

## Operazioni di modifica su un albero binario di ricerca

- Operazioni di modifica - operazioni su un insieme dinamico  $S$  che modificano le proprietà dell'insieme.

$\text{Insert}(S,x)$  - inserisce in  $S$  l'elemento puntato da  $x$

$\text{Delete}(S,x)$  - rimuove da  $S$  l'oggetto puntato da  $x$

## Inserzione

Possiamo sempre inserire un nuovo elemento  $z$  come foglia.

Per decidere la posizione del nuovo elemento possiamo usare la seguente strategia:

- discendiamo tutto l'albero, partendo dalla radice. In corrispondenza di ogni nodo  $x$ , andiamo a sinistra se  $key[z] < key[x]$ , andiamo a destra altrimenti.

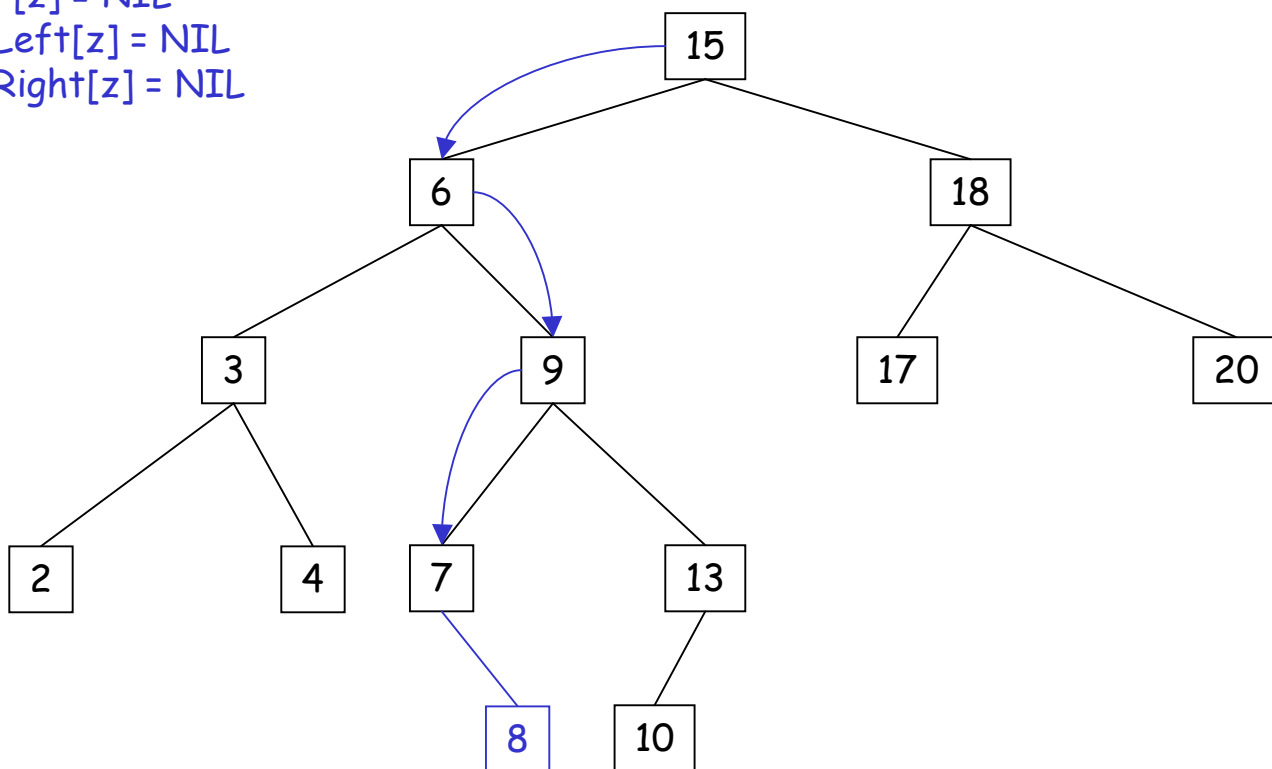
Supponiamo di voler inserire un nodo  
i cui campi sono inizializzati a:

$Key[z] = 8$

$P[z] = NIL$

$Left[z] = NIL$

$Right[z] = NIL$



Se seguo questo schema l'elemento  $z$  viene posizionato nella posizione giusta. Infatti, per costruzione, ogni antenato di  $z$  si ritrova  $z$  nel sotto-albero giusto.

