HeapSort

Il costo dell'intero algoritmo sarà quindi $O(n \log n)$

4.5 Quick Sort

Quick sort è un algoritmo che impiega, nel caso peggiore, $\Theta(n^2)$. Mediamente però, impiega $\Theta(n \log n)$ con costanti moltiplicative molto basse.

Nel quick sort, il divide et impera agisce in questo modo:

- **Divide**

Partiziona l'array $A[p..r]$ in due sottoarray $A[p..q-1]$ e $A[q+1..r]$

in modo tale che $A[p..q-1] \leq A[q] \leq A[q+1..r]$.

Ogni elemento del primo sottoarray è minore o uguale di $A[q]$ che a sua volta è minore o uguale di tutti gli elementi del secondo sottoarray.

I due sottoarray saranno disordinati, ma intanto si ha la proprietà che gli elementi del primo sono minori o uguali degli elementi del secondo.

- **Impera**

si chiama ricorsivamente QuickSort

- **Combina**

Non occorre ordinare i due sottoarray essendo già ordinati tra di loro

```

1) Algorithm QuickSort( $A, n$ )
2)   QuickSort( $A, 0, n-1$ )

```

```

1) QuickSort( $A, p, r$ )
2)   if  $p < r$  then
3)      $q \leftarrow \text{Partition}(A, p, r)$ 
4)     QuickSort( $A, p, q$ )
5)     QuickSort( $A, q+1, r$ )

```

```

1) Partition( $A, p, r$ )
2)    $x \leftarrow A[p]$ 
3)    $i \leftarrow p-1$ 
4)    $j \leftarrow r+1$ 
5)   repeat
6)     repeat
7)        $j--$ 
8)     until ( $A[j] \leq x$ )
9)     repeat
10)     $i++$ 
11)    until ( $A[i] \geq x$ )
12)    if  $i < j$  then
13)      Swap( $A, i, j$ )
14)  until ( $j \leq i$ )
15)  return  $j$ 

```

Il *repeat..until* è come un *do..while* ma con la condizione negata. Ripeti finché NON si verifica la condizione.

x sarà il pivot (perno) intorno al quale verrà partizionato l'array $A[p..r]$

Essendo i incrementato all'interno di un *repeat until* (quindi almeno una volta viene incrementato), allora partirà da $p-1$ e non da p . Stesso discorso per j .

Proprietà Partition

1. $p \leq q < r$
2. $\forall p \leq i \leq q \quad \forall q+1 \leq j \leq r, \quad A[i] \leq A[j]$

4.5.1 Analisi

Definiamo *rango del pivot* q , il numero di elementi nell'array che sono $\leq q$

$$| \{ i \in (p, r) : A[i] \leq A[q] \} |$$

In sostanza, quanti valori ci sono a sinistra del pivot

$$1 \leq \text{rango} \leq n = r - p + 1$$

Il rango sarà 1 se $A[q]$ è il minimo, e quindi NON ci sono elementi a sinistra del pivot.

Il rango sarà n se $A[q]$ è il massimo, e quindi TUTTI gli elementi sono a sinistra del pivot.

rango	q	$ [p, q] $
1	p	1
2	p	1
3	$p + 1$	2
\vdots	\vdots	\vdots
n	$p + n - 2$	

q è l'indice del pivot, ovvero j restituito dalla funzione *Partition*

Quando il rango è 1, l'unico elemento \leq del pivot, è il pivot stesso; quindi

j verrà decrementato fino ad arrivare a p ,

i viene incrementato solo una volta, quindi arriva a p ,

non viene fatto alcuno *Swap*,

esce dal *repeat..until* esterno,

restituisce j ovvero p .

Quando il rango è 2, c'è un elemento \leq del pivot e il pivot stesso; quindi

j verrà decrementato fino ad un certo indice,

i viene incrementato solo una volta, quindi arriva a p ,

viene fatto lo *Swap* fra gli elementi $A[i]$ e $A[j]$,

viene ripetuto il *repeat..until* esterno perché $j > i$,

nel secondo ciclo del *repeat..until* esterno,

j viene decrementato almeno una volta,

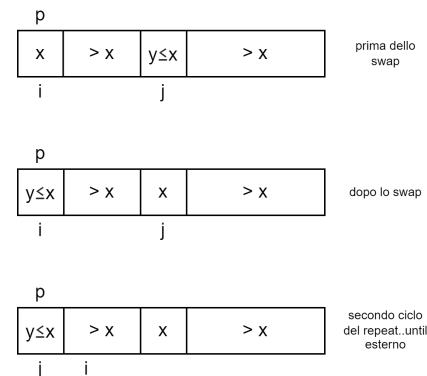
successivamente trova tutti elementi $> x$,

quindi viene decrementato fino ad arrivare a p

i viene incrementato almeno una volta, arriva a $p + 1$

esce dal *repeat..until* esterno perché $p = j \leq i = p + 1$,

restituisce j ovvero p



Stesso ragionamento quando il rango è 3, questa volta j si fermerà a $p + 1$, e così via

Analisi di *Partition*

Con i *while*, o *repeat..until*, è difficile misurare il tempo che impiegano.

Il *repeat..until* più esterno, farà al massimo $\frac{n}{2}$ iterazioni.

I due *repeat..until* interni, accederanno ad ogni cella una sola volta; dato che lavorano su celle diverse.

Complessivamente la funzione impiega $\Theta(n)$

Analisi del tempo medio

Sia α il rango del pivot

$$T_M(n) = \begin{cases} k_1 & \text{se } n \leq 1 \\ \frac{1}{n} \sum_{\alpha=1}^n [T_M(q_\alpha) + T_M(n - q_\alpha) + k_2 n] & \text{se } n > 1 \end{cases}$$

$$\frac{1}{n} \text{ perché sto calcolando la media } \frac{\sum_{\alpha=1}^n}{n}$$

$k_2 n$ viene da $\Theta(n)$ (tempo di *Partition*)

q_α è l'indice del pivot quando il rango è α

$$T_M(n) = \frac{1}{n} \sum_{\alpha=1}^n [T_M(q_\alpha) + T_M(n - q_\alpha) + k_2 n]$$

$$T_M(n) = \frac{1}{n} \left[T_M(q_1) + T_M(n - q_1) + k_2 n + \sum_{\alpha=2}^n [T_M(q_\alpha) + T_M(n - q_\alpha) + k_2 n] \right]$$

Estraggo il primo elemento della sommatoria.

α partirà da 2

$q_1 = 1$ dalla tabella sopra

rango	q	$ [p, q] $
1	p	1
2	p	1
3	$p + 1$	2
\vdots	\vdots	\vdots
n	$p + n - 2$	

$$\underbrace{T_M(1)}_{k_1} + \underbrace{T_M(n-1)}_{k_3 n^2} + k_2 n \quad \text{approssimo ad un andamento quadratico}$$

$$k_1 + k_3 n^2 + k_2 n = n \left(\frac{k_1}{n} + k_3 n + k_2 \right)$$

$$T_M(n) = \frac{1}{n} \left[n \left(\frac{k_1}{n} + k_3 n + k_2 \right) \right] + \frac{1}{n} \sum_{\alpha=2}^n [T_M(q_\alpha) + T_M(n - q_\alpha) + k_2 n] =$$

semplifico le due n a sinistra

$$= \underbrace{\left(\frac{k_1}{n} + k_3 n + k_2 \right)}_{k_4} + \frac{1}{n} \sum_{\alpha=2}^n [T_M(q_\alpha) + T_M(n - q_\alpha) + k_2 n]$$

$$= k_4 + \frac{1}{n} \sum_{\alpha=2}^n [T_M(q_\alpha) + T_M(n - q_\alpha) + k_2 n] \leq$$

$$\leq k_4 + \frac{1}{n} \sum_{\alpha=2}^n [T_M(\alpha - 1) + T_M(n - (\alpha - 1)) + k_2 n] = \quad \text{Imposto } \alpha - 1 = i$$

$$k_4 + \frac{1}{n} \sum_{i=1}^{n-1} [T_M(i) + T_M(n - i) + k_2 n] =$$

$$k_4 + \frac{1}{n} k_2 n(n-1) + \frac{1}{n} \sum_{i=1}^{n-1} [T_M(i) + T_M(n - i)] = \quad \text{Ho portato fuori dalla sommatoria } k_2 n \text{ per } (n-1) \text{ volte}$$

$$(k_4 k_2)n - k_2 + \frac{1}{n} \sum_{i=1}^{n-1} [T_M(i) + T_M(n - i)] =$$

$$(k_4 k_2)n - k_2 + \frac{2}{n} \sum_{i=1}^{n-1} T_M(i)$$

$$\frac{1}{n} \sum_{i=1}^{n-1} [T_M(i) + T_M(n - i)] = \frac{2}{n} \sum_{i=1}^{n-1} T_M(i) \quad \text{perché } T_M(i) \text{ compare due volte nella sommatoria}$$

$$i = 1, \quad T(i) = T(1) \quad i = n-1, \quad T(n-i) = T(n - (n-1)) = T(1)$$

$$i = 2, \quad T(i) = T(2) \quad i = n-2, \quad T(n-i) = T(n - (n-2)) = T(2)$$

e così via

$$T_M(n) = \begin{cases} k_1 & \text{se } n \leq 1 \\ (k_4 k_2)n - k_2 + \frac{2}{n} \sum_{i=1}^{n-1} T_M(i) & \text{altrimenti} \end{cases}$$

Ora si vuole dimostrare (per **induzione**) che $T(n) \leq c n \log n$ oppure che $T(n) = O(n \log n)$

Passo base: $n = 2$

$$T_M(2) = (k_4 k_2)2 - k_2 + \underbrace{\frac{2}{2} \sum_{i=1}^{2-1} T_M(i)}_{T_M(1)=k_1} = (k_4 k_2)2 - k_2 + k_1$$

Per una costante $c \geq (k_4 k_2)2 - \frac{k_2}{2} + \frac{k_1}{2}$

$$T_M(2) \leq c 2 \log 2$$

Ipotesi induttiva: $\forall p \leq n, \quad T_M(p) \leq c p \log p \Rightarrow T_M(n+1) \leq c n \log n$

Sia $(k_4 k_2) = k_5$

$$T_M(n) = k_5 n - k_2 + \frac{2}{n} \sum_{p=1}^{n-1} T_M(p) \leq k_5 n - k_2 + \frac{2}{n} c \underbrace{\sum_{p=1}^{n-1} [p \log p]}_{\text{sviluppiamo questa sommatoria a parte}}$$

$$\sum_{p=1}^{n-1} [p \log p] = \underbrace{\sum_{p=1}^{\frac{n-1}{2}-1} [p \log p]}_{\log(p) \mapsto \log \frac{n}{2}} + \underbrace{\sum_{p=\frac{n-1}{2}}^{n-1} [p \log p]}_{\log(p) \mapsto \log(n)} \leq \quad \text{divido la sommatoria in due per cercare una buona approssimazione **superiore**}$$

sia $h = \frac{n-1}{2}$

$$\leq \sum_{p=1}^{h-1} \left[p \log \frac{n}{2} \right] + \sum_{p=h}^{n-1} [p \log n] = (\log n - \underbrace{\log 2}_{=1}) \sum_{p=1}^{h-1} p + \log n \sum_{p=h}^{n-1} p =$$

$$\log n \sum_{p=1}^{h-1} p - \sum_{p=1}^{h-1} p + \log n \sum_{p=h}^{n-1} p = \log n \underbrace{\left(\sum_{p=1}^{h-1} p + \sum_{p=h}^{n-1} p \right)}_{\sum_{p=1}^{n-1} p} - \underbrace{\sum_{p=1}^{h-1} p}_{\frac{n(n-1)}{2}} =$$

$$(\log n) \left(\frac{n(n-1)}{2} \right) - \frac{h(h-1)}{2} \leq \frac{1}{2} (\log n) n^2 - \frac{n^2}{8}$$

Svolta quella sommatoria, abbiamo infine:

$$T_M(n) = k_5 n - k_2 + \frac{2}{n} \sum_{p=1}^{n-1} T_M(p) \leq k_5 n - k_2 + \frac{2c}{n} \left[\frac{n^2}{2} (\log n) - \frac{n^2}{8} \right] = c n \log n + k_5 n - \frac{c n}{4} - k_2$$

Si ricorda che si vuole dimostrare che $T_M(n) \leq c n \log n$, ovvero che $c n \log n + k_5 n - \frac{c n}{4} - k_2 \leq c n \log n$

Questo è vero quando $k_5 n - \frac{c n}{4} - k_2 \leq 0 \Rightarrow c \geq 4k_5 - \frac{4k_2}{n}$

4.6 Limite degli ordinamenti per confronto

Un qualsiasi algoritmo di ordinamento per confronto deve effettuare $\Omega(n \log n)$ nel caso peggiore, per ordinare n elementi.

Un ordinamento per confronti può essere visto come un *albero di decisione*.

4.6.1 Albero di decisione

Un albero di decisione è un albero binario pieno (tutti i nodi hanno entrambi i figli) che rappresenta i confronti fra gli elementi di un array.

L'ordinamento, in un'albero di decisione, consiste nel tracciare un percorso a partire dalla radice fino ad una foglia; ogni nodo attraversato rappresenta un confronto e la foglia contiene la sequenza ordinata.

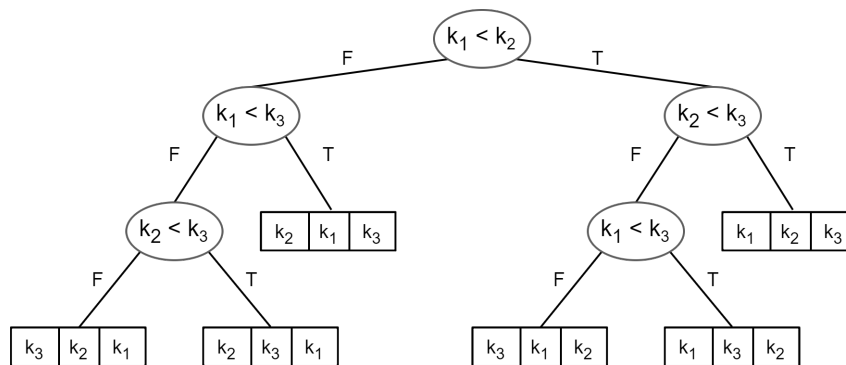
Un algoritmo di ordinamento per confronti, dev'essere in grado di generare tutte le $n!$ permutazioni dell'array in input, dato che solo una di questa sarà la sequenza ordinata.

Di conseguenza, l'albero di decisione avrà $n!$ foglie, dato che ogni foglia rappresenta la sequenza ordinata in base alla sequenza in input.

La lunghezza di un percorso è paragonabile al tempo di esecuzione dell'algoritmo d'ordinamento:

- il percorso più **lungo** è il caso peggiore dell'algoritmo
- il percorso più **breve** è il caso migliore dell'algoritmo

Prendiamo ad esempio, una sequenza (k_1, k_2, k_3)
questo è il suo albero di decisione



Un albero di altezza h ha 2^h foglie.

Essendo l'albero di decisione completo fino all'altezza $h - 1$, e avendo $n!$ foglie, si ha che $2^{h-1} \leq n! \leq 2^h$

$$2^{h-1} \leq n! \leq 2^h$$

$$h - 1 \leq \log(n!) \leq h$$

$$\log(n!) = \log\left(\sqrt{2\pi n} \left(\frac{n}{e}\right)^n\right) = \log(\sqrt{2\pi}) + \frac{1}{2}\log(n) + n\log(n) + n\log(e) = \Theta(n\log(n))$$

Si ottiene dunque che $\Theta(n\log(n)) \leq h$ ossia che $h = \Omega(n\log(n))$

Di conseguenza, un ordinamento per confronti, nel caso peggiore non può fare meno di $n\log(n)$ confronti

5 Strutture dati

In informatica, gli insiemi possono variare nel tempo (crescere, ridursi); questi sono detti **dinamici**.

Esistono diversi modi per manipolarli; in base al tipo di dato da salvare, si utilizzerà l'opportuna struttura dati e le opportune tecniche per inserire, eliminare e modificare.

Ogni elemento dell'insieme è rappresentato da un oggetto che contiene dei puntatori ad altri oggetti dell'insieme; avrà inoltre tutti gli attributi necessari per immagazzinare i dati.

5.1 Operazioni

Operazioni tipiche:

- $Search(S, d) : x \mid NULL$
Cerca nella struttura S un elemento che possiede il dato d .
Se lo trova, restituisce il puntatore x a quell'elemento, altrimenti restituisce $NULL$
- $Insert(S, d) : S'$
Inserisce nella struttura S un nuovo elemento con il dato d
Restituisce una nuova struttura S' , ovvero la struttura con il nuovo elemento
- $Delete(S, d) : S'$
Rimuove dalla struttura S un elemento con il dato d
Restituisce una nuova struttura S' , ovvero la struttura con l'elemento rimosso
- $IsEmpty(S) : bool$
Restituisce $true$ se la struttura S è vuota, $false$ altrimenti

Ogni struttura può avere delle operazioni specifiche, ad esempio:

$Stack(D)$	Nome della classe, in questo caso uno Stack (LIFO) D è un generico tipo di dato
$Push(D) : S$	Inserisce in testa nello stack e restituisce il nuovo stack
$Pop() : S$	Rimuove dallo stack e restituisce il nuovo stack
$Top() : D$	Restituisce la testa dello stack
$Pop\&Top() : (S, D)$	Restituisce la testa e il nuovo stack, ed la elimina la testa
...	...

$Queue(D)$	Coda (FIFO)
$Enqueue() : S$	Inserisce in coda e restituisce la nuova coda
$Dequeue() : S$	Rimuove dalla testa e restituisce la nuova coda
$Head() : D$	Restituisce la testa
$Head\&Dequeue() : (S, D)$	Restituisce la testa e la nuova coda, ed la elimina la testa
...	...

Ogni struttura può avere delle proprietà particolari, ad esempio:

- $Equality(D)$
Si richiede che gli elementi siano confrontabili (altrimenti non è possibile fare la ricerca)
- $Ordered(D)$
Si richiede che gli elementi siano ordinati
- ...

5.2 Alberi binari

Un albero binario, è un albero che ha al massimo due figli.

Un albero binario è un ottimo compromesso tra un array e una lista.

