

# Strutture dati - Parte 2

Dipartimento di Elettronica, Informazione e Bioingegneria  
Politecnico di Milano

27 maggio 2021

## Una struttura dati flessibile

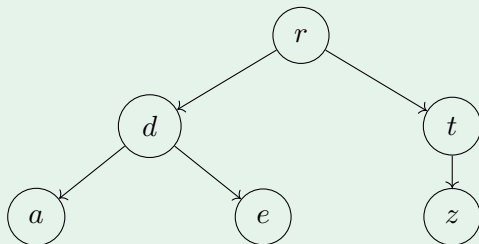
- Una struttura dati versatile è il cosiddetto *albero*
- Ne abbiamo già fatto uso informalmente (limite inferiore complessità dell'ordinamento per confronto)
- In estrema sintesi:
  - Un albero è costituito da un insieme di nodi e uno di archi che li collegano
  - Ogni nodo può avere al più un arco entrante, ma un numero arbitrario di archi uscenti
- Vedremo come gli alberi sono una rappresentazione efficiente per insiemi di dati ordinati

# Alberi

## Definizione

Un albero  $A$  è una coppia  $(V, E)$  dove  $V$  è un insieme di nodi e  $E$  un insieme di archi (coppie di nodi ordinate). Ogni nodo può apparire un'unica volta come destinazione di un arco (=secondo elemento della coppia). Non sono possibili cicli.

## Graficamente



- $V = \{r, a, d, e, t, z\}$
- $E = \{(r, d), (r, t), (d, a), (d, e), (t, z)\}$

## Nomenclatura

- **Radice:** É l'unico nodo dell'albero privo di un arco entrante
- **Foglia:** Un nodo senza archi uscenti
- **Padre** (o genitore): di un nodo  $n$ : il nodo da cui l'arco entrante in  $n$  ha origine
- **Figlio** (o discendente): di un nodo  $n$ : il nodo in cui uno degli archi uscenti da  $n$  termina
- Un albero in cui ogni nodo ha al più due figli è detto *albero binario*
- **Livello:** distanza, in numero di archi, di un nodo dalla radice
- **Albero completo:** un albero a cui non è possibile aggiungere un nodo con livello minore o uguale a quello dei presenti

# Alberi binari

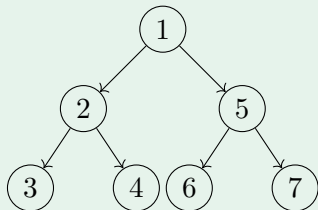
## In pseudocodice

- Ci occuperemo di alberi binari (= ogni padre ha 2 figli)
- É utile dare una definizione ricorsiva di albero:
  - Un albero è formato da un nodo radice a cui sono collegati due alberi, il sottoalbero destro e quello sinistro
  - Un albero può essere vuoto (NIL)
- Le azioni sull'albero indicizzeranno i nodi con una chiave
  - Simile a quanto fatto per le tabelle hash, la chiave è un intero
- Dato un nodo  $A$ :
  - $A.left$  è il riferimento al figlio sinistro,  $A.right$  al destro
  - $A.p$  è il riferimento al padre,  $A.key$  è la chiave
- Ogni albero  $A$  ha un riferimento  $A.root$  alla radice
  - $A.p$  è NIL solo per la radice

# Stoccaggio con struttura dati implicita

## Alberi binari stoccati con un vettore

- Un albero può essere materializzato in memoria naturalmente con una struttura basata su puntatori
- Alternativamente è possibile utilizzare un vettore per contenere le chiavi (efficiente se l'albero è completo)



1	2	5	3	4	6	7
---	---	---	---	---	---	---

- La radice dell'albero è stoccata in posizione 0 nel vettore
- Dato un nodo contenuto in posizione  $i$  il suo figlio sx è in posizione  $2i + 1$ , il dx in  $2i + 2$
- Il padre del nodo stoccato in posizione  $i$  (se esiste) si trova in posizione  $\lfloor \frac{i-1}{2} \rfloor$

