

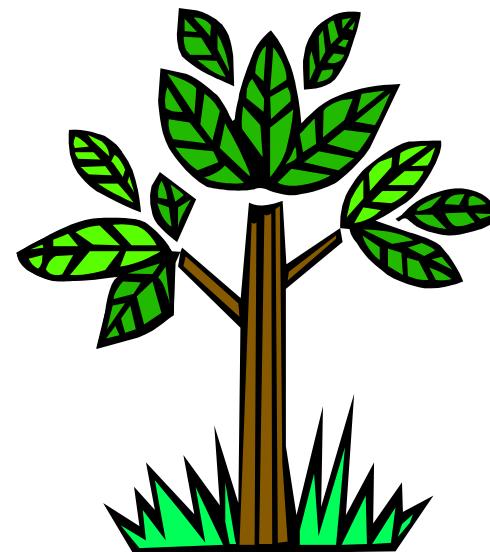
Alberi



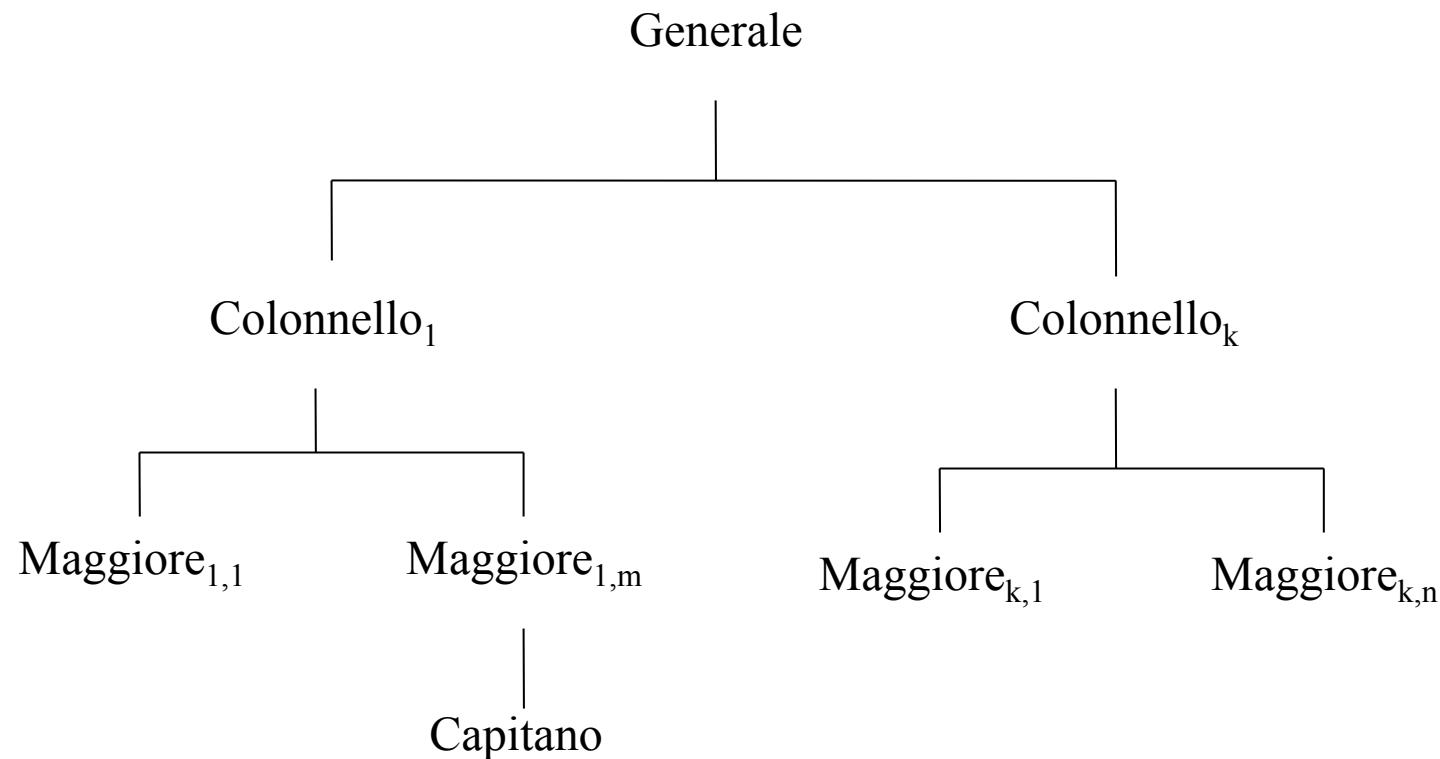
Algoritmi e strutture dati
Lezione 11, a.a. 2016-17

Ugo de'Liguoro, Andras Horvath

Cosa sono gli alberi?



Strutture gerarchiche di ogni tipo



Strutture gerarchiche di ogni tipo

Strutture dati

1. Tipi di dato e strutture dati
 1. Specifica e realizzazione
 2. Rappresentazione in memoria
2. Liste
 1. L'ADT delle liste
 2. Realizzazione con vettori
 3. Realizzazione con puntatori
3. Pile e code
 1. L'ADT delle pile ...

Definizione

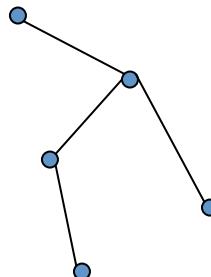
Dato un insieme A , l'insieme degli alberi su A , $\text{Tree}(A)$, è definito induttivamente:

$$a \in A, T_1, \dots, T_k \in \text{Tree}(A) \quad (k \geq 0)$$

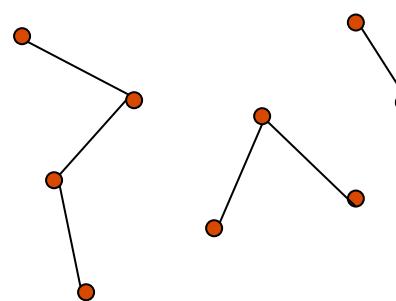
$$\Rightarrow \{a, T_1, \dots, T_k\} \in \text{Tree}(A)$$

Alberi come grafi

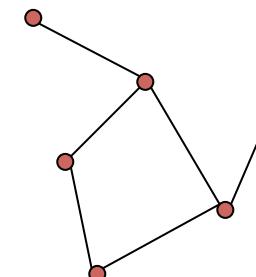
- Un *albero* è un grafo connesso aciclico; nel caso finito può essere definito induttivamente come un insieme tale che:
- Un insieme di alberi è una *foresta*.



albero libero

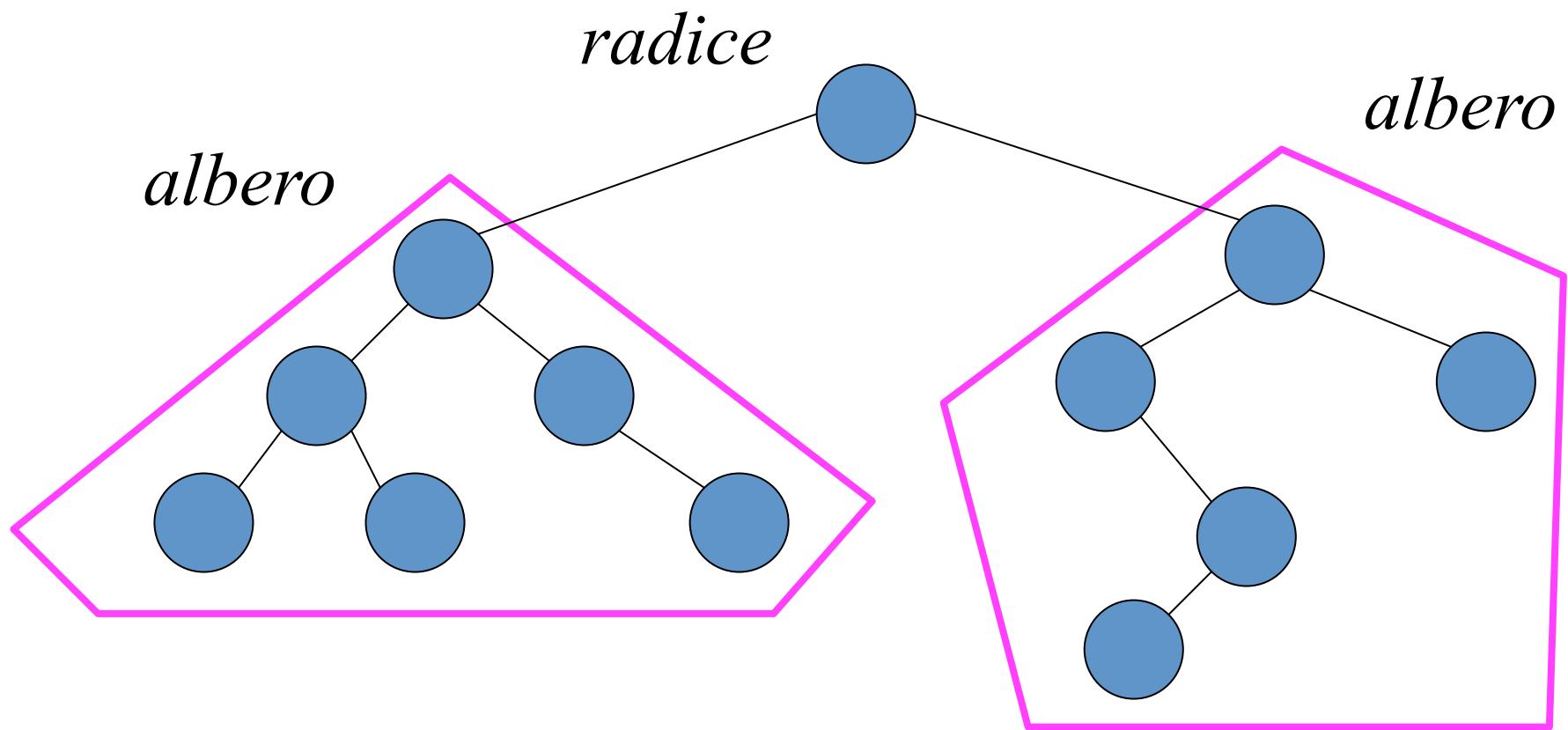


foresta



grafo ciclico

Alberi come grafi

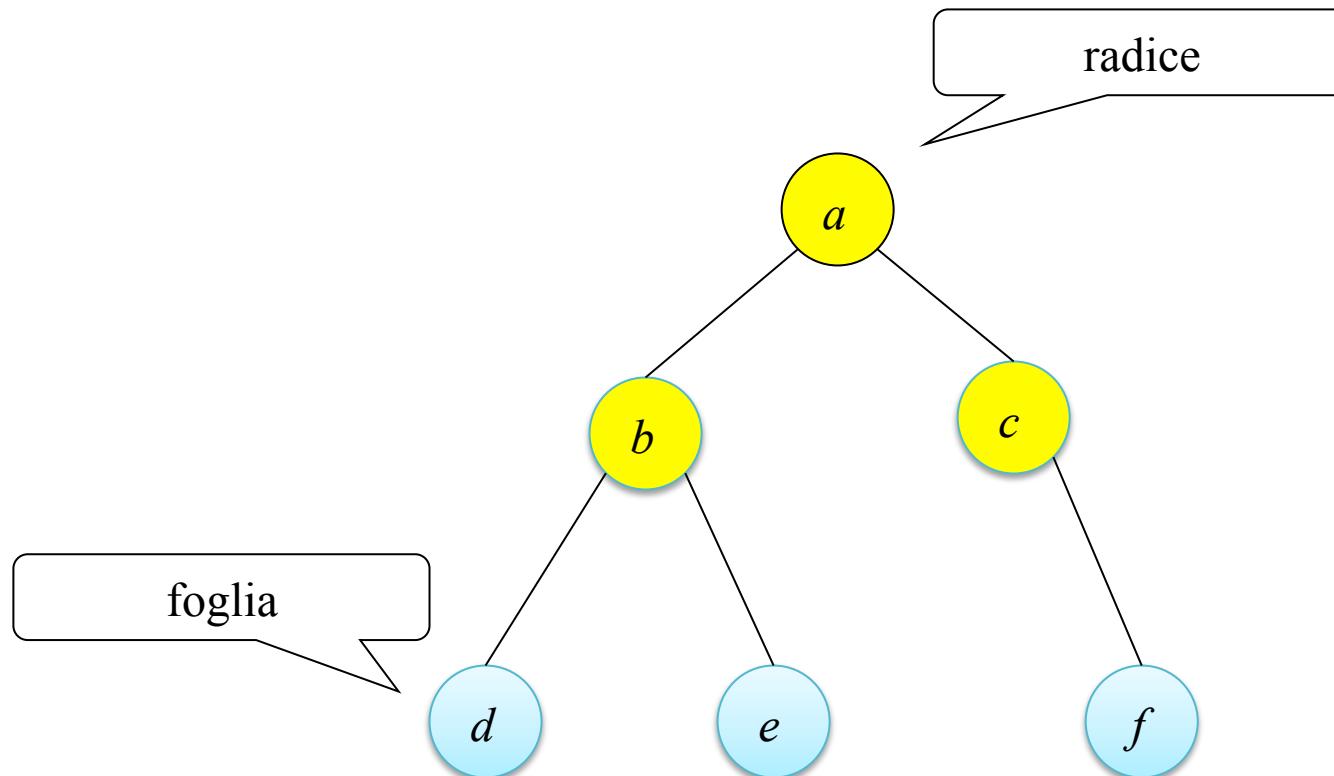


Alberi radicati

- La *radice* è un nodo privilegiato di un albero;
- Una *foglia* è un nodo da cui non esce alcun arco;
- Un nodo che non sia una foglia si dice *interno*

Alberi come grafi

$\{a, \{b, \{d\}, \{e\}\}, \{c, \{f\}\}\} =$

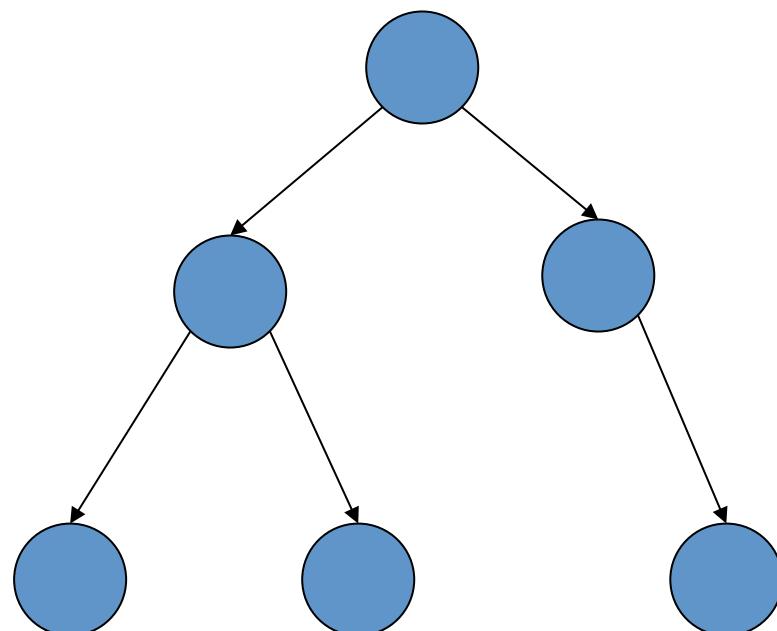


Alberi orientati

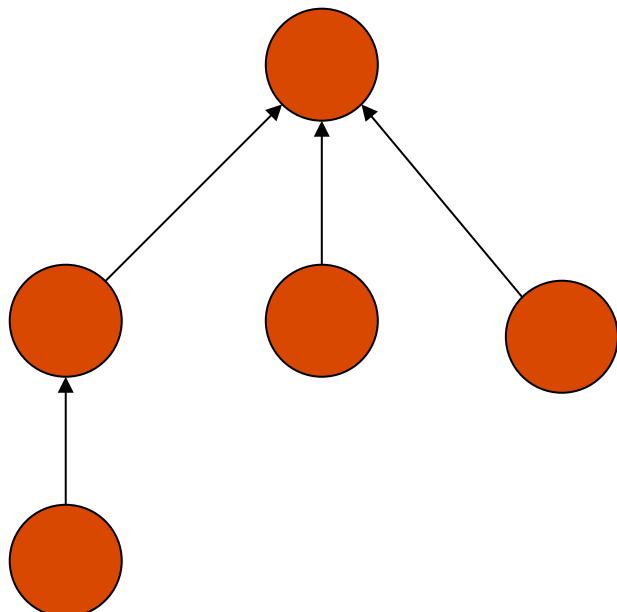
Se l'albero è un grafo orientato allora due casi:

1. la radice ha solo archi in uscita (albero *sorgente*)
2. la radice ha solo archi in entrata (albero *pozzo*)

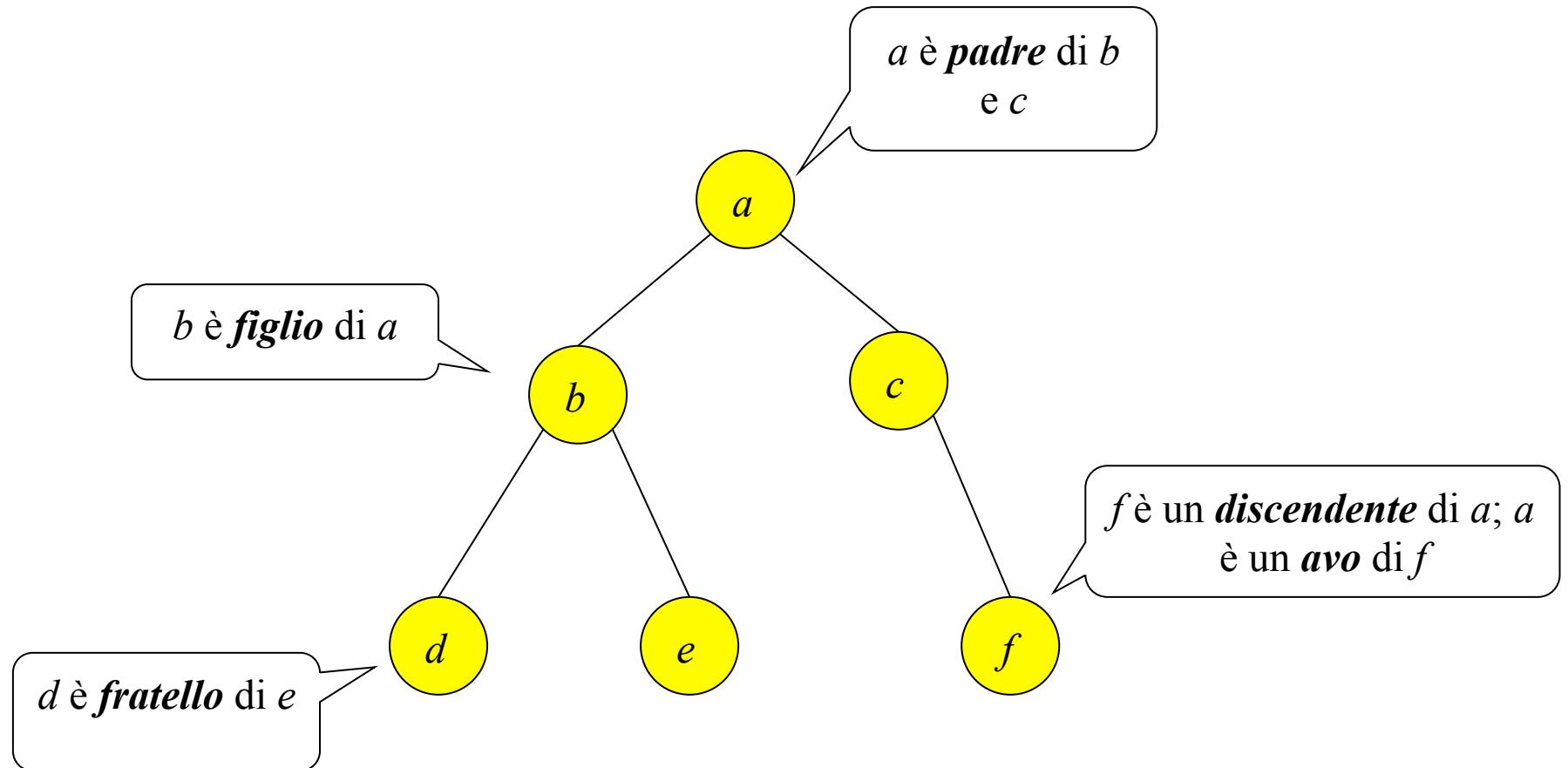
sorgente



pozzo



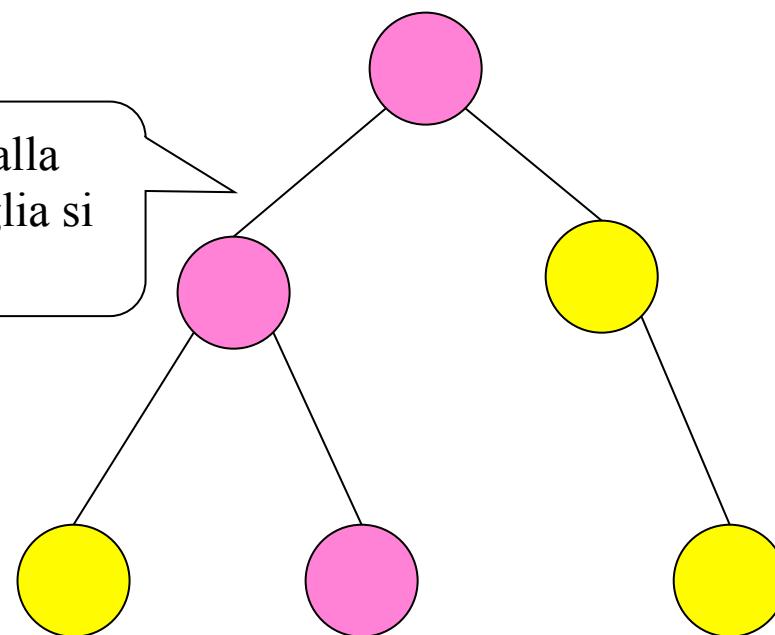
Parentele



Cammini

Cammino: sequenza di archi ciascuno incidente sul vertice di quello successivo

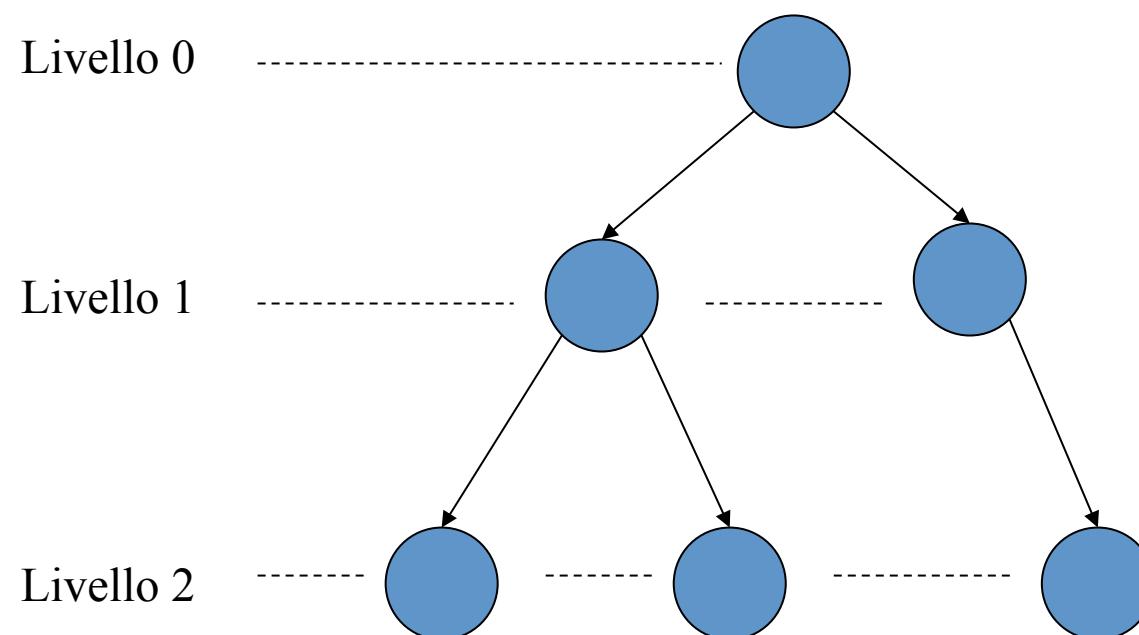
Un cammino dalla radice ad una foglia si dice *ramo*



