

Algoritmi e Strutture Dati

Introduzione

Informazioni utili

- T.H. Cormen, C.E. Leiserson, R.L Rivest, C. Stein “**Introduzione agli algoritmi e strutture dati**”. McGraw-Hill
- Sito web con le **slides** del corso:
<http://people.na.infn.it/~bene/ASD/>
- Orario di ricevimento: **Mercoledì ore 11:00 – 13:00**

Algoritmo

Un **algoritmo** è una procedura ben definita per risolvere un problema: una sequenza di passi che, se eseguiti da un *esecutore*, portano alla ***soluzione del problema***.

La sequenza di passi che definisce un algoritmo deve essere ***descritta in modo finito***.

Alcune proprietà degli algoritmi

Non ambiguità: tutti i passi che definiscono l'algoritmo devono essere non ambigui e chiaramente comprensibili all'esecutore;

Generalità: la sequenza di passi da eseguire dipende esclusivamente dal problema generale da risolvere, non dai dati che ne definiscono un'istanza specifica;

Correttezza: un algoritmo è corretto se produce il risultato corretto a fronte di qualsiasi istanza del problema ricevuta in ingresso. Può essere stabilita, ad esempio, tramite:

- dimostrazione formale (matematica);
- ispezione informale;

Efficienza: misura delle risorse computazionali che esso impiega per risolvere un problema. Alcuni esempi sono:

- tempo di esecuzione;
- memoria impiegata;
- altre risorse: banda di comunicazione.

Complessità degli algoritmi

- Analisi delle *prestazioni degli algoritmi*
- Utilizzeremo un *Modello Computazionale* astratto di riferimento.
- *Tempo di esecuzione* degli algoritmi
- *Notazione asintotica*
- Analisi del *Caso Migliore*, *Caso Peggior*e e del *Caso Medio*
- Applicazione delle tecniche di analisi del tempo di esecuzione ad *algoritmi di ordinamento*.

Analisi di un algoritmo

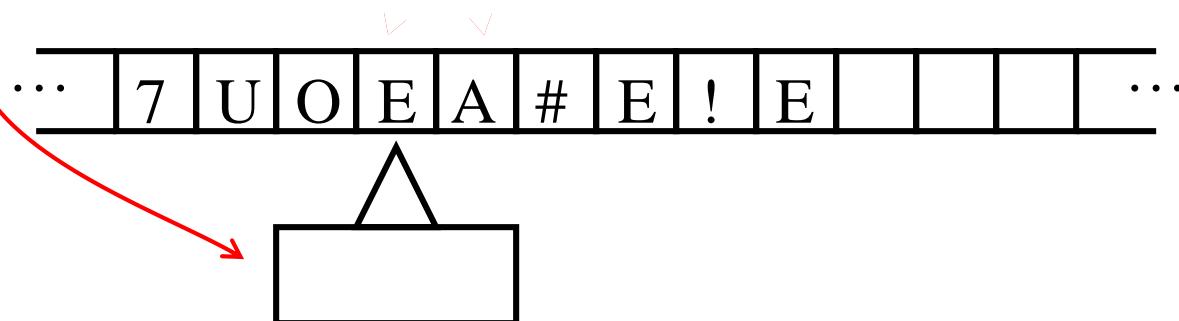
- **Correttezza**
 - **Dimostrazione formale (matematica)**
 - **Ispezione informale**
- **Utilizzo delle risorse**
 - **Tempo di esecuzione**
 - **Utilizzo della memoria**
 - **Altre risorse: banda di comunicazione**
- **Semplicità**
 - **Facile da capire, modificare e manutenere**

Tempo di esecuzione

- Il *tempo di esecuzione* di un *programma* può dipendere da vari fattori:
 - Hardware su cui viene eseguito
 - Compilatore/Interprete utilizzato
 - *Tipo e dimensione dell'input*
 - Altri fattori: casualità, ...
- Al fine di analizzare il *tempo intrinseco impiegato* da un algoritmo, procederemo a un'analisi più astratta, impiegando un *modello computazionale*.

Un noto modello computazionale

- Il modello della **Macchina di Turing**
 - **Nastro di lunghezza infinita**
 - o In ogni *cella* può essere contenuta una quantità di informazione finita (un simbolo)
- Una testina + un processore + programma
 - In 1 unità di tempo
 - Legge o scrive la cella di nastro corrente e
 - Si muove di 1 cella a sinistra, oppure di 1 cella a destra, oppure resta ferma



Il modello computazionale RAM

Modello RAM (Random-Access Memory)

- **Memoria principale infinita**
 - Ogni cella di memoria può contenere una quantità di dati finita.
 - Impiega lo stesso tempo per accedere a ogni cella di memoria.
- **Singolo processore + programma**
 - In 1 unità di tempo: operazioni di *lettura, passo di computazione elementare, scrittura*;
 - Passi di computazione: addizione, moltiplicazione, assegnamento, confronto, accesso a puntatore, ...

Il modello RAM è una semplificazione dei moderni computer.

Un problema di conteggio

- **Input**
 - Un intero N dove $N \geq 1$.
- **Output**
 - Il numero di coppie ordinate (i, j) tali che i e j siano interi e $1 \leq i \leq j \leq N$.
- Esempio:
 - Input: $N=4$
 - $(1,1), (1,2), (1,3), (1,4), (2,2), (2,3), (2,4), (3,3), (3,4), (4,4)$
 - Output: 10

Algoritmo 4: analisi asintotica

```
int Count_0( int N)
```

```
1   sum = 0
```

```
2   for i =1 to N
```

```
3       for j =1 to N
```

```
4           if i <= j then
```

```
5               sum = sum+1
```

```
6   return sum
```

1

$$2 \sum_{i=1}^{N+1} 1 = 2(N+1)$$

$$2 \sum_{i=1}^N \sum_{j=1}^{N+1} 1 = 2 \sum_{i=1}^N (N+1)$$

$$2 \sum_{i=1}^N N$$

$$\sum_{i=1}^N (N-i+1)$$

1

Ma notate
che:

$$\sum_{i=1}^N (N+1-i) = \sum_{i=1}^N i = N(N+1)/2$$

Algoritmo 4: analisi asintotica

```
int Count_0( int N)
1   sum = 0           1
2   for i =1 to N    2(N+1)
3       for j =1 to N 2 $\sum_{i=1}^N (N+1)$ 
4           if i <= j then 2 $\sum_{i=1}^N N$ 
5               sum = sum+1  $\frac{N(N+1)}{2}$ 
6   return sum         1
```

Il tempo di esecuzione è $T(N) = 9/2 N^2 + 9/2 N + 4$

Algoritmo 1

```
int Count_1(int N)
1    sum = 0           1
2    for i = 1 to N   2N + 2
3        for j = i to N 2 $\sum_{i=1}^N (N-i+2)$ 
4            sum = sum + 1 2 $\sum_{i=1}^N (N+1-i)$ 
5    return sum         1
```

Il tempo di esecuzione è

$$2 + 2N + 2 + 2\sum_{i=1}^N (N+2-i) + 2\sum_{i=1}^N (N+1-i) = 2N^2 + 6N + 4$$

Algoritmo 2

```
int Count_2(int N)
1   sum = 0
2   for i = 1 to N
3       sum = sum + (N+1-i)
4   return sum
```

The diagram illustrates the execution time for each line of the algorithm. A vertical line separates the code from its corresponding time values. The time values are: 1 for line 1, $2N + 2$ for line 2, $4N$ for line 3, and 1 for line 4.

Line Number	Time Value
1	1
2	$2N + 2$
3	$4N$
4	1

Il tempo di esecuzione è $6N + 4$

Algoritmo 3

$$\sum_{i=1}^N (N+1-i) = \sum_{i=1}^N i = N(N+1)/2$$

```
int Count_3(int N)
```

```
1     sum = N * (N+1) / 2
```

```
2     return sum
```

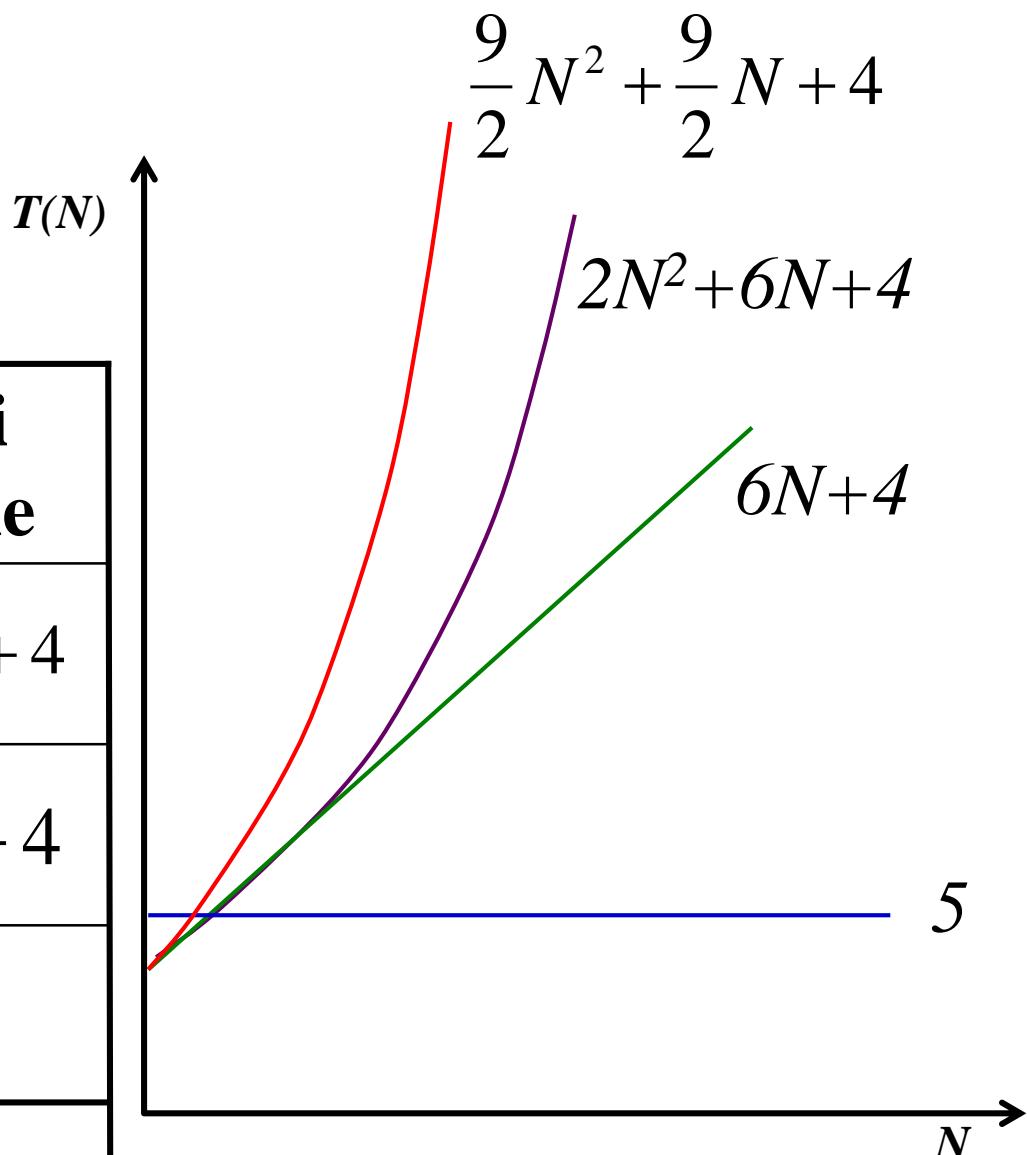
4

1

Il tempo di esecuzione è **5** unità di tempo

Riassunto dei tempi di esecuzione

Algoritmo	Tempo di Esecuzione
Algoritmo 0	$\frac{9}{2}N^2 + \frac{9}{2}N + 4$
Algoritmo 1	$2N^2 + 6N + 4$
Algoritmo 2	$6N+4$
Algoritmo 3	5



Ordine dei tempi di esecuzione

Supponiamo che 1 operazione atomica
impieghi $1 \text{ ns} = 10^{-9} \text{ s}$

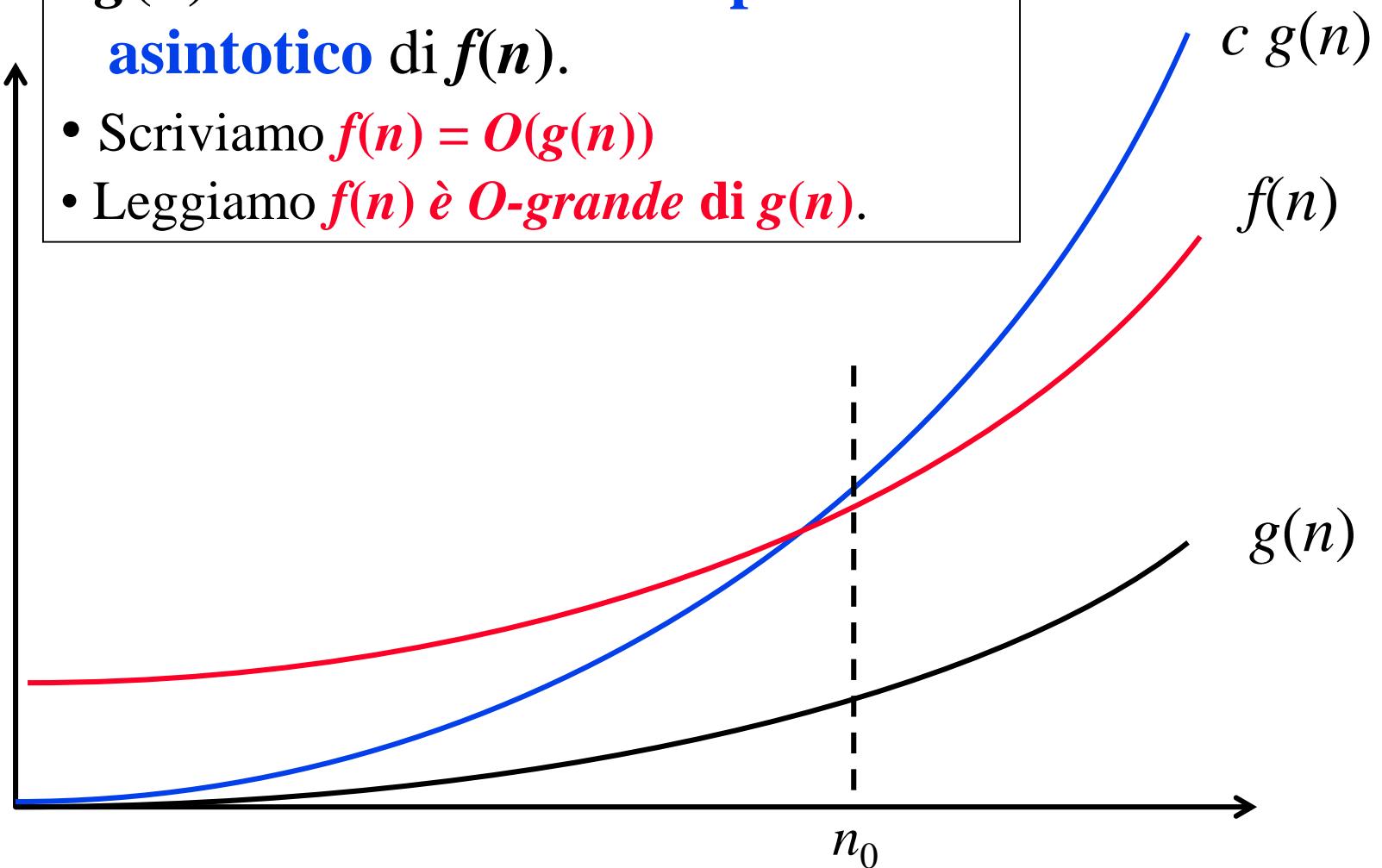
	1.000	10.000	100.000	1.000.000	10.000.000
N	1 μs	10 μs	100 μs	1 ms	10 ms
20N	20 μs	200 μs	2 ms	20 ms	200 ms
N Log N	9.96 μs	132 μs	1.66 ms	19.9 ms	232 ms
20N Log N	199 μs	2.7 ms	33 ms	398 ms	4.6 sec
N ²	1 ms	100 ms	10 sec	17 min	1.2 giorni
20N ²	20 ms	2 sec	3.3 min	5.6 ore	23 giorni
N ³	1 sec	17 min	12 gior.	32 anni	32 millenni

Riassunto dei tempi di esecuzione

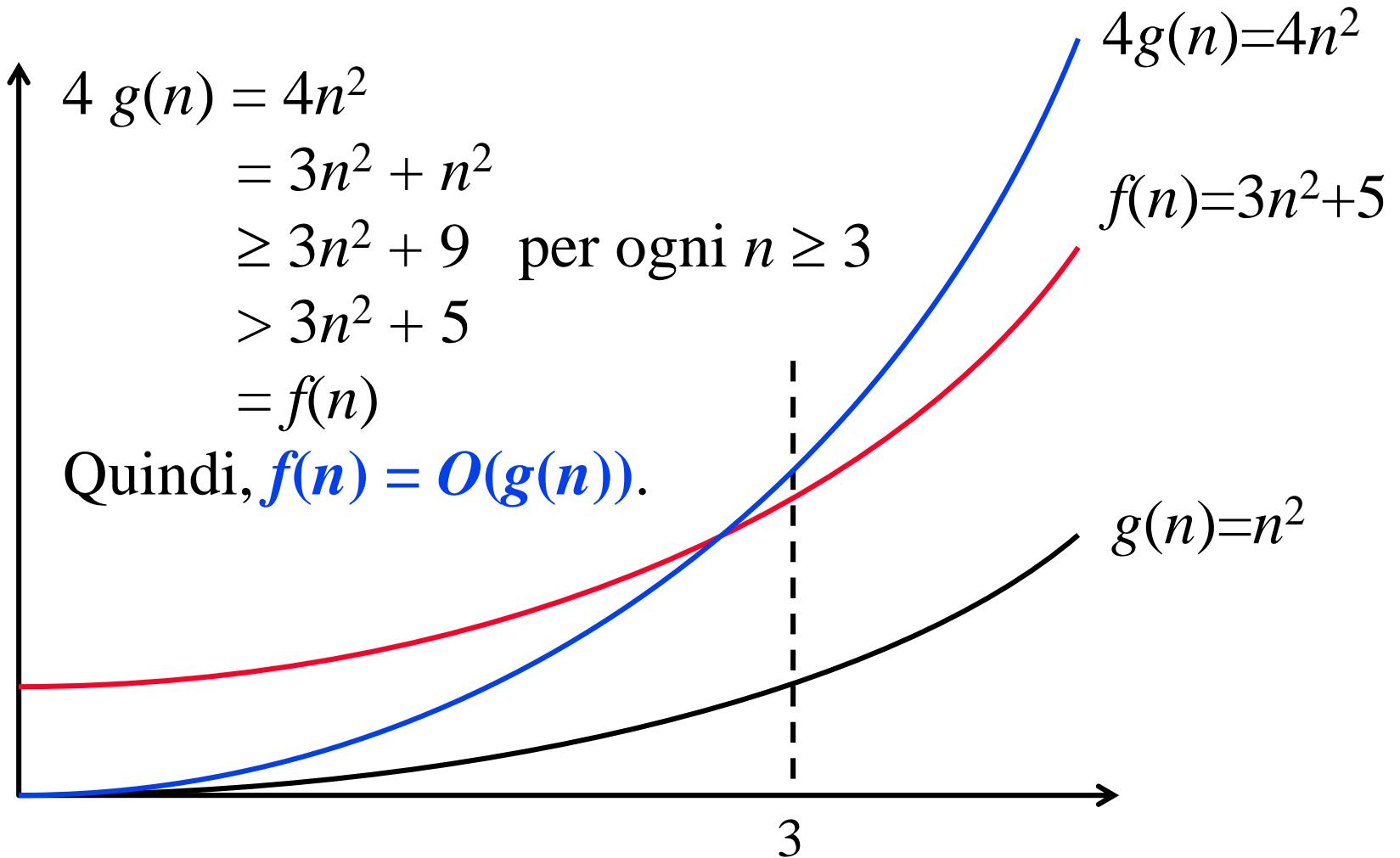
Algoritmo	Tempo di Esecuzione	Ordine del Tempo di Esecuzione
Algoritmo 0	$\frac{9}{2}N^2 + \frac{9}{2}N + 4$	N^2
Algoritmo 1	$2N^2 + 6N + 4$	N^2
Algoritmo 2	$6N+4$	N
Algoritmo 3	5	Costante

Limite superiore asintotico

- $\exists c > 0, n_0 > 0 \quad \forall n \geq n_0. \quad f(n) \leq c g(n)$
- $g(n)$ è detto un **limite superiore asintotico** di $f(n)$.
- Scriviamo $f(n) = O(g(n))$
- Leggiamo $f(n)$ è *O-grande* di $g(n)$.



Esempio di limite superiore asintotico



Esercizio sulla notazione O

- Mostrare che $3n^2 + 2n + 5 = O(n^2)$

$$\begin{aligned}10n^2 &= 3n^2 + 2n^2 + 5n^2 \\&\geq 3n^2 + 2n + 5 \text{ per } n \geq 1\end{aligned}$$

$$c = 10, n_0 = 1$$

Utilizzo della notazione O

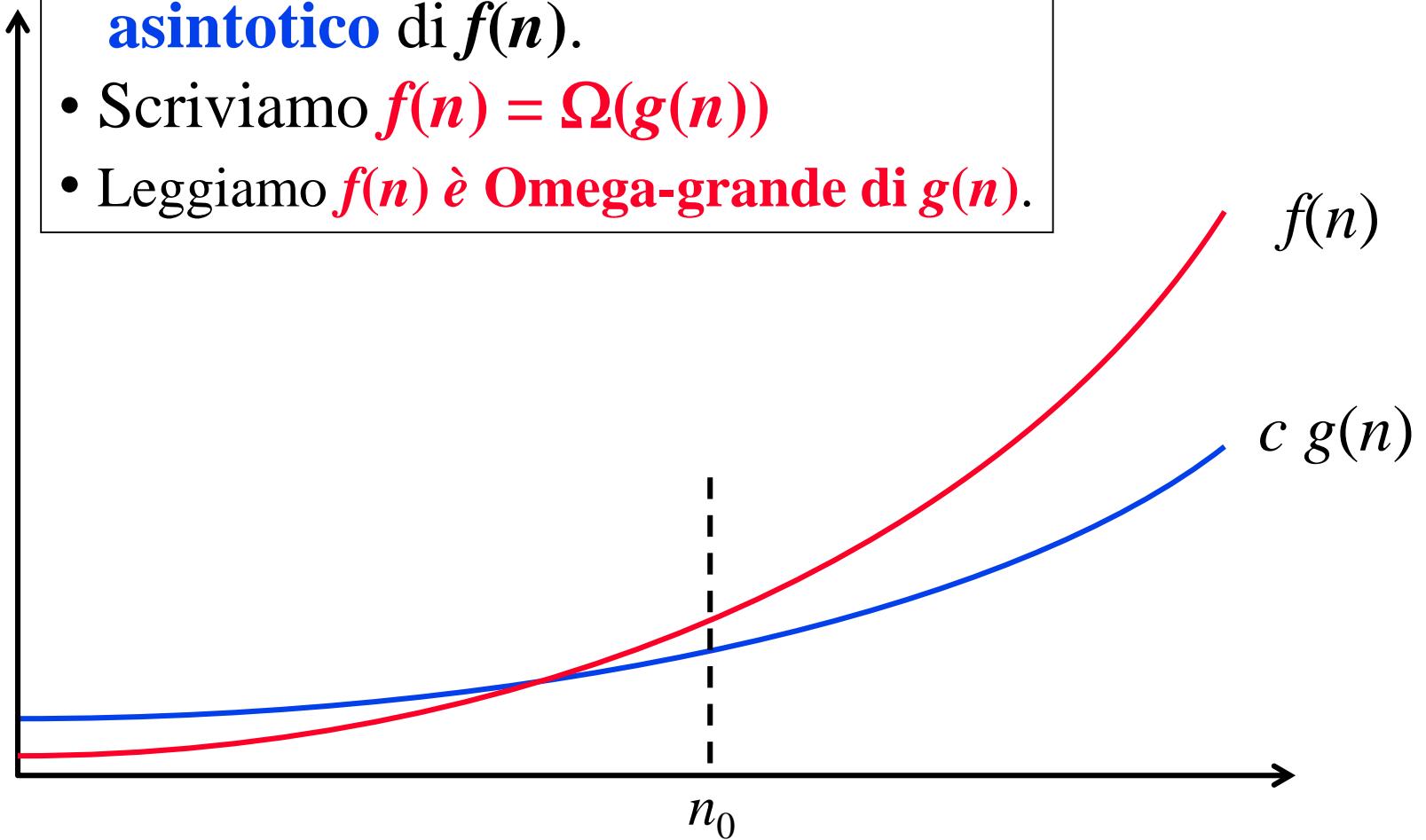
- In genere quando impieghiamo la notazione O , utilizziamo la formula più “*semplice*”.
 - Scriviamo:
 - $3n^2+2n+5 = O(n^2)$
 - Le seguenti sono tutte corrette ma in genere non le si userà:
 - $3n^2+2n+5 = O(3n^2+2n+5)$
 - $3n^2+2n+5 = O(n^2+n)$
 - $3n^2+2n+5 = O(3n^2)$

Esercizi sulla notazione O

- $f_1(n) = 10 n + 25 n^2$ • $O(n^2)$
- $f_2(n) = 20 n \log n + 5 n$ • $O(n \log n)$
- $f_3(n) = 12 n \log n + 0.05 n^2$ • $O(n^2)$
- $f_4(n) = n^{1/2} + 3 n \log n$ • $O(n \log n)$

Limite inferiore asintotico

- $\exists c > 0, n_0 > 0 \ \forall n \geq n_0. \ f(n) \geq c g(n)$
- $g(n)$ è detto un **limite inferiore asintotico** di $f(n)$.
- Scriviamo $f(n) = \Omega(g(n))$
- Leggiamo $f(n)$ è **Omega-grande** di $g(n)$.



Esempio di limite inferiore asintotico

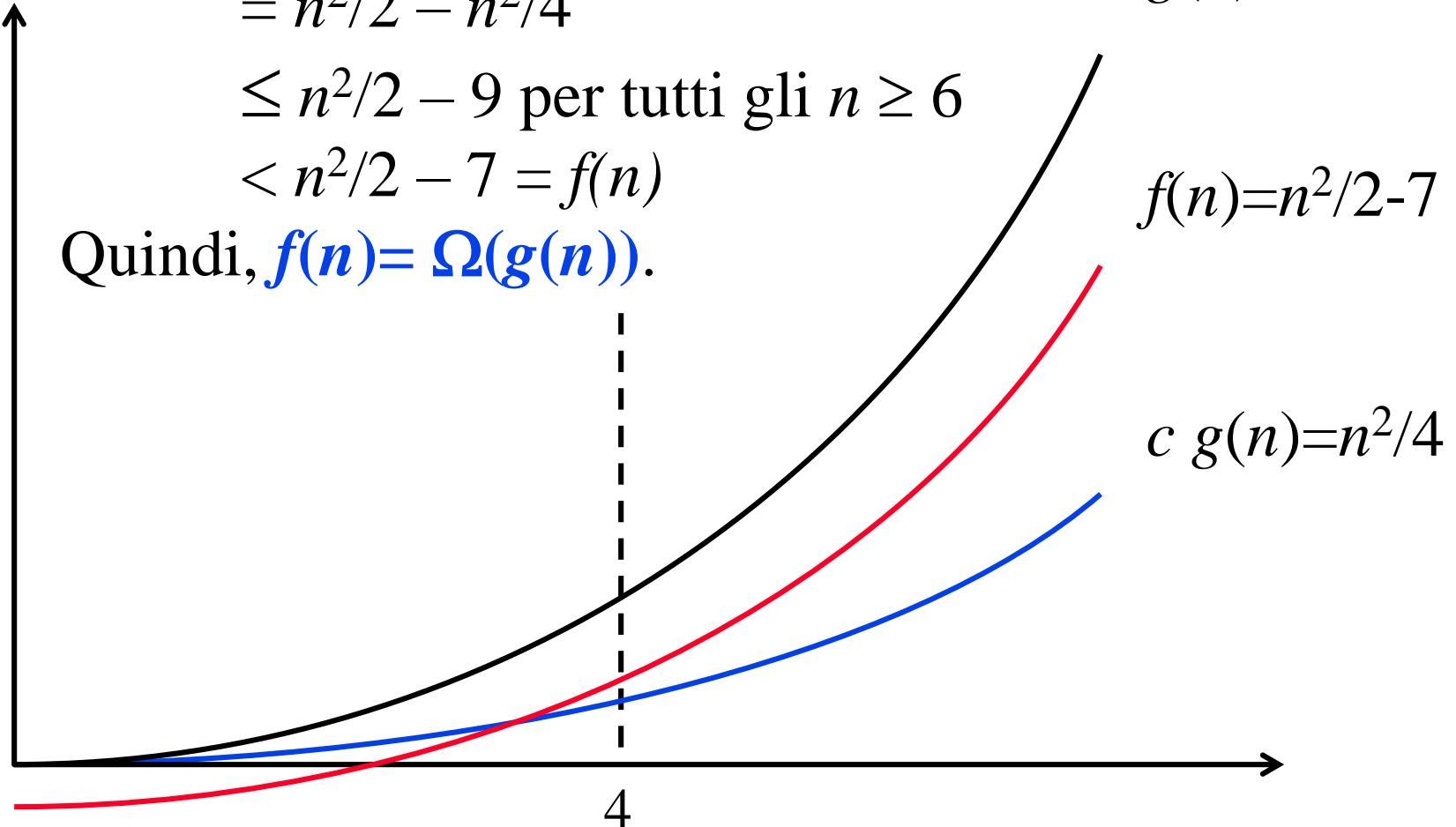
$$g(n)/4 = n^2/4$$

$$= n^2/2 - n^2/4$$

$$\leq n^2/2 - 9 \text{ per tutti gli } n \geq 6$$

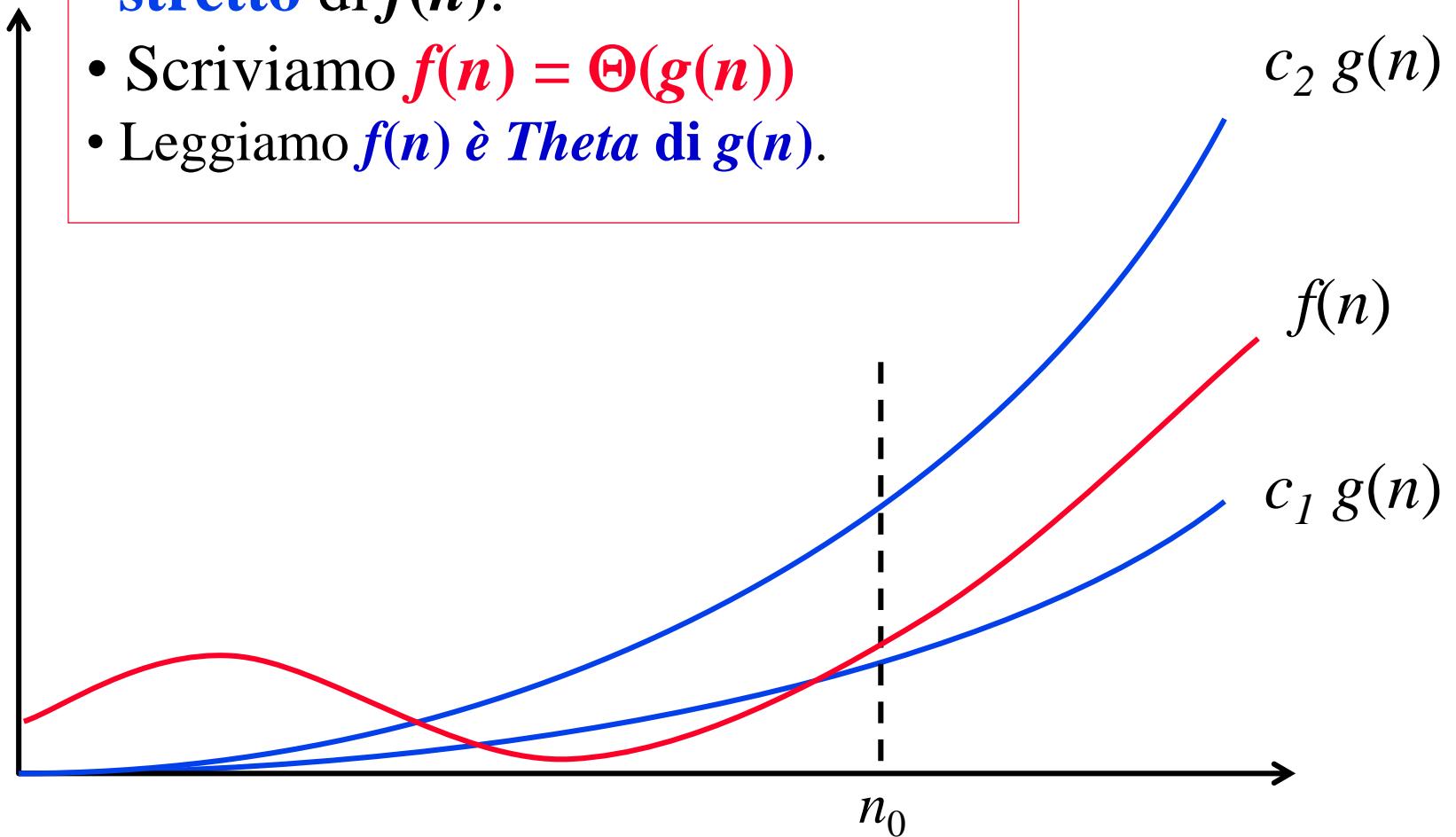
$$< n^2/2 - 7 = f(n)$$

Quindi, $f(n) = \Omega(g(n))$.



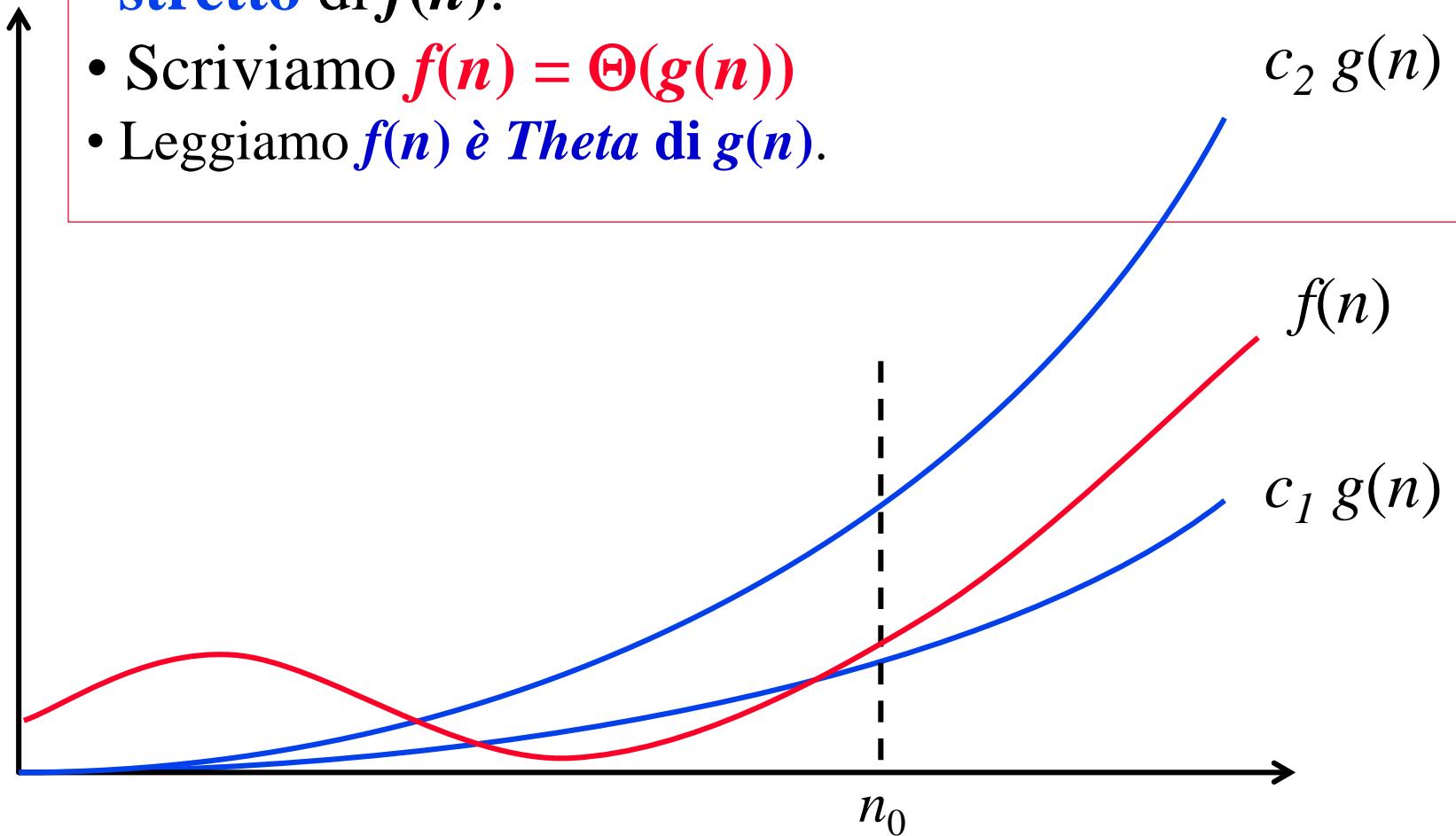
Limite asintotico stretto

- $f(n) = O(g(n))$ e $f(n) = \Omega(g(n))$
- $g(n)$ è detto un **limite asintotico stretto** di $f(n)$.
- Scriviamo $f(n) = \Theta(g(n))$
- Leggiamo $f(n)$ è *Theta* di $g(n)$.



Limite asintotico stretto

- $\exists c_1, c_2 > 0, n_0 > 0 \quad \forall n \geq n_0. \quad c_1 g(n) \leq f(n) \leq c_2 g(n)$
- $g(n)$ è detto un **limite asintotico stretto** di $f(n)$.
- Scriviamo $f(n) = \Theta(g(n))$
- Leggiamo $f(n)$ è *Theta* di $g(n)$.



Riassunto della notazione asintotica

- O : *O-grande*: limite superiore asintotico
- Ω : *Omega-grande*: limite inferiore asintotico
- Θ : *Theta*: limite asintotico stretto
- Usiamo la *notazione asintotica* per dare un limite ad una funzione ($f(n)$), a meno di un fattore costante (c).

Teoremi sulla notazione asintotica

Teoremi:

1. $f(n) = O(g(n))$ se e solo se $g(n) = \Omega(f(n))$.
2. Se $f_1(n) = O(f_2(n))$ e $f_2(n) = O(f_3(n))$, allora $f_1(n) = O(f_3(n))$
3. Se $f_1(n) = \Omega(f_2(n))$ e $f_2(n) = \Omega(f_3(n))$, allora $f_1(n) = \Omega(f_3(n))$
4. Se $f_1(n) = \Theta(f_2(n))$ e $f_2(n) = \Theta(f_3(n))$, allora $f_1(n) = \Theta(f_3(n))$
5. Se $f_1(n) = O(g_1(n))$ e $f_2(n) = O(g_2(n))$, allora
$$O(f_1(n) + f_2(n)) = O(\max\{g_1(n), g_2(n)\})$$
6. Se $f(n)$ è un *polinomio* di grado d , allora $f(n) = \Theta(n^d)$

Teoremi sulla notazione asintotica

Proprietà:

Se $\lim_{n \rightarrow \infty} f(n)/g(n) = 0$ allora $f(n) = O(g(n))$

Se $\lim_{n \rightarrow \infty} f(n)/g(n) = k > 0$ allora $f(n) = O(g(n))$

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

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

Se $\lim_{n \rightarrow \infty} f(n)/g(n) \rightarrow \infty$ allora $f(n) = \Omega(g(n))$

Tempi di esecuzione asintotici

Algoritmo	Tempo di Esecuzione	Limite asintotico
Algoritmo 1	$2N^2 + 6N + 4$	$O(N^2)$
Algoritmo 2	$6N+4$	$O(N)$
Algoritmo 3	5	$O(1)$
Algoritmo 4	$4N^2 + 5N + 4$	$O(N^2)$

Somma Massima di una sottosequenza contigua

- **Input**
 - Un intero N dove $N \geq 1$.
 - Una sequenza (a_1, a_2, \dots, a_N) di N interi.
- **Output**
 - Un intero S tale che $S = \sum_{k=i}^j a_k$ dove $1 \leq i, j \leq N$ e S è il più grande possibile.
 - (tutti gli elementi nella sommatoria devono essere contigui nella sequenza in input).
- Esempio:
 - $N=9, (2, -4, \textcolor{blue}{8}, \textcolor{blue}{3}, \textcolor{red}{-5}, \textcolor{blue}{4}, \textcolor{blue}{6}, -7, 2)$
 - Output = $\textcolor{blue}{8} + \textcolor{blue}{3} - \textcolor{red}{5} + \textcolor{blue}{4} + \textcolor{blue}{6} = \textcolor{red}{16}$

Algoritmo 1

```
int Max_seq_sum_1(int N, array a[])
```

```
    maxsum = 0
```

$O(1)$

```
    for i=1 to N
```

$O(N)$

```
        for j=i to N
```

$O(N^2)$

```
            sum = 0
```

```
            for k=i to j
```

$O(N^3)$

```
                sum = sum + a[k]
```

```
            maxsum = max(maxsum, sum)
```

```
return maxsum
```

Tempo di esecuzione $O(N^3)$

Algoritmo 2

- È facile osservare che l'algoritmo precedente effettua spesso le **stesse operazioni ripetutamente**.
- Poichè

$$\sum_{k=i}^{j+1} a_k = a_{j+1} + \sum_{k=i}^j a_k$$

è possibile ottenere il valore di **sum** per la sequenza da **i** a **j+1** in tempo costante, sommando **A[j+1]** al valore di **sum** già calcolato all'itazione precedente per la sequenza da **i** a **j**.

- A tal fine, è sufficiente mantenere inalterato il valore di **sum** tra le iterazioni che individuano sottosequenze che partono dallo stesso valore **i** e riazzzerare **sum** solo quando **i** viene incrementato.

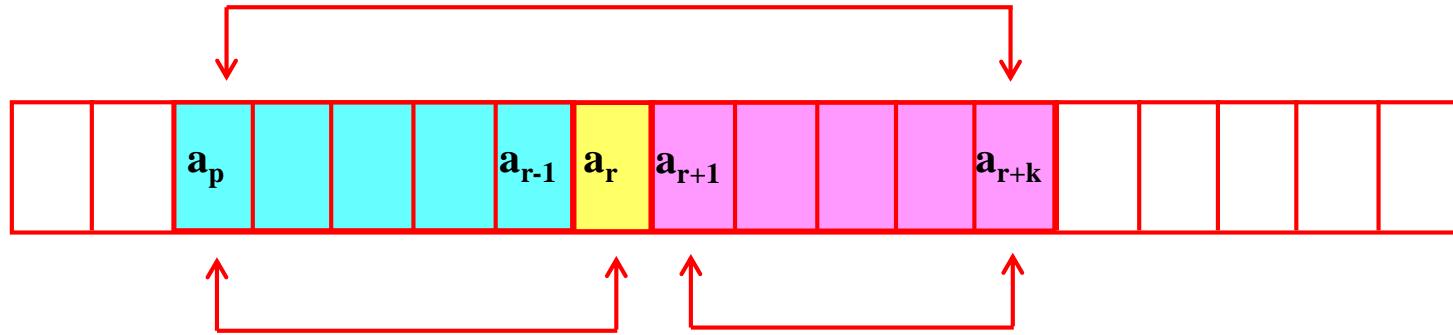
Algoritmo 2

```
int Max_seq_sum_2(int N, array a[])
    maxsum = 0 O(1)
    for i=1 to N O(N)
        sum = 0
        for j=i to N O(N^2)
            sum = sum + a[j]
            maxsum = max(maxsum, sum)
    return maxsum
```

Tempo di esecuzione $O(N^2)$

Esiste un algoritmo che risolve il problema in tempo $O(N)$

Algoritmo 3: intuizione



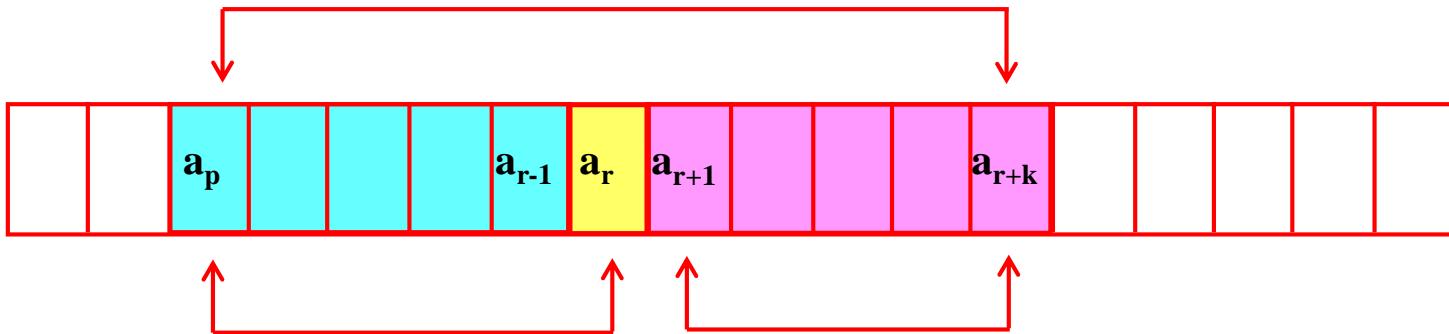
1. Se $a_p + \dots + a_r \geq 0$ allora

$$a_p + \dots + a_{r+k} \geq a_{r+1} + \dots + a_{r+k} \quad \forall k \geq 1$$

2. Se $a_p + \dots + a_{r-1} > 0$ ma $a_p + \dots + a_r < 0$ allora

$$a_p + \dots + a_{r+k} \leq a_{r+1} + \dots + a_{r+k} \quad \forall k \geq 1$$

Algoritmo 3: intuizione



1. Se $a_p + \dots + a_r \geq 0$ allora

$$a_p + \dots + a_{r+k} \geq a_{r+1} + \dots + a_{r+k} \quad \forall k \geq 1$$

2. Se $a_p + \dots + a_{r-1} > 0$ ma $a_p + \dots + a_r < 0$ allora

$$a_p + \dots + a_{r+k} \leq a_{r+1} + \dots + a_{r+k} \quad \forall k \geq 1$$

Nel caso 2, ogni sottosequenza di \mathbf{A} che inizia tra p e r e che termina oltre r avrà una *somma inferiore alla sua sottosequenza* che parte da $r+1$.

È dunque possibile *ignorare tutte queste sottosequenze* e considerare solo quelle che iniziano dall'indice $r+1$.

Algoritmo 3

```
int Max_seq_sum_3(int N, array a[])
```

```
maxsum = 0
```

$O(1)$

```
sum = 0
```

```
for j=1 to N
```

$O(N)$

```
    if (sum + a[j] > 0) then
```

```
        sum = sum + a[j]
```

```
    else
```

```
        sum = 0
```

```
    maxsum = max (maxsum, sum)
```

```
return maxsum
```

Tempo di esecuzione $O(N)$

Ordinamento di una sequenza

- Input : una sequenza di numeri.
- Output : una permutazione (riordinamento) tale che tra ogni 2 elementi adiacenti nella sequenza valga “qualche” relazione di ordinamento (ad es. \leq).
- **Insert Sort**
 - È efficiente solo per piccole sequenze di numeri;
 - Algoritmo di ordinamento sul posto.

- 1) La sequenza viene scandita dal primo elemento; l'indice i , inizialmente assegnato al primo elemento, indica l'elemento corrente;
- 2) Si considera la parte a sinistra di i (compreso) già ordinata;
- 3) Si seleziona il primo elemento successivo ad i nella sottosequenza non-ordinata assegnando $j = i+1$;
- 4) Si cerca il posto giusto per l'elemento j nella sottosequenza ordinata.
- 5) Si incrementa i , si torna al passo 3) se la sequenza non è terminata;

Insert Sort

Algoritmo :

- $A[1..n]$: sequenza numeri di input
- Key : valore corrente da inserire nell'ordinamento

```
1  for j = 2 to Length(A)
2      do Key = A[j]
           /* Scelta del j-esimo elemento da ordinare */
3      i = j-1    /* A[1...i] è la porzione ordinata */
4      while i > 0 and A[i] > Key do
5          A[i+1] = A[i]
6          i=i-1
7      A[i+1] = Key
```

Analisi di Insert Sort

```
1  for j = 2 to Length(A)
2      do Key = A[j]
        /* Commento */
3      i = j-1
4      while i>0 and A[i] > Key
5          do A[i+1] = A[i]
6              i=i-1
7          A[i+1] = Key
```

Numero Esecuzioni	Costo esecuzione singola
-------------------	--------------------------

n	c_1
-----	-------

$n-1$	c_2
-------	-------

$n-1$	0
-------	---

$n-1$	c_3
-------	-------

	c_4
--	-------

	c_5
--	-------

	c_6
--	-------

$n-1$	c_7
-------	-------

Analisi di Insert Sort

```

1  for j = 2 to Length(A)
2      do Key = A[j]
          /* Commento */
3      i = j-1
4      while i>0 and A[i] > Key
5          do A[i+1] = A[i]
6              i=i-1
7      A[i+1] = Key

```

Numero Esecuzioni	Costo esecuzione singola
-------------------	--------------------------

n c_1

$n-1$ c_2

$n-1$ 0

$n-1$ c_3

$\sum_{j=2}^n t_j$ c_4

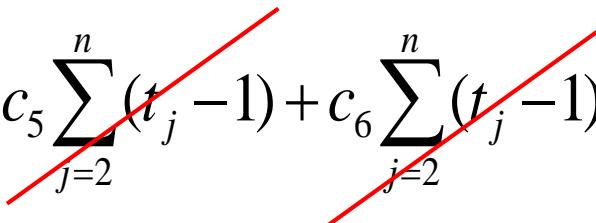
$\sum_{j=2}^n (t_j - 1)$ c_5

$\sum_{j=2}^n (t_j - 1)$ c_6

$n-1$ c_7

$$T(n) = c_1 n + c_2(n-1) + c_3(n-1) + c_4 \sum_{j=2}^n t_j + c_5 \sum_{j=2}^n (t_j - 1) + c_6 \sum_{j=2}^n (t_j - 1) + c_7(n-1)$$

Analisi di Insert Sort: Caso migliore

$$T(n) = c_1 n + c_2(n-1) + c_3(n-1) + c_4 \sum_{j=2}^n t_j + c_5 \sum_{j=2}^n (t_j - 1) + c_6 \sum_{j=2}^n (t_j - 1) + c_7(n-1)$$


Il caso migliore si ha quando l'array è già ordinato:

$$T(n) = c_1 n + c_2(n-1) + c_3(n-1) + c_4 \sum_{j=2}^n t_j + c_7(n-1)$$

Inoltre, in questo caso t_j è **1**, quindi:

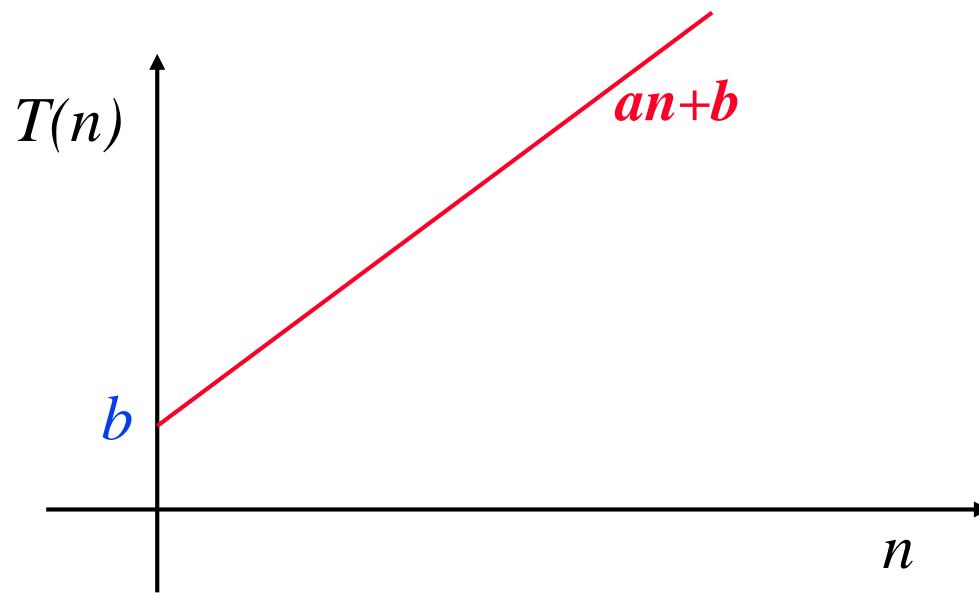
$$T(n) = (c_1 + c_2 + c_3 + c_4 + c_7)n - (c_2 + c_3 + c_4 + c_7)$$

$$\color{red}{T(n)} = \color{blue}{an+b}$$

Analisi di Insert Sort: Caso migliore

$$T(n) = (c_1 + c_2 + c_3 + c_4 + c_7)n - (c_2 + c_3 + c_4 + c_7)$$

$$\textcolor{red}{T(n)} = \textcolor{red}{an+b}$$



Analisi di Insert Sort: Caso peggiore

$$T(n) = c_1 n + c_2(n-1) + c_3(n-1) + c_4 \sum_{j=2}^n t_j + c_5 \sum_{j=2}^n (t_j - 1) + c_6 \sum_{j=2}^n (t_j - 1) + c_7(n-1)$$

Il caso peggiore si ha quando l'array è in ordine inverso.

In questo caso t_j è j (perché?)

$$\sum_{j=2}^n t_j = \sum_{j=1}^n t_j - 1 = \frac{n(n+1)}{2} - 1$$

$$\sum_{j=2}^n (t_j - 1) = \sum_{j=2}^n t_j - \sum_{j=2}^n 1 = \frac{n(n+1)}{2} - 1 - (n-1) = \frac{n(n-1)}{2}$$

Quindi:

$$\begin{aligned} T(n) &= c_1 n + c_2(n-1) + c_3(n-1) + c_4 \left(\frac{n(n+1)}{2} - 1 \right) + \\ &+ c_5 \left(\frac{n(n-1)}{2} \right) + c_6 \left(\frac{n(n-1)}{2} \right) + c_7(n-1) \end{aligned}$$

Analisi di Insert Sort: Caso peggiore

$$T(n) = c_1n + c_2(n-1) + c_3(n-1) + c_4\left(\frac{n(n+1)}{2} - 1\right) + c_5\left(\frac{n(n-1)}{2}\right) + c_6\left(\frac{n(n-1)}{2}\right) + c_7(n-1)$$

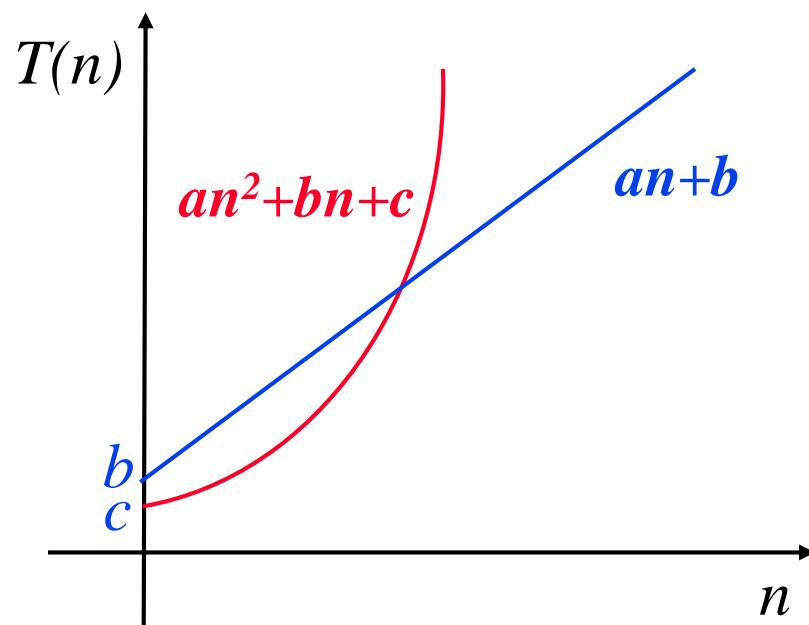
$$T(n) = \left(\frac{c_4 + c_5 + c_6}{2}\right)n^2 + \left(c_1 + c_2 + c_3 + \frac{c_4 - c_5 - c_6}{2} + c_7\right)n - (c_2 + c_3 + c_4 + c_7)$$

$$\color{red} T(n) = an^2 + bn + c$$

Analisi di Insert Sort: Caso peggiore

$$T(n) = \left(\frac{c_4 + c_5 + c_6}{2} \right) n^2 + \left(c_1 + c_2 + c_3 + \frac{c_4 - c_5 - c_6}{2} + c_7 \right) n - (c_2 + c_3 + c_4 + c_7)$$

$$T(n) = an^2 + bn + c$$



Analisi di Insert Sort: Caso medio

$$T(n) = c_1 n + c_2(n-1) + c_3(n-1) + c_4 \sum_{j=2}^n t_j + c_5 \sum_{j=2}^n (t_j - 1) + c_6 \sum_{j=2}^n (t_j - 1) + c_7(n-1)$$

Il **caso medio** è il valore medio del tempo di esecuzione.

Supponiamo di scegliere una **sequenza casuale** e che tutte le sequenze abbiano uguale probabilità di essere scelte.

In media, **metà degli elementi** ordinati saranno **maggiori** dell'elemento che dobbiamo sistemare.

In media **controlliamo metà del sottoarray** ad ogni ciclo **while**.

Quindi t_j è circa $j/2$.

$$\sum_{j=2}^n t_j = \sum_{j=2}^n \frac{j}{2} = \frac{1}{2} \left[\left(\sum_{j=1}^n j \right) - 1 \right] = \frac{n^2 + n - 2}{4}$$

$$\sum_{j=2}^n (t_j - 1) = \sum_{j=2}^n \left(\frac{j}{2} - 1 \right) = \frac{n^2 - 3n + 2}{4}$$

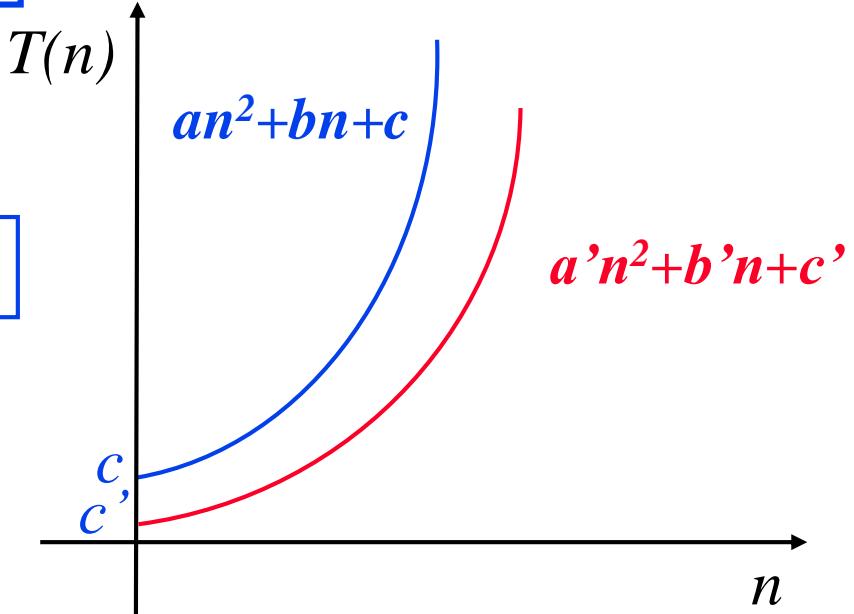
Analisi di Insert Sort: Caso medio

$$T(n) = c_1 n + c_2(n-1) + c_3(n-1) + c_4 \sum_{j=2}^n t_j + c_5 \sum_{j=2}^n (t_j - 1) + c_6 \sum_{j=2}^n (t_j - 1) + c_7(n-1)$$

$$\sum_{j=2}^n t_j = \sum_{j=2}^n \frac{j}{2} = \frac{1}{2} \left(\sum_{j=1}^n j - 1 \right) = \frac{n^2 + n - 2}{4}$$

$$\sum_{j=2}^n (t_j - 1) = \sum_{j=2}^n \left(\frac{j}{2} - 1 \right) = \frac{n^2 - 3n + 2}{4}$$

$$T(n) = a'n^2 + b'n + c'$$



Analisi del Caso Migliore e Caso Peggio

- **Analisi del Caso Migliore**
 - Ω -grande, limite inferiore, del tempo di esecuzione per un qualunque *input di dimensione N*.
- **Analisi del Caso Peggio**
 - O -grande, limite superiore, del tempo di esecuzione per un qualunque *input di dimensione N*.

Analisi del Caso Medio

- **Analisi del Caso Medio**
 - **Alcuni algoritmi sono efficienti in pratica.**
 - **L'analisi è in genere molto più difficile.**
 - **Bisogna generalmente assumere che tutti gli input siano ugualmente probabili.**
 - **A volte non è ovvio quale sia la media.**

Stima del limite asintotico superiore

- Nei prossimi lucidi vedremo un semplice metodo per *stimare il limite asintotico superiore* $O(\cdot)$ del tempo di esecuzione di *algoritmo iterativi*.
 - Stabilire il limite superiore per le operazioni elementari
 - Stabilire il limite superiore per le strutture di controllo
- Ci da un limite superiore che funge da stima, ma *non garantisce* di trovare la *funzione esatta* del *tempo di esecuzione*. La stima può essere a volte grossolana.

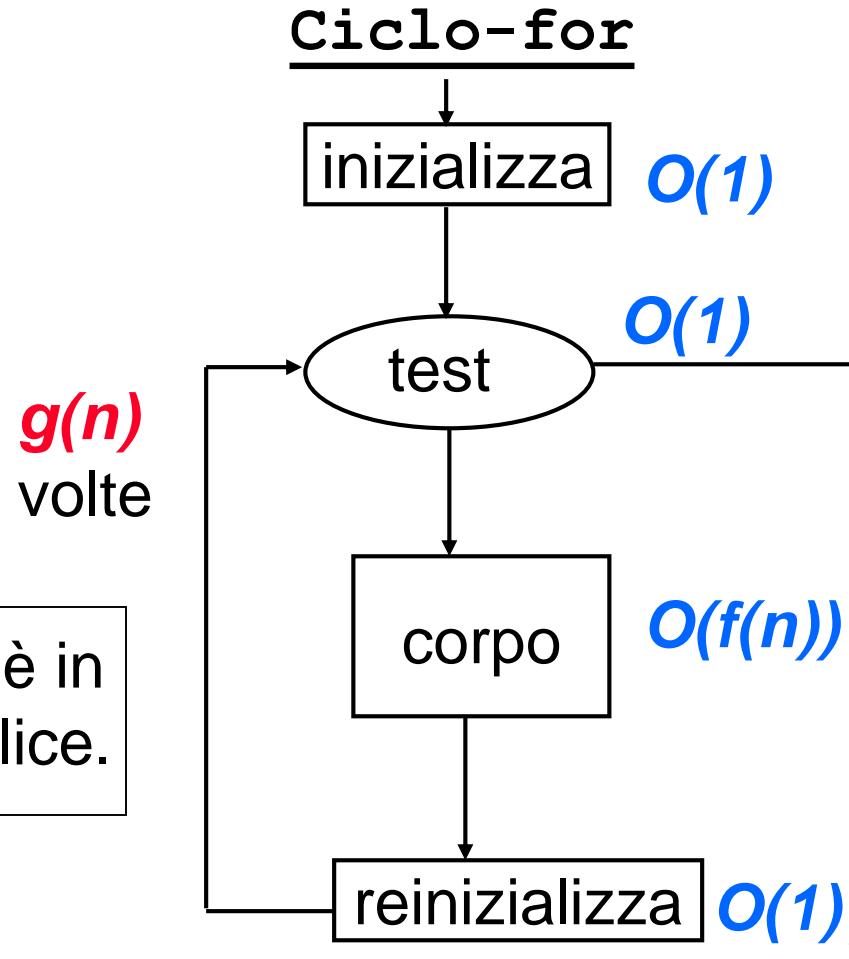
Tempo di esecuzione: operazioni semplici

Operazioni Semplici

- *operazioni aritmetiche* (+, *, ...)
- *operazioni logiche* (&&, ||,)
- *confronti* (≤ , ≥ , = ,...)
- *assegnamenti* (a = b) senza chiamate di funzione
- *operazioni di lettura* (read)
- *operazioni di controllo* (break, continue, return)

$$T(n) = \Theta(1) \Rightarrow T(n) = O(1)$$

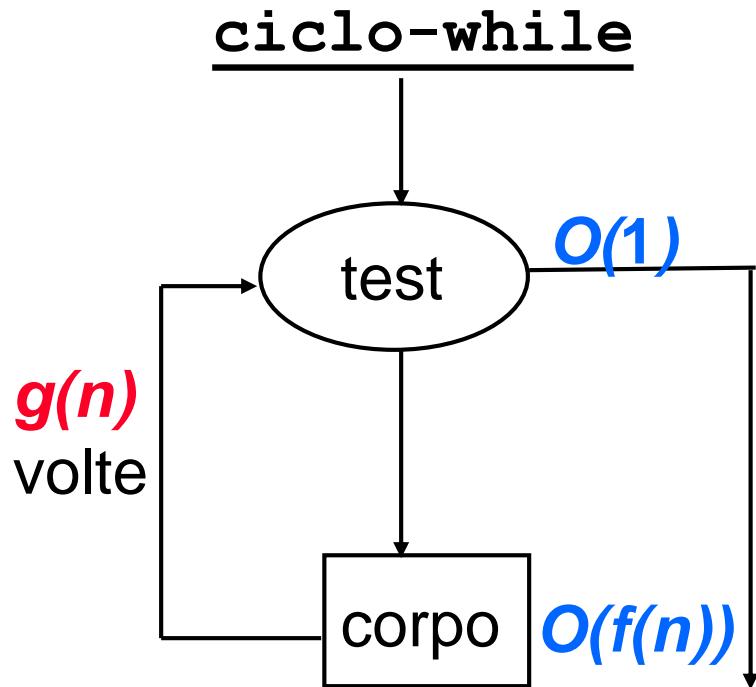
Tempo di esecuzione: ciclo for



stabilire $g(n)$ è in genere semplice.

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

Tempo di esecuzione: ciclo while



Bisogna stabilire un limite per il numero di iterazioni del ciclo, **$g(n)$** .

Può essere necessaria una prova induttiva per **$g(n)$** .

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

Ciclo while: esempio

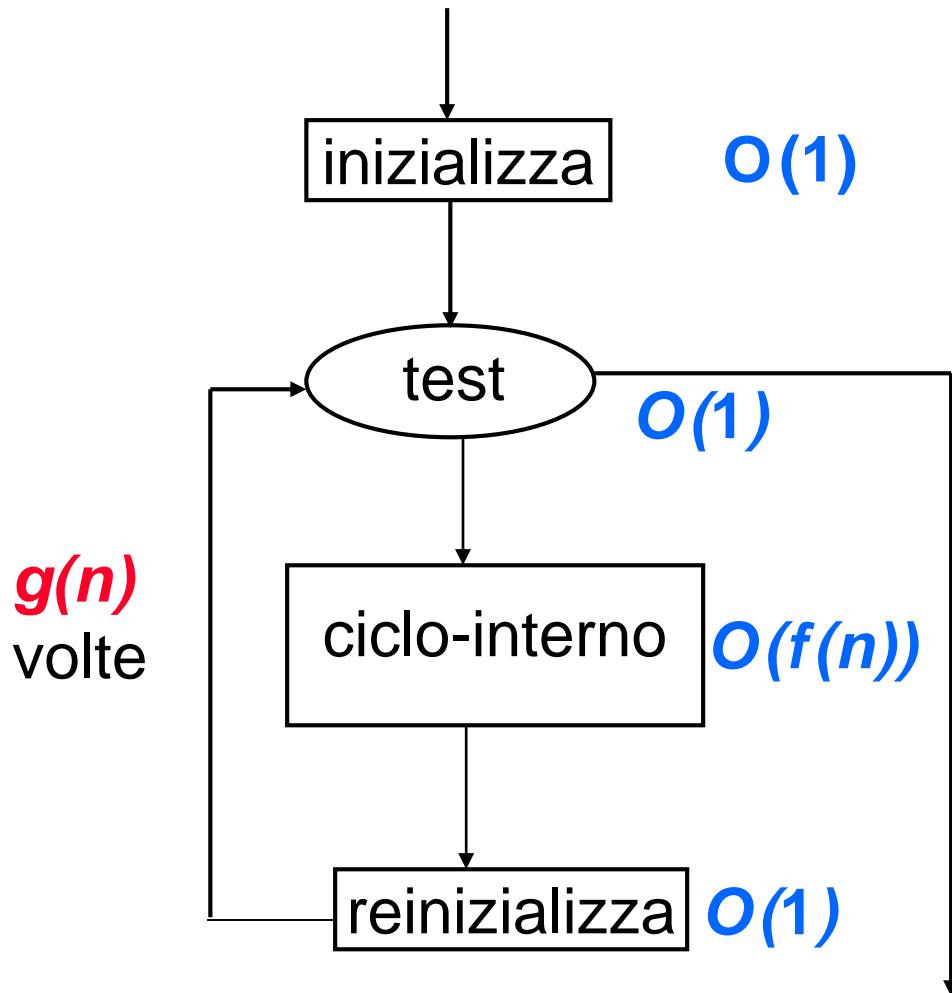
Ricerca dell'elemento x all'interno di un array $A[1...n]$:

```
i = 1                                (1)  
while (x ≠ A[i] && i≤n)           (2)  
    i = i+1                            (3)
```

(1)	$O(1)$
test in (2)	$O(1)$
(3)	$O(1)$
iterazioni	massimo $g(n) = n$

$$O(\text{ciclo-while}) = O(1) + n O(1) = O(n)$$

Tempo di esecuzione: cicli innestati



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

Cicli annidati: esempio

```
for i = 1 to n  
  for j = 1 to n  
    k = i + j
```

$$\left. \begin{array}{l} \text{for } i = 1 \text{ to } n \\ \text{ for } j = 1 \text{ to } n \\ \text{ } k = i + j \end{array} \right\} = O(n^2)$$

$$T(n) = O(n \times n) = O(n^2)$$

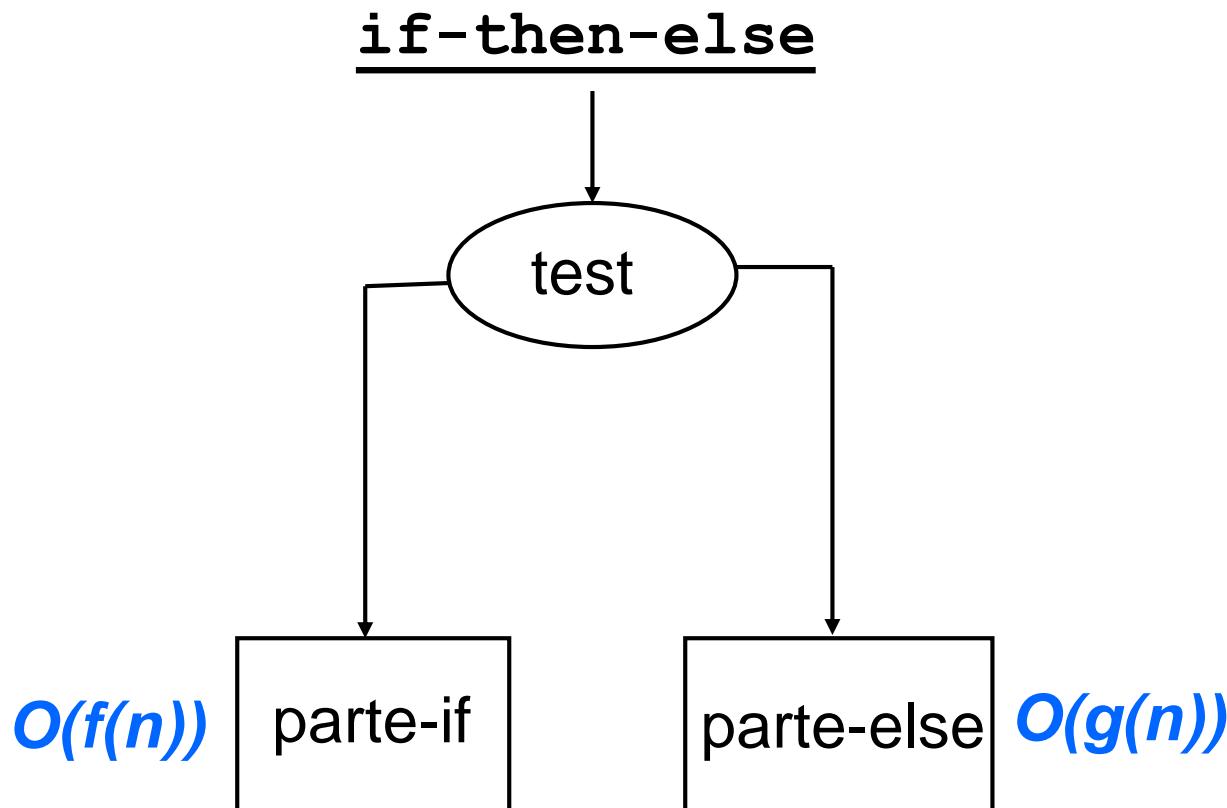
Cicli annidati: esempio

```
for i = 1 to n  
  for j = i to n  
    k = i + j
```

$$\left. \begin{array}{l} \\ \\ \end{array} \right\} = O(n - i) \quad \left. \begin{array}{l} \\ \\ \end{array} \right\} = O(n^2)$$

$$T(n) = O(n \times n) = O(n^2)$$

Tempo di esecuzione: If-Then-Else



$O(\max(f(n), g(n)))$

If-Then-Else: esempio

```
if A[1][i] = 0 then
    for i = 1 to n
        for j = 1 to n
            a[i][j] = 0
else
    for i = 1 to n
        A[i][i] = 1
```

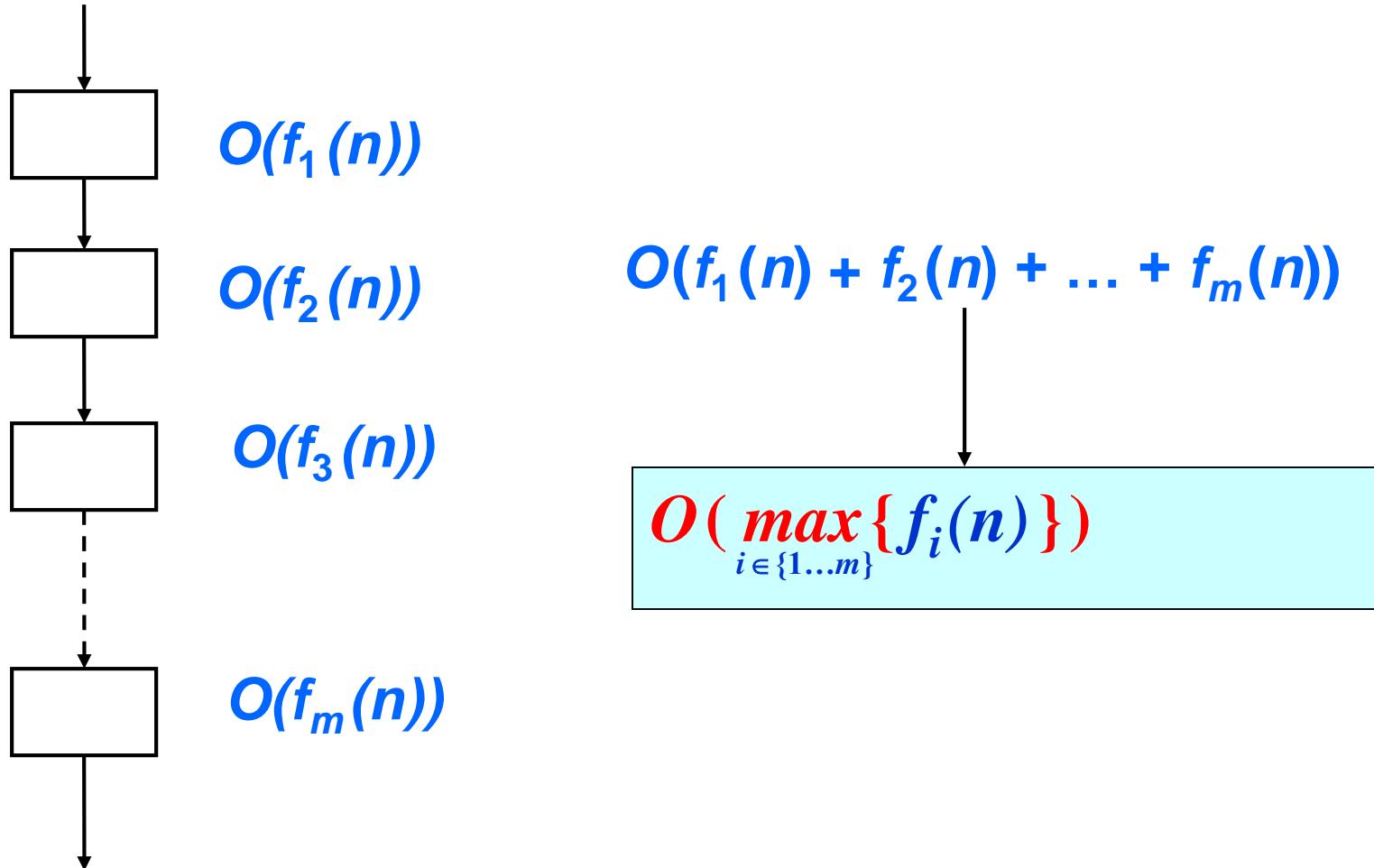
$= O(n) \} = O(n^2)$
 $= O(n) \}$

if: $T(n) = O(n^2)$

else : $T(n) = O(n)$

$$T(n) = \max(O(n^2), O(n)) = O(n^2)$$

Tempo di esecuzione: blocchi sequenziali



Blocchi sequenziali: esempio

```
for i = 1 to n } = O(n)
  A[1] = 0
for i = 1 to n
  for j = 1 to n } = O(n) } = O(n2)
    A[i] = A[i] + A[i]
```

$$\begin{aligned} T(n) &= O(\max(f(\text{ciclo-1}), f(\text{ciclo-2})) \\ &= O(\max(n, n^2)) \\ &= O(n^2) \end{aligned}$$

Esempio: Insert Sort

```
InsertSort(array A[1...n])
```

```
    for j = 2 to n
```

```
        key = A[j]
```

```
        i = j - 1
```

```
        while i > 0 and A[i] > key
```

```
            A[i+1] = A[i]
```

```
            i = i - 1
```

```
        A[i+1] = key
```

$= O(1)$

$= O(1)$

$\} = O(n)$

$= O(1)$

$$T(n) = O(g(n) \times \max(1, 1, n, 1))$$

$$= O(n \times n)$$

$$= O(n^2)$$

Tecniche di sviluppo di algoritmi

- Agli esempi visti fino ad ora seguono l'*approccio incrementale*: la soluzione viene costruita passo dopo passo.
- *Insert sort* avendo ordinato una sottoparte dell'array, inserisce al posto giusto un altro elemento ottenendo un sotto-array ordinato più grande.
- Esistono altre tecniche di sviluppo di algoritmi con filosofie differenti:
 - **Divide-et-Impera**

Divide-et-Impera

- Il problema viene suddiviso in sottoproblemi analoghi, che vengono risolti separatamente. Le soluzioni dei sottoproblemi vengono infine fuse insieme per ottenere la soluzione dei problemi più complessi.
- Consiste di 3 passi:
 - *Divide* il problema in vari sottoproblemi, tutti *simili* (tra loro e) al *problema originario* ma più semplici.
 - *Impera* (conquista) i sottoproblemi risolvendoli ricorsivamente. Quando un sottoproblema diviene banale, risolverlo direttamente.
 - *Fondi* le soluzioni dei sottoproblemi per ottenere la soluzione del (sotto)problema che li ha originati.

Divide-et-Impera e ordinamento

- **Input:** una sequenza di numeri.
- **Output:** una permutazione (riordinamento) tale che tra ogni 2 elementi adiacenti nella sequenza valga “qualche” relazione di ordinamento (ad es. \leq).
- *Merge Sort* (divide-et-impera)
 - *Divide:* scomponere la sequenza di n elementi in 2 sottosequenze di $n/2$ elementi ciascuna.
 - *Impera:* conquista i sottoproblemi ordinando ricorsivamente le sottosequenze con *Merge Sort* stesso. Quando una sottosequenza è unitaria, il sottoproblema è banale.
 - *Fondi:* compone insieme le soluzioni dei sottoproblemi per ottenere la sequenza ordinata del (sotto-)problema.

Merge Sort

Algoritmo :

- $A[1..n]$: sequenza dei numeri in input
- p,r : indici degli estremi della sottosequenza da ordinare

```
Merge_Sort(array A, int p,r)
```

```
1 if p < r then
```

```
2     q = ⌊(p+r)/2⌋
```

```
3         Merge_Sort(A,p,q)
```

```
4         Merge_Sort(A,q+1,r)
```

```
5     Merge(A,p,q,r)
```

Divide

Impera

Combina

Esercizio: definire la procedura Merge

Merge Sort: analisi

```
Merge_Sort(array A, int p,r)
```

```
1 if p < r then
```

```
2     q = ⌊(p+r)/2⌋
```

```
3         Merge_Sort(A,p,q)
```

```
4         Merge_Sort(A,q+1,r)
```

```
5     Merge(A,p,q,r)
```

$$T(n) = \Theta(1) \text{ se } n=1$$

$$T(n) = 2 T(n/2) + T_{\text{merge}}(n) + \Theta(1)$$

Equazione di Ricorrenza

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

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

Merge Sort: analisi

```
Merge_Sort(array A, int p,r)  
1  if p < r then  
2      q = ⌊(p+r)/2⌋  
3      Merge_Sort(A,p,q)  
4      Merge_Sort(A,q+1,r)  
5      Merge(A,p,q,r)
```

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

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

Divide-et-Impera: Equazioni di ricorrenza

- **Divide:** $D(n)$ tempo per dividere il problema
- **Impera:** se si divide il problema in a sottoproblemi, ciascuno di dimensione n/b , il tempo per conquistare i sottoproblemi sarà $aT(n/b)$.

Quando un sottoproblema diviene banale (*l'input è minore o uguale ad una costante c*), in tempo è $\Theta(1)$.

- **Fondi:** $C(n)$ tempo per comporre le soluzioni dei sottoproblemi nella soluzione più complessa.

$$T(n) = \begin{cases} \Theta(1) & \text{se } n \leq c \\ aT(n/b) + D(n) + C(n) & \text{se } n > c \end{cases}$$

Gli argomenti trattati

- Analisi della bontà di un algoritmo
 - Correttezza, utilizzo delle risorse, semplicità
- Modello computazionali: **modello RAM**
- **Tempo di esecuzione** degli algoritmi
- Notazione asintotica: **O -grande**, **Ω -grande**, **Θ**
- Analisi del **Caso Migliore**, **Caso Peggiore** e del **Caso Media**

Algoritmi e Strutture Dati

**Valutazione del tempo di esecuzione
degli algoritmi**

Stima del limite asintotico superiore

- Nelle prossime lucidi definiremo un semplice metodo per **stimare il limite asintotico superiore $O(.)$** del tempo di esecuzione di **algoritmo iterativi**.
 - Stabilire il limite superiore per le operazioni elementari
 - Stabilire il limite superiore per le strutture di controllo
- Ci da un limite superiore che funge da stima, **non garantisce** di trovare la **funzione precisa del tempo di esecuzione**.

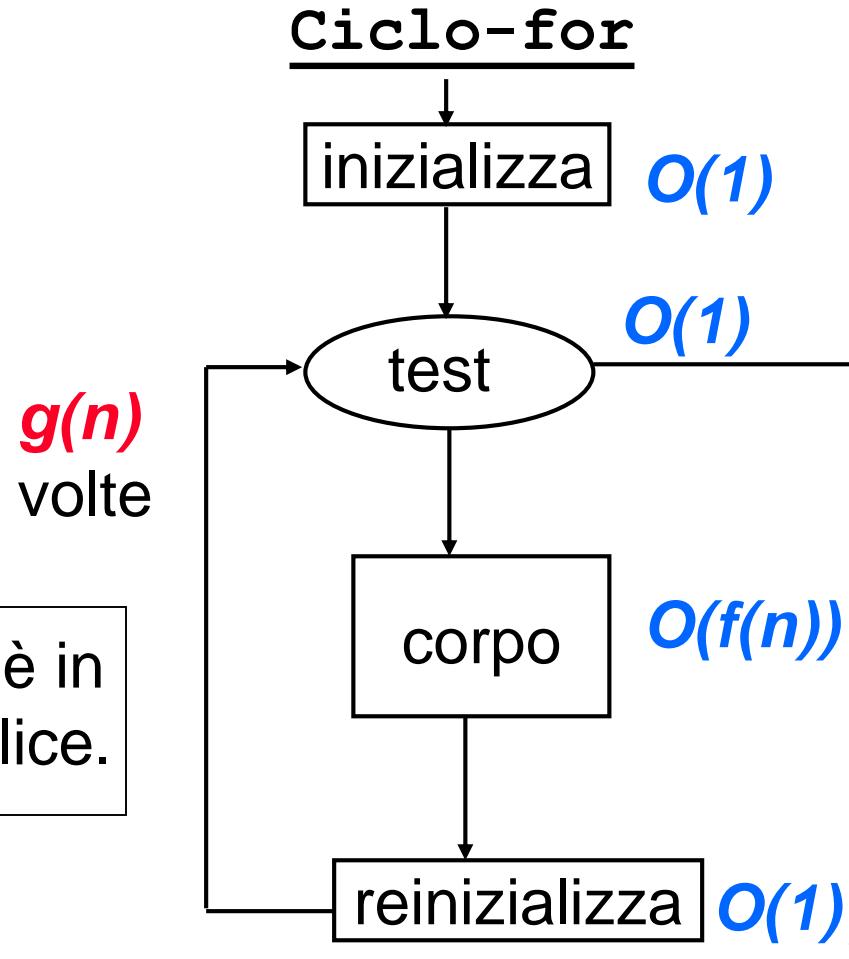
Tempo di esecuzione: operazioni semplici

Operazioni Semplici

- *operazioni aritmetiche* (+, *, ...)
- *operazioni logiche* (&&, ||, ...)
- *confronti* (\leq , \geq , = , ...)
- **assegnamenti** (a = b) senza chiamate di funzione
- *operazioni di lettura* (read)
- *operazioni di controllo* (break, continue, return)

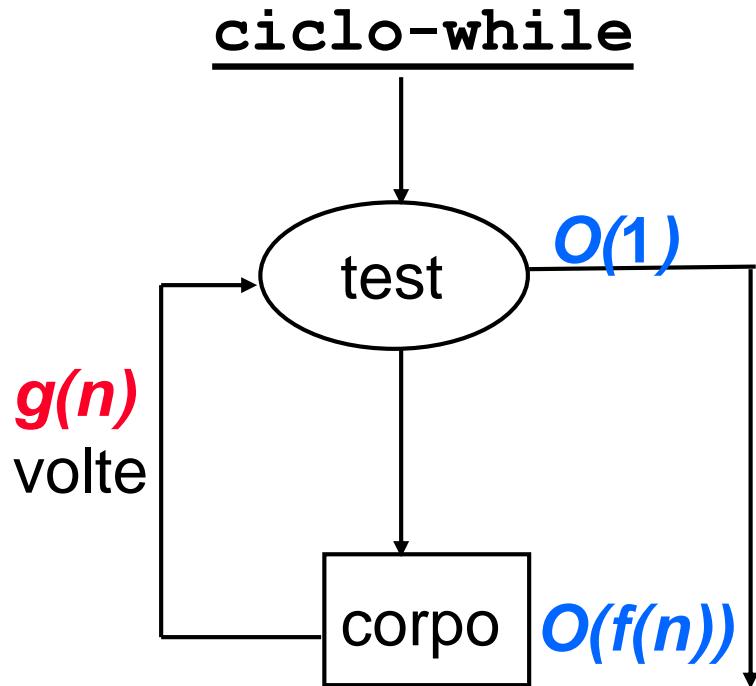
$$T(n) = \Theta(1) \Rightarrow T(n) = O(1)$$

Tempo di esecuzione: ciclo for



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

Tempo di esecuzione: ciclo while



Bisogna stabilire un limite per il numero di iterazioni del ciclo, **$g(n)$** .

Può essere necessaria una prova induttiva per **$g(n)$** .

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

Ciclo while: esempio

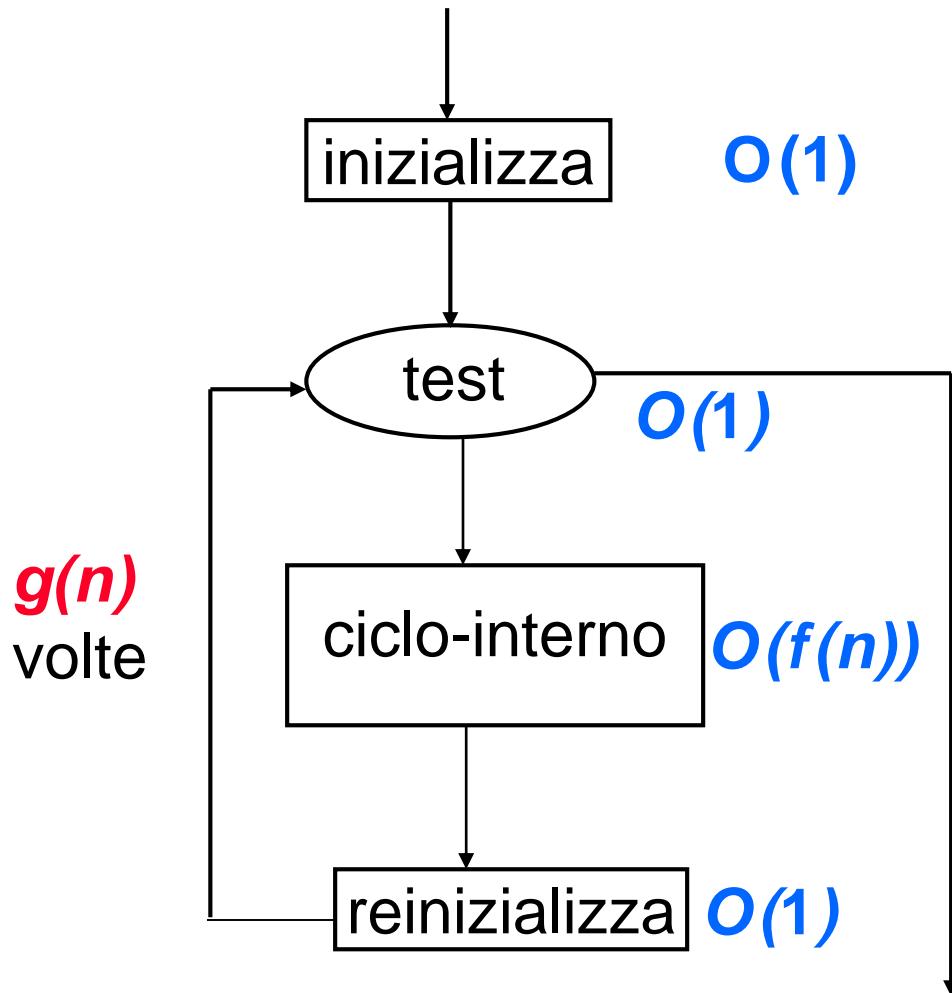
Ricerca dell'elemento x all'interno di un array $A[1...n]$:

```
i = 1                                (1)  
while (x ≠ A[i] && i≤n)           (2)  
    i = i+1                            (3)
```

(1)	$O(1)$
test in (2)	$O(1)$
(3)	$O(1)$
iterazioni	massimo n

$$O(\text{ciclo-while}) = O(1) + n O(1) = O(n)$$

Tempo di esecuzione: cicli innestati



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

Cicli annidati: esempio

```
for i = 1 to n  
  for j = 1 to n  
    k = i + j
```

$$\left. \begin{array}{l} \text{for } i = 1 \text{ to } n \\ \text{ for } j = 1 \text{ to } n \\ \text{ } k = i + j \end{array} \right\} = O(n^2)$$

$$T(n) = O(n \times n) = O(n^2)$$

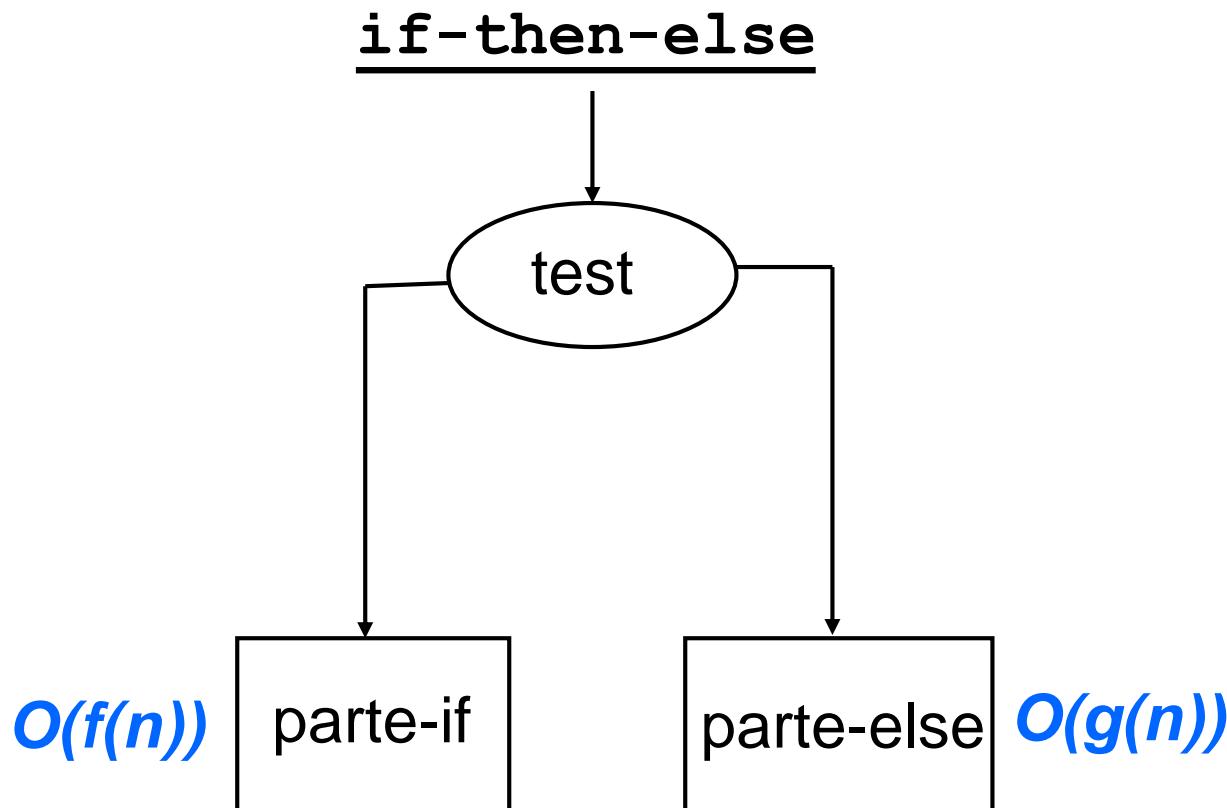
Cicli annidati: esempio

```
for i = 1 to n  
  for j = i to n  
    k = i + j
```

$$\left. \begin{array}{l} \\ \\ \end{array} \right\} = O(n - i) \quad \left. \begin{array}{l} \\ \\ \end{array} \right\} = O(n^2)$$

$$T(n) = O(n \times n) = O(n^2)$$

Tempo di esecuzione: If-Then-Else



$$O(\max(f(n), g(n)))$$

If-Then-Else: esempio

```
if A[1][i] = 0 then
    for i = 1 to n
        for j = 1 to n
            a[i][j] = 0
else
    for i = 1 to n
        A[i][i] = 1
```

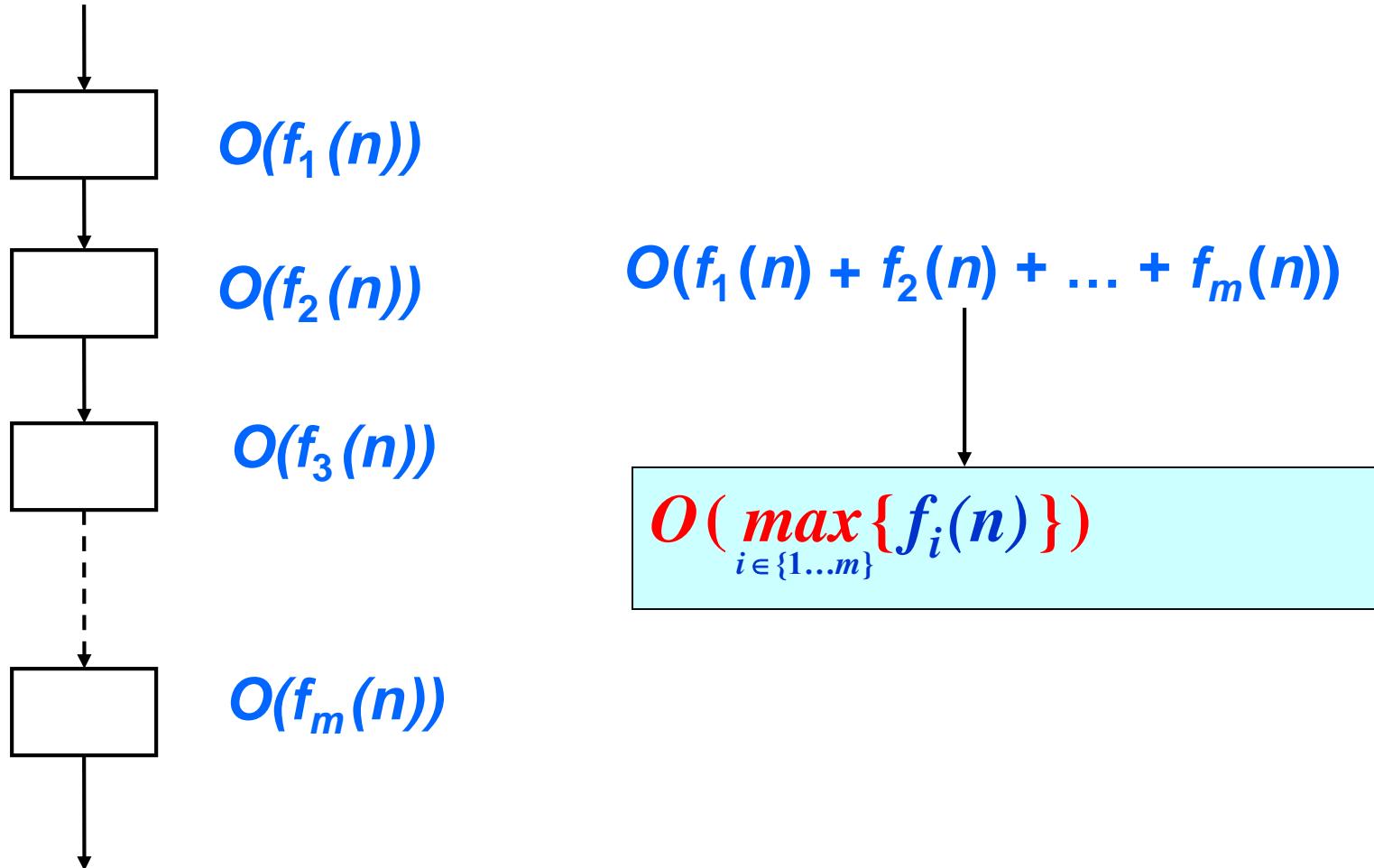
$\left. \begin{array}{l} \\ \\ \end{array} \right\} = O(n^2)$
 $\left. \begin{array}{l} \\ \end{array} \right\} = O(n)$

if: $T(n) = O(n^2)$

else : $T(n) = O(n)$

$$T(n) = \max(O(n^2), O(n)) = O(n^2)$$

Tempo di esecuzione: blocchi sequenziali



Blocchi sequenziali: esempio

```
for i = 1 to n  
  A[1] = 0
```

$\} = O(n)$

```
for i = 1 to n  
  for j = 1 to n  
    A[i] = A[i] + A[i]
```

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

$$\begin{aligned} T(n) &= O(\max(f(\text{ciclo-1}), f(\text{ciclo-2})) \\ &= O(\max(n, n^2)) \\ &= O(n^2) \end{aligned}$$

Esempio: Insert Sort

```
InsertSort(array A[1...n])
```

$$O(n^2) = \left\{ \begin{array}{ll} \text{for } j = 2 \text{ to } n & \\ \quad \text{key} = A[j] & = O(1) \\ \quad i = j - 1 & = O(1) \\ \quad \text{while } i > 0 \text{ and } A[i] > \text{key} & \\ \quad \quad A[i+1] = A[i] & \\ \quad \quad i = i - 1 & \\ \quad A[i+1] = \text{key} & = O(1) \end{array} \right\} = O(n)$$

$$\begin{aligned} T(n) &= O(g(n) \times \max(1, 1, n, 1)) \\ &= O(n \times n) \\ &= O(n^2) \end{aligned}$$

Tempo di esecuzione di algoritmi ricorsivi

- *E per gli algoritmi ricorsivi?*
 - Il tempo di esecuzione è espresso tramite una equazione di ricorrenza.

Esempio:

$$\text{Merge Sort: } T(n) = \begin{cases} \Theta(1) & \text{se } n = 1 \\ 2T(n/2) + \Theta(n) & \text{se } n > 1 \end{cases}$$

- Sono necessarie tecniche specifiche per risolvere le equazioni di ricorrenza

Algoritmi e Strutture Dati

Tempo di esecuzione di algoritmi ricorsivi

Tempo di esecuzione per algoritmi ricorsivi

Esempio: Fattoriale

```
fact(int n)
    if n <= 1 then
        return 1                      /* Caso Base
    else
        return n*fact(n-1) /* Passo Induttivo
```

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

Soluzione di equazioni di ricorrenza

- **Esistono molto metodi.** Ne mostreremo tre:

➤ **Il Metodo Iterativo**

- Si itera la regola induttiva di $T(n)$ in termini di n e del caso base.
- Richiede manipolazione delle somme

➤ **Il Metodo di Sostituzione**

- Si ipotizza una possibile soluzione
- Si sostituisce l'ipotetica soluzione nei casi base e induttivo
- Si dimostra la correttezza della ipotesi tramite induzione matematica

✗ **Il Metodo Principale**

II Metodo Iterativo

Base: $T(1) = a$

Induzione: $T(n) = b + T(n-1)$

I. Sostituire ad m i valori $n, n-1, n-2 \dots$ finché si ottiene il caso base

1) $T(n) = b + T(n-1)$ sostituire m con n

2) $T(n-1) = b + T(n-2)$ sostituire m con $n-1$

3) $T(n-2) = b + T(n-3)$ sostituire m con $n-2$

.....

$n-1).$ $T(2) = b + T(1)$ sostituire m con 2

$T(1) = a$ noto

II Metodo Iterativo

II. Sostituire $T(n-1)$, $T(n-2)...$ fino al caso base e sostituirlo.

$$\begin{aligned} T(n) &= b + T(n-1) &= \\ &= b + b + T(n-2) &= 2*b + T(n-2) = \\ &= b + b + b + T(n-3) &= 3*b + T(n-3) = \\ &= b + b + b + b + T(n-4) &= 4*b + T(n-4) = \end{aligned}$$

.....

$$T(n) = \sum_{i=1}^{n-1} b + T(1) = (n-1) \cdot b + T(1)$$

Inserire il caso base

$$T(n) = (n-1) \cdot b + a$$

III. Valutare l'espressione O -grande associata

$$T(n) = b*n - b + a = O(n)$$

Il Metodo iterativo: Fattoriale

Esempio: Fattoriale

```
fact(int n)
    if n <= 1 then
        return 1                  /* Caso Base
    else
        return n*fact(n-1) /* Passo Induttivo
```

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

Equazione di ricorrenza

Base: $T(1) = a$

Induzione: $T(m) = b + T(m-1)$

Il Metodo iterativo: Fattoriale

Analisi di fact

Caso Base:

$$T(0) = O(1),$$

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

Passo Induttivo:

$$O(1) + \max(O(1), T(n-1))$$

$$O(1) + T(n-1), \text{ per } n > 1$$

Per il fattoriale, l'analisi risulta

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

II Metodo iterativo: esempio

$$T(n) = 3 T(n/4) + n$$

II Metodo iterativo: esempio

$$\begin{aligned}T(n) &= 3 T(n/4) + n = \\&= 3(3 T(n/16) + n/4) + n\end{aligned}$$

II Metodo iterativo: esempio

$$\begin{aligned}T(n) &= 3 T(n/4) + n = \\&= 3(3 T(n/16) + n/4) + n = \\&= 9 T(n/16) + 3n/4 + n\end{aligned}$$

II Metodo iterativo: esempio

$$\begin{aligned}T(n) &= 3 T(n/4) + n = \\&= 3(3 T(n/16) + n/4) + n = \\&= 9 T(n/16) + 3n/4 + n = \\&= 27 T(n/64) + 9n/16 + 3n/4 + n\end{aligned}$$

II Metodo iterativo: esempio

$$\begin{aligned}T(n) &= 3 T(n/4) + n = \\&= 3(3 T(n/16) + n/4) + n = \\&= 9 T(n/16) + 3n/4 + n = \\&= 27 T(n/64) + 9n/16 + 3n/4 + n =\end{aligned}$$

....

Quando ci si ferma?

II Metodo iterativo: esempio

$$\begin{aligned}T(n) &= 3T(n/4) + n = \\&= 3(3T(n/16) + n/4) + n = \\&= 9T(n/16) + 3n/4 + n = \\&= 27T(n/64) + 9n/16 + 3n/4 + n =\end{aligned}$$

....

Quando ci si ferma?

quando $n/(4^i) = 1$

cioè quando $i > \log_4 n$

$$T(n) < n + 3n/4 + 9n/16 + 27T(n/64) + \dots + 3^{\log_4 n} \Theta(1)$$

II Metodo iterativo: esempio

$$T(n) < n + 3n/4 + 9n/16 + 27T(n/64) + \dots + 3^{\log_4 n} \Theta(1)$$

Contiene una serie geometrica, che è del tipo

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

$$\begin{aligned} T(n) &< n + 3n/4 + 9n/16 + 27T(n/64) + \dots + 3^{\log_4 n} \Theta(1) \\ &\leq n \sum_{i=0}^{\infty} (3/4)^i + \Theta(n^{\log_4 3}) \end{aligned}$$

$$3^{\log_4 n} = n^{\log_4 3}$$

II Metodo iterativo: esempio

$$T(n) < n + 3n/4 + 9n/16 + 27T(n/64) + \dots + 3^{\log_4 n} \Theta(1)$$

Contiene una serie geometrica, che è del tipo

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

quando $|x| < 1$ converge a

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

$$\sum_{i=0}^{\infty} (3/4)^i = \frac{1}{1-3/4} = 4$$

$$T(n) < n + 3n/4 + 9n/16 + 27T(n/64) + \dots + 3^{\log_4 n} \Theta(1)$$

$$\leq n \sum_{i=0}^{\infty} (3/4)^i + \Theta(n^{\log_4 3})$$

$$3^{\log_4 n} = n^{\log_4 3}$$

II Metodo iterativo: esempio

$$T(n) < n + 3n/4 + 9n/16 + 27T(n/64) + \dots + 3^{\log_4 n} \Theta(1)$$

Contiene una serie geometrica, che è del tipo

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

quando $|x| < 1$ converge a

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

$$\sum_{i=0}^{\infty} (3/4)^i = \frac{1}{1-3/4} = 4$$

$$T(n) < n + 3n/4 + 9n/16 + 27T(n/64) + \dots + 3^{\log_4 n} \Theta(1)$$

$$\leq n \sum_{i=0}^{\infty} (3/4)^i + \Theta(n^{\log_4 3}) = 4n + o(n)$$

$$3^{\log_4 n} = n^{\log_4 3} \text{ e } \log_4 3 < 1$$

II Metodo iterativo: esempio

$$T(n) < n + 3n/4 + 9n/16 + 27T(n/64) + \dots + 3^{\log_4 n} \Theta(1)$$

Contiene una serie geometrica, che è del tipo

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

quando $|x| < 1$ converge a

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

$$\sum_{i=0}^{\infty} (3/4)^i = \frac{1}{1-3/4} = 4$$

$$T(n) < n + 3n/4 + 9n/16 + 27T(n/64) + \dots + 3^{\log_4 n} \Theta(1)$$

$$\begin{aligned} &\leq n \sum_{i=0}^{\infty} (3/4)^i + \Theta(n^{\log_4 3}) = 4n + o(n) \\ &= O(n) \end{aligned}$$

$$3^{\log_4 n} = n^{\log_4 3} \text{ e } \log_4 3 < 1$$

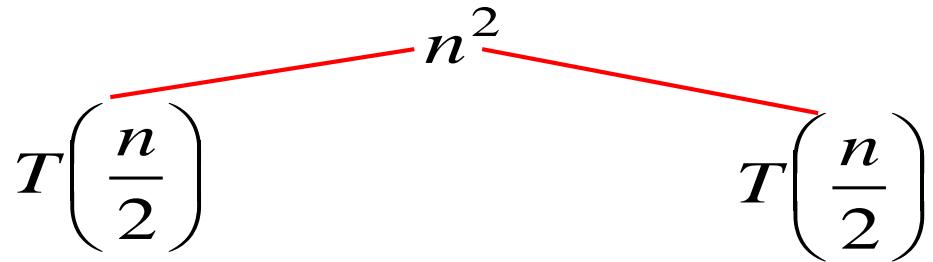
Metodo iterativo: alberi di Ricorrenza

Gli alberi di ricorrenza rappresentano un modo conveniente per visualizzare i passi di sostituzione necessari per risolvere una ricorrenza col **Metodo Iterativo**.

- Utili per semplificare i calcoli ed evidenziare le **condizioni limite** della ricorrenza.

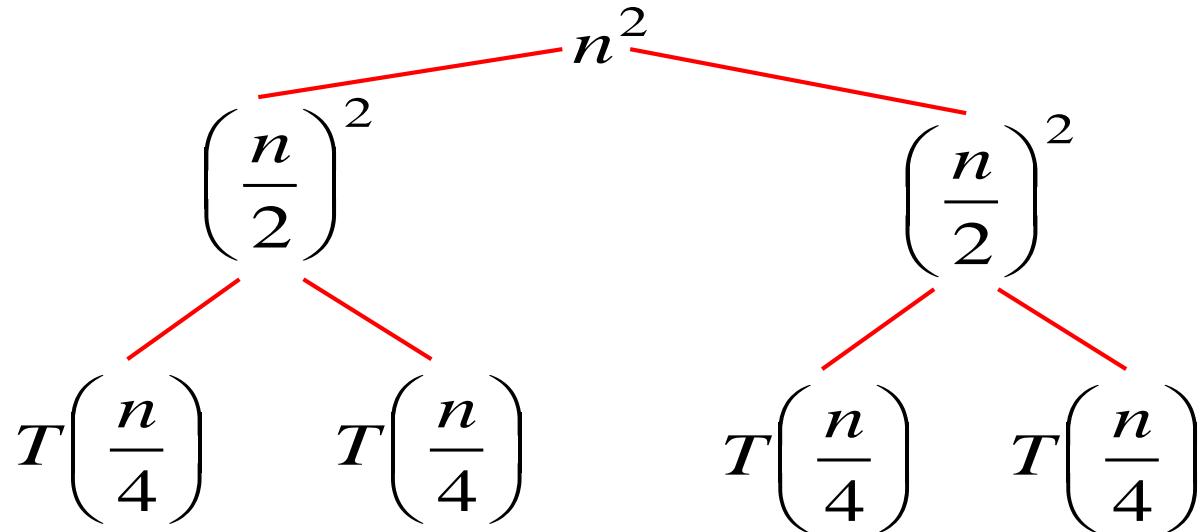
Metodo iterativo: alberi di Ricorrenza

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



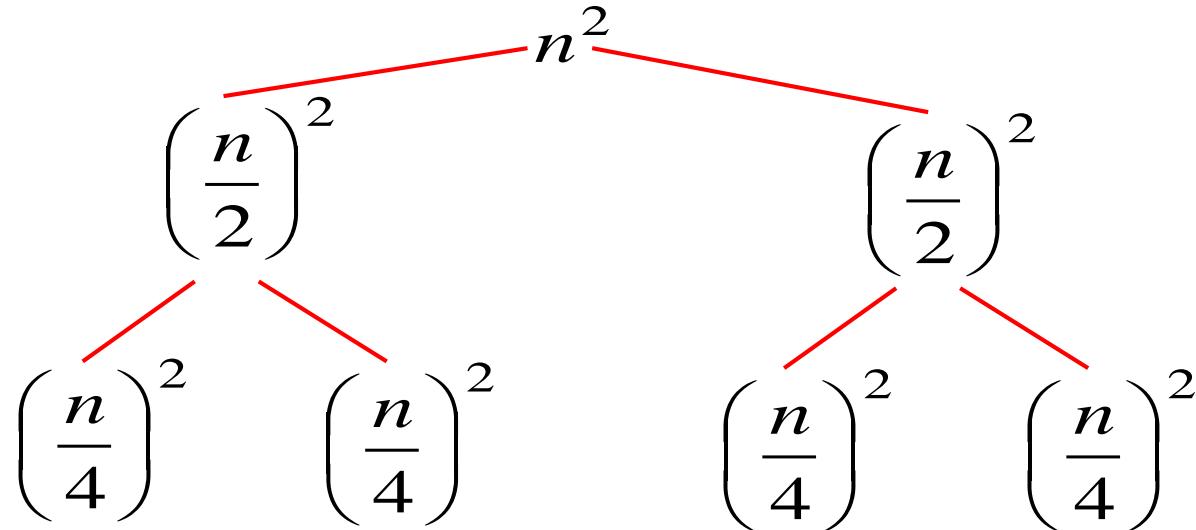
Metodo iterativo: alberi di Ricorrenza

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



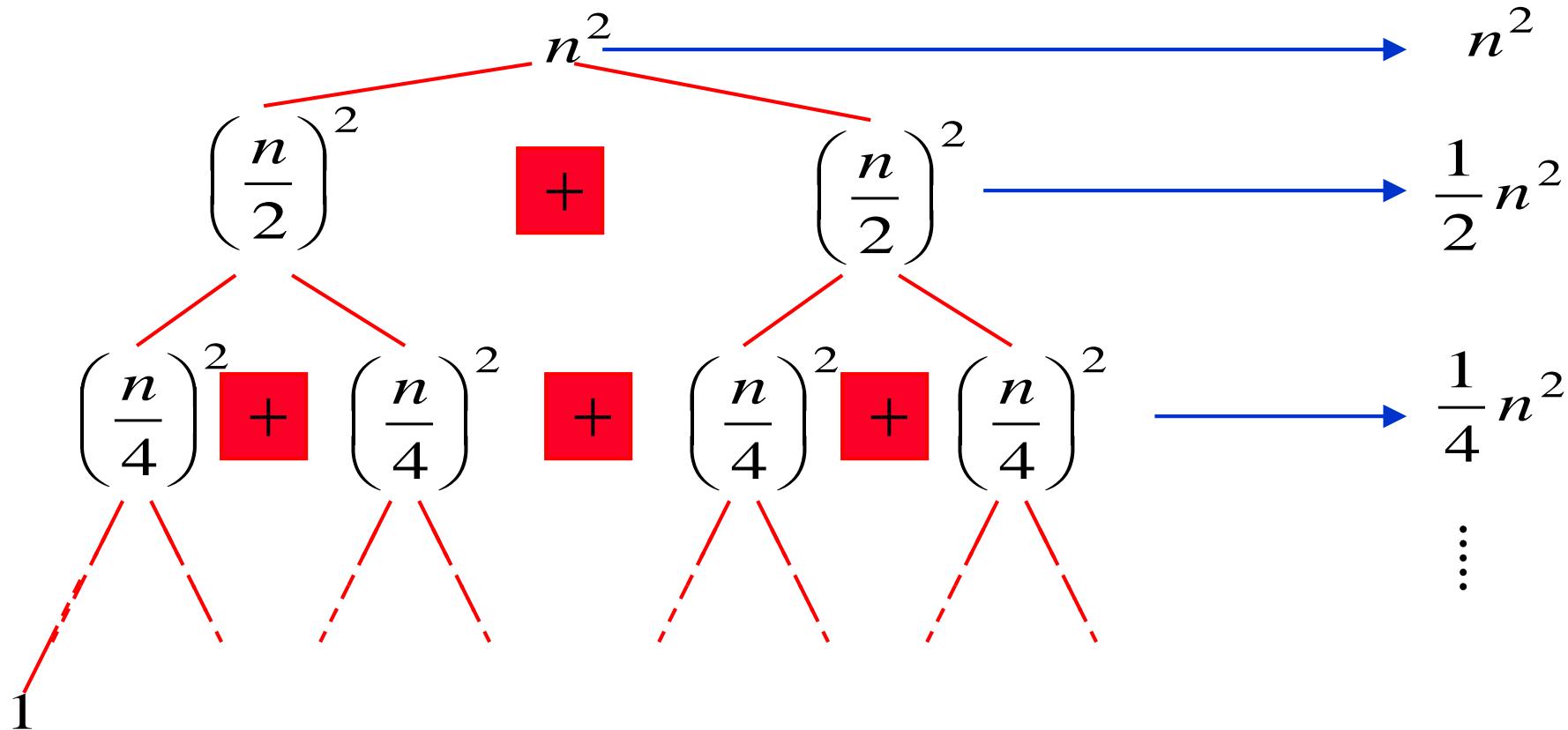
Metodo iterativo: alberi di Ricorrenza

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



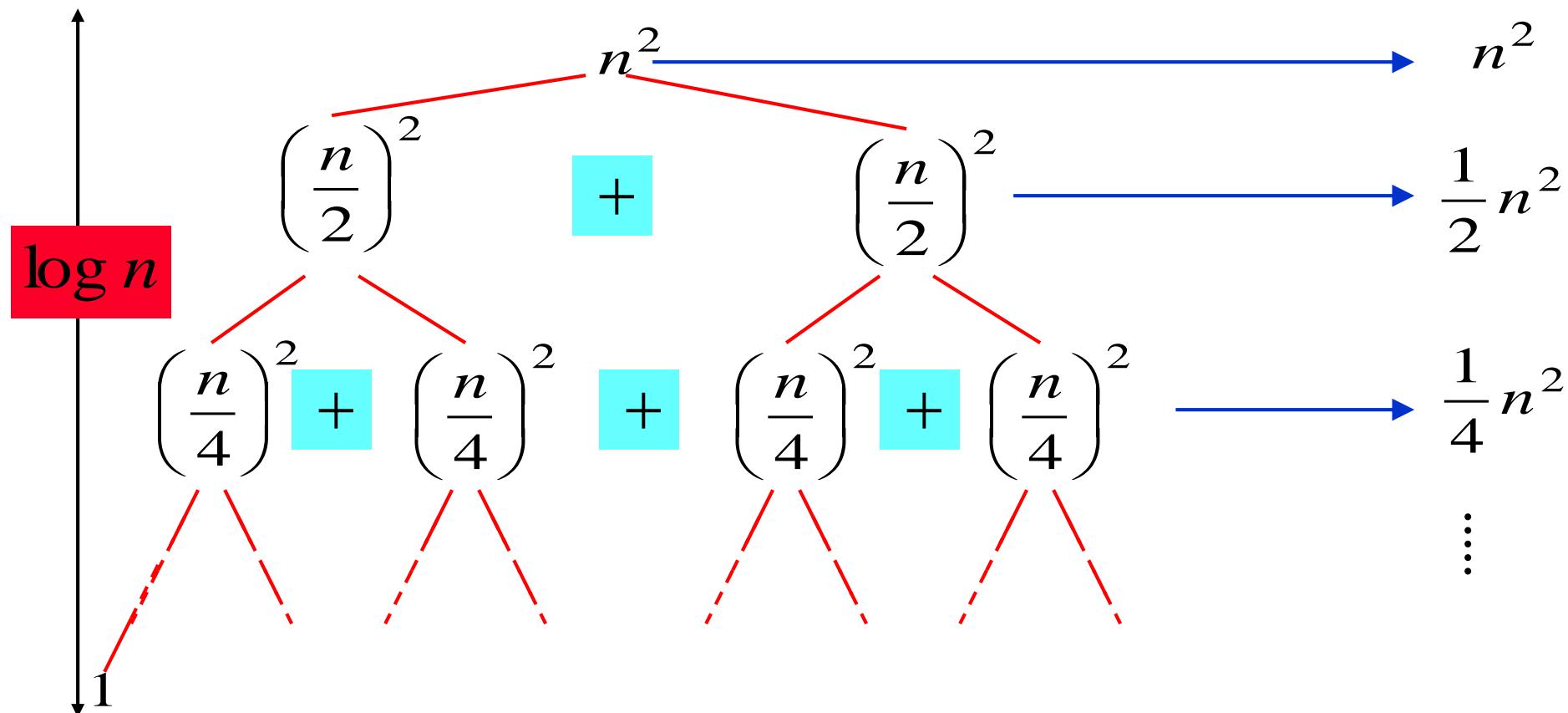
Metodo iterativo: alberi di Ricorrenza

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



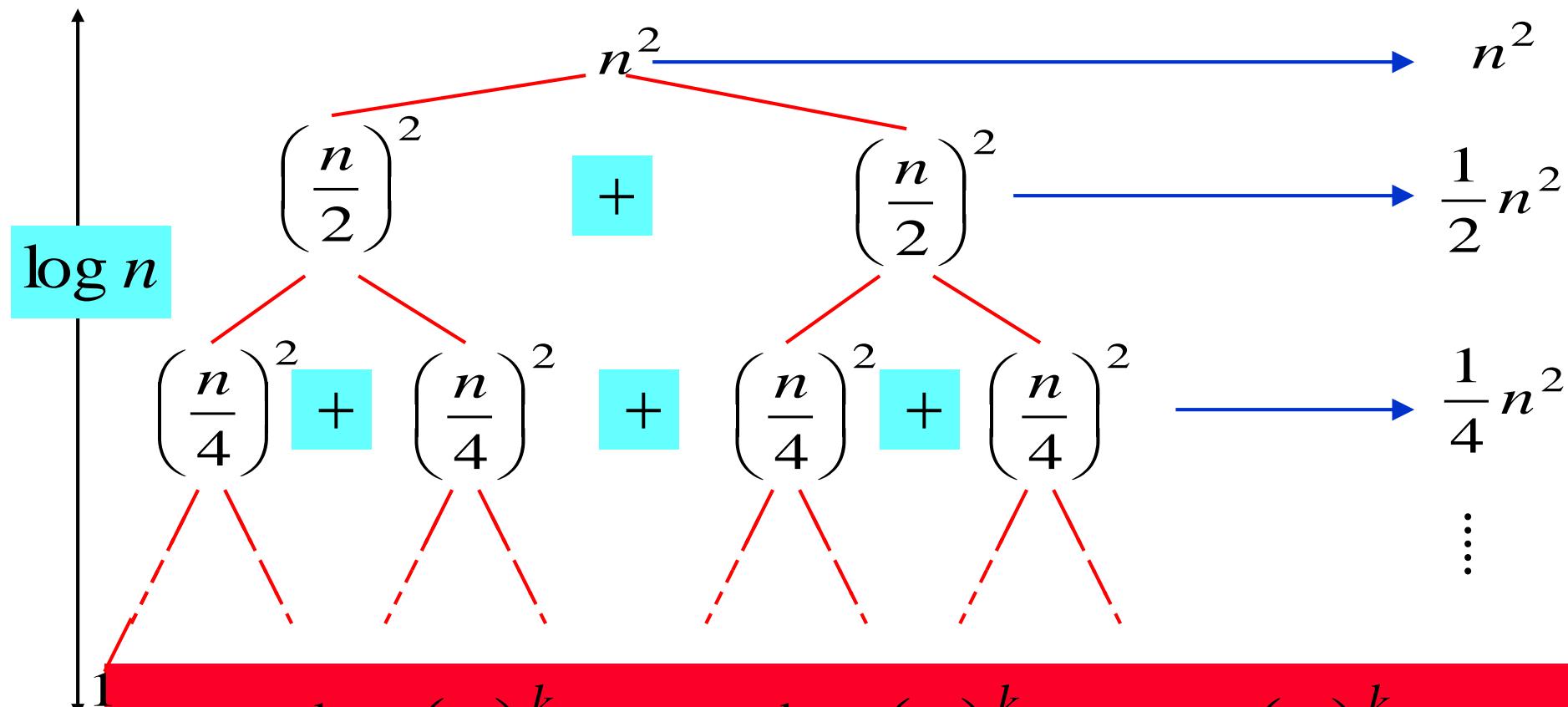
Metodo iterativo: alberi di Ricorrenza

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



Metodo iterativo: alberi di Ricorrenza

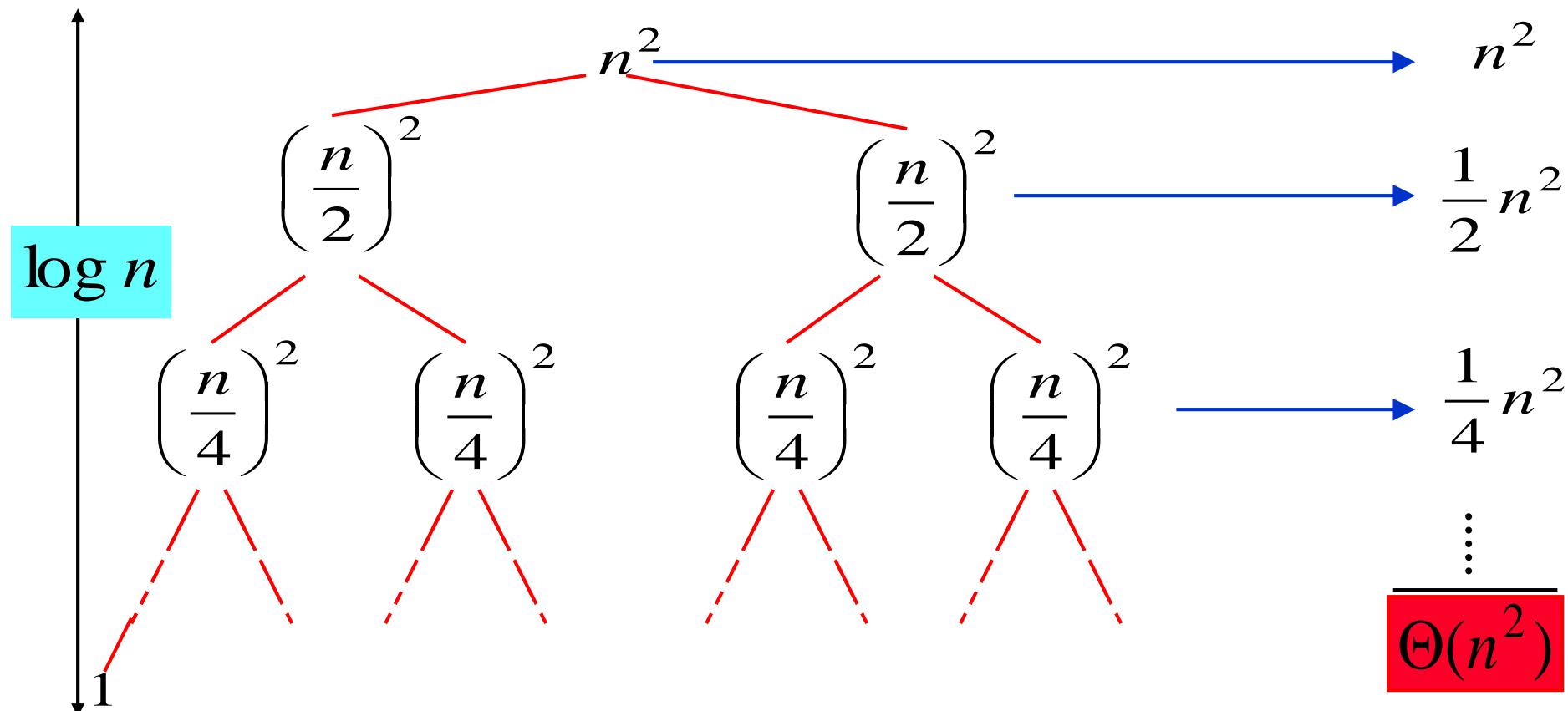
Esempio: $T(n) = 2T(n/2) + n^2$



$$T(n) = \sum_{k=0}^{\log n} \left(\frac{1}{2}\right)^k n^2 = n^2 \sum_{k=0}^{\log n} \left(\frac{1}{2}\right)^k \leq n^2 \sum_{k=0}^{\infty} \left(\frac{1}{2}\right)^k = 2n^2$$

Metodo iterativo: alberi di Ricorrenza

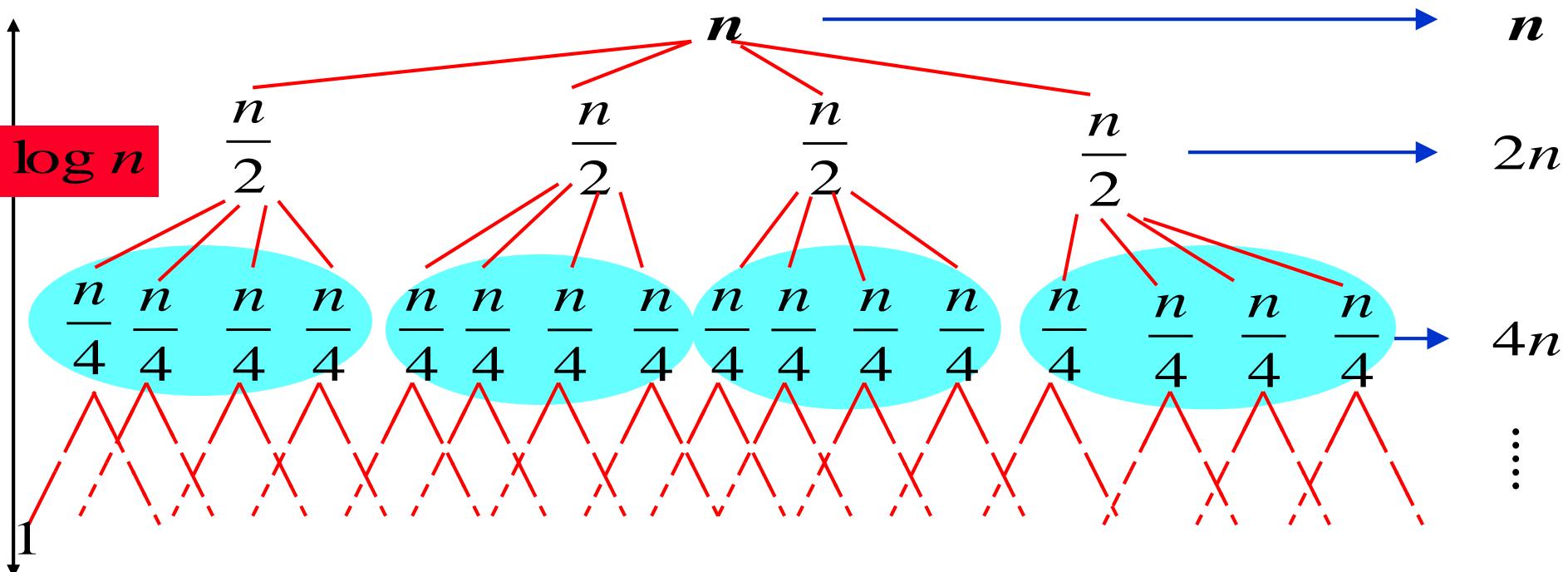
Esempio: $T(n) = 2T(n/2) + n^2$



$$T(n) = \sum_{k=0}^{\log n} \left(\frac{1}{2}\right)^k n^2 = n^2 \sum_{k=0}^{\log n} \left(\frac{1}{2}\right)^k \leq n^2 \sum_{k=0}^{\infty} \left(\frac{1}{2}\right)^k = 2n^2$$

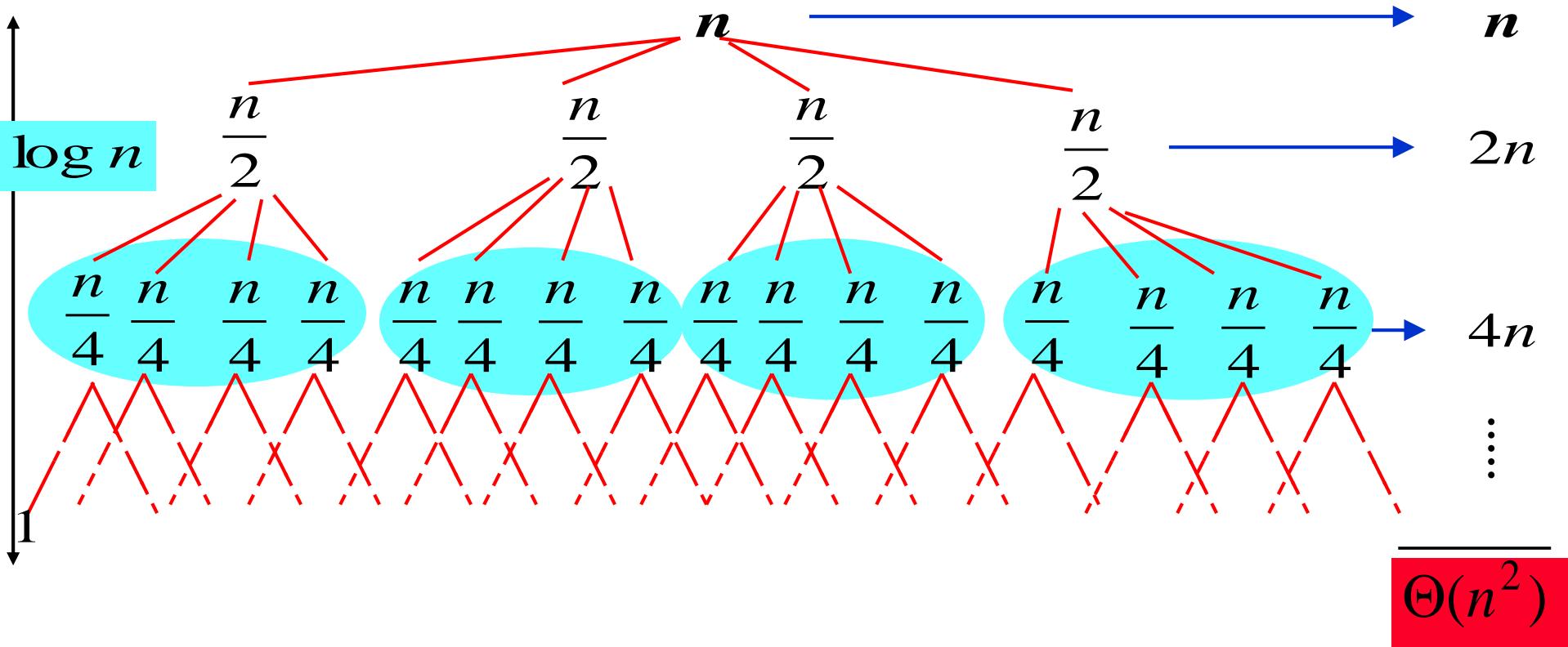
Metodo iterativo: alberi di Ricorrenza

Esempio: $T(n) = 4T(n/2) + n$



Metodo iterativo: alberi di Ricorrenza

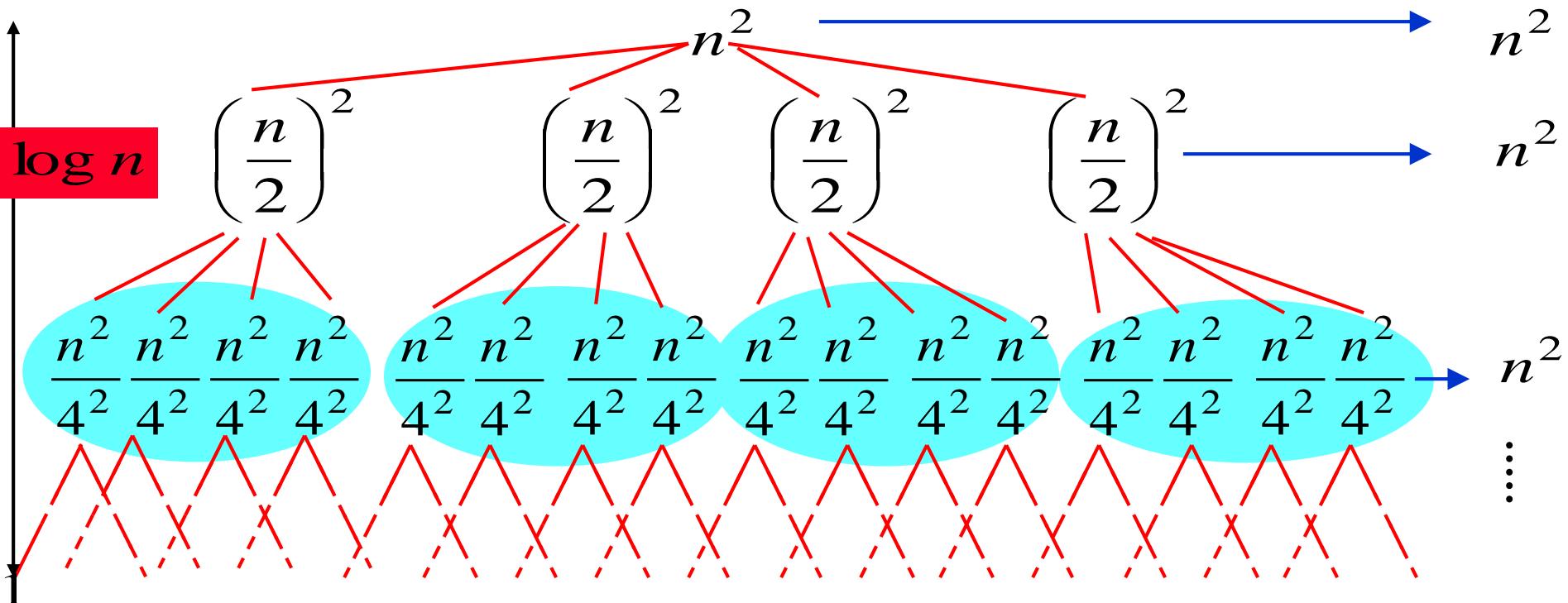
Esempio: $T(n) = 4T(n/2) + n$



$$T(n) = \sum_{k=0}^{\log n} n2^k = n \sum_{k=0}^{\log n} 2^k = \frac{2^{\log n + 1} - 1}{2 - 1} n = (2n - 1)n = 2n^2 - 1$$

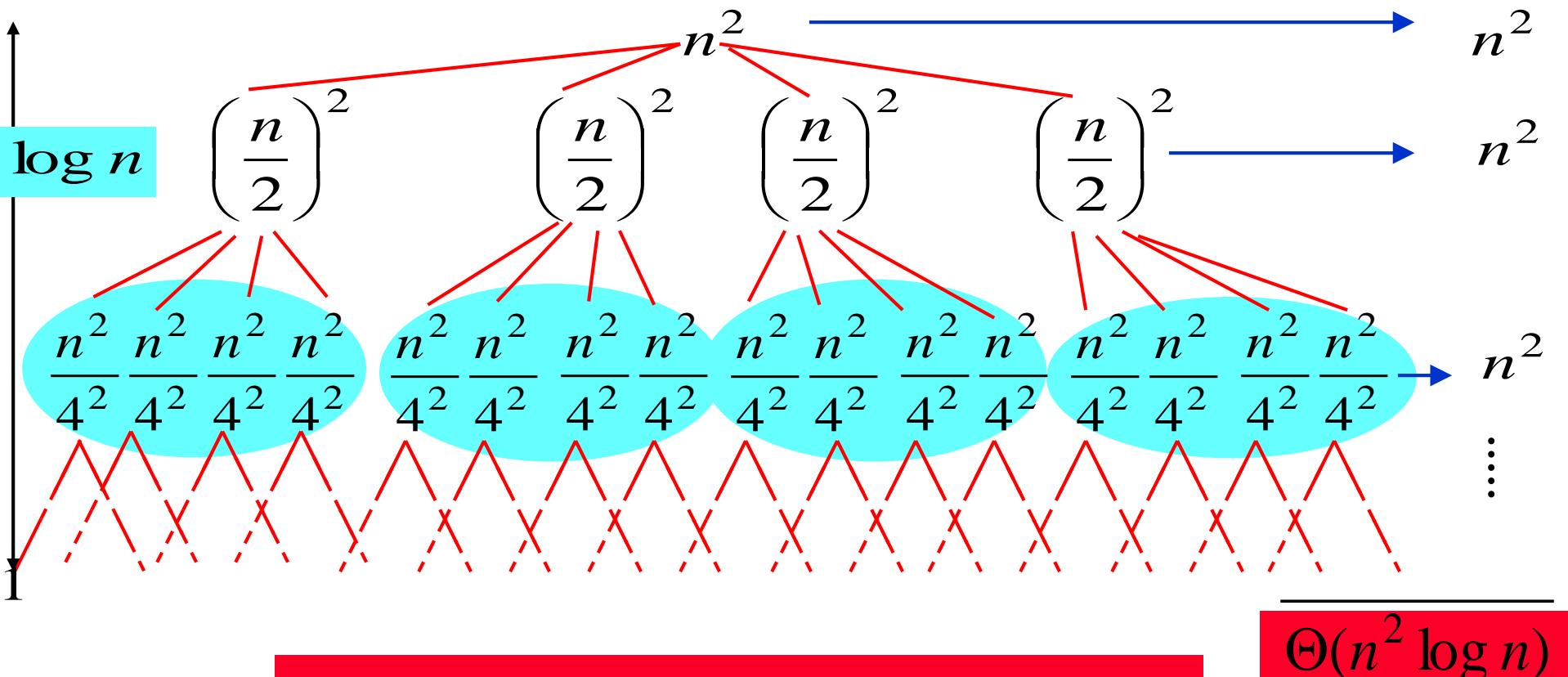
Metodo iterativo: alberi di Ricorrenza

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



Metodo iterativo: alberi di Ricorrenza

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

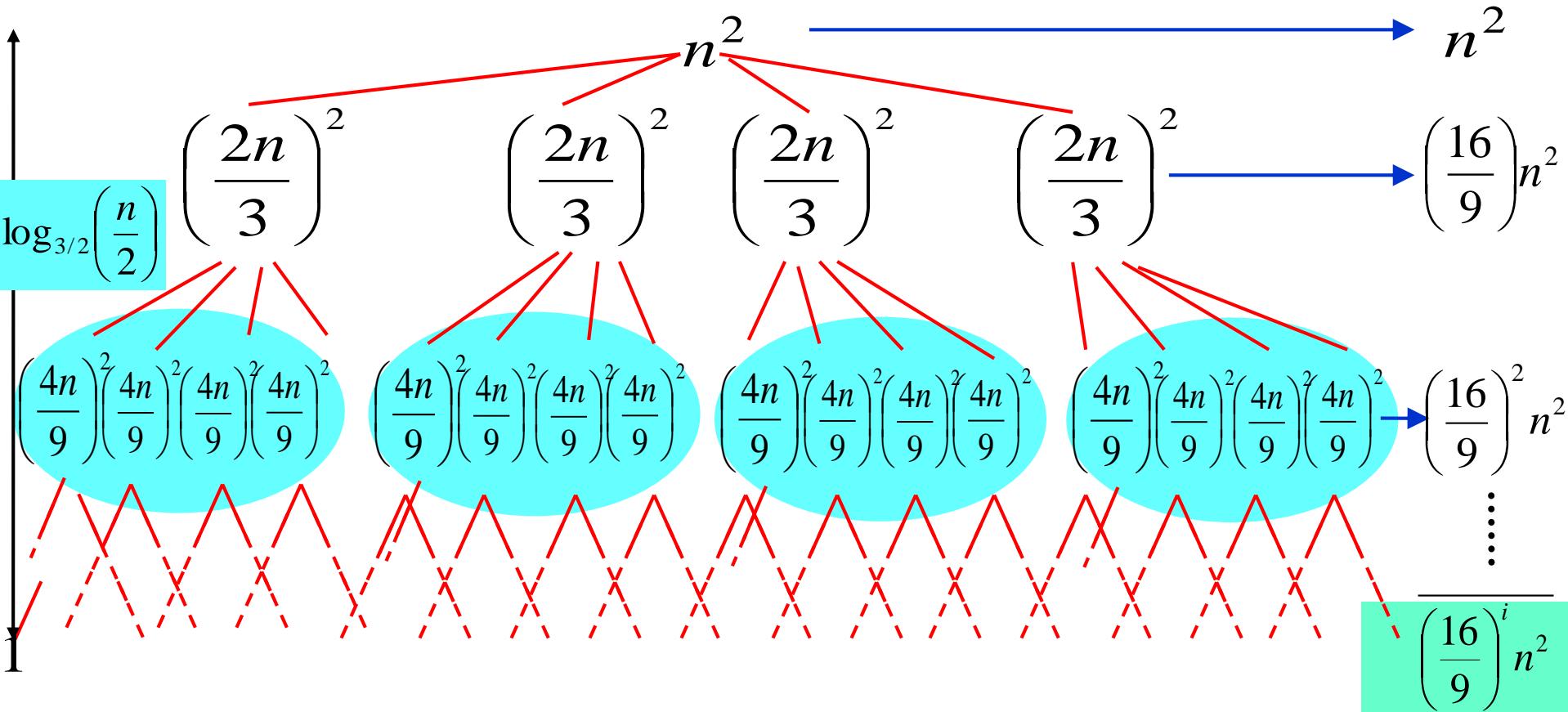


$$T(n) = \sum_{k=1}^{\log n} n^2 = n^2 \sum_{k=1}^{\log n} 1 = n^2 \log n$$

$\Theta(n^2 \log n)$

Metodo iterativo: alberi di Ricorrenza

Esempio: $T(n) = 4T(2 \cdot n/3) + n^2$



$$T(n) = \sum_{k=0}^{\log_{3/2}(n/2)} \left(\frac{16}{9}\right)^k n^2 = n^2 \frac{\left(\frac{16}{9}\right) \left(\frac{16}{9}\right)^{\log_{3/2}(n/2)}}{\frac{7}{9}} - 1 = \frac{16}{7} n^2 \left(\frac{n}{2}\right)^{\log_{3/2}(16/9)} - \left(\frac{9n^2}{7}\right) = \Theta\left(n^{\log_{3/2} 4}\right)$$

Metodo iterativo: alberi di Ricorrenza

Importante focalizzarsi su due parametri

- **il numero di volte in cui la ricorrenza deve essere iterata prima di giungere alla condizione limite (o base)**
- **la somma dei termini che compaiono ad ogni livello della iterazione.**

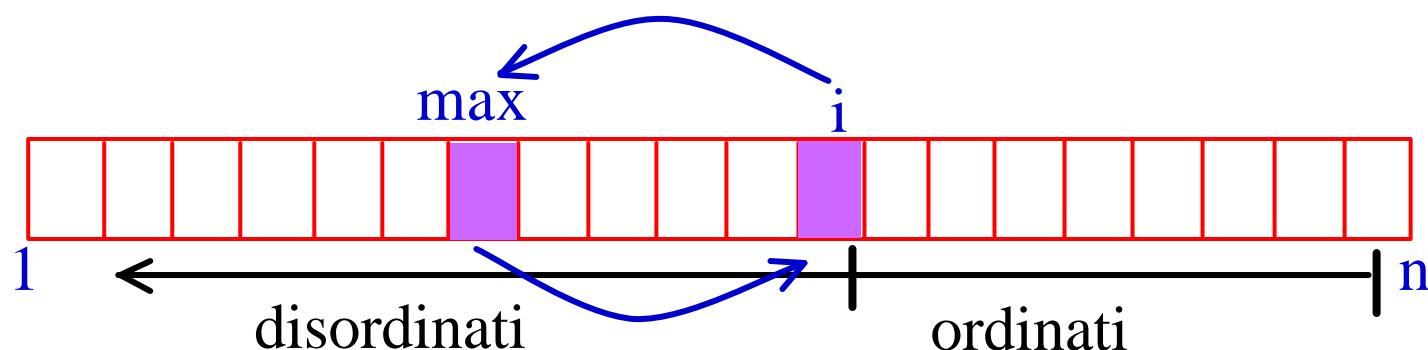
Algoritmi e Strutture Dati

HeapSort

Selection Sort: intuizioni

L'algoritmo Selection-Sort

- scandisce tutti gli elementi dell'array a partire dall'ultimo elemento fino all'inizio e ad ogni iterazione:
 - Viene cercato l'elemento massimo nella parte di array che precede l'elemento corrente
 - l'elemento massimo viene scambiato con l'elemento corrente



Select Sort

```
Selection-Sort(A)
```

```
    FOR i = length[A] DOWNTO 2
        DO max = Findmax(A,i)
            "scambia A[max] e A[i]"
```

```
Findmax(A,x)
```

```
    max = 1
    FOR i = 2 to x
        DO IF A[max] < A[i] THEN
            max = i
    return max
```

Heap Sort

L'algoritmo **Heap-Sort**:

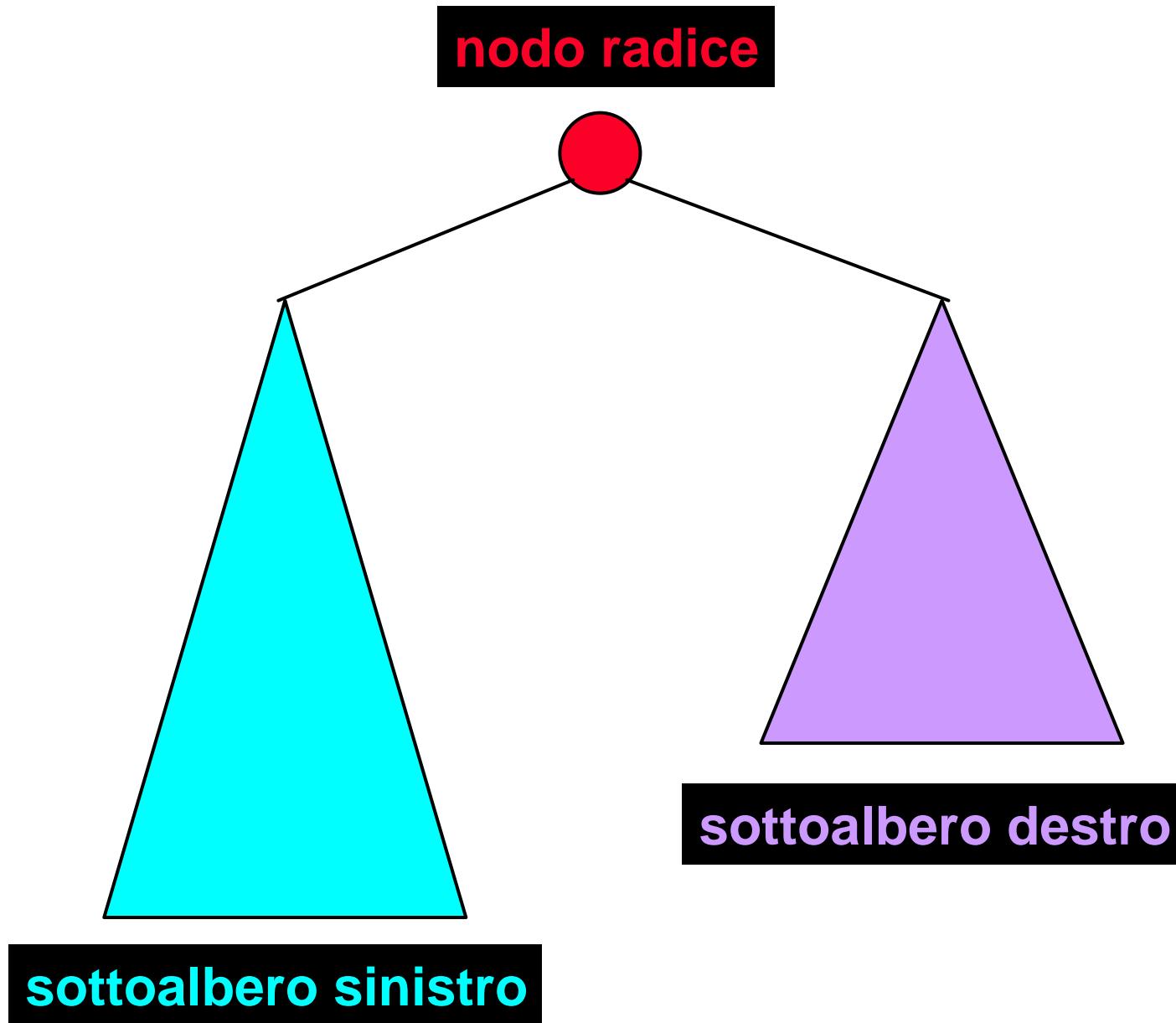
- è una variazione di **Selection-sort** in cui la ricerca dell'elemento massimo è facilitata dall'utilizzo di una opportuna struttura dati
- la struttura dati è chiamata **heap**
- lo **heap** è una variante della struttura dati **albero binario**
- in uno **heap** si può accedere all'**elemento massimo** in **tempo costante**
- si può ricostruire la struttura **heap** in **tempo sublineare**.

Alberi binari

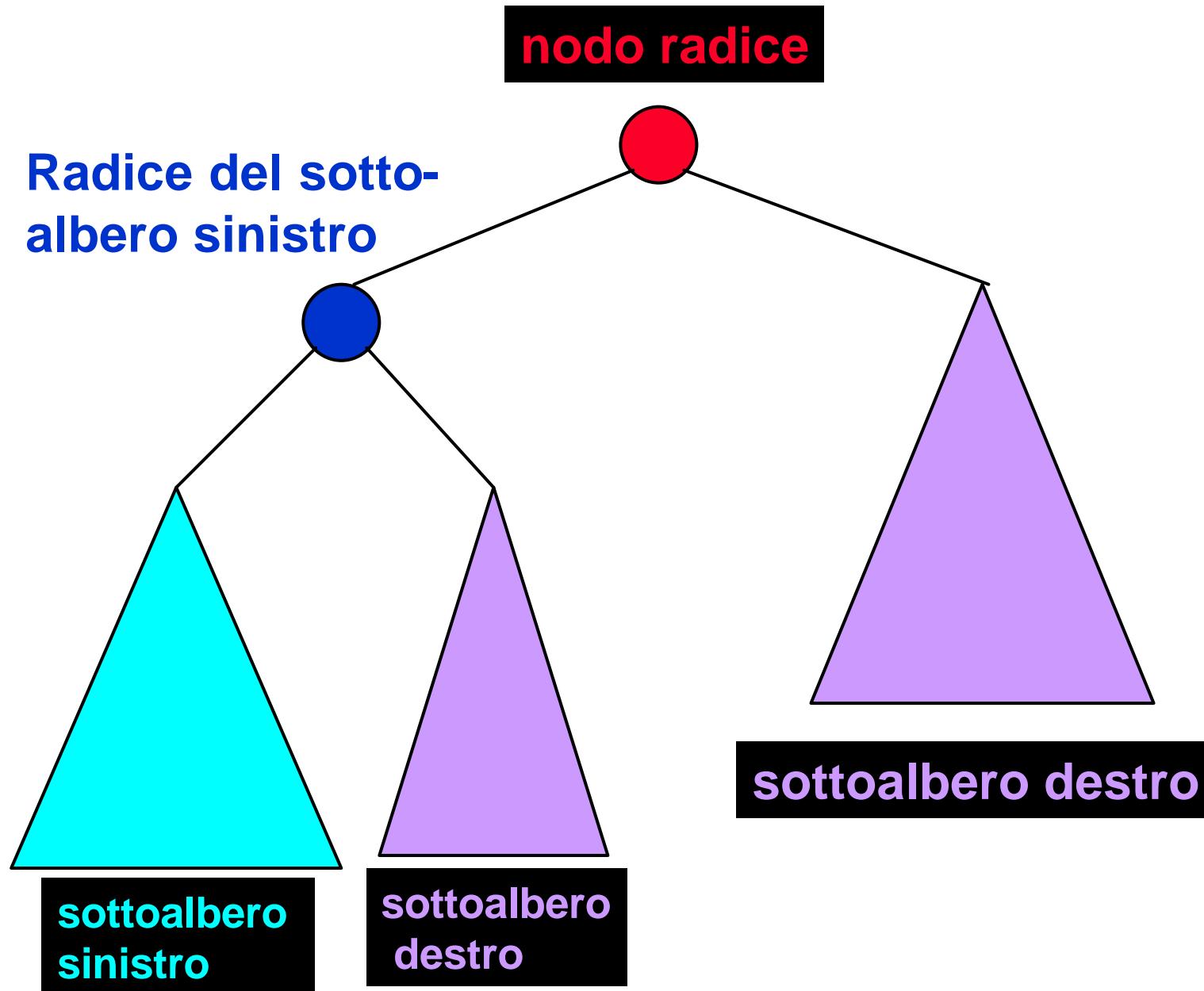
Un **albero binario** è una struttura dati **astratta** definita come un **insieme finito di nodi** che

- è **vuoto** oppure
- è composto da **tre insiemi disgiunti** di nodi:
 - un **nodo radice**
 - un albero binario detto **sottoalbero sinistro**
 - un albero binario detto **sottoalbero destro**

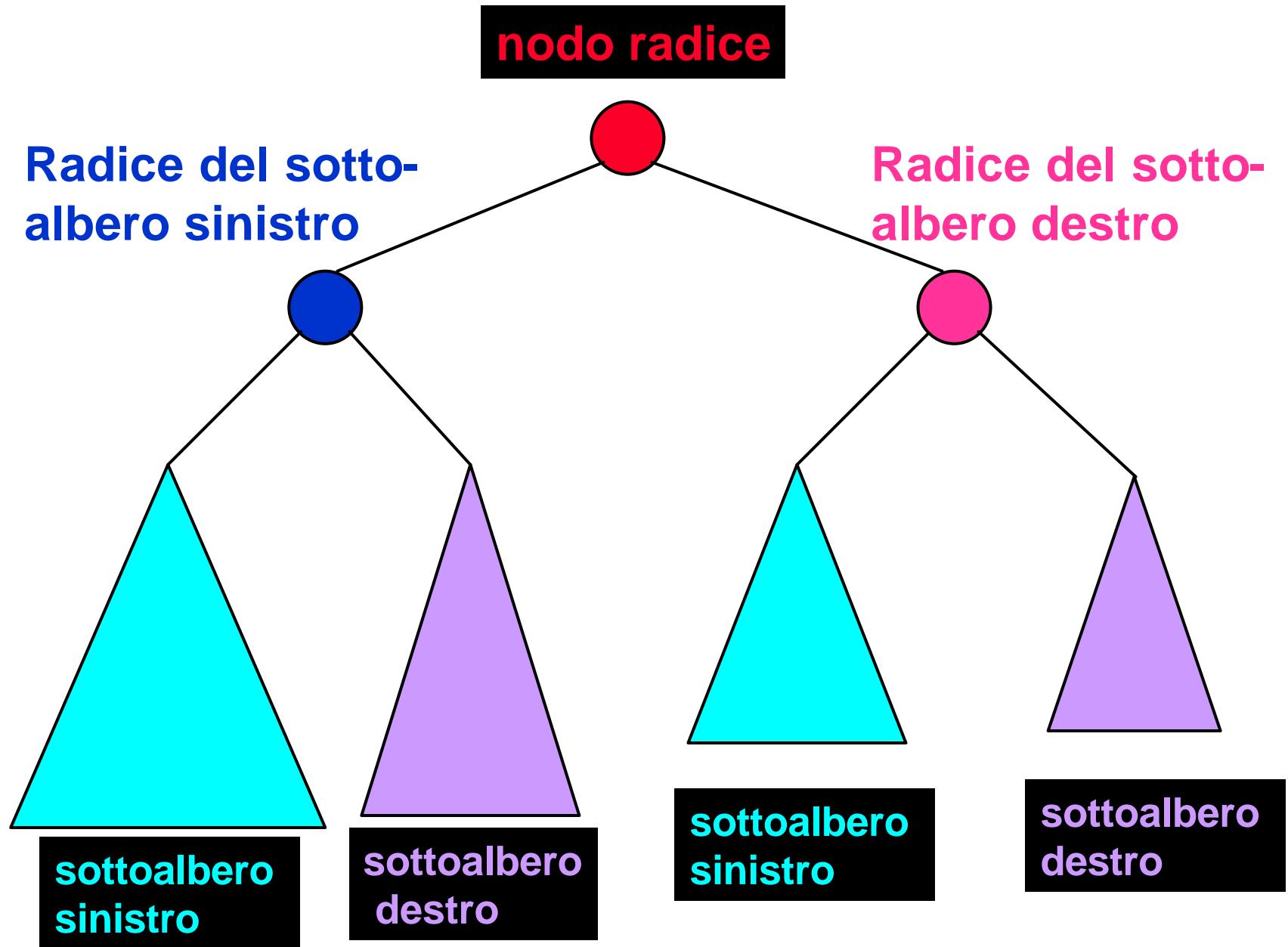
Alberi binari



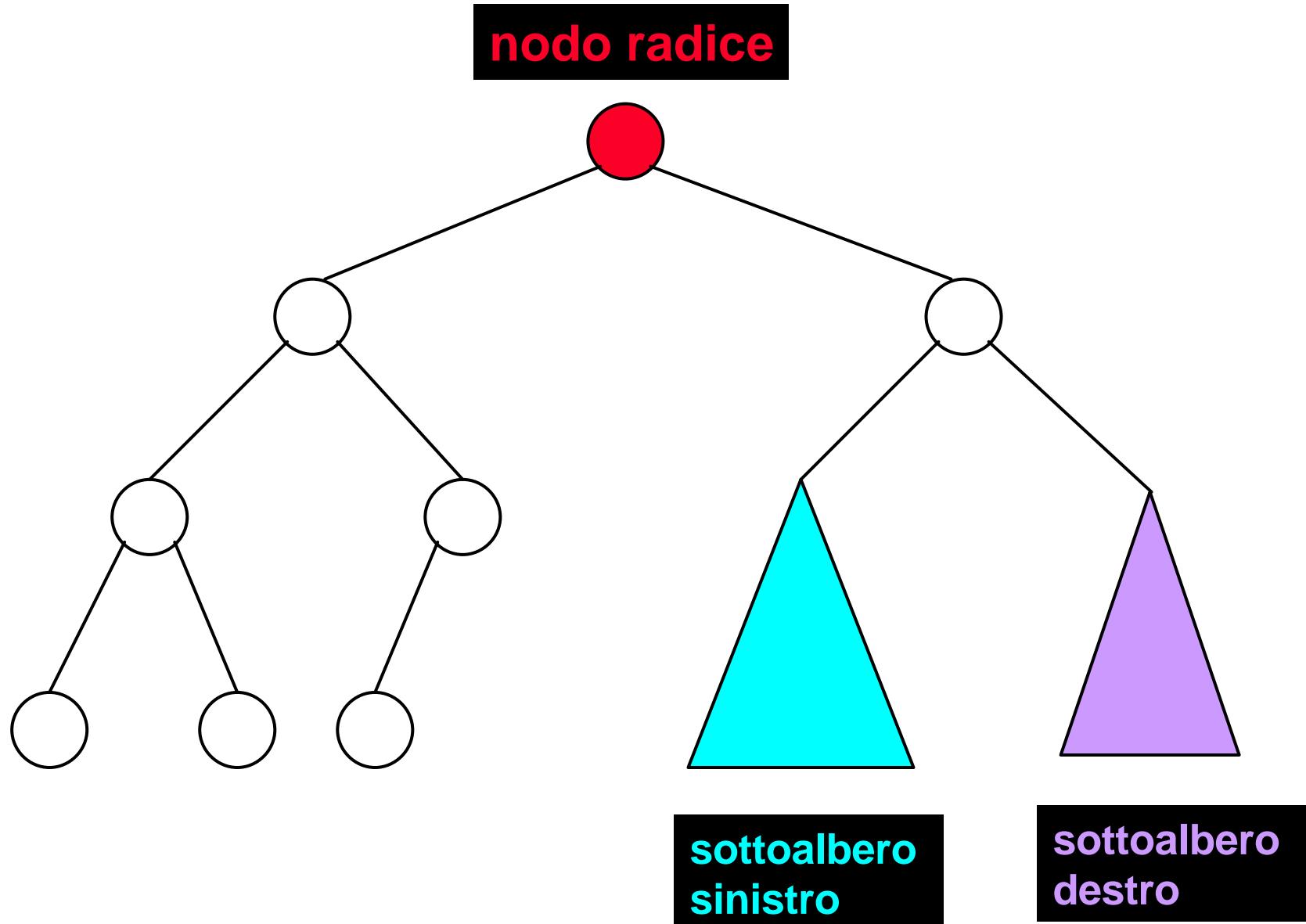
Alberi binari



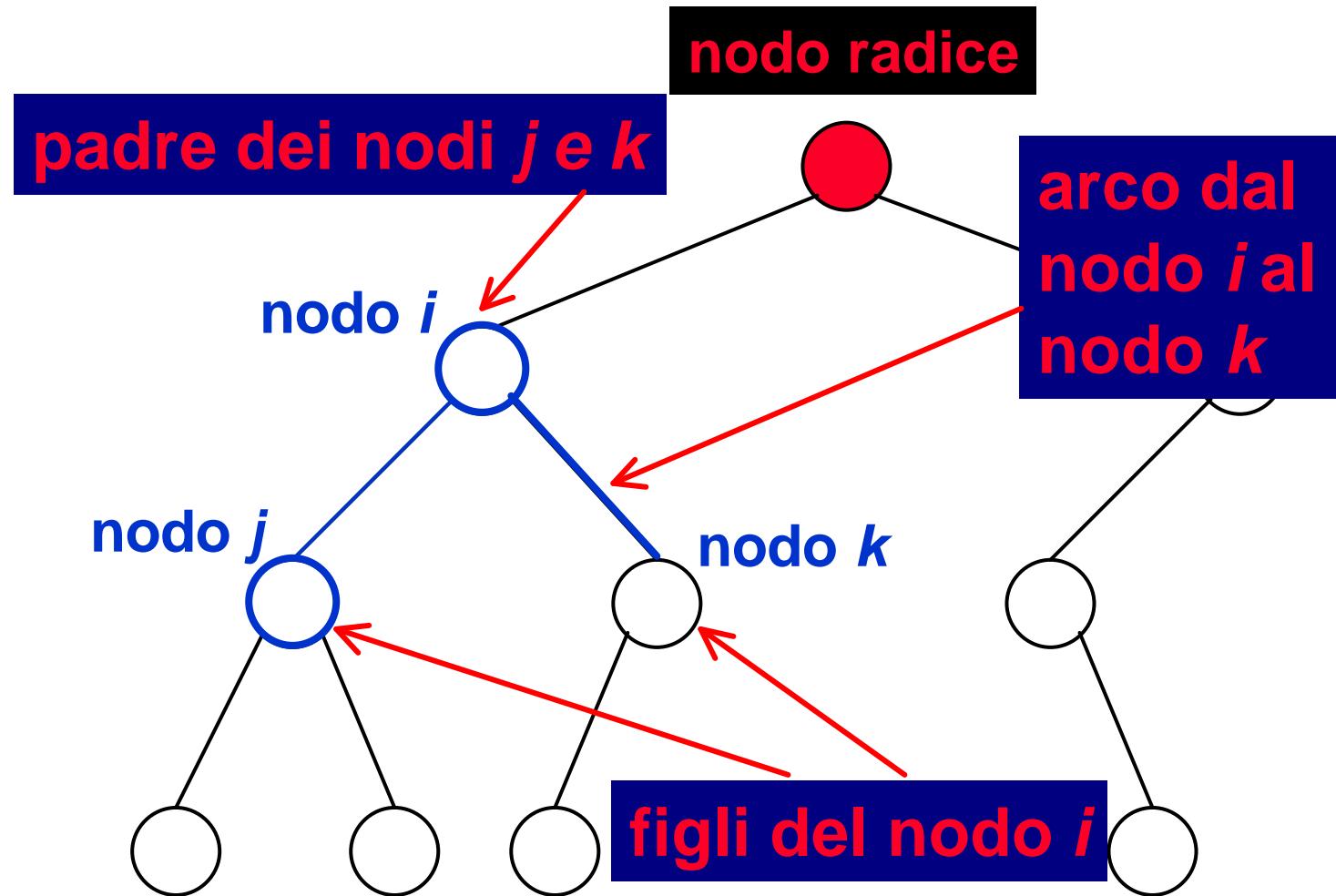
Alberi binari



Alberi binari



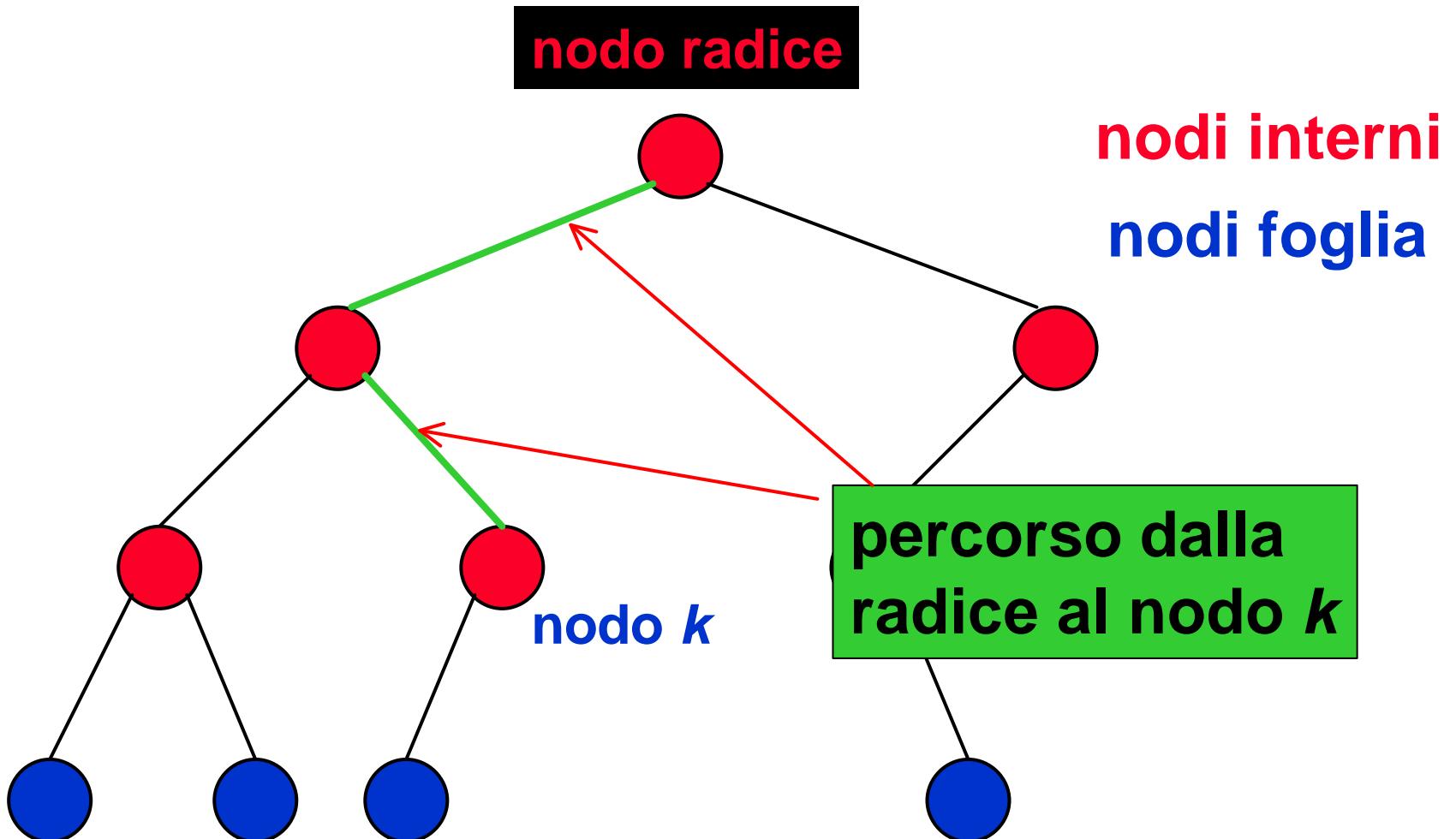
Alberi binari



Alberi binari

- In nodo di un albero binario si dice **nodo foglia** (o solo **foglia**) se non ha **figli** (cioè se entrambi i sottoalberi di cui è radice sono vuoti)
- Un nodo si dice **nodo interno** se ha **almeno un figlio**
- Un **percorso dal nodo *i* al nodo *j*** è la **sequenza di archi** che devono essere attraversati per raggiungere il **nodo *j*** dal **nodo *i***

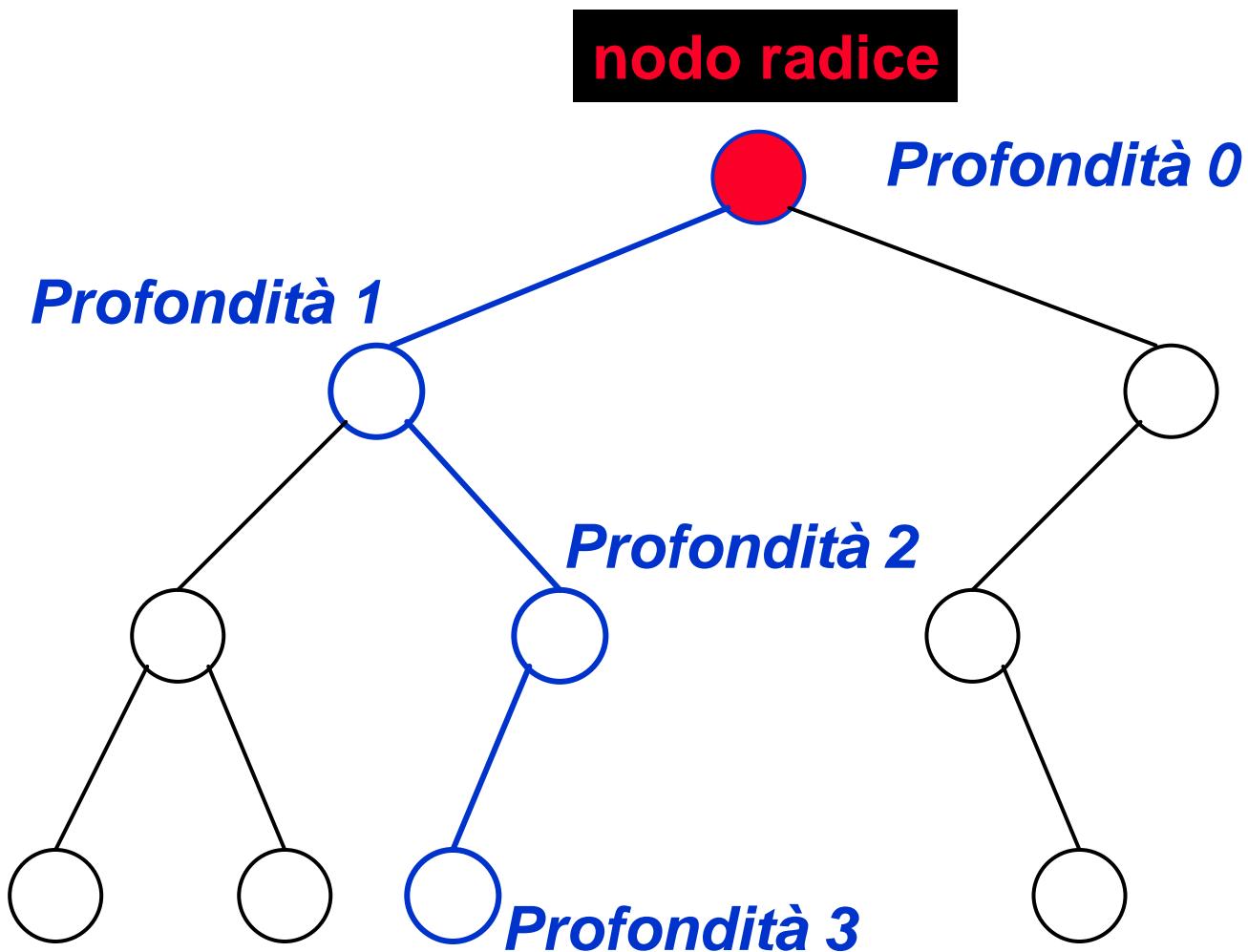
Alberi binari



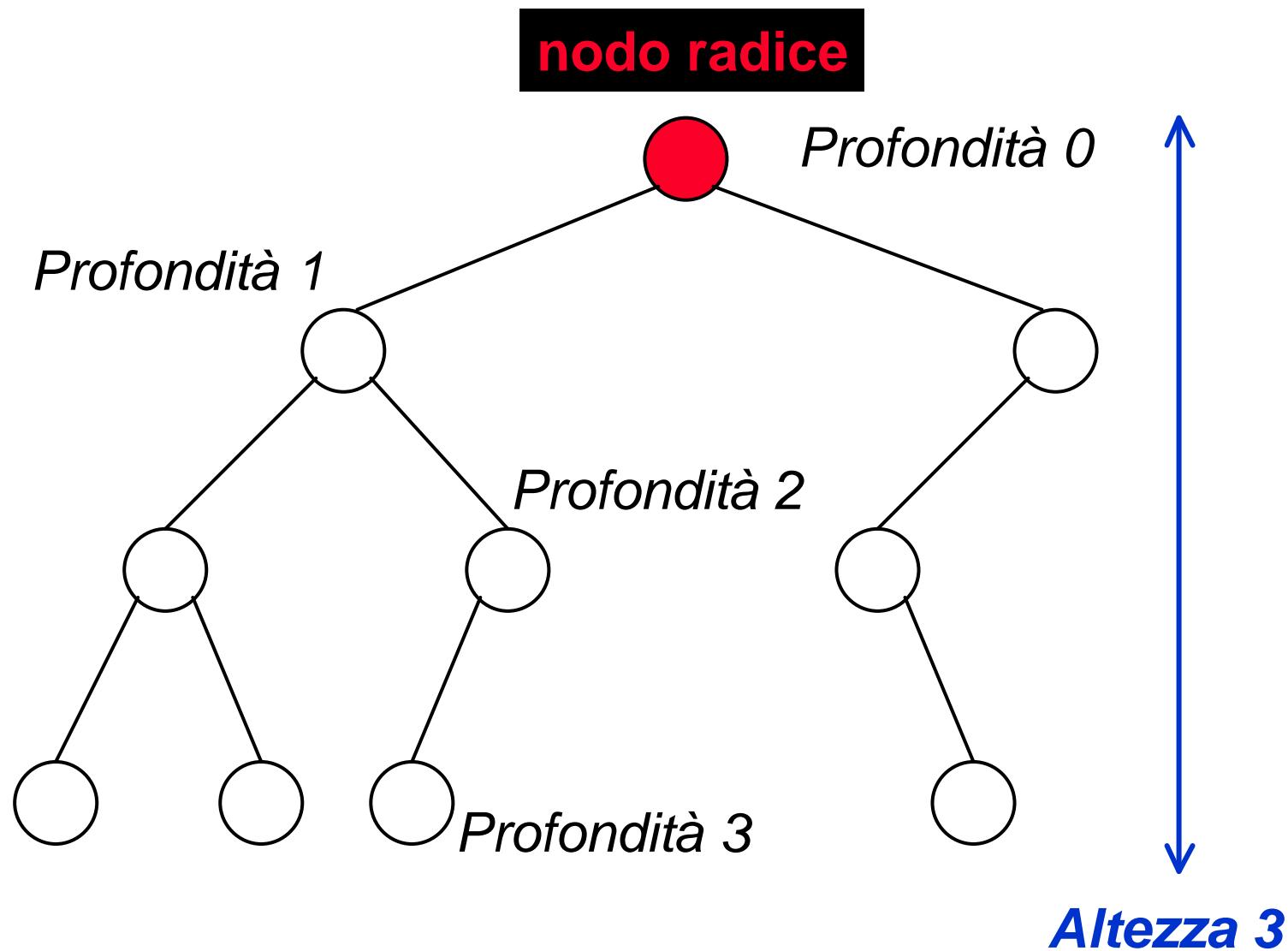
Alberi binari

- In un **albero binario** la **profondità** di un **nodo** è la **lunghezza del percorso** dalla **radice** al **nodo** (cioè il numero di archi tra la radice e il nodo)
- La **profondità massima** di un **nodo** all'interno di un albero è l'**altezza dell'albero**.
- Il **grado** di un **nodo** è il **numero di figli** di quel nodo.

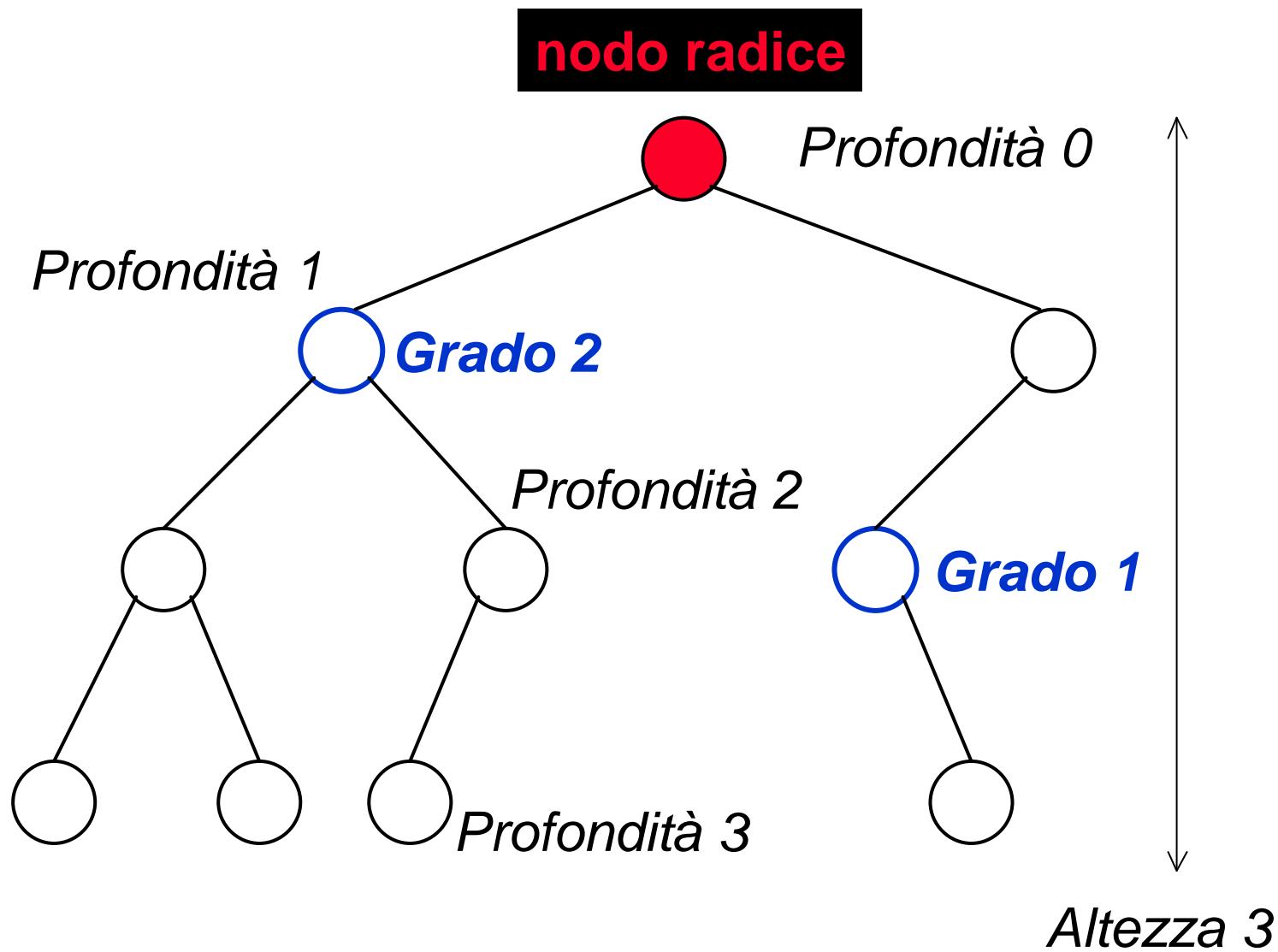
Alberi binari



Alberi binari



Alberi binari



Albero binario pieno

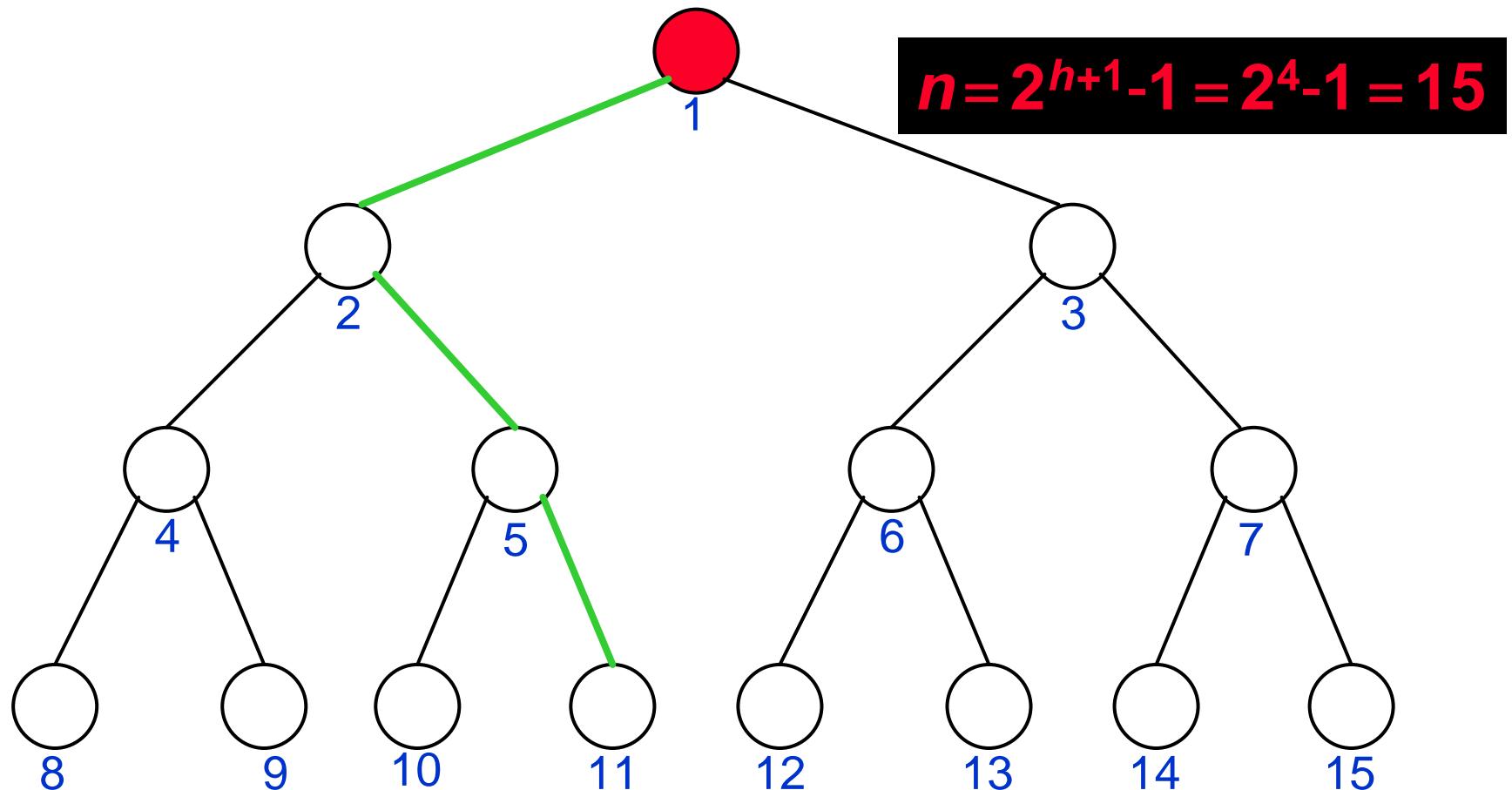
Un *albero binario* si dice *pieno* se

- ① tutte le *foglie* hanno la stessa *profondità* h
- ② tutti i *nodi interni* hanno *grado 2*

• Un albero pieno con n nodi ha altezza esattamente $\log_2 n$.

• Un albero pieno di altezza h ha esattamente $2^{h+1}-1$ nodi.

Albero binario pieno



$$n = 15 \text{ e altezza } h = \lceil \log_2 n \rceil = 3$$

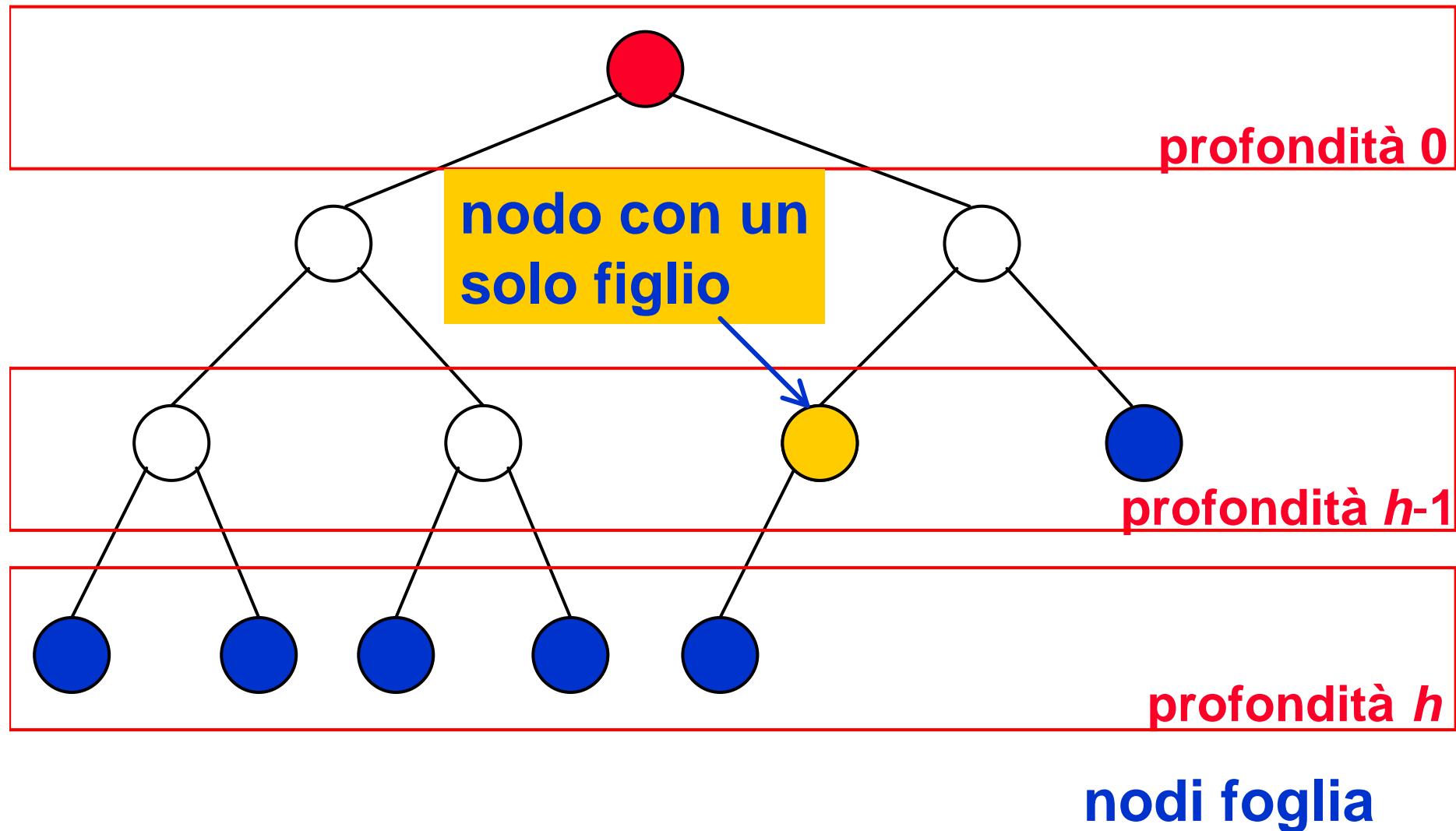
Albero binario completo

Un *albero binario* si dice *completo* se

- tutte le *foglie* hanno *profondità* h o $h-1$,
dove h è l'*altezza dell'albero*

- tutti i *nodi interni* hanno *grado 2*, eccetto al
più uno.

Albero binario completo



Proprietà di uno Heap

Un **Albero Heap** è un albero binario tale che per ogni nodo *i*:

- tutte le *foglie* hanno *profondità h o h-1*, dove *h* è l'*altezza dell'albero*;
- tutti i *nodi interni* hanno *grado 2*, eccetto al più uno;
- entrambi i *nodi j e k* figli di *i* sono **NON maggiori** di *i*.

Condizioni 1 e 2 definiscono la **forma dell'albero**

Proprietà di uno Heap

Un **Albero Heap** è un albero binario tale che per ogni nodo *i*:

- tutte le **foglie** hanno **profondità *h* o *h-1***, dove ***h*** è l'**altezza dell'albero**;
- tutti i **nodi interni** hanno **grado 2**, eccetto al più uno;
- entrambi i **nodi *j* e *k*** figli di *i* sono **NON maggiori** di *i*.

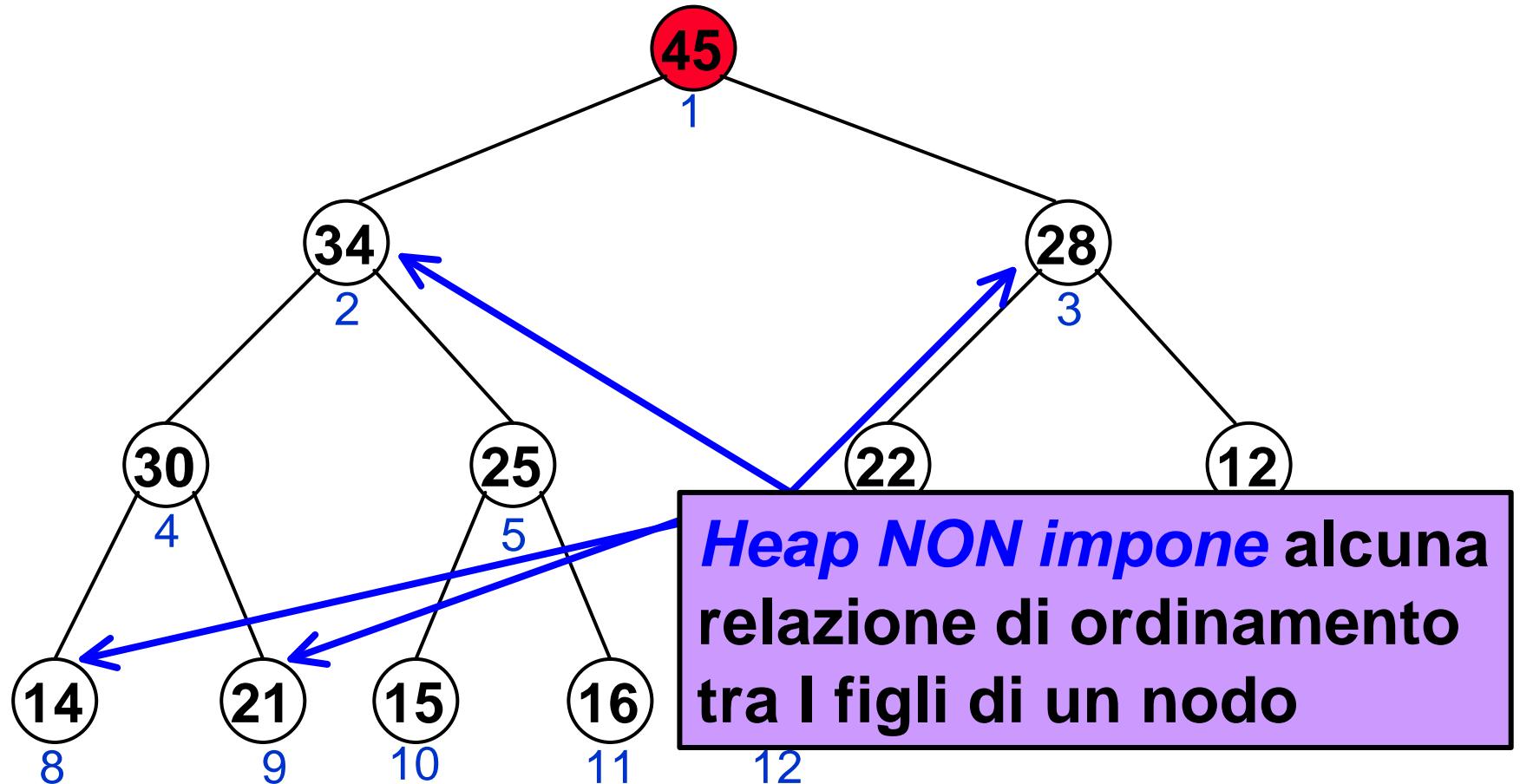
Condizione 3 definisce
l'*etichettatura dell'albero*

Proprietà di uno Heap

Un *Albero Heap* è un albero binario completo tale che per ogni nodo *i*:

- entrambi i nodi *j* e *k* figli di *i* sono **NON maggiori** di *i*.

Heap



Heap e Ordinamenti Parziali

Un *Albero Heap* è un *albero binario completo* tale che per ogni nodo i :

- entrambi i nodi j e k figli di i sono *NON* maggiori di i .

Uno *Heap* rappresenta un *Ordinamento Parziale*

Heap e Ordinamenti Parziali

Un **Ordinamento Parziale** è una relazione tra elementi di un insieme che è:

- **Riflessiva**: x è in relazione con se stesso
- **Anti-simmetrica**: se x è in relazione con y e y è in relazione con x allora $x=y$
- **Transitiva**: se x è in relazione con y e y è in relazione con z allora x è in relazione con z

Esempi:

- le relazioni \leq e \geq
- le relazioni $>$, $<$ **NON sono ordinamenti parziali**

Heap e Ordinamenti Parziali

Gli *Ordinamenti Parziali* possono essere usati per modellare, ad esempio, gerarchie con *informazione incompleta* o *elementi con uguali valori*.

L'ordinamento parziale definito da uno *Heap* è una nozione *più debole* di un *ordinamento totale*, infatti uno *Heap*

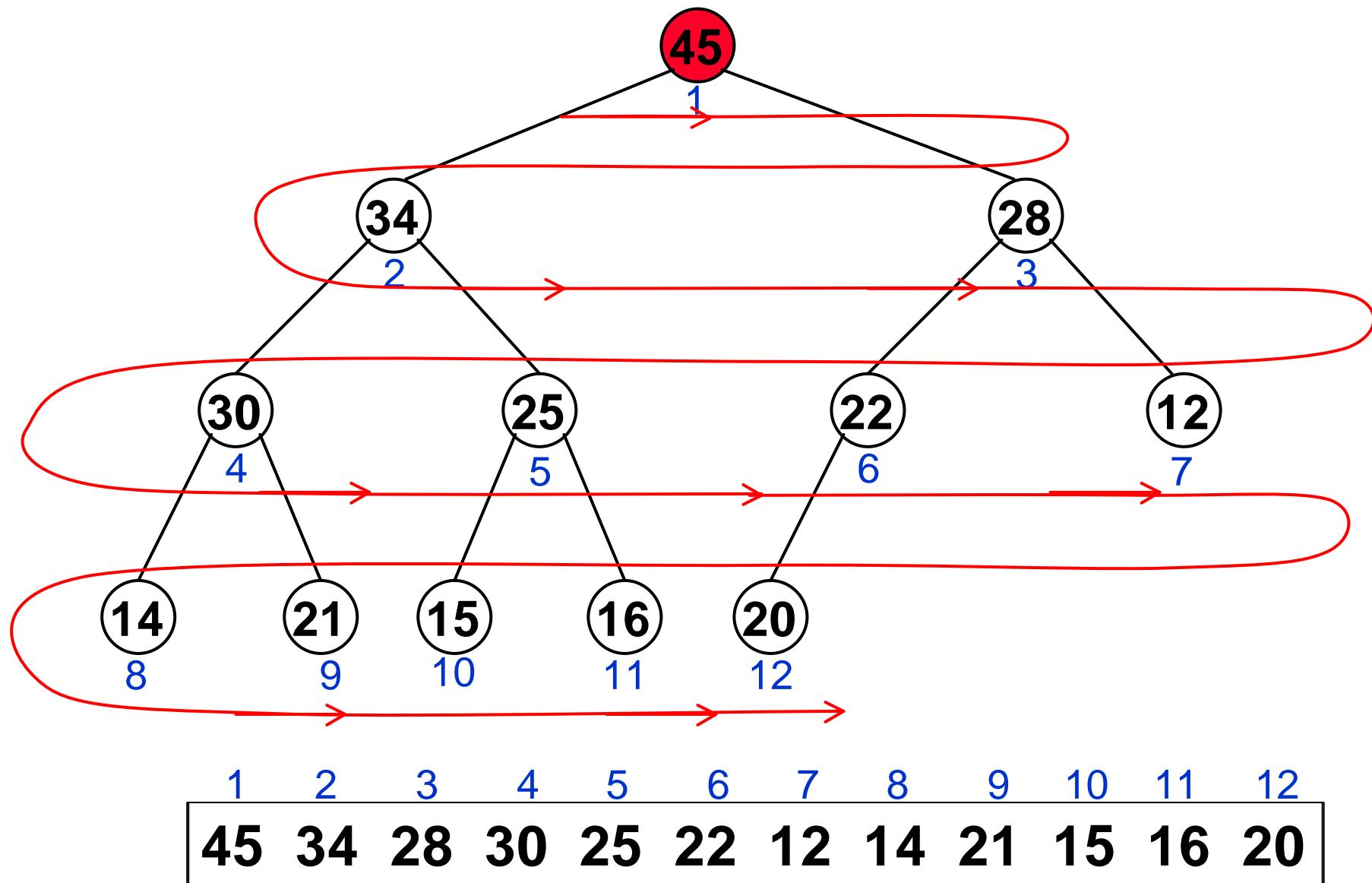
- è più semplice da costruire
- è molto utile ma meno dell'ordinamento totale

Implementazione di uno Heap

Uno **Heap** può essere implementato in vari modi:

- come un albero a puntatori
- come un array
-

Heap: da albero ad array



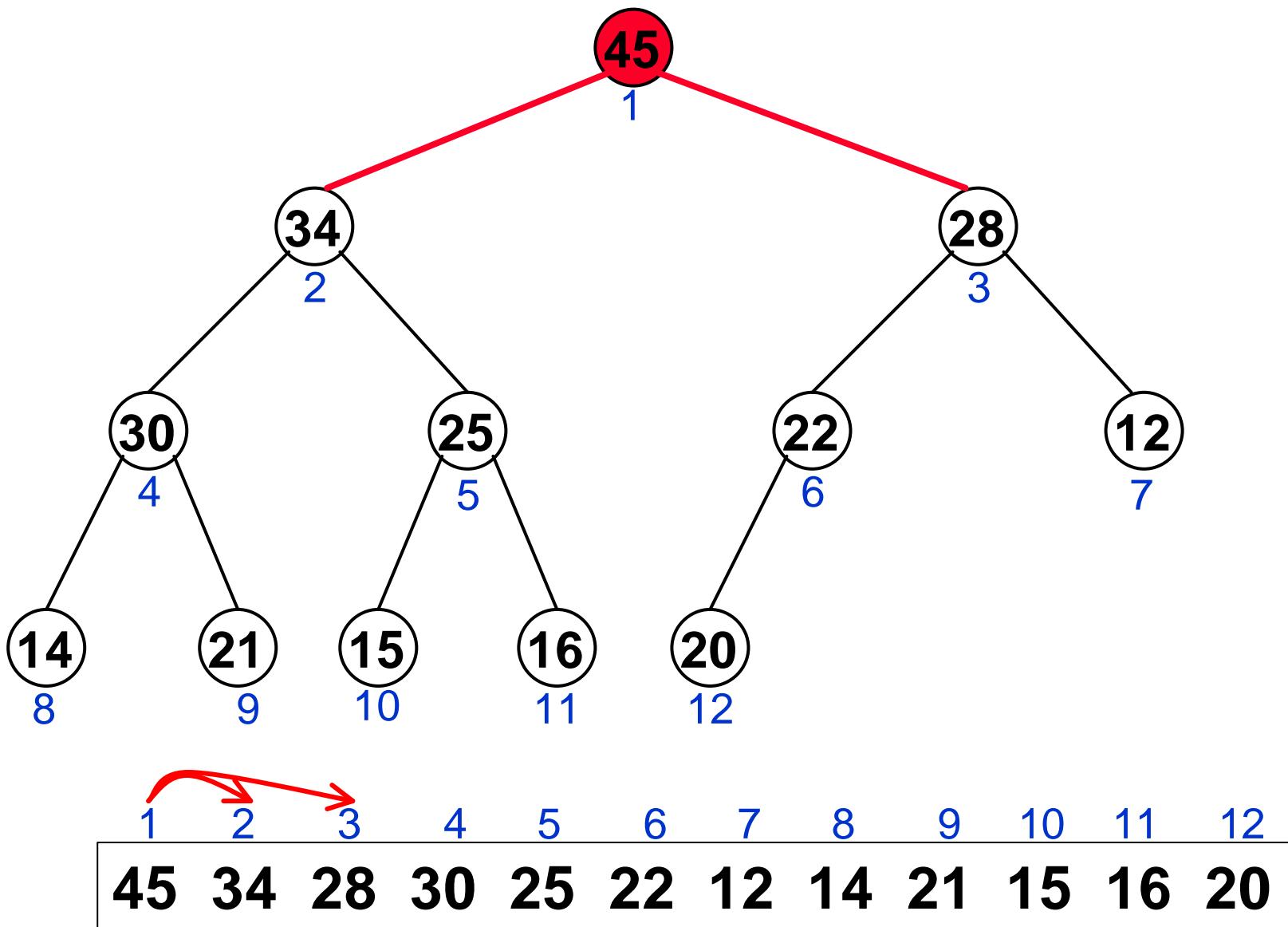
Heap: da albero ad array

Uno *Heap* può essere implementato come un array A in cui:

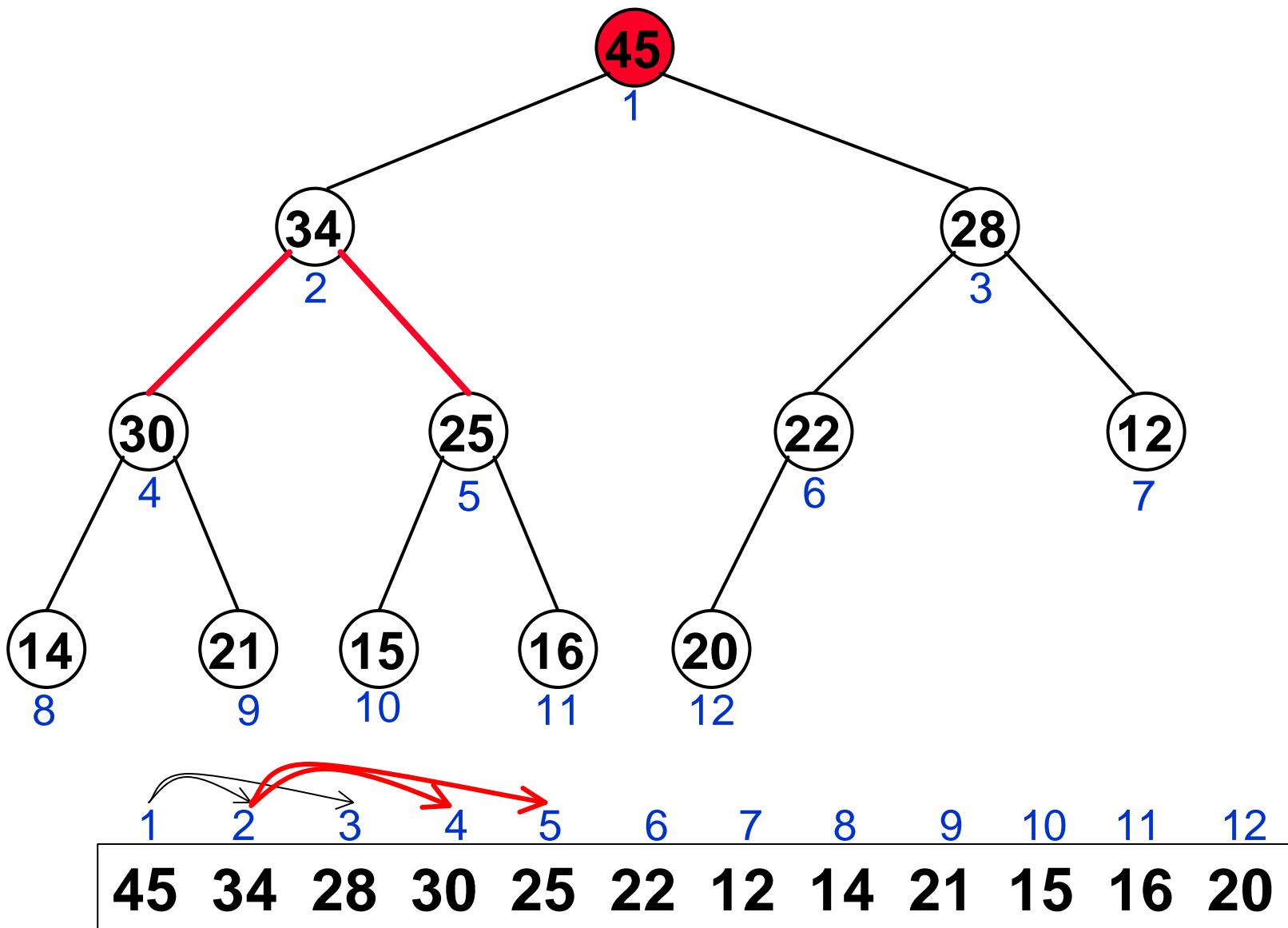
- la radice dello *Heap* sta nella posizione $A[0]$ dell'array
- se il nodo i dello *Heap* sta nella posizione i dell'array (cioè $A[i]$),
 - il figlio sinistro di i sta nella posizione $2i$
 - il figlio destro di i sta nella posizione $2i+1$

1	2	3	4	5	6	7	8	9	10	11	12
45	34	28	30	25	22	12	14	21	15	16	20

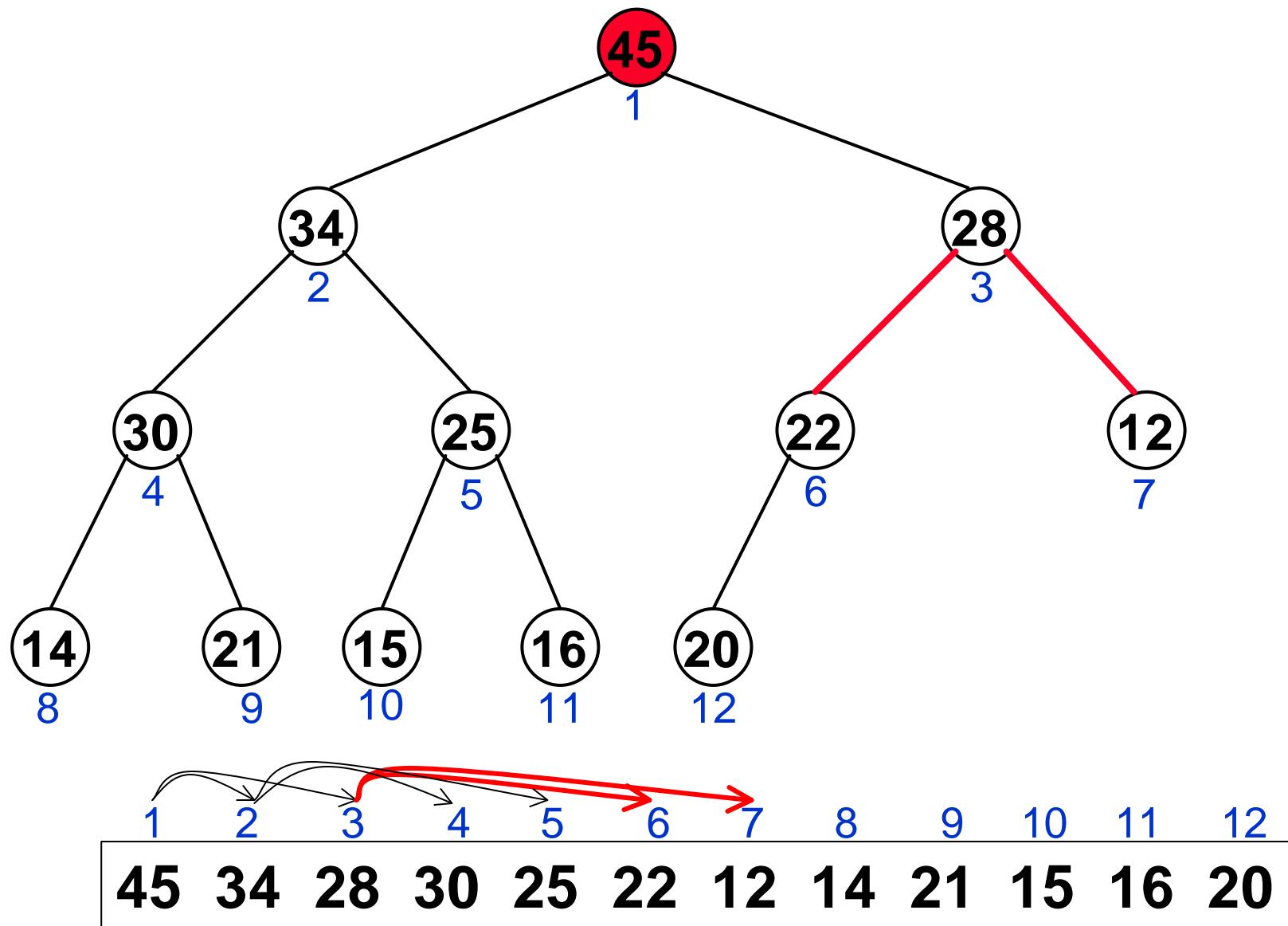
Heap: da albero ad array



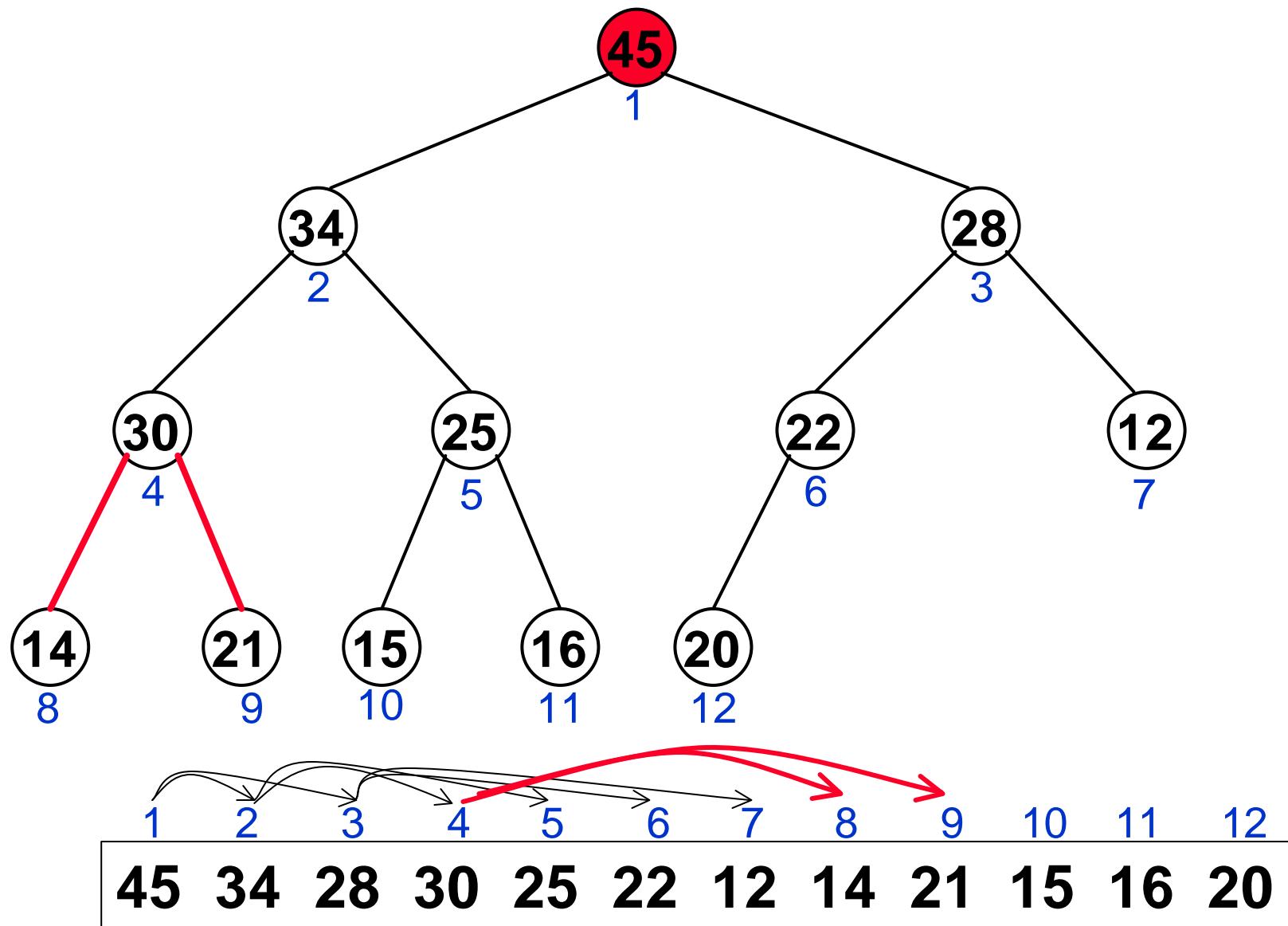
Heap: da albero ad array



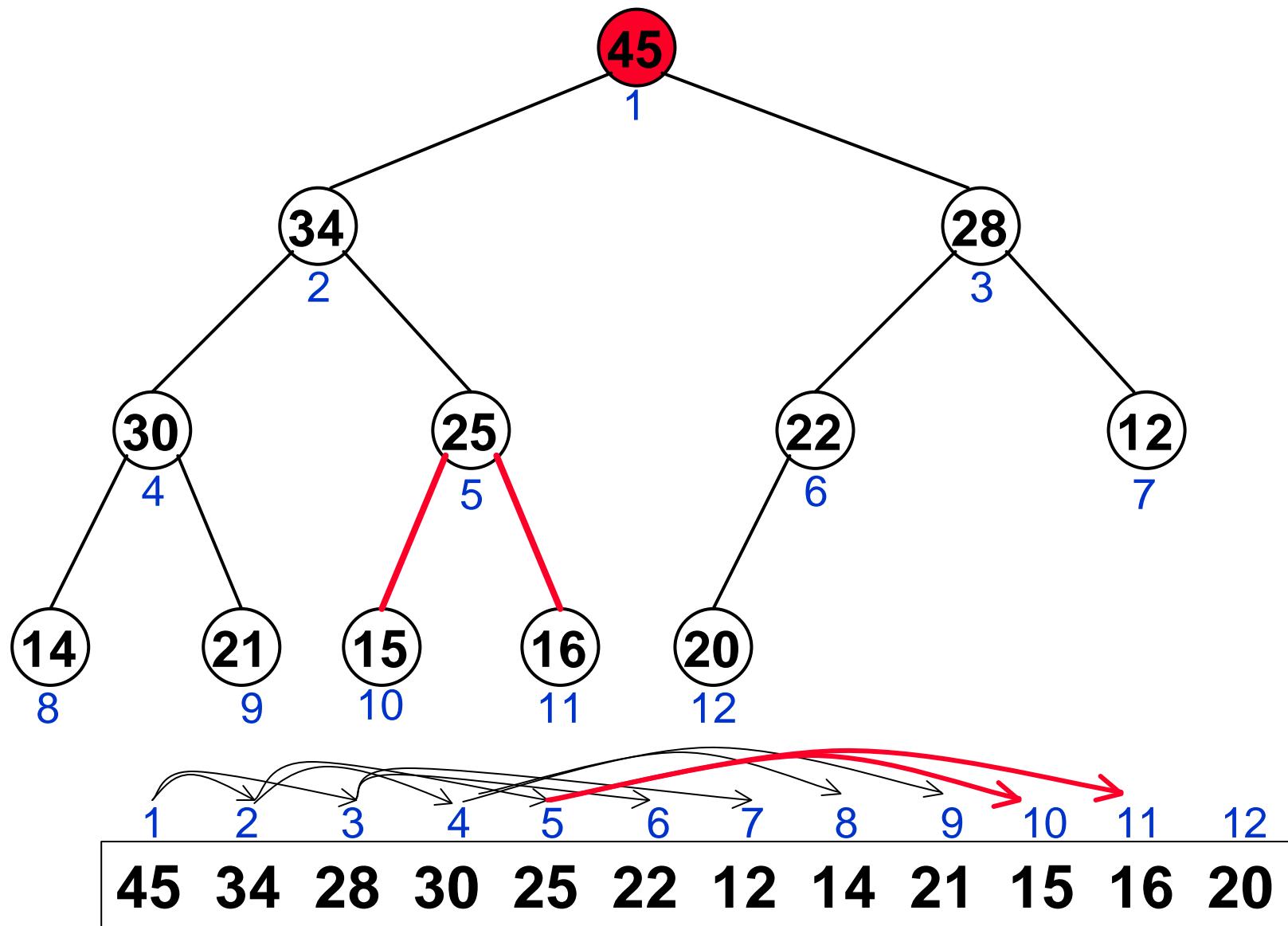
Heap: da albero ad array



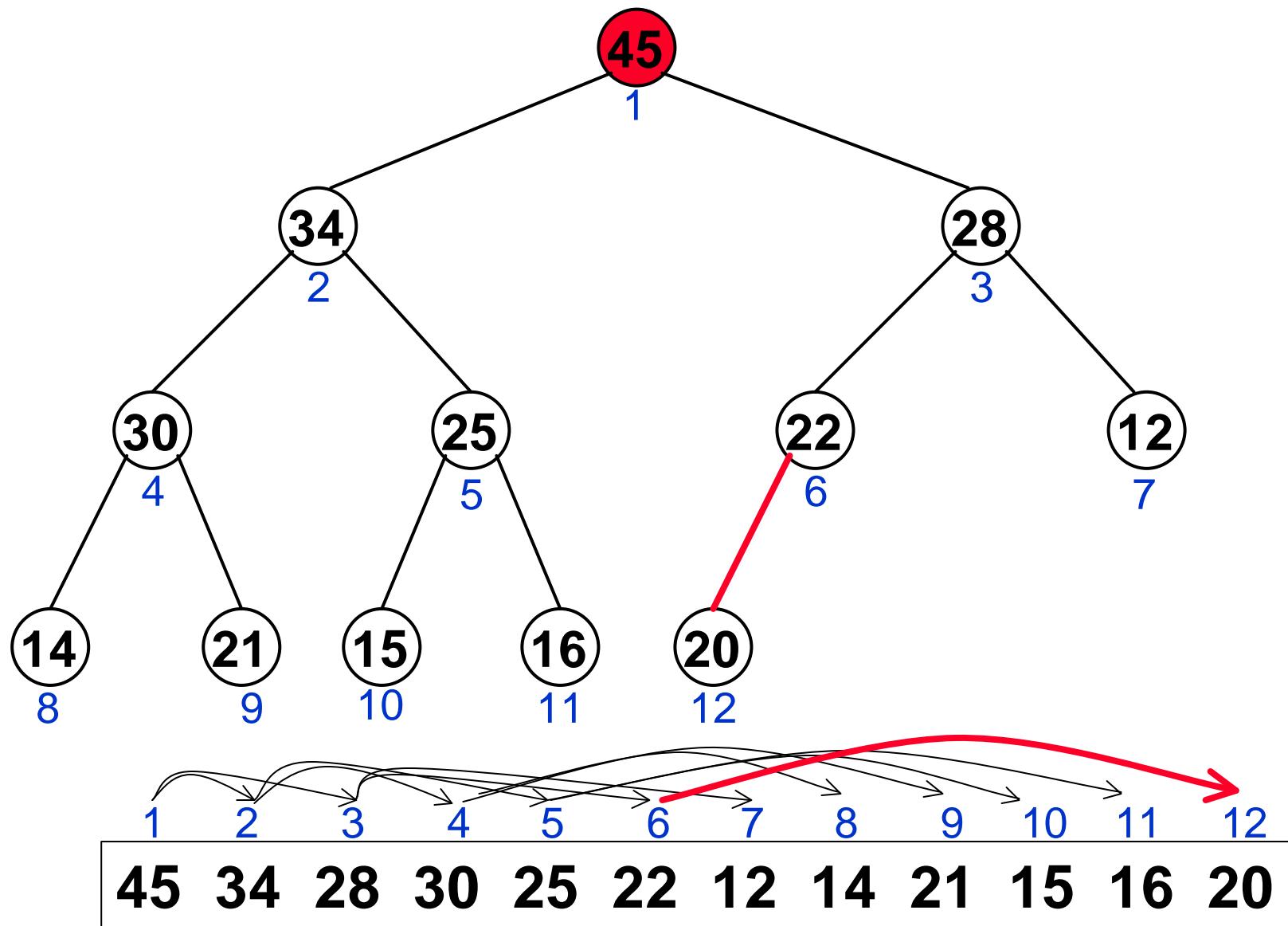
Heap: da albero ad array



Heap: da albero ad array



Heap: da albero ad array



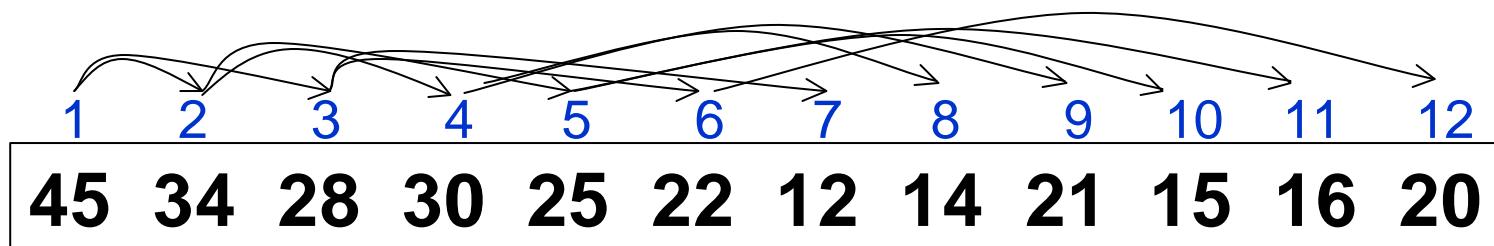
Proprietà di uno Heap

Un *Albero Heap* è un *albero binario completo* tale che per ogni nodo i :

- entrambi i nodi j e k figli di i sono NON maggiori di i .

Un *array A* è uno *Heap* se

$$A[i] \geq A[2i] \quad \text{e} \quad A[i] \geq A[2i+1]$$



Operazioni elementari su uno Heap

SINISTRO(*i*)

return $2i$

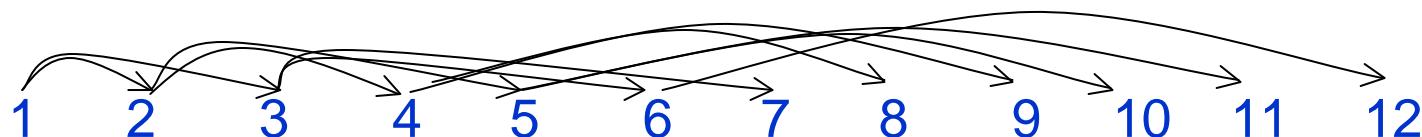
DESTRO(*i*)

return $2i + 1$

PADRE(*i*)

return $\lceil i/2 \rceil$

heapsize[A] £ n è la lunghezza dello *Heap*



45 34 28 30 25 22 12 14 21 15 16 20

Operazioni su uno Heap

- **Heapify(A, i)**: ripristina la proprietà di **Heap** al sottoalbero radicato in i assumendo che i suoi sottoalberi destro e sinistro siano già degli **Heap**
- **Costruisci-Heap(A)**: produce uno **Heap** a partire dall'array A non ordinato
- **HeapSort(A)**: ordina l'array A sul posto.

Heapify: Intuizioni

Heapify(A, i): dati due **Heap H_1** e **H_2** con radici **SINISTRO(i)** e **DESTRO(i)** e un nuovo elemento **v** in posizione **$A[i]$**

- se lo **Heap H** con radice **$A[i]=v$** **viola** la **proprietà di Heap** allora:
 - metti in **$A[i]$** la **più grande** tra le radici degli **Heap H_1** e **H_2**
 - applica **Heapify** ricorsivamente al sottoalbero modificato (con radice **$A[2i]$** o **$A[2i+1]$**) e all'elemento **v** (ora in posizione **$A[2i]$** o **$A[2i+1]$**).

Algoritmo Heapify

Heapify(A, i)

***l* = SINISTRO (*i*)**

***r* = DESTRO(*i*)**

IF *l* ≤ heapsize[A] AND A[*l*] > A[*i*]

THEN *maggiore* = *l*

ELSE *maggiore* = *i*

IF *r* ≤ heapsize[A] AND A[*r*] > A[*maggiore*]

THEN *maggiore* = *r*

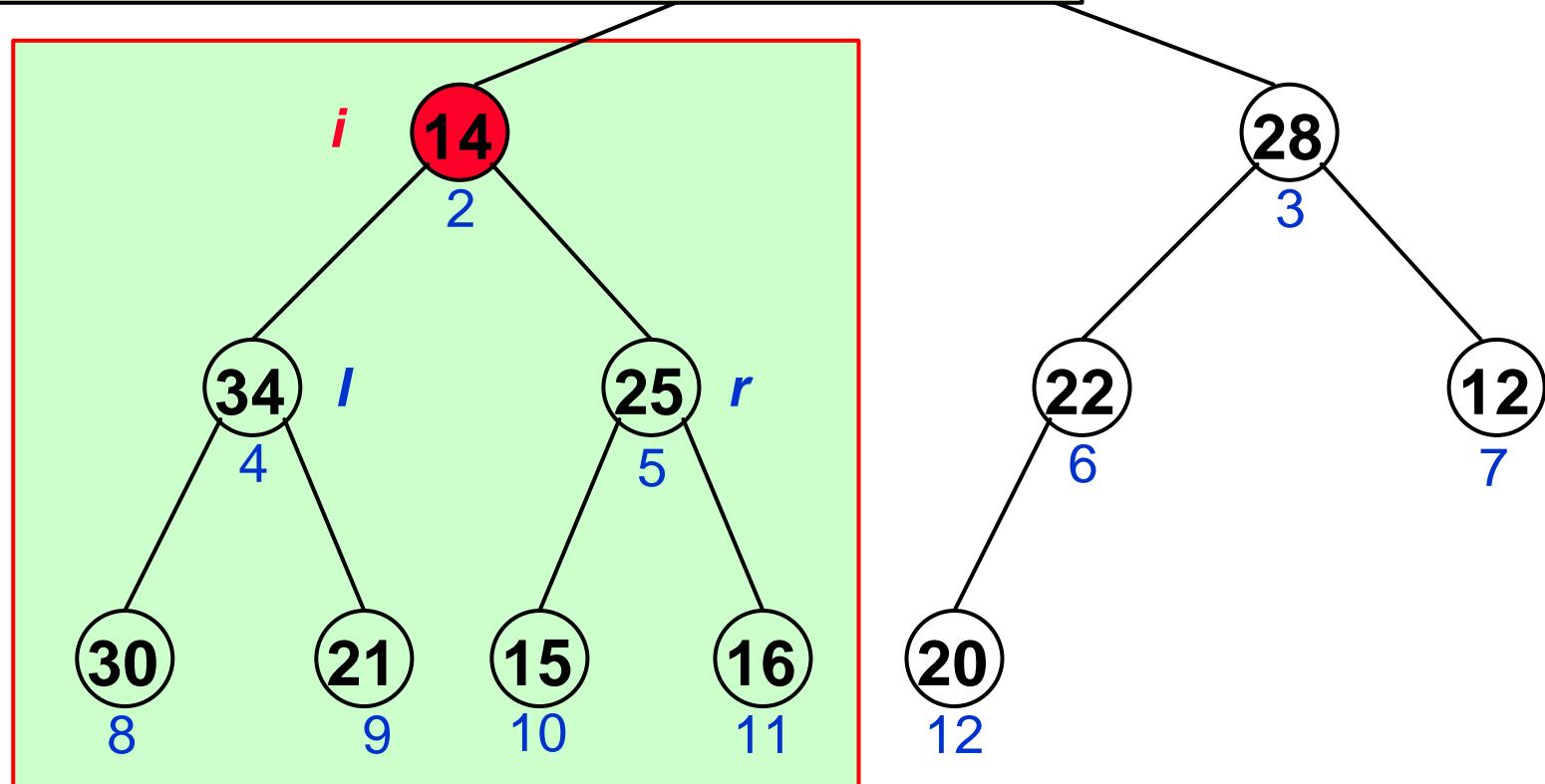
IF *maggiore* ≠ *i*

THEN “scambia A[*i*] e A[*maggiore*]”

Heapify(A, *maggiore*)

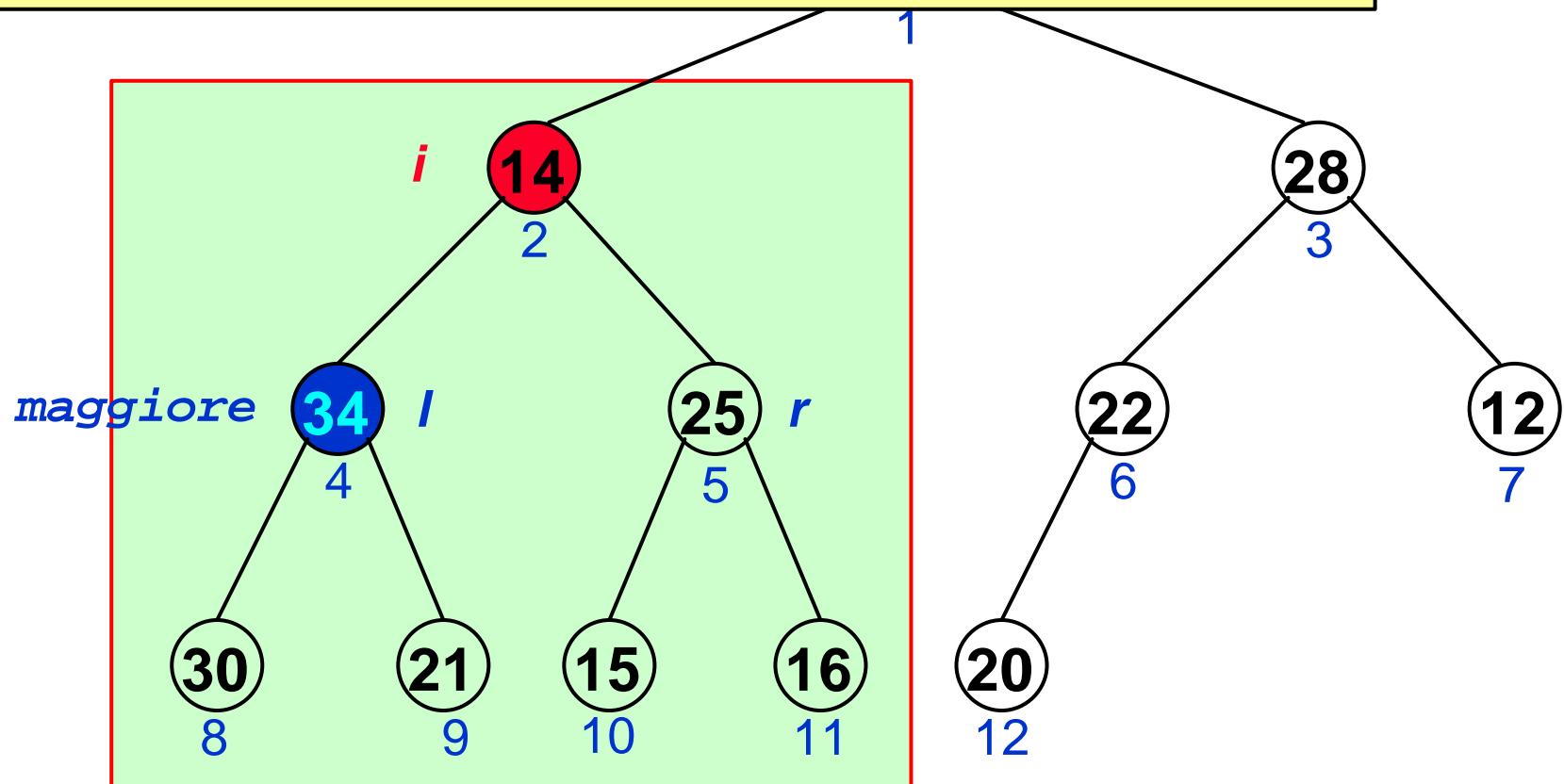
Heapify

```
...
IF  $l \leq \text{heapsize}[A]$  AND  $A[l] > A[i]$ 
    THEN maggiori =  $l$ 
ELSE maggiori =  $i$ 
...
...
```



Heapify

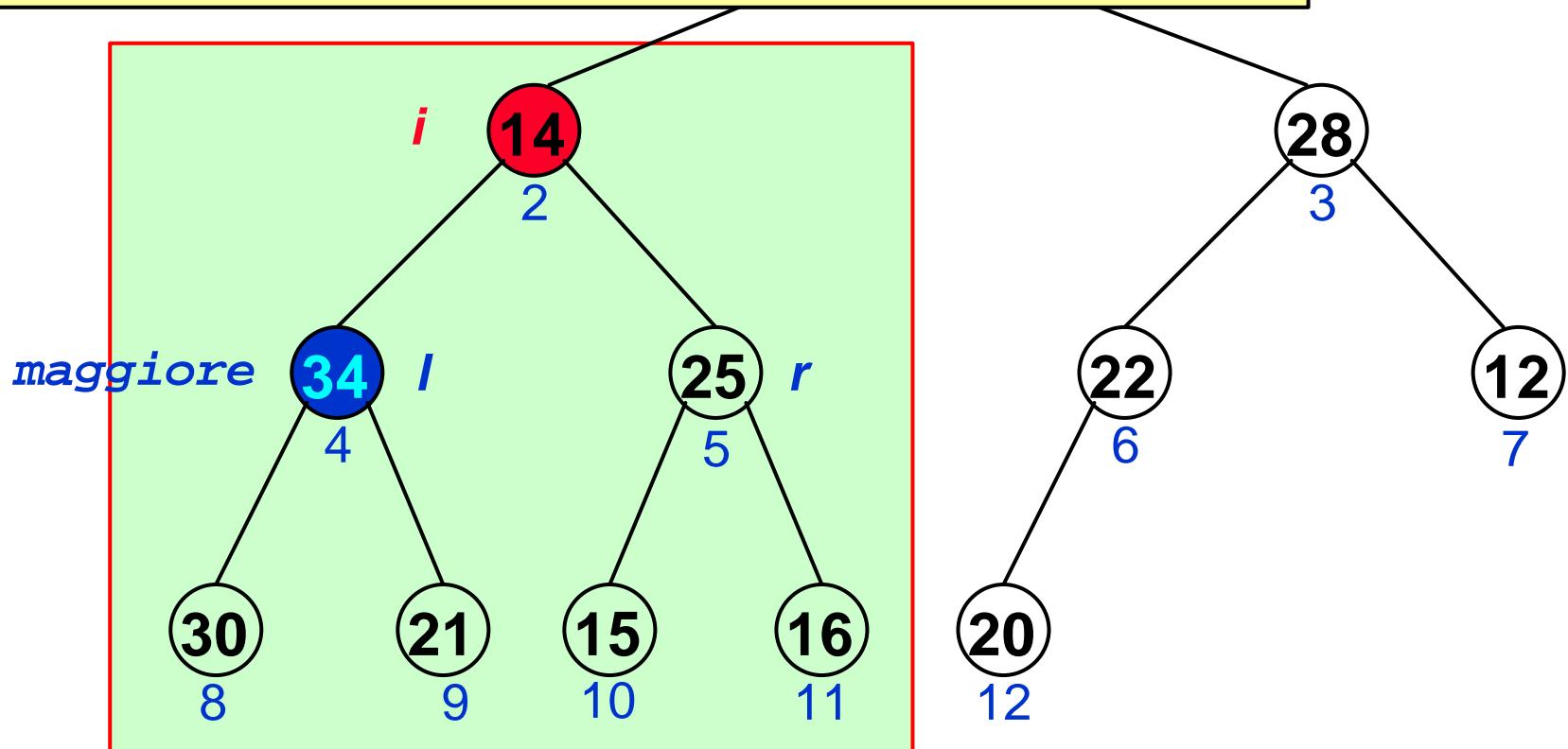
```
...
IF  $r \leq \text{heapsize}[A]$  AND  $A[r] > A[maggiore]$ 
    THEN maggiore =  $r$ 
...
```



Heapify

```
...
IF maggiore > i
    THEN "scambia A[i] e A[maggiore]"
        Heapify(A,maggiore)
...

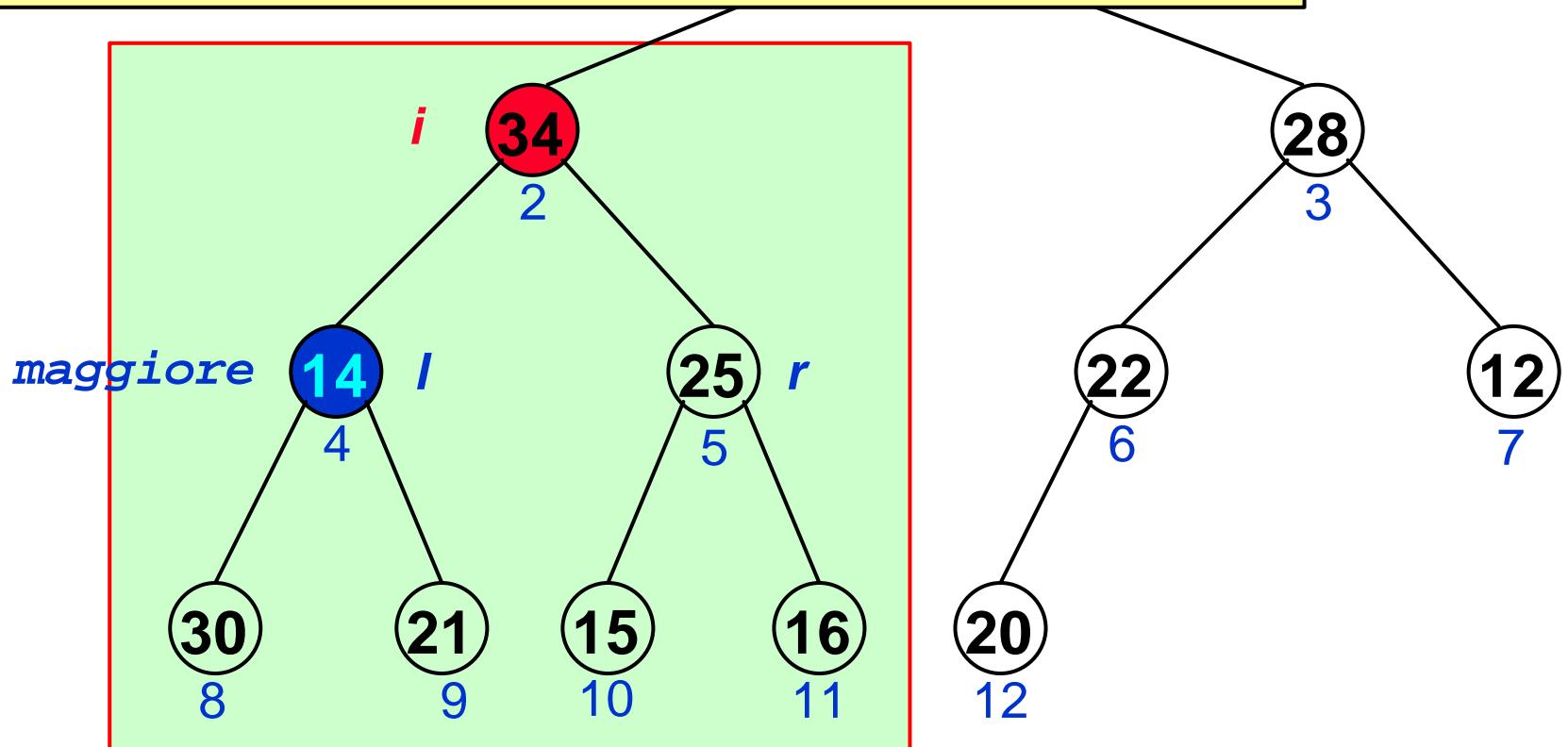
```



Heapify

```
...
IF maggiore 1 i
    THEN "scambia A[i] e A[maggiore]"
        Heapify(A,maggiore)
...

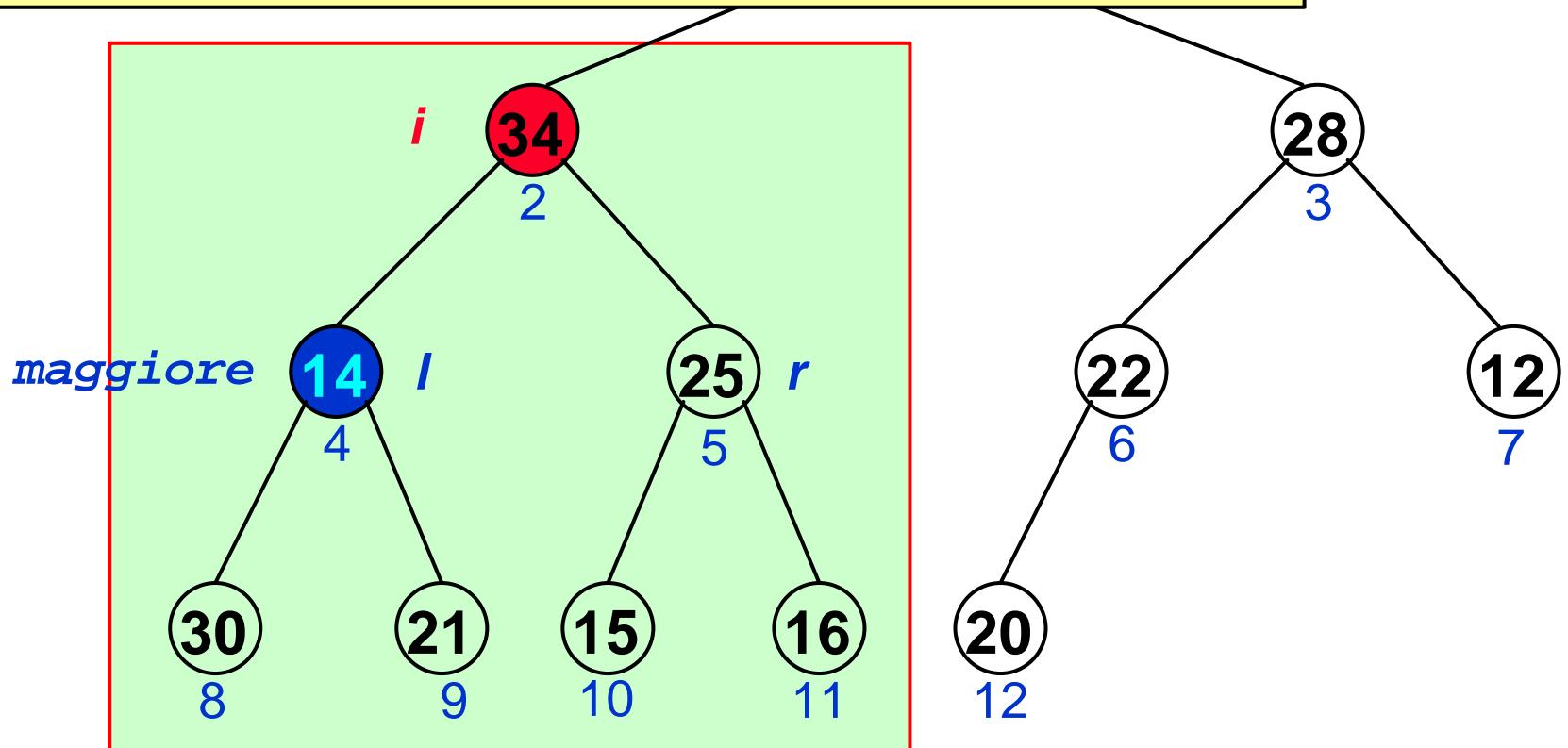
```



Heapify

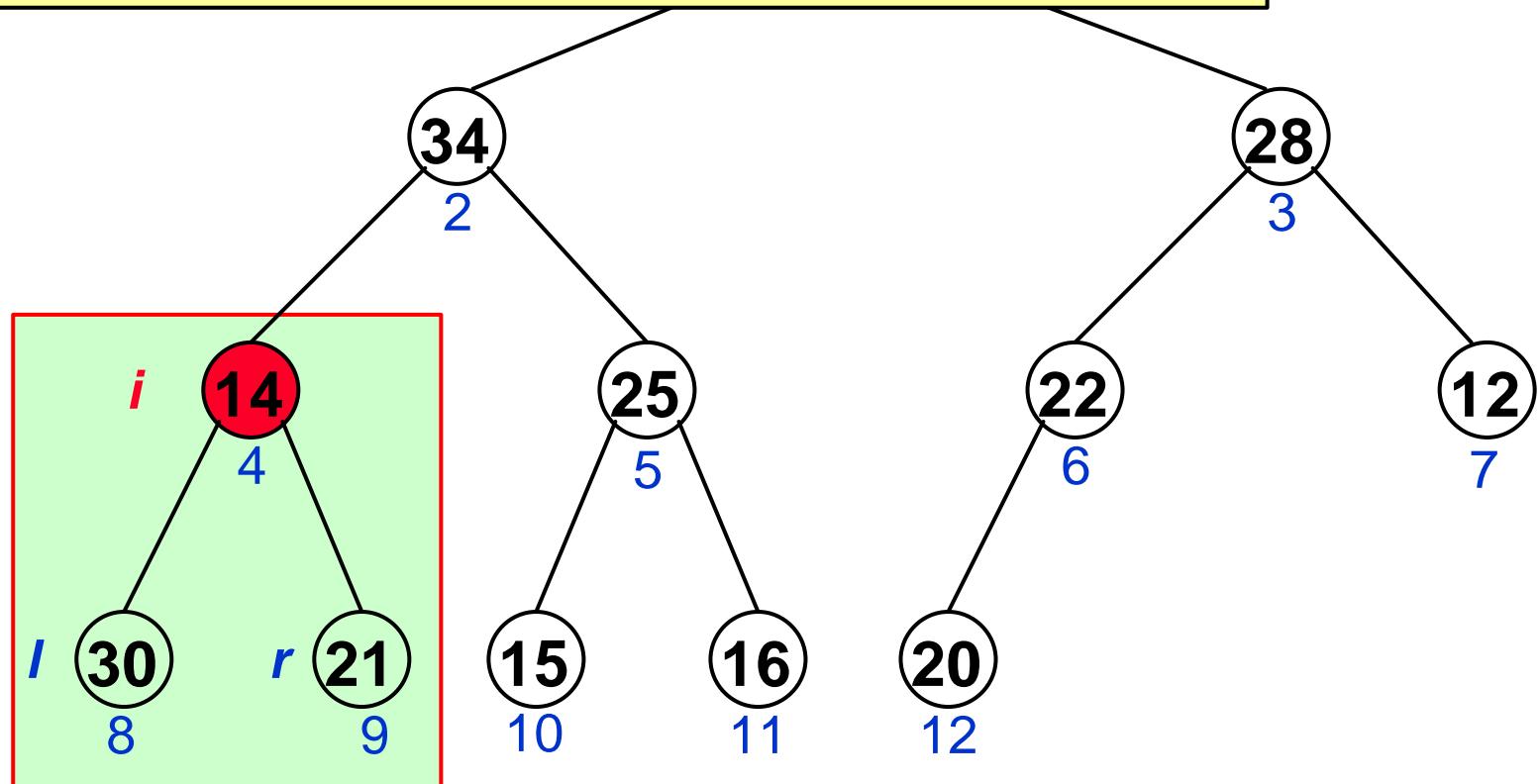
```
...
IF maggiore 1 i
    THEN "scambia A[i] e A[maggiore]"
        Heapify(A,maggiore)
...

```



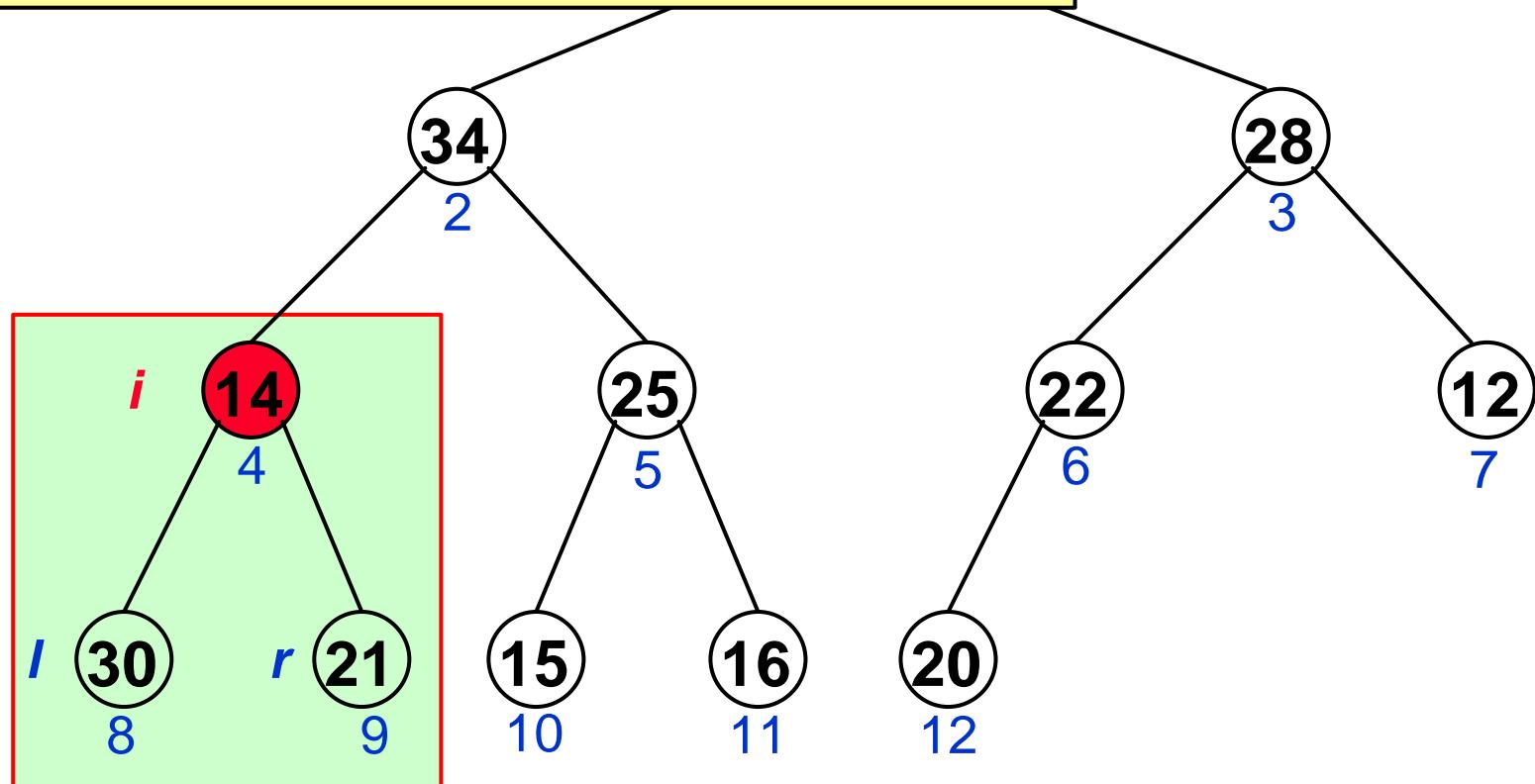
Heapify

```
...
IF maggiore 1 i
    THEN "scambia A[i] e A[maggiore]"
        Heapify(A,maggiore)
...
...
```



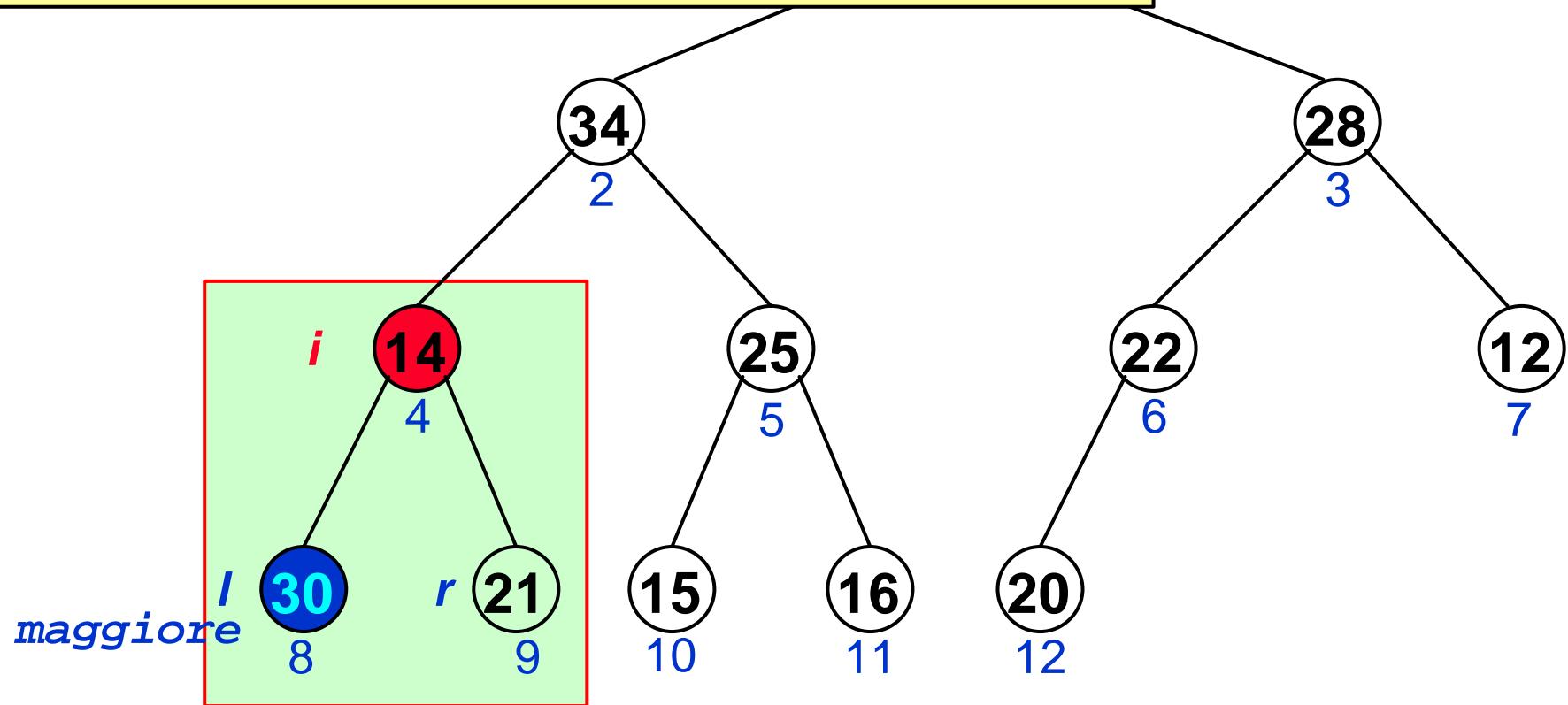
Heapify

```
...
IF  $l \leq \text{heapsize}[A]$  AND  $A[l] > A[i]$ 
    THEN maggiori =  $l$ 
ELSE maggiori =  $i$ 
...
...
```



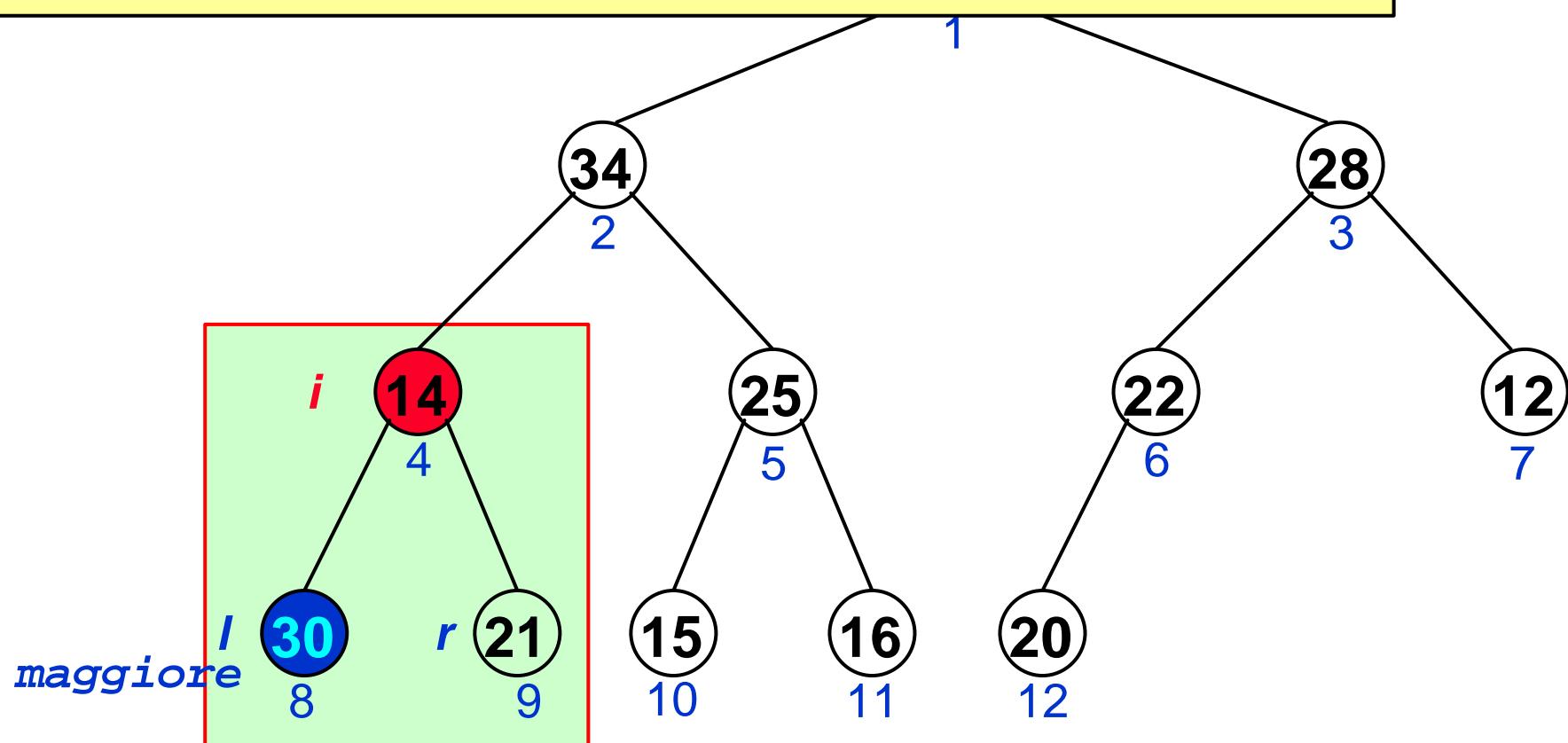
Heapify

```
...
IF  $l \leq \text{heapsize}[A]$  AND  $A[l] > A[i]$ 
    THEN maggiore =  $l$ 
ELSE maggiore =  $i$ 
...
...
```



Heapify

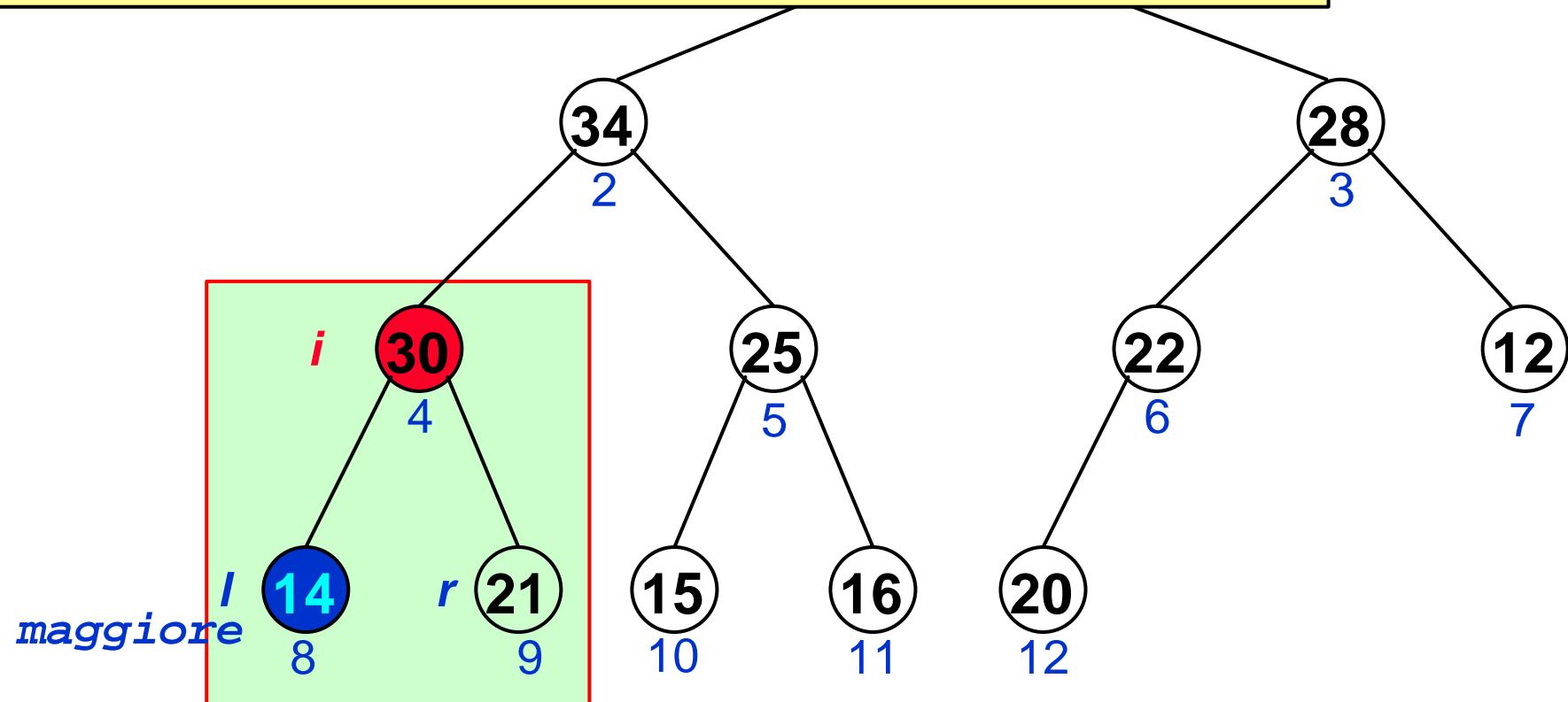
```
...
IF  $r \leq \text{heapsize}[A]$  AND  $A[r] > A[maggiore]$ 
    THEN maggiore = r
...
```



Heapify

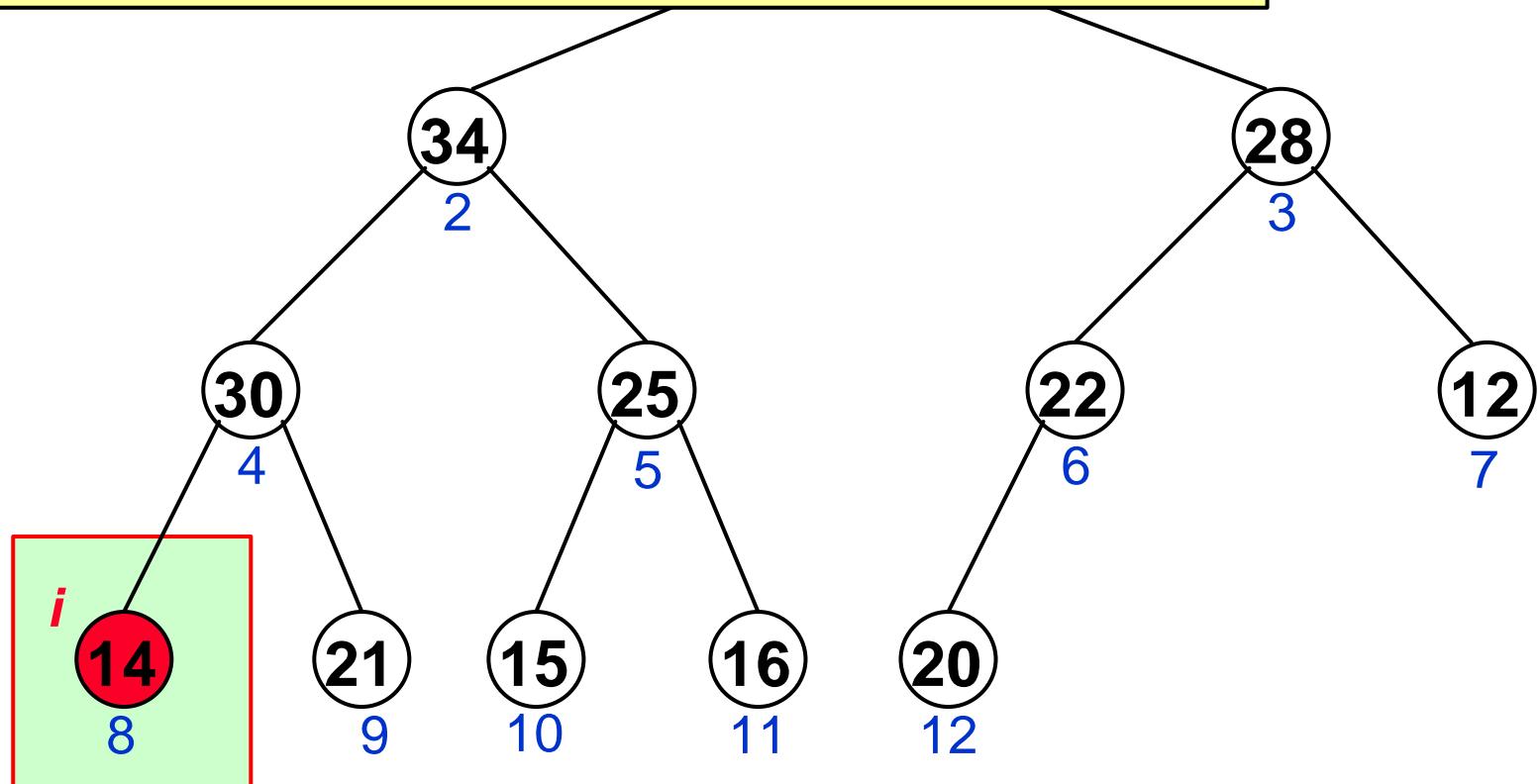
```
...
IF maggiore > i
    THEN "scambia A[i] e A[maggiore]"
        Heapify(A,maggiore)
...

```



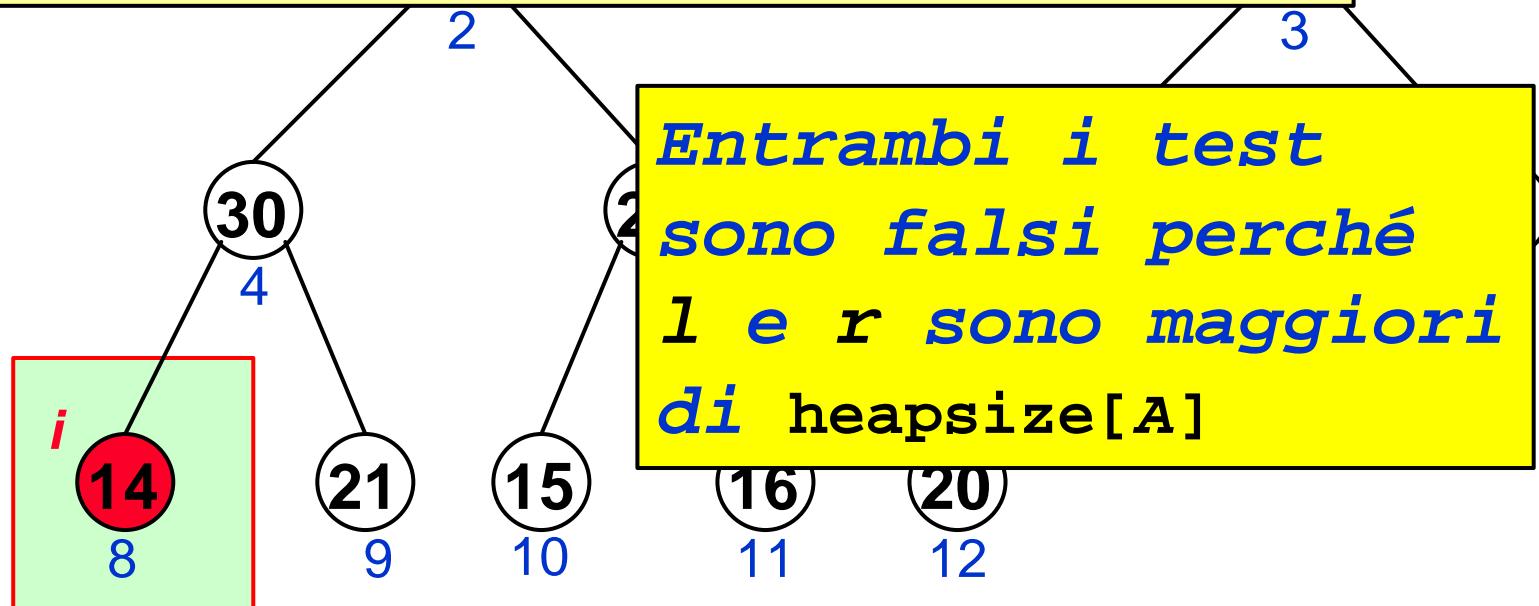
Heapify

```
...
IF maggiore 1 i
    THEN "scambia A[i] e A[maggiore]"
        Heapify(A,maggiore)
...
...
```



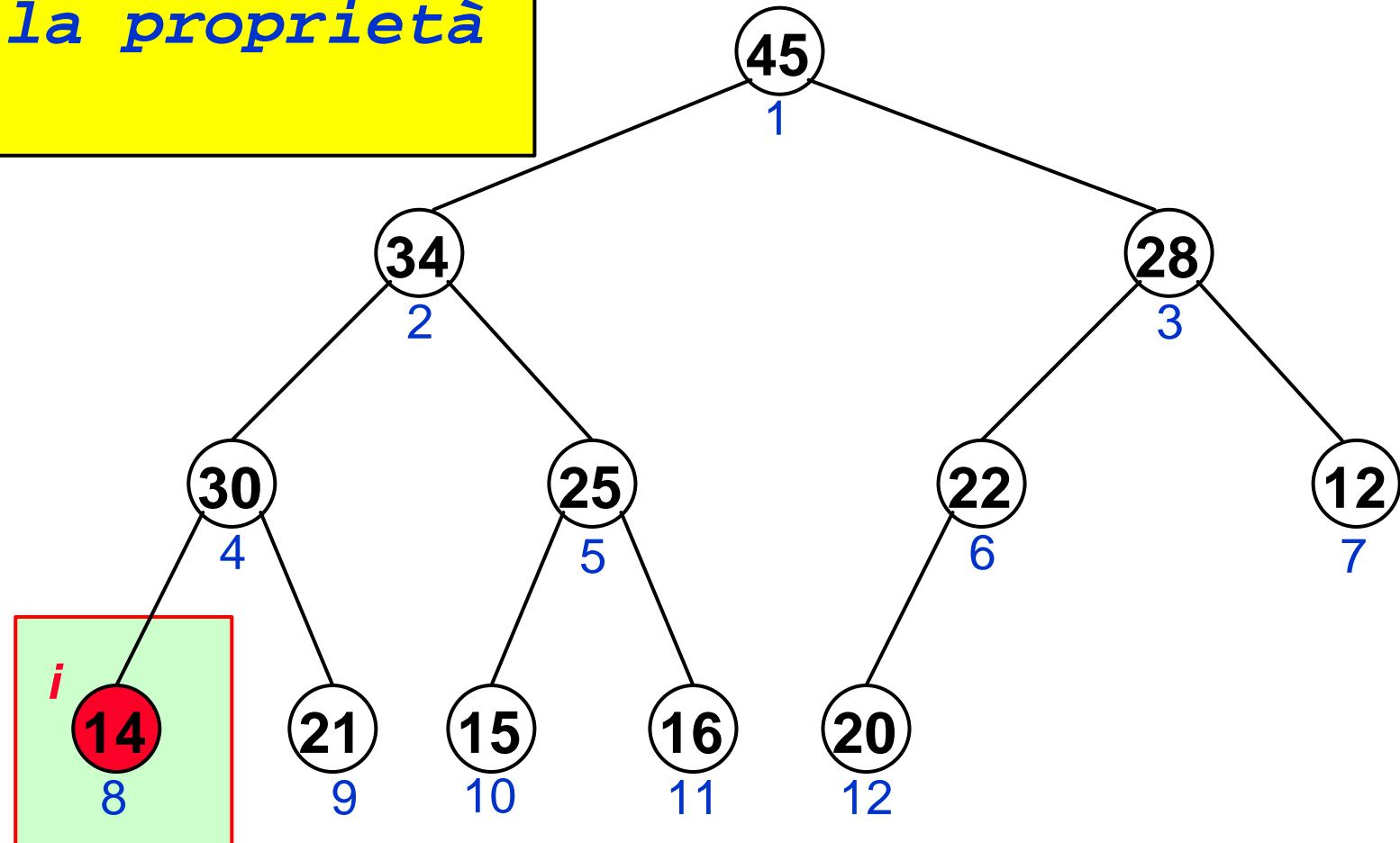
Heapify

```
...
IF  $l \leq \text{heapsize}[A]$  AND  $A[l] > A[i]$ 
    THEN maggiori =  $l$ 
ELSE maggiori =  $i$ 
IF  $r \leq \text{heapsize}[A]$  AND  $A[r] > A[\text{maggiori}]$ 
    THEN maggiori =  $r$ 
...
```



Heapify

Heapify termina!
È stata ripristinata la proprietà Heap



Algoritmi e Strutture Dati

HeapSort II

Complessità di Heapify

$$T(n) = \max(O(1), \max(O(1), T(?)) + O(1))$$

```
    l = SINISTRO(i)
    r = DESTRO(i)
    IF l ≤ heapsize[A] AND A[l] > A[i]
        THEN maggiore = l
    ELSE maggiore = i
    IF r ≤ heapsize[A] AND A[r] > A[maggiore]
        THEN maggiore = r
    IF maggiore ≠ i } = O(1)
        { THEN "scambia A[i] e A[maggiore]"
    T (?) = { Heapify(A, maggiore)
```

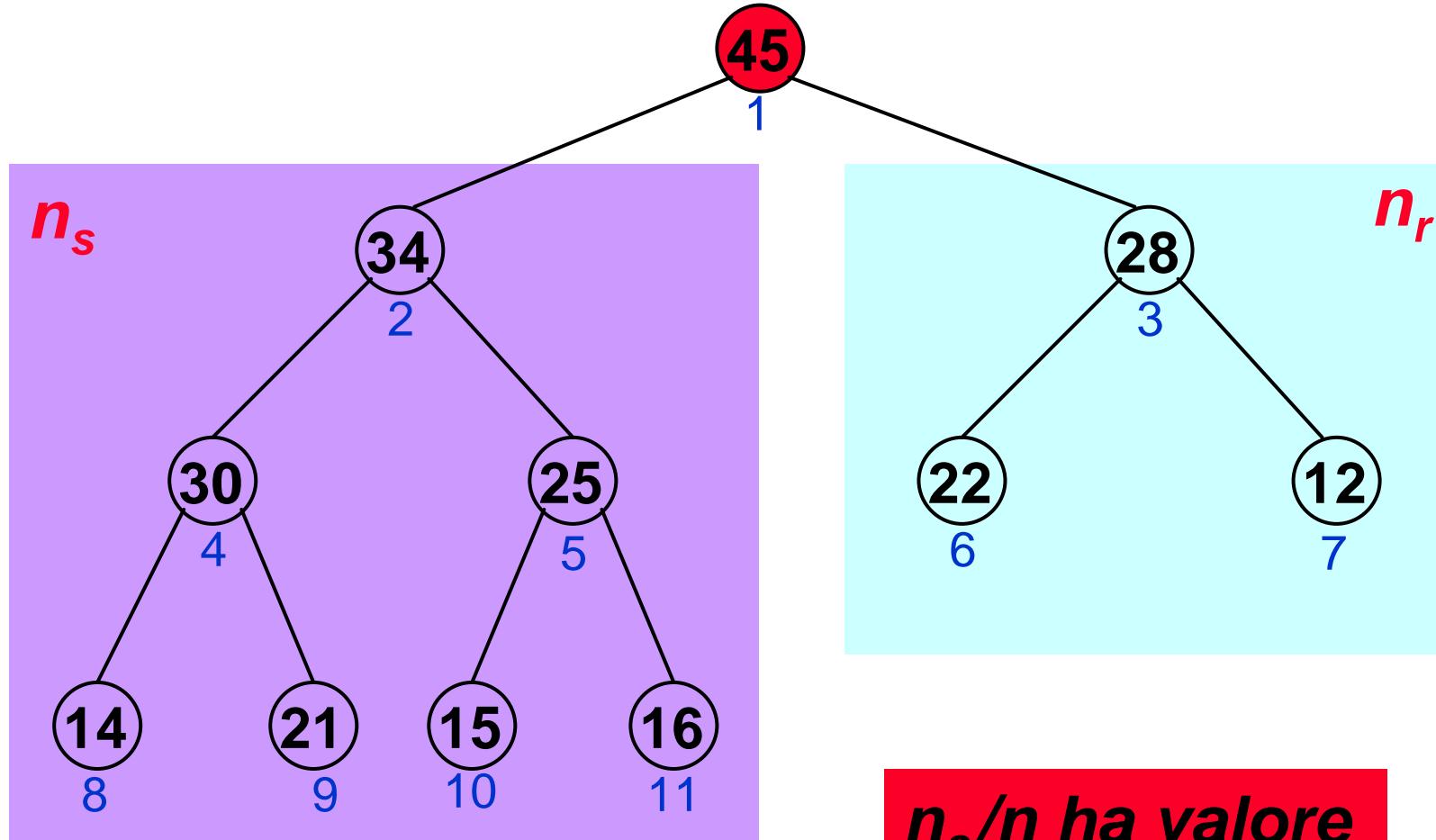
Complessità di Heapify:caso peggiore

$$T(n) = \max(O(1), \max(O(1), T(\cdot) + O(1)))$$

Nel *caso peggiore Heapify* ad ogni chiamata ricorsiva, viene eseguito su un numero di nodi che è minore dei $\frac{2}{3}$ del numero di nodi correnti n .

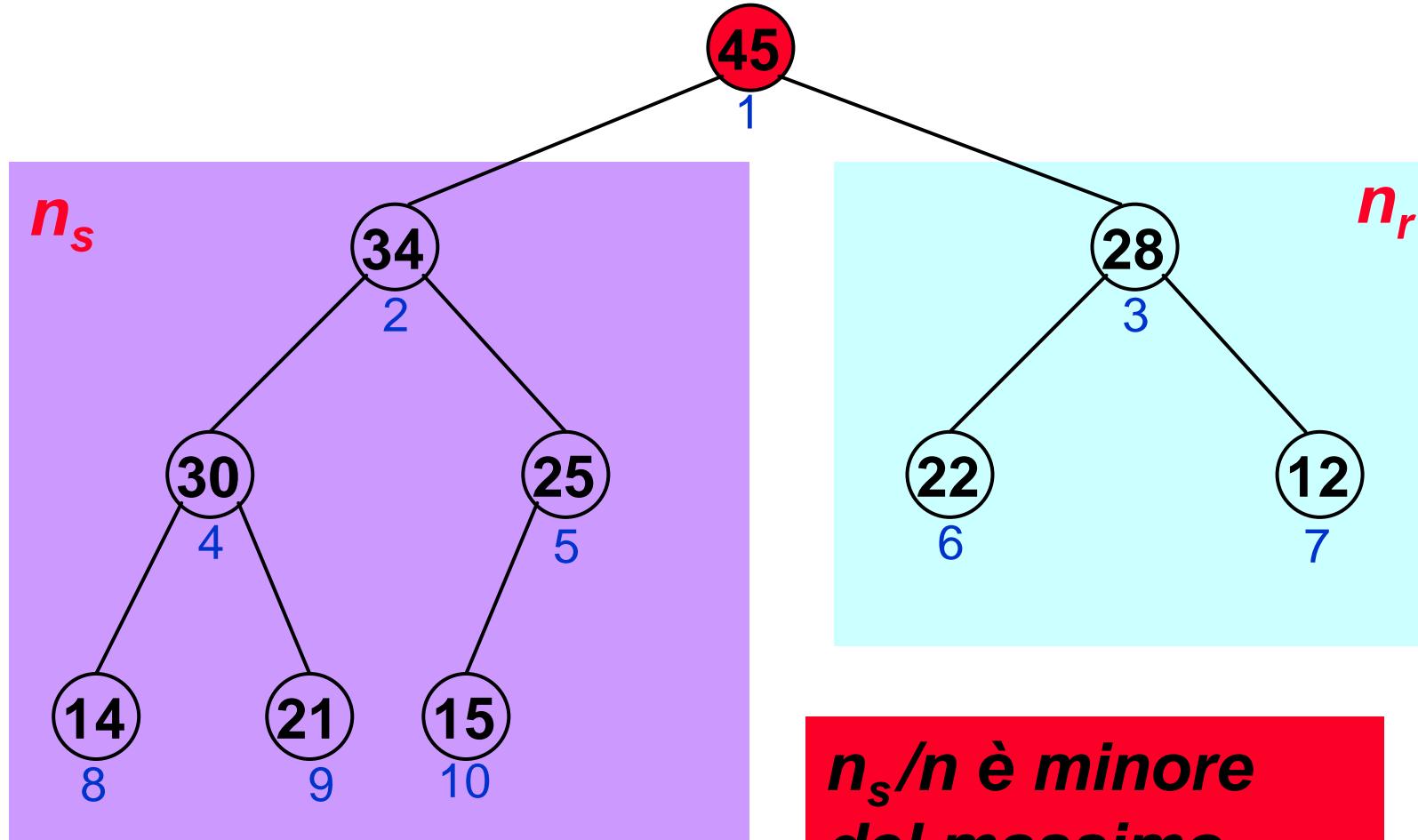
Cioè il numero di nodi n_s del sottoalbero su cui *Heapify* è chiamato ricorsivamente è al più $\frac{2}{3} n$ (o $n_s \leq \frac{2}{3} n$)

Complessità di Heapify:caso peggiore



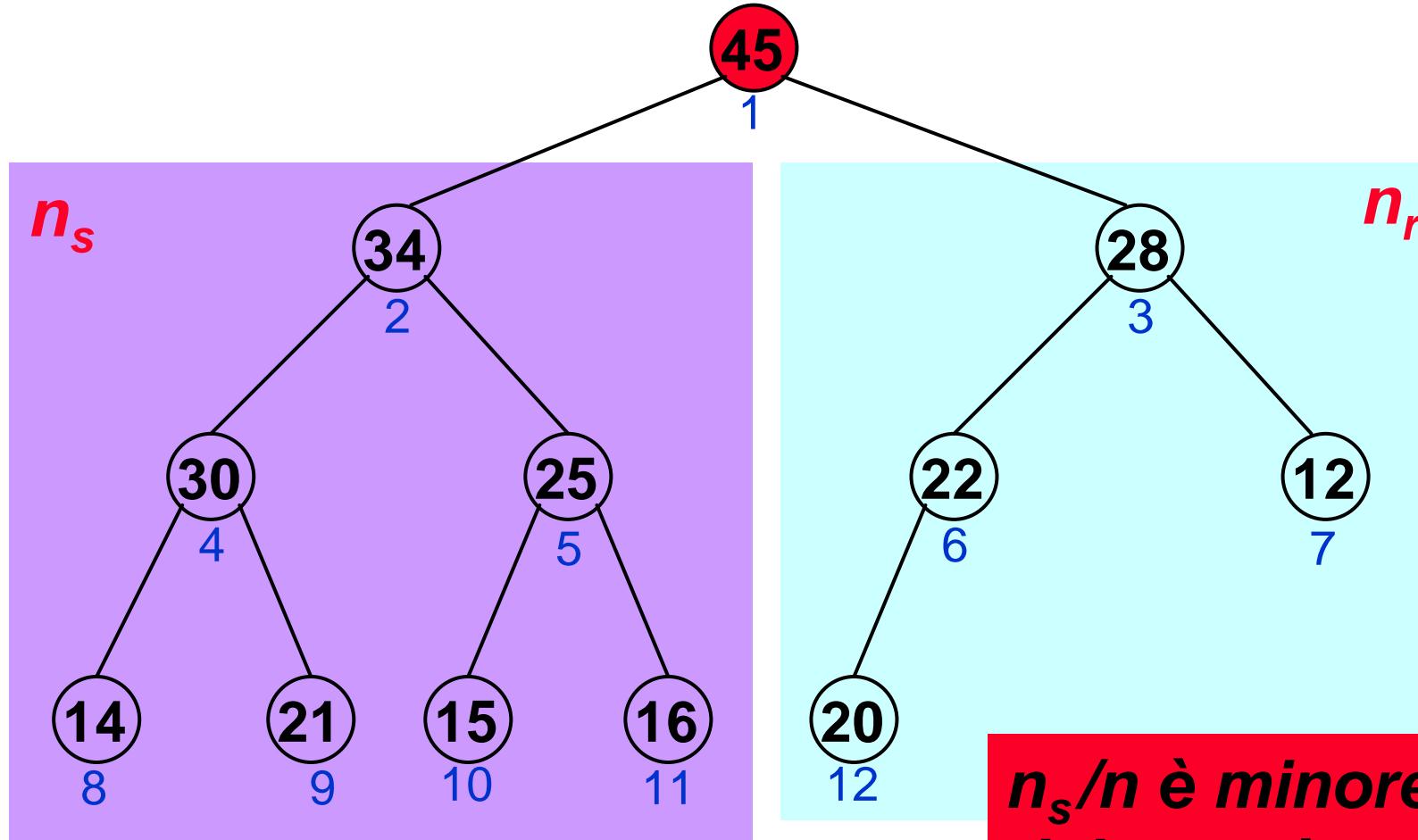
n_s/n ha valore
massimo

Complessità di Heapify:caso peggiore



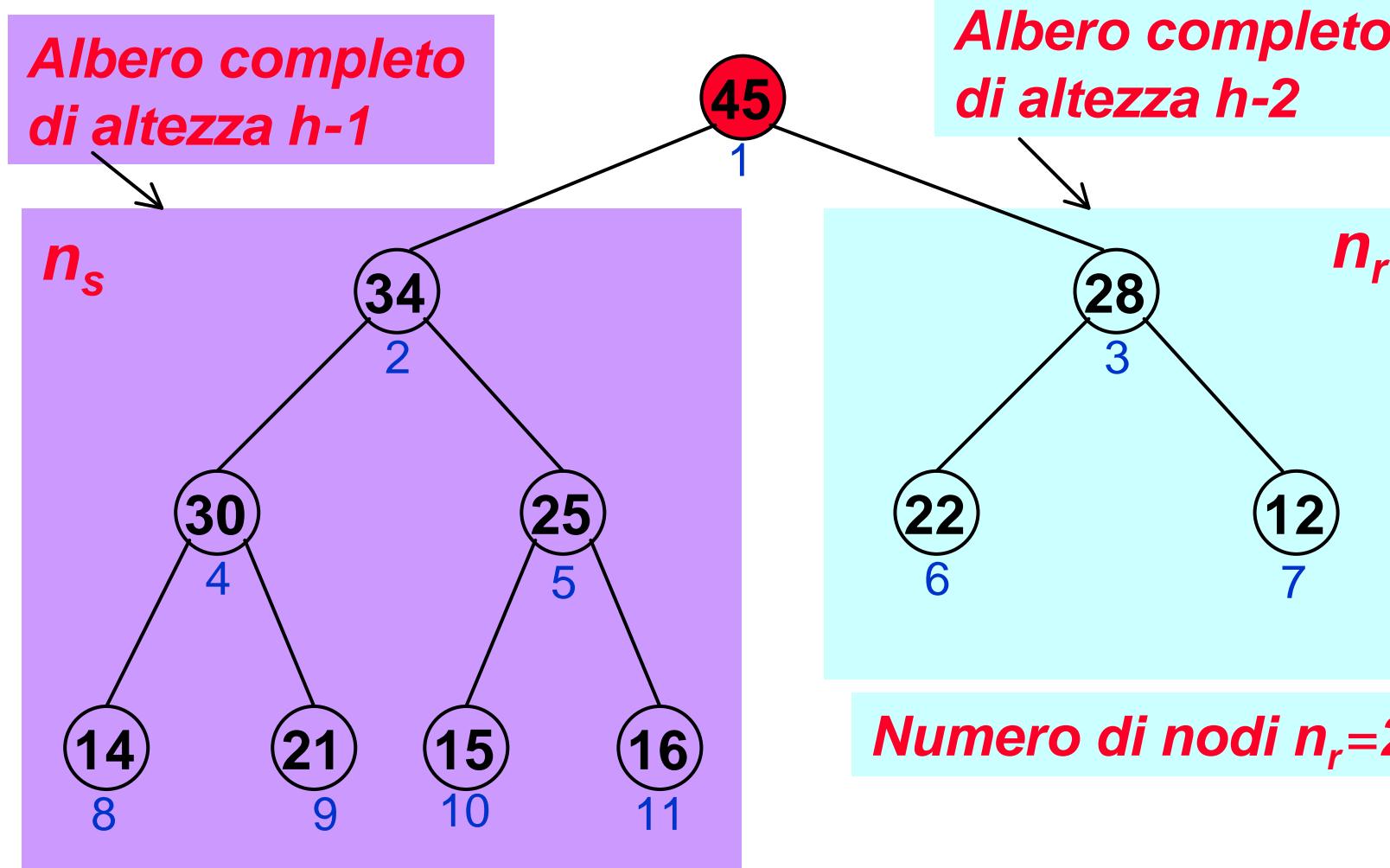
n_s/n è minore
del massimo
(n_s è più piccolo)

Complessità di Heapify:caso peggiore



n_s/n è minore
del massimo
(n è più grande)

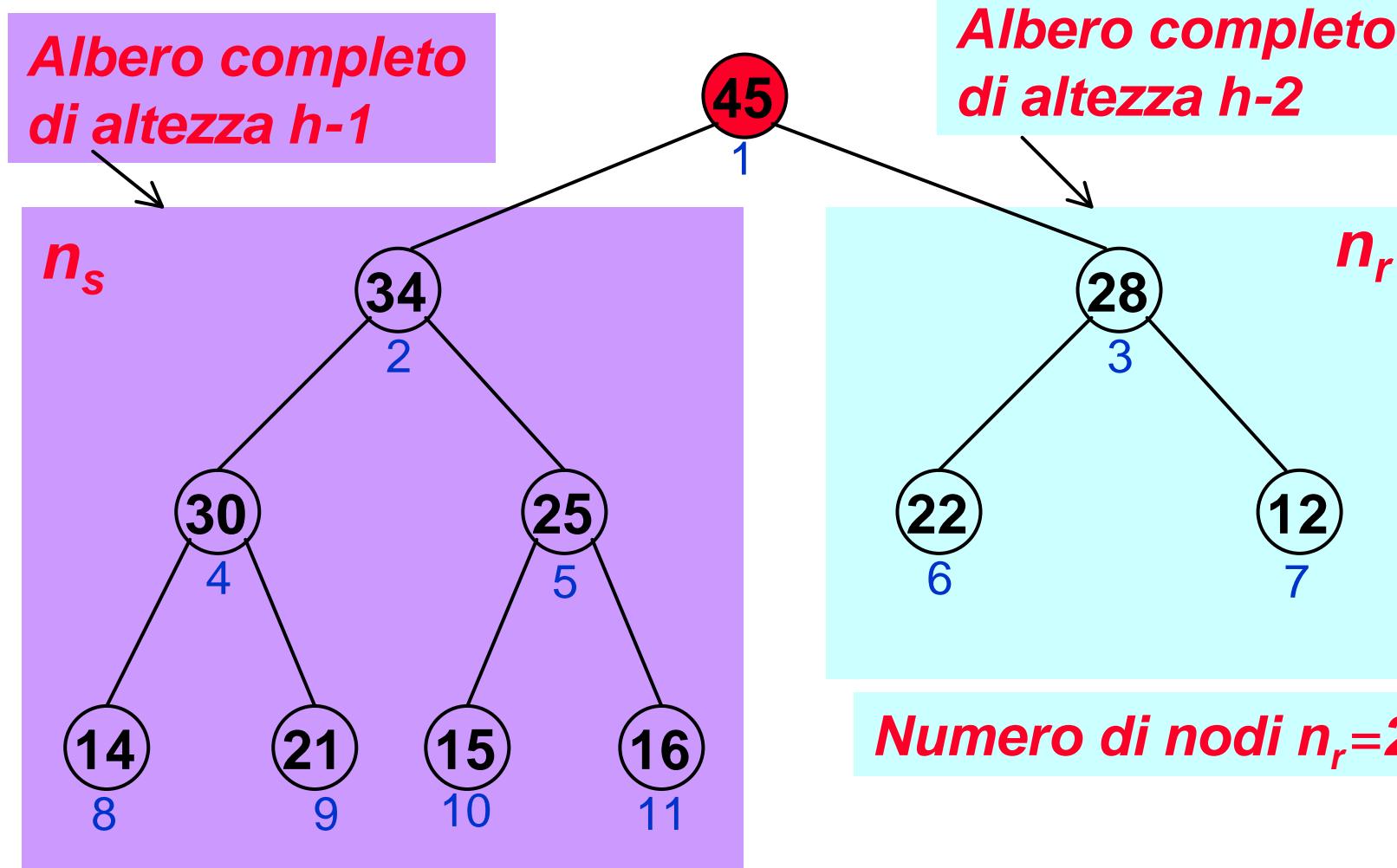
Complessità di Heapify:caso peggiore



Numero di nodi $n_s = 2^{h-1} - 1$

$$n = 1 + 2^{h-1} + 2^{h-1} - 1 = 3 \cdot 2^{h-1} - 1$$

Complessità di Heapify:caso peggiore



Numero di nodi $n_r = 2^{h-1} - 1$

Numero di nodi $n_s = 2^h - 1$

$n_s/n = 2^h - 1 / (3 \cdot 2^{h-1} - 1) \leq 2/3$

Complessità di Heapify:caso peggiore

$$\begin{aligned} T(n) &= \max(O(1), \max(O(1), T(\cdot) + O(1))) \\ &\in \max(O(1), \max(O(1), T(2n/3) + O(1))) \\ &\in T(2n/3) + Q(1) \end{aligned}$$

$$T'(n) = T'(2n/3) + Q(1)$$

Proviamo ad applicare il Metodo Iterativo!

$$T'(n) = Q(\log n)$$

Complessità di Heapify:caso peggiore

$$T(n) = \max(O(1), \max(O(1), T(?) + O(1)))$$

$$\in \max(O(1), \max(O(1), T(2n/3) + O(1)))$$

$$\in T(2n/3) + O(1)$$

Quindi

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

**Heapify impiega tempo proporzionale
all'altezza dell'albero su cui opera !**

Complessità di Heapify:caso migliore

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

Nel **caso migliore Heapify** ad ogni chiamata ricorsiva, viene eseguito su un numero di nodi che è maggiore di **1/3** del numero di nodi correnti **n** .