# Alberi binari

## Visita di un albero

- Su di un albero è possibile effettuare operazioni di inserimento, ricerca e cancellazione di nodi come sulle altre strutture dati
- L'operazione caratteristica degli alberi è il cosiddetto *attraversamento* o *visita* per enumerare le chiavi contenute
- La definizione naturale degli algoritmi di visita è ricorsiva
- Il fattore discriminante tra le diverse strategie è l'*ordine* in cui i nodi vengono visitati

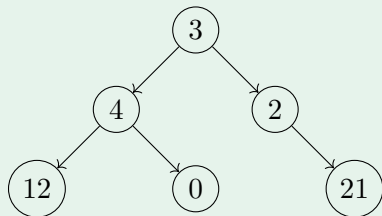
# Visita di un albero

## Visita *in-ordine* (*in-order*)

- Nella visita in ordine si visita prima il sottoalbero sx, quindi la radice, infine il sottoalbero dx

INORDER(*T*)

```
1 INORDER(T.left)  
2 PRINT(T.key)  
3 INORDER(T.right)  
4 return
```



- INORDER dell'esempio stampa: 12, 4, 0, 3, 2, 21
- Complessità:  $\Theta(n)$ , tocca una sola volta ogni nodo



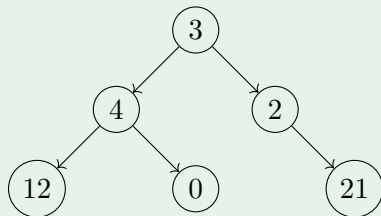
# Visita di un albero

## Visita *anticipata* (*preorder*)

- Nella visita in ordine si visita prima la radice, quindi il sottoalbero sx, infine il sottoalbero dx

PREORDER(*T*)

```
1 PRINT(T.key)  
2 PREORDER(T.left)  
3 PREORDER(T.right)  
4 return
```



- PREORDER dell'esempio stampa: 3, 4, 12, 0, 2, 21
- Complessità:  $\Theta(n)$ , tocca una sola volta ogni nodo

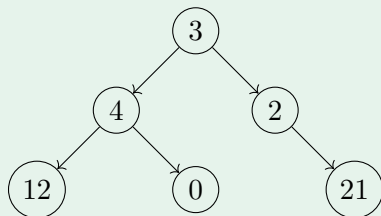
# Visita di un albero

## Visita *posticipata* (*postorder*)

- Nella visita in ordine si visita prima il sottoalbero sx, poi il sottoalbero dx e infine la radice

POSTORDER( $T$ )

```
1 POSTORDER( $T.left$ )  
2 POSTORDER( $T.right$ )  
3 PRINT( $T.key$ )  
4 return
```



- POSTORDER dell'esempio stampa: 12, 0, 4, 21, 2, 3
- Complessità:  $\Theta(n)$ , tocca una sola volta ogni nodo

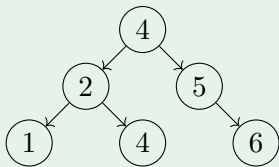
# Alberi binari di ricerca (BST)

## Definizione

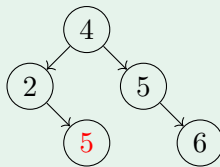
- Uno degli usi più comuni degli alberi binari è utilizzare quelli per cui è valida una data relazione tra le chiavi
- Un albero binario è detto *albero binario di ricerca* se per un qualunque suo nodo  $x$  valgono:
  - Se  $y$  è contenuto nel sottoalbero sinistro di  $x$ ,  $y.key \leq x.key$
  - Se  $y$  è contenuto nel sottoalbero destro di  $x$ ,  $y.key \geq x.key$
- Inserimenti e cancellazioni devono preservare la proprietà
- Una visita in-ordine del BST stampa le chiavi in ordine

# Alberi binari di ricerca (BST)

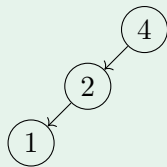
## Esempi



É un BST



NON É un BST



É un BST

## Nota

- Cambiare la condizione sui valori di chiave rimuovendo la possibilità che siano uguali rende gli elementi del BST unici

# Operazioni su BST

## Ricerca

- La struttura dei BST li rende naturali candidati per una ricerca efficace degli elementi per chiave