Livelli

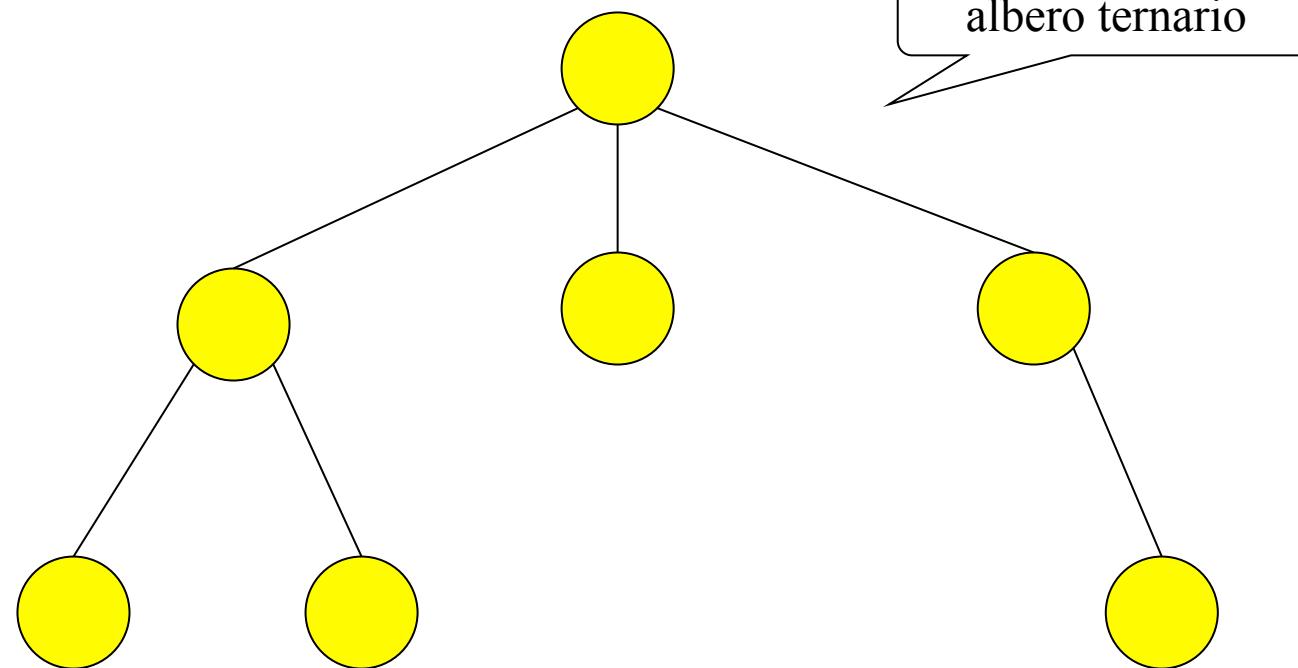
Livello: insieme di vertici equidistanti dalla radice

L'*altezza* è la massima distanza dalla radice di un livello non vuoto



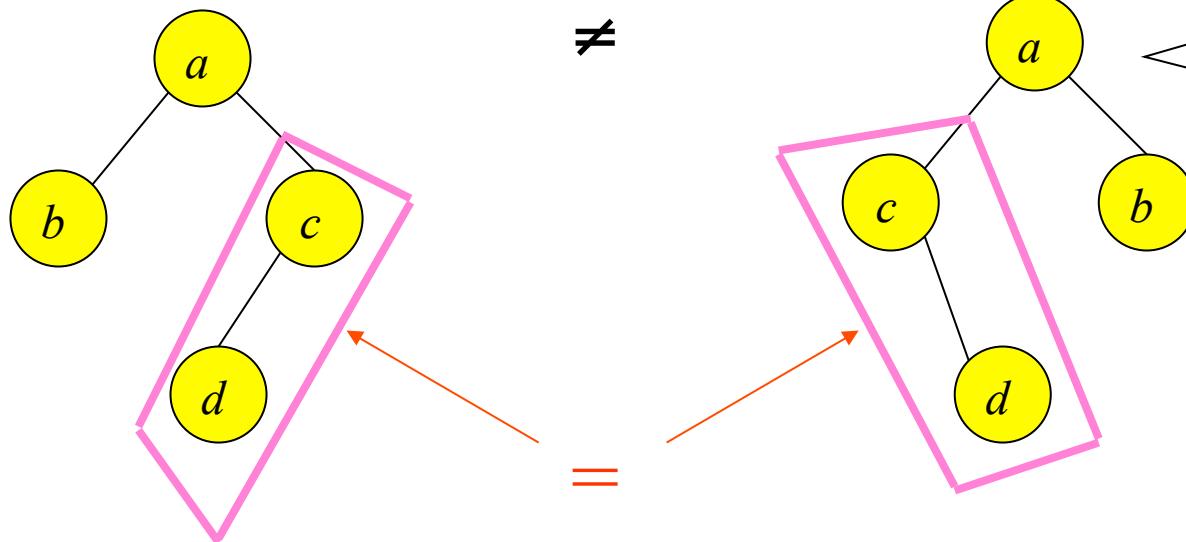
Alberi di grado k (k -ari)

Grado = massimo num. dei figli di qualche nodo



Alberi ordinati

Un albero è *ordinato* quando lo sono (linearmente) i suoi livelli



Come alberi ordinati siamo diversi

Conta solo l'ordine, non sinistra e destra

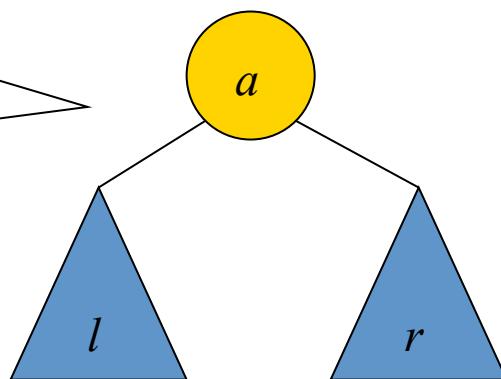
Alberi binari posizionali

L'insieme degli alberi binari etichettati in A , $\text{BT}(A)$, è definito induttivamente:

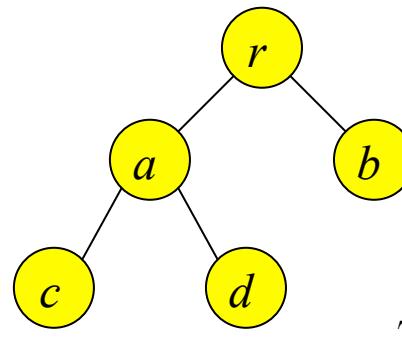
- a) $\emptyset \in \text{BT}(A)$ (albero vuoto)
- b) $a \in A, l \in \text{BT}(A), r \in \text{BT}(A) \Rightarrow$

$$\text{ConsTree}(a, l, r) \in \text{BT}(A)$$

Si introduce la nozione
di **sottoalbero sinistro** e
destro

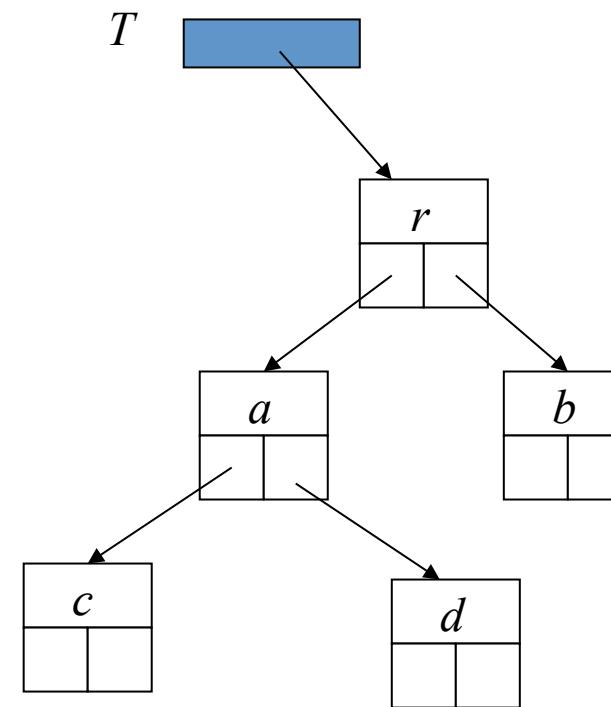


Alberi binari realizzati con puntatori

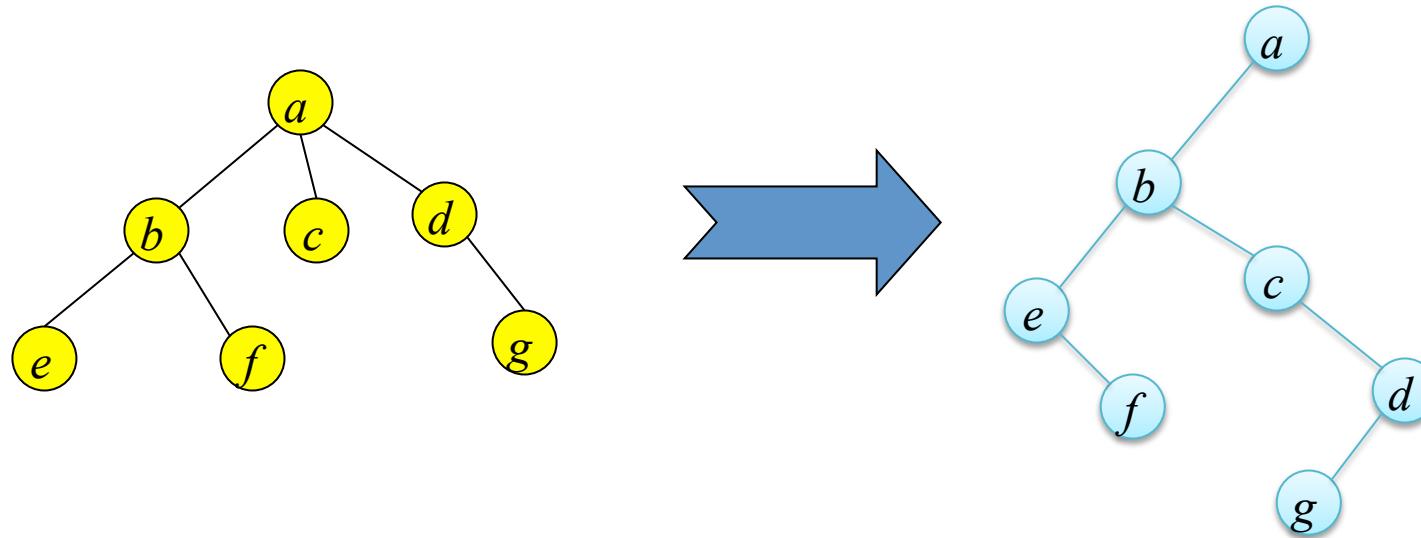


T

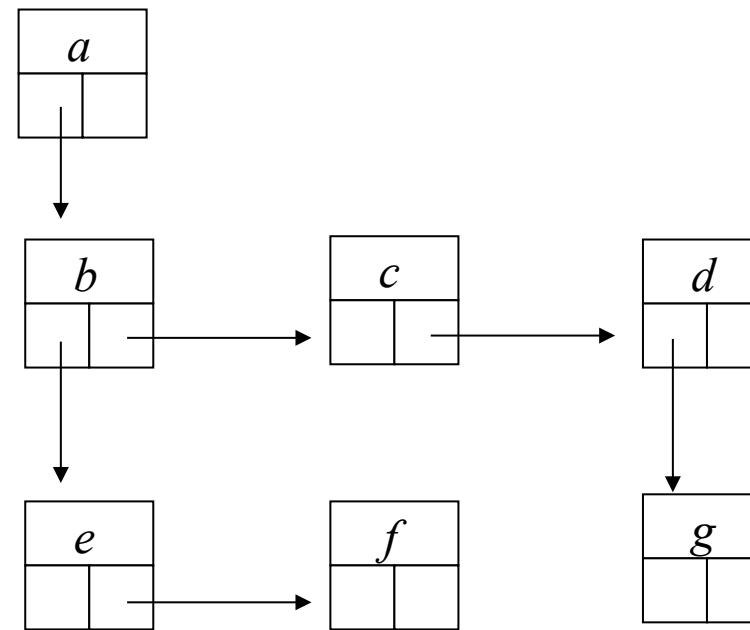
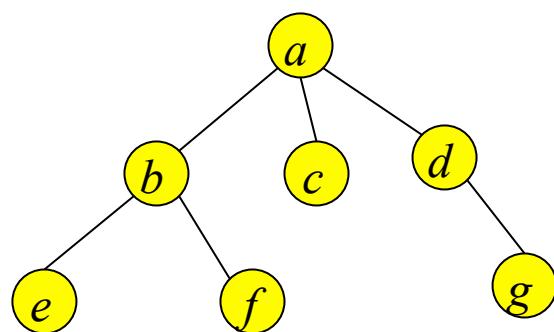
key	
left	right



Codifica binaria di alberi k -ari

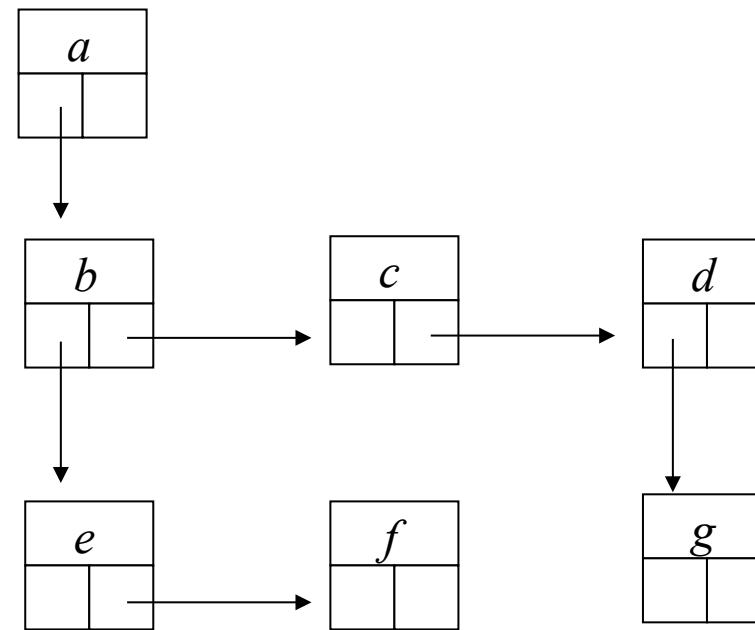
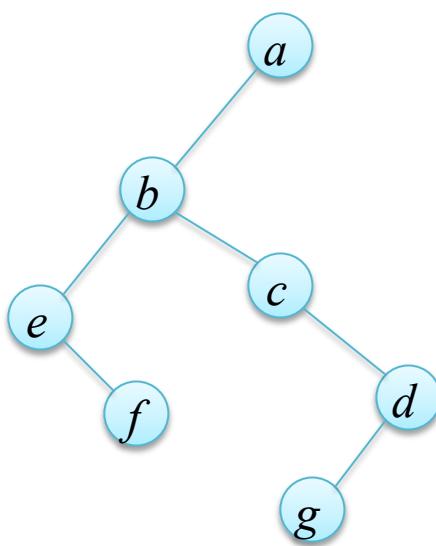


Codifica binaria di alberi k -ari



key	
child	sibling

Codifica binaria di alberi k -ari



key	
child	sibling

Cardinalità



La **cardinalità** di T
è numero dei suoi
nodi

```
2-TREE-CARD(2-Tree  $T$ )
if  $T = \text{nil}$  then
    return 0
else
     $l \leftarrow \text{2-TREE-CARD}(T.\text{left})$ 
     $r \leftarrow \text{2-TREE-CARD}(T.\text{right})$ 
    return  $l + r + 1$ 
end if
```

Nel caso degli alberi
binari ...

Cardinalità



La **cardinalità** di T
è numero dei suoi
nodi

```
 $k\text{-TREE-CARD}(k\text{-Tree } T)$ 
if  $T = \text{nil}$  then
    return 0
else
     $card \leftarrow 1$ 
     $C \leftarrow T.\text{child}$ 
    while  $C \neq \text{nil}$  do
         $card \leftarrow card + k\text{-TREE-CARD}(C)$ 
         $C \leftarrow C.\text{sibling}$ 
    end while
    return  $card$ 
end if
```

... ed in quello degli
alberi k -ari

Altezza

```
2-TREE-HIGHT(2-Tree  $T$ )      ▷ pre:  $T$  non è vuoto
if  $T.left = nil$  and  $T.right = nil$  then
    return 0      ▷  $T$  ha un solo nodo
else
     $hl, hr \leftarrow 0$ 
    if  $T.left \neq nil$  then
         $hl \leftarrow 2\text{-TREE-HIGHT}(T.left)$ 
    end if
    if  $T.right \neq nil$  then
         $hr \leftarrow 2\text{-TREE-HIGHT}(T.right)$ 
    end if
    return  $1 + \max\{hl, hr\}$ 
end if
```

L'altezza è il massimo dei livelli, ossia il massimo delle lunghezze dei rami



Altezza

k -TREE-HIGHT(k -Tree T) \triangleright pre: T non è vuoto

Sapreste calcolare l'altezza
di un albero di grado
arbitrario?

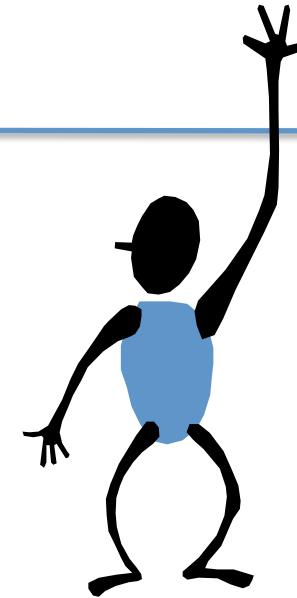


Visite



Per rispondere osserviamo che
hanno tutti la struttura di una
visita!

Qual è la complessità di
questi algoritmi?



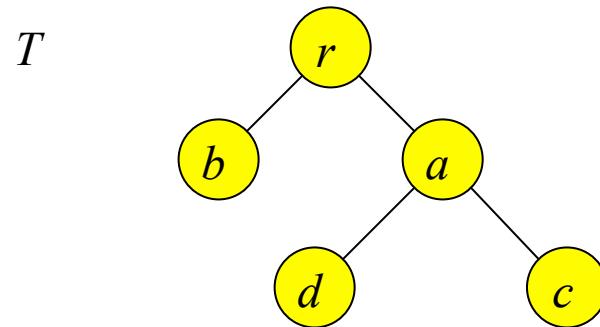
Visite

La *visita* (completa) di un albero consiste in un’ispezione dei nodi dell’albero in cui ciascun nodo sia “visitato” (ispezionato) esattamente una volta.

Visita in profondità (DFS): lungo i rami, dalla radice alle foglie

Visita in ampiezza (BFS): per livelli, da quello della radice in poi.