Cioè il numero di nodi **n_s** del sottoalbero su cui **Heapify** è chiamato ricorsivamente è al più **1/3 n** (o **$n_s \geq 1/3 n$**)

Complessità di Heapify:caso migliore

$$\begin{aligned} T(n) &= T(?) + O(1) \\ &\geq T(n/3) + Q(1) \end{aligned}$$

$$T'(n) = T'(n/3) + Q(1)$$

Applicando il Metodo Iterativo!

$$T'(n) = Q(\log n)$$

quindi

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

Costruisci Heap: intuizioni

Costruisci-Heap(A): utilizza l'algoritmo **Heapify**, per inserire ogni elemento dell'array in uno **Heap**, risistemando sul posto gli elementi:

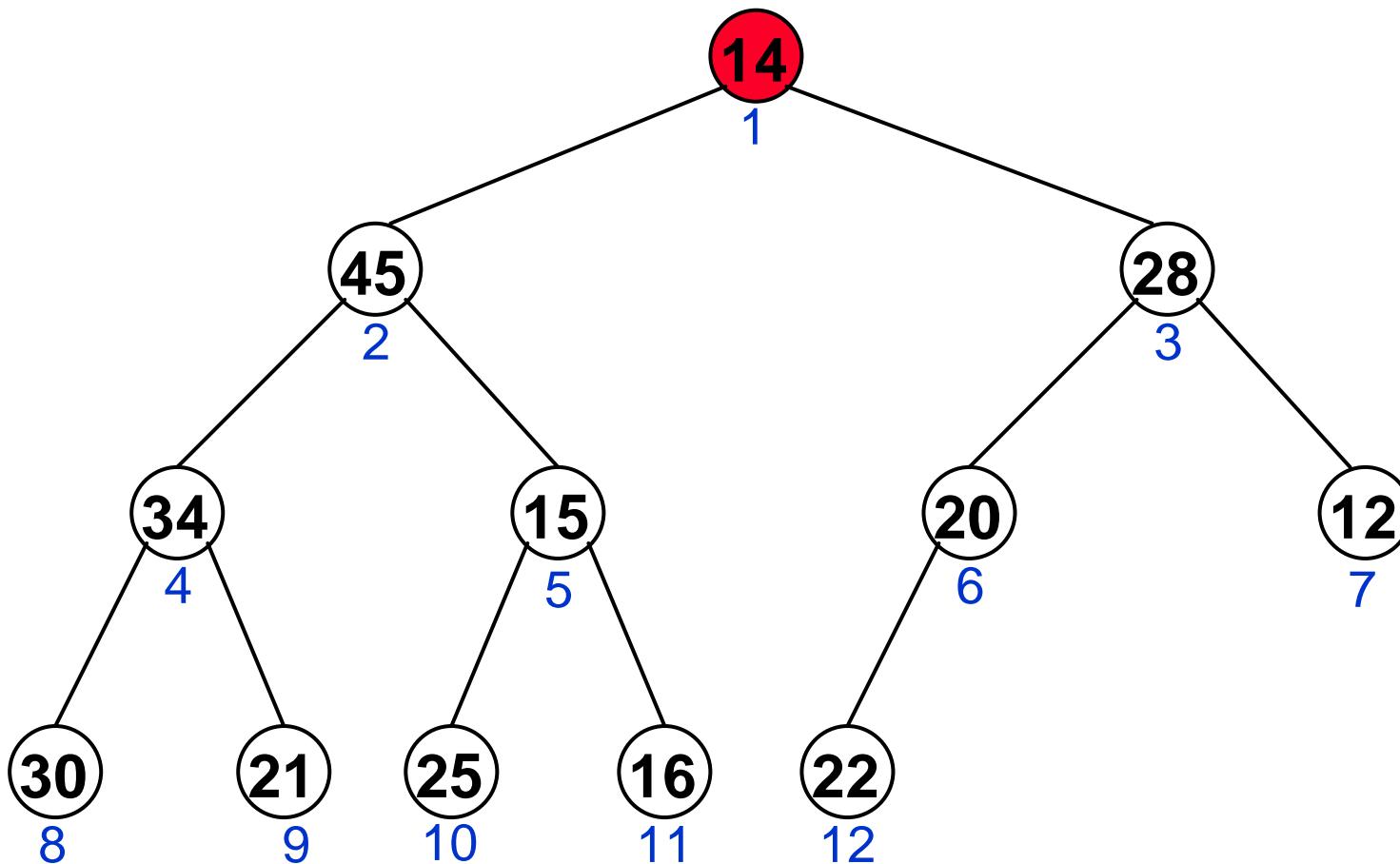
- gli ultimi $\lceil n/2 \rceil$ elementi dell'array sono foglie, cioè radici di sottoalberi vuoti, quindi sono già degli **Heap**
- è sufficiente inserire nello **Heap** solo i primi $\lceil n/2 \rceil$ elementi, utilizzando **Heapify** per ripristinare la proprietà **Heap** sul sottoalbero del nuovo elemento.

Costruisci Heap

```
Costruisci-Heap(A)
    heapsize[A] = length[A]
    FOR i = ¢length[A]/2ù DOWNTO 1
        DO Heapify(A,i)
```

Costruisci Heap

1	2	3	4	5	6	7	8	9	10	11	12
14	45	28	34	15	20	12	30	21	25	16	22



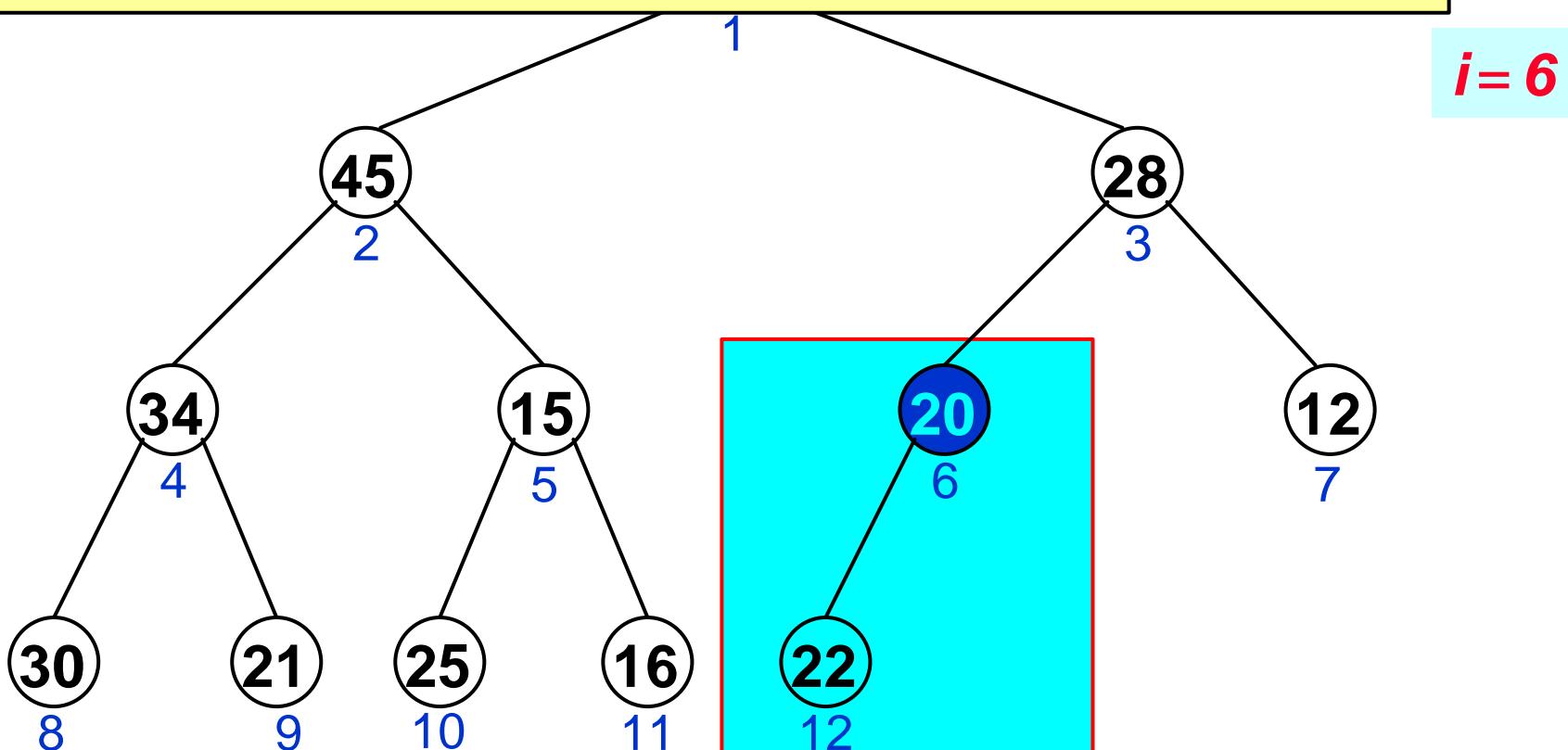
Costruisci Heap

Costruisci-Heap(*A*)

heapsize[*A*] = length[*A*]

FOR *i* = $\lceil \text{length}[A]/2 \rceil$ DOWNTO 1

DO Heapify(*A*, *i*)



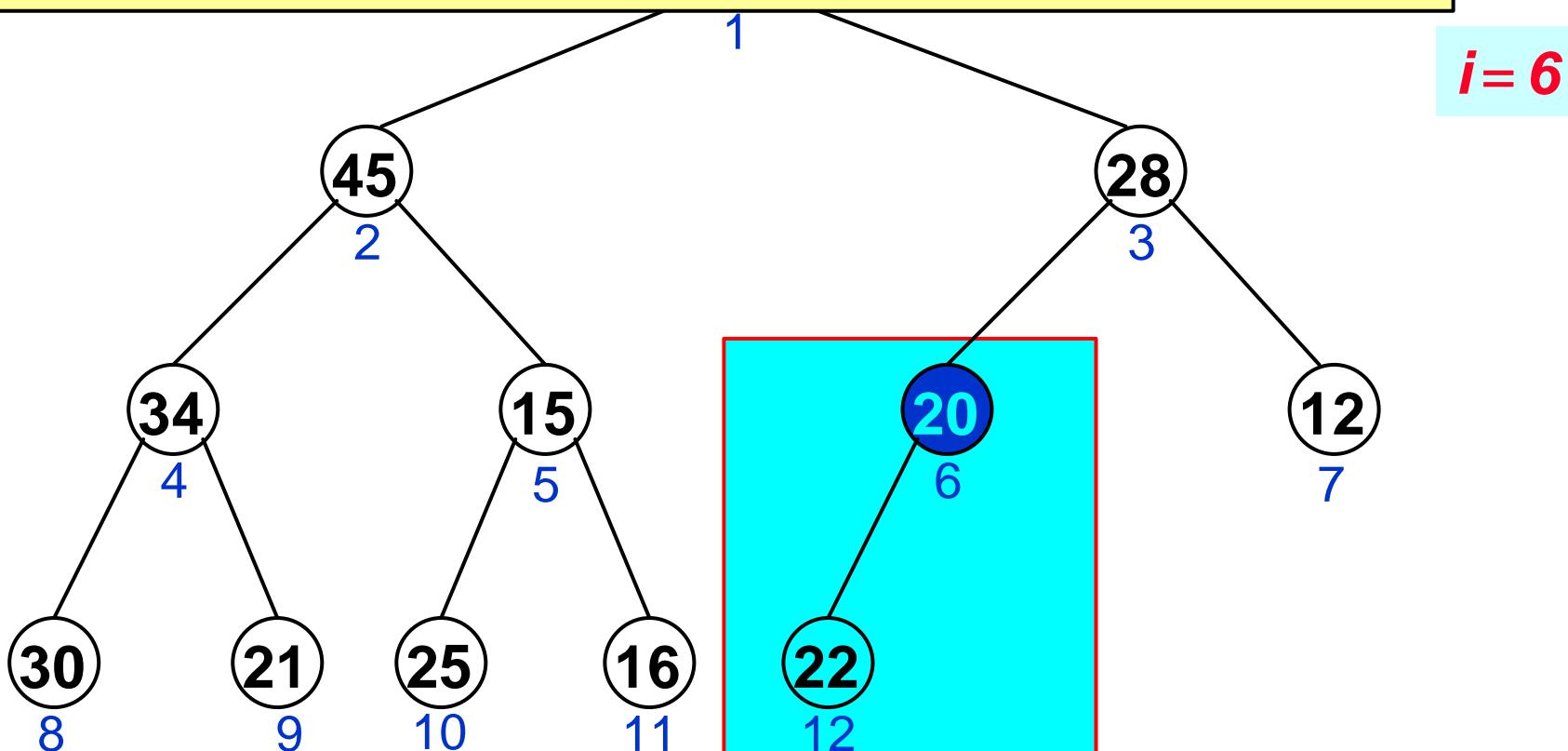
Costruisci Heap

Costruisci-Heap(*A*)

heapsize[*A*] = length[*A*]

FOR i = $\lceil \text{length}[A]/2 \rceil$ DOWNTO 1

DO Heapify(*A*, *i*)



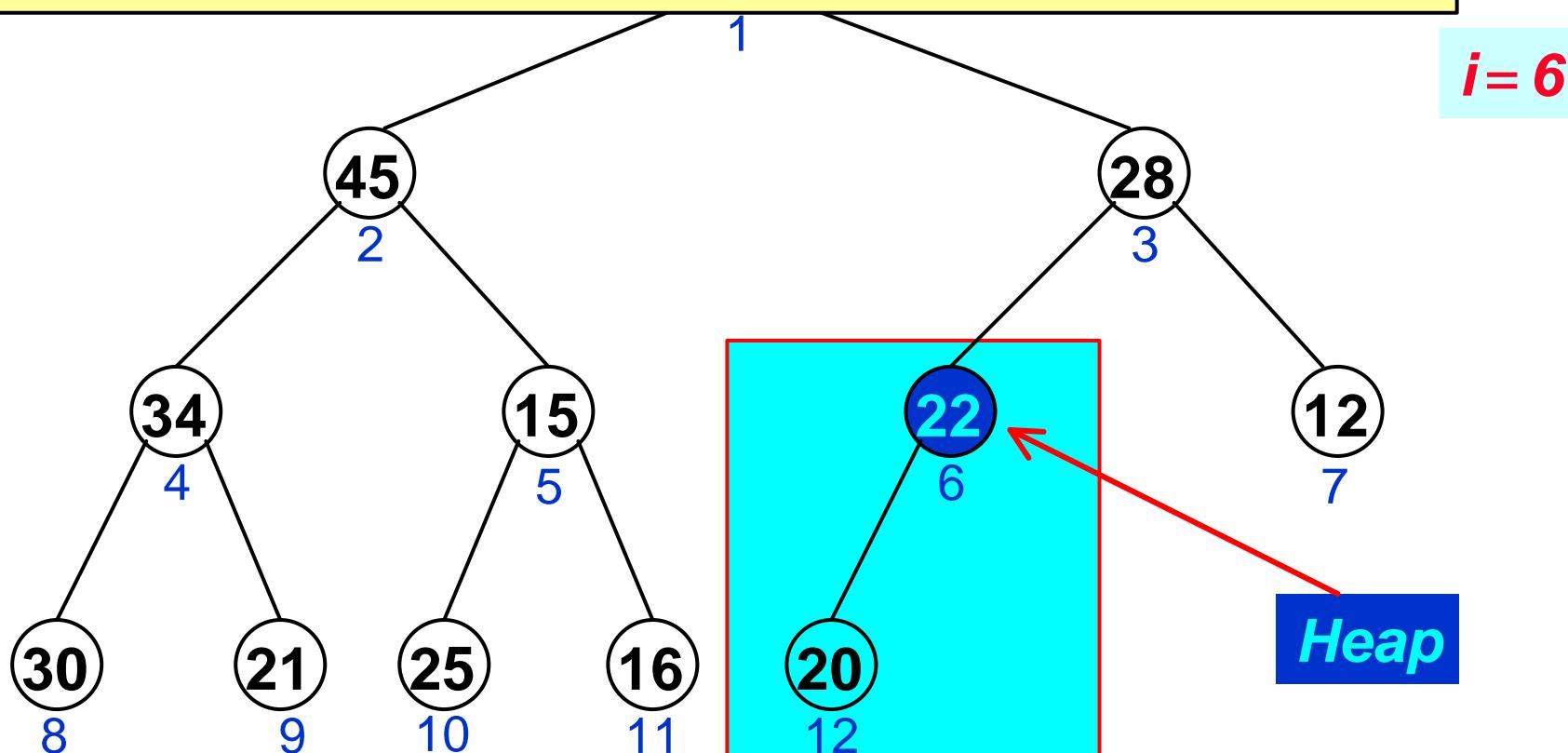
Costruisci Heap

Costruisci-Heap(*A*)

heapsize[*A*] = length[*A*]

FOR i = $\lceil \text{length}[A]/2 \rceil$ DOWNTO 1

DO Heapify(*A*, *i*)



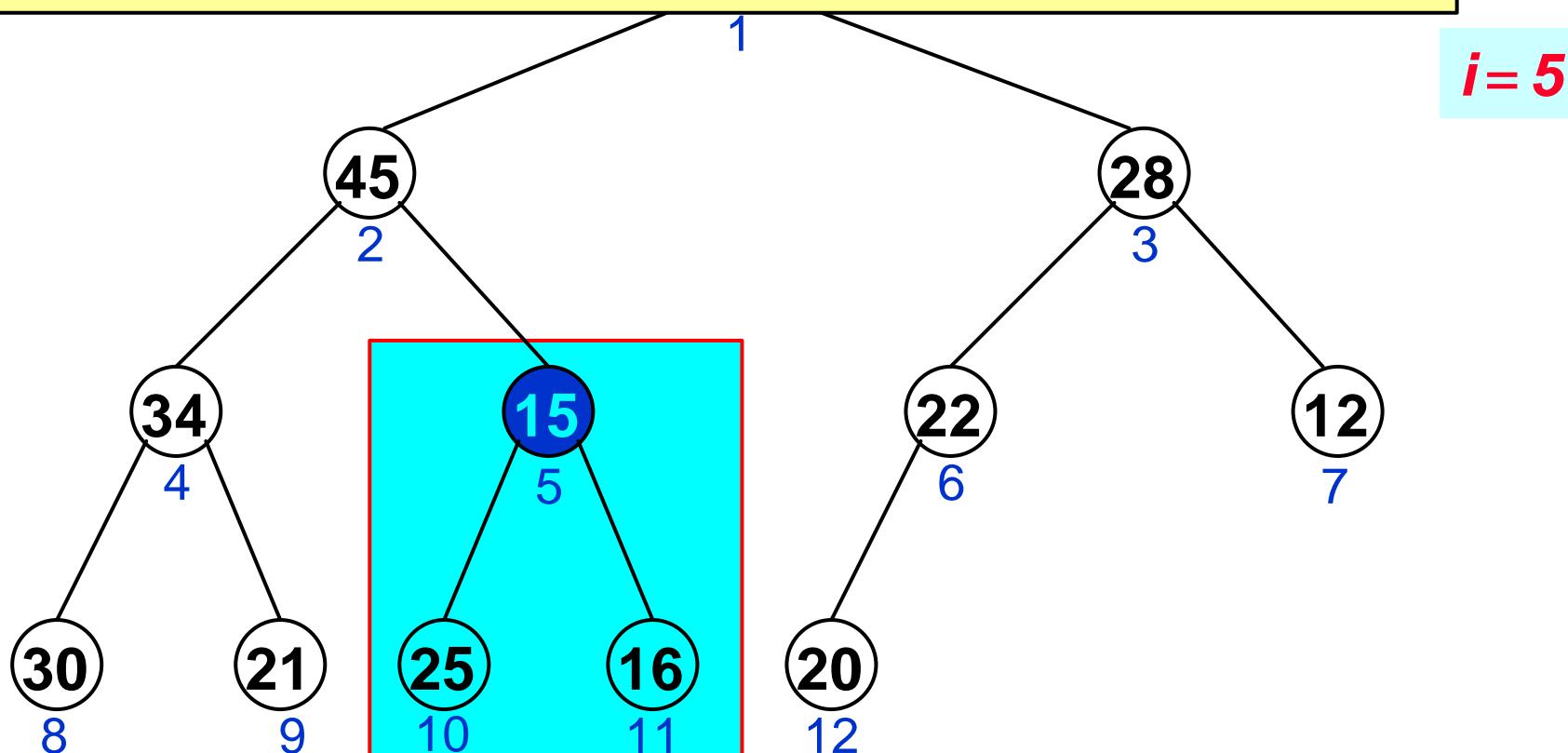
Costruisci Heap

Costruisci-Heap(*A*)

heapsize[*A*] = length[*A*]

FOR *i* = $\lceil \text{length}[A]/2 \rceil$ DOWNTO 1

DO Heapify(*A*, *i*)



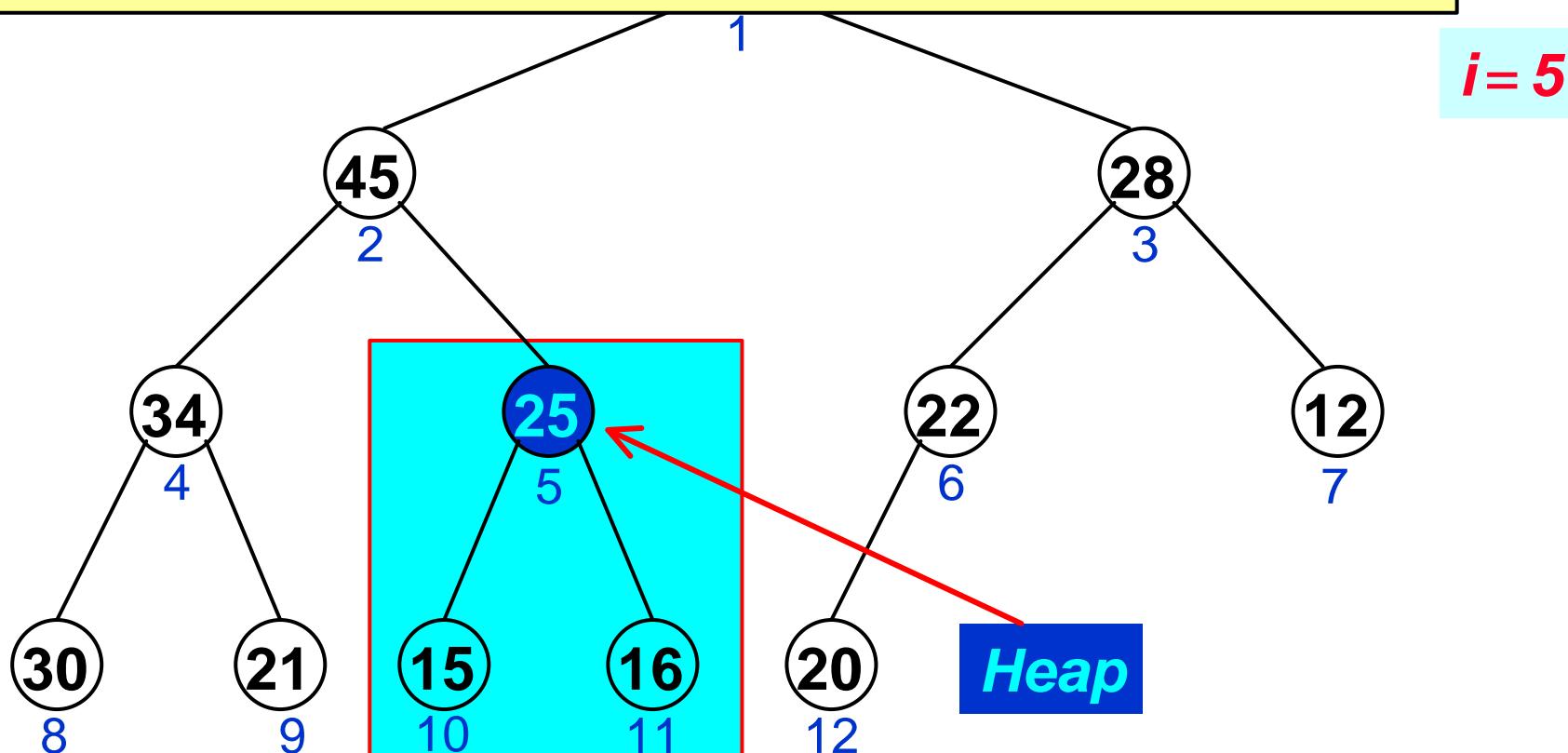
Costruisci Heap

Costruisci-Heap(*A*)

heapsize[*A*] = length[*A*]

FOR i = $\lceil \text{length}[A]/2 \rceil$ DOWNTO 1

DO Heapify(*A*, *i*)



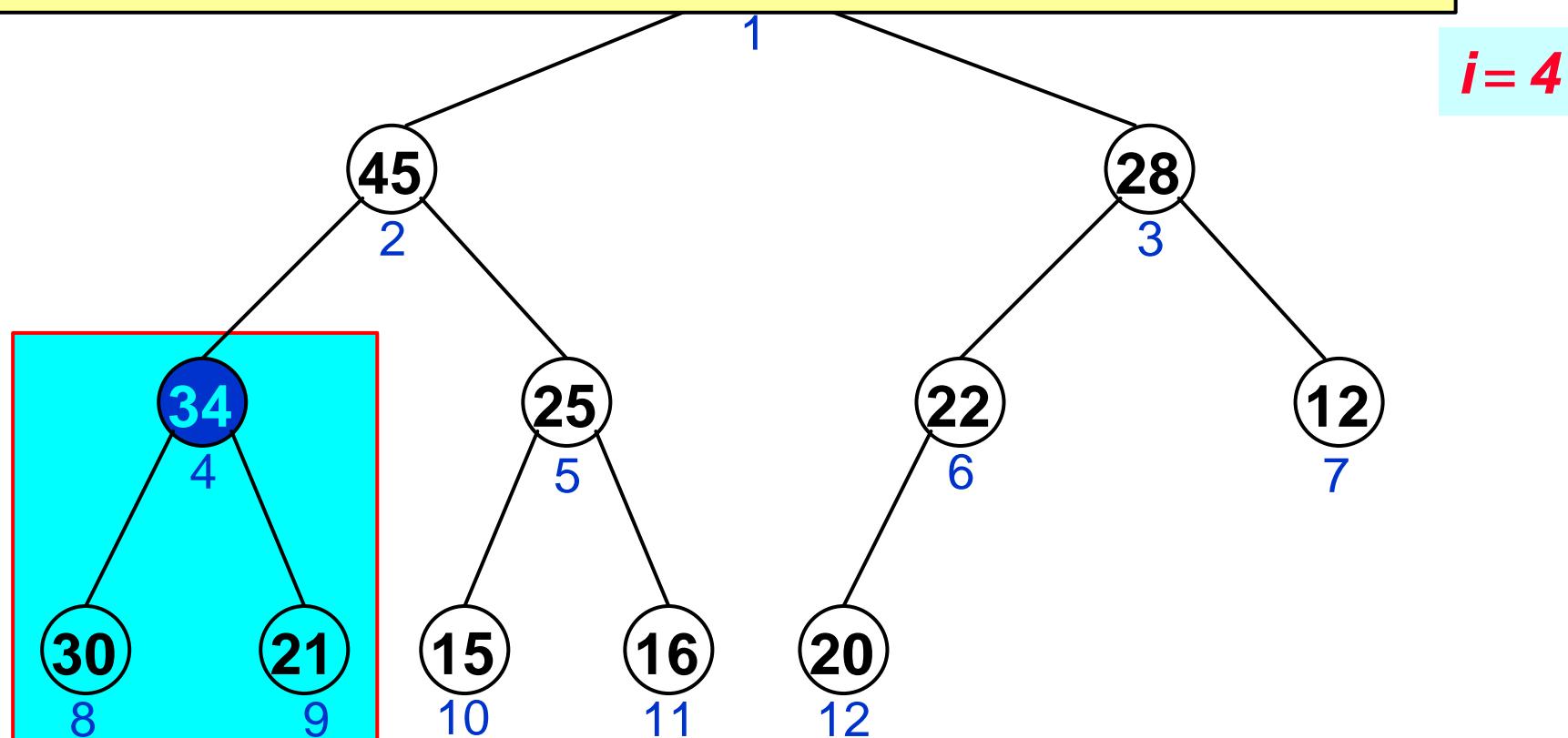
Costruisci Heap

Costruisci-Heap(*A*)

heapsize[*A*] = length[*A*]

FOR *i* = $\lceil \text{length}[A]/2 \rceil$ DOWNTO 1

DO Heapify(*A*, *i*)



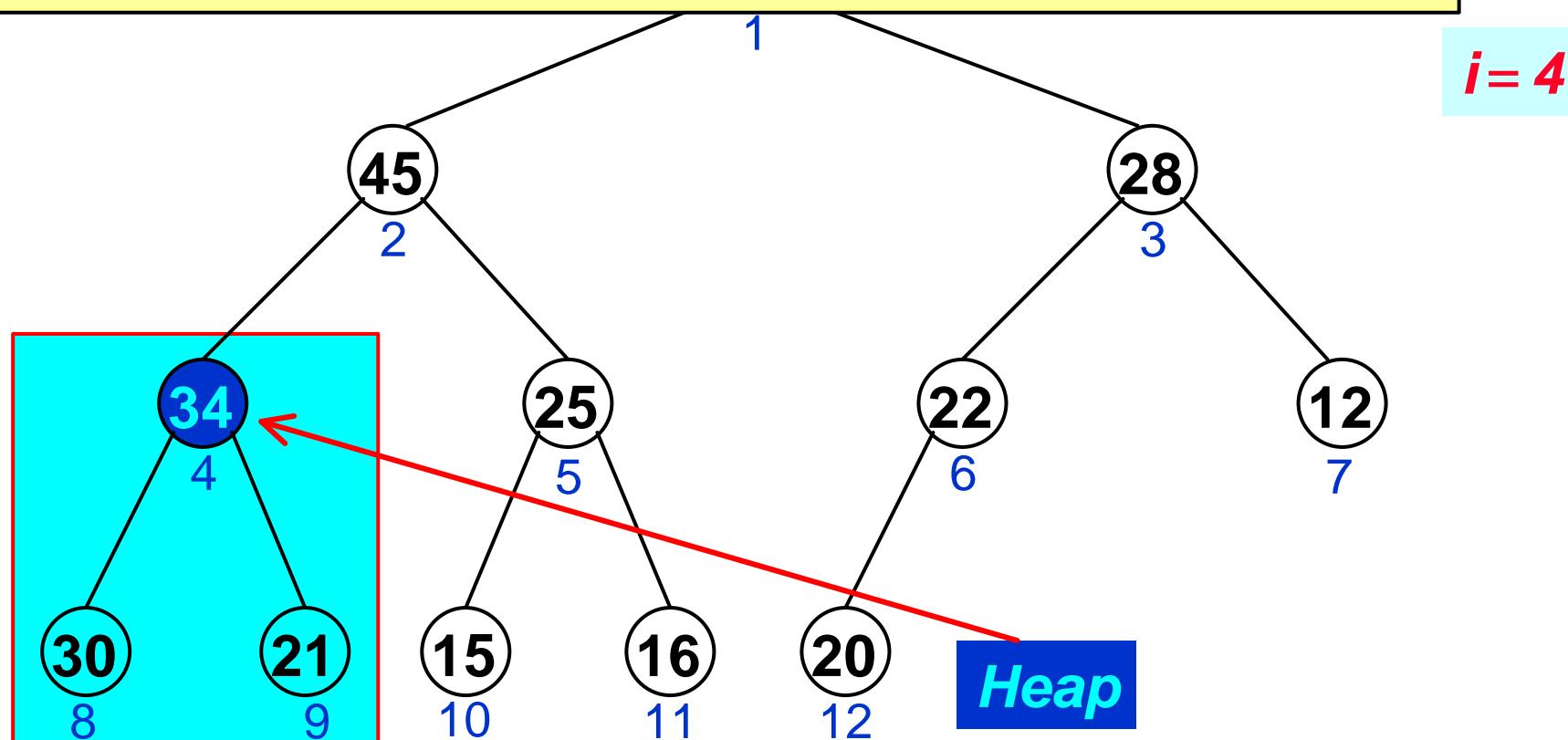
Costruisci Heap

Costruisci-Heap(*A*)

heapsize[*A*] = length[*A*]

FOR i = $\lceil \text{length}[A]/2 \rceil$ DOWNTO 1

DO Heapify(*A*, *i*)



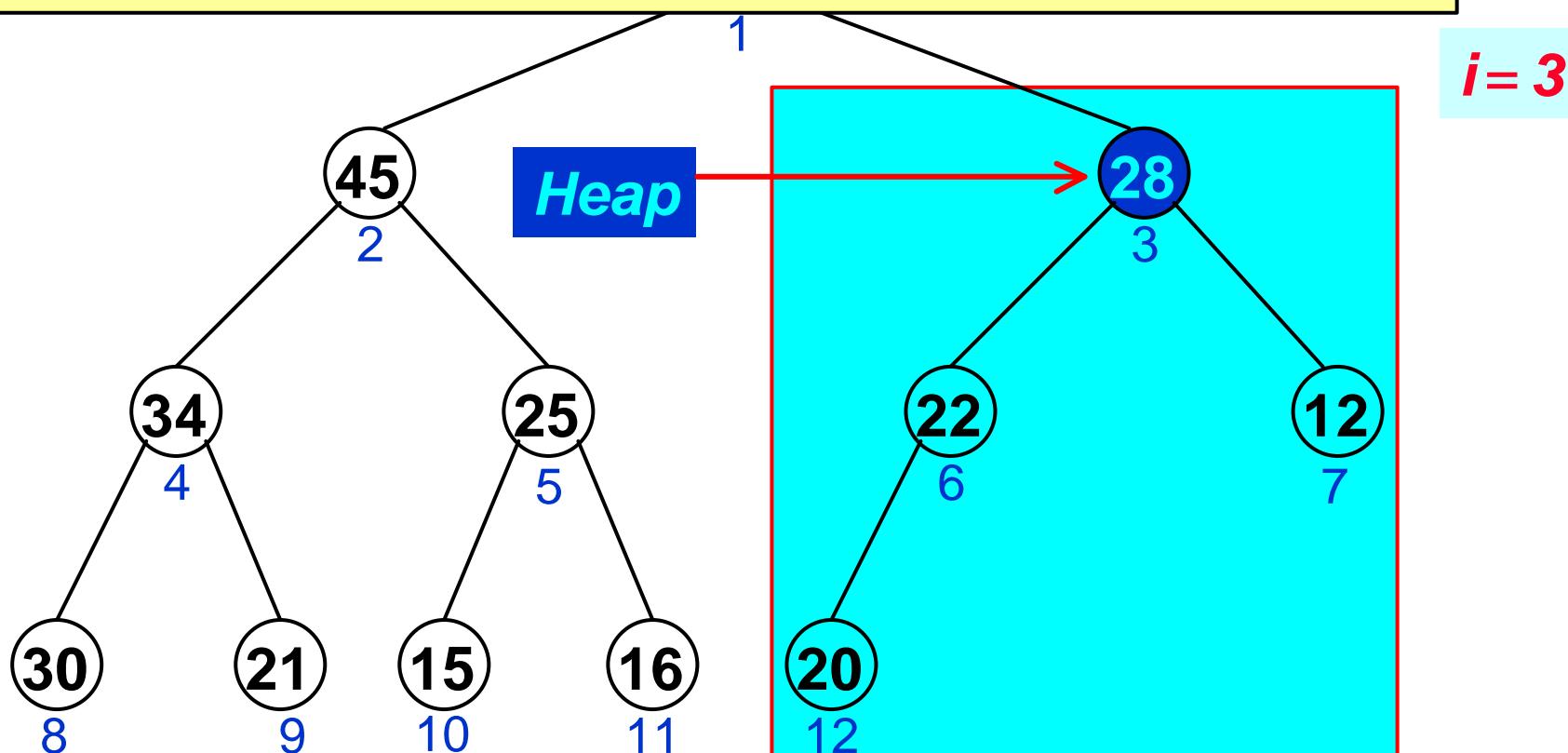
Costruisci Heap

Costruisci-Heap(*A*)

heapsize[*A*] = length[*A*]

FOR *i* = $\lceil \text{length}[A]/2 \rceil$ DOWNTO 1

DO Heapify(*A*, *i*)



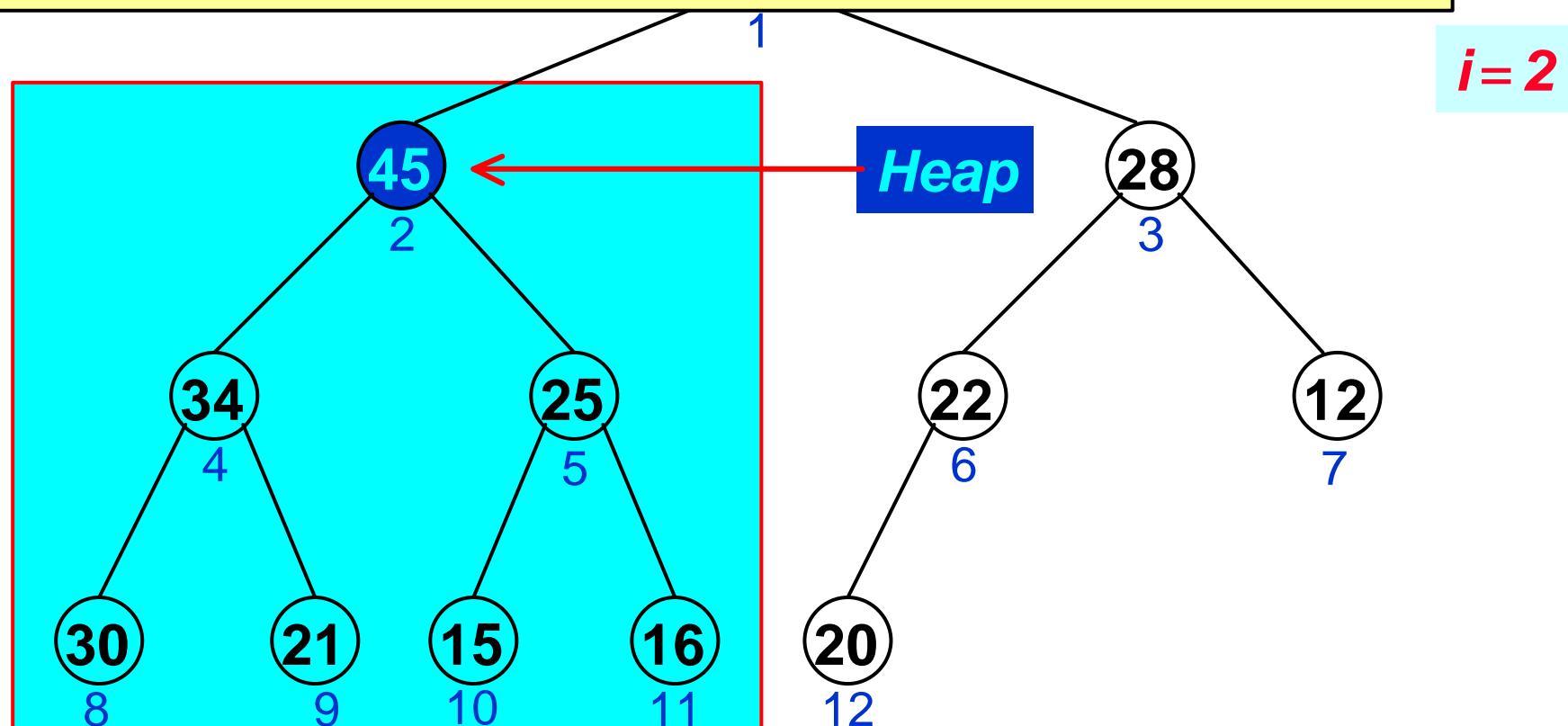
Costruisci Heap

Costruisci-Heap(*A*)

heapsize[*A*] = length[*A*]

FOR *i* = $\lceil \text{length}[A] / 2 \rceil$ DOWNTO 1

DO Heapify(*A*, *i*)



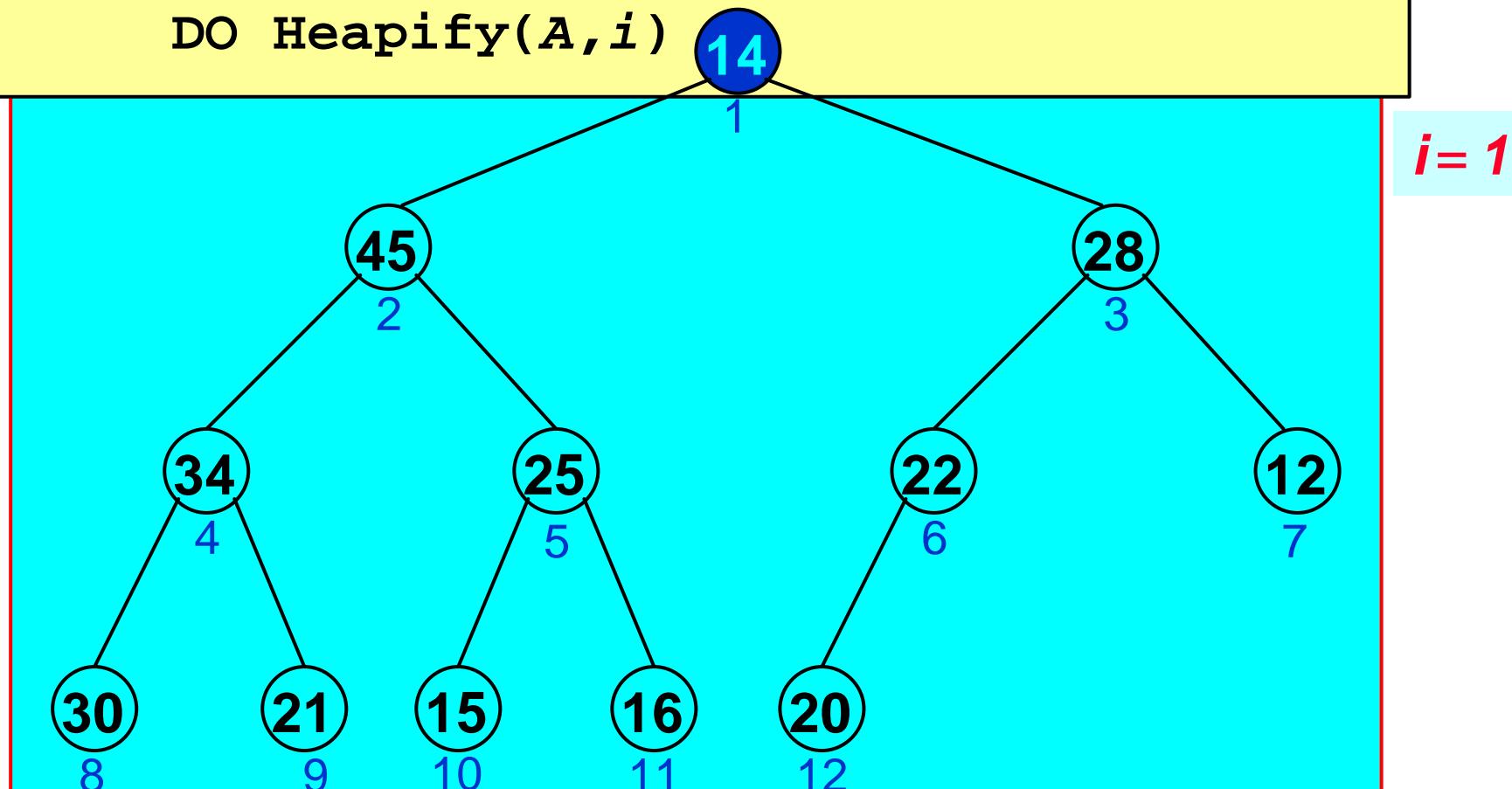
Costruisci Heap

Costruisci-Heap(*A*)

heapsize[*A*] = length[*A*]

FOR *i* = $\lceil \text{length}[A]/2 \rceil$ DOWNTO 1

DO Heapify(*A*, *i*)



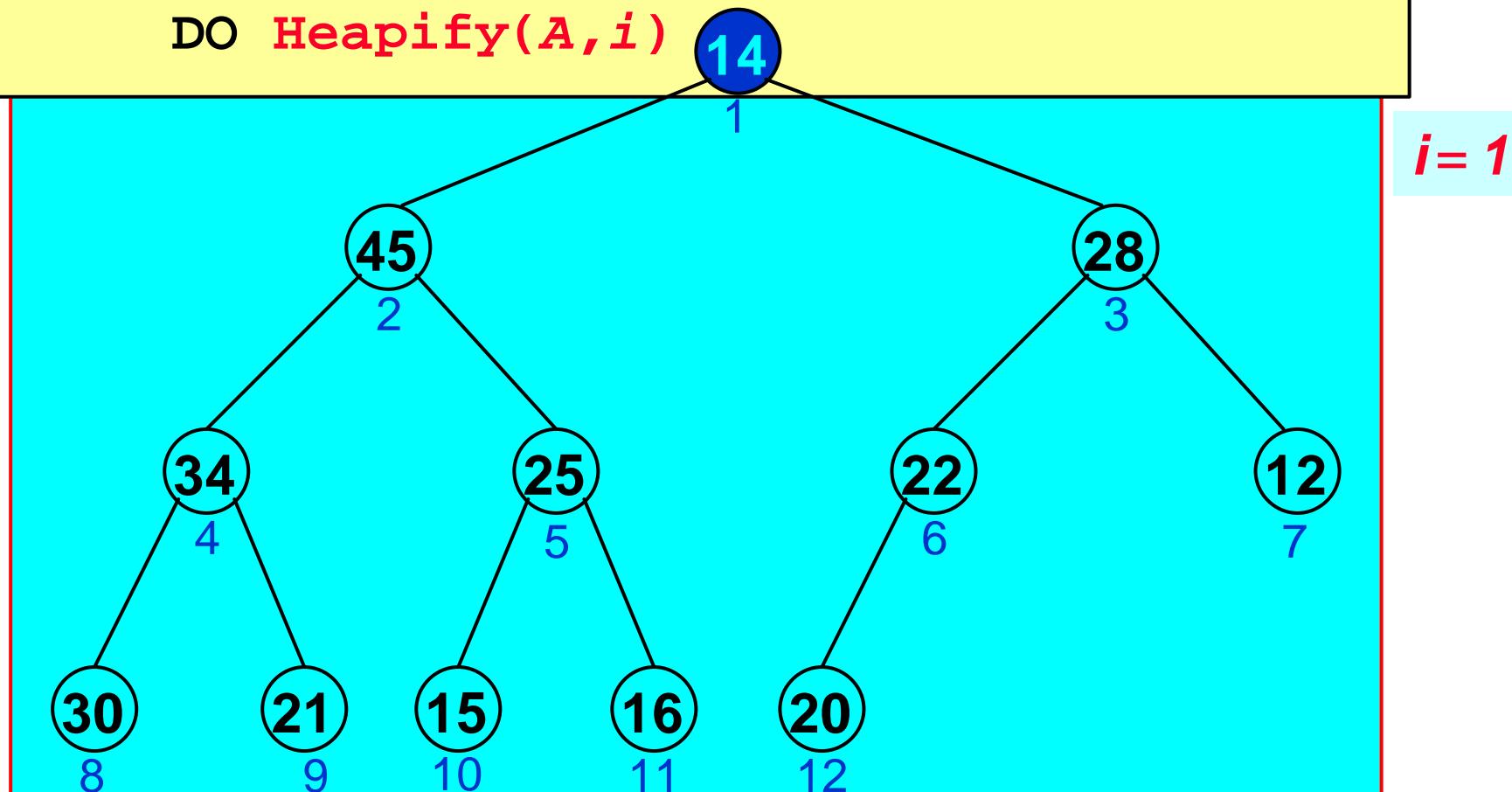
Costruisci Heap

Costruisci-Heap(*A*)

heapsize[*A*] = length[*A*]

FOR i = $\lceil \text{length}[A]/2 \rceil$ DOWNTO 1

DO Heapify(*A*, *i*)



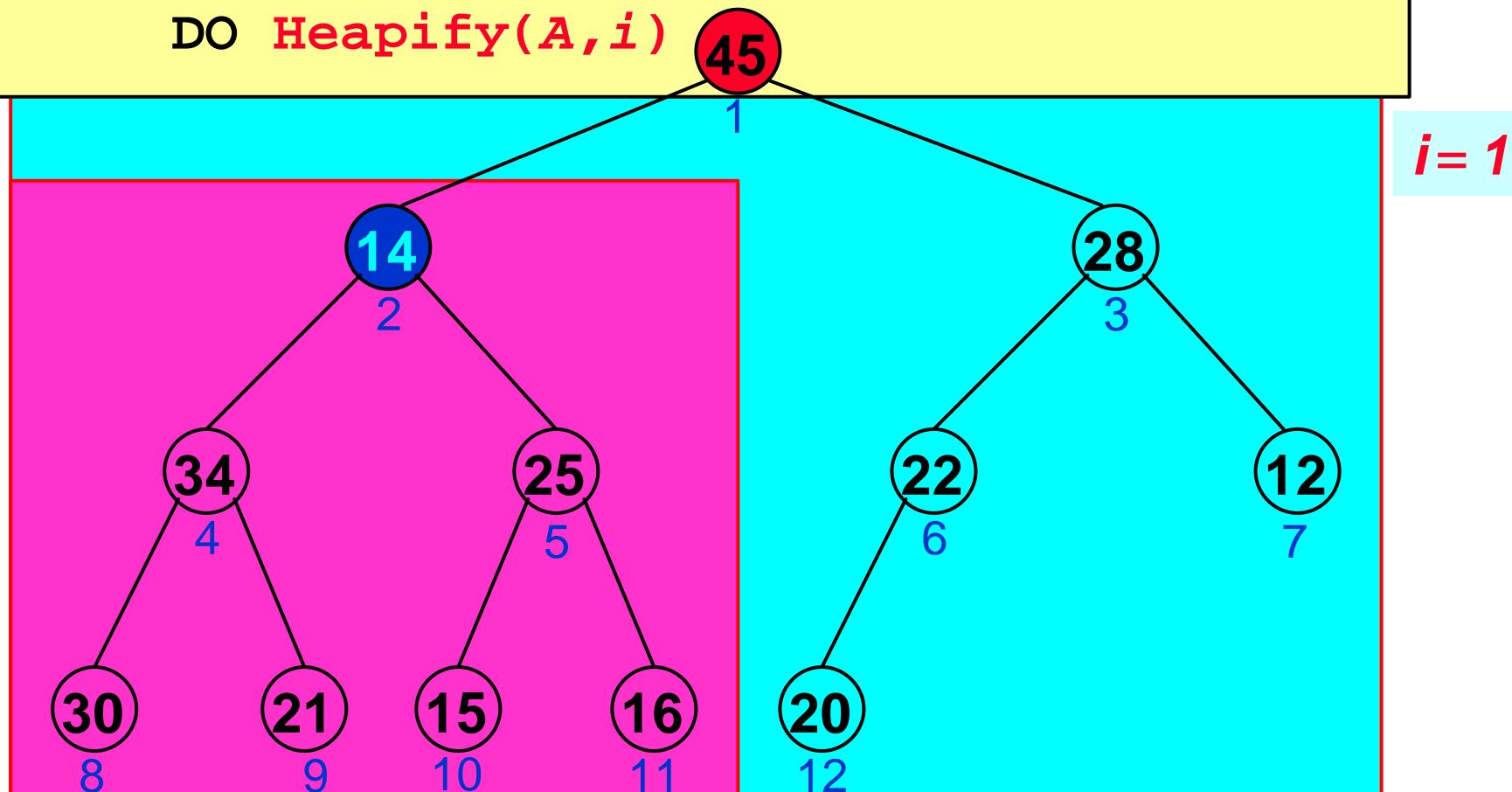
Costruisci Heap

Costruisci-Heap(A)

 heapsize[A] = length[A]

 FOR $i = \lceil \text{length}[A]/2 \rceil$ DOWNTO 1

 DO **Heapify**(A, i)



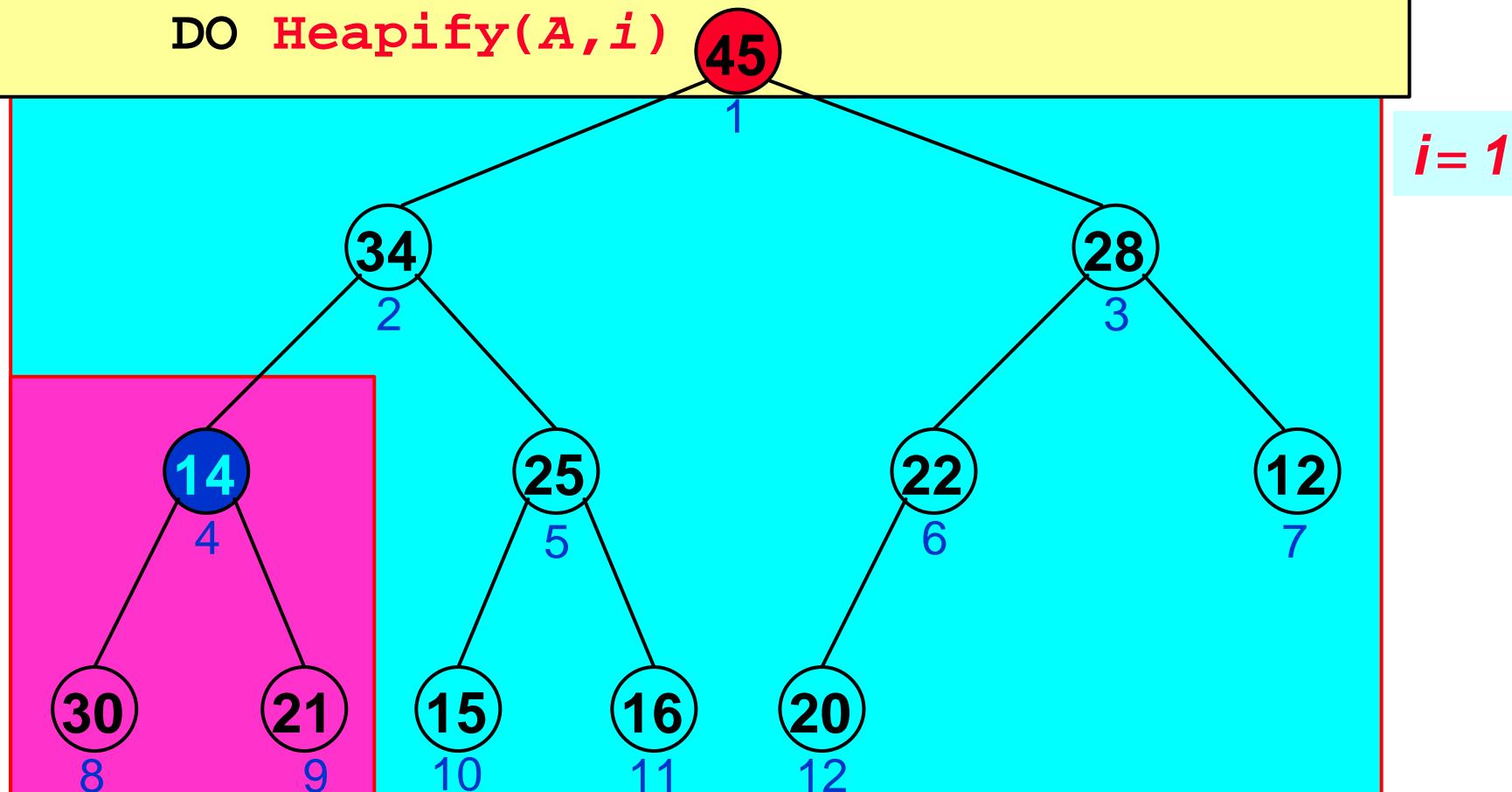
Costruisci Heap

Costruisci-Heap(*A*)

heapsize[*A*] = length[*A*]

FOR i = $\lceil \text{length}[A]/2 \rceil$ DOWNTO 1

DO Heapify(*A*, *i*)



i = 1

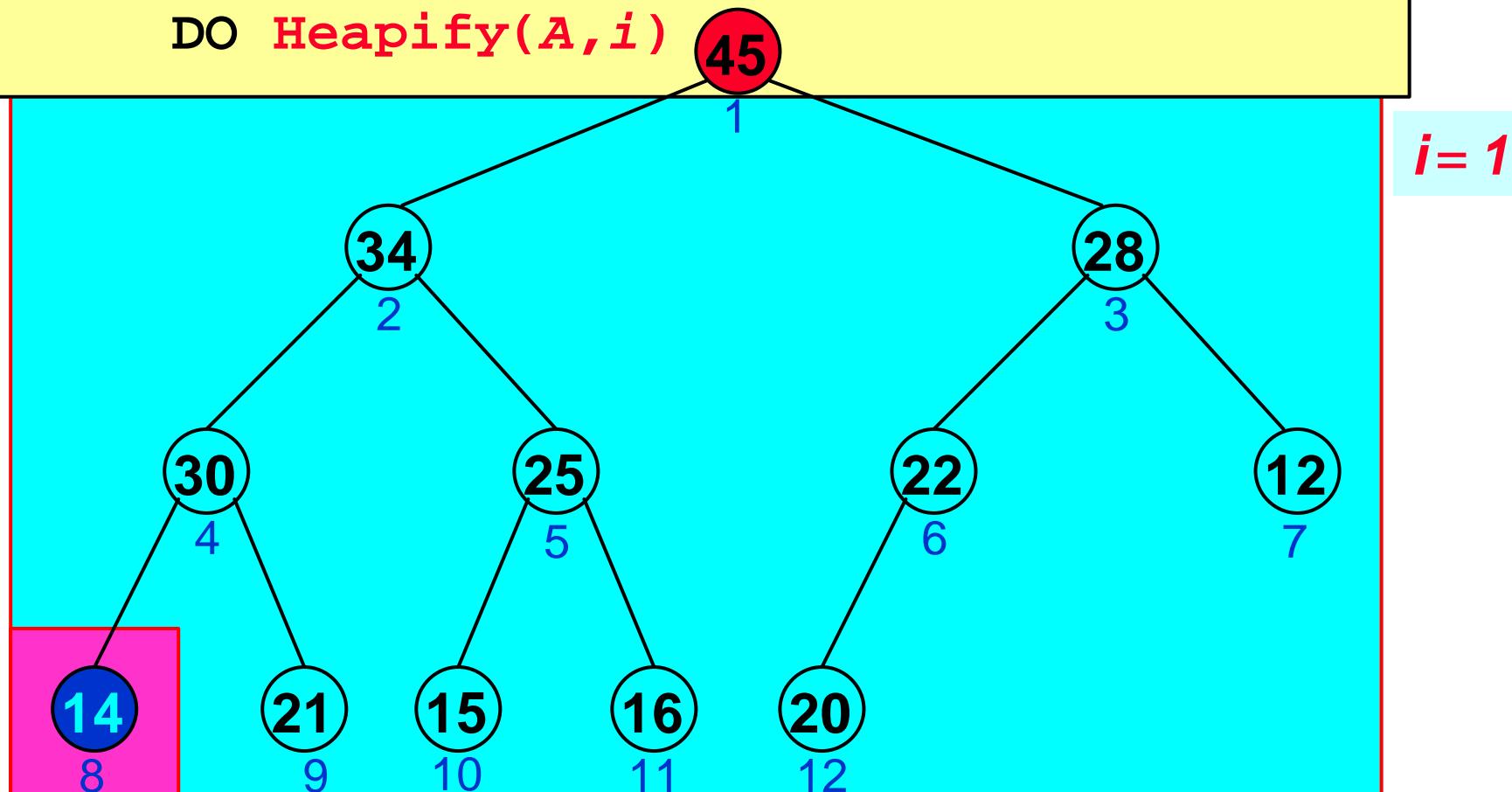
Costruisci Heap

Costruisci-Heap(*A*)

heapsize[*A*] = length[*A*]

FOR i = $\lceil \text{length}[A]/2 \rceil$ DOWNTO 1

DO Heapify(*A*, *i*)



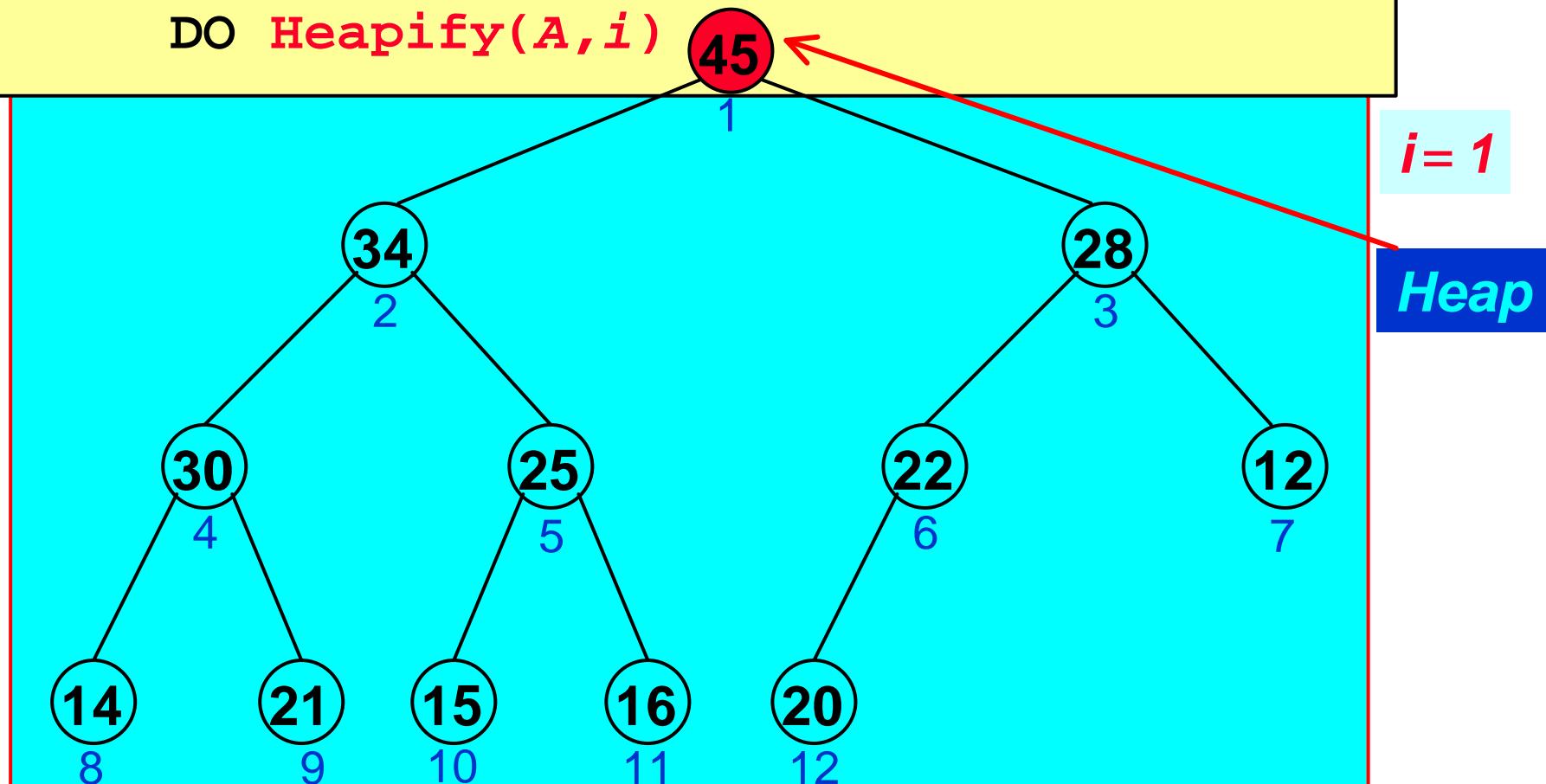
Costruisci Heap

Costruisci-Heap(*A*)

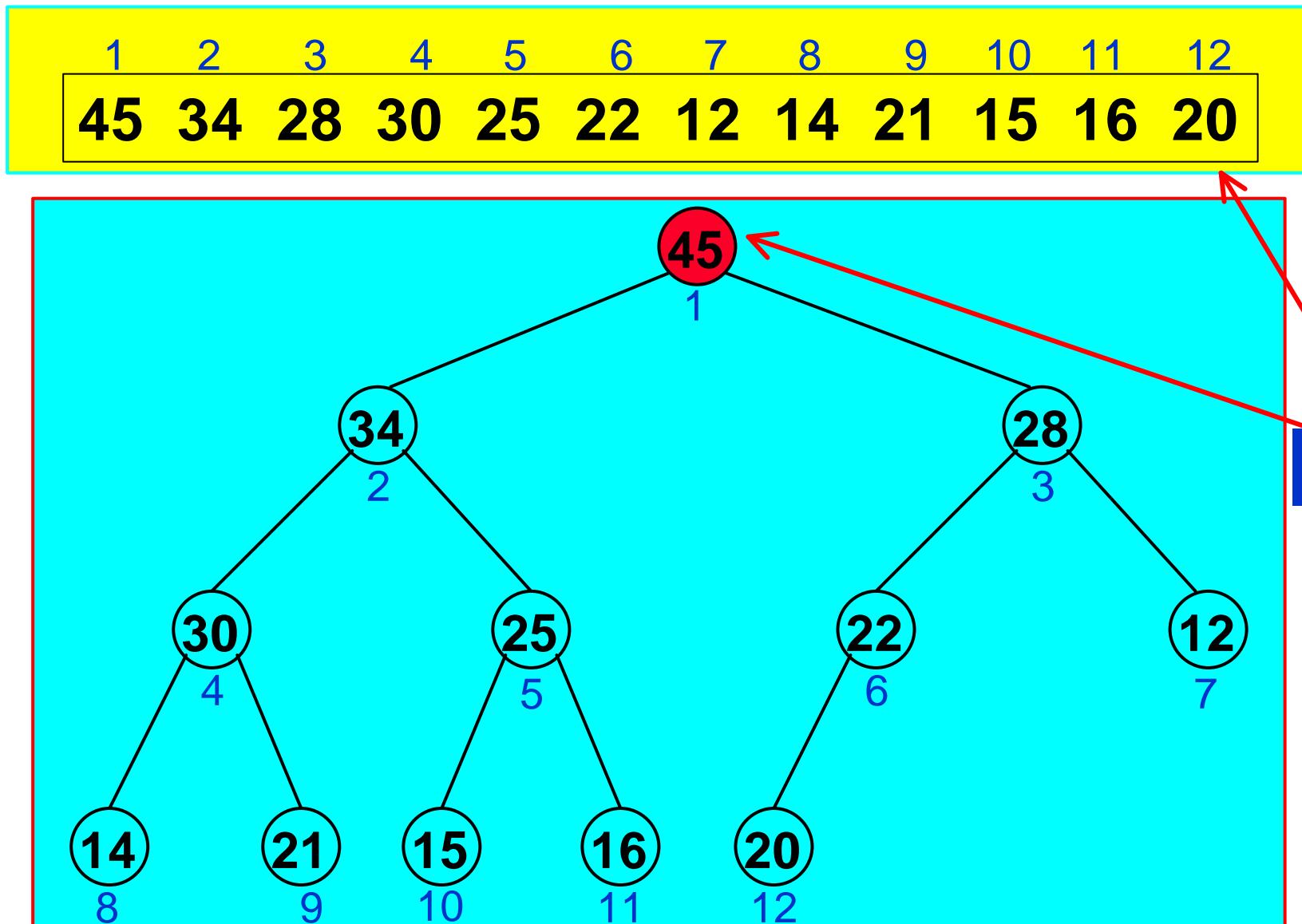
heapsize[A] = length[A]

 FOR *i* = $\lceil \text{length}[A]/2 \rceil$ DOWNTO 1

 DO **Heapify(*A, i*)**



Costruisci Heap



Complessità di Costruisci Heap

Costruisci-Heap(A)

```
heapsize[ $A$ ] = length[ $A$ ] } =  $O(1)$ 
FOR i =  $\lceil \text{length}[A]/2 \rceil$  DOWNTO 1 } =  $O(?)$ 
    DO Heapify( $A, i$ )
```

Complessità di Costruisci Heap

$$T(n) = \max(O(1), O(?)) = \max(O(1), O(f(n)))$$

*Poiché **Heapify** viene chiamata **$n/2$** volte si potrebbe ipotizzare*

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

e quindi

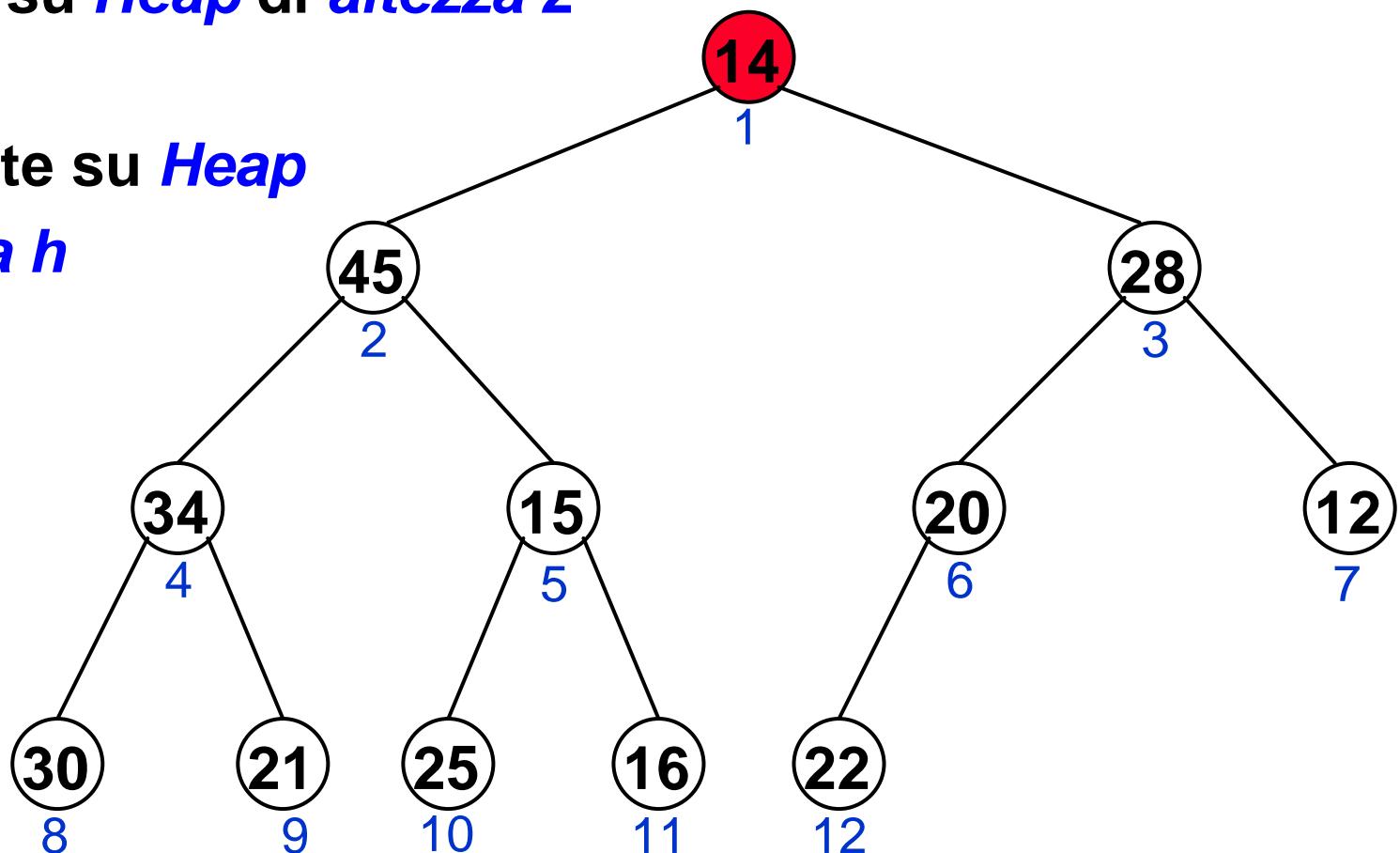
$$T(n) = \max(O(1), O(n \log n)) = O(n \log n)$$

ma....

Complessità di Costruisci Heap

Costruisci-Heap chiama *Heapify*

- $n/2$ volte su *Heap* di **altezza 0** (non eseguito)
- $n/4$ volte su *Heap* di **altezza 1**
- $n/8$ volte su *Heap* di **altezza 2**
- ...
- $n/2^{h+1}$ volte su *Heap* di **altezza h**



Complessità di Costruisci Heap

$$T(n) = \max(O(1), O(?)) = \max(O(1), O(f(n)))$$

$$f(n) = \sum_{h=0}^{\lfloor \log n \rfloor} \Theta(n / 2^{h+1}) \in O(h)$$

Costruisci-Heap chiama *Heapify*

- $n/2$ volte su *Heap* di **altezza 0** (in realtà non eseguito)
- $n/4$ volte su *Heap* di **altezza 1**
- $n/8$ volte su *Heap* di **altezza 2**
- ...
- $n/2^{h+1}$ volte su *Heap* di **altezza h**

Complessità di Costruisci Heap

$$T(n) = \max(O(1), O(?)) = \max(O(1), O(f(n)))$$

$$\begin{aligned} f(n) &= \sum_{h=0}^{\lfloor \log n \rfloor} h / 2^{h+1} \\ &= O\left(\frac{n}{2} \sum_{h=0}^{\lfloor \log n \rfloor} h / 2^h\right) \end{aligned}$$

Complessità di Costruisci Heap

$$T(n) = \max(O(1), O(?)) = \max(O(1), O(f(n)))$$

$$f(n) = \sum_{h=0}^{\lfloor \log n \rfloor} h / 2^{h+1} O(h)$$

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

$$= O\left(\frac{n}{2} \sum_{h=0}^{\infty} h / 2^h\right)$$

Complessità di Costruisci Heap

$$T(n) = \max(O(1), O(?)) = \max(O(1), O(f(n)))$$

$$f(n) = \sum_{h=0}^{\lfloor \log n \rfloor} h / 2^{h+1} O(h)$$

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

$$= O\left(\frac{n}{2} \sum_{h=0}^{\infty} h / 2^h\right)$$
$$= O(2n/2)$$

$$\sum_{h=0}^{\infty} h x^h = \frac{x}{(1-x)^2}$$

$$\frac{x}{(1-x)^2} = 2$$

$$x = 1/2 \Leftarrow 1$$

Complessità di Costruisci Heap

$$T(n) = \max(O(1), O(?)) = \max(O(1), O(f(n)))$$

$$\begin{aligned} f(n) &= \sum_{h=0}^{\lfloor \log n \rfloor} h / 2^{h+1} \\ &= O\left(\frac{n}{2} \sum_{h=0}^{\lfloor \log n \rfloor} h / 2^h\right) \\ &= O\left(\frac{n}{2} \sum_{h=0}^{\infty} h / 2^h\right) \\ &= O(n) \end{aligned}$$

Complessità di Costruisci Heap

$$T(n) = \max(O(1), O(?)) = \max(O(1), O(f(n)))$$

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

$$T(n) = \max(O(1), O(n)) = O(n)$$

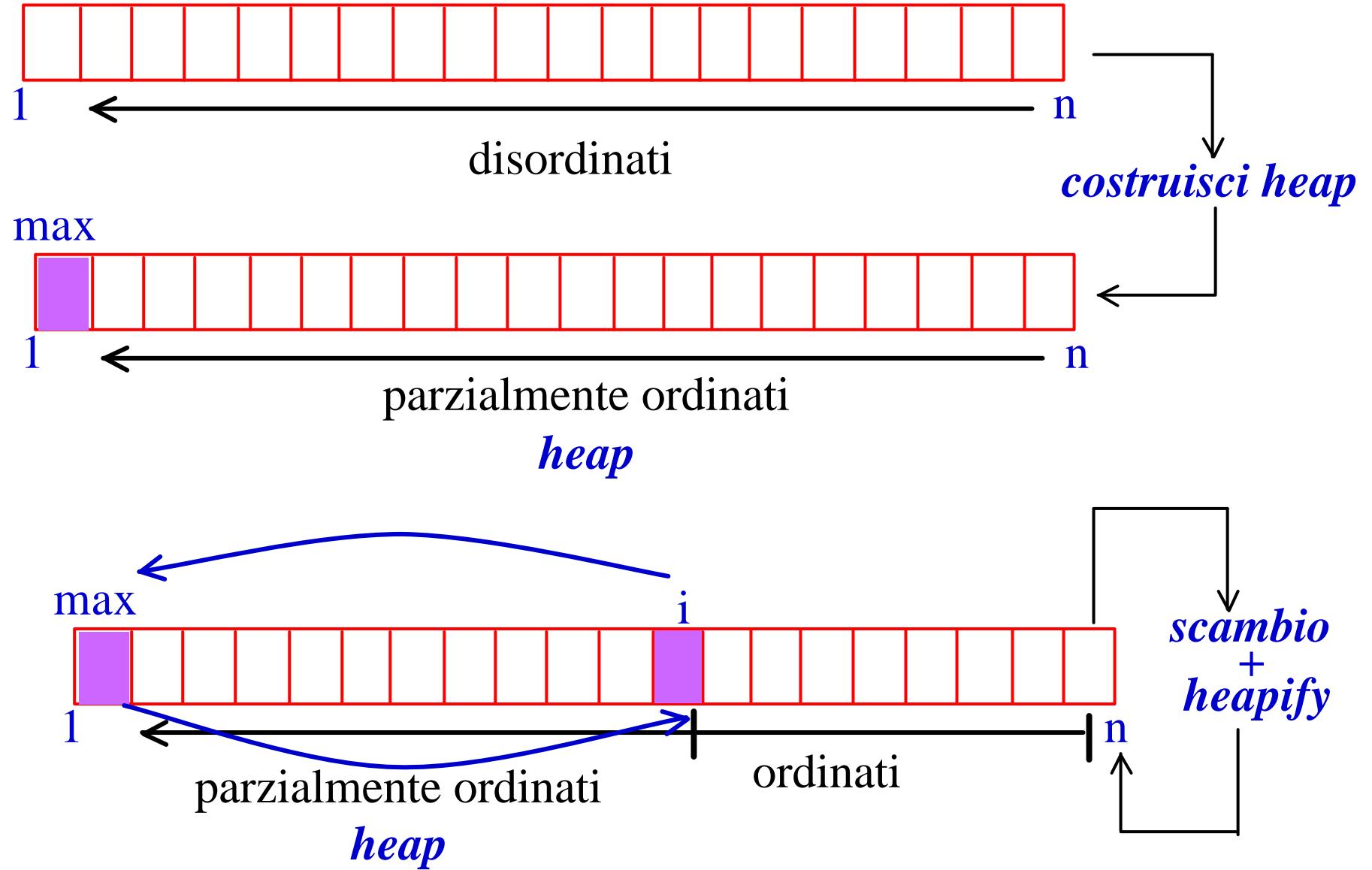
*Costruire uno Heap di
n elementi è poco costoso,
al più costa $O(n)$!*

Heap Sort: intuizioni

Heap-Sort: è una variazione di **Select-sort** in cui la ricerca dell'elemento massimo è facilitata dal mantenimento della sequenza in uno heap:

- si costruisce uno **Heap** a partire dall'array non ordinato in input.
- viene sfruttata la proprietà degli **Heap** per cui la radice **A[1]** dello **Heap** è sempre il massimo:
 - scandisce tutti gli elementi dell'array a partire dall'ultimo e ad ogni iterazione
 - la radice **A[1]** viene scambiata con l'elemento nell'ultima posizione corrente dello **Heap**
 - viene ridotta la dimensione dello **Heap** e
 - ripristinato lo **Heap** con **Heapify**

Heap Sort: intuizioni



Heap Sort

```
Select-Sort(A)
```

```
    FOR i = length[A] DOWNT0 2
        DO max = Findmax(A,i)
            "scambia A[max] e A[i]"
```

```
Heap-Sort(A)
```

```
    Costruisci-Heap(A)
```

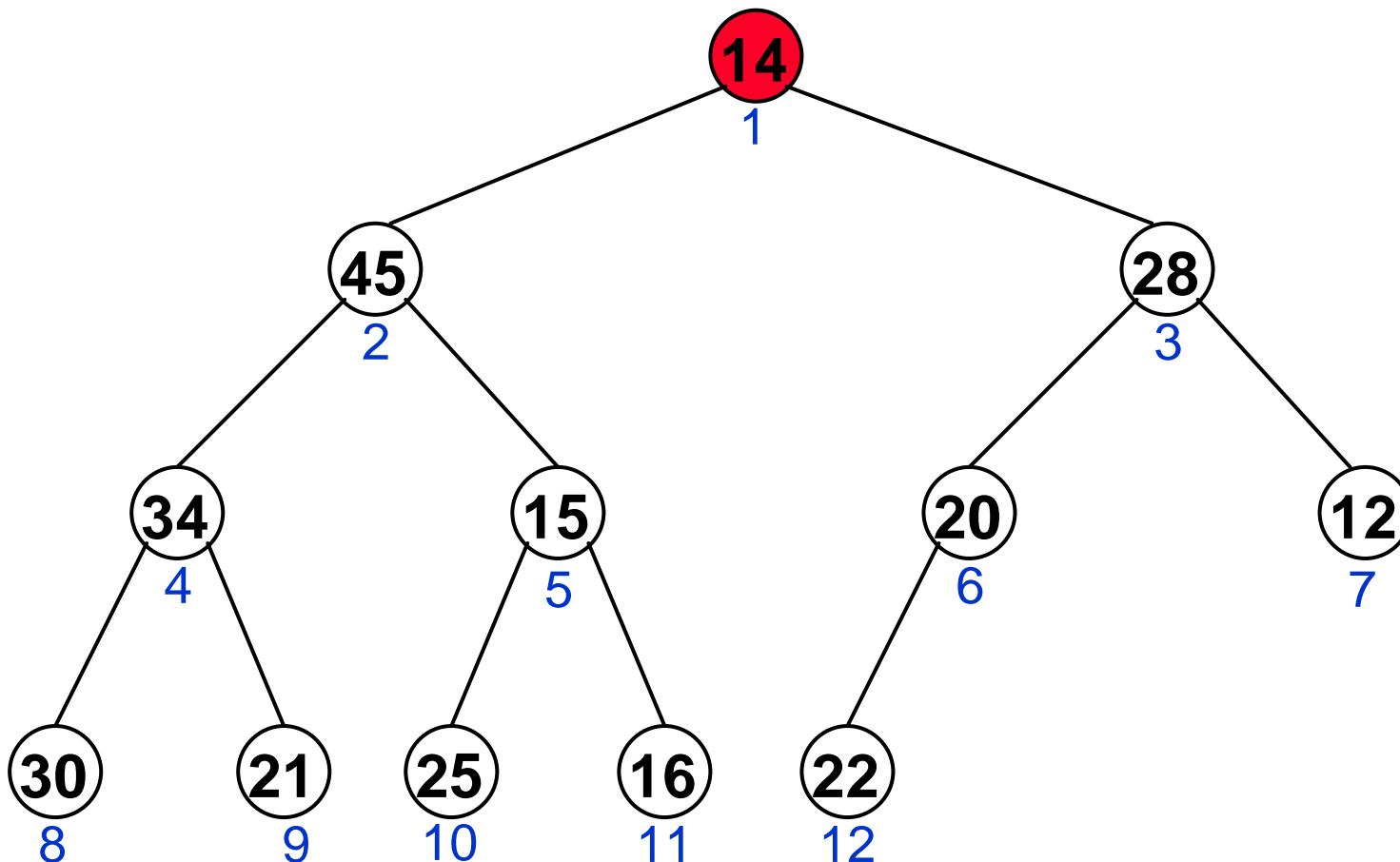
```
    FOR i = length[A] DOWNT0 2
        DO /* elemento massimo in A[1] */
            "scambia A[1] e A[i]"
            /* ripristina lo heap */
            heapsize[A] = heapsize[A]-1
            Heapify(A,1)
```

Heap Sort

Heap-Sort(A)

Costruisce-Heap(A)

...

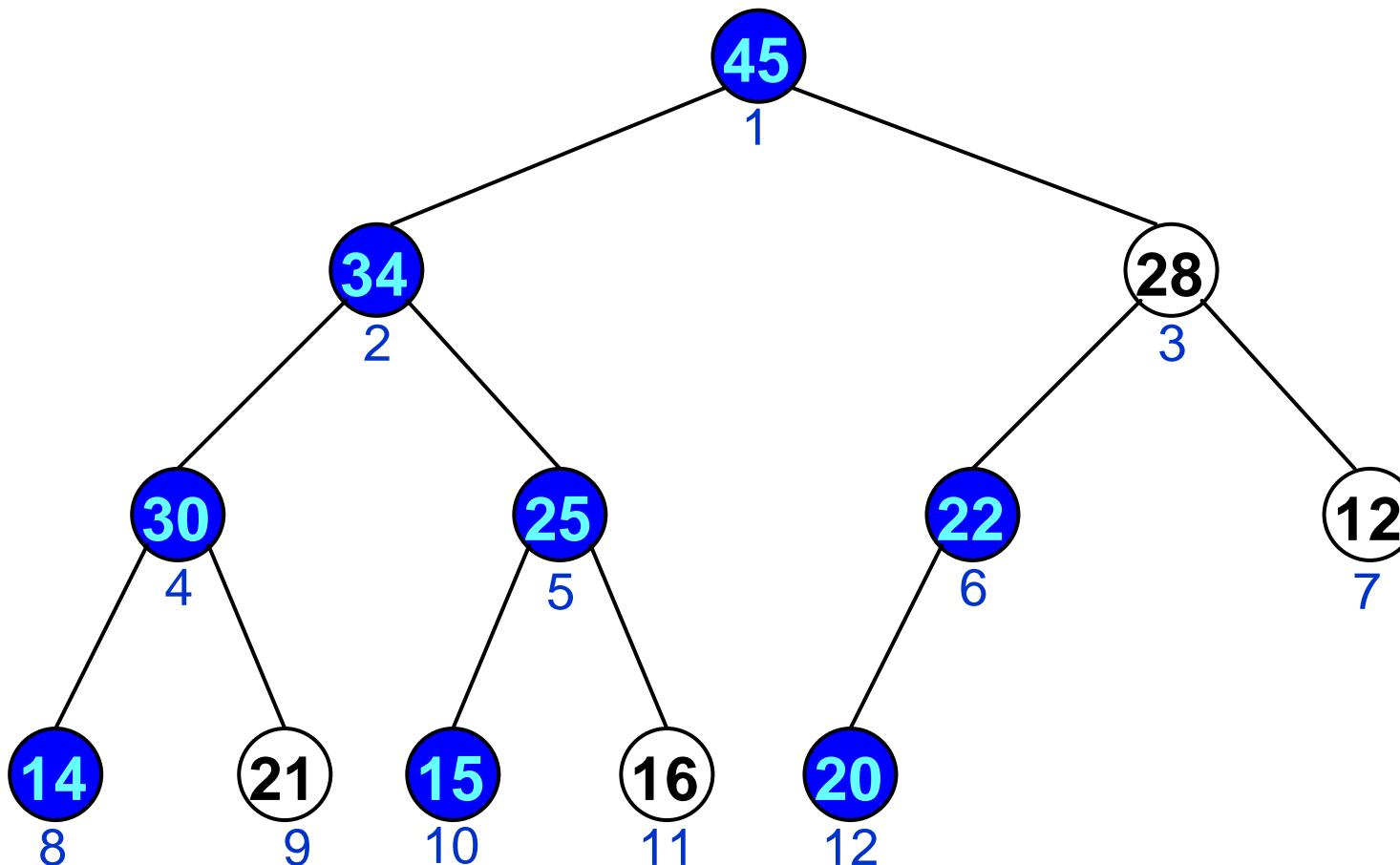


Heap Sort

Heap-Sort(A)

Costruisce-Heap(A)

...



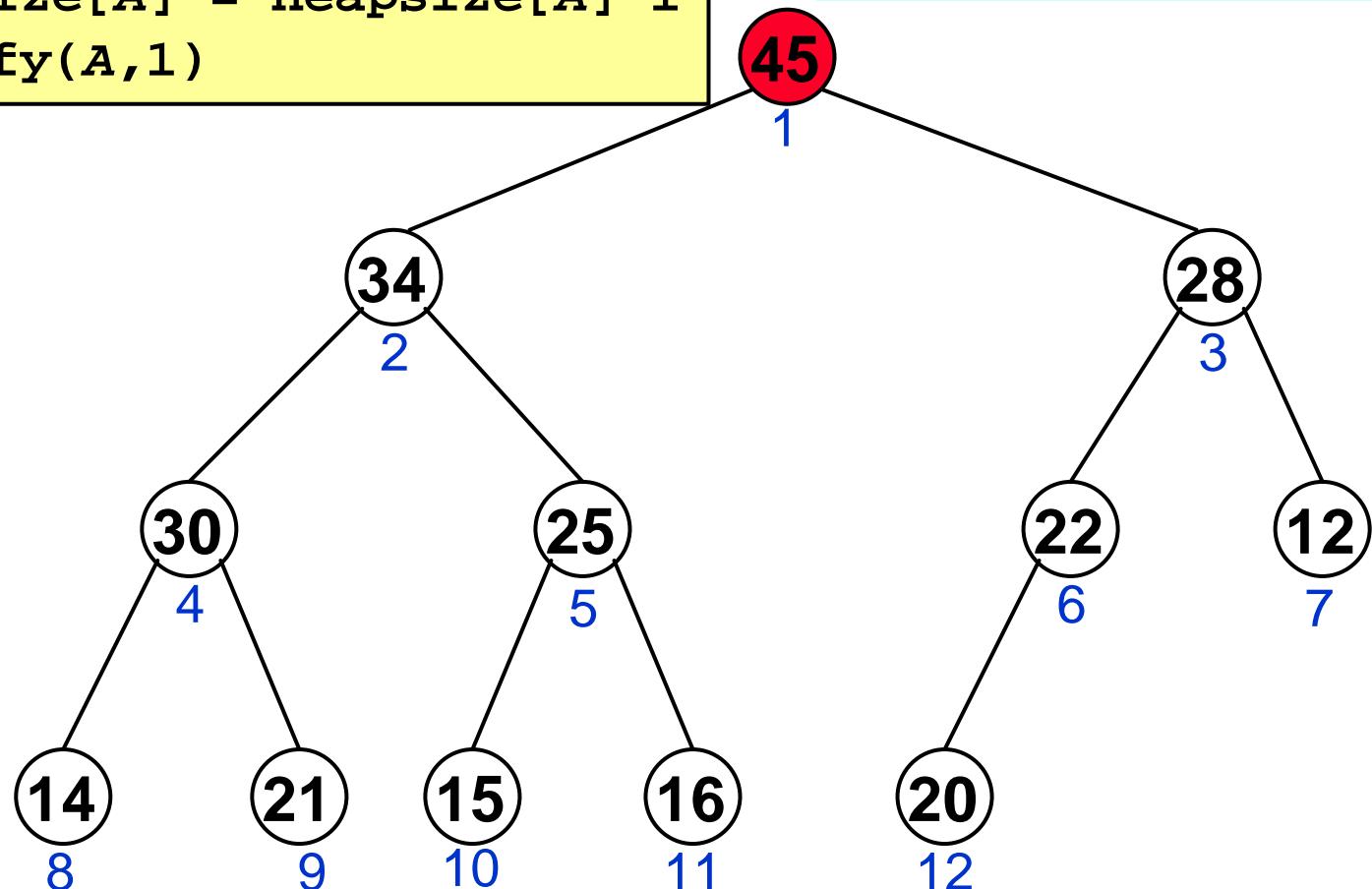
Heap Sort

```
Heap-Sort(A)
```

```
...
FOR i = length[A] DOWNTON 2
    DO "scambia A[1] e A[i]"
        heapsize[A] = heapsize[A]-1
    Heapify(A,1)
```

$i = 12$

$\text{heapsize}[A] = 12$



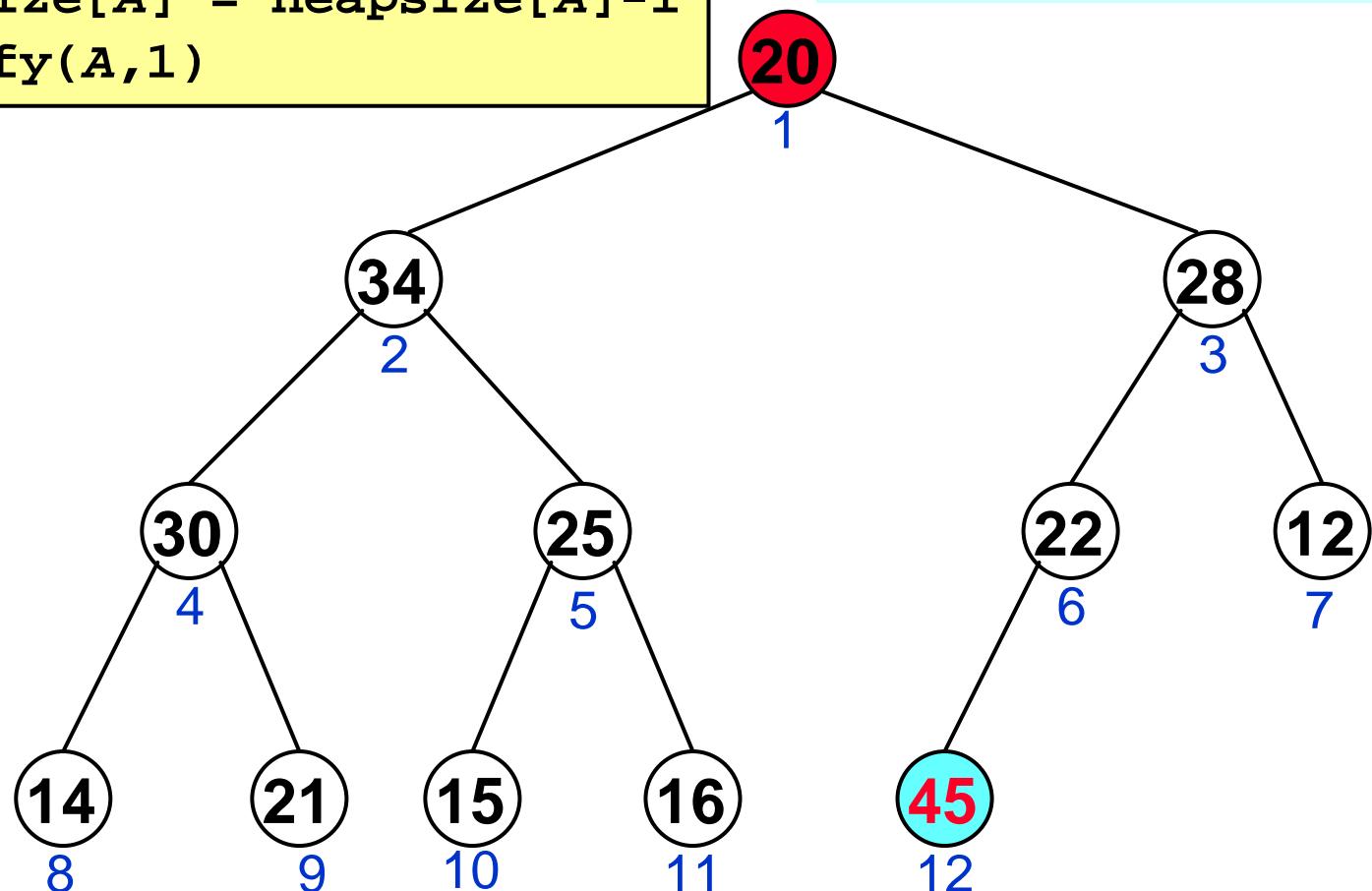
Heap Sort

```
Heap-Sort(A)
```

```
...
FOR i = length[A] DOWNTON 2
    DO "scambia A[1] e A[i]"
        heapsize[A] = heapsize[A]-1
    Heapify(A,1)
```

$i = 12$

$\text{heapsize}[A] = 12$



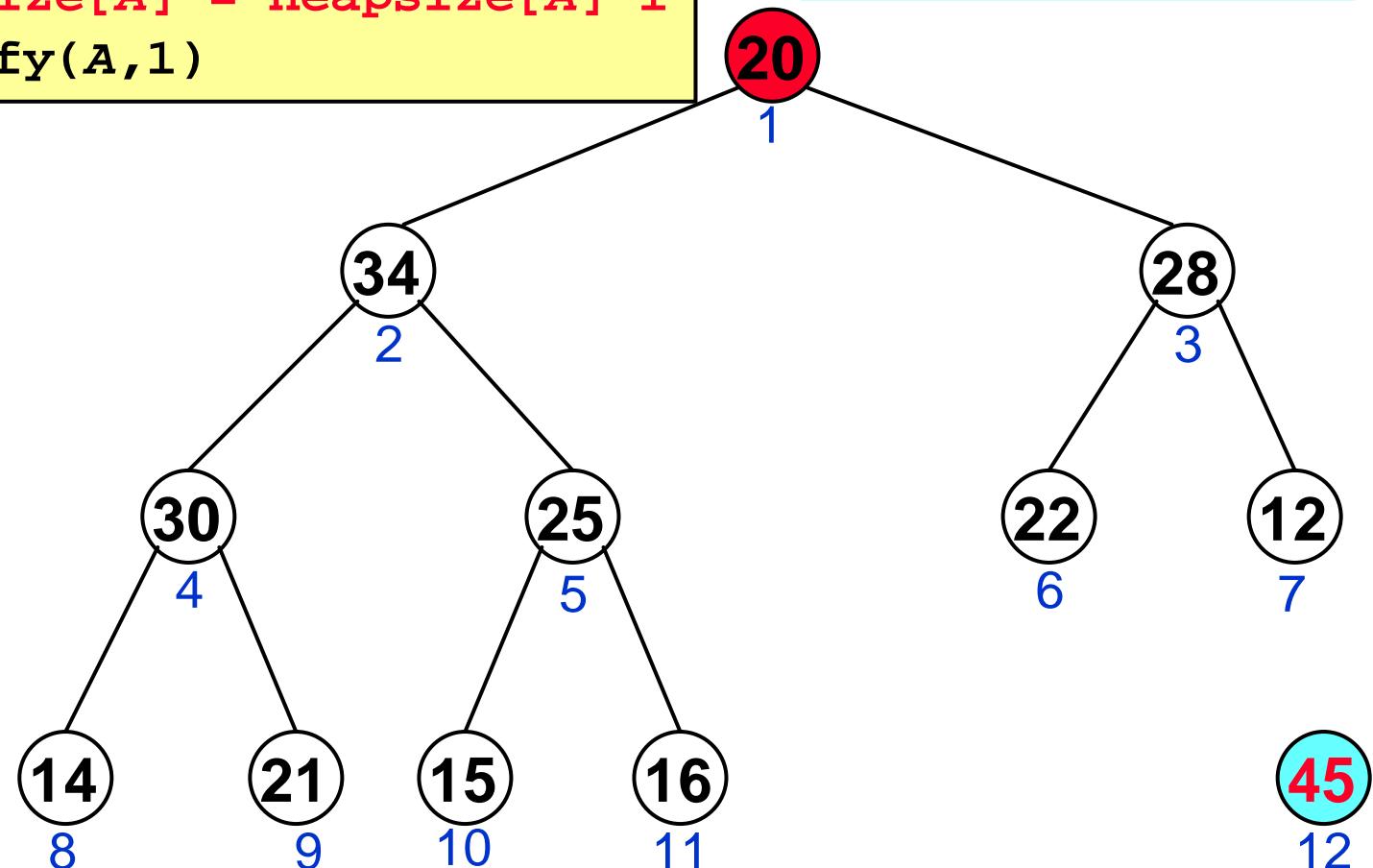
Heap Sort

```
Heap-Sort(A)
```

```
...
FOR i = length[A] DOWNTON 2
    DO "scambia A[1] e A[i]"
        heapsize[A] = heapsize[A]-1
    Heapify(A,1)
```

$i=12$

$\text{heapsize}[A]=11$



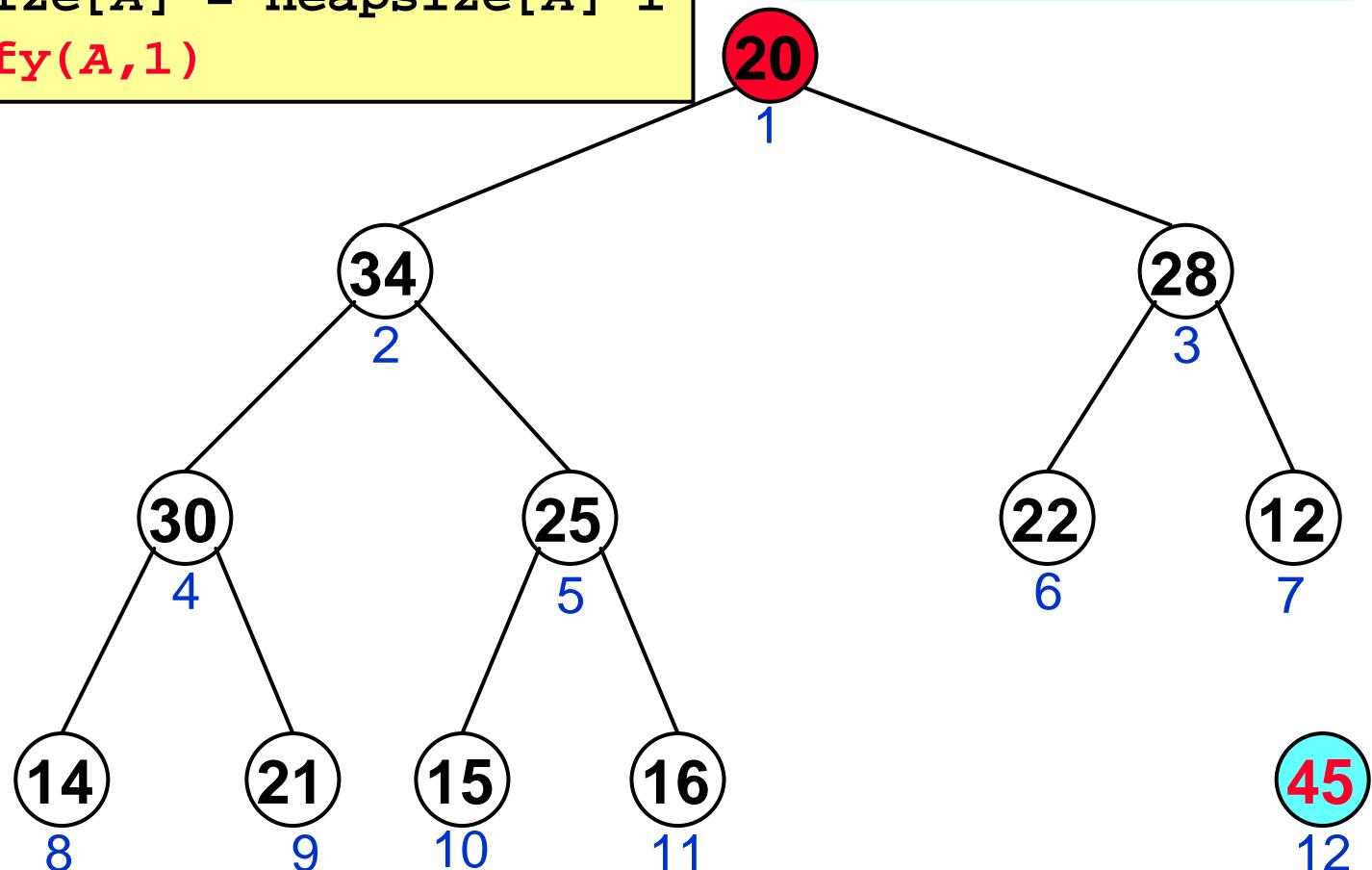
Heap Sort

```
Heap-Sort(A)
```

```
...
FOR i = length[A] DOWNTON 2
    DO "scambia A[1] e A[i]"
        heapsize[A] = heapsize[A]-1
    Heapify(A,1)
```

$i=12$

$\text{heapsize}[A]=11$



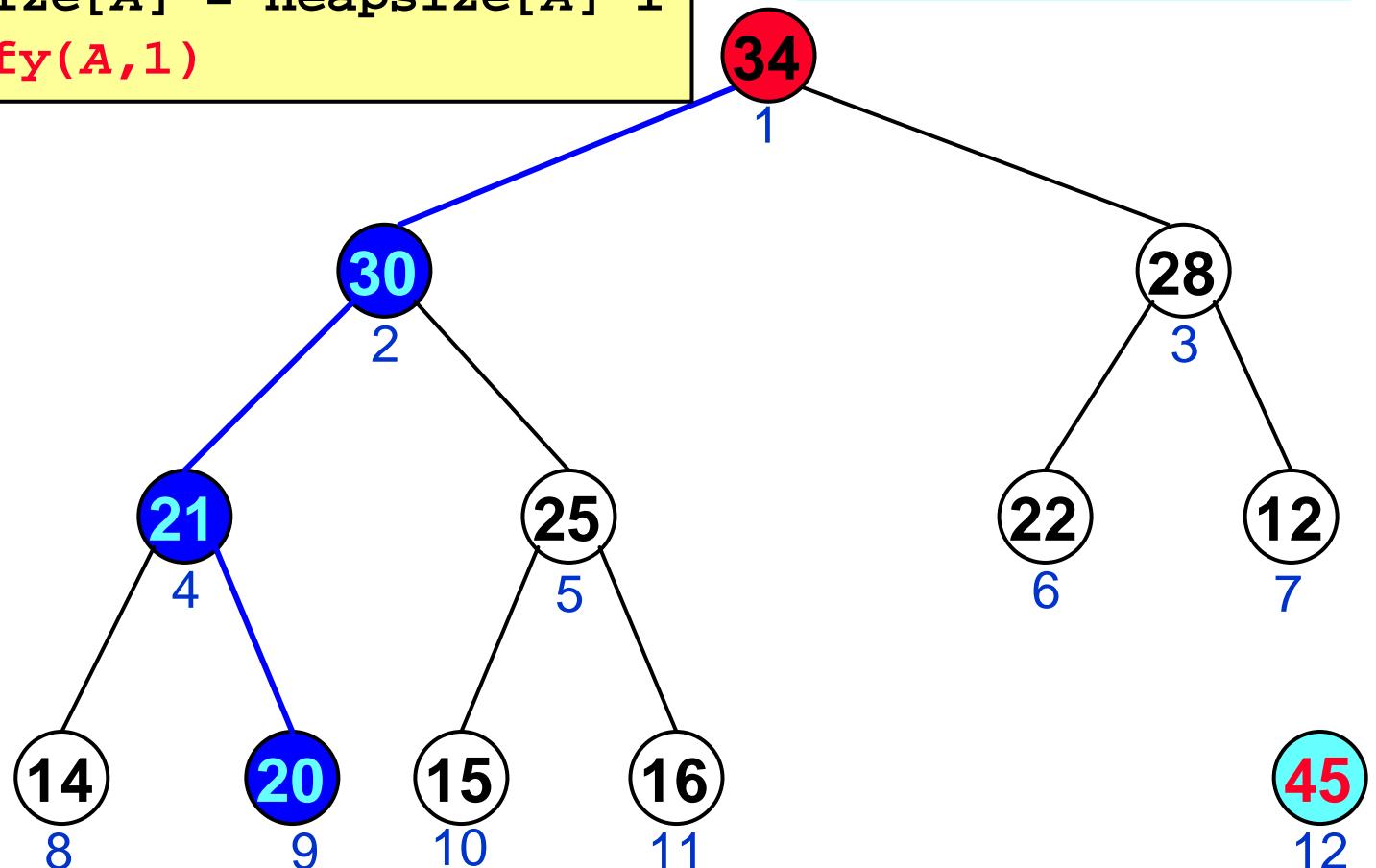
Heap Sort

```
Heap-Sort(A)
```

```
...
FOR i = length[A] DOWNTON 2
    DO "scambia A[1] e A[i]"
        heapsize[A] = heapsize[A]-1
    Heapify(A,1)
```

$i=12$

$\text{heapsize}[A]=11$



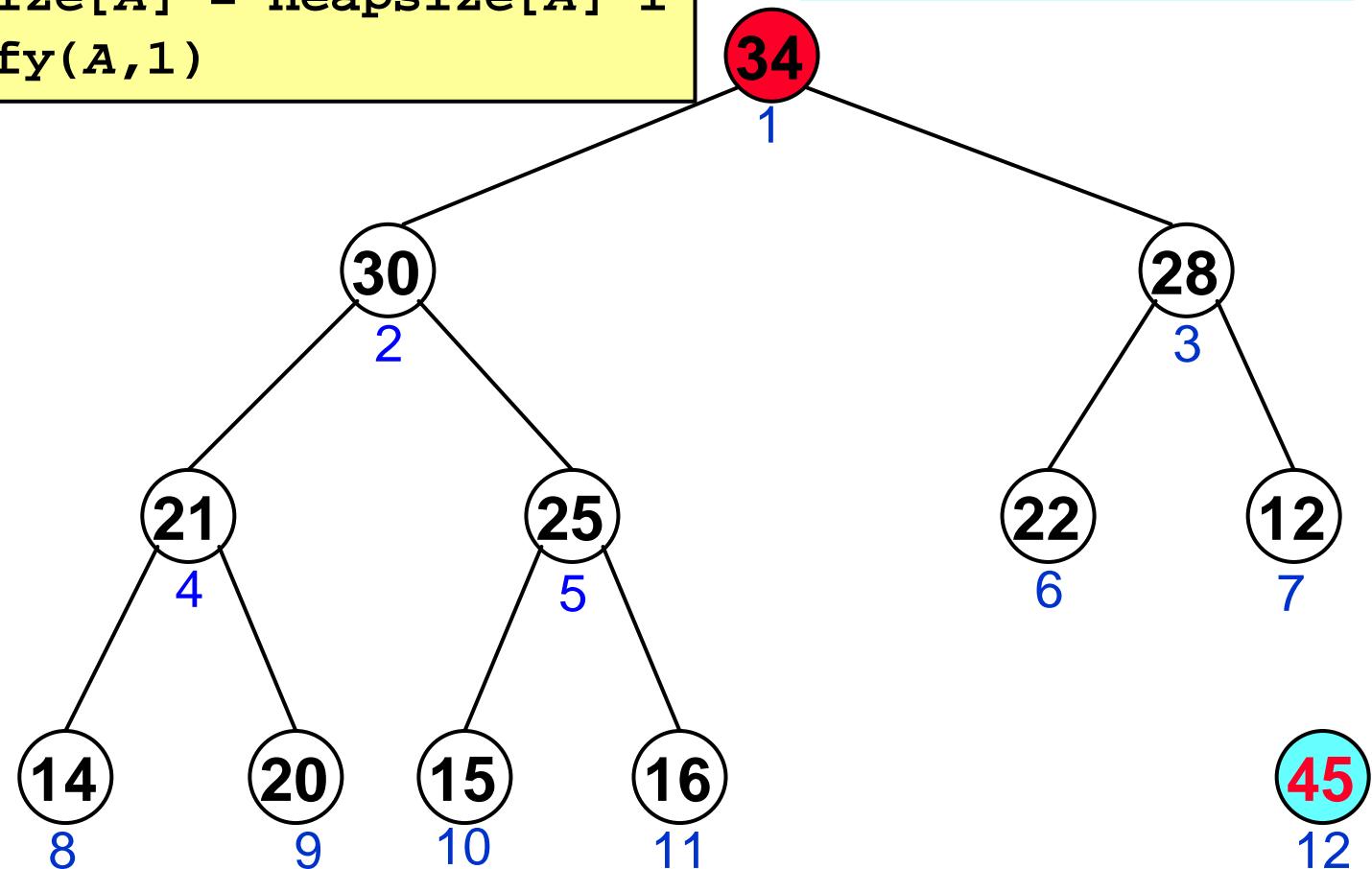
Heap Sort

```
Heap-Sort(A)
```

```
...
FOR i = length[A] DOWNTON 2
    DO "scambia A[1] e A[i]"
        heapsize[A] = heapsize[A]-1
    Heapify(A,1)
```

$i=11$

$\text{heapsize}[A]=11$



Heap Sort

```
Heap-Sort(A)
```

```
...
```

```
FOR i = length[A] DOWNTON 2
```

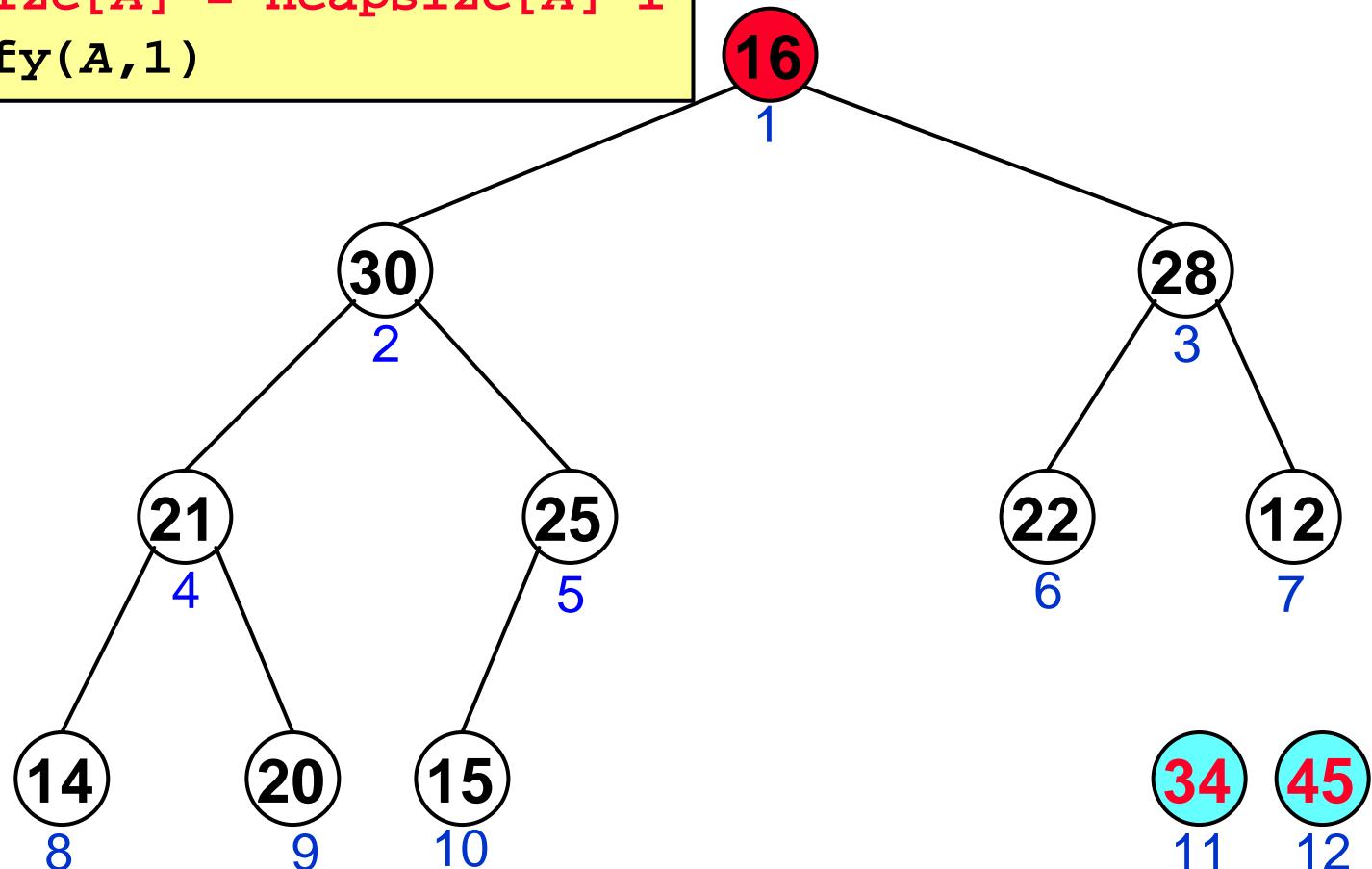
```
DO "scambia A[1] e A[i]"
```

```
heapsize[A] = heapsize[A]-1
```

```
Heapify(A,1)
```

i=11

heapsize[A]=10



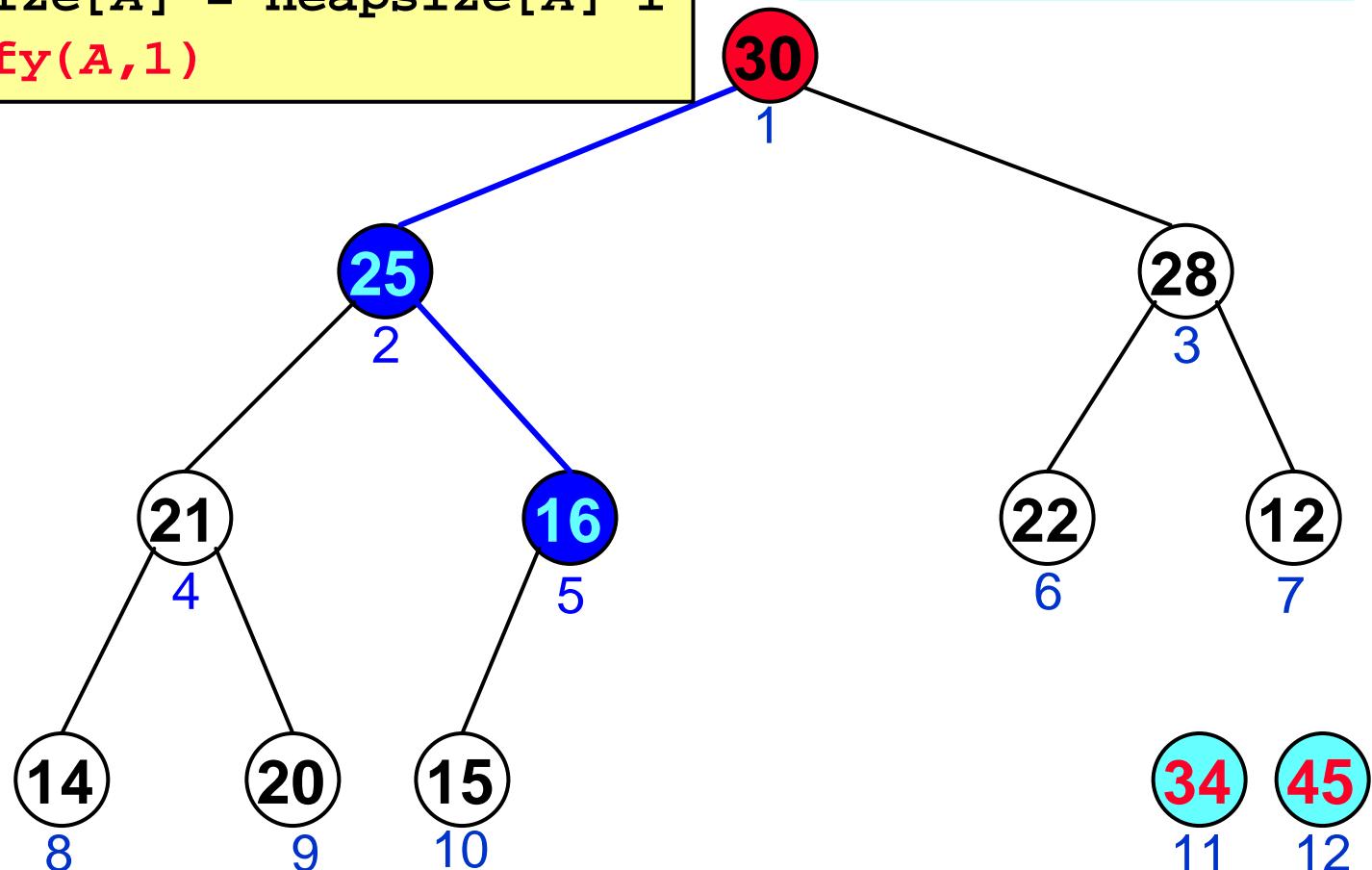
Heap Sort

```
Heap-Sort(A)
```

```
...
FOR i = length[A] DOWNTON 2
    DO "scambia A[1] e A[i]"
    heapsize[A] = heapsize[A]-1
    Heapify(A,1)
```

i=11

heapsize[A]=10



Heap Sort

```
Heap-Sort(A)
```

```
...
```

```
FOR i = length[A] DOWNTON 2
```

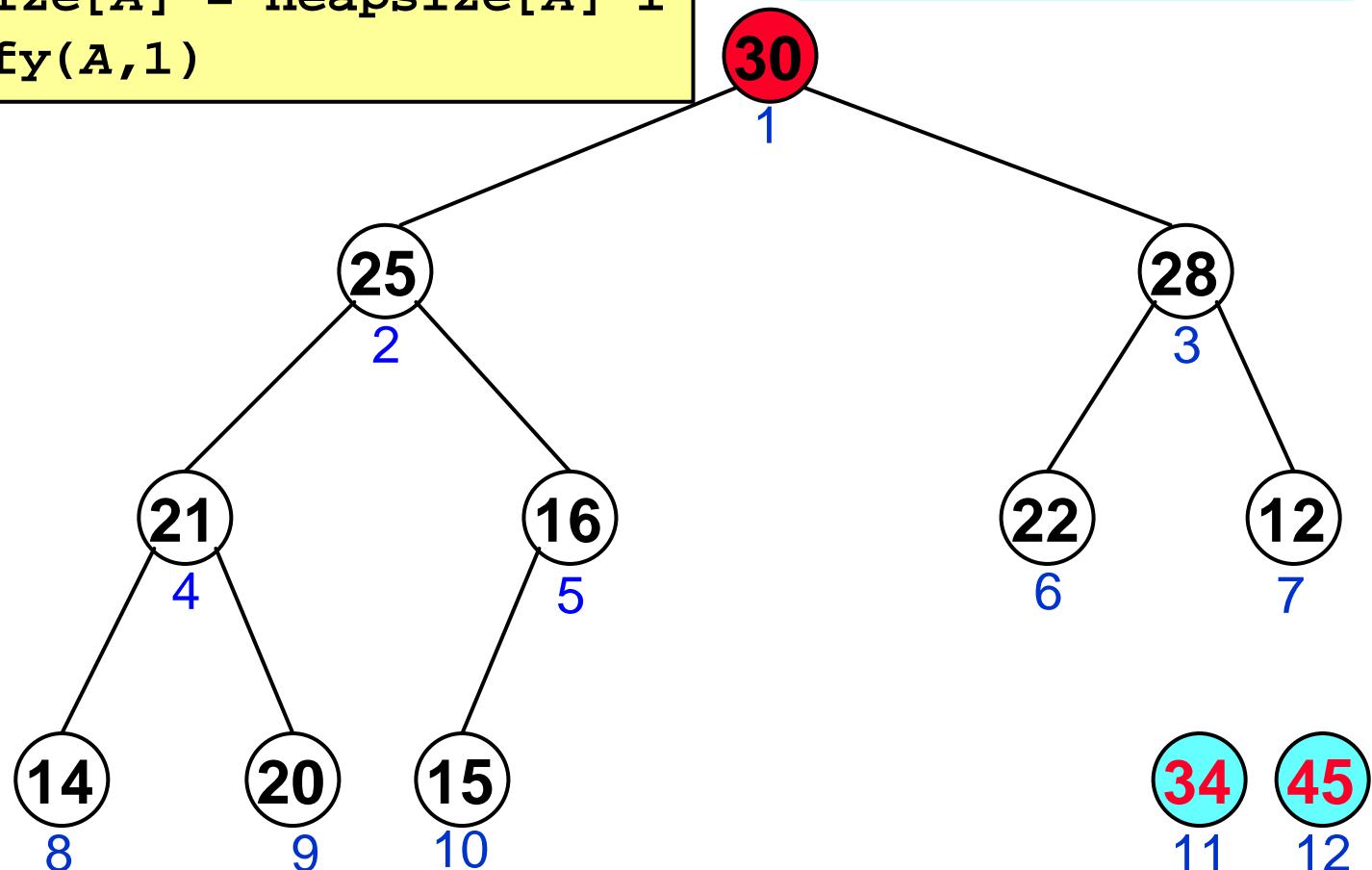
```
DO "scambia A[1] e A[i]"
```

```
heapsize[A] = heapsize[A]-1
```

```
Heapify(A,1)
```

i=10

heapsize[A]=10



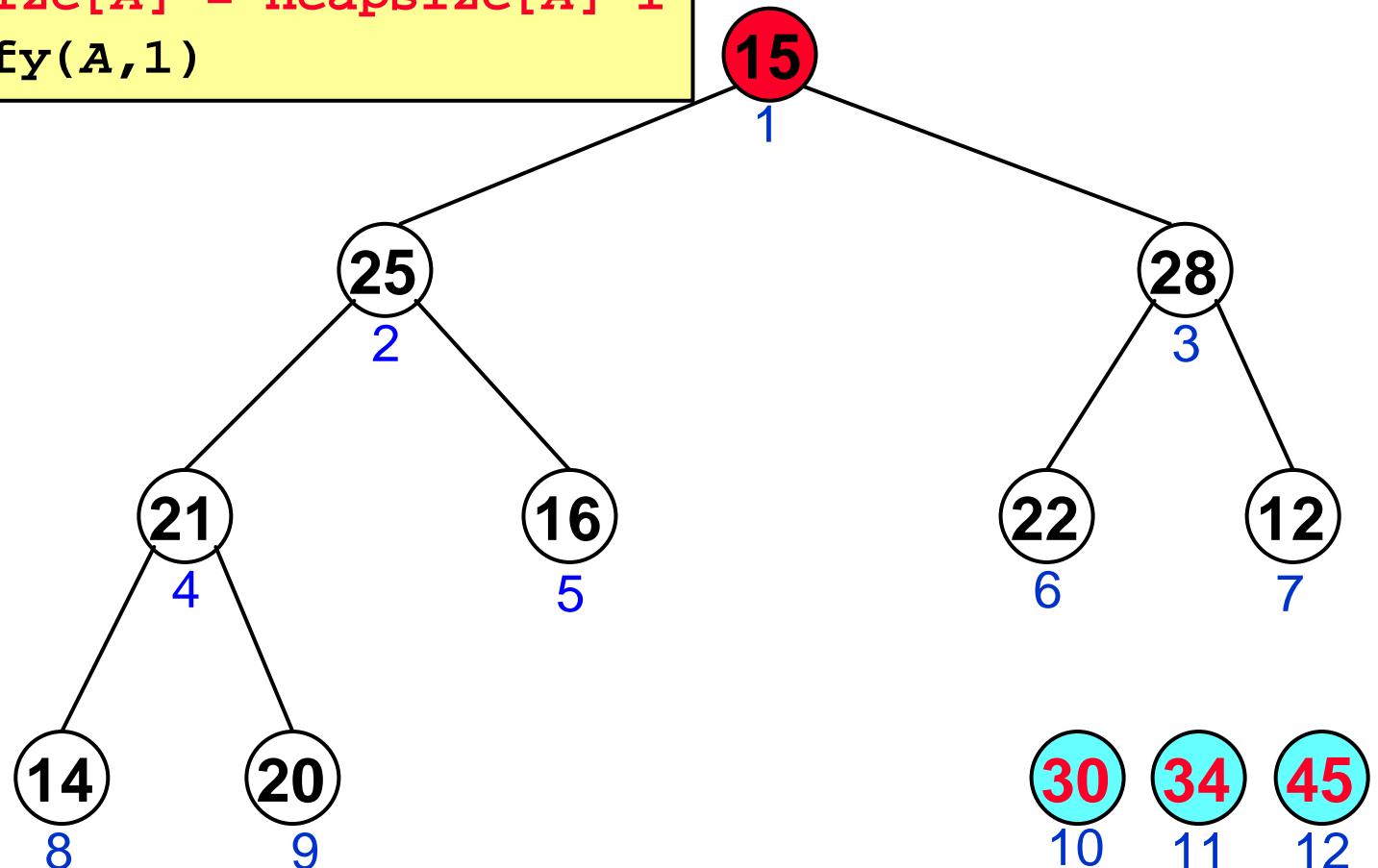
Heap Sort

```
Heap-Sort(A)
```

```
...
FOR i = length[A] DOWNTON 2
    DO "scambia A[1] e A[i]"
        heapsize[A] = heapsize[A]-1
    Heapify(A,1)
```

$i = 10$

$\text{heapsize}[A] = 9$



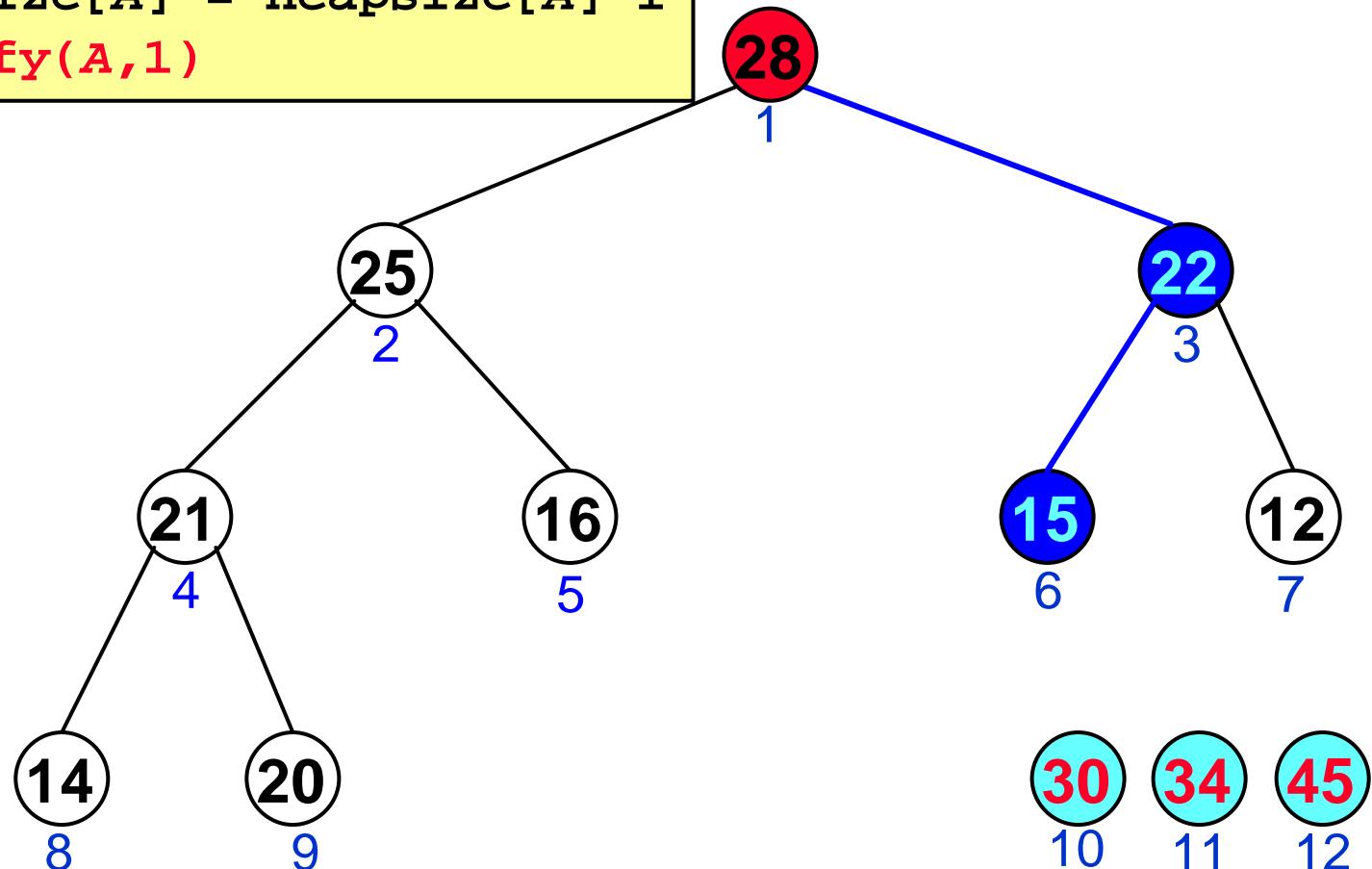
Heap Sort

```
Heap-Sort(A)
```

```
...
FOR i = length[A] DOWNTON 2
    DO "scambia A[1] e A[i]"
    heapsize[A] = heapsize[A]-1
    Heapify(A,1)
```

i=10

heapsize[A]=9



Heap Sort

```
Heap-Sort(A)
```

```
...
```

```
FOR i = length[A] DOWNTTO 2
```

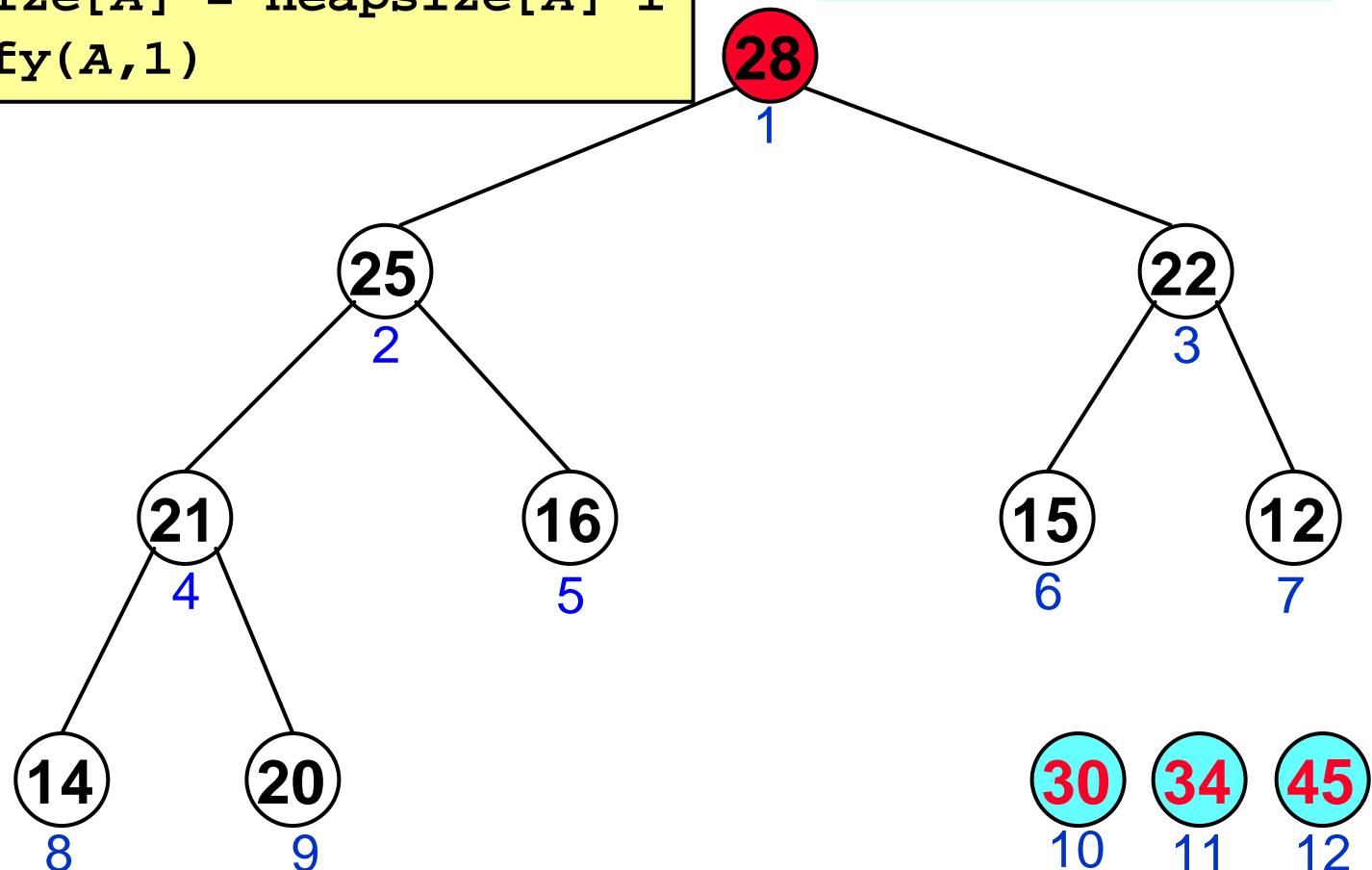
```
DO "scambia A[1] e A[i]"
```

```
heapsize[A] = heapsize[A]-1
```

```
Heapify(A,1)
```

i=9

heapsize[A]=9



Heap Sort

```
Heap-Sort(A)
```

```
...
```

```
FOR i = length[A] DOWNTON 2
```

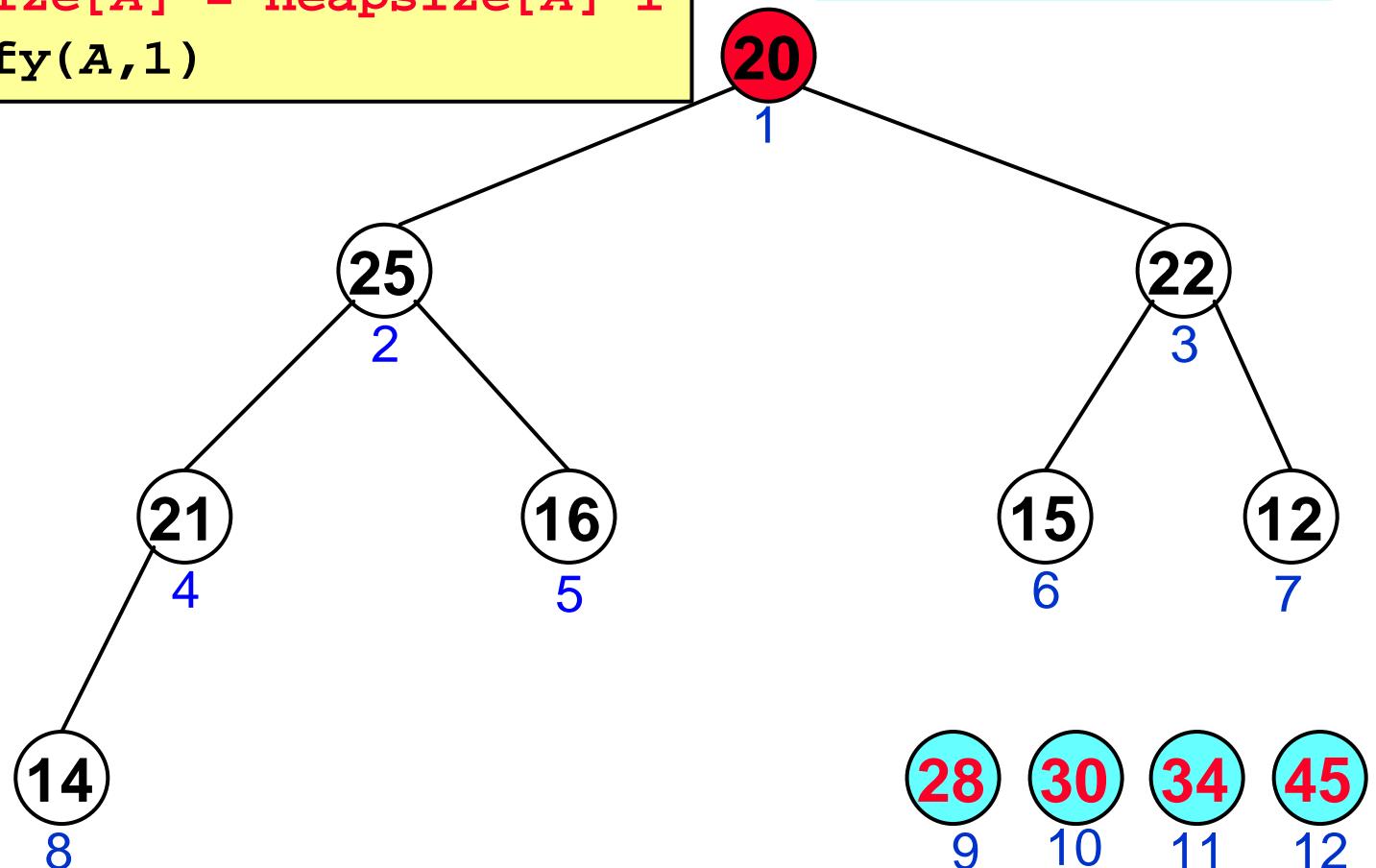
```
DO "scambia A[1] e A[i]"
```

```
heapsize[A] = heapsize[A]-1
```

```
Heapify(A,1)
```

i=9

heapsize[A]=8



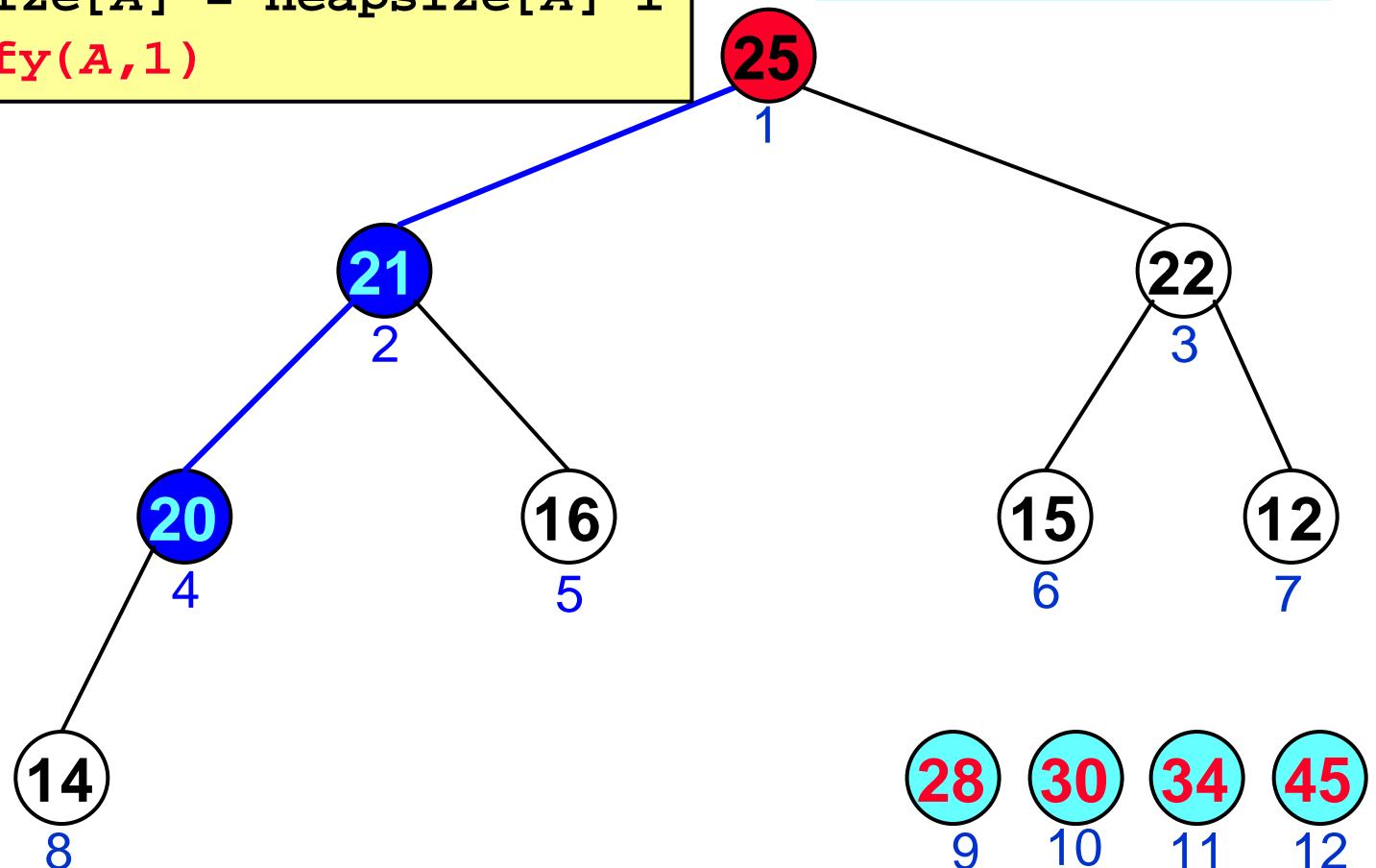
Heap Sort

```
Heap-Sort(A)
```

```
...
FOR i = length[A] DOWNTON 2
    DO "scambia A[1] e A[i]"
    heapsize[A] = heapsize[A]-1
    Heapify(A,1)
```

$i=9$

$\text{heapsize}[A]=8$



Heap Sort

```
Heap-Sort(A)
```

```
...
```

```
FOR i = length[A] DOWNTTO 2
```

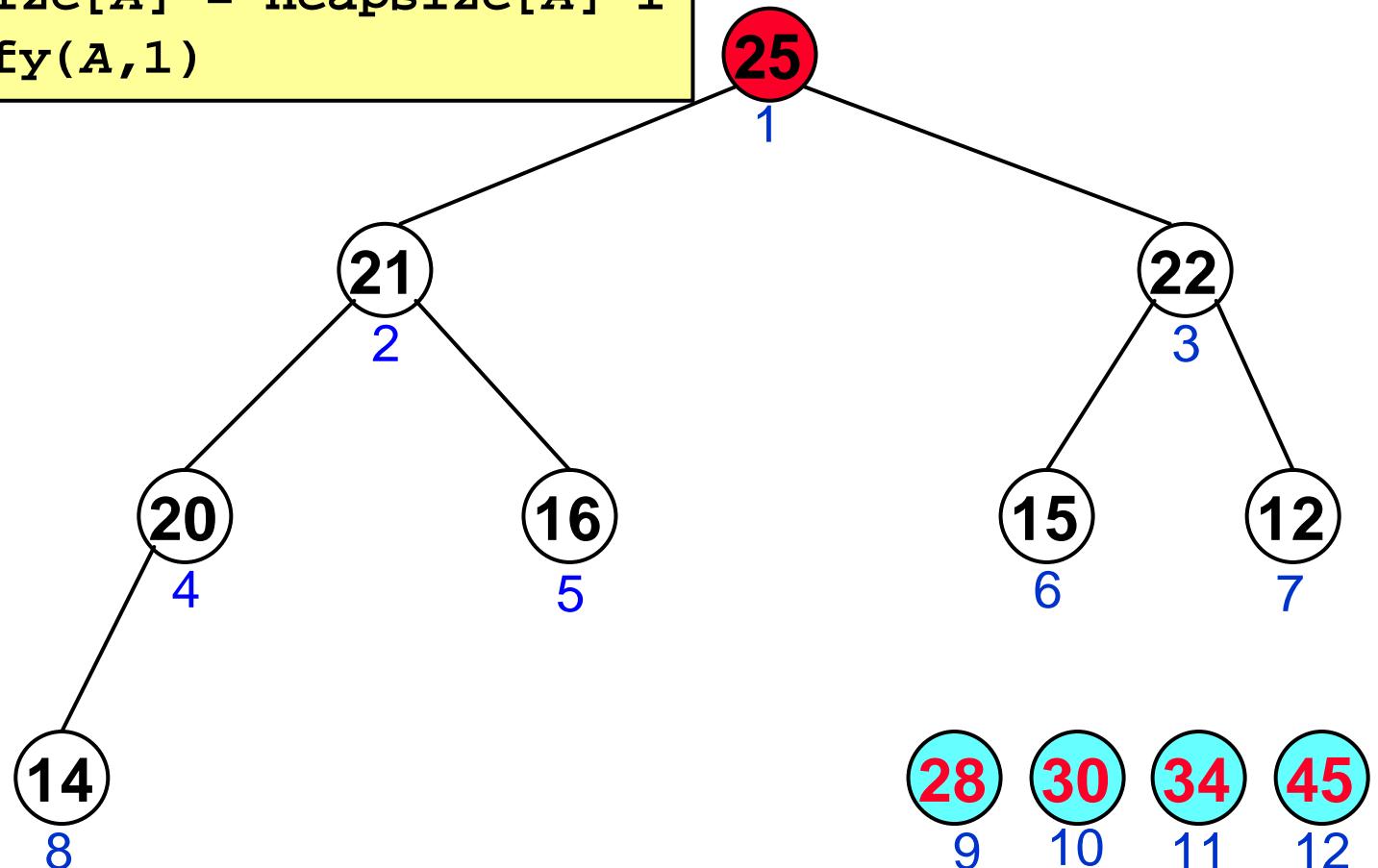
```
DO "scambia A[1] e A[i]"
```

```
heapsize[A] = heapsize[A]-1
```

```
Heapify(A,1)
```

$i=8$

$\text{heapsize}[A]=8$



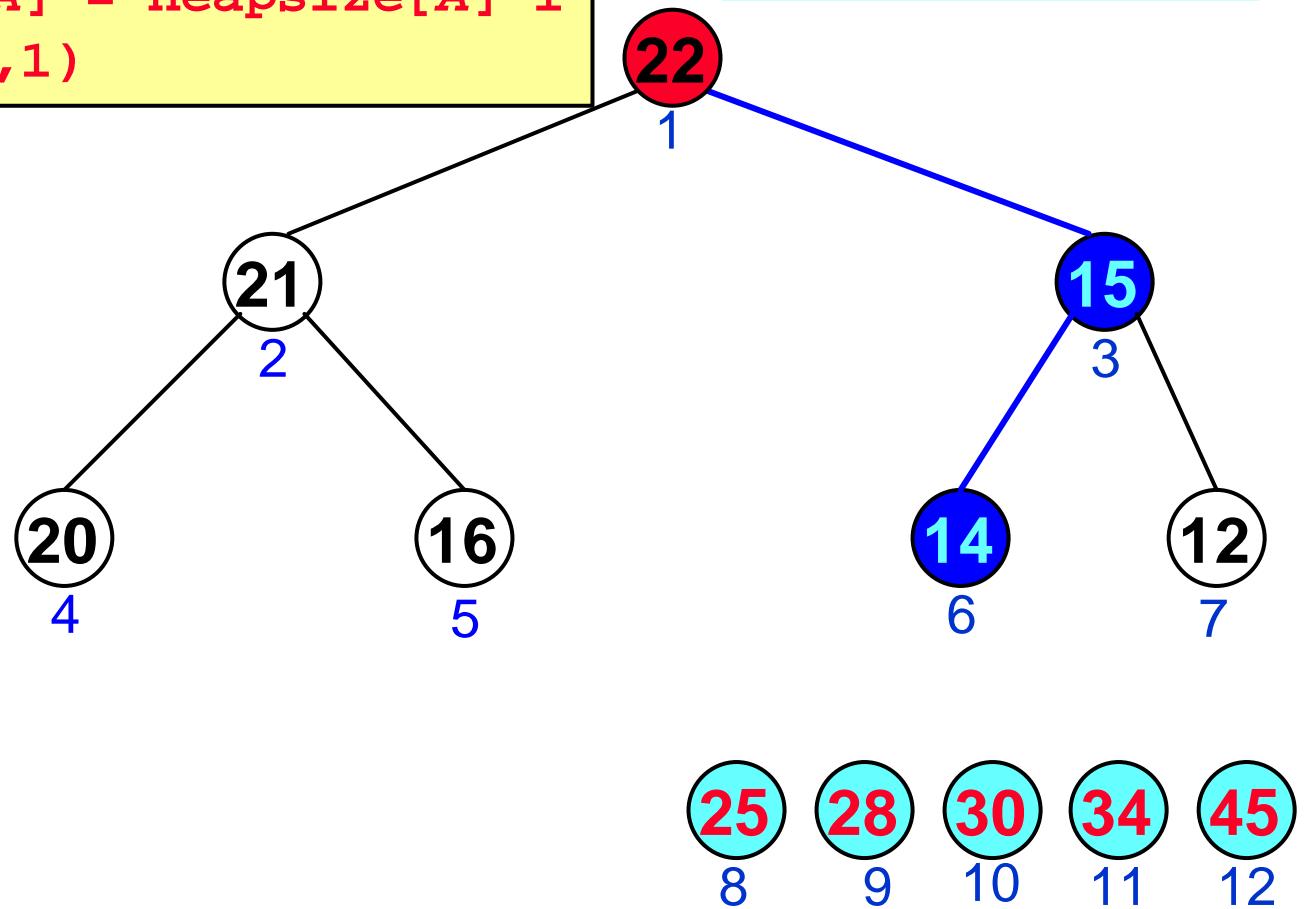
Heap Sort

```
Heap-Sort(A)
```

```
...
FOR i = length[A] DOWNTON 2
    DO "scambia A[1] e A[i]"
        heapsize[A] = heapsize[A]-1
    Heapify(A,1)
```

$i=8$

$\text{heapsize}[A]=7$



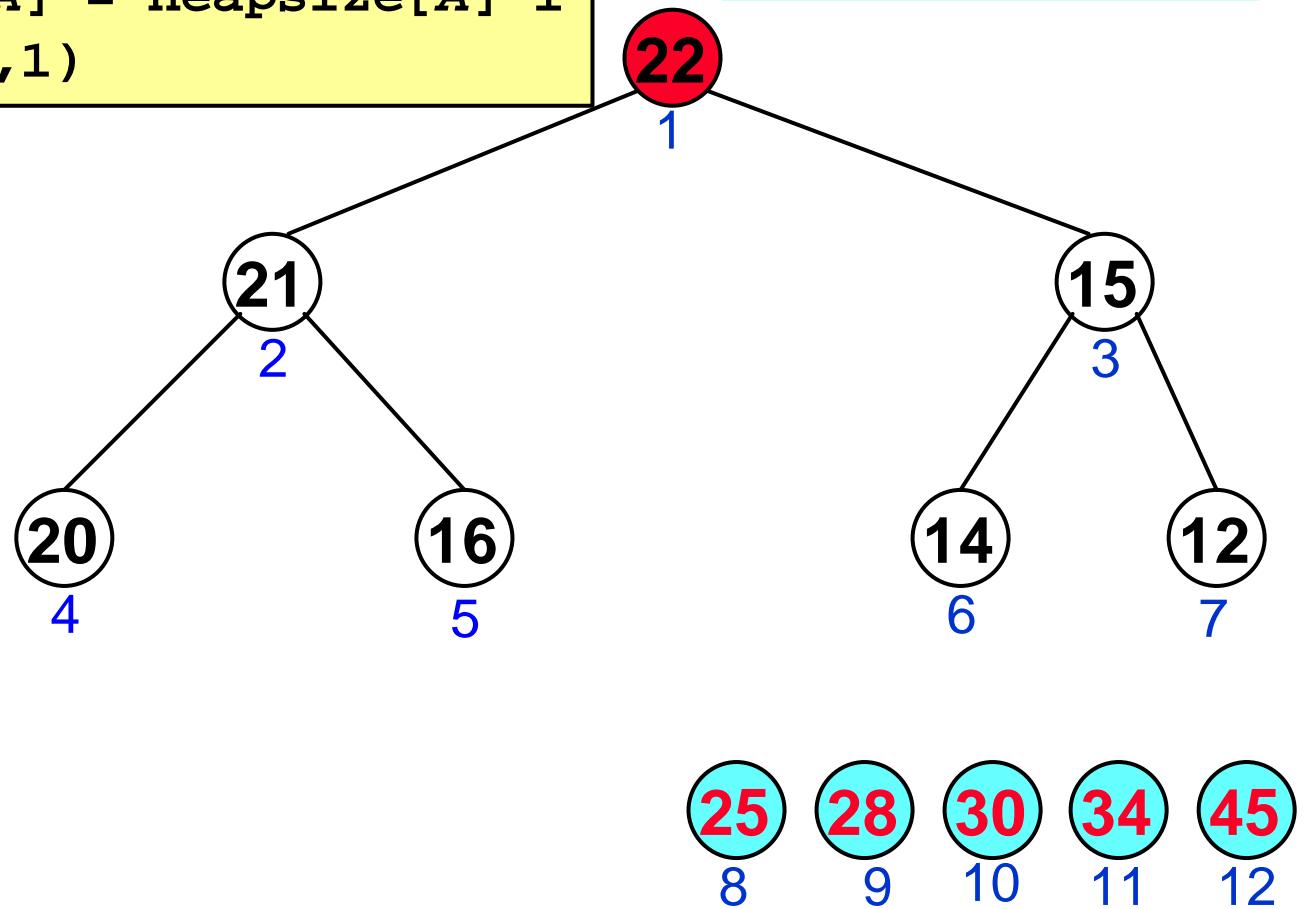
Heap Sort

```
Heap-Sort(A)
```

```
...
FOR i = length[A] DOWNTON 2
    DO "scambia A[1] e A[i]"
    heapsize[A] = heapsize[A]-1
    Heapify(A,1)
```

$i=7$

$\text{heapsize}[A]=7$



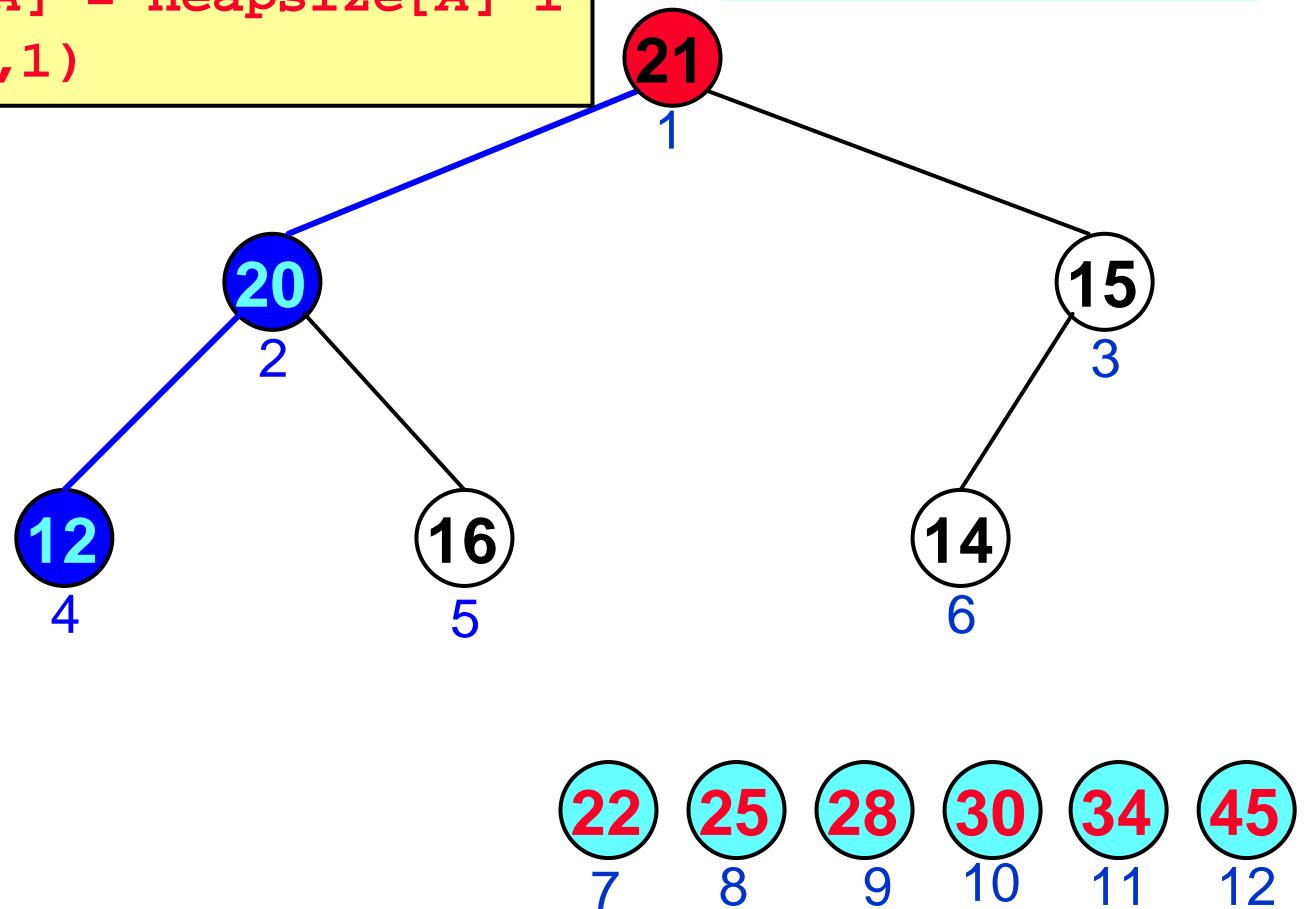
Heap Sort

```
Heap-Sort(A)
```

```
...
FOR i = length[A] DOWNTON 2
    DO "scambia A[1] e A[i]"
        heapsize[A] = heapsize[A]-1
    Heapify(A,1)
```

$i=7$

$\text{heapsize}[A]=6$



Heap Sort

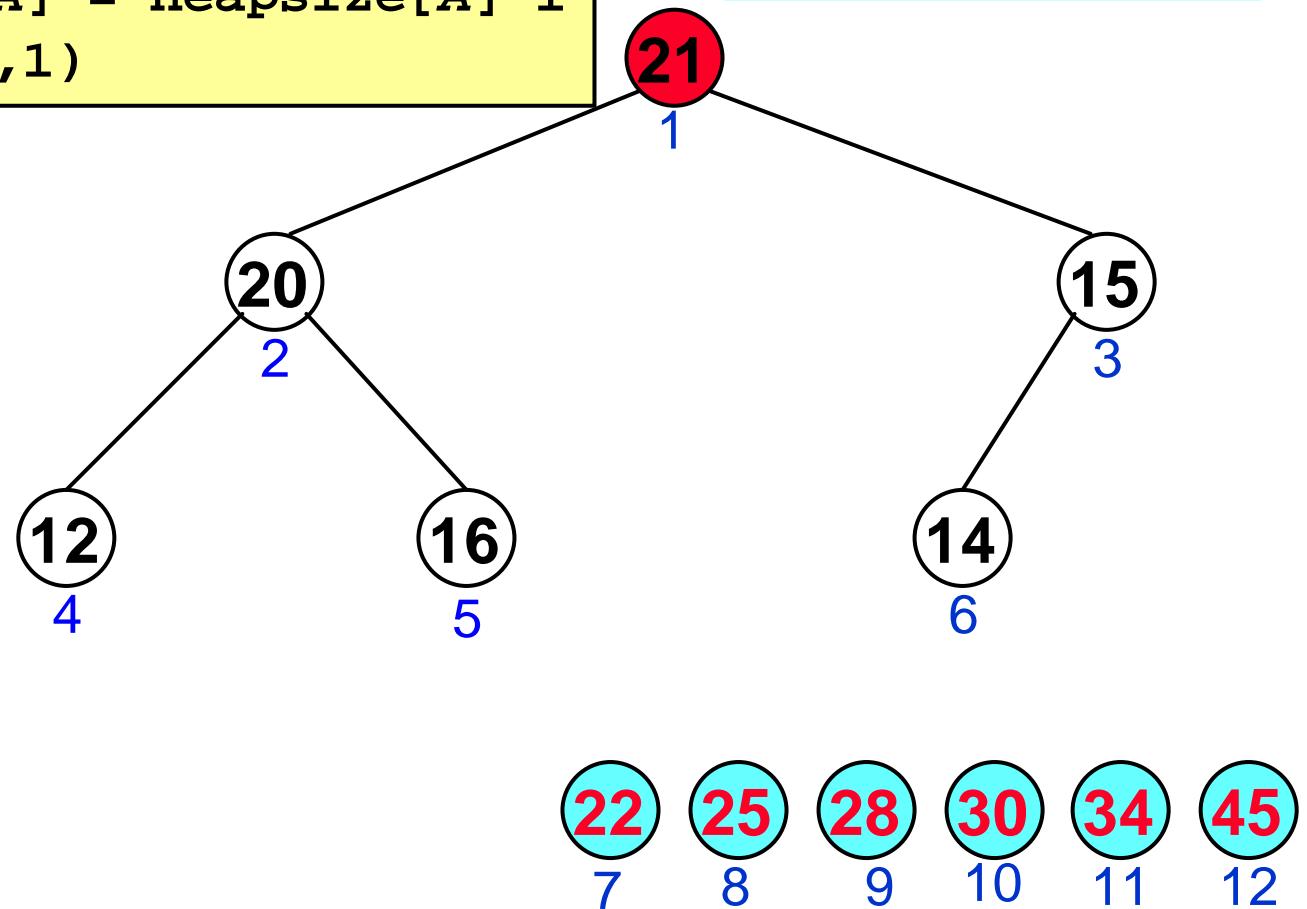
```
Heap-Sort(A)
```

```
...
```

```
FOR i = length[A] DOWNTON 2
    DO "scambia A[1] e A[i]"
    heapsize[A] = heapsize[A]-1
    Heapify(A,1)
```

i=6

heapsize[A]=6



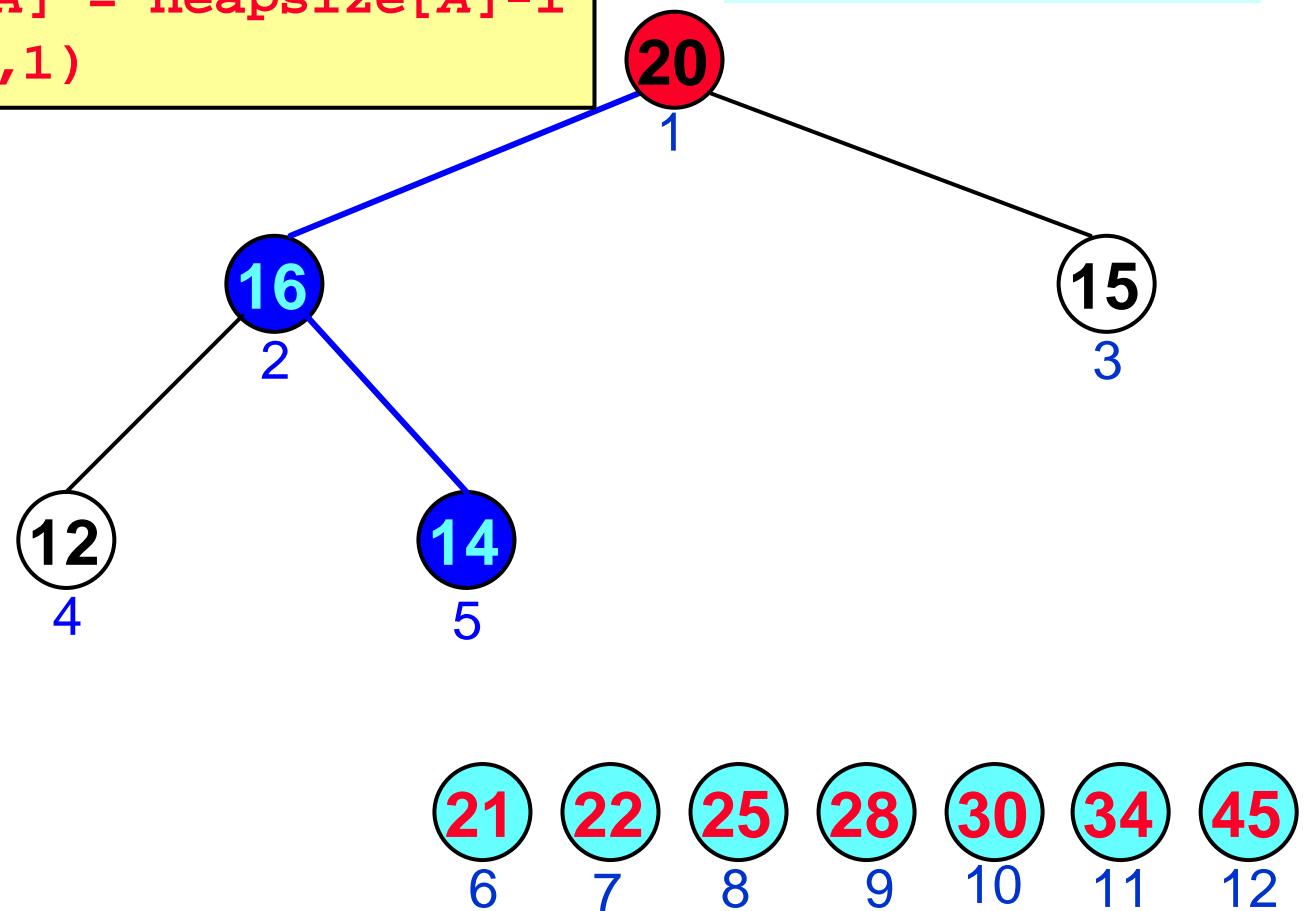
Heap Sort

```
Heap-Sort(A)
```

```
...
FOR i = length[A] DOWNTON 2
    DO "scambia A[1] e A[i]"
        heapsize[A] = heapsize[A]-1
    Heapify(A,1)
```

$i=6$

$\text{heapsize}[A]=5$



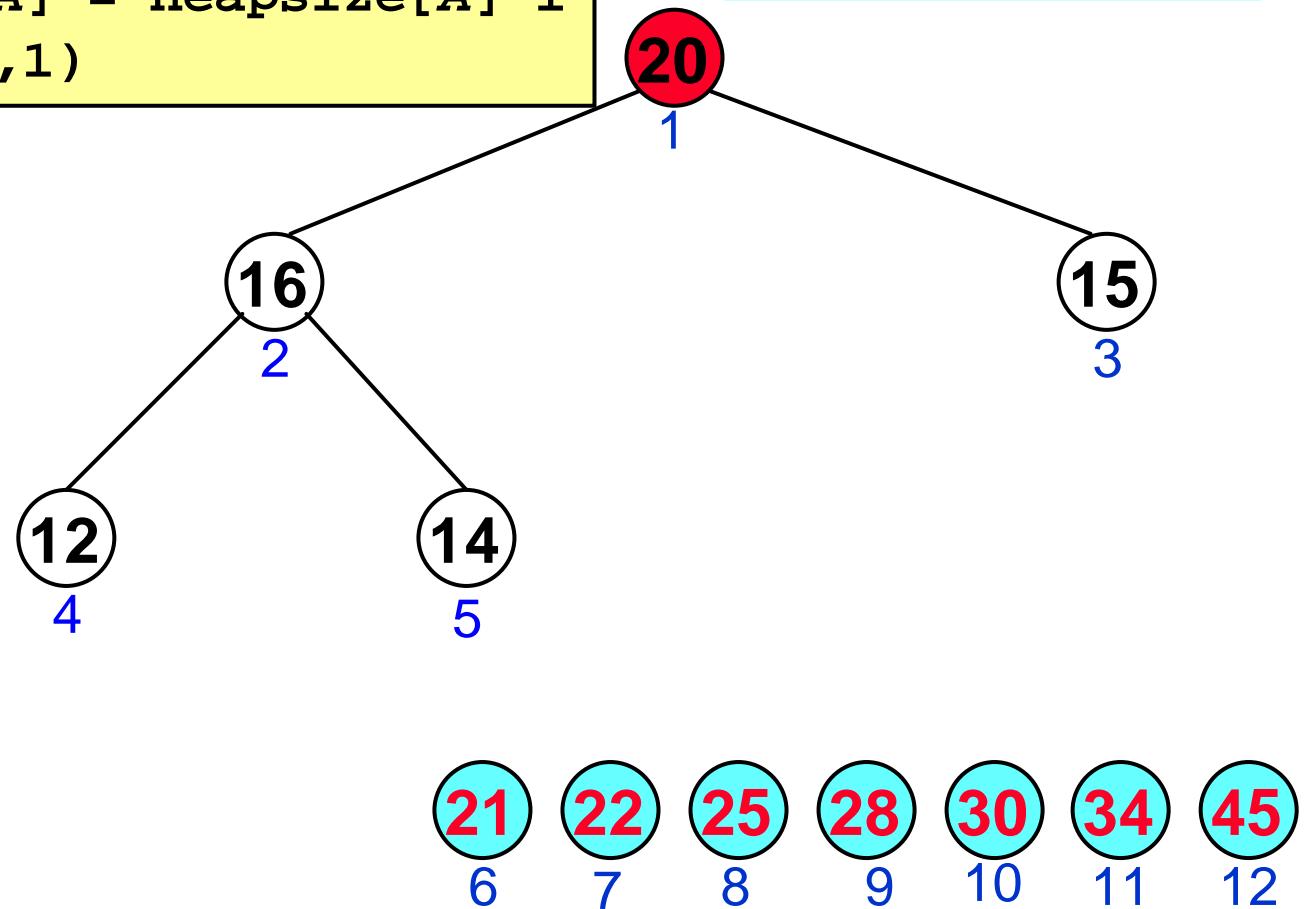
Heap Sort

```
Heap-Sort(A)
```

```
...
FOR i = length[A] DOWNTON 2
    DO "scambia A[1] e A[i]"
        heapsize[A] = heapsize[A]-1
    Heapify(A,1)
```

i=5

heapsize[A]=5



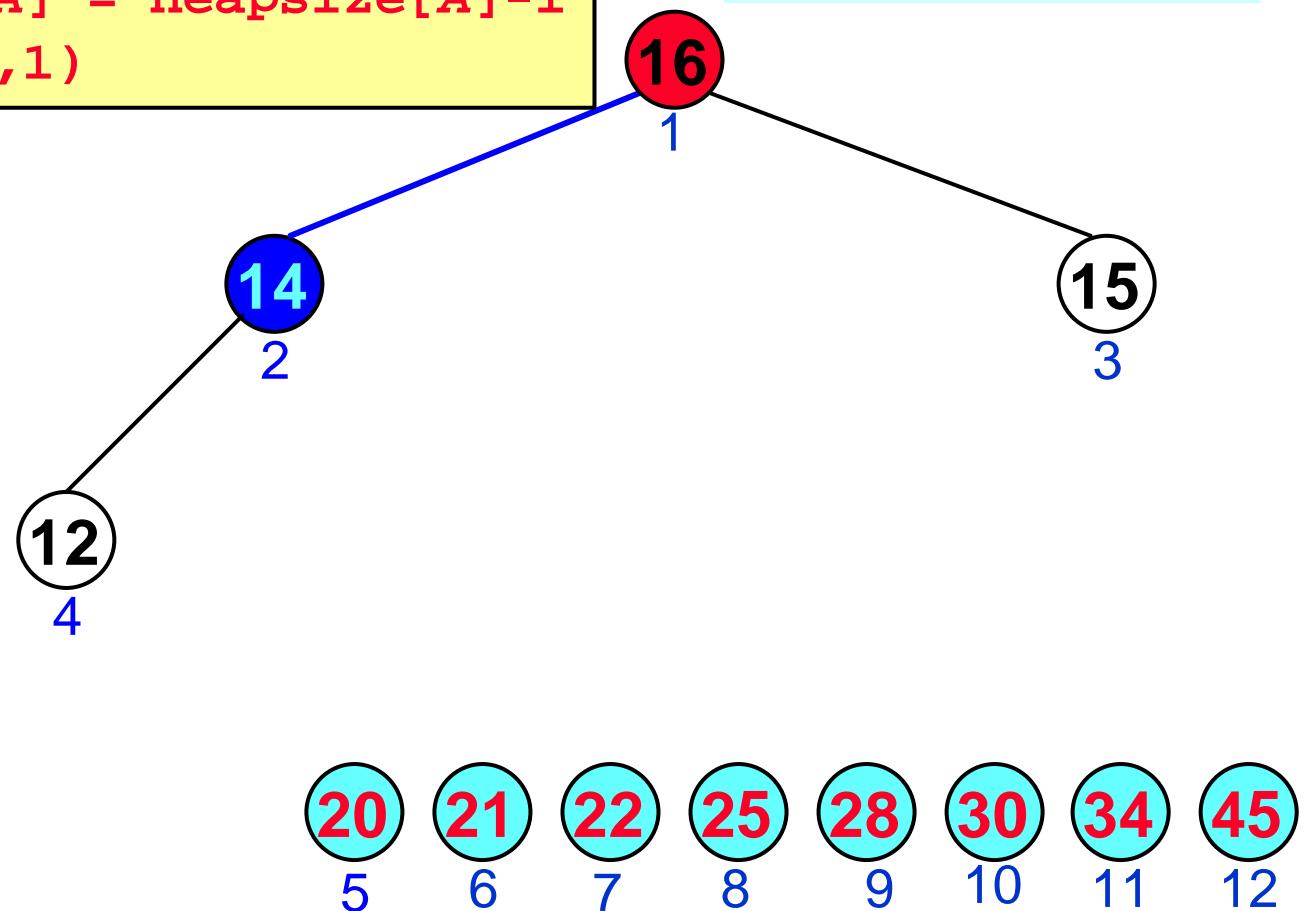
Heap Sort

```
Heap-Sort(A)
```

```
...
FOR i = length[A] DOWNTON 2
    DO "scambia A[1] e A[i]"
        heapsize[A] = heapsize[A]-1
    Heapify(A,1)
```

$i=5$

$\text{heapsize}[A]=4$



Heap Sort

```
Heap-Sort(A)
```

```
...
```

```
FOR i = length[A] DOWNTON 2
```

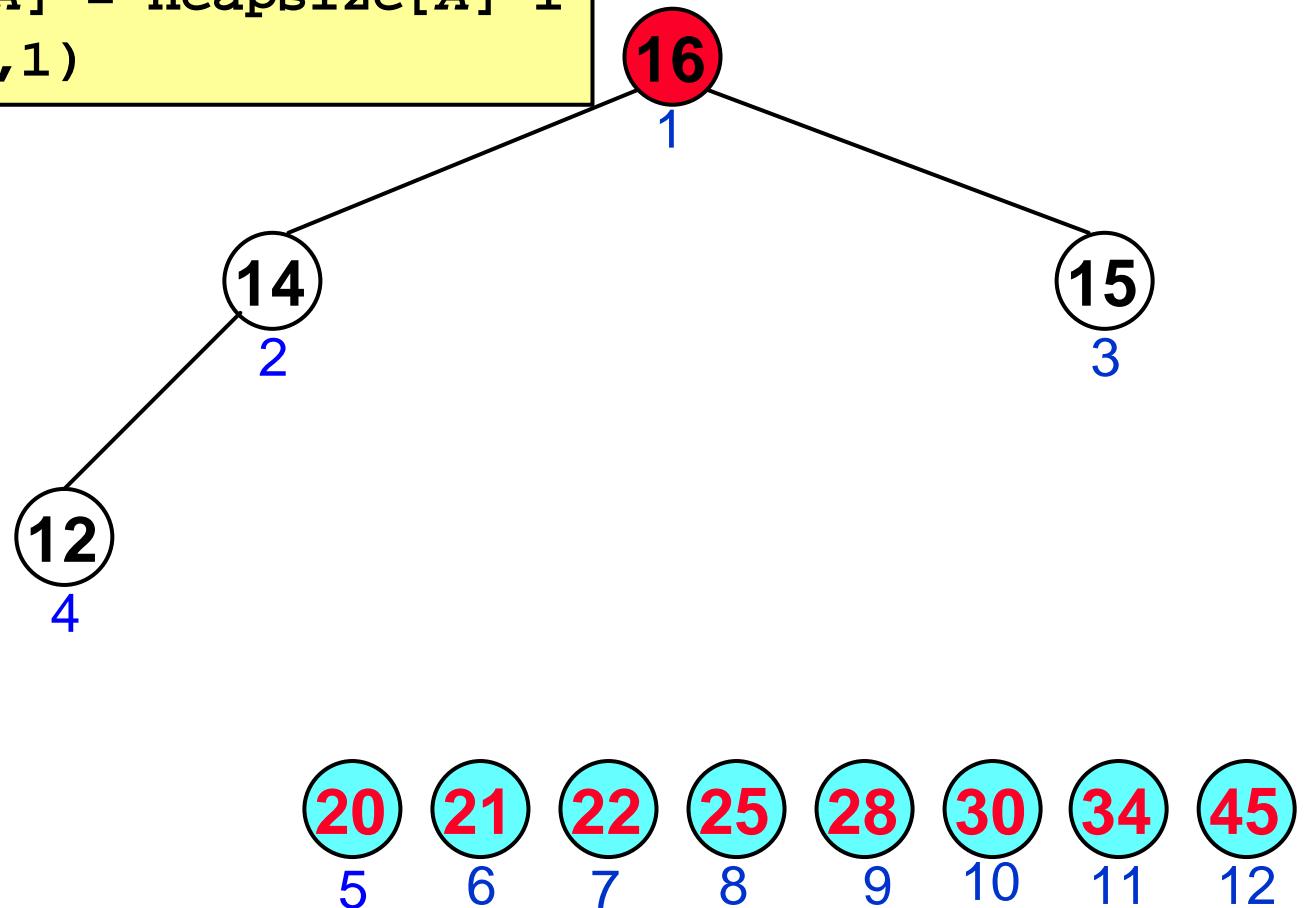
```
DO "scambia A[1] e A[i]"
```

```
heapsize[A] = heapsize[A]-1
```

```
Heapify(A,1)
```

i=4

heapsize[A]=4



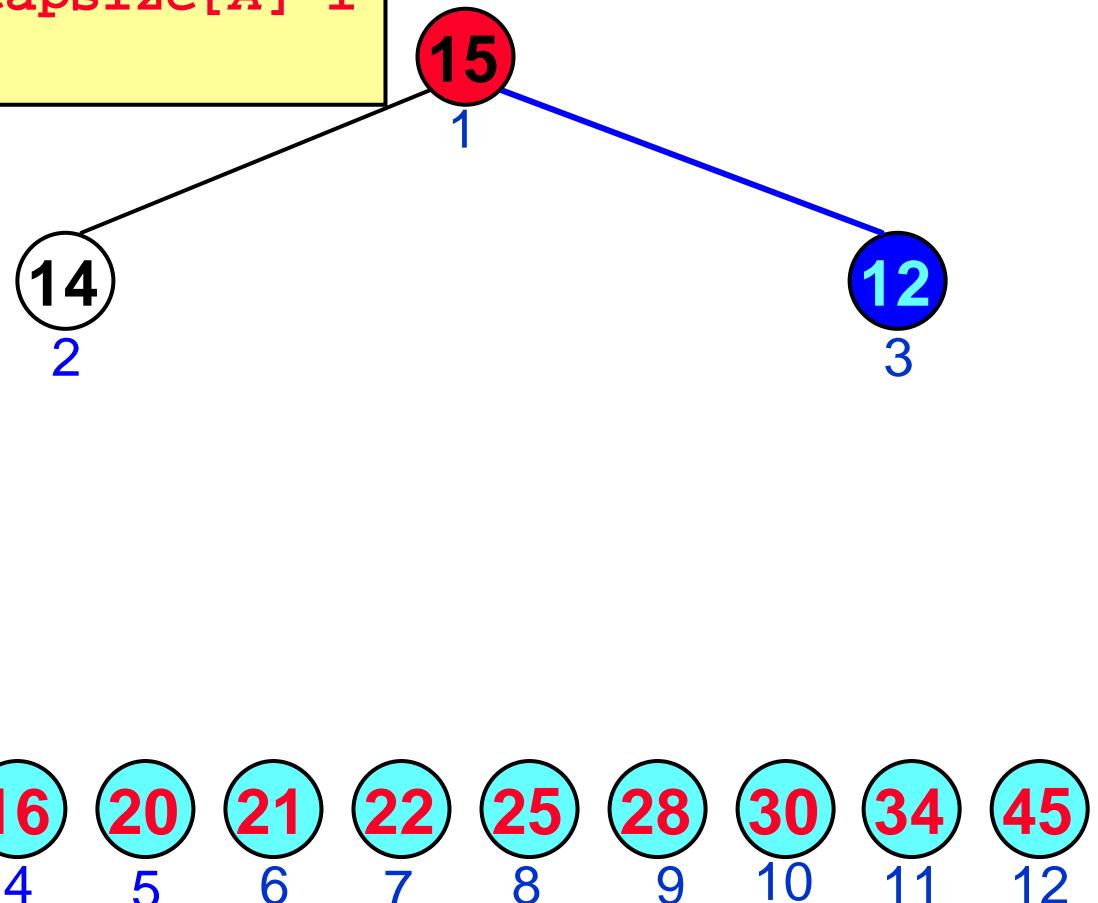
Heap Sort

```
Heap-Sort(A)
```

```
...
FOR i = length[A] DOWNTTO 2
    DO "scambia A[1] e A[i]"
        heapsize[A] = heapsize[A]-1
    Heapify(A,1)
```

$i=4$

$\text{heapsize}[A]=3$



Heap Sort

```
Heap-Sort(A)
```

```
...
```

```
FOR i = length[A] DOWNTTO 2
```

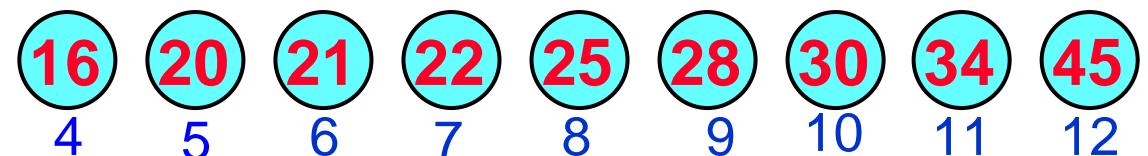
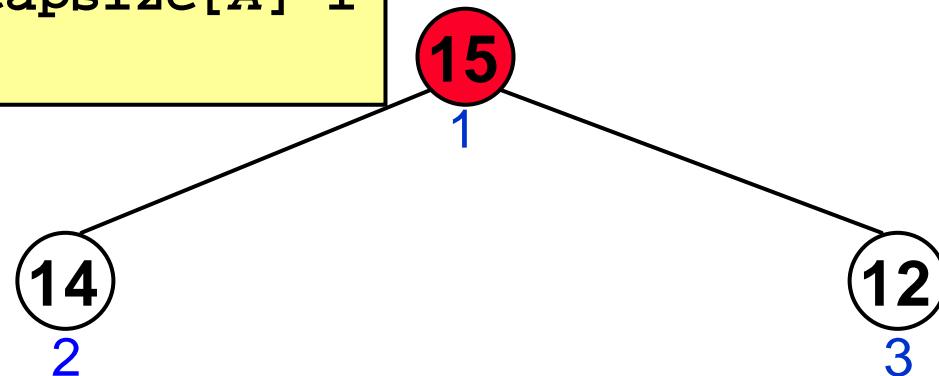
```
DO "scambia A[1] e A[i]"
```

```
heapsize[A] = heapsize[A]-1
```

```
Heapify(A,1)
```

$i=3$

$\text{heapsize}[A]=3$



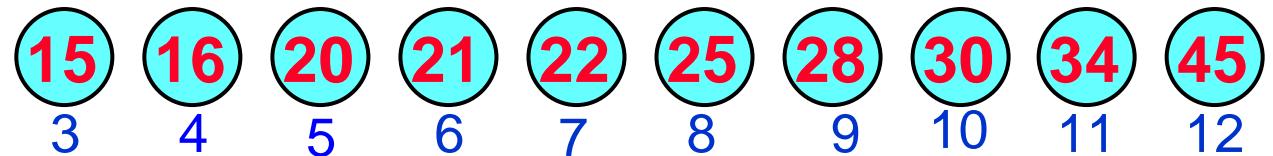
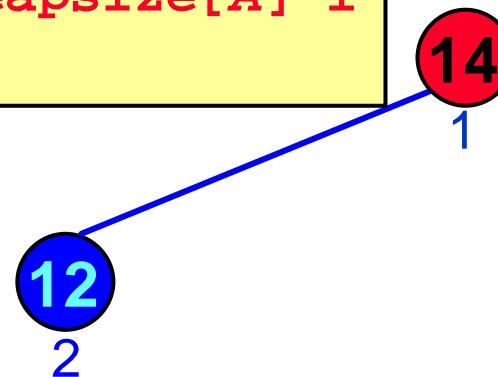
Heap Sort

```
Heap-Sort(A)
```

```
...
FOR i = length[A] DOWNTTO 2
    DO "scambia A[1] e A[i]"
        heapsize[A] = heapsize[A]-1
    Heapify(A,1)
```

$i=3$

$\text{heapsize}[A]=2$



Heap Sort

```
Heap-Sort(A)
```

```
...
```

```
FOR i = length[A] DOWNTTO 2
```

```
DO "scambia A[1] e A[i]"
```

```
heapsize[A] = heapsize[A]-1
```

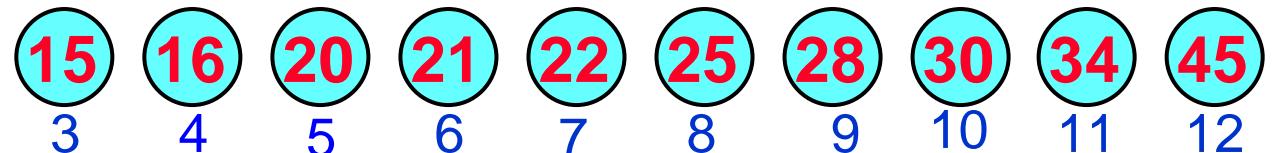
```
Heapify(A,1)
```

i=2

heapsize[A]=2

14
1

12
2



Heap Sort

```
Heap-Sort(A)
```

```
...
FOR i = length[A] DOWNTON 2
    DO "scambia A[1] e A[i]"
        heapsize[A] = heapsize[A]-1
    Heapify(A,1)
```

$i=2$

$\text{heapsize}[A]=1$

12
1



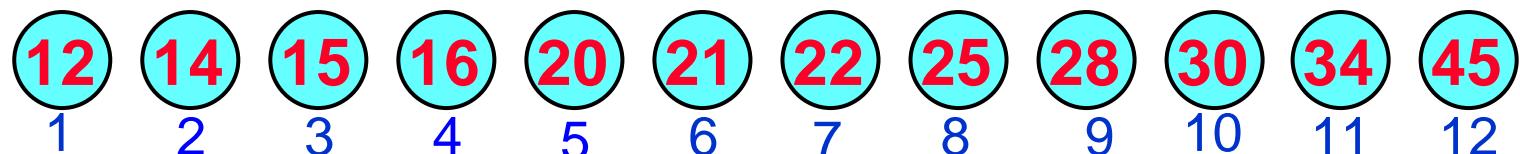
Heap Sort

```
Heap-Sort(A)
```

```
...
FOR i = length[A] DOWNTTO 2
    DO "scambia A[1] e A[i]"
        heapsize[A] = heapsize[A]-1
    Heapify(A,1)
```

i= 1

heapsize[A]=1



Heap Sort

```
Heap-Sort(A)
```

```
...
FOR i = length[A] DOWNTTO 2
    DO "scambia A[1] e A[i]"
        heapsize[A] = heapsize[A]-1
    Heapify(A,1)
```

i=1

heapsize[A]=1

L'array A è ordinato!

1	2	3	4	5	6	7	8	9	10	11	12
12	14	15	16	20	21	22	25	28	30	34	45

Heap Sort

Heap-Sort(A)

Costruisci-Heap(A) } = $O(n)$

FOR $i = \text{length}[A]$ DOWNTO 2

DO "scambia $A[1]$ e $A[i]$ " } = $O(1)$
heapsize[A] = heapsize[A] - 1
Heapify($A, 1$) } = $O(\log n)$

Complessità di Heap Sort

Nel caso peggiore *Heap-Sort* chiama

- *una volta Costruisci-Heap;*
- *n-1 volte Heapify sullo Heap corrente*

$$T(n) = \max(O(n), (n-1) \cdot \max(O(1), T(\text{Heapify})))$$

Complessità di Heap Sort

Nel caso peggiore **Heap-Sort** chiama

- **una volta Costruisci-Heap;**
- **$n-1$ volte Heapify sull'intero Heap .**

$$\begin{aligned} T(n) &= \max(O(n), (n-1) \cdot \max(O(1), T(\text{Heapify}))) \\ &= \max(O(n), \max(O(n), O(n \log n))) \end{aligned}$$

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

HeapSort: conclusioni

HeapSort

- Algoritmo di ordinamento *sul posto per confronto* che impiega tempo $O(n \log n)$.
- Algoritmo non immediato né ovvio.
- Sfrutta le proprietà della struttura dati astratta *Heap*.

HeapSort: conclusioni

HeapSort dimostra che:

- scegliere una buona rappresentazione per i dati spesso facilita la progettazione di buoni algoritmi;
- importante pensare a quale può essere una buona rappresentazione dei dati prima di implementare una soluzione.

Algoritmi e Strutture Dati

QuickSort

QuickSort

Algoritmo di ordinamento “*sul posto*” che ha tempo di esecuzione che *asintoticamente* è:

- $O(n^2)$ nel *caso peggiore*
- $O(n \log n)$ nel *caso medio*

Nonostante le cattive prestazioni nel *caso peggiore*, rimane il miglior algoritmo di ordinamento *in media*

QuickSort

È basato sulla metodologia *Divide et Impera*:

Dividi: L'array $A[p...r]$ viene “**partizionato**” (**tramite spostamenti di elementi**) in due sottoarray **non vuoti** $A[p...q]$ e $A[q+1...r]$ in cui:

- *ogni elemento di $A[p...q]$ è minore o uguale ad ogni elemento di $A[q+1...r]$*

Conquista: i due sottoarray $A[p...q]$ e $A[q+1...r]$ vengono ordinati ricorsivamente con *QuickSort*

Combina: i sottoarray vengono ordinati anche reciprocamente, quindi non è necessaria alcuna combinazione. $A[p...r]$ è già ordinato.

Algoritmo QuickSort

```
Quick-Sort( $A, p, r$ )
  IF  $p < r$ 
    THEN  $q = \text{Partiziona}(A, p, r)$ 
          Quick-Sort( $A, p, q$ )
          Quick-Sort( $A, q + 1, r$ )
```

Algoritmo QuickSort

Indice mediano

```
Quick-Sort( $A, p, r$ )
  IF  $p < r$ 
    THEN  $q = \text{Partiziona}(A, p, r)$ 
          Quick-Sort( $A, p, q$ )
          Quick-Sort( $A, q + 1, r$ )
```

q è l'indice che **divide** l'array in due **sottoarray** dove

- tutti gli elementi a sinistra di q (compreso l'elemento in posizione q) sono minori o uguali tutti gli elementi a destra di q

Algoritmo QuickSort

**Ordinamento dei
due sottoarray**

```
Quick-Sort( $A, p, r$ )
```

```
  IF  $p < r$ 
```

```
    THEN  $q = \text{Partiziona}(A, p, r)$ 
```

```
      Quick-Sort( $A, p, q$ )
```

```
      Quick-Sort( $A, q + 1, r$ )
```

Poiché il *sottoarray* di *sinistra* contiene elementi tutti *minori o uguali* a tutti quelli del *sottoarray* di *destra*, *ordinare i due sottoarray separatamente* fornisce la *soluzione del problema*

Algoritmo QuickSort

passo Dividi

Quick-Sort(A, p, r)

IF $p < r$

THEN $q = \text{Partiziona}(A, p, r)$

Quick-Sort(A, p, q)

Quick-Sort($A, q + 1, r$)

Partition è la **chiave** di tutto l'algoritmo !

IMPORTANTE: **q deve** essere **strettamente minore** di **r**:

$p \leq q < r$

Algoritmo Partiziona

L'array $A[p...r]$ viene “**suddiviso**” in due sotto-array “**non vuoti**” $A[p...q]$ e $A[q+1...r]$ in cui ogni elemento di $A[p...q]$ è minore o uguale ad ogni elemento di $A[q+1...r]$:

- l'algoritmo sceglie un valore dell'array che fungerà da elemento “**spartiacque**” tra i due sotto-array, detto valore **pivot**.
- sposta i **valori maggiori del pivot** verso l'estremità destra dell'array e i **valori minori** verso quella sinistra.

q dipenderà dal valore **pivot** scelto: sarà l'estremo della partizione a partire da sinistra nella quale, alla fine, si troveranno solo elementi **minori o uguali al pivot**.

Algoritmo Partiziona

```
int Partiziona(A,p,r)
    x = A[p]
    i = p - 1
    j = r + 1
    REPEAT
        REPEAT j = j - 1
            UNTIL A[j] ≤ x
        REPEAT i = i + 1
            UNTIL A[i] ≥ x
        IF i < j
            THEN "scambia A[i] con A[j]"
        UNTIL i ≥ j
    return j
```

Algoritmo Partiziona

```
int Partiziona(A, p, r)
    x = A[p]
    i = p - 1
    j = r + 1
    REPEAT
        REPEAT j = j
            UNTIL A[j] ≥ x
        REPEAT i = i
            UNTIL A[i] ≤ x
        IF i < j
            THEN "scambia A[i] con A[j]"
    UNTIL i ≥ j
return j
```

Elemento Pivot

Gli elementi minori o uguali al Pivot verranno spostati tutti verso sinistra

➤ Gli elementi maggiori o uguali al Pivot verranno spostati tutti verso destra

Algoritmo Partiziona

```
int Partiziona(A, p, r)
    x = A[p]
    i = p - 1
    j = r + 1
    REPEAT ←
        REPEAT j = j - 1
            UNTIL A[j] ≤ x
        REPEAT i = i + 1
            UNTIL A[i] ≥ x
        IF i < j
            THEN "scambia A[i] con A[j]"
    UNTIL i ≥ j
return j
```

*Il ciclo continua finché
i incrocia j*

Algoritmo Partiziona

```
int Partiziona(A, p, r)
    x = A[p]
    i = p - 1
    j = r + 1
    REPEAT
        REPEAT j = j - 1
            UNTIL A[j] ≤ x
        REPEAT i = i + 1
            UNTIL A[i] ≥ x
        IF i < j
            THEN "scambia A[i] con A[j]"
        UNTIL i ≥ j
    return j
```

Cerca il primo elemento da destra che sia minore o uguale al Pivot x

Algoritmo Partiziona

```
int Partiziona(A, p, r)
```

```
    x = A[p]
```

```
    i = p - 1
```

```
    j = r + 1
```

```
REPEAT
```

```
    REPEAT j = j - 1
```

```
        UNTIL A[j] ≤ x
```

```
    REPEAT i = i + 1
```

```
        UNTIL A[i] ≥ x
```

```
    IF i < j
```

```
        THEN "scambia A[i] con A[j]"
```

```
    UNTIL i ≥ j
```

```
return j
```

Cerca il primo elemento da sinistra che sia maggiore o uguale al Pivot x

Algoritmo Partiziona

```
int Partiziona(A,p,r)
    x = A[p]
    i = p - 1
    j = r + 1
    REPEAT
        REPEAT j = j
            UNTIL A[j] ≤ x
        REPEAT i = i + 1
            UNTIL A[i] ≥ x
        IF i < j
            THEN "scambia A[i] con A[j]"
        UNTIL i ≥ j
    return j
```

Se l'array non è stato scandito completamente $i < j$ (i due non indici si incrociano) allora :

- $A[i] \leq x$
- $A[j] \geq x$

gli elementi vengono scambiati



Algoritmo Partiziona

```
int Partiziona(A, p, r)
```

```
    x = A[p]
```

```
    i = p - 1
```

```
    j = r + 1
```

```
REPEAT
```

```
    REPEAT j = j - 1
```

```
        UNTIL A[j] ≤ x
```

```
    REPEAT i = i + 1
```

```
        UNTIL A[i] ≥ x
```

```
    IF i < j
```

```
        THEN "scambia A[i] con A[j]"
```

```
        UNTIL i ≥ j
```

```
return j
```

Se l'array è stato scandito completamente $i \geq j$ (i due indici si incrociano) allora termina il ciclo

Algoritmo Partiziona

```
int Partiziona(A, p, r)
```

```
    x = A[p]
```

```
    i = p - 1
```

```
    j = r + 1
```

```
REPEAT
```

```
    REPEAT j = j - 1
```

```
        UNTIL A[j] ≤ x
```

```
    REPEAT i = i + 1
```

```
        UNTIL A[i] ≥ x
```

```
    IF i < j
```

```
        THEN "scambia A[i] con A[j]"
```

```
    UNTIL i ≥ j
```

```
return j
```

*Alla fine j è ritornato
come indice mediano
dell'array*

Algoritmo Partiziona: partiziona (A, 1, 12)

```
int Partiziona(A, p, r)
```

```
    x = A[p]
```

```
    i = p - 1
```

```
    j = r + 1
```

```
REPEAT
```

```
    REPEAT j = j - 1
```

```
        UNTIL A[j] ≤ x
```

```
    REPEAT i = i + 1
```

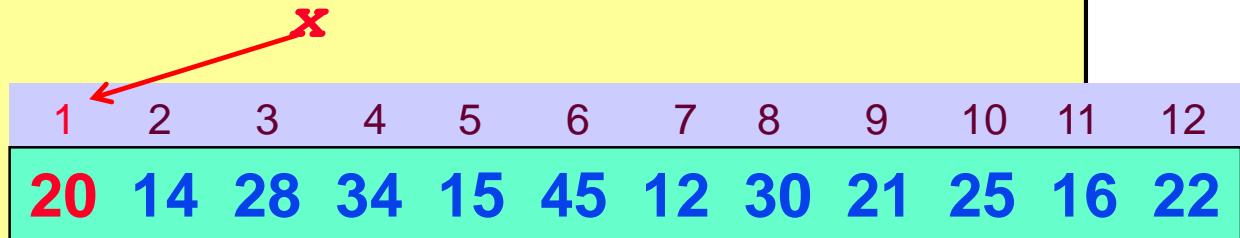
```
        UNTIL A[i] ≥ x
```

```
    IF i < j
```

```
        THEN "scambia A[i] con A[j]"
```

```
    UNTIL i ≥ j
```

```
return j
```



Algoritmo Partiziona: partiziona (A, 1, 12)

```
int Partiziona(A, p, r)
```

```
    x = A[p]
```

```
    i = p - 1
```

```
    j = r + 1
```

```
REPEAT
```

```
    REPEAT j = j - 1
```

```
        UNTIL A[j] ≤ x
```

```
    REPEAT i = i + 1
```

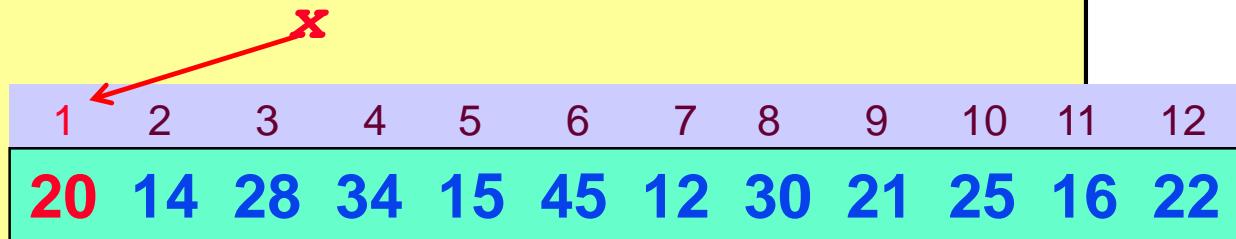
```
        UNTIL A[i] ≥ x
```

```
    IF i < j
```

```
        THEN "scambia A[i] con A[j]"
```

```
UNTIL i ≥ j
```

```
return j
```



Algoritmo Partiziona: partiziona (A, 1, 12)

```
int Partiziona(A, p, r)
```

```
    x = A[p]
```

```
    i = p - 1
```

```
    j = r + 1
```

```
    REPEAT
```

```
        REPEAT j = j - 1
```

```
            UNTIL A[j] ≤ x
```

```
        REPEAT i = i + 1
```

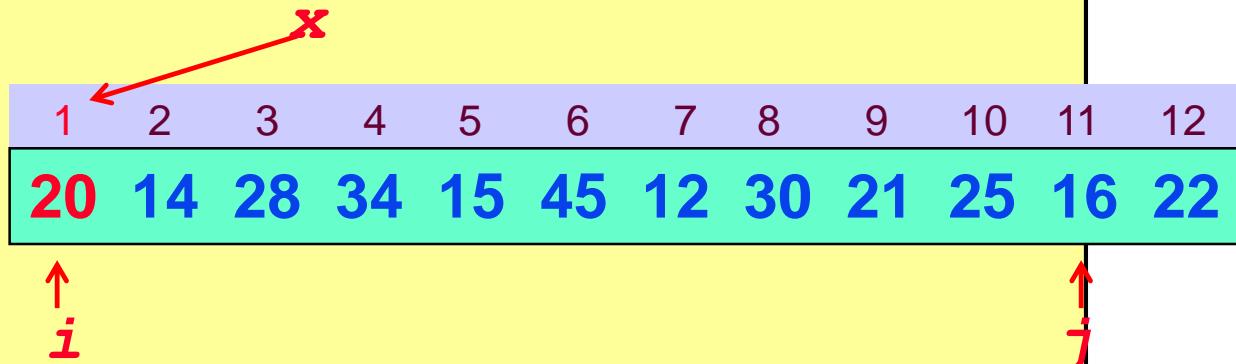
```
            UNTIL A[i] ≥ x
```

```
        IF i < j
```

```
            THEN "scambia A[i] con A[j]"
```

```
        UNTIL i ≥ j
```

```
    return j
```



Algoritmo Partiziona: partiziona (A, 1, 12)

```
int Partiziona(A, p, r)
```

```
    x = A[p]
```

```
    i = p - 1
```

```
    j = r + 1
```

```
    REPEAT
```

```
        REPEAT j = j - 1
```

```
            UNTIL A[j] ≤ x
```

```
        REPEAT i = i + 1
```

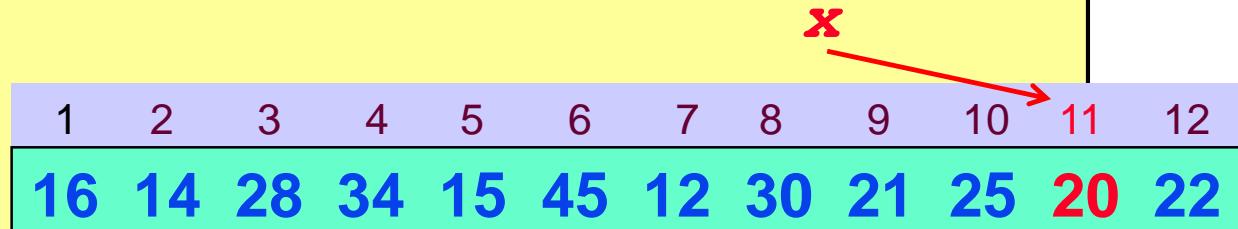
```
            UNTIL A[i] ≥ x
```

```
        IF i < j
```

```
            THEN "scambia A[i] con A[j]"
```

```
    UNTIL i ≥ j
```

```
return j
```



Algoritmo Partiziona: partiziona (A, 1, 12)

```
int Partiziona(A, p, r)
```

```
    x = A[p]
```

```
    i = p - 1
```

```
    j = r + 1
```

```
REPEAT
```

```
    REPEAT j = j - 1
```

```
        UNTIL A[j] ≤ x
```

```
    REPEAT i = i + 1
```

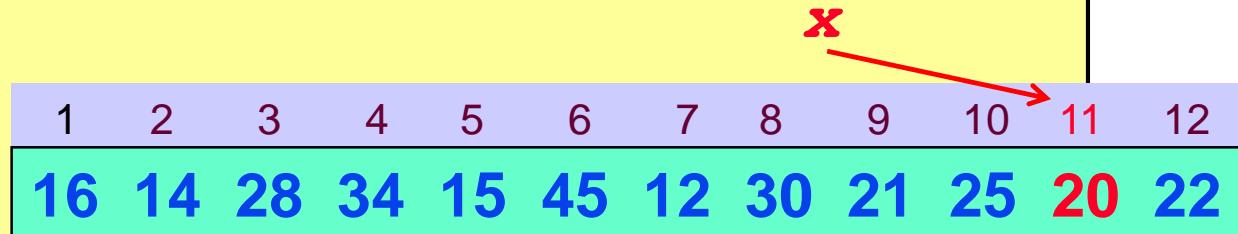
```
        UNTIL A[i] ≥ x
```

```
    IF i < j
```

```
        THEN "scambia A[i] con A[j]"
```

```
UNTIL i ≥ j
```

```
return j
```



Algoritmo Partiziona: partiziona (A, 1, 12)

```
int Partiziona(A, p, r)
```

```
    x = A[p]
```

```
    i = p - 1
```

```
    j = r + 1
```

```
REPEAT
```

```
    REPEAT j = j - 1
```

```
        UNTIL A[j] ≤ x
```

```
    REPEAT i = i + 1
```

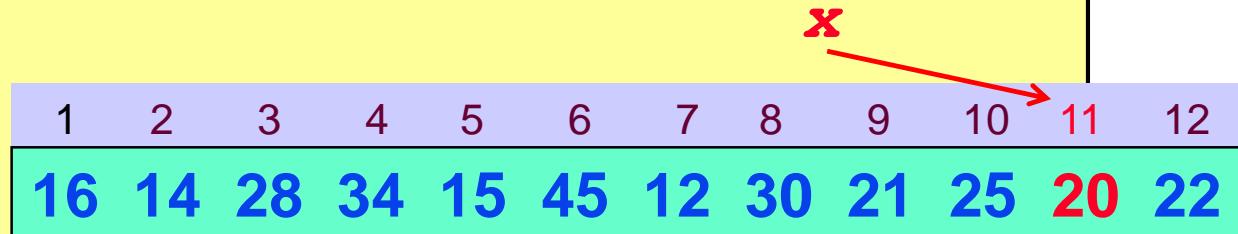
```
        UNTIL A[i] ≥ x
```

```
    IF i < j
```

```
        THEN "scambia A[i] con A[j]"
```

```
    UNTIL i ≥ j
```

```
return j
```



Algoritmo Partiziona: partiziona (A, 1, 12)

```
int Partiziona(A, p, r)
```

```
    x = A[p]
```

```
    i = p - 1
```

```
    j = r + 1
```

```
REPEAT
```

```
    REPEAT j = j - 1
```

```
        UNTIL A[j] ≤ x
```

```
    REPEAT i = i + 1
```

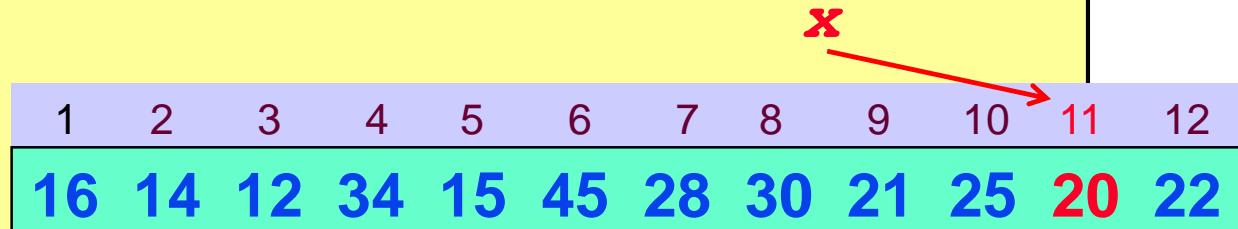
```
        UNTIL A[i] ≥ x
```

```
    IF i < j
```

```
        THEN "scambia A[i] con A[j]"
```

```
UNTIL i ≥ j
```

```
return j
```



Algoritmo Partiziona: partiziona (A, 1, 12)

```
int Partiziona(A, p, r)
```

```
    x = A[p]
```

```
    i = p - 1
```

```
    j = r + 1
```

```
REPEAT
```

```
    REPEAT j = j - 1
```

```
        UNTIL A[j] ≤ x
```

```
    REPEAT i = i + 1
```

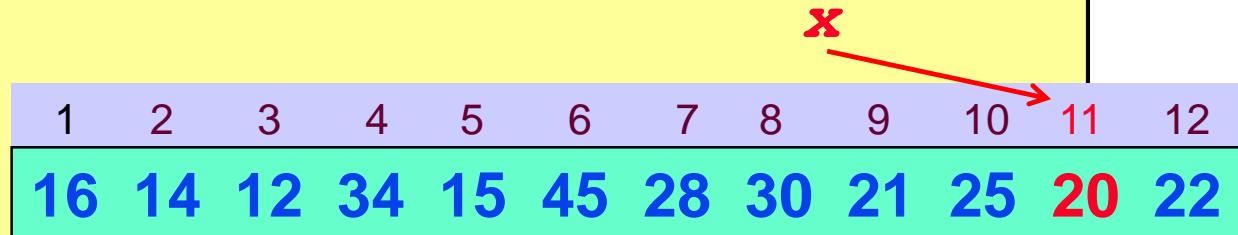
```
        UNTIL A[i] ≥ x
```

```
    IF i < j
```

```
        THEN "scambia A[i] con A[j]"
```

```
UNTIL i ≥ j
```

```
return j
```



Algoritmo Partiziona: partiziona (A, 1, 12)

```
int Partiziona(A, p, r)
```

```
    x = A[p]
```

```
    i = p - 1
```

```
    j = r + 1
```

```
REPEAT
```

```
    REPEAT j = j - 1
```

```
        UNTIL A[j] ≤ x
```

```
    REPEAT i = i + 1
```

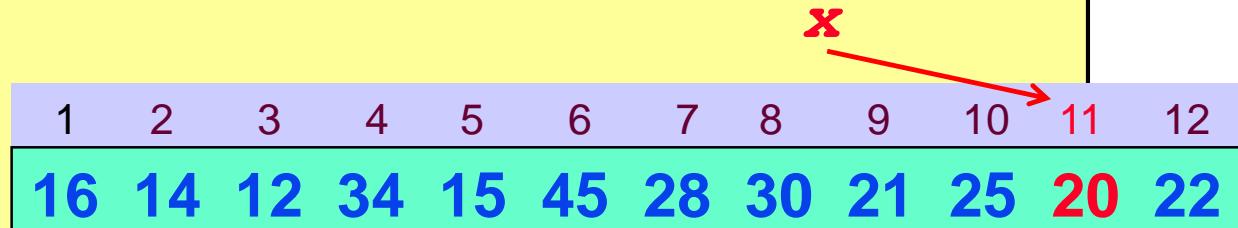
```
        UNTIL A[i] ≥ x
```

```
    IF i < j
```

```
        THEN "scambia A[i] con A[j]"
```

```
    UNTIL i ≥ j
```

```
return j
```



Algoritmo Partiziona: partiziona (A, 1, 12)

```
int Partiziona(A, p, r)
```

```
    x = A[p]
```

```
    i = p - 1
```

```
    j = r + 1
```

```
REPEAT
```

```
    REPEAT j = j - 1
```

```
        UNTIL A[j] ≤ x
```

```
    REPEAT i = i + 1
```

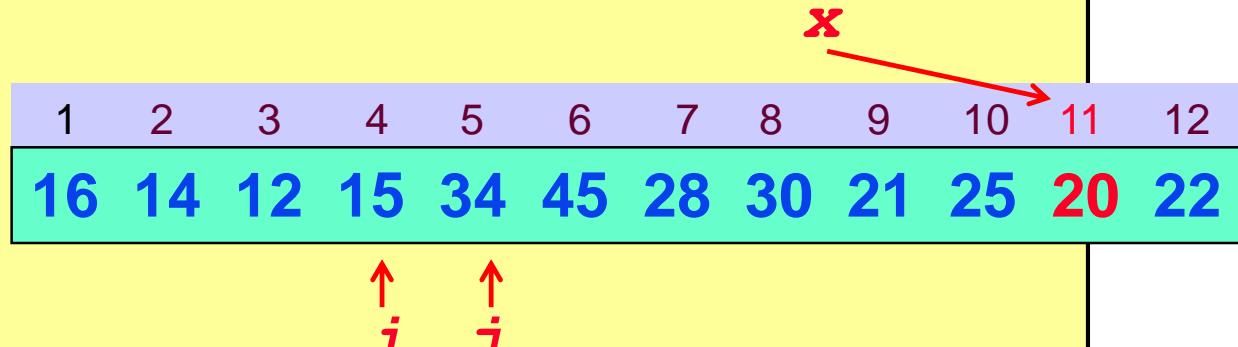
```
        UNTIL A[i] ≥ x
```

```
    IF i < j
```

```
        THEN "scambia A[i] con A[j]"
```

```
UNTIL i ≥ j
```

```
return j
```



Algoritmo Partiziona: partiziona (A, 1, 12)

```
int Partiziona(A, p, r)
```

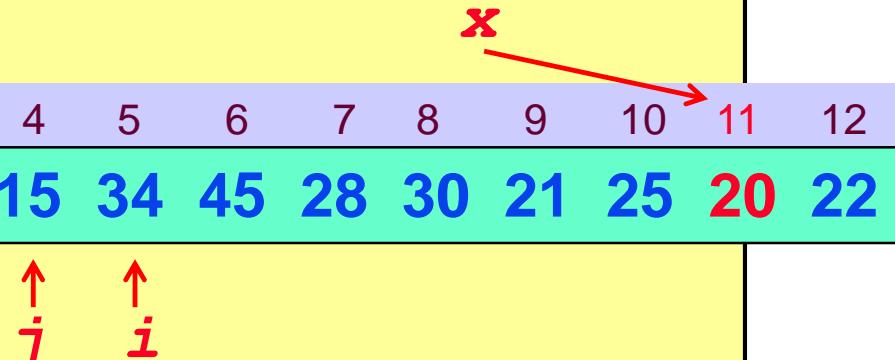
```
    x = A[p]
```

```
    i = p - 1
```

```
    j = r + 1
```

```
REPEAT
```

```
    REPEAT j = j - 1
```



```
        UNTIL A[j] ≤ x
```

```
    REPEAT i = i + 1
```

```
        UNTIL A[i] ≥ x
```

```
    IF i < j
```

```
        THEN "scambia A[i] con A[j]"
```

```
UNTIL i ≥ j
```

```
return j
```

Algoritmo Partiziona: partiziona (A, 1, 12)

```
int Partiziona(A, p, r)
```

```
    x = A[p]
```

```
    i = p - 1
```

```
    j = r + 1
```

```
REPEAT
```

```
    REPEAT j = j - 1
```

```
        UNTIL A[j] ≤ x
```

```
    REPEAT i = i + 1
```

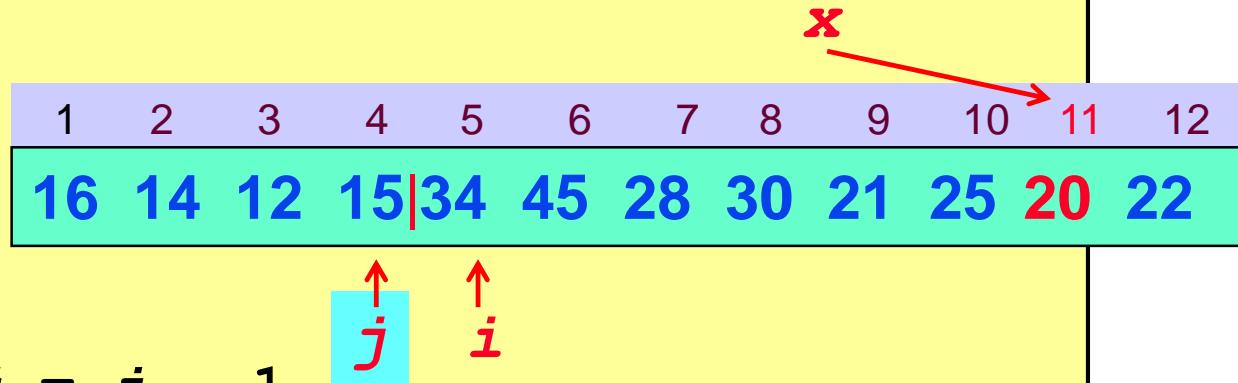
```
        UNTIL A[i] ≥ x
```

```
    IF i < j
```

```
        THEN "scambia A[i] con A[j]"
```

```
    UNTIL i ≥ j
```

```
return j
```



Algoritmo Partiziona: casi estremi

```
int Partiziona(A, p, r)
```

```
    x = A[p]
```

```
    i = p - 1
```

```
    j = r + 1
```

```
REPEAT
```

```
    REPEAT j = j - 1
```

```
        UNTIL A[j] ≤ x
```

```
    REPEAT i = i + 1
```

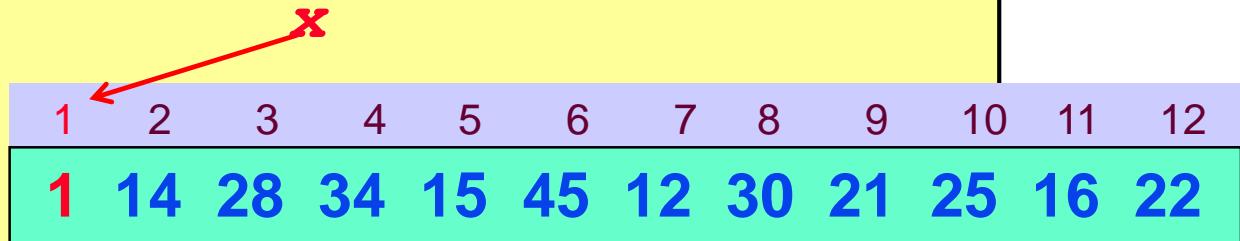
```
        UNTIL A[i] ≥ x
```

```
    IF i < j
```

```
        THEN "scambia A[i] con A[j]"
```

```
    UNTIL i ≥ j
```

```
return j
```



Se esiste un solo elemento minore o uguale al pivot, ...

Algoritmo Partiziona: casi estremi

```
int Partiziona(A, p, r)
```

```
    x = A[p]
```

```
    i = p - 1
```

```
    j = r + 1
```

```
REPEAT
```

```
    REPEAT j =  $j - 1$ 
```

```
        UNTIL A[j]  $\leq x$ 
```

```
    REPEAT i = i + 1
```

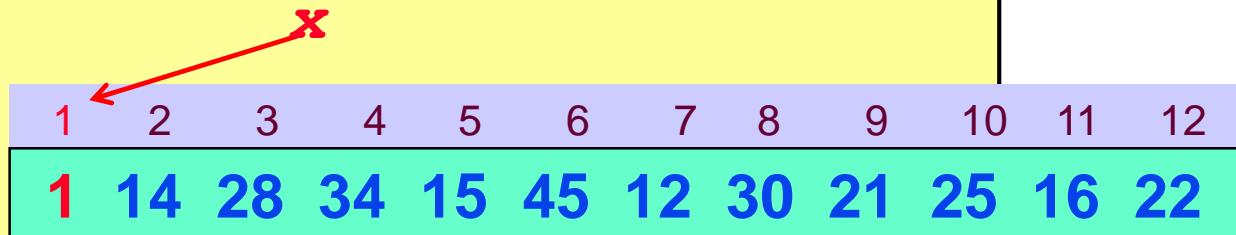
```
        UNTIL A[i]  $\geq x$ 
```

```
    IF i < j
```

```
        THEN "scambia A[i] con A[j]"
```

```
UNTIL i  $\geq j$ 
```

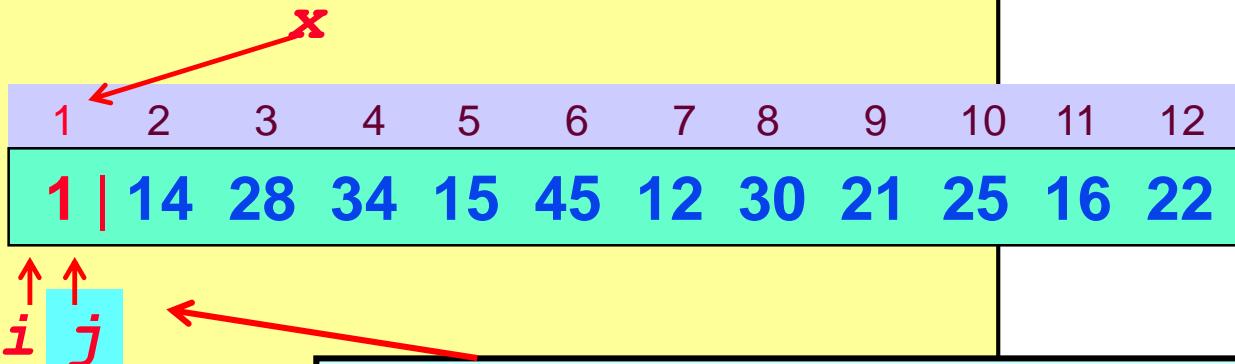
```
return j
```



Se esiste un solo elemento minore o uguale al pivot, ...

Algoritmo Partiziona: casi estremi

```
int Partiziona(A, p, r)
    x = A[p]
    i = p - 1
    j = r + 1
    REPEAT
        REPEAT j = j - 1
            UNTIL A[j] ≤ x
        REPEAT i = i + 1
            UNTIL A[i] ≥ x
        IF i < j
            THEN "scambia A"
        UNTIL i ≥ j
    return j
```



Se esiste un solo elemento minore o uguale al pivot, l'array è partizionato in due porzioni: quella sinistra ha dimensione 1 e quella destra ha dimensione n-1

Algoritmo Partiziona: casi estremi

```
int Partiziona(A, p, r)
```

```
    x = A[p]
```

```
    i = p - 1
```

```
    j = r + 1
```

```
REPEAT
```

```
    REPEAT j = j - 1
```

```
        UNTIL A[j] ≤ x
```

```
    REPEAT i = i + 1
```

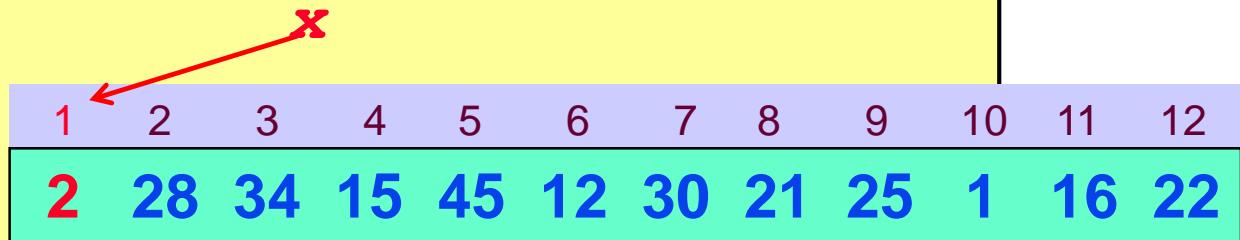
```
        UNTIL A[i] ≥ x
```

```
    IF i < j
```

```
        THEN "scambia A[i] con A[j]"
```

```
    UNTIL i ≥ j
```

```
return j
```



Se esistono solo due elementi minori o uguali al pivot, ...

Algoritmo Partiziona: casi estremi

```
int Partiziona(A, p, r)
```

```
    x = A[p]
```

```
    i = p - 1
```

```
    j = r + 1
```

```
    REPEAT
```

```
        REPEAT j = j - 1
```

```
            UNTIL A[j] ≤ x
```

```
        REPEAT i = i + 1
```

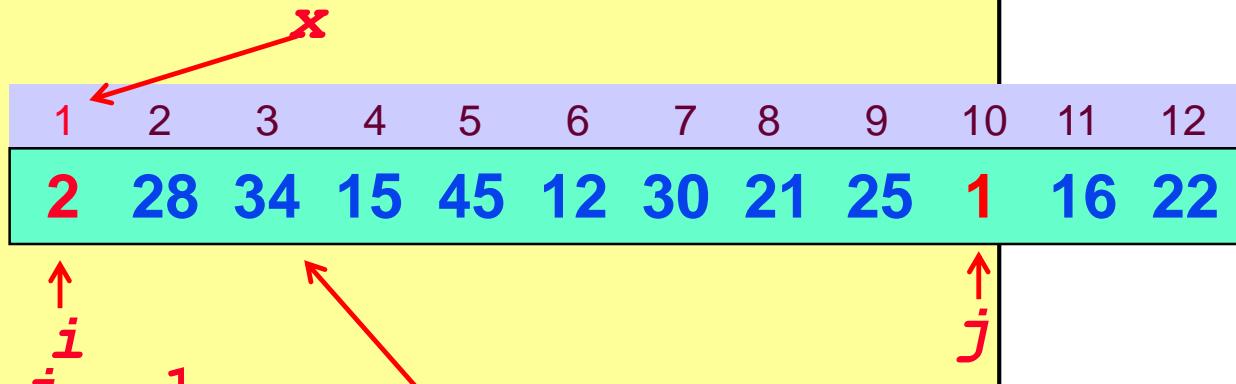
```
            UNTIL A[i] ≥ x
```

```
        IF i < j
```

```
            THEN "scambia A[i] con A[j]"
```

```
        UNTIL i ≥ j
```

```
    return j
```



Se esistono solo due elementi minori o uguali al pivot, ...

Algoritmo Partiziona: casi estremi

```
int Partiziona(A, p, r)
```

```
    x = A[p]
```

```
    i = p - 1
```

```
    j = r + 1
```

```
    REPEAT
```

```
        REPEAT j = j - 1
```

```
            UNTIL A[j] ≤ x
```

```
        REPEAT i = i + 1
```

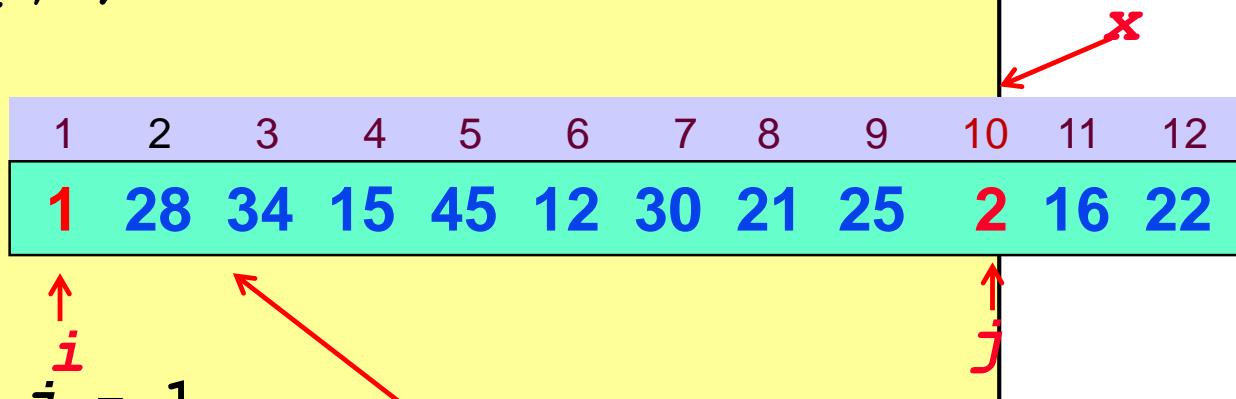
```
            UNTIL A[i] ≥ x
```

```
        IF i < j
```

```
            THEN "scambia A[i] con A[j]"
```

```
    UNTIL i ≥ j
```

```
return j
```



Se esistono solo due elementi minori o uguali al pivot, ...

Algoritmo Partiziona: casi estremi

```
int Partiziona(A, p, r)
```

```
    x = A[p]
```

```
    i = p - 1
```

```
    j = r + 1
```

```
    REPEAT
```

```
        REPEAT j = j - 1
```

```
            UNTIL A[j] ≤ x
```

```
        REPEAT i = i + 1
```

```
            UNTIL A[i] ≥ x
```

```
        IF i < j
```

```
            THEN "scambia A[i] con A[j]"
```

```
        UNTIL i ≥ j
```

```
    return j
```

1	2	3	4	5	6	7	8	9	10	11	12
1	28	34	15	45	12	30	21	25	2	16	22

\uparrow \uparrow
 j i

Se esistono solo due elementi minori o uguali al pivot, ...

Algoritmo Partiziona: casi estremi

```
int Partiziona(A, p, r)
    x = A[p]
    i = p - 1
    j = r + 1
    REPEAT
        REPEAT j = j - 1
            UNTIL A[j] ≤ x
        REPEAT i = i + 1
            UNTIL A[i] ≥ x
        IF i < j
            THEN "scambia A[i] e A[j]"
        UNTIL i ≥ j
    return j
```

1	2	3	4	5	6	7	8	9	10	11	12
1 28	34	15	45	12	30	21	25	2 16	22		

j

i

Se esistono solo due elementi minori o uguali al pivot, l'array è partizionato in due porzioni: quella sinistra ha dimensione 1 e quella destra ha dimensione n-1

Algoritmo Partiziona: casi estremi

Partiziona è quindi tale che:

SE il numero di elementi dell'array minori o uguali all'elemento $A[p]$, scelto come *pivot*, è pari a 1 (cioè $A[p]$ è l'elemento minimo) o a 2,

ALLORA le *dimensioni* delle partizioni restituite sono 1 per la *partizione di sinistra* e $n-1$ per *quella di destra*.

Algoritmo QuickSort

```
Quick-Sort( $A, p, r$ )
```

```
IF  $p < r$ 
```

```
THEN  $q = \text{Partiziona}(A, p, r)$ 
```

```
    Quick-Sort( $A, p, q$ )
```

```
    Quick-Sort( $A, q + 1, r$ )
```

1	2	3	4	5	6	7	8	9	10	11	12
20	14	28	34	15	45	12	30	21	25	16	22

\uparrow
 P

\uparrow
 r

Algoritmo QuickSort

```
Quick-Sort( $A, p, r$ )
```

```
IF  $p < r$ 
```

```
THEN  $q = \text{Partiziona}(A, p, r)$ 
```

```
    Quick-Sort( $A, p, q$ )
```

```
    Quick-Sort( $A, q + 1, r$ )
```

1	2	3	4	5	6	7	8	9	10	11	12
---	---	---	---	---	---	---	---	---	----	----	----

20	14	28	34	15	45	12	30	21	25	16	22
----	----	----	----	----	----	----	----	----	----	----	----

\uparrow
 P

\uparrow
 r

Algoritmo QuickSort

```
Quick-Sort( $A, p, r$ )
```

```
IF  $p < r$ 
```

```
THEN  $q = \text{Partiziona}(A, p, r)$ 
```

```
    Quick-Sort( $A, p, q$ )
```

```
    Quick-Sort( $A, q + 1, r$ )
```

1	2	3	4	5	6	7	8	9	10	11	12
---	---	---	---	---	---	---	---	---	----	----	----

16	14	12	15		34	45	28	30	21	25	20	22
----	----	----	----	--	----	----	----	----	----	----	----	----

\uparrow
 p

\uparrow
 q

\uparrow
 r

Algoritmo QuickSort

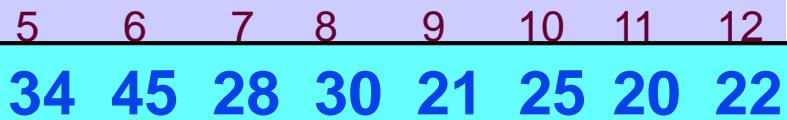
```
Quick-Sort( $A, p, r$ )
```

IF $p < r$

THEN $q = \text{Partiziona}(A, p, r)$

 Quick-Sort(A, p, q)

 Quick-Sort($A, q + 1, r$)



\uparrow
 p

\uparrow
 q

\uparrow
 r

Algoritmo QuickSort

Quick-Sort(A, p, r)

IF $p < r$

THEN $q = \text{Partiziona}(A, p, r)$

 Quick-Sort(A, p, q)

 Quick-Sort($A, q + 1, r$)



\uparrow
 p

\uparrow
 r

Algoritmo QuickSort

```
Quick-Sort( $A, p, r$ )
```

IF $p < r$

THEN $q = \text{Partiziona}(A, p, r)$

 Quick-Sort(A, p, q)

 Quick-Sort($A, q + 1, r$)



11



\uparrow
 p

\uparrow
 r

Algoritmo QuickSort

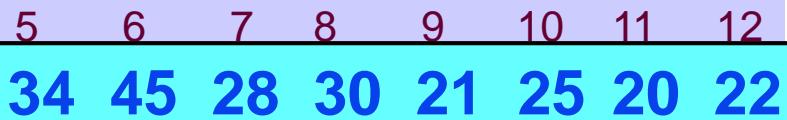
```
Quick-Sort( $A, p, r$ )
```

IF $p < r$

THEN $q = \text{Partiziona}(A, p, r)$

 Quick-Sort(A, p, q)

 Quick-Sort($A, q + 1, r$)



\uparrow
 P

\uparrow
 q

\uparrow
 r

Algoritmo QuickSort

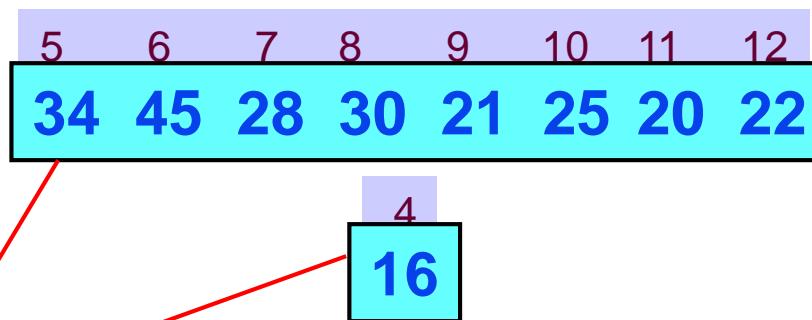
```
Quick-Sort( $A, p, r$ )
```

```
IF  $p < r$ 
```

```
THEN  $q = \text{Partiziona}(A, p, r)$ 
```

```
    Quick-Sort( $A, p, q$ )
```

```
    Quick-Sort( $A, q + 1, r$ )
```



```
1 2 3 4 5 6 7 8 9 10 11 12
```

```
15 14 12 | 16 | 34 45 28 30 21 25 20 22
```

\uparrow
 P

\uparrow
 q

\uparrow
 r

Algoritmo QuickSort

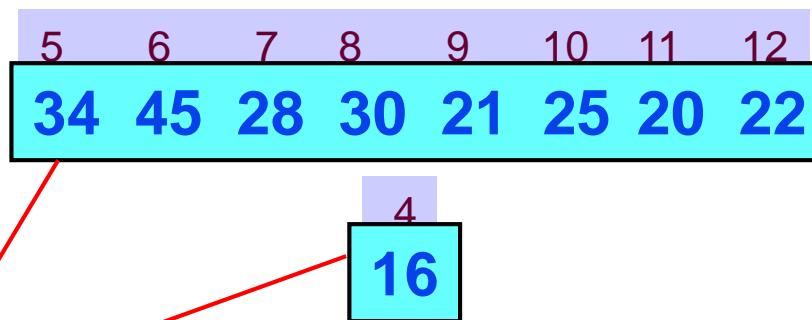
Quick-Sort(A, p, r)

IF $p < r$

THEN $q = \text{Partiziona}(A, p, r)$

Quick-Sort(A, p, q)

Quick-Sort($A, q + 1, r$)



1 2 3 4 5 6 7 8 9 10 11 12

15 14 12 | 16 | 34 45 28 30 21 25 20 22

↑ ↑
 p r

Algoritmo QuickSort

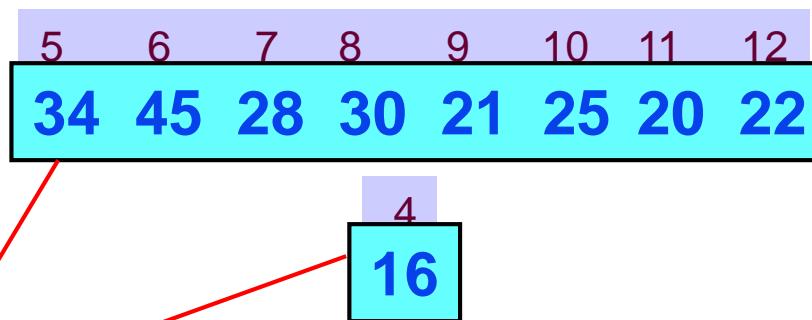
```
Quick-Sort( $A, p, r$ )
```

IF $p < r$

THEN $q = \text{Partiziona}(A, p, r)$

Quick-Sort(A, p, q)

Quick-Sort($A, q + 1, r$)



1 2 3 4 5 6 7 8 9 10 11 12

12 14 | 15| 16| 34 45 28 30 21 25 20 22

$\uparrow \uparrow \uparrow$
 $p \quad q \quad r$

Algoritmo QuickSort

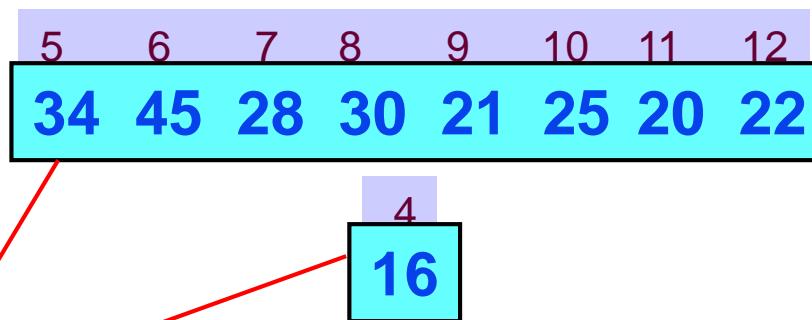
```
Quick-Sort( $A, p, r$ )
```

```
IF  $p < r$ 
```

```
THEN  $q = \text{Partiziona}(A, p, r)$ 
```

```
    Quick-Sort( $A, p, q$ )
```

```
    Quick-Sort( $A, q + 1, r$ )
```



```
1    2    3    4    5    6    7    8    9    10    11    12
```

```
12 14 | 15 | 16 | 34 45 28 30 21 25 20 22
```

$\uparrow \uparrow \uparrow$
 $p \quad q \quad r$

Algoritmo QuickSort

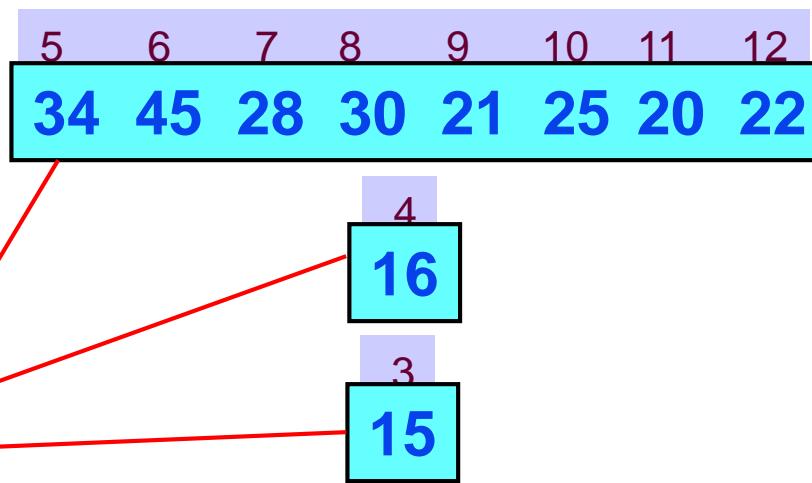
```
Quick-Sort(A, p, r)
```

```
IF p < r
```

```
THEN q = Partiziona(A, p, r)
```

```
Quick-Sort(A, p, q)
```

```
Quick-Sort(A, q + 1, r)
```



```
1 2 3 4 5 6 7 8 9 10 11 12
```

```
12 14 | 15 | 16 | 34 45 28 30 21 25 20 22
```

↑
P
↑
r

Algoritmo QuickSort

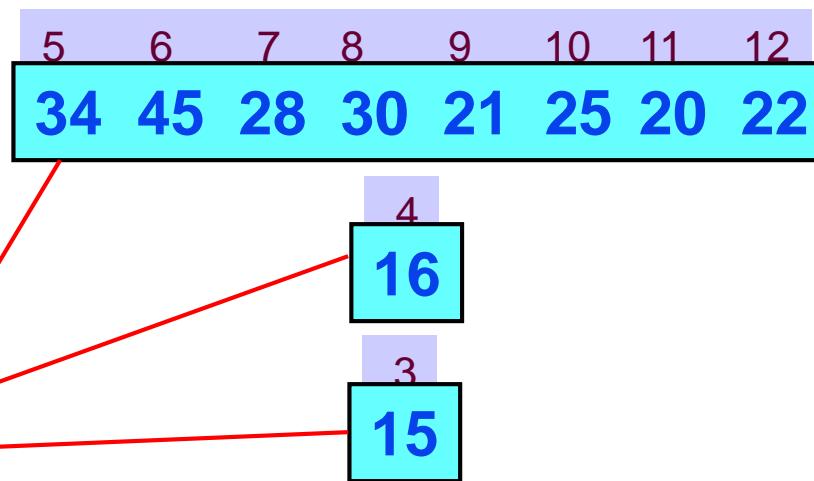
```
Quick-Sort( $A, p, r$ )
```

```
IF  $p < r$ 
```

```
THEN  $q = \text{Partiziona}(A, p, r)$ 
```

```
Quick-Sort( $A, p, q$ )
```

```
Quick-Sort( $A, q + 1, r$ )
```



```
1 2 3 4 5 6 7 8 9 10 11 12
```

```
12 14 | 15 | 16 | 34 45 28 30 21 25 20 22
```

$\uparrow \uparrow$
 $p \quad r$

Algoritmo QuickSort

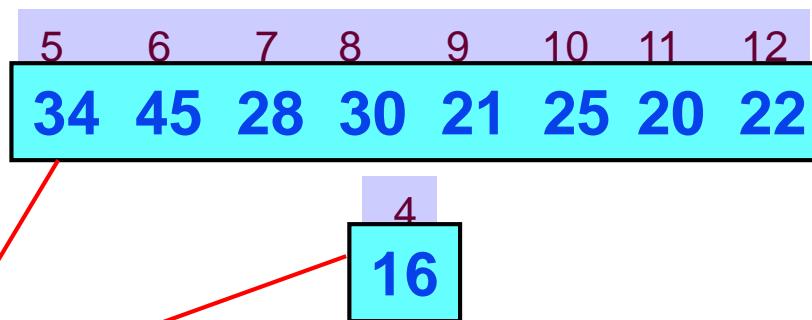
```
Quick-Sort( $A, p, r$ )
```

```
IF  $p < r$ 
```

```
THEN  $q = \text{Partiziona}(A, p, r)$ 
```

```
    Quick-Sort( $A, p, q$ )
```

```
    Quick-Sort( $A, q+1, r$ )
```



```
1 2 3 4 5 6 7 8 9 10 11 12
```

```
12 14 15 | 16 | 34 45 28 30 21 25 20 22
```

↑ ↑ ↑
 p q r

Algoritmo QuickSort

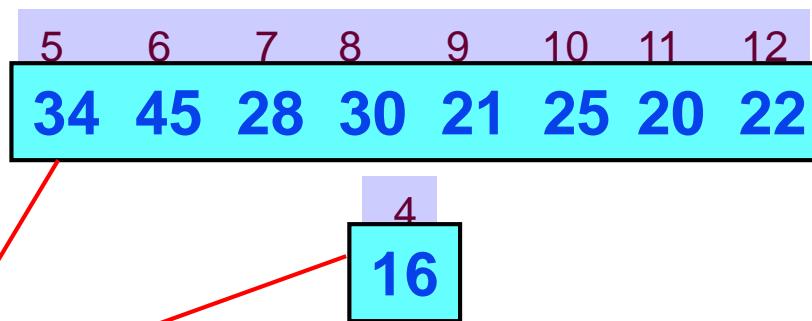
Quick-Sort(A, p, r)

IF $p < r$

THEN $q = \text{Partiziona}(A, p, r)$

Quick-Sort(A, p, q)

Quick-Sort($A, q + 1, r$)



1 2 3 4 5 6 7 8 9 10 11 12

12 14 15 | 16 | 34 45 28 30 21 25 20 22

↑↑
pr

Algoritmo QuickSort

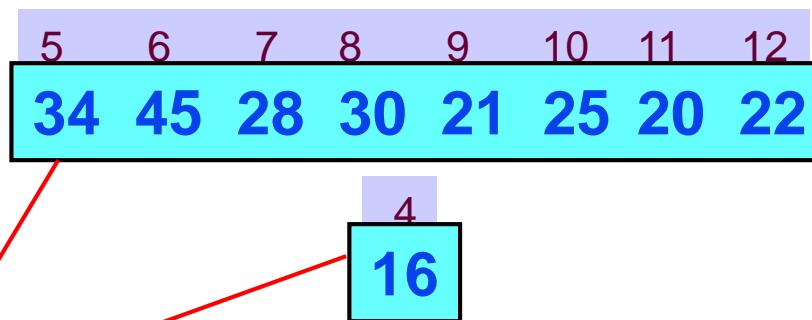
```
Quick-Sort( $A, p, r$ )
```

IF $p < r$

THEN $q = \text{Partiziona}(A, p, r)$

 Quick-Sort(A, p, q)

 Quick-Sort($A, q + 1, r$)



1 2 3 4 5 6 7 8 9 10 11 12

12 14 15 | 16 | 34 45 28 30 21 25 20 22

↑↑
 pr

Algoritmo QuickSort

```
Quick-Sort( $A, p, r$ )
```

IF $p < r$

THEN $q = \text{Partiziona}(A, p, r)$

 Quick-Sort(A, p, q)

 Quick-Sort($A, q+1, r$)



1 2 3 4 5 6 7 8 9 10 11 12

12 14 15 16 | 34 45 28 30 21 25 20 22

↑↑
 pr

Algoritmo QuickSort

```
Quick-Sort( $A, p, r$ )
```

```
IF  $p < r$ 
```

```
THEN  $q = \text{Partiziona}(A, p, r)$ 
```

```
    Quick-Sort( $A, p, q$ )
```

```
    Quick-Sort( $A, q+1, r$ )
```

1	2	3	4	5	6	7	8	9	10	11	12
---	---	---	---	---	---	---	---	---	----	----	----

12	14	15	16		34	45	28	30	21	25	20	22
----	----	----	----	--	----	----	----	----	----	----	----	----

\uparrow
 p

\uparrow
 q

\uparrow
 r

Algoritmo QuickSort

Quick-Sort(A, p, r)

IF $p < r$

THEN $q = \text{Partiziona}(A, p, r)$

 Quick-Sort(A, p, q)

 Quick-Sort($A, q + 1, r$)

1 2 3 4 5 6 7 8 9 10 11 12

12 14 15 16 | 34 45 28 30 21 25 20 22

↑
 p

↑
 r

Algoritmo QuickSort

```
Quick-Sort( $A, p, r$ )
```

```
IF  $p < r$ 
```

```
THEN  $q = \text{Partiziona}(A, p, r)$ 
```

```
    Quick-Sort( $A, p, q$ )
```

```
    Quick-Sort( $A, q + 1, r$ )
```

1	2	3	4	5	6	7	8	9	10	11	12
---	---	---	---	---	---	---	---	---	----	----	----

12	14	15	16		22	20	28	30	21	25		45	34
----	----	----	----	--	----	----	----	----	----	----	--	----	----

\uparrow
 p

\uparrow
 q

\uparrow
 r

Algoritmo QuickSort

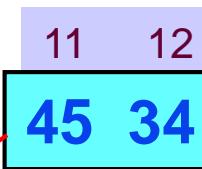
```
Quick-Sort( $A, p, r$ )
```

IF $p < r$

THEN $q = \text{Partiziona}(A, p, r)$

Quick-Sort(A, p, q)

Quick-Sort($A, q + 1, r$)



1 2 3 4 5 6 7 8 9 10 11 12

12 14 15 16 | 22 20 28 30 21 25 | 45 34

↑
 p

↑
 q

↑
 r

Algoritmo QuickSort

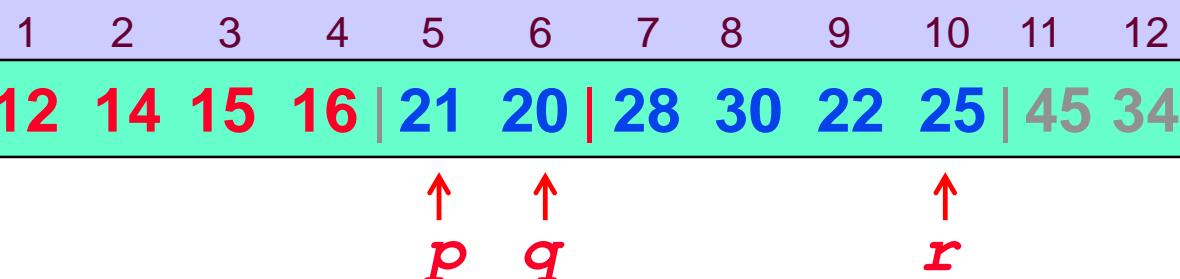
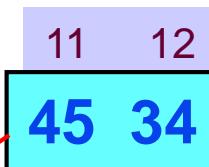
Quick-Sort(A, p, r)

IF $p < r$

THEN $q = \text{Partiziona}(A, p, r)$

Quick-Sort (A, p, q)

Quick-Sort($A, q+1, r$)



Algoritmo QuickSort

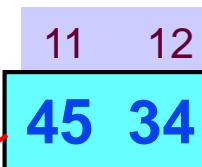
```
Quick-Sort( $A, p, r$ )
```

```
IF  $p < r$ 
```

```
THEN  $q = \text{Partiziona}(A, p, r)$ 
```

```
    Quick-Sort( $A, p, q$ )
```

```
    Quick-Sort( $A, q + 1, r$ )
```



1	2	3	4	5	6	7	8	9	10	11	12
---	---	---	---	---	---	---	---	---	----	----	----

12	14	15	16		21	20		28	30	22	25		45	34
----	----	----	----	--	----	----	--	----	----	----	----	--	----	----

\uparrow \uparrow \uparrow
 p q r

Algoritmo QuickSort

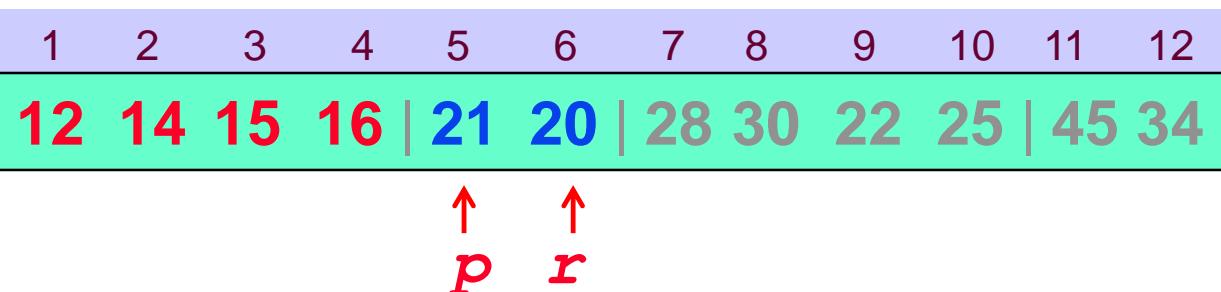
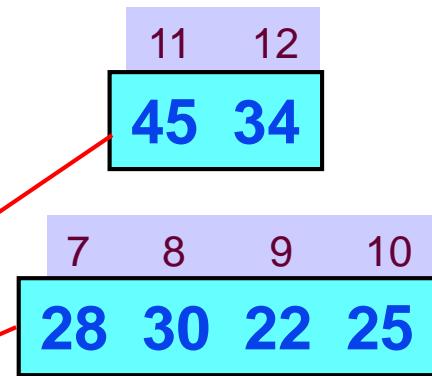
Quick-Sort(A, p, r)

IF $p < r$

THEN $q = \text{Partiziona}(A, p, r)$

Quick-Sort(A, p, q)

Quick-Sort($A, q + 1, r$)



Algoritmo QuickSort

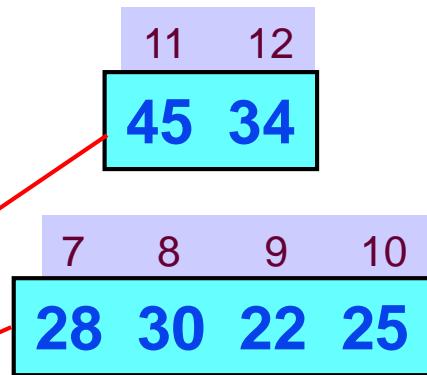
```
Quick-Sort(A, p, r)
```

```
IF  $p < r$ 
```

```
THEN  $q = \text{Partiziona}(A, p, r)$ 
```

```
    Quick-Sort(A, p, q)
```

```
    Quick-Sort(A, q + 1, r)
```



1 2 3 4 5 6 7 8 9 10 11 12

12 14 15 16 | 20 | 21 | 28 30 22 25 | 45 34

↑↑↑
 p q r

Algoritmo QuickSort

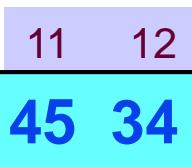
```
Quick-Sort( $A, p, r$ )
```

```
IF  $p < r$ 
```

```
THEN  $q = \text{Partiziona}(A, p, r)$ 
```

```
    Quick-Sort( $A, p, q$ )
```

```
    Quick-Sort( $A, q+1, r$ )
```



1 2 3 4 5 6 7 8 9 10 11 12

12 14 15 16 20 21 | 28 30 22 25 | 45 34

\uparrow
 p

\uparrow
 q

\uparrow
 r

Algoritmo QuickSort

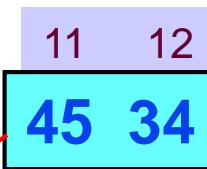
Quick-Sort(A, p, r)

IF $p < r$

THEN $q = \text{Partiziona}(A, p, r)$

 Quick-Sort(A, p, q)

 Quick-Sort($A, q + 1, r$)



1 2 3 4 5 6 7 8 9 10 11 12

12 14 15 16 20 21 | 28 30 22 25 | 45 34

↑
 p

↑
 r

Algoritmo QuickSort

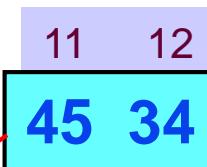
```
Quick-Sort( $A, p, r$ )
```

```
IF  $p < r$ 
```

```
THEN  $q = \text{Partiziona}(A, p, r)$ 
```

```
    Quick-Sort( $A, p, q$ )
```

```
    Quick-Sort( $A, q + 1, r$ )
```



1 2 3 4 5 6 7 8 9 10 11 12

12 14 15 16 20 21 | 25 22 | 30 28 | 45 34

↑ ↑ ↑
 p q r

Algoritmo QuickSort

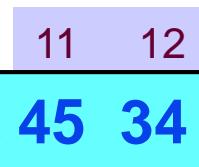
```
Quick-Sort( $A, p, r$ )
```

```
IF  $p < r$ 
```

```
THEN  $q = \text{Partiziona}(A, p, r)$ 
```

```
    Quick-Sort( $A, p, q$ )
```

```
    Quick-Sort( $A, q + 1, r$ )
```



1 2 3 4 5 6 7 8 9 10 11 12

12 14 15 16 20 21 | 22 25 | 28 30 | 45 34

↑ ↑ ↑
 p q r

Algoritmo QuickSort

Quick-Sort(A, p, r)

IF $p < r$

THEN $q = \text{Partiziona}(A, p, r)$

Quick-Sort(A, p, q)

Quick-Sort($A, q+1, r$)

1 2 3 4 5 6 7 8 9 10 11 12

12 14 15 16 20 21 22 25 28 30 | 45 34

↑
 p

↑
 q

↑
 r

Algoritmo QuickSort

Quick-Sort(A, p, r)

```
IF  $p < r$ 
  THEN  $q = \text{Partiziona}(A, p, r)$ 
        Quick-Sort( $A, p, q$ )
        Quick-Sort( $A, q + 1, r$ )
```

1 2 3 4 5 6 7 8 9 10 11 12

12 14 15 16 20 21 22 25 28 30 | 45 34

↑ ↑
 p r

Algoritmo QuickSort

Quick-Sort(A, p, r)

IF $p < r$

THEN $q = \text{Partiziona}(A, p, r)$

 Quick-Sort(A, p, q)

 Quick-Sort($A, q + 1, r$)

1 2 3 4 5 6 7 8 9 10 11 12

12 14 15 16 20 21 22 25 28 30 | 34 45

↑↑↑
 $pq\ r$

Algoritmo QuickSort

```
Quick-Sort( $A, p, r$ )
```

```
IF  $p < r$ 
```

```
THEN  $q = \text{Partiziona}(A, p, r)$ 
```

```
    Quick-Sort( $A, p, q$ )
```

```
    Quick-Sort( $A, q + 1, r$ )
```

1	2	3	4	5	6	7	8	9	10	11	12	
12	14	15	16	20	21	22	25	28	30		34	45

\uparrow
 p

\uparrow
 q

\uparrow
 r

Algoritmo QuickSort

```
Quick-Sort( $A, p, r$ )
```

```
IF  $p < r$ 
```

```
THEN  $q = \text{Partiziona}(A, p, r)$ 
```

```
    Quick-Sort( $A, p, q$ )
```

```
    Quick-Sort( $A, q + 1, r$ )
```

L'array A ora è ordinato!



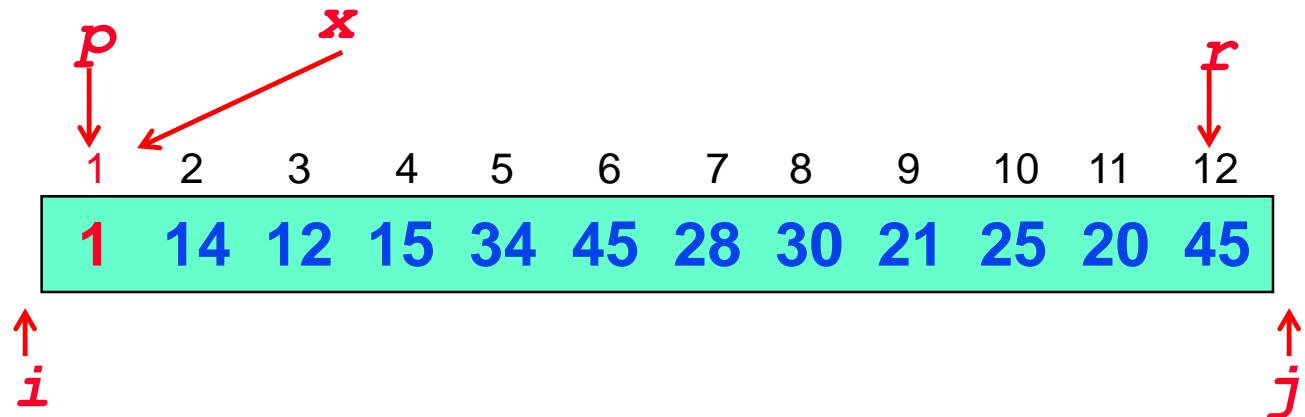
1	2	3	4	5	6	7	8	9	10	11	12
12	14	15	16	20	21	22	25	28	30	34	45

Algoritmo Partiziona: analisi

Gli indici i e j che scandiscono la sequenza non ne eccedono mai i limiti. Cioè vale sempre che
 $i \leq r$ e $j \geq p$

Algoritmo Partiziona: analisi

Gli indici i e j che scandiscono la sequenza non ne eccedono mai i limiti. Cioè vale sempre che $i \leq r$ e $j \geq p$

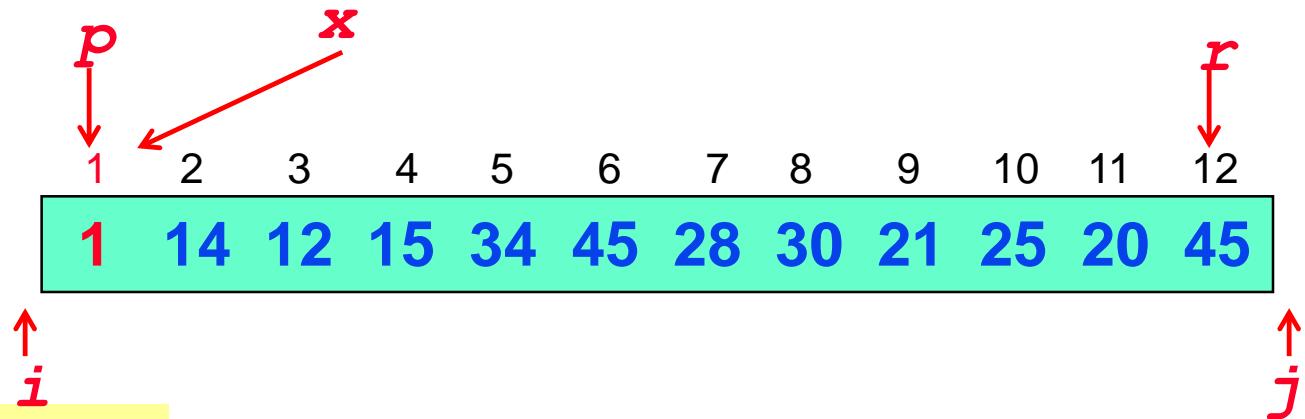


2 Casi. Partiziona effettua:

- *nessuno spostamento*
- *almeno uno spostamento*

Algoritmo Partiziona: analisi

Gli indici i e j che scandiscono la sequenza non ne eccedono mai i limiti. Cioè vale sempre che $i \leq r$ e $j \geq p$



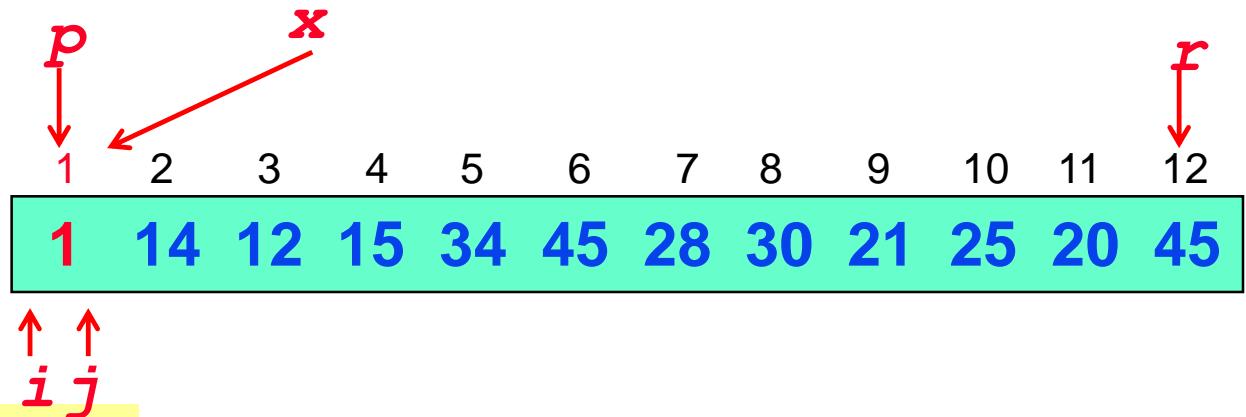
...

```
REPEAT  $j = j - 1$ 
      UNTIL  $A[j] \leq x$ 
REPEAT  $i = i + 1$ 
      UNTIL  $A[i] \geq x$ 
```

...

Algoritmo Partiziona: analisi

Gli indici i e j che scandiscono la sequenza non ne eccedono mai i limiti. Cioè vale sempre che $i \leq r$ e $j \geq p$



...

```
REPEAT  $j = j - 1$ 
      UNTIL  $A[j] \leq x$ 
REPEAT  $i = i + 1$ 
      UNTIL  $A[i] \geq x$ 
```

...

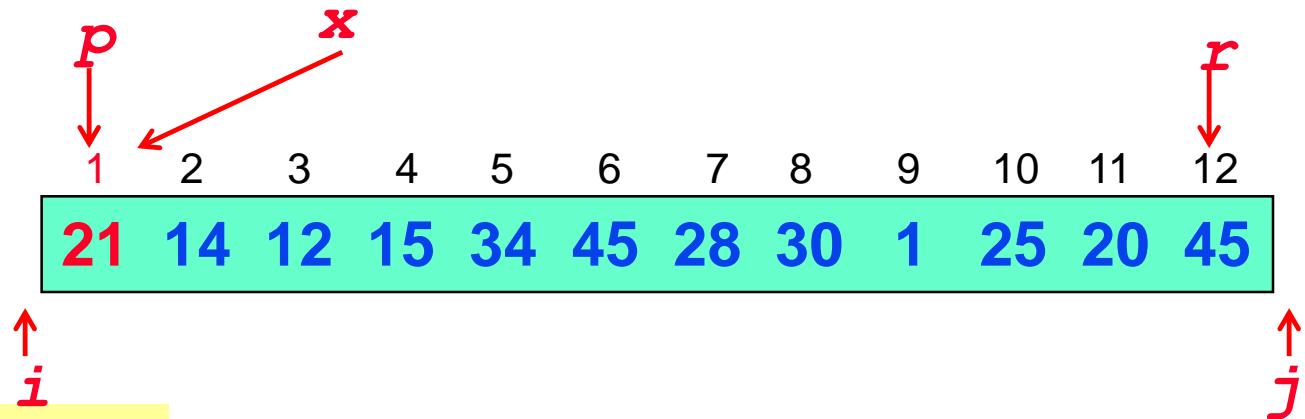
nessuno spostamento

$A[j] \leq x \quad \text{per } j \geq p$

$A[i] \geq x \quad \text{per } i \leq p$

Algoritmo Partiziona: analisi

Gli indici i e j che scandiscono la sequenza non ne eccedono mai i limiti. Cioè vale sempre che $i \leq r$ e $j \geq p$



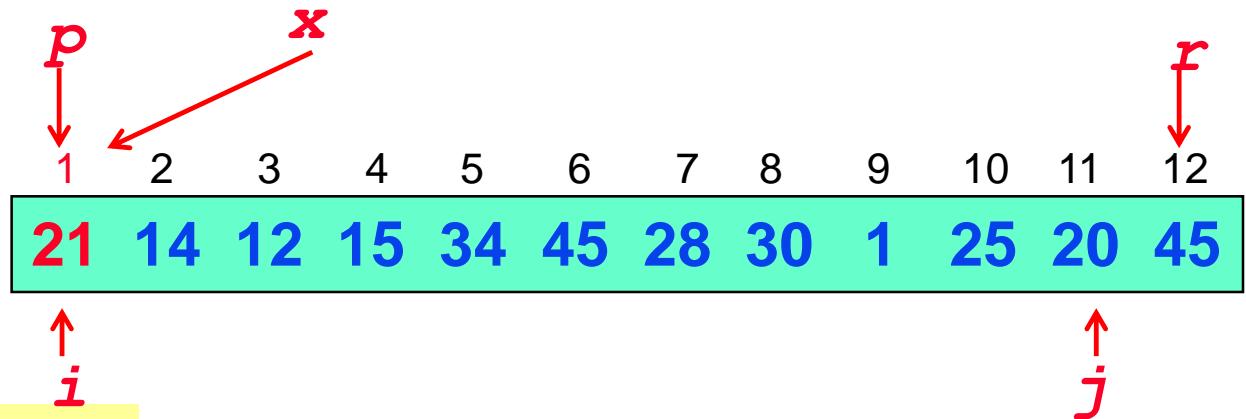
...

```
REPEAT  $j = j - 1$ 
    UNTIL  $A[j] \leq x$ 
REPEAT  $i = i + 1$ 
    UNTIL  $A[i] \geq x$ 
```

...

Algoritmo Partiziona: analisi

Gli indici i e j che scandiscono la sequenza non ne eccedono mai i limiti. Cioè vale sempre che $i \leq r$ e $j \geq p$



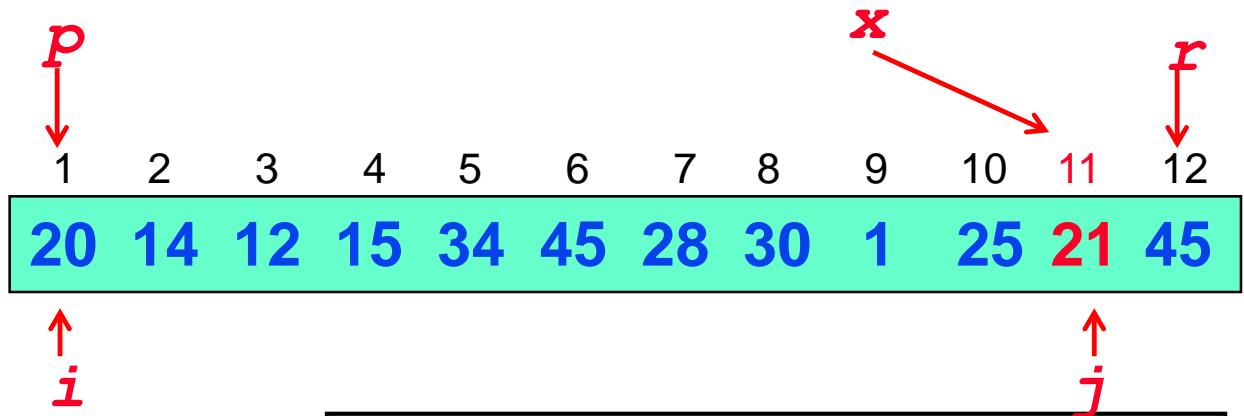
...

```
REPEAT  $j = j - 1$ 
      UNTIL  $A[j] \leq x$ 
REPEAT  $i = i + 1$ 
      UNTIL  $A[i] \geq x$ 
```

...

Algoritmo Partiziona: analisi

Gli indici i e j che scandiscono la sequenza non ne eccedono mai i limiti. Cioè vale sempre che $i \leq r$ e $j \geq p$

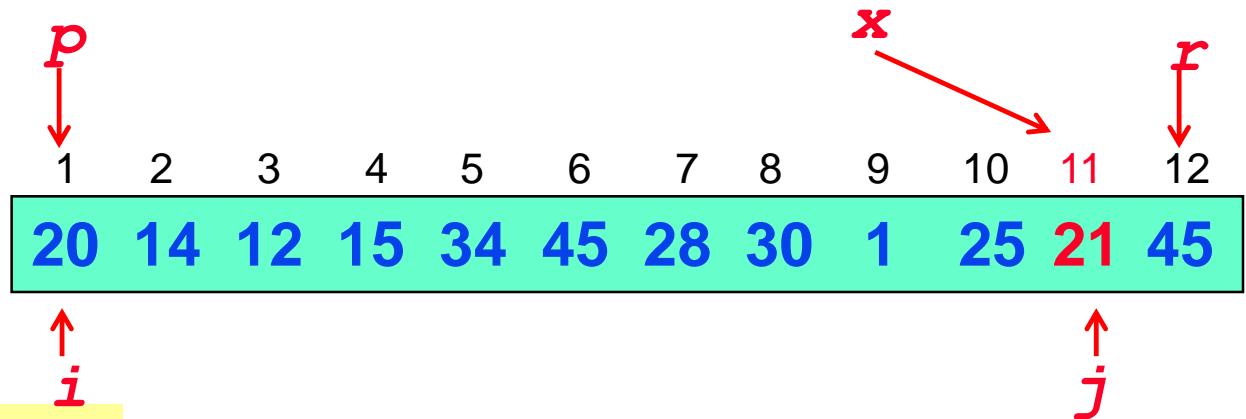


...
IF $i < j$
THEN "scambia
 $A[i]$ con $A[j]$ "
..."

dopo il primo spostamento,
esiste un k tale che
 $A[k] \leq x$ con $p \leq k \leq j$
esiste un z tale che
 $A[z] \geq x$ con $i \leq z \leq r$

Algoritmo Partiziona: analisi

Gli indici i e j che scandiscono la sequenza non ne eccedono mai i limiti. Cioè vale sempre che $i \leq r$ e $j \geq p$

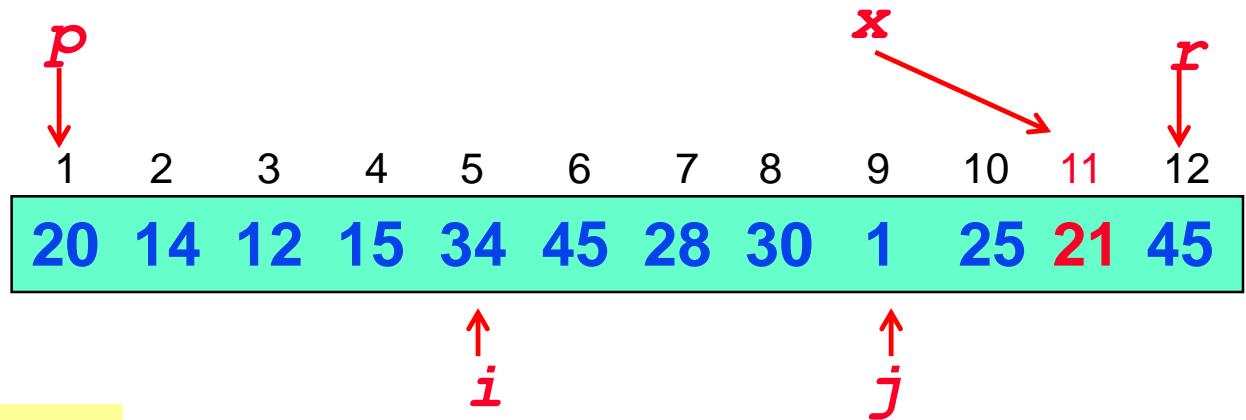


...
REPEAT $j = j - 1$
UNTIL $A[j] \leq x$
REPEAT $i = i + 1$
UNTIL $A[i] \geq x$

In generale, dopo ogni scambio:
- un elemento minore o uguale ad x viene spostato tra p e $j-1$
- un elemento maggiore o uguale ad x viene spostato tra $i+1$ e r

Algoritmo Partiziona: analisi

Gli indici i e j che scandiscono la sequenza non ne eccedono mai i limiti. Cioè vale sempre che $i \leq r$ e $j \geq p$

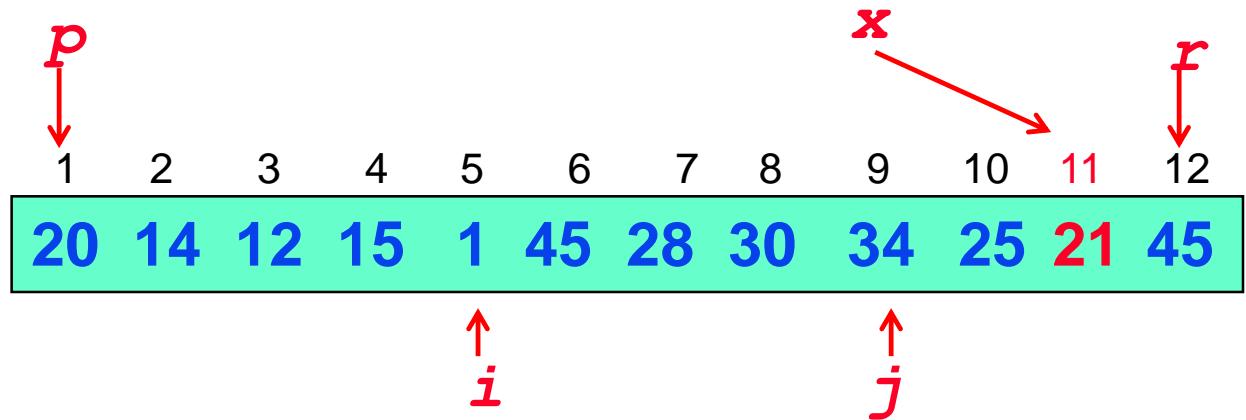


...
REPEAT $j = j - 1$
UNTIL $A[j] \leq x$
REPEAT $i = i + 1$
UNTIL $A[i] \geq x$

In generale, dopo ogni scambio:
- tra p e $j-1$ ci sarà sicuramente un elemento minore o uguale ad x
- tra $i+1$ e r ci sarà sicuramente un elemento maggiore o uguale ad x

Algoritmo Partiziona: analisi

Gli indici i e j che scandiscono la sequenza non ne eccedono mai i limiti. Cioè vale sempre che $i \leq r$ e $j \geq p$

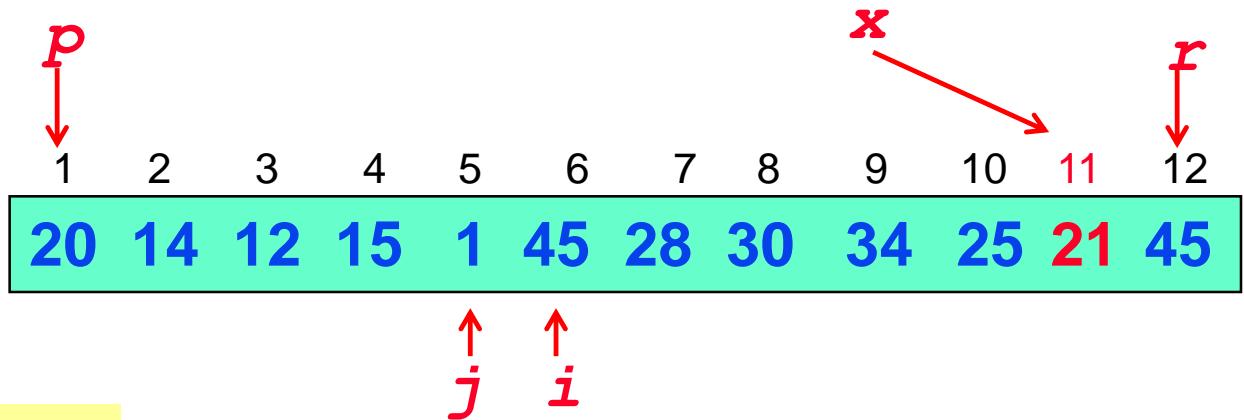


...
IF $i < j$
THEN "scambia
 $A[i]$ con $A[j]$ "
..."

In generale, dopo ogni scambio:
- tra p e $j-1$ ci sarà sicuramente un
elemento minore o uguale ad x
- tra $i+1$ e r ci sarà sicuramente un
elemento maggiore o uguale ad x

Algoritmo Partiziona: analisi

Gli indici i e j che scandiscono la sequenza non ne eccedono mai i limiti. Cioè vale sempre che $i \leq r$ e $j \geq p$



...
REPEAT $j = j - 1$
UNTIL $A[j] \leq x$
REPEAT $i = i + 1$
UNTIL $A[i] \geq x$
...

In generale, dopo ogni scambio:

- tra p e $j-1$ ci sarà sicuramente un elemento minore o uguale ad x
- tra $i+1$ e r ci sarà sicuramente un elemento maggiore o uguale ad x

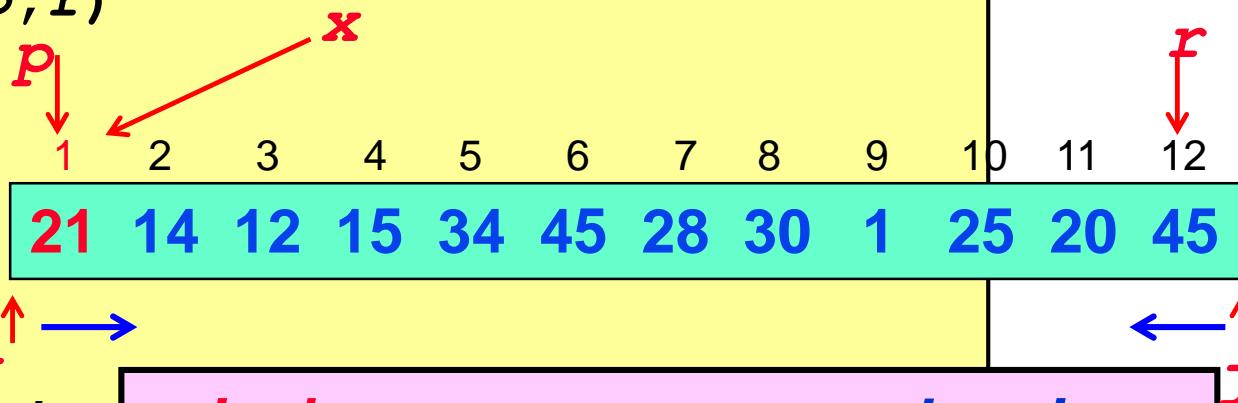
Algoritmo Partiziona: analisi

```
int Partiziona(A, p, r)
  x = A[p]
  i = p - 1
  j = r + 1
  REPEAT
    REPEAT j = j - 1
      UNTIL A[j] ≤ x
    REPEAT i = i + 1
      UNTIL A[i] ≥ x
    IF i < j
      THEN "scambia A[i] con A[j]"
  UNTIL i ≥ j
return j
```

$\Theta(1)$

Algoritmo Partiziona: analisi

```
int Partiziona(A, p, r)
    x = A[p]
    i = p - 1
    j = r + 1
    REPEAT
        REPEAT j = j -
            UNTIL A[j]
        REPEAT i = i +
            UNTIL A[i]
        IF i < j
            THEN "scam"
        UNTIL i ≥ j
    return j
```



- *i e j non possono eccedere i limiti dell'array,*
- *i e j sono sempre rispettivamente crescente e decrescente*
- *l'algoritmo termina quando $i \geq j$ quindi il costo del REPEAT sarà proporzionale ad n, cioè $\Theta(n)$*

Algoritmo Partiziona: analisi

```
int Partiziona(A, p, r)
  x = A[p]
  i = p - 1
  j = r + 1
  REPEAT
    REPEAT j = j - 1
      UNTIL A[j] ≤ x
    REPEAT i = i + 1
      UNTIL A[i] ≥ x
    IF i < j
      THEN "scambia A[i] con A[j]"
    UNTIL i ≥ j
  return j
```

$\Theta(1)$

$\Theta(n)$

Algoritmo Partiziona: analisi

```
int Partiziona(A,p,r)
x = A[p]
i = p - 1
j = r + 1
REPEAT
    REPEAT j = j - 1
        UNTIL A[j] ≤ x
    REPEAT i = i + 1
        UNTIL A[i] ≥ x
    IF i < j
        THEN "scambia A[i] con A[j]"
    UNTIL i ≥ j
return j
```

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

Analisi di QuickSort: intuizioni

Il **tempo di esecuzione** di QuickSort dipende dalla **bilanciamento** delle partizioni effettuate dall'algoritmo **partiziona**:

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

$$T(n) = T(q) + T(n-q) + \Theta(n) \quad \text{se } n > 1$$

- Il **caso migliore** si verifica quando le partizioni sono **perfettamente bilanciate**, entrambe di dimensione **$n/2$**
- Il **caso peggiore** si verifica quando una partizione è sempre di dimensione 1 (la seconda è quindi di dimensione **$n-1$**)

Analisi di QuickSort: caso migliore

```
Quick-Sort(A, p, r)
```

```
IF p < r
```

```
THEN q = Partiziona(A, p, r)
```

```
    Quick-Sort(A, p, q)
```

```
    Quick-Sort(A, q + 1, r)
```

Le partizioni sono di uguale dimensione:

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

e per il **caso 2** del **metodo principale**:

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

Analisi di QuickSort: caso migliore

```
Quick-Sort(A, p, r)
```

```
IF p < r
```

```
THEN q = Partiziona(A, p, r)
```

```
    Quick-Sort(A, p, q)
```

```
    Quick-Sort(A, q + 1, r)
```

*Quando si verifica il
caso migliore, ad
esempio?*

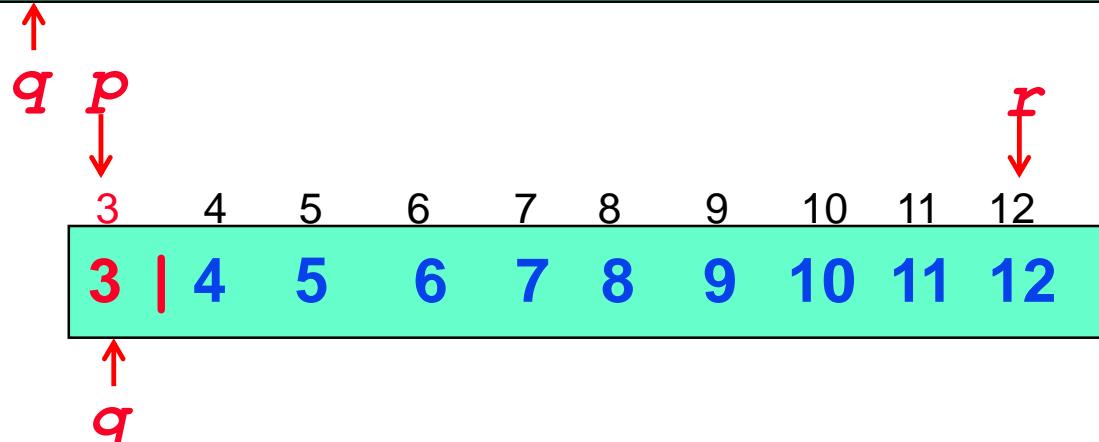
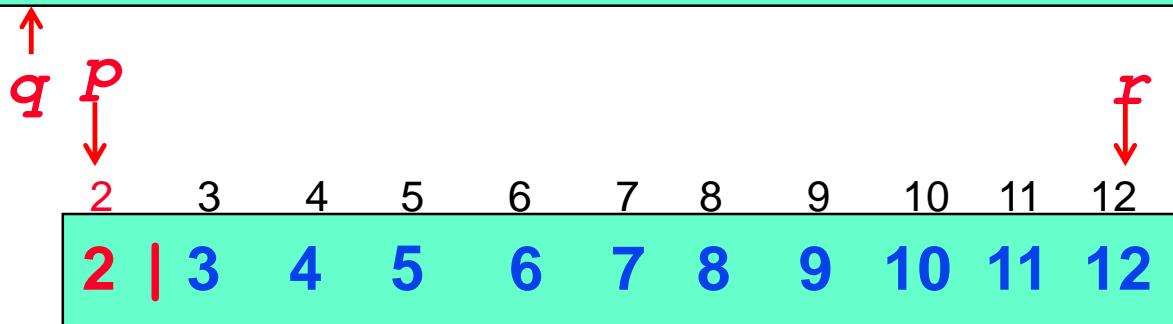
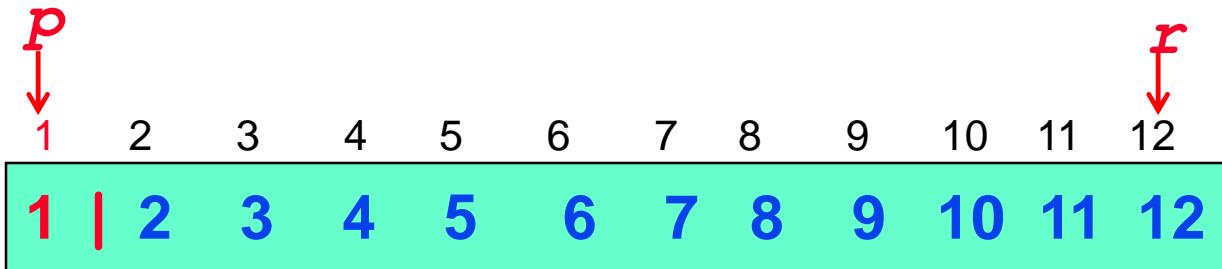
Le partizioni sono di uguale dimensione:

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

e per il **caso 2** del *metodo principale*:

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

Analisi di QuickSort: caso peggiore



Pivot = 1

Pivot = 2

Pivot = 3

Analisi di QuickSort: caso peggiore

```
Quick-Sort( $A, p, r$ )
```

```
IF  $p < r$ 
```

```
THEN  $q = \text{Partiziona}(A, p, r)$ 
```

```
    Quick-Sort( $A, p, q$ )
```

```
    Quick-Sort( $A, q + 1, r$ )
```

La partizione sinistra ha dimensione **1** mentre
quella destra ha dimensione **$n-1$** :

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

poiché **$T(1) = 1$** otteniamo

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

Analisi di QuickSort: caso peggiore

L'equazione di ricorrenza può essere risolta facilmente col *metodo iterativo*

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

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

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

$$= \Theta(n^2)$$

Analisi di QuickSort: caso peggiore

```
Quick-Sort( $A, p, r$ )
```

```
IF  $p < r$ 
```

```
THEN  $q = \text{Partiziona}(A, p, r)$ 
```

```
    Quick-Sort( $A, p, q$ )
```

```
    Quick-Sort( $A, q+1, r$ )
```

*Quando si verifica il
caso peggiore, ad
esempio?*

La partizione sinistra ha dimensione **1** mentre
quella destra ha dimensione **$n-1$** :

$$\begin{aligned} T(n) &= T(n-1) + \Theta(n) = \\ &= \Theta(n^2) \end{aligned}$$

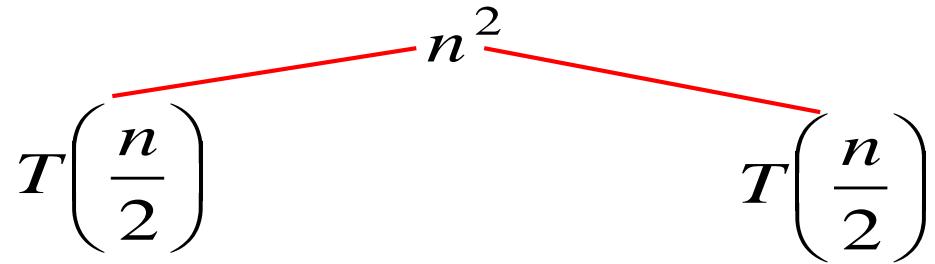
Alberi di Ricorrenza

Gli *alberi di ricorrenza* rappresentano un modo conveniente per visualizzare i passi di sostituzione necessari per risolvere una ricorrenza col *Metodo Iterativo*.

- Utili per semplificare i calcoli ed evidenziare le *condizioni limite* della ricorrenza.