## Inserzione

```
Tree-insert(T,z)
Y ← NIL
X ← root[T]
While (x≠NIL)
    do y ← x
        if (key[z] < key[x])
            then x ← left[x]
            else x ← right[x]
P[z] ← y
If (y ≠ NIL)
    then if (key[z] < key[y])
        then left[y] ← z
        else right[y] ← z
    else root[T] ← z
```

Cerchiamo la posizione giusta per una foglia con chiave pari a key[z].

y → indice del parente di z

Aggiorniamo il puntatore p[z]

Aggiorniamo i puntatori del nodo y.

Se y=NIL, l'albero contiene solo il nodo z.

Il nodo z e' allora la radice dell'albero. Non e' necessario aggiornare i puntatori left[z] e right[z]

$T(n) = O(h)$   
h = altezza dell'ABR

**N.B.** Operazioni di inserzione successive possono "linearizzare" l'albero.

## Cancellazione

3 situazioni possibili:

1) Cancellazione di un nodo  $z$  privo di figli -

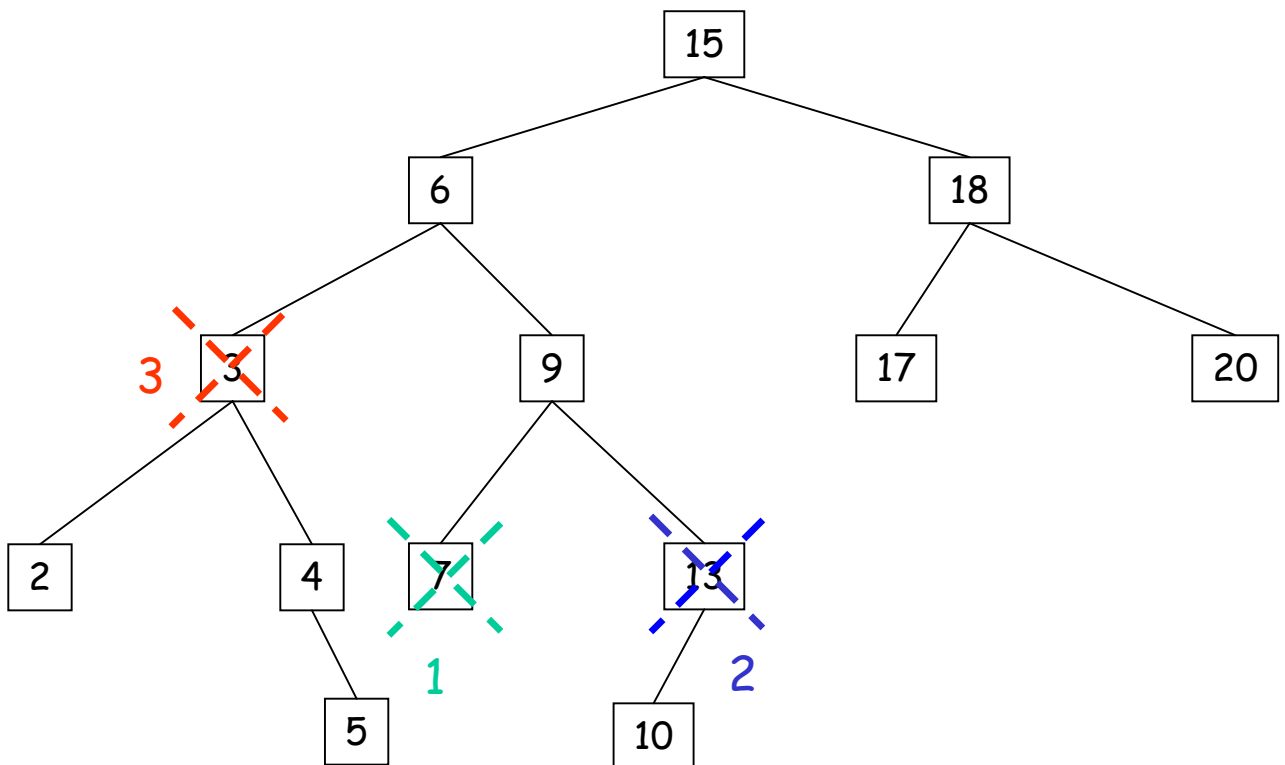
Modifichiamo il padre di  $z$  ( $p[z]$ ) in modo che non punti più a  $z$

2) Cancellazione di un nodo  $z$  con un unico figlio -

Estraiamo il nodo  $z$  creando un collegamento tra il padre di  $z$  ( $p[z]$ ) ed il figlio di  $z$