RICERCA( $T, x$ )

```
1  if  $T = \text{NIL}$  or  $T.\text{key} = x.\text{key}$ 
2      return  $T$ 
3  if  $T.\text{key} < x.\text{key}$ 
4      return RICERCA( $T.\text{right}, x$ )
5  else return RICERCA( $T.\text{left}, x$ )
6
```

- Complessità:  $\mathcal{O}(h)$  con  $h$  l'altezza dell'albero
- Nel caso ottimo (albero "ben bilanciato") diventa  $\mathcal{O}(\log(n))$
- Nel caso pessimo (albero degenerare in lista) è  $\mathcal{O}(n)$

# Operazioni su BST

## Minimo e massimo

- L'elemento con chiave minima (massima) è quello più a sinistra (destra) del BST

MIN( $T$ )

```
1   $cur \leftarrow T$   
2  while  $cur.left \neq \text{NIL}$   
3       $cur \leftarrow cur.left$   
4  return  $cur$ 
```

MAX( $T$ )

```
1   $cur \leftarrow T$   
2  while  $cur.right \neq \text{NIL}$   
3       $cur \leftarrow cur.right$   
4  return  $cur$ 
```

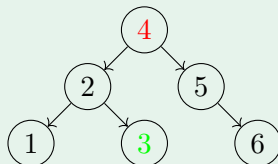
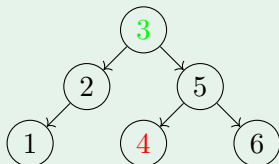
- Complessità:  $\mathcal{O}(h)$  con  $h$  l'altezza dell'albero

# Operazioni su BST

## Successore

- Il successore di un elemento  $x$  è l'elemento  $y$  con la più piccola chiave  $y.key > x.key$  presente nel BST
- Nel cercarlo sono possibili due casi:
  - 1 Il sottoalbero dx di  $x$  non è vuoto: il successore è il minimo di quel sottoalbero
  - 2 Il sottoalbero dx di  $x$  è vuoto: il successore è il progenitore più prossimo a  $x$  per cui  $x$  appare nel suo sottoalbero sx

## Esempi: Successore di 3



# Operazioni su BST

## Successore

- Lo pseudocodice per la ricerca del successore è il seguente:

SUCCESSORE( $x$ )

```
1  if  $x.right \neq \text{NIL}$ 
2      return MIN( $x.right$ )
3   $y \leftarrow x.p$ 
4  while  $y \neq \text{NIL}$  and  $y.right = x$ 
5       $x \leftarrow y$ 
6       $y \leftarrow y.p$ 
7  return  $y$ 
```

- Complessità nel caso 1:  
la stessa del calcolo del minimo:  $\mathcal{O}(h)$
- Complessità nel caso 2:  
caso pessimo,  $x$  è la foglia più distante dalla radice,  $\mathcal{O}(h)$



# Operazioni su BST

## Inserimento – Struttura

- L'inserimento di un nuovo elemento deve rispettare la proprietà fondamentale del BST
- Assunzione: non vogliamo che il BST contenga duplicati
- Idea: Cerco l'elemento che voglio inserire nel BST, non lo trovo, lo inserisco al posto del NIL trovato.
- Unica accortezza rispetto al codice della ricerca: tenere traccia dell'ultimo nodo non-NIL per poter inserire correttamente l'elemento

# Operazioni su BST

## Inserimento – Pseudocodice

INSERISCI( $T, x$ )

```
1   $pre \leftarrow \text{NIL}$ 
2   $cur \leftarrow T.root$ 
3  while  $cur \neq \text{NIL}$ 
4       $pre \leftarrow cur$ 
5      if  $x.key < cur.key$ 
6           $cur \leftarrow cur.left$ 
7      else  $cur \leftarrow cur.right$ 
8   $x.p \leftarrow pre$ 
9  if  $pre = \text{NIL}$ 
10      $T.root \leftarrow x$ 
11 elseif  $x.key < pre.key$ 
12      $pre.left \leftarrow x$ 
13 else  $pre.right \leftarrow x$ 
```