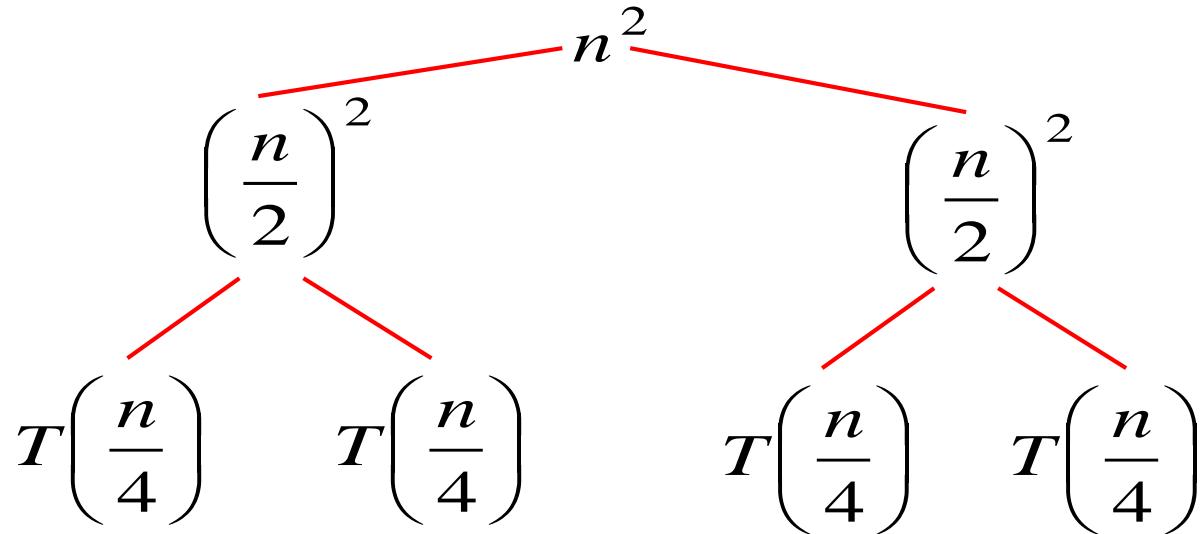
Alberi di Ricorrenza

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



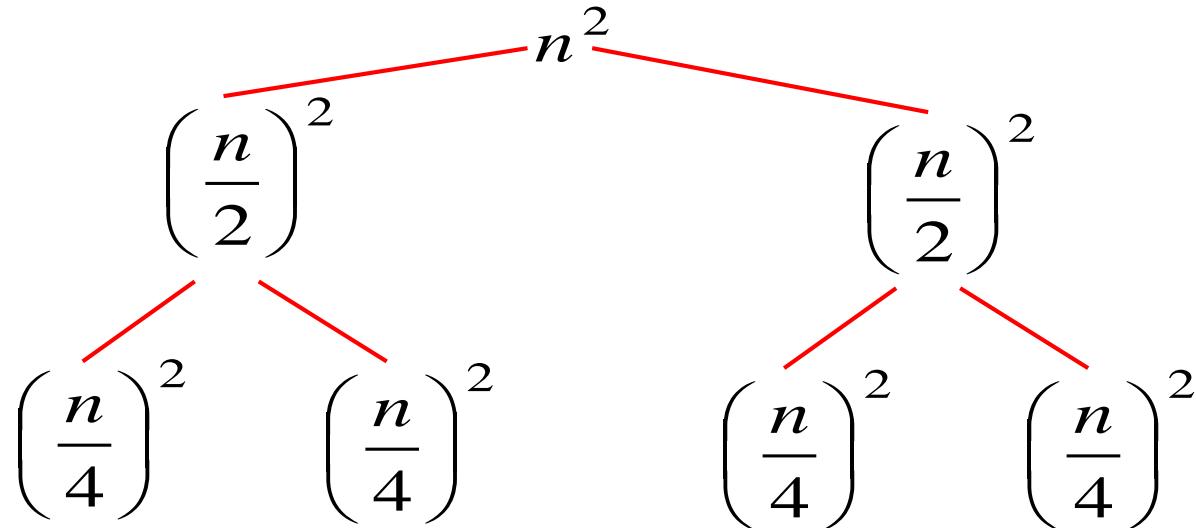
Alberi di Ricorrenza

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



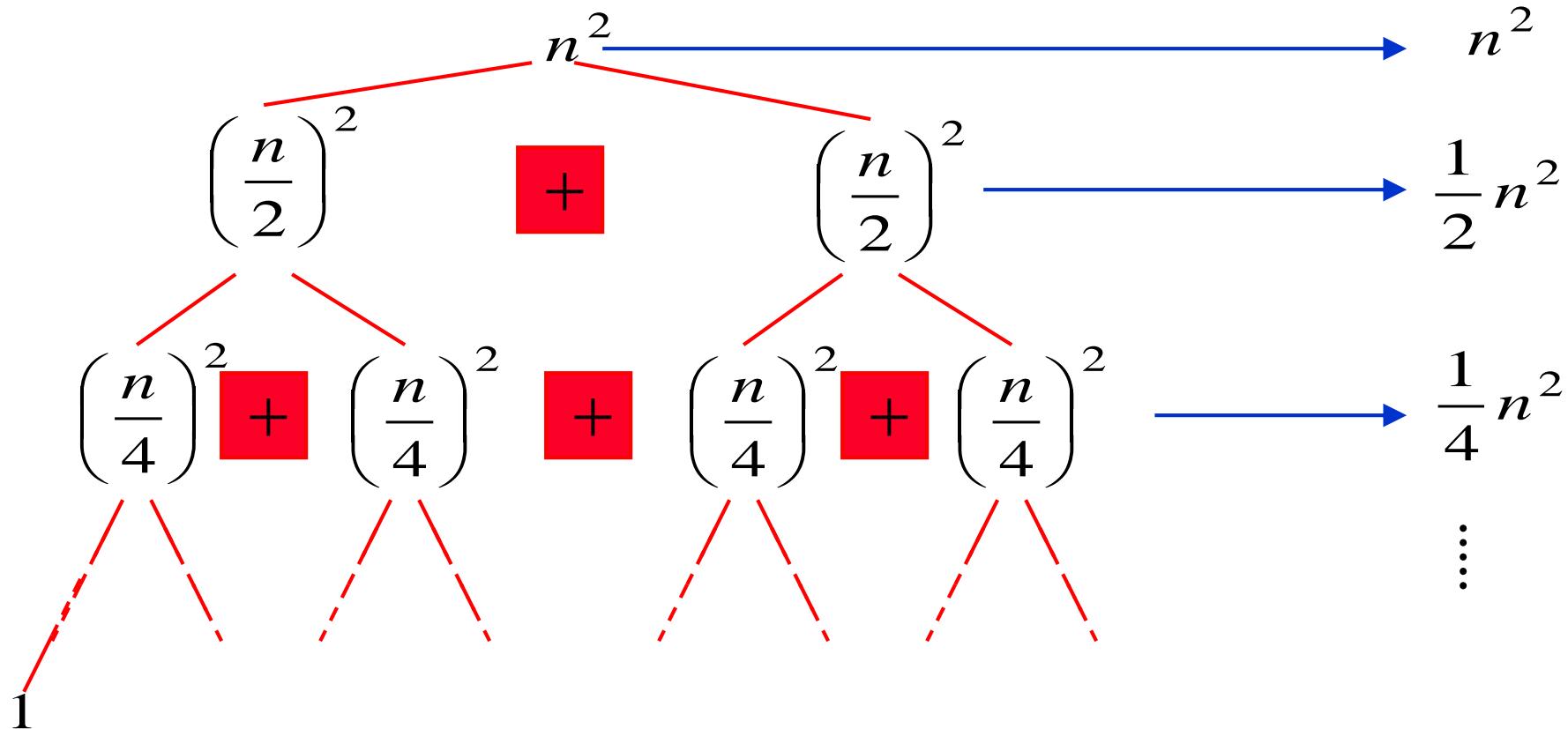
Alberi di Ricorrenza

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



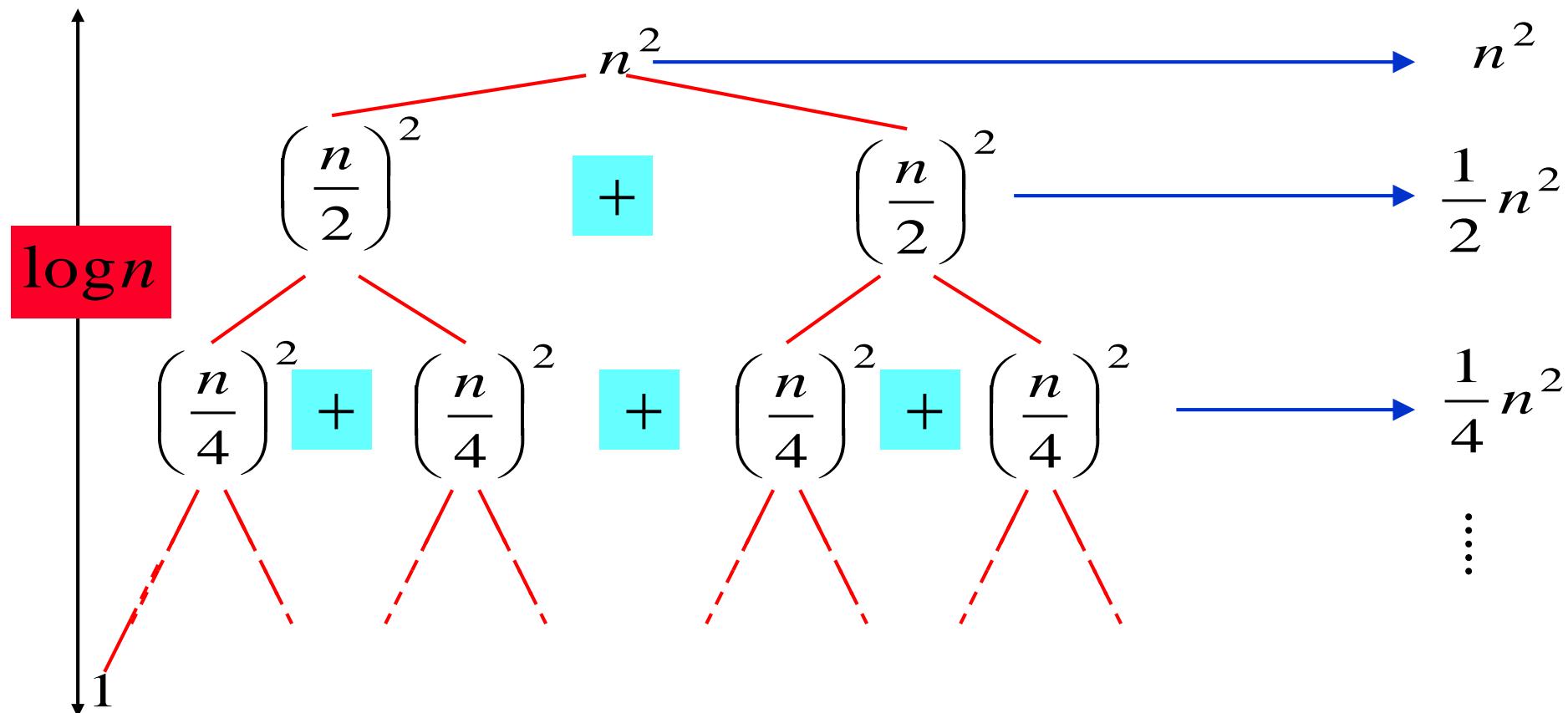
Alberi di Ricorrenza

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



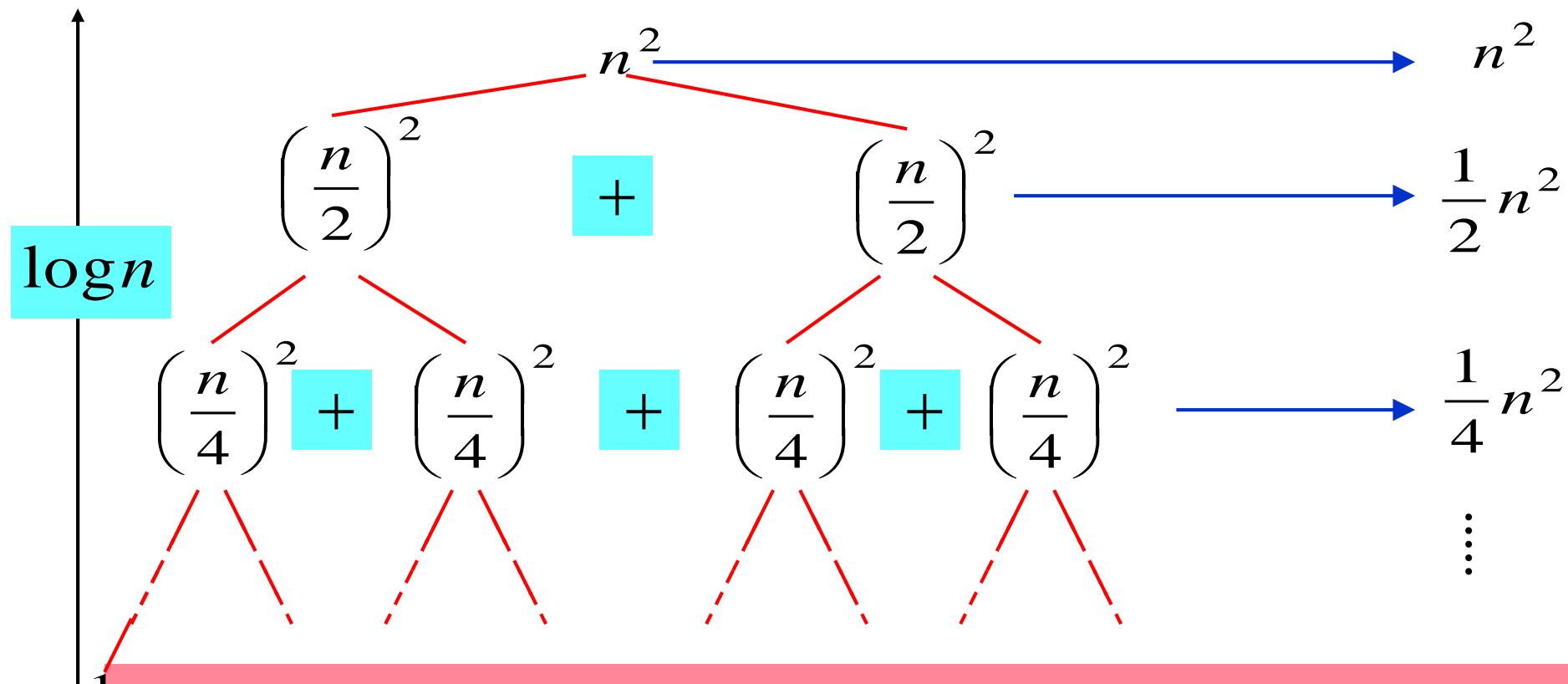
Alberi di Ricorrenza

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



Alberi di Ricorrenza

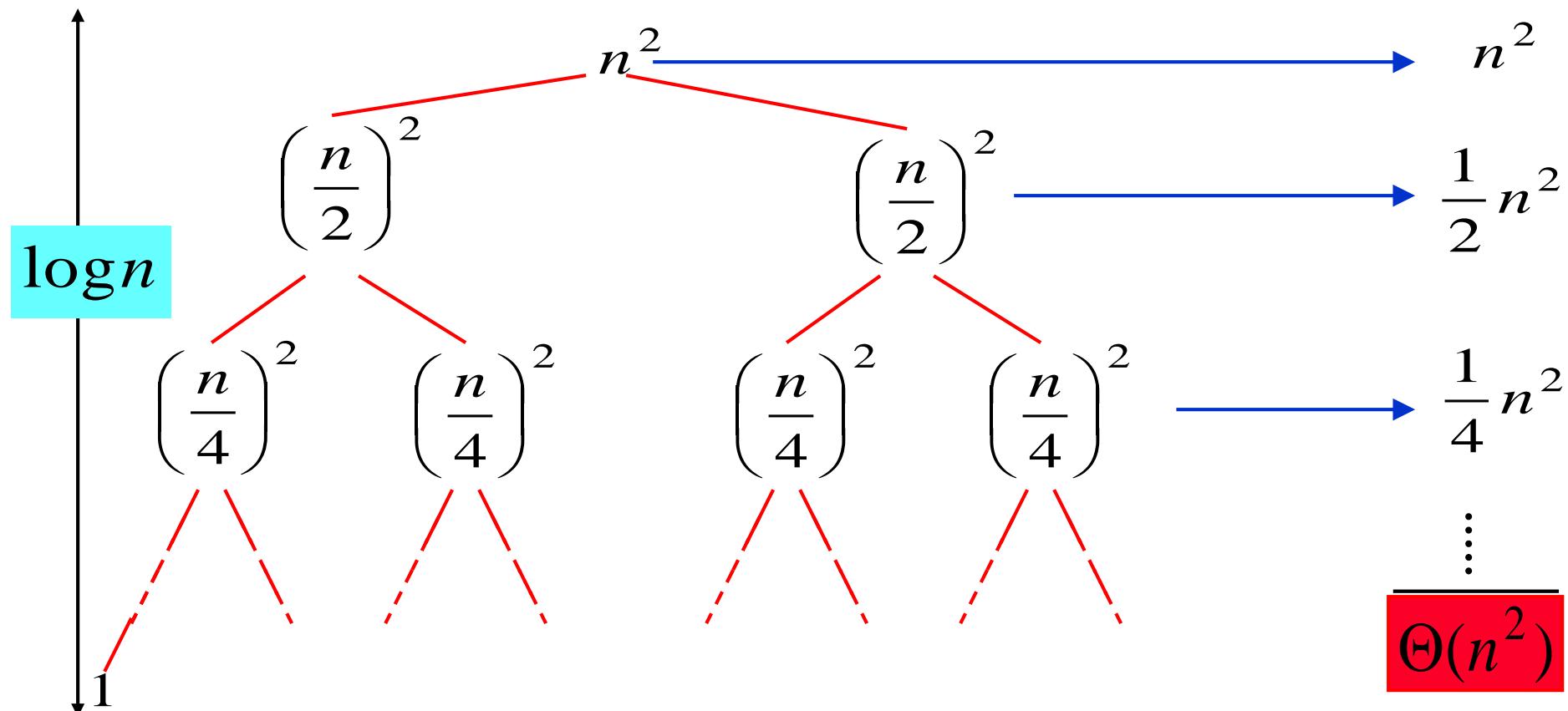
Esempio: $T(n) = 2T(n/2) + n^2$



$$T(n) = \sum_{k=0}^{\log n} \left(\frac{1}{2}\right)^k n^2 = n^2 \sum_{k=0}^{\log n} \left(\frac{1}{2}\right)^k \leq n^2 \sum_{k=0}^{\infty} \left(\frac{1}{2}\right)^k = 2n^2$$

Alberi di Ricorrenza

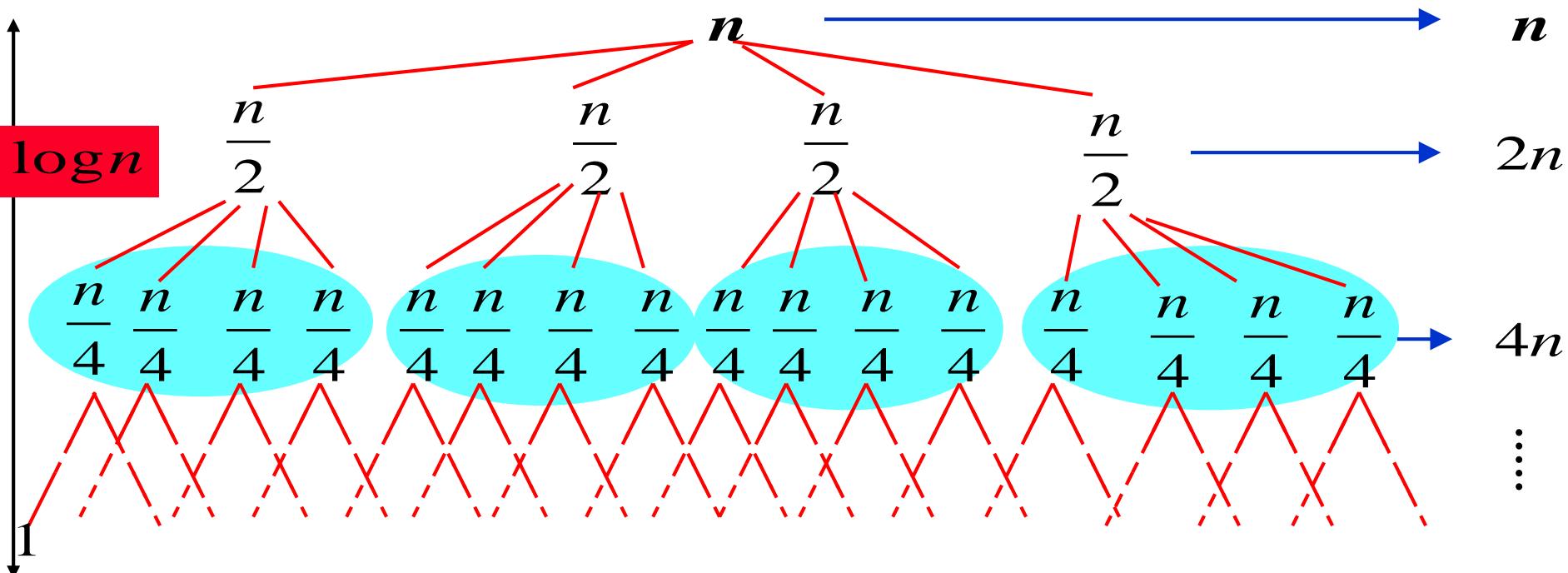
Esempio: $T(n) = 2T(n/2) + n^2$



$$T(n) = \sum_{k=0}^{\log n} \left(\frac{1}{2}\right)^k n^2 = n^2 \sum_{k=0}^{\log n} \left(\frac{1}{2}\right)^k \leq n^2 \sum_{k=0}^{\infty} \left(\frac{1}{2}\right)^k = 2n^2$$

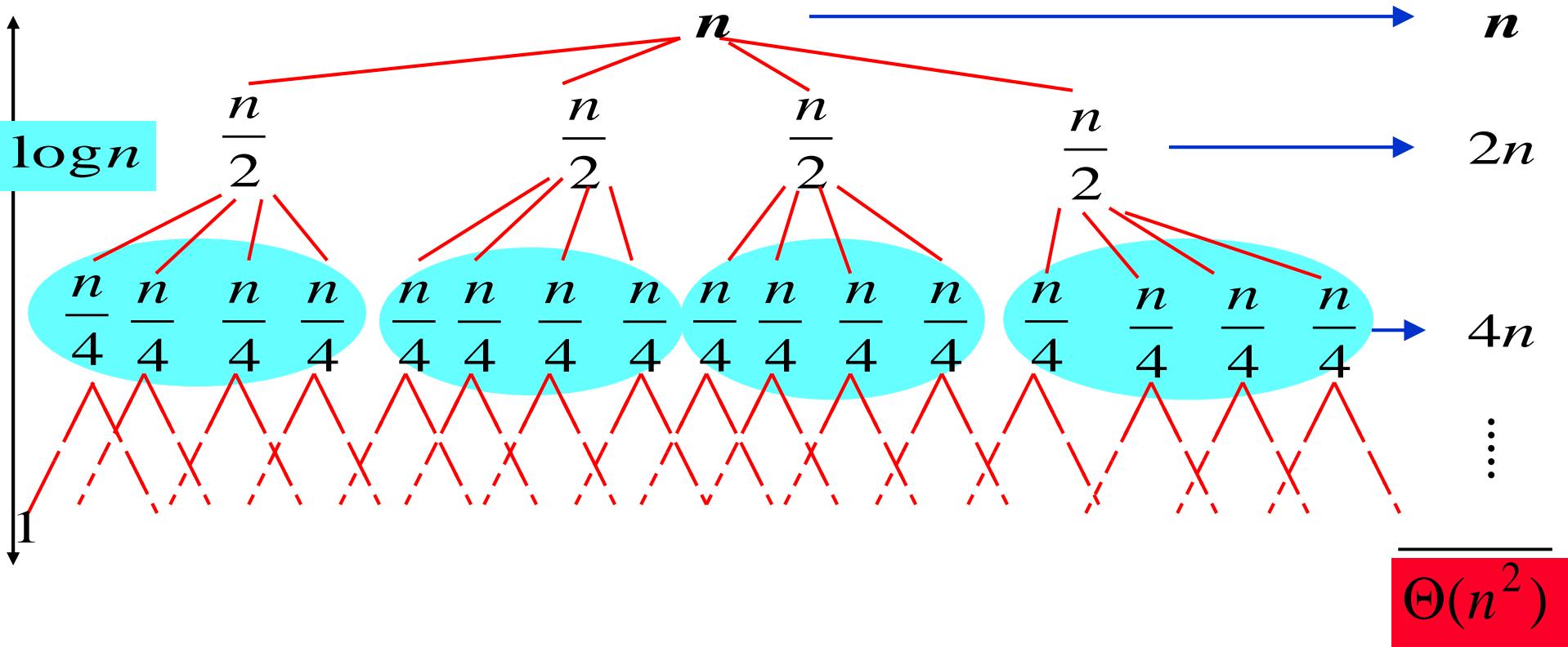
Alberi di Ricorrenza

Esempio: $T(n) = 4T(n/2) + n$



Alberi di Ricorrenza

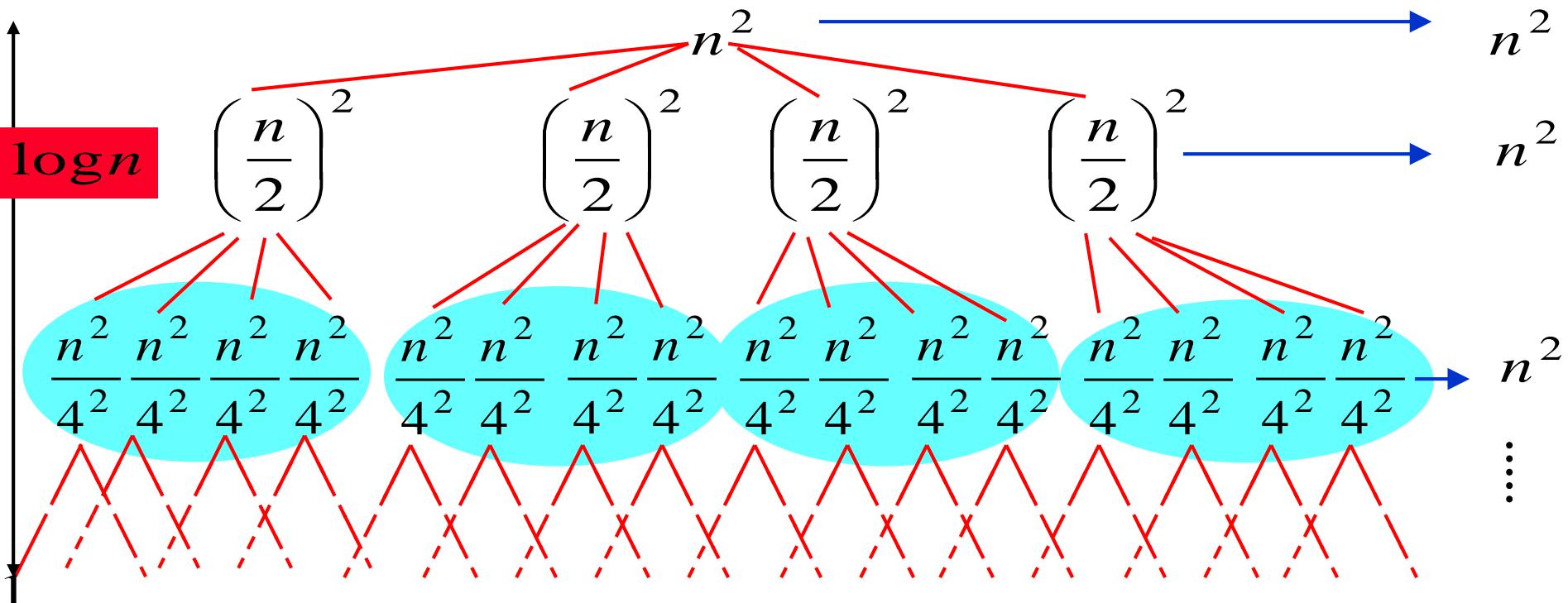
Esempio: $T(n) = 4T(n/2) + n$



$$T(n) = \sum_{k=0}^{\log n} n 2^k = n \sum_{k=0}^{\log n} 2^k = \frac{2^{\log n + 1} - 1}{2 - 1} n = (2n - 1)n = 2n^2 - n$$

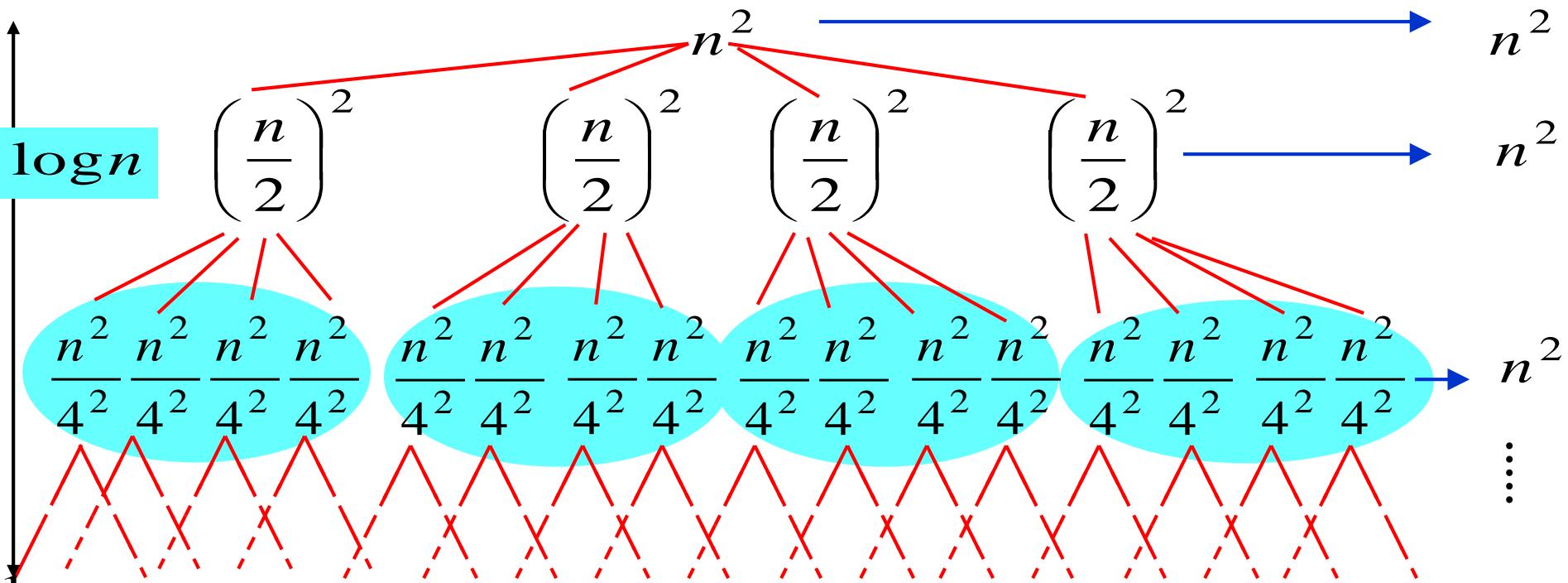
Alberi di Ricorrenza

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



Alberi di Ricorrenza

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



$$T(n) = \sum_{k=1}^{\log n} n^2 = n^2 \sum_{k=1}^{\log n} 1 = n^2 \log n$$

$\Theta(n^2 \log n)$

Analisi di QuickSort: caso medio

Il *tempo di esecuzione* di QuickSort dipende dal **bilanciamento** delle partizioni effettuate dall'algoritmo **Partiziona**

- Ci resta da capire come si comporta nel **caso medio**: è più vicino al **caso migliore** o al **caso peggiore**?

Analisi di QuickSort: caso medio

Analizziamo alcuni possibili casi di cattivo bilanciamento delle partizioni.

- Supponiamo che ad ogni chiamata l'algoritmo **Partiziona** produca una partizione che è **9** volte l'altra (**partizionamento sbilanciato**)
- Supponiamo che ad ogni chiamata l'algoritmo **Partiziona** produca una partizione che è **99** volte l'altra (**partizionamento molto sbilanciato**)

Analisi di QuickSort: caso medio

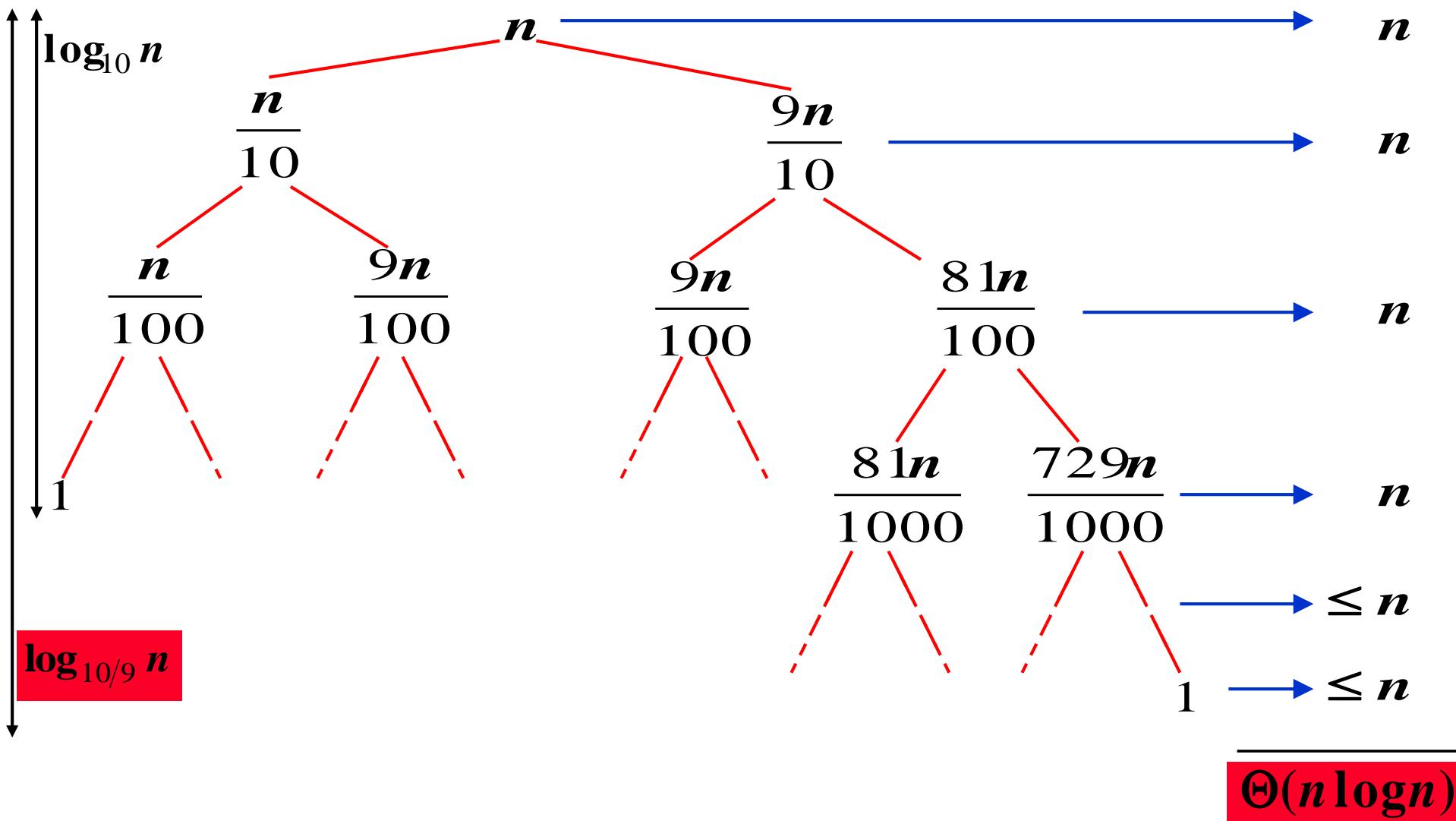
- Supponiamo che ad ogni chiamata l'algoritmo **Partiziona** produca una partizione che è 9 volte l'altra (**partizionamento sbilanciato**)

L'equazione di ricorrenza diventa quindi:

$$T(n) = T(9n/10) + T(n/10) + n$$

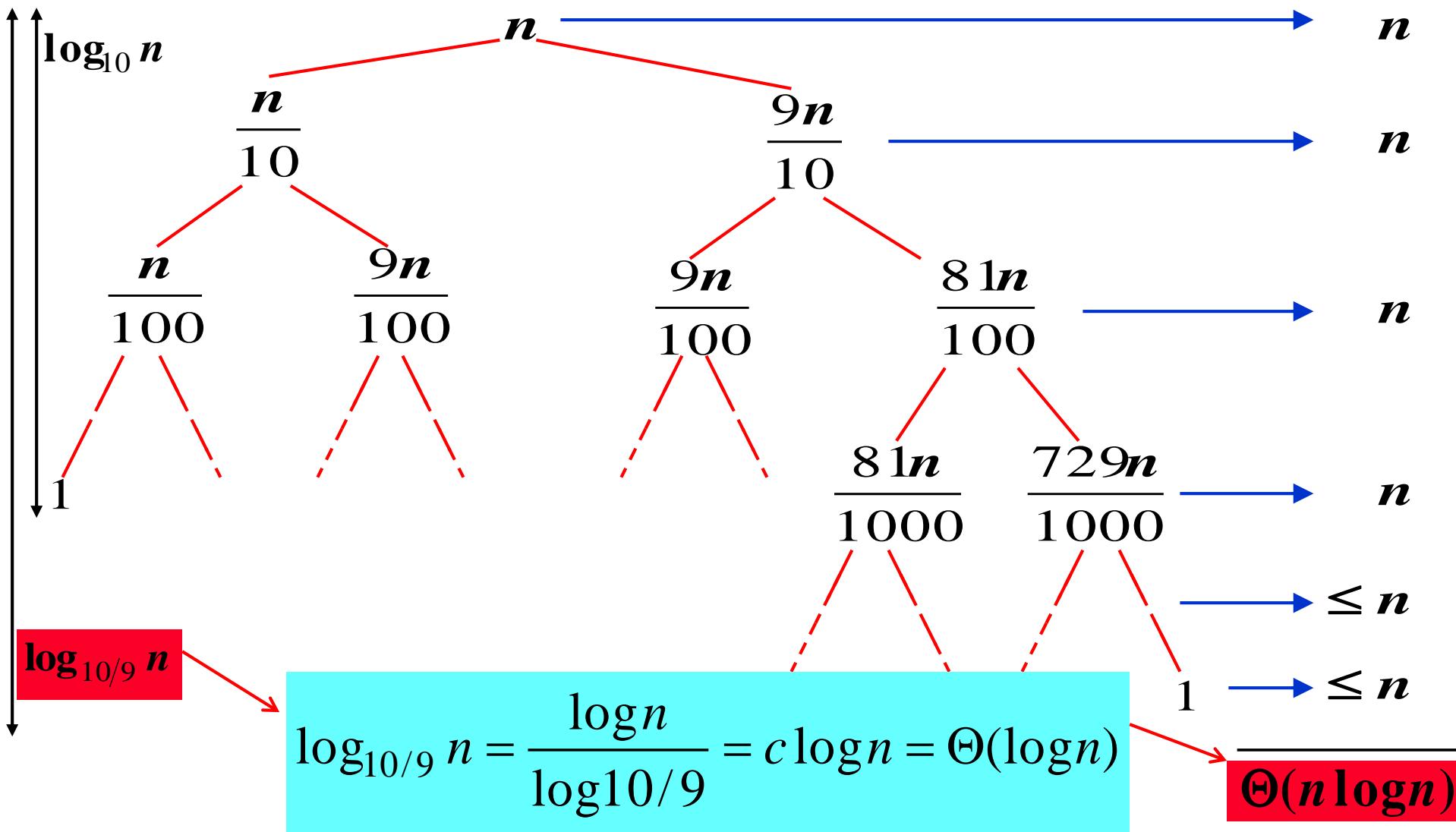
Analisi di QuickSort: caso medio

$$T(n) = T(9n/10) + T(n/10) + n$$



Analisi di QuickSort: caso medio

$$T(n) = T(9n/10) + T(n/10) + n$$



Analisi di QuickSort: caso medio

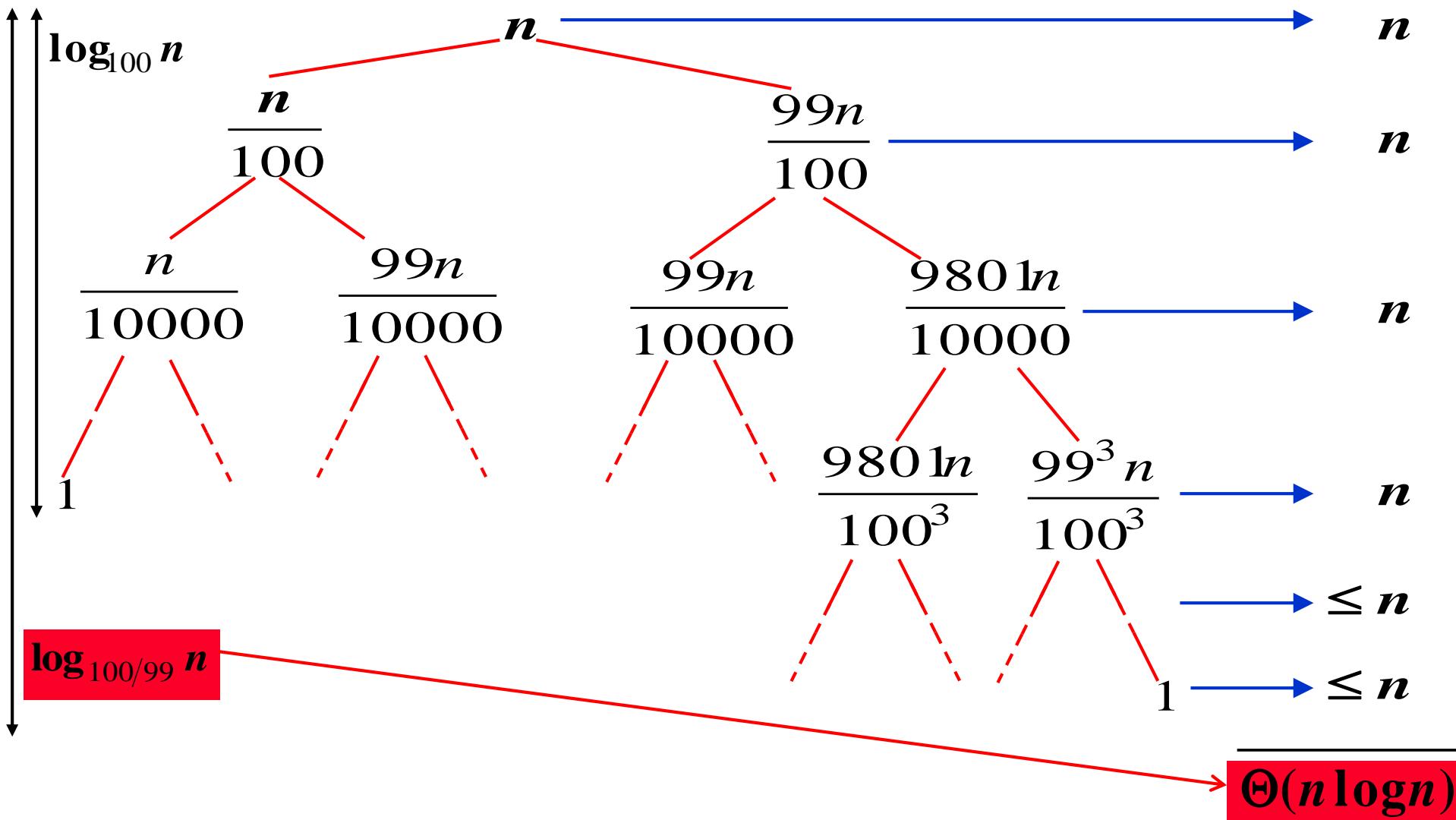
- Supponiamo che ad ogni chiamata l'algoritmo **Partiziona** produca una partizione che è **99 volte l'altra (*partizionamento sbilanciato*)**

L'equazione di ricorrenza diventa quindi:

$$T(n) = T(99n/100) + T(n/100) + n$$

Analisi di QuickSort: caso medio

$$T(n) = T(99n/100) + T(n/100) + n$$



Analisi di QuickSort: caso medio

In effetti si può dimostrare che:

ogni volta che **Partiziona** suddivide l'array
in *porzioni* che *differiscono* per un **fattore**
proporzionale costante,

il **Tempo di Esecuzione** è $\Theta(n \log n)$

Analisi di QuickSort: caso medio

Per fare un'analisi corretta del caso medio, è necessario definire una nozione chiara di *caso medio*.

Assumiamo allora che tutte le *permutazioni* dei valori in input abbiamo *uguale probabilità* di presentarsi.

In tal caso, se *QuickSort* è eseguito su un array di input casuale (*random*), ci aspettiamo che alcune *partizioni* siano *ben bilanciate* ed altre *mal bilanciate*.

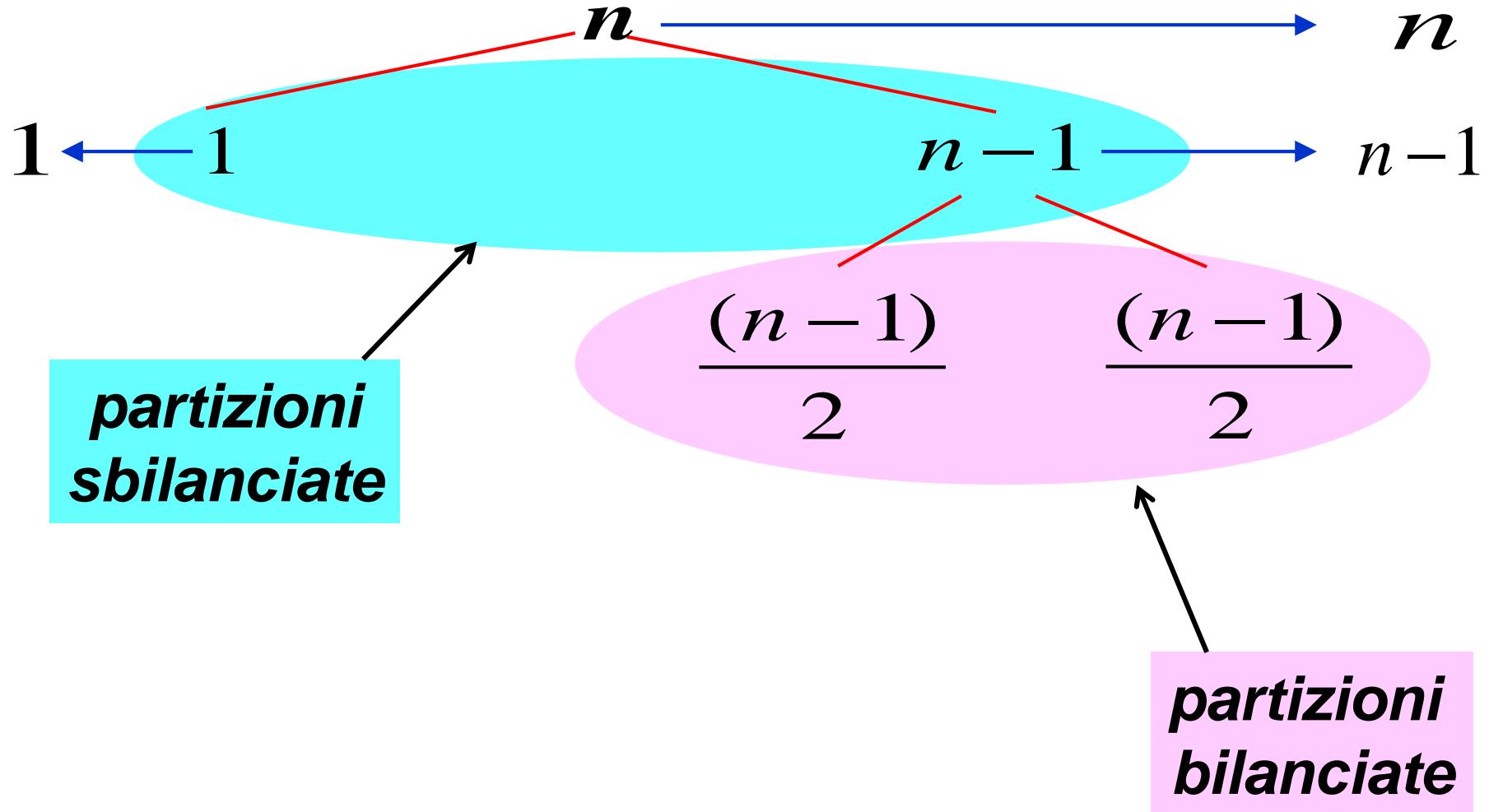
Analisi di QuickSort: caso medio

Nel caso medio **Partiziona** produrrà un “**mix**” di *partizioni ben bilanciate* e *mal bilanciate*, distribuite casualmente lungo l’albero di ricorsione.

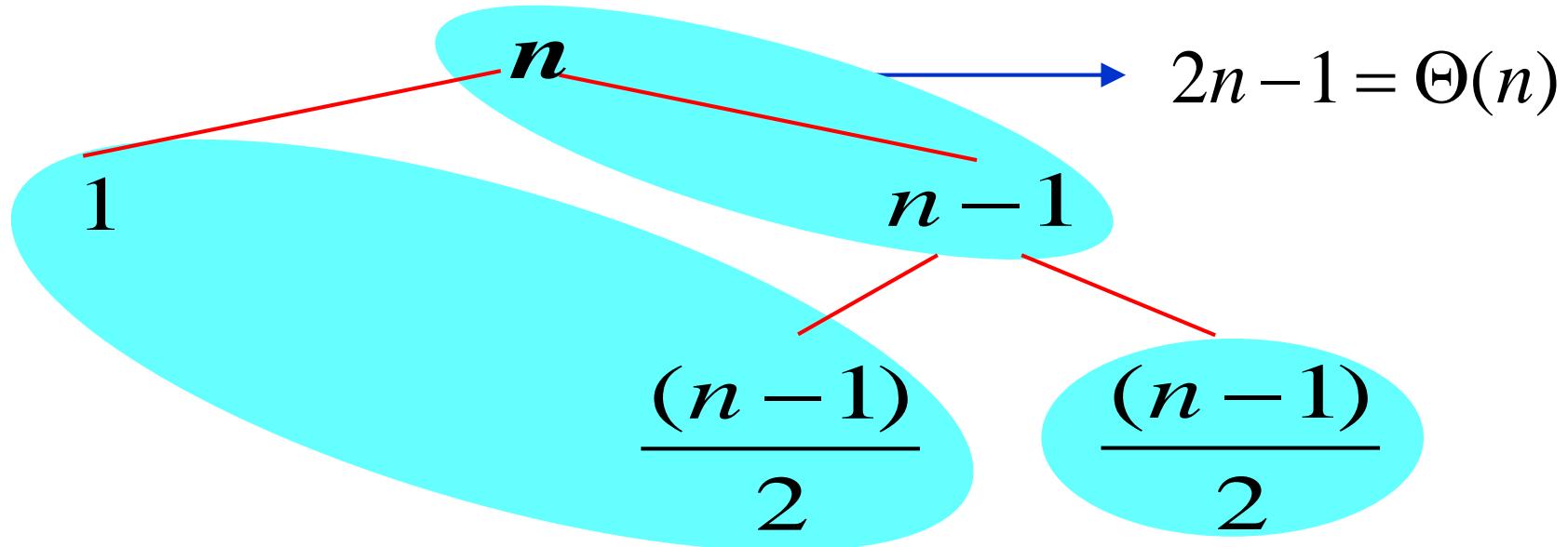
Supponiamo che le *partizioni ben bilanciate* e quelle *mal bilanciate* si alternino nei diversi livelli dell’albero, cioè:

- a livello i le partizioni sono di dimensioni 1 e $n - 1$
- a livello $i + 1$ le partizioni sono di dimensioni $n/2$ ed $n/2$

Analisi di QuickSort: caso medio

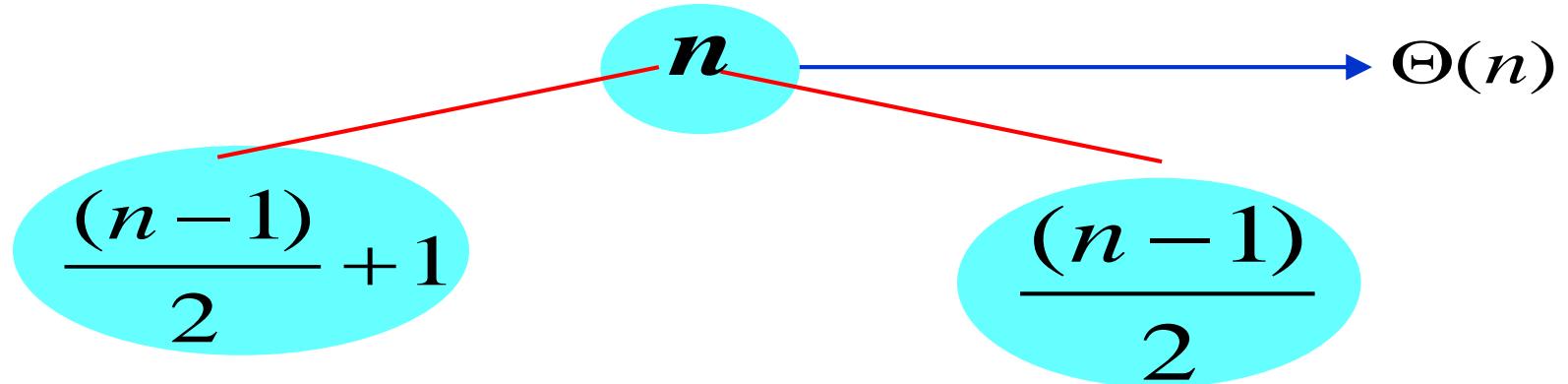


Analisi di QuickSort: caso medio



Combinando il costo di un partizionamento sbilanciato seguito da uno bilanciato, si ottiene un costo combinato sui due livelli che è $\Theta(n)$

Analisi di QuickSort: caso medio



*La situazione del **partizionamento precedente** non è asintoticamente peggiore di questa, che ha ancora costo dell'ordine di $\Theta(n)$ e rappresenta un **partizionamento piuttosto ben bilanciato***

Analisi di QuickSort: caso medio

Dunque, supponendo che le partizioni ben bilanciate e quelle mal bilanciate si alternino nei diversi livelli dell'albero:

otteniamo che in questo caso il costo medio è ancora $O(n \log n)$

dove, però, ora la notazione O-grande nasconde una costante maggiore che nel caso migliore

Analisi di QuickSort

*L'analisi che abbiamo fatto si basa sull'**assunzione** che ciascun input abbia uguale probabilità di presentarsi.*

*Questa non è però sempre un'**assunzione** sufficientemente generale!*

*Possiamo fare di più! Invece di assumere una distribuzione casuale, è possibile **imporla!***

ad esempio permutando in maniera casuale gli elementi dell'array in input

Analisi di QuickSort Random

```
int Partiziona-Random (A, p, r)
    i = Random (p, r)
    "scambia A[p] con A[i]"
    return Partiziona (A, p, r)
```

Random (*p, r*) : ritorna un valore
intero casuale compreso tra *p* ed *r*.

Analisi di QuickSort Random

```
int Partiziona-Random(A, p, r)
    i = Random(p, r)
    "scambia A[p] con A[i]"
    return Partiziona(A, p, r)
```

Sposta in $A[p]$ il valore contenuto
in $A[i]$ determinando così una
scelta casuale del *Pivot*.

Analisi di QuickSort Random

```
int Partiziona-Random(A, p, r)
    i = Random(p, r)
    "scambia A[p] con A[i]"
    return Partiziona(A, p, r)
```

```
Quick-Sort-Random(A, p, r)
IF p < r
THEN
    q = Partiziona-Random(A, p, r)
    Quick-Sort-Random(A, p, q)
    Quick-Sort-Random(A, q + 1, r)
```

Analisi di QuickSort Random

La versione random di QuickSort presentata:

- *non modifica le prestazioni nel caso peggiore (che rimane quadratico) Perché?*
- *né modifica le prestazioni nel caso migliore o medio.*
- *ma rende le prestazioni indipendenti dall'ordinamento iniziale dell'array di input.*
- *non c'è alcun particolare input che determina il verificarsi del caso peggiore (né del caso migliore).*

Analisi di QuickSort Random: Caso Peggio

Partiziona suddivide un array di dimensione n in due partizioni di dimensioni (casuali) non note, che diremo q e $n - q$, rispettivamente.

Per calcolare il caso peggiore, cercheremo di calcolare il valore massimo del tempo di esecuzione dato dalla ricorrenza

$$T(n) = T(q) + T(n-q) + \Theta(n)$$

Analisi di QuickSort Random: Caso Peggio

Partiziona suddivide un array di dimensione n in due partizioni di dimensioni (casuali) non note, che diremo q e $n - q$, rispettivamente.

Per calcolare il caso peggiore, cercheremo di calcolare il valore **massimo**, al variare di q , del tempo di esecuzione dato dalla ricorrenza

$$T(n) = T(q) + T(n-q) + \Theta(n)$$

Cioè:

$$T(n) = \max_{1 \leq q \leq n-1} \{ T(q) + T(n-q) \} + \Theta(n)$$

Analisi di QuickSort Random: Caso Peggior

$$T(n) = \max_{1 \leq q \leq n-1} \{ T(q) + T(n-q) \} + \Theta(n)$$

Usiamo il metodo di sostituzione

Ipotizziamo $T(n) \leq cn^2$

Sostituendo otteniamo

$$\begin{aligned} T(n) &\leq \max_{1 \leq q \leq n-1} \{ cq^2 + c(n-q)^2 \} + \Theta(n) \\ &\leq c \max_{1 \leq q \leq n-1} \{ q^2 + (n-q)^2 \} + \Theta(n) \end{aligned}$$

Analisi di QuickSort Random: Caso Peggior

$$T(n) = \max_{1 \leq q \leq n-1} \{ T(q) + T(n-q) \} + \Theta(n)$$

$$T(n) \leq c \max_{1 \leq q \leq n-1} \{ q^2 + (n-q)^2 \} + \Theta(n)$$

Ci serve sapere quando $q^2 + (n-q)^2$ raggiunge il valore massimo tra 1 e $n-1$

Calcoliamo la sua derivata prima:

$$2q - 2(n-q) = 4q - 2n$$

che è negativa per $q < n/2$ e positiva per $q > n/2$

Analisi di QuickSort Random: Caso Peggior

$$T(n) = \max_{1 \leq q \leq n-1} \{ T(q) + T(n-q) \} + \Theta(n)$$

$$T(n) \leq c \max_{1 \leq q \leq n-1} \{ q^2 + (n-q)^2 \} + \Theta(n)$$

La derivata prima:

$$2q - 2(n-q) = 4q - 2n$$

è negativa per $q < n/2$ e positiva per $q > n/2$

Quindi, $q^2 + (n-q)^2$ nell'intervallo $[1, n-1]$ raggiunge il valore massimo quando $q=1$ o $q=n-1$.

Analisi di QuickSort Random: Caso Peggior

$$T(n) = \max_{1 \leq q \leq n-1} \{ T(q) + T(n-q) \} + \Theta(n)$$

$$\begin{aligned} T(n) &\leq c \max_{1 \leq q \leq n-1} \{ q^2 + (n-q)^2 \} + \Theta(n) \\ &\leq c (1^2 + (n-1)^2) + \Theta(n) \\ &\leq c (n^2 - 2(n-1)) + \Theta(n) \\ &\leq c n^2 - 2c(n-1) + \Theta(n) \end{aligned}$$

Analisi di QuickSort Random: Caso Peggior

$$T(n) = \max_{1 \leq q \leq n-1} \{ T(q) + T(n-q) \} + \Theta(n)$$

$$\begin{aligned} T(n) &\leq c \max_{1 \leq q \leq n-1} \{ q^2 + (n-q)^2 \} + \Theta(n) \\ &\leq c (1^2 + (n-1)^2) + \Theta(n) \\ &\leq c (n^2 - 2(n-1)) + \Theta(n) \\ &\leq c n^2 - 2c(n-1) + \Theta(n) \\ &\leq c n^2 \end{aligned}$$

*poiché possiamo scegliere c abbastanza grande
da rendere $2c(n-1)$ dominante su $\Theta(n)$*

Analisi di QuickSort Random: Caso Migliore

Partiziona suddivide un array di dimensione ***n*** in due partizioni di dimensioni (casuali) non note, che chiamiamo ***q*** e ***n-q***, rispettivamente.

Per calcolare il caso migliore, cercheremo di calcolare il valore **minimo** del tempo di esecuzione dato dalla ricorrenza

$$T(n) = T(q) + T(n-q) + \Theta(n)$$

Cioè:

$$T(n) = \min_{1 \leq q \leq n-1} \{ T(q) + T(n-q) \} + \Theta(n)$$

Analisi di QuickSort Random: Caso Migliore

$$T(n) = \min_{1 \leq q \leq n-1} \{ T(q) + T(n-q) \} + \Theta(n)$$

Usiamo il metodo di sostituzione

Ipotizziamo $T(n) \leq c n \log n$

Analisi di QuickSort Random: Caso Migliore

$$T(n) = \min_{1 \leq q \leq n-1} \{ T(q) + T(n-q) \} + \Theta(n)$$

Usiamo il metodo di sostituzione

Ipotizziamo $T(n) \leq c n \log n$

Sostituendo otteniamo

$$\begin{aligned} T(n) &\leq \min_{1 \leq q \leq n-1} \{ c q \log q + c (n-q) \log (n-q) \} + \Theta(n) \\ &\leq c \min_{1 \leq q \leq n-1} \{ q \log q + (n-q) \log (n-q) \} + \Theta(n) \end{aligned}$$

Analisi di QuickSort Random: Caso Migliore

$$T(n) = \min_{1 \leq q \leq n-1} \{ T(q) + T(n-q) \} + \Theta(n)$$

$$T(n) \leq c \min_{1 \leq q \leq n-1} \{ q \log q + (n-q) \log (n-q) \} + \Theta(n)$$

Ci serve sapere quando $q \log q + (n-q) \log (n-q)$ raggiunge il valore minimo tra 1 e $n-1$

Calcoliamo la sua derivata prima:

$$\log q - \log(n-q)$$

che è nulla per $q=n/2$, negativa per $q < n/2$ e positiva per $q > n/2$ (quindi $q=n/2$ è un minimo)

Analisi di QuickSort Random: Caso Migliore

$$T(n) = \min_{1 \leq q \leq n-1} \{ T(q) + T(n-q) \} + \Theta(n)$$

$$T(n) \leq c \min_{1 \leq q \leq n-1} \{ q \log q + (n-q) \log (n-q) \} + \Theta(n)$$

La derivata prima:

$$\log q - \log(n-q)$$

che è nulla per $q=n/2$, negativa per $q < n/2$ e positiva per $q > n/2$ (cioè $q=n/2$ è un minimo)

Quindi $q \log q + (n-q) \log (n-q)$ raggiunge il valore minimo tra 1 e $n-1$ quando $q=n/2$

Analisi di QuickSort Random: Caso Migliore

$$T(n) = \min_{1 \leq q \leq n-1} \{ T(q) + T(n-q) \} + \Theta(n)$$

$$\begin{aligned} T(n) &\leq c \min_{1 \leq q \leq n-1} \{ q \log q + (n-q) \log (n-q) \} + \Theta(n) \\ &\leq c (n \log n/2) + \Theta(n) \\ &\leq c n \log n - c n + \Theta(n) \\ &\leq c n \log n - c n + \Theta(n) \end{aligned}$$

Analisi di QuickSort Random: Caso Migliore

$$T(n) = \min_{1 \leq q \leq n-1} \{ T(q) + T(n-q) \} + \Theta(n)$$

$$\begin{aligned} T(n) &\leq c \min_{1 \leq q \leq n-1} \{ q \log q + (n-q) \log (n-q) \} + \Theta(n) \\ &\leq c (n \log n/2) + \Theta(n) \\ &\leq c n \log n - c n + \Theta(n) \\ &\leq c n \log n - c n + \Theta(n) \\ &\leq c n \log n \end{aligned}$$

poiché possiamo scegliere c abbastanza grande
da rendere $c n$ dominante su $\Theta(n)$

Analisi di QuickSort Random: Caso Medio

*Quello che dobbiamo fare è costruire l'**equazione di ricorrenza** per il caso medio.*

- *Per semplificare l'analisi, assumeremo che tutti gli elementi siano distinti.*
- **Partiziona-Random** chiama **Partiziona** dopo aver scambiato **A[p]** con un elemento a caso dell'array
- *quale sarà allora il valore di **q** ritornato da Partiziona?*

Analisi di QuickSort Random: Caso Medio

*Quale sarà allora il valore di q ritornato
Partiziona?*

- Dipenderà dal **rango** di $A[p]$ (che è un elemento casuale dell'array).
- Il **rango** di un numero x rispetto a $A[p, \dots, r]$ è il numero di elementi di $A[p, \dots, r]$ che sono **minori o uguali** ad x

Analisi di QuickSort Random: Caso Medio

*Quale sarà allora il valore di q ritornato
Partiziona?*

- *Dipenderà dal rango di $A[p]$ (che è un elemento casuale dell'array).*
- *Essendo $A[p]$ un elemento casuale dell'array, la probabilità che il rango di $A[p]$ sia i (con $i = 1, \dots, n$) sarà $1/n$ (dove $n = r - p + 1$) poiché tutti gli elementi hanno uguale probabilità di essere scelti e sono tutti distinti.*

Analisi di QuickSort Random: Caso Medio

Quale sarà allora il valore di q ritornato Partiziona?

- *Se il rango è 1, Partiziona ritornerà una partizione lunga 1 e una lunga $n-1$*
- *Se il rango è 2, Partiziona ritornerà ancora una partizione lunga 1 e una lunga $n-1$*
- ...
- *Se il rango è h , Partiziona ritornerà una partizione lunga $h-1$ e una lunga $n-h+1$*
- *Se il rango è n , Partiziona ritornerà una partizione lunga $n-1$ e una lunga 1*

Analisi di QuickSort Random: Caso Medio

Quale sarà allora il valore di q ritornato Partiziona?

- **Se il rango è 1, Partiziona ritornerà una partizione lunga 1 e una lunga $n-1$**
- **Se il rango è h (per $h \geq 2$), Partiziona ritornerà una partizione lunga $h-1$ e una lunga $n-h+1$**

ciascun caso ha probabilità $1/n$

Nota: tutto questo è garantito solo se gli elementi sono tutti distinti!

Analisi di QuickSort Random: Caso Medio

Quale sarà allora il valore di q ritornato Partiziona?

- **Se il rango è 1 Partiziona ritornerà una partizione lunga 1 e una lunga $n-1$ allora $q = 1$ e QuickSort sarà chiamato ricorsivamente su partizioni di dimensioni rispettivamente 1 ed $n-1$ con probabilità $1/n$**

Nota: tutto questo è garantito solo se gli elementi sono tutti distinti!

Analisi di QuickSort Random: Caso Medio

Quale sarà allora il valore di q ritornato Partiziona?

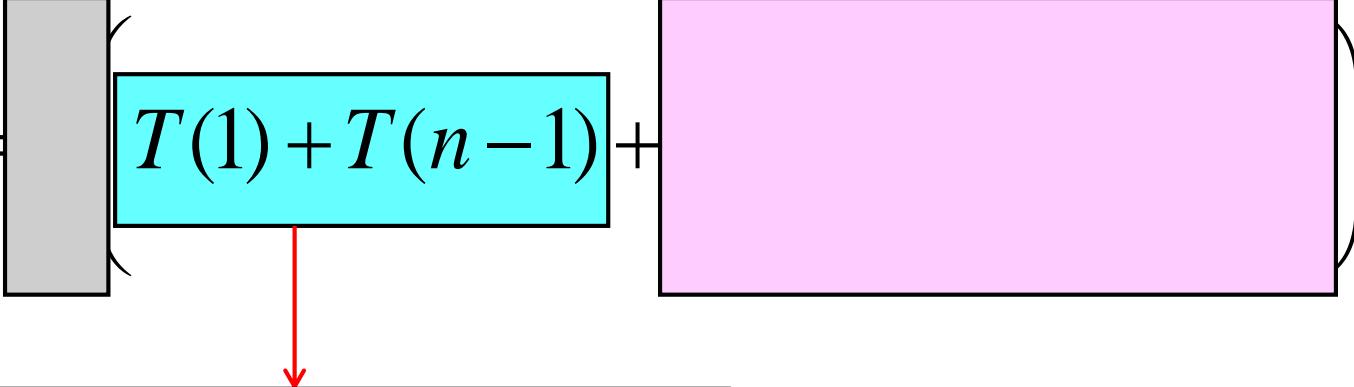
- *Se il rango è h (per $h \geq 2$) Partiziona ritornerà una partizione lunga $h-1$ e una lunga $n-h+1$*

allora $q = h - 1$ e QuickSort sarà chiamato ricorsivamente su partizioni di dimensioni $h-1$ e $n-h+1$

con probabilità $1/n$

***Nota:** tutto questo è garantito solo se gli elementi sono tutti distinti!*

Analisi di QuickSort Random: Caso Medio

$$T(n) = \boxed{T(1) + T(n-1)} + \boxed{\Theta(n)}$$


**Se il rango è 1 Partiziona
ritornerà una partizione
lunga 1 e una lunga n-1**

Analisi di QuickSort Random: Caso Medio

$$T(n) = \left[T(1) + T(n-1) \right] + \sum_{q=1}^{n-1} (T(q) + T(n-q)) + \Theta(n)$$

Se il **rango** è 1 Partiziona
ritornerà una partizione
lunga 1 e una lunga **n-1**

Se il **rango** è **h** (per $2 \leq h \leq n$)
Partiziona ritornerà una
partizione lunga **h-1** e una lunga
n-h+1 (e **q** varia tra 1 e **n-1**)

Analisi di QuickSort Random: Caso Medio

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

Se il **rango** è 1 Partiziona
ritornerà una partizione
lunga 1 e una lunga **n-1**

ciascun caso ha
probabilità $1/n$

Se il **rango** è **h** (per **$h \geq 2$**)
Partiziona ritornerà una
partizione lunga **$h-1$** e una lunga
 $n-h+1$ (**q** varia tra 1 e **n-1**)

Analisi di QuickSort Random: Caso Medio

L'equazione di ricorrenza per il caso medio sarà quindi:

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

Analisi di QuickSort Random: Caso Medio

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

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

Analisi di QuickSort Random: Caso Medio

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

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

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

Analisi di QuickSort Random: Caso Medio

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

poiché $O(n)$ viene assorbito da $\Theta(n)!$ (Perché?)

$$\frac{1}{n} (T(1) + T(n-1)) = O(n)$$

Analisi di QuickSort Random: Caso Medio

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

$$= \frac{2}{n} \left(\sum_{q=1}^{n-1} T(q) \right) + \Theta(n)$$

poiché per q che varia fra 1 e $n-1$ ciascun valore di $T(q)$ compare due volte nella sommatoria, una volta come $T(q)$ ed una come $T(n-q)$.

Analisi di QuickSort Random: Caso Medio

L'equazione di ricorrenza diviene:

$$T(n) = \frac{2}{n} \left(\sum_{q=1}^{n-1} T(q) \right) + \Theta(n)$$

La risolveremo col metodo di sostituzione

Analisi di QuickSort Random: Caso Medio

L'equazione di ricorrenza diviene:

$$T(n) = \frac{2}{n} \left(\sum_{q=1}^{n-1} T(q) \right) + \Theta(n)$$

Vogliamo dimostrare che $T(n) = O(n \log n)$

Analisi di QuickSort Random: Caso Medio

L'equazione di ricorrenza diviene:

$$T(n) = \frac{2}{n} \left(\sum_{q=1}^{n-1} T(q) \right) + \Theta(n)$$

Ipotizziamo

$$T(n) \leq a n \log n$$

Analisi di QuickSort Random: Caso Medio

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

$$T(n) = \frac{2}{n} \left(\sum_{q=1}^{n-1} T(q) \right) + \Theta(n)$$

$$\leq \frac{2}{n} \left(\sum_{q=1}^{n-1} aq \log q \right) + \Theta(n)$$

Analisi di QuickSort Random: Caso Medio

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

$$T(n) = \frac{2}{n} \left(\sum_{q=1}^{n-1} T(q) \right) + \Theta(n)$$

$$\leq \frac{2}{n} \left(\sum_{q=1}^{n-1} aq \log q \right) + \Theta(n)$$

$$\leq \frac{2a}{n} \sum_{q=1}^{n-1} q \log q + \Theta(n)$$

Analisi di QuickSort Random: Caso Medio

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

$$T(n) = \frac{2}{n} \left(\sum_{q=1}^{n-1} T(q) \right) + \Theta(n)$$

$$\leq \frac{2a}{n} \sum_{q=1}^{n-1} q \log q + \Theta(n)$$

poiché si può dimostrare che

$$\sum_{q=1}^{n-1} q \log q \leq \frac{1}{2} n^2 \log n - \frac{1}{8} n^2$$

Analisi di QuickSort Random: Caso Medio

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

$$T(n) = \frac{2}{n} \left(\sum_{q=1}^{n-1} T(q) \right) + \Theta(n)$$

$$\leq \frac{2a}{n} \sum_{q=1}^{n-1} q \log q + \Theta(n)$$

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

$$\sum_{q=1}^{n-1} q \log q \leq \frac{1}{2} n^2 \log n - \frac{1}{8} n^2$$

Analisi di QuickSort Random: Caso Medio

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

$$T(n) = \frac{2}{n} \left(\sum_{q=1}^{n-1} T(q) \right) + \Theta(n)$$

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

$$\leq an \log n - \frac{a}{4} n + \Theta(n)$$

Analisi di QuickSort Random: Caso Medio

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

$$T(n) = \frac{2}{n} \left(\sum_{q=1}^{n-1} T(q) \right) + \Theta(n)$$

$$\leq \frac{2a}{n} \left(\frac{1}{2} n^2 \log n - \frac{1}{8} n^2 \right) + \frac{2b}{n} (n-1) + \Theta(n)$$

$$\leq an \log n - \frac{a}{4} n + 2b + \Theta(n)$$

$$\leq an \log n + \left(\Theta(n) - \frac{a}{4} n \right)$$

Analisi di QuickSort Random: Caso Medio

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

$$T(n) = \frac{2}{n} \left(\sum_{q=1}^{n-1} T(q) \right) + \Theta(n)$$

$$\leq an \log n - \frac{a}{4} n + \Theta(n)$$

$$\leq an \log n + \left(\Theta(n) - \frac{a}{4} n \right)$$

$$\leq an \log n$$

Scegliendo a grande
abbastanza da
rendere $a n/4$
dominante su $\Theta(n)$

Analisi di QuickSort Random: Caso Medio

Possiamo concludere che

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

A patto di dimostrare che

$$\sum_{q=1}^{n-1} q \log q \leq \frac{1}{2} n^2 \log n - \frac{1}{8} n^2$$

Analisi di QuickSort Random: Caso Medio

$$\sum_{k=1}^{n-1} k \log k \leq \log n \sum_{k=1}^{n-1} k$$

$$\leq \frac{1}{2} n(n-1) \log n = \frac{n^2 - n}{2} \log n$$

$$\leq n^2 \log n$$

Questo limite non è però sufficiente per risolvere la ricorrenza, ma quello che abbiamo calcolato sarà utile per trovarne uno adeguato!

Analisi di QuickSort Random: Caso Medio

$$\sum_{k=1}^{n-1} k \log k = \sum_{k=1}^{\lceil n/2 \rceil - 1} k \log k + \sum_{k=\lceil n/2 \rceil}^{n-1} k \log k$$



$$\sum_{k=1}^{\lceil n/2 \rceil - 1} k \log k \leq \log(n/2) \sum_{k=1}^{\lceil n/2 \rceil - 1} k$$

$$\leq (\log n - 1) \sum_{k=1}^{\lceil n/2 \rceil - 1} k$$

$$\sum_{k=1}^{n-1} k \log k \leq \log n \sum_{k=1}^{n-1} k$$

Analisi di QuickSort Random: Caso Medio

$$\sum_{k=1}^{n-1} k \log k = \sum_{k=1}^{\lceil n/2 \rceil - 1} k \log k + \sum_{k=\lceil n/2 \rceil}^{n-1} k \log k$$

$$\sum_{k=1}^{\lceil n/2 \rceil - 1} k \log k \leq (\log n - 1) \sum_{k=1}^{\lceil n/2 \rceil - 1} k$$

$$\sum_{k=\lceil n/2 \rceil}^{n-1} k \log k \leq \log n \sum_{k=\lceil n/2 \rceil}^{n-1} k$$

$$\sum_{k=1}^{n-1} k \log k \leq \log n \sum_{k=1}^{n-1} k$$

Analisi di QuickSort Random: Caso Medio

$$\sum_{k=1}^{n-1} k \log k = \sum_{k=1}^{\lceil n/2 \rceil - 1} k \log k + \sum_{k=\lceil n/2 \rceil}^{n-1} k \log k$$

$$\sum_{k=1}^{\lceil n/2 \rceil - 1} k \log k \leq (\log n - 1) \sum_{k=1}^{\lceil n/2 \rceil - 1} k$$

$$\sum_{k=\lceil n/2 \rceil}^{n-1} k \log k \leq \log n \sum_{k=\lceil n/2 \rceil}^{n-1} k$$

Analisi di QuickSort Random: Caso Medio

$$\sum_{k=1}^{n-1} k \log k \leq (\log n - 1) \sum_{k=1}^{\lceil n/2 \rceil - 1} k + \log n \sum_{k=\lceil n/2 \rceil}^{n-1} k$$

$$\sum_{k=1}^{\lceil n/2 \rceil - 1} k \log k \leq (\log n - 1) \sum_{k=1}^{\lceil n/2 \rceil - 1} k$$

$$\sum_{k=\lceil n/2 \rceil}^{n-1} k \log k \leq \log n \sum_{k=\lceil n/2 \rceil}^{n-1} k$$

Analisi di QuickSort Random: Caso Medio

$$\begin{aligned} \sum_{k=1}^{n-1} k \log k &\leq (\log n - 1) \sum_{k=1}^{\lceil n/2 \rceil - 1} k + \log n \sum_{k=\lceil n/2 \rceil}^{n-1} k \\ &\leq \log n \sum_{k=1}^{\lceil n/2 \rceil - 1} k + \log n \sum_{k=\lceil n/2 \rceil}^{n-1} k - \sum_{k=1}^{\lceil n/2 \rceil - 1} k \end{aligned}$$

Analisi di QuickSort Random: Caso Medio

$$\begin{aligned} \sum_{k=1}^{n-1} k \log k &\leq (\log n - 1) \sum_{k=1}^{\lceil n/2 \rceil - 1} k + \log n \sum_{k=\lceil n/2 \rceil}^{n-1} k \\ &\leq \log n \sum_{k=1}^{\lceil n/2 \rceil - 1} k + \log n \sum_{k=\lceil n/2 \rceil}^{n-1} k - \sum_{k=1}^{\lceil n/2 \rceil - 1} k \\ &\leq \log n \sum_{k=1}^{n-1} k - \sum_{k=1}^{\lceil n/2 \rceil - 1} k \end{aligned}$$

Analisi di QuickSort Random: Caso Medio

$$\sum_{k=1}^{n-1} k \log k \leq (\log n - 1) \sum_{k=1}^{\lceil n/2 \rceil - 1} k + \log n \sum_{k=\lceil n/2 \rceil}^{n-1} k$$

$$\leq \log n \sum_{k=1}^{n-1} k - \sum_{k=1}^{\lceil n/2 \rceil - 1} k$$

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

$$\sum_{k=1}^{\lceil n/2 \rceil - 1} k = \frac{1}{2} \frac{n}{2} \left(\frac{n}{2} - 1 \right)$$

Analisi di QuickSort Random: Caso Medio

$$\sum_{k=1}^{n-1} k \log k \leq (\log n - 1) \sum_{k=1}^{\lceil n/2 \rceil - 1} k + \log n \sum_{k=\lceil n/2 \rceil}^{n-1} k$$

$$\leq \log n \sum_{k=1}^{n-1} k - \sum_{k=1}^{\lceil n/2 \rceil - 1} k$$

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

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

$$\sum_{k=1}^{\lceil n/2 \rceil - 1} k = \frac{1}{2} \frac{n}{2} \left(\frac{n}{2} - 1 \right)$$

Analisi di QuickSort Random: Caso Medio

$$\sum_{k=1}^{n-1} k \log k \leq (\log n - 1) \sum_{k=1}^{\lceil n/2 \rceil - 1} k + \log n \sum_{k=\lceil n/2 \rceil}^{n-1} k$$

$$\leq \log n \sum_{k=1}^{n-1} k - \sum_{k=1}^{\lceil n/2 \rceil - 1} k$$

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

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

$$\sum_{k=1}^{\lceil n/2 \rceil - 1} k = \frac{1}{2} \frac{n}{2} \left(\frac{n}{2} - 1 \right)$$

$$\leq \frac{1}{2} n^2 \log n - \frac{1}{8} n^2$$

Analisi di QuickSort Random: Caso Medio

Possiamo concludere che:

- *nel caso medio, il tempo di esecuzione è:*

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

- *nel caso migliore, il tempo di esecuzione è:*

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

- *nel caso peggiore, il tempo di esecuzione è:*

$$T(n) = O(n^2)$$

Algoritmi e Strutture Dati (Mod. A)

Limite Inferiore per l'Ordinamento

Limite Inferiore per l'Ordinamento

Ma quanto può essere efficiente, in principio, un algoritmo di ordinamento?

Questo tipologia di domande è una delle più ambiziose e interessanti

ma anche una delle più difficili!

Limite Inferiore per l'Ordinamento

Ma quanto può essere efficiente, in principio, un algoritmo di ordinamento?

*La difficoltà risiede nel fatto che **non** ci stiamo chiedendo quale sia l'efficienza di uno specifico algoritmo di ordinamento, ma qual è il **minimo tempo di esecuzione** di un qualunque algoritmo di ordinamento.*

La risposta richiederebbe quindi di considerare tutti i possibili algoritmi di ordinamento, anche quelli mai sviluppati.

Limite Inferiore per l'Ordinamento

In generale, per rispondere ad una domanda del tipo “qual è il modo più veloce per eseguire un compito” dobbiamo definire prima

quali strumenti abbiamo a disposizione

La risposta infatti dipende in genere proprio da questo.

Limite Inferiore per l'Ordinamento

In generale, per rispondere ad una domanda del tipo “qual è il modo più veloce per eseguire un compito” dobbiamo definire prima

quali strumenti abbiamo a disposizione

Nel caso dell’ordinamento considereremo come unico strumento

il confronto di elementi a coppie

*Il problema dell’ordinamento può essere risolto utilizzando solo dei **confronti** tra elementi*

Assunzione sugli elementi

Supponiamo di voler ordinare n elementi

$$K_1, K_2, \dots, K_n$$

Assumiamo, per semplicità, che tutti gli elementi siano distinti

Questo significa che:

per ogni coppia di elementi K_i e K_j , se $i \neq j$ allora

- $K_i < K_j$ oppure
- $K_i > K_j$

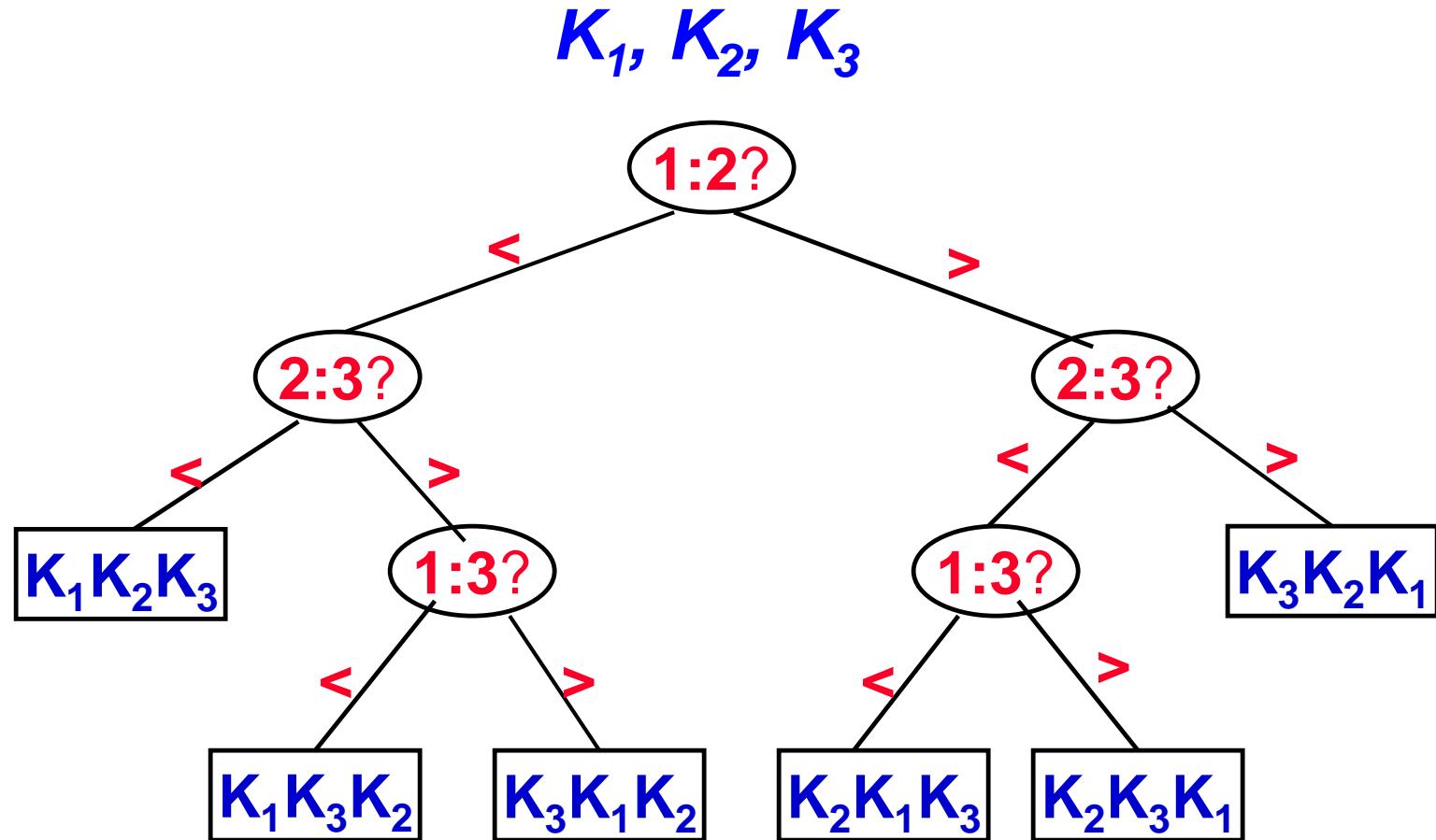
Alberi di Decisione

*Per analizzare il problema che ci siamo posti, utilizzeremo come modello astratto quello degli “Alberi di Decisione” (o *Alberi di Confronto*).*

*Gli Alberi di Decisione ci permettono di rappresentare, a livello astratto, il **comportamento** di qualsiasi algoritmo di ordinamento basato su **confronto di elementi***

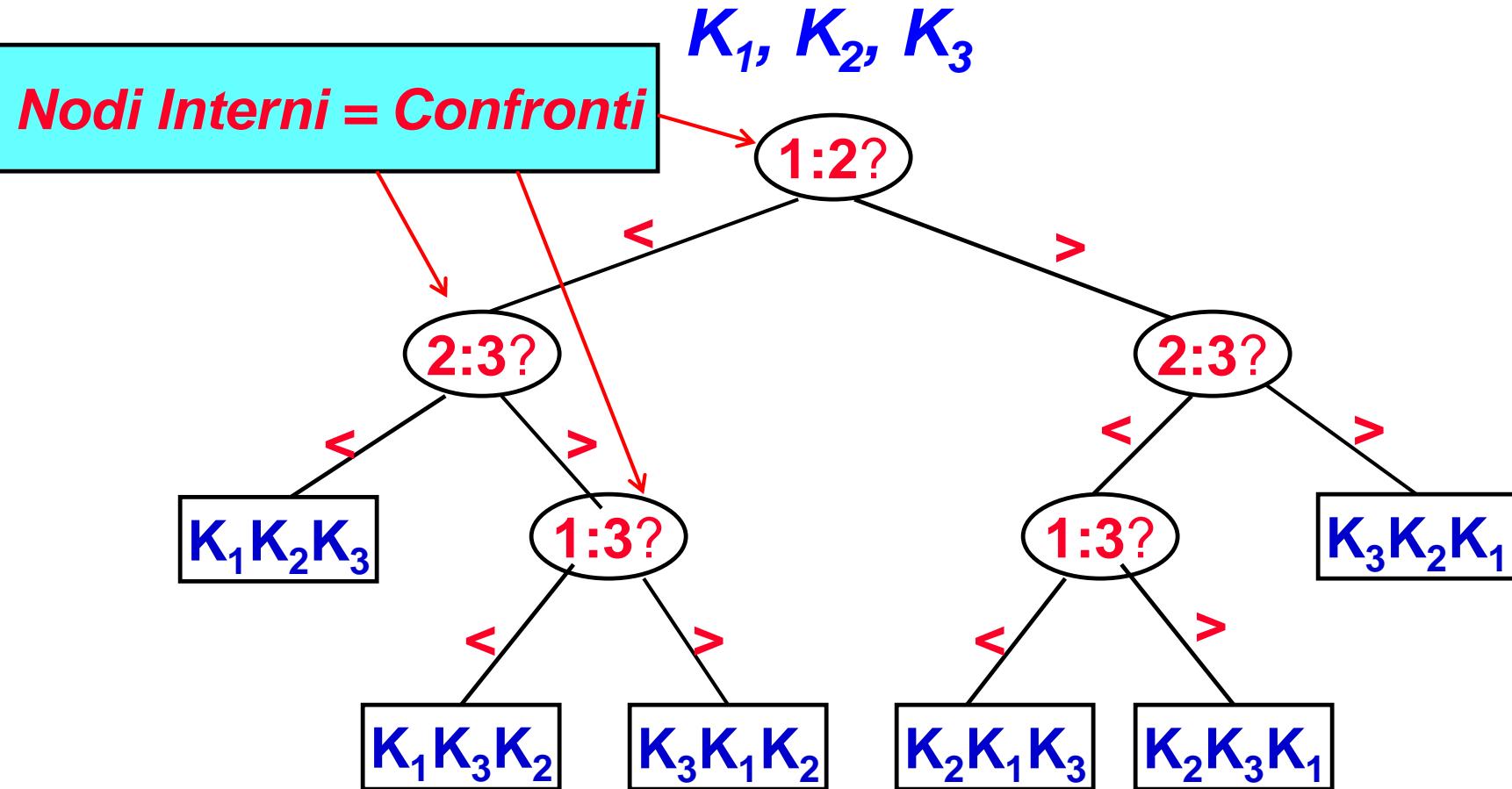
Alberi di Decisione: esempio

Siano dati tre elementi arbitrari:



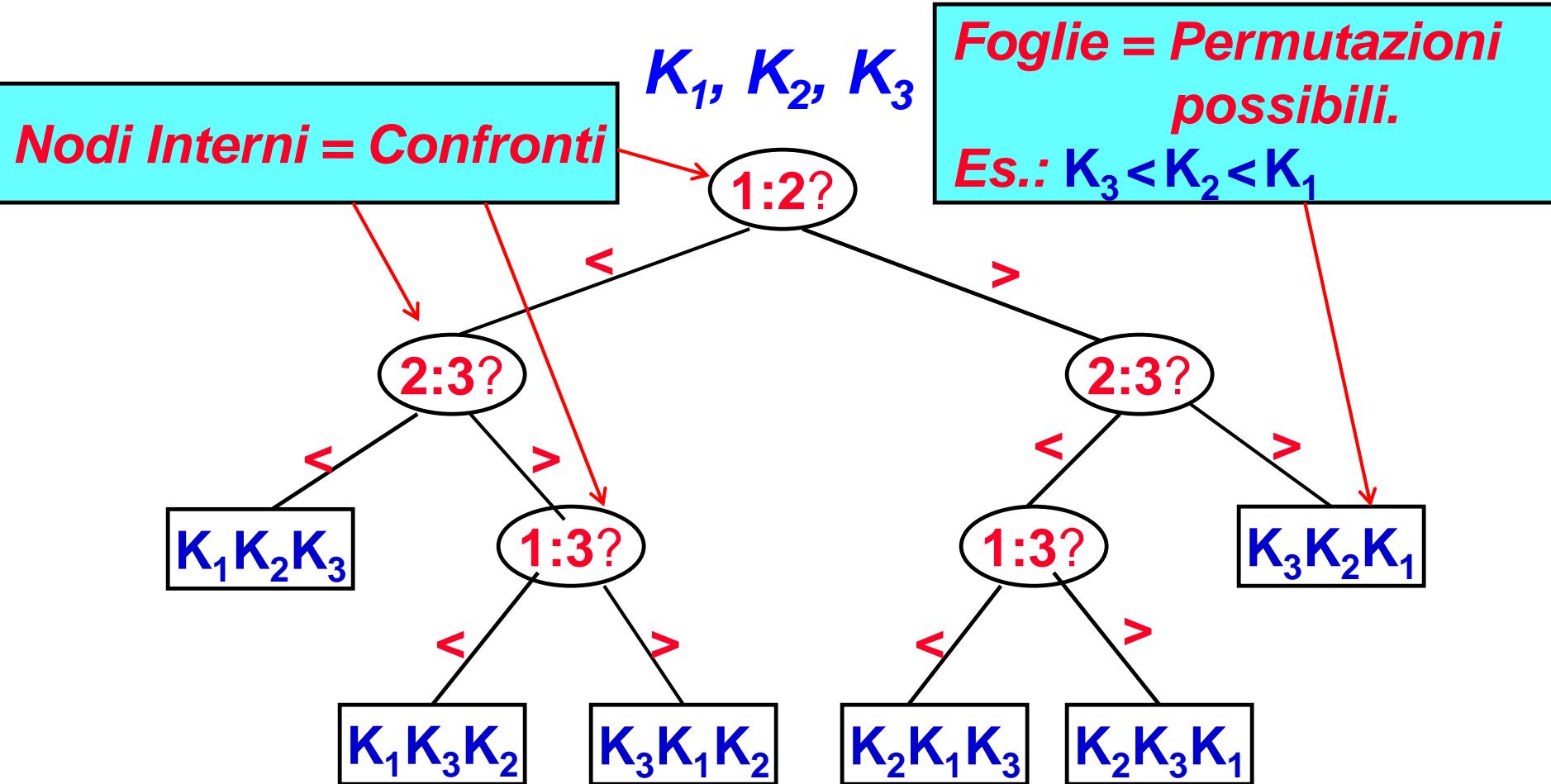
Alberi di Decisione: esempio

Siano dati tre elementi arbitrari:



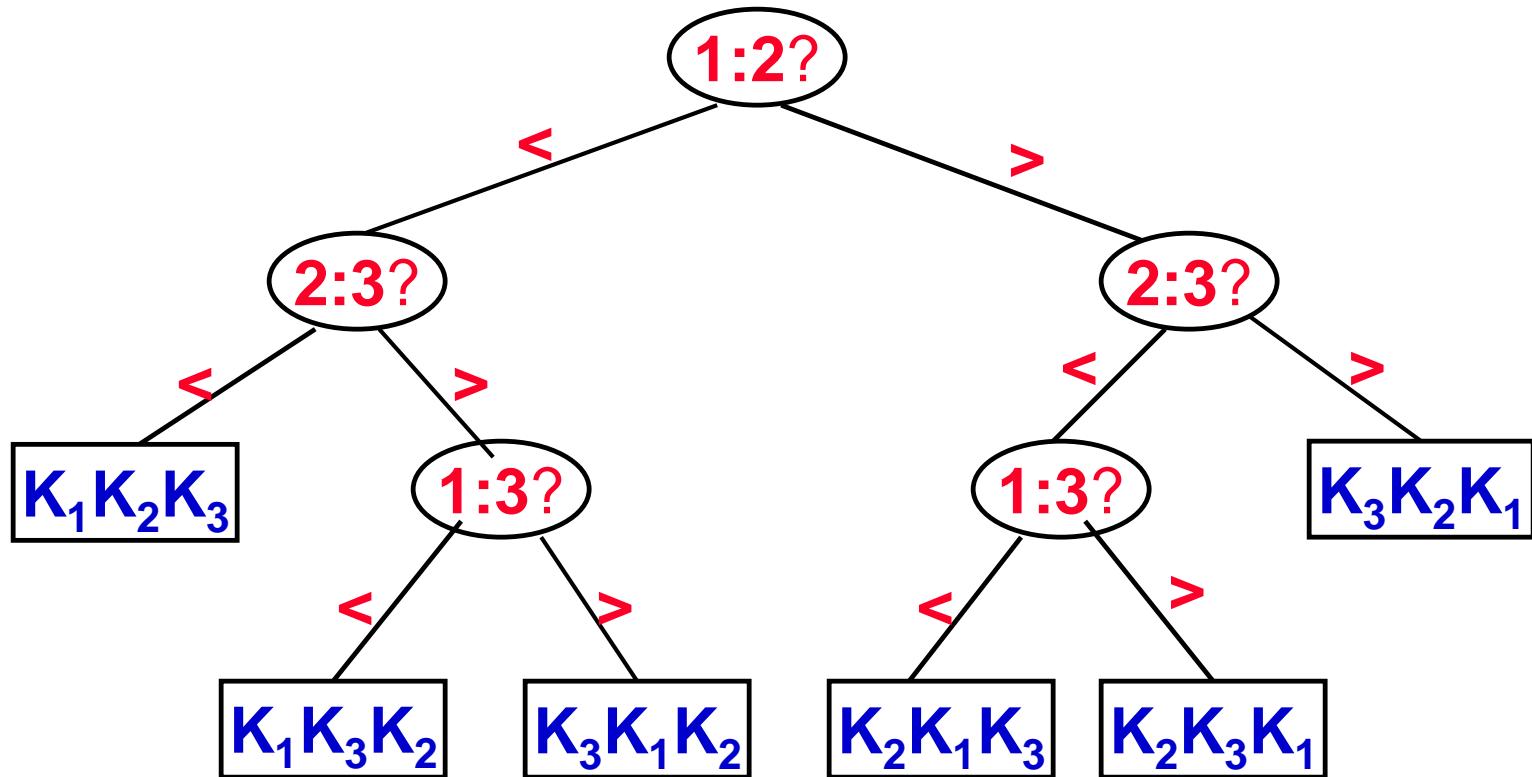
Alberi di Decisione: esempio

Siano dati tre elementi arbitrari:



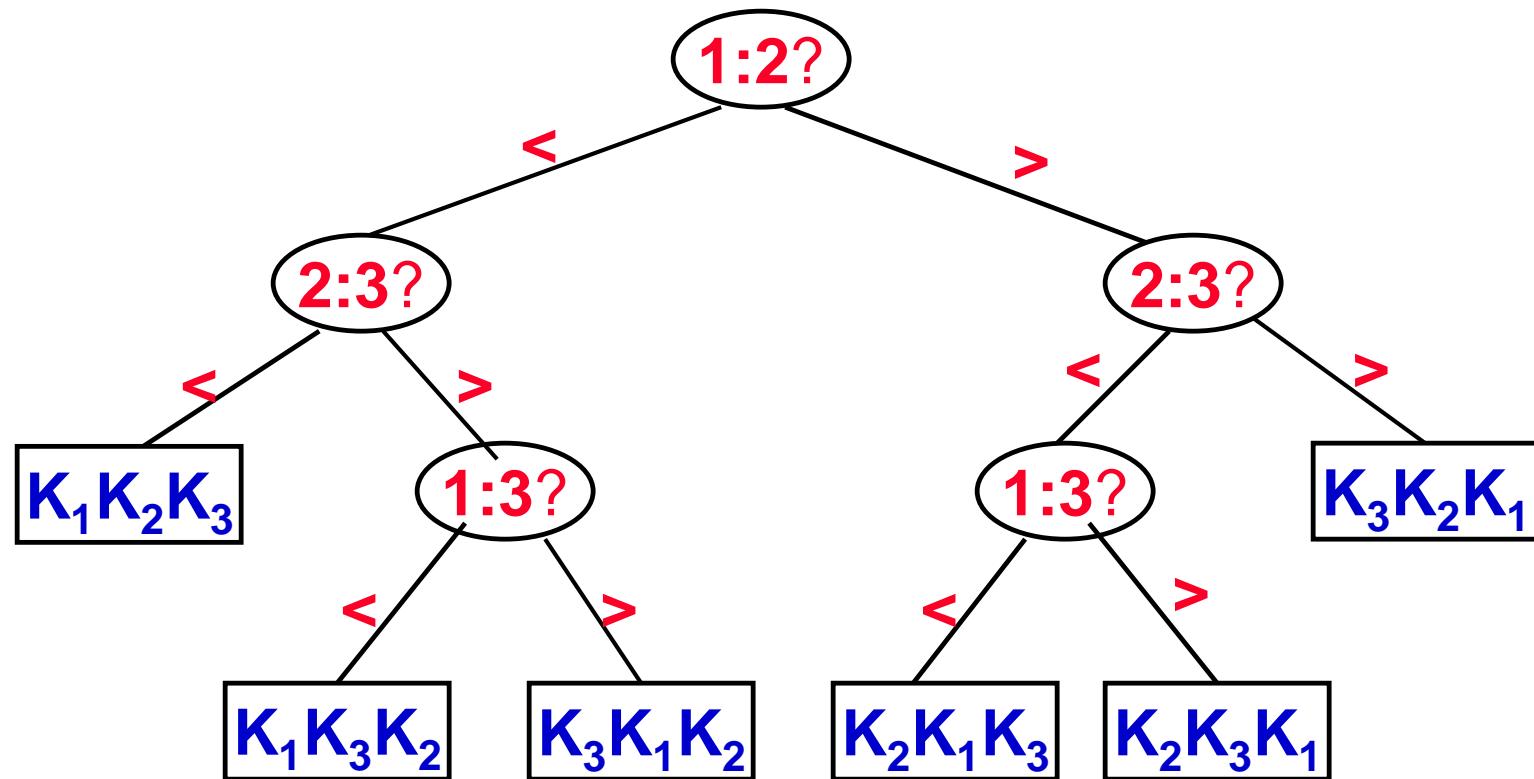
Alberi di Decisione

L'Albero di Decisione specifica la sequenza di confronti che un algoritmo deve effettuare per ordinare 3 elementi.



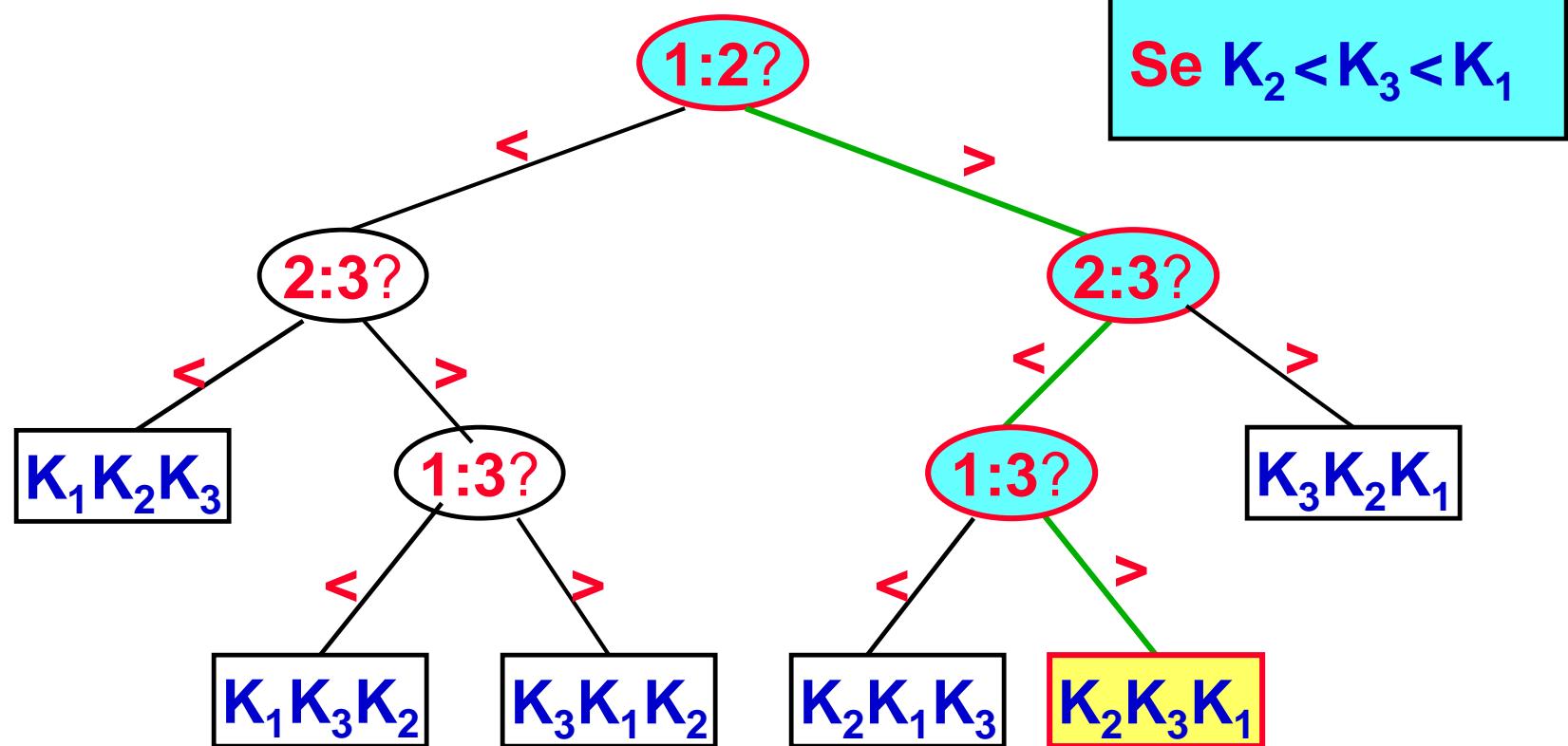
Alberi di Decisione

Una esecuzione dell'algoritmo per un dato input (di 3 elementi) corrisponde ad un percorso dalla radice ad una singola foglia.



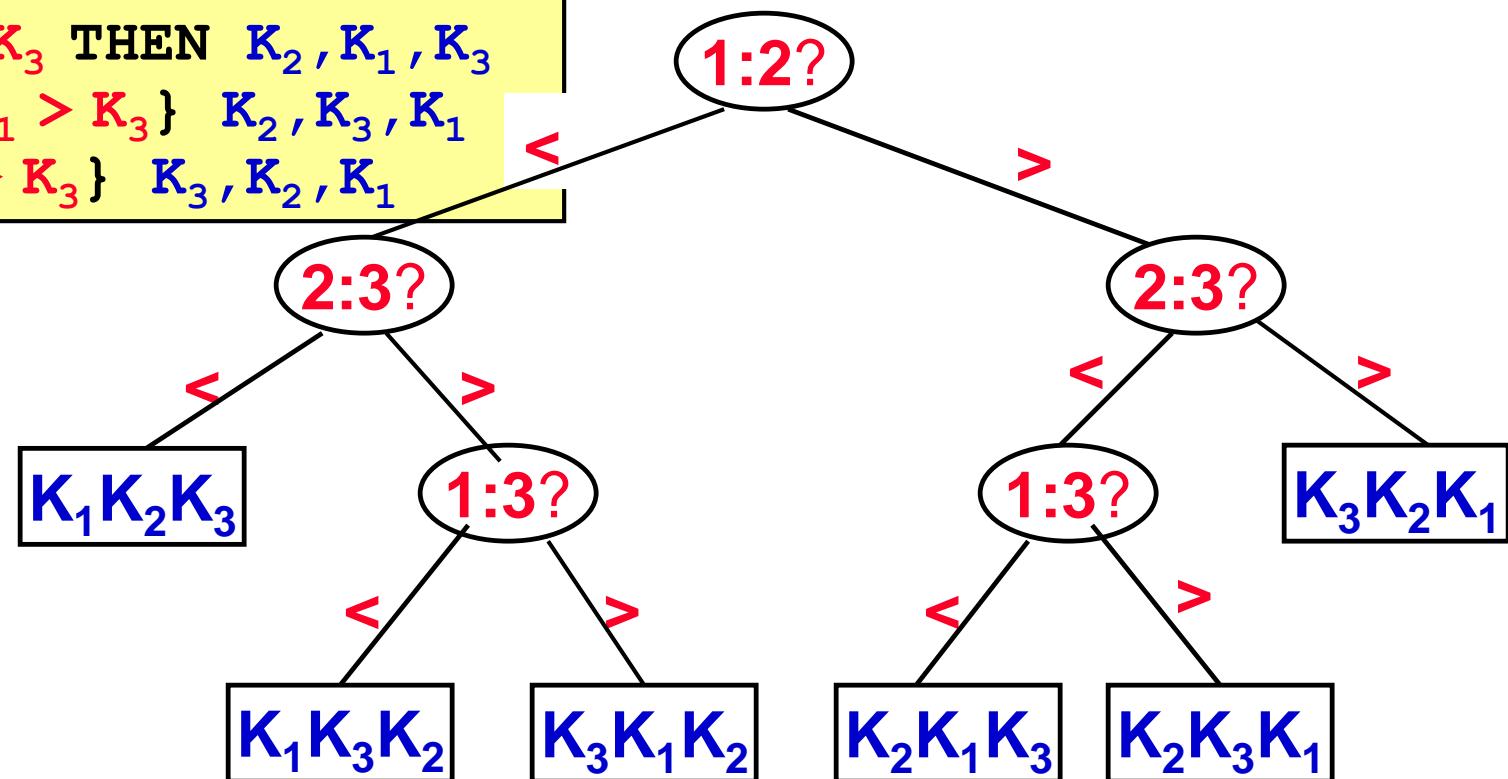
Alberi di Decisione: esempio

Un esecuzione dell'algoritmo per un dato input (di 3 elementi) corrisponde ad un percorso dalla radice ad una singola foglia.



Alberi di Decisione: algoritmo

```
IF K1 < K2 THEN
    IF K2 < K3 THEN K1,K2,K3
    ELSE {K2 > K3}
        IF K1 < K3 THEN K1,K3,K2
        ELSE {K1 > K3} K3,K1,K2
ELSE {K1 > K2}
    IF K2 < K3 THEN
        IF K1 < K3 THEN K2,K1,K3
        ELSE {K1 > K3} K2,K3,K1
    ELSE {K2 > K3} K3,K2,K1
```



Alberi di Decisione

Intuitivamente:

- *ogni foglia corrisponde ad un possibile risultato dell'ordinamento di n elementi distinti.*
- *i nodi interni corrispondono ai confronti tra gli elementi:*
 - *se il risultato è $K_i < K_j$ allora il sottoalbero sinistro del nodo “ $i:j$ ” contiene il confronto successivo*
 - *se il risultato è $K_i > K_j$ allora il sottoalbero destro del nodo “ $i:j$ ” contiene il confronto successivo*

finché non viene determinato l’ordine completo.

Alberi di Decisione

Un **albero di decisione** di ordine ***n*** è un albero binario tale che:

1. ha ***n!*** foglie, ciascuna etichettata con una diversa **permutazione** degli elementi
2. i ***nodi interni*** hanno tutti **grado 2** e sono etichettati con coppie di indici “***i:j***”, per ***i,j = 1,...,n***
3. nel percorso dalla radice alla foglia etichettata “ **$K_{i_1}, K_{i_2}, \dots, K_{i_n}$** ” compare **almeno**:
 - o un nodo “ **$i_j:i_{j+1}$** ”, e il percorso procede a sinistra del nodo;
 - o un nodo “ **$i_{j+1}:i_j$** ”, e il percorso procede a destra del nodo.

Alberi di Decisione

Notate che:

- *un albero di decisione di ordine n rappresenta tutte le possibili esecuzioni di un algoritmo di ordinamento con input di dimensione n*
- *ma, fissata la dimensione n , ad ogni algoritmo di ordinamento corrisponde un differente albero di decisione di ordine n .*

Albero di Decisione di Insert-Sort

Insert-Sort (A)

FOR j=2 to Length (A)

DO Key := A[j]

i=j-1

WHILE i>0 AND A[i]>Key

DO A[i+1]=A[i]

i=i-1

A[i+1]=Key

$j = 2$
 $i = 1$

1 2 3
3 5 6
↑ ↑

2:1?

<

>

3:1?

<

>

3:2?

<

>

K₂K₁K₃

3:2?

<

>

K₁K₃K₂

K₃K₁K₂

K₃K₂K₁

K₂K₃K₁

Albero di Decisione di Insert-Sort

Insert-Sort (A)

FOR j=2 to Length (A)

DO Key := A[j]

i=j-1

WHILE i>0 AND A[i]>Key

DO A[i+1]=A[i]

i=i-1

A[i+1]=Key

$j = 3$
 $i = 2$

1 2 3
3 5 6

1 2 3
3 5 6



2:1?

<

3:1?

<

3:2?

>

K₂K₁K₃

3:2?

<

3:1?

>

K₁K₂K₃

K₃K₂K₁

K₂K₃K₁

K₃K₁K₂

K₁K₃K₂

Albero di Decisione di Insert-Sort

Insert-Sort (A)

FOR j=2 to Length (A)

DO Key := A[j]

i=j-1

WHILE i>0 AND A[i]>Key

DO A[i+1]=A[i]

i=i-1

A[i+1]=Key

j = 3
i = 2

1 2 3
3 5 6

1 2 3
3 5 6

1 2 3
3 5 6

2:1?

<

>

3:1?

<

>

3:2?

<

>

K₂K₁K₃

K₃K₂K₁

K₂K₃K₁

3:2?

<

K₁K₂K₃

3:1?

<

>

K₃K₁K₂

K₁K₃K₂

Albero di Decisione di Insert-Sort

Insert-Sort (A)

FOR j=2 to Length (A)

DO Key := A[j]

i=j-1

WHILE i>0 AND A[i]>Key

DO A[i+1]=A[i]

i=i-1

A[i+1]=Key

$j = 2$
 $i = 1$

1 2 3
6 3 5
↑ ↑

2:1?

<

>

3:1?

<

>

3:2?

<

>

K₂K₁K₃

3:2?

<

>

K₁K₃K₂

K₃K₁K₂

K₃K₂K₁

K₂K₃K₁

Albero di Decisione di Insert-Sort

Insert-Sort (A)

FOR j=2 to Length (A)

DO Key := A[j]

i=j-1

WHILE i>0 AND A[i]>Key

DO A[i+1]=A[i]

i=i-1

A[i+1]=Key

j = 3
i = 2

1 2 3
6 3 5

1 2 3
3 6 5



2:1?

<

>

3:1?

<

>

3:2?

<

>

K₂K₁K₃

K₃K₂K₁

K₂K₃K₁

3:2?

<

K₁K₂K₃

3:1?

<

>

K₃K₁K₂

K₁K₃K₂

Albero di Decisione di Insert-Sort

Insert-Sort (A)

FOR j=2 to Length (A)

DO Key := A[j]

i=j-1

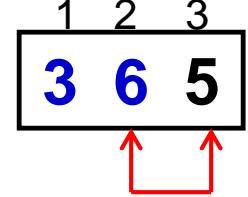
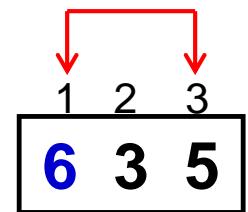
WHILE i>0 AND A[i]>Key

DO A[i+1]=A[i]

i=i-1

A[i+1]=Key

$j = 3$
 $i = 2$



2:1?

<

>

3:1?

<

>

3:2?

<

>

K₂K₁K₃

K₃K₂K₁

K₂K₃K₁

3:2?

<

K₁K₂K₃

3:1?

<

>

K₃K₁K₂

K₁K₃K₂

Albero di Decisione di Insert-Sort

Insert-Sort (A)

FOR $j=2$ to Length (A)

DO Key := $A[j]$

$i=j-1$

WHILE $i>0$ AND $A[i]>\text{Key}$

DO $A[i+1]=A[i]$

$i=i-1$

$A[i+1]=\text{Key}$

$j=3$
 $i=1$

1 2 3
6 3 5

1 2 3
3 6 5

2:1?

<

>

3:1?

<

>

3:2?

<

>

$K_2 K_1 K_3$

3:2?

<

>

3:1?

<

>

$K_1 K_2 K_3$

$K_3 K_2 K_1$

$K_2 K_3 K_1$

$K_3 K_1 K_2$

$K_1 K_3 K_2$

Albero di Decisione di Insert-Sort

Insert-Sort (A)

FOR $j=2$ to Length (A)

DO Key := $A[j]$

$i=j-1$

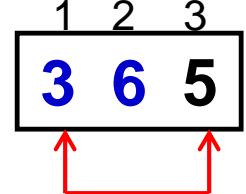
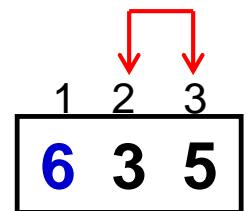
WHILE $i>0$ AND $A[i]>\text{Key}$

DO $A[i+1]=A[i]$

$i=i-1$

$A[i+1]=\text{Key}$

$j=3$
 $i=1$



2:1?

<

>

3:1?

<

>

3:2?

<

>

$K_2 K_1 K_3$

3:2?

<

>

3:1?

<

>

$K_1 K_2 K_3$

$K_3 K_2 K_1$

$K_2 K_3 K_1$

$K_3 K_1 K_2$

$K_1 K_3 K_2$

Albero di Decisione di Insert-Sort

Insert-Sort (A)

FOR j=2 to Length (A)

DO Key := A[j]

i=j-1

WHILE i>0 AND A[i]>Key

DO A[i+1]=A[i]

i=i-1

A[i+1]=Key

$j = 3$
 $i = 0$

1 2 3
6 3 5

1 2 3
3 6 5

1 2 3
3 5 6

2:1?

<

>

3:1?

<

>

3:2?

<

>

K₂K₁K₃

K₃K₂K₁

K₂K₃K₁

3:2?

<

>

3:1?

<

>

K₃K₁K₂

K₁K₃K₂

Alberi di Decisione

É importante quindi capire che:

- *un nodo etichettato “ $i:j$ ” nell’albero di decisione specifica un confronto tra gli elementi K_i e K_j secondo la loro posizione nell’array iniziale*
- e **NON** gli elementi che ad un certo punto dell’esecuzione compaiono nelle posizioni *i-esima* e *j-esima* dell’array
- *gli alberi di decisione non menzionano alcuno spostamento degli elementi*

Limite Inferiore per il Caso Peggio

Teorema: Il numero minimo di confronti che un algoritmo di ordinamento deve effettuare è $\Omega(n \log n)$ nel caso peggiore

- *Intuitivamente, il numero massimo di confronti che deve essere effettuato da un algoritmo di ordinamento in una sua esecuzione sarà pari all'altezza del suo albero di decisione.*
- *Il migliore algoritmo di decisione possibile, sarà quello il cui albero di decisione ha altezza minima tra tutti gli alberi di decisione possibili.*

Limite Inferiore per il Caso Peggio

Lemma: *Ogni albero di decisione che ordina n elementi ha altezza $\Omega(n \log n)$*

Sia T un albero di decisione di altezza h che ordina n elementi.

Ci sono $n!$ possibili permutazioni di n elementi, ognuna delle quali è un possibile ordinamento.

L'albero di decisione avrà quindi $n!$ foglie.

Limite Inferiore per il Caso Peggio

Lemma: *Ogni albero di decisione che ordina n elementi ha altezza $\Omega(n \log n)$*

Sia T un albero di decisione di altezza h che ordina n elementi.

L'albero di decisione avrà quindi $n!$ foglie.

Ma ogni albero binario di altezza h non può avere più di 2^h foglie (perché?).

Quindi deve essere: $n! \leq 2^h$

Limite Inferiore per il Caso Peggio

Lemma: *Ogni albero di decisione che ordina n elementi ha altezza $\Omega(n \log n)$*

Sia T un albero di decisione di altezza h che ordina n elementi.

Quindi deve essere: $n! \leq 2^h$

Prendendo il logaritmo di entrambi i membri, poiché entrambi sono funzioni crescenti monotone, otteniamo:

$$\log n! \leq h$$

Limite Inferiore per il Caso Peggio

Lemma: *Ogni albero di decisione che ordina **n** elementi ha altezza $\Omega(n \log n)$*

Sia T un albero di decisione di altezza h che ordina **n elementi.**

Quindi deve essere: $\log n! \leq h$

Per l'approssimazione di Stirling abbiamo che:

$$n! > \left(\frac{n}{e}\right)^n$$

e = 2.71828...

Limite Inferiore per il Caso Peggio

Lemma: *Ogni albero di decisione che ordina n elementi ha altezza $\Omega(n \log n)$*

Sia T un albero di decisione di altezza h che ordina n elementi.

Quindi deve essere: $\log n! \leq h$

Otteniamo che

$$n! > \left(\frac{n}{e}\right)^n$$

$$h \geq \log\left(\frac{n}{e}\right)^n$$

Limite Inferiore per il Caso Peggio

Lemma: *Ogni albero di decisione che ordina n elementi ha altezza $\Omega(n \log n)$*

Sia T un albero di decisione di altezza h che ordina n elementi.

Otteniamo che

$$n! > \left(\frac{n}{e}\right)^n$$

$$h \geq \log\left(\frac{n}{e}\right)^n$$

$$\geq n \log \frac{n}{e}$$

Limite Inferiore per il Caso Peggio

Lemma: *Ogni albero di decisione che ordina n elementi ha altezza $\Omega(n \log n)$*

Sia T un albero di decisione di altezza h che ordina n elementi.

Otteniamo che

$$n! > \left(\frac{n}{e}\right)^n$$

$$h \geq \log\left(\frac{n}{e}\right)^n$$

$$\geq n \log \frac{n}{e}$$

$$\geq n \log n - n \log e$$

Limite Inferiore per il Caso Peggio

Lemma: *Ogni albero di decisione che ordina n elementi ha altezza $\Omega(n \log n)$*

Sia T un albero di decisione di altezza h che ordina n elementi.

Otteniamo che

$$n! > \left(\frac{n}{e}\right)^n$$

$$h \geq \log\left(\frac{n}{e}\right)^n$$

$$\geq n \log \frac{n}{e}$$

$$\geq n \log n - n \log e$$

$$= \Omega(n \log n)$$

Limite Inferiore per il Caso Peggio

Corollario: *HeapSort e MergeSort sono algoritmi di ordinamento per confronto asintoticamente ottimi nel caso peggiore.*

Limite Inferiore per il Caso Peggio

Corollario: *HeapSort e MergeSort sono algoritmi di ordinamento per confronto asintoticamente ottimi nel caso peggiore.*

Abbiamo già calcolato che il limite superiore del tempo di esecuzione nel caso peggiore di entrambi gli algoritmi è $O(n \log n)$.

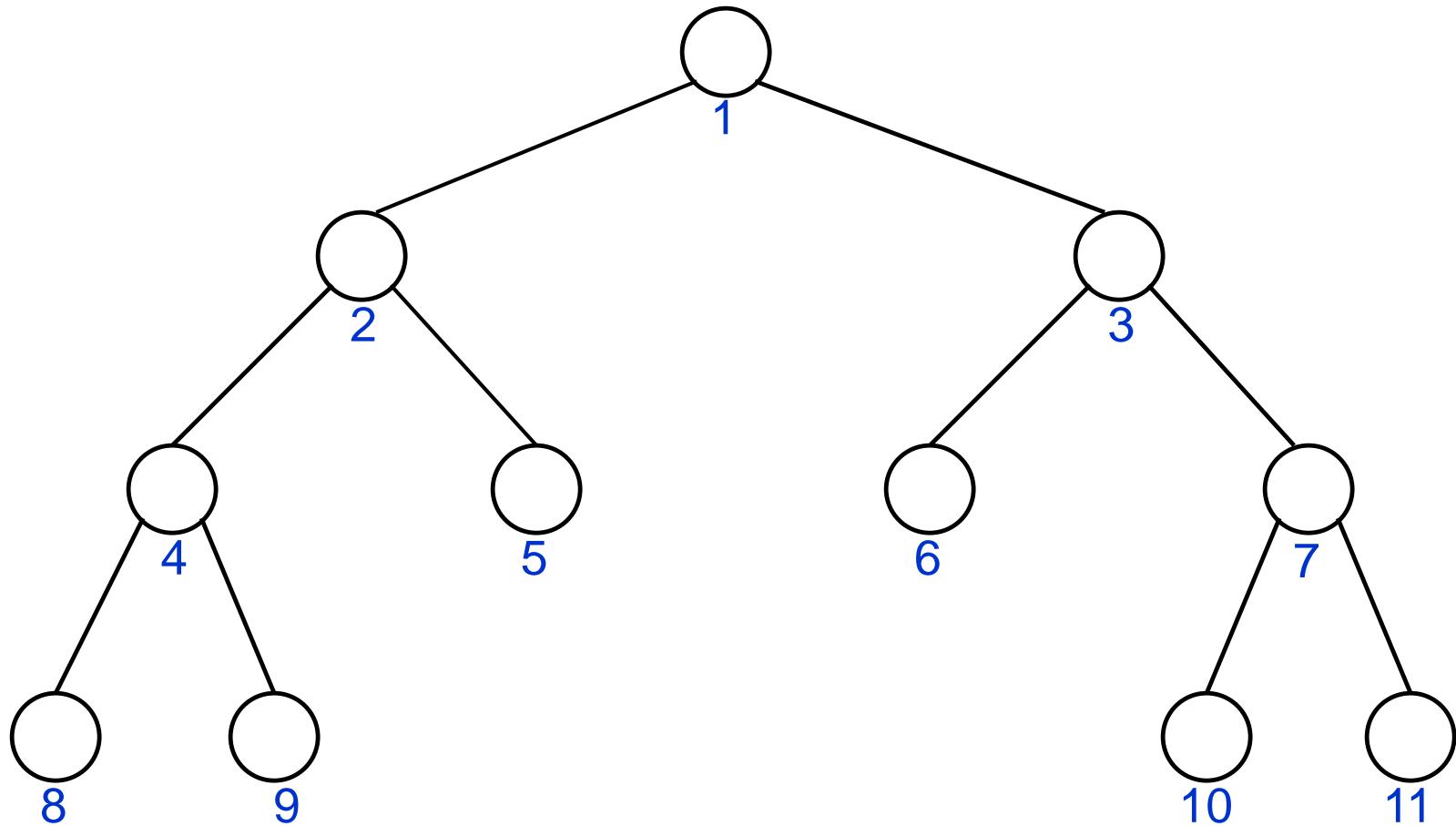
Ma questo limite corrisponde esattamente a limite inferiore $\Omega(n \log n)$ appena calcolato per il caso peggiore.

Da queste due osservazioni segue il corollario!

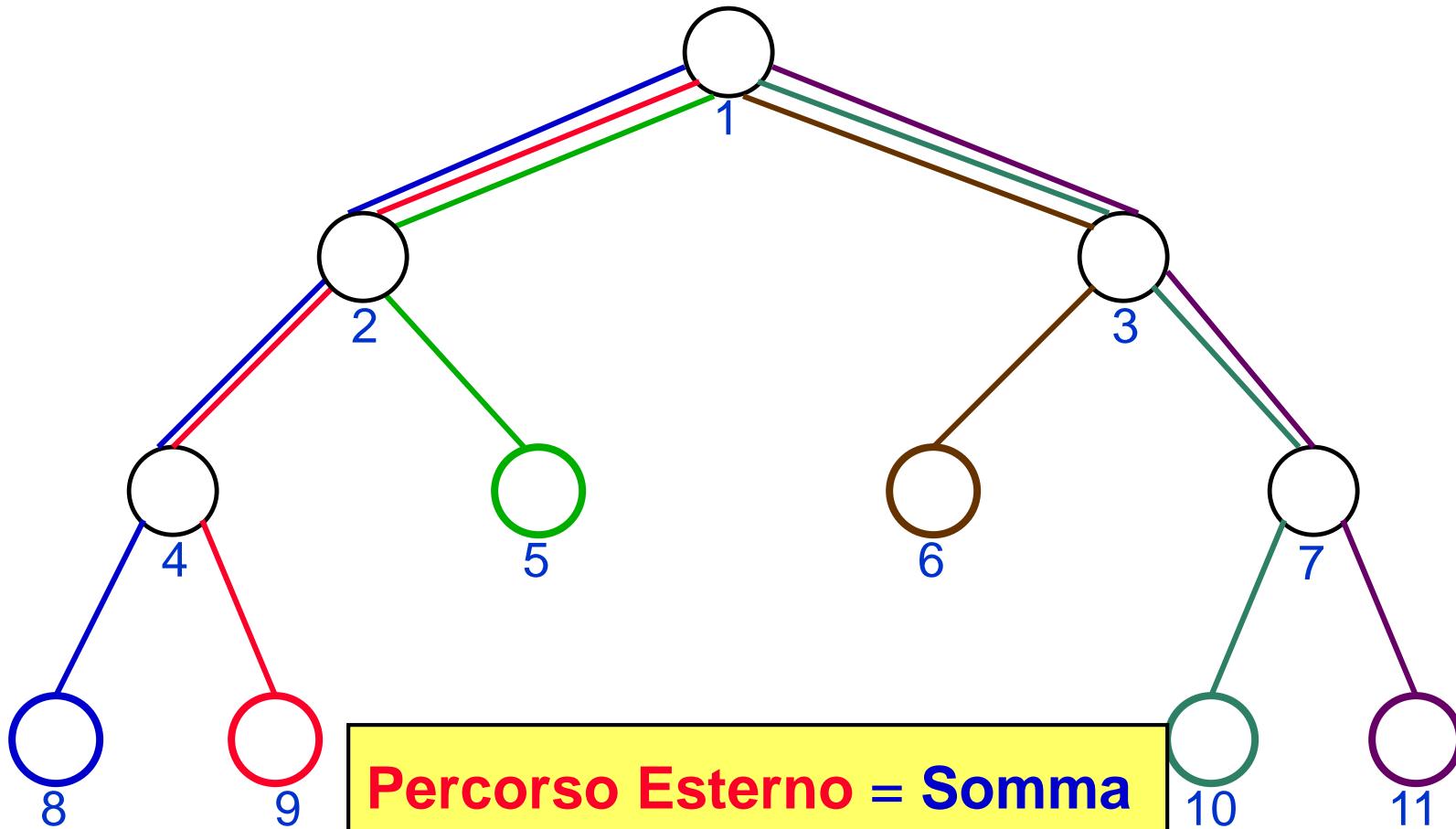
Limite Inferiore per il Caso Medio

Teorema: Il numero minimo di confronti che un algoritmo di ordinamento deve effettuare è $\Omega(n \log n)$ nel caso medio

Percorso Esterno di un Albero Binario

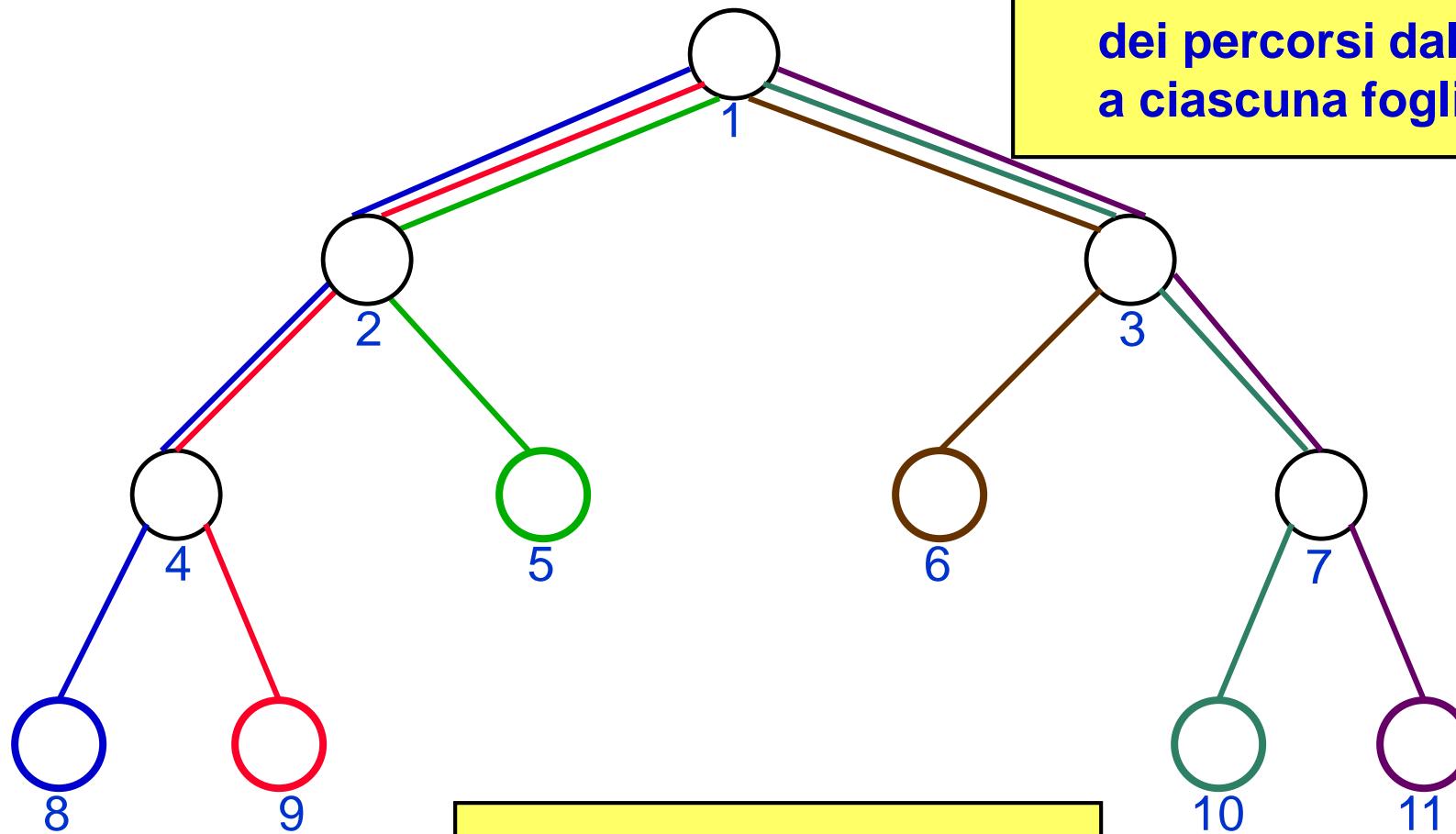


Percorso Esterno di un Albero Binario



**Percorso Esterno = Somma
dei percorsi dalla radice
a ciascuna foglia**

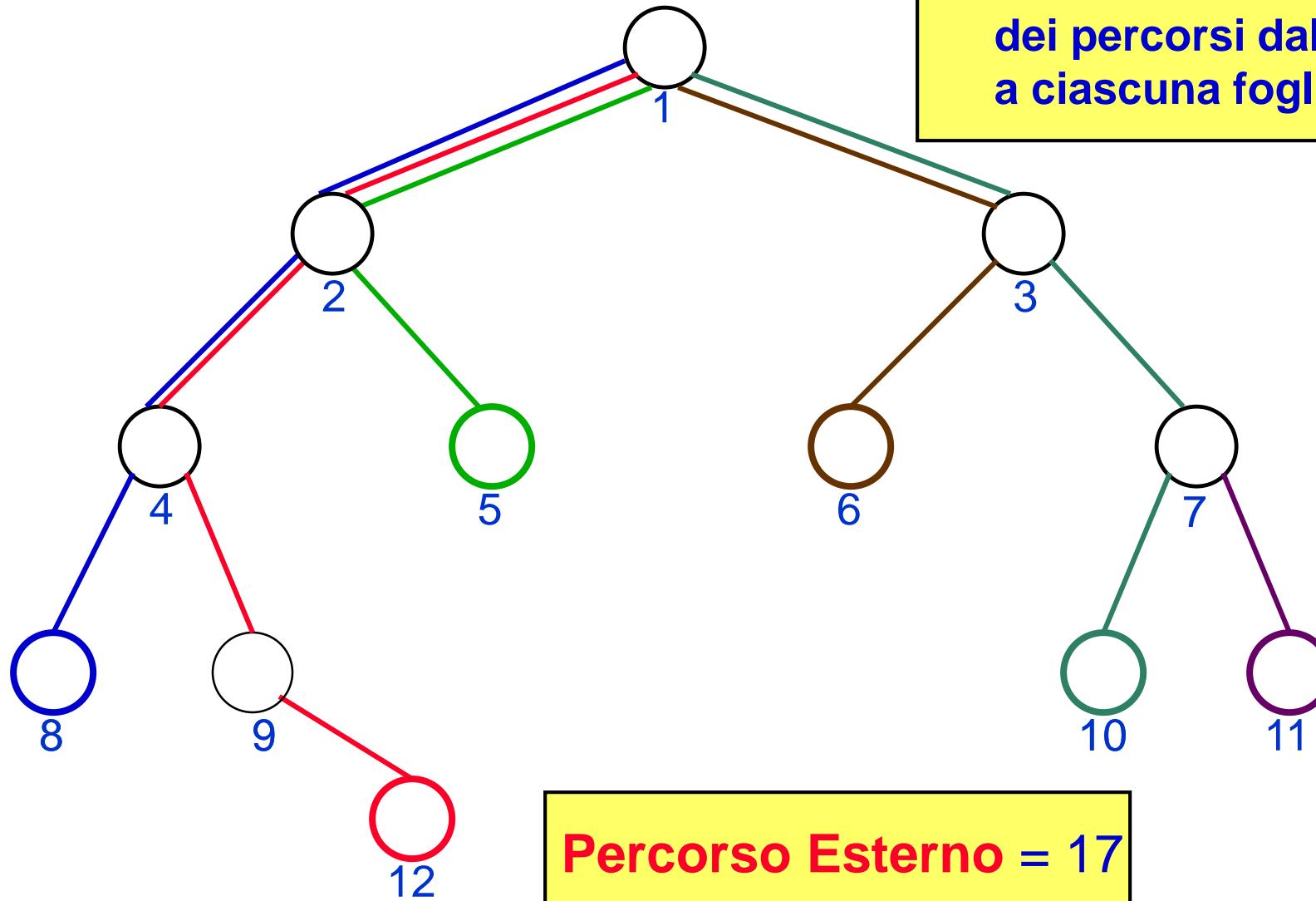
Percorso Esterno di un Albero Binario



Percorso Estero = Somma
dei percorsi dalla radice
a ciascuna foglia

Percorso Esterno = 16

Percorso Esterno di un Albero Binario



Percorso Estero = Somma
dei percorsi dalla radice
a ciascuna foglia

Percorso Esterno = 17

Limite Inferiore per il Caso Medio

Teorema: Il numero minimo di confronti che un algoritmo di ordinamento deve effettuare è $\Omega(n \log n)$ nel caso medio

- Assumiamo che ogni permutazione iniziale di elementi in input abbia uguale probabilità.
- Consideriamo l'albero di decisione di ordine n associato ad un algoritmo di ordinamento
- Il numero medio di confronti necessari ad un algoritmo di ordinamento è quindi pari alla lunghezza del percorso esterno diviso per il numero di foglie dell'albero.

Limite Inferiore per il Caso Medio

Teorema: Il numero minimo di confronti che un algoritmo di ordinamento deve effettuare è $\Omega(n \log n)$ nel caso medio

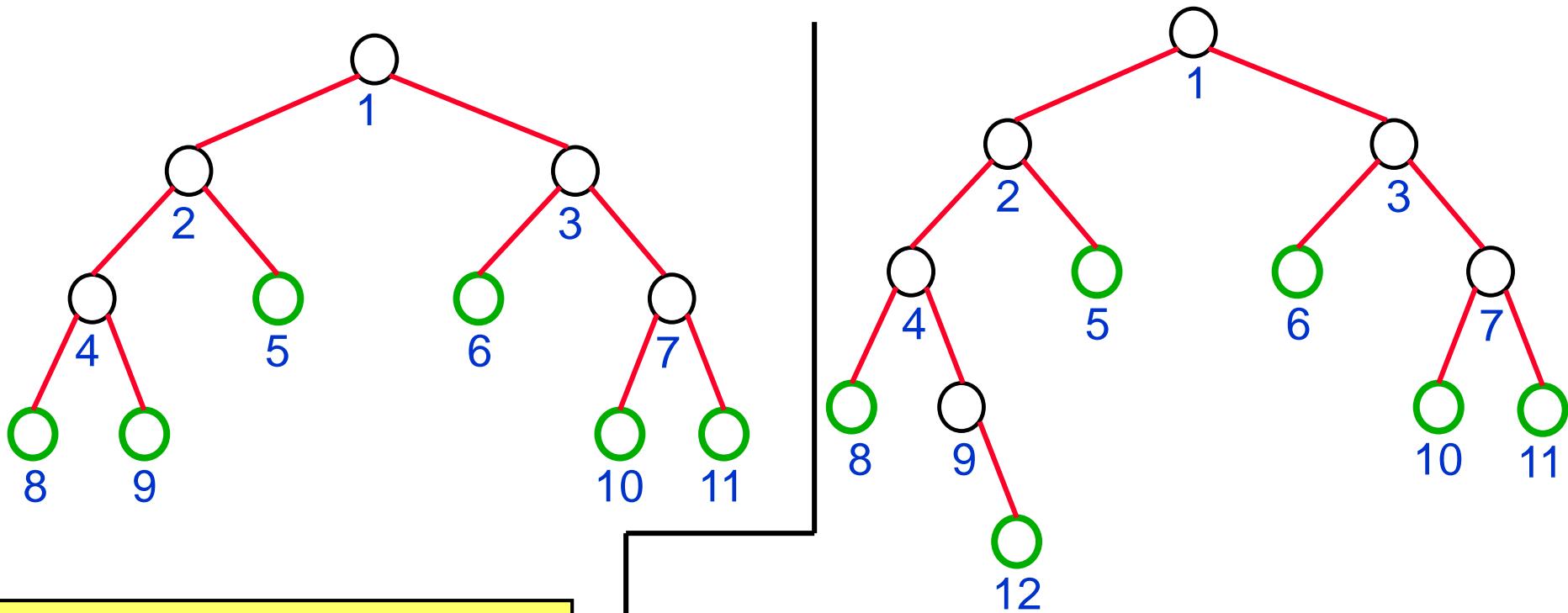
Il numero medio di confronti è pari alla lunghezza del percorso esterno diviso per il numero di foglie dell'albero.

Per ottenere il limite inferiore nel caso medio ci serve calcolare il minimo numero medio di confronti.

In altre parole, calcolare il rapporto tra la lunghezza percorso esterno e il numero di foglie dell'albero di decisione con la minima lunghezza del percorso esterno.

Percorso Esterno Minimo

FATTO: L'albero che minimizza il percorso esterno è quello in cui tutte le n foglie occorrono al più sui due livelli h e $h-1$, per qualche h .



Percorso Estero = k

Percorso Estero = $k-h+(h+1)=k+1$

Limite Inferiore per il Caso Medio

Teorema: Il numero minimo di confronti che un algoritmo di ordinamento deve effettuare è $\Omega(n \log n)$ nel caso medio

Il minimo numero medio di confronti è pari alla lunghezza del percorso esterno diviso per il numero di foglie dell'albero.

FATTO: L'albero che minimizza il percorso esterno è quello in cui tutte le n foglie occorrono al più sui due livelli h e $h - 1$, per qualche h .

Siano N_h e N_{h-1} il numero di foglie ai livelli h e $h - 1$

Limite Inferiore per il Caso Medio

Teorema: Il numero minimo di confronti che un algoritmo di ordinamento deve effettuare è $\Omega(n \log n)$ nel caso medio

Il minimo numero medio di confronti è pari alla lunghezza del percorso esterno diviso per il numero di foglie dell'albero.

Siano N_h e N_{h-1} il numero di foglie ai livelli h e $h-1$

Il numero medio di confronti nell'albero sarà quindi

$$C_n = [(h-1)N_{h-1} + hN_h] / n!$$

Limite Inferiore per il Caso Medio

Teorema: Il numero minimo di confronti che un algoritmo di ordinamento deve effettuare è $\Omega(n \log n)$ nel caso medio

Siano N_h e N_{h-1} il numero di foglie ai livelli h e $h-1$

Il numero medio di confronti nell'albero sarà quindi

$$C_n = [(h-1)N_{h-1} + hN_h] / n!$$

Ma sappiamo anche che

$$N_{h-1} + N_h = n!$$

e

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

Limite Inferiore per il Caso Medio

Teorema: Il numero minimo di confronti che un algoritmo di ordinamento deve effettuare è $\Omega(n \log n)$ nel caso medio

Siano N_h e N_{h-1} il numero di foglie ai livelli h e $h-1$

Il numero medio di confronti quindi

$$C_n = [(h-1)N_{h-1} +$$

Poichè un albero pieno alto h ha 2^h foglie e ogni nodo interno ha grado due (cioè ha 2 figli)

Ma sappiamo anche che

$$N_{h-1} + N_h = n!$$

e

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

Limite Inferiore per il Caso Medio

Teorema: Il numero minimo di confronti che un algoritmo di ordinamento deve effettuare è $\Omega(n \log n)$ nel caso medio

Siano N_h e N_{h-1} il numero di foglie ai livelli h e $h-1$

Il numero medio di confronti nell'albero sarà quindi

$$C_n = [(h-1)N_{h-1} + hN_h] / n!$$

Quindi:

$$N_h = 2n! - 2^h$$

$$N_{h-1} = 2^h - n!$$

$$N_{h-1} + N_h = n!$$

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

Limite Inferiore per il Caso Medio

Teorema: Il numero minimo di confronti che un algoritmo di ordinamento deve effettuare è $\Omega(n \log n)$ nel caso medio

Il numero medio di confronti nell'albero sarà quindi

$$C_n = [(h-1)N_{h-1} + hN_h] / n!$$

Quindi:

$$N_h = 2n! - 2^h$$

$$N_{h-1} = 2^h - n!$$

Sostituendo:

$$C_n = (h n! + n! - 2^h)/n!$$

$$N_{h-1} + N_h = n!$$

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

Limite Inferiore per il Caso Medio

Teorema: Il numero minimo di confronti che un algoritmo di ordinamento deve effettuare è $\Omega(n \log n)$ nel caso medio

Sostituendo: $C_n = (hn! + n! - 2^h)/n!$

Ma $h = \lceil \log n! \rceil = \log n! + \varepsilon$ per $0 \leq \varepsilon < 1$ quindi

$$C_n = (n! \log n! + n! \varepsilon + n! - n! 2^\varepsilon)/n!$$

Limite Inferiore per il Caso Medio

Teorema: Il numero minimo di confronti che un algoritmo di ordinamento deve effettuare è $\Omega(n \log n)$ nel caso medio

Sostituendo: $C_n = (h n! + n! - 2^h)/n!$

Ma $h = \lceil \log n! \rceil = \log n! + \varepsilon$ per $0 \leq \varepsilon < 1$ quindi

$$\begin{aligned} C_n &= (n! \log n! + n! \varepsilon + n! - n! 2^\varepsilon)/n! \\ &= \log n! + (1 + \varepsilon - 2^\varepsilon) \end{aligned}$$

Limite Inferiore per il Caso Medio

Teorema: Il numero minimo di confronti che un algoritmo di ordinamento deve effettuare è $\Omega(n \log n)$ nel caso medio

Sostituendo: $C_n = (h n! + n! - 2^h)/n!$

Ma $h = \lceil \log n! \rceil = \log n! + \varepsilon$ per $0 \leq \varepsilon < 1$ quindi

$$\begin{aligned} C_n &= (n! \log n! + n! \varepsilon + n! - n! 2^\varepsilon)/n! = \\ &= \log n! + (1 + \varepsilon - 2^\varepsilon) \geq \\ &\geq \log n! = n \log n - n \log e \\ &= \Omega(n \log n) \end{aligned}$$

Limite Inferiore per il Caso Medio

Corollario: *HeapSort e MergeSort sono algoritmi di ordinamento per confronto asintoticamente ottimi.*

*Abbiamo già calcolato che il limite superiore del tempo di esecuzione **medio** di entrambi gli algoritmi è $O(n \log n)$.*

Ma questo limite corrisponde esattamente a limite inferiore $\Omega(n \log n)$ appena calcolato per il caso medio.

Da queste due osservazioni segue il corollario!

Algoritmi e Strutture Dati

Strutture Dati Elementari

Insiemi

- Un insieme è una *collezione di oggetti* distinguibili chiamati *elementi* (o membri) dell'insieme.
- $a \in S$ significa che a è *un membro* de (o appartiene a) *l'insieme S*
- $b \notin S$ significa che b **NON** è un *membro* de (o NON appartiene a) *l'insieme S*
- *Esempi:*
 - \mathbb{N} denota l'insieme dei *numeri naturali*
 - \mathbb{R} denota l'insieme dei *numeri reali*
 - \emptyset denota l'*insieme vuoto*

Insiemi Dinamici

- Gli *algoritmi* manipolano *collezioni di dati* come insiemi di elementi
- Gli insiemi rappresentati e manipolati da algoritmi in generale cambiano nel tempo:
 - *crescono in dimensione* (cioè nel numero di elementi che contengono)
 - *diminuiscono in dimensione*
 - *la collezione di elementi che contengono può mutare nel tempo*

Per questo vengono chiamati Insiemi Dinamici

Insiemi Dinamici

*Spesso gli **elementi** di un insieme dinamico sono oggetti strutturati che contengono*

- *una “**chiave**” identificativa **k** dell’elemento all’interno dell’insieme*
 - *altri “**dati satellite**”, contenuti in opportuni campi di cui sono costituiti gli elementi dell’insieme*
- I **dati satellite** non vengono in genere direttamente usati per implementare le operazioni sull’insieme.*

Operazioni su Insiemi Dinamici

Esempi di operazioni su insiemi dinamici

➤ *Operazioni di Ricerca:*

- *Ricerca(S, k):*
- *Minimo(S):*
- *Massimo(S):*
- *Successore(S, x):*
- *Predecessore(S, x):*

Operazioni su Insiemi Dinamici

Esempi di operazioni su insiemi dinamici

➤ *Operazioni di Modifica:*

- $S = \text{Inserimento}(S, x)$:
- $S = \text{Cancellazione}(S, x)$:

Stack

Uno Stack è un insieme dinamico in cui l'elemento rimosso dall'operazione di cancellazione è predeterminato.

In uno Stack questo elemento è l'ultimo elemento inserito.

Uno Stack può essere visto come una lista di tipo “last in, first out” (LIFO)

- **Nuovi elementi vengono inseriti in testa e prelevati dalla testa**

Operazioni su Stack

Due Operazioni di Modifica:

Inserimento: Push (S, x)

- **aggiunge un elemento in cima allo Stack**

Cancellazione: Pop (S)

- **rimuove un elemento dalla cima dello Stack**

Altre operazioni: Stack-Vuoto (S)

- **verifica se lo Stack è vuoto (ritorna True o False)**

Operazioni su Stack

Due Operazioni di Modifica:

Inserimento: $\text{Push}(S, x)$

- aggiunge un elemento *in cima allo Stack*

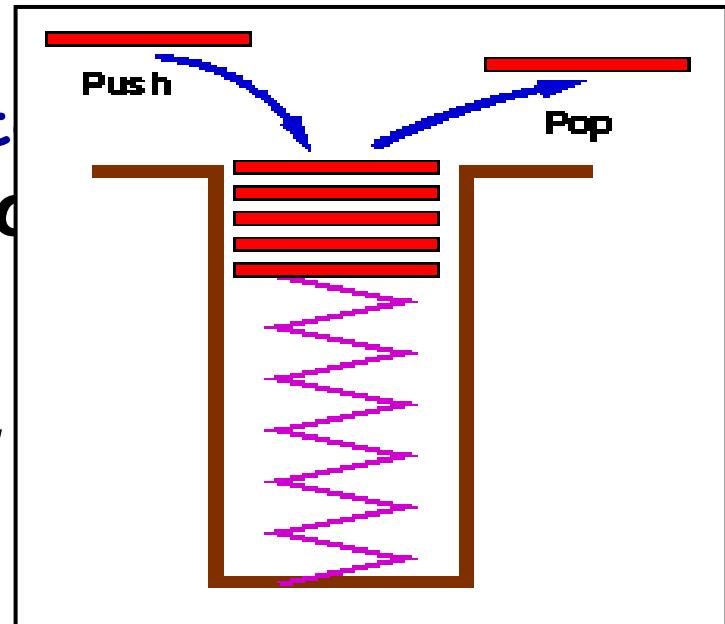
Cancellazione: $\text{Pop}(S)$

- rimuove un elemento *dalla cima dello Stack*

Altre operazioni: Stack-Vuoto

- verifica se lo Stack è vuoto (*True* o *False*)

Uno Stack può essere immaginato come una pila di piatti!



Operazioni su Stack

Algoritmo Stack-Vuoto (S)

```
IF ( $top\_S = 0$ )
THEN return TRUE
ELSE return FALSE
```

top_S : un intero che rappresenta, in ogni istante, il numero di elementi presenti nello Stack

Operazioni su Stack

Algoritmo Stack-Vuoto (S)

```
IF ( $top\_S = 0$ )
THEN return TRUE
ELSE return FALSE
```

Algoritmo Push (S, x)

```
 $top\_S = top\_S + 1$ 
 $S[top\_S] = x$ 
```

Assumiamo qui che l'operazione di **aggiunta** di un **elemento** nello Stack **S** sia realizzata come **l'aggiunta** di un **elemento ad un array**

Operazioni su Stack

- **Problema:**
 - Che succede se eseguiamo un operazione di pop (estrazione) di un elemento quando lo Stack è vuoto?
 - Questo è chiamato **Stack Underflow**. È necessario implementare l'operazione di pop con un meccanismo per verificare se questo è il caso.

Operazioni su Stack

Algoritmo Stack-Vuoto (S)

```
IF  $top\_S = 0$ 
    THEN return TRUE
ELSE return FALSE
```

Algoritmo Push (S, x)

```
 $top\_S = top\_S + 1$ 
 $S[top\_S] = x$ 
```

Algoritmo Pop (S)

```
IF Stack-Vuoto ( $S$ )
    THEN ERROR "underflow"
ELSE  $top\_S = top\_S - 1$ 
        return  $S[top\_S + 1]$ 
```

Stack: implementazione

- **Problema:**
 - Che succede se eseguiamo un operazione di push (inserimento) di un elemento quando lo Stack è pieno?
 - Questo è chiamato **Stack Overflow**. È necessario implementare l'operazione di push con un meccanismo per verificare se questo è il caso. (**SEMPLICE ESERCIZIO**)

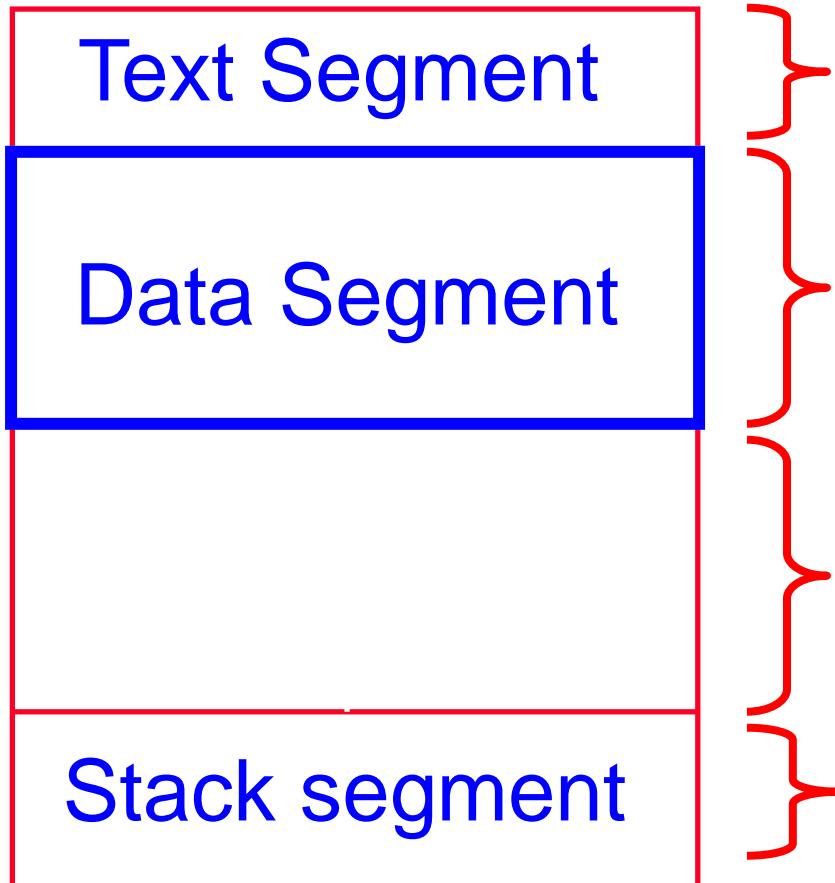
Stack: implementazione

- **Arrays**
 - Permettono di implementare stack in modo semplice
 - Flessibilità limitata, *ma incontra parecchi casi di utilizzo*
 - La capacità dello Stack è tipicamente limitata ad una quantità costante:
 - dalla dimensione dell'array utilizzato
 - in generale dalla memoria disponibile del computer.
- Possibile implementarlo con **Liste Puntate**.

Stack: applicazione

- Stack è molto frequente in Informatica:
 - Elemento chiave nel meccanismo che implementa la chiamata/ritorno di funzioni/procedure
 - *Record di attivazione* permettono la ricorsione.
 - Chiamata: *push* di un record di attivazione
 - Return: *pop* di un record di attivazione

Gestione della memoria dei processi



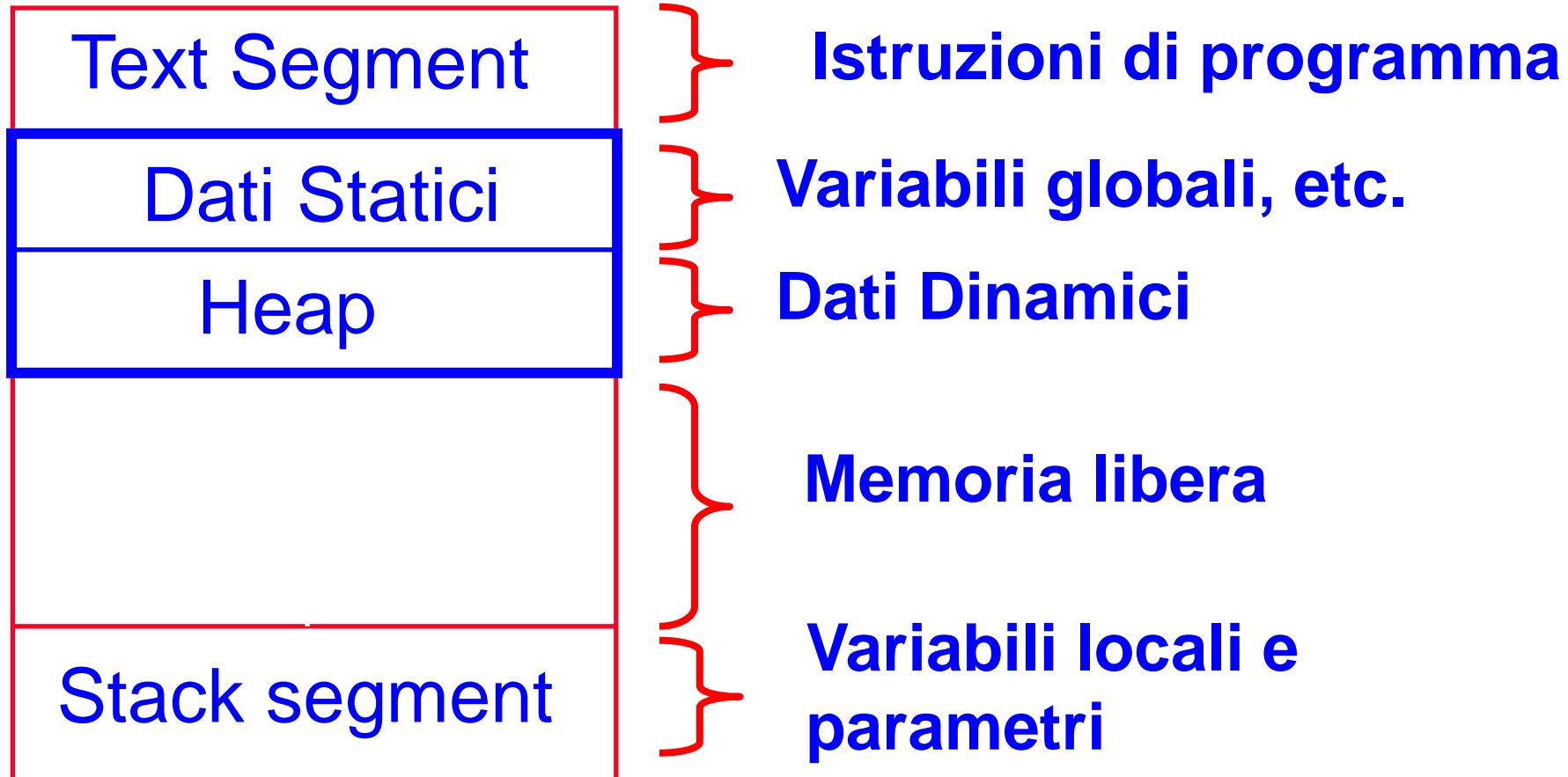
Istruzioni di programma

Dati statici e dinamici

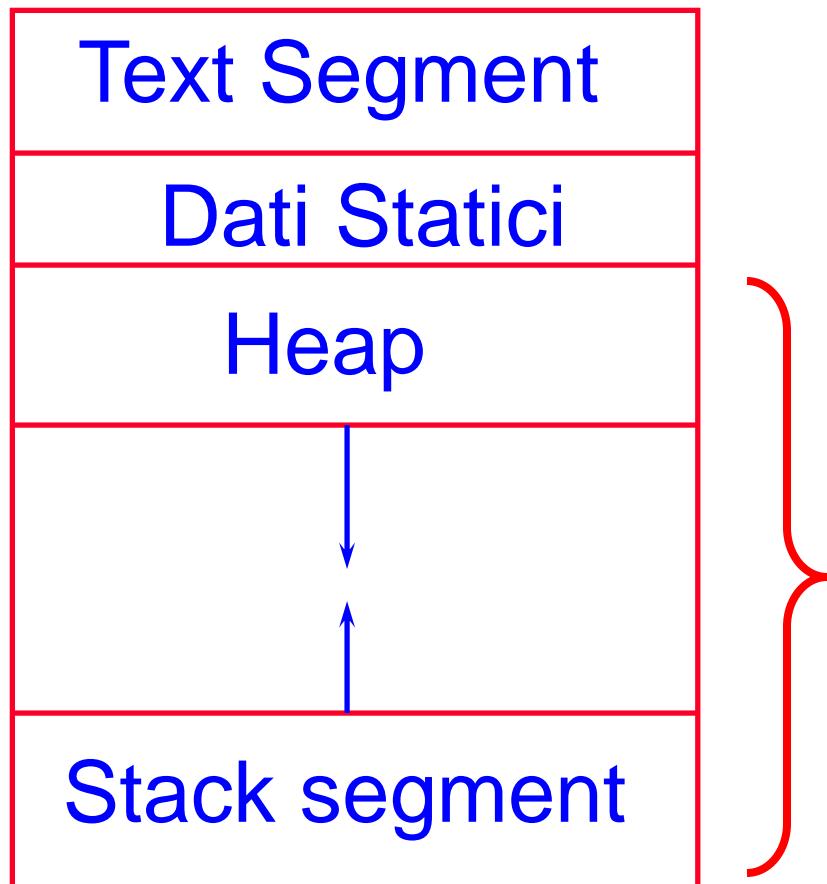
Memoria libera

Variabili locali e parametri

Gestione della memoria dei processi



Gestione della memoria dei processi



**La memoria è allocata
e deallocata secondo
necessità**

Stack: applicazioni

- Stacks è molto frequente:
 - Elemento chiave nel meccanismo che implementa la chiamata/return a funzioni/procedure
 - *Record di attivazione* permettono la ricorsione.
 - Chiamata: *push* di un record di attivazione
 - Return: *pop* di un record di attivazione
- Record di Attivazione contiene
 - Argomenti (parametri formali) della funzione
 - Indirizzo di ritorno
 - Valore di ritorno della funzione
 - Variabili locali della funzione

Stack di Record di Attivazione in LP

Programma

```
function f(int x,int y)
{
    int a;
    if ( term_cond )
        return ...;
    a = ...;
    return g( a );
}
```

```
function g( int z )
{
    int p, q;
    p = ... ;
    q = ... ;
    return f(p,q) ;
}
```

Stack di Record di Attivazione in LP

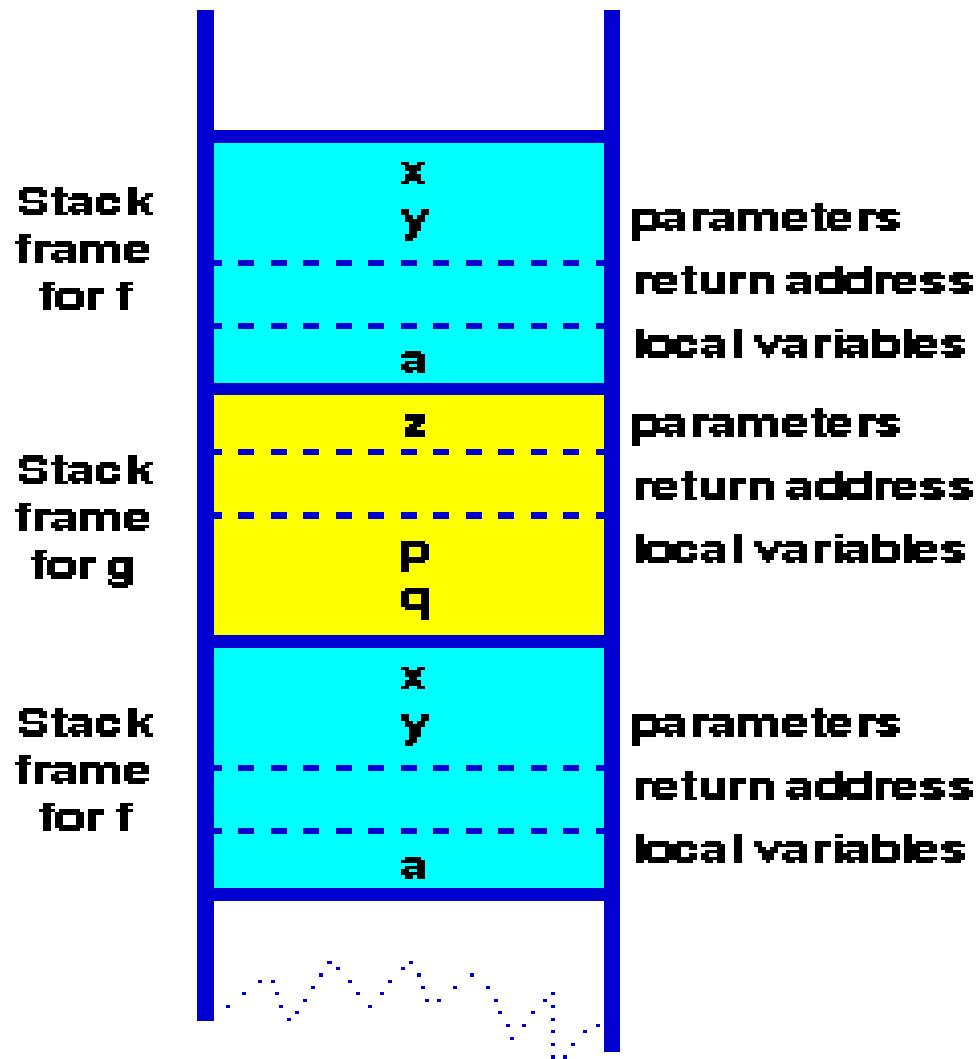
Programma

```
function f(int x,int y)
```

```
{  
    int a;  
    if ( term_cond )  
        return ...;  
    a = ...;  
    return g( a );  
}
```

```
function g( int z )
```

```
{  
    int p, q;  
    p = ... ;  
    q = ... ;  
    return f(p,q);  
}
```



Stack di Record di Attivazione in LP

Programma

```
function f(int x,int y)
```

```
{  
    int a;  
    if ( term_cond )  
        return ...;  
    a = ...;  
    return g( a );  
}
```

```
function g( int z )
```

```
{  
    int p, q;  
    p = ... ;  
    q = ... ;  
    return f(p,q);  
}
```

Stack
frame
for f

Stack
frame
for g

Stack
frame
for f

parameters
return address
local variables

parameters
return address
local variables

parameters
return address
local variables

Contesto
di esecuzione di f

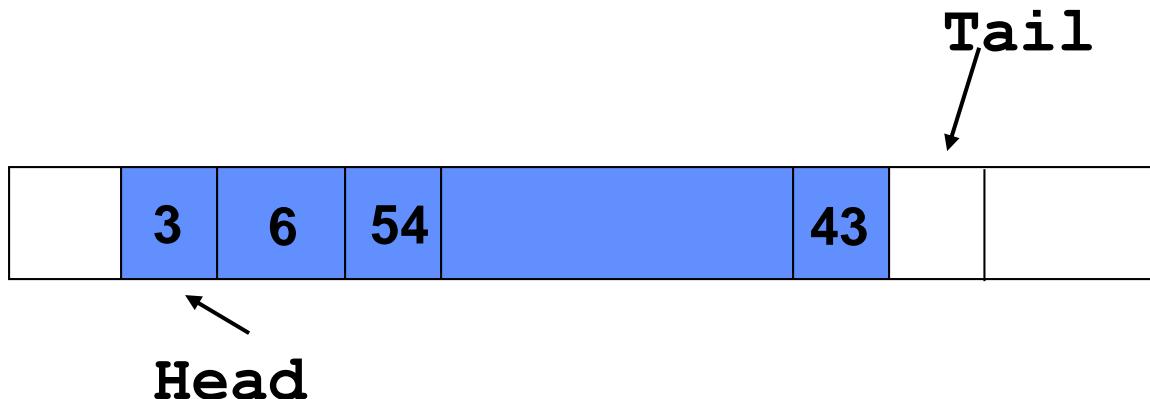
Code

Una Coda è un insieme dinamico in cui l'elemento rimosso dall'operazione di cancellazione è predeterminato.

In una Coda questo elemento è l'elemento che per più tempo è rimasto nell'insieme.

Una Coda implementa una lista di tipo “first in, first out” (FIFO)

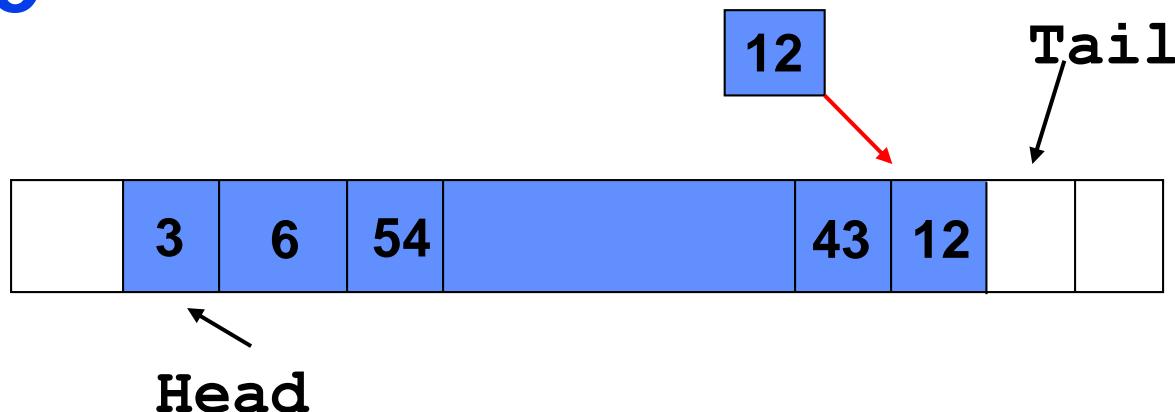
Code



*Una Coda implementa una lista di tipo “**first in, first out**” (**FIFO**)*

- Possiede una testa (**Head**) ed una coda (**Tail**)

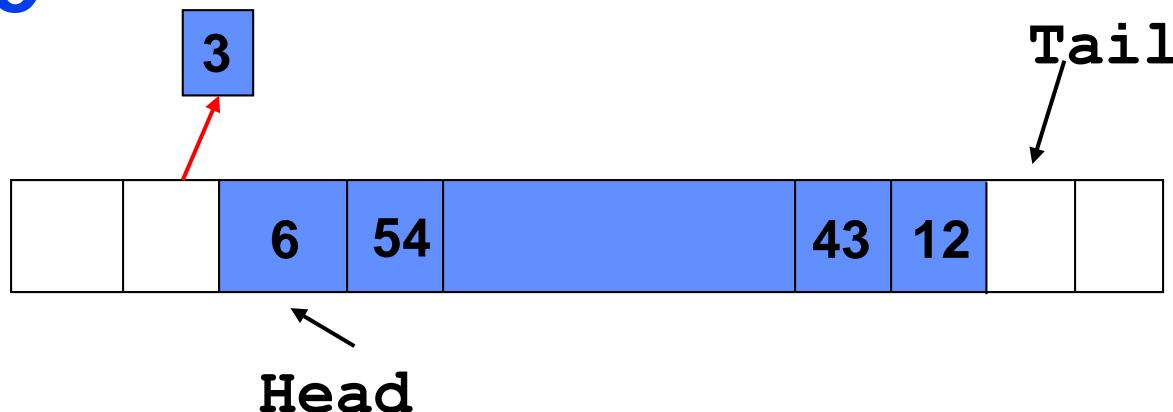
Code



Una Coda implementa una lista di tipo “first in, first out” (FIFO)

- Possiede una testa (**Head**) ed una coda (**Tail**)
- Quando si aggiunge un elemento, viene inserito in coda al posto referenziato da Tail

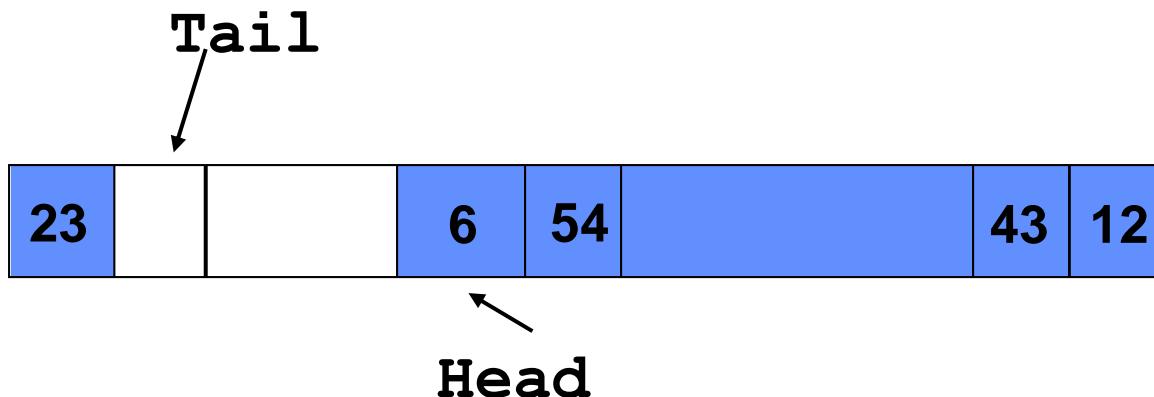
Code



Una Coda implementa una lista di tipo “first in, first out” (FIFO)

- Possiede una testa (**Head**) ed una coda (**Tail**)
- Quando si aggiunge un elemento, viene inserito in coda al posto referenziato da Tail
- Quando si estrae un elemento, viene estratto dalla testa

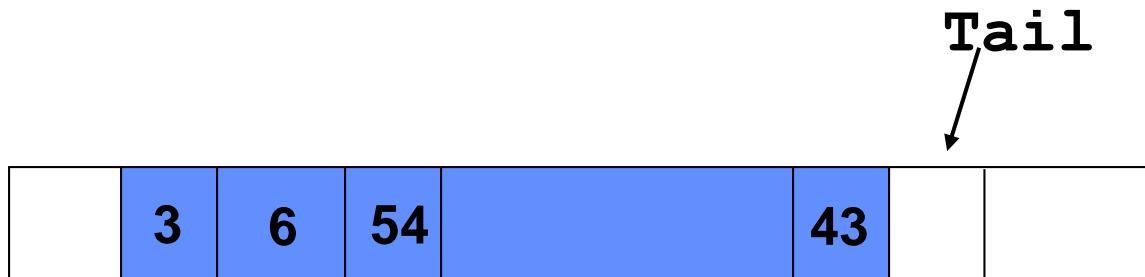
Code



Una Coda implementa una lista di tipo “first in, first out” (FIFO)

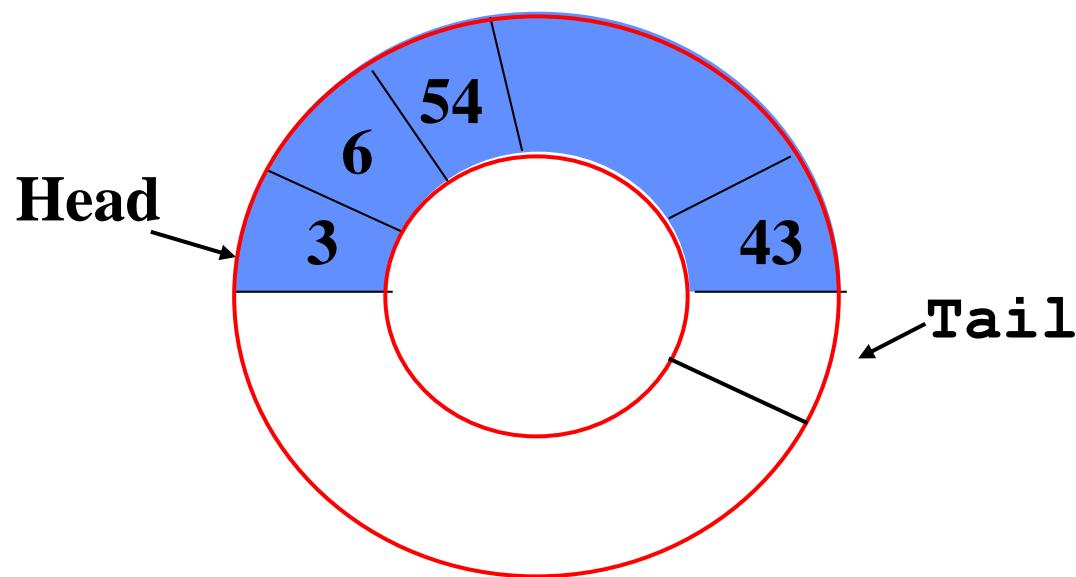
- La “finestra” dell’array occupata dalla coda si sposta lungo l’array!

Code



Head

Tail



La “**finestra**” dell’array
occupata dalla **coda** si
sposta lungo l’array!

Array Circolare
implementato ad esempio
con una operazione di
modulo

Operazioni su Code

Algoritmo Accoda(Q, x)

$Q[Tail_Q] = x$

IF ($Tail_Q = Length(Q)$)

THEN $Tail_Q = 1$

ELSE $Tail_Q = Tail_Q + 1$

Operazioni su Code

Algoritmo Accoda(Q, x)

```
 $Q[\text{Tail\_}Q] = x$ 
IF ( $\text{Tail\_}Q = \text{Length}(Q)$ )
    THEN  $\text{Tail\_}Q = 1$ 
ELSE  $\text{Tail\_}Q = \text{Tail\_}Q + 1$ 
```

Algoritmo Estrai-da-Coda(Q)

```
 $x = Q[\text{Head\_}Q]$ 
IF ( $\text{Head\_}Q = \text{Length}(Q)$ )
    THEN  $\text{Head\_}Q = 1$ 
ELSE  $\text{Head\_}Q = \text{Head\_}Q + 1$ 
return  $x$ 
```

Operazioni su Code: con modulo

Assumendo che l'array Q abbia indici 0...(n-1):

Algoritmo Accoda (Q, x)

$Q[Tail_Q] = x$

$Tail_Q = (Tail_Q + 1) \bmod n$

Algoritmo Estrai-da-Coda (Q)

$x = Q[Head_Q]$

$Head_Q = (Head_Q + 1) \bmod n$

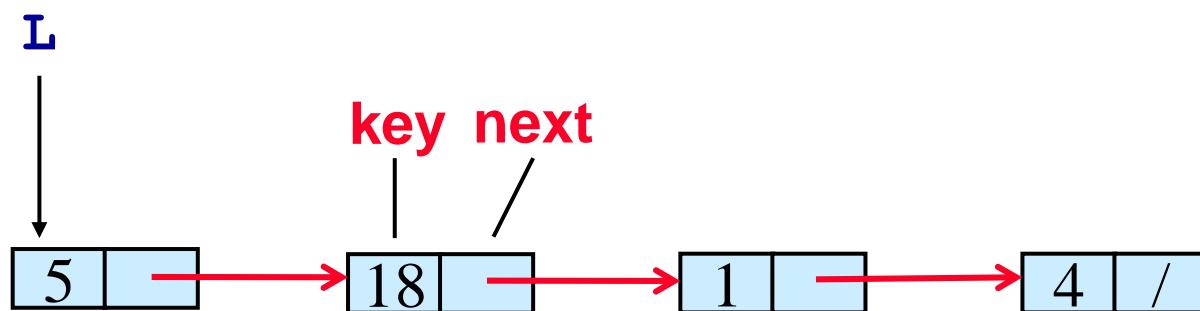
return x

Mancano anche qui le verifiche del caso in cui la coda sia piena e/o vuota. (ESERCIZIO)

Liste Puntate

Una Lista Puntata è un insieme dinamico in cui ogni elemento ha una chiave (key) ed un riferimento all'elemento successivo (next) dell'insieme.

È una struttura dati ad accesso strettamente sequenziale!

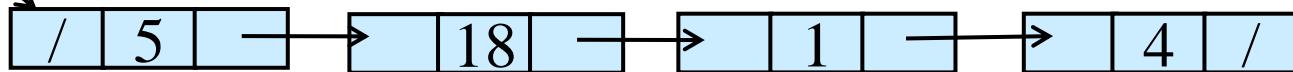


Operazioni su Liste Puntate

```
algoritmo Lista-Cerca-ric(L,k)
  IF L ≠ NIL and L->key ≠ k THEN
    L = Lista-Cerca-ric(L->next,k)
  return L
```

```
...
  elem = Lista-Cerca-ric(L,k)
...
```

L



Operazioni su Liste Puntate

Algoritmo **Lista-Insert(*L*, *k*)**

“**alloca nodo new**”

new->key = *k*

new->next = *L*

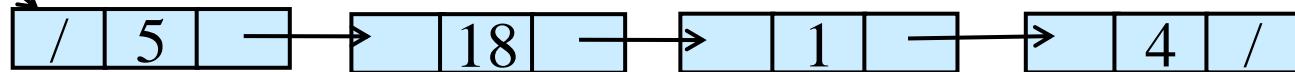
return new

...

L = **Lista-insert(*L*, *k*)**

...

L



Operazioni su Liste Puntate (ordinata)

Algoritmo **Lista-Ord-insert**(L, k)

```
IF  $L \neq \text{NIL}$  &&  $L\text{-}key < k$  THEN
     $L\text{-}next = \text{Lista-Ord-insert}(L\text{-}next, k)$ 
ELSE /* chiave  $k$  è la più piccola in  $L$  */
    "alloca nodo elem"
    elem $\text{-}key = k$ 
    elem $\text{-}next = L$ 
     $L = \text{elem}$ 
return  $L$ 
```

...

$L = \text{Lista-Ord-insert}(L, k)$

...



Operazioni su Liste Puntate

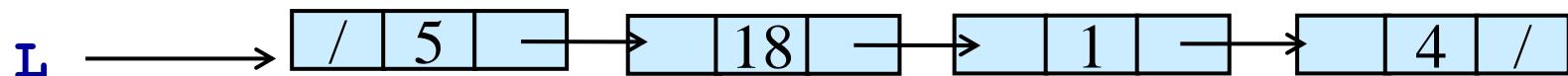
Algoritmo **Lista-cancella-r**(L, k)

```
IF  $L \neq \text{NIL}$  THEN
    IF  $L \rightarrow \text{key} = k$  THEN
        elem = L
        L = L  $\rightarrow$  next
        "dealloca elem"
    ELSE /*  $k$  non trovata in  $L$  */
        L  $\rightarrow$  next = Lista-cancella-r(L  $\rightarrow$  next, k)
return L
```

...

$L = \text{Lista-cancella-r}(L, k)$

...



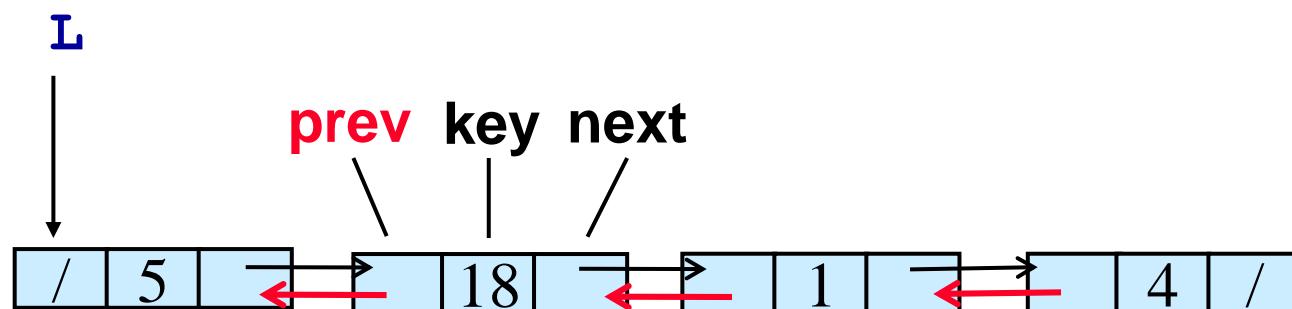
Esercizio

Scrivere un algoritmo ricorsivo che cancelli da una lista ordinata L tutti gli elementi con chiave compresa tra i valori k_1 e k_2 (con $k_1 \leq k_2$).

Liste Puntate Doppie

Una Lista Doppia Puntata è un insieme dinamico in cui in cui ogni elemento ha:

- una chiave (**key**)
- un riferimento (**next**) all'elemento successivo dell'insieme
- un riferimento (**prev**) all'elemento precedente dell'insieme.

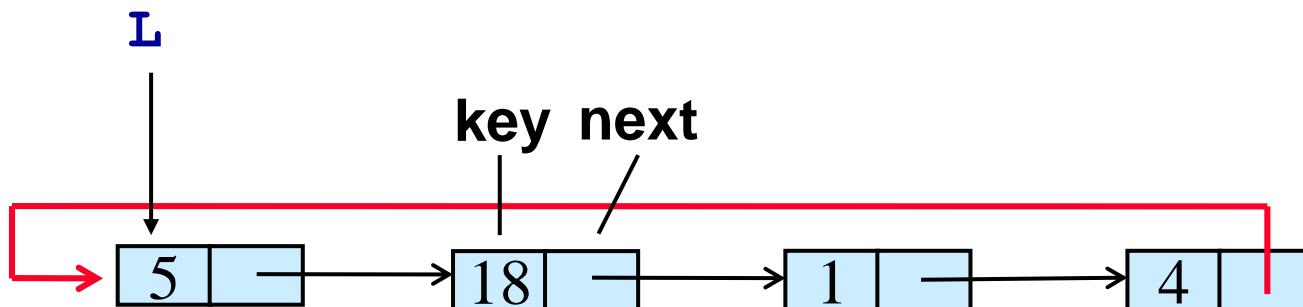


Liste Puntate Circolare

Una Lista Circolare puntata è un insieme dinamico in cui in cui ogni elemento ha:

- una chiave (**key**) e
- un riferimento (**next**) all'elemento successivo dell'insieme.

L'ultimo elemento ha un riferimento alla testa della lista

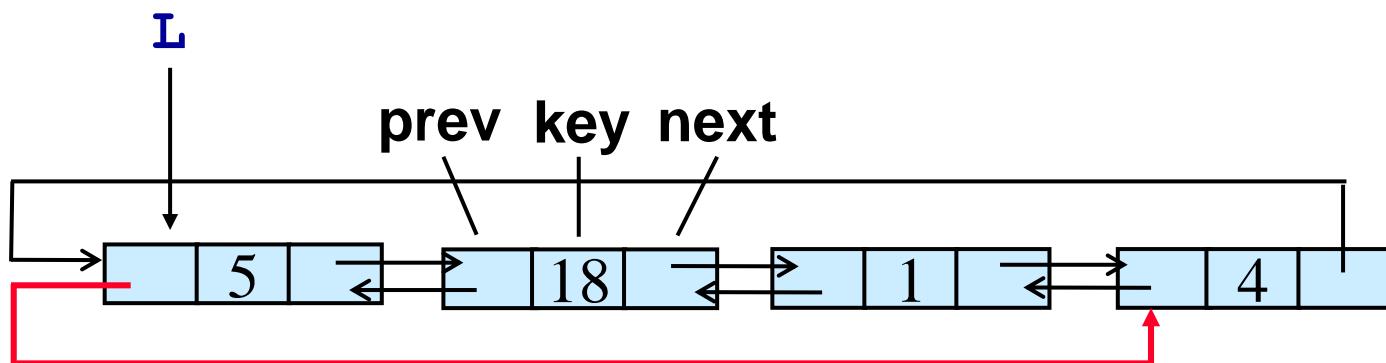


Liste Puntate Circolare Doppia

Una **Lista Circolare** puntata è un insieme dinamico in cui in cui ogni elemento ha:

- una chiave (**key**)
- un riferimento (**next**) all'elemento successivo dell'insieme
- un riferimento (**prev**) all'elemento pre-dente dell'insieme.

L'ultimo elemento ha un riferimento (**prev**) alla testa della lista, il primo ha un riferimento (**next**) alla coda della lista



Operazioni su Liste Puntate Doppie

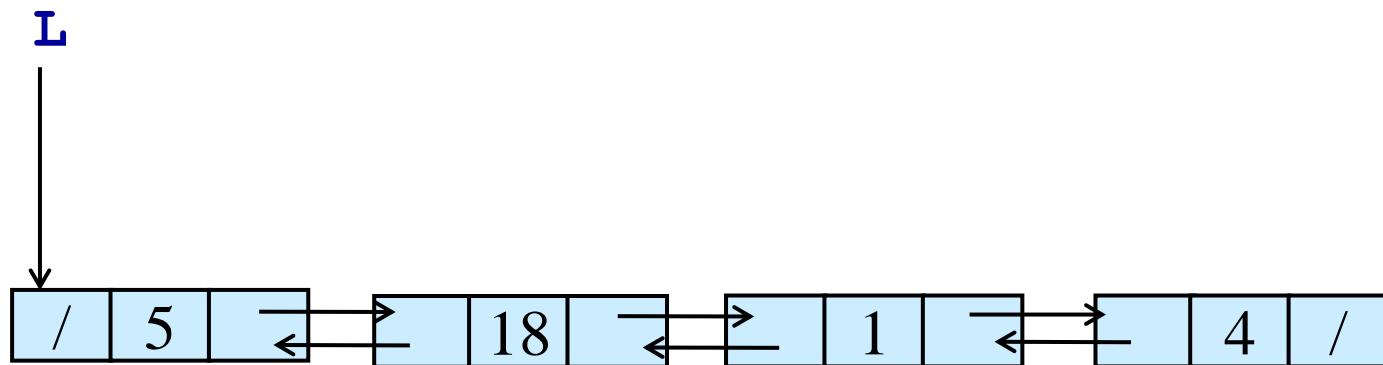
Algoritmo Lista-cerca-iter(*L*, *k*)

x* = *L

WHILE *x* ≠ NIL and *x*->*key* ≠ *k*

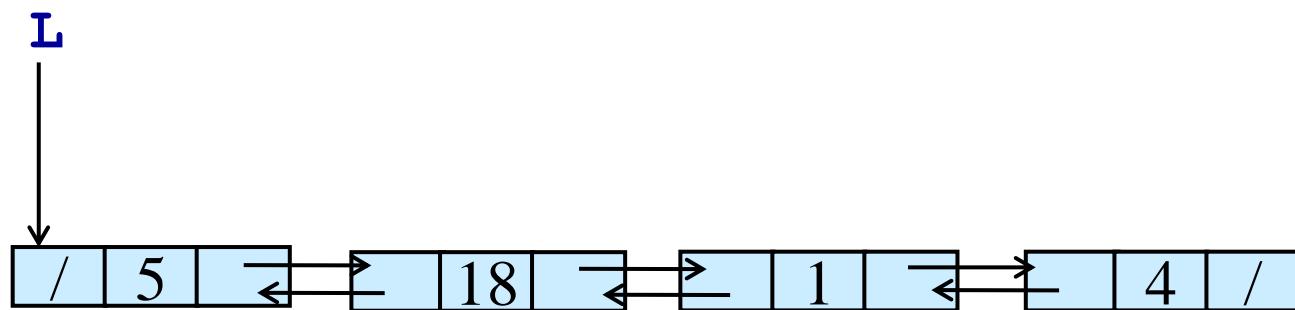
DO *x* = *x*->*next*

return *x*



Operazioni su Liste Puntate

```
Algoritmo ListaD-Inserisci (L, k)
    "alloca nodo elem"
    elem->key = k
    elem->next = L
    IF L ≠ NIL
        THEN L->prev = elem
    L = elem
    L->prev = NIL
```

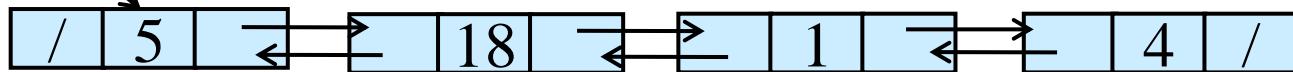


Operazioni su Liste Puntate Doppie

Algoritmo ListaD-Cancella (L, k)

```
x = Lista-Cerca (L, k)
IF x->prev ≠ NIL
    THEN x->prev->next = x->next
ELSE L = x->next
IF x->next ≠ NIL
    THEN x->next->prev = x->prev
"dealloca x"
```

L



Operazioni su Liste Puntate Doppie

Algoritmo **ListaD-canc**(*L*, *k*)

x = **Lista-Cerca-ric**(*L*, *k*)

IF *x* ≠ NIL THEN

 IF *x*->*next* ≠ NIL

 THEN *x*->*next*->*prev* = *x*->*prev*

 IF *x*->*prev* ≠ NIL

 THEN *x*->*prev*->*next* = *x*->*next*

 ELSE *L* = *x*->*prev*

“**dealloca** *x*”

return *L*

...

L = **ListaD-canc**(*L*, *k*)

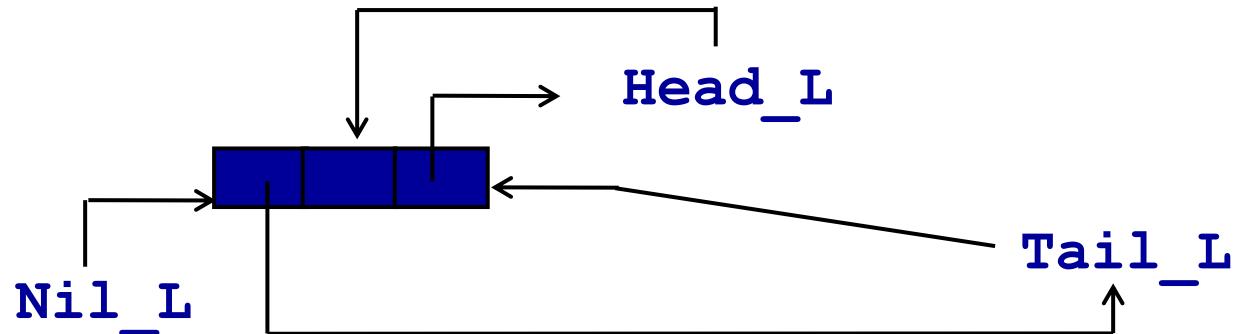
...



Liste con Sentinella

La Sentinella è un elemento **fittizio** Nil_L che permette di realizzare le operazioni di modifica di una lista puntata in modo più semplice.

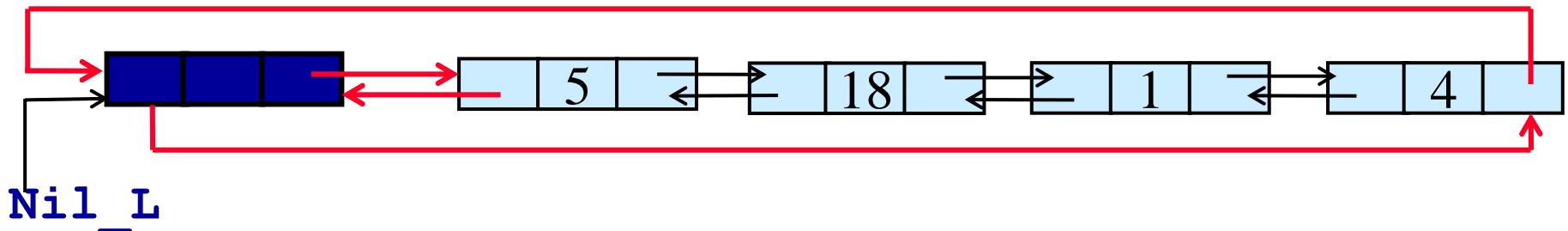
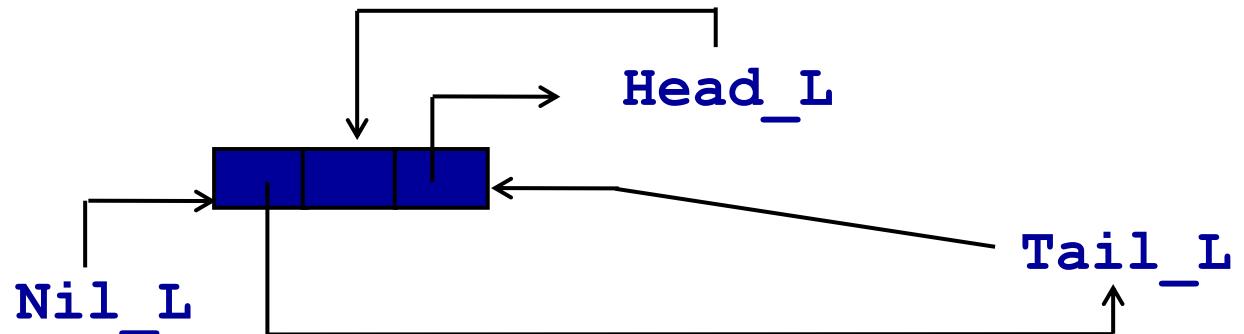
Nil_L viene inserito tra la testa e la coda della lista.



Liste con Sentinella

La Sentinella è un elemento **fittizio** Nil_L che permette di realizzare le operazioni di modifica di una lista puntata in modo più semplice.

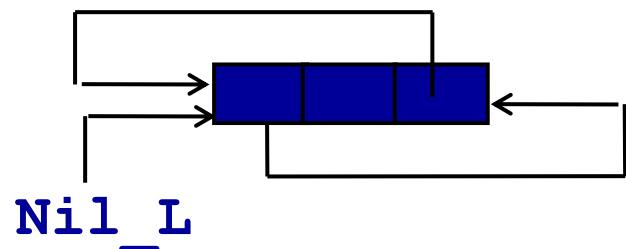
Nil_L viene inserito tra la testa e la coda della lista.
(Head_L può essere sostituito con un puntatore a Nil_L)



Liste con Sentinella

Nil_L viene inserito tra la testa e la coda della lista.

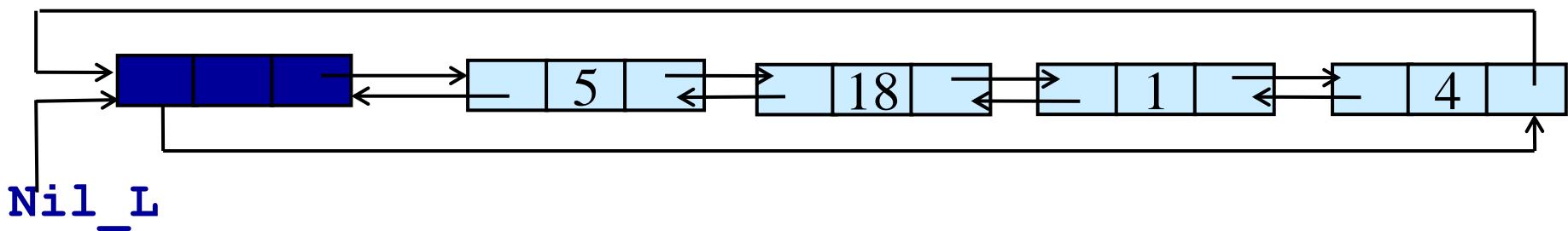
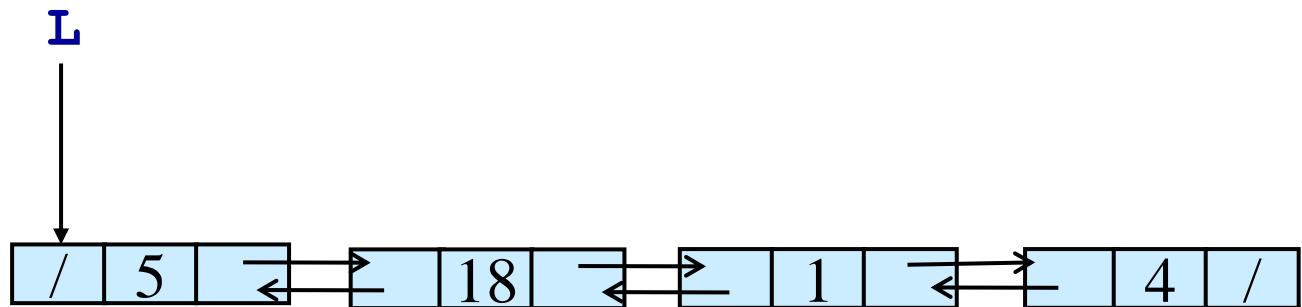
Nil_L da solo rappresenta la lista vuota (viene sostituito ad ogni occorrenza di NIL)



Liste con Sentinella

Nil_L viene inserito tra la testa e la coda della lista.

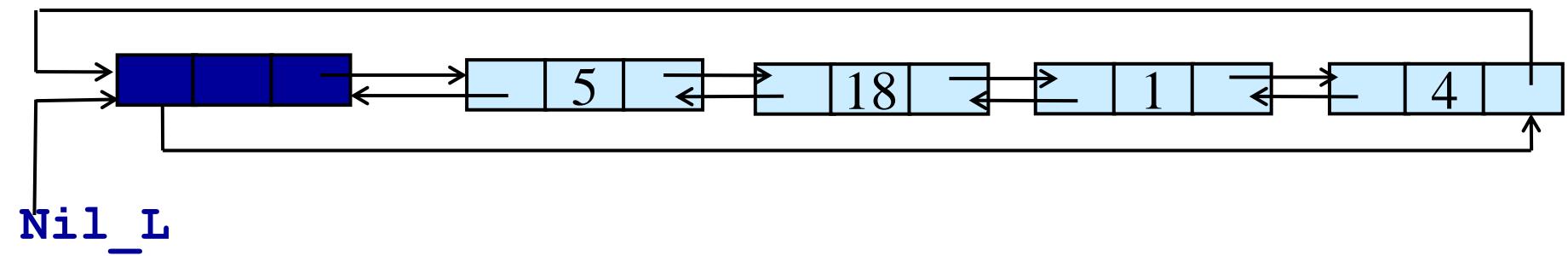
Questo trasforma una **lista (doppia)** in una **lista (doppia) circolare**



Liste con Sentinella

- La **Sentinella** è un elemento **fittizio Nil_L** che permette di realizzare le operazioni di modifica di una lista puntata in modo più semplice.

Perché non è più necessario preoccuparsi dei **casi limite** (ad esempio **cancellazione in testa/coda**)



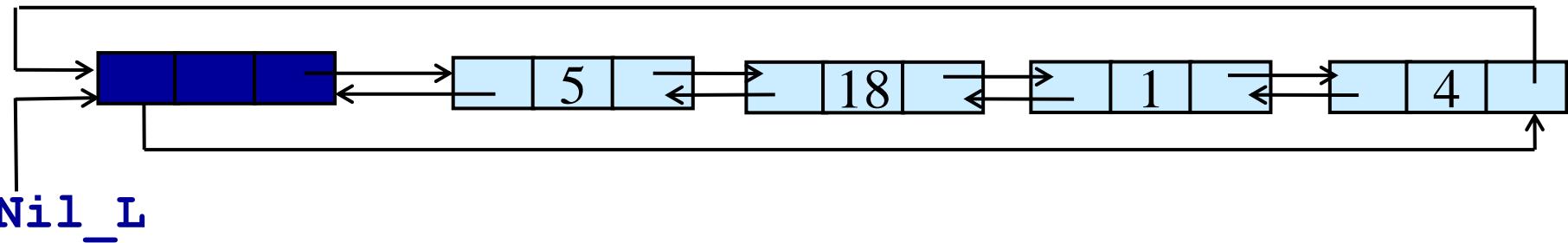
Operazioni su Liste con Sentinella

Algoritmo Lista-Cancella' (L, k)

$x = \text{Lista-Cerca'}(L, k)$

$x->\text{prev}->\text{next} = x->\text{next}$

$x->\text{next}->\text{prev} = x->\text{prev}$



Operazioni su Liste con Sentinella

Algoritmo Lista-Cancella' (L, k)

$x = \text{Lista-Cerca'}(L, k)$

$x->prev->next = x->next$

$x->next->prev = x->prev$

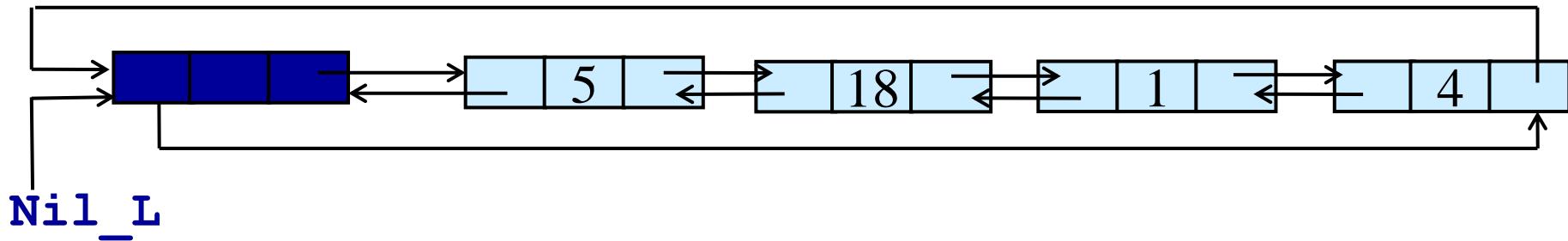
Algoritmo Lista-Inserisci' (L, x)

$x->next = \text{Nil_L}->next$

$\text{Nil_L}->next->prev = x$

$\text{Nil_L}->next = x$

$x->prev = \text{Nil_L}$

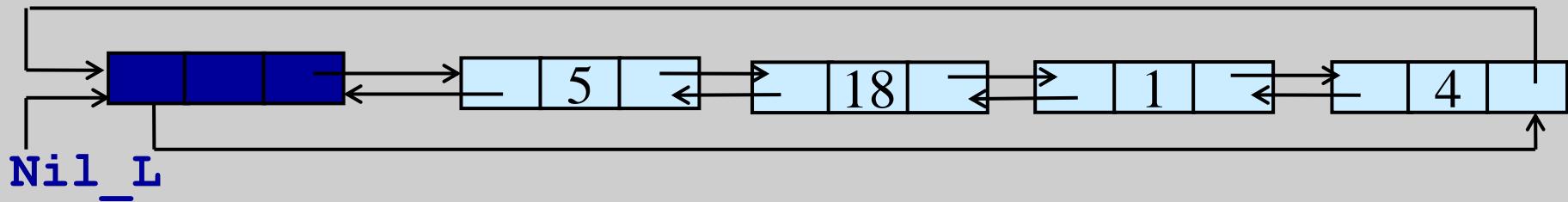


Operazioni su Liste con Sentinella

Algoritmo Lista-Cancella' (L, k)

$x = \text{Lista-Cerca'}(L, k)$

$x->prev->next = x->next$



$\text{Nil_L}->next->prev = x$

$\text{Nil_L}->next = x$

$x->prev = \text{Nil_L}$

Algoritmo Lista-Cerca' (L, k)

$x = \text{Nil_L}->next$

WHILE $x \neq \text{Nil_L}$ and $x->key \neq k$

DO $x = x->next$

return x

Liste LIFO e FIFO

Tramite le liste puntate e loro varianti è possibile realizzare ad esempio implementazioni generali di:

- **Stack come liste LIFO**
- **Code come liste FIFO (necessita in alcuni casi l'aggiunta di un puntatore alla coda della lista)**

Esercizio: Pensare a quali tipi di lista sono adeguati per i due casi e riscrivere le operazioni corrispondenti

Alberi

Una **Albero** è un insieme dinamico che

- è **vuoto** oppure
- è composto da **$k+1$ insiemi disgiunti** di nodi:
 - un insieme di cardinalità uno, detto **nodo radice**
 - **k alberi**, ciascuno dei quali è detto **sottoalbero i -esimo** della radice (dove $1 \leq i \leq k$)
- Un tale albero si dice albero di **grado k**
- Quando **$k=2$** , l'albero si dice **binario**

Visita di Alberi

Gli alberi possono essere visitati (o attraversati) in diversi modi:

Visita in Profondità: si visitano tutti i nodi lungo un percorso, poi quelli lungo un altro percorso, etc.

Visita in Ampiezza: si visitano tutti i nodi a livello 0, poi quelli a livello 1,..., poi quelli a livello h

Visite in Profondità

Gli alberi possono essere visitati (o attraversati) in profondità in diversi modi:

- **Visita in Preordine**: prima si visita il nodo e poi i suoi sottoalberi;
- **Visita Inordine (se binario)**: prima si visita il sottoalbero sinistro, poi il nodo e infine il sottoalbero destro;
- **Visita in Postordine** : prima si visitano i sottoalberi, poi il nodo.

Visita di Alberi Binari: in profondità preordine

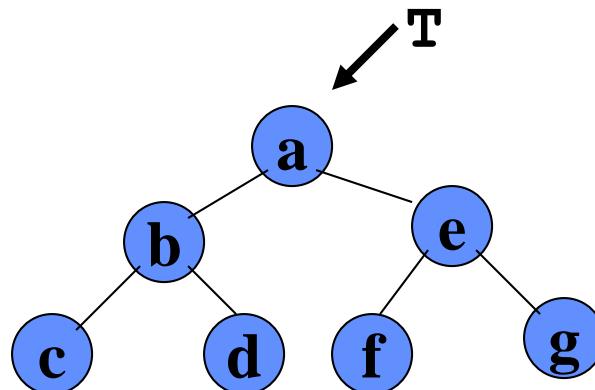
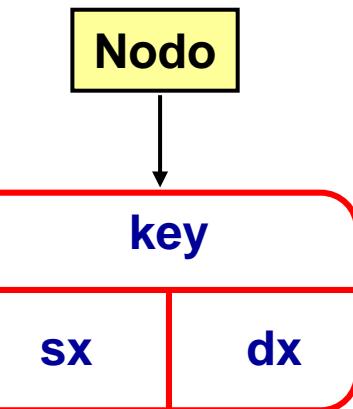
Visita-Preordine (T)

IF T ≠ NIL THEN

 Visita-Inordine (T->sx)

 “vista T”

 Visita-Inordine (T->dx)



Sequenza: a b c d e f g

Visita di Alberi Binari: in profondità inordine

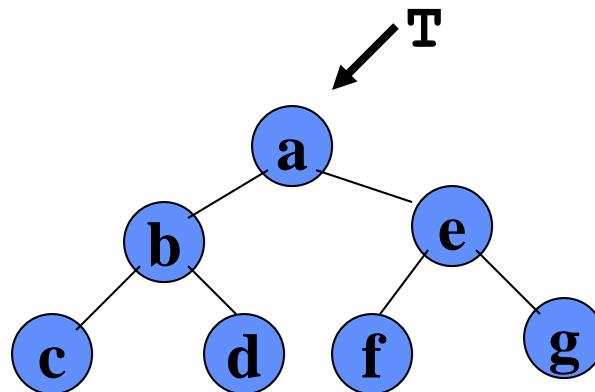
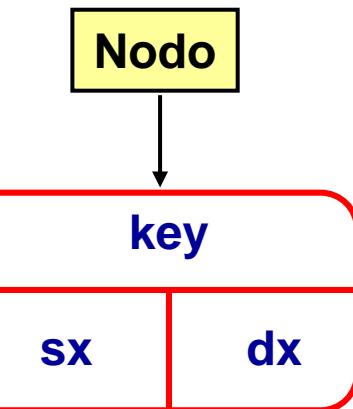
```
Visita-Inordine(T)
```

```
  IF T ≠ NIL THEN
```

```
    Visita-Inordine(T->sx)
```

"vista T"

```
    Visita-Inordine(T->dx)
```



Sequenza: c b d a f e g

Visita di Alberi Binari: in profondità postordine

Visita-Postordine (T)

IF T ≠ NIL THEN

 Visita-Postordine (T->sx)

 Visita-Postordine (T->dx)

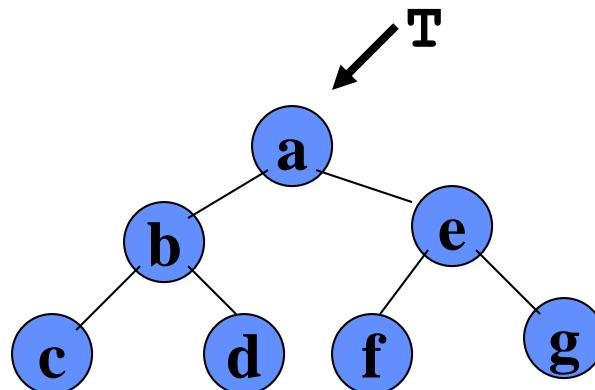
 “vista T”

Nodo

key

sx

dx



Sequenza: c d b f g e a

Visita Preorder Iterativa

```
Visita-preorder-iter(T)
```

```
    stack = NIL
```

```
    curr = T
```

```
WHILE (stack ≠ NIL OR curr ≠ NIL) DO
```

```
    IF (curr ≠ NULL) THEN /* Discesa a sx */
```

```
        "vista curr"
```

```
        push(stack,curr)
```

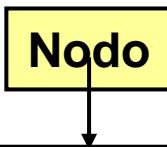
```
        curr = curr->sx
```

```
    ELSE      /* Risalita e discesa a dx */
```

```
        curr = top(stack)
```

```
        pop(stack)
```

```
        curr = curr->dx
```



Visita Inorder Iterativa

```
Visita-inorder-iter(T)
```

```
    stack = NIL
```

```
    curr = T
```

```
WHILE (stack ≠ NIL OR curr ≠ NIL) DO
```

```
    IF (curr ≠ NULL) THEN /* Discesa a sx */
```

```
        push(stack, curr)
```

```
        curr = curr->sx
```

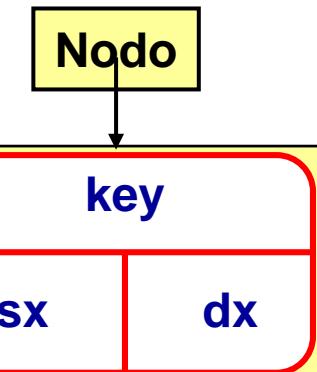
```
    ELSE /* Risalita e discesa a dx */
```

```
        curr = top(stack)
```

```
        pop(stack)
```

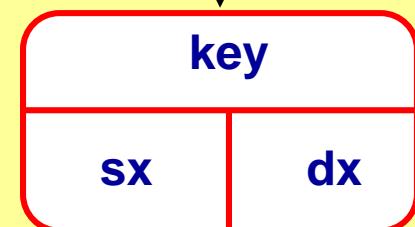
"vista curr"

```
        curr = curr->dx
```



Visita Postorder Iterativa

Nodo



Visita-postorder-iter(T)

stack = NIL, next = NIL

curr = T, last = NIL

WHILE (stack ≠ NIL OR curr ≠ NIL) DO

IF (curr ≠ NIL) THEN /* Discesa a sx */

stack = push(stack, curr)

next = curr->sx

ELSE

curr = top(stack) /* Risalita */

IF (curr->dx ≠ NIL AND curr->dx ≠ last) THEN

next = curr->dx /* Discesa a dx */

ELSE /* Risale da dx o dx vuoto */

“vista curr”

stack = pop(stack)

next = NIL

last = curr /* applica operazione di discesa */

curr = next /* o di risalita */

Visita Preorder Iterativa II

Nodo

padre

key

sx

dx

Visita-preorder-iter-2(T)

curr = T

last = NIL

WHILE (curr ≠ NIL) DO

IF (last = NIL) THEN

"vista curr"

IF (last = NIL AND curr->sx ≠ NIL) THEN

curr = curr->sx /* Discesa a sx */

ELSE IF (last ≠ curr->dx AND curr->dx ≠ NIL) THEN

curr = curr->dx /* Discesa a dx */

last = NIL

ELSE /* Risale da dx o dx vuoto */

last = curr

curr = curr->padre

Visita Preorder Iterativa II

Nodo

padre

key

sx

dx

Visita-preorder-iter-2(T)

curr = T

last = NIL

WHILE (curr ≠ NIL) DO

IF (last = c->padre) THEN

"vista curr"

IF (last = c->padre AND curr->sx ≠ NIL) THEN

next = curr->sx /* Discesa a sx */

ELSE IF (last ≠ curr->dx AND curr->dx ≠ NIL) THEN

next = curr->dx /* Discesa a dx */

ELSE /* Risale da dx o dx vuoto */

next = curr->padre

last = curr

curr = next

Visita Inorder Iterativa II

Nodo

padre

key

sx

dx

Visita-inorder-iter-2(T)

curr = T

last = NIL

WHILE (curr ≠ NIL) DO

IF (last = NIL AND curr->sx ≠ NIL) THEN

 curr = curr->sx /* Discesa a sx */

ELSE

 IF (last = curr->sx) THEN /* Risale da sx */

 “vista curr”

 IF (last = curr->sx AND curr->dx ≠ NIL)

 curr = curr->dx /* Discesa a dx */

 last = NIL

 ELSE /* Risale da dx */

 last = curr

 curr = curr->padre

Visita Postorder Iterativa II

Nodo



Visita-postorder-iter-2(T)

curr = T

last = NIL

WHILE (curr ≠ NIL) DO

IF (last = NIL AND curr->sx ≠ NIL) THEN

 curr = curr->sx /* Discesa a sx */

ELSE IF (last = curr->sx AND curr->dx ≠ NIL) THEN

 curr = curr->dx /* Discesa a dx */

 last = NIL

ELSE /* Risale da dx o dx vuoto */

 “vista curr”

 last = curr

 curr = curr->padre

Visita di Alberi k-ari: in ampiezza

Visita-Aampiezza (T)

IF T ≠ NIL THEN

“crea la coda vuota Q di dimensione $\left\lceil \frac{(k-1)n}{k} \right\rceil$ ”
Accoda (Q, T)

REPEAT

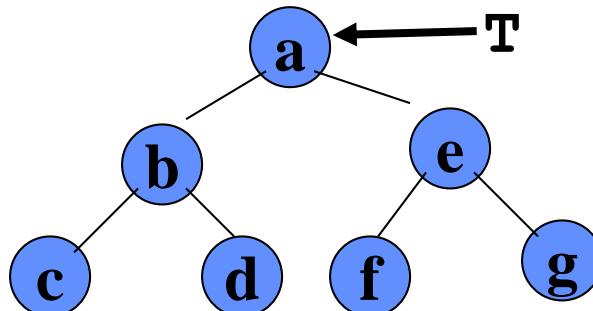
P = Estrai-da-Coda (Q)

“visita P”

FOR “ogni figlio F di P da sinistra”

DO Accoda (Q, F)

UNTIL Coda-Vuota (Q)



Sequenza: a b e c d f g

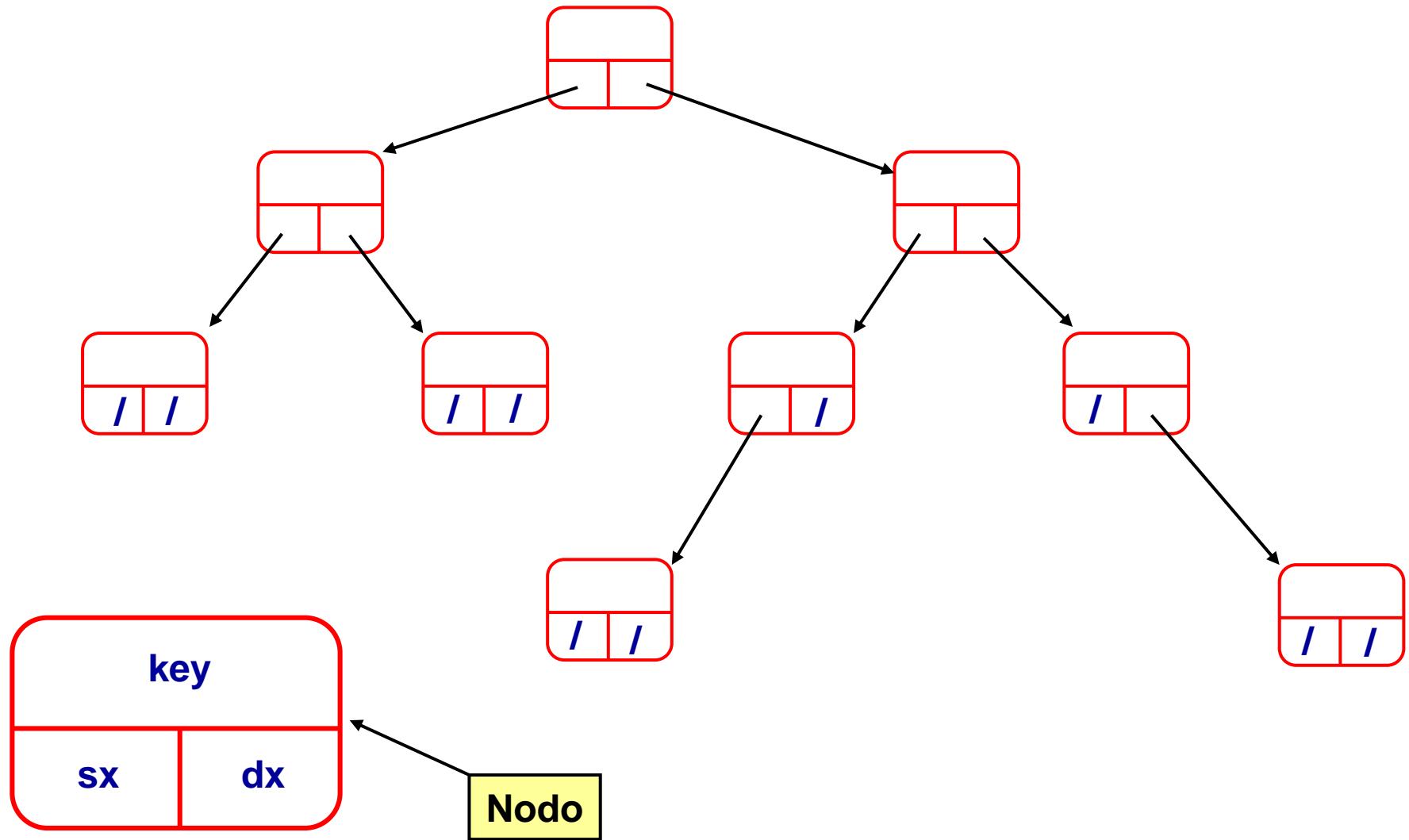
Implementazione di Alberi Binari

Come è possibile *implementare strutture dati puntate di tipo Albero*?

Gli alberi possono essere implementati facilmente utilizzando *tecniche simili* a quelle che impieghiamo per implementare *liste puntate*.

Se non abbiamo a disposizione puntatori, si possono utilizzare ad esempio *opportuni array*, simulando il meccanismo di gestione della memoria (*allocazione, deallocazione*)

Implementazione di Alberi Binari

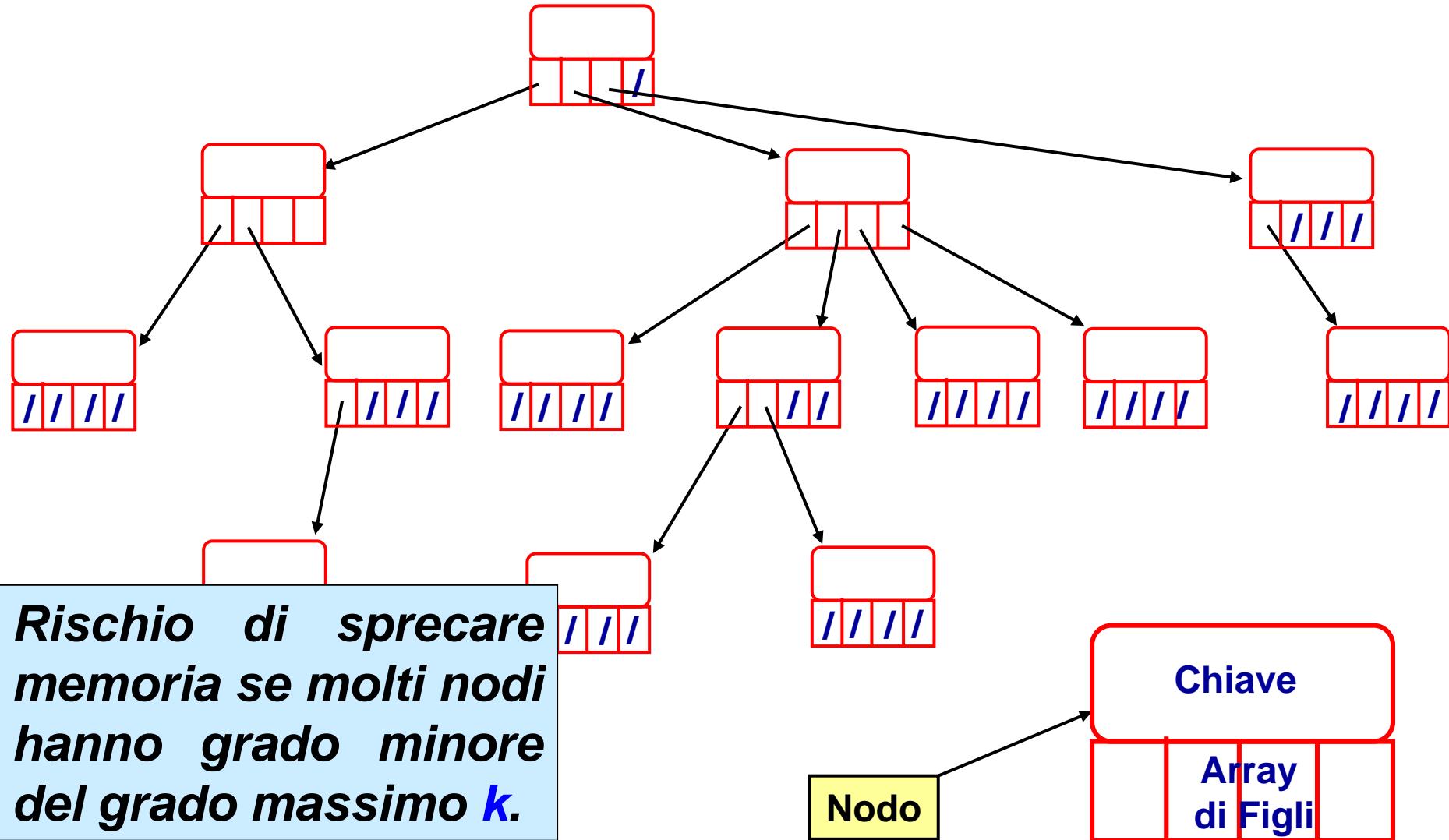


Ricerca in un albero binario

```
tree Search_Tree(T,k)
if (T ≠ NIL)
    if (T->Key = k) then
        ris = T;
    else
        ris = Search_Tree(T->sx,k);
        if (ris = NIL) then
            ris = Search_Tree(T->dx,k);
return ris;
```

Indipendentemente dalla forma dell'albero, il tempo di ricerca è chiaramente lineare nel numero di nodi dell'albero nel caso peggiore (es. se la chiave non è presente).

Implementazione di Alberi Arbitrari



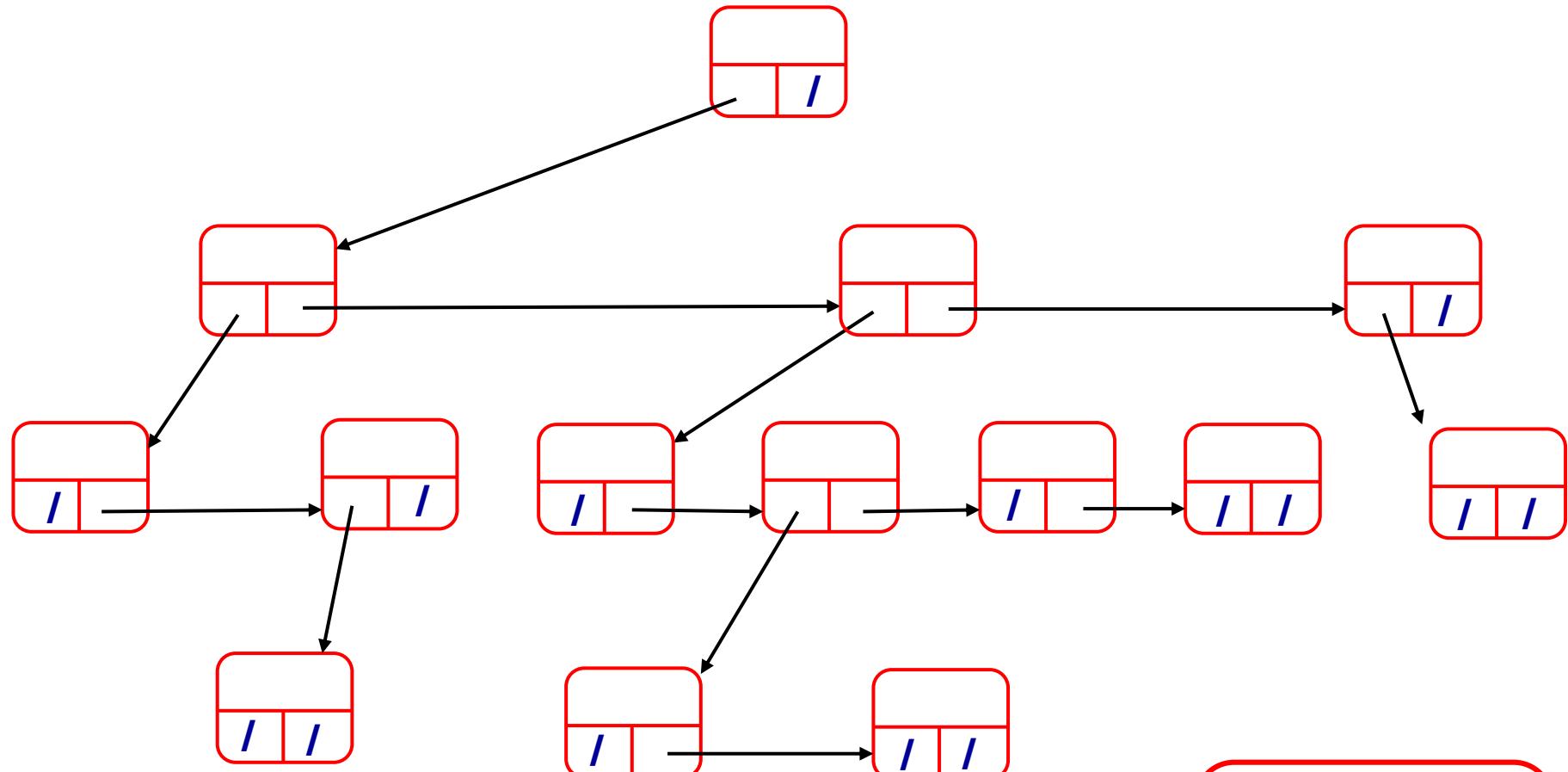
Ricerca in un albero generico

```
tree Search_Tree(T,k)
  if (T ≠ NIL) then
    if (T->Key = k) then
      ris = T;
    else
      i = Next_Son(T,0);
      while (i ≠ 0) do
        ris = Search_Tree(T->son[i],k);
        if (ris = NIL) then
          i = Next_Son(T,i);
        else
          i = 0;
      return ris;
```

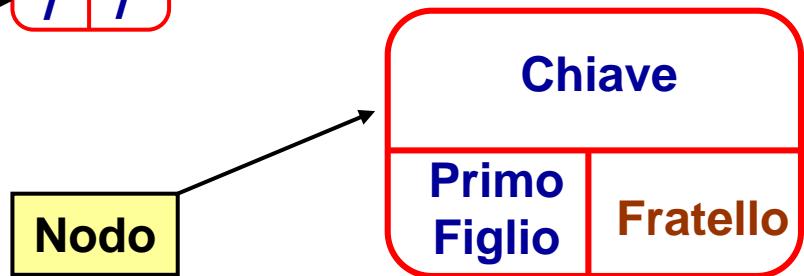
```
int Next_Son(T,id)
  s = id + 1;
  while (s ≤ MAX_SONS &&
         T->son[s] = NIL) do
    s = s + 1;
  if (s ≤ MAX_SONS)
    return s;
  else
    return 0;
```

Assumendo gli indici dell'array $T->\text{son}$ da 1 a k

Implementazione di Alberi Arbitrari



Soluzione: usare una lista di figli (fratelli).



Ricerca in un albero generico (basata su ricerca in alberi binari)

```
tree Search_Tree(T,k)
  if (T ≠ NIL)
    if (T->Key = k) then
      ris = T;
    else
      ris = Search_Tree(T->sx,k);
      if (ris = NIL) then
        ris = Search_Tree(T->dx,k);
  return ris;
```

Implementazione di Puntatori

È possibile implementare strutture dati puntate come le Liste o gli Alberi senza utilizzare i puntatori?

Alcuni linguaggi di programmazione non ammettono puntatori (ad esempio il Fortran)

È possibile utilizzare gli stessi algoritmi che abbiamo visto fin'ora in questi linguaggi di programmazione?

Implementazione di Puntatori

È necessario simulare il meccanismo di gestione della memoria utilizzando le strutture dati a disposizione.

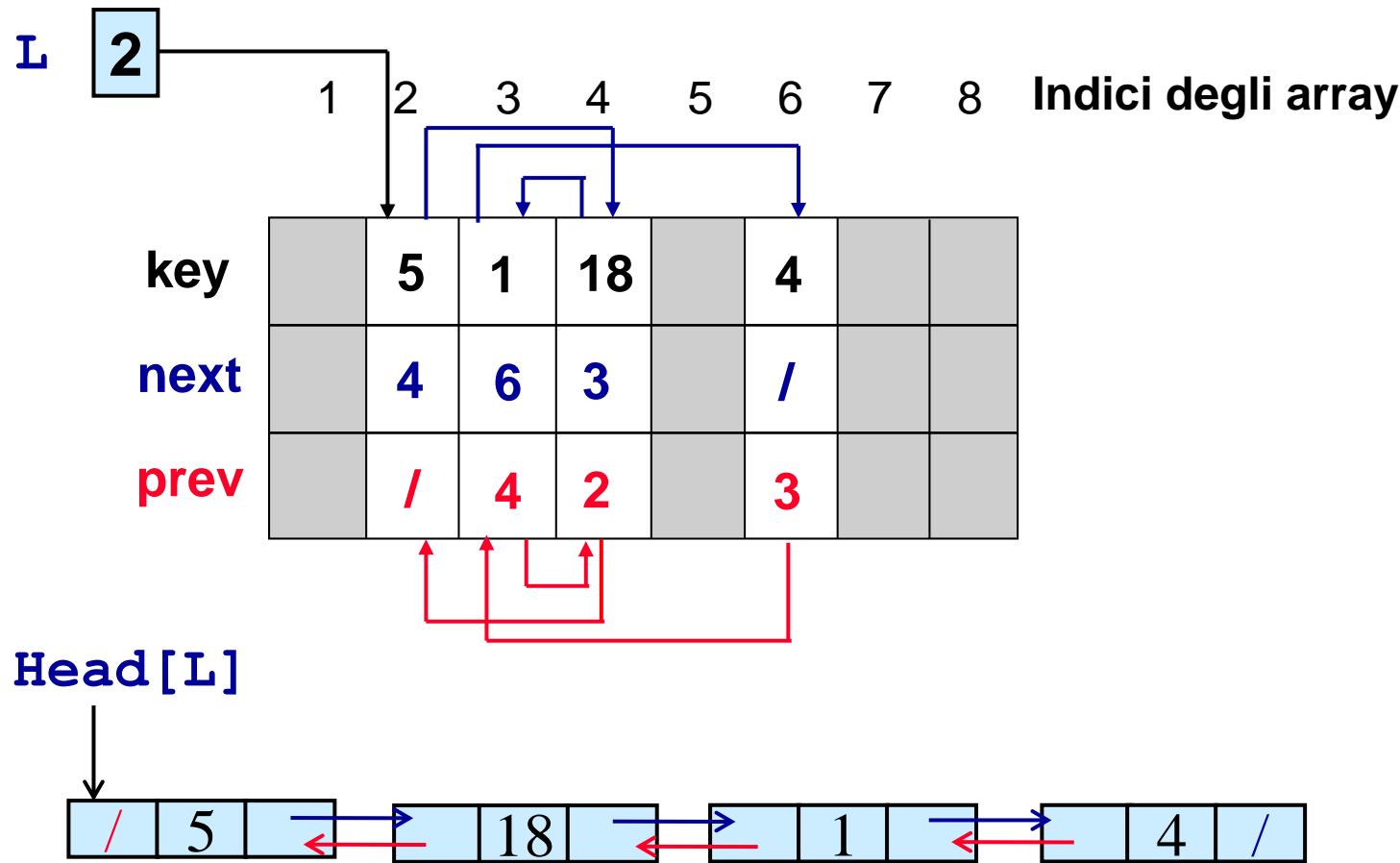
Ad esempio è possibile utilizzare array come contenitori di elementi di memoria.

Possiamo usare:

- *un array key[] per contenere i valori delle chiavi della lista*
- *un array next[] per contenere i puntatori (valori di indici) all'elemento successivo*
- *un array prev[] per contenere i puntatori (valori di indici) all'elemento precedente*

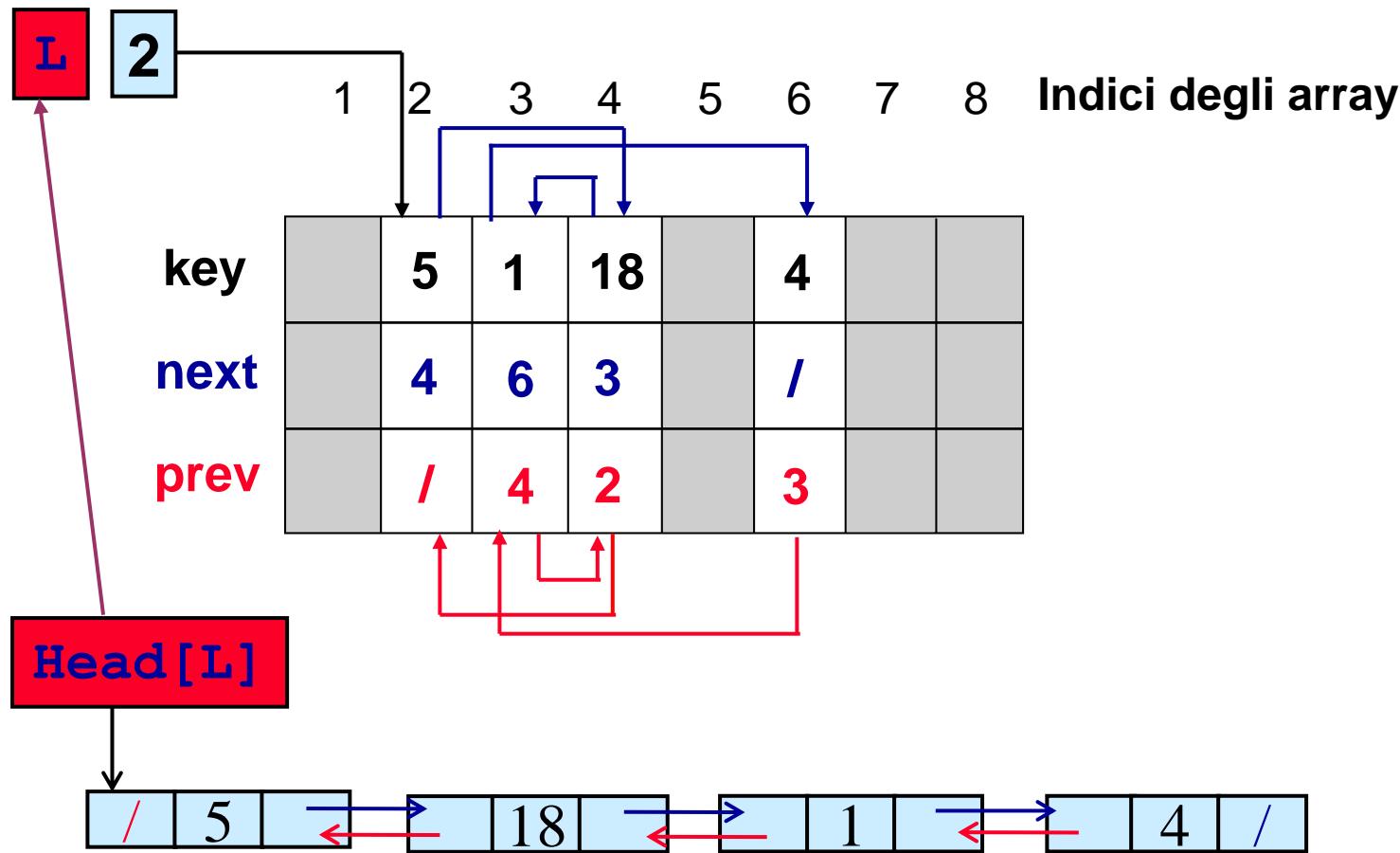
Implementazione di Puntatori

Implementazione di liste puntate doppie con tre array: key[], next[] e prev[]



Implementazione di Puntatori

Implementazione di liste puntate doppie con tre array: key[], next[] e prev[]



Implementazione di Puntatori

È possibile utilizzare array come contenitori di elementi di memoria.

Ma gli array hanno dimensione fissa e implementarvi strutture dinamiche può portare a sprechi di memoria

Possiamo allora sviluppare un vero e proprio meccanismo di allocazione e deallocazione degli elementi di memoria negli array.

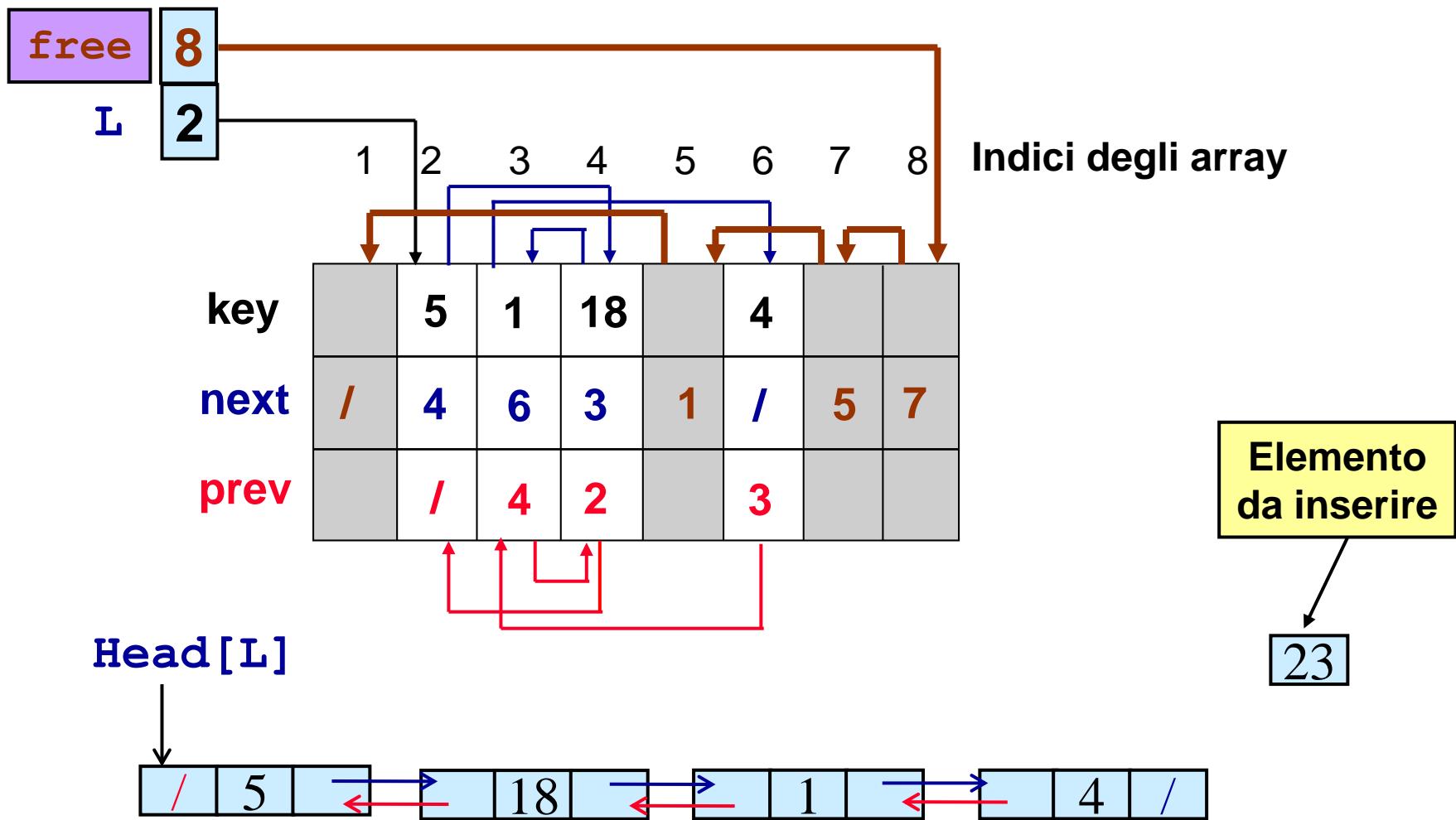
Implementazione di Puntatori

Possiamo usare:

- *un array key[] per contenere i valori delle chiavi della lista*
- *un array next[] per contenere i puntatori (valori di indici) all'elemento successivo*
- *un array prev[] per contenere i puntatori (valori di indici) all'elemento precedente*
- *e una variabile free per indicare l'inizio di una lista di elementi ancora liberi (free list)*

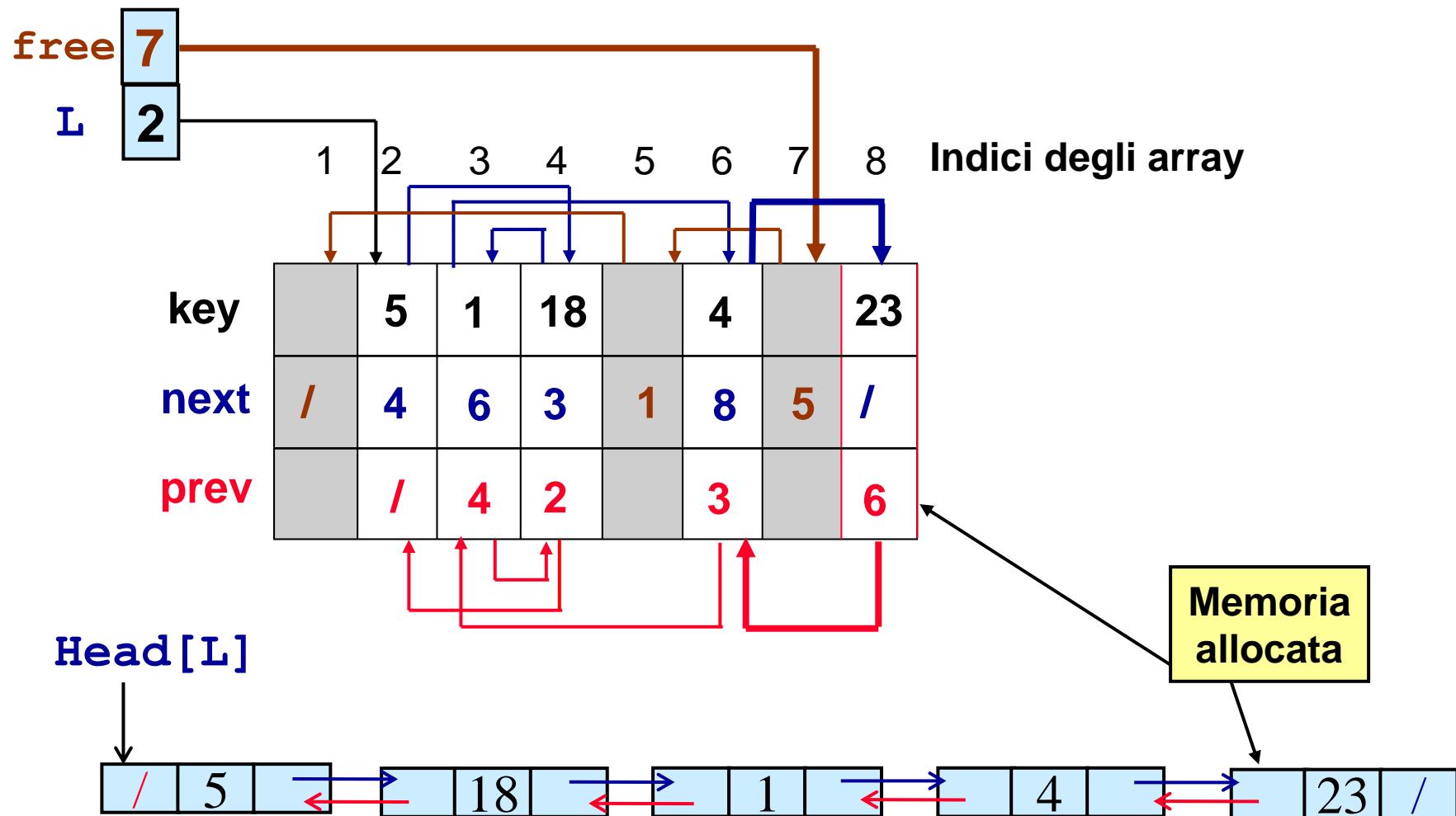
Allocazione memoria

Implementazione di liste puntate doppie con tre array: key[], next[] e prev[], free è la free list



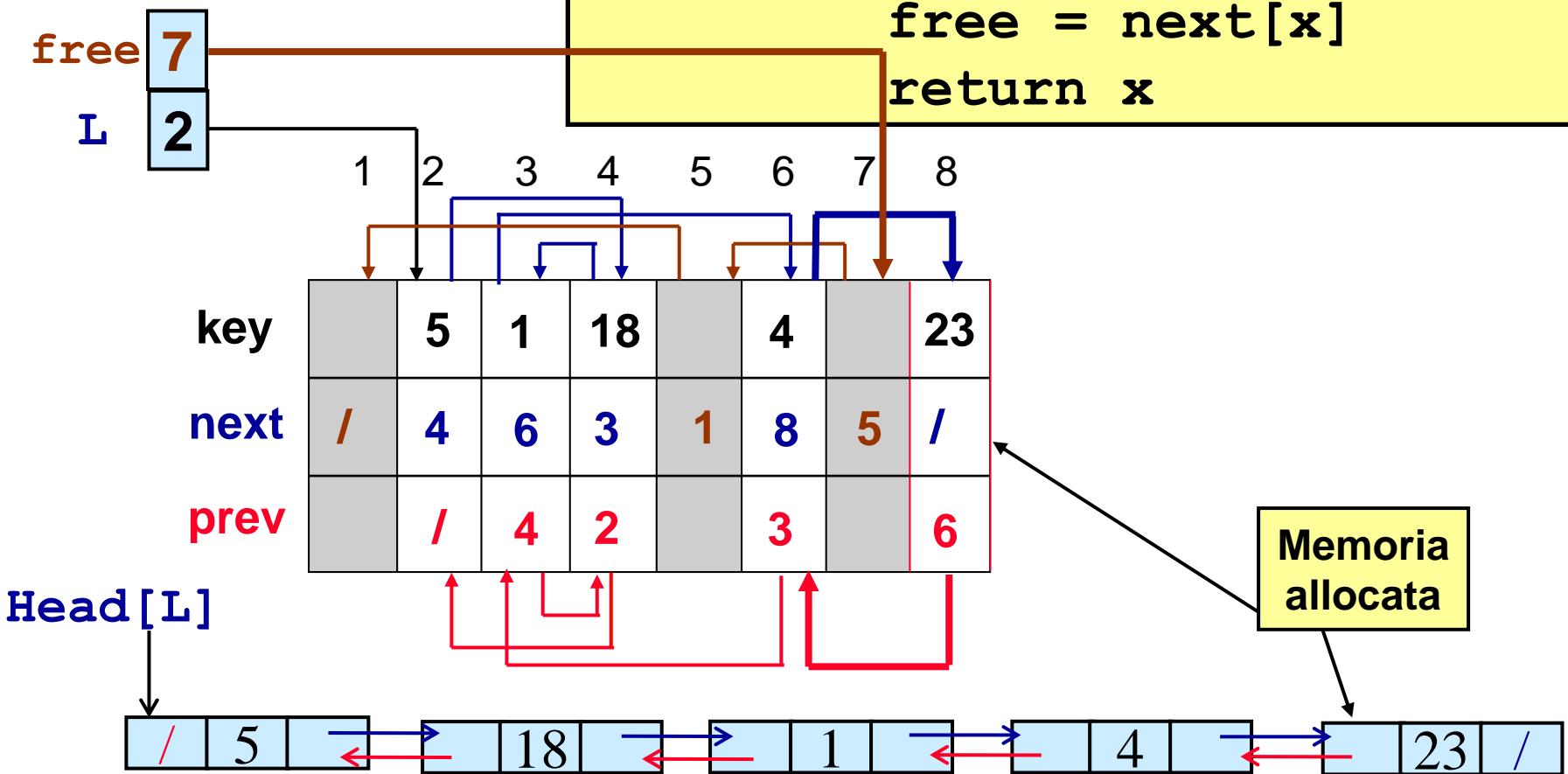
Allocazione memoria

Implementazione di liste puntate doppie con tre array: key[], next[] e prev[], free è la free list



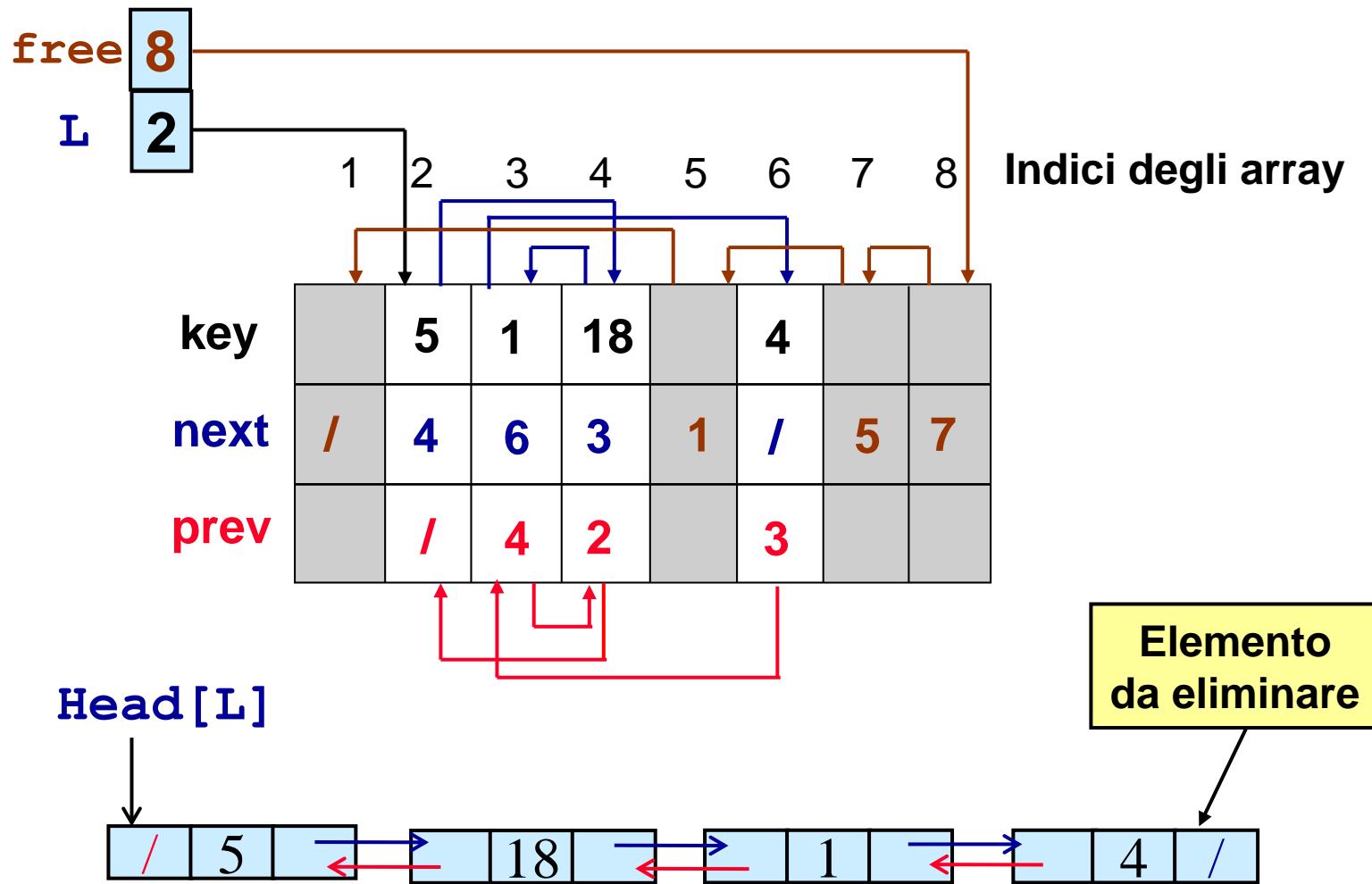
Allocazione memoria

Implementazione
array: key[], next



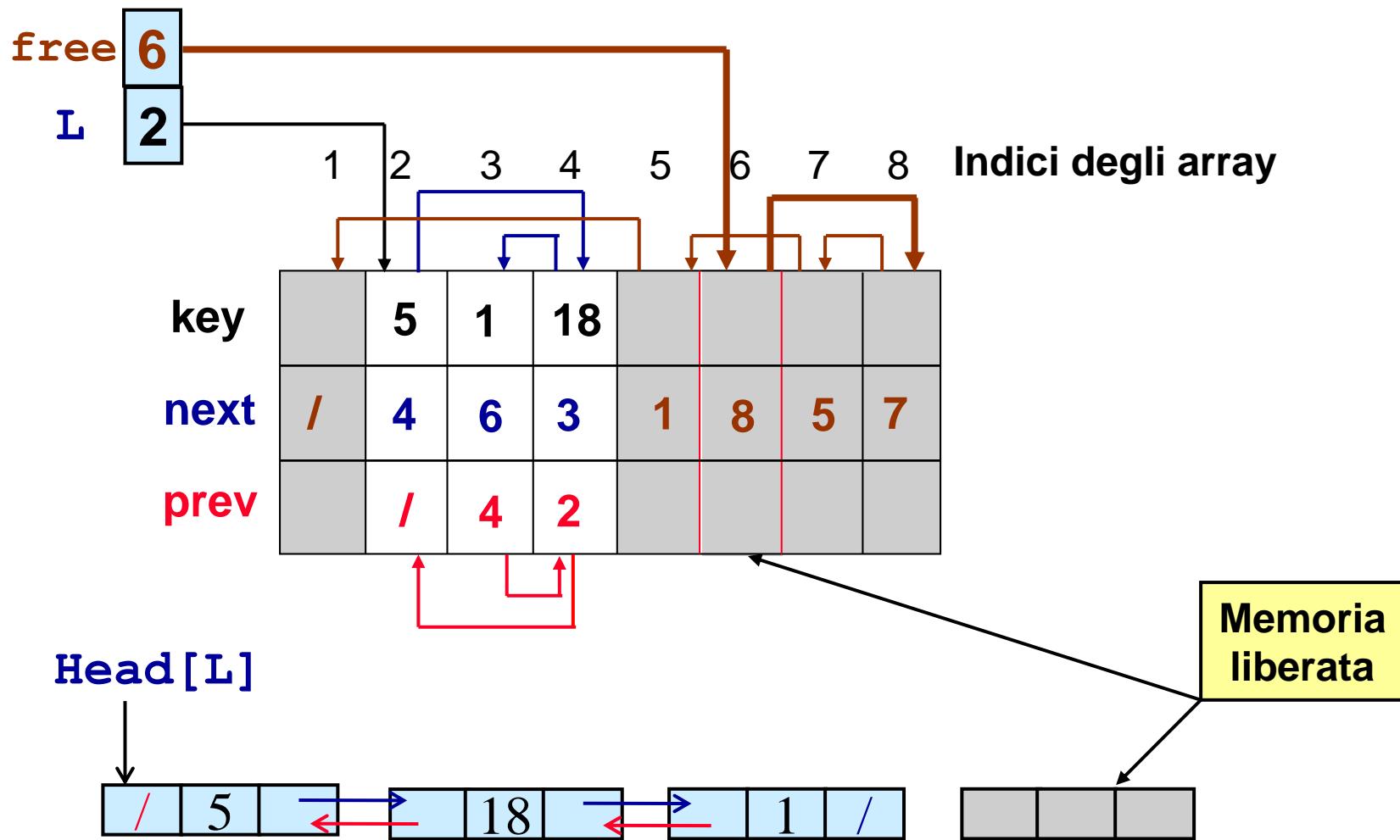
Deallocazione memoria

Implementazione di liste puntate doppie con tre array: key[], next[] e prev[], free è la free list



Deallocazione memoria

Implementazione di liste puntate doppie con tre array: key[], next[] e prev[], free è la free list



Deallocazione memoria

Implementazione di liste
array: **key[]**, **next[]** e **prev**

free

L

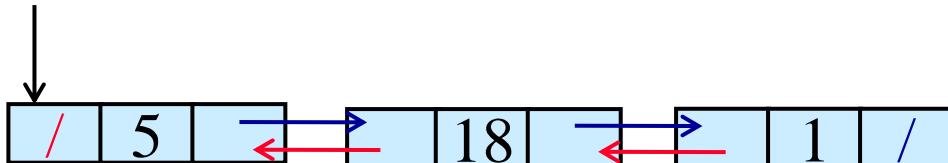
key

next

prev

	1	2	3	4	5	6	7	8
key	5	1	18					
next	/	4	6	3	1	8	5	7
prev	/	4	2					

Head [L]



```
Dealloca-elemento(x)
next[x] = free
free=x
```

Indici degli array

Memoria
liberata

Algoritmi e Strutture Dati

Alberi Binari di Ricerca

Alberi binari di ricerca

Motivazioni

- gestione e ricerche in grosse quantità di dati
- *liste, array* e *alberi non sono adeguati* perché inefficienti in tempo $O(n)$ o in spazio

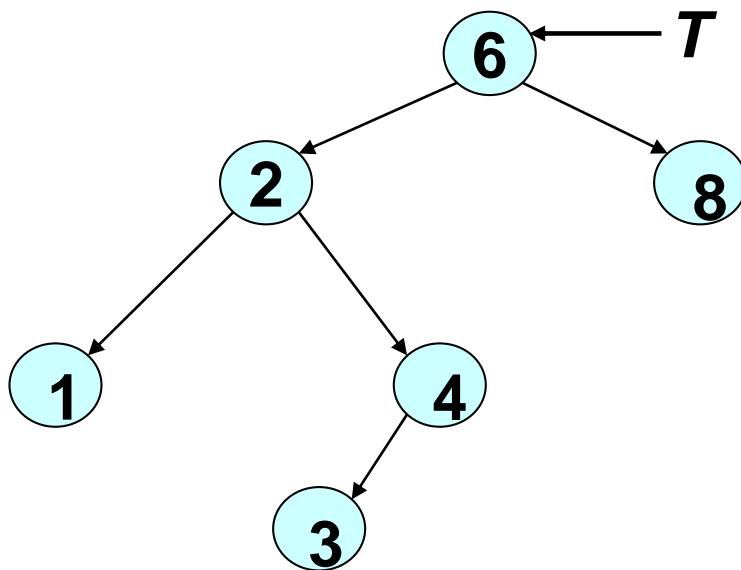
Esempi:

- Mantenimento di archivi (*DataBase*)
- In generale, mantenimento e gestione di corpi di *dati* su cui si effettuano *molte ricerche*, eventualmente alternate a operazioni di inserimento e cancellazione.

Alberi binari di ricerca

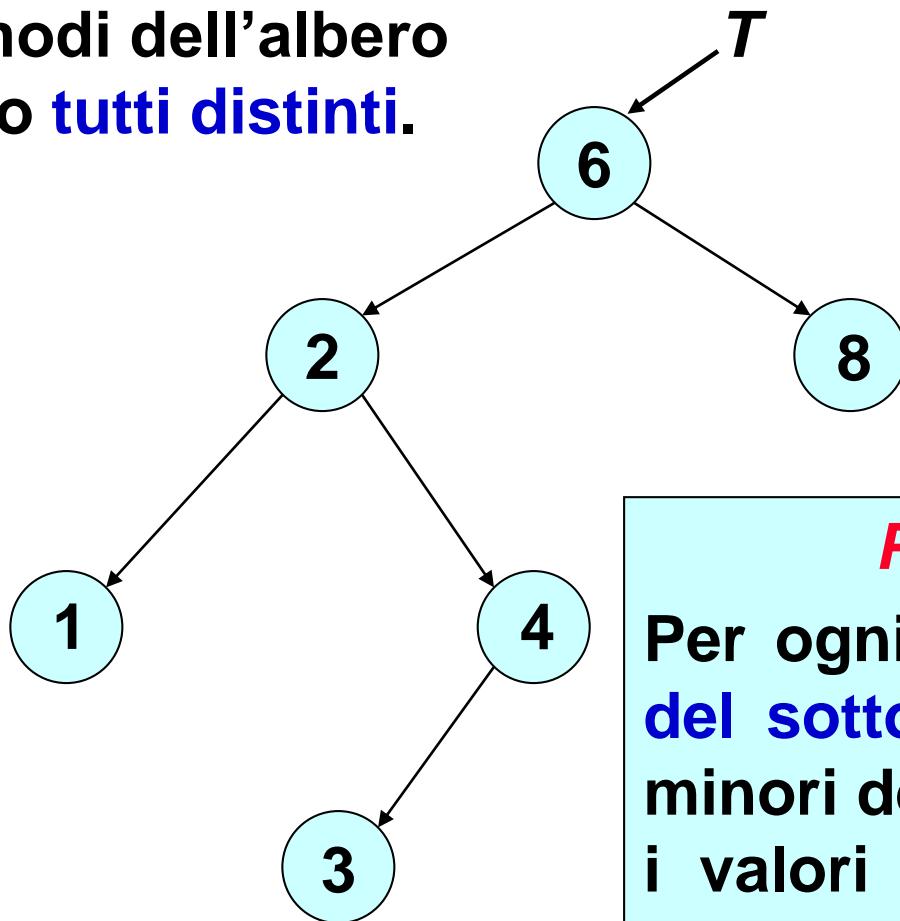
Definizione: Un albero binario di ricerca è un albero binario che soddisfa la seguente proprietà:

se X è un nodo e Y è qualsiasi un nodo nel **sottoalbero sinistro** di X , allora $Y->key \leq X->key$;
inoltre, se Y è qualsiasi un nodo nel **sottoalbero destro** di X allora $Y->key \geq X->key$



Alberi binari di ricerca

Assumiamo che i valori nei nodi dell'albero siano tutti distinti.



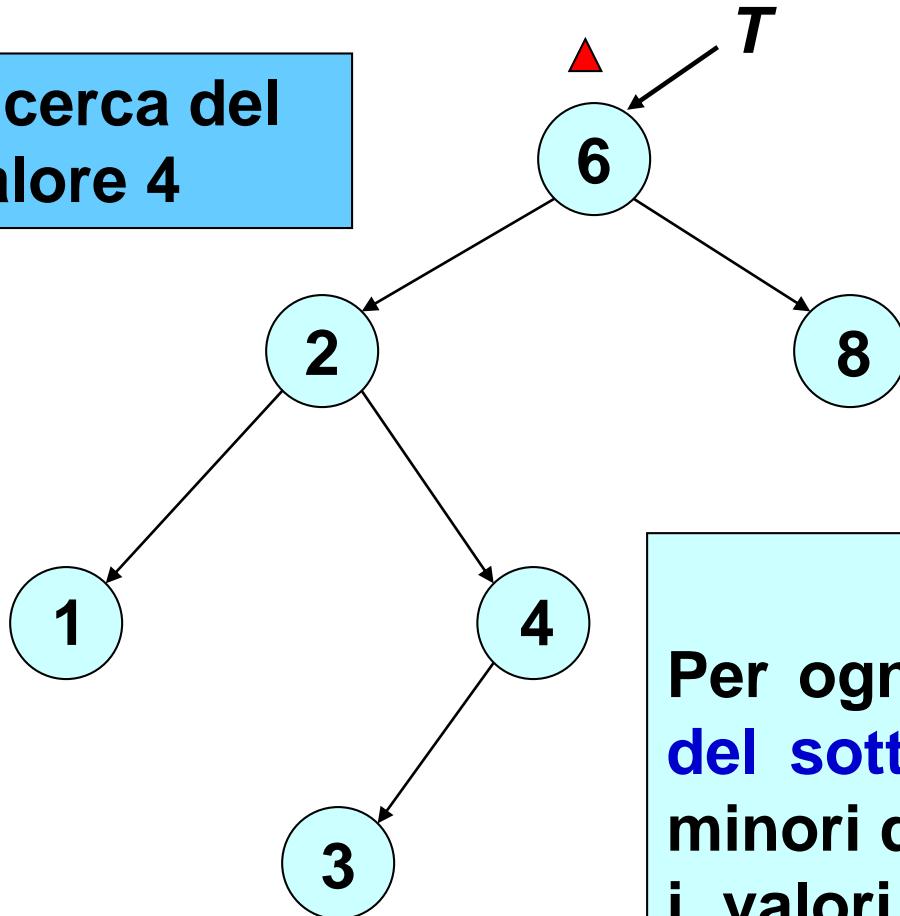
Assumiamo che i valori nei nodi (le chiavi) possano essere ordinati.

Proprietà degli ABR

Per ogni nodo X , i valori nei nodi del sottoalbero sinistro sono tutti minori del valore nel nodo X , e tutti i valori nei nodi del sotto-albero destro sono maggiori del valore di X

Alberi binari di ricerca: esempio

Ricerca del
valore 4

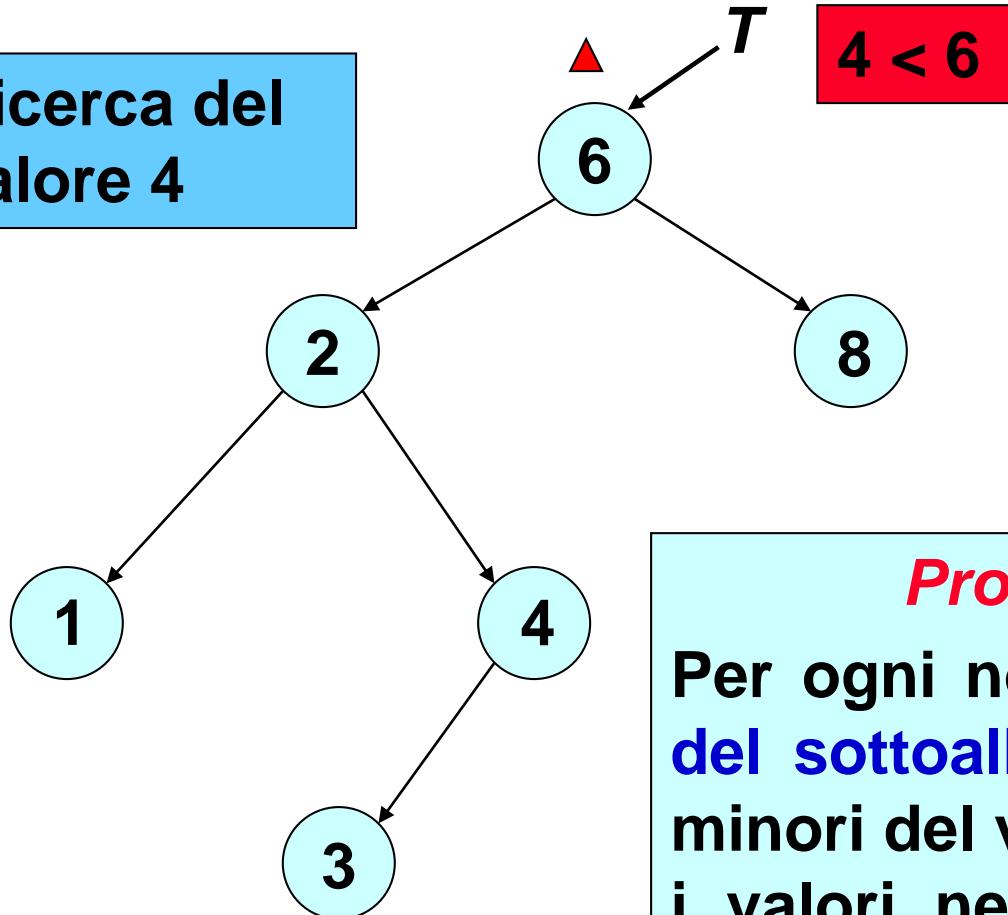


Proprietà degli ABR

Per ogni nodo **X**, i valori nei **nodi del sottoalbero sinistro** sono tutti minori del valore nel nodo **X**, e tutti i valori nei **nodi del sotto-albero destro** sono maggiori del valore di **X**

Alberi binari di ricerca: esempio

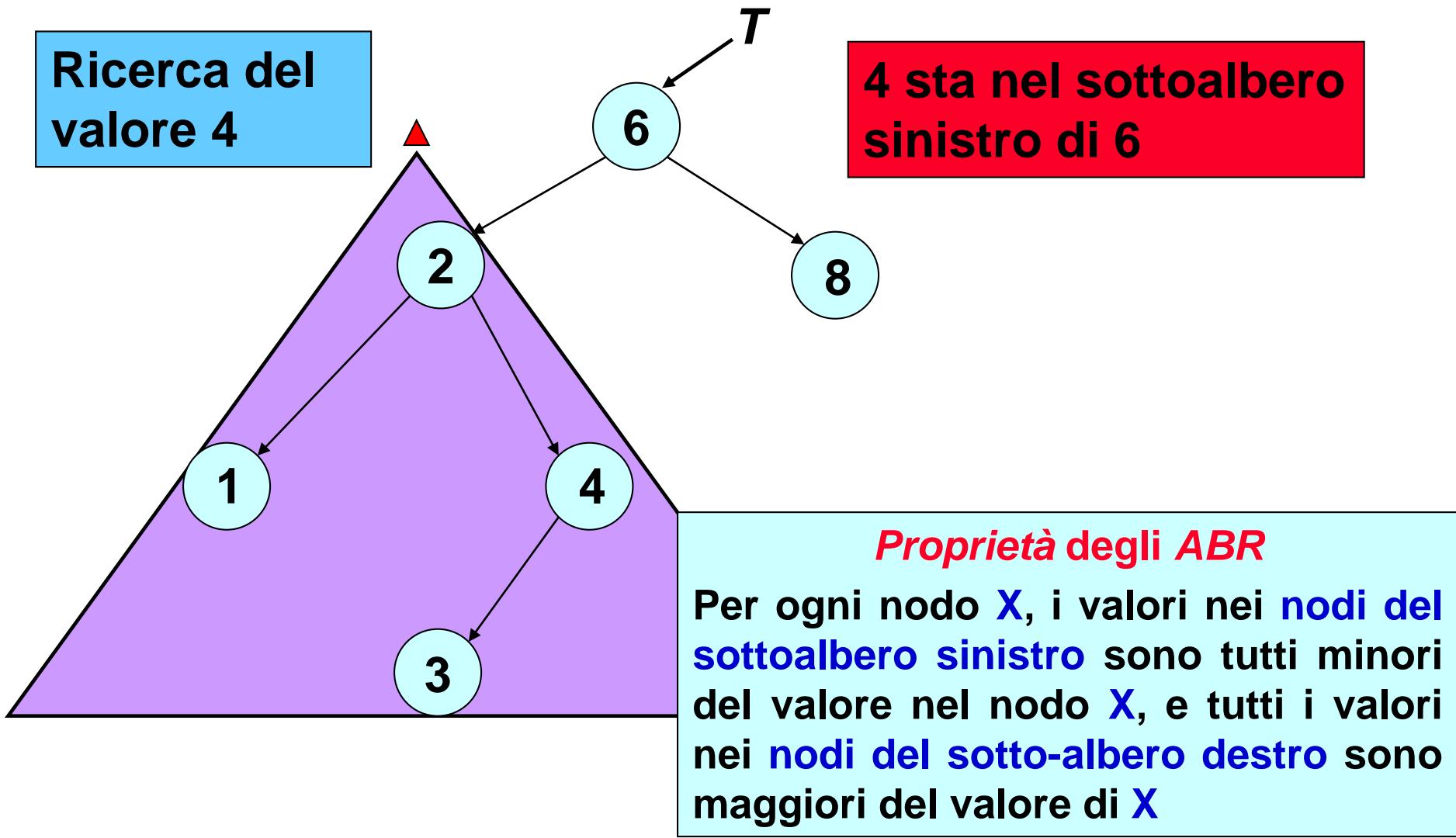
Ricerca del
valore 4



Proprietà degli ABR

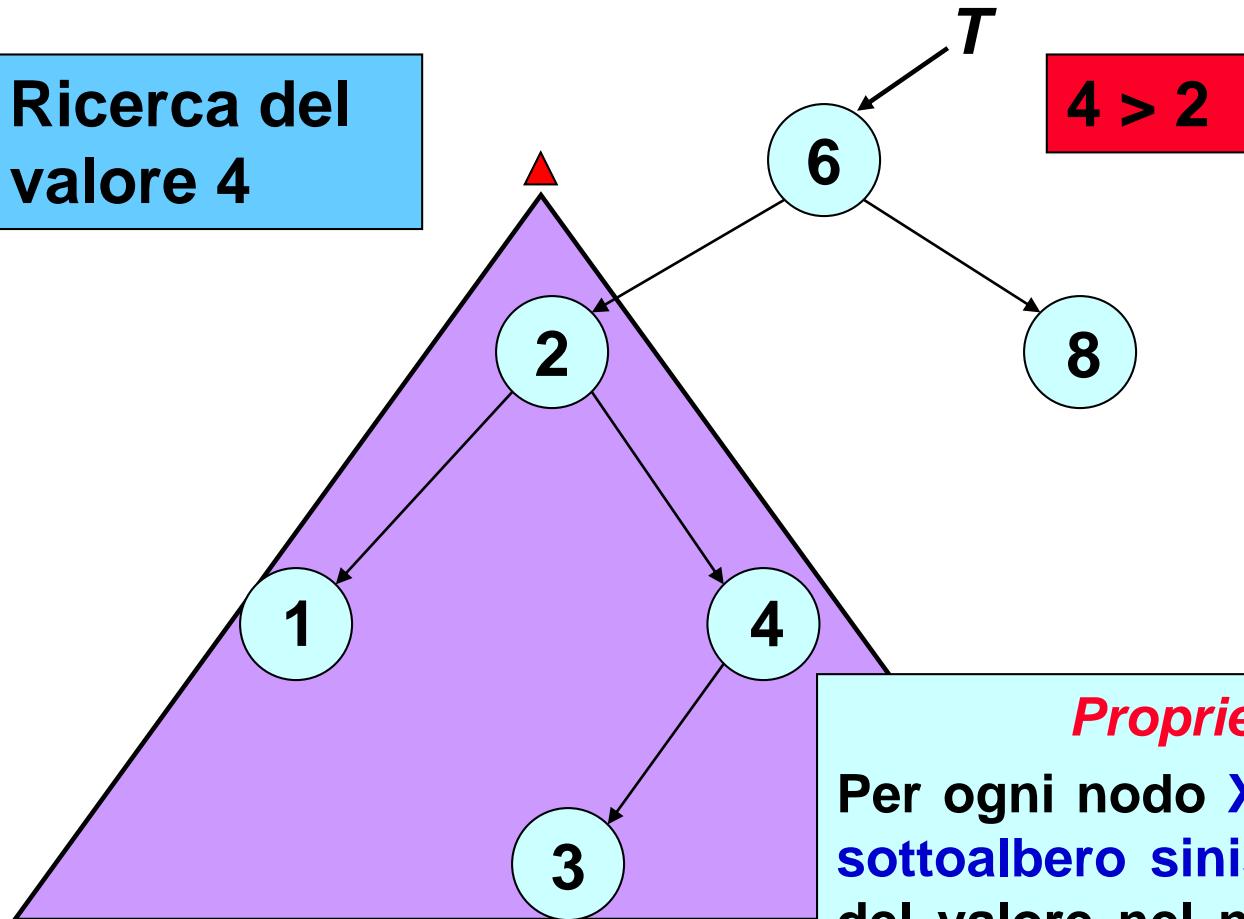
Per ogni nodo X , i valori nei nodi del sottoalbero sinistro sono tutti minori del valore nel nodo X , e tutti i valori nei nodi del sotto-albero destro sono maggiori del valore di X

Alberi binari di ricerca: esempio



Alberi binari di ricerca: esempio

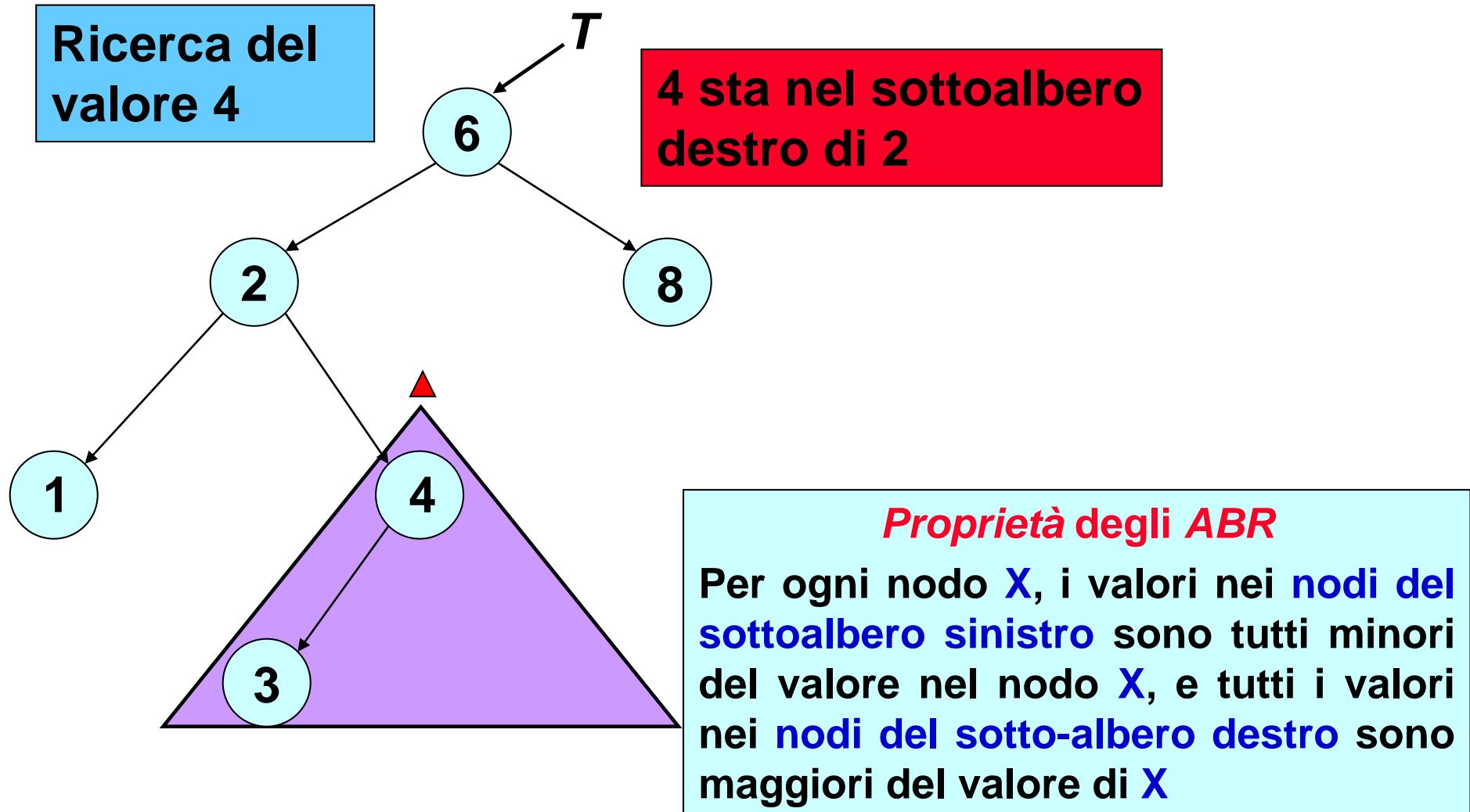
Ricerca del
valore 4



Proprietà degli ABR

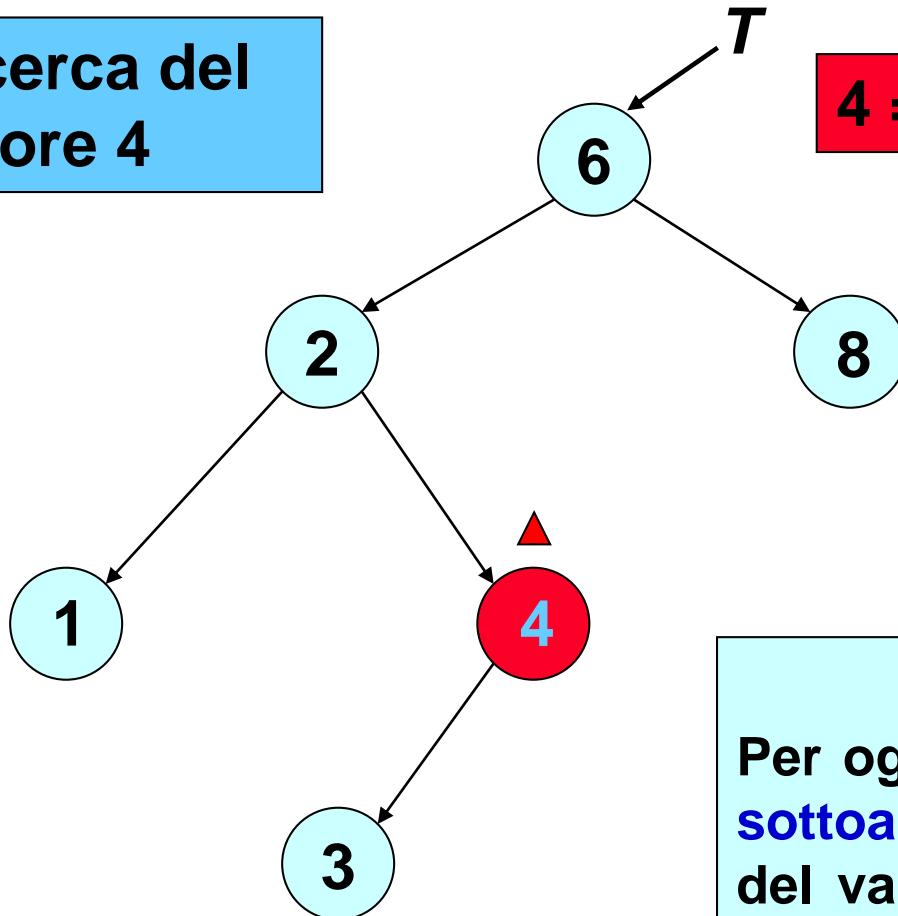
Per ogni nodo X , i valori nei nodi del sottoalbero sinistro sono tutti minori del valore nel nodo X , e tutti i valori nei nodi del sotto-albero destro sono maggiori del valore di X

Alberi binari di ricerca: esempio



Alberi binari di ricerca: esempio

Ricerca del
valore 4



4 = 4 : Trovato!

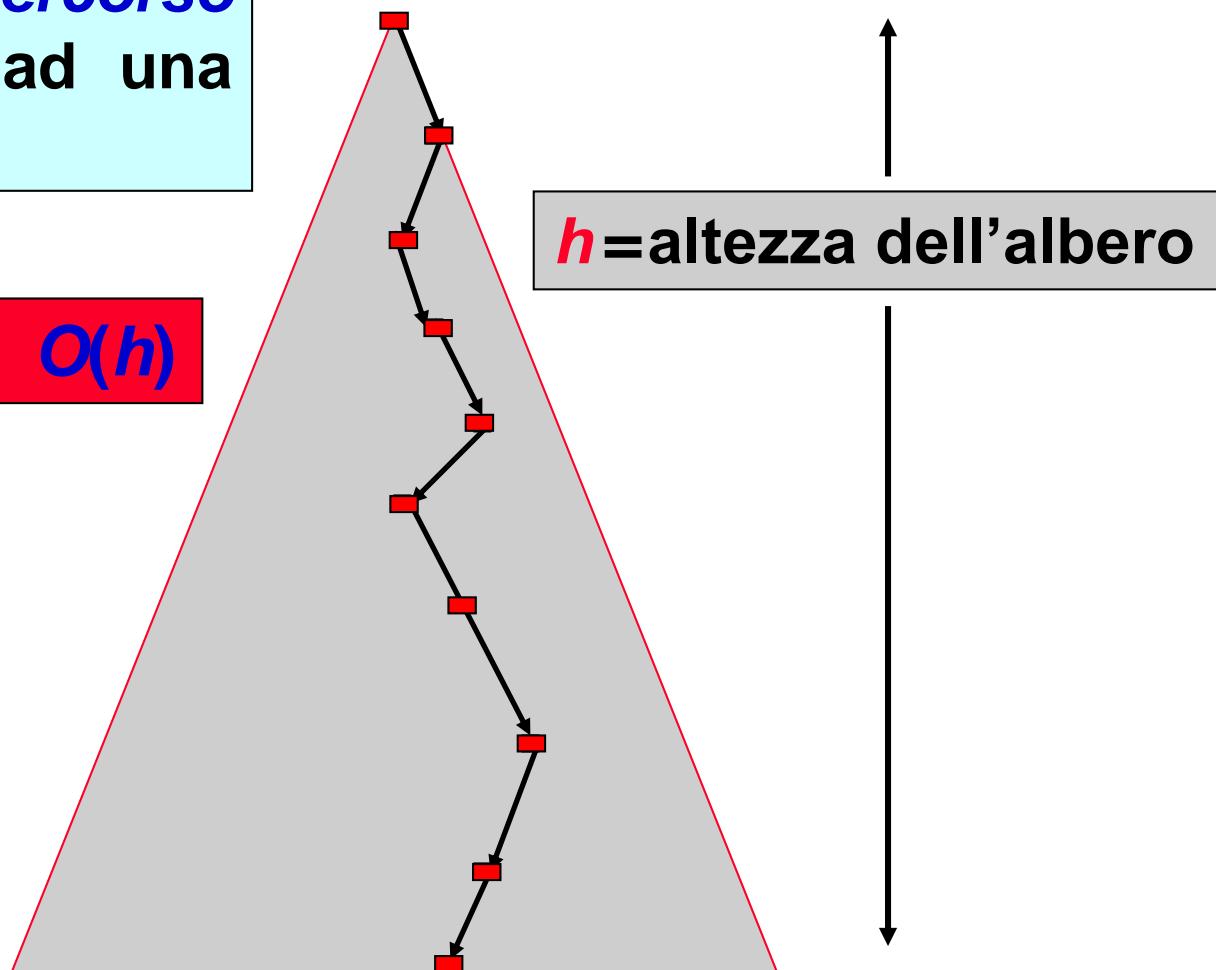
Proprietà degli ABR

Per ogni nodo X , i valori nei nodi del **sottoalbero sinistro** sono tutti minori del valore nel nodo X , e tutti i valori nei nodi del **sotto-albero destro** sono maggiori del valore di X

Alberi binari di ricerca

In generale, la *ricerca* è confinata ai *nodi* posizionati *lungo un singolo percorso* (path) dalla radice ad una foglia

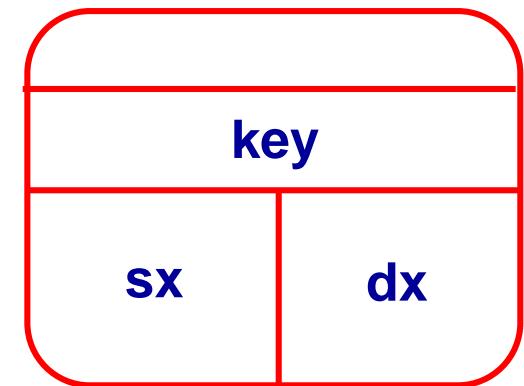
Tempo di ricerca = $O(h)$



ADT albero binario di ricerca: tipo di dato

- È una specializzazione dell'ADT albero binario
- Gli elementi statici sono essenzialmente gli stessi, l'unica differenza è che si assume che i dati contenuti (le chiavi) siano ordinabili secondo qualche relazione d'ordine.

```
typedef *nodo ARB;  
struct {  
    elemento key;  
    ARB sx, dx;  
} nodo;
```



ADT albero binario di ricerca: funzioni

➤ **Selettori:**

- $\text{root}(T)$
- $\text{dx}(T)$
- $\text{sx}(T)$
- $\text{key}(T)$

➤ **Costruttori/Distruttori:**

- $\text{crea_albero}()$
- $\text{ARB_inserisci}(T,x)$
- $\text{ARB_cancella } (T,x)$

➤ **Proprietà:**

- $\text{vuoto}(T) = \text{return } (T=\text{Nil})$

➤ **Operazioni di Ricerca**

- $\text{ARB_ricerca}(T,k)$
- $\text{ARB_minimo}(T)$
- $\text{ARB_massimo}(T)$
- $\text{ARB_successore}(T,x)$
- $\text{ARB_predecessore}(T,x)$

Ritorna il valore
del test di
uguaglianza

Ricerca in Alberi binari di ricerca

```
ARB_ricerca(T, k)
  IF T ≠ NIL THEN
    IF k ≠ T->Key THEN
      IF k < T->Key THEN
        return ARB_ricerca(T->sx, k)
      ELSE
        return ARB_ricerca(T->dx, k)
  return T
```

NOTA: Questo algoritmo **cerca il nodo con chiave *k*** nell'albero *T* e ne **ritorna il puntatore**. Ritorna NIL nel caso non esista alcun nodo con chiave ***k***.

Ricerca in Alberi binari di ricerca

```
ARB_ricerca' (T, k)
  IF T = NIL OR k = T->Key THEN
    return T
  ELSE IF k < T->Key THEN
    return ARB_ricerca' (T->sx, k)
  ELSE
    return ARB_ricerca' (T->dx, k)
```

NOTA: *Variante sintattica del precedente algoritmo!*

Ricerca in Alberi binari di ricerca

```
ARB_ricerca(T, k)
    cur = T
    while cur ≠ NIL AND k ≠ cur->Key THEN
        IF k < T->Key THEN
            cur = cur->sx
        ELSE
            cur = cur->dx
    return cur
```

NOTA: *Versione iterativa!*

Ricerca in Alberi binari di ricerca

In generale, la **ricerca** è confinata ai **nodi** posizionati **lungo un singolo percorso (path)** dalla radice ad una foglia

Tempo di ricerca = **$O(h)$**

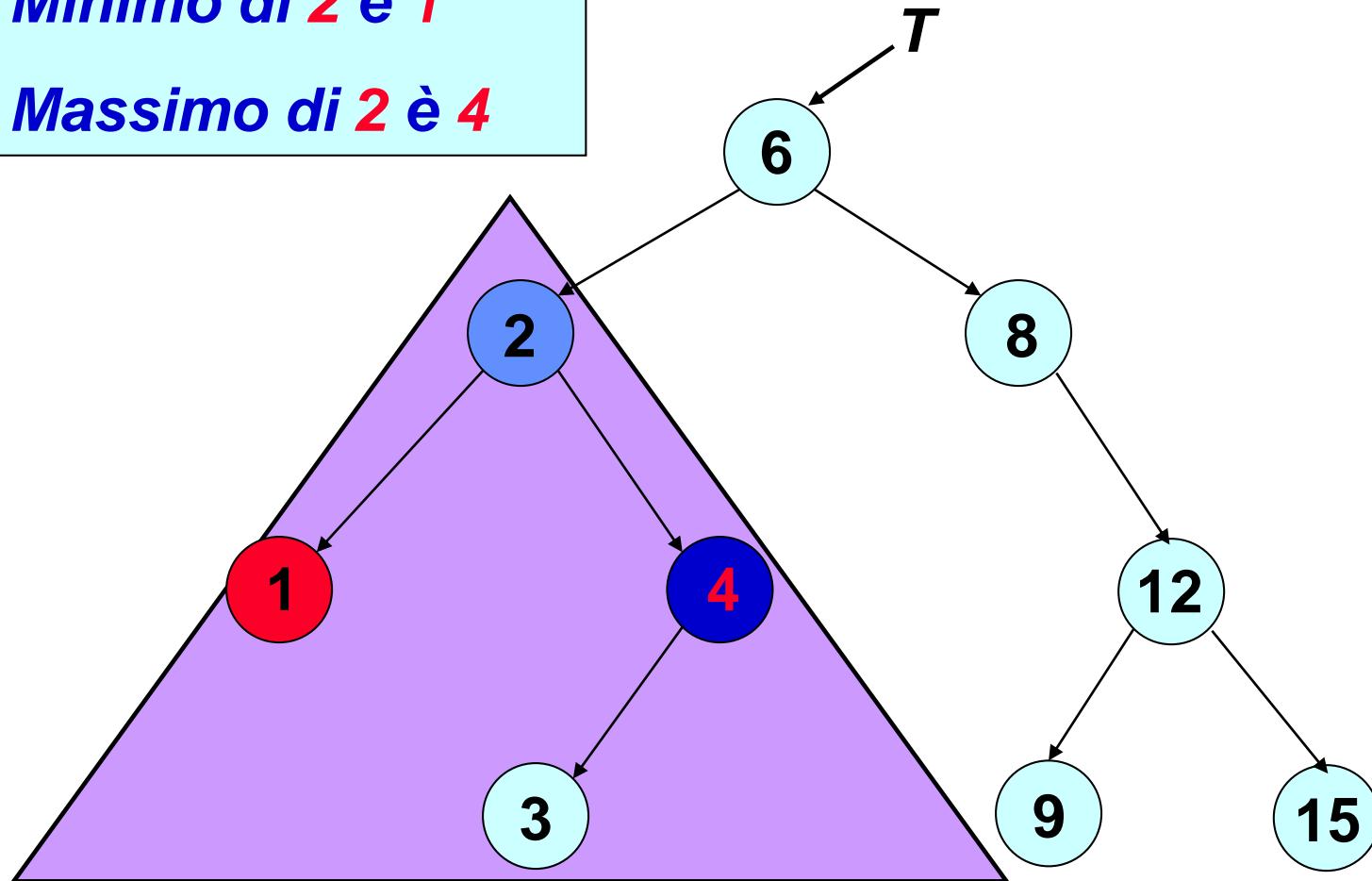
$O(h) = O(\log N)$, dove **N** è il numero di nodi nell'albero, solo se l'albero è **balanciato** (cioè la lunghezza del percorso minimo è vicino a quella del percorso massimo).