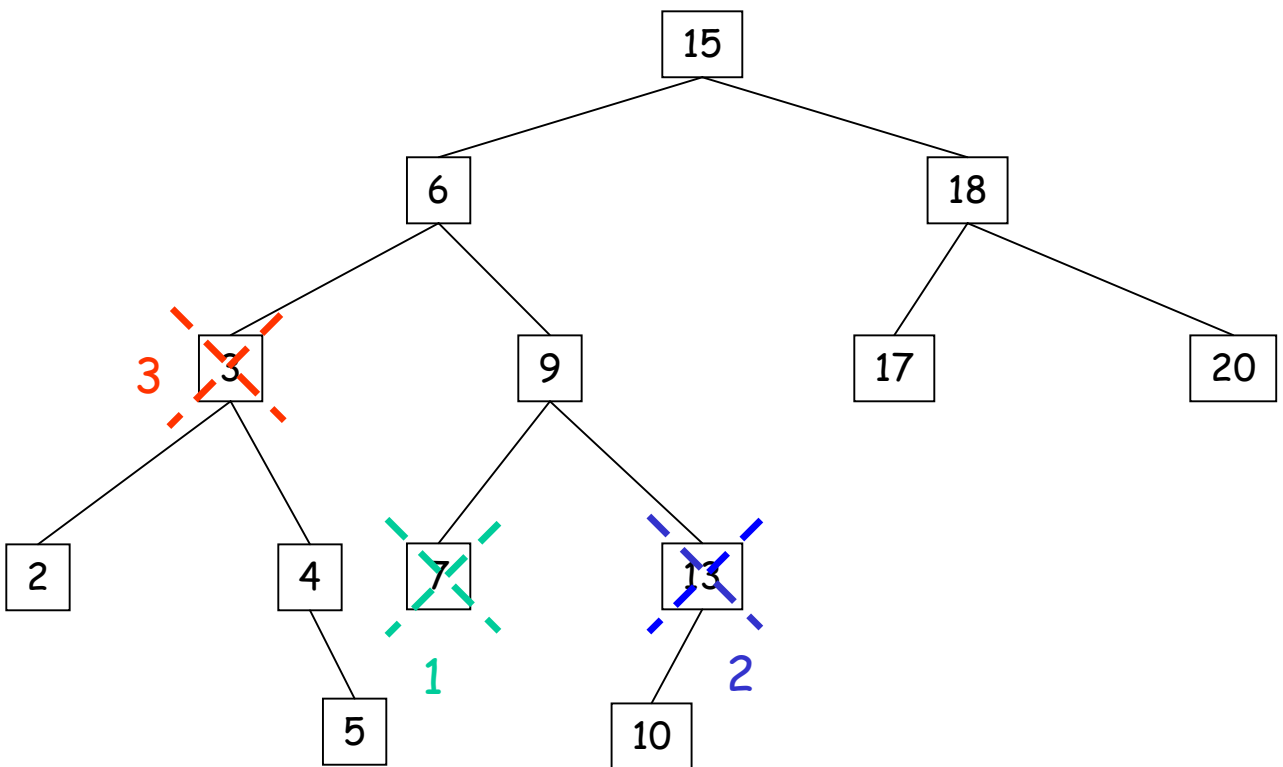
3) Cancellazione di un nodo  $z$  avente due figli.

Estraiamo il successore  $y$  di  $z$  e sostituiamo il contenuto di  $z$  con il contenuto di  $y$



## Cancellazione

```
Tree-Delete(T,z)
  if (left[z]=NIL) o (right[z]=NIL)
    then y ← z
    else y ← Tree-Successor(z)
  If (left[y] ≠ NIL)
    then x ← left[y]
    else x ← right[y]
  If (x ≠ NIL)
    then p[x] ← p[y]
  If (p[y] = NIL)
    then root[T] ← x
    else if (y = left[p[y]])
      then left[p[y]] ← x
      else right[p[y]] ← x
  If (y≠z)
    then key[z] ← key[y]
  Return y
```



## Nodo senza figli

Tree-Delete(T,z)

if (left[z]=NIL) o (right[z]=NIL)

then  $y \leftarrow z$

else  $y \leftarrow \text{Tree-Successor}(z)$

If (left[y]  $\neq$  NIL)

then  $x \leftarrow \text{left}[y]$

else  $x \leftarrow \text{right}[y]$  (NIL)