Visite



DFS (preordine destro) di T : r, a, c, d, b

BFS di T : r, b, a, d, c

DFS (ricorsiva)

TREE-DFS(k -Tree T) \triangleright pre: T non è vuoto

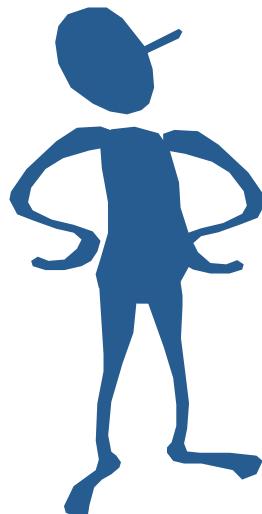
visita la radice di T

for all C figlio della radice di T **do**

 TREE-DFS(C)

end for

o più precisamente



TREE-DFS(k -Tree T)

visita $T.key$

$C \leftarrow T.child$

while $C \neq \text{nil}$ **do**

 TREE-DFS(C)

$C \leftarrow C.sibling$

end while

DFS con l'ausilio di una pila

TREE-DFS-STACK(k -Tree T) \triangleright pre: T non è vuoto

$S \leftarrow$ pila vuota

$Push(S, T)$

while $S \neq$ la pila vuota **do**

$T' \leftarrow Pop(S)$

 visita $T'.key$

for all C figlio di T' **do**

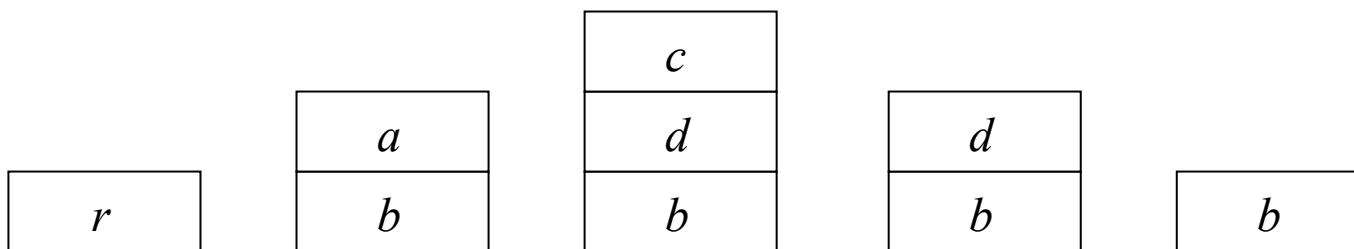
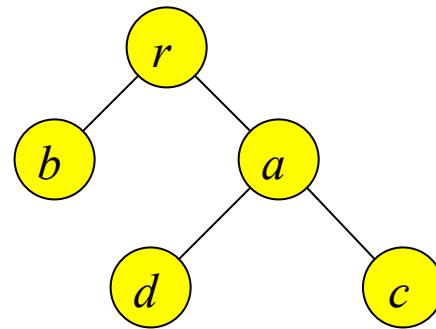
$Push(S, C)$

end for

end while

DFS con l'ausilio di una pila

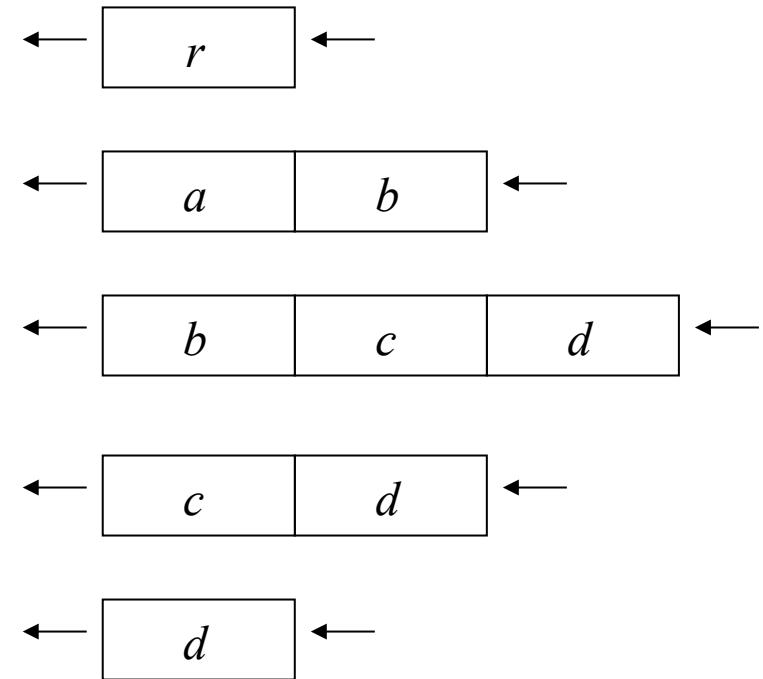
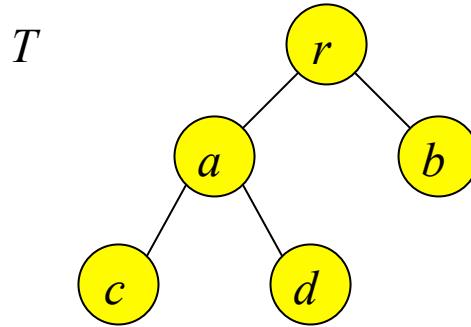
T



Visita in ampiezza (BFS)

```
TREE-BFS( $k$ -Tree  $T$ )       $\triangleright$  pre:  $T$  non è vuoto  
 $Q \leftarrow$  coda vuota  
 $Enqueue(Q, T)$   
while  $Q \neq$  la coda vuota do  
     $T' \leftarrow Dequeue(Q)$   
    visita  $T'.key$   
    for all  $C$  figlio di  $T'$  do  
         $Enqueue(Q, C)$   
    end for  
end while
```

BFS con l'ausilio di una coda



DFS versus BFS

TREE-DFS-STACK(k -Tree T)

```
 $S \leftarrow$  pila vuota  
 $Push(S, T)$   
while  $S \neq$  la pila vuota do  
     $T' \leftarrow Pop(S)$   
    visita  $T'.key$   
    for all  $C$  figlio di  $T'$  do  
         $Push(S, C)$   
    end for  
end while
```

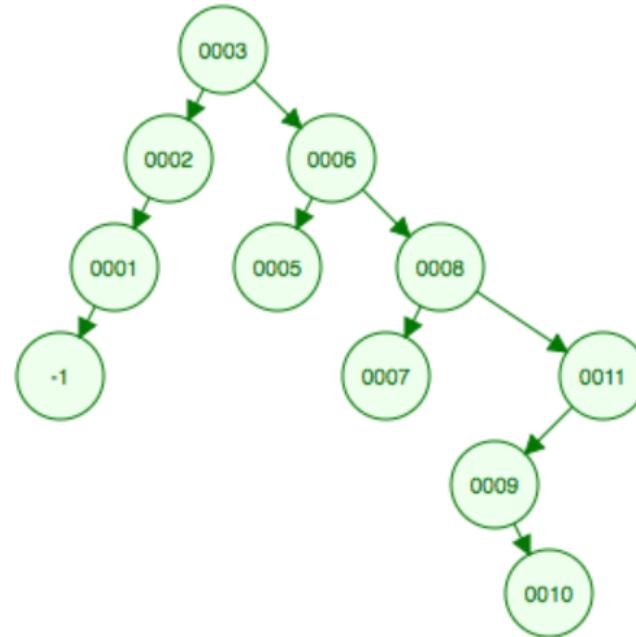
TREE-BFS(k -Tree T)

```
 $Q \leftarrow$  coda vuota  
 $Enqueue(Q, T)$   
while  $Q \neq$  la coda vuota do  
     $T' \leftarrow Dequeue(Q)$   
    visita  $T'.key$   
    for all  $C$  figlio di  $T'$  do  
         $Enqueue(Q, C)$   
    end for  
end while
```

Complessità delle visite

- La dimensione n di un albero è la sua cardinalità
- Per limitare il tempo di una visita possiamo contare quante operazioni Push/Pop ovvero Enqueue/Dequeue avvengono in una DFS o BFS
- Ogni nodo dell'albero viene inserito ed estratto esattamente una volta
- Dunque DFS e BFS hanno costo $O(2n) = O(n)$

Alberi binari di ricerca



Algoritmi e strutture dati

Lezione 12, a.a. 2016-17

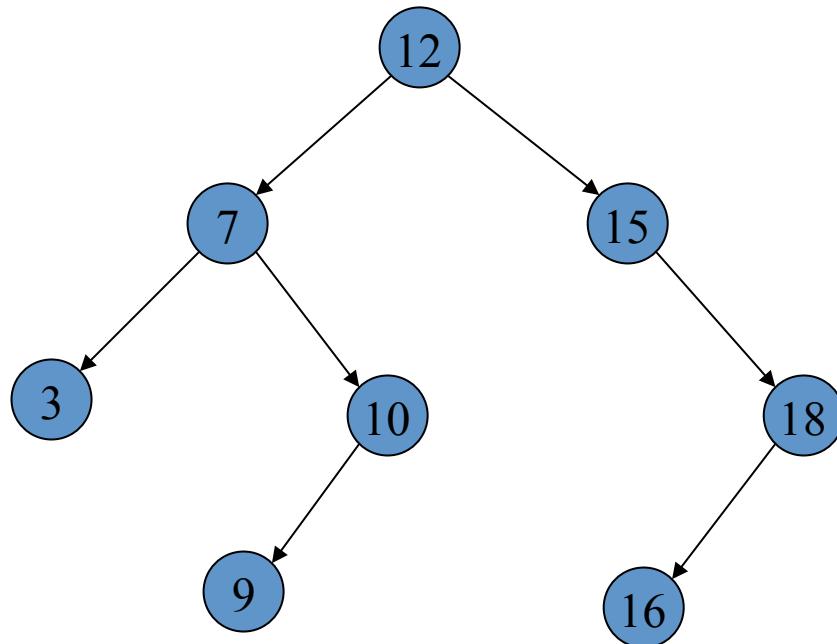
Ugo de'Liguoro, Andras Horvath

Alberi binari di ricerca

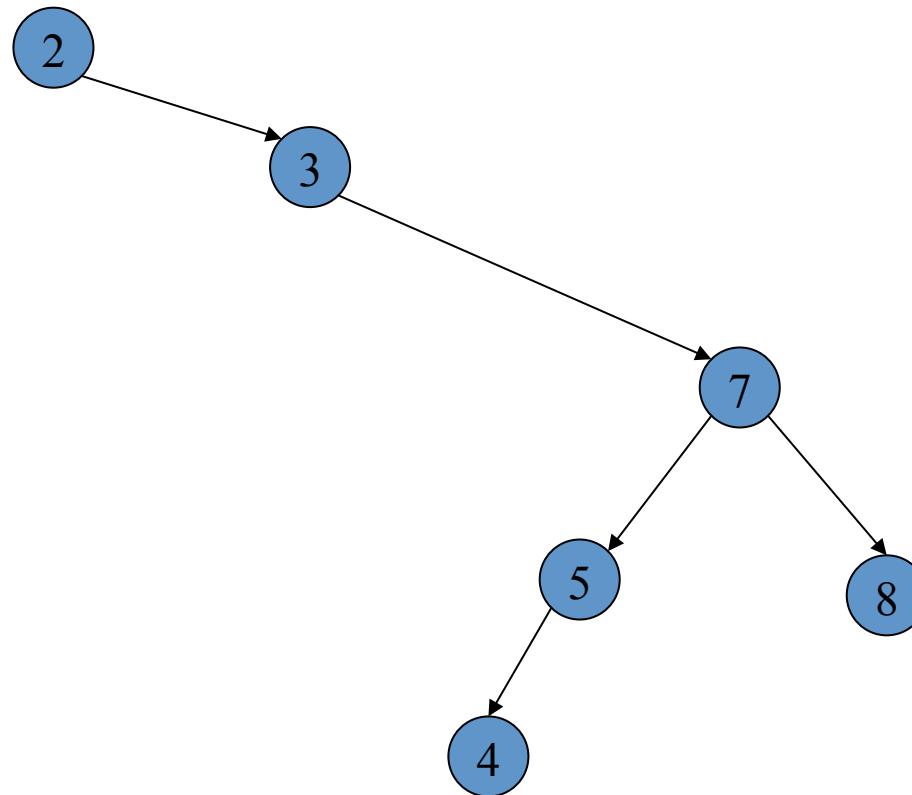
Supposto che l'insieme delle etichette (chiavi) sia ordinato, un albero binario T è *di ricerca* se.

- i. $T = \emptyset$ oppure
- ii. $T = \text{ConsTree}(a, L, R)$:
 - a. $\forall x \in \text{Chiavi}(L). x < a$
 - b. $\forall y \in \text{Chiavi}(R). y > a$
 - c. L, R alberi bin. di ricerca

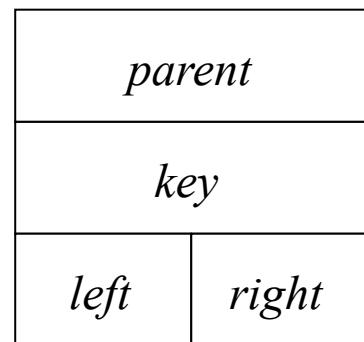
Esempio di albero di ricerca



Esempio di albero di ricerca



Realizzazione con puntatori

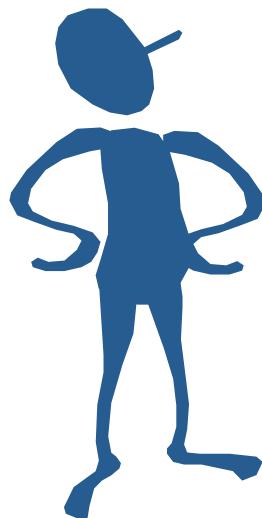


Ricerca

RIC-SEARCH(x, T)

- ▷ pre: x chiave, T binario di ricerca
- ▷ post: il nodo $S \in T$ con $S.key = x$ se esiste, nil altrimenti

```
if  $T = nil$  then return  $nil$ 
else
  if  $x = T.key$  then return  $T$ 
  else
    if  $x < T.key$  then return RIC-SEARCH( $x, T.left$ )
    else      ▷  $x > T.key$ 
      return RIC-SEARCH( $x, T.right$ )
    end if
  end if
end if
```



Complessità $O(h)$ dove
 h = altezza di T

Ricerca

IT-SEARCH(x, T)

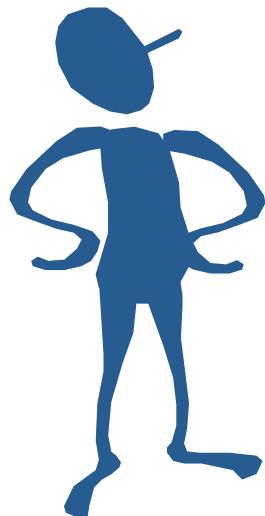
- ▷ pre: x chiave, T binario di ricerca
- ▷ post: il nodo $S \in T$ con $S.key = x$ se esiste, nil altrimenti

while $T \neq nil$ **and** $x \neq T.key$ **do**

- if** $x < T.key$ **then**
- $T \leftarrow T.left$
- else**
- $T \leftarrow T.right$
- end if**

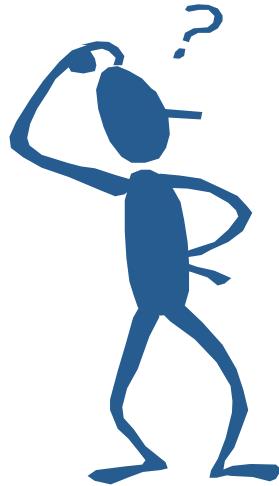
end while

return T



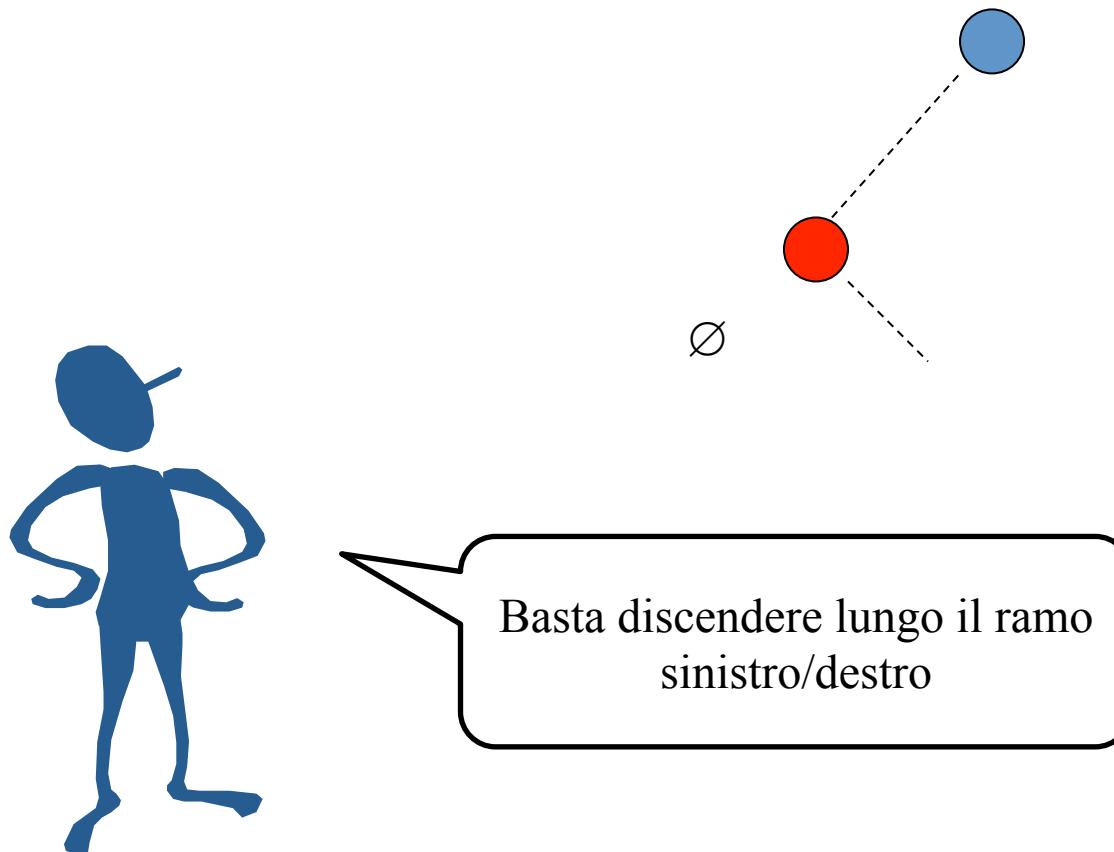
Complessità $O(h)$ dove
 h = altezza di T

Minimo e massimo



Come torvare il nodo con
chiave minima/massima in
 $T \neq \emptyset$?

Minimo e massimo



Minimo e massimo

TREE-MIN(T)

▷ pre: T binario di ricerca non vuoto

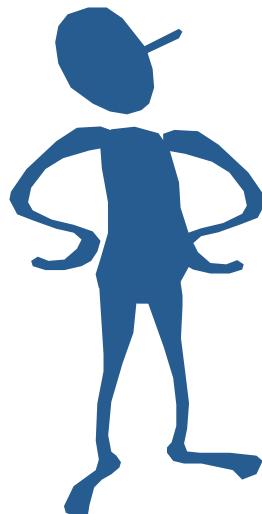
▷ post: il nodo $S \in T$ con $S.key$ minimo

while $T.left \neq nil$ **do**

$T \leftarrow T.left$

end while

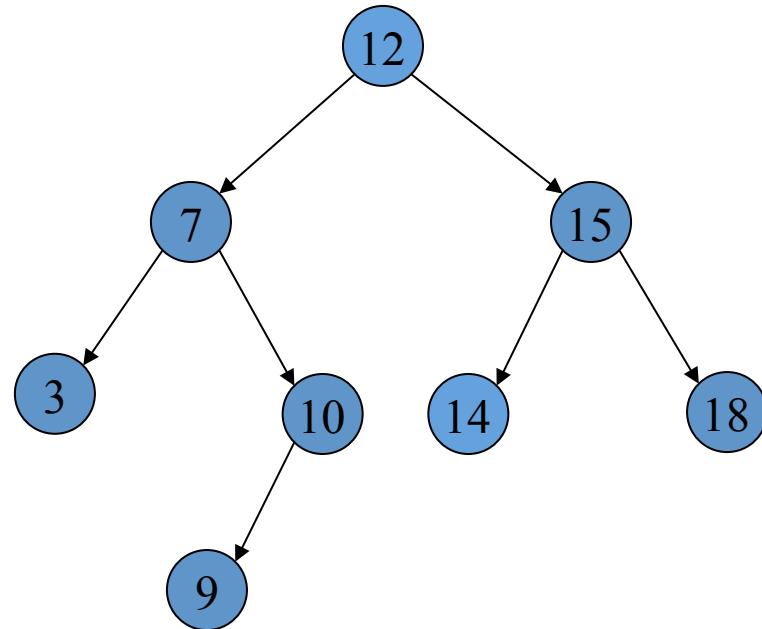
return T



Basta discendere lungo il ramo
sinistro/destro

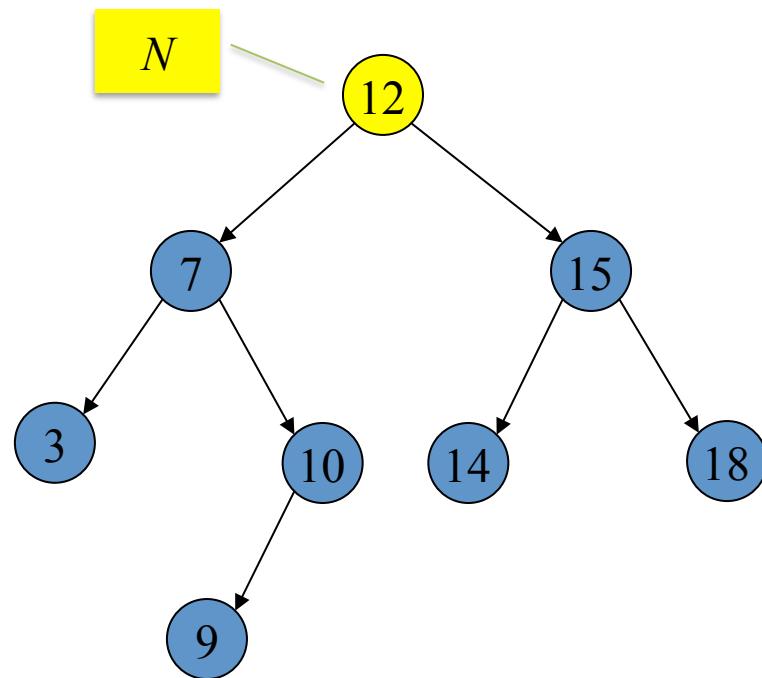
Successore

Il successore di un nodo N in un albero di ricerca T è il nodo con etichetta minima tra quelle maggiori di $N.key$



Successore

Il successore di un nodo N in un albero di ricerca T è il nodo con etichetta minima tra quelle maggiori di $N.key$