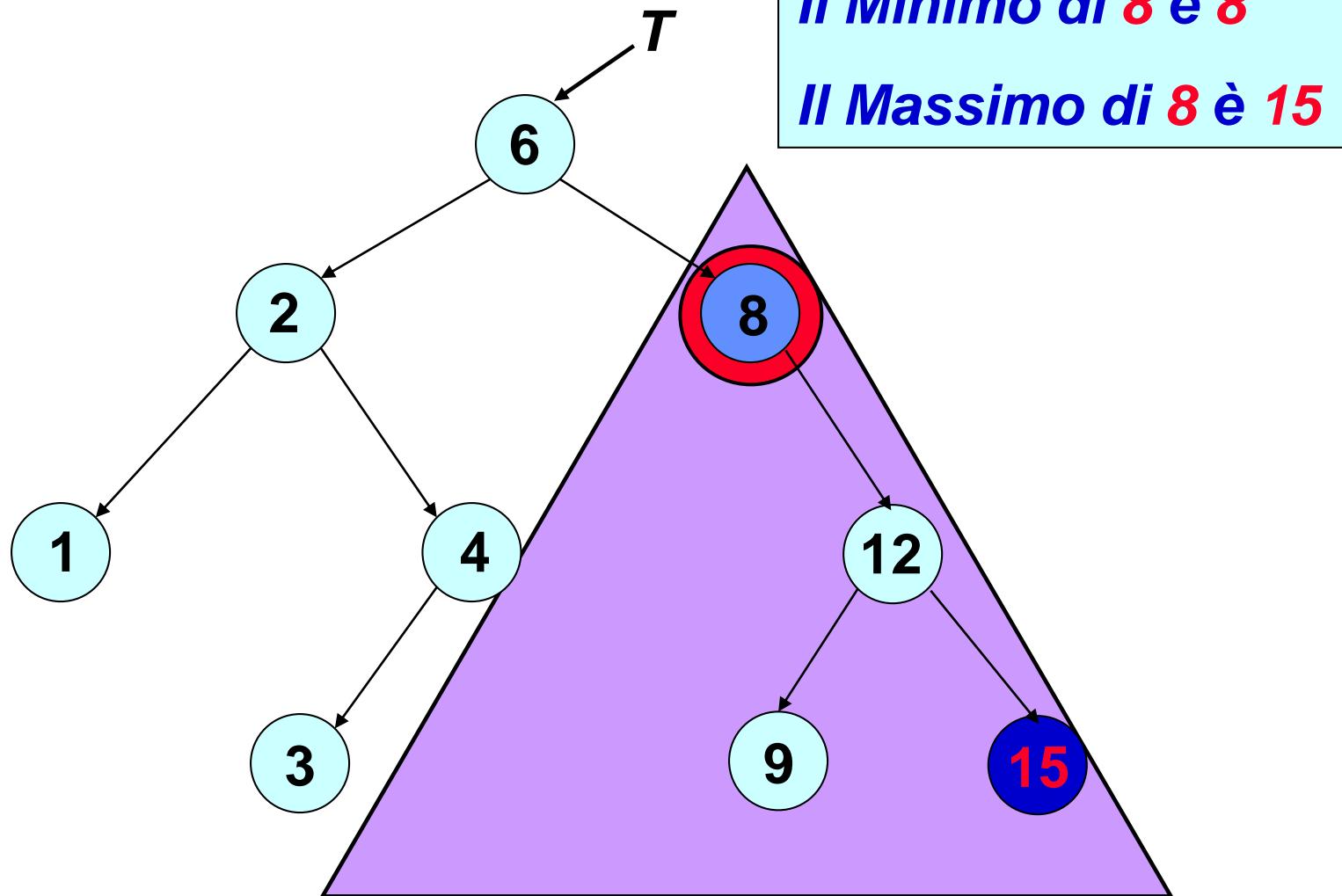
ARB: ricerca del minimo e massimo

Il Minimo di 2 è 1

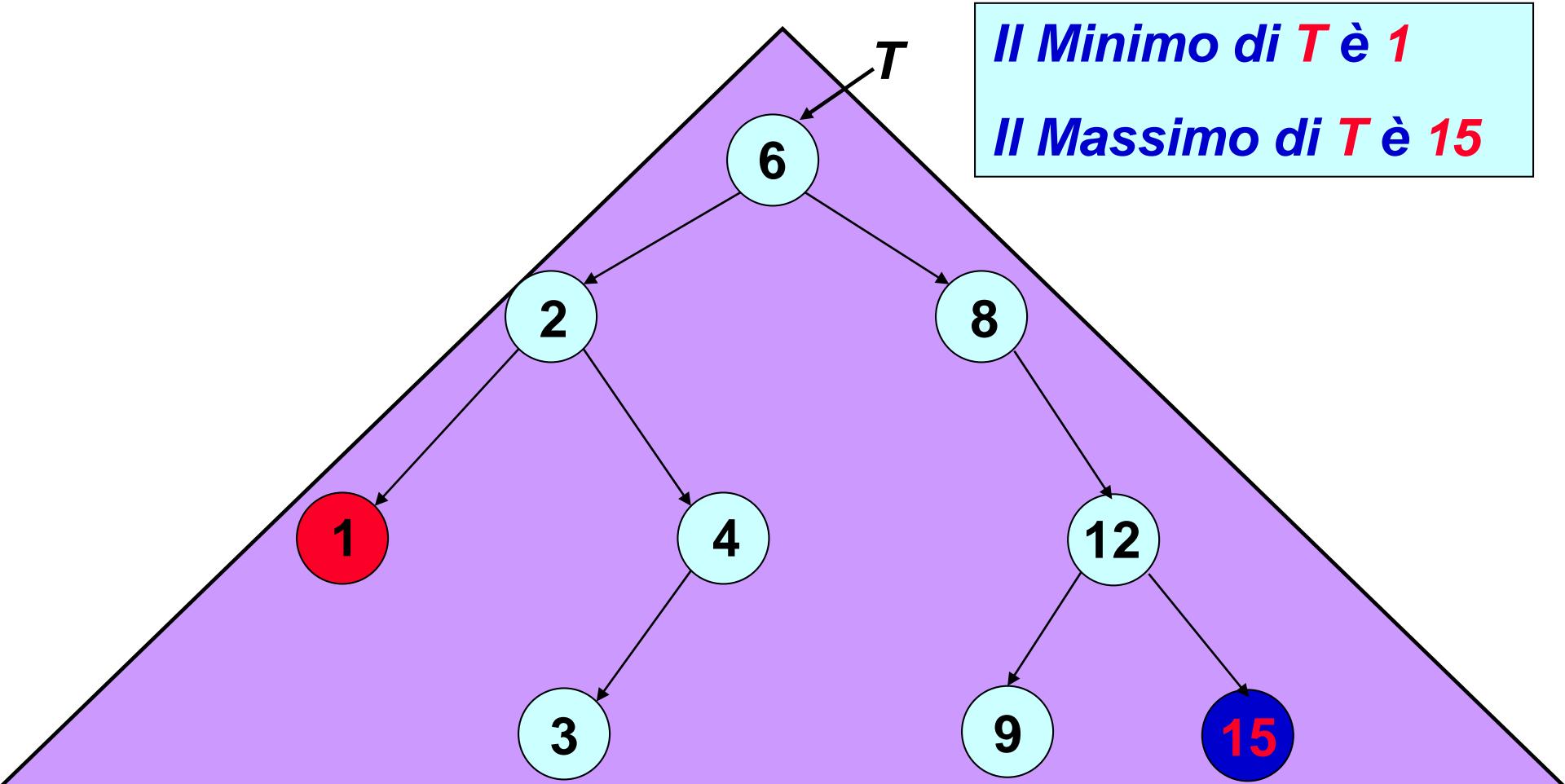
Il Massimo di 2 è 4



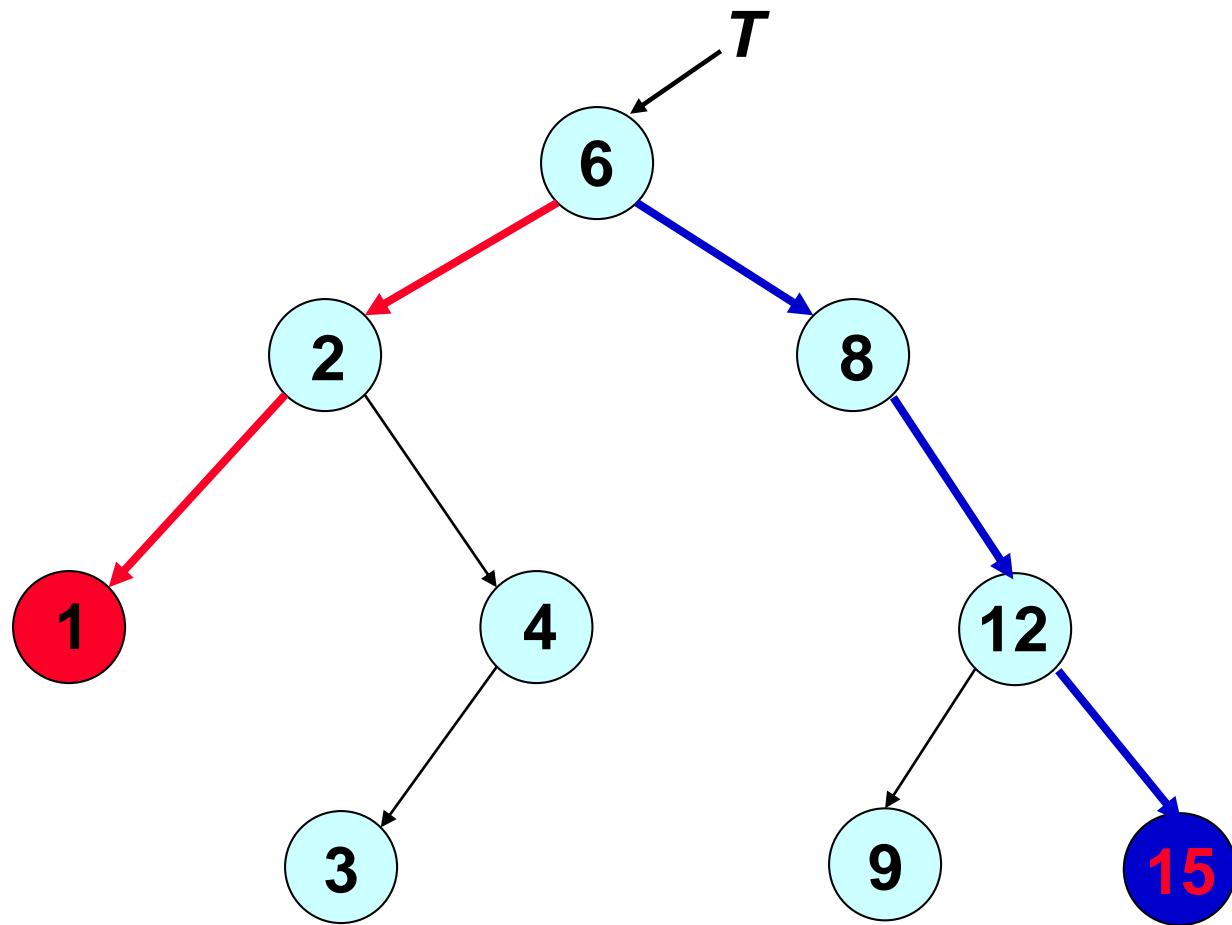
ARB: ricerca del minimo e massimo



ARB: ricerca del minimo e massimo



ARB: ricerca del minimo e massimo



ARB: ricerca del minimo e massimo

```
ARB ABR-Minimo (x:ARB)
```

```
  WHILE x->sx ≠ NIL DO
```

```
    x = x->sx
```

```
  return x
```

```
ARB ABR-Massimo (x: ARB)
```

```
  WHILE x->dx ≠ NIL DO
```

```
    x = x->dx
```

```
  return x
```

ARB: ricerca del minimo e massimo

```
ARB ABR-Minimo (x:ARB)
```

```
  WHILE x->sx ≠ NIL DO  
    x = x->sx  
  return x
```

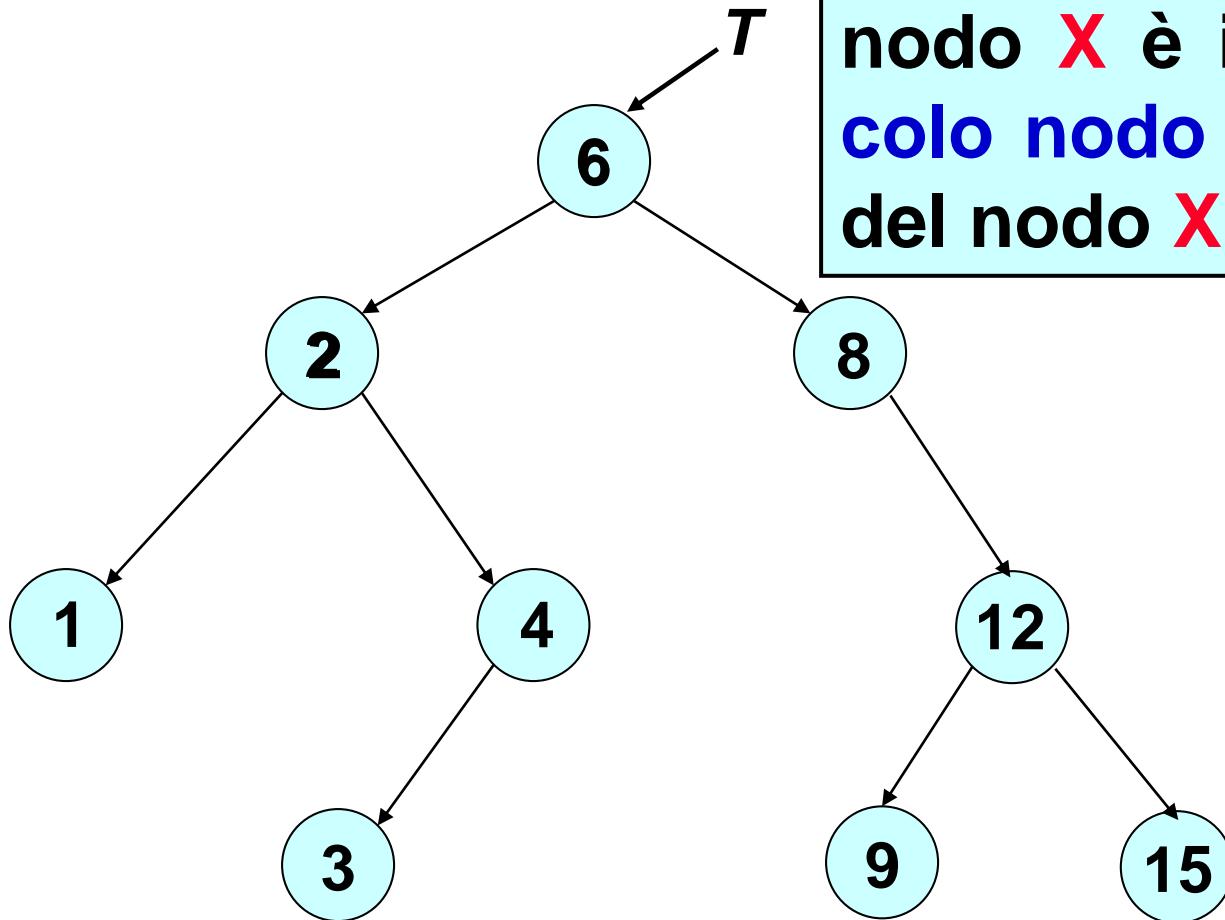
```
ARB ABR-Massimo (x: ARB)
```

```
  WHILE x->dx ≠ NIL DO  
    x = x->dx  
  return x
```

```
ARB ARB_Minimo (x:ARB)
```

```
  IF x ≠ NIL AND x->sx ≠ NIL THEN  
    return ARB_Minimo (x->sx)  
  return x
```

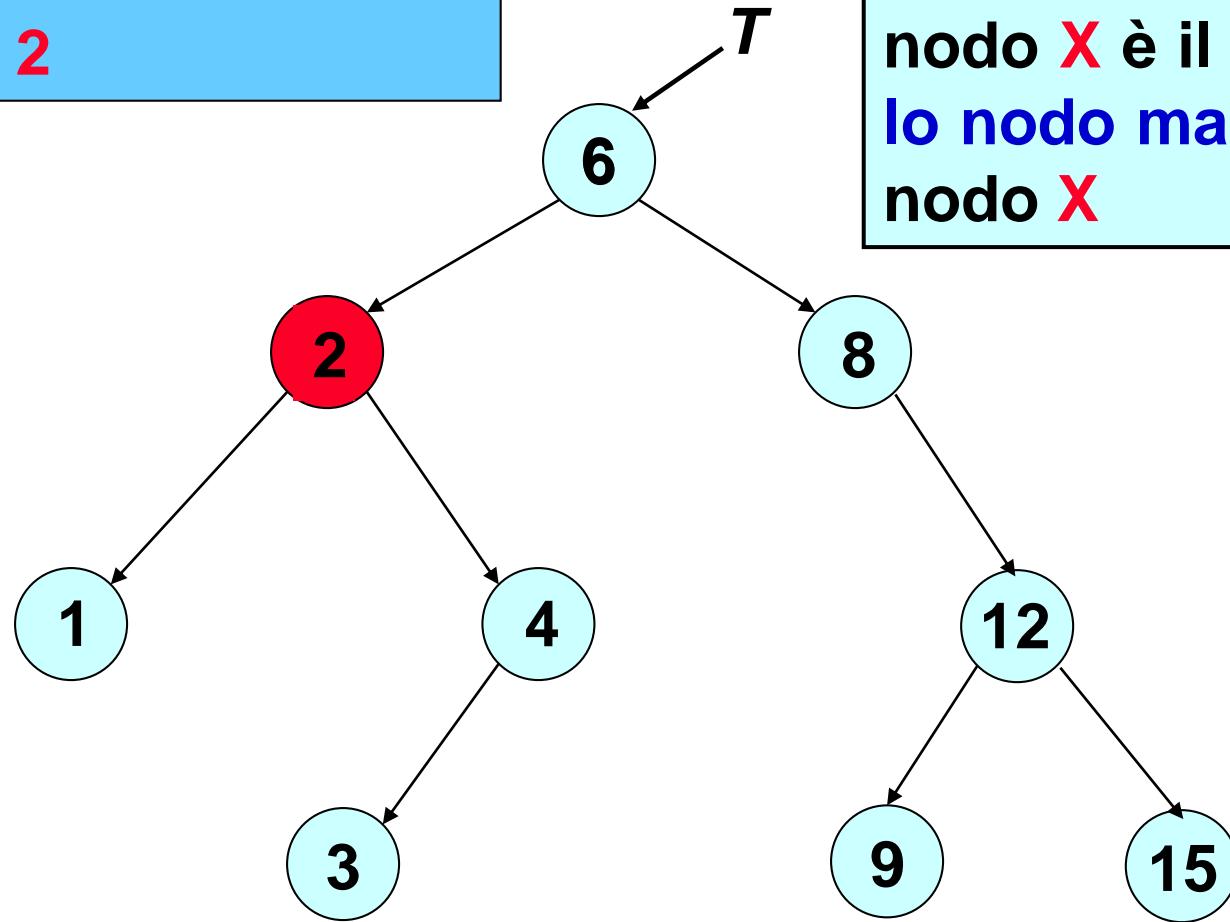
ARB: ricerca del successore



Il **successore** di un nodo **X** è il più piccolo nodo maggiore del nodo **X**

ARB: ricerca del successore

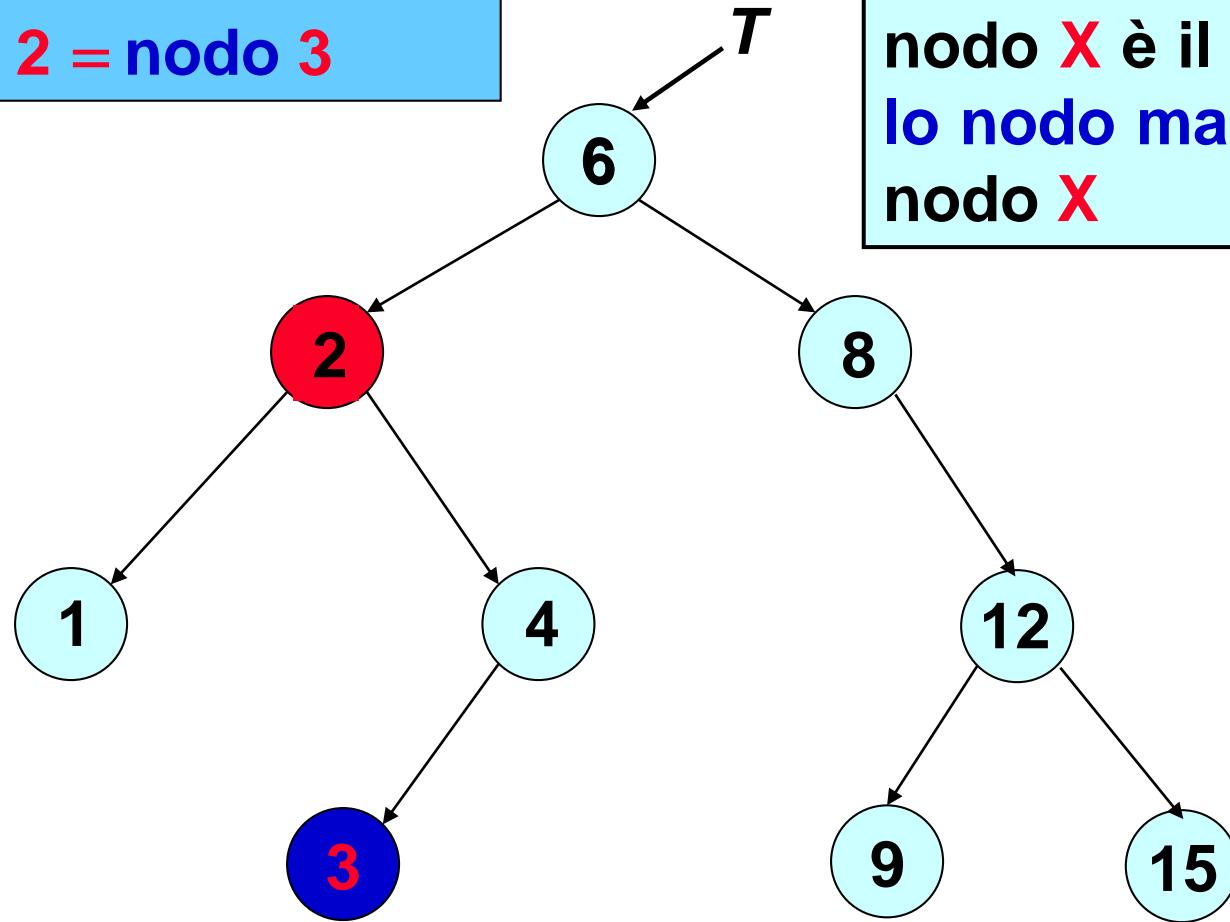
**Ricerca del successore
del nodo 2**



**Il successore di un
nodo X è il più picco-
lo nodo maggiore del
nodo X**

ARB: ricerca del successore

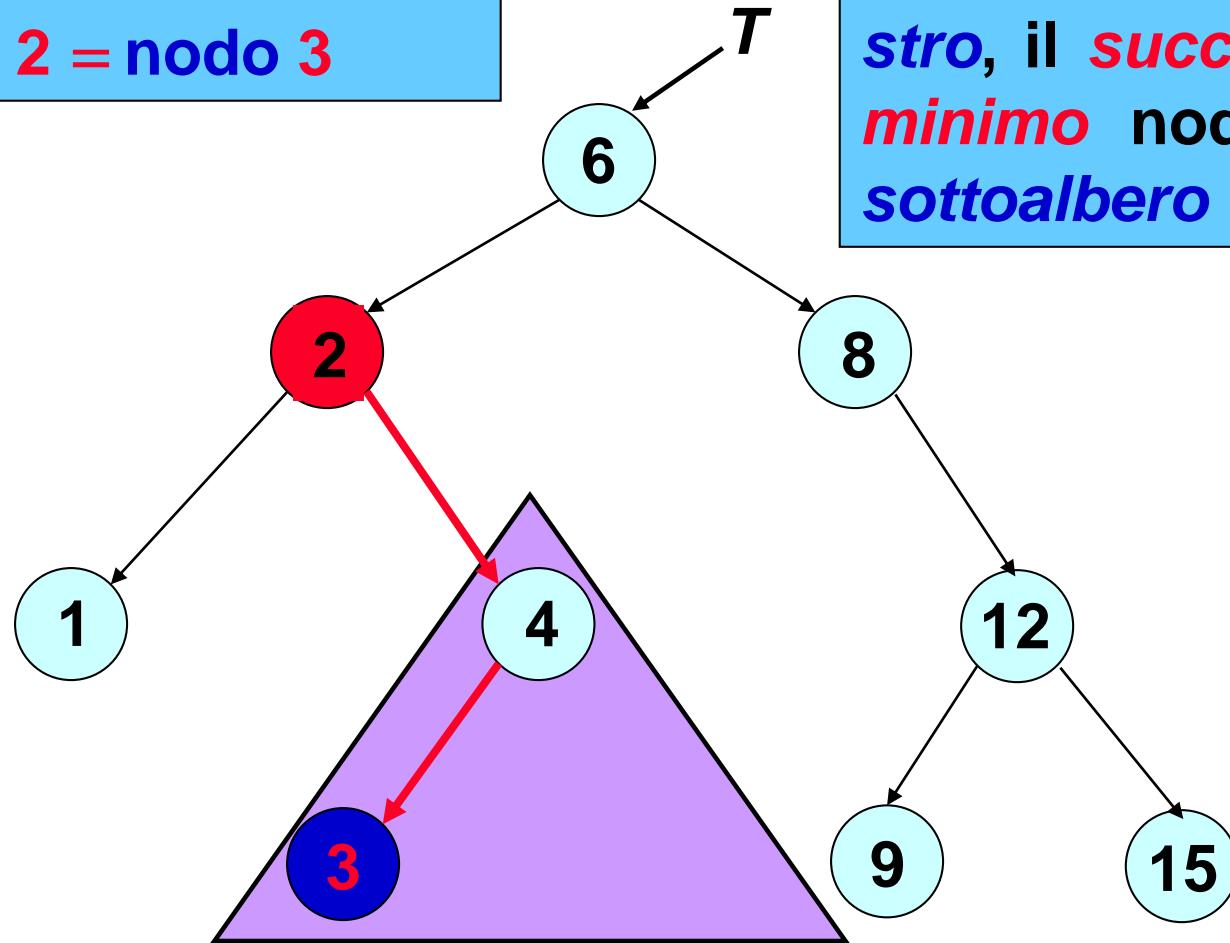
**Ricerca del successore
del nodo 2 = nodo 3**



**Il successore di un
nodo X è il più picco-
lo nodo maggiore del
nodo X**

ARB: ricerca del successore

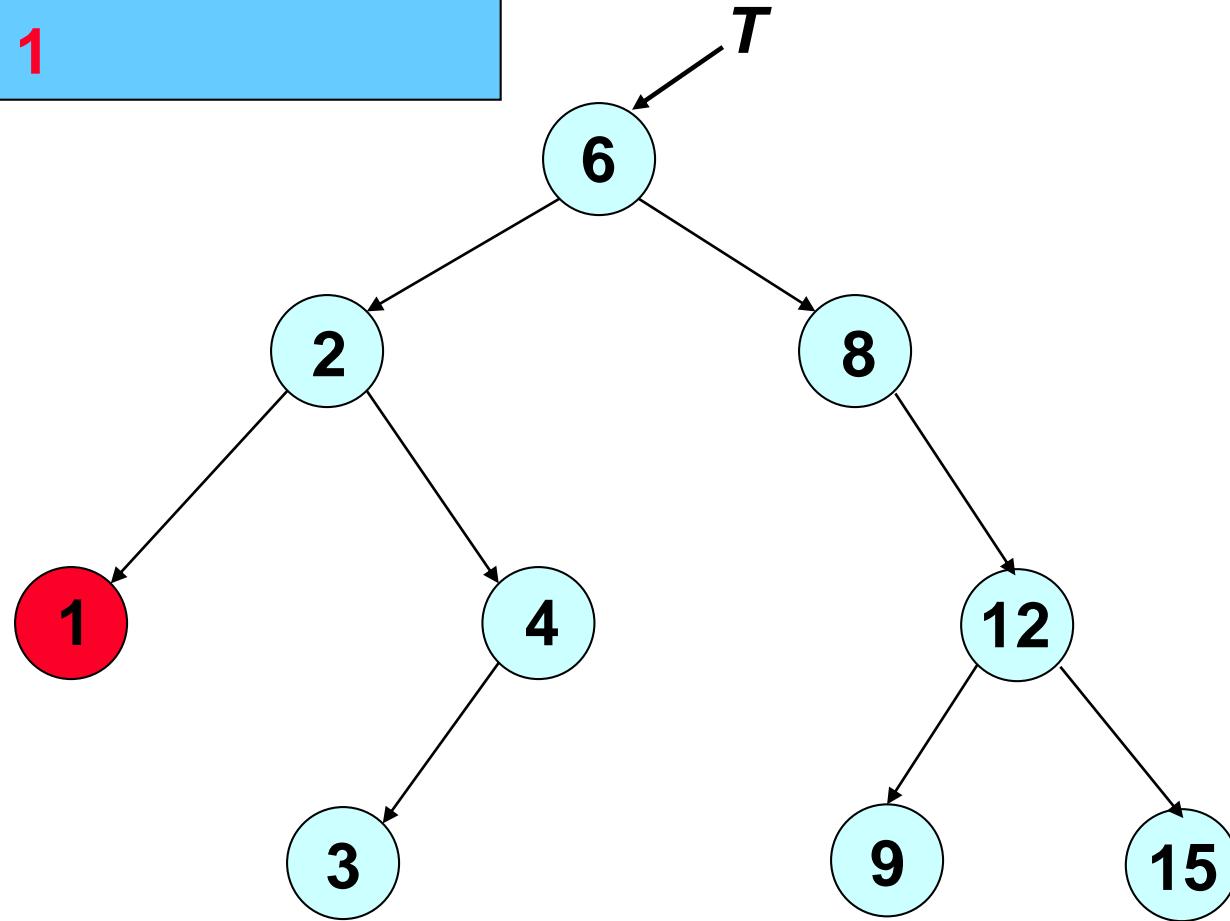
Ricerca del successore
del nodo 2 = nodo 3



Se x ha un figlio destro, il successore è il minimo nodo di quel sottoalbero

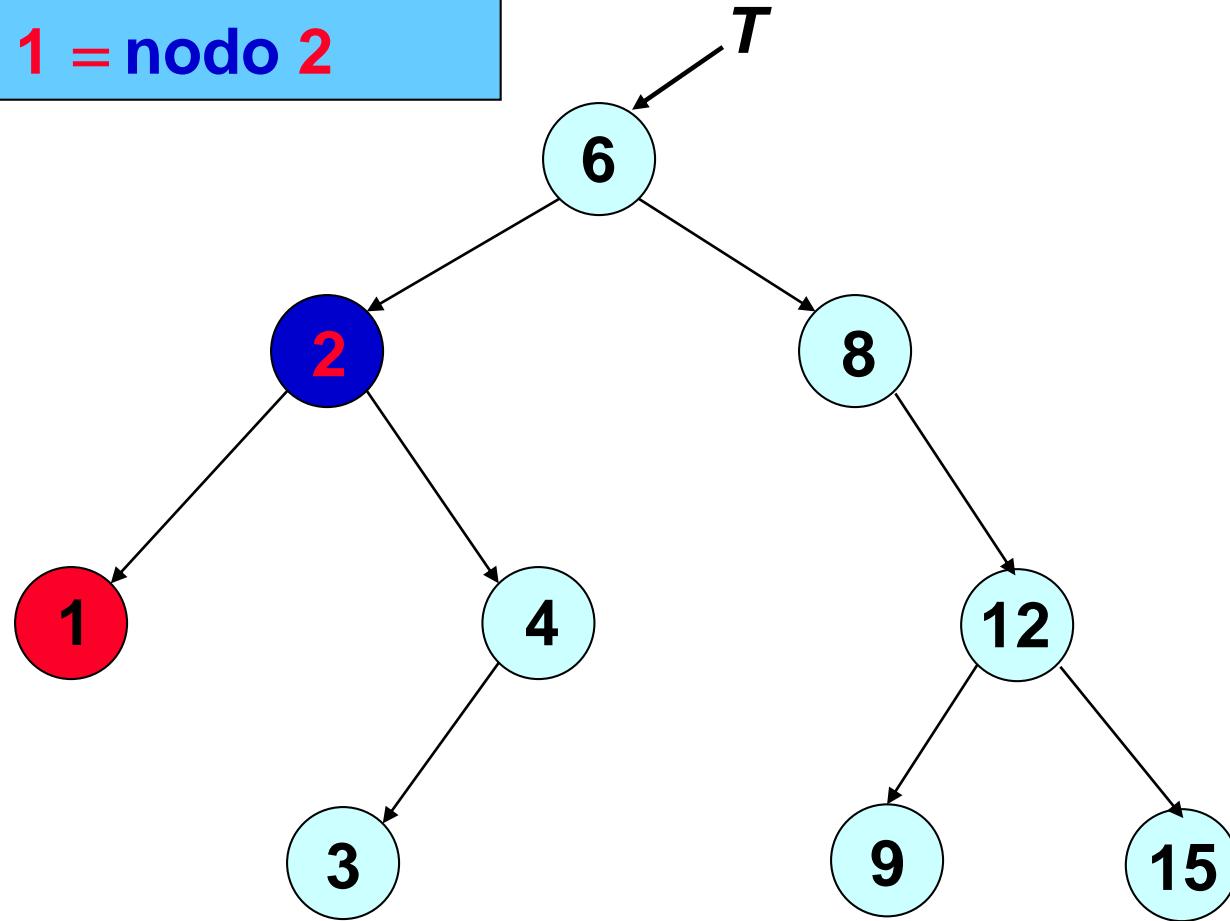
ARB: ricerca del successore

**Ricerca del successore
del nodo 1**



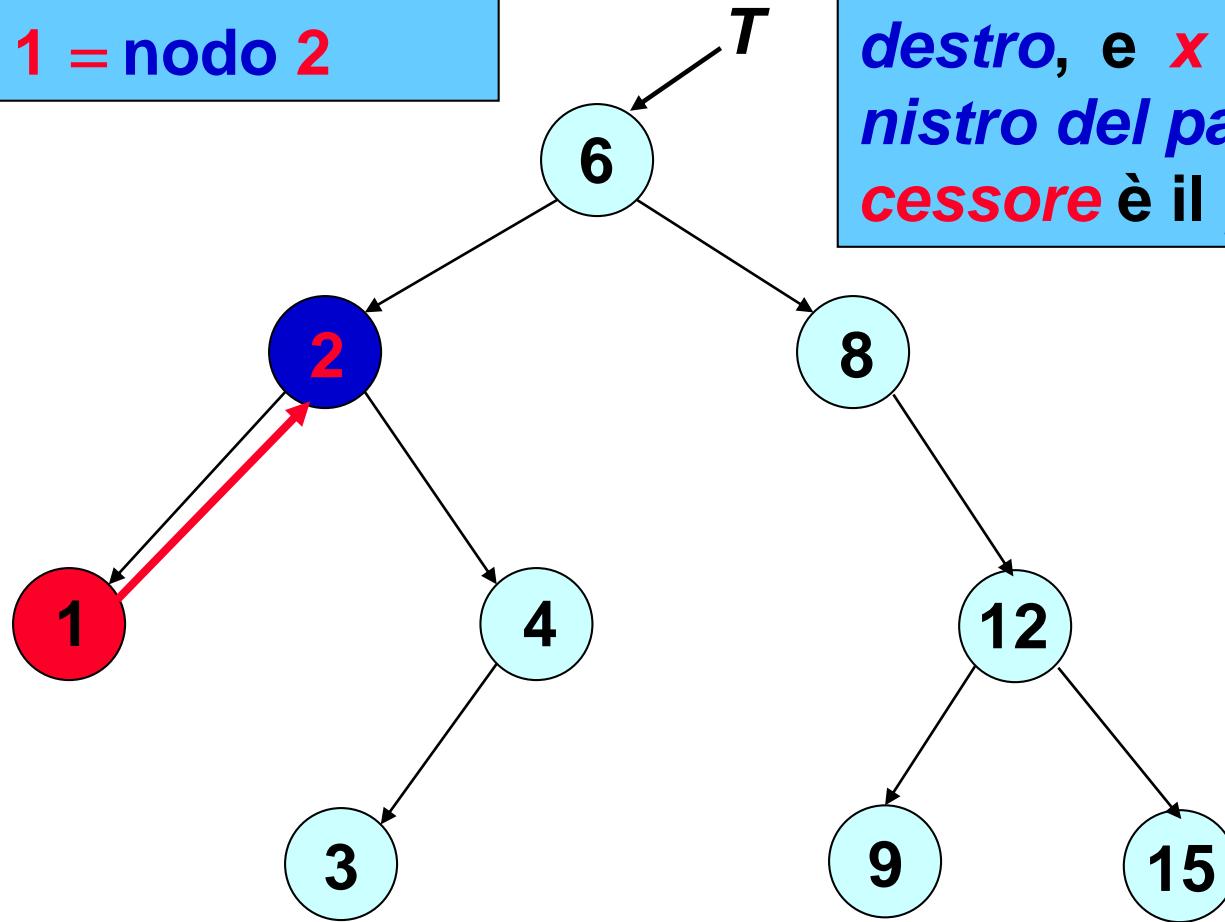
ARB: ricerca del successore

**Ricerca del successore
del nodo 1 = nodo 2**



ARB: ricerca del successore

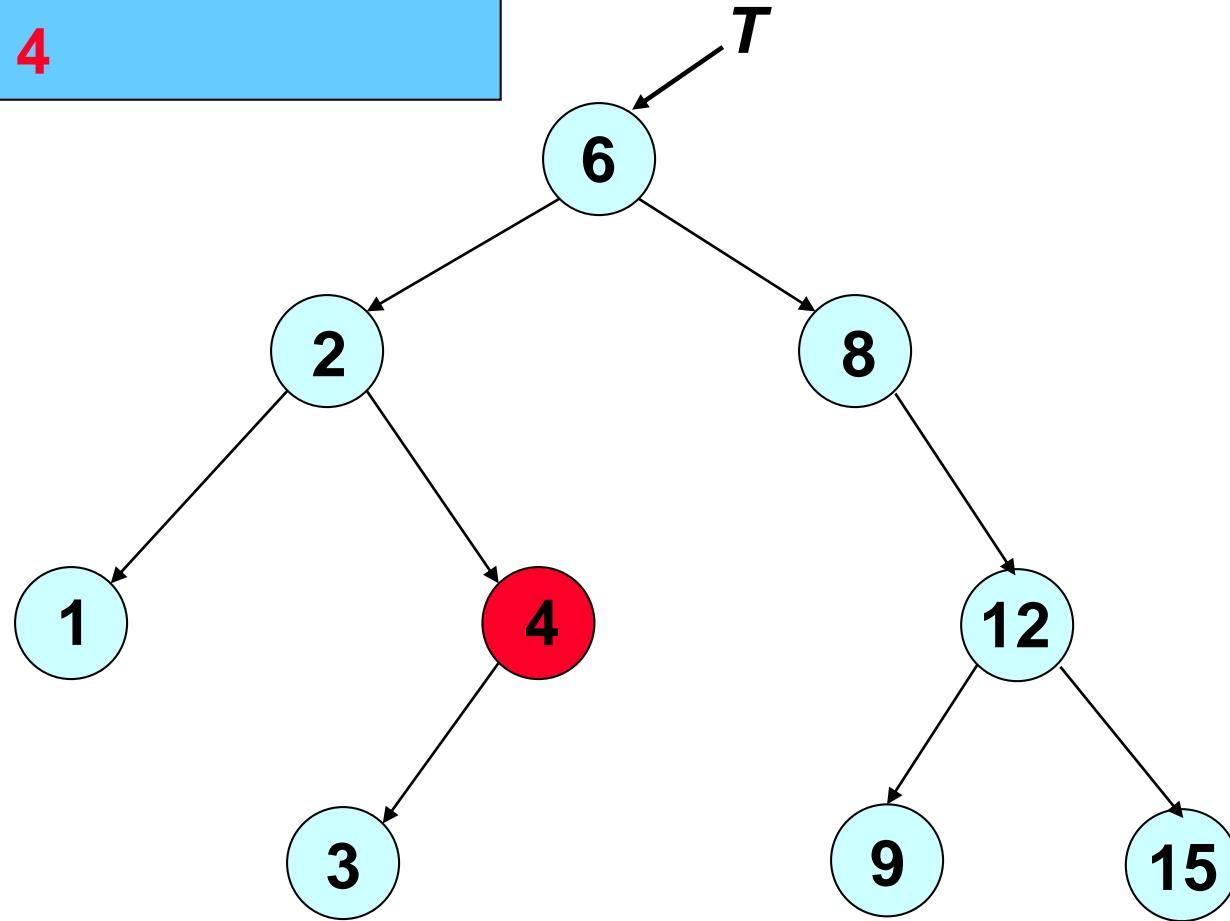
Ricerca del successore
del nodo 1 = nodo 2



Se x NON ha un *figlio destro*, e x è *figlio sinistro del padre*, il **successore** è il *padre*.

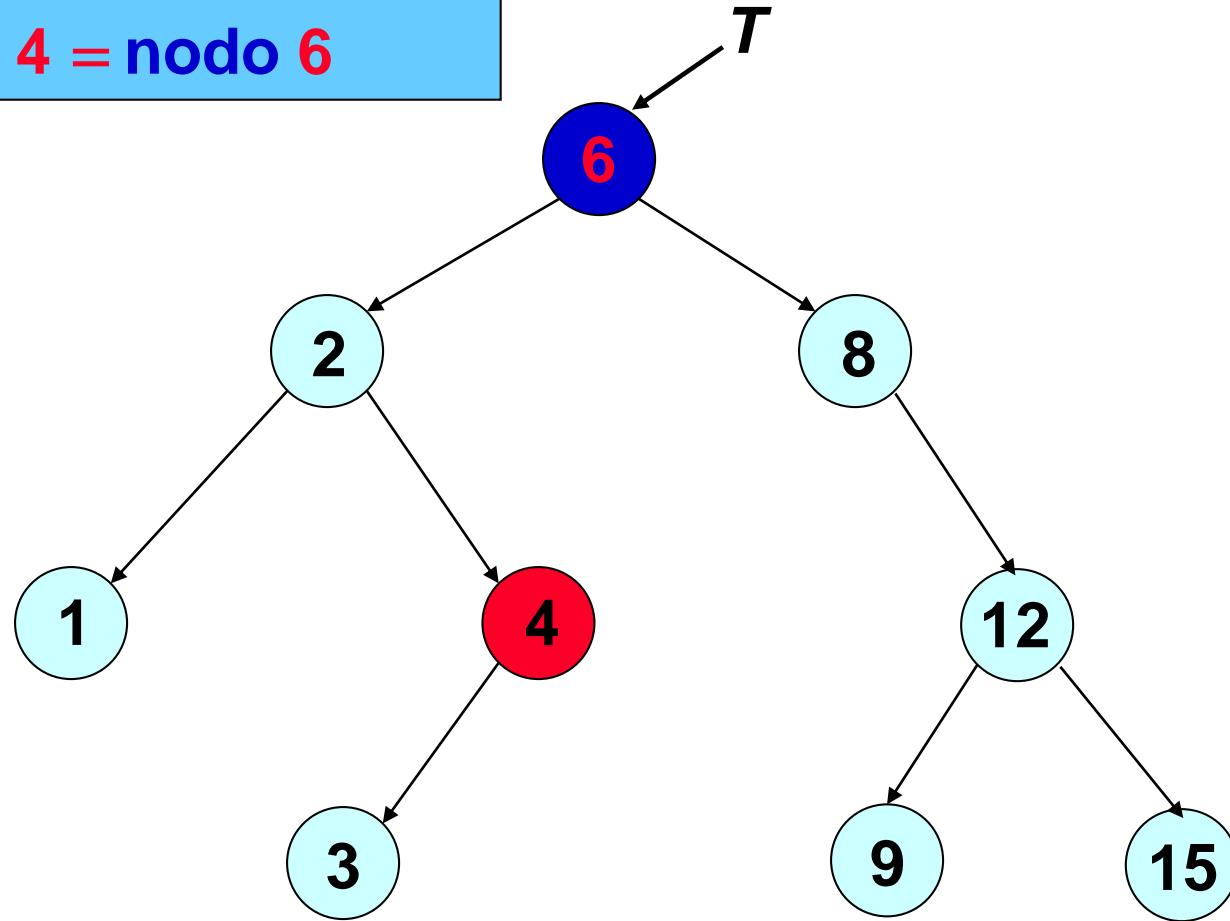
ARB: ricerca del successore

Ricerca del successore
del nodo 4



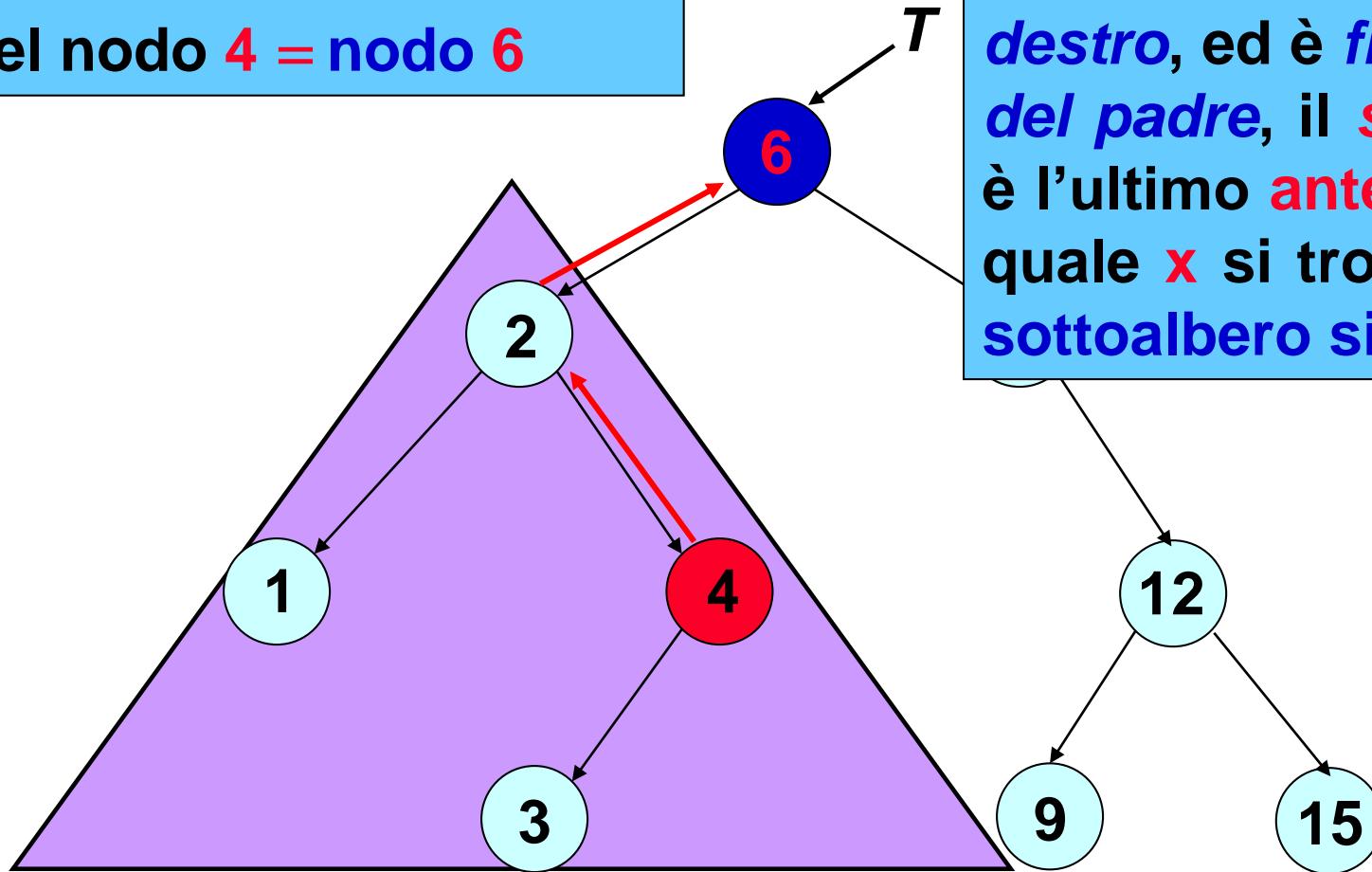
ARB: ricerca del successore

**Ricerca del successore
del nodo 4 = nodo 6**



ARB: ricerca del successore

Ricerca del successore
del nodo 4 = nodo 6



Se x NON ha un *figlio destro*, ed è *figlio destro del padre*, il *successore* è l'ultimo *antenato* per il quale x si trova nel suo sottoalbero sinistro.

ARB: ricerca del successore

ABR-Successore(T, k)

```
Z = T
P = NIL
WHILE (Z ≠ NIL && Z->key ≠ k)
    P = T
    IF (Z->key < k) THEN
        Z = Z->dx
    ELSE
        Z = Z->sx
    IF (Z ≠ NIL && Z->dx ≠ NIL) THEN
        return ABR-Minimo(Z->dx)
ELSE
```

```
WHILE (P ≠ NIL && (Z ≠ P->sx || P->key < k)) DO
    Z = P
    P = Z->padre
return P
```

Se Z NON ha un *figlio destro*, ed è *figlio destro del padre*, il *successore* è l'ultimo antenato per il quale Z si trova nel suo sottoalbero sinistro.

ARB: ricerca del successore

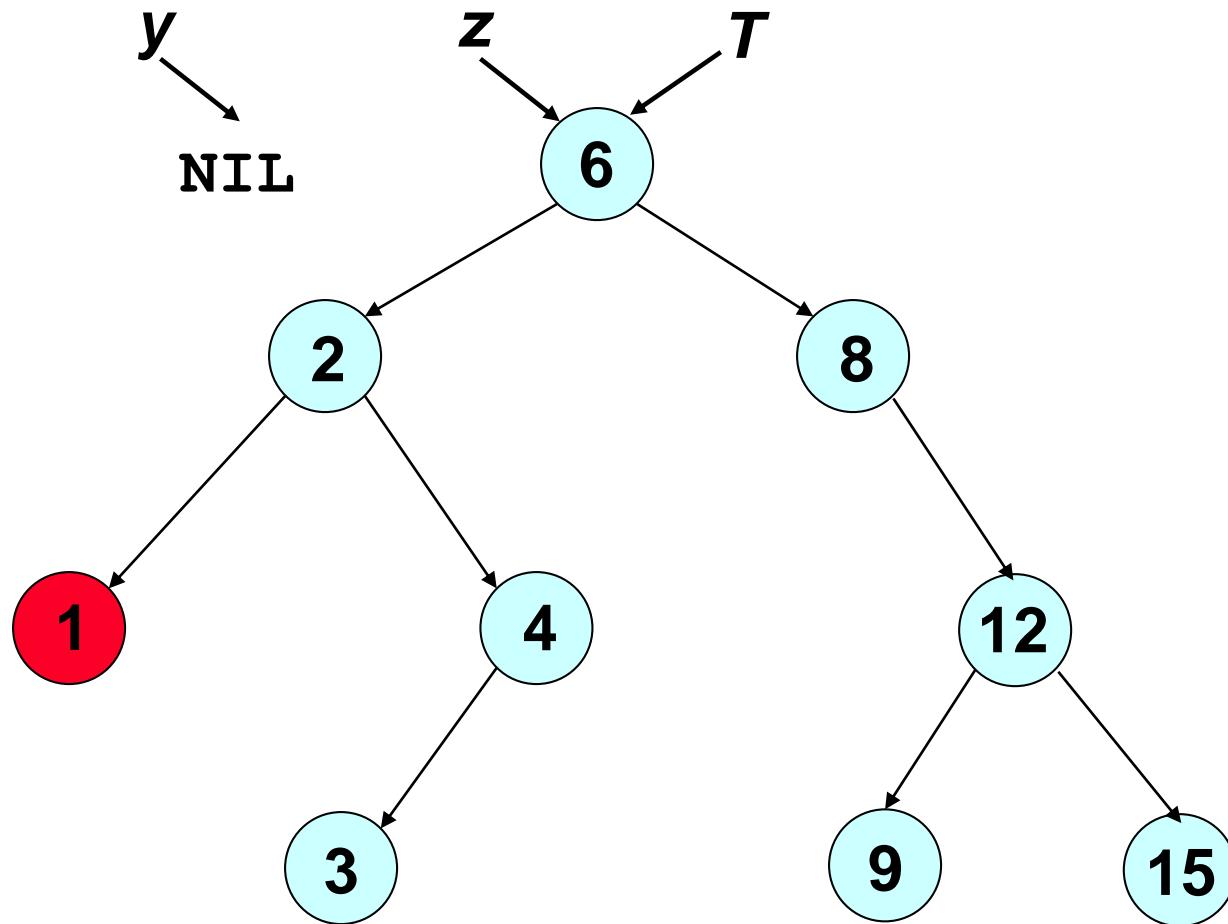
ABR-Successore (T, k)

```
Z = T
P = NIL
WHILE (Z ≠ NIL && Z->key ≠ k)
    P = T
    IF (Z->key < k) THEN
        Z = Z->dx
```

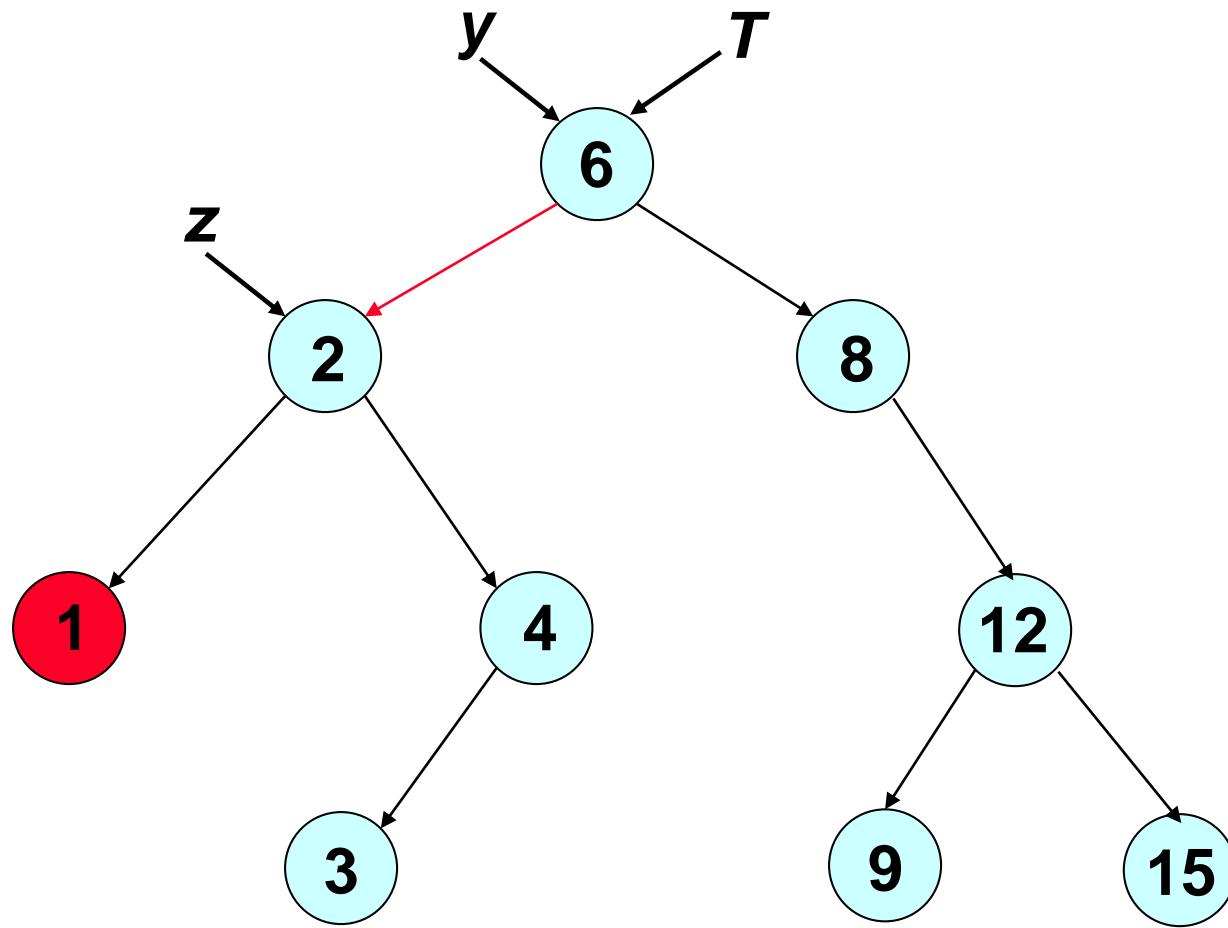
Questo algoritmo **assume** che ogni nodo abbia il puntatore al padre

```
IF (Z ≠ NIL && Z->dx ≠ NIL) THEN
    return ABR-Minimo (Z->dx)
ELSE
    WHILE (P ≠ NIL && (Z ≠ P->sx || P->key < k)) DO
        Z = P
        P = Z->padre
    return P
```

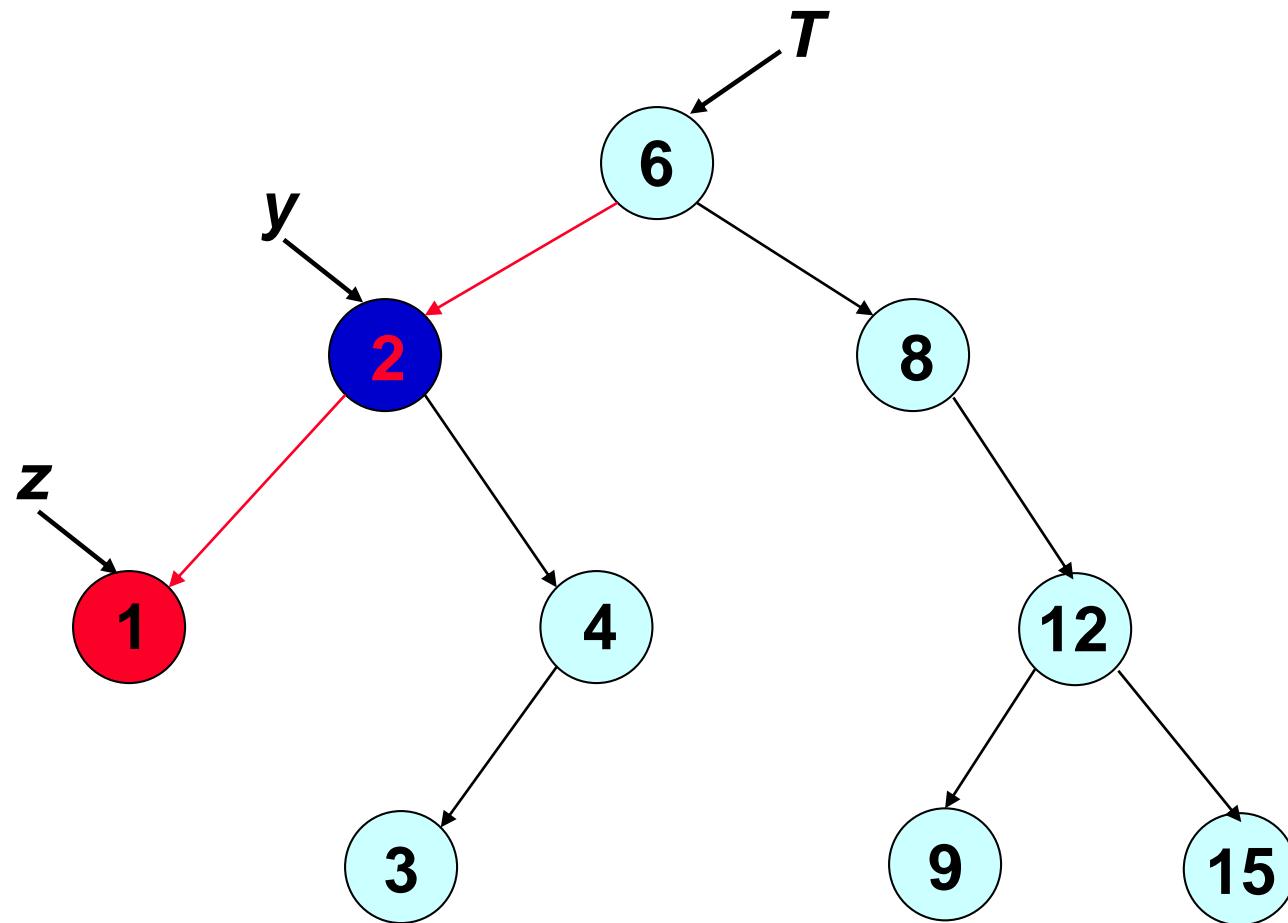
ARB: ricerca del successore II



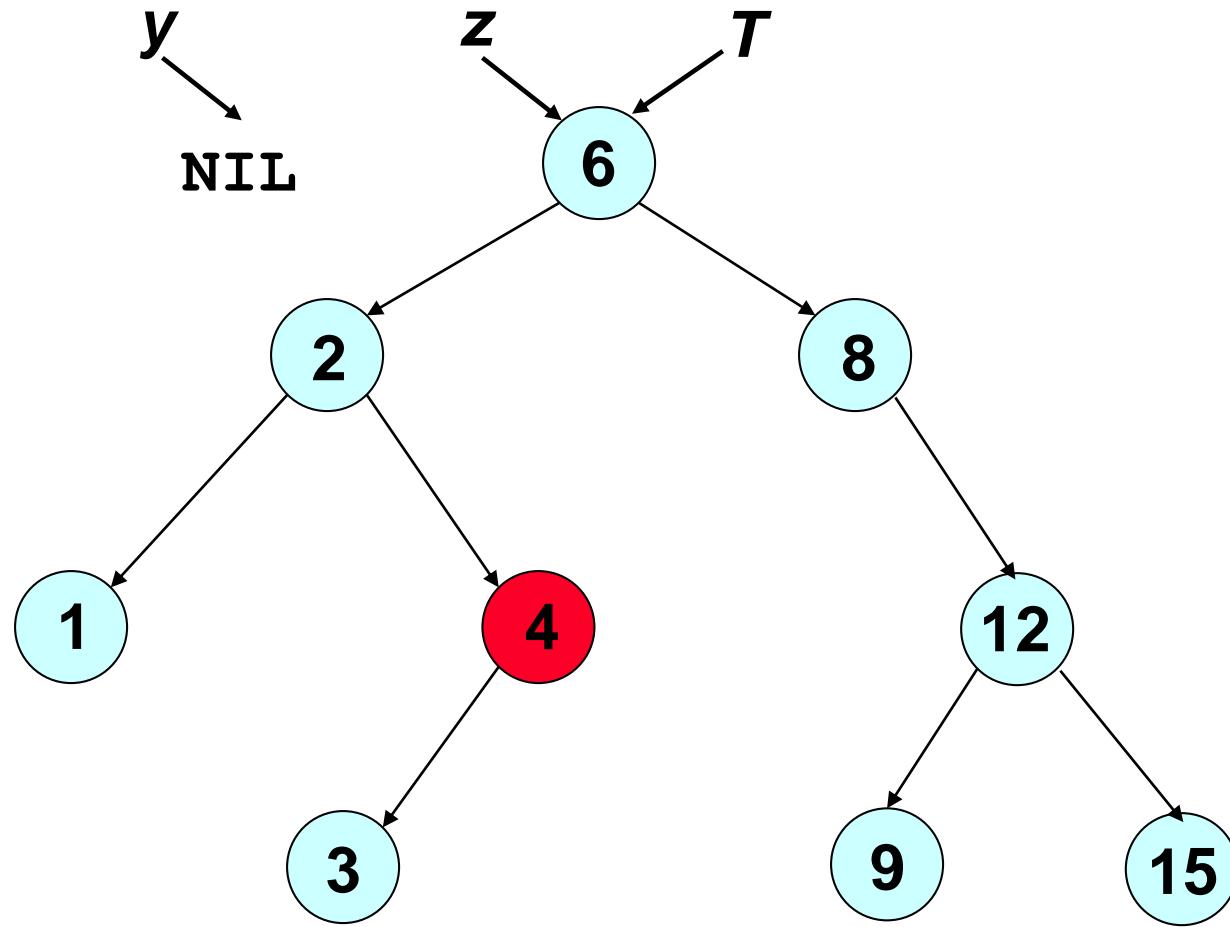
ARB: ricerca del successore II



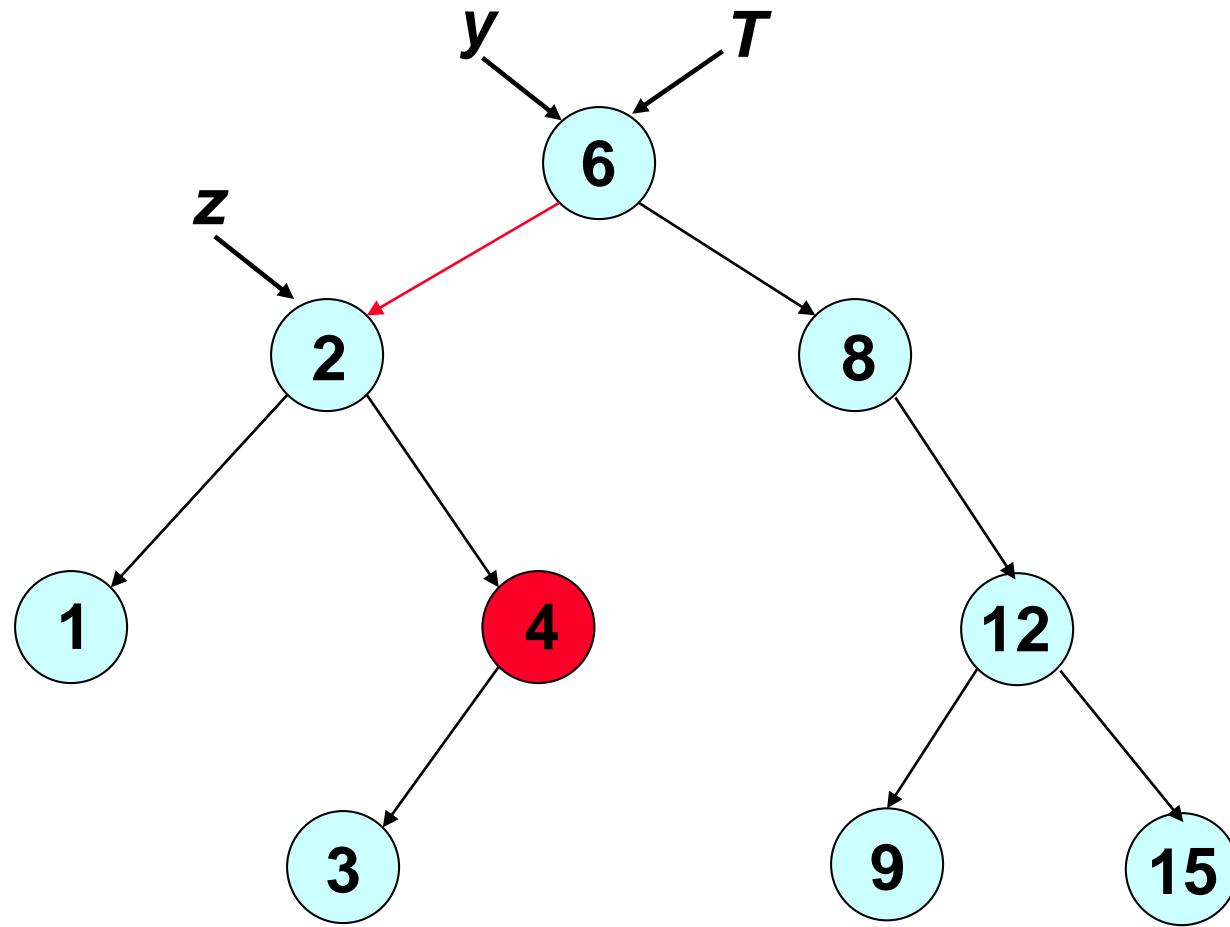
ARB: ricerca del successore II



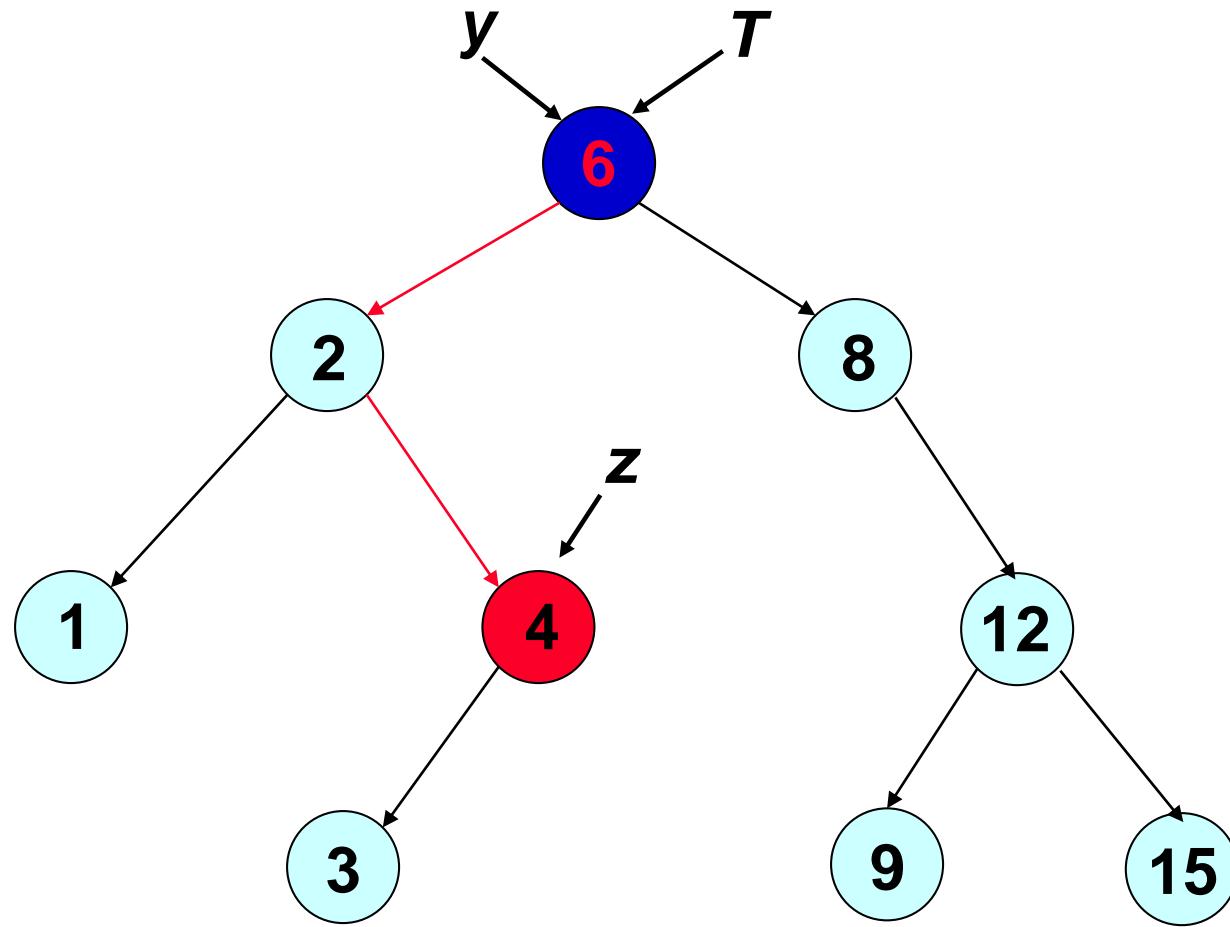
ARB: ricerca del successore II



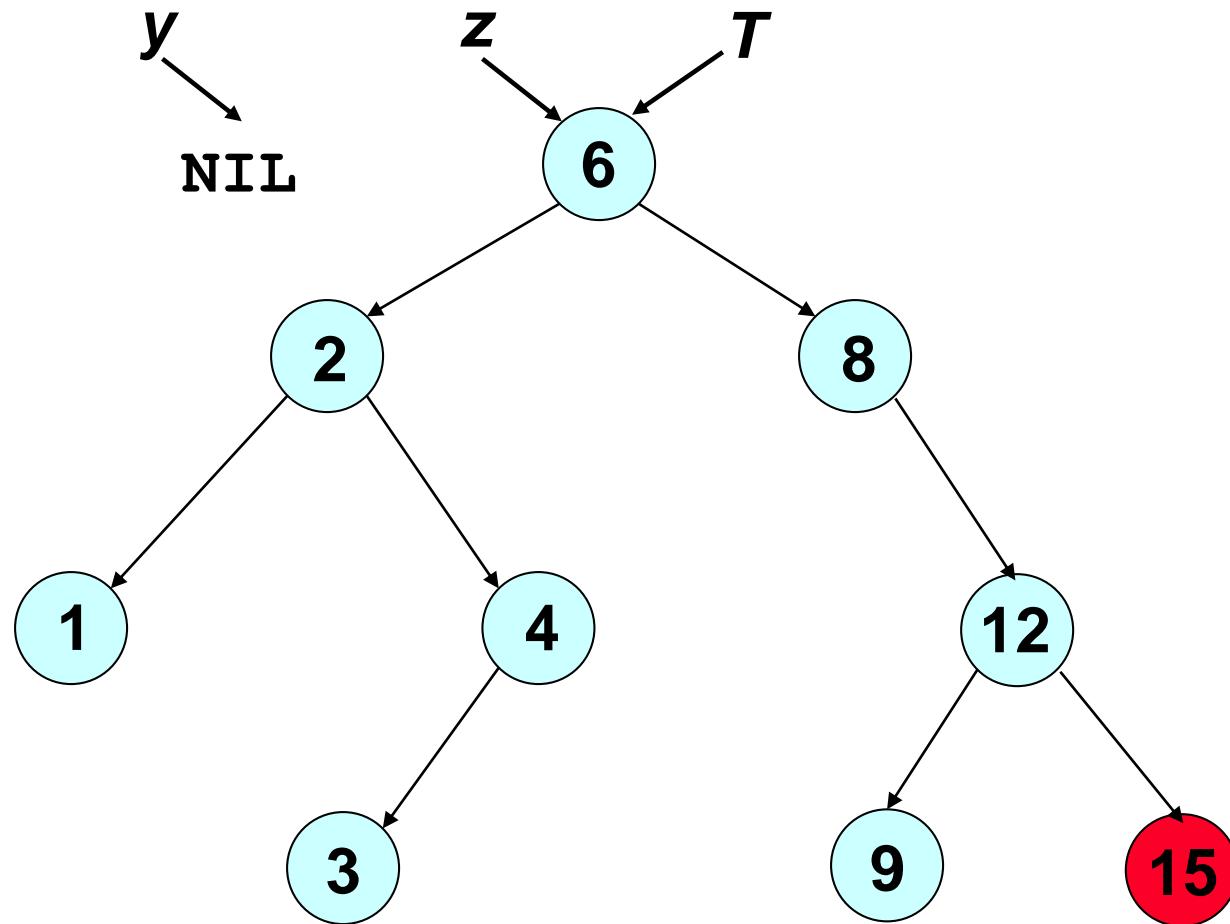
ARB: ricerca del successore II



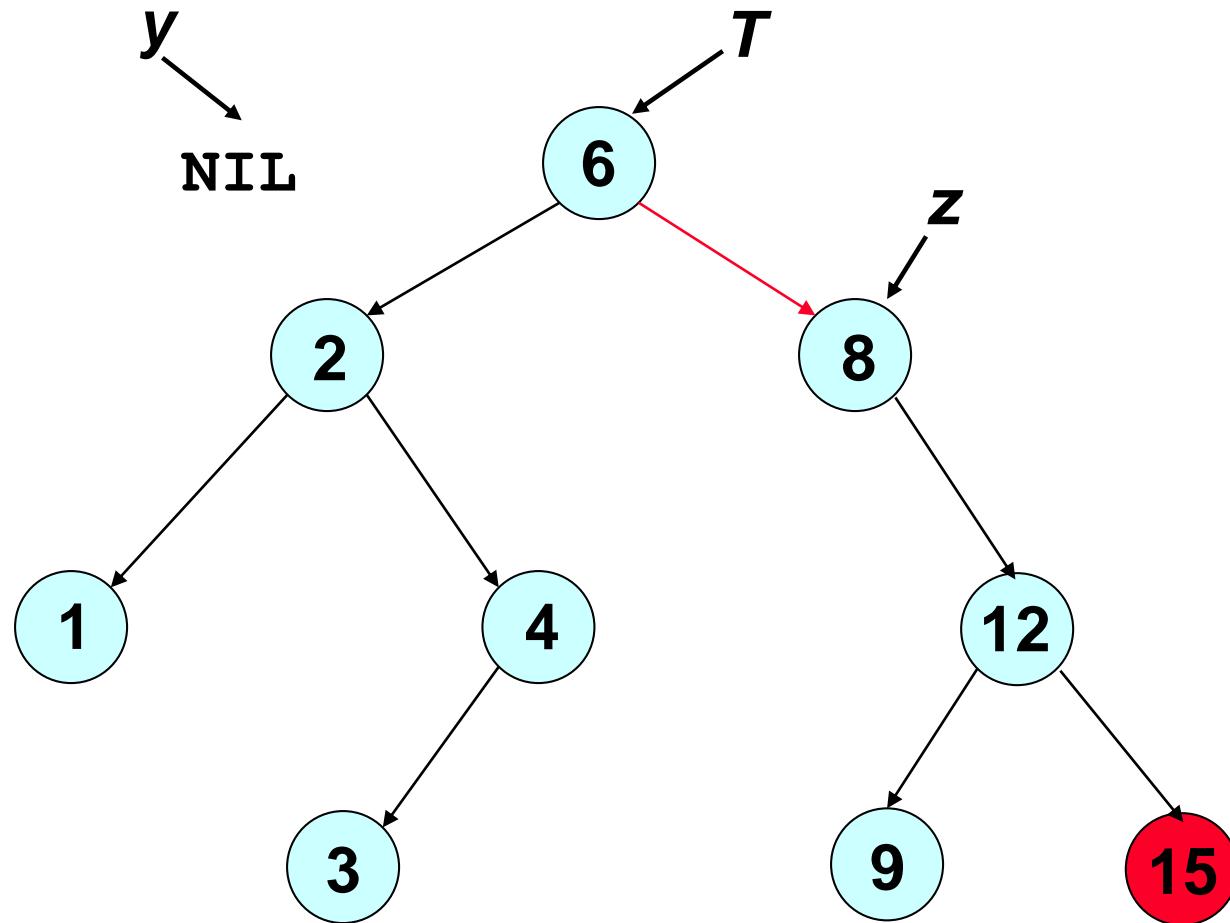
ARB: ricerca del successore II



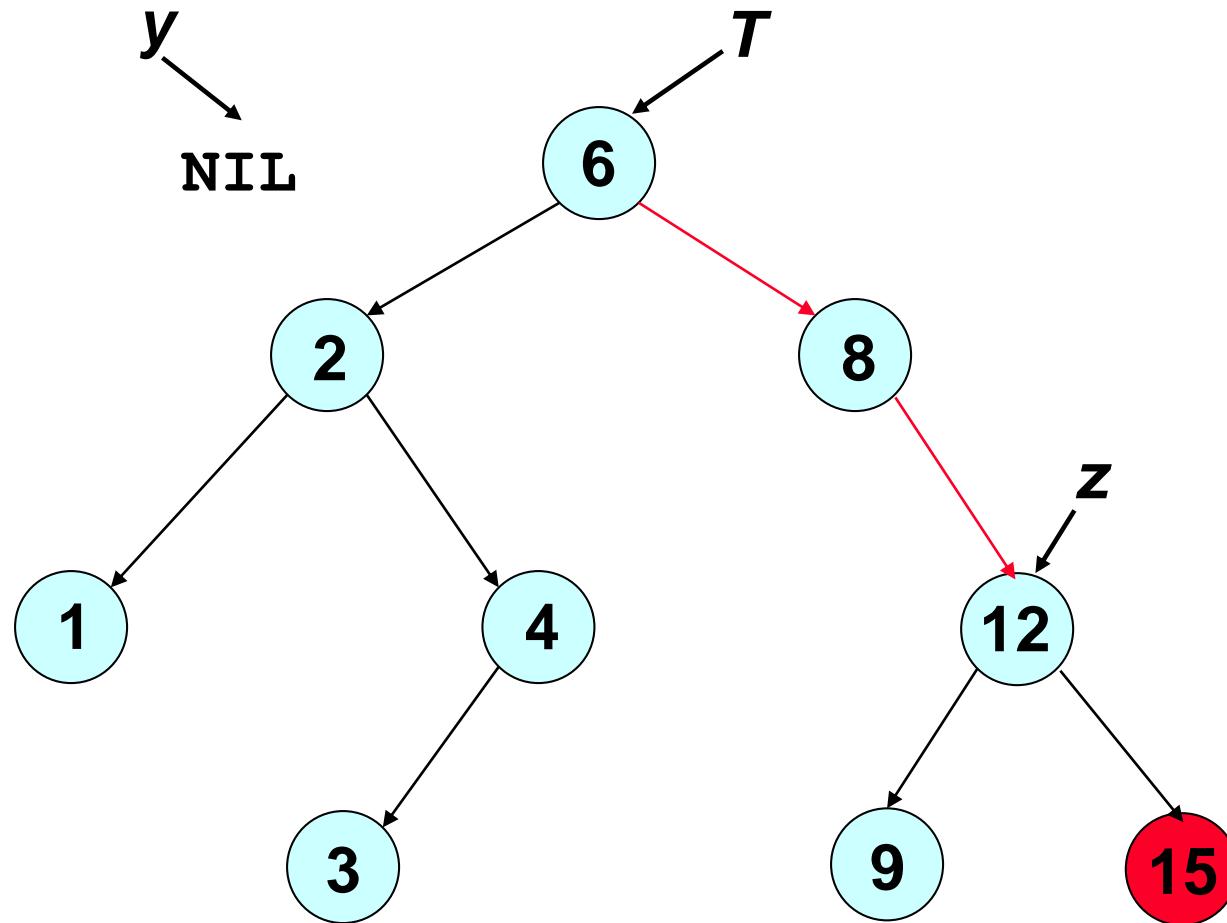
ARB: ricerca del successore II



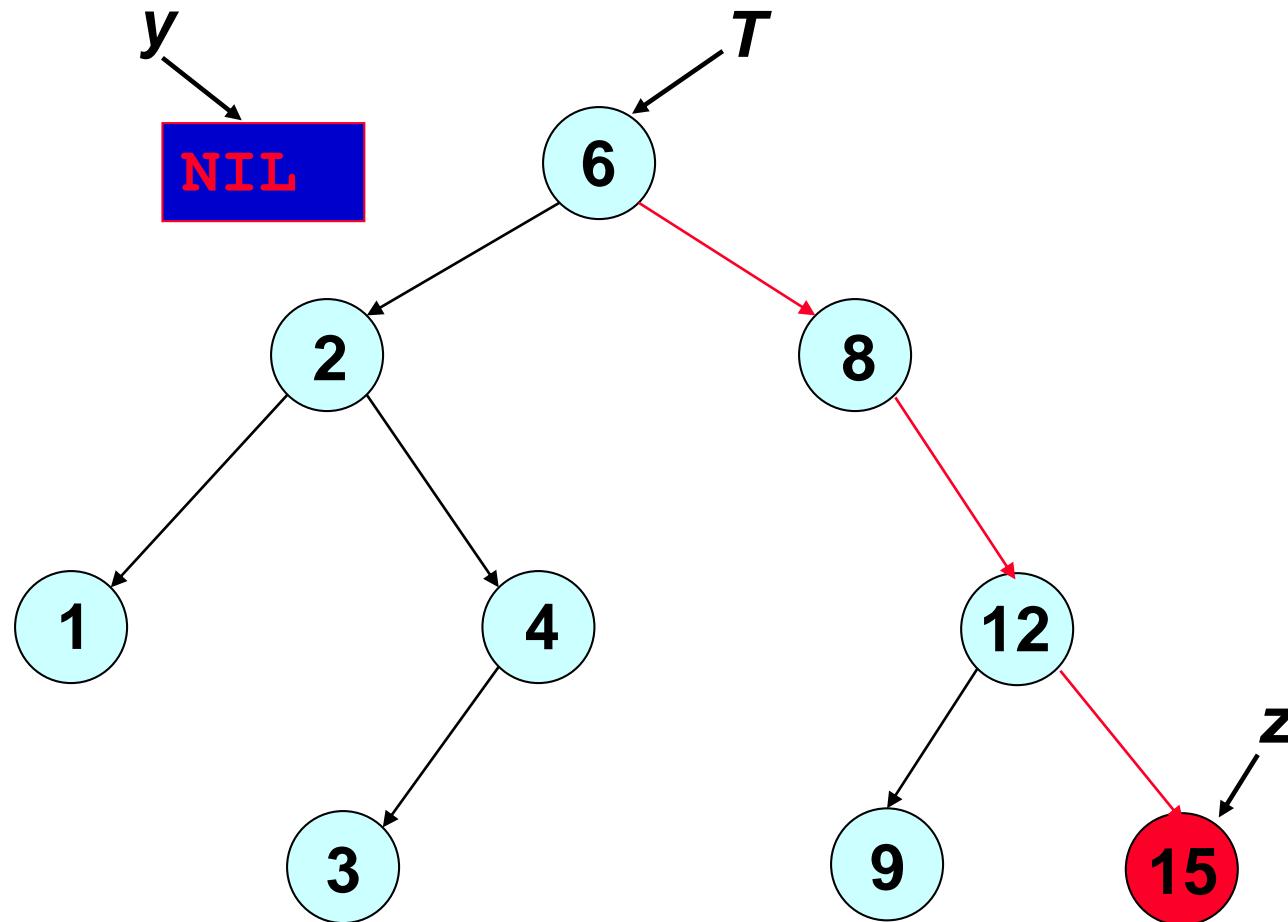
ARB: ricerca del successore II



ARB: ricerca del successore II



ARB: ricerca del successore II



ARB: ricerca del successore II

- Inizializziamo il *successore* a *NIL*
- Partendo dalla radice dell'albero:
 - ogni volta che si segue il *ramo sinistro* per raggiungere il nodo, **si aggiorna il successore al nodo padre**;
 - ogni volta che si segue un *ramo destro* per raggiungere il nodo, **NON si aggiorna il successore al nodo padre**;

ARB: ricerca del successore ricorsiva

ABR-Successore_ric(T, k)

```
IF (T ≠ NIL) THEN
    IF (T->key = k)
        return ABR-Minimo(T->dx)
    ELSE IF (T->key < k) THEN
        return ABR-Successore_ric(T->dx, k)
    ELSE /* key[T] > key */
        succ = ABR-Successore_ric(T->sx, k)
        IF (succ ≠ NIL) THEN
            return succ
return T
```

ARB: ricerca del successore

ARB ABR-Successore' (T, k)

$z = T$

$y = NIL$

WHILE ($z \neq NIL \ \&\& z->key \neq k$)

 IF ($z->key < k$)

$z = z->dx$

 ELSE IF ($z->key > k$)

$y = z$

$z = z->sx$

 IF ($z \neq NIL \ \&\& z->dx \neq NIL$) THEN

$y = ABR\text{-Minimo}(z->dx)$

return y

y punta sempre al miglior candidato a successore

ARB: ricerca del successore ricorsiva

ABR-Successore_ric' (T, k, y)

```
IF (T ≠ NIL) THEN
    IF (k > T->key) THEN
        return ABR-Successore_ric' (T->dx, k, y)
    ELSE IF (k < T->key) THEN
        return ABR-Successore_ric' (T->sx, k, T)
    ELSE /* k = T->key */
        IF (T->dx ≠ NIL) THEN
            return ABR-Minimo (T->dx)
return y
```

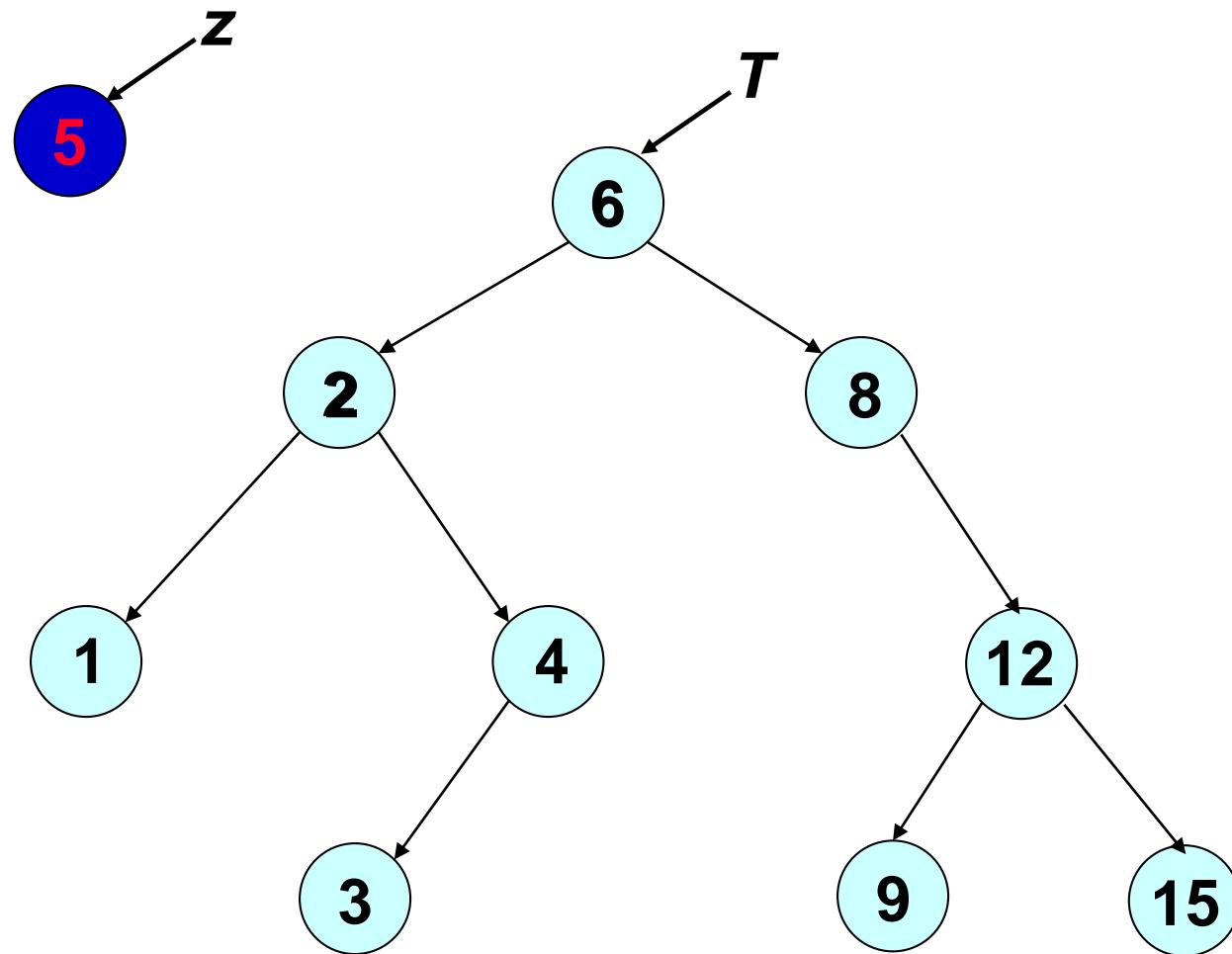
ABR-Successore' (T, k)

```
return ABR-Successore_ric' (T, k, NIL)
```

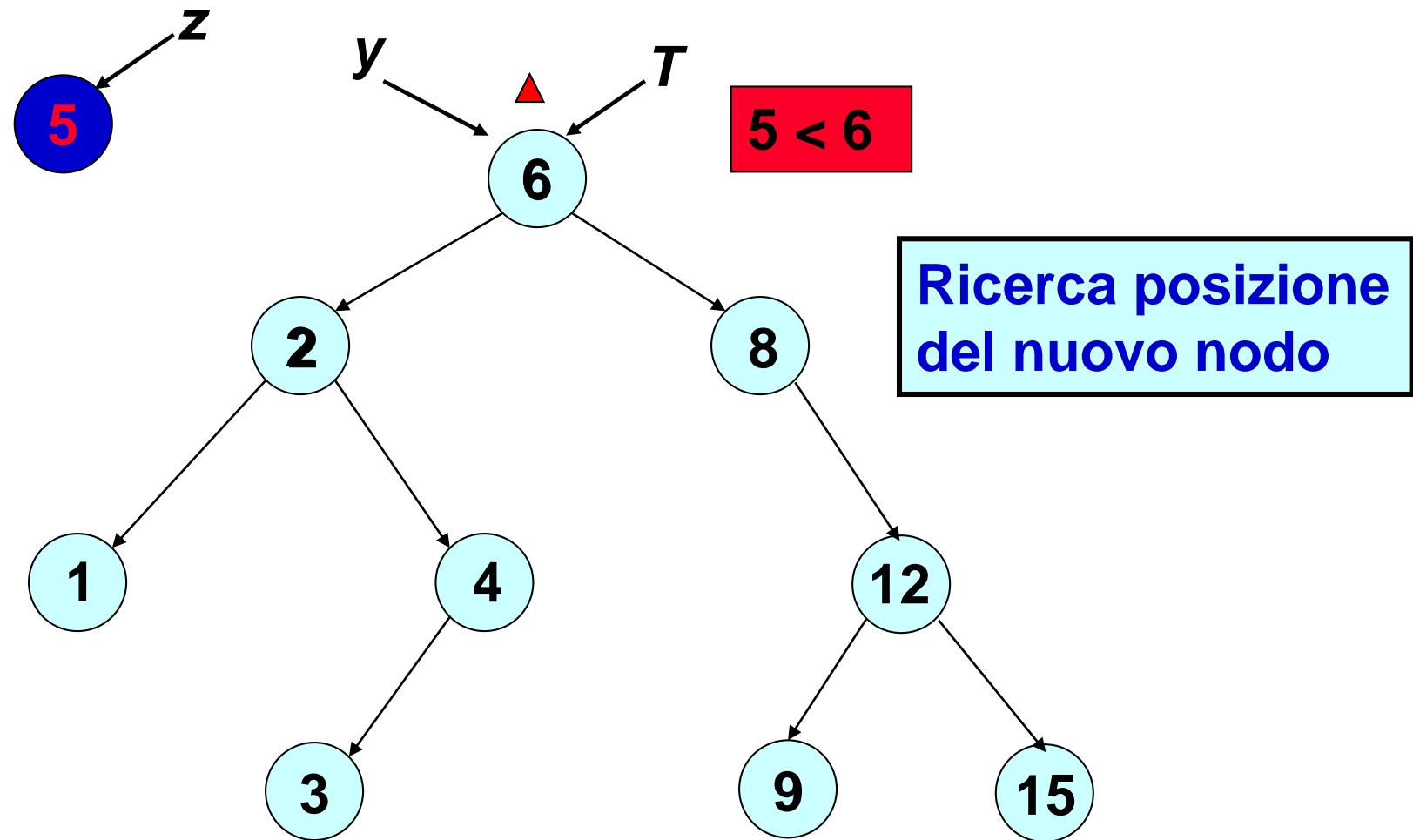
ARB: costo delle operazioni

Teorema. Le operazioni di **Ricerca**, **Minimo**, **Massimo**, **Successore** e **Predecessore** su di un **Albero Binario di Ricerca** possono essere eseguite in tempo $O(h)$, dove h è l'altezza dell'albero.

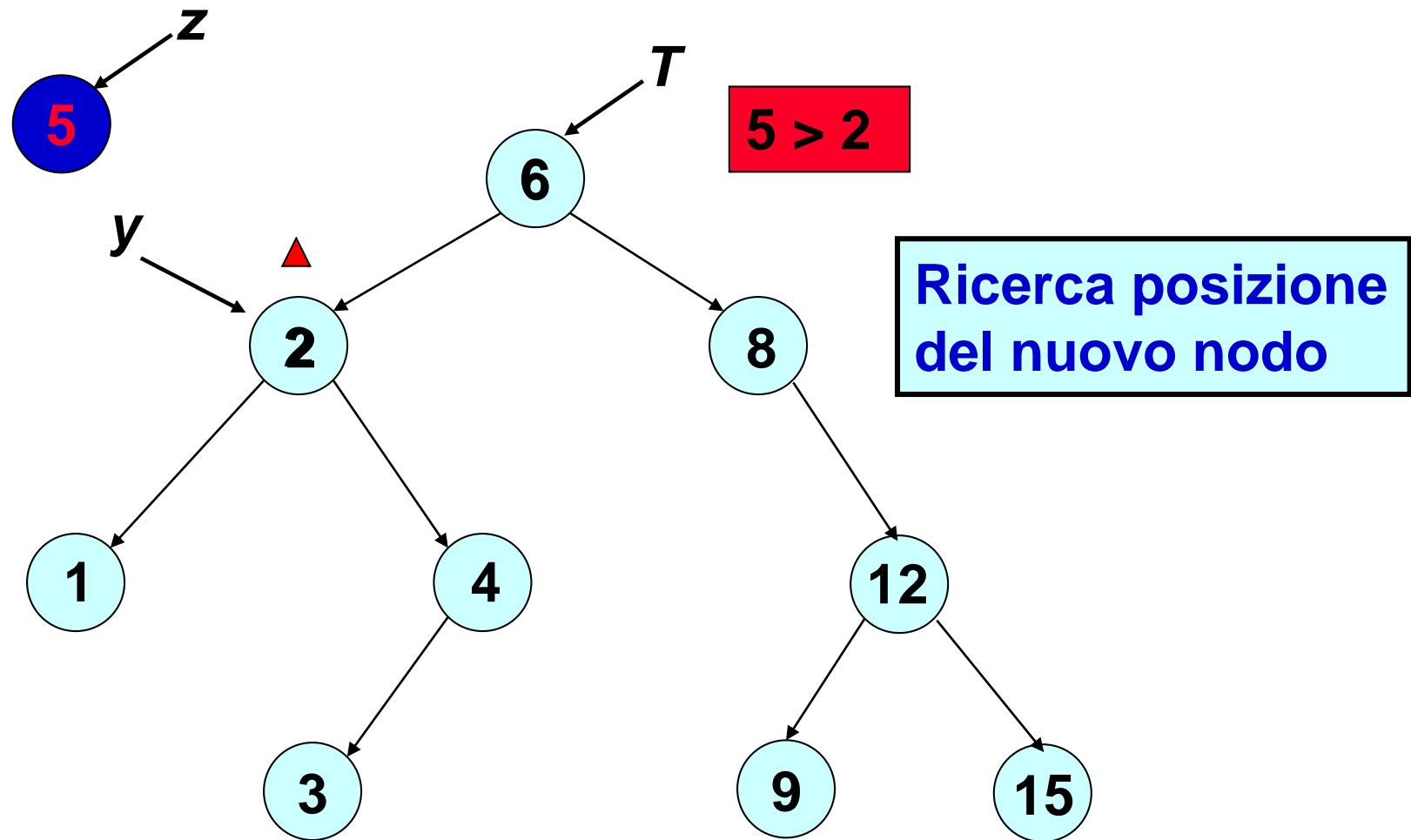
ARB: Inserimento di un nodo (caso I)



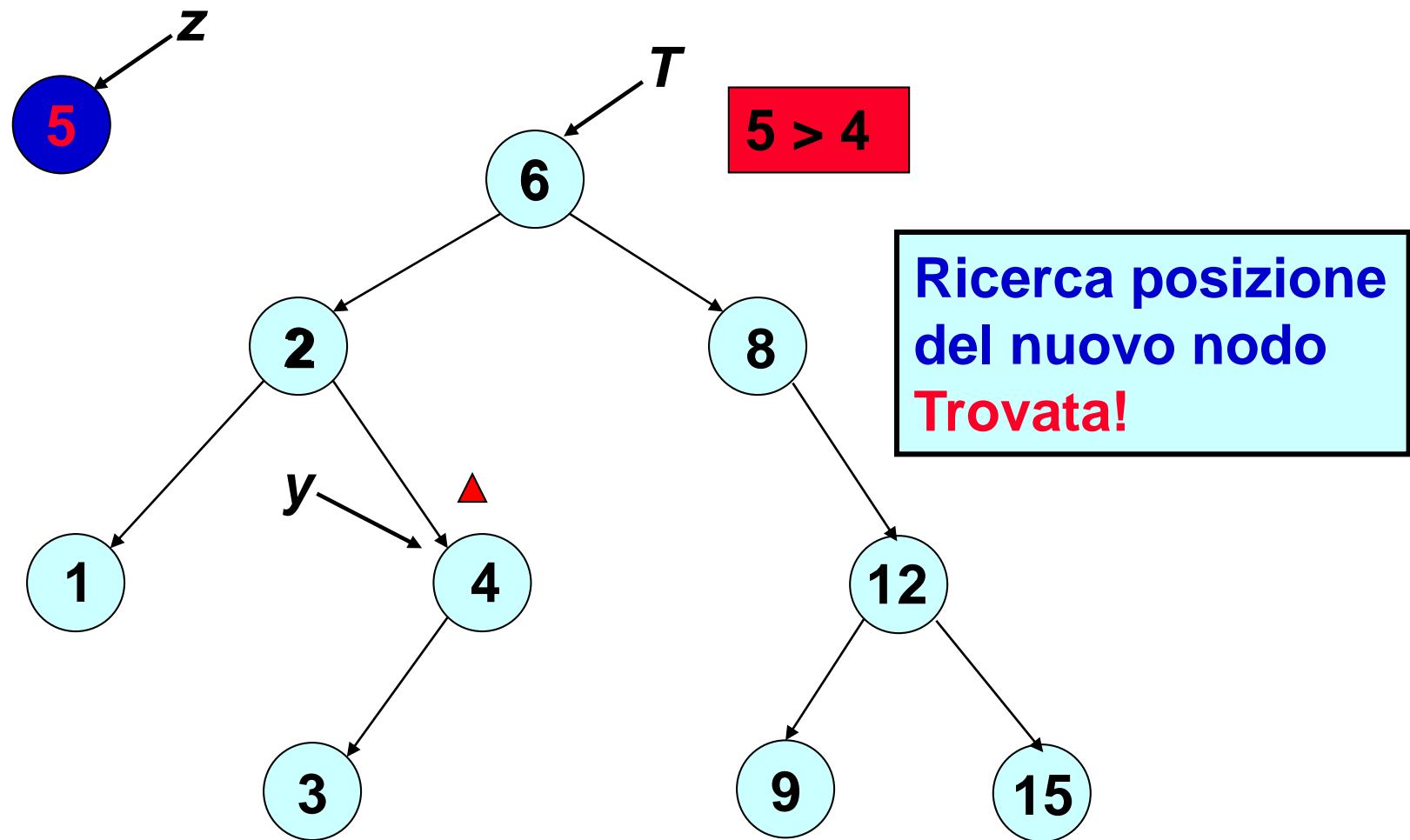
ARB: Inserimento di un nodo (caso I)



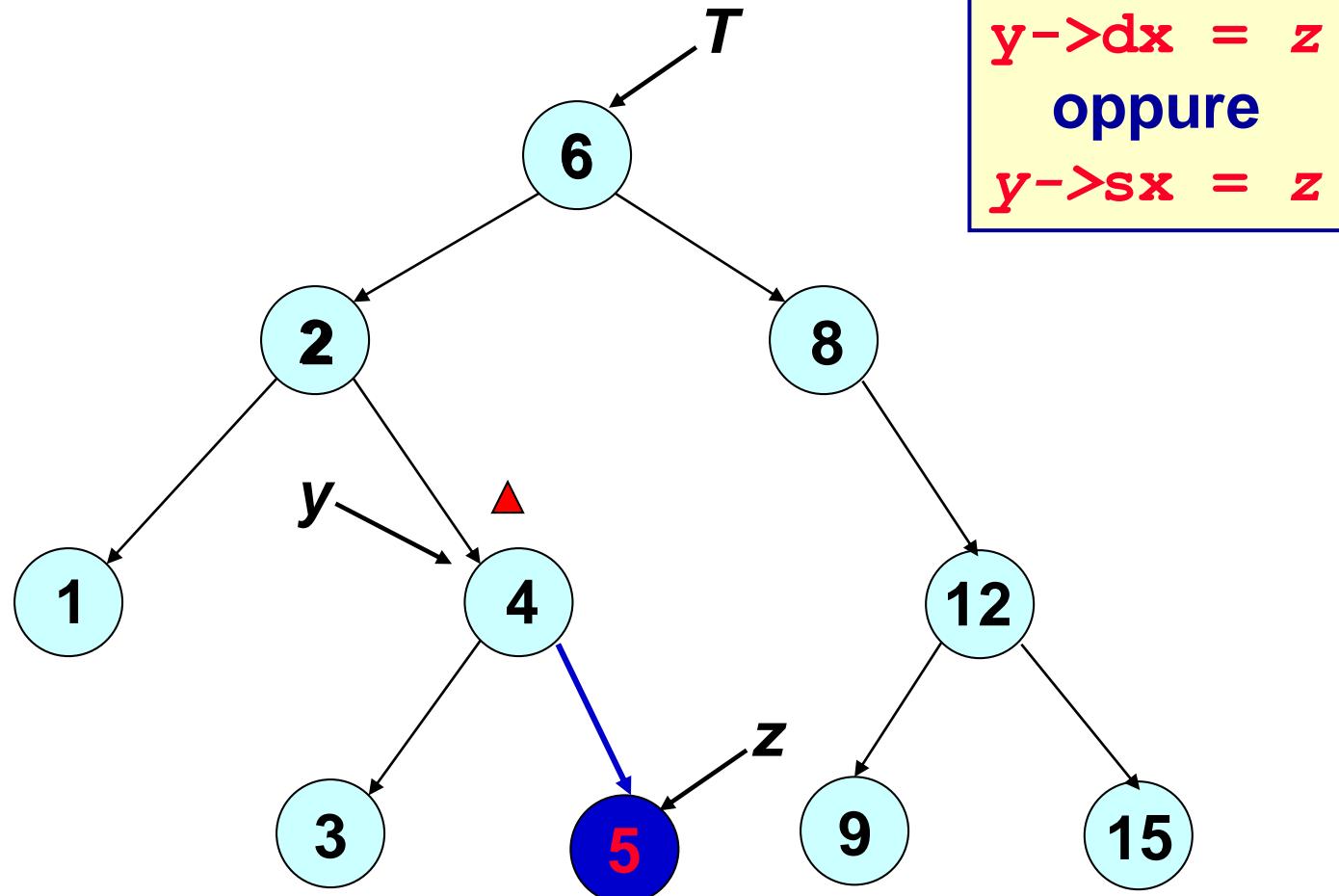
ARB: Inserimento di un nodo (caso I)



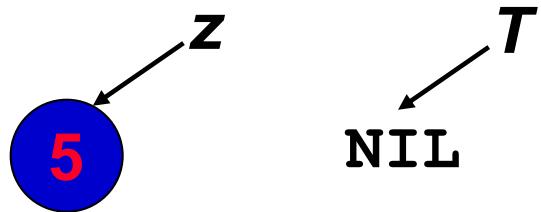
ARB: Inserimento di un nodo (caso I)



ARB: Inserimento di un nodo (caso I)

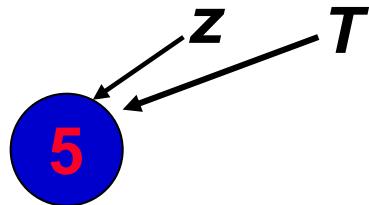


ARB: Inserimento di un nodo (caso II)



Albero è vuoto

ARB: Inserimento di un nodo (caso II)



Root [T] = z

Albero è vuoto
Il nuovo nodo da inserire diviene la radice

ARB: Inserimento di un nodo

```
ABR-inserisci(T,k)
  P = NIL
  x = T
  WHILE (x ≠ NIL) DO
    P = x
    IF (z->key < x->key) THEN
      x = x->sx
    ELSE
      x = x->dx
  z = alloca nodo ARB
  z->key = k
  IF P = NIL THEN
    T = z
  ELSE IF z->key < P->key THEN
    P->sx = z
  ELSE
    P->dx = z
```

ARB: Inserimento di un nodo

```
ABR-inserisci( $T, k$ )
```

```
     $P = \text{NIL}$ 
```

```
     $x = T$ 
```

```
    WHILE ( $x \neq \text{NIL}$ ) DO
         $P = x$ 
        IF ( $z \rightarrow \text{key} < x \rightarrow \text{key}$ ) THEN
             $x = x \rightarrow \text{sx}$ 
        ELSE
             $x = x \rightarrow \text{dx}$ 
```

```
     $z = \text{alloca nodo ARB}$ 
```

```
     $z \rightarrow \text{key} = k$ 
```

```
    IF  $P = \text{NIL}$  THEN
```

```
         $T = z$ 
```

```
    ELSE IF  $z \rightarrow \text{key} < P \rightarrow \text{key}$  THEN
```

```
         $P \rightarrow \text{sx} = z$ 
```

```
    ELSE
```

```
         $P \rightarrow \text{dx} = z$ 
```

Ricerca posizione
del nuovo nodo

(caso II)

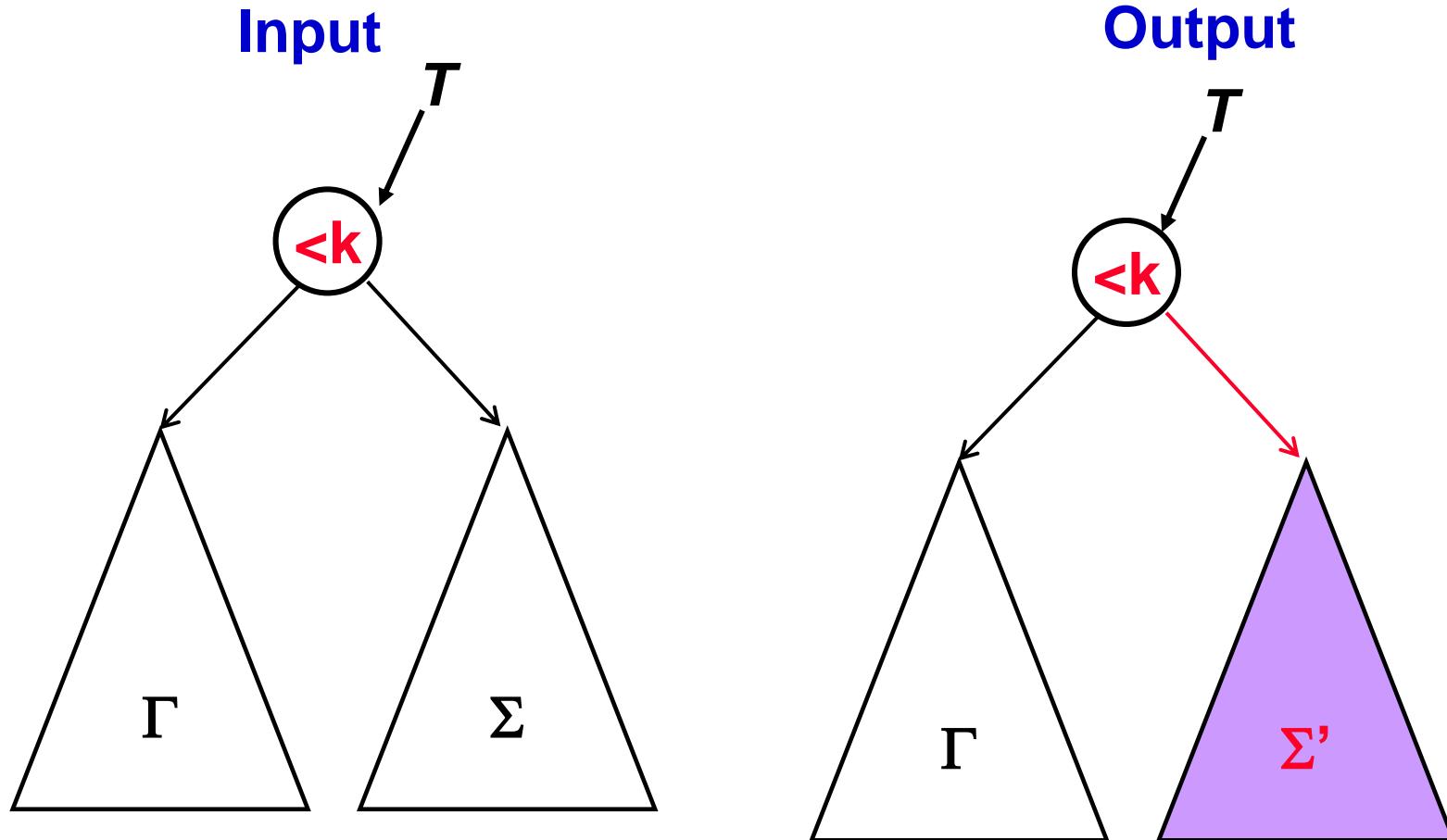
(caso I)

ARB: Inserimento di un nodo

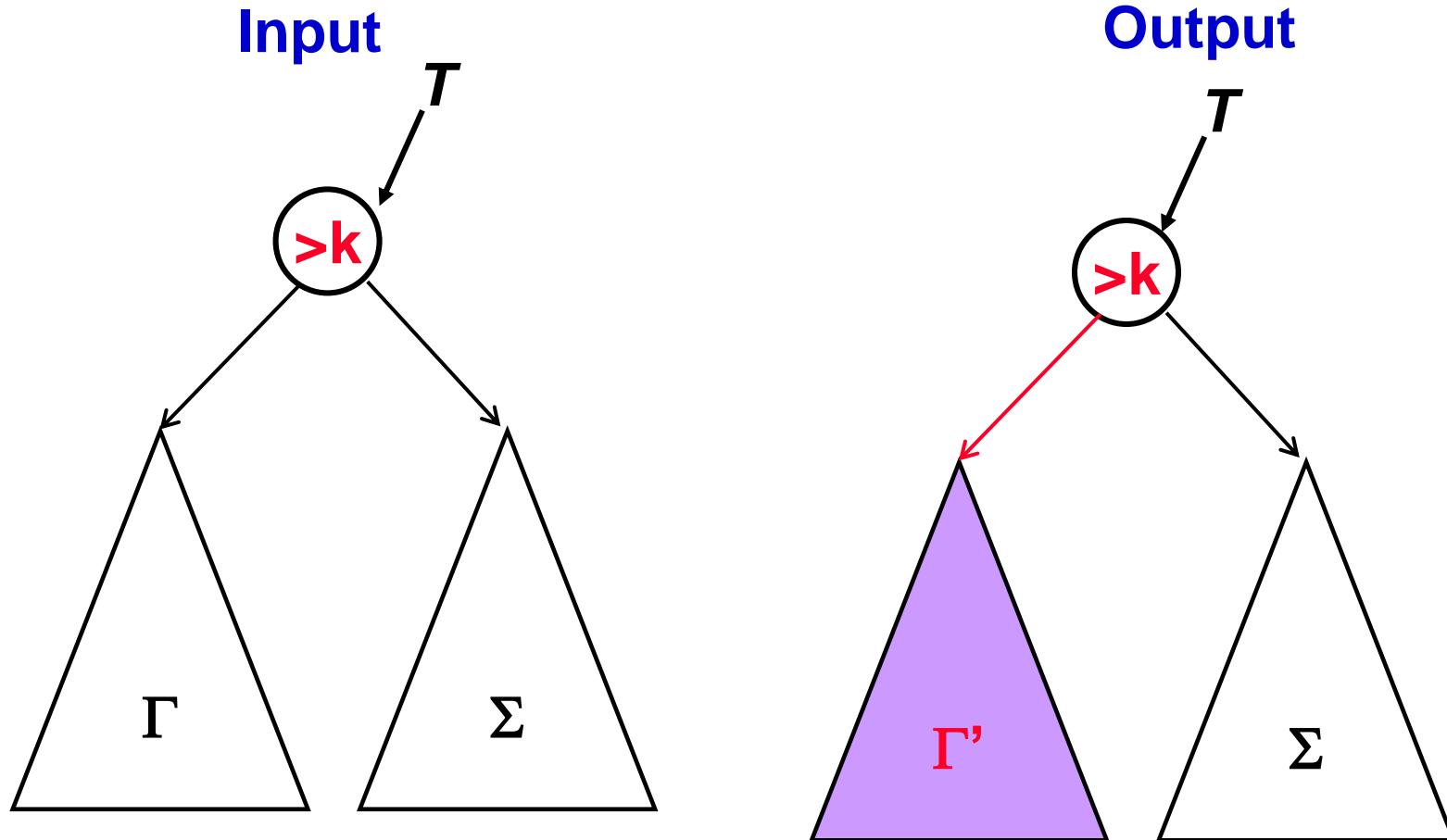
```
ABR-insert_ric(T,k)
IF T ≠ NIL THEN
    IF k < T->key THEN
        T->sx = ABR-insert_ric(T->sx,k)
    ELSE
        T->dx = ABR-insert_ric(T->dx,k)
    ELSE
        T = alloca nodo ARB
        T->key = k
return z
```

k è la chiave da inserire.

Cancellazione ricorsiva



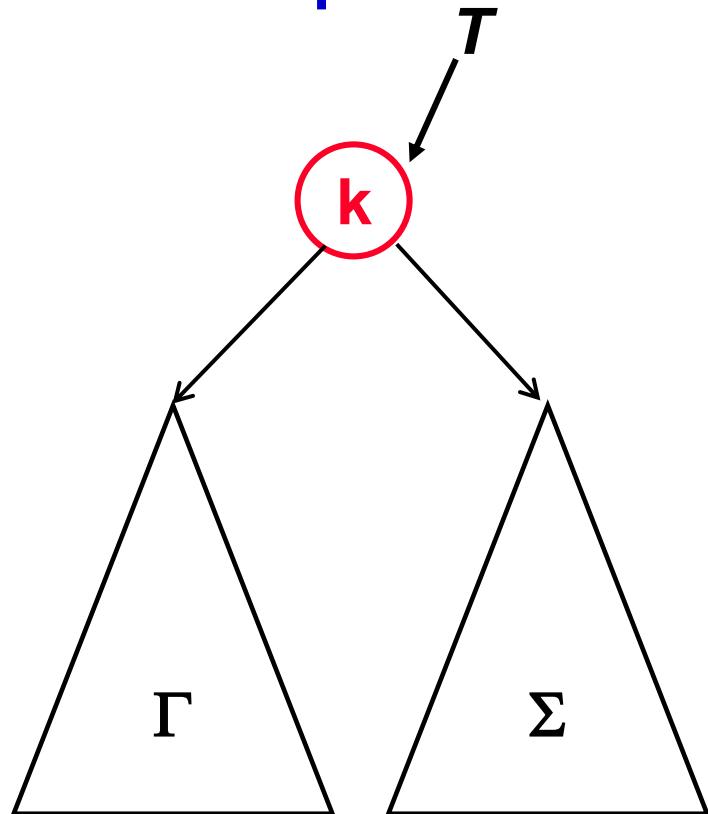
Cancellazione ricorsiva



$$\Gamma' = \text{cancella}(\Gamma, k)$$

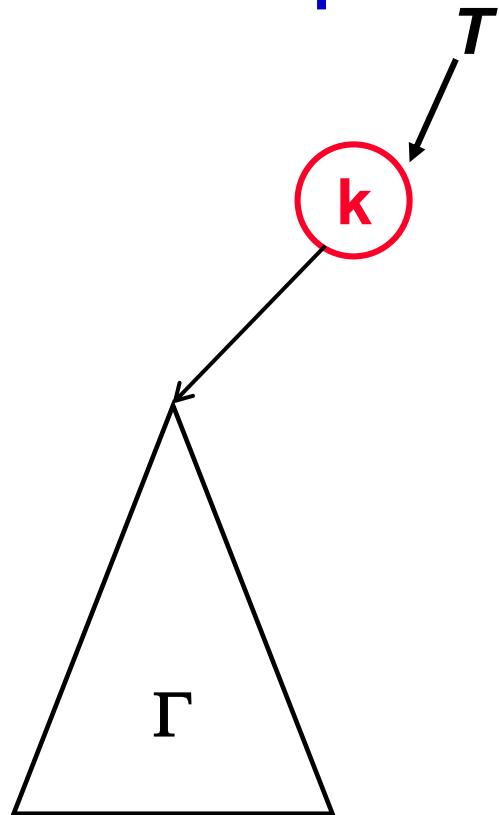
Cancellazione ricorsiva

Input

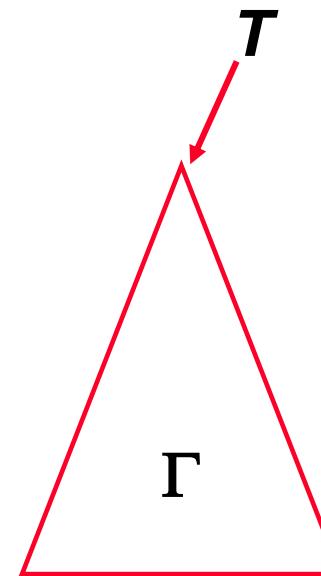


Cancellazione ricorsiva (caso I)

Input

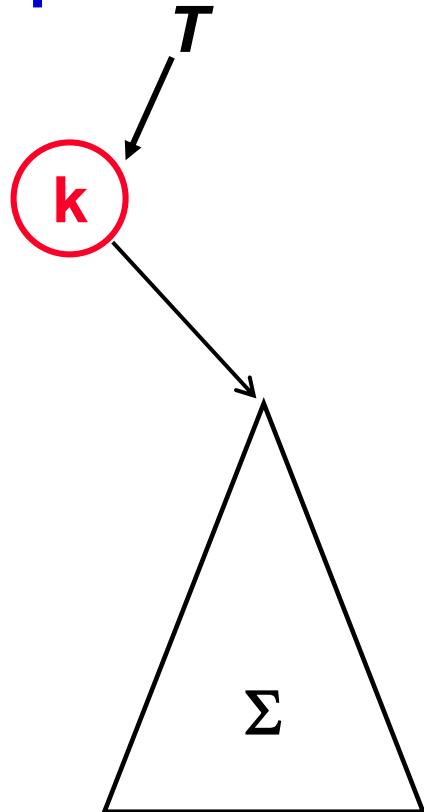


Output



Cancellazione ricorsiva (caso II)

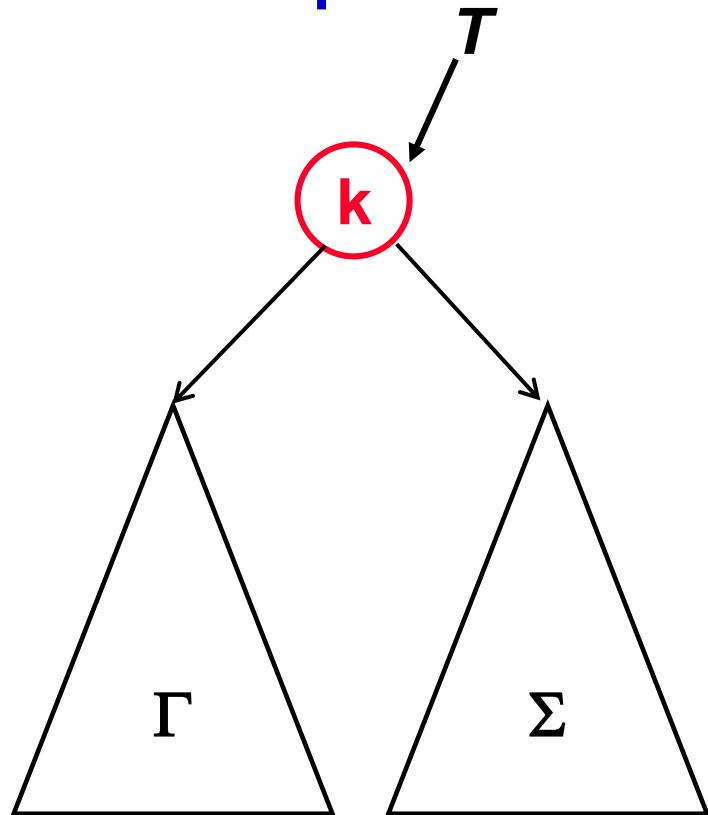
Input



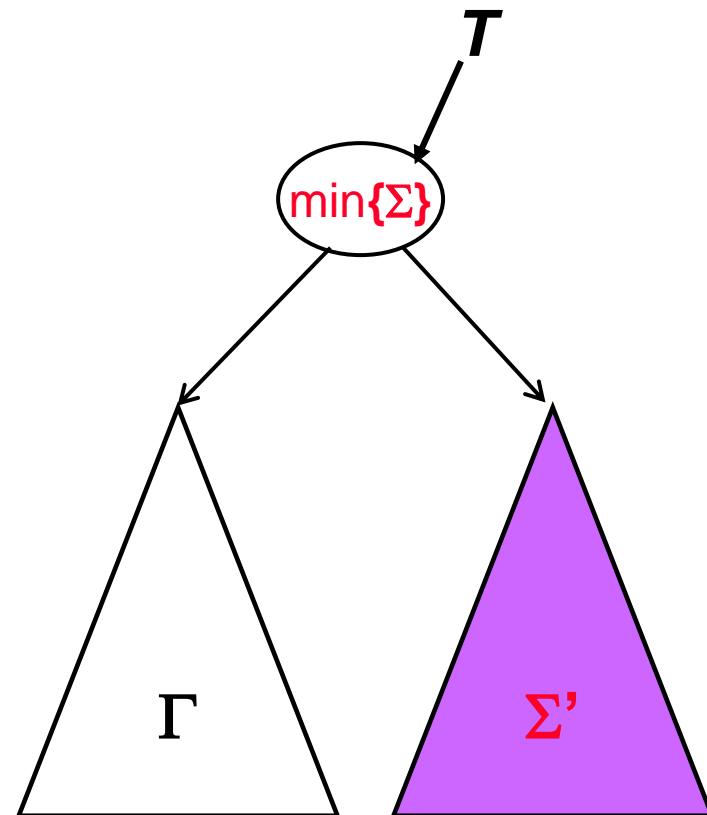
Output

Cancellazione ricorsiva (caso III)

Input



Output



$$\Sigma' = \text{stacca-minimo}(\Sigma)$$

ARB: Cancellazione ricorsiva

```
ABR-Cancella-ric(k, T)
```

```
  IF T ≠ NIL THEN
    IF k < T->key THEN
      T->sx = ABR-Cancella-ric(k, T->sx)
    ELSE IF k > T->key THEN
      T->sx = ABR-Cancella-ric(k, T->dx)
    ELSE /* k = T->key */
      T = ABR-Cancella-Root(T)
  return T
```

key

sx

dx

```
ABR-Cancella-Root(T)
```

```
  IF T ≠ NIL THEN
    IF T->sx ≠ NIL && T->dx ≠ NIL THEN
      tmp = Stacca-Min(T, T->dx)
      "Copia tmp->key in T->key"
  ELSE
```

```
    tmp = T
    IF T->dx ≠ NIL THEN T = T->dx
                           ELSE T = T->sx
```

dealloca *tmp*

```
return T
```

Ricerca successore
Caso III

Casi I e II

ARB: Cancellazione ricorsiva

```
Stacca-min(T, P)
```

```
  IF T ≠ NIL THEN
```

```
    IF T->sx ≠ NIL THEN
```

```
      return Stacca-min(T->sx, T)
```

```
    ELSE /* successore trovato */
```

```
      IF T = P->sx THEN
```

```
        P->sx = T->dx
```

```
      ELSE /* min è il primo nodo passato */
```

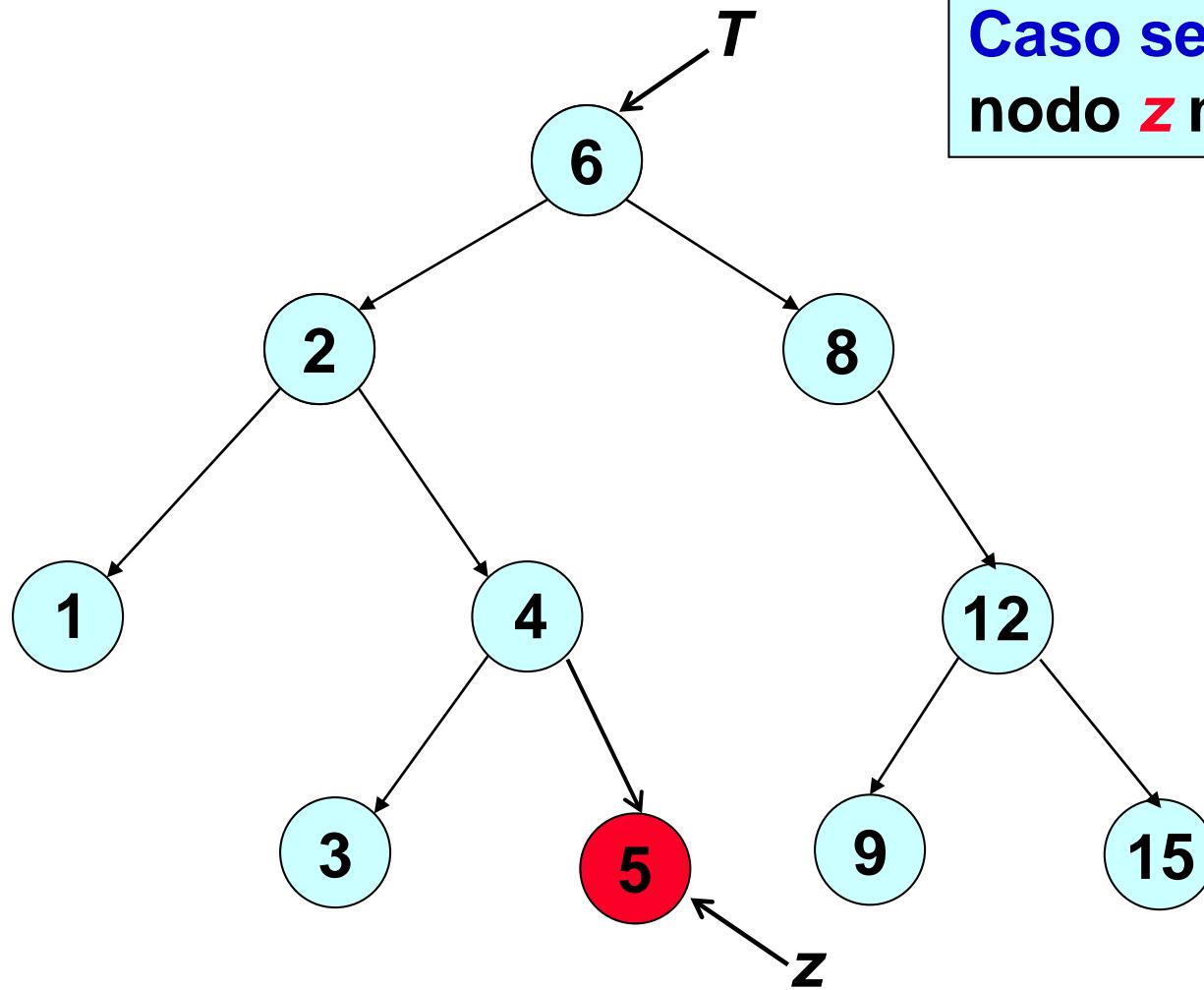
```
        P->dx = T->dx
```

```
  return T
```

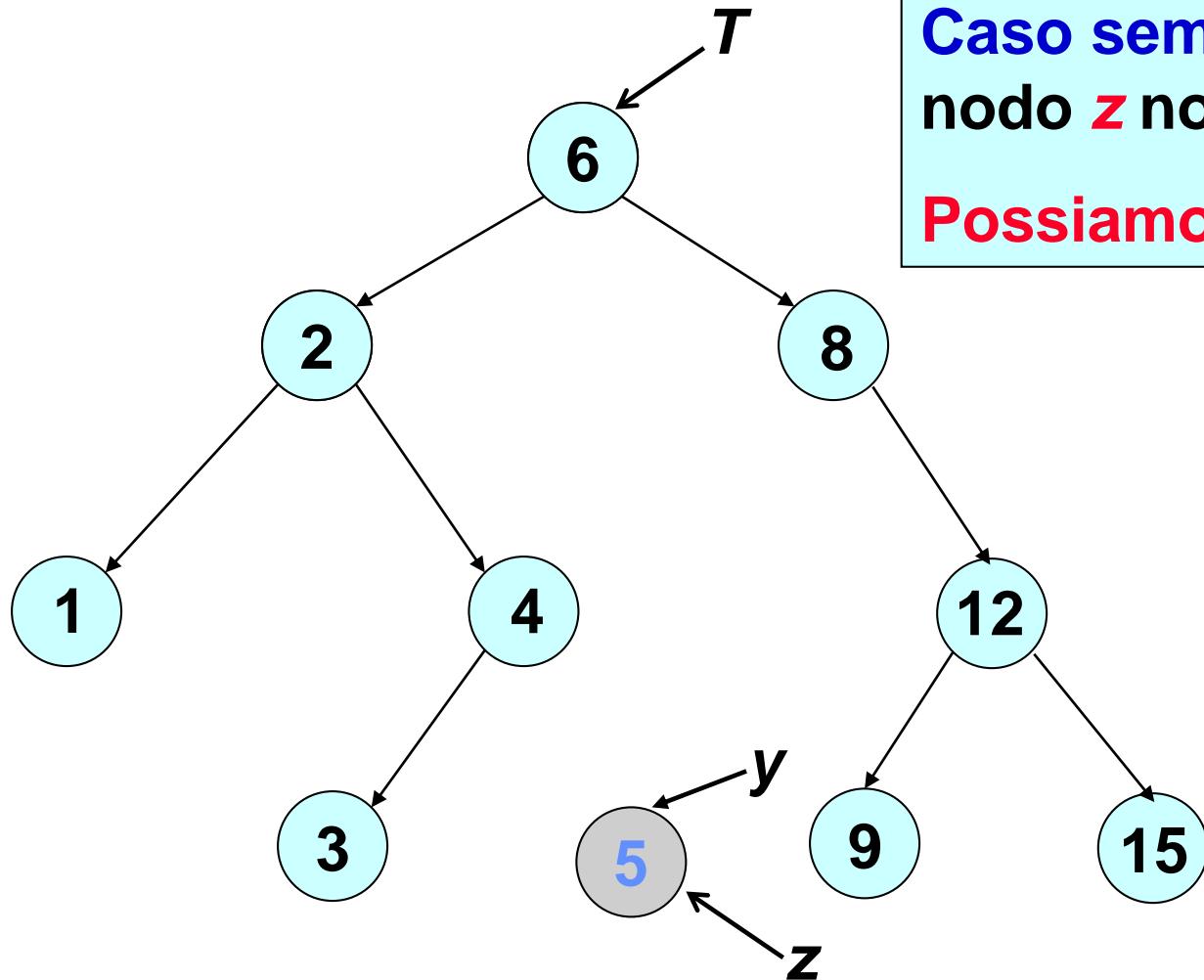
Il parametro **P** serve per ricordarsi il **padre** di **T** durante la discesa

NOTA. L'algoritmo **stacca** il **nodo minimo** dell'albero **T** e ne **ritorna il puntatore**. Può anche ritornare **NIL** in caso non esista un **minimo** (**T** è vuoto). Il valore di ritorno dovrebbe essere quindi verificato dal chiamante prima dell'uso.
Nel caso della **cancellazione ricorsiva** però siamo sicuri che il minimo esiste sempre e quindi non è necessario eseguire alcun controllo!

ARB: Cancellazione di un nodo (caso I)

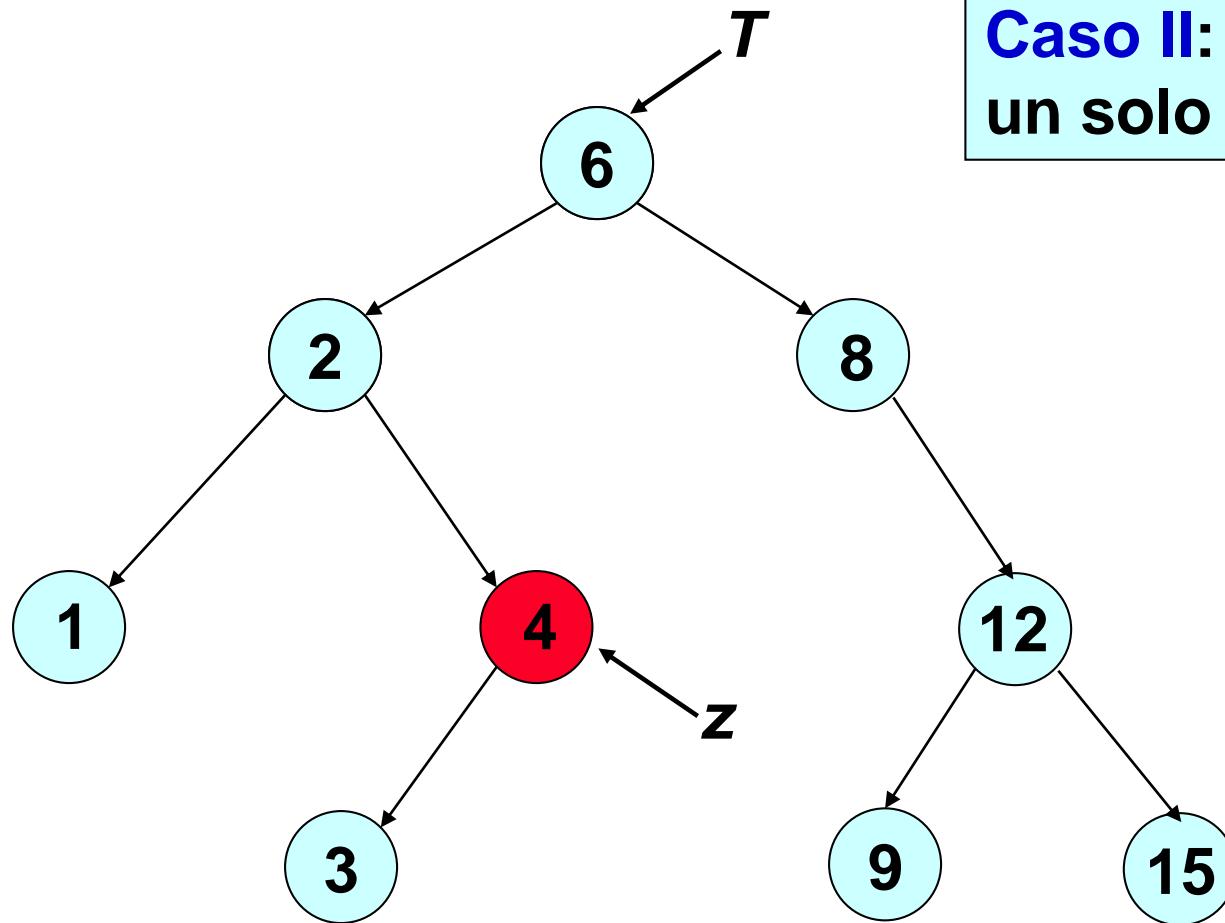


ARB: Cancellazione di un nodo (caso I)



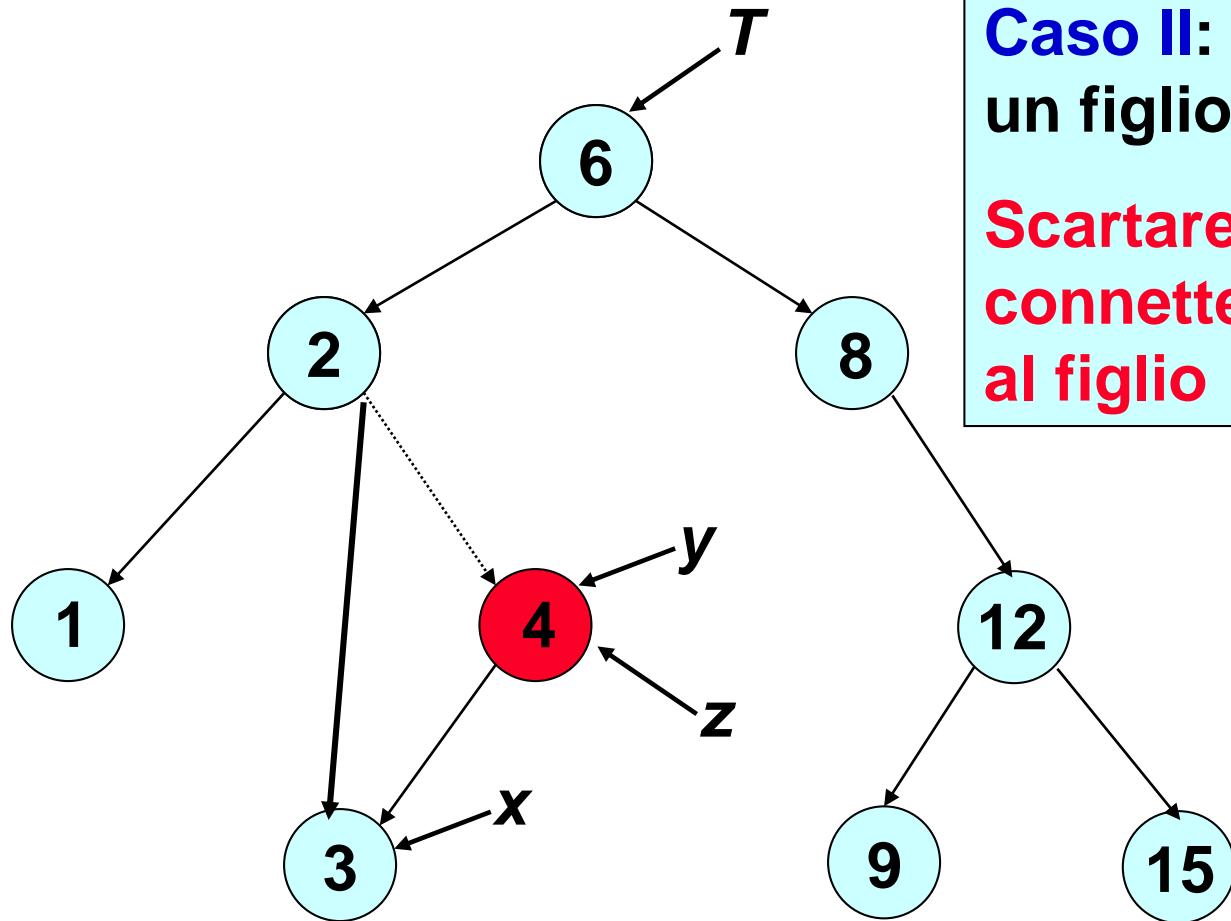
Caso semplice: il
nodo **z** non ha figli
Possiamo eliminarlo

ARB: Cancellazione di un nodo (caso II)



Caso II: il nodo ha
un solo figlio

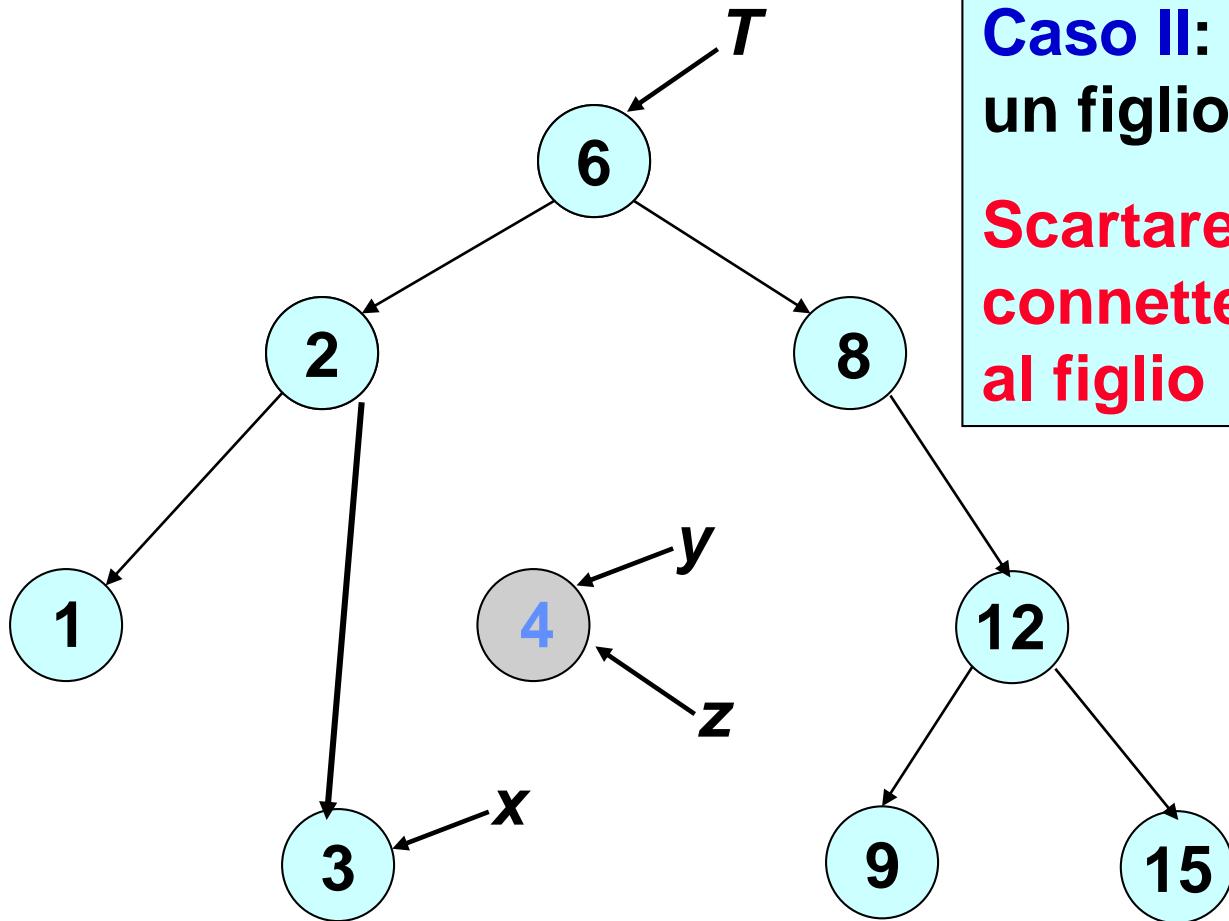
ARB: Cancellazione di un nodo (caso II)



Caso II: il nodo ha
un figlio

**Scartare il nodo e
connettere il padre
al figlio**

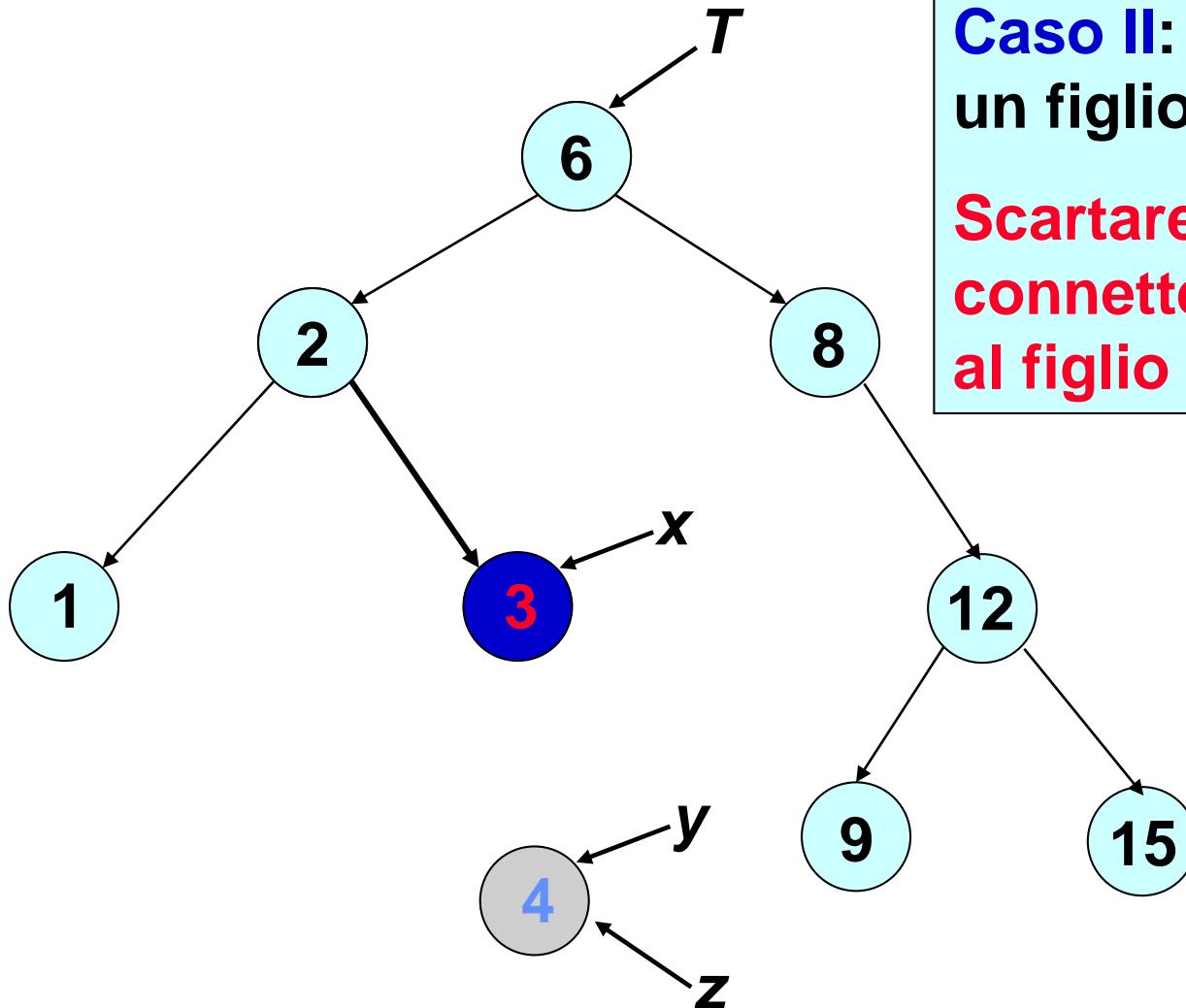
ARB: Cancellazione di un nodo (caso II)



Caso II: il nodo ha
un figlio

Scartare il nodo e
connettere il padre
al figlio

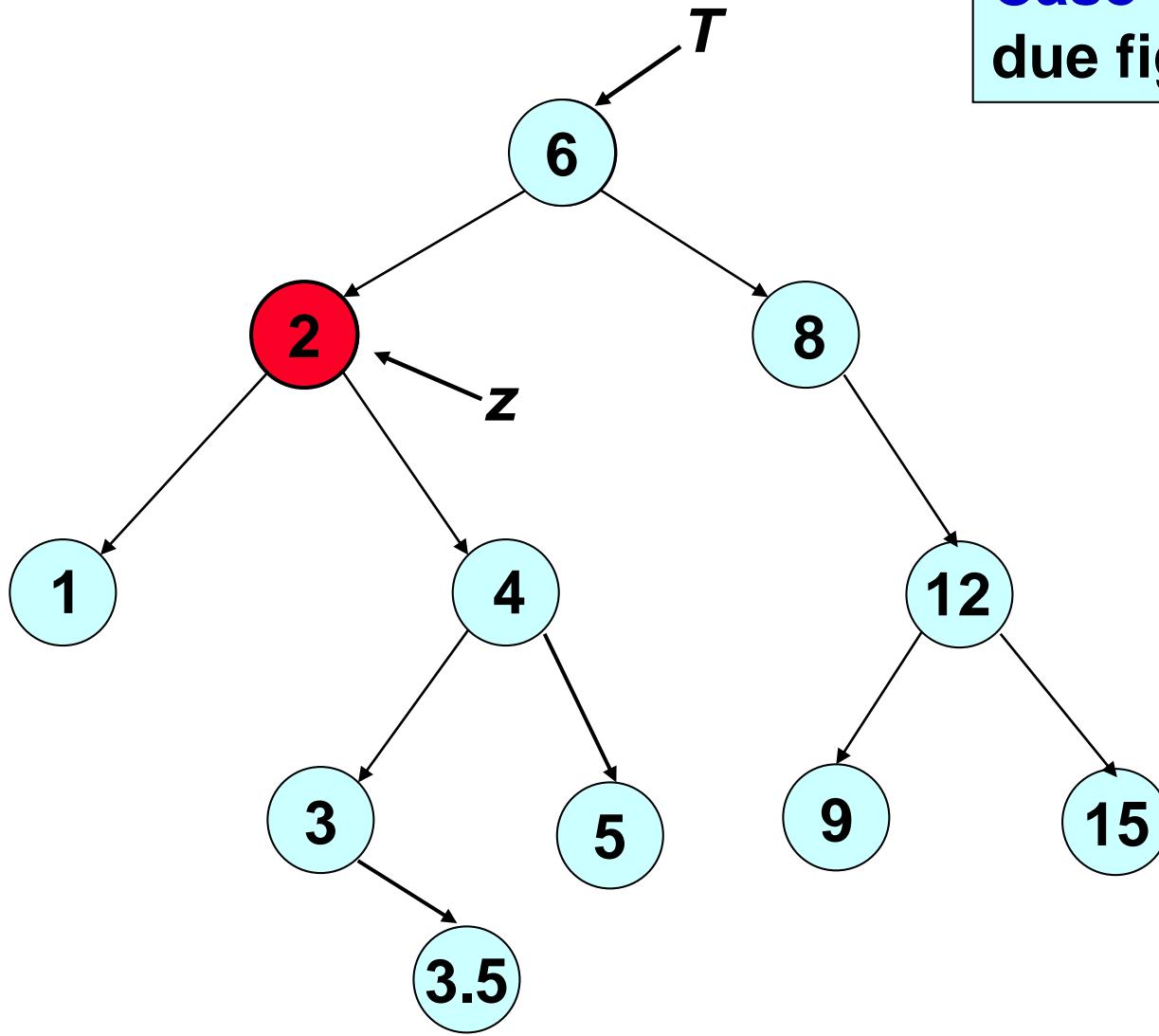
ARB: Cancellazione di un nodo (caso II)



Caso II: il nodo ha
un figlio

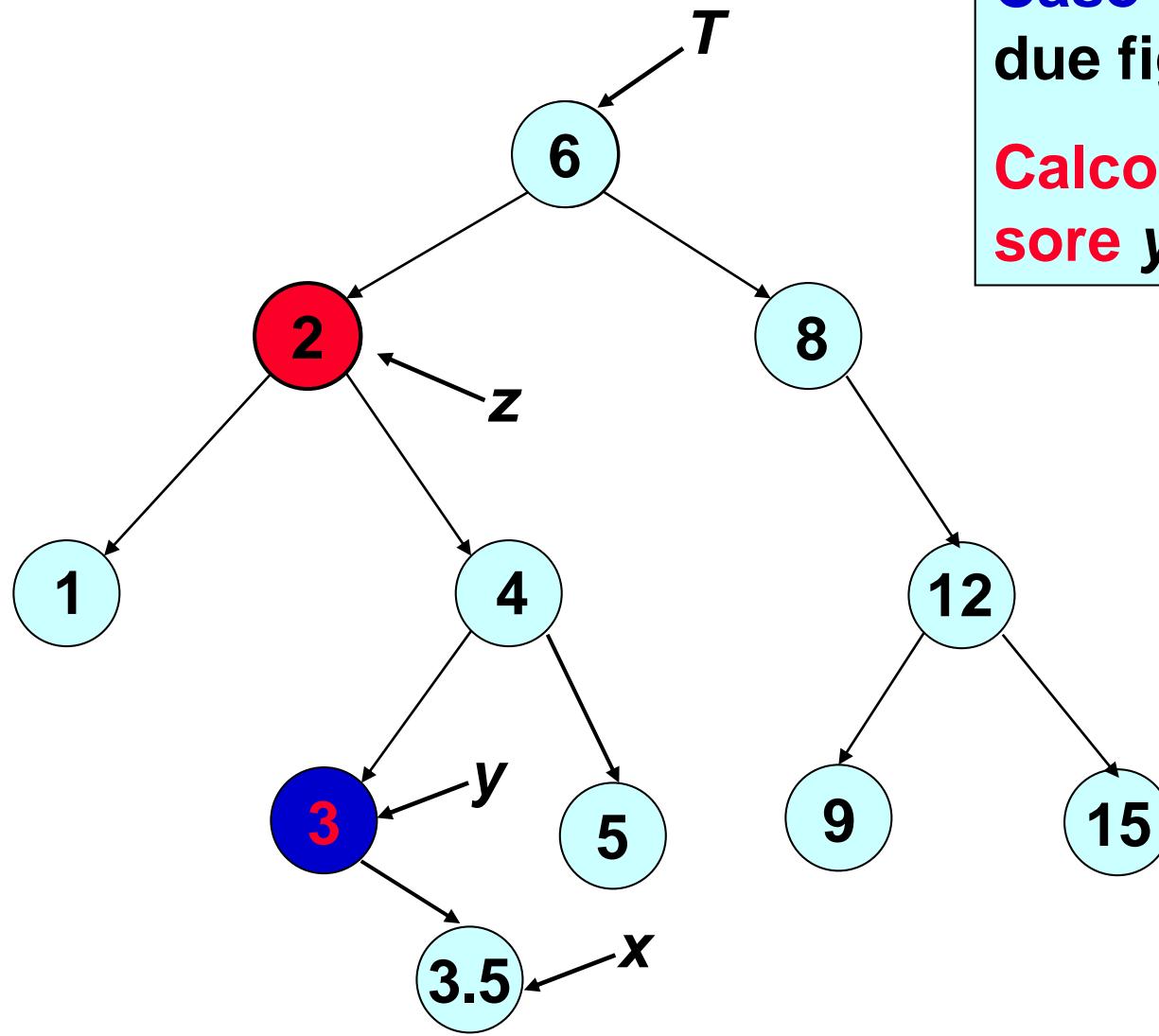
**Scartare il nodo e
connettere il padre
al figlio**

ARB: Cancellazione di un nodo (caso III)



Caso III: il nodo ha
due figli

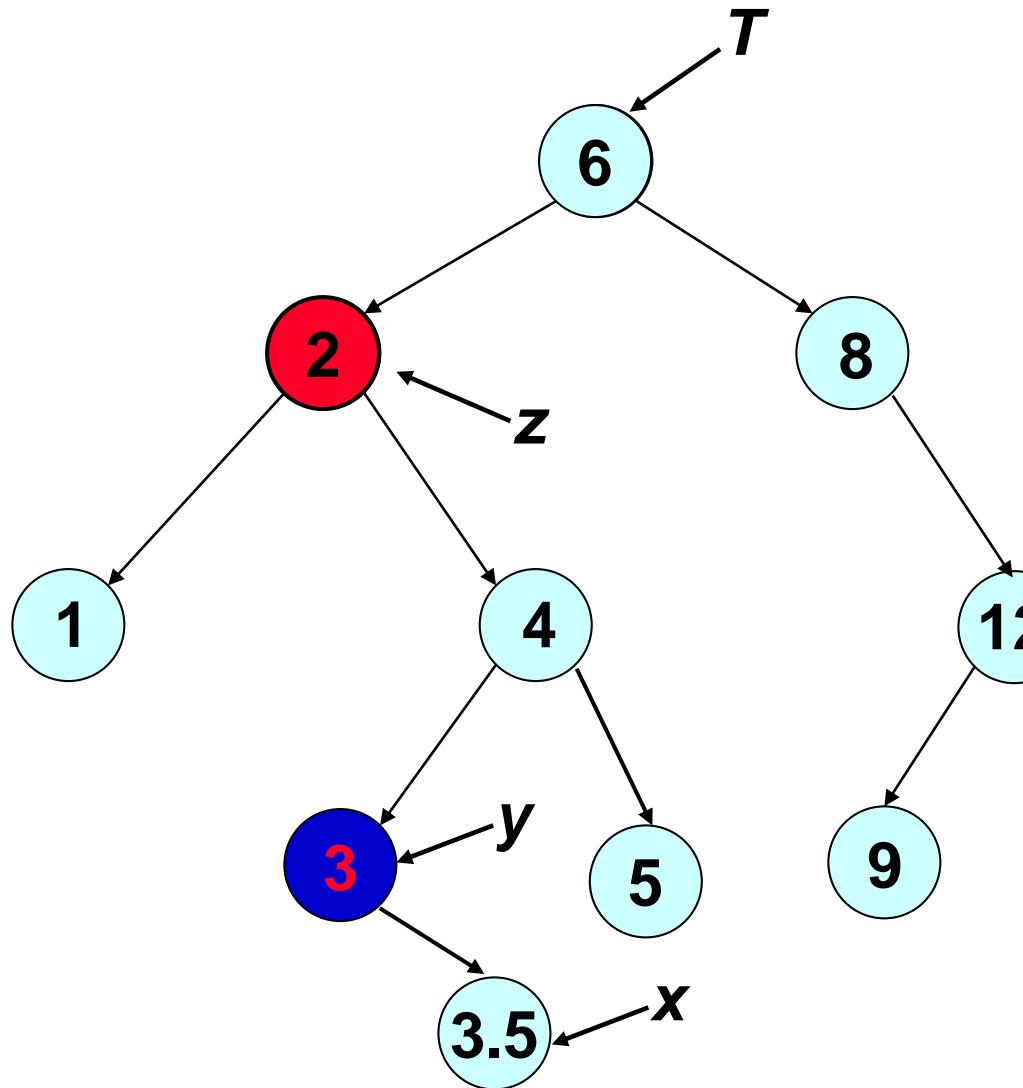
ARB: Cancellazione di un nodo (caso III)



Caso III: il nodo ha
due figli

Calcolare il succe-
sore y

ARB: Cancellazione di un nodo (caso III)

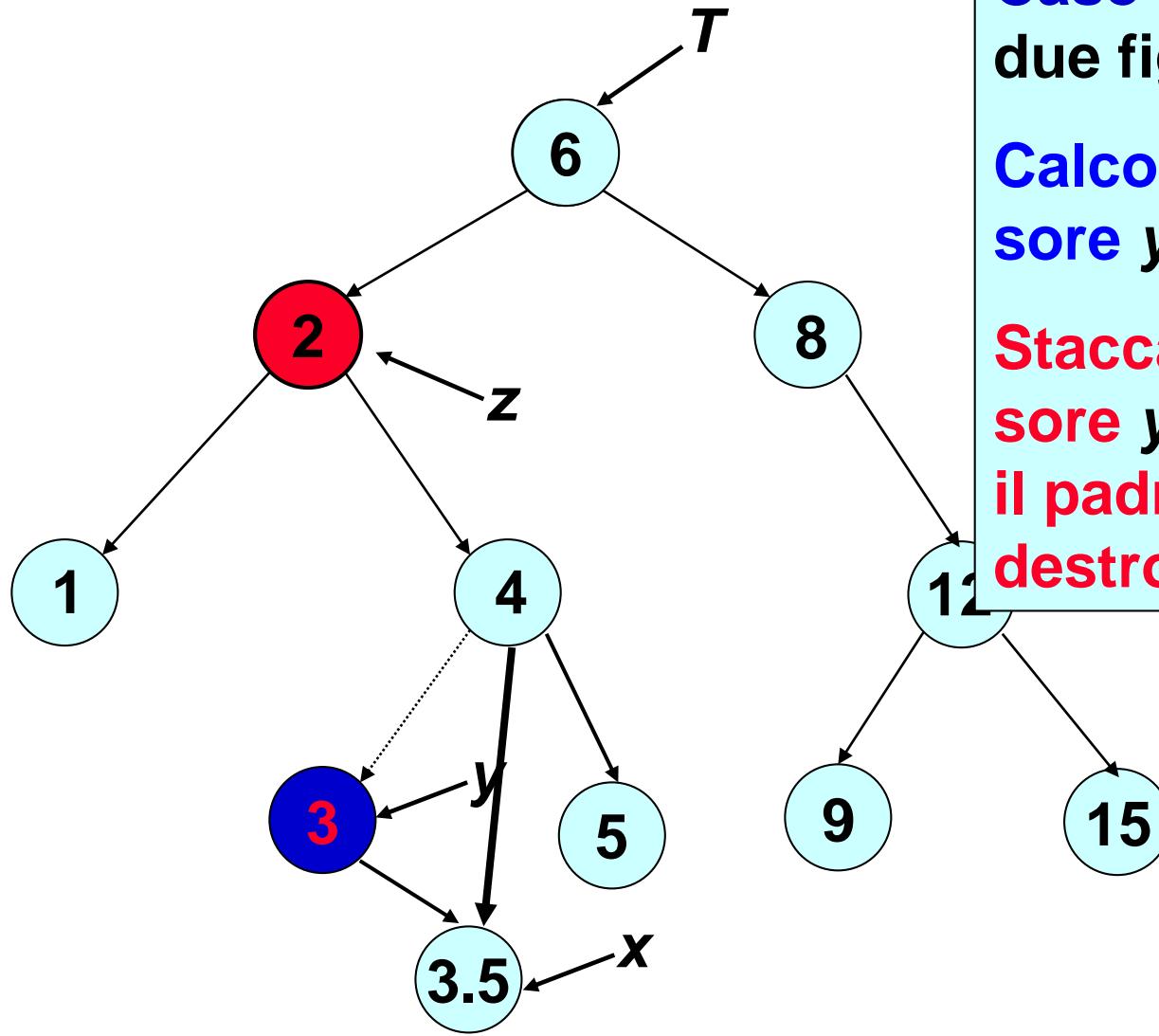


Caso III: il nodo ha
due figli

Calcolare il succe-
sore y

NOTA: Il successore
di un nodo con due
figli **non** può avere
un figlio sinistro

ARB: Cancellazione di un nodo (caso III)

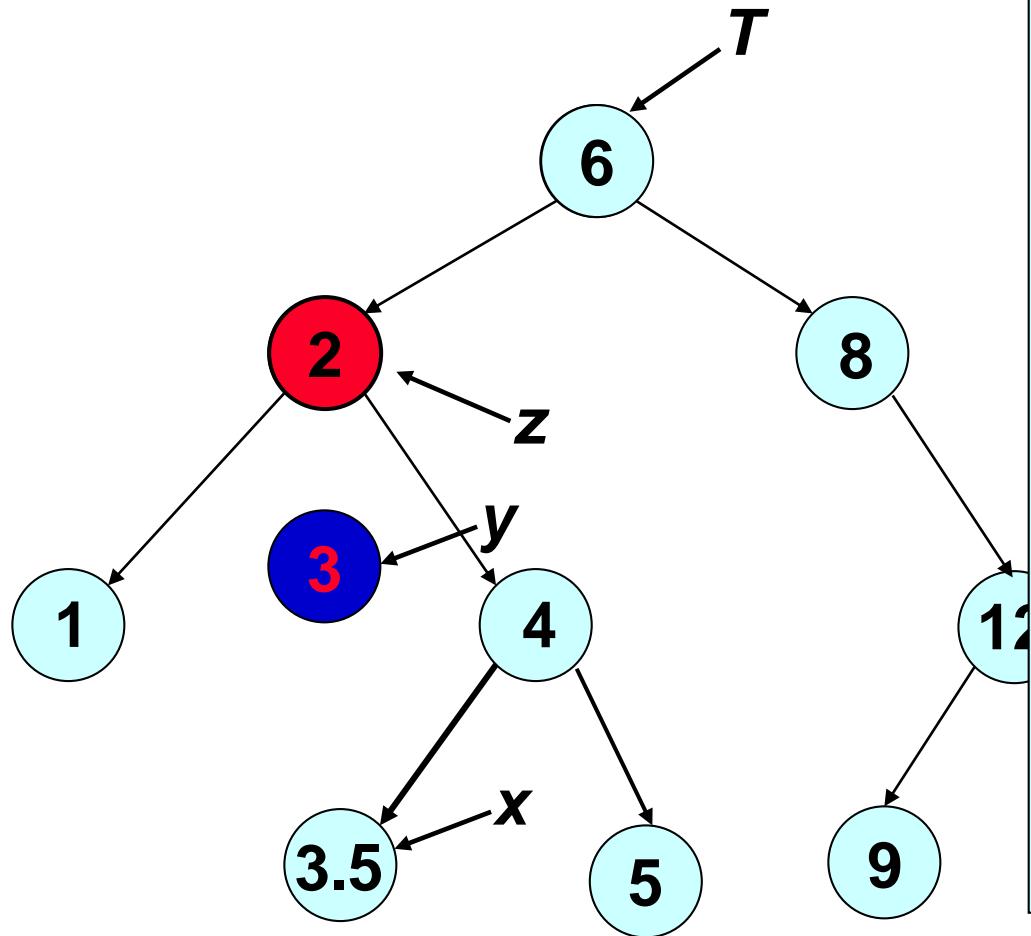


Caso III: il nodo ha due figli

Calcolare il successore y

Staccare il successore y e connettere il padre al figlio destro

ARB: Cancellazione di un nodo (caso III)



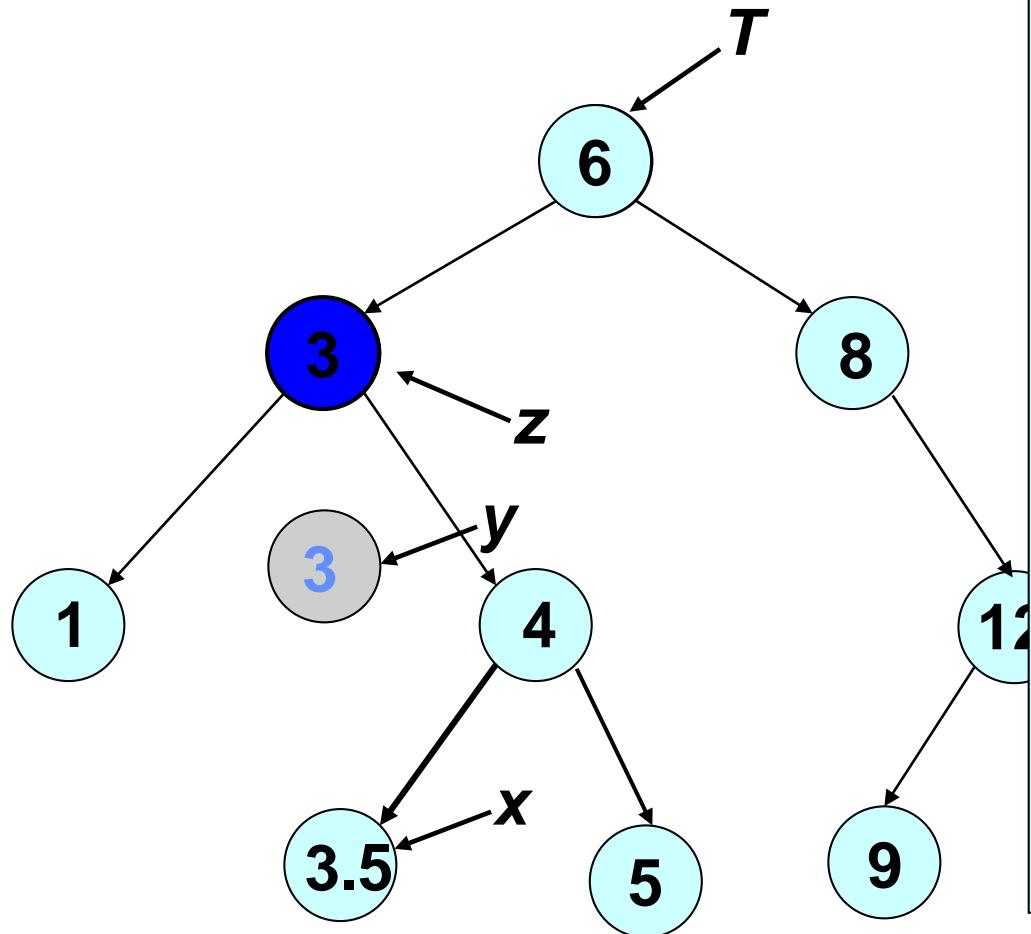
Caso III: il nodo ha due figli

Calcolare il successore y

Staccare il successore y e connettere il padre al figlio destro

Copia il contenuto del successore nel nodo da cancellare

ARB: Cancellazione di un nodo (caso III)



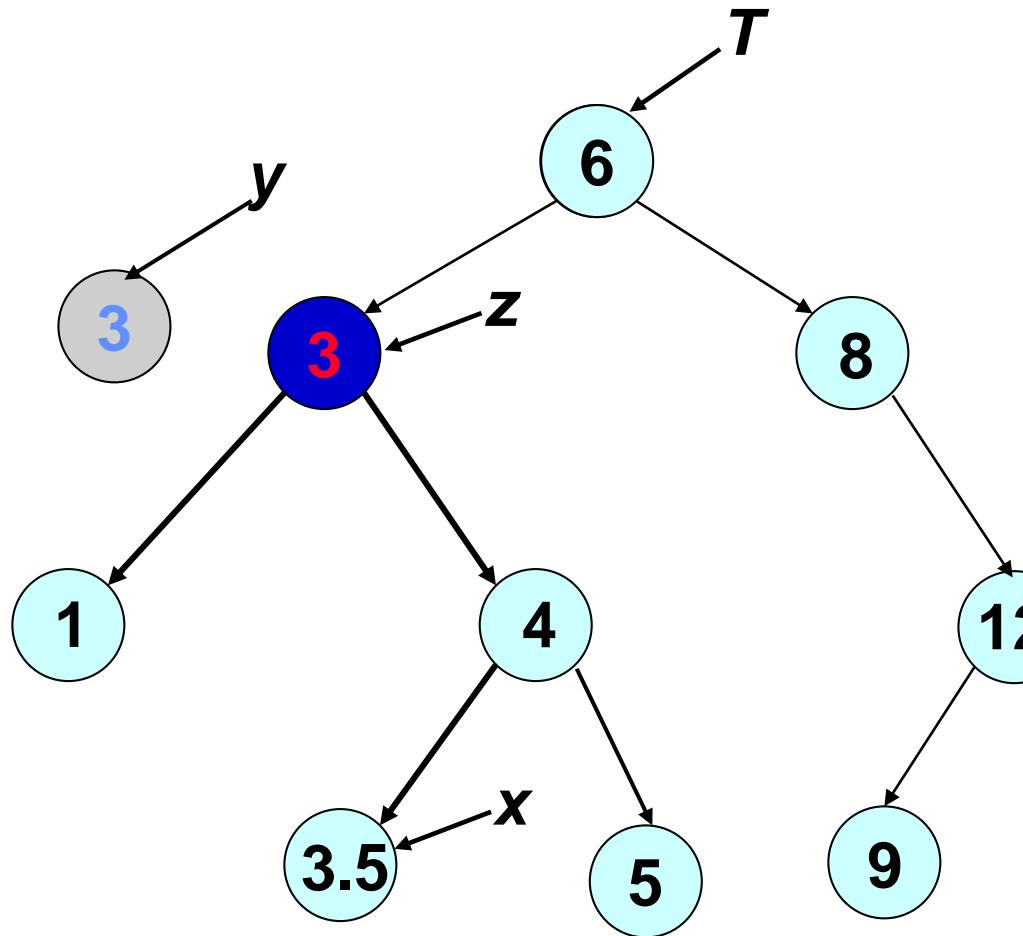
Caso III: il nodo ha due figli

Calcolare il successore y

Staccare il successore y e connettere il padre al figlio destro

Copia il contenuto del successore nel nodo da cancellare

ARB: Cancellazione di un nodo (caso III)



Caso III: il nodo ha due figli

Calcolare il successore y

Staccare il successore y e connettere il padre al figlio destro

Copia il contenuto del successore y nel nodo da cancellare

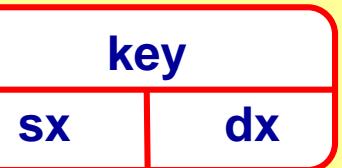
Deallocare il nodo staccato y

ARB: Cancellazione di un nodo

- **Caso I:** Il nodo **non ha figli**. Semplicemente si elimina.
- **Caso II:** Il nodo ha **un solo figlio**. Si **collega il padre del nodo al figlio** e si elimina il nodo.
- **Caso III:** Il nodo ha **due figli**.
 - si **cerca il suo successore** (che ha **un solo figlio destro**);
 - si **elimina il successore** (come in **Caso II**);
 - si **copiano i campi valore** del successore **nel nodo da eliminare**.

ABR-Cancella-iter(T, k)

```
P = NIL
z = T
WHILE (z ≠ NIL && z->key ≠ k) DO
    P = z
    IF (z->key > k) THEN z = z->sx
    ELSE z = z->dx
IF (z ≠ NIL) THEN /* k trovato */
    x = ABR-Cancella-Root(z)
    IF P = NIL THEN T = x /* si cancella la radice di T */
    ELSE IF z = P->sx THEN P->sx = x
    ELSE P->dx = x
return T
```



ABR-Cancella-Root(T)

```
IF T ≠ NIL THEN
    IF T->sx ≠ NIL && T->dx ≠ NIL THEN
        tmp = Stacca-Min(T, T->dx)
        "Copia tmp->key in T->key"
    ELSE
        tmp = T
        IF T->dx ≠ NIL THEN T = T->dx
        ELSE T = T->sx
dealloca tmp
return T
```

**Ricerca successore
Caso III**

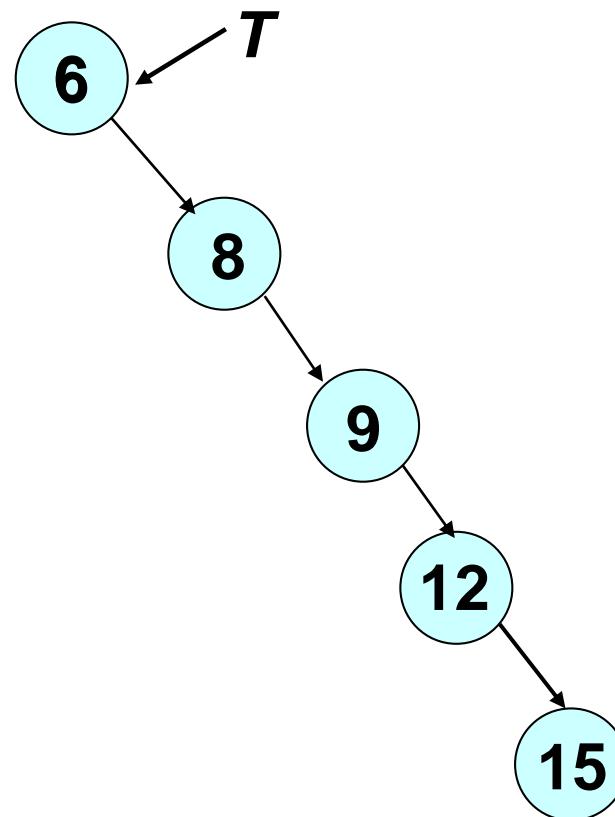
Casi I e II

ARB: costo di Inserimento e Cancellazione

Teorema. Le operazioni di **Inserimento** e **Cancellazione** sull'insieme dinamico **Albero Binario di Ricerca** possono essere eseguite **in tempo $O(h)$** dove **h** è l'altezza dell'albero

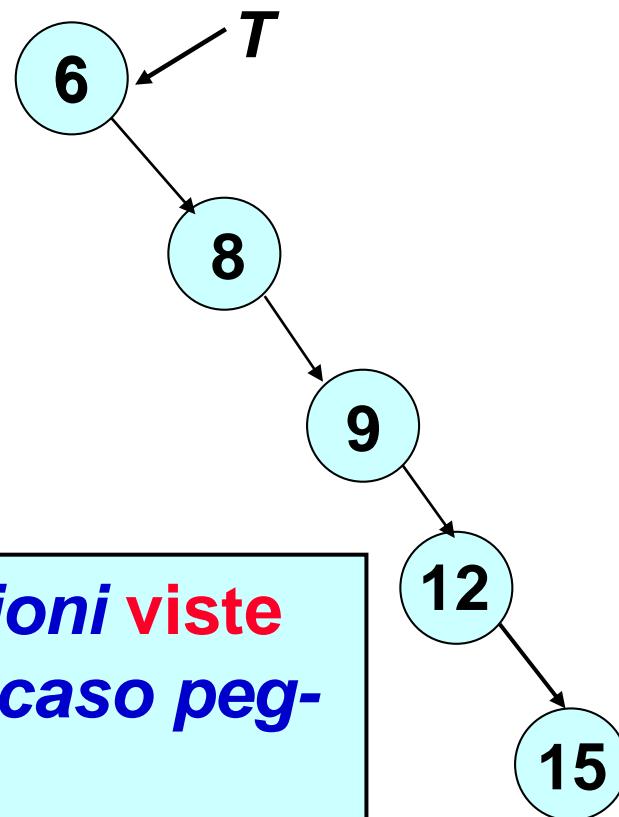
Costo delle operazioni su ABR

L'algoritmo di inserimento *NON* garantisce che l'albero risultante sia bilaciato. Nel caso peggiore l'altezza h può essere pari ad N (numero dei nodi)



Costo delle operazioni su ABR

L'algoritmo di inserimento *NON* garantisce che l'albero risultante sia bilaciato. Nel caso peggiore l'altezza h può essere pari ad N (numero dei nodi)



Quindi tutte le operazioni viste hanno costo $O(N)$ nel caso peggiore

Costo medio delle operazioni su ABR

Dobbiamo calcolare la *lunghezza media* $a(n)$ del *percorso di ricerca*.

- Assumiamo che le chiavi arrivino in ordine casuale (e che tutte abbiano *uguale probabilità* di presentarsi)
- La probabilità che la chiave i sia la radice è allora $1/n$

$$a(n) = \frac{1}{n} \sum_{j=1}^n p_j$$

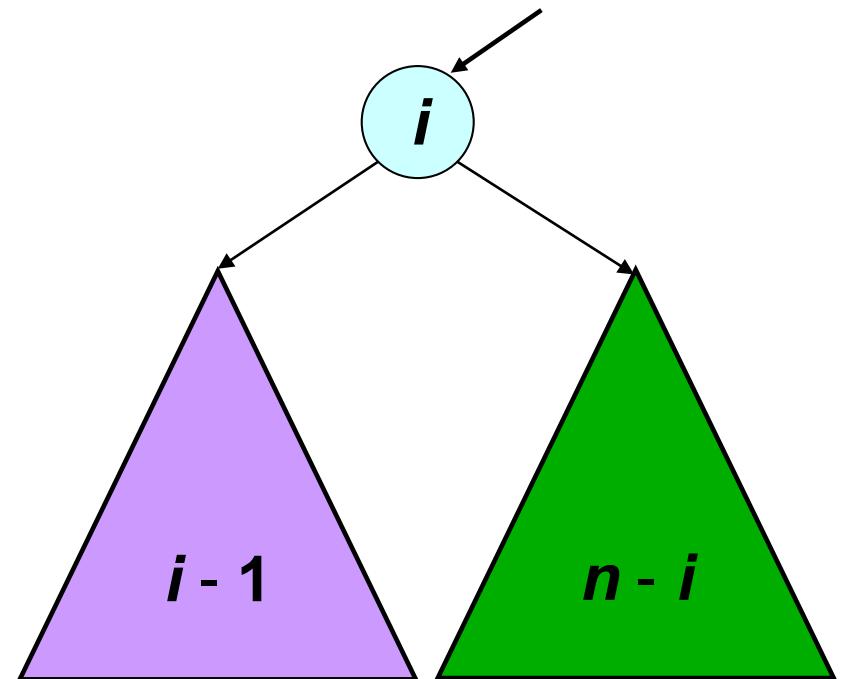
p_j è la lunghezza media del percorso al nodo j

Costo delle operazioni su ABR

Se i è la radice, allora

- il sottoalbero sinistro avrà $i - 1$ nodi e
- il sottoalbero destro avrà $n - i$ nodi

$$a(n) = \frac{1}{n} \sum_{j=1}^n p_j$$

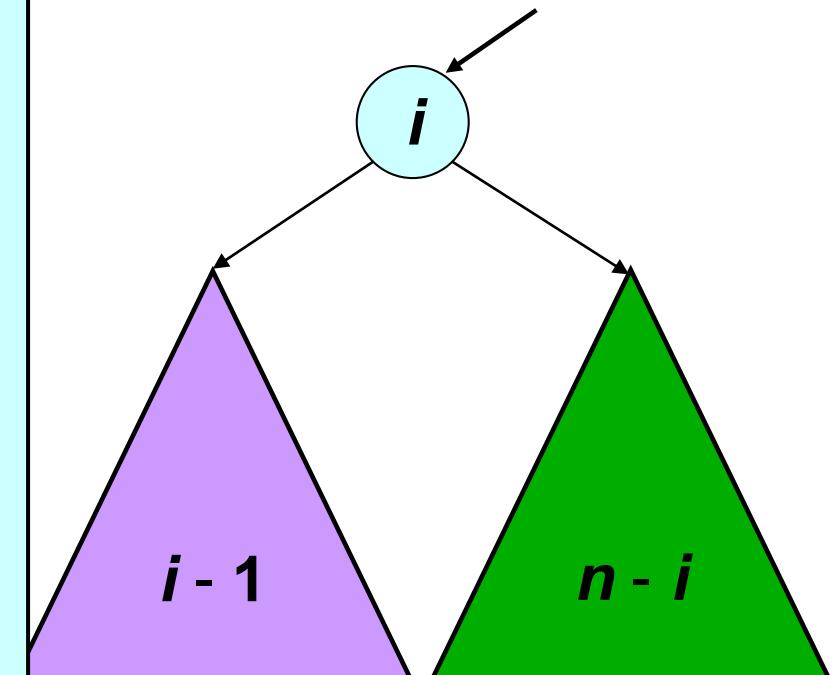


Costo delle operazioni su ABR

Se i è la radice, allora

- il sottoalbero sinistro avrà $i - 1$ nodi e
- il sottoalbero destro avrà $n - i$ nodi
- gli $i - 1$ nodi a sinistra hanno lunghezza media del percorso $a(i-1)+1$
- la radice ha lunghezza del percorso pari ad 1
- gli $n - i$ nodi a sinistra hanno lunghezza media del percorso $a(n-i)+1$

$$a(n) = \frac{1}{n} \sum_{j=1}^n p_j$$



Costo delle operazioni su ABR

$a^i(n)$ sia la lunghezza media del percorso di ricerca con n chiavi quando la radice è la chiave i

$$a^i(n) = [a(i-1) + 1] \frac{i-1}{n} + 1 \frac{1}{n} + [a(n-i) + 1] \frac{n-i}{n}$$

$a(i-1)$ è la lunghezza media del percorso di ricerca con $i-1$ chiavi

$a(n-i)$ è la lunghezza media del percorso di ricerca con $n-i$ chiavi

Costo delle operazioni su ABR

$a^i(n)$ sia la lunghezza media del percorso di ricerca con n chiavi quando la radice è la chiave i

$$a^i(n) = [a(i-1)+1] \frac{i-1}{n} + 1 \frac{1}{n} + [a(n-i)+1] \frac{n-i}{n}$$

allora

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

$a(n)$ è la media degli $a^i(n)$, dove ciascun $a^i(n)$ ha probabilità $1/n$, cioè la probabilità che proprio la chiave i sia la radice dell'albero.

Costo delle operazioni su ABR

$$a^i(n) = [a(i-1)+1] \frac{i-1}{n} + 1 \frac{1}{n} + [a(n-i)+1] \frac{n-i}{n}$$

allora

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

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

a(n) è la media degli ***aⁱ(n)***, dove ciascun ***aⁱ(n)*** ha probabilità ***1/n***

Costo delle operazioni su ABR

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

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

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

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

Costo delle operazioni su ABR

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

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

Costo delle operazioni su ABR

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

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

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

Costo delle operazioni su ABR

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

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

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

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

Costo delle operazioni su ABR

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

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

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

Costo delle operazioni su ABR

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

Dimostrare per induzione

$$a(n) = 2 \frac{n+1}{n} H(n) - 3$$

$$H(n) = 1 + \frac{1}{2} + \frac{1}{3} + \dots + \frac{1}{n}$$

Funzione armonica

Costo delle operazioni su ABR

$$a(n) = 2 \frac{n+1}{n} H(n) - 3$$

$$a(n) = 2(\ln n + \gamma) - 3 = 2 \ln n - c$$

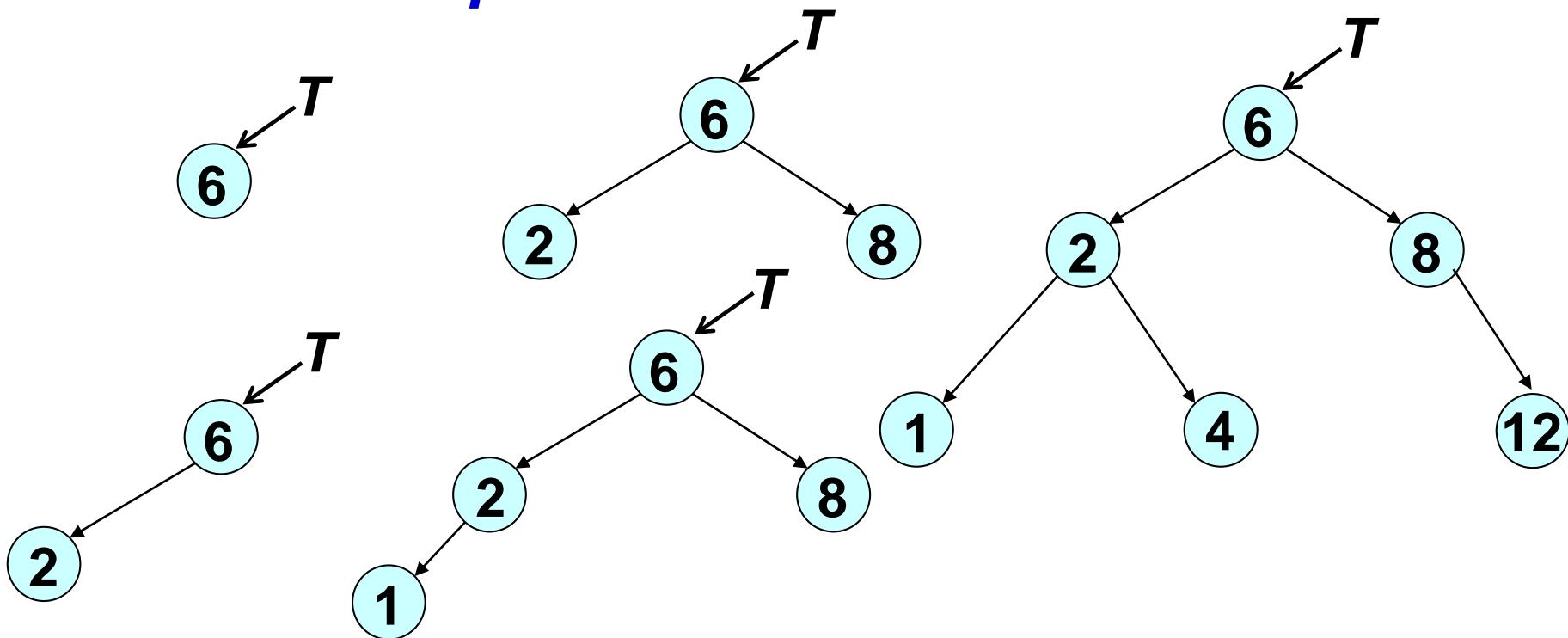
Formula di Eulero

$$H(n) = \gamma + \ln n + \frac{1}{2n} - \frac{1}{12n^2} + \dots$$

dove $\gamma \approx 0.577$

Alberi perfettamente bilanciati

Definizione: Un albero binario si dice **Perfettamente Bilanciato** se, per ogni nodo *i*, il **numero dei nodi** nel suo **sottoalbero sinistro** e il **numero dei nodi** del suo **sottoalbero destro** **differiscono al più di 1**



Alberi perfettamente bilanciati

Definizione: Un albero binario si dice **Perfettamente Bilanciato** se, per ogni nodo *i*, il **numero dei nodi** nel suo **sottoalbero sinistro** e il **numero dei nodi** del suo **sottoalbero destro** **differiscono al più** di 1

La **lunghezza media $a'(n)$ del percorso** in un **albero perfettamente bilanciato (APB)** con ***n*** nodi è approssimativamente

$$a'(n) = \log n - 1$$

Confronto tra ABR e APB

Il **rappporto** tra la **lunghezza media $a(n)$ del percorso** in un **albero di ricerca** e la **lunghezza media $a'(n)$** nell'**albero perfettamente bilanciato** è (per n sufficientemente grande) è approssimativamente

$$\frac{a(n)}{a'(n)} = \frac{2 \ln n - c}{\log n - 1} \cong \frac{2 \ln n}{\log n} = 2 \ln 2 \cong 1,386$$

(trascurando i termini costanti)

Confronto tra ABR e APB

Ciò significa che, se anche *bilanciassimo* perfettamente l'albero *dopo ogni inserimento* il *guadagno sul percorso medio* che otterremmo **NON supererebbe il 39%**.

$$\frac{a_n}{a'_n} = \frac{2 \ln n - c}{\log n - 1} = \frac{2 \ln n}{\log n} = 2 \ln 2 \cong 1.386$$

Sconsigliabile nella maggior parte dei casi, **a meno che** il *numero dei nodi* e il *rapporto tra ricerche e inserimenti* **siano molto grandi**.

Algoritmi e Strutture Dati (Mod. B)

Grafi

Grafi

I grafi sono uno *strumento di rappresentazione* (modellazione) di problemi.

La soluzione di molti problemi può essere ricondotta alla soluzione di opportuni problemi su grafi.

- *Introduzione ai grafi*
 - Definizioni e rappresentazione di grafi
 - Algoritmi di visita di grafi
 - Visita in ampiezza (*BFS*)
 - Visita in profondità (*DFS*)
 - Applicazioni: Ordinamento Topologico, Componenti Fortemente Connesse,...

Cos'è un grafo

Esempio:

Studenti

Marco

Carla

Andrea

Laura

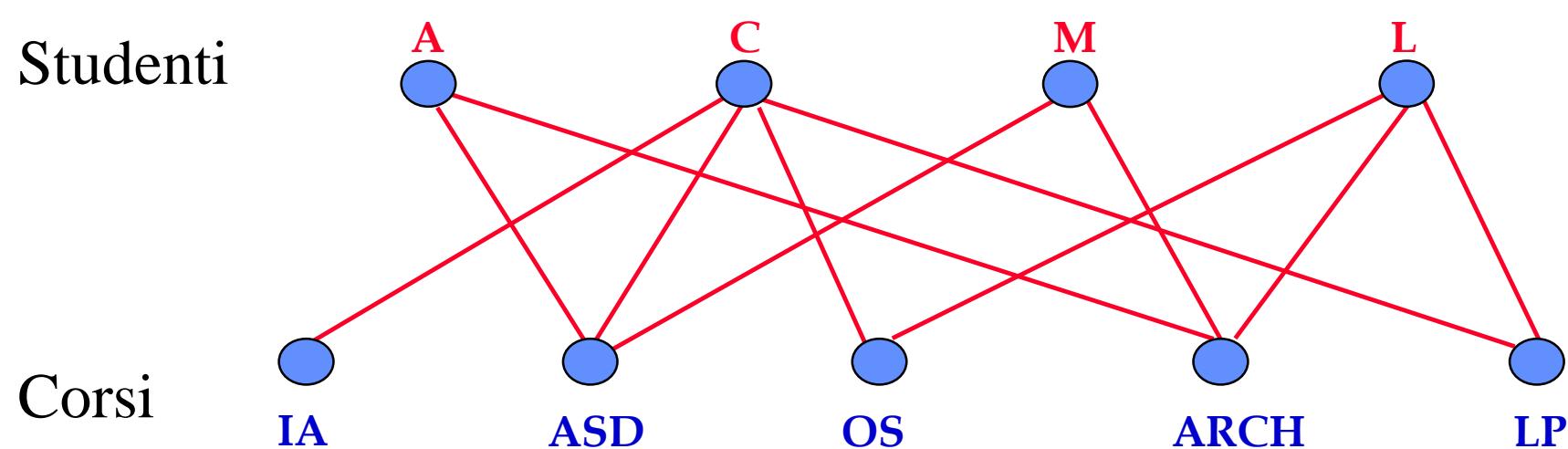
Corsi

ASD, ARCH

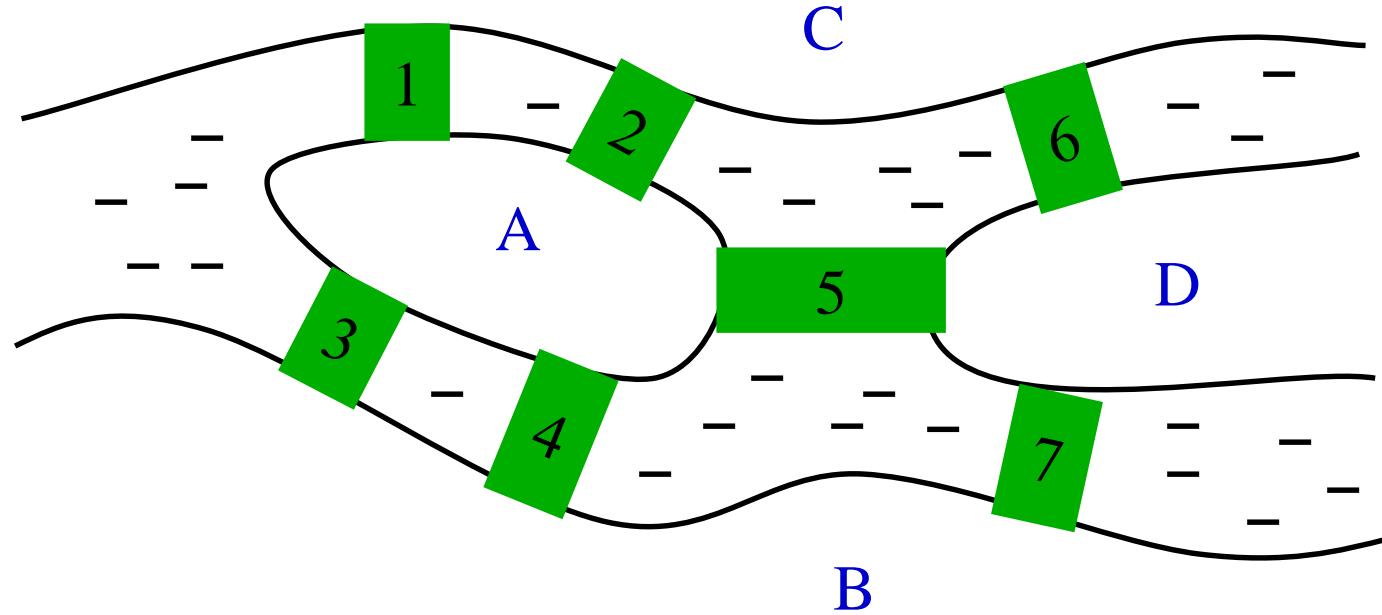
IA, ASD, OS, LP

ASD, ARCH

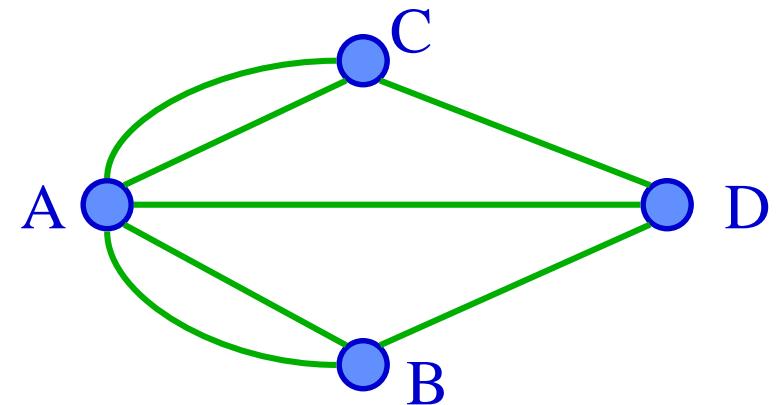
OS, ARCH, LP



I ponti di Königsberg



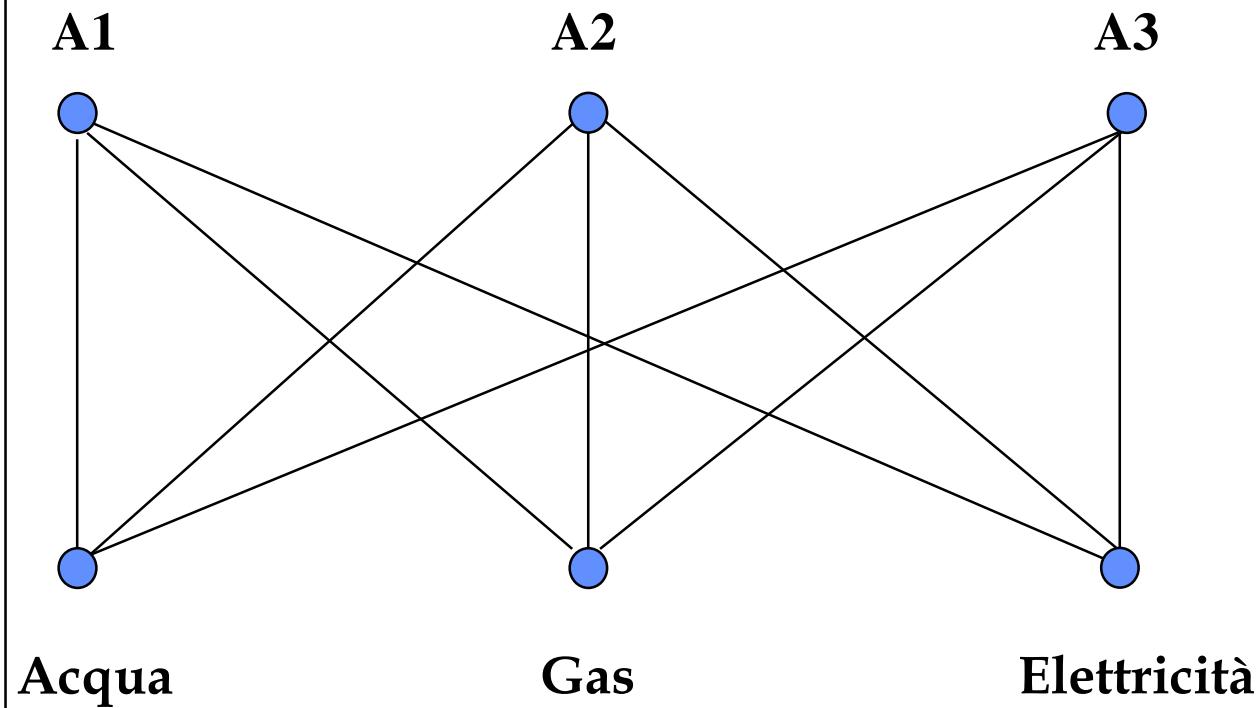
È possibile attraversare tutti i ponti esattamente una sola volta?



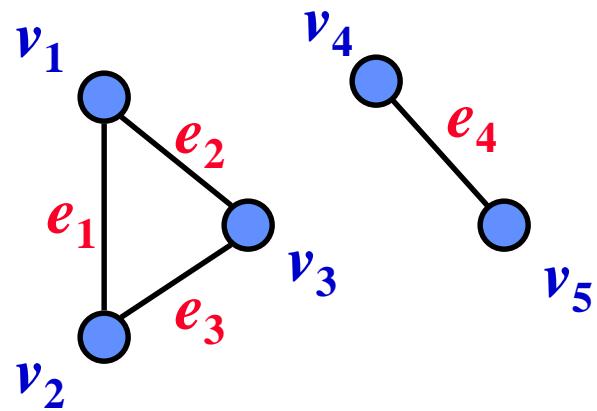
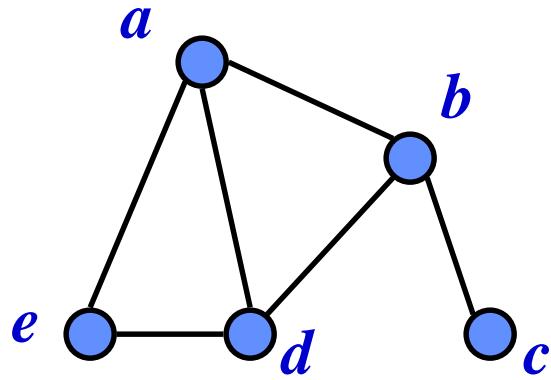
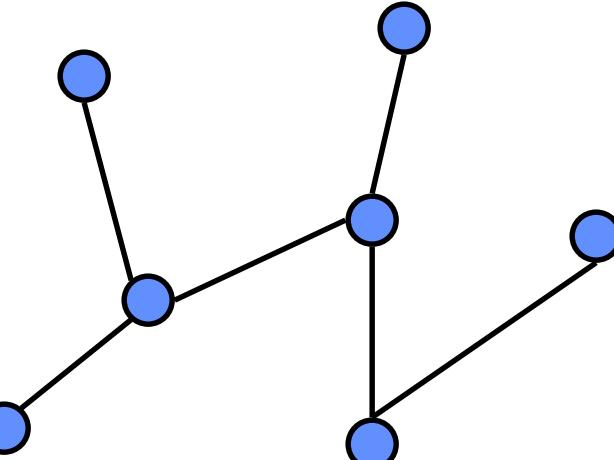
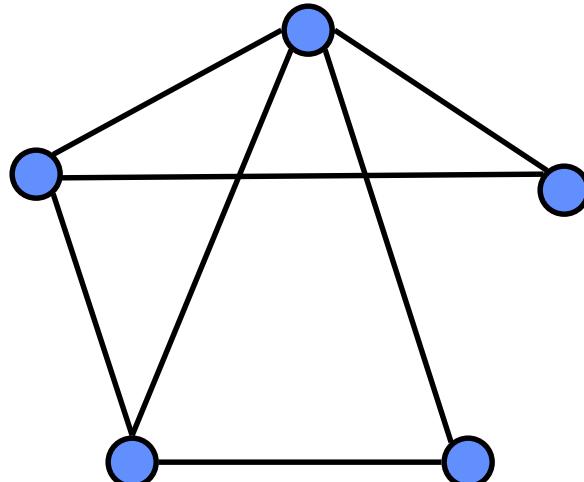
Rappresentazione a grafi di problemi

Problema: Supponiamo dover connettere tre abitazioni A1, A2 e A3 tramite tubature per fornile di Acqua, Gas ed Elettricità.

Se però assumiamo che le tubature vadano posizionate alla stessa profondità, è possibile offrire la fornitura a tutte le abitazioni senza far incrociare le tubature?



Esempi di grafi



Definizione di grafo

Un **grafo G** è una coppia di elementi (V, E) dove:

V è un insieme detto insieme dei **vertici**

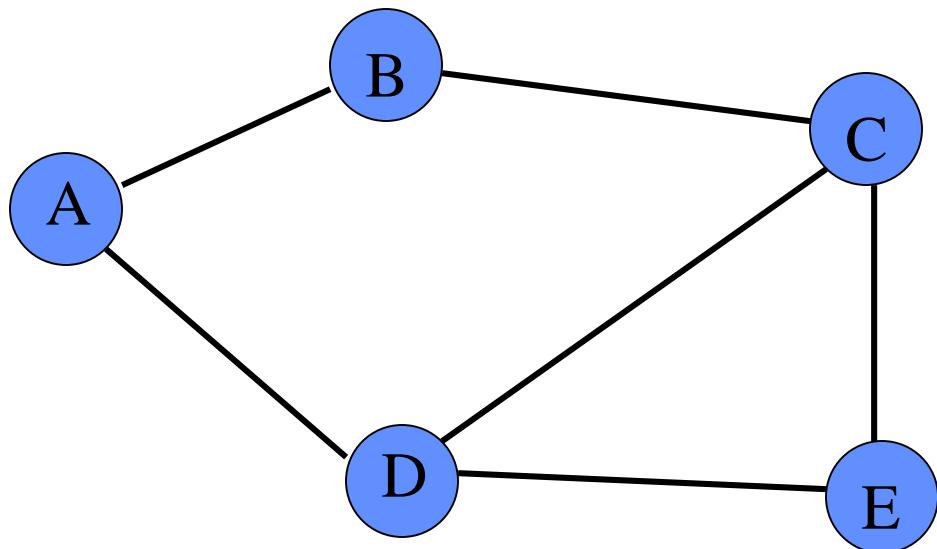
E è un insieme di **copie di vertici** detto insieme degli **archi**

Definizione di grafo

Un **grafo G** è una coppia di elementi (V, E) dove:

V è un insieme detto insieme dei **vertici**

E è un insieme di **copie di vertici** detto insieme degli **archi**

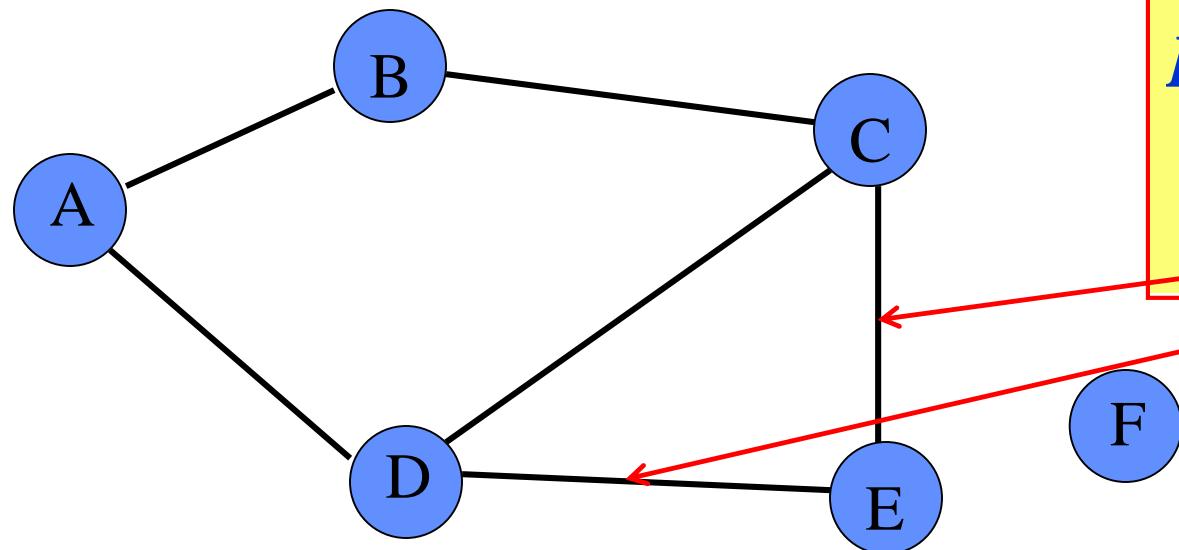


$$V = \{A, B, C, D, E, F\}$$
$$E = \{(A,B), (A,D), (B,C), (C,D), (C,E), (D,E)\}$$

Definizione di grafo

Un **arco** è una coppia (v, w) di vertici in V , cioè

- $v \in V$ e $w \in V$



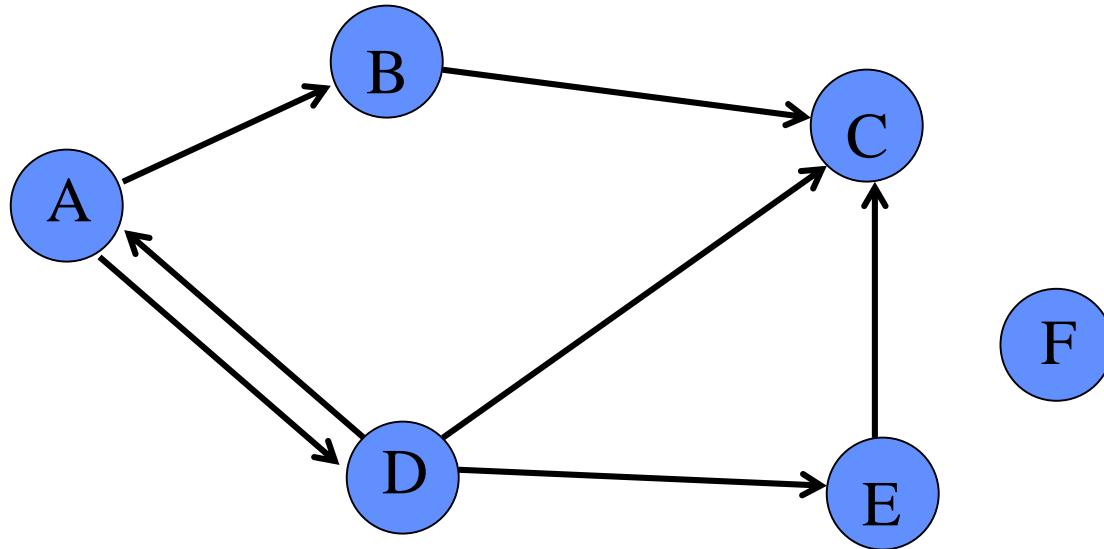
$$V = \{A, B, C, D, E, F\}$$
$$E = \{(A, B), (A, D), (B, C), (C, D), (C, E), (D, E)\}$$

Tipi di grafi: grafi orientati

Un **grafo orientato** G è una coppia (V, E) dove:

V è un insieme detto insieme dei **vertici**

E è una *relazione binaria* tra vertici detta insieme degli **archi** (cioè, $E \subseteq V \times V$)

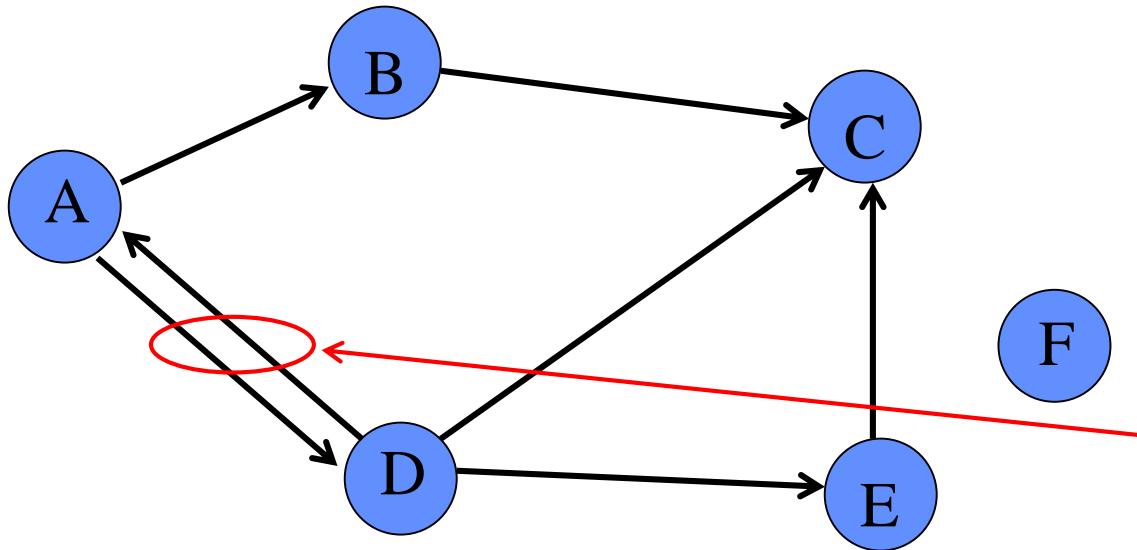


I tipi di grafi: grafi orientati

Un **grafo orientato** G è una coppia (V, E) dove:

V è un insieme detto insieme dei **vertici**

E è una *relazione binaria* tra vertici detta insieme degli **archi** (cioè, $E \subseteq V \times V$)



$V = \{A, B, C, D, E, F\}$
 $E = \{(A, B), (A, D), (B, C), (D, C), (E, C), (D, E), (D, A)\}$

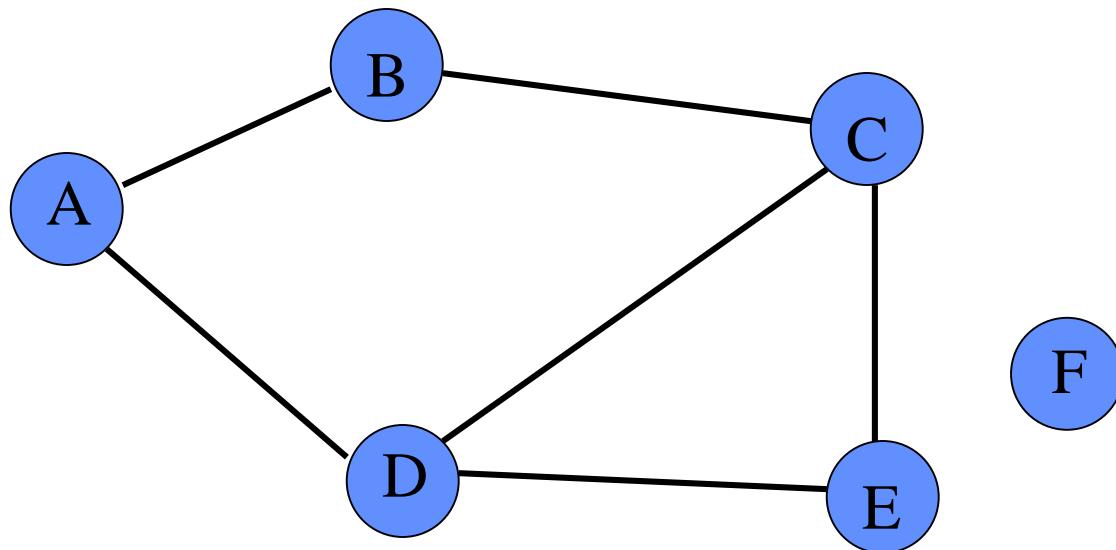
(A, D) e (D, A) denotano due archi diversi

Tipi di grafi: grafi non orientati

Un **grafo non orientato** G è una coppia (V, E) dove:

V è un insieme detto insieme dei **vertici**

E è un insieme di coppie ***non ordinate*** di vertici
detto insieme degli **archi** (cioè, $E \subseteq V \times V$)



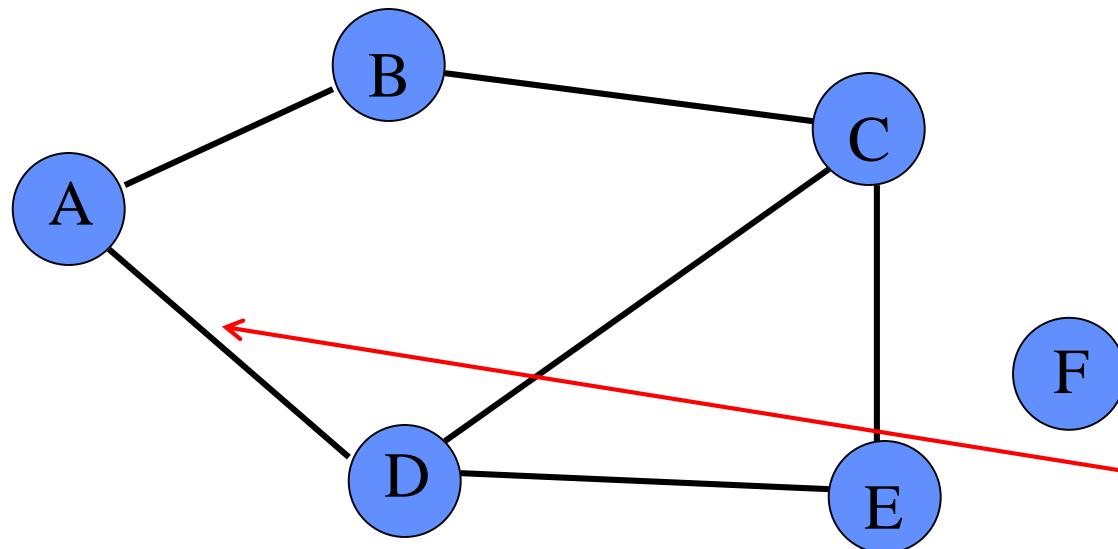
$$V = \{A, B, C, D, E, F\}$$
$$E = \{(A,B), (A,D), (B,C), (C,D), (C,E), (D,E)\}$$

Tipi di grafi: grafi non orientati

Un **grafo non orientato** G è una coppia (V, E) dove:

V è un insieme detto **insieme dei vertici**

E è un insieme di coppie ***non ordinate*** di vertici
detto **insieme degli archi** (cioè, $E \subseteq V \times V$)

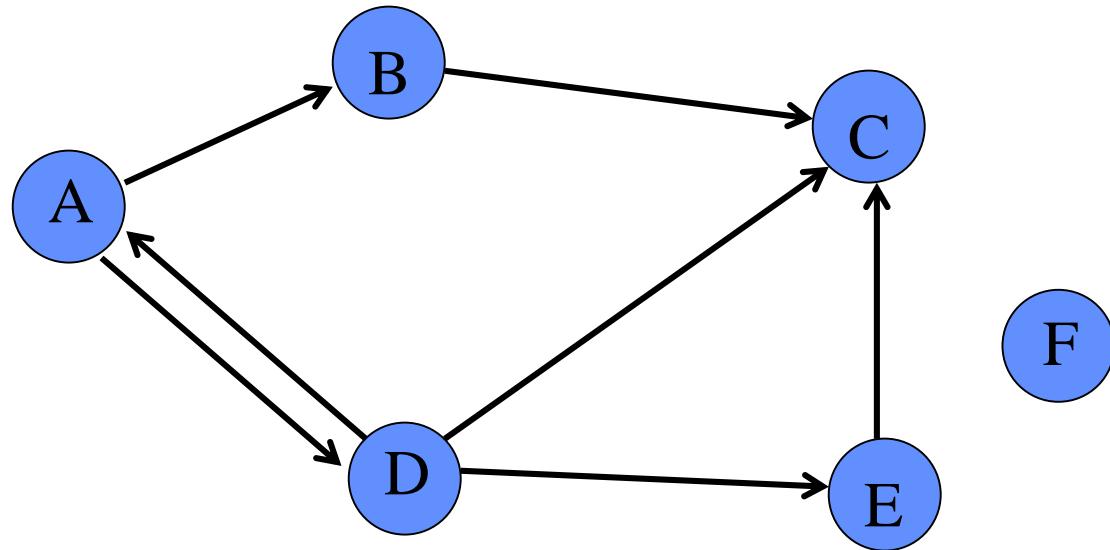


$$V = \{A, B, C, D, E, F\}$$
$$E = \{(A, B), (A, D), (B, C), (C, D), (C, E), (D, E)\}$$

(A, D) e (D, A) denotano
lo stesso arco

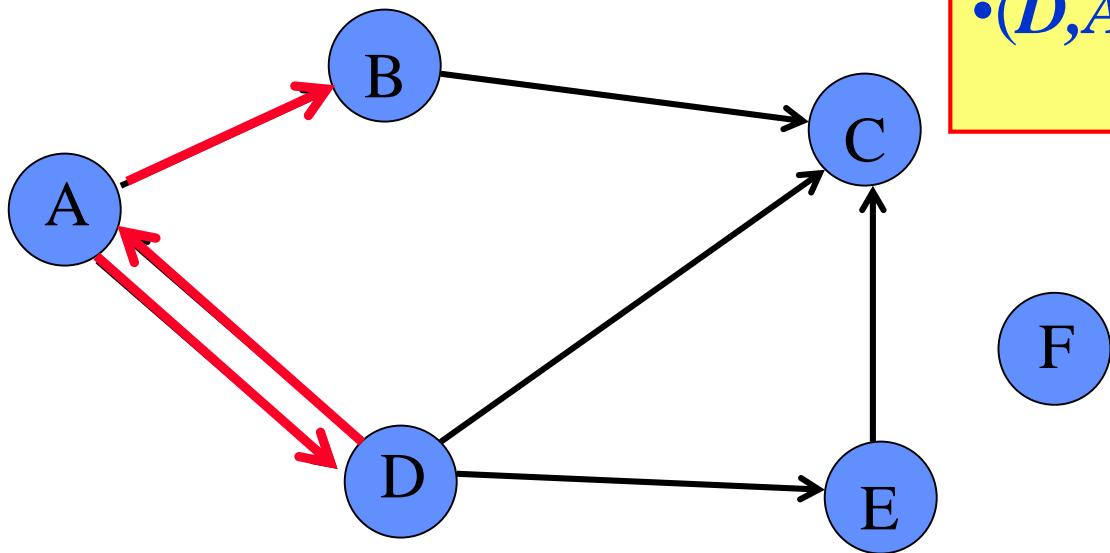
Definizioni sui grafi

In un grafo orientato, un arco $(w,v) \in E$ si dice *incidente* da w in v



Definizioni sui grafi

In un grafo orientato, un arco $(w,v) \in E$ si dice *incidente* da w in v



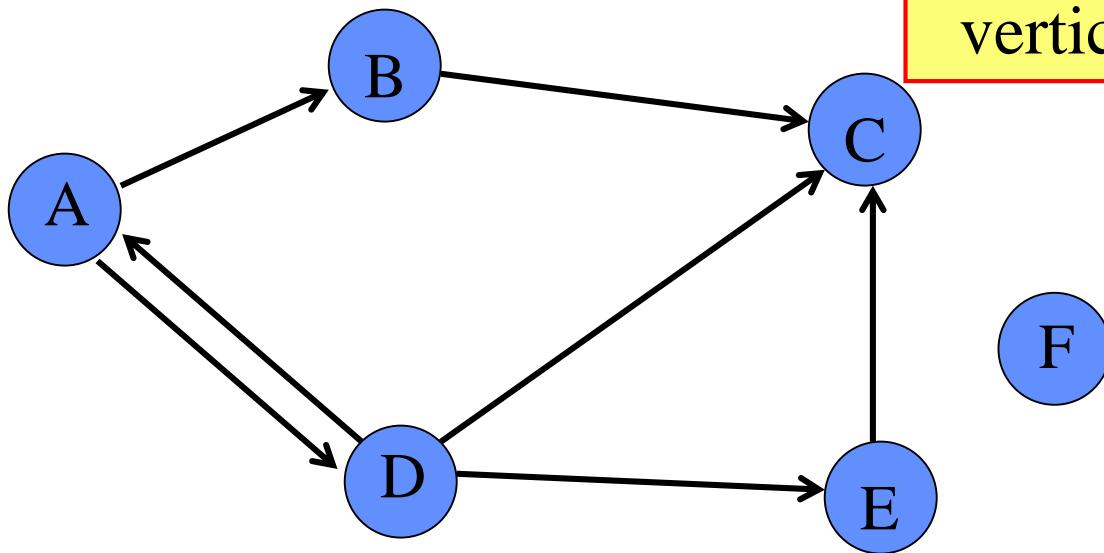
- (A,B) è *incidente* da A a B
- (A,D) è *incidente* da A a D
- (D,A) è *incidente* da D a A

Definizioni sui grafi

Un vertice w si dice **adiacente** a v se e solo se

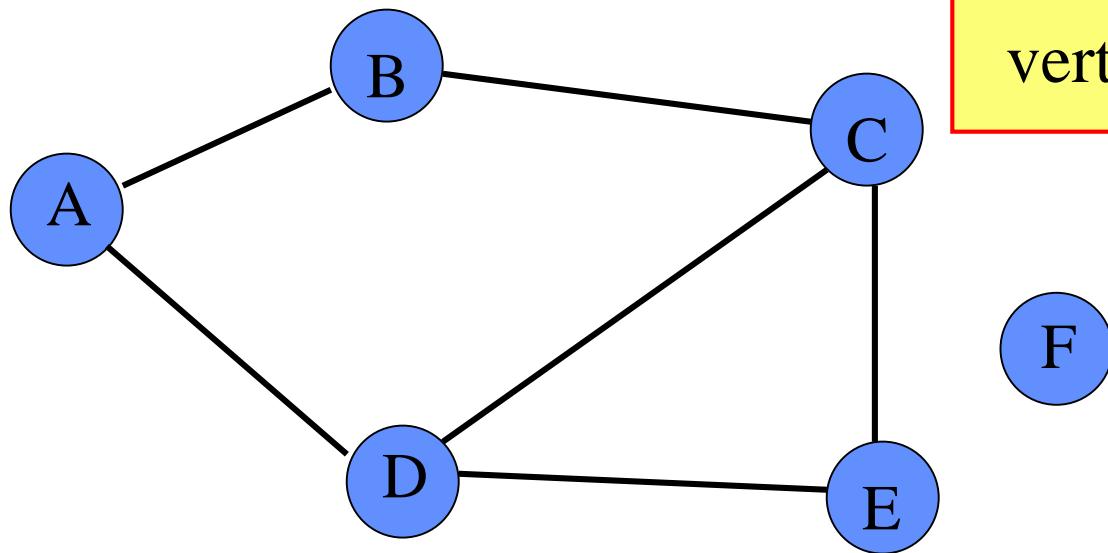
- $(v, w) \in E$.

- B è **adiacente** ad A
- C è **adiacente** a B e a D
- A è **adiacente** a D e vice versa
- B **NON** è **adiacente** a D **NÉ** a C
- F **NON** è **adiacente** ad alcun vertice



Definizioni sui grafi

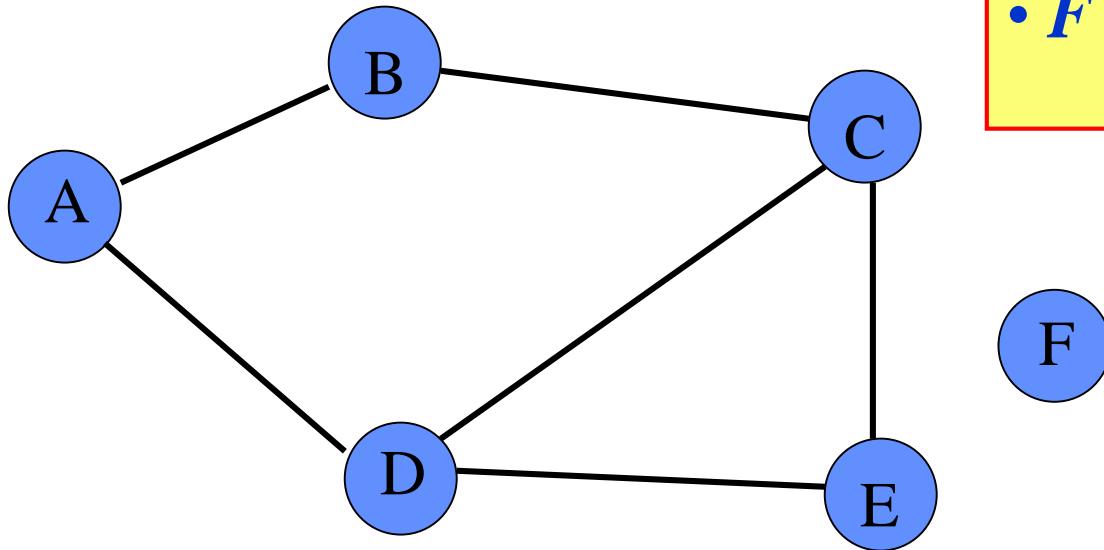
In un *grafo non orientato* la relazione di *adiacenza* tra vertici è *simmetrica*



- *A* è *adiacente* a *D* e vice versa
- *B* è *adiacente* a *A* e vice versa
- *F* ***NON*** è *adiacente* ad alcun vertice

Definizioni sui grafi

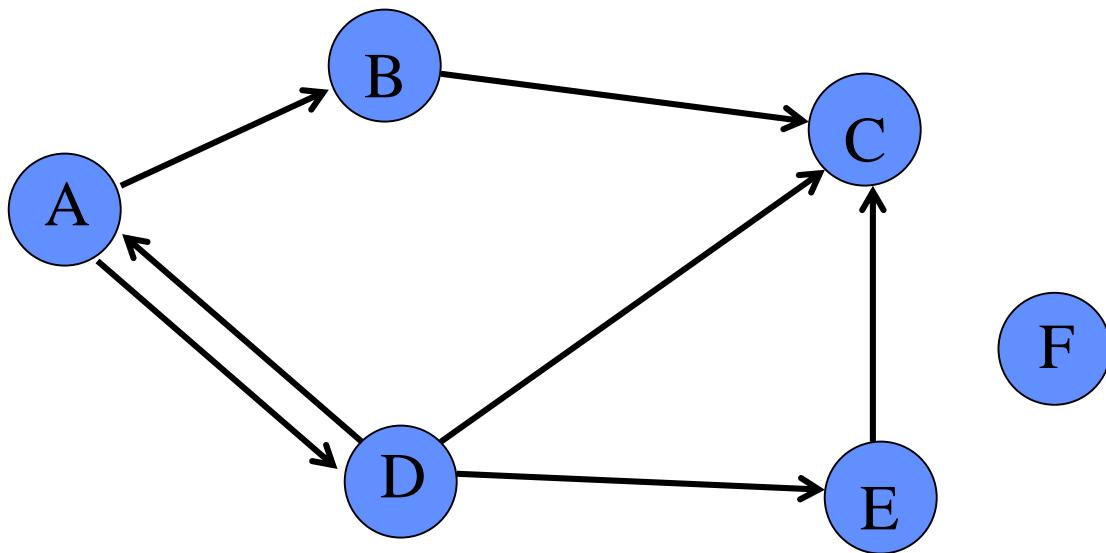
In un *grafo non orientato* il *grado* di un *vertice* è il *numero di archi* che da esso si dipartono



- *A, B ed E* hanno *grado 2*
- *C e D* hanno *grado 3*
- *F* ha *grado 0*

Definizioni sui grafi

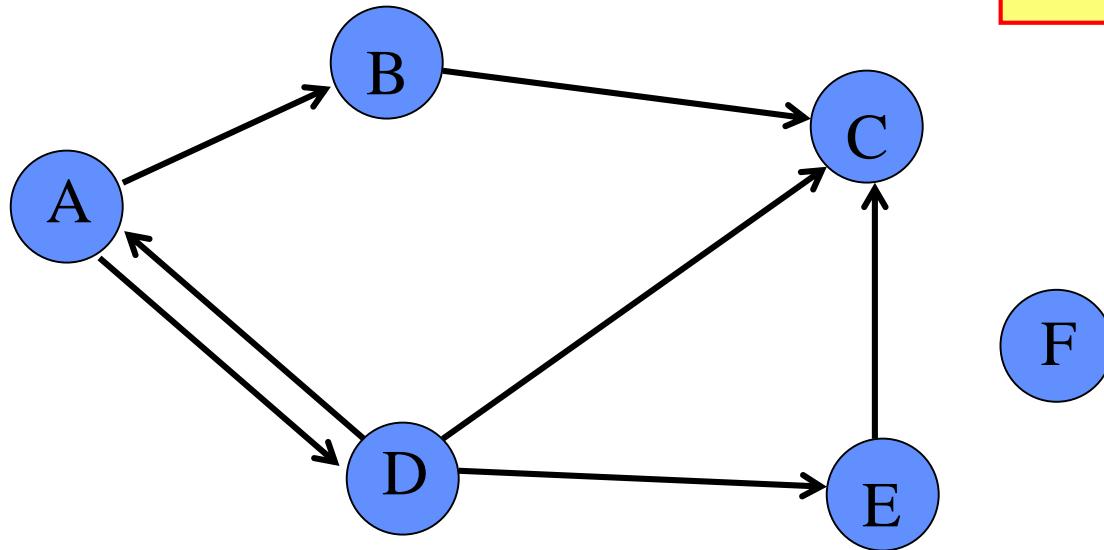
In un *grafo orientato* il *grado entrante (uscente)* di un *vertice* è il *numero di archi incidenti in (uscenti da)* esso



- A ha *grado uscente 2* e *grado entrante 1*
- B ha *grado uscente 1* e *grado entrante 1*
- C ha *grado uscente 0* e *grado entrante 3*
- D ha *grado uscente 3* e *grado entrante 1*

Definizioni sui grafi

In un *grafo orientato* il *grado* di un *vertice* è la somma del suo *grado entrante* e del suo *grado uscente*

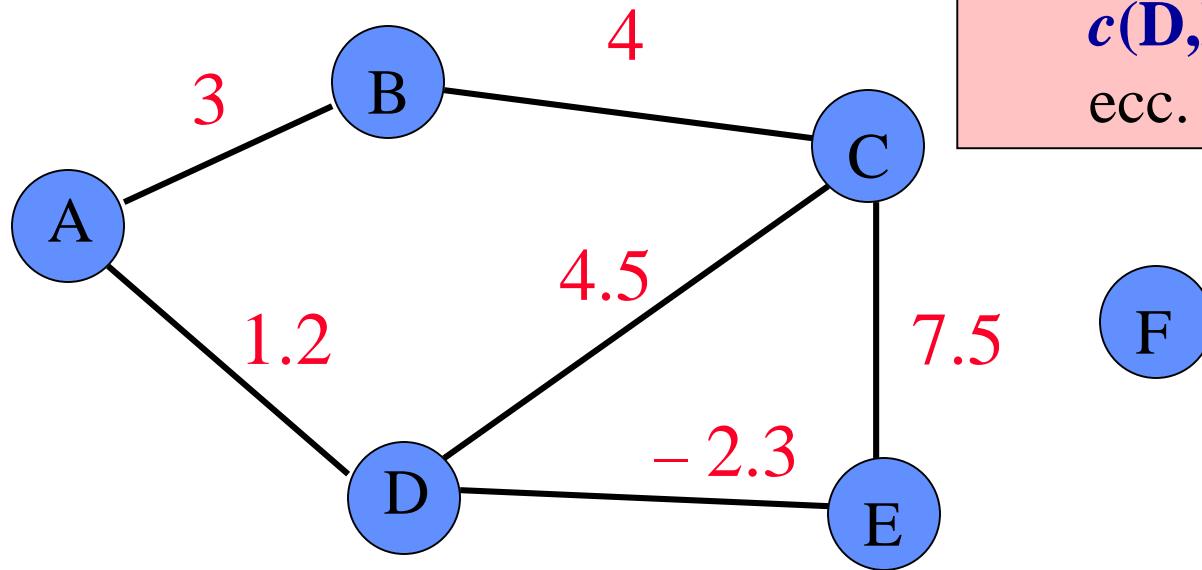


- A e C hanno *grado 3*
- B ha *grado 2*
- D ha *grado 4*

Definizioni sui grafi

In alcuni casi, gli archi hanno un **peso** (o **costo**) associato.

Il costo può essere rappresentato da una **funzione di costo**, $c: E \rightarrow \mathbf{R}$, dove **R** è l'insieme dei numeri reali (o interi).

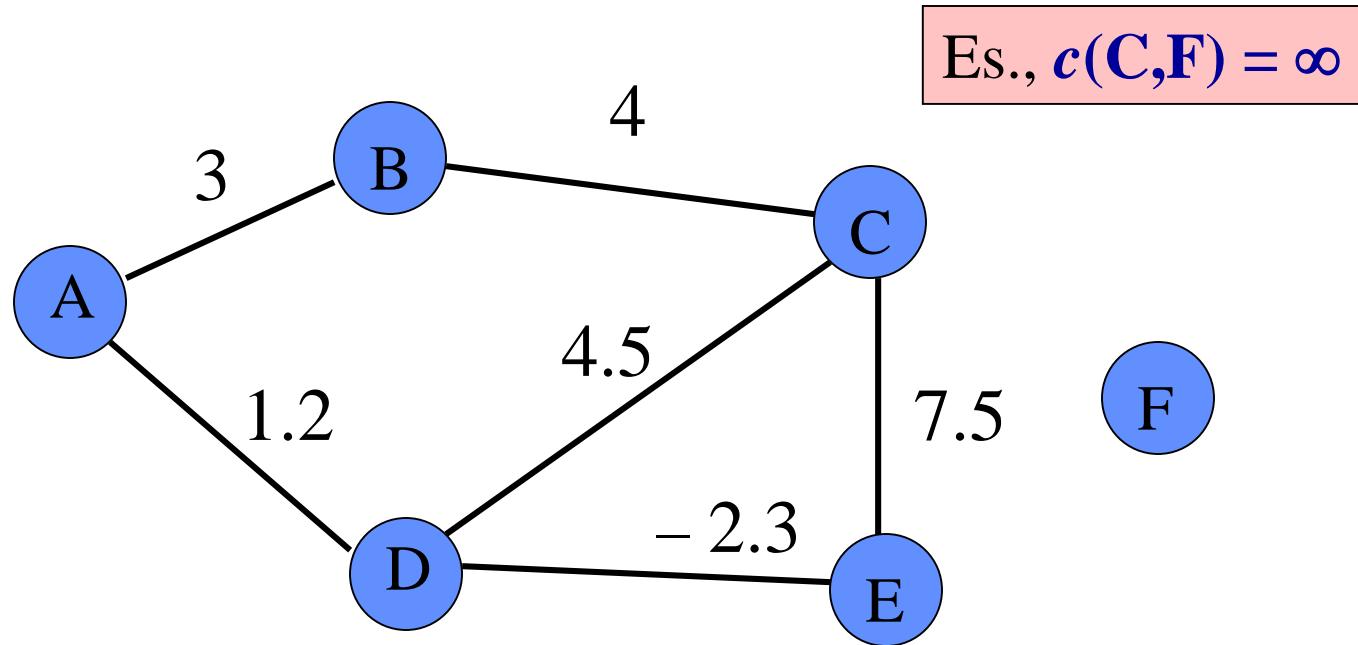


Es., $c(A, B) = 3$,
 $c(D, E) = -2.3$,
ecc.

Definizioni sui grafi

In alcuni casi, gli archi hanno un **peso** (o **costo**) associato.

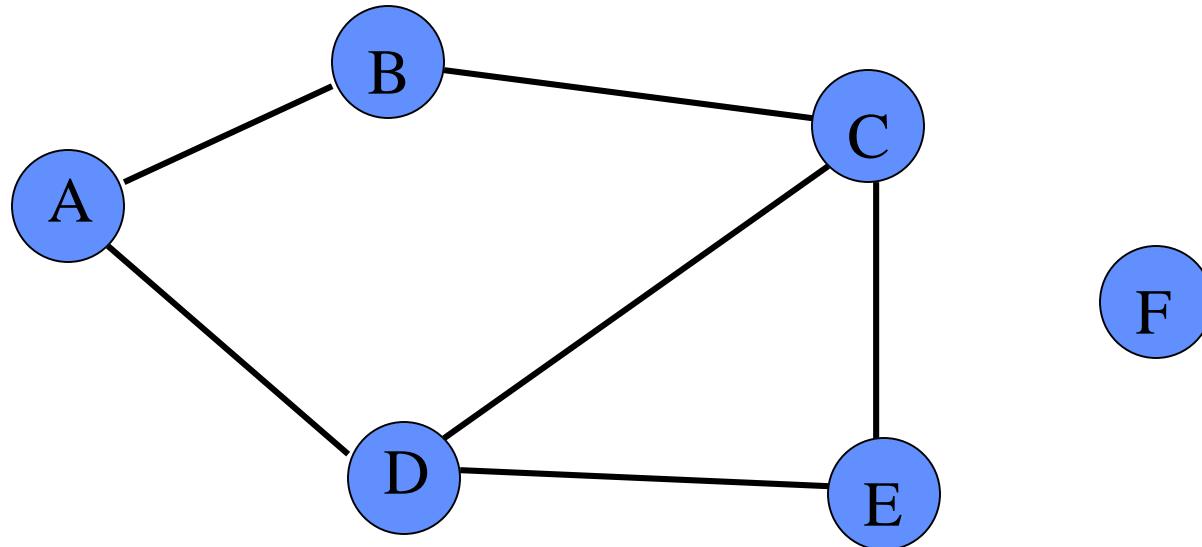
Quando tra due vertici **non esiste** un arco, si dice che il costo è **infinito**.



Definizioni sui grafi

Sia $G = (V, E)$ un grafo.

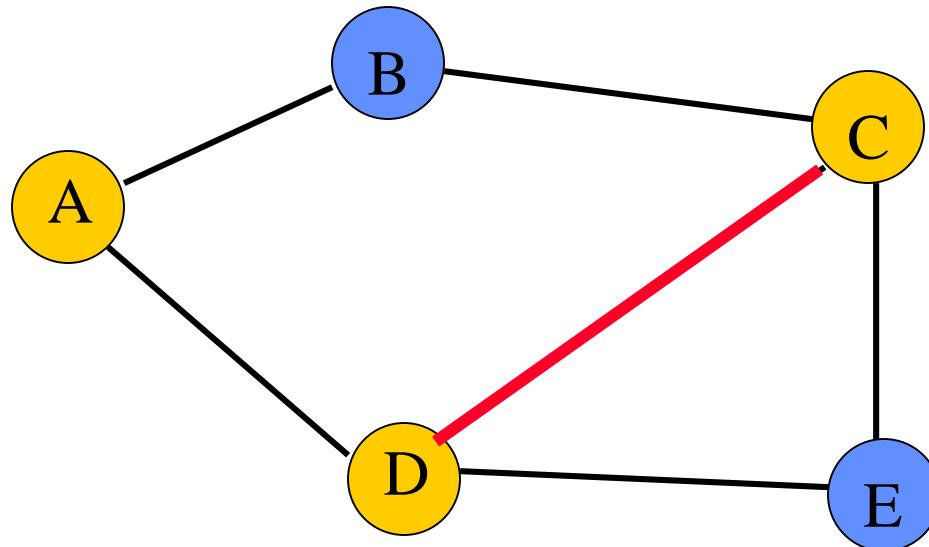
Un *sottografo* di G è un grafo $H = (V^*, E^*)$ tale che $V^* \subseteq V$ e $E^* \subseteq E$. (e poiché H è un grafo, deve valere che $E^* \subseteq V^* \times V^*$.)



Definizioni sui grafi

Sia $G = (V, E)$ un grafo.

Un *sottografo* di G è un grafo $H = (V^*, E^*)$ tale che $V^* \subseteq V$ e $E^* \subseteq E$. (e poiché H è un grafo, deve valere che $E^* \subseteq V^* \times V^*$.)



Es.,

$$V^* = \{A, C, D\},$$

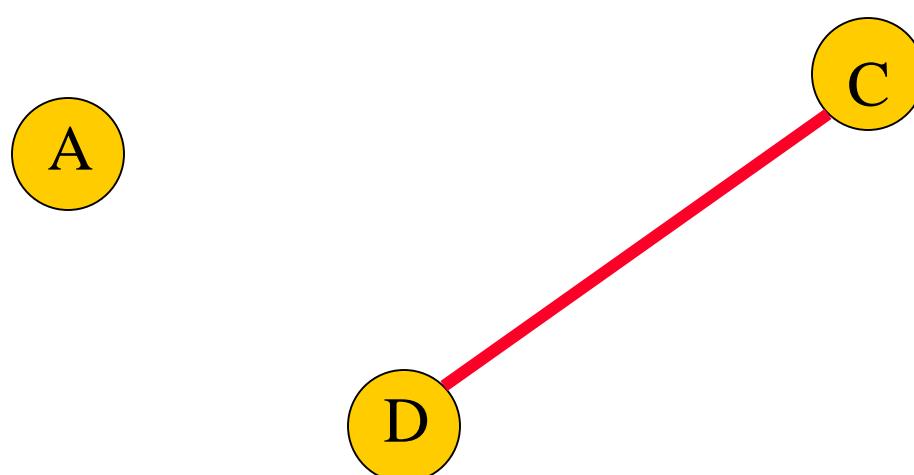
$$E^* = \{(C, D)\}.$$



Definizioni sui grafi

Sia $G = (V, E)$ un grafo.

Un *sottografo* di G è un grafo $H = (V^*, E^*)$ tale che $V^* \subseteq V$ e $E^* \subseteq E$. (e poiché H è un grafo, deve valere che $E^* \subseteq V^* \times V^*$.)



Es.,

$$V^* = \{A, C, D\},$$

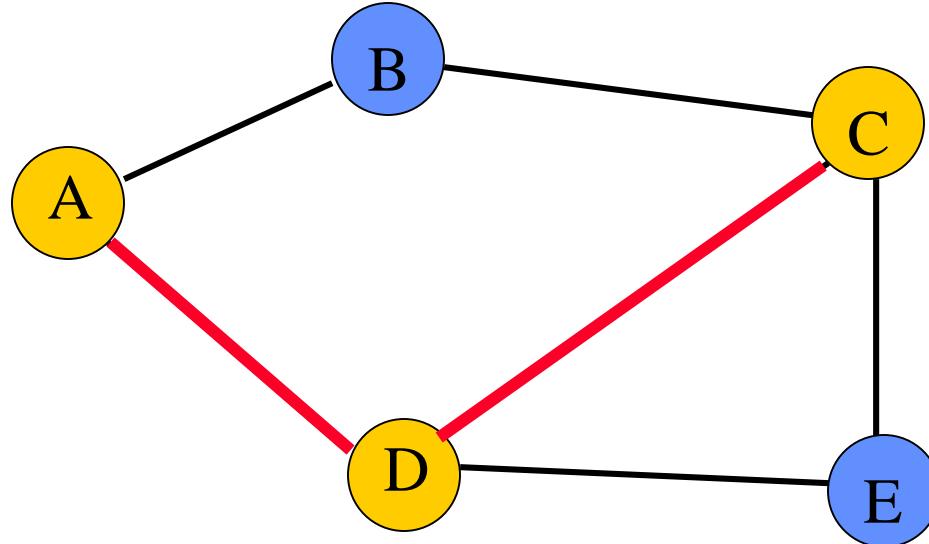
$$E^* = \{(C, D)\}.$$

Definizioni sui grafi

Sia $G = (V, E)$ un grafo e $V^* \subseteq V$ un insieme di vertici.

Il **sottografo** di G **indotto** da V^* è il grafo $H=(V^*, E^*)$ tale che:

$$E^* = \{(w, v) \in E / w, v \in V^*\} = E^* = E \cap (V^* \times V^*)$$



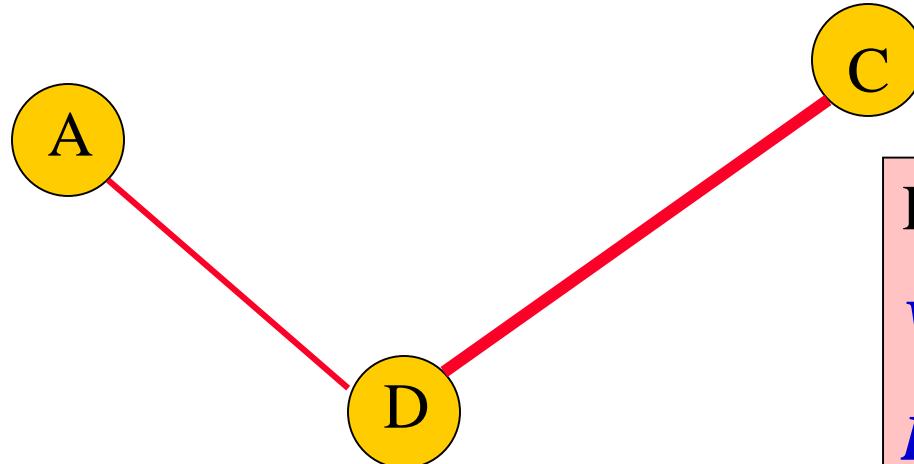
Es.,
 $V^* = \{A, C, D\}$

Definizioni sui grafi

Sia $G = (V, E)$ un grafo e $V^* \subseteq V$ un insieme di vertici.

Il *sottografo* di G *indotto* da V^* è il grafo $H=(V^*, E^*)$ tale che:

$$E^* = E \cap (V^* \times V^*)$$



Es.,

$$V^* = \{A, C, D\},$$

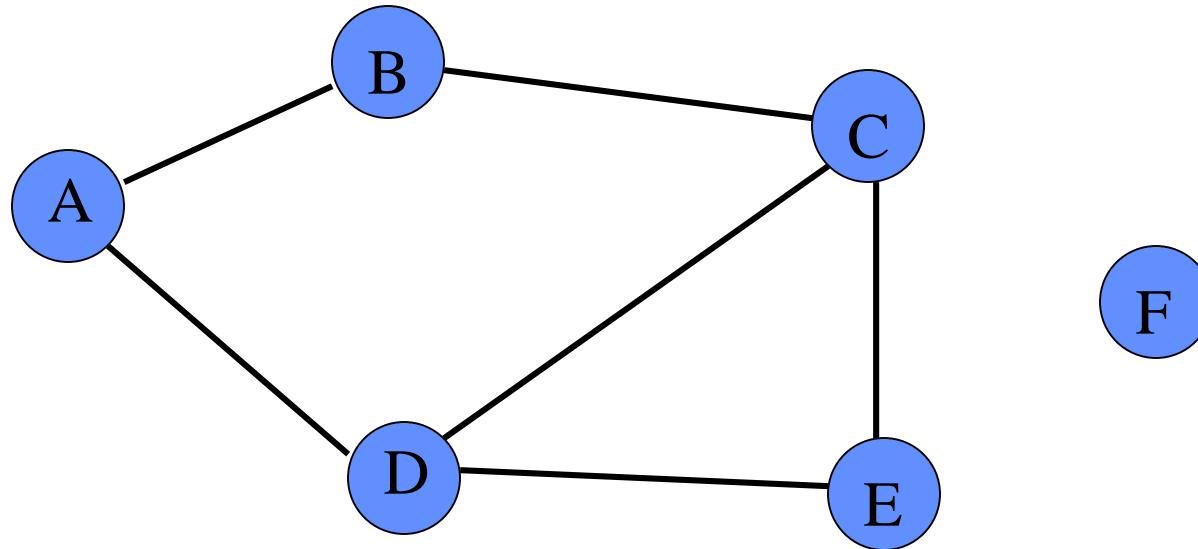
$$E^* = \{(C, D), (A, D)\}.$$

Definizioni sui grafi

Sia $G = (V, E)$ un grafo.

Un *sottografo* $H=(V^*, E^*)$ di G è detto *di supporto* se:

$$V^* = V$$

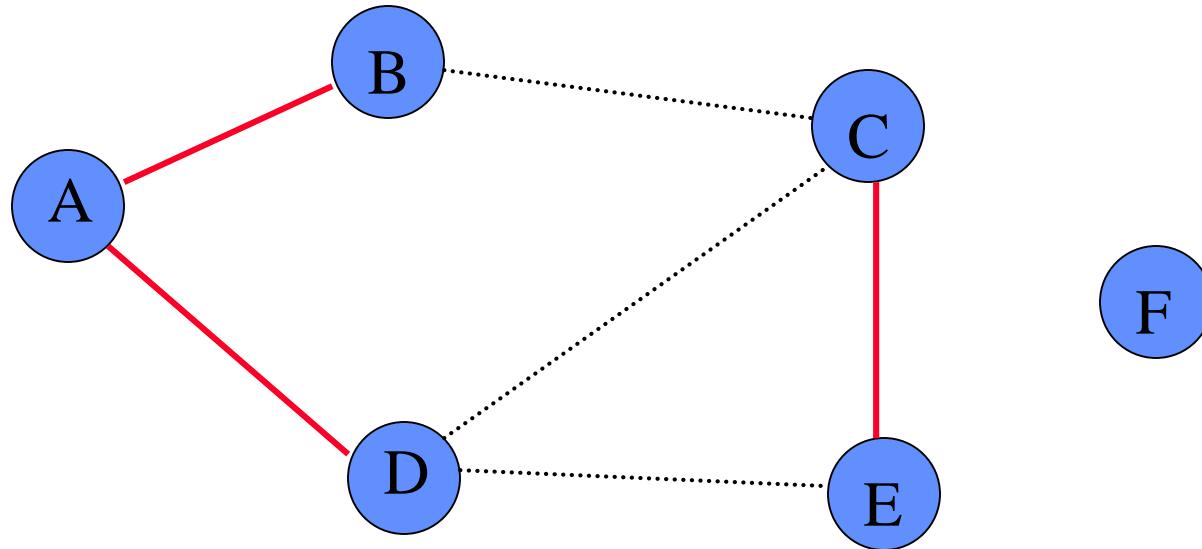


Definizioni sui grafi

Sia $G = (V, E)$ un grafo.

Un *sottografo* $H=(V^*, E^*)$ di G è detto *di supporto* se:

$$V^* = V$$

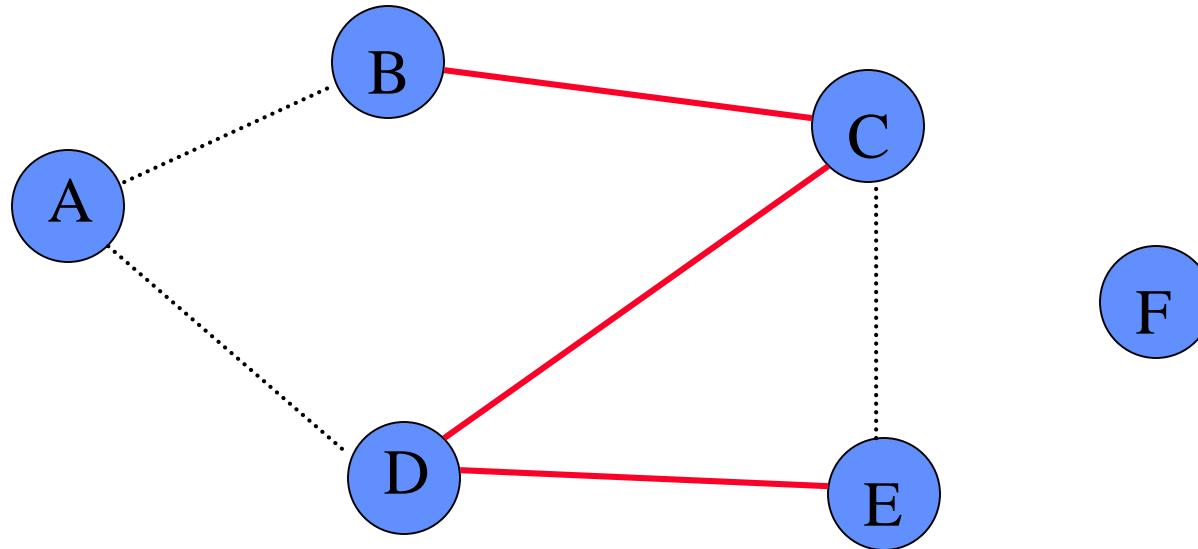


Definizioni sui grafi

Sia $G = (V, E)$ un grafo.

Un *sottografo* $H=(V^*, E^*)$ di G è detto *di supporto* se:

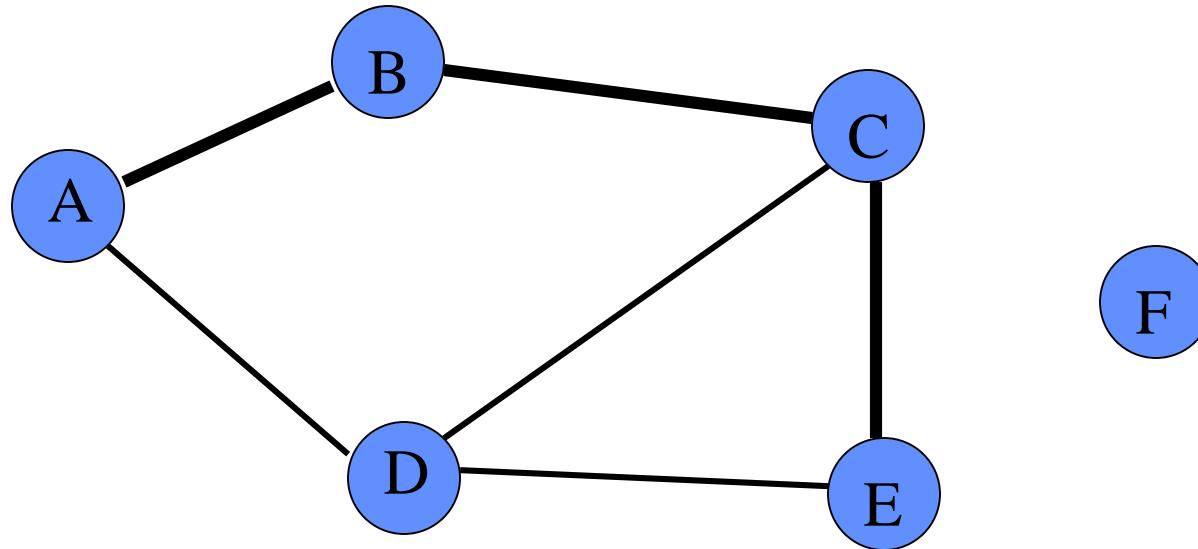
$$V^* = V$$



Definizioni sui grafi

Sia $G = (V, E)$ un grafo.

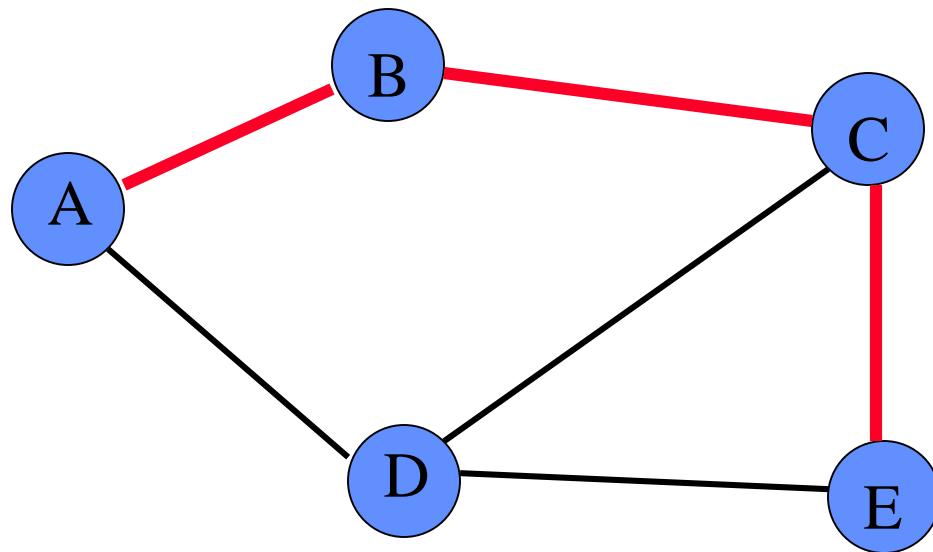
Un *percorso* nel grafo è una sequenza di vertici $\langle w_1, w_2, \dots, w_n \rangle$ tale che $(w_i, w_{i+1}) \in E$ per $1 \leq i \leq n-1$.



Definizioni sui grafi

Sia $G = (V, E)$ un grafo.

Un *percorso* nel grafo è una sequenza di vertici $\langle w_1, w_2, \dots, w_n \rangle$ tale che $(w_i, w_{i+1}) \in E$ per $1 \leq i \leq n-1$.



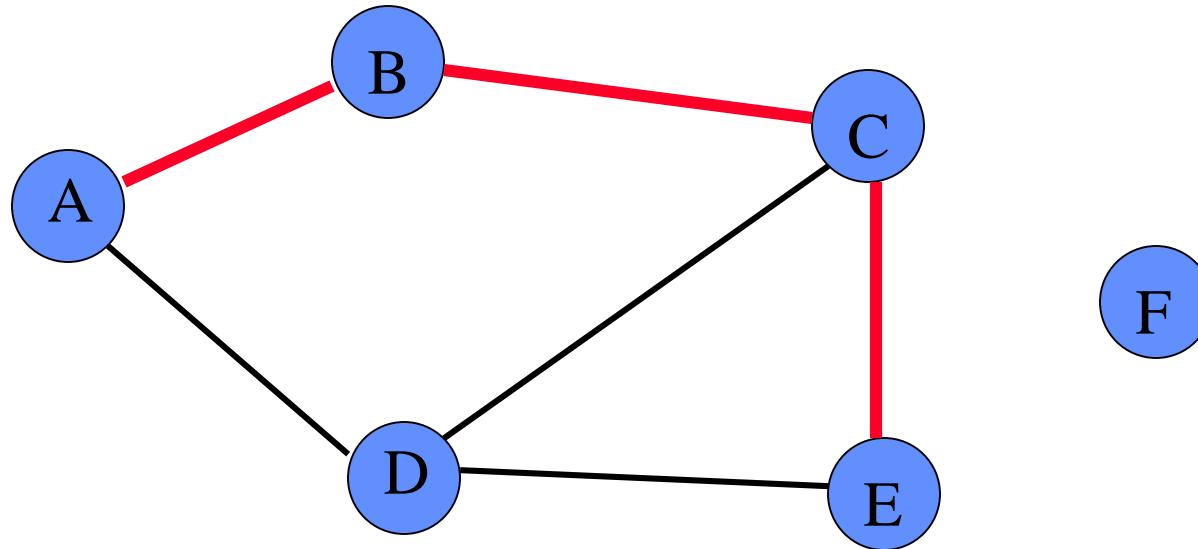
Es.,
 $\langle A, B, C, E \rangle$
è un percorso nel
grafo

Definizioni sui grafi

Sia $G = (V, E)$ un grafo.

Un *percorso* nel grafo è una sequenza di vertici $\langle w_1, w_2, \dots, w_n \rangle$ tale che $(w_i, w_{i+1}) \in E$ per $1 \leq i \leq n-1$.

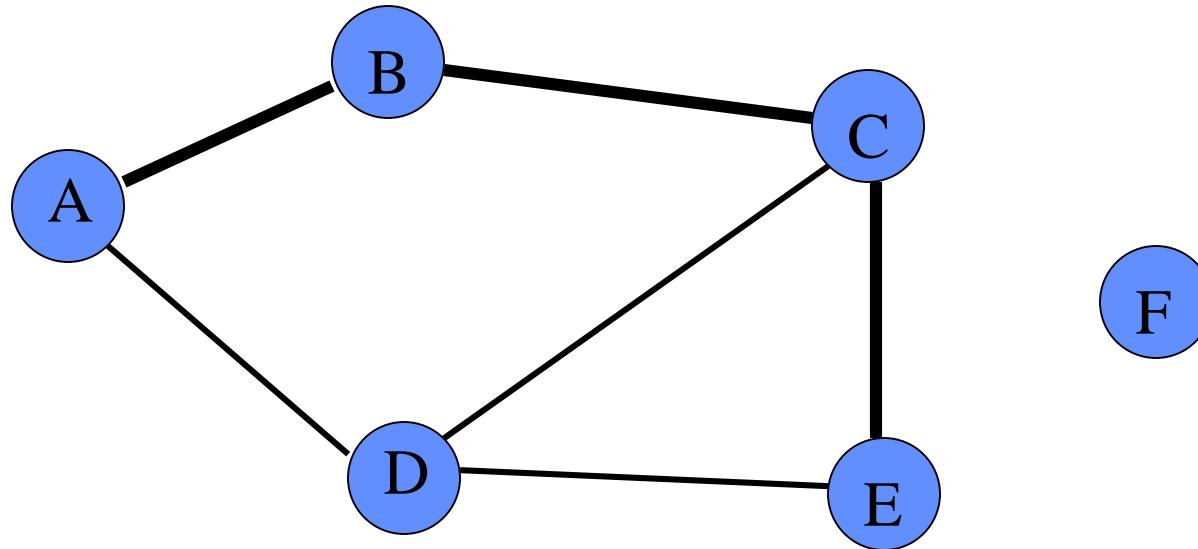
Il *percorso* $\langle w_1, w_2, \dots, w_n \rangle$ si dice che *contiene* i vertici w_1, w_2, \dots, w_n e gli archi $(w_1, w_2), (w_2, w_3), \dots, (w_{n-1}, w_n)$



Definizioni sui grafi

Sia $\langle w_1, w_2, \dots, w_n \rangle$ un *percorso*.

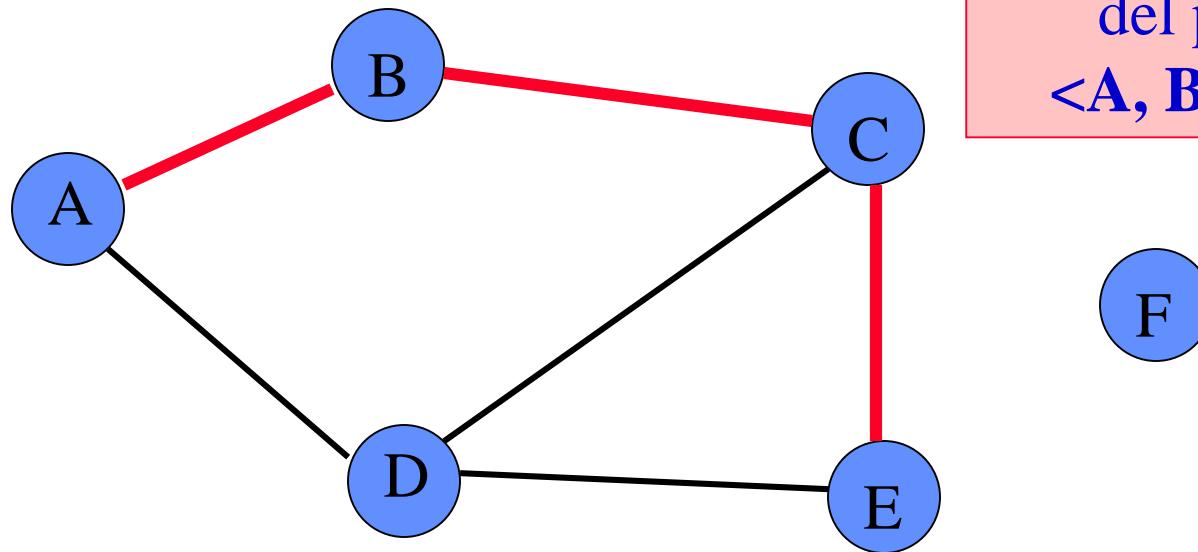
La *lunghezza* del percorso è il *numero totale di archi* che connettono i vertici nell'ordine della sequenza (se il numero di vertici nella sequenza è ***n***, il numero di archi sarà ***n-1***).



Definizioni sui grafi

Sia $\langle w_1, w_2, \dots, w_n \rangle$ un *percorso*.

La *lunghezza* del percorso è il *numero totale di archi* che connettono i vertici nell'ordine della sequenza (se il numero di vertici nella sequenza è ***n***, il numero di archi sarà ***n-1***).

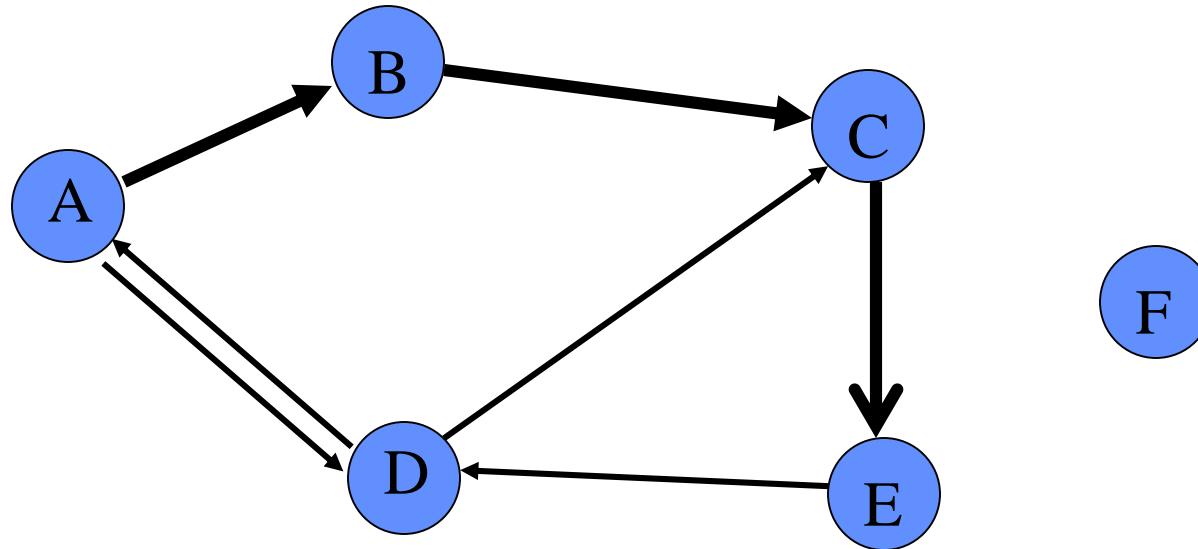


Es., la lunghezza
del percorso
 $\langle A, B, C, E \rangle$ è 3.

Definizioni sui grafi

Sia $\langle w_1, w_2, \dots, w_n \rangle$ un **percorso** in un **grafo orientato**.

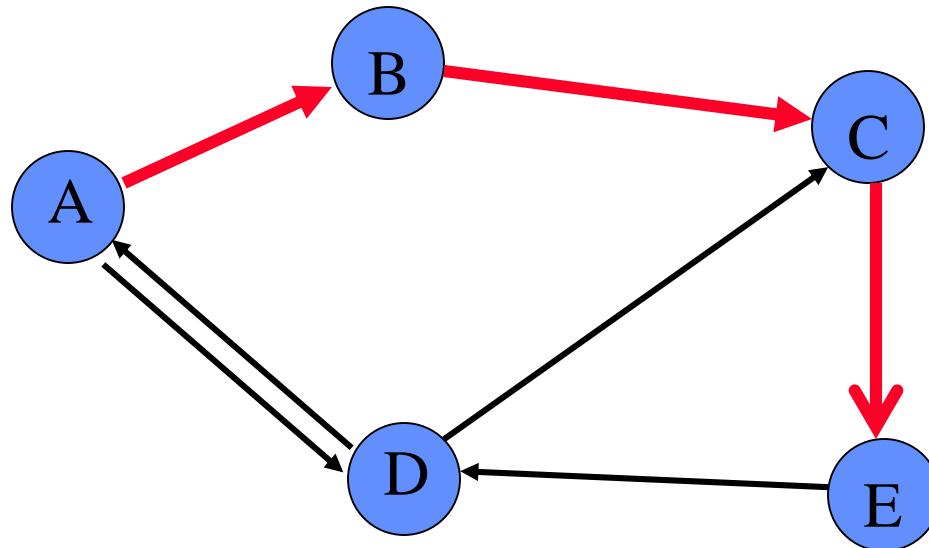
Poiché **ogni arco** (w_i, w_{i+1}) nel percorso è una **coppia ordinata di vertici**, gli **archi** del percorso sono sempre **orientati lungo il percorso**.



Definizioni sui grafi

Sia $\langle w_1, w_2, \dots, w_n \rangle$ un **percorso** in un **grafo orientato**.

Poiché **ogni arco** (w_i, w_{i+1}) nel percorso è una **coppia ordinata di vertici**, gli **archi** del percorso sono sempre **orientati lungo il percorso**.

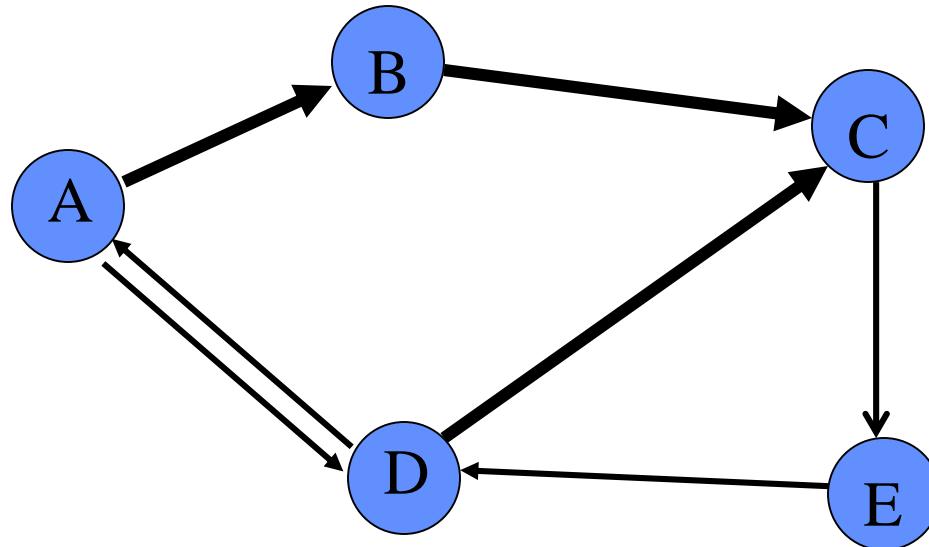


Es., $\langle A, B, C, E \rangle$ è un percorso in questo grafo orientato, ma ...

Definizioni sui grafi

Sia $\langle w_1, w_2, \dots, w_n \rangle$ un **percorso** in un **grafo orientato**.

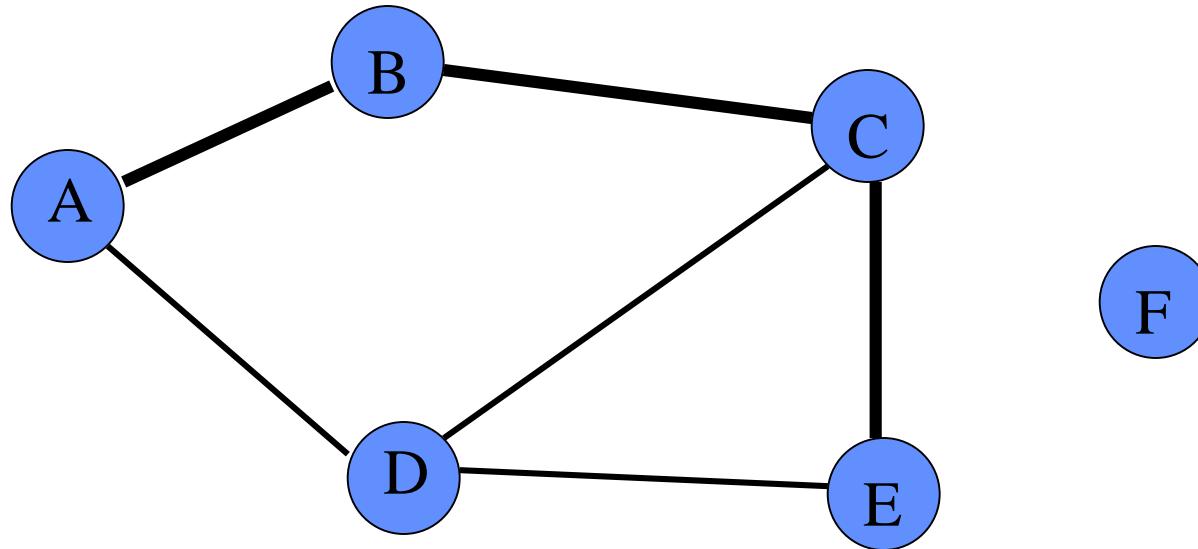
Poiché **ogni arco** (w_i, w_{i+1}) nel percorso è una **coppia ordinata di vertici**, gli **archi** del percorso sono sempre **orientati lungo il percorso**.



... ma $\langle A, B, C, D \rangle$ **non è** un percorso,
poiché (C, D) non è
un arco.

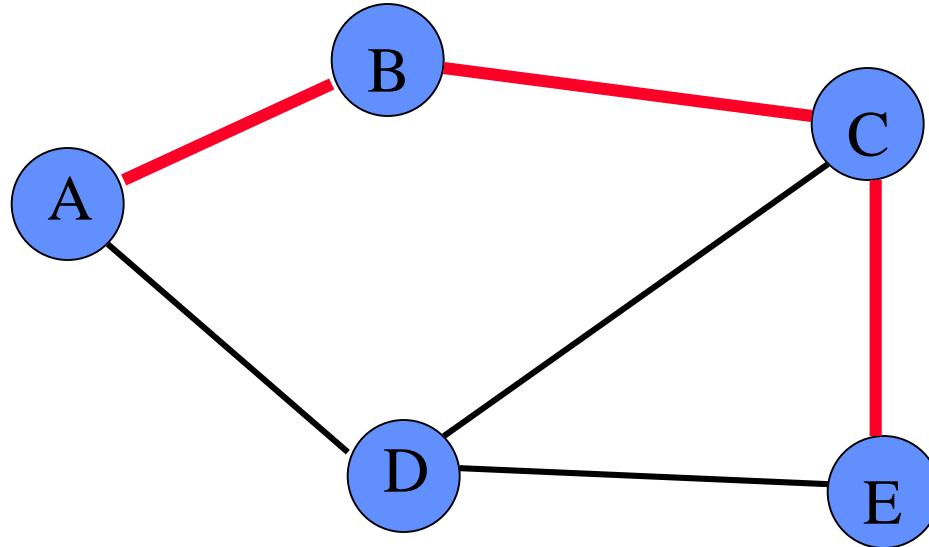
Definizioni sui grafi

Un percorso si dice *semplice* se tutti i *suoi vertici sono distinti* (compaiono una sola volta nella sequenza), *eccetto* al più il primo e l'ultimo che possono coincidere.



Definizioni sui grafi

Un percorso si dice *semplice* se tutti i *suoi vertici sono distinti* (compaiono una sola volta nella sequenza), *eccetto* al più il primo e l'ultimo che possono coincidere.

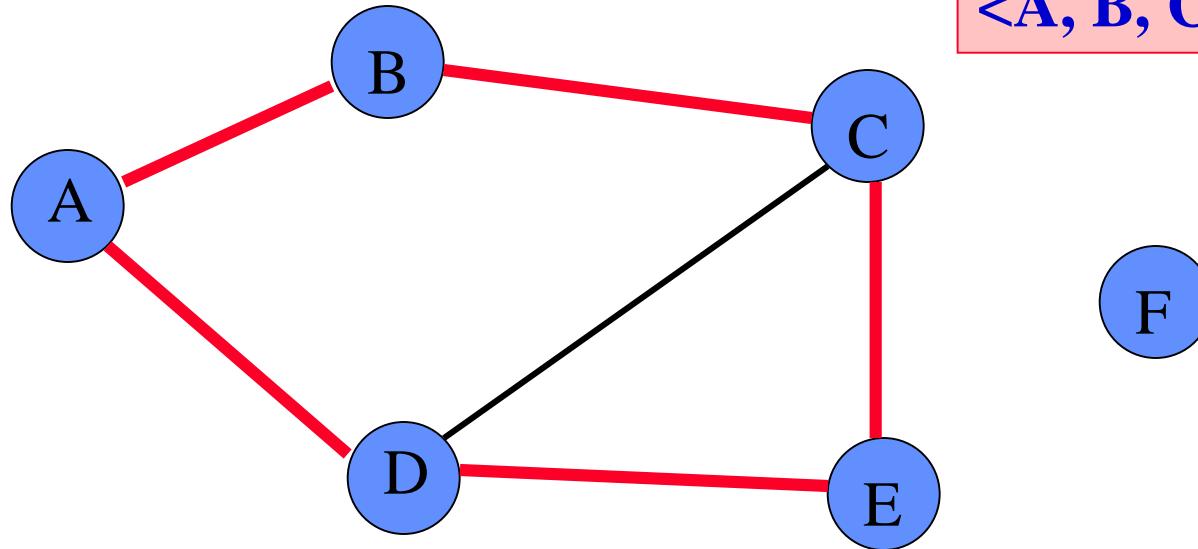


Es., il percorso
 $\langle A, B, C, E \rangle$
è *semplice* ...



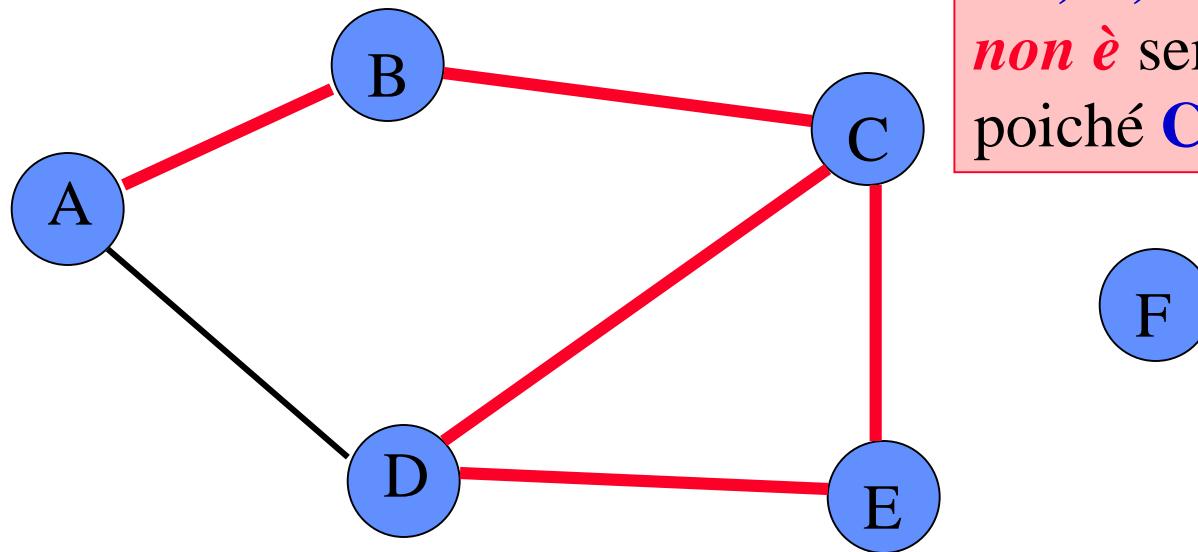
Definizioni sui grafi

Un percorso si dice *semplice* se tutti i *suoi vertici sono distinti* (compaiono una sola volta nella sequenza), *eccetto* al più il primo e l'ultimo che possono coincidere.



Definizioni sui grafi

Un percorso si dice *semplice* se tutti i *suoi vertici sono distinti* (compaiono una sola volta nella sequenza), *eccetto* al più il primo e l'ultimo che possono coincidere.



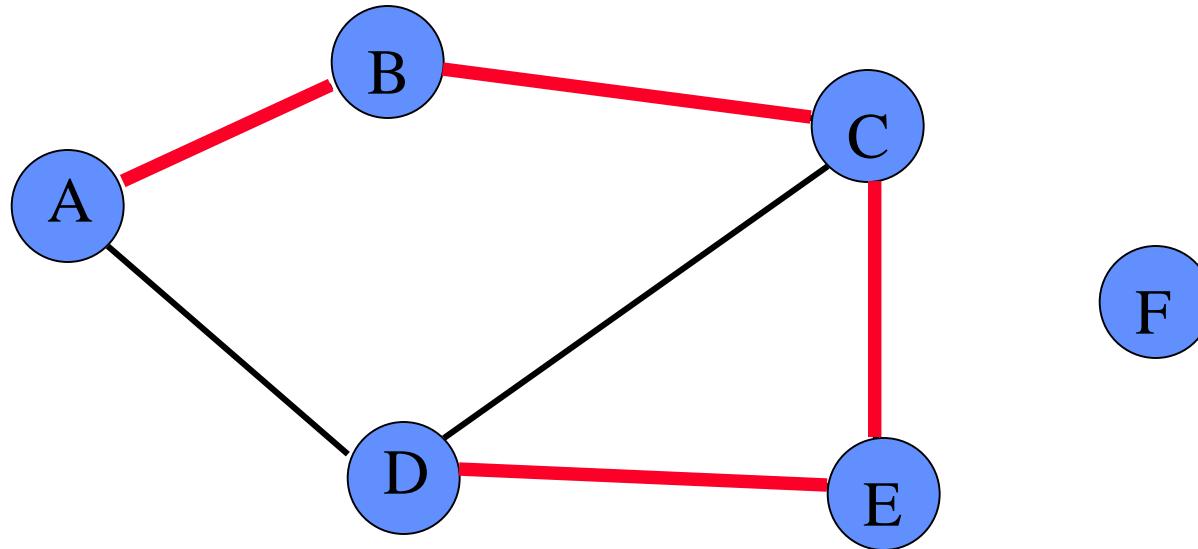
... ma il percorso
<A, B, C, E, D, C >
non è semplice,
poiché **C** è ripetuto.

Definizioni sui grafi

Se esiste un percorso p tra i vertici v e w , si dice che w è raggiungibile da v tramite p

$$v \xrightarrow{p} w$$

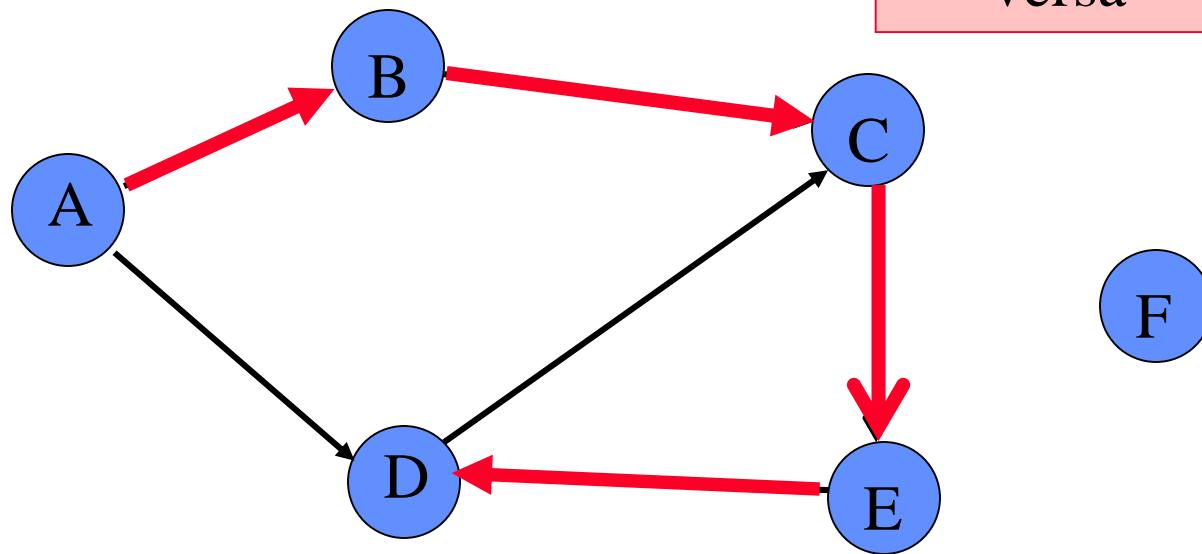
Es.: A è raggiungibile
da D e vice versa



Definizioni sui grafi

Se esiste un percorso p tra i vertici v e w , si dice che w è raggiungibile da v tramite p

$$v \xrightarrow{p} w$$



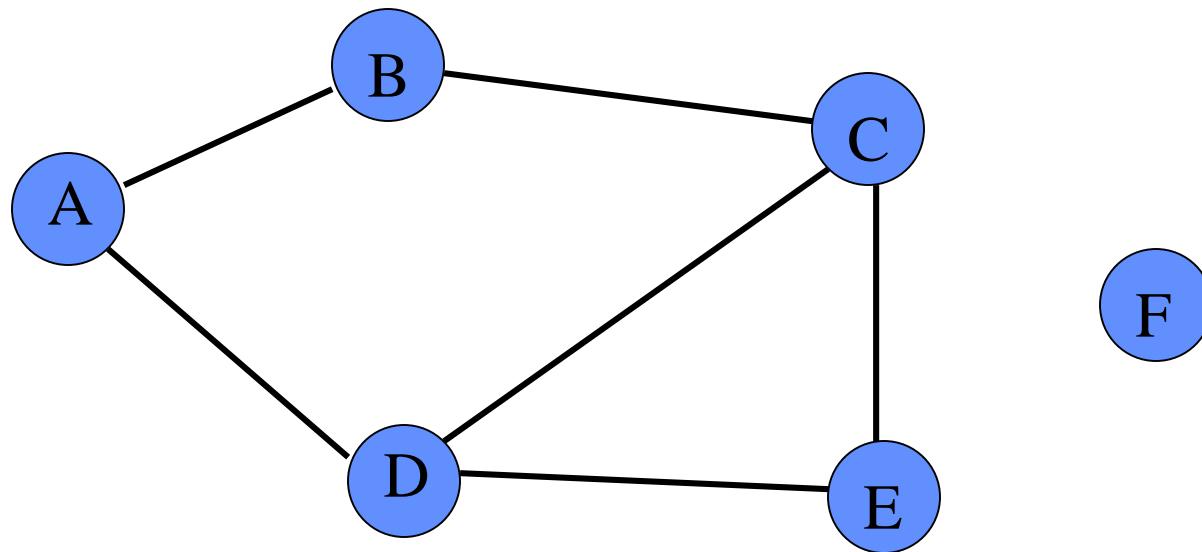
Es.: A è raggiungibile
da D ma non vice
versa

Definizioni sui grafi

Se G è un *grafo non orientato*, diciamo che G è *connesso* se esiste un *percorso* da *ogni vertice* ad *ogni altro vertice*.

Un grafo non orientato *non connesso* si dice *sconnesso*.

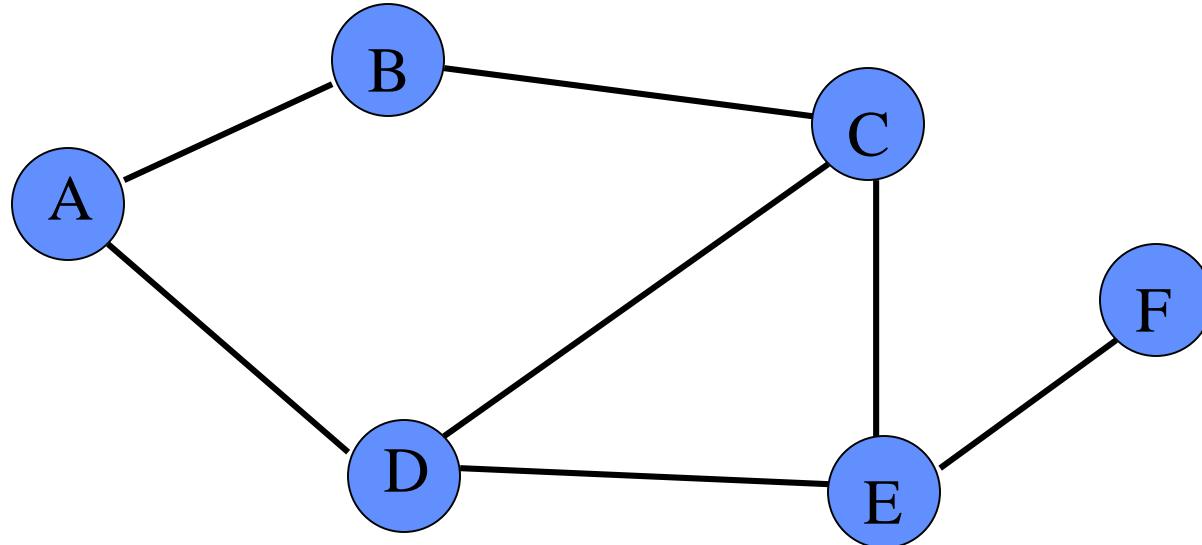
Questo grafo non orientato *non è connesso*.



Definizioni sui grafi

Se G è un *grafo non orientato*, diciamo che G è *connesso* se esiste un *percorso* da *ogni vertice* ad *ogni altro vertice*.

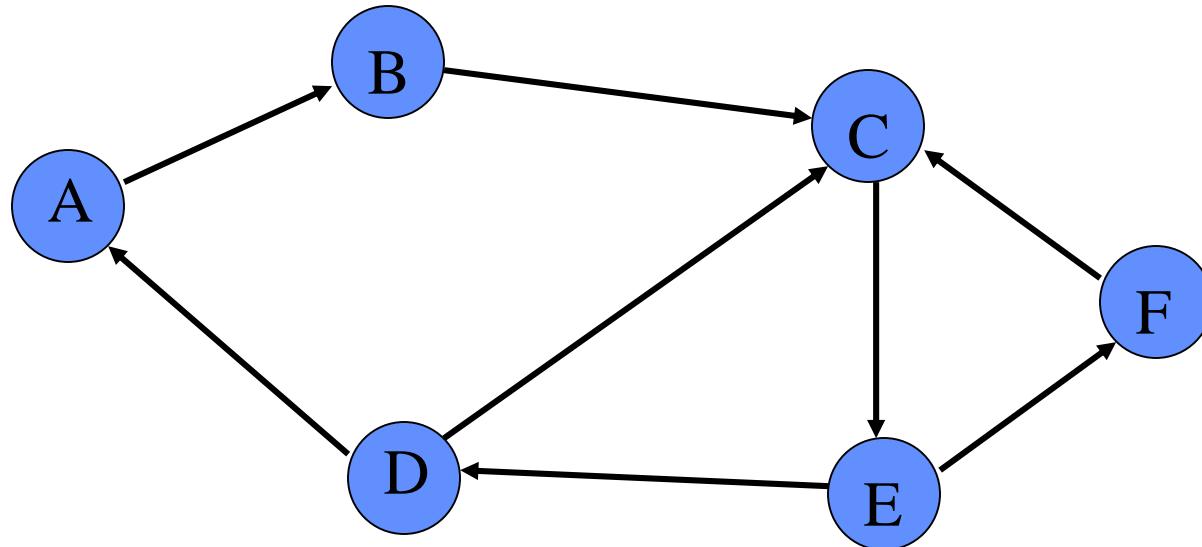
Questo *è connesso*.



Definizioni sui grafi

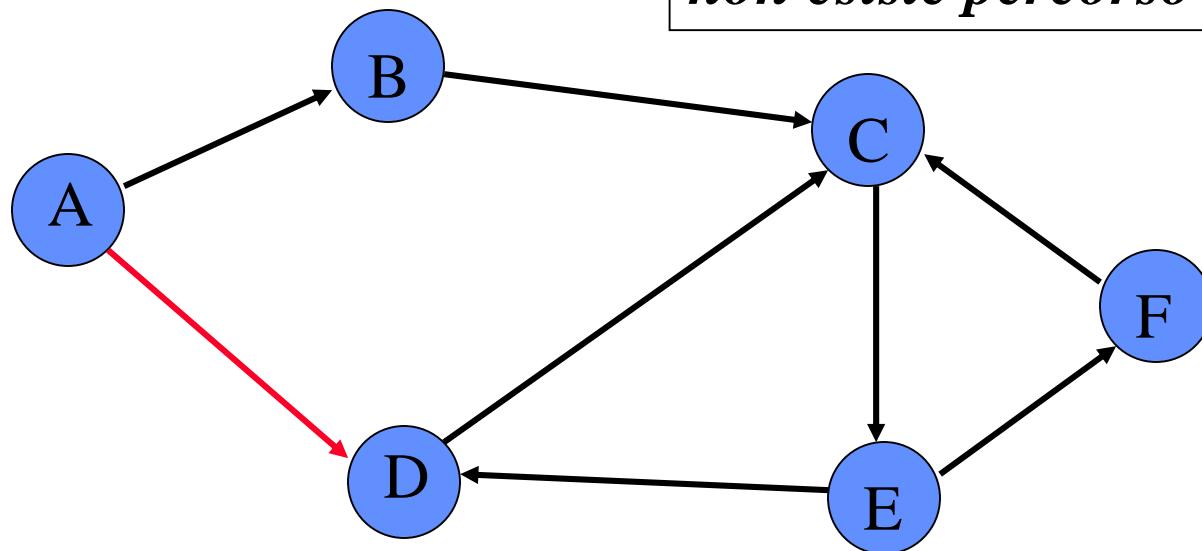
Se G è un *grafo orientato*, diciamo che G è **fortemente connesso** se esiste un *percorso* da *ogni vertice* ad *ogni altro vertice*.

Questo grafo orientato è
fortemente connesso.



Definizioni sui grafi

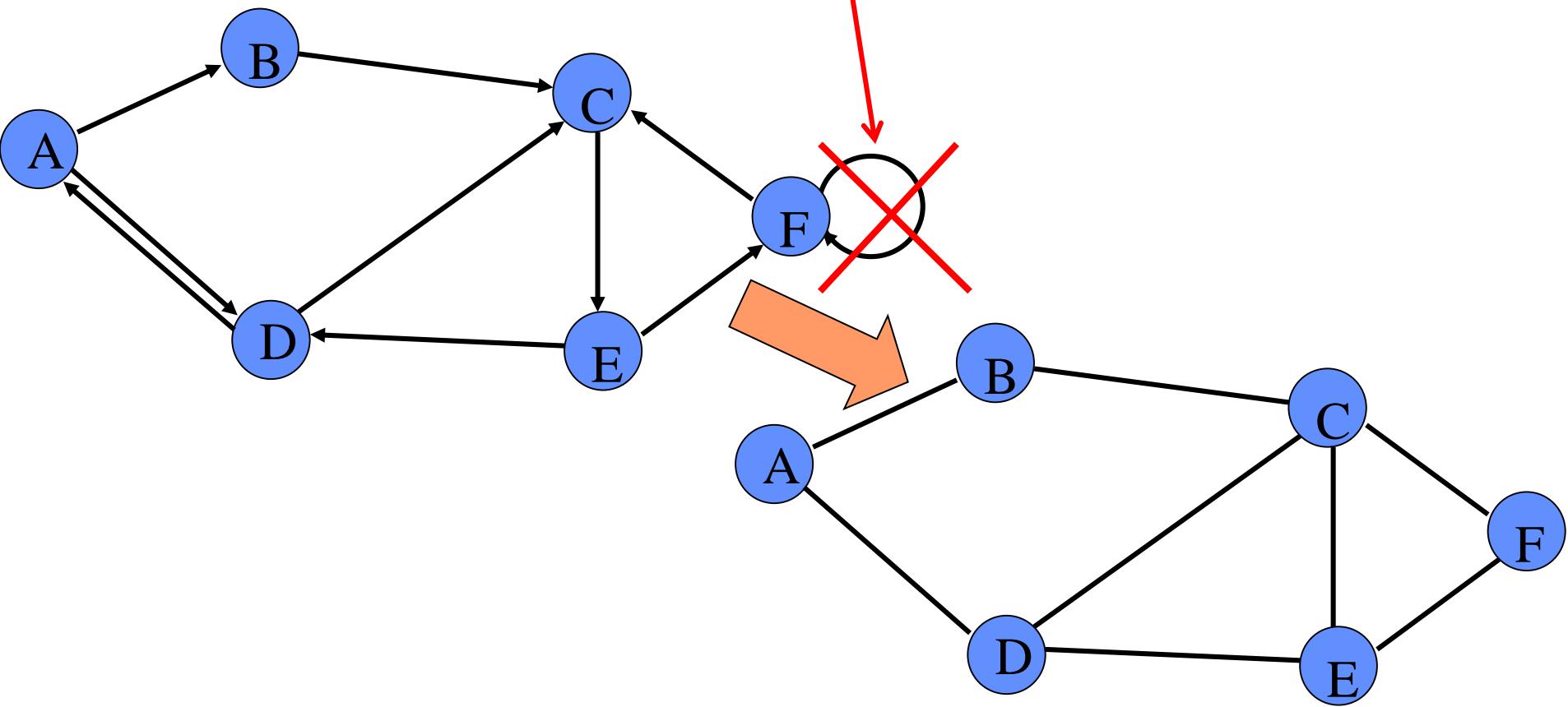
Se G è un *grafo orientato*, diciamo che G è **fortemente connesso** se esiste un *percorso* da *ogni vertice* ad *ogni altro vertice*.



Questo grafo orientato **non è fortemente connesso**; ad es., *non esiste percorso da D a A*.

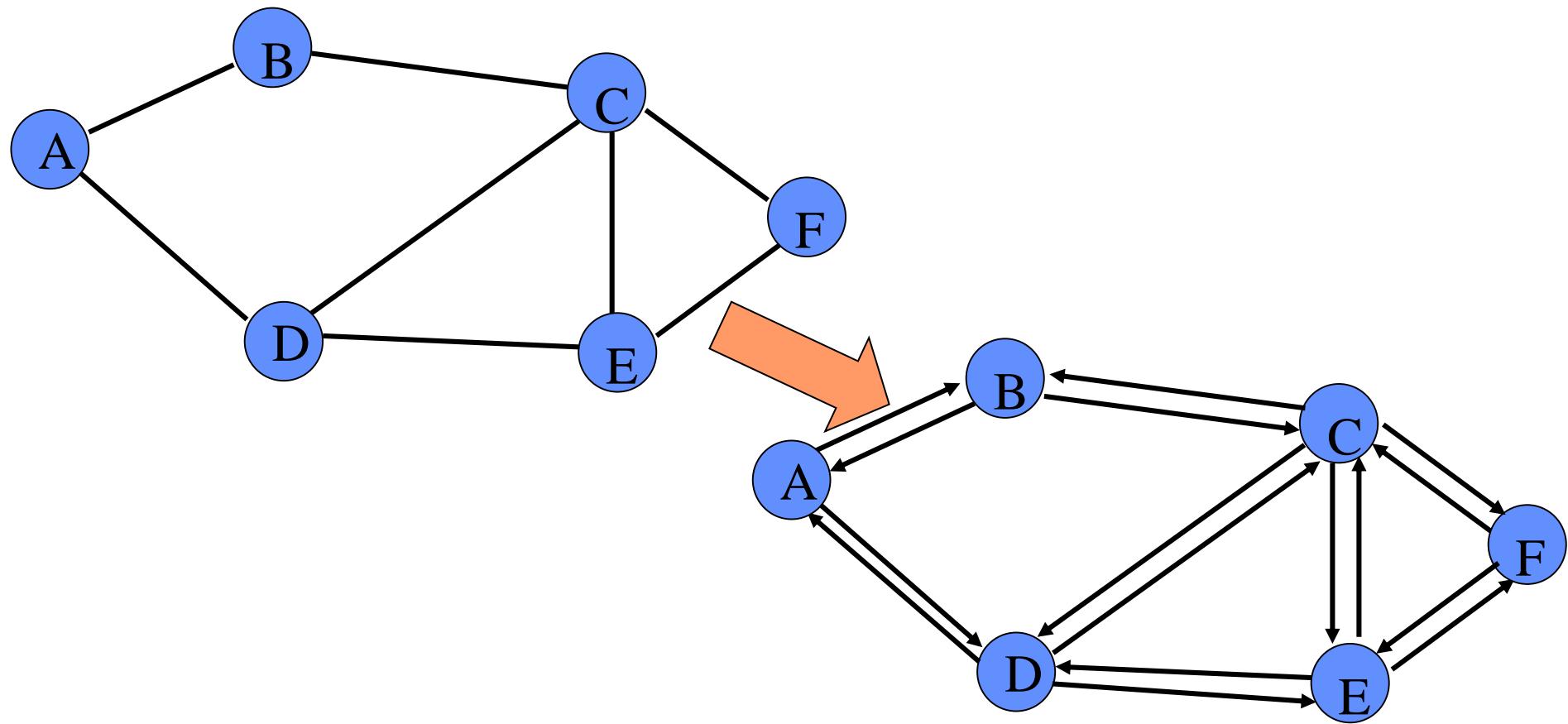
Definizioni sui grafi

Se G è un *grafo orientato*, il grafo ottenuto ignorando la direzione degli archi e i archi ciclici è detto il *grafo non orientato sottostante* o anche *versione non orientata di G* .



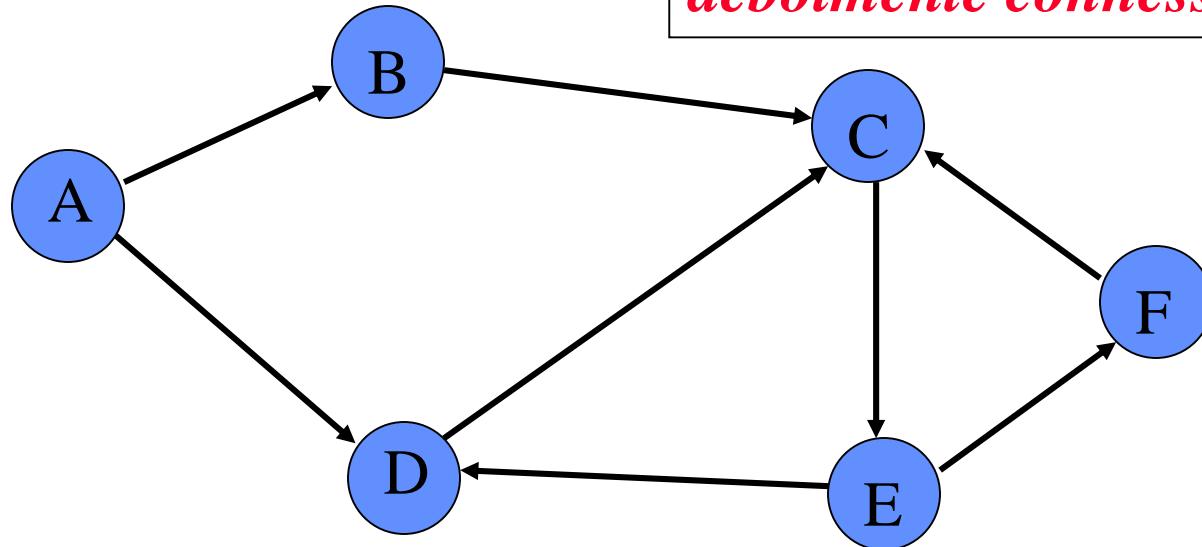
Definizioni sui grafi

Se G è un *grafo non orientato*, il grafo ottenuto inserendo due archi orientati per ogni arco non orientato del grafo è detto il *versione orientata di G* .



Definizioni sui grafi

Se G è un *grafo orientato non fortemente连通的*, ma se il *grafo non orientato sottostante* (cioè senza la direzione degli archi) è *connesso*, diciamo che G è *debolmente连通的*.

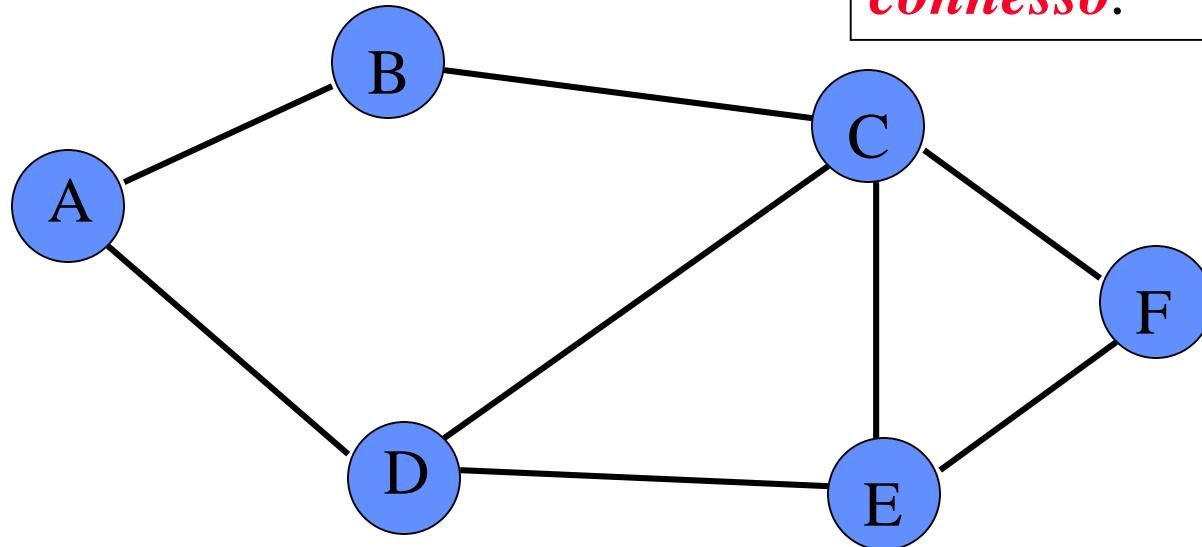


Questo grafo orientato *non è fortemente连通的*, ma *è debolmente连通的*, poiché...