If ( $x \neq \text{NIL}$ )

then  $p[x] \leftarrow p[y]$

If ( $p[y] = \text{NIL}$ )

then  $\text{root}[T] \leftarrow x$

else if ( $y = \text{left}[p[y]]$ )

then  $\text{left}[p[y]] \leftarrow x$

else  $\text{right}[p[y]] \leftarrow x$

If ( $y \neq z$ )

then  $\text{key}[z] \leftarrow \text{key}[y]$

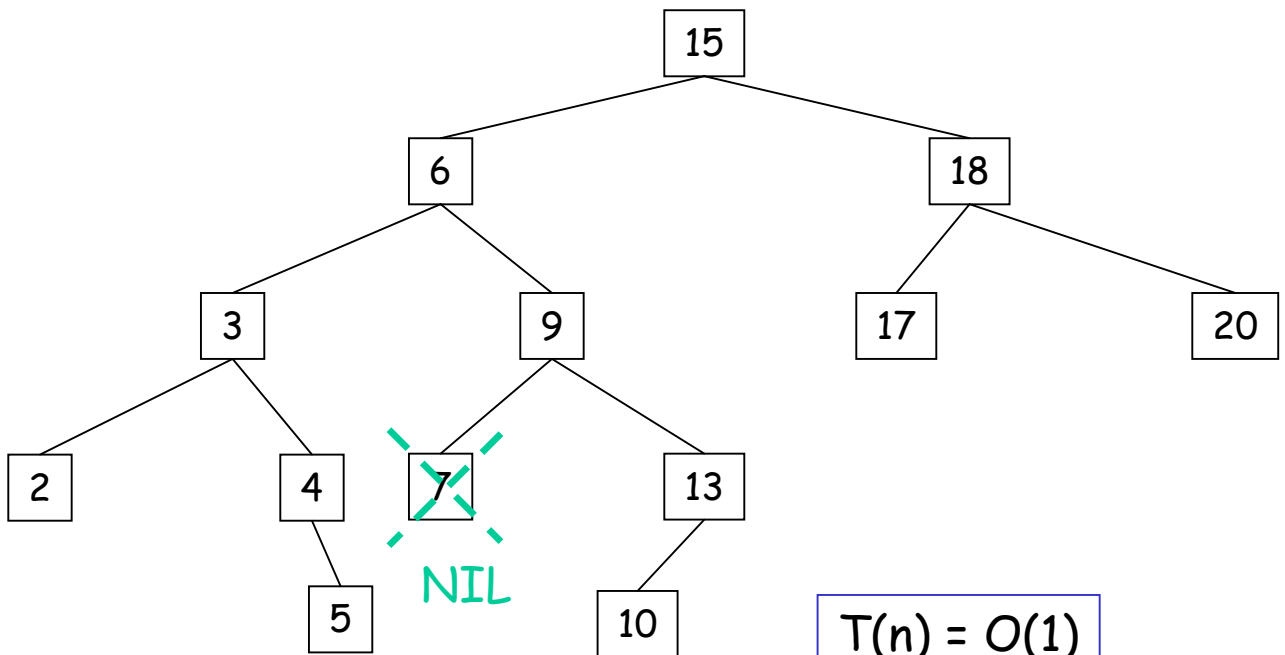
Return y

Se il nodo z era la radice, allora

$\text{root}[T] = \text{NIL}$

altrimenti

sostituisce il puntatore a z con NIL



## Nodo con un figlio

Tree-Delete(T,z)

if (left[z]=NIL) o (right[z]=NIL)

then  $y \leftarrow z$   $y = z$

else  $y \leftarrow \text{Tree-Successor}(z)$

If (left[y]  $\neq$  NIL)

then  $x \leftarrow \text{left}[y]$

else  $x \leftarrow \text{right}[y]$

$x$  = puntatore al figlio di z

If ( $x \neq \text{NIL}$ )

then  $p[x] \leftarrow p[y]$

$p[x]$  = puntatore al padre di z

If ( $p[y] = \text{NIL}$ )

se z era la radice

then  $\text{root}[T] \leftarrow x$

else if ( $y = \text{left}[p[y]]$ )