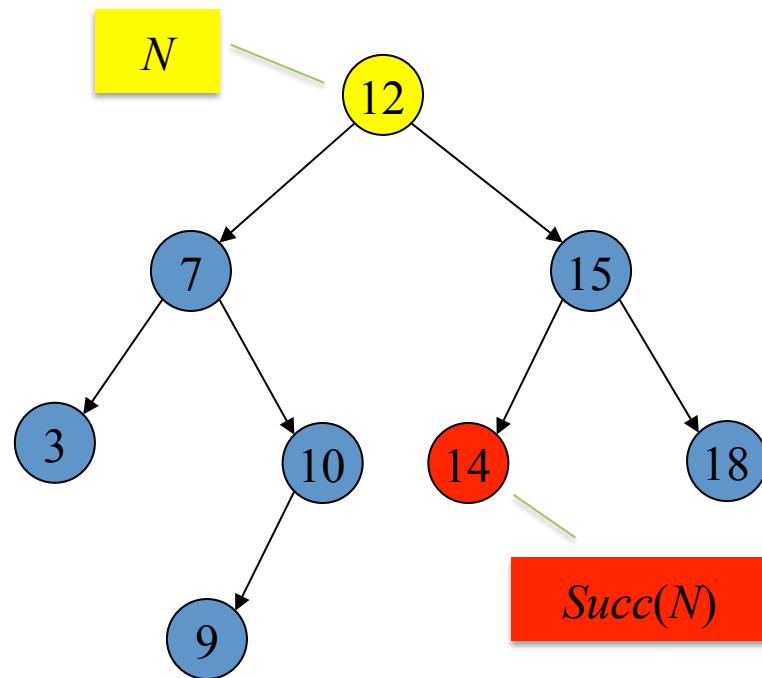


Il nodo N ha
un discendente
destro



Successore

Il successore di un nodo N in un albero di ricerca T è il nodo con etichetta minima tra quelle maggiori di $N.key$

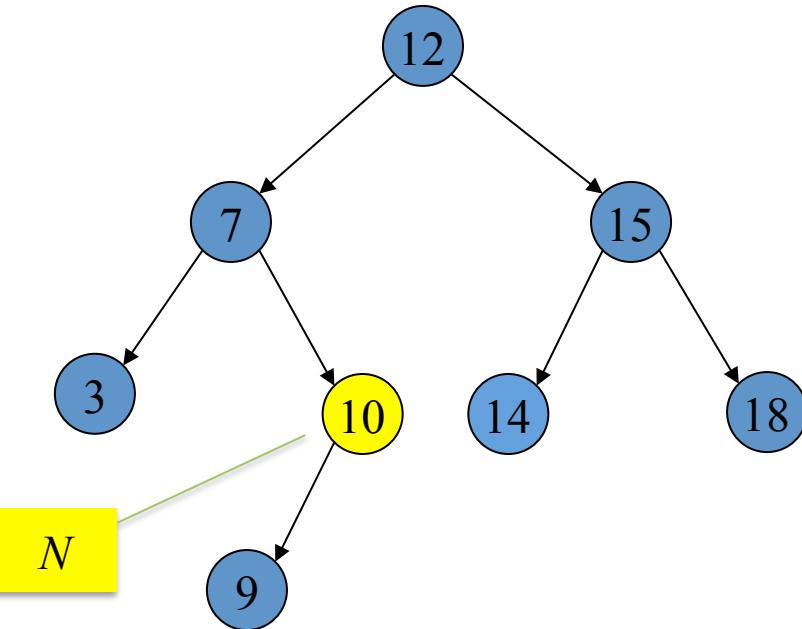


Il nodo N ha
un discendente
destro



Successore

Il successore di un nodo N in un albero di ricerca T è il nodo con etichetta minima tra quelle maggiori di $N.key$

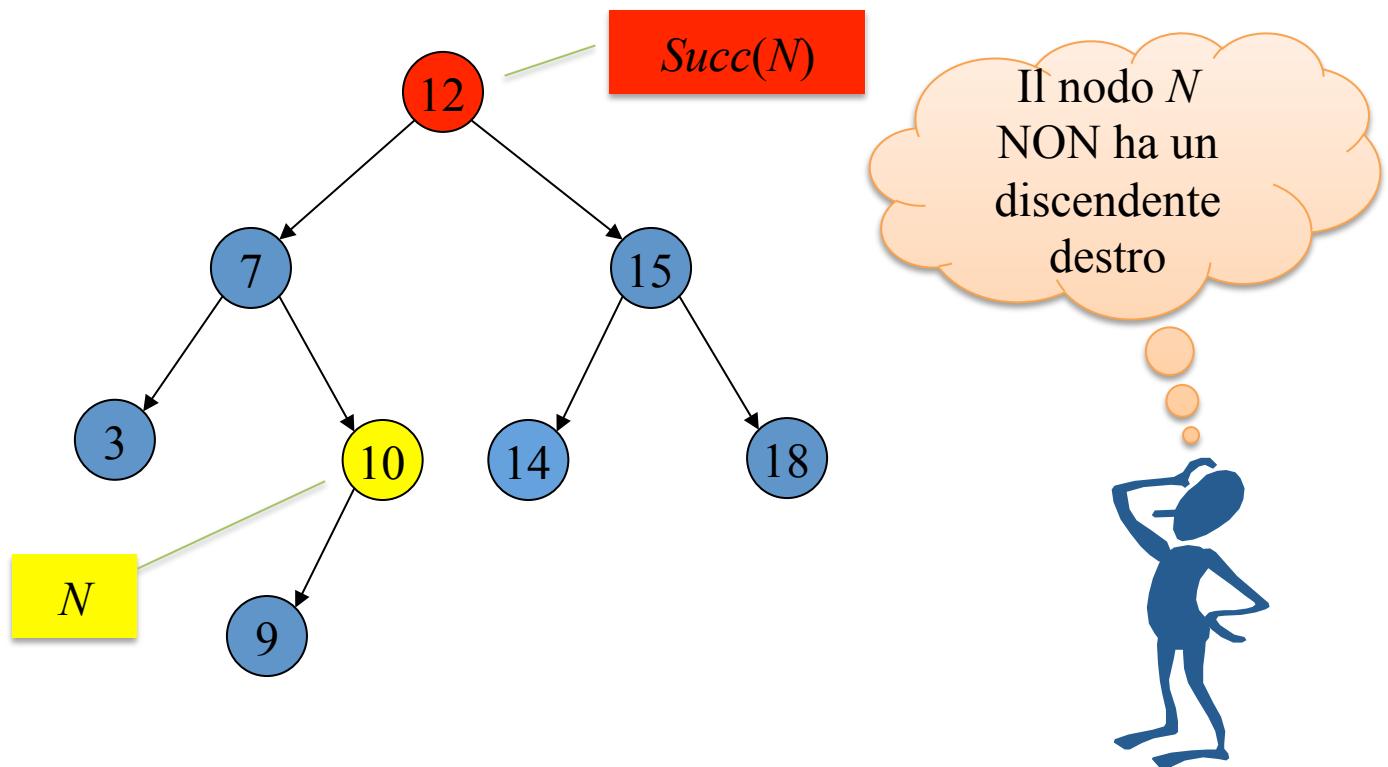


Il nodo N
NON ha un
discendente
destro



Successore

Il successore di un nodo N in un albero di ricerca T è il nodo con etichetta minima tra quelle maggiori di $N.key$



Successore

TREE-SUCC(N)

▷ pre: N nodo di un albero bin. di ricerca

▷ post: il successore di N se esiste, nil altr.

if $N.right \neq nil$ **then**

return TREE-MIN($N.right$)

else ▷ il successore è l'avo sinistro più prossimo

$P \leftarrow N.parent$

while $P \neq nil$ **and** $N = P.right$ **do**

$N \leftarrow P$

$P \leftarrow N.parent$

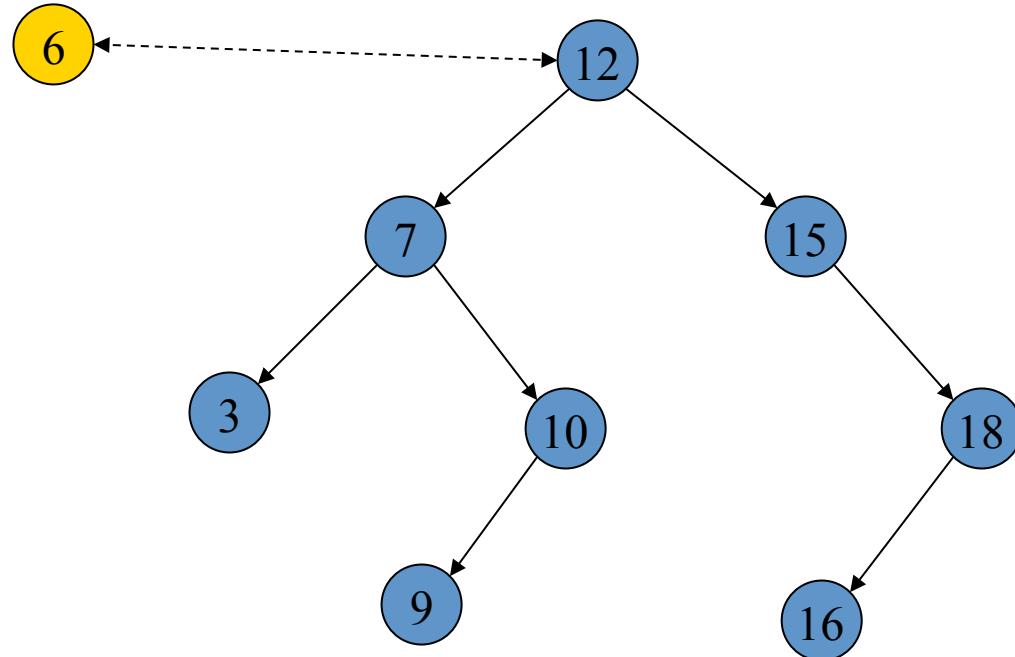
end while

return P

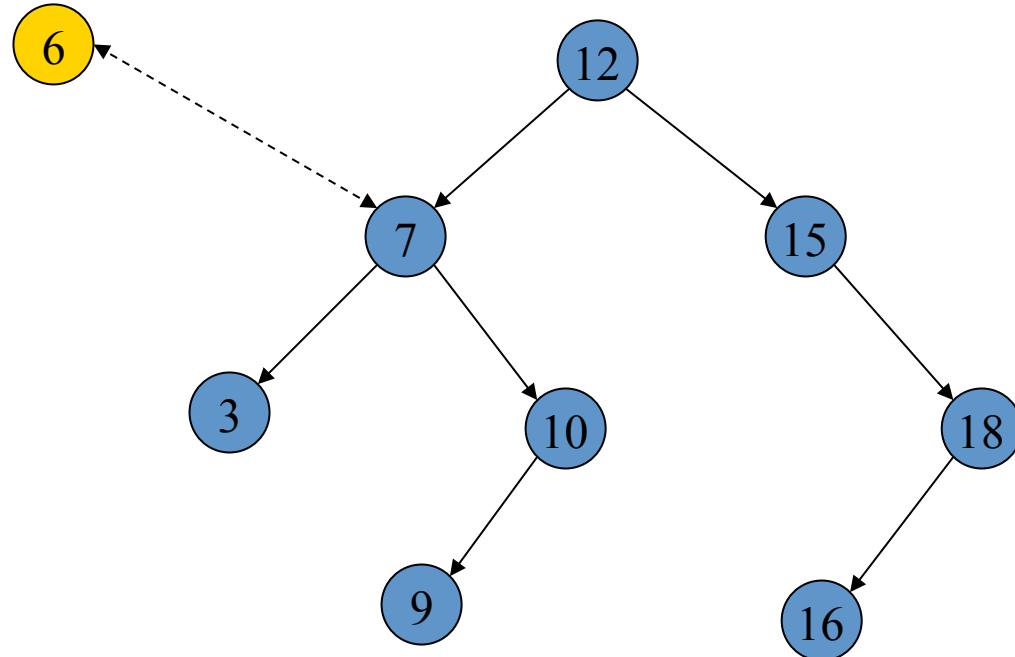
end if

Chiamando “avo sinistro” di
 N un avo A t.c. N sia
nell’albero radicato in $A.left$

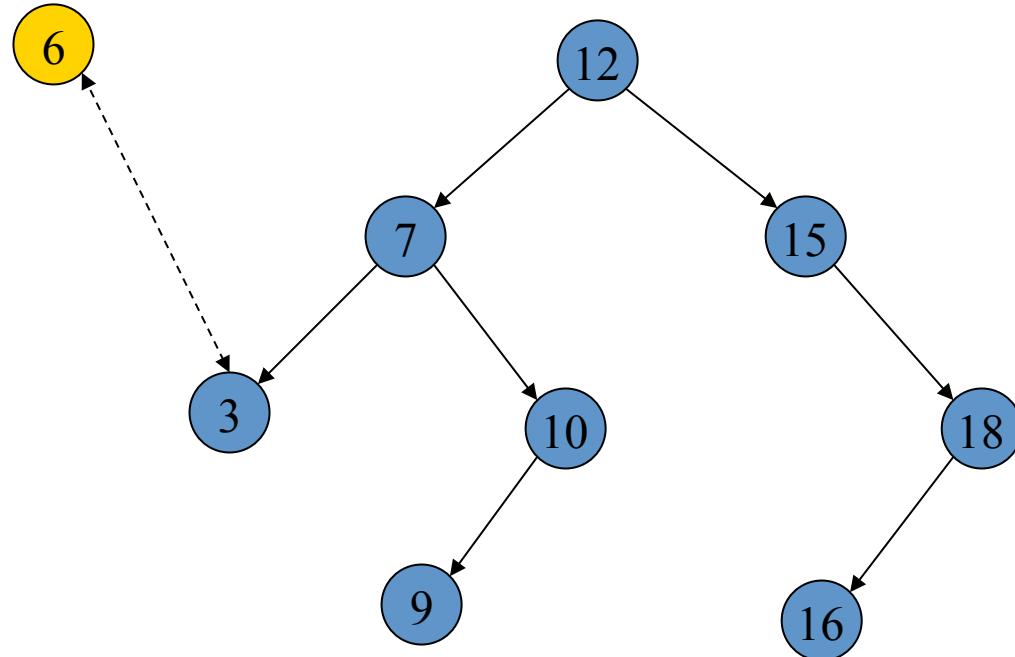
Esempio di inserimento



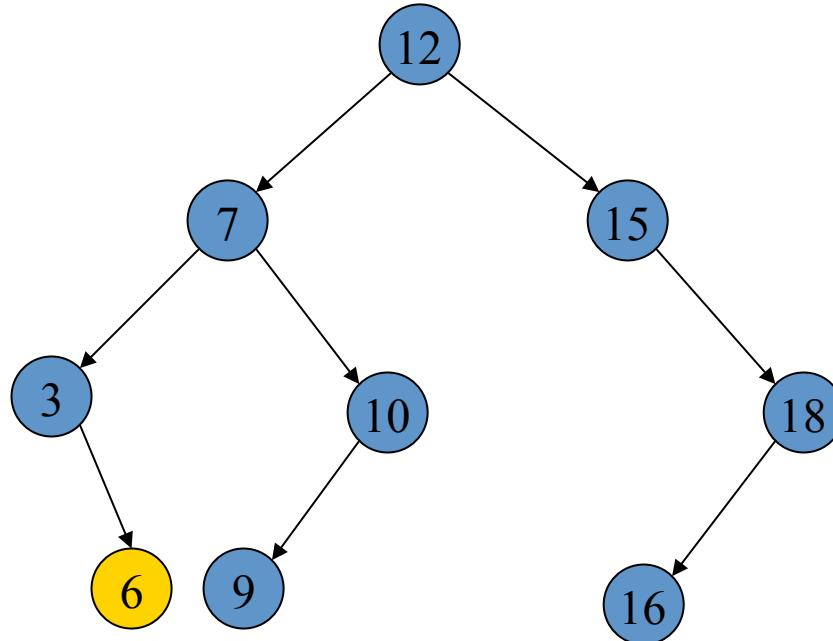
Esempio di inserimento



Esempio di inserimento



Esempio di inserimento



Gli inserimenti in un albero di ricerca avvengono sempre al livello delle foglie.

Inserimento

TREE-INSERT(N, T)

▷ pre: N nuovo nodo, T albero di ric.

▷ post: N è un nodo di T , albero di ric.

$P \leftarrow nil$

$S \leftarrow T$

while $S \neq nil$ **do** ▷ inv: se $P \neq nil$ allora è il padre di S

$P \leftarrow S$

if $N.key = S.key$ **then return**

else

if $N.key < S.key$ **then** $S \leftarrow S.left$

else $S \leftarrow S.right$

end if

end if

end while

$N.parent \leftarrow P$

if $P = nil$ **then** $T \leftarrow N$

else

if $N.key < P.key$ **then** $P.left \leftarrow N$

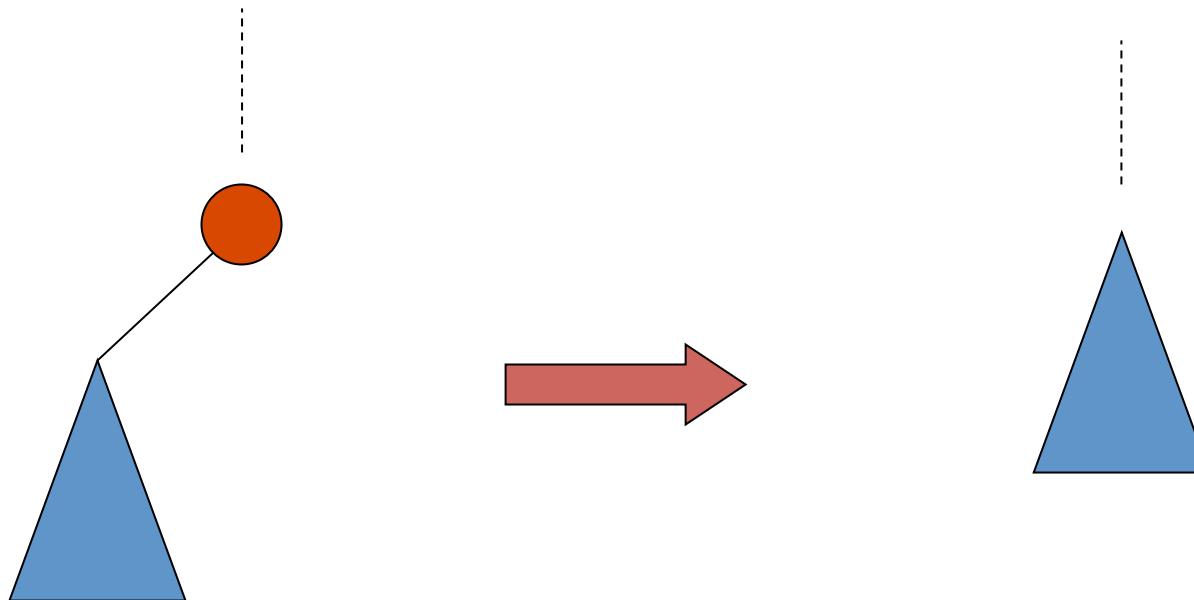
else $P.right \leftarrow N$

end if

end if

Cancellazione

Caso: nodo da eliminare non ha discendente destro



Cancellazione

1-DELETE(Z, T)