- Le righe 3–7 effettuano la ricerca della posizione di inserimento nell'albero
- Le righe 8–13 effettuano l'inserimento vero e proprio
- Complessità: la stessa della ricerca  $\mathcal{O}(h)$  più una porzione a tempo costante (inserimento)

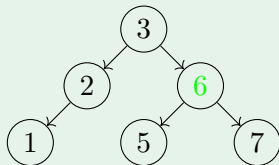
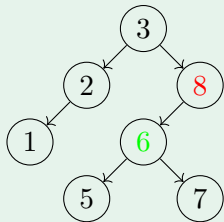
# Operazioni su BST

## Cancellazione – Struttura – 1

- La strategia di cancellazione di un elemento da un BST dipende dal numero di figli dell'elemento in questione

**Caso 1** L'elemento non ha figli: è sufficiente eliminarlo dall'albero deallocandolo e impostando il puntatore del padre a NIL

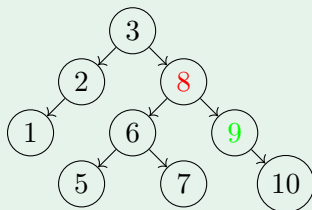
**Caso 2** L'elemento ha un figlio: L'elemento viene sostituito dal figlio nel suo ruolo nell'albero



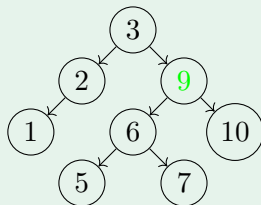
# Operazioni su BST

## Cancellazione – Struttura – 2

- **Caso 3** L'elemento ha due figli: copio il valore del suo successore su di esso ed elimino il successore
- Il successore  $s$  di un elemento con due figli  $x$  non ha mai il figlio  $sx$   $f$ : si avrebbe  $s.key < f.key < x.key$ , ma questo è impossibile per definizione di successore



Prima



Dopo

# Operazioni su BST

## Cancellazione – Pseudocodice

CANCELLA( $T, x$ )

```
1  if  $x.left = \text{NIL}$  or  $x.right = \text{NIL}$ 
2       $da\_canc \leftarrow x$ 
3  else  $da\_canc \leftarrow \text{SUCCESORE}(x)$ 
4  if  $da\_canc.left \neq \text{NIL}$ 
5       $sottoa \leftarrow da\_canc.left$ 
6  else  $sottoa \leftarrow da\_canc.right$ 
7  if  $sottoa \neq \text{NIL}$ 
8       $sottoa.p \leftarrow da\_canc.p$ 
9  if  $da\_canc.p = \text{NIL}$ 
10      $T.root \leftarrow sottoa$ 
11 elseif  $da\_canc = da\_canc.p.left$ 
12      $da\_canc.p.left \leftarrow sottoa$ 
13 else  $da\_canc.p.right \leftarrow sottoa$ 
14 if  $da\_canc \neq x$ 
15      $x.key \leftarrow da\_canc.key$ 
16 FREE( $da\_canc$ )
```

- Le righe 1–3 individuano il nodo da cancellare
- Le righe 4–8 individuano il sottoalbero da spostare e correggono il riferimento al padre
- Le righe 9–13 correggono il riferimento del padre
- Le righe 14–15 copiano il valore della chiave

# Analisi di complessità

## Sommario

- Tutte le operazioni sono  $\mathcal{O}(h)$  con  $h$  l'altezza del BST
- Nel migliore dei casi  $h = \log(n)$  (albero completo o quasi completo), nel peggiore  $h = n$  (lista)
- É critico per avere buone prestazioni mantenere il BST il più possibile vicino al caso ottimo
- Si può dimostrare che l'altezza attesa di un BST è  $\mathcal{O}(\log(n))$  se le chiavi inserite hanno distribuzione uniforme
- Volendo un metodo deterministico ci serve una definizione di albero *ben bilanciato*