L'array è rigido ma poco flessibile

La lista è il viceversa

È possibile scorrere un albero binario principalmente in due modi:

- In **ampiezza** (Breadth First Visit)
- In **profondità** (Depth First Visit):
 - **pre-order**
viene visitato prima il padre, poi il sinistro, poi il destro
 - **in-order**
viene visitato prima il figlio sinistro, poi il padre, poi il destro
 - **post-order**
viene visitato prima il figlio sinistro, poi il destro, poi il padre

```

1) DFVPre(X, F, a)
2)   if x ≠ ⊥ then
3)     a ← F(x.dato, a)
4)     a ← DFVPre(x.sx, F, a)
5)     a ← DFVPre(x.dx, F, a)
6)   return a

```

```

1) DFVIn(X, F, a)
2)   if x ≠ ⊥ then
3)     a ← DFVIn(x.sx, F, a)
4)     a ← F(x.dato, a)
5)     a ← DFVIn(x.dx, F, a)
6)   return a

```

```

1) DFVPost(X, F, a)
2)   if x ≠ ⊥ then
3)     a ← DFVPost(x.sx, F, a)
4)     a ← DFVPost(x.dx, F, a)
5)     a ← F(x.dato, a)
6)   return a

```

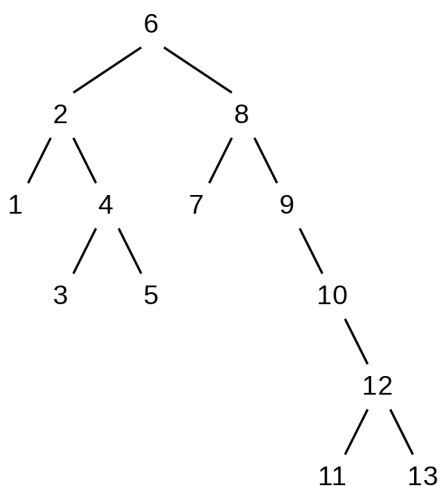
x è il nodo della radice di un albero (quindi anche i sottoalberi)

F è una funzione che utilizza il dato e restituisce un valore

$$F : D \times A \rightarrow A$$

a valore iniziale dell'accumulatore

Esempio:



Ampiezza	6	2	8	1	4	7	9	3	5	10	12	11	13
Pre-Order	6	2	1	4	3	5	8	7	9	10	12	11	13
In-Order	1	2	3	4	5	6	7	8	9	10	11	12	13
Post-Order	1	3	5	4	2	7	11	13	12	10	9	8	6

Per la visita in ampiezza c'è bisogno di una struttura dati accessoria, la coda.

Estraggo il nodo in testa, lo utilizzo, poi accodo i figli nel caso li abbia.
verrà inizialmente accodata la radice fuori dal *repeat..until*

```

1) BFV( $X, F, a$ )
2)   if  $x \neq \perp$  then
3)      $Q \leftarrow \text{Enqueue}(\text{EmptyQueue}, x)$ 
4)     repeat
5)        $x \leftarrow \text{Head\&Dequeue}(Q)$ 
6)        $a \leftarrow F(x.dato, a)$ 
7)       if  $x.sx \neq \perp$  then  $Q \leftarrow \text{Enqueue}(Q, x.sx)$ 
8)       if  $x.dx \neq \perp$  then  $Q \leftarrow \text{Enqueue}(Q, x.dx)$ 
9)     until (  $\text{isEmpty}(Q)$  )
10)  return  $a$ 

```

Q è la coda, la struttura d'appoggio

Enqueue inserisce la radice x in *EmptyQueue*

EmptyQueue è una generica coda vuota

Dato che l'algoritmo non sfrutta la struttura dell'albero, ma quella della coda, conviene farlo iterativo.

Lo scorrimento in ampiezza scorre tutti i nodi, quindi non può fare meglio di $n \quad \Omega(n)$

Un nodo, una volta visitato, non verrà più visitato, quindi non può fare peggio di $n \quad O(n)$

5.3 Casi d'uso degli scorrimenti di un albero

Quali fra gli scorrimenti visti conviene scegliere?

Dipende dal dominio di applicazione dell'albero e di conseguenza da quale forma assumerà.

Se l'albero tende ad essere **basso e ampio**, conviene uno scorrimento in **profondità**;

la cui complessità sarà $O(h)$ o $\log(n)$ h : altezza n : numero di nodi

Conviene invece uno scorrimento in **ampiezza** se l'albero è **sbilanciato**

5.4 Alberi binari di ricerca (Binary Search Tree)

Un BST è un particolare tipo di struttura dati che permette di effettuare in modo efficiente operazioni come: *ricerca*, *inserimento*, *cancellazione*, *ricerca del massimo/minimo*, ...

Un BST, come dice la parola, è organizzato in un albero binario;

ogni nodo, oltre agli attributi per i dati, contiene gli attributi sx e dx che puntano al figlio sinistro e destro.

Un BST ha le seguenti proprietà:

- Il **sottoalbero sinistro** di un nodo x **contiene soltanto i nodi con chiavi minori** della chiave del nodo x
- Il **sottoalbero destro** di un nodo x **contiene soltanto i nodi con chiavi maggiori** della chiave del nodo x
- Il sottoalbero destro e il sottoalbero sinistro devono essere entrambi due alberi binari di ricerca.

$$\forall x \in T, \quad \underbrace{\forall y \in T_{x.sx}}_{\text{discendenti sinistri}}, \quad \underbrace{\forall z \in T_{x.dx}}_{\text{discendenti destri}} \quad y.dato < x.dato < z.dato$$

Un BST, di base viene letto **in-order**

5.4.1 Ricerca

Dato un puntatore alla radice e un dato, *Search* **restituisce il puntatore al nodo con quella chiave (NULL se la chiave non è presente)**

```

1) Search(x, d)
2)   if x = ⊥ then
3)     return ⊥
4)   else
5)     if d > x.dato then
6)       return Search(x.dx, d)
7)     else if d < x.dato then
8)       return Search(x.sx, d)
9)     else
10)      return x

```

Se il nodo attuale è NULL, allora non ho trovato l'elemento e restituisco NULL.

Se invece il nodo attuale non è NULL, cerco a destra se **il dato da cercare** è più grande del dato contenuto nel nodo, altrimenti cerco a sinistra.

Continuo così finché o trovo il dato o sono arrivato alle foglie.

Nel caso peggiore, il percorso più lungo sarà quanto l'altezza dell'albero.

L'operazione di ricerca impiega quindi $O(h)$

5.4.2 Inserimento

È importante che la struttura di BST persista anche dopo l'inserimento.

Per semplicità non considereremo anche il caso dei duplicati; dato che se ci fossero, basterebbe tener traccia di quanti valori duplicati ci sono con un contatore nel nodo in questione.

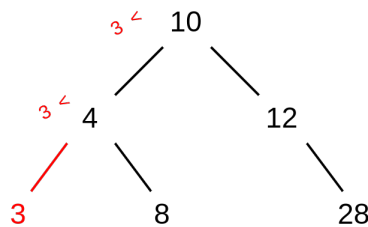
```

1) Insert( $x, d$ )
2)   if  $x = \perp$  then
3)     return BuildNode( $d$ )
4)   else
5)     if  $d > x.dato$  then
6)        $x.dx \leftarrow \text{Insert}(x.dx, d)$ 
7)     else if  $d < x.dato$  then
8)        $x.sx \leftarrow \text{Insert}(x.sx, d)$ 
9)   return  $x$ 

```

BuildNode(d) Crea un nuovo nodo con dato d

Va a sinistra o a destra a seconda se il dato è maggiore o minore del nodo alla ricorsione corrente, fino ad arrivare ad un nodo NULL dove verrà inserito il nuovo nodo.



Nell'esempio sopra, aggiungo 3 nell'albero colorato in nero.

Se il nodo è *null*, lo inserisco, altrimenti controllo se ciò che voglio inserire è maggiore o minore del dato corrente. Scorro ricorsivamente a destra se il dato è maggiore, altrimenti a sinistra; finché non trovo un nodo *null* dove inserirò il dato.

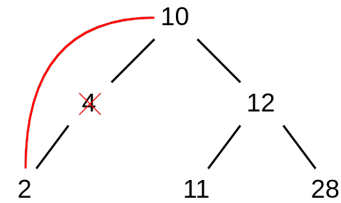
L'inserimento fatto in questo modo potrebbe sbilanciare l'albero.

Esistono metodi per inserire e mantenere l'albero bilanciato; è più costoso ma ne giovano tutti gli altri algoritmi.

5.4.3 Cancellazione

Se il nodo da eliminare **non ha figli**, basta eliminare il nodo.

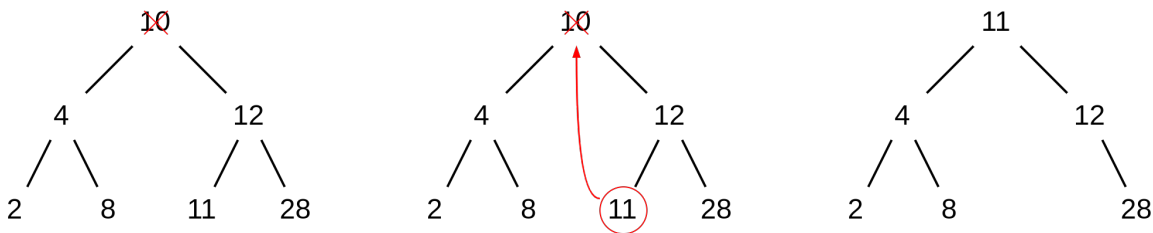
Se il nodo x da eliminare **ha un solo figlio**, basta collegare il figlio di x al padre di x



Se il nodo da eliminare **ha entrambi i figli**, ci sono due possibili modi per eliminare il nodo:

- Sposto il **MAX del ramo sinistro**
Il MAX del ramo sinistro può avere al massimo un figlio sinistro (che è più piccolo).
Se avesse anche il figlio destro, non sarebbe il massimo
- Sposto il **MIN del ramo destro**
Il MIN del ramo destro può avere al massimo un figlio destro (che è più grande).
Se avesse anche il figlio sinistro, non sarebbe il minimo

In questo esempio è stato spostato il MIN del ramo destro, dovendo eliminare il 10



Prima definire il metodo $Delete(x, d)$ (il metodo per cancellare il nodo con dato d),
bisogna trovare il minimo (o il massimo)

Minimo Ricorsivo

```

1) Min( $x$ )
2)   if  $x.sx \neq \perp$  then
3)     return  $x$ 
4)   else
5)     return Min( $x.sx$ )

```

Minimo Iterativo

```

1) Min( $x$ )
2)   while  $x.sx \neq \perp$  do
3)      $x \leftarrow x.sx$ 
4)   return  $x$ 

```

Massimo Ricorsivo

```

1) Max( $x$ )
2)   if  $x.dx \neq \perp$  then
3)     return  $x$ 
4)   else
5)     return Max( $x.dx$ )

```

Massimo Iterativo

```

1) Max( $x$ )
2)   while  $x.dx \neq \perp$  do
3)      $x \leftarrow x.dx$ 
4)   return  $x$ 

```

Get&DeleteMin

Come prima, lavoriamo con il MIN del ramo destro e definiamo la funzione che restituisce il minimo ed elimina il nodo

```

1) Get&DeleteMin( $x$ )
2)   if  $x.sx = \perp$  then
3)      $d \leftarrow x.dato$ 
4)      $r \leftarrow x.dx$ 
5)     Deallocate( $x$ )
6)     return ( $d, r$ )
7)   else
8)     ( $d, r$ )  $\leftarrow$  Get&DeleteMin( $x.sx$ )
9)      $x.sx \leftarrow r$ 
10)  return ( $d, r$ )

```

Le istruzioni 4, 5, 6 definiscono una funzione chiamata **SkipRight**.

Il padre "skipa" al figlio destro rispetto a.

Sostituisce suo figlio sinistro, con il figlio destro del sinistro.

Questo succede solo al nodo che contiene il minimo.

Quando la ricorsione risale, non fa altro che ricopiare i figli **sinistri** nei figli sinistri.

È un lavoro inutile ma sarà fondamentale per l'ultima chiamata prima della risalita, la quale sposterà il figlio **destro** del nodo che contiene il minimo.

Il dato d sarà lo stesso per tutta la risalita e sarà quello che verrà sostituito al nodo da eliminare.

```

1) SkipRight( $x$ )
2)    $tmp \leftarrow x.dx$ 
3)   Deallocate( $x$ )
4)   return  $tmp$ 

```

- Salvo il figlio destro in tmp
- Dealloco x
- restituisco tmp

```

1) SkipLeft( $x$ )
2)    $tmp \leftarrow x.sx$ 
3)   Deallocate( $x$ )
4)   return  $tmp$ 

```

- Salvo il figlio sinistro in tmp
- Dealloco x
- restituisco tmp

Get&DeleteMin (Versione con il padre p passato per riferimento)

```

1) Get&DeleteMin( $x, p$ )
2)   if  $x.sx = \perp$  then
3)      $d \leftarrow x.dato$ 
4)     SwapChild( $p, x, x.dx$ )
5)     return  $d$ 
6)   else
7)     return Get&DeleteMin( $x.sx, x$ )

```

```

1) SwapChild( $p, x, y$ )
2)   if  $p.sx = x$  then
3)      $p.sx \leftarrow y$ 
4)   else
5)      $p.dx \leftarrow y$ 
6)   Deallocate( $x$ )

```

Rispetto alla versione precedente, non c'è bisogno di restituire il padre ma lo si passa per riferimento.

SwapChild scambia il figlio x (di p) con un certo nodo y .

L'*if* controlla se x è il figlio sinistro o destro di p

Avviene lo scambio con y e poi dealloca x

Delete

Questa è la funzione che verrà chiamata per eliminare il nodo contenente il dato d .

- Cerca il nodo
- Se trovato, chiama la funzione DeleteNode
 - Se il nodo ha un solo figlio, verrà fatto lo *SkipRight* se non ha il sinistro, *SkipLeft* se non ha il destro.
 - Altrimenti posso eliminarlo in due modi possibili:
 - 1) cerco il MAX del ramo sinistro
 - 2) cerco il MIN del ramo destro
 - In questo caso si procede con il MIN
- Sia *Skip* che *Get&Delete* restituiscono un nodo a *DeleteNode*
- La funzione *DeleteNode* restituirà un nodo che sarà restituito alla chiamata ricorsiva di *Delete*. Questo verrà inserito al posto del nodo da eliminare

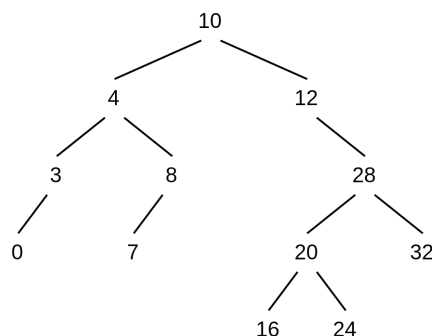

```
1) Delete( $x, d$ )
2)   if  $x = \perp$  then // Dato non trovato
3)   |   return  $\perp$ 
4)   |
5)   |   // Cerco il nodo che contiene il dato  $d$ 
6)   |   if  $d > x.dato$  then
7)   |   |    $x.dx \leftarrow \text{Delete}(x.dx, d)$ 
8)   |   |   else if  $d < x.dato$  then
9)   |   |   |    $x.sx \leftarrow \text{Delete}(x.sx, d)$ 
10)  |   |   else // Nodo trovato
11)  |   |   |   return DeleteNode( $x$ )
11)  |   return  $x$ 
```

```
1) DeleteNode( $x$ )
2)   if  $x.sx = \perp$  then
3)   |   return SkipRight( $x$ )
4)   |   else if  $x.dx = \perp$  then
5)   |   |   return SkipLeft( $x$ )
6)   |   else
7)   |   |    $x.dato \leftarrow \text{Get\&DeleteMin}(x.dx, x)$ 
8)   |   |   return  $x$ 
```

5.4.4 Ricerca del successore

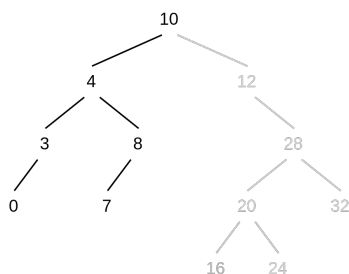
In un BST, il successore un qualsiasi numero (anche non presente nell'albero) è il **minimo dei maggioranti**.

Partendo dalla radice, non so se esiste il successore, tantomeno non so se esiste un successore migliore; quindi man mano che scendo l'albero, salvo e aggiorno il successore con quello migliore trovato.

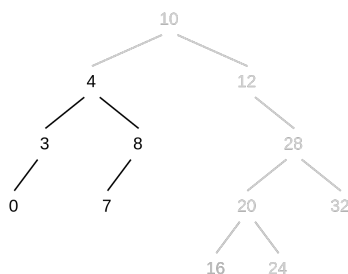


d	Successor(d)
3	3
5	7
13	16
8	10
12	16
32	\perp
-1	0

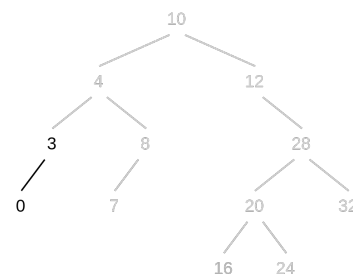
Esempio: supponiamo di voler trovare il successore di 1



Posso andare a sinistra, quindi il successore sicuramente esiste; per il momento è 10



Posso ancora andare a sinistra, quindi il successore migliora in 4



Posso ancora andare a sinistra, quindi il successore migliora in 3
Dopodiché non posso più scendere.

Quindi l'idea è quella di aggiornare la stima con x quando il dato d è minore di x , ovvero quando scendo a sinistra.

Vorrà dire che x è un maggiorante di d , ma non sappiamo se è il minimo dei maggioranti.

Se scendo a destra, non aggiorno la stima poiché x è minore, non è un maggiorante, del dato d

```

1) Successor( $x, d$ )
2)   if  $x = \perp$  then
3)     return  $\perp$ 
4)   else
5)     if  $d \geq x.dato$  then
6)       return Successor( $x.dx, d$ )
7)     else
8)        $s \leftarrow$  Successor( $x.sx, d$ )
9)       if  $s = \perp$  then
10)        return  $x$ 
11)      else
12)        return  $s$ 
```

```

1) Successor( $x, d, s$ )
2)   if  $x = \perp$  then
3)     return  $s$ 
4)   else
5)     if  $d \geq x.dato$  then
6)       return Successor( $x.dx, d, s$ )
7)     else
8)       return Successor( $x.sx, d, x$ )
```

Versione **ricorsiva in coda**.

Viene passata la stima s come parametro la quale verrà aggiornata solo se si scende a sinistra.

Nella prima chiamata sarà \perp

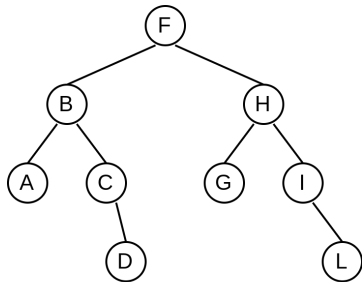
5.5 Albero perfettamente bilanciato

APB: Albero Perfettamente Bilanciato oppure PBT: Perfect Balanced Tree

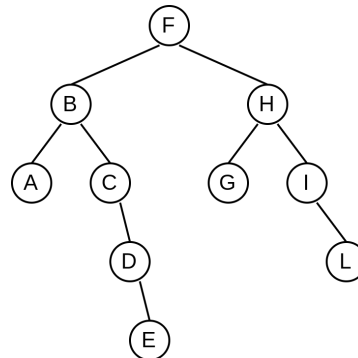
Un albero è perfettamente bilanciato quando il numero dei nodi del sottoalbero sinistro differisce al più di 1 dal numero dei nodi del sottoalbero destro.

Un albero T è perfettamente bilanciato se $\forall x \in T, \left| |T_{x.sx}| - |T_{x.dsx}| \right| \leq 1$

Un PBT è l'albero più basso possibile, con altezza al più $\lfloor \log_2(n) \rfloor$



Esempio di albero perfettamente bilanciato.