▷ pre: Z nodo di T con un solo discendente

▷ post: Z non è più un nodo di T

if $Z = T$ **then**

if $Z.left \neq nil$ **then**

$T \leftarrow Z.left$

else

$T \leftarrow Z.right$

end if

else

if $Z.left \neq nil$ **then**

$Z.left.parent \leftarrow Z.parent$

else

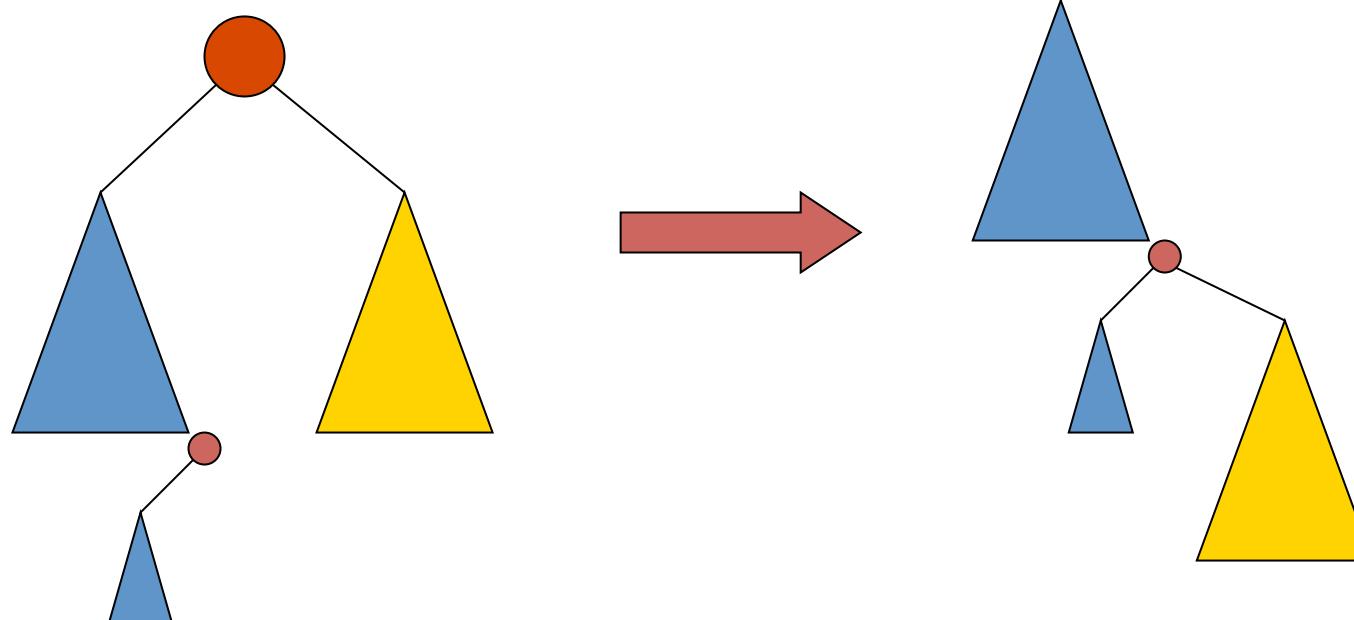
$Z.right.parent \leftarrow Z.parent$

end if

end if

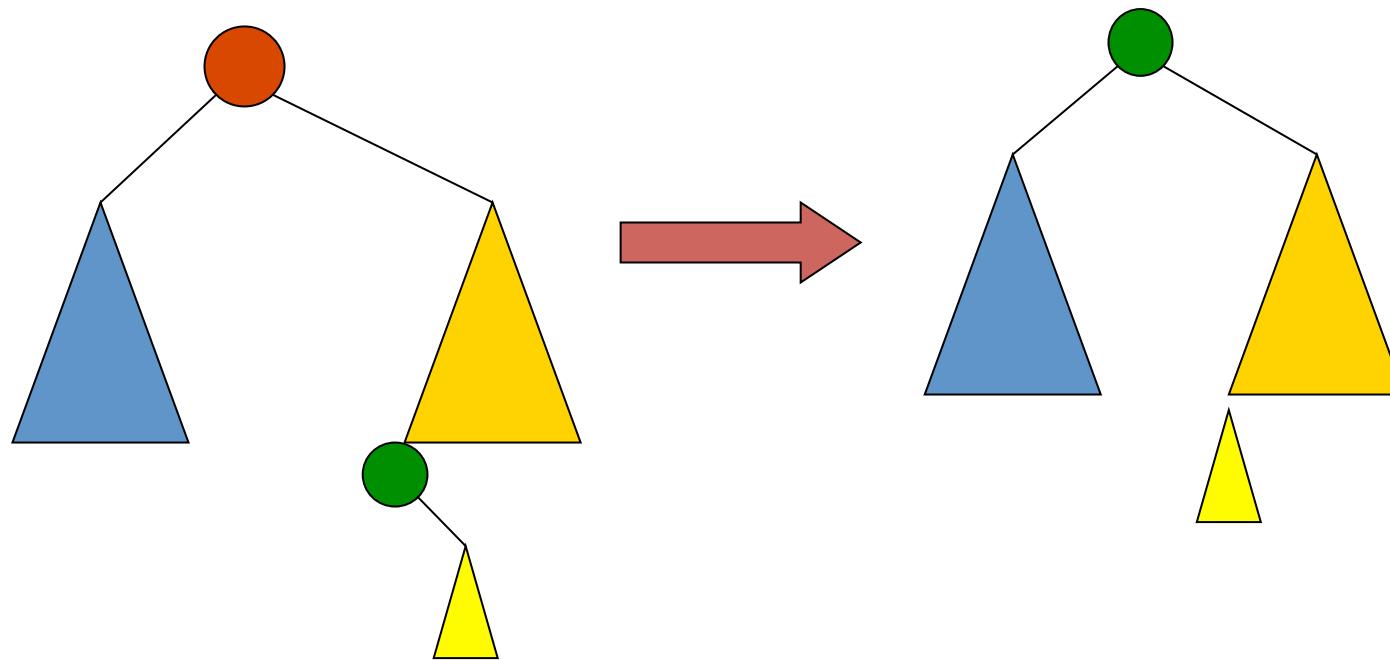
Cancellazione: fusione

Caso: nodo da eliminare ha entrambi i discendenti



Cancellazione: copia

Caso: nodo da eliminare ha entrambi i discendenti



Cancellazione

```
TREE-DELETE( $Z, T$ )
     $\triangleright$  pre:  $Z$  nodo di  $T$ 
     $\triangleright$  post:  $Z$  non è più un nodo di  $T$ 
if  $Z.left = nil$  and  $Z.right = nil$  then       $\triangleright Z$  è una foglia
    if  $Z.parent.left = Z$  then       $\triangleright Z$  è figlio sin.
         $Z.parent.left \leftarrow nil$ 
    else       $\triangleright Z$  è figlio des.
         $Z.parent.right \leftarrow nil$ 
    end if
else
    if  $Z.left = nil$  or  $Z.right = nil$  then
        1-DELETE( $Z, T$ )
    else       $\triangleright Z$  ha due figli e dunque un successore in  $T.right$ 
         $Y \leftarrow$  TREE-SUCC( $Z$ )
         $Z.key \leftarrow Y.key$ 
        DELETE( $Y, T$ )
    end if
end if
```

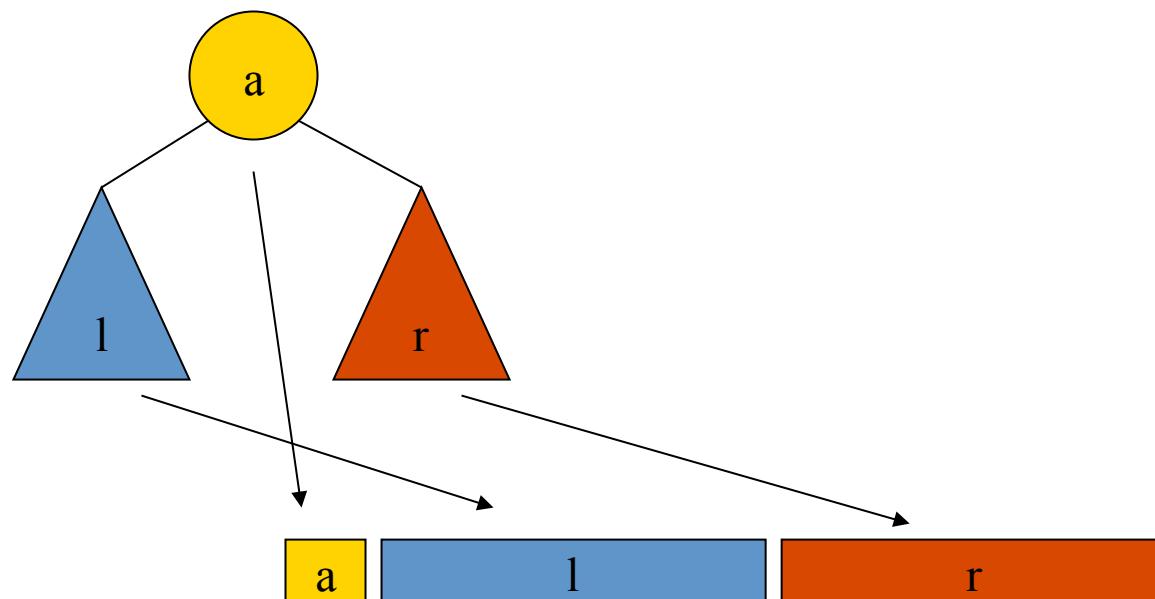
Salvataggio di un albero di ricerca



Come è possibile riprodurre una copia di un albero di ricerca avendolo salvato su un file (o altra struttura lineare) ?

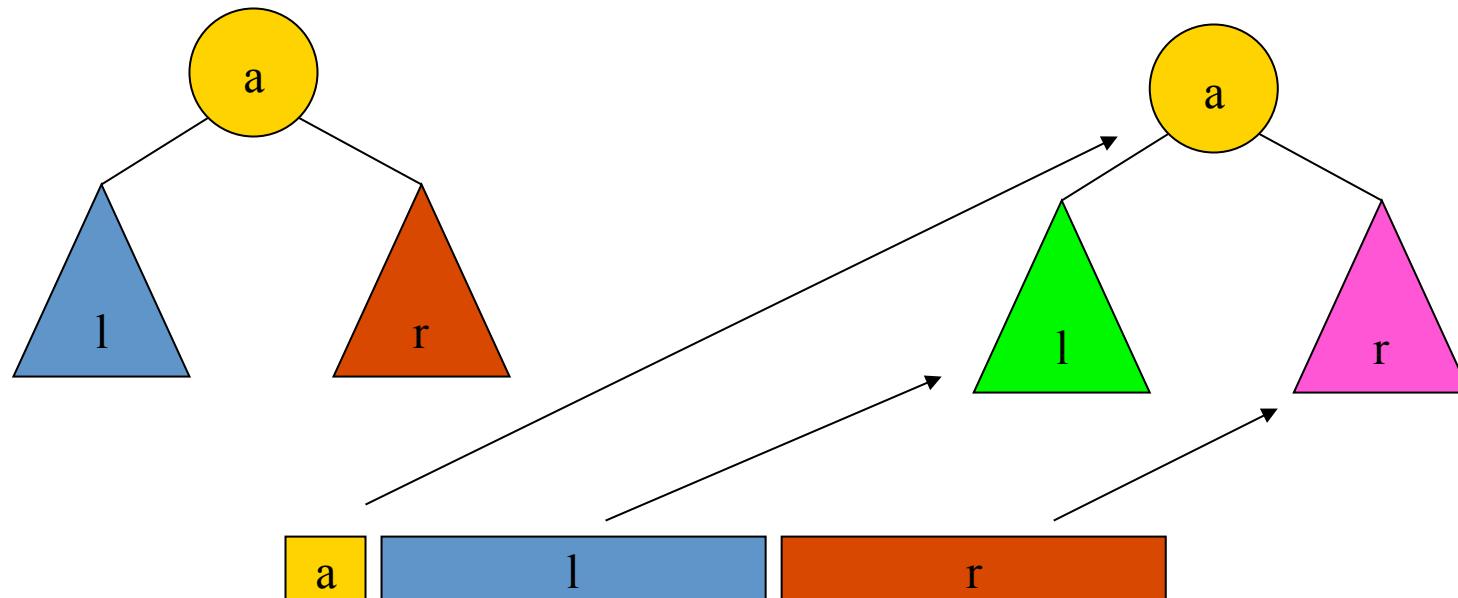
Visita di alberi di ricerca (2)

Prop. Sia T un albero di ricerca ed L la lista prodotta dalla visita in preordine di T : se T' è costruito per inserimenti successivi degli elementi di L (da sinistra a destra) allora T e T' sono isomorfi.



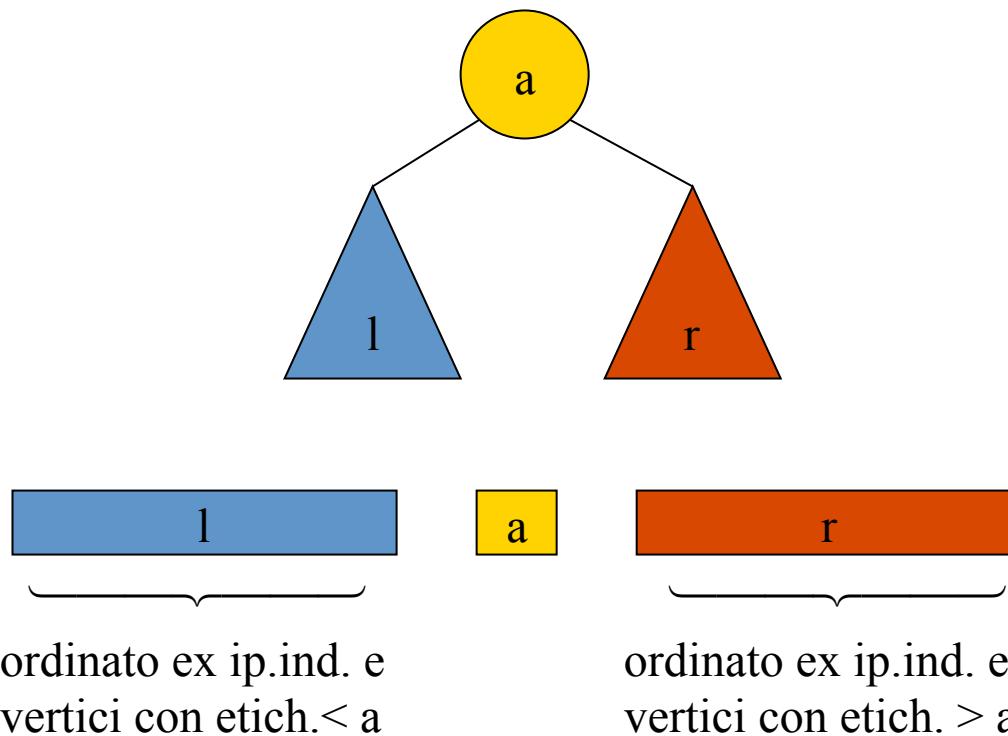
Visita di alberi di ricerca (2)

Prop. Sia T un albero di ricerca ed L la lista prodotta dalla visita in preordine di T : se T' è costruito per inserimenti successivi degli elementi di L (da sinistra a destra) allora T e T' sono isomorfi.



Visita di alberi di ricerca

Prop. Se T è un albero di ricerca ed L la lista delle etichette di T visitato in *inordine*, allora L è ordinata in modo crescente.



La lista ordinata dei vertici

La soluzione ovvia è $O(n^2)$:

TREE-INORDER(T)

- ▷ pre: T binario di ricerca
- ▷ post: ritorna la lista ordinata delle chiavi in T

if $T = \text{nil}$ **then**
 return nil
else $L \leftarrow \text{TREE-INORDER}(T.\text{left})$, $R \leftarrow \text{TREE-INORDER}(T.\text{right})$
 return APPEND(L , CONS($T.\text{key}$, R))
end if



La complessità quadratica
deriva dall'uso di *Append* nel
caso in cui l'albero sia
degenero sinistro.

APPEND(L, M)

if $L = \text{nil}$ **then**
 return M
else
 $L.\text{next} \leftarrow \text{APPEND}(\text{TAIL}(L), M)$
 return L
end if

La lista ordinata dei vertici

Esiste una soluzione (banalmente ottima) $O(n)$:

TREE-INORDER(T, L)

- ▷ pre: T binario di ricerca
- ▷ post: ritorna la lista ordinata delle chiavi in T concatenata con L

if $T = \text{nil}$ **then**
 return L
else $L \leftarrow \text{TREE-INORDER}(T.\text{right}, L)$
 return $\text{TREE-INORDER}(T.\text{left}, \text{CONS}(T.\text{key}, L))$
end if



Alberi Rosso-Neri



Algoritmi e strutture dati

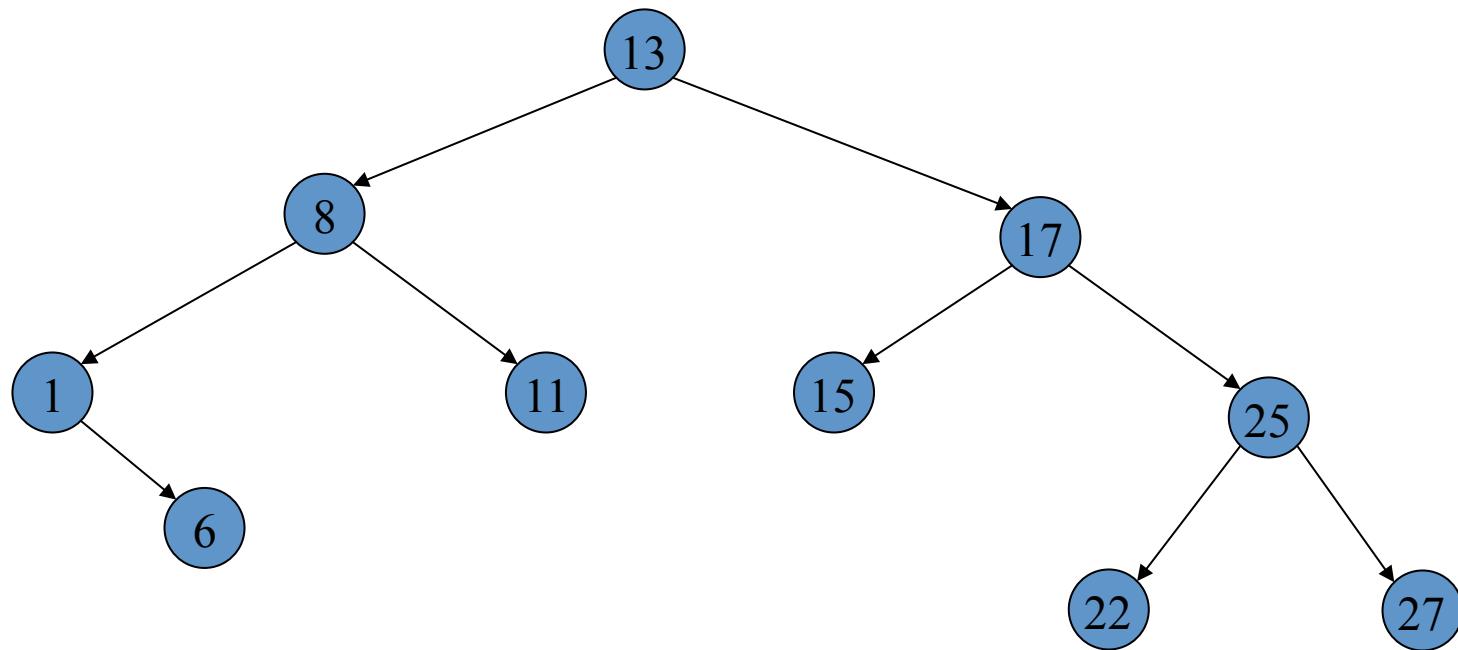
Lezione 13, a.a. 2016-17

Ugo de'Liguoro, Andras Horvath

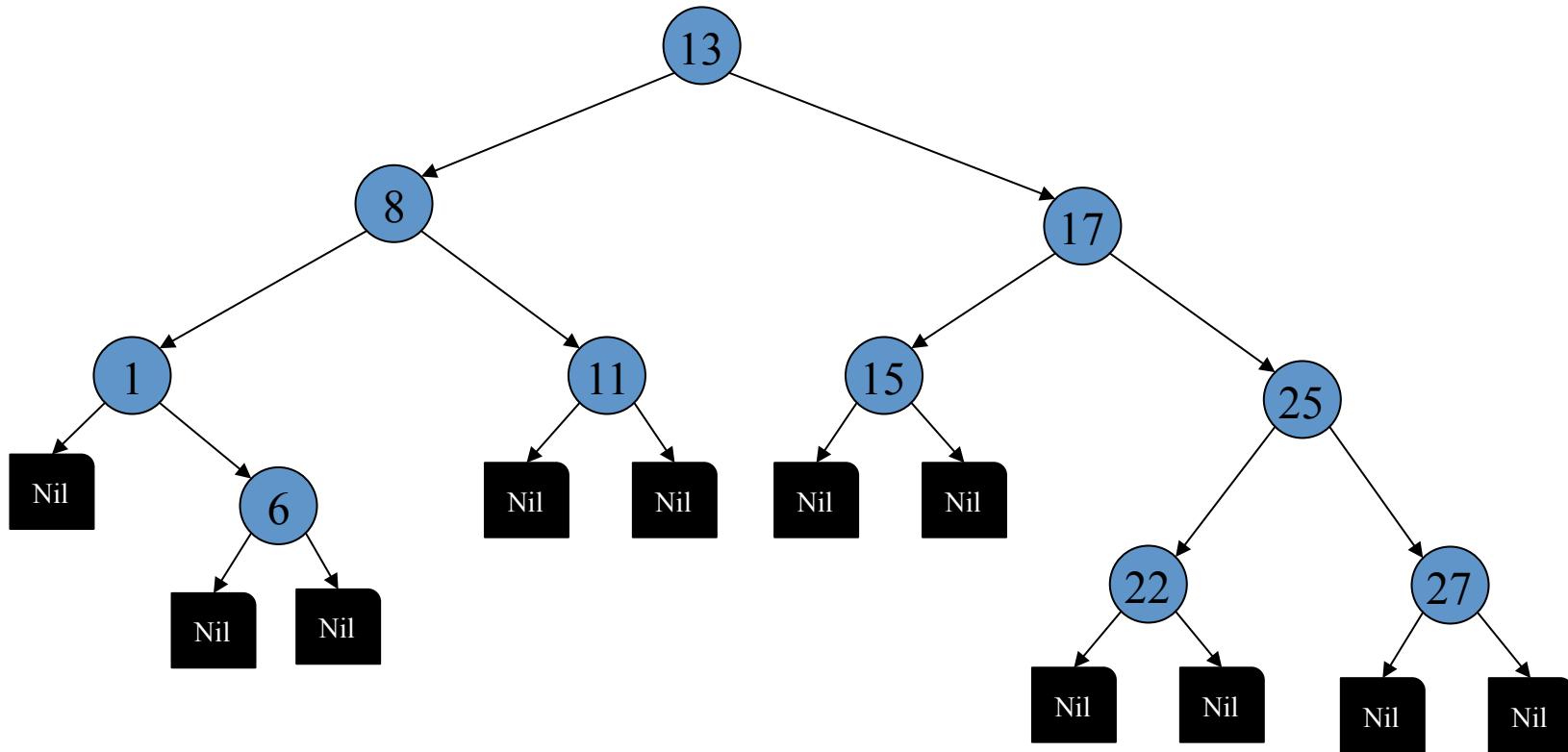
Alberi bilanciati

- Alberi di ricerca
- R-NAlberi

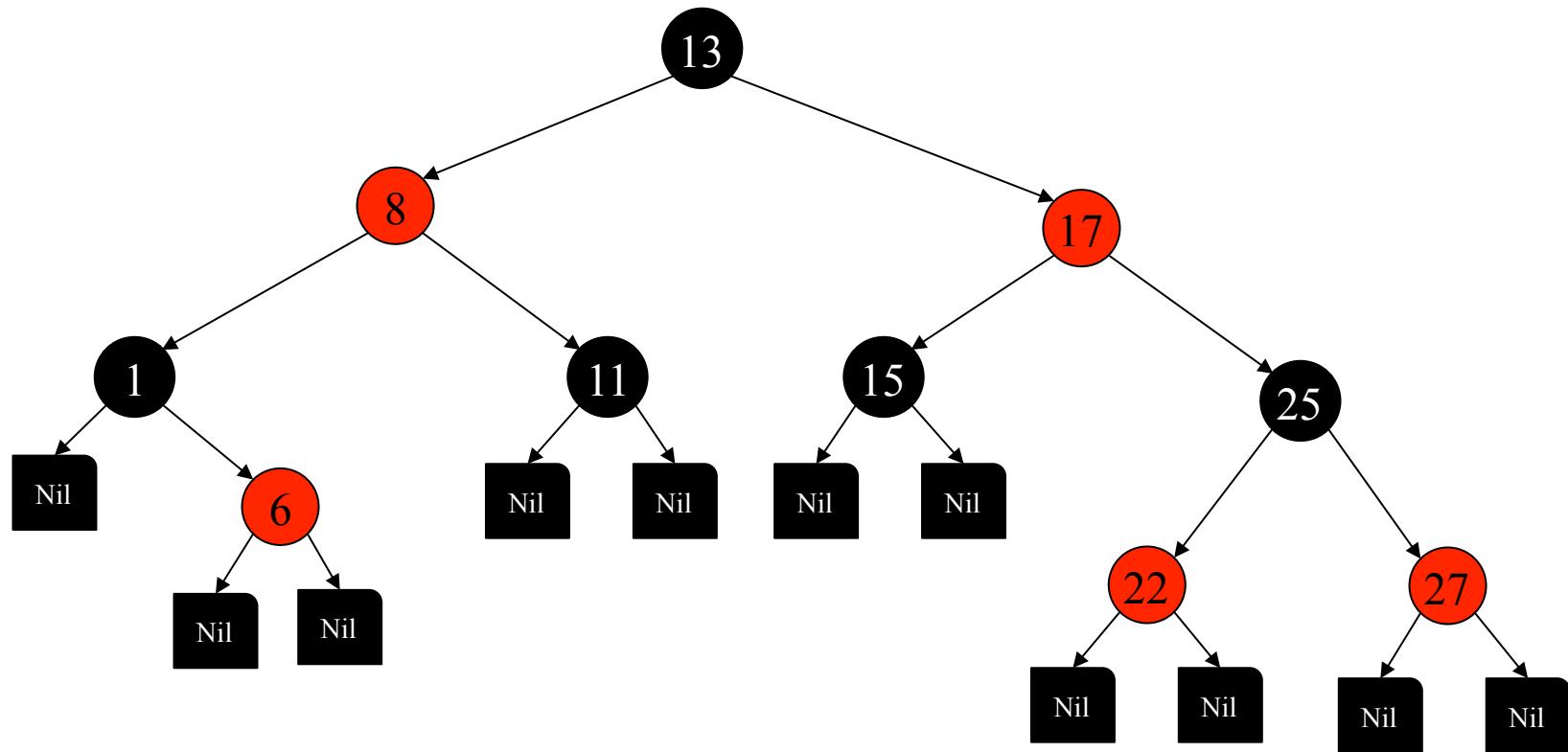
Alberi di ricerca



Alberi di ricerca aumentati



Alberi rosso-neri (R-N)