altrimenti

then  $\text{left}[p[y]] \leftarrow x$

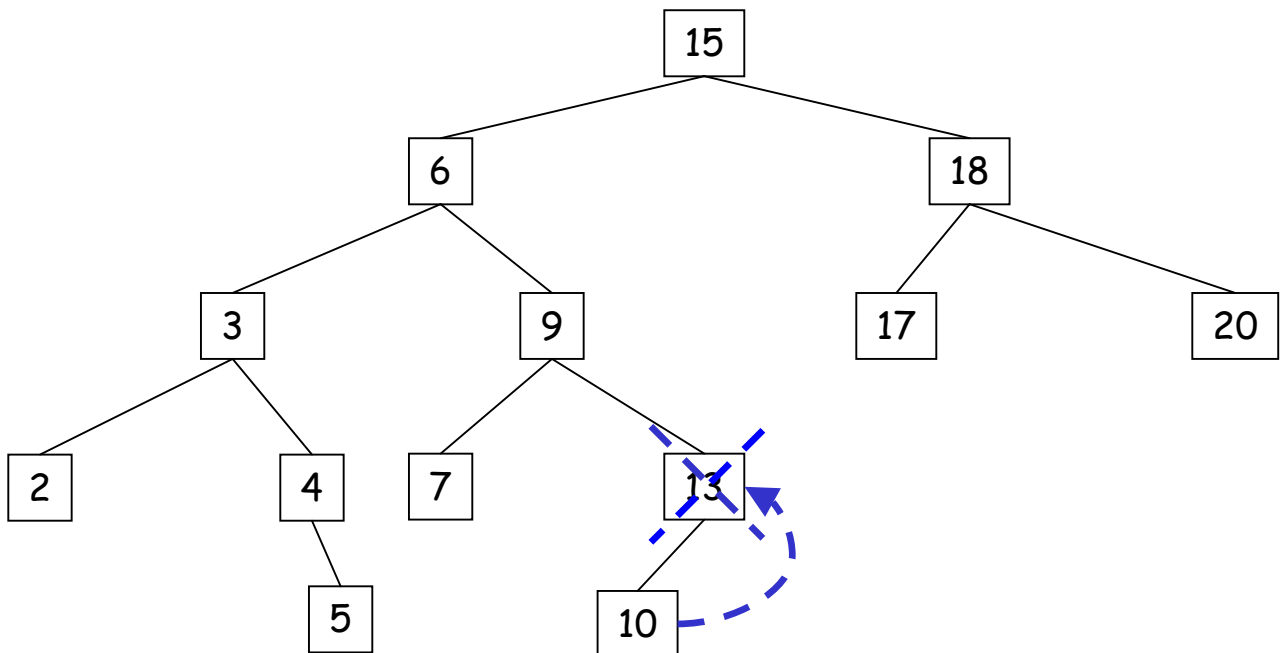
crea connessione tra il padre di z  
ed il figlio di z

else  $\text{right}[p[y]] \leftarrow x$

If ( $y \neq z$ )

then  $\text{key}[z] \leftarrow \text{key}[y]$

Return y



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

## Nodo con due figli

Tree-Delete(T,z)

if (left[z]=NIL) o (right[z]=NIL)

then  $y \leftarrow z$

else  $y \leftarrow \text{Tree-Successor}(z)$

$y = \text{successore di } z$

If (left[y]  $\neq$  NIL)

then  $x \leftarrow \text{left}[y]$

else  $x \leftarrow \text{right}[y]$

$x = \text{figlio destro del succes. di } z$

If ( $x \neq \text{NIL}$ )

then  $p[x] \leftarrow p[y]$

N.B. - il succes. di y non puo' avere f.sinistri

aggiorna il puntatore al padre di x

If ( $p[y] = \text{NIL}$ )

se z era la radice

then  $\text{root}[T] \leftarrow x$

else if ( $y = \text{left}[p[y]]$ )