Esempio di albero NON perfettamente bilanciato.
L'albero con radice B ha il sottoalbero sinistro con 1 nodo e il destro con 3, con una differenza quindi di 2

Con l'aggiunta di un nodo bisogna riposizionare i nodi affinché l'albero rimanga perfettamente bilanciato.

In generale conviene usare i PBT se gli algoritmi che utilizzeremo lo scorrono in altezza $\log(n)$; come detto prima, un PBT è l'albero più basso possibile.

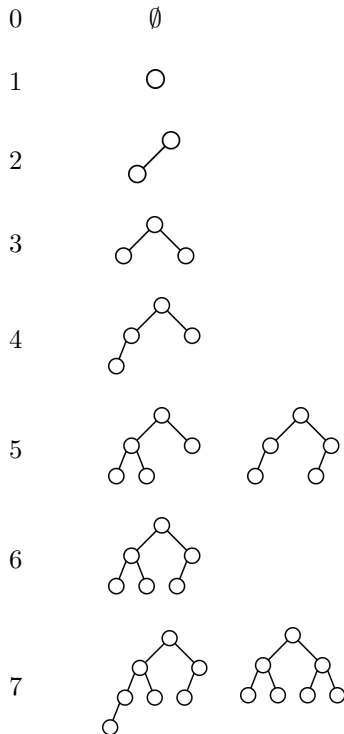
Esistono diversi tipi di PBT, uno di questi sono gli **AVL**

5.6 Alberi AVL

Proprietà

- È un Binary Search Tree (quindi ne eredita le proprietà)
- $\forall x \in T, |h(T_{x.sx}) - h(T_{x.dx})| \leq 1$ dove T è l'albero
Ossia, l'altezza dei sottoalberi, differiscono al più di 1

N. di nodi Albero



5.6.1 Alberi AVL minimi

Un **AVL minimo** di altezza h , è l'AVL di altezza h con il minimo numero di nodi






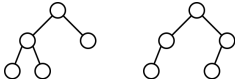

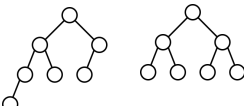
La classe degli AVL minimi, sono i peggiori riguardo l'altezza.

Questo perché gli algoritmi che scorrono l'altezza non dipendono dal numero di nodi.

Di conseguenza, se all'aumentare dei nodi non aumenta l'altezza, allora l'albero è migliore rispetto ad avere meno nodi con la stessa altezza (aumento la quantità dei dati mantenendo la stessa efficienza).

In particolare, se ad una certa altezza ho il minimo numero di nodi, allora ho l'albero peggiore per quell'altezza.

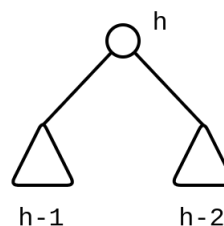
Si osserva dunque, che se si dimostra che per gli AVL minimi $h = \Theta(\log(n))$ allora lo è anche per quelli non minimi (ovvero per tutti gli AVL).

N. di nodi	Albero	AVL minimo
0		✓
1		✓
2		✓
3		✗
4		✓
5		✗
6		✗
7		✓ (sinistra)

Si nota come i sottoalberi di un AVL minimo, siano i due AVL minimi precedenti.

Ad esempio l'AVL minimo con 7 nodi, ha il sottoalbero sinistro che è l'AVL minimo precedente, ovvero quello con 4 nodi; e il sottoalbero destro è quello ancora precedente, ovvero quello con 2 nodi.

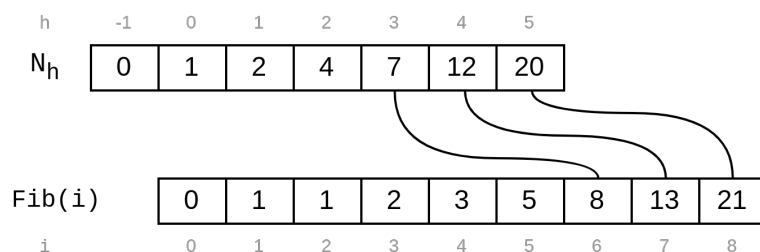
Sembra ricordare la sequenza di Fibonacci, di conseguenza possiamo dire che la struttura sia di questo tipo:



$$\begin{cases} N_h &= 1 + N_{h-1} + N_{h-2} \\ N_{-1} &= 0 \\ N_0 &= 1 \end{cases} \quad N_h : \text{Numero di nodi di un albero alto } h$$

Si può osservare che

$$N_h = Fib(h + 3) - 1$$



Dimostrazione (induzione)

Si vuole dimostrare che $N_h = Fib(h + 3) - 1$

Caso base

$$h = -1 \quad N_h = N_{-1} = 0 = Fib(h+3) - 1 = Fib(2) - 1 = 1 - 1 = 0$$

$$h = 0 \quad N_h = N_0 = 1 = Fib(h + 3) - 1 = Fib(3) - 1 = 2 - 1 = 1$$

Caso Induttivo $h \geq 1$:

$$N_h = 1 + N_{h-1} + N_{h-2}$$

$$\text{Ipotesi induttiva: } \begin{cases} N_{h-1} = Fib((h-1)+3) - 1 = Fib(h+2) - 1 \\ N_{h-2} = Fib((h-2)+3) - 1 = Fib(h+1) - 1 \end{cases}$$

$$N_h = 1 + [Fib(h+2) - 1] + [Fib(h+1) - 1]$$

$$N_h = \underbrace{[Fib(h+2) + Fib(h+1)]}_{Fib(h+3)} - 1$$

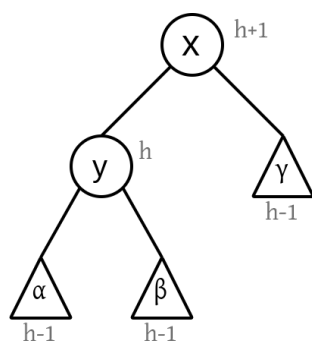
$$N_h = Fib(h + 3) - 1$$

$Fib(h+2) + Fib(h+1) = Fib(h+3)$ perché, per definizione, la sequenza di fibonacci è la somma dei due valori precedenti

5.6.2 Inserimento in un AVL

L'inserimento in di un nodo, può rompere la struttura di un AVL.

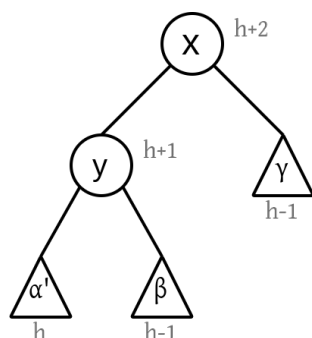
L'algoritmo di inserimento quindi, dovrà anche risistemare l'albero affinché rimanga AVL.



Prendiamo ad esempio questo AVL

Si possono verificare **due casi** che rovinano la struttura di AVL:

Caso 1: Inserisco il nodo nel sottoalbero α

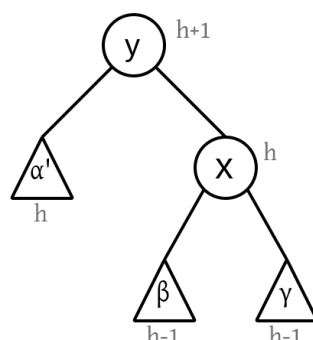
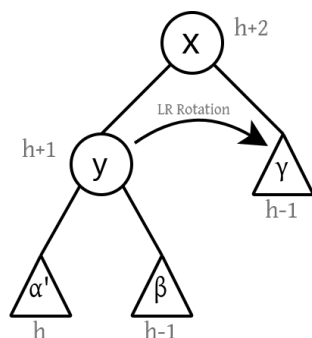


Inserendo il nodo nel sottoalbero α , questo aumenta di altezza (da $h-1$ a h)

Di conseguenza, anche y aumenta di 1, lo stesso vale per x

A questo punto i sottoalberi di x non rispettano le condizioni per essere un AVL: c'è una differenza di 2 tra il sottoalbero destro e sinistro di x

Spostando in un certo modo gli alberi α', β, γ è possibile mantenere la struttura di AVL.



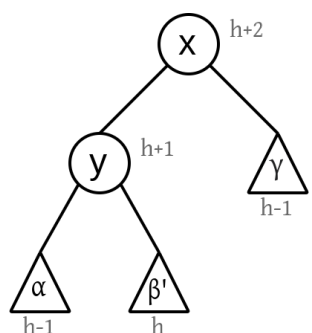
Facendo una rotazione da sinistra a destra centrata sul nodo x (ovvero il nodo dove c'è il problema), riesco a mantenere la struttura di AVL.

Inoltre, continua a mantenere la struttura di BST.

β continua ad essere più grande di y e più piccolo di x

α' e γ non vengono spostati quindi mantengono la proprietà.

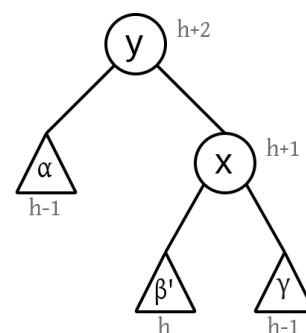
Caso 2: Inserisco il nodo nel sottoalbero β

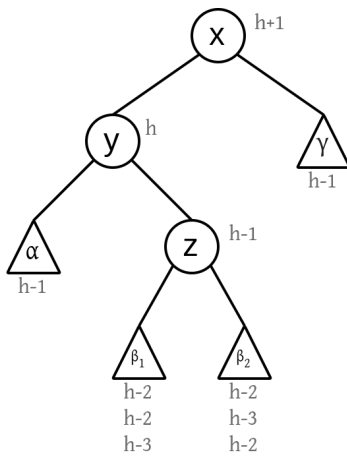


Inserendo il nodo nel sottoalbero β , come nel caso precedente, aumentano le altezze e l'albero si sbilancia, perdendo le proprietà di AVL.

Una rotazione a destra come nel caso precedente, sposta il problema sull'altro lato dell'albero, ma non lo risolve.

(nella figura a destra è stata effettuata una LRRotation(x))





Per risolvere il problema, si divide il sottoalbero β

Si estrae la radice z (di β) che avrà altezza $h - 1$

Le altezze dei due sottoalberi figli (β_1, β_2) possono essere:

$$h - 2 \text{ e } h - 2, \quad h - 2 \text{ e } h - 3, \quad h - 3 \text{ e } h - 2$$

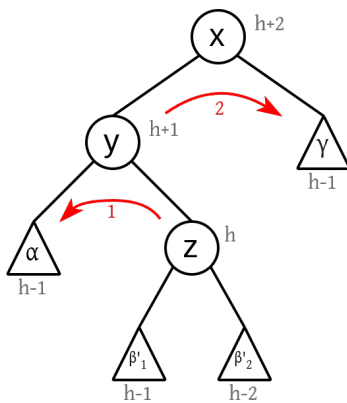
Almeno uno dei due sottoalberi sarà alto $h - 2$;

ad esempio, se fossero entrambi alti $h - 3$ vuol dire che $h(z) = h - 2$, il che non è vero dato che $h(z) = h - 1$

Inoltre, non possono essere più bassi di $h - 3$

ad esempio, se uno dei due fosse alto $h - 4$,

e l'altro per forza sarà alto $h - 2$, vuol dire che non è un AVL



Supponiamo di aggiungere un nodo in β_1 stesso ragionamento per β_2

Inserendo in β_1 , avrò il caso in cui sia β_1 sia β_2 sono alti $h - 2$

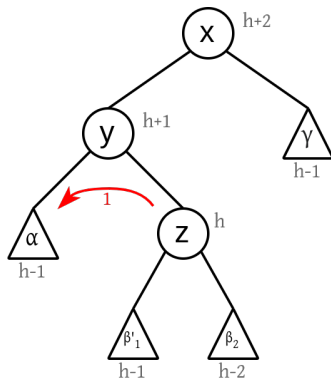
altrimenti il problema si troverà in un altro punto, che andrà risolto comunque con uno dei due casi.

In questo caso servono due rotazioni:

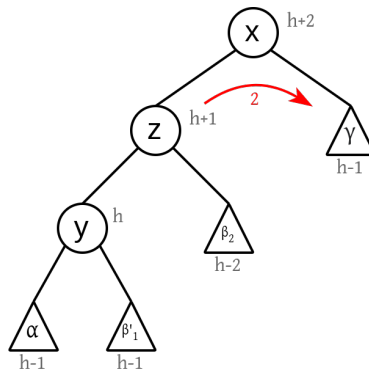
una verso sinistra su y e successivamente una verso destra su x

È **importante l'ordine** con cui si effettuano le rotazioni, altrimenti non funziona.

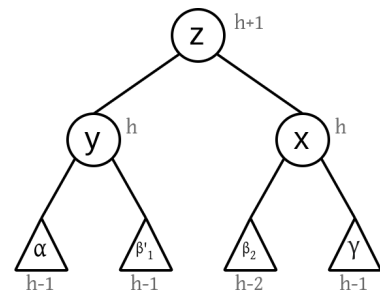
Rotazioni:



effettuo una RLRotation(y)



effettuo una LRRotation(x)



Risultato finale, proprietà AVL mantenute

Così facendo si è preservata la proprietà sia di AVL che di BST:

- β_1' continua ad essere $y < \beta_1' < z$
- β_2 continua ad essere $z < \beta_2 < x$
- z continua ad essere $y < \beta_2 < x$

Costo

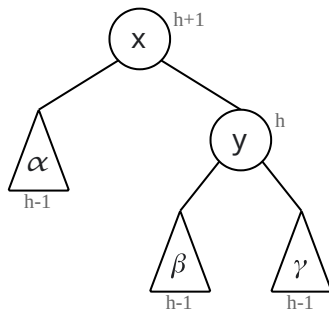
Riparare, e quindi mantenere, la struttura di AVL dopo un inserimento, ha tempo costante dato che vengono effettuati degli scambi fra puntatori.

Serve riparare l'albero ad ogni inserimento che rompe l'albero.

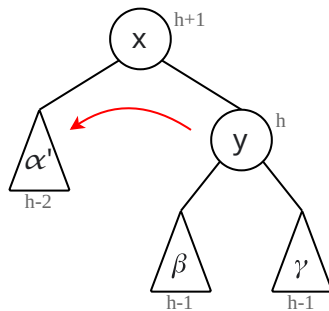
5.6.3 Cancellazione in un AVL

La cancellazione porta gli stessi problemi dell'inserimento e si risolvono allo stesso modo.

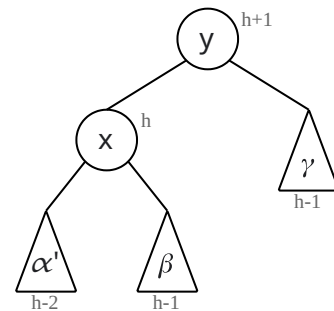
Singola rotazione



AVL iniziale, cancello in α

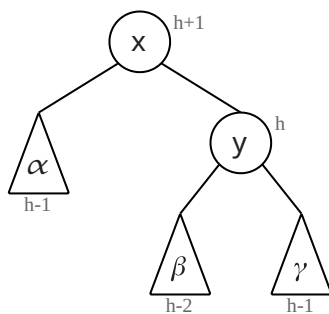


La radice x ha perso le prop. di AVL
Effettuo una RLRotation(x)

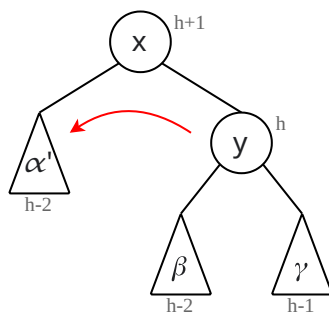


Risultato finale, nodo cancellato e
proprietà AVL mantenute

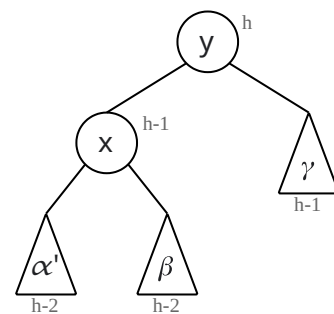
Singola rotazione, caso in cui si abbassa l'albero NB: $h(\beta) = h - 2$ e $h(y) = h$



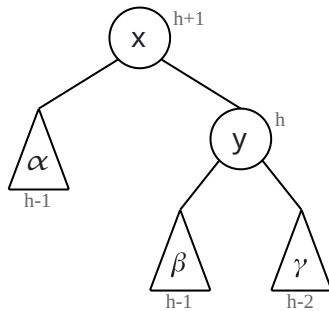
AVL iniziale, cancello in α



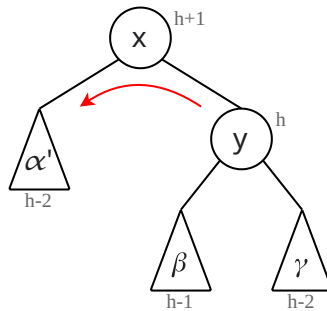
La radice x ha perso le prop. di AVL
Effettuo una RLRotation(x)



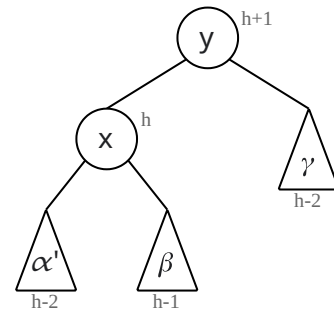
Risultato finale, nodo cancellato e
proprietà AVL mantenute

Doppia rotazione

AVL iniziale, cancello in α

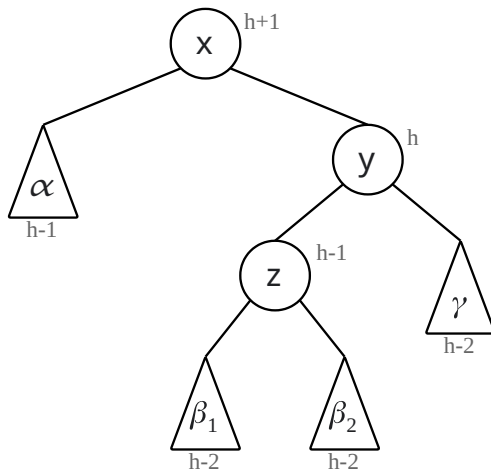


La radice x ha perso le prop. di AVL
Effettuo una RLRotation(x)

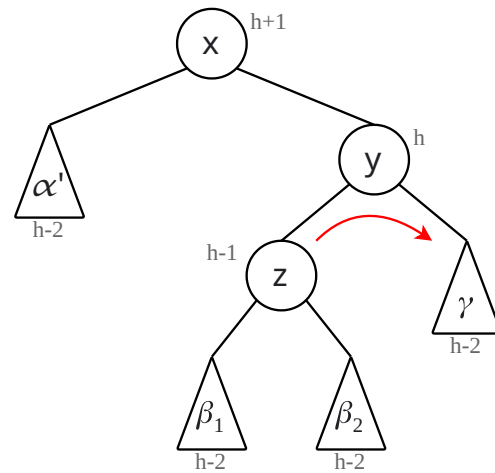


Il problema rimane

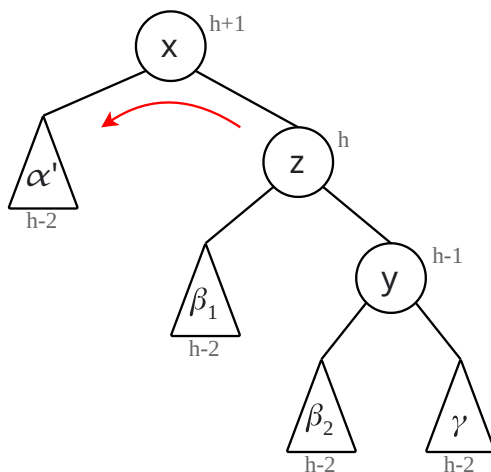
Bisogna effettuare ulteriori passaggi:



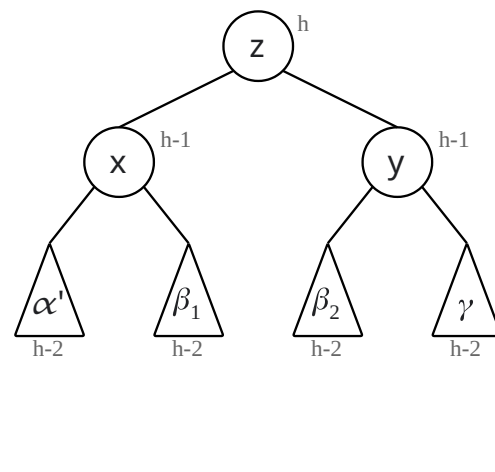
Parto dall'AVL iniziale e divido β
Canello poi in α



Effettuo una prima rotazione LRRotation(y)



poi una seconda rotazione RLRotation(x)



Il problema è stato risolto

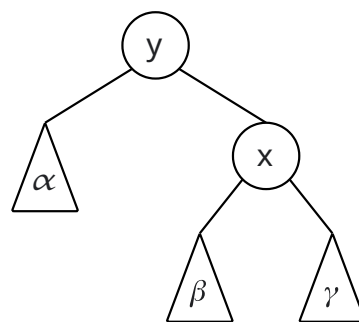
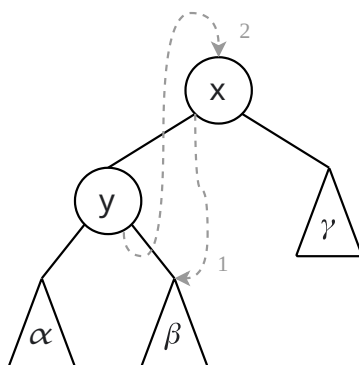
5.6.4 Algoritmi

1) **L-R-Rotation**(x)

```

2)    $y \leftarrow x.sx$ 
3)    $x.sx \leftarrow y.dx$  // 1
4)    $y.dx \leftarrow x$    // 2
5)   return  $y$ 

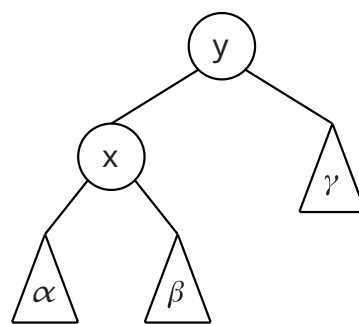
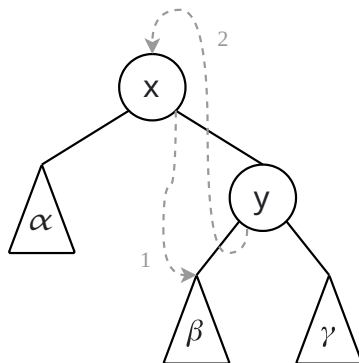
```

1) **R-L-Rotation**(x)

```

2)    $y \leftarrow x.dx$ 
3)    $x.dx \leftarrow y.sx$  // 1
4)    $y.sx \leftarrow x$    // 2
5)   return  $y$ 

```



AVL Node

D	ht
SX	DX

BST Node

D	
SX	DX

I nodi di un AVL, a differenza di quelli di un BST, possiede un ulteriore attributo per memorizzare l'altezza.

In questo modo è possibile sapere se ci sono violazioni.

Restituisce l'altezza dell'AVL con radice in x , se questa è \perp allora restituisce -1

1) **HgtAVL**(x)

```

2)   return ( $x = \perp$  ?  $-1$  :  $x.ht$ )

```

Essendo un semplice *if* la funzione ha tempo costante $\Theta(1)$

Aggiorna l'altezza dell'AVL con radice in x (utilizzata dopo una rotazione)

1) **UpdateHgtAVL**(x)

```

2)    $h_L \leftarrow \text{HgtAVL}(x.sx)$  // Prendo l'altezza del figlio sinistro
3)    $h_R \leftarrow \text{HgtAVL}(x.dx)$  // Prendo l'altezza del figlio destro
4)    $x.ht \leftarrow \max\{h_L, h_R\}$ 

```

Aggiorno l'altezza dell'albero radicato in x con il massimo fra le altezze dei due figli

Tutte e tre le istruzioni sono chiamate a funzioni a tempo costante, quindi l'intera funzione è costante $\Theta(1)$

1) **L-R-AVLRotation**(x)

```

2)    $y \leftarrow \text{L-R-Rotation}(x)$ 
3)    $\text{UpdateHgtAVL}(x)$ 
4)    $\text{UpdateHgtAVL}(y)$ 
5)   return  $y$ 

```

La rotazione su un AVL ha bisogno successivamente di aggiustare le altezze. Questa funzione racchiude le tre funzioni per fare ciò.

Dopo la rotazione, x si trova "più in basso" di y , logicamente è importante che venga aggiornata prima la sua di altezza e poi quella di y

Essendo tutte le funzioni chiamate $\Theta(1)$, il costo totale sarà costante $\Theta(1)$

1) **L-D-AVLRotation**(x)

```

2)    $x.sx \leftarrow \text{R-L-AVLRotation}(x.sx)$ 
3)   return  $\text{L-R-AVLRotation}(x)$ 

```

Funzione che effettua la doppia rotazione al figlio sinistro (**Left-Double**) Farà quindi prima una R-L Rotation e poi una L-R rotation. Speculare sarà per R-D-AVLRotation

Anche questa è costante $\Theta(1)$

```

1) LBalanceAVL( $x$ )
2)   if  $\text{HgtAVL}(x.sx) - \text{HgtAVL}(x.dx) = 2$  then
3)     if  $\text{HgtAVL}(x.sx.dx) < \text{HgtAVL}(x.sx.sx)$  then
4)        $x \leftarrow \text{L-R-AVLRotation}(x)$ 
5)     else
6)        $x \leftarrow \text{L-D-AVLRotation}(x)$ 
7)   else
8)      $\text{UpdateHgtAVL}(x)$ 
9)   return  $x$ 

```

Bilancia l'albero radicato in x (input) nel caso in $h(x.sx) = h(x.dx) + 2$, ovvero l'altezza del sottoalbero sinistro è più 2 del destro.

Speculare l'algoritmo **RBalanceAVL** nel caso in cui sia il sottoalbero destro ad essere più alto di 2.

Ad esempio, è importante chiamare questa funzione dopo un inserimento o una cancellazione.

Ha tempo costante, in quanto tutte le funzioni che chiama sono a tempo costante

Inserimento e cancellazione

```

1) AVLInsert( $x, d$ )
2)   if  $x = \perp$  then
3)     return  $\text{BuildNodeAVL}(d)$ 
4)   else
5)     if  $d < x.dato$  then
6)        $x.sx \leftarrow \text{AVLInsert}(x.sx, d)$ 
7)        $x \leftarrow \text{LBalanceAVL}(x)$ 
8)     else if  $d > x.dato$  then
9)        $x.dx \leftarrow \text{AVLInsert}(x.dx, d)$ 
10)       $x \leftarrow \text{RBalanceAVL}(x)$ 
11)   return  $x$ 

```

Simile all'inserimento in un BST ma con la differenza che bisogna bilanciare l'albero.

Se inserisco a sinistra, rendo potenzialmente il sottoalbero sinistro più profondo del destro, effettuo quindi un **LeftBalanceAVL**.

Discorso speculare se inserisco destra.

```

1) AVLDelete( $x, d$ )
2)   if  $x \neq \perp$  then
3)     if  $d < x.dato$  then
4)        $x.sx \leftarrow \text{AVLDelete}(x.sx, d)$ 
5)        $x \leftarrow \text{RBalanceAVL}(x)$ 
6)     else if  $d > x.dato$  then
7)        $x.dx \leftarrow \text{AVLDelete}(x.dx, d)$ 
8)        $x \leftarrow \text{LBalanceAVL}(x)$ 
9)     else
10)       $x \leftarrow \text{DeleteNodeAVL}(x)$ 
11)   return  $x$ 

```

Simile alla cancellazione in un BST ma con la differenza che bisogna bilanciare l'albero.

Se cancello a destra, rendo potenzialmente il sottoalbero destro più basso (o meno alto) e di conseguenza rendo il sinistro più profondo, effettuo quindi un **LeftBalanceAVL**.

Discorso speculare se cancello sinistra.

```

1) DeleteNodeAVL( $x$ )
2)   if  $x.sx = \perp$  then
3)      $x \leftarrow \text{SkipRight}(x)$ 
4)   else if  $x.dx = \perp$  then
5)      $x \leftarrow \text{SkipLeft}(x)$ 
6)   else
7)      $x.dato \leftarrow \text{Get\&DeleteMinAVL}(x.dx, x)$ 
8)      $x \leftarrow \text{LBalanceAVL}(x)$ 
9)   return  $x$ 

```

```

1) Get\&DeleteMinAVL( $x, p$ )
2)   if  $x.sx = \perp$  then
3)      $d \leftarrow x.dato$ 
4)      $y \leftarrow \text{SkipRight}(x)$ 
5)   else
6)      $d \leftarrow \text{Get\&DeleteMinAVL}(x.sx, x)$ 
7)      $y \leftarrow \text{RBalanceAVL}(x)$ 
8)    $\text{SwapChild}(p, x, y)$ 
9)   return  $d$ 

```

5.7 Alberi Red Black

Un albero binario di ricerca di altezza h , è in grado di eseguire operazioni elementari sugli insiemi (insert, delete, max, min, search, ...) nel tempo $O(h)$.

Queste operazioni sono veloci se l'albero è basso, ma se l'altezza è grande, le prestazioni ne risentono (si potrebbe arrivare anche ad un costo lineare).

Gli alberi **red-black** rappresentano uno dei modi in cui gli alberi di ricerca vengono bilanciati, in modo tale da garantire alle operazioni elementari di essere eseguite in tempo $O(\log n)$

Negli alberi red-black, i nodi contengono anche il loro *colore*, oltre che il puntatore al figlio *destra* e *sinistra*, e le variabili per i *dati*.

Proprietà

- È un BST
- I dati sono presenti solo nei nodi interni
 - Le foglie non contengono dati (contengono il valore NULL), sono quindi tutte identiche.
 - I genitori delle foglie punteranno tutti ad un unico nodo NULL così da evitare spreco di memoria.
- **Vincoli:**
 1. Tutti i nodi sono o **rossi** o **neri**
 2. Tutte le foglie sono **neri**
 3. Se un nodo è **rosso**, allora entrambi i figli sono **neri**
 - Ossia i nodi **rossi** non possono avere figli **rossi**
 - $\forall x \in T, \exists h \in \mathbb{N} : \forall \pi \in Path(x) \quad |\pi| = h$
- Ogni cammino dalla radice a una foglia deve contenere lo stesso numero di nodi neri

5.7.1 Altezza nera

Definiamo **altezza nera** di un nodo x , indicata con $bh(x)$ (black height), il numero di nodi **neri** lungo un cammino semplice che inizia dal nodo x e finisce a una foglia.

Il nodo x è escluso dato che non influisce sull'altezza nera, che sia **rosso** o **nero** l'altezza nera rimane la stessa.

L'altezza nera è pari alla metà dell'altezza $bh(x) = \lceil h(x)/2 \rceil$

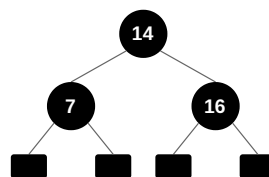
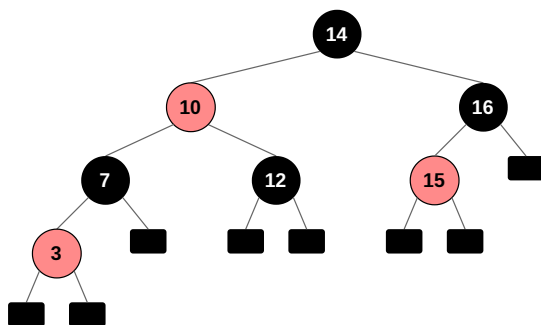
Quindi se l'albero ha almeno un percorso la cui lunghezza è inferiore all'altezza nera, quell'albero così com'è non può essere un red-black (con opportune rotazioni lo può diventare).

Se non c'è neanche un percorso con lunghezza inferiore all'altezza nera, è possibile colorare l'albero in un certo modo per renderlo red-black.

Il numero di nodi interni è: (Internal Nodes) $IN(x) \geq 2^{bh(x)} - 1$ (vedi osservazione sotto)

Il numero di nodi in un albero binario è $2^{h+1} - 1$, escludendo le foglie $2^h - 1$

Esempio di albero red black:



Si osserva che eliminando i nodi rossi si ottiene un albero pieno.

È solo una rappresentazione grafica per osservare la cosa, non si utilizza una cosa simile perché (come si può notare) si perdono dei dati.

5.7.2 Teorema

$$\#IN(x) \geq 2^{bh(x)} - 1 \quad \begin{array}{l} \#IN(x) : \text{ numero di nodi interni} \\ bh(x) : \text{ Black Height} \end{array}$$

Dimostrazione per induzione sull'altezza dell'albero $h(x)$

Caso base: $h(x) = 0$

Quando l'altezza è 0 vuol dire che non ci sono nodi interni (escluse le foglie).

L'insieme dei nodi avrà solamente una foglia $\{ \blacksquare \}$

$$\#IN(x) = 0 \geq 2^0 - 1 = 0 \quad \checkmark$$

Caso induttivo: $h(x) > 0$

Quindi x ha due figli α e β

$$h(x) = 1 + \max\{h(\alpha), h(\beta)\} \Rightarrow h(x) - 1 \geq h(\alpha), h(\beta)$$

$$\#IN(\alpha) \geq 2^{bh(\alpha)} - 1$$

$$\#IN(\beta) \geq 2^{bh(\beta)} - 1$$

Osservazioni:

1. $bh(\alpha) = bh(\beta)$ ossia hanno lo stesso numero di nodi **neri**, altrimenti x non sarebbe red-black
2. $bh(x) \leq 1 + bh(\alpha) = 1 + bh(\beta)$

$$\#IN(x) = \underset{\text{radice } x}{1} + \#IN(\alpha) + \#IN(\beta) = 1 + (2^{bh(\alpha)} - 1) + (2^{bh(\beta)} - 1) =$$

$$2 \cdot 2^{bh(\alpha)} - 1 = 2^{bh(\alpha)+1} - 1 \underset{\text{dalla seconda osservazione}}{\geq} 2^{bh(x)} - 1$$

$$\text{E quindi che } \#IN(x) \geq 2^{bh(x)} - 1$$

Osservazione:

Da questo teorema si può osservare, inoltre, che l'altezza di un albero red-black è logaritmica nel numero di nodi

$$\underbrace{\#IN(x)}_n \geq 2^{bh(x)} - 1 \Rightarrow n + 1 \geq 2^{bh(x)} \Rightarrow bh(x) \leq \log(n + 1)$$

$$\text{Come visto prima, } bh = \left\lceil \frac{h}{2} \right\rceil \quad \text{quindi } h \leq 2 bh(x) \leq \log(n + 1) \Rightarrow h(x) = O(\log n)$$

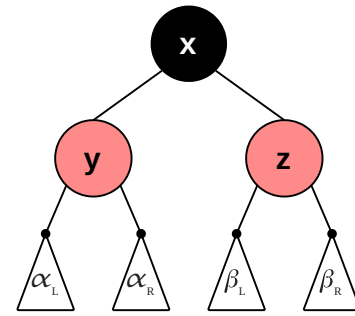
5.7.3 Inserimento in red-black

L'inserimento avviene come in un qualsiasi BST, successivamente si colora il nodo di **rosso** per evitare di aumentare l'altezza **nera** solo su quel percorso, il che avrebbe portato alla violazione di una proprietà.

Essendo colorato di **rosso**,

se il padre y è **rosso**, viola: "un nodo **rosso** ha entrambi i figli **neri**"

se il padre y è **nero**, allora non c'è violazione

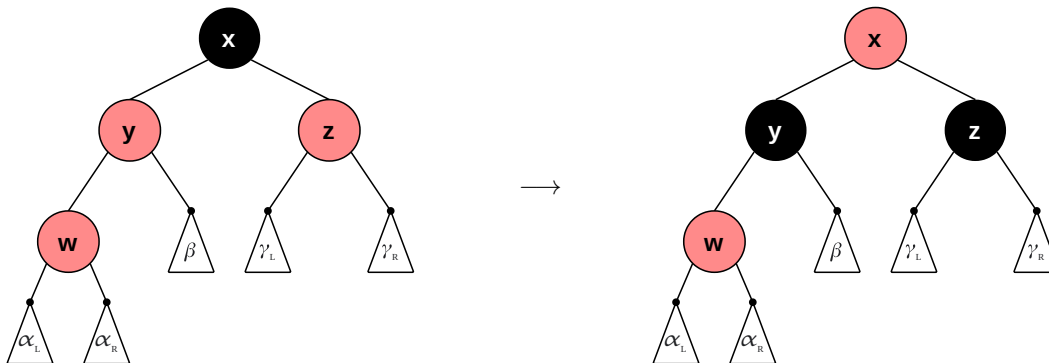


Albero red-black di partenza

Le violazioni si dividono in 3 casi:

1. lo zio z di w è rosso (i fratelli y e z sono entrambi rossi)

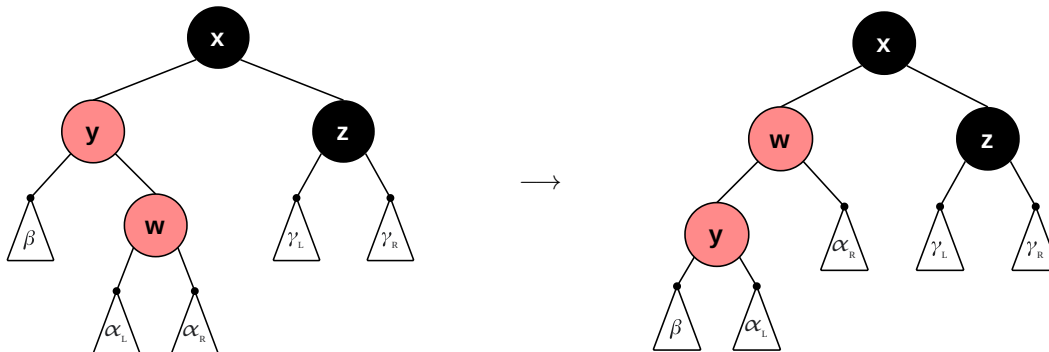
Si colorano y e z di **nero** e il loro padre x (nonno di w) si colora di **rosso**.