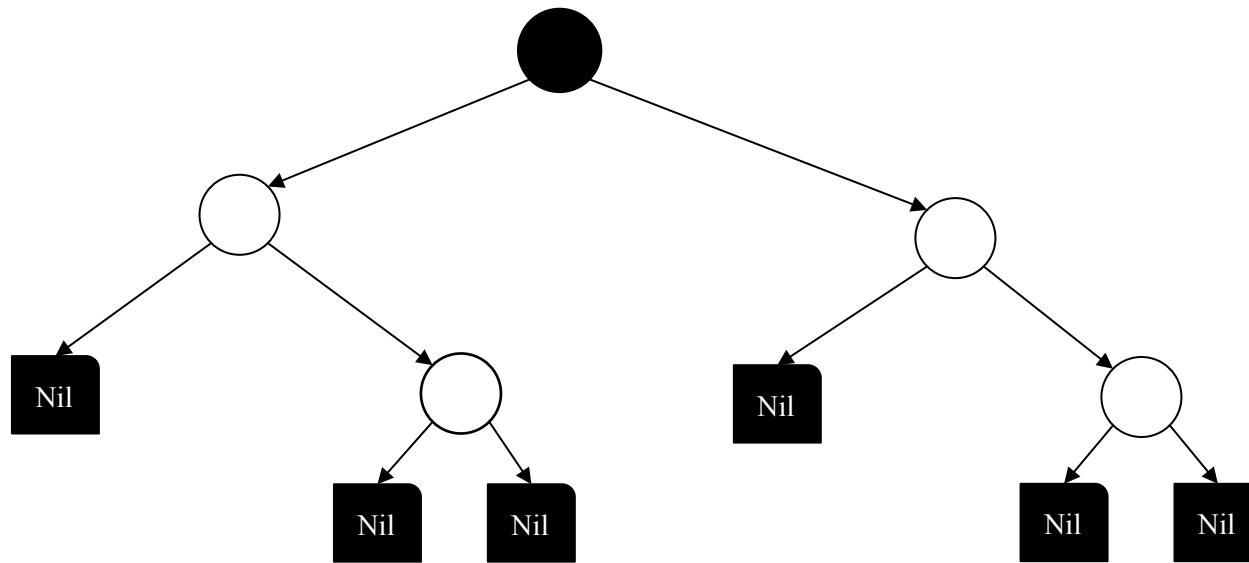
Alberi rosso-neri

Def. Un *albero rosso-nero (R-N)* è un albero binario di ricerca aumentato, i cui vertici sono colorati di rosso o nero in modo che:

- (Regola del *nero*) la radice e tutte le foglie sono nere
- (Regola del *rosso*) se un nodo è rosso tutti i suoi figli sono neri
- (Regola del *cammino*) per ogni nodo x tutti i cammini da x ad una foglia hanno lo stesso numero di nodi neri

$$\begin{aligned} \text{bh}(x) &= \text{altezza nera di } x \\ &= \text{il numero dei nodi neri su un cammino da } x \text{ ad una foglia} \\ &\quad (x \text{ escluso}) \end{aligned}$$

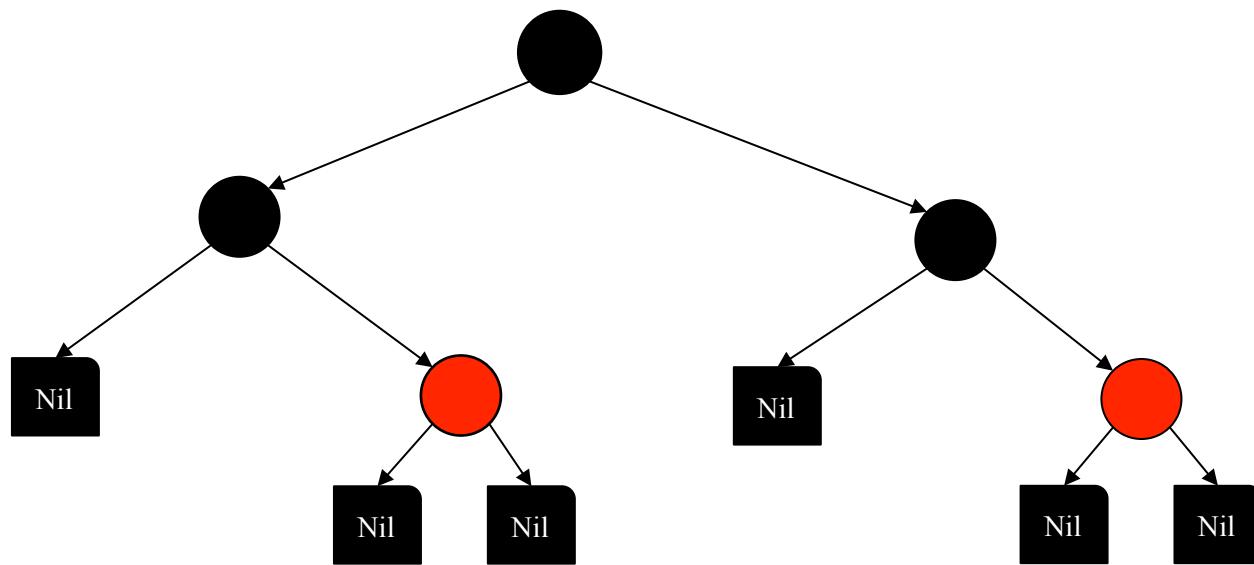
Esempio 1



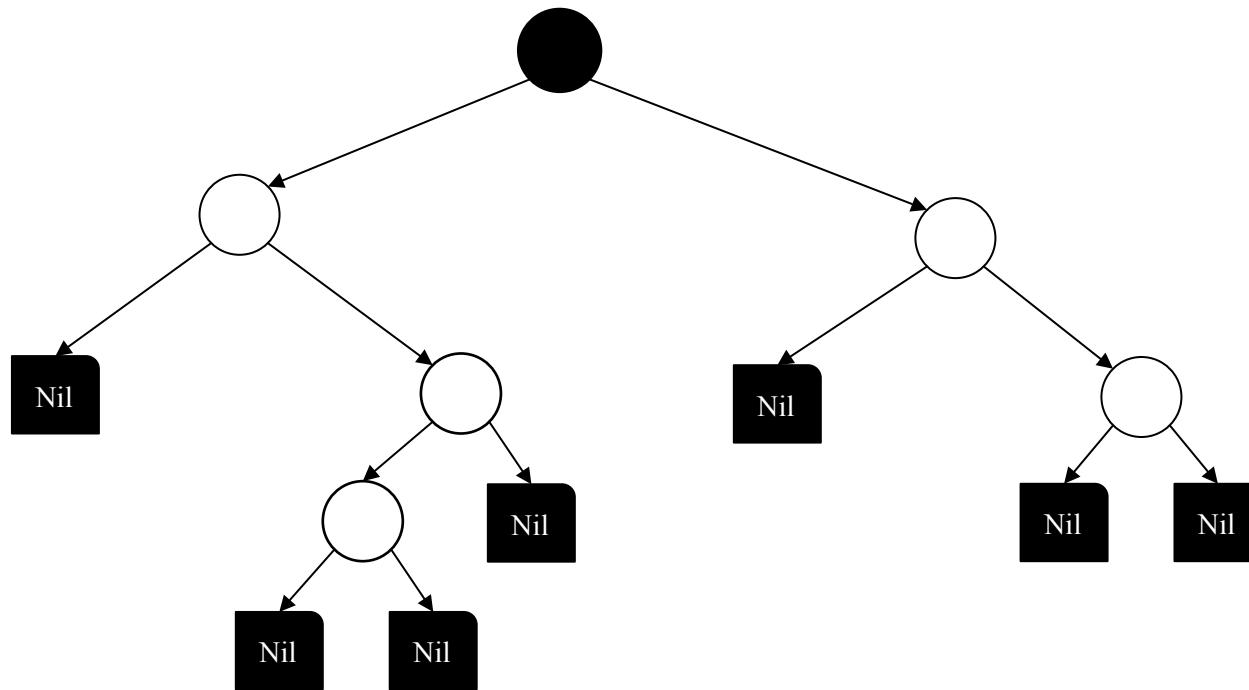
E' colorabile secondo le regole degli alberi R-N?



Esempio 1



Esempio 2



Proprietà degli alberi R-N

Prop. L'altezza massima di un albero R-N con n nodi è $2 \log_2(n + 1)$.

Dim. Sia T un albero R-N con n nodi interni.

- Per ogni nodo x il sottoalbero T_x con radice in x ha $\geq 2^{\text{bh}(x)} - 1$ nodi interni
Per induzione sull'altezza k di x :

- $k = 0$: x è una foglia e $\text{bh}(x) = 0$; T_x ha $2^0 - 1 = 0$ nodi interni
- $k > 0$: x è interno, ha 2 figli y_1, y_2 di altezza $k - 1$ e $\text{bh}(y_i) \geq \text{bh}(x) - 1$
quindi i nodi interni di T_x sono almeno:

$$\begin{aligned} & (2^{\text{bh}(y_1)} - 1) + (2^{\text{bh}(y_2)} - 1) + 1 && \text{ip. ind.} \\ & \geq (2^{\text{bh}(x)-1} - 1) + (2^{\text{bh}(x)-1} - 1) + 1 && \text{bh}(y_i) \geq \text{bh}(x) - 1 \\ & = 2^{\text{bh}(x)} - 1 \end{aligned}$$

Proprietà degli alberi R-N

Prop. L'altezza massima di un albero R-N con n nodi è $2 \log_2(n + 1)$.

Dim. Sia T un albero R-N con n nodi interni.

- Per ogni nodo x il sottoalbero T_x con radice in x ha $\geq 2^{\text{bh}(x)} - 1$ nodi interni
- Se $h = \text{height}(T)$ per la regola del rosso $\text{bh}(T) \leq h/2$

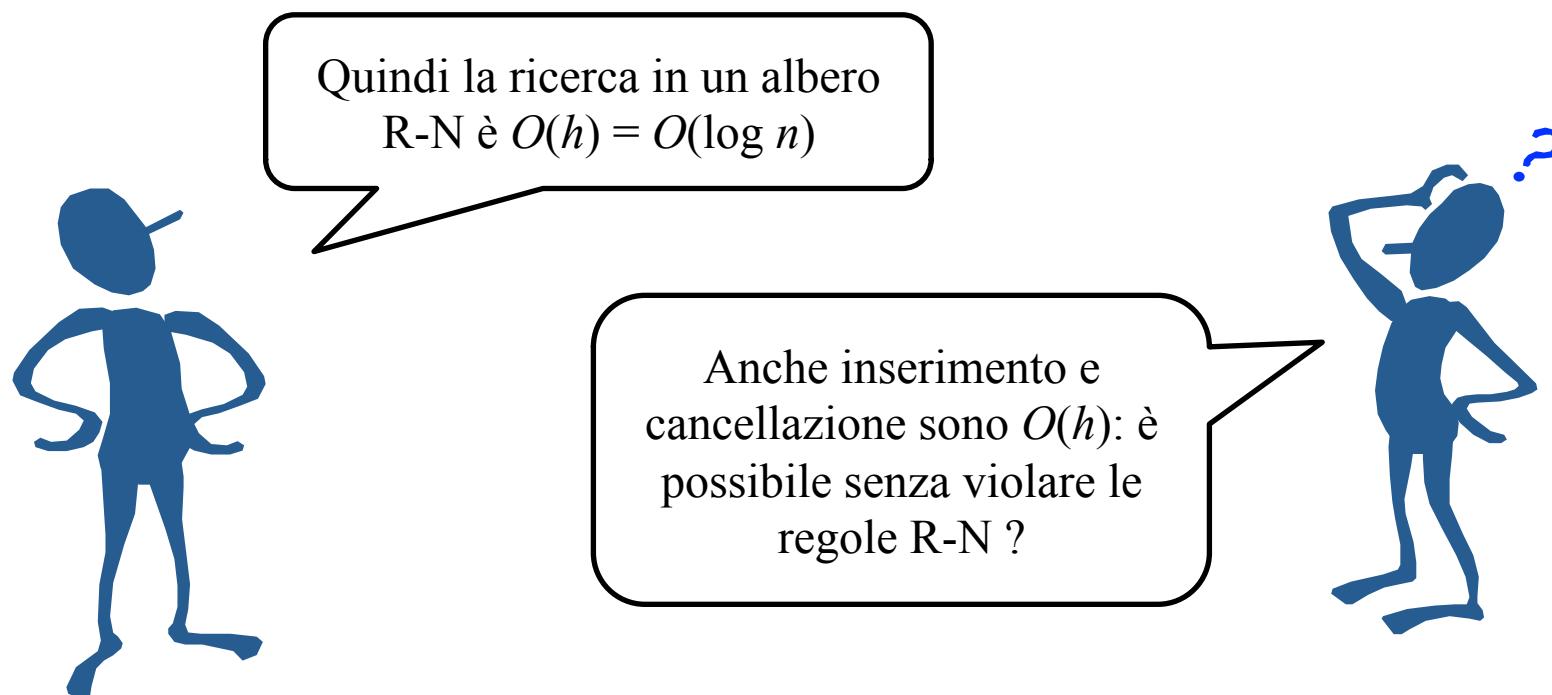
Quindi:

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

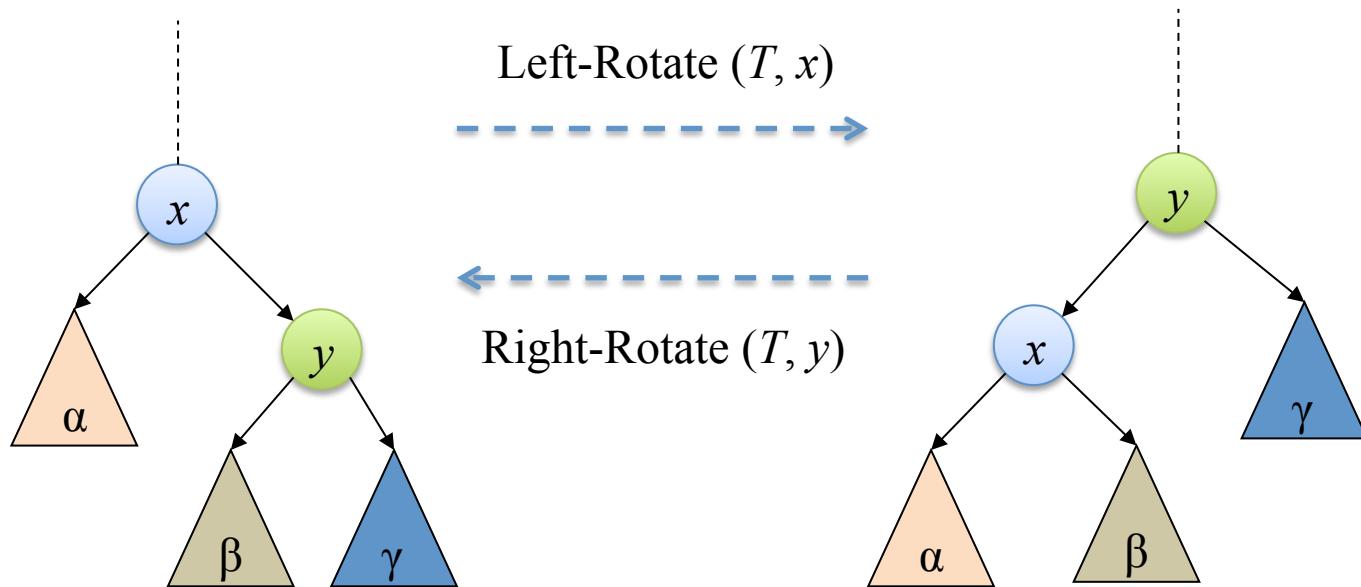
□

Proprietà degli alberi R-N

Prop. L'altezza massima di un albero R-N con n nodi è $2 \log_2(n + 1)$.



Rotazioni



Dopo una rotazione
l'albero rimane di ricerca

Rotazioni

LEFT-ROTATE(T, x)

$y \leftarrow x.right$

$x.right \leftarrow y.left$ (β)

if $y.left \neq nil$ **then** $y.left.parent \leftarrow x$
end if

$y.parent \leftarrow x.parent$

if $x.parent = nil$ **then** $T \leftarrow y$

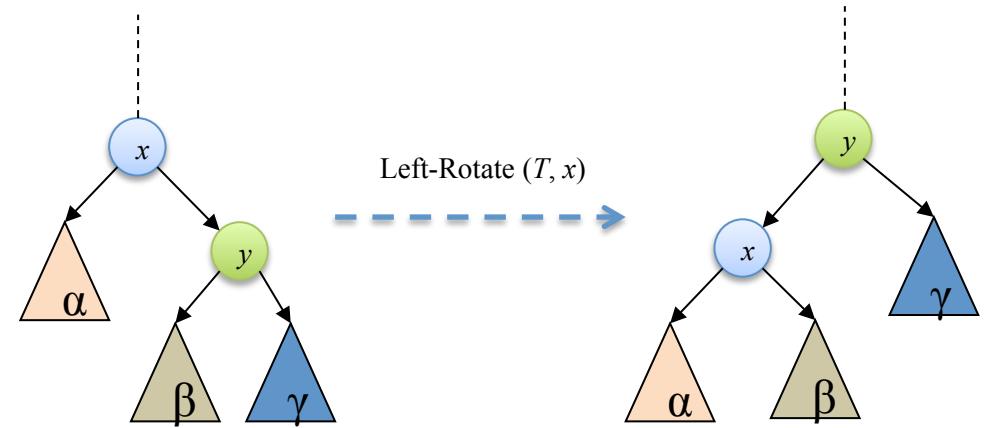
else

if $x = x.parent.left$ **then** $x.parent.left \leftarrow y$
else $x.parent.right \leftarrow y$
end if

end if

$y.left \leftarrow x$

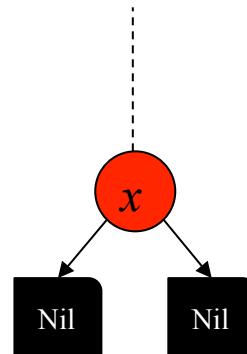
$x.parent \leftarrow y$



Inserimento

L'inserimento di x in T avviene in due fasi:

1. Inserimento di x **rosso** come per gli alberi di ricerca



Inserimento

L'inserimento di x in T avviene in due fasi:

1. Inserimento di x **rosso** come per gli alberi di ricerca
2. Ripristino delle proprietà R-N con rotazioni e ricolorazioni

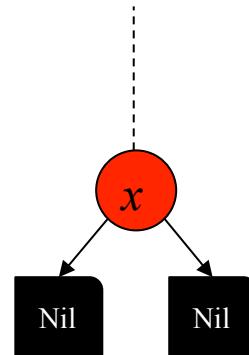
Inserimento

- x il nuovo nodo
- $p = x.parent$ (padre)
- $g = p.parent$ (nonno)
- u = il fratello di p (zio)