Definizioni sui grafi

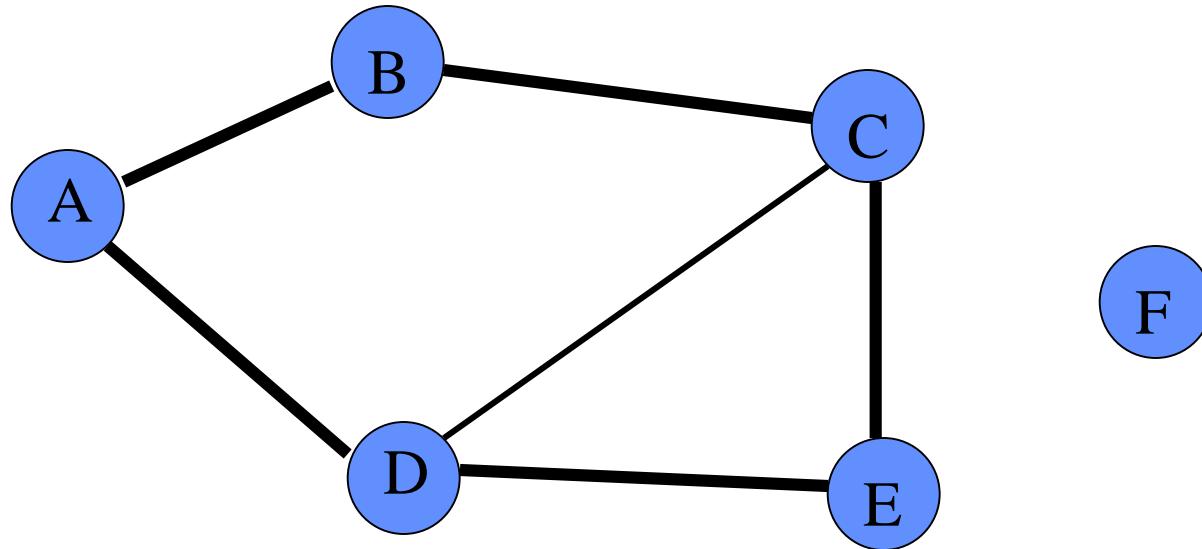
Se G è un *grafo orientato non fortemente connesso*, ma se il *grafo non orientato sottostante* (cioè senza la direzione degli archi) è *connesso*, diciamo che G è *debolmente connesso*.

... poiché il *grafo* non
orientato *sottostante* è
connesso.



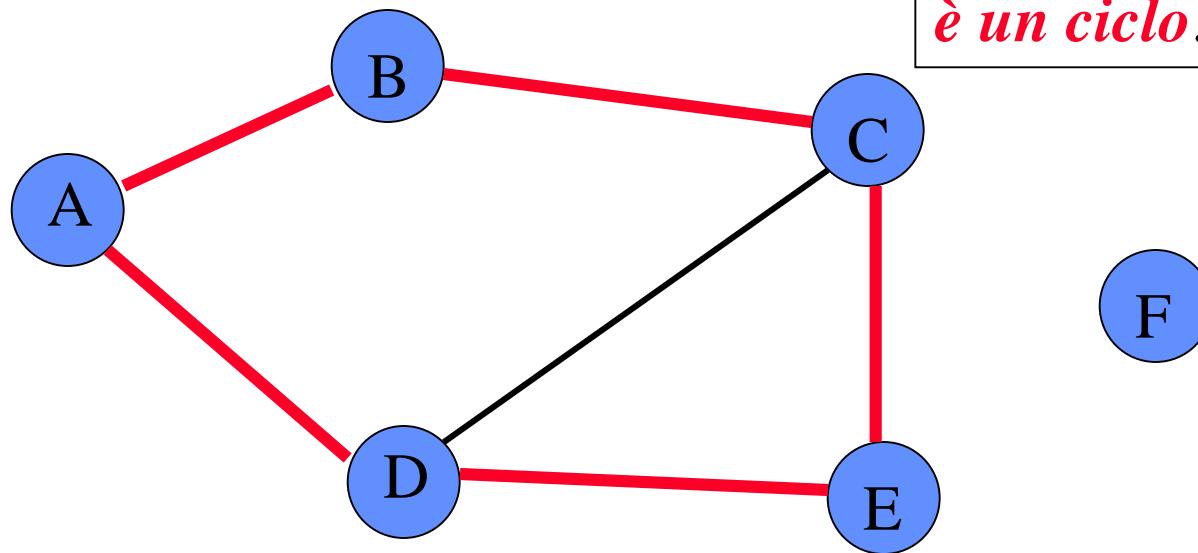
Definizioni sui grafi

Un **ciclo** in un grafo è un percorso $\langle w_1, w_2, \dots, w_n \rangle$ di lunghezza almeno 1, tale che $w_1 = w_n$.



Definizioni sui grafi

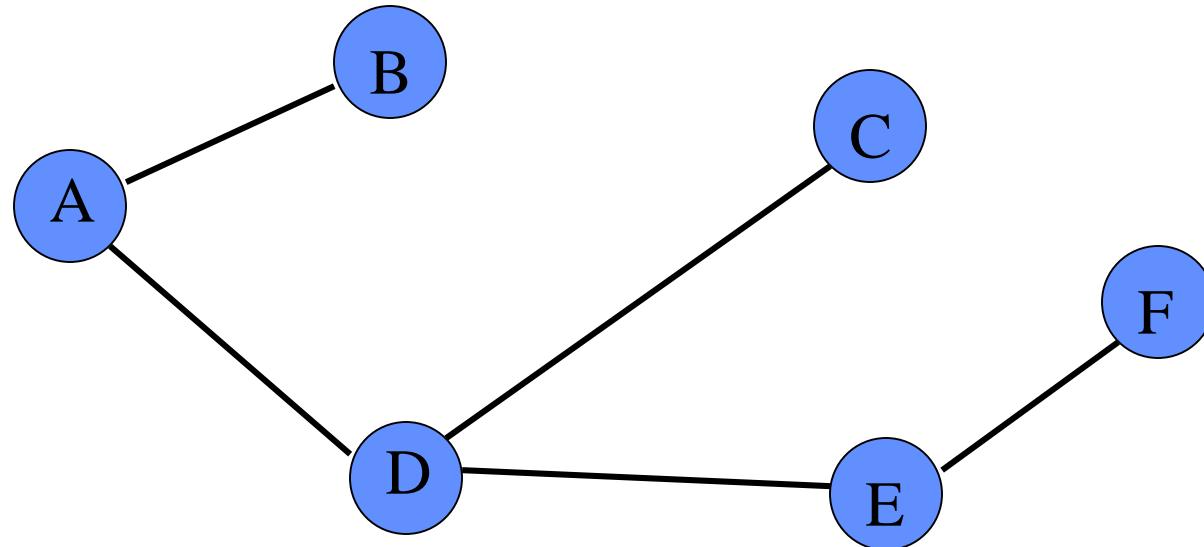
Un **ciclo** in un grafo è un percorso $\langle w_1, w_2, \dots, w_n \rangle$ di lunghezza almeno 1, tale che $w_1 = w_n$.



Es.: il percorso
 $\langle A, B, C, E, D, A \rangle$
è un **ciclo**.

Definizioni sui grafi

Un *grafo senza cicli* è detto *aciclico*.

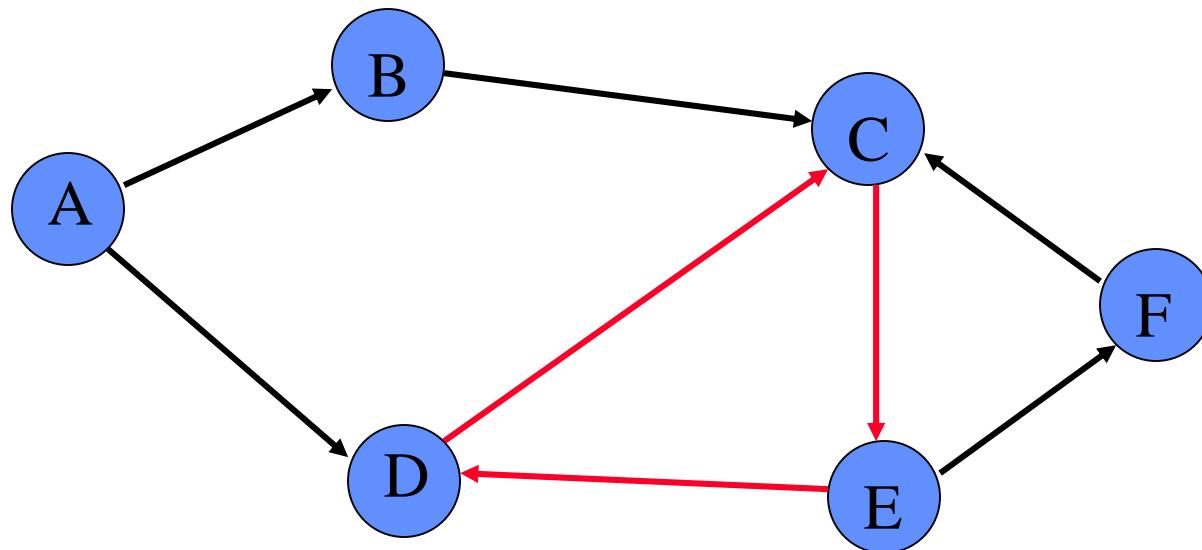


Questo grafo **è**
aciclico.

Definizioni sui grafi

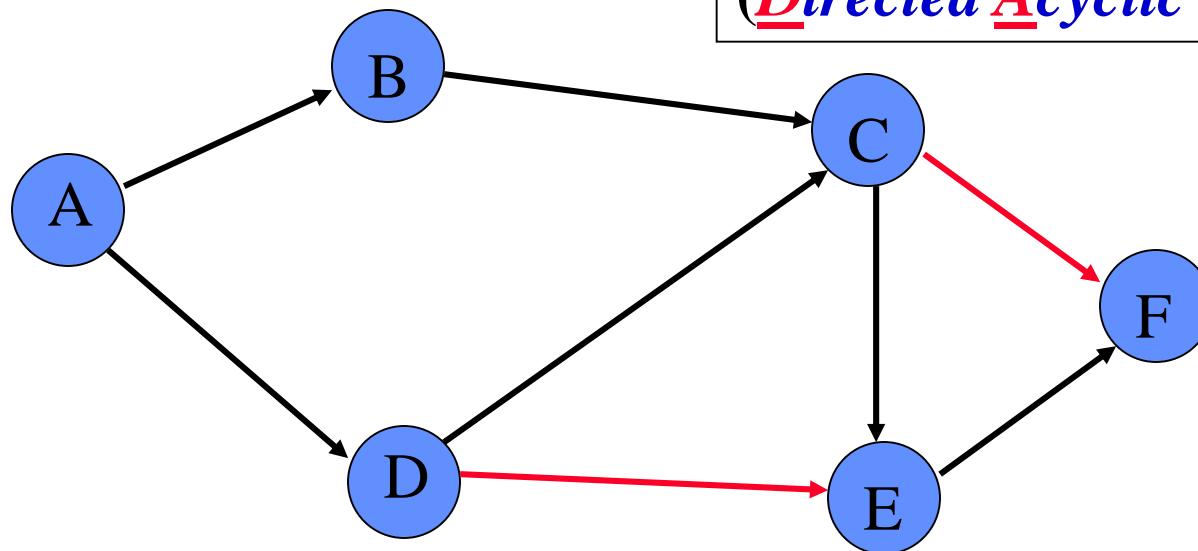
Un *grafo senza cicli* è detto *aciclico*.

Questo grafo orientato *non*
è aciclico, ...



Definizioni sui grafi

Un *grafo senza cicli* è detto *aciclico*.

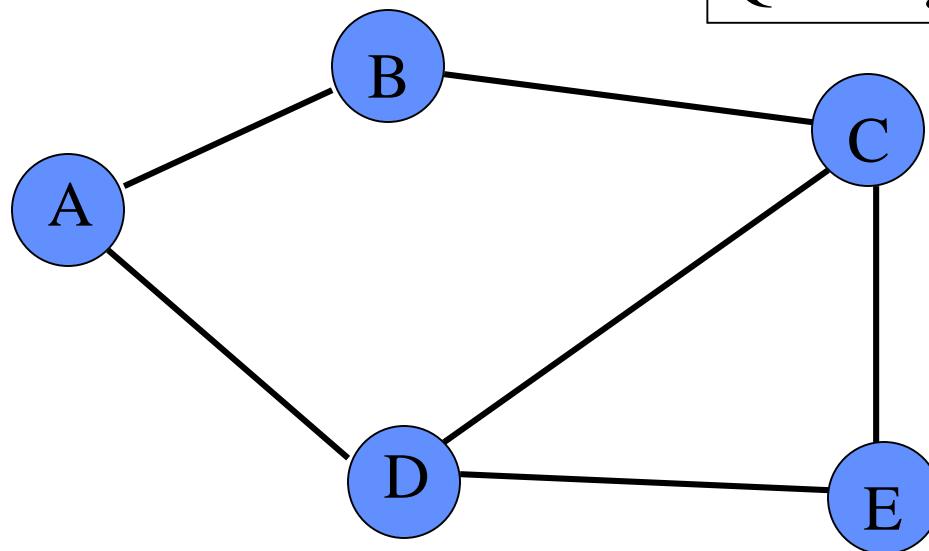


... ma questo lo è.

Un *grafo orientato aciclico*
è spesso chiamato **DAG**
(*Directed Acyclic Graph*).

Definizioni sui grafi

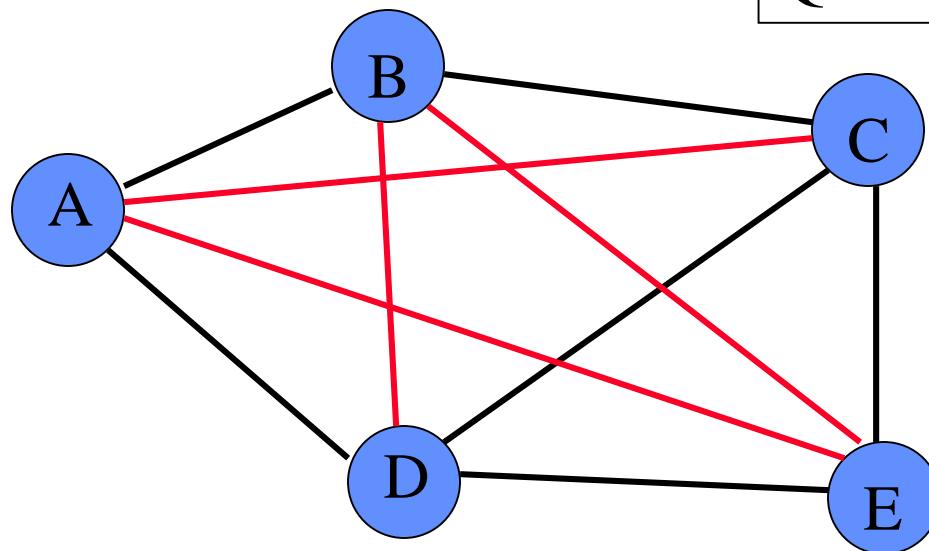
Un *grafo completo* è un grafo che ha un *arco tra ogni coppia di vertici*.



Questo grafo *non è completo*

Definizioni sui grafi

Un **grafo completo** è un grafo che ha un *arco tra ogni coppia di vertici*.



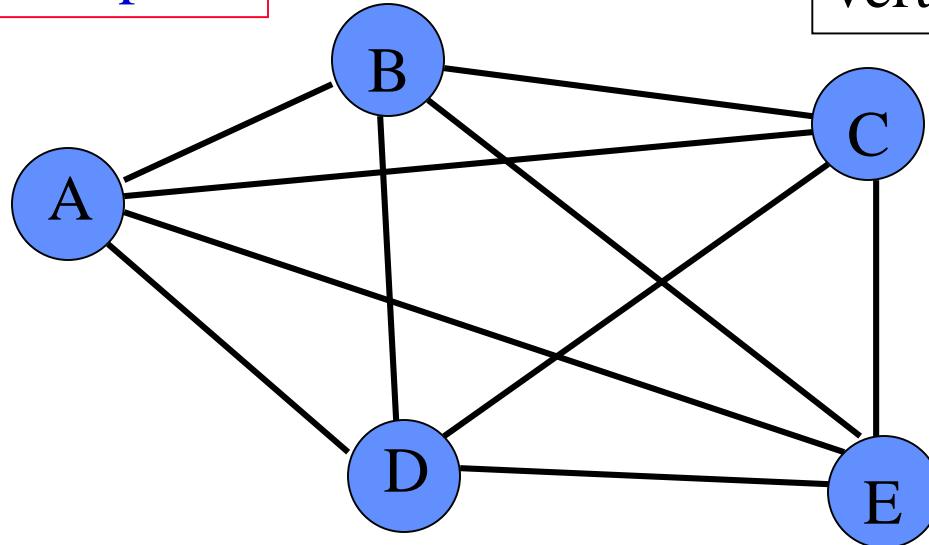
Questo grafo **è completo**

Definizioni sui grafi

Un **grafo completo** è un grafo che ha un **arco tra ogni coppia di vertici**.

Supponiamo che $G = (V, E)$ sia **completo**. In questo caso è possibile esprimere $|E|$ come funzione di $|V|$?

Grafo completo



Questo grafo ha $|V| = 5$ vertici e $|E| = 10$ archi.

Definizioni sui grafi

Un **grafo completo** è un grafo che ha un *arco tra ogni coppia di vertici*.

Supponiamo che $G = (V, E)$ sia **completo**. In questo caso è possibile esprimere $|E|$ come funzione di $|V|$?

Grafo Completo



Usiamo una Tabella:

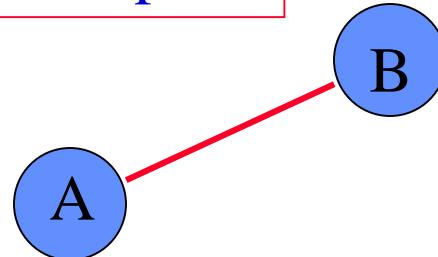
$ V $	$ E $
1	0

Definizioni sui grafi

Un **grafo completo** è un grafo che ha un **arco tra ogni coppia di vertici**.

Supponiamo che $G = (V, E)$ sia **completo**. In questo caso è possibile esprimere $|E|$ come funzione di $|V|$?

Grafo Completo



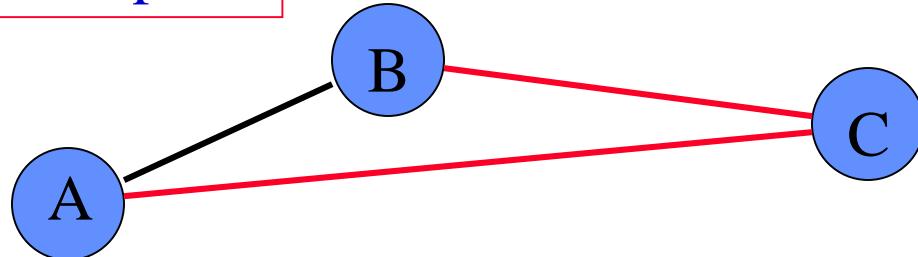
$ V $	$ E $
1	0
2	1

Definizioni sui grafi

Un **grafo completo** è un grafo che ha un **arco tra ogni coppia di vertici**.

Supponiamo che $G = (V, E)$ sia **completo**. In questo caso è possibile esprimere $|E|$ come funzione di $|V|$?

Grafo Completo



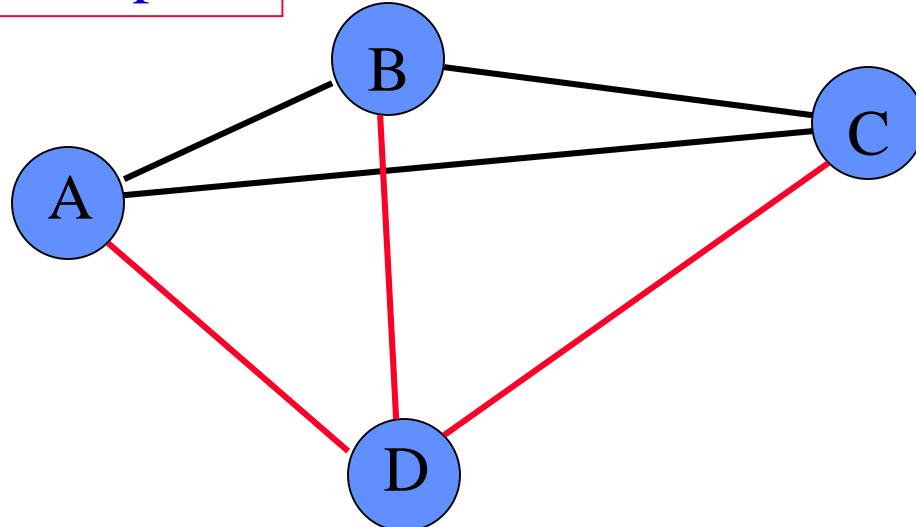
$ V $	$ E $
1	0
2	1
3	3

Definizioni sui grafi

Un **grafo completo** è un grafo che ha un **arco tra ogni coppia di vertici**.

Supponiamo che $G = (V, E)$ sia **completo**. In questo caso è possibile esprimere $|E|$ come funzione di $|V|$?

Grafo Completo



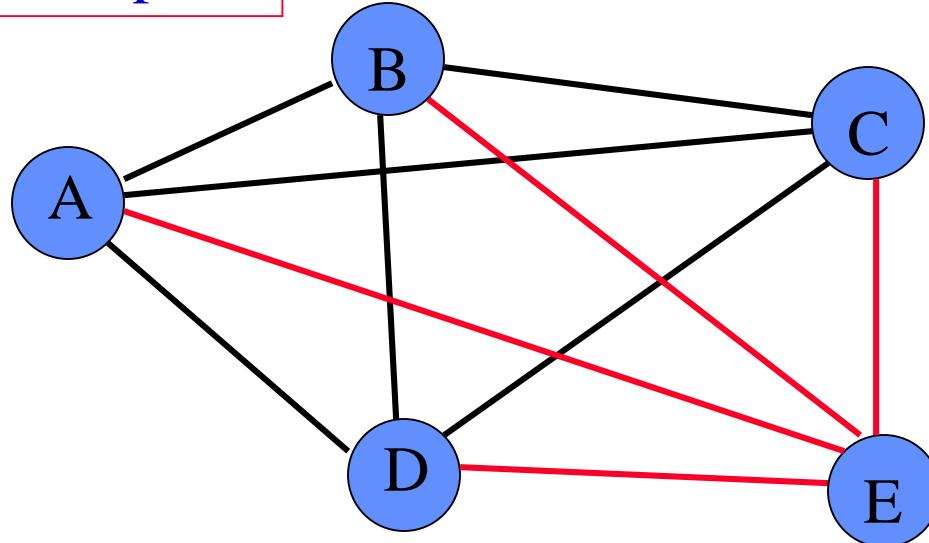
$ V $	$ E $
1	0
2	1
3	3
4	6

Definizioni sui grafi

Un **grafo completo** è un grafo che ha un **arco tra ogni coppia di vertici**.

Supponiamo che $G = (V, E)$ sia **completo**. In questo caso è possibile esprimere $|E|$ come funzione di $|V|$?

Grafo Completo

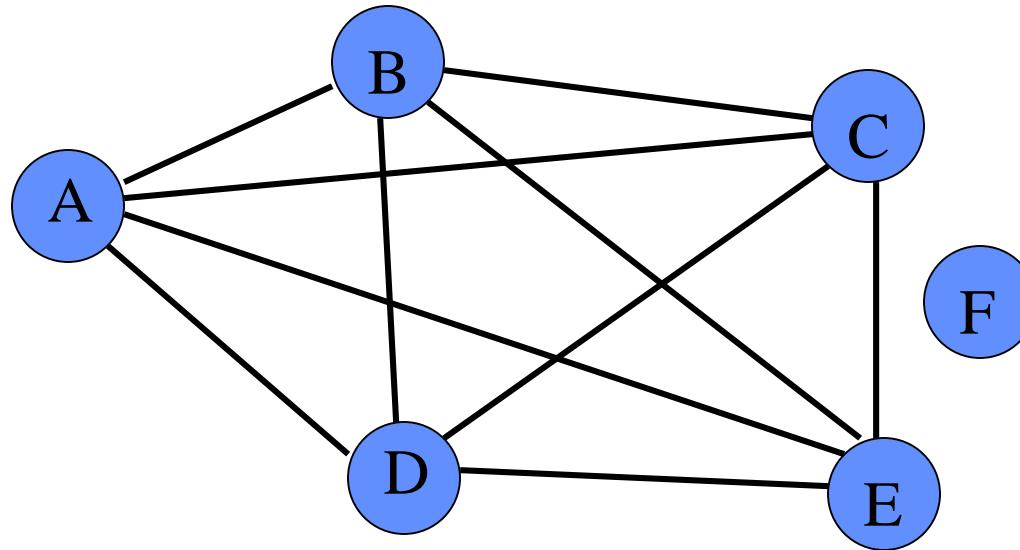


$ V $	$ E $
1	0
2	1
3	3
4	6
5	10

Definizioni sui grafi

Un **grafo completo** è un grafo che ha un **arco tra ogni coppia di vertici**.

Supponiamo che $G = (V, E)$ sia **completo**. In questo caso è possibile esprimere $|E|$ come funzione di $|V|$?

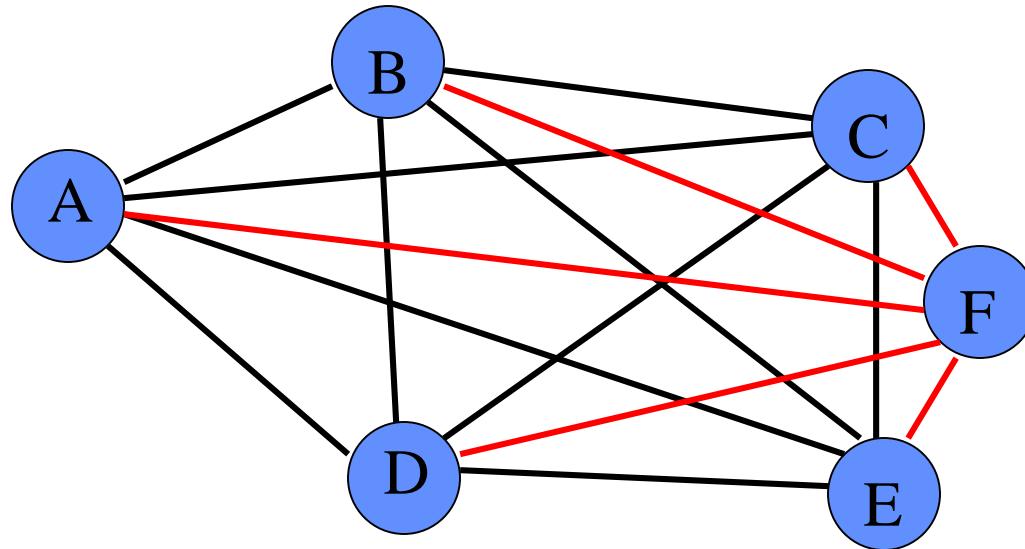


$ V $	$ E $
1	0
2	1
3	3
4	6
5	10
6	?

Definizioni sui grafi

Un **grafo completo** è un grafo che ha un **arco tra ogni coppia di vertici**.

Supponiamo che $G = (V, E)$ sia **completo**. In questo caso è possibile esprimere $|E|$ come funzione di $|V|$?



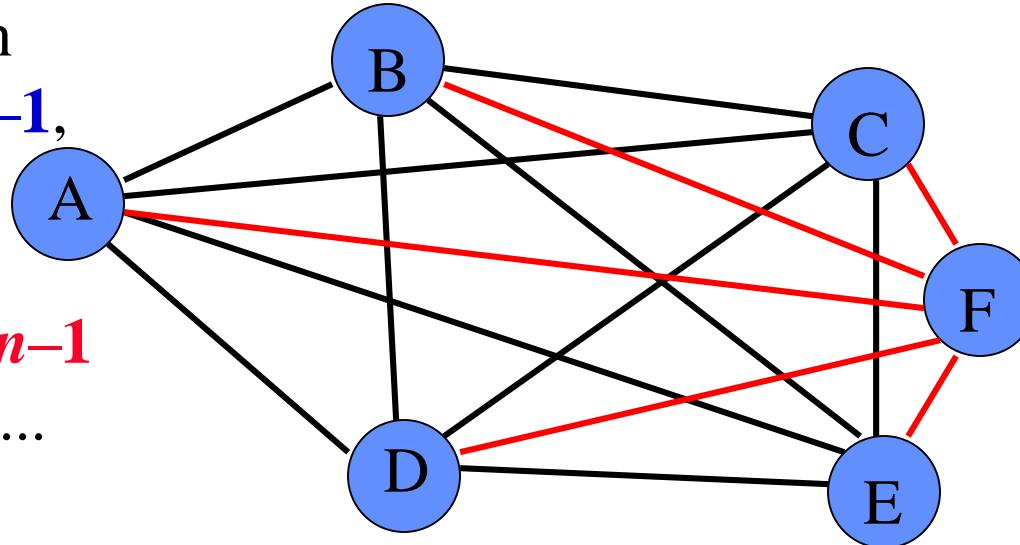
$ V $	$ E $
1	0
2	1
3	3
4	6
5	10
6	15

Definizioni sui grafi

Un **grafo completo** è un grafo che ha un **arco tra ogni coppia di vertici**.

Supponiamo che $G = (V, E)$ sia **completo**. In questo caso è possibile esprimere $|E|$ come funzione di $|V|$?

Per ottenere un grafo con n vertici da un grafo con $n-1$, si devono aggiungere $n-1$ nuovi archi ...



$ V $	$ E $
1	0
2	1
3	3
4	6
5	10
6	15

+1
+2
+3
+4
+5

Definizioni sui grafi

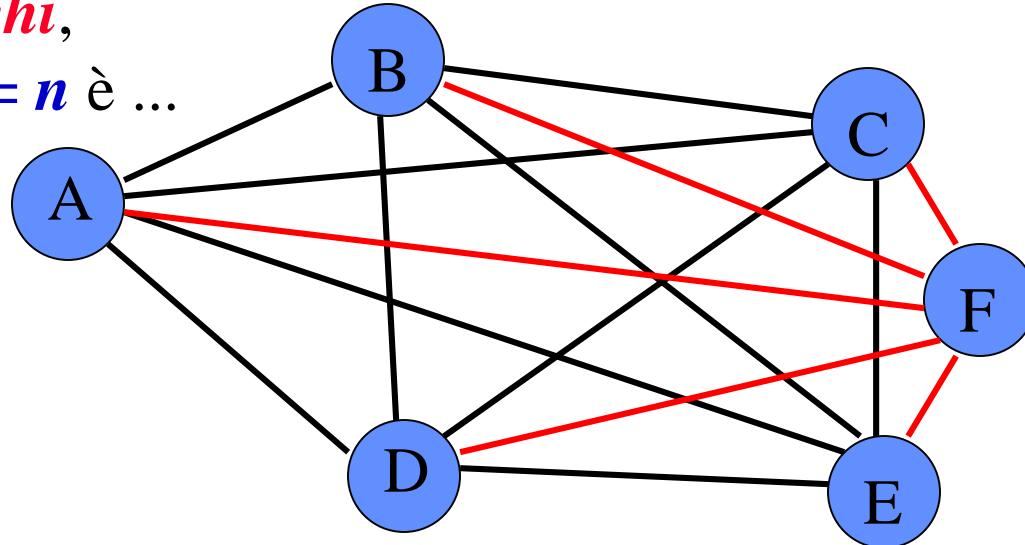
Un **grafo completo** è un grafo che ha un **arco tra ogni coppia di vertici**.

Supponiamo che $G = (V, E)$ sia **completo**. In questo caso è possibile esprimere $|E|$ come funzione di $|V|$?

...quindi il **numero**

totale di archi,

quando $|V| = n$ è ...



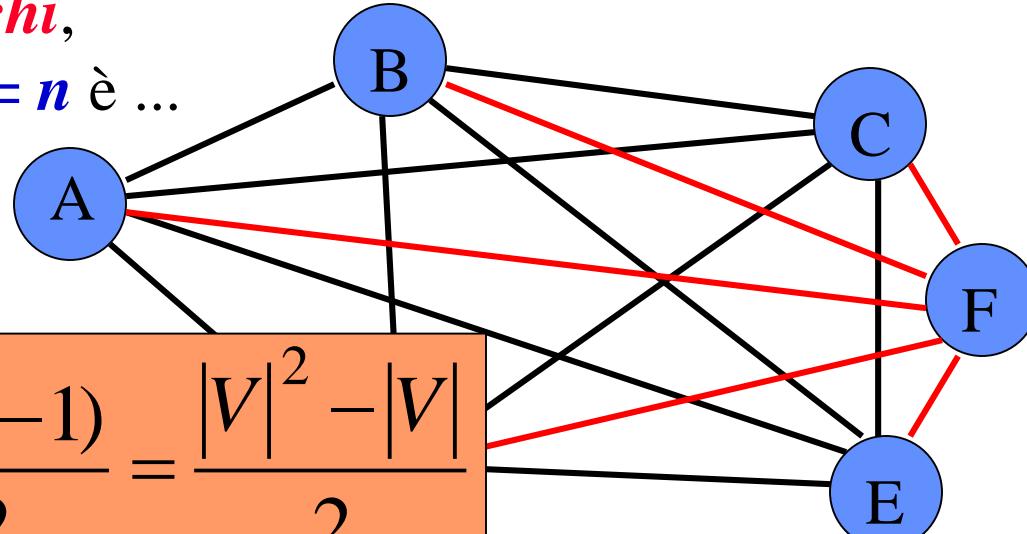
$ V $	$ E $
1	0
2	+1
3	+2
4	+3
5	+4
6	10
	+5
	15

Definizioni sui grafi

Un **grafo completo** è un grafo che ha un **arco tra ogni coppia di vertici**.

Supponiamo che $G = (V, E)$ sia **completo**. In questo caso è possibile esprimere $|E|$ come funzione di $|V|$?

...quindi il **numero totale di archi**,
quando $|V| = n$ è ...



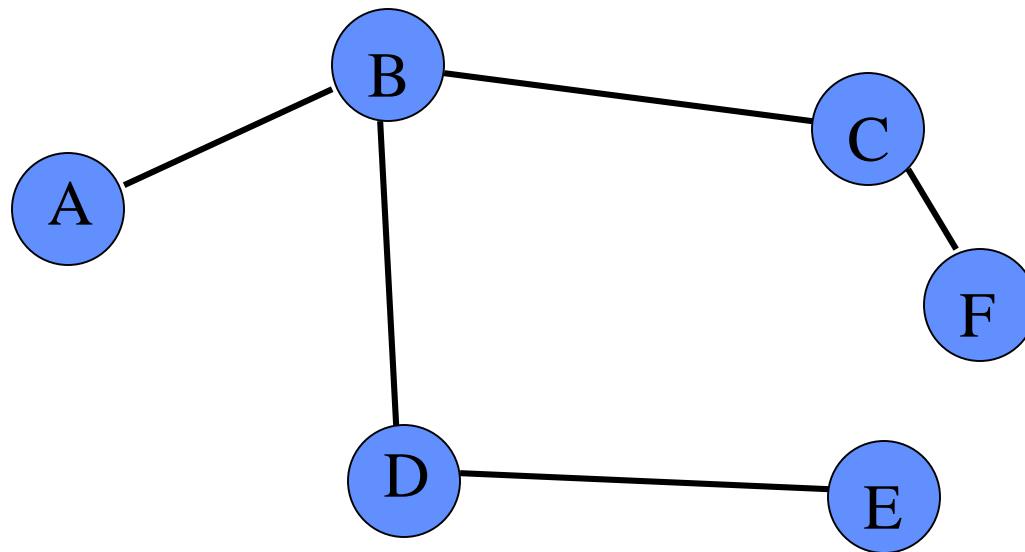
$$\sum_{i=1}^{n-1} i = \frac{n(n-1)}{2} = \frac{|V|^2 - |V|}{2}$$

$ V $	$ E $
1	0
2	+1
3	+2
4	+3
5	+4
6	6
7	10
8	+5
9	15

Definizioni sui grafi

Un *albero libero* è un *grafo non orientato connesso, aciclico*.

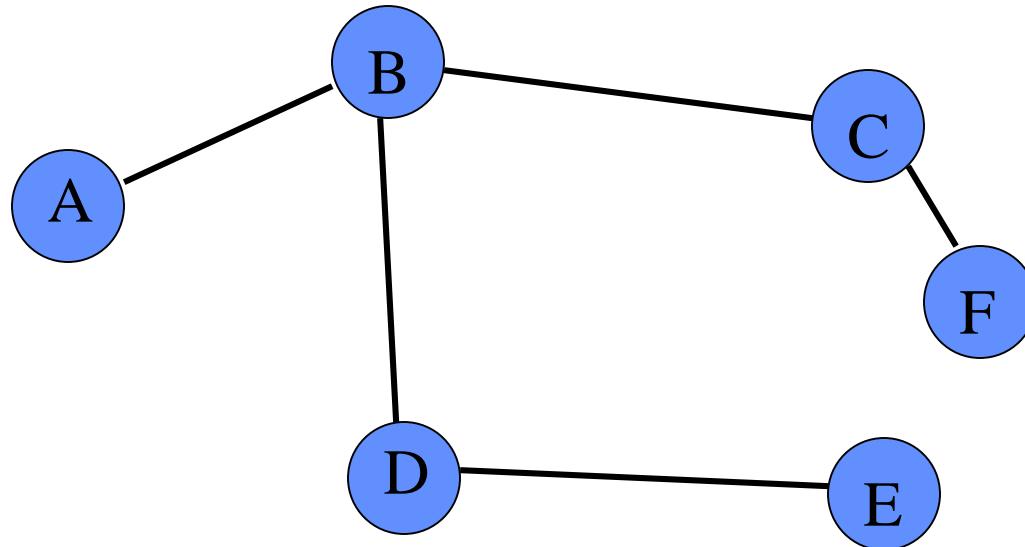
Questo è un *albero libero*



Definizioni sui grafi

Un *albero libero* è un *grafo non orientato connesso, aciclico*.

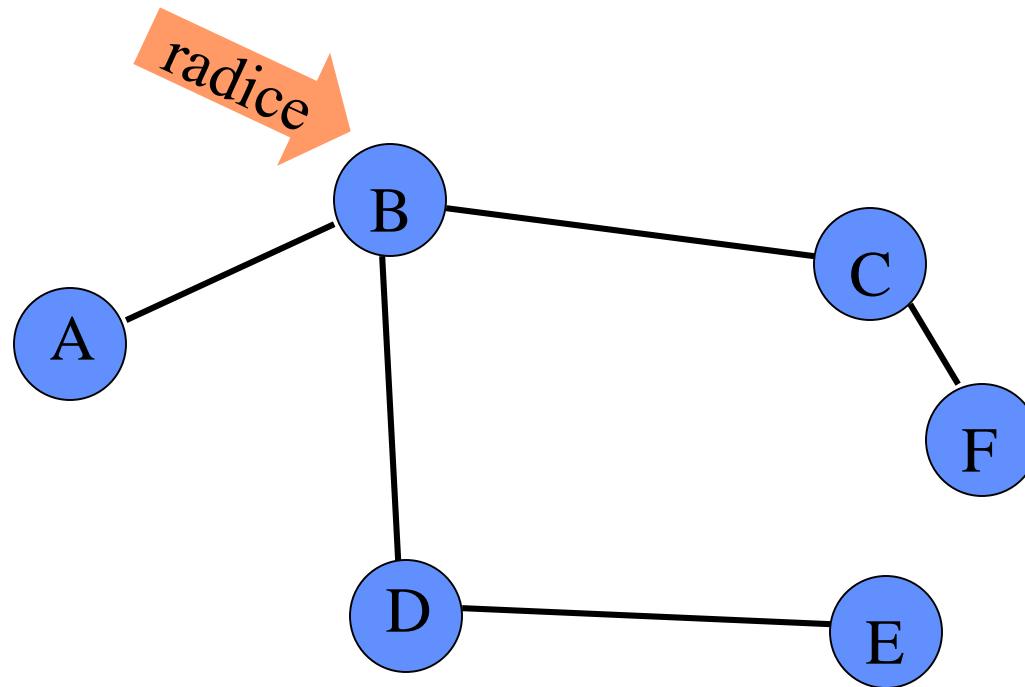
“*libero*” si riferisce al fatto che non esiste un vertice designato ad essere la “*radice*”



Definizioni sui grafi

Un *albero libero* è un *grafo non orientato连通的, aciclico*.

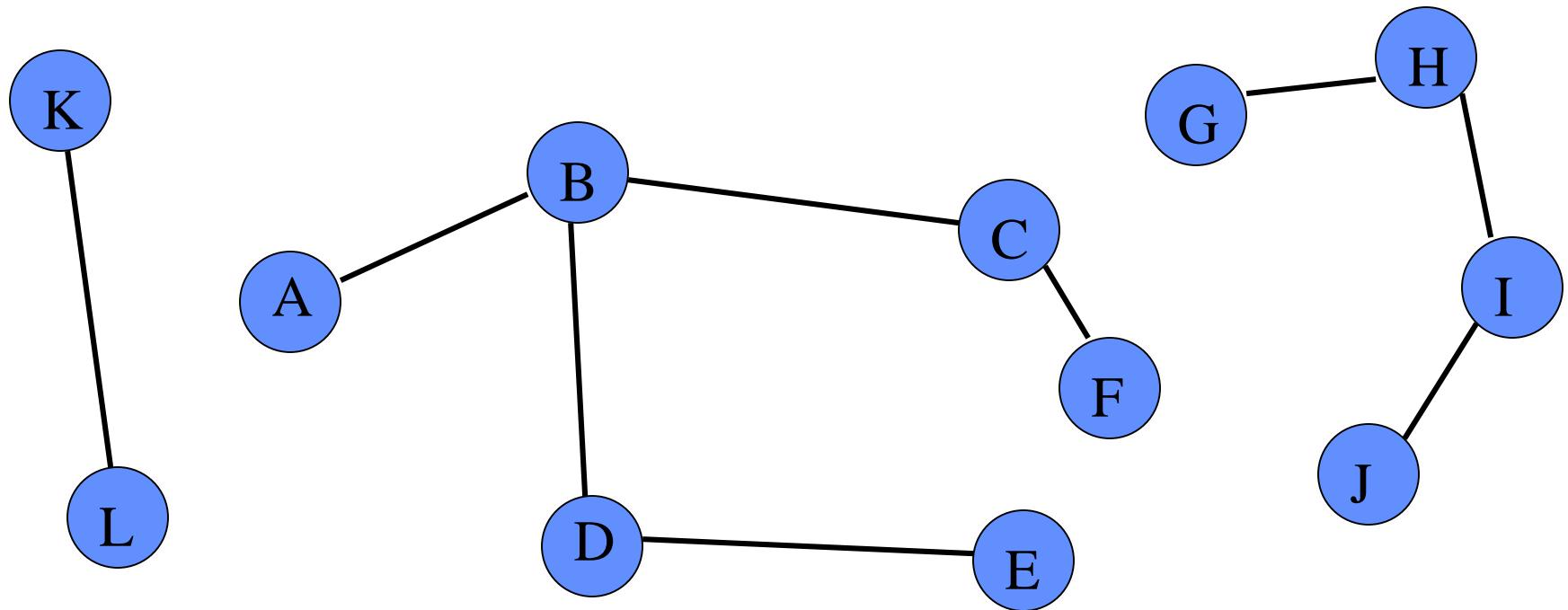
Se qualche *vertice* è designato ad essere la *radice*, otteniamo un *albero radicato*.



Definizioni sui grafi

Se un *grafo non orientato* è *aciclico* ma *sconnesso*, prende il nome di *foresta*.

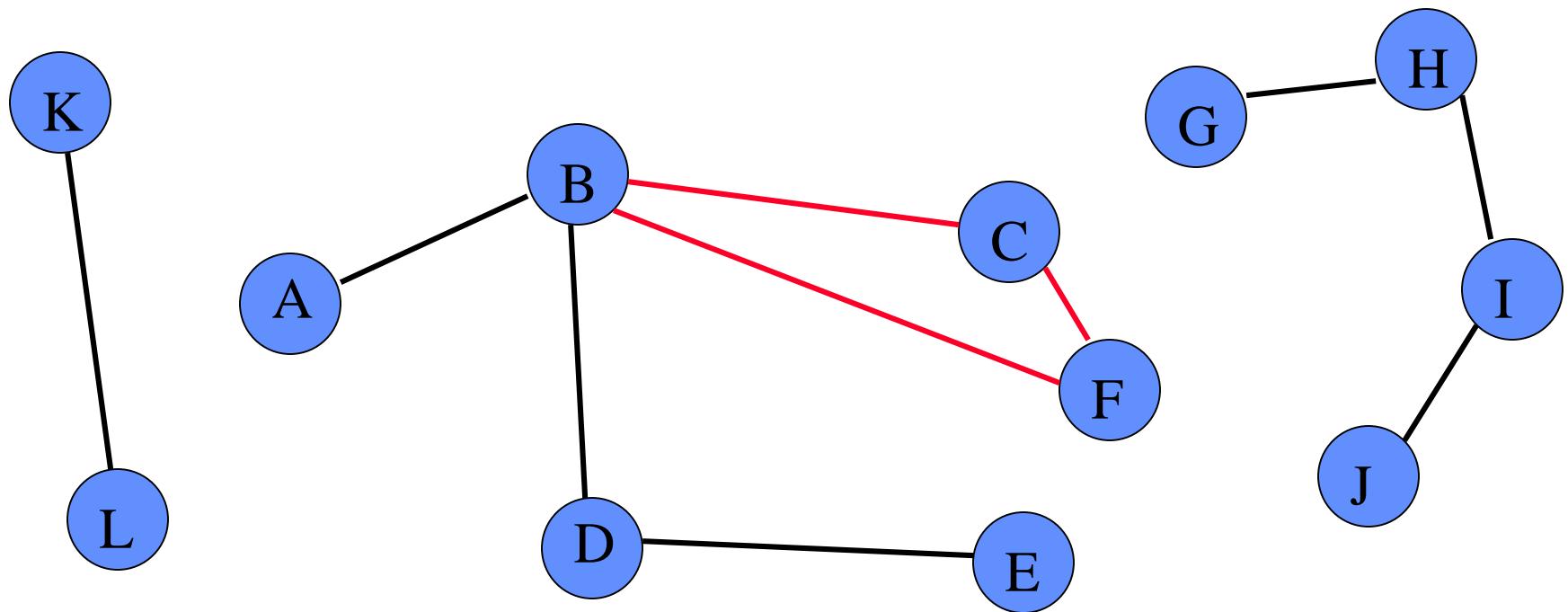
Questa è una *foresta*. Contiene tre alberi liberi.



Definizioni sui grafi

Se un *grafo non orientato* è *aciclico* ma *sconnesso*, prende il nome di *foresta*.

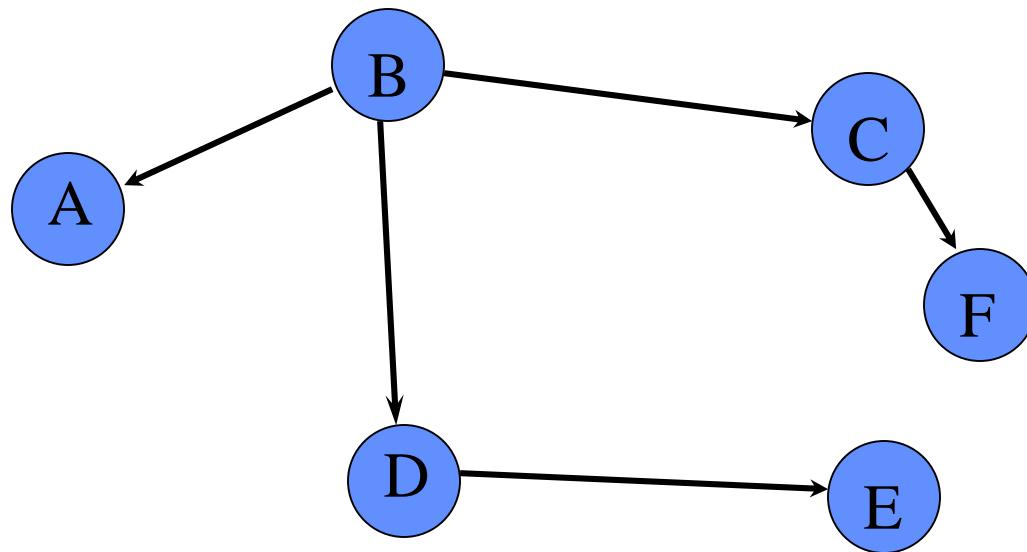
Questo *grafo contiene un ciclo*. Perciò *non é* un *né albero libero né una foresta*.



Definizioni sui grafi

Un **albero orientato** è un **grafo orientato aciclico** in cui

- 1.esiste solo nodo con grado entrante zero (la radice) e
- 2.ogni altro vertice ha grado entrante 1.



Rappresentazione di grafi

Ci sono due tipi di rappresentazione *standard* per grafi in un computer:

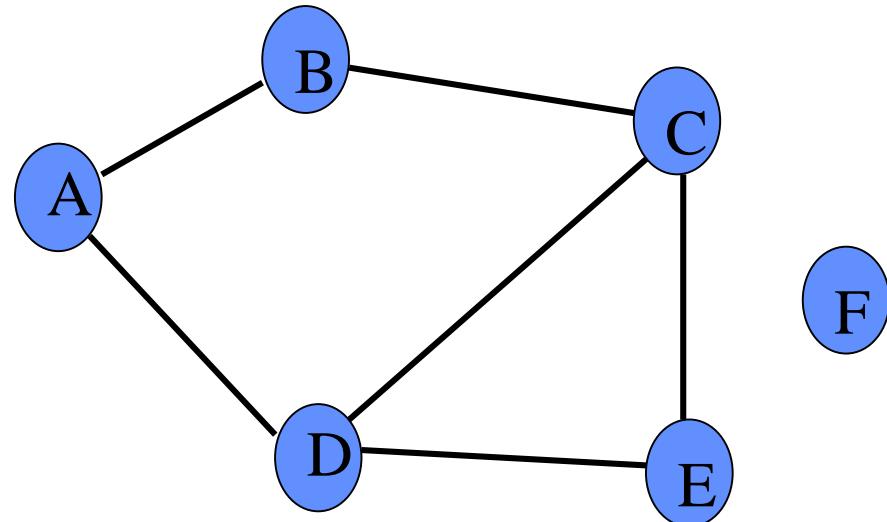
- Rappresentazione a *matrice di adiacenza*
- Rappresentazione a *liste di adiacenza*

Rappresentazione di grafi non orientati

Rappresentazione a *matrice di adiacenza*:

$$M(v, w) = \begin{cases} 1 & \text{se } (v, w) \in E \\ 0 & \text{altrimenti} \end{cases}$$

Spazio: $|V|^2$

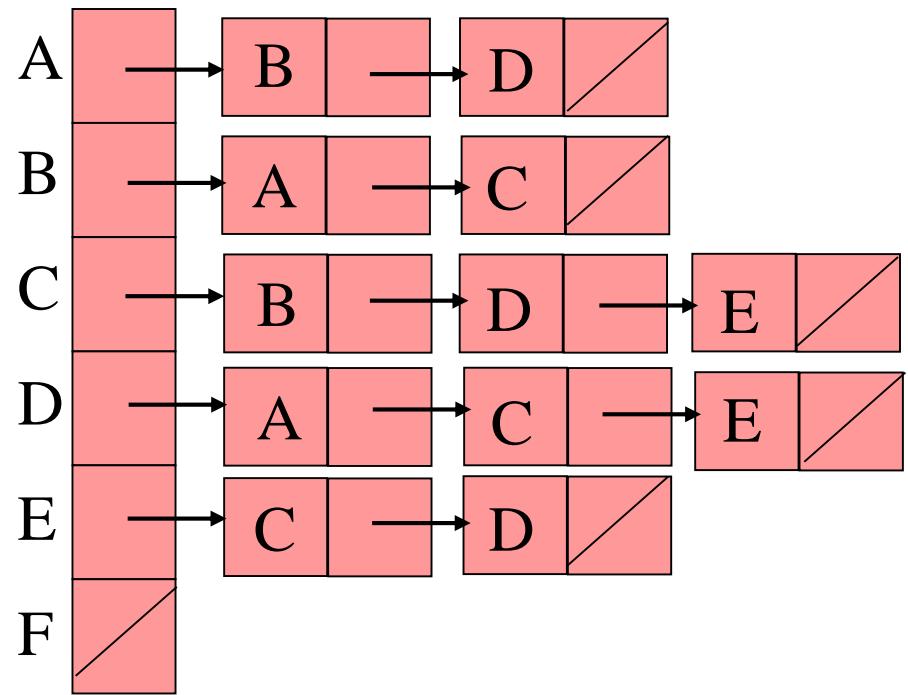
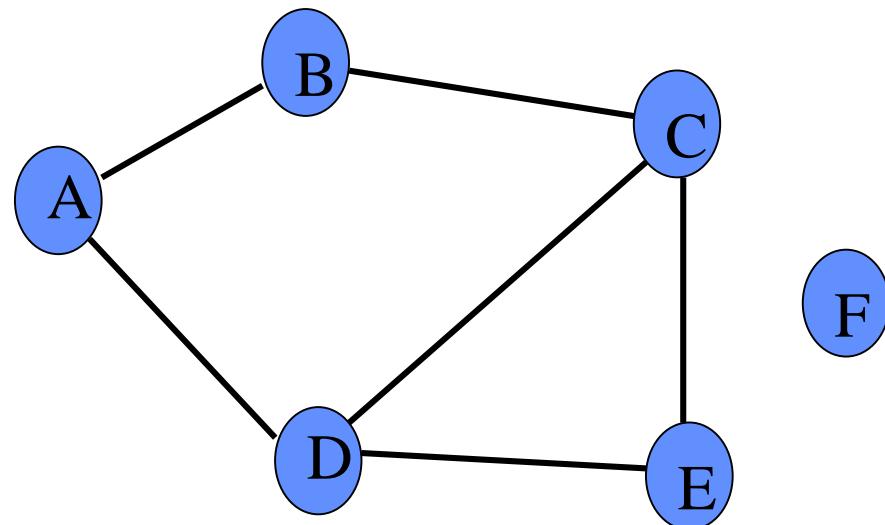


	A	B	C	D	E	F
A	0	1	0	1	0	0
B	1	0	1	0	0	0
C	0	1	0	1	1	0
D	1	0	1	0	1	0
E	0	0	1	1	0	0
F	0	0	0	0	0	0

Rappresentazione di grafi non orientati

Rappresentazione a *liste di adiacenza*:

$L(v)$ = lista di w , tale che $(v, w) \in E$,
per $v \in V$

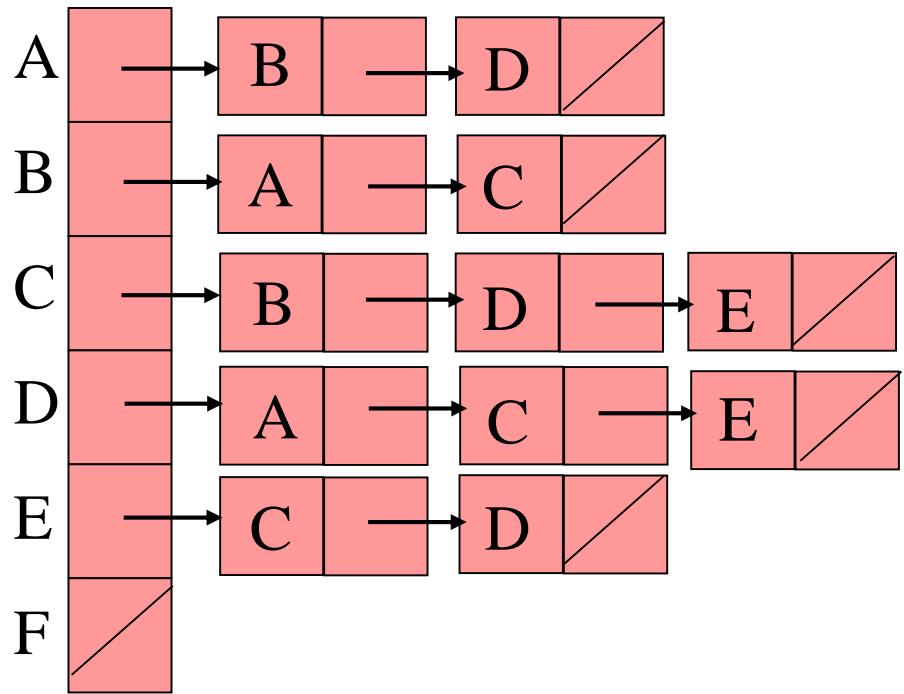
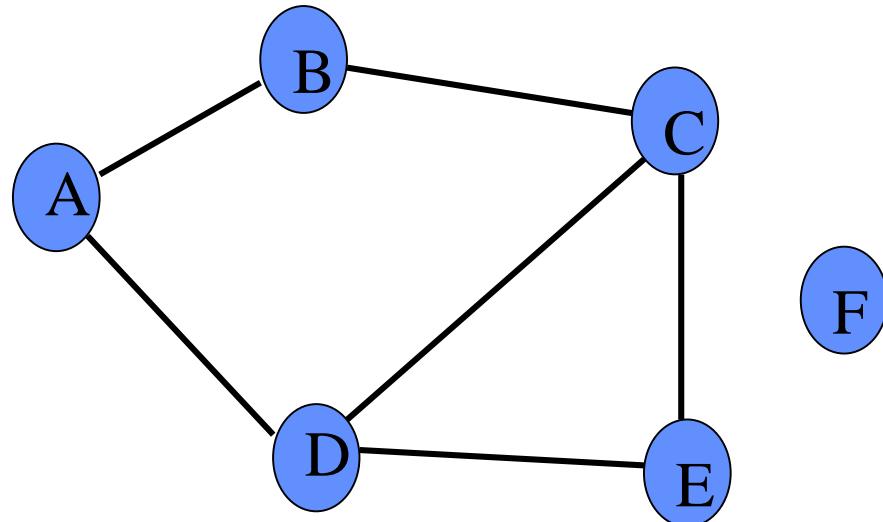


Rappresentazione di grafi non orientati

Rappresentazione a *liste di adiacenza*:

$L(v)$ = lista di w , tale che $(v, w) \in E$,
per $v \in V$

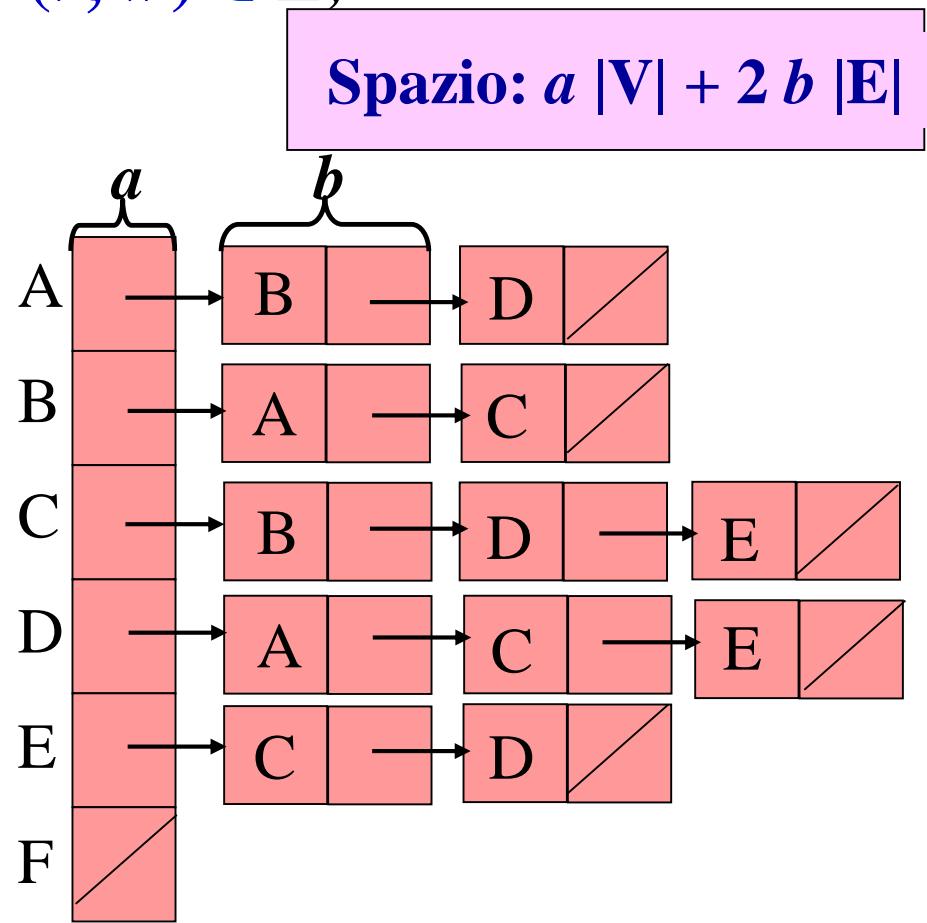
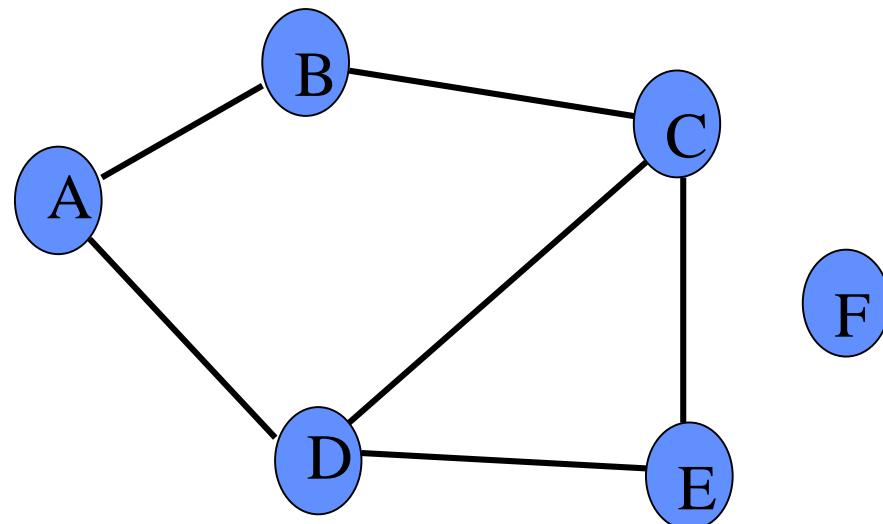
Quanto spazio ?



Rappresentazione di grafi non orientati

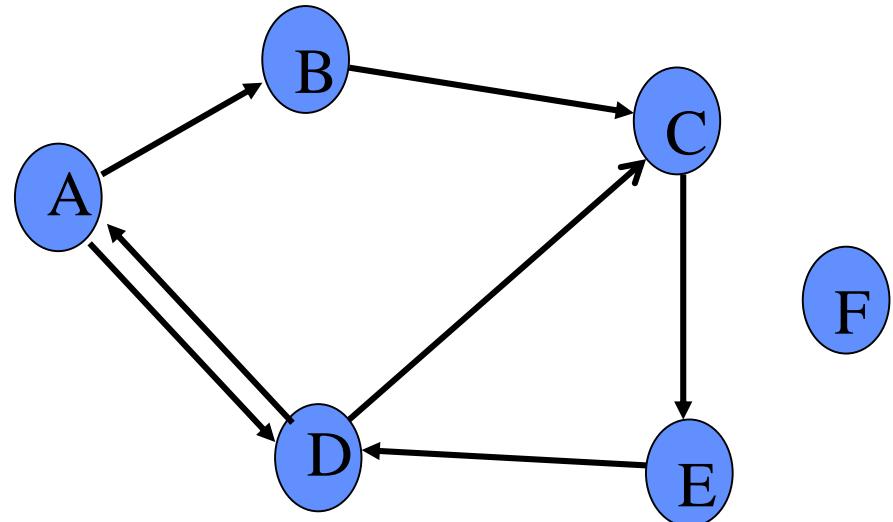
Rappresentazione a *liste di adiacenza*:

$L(v)$ = lista di w , tale che $(v, w) \in E$,
per $v \in V$



Rappresentazione di grafi orientati

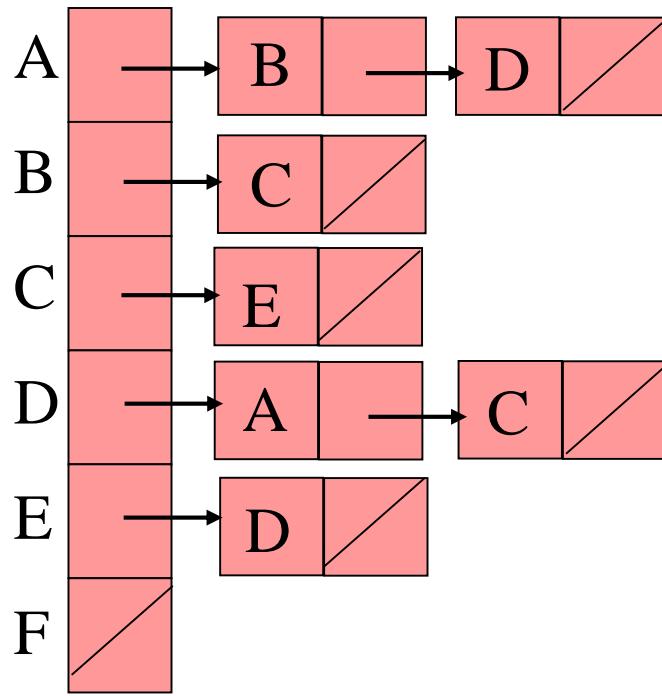
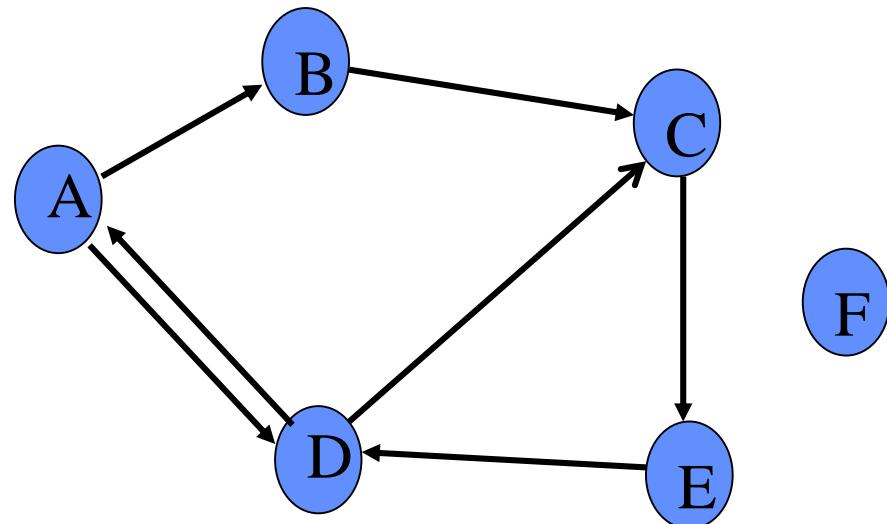
Rappresentazione a *matrice di adiacenza* questa volta per rappresentare un *grafo orientato*.



	A	B	C	D	E	F
A	0	1	0	1	0	0
B	0	0	1	0	0	0
C	0	0	0	0	1	0
D	1	0	1	0	0	0
E	0	0	0	1	0	0
F	0	0	0	0	0	0

Rappresentazione di grafi orientati

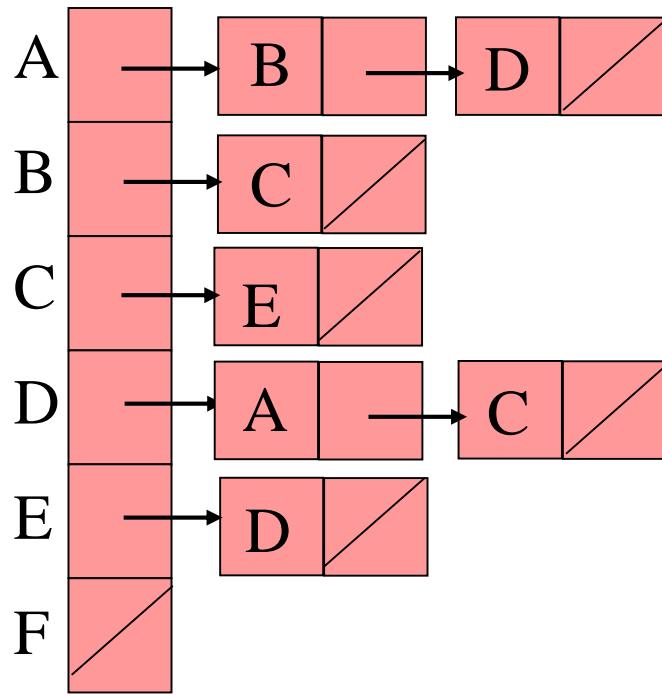
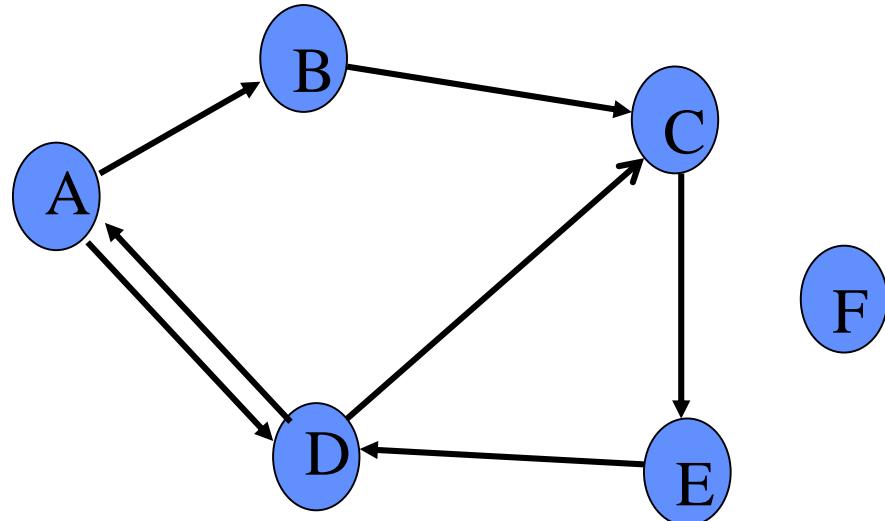
Rappresentazione a *liste di adiacenza* questa volta per rappresentare un *grafo orientato*.



Rappresentazione di grafi orientati

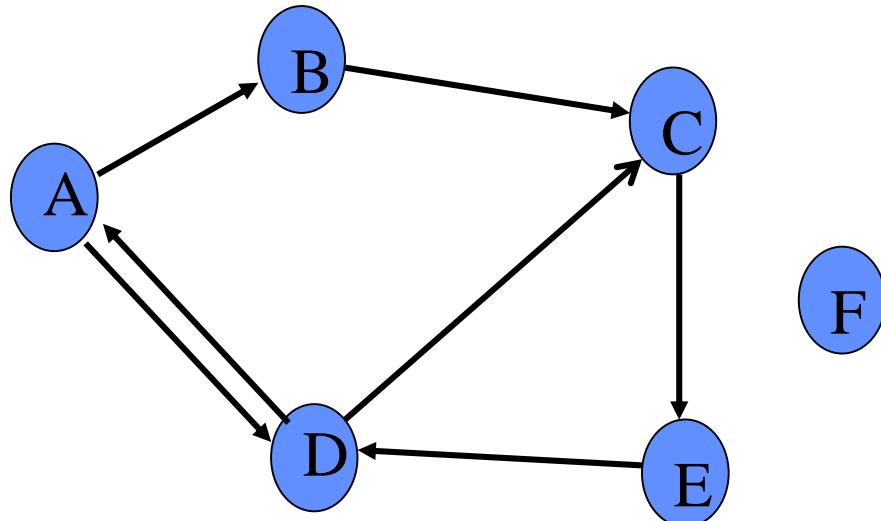
Rappresentazione a *liste di adiacenza* questa volta per rappresentare un *grafo orientato*.

Quanto spazio?

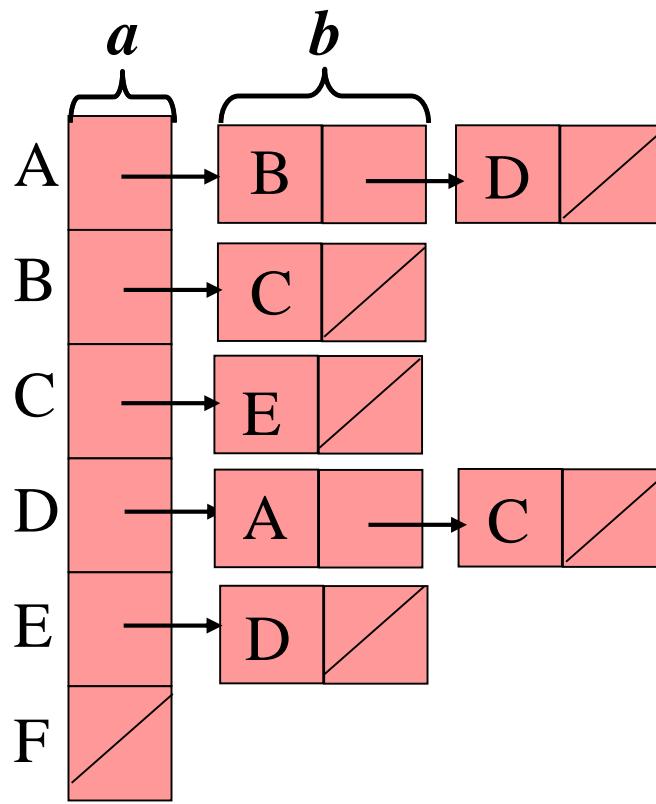


Rappresentazione di grafi orientati

Rappresentazione a *liste di adiacenza* questa volta per rappresentare un *grafo orientato*.



Spazio: $a |V| + b |E|$



Rappresentazione di grafi

- **Matrice di adiacenza**
 - Spazio richiesto $O(|V|^2)$
 - Verificare se i vertici u e v sono adiacenti richiede tempo $O(1)$.
 - Molti **0** nel caso di *grafi sparsi*
- **Liste di adiacenza**
 - Spazio richiesto $O(|E|+|V|)$
 - Verificare se i vertici u e v sono adiacenti richiede tempo $O(|V|)$.

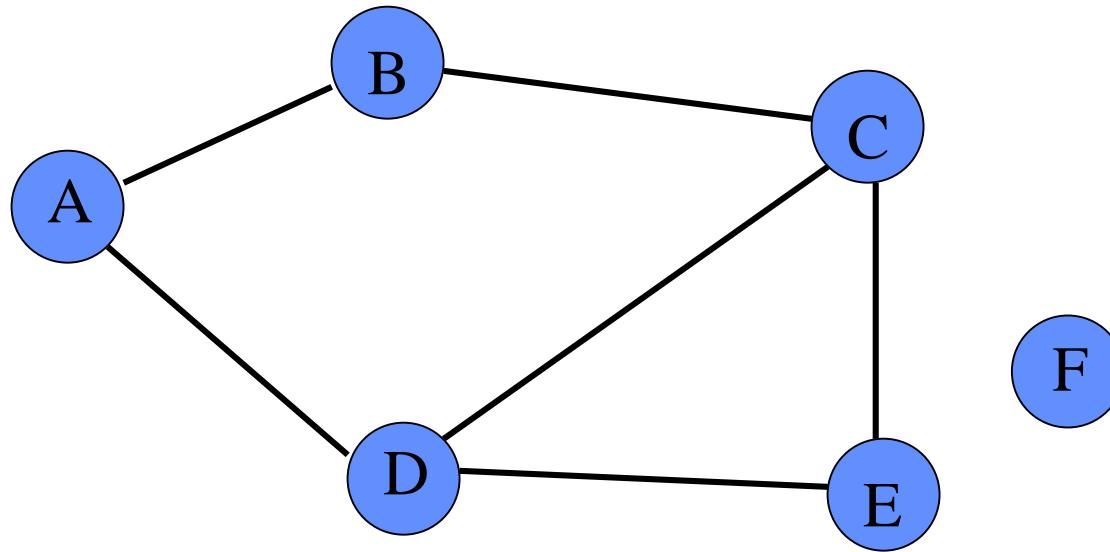
Algoritmi e Strutture Dati (Mod. B)

Algoritmi su grafi
Ricerca in ampiezza
(Breadth-First Search)

Definizione del problema

Attraversamento di un grafo

Dato un grafo $G = \langle V, E \rangle$ ed un vertice s di V (detto *sorgente*), esplorare *ogni vertice raggiungibile* nel grafo dal vertice s

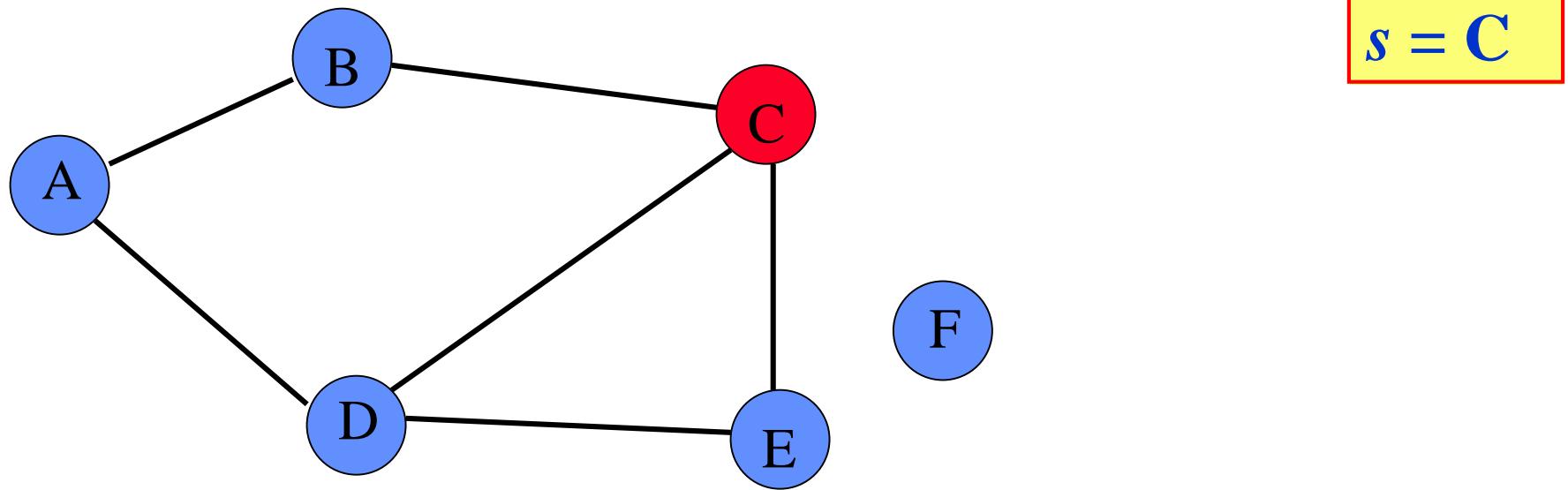


$s = C$

Definizione del problema

Attraversamento di un grafo

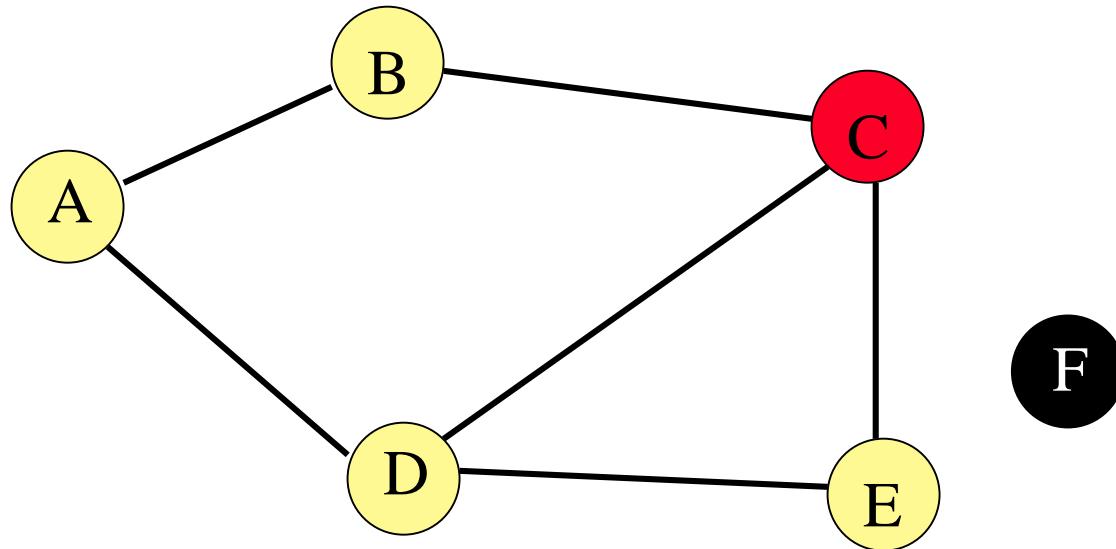
Dato un grafo $G = \langle V, E \rangle$ ed un vertice s di V (detto *sorgente*), esplorare *ogni vertice raggiungibile* nel grafo dal vertice s



Definizione del problema

Attraversamento di un grafo

Dato un grafo $G = \langle V, E \rangle$ ed un vertice s di V (detto *sorgente*), esplorare *ogni vertice raggiungibile* nel grafo dal vertice s



$s = C$

F è l'unico vertice non raggiungibile

Algoritmo BFS per alberi

Visita-Aampiezza (*T*:albero)

Coda = {*T*}

while *Coda* ≠ Ø do

u = *Testa*[*Coda*]

 “visita *u*”

 for each “figlio *F* di *u* da sinistra” do

 Accoda(*Coda*, *F*)

 Decoda(*Coda*)

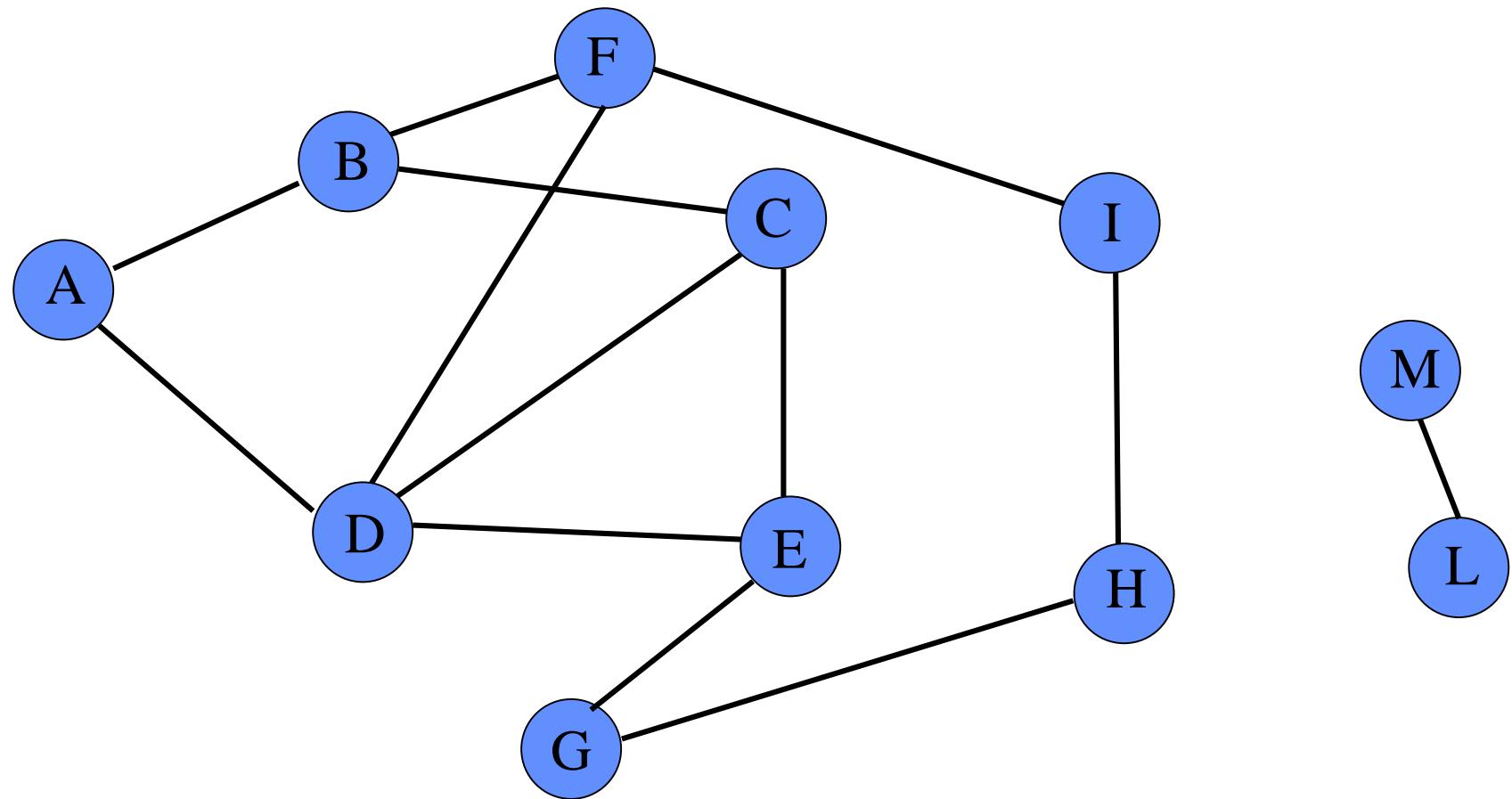
Nel nostro algoritmo generico per i grafi, come operazione di “visita” di un vertice useremo la memorizzazione di quale sia il “padre” (o predecessore) del vertice durante la *BFS*.

Algoritmo BFS: I

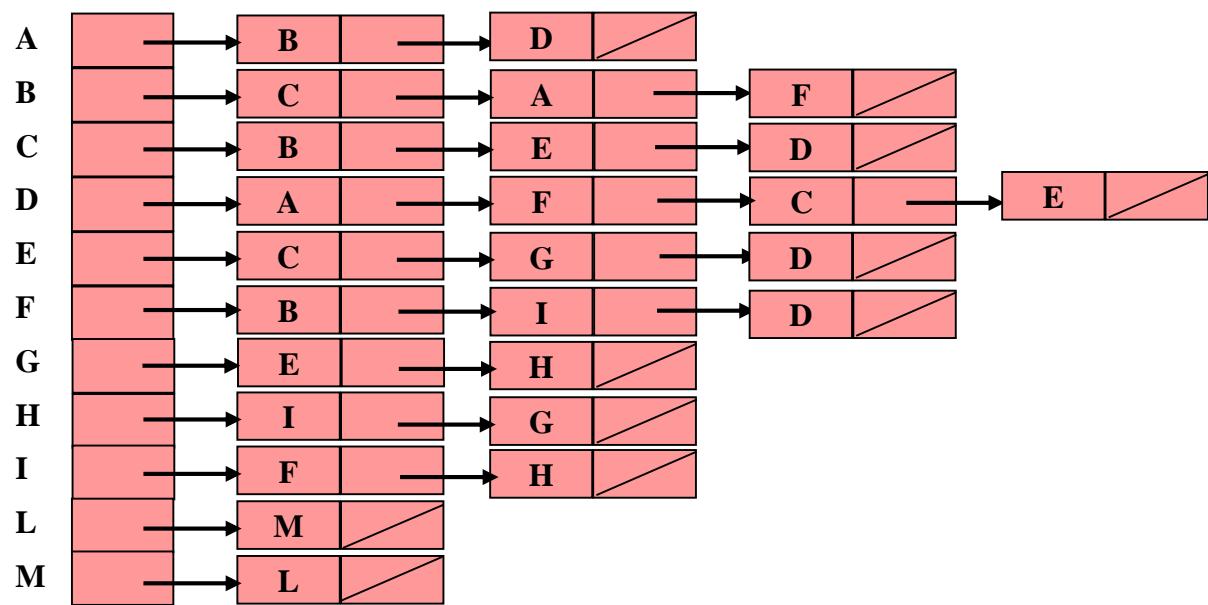
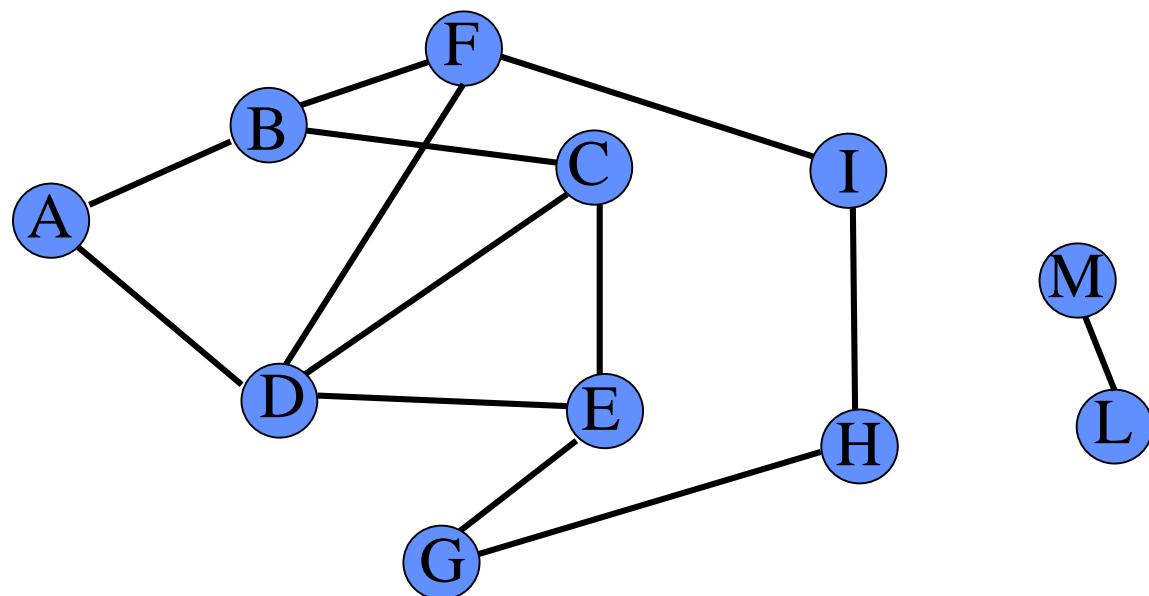
```
BSF(G:grafo, s:vertice)
  pred[s] = Nil
  Coda = {s}
  while Coda ≠ ∅ do
    u = Testa[Coda]
    for each v ∈ Adiac(u) do
      pred[v] = u
      Accoda(Coda, v)
    Decoda(Coda)
```

```
Visita-Ampiezza(T:albero)
  Coda = {T}
  while Coda ≠ ∅ do
    u = Testa[Coda ]
    "visita u"
    for each "figlio F di u da sinistra" do
      Accoda(Coda, F)
    Decoda(Coda)
```

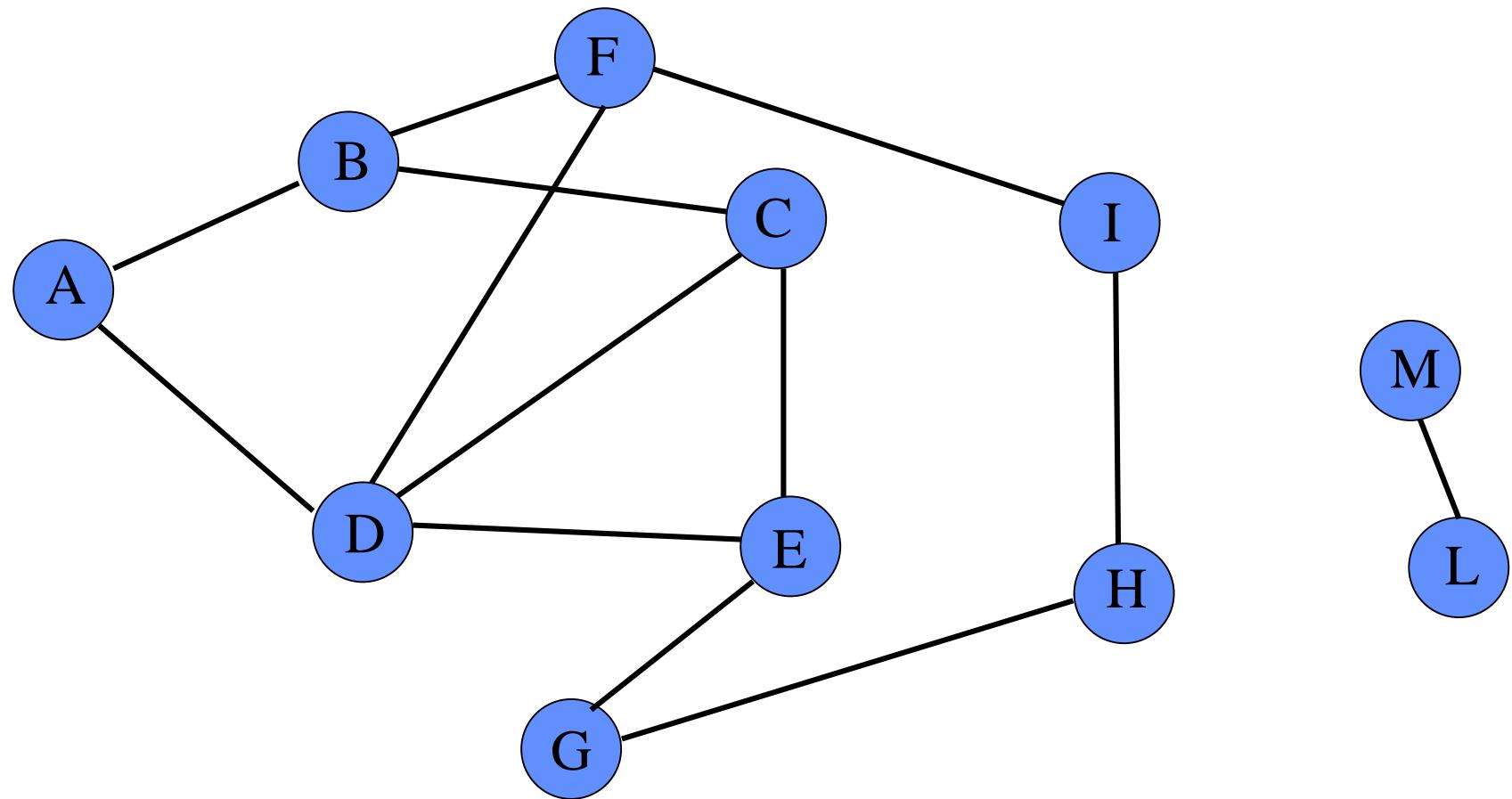
Algoritmo BFS I



Algoritmo BFS I



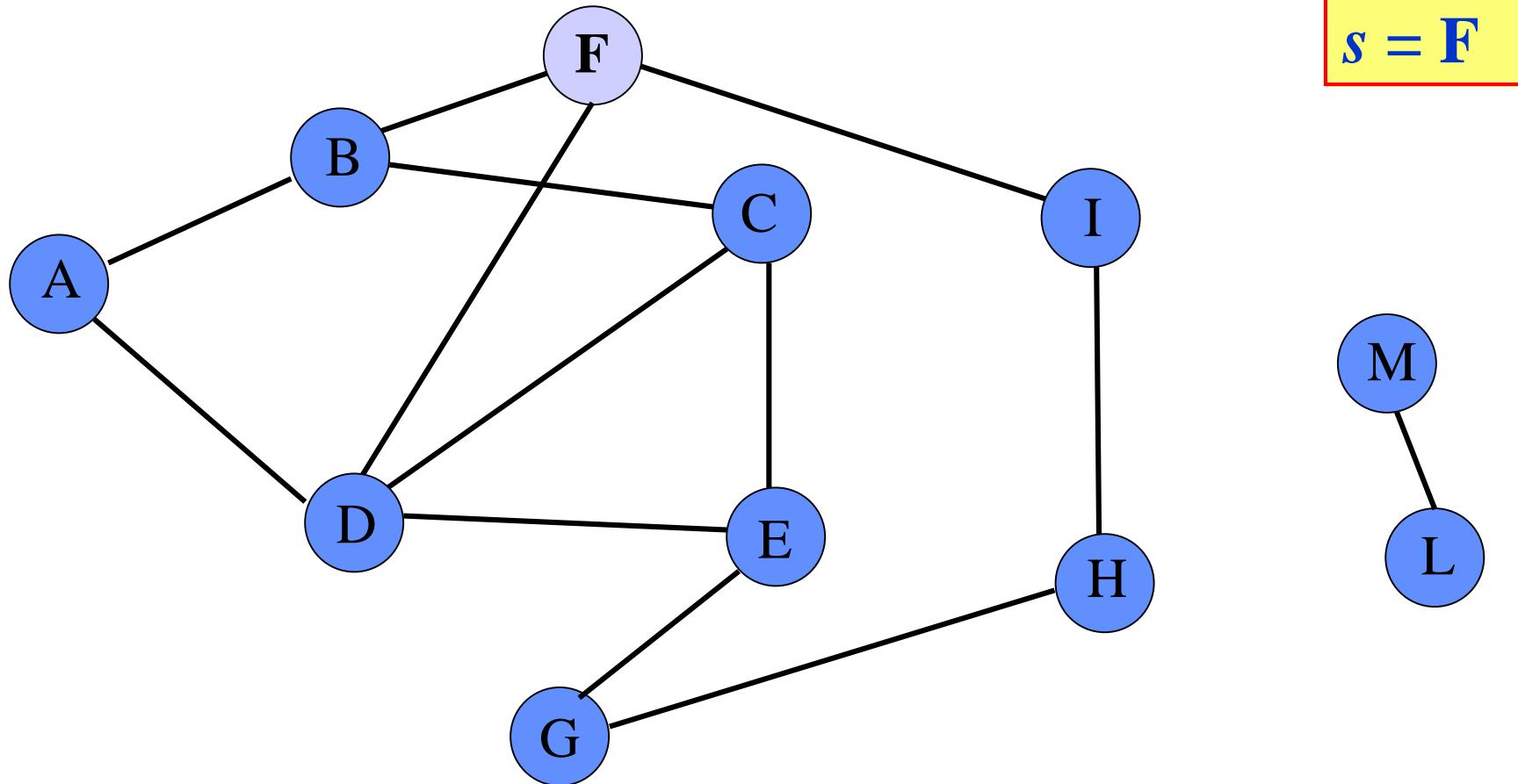
Algoritmo BFS I



Coda : { }

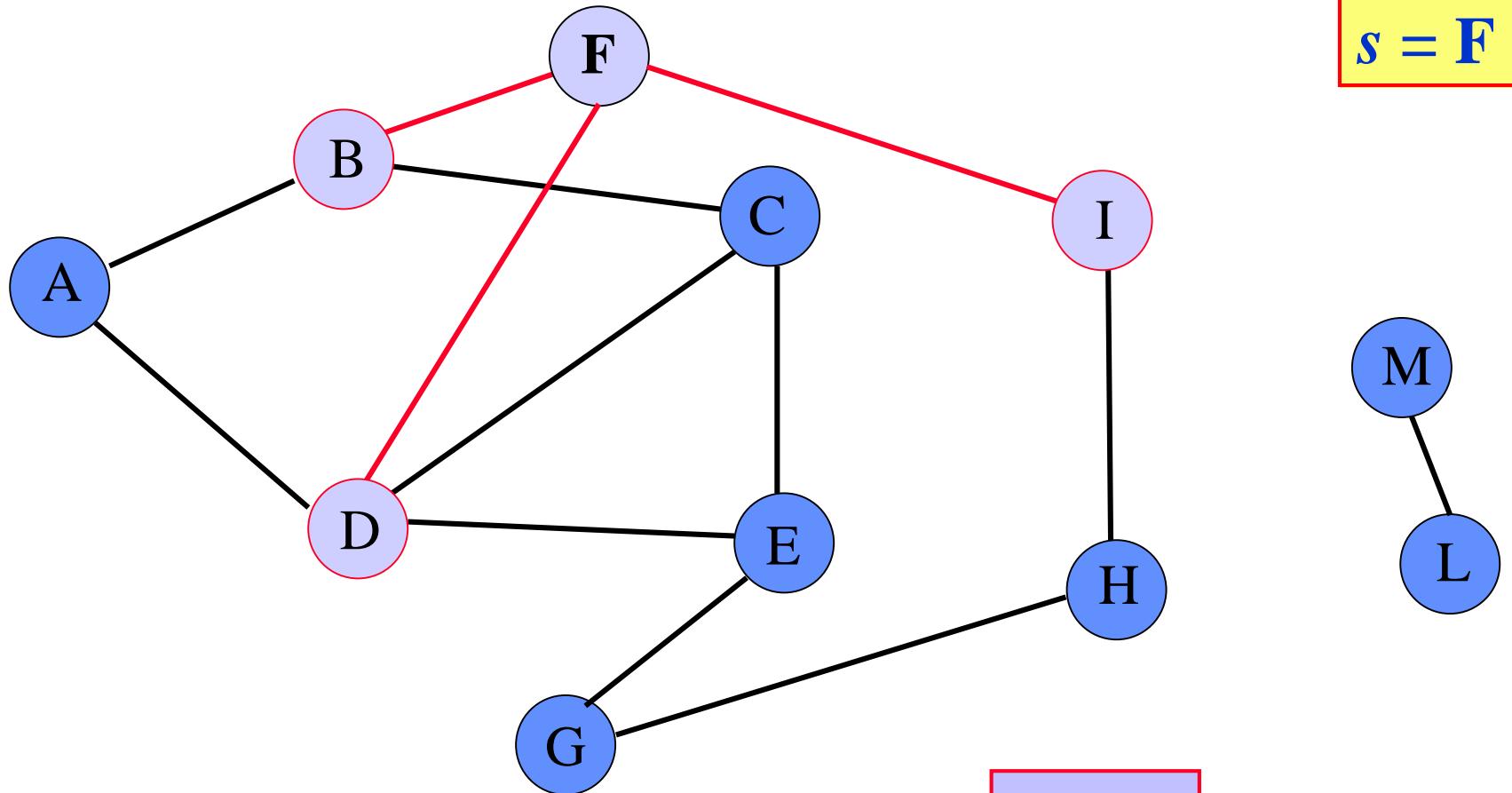
Algoritmo BFS I

```
for each  $v \in \text{Adiac}(u)$ 
  do  $\text{pred}[v] = u$ 
      Accoda (Coda,  $v$ )
Decoda ( $u$ )
```

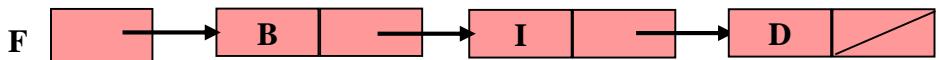


Algoritmo BFS I

```
for each  $v \in \text{Adiac}(u)$ 
  do  $\text{pred}[v] = u$ 
      Accoda (Coda,  $v$ )
Decoda ( $u$ )
```

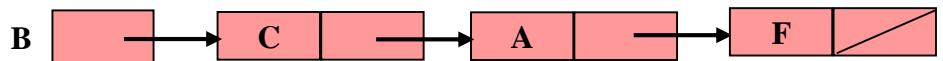
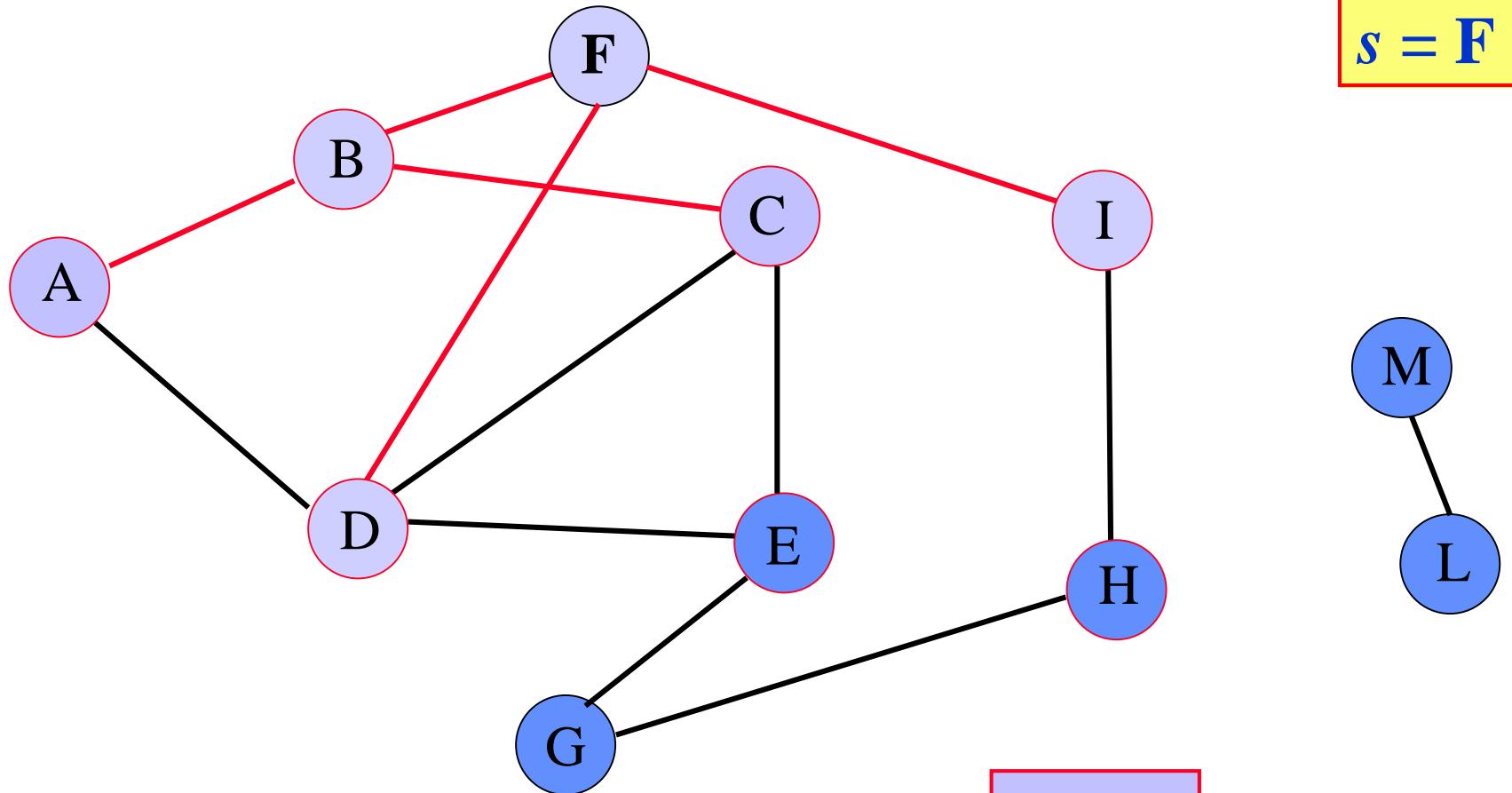


Coda : {**B , I , D**}



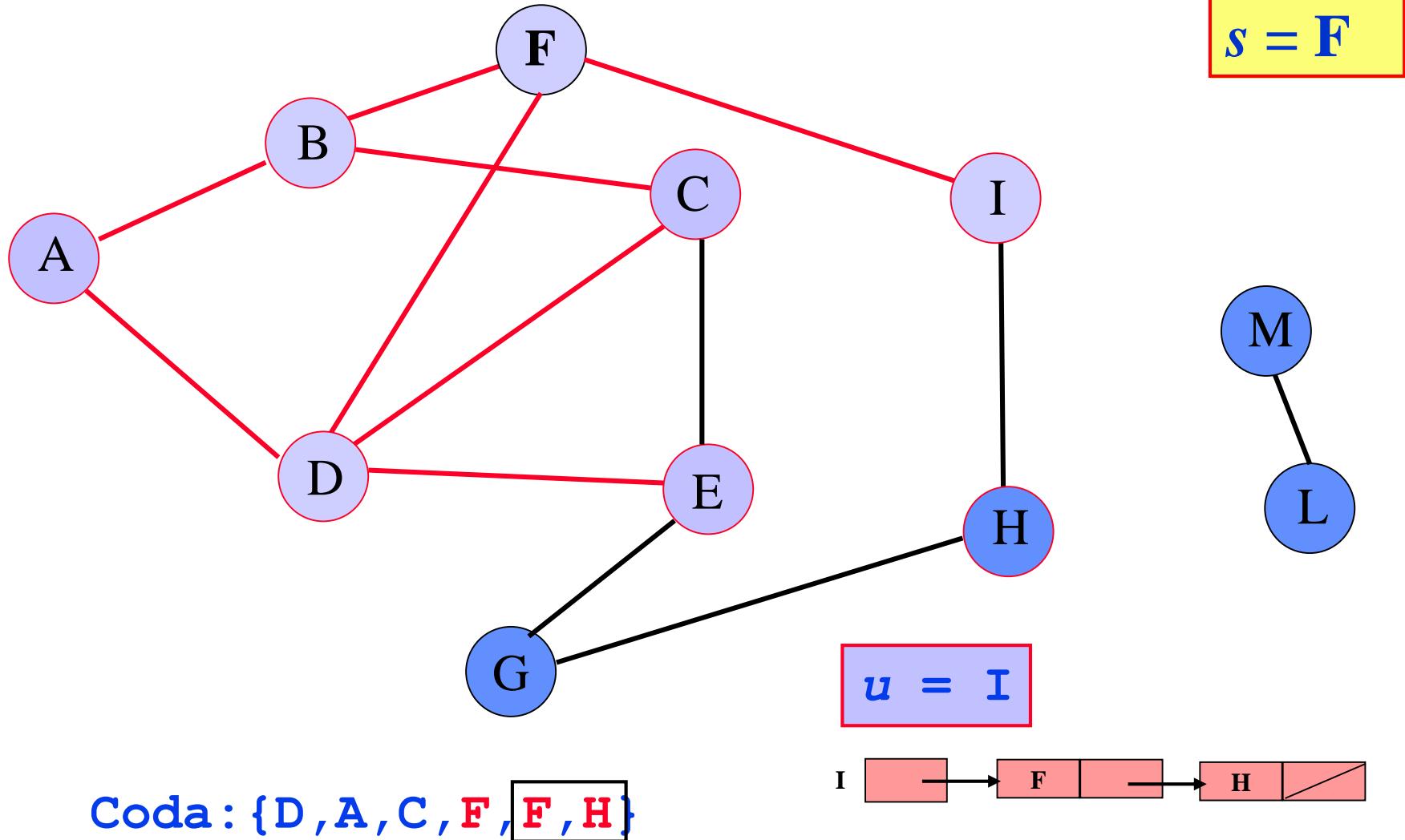
Algoritmo BFS I

```
for each  $v \in \text{Adiac}(u)$ 
do  $\text{pred}[v] = u$ 
    Accoda (Coda,  $v$ )
Decoda ( $u$ )
```



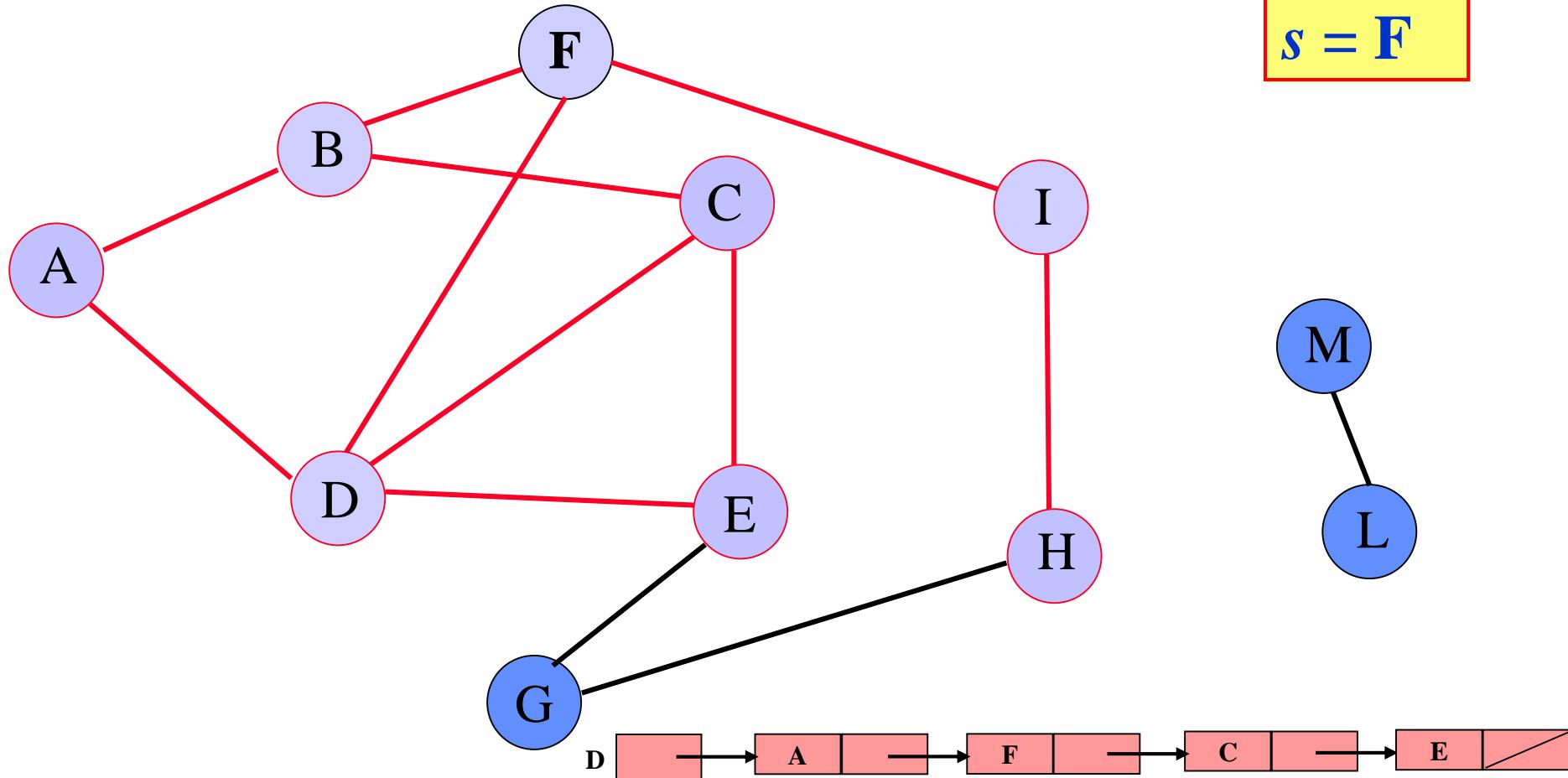
Algoritmo BFS I

```
for each  $v \in \text{Adiac}(u)$ 
  do  $\text{pred}[v] = u$ 
      Accoda (Coda,  $v$ )
Decoda ( $u$ )
```



Algoritmo BFS I

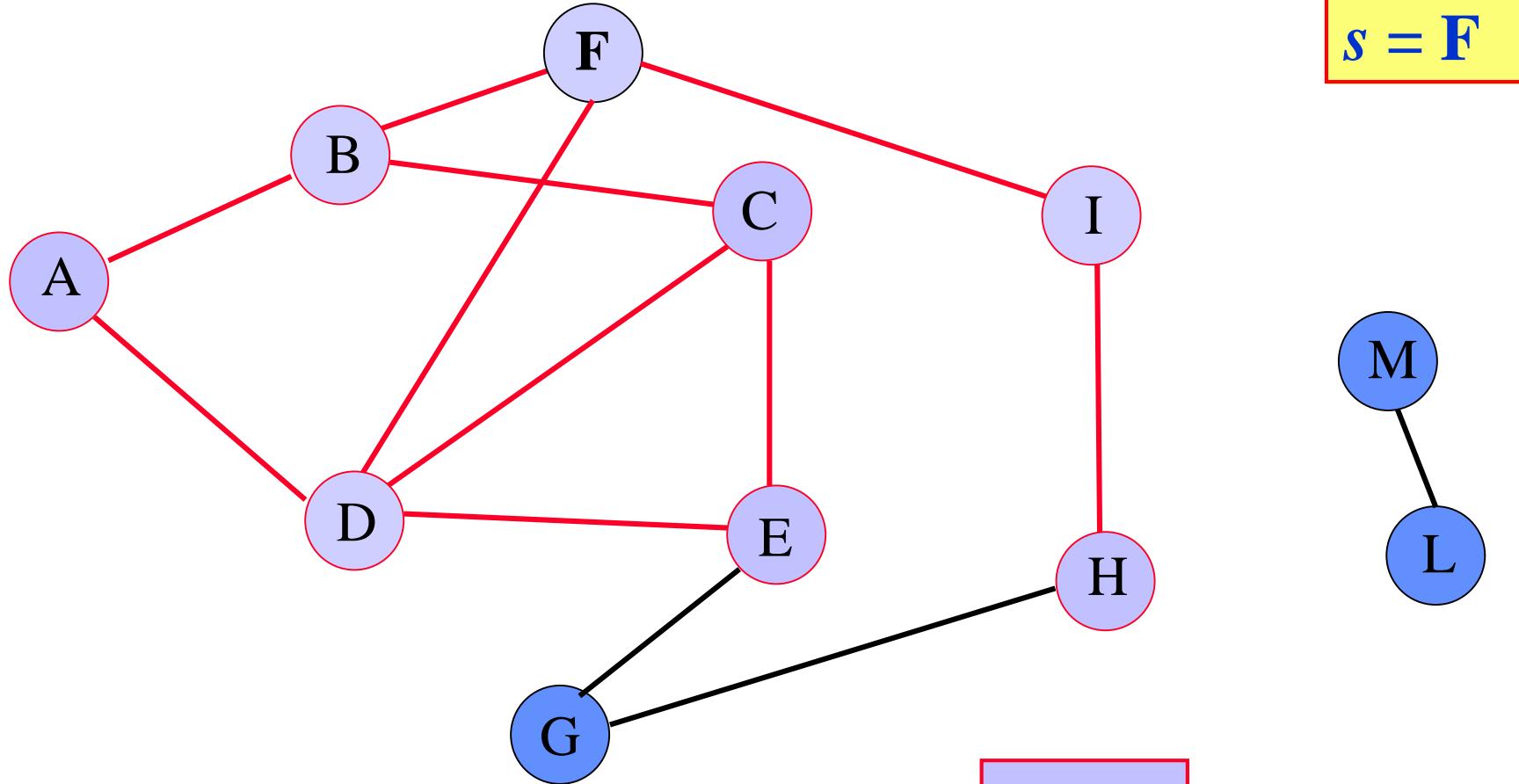
```
for each  $v \in \text{Adiac}(u)$ 
do  $\text{pred}[v] = u$ 
    Accoda (Coda,  $v$ )
Decoda ( $u$ )
```



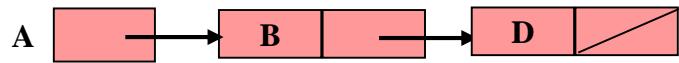
Coda : {A, C, F, F, H, A, F, C, E}

Algoritmo BFS I

```
for each  $v \in \text{Adiac}(u)$ 
do  $\text{pred}[v] = u$ 
    Accoda(Coda, v)
Decoda(u)
```

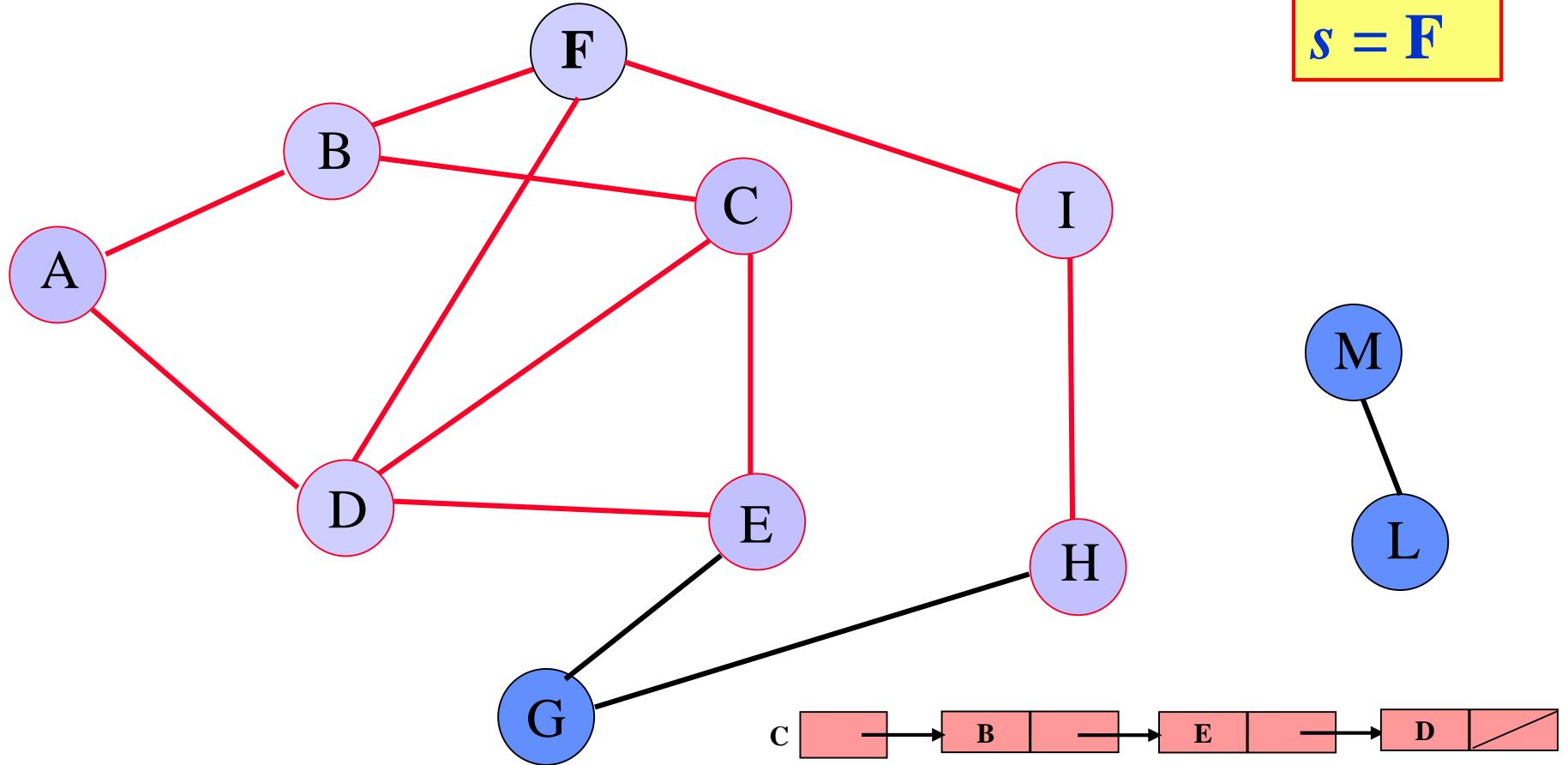


Coda : {C, F, E, H, A, F, C, E, B, D}



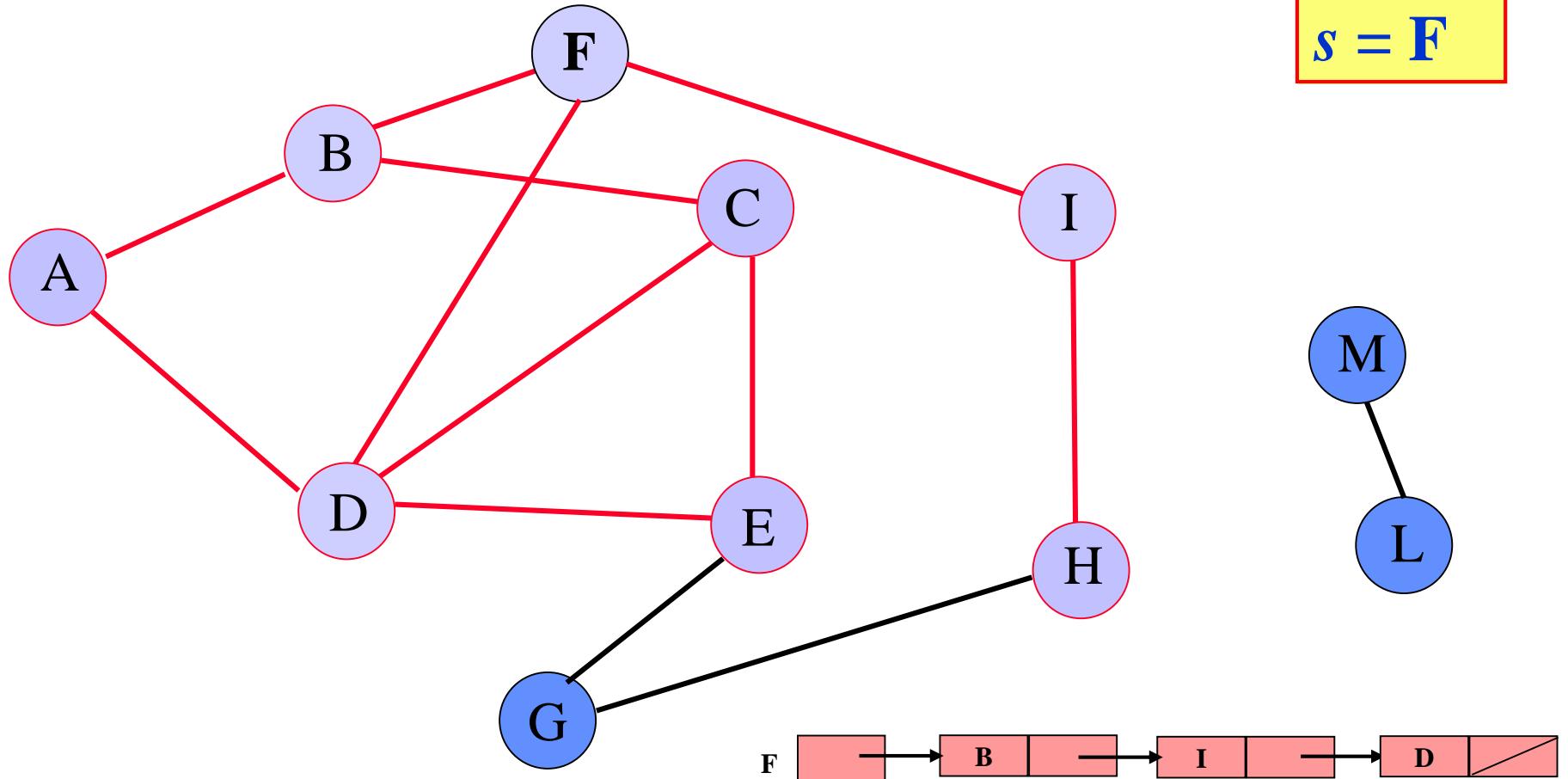
Algoritmo BFS I

```
for each  $v \in \text{Adiac}(u)$ 
do  $\text{pred}[v] = u$ 
    Accoda(Coda, v)
Decoda(u)
```



Algoritmo BFS I

```
for each  $v \in \text{Adiac}(u)$ 
do  $\text{pred}[v] = u$ 
    Accoda(Coda, v)
Decoda(u)
```



Coda: {F, F, H, A, F, C, E, B, D, B, E, D, B, I, D}

Algoritmo BFS I: problema

- È necessario ricordarsi dei nodi che abbiamo già visitato per non rivisitarli nuovamente.
- Dobbiamo distinguere tra i vertici *non scoperti*, quelli *scoperti* e quelli *visitati*.

Algoritmo BFS I: problema

- È necessario ricordarsi dei nodi che abbiamo già visitato per non rivisitarli nuovamente.
- Dobbiamo distinguere tra i vertici *non scoperti*, quelli *scoperti* e quelli *visitati*.
 - un vertice è stato *scoperto* se è comparso nella coda
 - un vertice è stato *non scoperto* se non è mai comparso nella coda
 - un vertice è stato *visitato* se è comparso in coda ma non è più in coda (tutti i vertici ad esso adiacenti sono già stati visitati).

Algoritmo BFS II: soluzione

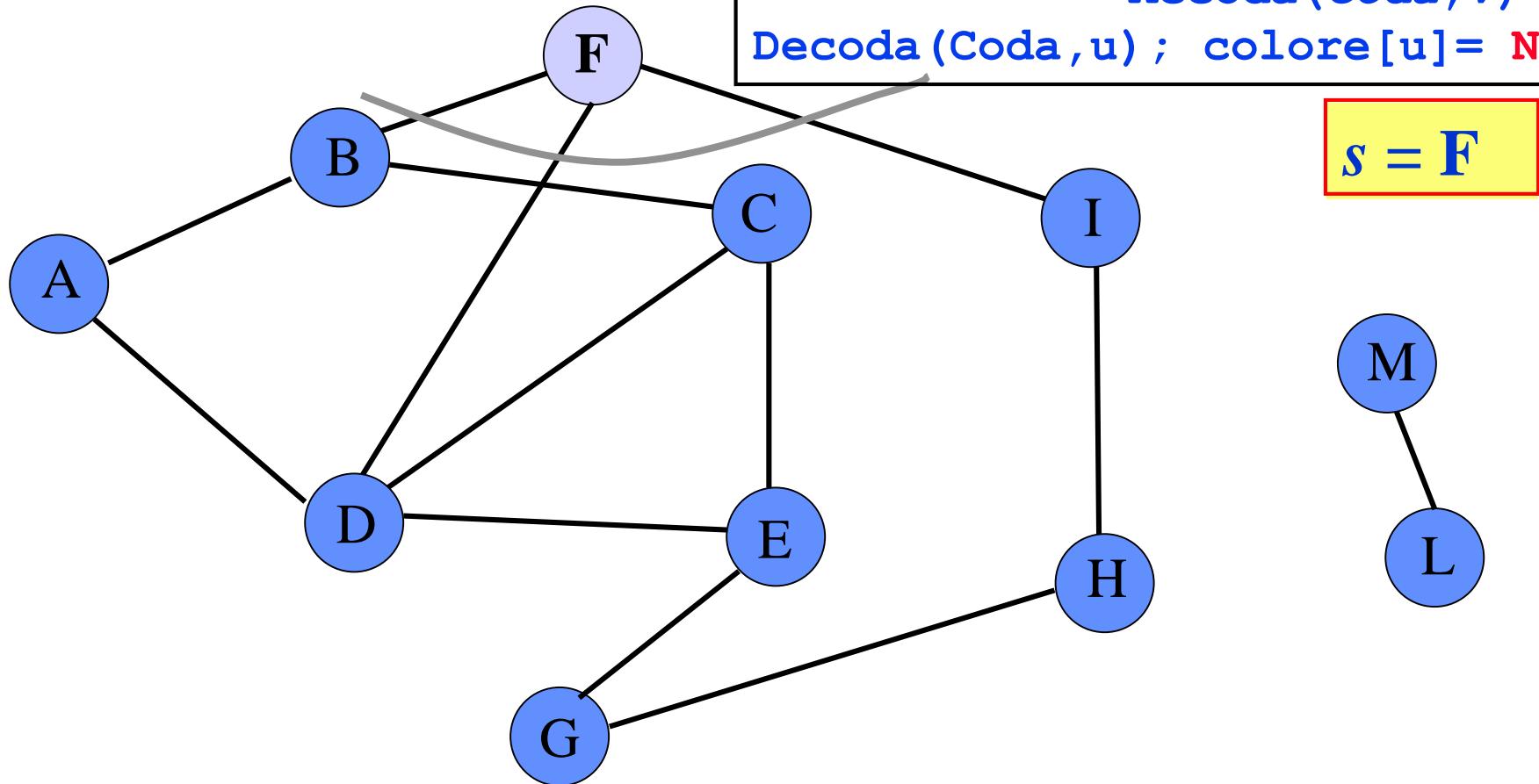
- Per distinguere tra i vertici non visitati, quelli visitati, e quelli processati coloreremo:
 - ogni vertice *scoperto* di grigio
 - ogni vertice *non scoperto* di bianco
 - ogni vertice *visitato* di nero

Algoritmo BFS II: soluzione

- Per distinguere tra i vertici non visitati, quelli visitati, e quelli processati coloreremo:
 - ogni vertice **scoperto** di grigio
 - ogni vertice **non scoperto** di bianco
 - ogni vertice **visitato** di nero
- Vengono **accodati** solo i vertici che **non sono ancora stati scoperti** (cioè **bianchi**)
- I vertici in **coda** saranno i vertici **scoperti** e **non ancora visitati** (cioè **grigi**)
- I vertici già **scoperti** o **visitati** non vengono più riconsiderati.

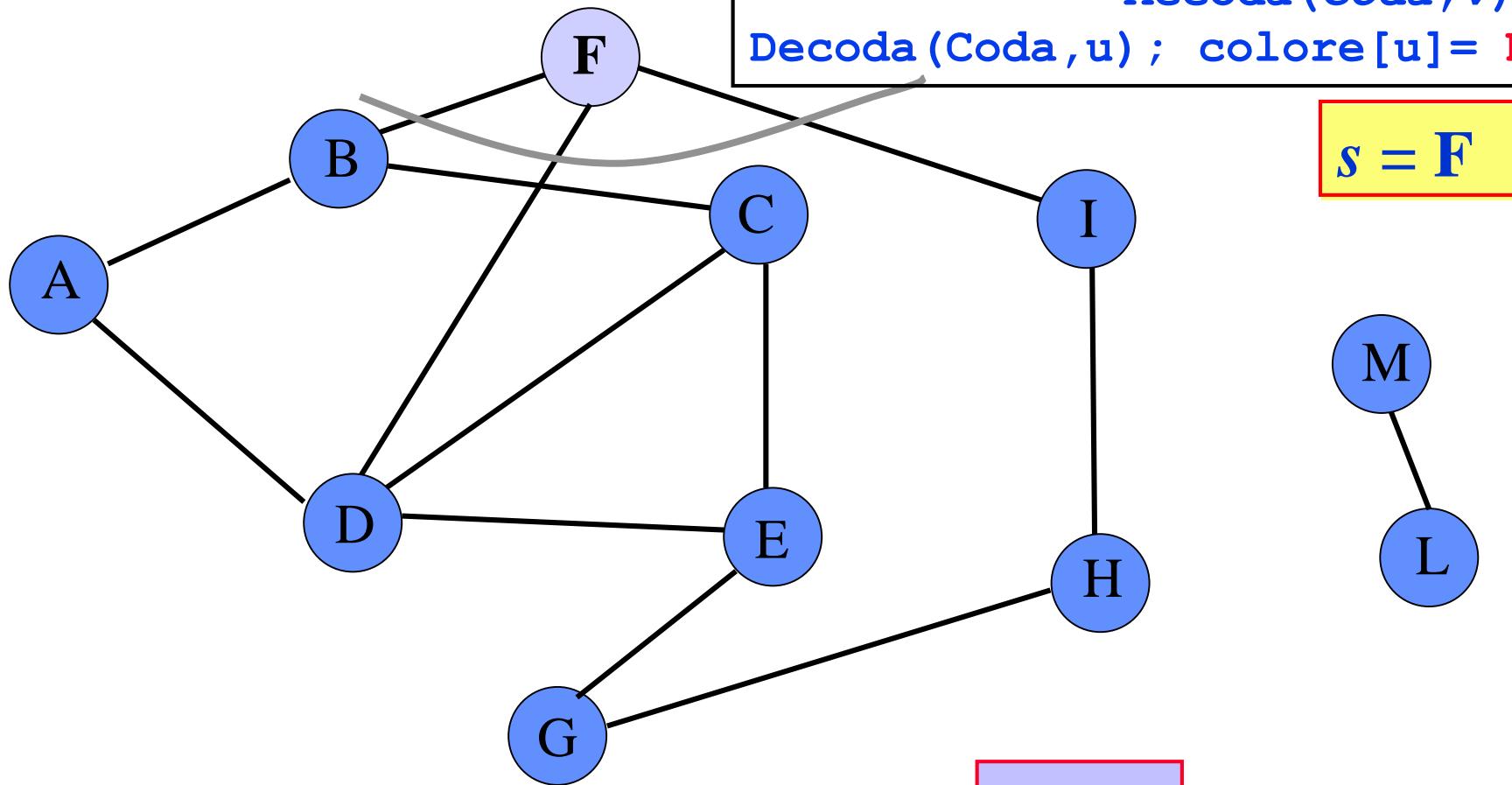
Algoritmo BFS II

```
for each v ∈ Adiac(u)
  do if colore[v] = Bianco
    then colore[v] = Grigio
        pred[v] = u
        Accoda(Coda, v)
Decoda(Coda, u) ; colore[u] = Nero
```

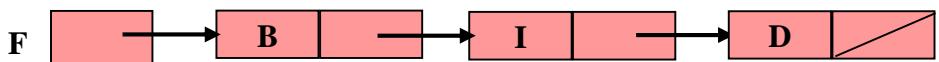


Algoritmo BFS II

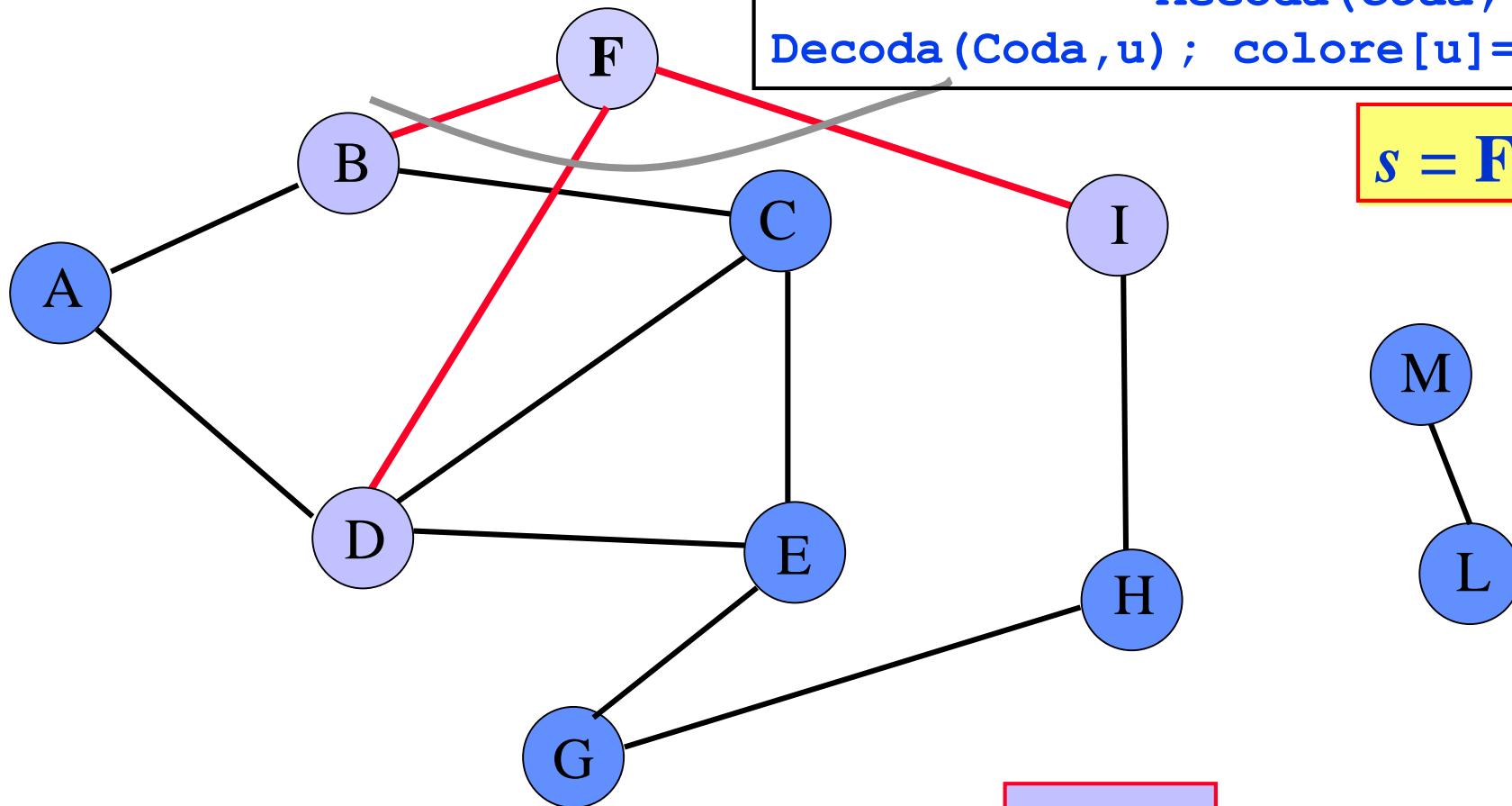
```
for each  $v \in \text{Adiac}(u)$ 
  do if  $\text{colore}[v] = \text{Bianco}$ 
    then  $\text{colore}[v] = \text{Grigio}$ 
       $\text{pred}[v] = u$ 
       $\text{Accoda}(\text{Coda}, v)$ 
  Decoda( $\text{Coda}, u$ ) ;  $\text{colore}[u] = \text{Nero}$ 
```



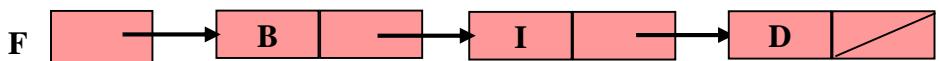
Coda : {F}



Algoritmo BFS II



Coda : {F, B, I, D}

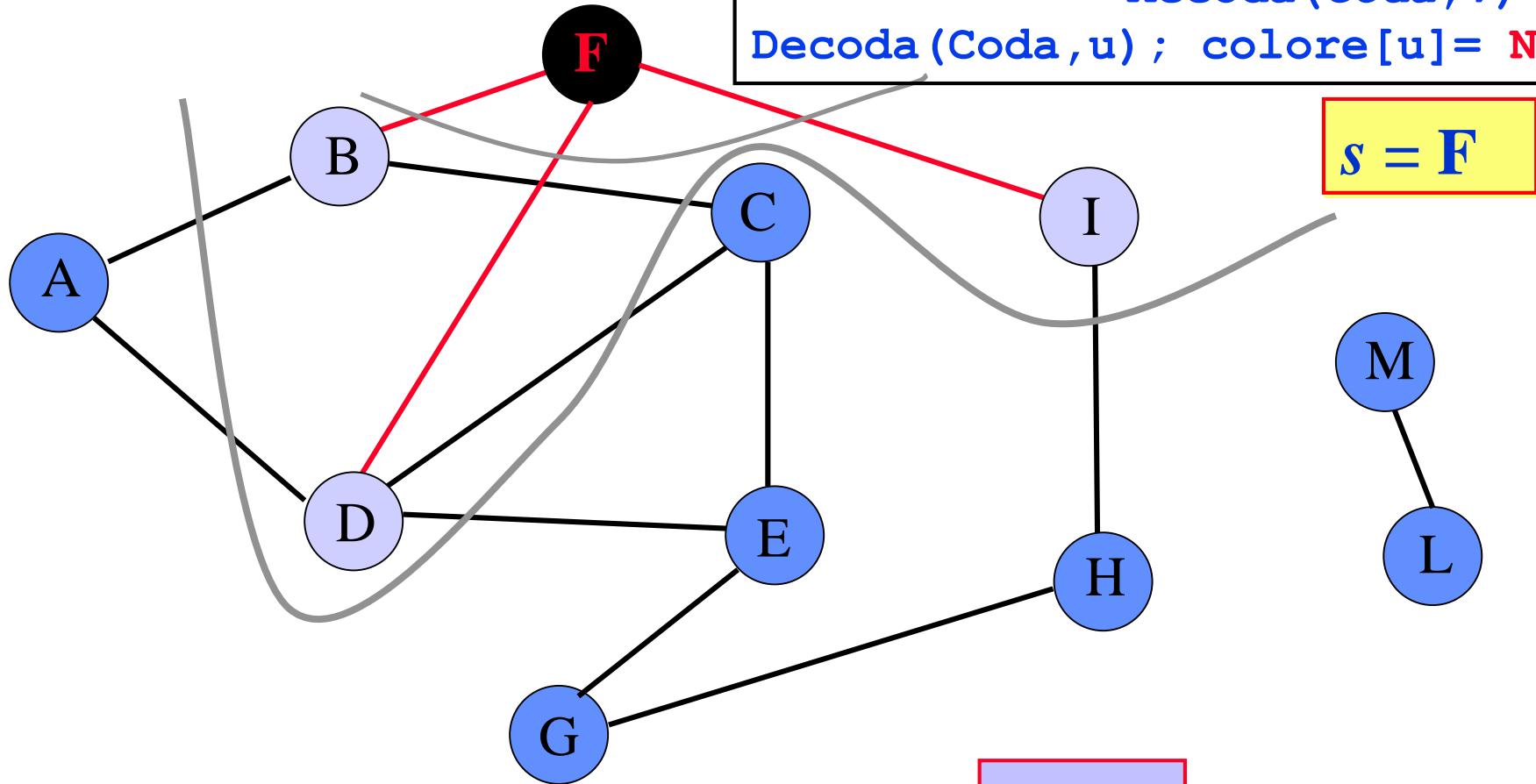


$u = F$

$s = F$

Algoritmo BFS II

```
for each  $v \in \text{Adiac}(u)$ 
do if  $\text{colore}[v] = \text{Bianco}$ 
then  $\text{colore}[v] = \text{Grigio}$ 
 $\text{pred}[v] = u$ 
 $\text{Accoda}(\text{Coda}, v)$ 
Decoda( $\text{Coda}, u$ ) ;  $\text{colore}[u] = \text{Nero}$ 
```

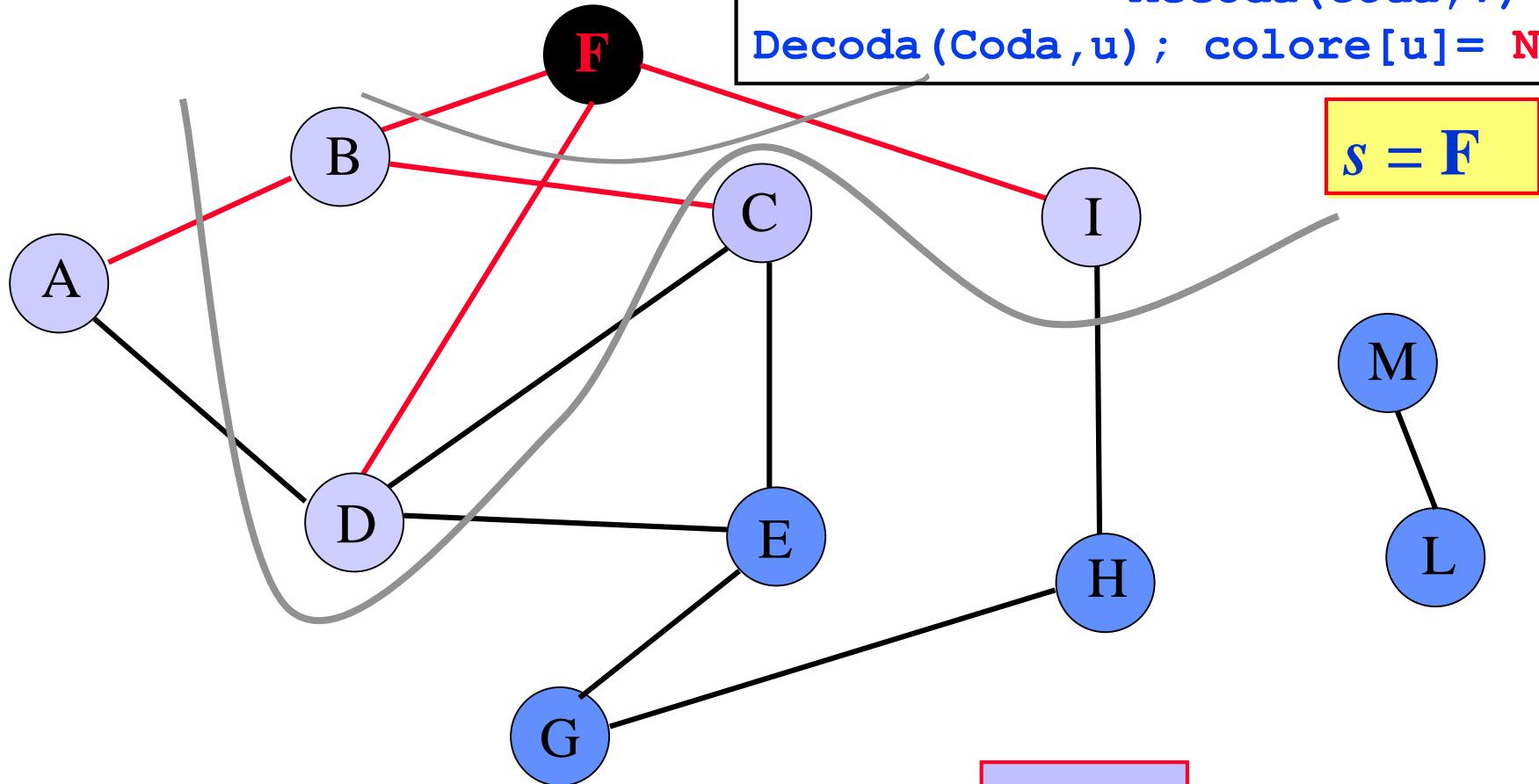


Coda : {B , I , D}

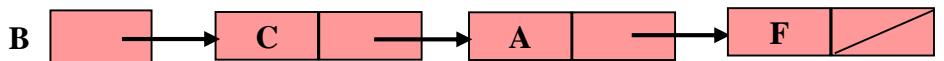


Algoritmo BFS II

```
for each  $v \in \text{Adiac}(u)$ 
  do if  $\text{colore}[v] = \text{Bianco}$ 
    then  $\text{colore}[v] = \text{Grigio}$ 
          $\text{pred}[v] = u$ 
          $\text{Accoda}(\text{Coda}, v)$ 
  Decoda( $\text{Coda}, u$ ) ;  $\text{colore}[u] = \text{Nero}$ 
```



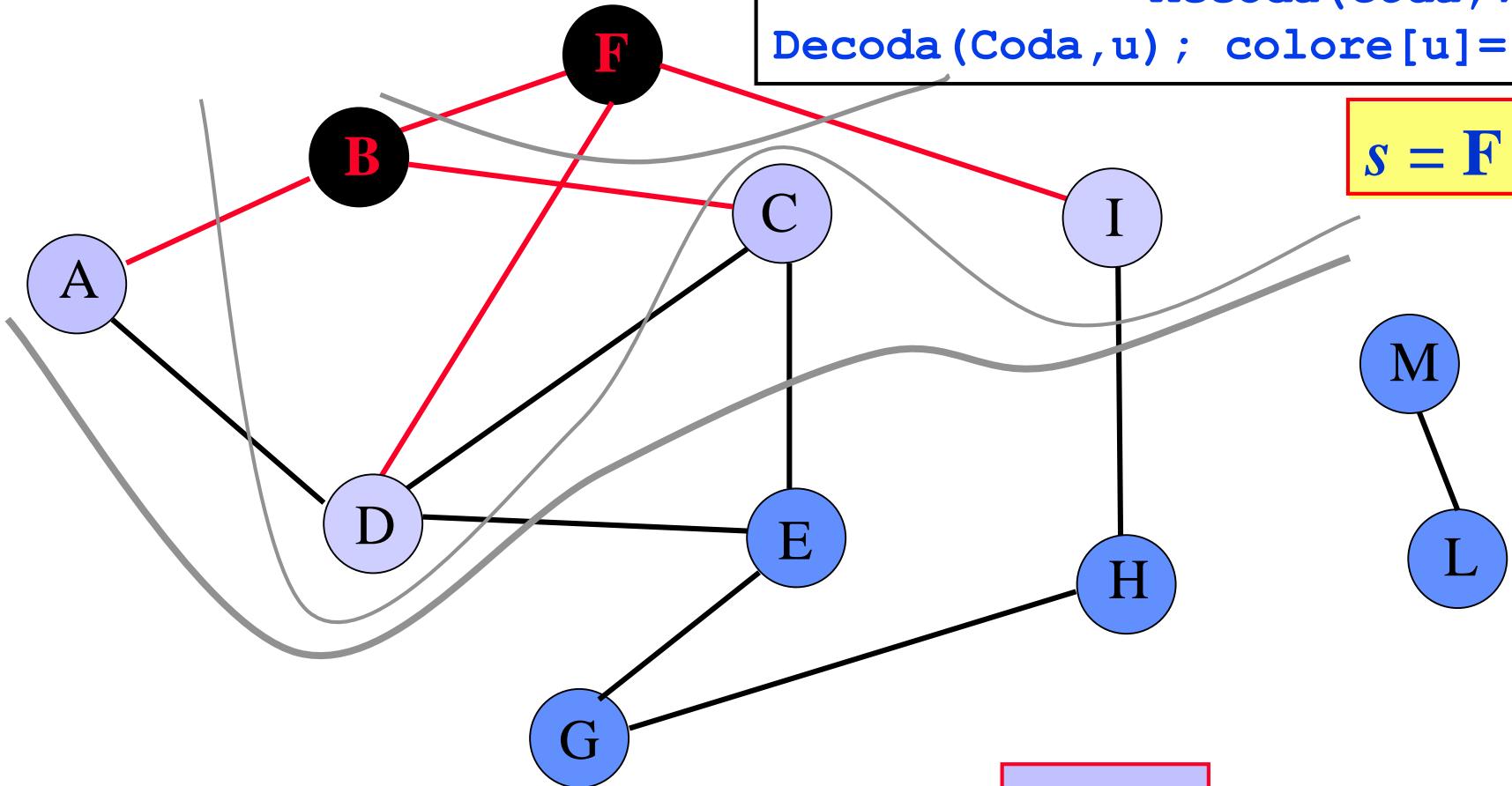
Coda : {B, I, D, C, A}



Algoritmo BFS II

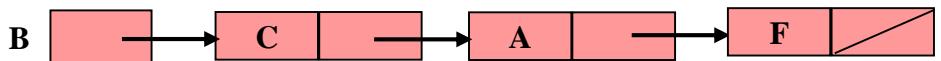
```
for each  $v \in \text{Adiac}(u)$ 
  do if  $\text{colore}[v] = \text{Bianco}$ 
    then  $\text{colore}[v] = \text{Grigio}$ 
       $\text{pred}[v] = u$ 
       $\text{Accoda}(\text{Coda}, v)$ 
  Decoda( $\text{Coda}, u$ ) ;  $\text{colore}[u] = \text{Nero}$ 
```

$s = F$



$u = B$

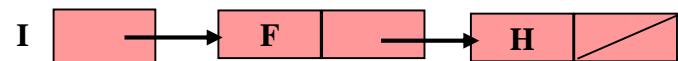
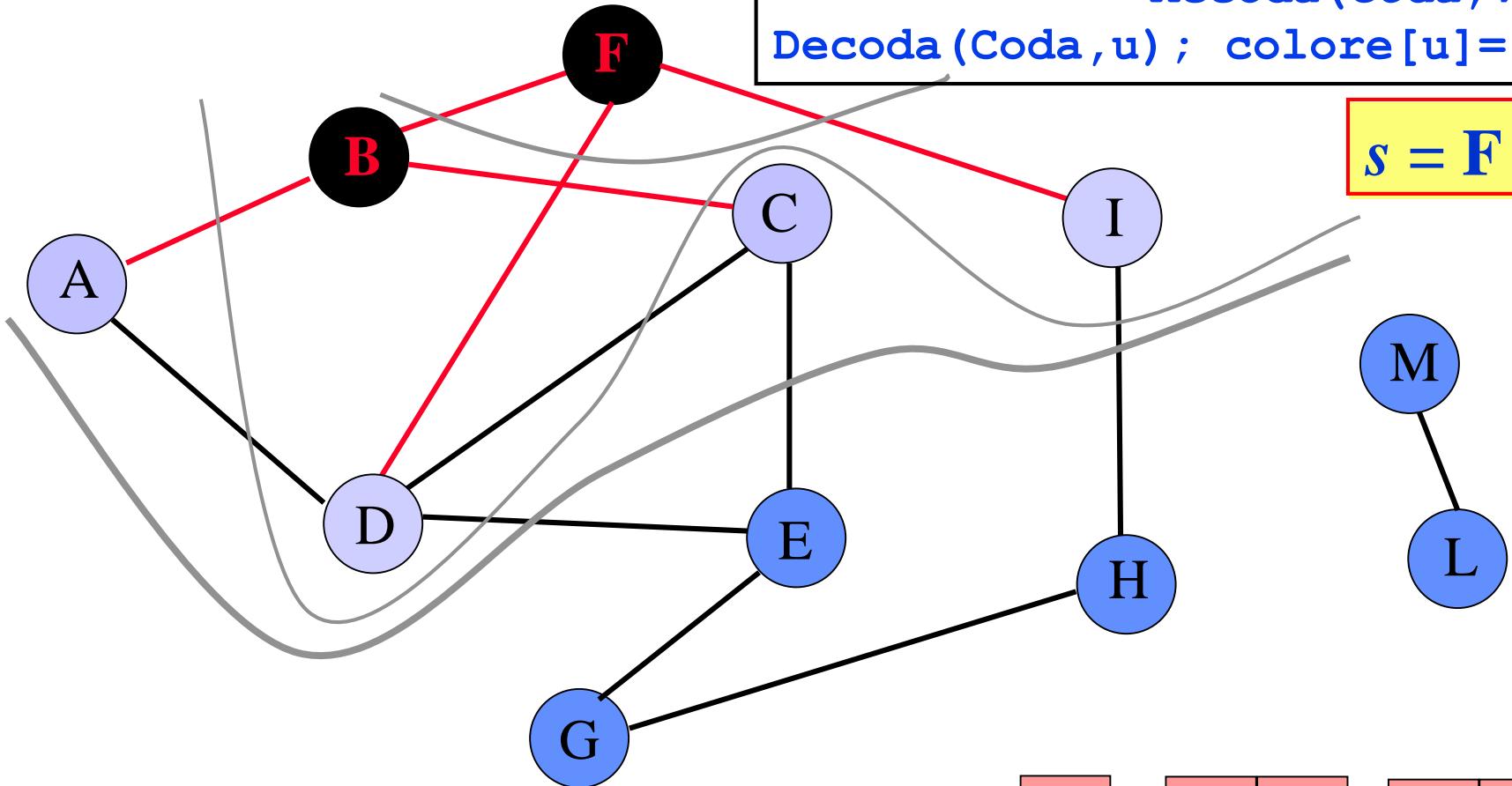
Coda : {B, I, D, C, A}



Algoritmo BFS II

```
for each  $v \in \text{Adiac}(u)$ 
do if  $\text{colore}[v] = \text{Bianco}$ 
then  $\text{colore}[v] = \text{Grigio}$ 
 $\text{pred}[v] = u$ 
 $\text{Accoda}(\text{Coda}, v)$ 
Decoda( $\text{Coda}, u$ ) ;  $\text{colore}[u] = \text{Nero}$ 
```

$s = F$



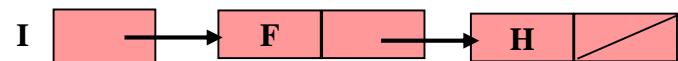
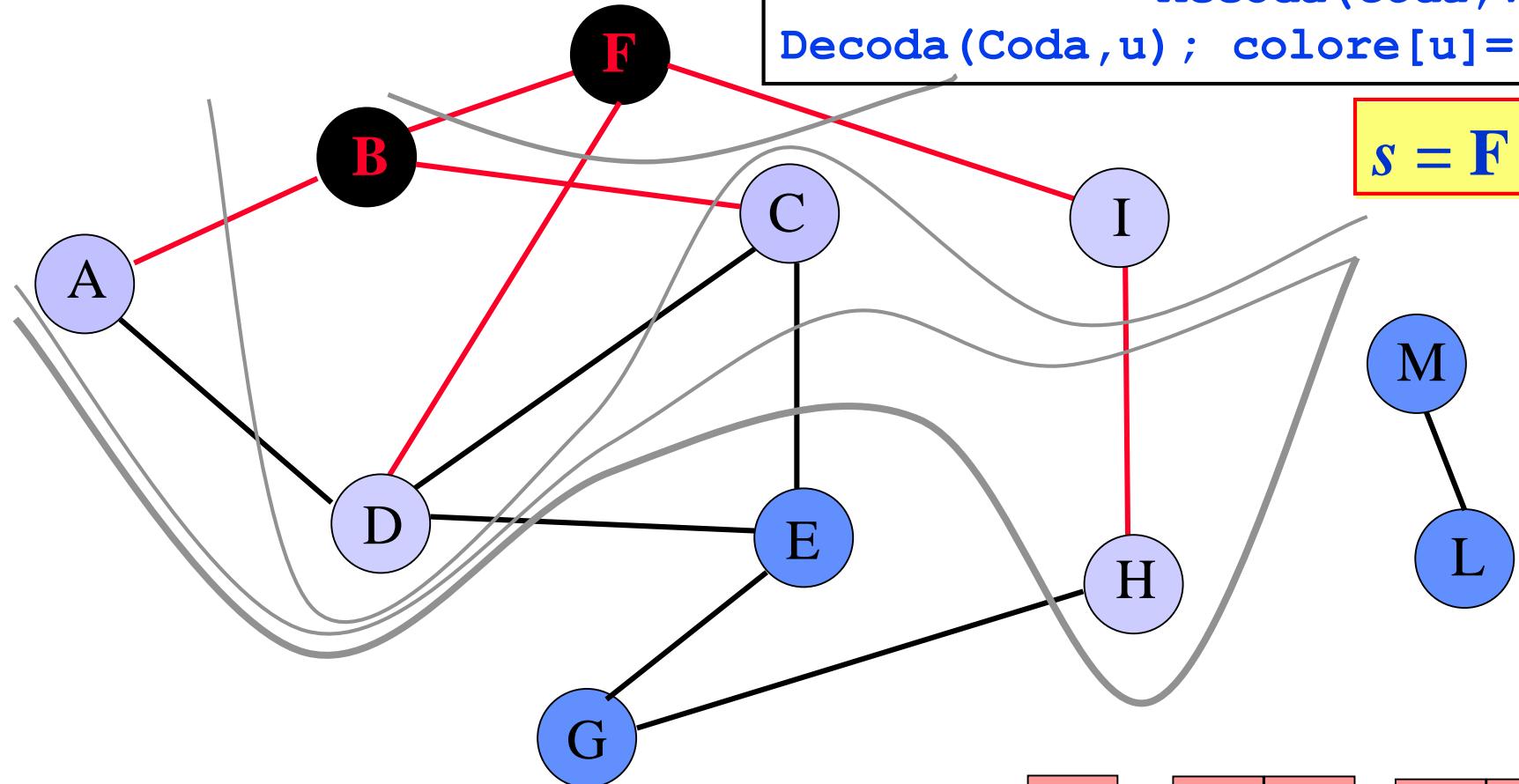
Coda: {I, D, C, A}

$u = I$

Algoritmo BFS II

```
for each v ∈ Adiac(u)
  do if colore[v] = Bianco
    then colore[v] = Grigio
        pred[v] = u
        Accoda(Coda, v)
Decoda(Coda, u) ; colore[u] = Nero
```

s = F

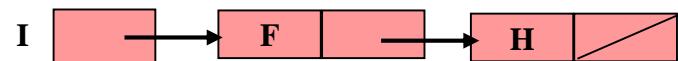
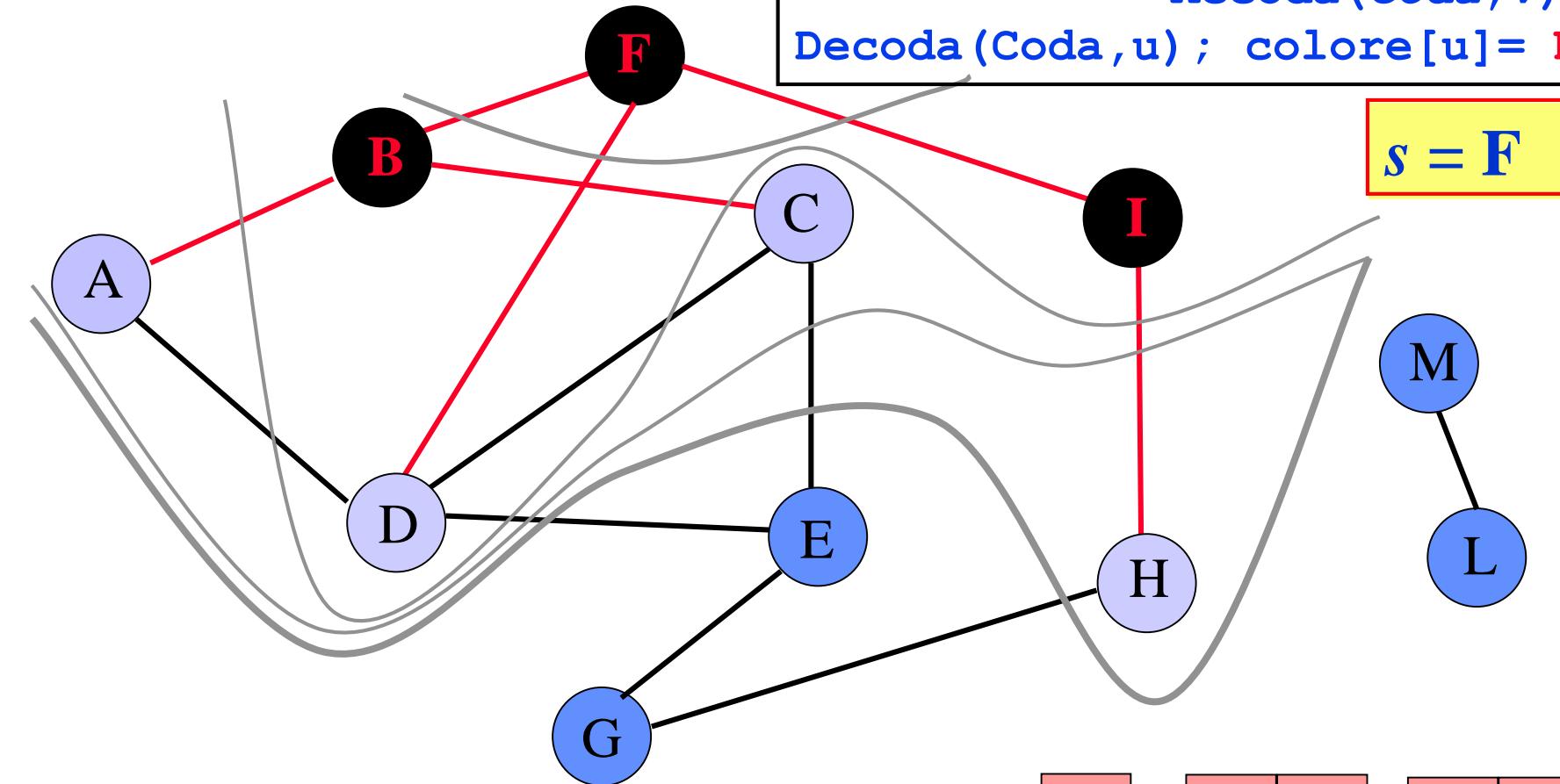


Coda: {I, D, C, A, H}

u = I

Algoritmo BFS II

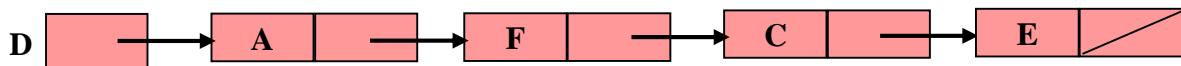
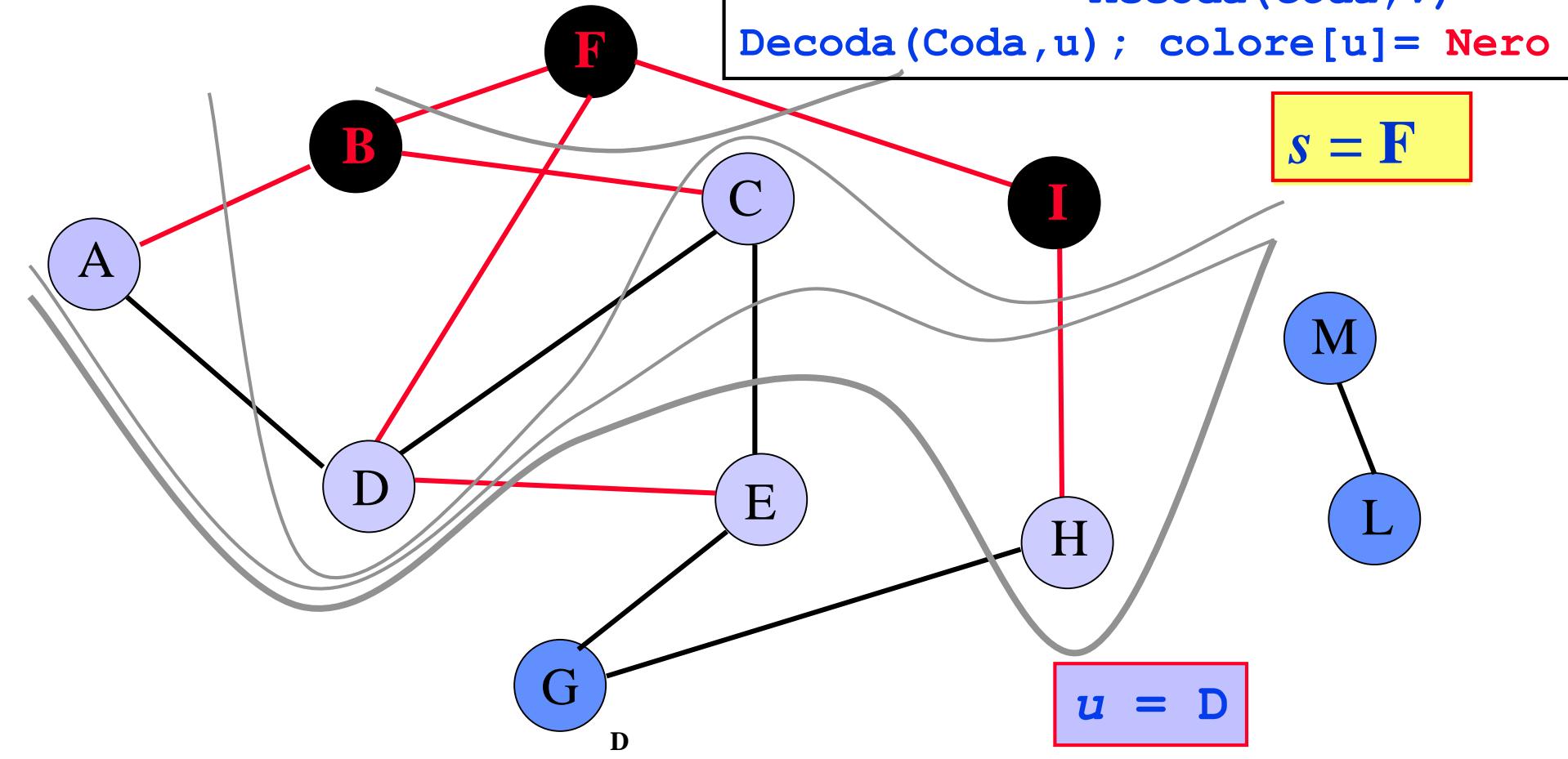
```
for each v ∈ Adiac(u)
  do if colore[v] = Bianco
    then colore[v] = Grigio
        pred[v] = u
        Accoda(Coda, v)
Decoda(Coda, u) ; colore[u] = Nero
```



Coda : {D, I, C, A, H}

Algoritmo BFS II

```
for each  $v \in \text{Adiac}(u)$ 
do if  $\text{colore}[v] = \text{Bianco}$ 
then  $\text{colore}[v] = \text{Grigio}$ 
 $\text{pred}[v] = u$ 
 $\text{Accoda}(\text{Coda}, v)$ 
Decoda( $\text{Coda}, u$ ) ;  $\text{colore}[u] = \text{Nero}$ 
```

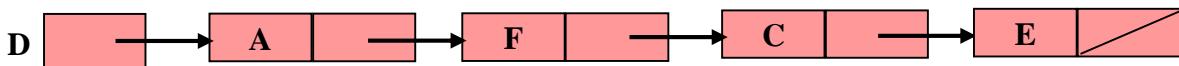
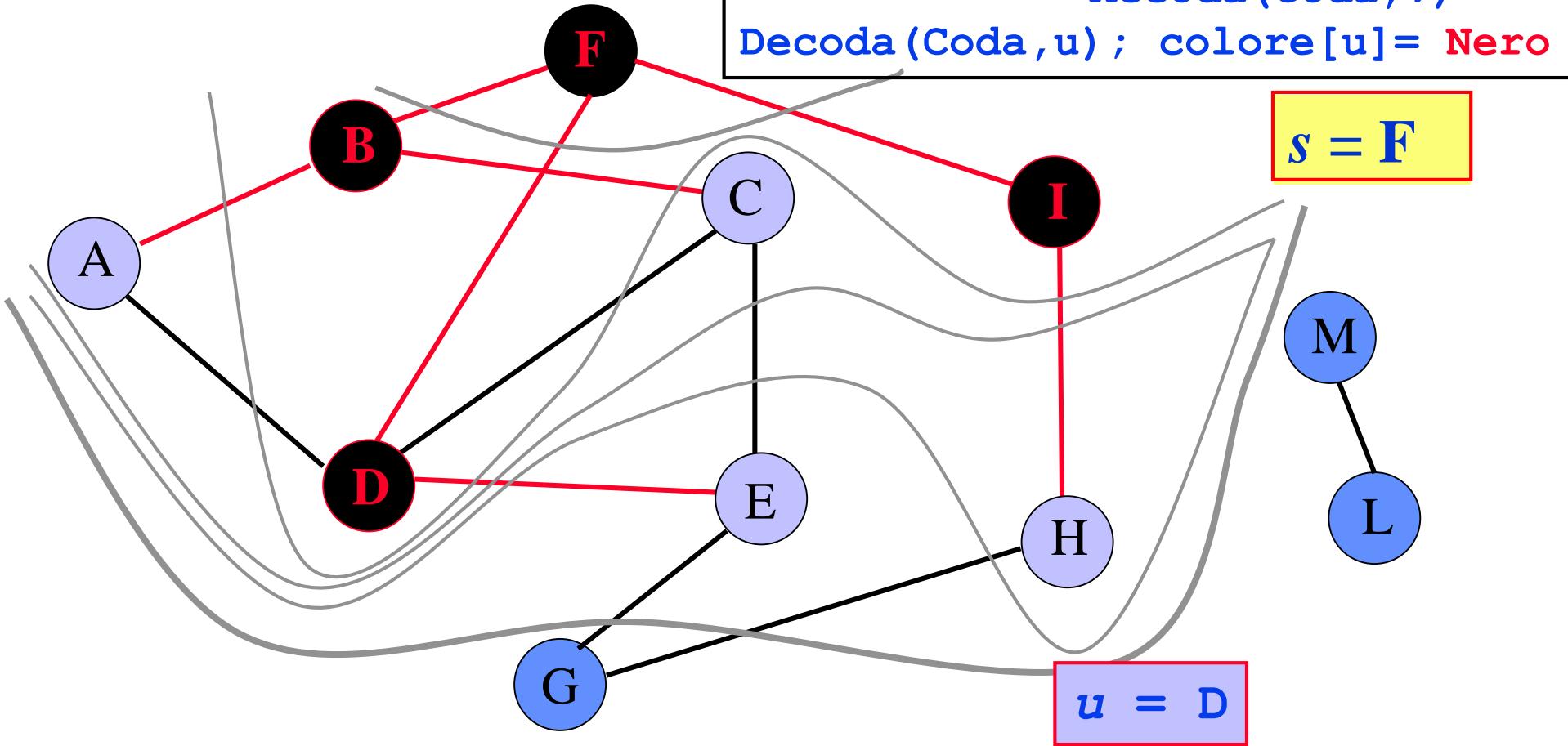


Algoritmo BFS II

```

for each  $v \in \text{Adiac}(u)$ 
do if  $\text{colore}[v] = \text{Bianco}$ 
then  $\text{colore}[v] = \text{Grigio}$ 
       $\text{pred}[v] = u$ 
       $\text{Accoda}(\text{Coda}, v)$ 
Decoda( $\text{Coda}, u$ ) ;  $\text{colore}[u] = \text{Nero}$ 

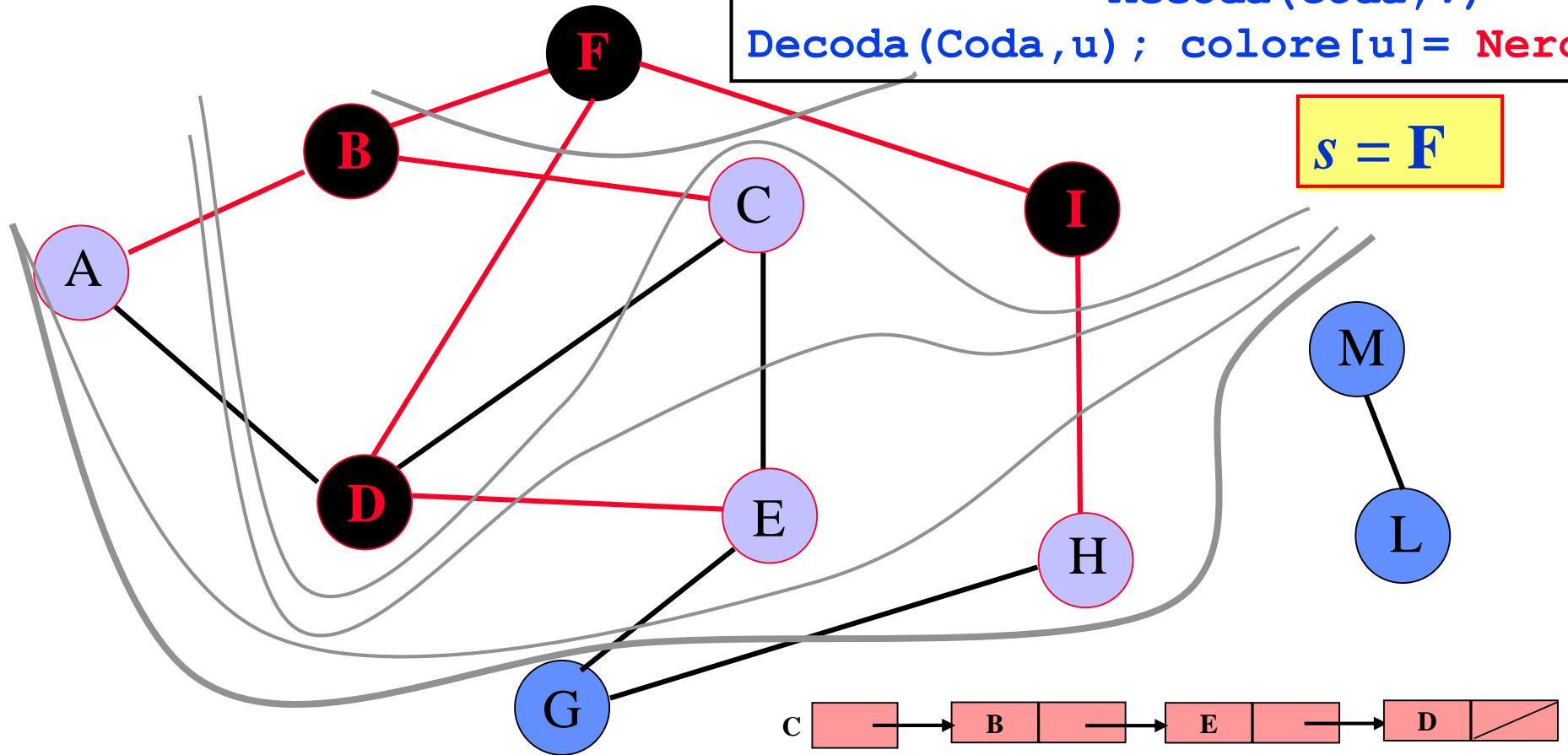
```



Algoritmo BFS II

```
for each  $v \in \text{Adiac}(u)$ 
  do if  $\text{colore}[v] = \text{Bianco}$ 
    then  $\text{colore}[v] = \text{Grigio}$ 
          $\text{pred}[v] = u$ 
          $\text{Accoda}(\text{Coda}, v)$ 
  Decoda( $\text{Coda}, u$ ) ;  $\text{colore}[u] = \text{Nero}$ 
```

$s = F$



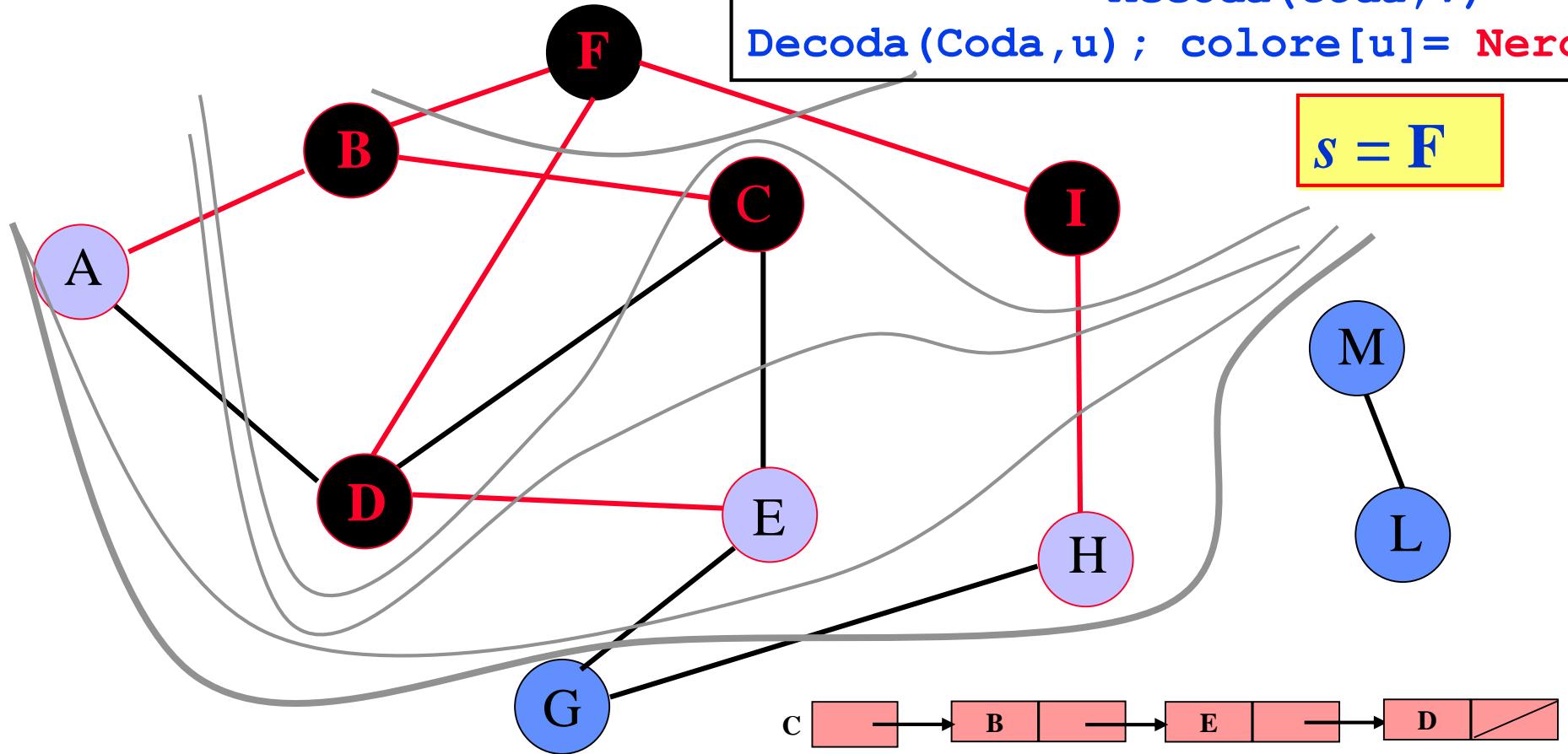
$\text{Coda} : \{C, A, H, E\}$

$u = C$

Algoritmo BFS II

```
for each  $v \in \text{Adiac}(u)$ 
do if  $\text{colore}[v] = \text{Bianco}$ 
then  $\text{colore}[v] = \text{Grigio}$ 
 $\text{pred}[v] = u$ 
 $\text{Accoda}(\text{Coda}, v)$ 
Decoda( $\text{Coda}, u$ ) ;  $\text{colore}[u] = \text{Nero}$ 
```

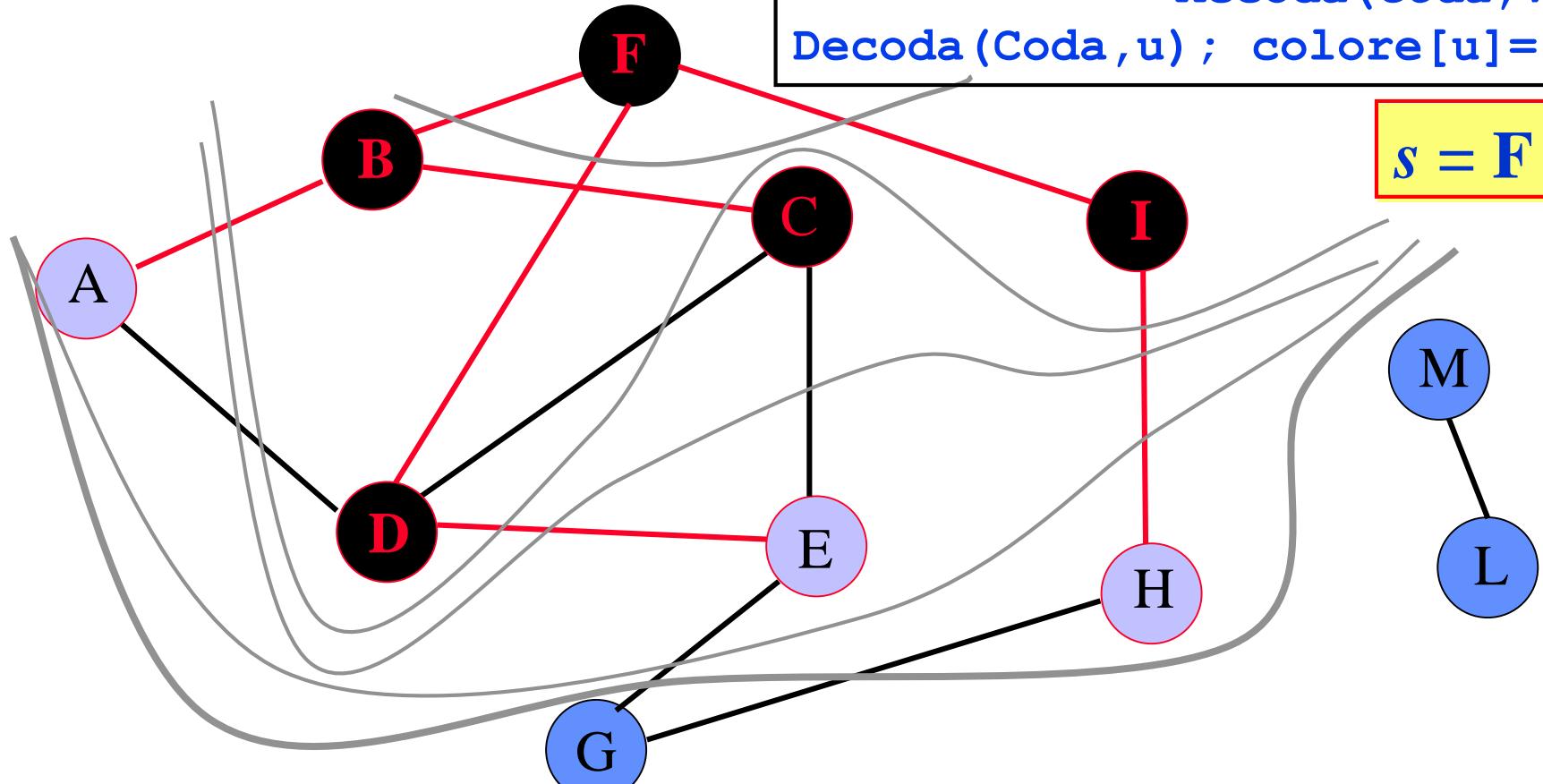
$s = F$



Algoritmo BFS II

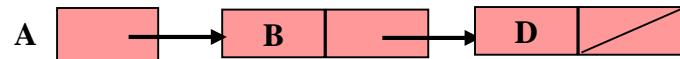
```
for each  $v \in \text{Adiac}(u)$ 
do if  $\text{colore}[v] = \text{Bianco}$ 
then  $\text{colore}[v] = \text{Grigio}$ 
 $\text{pred}[v] = u$ 
 $\text{Accoda}(\text{Coda}, v)$ 
Decoda( $\text{Coda}, u$ ) ;  $\text{colore}[u] = \text{Nero}$ 
```

$s = F$



Coda : {A, H, E, G}

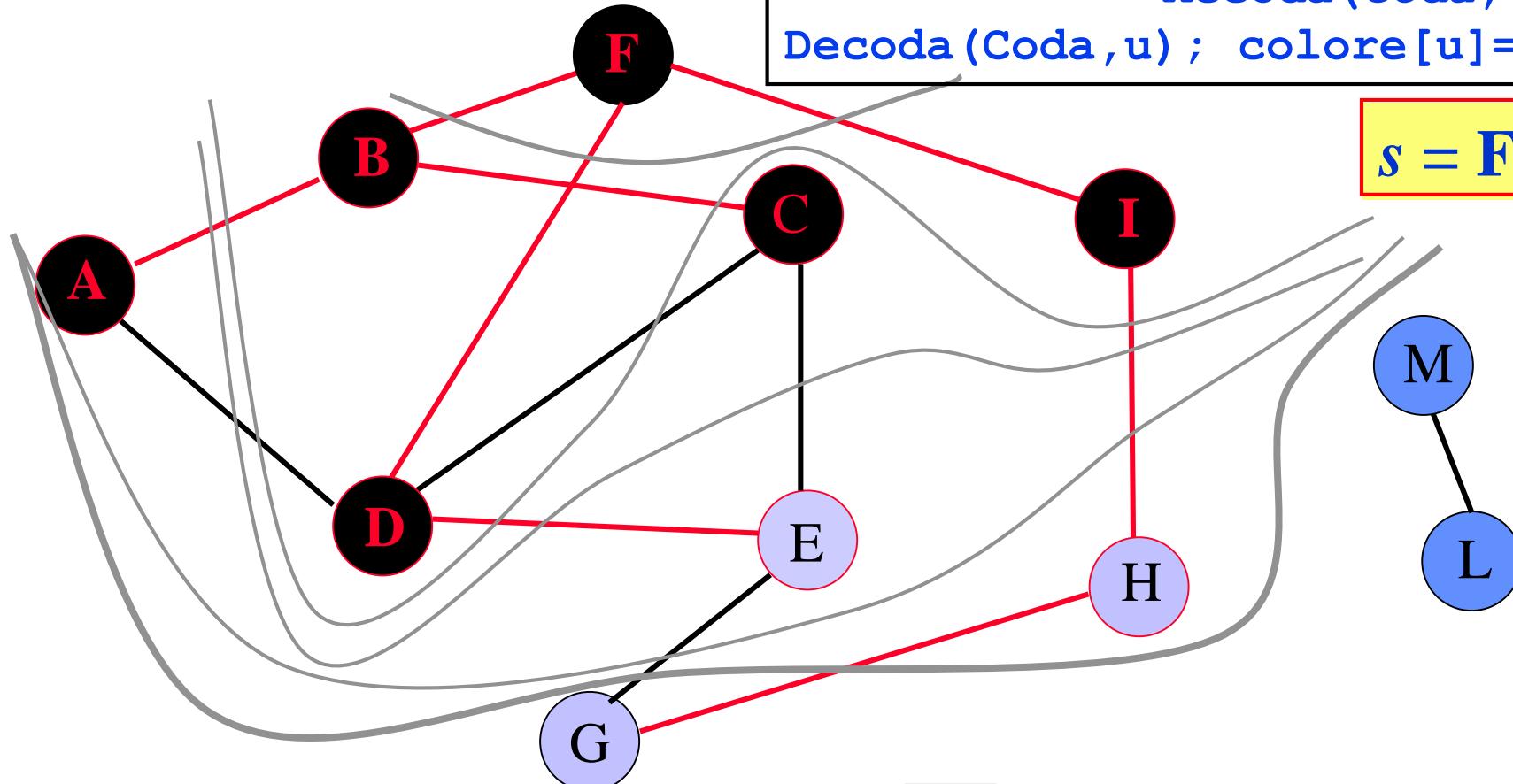
$u = A$



Algoritmo BFS II

```
for each  $v \in \text{Adiac}(u)$ 
do if  $\text{colore}[v] = \text{Bianco}$ 
then  $\text{colore}[v] = \text{Grigio}$ 
 $\text{pred}[v] = u$ 
 $\text{Accoda}(\text{Coda}, v)$ 
Decoda( $\text{Coda}, u$ ) ;  $\text{colore}[u] = \text{Nero}$ 
```

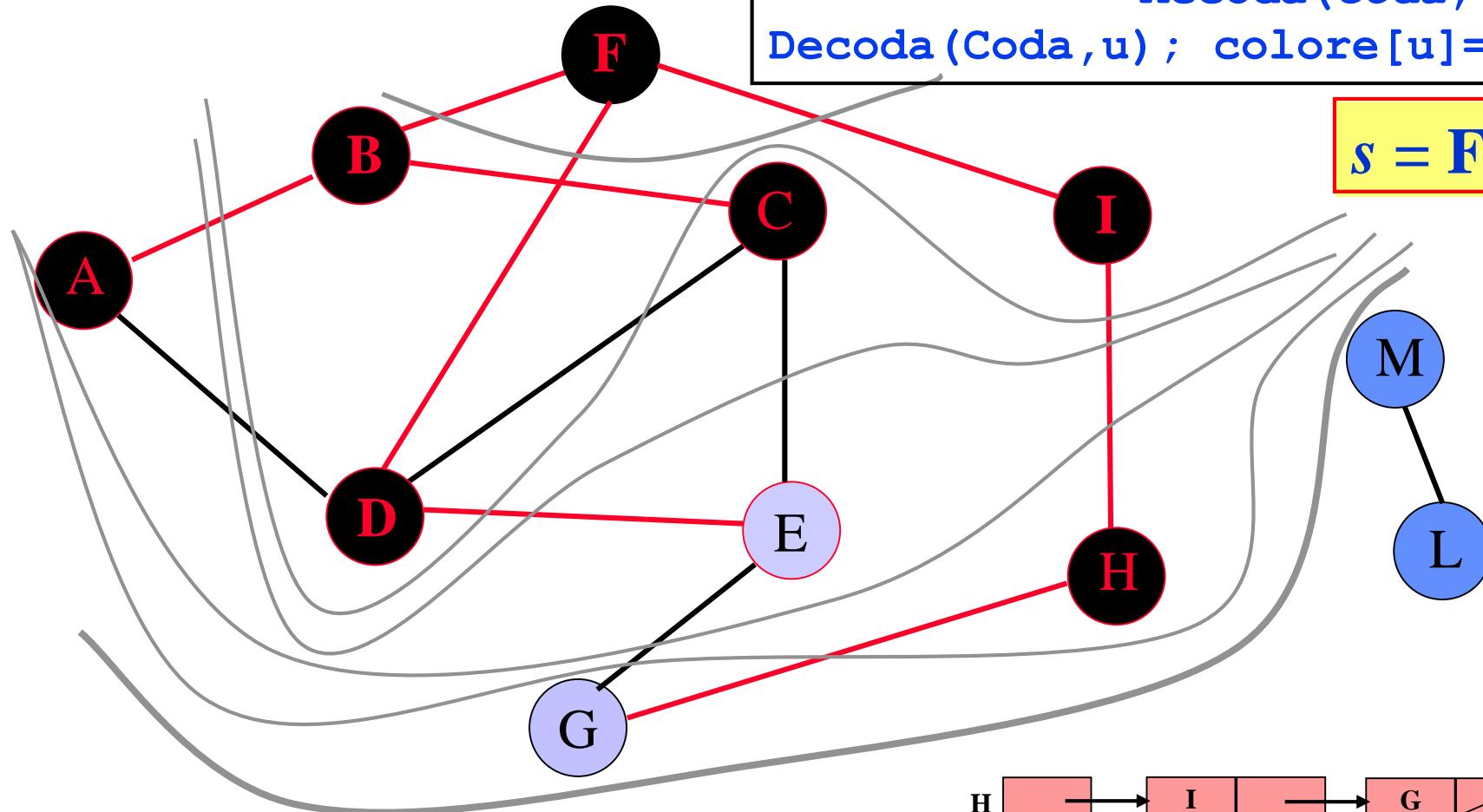
$s = F$



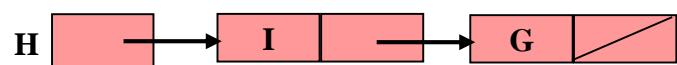
Algoritmo BFS II

```
for each  $v \in \text{Adiac}(u)$ 
do if  $\text{colore}[v] = \text{Bianco}$ 
then  $\text{colore}[v] = \text{Grigio}$ 
 $\text{pred}[v] = u$ 
 $\text{Accoda}(\text{Coda}, v)$ 
Decoda( $\text{Coda}, u$ ) ;  $\text{colore}[u] = \text{Nero}$ 
```

$s = F$



Coda : {E, G}

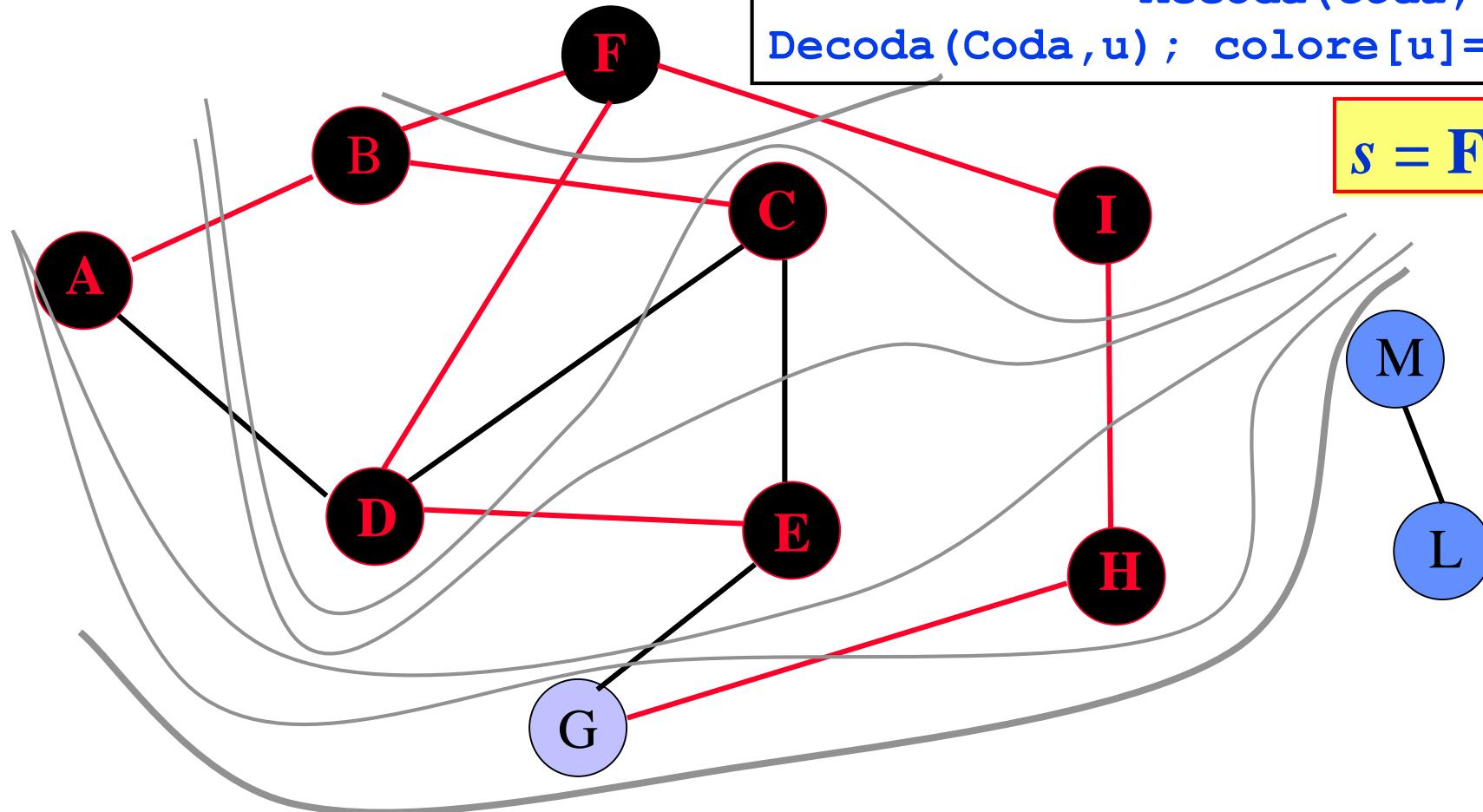


$u = H$

Algoritmo BFS II

```
for each  $v \in \text{Adiac}(u)$ 
do if  $\text{colore}[v] = \text{Bianco}$ 
then  $\text{colore}[v] = \text{Grigio}$ 
 $\text{pred}[v] = u$ 
 $\text{Accoda}(\text{Coda}, v)$ 
Decoda( $\text{Coda}, u$ ) ;  $\text{colore}[u] = \text{Nero}$ 
```

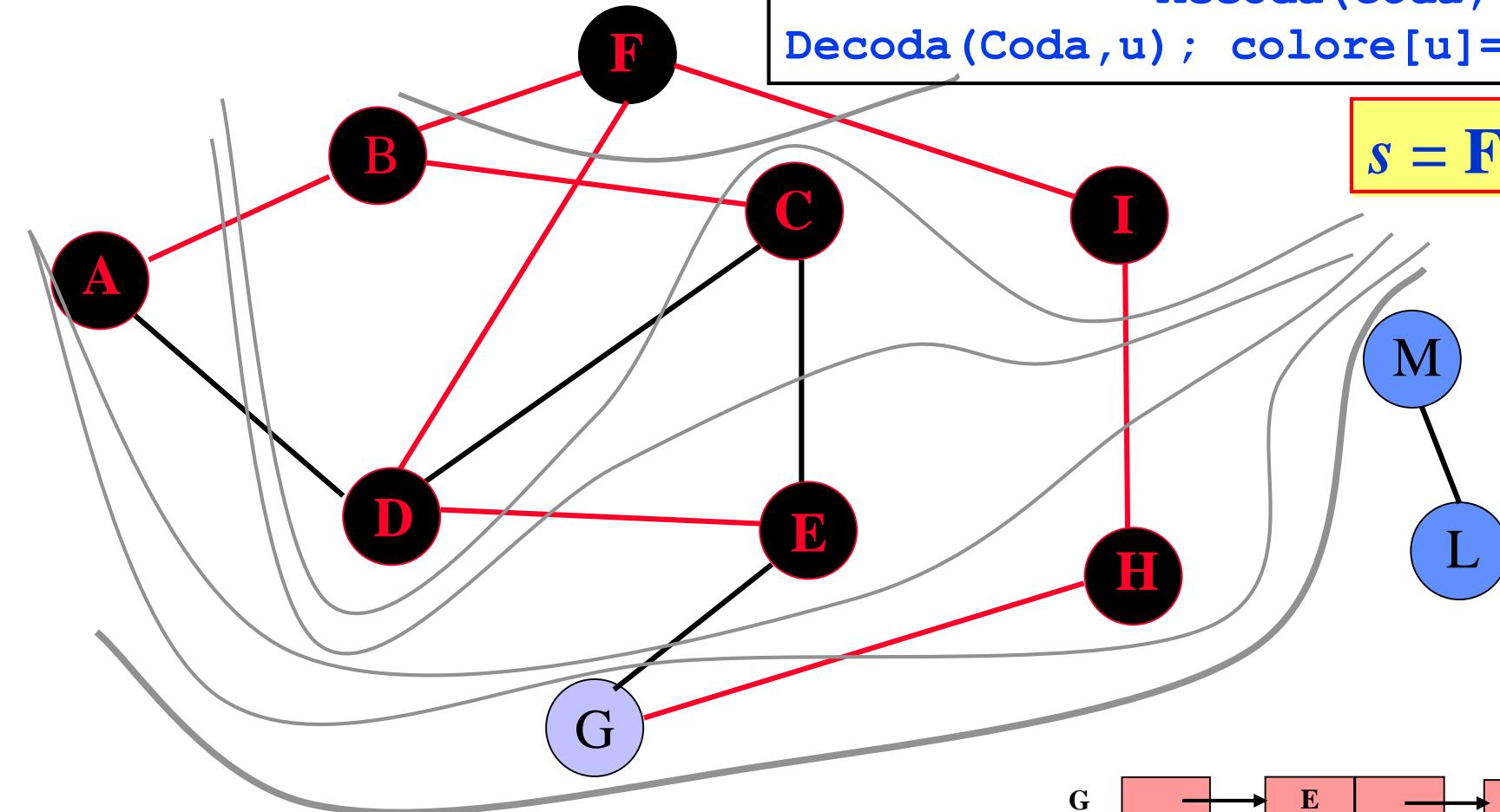
$s = F$



Algoritmo BFS II

```
for each  $v \in \text{Adiac}(u)$ 
do if  $\text{colore}[v] = \text{Bianco}$ 
then  $\text{colore}[v] = \text{Grigio}$ 
 $\text{pred}[v] = u$ 
 $\text{Accoda}(\text{Coda}, v)$ 
Decoda( $\text{Coda}, u$ ) ;  $\text{colore}[u] = \text{Nero}$ 
```

$s = F$

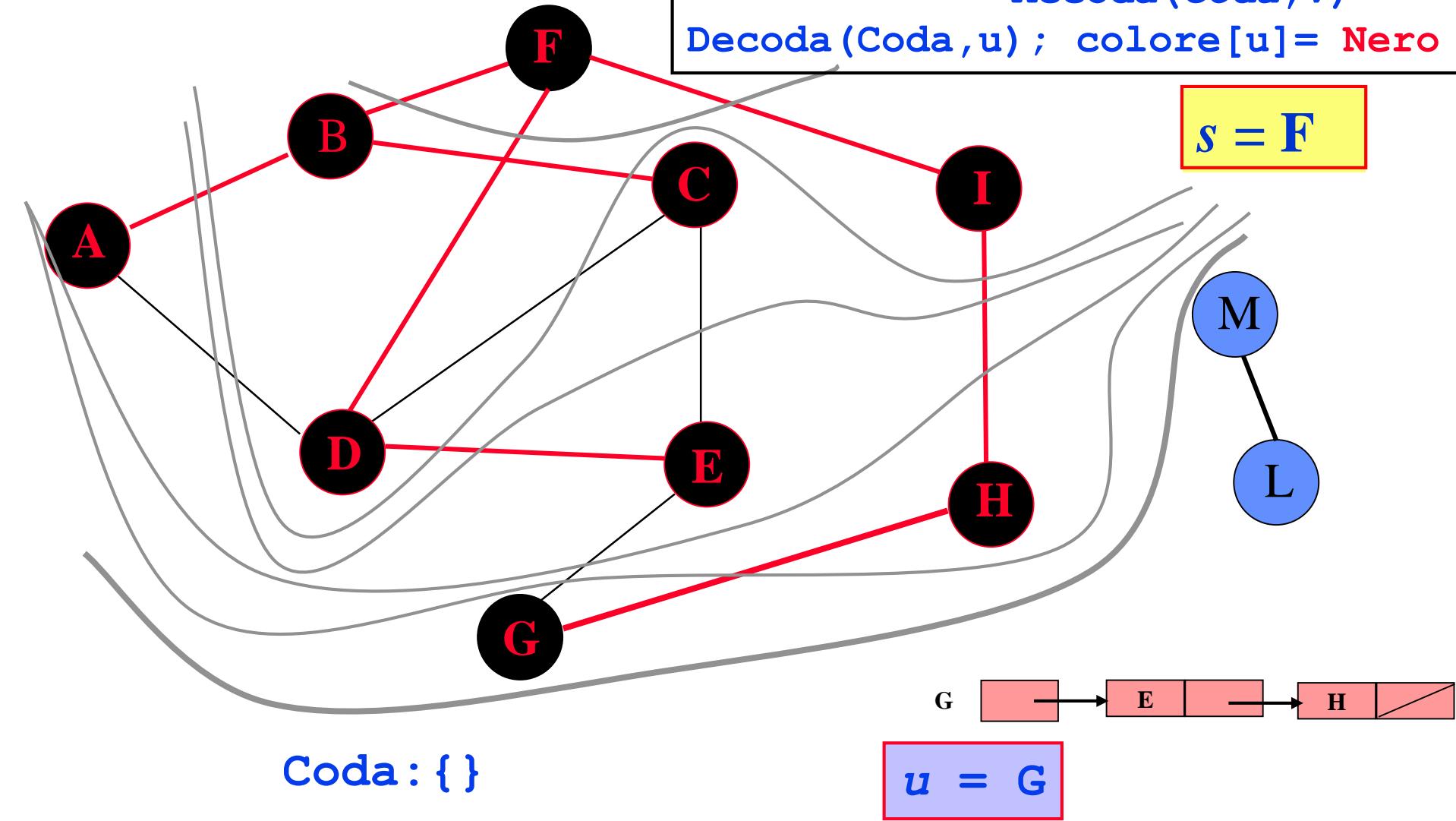


Coda : {G}

$u = G$

Algoritmo BFS II

```
for each  $v \in \text{Adiac}(u)$ 
do if  $\text{colore}[v] = \text{Bianco}$ 
then  $\text{colore}[v] = \text{Grigio}$ 
 $\text{pred}[v] = u$ 
 $\text{Accoda}(\text{Coda}, v)$ 
Decoda( $\text{Coda}, u$ ) ;  $\text{colore}[u] = \text{Nero}$ 
```



Algoritmo BFS II

BSF (G:grafo, s:vertice)

```
for each vertice  $u \in V(G) - \{s\}$  do
    colore[u] = Bianco
    pred[u] = Nil
colore[s] = Grigio
pred[s] = Nil
Coda = {s}
```

```
while Coda  $\neq \emptyset$  do
    u = Testa[Coda]
    for each  $v \in \text{Adiac}(u)$  do
```

```
        if colore[v] = Bianco then
            colore[v] = Grigio
            pred[v] = u
            Accoda(Coda, v)
```

```
Decoda(Coda)
colore[u] = Nero
```

Inizializzazione

Accodamento
dei soli nodi
non visitati

Algoritmo BFS II: complessità

BSF (G:grafo, s:vertice)

```
for each vertice  $u \in V(G) - \{s\}$ 
```

```
do colore[u] = Bianco
```

```
pred[u] = Nil
```

```
colore[s] = Grigio
```

```
pred[s] = Nil
```

```
Coda = {s}
```

```
while Coda  $\neq \emptyset$ 
```

```
do  $u = \text{Testa}[Coda]$ 
```

```
for each  $v \in \text{Adiac}(u)$ 
```

```
do if colore[v] = Bianco
```

```
then colore[v] = Grigio
```

```
pred[v] = u
```

```
Accoda(Coda, v)
```

```
Decoda(Coda)
```

```
colore[u] = Nero
```

$O(|V|)$

$O(|E_u|)$

E_u = lunghezza
della lista di
adiacenza di u

Algoritmo BFS II: complessità

BSF (G:grafo, s:vertice)

```
for each vertice  $u \in V(G) - \{s\}$ 
```

```
do colore[u] = Bianco
```

```
pred[u] = Nil
```

```
colore[s] = Grigio
```

```
pred[s] = Nil
```

```
Coda = {s}
```

```
while Coda  $\neq \emptyset$ 
```

```
do  $u = \text{Testa}[Coda]$ 
```

```
for each  $v \in \text{Adiac}(u)$ 
```

```
do if colore[v] = Bianco
```

```
then colore[v] = Grigio
```

```
pred[v] = u
```

```
Accoda(Coda, v)
```

```
Decoda(Coda)
```

```
colore[u] = Nero
```

$O(|V|)$

$O(|E|)$

E = dimensione delle liste di adiacenza.
Numero di archi

Algoritmo BFS II: complessità

L'algoritmo di visita in *Breadth-First* impiega **tempo** proporzionale alla **somma** del **numero di vertici** e del **numero di archi** (dimensione delle liste di adiacenza).

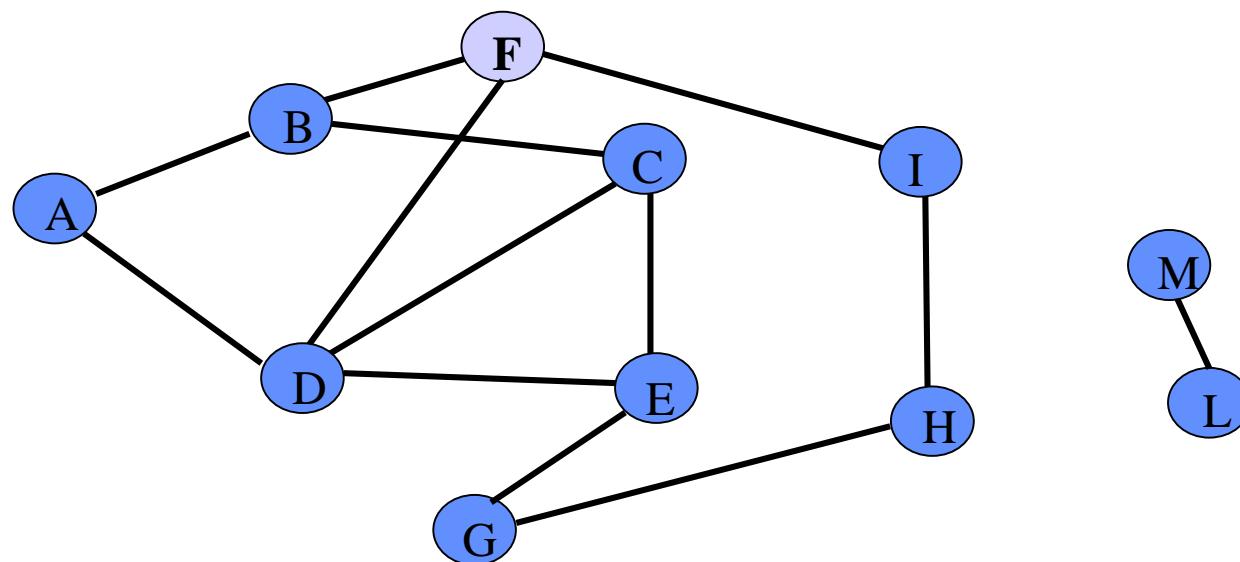
$$T(V, E) = O(|V| + |E|)$$

Sottografo dei predecessori e BFS

- L'*algoritmo BFS* eseguito su un grafo $G = \langle V, E \rangle$ costruisce (nell'array $\text{pred}[]$) il *sottografo dei predecessori* denotato con $G_{\text{pred}} = \langle V_{\text{pred}}, E_{\text{pred}} \rangle$, dove:

$$V_{\text{pred}} = \{ v \in V : \text{pred}[v] \neq \text{Nil} \} \cup \{ s \}$$

$$E_{\text{pred}} = \{ (\text{pred}[v], v) \in E : v \in V_{\text{pred}} - \{ s \} \}$$

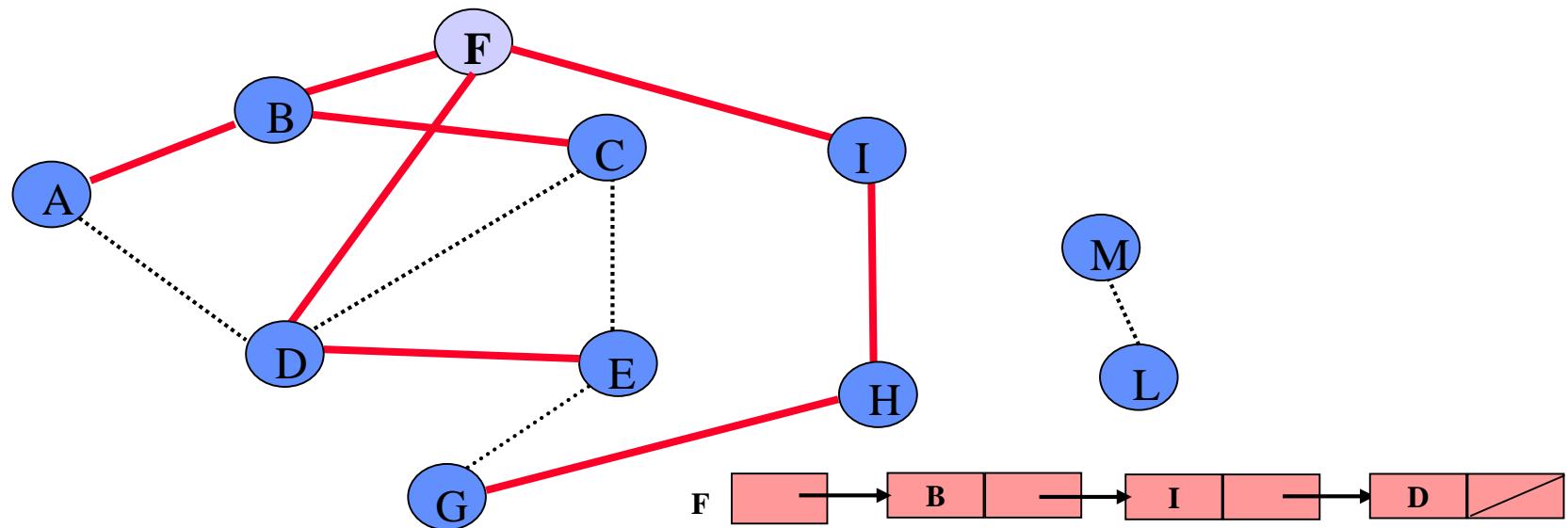


Sottografo dei predecessori e BFS

- L'algorithm *BFS* eseguito su un grafo $G = \langle V, E \rangle$ costruisce (nell'array *pred[]*) il *sottografo dei predecessori* denotato con $G_{pred} = \langle V_{pred}, E_{pred} \rangle$, dove:

$$V_{pred} = \{ v \in V : pred[v] \neq \text{Nil} \} \cup \{ s \}$$

$$E_{pred} = \{ (pred[v], v) \in E : v \in V_{pred} - \{ s \} \}$$

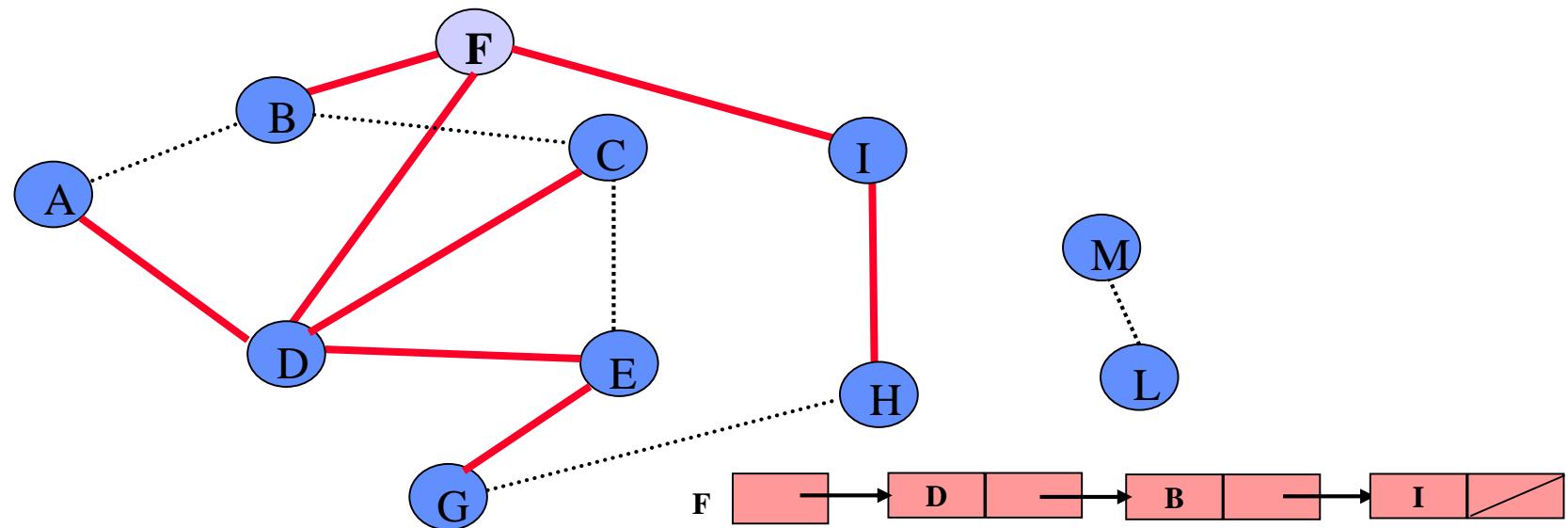


Sottografo dei predecessori e BFS

- L'algorithm *BFS* eseguito su un grafo $G = \langle V, E \rangle$ costruisce (nell'array *pred[]*) il *sottografo dei predecessori* denotato con $G_{pred} = \langle V_{pred}, E_{pred} \rangle$, dove:

$$V_{pred} = \{ v \in V : pred[v] \neq \text{Nil} \} \cup \{ s \}$$

$$E_{pred} = \{ (pred[v], v) \in E : v \in V_{pred} - \{ s \} \}$$

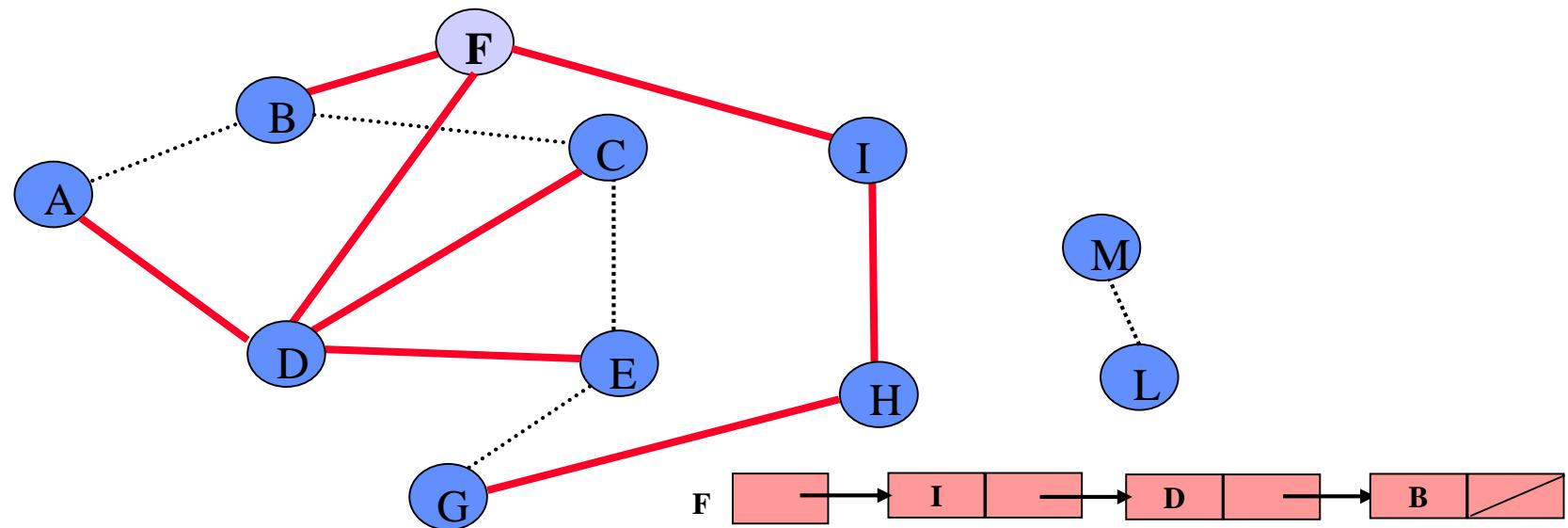


Sottografo dei predecessori e BFS

- L'algorithm *BFS* eseguito su un grafo $G = \langle V, E \rangle$ costruisce (nell'array *pred[]*) il *sottografo dei predecessori* denotato con $G_{pred} = \langle V_{pred}, E_{pred} \rangle$, dove:

$$V_{pred} = \{ v \in V : pred[v] \neq \text{Nil} \} \cup \{ s \}$$

$$E_{pred} = \{ (pred[v], v) \in E : v \in V_{pred} - \{ s \} \}$$

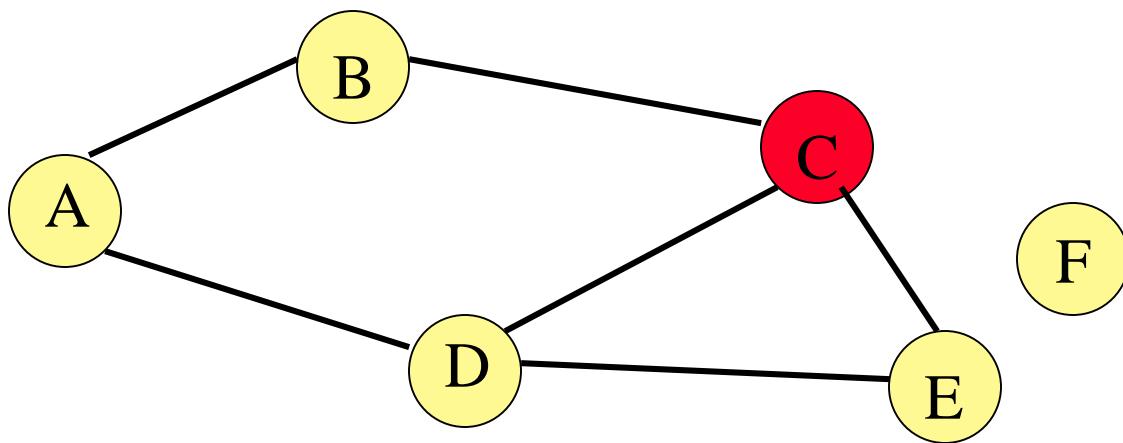


Definizione del problema

Attraversamento di un grafo

Dato un grafo $G = \langle V, E \rangle$ ed un vertice s di V (detto *sorgente*), esplorare *ogni vertice raggiungibile* nel grafo dal vertice s

Calcolare, inoltre, la *distanza* da s di tutti i vertici raggiungibili



$s = C$

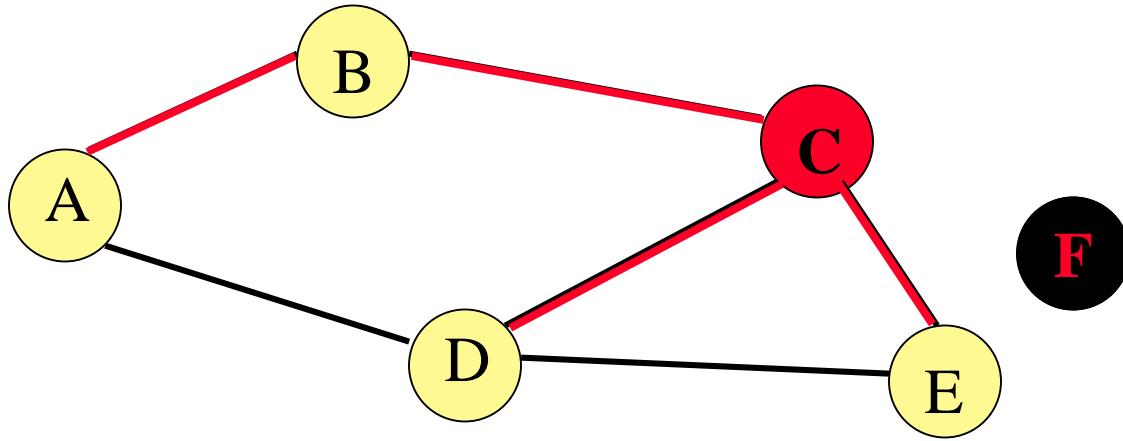
Numero minimo di archi tra i vertici

Definizione del problema

Attraversamento di un grafo

Dato un grafo $G = \langle V, E \rangle$ ed un vertice s di V (detto *sorgente*), esplorare *ogni vertice raggiungibile* nel grafo dal vertice s

Calcolare, inoltre, la *distanza* da s di tutti i vertici raggiungibili



$s = C$

$dist[B] = dist[E] =$
 $= dist[D] = 1$

$dist[A] = 2$

$dist[F] = \infty$

Algoritmo BFS III

BSF(G :grafo, s :vertice)

for each vertice $u \in V(G) - \{s\}$ do

colore[u] = Bianco

dist[u] = ∞

pred[u] = Nil

colore[s] = Grigio

pred[s] = Nil

dist[s] = 0

 Coda = { s }

 while Coda $\neq \emptyset$ do

u = **Testa**[Coda]

 for each $v \in \text{Adiac}(u)$ do

 if **colore**[v] = Bianco then

colore[v] = Grigio

dist[v] = **dist**[u] + 1

pred[v] = u

 Accoda(Coda, v)

 Decoda(Coda)

colore[u] = Nero

Inizializzazione

Aggiornamento delle distanze

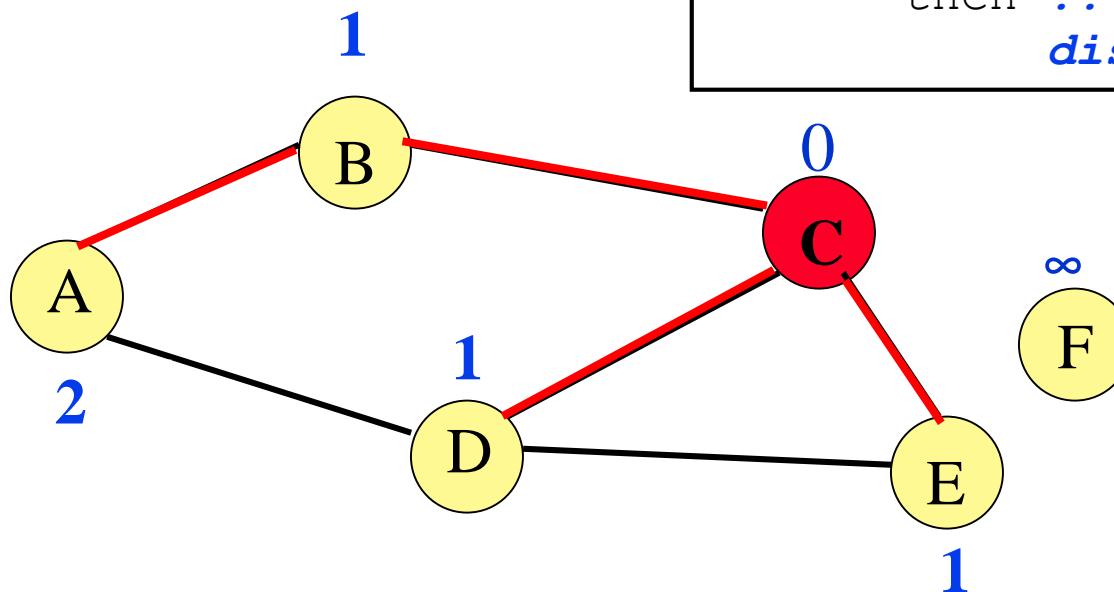
Algoritmo BFS III: calcolo distanze

Inizializzazione

```
for each vertice  $u \in V(G) - \{s\}$ 
    do ...
         $dist[u] = \infty$ 
    ...
 $dist[s] = 0$ 
```

Aggiornamento delle distanze

```
for each  $v \in Adiac(u)$ 
do if  $colore[v] = \text{Bianco}$ 
then ...
 $dist[v] = dist[u] + 1$ 
```



Correttezza di BFS III

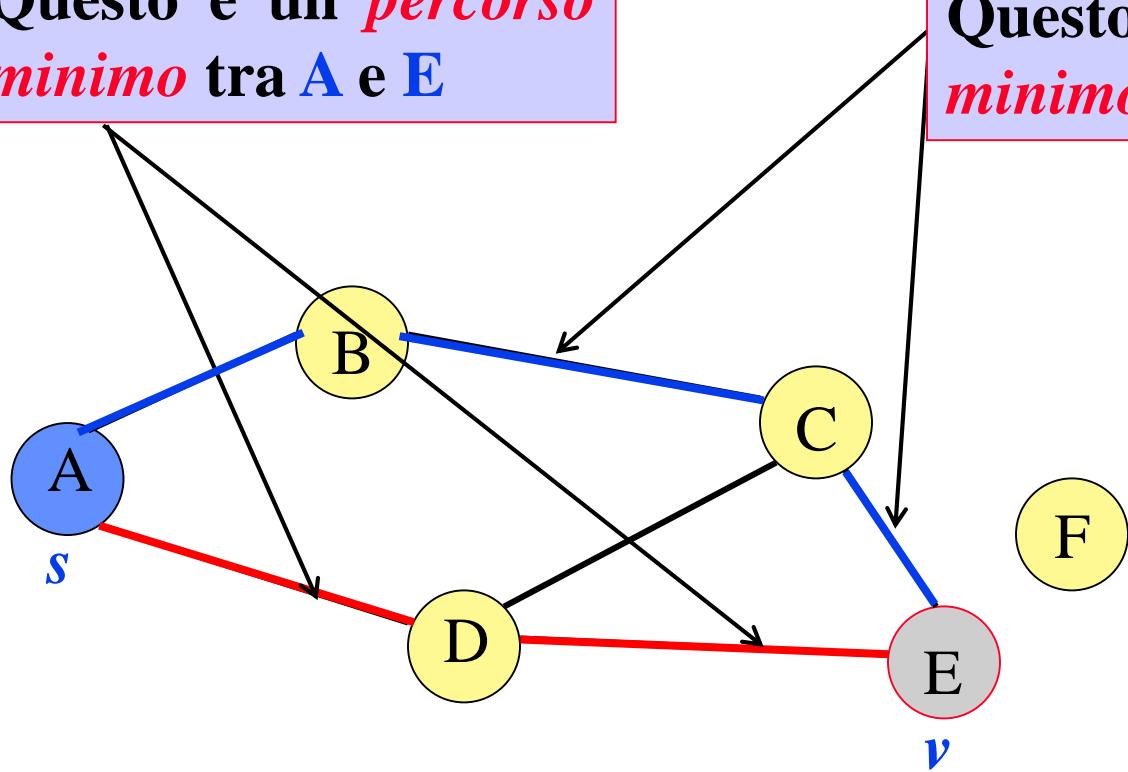
Sia dato un *grafo* $G=<V,E>$ (orientato o non) e un vertice sorgente s :

- durante l'esecuzione dell'algoritmo $BFS(G,s)$, vengono esaminati *tutti i vertici* di V *raggiungibili* da s ;
- Prima di dimostrare la correttezza di BFS , dimostreremo alcune proprietà dei *percorsi minimi*.

Percorsi minimi

Un *percorso minimo* in un *grafo* $G=<V,E>$ tra due vertici s e v è un percorso da s a v che contiene il minimo numero di archi.

Questo è un *percorso minimo* tra A e E



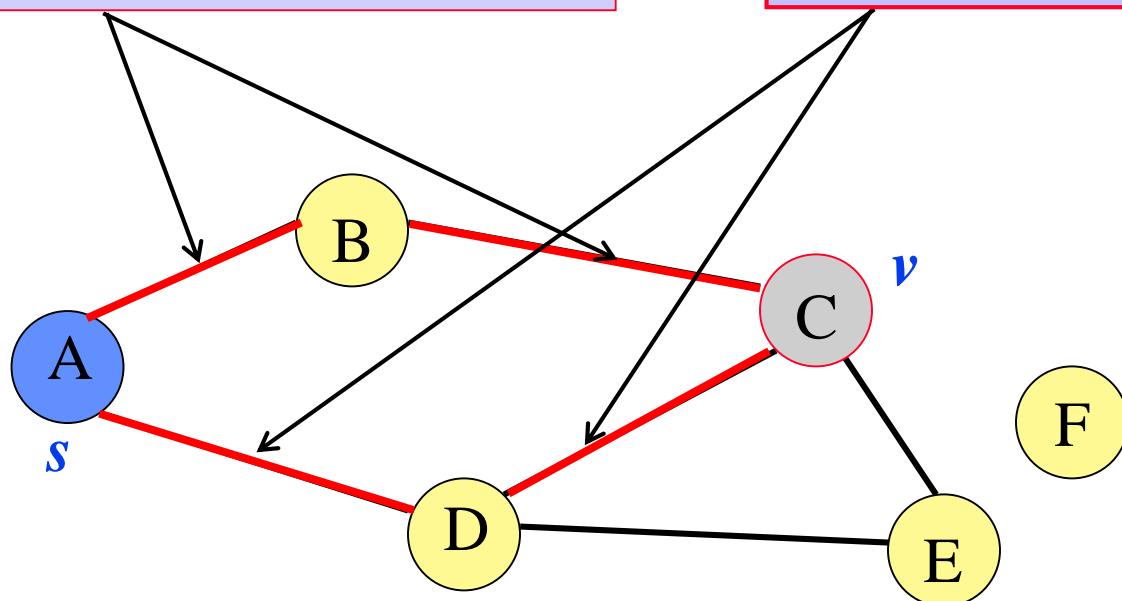
Questo *NON* è un *percorso minimo* tra A e E

Percorsi minimi

Un *percorso minimo* in un *grafo* $G=<V,E>$ tra due vertici s e v è un percorso da s a v che contiene il minimo numero di archi.

Questo è un *percorso minimo* tra A e C

Questo è un *percorso minimo* tra A e C

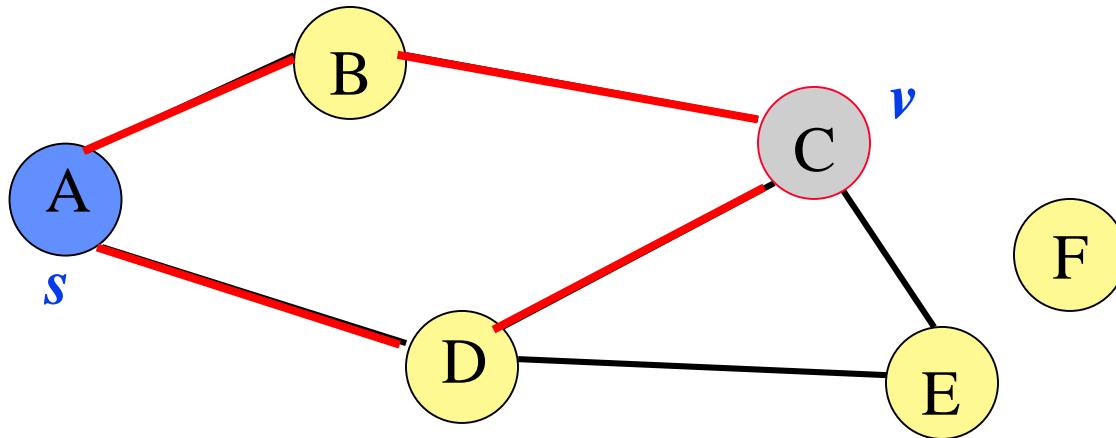


Possono esistere *più percorsi minimi* tra due vertici

Percorsi minimi

Un *percorso minimo* in un *grafo* $G=<V,E>$ tra due vertici s e v è un percorso da s a v che contiene il minimo numero di archi.

La *distanza* $\delta(s,v)$ tra due vertici s e v è la *lunghezza* (numero di archi) di un *percorso minimo* tra s e v .



$$\delta(A,C) = 2$$

La *distanza* tra due vertici è *unica*

Percorsi minimi: proprietà I

Sia $G = \langle V, E \rangle$ un *grafo* (orientato o non) e s un vertice di G . Allora per ogni arco (u, v) di E , vale quanto segue:

$$\delta(s, v) \leq \delta(s, u) + 1$$

Proprietà I: dimostrazione

Ci sono 2 casi:

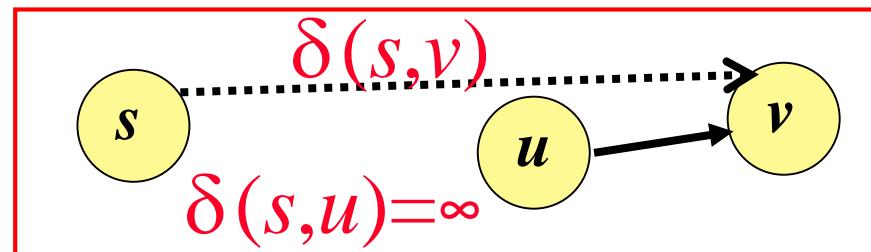
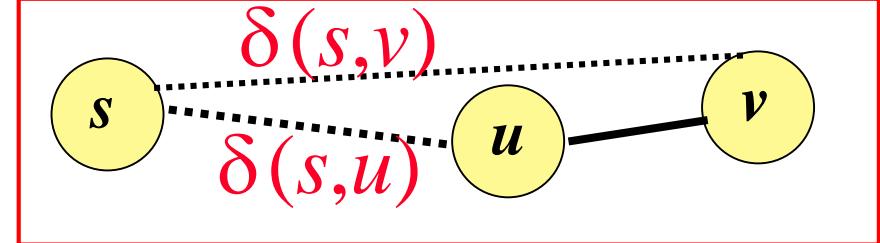
- u è raggiungibile da s
- u non è raggiungibile da s
- u è raggiungibile da s :

Allora anche v lo è.

Il *percorso minore* da s a v in tal caso *non* può essere *più lungo* del percorso minore da s a u seguito dall'arco (v,u) , e quindi $\delta(s,v) \leq \delta(s,u) + 1$

- u non è raggiungibile da s

Allora $\delta(s,u)=\infty$, e nuovamente la diseguaglianza vale.



Percorsi minimi: proprietà II

Sia $G=<V,E>$ un *grafo* (orientato o non).

Supponiamo di eseguire $BFS(G,s)$. Al termine dell'algoritmo, per ogni vertice v di V , vale:

$$dist[v] \geq \delta(s,v)$$

Proprietà II: dimostrazione

Induzione sul *numero di operazioni di inserimento* di vertici in coda.

Ipotesi Induttiva: “per ogni accodamento precedente, vale $dist[v] \geq \delta(s,v)$ per ogni vartice v ”.

Passo Base: è quando s viene posto nella coda. Poiché $dist[s] = 0 = \delta(s,s)$ e $dist[v] = \infty \geq \delta(s,v)$ per ogni altro vertice v , la tesi è banalmente verificata!

```
BSF(G:grafo, s:vertice)
for each vertice u ∈ V(G) - {s}
    do colore[u] = Bianco
        dist[u] =  $\infty$  ←
        pred[u] = Nil
    dist[s] = 0 ←
    Coda = {s}
    ...
    ...
```

Proprietà II: dimostrazione

Passo Induttivo: un vertice bianco v viene posto in coda scorrendo la lista di adiacenza del vertice u in testa. Per ipotesi induttiva $dist[u] \geq \delta(s, u)$.

Dall'assegnamento e dalla Proprietà I risulta che:

```
BSF(G:grafo, s:vertice)
```

```
...
```

```
while Coda ≠ ∅
```

```
    do u = Testa[Coda]
```

```
        for each v ∈ Adiac(u)
```

```
            do if colore[v] = Bianco
```

```
                then colore[v] = Grigio
```

```
                dist[v] = dist[u] + 1 ←
```

```
                Accoda(Coda, v)
```

```
Decoda(Coda)
```

$$dist[v] = dist[u] + 1$$

$$\geq \delta(s, u) + 1$$

$$\geq \delta(s, v)$$

Percorsi minimi: proprietà III

Sia $G = \langle V, E \rangle$ un grafo (orientato o non).

Supponiamo di eseguire $BFS(G, s)$ e che in **coda** siano presenti i vertici $[v_1, \dots, v_n]$ (v_1 è la testa).

Allora:

$$dist[v_n] \leq dist[v_1] + 1$$

$$dist[v_i] \leq dist[v_{i+1}] \quad \text{per ogni } i=1, \dots, n-1$$

Proprietà III: dimostrazione

Dimostriamo per induzione sul numero di operazioni sulla coda.

- *Passo Base:* Inizialmente (*1 operazione sulla coda*), quando la coda contiene solo *s*, la proprietà certamente vale.

Proprietà III: dimostrazione

Ipotesi Induttiva

Dobbiamo dimostrare che la proprietà vale sia per qualsiasi *operazione di accodamento* o di *estrazione* di un vertice dalla coda.

Denotiamo con $[v_1 v_2 \dots v_r]$ coda, dove v_1 è la testa.

*Supponiamo (ipotesi induttiva) che la proprietà valga dopo la **(k-1)-esima operazione sulla coda**, che sarà $[v_1 v_2 \dots v_r]$. Cioè che valga:*

$$dist[v_r] \leq dist[v_1] + 1$$

$$dist[v_i] \leq dist[v_{i+1}]$$

Proprietà III: dimostrazione

- *Passo Induttivo* consideriamo la k -esima operazione
 - 1) quando v_1 viene estratto, v_2 diventa la nuova testa (quando si svuota la proprietà vale banalmente). Allora, poiché si ha $dist[v_1] \leq dist[v_2]$, risulta
$$dist[v_r] \leq dist[v_1] + 1 \leq dist[v_2] + 1$$
e il resto delle diseguaglianze resta identico. Quindi la proprietà vale con v_2 come testa

Proprietà III: dimostrazione

- *Passo Induttivo* consideriamo la k -esima operazione
 - 2) quando si accoda a $[v_1 \ v_2 \ \dots \ v_r]$ il vertice v (nel codice) diventa il nuovo v_{r+1} , $[v_1 \ v_2 \ \dots \ v_r \ v_{r+1}]$, mentre il vertice v_1 è il vertice u la cui lista di adiacenza viene esaminata (nel codice). Allora vale $dist[v_{r+1}] = dist[v] \leq dist[u]+1 = dist[v_1]+1$ e $dist[v_r] \leq dist[v_1]+1 = dist[u]+1 = dist[v] = dist[v_{r+1}]$

Le altre uguaglianze restano invariate...
... e la proprietà vale!

```
u = Testa[Coda]
for each v ∈ Adiac(u)
    do if colore[v] = Bianco
        then dist[v] = dist[u] + 1
            Accoda(Coda, v)
Decoda(Coda)
```

Correttezza di BFS III

Sia dato un *grafo* $G=<V,E>$ (orientato o non) e un vertice sorgente s :

- durante l'esecuzione dell'algoritmo $BFS(G,s)$, vengono esaminati *tutti i vertici* di V *raggiungibili* da s ;
- al termine $dist[v] = \delta(s,v)$ per ogni $v \in V$;
- se $v \neq s$, *uno dei percorsi minimi* tra s e v è il *percorso minimo* da s a $pred[v]$ seguito dall'arco $(pred[v],v)$.

Correttezza di BFS III: dimostrazione

Dimostrazione: consideriamo il caso in cui il vertice v sia raggiungibile da s (vedere sul libro di testo il caso in cui v non è raggiungibile).

- Sia V_k l'insieme dei vertici a distanza (minima) k da s (cioè $V_k = \{v \in V : \delta(s, v) = k\}$).
- La dimostrazione procede per *induzione su k* , cioè sulla distanza di un nodo v da s .

Ipotesi induttiva: per ogni $j < k$, per ogni $v \in V_j$, c'è *solo un istante* in cui l'algoritmo di BFS:

- colora v di *grigio*
- assegna a $dist[v]$ il valore j
- se $v \neq s$, allora assegna a $pred[v]$ il valore u , per qualche $u \in V_{j-1}$
- inserisce v nella coda

Correttezza di BFS III: dimostrazione

Caso Base: Per $k = 0$, $V_0 = \{s\}$ (unico vertice a distanza 0 da s):

- l'inizializzazione colora s di grigio;
- $dist[s]$ viene posto a 0;
- s è messo nella coda.

Quindi la tesi è dimostrata per $k=0$!

Ipotesi induttiva: per ogni $j < k$, per ogni $v \in V_j$, c'è *solo un istante* in cui l'algoritmo di BFS:

- colora v di grigio
- assegna a $dist[v]$ il valore j
- se $v \neq s$, allora assegna a $pred[v]$ il valore u , per qualche $u \in V_{j-1}$
- inserisce v nella coda

Correttezza di BFS III: dimostrazione

Caso induttivo: per $k \geq 1$

- La coda non è mai vuota fino al termine.
- Una volta inserito un vertice u nella coda, né $dist[u]$ né $pred[u]$ cambiano il loro valore.
- Per il teorema precedente, se i vertici sono inseriti nell'ordine v_1, v_2, \dots, v_r , la sequenza delle distanze è crescente monotonicamente ($d[v_i] \leq d[v_{i+1}]$)

Ipotesi induttiva: per ogni $j < k$, per ogni $v \in V_j$, c'è *solo un istante* in cui l'algoritmo di BFS:

- colora v di *grigio*
- assegna a $dist[v]$ il valore j
- se $v \neq s$, allora assegna a $pred[v]$ il valore u , per qualche $u \in V_{j-1}$
- inserisce v nella coda

Correttezza di BFS III: dimostrazione

Caso induttivo:

- Sia ora $v \in V_k$ ($k \geq 1$).
- Dalla proprietà di monotonicità (Prop. III), dal fatto che $\text{dist}[v] \geq k$ (Prop. II) e dall'*ipotesi induttiva*, segue che v (se viene visitato) deve essere visitato dopo che tutti i vertici nell'insieme V_{k-1} sono stati accodati.
- Poiché $\delta(s, v) = k$, esiste un percorso di $k-1$ archi da s ad un vertice u tale che $(u, v) \in E$, e quindi esiste un vertice $u \in V_{k-1}$ con v adiacente a u .

Ipotesi induttiva: per ogni $j < k$, per ogni $v \in V_j$, c'è *solo un istante* in cui l'algoritmo di BFS:

- colora v di *grigio*
- assegna a $\text{dist}[v]$ il valore j
- se $v \neq s$, allora assegna a $\text{pred}[v]$ il valore u , per qualche $u \in V_{j-1}$
- inserisce v nella coda

Correttezza di BFS III: dimostrazione

Caso induttivo:

- Supponiamo che u sia il primo dei vertici in V_{k-1} a cui v è adiacente che è stato colorato di grigio (per Hp. Ind. tutti i vertici in V_{k-1} saranno grigi prima che v venga scoperto).
- Quando verrà *esaminata la lista di adiacenza* di u , v verrà scoperto (ciò non accade prima perché v sta in V_k e non è quindi adiacente a vertici in V_j con $j < k-1$, e u è il primo adiacente di v incontrato per l'ipotesi in alto)

Ipotesi induttiva: per ogni $j < k$, per ogni $v \in V_j$, c'è *solo un istante* in cui l'algoritmo di BFS:

- colora v di *grigio*
- assegna a *dist*[v] il valore j
- se $v \neq s$, allora assegna a *pred*[v] il valore u , per qualche $u \in V_{j-1}$
- inserisce v nella coda

Correttezza di BFS III: dimostrazione

Caso induttivo:

- Quindi v viene colorato di grigio da BFS.
- Viene assegnato $dist[v] = dist[u] + 1 = (k-1) + 1 = k$
- Viene eseguito $pred[v] = u$ e sappiamo (per *ipotesi induttiva*) che $u \in V_{k-1}$
- Viene messo v in coda.

Essendo v un vertice arbitrario in V_k , l'ipotesi induttiva è dimostrata per ogni $k \geq 1$!

Ipotesi induttiva: per ogni $j < k$, per ogni $v \in V_j$, c'è *solo un istante* in cui l'algoritmo di BFS:

- colora v di *grigio*
- assegna a $dist[v]$ il valore j
- se $v \neq s$, allora assegna a $pred[v]$ il valore u , per qualche $u \in V_{j-1}$
- inserisce v nella coda

Correttezza di BFS III: dimostrazione

Quindi, se $v \in V_k$, allora certamente $\text{pred}[v] \in V_{k-1}$

È quindi possibile ottenere il *percorso minimo da s a v* estendendo il percorso minimo da s a $\text{pred}[v]$ con l'arco $(\text{pred}[v], v)$.

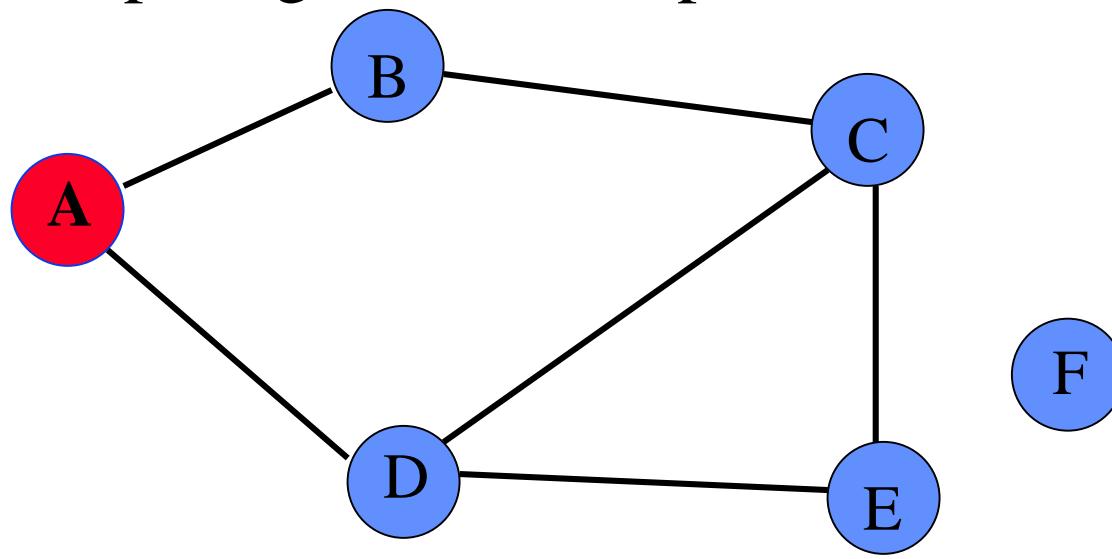
Ipotesi induttiva: per ogni $j < k$, per ogni $v \in V_j$, c'è *solo un istante* in cui l'algoritmo di BFS:

- colora v di *grigio*
- assegna a $\text{dist}[v]$ il valore j
- se $v \neq s$, allora assegna a $\text{pred}[v]$ il valore u , per qualche $u \in V_{j-1}$
- inserisce v nella coda

Alberi breadth-first

Un *albero breadth-first* di un grafo non orientato $G = \langle V, E \rangle$ con *sorgente* s , è un *albero libero* $G' = \langle V', E' \rangle$ tale che:

- G' è un sottografo del grafo non orientato sottostante di G
- $v \in V'$ se e solo se v è raggiungibile da s
- per ogni $v \in V'$, il percorso da s a v è *minimo*

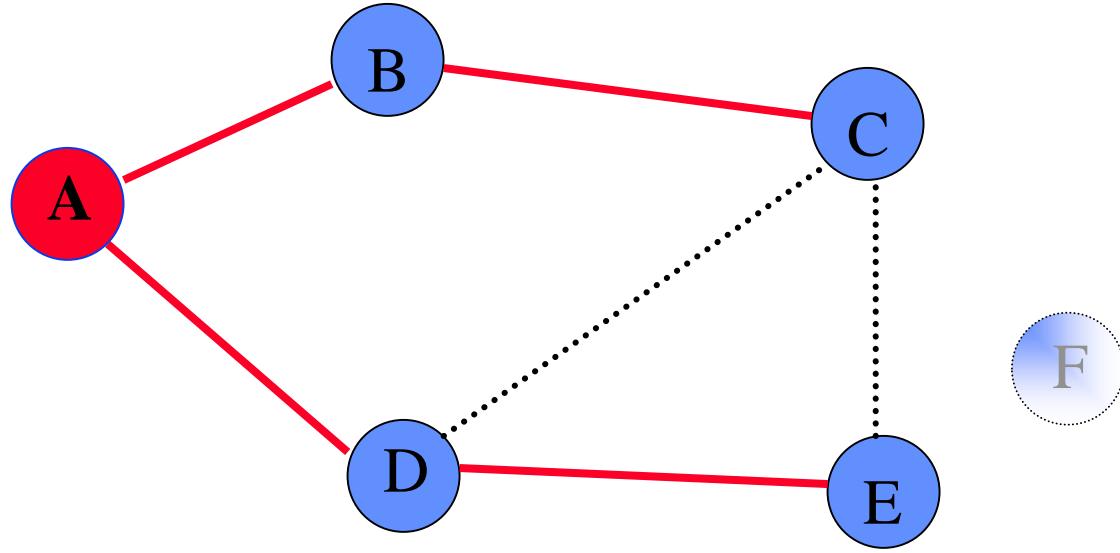


$s = A$

Alberi breadth-first

Un *albero breadth-first* di un grafo non orientato $G = \langle V, E \rangle$ con *sorgente* s , è un *albero libero* $G' = \langle V', E' \rangle$ tale che:

- G' è un sottografo del grafo non orientato sottostante G
- $v \in V'$ se e solo se v è raggiungibile da s
- per ogni $v \in V'$, il percorso da s a v è **minimo**



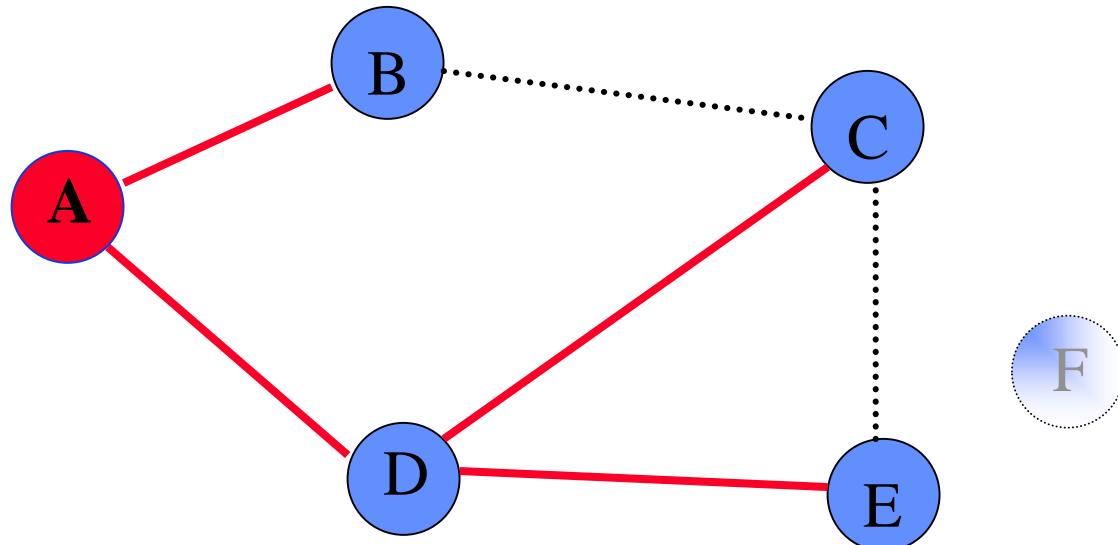
$s = A$

Questo è un albero
breadth-first

Alberi breadth-first

Un *albero breadth-first* di un grafo non orientato $G = \langle V, E \rangle$ con *sorgente* s , è un *albero libero* $G' = \langle V', E' \rangle$ tale che:

- G' è un sottografo del grafo non orientato sottostante G
- $v \in V'$ se e solo se v è raggiungibile da s
- per ogni $v \in V'$, il percorso da s a v è **minimo**



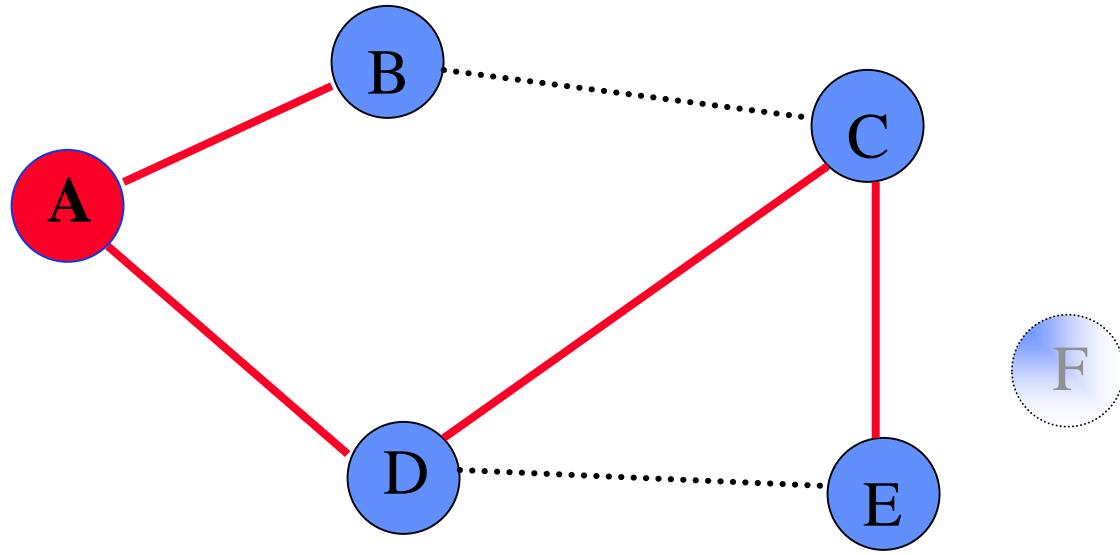
$s = A$

Questo è un altro
albero breadth-first

Alberi breadth-first

Un *albero breadth-first* di un grafo non orientato $G = \langle V, E \rangle$ con *sorgente* s , è un *albero libero* $G' = \langle V', E' \rangle$ tale che:

- G' è un sottografo del grafo non orientato sottostante G
- $v \in V'$ se e solo se v è raggiungibile da s
- per ogni $v \in V'$, il percorso da s a v è **minimo**



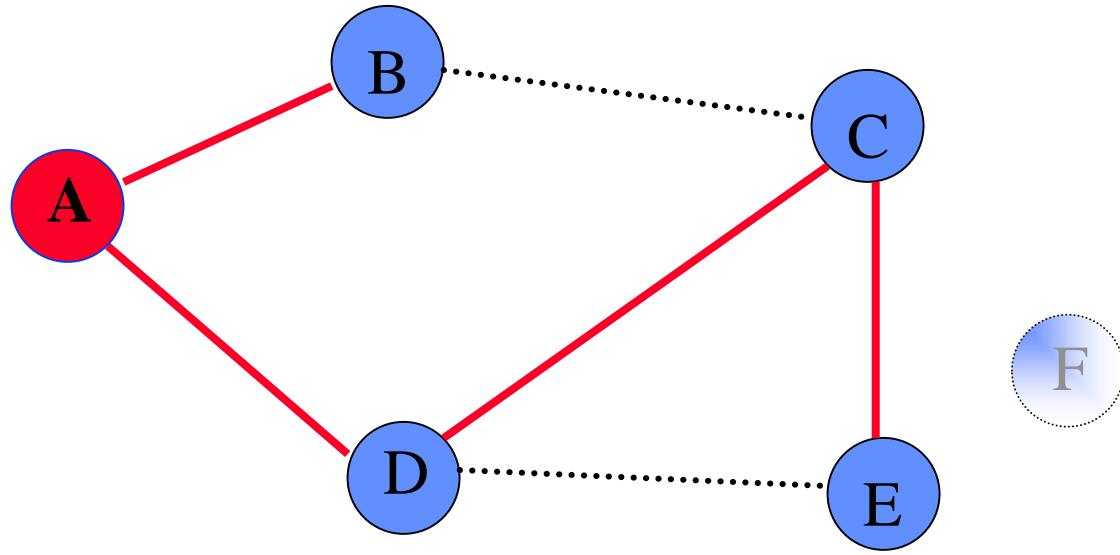
$s = A$

Questo **NON** è un albero
breadth-first! *Perché?*

Alberi breadth-first

Un *albero breadth-first* di un grafo non orientato $G = \langle V, E \rangle$ con *sorgente* s , è un *albero libero* $G' = \langle V', E' \rangle$ tale che:

- G' è un sottografo del grafo non orientato sottostante G
- $v \in V'$ se e solo se v è raggiungibile da s
- per ogni $v \in V'$, il percorso da s a v è **minimo**



$s = A$

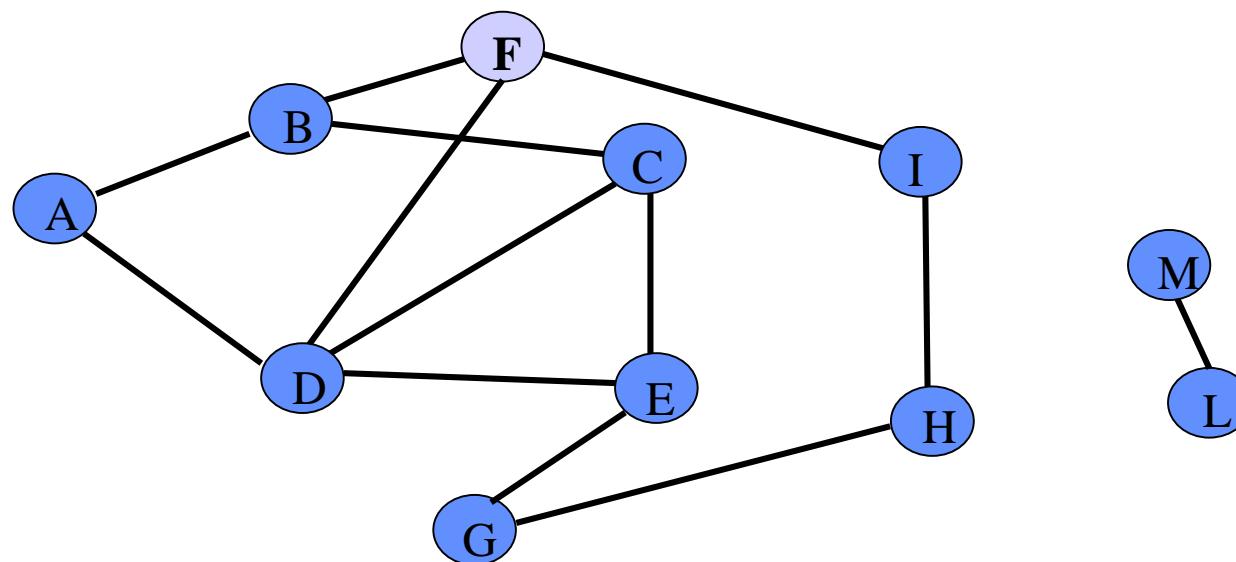
Perché il *percorso* da A ad E *NON* è minimo

Sottografo dei predecessori e BFS

- L'*algoritmo BFS* eseguito su un grafo $G = \langle V, E \rangle$ costruisce (nell'array $\text{pred}[]$) il *sottografo dei predecessori* denotato con $G_{\text{pred}} = \langle V_{\text{pred}}, E_{\text{pred}} \rangle$, dove:

$$V_{\text{pred}} = \{ v \in V : \text{pred}[v] \neq \text{Nil} \} \cup \{ s \}$$

$$E_{\text{pred}} = \{ (\text{pred}[v], v) \in E : v \in V_{\text{pred}} - \{ s \} \}$$

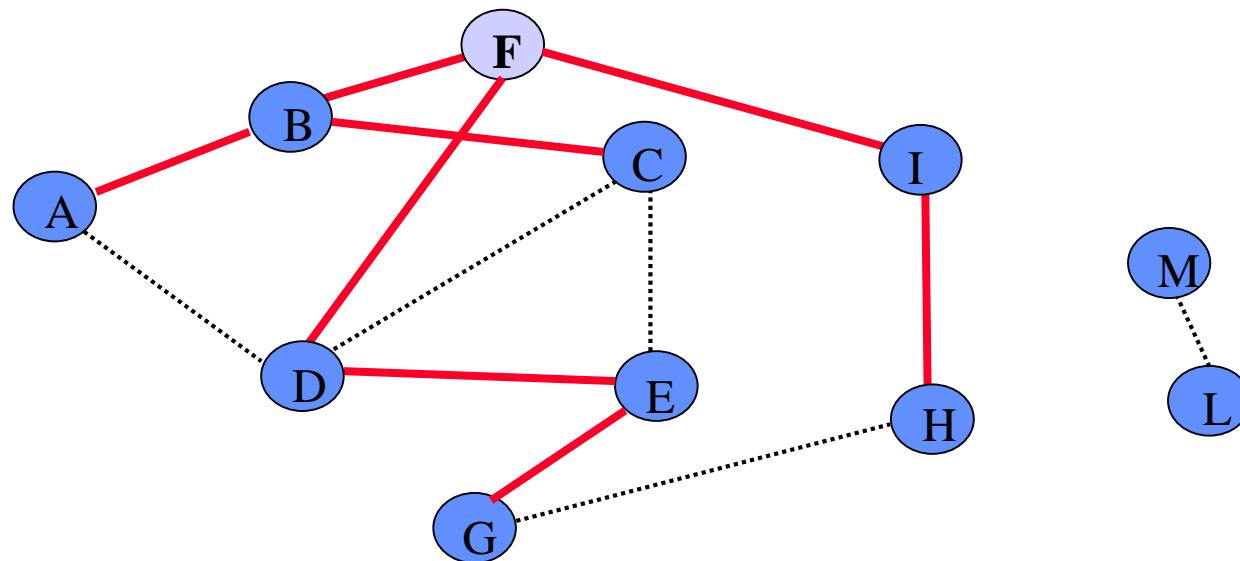


Sottografo dei predecessori e BFS

- L'*algoritmo BFS* eseguito su un grafo $G = \langle V, E \rangle$ costruisce (nell'array $\text{pred}[]$) il *sottografo dei predecessori* denotato con $G_{\text{pred}} = \langle V_{\text{pred}}, E_{\text{pred}} \rangle$, dove:

$$V_{\text{pred}} = \{ v \in V : \text{pred}[v] \neq \text{Nil} \} \cup \{ s \}$$

$$E_{\text{pred}} = \{ (\text{pred}[v], v) \in E : v \in V_{\text{pred}} - \{ s \} \}$$

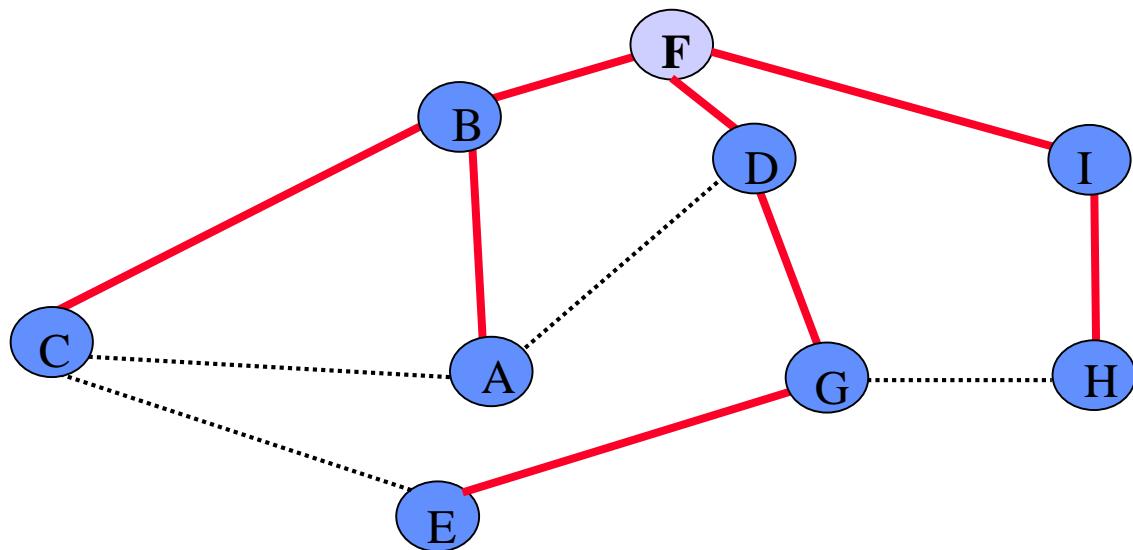


Sottografo dei predecessori e BFS

- L'algorithm *BFS* eseguito su un grafo $G = \langle V, E \rangle$ costruisce (nell'array *pred*) il *sottografo dei predecessori* denotato con $G_{pred} = \langle V_{pred}, E_{pred} \rangle$, dove:

$$V_{pred} = \{ v \in V : pred[v] \neq \text{Nil} \} \cup \{ s \}$$

$$E_{pred} = \{ (pred[v], v) \in E : v \in V_{pred} - \{ s \} \}$$



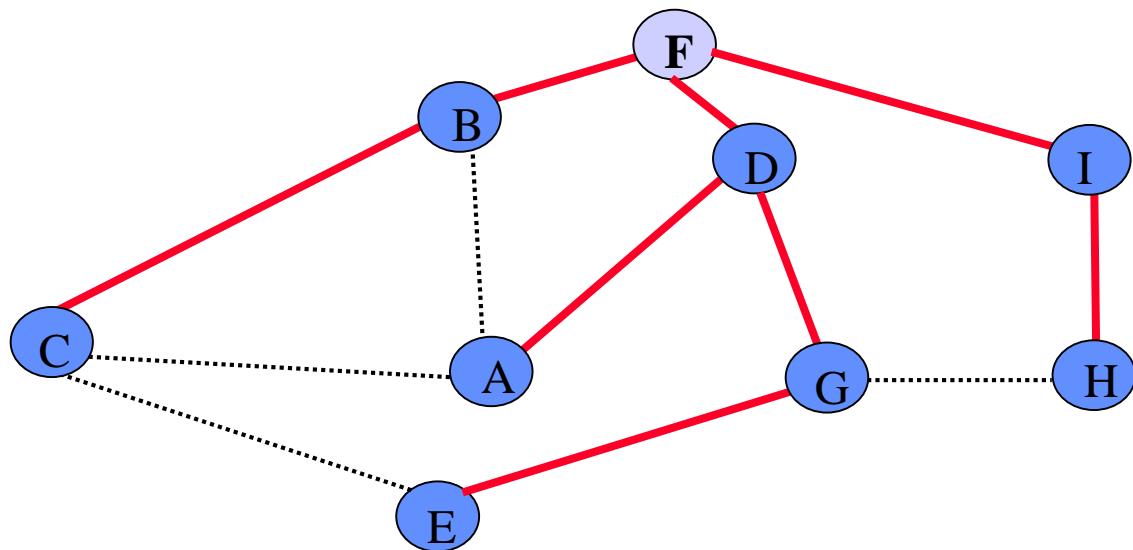
Questo è un albero *breadth-first* di G

Sottografo dei predecessori e BFS

- L'algorithm *BFS* eseguito su un grafo $G = \langle V, E \rangle$ costruisce (nell'array *pred[]*) il *sottografo dei predecessori* denotato con $G_{pred} = \langle V_{pred}, E_{pred} \rangle$, dove:

$$V_{pred} = \{ v \in V : pred[v] \neq \text{Nil} \} \cup \{ s \}$$

$$E_{pred} = \{ (pred[v], v) \in E : v \in V_{pred} - \{ s \} \}$$



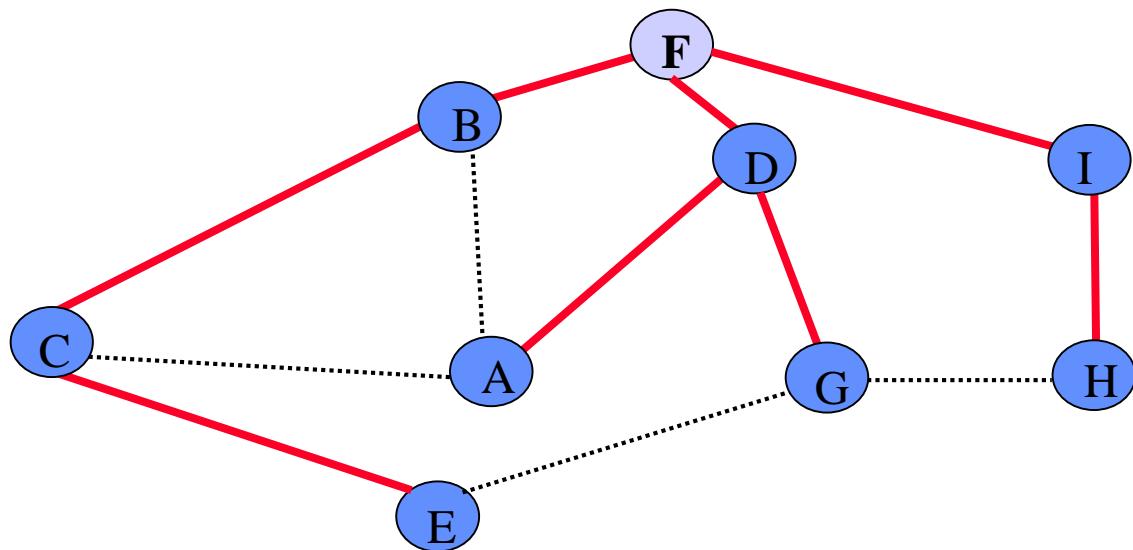
Questo è un altro albero *breadth-first* di G

Sottografo dei predecessori e BFS

- L'algorithm *BFS* eseguito su un grafo $G = \langle V, E \rangle$ costruisce (nell'array *pred[]*) il *sottografo dei predecessori* denotato con $G_{pred} = \langle V_{pred}, E_{pred} \rangle$, dove:

$$V_{pred} = \{ v \in V : pred[v] \neq \text{Nil} \} \cup \{s\}$$

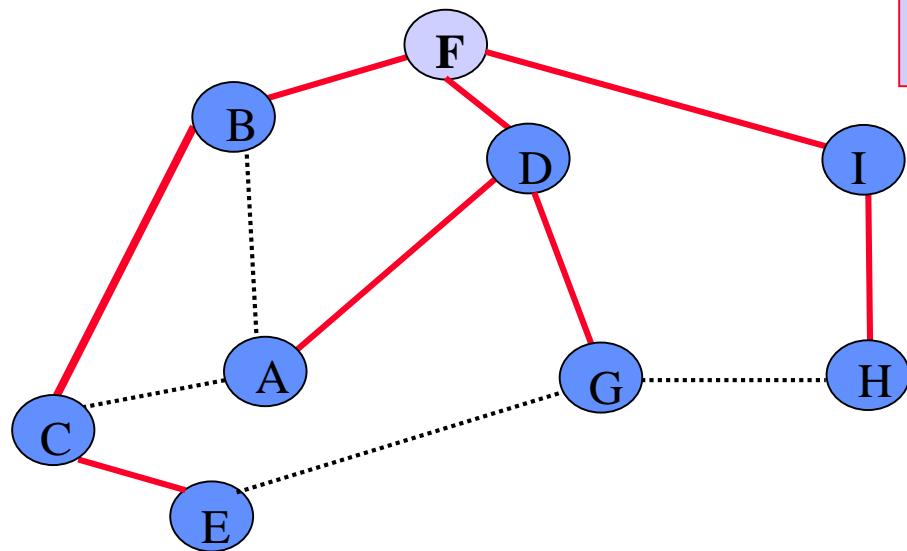
$$E_{pred} = \{ (pred[v], v) \in E : v \in V_{pred} - \{s\} \}$$



Questo è un altro albero breadth-first di G

Alberi breadth-first e BFS

- L'*algoritmo BFS* eseguito su un grafo $G=<V,E>$ costruisce in *pred[]* il *sottografo dei predecessori* $G_{pred}=<V_{pred}, E_{pred}>$ in modo tale che G_{pred} sia un *albero breadth-first* di G .



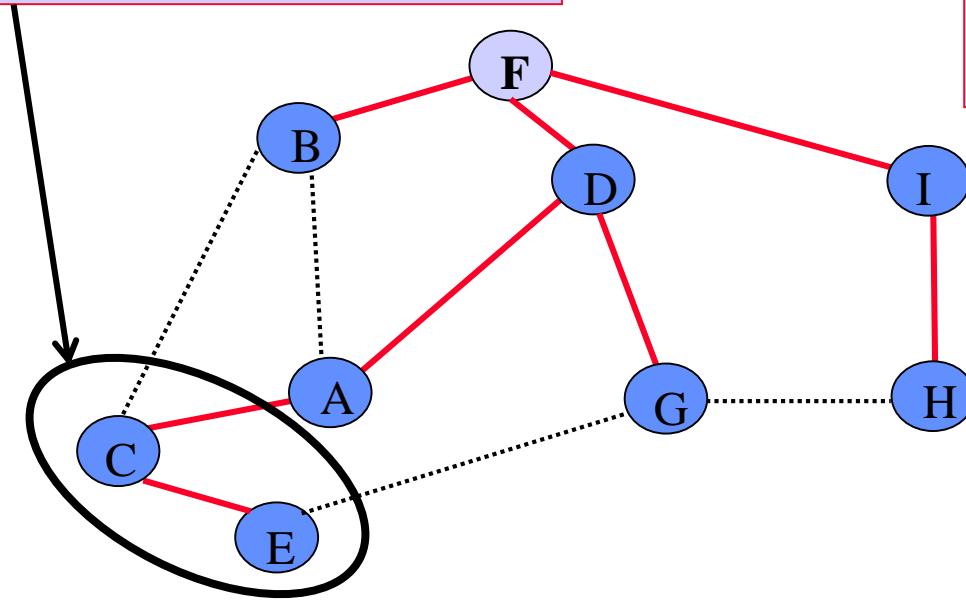
Questo è un grafo dei predecessori di G costruito da BFS

Questo è un albero breadth-first di G

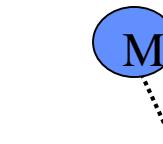
Alberi breadth-first e BFS

- L'*algoritmo BFS* eseguito su un grafo $G=<V,E>$ costruisce in *pred[]* il *sottografo dei predecessori* $G_{pred}=<V_{pred},E_{pred}>$ in modo tale che G_{pred} sia un *albero breadth-first* di G .

I percorsi da F a C e da F ad E non sono minimi



Questo *non* è un grafo dei predecessori di G costruito da BFS



Questo *non* è un albero breadth-first di G

Alberi breadth-first e BFS

Teorema: L'*algoritmo BFS* eseguito su un grafo $G = \langle V, E \rangle$ costruisce in $\text{pred}[]$ il *sottografo dei predecessori* $G_{\text{pred}} = \langle V_{\text{pred}}, E_{\text{pred}} \rangle$ in modo tale che G_{pred} sia un *albero breadth-first* di G .

Dimostrazione: BFS assegna $\text{pred}[v] = u$ solo se $(u, v) \in E$ e $\delta(s, v) < \infty$ (solo se v è raggiungibile da s).

Quindi V_{pred} consiste di vertici in V tutti raggiungibili da s

Poiché G_{pred} è un albero, contiene un unico percorso da s ad ogni vertice in V_{pred}

Usando *induttivamente* il *teorema di correttezza* (parte finale), segue che ognuno di questi percorsi è minimo.

Alberi breadth-first e BFS

Teorema: L'*algoritmo BFS* eseguito su un grafo $G = \langle V, E \rangle$ costruisce in $\text{pred}[]$ il *sottografo dei predecessori* $G_{\text{pred}} = \langle V_{\text{pred}}, E_{\text{pred}} \rangle$ in modo tale che G_{pred} sia un *albero breadth-first* di G .

Dimostrazione: Usando *induttivamente* il *teorema di correttezza* (parte finale), segue che ognuno di questi percorsi è minimo. Induzione sulla distanza k di v da s .

Passo Base: Se $k=0$ segue banalmente.

Sia dato un *grafo* $G = \langle V, E \rangle$ (orientato o non) e un vertice sorgente s :

- se $v \neq s$, *uno dei percorsi minimi* tra s e v è il *percorso minimo* da s a $\text{pred}[v]$ seguito dall'arco $(\text{pred}[v], v)$.

Alberi breadth-first e BFS

Teorema: L'*algoritmo BFS* eseguito su un grafo $G = \langle V, E \rangle$ costruisce in $\text{pred}[]$ il *sottografo dei predecessori* $G_{\text{pred}} = \langle V_{\text{pred}}, E_{\text{pred}} \rangle$ in modo tale che G_{pred} sia un *albero breadth-first* di G .

Dimostrazione: Usando *induttivamente* il *teorema di correttezza* (parte finale), segue che ognuno di questi percorsi è minimo. Induzione sulla distanza k di v da s .

Passo Induttivo: Il percorso tra s e $\text{pred}[v]$ è minimo per induzione

Sia dato un *grafo* $G = \langle V, E \rangle$ (orientato o non) e un vertice sorgente s :

- se $v \neq s$, uno dei percorsi minimi tra s e v è il *percorso minimo* da s a $\text{pred}[v]$ seguito dall'arco $(\text{pred}[v], v)$.

Ma allora per il *teorema di correttezza* lo è anche il *percorso* da s a $\text{pred}[v]$ seguito dall'arco $(\text{pred}[v], v)$.

Applicazione di BFS: calcolo del percorso minimo tra due vertici

Definizione del problema:

Dato un grafo G e due vertici s e v , stampare il *percorso minimo* che congiunge s e v .

Sfruttando le *proprietà* di BFS che abbiamo dimostrato fin qui, possiamo facilmente definire un algoritmo che utilizza BFS opportunamente e che risolve il problema.

Stampa del percorso minimo

Percorso-minimo (G :grafo, s , v :vertice)

BFS ($G, s, pred[]$)

Stampa-percorso ($G, s, v, pred$)

Stampa-percorso (G :grafo, s , v :vertice, $pred[]$:array)

if $v = s$ **then**

stampa s

else if $pred[v] = NIL$ **then**

stampa "non esiste cammino tra s e v "

else

Stampa-percorso ($G, s, pred[v], pred$)

print v

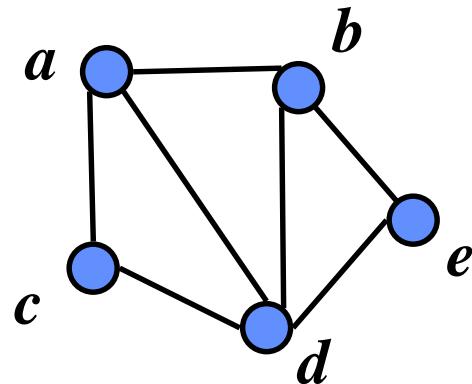
Algoritmi e Strutture Dati (Mod. B)

Algoritmi su grafi
Ricerca in profondità
(Deepth-First Search) Parte I

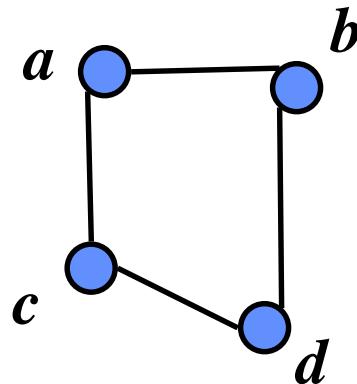
Sottografo di copertura

Un *sottografo* di $G = (V, E)$ è un grafo $H = (V^*, E^*)$ tale che $V^* \subseteq V$ e $E^* \subseteq E$.

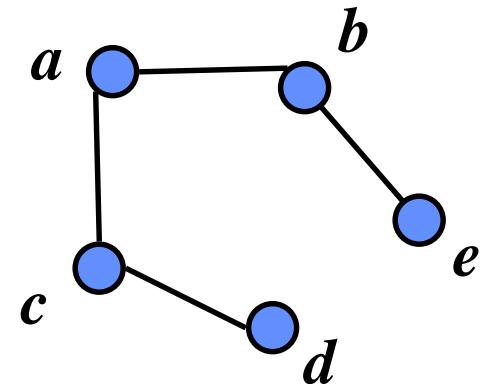
- H' è un *sottografo di copertura* (o *di supporto* o sottografo “*spanning*”) di G se
 - $V^* = V$ e $E^* \subseteq E$



G



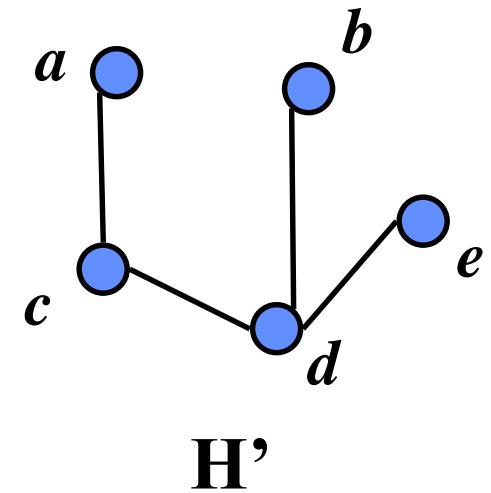
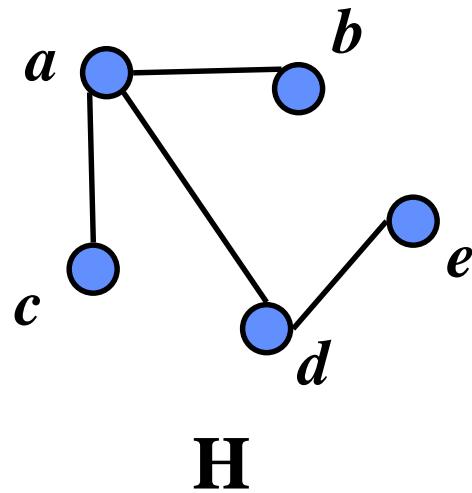
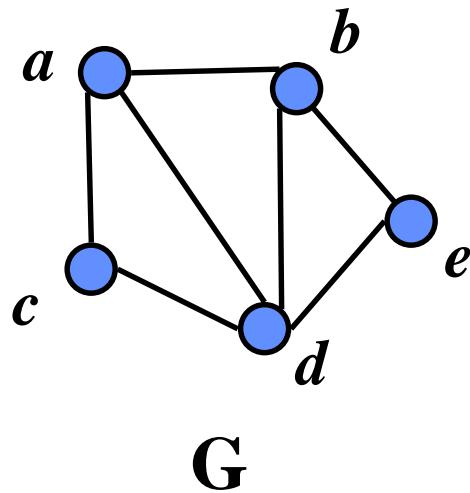
H



H'

Albero di copertura

- Un grafo $H = (V^*, E^*)$ è un *albero di copertura* (o *albero “spanning”*) del grafo $G = (V, E)$ se
 - H è un *grafo di copertura* di G
 - H è un *albero*



Visita in Profondità (DFS)

- **Tecnica di visita di un grafo**
 - È una variazione della *visita in profondità* per alberi binari
- **La visita di s procede come segue:**
 - Si visitano ricorsivamente tutti i vertici adiacenti ad s ;
 - Si termina la visita del vertice s e si ritorna.
- **Bisogna evitare di rivisitare vertici già visitati**
 - Bisogna anche qui evitare i cicli
 - Nuovamente, quando un vertice è stato scoperto e (poi) visitato viene marcato opportunamente (*colorandolo*)

Algoritmo DFS

Manterremo traccia del *momento* (*tempo*) in cui ogni vertice v viene *scoperto* e del momento in cui viene *visitato* (o *terminato*).

Useremo inoltre due array $d[v]$ e $f[v]$ che registrano il momento in cui v verrà scoperto e quello in cui verrà visitato.

La variabile globale *tempo* serve a registrare il passaggio del tempo.

Il tempo viene usato per *studiare* le *proprietà* di *DFS*

DFS: intuizioni

I passi dell'algoritmo *DFS*

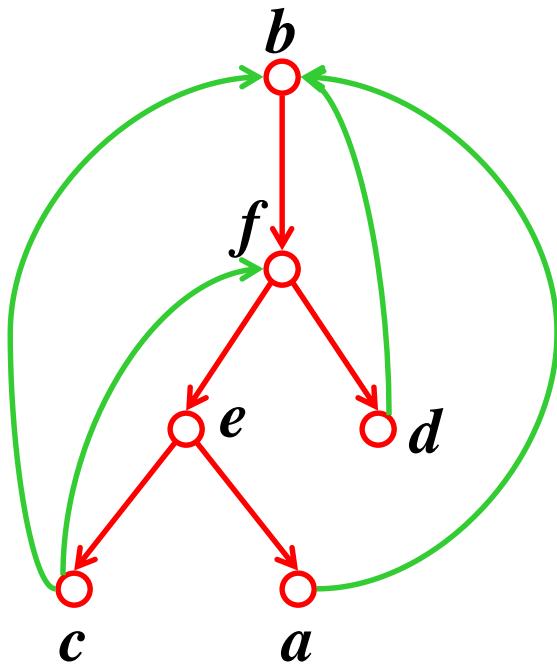
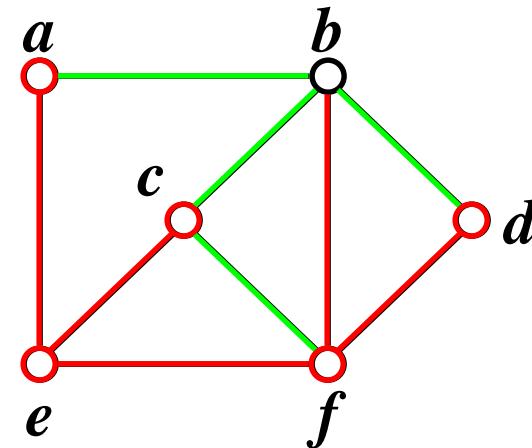
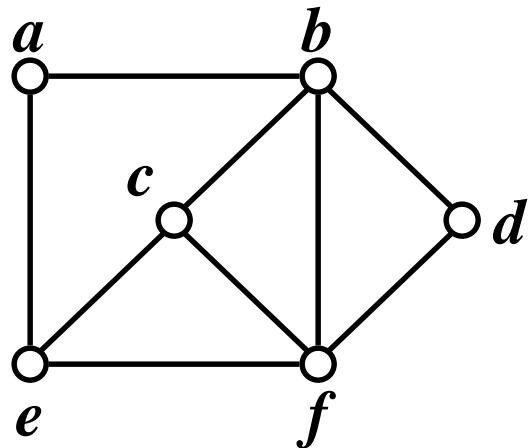
- si parte da un vertice *non visitato* s e lo si visita
- si sceglie un vertice *non scoperto* adiacente ad s .
- da s si attraversa quindi un percorso di vertici adiacenti (visitandoli) finché possibile (*DFS-Visita*):
 - cioè finché non si incontra un vertice già scoperto/visitato
- appena si resta “*bloccati*” (tutti gli archi da un vertice sono stati scoperti), si torna indietro (backtracking) di un passo (vertice) nel percorso attraversato (aggiornando il vertice s al vertice corrente dopo il passo all'indietro).
- si ripete il processo ripartendo dal passo.

DFS: DFS-Visita

DFS-Visita: algoritmo principale della ***DFS***
sia dato un vertice u di colore bianco in ingresso

- visitare il vertice u : colorare u di grigio e assegnare il tempo di inizio visita $d[u]$
- visitare in ***DFS ricorsivamente*** ogni vertice bianco adiacente ad u con ***DFS-Visita***
- colorare di nero u e assegnare il tempo di fine visita $f[u]$.

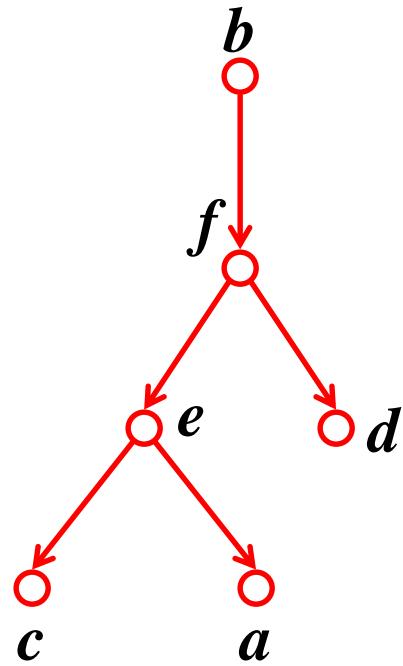
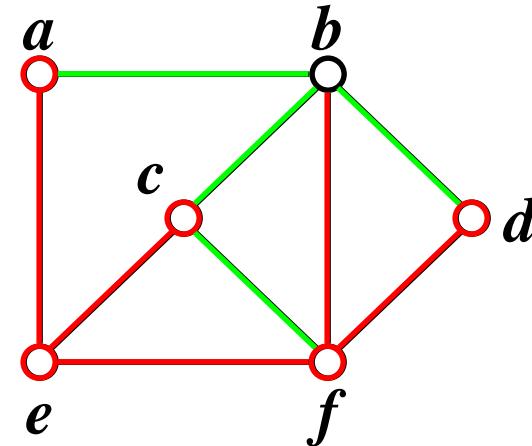
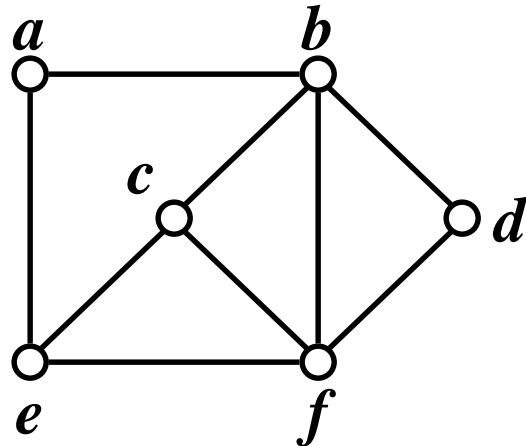
Chiamata ricorsiva



b f e c a d

Albero di copertura Depth-first

Archi dell'albero →
Archi di ritorno →



b f e c a d

Albero di copertura è la Foresta Depth-First

Archi dell'albero →

Algoritmo DFS

DSF (G :grafo)

```
for each vertice  $u \in V$ 
    do  $\text{colore}[u] = \text{Bianco}$ 
         $\text{pred}[u] = \text{NIL}$ 
 $\text{tempo} = 0$ 
```

```
for each vertice  $u \in V$ 
    do if  $\text{colore}[u] = \text{Bianco}$ 
        then DFS-Visita ( $G, u$ )
```

Inizializzazione del grafo e della variabile tempo

DSF-Visita (G :grafo, u :vertice)

```
 $\text{colore}[u] = \text{Grigio}$ 
 $d[u] = \text{tempo} = \text{tempo} + 1$ 
```

```
for each vertice  $v \in \text{Adiac}[u]$ 
    do if  $\text{colore}[v] = \text{Bianco}$ 
        then  $\text{pred}[v] = u$ 
            DFS-Visita ( $G, v$ )
```

```
 $\text{colore}[u] = \text{Nero}$ 
```

```
 $f[u] = \text{tempo} = \text{tempo} + 1$ 
```

Abbreviazione per:
 $\text{tempo} = \text{tempo} + 1$
 $d[u] = \text{tempo}$

Abbreviazione per:
 $\text{tempo} = \text{tempo} + 1$
 $f[u] = \text{tempo}$

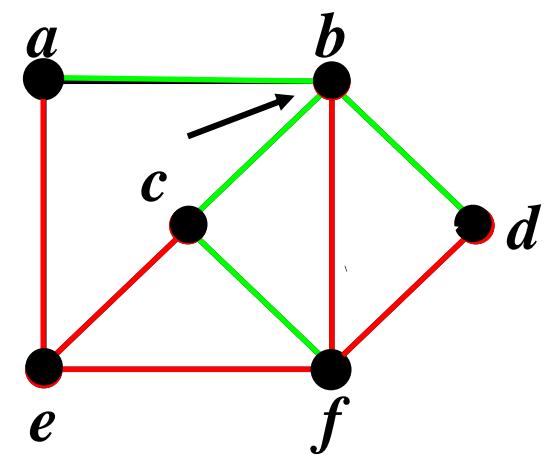
DFS: simulazione

DSF(G :grafo)

```
for each vertice  $u \in V$ 
    do colore[ $u$ ] = Bianco
        pred[ $u$ ] = NIL
tempo = 0
for each vertice  $u \in V$ 
    do if colore[ $u$ ] = Bianco
        then DFS-Visita( $G, u$ )
```

DSF-Visita(G :grafo, u :vertice)

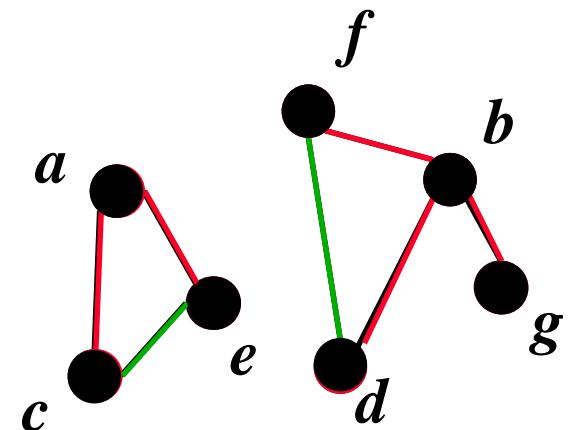
```
colore[ $u$ ] = Grigio
d[ $u$ ] = tempo = tempo + 1
for each vertice  $v \in \text{Adiac}[u]$ 
    do if colore[ $v$ ] = Bianco
        then pred[ $v$ ] =  $u$ 
            DFS-Visit( $G, v$ )
colore[ $u$ ] = Nero
f[ $u$ ] = tempo = tempo + 1
```



Alberi di copertura multipli

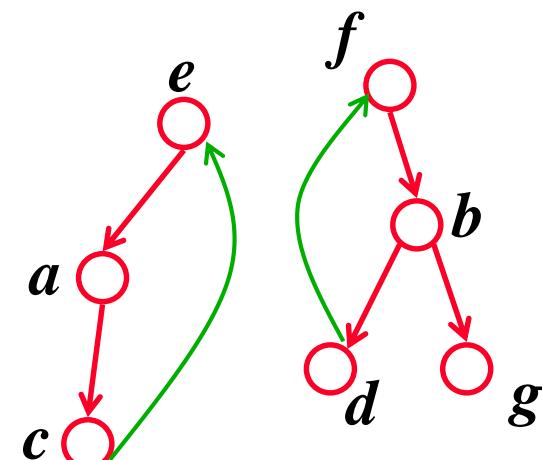
DSF(G :grafo)

```
for each vertice  $u \in V$ 
    do colore[ $u$ ] = Bianco
        pred[ $u$ ] = NIL
tempo = 0
for each vertice  $u \in V$ 
    do if colore[ $u$ ] = Bianco
        then DFS-Visita( $G, u$ )
```



DSF-Visita(G :grafo, u :vertice)

```
colore[ $u$ ] = Grigio
d[ $u$ ] = tempo = tempo + 1
for each vertice  $v \in \text{Adiac}[u]$ 
    do if colore[ $v$ ] = Bianco
        then pred[ $v$ ] =  $u$ 
            DFS-Visit( $G, v$ )
colore[ $u$ ] = Nero
f[ $u$ ] = tempo = tempo + 1
```



Tempo di esecuzione di DFS

DSF(G :grafo)

```
for each vertice  $u \in V$ 
    do  $\text{colore}[u] = \text{Bianco}$ 
         $\text{pred}[u] = \text{NIL}$ 
 $\text{tempo} = 0$ 
```

```
for each vertice  $u \in V$ 
    do if  $\text{colore}[u] = \text{Bianco}$ 
        then  $\text{DFS-Visita}(G, u)$ 
```

$\Theta(|V|)$

$\Theta(|V|)$

DSF-Visita(G :grafo, u :vertice)

```
 $\text{colore}[u] = \text{Grigio}$ 
 $d[u] = \text{tempo} = \text{tempo} + 1$ 
for each vertice  $v \in \text{Adiac}[u]$ 
    do if  $\text{colore}[v] = \text{Bianco}$ 
        then  $\text{pred}[v] = u$ 
             $\text{DFS-Visita}(G, v)$ 
```

```
 $\text{colore}[u] = \text{Nero}$ 
 $f[u] = \text{tempo} = \text{tempo} + 1$ 
```

$|V|$ volte

$\Theta(|E_u|)$

Chiamata solo per vertici
non ancora visitati

Tempo di esecuzione di DFS

DSF(G :grafo)

```
for each vertice  $u \in V$ 
    do colore[ $u$ ] = Bianco
        pred[ $u$ ] = NIL
tempo = 0
for each vertice  $u \in V$ 
    do if colore[ $u$ ] = Bianco
        then DFS-Visita( $G, u$ )
```

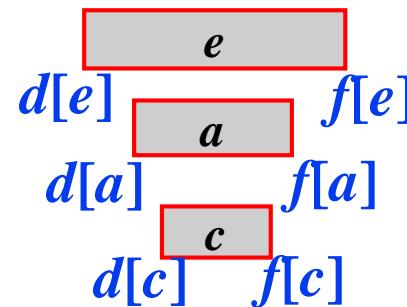
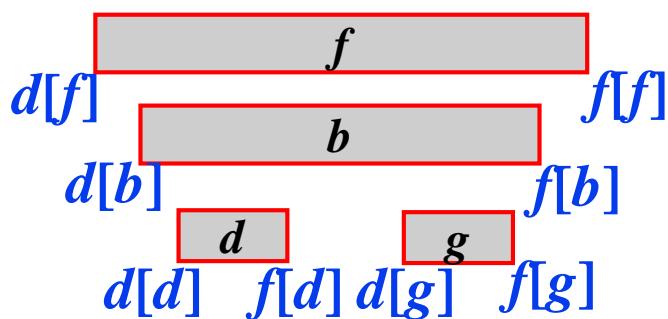
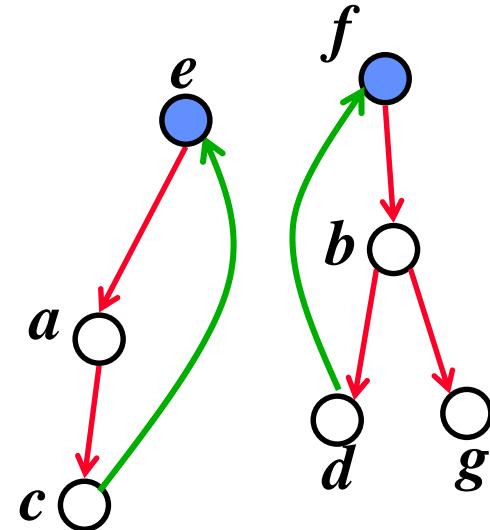
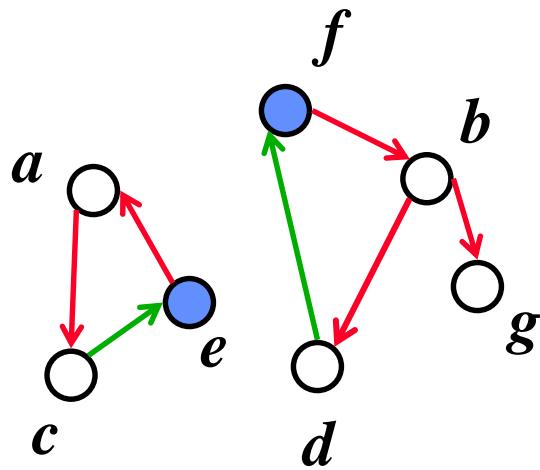
$$\Theta(|V| + |E|)$$

Proprietà di DFS: struttura a parentesi

Teorema: In ogni DFS di un grafo G , per ogni coppia di vertici u e v , *una sola* delle condizioni seguenti vale:

- Gli intervalli $[d[u], f[u]]$ e $[d[v], f[v]]$ sono interamente disgiunti
- L'intervallo $[d[u], f[u]]$ è interamente contenuto nell'intervallo $[d[v], f[v]]$ e u è in *descendente* di v nell'albero DF.
- L'intervallo $[d[v], f[v]]$ è interamente contenuto nell'intervallo $[d[u], f[u]]$ e v è in *descendente* di u nell'albero DF.