altrimenti

then  $\text{left}[p[y]] \leftarrow x$

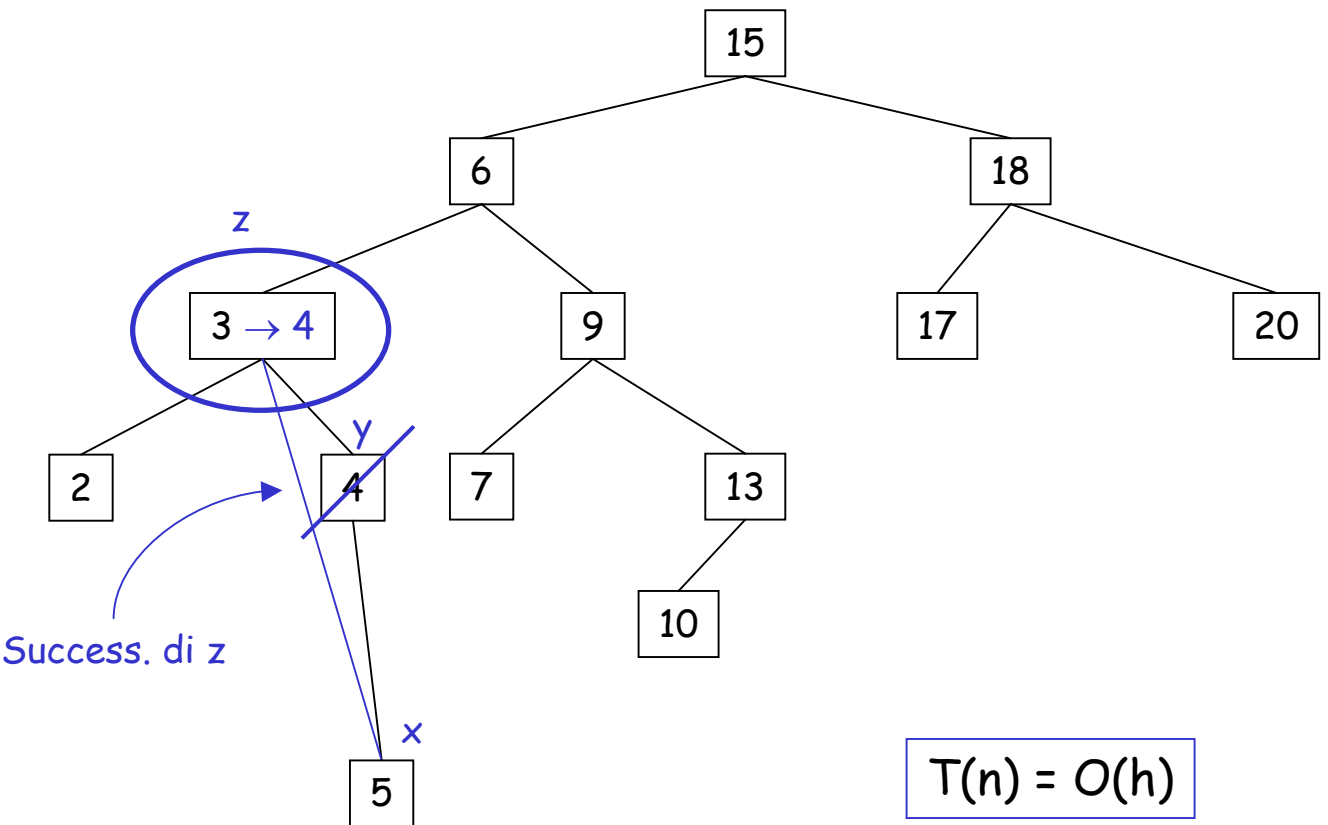
else  $\text{right}[p[y]] \leftarrow x$

If ( $y \neq z$ )

then  $\text{key}[z] \leftarrow \text{key}[y]$

sostituisce chiave di z con chiave di y

Return y





In conclusione:

Le operazioni di **inserzione** e di **cancellazione** di un nodo in un ABR hanno tempo di esecuzione  $T(n) = O(h)$  dove **h** e' **l'altezza** dell'albero.

## RB-Alberi

Le operazioni di inserzione e cancellazione descritte precedentemente possono "linearizzare" un ABR.

**Es.** - Supponiamo di introdurre un elemento con chiave minore della chiave minima dell'ABR, poi un altro elemento con chiave ancora minore, e così via ...

Noi siamo interessati a mantenere l'albero **bilanciato** (vogliamo cioè che tutti i cammini dalla radice alle varie foglie abbiano approssimativamente la stessa lunghezza)

Le varie operazioni hanno infatti un tempo di esecuzione

$$T(n) = O(h) \quad h = \text{altezza dell' albero}$$

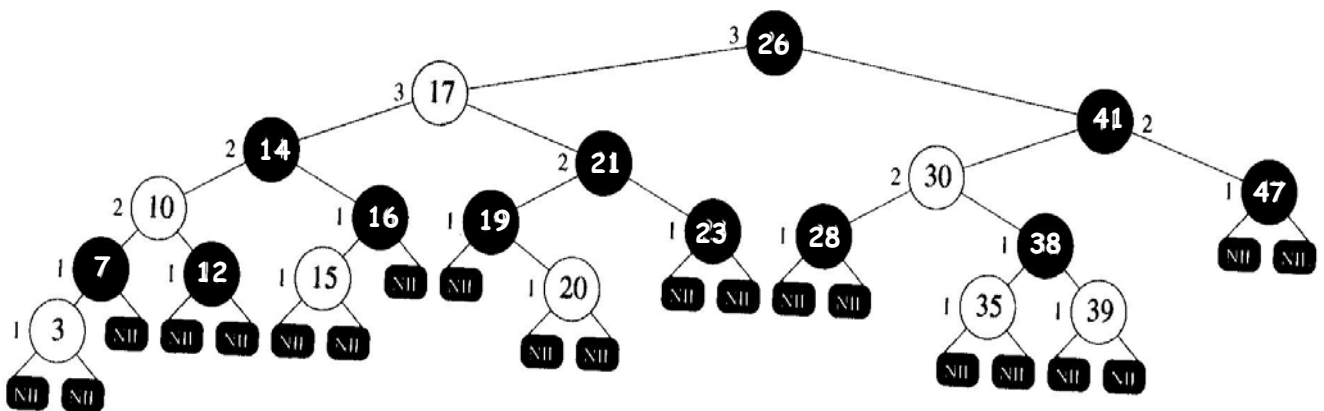
Per un fissato numero  $n$  di elementi l'altezza  $e'$  minima se l'albero  $e'$  completo:

$$n \cong 2^{h+1}$$

**RB-alberi** - schema per la costruzione di alberi bilanciati.  
Albero Binario di ricerca con un campo binario aggiuntivo in ogni suo nodo  
**Color[x] = Red, Black**