2. lo zio z è **nero** e y ha il figlio **rosso** a **destra**

Si effettua una *RLRotation*(y)

Non si risolve il problema, ma lo sposto a sinistra risolvendolo con il caso 3

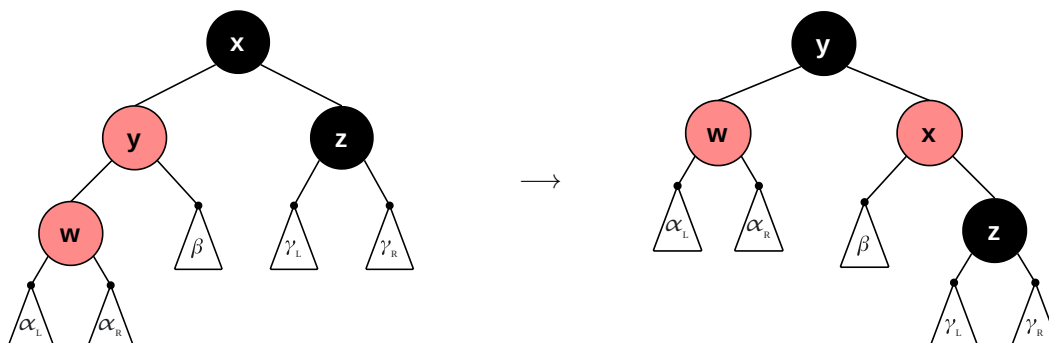


3. lo zio z è **nero** e y ha il figlio **rosso** a **sinistra**

Si colora il padre y di **nero** e il nonno x di **rosso**

Infine si effettua una *LRRotation*(x)

- Colorando y di **nero** si risolve il problema "padre **rosso** figlio **rosso**" ma si alza l'altezza **nera**
- quindi coloro il padre x di **rosso**; così facendo abbasso l'altezza nera dell'albero destro in z
- si effettua quindi una *LRRotation*(x)



5.7.4 Cancellazione in red-black

La cancellazione di un nodo **rosso** non crea problemi, perché questo avrà padre **nero** e figli **neri**, quindi al più si sono collegati due nodi **neri**.

Invece, la cancellazione di un nodo **nero** porta ad uno sbilanciamento dell'altezza nera.

Per risolvere **momentaneamente** il problema dello sbilanciamento dell'altezza nera, si propaga il colore **nero**. Ovvero, il figlio del nodo da eliminare (a seconda del caso *SkipRight* o *SkipLeft*):

- se è **rosso**, lo si colora di **nero**
- se è **nero**, diventa un **doppio nero**

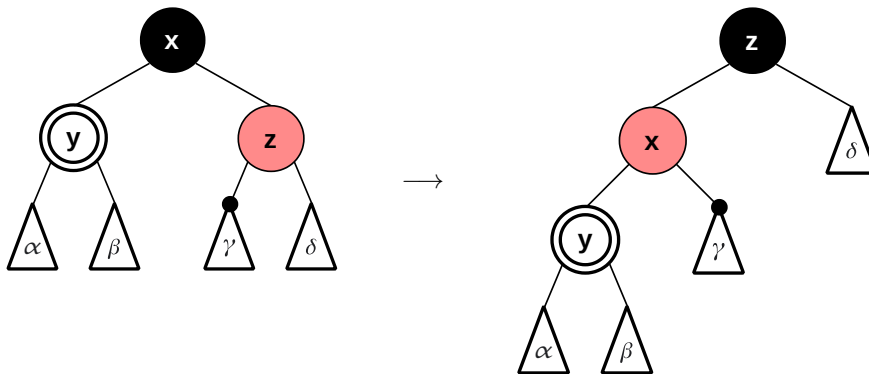
Così facendo, l'altezza rimane bilanciata.

Il colore **doppio nero** però va rimosso; sono 4 i casi possibili:

1. il fratello z , del **doppio nero** y , è **rosso**
2. il fratello z , del **doppio nero** y , è **nero** e z ha entrambi i figli **neri**
3. il fratello z , del **doppio nero** y , è **nero** e z ha il sinistro **rosso** e il destro **nero**
4. il fratello z , del **doppio nero** y , è **nero** e z ha il destro **rosso**

Caso 1

- colore x di **rosso**
- colore z di **nero**
- $RLRotation(x)$



Caso 2

- colore z di **rosso**
- colore y di **nero** (tolgo il doppio nero)
- se x è **rosso**, diventa **nero**
se x è **nero**, diventa **doppio nero**

Se z ha entrambi i figli **neri**, basterà colorarlo di **rosso** e togliere il **doppio nero** ad y .

Così facendo le altezze **neri** dei figli di x sono corrette.

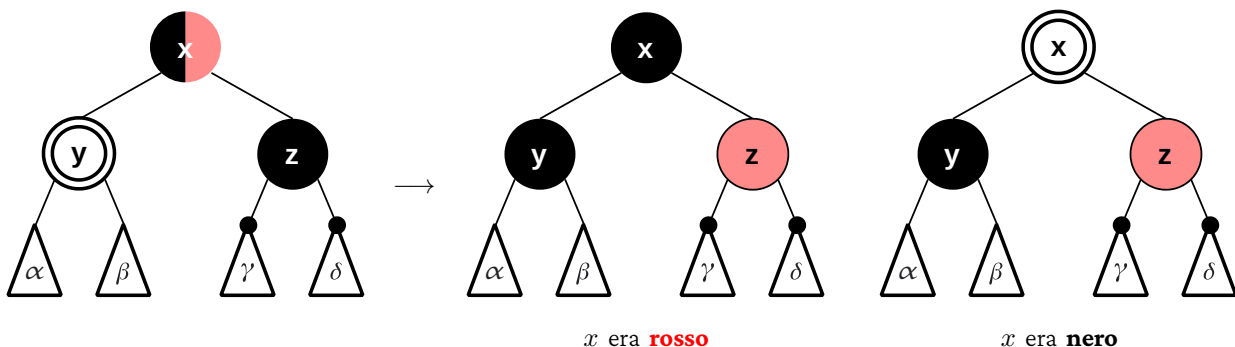
Il problema su y è risolto, al più viene spostato più in alto e passato ad x .

x potrebbe essere **nero** ma anche **rosso** (visto che entrambi i figli sono **neri**), **va quindi ricolorato**.

Mi basta aggiungere un livello di altezza **nera** ad x perché

- Avendo tolto il **doppio nero** a sinistra, l'altezza **nera a sinistra** si abbassa di 1
- Avendo colorato di **rosso** il figlio *destra* di x , l'altezza **nera a destra** si abbassa di 1

Se x dovesse essere ricolorato come **doppio nero**, il problema verrà risolto risalendo la ricorsione.



x era **rosso**

x era **nero**

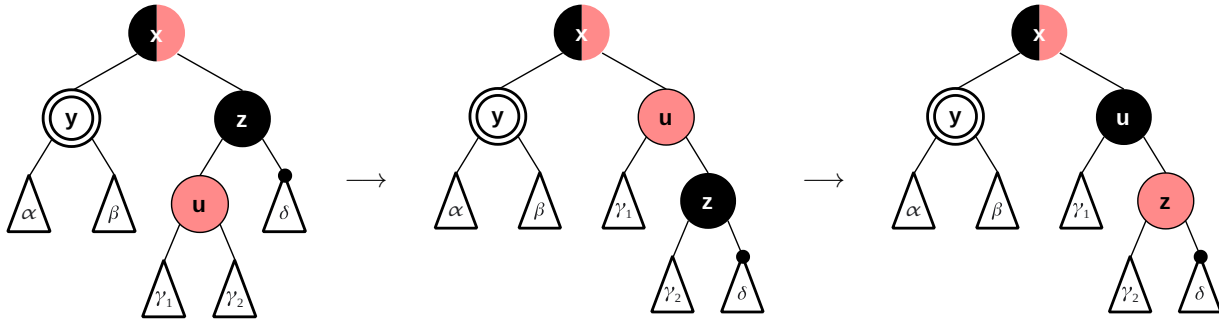
Caso 3

L'obiettivo è di spostare il figlio sinistro **rosso** (del fratello del nodo **doppio nero**), a destra per ricondurci al caso 4

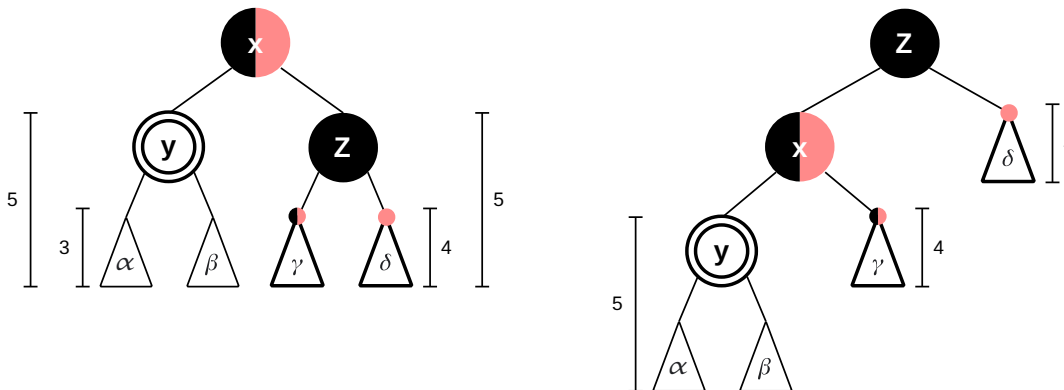
- LRRotation(z)

- Colore u di **nero** e z di **rosso**.

Questo sia per bilanciare l'altezza nera, sia per non avere il problema **rosso-rosso** con il padre x nel caso fosse **rosso**, ma anche perché è l'obiettivo (avere il **rosso** a destra).

**Caso 4**

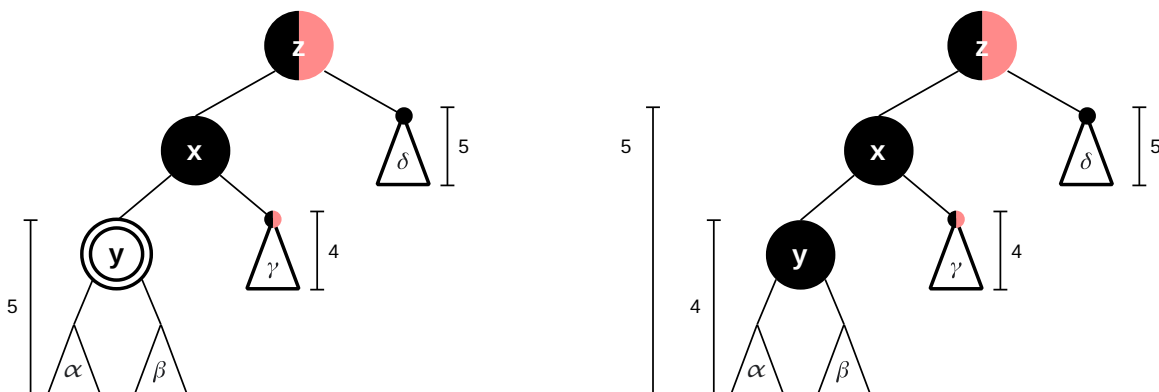
1. RLRotation(x)
2. Scambio i colori di x e di z
Ovvero colore x di **nero** e z di **rosso/nero**
3. Colore il figlio di destro di z di **nero**
altrimenti avrei l'altezza nera sbilanciata e un probabile **rosso-rosso**
4. Rimuovo il **doppio nero**
sul percorso sinistro di z c'è infatti un **nero** in più



Il figlio destro del fratello del **doppio nero**, è **rosso**

Albero dopo il passo 1

N.B.: I valori delle altezze nere sono solo un esempio per capire come variano passo dopo passo.



Albero dopo il passo 2 e 3

Albero dopo il passo 4

N.B.: L'albero in x abbia a sinistra un **nero** in più rispetto a destra.

N.B.: z (**rosso/nero**) non crea problemi nel caso in cui sia **rosso**, inizialmente x era **rosso/nero**. Se il problema non c'era prima, non ci sarà neanche dopo.

5.7.5 Algoritmi di inserimento e cancellazione

NULLRB è il puntatore al nodo foglia di un albero red-black, che è unico per evitare sprechi di memoria.

Inserimento

```

1) RBInsert( $x, d$ )
2)   if  $x = \text{NULLRB}$  then
3)     return BuildNodeRB( $d$ )
4)   else
5)     if  $d < x.dato$  then
6)        $x.sx \leftarrow \text{RBInsert}(x.sx, d)$ 
7)        $x \leftarrow \text{LInsBalanceRB}(x)$ 
8)     else if  $d > x.dato$  then
9)        $x.dx \leftarrow \text{AVLInsert}(x.dx, d)$ 
10)       $x \leftarrow \text{RInsBalanceRB}(x)$ 
11)   return  $x$ 

```

```

1) LInsBalanceRB( $x$ )
2)   if  $x.sx \neq \text{NULLRB}$  then
3)     switch LInsViolationRB( $x$ ) do
4)       case 1 :  $x \leftarrow \text{LInsBalanceRB1}(x)$ 
5)       case 2 :  $x \leftarrow \text{LInsBalanceRB2}(x)$ 
6)       case 3 :  $x \leftarrow \text{LInsBalanceRB3}(x)$ 
7)   return  $x$ 

```

Esiste anche il duale RInsBalanceRB

```

1) LInsViolationRB( $x$ )
2)    $(v, l, r) \leftarrow (0, x.sx, x.dx)$ 
3)   if  $l.cl = R$  then
4)     if  $r.cl = R$  then
5)       if  $l.sx.cl = R \vee l.dx.cl = R$  then
6)          $v \leftarrow 1$ 
7)     else
8)       if  $l.dx.cl = R$  then
9)          $v \leftarrow 2$ 
10)      else if  $l.sx.cl = R$  then
11)         $v \leftarrow 3$ 
12)   return  $v$ 

```

Cancellazione

```

1) DeleteRB(x)
2)   if x ≠ NULLRB then
3)     if d < x.dato then
4)       x.sx ← DeleteRB(x.sx, d)
5)       x ← LDelBalanceRB(x)
6)     else if d > x.dato then
7)       x.dx ← DeleteRB(x.dx, d)
8)       x ← RDelBalanceRB(x)
9)     else
10)      x ← DeleteNodeRB(x)
11)   return x

```

```

1) DeleteNodeRB(x)
2)   if d < x.dato then
3)     x ← SkipRightRB(x)
4)   else if d > x.dato then
5)     x ← SkipLeftRB(x)
6)   else
7)     x.dato ← Get&DeleteMinRB(x.dx, x)
8)     x ← RDelBalanceRB(x)
9)   return x

```

```

1) Get&DeleteMinRB(x, p)
2)   if x.sx ≠ NULLRB then
3)     d ← x.dato
4)     y ← SkipRightRB(x)
5)   else
6)     d ← Get&DeleteMinRB(x.sx, x)
7)     y ← LDelBalanceRB(x)
8)   SwapChild(p, x, y)
9)   return d

```

```

1) SkipRightRB(x)
2)   if x.cl = B then PropagateBlackRB(x.dx)
3)   return SkipRight(x)

```

```

1) PropagateBlackRB(x)
2)   x.cl ← (x.cl = R) ? B : BB

```

Se il colore di x è **rosso**, diventa **nero**
 Se il colore di x è **nero**, diventa **doppio nero**

```

1) LDelBalanceRB(x)
2)   if x.sx ≠ NULLRB then
3)     switch LDelViolationRB(x) do
4)       case 1 : x ← LDelBalanceRB1(x)
5)       case 2 : x ← LDelBalanceRB2(x)
6)       case 3 : x ← LDelBalanceRB3(x)
7)       case 4 : x ← LDelBalanceRB4(x)
8)   return x

```

```

1) LDelViolationRB(x)
2)   (v, l, r) ← (0, x.sx, x.dx)
3)   if l.cl = BB then
4)     if r.cl = R then
5)       v ← 1
6)     else
7)       if r.dx.cl = R then
8)         v ← 4
9)       else if r.sx.cl = R then
10)        v ← 3
11)      else
12)        v ← 2
13)   return v

```

6 Da ricorsivo a iterativo

Qualsiasi algoritmo ricorsivo può essere trasformato in iterativo e viceversa.

La ricorsione permette di risolvere problemi difficili in modo semplice rispetto all'iterazione, però è svantaggiosa in termini di memoria, dato che ogni chiamata alloca memoria per tenere traccia delle variabili locali, il valore di ritorno, e così via.

```

1) RecursiveMin( $x$ )
2)   if  $x.sx = \perp$  then
3)     return  $x.dato$ 
4)   else
5)     return RecursiveMin( $x.sx$ )

```

```

1) IterativeMin( $x$ )
2)   while  $x.sx \neq \perp$  do
3)      $x \leftarrow x.sx$ 
4)   return  $x.dato$ 

```

Questa funzione ricorsiva è **tail recursive** (ricorsiva in coda), cioè la chiamata ricorsiva è eseguita come ultima istruzione. Inoltre modifico x senza riutilizzare il suo vecchio valore, quindi si spreca solo memoria.

(x viene modificata in riga 5, dato che la chiamata ricorsiva sostituisce x con $x.sx$)

Un altro esempio:

Scorrimento in pre-ordine e post-ordine su una lista

L lista

F funzione che effettua operazioni sull'accumulatore

a accumulatore: variabile che sarà restituita come risultato della funzione

```

1) PreFold( $L, F, a$ )
2)   if  $L = \perp$  then
3)     return  $a$ 
4)   else
5)      $a \leftarrow F(L.dato, a)$ 
6)   return PreFold( $L.next, F, a$ )

```

```

1) IterativePreFold( $L, F, a$ )
2)   while  $L \neq \perp$  do
3)      $a \leftarrow F(L.dato, a)$ 
4)      $L \leftarrow L.next$ 
5)   return  $a$ 

```

La funzione ricorsiva è tail recursive, inoltre L ed a vengono modificati ma non utilizzati dopo la chiamata ricorsiva, F invece viene modificato ma con lo stesso valore (è come se non venisse mai cambiato).

Quindi non c'è bisogno dello stack.

```

1) PostFold( $L, F, a$ )
2)   if  $L = \perp$  then
3)     return  $a$ 
4)   else
5)      $a \leftarrow \text{PostFold}(L.next, F, a)$ 
6)   return  $F(L.dato, a)$ 

```

Nella versione in Post-Order, non essendo tail recursive, non si può applicare lo stesso ragionamento

Se si hanno delle variabili locali, quindi anche i parametri della funzione, che sono utilizzati prima della chiamata ricorsiva e utilizzati dopo, allora serve salvarli.

Dato che nella versione iterativa si avrà che quella variabile ad ogni iterazione verrà sovrascritta dal nuovo valore, se dopo averla modificata si volesse utilizzare il valore precedente, non si potrebbe, quindi va salvato.

Nelle tail recursive il valore può essere sovrascritto senza problemi, tanto non ci saranno utilizzi dopo la chiamata.