Struttura a parentesi: intuizione



Proprietà di DFS: struttura a parentesi

Dimostrazione: Due sono i casi

- $d[u] < d[v]$

Due sottocasi:

① $d[v] < f[u]$. Quindi v è scoperto mentre u è ancora grigio.

Questo implica che v è *descendente* di u (*perché?*)

Inoltre, v è stato scoperto più recentemente di (dopo) u ; perciò la sua lista di archi uscenti viene esplorata, e v viene visitato (terminato) e a $f[v]$ assegnato un valore.

Quindi $[d[v], f[v]]$ è totalmente incluso in $[d[u], f[u]]$

② $f[u] < d[v]$. Poiché $d[u] < f[u]$, segue che $[d[u], f[u]]$ e $[d[v], f[v]]$ sono totalmente disgiunti

- $d[u] > d[v]$

Proprietà di DFS: struttura a parentesi

Dimostrazione: Due sono i casi

- $d[u] < d[v]$ ✓
- $d[u] > d[v]$

Due sottocasi: il ragionamento è simile a prima ma con i ruoli di u e v invertiti

① $d[u] < f[v]$.

Risulta che $[d[u], f[u]]$ è completamente incluso in $[d[v], f[v]]$ e u *discendente* di v

② $f[v] < d[u]$.

Poiché $d[u] < f[u]$, segue che $[d[v], f[v]]$ e $[d[u], f[u]]$ sono totalmente disgiunti (e in due sottoalberi distinti)

Proprietà di DFS: struttura a parentesi

Corollario: Un vertice v è un *discendente* di u nella *foresta DF* di un grafo G se e solo se

$$d[u] < d[v] < f[v] < f[u].$$

Dimostrazione: Immediata conseguenza del teorema precedente.

Proprietà di DFS: percorso bianco

Teorema: Nella *foresta DF* di un grafo G , un vertice v è *discendente* del vertice u se e solo se al tempo $d[u]$ in cui la ricerca visita u , il vertice v può essere raggiunto da u lungo un *percorso composto da soli vertici bianchi*.

Dimostrazione:

solo se: Assumiamo che v sia discendente di u nella *foresta DF* e che w sia un arbitrario vertice nel percorso tra u e v nella *foresta DF*.

Allora anche w è discendente di u nella *foresta DF*.

Per il corollario precedente, $d[u] < d[w]$, quindi w è bianco al tempo $d[u]$

Proprietà di DFS: percorso bianco

Teorema: Nella *foresta DF* di un grafo G , un vertice v è *discendente* del vertice u se e solo se al tempo $d[u]$ in cui la ricerca visita u , il vertice v può essere raggiunto da u lungo un *percorso composto da soli vertici bianchi*.

Dimostrazione:

se: Assumiamo che v sia il *primo vertice raggiungibile* da u lungo un *percorso bianco* al tempo $d[u]$, ma che non diventi un discendente di u nell'*albero DF*.

Quindi tutti i vertici che precedono v nel percorso saranno discendenti di u .

Sia w il predecessore di v nel percorso (v è quindi adiacente a w).

Proprietà di DFS: percorso bianco

Teorema: Nella *foresta DF* di un grafo G , un vertice v è *discendente* del vertice u se e solo se al tempo $d[u]$ in cui la ricerca visita u , il vertice v può essere raggiunto da u lungo un *percorso composto da soli vertici bianchi*.

Dimostrazione:

se: per il *Corollario precedente*, abbiamo che $f[w] < f[u]$.

Poiché $v \in \text{Adiac}[w]$, la chiamata a *DFS-Visita*(w) garantisce che v venga visitato (e *terminato*) prima di w .

Perciò, $f[v] < f[w] < f[u]$.

Poiché quindi v è *bianco* al tempo $d[u]$, vale $d[u] < d[v]$, e il *Corollario precedente* garantisce che v deve essere un discendente di u nell'*albero DF*.

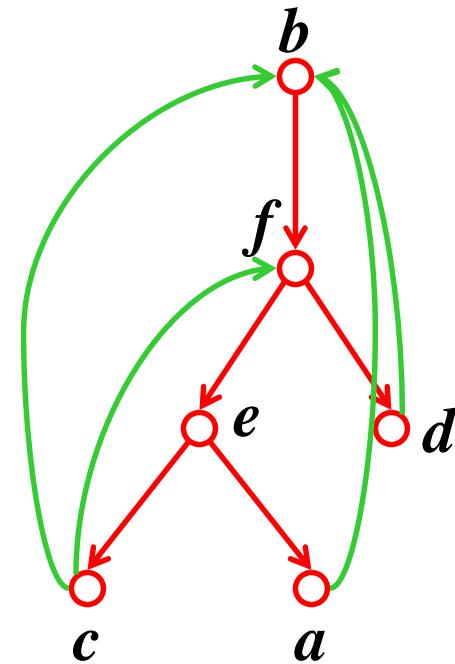
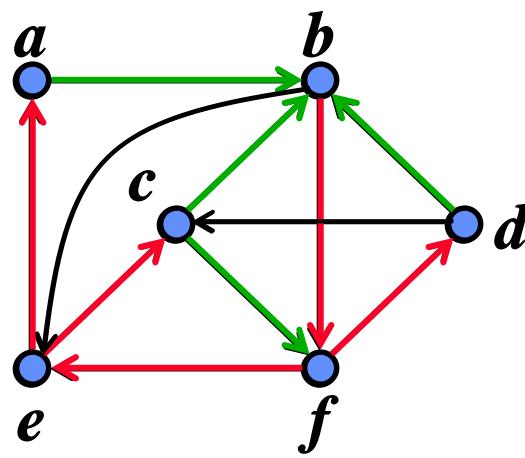
Algoritmi e Strutture Dati (Mod. B)

**Algoritmi su grafi
Ricerca in profondità
(Deepth-First Search) Parte II**

Classificazione degli archi

Sia G_π la foresta DF generata da DFS sul grafo G .

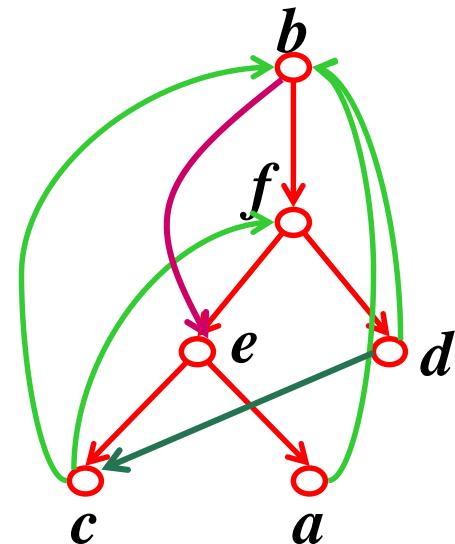
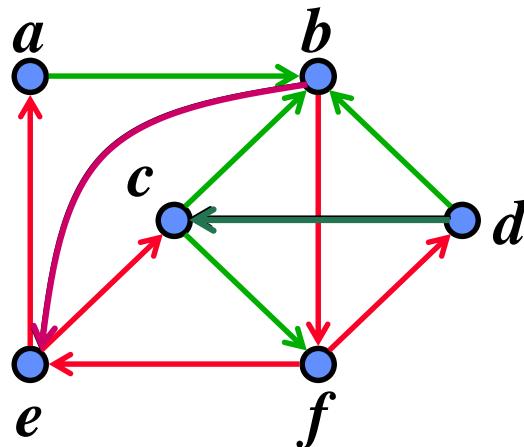
- **Arco d'albero:** gli archi della foresta G_π , tali che l'arco $(u,v) \in E_\pi$ se v è stato scoperto esplorando l'arco (u,v) .
- **Arco di ritorno:** gli archi (u,v) che connettono un vertice u con un *antenato* v nell'*albero DF*.



Classificazione degli archi

Sia G_π la *foresta DF* generata da DFS sul grafo G .

- *Arco in avanti*: archi (u,v) *non* appartenenti all'*albero DF* che connettono l'arco u con un *descendente* v
- *Arco di attraversamento (cross)*: tutti gli altri archi. Possono connettere *vertici nello stesso albero DF* (a patto che un vertice non sia antenato dell'altro nell'albero) o vertici in *alberi DF differenti*.

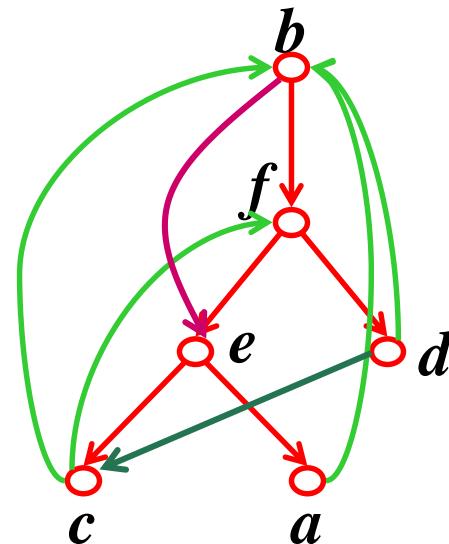
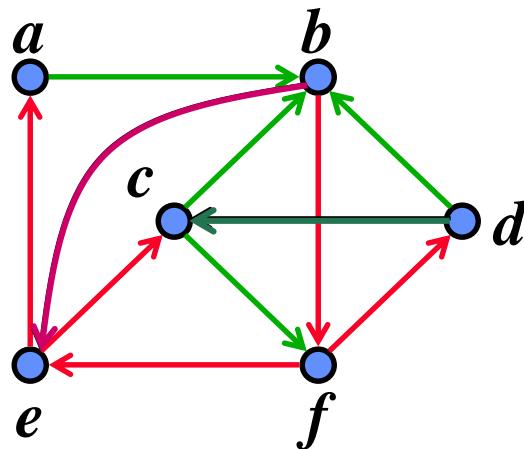


DFS per la classificazione degli archi

DFS può essere usata per classificare gli archi di un grafo G .

Si utilizza il colore del vertice che si raggiunge durante la visita dell'arco (u,v) :

- se v è *bianco*: allora l'arco è un *arco d'albero*
- se v è *grigio*: allora l'arco è un *arco di ritorno*
- se v è *nero*: allora l'arco è un *arco in avanti* o un *arco di attraversamento*

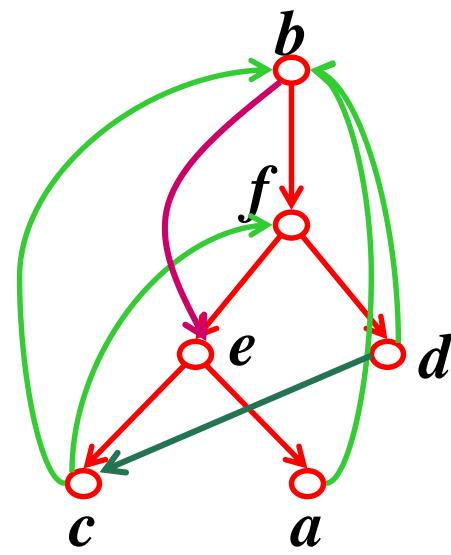
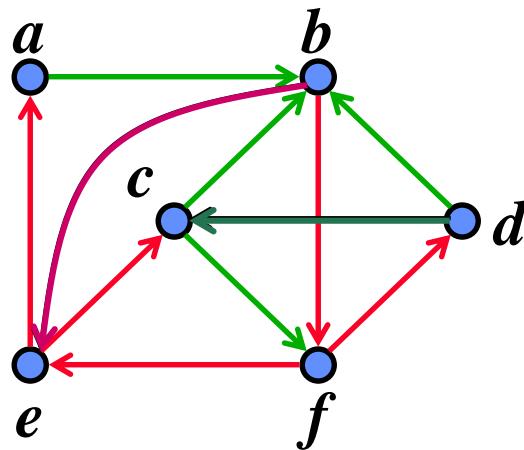


DFS per la classificazione degli archi

DFS può essere usata per classificare gli archi di un grafo G.

Si utilizza il colore del vertice che si raggiunge durante la visita dell'arco (u,v) :

- v è nero: allora l'arco è un *arco in avanti* o un *arco di attraversamento*
 - se inoltre $d[u] < d[v]$ allora è un *arco in avanti*
 - se $d[v] < d[u]$ allora è un *arco di attraversamento*



Proprietà di DFS

Teorema: Durante la DFS di un *grafo non orientato* G , ogni arco è un *arco dell'albero* o un *arco di ritorno*.

Dimostrazione: Sia (u,v) un arco arbitrario di G . Consideriamo il caso in cui $d[u] < d[v]$ (il caso $d[v] < d[u]$ è simmetrico).

Se $d[u] < d[v]$, v deve essere stato scoperto e visitato prima che si termini u (poiché v è nella lista di adiacenza di u).

Ora, se l'arco (u,v) viene esplorato prima nella direzione da u a v , allora diventa un *arco dell'albero*.

Se invece l'arco (u,v) viene esplorato prima nella direzione da v a u , allora l'arco (u,v) diventa un *arco di ritorno*, poiché u è già grigio quando l'arco viene attraversato.

Esercizi

Dal libro di testo:

- **Es. 23.1-3 (calcolo del grafo trasposto G^T di G)**
- **Es. 23.3-4**
- **Es. 23.3-6**
- **Es. 23.3-7**
- **Es. 23.3-8**

Applicazioni di DFS

Due problemi:

- calcolare l'*ordinamento topologico* indotto da un *grafo aciclico*.
- calcolare le *componenti (fortemente) connesse (CFC)* di un *grafo (non) orientato*.

Vedremo che entrambi i problemi possono essere risolti *impiegando* opportunamente l'*algoritmo* di *DFS*

Ordinamento topologico

Definizione: Dato un *grafo orientato aciclico* G (un *DAG*), un *ordinamento topologico* su G è un ordinamento lineare dei suoi vertici tale che:

- se G contiene l'arco (u,v) , allora u compare prima di v nell'ordinamento.

Ordinamento dei vertici in un *DAG* tale che

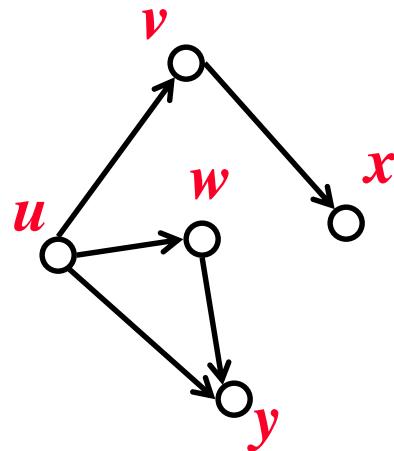
- più in generale, se esiste un *percorso* da u a v , allora u compare prima di v nell'ordinamento

Ordinamento topologico

Ordinamento dei vertici in un *DAG* tale che

- se esiste un percorso da u a v , allora u compare prima di v nell'ordinamento

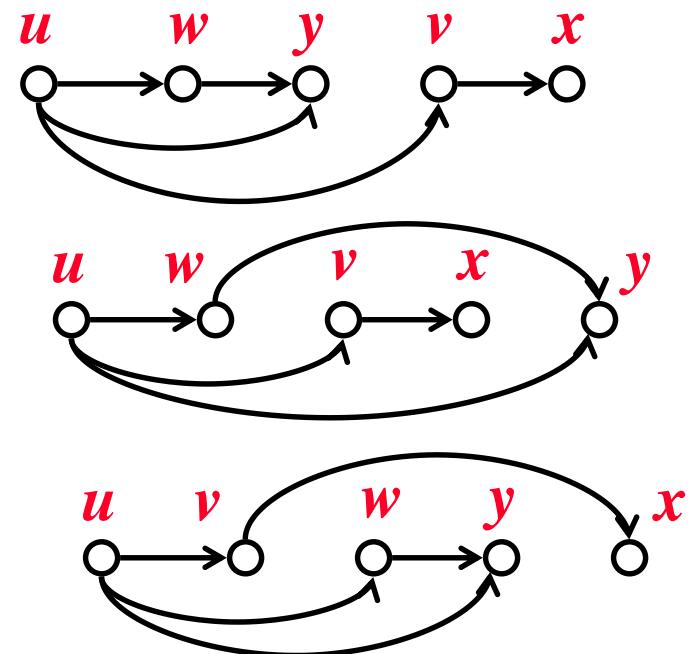
Ci possono essere *più ordinamenti topologici*.



$u \ w \ y \ v \ x$

$u \ w \ v \ x \ y$

$u \ v \ w \ y \ x$



Ordinamento topologico

Problema: Fornire un algoritmo che *dato un grafo orientato aciclico*, ne calcoli e ritorni un *ordinamento topologico*.

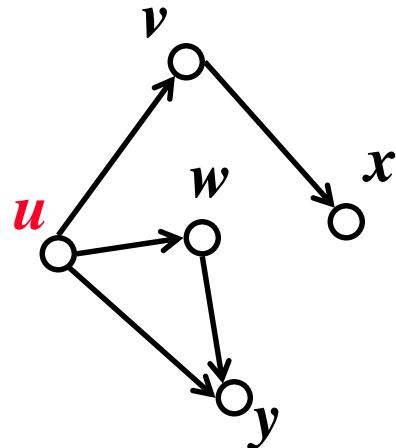
Soluzioni:

- Soluzione *diretta*
- Soluzione che utilizza *DFS*

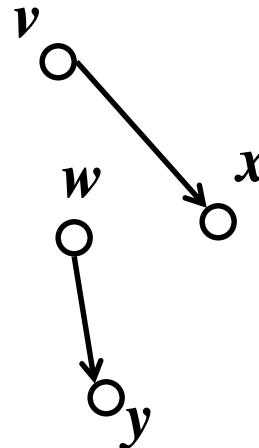
Ordinamento topologico: algoritmo I

- ➤ *Trovare* ogni *vertice* che *non* ha *alcun arco incidente* in ingresso
- Stampare questo vertice e “*rimuoverlo*” (virtualmente) insieme ai suoi archi
- *Ripetere* la procedura finché tutti vertici risultano “*rimossi*”.

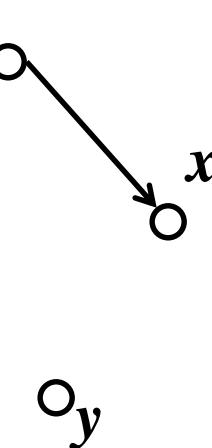
Ordinamento topologico: algoritmo I



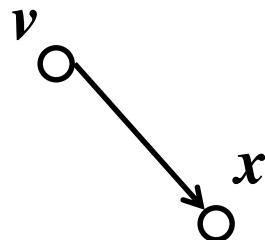
output:



output: **u**



output: **u w**



output: **u w y**



output: **u w y v**



output: **u w y v x**

Esercizio

Es 23.4-5: Terminare l'esercizio fornendo un *algoritmo* che in tempo $O(V+E)$ computa l'*Ordinamento Topologico* di un grafo G secondo l'idea appena illustrata.

Ordinamento topologico

Teorema: Un grafo orientato è *aciclico se e solo se DFS* su G *non* trova alcun *arco di ritorno*.

Dimostrazione:

se: Supponiamo che G contenga un ciclo c .

Allora *DFS* necessariamente troverà un *arco di ritorno*.

Infatti, sia v è il *primo* vertice che viene scoperto in c , e (u,v) l'arco del ciclo c che entra in v .

Poiché v è il *primo* vertice di c che viene scoperto, al tempo $d[v]$ c'è un percorso bianco da v a u (quello che segue c fino ad u).

Per il *teorema del percorso bianco*, u diventa un discendente di v nella *foresta DF*. Perciò, (u,v) deve essere un *arco di ritorno*.

Ordinamento topologico

Teorema: Un grafo orientato è *aciclico se e solo se DFS* su G *non* trova alcun *arco di ritorno*.

Dimostrazione:

solo se: Supponiamo che *DFS* incontri un *arco di ritorno* (u,v) .

Allora il vertice v è un *antenato* di u nella *foresta DF*.
Quindi esiste certamente un percorso che va da v a u nel grafo G .

Tale percorso, concatenato con l'*arco di ritorno* (u,v) , forma un ciclo, quindi il grafo G *non è aciclico*.

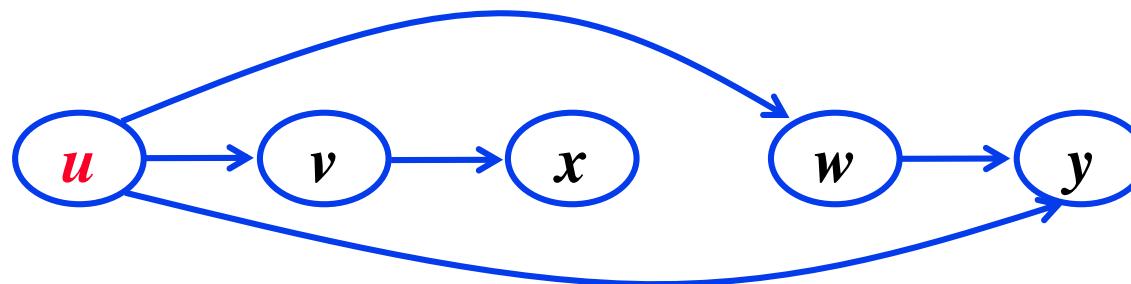
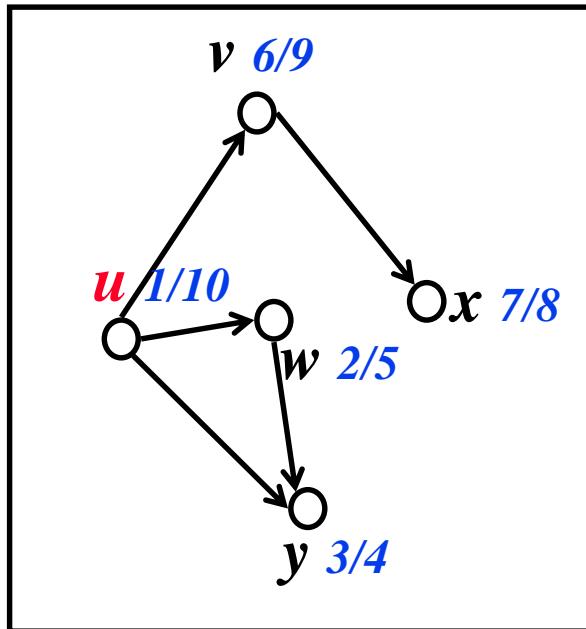
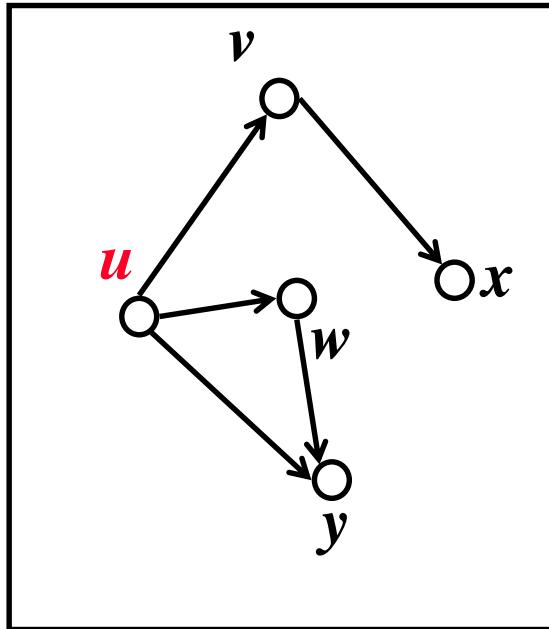
Ordinamento topologico: algoritmo II

Ordinamento-Topologico (G : grafo)

- 1 $DFS(G)$ per calcolare i tempi $f[v]$
- 2 Durente la DFS , ogni volta che un vertice è terminato, *aggiungerlo* in testa ad una *lista*
- 3 Ritornare la *lista* di vertici

In altre parole, *l'algoritmo costruisce una lista di vertici seguendo l'ordine inverso dei tempi di fine visita.*

Ordinamento topologico: algoritmo II



output: $u \ v \ x \ w \ y$

Correttezza dell'algoritmo II

Teorema: *Ordinamento-Topologico(G)* calcola correttamente l'ordinamento topologico di un grafo aciclico G .

Dimostrazione: *Dobbiamo dimostrare che vale la proprietà di ordinamento topologico: per ogni arco $(u,v) \in E$, u precede v nell'ordinamento.*

Ma questo equivale a dimostrare che, *dopo la DFS, per ogni coppia di vertici u e v , se abbiamo che $(u,v) \in E$, allora $f[v] < f[u]$.*

In tal caso abbiamo appunto che u precederà v nell'ordinamento (vedi Algoritmo).

Correttezza dell'algoritmo II

Teorema: *Ordinamento-Topologico(G)* calcola correttamente l'ordinamento topologico di un grafo aciclico G .

Dimostrazione: Dimostriamo che, dopo *DFS*, per ogni coppia di vertici u e v , se $(u,v) \in E$, allora $f[v] < f[u]$.

Preso un qualsiasi arco $(u,v) \in E$ esplorato da *DFS*, quando l'arco viene esplorato, v non può essere grigio,

altrimenti v sarebbe un antenato di u e (u,v) un *arco di ritorno*, contraddicendo il *teorema precedente*.

Quindi v o è bianco o è nero.

Correttezza dell'algoritmo II

Teorema: *Ordinamento-Topologico(G)* produce correttamente l'ordinamento topologico di un grafo aciclico G .

Dimostrazione: il vertice v è o bianco o nero.

- a) Se v è bianco, allora diventa un *descendente* di u e $f[v] < f[u]$ (per il teorema della Struttura a parentesi)
- b) Se v è nero, allora ovviamente sarà $f[v] < f[u]$.

In conclusione, dato l'ordine di inserimento nella lista e l'aciclicità di G , segue la *correttezza*.

Algoritmi e Strutture Dati (Mod. B)

Algoritmi su grafi
Ricerca in profondità
(Deepth-First Search) Parte III

Applicazioni di DFS

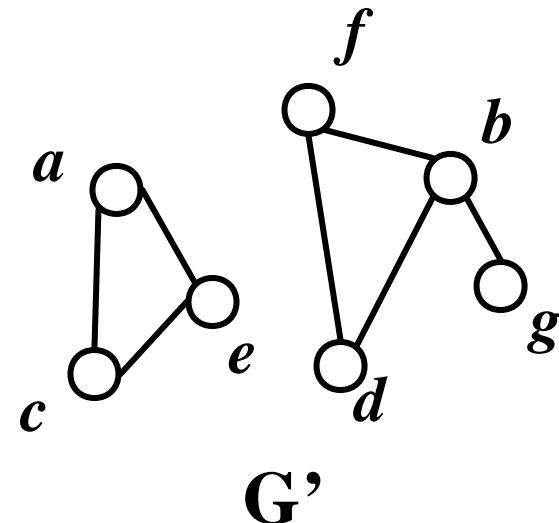
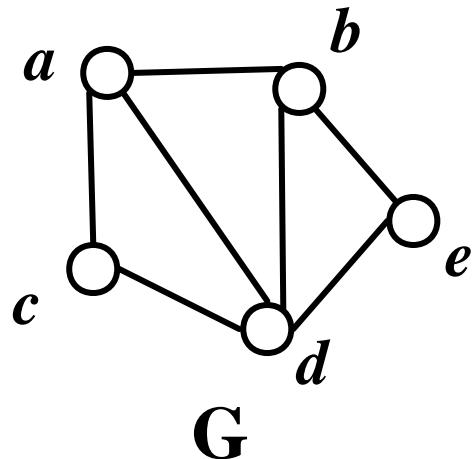
Due problemi:

- ✓ calcolare l'*ordinamento topologico* indotto da un *grafo aciclico*.
- calcolare le *componenti (fortemente) connesse (CFC)* di un *grafo (non) orientato*.

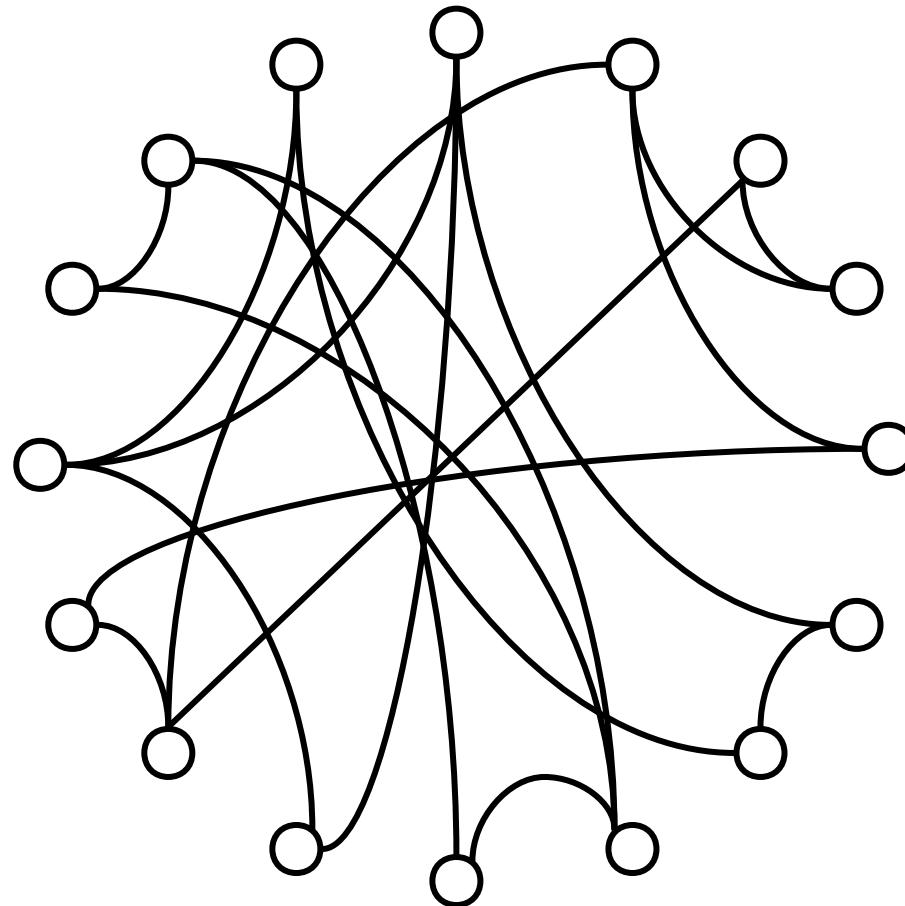
Vedremo che entrambi i problemi possono essere risolti *impiegando* opportunamente l'*algoritmo* di *DFS*

Connettività

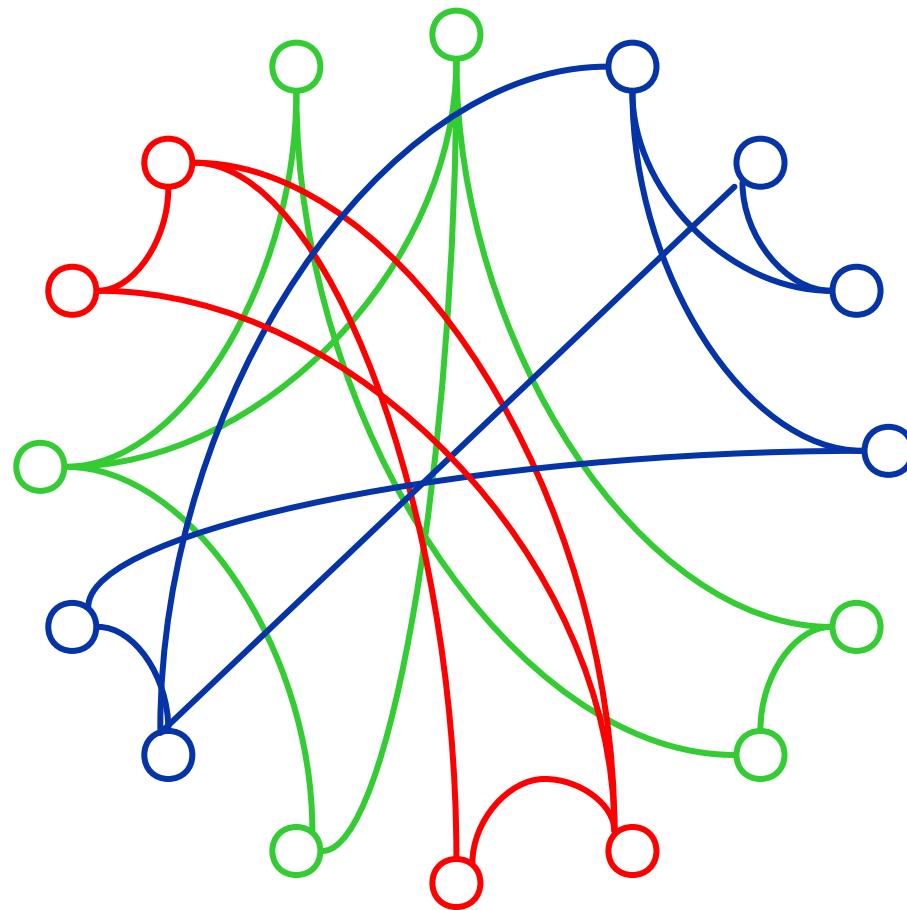
- Un vertice u si dice *connesso* ad un vertice v in un grafo G se esiste un percorso da u a v in G .
- Un grafo non orientato (un *grafo orientato*) G si dice (*fortemente*) *connesso* se ogni coppia di vertici è *connessa*.
- Altrimenti, G è *sconnesso*.



Verifica della connettività



Verifica della connettività



Verifica della connettività

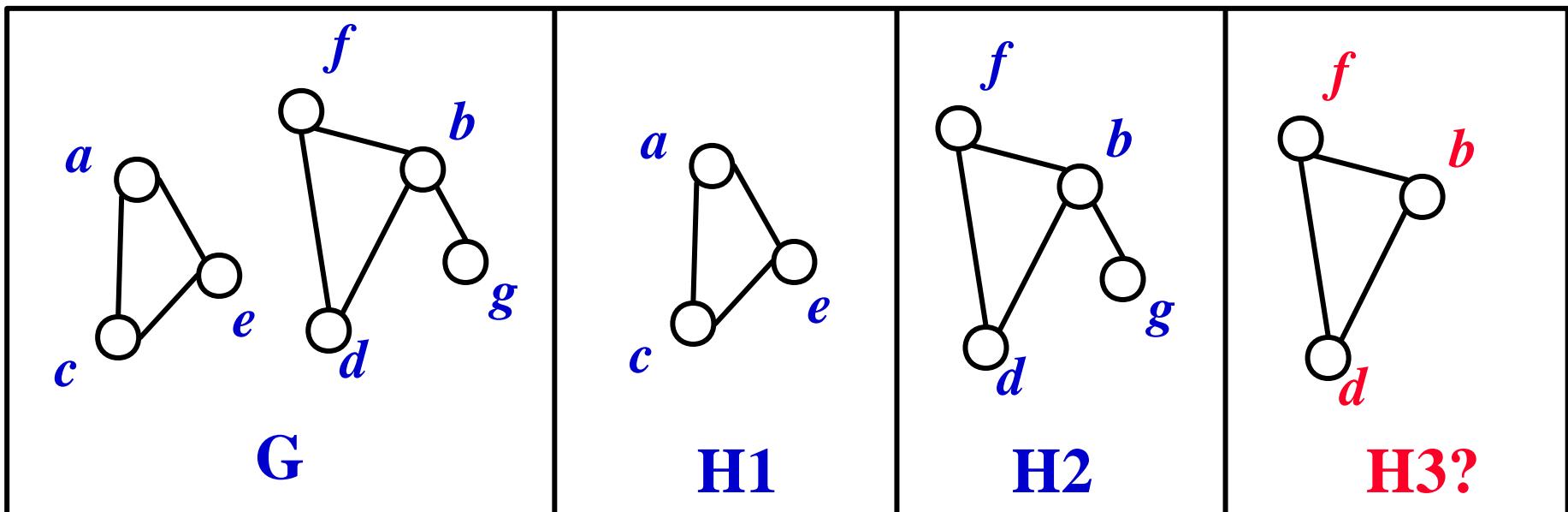
Per verificare se un *grafo non orientato* è *connesso* possiamo:

- usare l'algoritmo *DFS*
 - il grafo è connesso *se e solo se* una chiamata di *DFS* raggiungerà tutti i vertici.
-
- Se al termine della *DFS* c'è *più* di un vertice u con $\text{pred}[u] = \text{NIL}$ il grafo *non può* essere *connesso*.

Componenti connesse

Un sottografo massimale connesso di un *grafo non orientato G* si dice **componente连通子图** di *G*.

- Un *sottografo connesso H di G* è “massimale” se
 - non si possono aggiungere ad *H* altri vertici o archi
 - in modo che l’*H* risultante sia ancora un sottografo connesso di *G*.



Verifica della connettività

Per calcolare le *componenti connesse* un *grafo non orientato* possiamo usare l'algoritmo *DFS*:

- Ogni chiamata esterna (non ricorsiva) a *DFS-Visita* raggiungerà tutti i vertici contenuti *esattamente* in una *componente连通子图*. *Perché?*
- Le *componenti connesse* sono pari al numero di vertici u per cui, al termine di una *DFS* sul grafo, $\text{pred}[u] = \text{NIL}$.

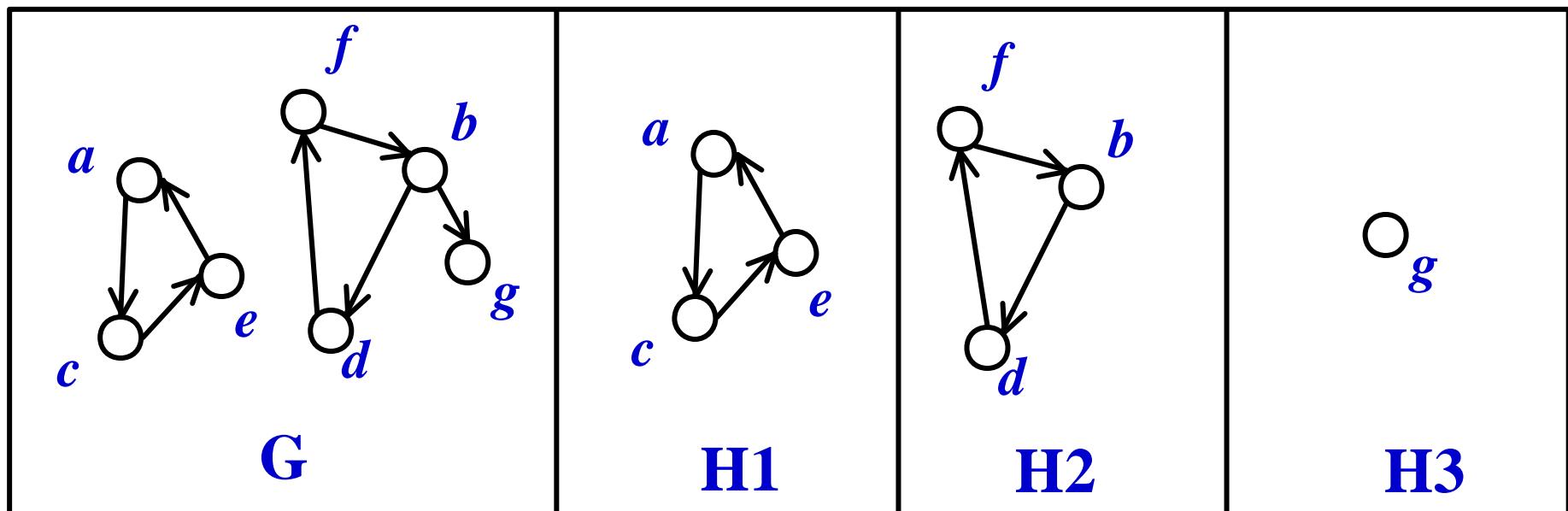
Esercizio

Es. 23.3.9: Mostrare che DFS su un *grafo non orientato* G può essere usata per identificare le Componenti Connesse e che la *foresta DF* contiene tanti *alberi* quante CC . Modificare DFS in modo che ogni vertice sia etichettato con $cc[v]$ tra 1 e k (k numero di CC) in modo che $cc[u]=cc[v]$ se e solo se u e v se appartengono alla stessa CC .

Componenti fortemente connesse

Un sottografo massimale fortemente connesso di un grafo orientato G si dice **componente fortemente connessa (CFC)**.

- Un *sottografo fortemente connesso H di G* è **massimale** se
 - non si possono aggiungere ad *H* altri vertici o archi
 - in modo che l'*H'* risultante sia ancora un sottografo fortemente connesso di *G*.



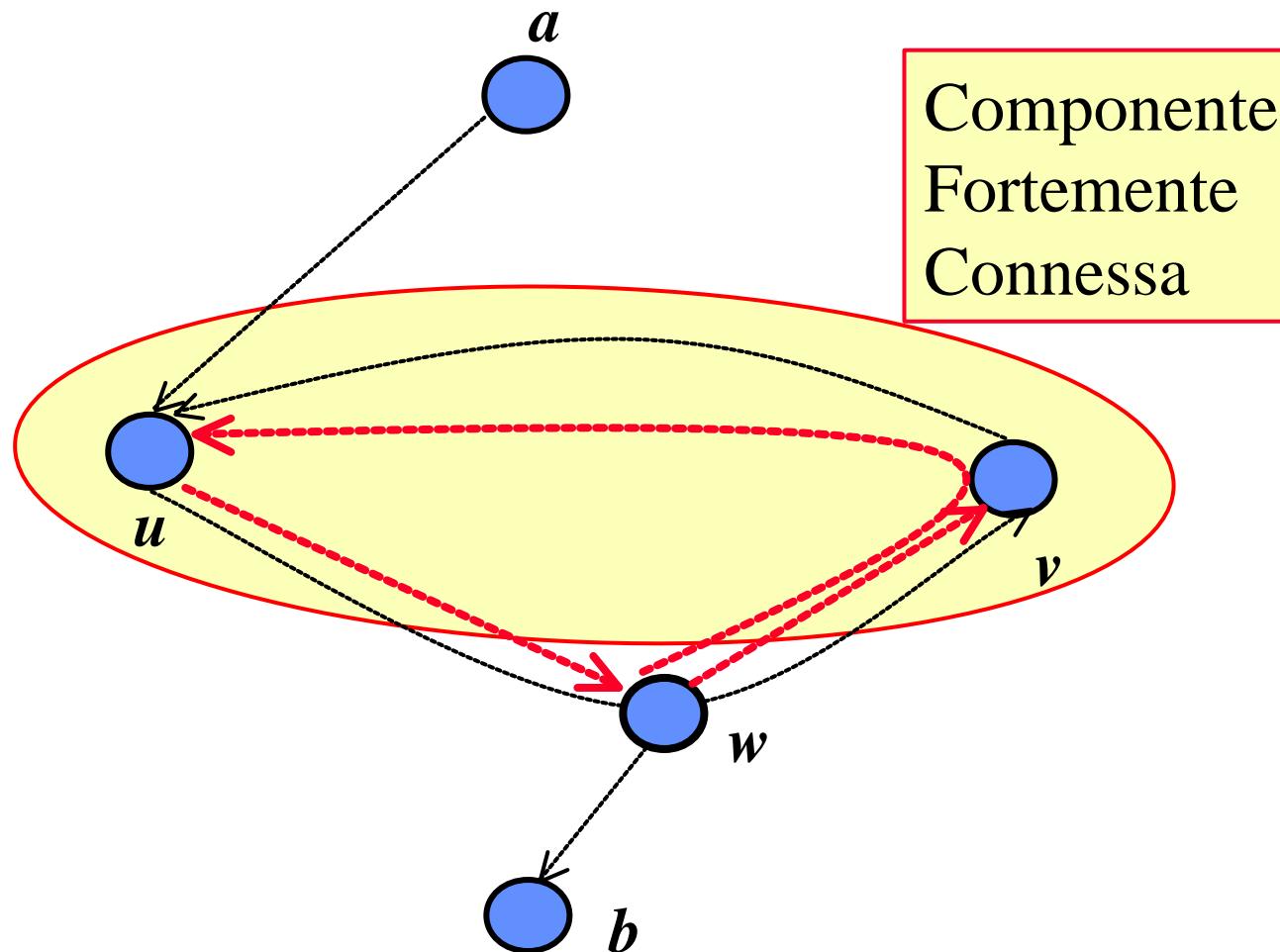
Proprietà delle CFC

Teorema 1: Se due vertici compaiono nella stessa *componente fortemente connessa*, allora *nessun percorso* tra i due vertici *esce* da quella *componente*.

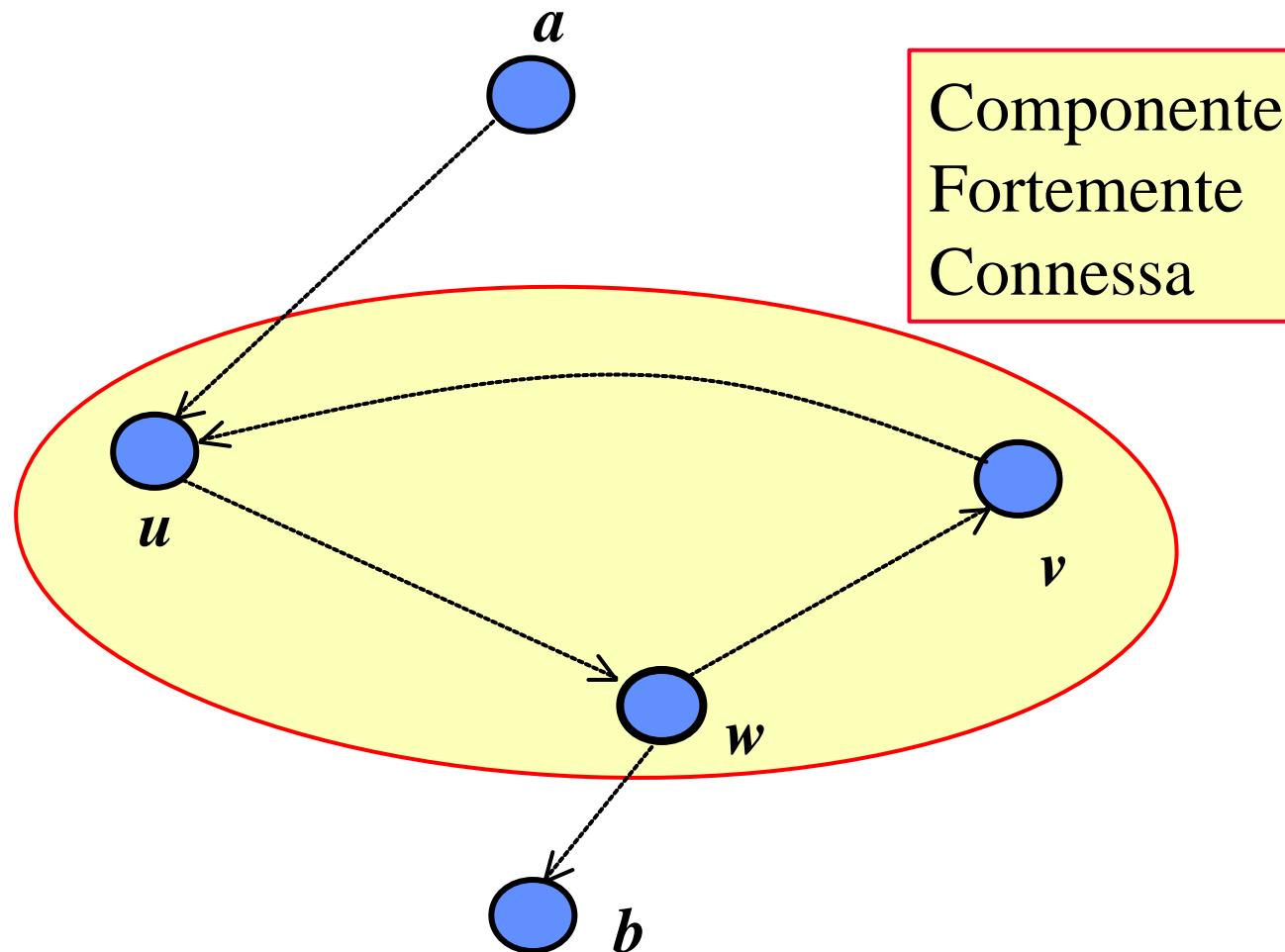
Dimostrazione: Siano u e v due vertici nella stessa **componente fortemente connessa**.

- Esistono percorsi sia da v a u che da u a v . Sia w un vertice lungo qualche percorso $v \circ w \circ u$.
- Poiché c'è un percorso $v \circ u$, u è raggiungibile da w tramite $w \circ v \circ u$. Quindi w e u sono nella stessa **componente fortemente connessa**.
- Poiché w è stato scelto arbitrario, il teorema segue.

Proprietà delle CFC



Proprietà delle CFC



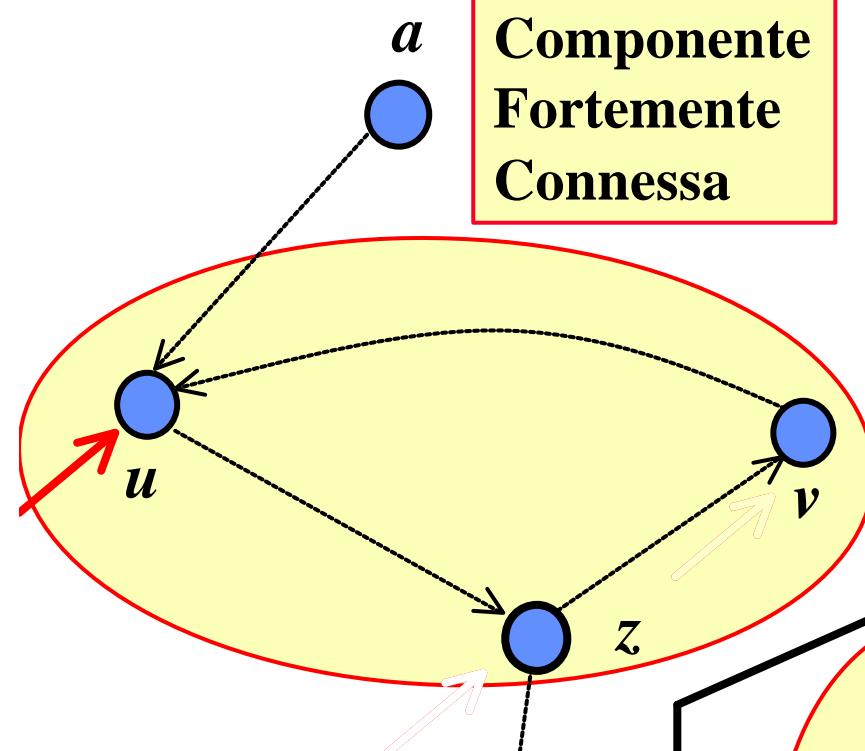
Proprietà delle CFC

Teorema 2: In ogni DFS, tutti i vertici nella *stessa componente fortemente connessa* compaiono nello stesso albero DF.

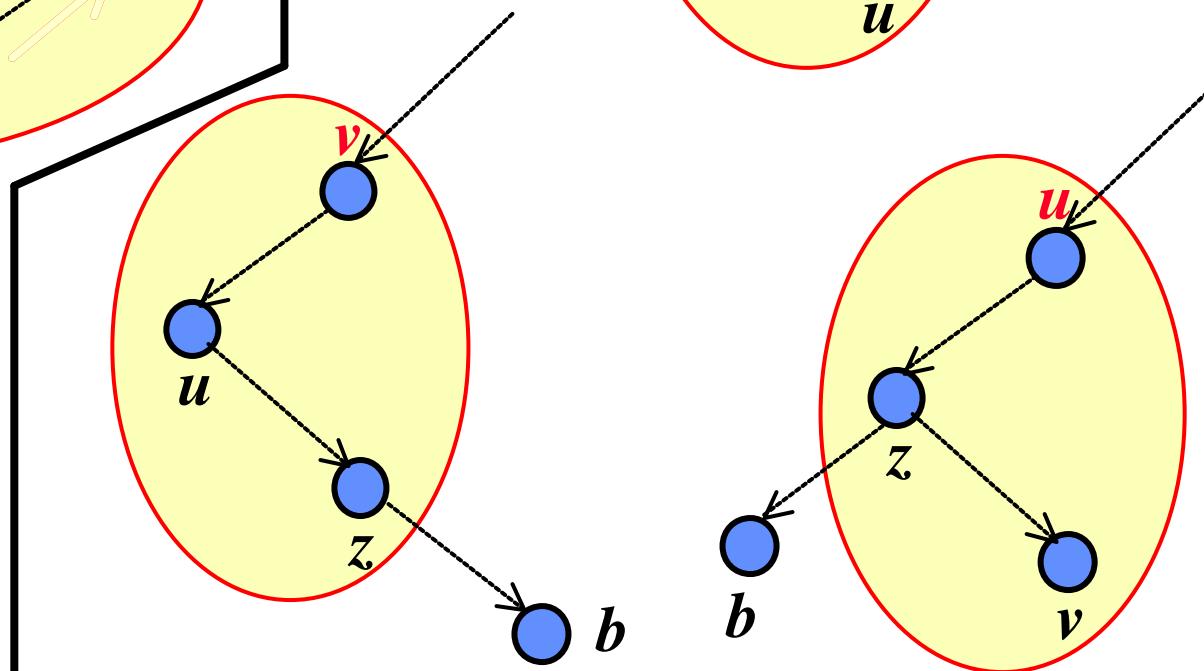
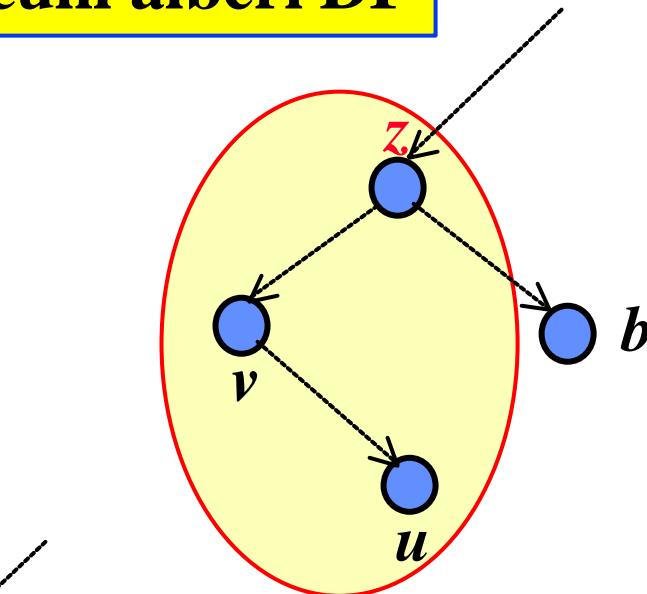
Dimostrazione: Sia r il primo vertice di una componente fortemente connessa (CFC), visitato da DFS.

- Poiché è il primo, tutti gli altri vertici nella CFC devono essere ancora bianchi.
- Esiste quindi un percorso da r a tutti gli altri vertici nella CFC... (*perché?*)...
- ...infatti questi percorsi non escono mai dalla CFC di r (per il *teorema precedente*) e i vertici di tutti i percorsi nella CFC sono bianchi.
- Quindi per il *teorema del percorso bianco*, ogni vertice nella CFC sarà un discendente di r nell'*albero DF*.

Proprietà delle CFC



Alcuni alberi DF



Proprietà delle CFC

Definizione: Dato un vertice u di un grafo G , l'*avvo di u* , in simboli $f(u)$, è il vertice w raggiungibile da u che viene terminato per ultimo in una *DFS* del grafo G , cioè:

$f(u) = w$ tale che $u \xrightarrow[p]{} w$ e $f[w]$ è massimo tra i vertici raggiungibili da u

- Notate che è possibile che sia $f(u) = u$, perché u è raggiungibile da se stesso e quindi vale anche $f[u] \leq f[f(u)]$
- Inoltre si può dimostrare che $f(f(u)) = f(u)$

Proprietà delle CFC

Si può dimostrare che $f(f(u)) = f(u)$

- $u \xrightarrow{p^R} v$ implica che $f[f(v)] \leq f[f(u)]$
- infatti l'insieme $\{w : v \xrightarrow{p^R} w\} \subseteq \{w : u \xrightarrow{p^R} w\}$, e il tempo di terminazione ($f[]$) dell'*avo* è il **massimo** tra tutti i vertici raggiungibili.
- ma poiché $u \xrightarrow{p^R} f(u)$ allora $f[f(f(u))] \leq f[f(u)]$
- e per quello che abbiamo visto nel lucido precedente vale anche $f[f(u)] \leq f[f(f(u))]$
- quindi risulta che $f[f(u)] = f[f(f(u))]$
- ma allora $f(f(u)) = f(u)$, perché due vertici con lo stesso tempo di terminazione in **DFS** non possono che essere lo stesso vertice.

Proprietà delle CFC

Teorema 3: In un grafo orientato G , l'avo $f(u)$ di un qualsiasi vertice u è un *antenato* di u nell'albero DF di G .

Dimostrazione: Se $f(u) = u$ il teorema è banalmente vero.

- Se $f(u) \neq u$, poiché $u \rightarrow_p f(u)$, consideriamo il colore del vertice $f(u)$ al tempo $d[u]$:
 - se $f(u)$ è nero, allora $f[f(u)] < f[u]$, contraddicendo il fatto che deve essere $f[u] \leq f[f(u)]$, per definizione.
 - se $f(u)$ è grigio, allora $f(u)$ è un *antenato* di u .

Dimostriamo ora che $f(u)$ non può essere bianco

Proprietà delle CFC

Teorema 3: In un grafo orientato G , il *avo* $f(u)$ di un qualsiasi vertice u in un *albero* DF di G è un *antenato* di u .

Dimostrazione: $f(u)$ non può essere bianco

Due casi (ricordate che $u \xrightarrow{p^{\text{R}}} f(u)$):

1. Ogni vertice intermedio tra u e $f(u)$ è bianco
2. Qualche vertice intermedio tra u e $f(u)$ non è bianco

Proprietà delle CFC

Teorema 3: In un grafo orientato G , il *avo* $f(u)$ di un qualsiasi vertice u in un *albero* DF di G è un *antenato* di u .

Dimostrazione: $f(u)$ non è bianco

1. Ogni vertice intermedio tra u e $f(u)$ è bianco allora $f(u)$ sarà un discendente di u per il teorema del percorso bianco.

Questo significa però anche che $f[f(u)] < f[u]$, e ciò contraddice la definizione di $f(u)$.

Proprietà delle CFC

Teorema 3: In un grafo orientato G , il *avo* $f(u)$ di un qualsiasi vertice u in un *albero* DF di G è un *antenato* di u .

Dimostrazione: $f(u)$ non è bianco

2. Qualche vertice intermedio tra u e $f(u)$ non è bianco sia t l'ultimo vertice non bianco nel percorso tra u e $f(u)$. Allora t deve essere grigio (non ci sono archi da vertici neri a vertici bianchi) e il successore di t è bianco. Ma allora c'è un percorso bianco tra t e $f(u)$ al tempo $d[u]$, e quindi anche al tempo $d[t]$ tale percorso bianco esisterà (*Perché?*). Ma allora $f(u)$ sarà un discendente di t per il *Teorema del Percorso Bianco*.

Quindi $f[t] > f[f(u)]$, contraddicendo la scelta di $f(u)$.

Proprietà delle CFC

Corollario: Durante una *DFS* di un grafo orientato G , per ogni vertice u , i u e $f(u)$ appartengono alla stessa *CFC*.

Dimostrazione: Per definizione di avo, abbiamo che $u \xrightarrow{p^R} f(u)$.

Ma poiché $f(u)$ è un antenato di u nella *foresta DF (teorema precedente)*, sappiamo anche che vale $f(u) \xrightarrow{p^R} u$.

In conclusione, entrambi i vertici, essendo mutuamente raggiungibili, *devono* stare nella stessa *CFC*.

Proprietà delle CFC

Teorema 4: In un grafo orientato G , due vertici u e v , compaiono nella stessa CFC se e solo se hanno lo stesso *avo* nella DFS di G .

Dimostrazione:

- *solo se*: Assumiamo u e v siano nella stessa CFC .
 - Ogni vertice raggiungibile da u è anche raggiungibile da v e vice versa.
 - Dalla definizione di *avo* segue che $f(u)=f(v)$ infatti, $u \xrightarrow{p} v$ implica che $f[f(u)] \geq f[f(v)]$ mentre $v \xrightarrow{p} u$ implica che $f[f(v)] \geq f[f(u)]$.

Quindi $f[f(v)] = f[f(u)]$.

Proprietà delle CFC

Teorema 4: In un grafo orientato G , due vertici u e v , compaiono nella stessa CFC *se e solo se* hanno lo stesso *avo* nella DFS di G .

Dimostrazione:

- *se*: Assumiamo ora $f(u) = f(v)$.
 - Per il *teorema 4*, u compare nella stessa CFC di $f(u)$, e v compare nella stessa CFC di $f(v)$
 - quindi, ovviamente u e v compaiono nella stessa CFC.

Calcolo delle CFC

Possiamo quindi concludere che:

- Le **CFC** sono insiemi di vertici che hanno lo *stesso avo*.
- Durante **DFS**, l'*avo* $f(u)$ di un vertice u è sia il *primo vertice scoperto* (visitato) che l'*ultimo vertice terminato* (processato) nella **CFC** contenente u (dal *teorema del percorso bianco* e dal *teorema 4*)
- L'*ultimo vertice terminato* r in una **DFS** è certamente un *avo*. Infatti, è almeno l'*avo* di se stesso poiché nessun altro vertice nell'albero ha *tempo di terminazione* maggiore di r .

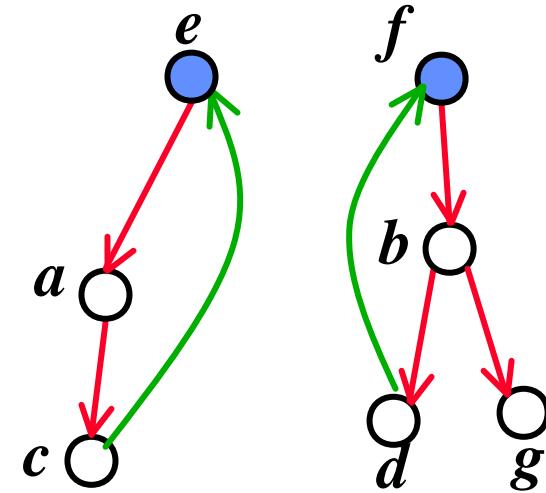
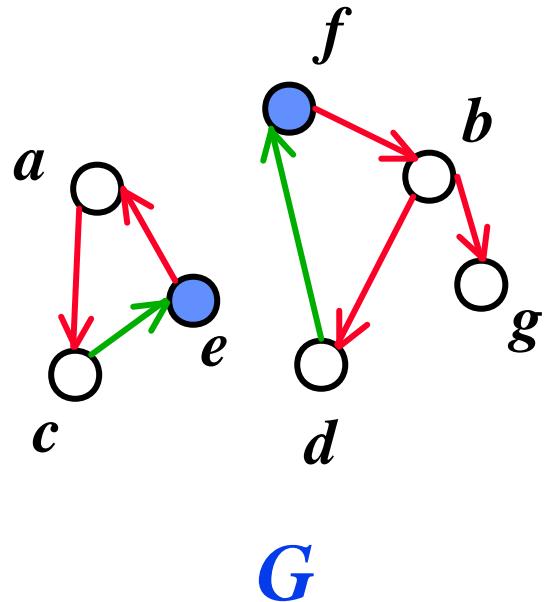
Calcolo delle CFC

- L'ultimo vertice terminato r è certamente un *avo*. Infatti, r è *avo* di se stesso, nessun altro vertice nell'albero ha *tempo di terminazione* maggiore.
- In generale, quali sono i vertici della *CFC* di un *avo* z ?
 - sono tutti quelli che hanno z come *avo*, cioè: tutti quei vertici che possono raggiungere z , ma nessun altro vertice con tempo di terminazione maggiore di z .
- Se r è il vertice con il *massimo valore di $f[r]$* , allora ci basta cercare i vertici che lo possono raggiungere.

Calcolo delle CFC

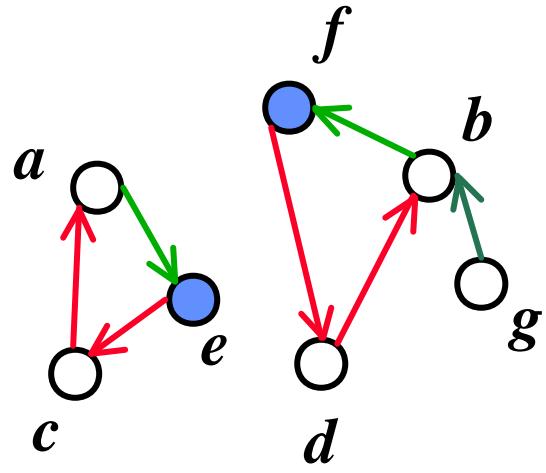
- Se r è il vertice con il *massimo valore* di $f[r]$, allora ci basta cercare i vertici che lo possono raggiungere.
- Ma, dalla definizione di grafo trasposto G^T di G , questi vertici sono proprio i *vertici raggiungibili* da r nel *grafo trasposto* G^T . Questi si possono ottenere *con una seconda DFS* su G^T a partire da r .
- Lo stesso procedimento viene ripetuto con tutti i vertici del grafo non raggiunti al passo precedente, scegliendo i vertici in *ordine decrescente di tempo di terminazione della prima DFS* (prima quelli terminati più tardi *durante la prima DFS*).

Calcolo delle CFC

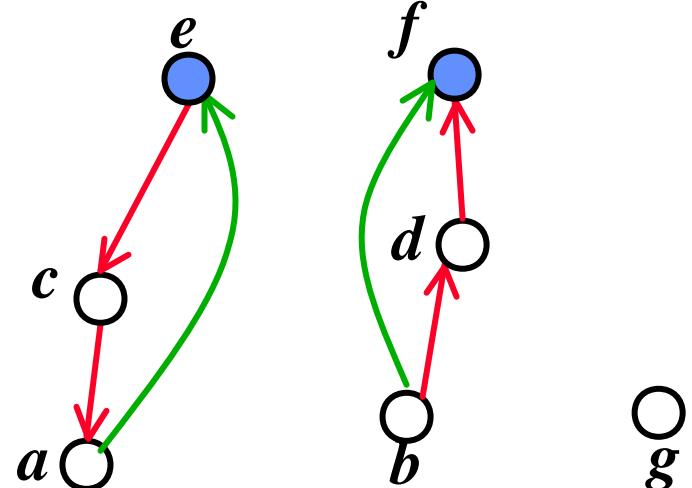


$$f[e] > f[a] > f[c] > f[f] > f[b] > f[g] > f[d]$$

Calcolo delle CFC



Trasposto G^T



CFC_1

CFC_2

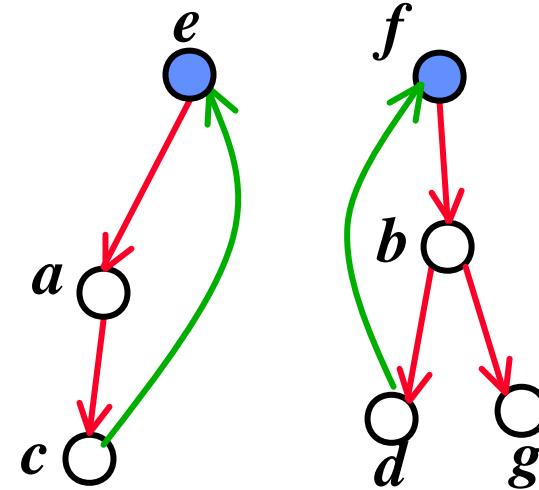
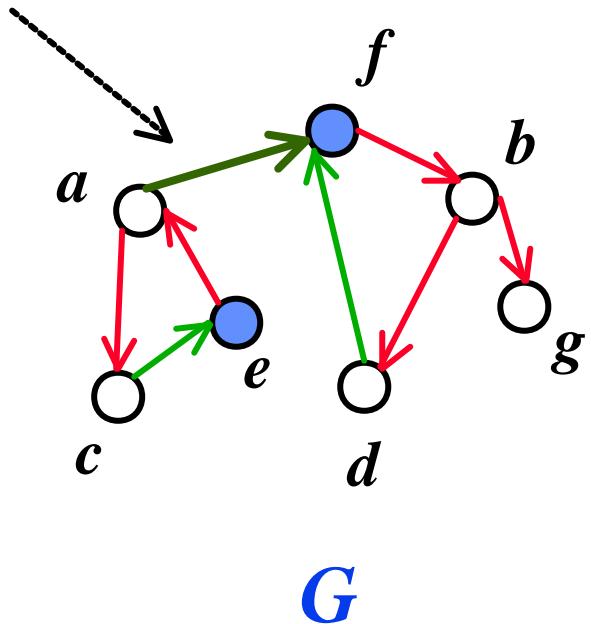
CFC_3

$$f[e] > f[a] > f[c] > f[f] > f[b] > f[g] > f[d]$$

$$f[f] > f[b] > f[g] > f[d]$$

$$f[g] > f[d]$$

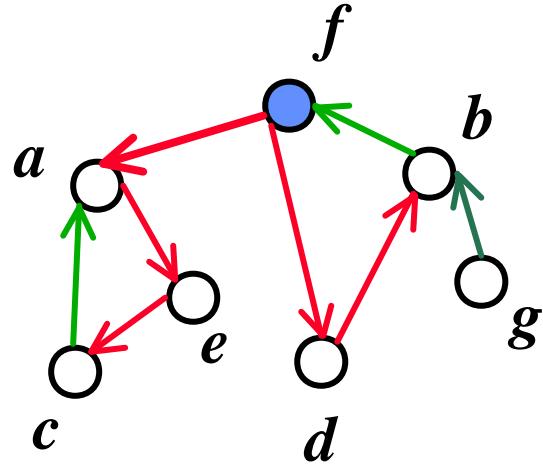
Calcolo delle CFC



$$f[e] > f[a] > f[c] > f[f] > f[b] > f[g] > f[d]$$

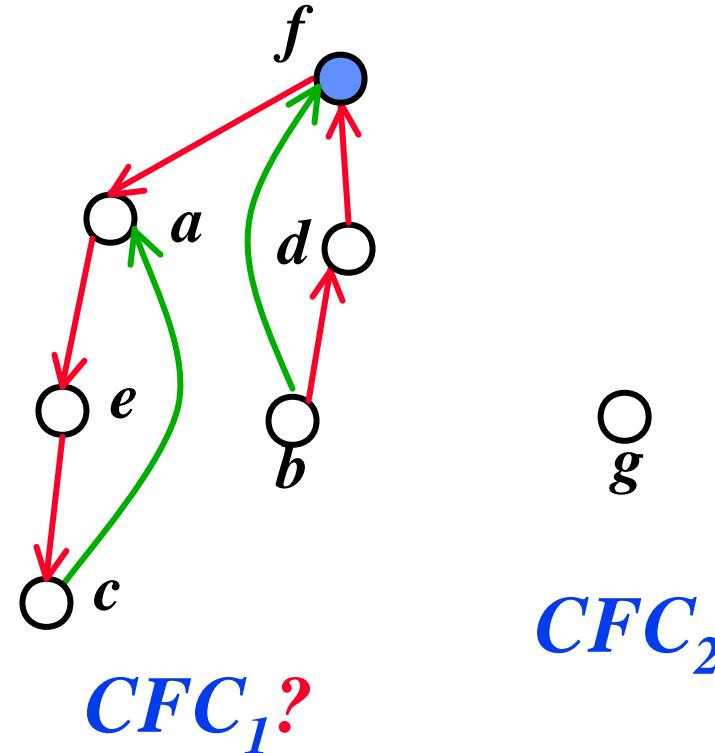
Supponiamo di non rispettare l'ordine decrescente e di scegliere il nodo f per primo.

Calcolo delle CFC



Trasposto G^T

Il risultato è scorretto!



Supponiamo di non rispettare l'ordine decrescente e di scegliere il nodo f per primo.

Ci basta cercare i vertici che lo possono raggiungere.

Algoritmo per il calcolo delle CFC

- 1 $DFS(G)$
- 2 Calcolare il *grafo trasposto* G^T
- 3 $DFS(G^T)$ ma esaminando i vertici in ordine decrescente di tempo $f[v]$ di fine visita
- 4 fornire i vertici di ogni albero della *foresta DF* prodotta al passo 3 come una *diversa CFC*

Correttezza di CFC(G)

Teorema: CFC(G) calcola correttamente le *componenti fortemente connesse* del grafo orientato G .

Dimostrazione: Per induzione sul numero di *alberi DF* trovati durante la *DFS* di G^T , dimostriamo che i vertici di ogni *albero DF* formano una CFC.

Dimostriamo che u è aggiunto a T **sse** $u \in \{v \in V : f(v)=r\}$.

Passo Base: inizialmente non ci sono alberi precedenti. Il verso **se** $u \in \{v \in V : f(v)=r\}$ **u allora è aggiunto a** T , discende immediatamente dal fatto che G e G^T hanno le stesse CFC unitamente al **Teorema 2**.

L'**altro verso** discende dal fatto che r è la radice dell'**ultimo albero della prima DFS**, quindi $f[r]$ è massimo. Inoltre, se u viene aggiunto a T , allora c'è un percorso da u a r in G e, poiché $f[r]$ è massimo, chiaramente $f(u)=r$ (per def. di avo).

Correttezza di CFC(G)

Teorema: CFC(G) calcola correttamente le *componenti fortemente connesse* del grafo orientato G .

Dimostrazione:

Passo Induttivo: Consideriamo l' n -esimo albero DF T con radice r prodotto da **DFS** su G^T e sia $C(r) = \{v \in V : f(v) = r\}$.

Dimostriamo che u è aggiunto a T se e solo se $u \in C(r)$.

se: Per il **Teorema 2** (sotto), ogni vertice in $C(r)$ viene messo nello stesso *albero DF* dalla prima **DFS**.

Poiché $r \in C(r)$, e r è la radice del nuovo *albero DF*, ogni elemento di $C(r)$ verrà messo in T dalla **DFS** su G^T

Teorema 2: In ogni **DFS**, tutti i vertici nella stessa CFC compaiono nello stesso *albero DF*.

Correttezza di $CFC(G)$

Teorema: $CFC(G)$ calcola correttamente le *componenti fortemente connesse* del grafo orientato G .

Dimostrazione: Dimostriamo che u è aggiunto T se e solo se $u \in C(r)$.

solo se: Dimostriamo (per contrapposizione) che se per un vertice w vale $f[f(w)] < f[r]$ o $f[f(w)] < f[r]$, allora w non viene aggiunto a T .

Per ipotesi induttiva, ogni w tale che $f[f(w)] > f[r]$ non può essere messo in T , poiché quando r è selezionato dal ciclo in DFS , w è già stato messo nell'albero con radice $f(w)$.

Ogni w tale che $f[f(w)] < f[r]$ non può essere posto in T , se così fosse, allora $w \rightarrow_p^{\circ} r$ e dalla formula (1) e da $r = f(r)$ segue che $f[f(w)] \geq f[f(r)] = f[r]$, ma ciò contraddice $f[f(w)] < f[r]$.

$$u \rightarrow_p^{\circ} v \text{ implica che } f[f(v)] \leq f[f(u)] \quad (1)$$

Correttezza di $CFC(G)$

Teorema: $CFC(G)$ calcola correttamente le componenti fortemente connesse del grafo orientato G .

Dimostrazione: Dimostriamo che u è aggiunto a T se e solo se $u \in C(r)$.

solo se: Ogni w tale che $f[f(w)] < f[r]$ non può essere posto in T , se così fosse, allora $w \circledR r$ e dalla formula (1) e da $r = f(r)$ segue che $f[f(w)]^3 = f[f(r)] = f[r]$, che contraddice $f[f(w)] < f[r]$.

Quindi, T contiene solo quei vertici u per i quali $f[f(u)] = r$.

Cioè, T è proprio uguale alla componente fortemente connessa $C(r)$, e ciò completa la dimostrazione

Esercizi su CFC

Dal libro di testo:

Esercizio 23.5-4

Esercizio 23.5-5

Algoritmi e Strutture Dati

Alberi di Ricerca
Alberi Bilanciati I (Alberi AVL)

Alberi bilanciati di ricerca

- Gli *alberi binari di ricerca* sono semplici da gestire (inserimenti e cancellazioni facili da implementare) ma hanno prestazioni poco prevedibili e potenzialmente basse
- Gli *alberi perfettamente bilanciati* hanno prestazioni ottimali ($\log N$ garantito) ma inserimenti e cancellazioni complesse (ribilanciamenti)
- Alberi AVL (*Adelson-Velskii e Landis*): *classe di alberi bilanciati* (non ottimale come la precedente). Hanno buone prestazioni e gestione relativamente semplice.

Alberi AVL: definizione

Definizione: Un albero binario di ricerca è un Albero AVL se per ogni nodo **x**:

- l'*altezza* del *sottoalbero sinistro di x* e quella del *sottoalbero destro di x* differiscono al più di uno, e

NOTA: In ogni albero AVL entrambi i sottoalberi sinistro e destro di un qualsiasi nodo sono *alberi AVL*.

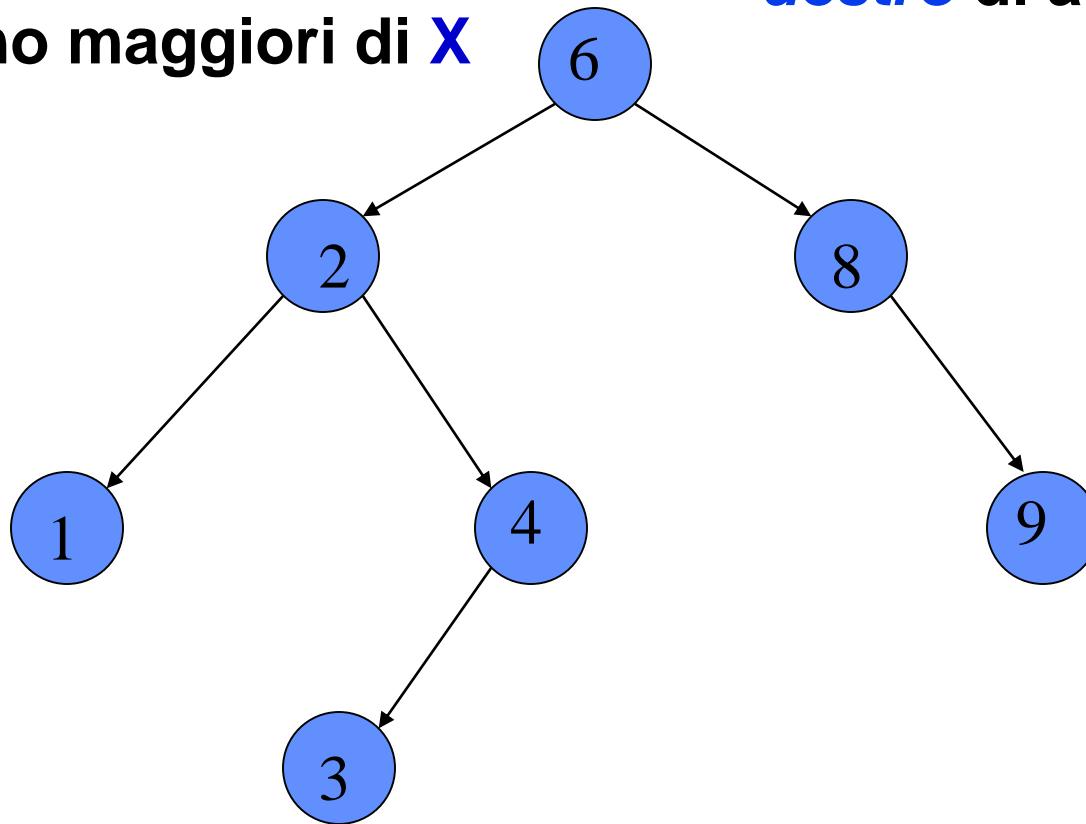
Alberi AVL: alberi binari bilanciati

Proprietà degli ABR

Per ogni nodo **X**, i nodi del sottoalbero sinistro sono minori del nodo **X**, e i nodi del sottoalbero destro sono maggiori di **X**

Properietà AVL

Ad ogni nodo **X**, l'altezza del sottoalbero sinistro differisce da quella del destro di al massimo 1.

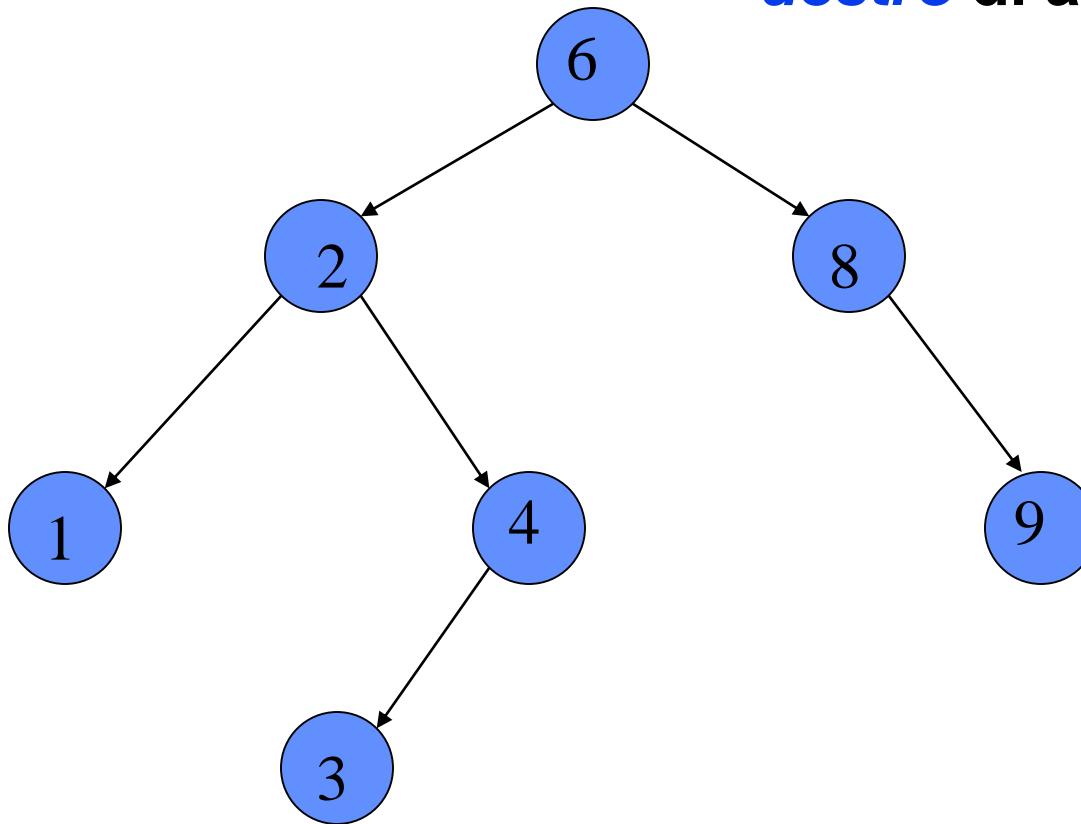


Alberi AVL: alberi binari bilanciati

Altezza(X) =
max(Altezza(sinistro(X)),
Altezza(destro(X))) + 1

Properietà AVL

Ad ogni nodo X , l'altezza del *sottoalbero sinistro* differisce da quella del *destro* di al massimo 1.



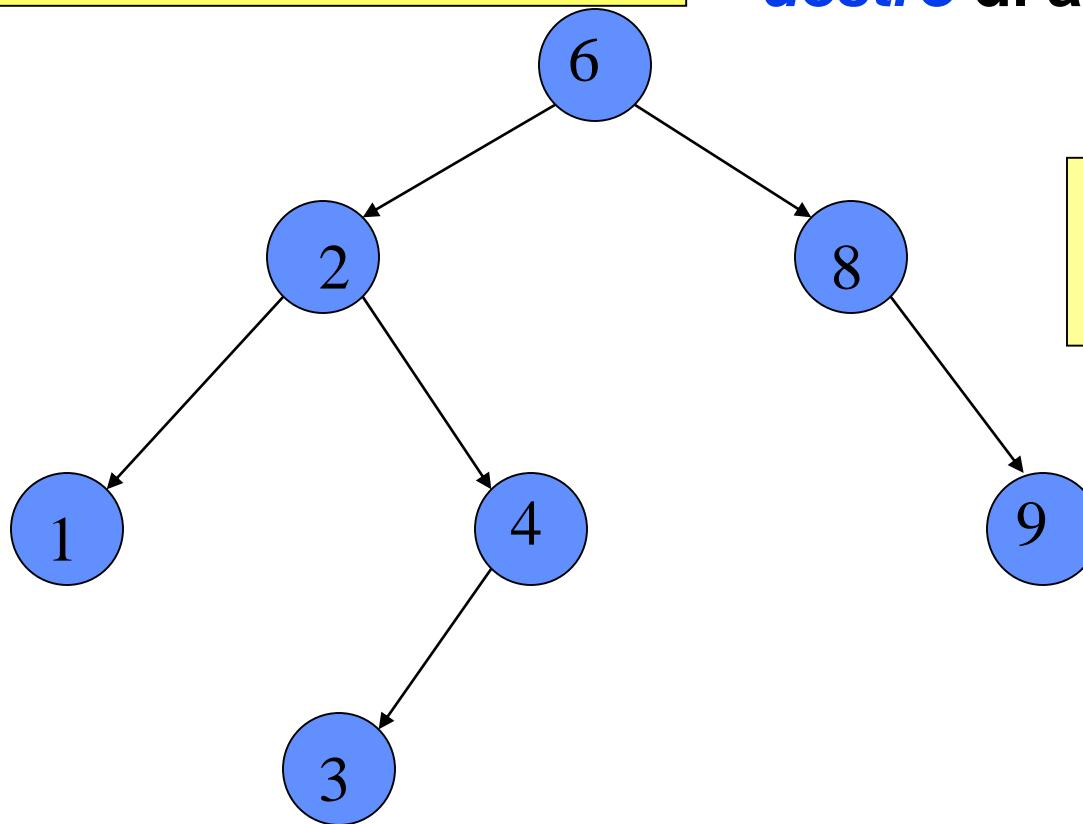
Alberi AVL: alberi binari bilanciati

$\text{Altezza}(X) = \max(\text{Altezza}(\text{sinistro}(X)), \text{Altezza}(\text{destro}(X))) + 1$

$\text{Altezza}(\emptyset) = -1$

Properietà AVL

Ad ogni nodo X , l'altezza del sottoalbero sinistro differisce da quella del destro di al massimo 1.



Questo è un albero AVL?

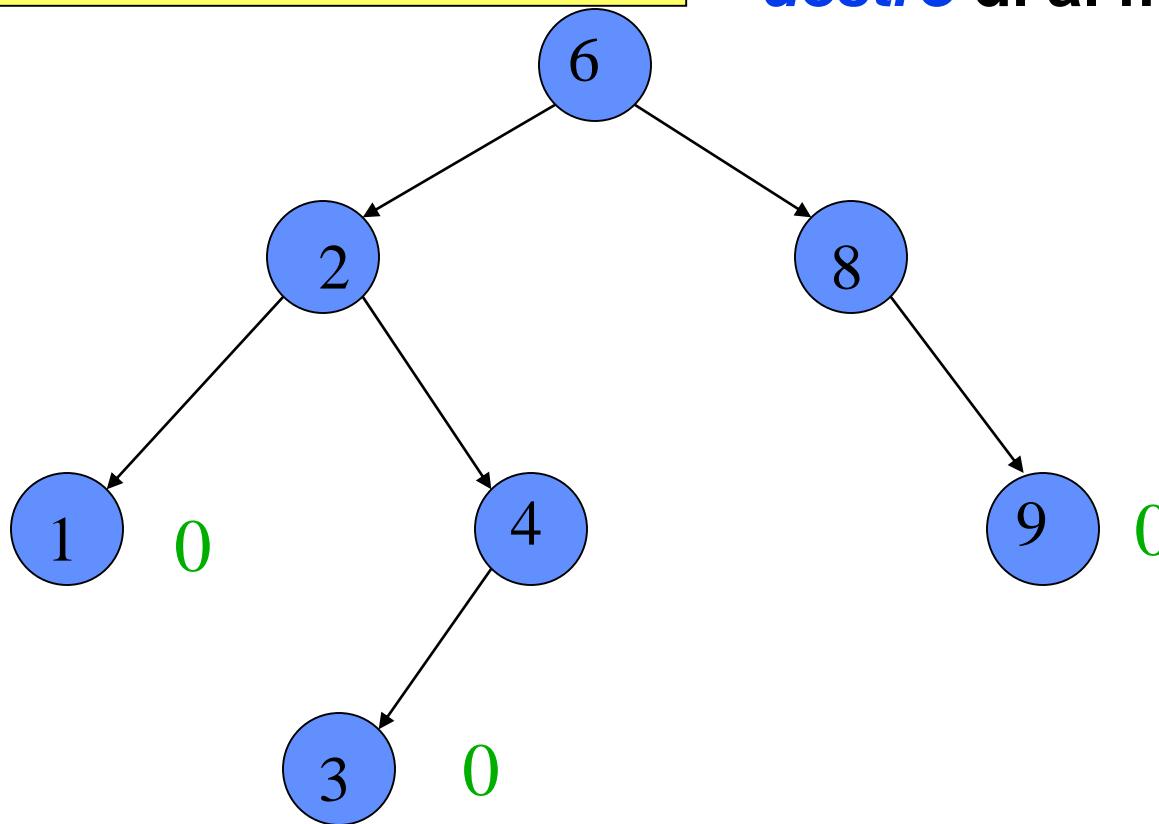
Alberi AVL: alberi binari bilanciati

$\text{Altezza}(X) = \max(\text{Altezza}(\text{sinistro}(X)), \text{Altezza}(\text{destro}(X))) + 1$

$\text{Altezza}(\emptyset) = -1$

Properietà AVL

Ad ogni nodo X , l'altezza del sottoalbero sinistro differisce da quella del destro di al massimo 1.

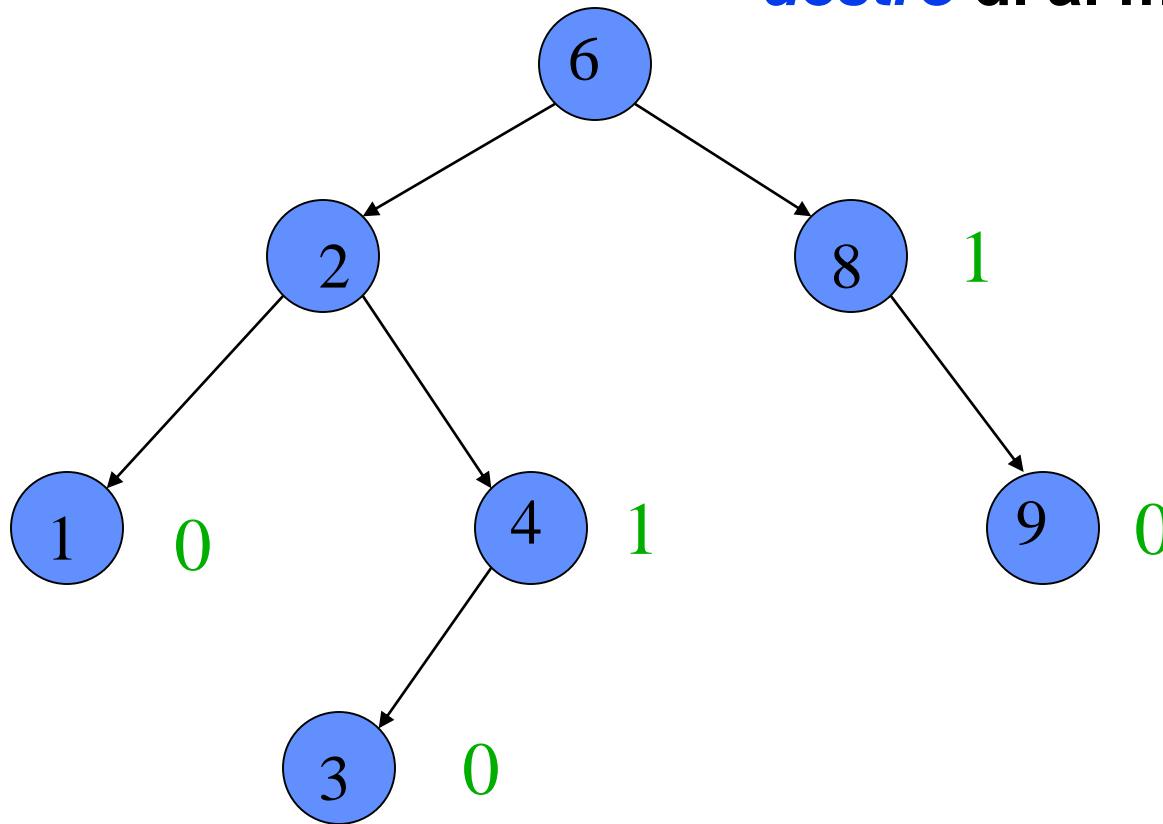


Alberi AVL: alberi binari bilanciati

Altezza(X) =
max(Altezza(sinistro(X)),
Altezza(destro(X))) + 1

Properietà AVL

Ad ogni nodo X , l'altezza del *sottoalbero sinistro* differisce da quella del *destro* di al massimo 1.

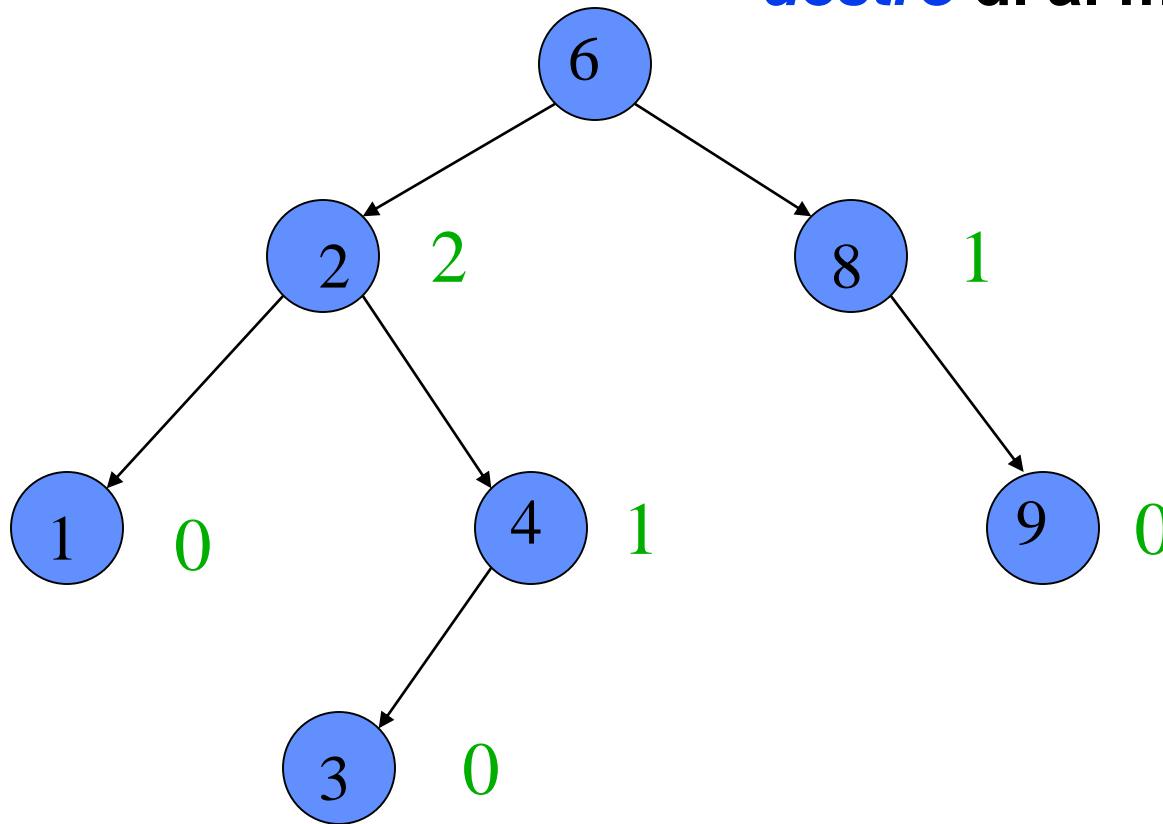


Alberi AVL: alberi binari bilanciati

Altezza(X) =
max(Altezza(sinistro(X)),
Altezza(destro(X))) + 1

Properietà AVL

Ad ogni nodo X , l'altezza del *sottoalbero sinistro* differisce da quella del *destro* di al massimo 1.

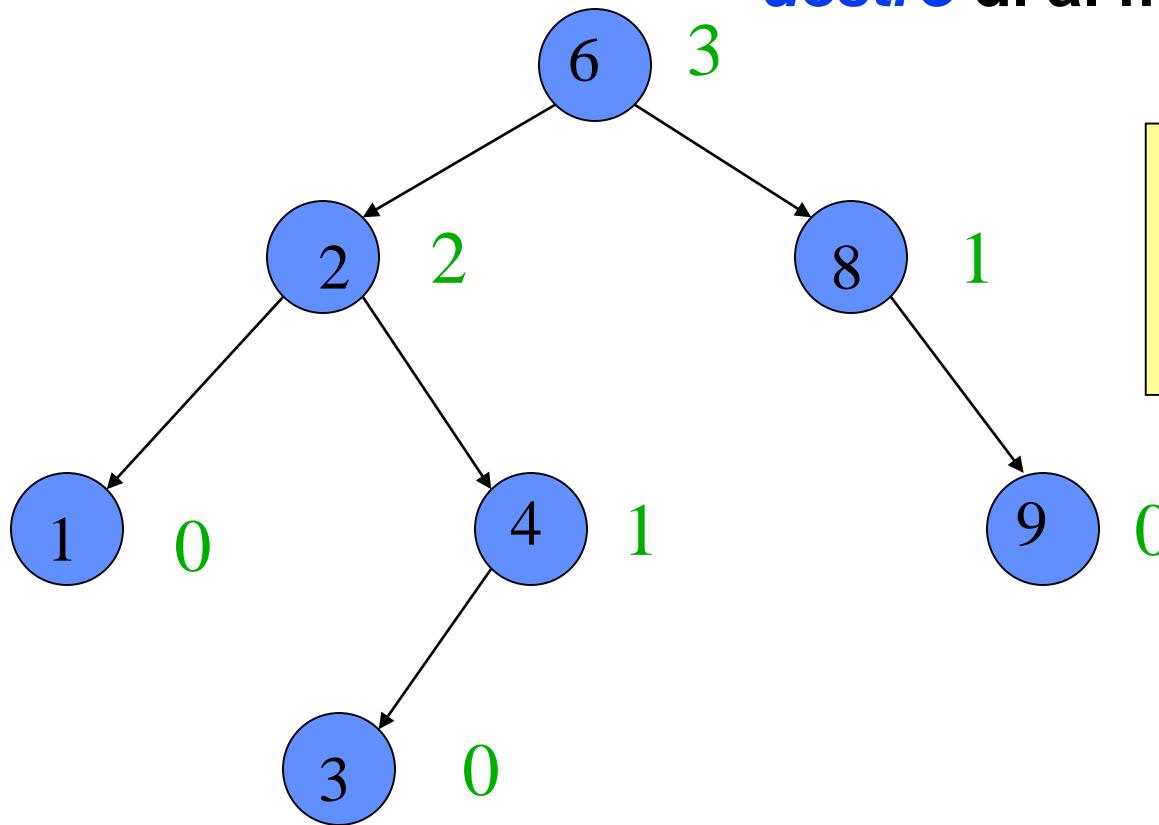


Alberi AVL: alberi binari bilanciati

Altezza(X) =
max(Altezza(sinistro(X)),
Altezza(destro(X))) + 1

Properietà AVL

Ad ogni nodo X , l'altezza del *sottoalbero sinistro* differisce da quella del *destro* di al massimo 1.



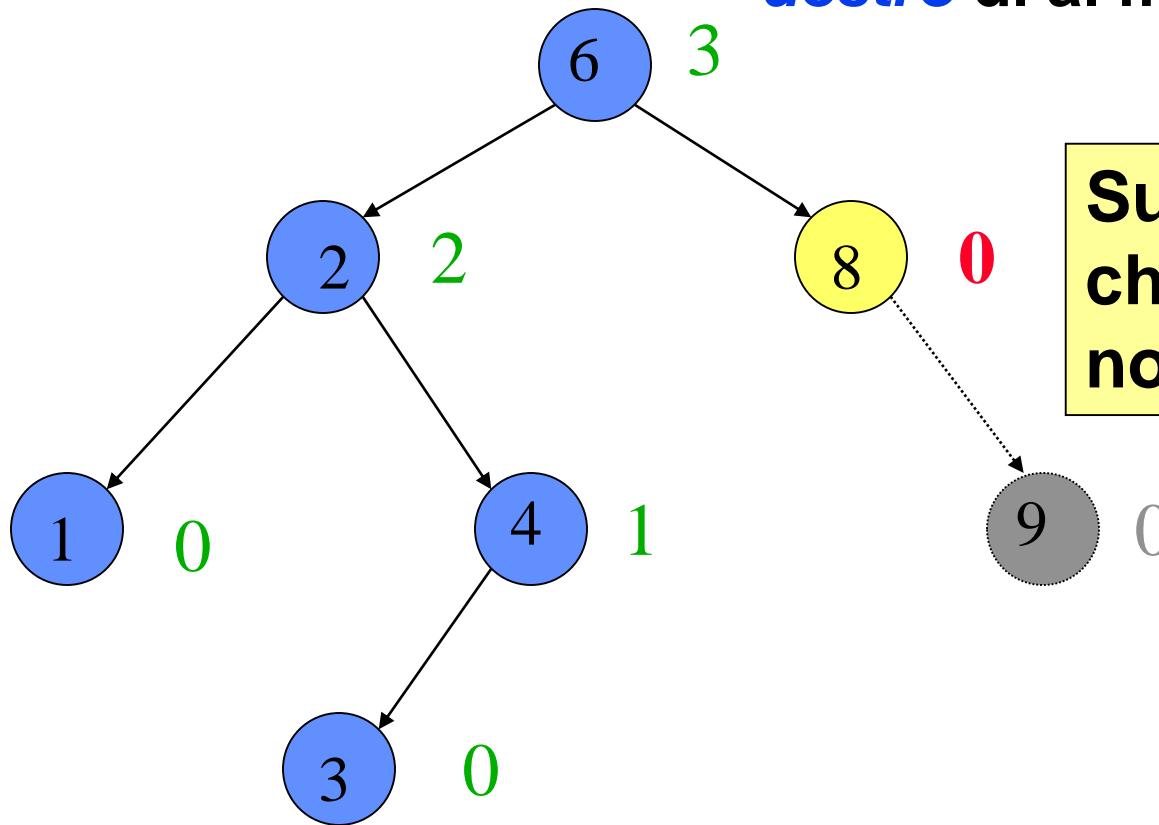
Sì! Questo
è un albero
AVL.

Alberi AVL: alberi binari bilanciati

Altezza(X) =
max(Altezza(sinistro(X)),
Altezza(destro(X))) + 1

Properietà AVL

Ad ogni nodo X , l'altezza del *sottoalbero sinistro* differisce da quella del *destro* di al massimo 1.



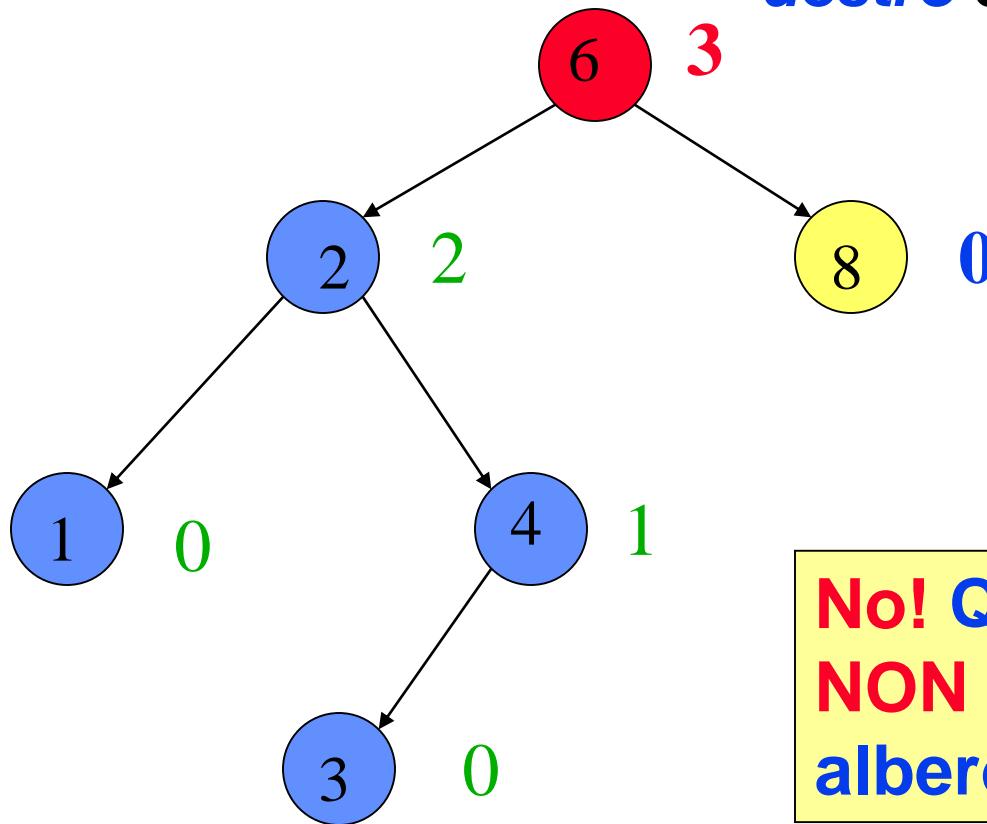
Supponiamo che il nodo 9 non ci sia.

Alberi AVL: alberi binari bilanciati

Altezza(X) =
max(Altezza(sinistro(X)),
Altezza(destro(X))) + 1

Properietà AVL

Ad ogni nodo X , l'altezza del *sottoalbero sinistro* differisce da quella del *destro* di al massimo 1.



La Proprietà AVL non è soddisfatta dal nodo 6.

No! Questo NON è un albero AVL.

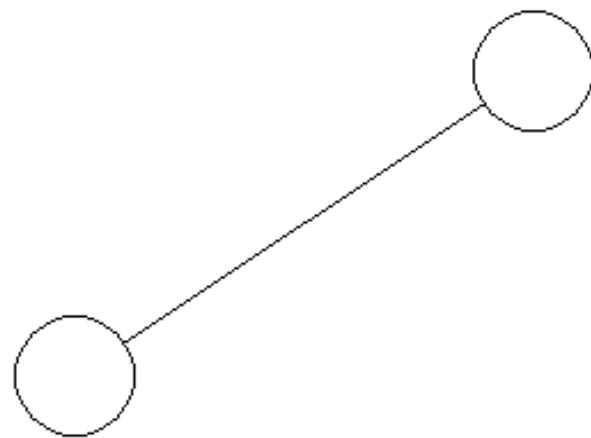
Alberi AVL minimi

Definizione: fissato h , l'albero AVL minimo di altezza h è l'albero AVL di altezza h col minor numero di nodi possibile.

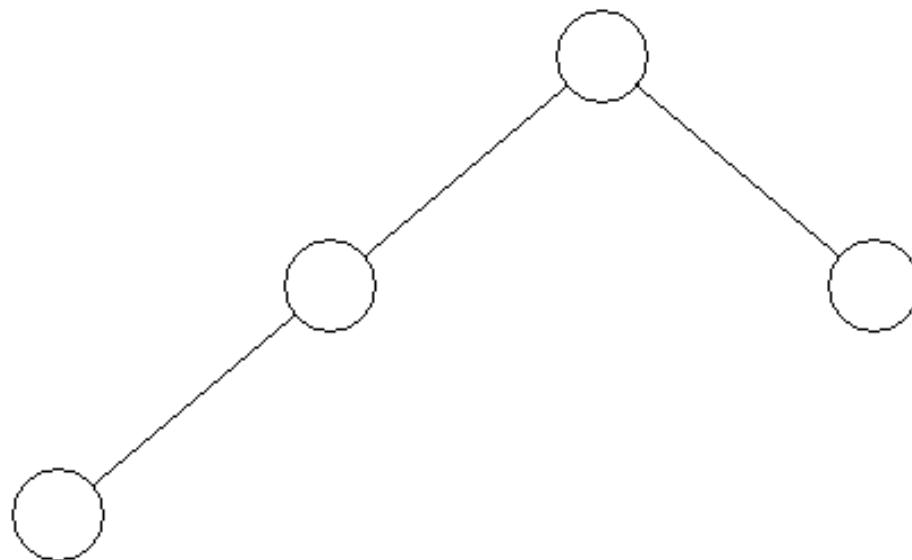
Un **Albero AVL minimo** di altezza h e con numero di **nodi pari** n è l'**Albero AVL di altezza massima** tra tutti gli AVL con n nodi.

*Esempi di alberi AVL minimi di diverse altezze
(cioè con il numero minimo di nodi)*

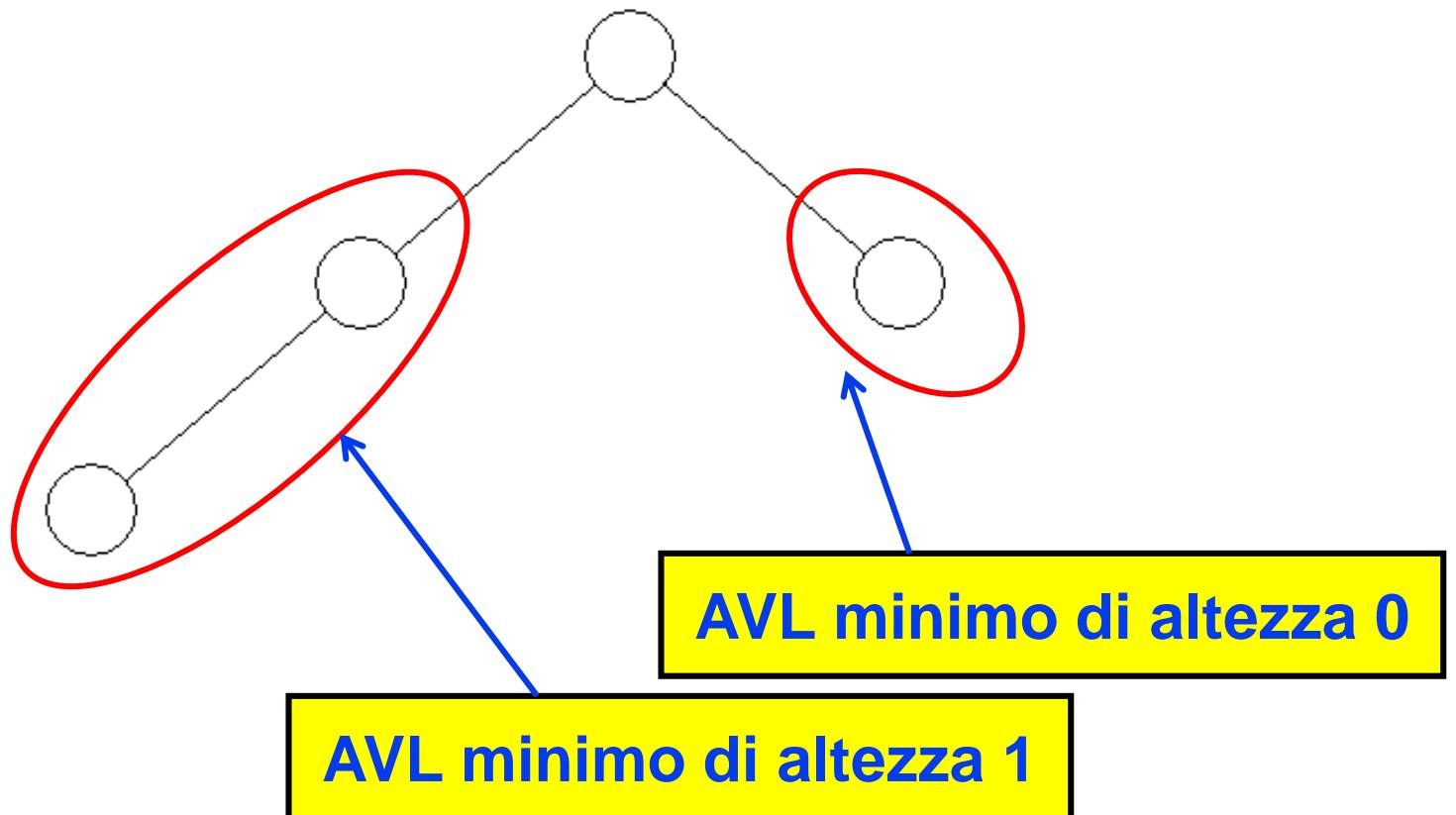
Albero AVL minimo di altezza 1



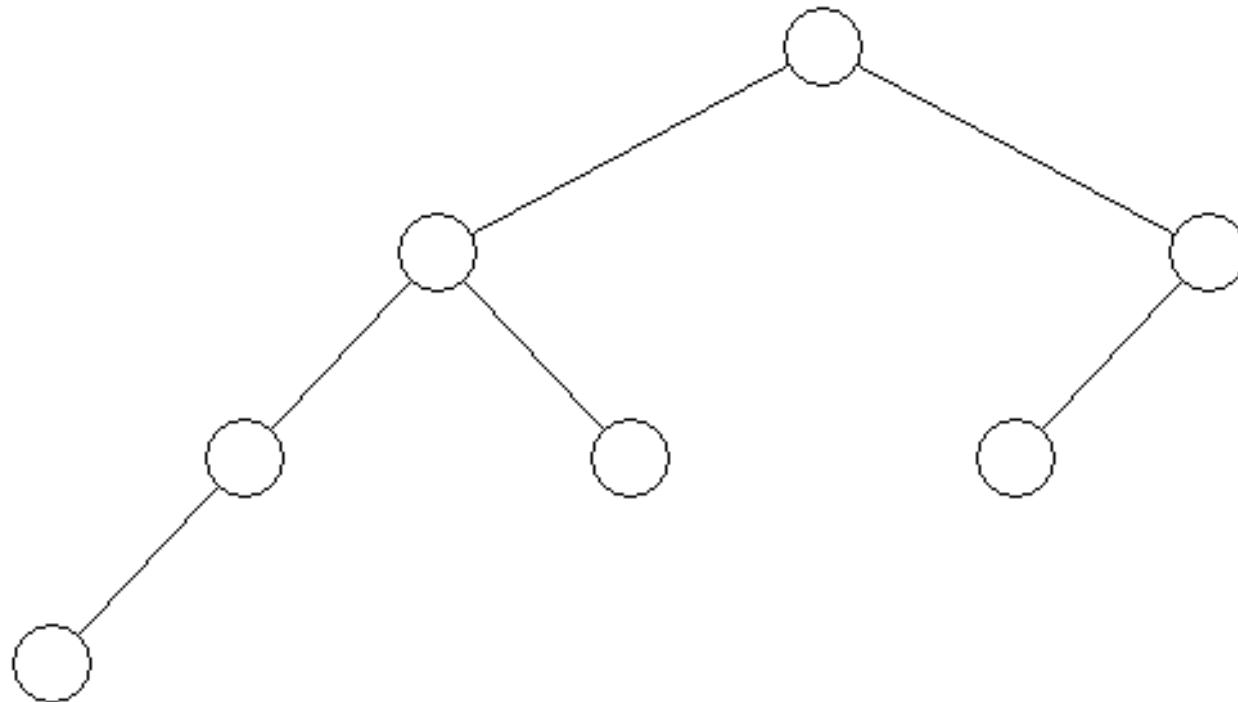
Albero AVL minimo di altezza 2



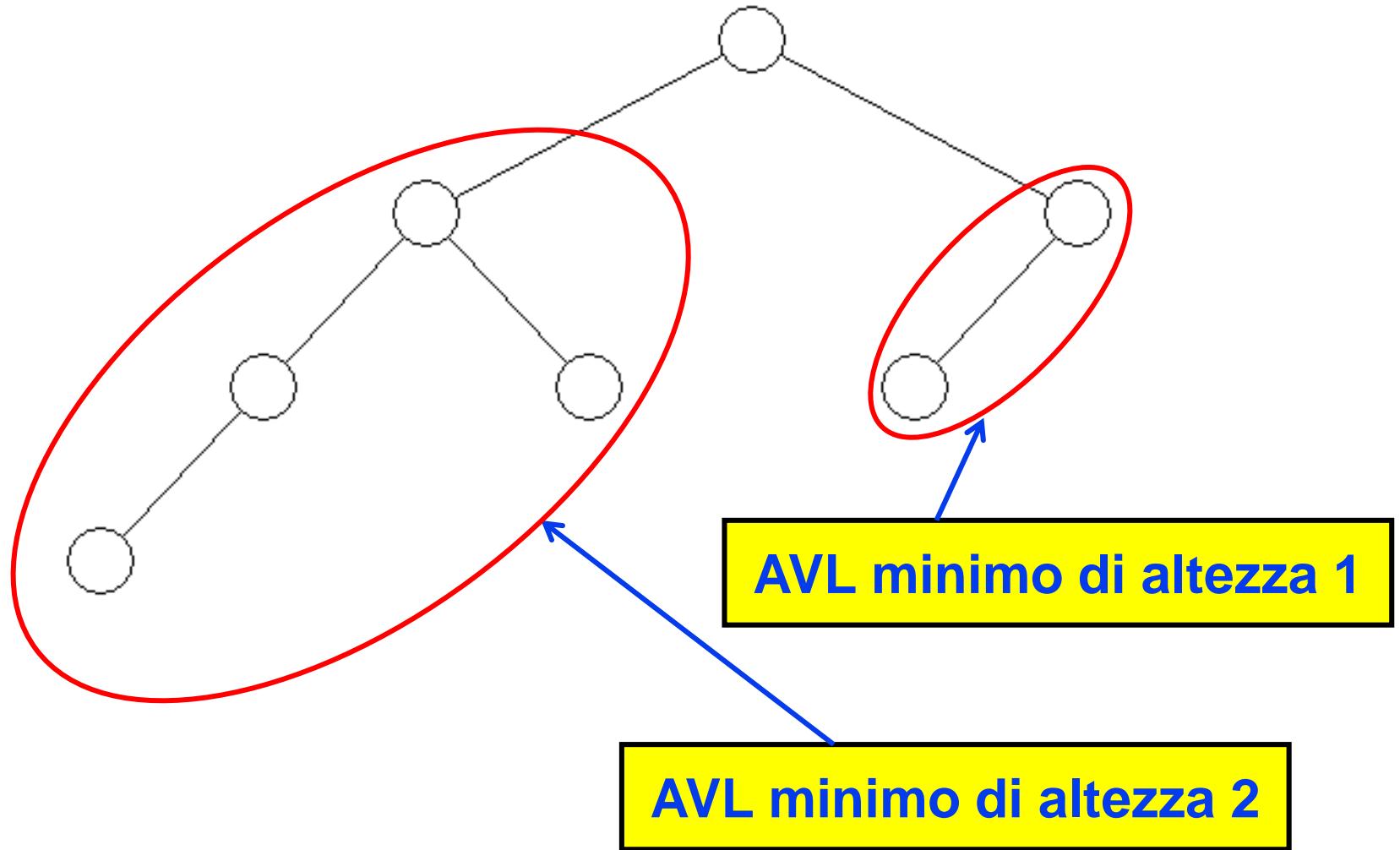
Albero AVL minimo di altezza 2



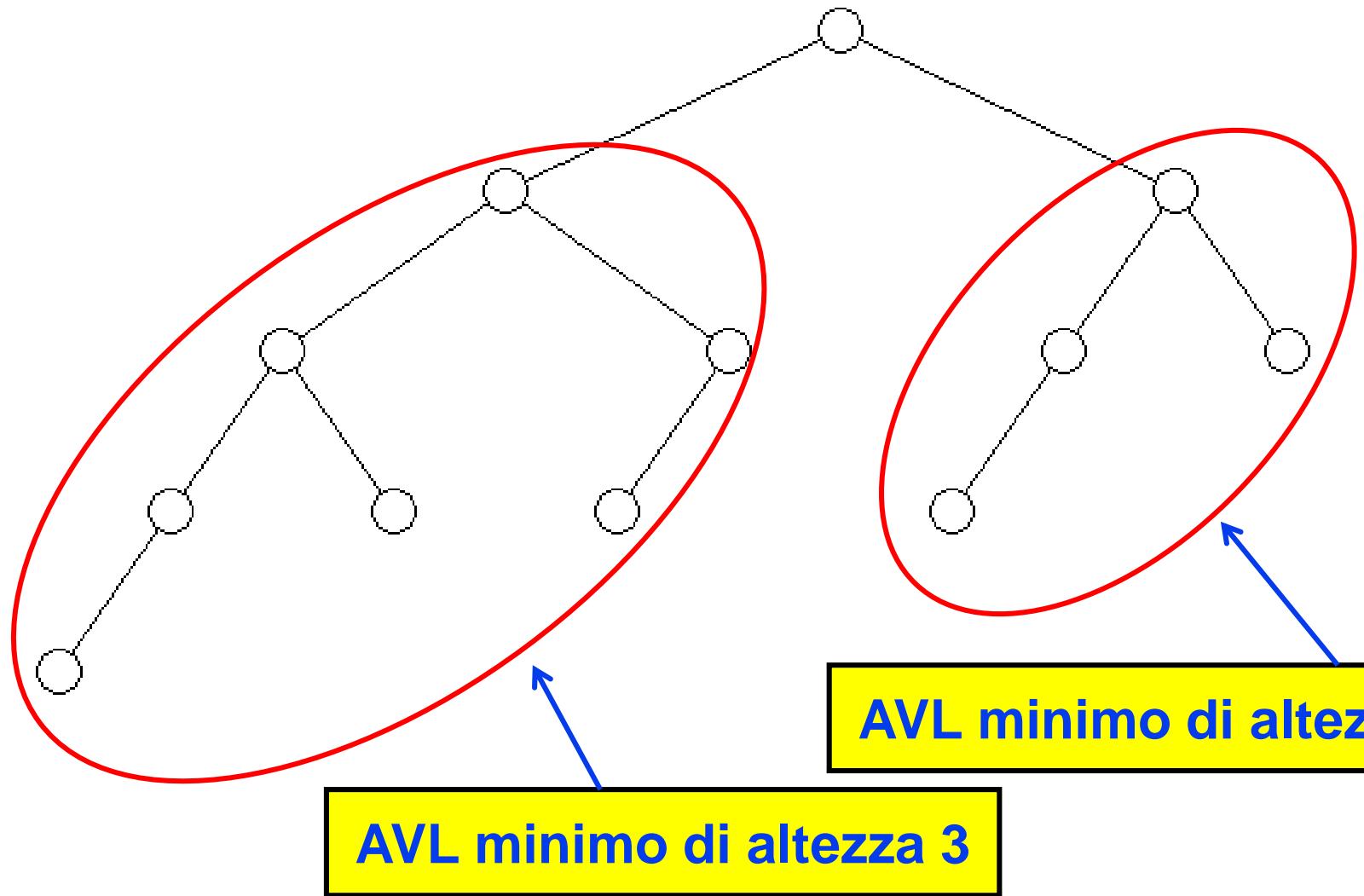
Albero AVL minimo di altezza 3



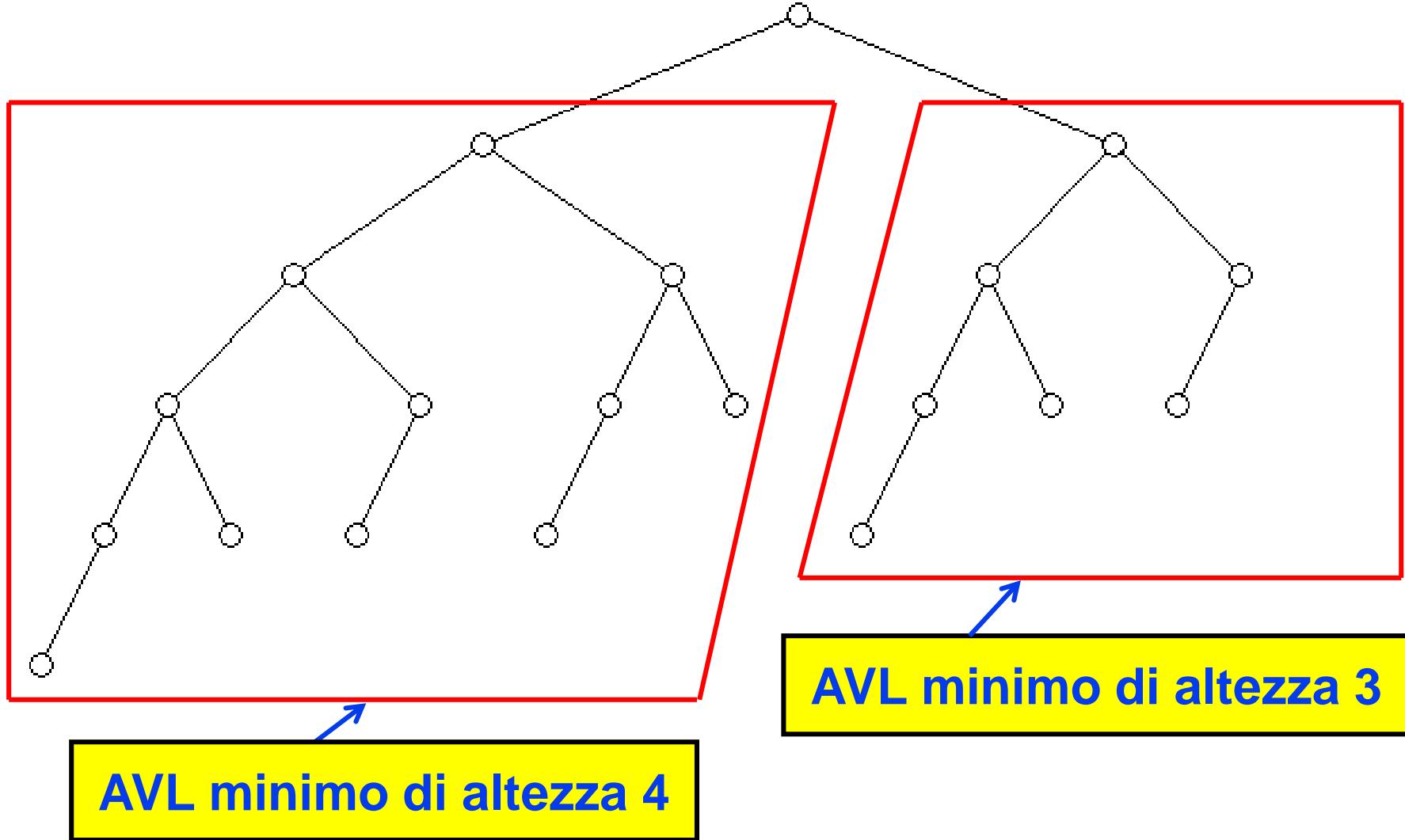
Albero AVL minimo di altezza 3



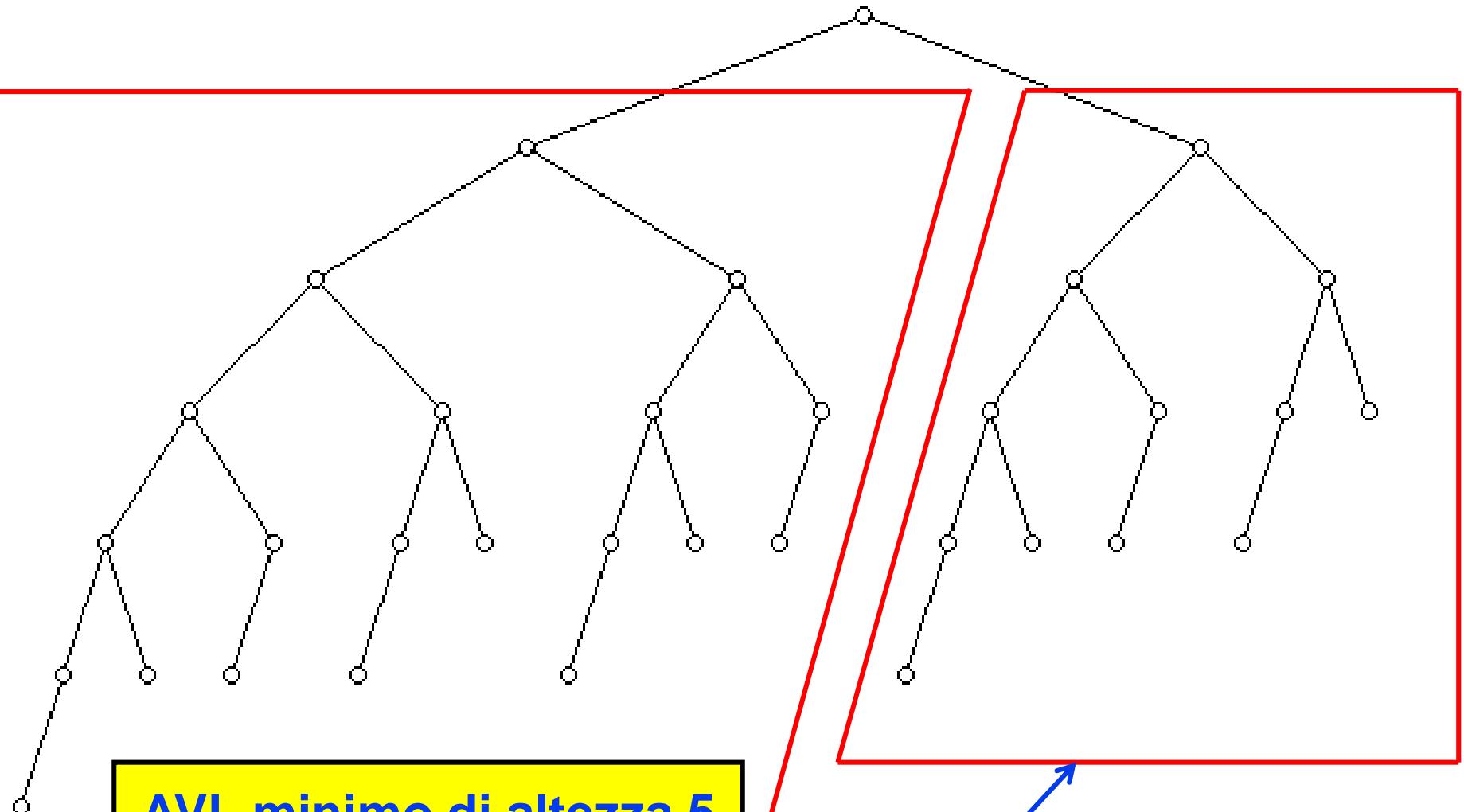
Albero AVL minimo di altezza 4



Albero AVL minimo di altezza 5



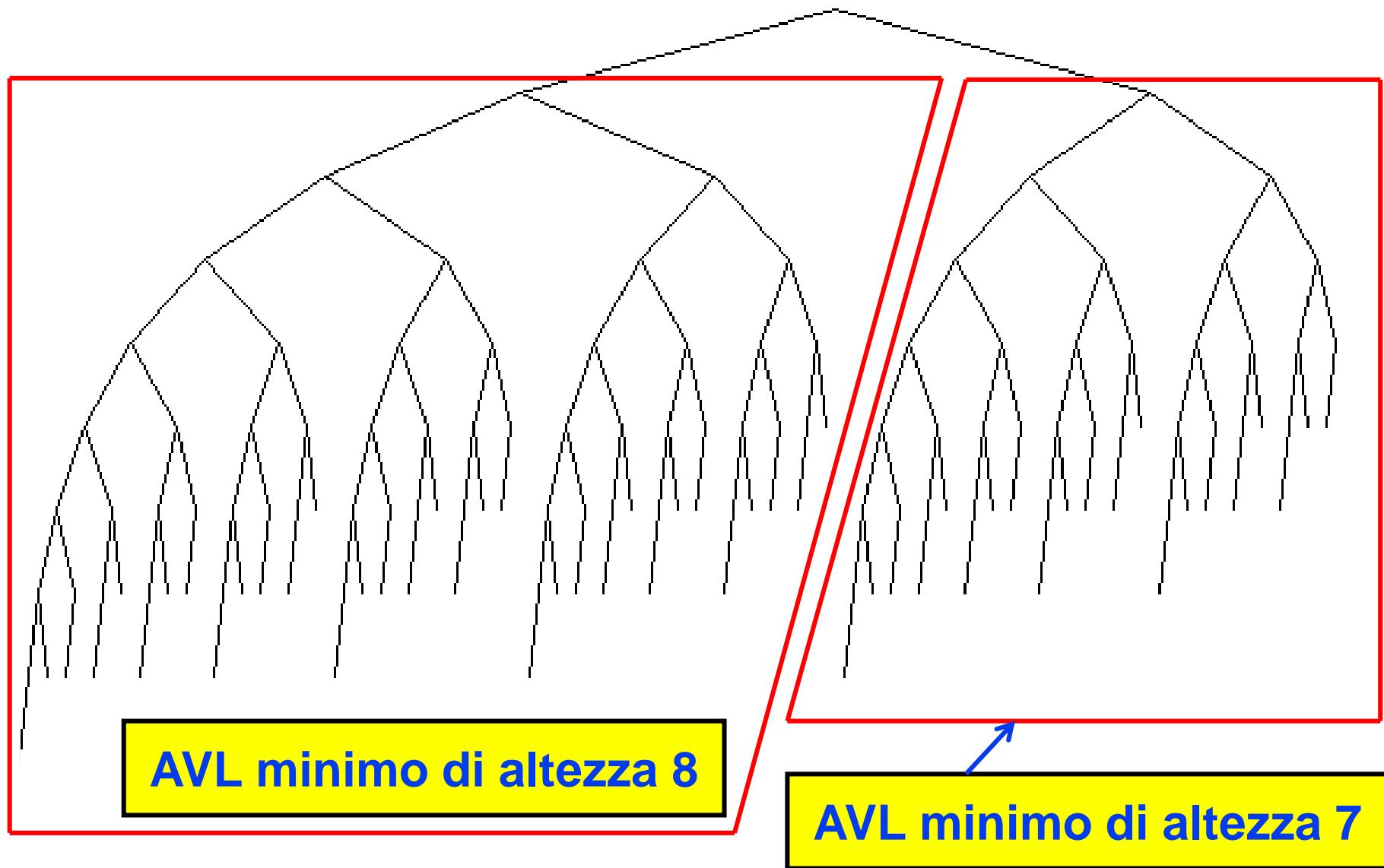
Albero AVL minimo di altezza 6



AVL minimo di altezza 5

AVL minimo di altezza 4

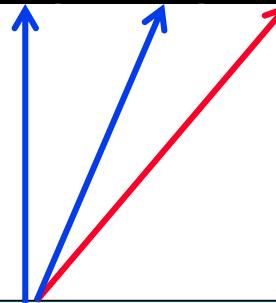
Albero AVL minimo di altezza 9



Numero minimo di nodi di un albero AVL

- Sia N è il *numero minimo di nodi* di un albero AVL di altezza H

Altezza H	0	1	2	3	4	5	6	7	8	9	10
Nodi $N(H)$	1	2	4	7	12	20	33	54	88	143	232



$$N(H) = N(H-1) + N(H-2) + 1$$

Numero minimo di nodi di un albero AVL

Altezza H	0	1	2	3	4	5	6	7	8	9	10
Nodi $N(H)$	1	2	4	7	12	20	33	54	88	143	232

- Sia N è il *numero minimo di nodi* di un albero AVL di altezza H

H	0	1	2	3	4	5	6	7	8	9	10
$Fib(H)$	0	1	1	2	3	5	8	13	21	34	55

- Si dimostra che

$$N(H) = Fib(H+3) - 1$$

dove $Fib(k)$ è il k -esimo *numero di Fibonacci*

Numero minimo di nodi di un albero AVL

Altezza H	0	1	2	3	4	5	6	7	8	9	10
Nodi $N(H)$	1	2	4	7	12	20	33	54	88	143	232

- Sia N è il *numero minimo di nodi* di un albero AVL di altezza H
- Si dimostra (ad esempio *per induzione*) che

$$N(H) = Fib(H+3) - 1$$

dove $Fib(k)$ è il k -esimo *numero di Fibonacci*

- Per grandi valori di k ,

$$Fib(k) \cong \frac{1}{\sqrt{5}} \left[\frac{1 + \sqrt{5}}{2} \right]^k$$

Numero minimo di nodi di un albero AVL

Altezza H	0	1	2	3	4	5	6	7	8	9	10
Nodi $N(H)$	1	2	4	7	12	20	33	54	88	143	232

- Si dimostra che $N(H) = F(H+3) - 1$
- Il *numero minimo N* di *nodi* di un albero AVL di altezza H è allora dato da

$$N(H) \cong \frac{1}{\sqrt{5}} \left[\frac{1+\sqrt{5}}{2} \right]^{H+3} - 1$$

$$Fib(k) \cong \frac{1}{\sqrt{5}} \left[\frac{1+\sqrt{5}}{2} \right]^k$$

Numero minimo di nodi di un albero AVL

Altezza H	0	1	2	3	4	5	6	7	8	9	10
Nodi $N(H)$	1	2	4	7	12	20	33	54	88	143	232

- Risolvendo per H si ottiene:

$$H \approx 1.44 \log(N + 2) - .328$$

- Il *numero minimo N* di *nodi* di un albero AVL di altezza H è allora dato da

$$N(H) \approx \frac{1}{\sqrt{5}} \left[\frac{1 + \sqrt{5}}{2} \right]^{H+3} - 1$$

Numero minimo di nodi di un albero AVL

Altezza H	0	1	2	3	4	5	6	7	8	9	10
Nodi $N(H)$	1	2	4	7	12	20	33	54	88	143	232

- Risolvendo per H si ottiene:

$$H \approx 1.44 \log(N + 2) - .328$$

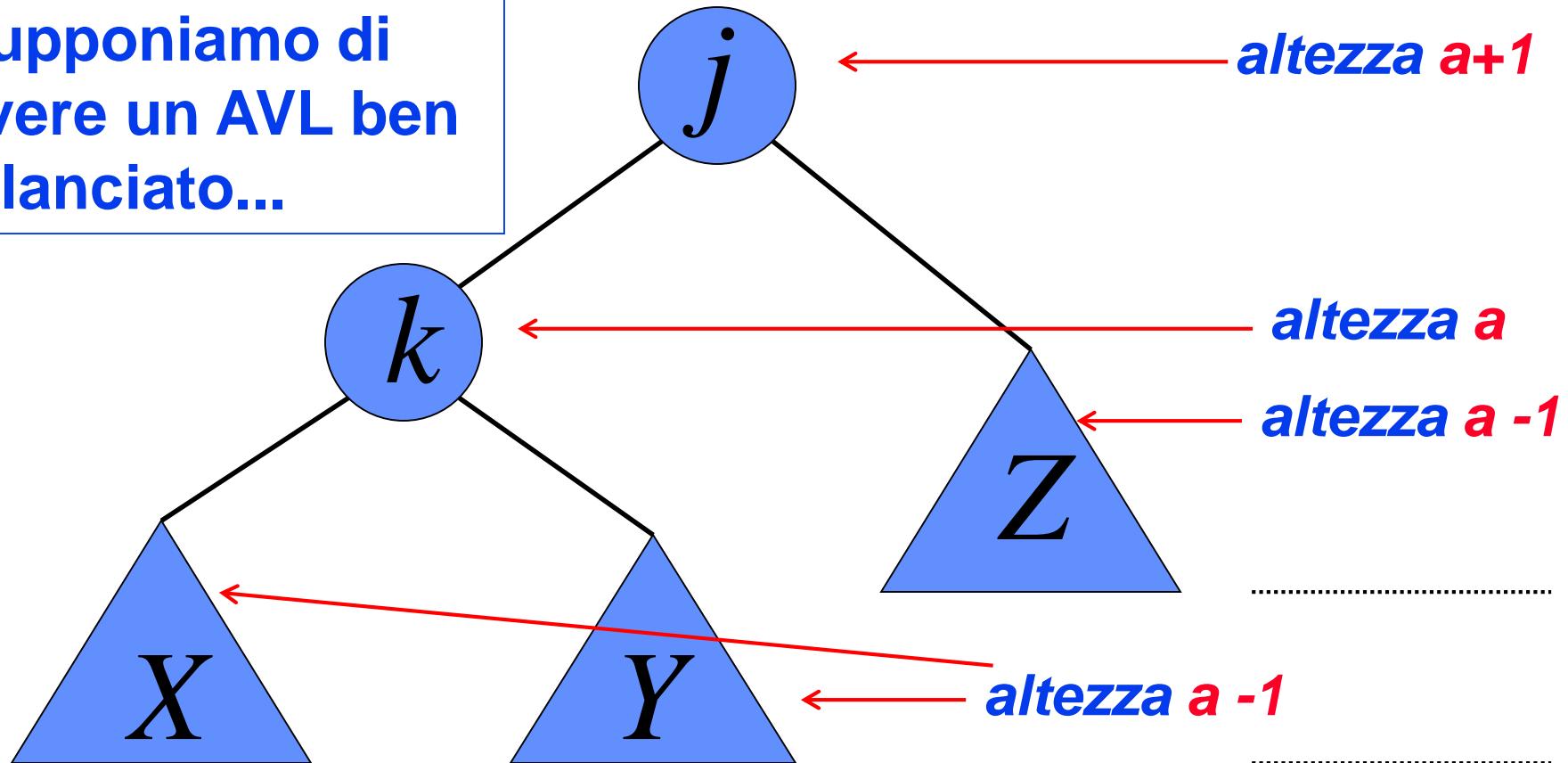
- Ciò significa che l'*altezza* di un *albero AVL* con N nodi **NON** è mai più del **44% maggiore** dell'*altezza ottima* di un *albero perfettamente bilanciato* con N nodi.

Costruzione di un albero AVL

- Durante la **costruzione** di un **albero AVL**, l'unica volta che è possibile **violare la condizione di bilanciamento** è quando un **nuovo nodo** viene **inserito**.
- Ci concentreremo sull'avo più vicino al nuovo nodo inserito, che è divenuto non bilanciato
- **Ci sono essenzialmente due casi, chiamati Rotazione singola e Ritazione doppia**

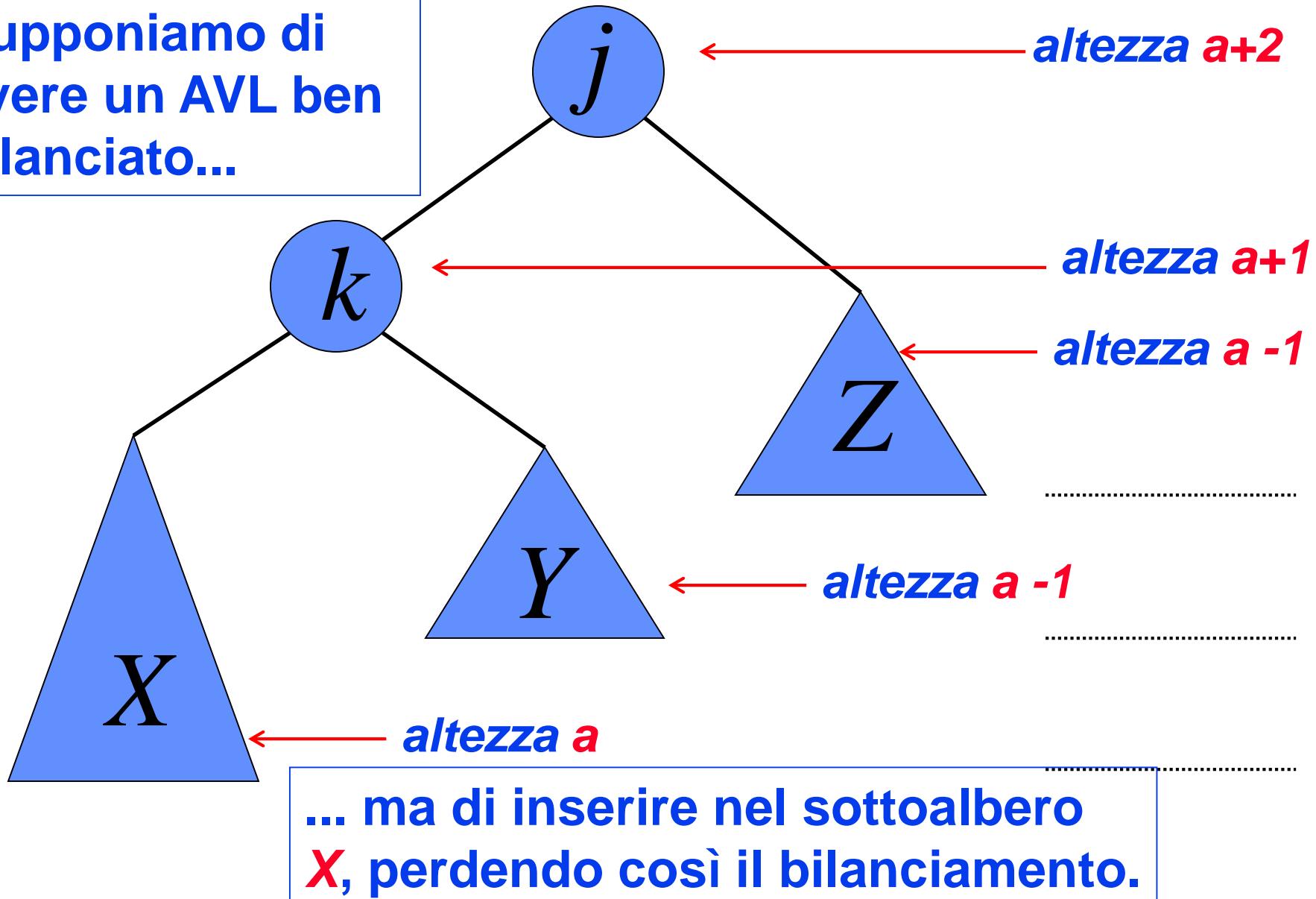
Rotazione in alberi AVL: caso I

Supponiamo di avere un AVL ben bilanciato...

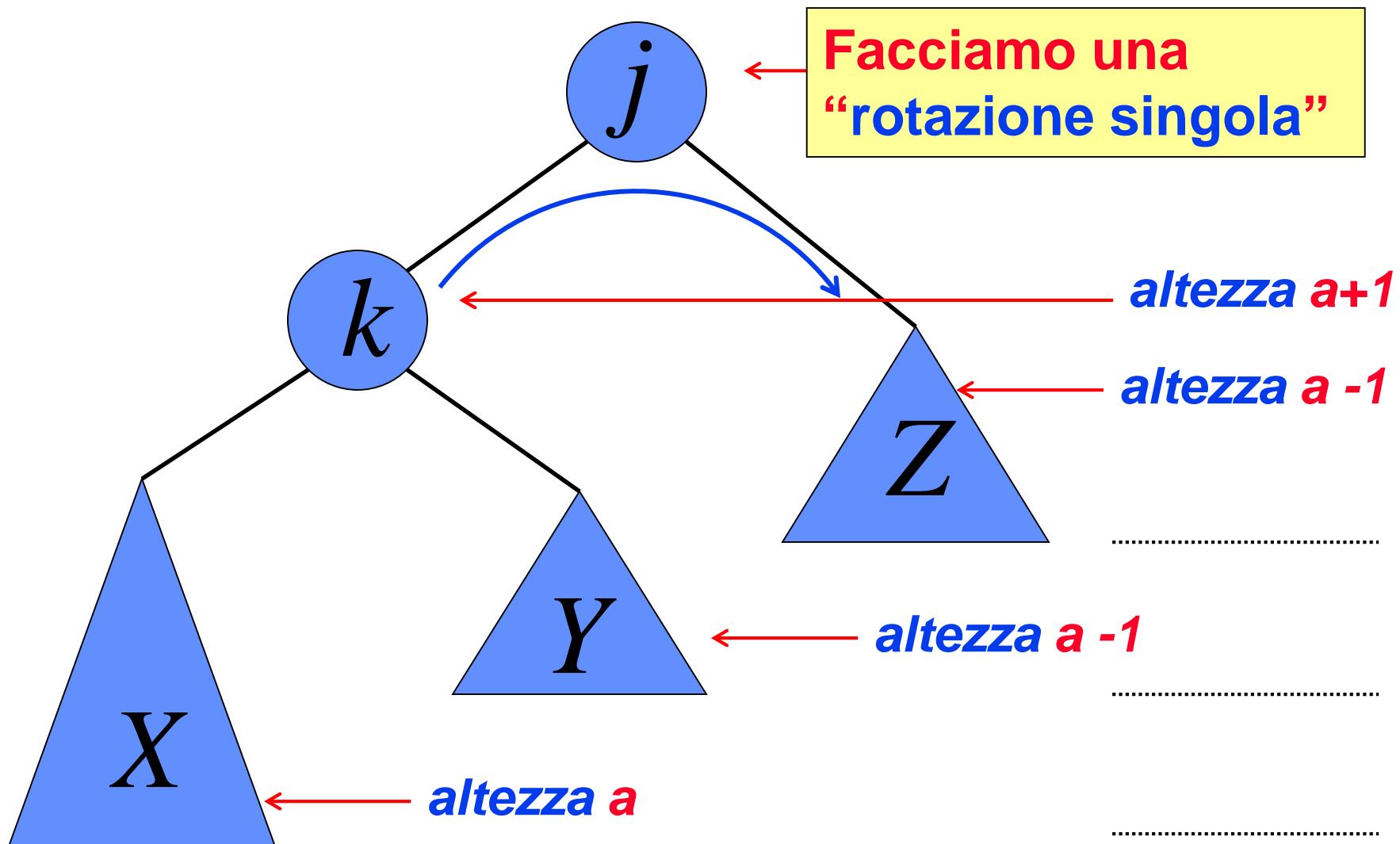


Rotazione in alberi AVL: caso I

Supponiamo di avere un AVL ben bilanciato...



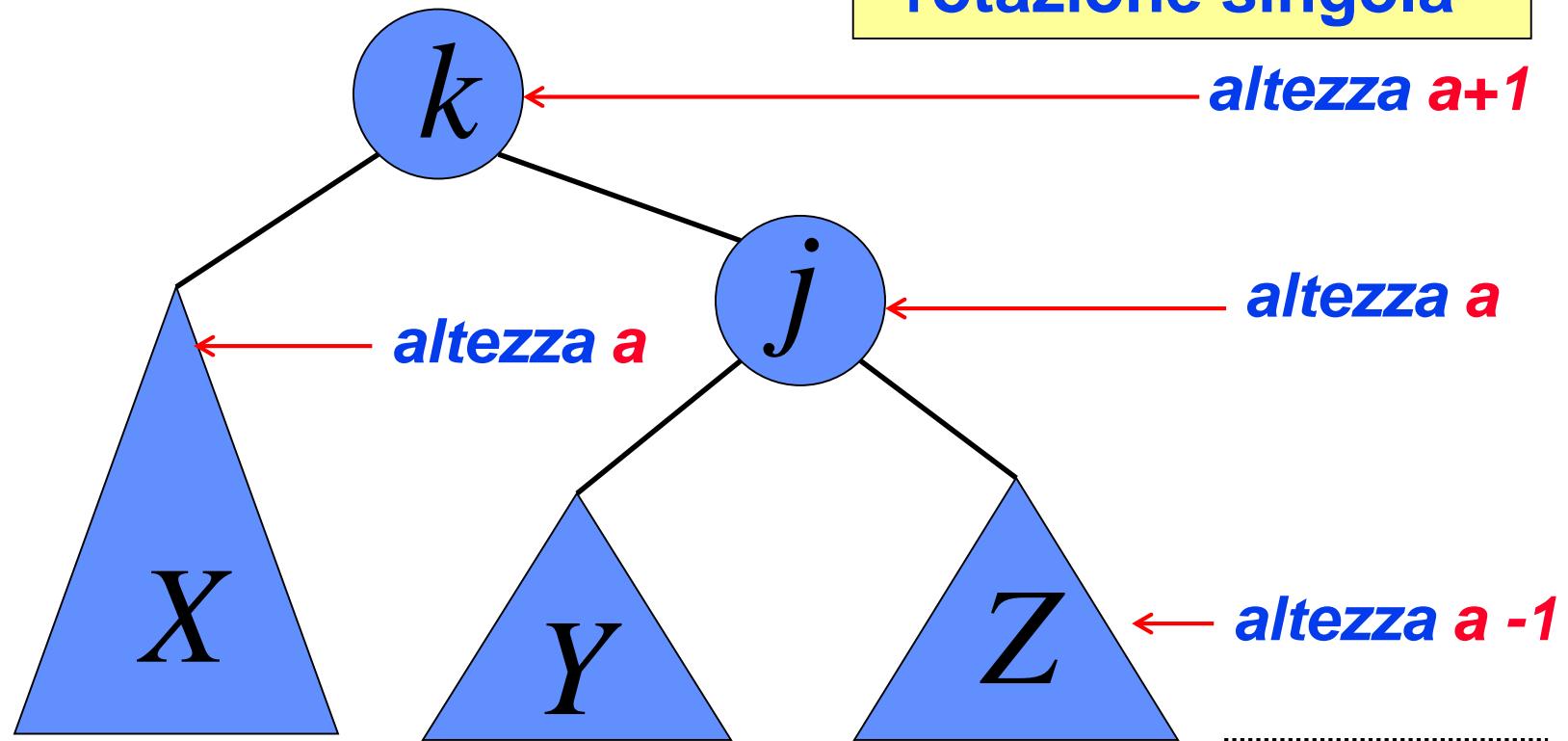
Rotazione singola in alberi AVL



Rotazione singola in alberi AVL

In un singolo passo

Facciamo una
“rotazione singola”

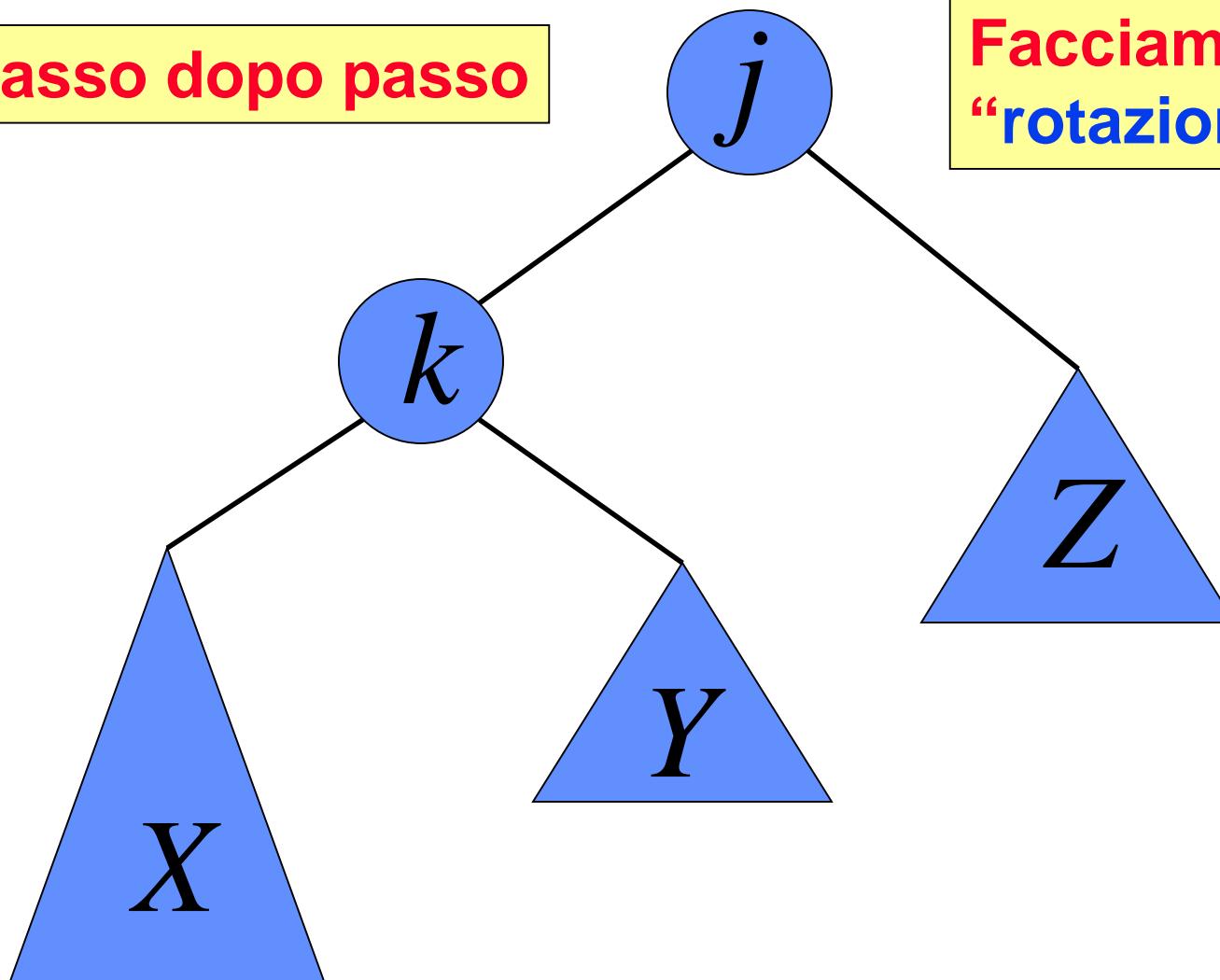


Rotazione singola sinistra

Rotazione singola in alberi AVL

Passo dopo passo

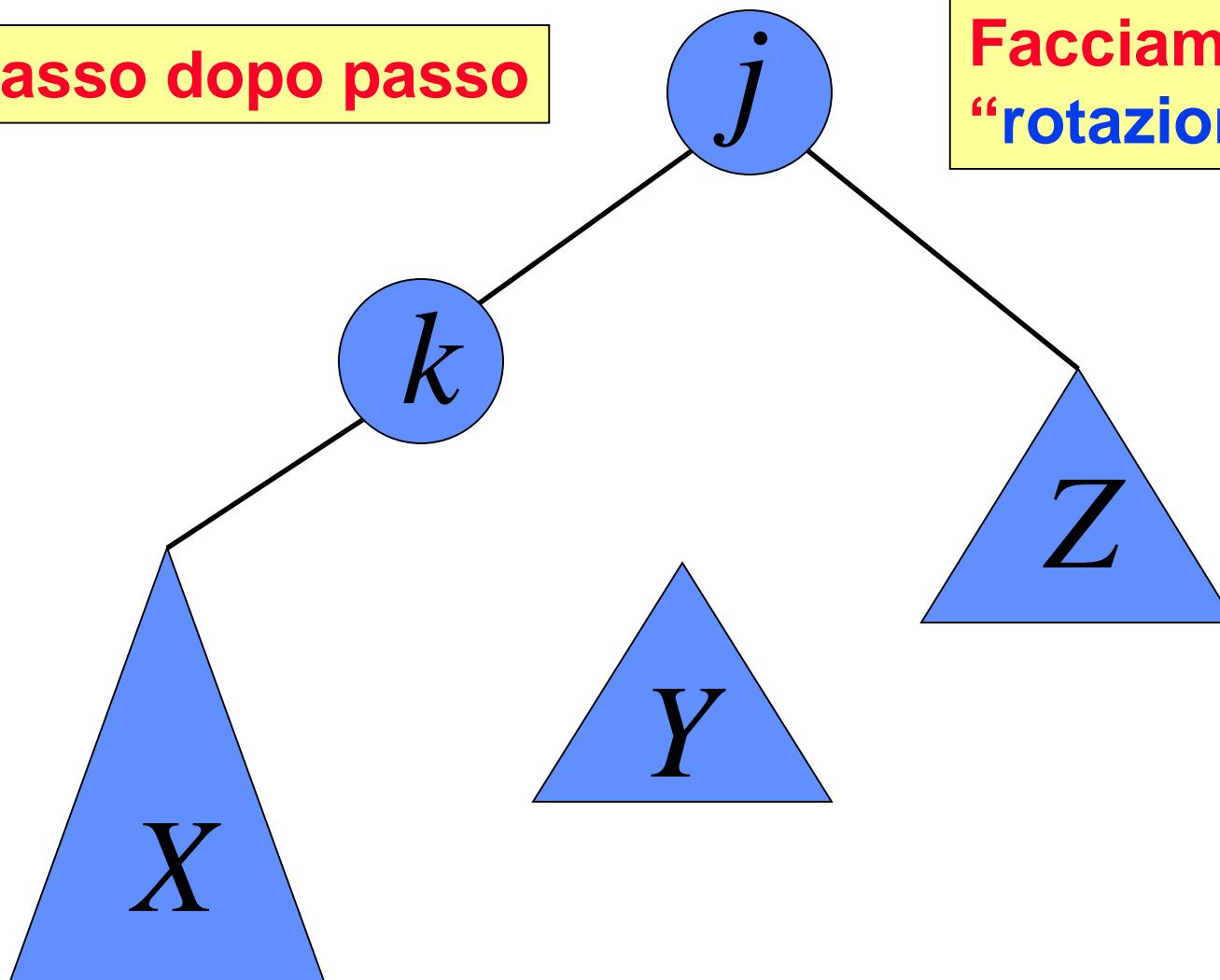
Facciamo una
“rotazione singola”



Rotazione singola in alberi AVL

Passo dopo passo

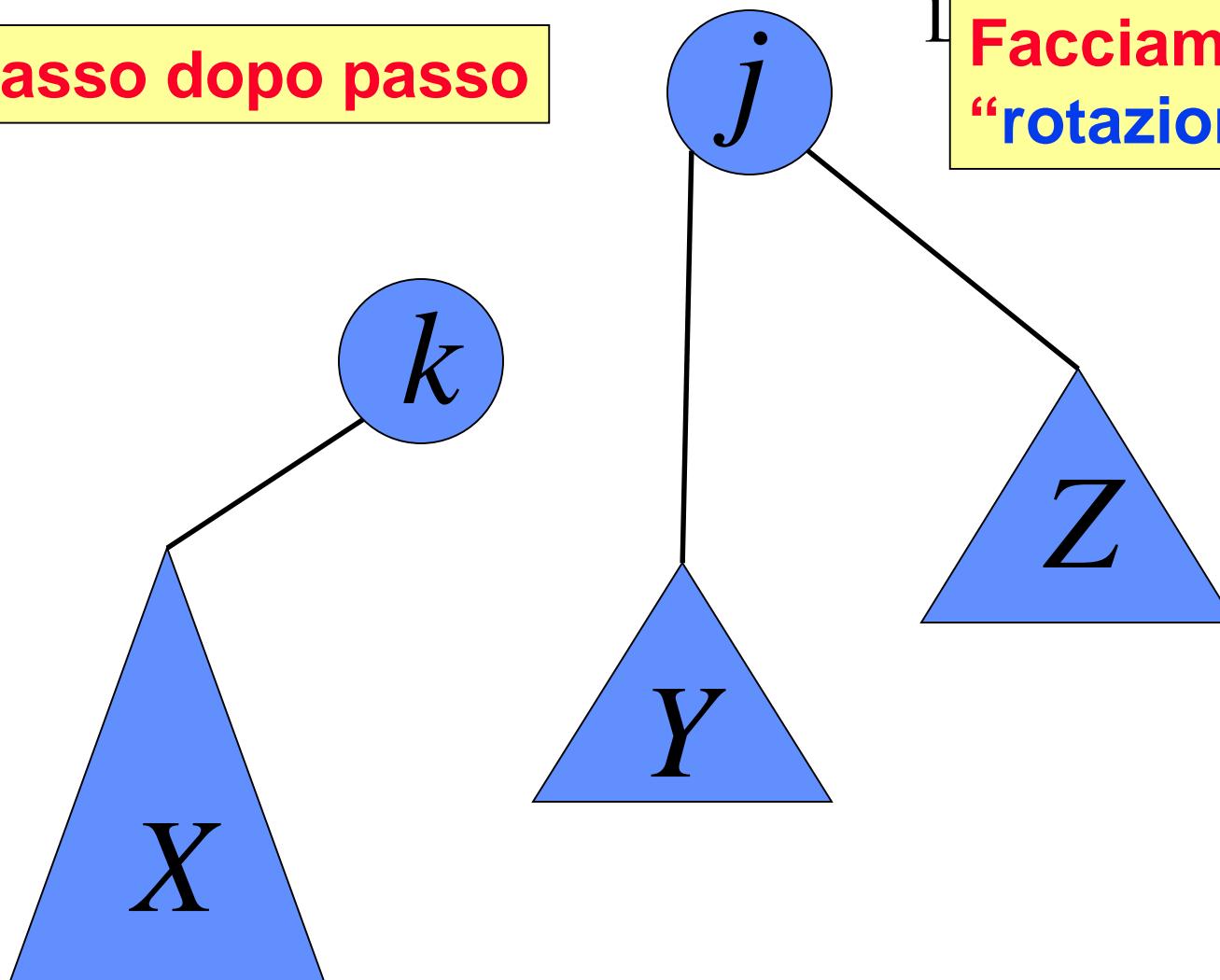
Facciamo una
“rotazione singola”



Rotazione singola in alberi AVL

Passo dopo passo

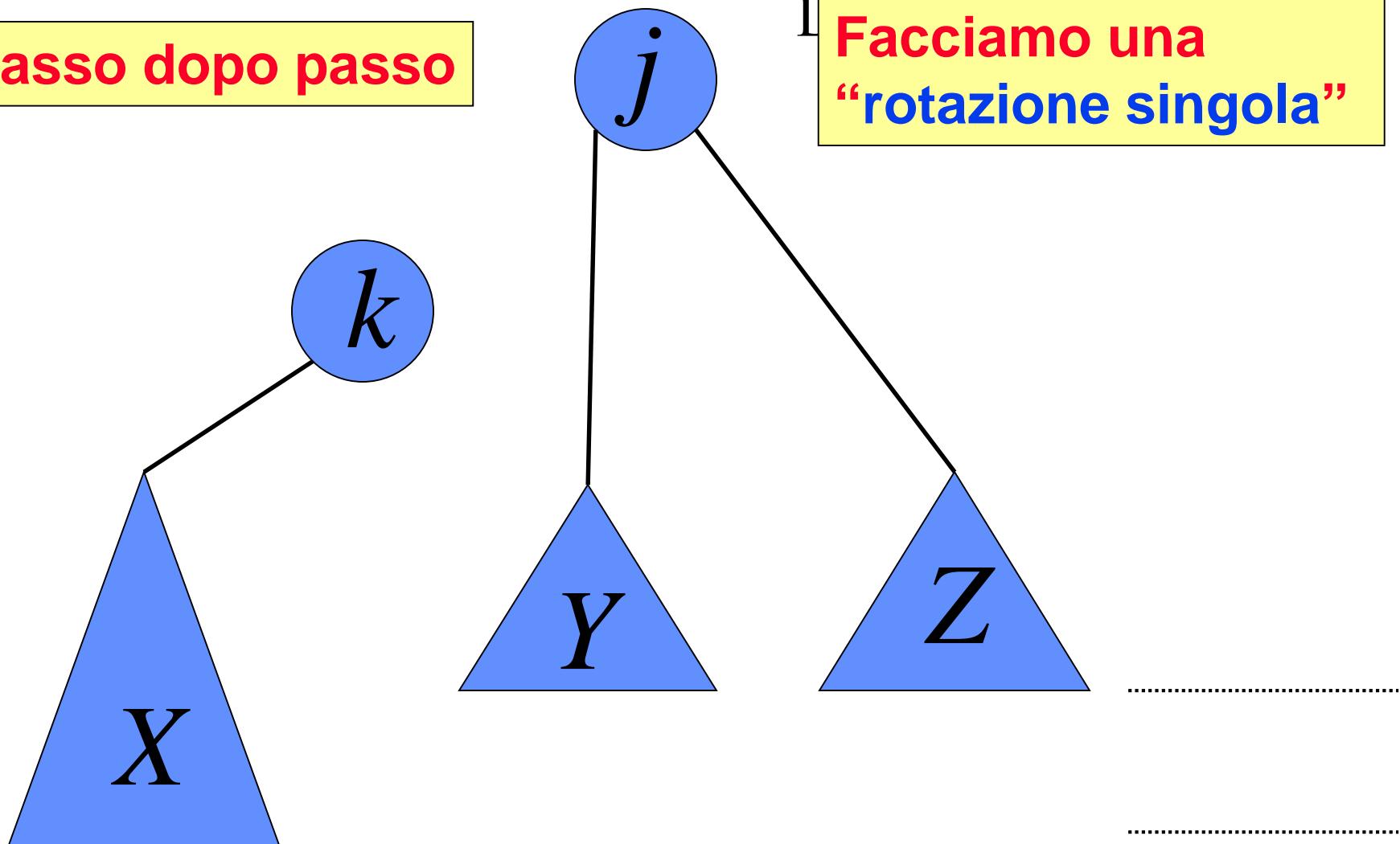
Facciamo una
“rotazione singola”



Rotazione singola in alberi AVL

Passo dopo passo

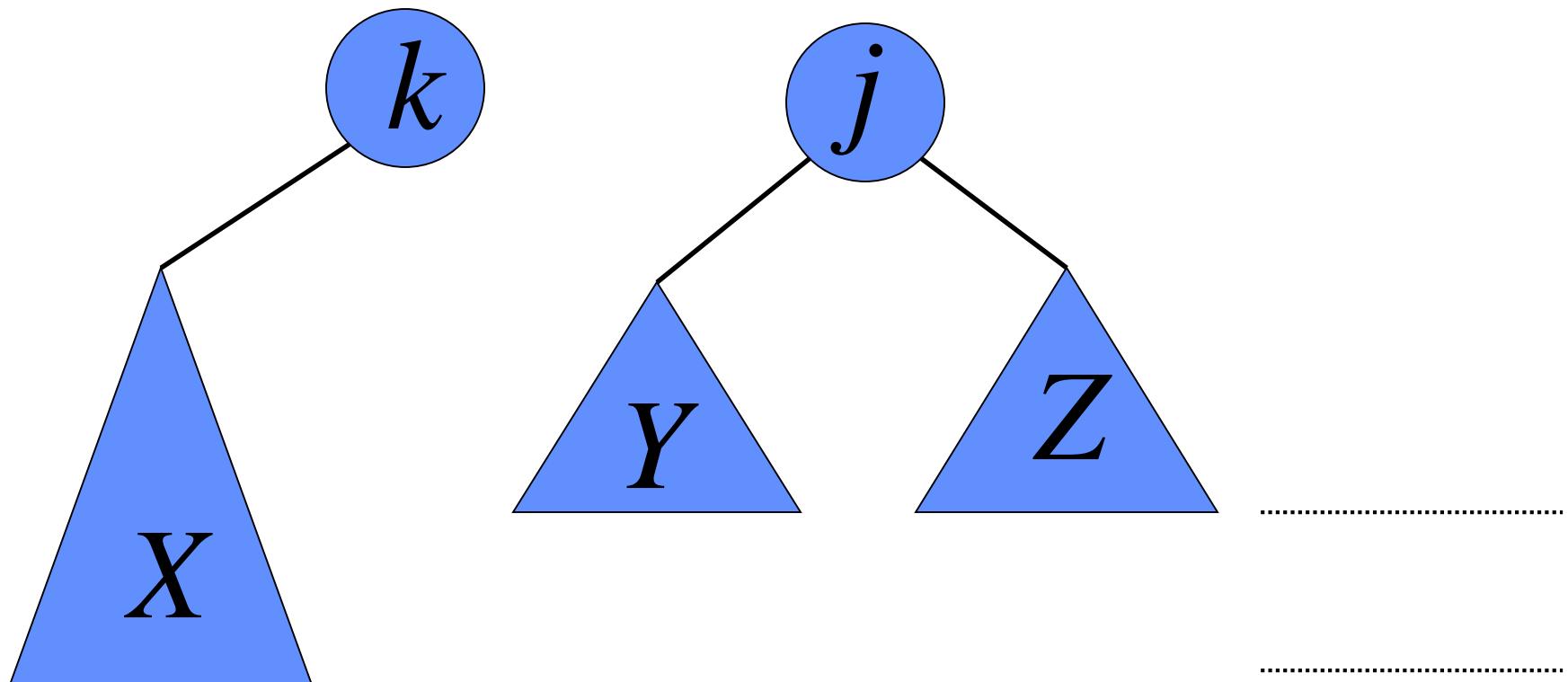
Facciamo una
“rotazione singola”



Rotazione singola in alberi AVL

Passo dopo passo

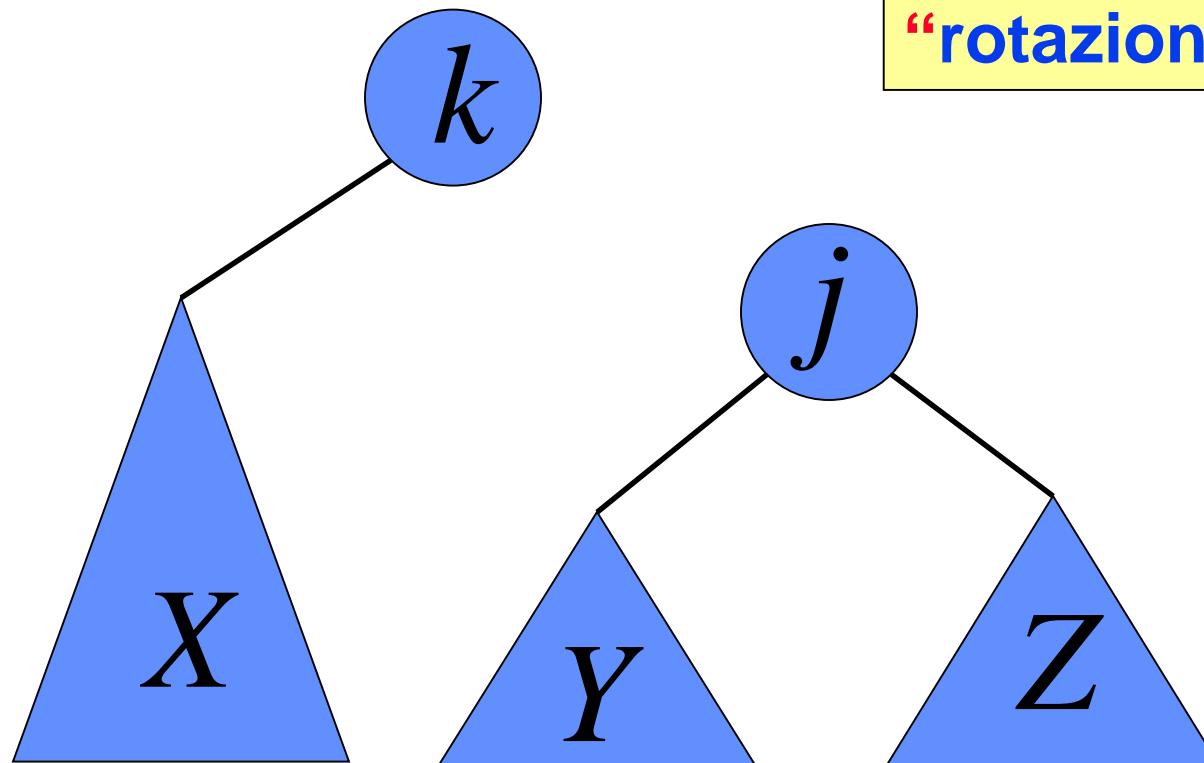
Facciamo una
“rotazione singola”



Rotazione singola in alberi AVL

Passo dopo passo

Facciamo una
“rotazione singola”



Rotazione singola in alberi AVL

Passo dopo passo

altezza $a + 1$

k

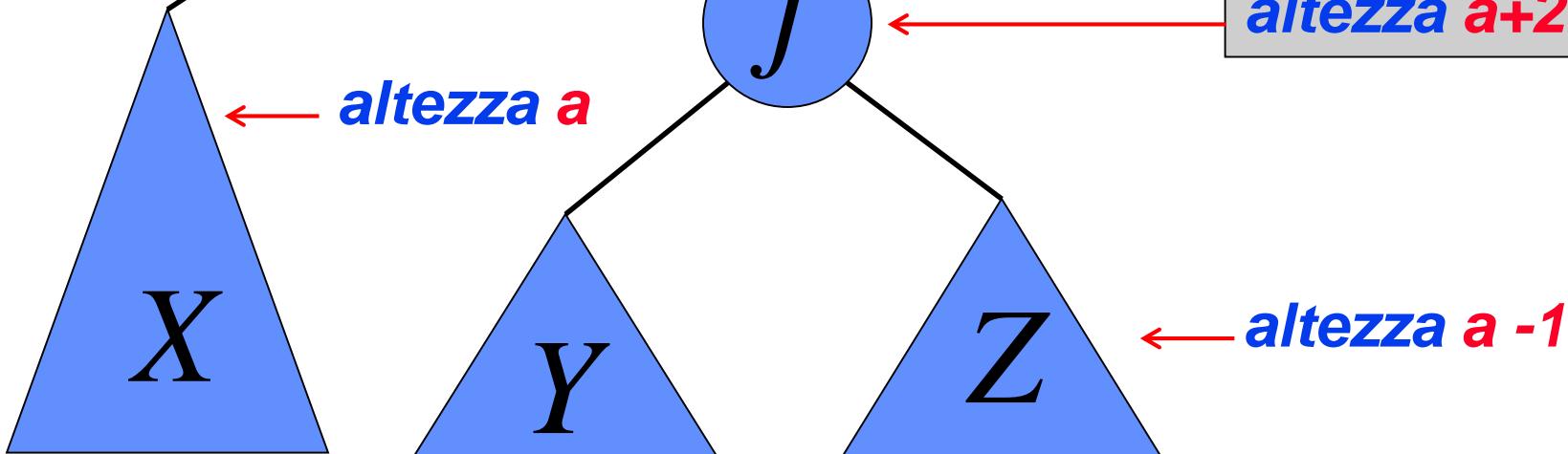
Facciamo una
“rotazione singola”

Fatto!!!

altezza $a+2$

j

altezza a



Le altezze dei nodi K e J sono state aggiornate

Rotazione singola in alberi AVL

Passo dopo passo

altezza $a + 1$

k

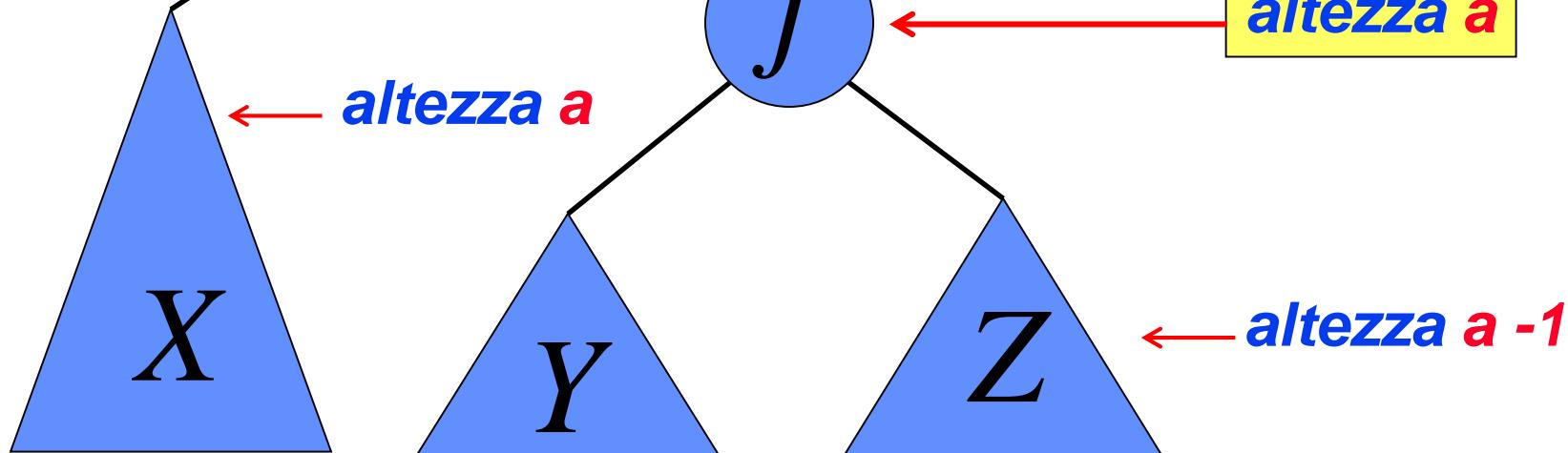
Facciamo una
“rotazione singola”

Fatto!!!

altezza a

j

altezza a



Le altezze dei nodi K e J sono state aggiornate

Rotazione singola: algoritmo

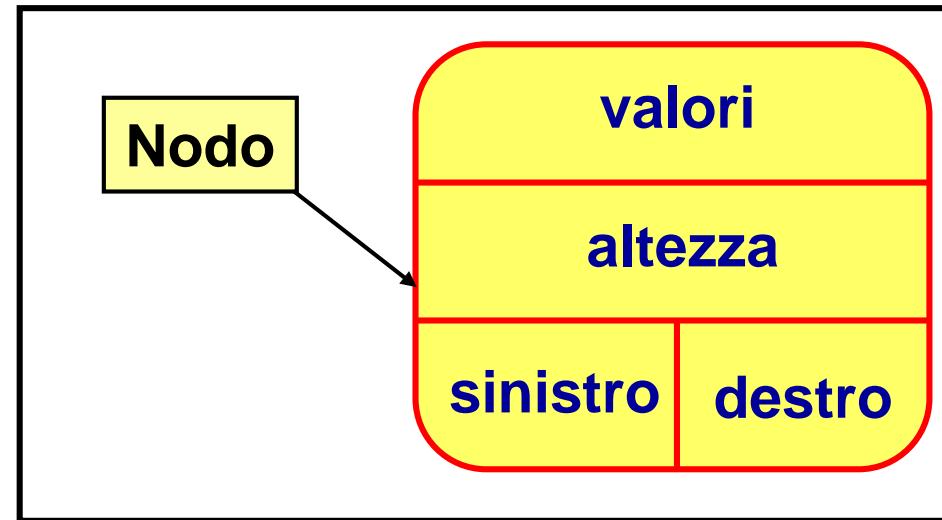
Altezza(T : albero-AVL)

IF $T = NIL$ **THEN**

return -1

ELSE

return $T->altezza$



Rotazione singola: algoritmo

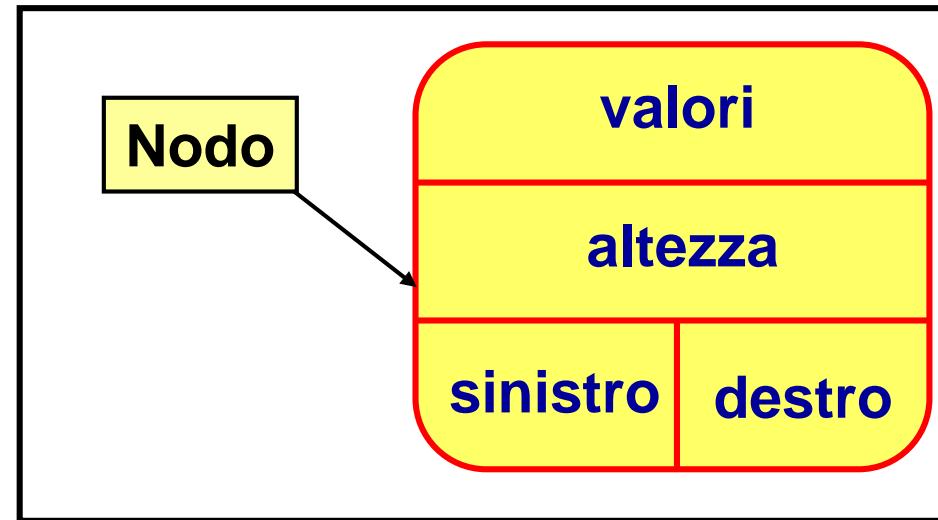
Altezza(T : albero-AVL)

IF $T = NIL$ **THEN**

return -1

ELSE

return $T->altezza$



Rotazione-SS(T : albero-AVL)

$\text{newroot} = T->sx$

$T->sx = \text{newroot}->dx$

$\text{newroot}->dx = T$

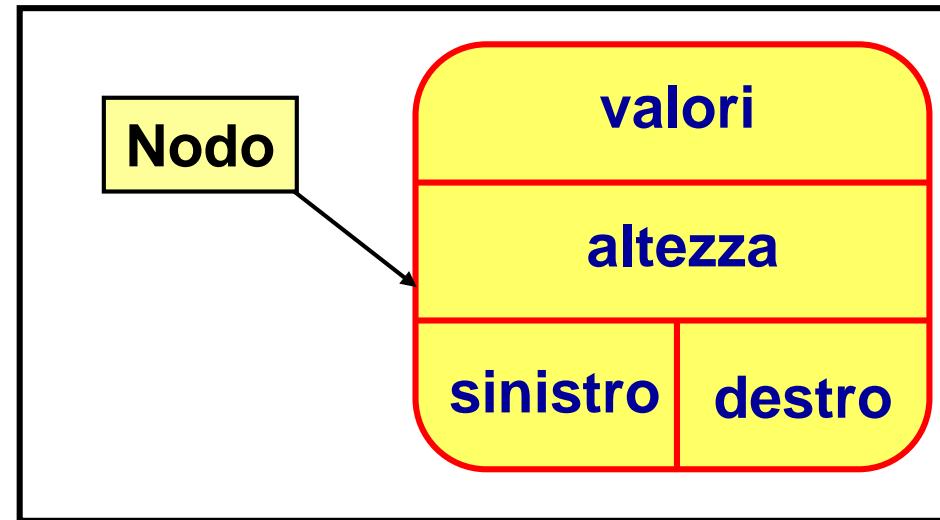
$T->altezza = \max(\text{Altezza}(T->sx), \text{Altezza}(T->dx)) + 1$

$\text{newroot}->altezza = \max(\text{Altezza}(\text{newroot}->sx),$
 $\text{Altezza}(\text{newroot}->dx)) + 1$

return newroot

Rotazione singola: algoritmo

```
Altezza(T : albero-AVL)
  IF T = NIL
    THEN return -1
  ELSE return T->altezza
```

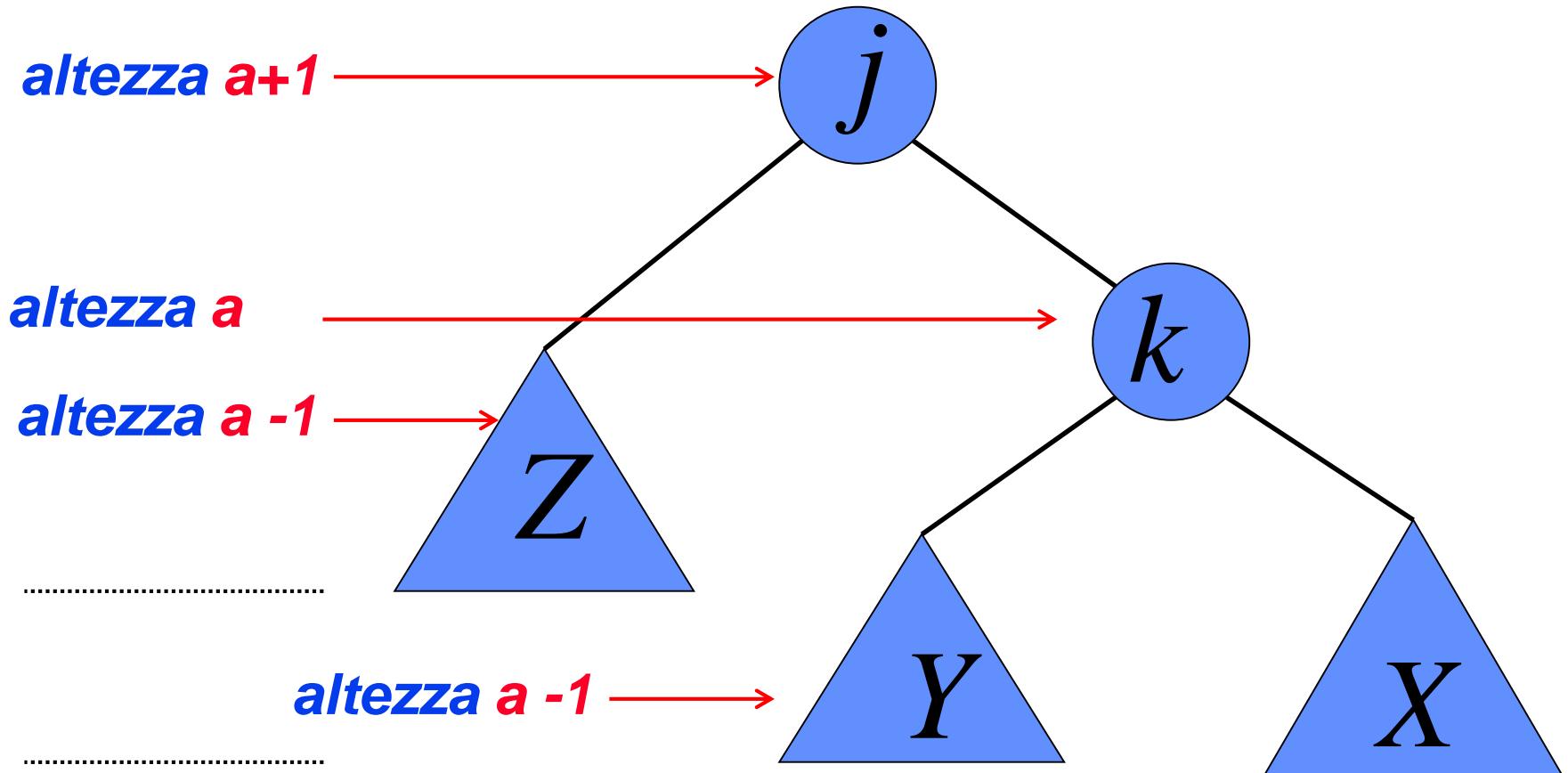


```
Rotazione-SS(T : albero-AVL)
  newroot = T->sx
  T->sx = newroot->dx
  newroot->dx = T
  T->altezza = max(Altezza(T->sx), Altezza(T->dx)) + 1
  newroot->altezza = max(Altezza(newroot->sx),
                           Altezza(newroot->dx)) + 1
  return newroot
```

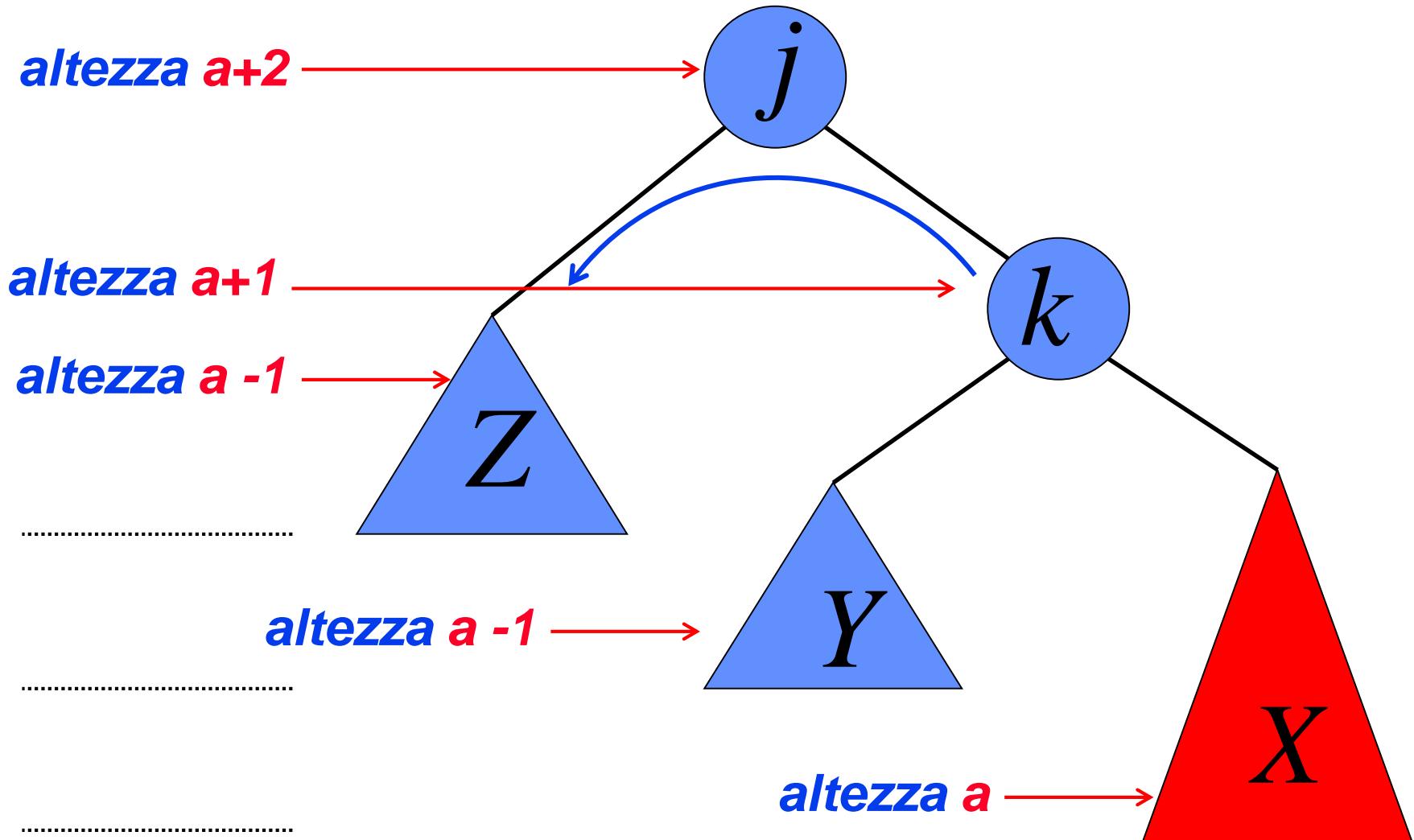
Rotazione

Aggiornamento altezze

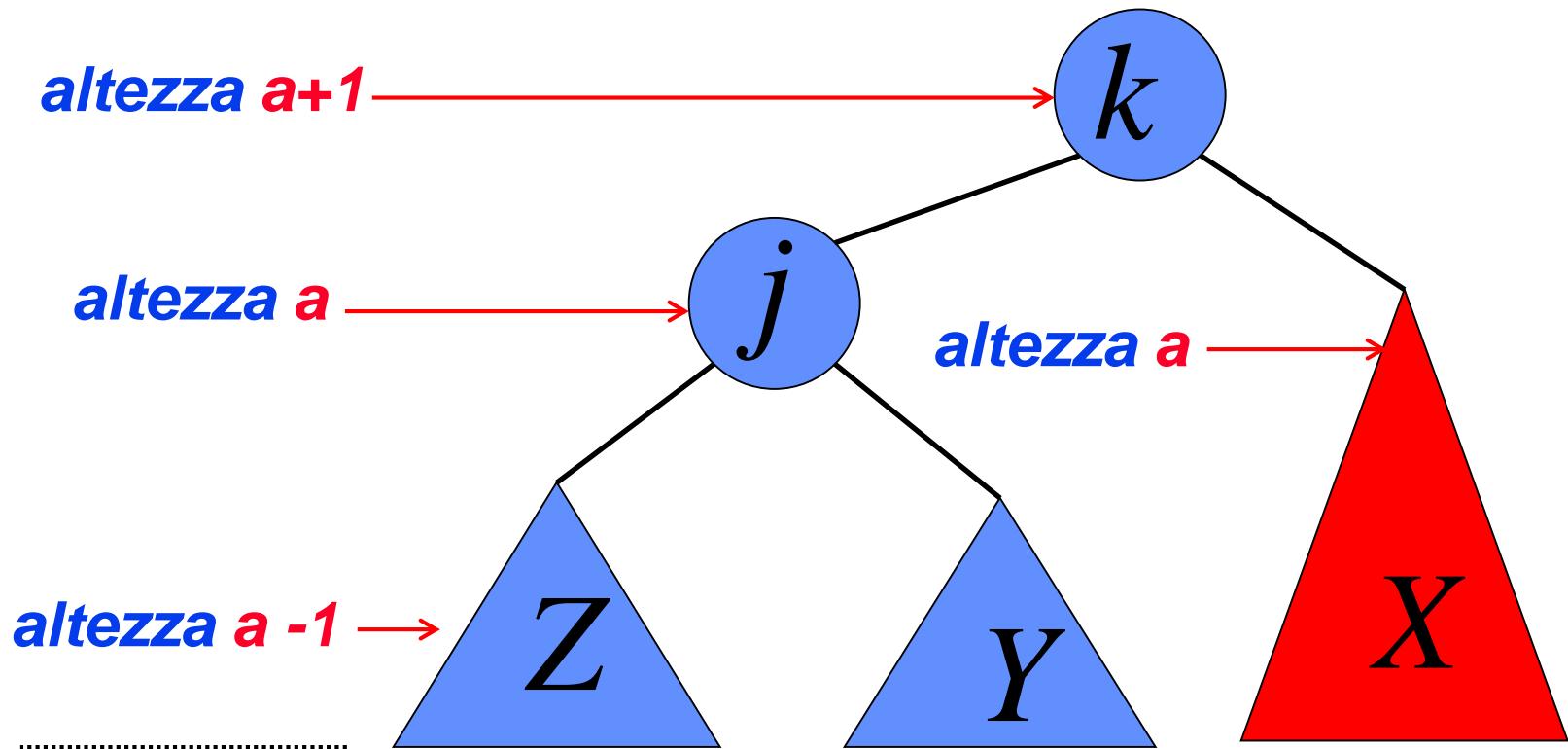
Rotazione in alberi AVL: caso I



Rotazione in alberi AVL: caso I



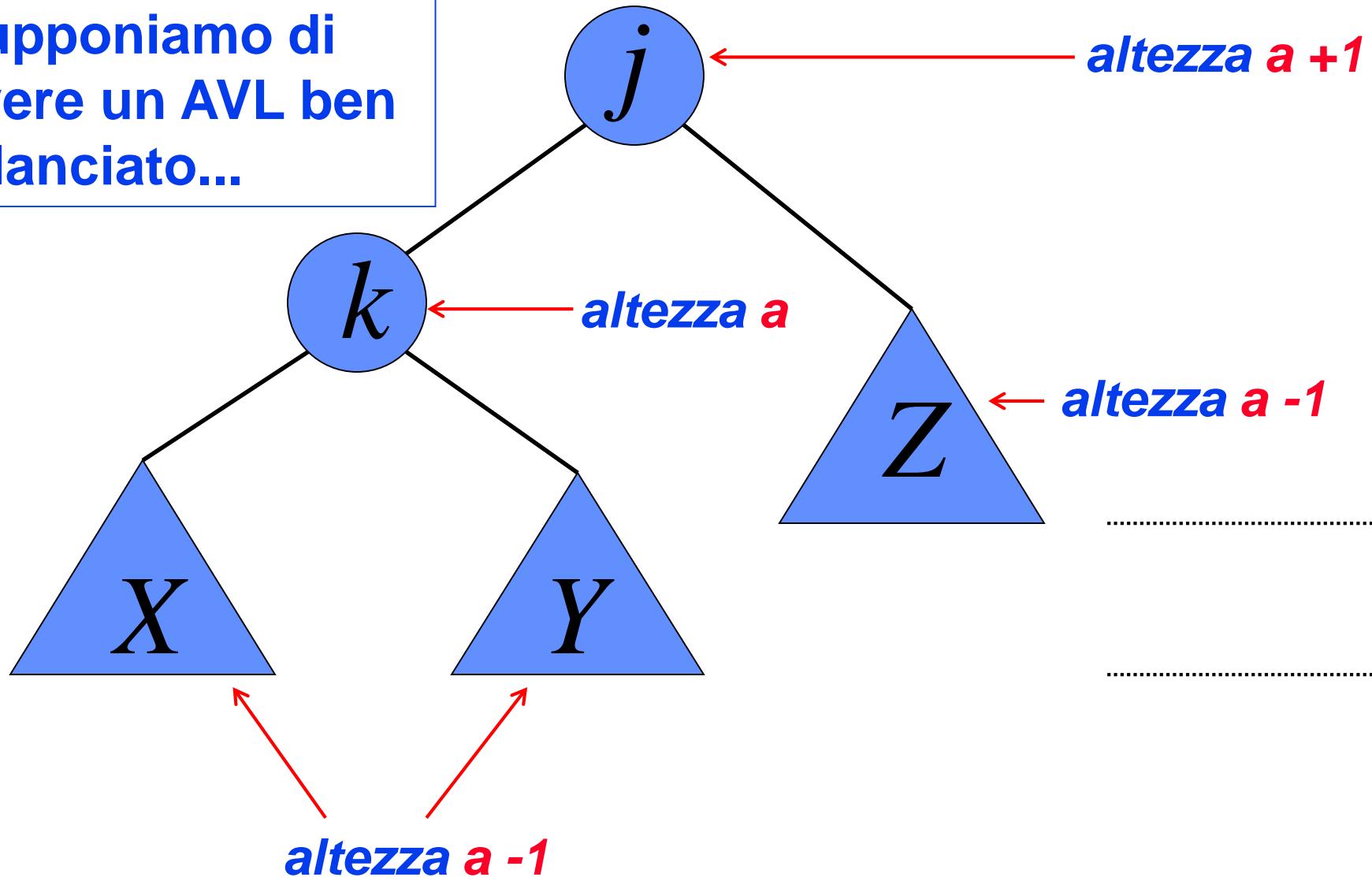
Rotazione in alberi AVL: caso I



Rotazione singola destra

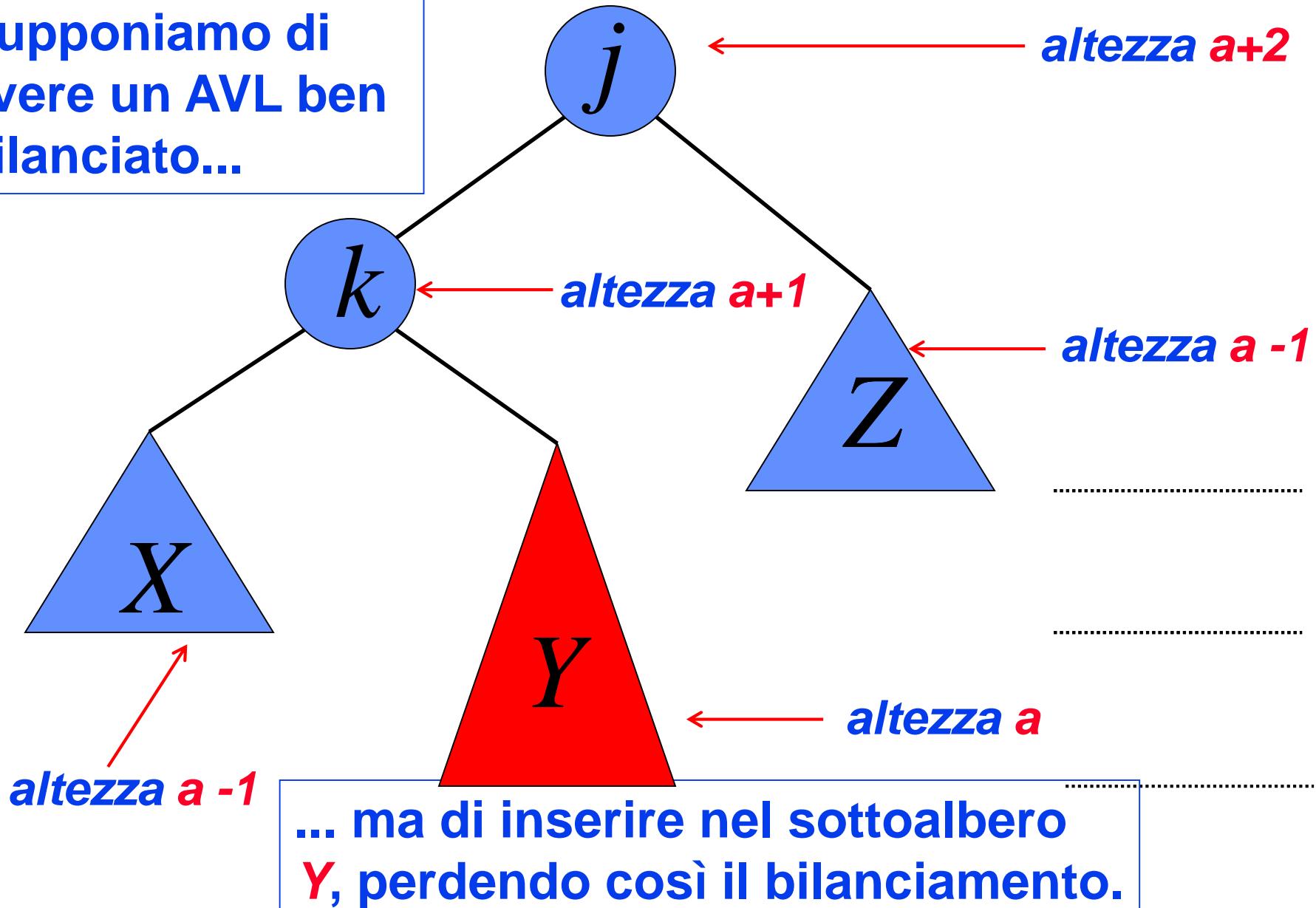
Rotazione in alberi AVL: caso II

Supponiamo di avere un AVL ben bilanciato...

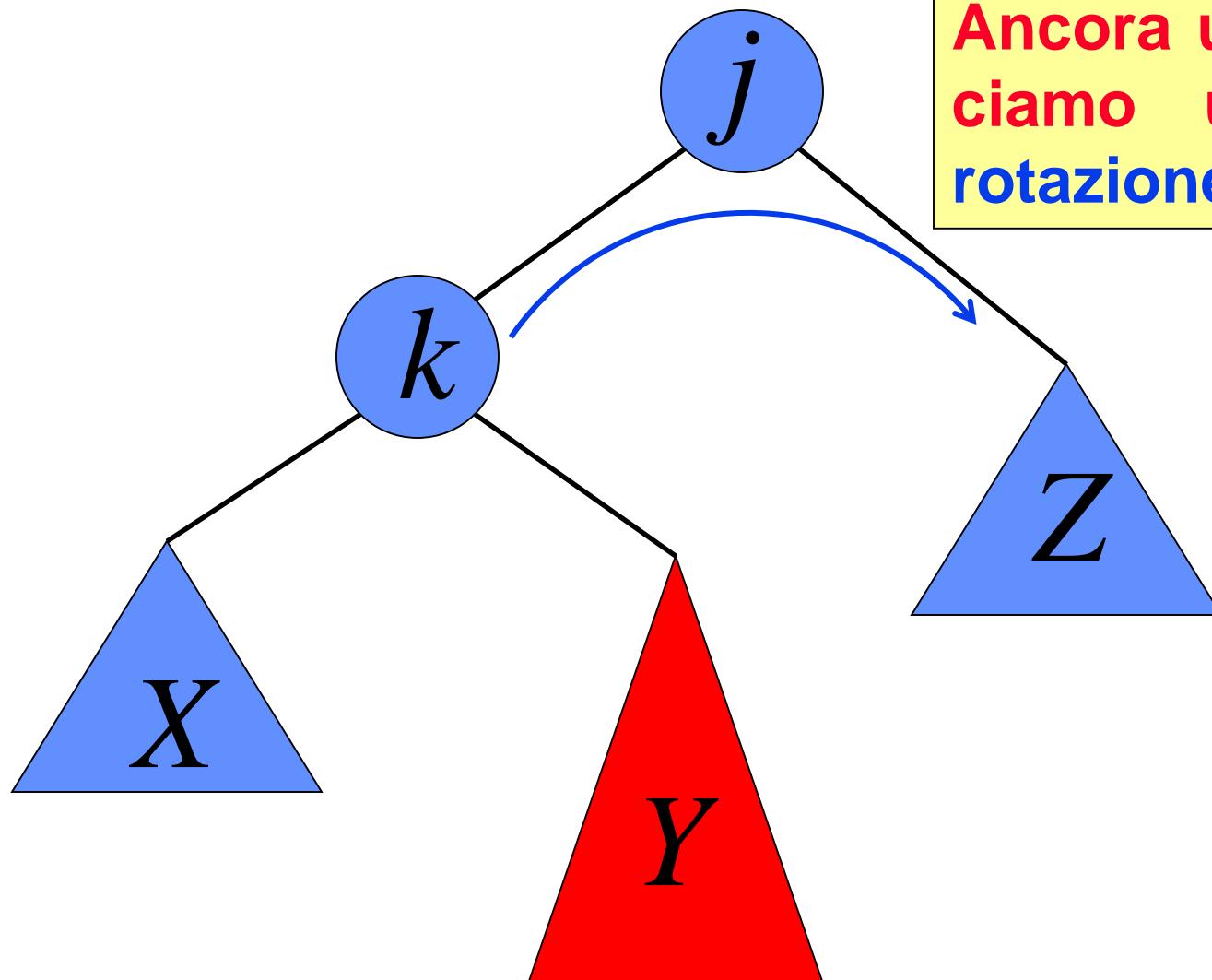


Rotazione in alberi AVL: caso II

Supponiamo di avere un AVL ben bilanciato...

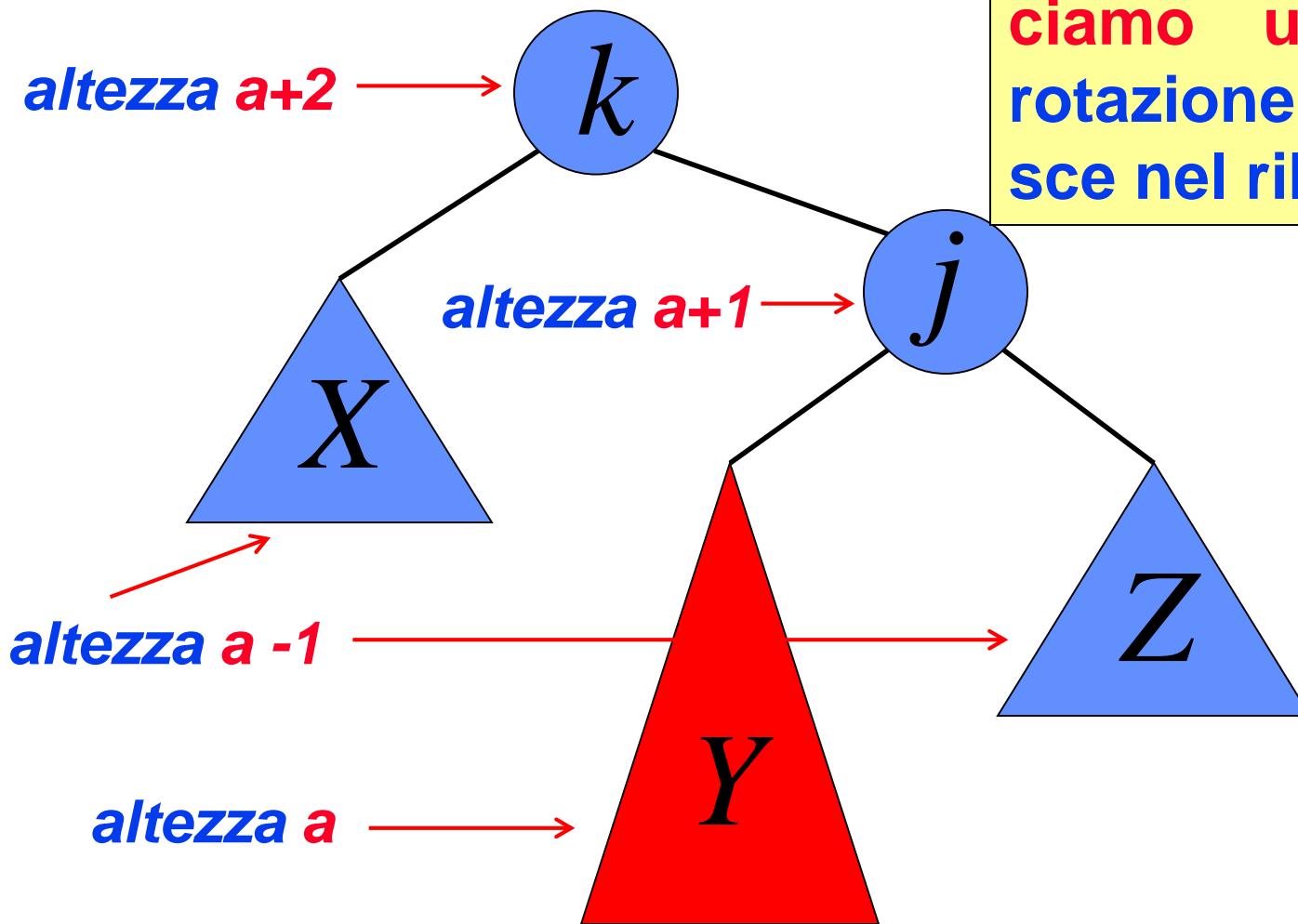


Rotazione in alberi AVL: caso II



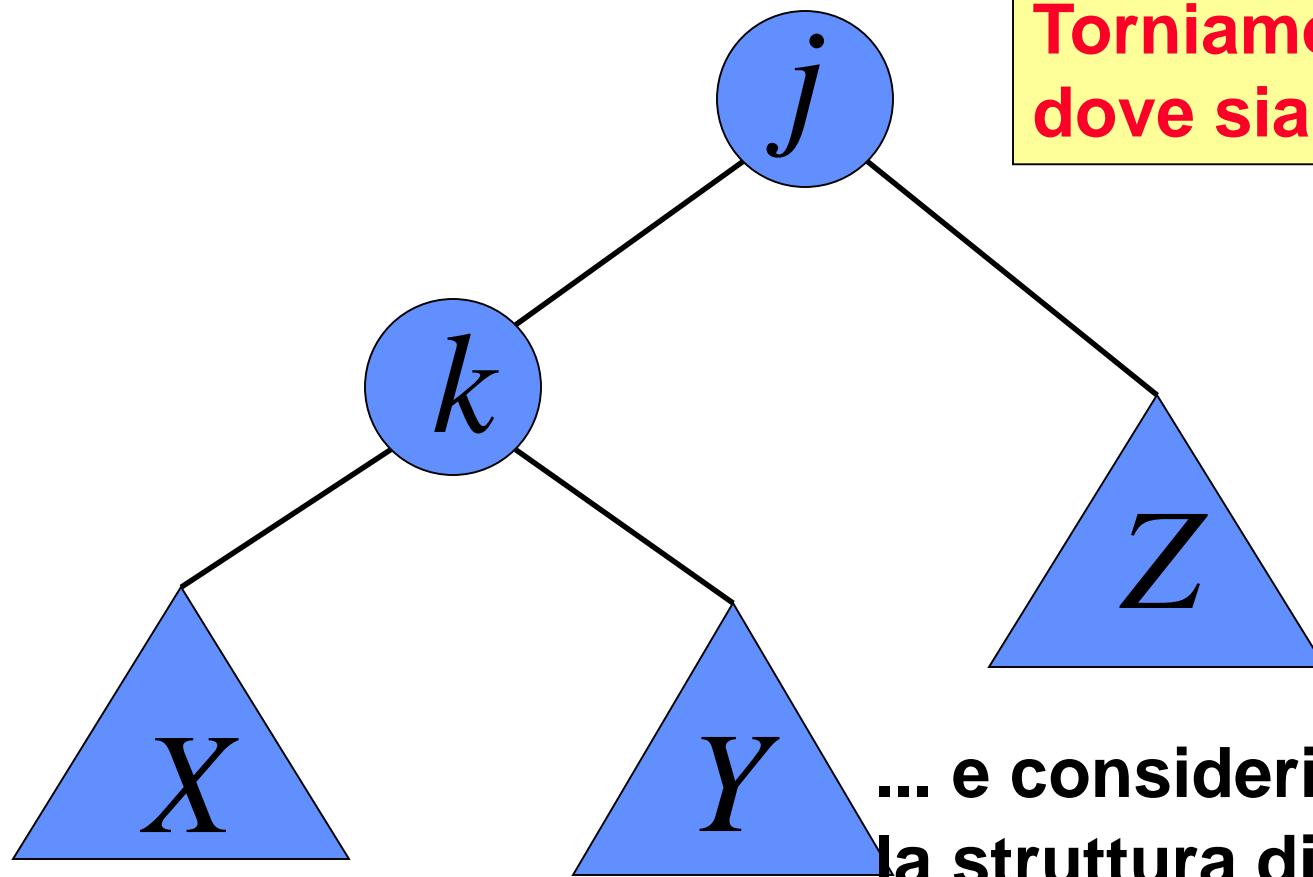
Ancora una volta, facciamo una “singola rotazione”

Rotazione in alberi AVL: caso II



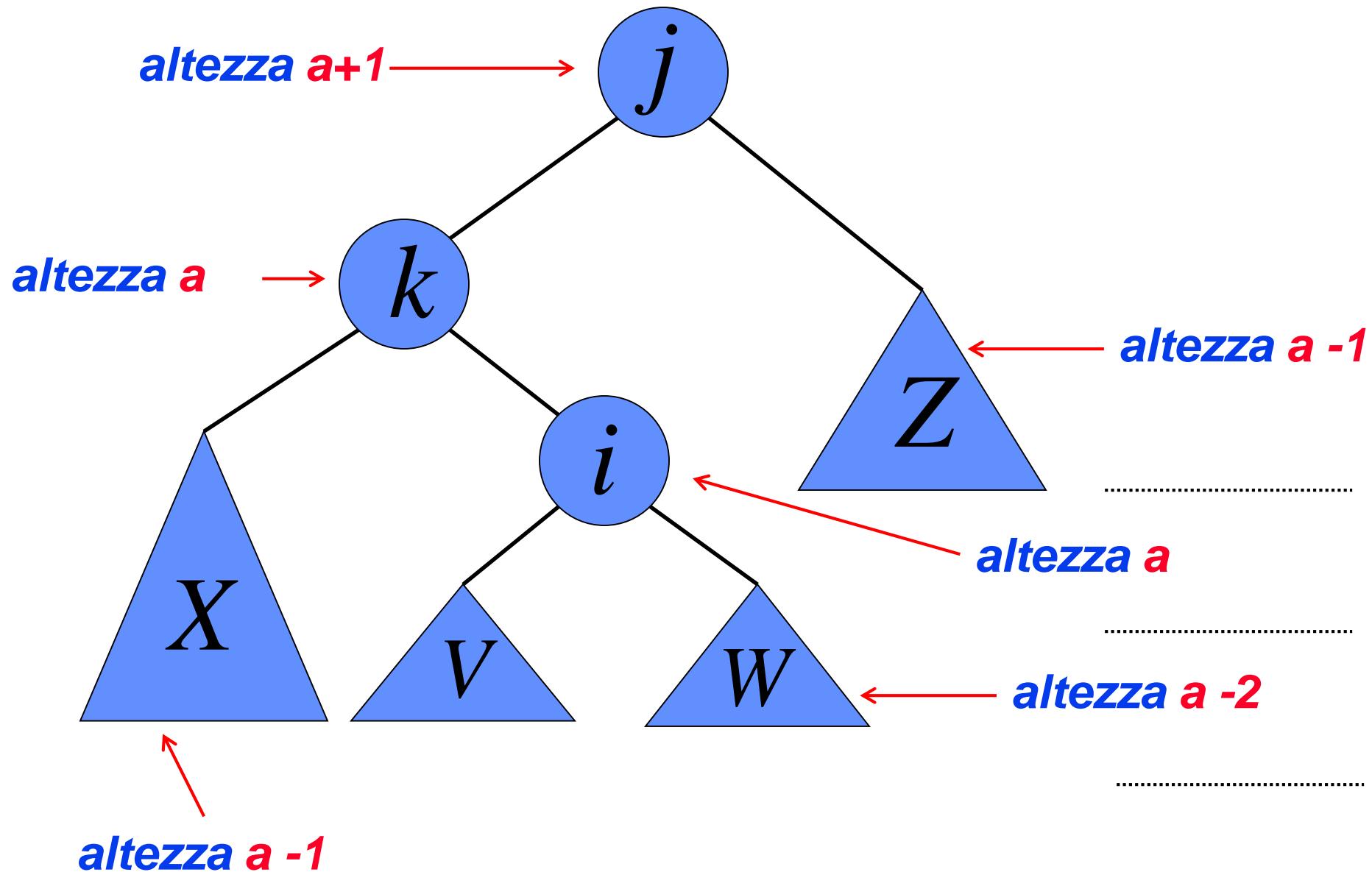
Ancora una volta, facciamo una “singola rotazione”... ma fallisce nel ribilanciare!

Rotazione in alberi AVL: caso II

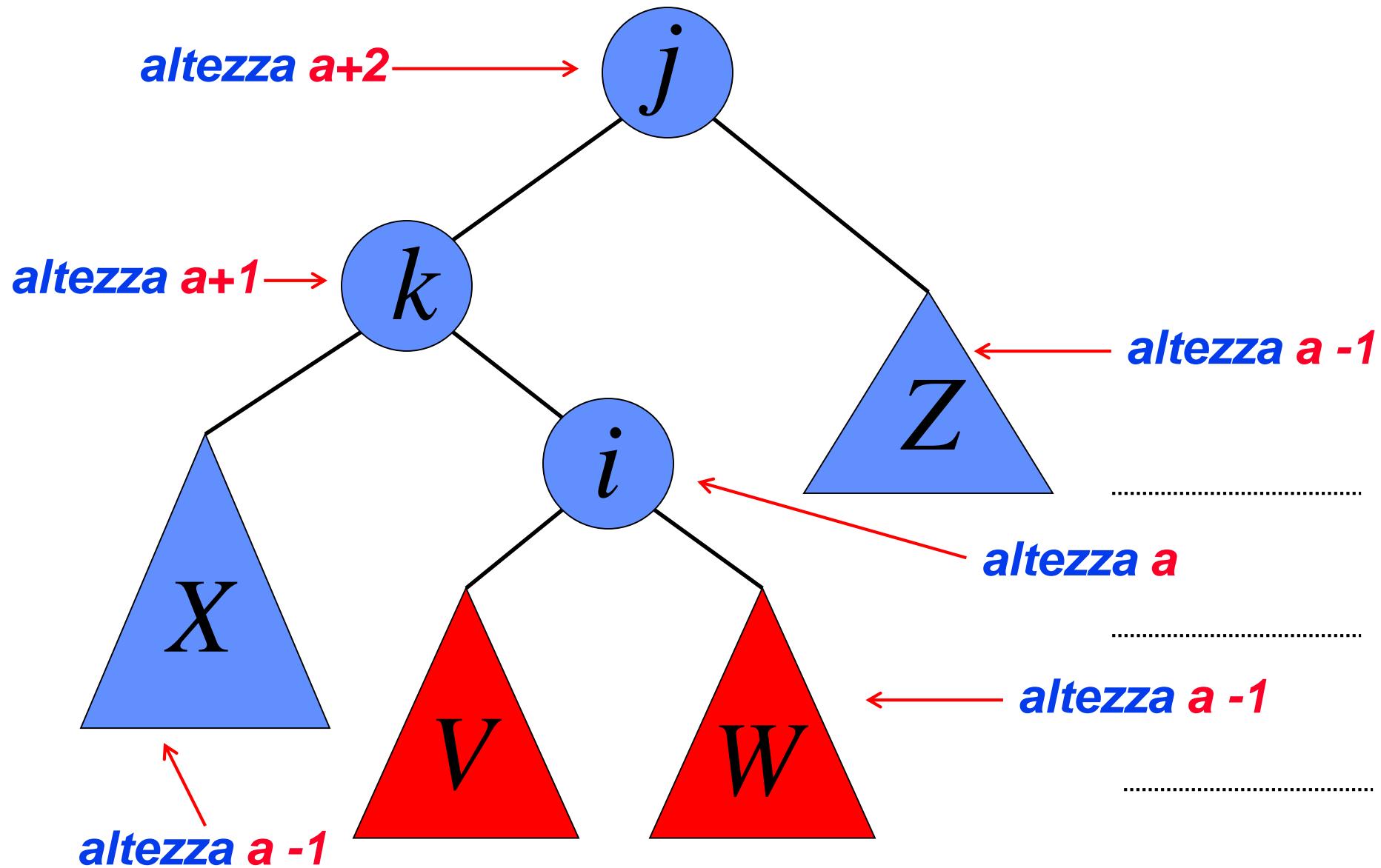


... e consideriamo
la struttura di Y

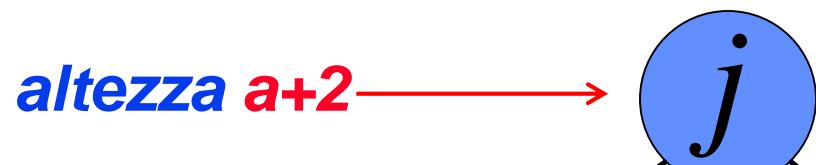
Rotazione in alberi AVL: caso II



Rotazione in alberi AVL: caso II

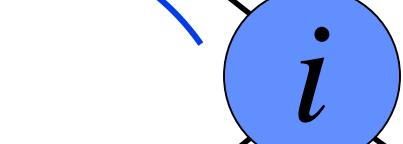
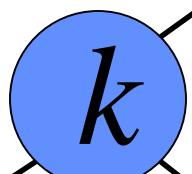


Rotazione in alberi AVL: caso II

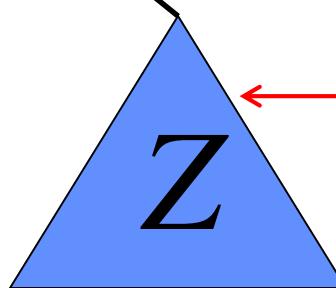


Facciamo una “singola rotazione destra”

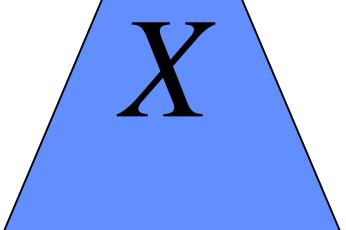
altezza $a+1 \rightarrow$



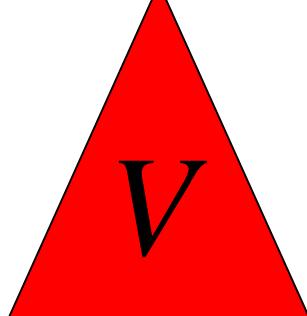
altezza $a - 1$



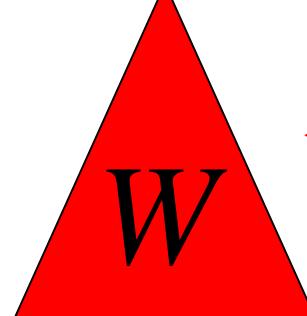
altezza a



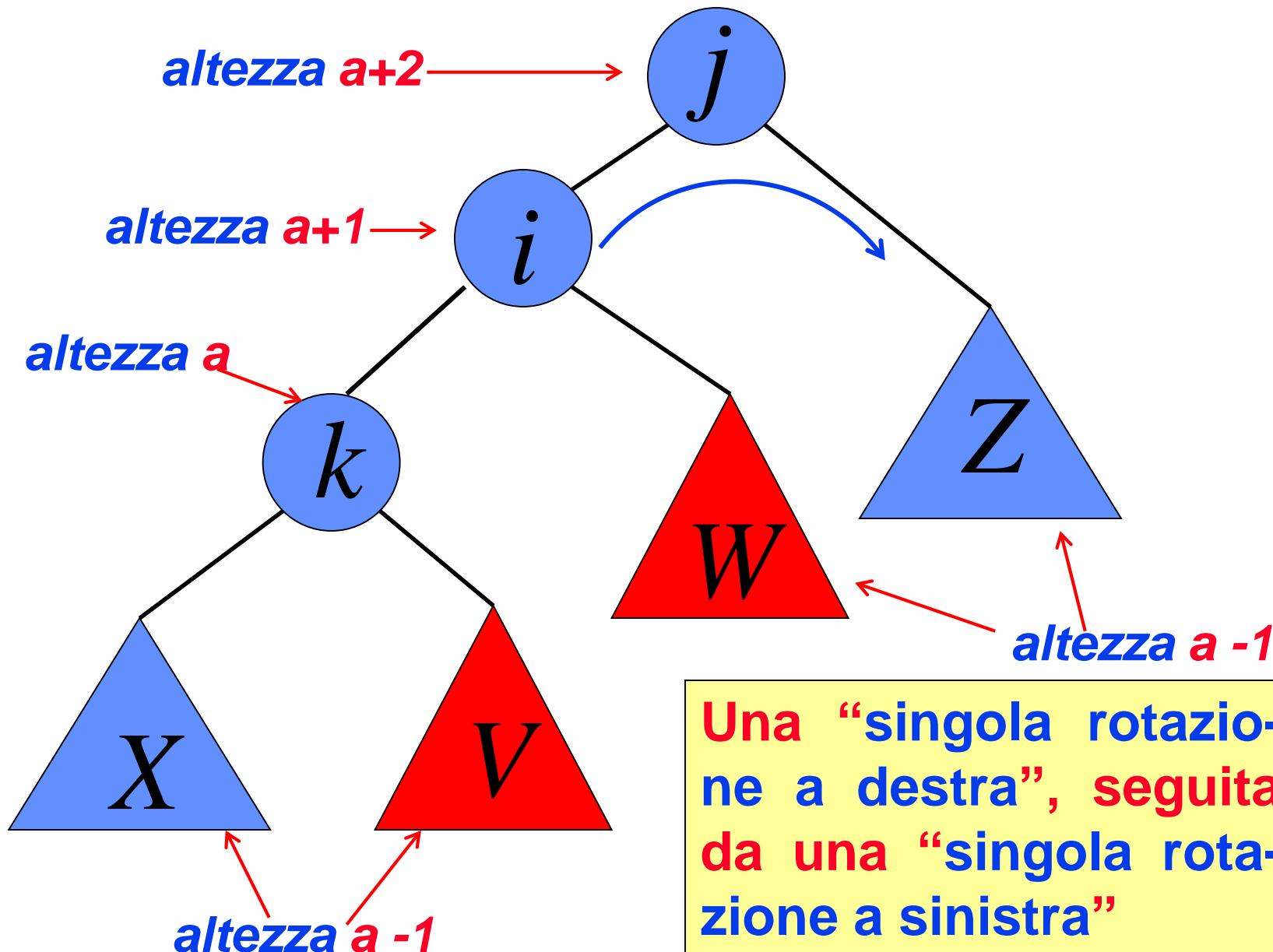
altezza $a - 1$



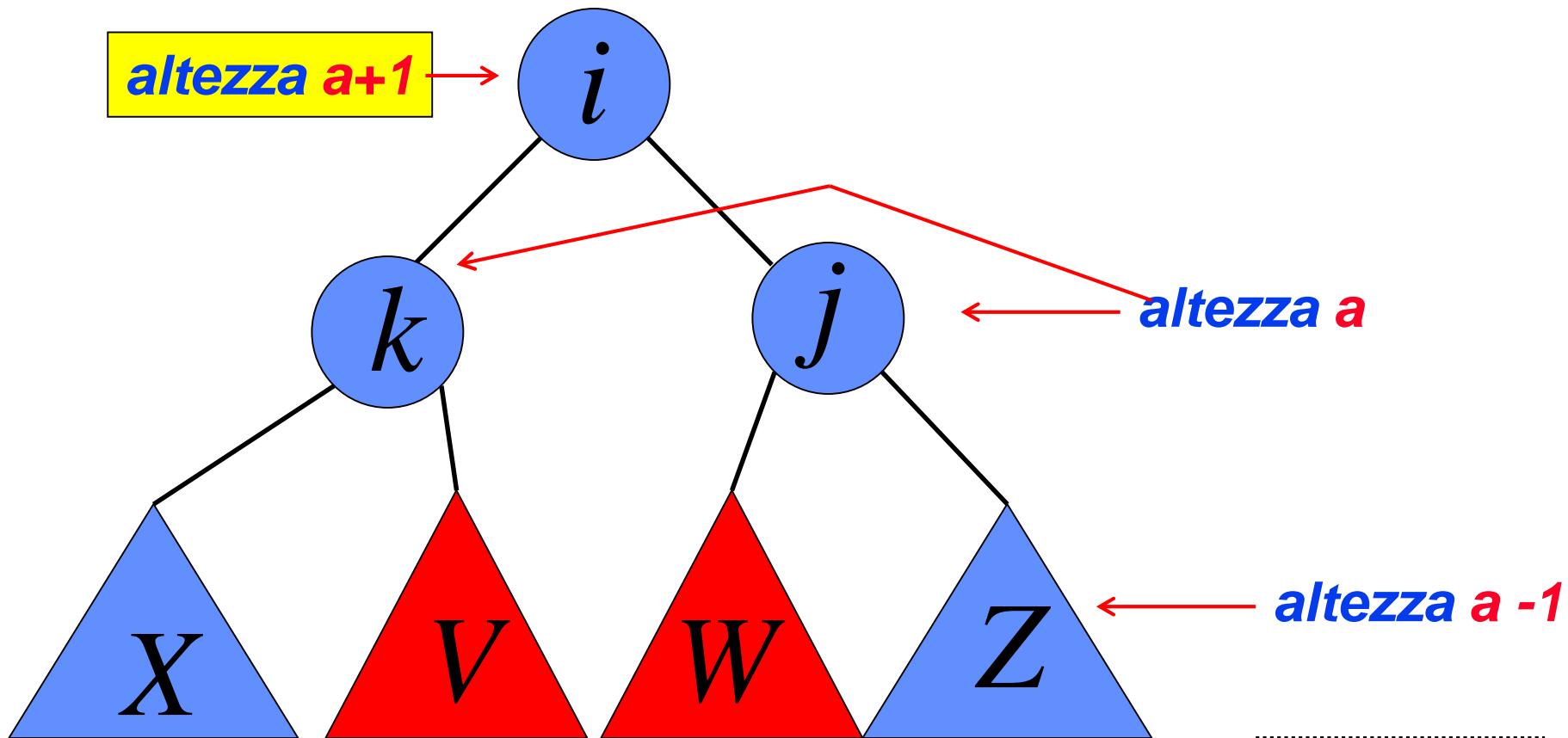
altezza $a - 1$



Rotazione in alberi AVL: caso II

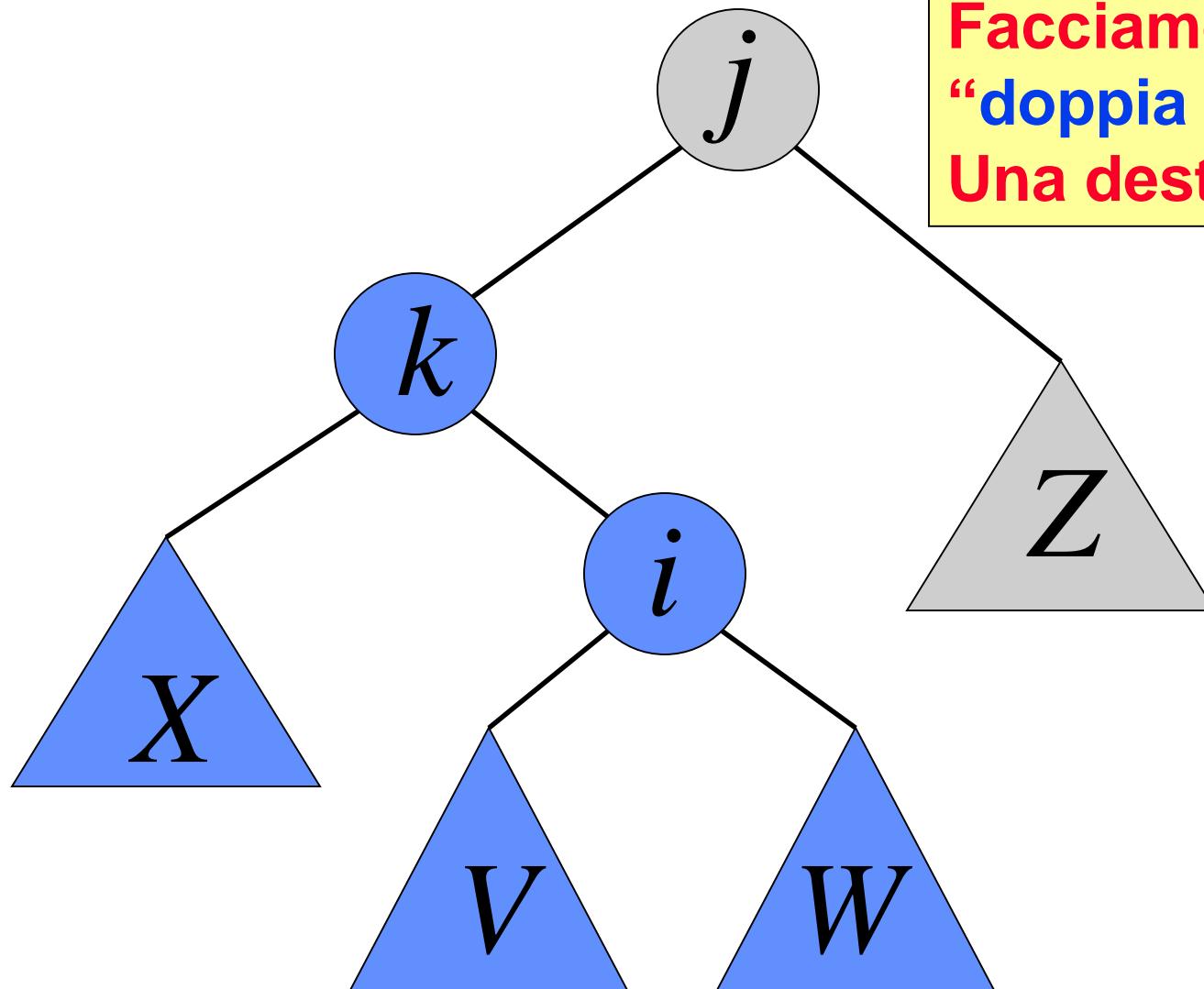


Rotazione in alberi AVL: caso II



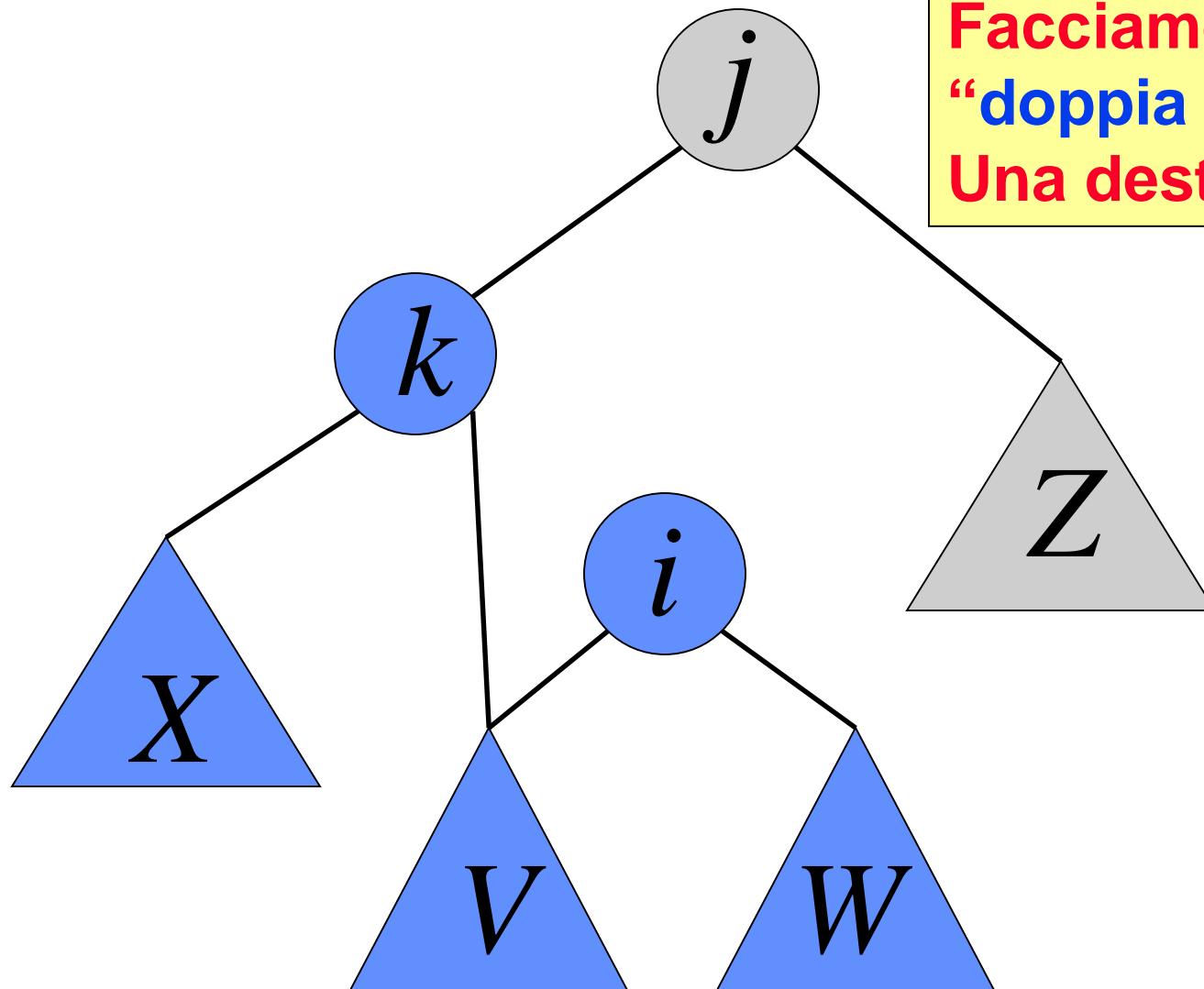
Rotazione doppia sinistra

Rotazione doppia in alberi AVL



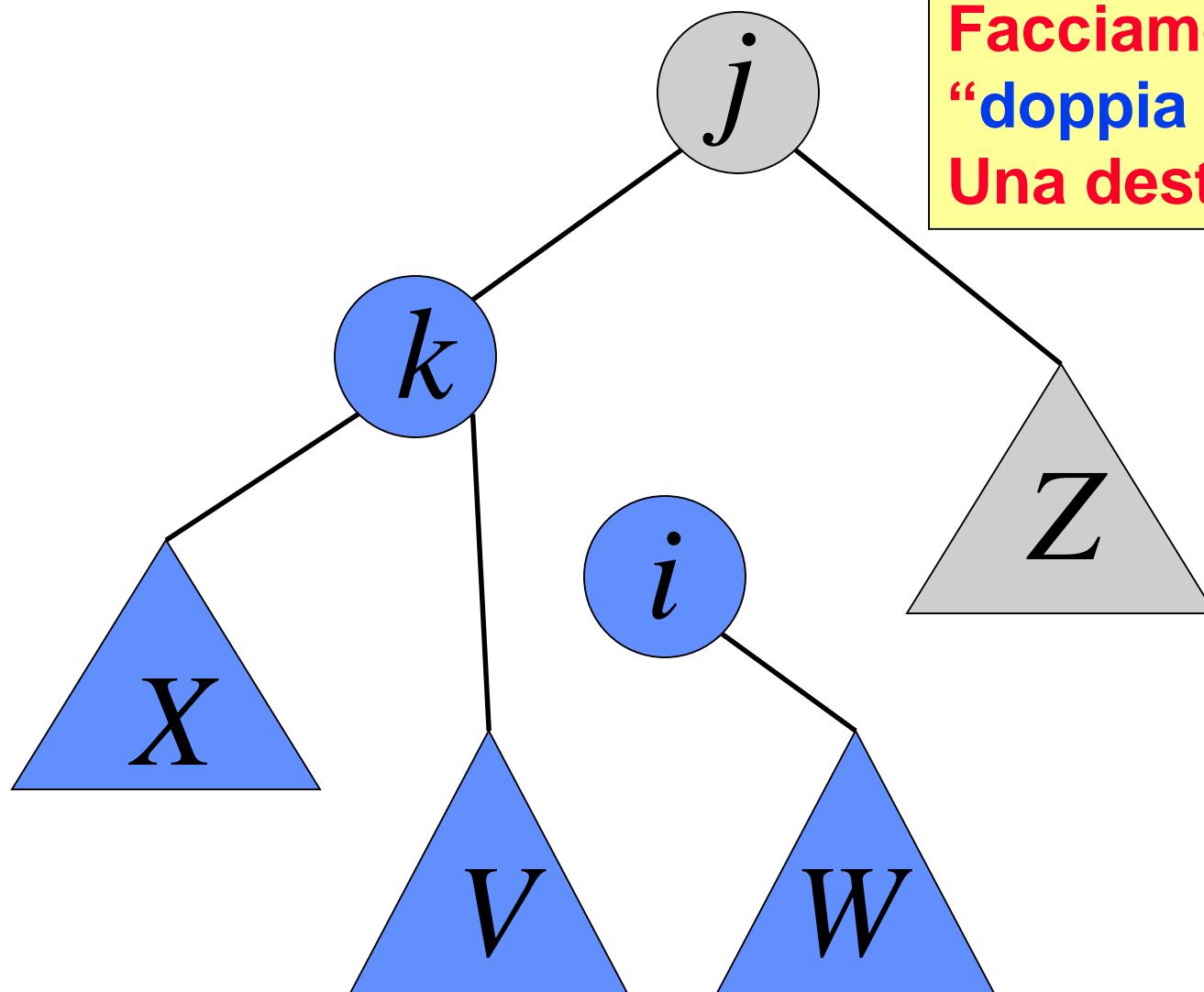
Facciamo ora una
“doppia rotazione”
Una destra

Rotazione doppia in alberi AVL



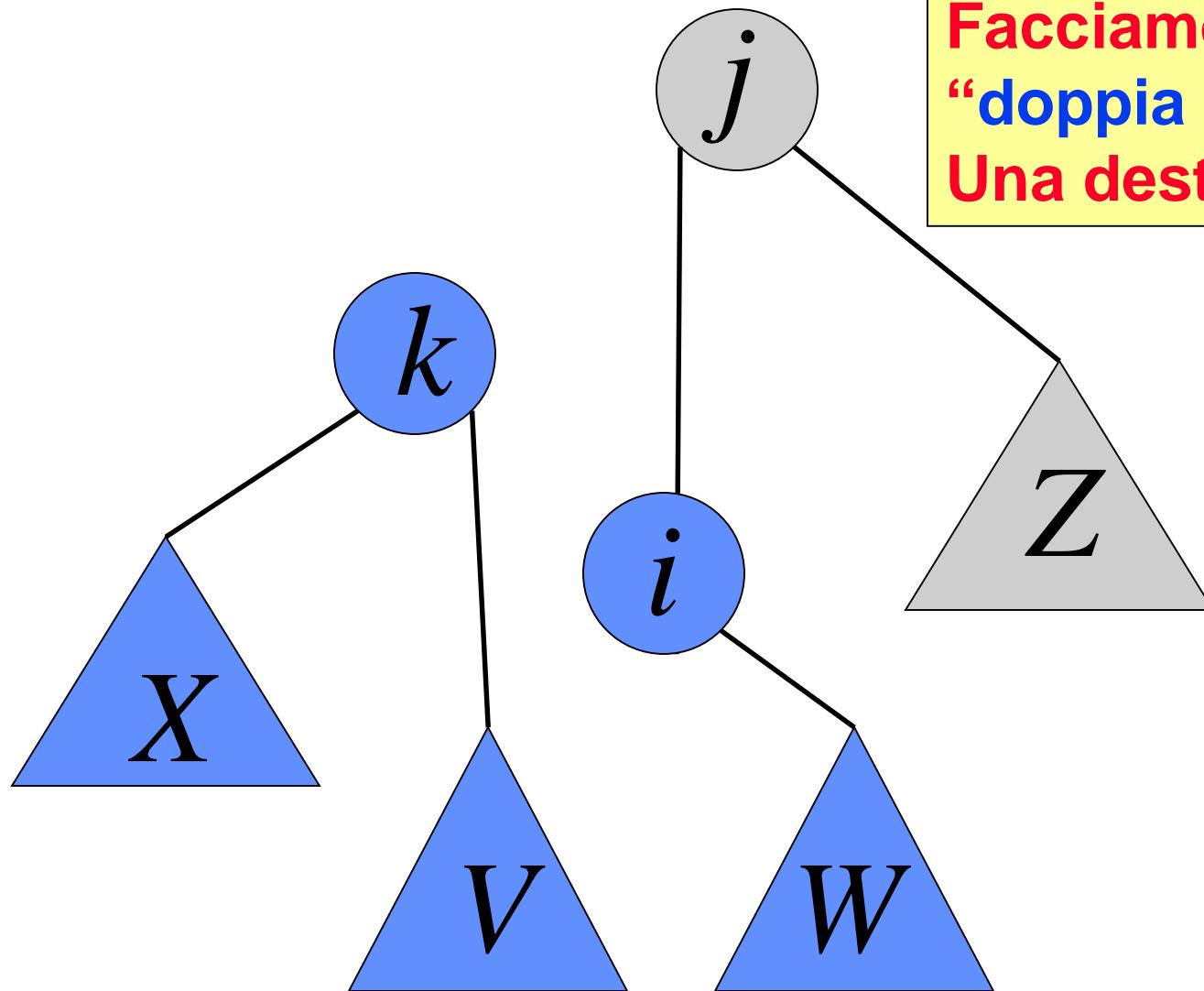
Facciamo ora una
“doppia rotazione”
Una destra

Rotazione doppia in alberi AVL



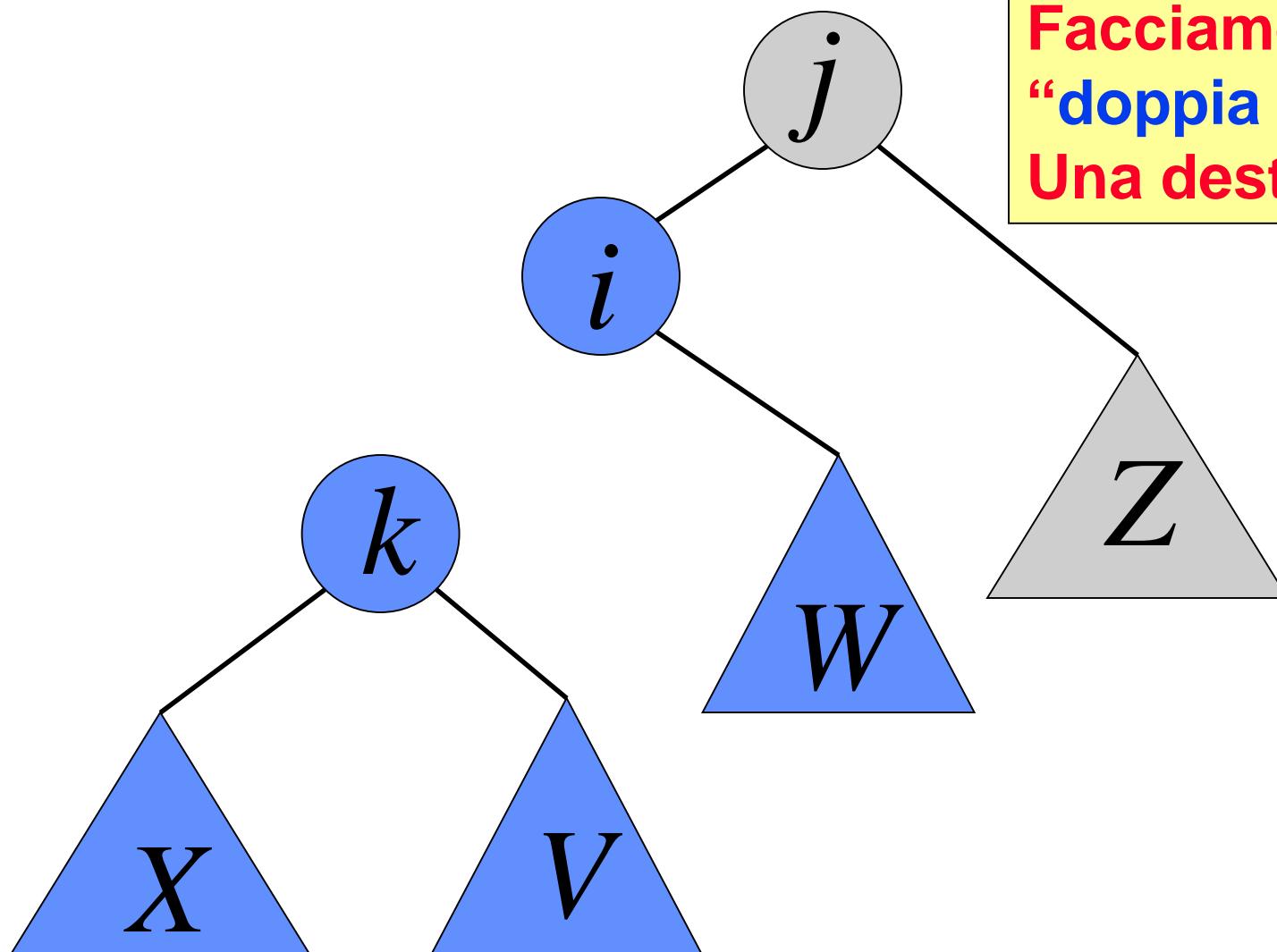
Facciamo ora una
“doppia rotazione”
Una destra

Rotazione doppia in alberi AVL



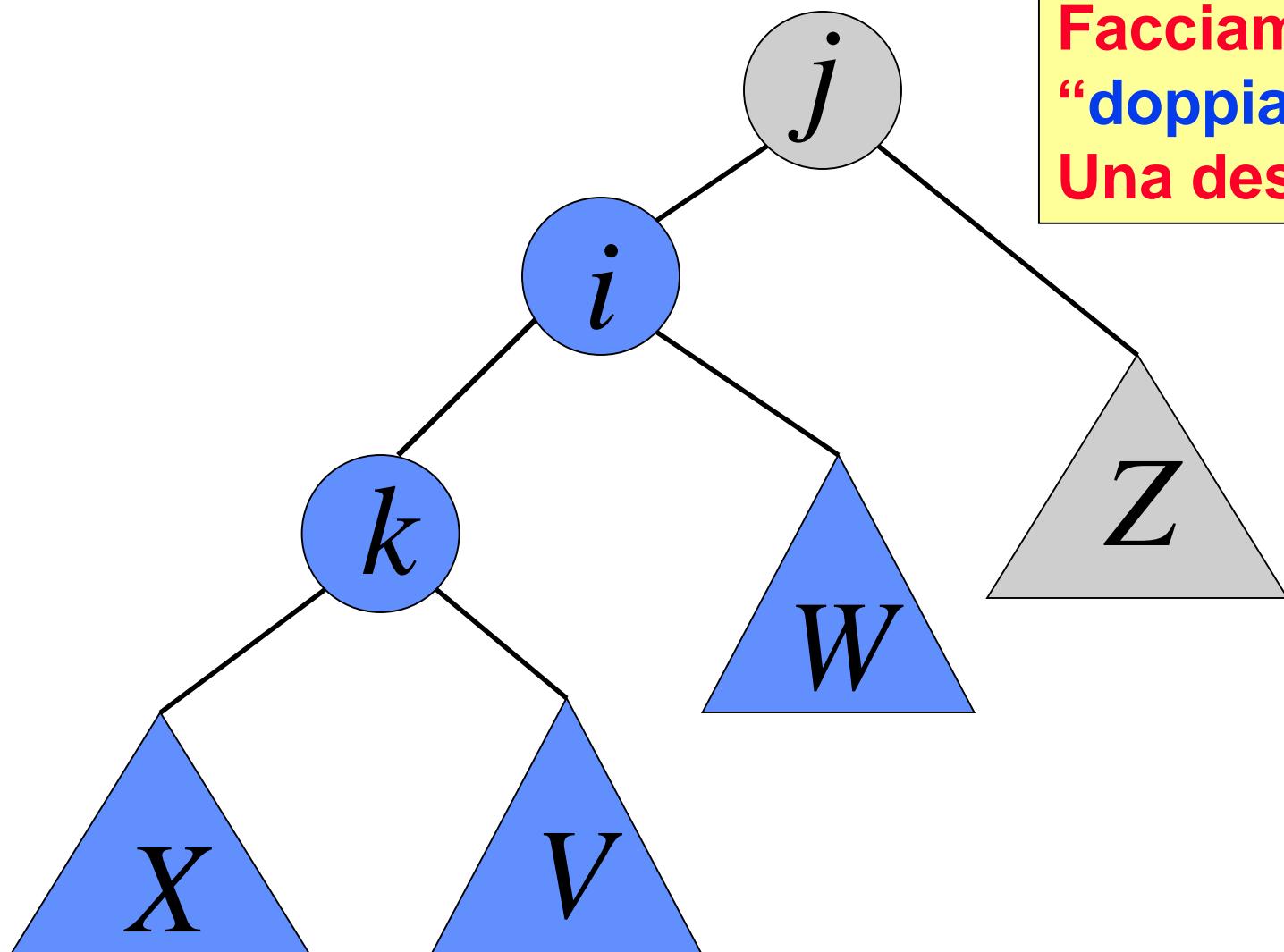
Facciamo ora una
“doppia rotazione”
Una destra

Rotazione doppia in alberi AVL



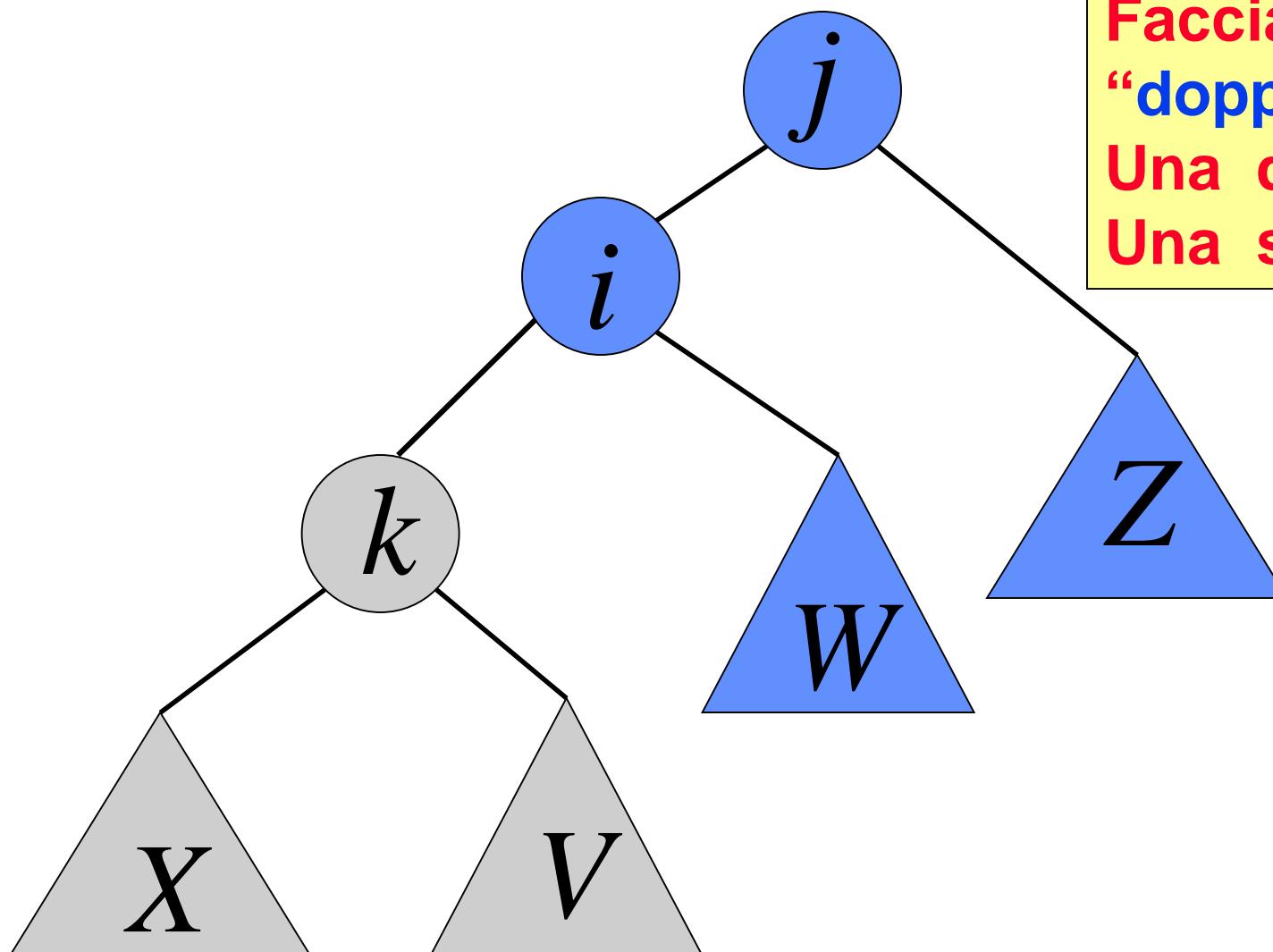
Facciamo ora una
“doppia rotazione”
Una destra

Rotazione doppia in alberi AVL



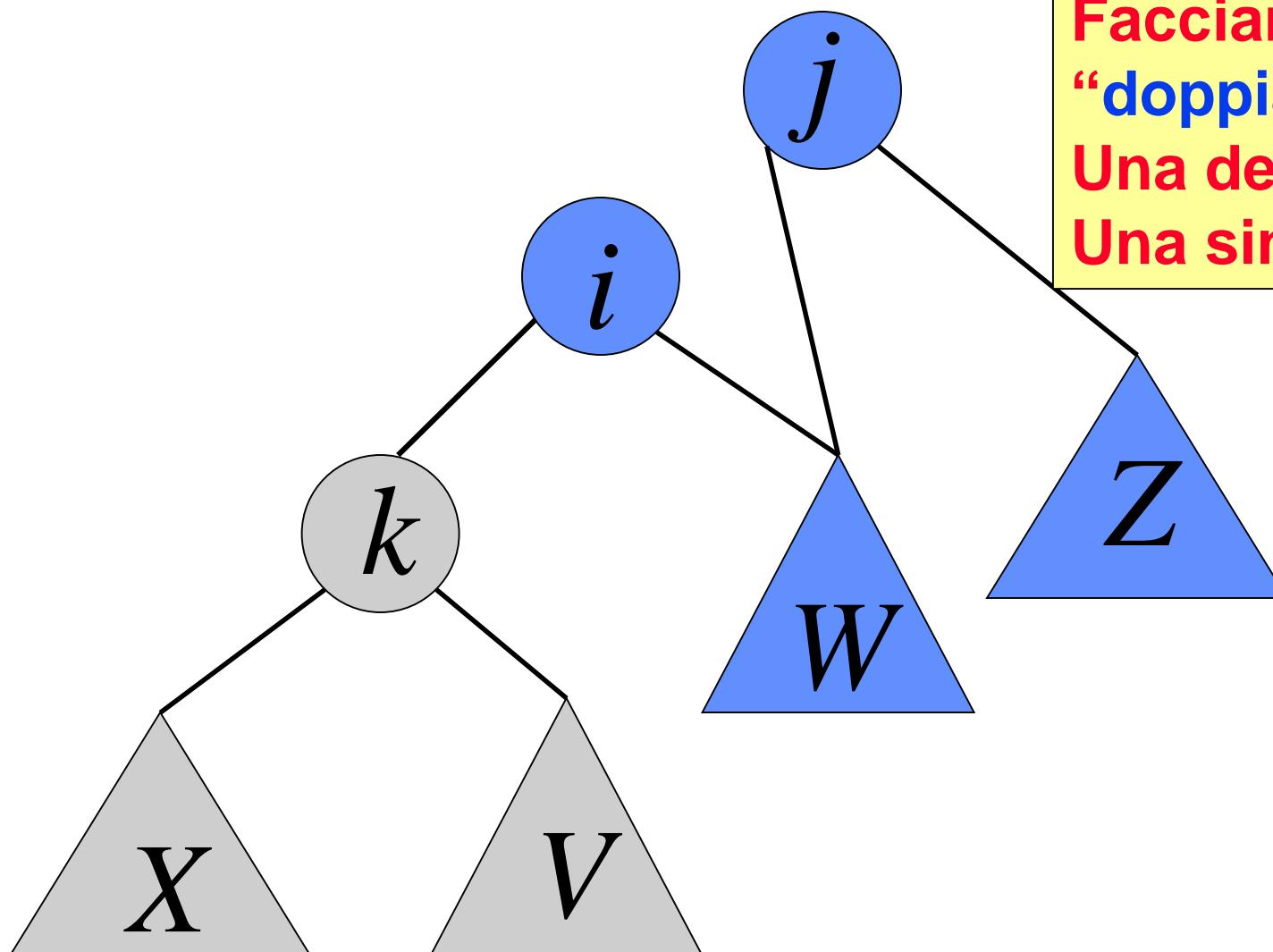
Facciamo ora una
“doppia rotazione”
Una destra

Rotazione doppia in alberi AVL



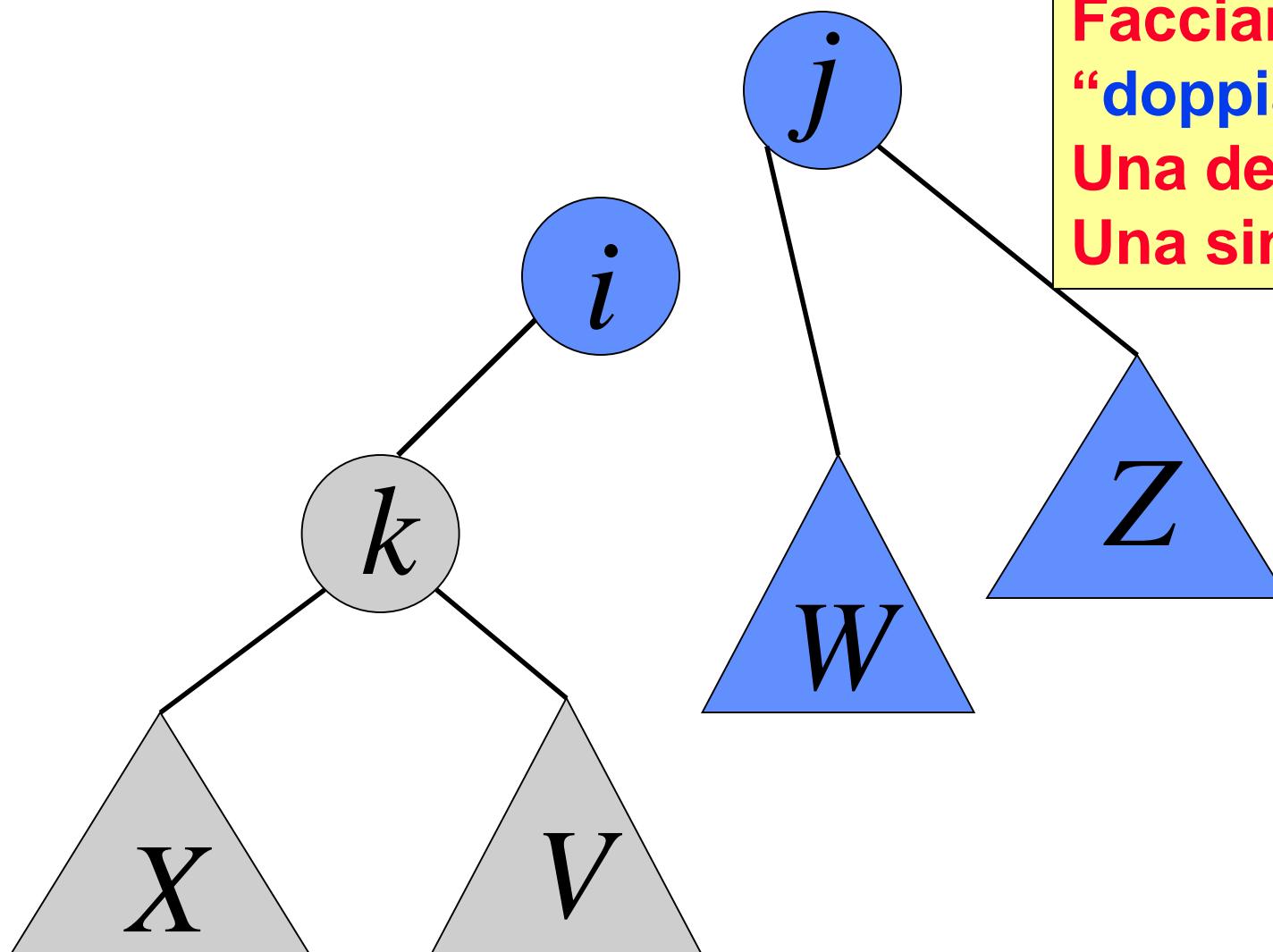
Facciamo ora una
“doppia rotazione”
Una destra
Una sinistra

Rotazione doppia in alberi AVL



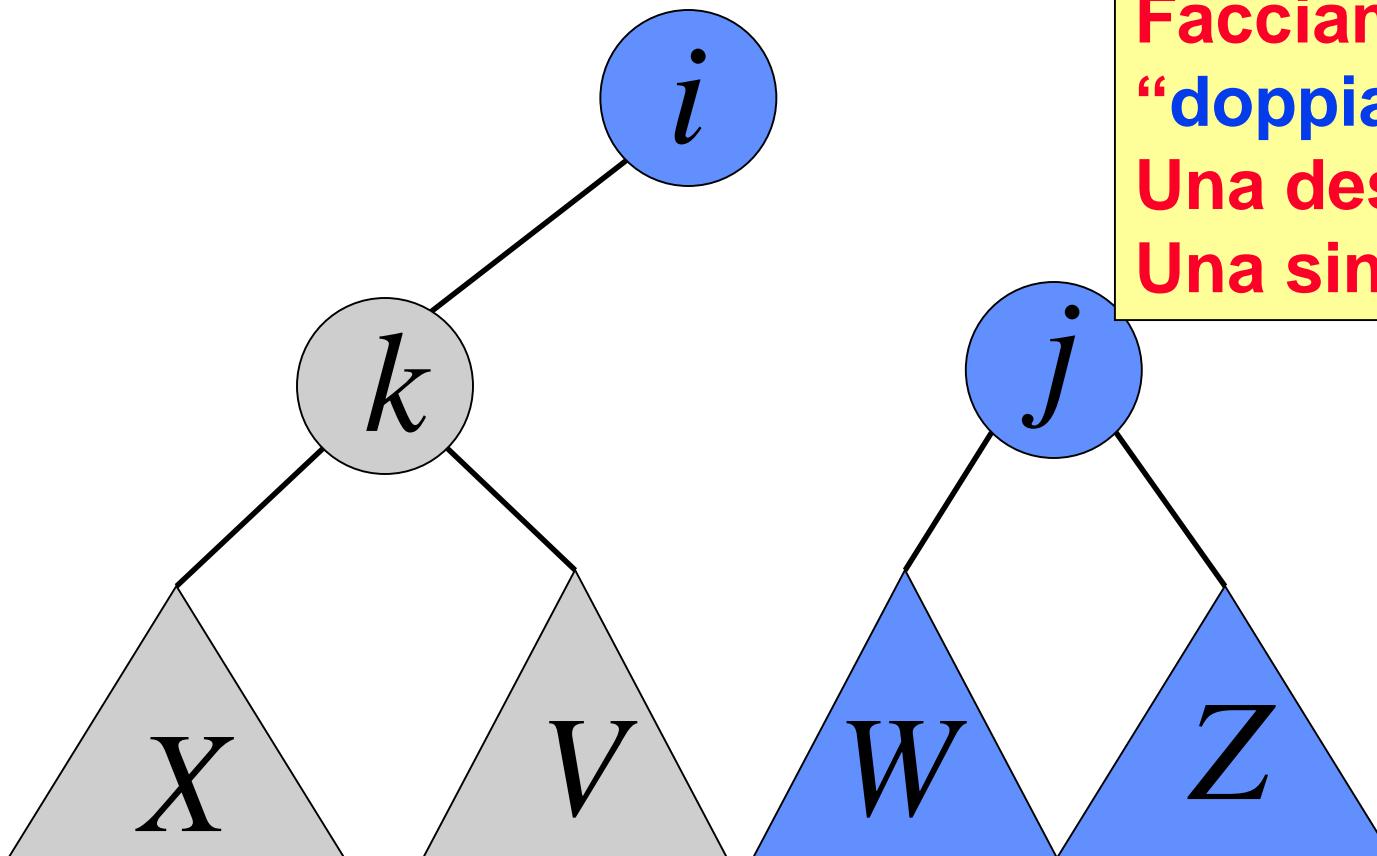
Facciamo ora una
“doppia rotazione”
Una destra
Una sinistra

Rotazione doppia in alberi AVL



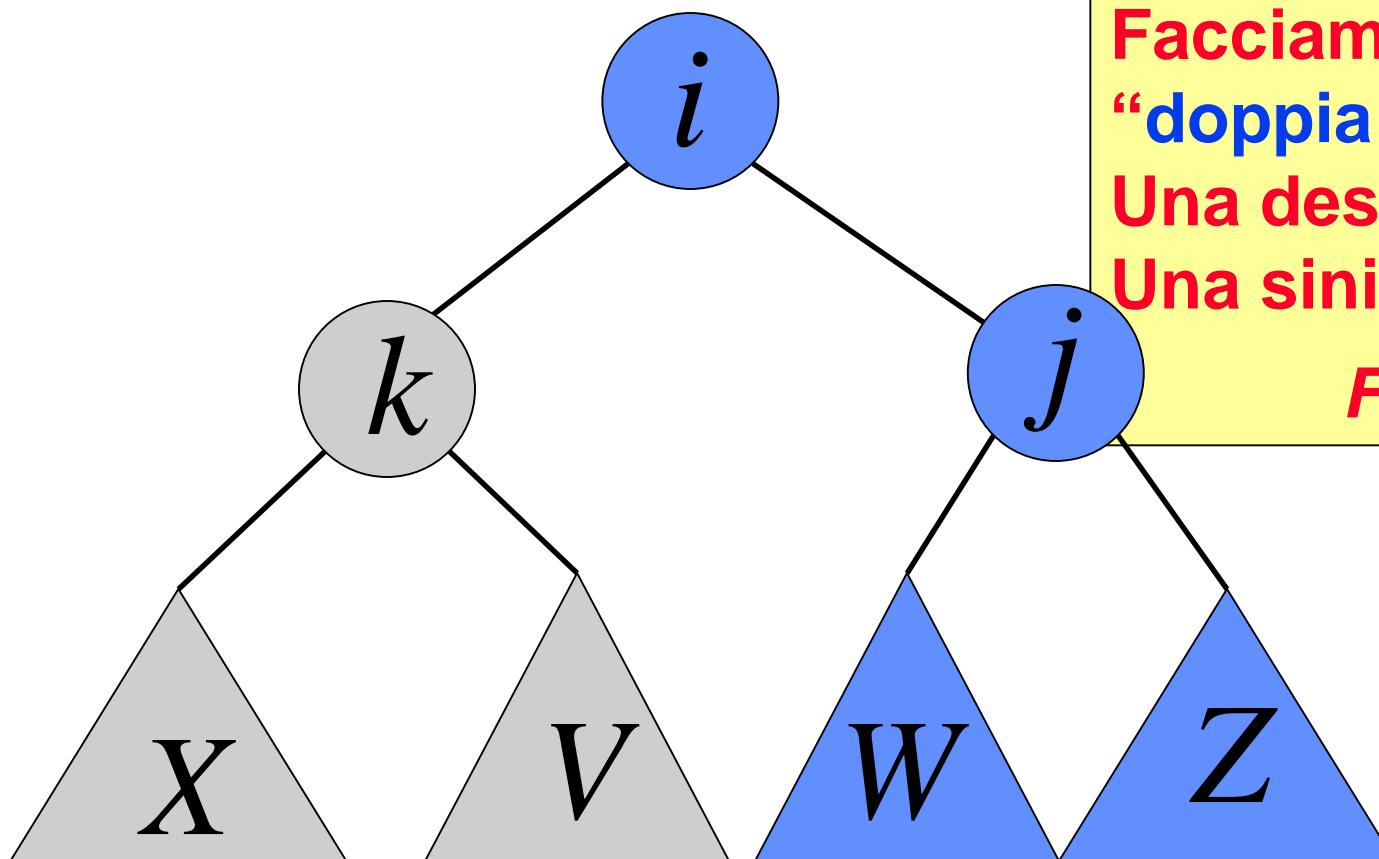
Facciamo ora una
“doppia rotazione”
Una destra
Una sinistra

Rotazione doppia in alberi AVL



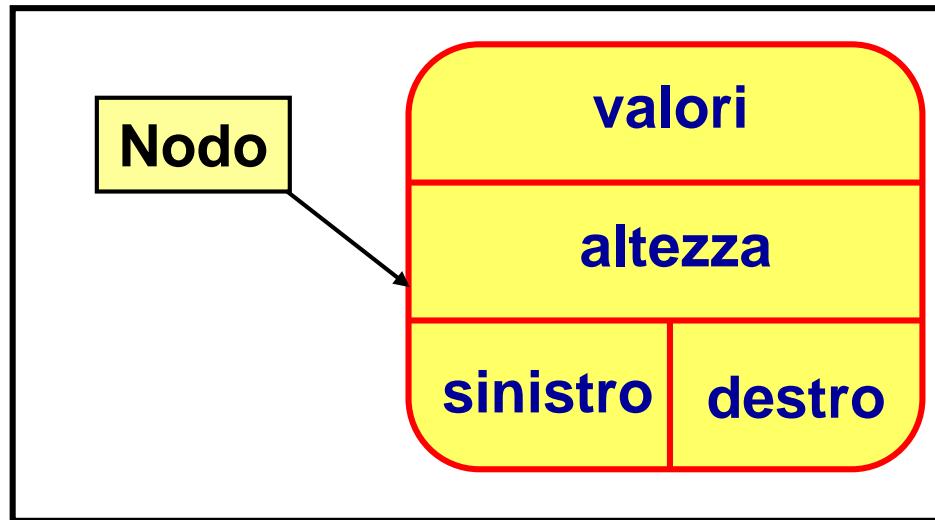
Facciamo ora una
“doppia rotazione”
Una destra
Una sinistra

Rotazione doppia in alberi AVL



Facciamo ora una
“doppia rotazione”
Una destra
Una sinistra
Fatto!!!