Un RB-albero deve soddisfare le seguenti **Proprieta' RB**:

- 1) Ciascun nodo e' rosso o nero;
- 2) Ciascuna foglia NIL e' nera;
- 3) Se un nodo e' rosso entrambi i suoi figli sono neri
- 4) Ogni cammino semplice da un nodo ad una foglia sua discendente contiene lo stesso numero di nodi neri.



**N.B.** - abbiamo rappresentato i valori NIL come puntatori a nodi esterni (foglie) dell'albero binario di ricerca. I nodi "fisici", in questo schema, sono tutti nodi interni.

**N.B.** - La proprieta' 4) consente di definire la b-altezza  $bh(x)$  di un nodo  $x$ .-

$bh(x)$  = numero di nodi neri neri su un cammino da  $x$  ad una foglia.

## Teorema

Un RB-albero con  $n$  nodi interni ha al piu' un'altezza  $2\log(n+1)$   
In altre parole:

Per un RB-albero  $\rightarrow h = O(\log(n))$

-----  
La validita' della affermazione precedente puo' essere dimostrata per induzione.

Ma ... Qual e' il trucco? Perche' un RB-albero deve essere bilanciato?

Intuitivamente:

$\rightarrow$  La situazione migliore e' quella in cui l'albero e' completo  $\rightarrow$  tutti i nodi hanno due figli e tutti i cammini da un nodo alle varie foglie hanno la stessa lunghezza.

Questa condizione e' difficile. Chiedo pero' qualcosa di equivalente:

$\rightarrow$  Riempio il mio albero introducendo nuove foglie (e' piu' facile ragionare su alberi pieni);

$\rightarrow$  Invece di misurare la lunghezza dei vari cammini misuro la sua "b-lunghezza" (numero di nodi neri che incontro lungo il cammino). Ha senso farlo perche' la proprieta' 2) mi dice che la "b-lunghezza" e' strettamente legata alla lunghezza "fisica" di un cammino discendente;

"b-lunghezza"  $\geq$  (lunghezza/2)

$\rightarrow$  Richiedo (proprieta' 4) che le "b-lunghezze" di tutti i cammini da un nodo alle varie foglie abbiano la stessa lunghezza.

Di fatto  $\rightarrow$  richiedo che l'albero sia "completo" in termini di "b-altezze".

## Rotazioni

In genere un ABR di ricerca non e' un RB-albero. Inoltre, dato un RB-albero, le operazioni Tree-Insert e Tree-Delete non preservano le proprieta' degli RB-Alberi.

Per mantenere le proprieta' degli RB-Alberi devo, di volta in volta, modificare la struttura dell'albero (i puntatori dei vari nodi) ed i colori di qualche nodo dell'albero, stando attento a **preservare** la PROPRIETA' di ORDINAMENTO DEGLI ABR.

L'operazione che cambia la struttura dei puntatori si chiama **rotazione**

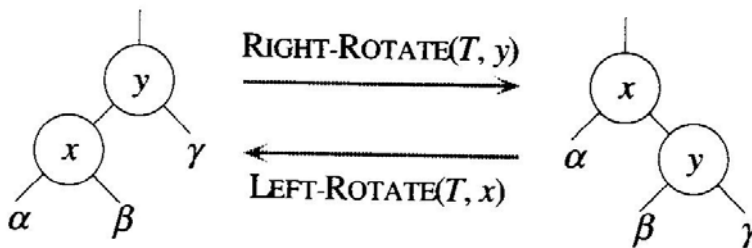


Fig. 14.2 Le operazioni di rotazione su un albero binario di ricerca. L'operazione  $\text{RIGHT-ROTATE}(T, x)$  trasforma la configurazione dei due nodi sulla sinistra nella configurazione che è sulla destra cambiando un numero costante di puntatori. La configurazione di destra può essere trasformata nella configurazione di sinistra attraverso l'operazione inversa  $\text{LEFT-ROTATE}(T, y)$ . I due nodi potrebbero essere ovunque in un albero binario di ricerca. Le lettere  $\alpha$ ,  $\beta$  e  $\gamma$  rappresentano sottoalberi arbitrari. Un'operazione di rotazione mantiene l'ordinamento delle chiavi secondo la visita in ordine simmetrico dell'albero: le chiavi in  $\alpha$  precedono  $\text{key}[x]$ , che precede le chiavi in  $\beta$ , che precede  $\text{key}[y]$ , che precedono le chiavi in  $\gamma$ .