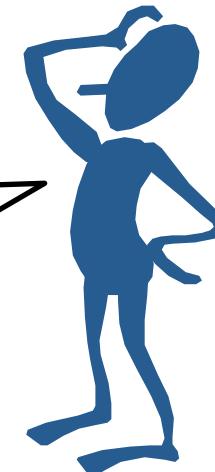
Siano x e p entrambi rossi



Inserimento

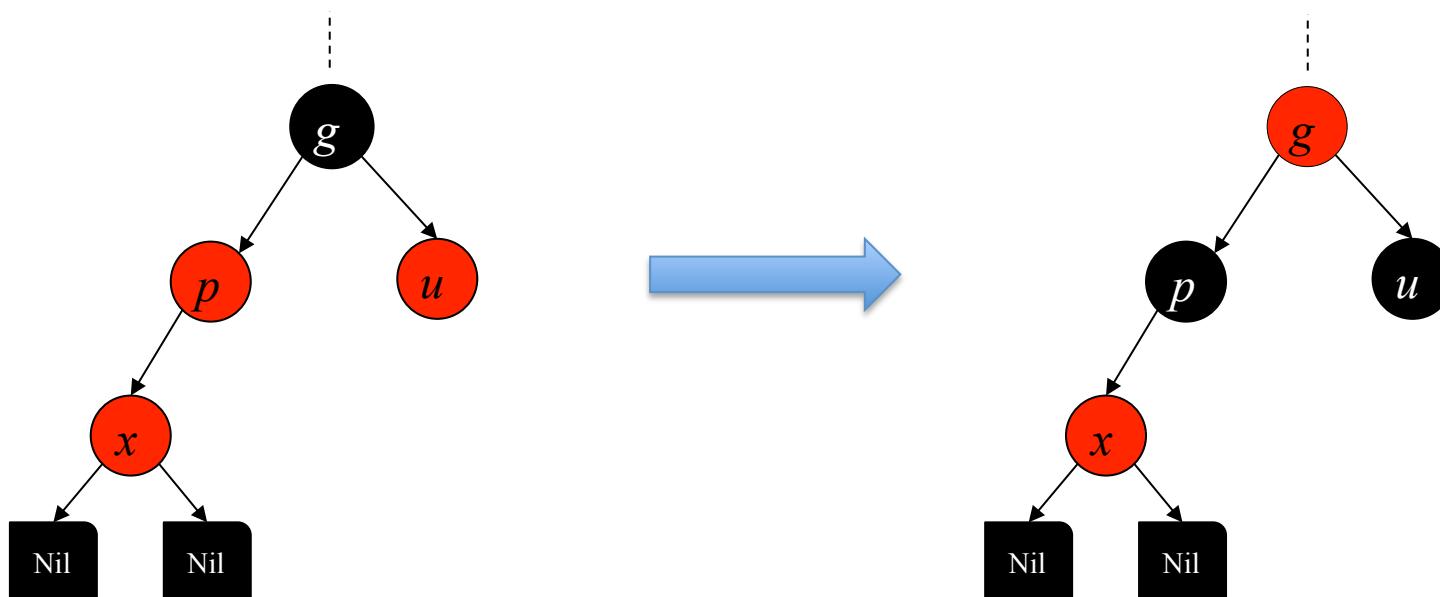


Se l'inserimento è in radice basta
cambiare il colore in nero;
altrimenti posso avere due rossi
contigui



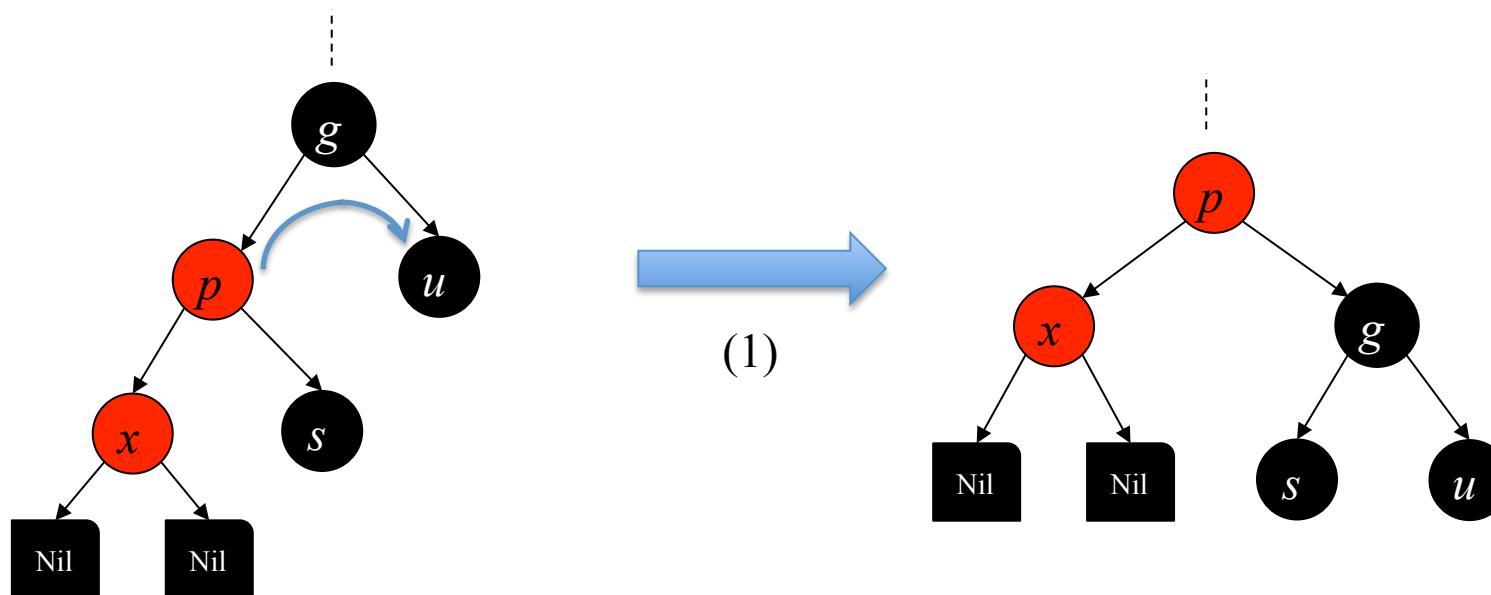
Inserimento

- Caso 1: lo zio u è rosso



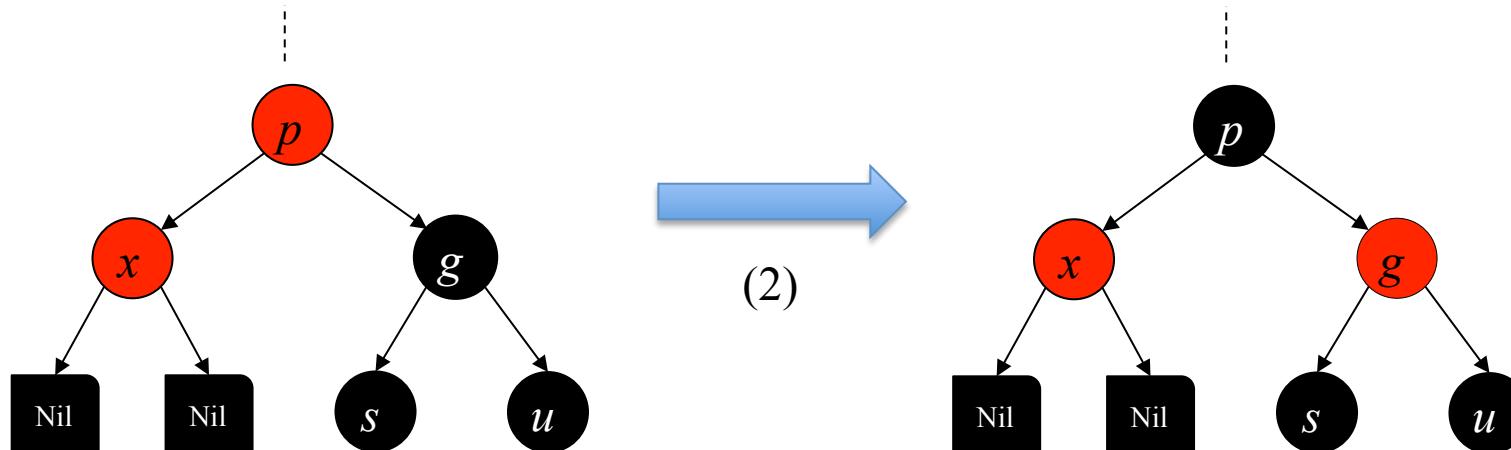
Inserimento

- Caso 2: lo zio u è nero e x figlio sinistro



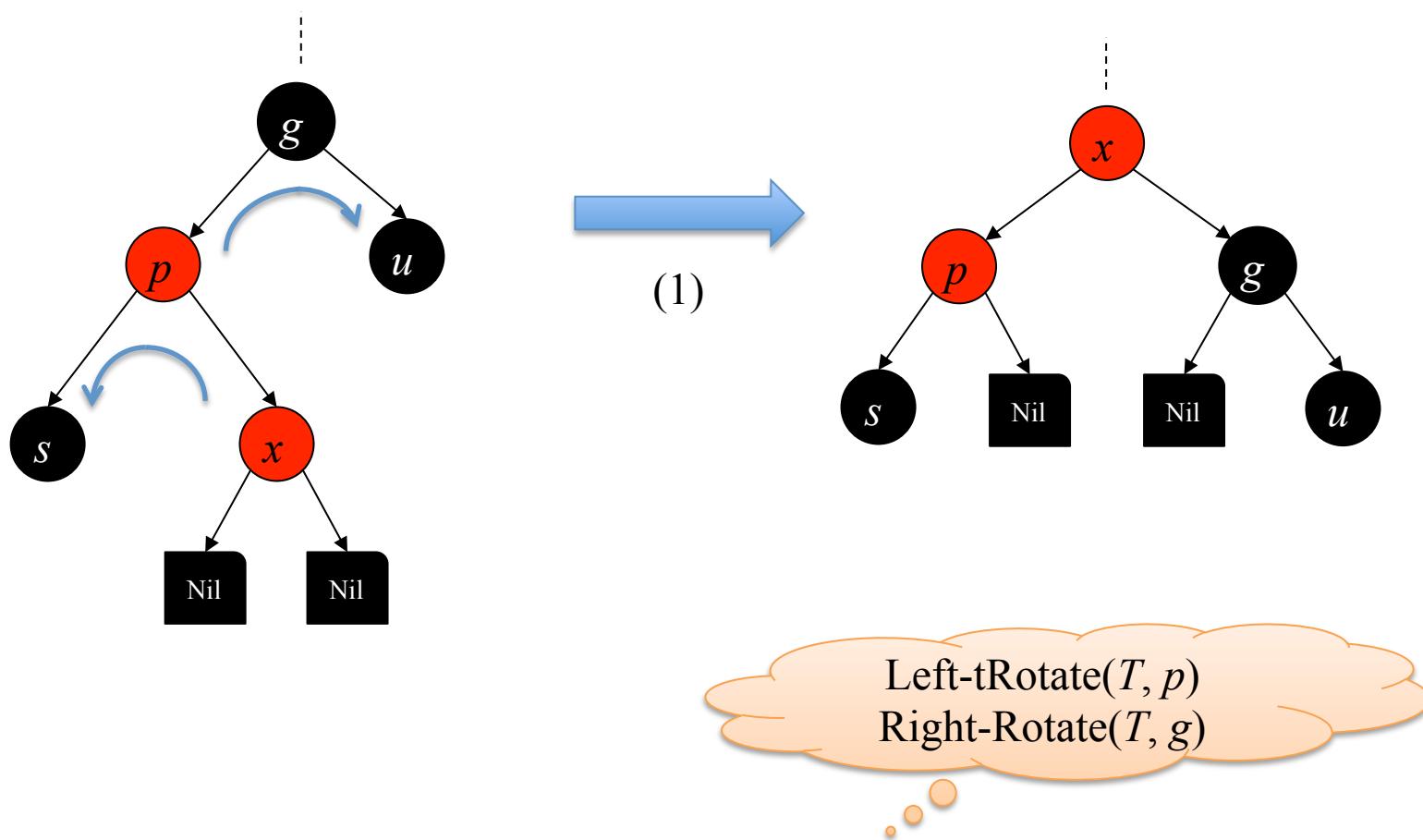
Inserimento

- Caso 2: lo zio u è nero e x figlio sinistro



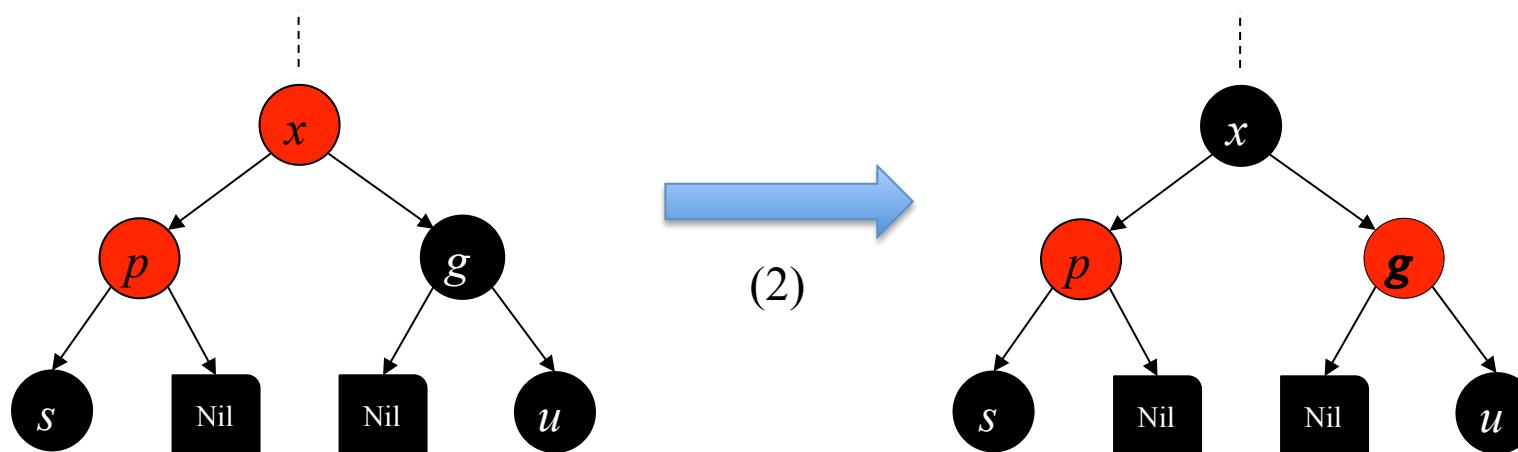
Inserimento

- Caso 3: lo zio u è nero e x figlio destro



Inserimento

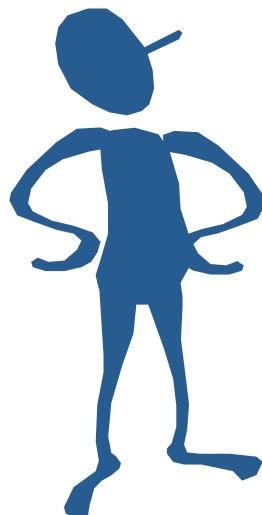
- Caso 3: lo zio u è nero e x figlio destro



Cancellazione

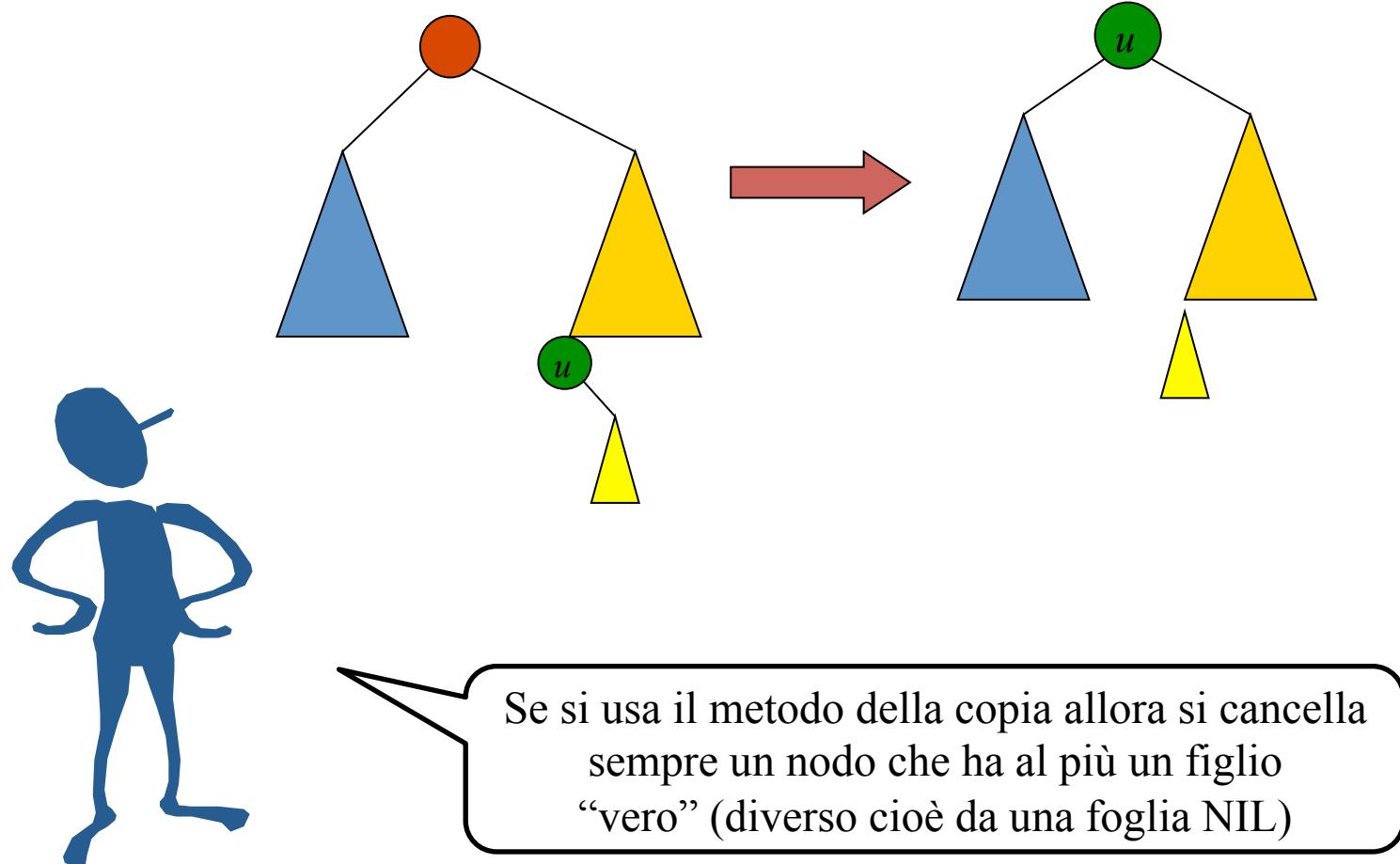
Anche la cancellazione avviene in due tempi:

- Cancellazione come in un albero di ricerca ordinario
- Riorganizzazione per ricolorazione/rotazione

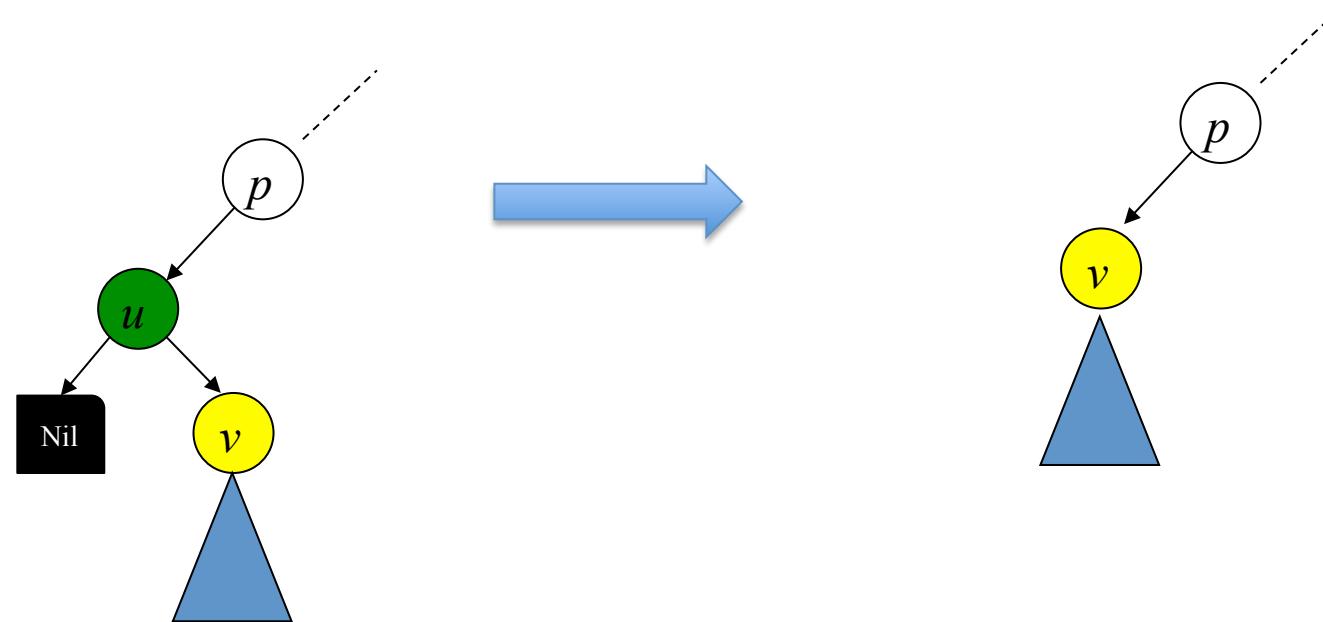


Se si usa il metodo della copia allora si cancella
sempre un nodo che ha al più un figlio
“vero” (diverso cioè da una foglia NIL)