Nelle non-tail recursive salviamo le variabili in uno stack. Non è possibile quindi fare a meno di uno stack, ma si farà a meno dello stack di sistema.

6.1 Traduzione di un algoritmo

Questo è un algoritmo che lavora su un albero; non c'è semantica, serve solo come esempio per vedere la traduzione da ricorsivo a iterativo.

```

1) RecFun( $x, i, j$ )
2)    $a \leftarrow F_{ini}(i, j)$ 
3)   if  $x = \perp$  then
4)      $m \leftarrow F_{\perp}(a)$ 
5)   else
6)      $(k_L, h_L) \leftarrow F_{pre}(x, a)$ 
7)      $z_L \leftarrow \mathbf{RecFun}(x.sx, i, k_L)$ 
8)      $(k_R, h_R) \leftarrow F_{in}(x, h_L, z_L)$ 
9)      $z_R \leftarrow \mathbf{RecFun}(x.dx, k_R, j)$ 
10)     $m \leftarrow F_{post}(x, h_L, z_L, h_R, z_R)$ 
11)   $z \leftarrow F_{fin}(m)$ 
12)  return  $z$ ;

```

F , in generale, è una certa funzione che elabora i dati e restituisce un valore, il pedice di ognuna serve solo per dare un minimo di significato.

F_{ini} **iniziale**, viene eseguita all'inizio della funzione ricorsiva

F_{\perp} viene eseguita quando $x = \perp$

F_{pre} eseguita in **pre-order**

F_{in} eseguita in **in-order**

F_{post} eseguita in **post-order**

F_{fin} eseguita alla fine della funzione ricorsiva

Analizziamo quali variabili hanno bisogno dello stack, iniziamo con i parametri:

- x : riga 7, viene sostituita con $x.sx$ (viene sostituito il primo parametro nella chiamata ricorsiva, cioè x)
riga 8, viene letta x che però ho perso avendola sostituita prima

Ha bisogno dello stack

- i : riga 7, viene sovrascritto con lo stesso valore, quindi non crea problemi
riga 9, viene sostituita con k_R , ma i non verrà utilizzata dopo, quindi non crea problemi

Non ha bisogno dello stack

- j : riga 7, viene sostituita con k_L
riga 9, viene letta j che però ho perso avendola sostituita prima

Ha bisogno dello stack

Analizziamo poi le variabili interne:

- a : viene letta ma prima delle chiamate ricorsive ma mai dopo

Non ha bisogno dello stack

- m : non viene mai letta

Non ha bisogno dello stack

- a : viene letta ma prima delle chiamate ricorsive ma mai dopo

Non ha bisogno dello stack

- z_R : viene scritta in riga 9 e letta in riga 10, ma la scrittura avviene dopo le chiamate ricorsive

Non ha bisogno dello stack

- k_R, k_L : vengono scritte prima di una chiamata ricorsiva, ma non lette dopo

Non hanno bisogno dello stack

- h_R, h_L, z_L : vengono scritte prima di una chiamata ricorsiva e lette dopo una chiamata ricorsiva

Hanno bisogno dello stack

Hanno bisogno dello stack: x, j, h_R, h_L, z_L

Utilizziamo una variabile booleana **call** per capire se è stata fatta una chiamata ricorsiva (quindi sto "scendendo", $call = true$) oppure è stato fatto un return (quindi sto "risalendo", $call = false$)

Quando c'è più di una chiamata ricorsiva, bisogna tenere traccia da quale (due in questo caso) si sta ritornando.

In questo caso utilizziamo come discriminante il fatto che la prima chiamata venga fatta sul figlio sinistro di x e la seconda chiamata sul figlio destro. Quindi si salva nella variabile **last** l'ultimo nodo utilizzato. Con un opportuno controllo è possibile sapere quindi se si ritorna dalla prima o dalla seconda chiamata.

Il **return** viene simulato assegnando il valore da restituire in una variabile (**ret**, ad esempio)

In il codice colorato di **rosso**, è il codice della funzione ricorsiva copiato così com'è

```

1) IterativeFun( $x, i, j$ )
2)    $Stack \ S_x, S_j, S_{h_L}, S_{h_R}, S_{z_L}$  // Dichiarazione degli Stack individuati
3)    $call \leftarrow true$ 
4)   while ( $call \vee isNotEmpty(S_x)$ ) do // Scorre finché è indiscesa o c'è ancora qualcosa nello stack
5)     if  $call$  then // La funzione è in discesa ( $call = true$ )
6)        $a \leftarrow F_{ini}(i, j)$ 
7)       if  $x = \perp$  then
8)          $m \leftarrow F_{\perp}(a)$ 
9)          $z \leftarrow F_{fin}(m)$ 
10)         $ret \leftarrow z$ 
11)         $(call, last) \leftarrow (false, x)$  // metto call a false perché ho incontrato un return
                                           // salvo in last l'ultimo nodo letto, ovvero x
12)      else
13)         $(k_L, h_L) \leftarrow F_{pre}(x, a)$ 
14)         $S_x \leftarrow Push(S_x, x)$  // Essendo arrivati ad una chiamata ricorsiva (la prima),
15)         $S_j \leftarrow Push(S_j, j)$  // faccio il push negli stack delle variabili
16)         $S_{h_L} \leftarrow Push(S_{h_L}, h_L)$  // che ne hanno bisogno e che sono state utilizzate fin'ora
17)         $(x, j) \leftarrow (x.sx, k_L)$  // Sostituisco le variabili dei parametri, con gli argomenti
                                           della chiamata
18)      else // La funzione è in risalita ( $call = false$ )
19)         $x \leftarrow Top(S_x)$  // Ho finito di lavorare con la  $x$  corrente, quindi prendo
                               quella precedente senza rimuoverla dallo stack
20)        if  $last = x.sx$  then // Torno da sinistra
21)           $z_L \leftarrow ret$  // Sto tornando dalla prima chiamata ricorsiva,
22)           $j \leftarrow Top(S_j)$  // quindi inserisce il valore di ritorno in  $z_L$ 
23)           $h_L \leftarrow Top(S_{h_L})$  // e riprende i valori delle variabile pushate
24)           $(k_R, h_R) \leftarrow F_{in}(x, h_L, z_L)$ 
25)          if  $x.dx = \perp$  then // Vedi note sotto per la spiegazione di questo if
26)             $z_R \leftarrow F_{fin}(F_{\perp}(F_{ini}(k_R, j)))$ 
27)             $m \leftarrow F_{post}(x, h_L, z_L, h_R, z_R)$ 
28)             $z \leftarrow F_{fin}(m)$ 
29)             $ret \leftarrow z$  // È stato fatto un return, in questo caso non imposto
30)             $last \leftarrow x$  // call a false perché è già false
31)             $(S_x, S_j, S_{h_L}) \leftarrow (Pop(S_x), Pop(S_j), Pop(S_{h_L}))$ 
32)          else
33)             $S_{z_L} \leftarrow Push(S_{z_L}, z_L)$  // Preparo gli stack per fare la chiamata ricorsiva, il push di  $j$ 
34)             $S_{h_R} \leftarrow Push(S_{h_R}, h_R)$  // in questo caso è inutile perché il suo valore non cambia
35)             $(x, i) \leftarrow (x.dx, k_R)$  // Non pusho  $x$  perché prima è stato fatto il Top, non il Pop.
                                           Inserirei lo stesso valore
36)             $call \leftarrow true$  // È stata fatta una chiamata, quindi la funzione scende
37)          else // Torno da destra
38)             $z_R \leftarrow ret$ 
39)             $(S_{h_L}, h_L) \leftarrow Top\&Pop(S_{h_L})$ 
40)             $(S_{h_R}, h_R) \leftarrow Top\&Pop(S_{h_R})$ 
41)             $(S_{z_L}, z_L) \leftarrow Top\&Pop(S_{z_L})$ 
42)             $S_j \leftarrow Pop(S_j)$ 
43)             $ret \leftarrow F_{fin}(F_{post}(x, h_L, z_L, h_R, z_R))$ 
44)             $S_x \leftarrow Pop(S_x)$ 
45)             $last \leftarrow x$ 
46)   return  $ret$ 

```

call è inizialmente *true* perché sta simulando la prima chiamata, oltre al fatto che gli stack sono inizialmente tutti vuoti.

C'è lavoro da fare finché ci sono ancora valori negli stack. In questo caso basta controllare solo lo stack di un parametro della funzione, svuotato quello sicuro lo saranno anche gli altri.

L'*if* in riga 25 simula il caso in cui $x.dx = \perp$. Questo perché il caso in cui è il **sinistro** ad essere \perp viene gestito dall'*if* in riga 7, ma non viene gestito il caso del figlio **destro**.

Questa simulazione viene fatta quando si sta ritornando dalla chiamata ricorsiva sul figlio sinistro, per questo motivo si trova all'interno dell' *if* $last = x.sx$

Ricapitolando:

- Individuare gli stack
- $call \leftarrow true$
- Il *while* scorre finché è in discesa ($call = true$) oppure finché gli stack non sono vuoti.
Al suo interno viene gestita la discesa e la risalita:
 - Se $call = true$ allora la funzione scende, altrimenti risale
 - Dopodiché ricopio il corpo della funzione ricorsiva finché non si incontra un *return* oppure una *chiamata ricorsiva*
- Se incontro un *return*
 - Inserisco in *ret* il valore restituito
 - Imposto il discriminante *last*
 - Eseguo il *Pop* delle variabili utilizzate (pushate in precedenza)
 - Imposto $call \leftarrow false$ perché dopo il *return* la funzione risale
- Se incontro una *chiamata ricorsiva*
 1. Preparo gli stack, quindi faccio il *Push* delle variabili che sono state utilizzate
 2. Sostituisco le variabili dei parametri con gli argomenti della chiamata
 3. Imposto $call \leftarrow true$ perché dopo la *chiamata ricorsiva* la funzione scende

7 Grafi

Un grafo è un insieme di elementi detti nodi o vertici che possono essere collegati fra loro da linee chiamate archi.

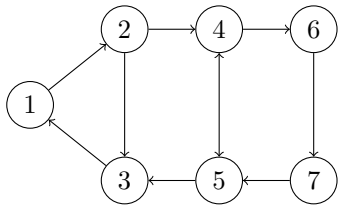
Formalmente, un grafo è una coppia ordinata $G = (V, E)$ di insiemi, dove

- V è l'insieme dei vertici
- E è l'insieme degli archi
Gli archi sono rappresentati da coppie di vertici, ossia il vertice di partenza e il vertice di arrivo dell'arco.
 $E \subseteq V \times V$ da questo si deduce che $|E| \leq |V|^2$

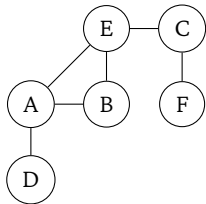
Proprietà

- Un grafo si dice **completo** quando ogni vertice è collegato direttamente a tutti i vertici rimanenti.
- Due vertici si dicono **adiacenti** se esiste un arco tra essi.
- Un **percorso** è una sequenza finita di vertici dove ogni coppia di nodi risulta collegata da un arco presente in E
 $Path(G) = \{(v_i, v_{i+1}) \mid \forall i \in [0, |\pi|) \text{ e } (\pi_i, \pi_{i+1}) \in E\}$
 $\pi \in Path(G) = \{(v_0, v_1), (v_1, v_2), (v_2, v_3), \dots, (v_{|\pi|-2}, v_{|\pi|-1})\} = (v_1, v_2, v_3, \dots, v_{|\pi|-1})$
- Si dice **percorso semplice** un percorso in cui i vertici non si ripetono (ogni vertice compare una sola volta)
 $\pi \in SimplePath(G) \subseteq Path(G) : \forall i, j \in [0, |\pi|), i \neq j, \pi_i \neq \pi_j$
- Un **ciclo** è un percorso dove il primo vertice è anche l'ultimo (ci possono essere cicli interni).
 $\pi \in Cycle(G) \subseteq Path(G) : \pi_0 = \pi_{|\pi|-1}$
- Un ciclo si dice **semplice** se nel percorso non ci sono vertici che si ripetono tranne il primo e l'ultimo
- La **distanza** tra due vertici v_1, v_2 in un grafo G è il numero di archi nel cammino **più corto** tra v_1 e v_2 in G
 $\delta(v_1, v_2) \triangleq \min(|\pi| - 1), \pi \in Path(G)$
- **Reach(G)** è l'insieme di coppie di vertici (v, u) tale che esiste un percorso da v ad u
 $\forall v, u \in V, (v, u) \in Reach(G) \iff \exists \pi \in Path(G) \text{ dove } first(\pi) = v \wedge last(\pi) = u$
- Un grafo non orientato si dice **connesso** se esiste almeno un percorso tra ogni coppia di vertici
- Un grafo orientato si dice **fortemente connesso** se esiste un percorso da ogni vertice ad ogni altro vertice
- Una **componente/componente connessa** di un grafo non orientato G , è un sottografo $G' \sqsubseteq G$ in cui:
 - esiste almeno un percorso tra ogni coppia di vertici, ovvero il sottografo è connesso
 - il sottografo non è connesso a nessun vertice del supergrafo
- Una **componente fortemente connessa** di un grafo orientato G è un sottografo $G' \sqsubseteq G$ massimale e fortemente connesso
- Un sottografo è **massimale** quando non si possono più aggiungere vertici affinché questo rimanga fortemente connesso

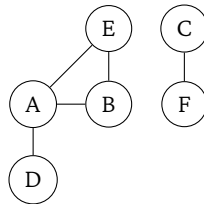
Esempi



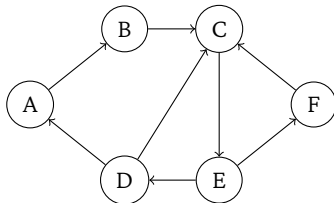
$$G = \left(\underbrace{\{1, 2, 3, 4, 5, 6, 7\}}_V, \underbrace{\{(1, 2), (2, 4), (4, 6), (6, 7), (7, 5), (5, 3), (3, 1), (2, 3), (4, 5), (5, 4)\}}_E \right)$$



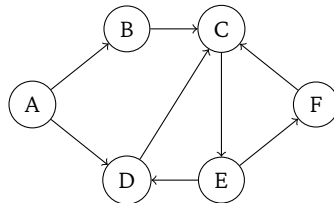
Grafo connesso



Grafo non connesso

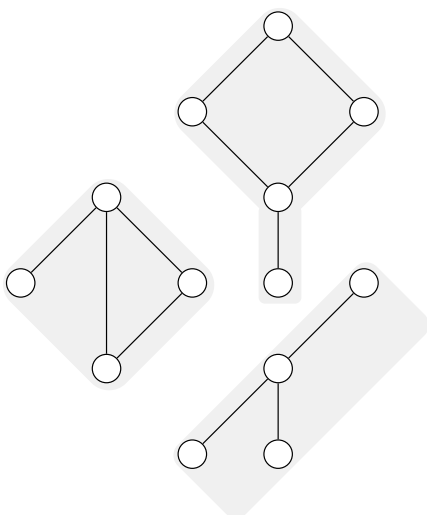


Grafo fortemente connesso



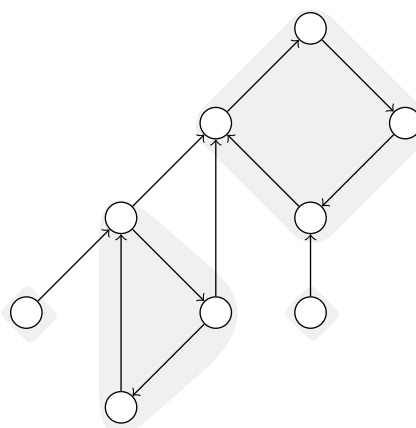
Grafo non fortemente connesso,
è impossibile arrivare al vertice A
partendo da un qualsiasi altro vertice

Componenti



Il grafo possiede 3 componenti
(evidenziate in grigio)

Componenti fortemente connesse



Il grafo possiede 4 componenti
fortemente connesse
(evidenziate in grigio)

7.1 Rappresentazione dei grafi

Ci sono due modi per rappresentare un grafo in un computer:

- come una collezione di **liste di adiacenza**
- come **matrice di adiacenza**

La rappresentazione con **liste di adiacenza** consiste in un array Adj (adjacent - adiacenti) di $|V|$ liste, una per ogni vertice di V ; per ogni $u \in V$, la lista $Adj[u]$ contiene tutti i vertici adiacenti a u .

Metodo maggiormente utilizzato per i grafi **sparsi**, ovvero $|E|$ è molto più piccola di $|V|^2$

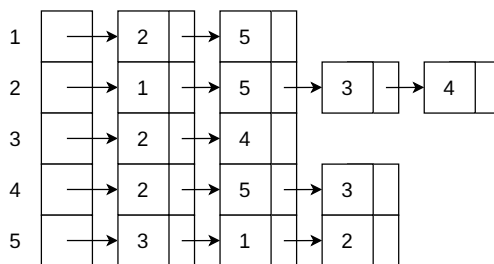
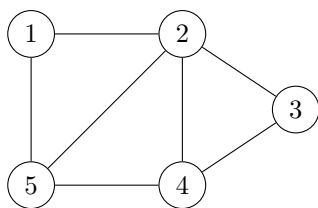
La rappresentazione con **matrice di adiacenza** suppone che i vertici siano numerati (non importa l'ordine).

Consiste in una matrice A di dimensione $|V| \times |V|$ tale che

$$a_{ij} = \begin{cases} 1 & \text{se } (i, j) \in E \\ 0 & \text{altrimenti} \end{cases} \quad \begin{array}{l} \text{Quindi 1 rappresenta l'esistenza di un arco fra il vertice all'indice } i \text{ e il vertice all'indice } j, \\ 0 \text{ la non esistenza} \end{array}$$

Metodo maggiormente utilizzato per i grafi **densi**, ovvero $|E|$ è vicina a $|V|^2$

Esempio



$$A = \begin{matrix} & \begin{matrix} 1 & 2 & 3 & 4 & 5 \end{matrix} \\ \begin{matrix} 1 \\ 2 \\ 3 \\ 4 \\ 5 \end{matrix} & \begin{bmatrix} 0 & 1 & 0 & 0 & 1 \\ 1 & 0 & 1 & 1 & 1 \\ 0 & 1 & 0 & 1 & 0 \\ 0 & 1 & 1 & 0 & 1 \\ 1 & 1 & 0 & 1 & 0 \end{bmatrix} \end{matrix}$$

7.2 Esplorazione di un grafo

Sono due i principali metodi per esplorare un grafo:

- Visita in **ampiezza**: permette di calcolare la distanza fra i nodi; se sono presenti dei cicli può non terminare o impiegare troppo tempo.
- Visita in **profondità**: permette di trovare i cicli.

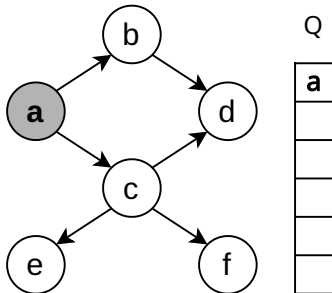
7.2.1 Visita in ampiezza

Se il grafo è rappresentato con le liste, non è possibile esplorarlo in ampiezza senza una coda.