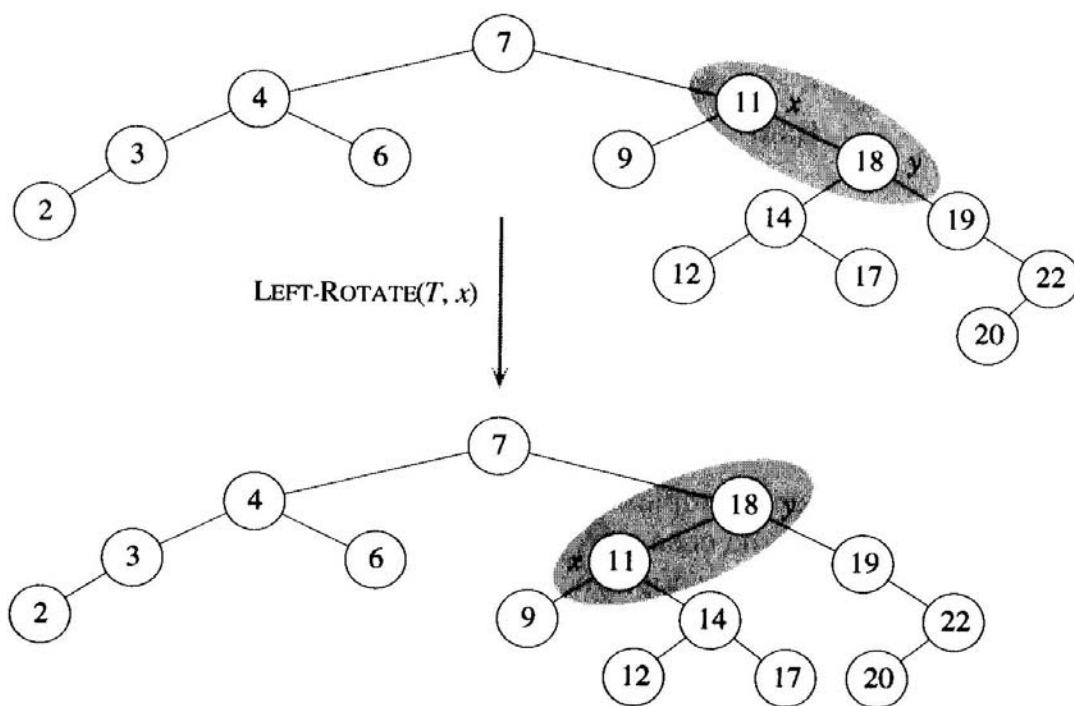


Fig. 14.3 Un esempio di come la procedura `LEFT-ROTATE(T,x)` modifichi un albero binario di ricerca. Le foglie `NIL` sono omesse. Le visite in ordine simmetrico dell'albero in input e di quello modificato producono la stessa sequenza di valori delle chiavi.

```

Left-Rotate(T,x)
  y ← right[x]
  right[x] ← left[y]
  if (left[y] ≠ NIL)
    then p[left[y]] ← x
    p[y] ← p[x]
  if (p[x] = NIL)
    then root[T] ← y
    else if (x=left[p[x]])
      then left[p[x]] ← y
      else right[p[x]] ← y
  Left[y] ← x
  P[x] ← y

```

$T(n) = O(1)$   
 Opera sui puntatori  
 e lascia invariati  
 tutti gli altri campi  
 di un nodo

## Inserzione

```
RB-Insert(T,x)
  Tree-Insert(T,x)
  color[x] ← RED
  while (x≠root[T]) e (color[p[x]] = Red)
    do if (p[x] = left[p[p[x]])
        then y ← right[p[p[x]])
        if color[y] = RED
          then color[p[x]] ← black
              color[y] ← black
              color[p[p[x]]) ← red
              x ← p[p[x]]
        else if (x=right[p[x]])
          then x ← p[x]
              left-rotate(T, x)
              color[p[x]] ← Black
              color [p[px]]) ← red
              right-rotate (T,p[p[x]])
    else (right←→left)
  Color[root[t]] ← black
```