Cancellazione

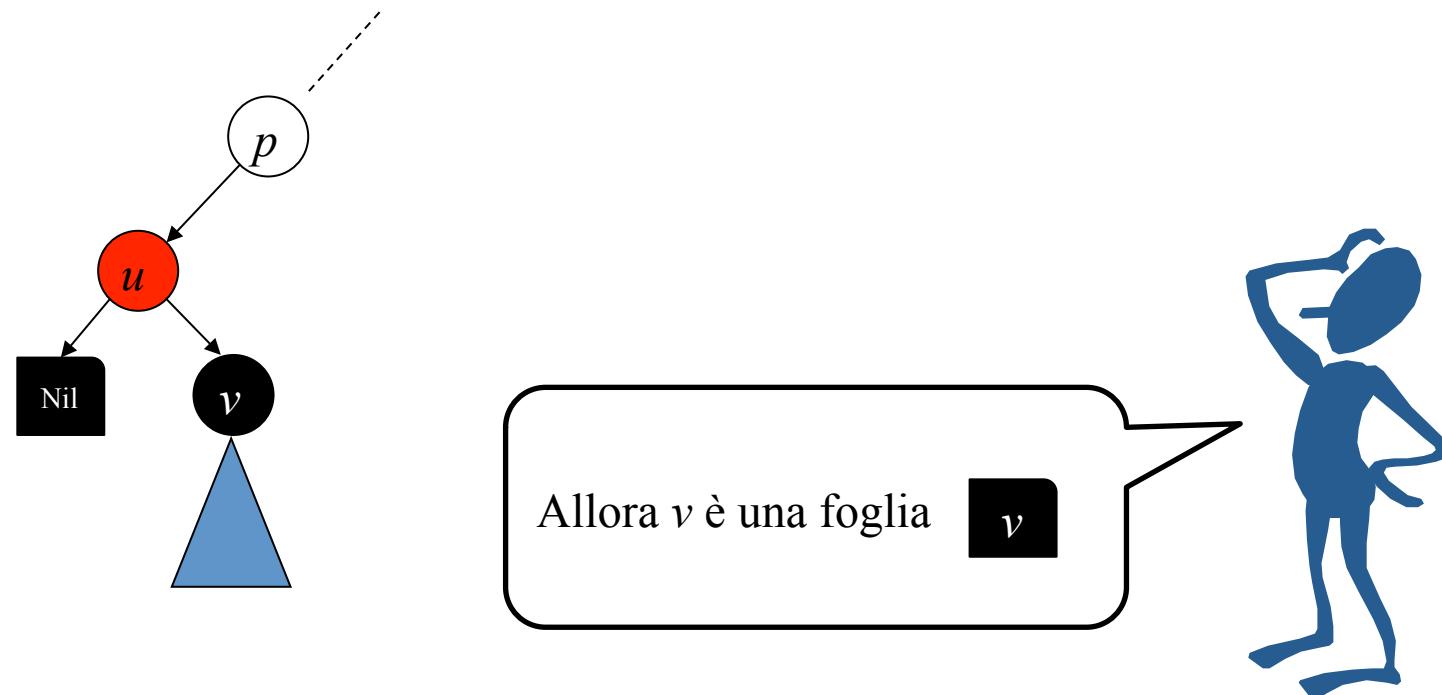


Cancellazione



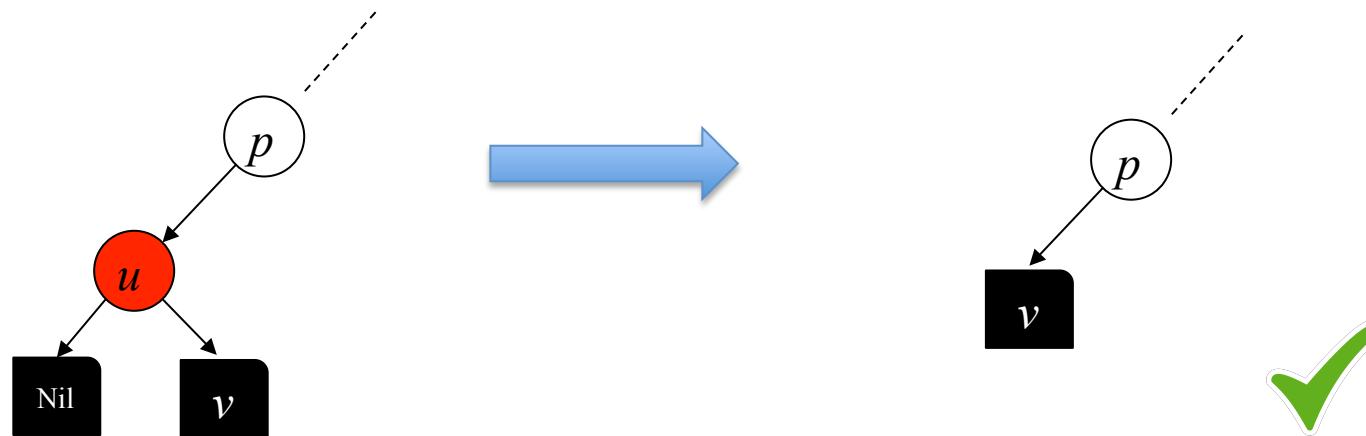
Cancellazione

- Caso 1: u era rosso, dunque v è nero



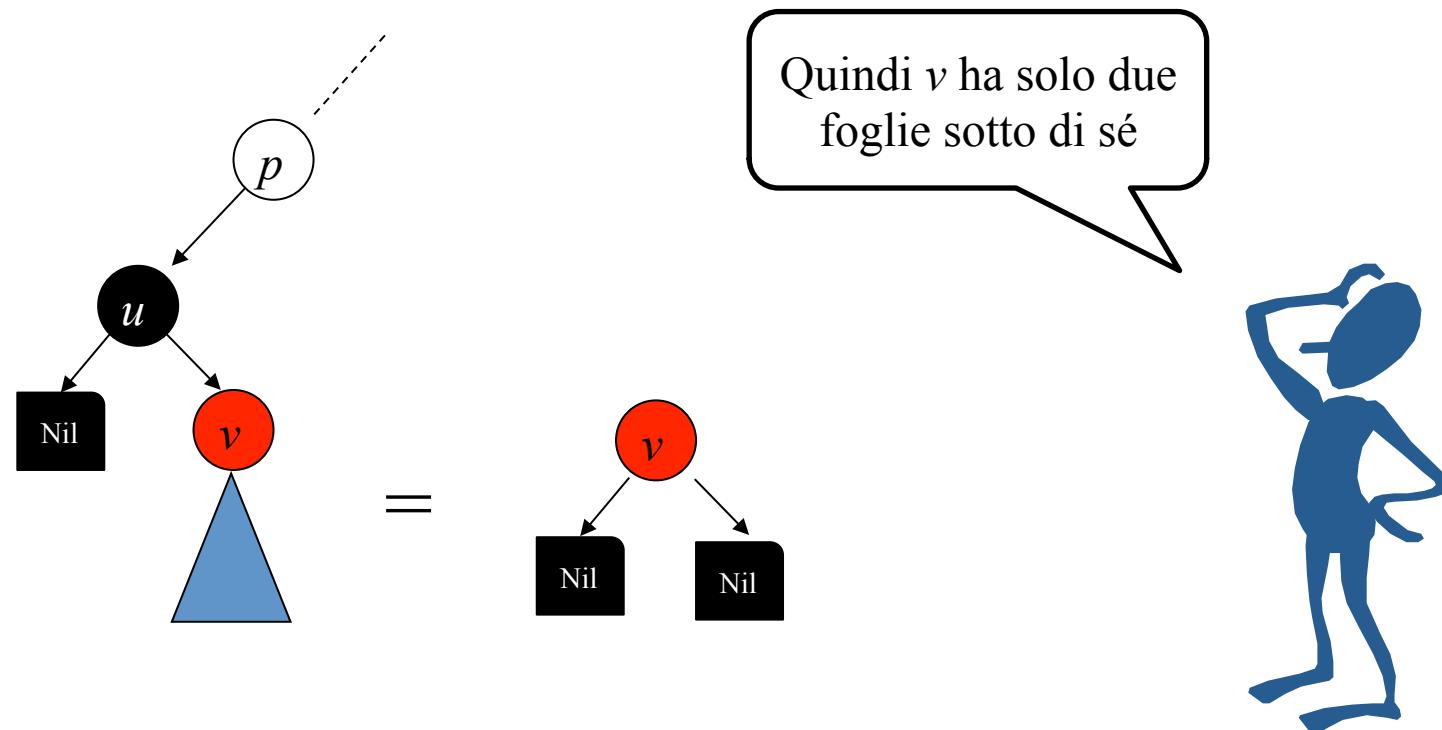
Cancellazione

- Caso 1: u era rosso, dunque v è nero



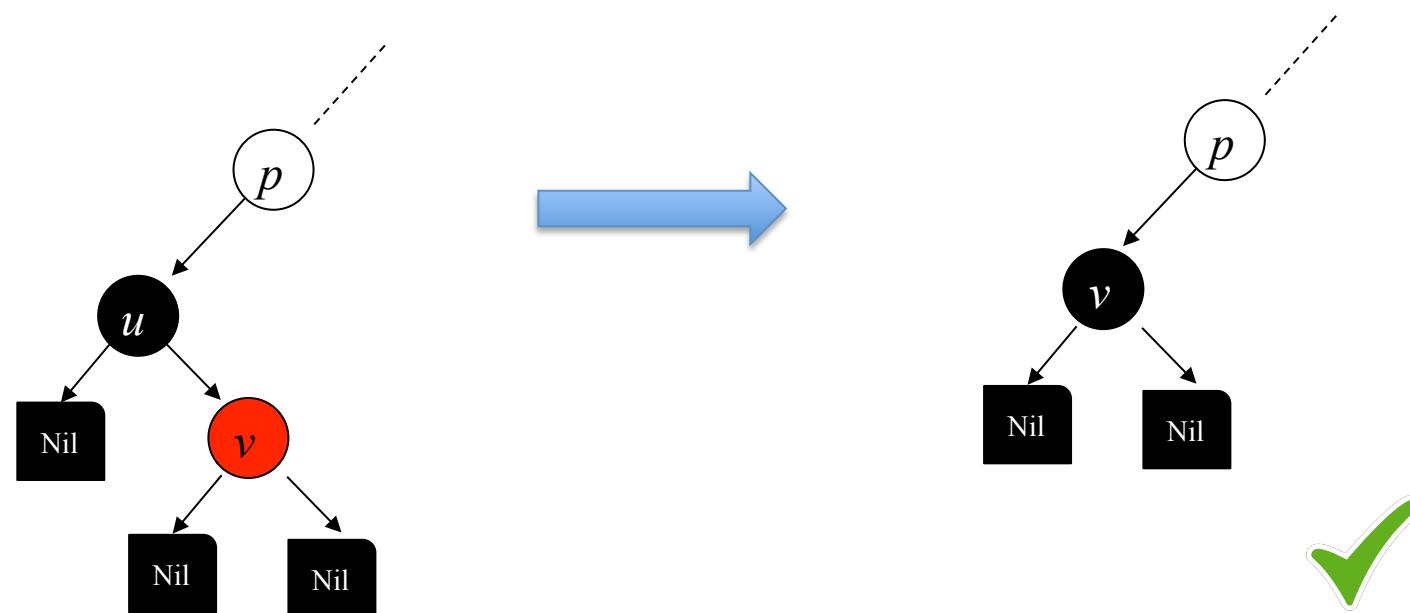
Cancellazione

- Caso 2: u era nero, e v è rosso



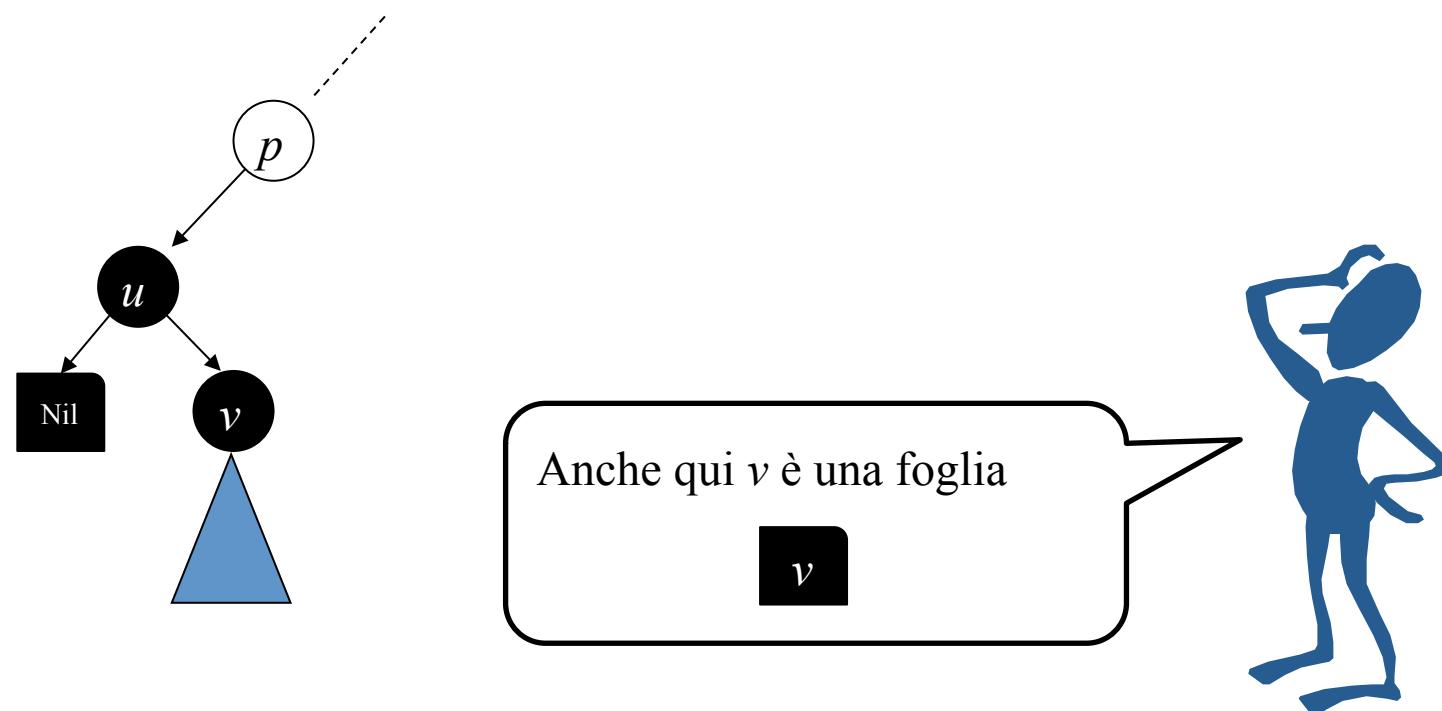
Cancellazione

- Caso 2: u era nero, e v è rosso



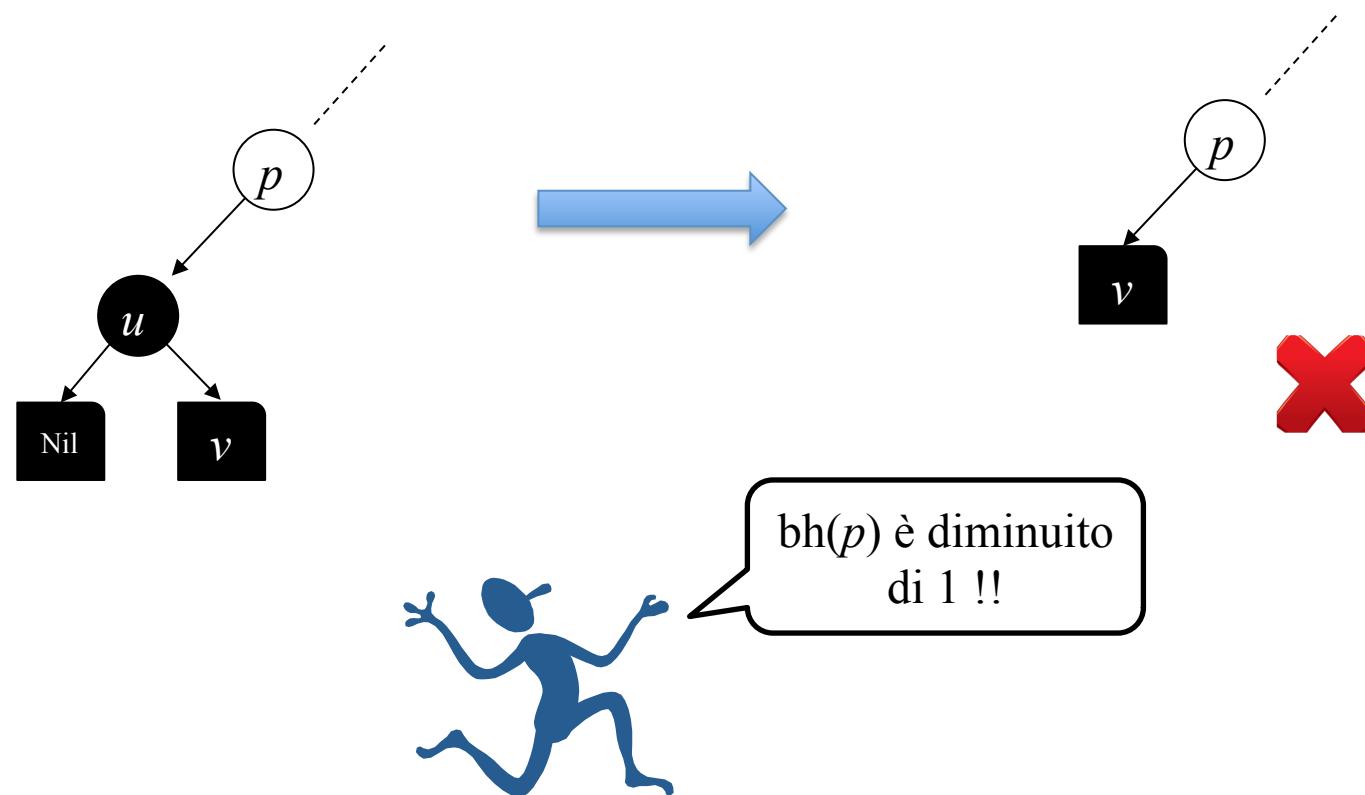
Cancellazione

- Caso 3: u era nero, e v è nero



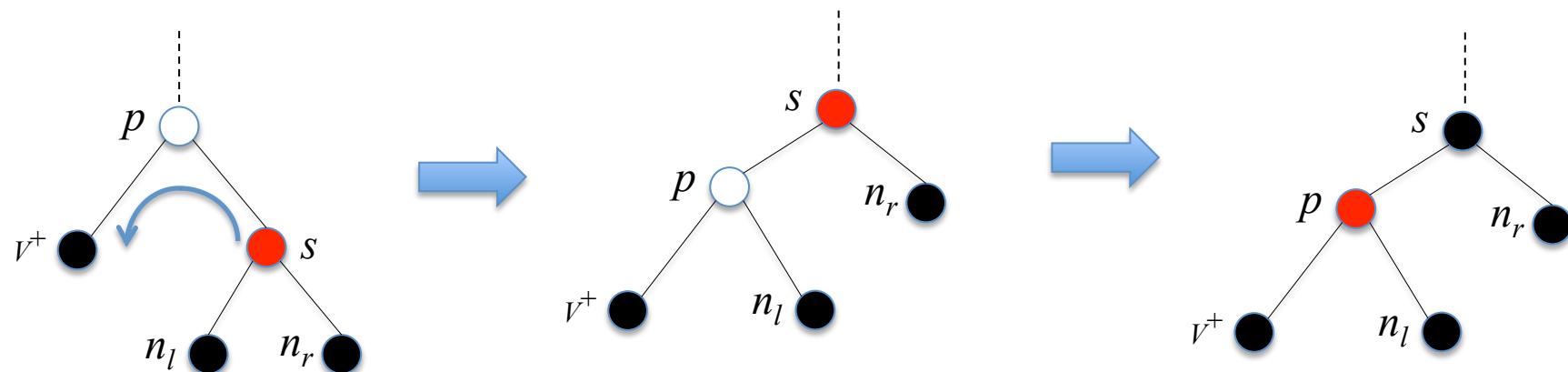
Cancellazione

- Caso 3: u era nero, e v è nero



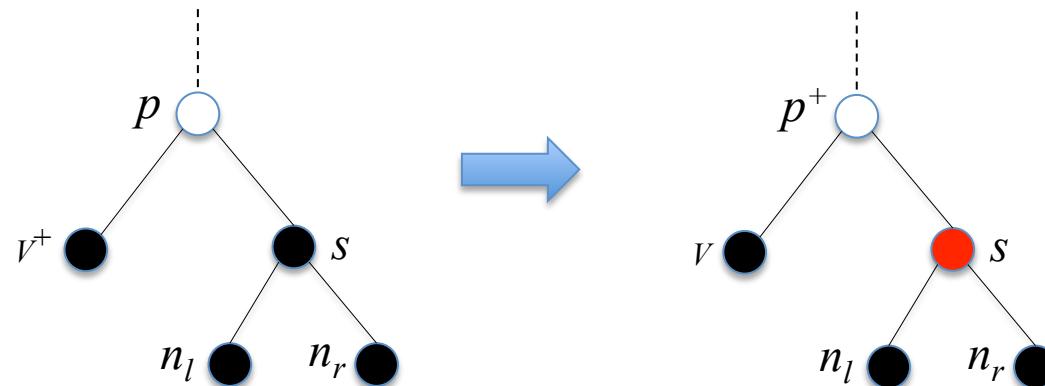
Cancellazione

- Caso 3.1 v ha un fratello s rosso



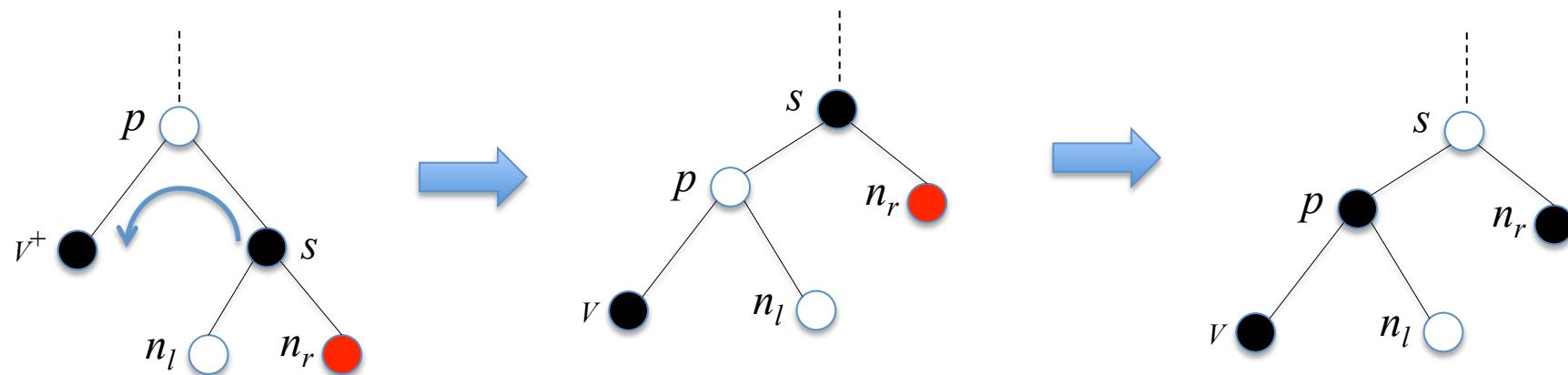
Cancellazione

- Caso 3.2 v ha un fratello s nero con figli neri



Cancellazione

- Caso 3.3 v ha un fratello s nero con un figlio rosso



Cancellazione

- Caso 3.4 v ha un fratello s nero con un figlio nero ed uno rosso