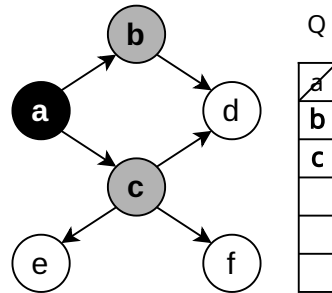
I nodi del grafo assumeranno, nel corso della visita, tre stati:

- Visitato
- Scoperto non visitato (coda)
- Non scoperto

Per distinguere questi tre stati, si utilizzerà una colorazione del nodo.



Supponiamo di iniziare la visita dal nodo *a*, quindi lo incodo e lo coloro di **grigio** perché l'ho scoperto.

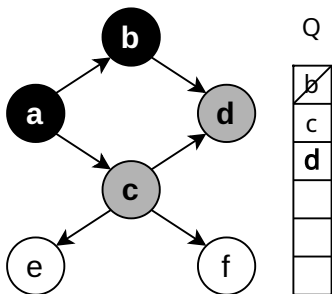


Itero finché la coda non è vuota.

In questo caso c'è *a* quindi lo estraggo.

Incodo e coloro di **grigio** la sua stella uscente, ovvero i nodi *b* e *c*.

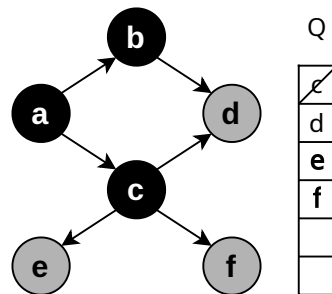
Successivamente *a* ha finito la visita e viene colorato di **nero**.



Ora in testa c'è *b* quindi lo estraggo.

Incodo e coloro di **grigio** la sua stella uscente, ovvero il nodo *d*.

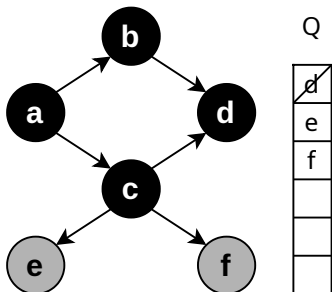
Successivamente *b* ha finito la visita e viene colorato di **nero**.



Ora in testa c'è *c* quindi lo estraggo.

Incodo e coloro di **grigio** la sua stella uscente, ovvero i nodi *e* ed *f*.

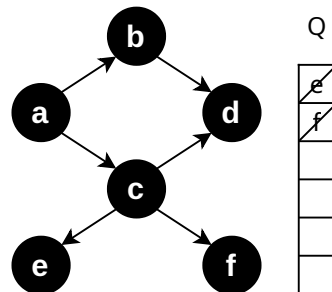
Successivamente *c* ha finito la visita e viene colorato di **nero**.



Ora in testa c'è *d* quindi lo estraggo.

Incodo e coloro di **grigio** la sua stella uscente.

Non essendoci alcun arco uscente, *d* ha finito la visita e viene colorato di **nero**.



Ora in testa c'è *e* quindi lo estraggo.

Incodo e coloro di **grigio** la sua stella uscente.

Non essendoci alcun arco uscente, *d* ha finito la visita e viene colorato di **nero**.

Stesso discorso per *f*

Algoritmo: BFS (Breadth First Search)

```

1) BFS( $G, v$ )
2)    $Init(G, c)$ 
3)    $(Q, c(v)) \leftarrow (Enqueue(Q, v), gr)$ 
4)   while  $isNotEmpty(Q)$  do
5)      $(Q, v) \leftarrow Head\&Dequeue(Q)$ 
6)     for each  $u \in Adj[v]$  do
7)       if  $c(u) = bn$  then
8)          $(Q, c(u)) \leftarrow (Enqueue(Q, u), gr)$ 
9)    $c(v) \leftarrow nr$ 

```

```

1) Init( $G, c$ )
2)   for each  $v \in V$  do
3)      $c(v) \leftarrow bn$ 

```

Colori:

- bn Bianco: Non scoperto
- gr Grigio: Scoperto ma non visitato
- nr Nero: Visitato

Init : prende in ingresso un grafo G e un array c il quale tiene traccia del colore del nodo v
 Imposta a **bianco** di tutti i nodi del grafo G

Dato che un grafo non ha un nodo radice dal quale partire, basta quindi un nodo arbitrario v dal quale partire.
 La funzione *BFS* quindi prende in ingresso il grafo e il nodo dal quale partire.

1. Si inizializza il vettore dei colori dei vertici c con il colore che identifica che il nodo non è stato scoperto, ovvero il **bianco** (avendo prima specificato il significato dei tre colori).
2. Ogni volta che si incontra un nodo **bianco** vuol dire che lo si vuole visitare, quindi lo inserisco in coda e imposto il colore a **grigio**; ovvero lo scopro ma non lo visito, lo visiterò quando arriverà il suo turno.
 Quindi come seconda cosa, scopro v . Lo metto in coda e lo coloro di **grigio**.
3. C'è lavoro da fare finché ci sono nodi nella coda, quindi con un **while** scorro finché la coda non è vuota.
4. Devo visitare la stella uscente del nodo v in testa alla coda, quindi lo estraggo. per visitarlo.
5. Dovrò visitare ogni nodo u adiacente al nodo v appena estratto, a patto che questo sia **bianco**.
 Quindi, se u è **bianco**, lo inserisco in coda e imposto il suo colore a **grigio**.
6. Una volta tutta la stella uscente di v , ho finito di visitarlo, quindi imposto il colore a **nero**.

Quindi una semplice BFS ci permette di calcolare solo ciò che il nodo di partenza v raggiunge nel grafo G .
 Lo si può capire guardando i colori dei nodi alla fine della funzione.

I nodi colorati di **nero** saranno quelli che il nodo di partenza v raggiunge nel grafo.

$$\forall u \in V \quad c(u) = nr \iff (v, u) \in Reach(G)$$

Se il nodo u è **nero**, allora c'è un percorso che va da v ad u , quindi v raggiunge u

Se v raggiunge u , allora vuol dire che u è **nero**

Si può estendere l'algoritmo per:

- calcolare la distanza dal nodo di partenza a tutti gli altri nodi
- calcolare il percorso con lunghezza minima.

7.2.2 Calcolo delle distanze

```

1) DistanceBFS(  $G, v$  )
2)    $(c, d, p) \leftarrow \text{Init}(G)$ 
3)    $(Q, c(v), d(v)) \leftarrow (\text{SingletonQueue}(v), gr, 0)$ 
4)   while  $\text{isNotEmpty}(Q)$  do
5)      $(Q, v) \leftarrow \text{Head\&Dequeue}(Q)$ 
6)     for each  $u \in \text{Adj}[v]$  do
7)       if  $c(u) = bn$  then
8)          $(Q, c(u), d(u), p(u)) \leftarrow (\text{Enqueue}(Q, u), gr, d(v)+1, v)$ 
9)      $c(v) \leftarrow nr$ 
10)  return  $(c, d, p)$ 

```

```

1) Init(  $G$  )
2)   for each  $v \in V$  do
3)      $(c(v), d(v), p(v)) \leftarrow (bn, \infty, \perp)$ 
4)   return  $(c, d, p)$ 

```

- Imposta il colore di tutti i nodi del grafo a **bianco**
- Imposta le distanze a infinito
- Imposta i predecessori a *null* (\perp bottom)

c Vettore dei colori: $\forall u \in V, c(u) = \text{colore}$

d Vettore delle distanze: $\forall u \in V, d(u) = \text{numero}$ ovvero la *distanza dal vertice di partenza v a u*

p Vettore dei predecessori: $\forall u \in V, p(u) = \text{predecessore}$

Complessità di tempo

Un vertice può non entrare in coda, ma se entra lo farà una sola volta (data la colorazione).

Nel caso peggiore entrano tutti vertici.

Il **for each** si traduce nella somma del numero di adiacenti di u , questo $\forall u \in V$. Non stiamo facendo altro che contare gli archi $|E|$. $\sum_{u \in V} \Theta(|\text{Adj}[u]|) = |E|$

Le istruzioni nel **while** sono $\Theta(1)$; come detto prima, nel caso peggiore entrano tutti i vertici, quindi il **while** si traduce nella $\sum_{v \in V} \Theta(1) = |V|$

Nel totale avremo: $\sum_{v \in V} (\Theta(1) + \Theta(|\text{Adj}[u]|)) = |V| + |E| = |G|$

7.2.3 Percorso minimo

Calcola il percorso minimo da v ad u . Restituisce uno stack di vertici che rappresentano il percorso.

```

1) MinimalPath(  $G, v, u$  )
2)    $(c, \_, p) \leftarrow DistanceBFS(G, v)$ 
3)   if  $c(u) = nr$  then
4)     return  $BuildMinimalPath(EmptyStack, p, v, u)$ 
5)   return  $EmptyStack$ 

```

- La funzione *DistanceBFS* colorerà di **nero** tutti i percorsi attraversati da v , questo servirà a capire se c'è un percorso da v a u , ovvero $(v, u) \in Reach(G)$.
- Inoltre, la funzione *DistanceBFS* imposta anche i predecessori di ogni nodo e, per come è strutturata la funzione, seguendo a ritroso i predecessori, si ricava il percorso minimo.

Viene impostato il predecessore di un vertice v nel momento in cui questo viene **scoperto**, ma un vertice viene **scoperto** quando il predecessore visita i suoi adiacenti.

Se il vertice v ha più padri, allora v si troverà nella lista di adiacenza di più vertici. Ma v sarà **scoperto la prima volta** dal vertice che ci arriva prima. Chi arriverà dopo, troverà il vertice v già colorato.

Di conseguenza, se si segue il percorso dei predecessori a ritroso, si ricava il percorso minimo.

- Se u è colorato di **nero**, allora vuol dire che v raggiunge u (ovvero $(v, u) \in Reach(G)$) ed è quindi possibile costruire il percorso.
Se $(v, u) \notin Reach(G)$ allora restituisce uno stack vuoto, dato che non è possibile costruire il percorso minimo.

```

1) BuildMinimalPath(  $S_\pi, p, v, u$  )
2)    $(S_\pi) \leftarrow Push(S_\pi, u)$ 
3)   if  $u \neq v$  then
4)      $S_\pi \leftarrow BuildMinimalPath(S_\pi, p, v, p(u))$ 
5)   return  $S_\pi$ 

```

S_π Stack di vertici raffiguranti il percorso

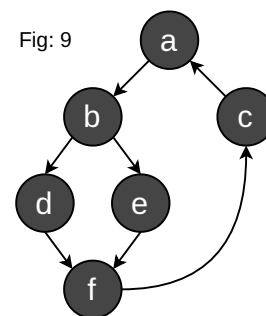
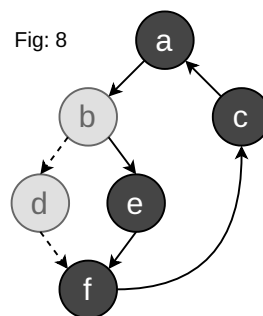
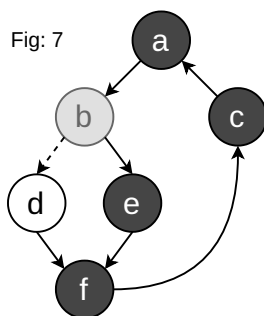
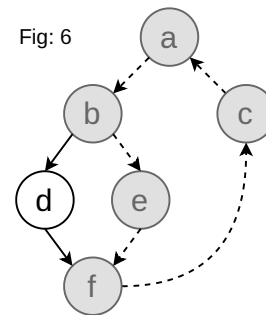
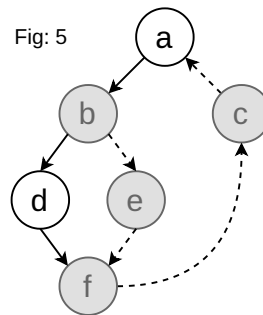
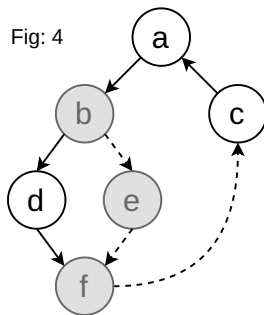
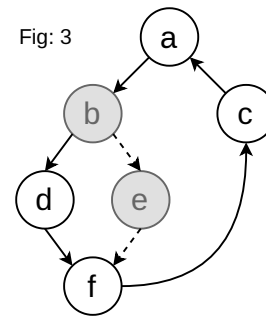
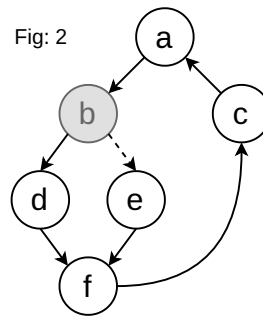
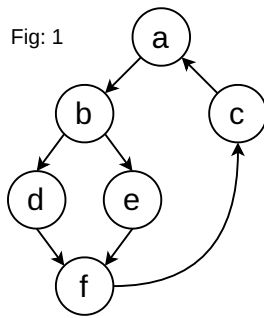
p Vettore dei predecessori

Dato che il percorso viene costruito partendo dall'ultimo vertice e procedendo a ritroso, questi vengono inseriti in uno stack. Così facendo, quando si leggerà lo stack, lo si leggerà partendo dal primo vertice del percorso.

La ricorsione avviene sul parametro u sostituito con il suo predecessore $p(u)$

La ricorsione termina quando il predecessore è il vertice iniziale.

7.2.4 Visita in profondità



- Inizialmente tutti i nodi vengono colorati di **bianco**.
- Supponiamo che la visita parta dal nodo *b*, questo viene scoperto e quindi colorato di **grigio**. Successivamente esplora la stella uscente.
- Supponiamo che il nodo *b* visiti prima il nodo *e*, questo viene scoperto e quindi colorato di **grigio**.
- Il controllo passa al nodo *e* il quale farà la stessa cosa, ovvero esplorare la stella uscente. In questo caso è presente solo *f* come adiacente.
- Viene ripetuto il procedimento finché il nodo corrente non ha più nodi da esplorare. In questo caso si procede fino al nodo *a* il quale **tenta** di visitare *b* che però è già stato scoperto.
- Il nodo *a* quindi termina la sua visita e viene colorato di **nero**. Passa il controllo al chiamante, in questo caso *c* il quale a sua volta non ha altro da visitare, termina la visita e viene colorato di **nero**. E così via fino ad arrivare a *b* che ha da visitare ancora *d*.
- Il nodo *d* tenta di visitare *f* ma che è già stato visitato, termina la sua visita, viene colorato di **nero** e passa il controllo a *b*.
- Anche *b* ha terminato e viene colorato di **nero**.
- Non essendoci più nodi da visitare, l'algoritmo termina.

Visita in profondità: algoritmo

```

1) DFS( $G$ )
2)    $(c, p) \leftarrow \text{Init}(G)$ 
3)    $t \leftarrow 0$ 
4)   for each  $v \in V$  do
5)     if  $c(v) = \text{bn}$  then
6)        $(c, p, d, f, t) \leftarrow \text{DFSVisit}(G, v, c, p, d, f, t)$ 
7)   return  $(c, p, d, f)$ 

```

La funzione *DFS* esegue la visita su tutti i nodi **bianchi** del grafo.

La funzione di visita *DFSVisit* partirà da un certo nodo e scorrerà fino in fondo, colorando i nodi che incontra.

Di conseguenza, il **for each** in *DFS* non visiterà nuovamente i nodi già visitati da qualche altro nodo durante la visita *DFSVisit*.

```

1) DFSVisit(  $G, v, c, p, d, f, t$  )
2)    $(c(v), d(v), t) \leftarrow (gr, t, t + 1)$ 
3)   for each  $w \in \text{Adj}[v]$  do
4)     if  $c(w) = \text{bn}$  then
5)        $p(w) \leftarrow v$  // Imposto il predecessore di w
6)        $(c, p, d, f, t) \leftarrow \text{DFSVisit}(G, w, c, p, d, f, t)$ 
7)    $(c(v), f(v), t) \leftarrow (nr, t, t + 1)$ 
8)   return  $(c, p, d, f, t)$ 

```

La funzione *DFSVisit* esegue la vera e propria visita.

Dato un nodo, *DFSVisit* ha il compito di scendere, e quindi effettuare la visita, fin tanto che può; ossia finché arriva ad un nodo che non ha archi uscenti.

t Tempo, è un intero che rappresenta il momento in cui un nodo inizia o finisce la visita

d Vettore del tempo di inizio visita

f Vettore del tempo di fine visita

c Vettore dei colori

p Vettore dei predecessori

d, f, t sono solo utili a dimostrare il **teorema della struttura a parentesi delle DFS**, non servono per la visita in profondità.

- Ho scoperto il nodo v quindi lo coloro di **grigio**, imposto il tempo di inizio visita con t , poi incremento t .
- Il nodo corrente v visita la stella uscente.
 - Se il nodo adiacente w non è stato ancora scoperto, ovvero è **bianco**, allora prima di passare al nodo adiacente successivo, scende a visitare w .
- Terminata la visita dei suoi adiacenti, v termina il suo lavoro: viene colorato di **nero** e viene impostato il tempo di fine visita
(NB.: viene assegnato t come tempo di fine visita, ma le chiamate ricorsive l'avranno incrementato, quindi non è lo stesso di prima)

7.3 Teorema della struttura a parentesi (DFS)

Una proprietà della visita in profondità è che i tempi di scoperta e di completamento di un vertice, hanno una **struttura a parentesi**.

Se rappresentiamo la scoperta di un vertice con una parentesi aperta " v " e il completamento con una parentesi chiusa " v ", allora la storia delle scoperte e dei completamenti produce un'espressione in cui le parentesi sono ben annidate.

Giusto: $(v (w w) v)$

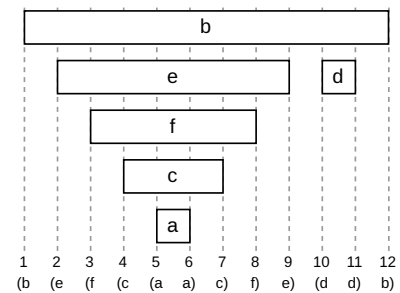
Sbagliato: $(v (w v) w)$

Sono solo due i casi giusti che possono verificarsi:

1. Aperto il vertice v , il vertice w inizia e termina **prima** che termini v
2. Aperto il vertice v , il vertice w inizia **dopo** la chiusura di v

$$1a. d(v) < d(w) < f(w) < f(v) \quad 2a. d(v) < f(v) < d(w) < f(w)$$

$$1b. d(w) < d(v) < f(v) < f(w) \quad 2b. d(w) < f(w) < d(v) < f(v)$$

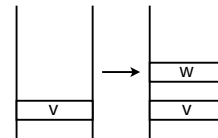


Esempio del grafo precedente

Dimostrazione per assurdo

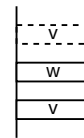
Supponiamo per assurdo che questo sia corretto: $(v (w v) w)$

Se rappresentassimo l'apertura e la chiusura dei vertici come il *push* e il *pop* in uno stack, avremmo uno stack del genere dove prima viene fatto il *push* di v e poi il *push* di w



Successivamente viene chiuso v , ma per fare il *pop* di v dallo stack:

- o questo si trova al *top* dello stack, ma ciò vuol dire che v è stato scoperto **due volte** e questo non può accadere
- o è assurdo dato che devo fare prima il *pop* di w



Foresta di visita

$$\mathcal{F} = \left\{ (p(v), v) \mid v \in V, p(v) \neq \perp \right\}$$

Insieme formato dagli archi che vanno **dal** predecessore di v **a** v $p(v) \rightarrow v$

Lemma

$$\forall v, w \in V, v \neq w$$

$$d(v) < d(w) < f(w) < f(v) \iff \exists \pi \in \mathcal{F} \text{ da } v \text{ a } w$$

7.4 Teorema del percorso bianco

$\forall v, w \in V$

Al tempo $d(v)$, ovvero quando v viene scoperto, w è discendente di v se e solo se w può essere raggiunto da v solo da un percorso **bianco**.