Rotazione doppia: algoritmo



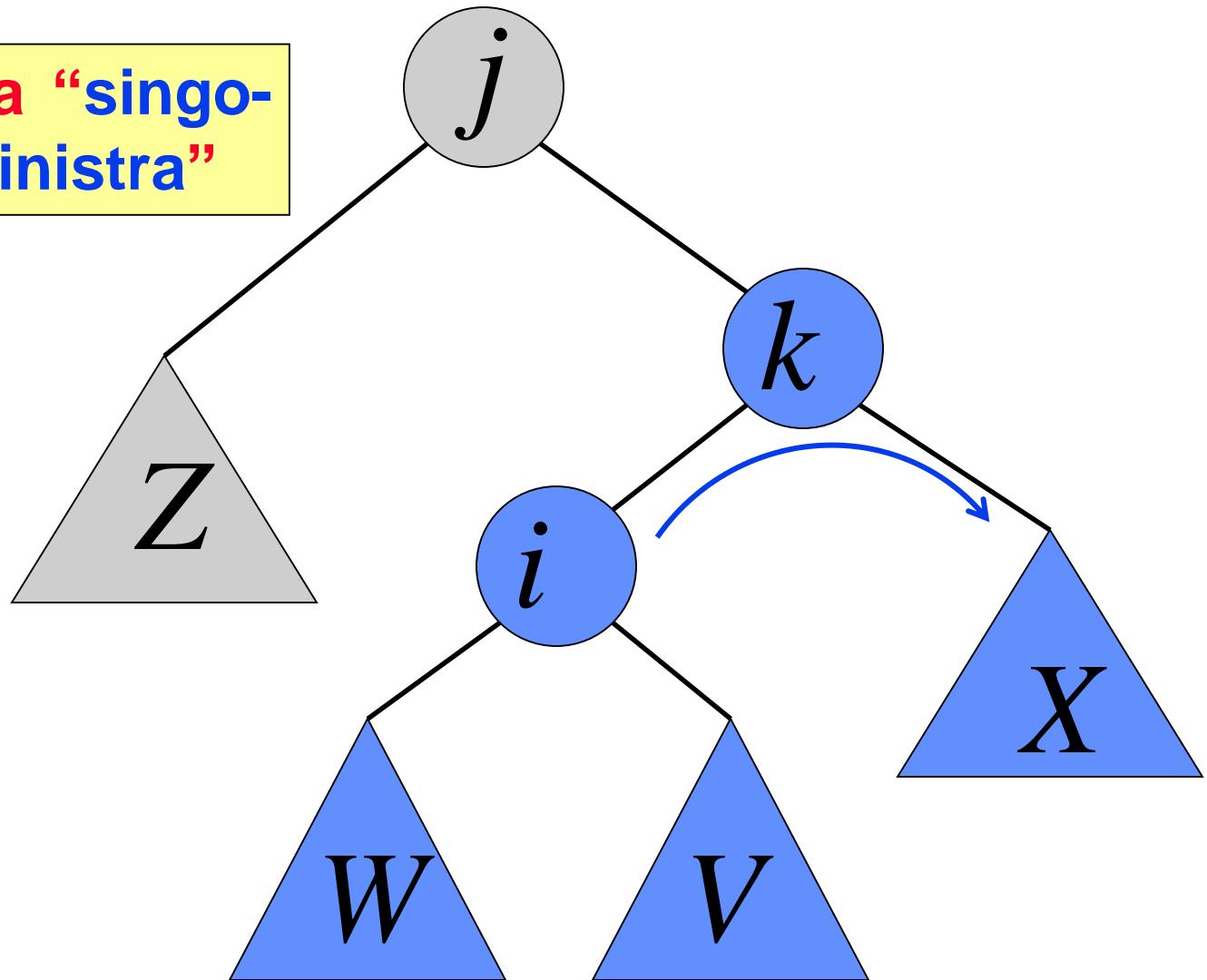
Rotazione-DS(T : albero-AVL)

$T\rightarrow sx = \text{Rotazione-SD}(T\rightarrow sx)$

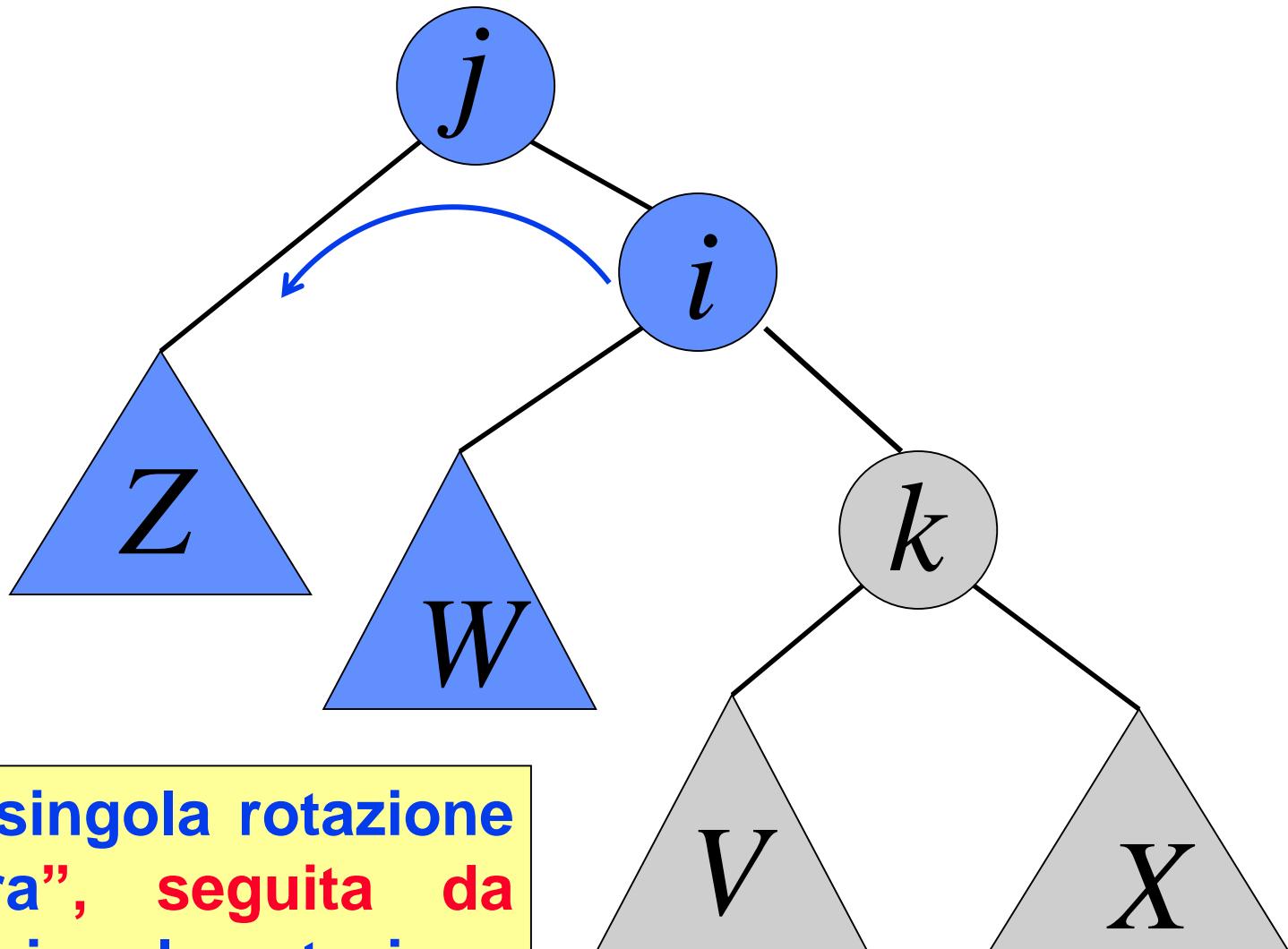
return Rotazione-SS(T)

Rotazione doppia in alberi AVL

Facciamo una “singola rotazione sinistra”

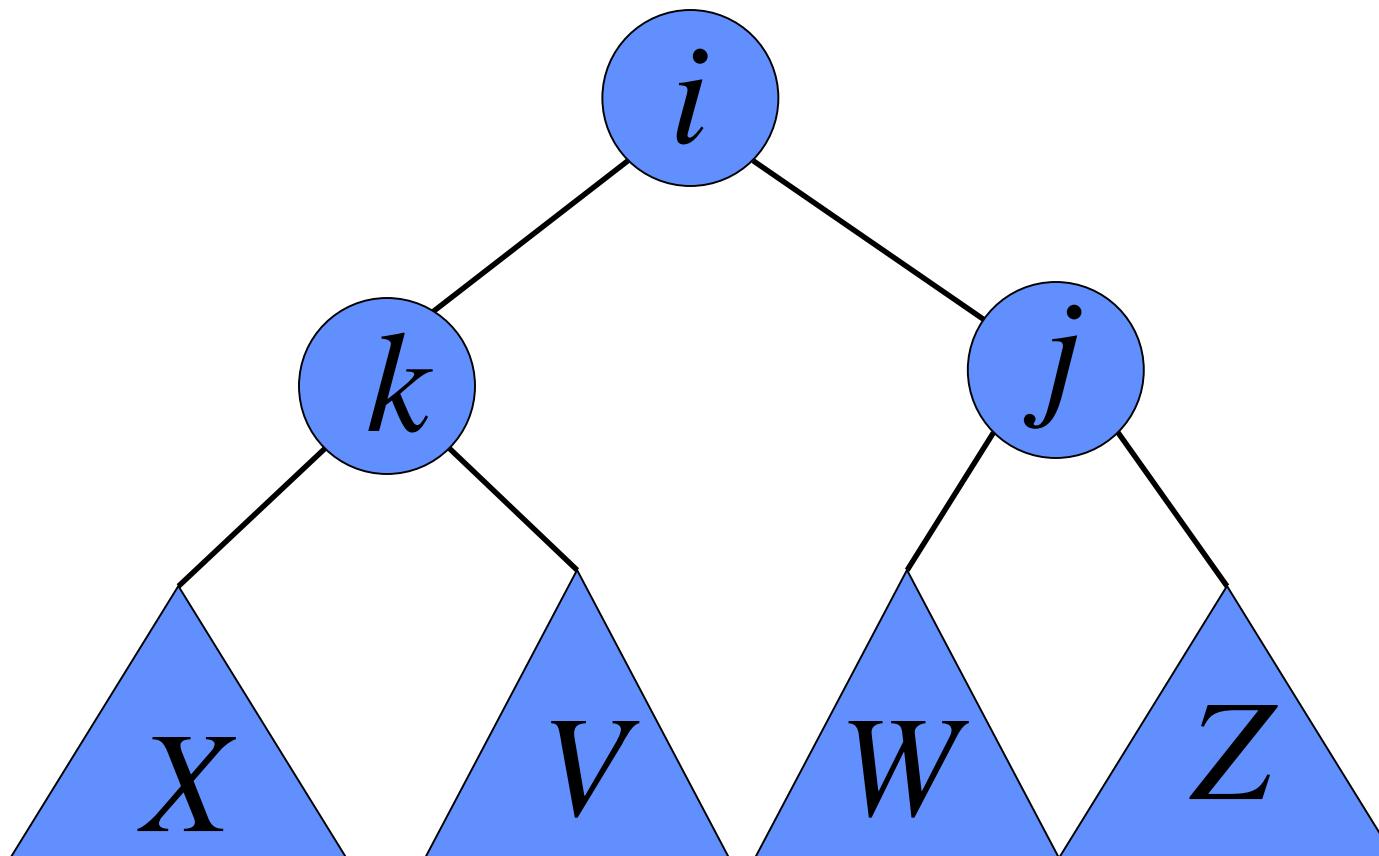


Rotazione doppia in alberi AVL



Una “singola rotazione sinistra”, seguita da una “singola rotazione destra”

Rotazione doppia in alberi AVL



Rotazione doppia destra

Inserimento in alberi AVL: algoritmo

Inserimento-in-AVL(x : chiave, T : albero-AVL)

IF $T = NIL$ **THEN** Alloca T

$T \rightarrow altezza = 0$

$T \rightarrow key = x$

ELSE

IF $x < T \rightarrow Key$ **THEN**

$T \rightarrow sx = \text{Inserimento-in-AVL}(x, T \rightarrow sx)$

$T = \text{Ribilancia-Sx}(T)$

ELSE

IF $x > T.Key$ **THEN**

$T \rightarrow dx = \text{Inserimento-in-AVL}(x, T \rightarrow dx)$

$T = \text{Ribilancia-Dx}(T)$

ELSE { x è già presente}

return T

Inserimento in alberi AVL: algoritmo

Ribilancia-Sx(T)

IF **Altezza($T->sx$) - Altezza($T->dx$) = 2** THEN

 IF *is-SS($T->sx$)* THEN

 T = *Rotazione-SS(T)*

 ELSE

 T = *Rotazione-DS(T)*

 ELSE

$T->altezza = 1 + \max(\text{Altezza}(T->sx), \text{Altezza}(T->dx))$

return T

is-SS(T : albero-AVL)

return **Altezza($T->sx$) > Altezza($T->dx$)**

Inserimento in alberi AVL: algoritmo

- **Un *inserimento a quante operazioni di bilanciamento può dar luogo?***

Inserimento in alberi AVL: algoritmo

- Un *inserimento a quante operazioni di bilanciamento può dar luogo?*

*La risposta è: al più una
Perché?*

Cancellazione-AVL(T:albero-AVL, key:chiave)

IF T != NIL **THEN**

IF T.Key > key **THEN**

T->sx = Cancellazione-AVL(T->sx, key)

T=Bilancia_DX(T)

ELSE IF T->Key < key **THEN**

/* cancellazione simmetrica a destra */

ELSE IF T->sx = NIL **OR** T->dx = NIL **THEN**

T = elimina-nodo(T)

ELSE

tmp = Stacca-Min-AVL(T->dx, T)

scambia T->Key con tmp->Key

T = Bilancia_SX(T)

dealloca(tmp)

return T

Stacca-Min-AVL(T:albero-AVL , P:albero-AVL)

IF T != NIL THEN

IF T->sx != NIL THEN

ret = Stacca-Min-AVL(T->sx, T)

newroot = Bilancia_DX(T)

ELSE

ret = T

newroot = T->dx

IF T = P->sx THEN P->sx = newroot

ELSE P->dx = newroot

return ret

Bilancia_DX(T)

IF Altezza(T->dx) - Altezza(T->sx) = 2 THEN

IF is-SD(T->dx) THEN T = Rotazione-SD(T)

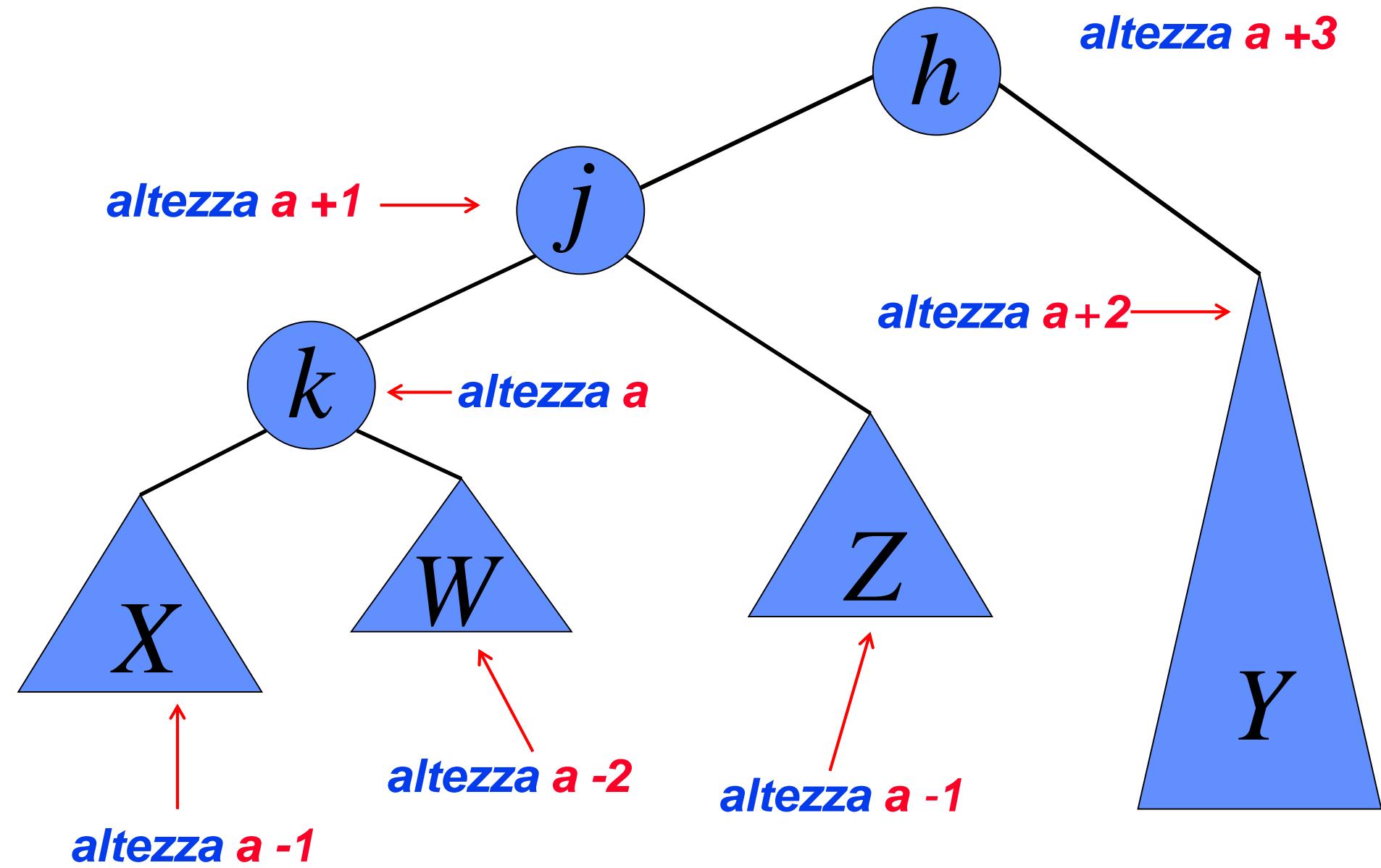
ELSE T = Rotazione-DD(T)

ELSE

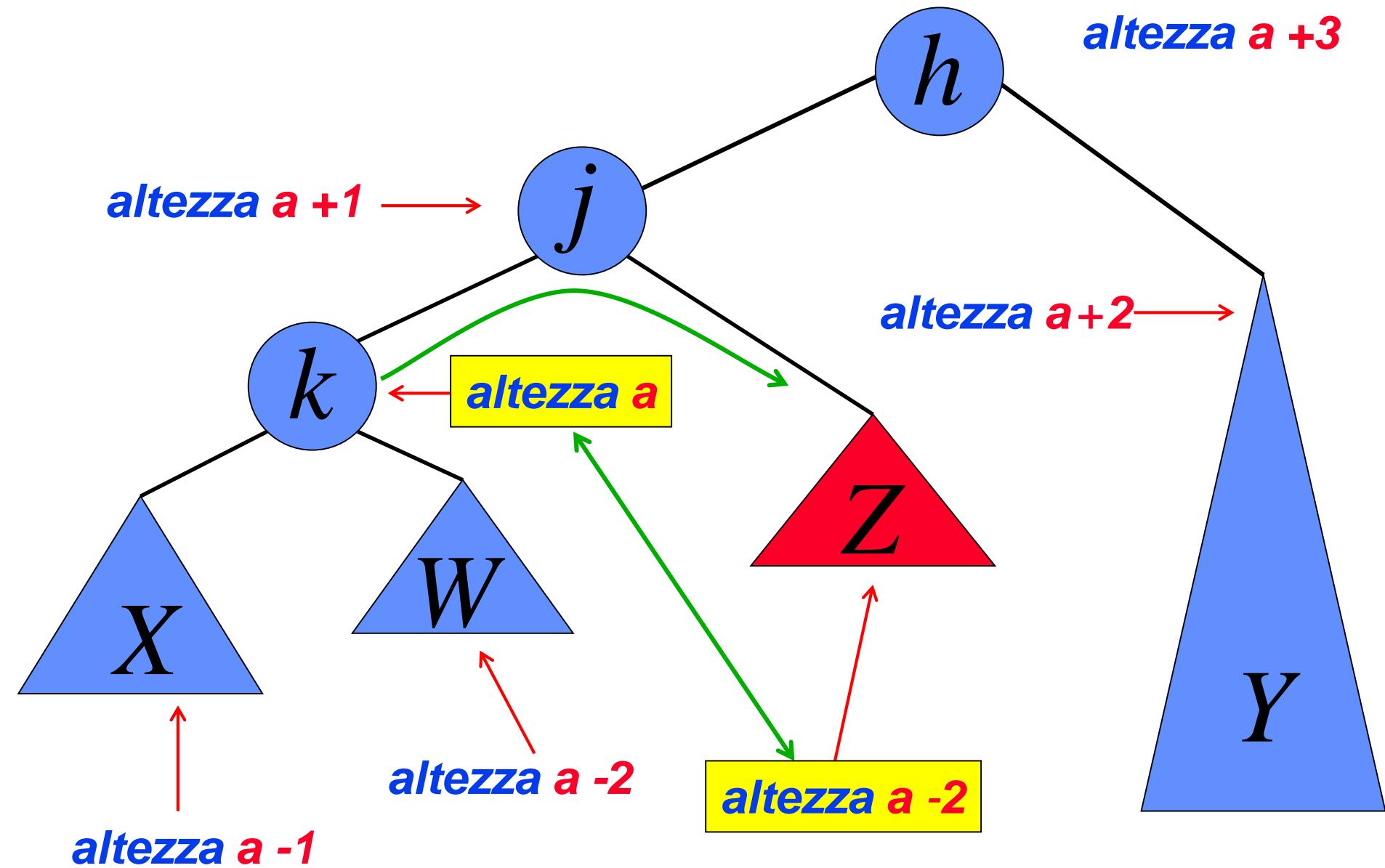
T->altezza = 1 + max(Altezza(T->sx), Altezza(T->dx))

return T

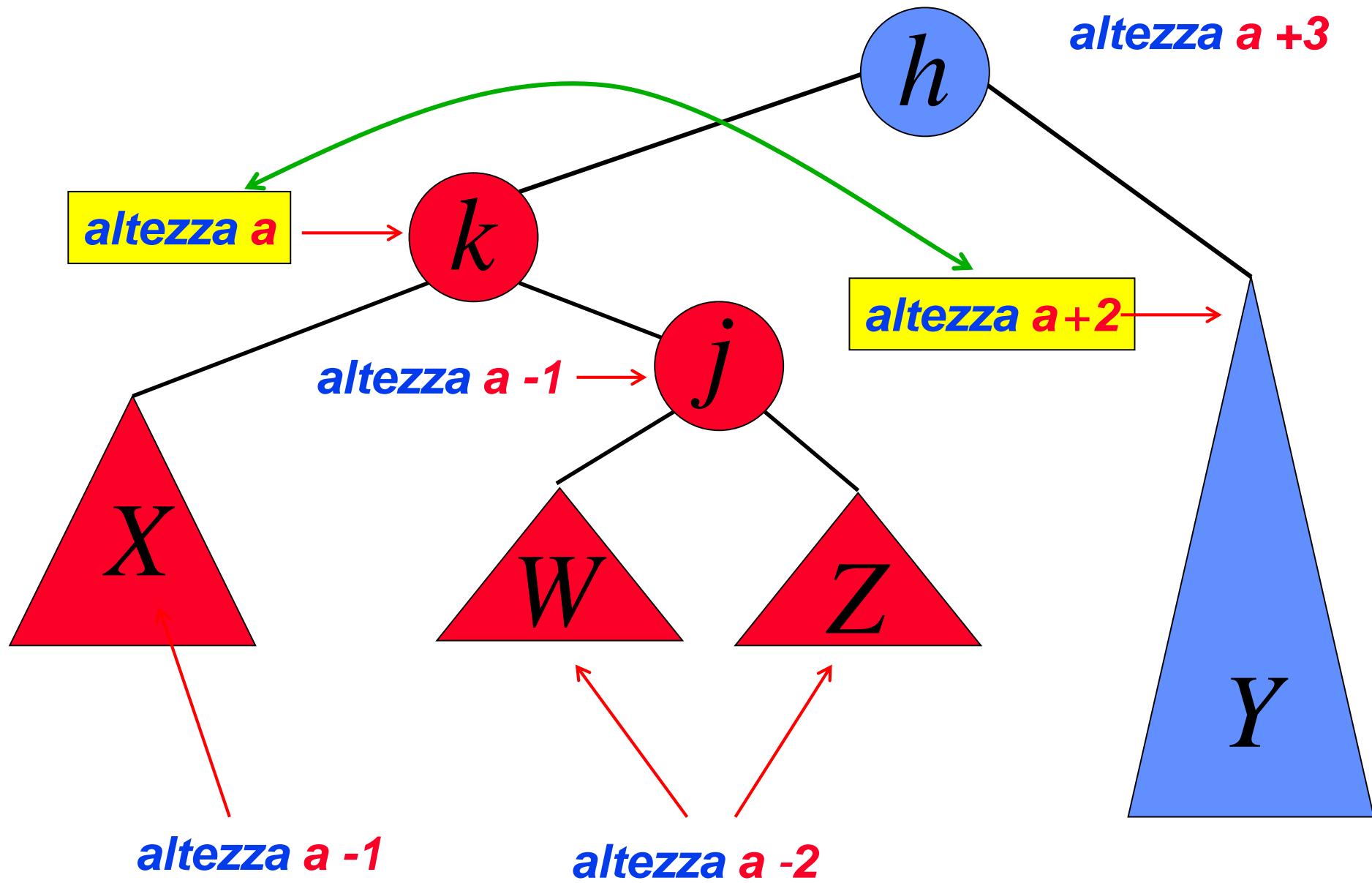
Cancellazione in alberi AVL



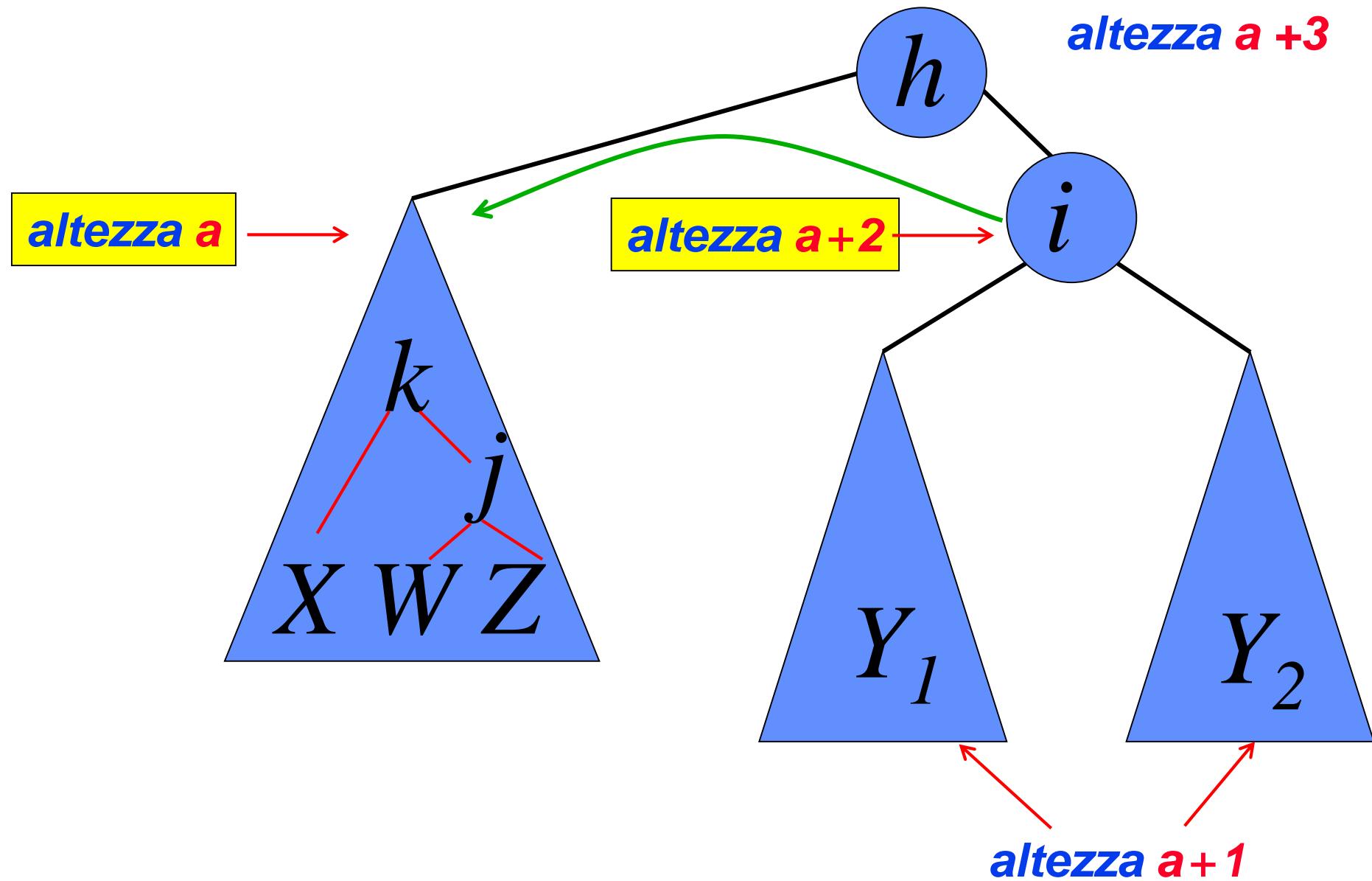
Cancellazione in alberi AVL



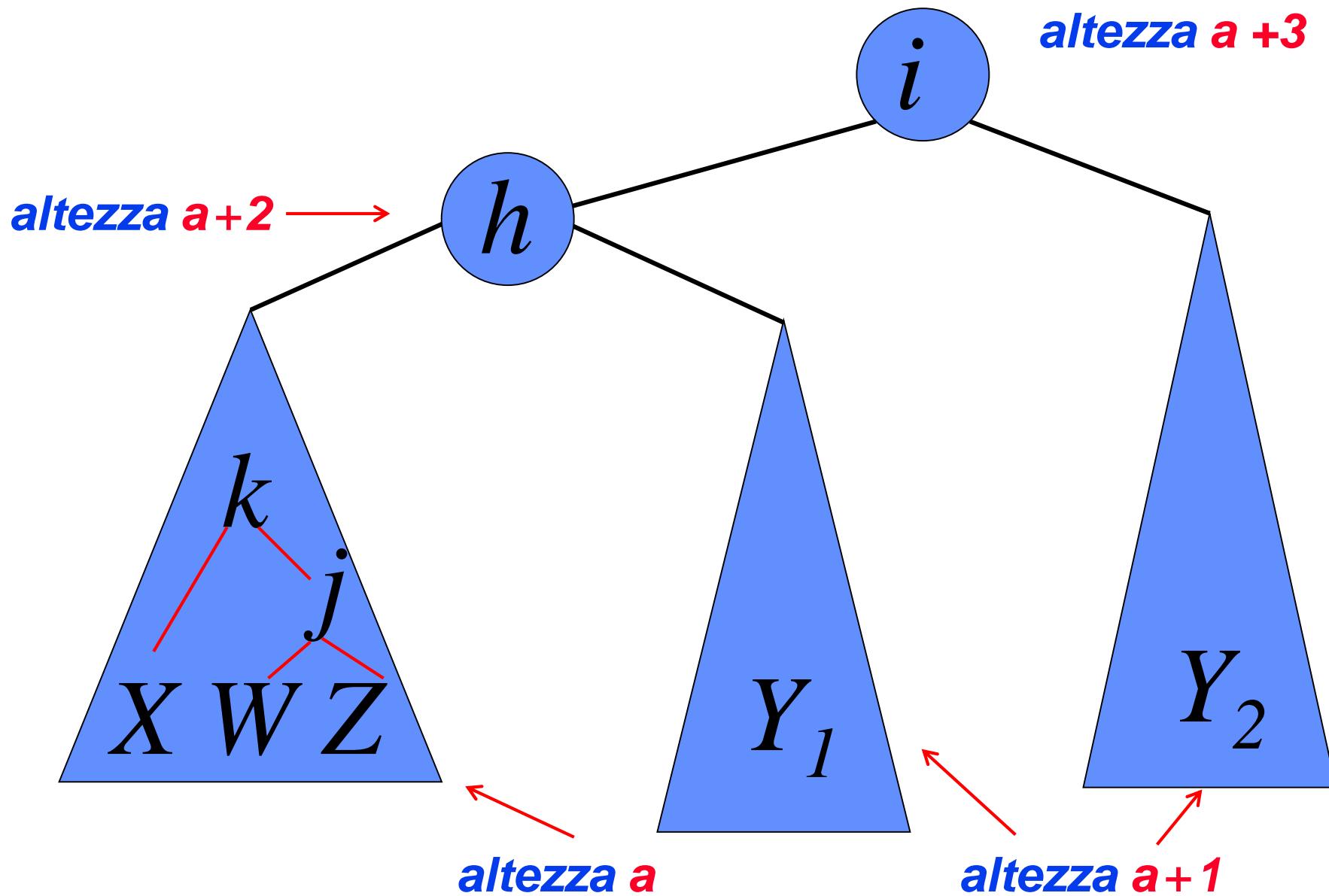
Cancellazione in alberi AVL



Cancellazione in alberi AVL



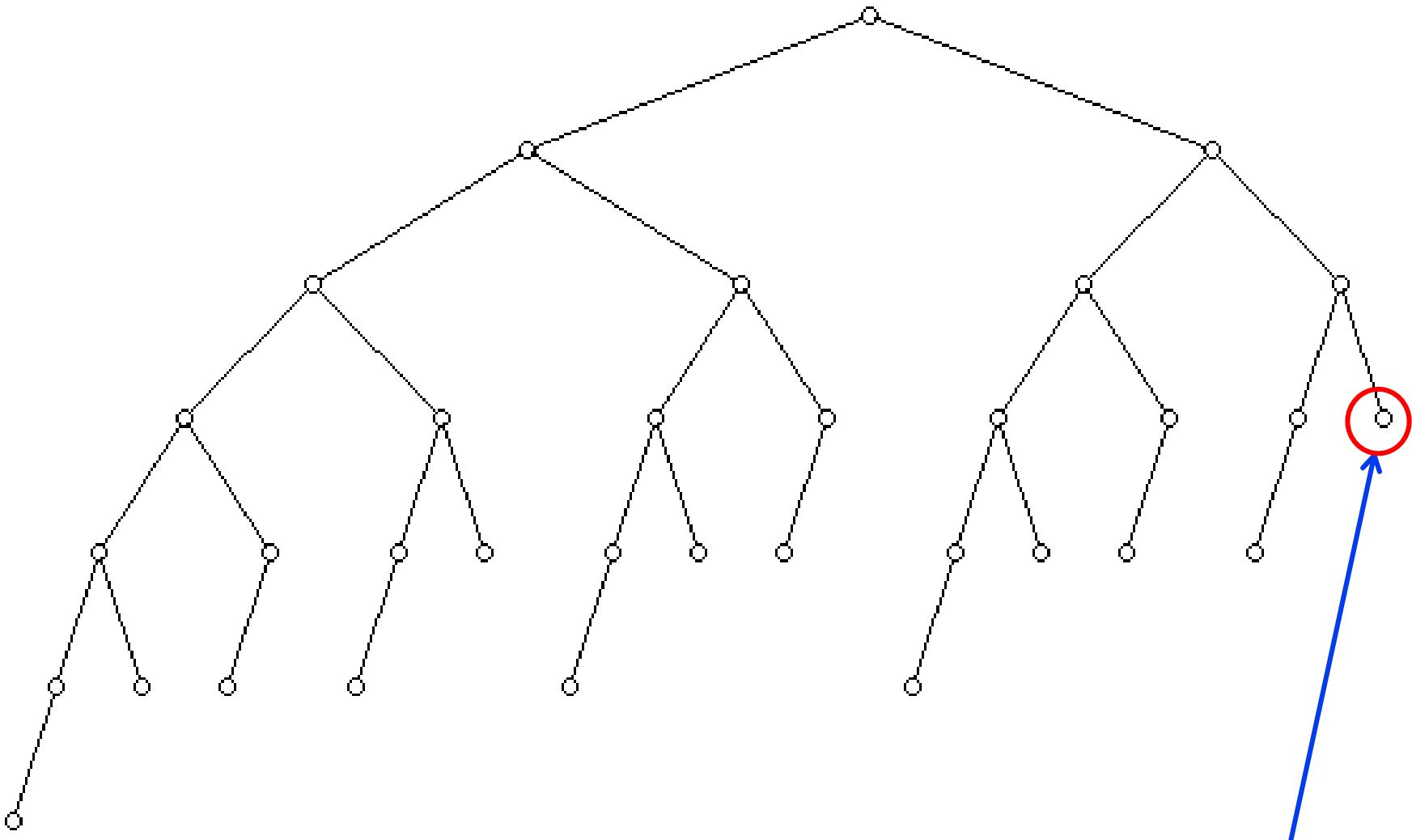
Cancellazione in alberi AVL



Cancellazione in alberi AVL

- Dall'esempio appena visto si conclude che *una cancellazione può dar luogo a più operazioni di bilanciamento*
- *L'algoritmo di cancellazione* è più complesso perché deve tener conto di questa possibilità
- Può *anche* verificarsi la necessità di *una operazione di bilanciamento per ogni livello (in quale caso?)*
- Il numero di operazioni bilanciamento è nel caso peggiore *O(h)* (*h altezza dell'albero*)

Cancellazione in alberi AVL



Necessità di una operazione di bilanciamento per ogni
livello se si cancella l'elemento più a destra!

Algoritmi e Strutture Dati

Alberi Bilanciati: Alberi Red-Black

Alberi bilanciati di ricerca

- Gli *alberi binari di ricerca* sono semplici da gestire (inserimenti e cancellazioni facili da implementare) ma hanno prestazioni poco prevedibili e potenzialmente basse
- Gli *alberi perfettamente bilanciati* hanno prestazioni ottimali ($\log n$ garantito) ma inserimenti e cancellazioni complesse (ribilanciamenti)
- Alberi AVL (*Adelson-Velskii e Landis*): *alberi quasi bilanciati*. Buone prestazioni e gestione relativamente semplice.

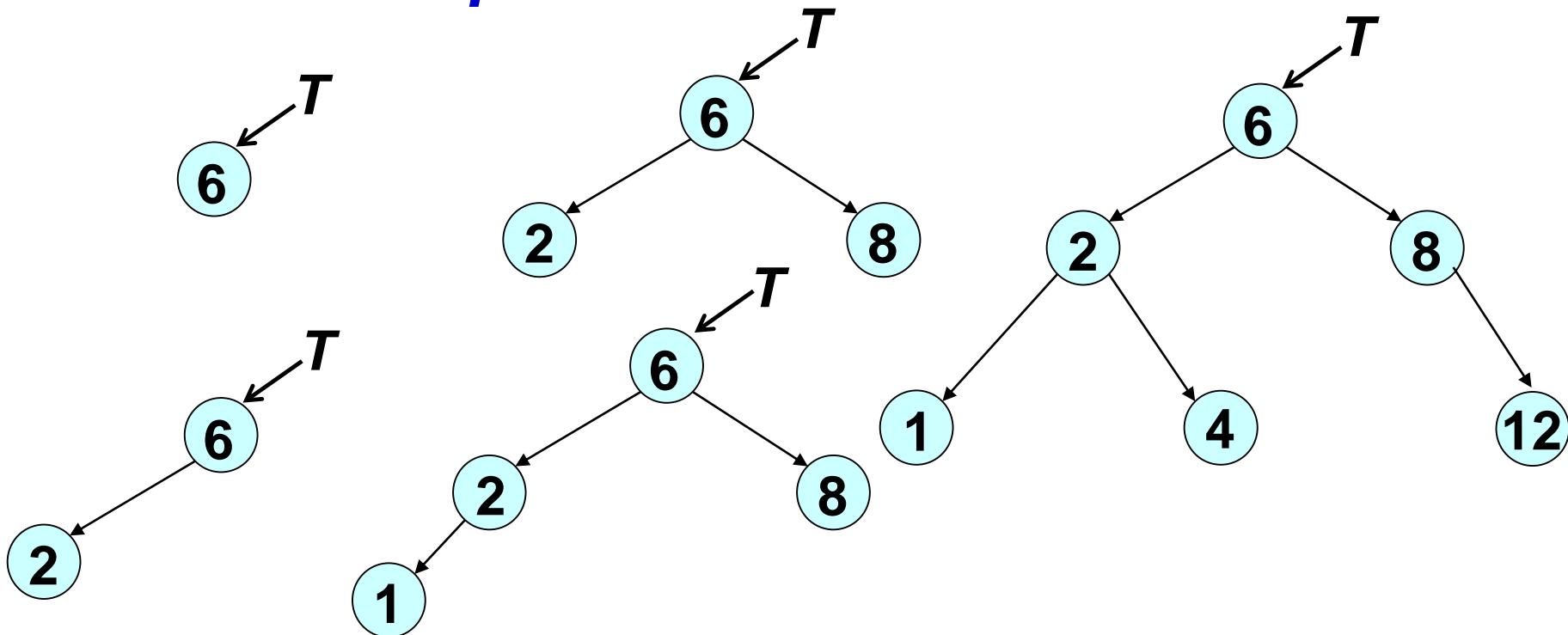
Alberi AVL: definizione

Definizione: Un albero binario di ricerca è un Albero AVL se per ogni nodo **x**:

- l'*altezza del sottoalbero sinistro di x* e quella del *sottoalbero destro di x* differiscono al più di uno, e
- *entrambi i sottoalberi* sinistro e destro di **x** sono **alberi AVL**

Alberi perfettamente bilanciati

Definizione: Un albero binario si dice Perfettamente Bilanciato se, per ogni nodo *i*, il numero dei nodi nel suo **sottoalbero sinistro** e il numero dei nodi del suo **sottoalbero destro** **differiscono al più di 1**



Alberi perfettamente bilanciati

Definizione: Un albero binario si dice **Perfettamente Bilanciato** se, per ogni nodo *i*, il **numero dei nodi** nel suo **sottoalbero sinistro** e il **numero dei nodi** del suo **sottoalbero destro** **differiscono al più** di 1

L'**altezza** di un **albero perfettamente bilanciato (APB)** con **n nodi** è **$h = \log_2 n$**

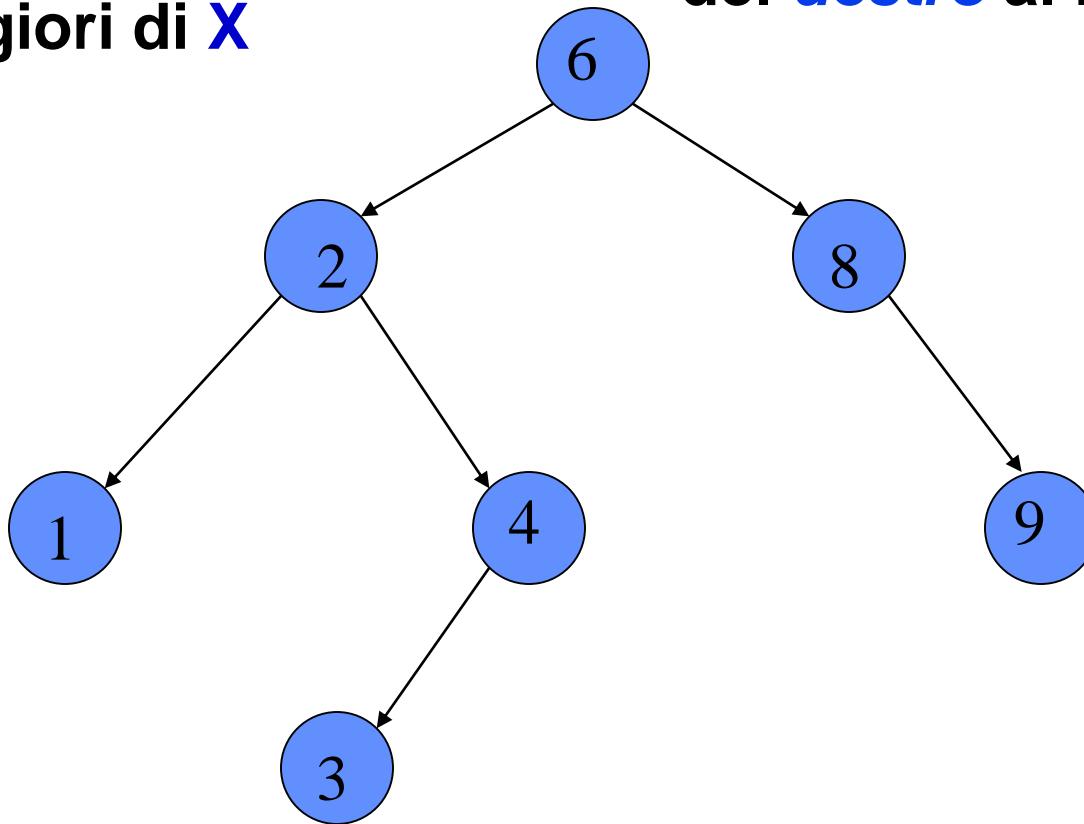
Alberi AVL: alberi binari bilanciati

Proprietà degli ABR

Per ogni nodo **X**, i nodi del **sottoalbero sinistro** sono minori del nodo **X**, e i nodi del **sottoalbero destro** sono maggiori di **X**

Properietà AVL

ABR dove per ogni nodo **X**, l'**altezza del sottoalbero sinistro differisce** da quella del **destro** al massimo di **1**.

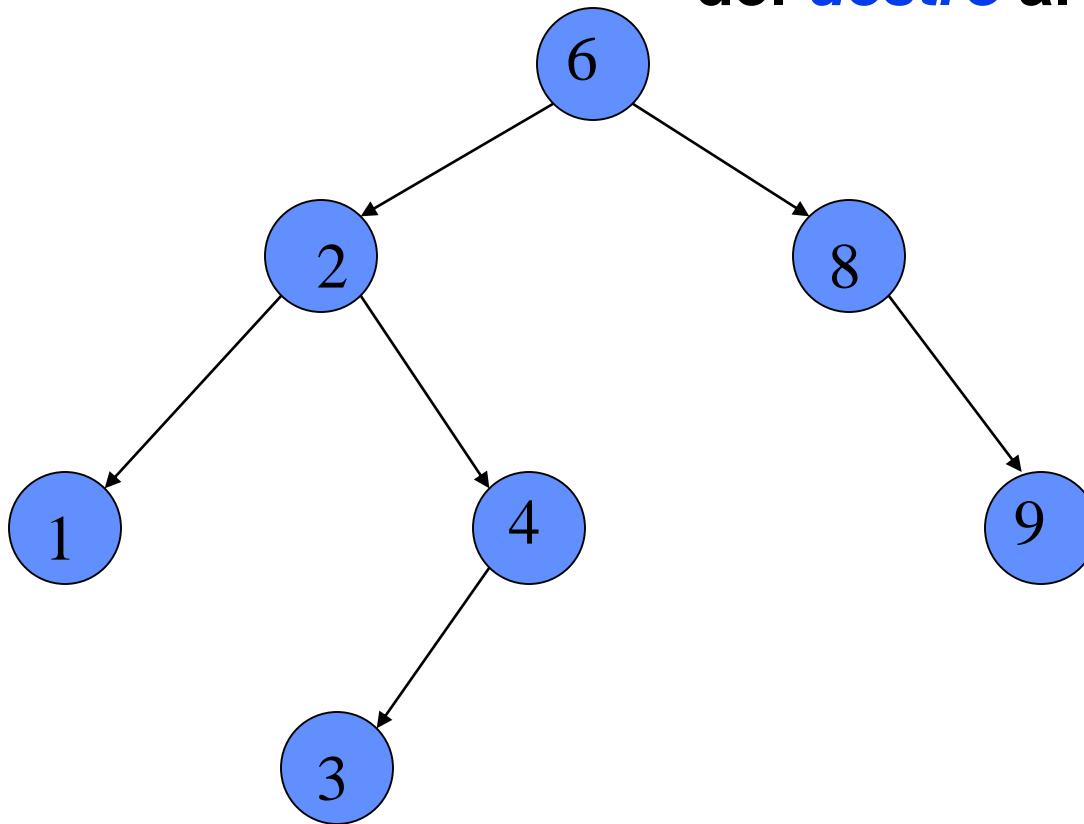


Alberi AVL: alberi binari bilanciati

Altezza(X) =
max(Altezza(sinistro(X)),
Altezza(destro(X))) + 1

Properietà AVL

ABR dove per ogni nodo X ,
l'**altezza del sottoalbero
sinistro differisce** da quella
del **destro** al massimo di 1.

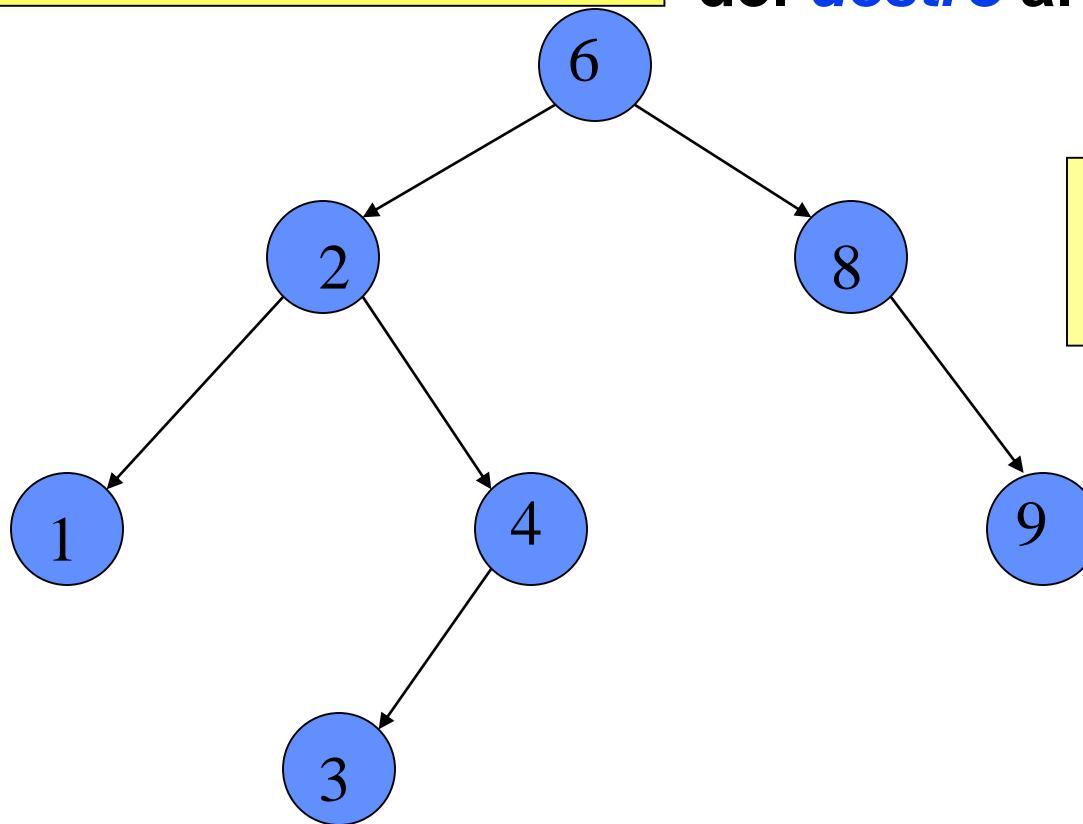


Alberi AVL: alberi binari bilanciati

Altezza(X) =
max(Altezza(sinistro(X)),
Altezza(destro(X))) + 1
Altezza(\emptyset) = -1

Properietà AVL

ABR dove per ogni nodo X ,
l'**altezza del sottoalbero
sinistro differisce** da quella
del **destro** al massimo di 1.



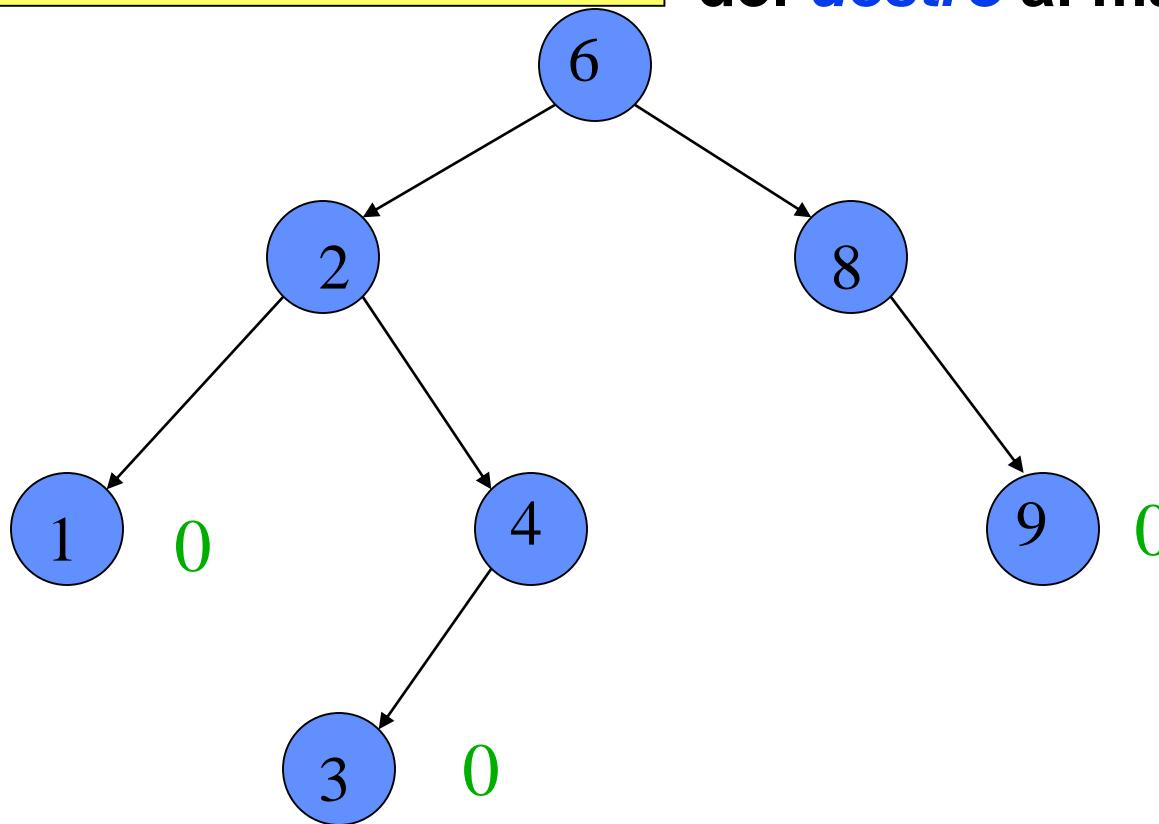
Questo è un
albero AVL?

Alberi AVL: alberi binari bilanciati

Altezza(X) =
max(Altezza(sinistro(X)),
Altezza(destro(X))) + 1
Altezza(\emptyset) = -1

Properietà AVL

ABR dove per ogni nodo X ,
l'**altezza del sottoalbero sinistro differisce** da quella
del **destro** al massimo di 1.

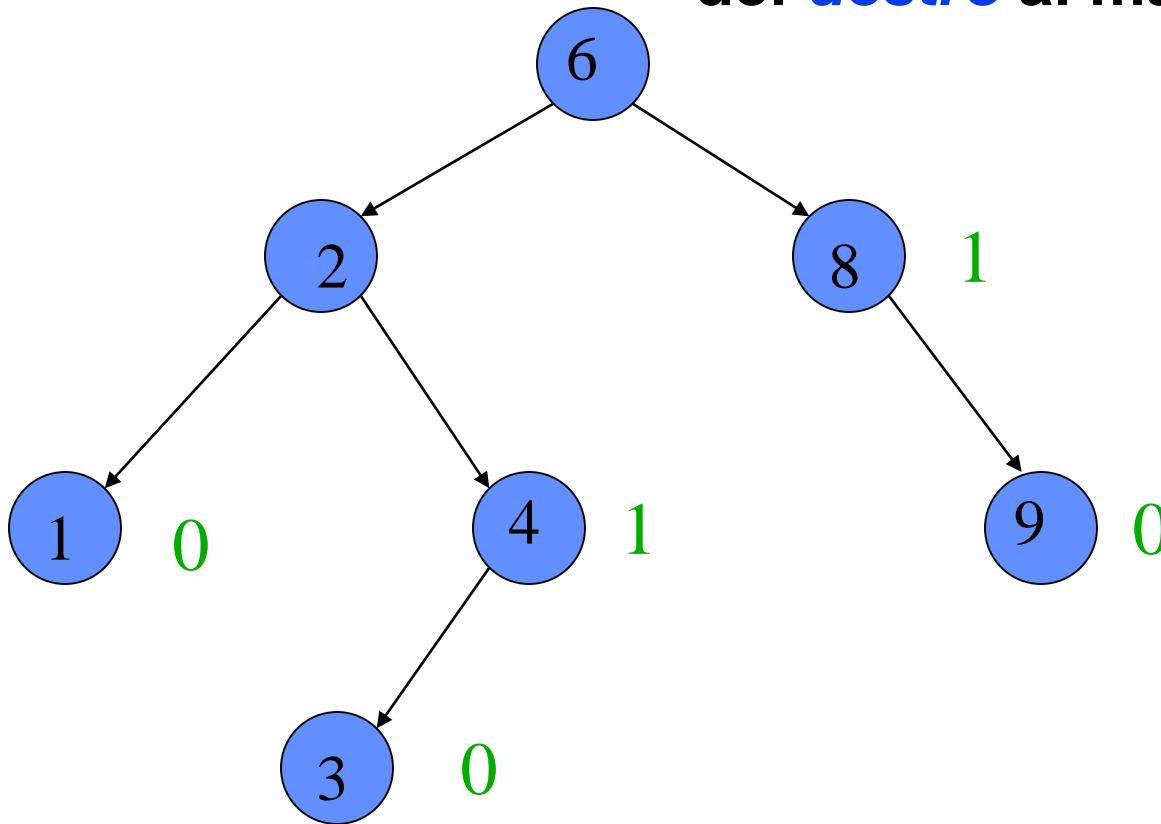


Alberi AVL: alberi binari bilanciati

Altezza(X) =
max(Altezza(sinistro(X)),
Altezza(destro(X))) + 1

Properietà AVL

ABR dove per ogni nodo X ,
l'**altezza del sottoalbero
sinistro differisce** da quella
del **destro** al massimo di 1.

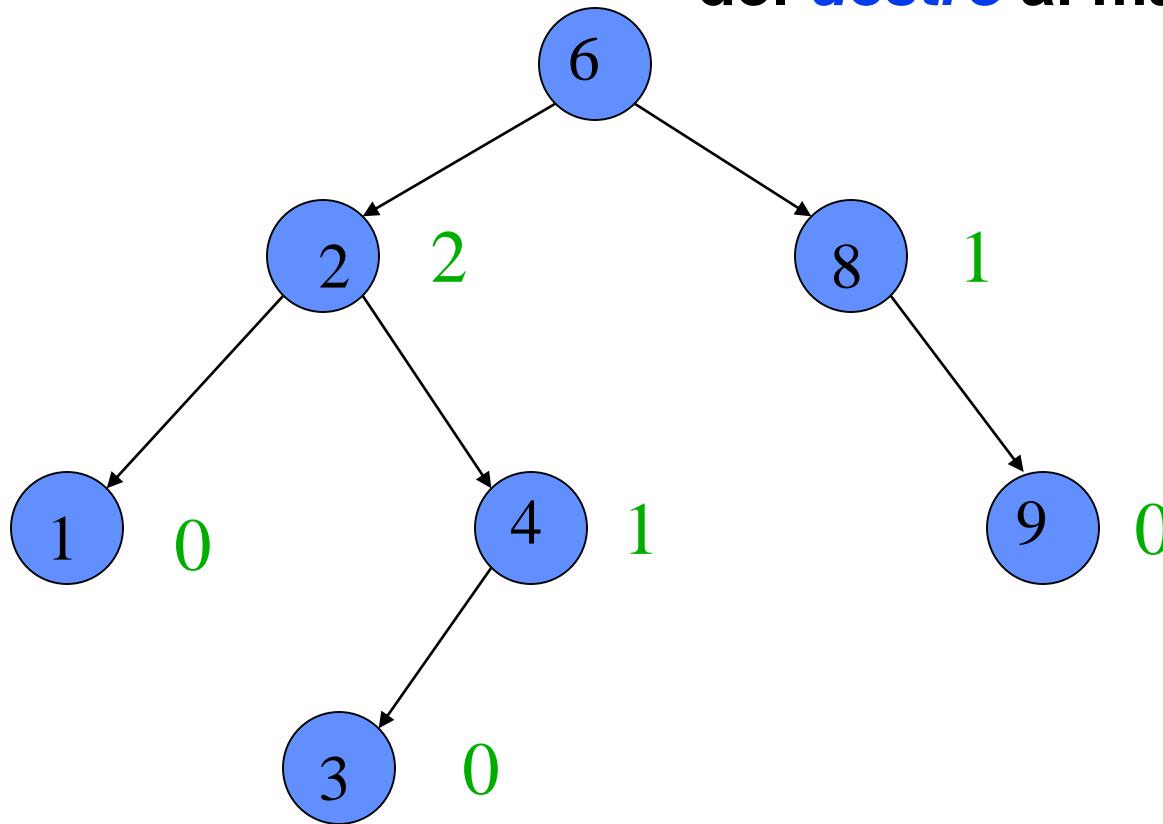


Alberi AVL: alberi binari bilanciati

Altezza(X) =
max(Altezza(sinistro(X)),
Altezza(destro(X))) + 1

Properietà AVL

ABR dove per ogni nodo X ,
l'**altezza del sottoalbero
sinistro differisce** da quella
del **destro** al massimo di 1.

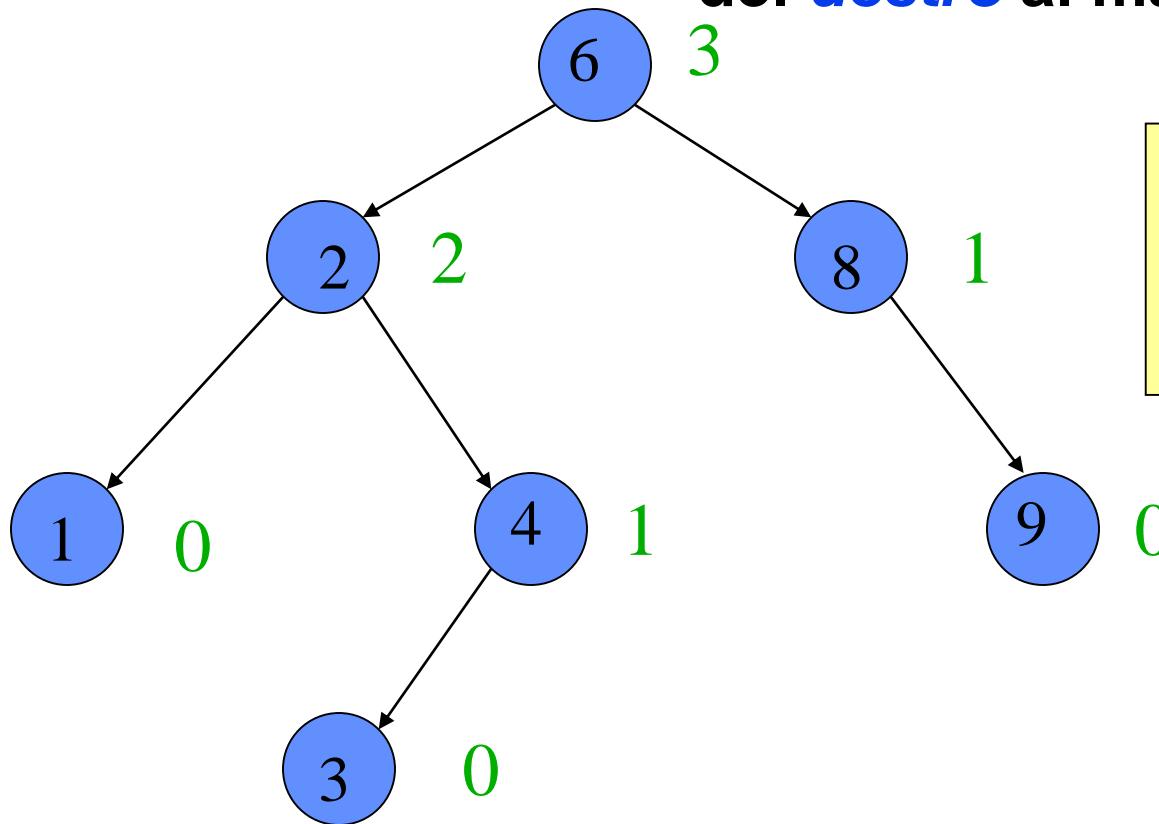


Alberi AVL: alberi binari bilanciati

Altezza(X) =
max(Altezza(sinistro(X)),
Altezza(destro(X))) + 1

Properietà AVL

ABR dove per ogni nodo X ,
l'**altezza del sottosalbero
sinistro differisce** da quella
del **destro** al massimo di 1.



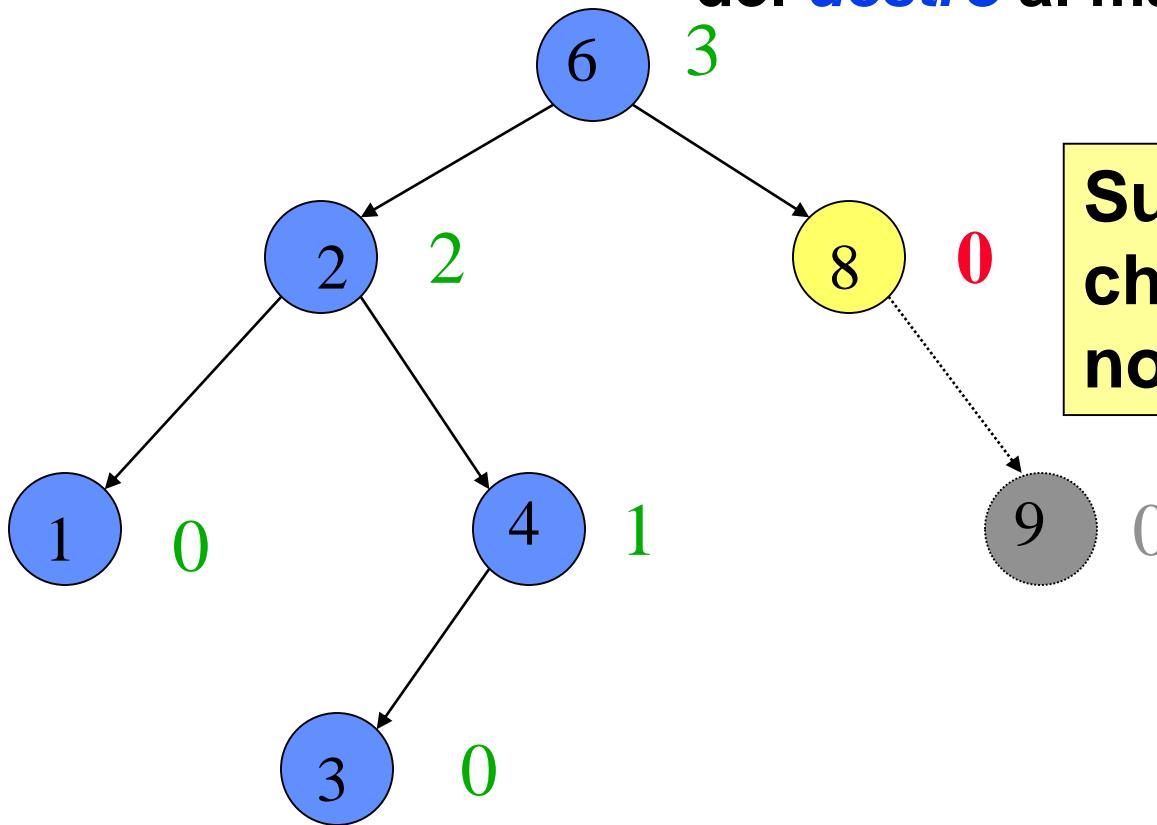
Sì! Questo
è un albero
AVL.

Alberi AVL: alberi binari bilanciati

Altezza(X) =
max(Altezza(sinistro(X)),
Altezza(destro(X))) + 1

Properietà AVL

ABR dove per ogni nodo X ,
l'**altezza del sottoalbero
sinistro differisce** da quella
del **destro** al massimo di 1.



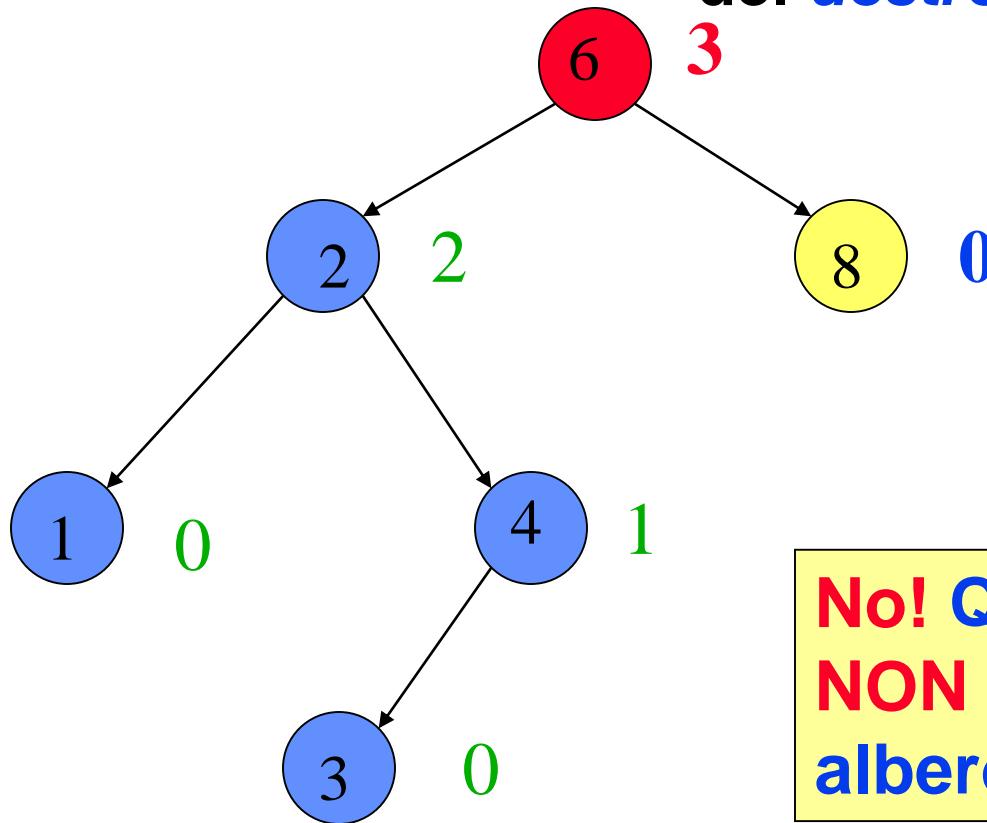
Supponiamo
che il nodo 9
non ci sia.

Alberi AVL: alberi binari bilanciati

**Altezza(X) =
max(Altezza(sinistro(X)),
Altezza(destro(X))) + 1**

Properietà AVL

ABR dove per ogni nodo X ,
l'**altezza del sottoalbero
sinistro differisce** da quella
del **destro** al massimo di 1.



La **Proprietà
AVL non è
soddisfatta**
dal **nodo 6**.

No! Questo
NON è un
albero AVL.

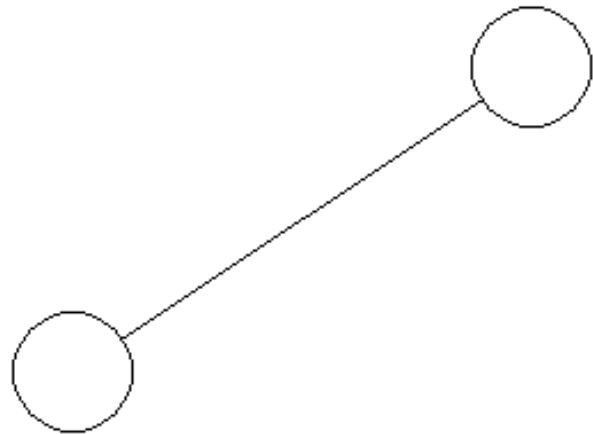
Alberi AVL: definizione

Definizione: Un albero binario di ricerca è un Albero AVL se per ogni nodo x :

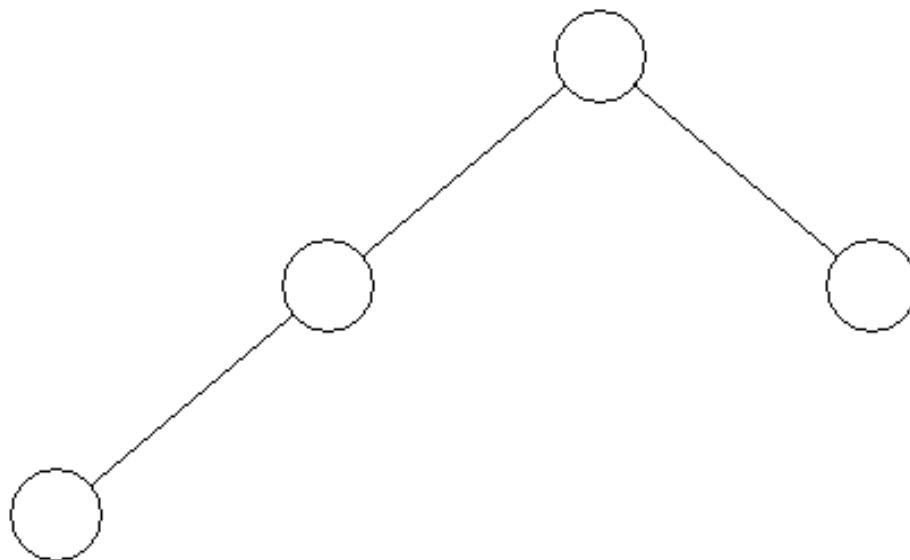
- l'altezza del *sottoalbero sinistro di x* e quella del *sottoalbero destro di x* differiscono al più di uno, e
- entrambi i *sottoalberi* sinistro e destro di x sono *alberi AVL*

**Esempi di alberi AVL minimi di diverse altezze
(cioè con il numero minimo possibile di nodi)**

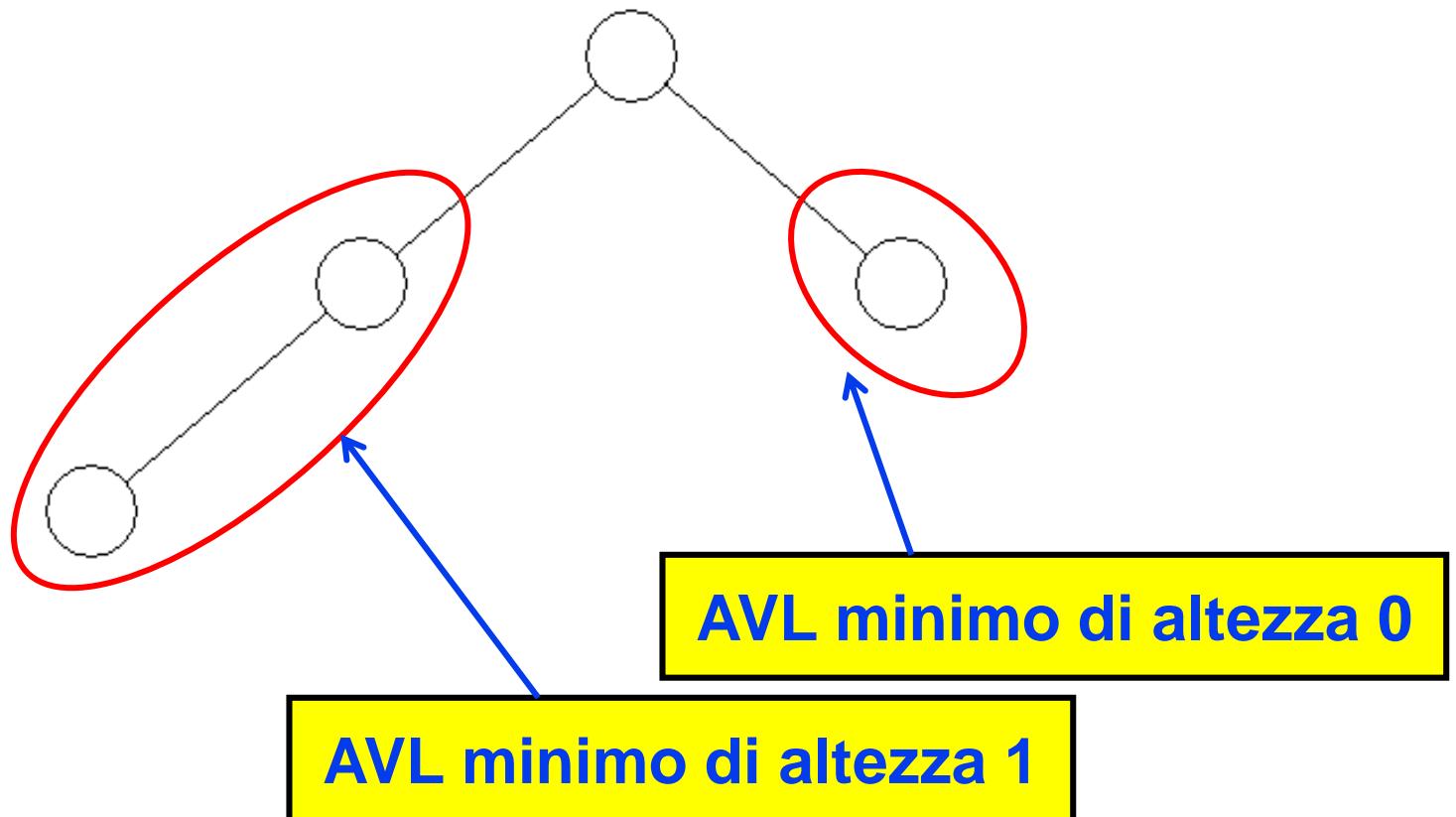
Albero AVL minimo di altezza 1



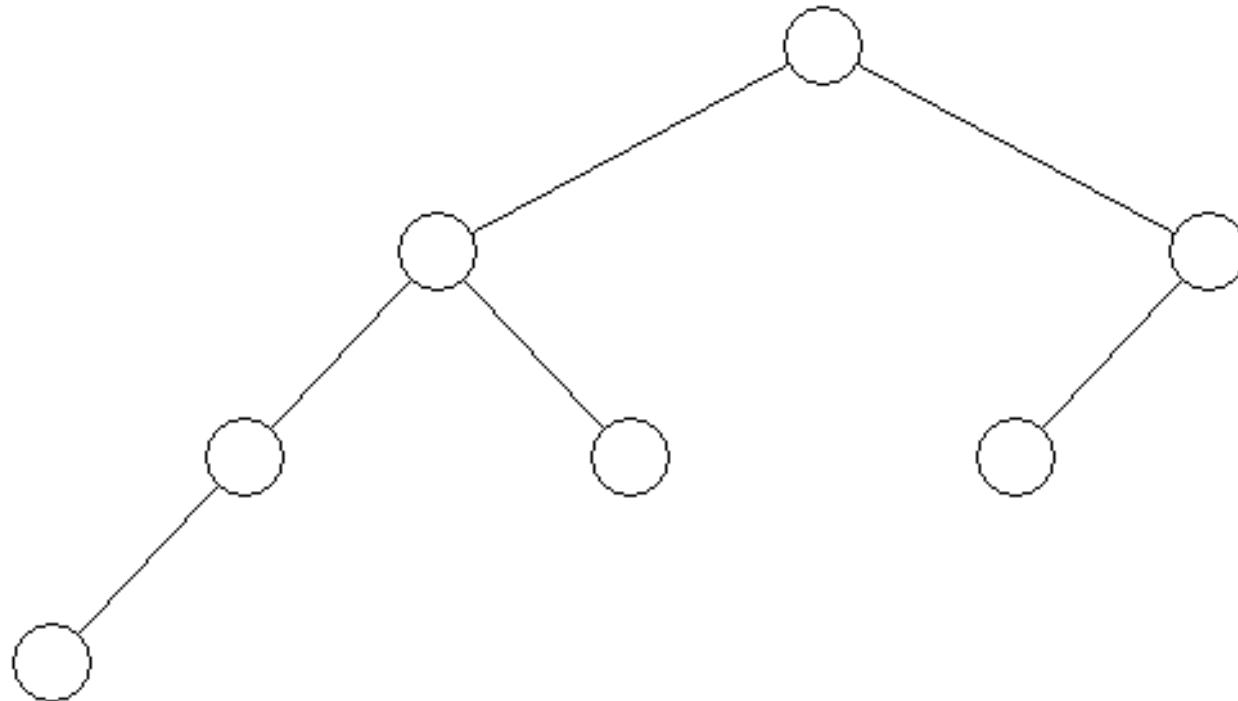
Albero AVL minimo di altezza 2



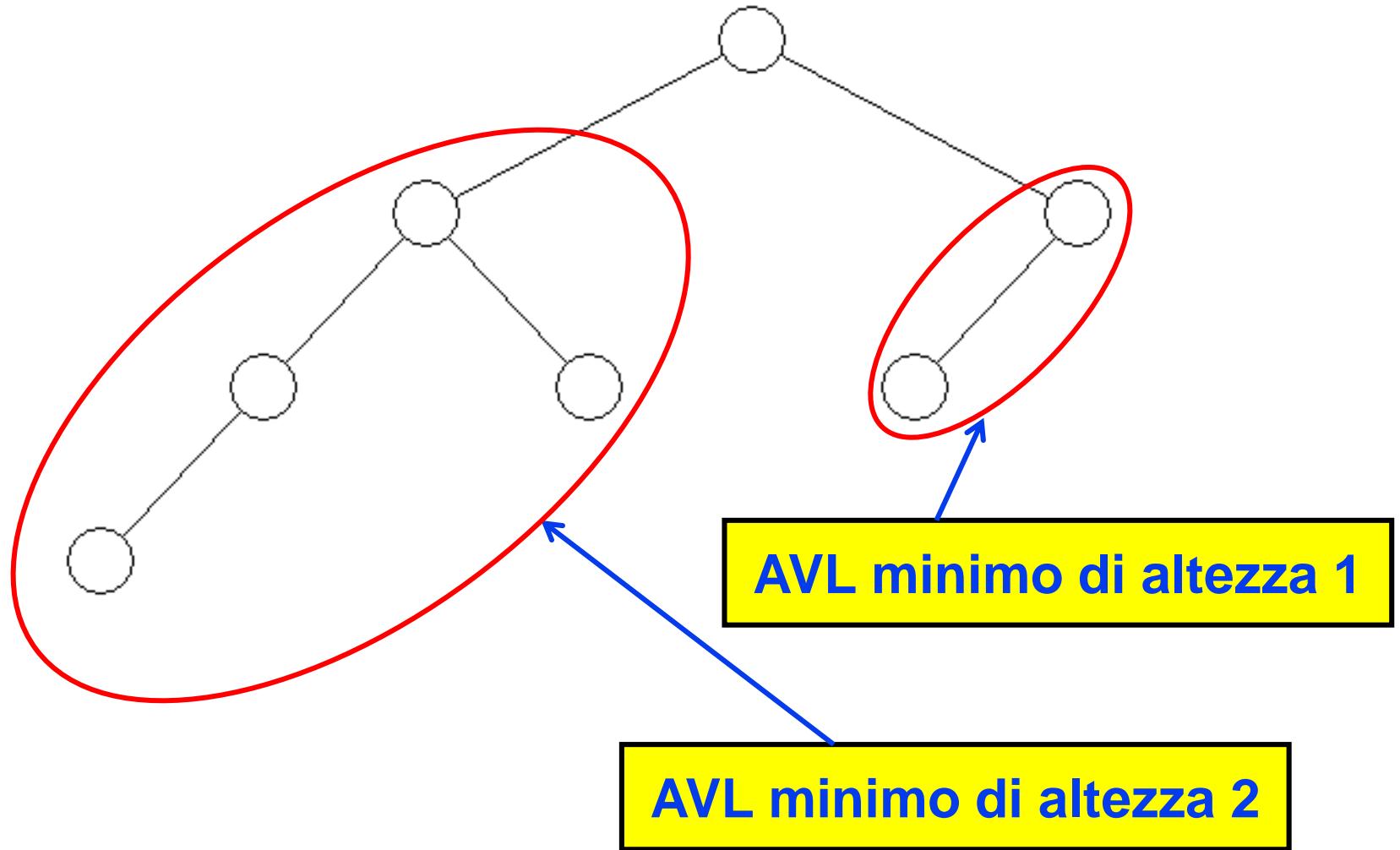
Albero AVL minimo di altezza 2



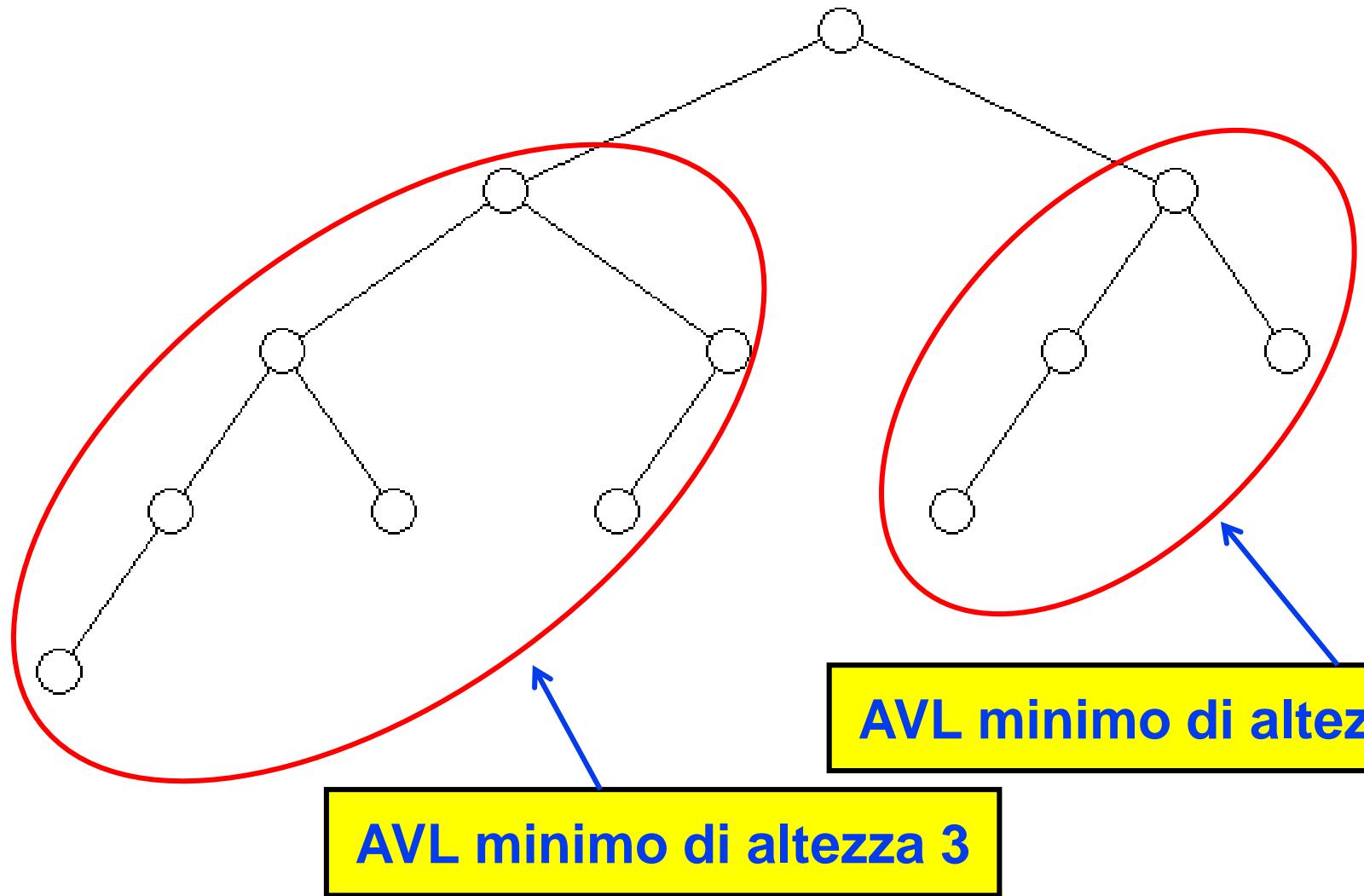
Albero AVL minimo di altezza 3



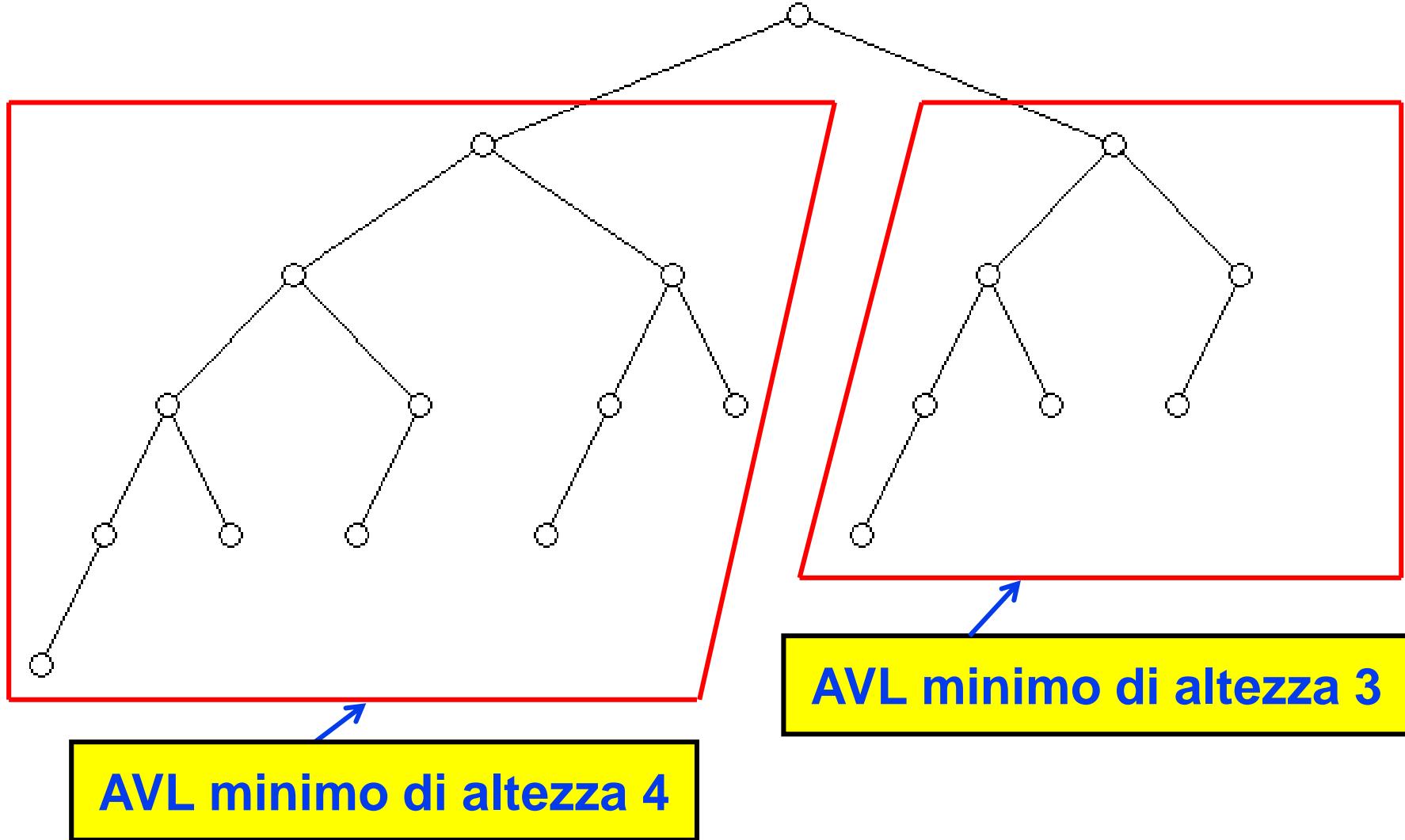
Albero AVL minimo di altezza 3



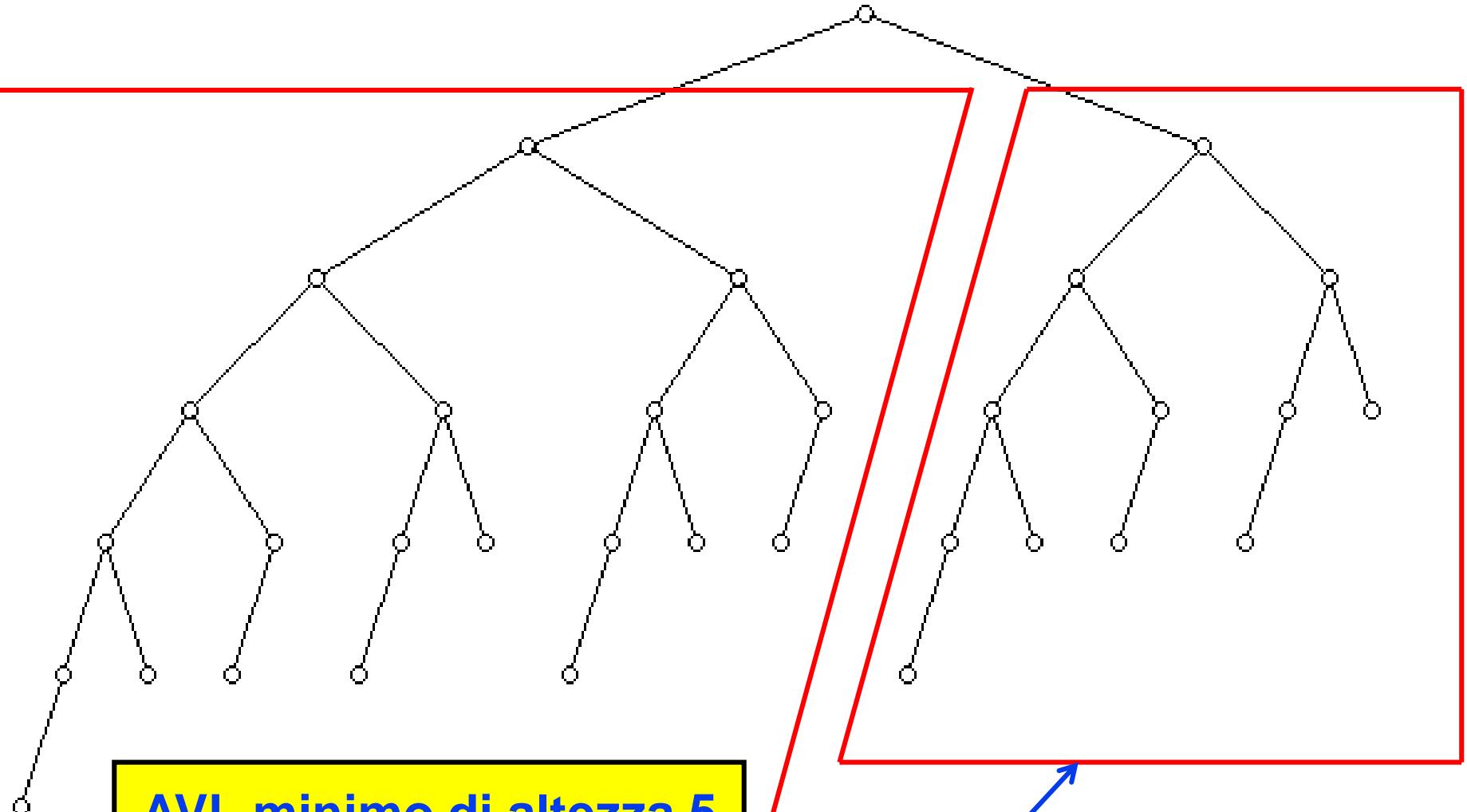
Albero AVL minimo di altezza 4



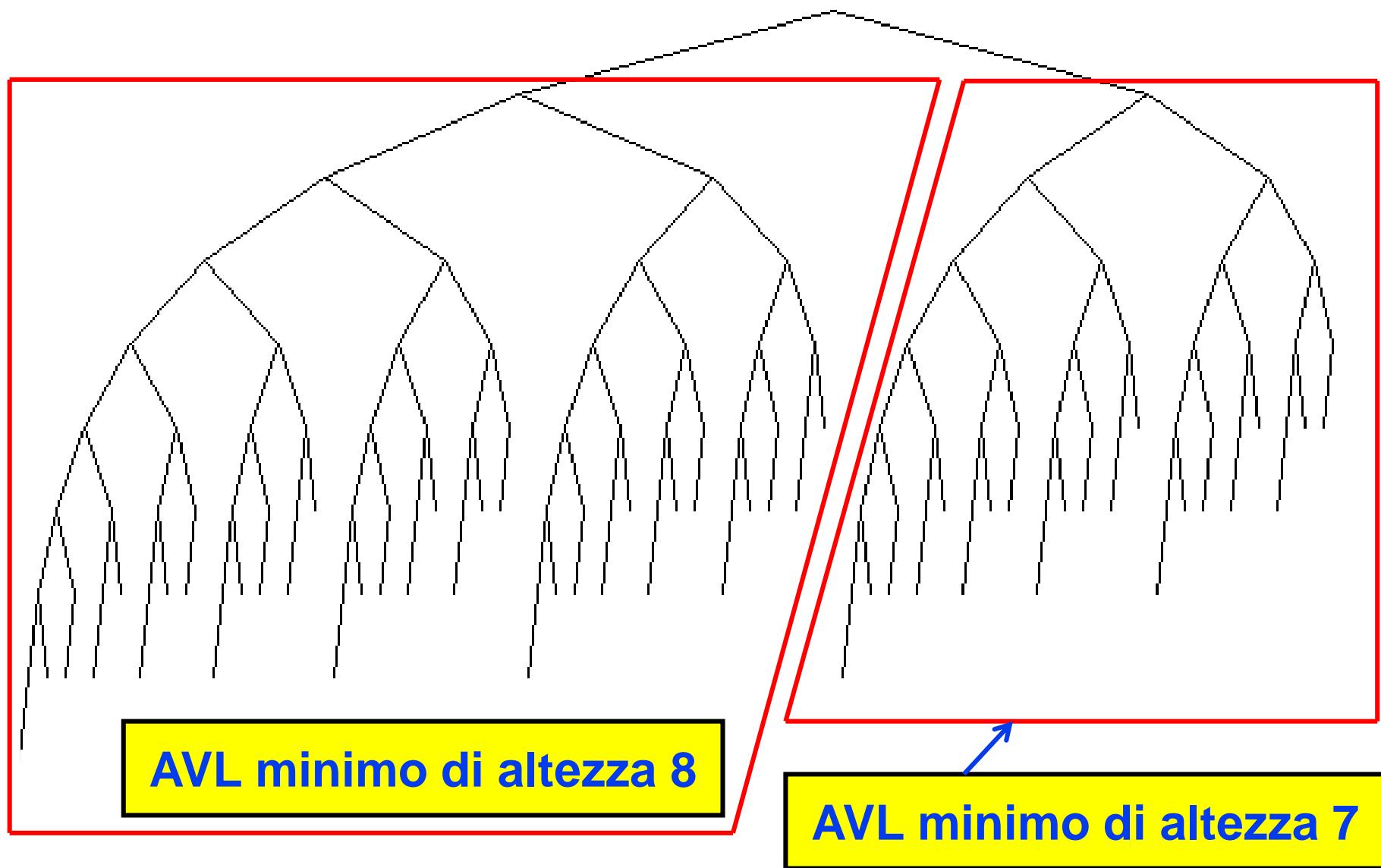
Albero AVL minimo di altezza 5



Albero AVL minimo di altezza 6



Albero AVL minimo di altezza 9



Proprietà degli alberi AVL

Dato un qualsiasi **albero AVL** con n nodi, si può dimostrare che la sua **altezza** è determinata dalla seguente formula:

$$h \cong 1.44 \log(n + 2) - 0.328$$

Alberi Red-Black (alberi rossi-neri)

Un Albero Red-Black (rosso-nero) è essenzialmente un albero binario di ricerca in cui:

- 1 Le *chiavi* vengono mantenute solo nei *nodi interni* dell'albero
- 2 Le foglie sono costituite da speciali *nodi NIL*, cioè nodi “*sentinella*” il cui contenuto è *irrilevante* e che evitano di trattare diversamente i puntatori ai nodi dai puntatori *NIL*.
 - In altre parole, al posto di un puntatore *NIL* si usa un puntatore ad un *nodo NIL*.
 - Quando un nodo ha come figli *nodi NIL*, quel nodo sarebbe una foglia nell'albero binario di ricerca corrispondente.

Alberi Red-Black (alberi rossi-neri)

Un *Albero Red-Black* (*rosso-nero*) deve soddisfare le seguenti proprietà (vincoli):

- 1 *Ogni nodo* è colorato o di *rosso* o di *nero*;
- 2 Per convezione, i *nodi NIL* si considerano *nodi neri*;
- 3 Se un *nodo* è *rosso*, allora entrambi i *suoi figli sono neri*;
- 4 Ogni percorso da un *nodo interno* ad un *nodo NIL* (figlio di una foglia) ha lo stesso *numero* di *nodi neri*;

Alberi Red-Black (alberi rossi-neri)

Un *Albero Red-Black* (*rosso-nero*) deve soddisfare le seguenti proprietà (vincoli):

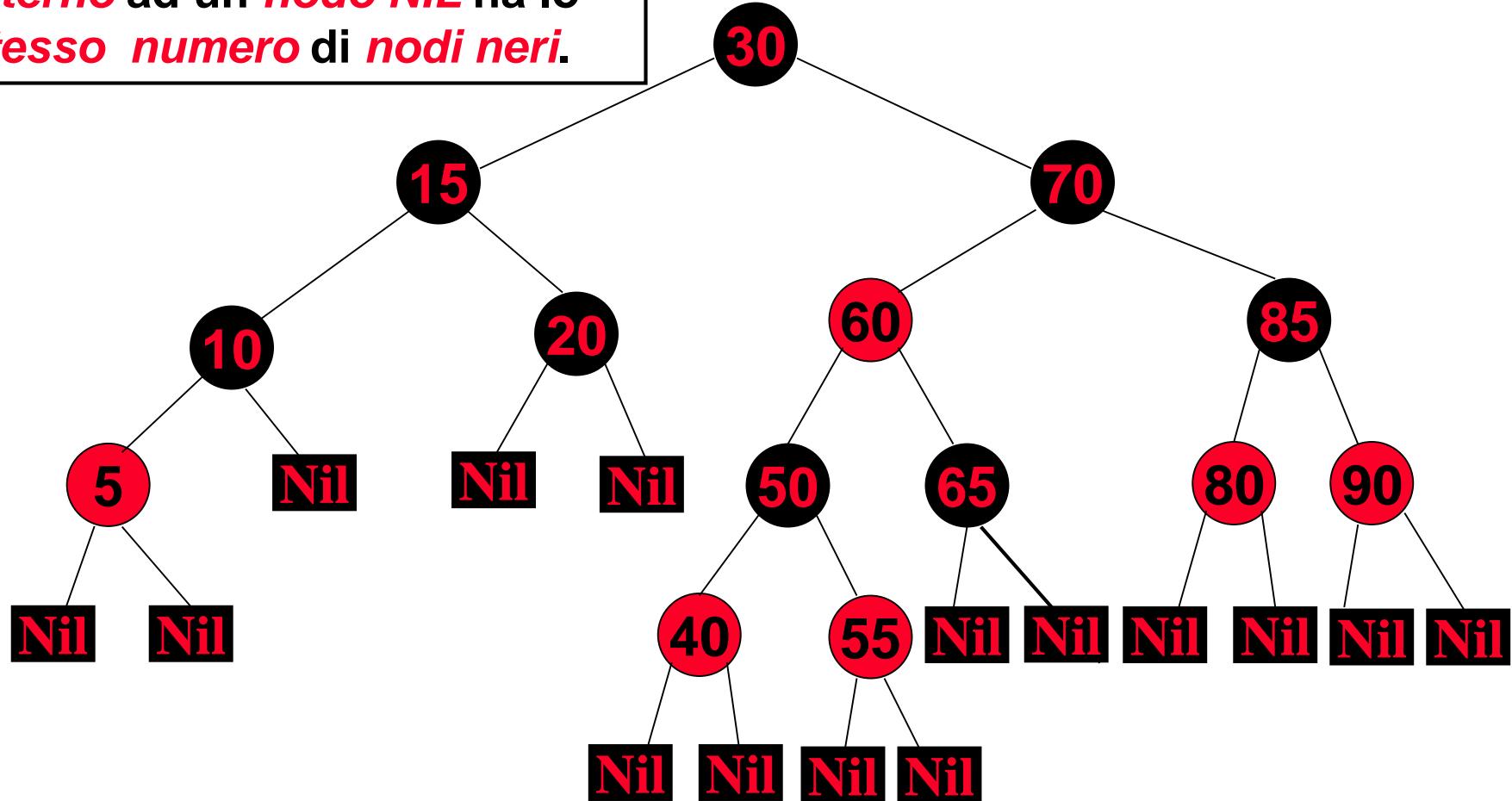
- 1 *Ogni nodo è colorato o di rosso o di nero;*
- 2 Per convezione, i *nodi NIL* si considerano *nodi neri*;
- 3 Se un *nodo è rosso*, allora entrambi i *suoi figli sono neri*;
- 4 Ogni percorso da un *nodo interno* ad un *nodo NIL* (figlio di una foglia) ha lo stesso *numero di nodi neri*;

Considereremo solo *alberi Red-Black* in cui la *radice è nera*.

Alberi Red-Black: esempio I

3 Se un *nodo* è *rosso*, allora entrambi i *suoi figli* sono *neri*;

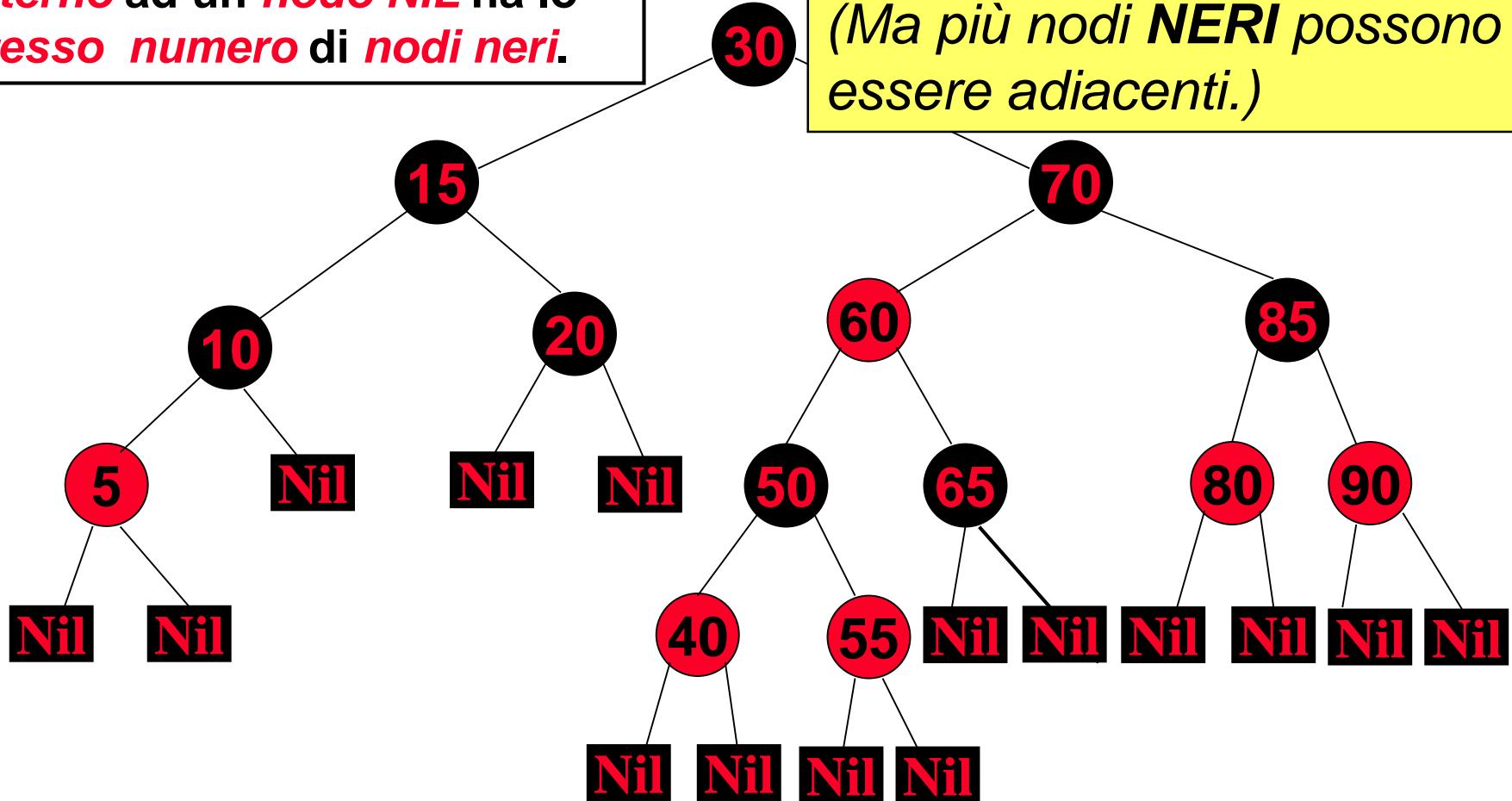
4 Ogni percorso da un *nodo interno* ad un *nodo NIL* ha lo stesso *numero di nodi neri*.



Alberi Red-Black: esempio I

- 3 Se un *nodo è rosso*, allora entrambi i *suoi figli sono neri*;
4 Ogni percorso da un *nodo interno* ad un *nodo NIL* ha lo stesso *numero di nodi neri*.

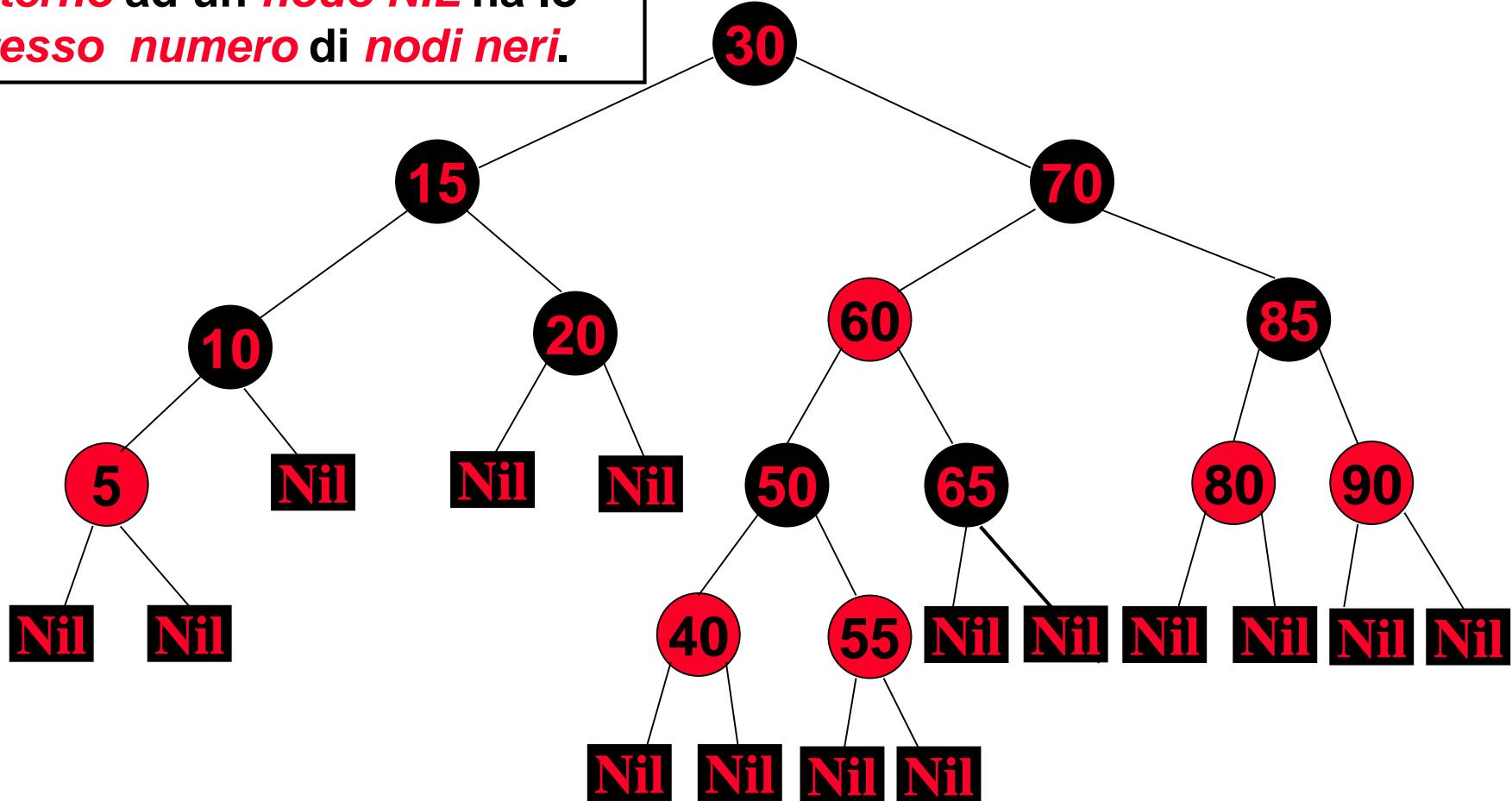
Vincolo 3 impone che *non possano esserci due nodi ROSSI adiacenti.*
(Ma più nodi NERI possono essere adiacenti.)



Alberi Red-Black: esempio I

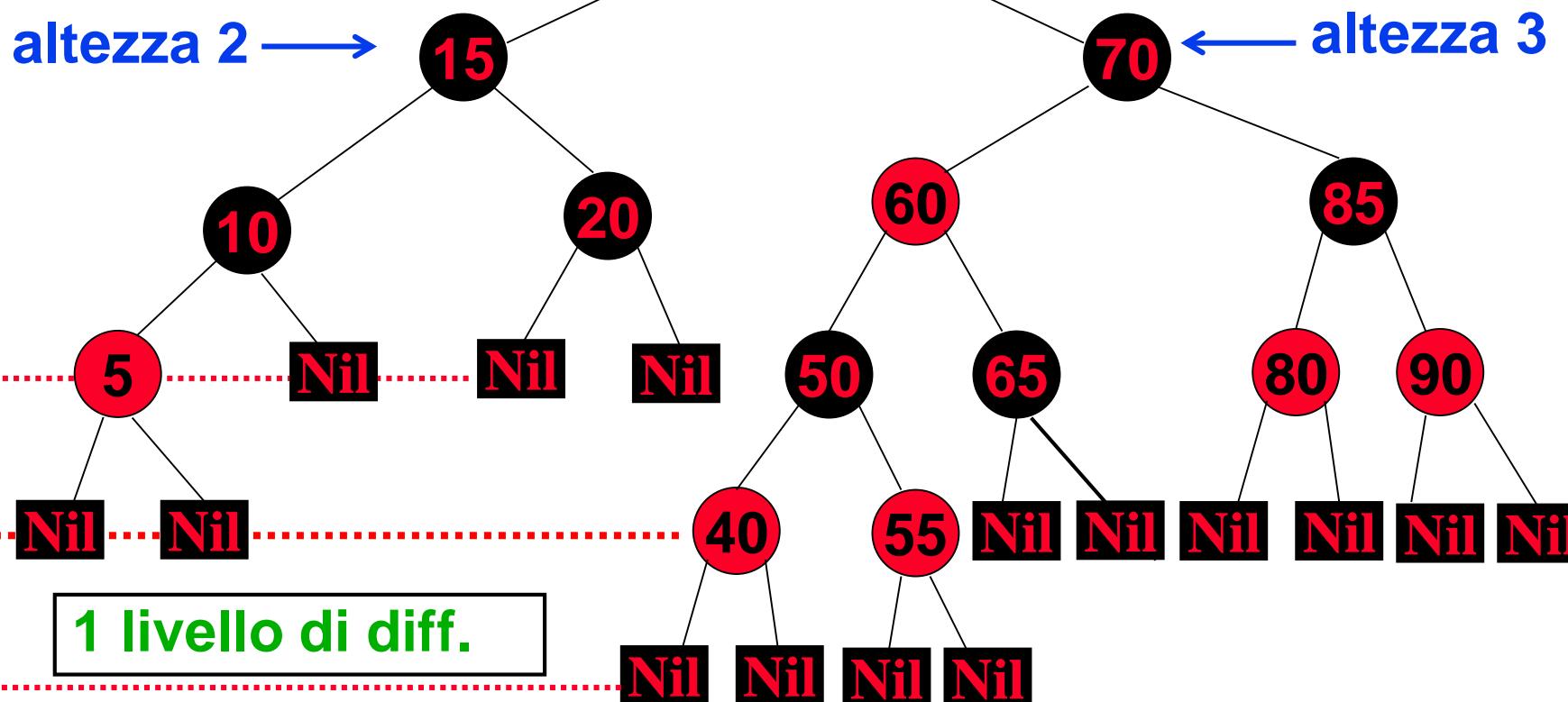
- 3 Se un *nodo è rosso*, allora entrambi i *suoi figli sono neri*;
- 4 Ogni percorso da un *nodo interno* ad un *nodo NIL* ha lo stesso *numero di nodi neri*.

Ci sono 3 *nodi neri* lungo *ogni percorso* dalla radice ad un *nodo NIL*



Alberi Red-Black: esempio I

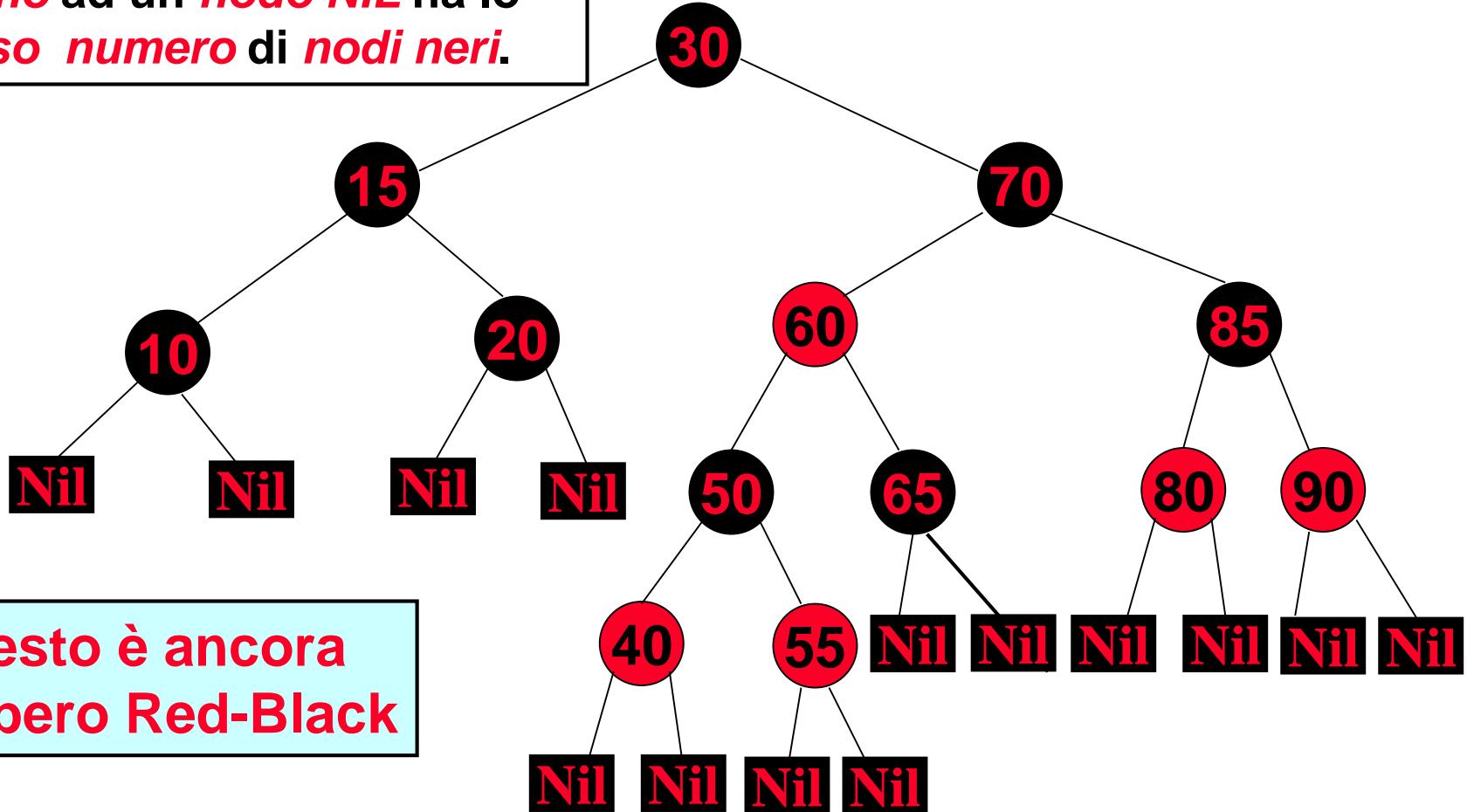
- 3 Se un *nodo è rosso*, allora entrambi i *suoi figli sono neri*;
- 4 Ogni percorso da un *nodo interno* ad un *nodo NIL* ha lo stesso *numero di nodi neri*.



Alberi Red-Black: esempio II

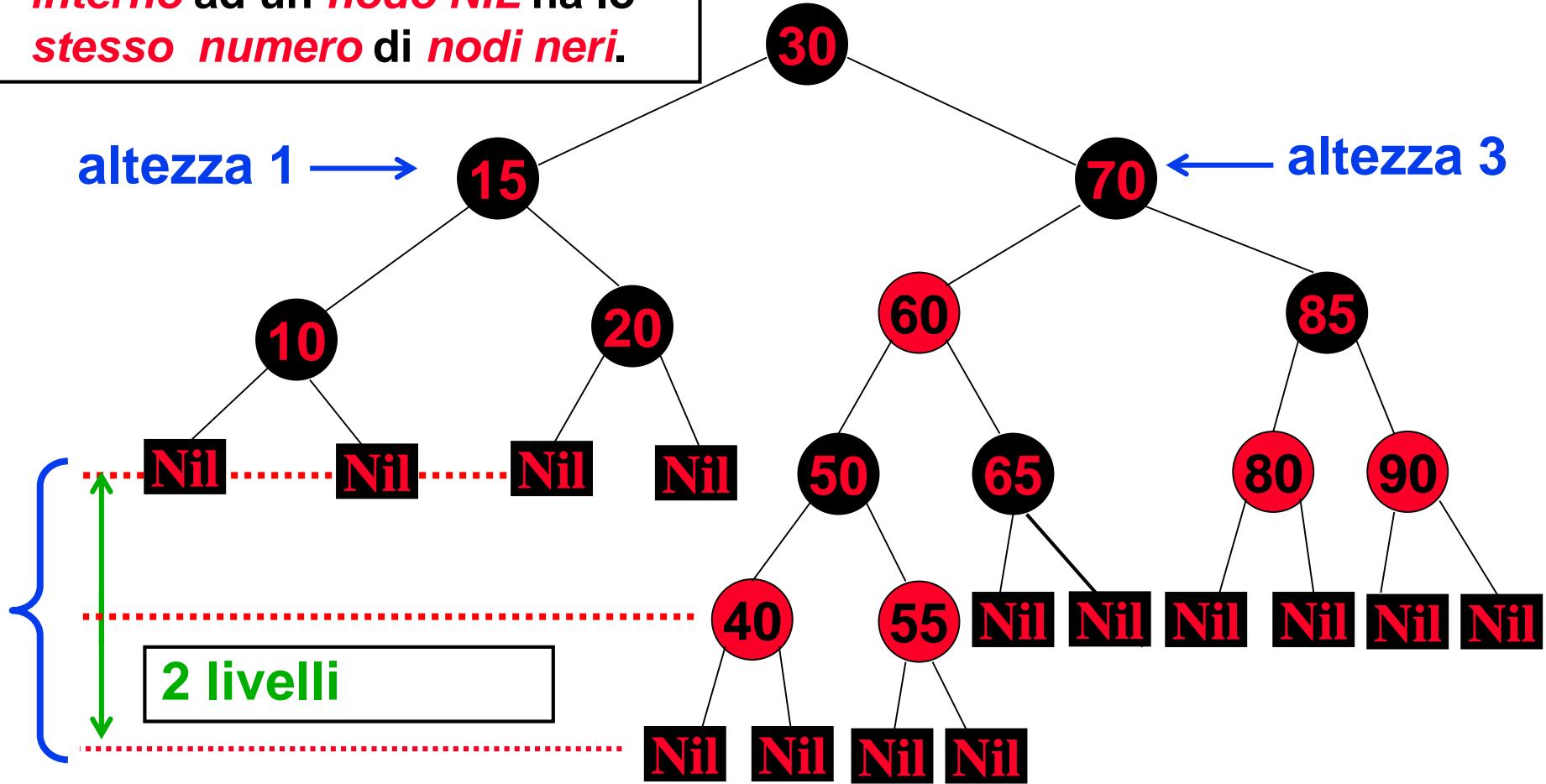
3 Se un *nodo* è *rosso*, allora entrambi i *suoi figli* sono *neri*;

4 Ogni percorso da un *nodo interno* ad un *nodo NIL* ha lo stesso *numero di nodi neri*.



Alberi Red-Black: esempio II

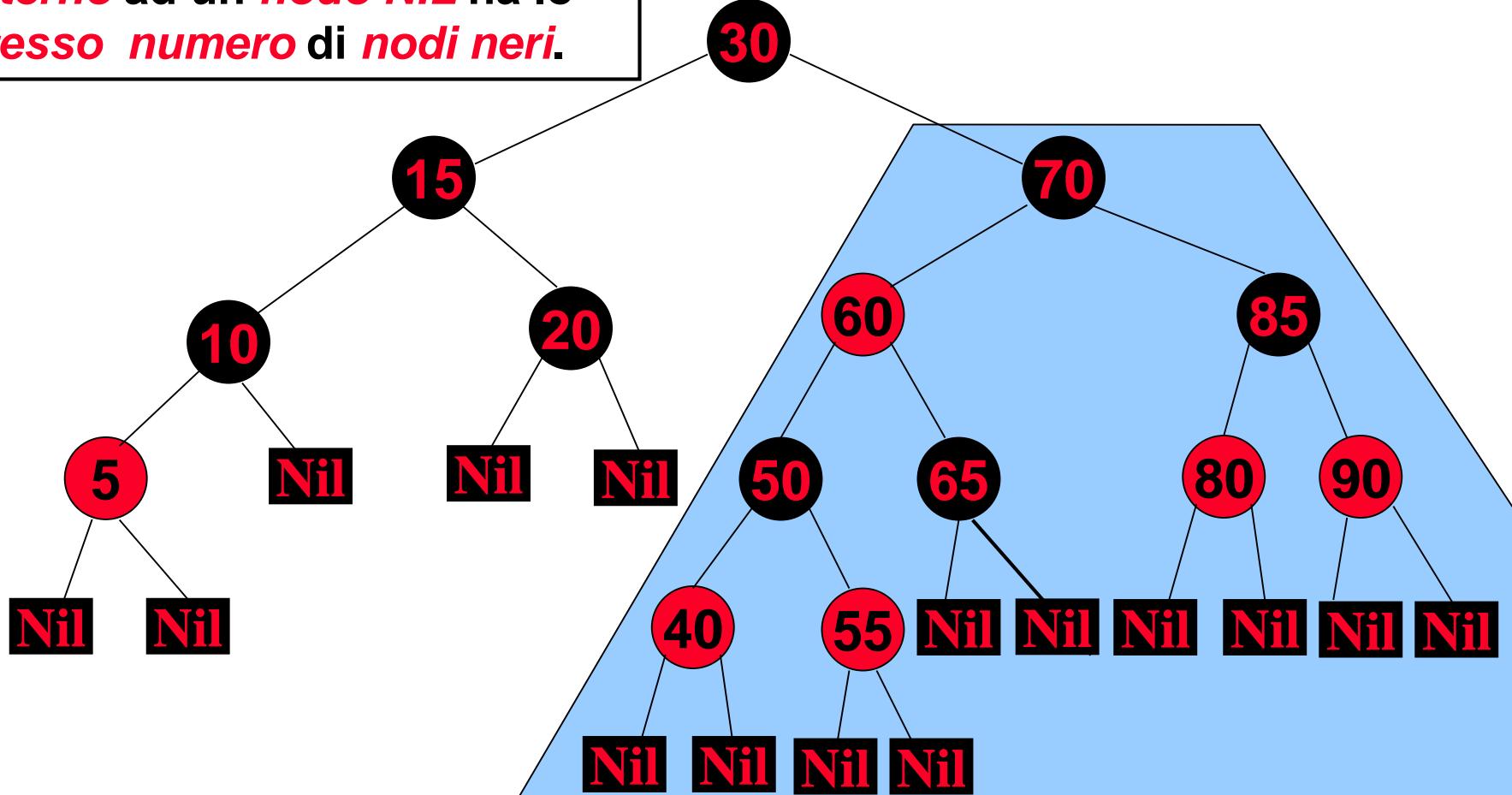
- 3 Se un *nodo* è *rosso*, allora entrambi i *suoi figli* sono *neri*;
- 4 Ogni percorso da un *nodo interno* ad un *nodo NIL* ha lo stesso *numero di nodi neri*.



Alberi Red-Black: esempio I

- 3 Se un *nodo è rosso*, allora entrambi i *suoi figli sono neri*;
- 4 Ogni percorso da un *nodo interno* ad un *nodo NIL* ha lo stesso *numero di nodi neri*.

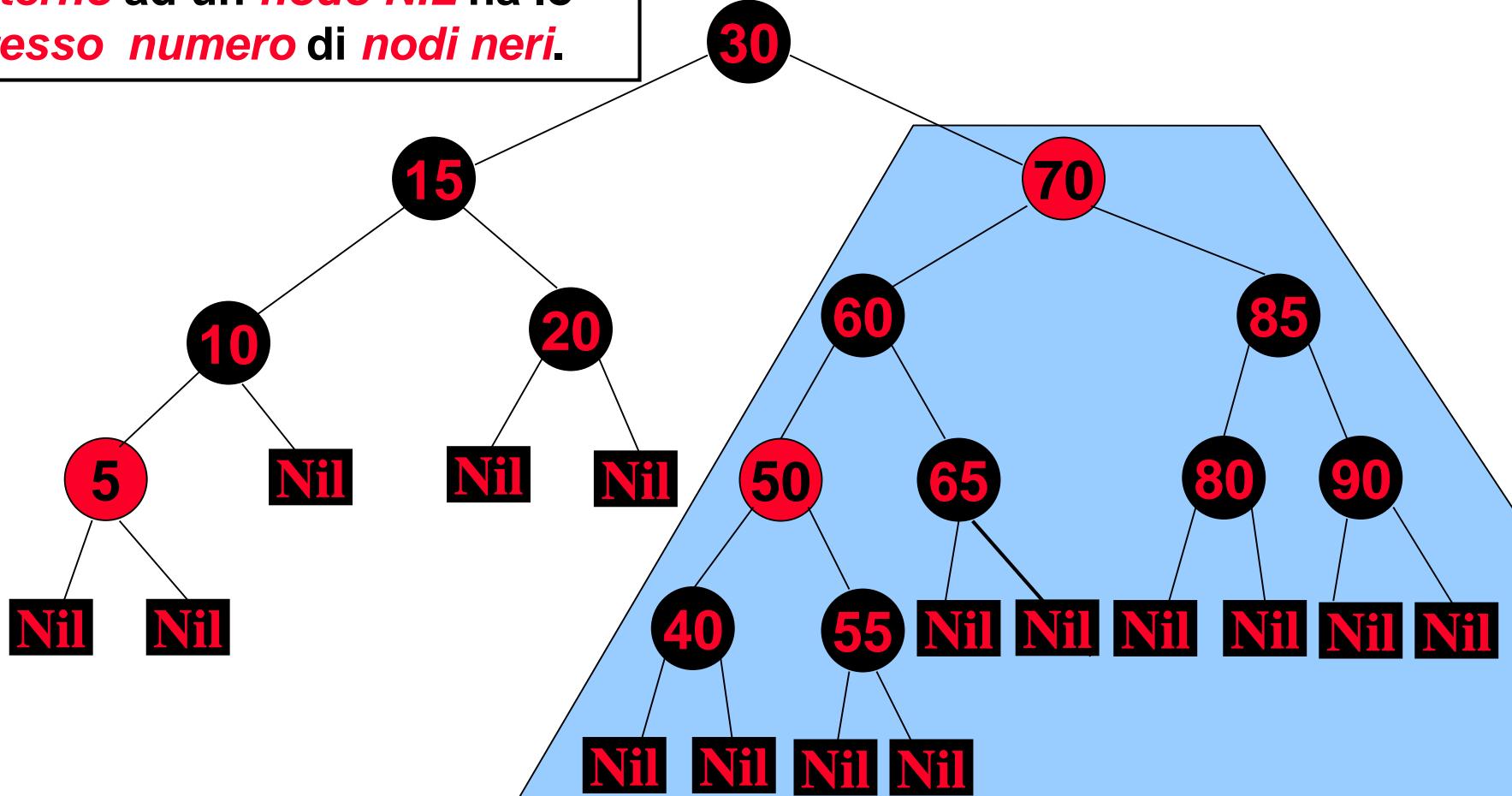
Ci sono 3 *nodi neri* lungo *ogni percorso* dalla radice ad un *nodo NIL*



Alberi Red-Black: esempio III

- 3 Se un *nodo è rosso*, allora entrambi i *suoi figli sono neri*;
4 Ogni percorso da un *nodo interno* ad un *nodo NIL* ha lo stesso *numero di nodi neri*.

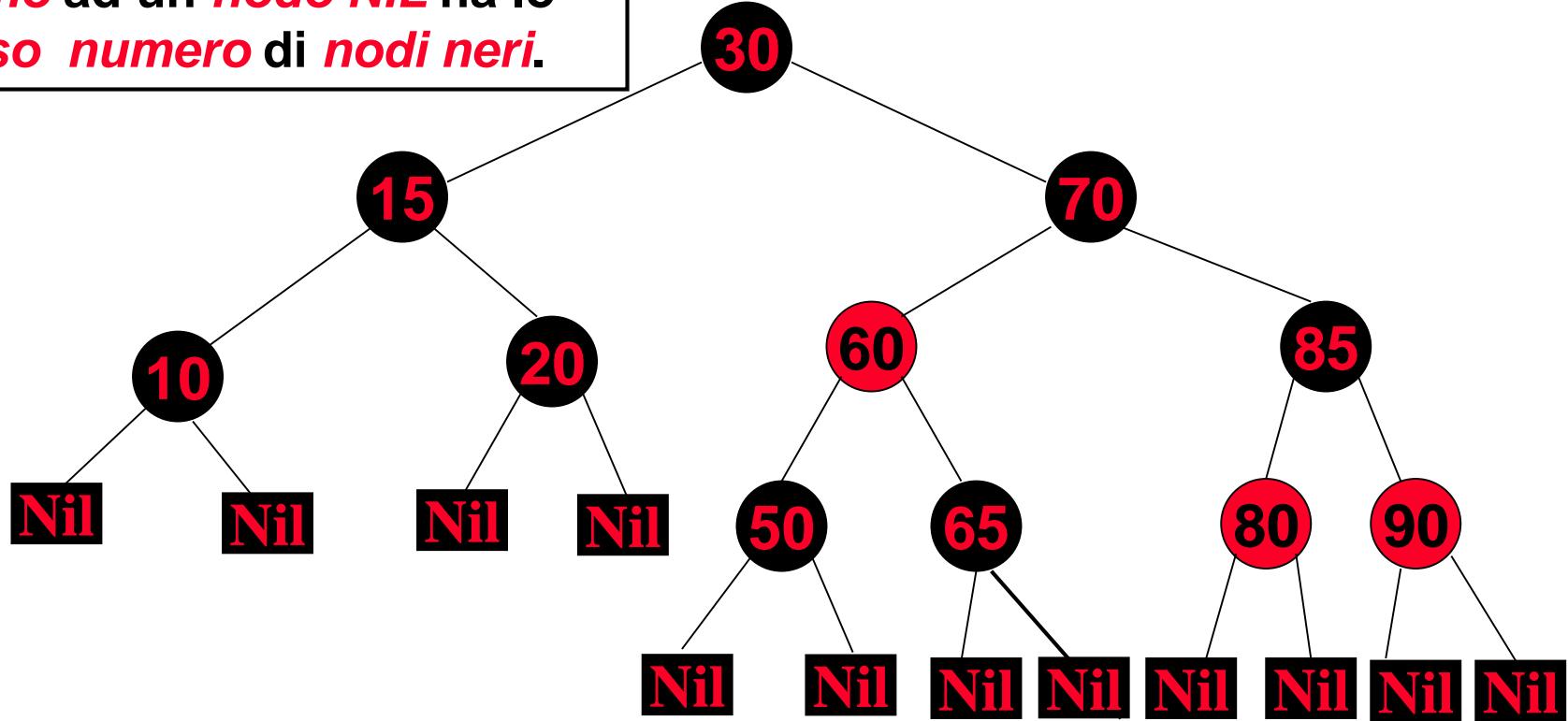
Ci sono 3 *nodi neri* lungo *ogni percorso* dalla radice ad un *nodo NIL*



Alberi Red-Black: esempio IV

- 3 Se un *nodo* è *rosso*, allora entrambi i *suoi figli* sono *neri*;
4 Ogni percorso da un *nodo interno* ad un *nodo NIL* ha lo stesso *numero di nodi neri*.

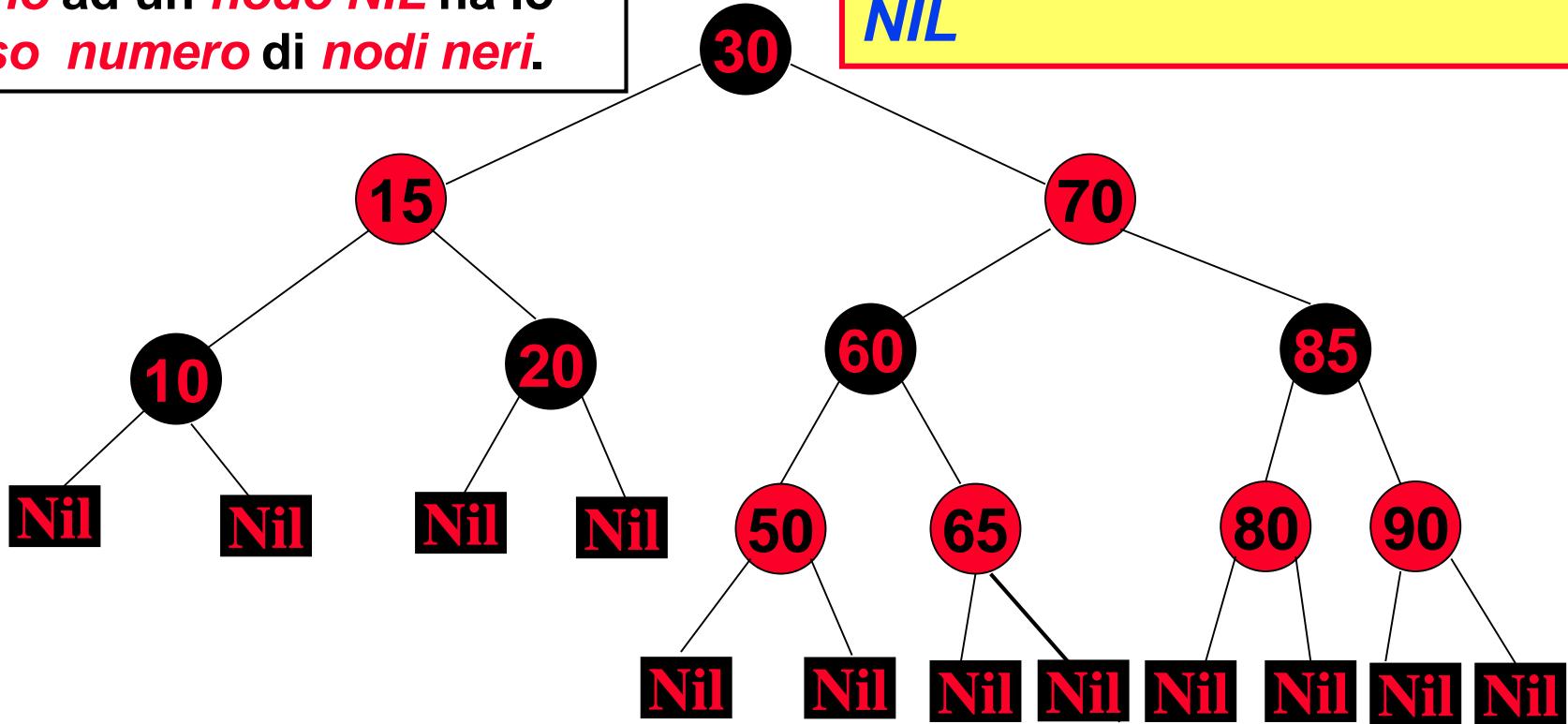
Albero RB con 3 nodi neri lungo *ogni percorso* dalla radice ad un *nodo NIL*



Alberi Red-Black: esempio V

- 3 Se un *nodo è rosso*, allora entrambi i *suoi figli sono neri*;
- 4 Ogni percorso da un *nodo interno* ad un *nodo NIL* ha lo stesso *numero di nodi neri*.

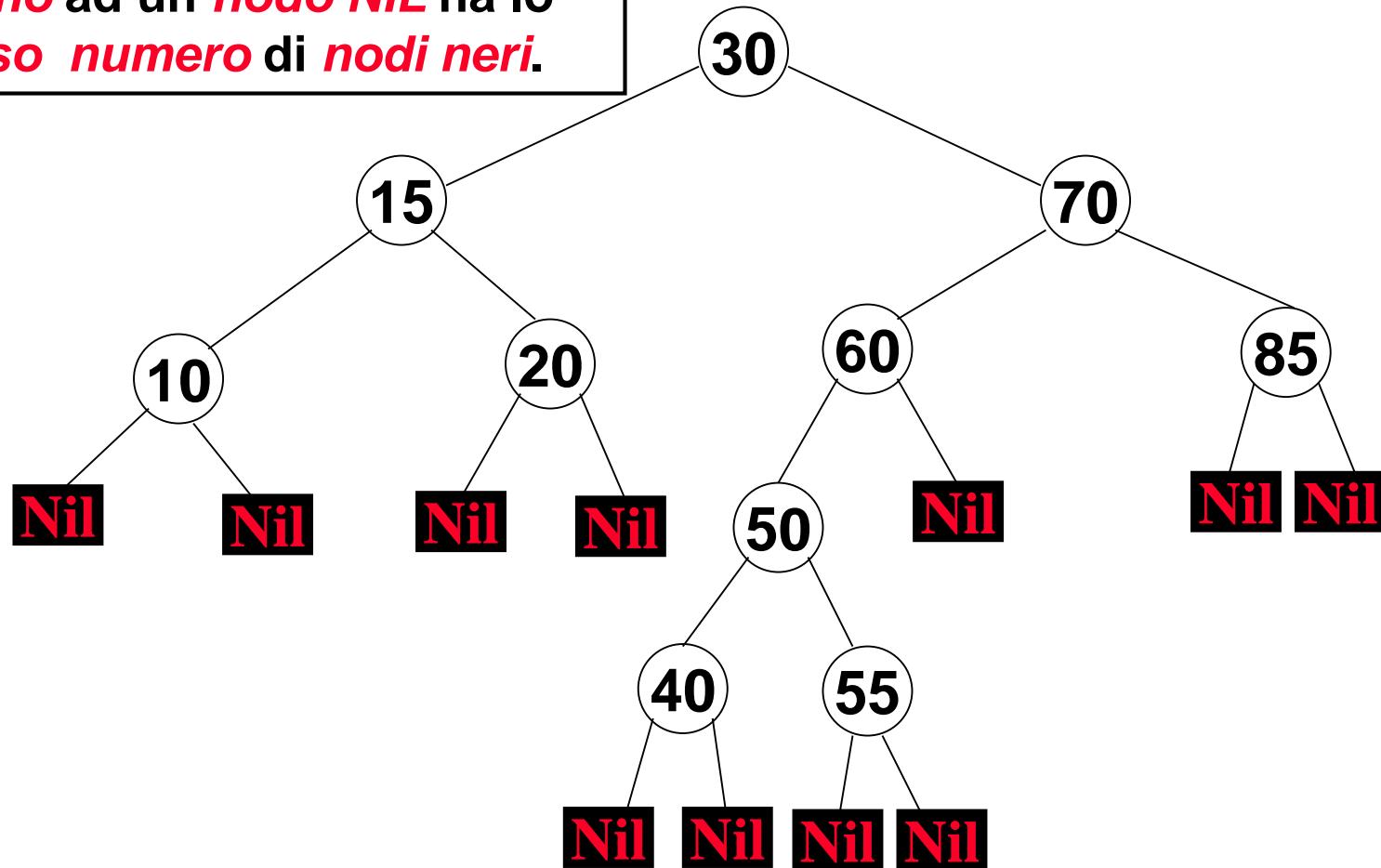
Stesso albero con 2 *nodi neri* lungo *ogni percorso* dalla radice ad un *nodo NIL*



Alberi Red-Black: esempio VI

- 3 Se un *nodo è rosso*, allora entrambi i *suoi figli sono neri*;
4 Ogni percorso da un *nodo interno* ad un *nodo NIL* ha lo stesso *numero di nodi neri*.

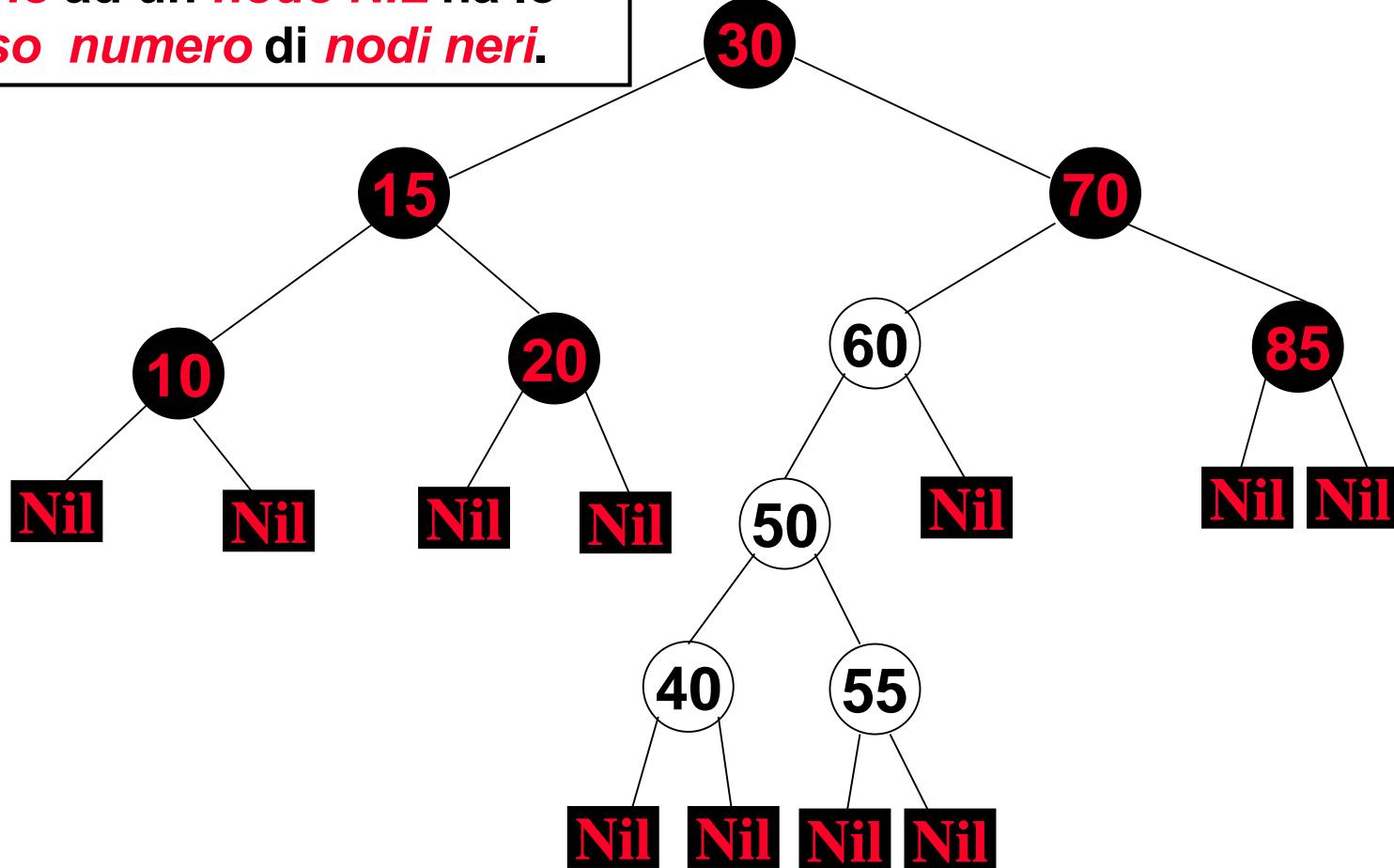
Questo albero è un albero Red-Black?



Alberi Red-Black: esempio VI

- 3 Se un *nodo è rosso*, allora entrambi i *suoi figli sono neri*;
4 Ogni percorso da un *nodo interno* ad un *nodo NIL* ha lo stesso *numero di nodi neri*.

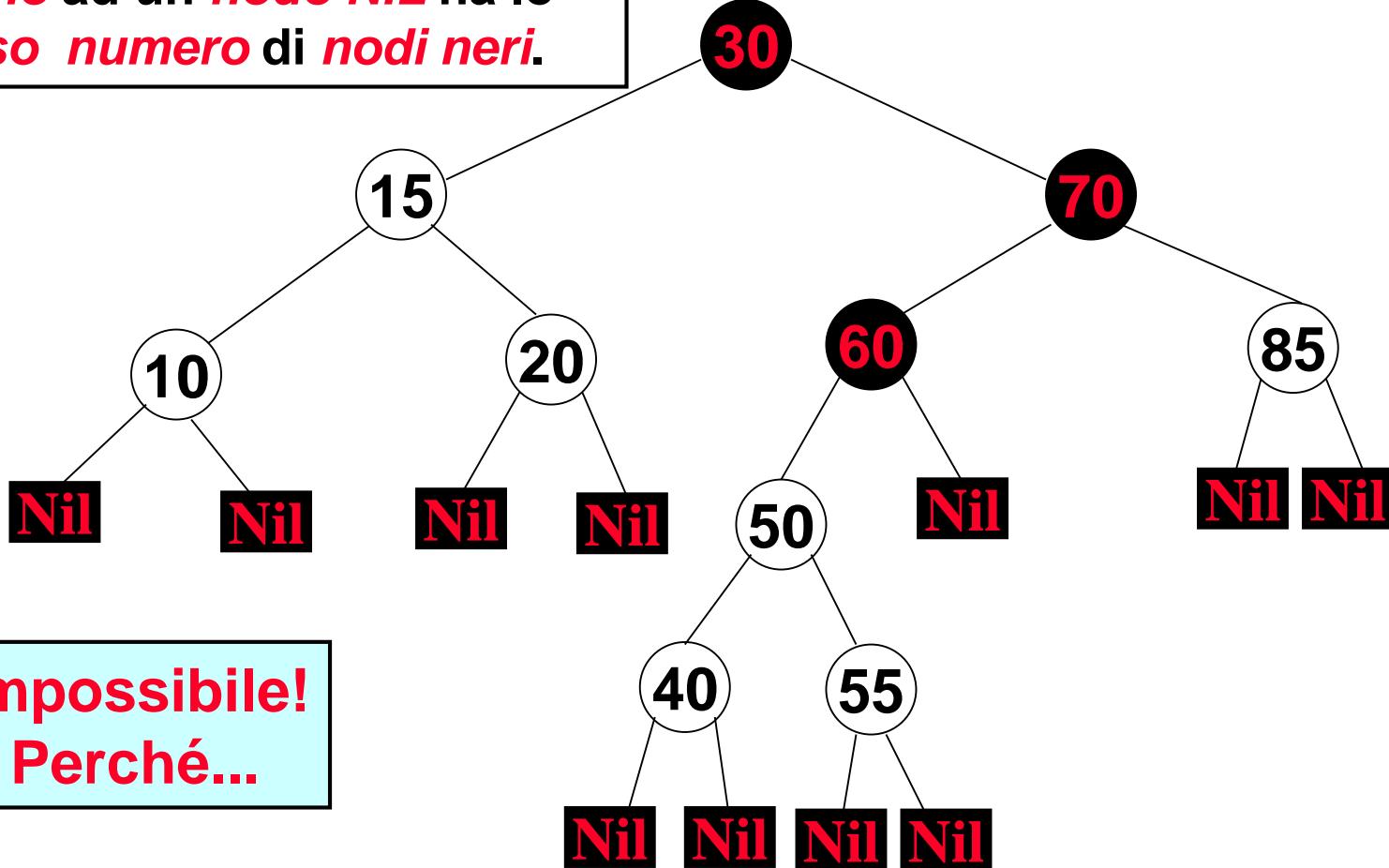
Per vincolo 4 ci possono essere al più 3 nodi neri lungo un percorso!



Alberi Red-Black: esempio VI

- 3 Se un *nodo è rosso*, allora entrambi i *suoi figli sono neri*;
4 Ogni percorso da un *nodo interno* ad un *nodo NIL* ha lo stesso *numero di nodi neri*.

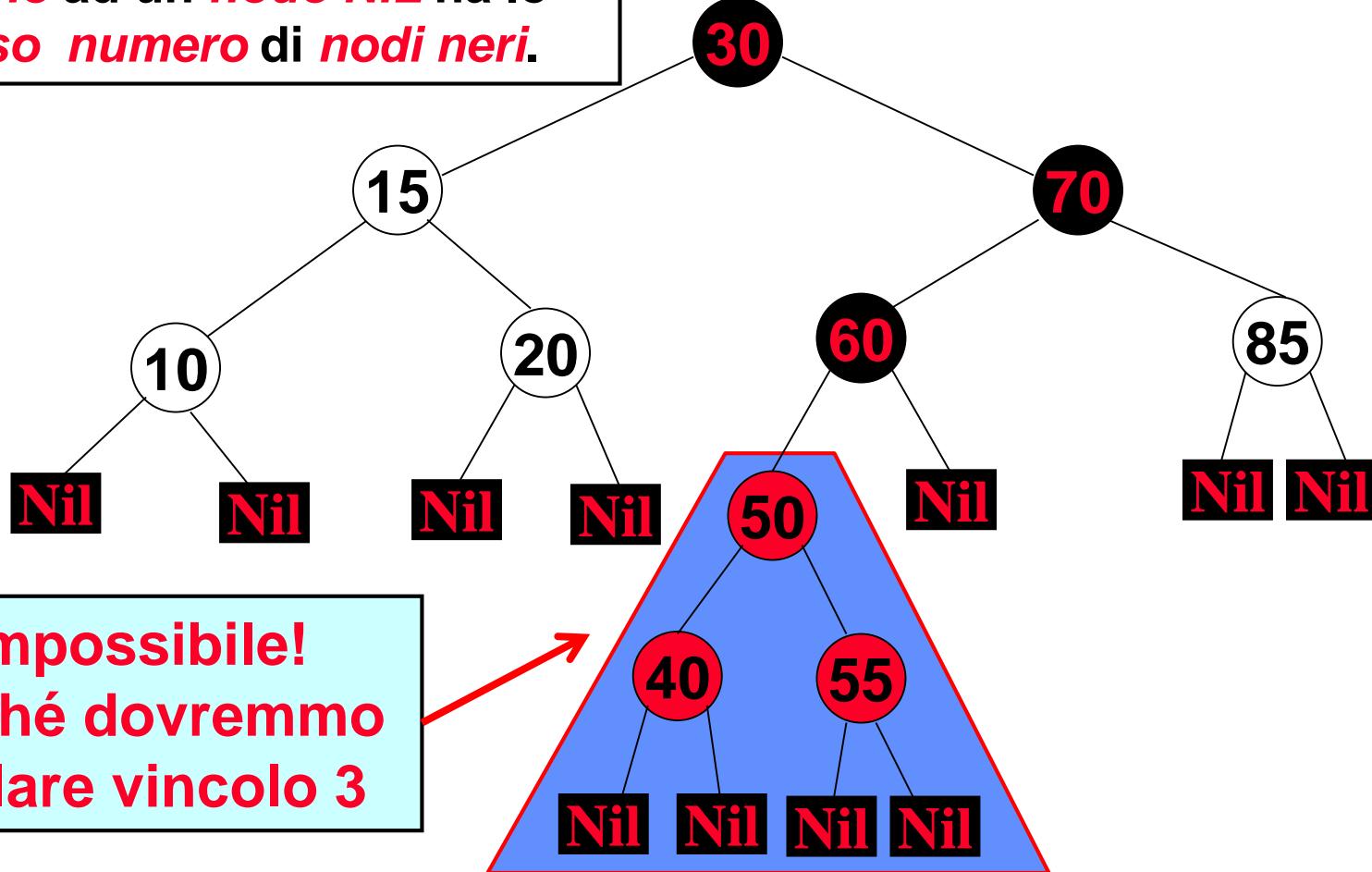
Per vincolo 4 ci possono essere al più 3 nodi neri lungo un percorso!



Alberi Red-Black: esempio VI

- 3 Se un *nodo è rosso*, allora entrambi i *suoi figli sono neri*;
4 Ogni percorso da un *nodo interno* ad un *nodo NIL* ha lo stesso *numero di nodi neri*.

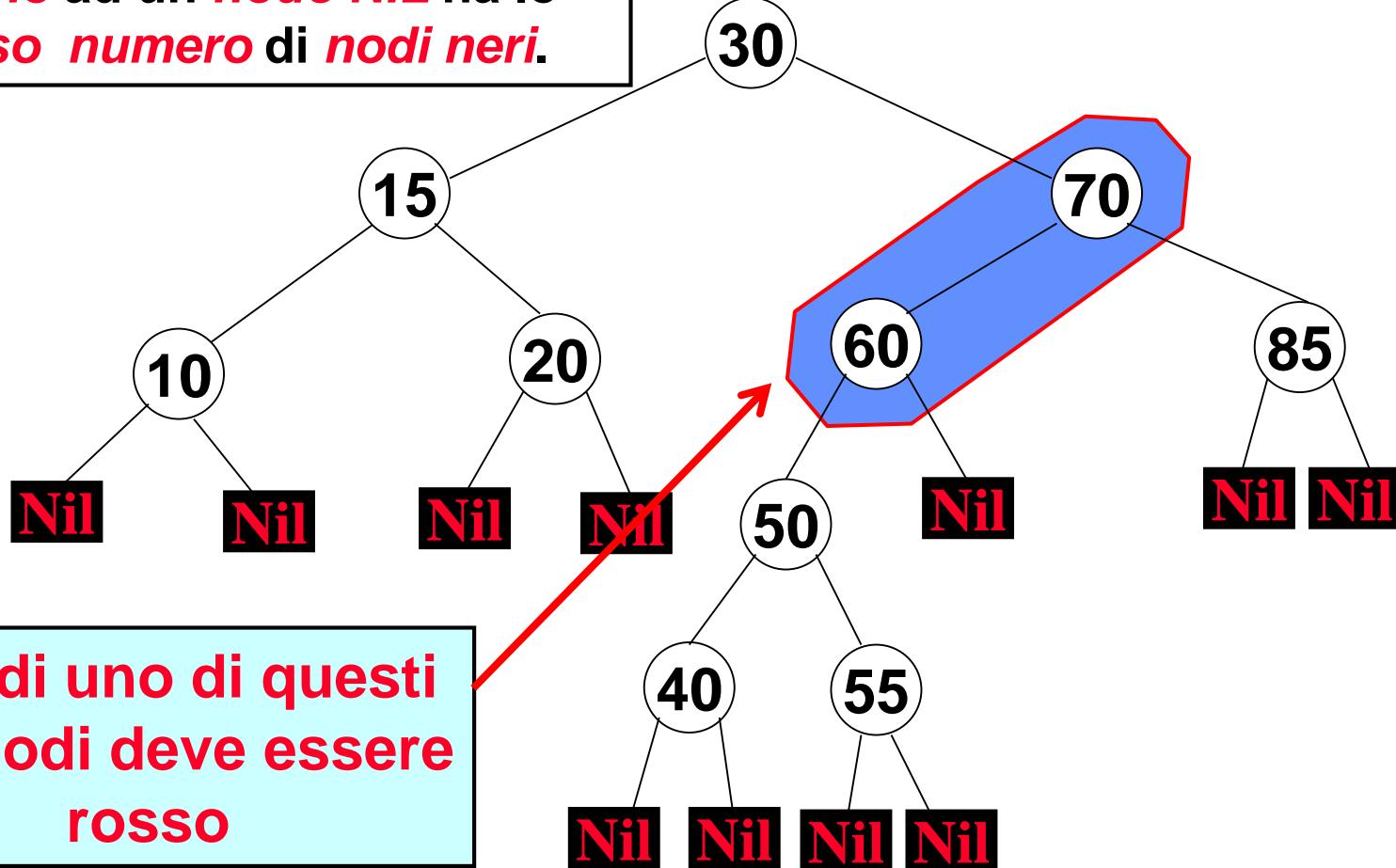
Per vincolo 4 ci possono essere al più 3 nodi neri lungo un percorso!



Alberi Red-Black: esempio VI

- 3 Se un *nodo* è *rosso*, allora entrambi i *suoi figli* sono *neri*;
4 Ogni percorso da un *nodo interno* ad un *nodo NIL* ha lo stesso *numero di nodi neri*.

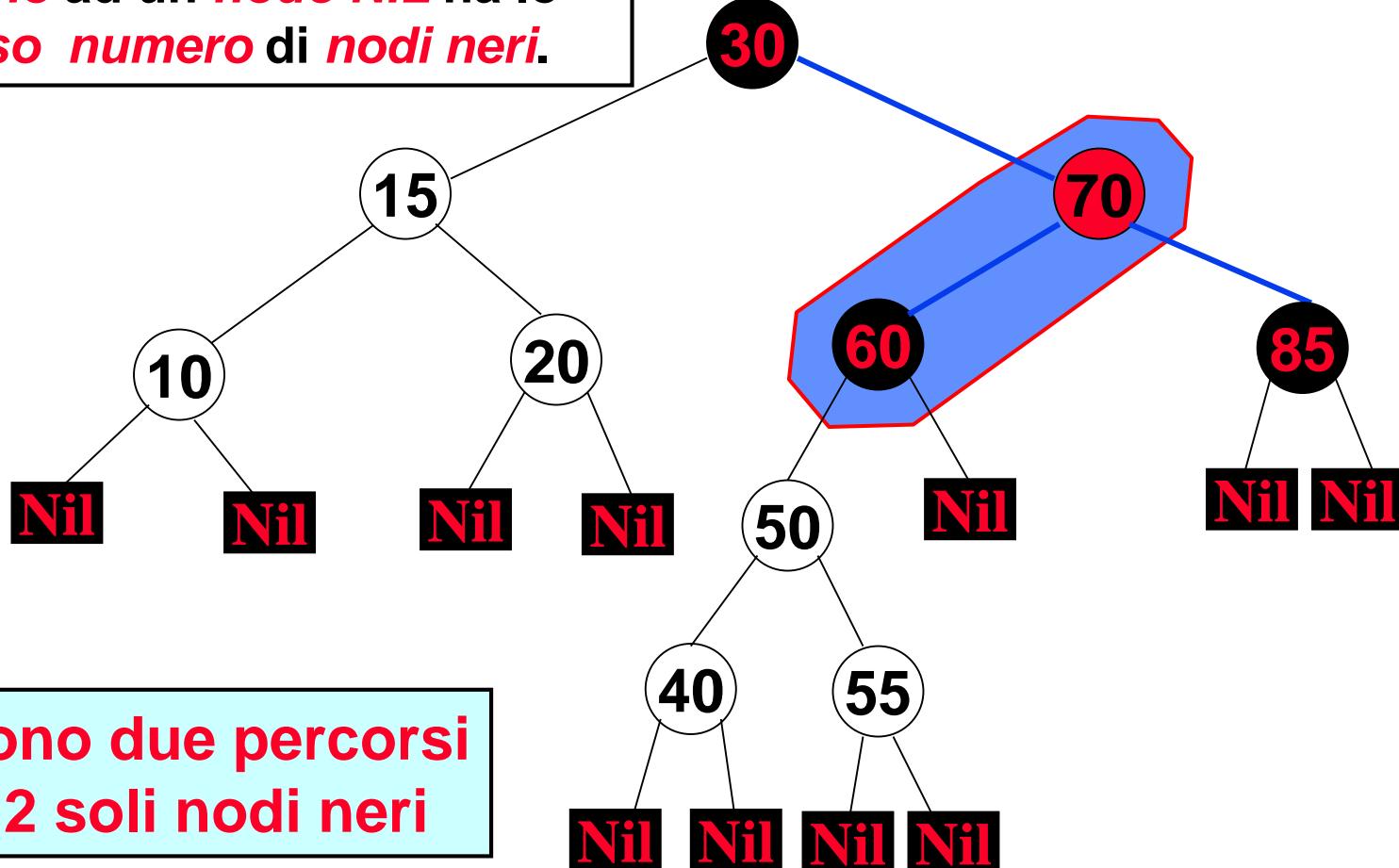
Per vincolo 4 ci possono essere al più 3 nodi neri lungo un percorso!



Alberi Red-Black: esempio VI

- 3 Se un *nodo è rosso*, allora entrambi i *suoi figli sono neri*;
4 Ogni percorso da un *nodo interno* ad un *nodo NIL* ha lo stesso *numero di nodi neri*.

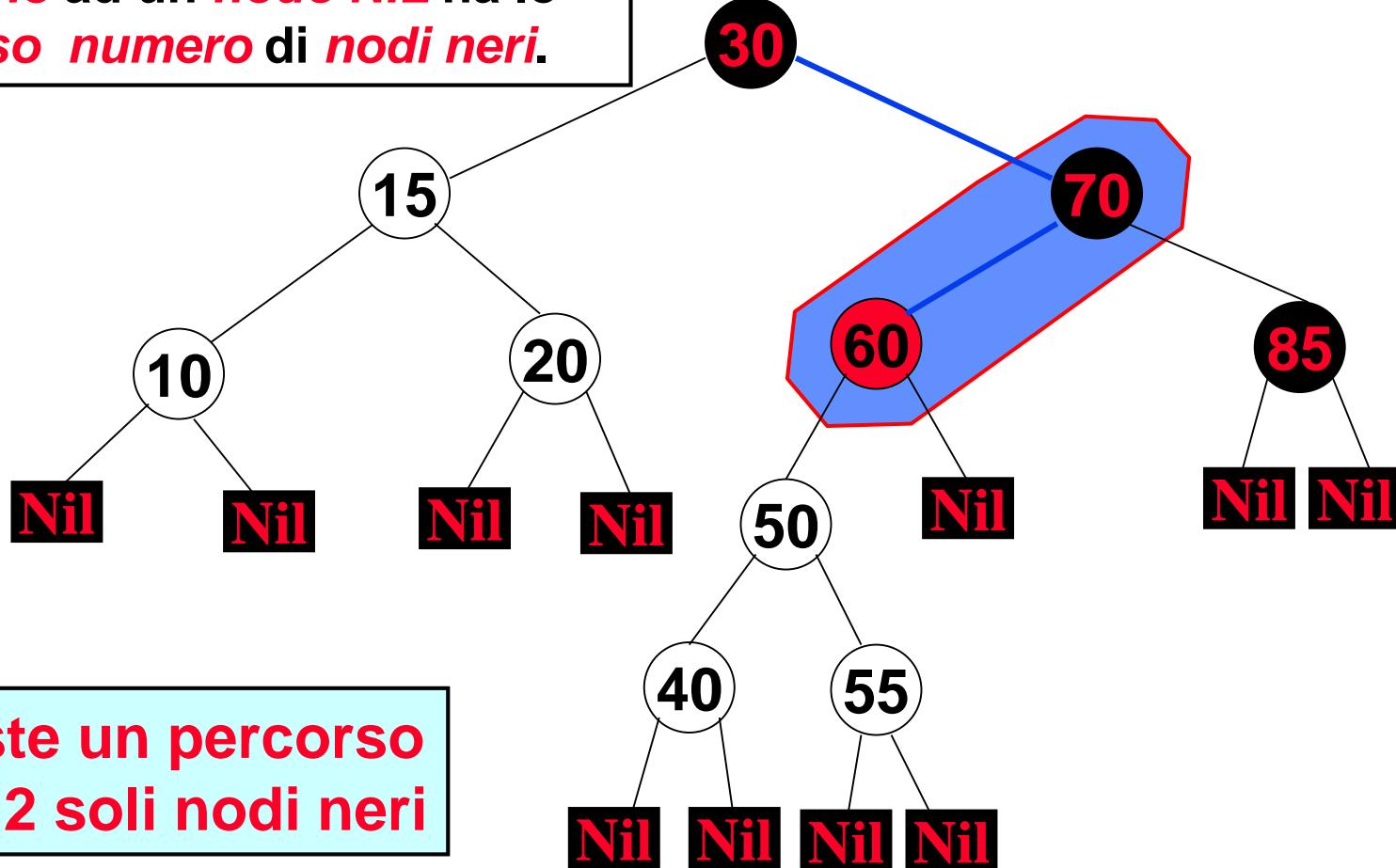
Per vincolo 4 ci possono essere al più 3 nodi neri lungo un percorso!



Alberi Red-Black: esempio VI

- 3 Se un *nodo è rosso*, allora entrambi i *suoi figli sono neri*;
4 Ogni percorso da un *nodo interno* ad un *nodo NIL* ha lo stesso *numero di nodi neri*.

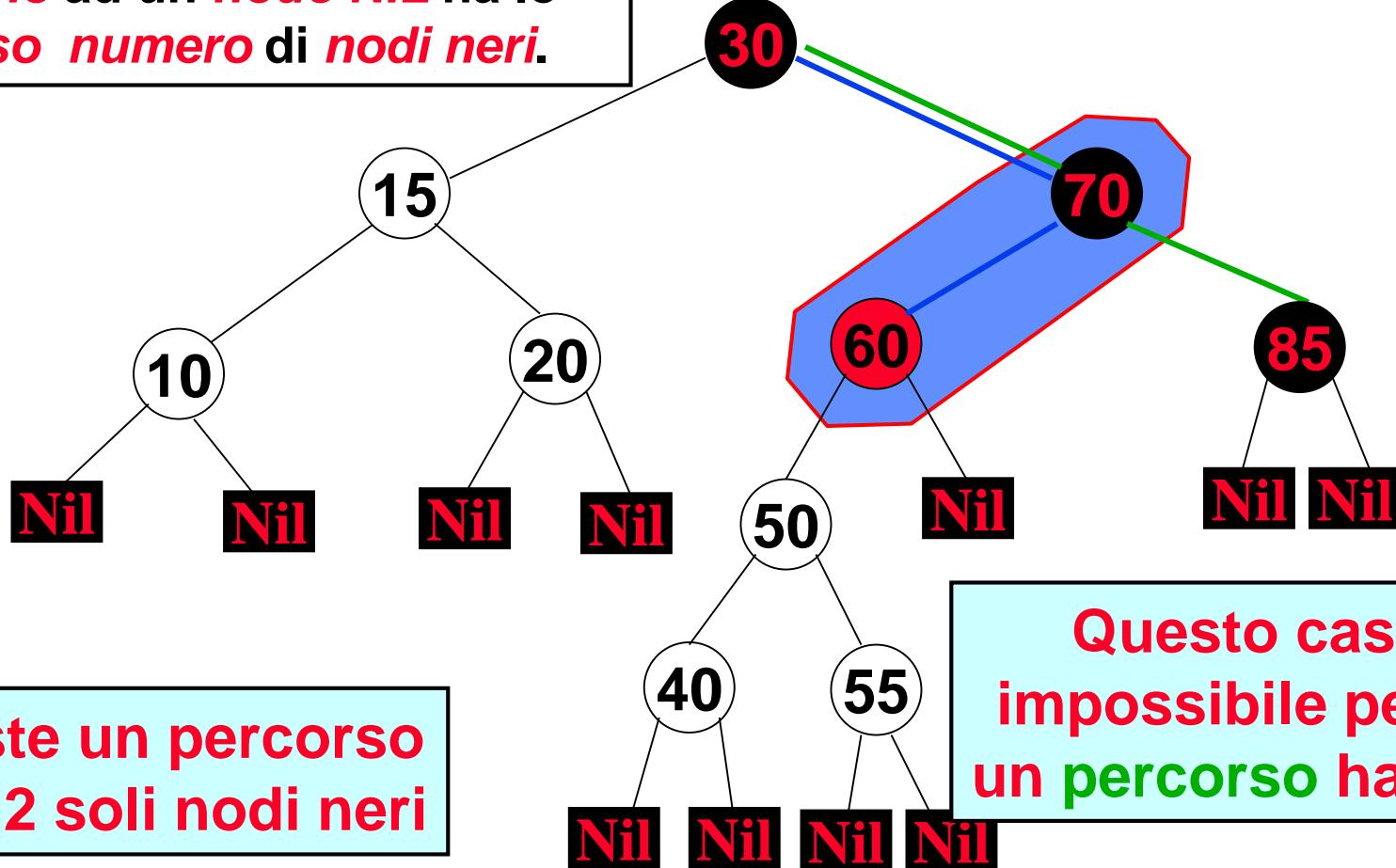
Per vincolo 4 ci possono essere al più 3 nodi neri lungo un percorso!



Alberi Red-Black: esempio VI

- 3 Se un *nodo è rosso*, allora entrambi i *suoi figli sono neri*;
4 Ogni percorso da un *nodo interno* ad un *nodo NIL* ha lo stesso *numero di nodi neri*.

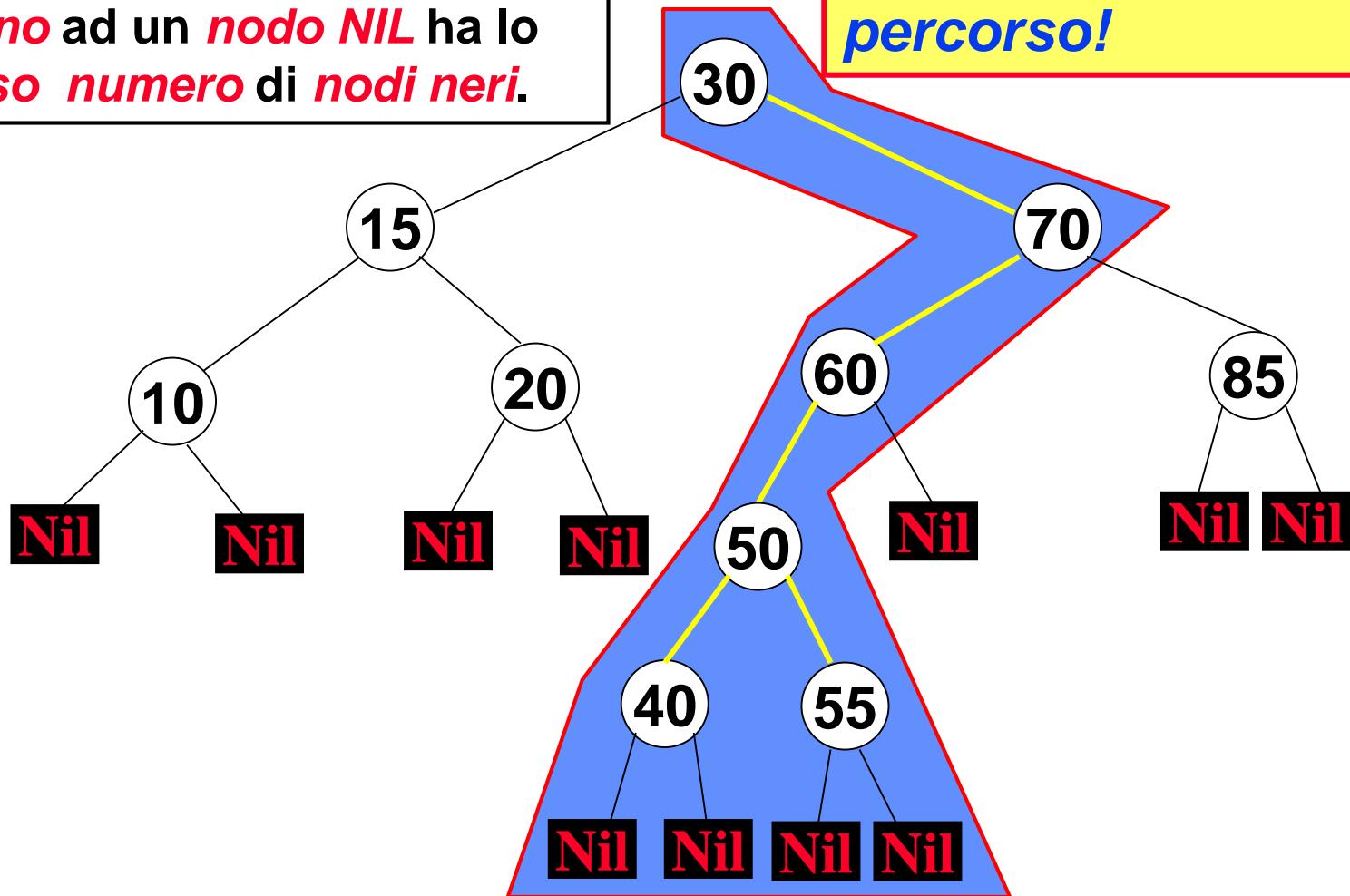
Per vincolo 4 ci possono essere al più 3 nodi neri lungo un percorso!



Alberi Red-Black: esempio VI

- 3 Se un *nodo è rosso*, allora entrambi i *suoi figli sono neri*;
4 Ogni percorso da un *nodo interno* ad un *nodo NIL* ha lo stesso *numero di nodi neri*.

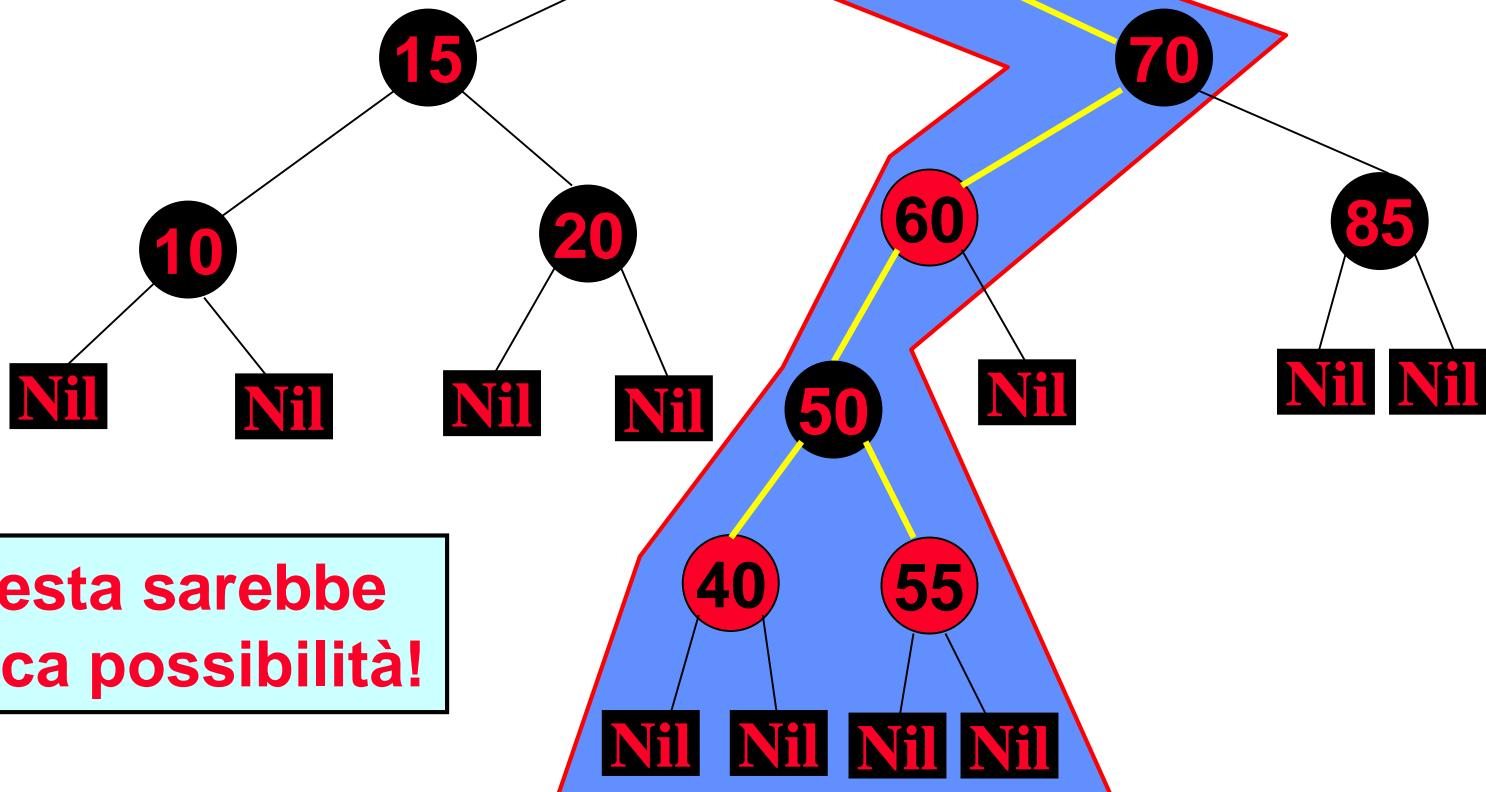
Per *vincolo 4 e il vincolo 3*, ci possono essere al più 2 nodi neri lungo un percorso!



Alberi Red-Black: esempio VI

- 3 Se un *nodo è rosso*, allora entrambi i *suoi figli sono neri*;
4 Ogni percorso da un *nodo interno* ad un *nodo NIL* ha lo stesso *numero di nodi neri*.

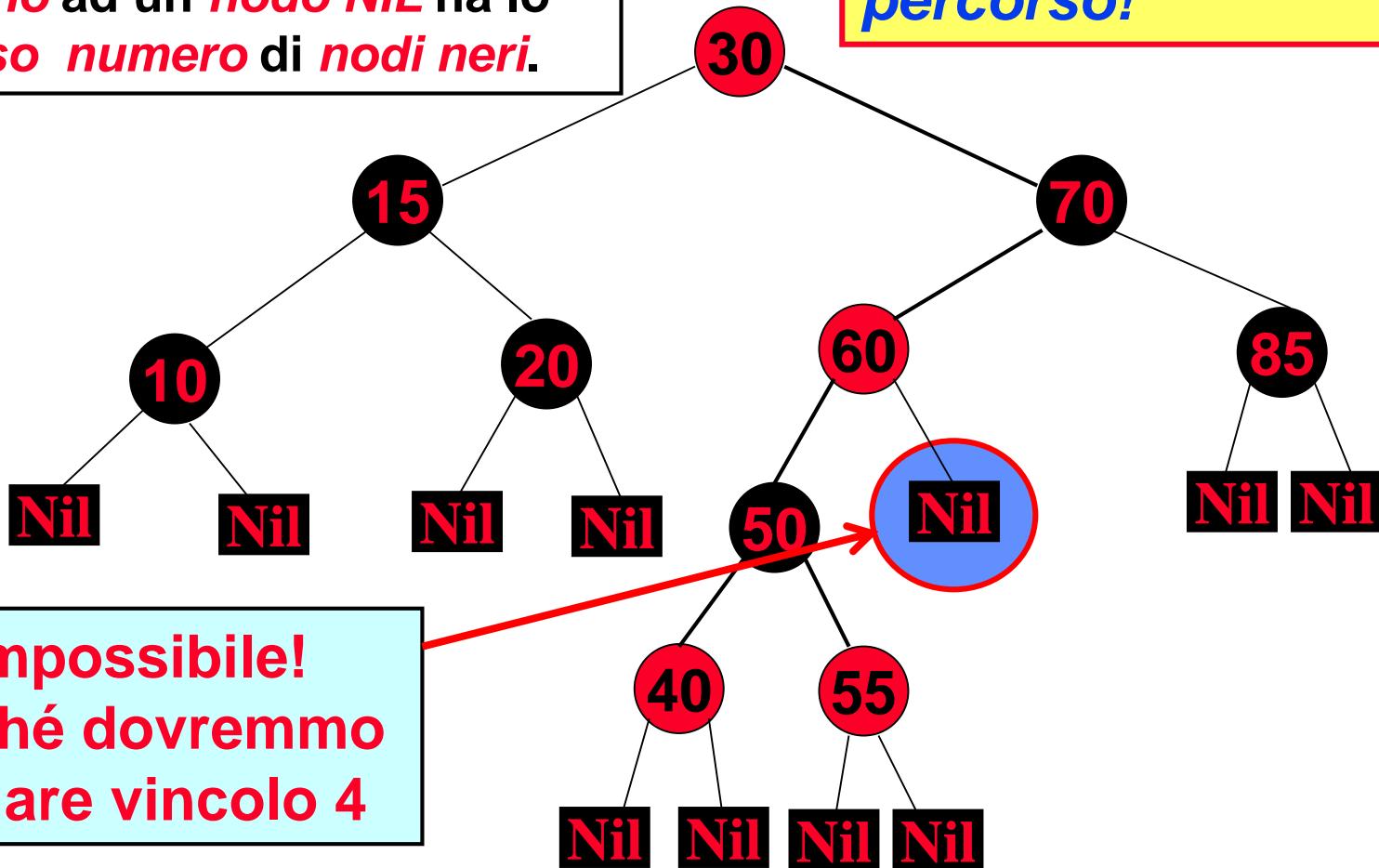
Per *vincolo 4 e il vincolo 3*, ci possono essere al più 2 nodi neri lungo un percorso!



Alberi Red-Black: esempio VI

- 3 Se un *nodo è rosso*, allora entrambi i *suoi figli sono neri*;
4 Ogni percorso da un *nodo interno* ad un *nodo NIL* ha lo stesso *numero di nodi neri*.

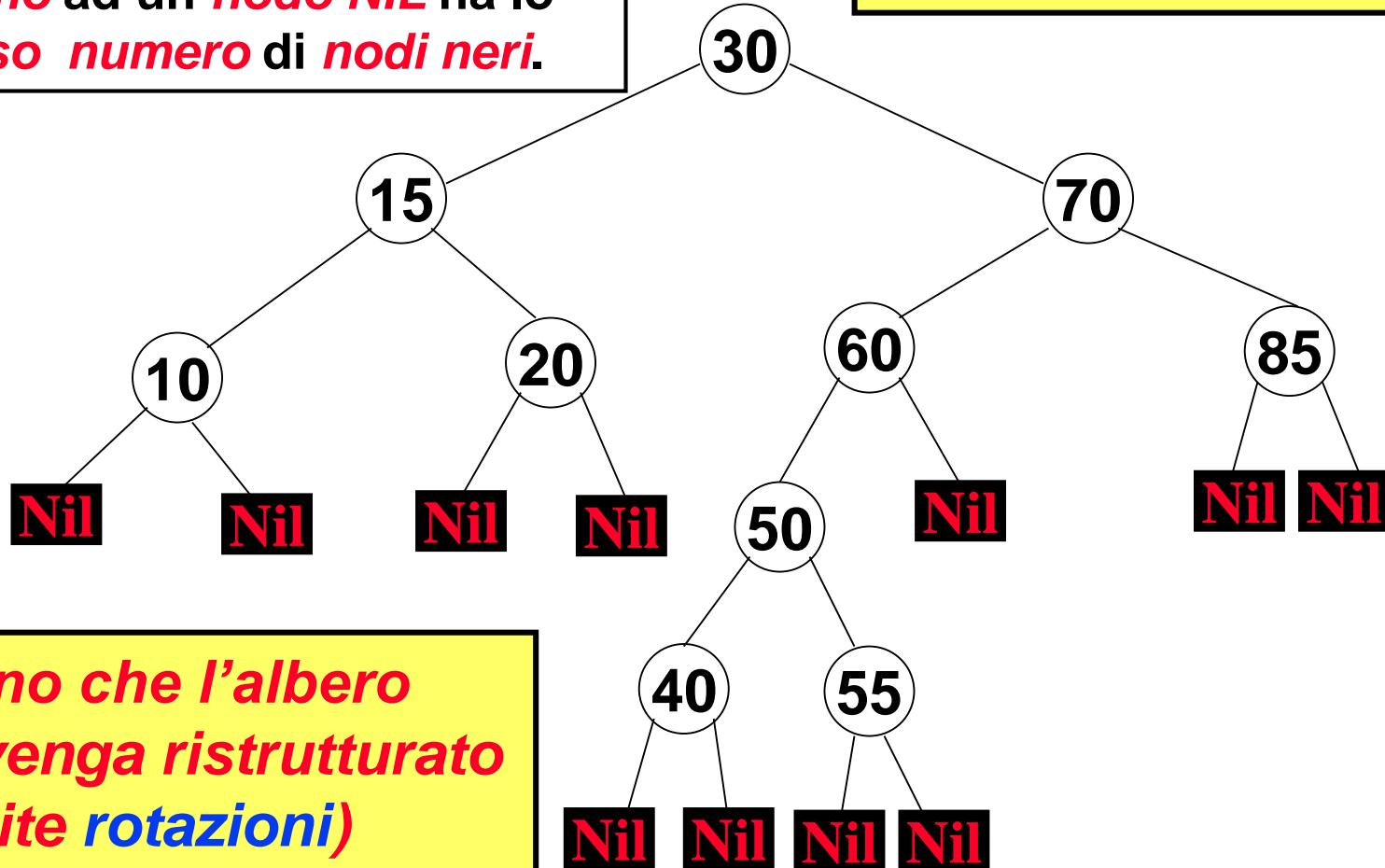
Per *vincolo 4 e il vincolo 3*, ci possono essere al più 2 nodi neri lungo un percorso!



Alberi Red-Black: esempio VI

- 3 Se un *nodo* è *rosso*, allora entrambi i *suoi figli* sono *neri*;
4 Ogni percorso da un *nodo interno* ad un *nodo NIL* ha lo stesso *numero di nodi neri*.

Questo albero *NON* può essere un albero Red-Black!



A meno che l'albero non venga ristrutturato (tramite rotazioni)

Percorso Nero in alberi Red-Black

Definizione: Il *numero* di *nodi neri* lungo ogni percorso da un *nodo x* (escluso) ad una *foglia* (nodo *NIL*) è detto *altezza nera di x*

Definizione: L'*altezza nera di un albero Red-Black* è l'*altezza nera della sua radice*.

Percorso massimo in alberi Red-Black

Lemma: Un *albero Red-Black* con n nodi ha altezza pari al più a:

$$2 \log(n+1)$$

Dimostrazione: Iniziamo col dimostrare per *induzione che per il sottoalbero radicato in x vale*

$$\text{Nodi_Interni}(x) \geq 2^{bh(x)} - 1$$

dove $bh(x)$ è l'*altezza nera dell'albero radicato in x* .

Faremo *induzione sull'altezza del nodo x* .

Percorso massimo in alberi Red-Black

Per il sottoalbero con radice x vale

$Nodi_Interni(x) \geq$

$$2^{bh(x)} - 1$$

dove $bh(x)$ è l'altezza nera di x .

Passo Base:

Se x ha altezza 0: allora x è un nodo NIL e il suo sottoalbero contiene 0 nodi interni.

La sua altezza nera è inoltre $bh(x) = 0$.

Otteniamo quindi:

$$0 \geq 2^{bh(x)} - 1 = 2^0 - 1 = 0$$

Percorso massimo in alberi Red-Black

Per il sottoalbero con radice x vale

Nodi_Interni(x) \geq

$$2^{bh(x)} - 1$$

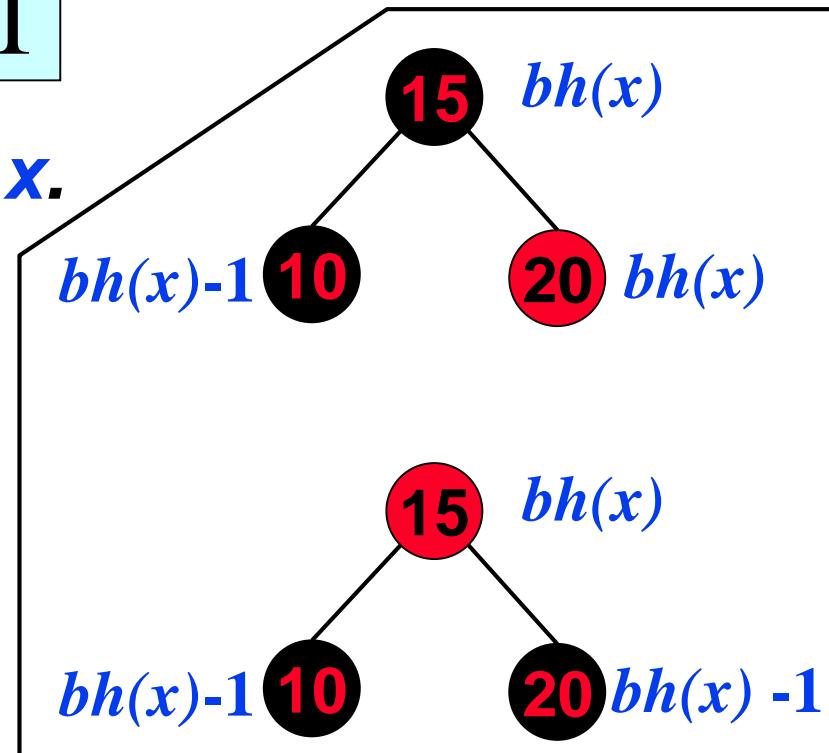
dove $bh(x)$ è l'altezza nera di x .

Passo Induttivo:

Se x sia interno con 2 figli e

abbia altezza $bh(x) > 1$

Entrambi i suoi figli avranno
altezza $bh(x)$ o $bh(x)-1$



Percorso massimo in alberi Red-Black

Per il sottoalbero con radice x vale

Nodi_Interni(x) \geq

$$2^{bh(x)} - 1$$

dove $bh(x)$ è l'altezza nera di x .

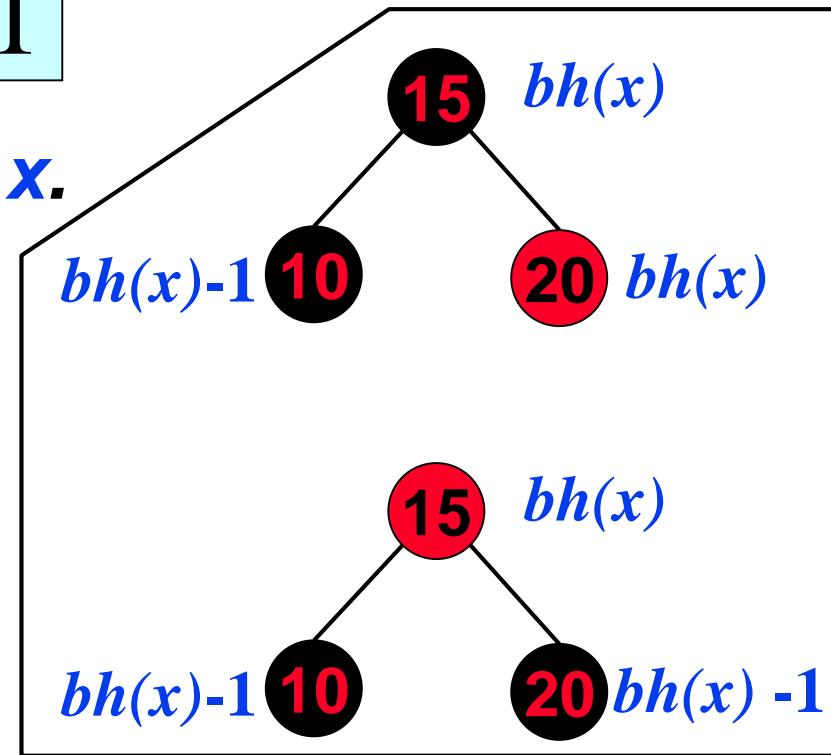
Passo Induttivo:

Sia x nodo interno con 2 figli

e abbia $\text{altezza} > 0$

Entrambi i suoi figli avranno
altezza nera $bh(x)$ o $bh(x)-1$

Entrambi i suoi figli avranno certamente **altezza minore** di x , quindi possiamo usare **ipotesi induttiva**



Percorso massimo in alberi Red-Black

Per il sottoalbero con radice x vale

Nodi_Interni(x) \geq

$$2^{bh(x)} - 1$$

dove $bh(x)$ è l'altezza nera di x .

Passo Induttivo:

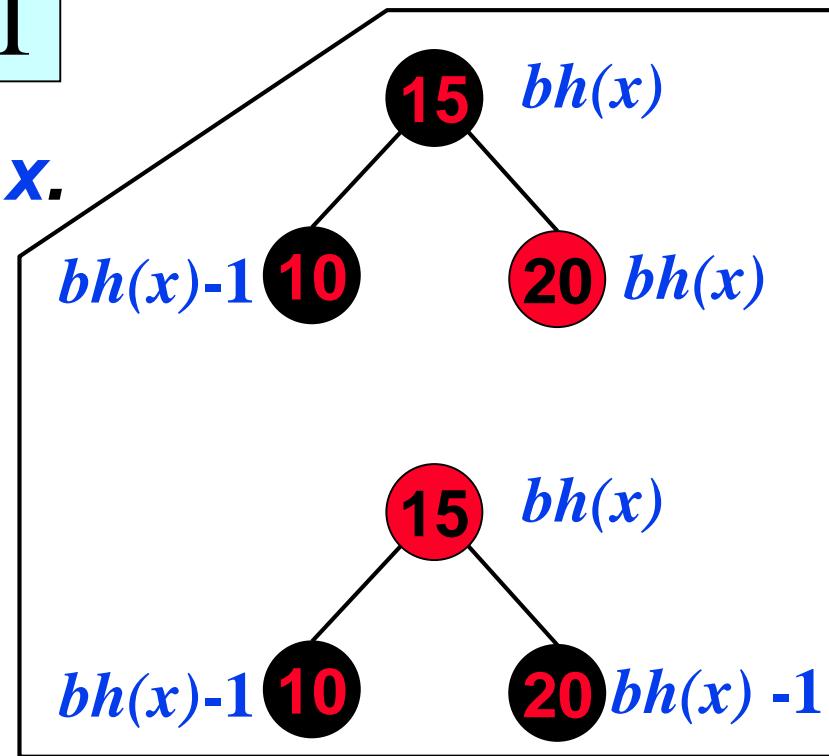
Se x è interno con 2 figli e

ha altezza > 0 allora

entrambi i suoi figli avranno
altezza nera $bh(x)$ o $bh(x)-1$

Ogni figlio, per l'ipotesi induttiva, avrà almeno

$$2^{bh(x)-1} - 1$$



nodi interni

Percorso massimo in alberi Red-Black

Per il sottoalbero con radice x vale

$$\text{Nodi_Interni}(x) \geq 2^{bh(x)} - 1$$

dove $bh(x)$ è l'altezza nera di x .

Passo Induttivo: Sia x nodo interno con 2 figli e abbia altezza >0 . Ogni figlio avrà almeno

$$2^{bh(x)-1} - 1 \quad \text{nodi interni}$$

Quindi il sottoalbero con radice x conterrà almeno

$$(2^{bh(x)-1} - 1) + (2^{bh(x)-1} - 1) + 1 = (2^{bh(x)} - 1) \quad \text{nodi interni}$$

Percorso massimo in alberi Red-Black

Per il sottoalbero con radice x vale

Nodi_Interni(x) \geq

$$2^{bh(x)} - 1$$

dove $bh(x)$ è l'altezza nera di x .

Completiamo la dimostrazione del lemma.

Sia ora h l'altezza dell'albero.

Per il vincolo 3 almeno metà dei nodi lungo ogni percorso (esclusa la radice) sono neri.

Quindi l'altezza nera dell'albero dovrà essere almeno $h/2$ (cioè $bh \geq h/2$).

Percorso massimo in alberi Red-Black

Per il sottoalbero con radice x vale

$Nodi_Interni(x) \geq$

$$2^{bh(x)} - 1$$

dove $bh(x)$ è l'altezza nera di x .

Completiamo la dimostrazione del lemma.

Sia ora h l'altezza dell'albero.

Quindi l'altezza nera dell'albero dovrà essere almeno $h/2$ (cioè $bh \geq h/2$).

Ma allora, il numero di nodi dell'albero sarà

$$n \geq (2^{bh(x)} - 1) \geq 2^{h/2} - 1$$

Percorso massimo in alberi Red-Black

Lemma: Un *albero Red-Black* con n nodi ha altezza al più $2 \log(n+1)$

Dimostrazione:

Completiamo la dimostrazione del lemma.

Sia ora h l'altezza dell'albero.

Ma allora, il numero di nodi dell'albero sarà

$$n \geq (2^{bh(x)} - 1) \geq 2^{h/2} - 1$$

Portando 1 a sinistra e applicando il logaritmo:

$$\log(n+1) \geq h/2$$

cioè

$$h \leq 2 \log(n+1)$$

Costo operazioni su alberi Red-Black

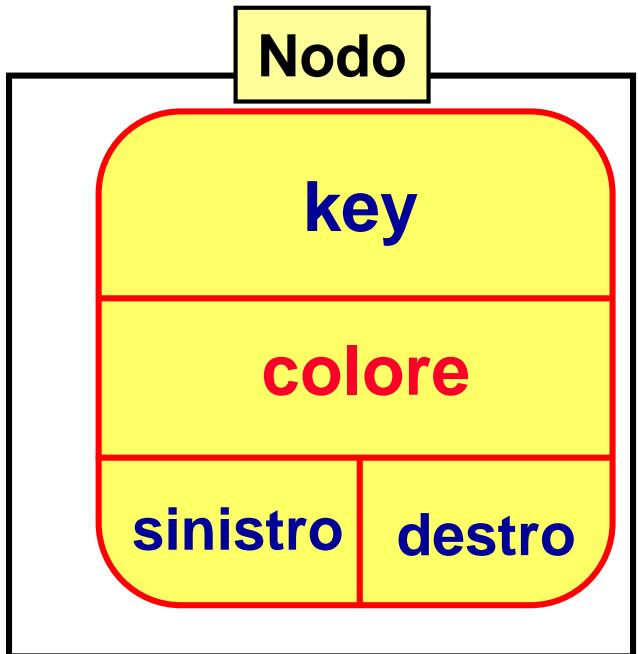
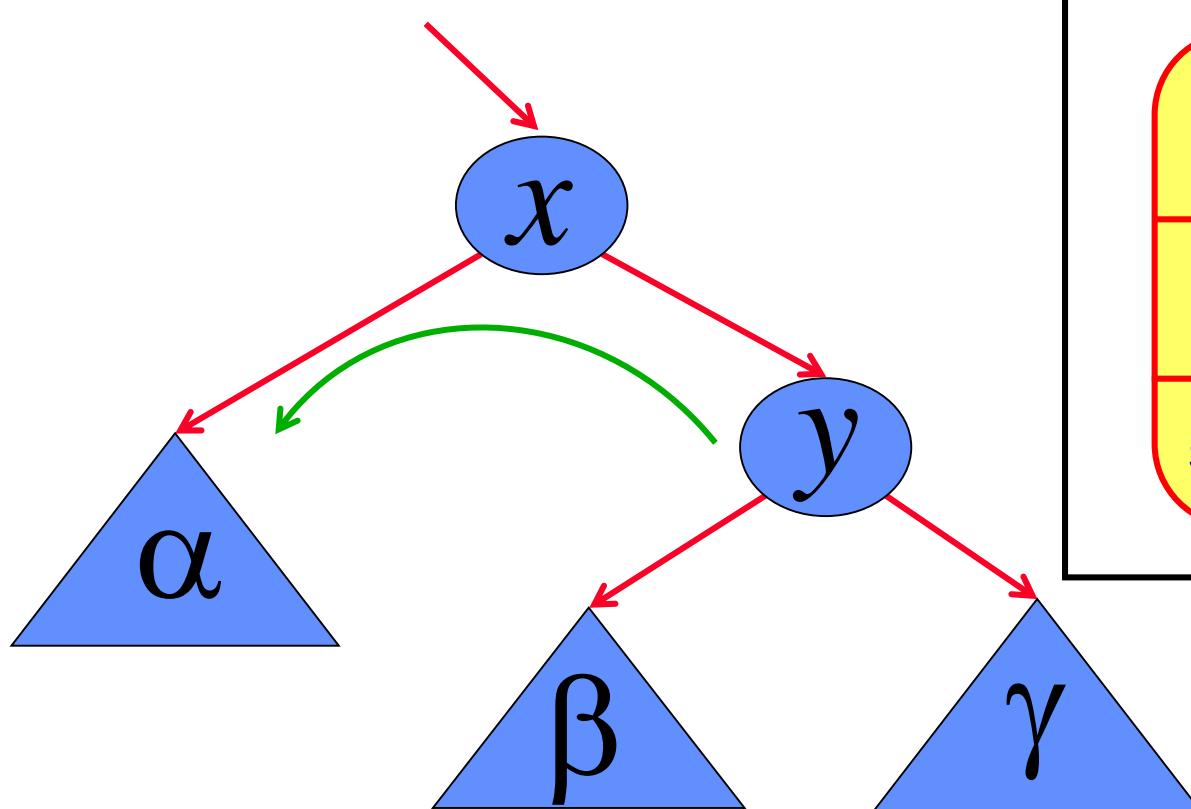
Teorema: In un *albero Red-Black* le operazioni di *ricerca*, *inserimento* e *cancellazione* hanno costo $O(\log(n))$.

Dimostrazione: Conseguenza diretta del *Lemma precedente* e dal fatto che tutte le operazioni hanno costo $O(h)$, con h l'altezza dell'albero.

Violazione delle proprietà per inserimento

- Durante la *costruzione* di un *albero Red-Black*, quando un *nuovo nodo* viene *inserito* è possibile che *le condizioni di bilanciamento* risultino *violate*.
- Lo stesso accade per le *cancellazioni*.
- Quando le *proprietà Red-Black* vengono *violate* si può agire:
 - *modificando i colori nella zona della violazione*;
 - *operando dei ribilanciamenti dell'albero tramite rotazioni: Rotazione destra e Rotazione sinistra*;

Alberi Red-Black: Rotazione destra

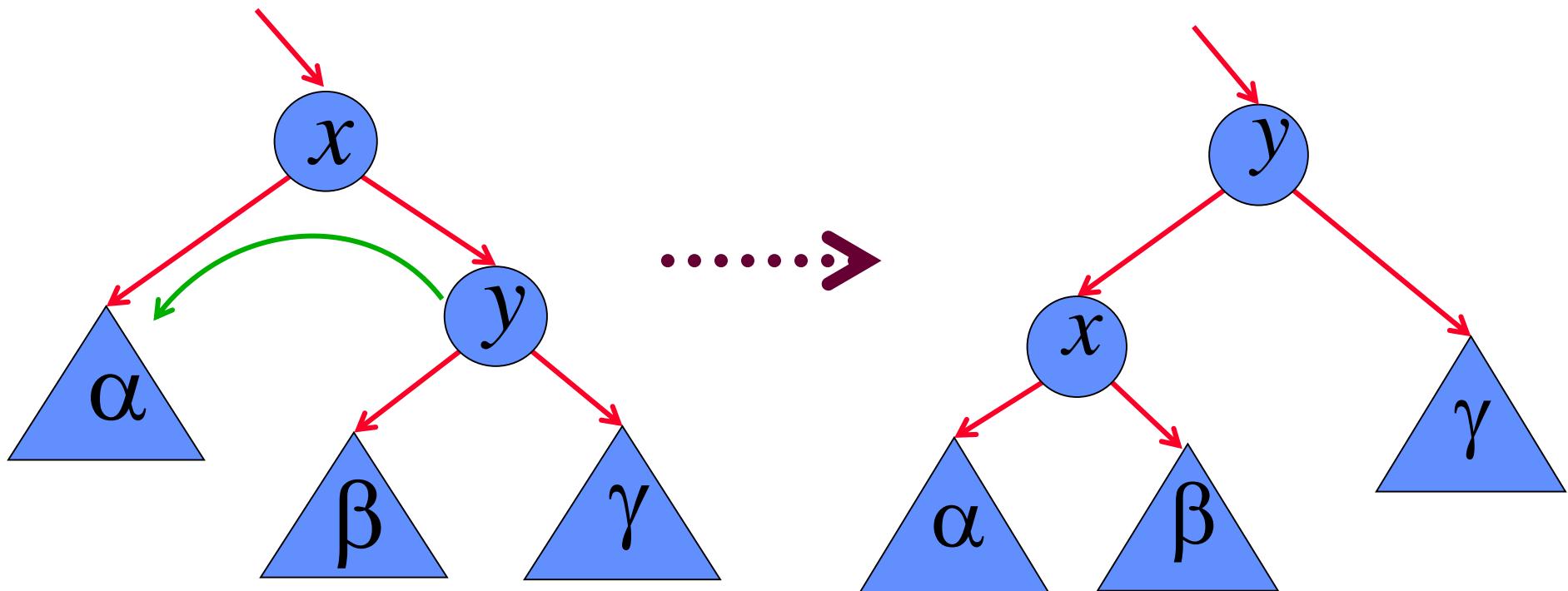


Rotazione col figlio destro

Rotazione va da destra a sinistra

Il Cormen la chiama Rotazione a Sinistra

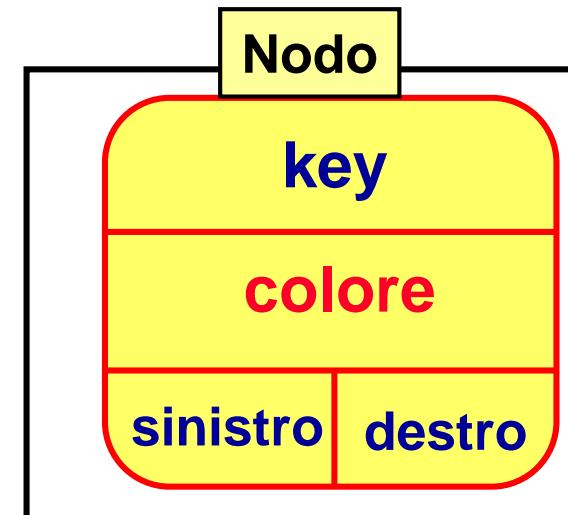
Alberi Red-Black: Rotazione destra



Rotazione col figlio destro

Rotazione va da destra a sinistra

Il Cormen la chiama Rotazione a Sinistra



Alberi Red-Black: Rotazioni

albero-RB Ruota-destro (T :albero-RB)

$fd = T->dx$

$T->sx = fd->sx$

$fd->sx = T$

$return fd$

albero-RB Ruota-sinistro (T :albero-RB)

$fs = T->sx$

$T->sx = fd->dx$

$fd->dx = T$

$return fs$

Alberi Red-Black: Rotazioni

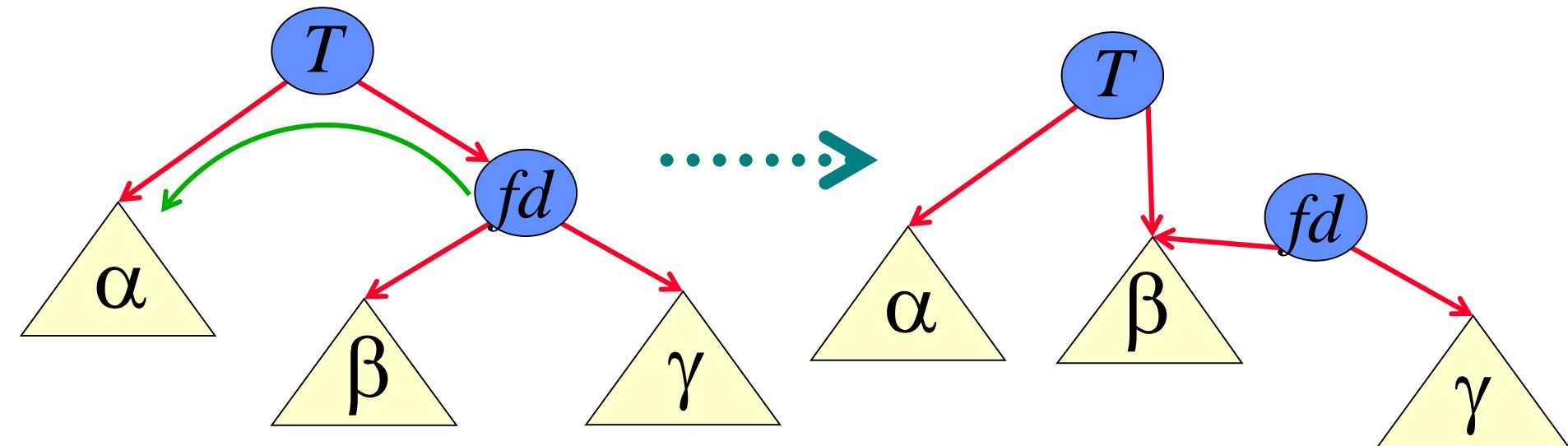
albero-RB Ruota-destro (T : albero-RB)

$fd = T->dx$

$T->dx = fd->sx$

$fd->sx = T$

return fd



Alberi Red-Black: Rotazioni

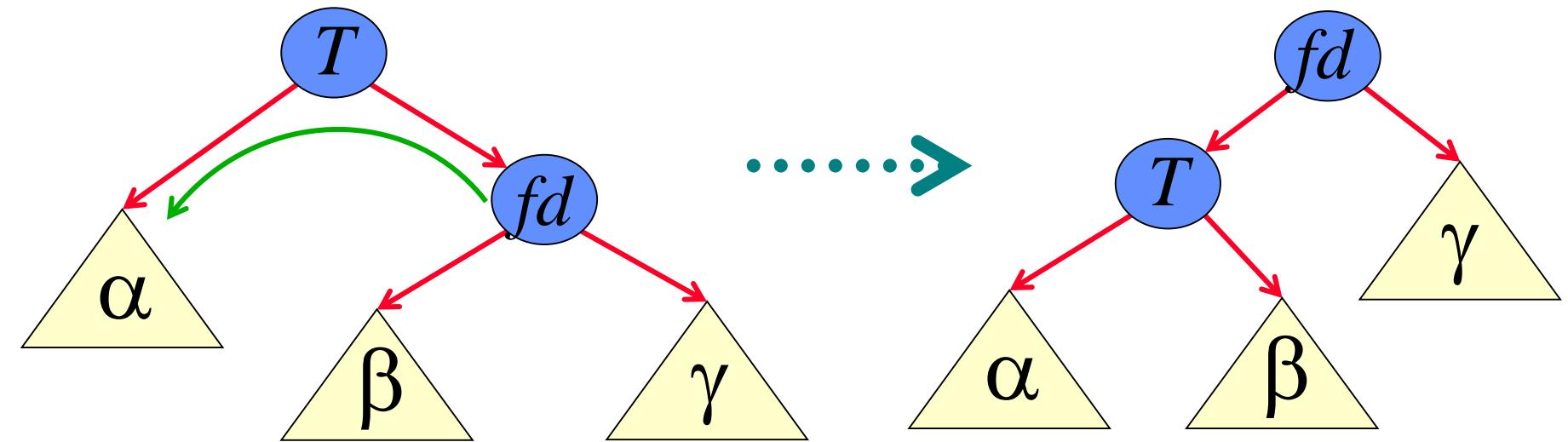
albero-RB Ruota-destro (T : albero-RB)

$fd = T->dx$

$T->dx = fd->sx$

$fd->sx = T$

return fd



Inserimento in alberi Red-Black

- Inseriamo un **nuovo nodo (chiave)** usando la stessa **procedura usata per gli ABR**;
- Coloriamo il nuovo **nodo di rosso** ;
- La ritorno dalle chiamate ricorsive di inserimento, ripristiniamo la proprietà **Red-Black** se è il caso:
 - spostiamo le violazioni verso l'alto rispettando sempre il vincolo 4 (mantenendo l'altezza nera dell'albero);
- Al termine dell'inserimento, coloriamo sempre la “**radice di nero**”.

Le operazioni di ripristino sono necessarie solo quando due nodi consecutivi sono rossi!

Inserimento in alberi Red-Black

albero-RB **INS_RB(*k, T*)**

IF **not IS-NIL(*T*)** THEN

IF ***k < T->key*** THEN

T->sx = INS_RB(*k, T->sx*);

T = Bilancia_sx(*T*);

ELSE /* $k \geq K[T]$ */

T->dx = INS_RB(*k, T->dx*);

T = Bilancia_dx(*T*);

ELSE

T = alloca nodo; ***T->key = k;***

T->dx = NIL[*T*]; ***T->sx = NIL[*T*];***

T->color = red;

return ***T;***

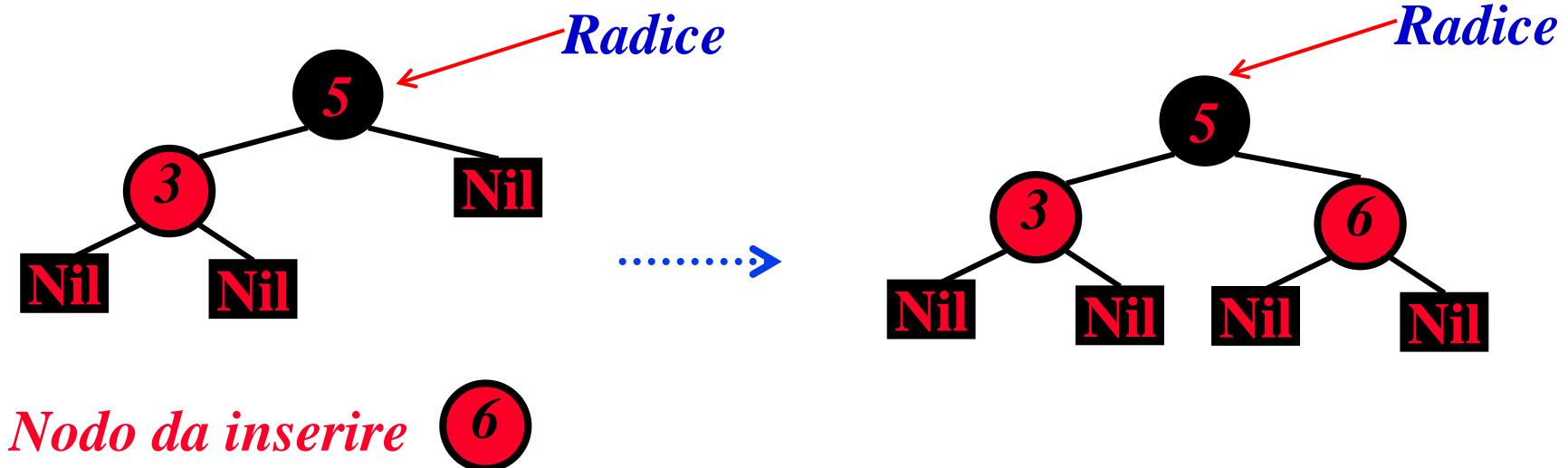
Al termine dell'inserimento, la proc. chiamante deve sempre colorare la “radice di nero”.

Inserimento in alberi Red-Black

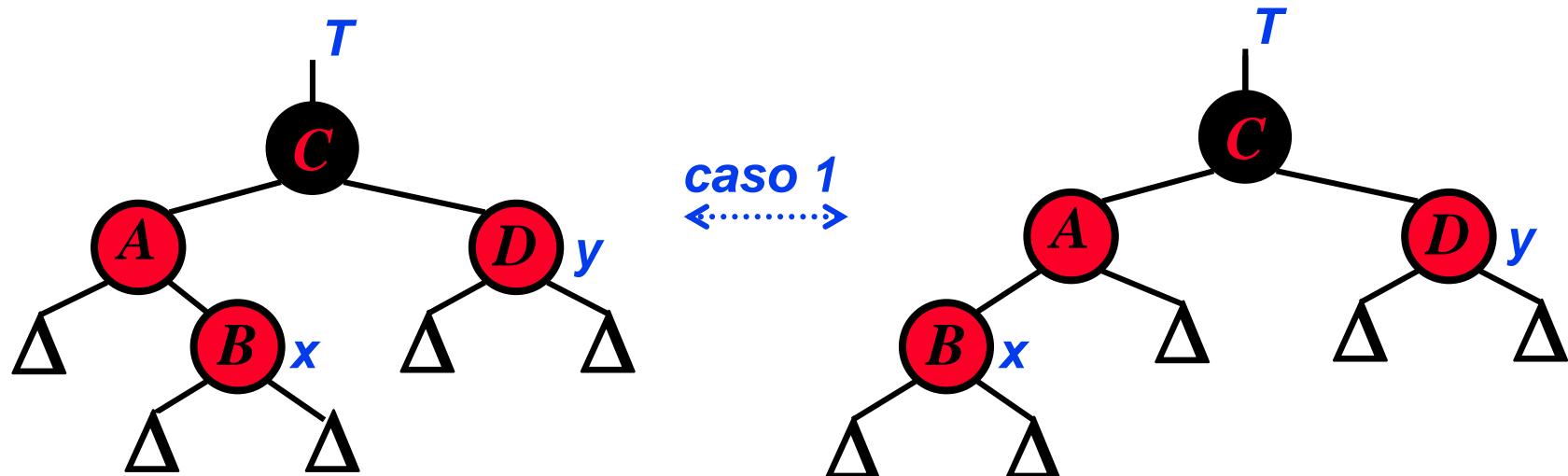
Le operazioni di ripristino sono necessarie solo quando due nodi consecutivi sono rossi!

Se la radice dell'albero è sempre nera, non si presenta mai la necessità di ribilanciare in un albero (o sottoalbero) di altezza minore di 3!

Non si possono, infatti, verificare violazioni!



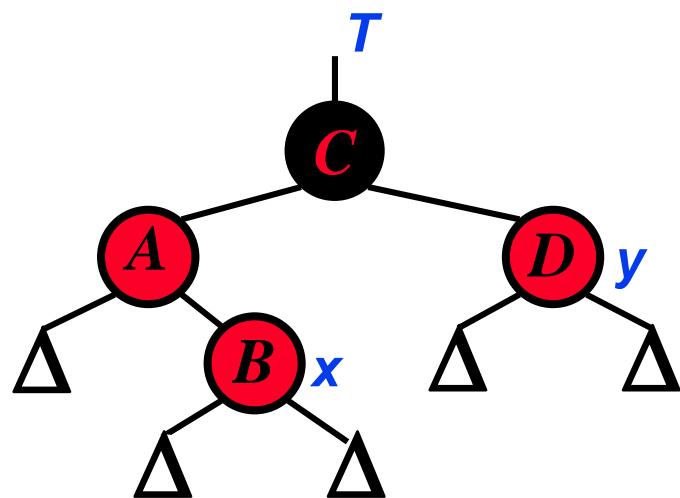
Ribilanciamenti: casi 1-3



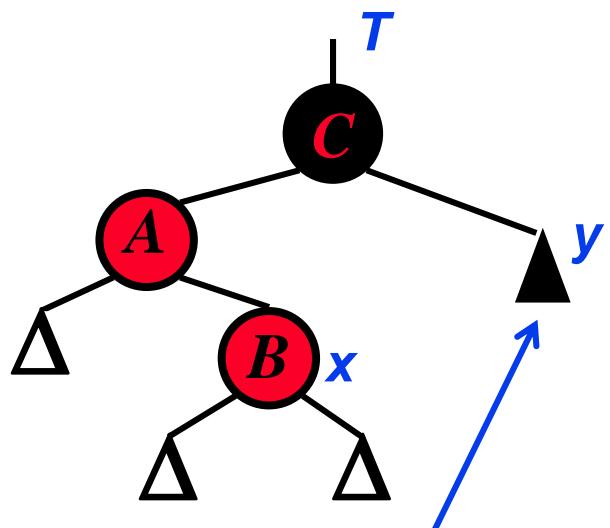
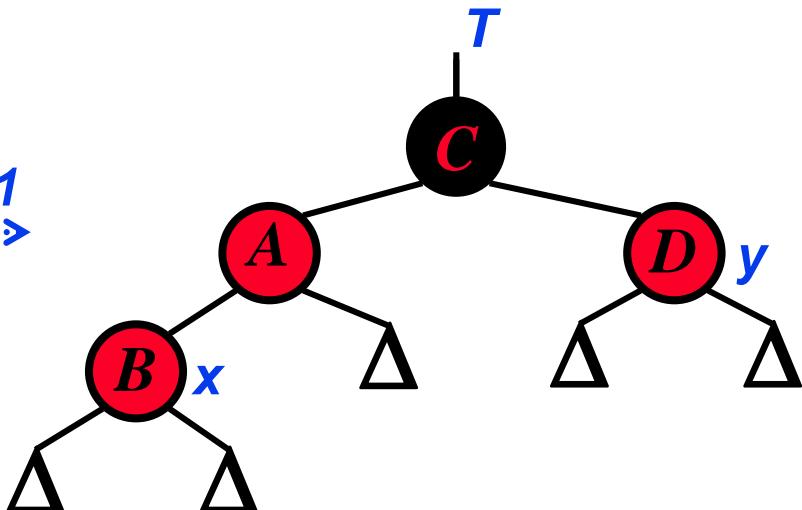
Caso 1: il figlio y di T è rosso

- x è il *nodo modificato* che provoca la **violazione Red-Black**
- y è il figlio destro di T

Ribilanciamenti: casi 1-3



caso 1
↔

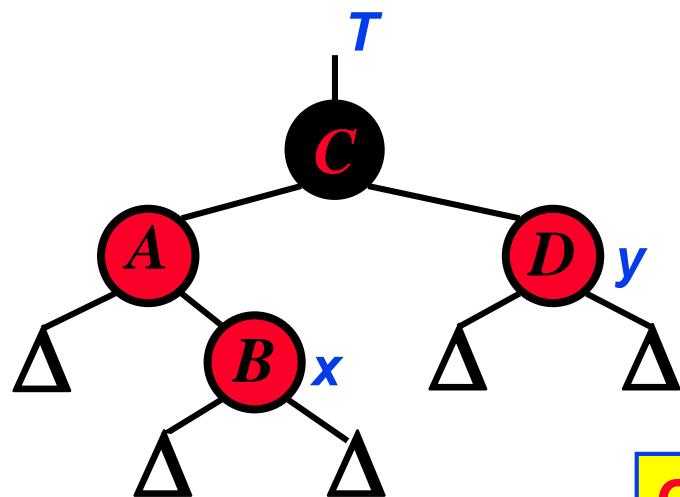


caso 2
↔

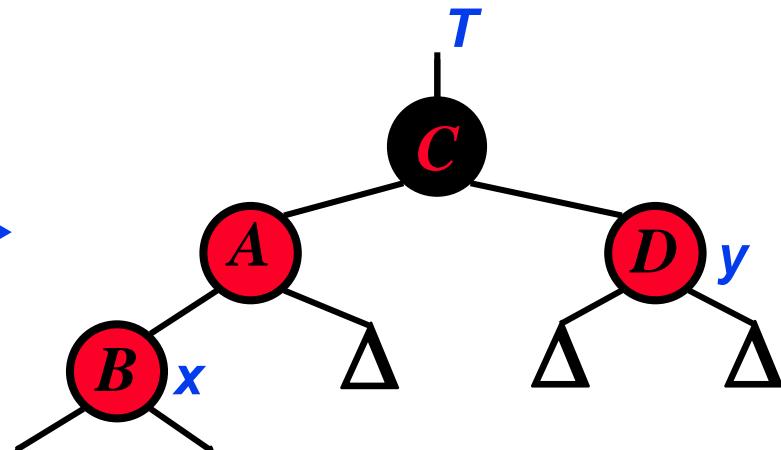
Caso 2: il figlio y di T è nero e il nipote x è un figlio destro

Questa radice
è nera

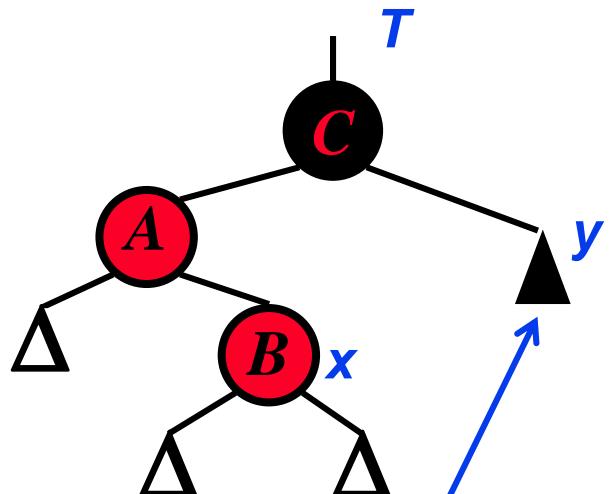
Ribilanciamenti: casi 1-3



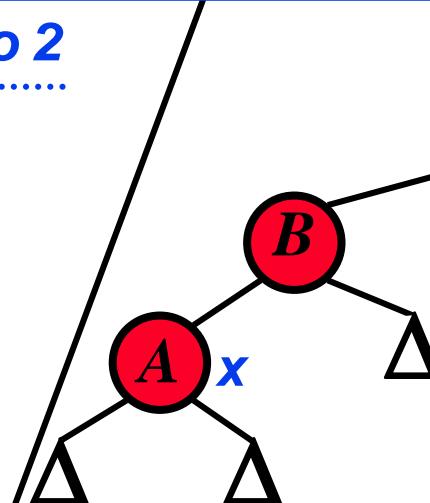
caso 1
↔



Caso 3: il figlio y di T è nero e il nipote x è un figlio sinistro



caso 2
↔



caso 3
↔

La radice di y è nera

La radice di y è nera

Bilanciamento del sottoalbero sinistro

Verifica se l'altezza del sottoalbero sinistro di T è maggiore di 0

albero-RB **Bilancia_sx(T)**

IF ha_figlio($T->sx$)

$v = \text{Tipo_violazione_sx}(T->sx, T->dx)$

/* $v = 0$ nessuna violazione */

CASE v OF

1: $T = \text{Bilancia_sx_1}(T);$

2: $T = \text{Bilancia_sx_2}(T);$

$T = \text{Bilancia_sx_3}(T);$

3: $T = \text{Bilancia_sx_3}(T);$

return $T;$

Calcolo del tipo di violazione

```
int Tipo_violazione_sx(s,d)
```

```
    violazione = 0
```

```
    IF s->color = red AND d->color = red THEN
```

```
        IF s->sx->color = red OR s->dx->color = red THEN
```

```
            violazione = 1;
```

```
    ELSE /* d->color = black o s->color = black */
```

```
        IF s->color = red THEN
```

```
            IF s->dx->color = red
```

```
                violazione = 2;
```

```
        ELSE
```

```
            IF s->sx->color = red
```

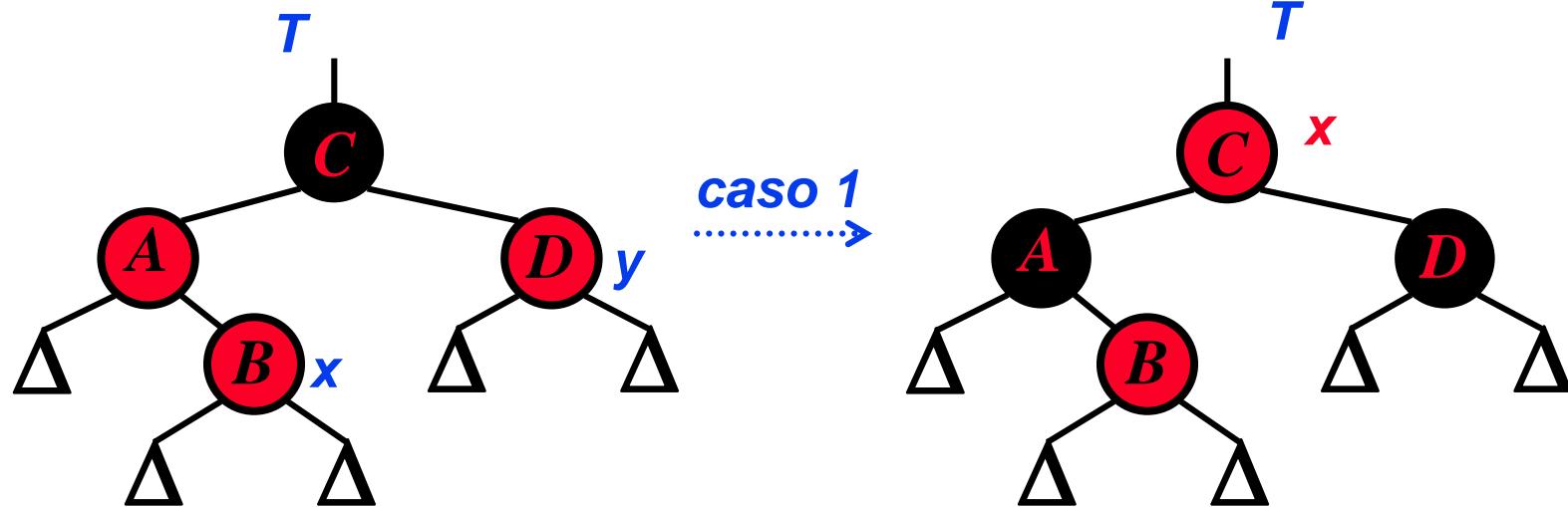
```
                violazione = 3;
```

```
return violazione;
```

Inserimento in alberi RB: Caso 1

Caso 1: il figlio y di T è rosso

Tutti i Δ sono sottoalberi di uguale altezza nera



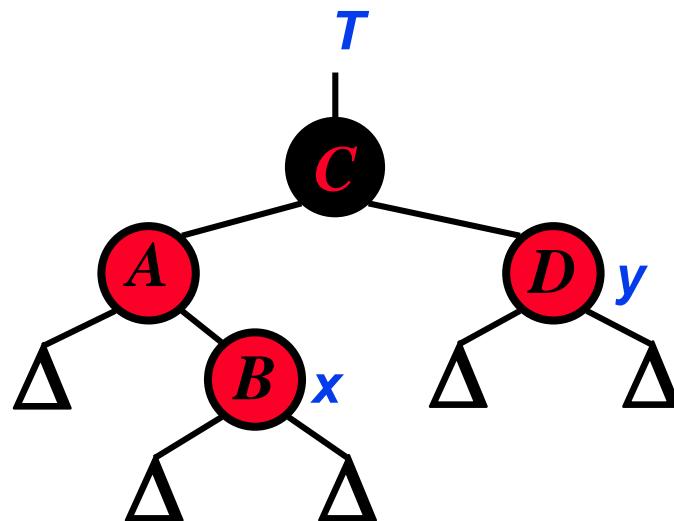
Cambiamo i colori di alcuni nodi, preservando vincolo 4: tutti i percorsi sotto a questi nodi hanno altezza nera uguale.

Inserimento in alberi RB: Caso 1

```
T->dx->color = black;  
T->sx->color = black;  
T->color = red;  
return T;
```

Caso 1: il figlio y del di T è rosso

Tutti i Δ sono sottoalberi di uguale altezza nera



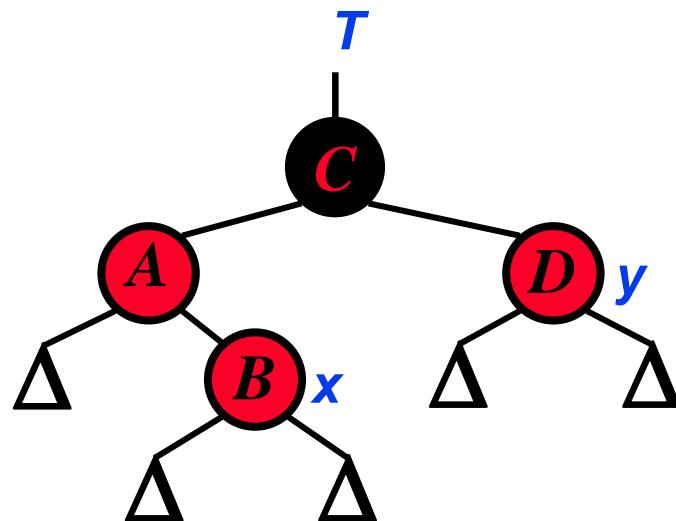
Cambiamo i colori di alcuni nodi, preservando vincolo 4: tutti i percorsi sotto a questi nodi hanno altezza nera uguale.

Inserimento in alberi RB: Caso 1

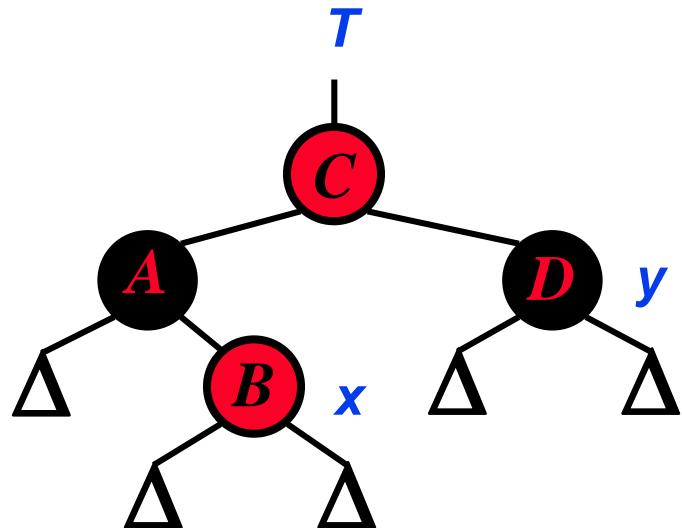
```
T->dx->color = black;  
T->sx->color = black;  
T->color = red;  
return T;
```

Caso 1: il figlio y del di T è rosso

Tutti i Δ sono sottoalberi di uguale altezza nera



caso 1
.....
→

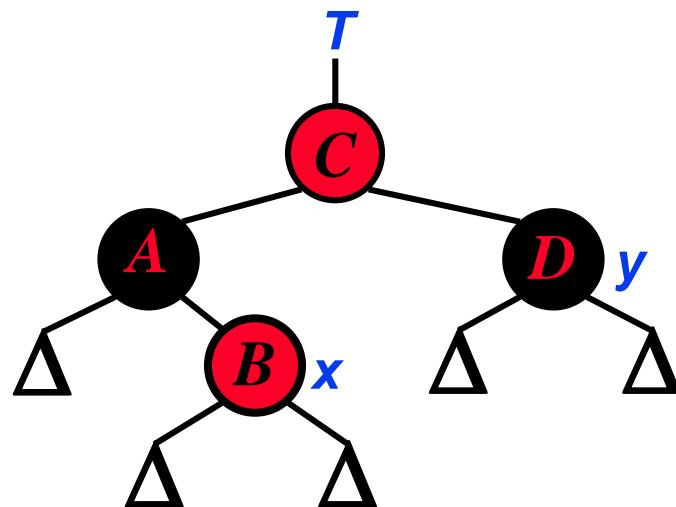


Cambiamo i colori di alcuni nodi, preservando vincolo 4:
tutti i percorsi sotto a questi nodi hanno altezza nera uguale.

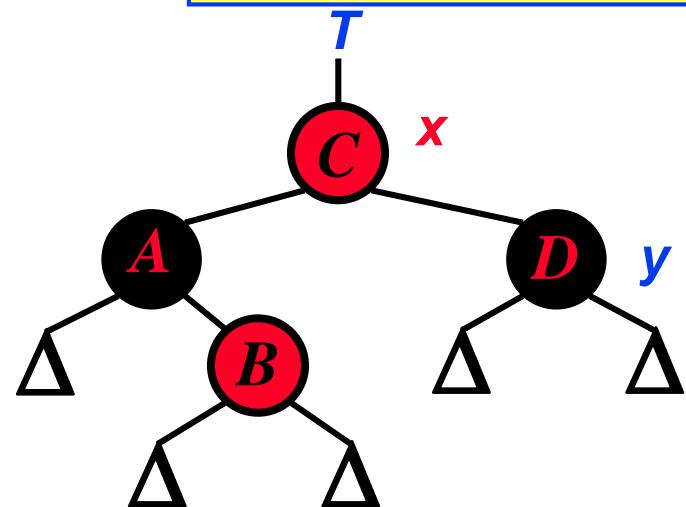
Inserimento in alberi RB: Caso 1

```
T->dx->color=black;  
T->sx->color=black;  
T->color=red;  
return T;
```

Poiché il padre di C può essere rosso, può essere necessario ripristinare la proprietà RB più in alto



caso 1



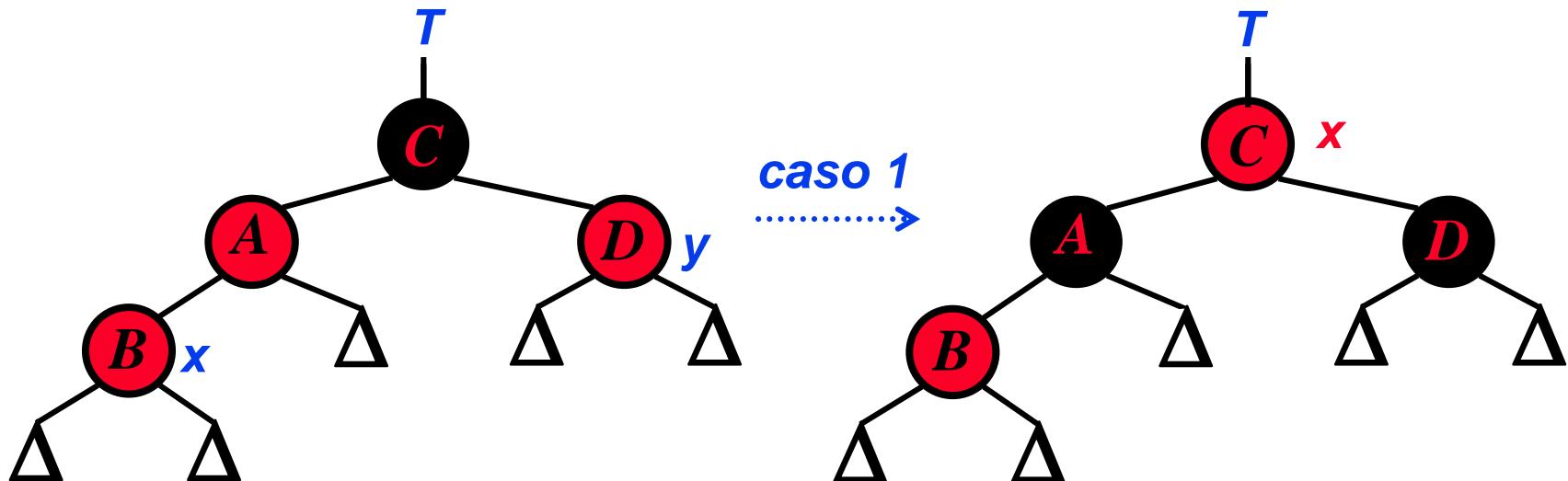
Cambiamo i colori di alcuni nodi, preservando vincolo 4: tutti i percorsi sotto a questi nodi hanno altezza nera uguale.

Inserimento in alberi RB: Caso 1

```
T->dx->color=black;  
T->sx->color=black;  
T->color=red;  
return T;
```

Caso 1: il figlio **y** del padre del padre di **x** è rosso

Tutti i Δ sono sottoalberi di uguale altezza nera

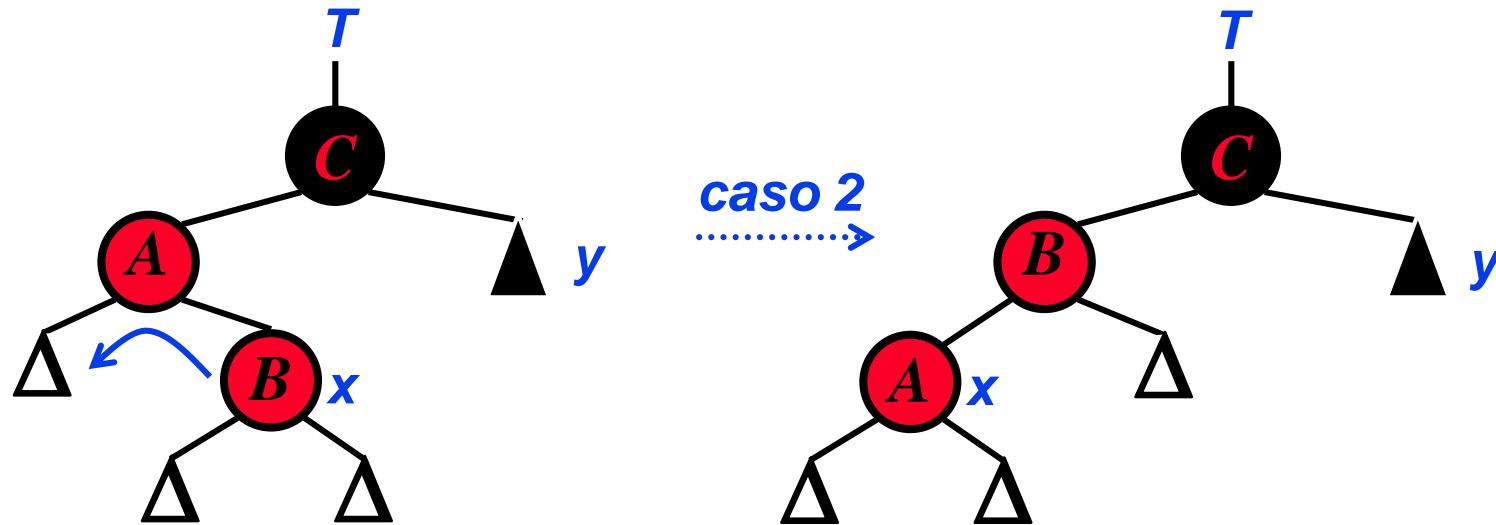


Nel Caso 1, si eseguono le stesse azioni sia quando **x** è **figlio sinistro** o **figlio destro**.

Inserimento in alberi RB: Caso 2

Caso 2:

- il figlio y di T è nero
- x è un figlio destro
- Trasformiamo nel caso 3 con rotazione destra



Trasformiamo il Caso 2 nel Caso 3 (x è figlio sinistro) con una rotazione destra. Ciò preserva il vincolo 4: tutti i percorsi sotto x contengono lo stesso numero di nodi neri

Inserimento in alberi RB: Caso 2

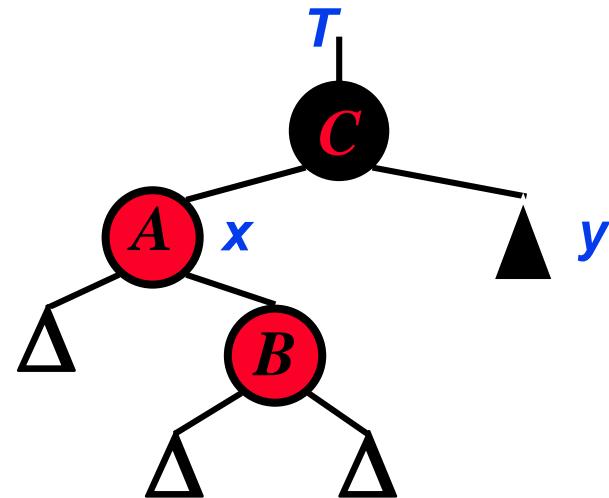
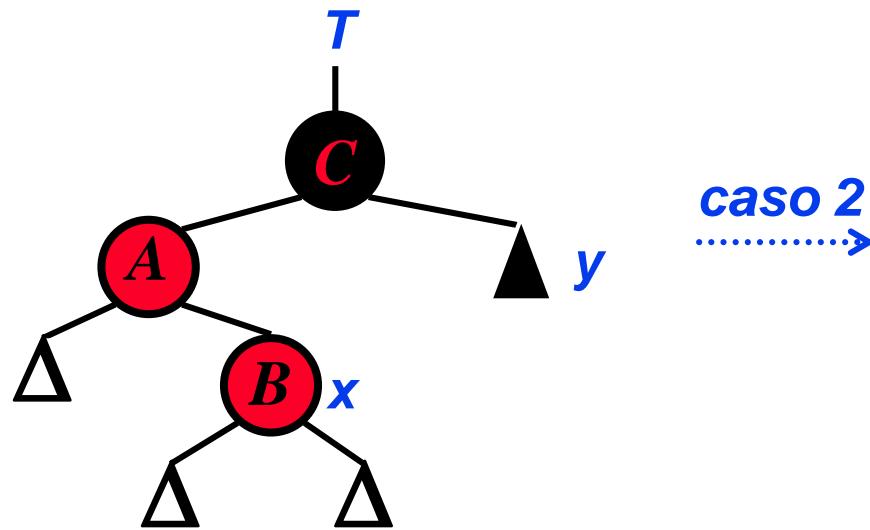
$T \rightarrow \text{sx} = \text{Ruota-dx}(T \rightarrow \text{sx})$

return T

// continua con caso 3

Caso 2:

- il figlio y di T è nero
- x è un figlio destro
- Trasformiamo nel caso 3 con rotazione destra



Trasformiamo il Caso 2 nel Caso 3 (x è figlio sinistro) con una rotazione destra. Ciò preserva il vincolo 4: tutti i percorsi sotto x contengono lo stesso numero di nodi neri

Inserimento in alberi RB: Caso 2

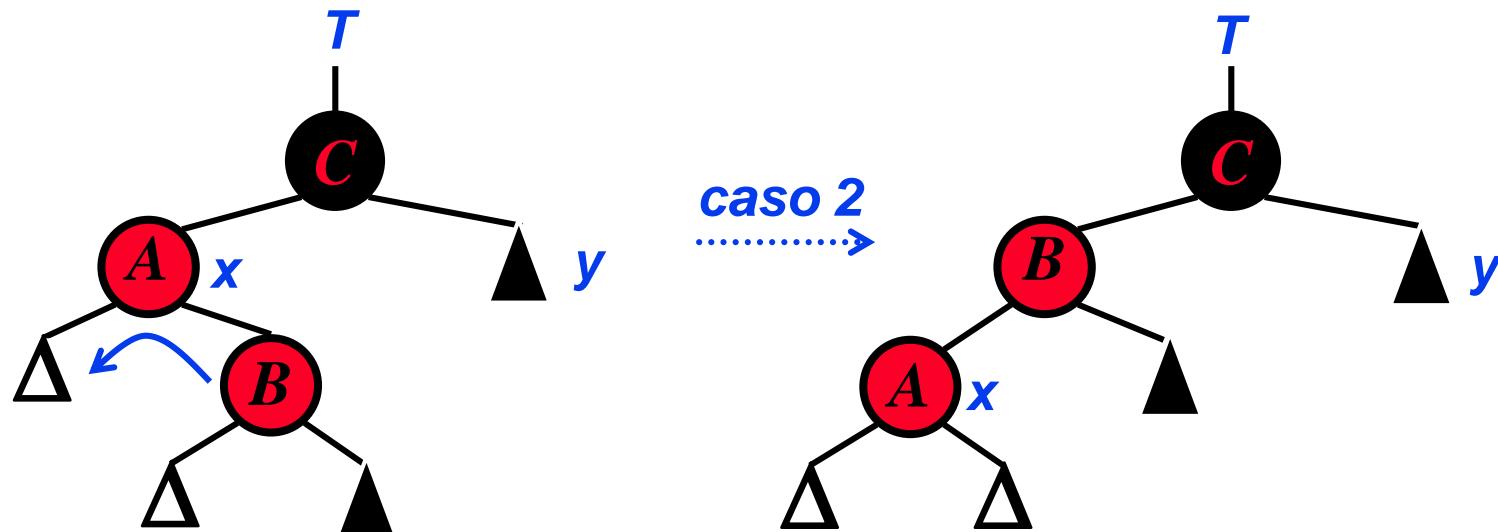
$T \rightarrow \text{sx} = \text{Ruota-dx}(T \rightarrow \text{sx})$

return T

// continua con caso 3

Caso 2:

- il figlio y di T è nero
- x è un figlio destro
- Trasformiamo nel caso 3 con rotazione destra

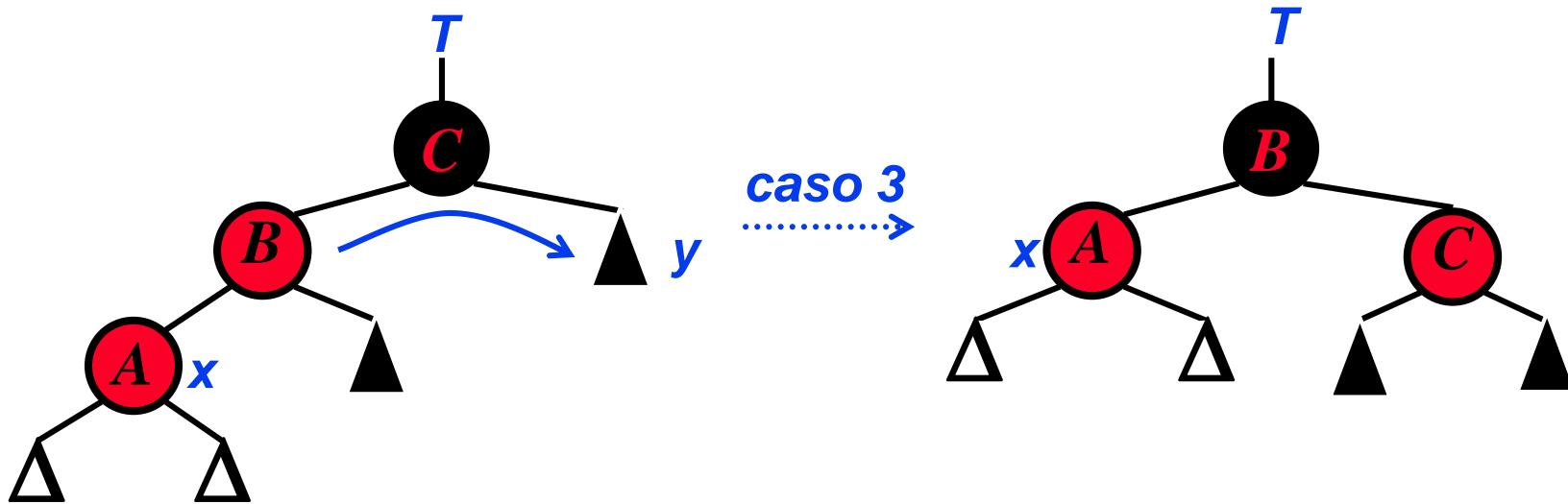


Trasformiamo il Caso 2 nel Caso 3 (x è figlio sinistro) con una rotazione destra. Ciò preserva il vincolo 4: tutti i percorsi sotto x contengono lo stesso numero di nodi neri

Inserimento in alberi RB: Caso 3

Caso 3:

- il figlio y di T è nero
- x è un figlio sinistro
- Cambiare colori e rotazione sinistra



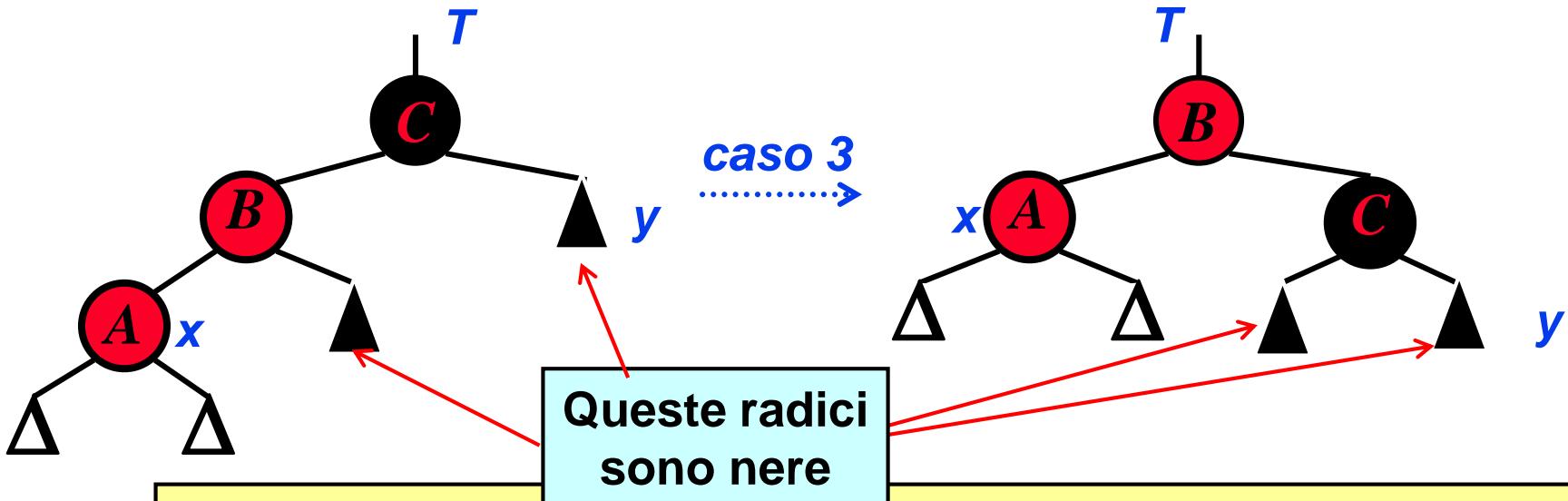
Eseguiamo alcuni **cambi di colore** e facciamo **una rotazione sinistra**. Di nuovo, preserviamo il vincolo 4: tutti i percorsi sotto x contengono lo stesso numero di nodi neri.

Inserimento in alberi RB: Caso 3

```
T = Ruota_sx(T);  
T->color = black;  
T->dx->color = red;  
return T;
```

Caso 3:

- il figlio y del padre del padre di x è nero
- x è un figlio sinistro
- Cambiare colori e rotazione sinistra



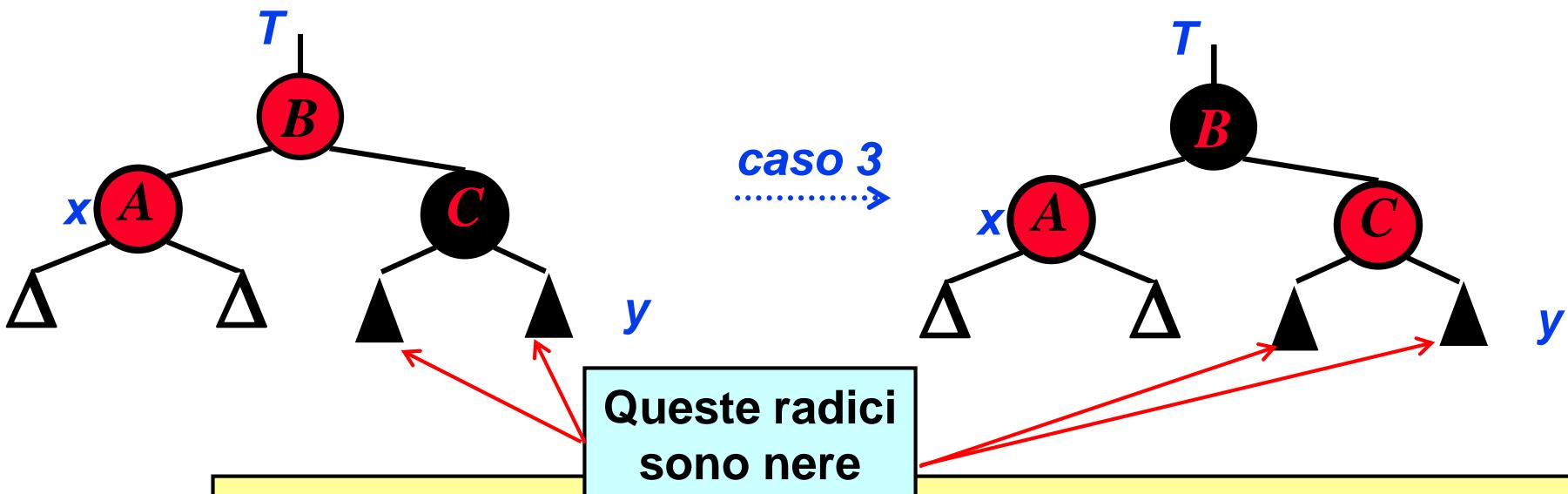
Eseguiamo alcuni cambi di colore e facciamo una rotazione sinistra. Di nuovo, preserviamo il vincolo 4: tutti i percorsi sotto x contengono lo stesso numero di nodi neri.

Inserimento in alberi RB: Caso 3

```
T = Ruota_sx(T)  
T->color = black;  
T->dx->color = red;  
return T
```

Caso 3:

- il figlio y del padre del padre di x è nero
- x è un figlio sinistro
- Cambiare colori e rotazione sinistra



Eseguiamo alcuni cambi di colore e facciamo una rotazione sinistra. Di nuovo, preserviamo il vincolo 4: tutti i percorsi sotto x contengono lo stesso numero di nodi neri.

Bilanciamento in alberi Red-Black

albero-RB Bilancia_sx_1(T)

$T->color = red;$

$T->dx->color = black;$

$T->sx->color = black;$

return T ;

albero-RB Bilancia_sx_2(T)

$T->sx = Ruota_dx(T->sx);$

return T ;

albero-RB Bilancia_sx_3(T)

$T = Ruota_sx(T)$

$T->color = black;$

$T->dx->color = red;$

return T ;

Bilanciamento del sottoalbero sinistro

albero-RB **Bilancia_sx(T)**

IF **ha_figlio(T ->sx)**

v = Tipo_violazione_sx(T ->sx, T ->dx)

/* v = 0 nessuna violazione */

CASE **v** OF

1: **T = Bilancia_sx_1(T);**

2: **T = Bilancia_sx_2(T);**

T = Bilancia_sx_3(T);

3: **T = Bilancia_sx_3(T);**

return T ;

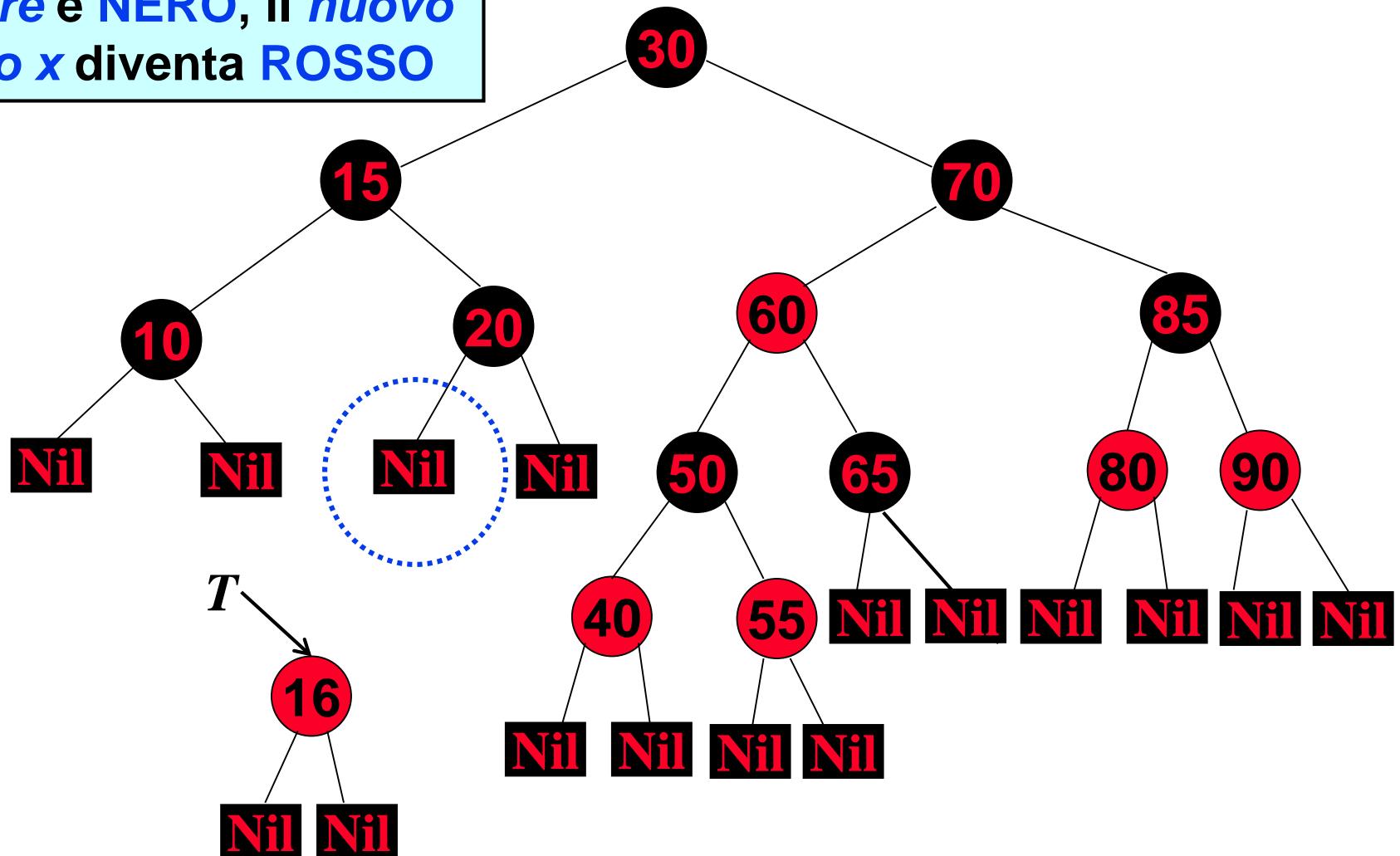
Inserimento in alberi RB: Casi 4-6

- I casi 1-3 assumono che il **figlio** di T, che **viola** con un **nipote** di T, sia un **figlio sinistro**.
- Se il **figlio** di T, che **viola** con **nipote** di T, è un **figlio destro** si applicano i **casi 4-6**, che sono simmetrici (dobbiamo solo scambiare **sinistro** con **destro** e viceversa).

L'estensione dell'algoritmo ai **casi 4-6** è lasciato come **esercizio**

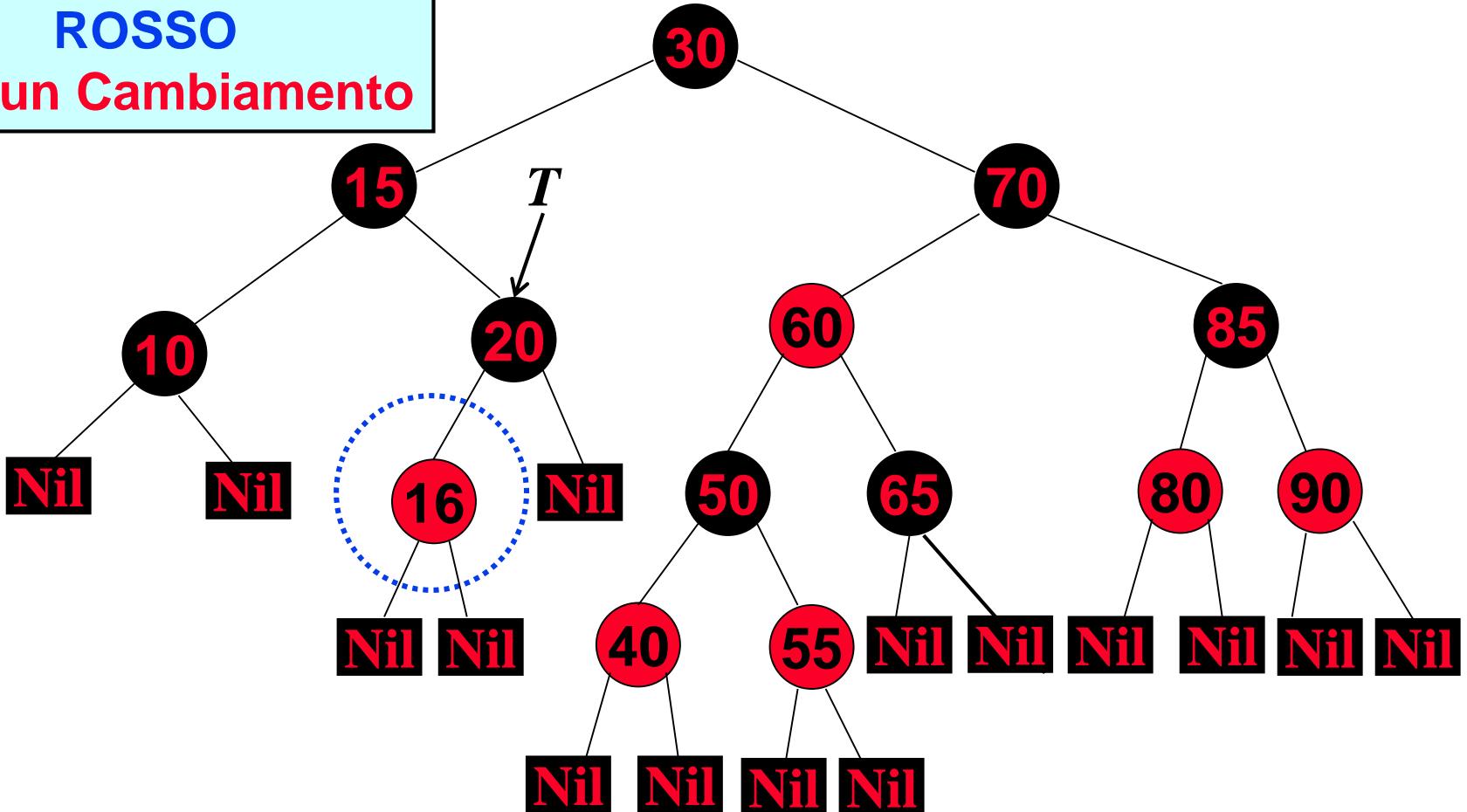
Inserimento in alberi Red-Black: I

Il *padre* è NERO, il *nuovo nodo x* diventa ROSSO



Inserimento in alberi Red-Black: I

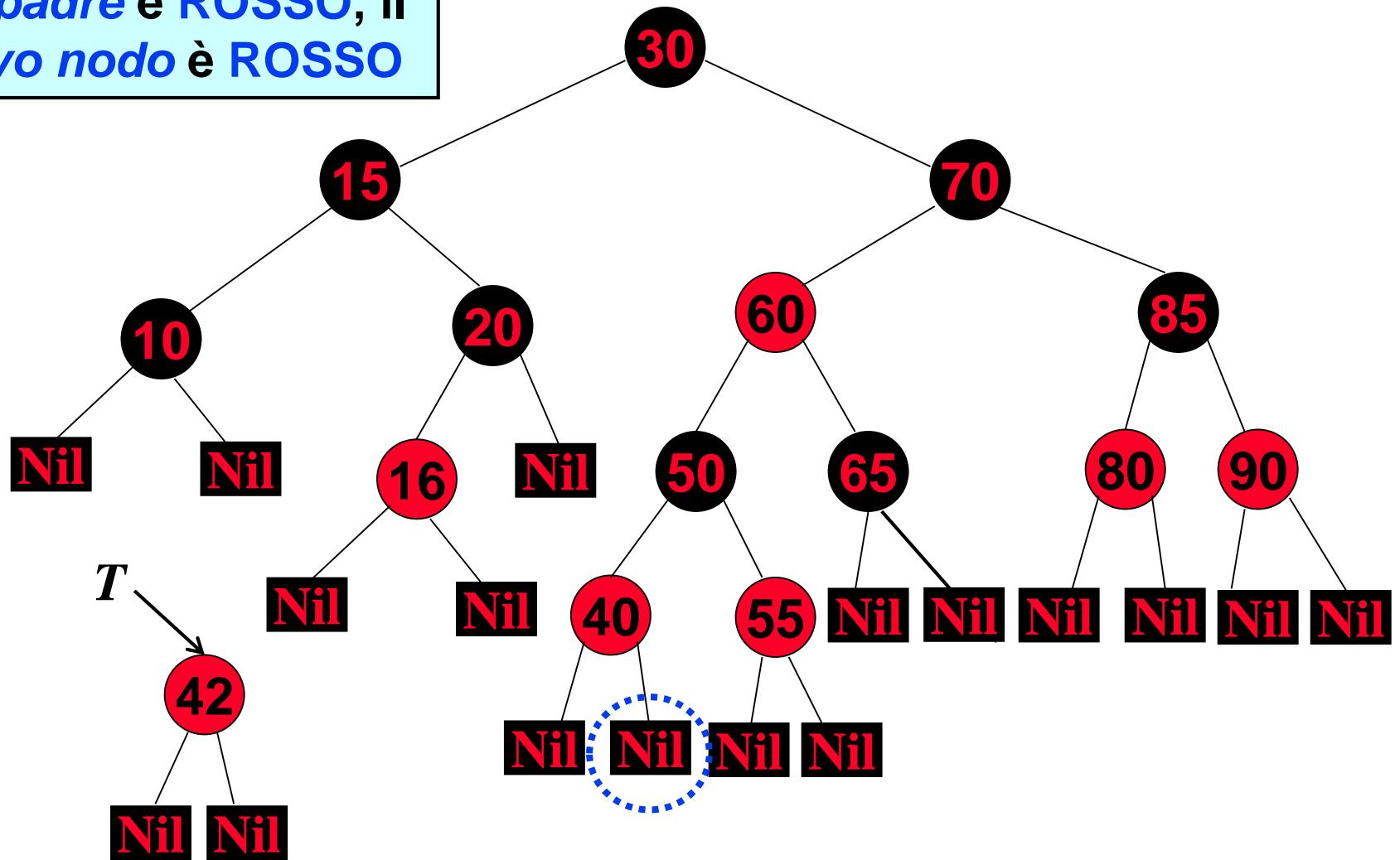
Il T è NERO, il *nodo inserito* diventa ROSSO
Nessun Cambiamento



Non cambia l'altezza nera
di nessun nodo!

Inserimento in alberi Red-Black: II

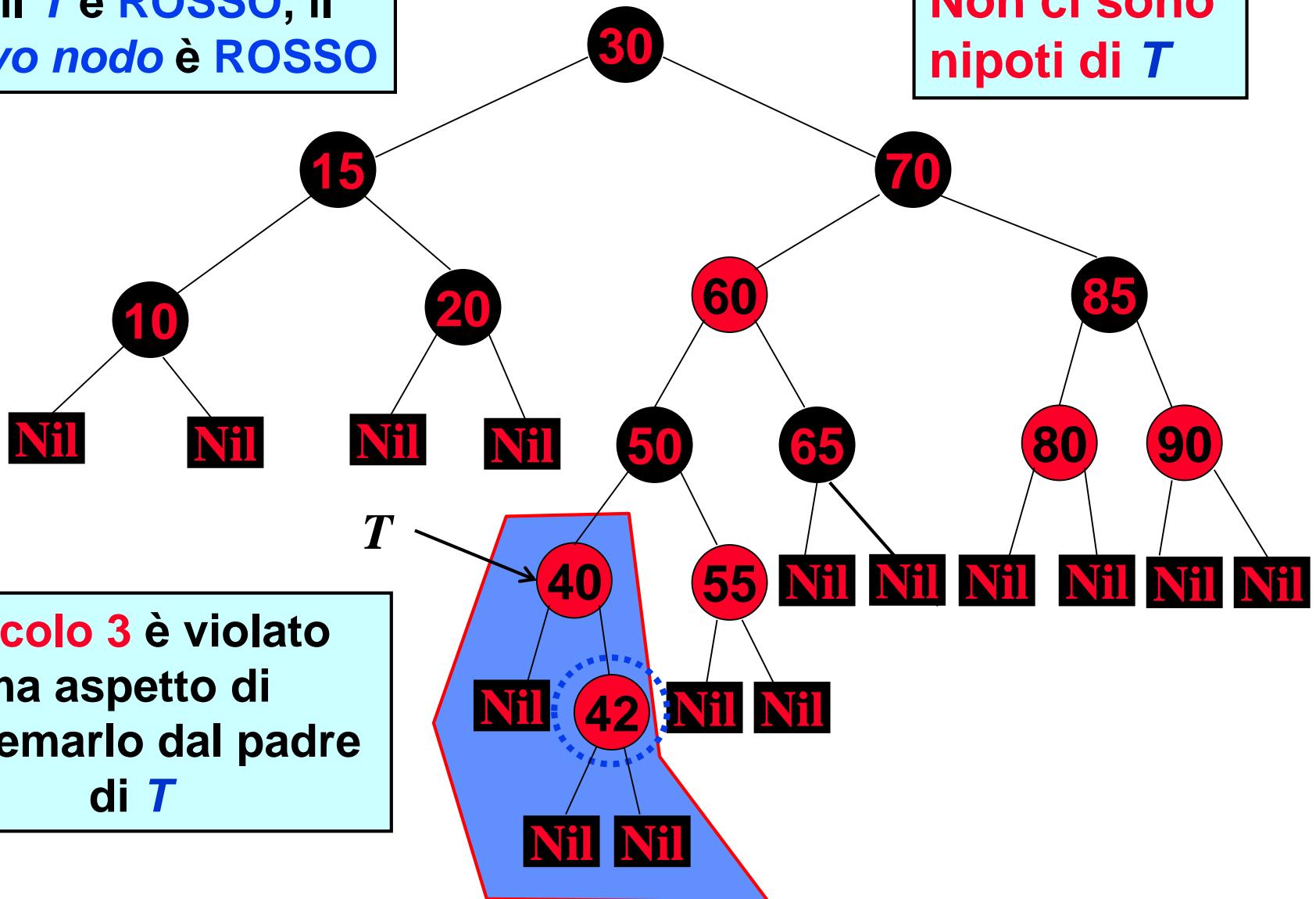
Se il *padre* è ROSSO, il
nuovo nodo è ROSSO



Inserimento in alberi Red-Black: II

Se il T è ROSSO, il
nuovo nodo è ROSSO

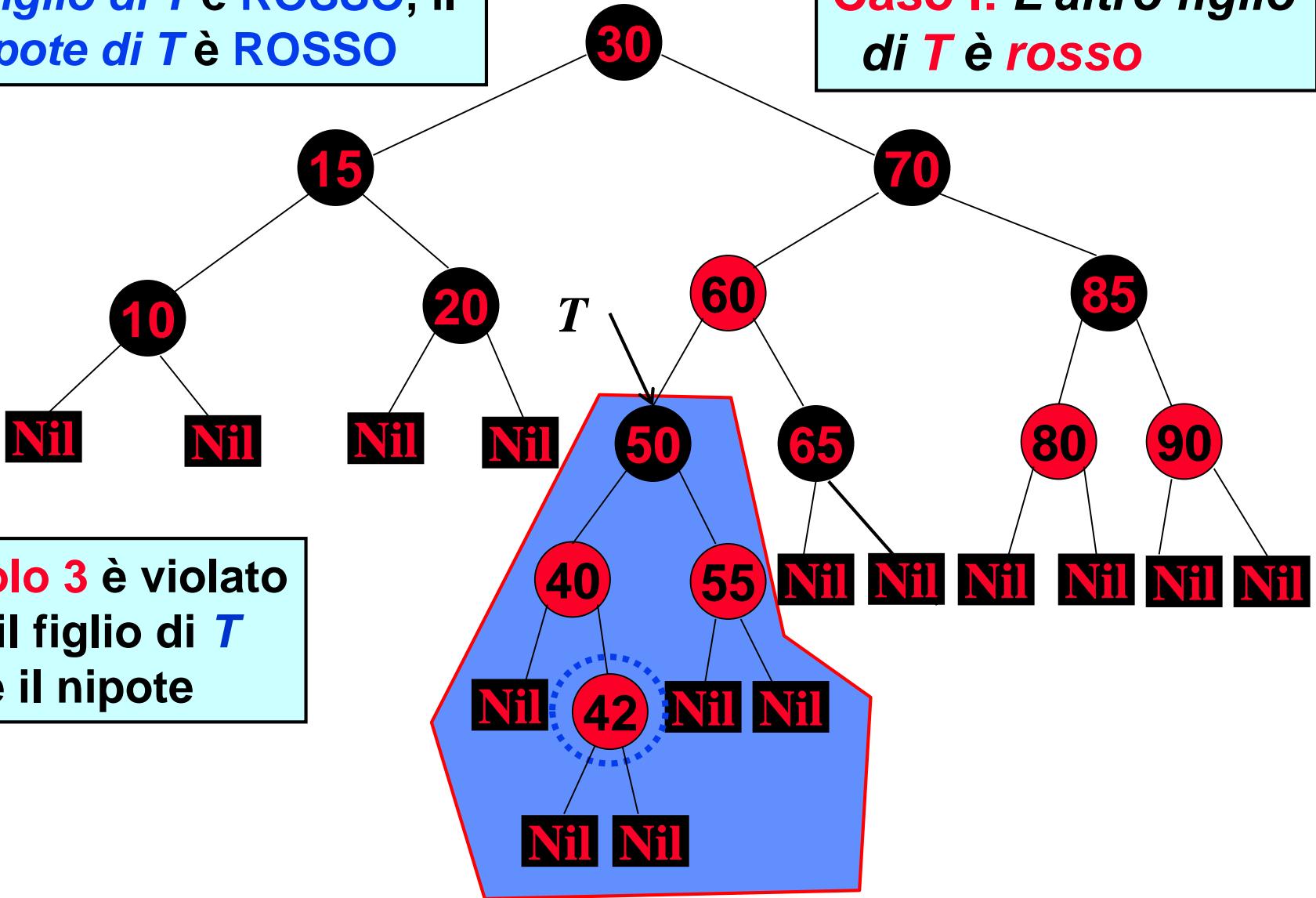
Non ci sono
nipoti di T



Inserimento in alberi Red-Black: II

Se il *figlio di T* è ROSSO, il
nipote di T è ROSSO

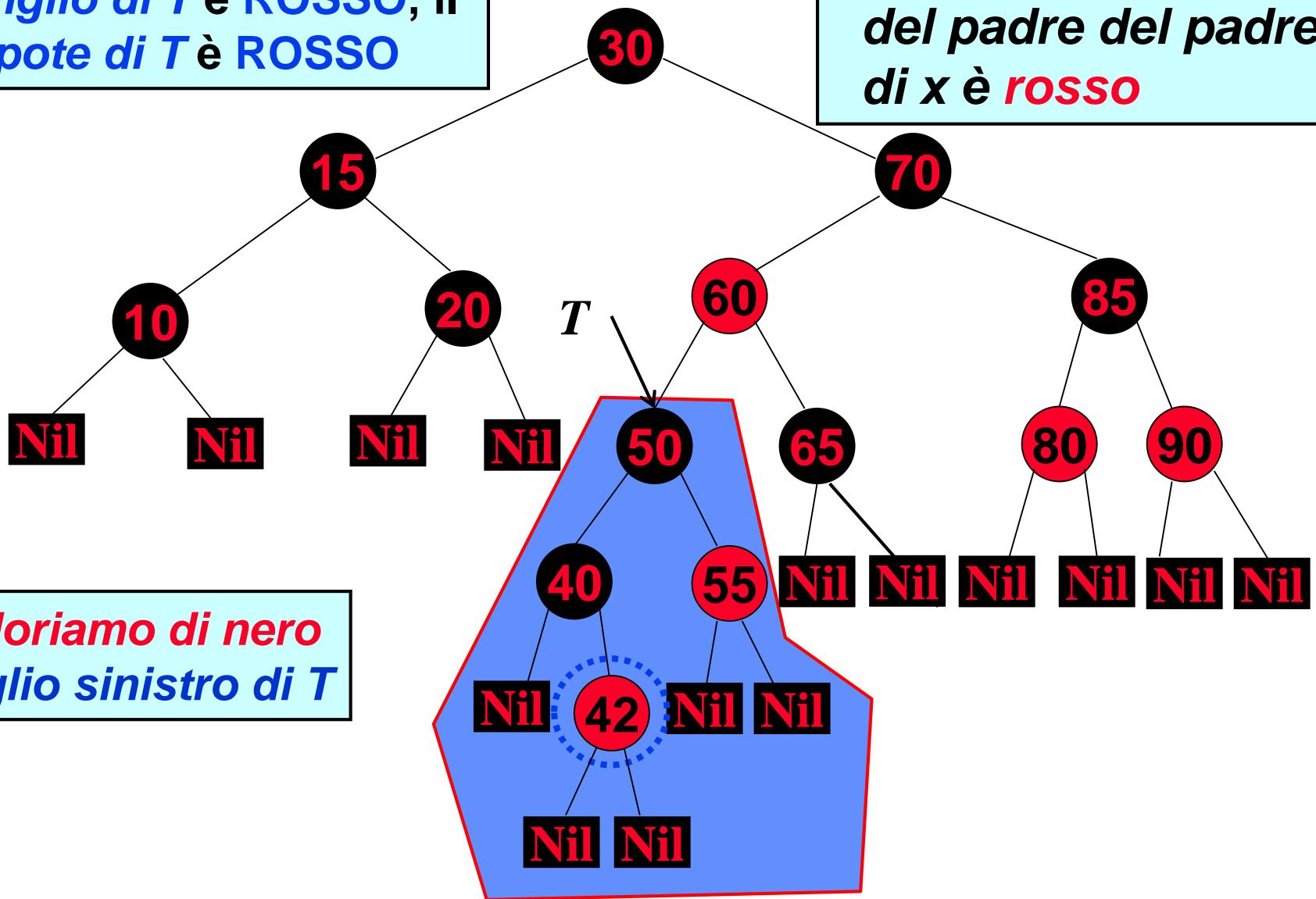
Caso I: L'*altro figlio*
di *T* è rosso



Inserimento in alberi Red-Black: II

Se il *figlio di T* è ROSSO, il
nipote di T è ROSSO

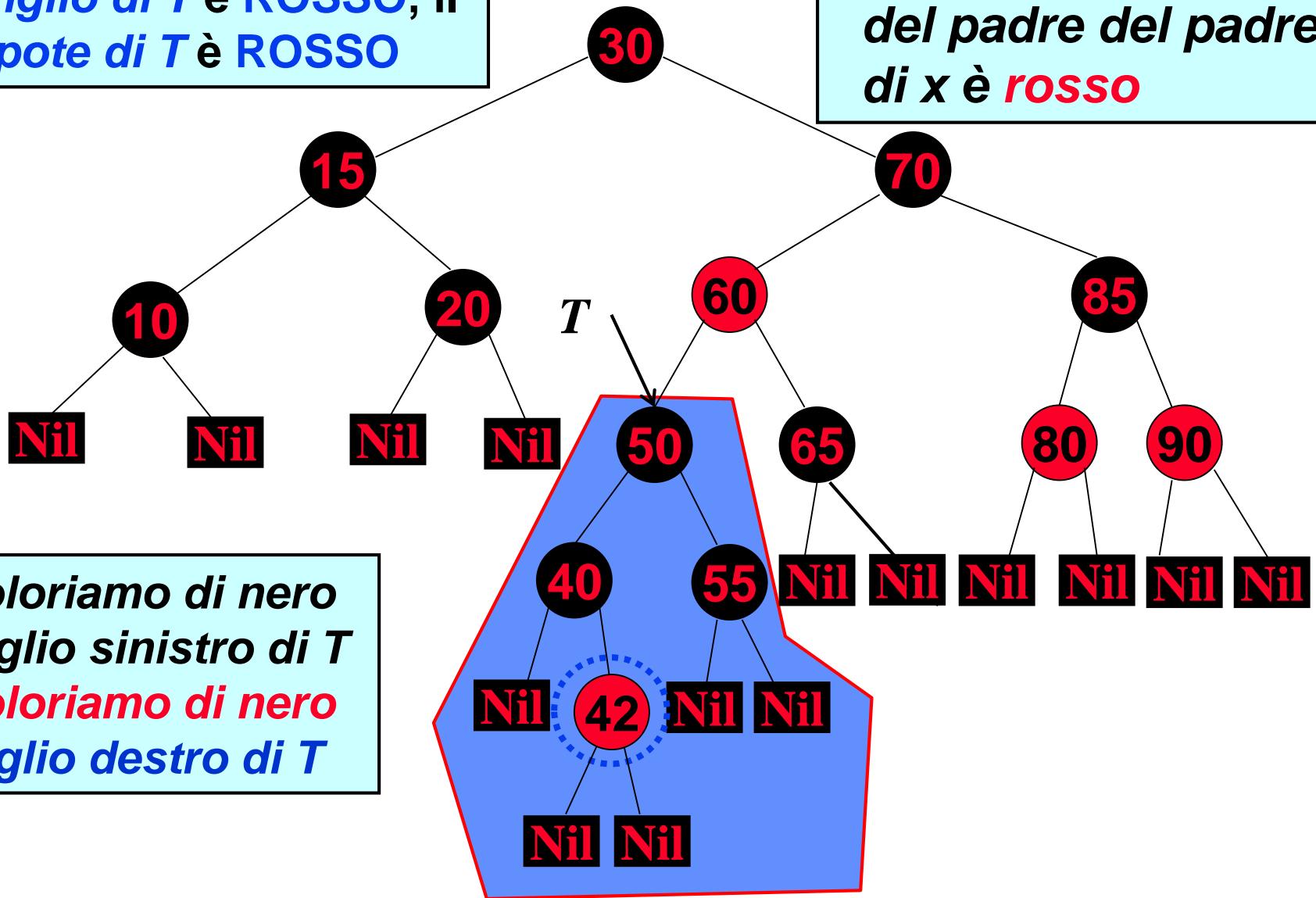
Caso I: L'altro figlio
del padre del padre
di x è rosso



Inserimento in alberi Red-Black: II

Se il *figlio di T* è ROSSO, il
nipote di T è ROSSO

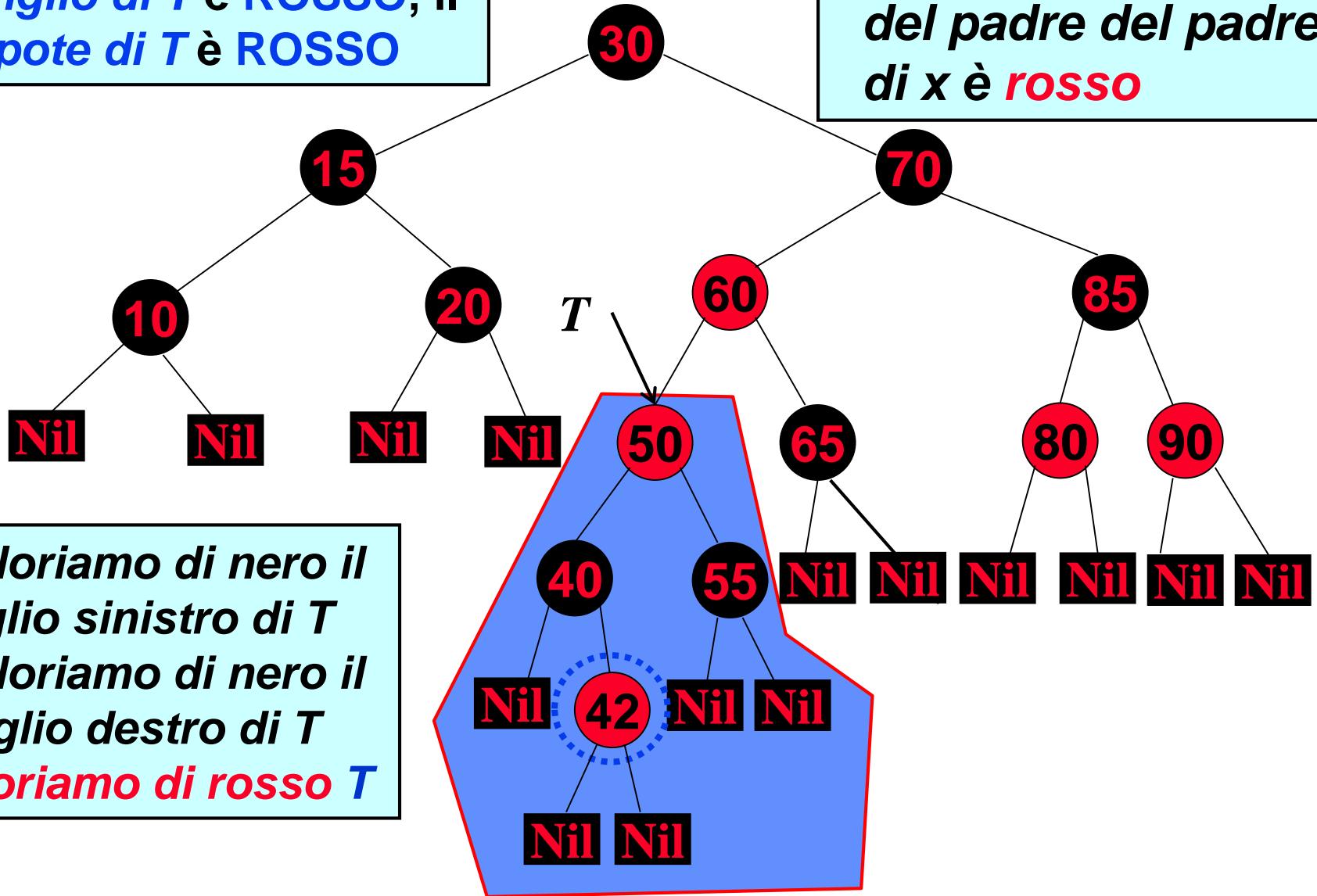
Caso I: L'altro figlio
del padre del padre
di x è rosso



Inserimento in alberi Red-Black: II

Se il *figlio di T* è ROSSO, il
nipote di T è ROSSO

Caso I: L'altro figlio
del padre del padre
di x è rosso



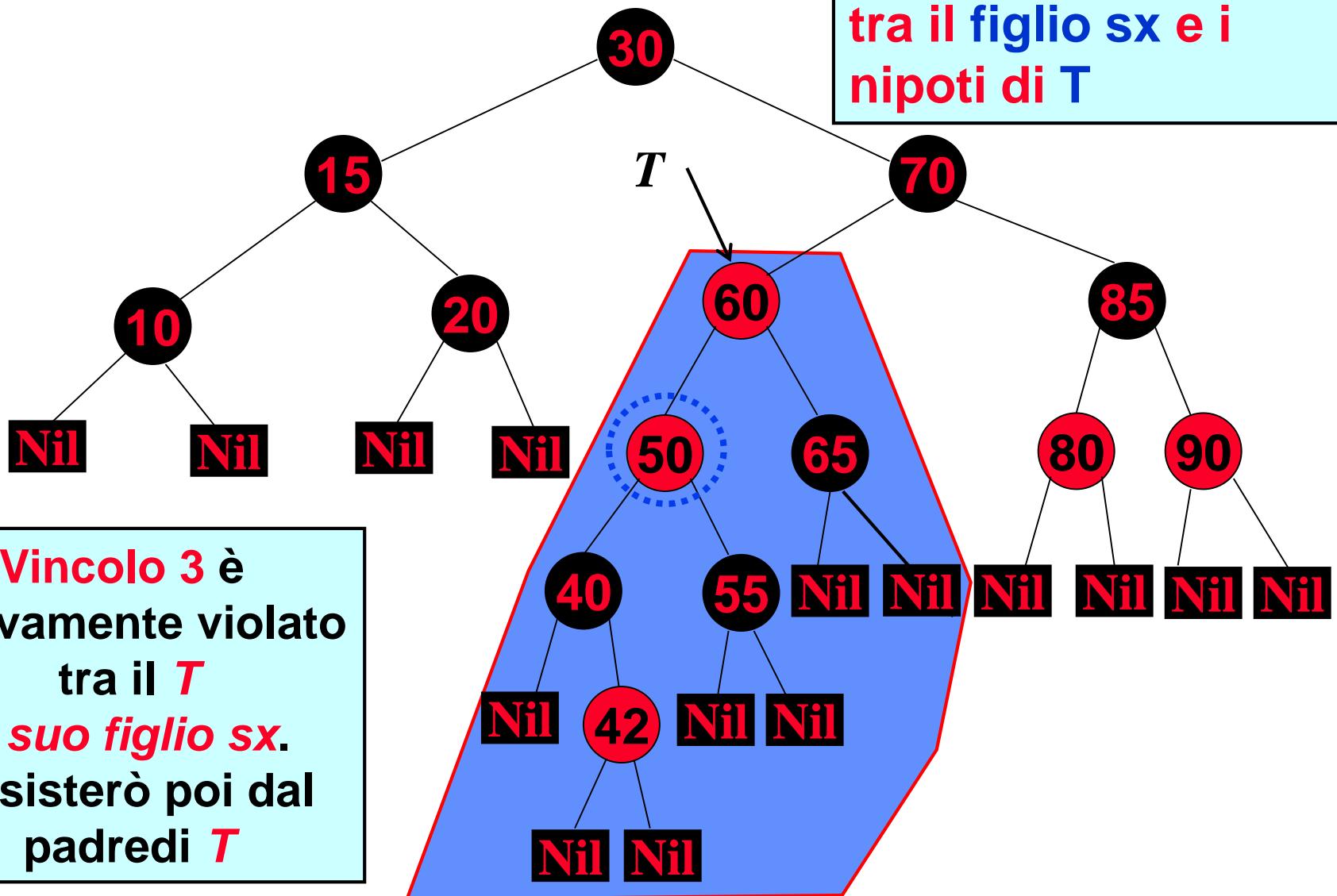
Inserimento in alberi Red-Black: II

Se il *figlio di T* è ROSSO, il
nipote di T è ROSSO

Caso I: L'altro figlio
del padre del padre
di x è rosso



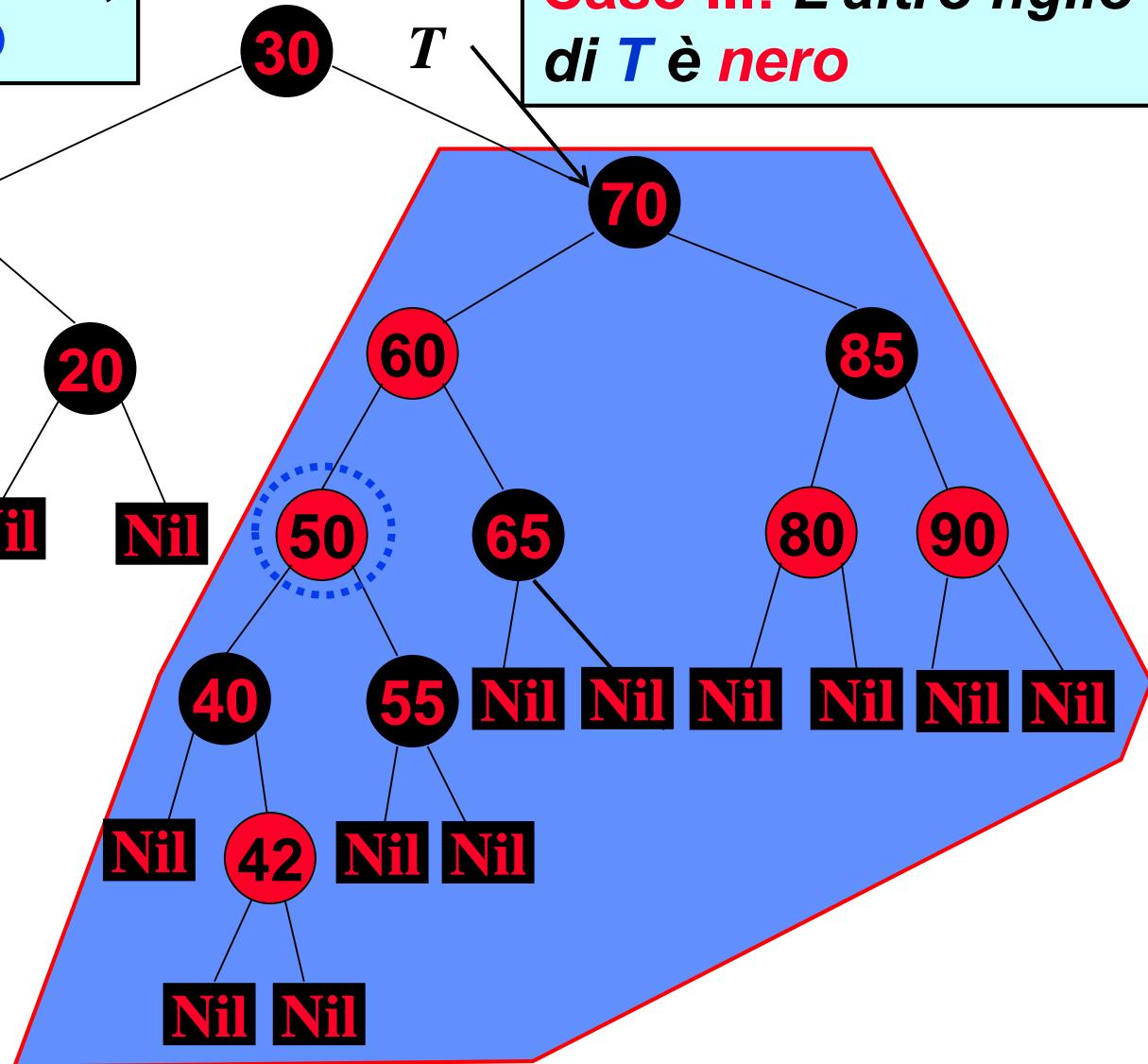
Inserimento in alberi Red-Black: II



Inserimento in alberi Red-Black: II

Se il *figlio sx di T* è ROSSO,
il *nopote* è ROSSO

Caso III: L'altro figlio
di *T* è nero

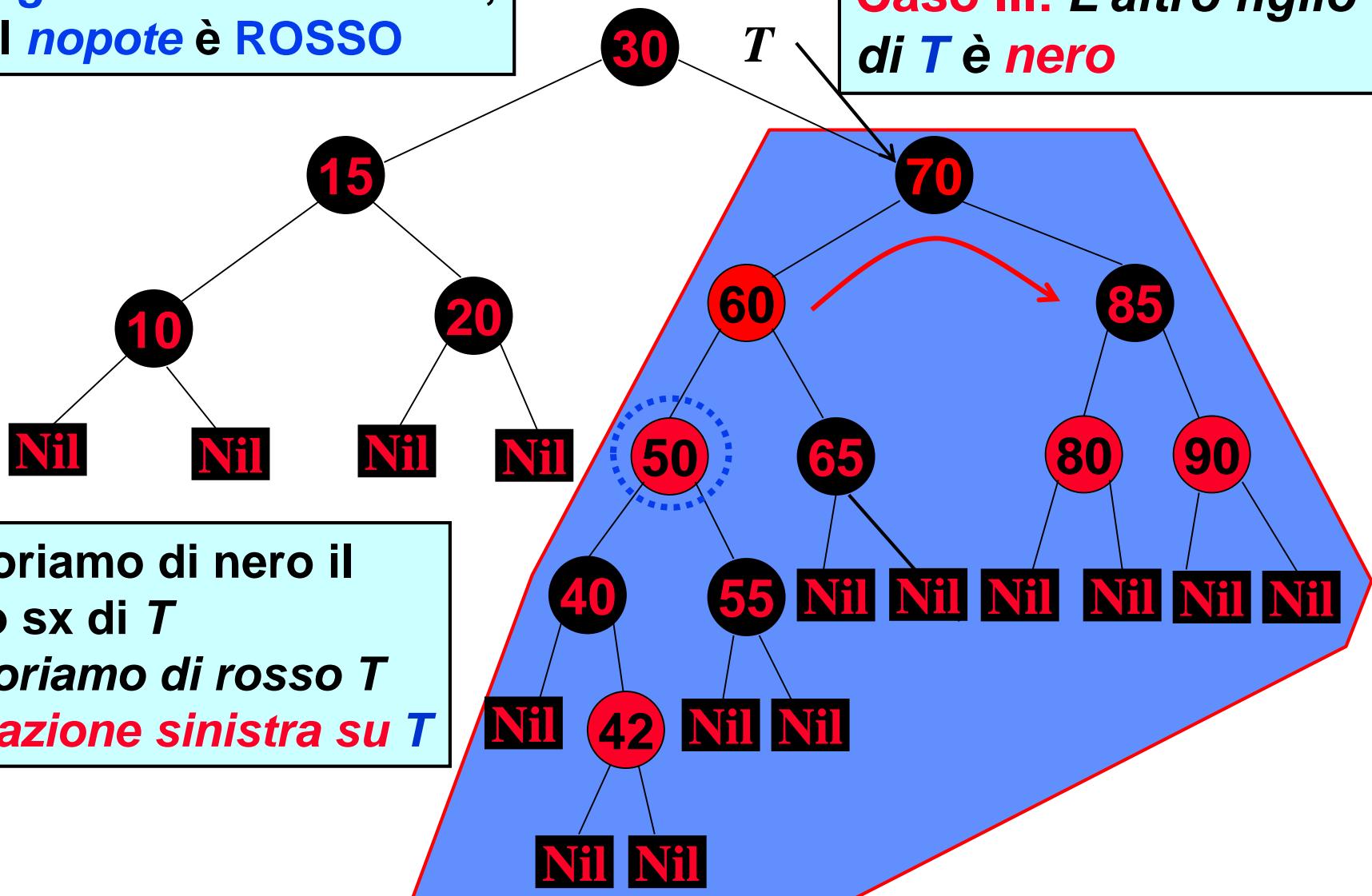


Vincolo 3 è
nuovamente violato
tra il *figlio sx* *T*
e il *suo figlio sx*

Inserimento in alberi Red-Black: II

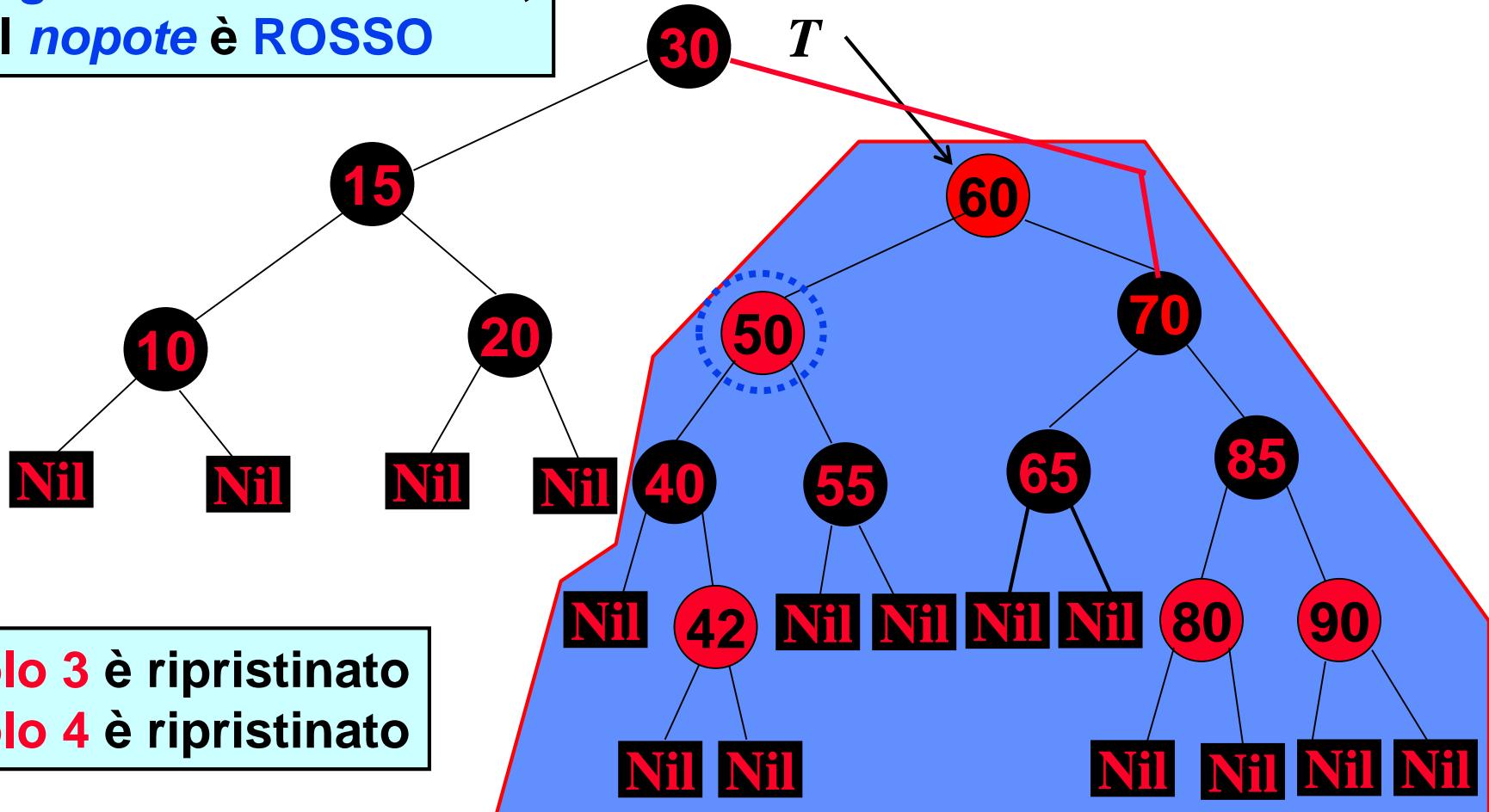
Se il *figlio sx di T* è ROSSO,
il *nopote* è ROSSO

Caso III: L'altro figlio
di T è nero



Inserimento in alberi Red-Black: II

Se il *figlio sx di T è ROSSO*,
il *nopote è ROSSO*

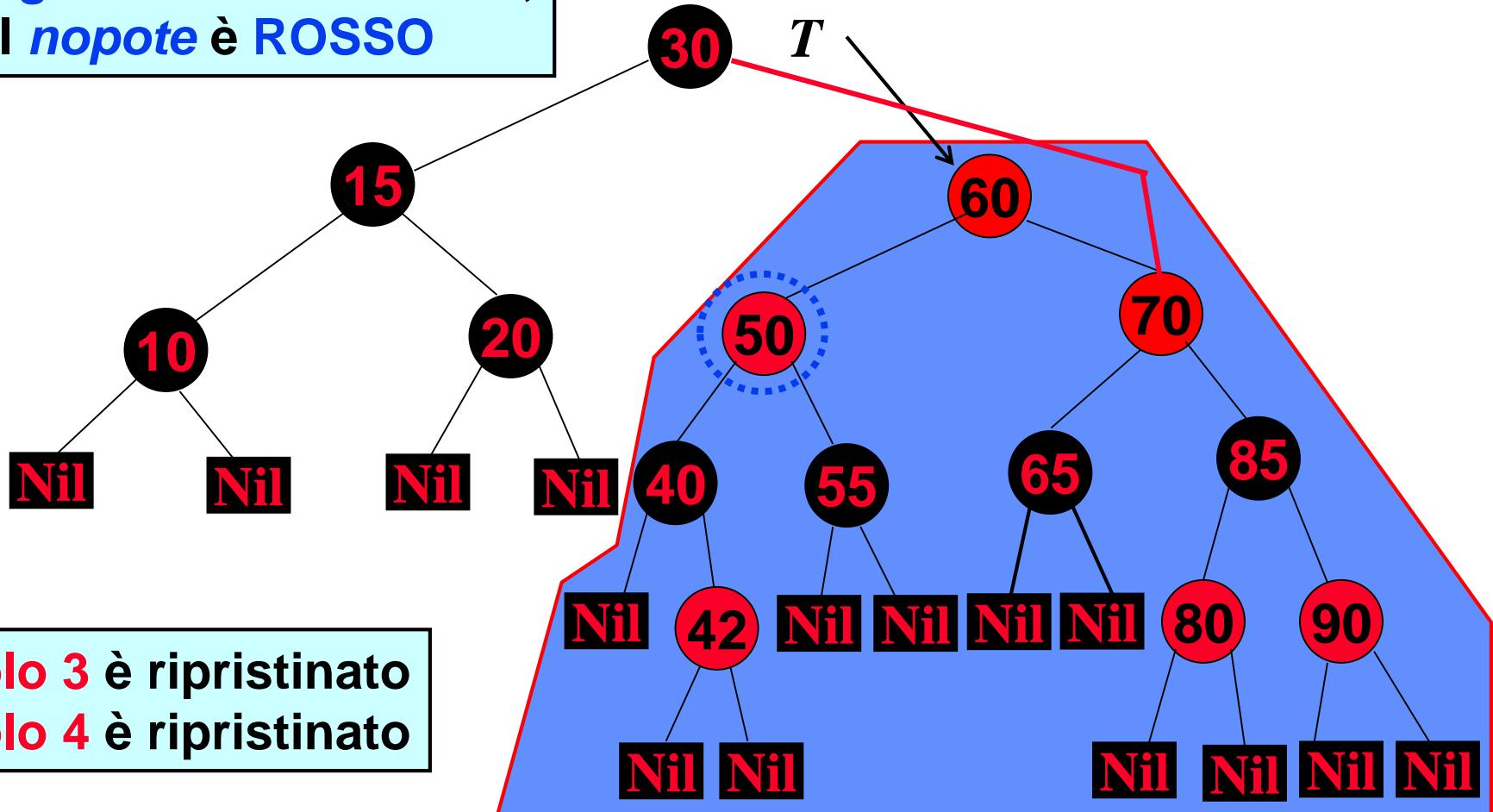


Vincolo 3 è ripristinato
Vincolo 4 è ripristinato

Ma attenzione: la radice del sottoalbero è cambiata

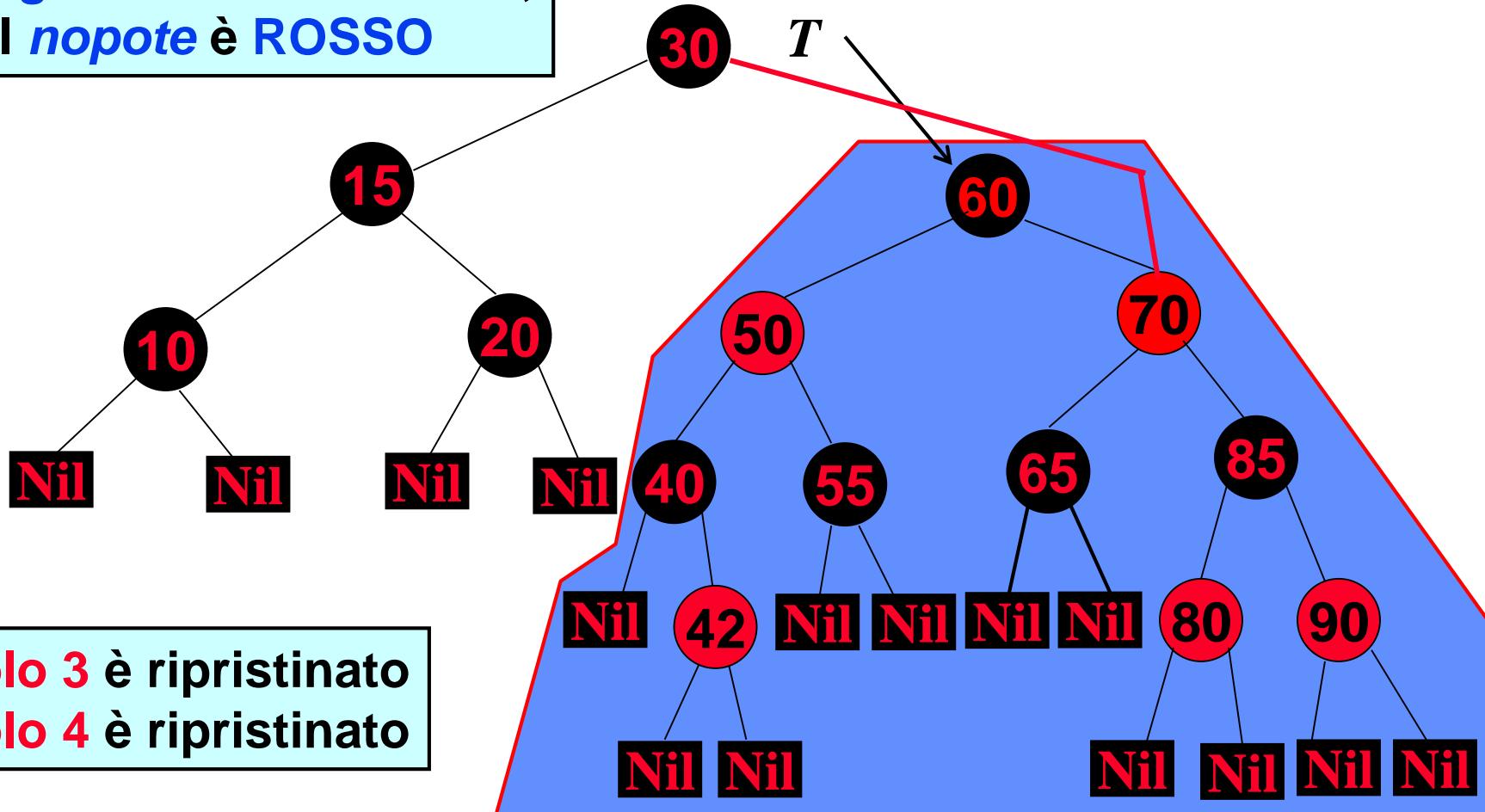
Inserimento in alberi Red-Black: II

Se il *figlio sx di T è ROSSO*,
il *nopote è ROSSO*



Inserimento in alberi Red-Black: II

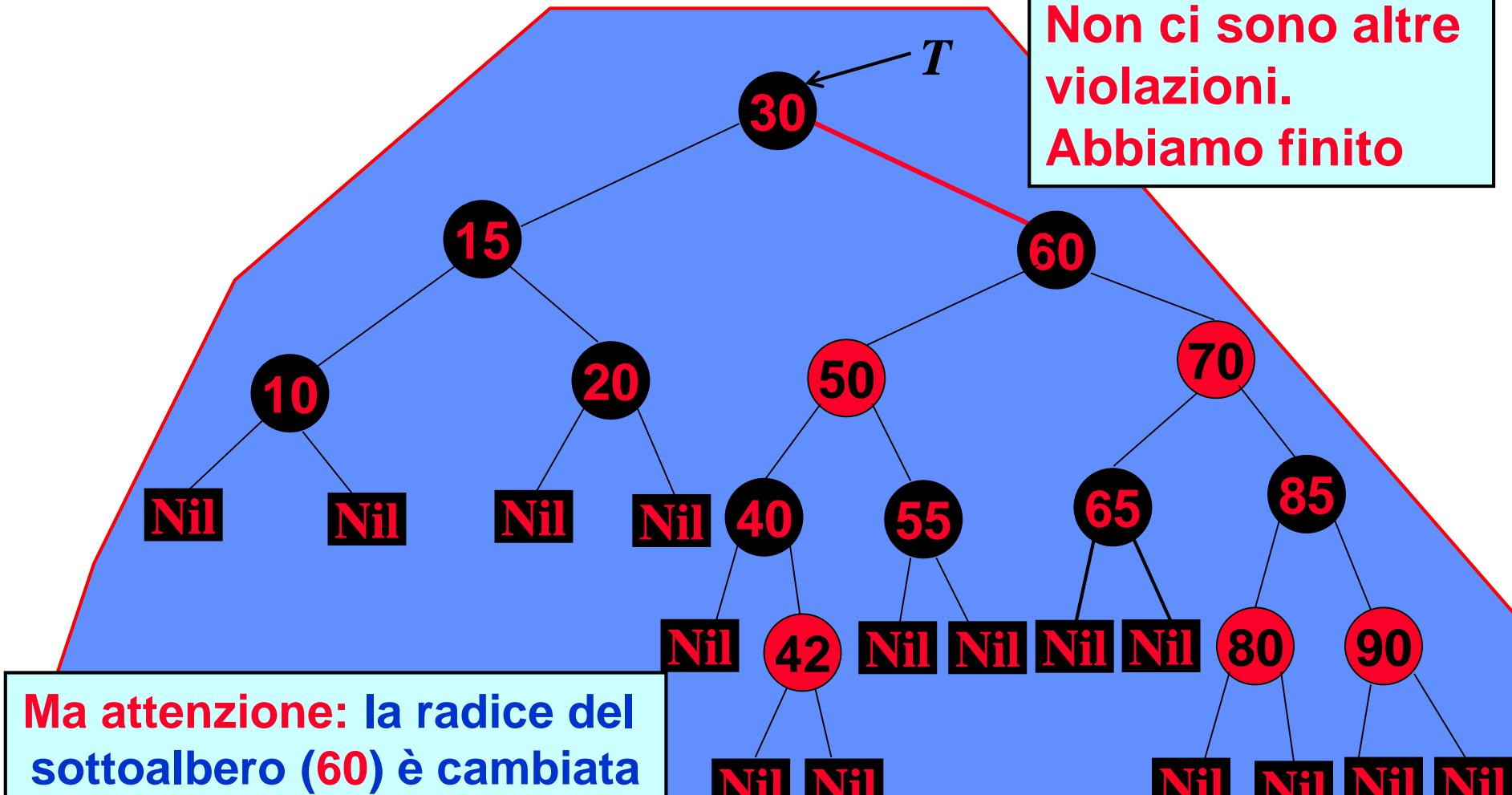
Se il *figlio sx di T è ROSSO*,
il *nopote è ROSSO*



Ma attenzione: la radice del sottoalbero è cambiata

Inserimento in alberi Red-Black- II

Non ci sono altre violazioni.
Abbiamo finito



$T->dx = \text{INS_RB}(k, T->dx, T)$

*al ritorno della chiamata ricorsiva
risistema l'albero dopo le rotazioni*

Cancellazione in RB

- L'algoritmo di cancellazione per alberi RB è costruito sull'algoritmo di cancellazione per alberi binari di ricerca
 - Useremo una variante con delle *sentinelle Nil[T]*, una per ogni nodo NIL per semplificare l'algoritmo
- Dopo la cancellazione si deve decidere se è necessario ribilanciare o meno
- *Le operazioni di ripristino del bilanciamento sono necessarie solo quando il nodo cancellato è nero! (perché?)*

Cancellazione in RB

- Dobbiamo ribilanciare se il *nodo y cancellato è nero* (perché è *cambiata l'altezza nera*)
- Possiamo immaginare di *spostare di nero y sul nodo x* che *sostituisce* il *nodo cancellato y*
- In tal modo la *cancellazione non viola* più il *vincolo 4* ...
- ... ma potrebbe violare il *vincolo 1 (perché?)*
- Gli algoritmi **Canc-Bil-dx(T)** e **Canc-Bil-sx(T)** tentano di ripristinare il *vincolo 1* con rotazioni e cambiamenti di colore:
 - ci sono *4 casi possibili* (e 4 simmetrici)

Canc-RB (T , K)

```
IF not IS-NIL(T) THEN
    IF T->key < K THEN
        T->dx = Canc-RB (T->dx , K)
        T = Canc-Bil-dx (T)
    ELSE IF T->key > K THEN
        T->sx = Canc-RB (T->sx , K)
        T = Canc-Bil-sx (T)
    ELSE
        T = Canc-Radice-RB (T)
RETURN T
```

Canc-Radice-RB (T)

```
IF IS-NIL (T->sx) || IS-NIL (T->dx) THEN
    tmp = T
    IF IS-NIL (T->sx) THEN
        T = T->dx
    ELSE IF IS-NIL (T->dx) THEN
        T = T->sx
    IF tmp->color = black THEN
        Propagate-Black (T)
    ELSE
        tmp = Stacca-Min-RB (T->dx , T)
        T->Key = tmp->Key
        T = Canc-Bil-dx (T)
    dealloca tmp
RETURN T
```

Propagate-Black (T)

```
IF T->color = red THEN
    T->color = black
ELSE
    T->color = d-black
```

Stacca-Min-RB (T, P)

```
IF not IS-NIL(T) THEN
    IF not IS-NIL(T->sx) THEN
        tmp = Stacca-Min-RB(T->sx, T)
        IF T = P->sx THEN
            P->sx = Canc-Bil-sx(T)
        ELSE
            P->dx = Canc-Bil-dx(T)
        T = tmp
    ELSE
        tmp = T
        IF T = P->sx THEN
            P->sx = T->dx
        ELSE
            P->dx = T->dx
        IF T->color = black
            Propagate-Black(T->dx)
return tmp
```

Propagate-Black (T)

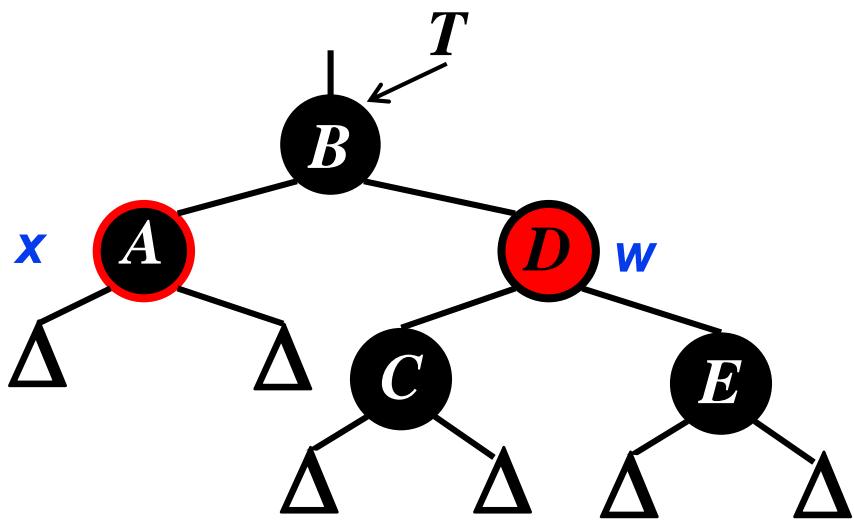
```
IF T->color = red THEN
    T->color = black
ELSE
    T->color = d-black
```

Cancellazione in RB: casi

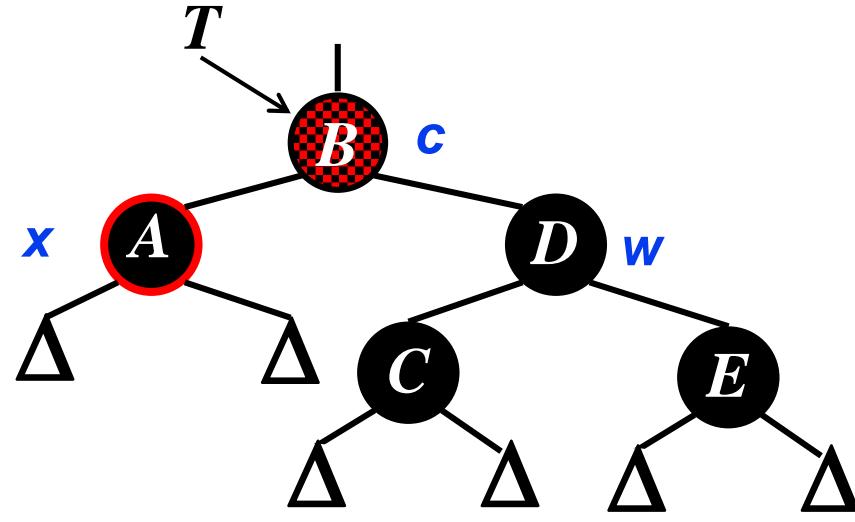
- Abbiamo visto i **4 casi possibili** quando la cancellazione è avvenuta a **sinistra**
- Esistono anche i **4 casi simmetrici** (con destro e sinistro scambiati) quando la cancellazione è avvenuta a **destra**

Esercizio: Illustrare i 4 casi simmetrici e scrivere lo pseudo-codice che li gestisce

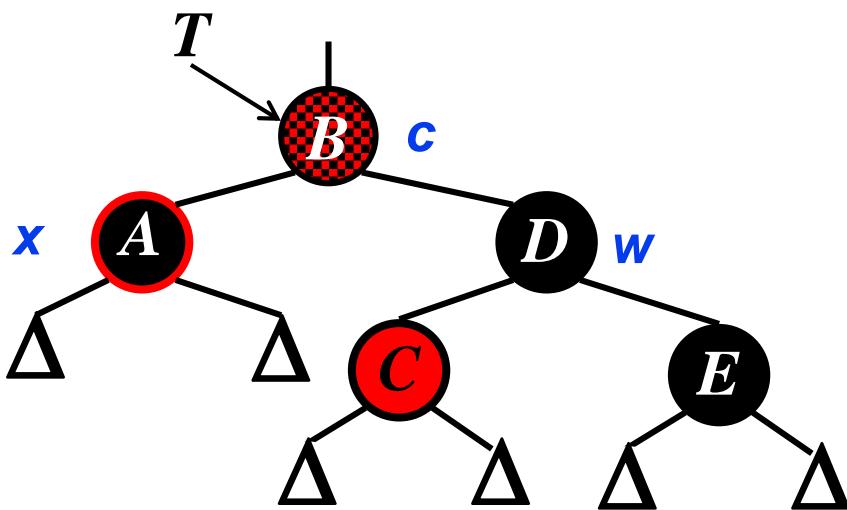
caso 1



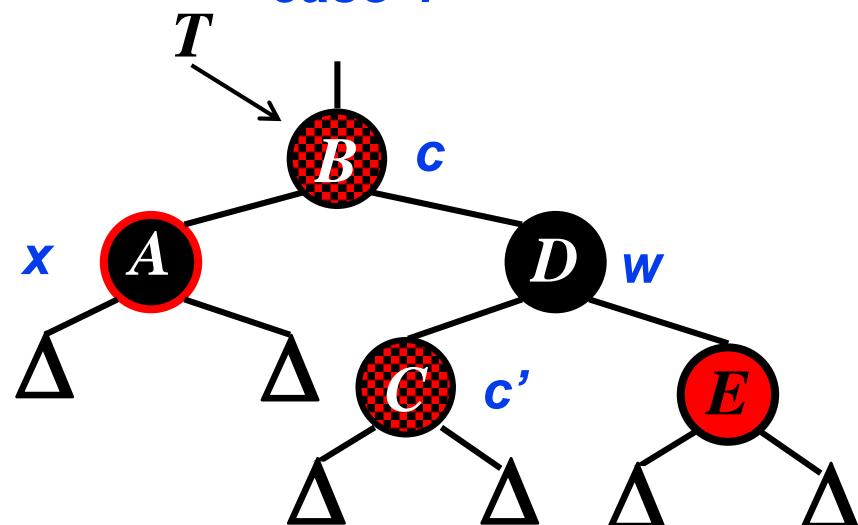
caso 2



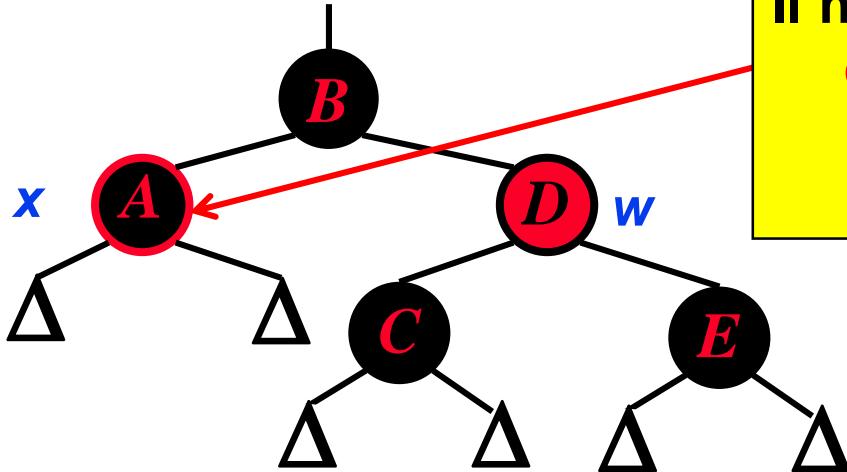
caso 3



caso 4



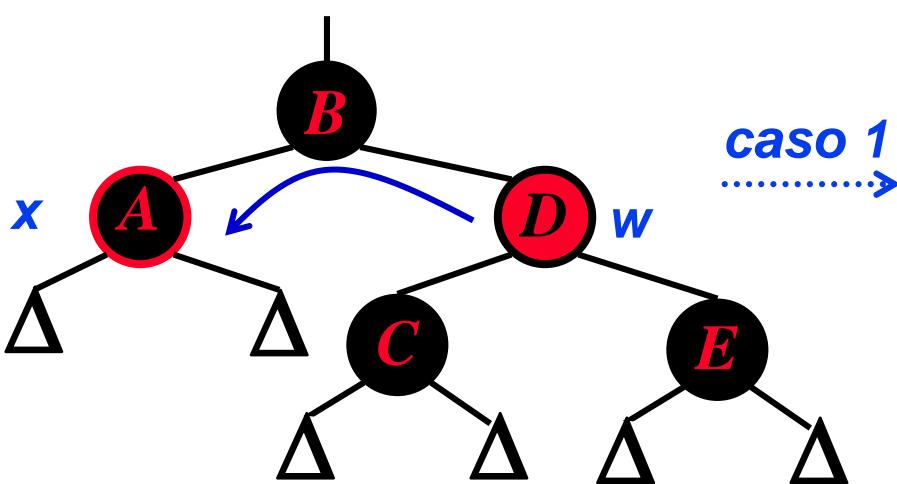
Cancellazione in RB: caso 1



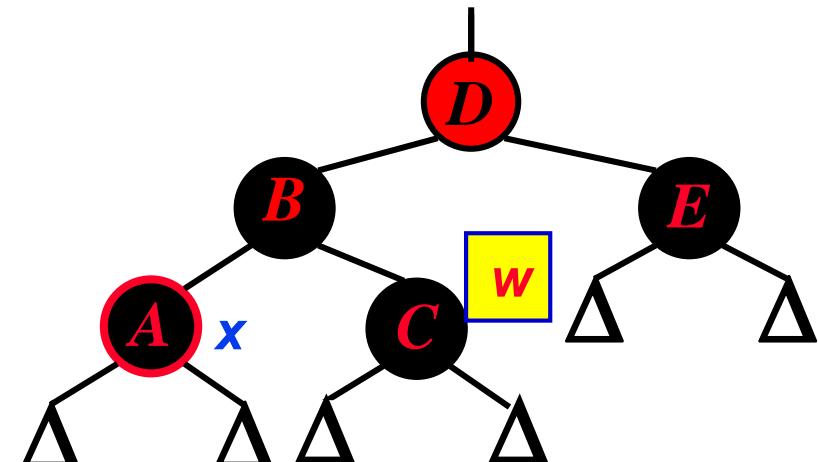
Il nodo x (cioè A nell'esempio) è bordato di rosso ad indicare che è il nodo con il colore nero in più da ridistribuire nell'albero

- Il fratello w di x è rosso
- w deve avere figli neri

Cancellazione in RB: caso 1

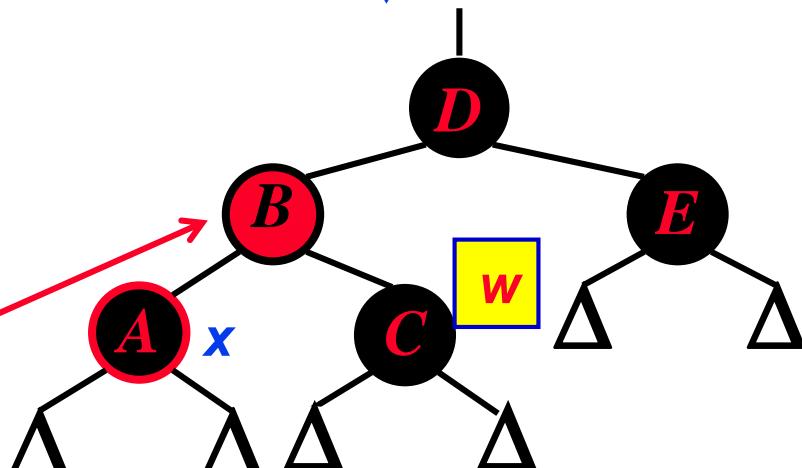


caso 1
.....→

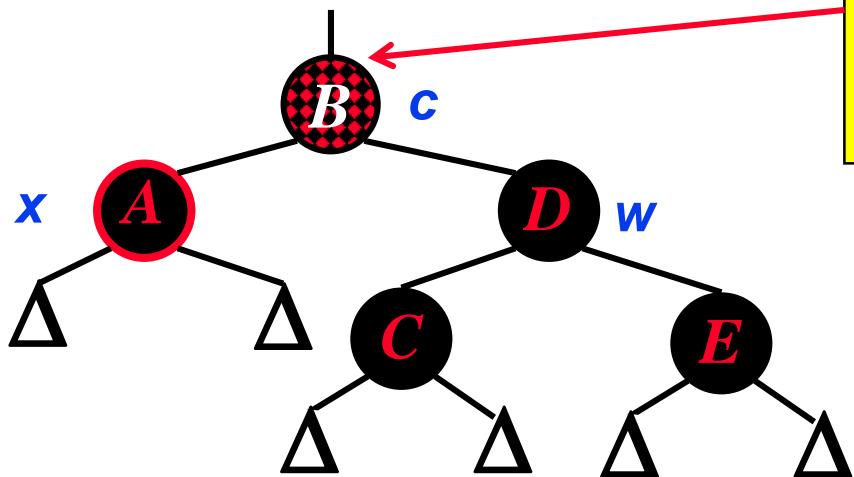


- Il fratello *w* di *x* è rosso
- *w* deve avere figli neri
- cambiamo i colori di *w* e del padre di *x* e li ruotiamo tra loro

Non violiamo né il *vincolo 3* né il *4*
e ci riduciamo ad uno degli altri
casi, chiamando nuovamente il
bilanciamento sul nodo B



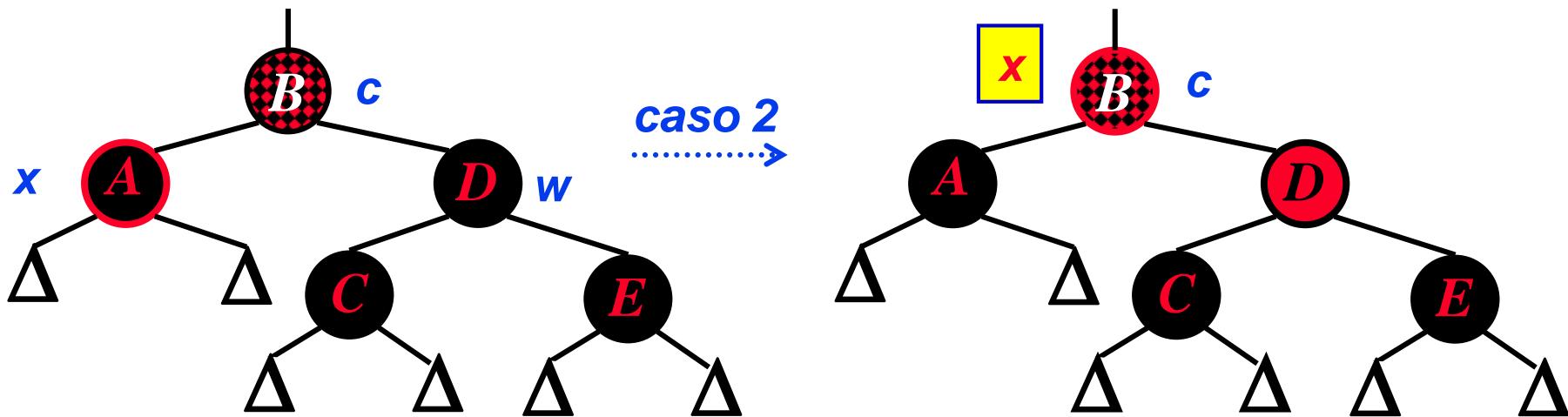
Cancellazione in RB: caso 2



Il nodo **c** (cioè **B** nell'esempio) può essere sia rosso che nero!

- Il **fratello w** di **x** è **nero**
- **w** ha in questo caso entrambi i figli **neri**

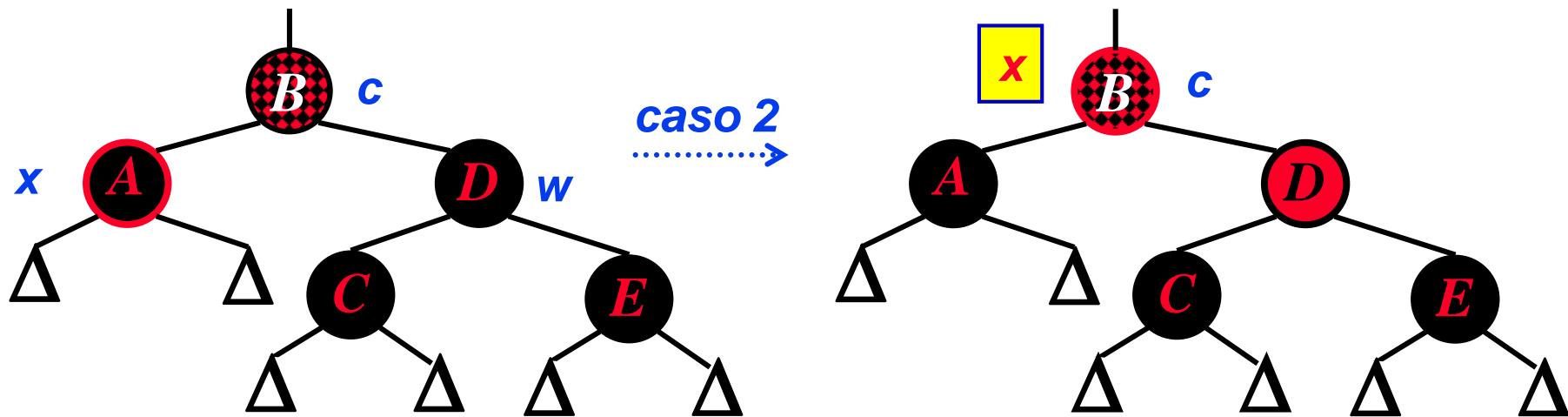
Cancellazione in RB: caso 2



- Il fratello **w** di **x** è nero
- **w** ha in questo caso entrambi i figli neri
- cambiamo il colore di **w** e il nuovo **x** diventa il padre

Spostiamo il nero in più da **x** a **c** (il padre) e togliamo il nero da **w** per rispettare vincolo 4 lungo il sottoalbero destro di **c**

Cancellazione in RB: caso 2

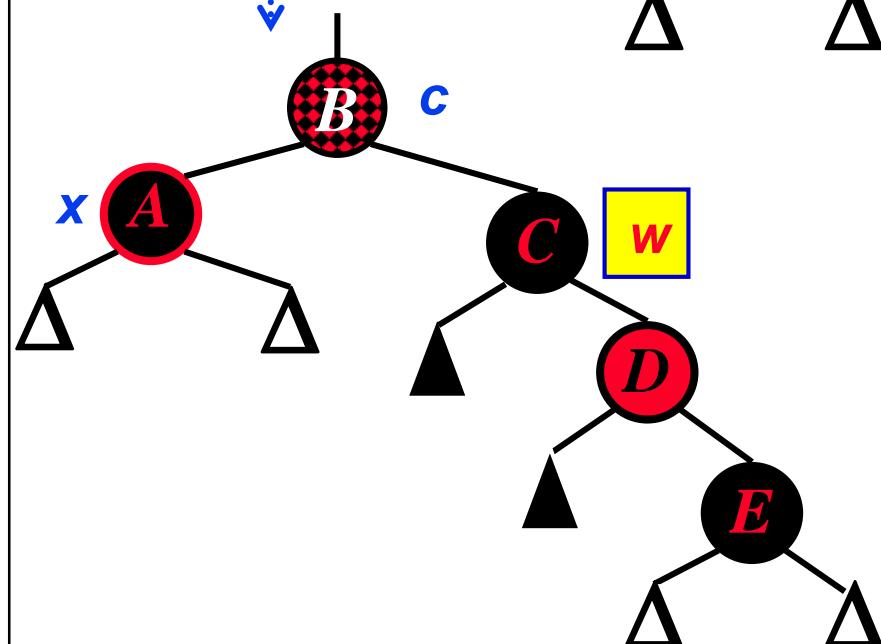
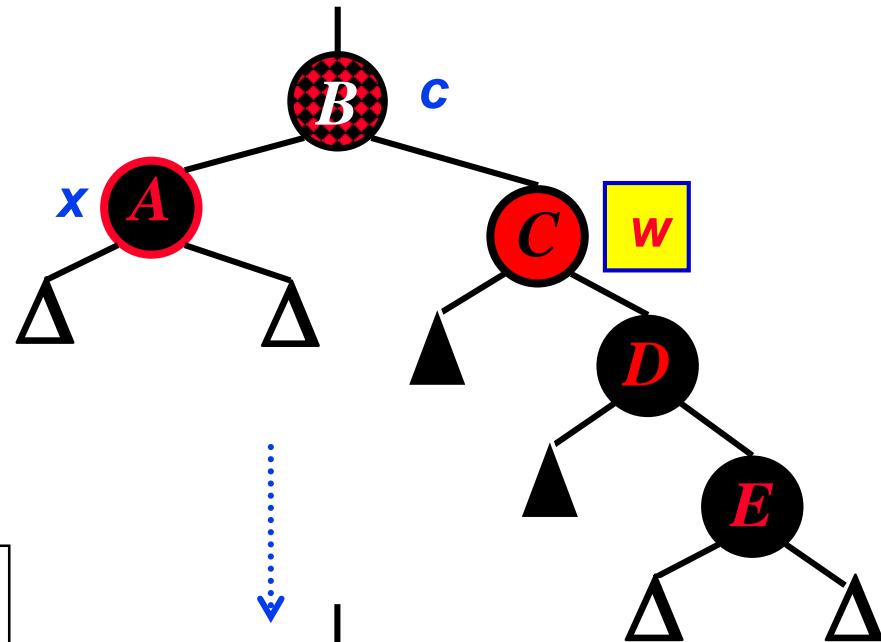
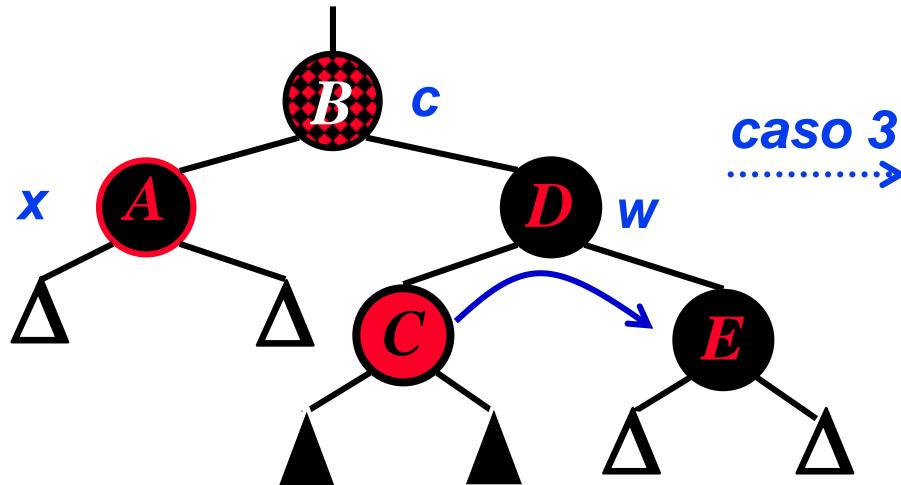


- Il fratello **w** di **x** è **nero**
- **w** ha in questo caso entrambi i figli **neri**
- cambiamo il colore di **w** e il nuovo **x** diventa il padre

Spostiamo il **nero in più** da **x** a **c** (il **padre**) e togliamo il nero da **w** per rispettare **vincoli**

Se si arriva dal **caso 1**, **B** è sicuramente **rosso**, quindi dopo il **caso 2** non c'è più bisogno di ribilanciare, perché ora **B** ha un solo nero (il **nero in più**).

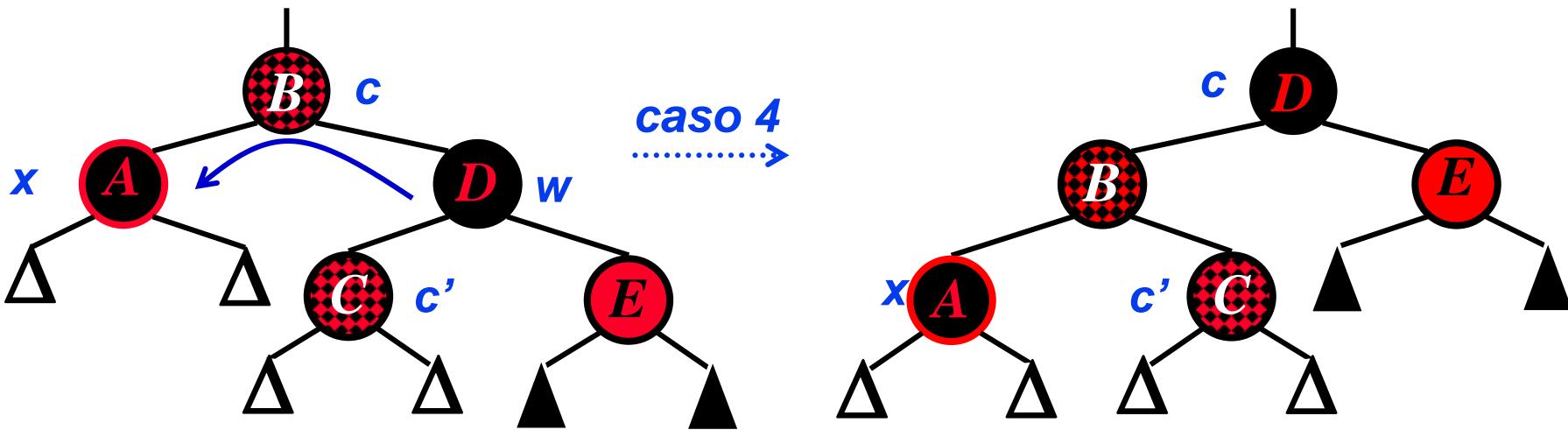
Cancellazione in RB: caso 3



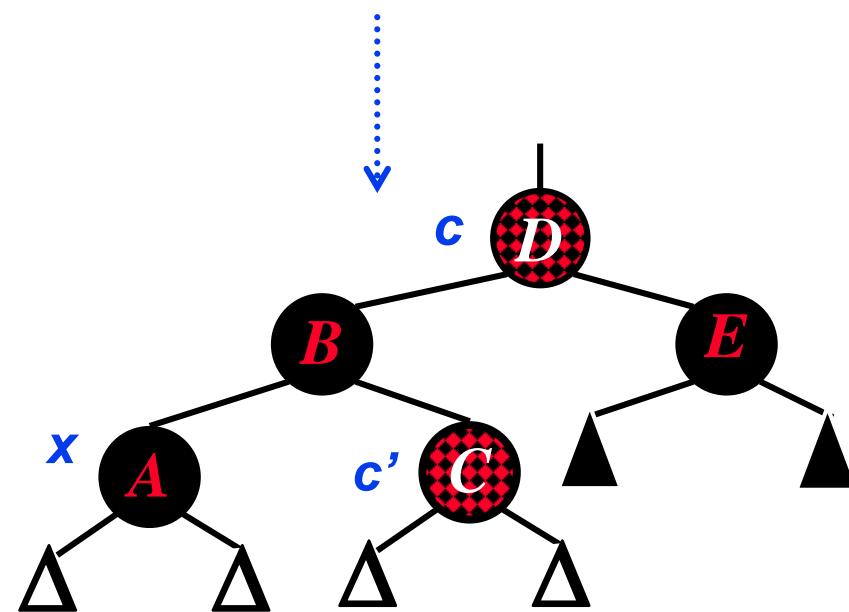
- Il fratello **w** di **x** è **nero**
- **w** ha in questo caso solo il figlio sinistro **rosso**
- *ruotiamo w col suo figlio sinistro*
- cambiamo il colore di **w** e quello del destro di **w**

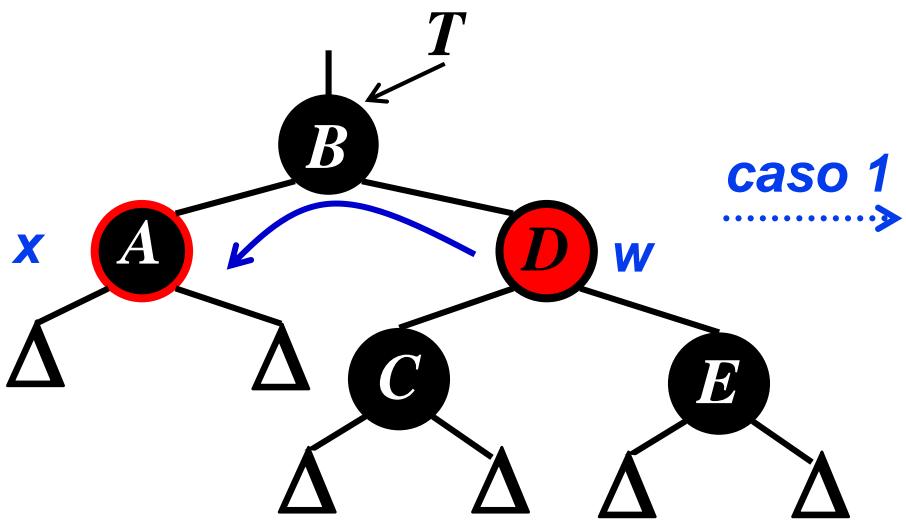
Non violiamo alcun **vincolo (3 e 4)** e il nuovo fratello si **x** è ora nero con figlio sinistro nero, quindi andiamo nel **caso 4**

Cancellazione in RB: caso 4

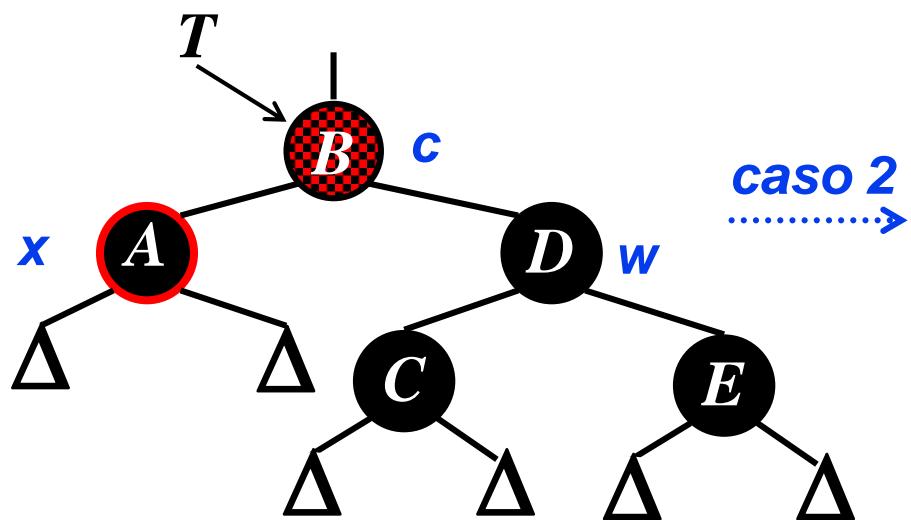
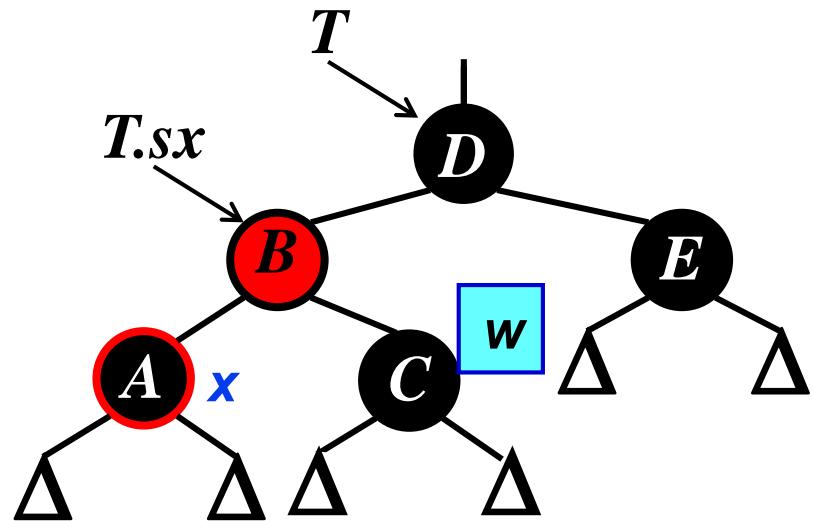


- Il *fratello w* di x è *nero*
 - w ha in questo caso solo il figlio destro *rosso*
 - eseguiamo una *rotazione del padre di x con w* e cambiamo i colori opportunamente, *eliminando il nero in più su x*
- Non violiamo alcun vincolo (3 e 4) e abbiamo finito!*

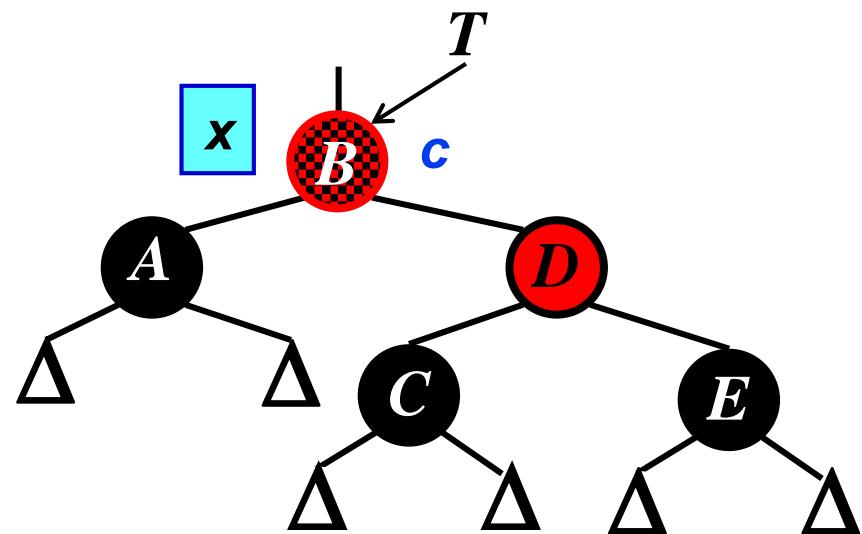


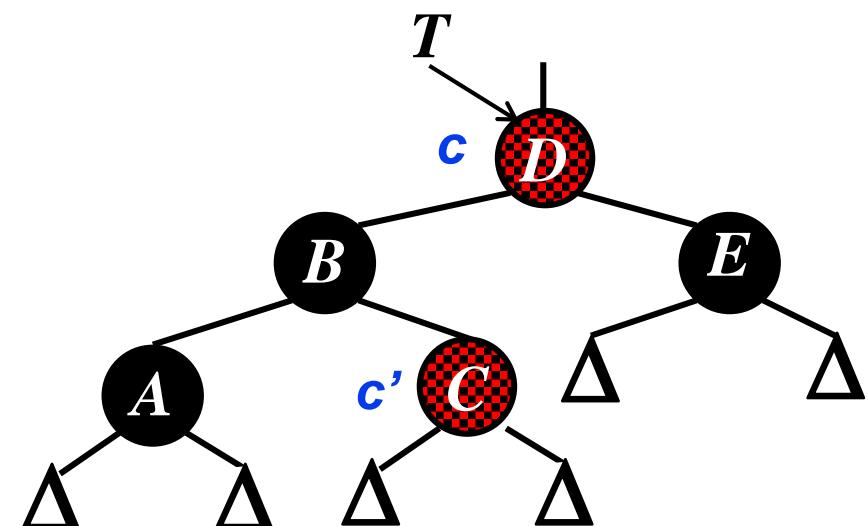
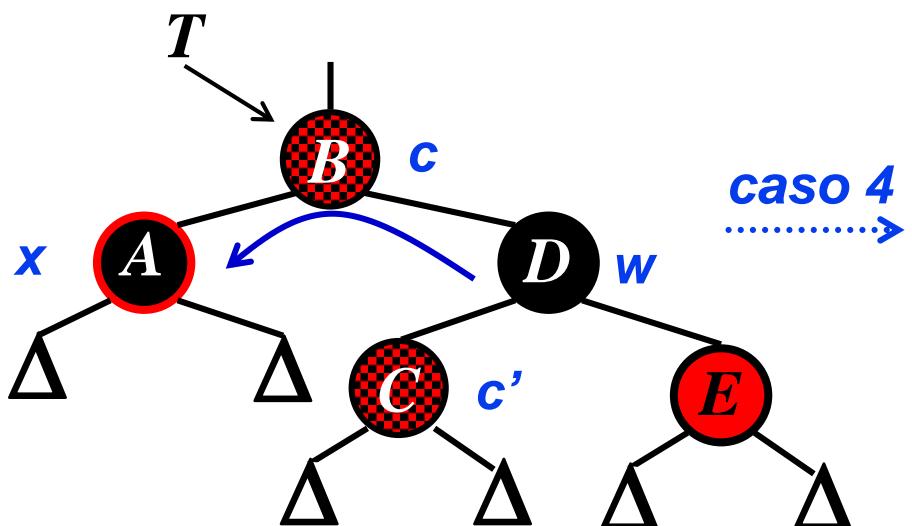
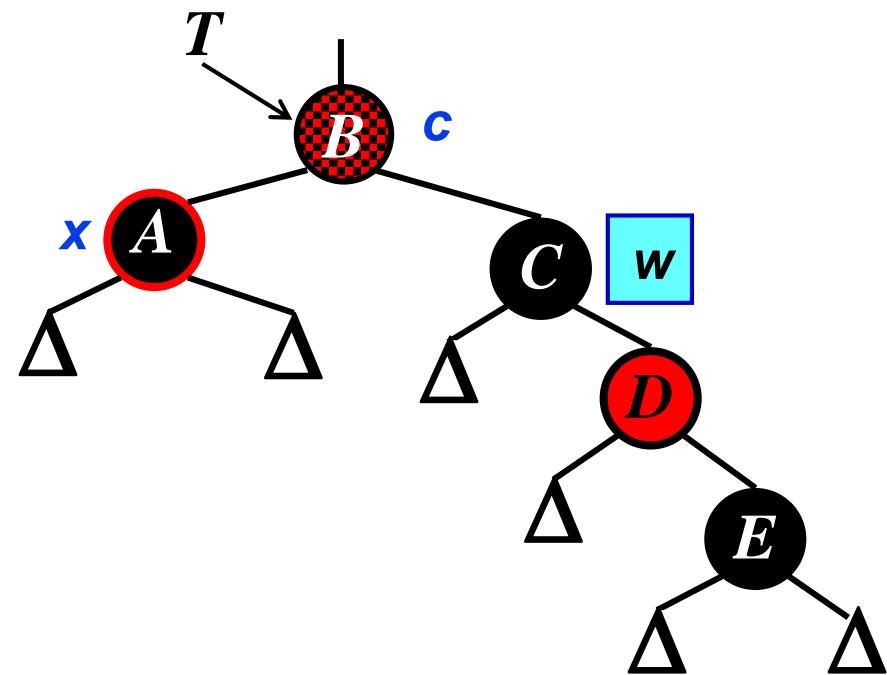
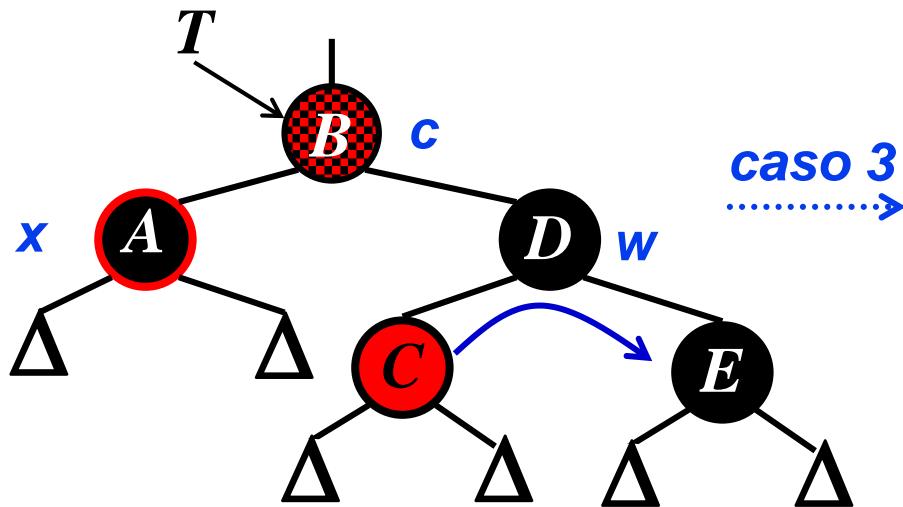


caso 1



caso 2





Cancellazione in RB

```
Cancella_Bil_sx(T)
  IF ha_figli(T)
    v = Violazione_sx(T->sx, T->dx)
    /* nessuna violazione se v = 0 */
    CASE v OF
      1: T = Canc_bil_1_sx(T);
          T->sx = Cancella_Bil_sx(T->sx);
      2: T = Canc_bil_2_sx(T);
      3: T = Canc_bil_3_sx(T);
      4: T = Canc_bil_4_sx(T);
  return T;
```

Violazione-sx(X,W)

```
viola = 0
IF X->color = d-black THEN
    IF W->color=red THEN
        viola = 1
    ELSE IF W->dx = black &
            W->sx = black THEN
        viola = 2
    ELSE IF W->dx = black THEN
        viola = 3
    ELSE /* W->dx = red */
        viola = 4
return viola
```

Canc-Bil-1-sx(T)

```
T = ruota-dx(T)
T->color = black
T->sx->color = red
return T
```

Canc-Bil-2-sx(T)

```
T->dx->color = red
T->sx->color = black
propagate-black(T)
return T
```

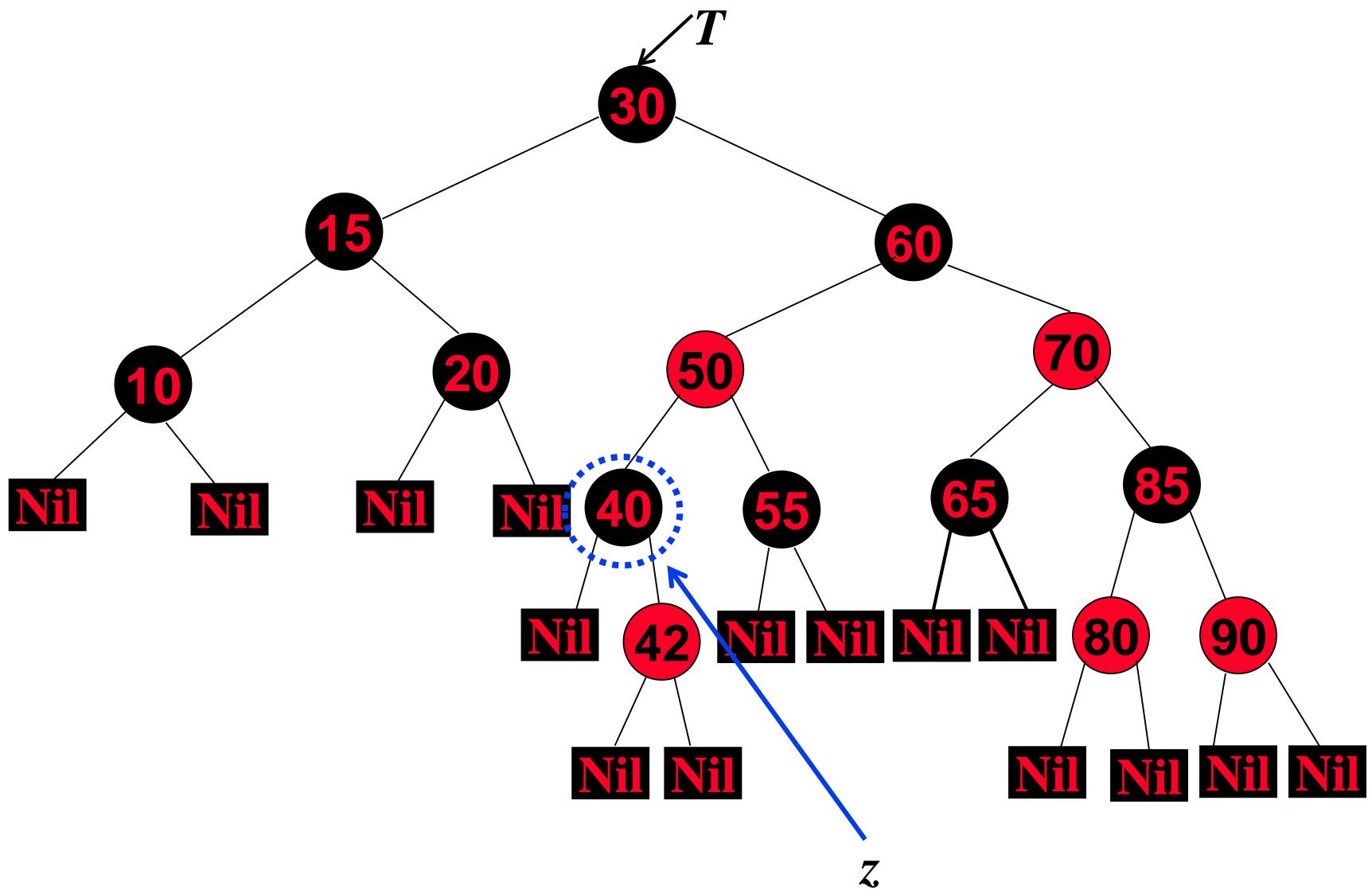
Canc-Bil-3-sx(T)

```
T->dx = ruota-sx(T->dx)
T->dx->color = black
T->dx->dx->color = red
T = Canc-Bil-4-sx(T)
return T
```

Canc-Bil-4-sx(T)

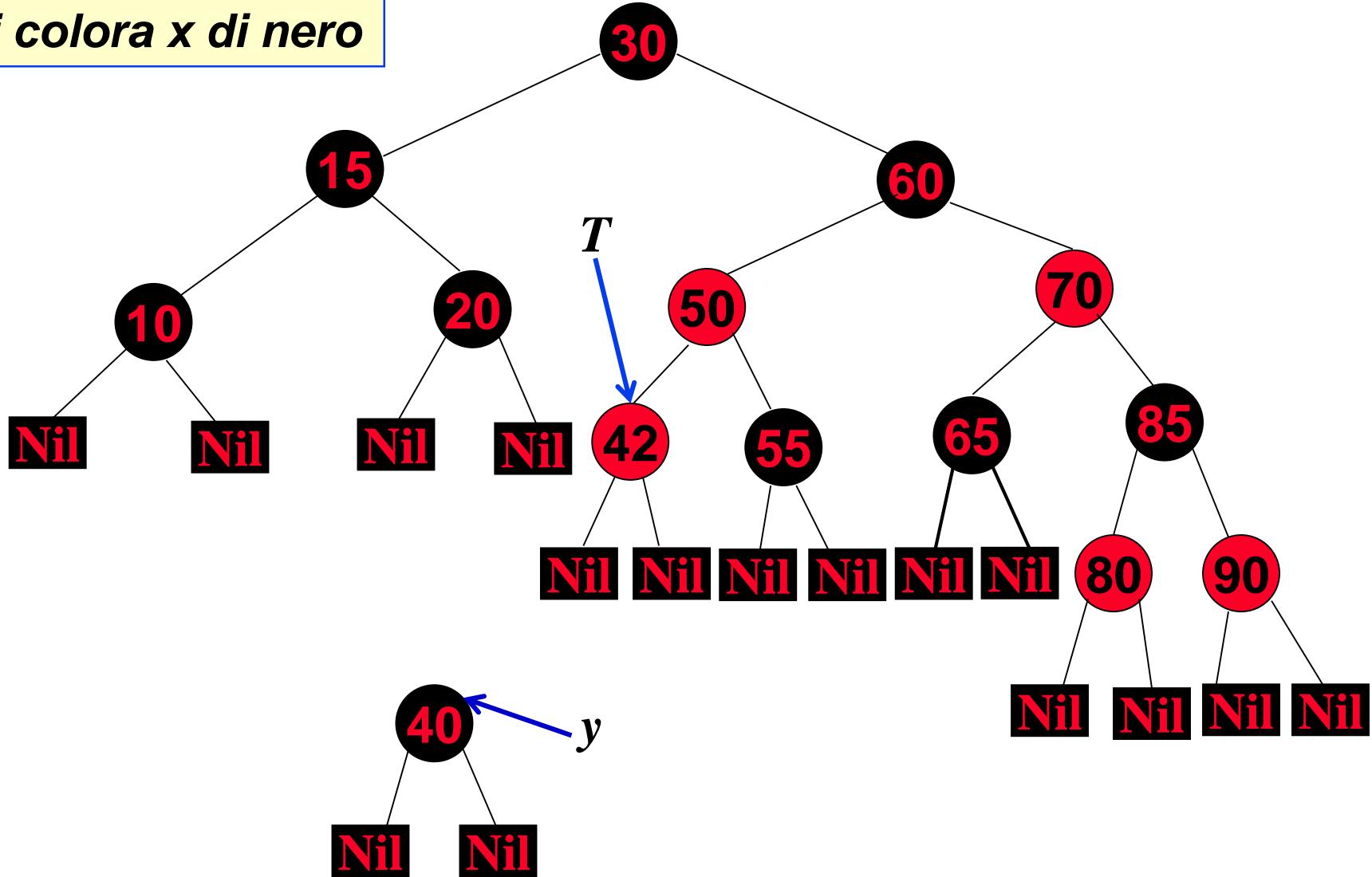
```
T = ruota-sx(T)
T->dx->color = T->color
T->color = T->sx->color
T->sx->color = black
T->sx->sx->color = black
return T
```

Cancellazione in RB: esempio



Cancellazione in RB: esempio

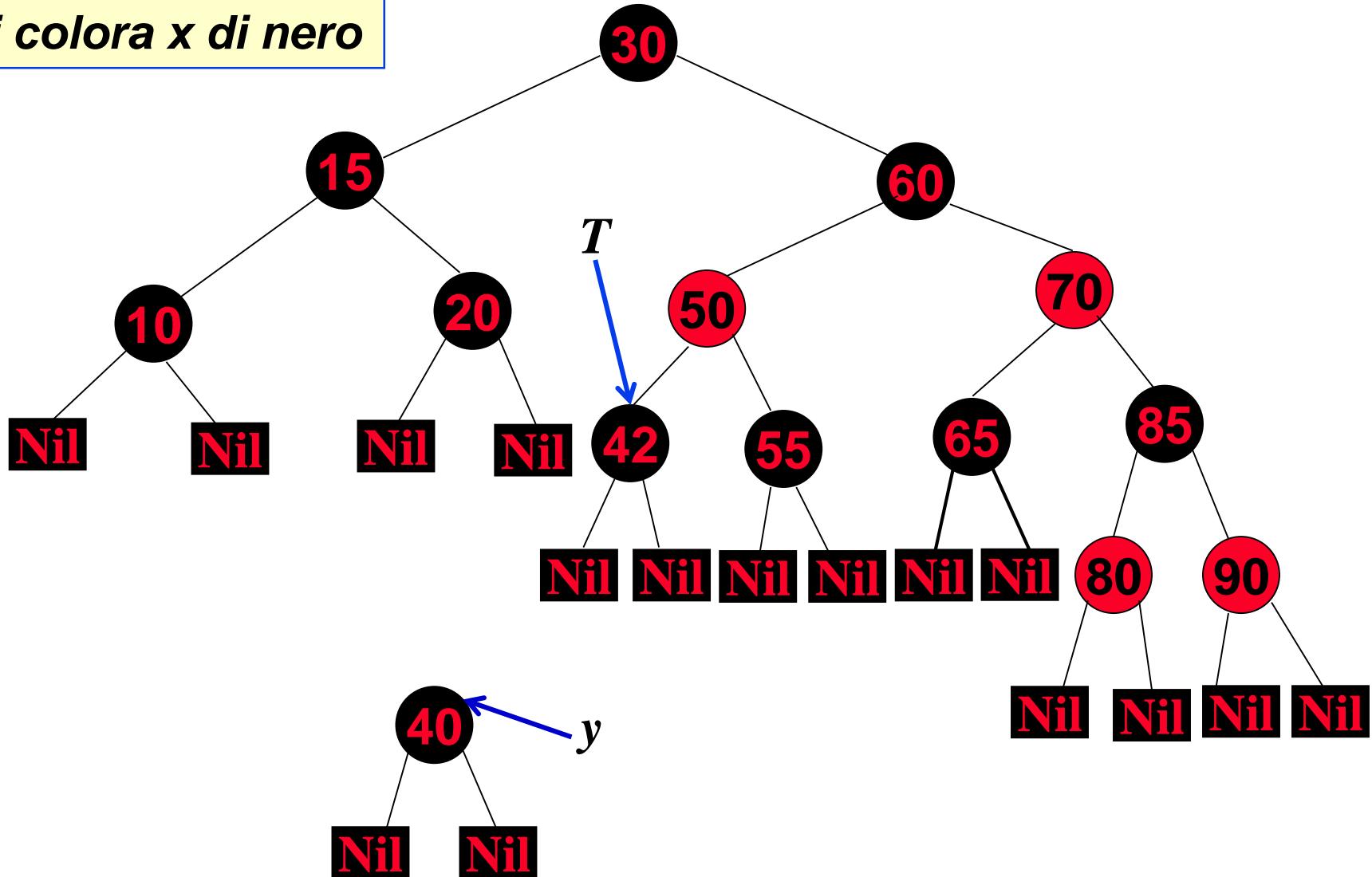
Il colore di x è rosso
e si colora x di nero



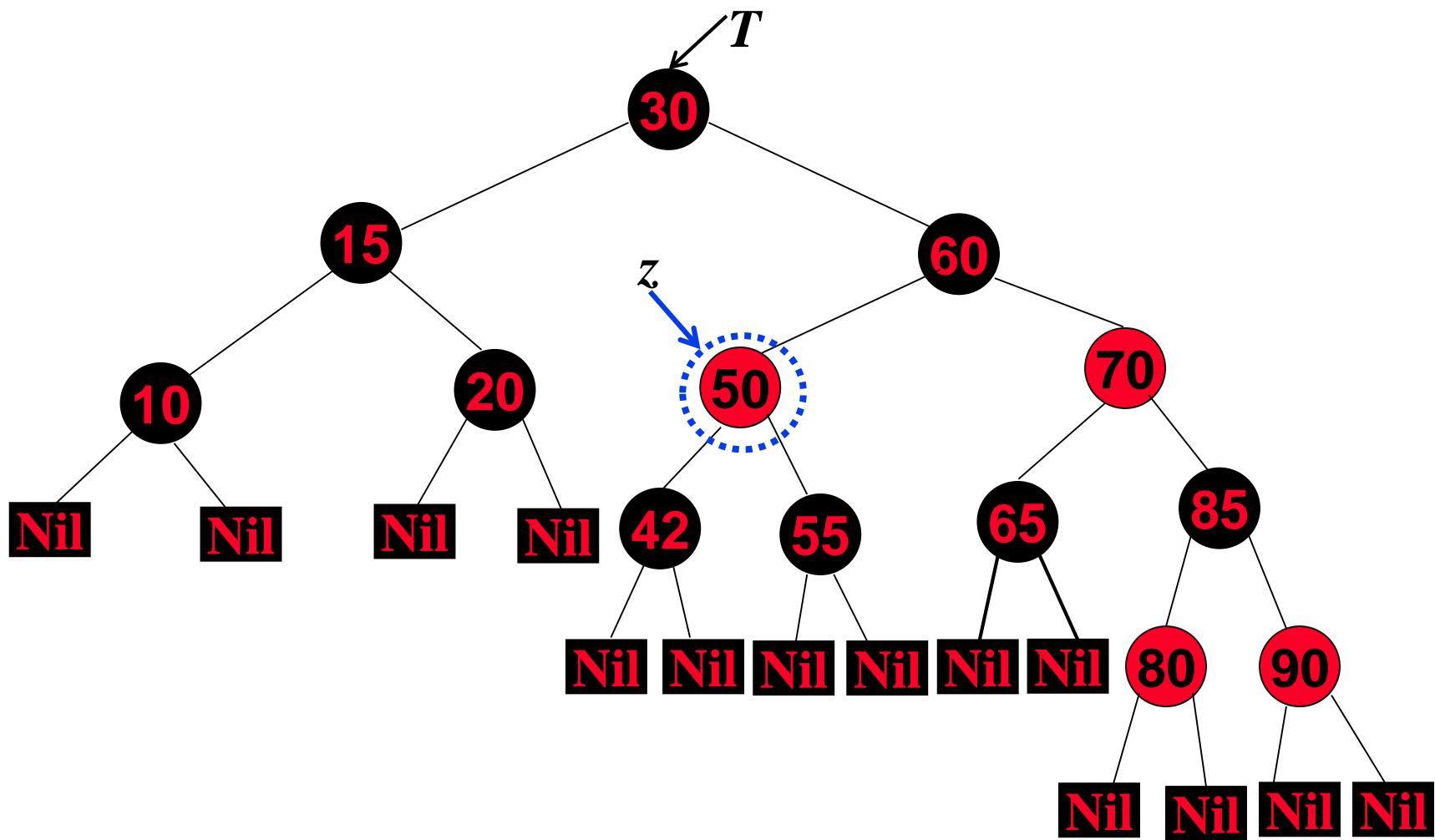
Cancellazione in RB: esempio

Il colore di x è rosso
e si colora x di nero

Fatto!



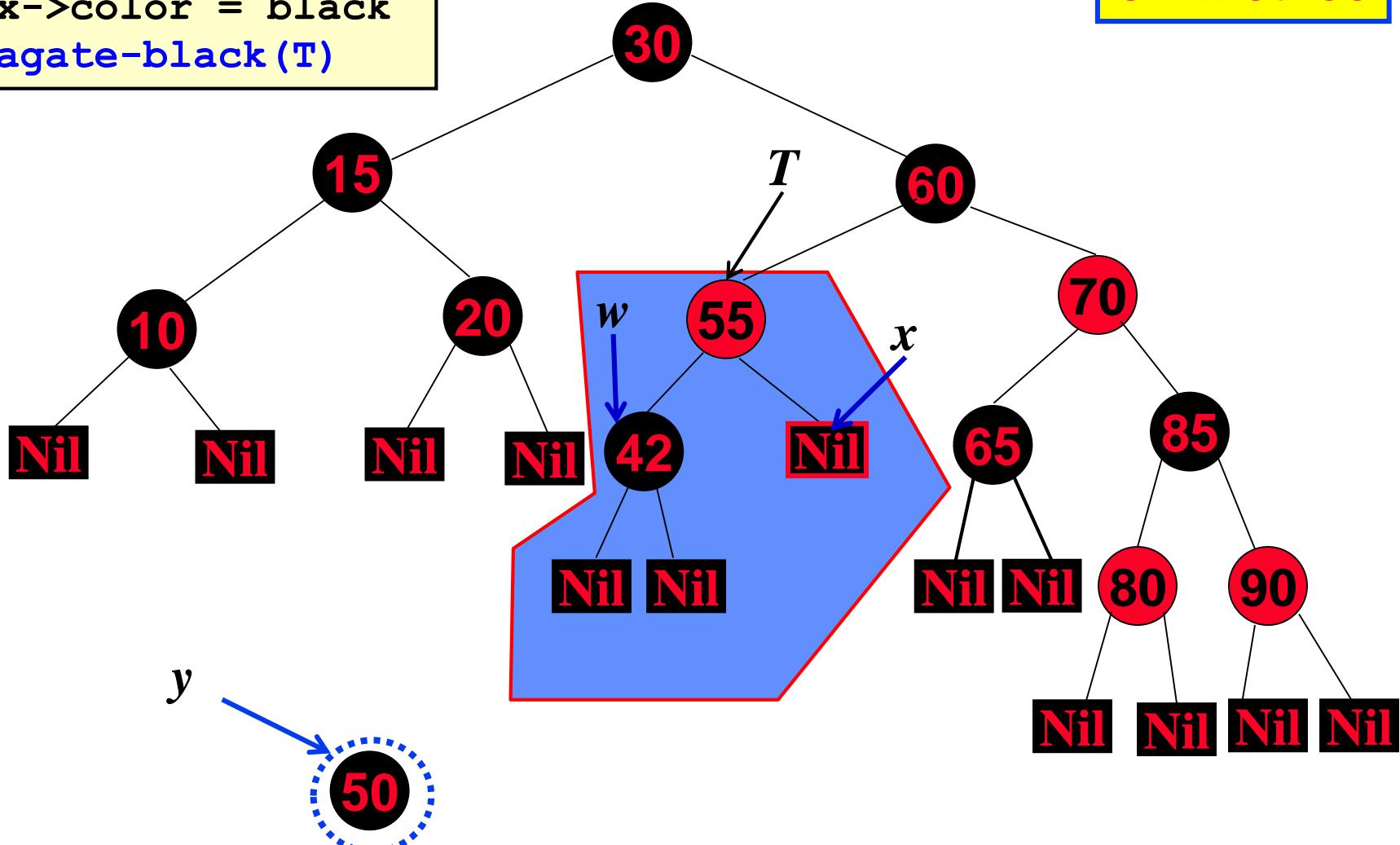
Cancellazione in RB: esempio II



Cancellazione in RB: esempio II

```
T->sx->color = red  
T->dx->color = black  
propagate-black (T)
```

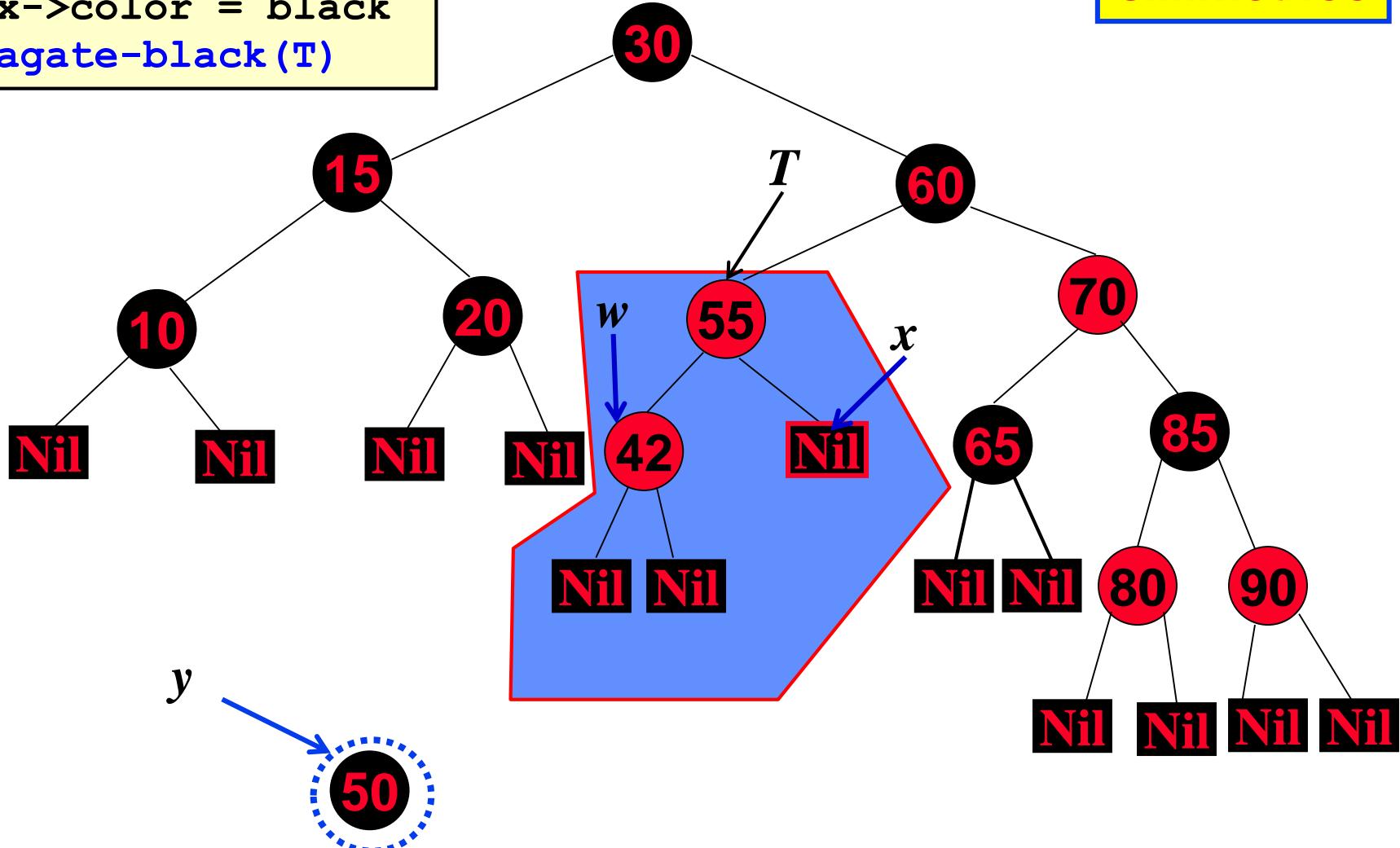
Caso II
simmetrico



Cancellazione in RB: esempio II

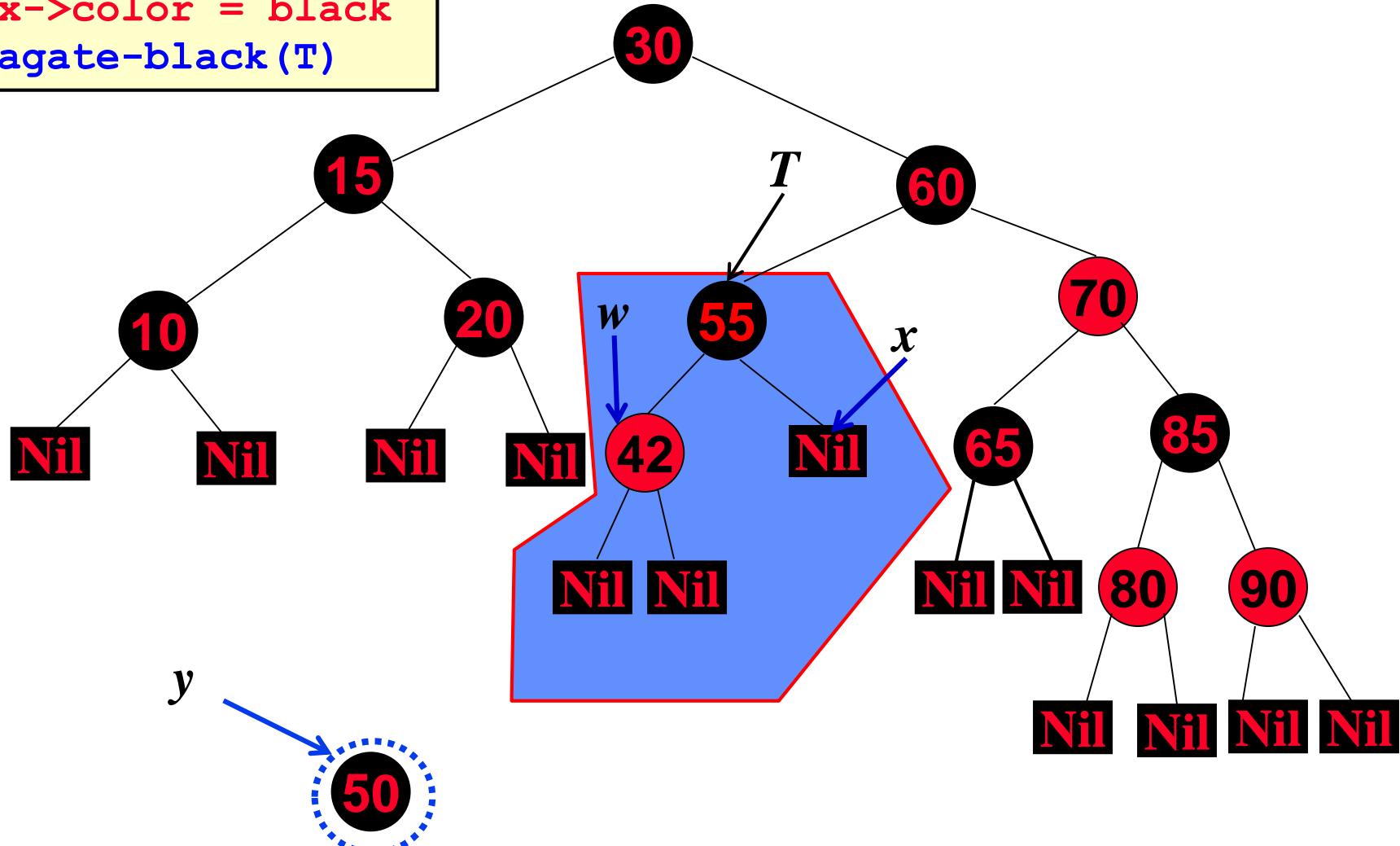
```
T->sx->color = red  
T->dx->color = black  
propagate-black (T)
```

Caso II
simmetrico



Cancellazione in RB: esempio II

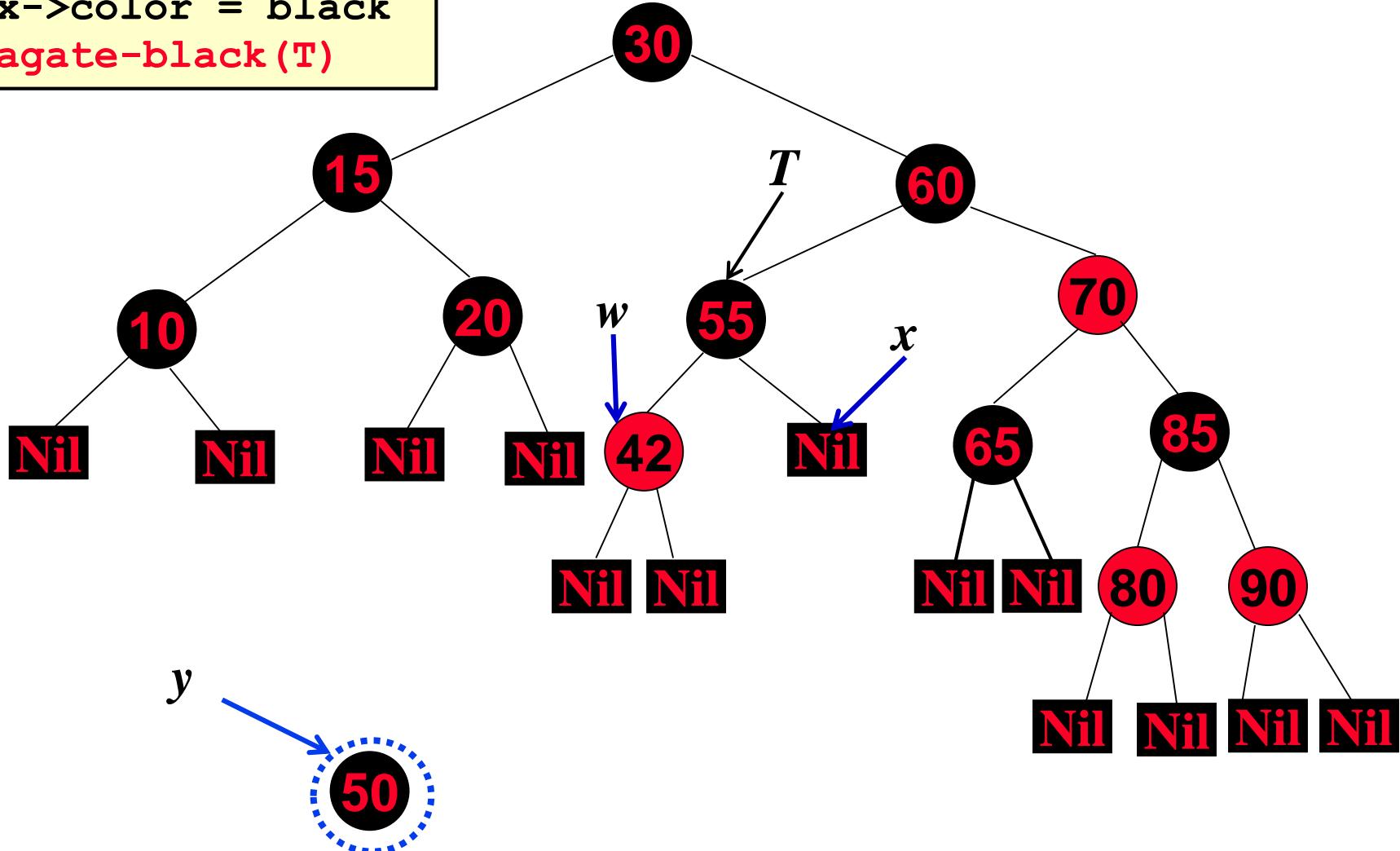
```
T->sx->color= red  
T->dx->color = black  
propagate-black (T)
```



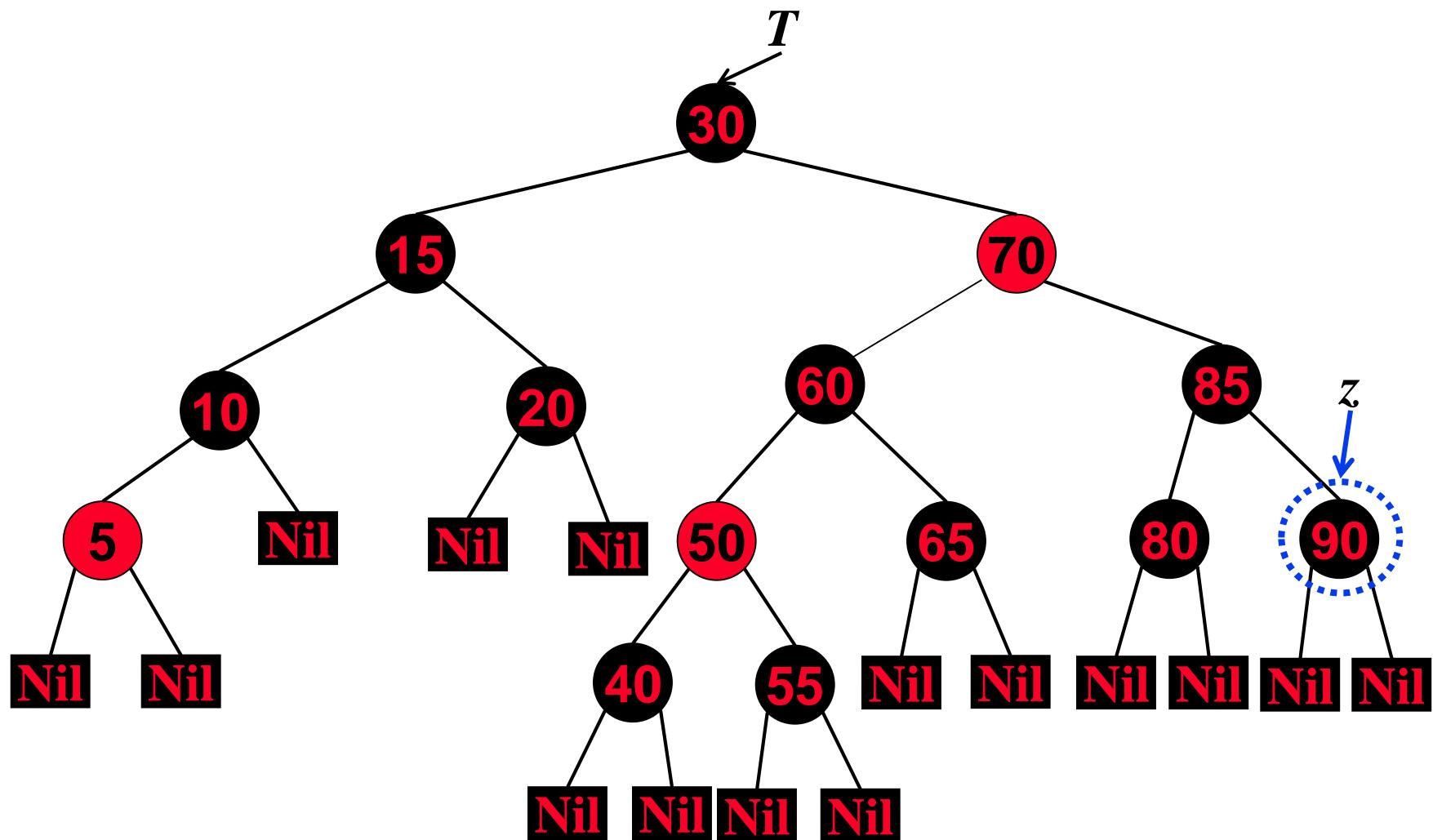
Cancellazione in RB: esempio II

```
T->sx->color = red  
T->dx->color = black  
propagate-black (T)
```

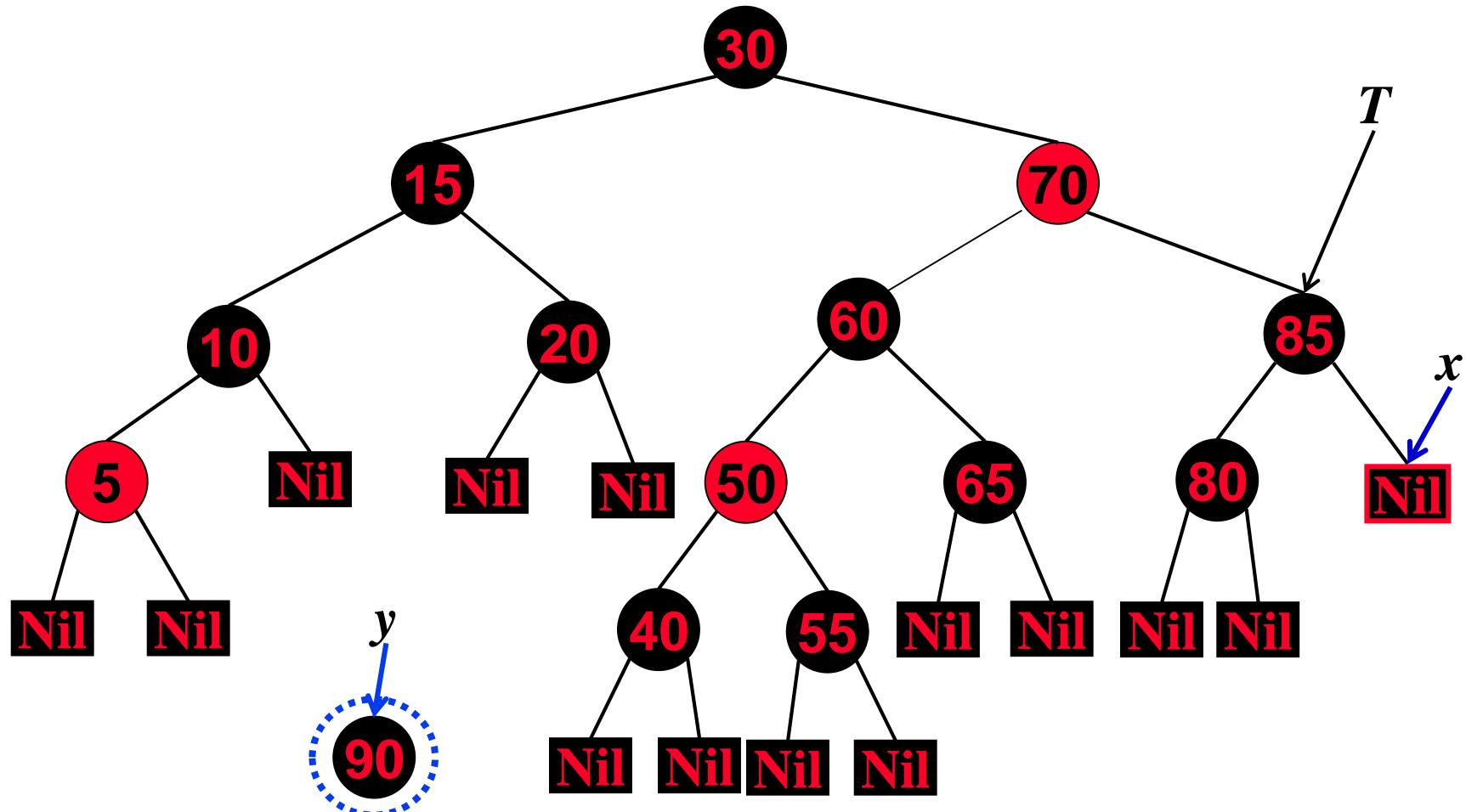
Fatto!



Cancellazione in RB: esempio III



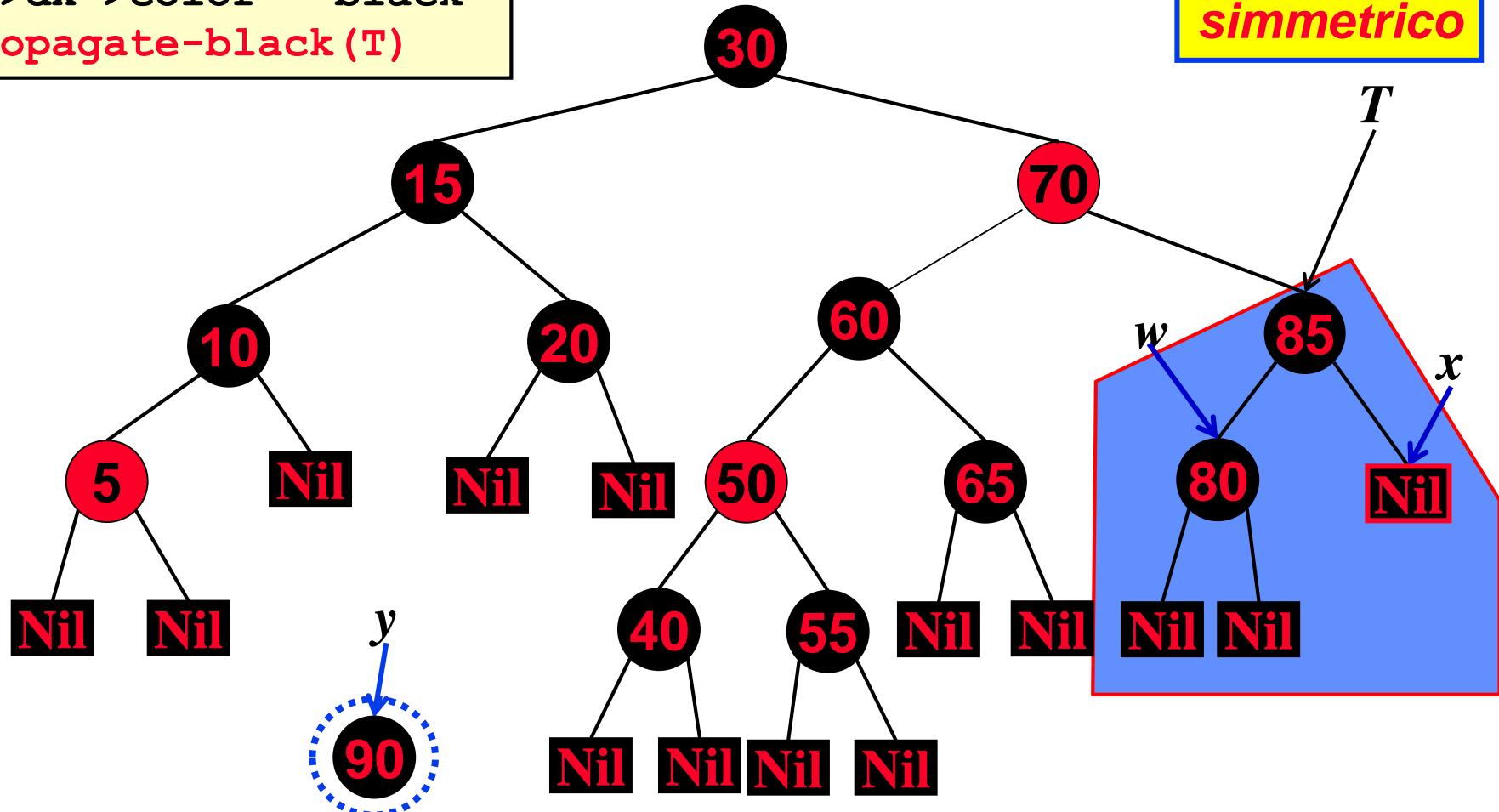
Cancellazione in RB: esempio III



Cancellazione in RB: esempio III

```
T->sx->color = red  
T->dx->color = black  
propagate-black (T)
```

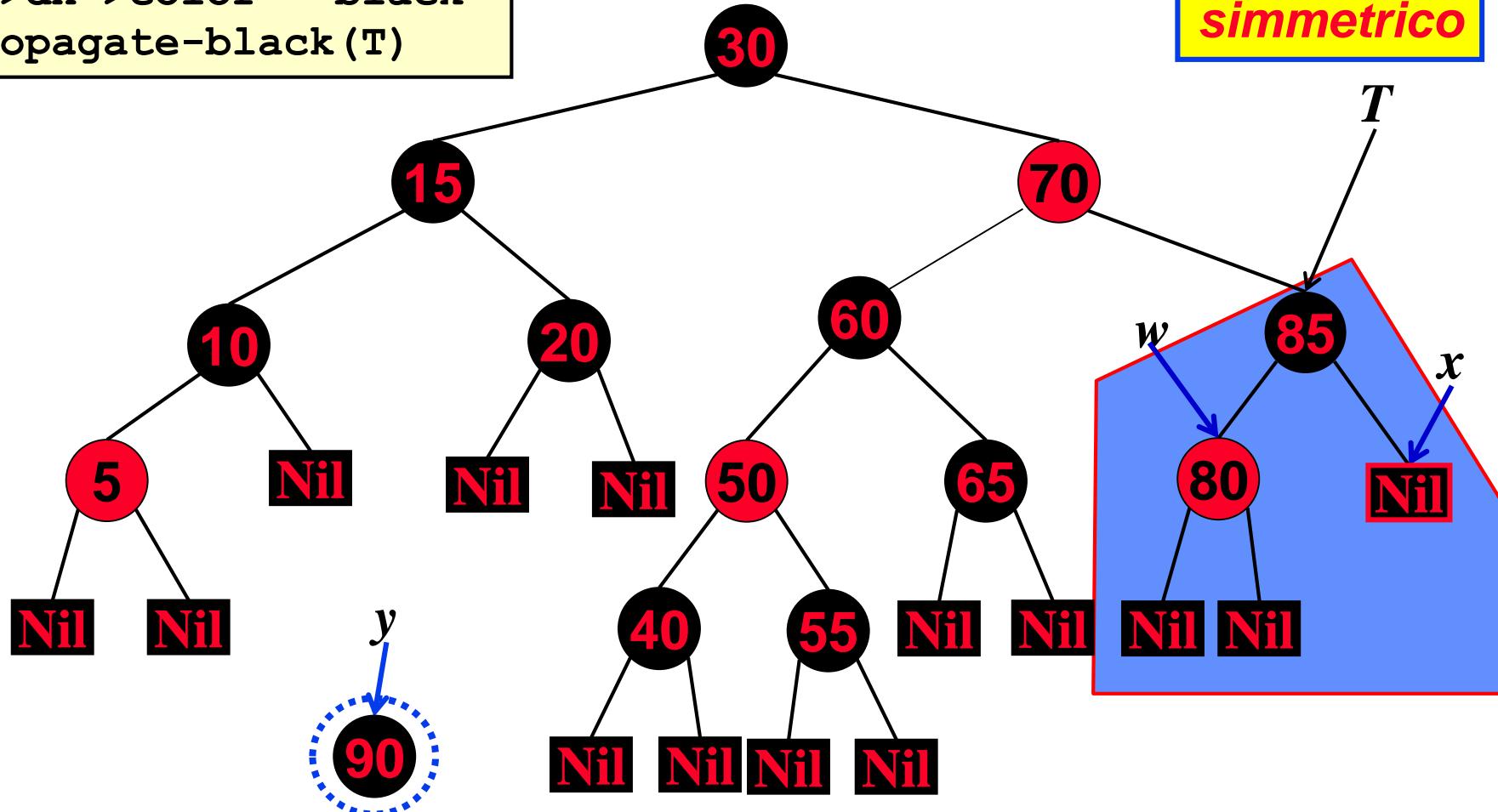
Caso II
simmetrico



Cancellazione in RB: esempio III

```
T->sx->color = red  
T->dx->color = black  
propagate-black (T)
```

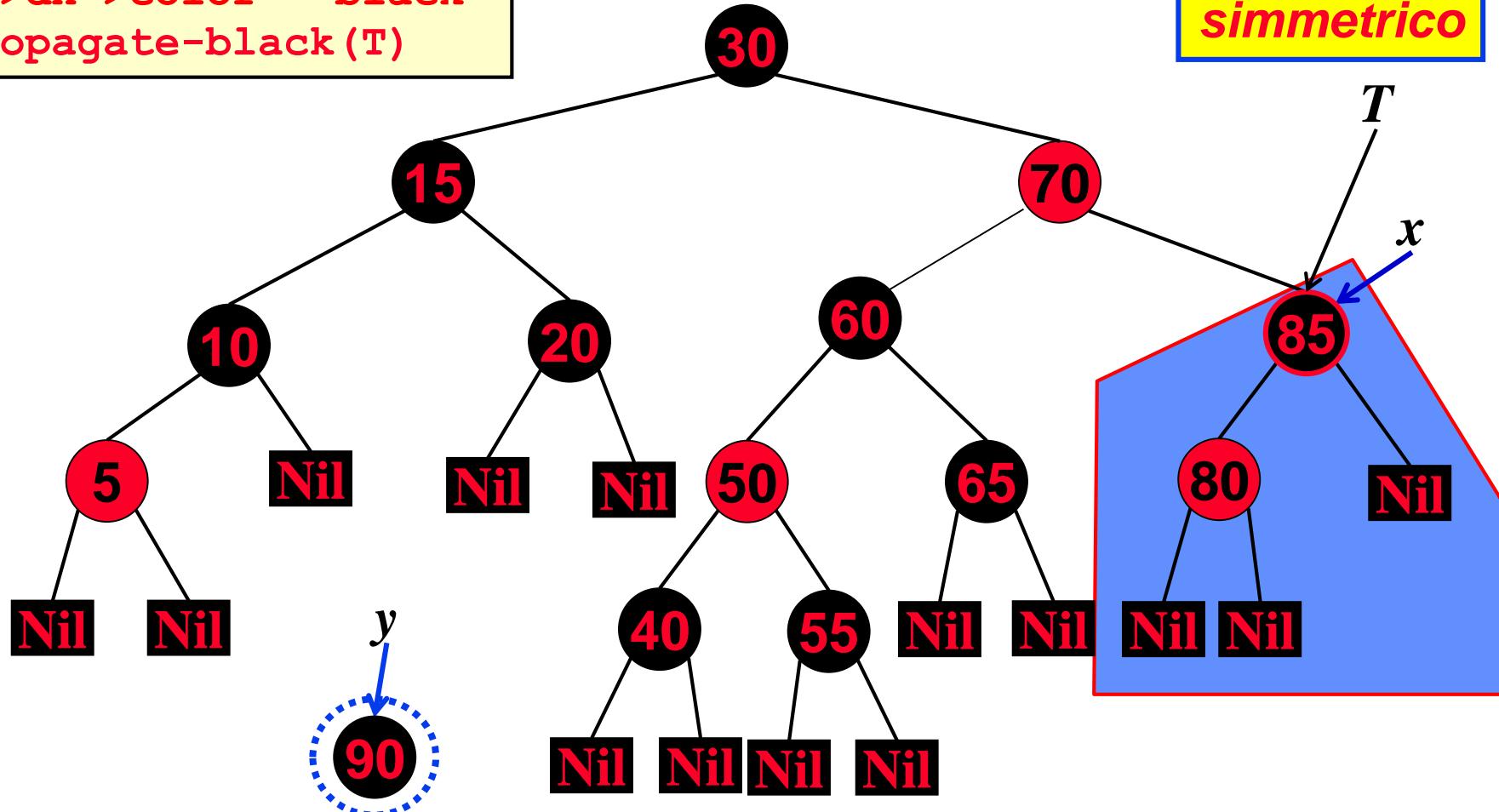
Caso II
simmetrico



Cancellazione in RB: esempio III

```
T->sx->color = red  
T->dx->color = black  
propagate-black (T)
```

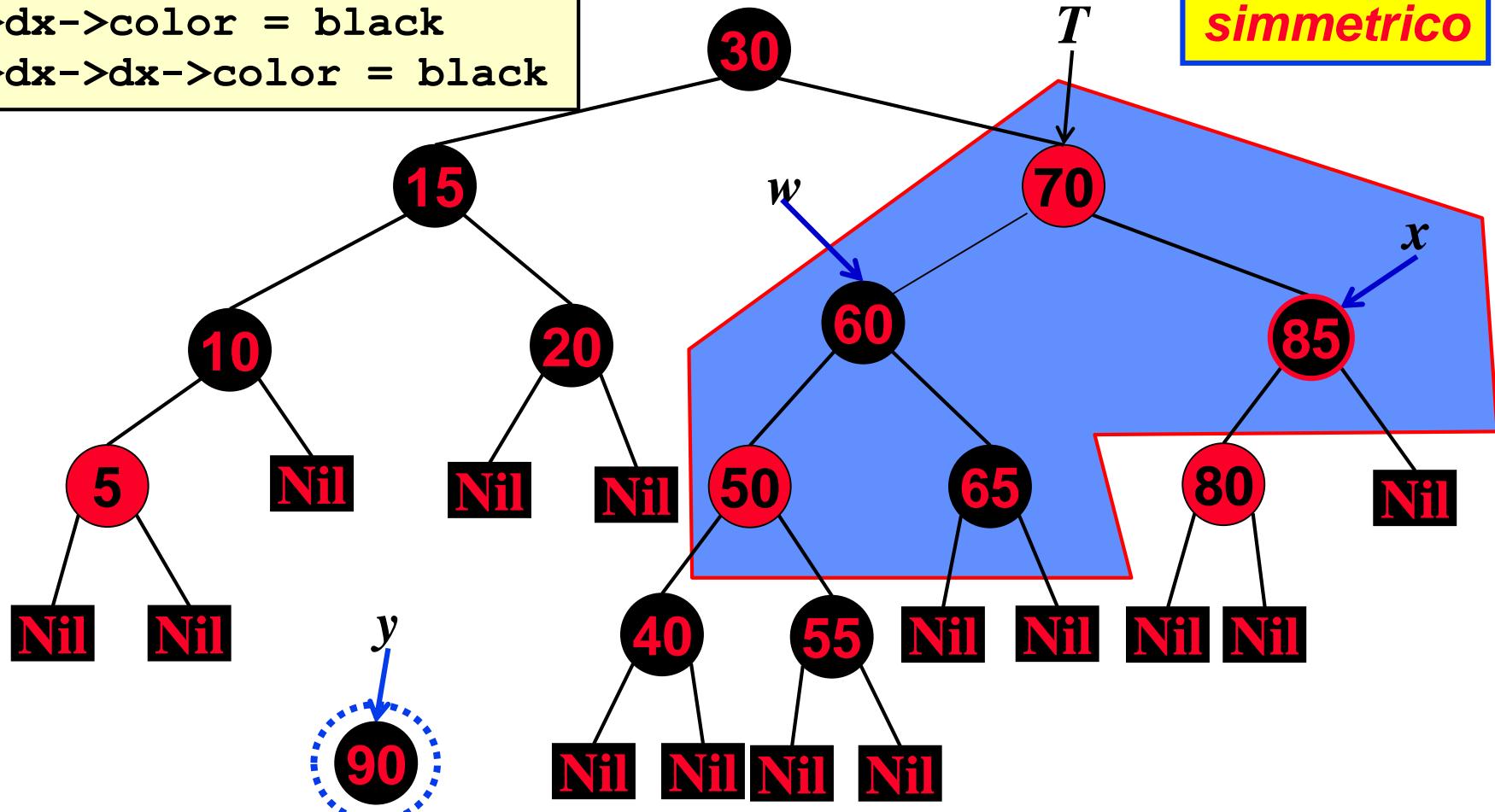
Caso II
simmetrico



Cancellazione in RB: esempio III

```
T = ruota-sx(T)
T->sx->color = T->color
T->color = T->dx->color
T->dx->color = black
T->dx->dx->color = black
```

**Caso IV
simmetrico**



Cancellazione in RB: esempio III

T = ruota-sx(T)

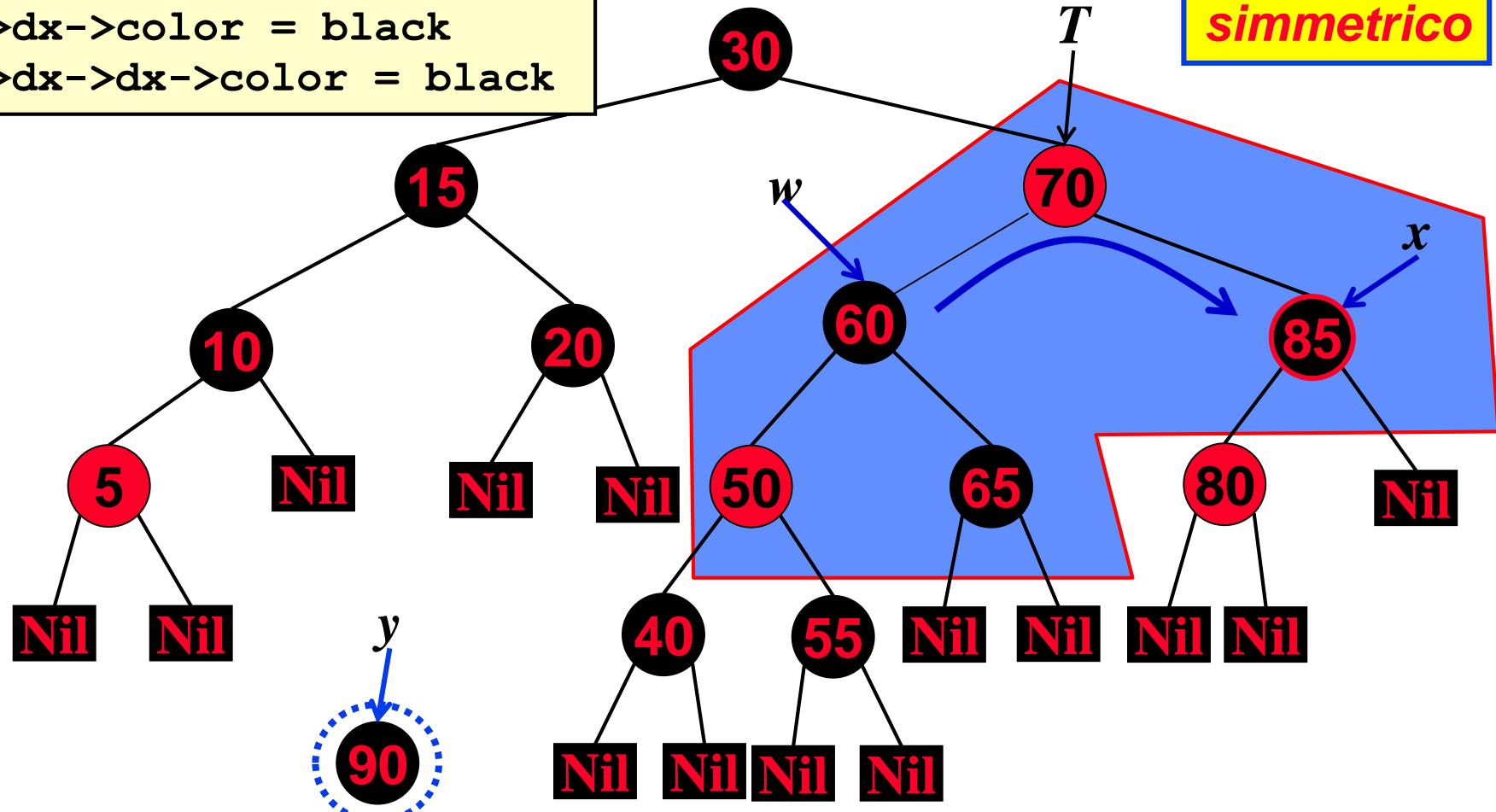
T->sx->color = T->color

T->color = T->dx->color

T->dx->color = black

T->dx->dx->color = black

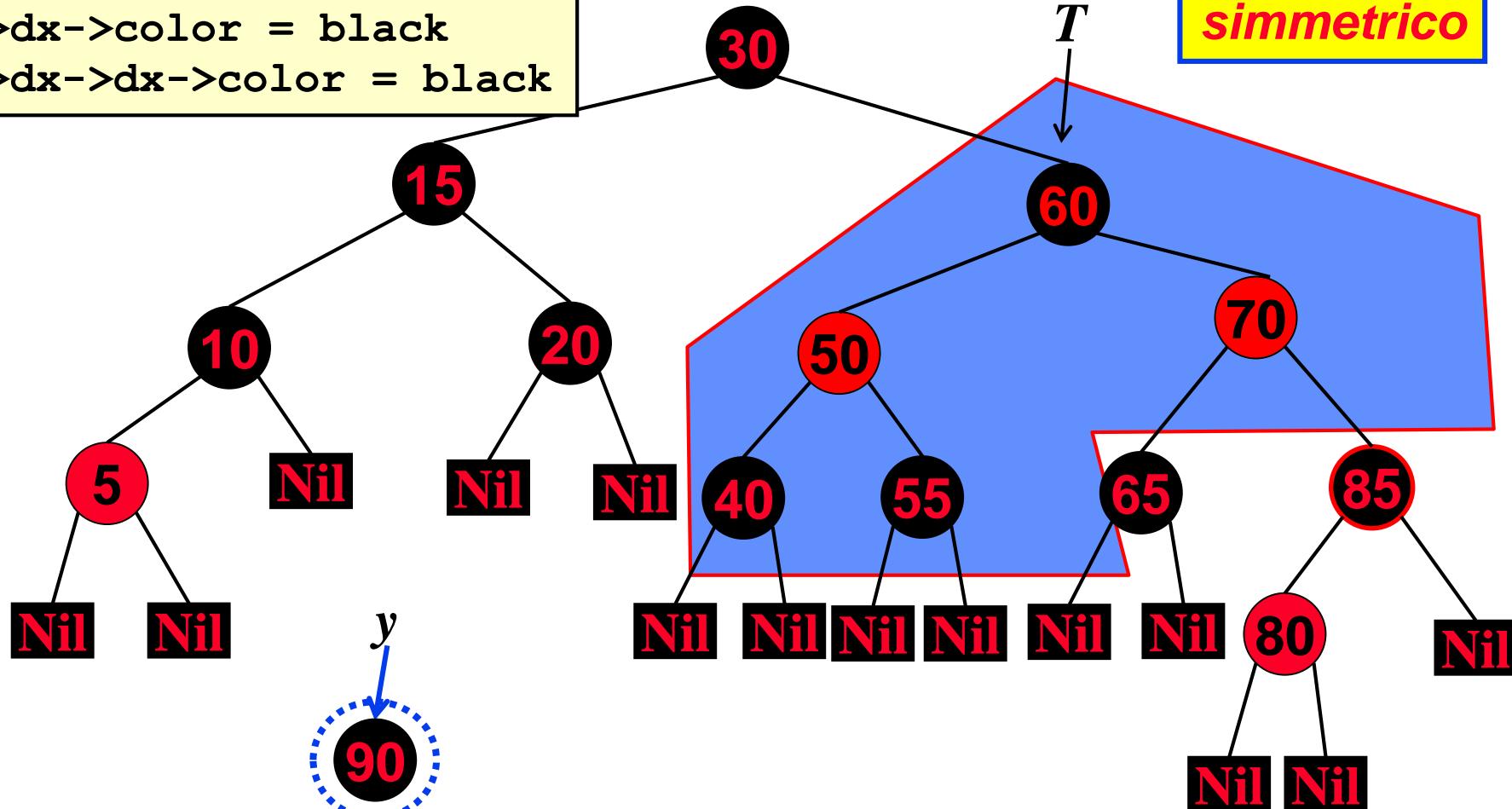
Caso IV
simmetrico



Cancellazione in RB: esempio III

```
T = ruota-sx(T)  
T->sx->color = T->color  
T->color = T->dx->color  
T->dx->color = black  
T->dx->dx->color = black
```

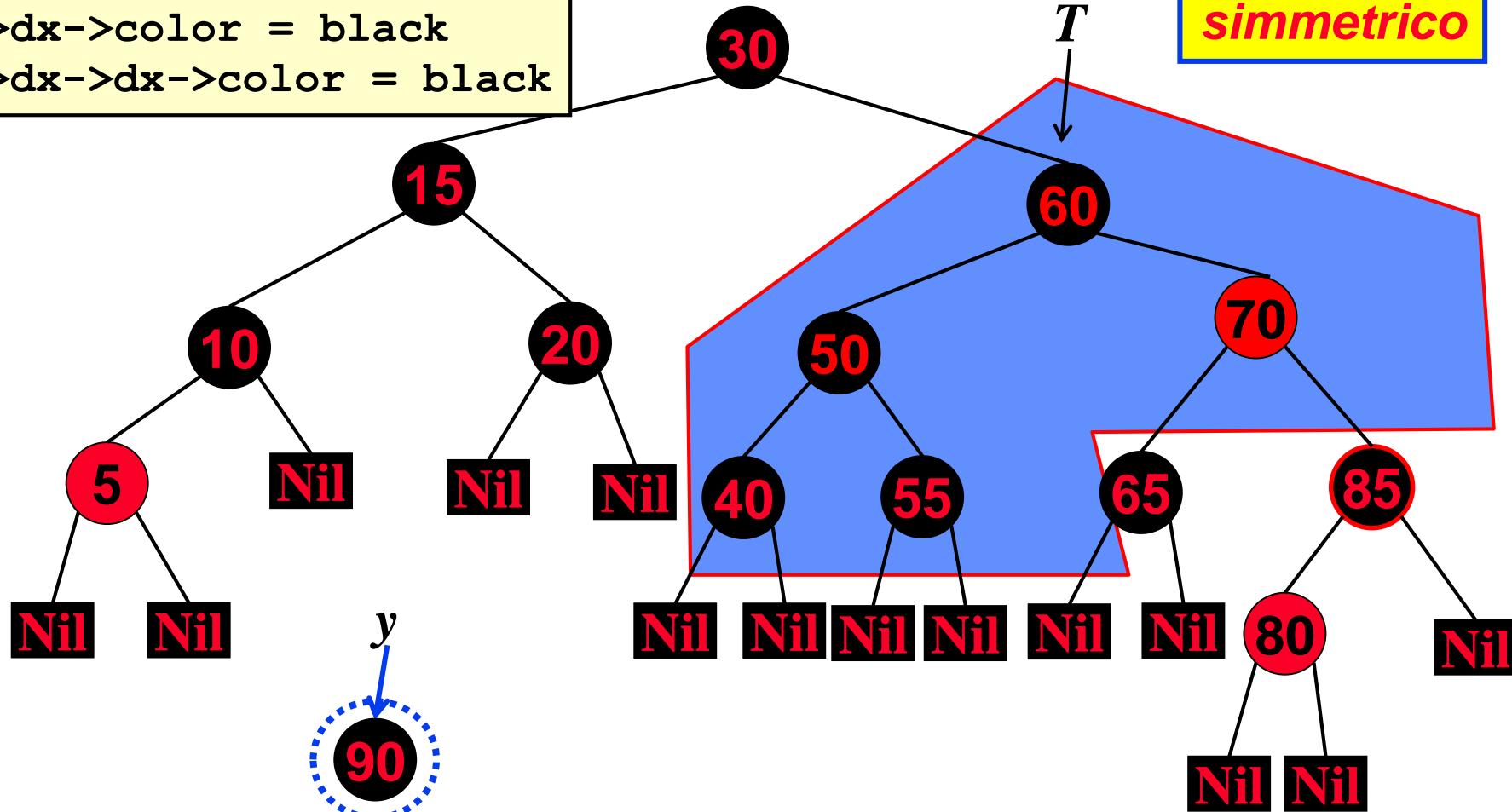
**Caso IV
simmetrico**



Cancella zoine in RB: esempio III

```
T = ruota-sx(T)  
T->sx->color = T->color  
T->color = T->dx->color  
T->dx->color = black  
T->dx->dx->color = black
```

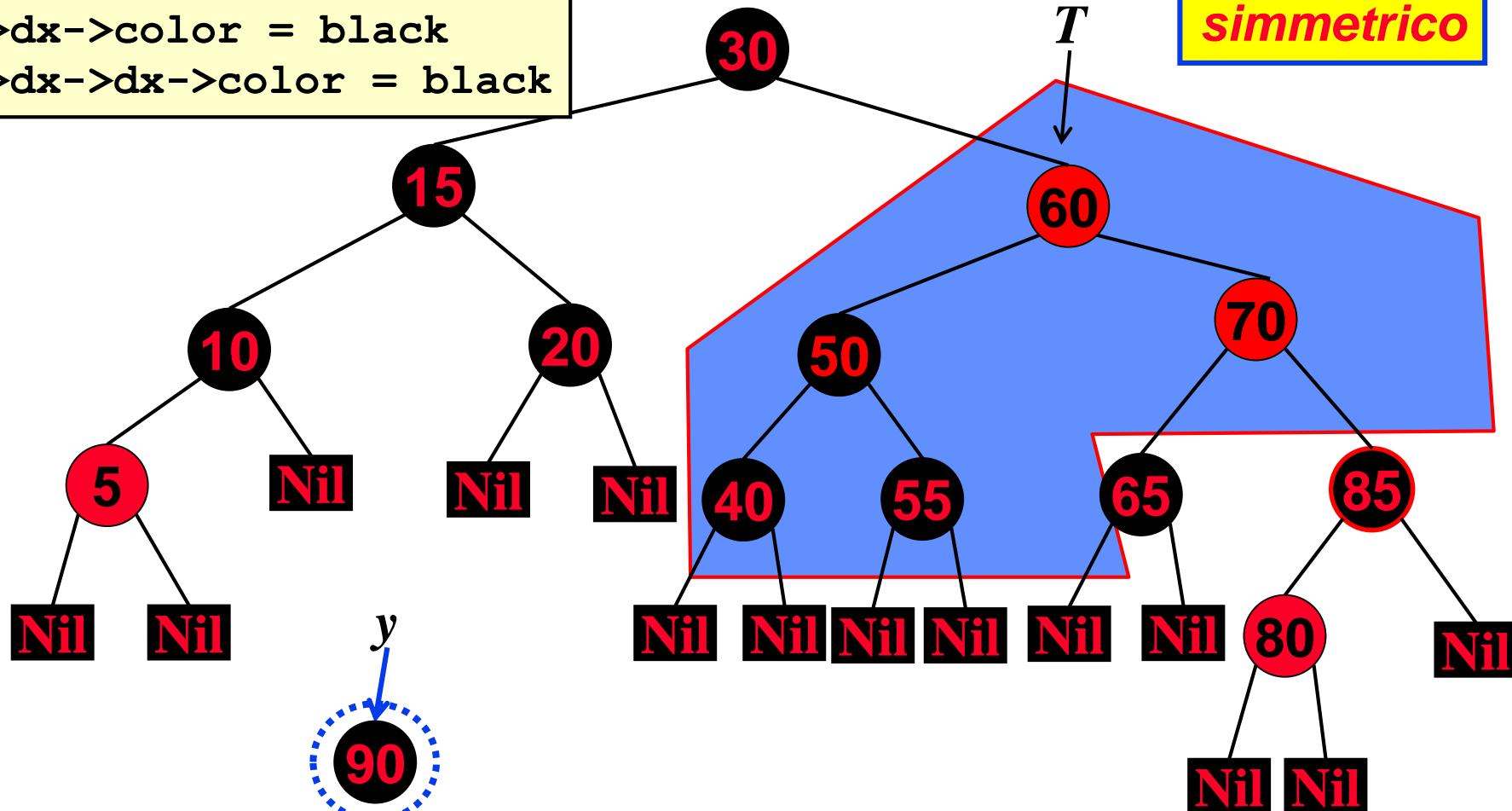
Caso IV
simmetrico



Cancella zoine in RB: esempio III

```
T = ruota-sx(T)  
T->sx->color = T->color  
T->color = T->dx->color  
T->dx->color = black  
T->dx->dx->color = black
```

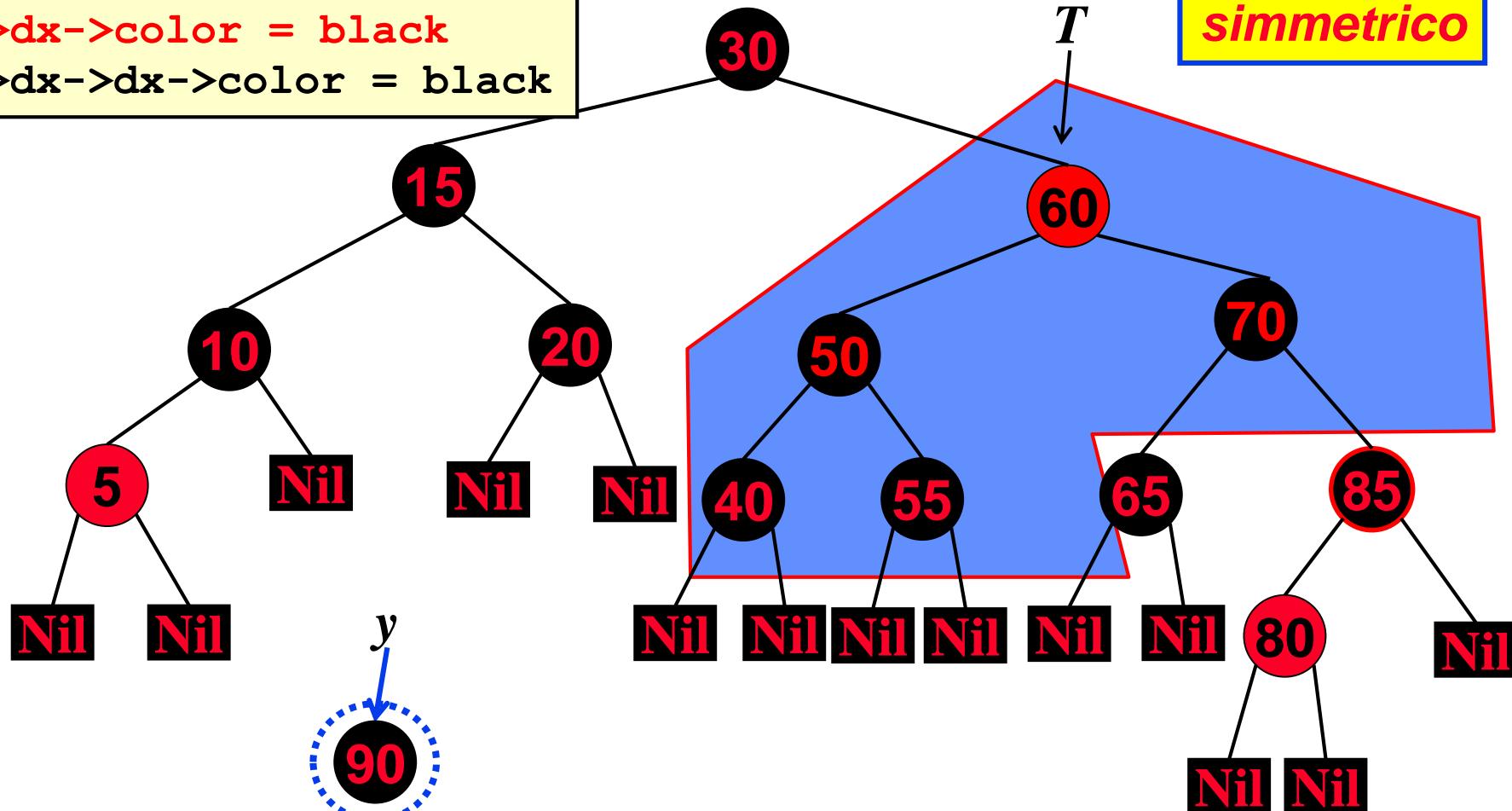
**Caso IV
simmetrico**



Cancellazione in RB: esempio III

```
T = ruota-sx(T)  
T->sx->color = T->color  
T->color = T->dx->color  
T->dx->color = black  
T->dx->dx->color = black
```

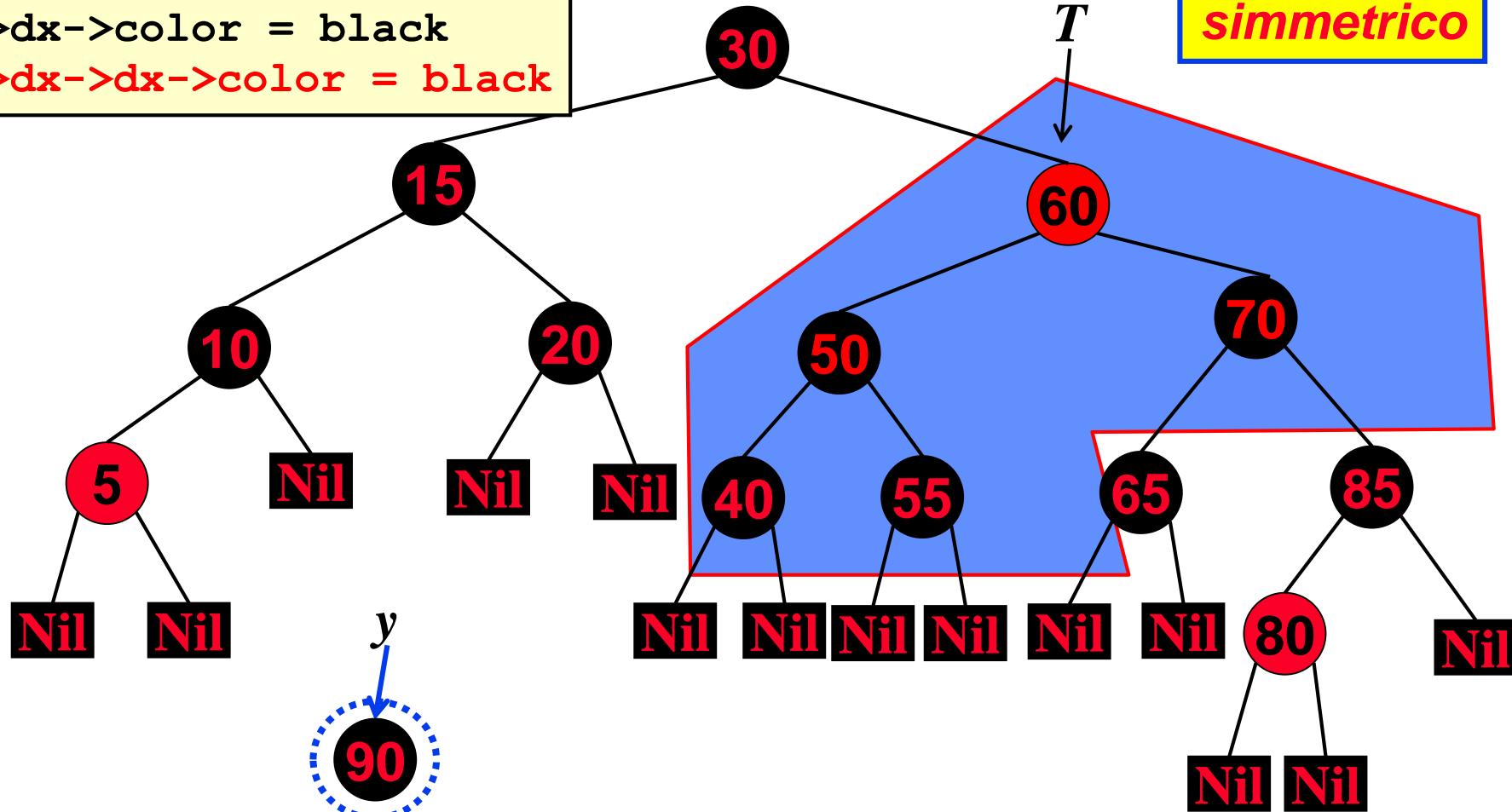
**Caso IV
simmetrico**



Cancella zoine in RB: esempio III

```
T = ruota-sx(T)  
T->sx->color = T->color  
T->color = T->dx->color  
T->dx->color = black  
T->dx->dx->color = black
```

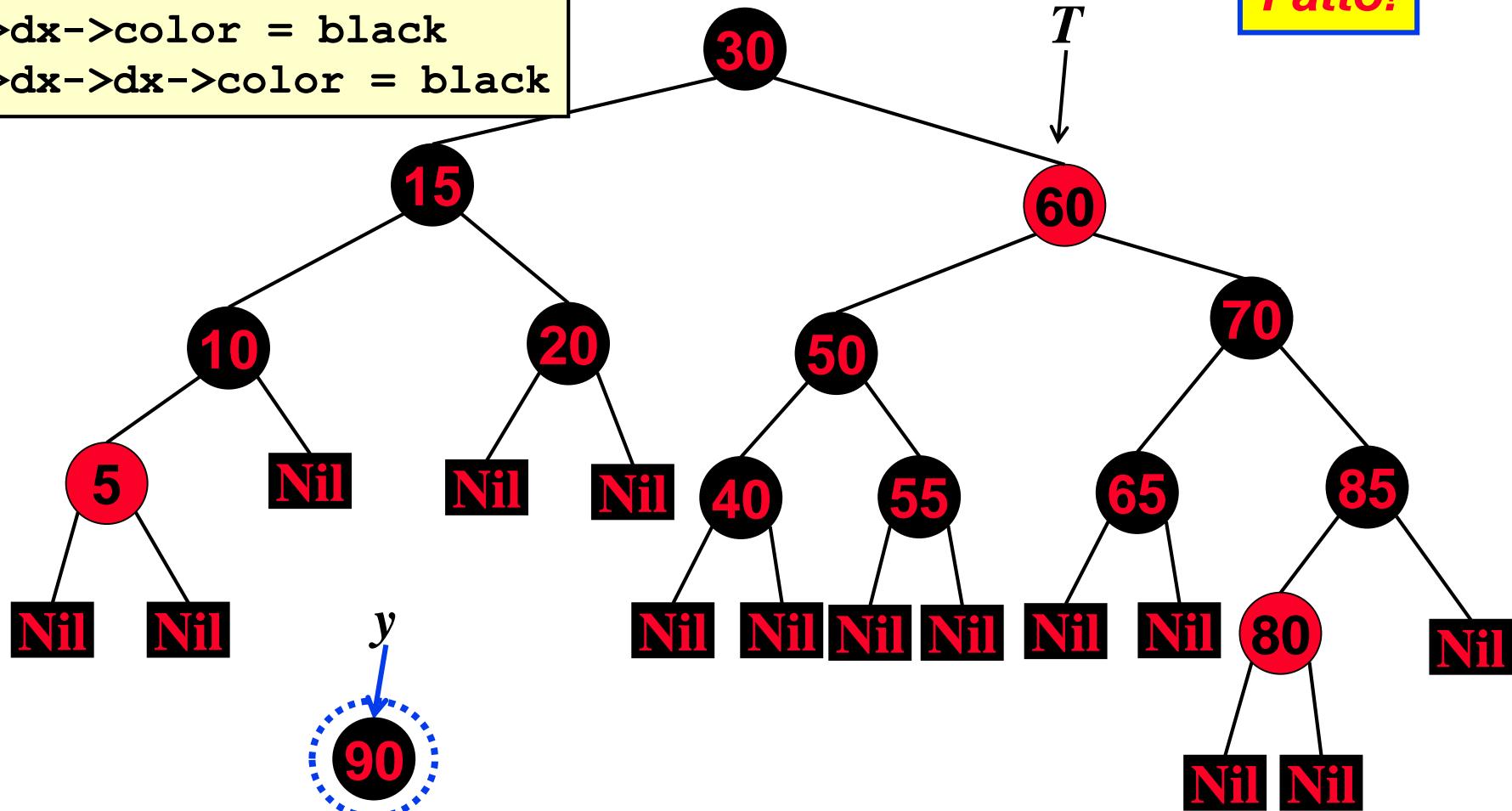
Caso IV
simmetrico



Cancella zoine in RB: esempio III

```
T = ruota-sx(T)  
T->sx->color = T->color  
T->color = T->dx->color  
T->dx->color = black  
T->dx->dx->color = black
```

Fatto!



Cancellazione in RB

- L'operazione di cancellazione è concettualmente complicata!
- Ma efficiente:
 - il caso 4 è risolutivo e applica 1 sola rotazione
 - il caso 3 applica una rotazione e passa nel caso 4
 - il caso 2 non fa rotazioni e passa in uno qualsiasi dei casi *ma salendo lungo il percorso di cancellazione*
 - il caso 1 fa una rotazione e passa in uno degli altri casi (ma se va nel caso 2, il caso 2 termina)

Quindi

al massimo vengono eseguite 3 rotazioni



Massimo Benerecetti

Tabelle Hash

Lezione n.#

Parole chiave:

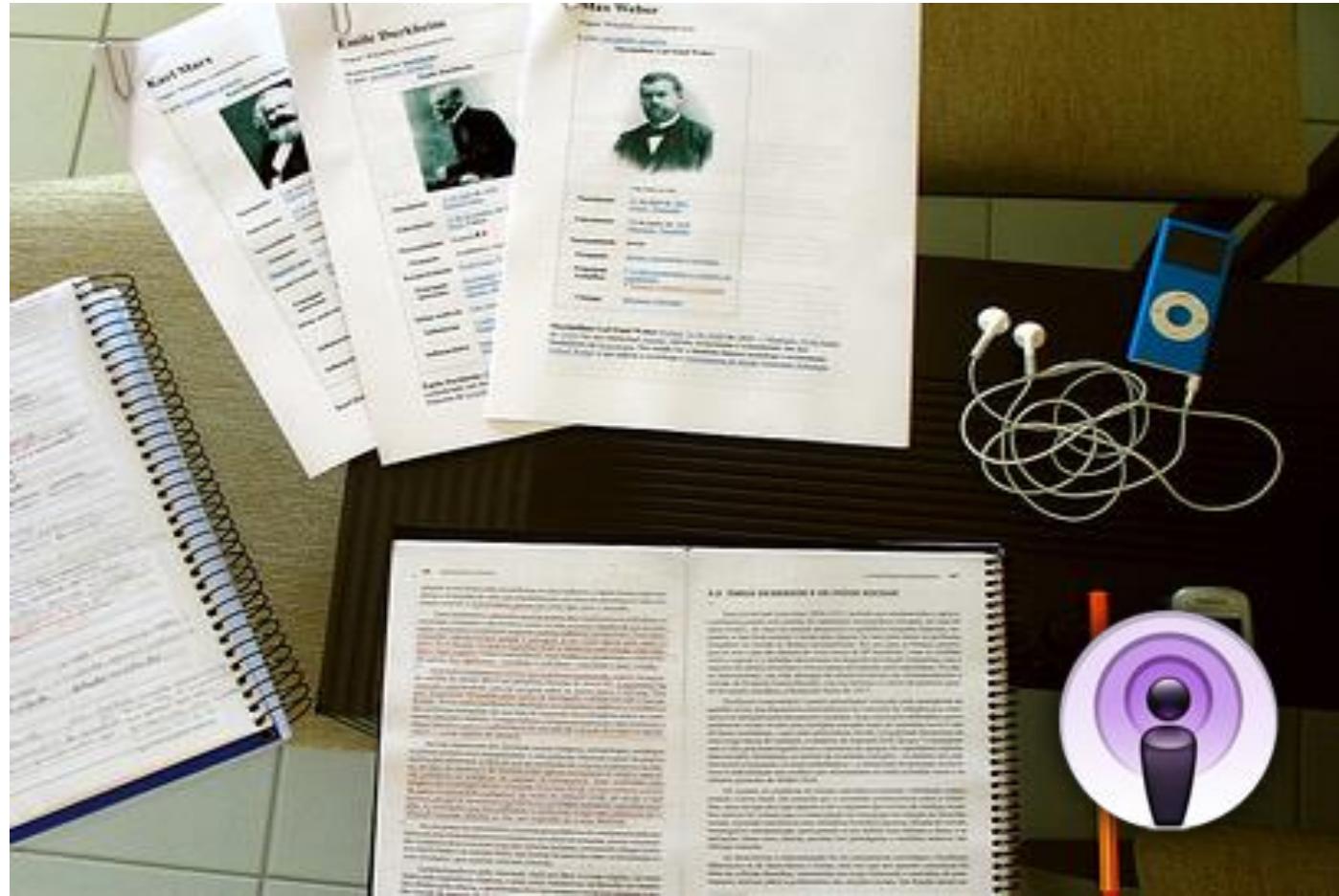
Corso di Laurea:
Informatica

Insegnamento:

Algoritmi e
Strutture Dati I

Email Docente:
bene@na.infn.it

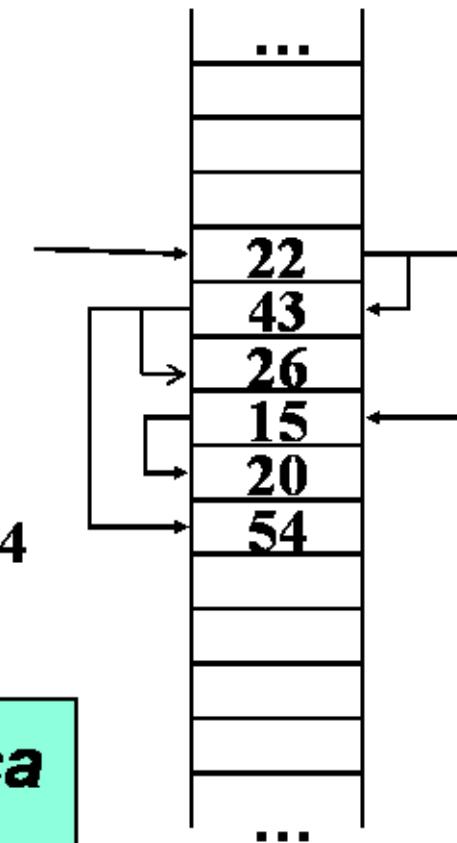
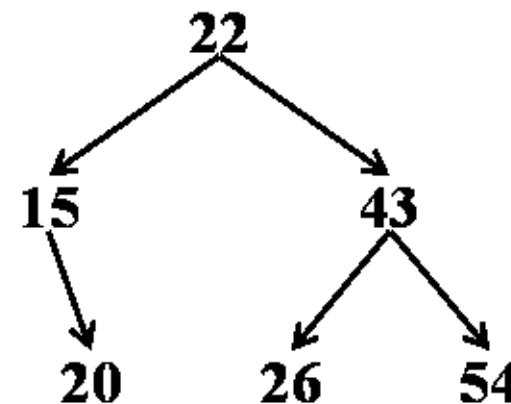
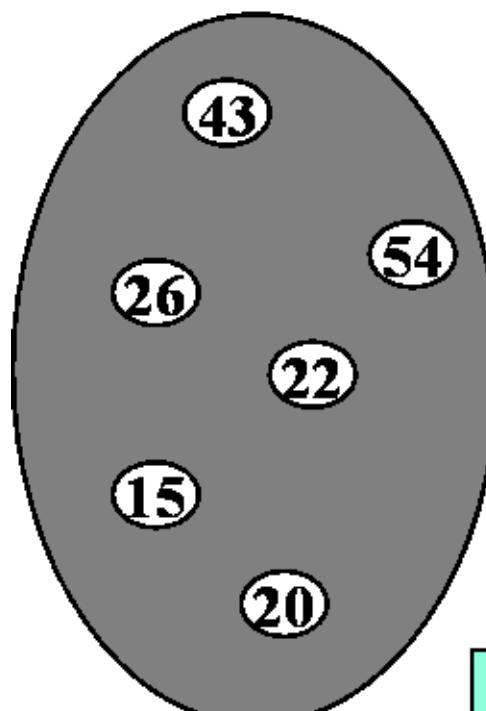
A.A. 2009-2010



Rappresentazione di insiemi dinamici

- Gli insiemi dinamici possono essere rappresentati con varie strutture dati, ciascuna con caratteristiche di flessibilità e di prestazioni differenti.
- Array, liste ed alberi sono tra le rappresentazioni più diffuse.
- Gli alberi binari di ricerca bilanciati offrono un buon compromesso tra flessibilità e prestazioni, garantendo tempi di ricerca logaritmici rispetto al numero di elementi.
- Rinunciando ad un po' della flessibilità degli alberi, è possibile però ottenere strutture dati con migliori prestazioni per la ricerca degli elementi.

Insiemi dinamici come alberi



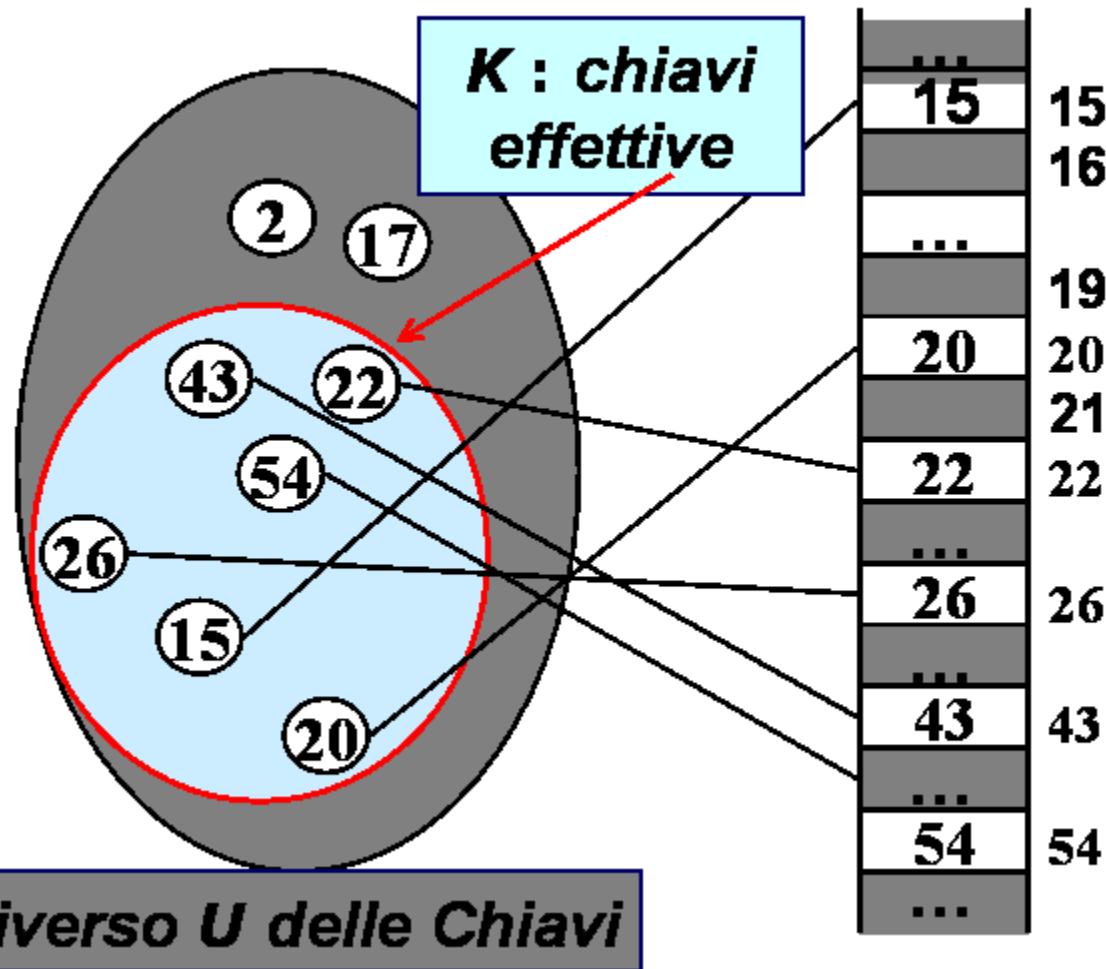
*Tempo di ricerca
O(log n)
in alberi bilanciati*

Rappresentazione ad **albero** di un insieme dinamico di chiavi prese da un universo **U**.

Tabelle ad accesso diretto

- Una **tabella ad accesso diretto** è una struttura dati che *supporta SOLO* le operazioni di:
 - inserimento
 - ricerca
 - cancellazione
- in tempo che è **$O(1)$**
- Non supporta direttamente Minimo, Massimo, Successore, Predecessore (cioè gli Ordinamenti)

Insiemi dinamici come tabelle ad accesso diretto



Rappresentazione con **tabella ad accesso diretto** per un insieme dinamico di chiavi prese da un universo **U**.

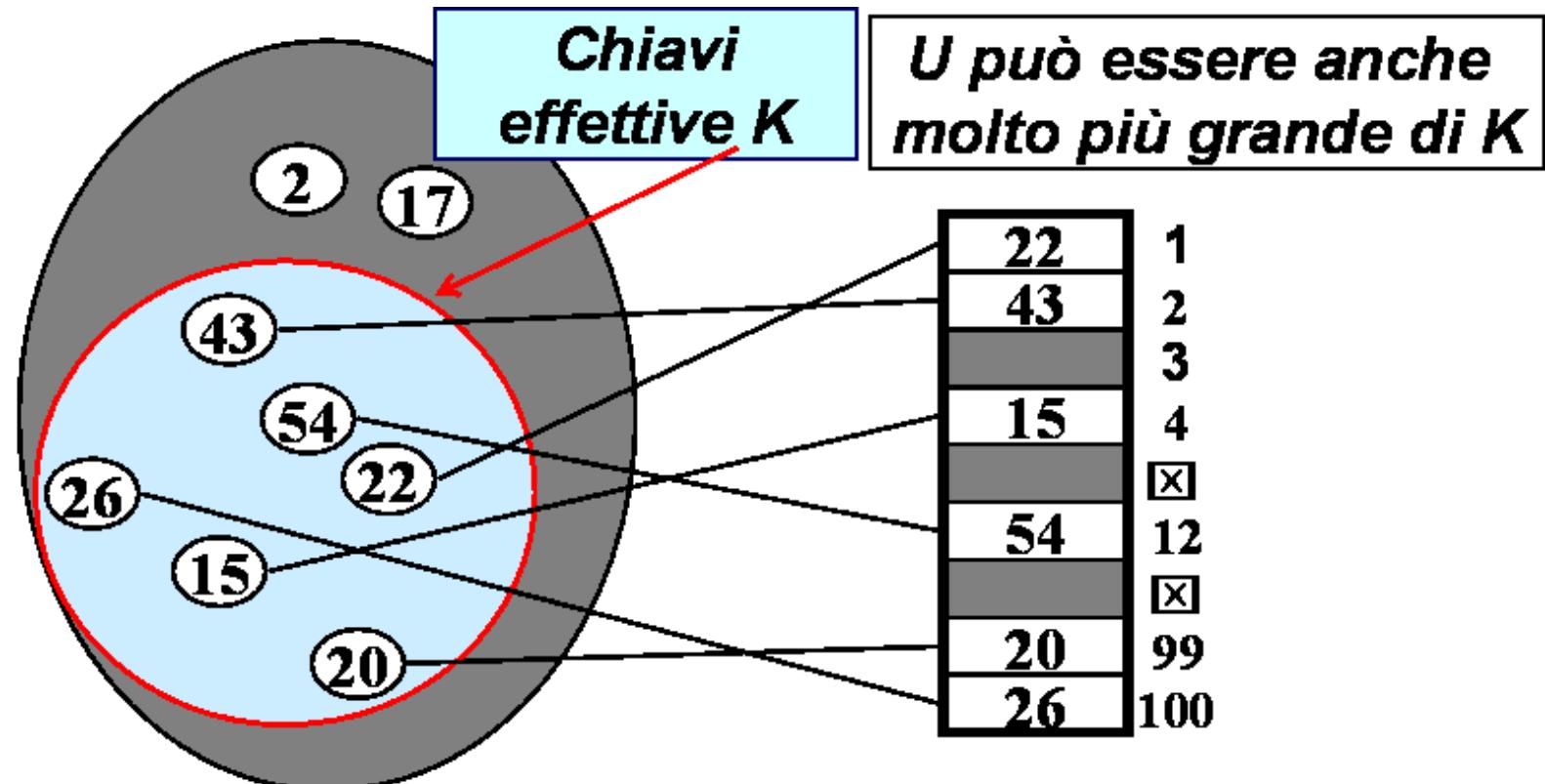
- La più semplice implementazione di una **tabella ad accesso diretto** è un **array**.
- Per memorizzare gli interi a 16-bit possiamo utilizzare un array **A** di dimensione **2^{16}** .
- Le operazioni potrebbero essere definite come segue:
 - **inserisci(i)** : $A[i] = A[i] + 1$
 - **ricerca(i)** : $(A[i] > 0) ?$
 - **cancella(i)** : $A[i] = A[i] - 1$

...	15
0	16
0	19
...	20
1	21
0	22
1	26
...	43
0	43
...	54
1	...

Inserisci: 20, 22, 26, 43, 54, 43

Esempio di funzione indice e tabella ad accesso diretto.

- Se le **chiavi** sono **stringhe di 8 lettere** alfabetiche, ci sono **26⁸** (o circa 200 miliadri) di possibili chiavi [circa 200 'giga' di chiavi].
- Quasi sempre solo una **piccola frazione** di queste chiavi verrà effettivamente **impiegata**.
- Ne risulterebbe la necessità di un **array molto grande**, ma con **pochissime celle occupate**.
- Ci serve, quindi, una **soluzione migliore!**



Rappresentazione di una **tabella hash** per un insieme dinamico di chiavi prese da un universo U .

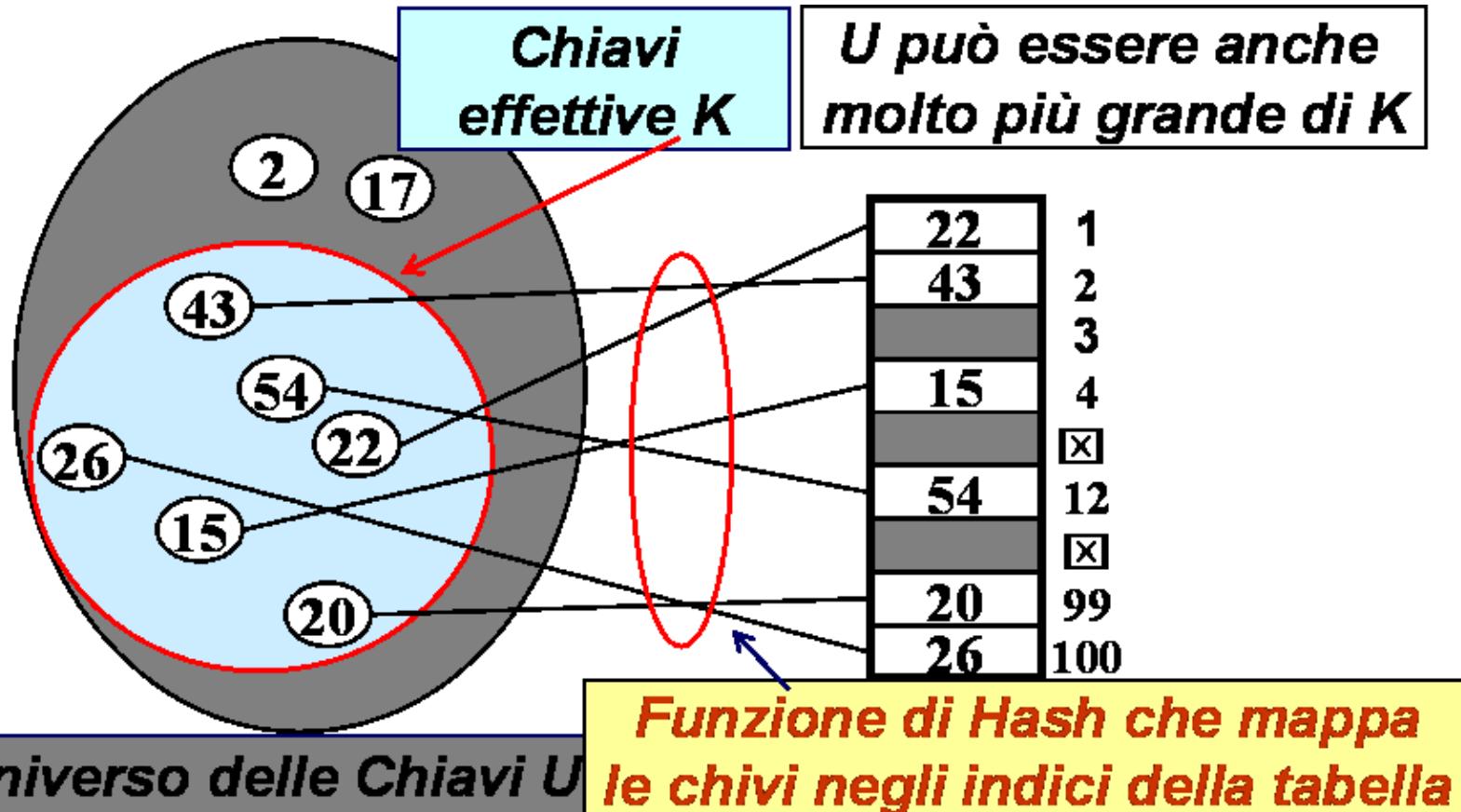


Illustrazione di una **tabella hash** e della **funzione di hashing**.

- Uno ***schema di hashing*** consiste di una ***tabella ad accesso diretto*** (la ***tabella hash***) e di una ***funzione di hashing*** con dominio l'universo delle chiavi e codominio l'insieme degli indici della tabella.
- Una ***funzione hash*** prende in input una chiave e la **"mappa"** su qualche ***indice*** all'interno della tabella.
- Uno ***schema di hashing*** ammette che differenti chiavi possibili vengano **"mappate"** nella stessa locazione (la ***funzione di hash NON è iniettiva***). Quando ciò avviene, si parla di ***collisione*** tra chiavi.
- È quindi necessario definire dei meccanismi opportuni per la gestione delle ***collisioni***.

Data una funzione di hash **hash(key)** che ritorna un intero, l'approccio semplicistico potrebbe essere il seguente

- **inserisci(key)** : $A[hash(key)] = \text{oggetto da inserire}$
- **ricerca(key)** : l'oggetto è presente in $A[hash(key)]$?
- **cancella(key)** : $A[hash(key)] = \text{NULL}$

Nella definizione di una **tabella hash** adeguata alle necessità applicative si deve scegliere:

- una opportuna **funzione di hash** che abbia buone proprietà di distribuzione uniforme delle chiavi sugli indici
- la **dimensione della tabella** che spesso dipende dal tipo di funzione hash scelto
- la *politica di gestione e soluzione* delle **collisioni**

TSIZE = 10

0	10, 100
1	
2	2
3	
4	
5	5
6	
7	
8	18
9	9

hash(k) = k mod TSIZE

Semplice funzione di Hash su interi:

TSIZE: dimensione della tabella

mod: operazione di modulo

Esempio di tabella hash e funzione hash.

TSIZE = 10

0	10, 100
1	
2	2, 12, 22
3	
4	
5	5, 15, 25
6	
7	
8	18
9	9

$$\text{hash}(k) = k \bmod \text{Tsize}$$

Le collisioni sono molto frequenti.

La dimensione della tabella è inadeguata

Si noti che **10 = 2·5**, e i multipli di **2** (pari) possono solo finire nelle **celle di indice pari**, mentre multipli di **5** solo nelle celle di **indice 0 e 5**.

hash(k) = k mod TSIZE

TSIZE = 10

0	10, 100
1	
2	2, 12, 22
3	
4	
5	5, 15, 25
6	
7	
8	18
9	9

TSIZE = 11

0	22
1	12, 100
2	2
3	25
4	15
5	5
6	
7	7, 18
8	
9	9
10	10

Si noti che **11** è primo e ora i multipli di **2** e di **5** coprono tutte le celle della tabella (minor possibilità di collisioni).

La **dimensione della tabella** può influire sulla **frequenza delle collisioni**

- **TSIZE** Numero Composto
 - **10: 2×5**
 - **300: $2 \times 2 \times 3 \times 5 \times 5$**
 - **Scarsa uniformità** della distribuzione delle chiavi che determina *maggiori possibilità di collisioni*
- **TSIZE** Numero Primo
 - **11**
 - **10007**
 - **Maggiore uniformità** della distribuzione delle chiavi porta a *minori possibilità di collisioni*

Una proprietà importante di un meccanismo di hashing è quella di

- **Hashing Uniforme Semplice**: ogni chiave ha la stessa probabilità di essere mappata in una delle n celle della tabella, indipendentemente dalla cella in cui è mappata ogni altra chiave.

Proprietà desiderabili di una “**buona” funzione di hash** sono:

- **efficienza** e facilità di calcolo
- **distribuzione uniforme** delle chiavi sul dominio degli indici
- **minimizzazione** delle **collisioni**

1. **Metodo della Divisione**

Convertire la chiave in un intero e calcolare il modulo (**mod**) rispetto alla dimensione della tabella

2. **Metodo de Moltiplicazione**

Convertire la chiave in un intero utilizzando operazioni di moltiplicazione

3. **Troncamento**

Ignorare parte della chiave e usare la porzione che rimane come indice

4. **Folding**

Partizionare la chiave in parti differenti e combinare queste parti in modo da ottenere l'indice

Metodo della divisione

- e.g. Semplicemente calcolare il **modulo** rispetto alla dimensione della tabella:

$$\mathbf{hash(62538194) = 62538194 \ mod \ 1000 = 194}$$

- Il metodo è sensibile al valore scelto per la dimensione della tabella.
- es. l'uso di una potenza di due per la dimensione può causare scarsa uniformità della funzione. Se $n = 2^p$, allora vengono considerati solo i **p** bit meno significativi della chiave.
- Nel caso dell'aritmetica modulare, la migliore scelta della **dimensione della tabella hash** è un **numero primo**.
- Nel caso sopra, tabelle di dimensione 997 o 1009 (entrambi numeri primi) darebbero migliori prestazioni dal punto di vista della distribuzione delle chiavi sugli indici.

Metodo della moltiplicazione

- Prima si moltiplica la chiave k per una costante a nell'intervallo $0 < a < 1$.
- Poi si estraе la parte frazionaria del prodotto $k \cdot a$, cioè il valore
$$k \cdot a - \lfloor k \cdot a \rfloor$$
- Infine si moltiplica il valore ottenuto per la dimensione n della tabella.

$$\text{hash}(k) = \lfloor n \cdot (k \cdot a - \lfloor k \cdot a \rfloor) \rfloor$$

- Chiaramente, $0 \leq (k \cdot a - \lfloor k \cdot a \rfloor) < 1$ e, quindi, $0 \leq \text{hash}(k) < n$.
- Questo metodo non è particolarmente sensibile al valore n della dimensione della tabella.

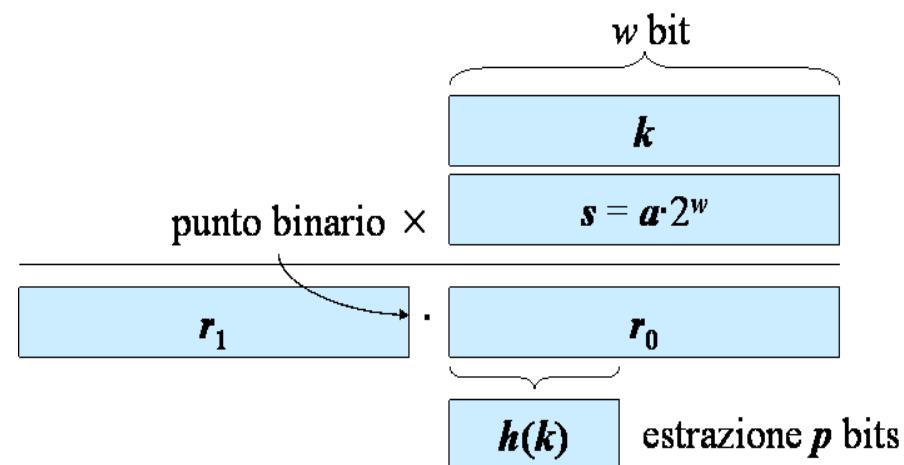
Il metodo della moltiplicazione può essere implementato facilmente e in **modo efficiente**, scegliendo il valore n come una potenza di 2 .

- Sia $n = 2^p$ per qualche intero p .
- Sia w la dimensione della parola della macchina (numero di bit in una parola).
- Scegliamo $a = s/2^w$, con $0 < s < 2^w$ intero.
- Moltiplicando k per $s = a \cdot 2^w$. Il risultato sarà un valore di $2 \cdot w$ bit, della forma

$$k \cdot s = r_1 \cdot 2^w + r_0$$

dove r_1 è la parte più significativa del prodotto, e r_0 quella meno significativa.

- Il valore $\text{hash}(k)$ è rappresentato dai p bit più significativi di r_0 .



Implementazione del metodo della moltiplicazione.

Si noti che

$$k \cdot a - \lfloor k \cdot a \rfloor = \frac{k \cdot s}{2^w} = r_0$$

E che, posto $n = 2^p$, si ottiene

$$\lfloor n \cdot (k \cdot a - \lfloor k \cdot a \rfloor) \rfloor = \lfloor n \cdot r_0 \rfloor = 2^p \cdot r_0$$

Cioè i p bit più significativi di r_0 .

Dati interi di 8 cifre e una tabella di dimensione 1000

Troncamento

- es. — usare congiuntamente solo la 4^a, 7^a e 8^a cifra per formare l'indice: $\text{hash}(62538194) = 394$

Folding

- es. — suddividere ogni chiave in gruppi di 3, 3, e 2 cifre, sommare le parti e troncare se necessario:

$$\begin{aligned}\text{hash}(62538194) &= (625+381+94) \text{ mod } 1000 = \\ &= 1100 \text{ mod } 1000 = 100\end{aligned}$$

Definire una funzione che trasforma una stringa in un intero.

- Ad esempio, sommando il valore ASCII di tutti i caratteri:
 - ad esempio $\text{hash1}(K) = \sum_i K[i]$
 - oppure, meglio, $\text{hash2}(K) = \sum_i d^i \cdot K[n-i]$ (per d intero > 1)
- Problema **hash1()**: quando le chiavi sono corte e la tabella è grande
 1. 8 caratteri, $\text{TSIZE} = 10007$, ma $8 \cdot 256 = 2048$. La tabella può, quindi, risultare sproporzionata, determinando **spreco di spazio**.
 2. tutte le permutazioni della stessa stringa **collidono allo stesso valore hash**.

Una possibile soluzione è quella di usare solo alcuni caratteri e **moltiplicare tra loro i valori dei caratteri** (*metodo del troncamento*)

- :
 - C a p o V e r d e
 - Numero di possibili valori di indice: $27*27*27 = 17576 > 10007$
 - Può essere necessario, quindi, integrarlo con il *metodo della divisione*.
- Problema: le lingue non sono casuali
 - molte meno combinazioni effettivamente possibili di quelle permesse
 - rischio di spazio sprecato

Le seguenti funzioni di hash pesano diversamente ciascun carattere della stringa e impiegano il metodo della divisione (ipotizzando 27 diversi caratteri alfabetici) :

$$\begin{aligned}1. \ hash_1(K) &= (\dots + 27^2 K[2] + 27 K[1] + K[0] \dots) \text{ mod TSIZE} \\&= ((\dots + K[2]) * 27 + K[1]) * 27 + K[0] \dots) \text{ mod TSIZE}\end{aligned}$$

$$2. \ \textcolor{blue}{hash_2(K) = ((\dots + K[2]) * 32 + K[1]) * 32 + K[0] \dots) \text{ mod TSIZE}}$$

L'algoritmo sotto riportato calcola la seconda funzione di hash nell'esempio:

```
Hash2(K[])
  i = 1
  WHILE (K[i] ≠ '\0') DO
    hash = (shift(hash,5)) + K[i] /* hash = hash * 25 + key[i] */
    i = i + 1
  return (hash mod TSIZE)
```

Si noti che l'espressione **shift(hash,5)** corrisponde alla moltiplicazione del valore contenuto in **hash** per **32** (cioè per **2⁵**).

hash₃(0) = 5381

hash₃(i) = hash₃(i - 1) * 33 + K[i]

```
Hash3(K[])
    hash = 5381
    i = 1
    WHILE (K[i] ≠ '\0') DO
        hash = ((shift(hash,5)) + hash) + K[i]
        i = i + 1
    return (hash mod TSIZE)
```

Questa funzione di hash ha mostrato prestazioni di uniformità particolarmente buone in pratica.

Un'altra possibilità è **usare il folding**: elaborare la stringa 4 byte alla volta, convertendo ogni gruppo di 4 byte in un intero, usando uno dei metodi sopra descritti. I valori interi di ogni gruppo vengono poi sommati tra di loro. Infine, si converte il risultato in un intero tra 0 e TSIZE tramite operazione di modulo.

- Semplice somma dei valori numerici dei caratteri della stringa
 - Molto semplice da implementare.
 - Può impiegare molto tempo se le chiavi sono lunghe.
 - I primi caratteri possono non venir considerati.
 - Posso essere spostati (`shift`) fuori dal range.
- Una possibile soluzione può essere quella di adottare una variante del folding:
 - usare solo alcuni caratteri
 - e.g. Via Cintia 345, Napoli, I-81100
- Un'altra soluzione può essere quella utilizzare il metodo della divisione insieme alla somma (pesata) dei diversi caratteri.

Tabelle Hash: gestione delle collisioni

Lezione n.#
Parole chiave:

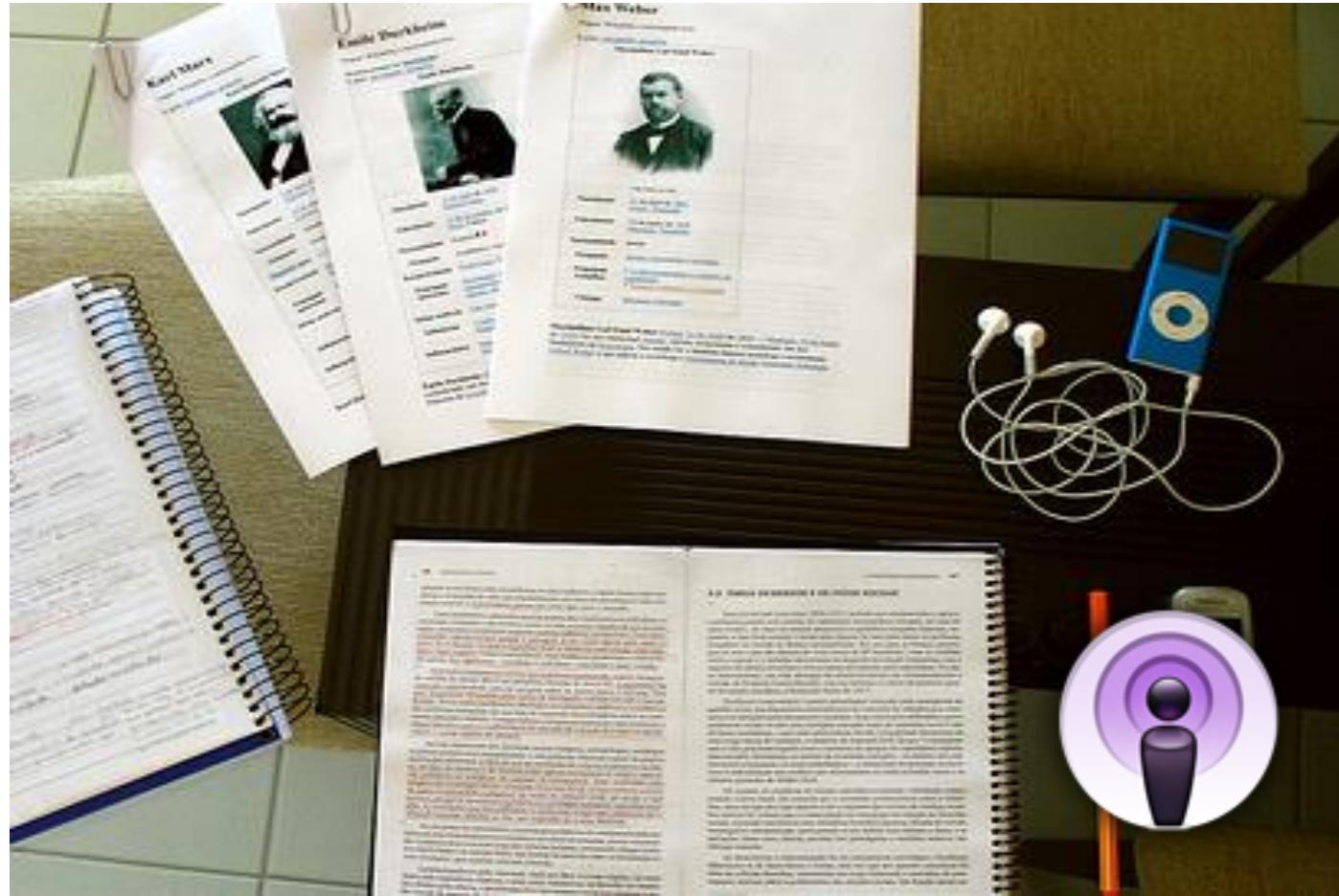
Corso di Laurea:
Informatica

Insegnamento:

Algoritmi e
Strutture Dati I

Email Docente:
bene@na.infn.it

A.A. 2009-2010



Nessuno schema di hashing può garantire, in generale, l'assenza di collisioni. Si pone quindi il problema di **come risolvere le collisioni?**

Esistono due tecniche principali:

- ***Indirizzamento aperto***

- la tabella è un array che contiene al massimo un oggetto per indice
- spazio di memoria contiguo

- ***Indirizzamento chiuso (concatenamento)***

- la tabella è un array di catene (ad esempio liste concatenate). Tutti gli elementi su una catena hanno lo stesso indice
- spazio di memoria dinamico

Tutti i dati sono contenuti all'interno della tabella.

Non è necessaria memoria aggiuntiva oltre alla tabella.

In caso di collisioni, vengono **tentate posizioni alternative** finché non se ne trova una vuota.

Per avere buone prestazione è tipicamente necessaria una tabella assai più grande del numero di elementi da memorizzare.

Il **tasso di riempimento** (**m/n**, numero di chiavi in tabella diviso dimensione della tabella) dovrebbe essere mantenuto al di sotto del **50%**.

La figura a destra riporta il risultato dell'inserimento, nell'ordine, delle chiavi **0, 1, 4, 9, 16, 25, 36, 49**.

La chiave **36** collide in posizione **6** con la chiave **16. 36** viene così inserita nella prima cella libera in posizione **7**. Analogamente accade per la chiave **49** (che collide con **9**). La prima cella libera è, in questo caso, in posizione **2**.

$$h(k) = k \bmod 10$$

0	0
1	1
2	49
3	
4	4
5	25
6	16
7	36
8	
9	9

Hash-Search(T,k)

```
i = 0
repeat
    j = h(k,i)
    if (T[j] = k) then
        return j
    else
        i = i + 1
until (T[j] = NIL or i = n)
return NIL
```

Hash-Insert(T,k)

```
i = 0
repeat
    j = h(k,i)
    if (T[j] = NIL) then
        T[j] = k
        return j
    else
        i = i + 1
until i = n
error: "Table Overflow"
```

Data una funzione di hash **$h(k)$** (detta **hash primario**) definiamo una nuova **funzione di hash $h(k,i)$** , con parametri la chiave **k** e l'indice **i** del sondaggio tale che **$h(k,0)=h(k)$** .

- Tentare altre celle
 - Le celle $h(x,0)$, $h(x,1)$, $h(x,2)$, ... vengono sondate in successione finché non se ne trova una libera.
 - La **funzione di Hash $h(x,i)$** ideale soddisfa la condizione di **Hashing Uniforme**:
«ogni chiave k ha la stessa probabilità di determinare, come sequenza di sondaggi, una qualsiasi delle $n!$ permutazioni della sequenza $0,1,\dots,n-1$ »
 - In pratica si riescono a realizzare funzioni di Hash che **approssimano** più o meno bene l'**Hashing Uniforme**.

- Tentare altre celle
 - Le celle $h(x,0)$, $h(x,1)$, $h(x,2)$, ... vengono sondate in successione.
$$h(x,i) = (h(x) + f(i)) \text{ mod TSIZE}$$
 - $f(0) = 0$
 - f definisce la **strategia di risoluzione delle collisioni.**
- **Sondaggio (probing) lineare**
 - $f(i) = i$
- **Sondaggio (probing) quadratico**
 - $f(i) = i^2$

A titolo di esempio, definiamo il seguente schema di hashing con sondaggio lineare:

$$\bullet \quad h(x) = x \bmod \text{TSIZE} \quad \text{e} \quad f(i) = i$$

$$\begin{aligned} h(x,i) &= (h(x) + f(i)) \bmod \text{TSIZE} \\ &= (x + i) \bmod \text{TSIZE} \end{aligned}$$

Nella figura a fianco è riportata la simulazione dello schema a fronte delle seguenti operazioni:

- **Inserimento** di 19, 28, 39, 48, 29, 18
- **Cancellazione** di 28
- **Ricerca** di 18

Si noti che l'eventuale cancellazione fisica di **28** dalla tabella impedirebbe una ricerca con successo di **18**.

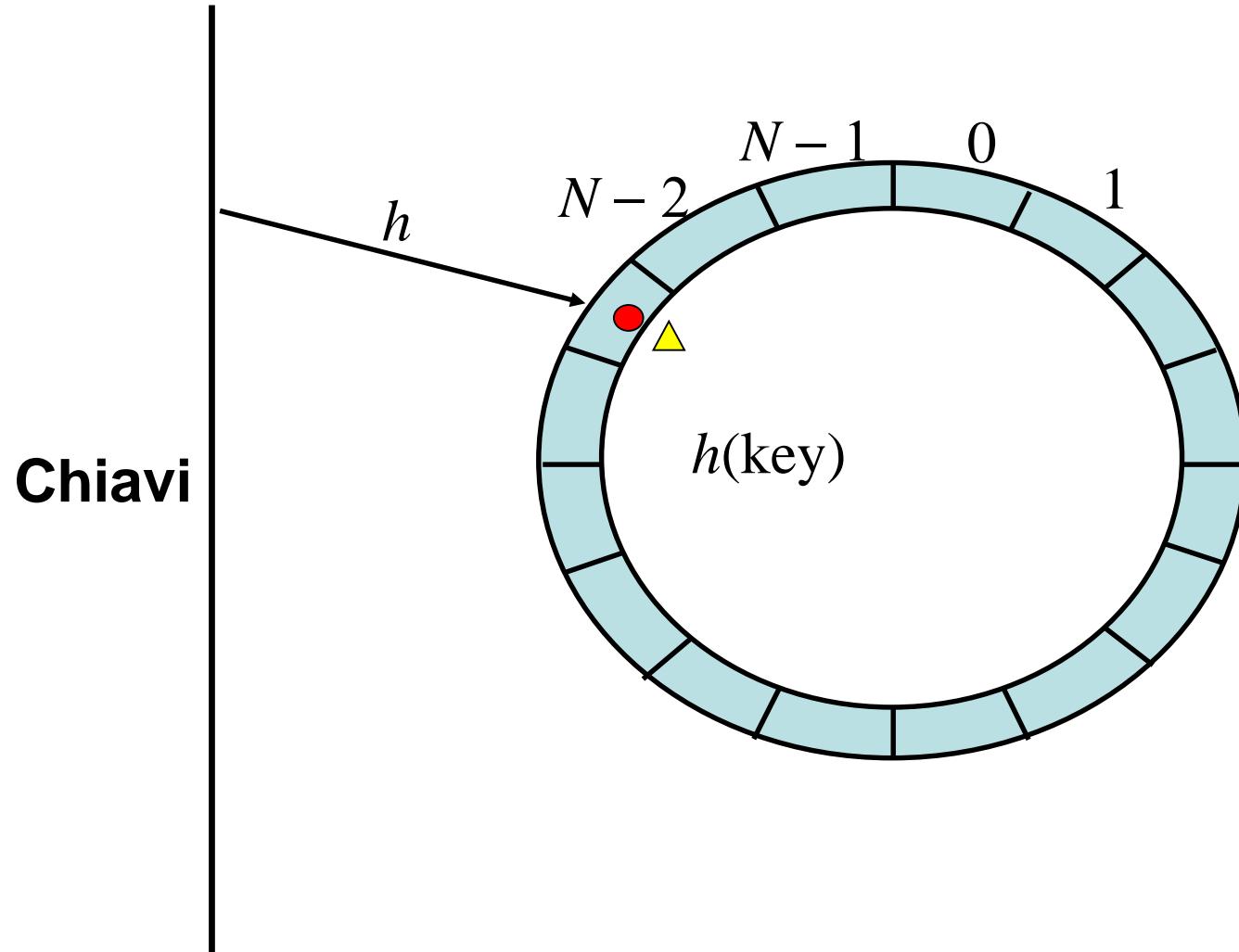
Infatti, la cancellazione di **28** interromperebbe la catena dei sondaggi necessari a trovare **18**.

0			39	39	39	39	39	39	39
1									
2									
3									
4									
5									
6									
7									
8		28	28	28	28	28	28	28	28
9	19	19	19	19	19	19	19	19	19

Esempio di operazioni su tabella hash con sondaggio lineare

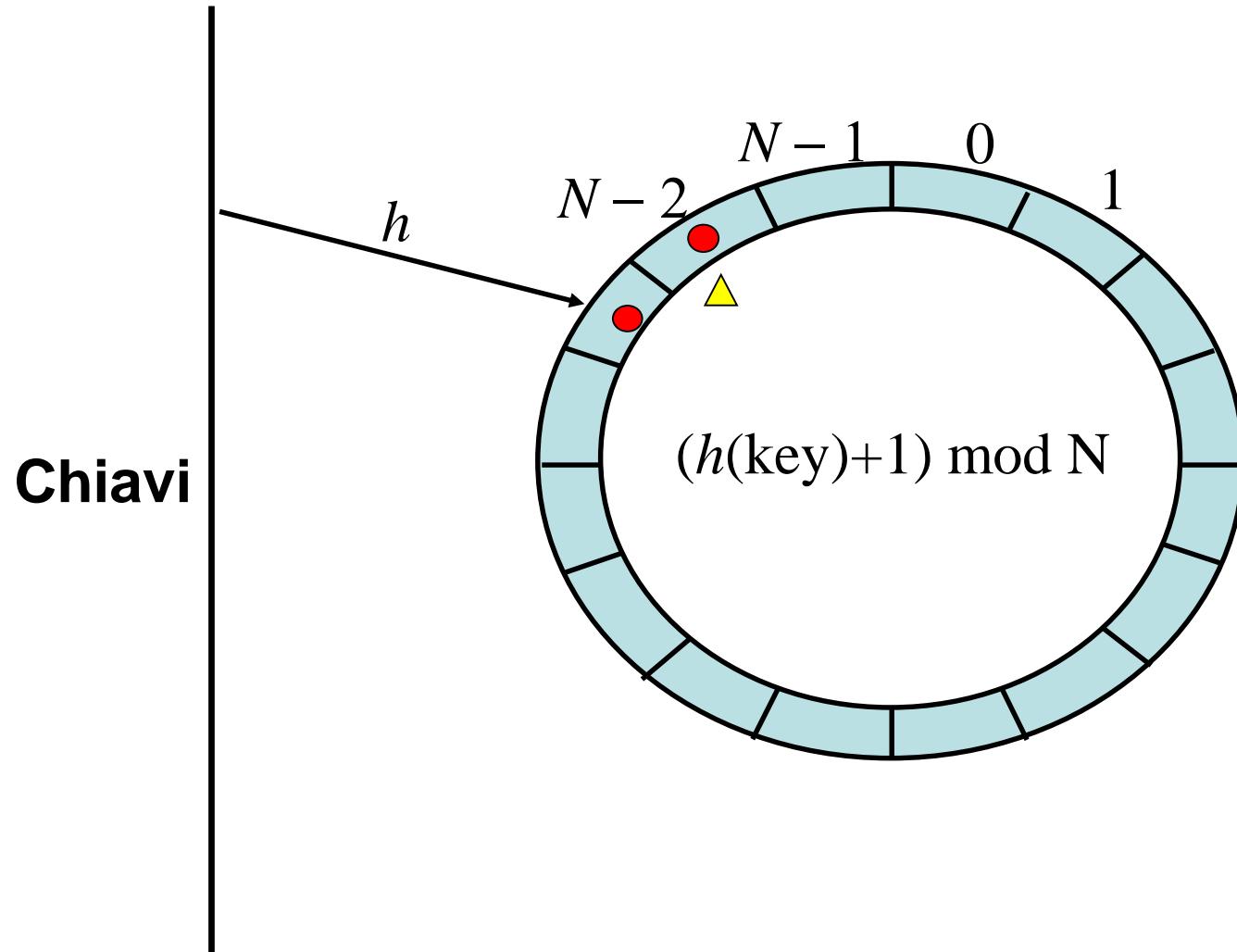
Sondaggio (probing) lineare

- È il *metodo* di risoluzione delle collisioni *più semplice*
- Dato un *indice hash*, esegue una ricerca lineare della chiave desiderata una locazione vuota (per l'inserimento)
- La tabella è vista come un **array circolare**, al raggiungimento dell'ultimo indice, la ricerca riparte dal primo indice (realizzato tramite operazione di modulo)
- Se la tabella non è piena, è sempre possibile trovare una posizione vuota.



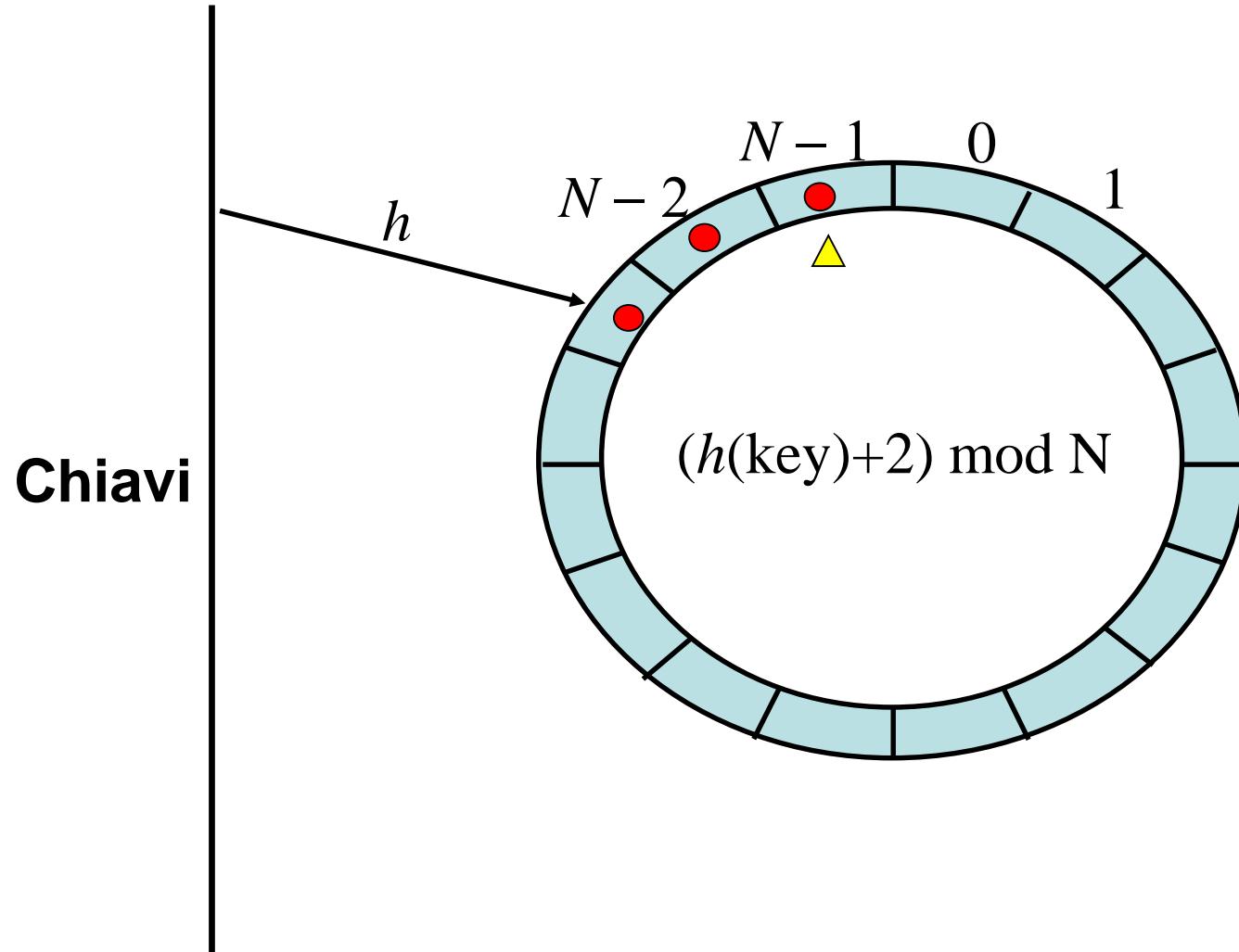
Chiavi

Simulazione del sondaggio lineare.



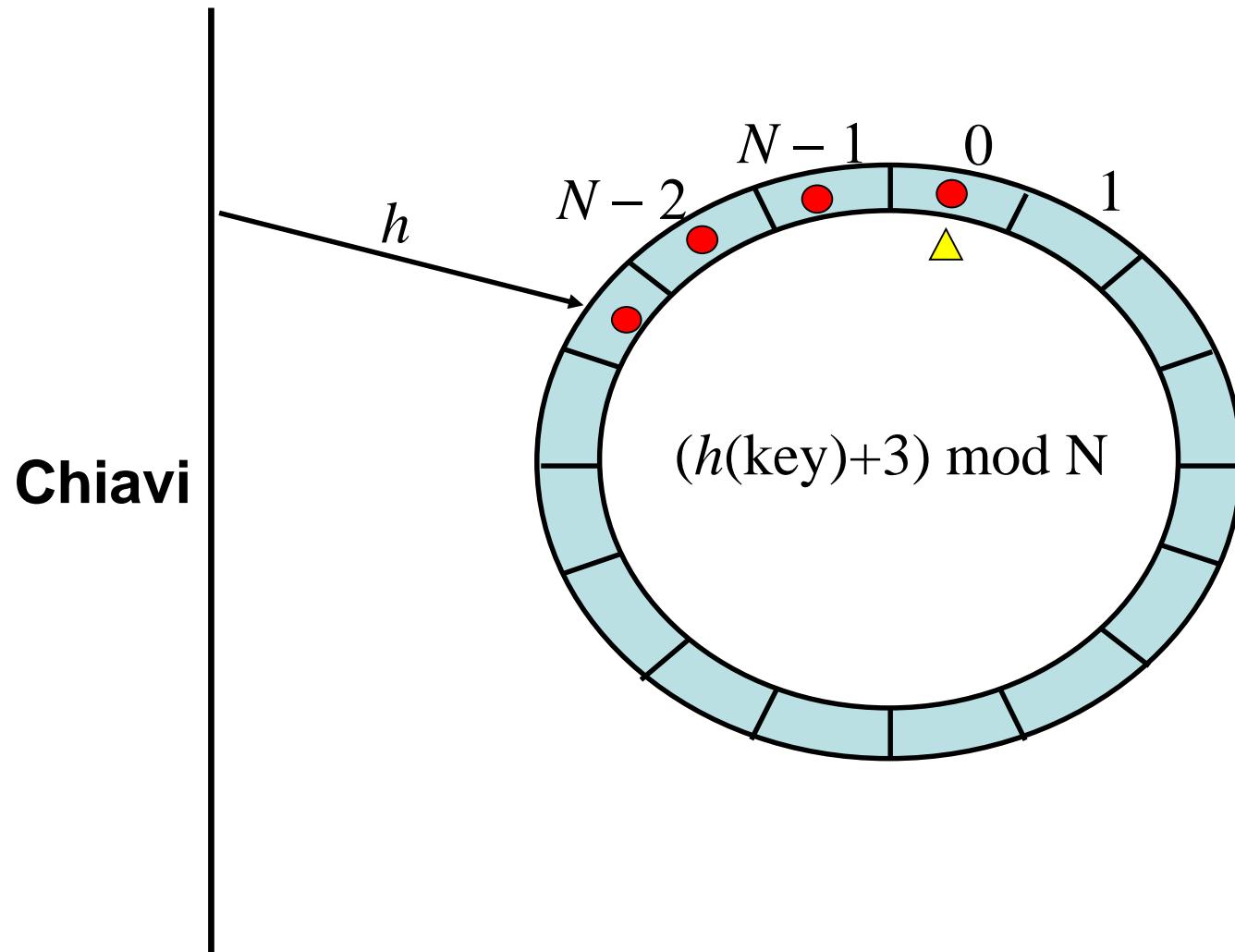
Chiavi

Simulazione del sondaggio lineare.

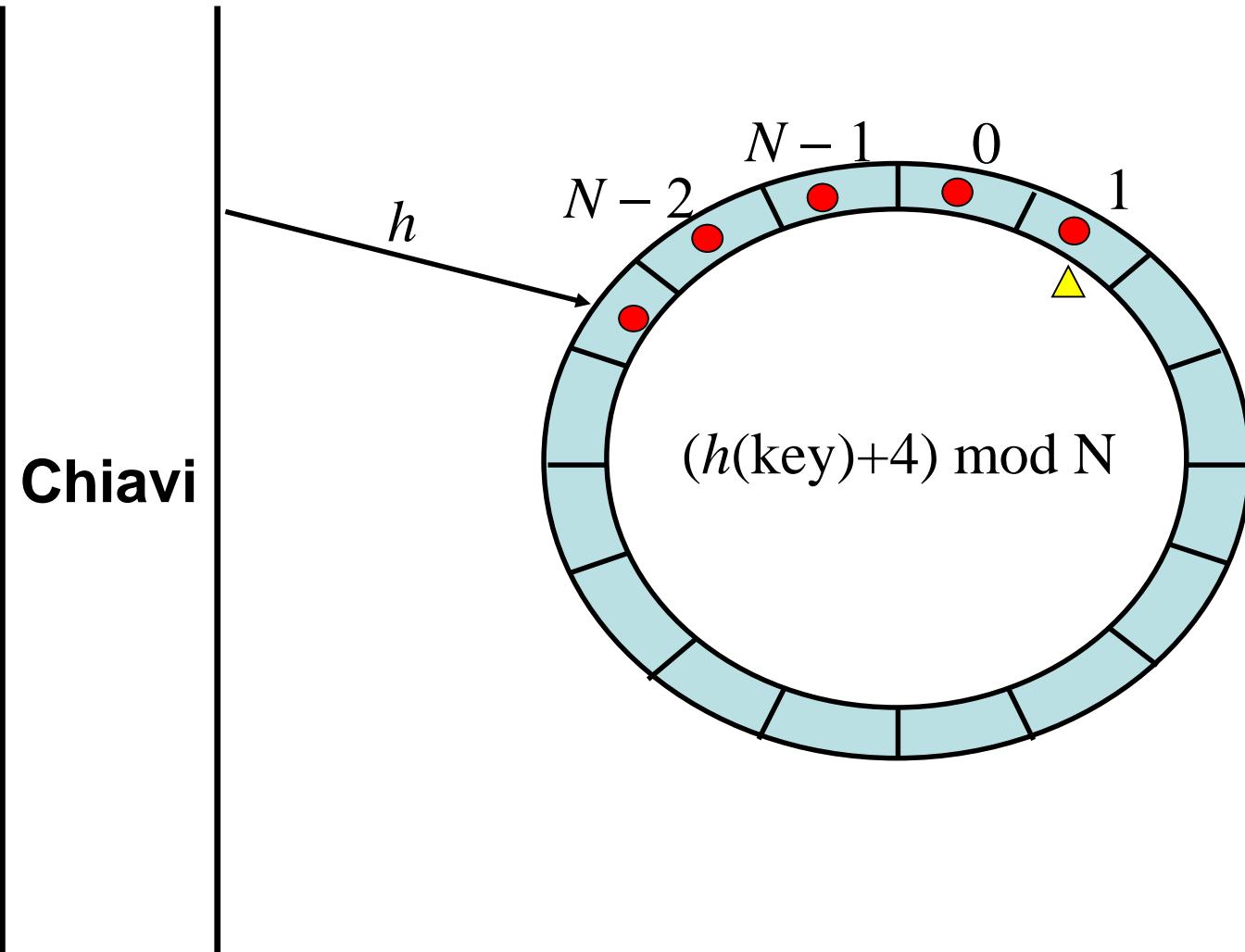


Chiavi

Simulazione del sondaggio lineare.



Simulazione del sondaggio lineare.

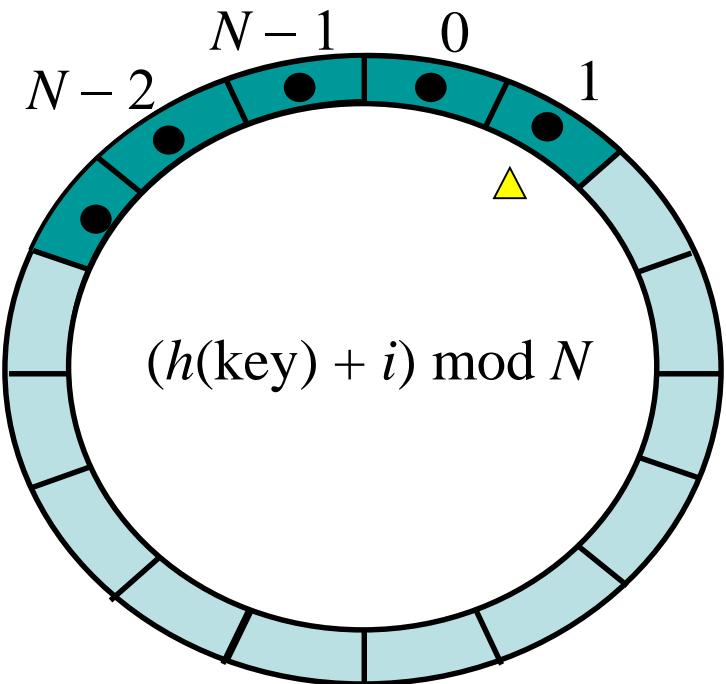


Simulazione del sondaggio lineare.

Clustering primario

- Lo **svantaggio** maggiore del **sondaggio lineare** è che quando la tabella diventa piena quasi per metà, si verifica una tendenza al “**Clustering**” (raggruppamento) **primario** (sul valore primario dell'hash)
- Il “**clustering primario**” si verifica quando gli elementi iniziano a delinearsi come lunghe file di posizioni adiacenti occupate. Tutte le chiavi il cui hash è nel cluster, aumentano la dimensione del cluster.
 - anche con chiavi che hanno valori di **hash** diversi dalla chiave che ha dato origine al cluster.
- La ricerca lineare di una locazione libera (vuota) diviene sempre più lunga

- Il “**clustering primario**” si verifica quando gli *elementi* iniziano a delinearsi come **lunghe file di posizioni adiacenti occupate**.
- Tutte le chiavi il cui **hash ricade nel cluster** aumentano la dimensione del cluster, anche chiavi che hanno valori di **hash** primari diversi quello dalla chiave che ha dato origine al cluster.
- La **ricerca** di una **locazione libera** (vuota) diviene **sempre più lunga**.

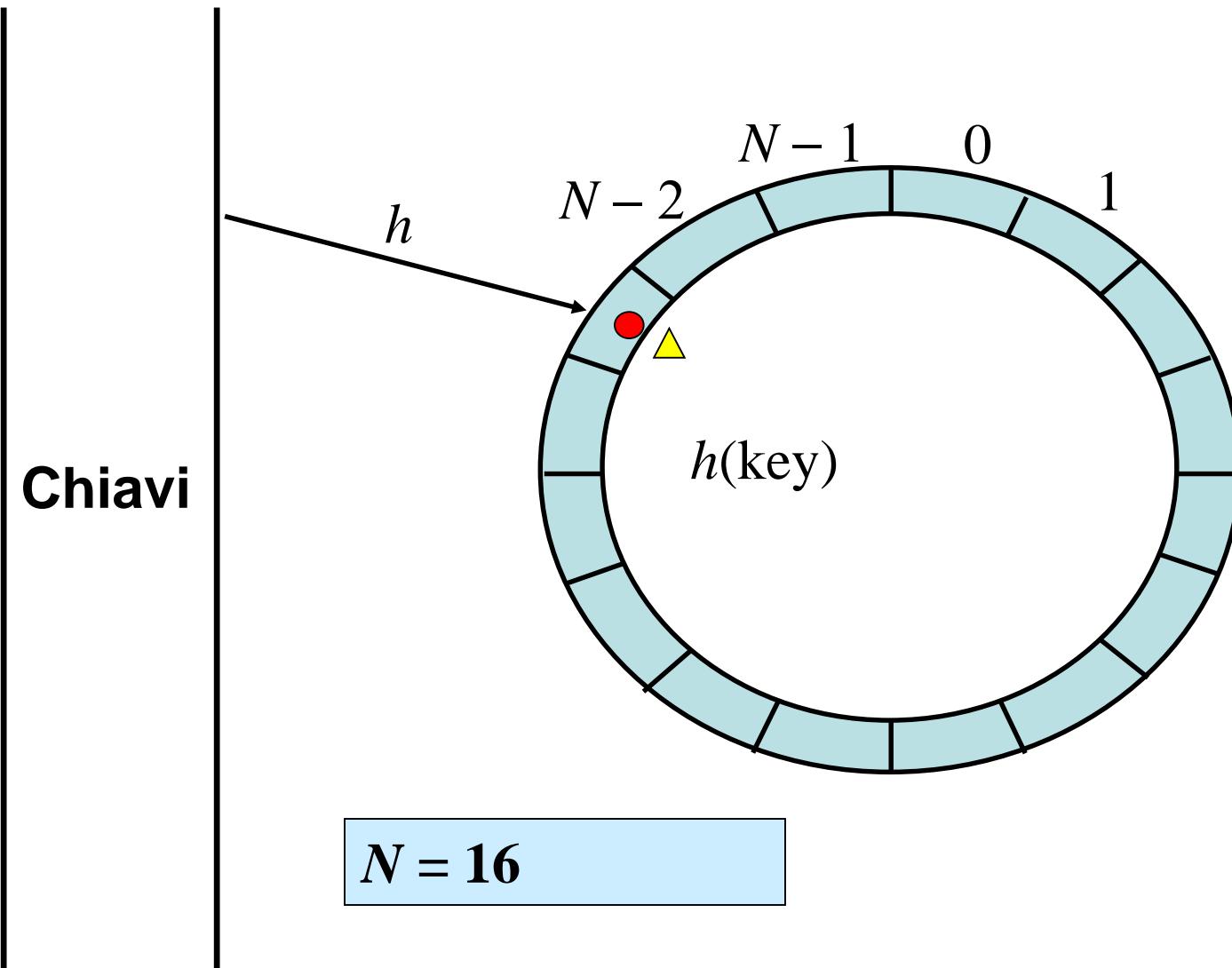


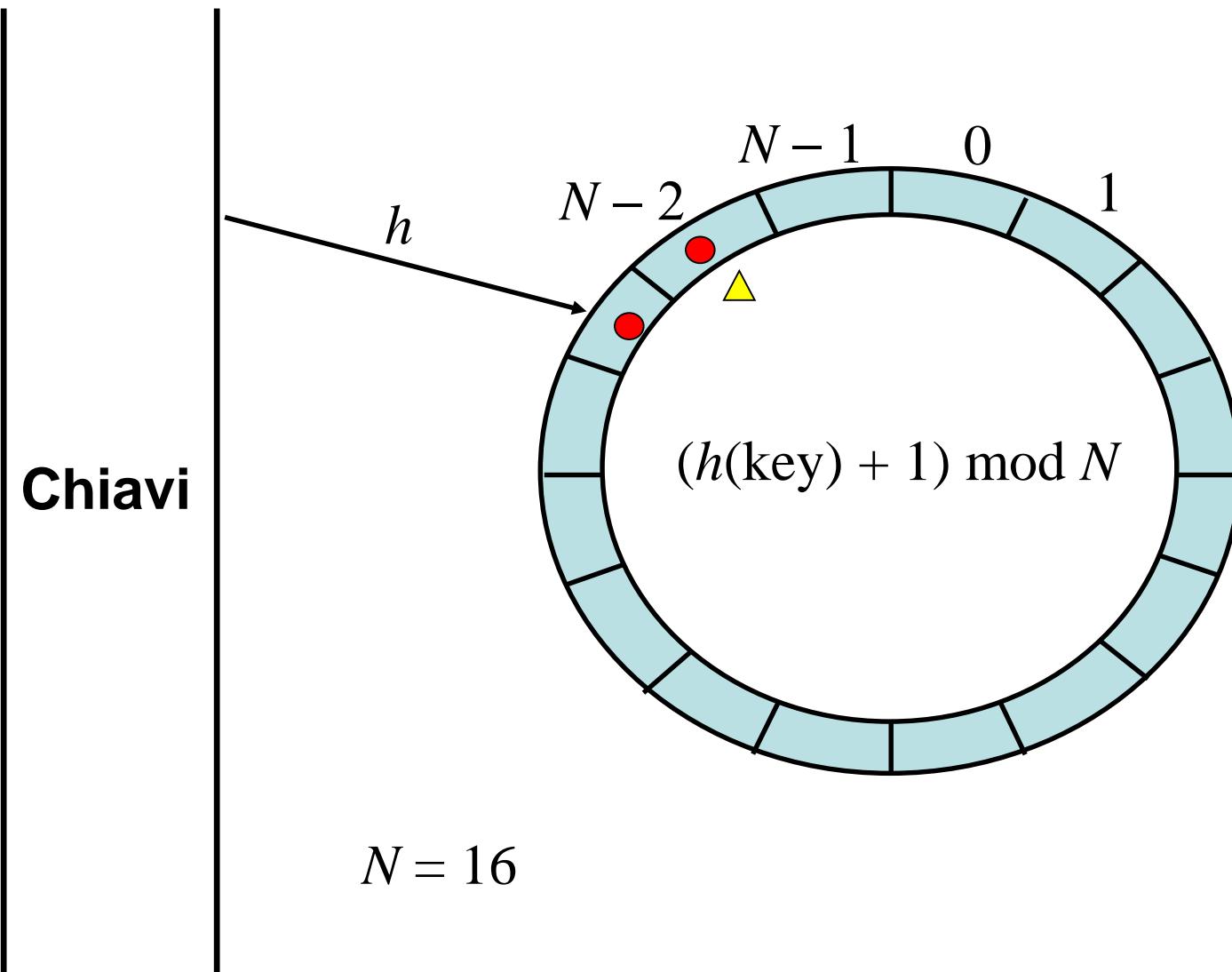
Sondaggio (probing) quadratico

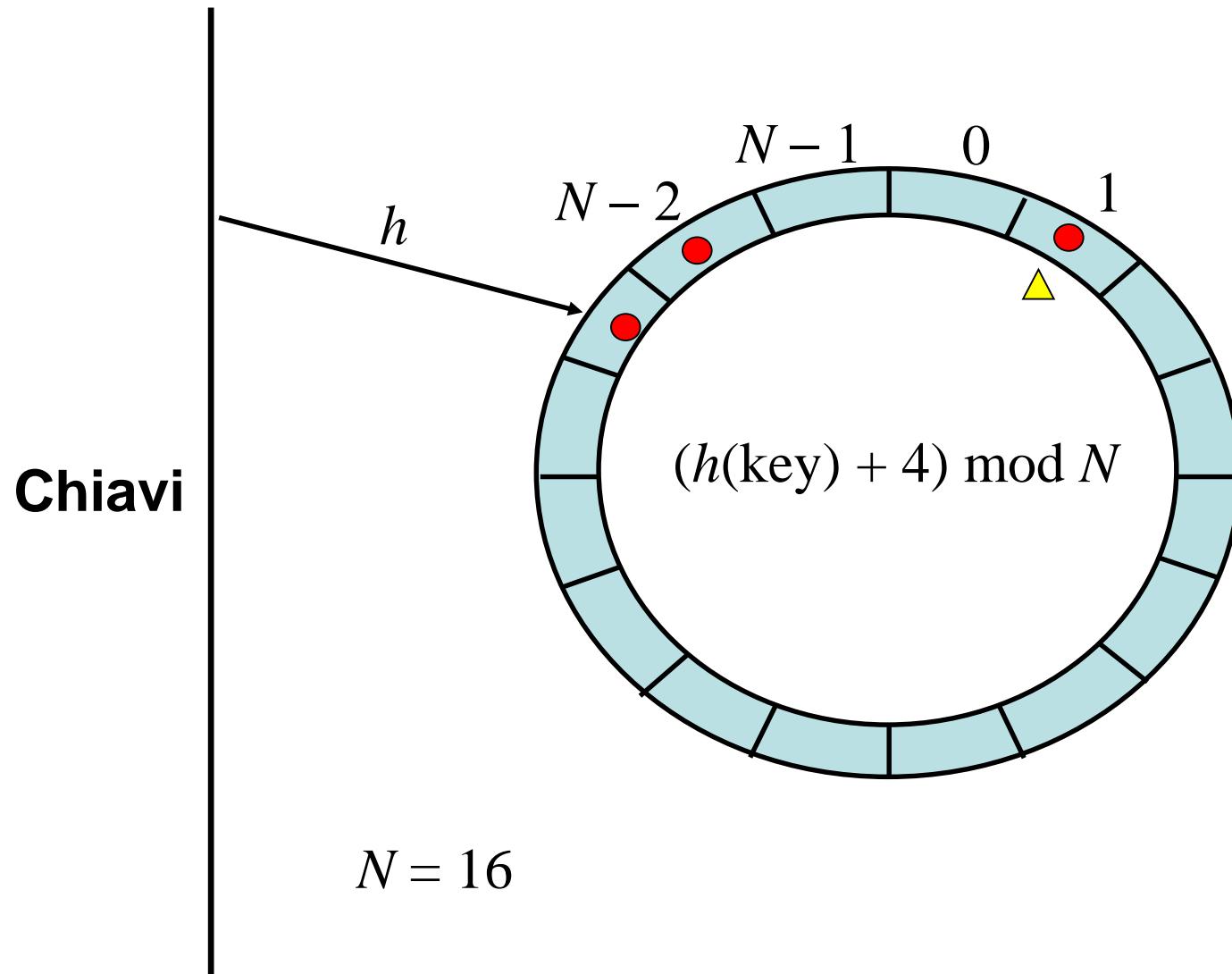
- Se $\text{hash}(\text{key}) = h$, tentare le locazioni $h+1, h+4, h+9, h+16$, ecc. Cioè tentare le locazioni

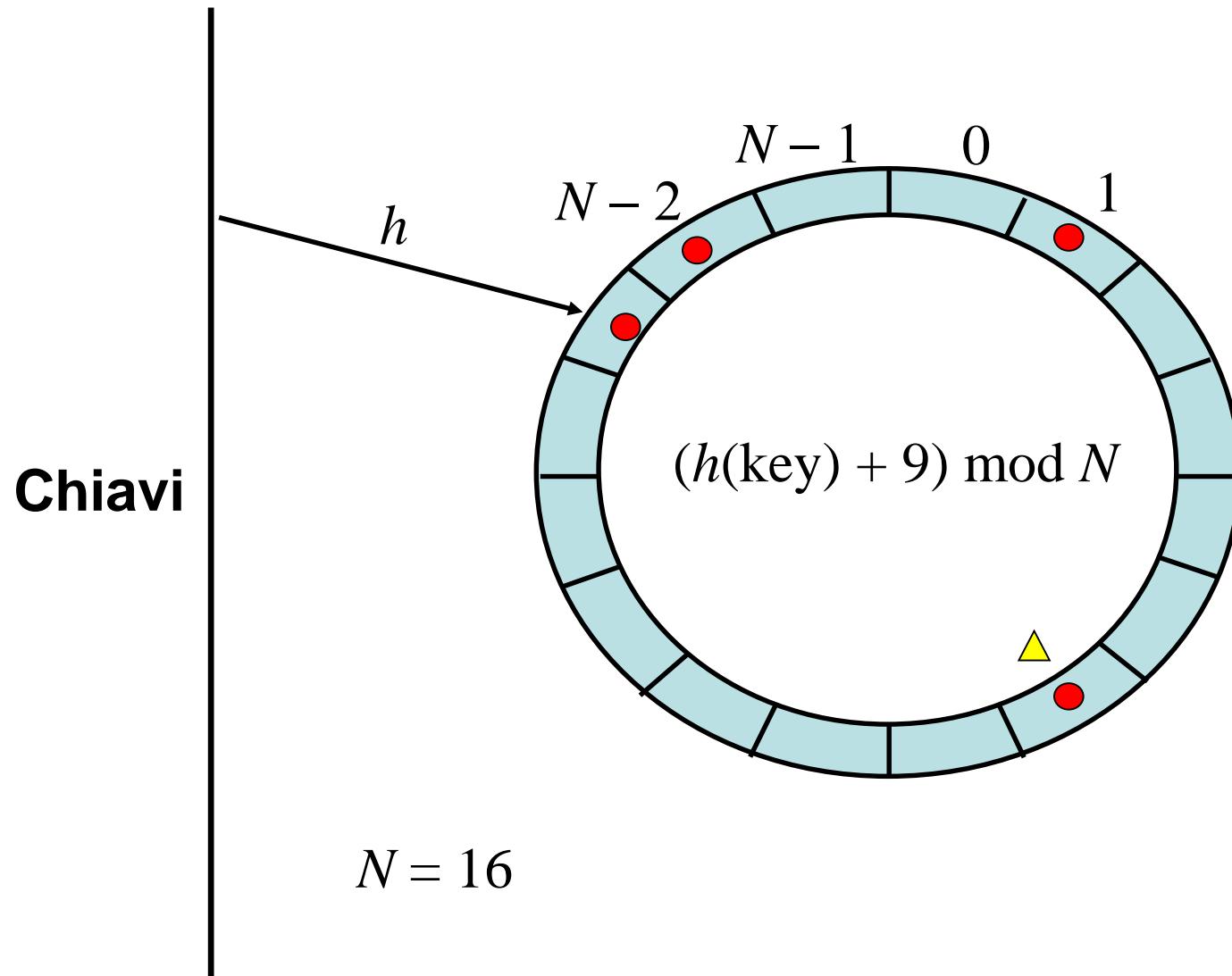
$$\mathbf{h(key,i) = (hash(key) + i^2) \ mod \ N} \quad \text{per } i = 1,2,3,4,\dots$$

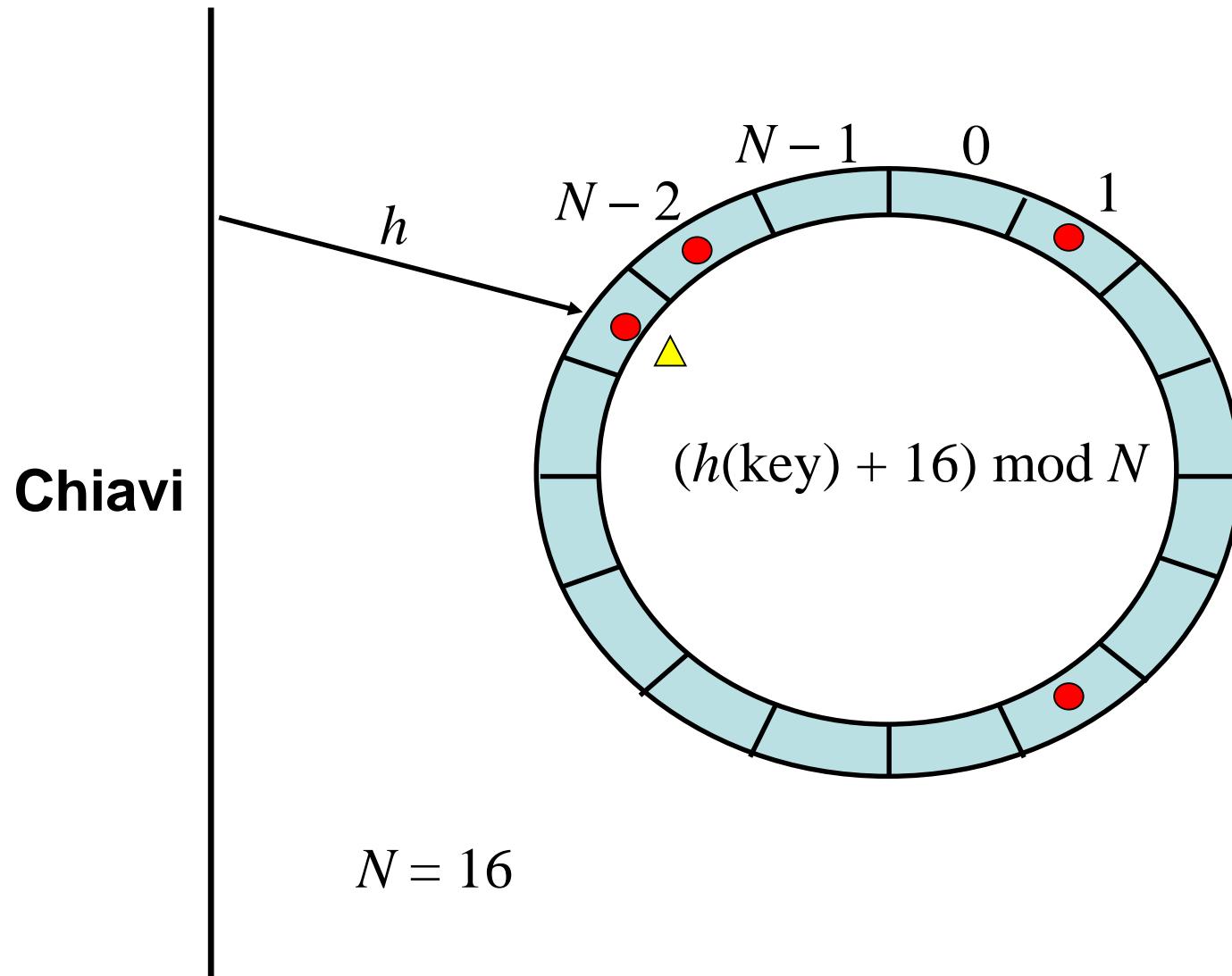
- Elimina il problema del **clustering primario** (i sondaggi non sono contigui)
- Ma anche se la tabella non è piena, **non** è garantito che si riesca a trovare una cella vuota.
- Gli elementi che hanno lo **stesso valore di hash primario**, indirizzeranno lo stesso insieme di celle alternative:
 - **clustering secondario**
 - in pratica non costituisce un grosso problema

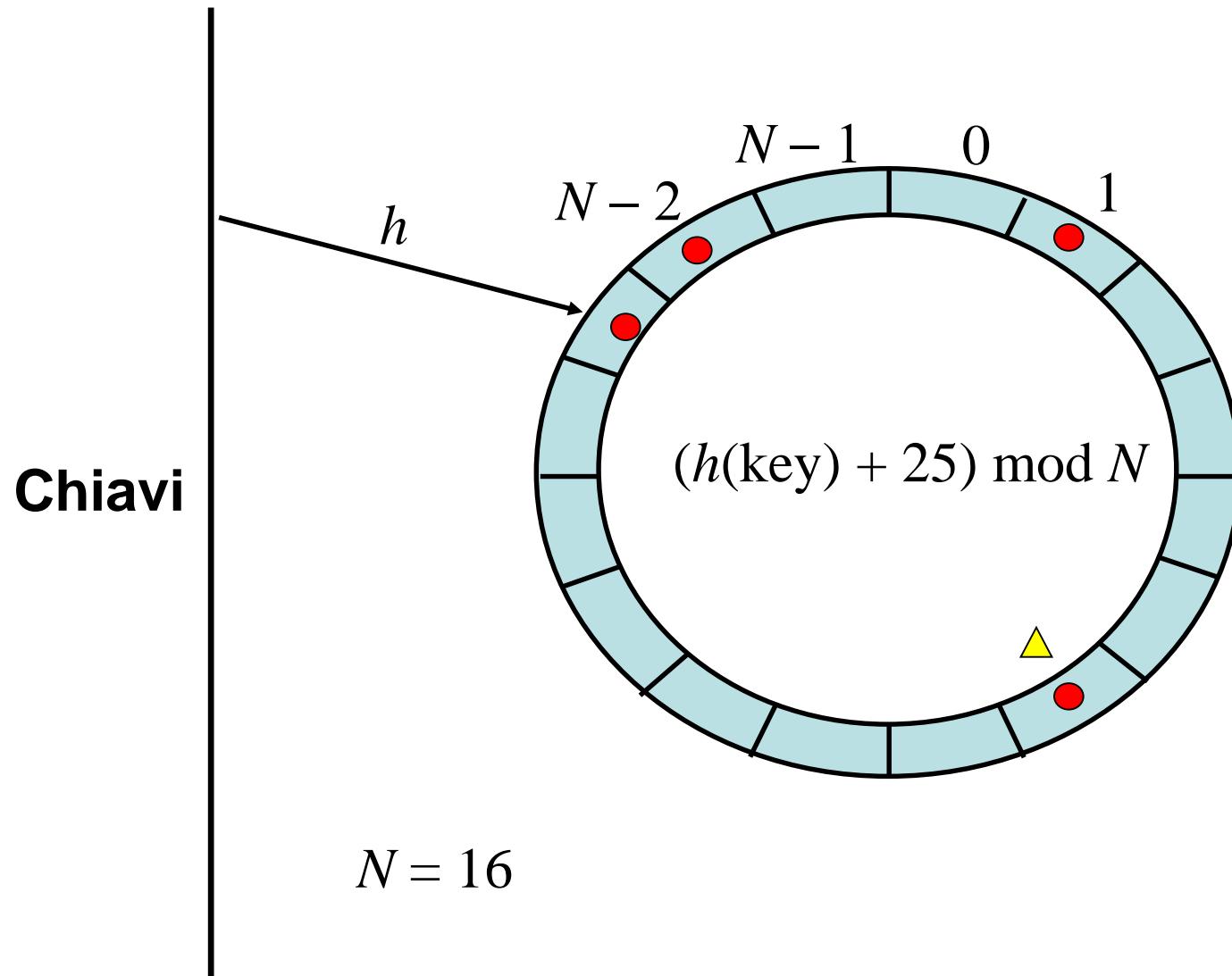


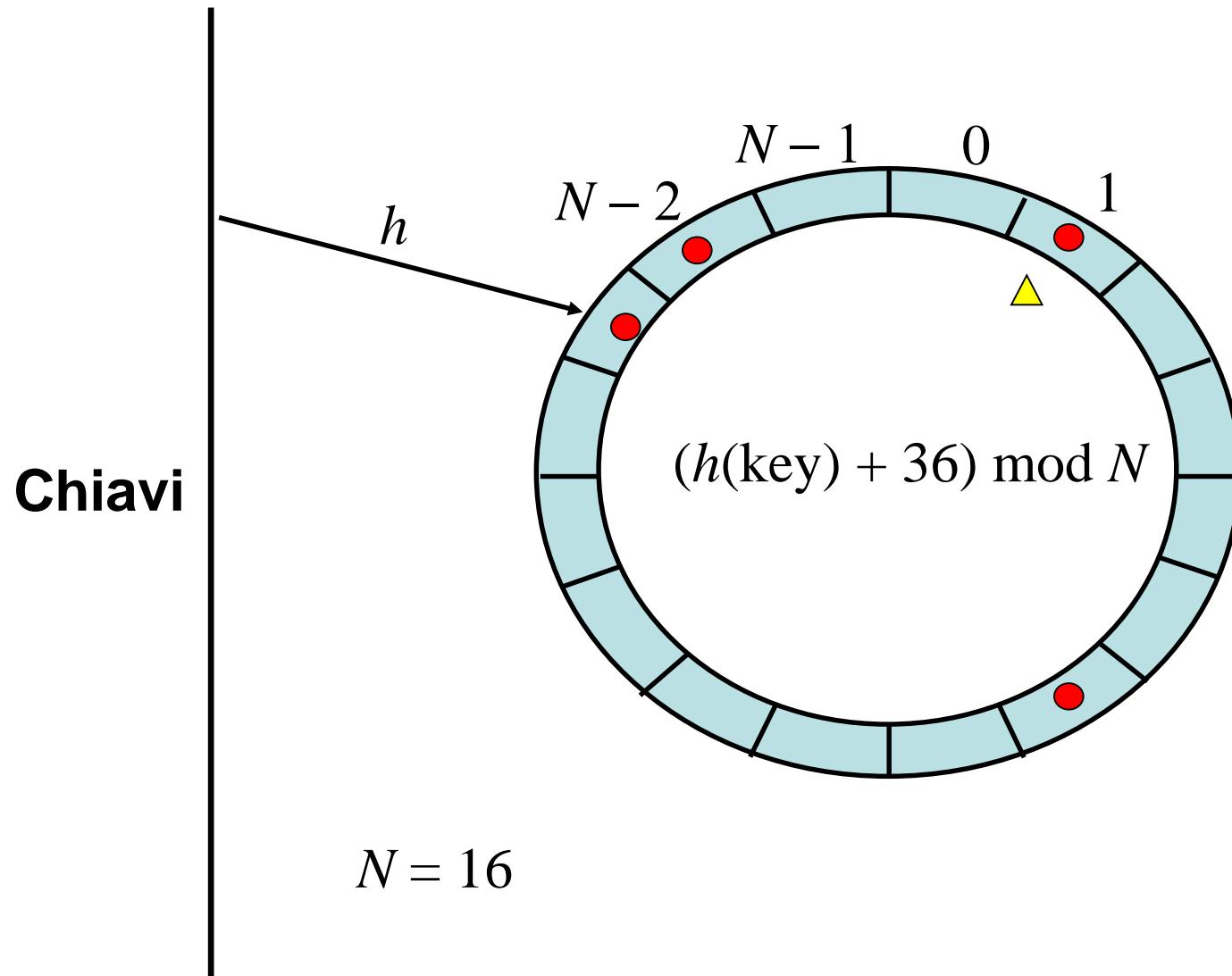


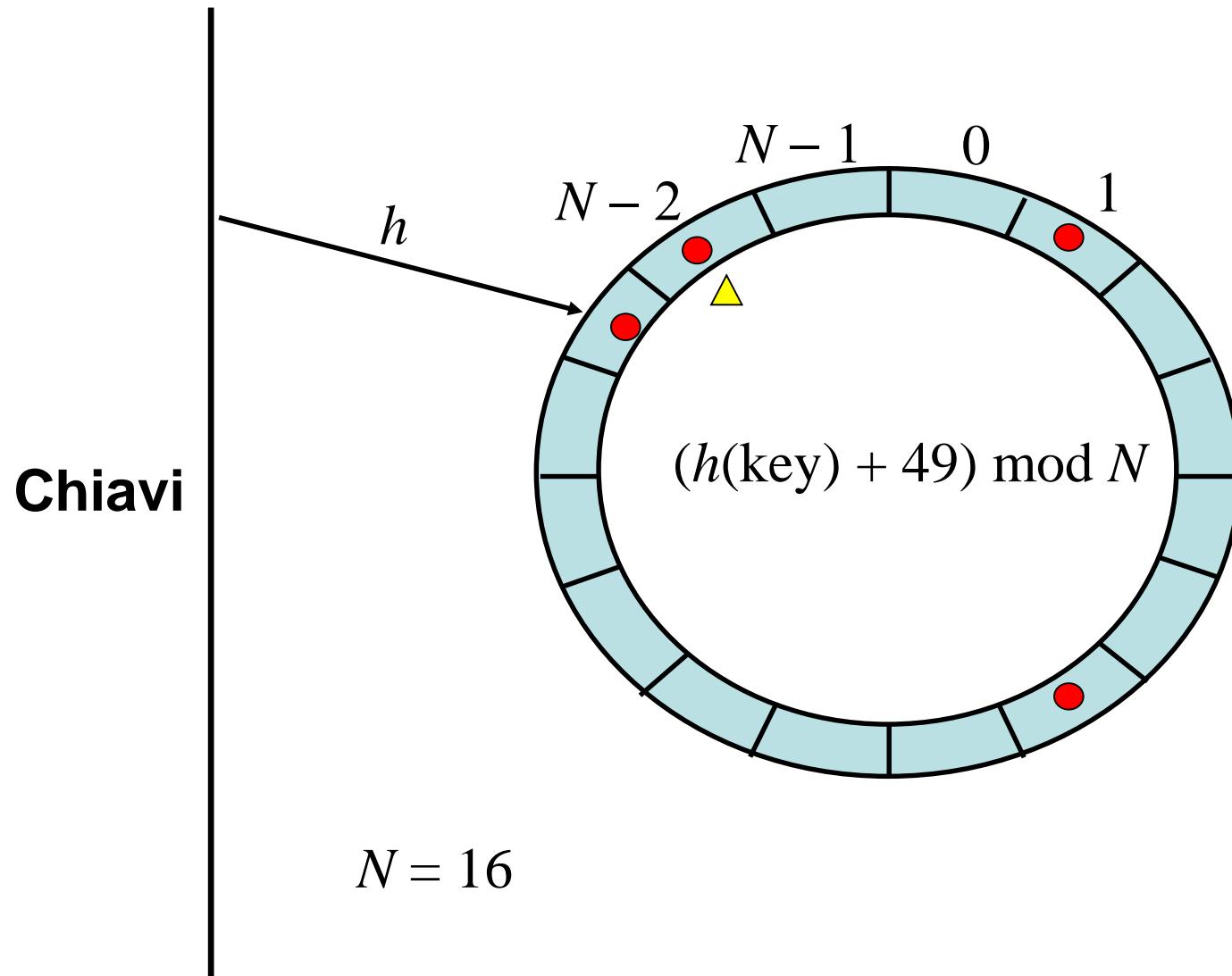






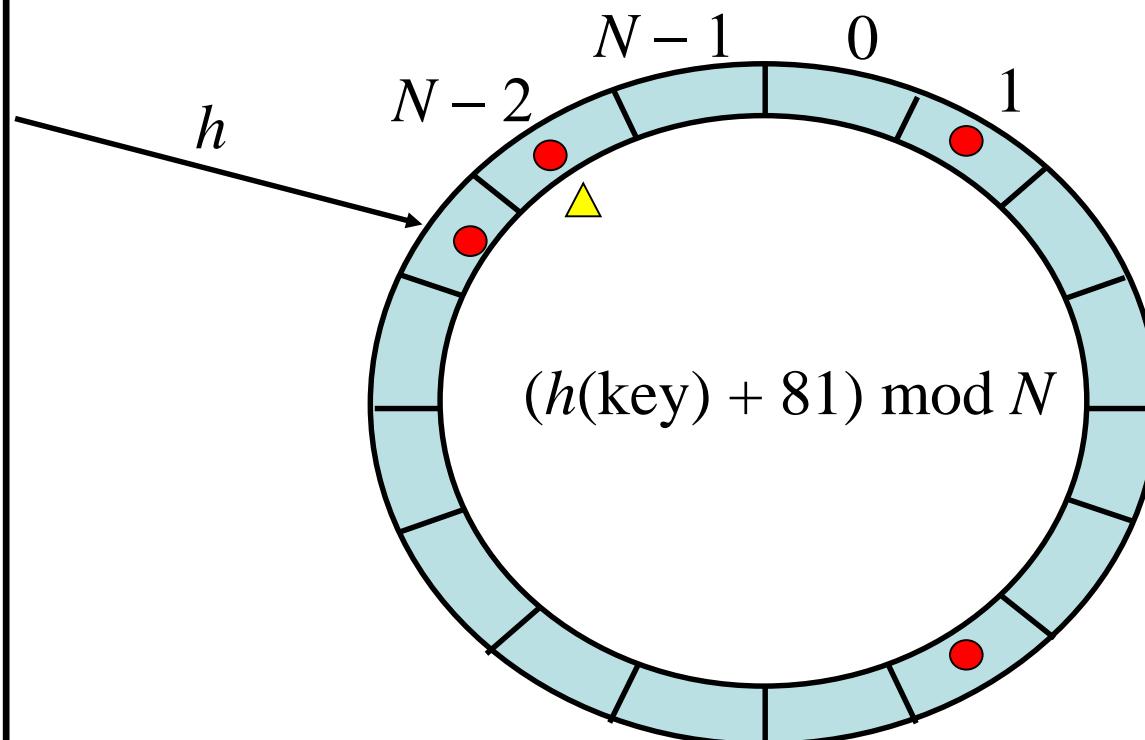




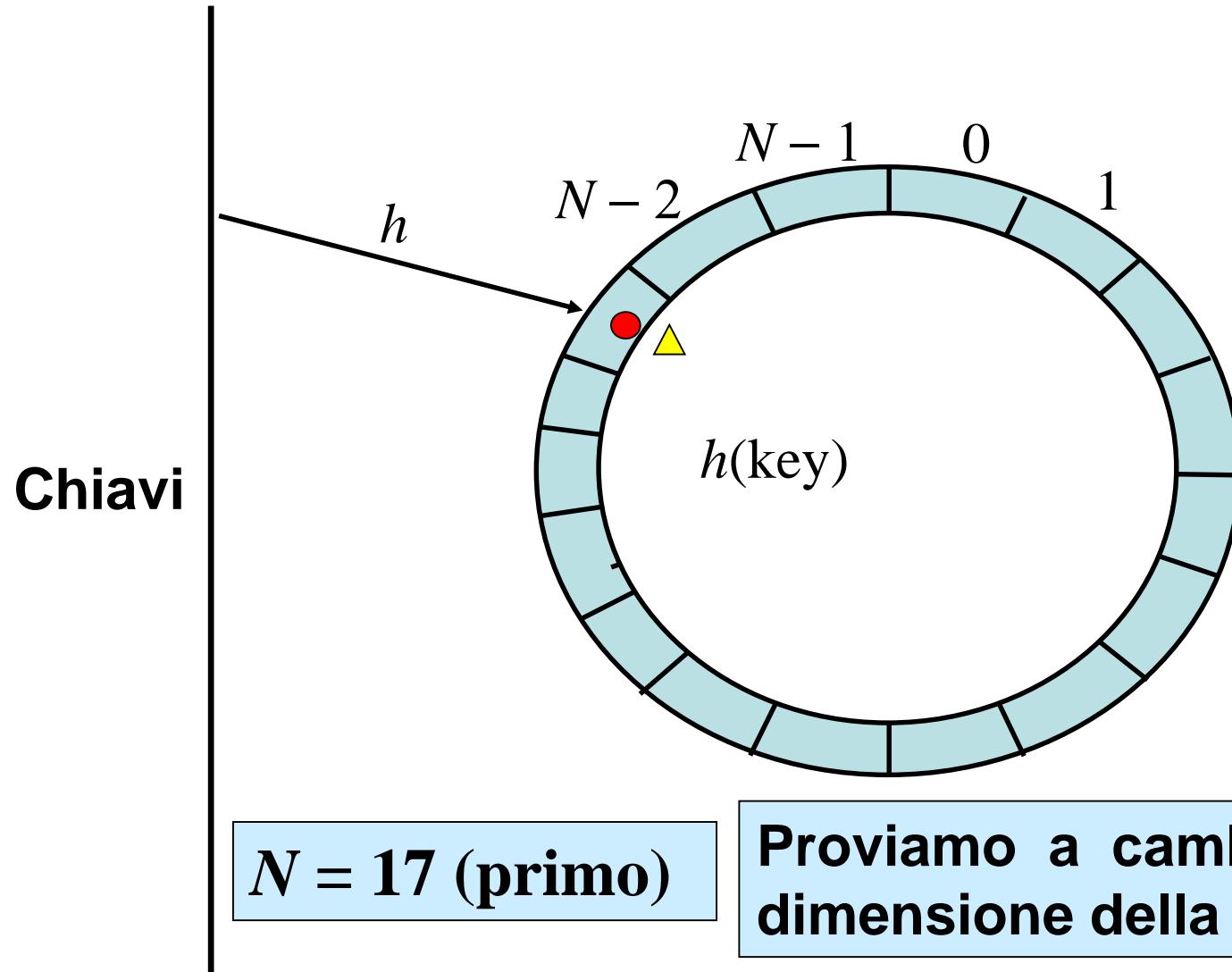


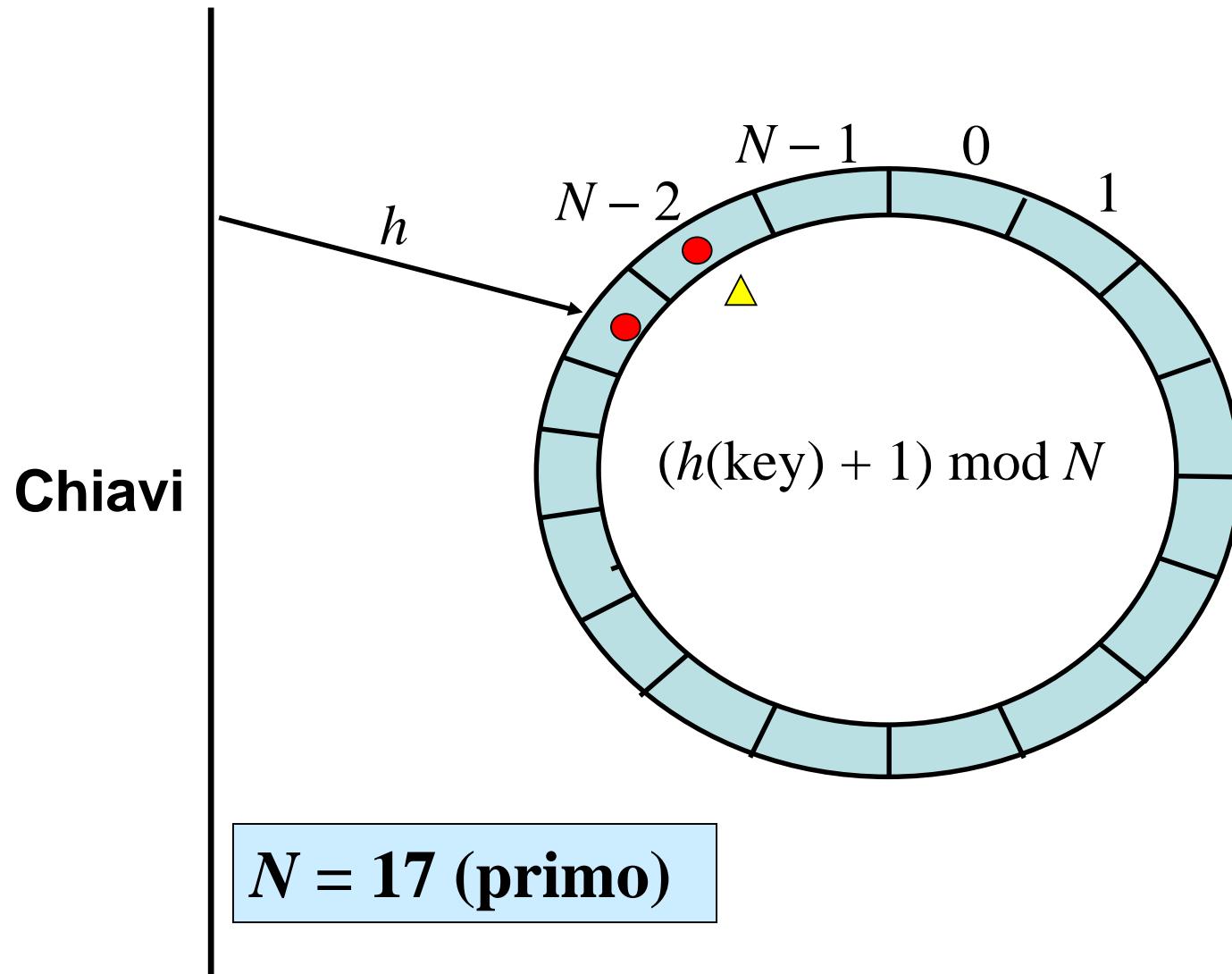
Chiavi

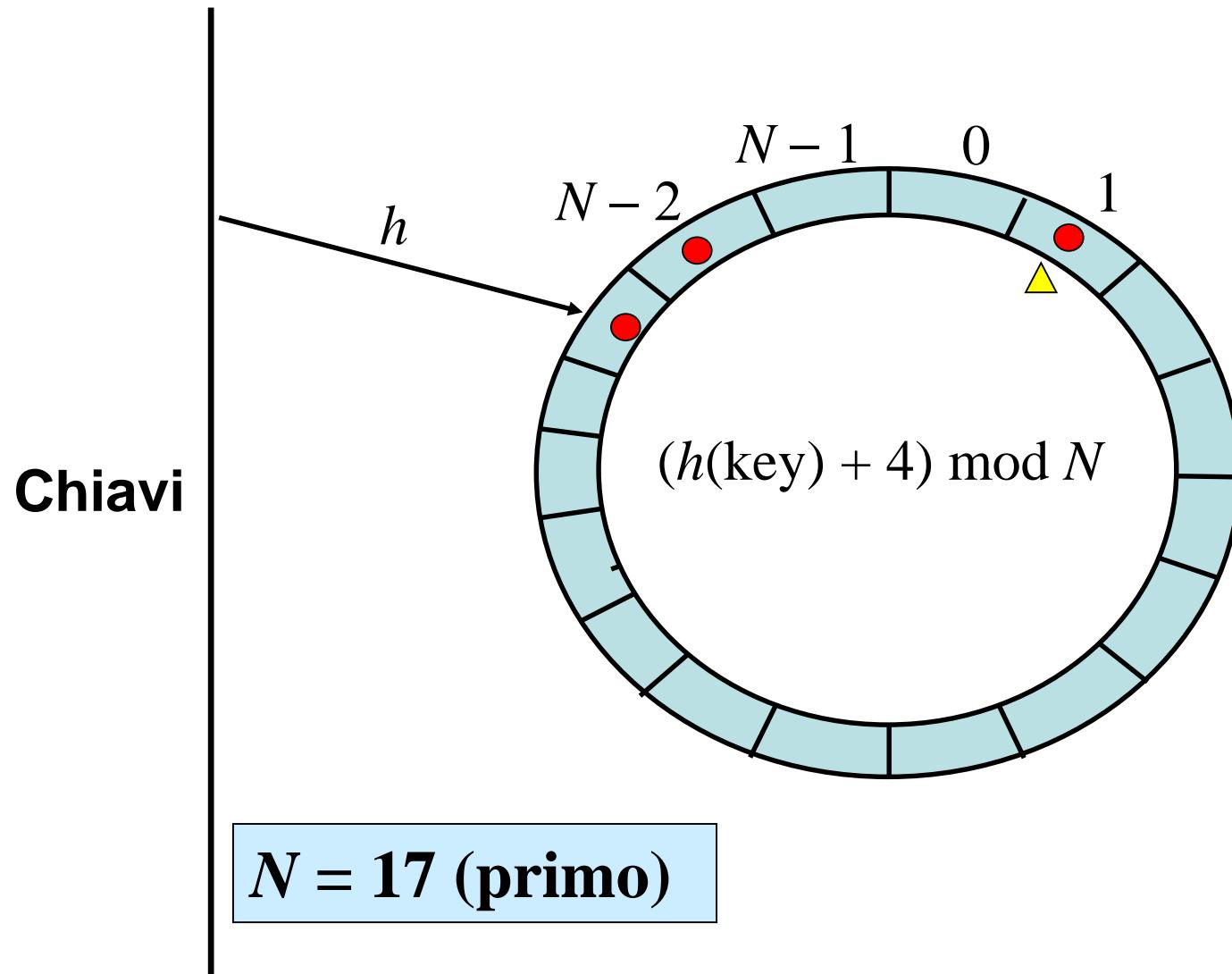
$$N=16$$

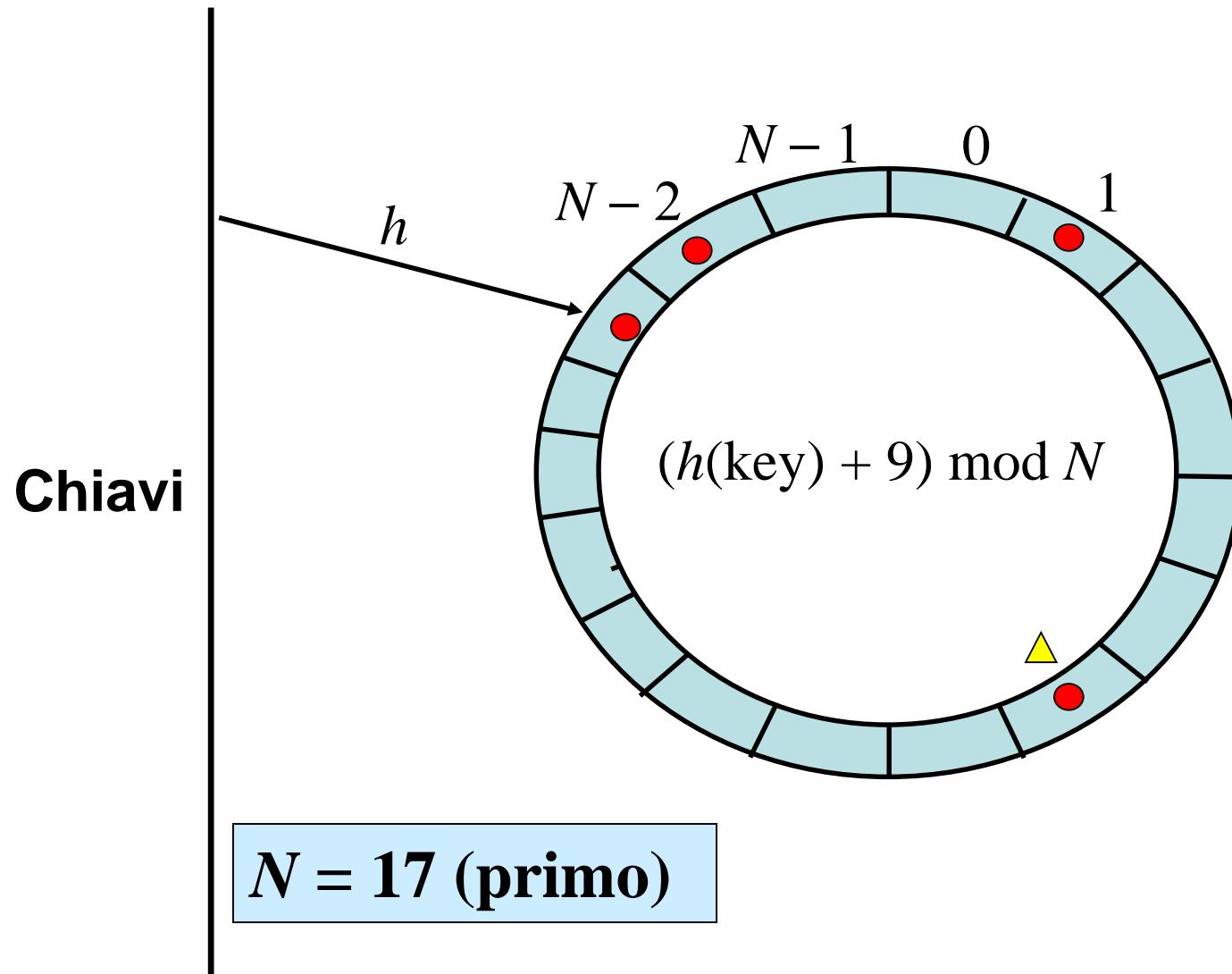


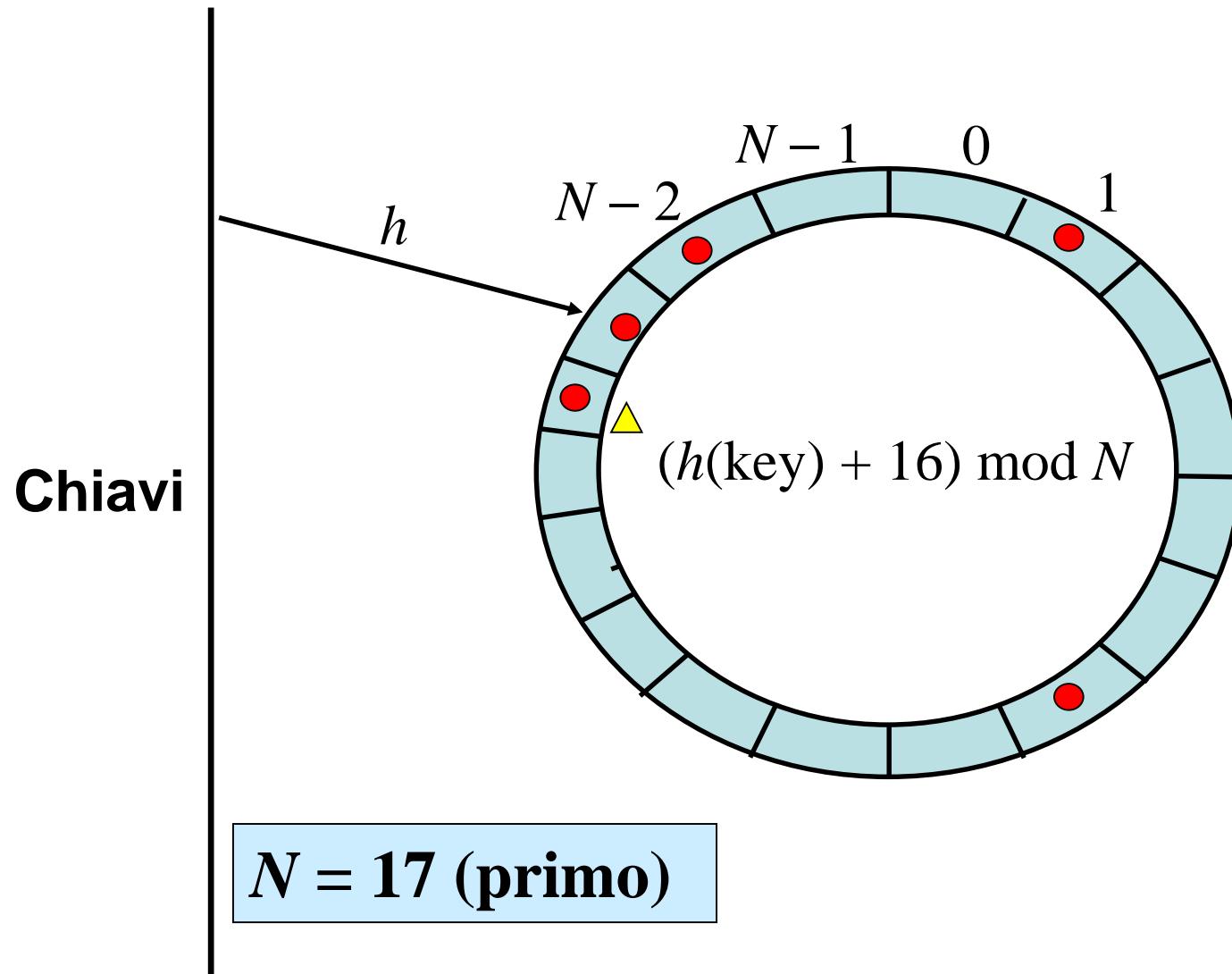
Possono essere sondate solo 4 su 16 locazioni!

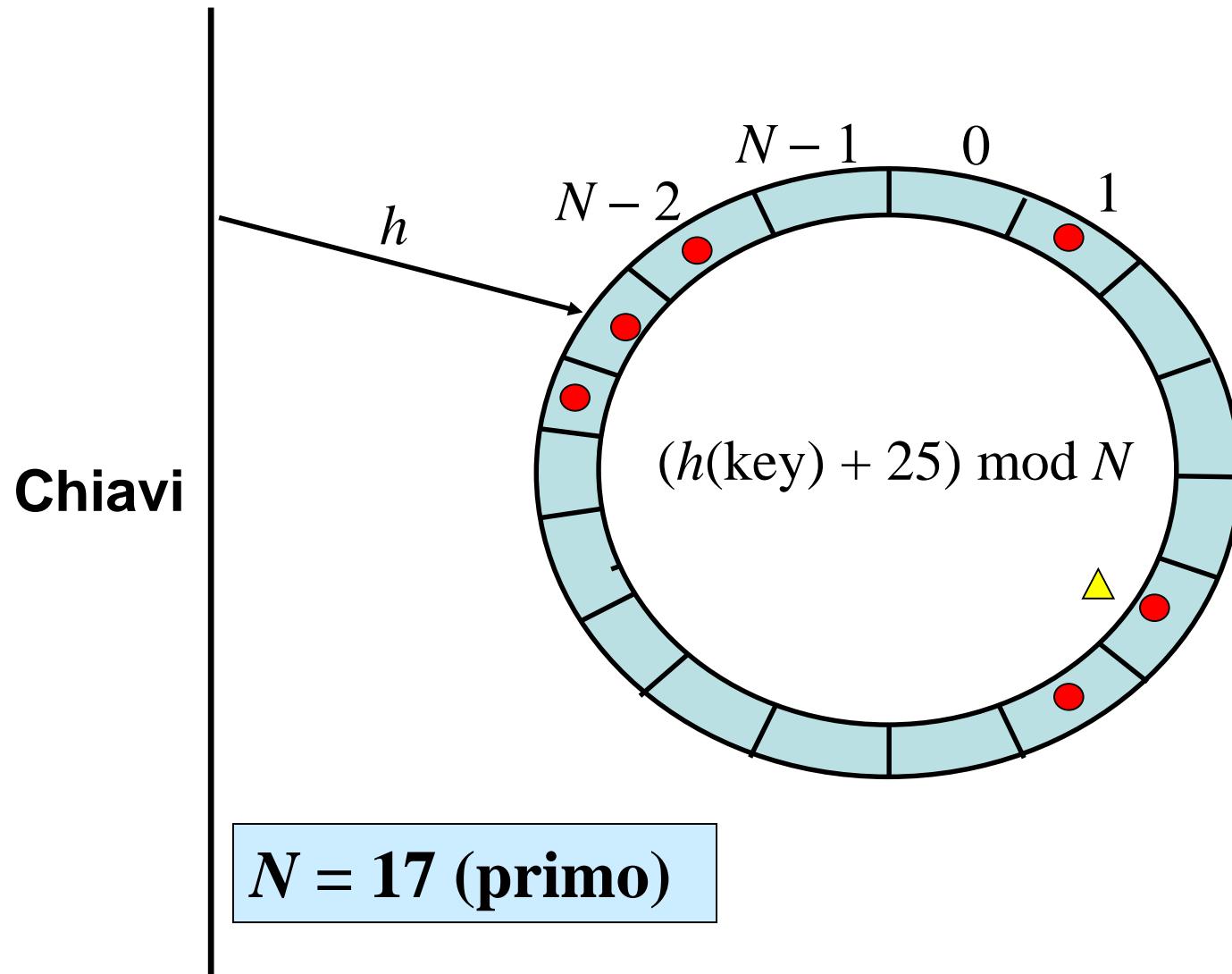


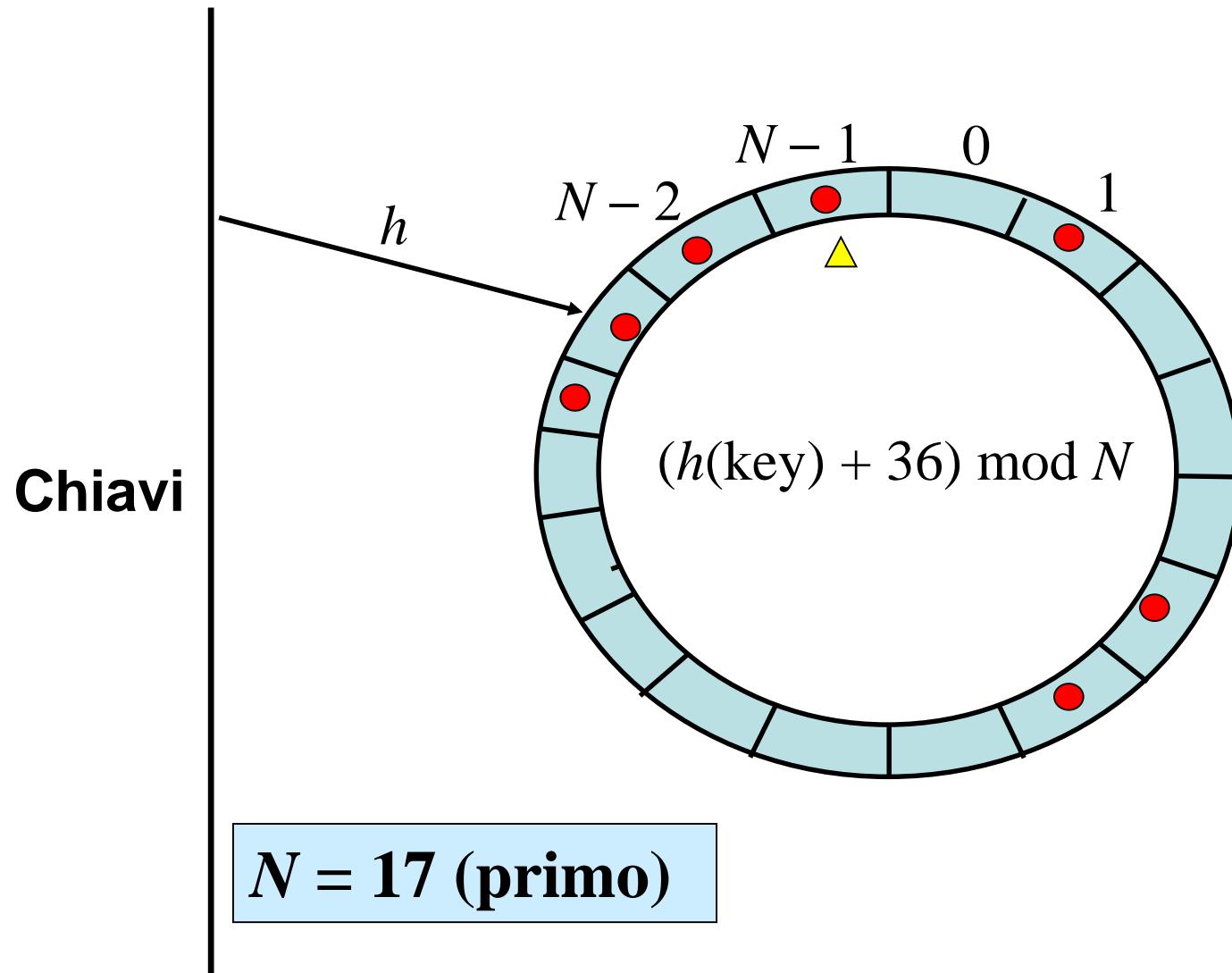


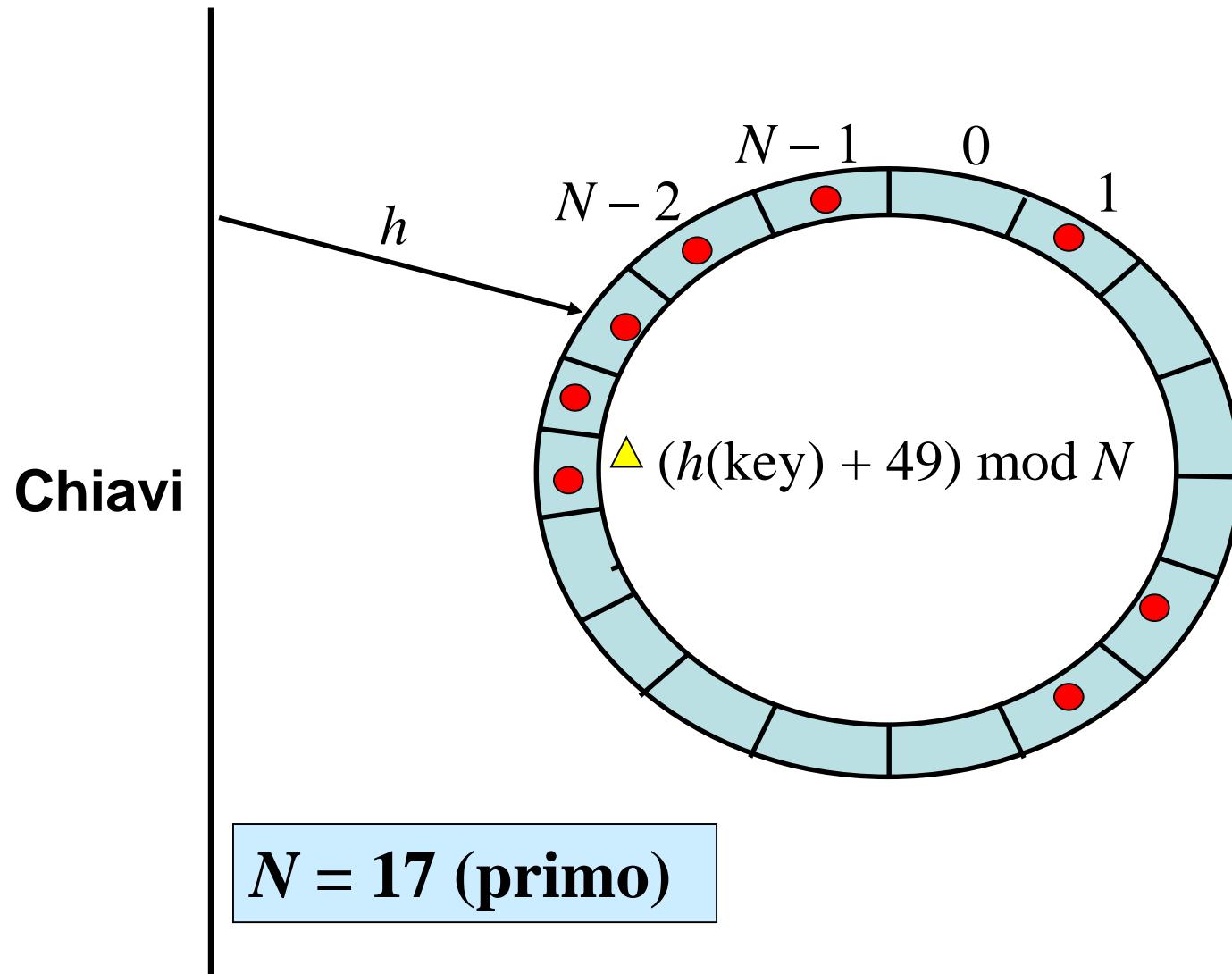


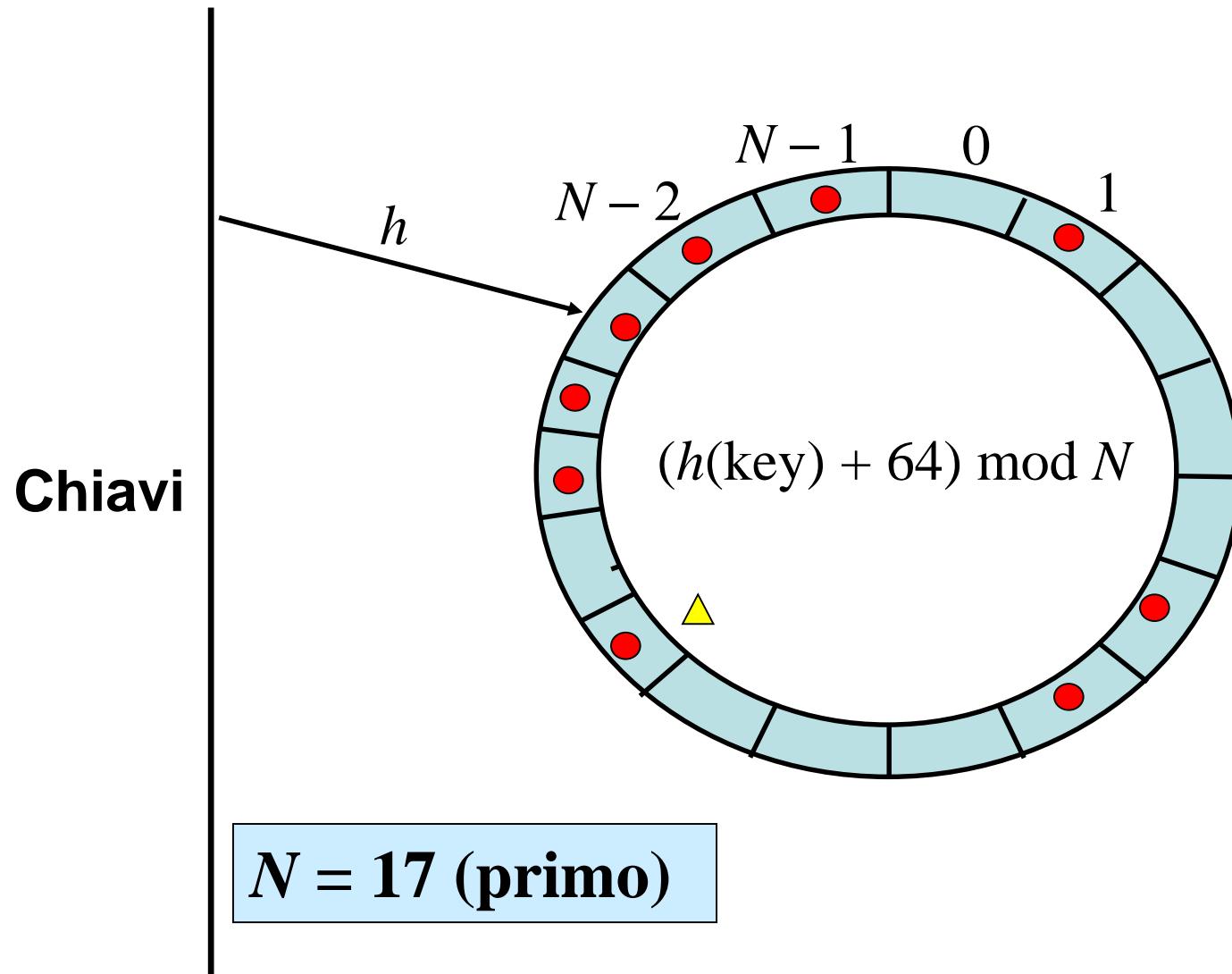


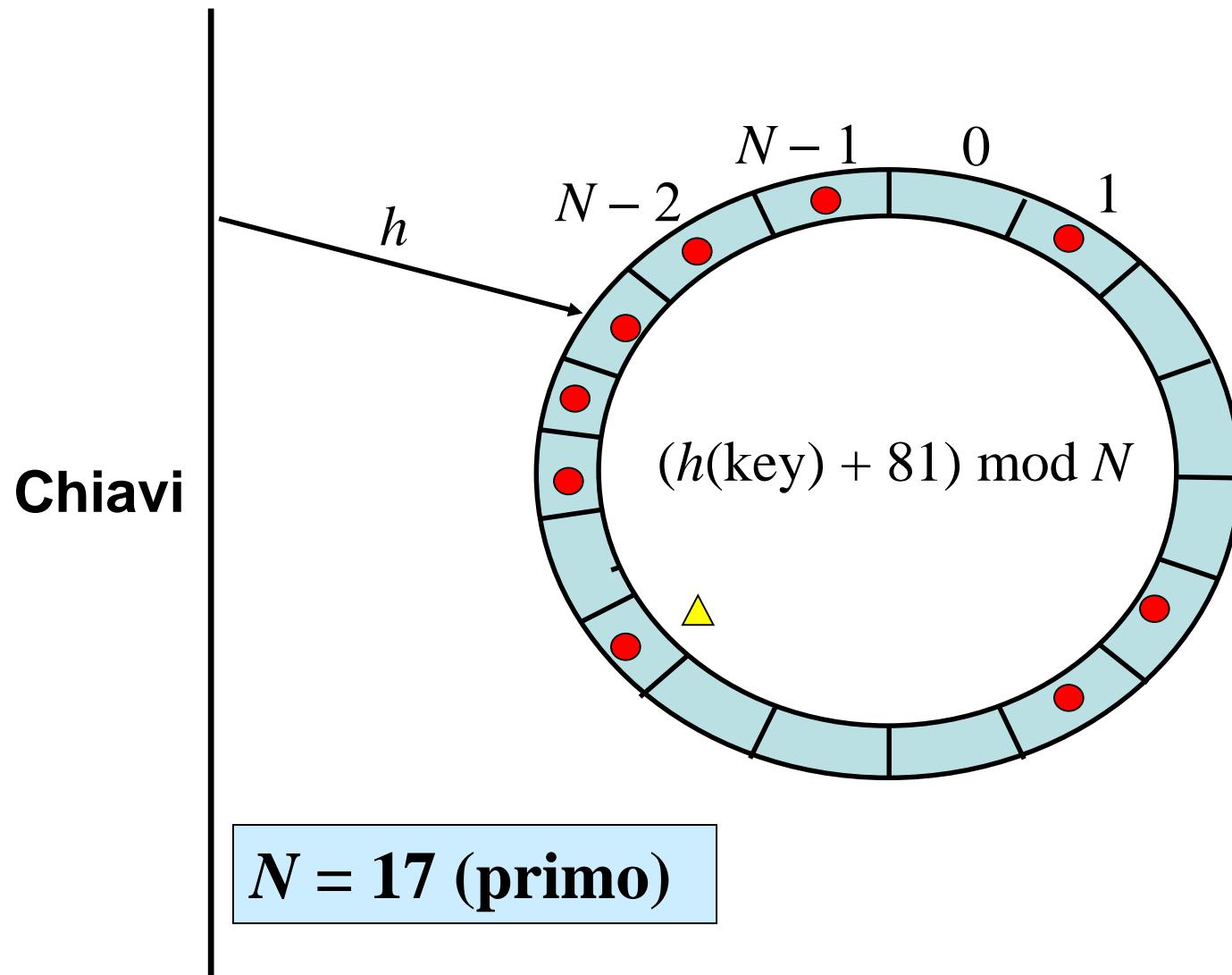


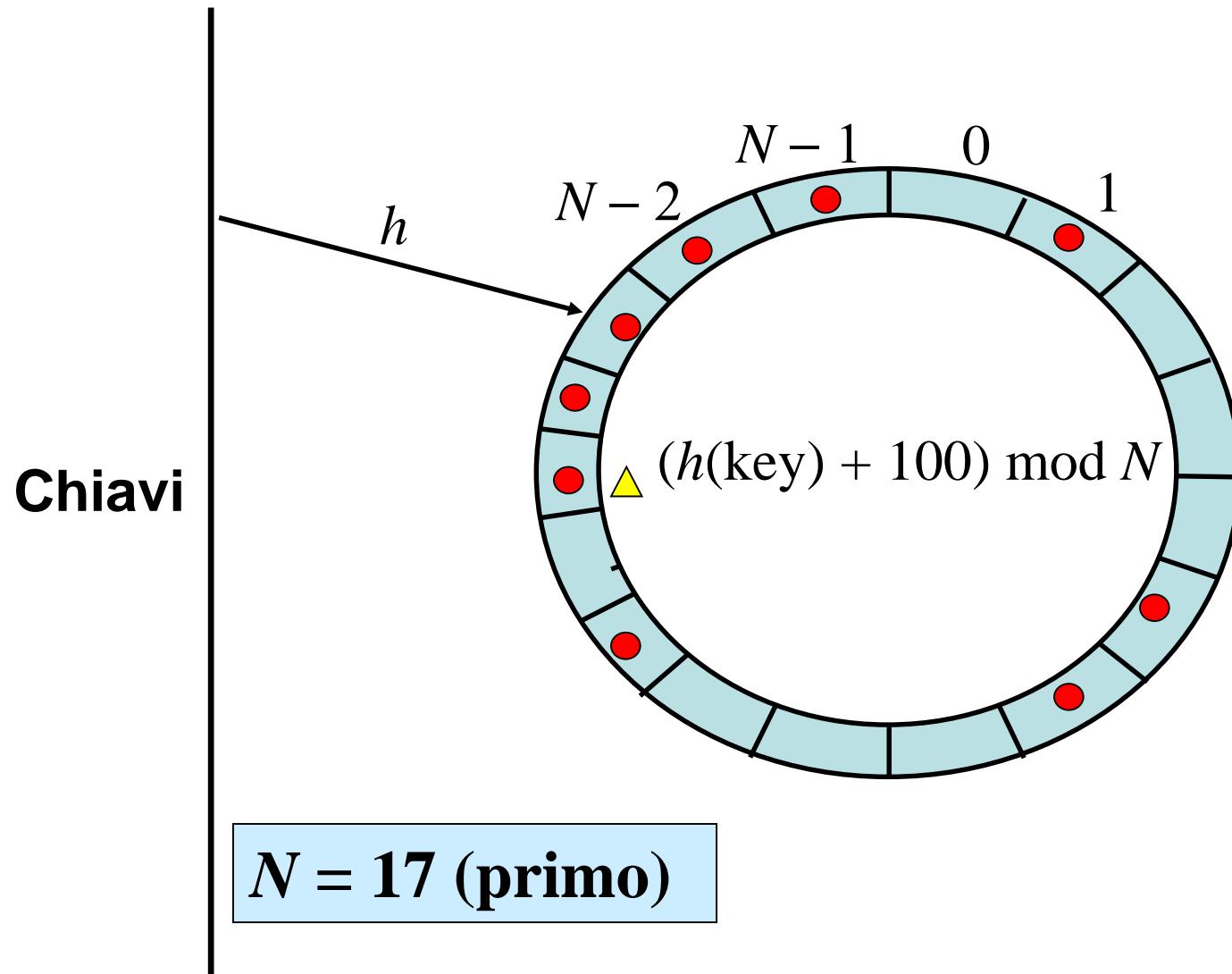


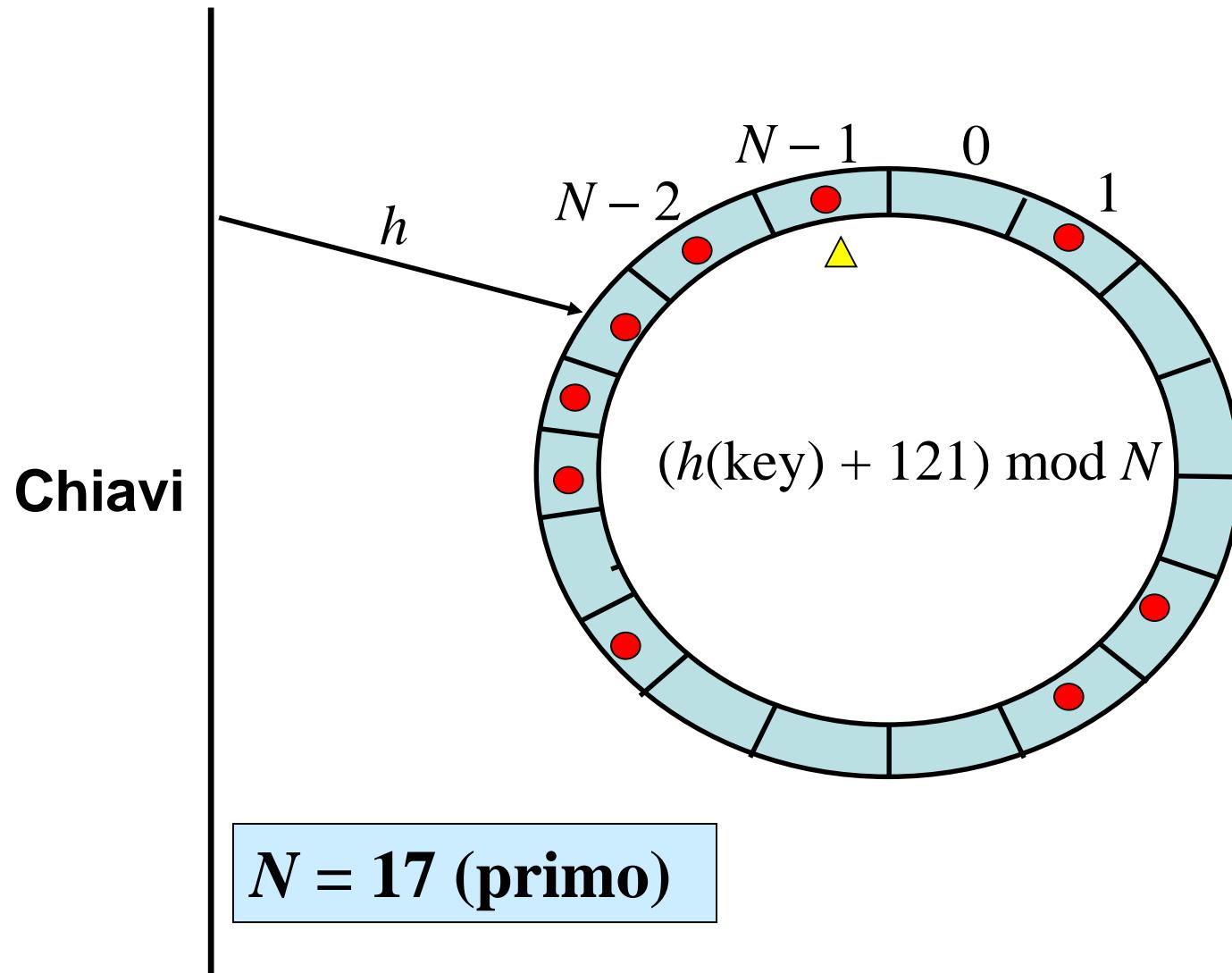


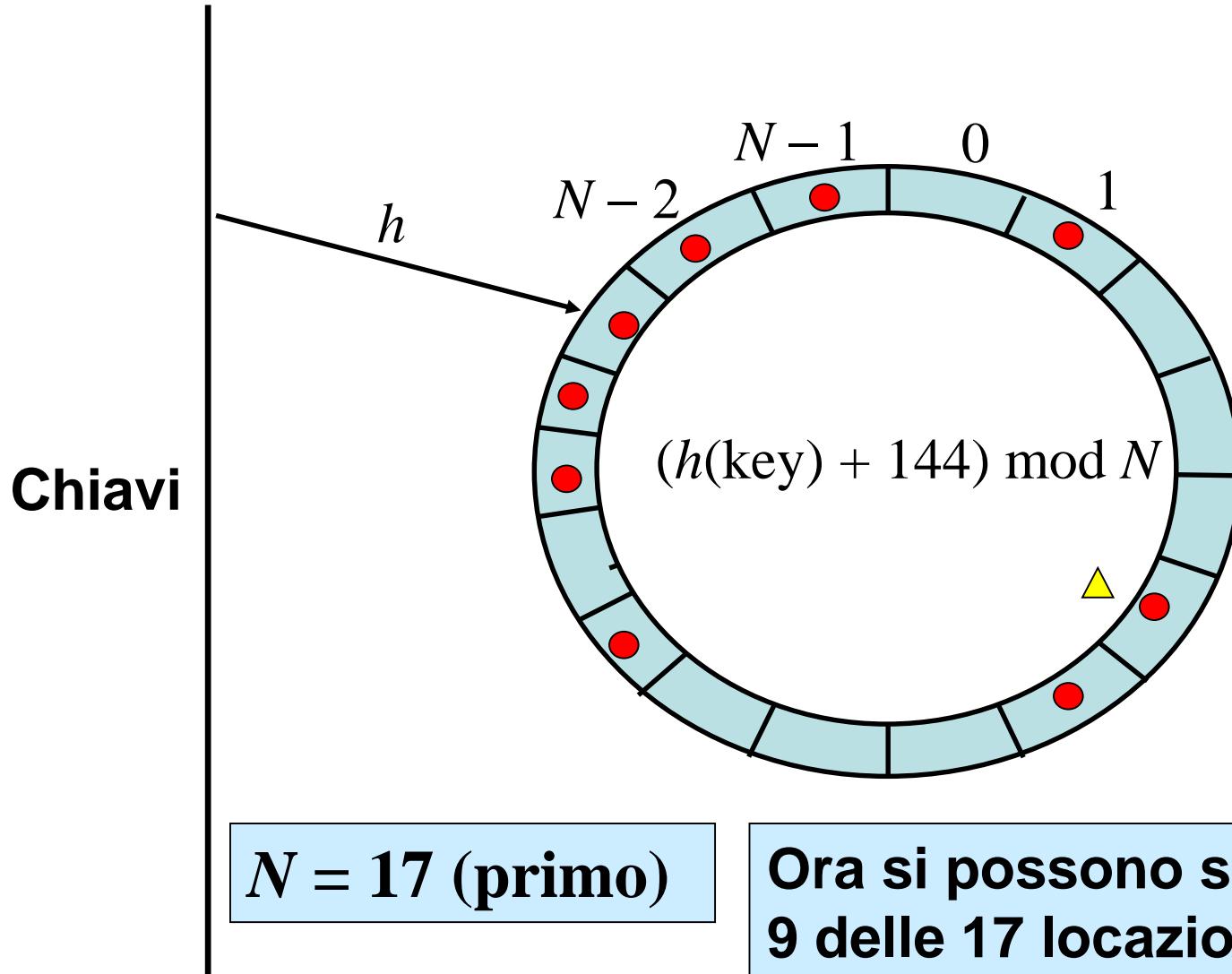












Teorema: Nel sondaggio quadratico con dimensione della tabella numero primo, un nuovo elemento trova sempre spazio se la tabella è vuota almeno per metà.

Dimostrazione:

- Sia la dimensione della tabella, **TSIZE**, un numero primo > 3 , mostreremo che le prime $\lfloor \frac{\text{TSIZE}}{2} \rfloor$ locazioni alternative (sondaggi) sono tutte distinte.
- Sia $0 \leq i < j \leq \lfloor \frac{\text{TSIZE}}{2} \rfloor$ due diversi indici di sondaggio.
- Dimostriamo che
$$(h(x) + i^2) \bmod \text{TSIZE} \neq (h(x) + j^2) \bmod \text{TSIZE}$$
- Assumiamo, al contrario, che
$$(h(x) + i^2) \bmod \text{TSIZE} = (h(x) + j^2) \bmod \text{TSIZE}$$
- Quindi, deve essere $i^2 \bmod \text{TSIZE} = j^2 \bmod \text{TSIZE}$, cioè
 - $(j^2 - i^2) \bmod \text{TSIZE} = 0$
 - $(j-i)(j+i) \bmod \text{TSIZE} = 0$
- ...

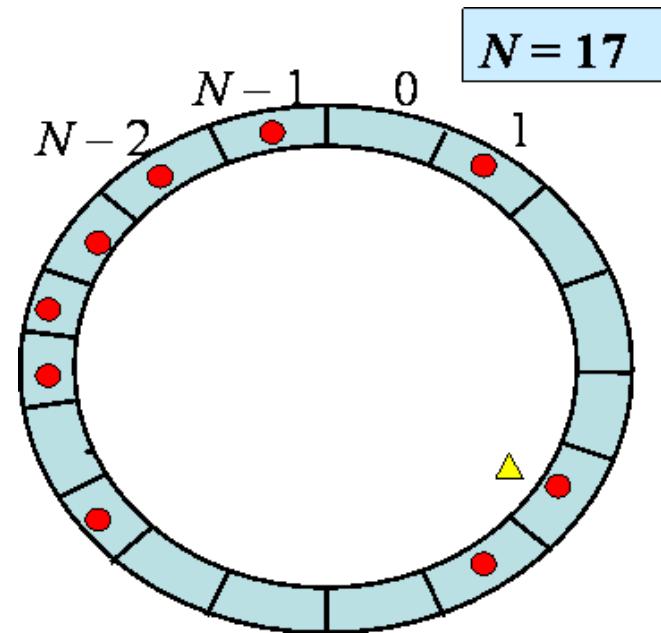
Teorema: Se si usa il sondaggio quadratico, e la dimensione della tabella è un numero primo, allora: un elemento nuovo può sempre essere inserito se la tabella è vuota almeno per metà.

Dimostrazione:

- Quindi, deve essere $i^2 \bmod \text{TSIZE} = j^2 \bmod \text{TSIZE}$, cioè
 - $(j^2 - i^2) \bmod \text{TSIZE} = 0$
 - $(j-i)(j+i) \bmod \text{TSIZE} = 0$
- Ma $0 \leq i < j \leq \lfloor \text{TSIZE}/2 \rfloor$, quindi $j + i < \text{TSIZE}$ (TSIZE è dispari)
 - $(j-i)(j+i) \bmod \text{TSIZE} = 0$ implica che TSIZE divide $(j-i)(j+i)$
- (per **lemma di Euclide**) se un primo p divide $a \cdot b$ allora p divide a o b .
- Poiché TSIZE è primo, deve valere che TSIZE divide $(j-i)$ o $(j+i)$
- Ma entrambi i casi sono **impossibili**, poiché $0 < (j-i), (j+i) < \text{TSIZE}$, quindi TSIZE non può dividere nessuno dei due numeri.
- Concludiamo, quindi, che $i^2 \bmod \text{TSIZE} \neq j^2 \bmod \text{TSIZE}$.

Teorema: Se si usa il **sondaggio quadratico**, e la dimensione della tabella è un **numero primo**, allora un elemento nuovo può sempre essere inserito se la tabella è vuota almeno per metà.

Applicazione: Il sondaggio visita solo 9 delle 17 posizioni, ma se la tabella è metà vuota, non tutte le 9 posizioni possono essere occupate, quindi possiamo inserire un nuovo elemento in una di esse.



Esempio di sondaggio quadratico

Clustering secondario: gli elementi con lo **stesso valore primario di hash**, genereranno la stessa sequenza di sondaggi:

- in pratica non costituisce un grosso problema

Sia data una **tabella hash** con **fattore di carico** pari a $\lambda = m/n < 1$ (m numero di chiavi inserite, n dimensione della tabella), allora:

- Il **numero medio di sondaggi** per una ricerca **senza successo** è al massimo:

$$\frac{1}{1 - \lambda}$$

Il **numero medio di sondaggi** per una **ricerca con successo** è al massimo:

$$\frac{1}{\lambda} \cdot \ln\left(\frac{1}{1 - \lambda}\right)$$

Sondaggio casuale (random)

- usare un **generatore di numeri pseudo-casuali** per ottenere una sequenza, ripetibile nella stessa sessione di esecuzione, per identificare lo *i*-esimo sondaggio.

$$h(k,i) = (\text{hash}(k) + r_i) \bmod N$$

- dove r_i è l'*i*-esimo valore in una **permutazione random** dei numeri **1...N-1**.
- più semplice da implementare del sondaggio quadratico ma soffre degli stessi problemi: **clustering secondario**.
- infatti la sequenza di sondaggio **dipende solo dalla** dal valore di hash primario **hash(k)**.

Tutte le soluzioni viste permettono di generare **N** differenti sequenze di sondaggio (una per ogni locazione della tabella).

Doppio Hashing

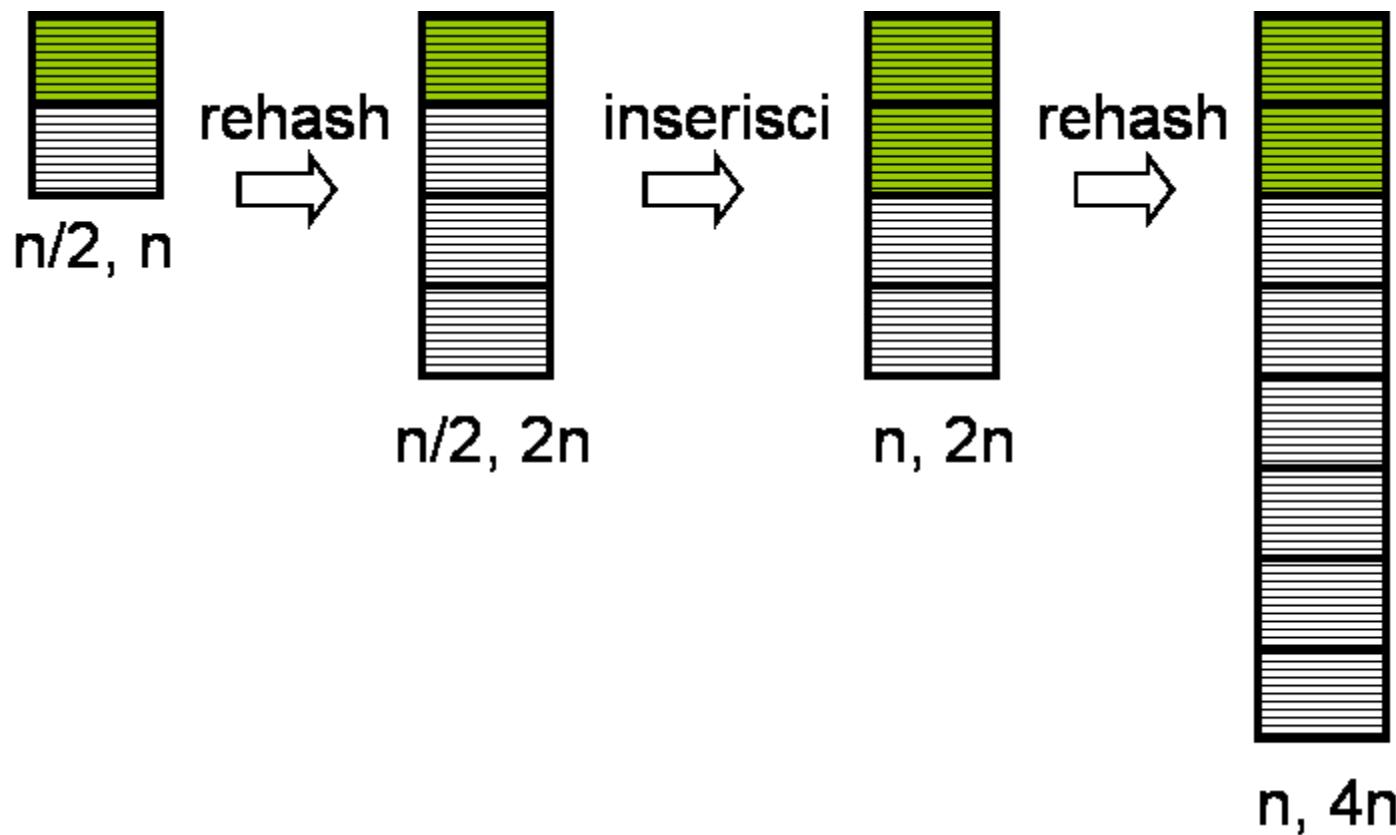
- usare una **seconda funzione hash** per trovare una locazione alternativa
$$h(k,i) = (h_1(k) + i \cdot h_2(k)) \bmod N$$
- è la soluzione migliore in quanto permette di generare **N^2** **differenti sequenze di sondaggio** (la sequenza, infatti, dipende in *due modi differenti*, e tra loro indipendenti, dalla chiave **k**).
- **$h_2(k)$** deve essere **relativamente primo** con **N** , altrimenti non tutta la tabella potrebbe venire ispezionata.
- es. 1: **N** primo, **$h_1(k) = k \bmod N$** e **$h_2(k) = 1 + (k \bmod N-1)$** .
- es. 2: **$N = 2^p$** e **$h_2(k)$** produce sempre **numeri dispari**.

Cancellazioni

- le *cancellazioni* con l'*indirizzamento aperto* sono *difficili*
- la **cancellazione "lazy"** è il metodo preferibile
 - si marcano gli elementi cancellati come non allocati, in modo che possano essere riutilizzati
 - quando il numero di record cancellati raggiunge qualche livello di soglia (predeterminato), si può effettuare un'operazione di "**garbage collection**" per riottenere le locazioni cancellate (**perché è utile?**)

Nel caso di **indirizzamento aperto**

- Tabella troppo piena
 - Tempo di esecuzione troppo lungo
 - Inserimento può fallire (quando?)
- La dimensione **DEVE** essere scelta in anticipo
 - Ma non sappiamo il numero di elementi
- **Rehashing**
 - Costruire una nuova tabella di dimensione circa il doppio
 - Rimappare (via Hash) tutti gli elementi nella nuova tabella (*perché?*)



Reashing: per $n/2$ elementi, la dimensione della tabella è n . Il costo addizionale per ogni inserimento è $O(n)/(n/2) = O(1)$, dove $O(n)$ è il costo per rimappare nella nuova tabella tutte le chiavi presenti in tabella.

- Quando applicare l'**estensione** della **tabella hash**?
 - La tabella è piena per metà
 - L'inserimento fallisce
 - λ raggiunge una soglia **t** predeterminata
- Quale deve essere la **dimensione** della nuova tabella?
 - circa **2** volte più grande o, più in generale,
 - circa **x** volte più grande: con **x = 1.5, 2, 2.5, 3**

Si supponga uno schema di hashing in cui

- Valore iniziale di $n = \text{TSIZE} = 7$
- Funzione di hashing: $h(x) = x \bmod \text{TSIZE}$
- Risoluzione delle collisioni con sondaggio lineare
- Soglia di rehashing $m/n = 66\%$ (con m numero di chiavi presenti)
- Inserimento delle chiavi: 19, 10, 5, 22, 15
- Rehashing con nuovo valore $\text{TSIZE} = 17$

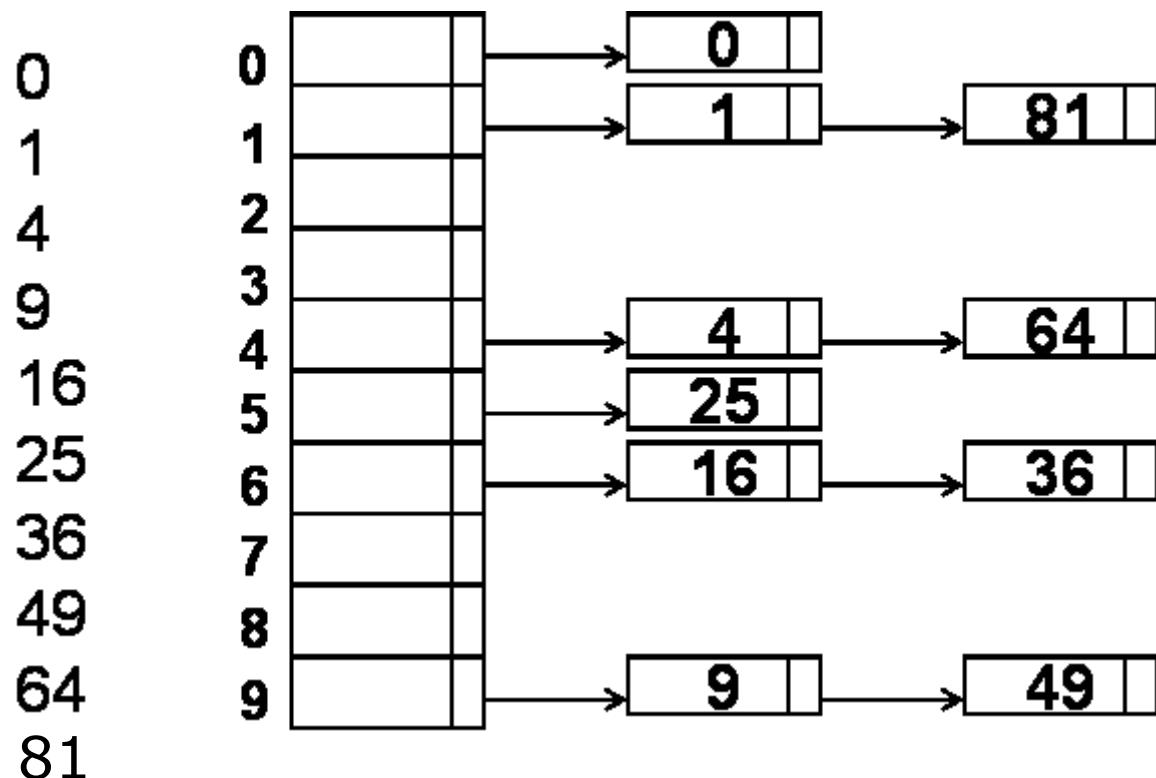
La figura a destra illustra il risultato della operazione di **rehashing**.

0	
1	
2	19
3	
4	
5	
6	22
7	
8	
9	
10	5
11	
12	
13	
14	
15	10
16	15
17	
18	
19	
20	
21	
22	
23	
24	
25	
26	
27	
28	
29	
30	
31	
32	
33	
34	
35	
36	
37	
38	
39	
40	
41	
42	
43	
44	
45	
46	
47	
48	
49	
50	
51	
52	
53	
54	
55	
56	
57	
58	
59	
60	
61	
62	
63	
64	
65	
66	
67	
68	
69	
70	
71	
72	
73	
74	
75	
76	
77	
78	
79	
80	
81	
82	
83	
84	
85	
86	
87	
88	
89	
90	
91	
92	
93	
94	
95	
96	
97	
98	
99	

Esempio di Rehashing.

- È necessaria una **Struttura Dati Secondaria**
 - Lista
 - Albero
 - Una seconda Tabella Hash
- Se ci aspettiamo poche collisioni, possiamo utilizzare un lista
 - Semplice
 - Poco lavoro aggiuntivo necessario
 - Inserimenti nelle catene in tempo costante
- **Vantaggi:** permette **maggior utilizzo** della tabella e non necessita di rehashing.
- **Svantaggi:** maggiore occupazione di memoria (puntatori) e, in generale, accessi meno veloci.

Indirizzamento Chiuso: concatenamento

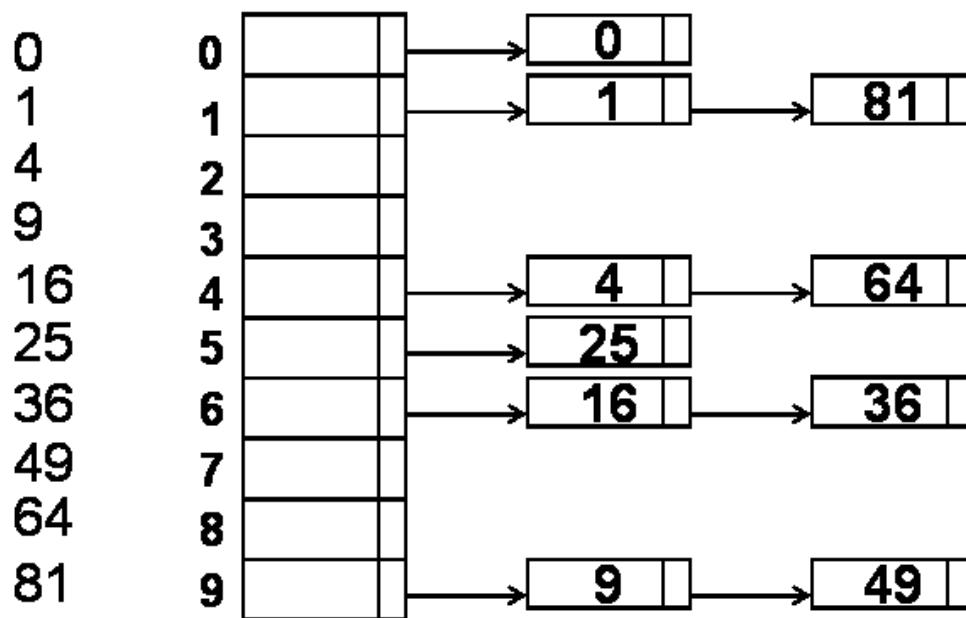


Risultato dell'inserimento delle chiavi a sinistra con funzione di hashing
hash(x) = x mod 10.

Lunghezza media lista di concatenamento

$$\frac{\sum_{i=0}^{T\text{SIZE}-1} l_i}{T\text{SIZE}} = \frac{4 * 2 + 2 * 1 + 4 * 0}{10} = 1$$

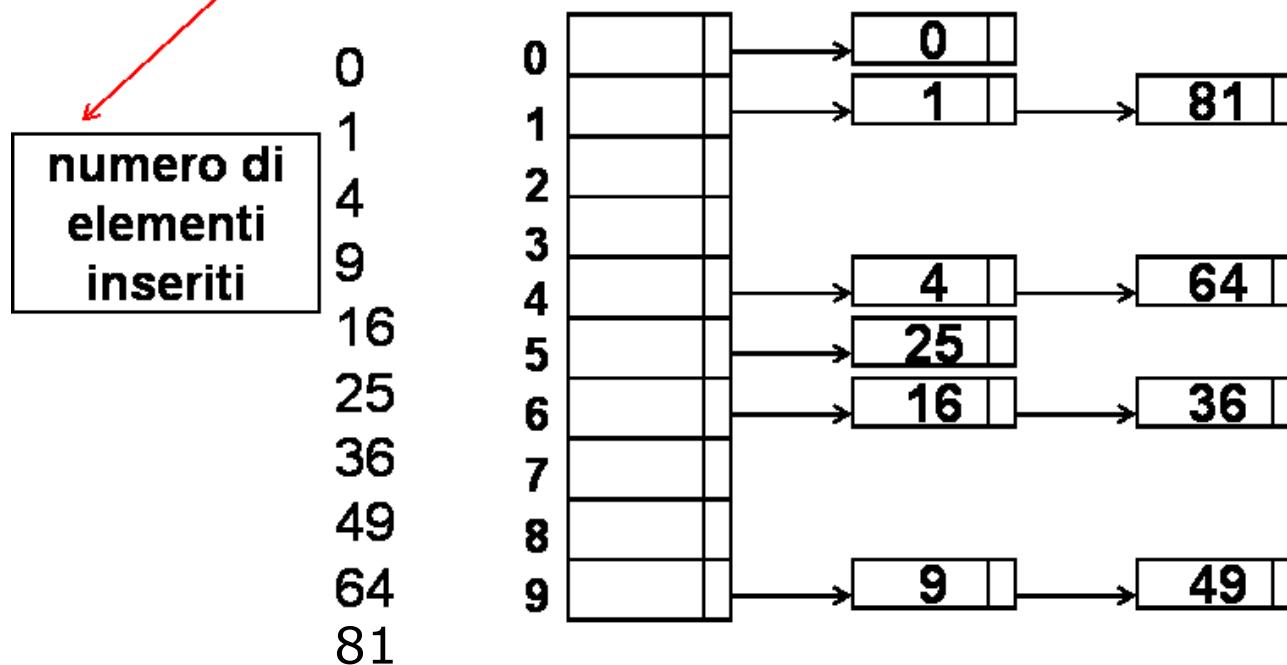
l_i = lunghezza
lista i -esima



Calcolo della lunghezza media di una lista di collisione.

l_i = lunghezza
lista i -esima

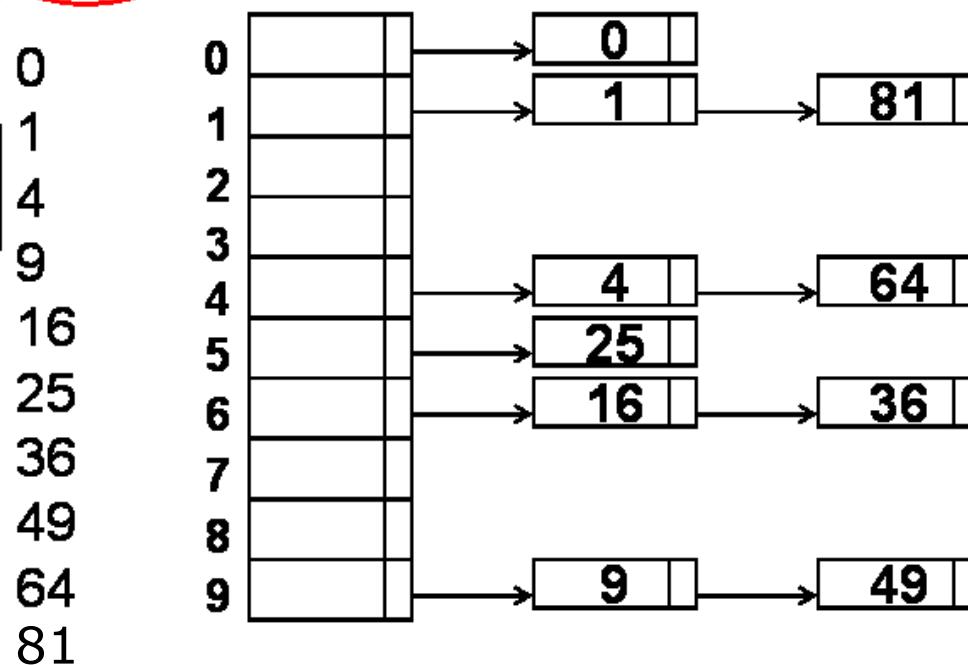
$$\frac{\sum_{i=0}^{TSIZE-1} l_i}{TSIZE} = \frac{4 * 0 + 2 * 1 + 4 * 2}{10} = 1$$



l_i = lunghezza
lista i -esima

$$\frac{\sum_{i=0}^{TSIZE-1} l_i}{TSIZE} = \frac{4 * 0 + 2 * 1 + 4 * 2}{10} = 1$$

Fattore di
carico λ



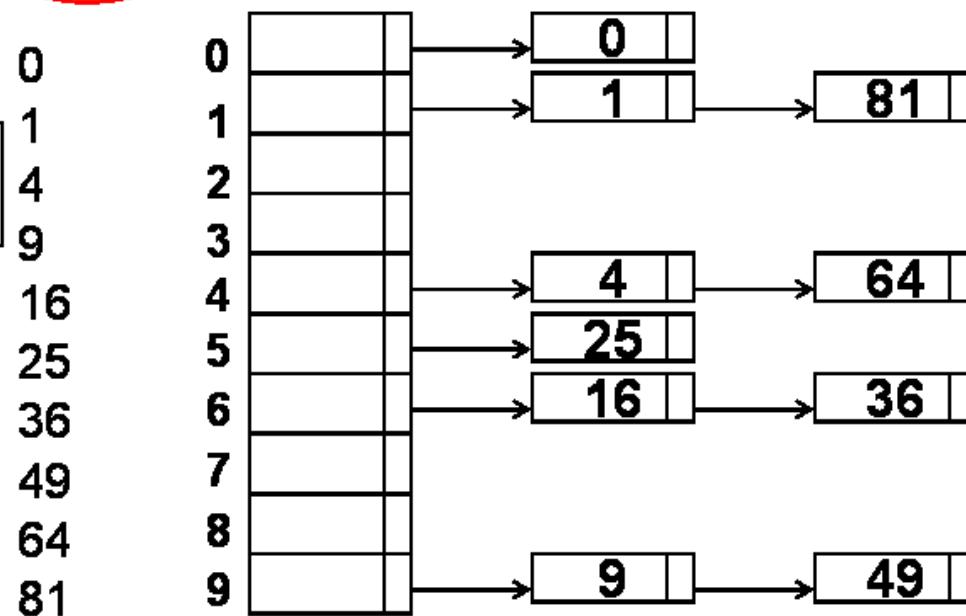
Calcolo della lunghezza media di una lista di collisione.

$$\frac{\sum_{i=0}^{T\text{SIZE}-1} l_i}{T\text{SIZE}}$$

$$\frac{4 * 0 + 2 * 1 + 4 * 2}{10} = 1$$

Il **fattore di carico**
 λ è la lunghezza
media delle liste

Fattore di
carico λ



La **lunghezza media di una lista di collisione** equivale al **fattore di carico**

- $\lambda = m/n$: numero di *elementi inseriti* (**m**) *diviso* per la *dimensione della tabella* (**n**).
- λ : rappresenta la *lunghezza media di una lista*
- Tempo medio di esecuzione della ricerca
 - hashing (**$\Theta(1)$**) + attraversamento della lista
 - elemento non trovato: λ
 - elemento trovato: **$1 + \lambda/2$**
 - Almeno **1** link viene ispezionato
 - La lunghezza media di una lista è λ
 - Ci si aspetta *in media* di trovare un elemento presente dopo aver attraversato *metà della lista*.
- Regola euristica: mantenere $\lambda \approx 1$.

Data una **tabella hash** con **fattore di carico** $\lambda = m/n$ (m numero di chiavi inserite, n dimensione della tabella), allora:

- il tempo medio impiegato per una **ricerca senza successo** è al massimo:

$$\Theta(1 + \lambda)$$

- il tempo medio impiegato per una **ricerca con successo** è al massimo:

$$\Theta(1 + \lambda)$$

- Vantaggi del concatenamento a liste
 - con una buona funziona hash le *liste puntate risultano corte*
 - il *clustering* non è un problema — gli elementi con chiavi differenti stanno su catene differenti
 - la *dimensione della tabella è meno rilevante*
 - le *cancellazioni sono facili ed efficienti*
 - le *catene* potrebbero anche essere implementate come *alberi binari di ricerca* o loro varianti più efficienti

- *Dimensione della tabella: Numero Primo*
- *Funzione Hash*: basata su *operazioni di modulo*
- *Gestione delle Collisioni*
 - Indirizzamento chiuso (concatenamento)
 - Indirizzamento aperto (sondaggi)
 - Sondaggio lineare, quadratico, doppio
- λ : *fattore di carico* (numero di elementi diviso per la dimensione della tabella).
 - Indirizzamento chiuso: $\lambda \approx 1$
 - Indirizzamento aperto: $\lambda \approx 0.5$
- *Rehashing*