1) w è discendente di v , ossia $\exists \pi \in Path(\mathcal{F}, v, w)$

\Updownarrow

2) al tempo $d(v)$, $\exists \hat{\pi} \in Path(G, v, w) : \forall i \in (0, |\hat{\pi}|) \ c(\hat{\pi}_i) = bn$

\Downarrow) Se w è discendente di v , allora quando l'algoritmo si trova al tempo di inizio visita di v , esiste un percorso che va da v a w totalmente **bianco**.

Se il percorso non fosse **bianco**, w non sarebbe discendente di v , questo perché l'algoritmo scende in profondità solo se il nodo in cui dovrà scendere è **bianco**.

\Uparrow) Se quando inizia la visita di v esiste un percorso **bianco** che va da v a w , allora w è discendente di v .

w è discendente di v ma supponiamo per assurdo che a tempo $d(v)$ il percorso che va da v a w non sia tutto **bianco**.

Dato che w è un discendente di v , vuol dire che esiste un suo predecessore $p(w) = z$.

Questo vuol dire che anche z è un discendente di v .

Per il teorema della struttura a parentesi, z inizia dopo e termina prima di v $\underbrace{d(v) < d(z) < f(z) < f(v)}$

Ma per come è formato l'algoritmo, z non termina se non esplora tutta la sua stella uscente, quindi esplora per forza anche w $\underbrace{d(v) < d(z) < d(w) < f(w) < f(z) < f(v)}$

Di conseguenza non è possibile che w sia discendente di v senza che ci sia un percorso **bianco** quando inizia v

7.5 Aciclicità

Algoritmo che determina se un grafo è aciclico (non ha cicli)

```

1) Acyclic(  $G$  )
2)    $c \leftarrow \text{Init}(C)$ 
3)   for each  $v \in V$  do
4)     if  $c(v) = bn$  then
5)        $acyclic \leftarrow \text{DFSAcyclic}(G, v)$ 
6)       if  $\neg acyclic$  then //
7)         return  $\perp$ 
8)   return  $\top$ 
```

```

1) DFSAcyclic(  $G, v$  )
2)    $c(v) \leftarrow gr$ 
3)   for each  $w \in \text{Adj}[v]$  do
4)     if  $c(w) = bn$  then
5)        $acyclic \leftarrow \text{DFSAcyclic}(G, w)$ 
6)       if  $\neg acyclic$  then
7)          $c(v) \leftarrow nr$ 
8)         return  $\perp$ 
9)     else if  $c(w) = gr$  then
10)      return  $\perp$ 
11)   $c(v) \leftarrow nr$ 
12)  return  $\top$ 
```

La struttura è identica ad una *DFS*.

Inizializza il colore di tutti i vertici a **bianco**.

Controlla ogni vertice **non scoperto (bianco)** del grafo, se solo uno di questi trova un ciclo, il grafo non è aciclico e restituisce *false*.

Controllati tutti i vertici, se nessuno di questi ha trovato un ciclo, allora il grafo è aciclico e restituisce *true*

DFSAcyclic:

Il vertice v è stato scoperto, quindi viene colorato di **grigio** e viene esplorata la stella uscente:

- se il nodo adiacente w è **bianco**, allora scende ricorsivamente in w
- se il nodo adiacente w è **grigio**, allora è stato incontrato un nodo che è stato già visitato durante la discesa, quindi è stato trovato un ciclo

DFSAcyclic:

- Nel caso migliore, impiega tempo costante $\Omega(1)$
- Nel caso peggiore, impiega $O(|G|)$

7.6 Ordinamento topologico

Un ordinamento topologico di un DAG (Directed Acyclic Graph) è un ordinamento lineare di tutti i suoi vertici tale che se G contiene un arco (v, w) , allora v appare prima di w nell'ordinamento.

Possiamo anche vedere un ordinamento topologico di G , come una permutazione¹ dei suoi nodi.

Quindi, un ordinamento topologico è un percorso $\pi \in \text{Perm}(V)$, di conseguenza possono esistere anche più ordinamenti topologici per lo stesso grafo.

$$\exists i, j \in [0, |V|), i < j : \pi_i = v \wedge \pi_j = w$$

Non può esistere un ordinamento topologico se il grafo ha dei cicli, **deve essere aciclico**

7.6.1 Teorema

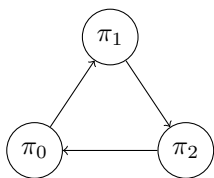
Se G è aciclico allora esiste un ordinamento topologico

G è aciclico $\iff \exists$ un ordinamento topologico per G

Dimostrazione

\Leftarrow per assurdo

Se esiste un ordinamento topologico, allora il grafo è aciclico



Possibile ordinamento topologico: $\pi_0 \rightarrow \pi_1 \rightarrow \pi_2 \rightarrow \pi_0$

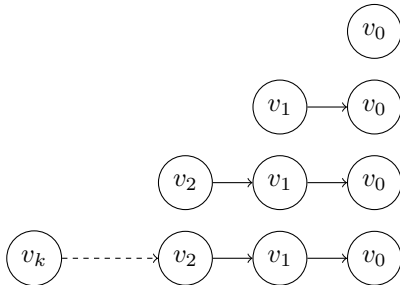
Essendo l'ordinamento topologico una permutazione (di V), vuol dire che i vertici **non possono ripetersi**.

In questo caso si ripete π_0

\Rightarrow per assurdo

Se il grafo è aciclico, allora esiste almeno un ordinamento topologico

Supponiamo che non esistono vertici senza archi entranti.



Partiamo da v_0 che ha v_1 come arco entrante.

A sua volta v_1 deve avere un arco entrante, v_2

v_2 a sua volta deve avere un arco entrante e così via, fino a non avere più nodi in V .

L'ultimo nodo v_k non può avere archi entranti:

- sono finiti i nodi e in una permutazione non possono esserci ripetizioni
- se pure ci fosse una ripetizione, si formerebbe un ciclo

Ma dato che il grafo è aciclico e finito, v_k non ha archi entranti, il che va in contraddizione con quanto supposto prima.

Osservazione

Dato un grafo G aciclico, anche un suo sottografo G' è aciclico

G aciclico $\wedge G' \subseteq G \Rightarrow G'$ è aciclico

Dimostrazione per assurdo

Supponiamo che il sottografo G' sia ciclico e G aciclico.

Per definizione di sottografo, gli archi di G' sono anche archi di G .

Ciò vuol dire che se G' ha dei cicli, anche G avrà quegli stessi cicli e questo va in contraddizione con quanto supposto prima

¹Una permutazione di un insieme X , è una sequenza degli elementi di X dove ogni elemento appare una sola volta. Possono esserci al massimo $|X|!$ permutazioni.

7.6.2 Algoritmo (ampiezza)

Dato un grafo G , la funzione restituisce una lista π di vertici di G che rappresenta un ordinamento topologico. Restituisce solo un ordinamento dei tanti che potrebbero esserci

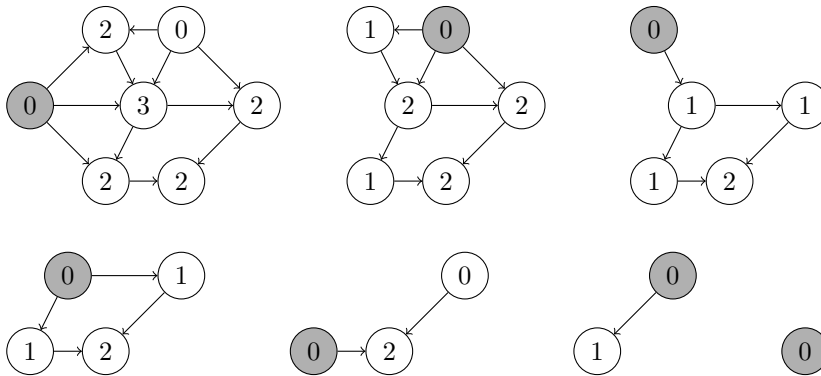
Grado entrante di un vertice

Si definisce **grado entrante** del vertice v , il numero di archi entranti in v

$$gr(v) \triangleq \left| \left\{ \forall w \in V \mid (w, v) \in E \right\} \right|$$

Quando un nodo ha 0 archi entranti, non è vincolato da nessun altro nodo, può essere quindi inserito nella lista π . Possiamo immaginare che il nodo appena inserito in π sia stato anche eliminato dal grafo, di conseguenza i gradi dei nodi adiacenti diminuiscono di 1.

Nell'esempio sotto, i numeri all'interno dei nodi rappresentano il numero di archi entranti.



L'algoritmo sfrutta la visita in ampiezza.

```

1) TopologicalOrdering(  $G$  )
2)    $ge \leftarrow EnteringDegree(G)$ 
3)    $Q \leftarrow InitQueue(G, ge)$ 
4)   while  $isNotEmpty(Q)$  do
5)      $(Q, v) \leftarrow Head\&Dequeue(Q)$ 
6)     for each  $w \in Adj[v]$  do
7)        $ge(w) \leftarrow ge(w) - 1$ 
8)       if  $ge(w) = 0$  then
9)          $Q \leftarrow Enqueue(Q, w)$ 
10)     $\pi \leftarrow Append(\pi, v)$ 
11) return  $\pi$ 

```

In coda entrano solo i nodi **senza** archi entranti.

Estraggo il nodo v in coda e diminuisco il grado entrante di tutti i suoi adiacenti.

Se uno di questi, una volta decrementato, ha grado pari a 0, allora lo incodo anche.

Finito con gli adiacenti, aggiunge v alla lista π

```

1) EnteringDegree(  $G$  )
2)   for each  $v \in V$  do
3)      $ge(v) \leftarrow 0$ 
4)   for each  $v \in V$  do
5)     for each  $w \in Adj[v]$  do
6)        $ge(w) \leftarrow ge(w) + 1$ 
7) return  $ge$ 

```

Imposta prima tutti i gradi a 0.

Successivamente se un nodo w è adiacente di un altro nodo v , allora w ha l'arco entrante (v, w) , quindi il grado di w aumenta di 1

```

1) InitQueue(  $G, ge$  )
2)   for each  $v \in V$  do
3)     if  $ge(v) = 0$  then
4)        $Q \leftarrow Enqueue(Q, v)$ 
5) return  $Q$ 

```

Inserisce in coda solo i nodi che non hanno archi entranti.

7.6.3 Algoritmo (profondità)

È possibile calcolare un ordinamento topologico anche usando una DFS.

In questo caso non viene restituita una lista ma uno stack.

La funzione inserisce nello stack a partire dal nodo più vincolato, quindi dall'ultimo che avremo nell'ordinamento. Inserendolo in uno stack, quando lo si andrà leggere, lo si leggerà in maniera ordinata.

```
1) TopologicalOrdering(  $G$  )
2)    $c \leftarrow \text{Init}(G)$  // Colora tutti i vertici di bianco
3)   for each  $v \in V$  do
4)     if  $c(v) = bn$  then
5)        $S \leftarrow \text{DFS TopologicalOrdering}(G, S, v, c)$ 
6)   return  $S$ 
```

```
1) DFS TopologicalOrdering(  $G, S, v, c$  )
2)    $c(v) \leftarrow gr$ 
3)   for each  $w \in \text{Adj}[v]$  do
4)     if  $c(w) = bn$  then
5)        $S \leftarrow \text{DFS TopologicalOrdering}(G, S, w, c)$ 
6)    $(c(v), S) \leftarrow (nr, \text{Push}(S, v))$ 
7)   return  $S$ 
```

7.7 Calcolo delle componenti fortemente connesse

Ricordiamo che in un grafo G orientato, una componente fortemente connessa è un sottografo $G' \subseteq G$ massimale e fortemente connesso.

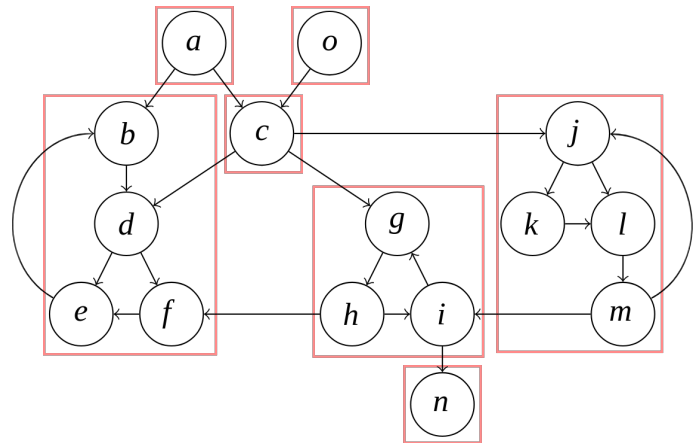
In questo esempio sono cerchiare le componenti fortemente connesse.

Così com'è non è possibile calcolare le componenti fortemente connesse, dato che non sappiamo da dove parta la visita.

Se ad esempio la visita in profondità partisse da c , troverebbe anche altre componenti:

- b, d, e, f
- n
- g, h, i
- j, k, l, m

e la includerebbe nella sua, formando quindi un'unica componente, il che è sbagliato.



Bisogna quindi seguire tre step:

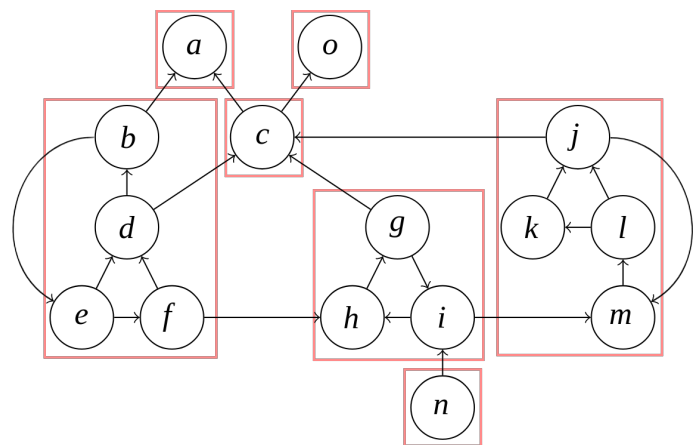
1. Eseguire l'ordinamento topologico in profondità su G
2. Calcolare il grafo trasposto² G^T di G
3. Eseguire una DFS leggermente modificata su G^T e sullo stack S restituito dall'ordinamento topologico

L'ordinamento topologico in profondità, restituisce uno stack S con i vertici ordinati in modo decrescente per tempo di fine visita, ovvero dall'ultimo che finisce al primo che finisce.

La visita in G^T viene fatta sui vertici nello stack S (dall'ord. topologico di prima)

- Parte da a ma non ci sono archi uscenti, quindi termina la sua visita.
- Continua con o ma non ci sono archi uscenti, quindi termina la sua visita.
- Continua con c ma non ci sono archi uscenti, quindi termina la sua visita.
- Continua con j , c'è un arco uscente quindi scende in m
 - m scende in l
 - l prova a visitare j ma è stato già scoperto; visita k
 - k prova a visitare j ma è stato già scoperto.
 - Le visite terminano.
- E così via per tutti gli altri nodi.

Un possibile ordinamento topologico:
 $S : \underline{a}, \underline{o}, \underline{c}, \underline{j, k, l, m}, \underline{g, h, i}, \underline{n}, \underline{d, f, e, b}$



Grafo trasposto

Se non avessimo trasposto il grafo, nella prima visita, il nodo a avrebbe trovato b e c come archi uscenti e il risultato sarebbe stato sbagliato.

²Grafo in cui tutti gli archi sono stati invertiti di direzione

Algoritmi

 $G = \langle V, E \rangle \quad G^T = \langle V, E^{-1} \rangle$

```

1) Transpose(  $G$  )
2)    $V_{G^T} \leftarrow V_G$ 
3)   for each  $v \in V_G$  do
4)     for each  $w \in Adj[v]$  do
5)        $E_{G^T} \leftarrow Insert(E_{G^T}, (w, v))$ 
6)   return  $(V_{G^T}, E_{G^T})$ 

```

I vertici rimangono gli stessi, quindi faccio una semplice copia.

Gli archi sono rappresentati dagli adiacenti di un vertice, in questo caso da v a w .

Basterà quindi inserire nell'insieme degli archi del grafo trasposto E_{G^T} , l'arco che va da w a v .

Strongly Connected Component

```

1) SCC(  $G$  )
2)    $S \leftarrow TopologicalOrdering(G)$ 
3)    $G^T \leftarrow Transpose(G)$ 
4)    $scc \leftarrow DFS\_SCC(G^T, S)$ 
5)   return  $scc$ 

```

Il vettore **scc** contiene il vertice rappresentante della componente alla quale l' i -esimo vertice partecipa.

Ad esempio, la componente j, k, l, m ha come rappresentante j solo perché compare prima nell'ordinamento topologico, ma ognuno di questi vertici sarebbe potuto esserlo.

- $scc(j)$: j j si trova nella componente con rappresentante j
- $scc(k)$: j k si trova nella componente con rappresentante j
- $scc(l)$: j l si trova nella componente con rappresentante j
- $scc(m)$: j m si trova nella componente con rappresentante j

```

1) DFS\_SCC(  $G, S$  )
2)    $(c, scc) \leftarrow Init(G)$ 
3)   while  $isNotEmpty(S)$  do
4)      $(S, v) \leftarrow Top\&Pop(S)$ 
5)     if  $c(v) = bn$  then
6)        $(c, scc) \leftarrow DFS\_SCC\_Visit(G, c, scc, v, v)$ 
7)   return  $scc$ 

```

Init imposta i colori dei vertici a **bianco** e dichiara un vettore scc

Scorre i vertici dello stack, ovvero i vertici dell'ordinamento topologico ordinati in ordine inverso sul tempo di fine visita, ed esegue una DFS su di essi.

```

1) DFS\_SCC\_Visit(  $G, c, scc, v, w$  )
2)    $(c(w), scc(w)) \leftarrow (gr, v)$ 
3)   for each  $z \in Adj[w]$  do
4)     if  $c(z) = bn$  then
5)        $(c, scc) \leftarrow DFS\_SCC\_Visit(G, c, scc, v, z)$ 
6)    $c(w) \leftarrow nr$ 
7)   return  $(c, scc)$ 

```

Il parametro v è il vertice rappresentante della componente.

Il parametro w è il vertice attuale.

Lo scopo è quello di scendere in profondità impostando la componente rappresentante di ogni vertice della componente, ovvero $scc(w) \leftarrow v$