# Albero bilanciato

## Vicinanza all'albero completo

- Intuitivamente, vogliamo che la distanza delle foglie dalla radice sia limitata superiormente, per tutte le foglie
- Una definizione operativa (Adelson-Velskii e Landis, 1962)
  - Un albero è bilanciato se, per ogni nodo, le altezze dei due sottoalberi differiscono al più di 1
- Adelson-Velskii e Landis proposero, insieme alla definizione, una modifica ai BST ed ai metodi per accedervi in grado di tenerli bilanciati (alberi AVL)
- Vediamo un'ottimizzazione degli alberi AVL che sacrifica parte del bilanciamento per ottenere inserimenti/cancellazioni più efficienti: gli alberi rosso-neri (red-black trees, RB-trees)
  - Sia la perdita in bilanciamento, che il guadagno sono costanti

# Alberi rosso-neri

## Struttura e definizione

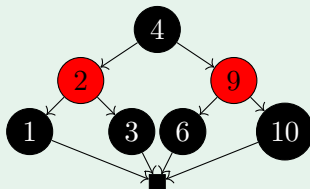
- Un albero rosso-nero è un BST i cui nodi sono dotati di un attributo aggiuntivo, detto *colore*  $\in \{\text{rosso}, \text{nero}\}$ , e soddisfacente le seguenti 5 proprietà:
  - ❶ Ogni nodo è rosso o nero
  - ❷ La radice è nera
  - ❸ Le foglie sono nere
  - ❹ I figli di un nodo rosso sono entrambi neri
  - ❺ Per ogni nodo dell'albero, tutti i cammini dai suoi discendenti alle foglie contenute nei suoi sottoalberi hanno lo stesso numero di nodi neri
- Chiamiamo, per comodità, altezza nera (black height) di un nodo  $x$  il valore  $bh(x)$  pari al numero di nodi neri, escluso  $x$  se è il caso, nel percorso che va da  $x$  alle foglie



# Alberi rosso-neri

## Convenzioni

- I dati sono mantenuti unicamente nei nodi interni, le foglie sono tutte NIL
- Per semplicità, tutte le foglie sono fisicamente rappresentate da un singolo nodo, il cui unico riferimento `T.nil` è conservato nella struttura dati
- Il padre del nodo radice punta anch'esso a `T.nil`



# Alberi rosso-neri

## Azioni sugli alberi rosso-neri

- Tutte le operazioni che non vanno a modificare la struttura dell'albero sono identiche ai BST:  
RICERCA, MIN, MAX, SUCCESSORE, PREDECESSORE
- Le operazioni di INSERISCI e CANCELLA hanno necessità di mantenere le proprietà degli alberi rosso-neri
  - Idea di massima: opero come se si trattasse di un BST generico, dopodiché compenso le eventuali violazioni
- É necessario essere in grado di ri-bilanciare l' albero con modifiche solamente locali (no ricostruzione dell' albero)

# Alberi rosso-neri

## Teorema (Proprietà di buon bilanciamento)

*Un albero RB con  $n$  nodi interni ha altezza massima  $2 \log(n + 1)$*

## Dimostrazione - 1

- Dim. che un sottoalbero con radice  $x$  ha almeno  $2^{bh(x)} - 1$  nodi interni, per induzione sull'altezza del sottoalbero
  - Caso base (altezza 0):  $x$  è una foglia, il sottoalbero contiene almeno  $2^{bh(x)} - 1 = 2^0 - 1 = 0$  nodi interni
  - Passo: dato  $x$ , entrambi i suoi figli hanno altezza nera  $bh(x)$  o  $bh(x) - 1$ . Dato che l'altezza dei figli è minore di quella di  $x$  (per hp. ind.) i loro sottoalberi hanno almeno  $2^{bh(x)-1} - 1$  nodi interni. L'albero radicato in  $x$  contiene quindi almeno  $2^{bh(x)-1} - 1 + 2^{bh(x)-1} - 1 + 2 = 2^{bh(x)}$  nodi

# Alberi rosso-neri

## Dimostrazione.

- Per la proprietà 4, almeno metà dei nodi su un qualunque percorso radice-foglia sono neri
- L'altezza nera della radice è almeno  $\frac{h}{2}$ ; per quanto detto prima il sottoalbero radicato in essa contiene almeno  $2^{\frac{h}{2}} - 1$  nodi
- $n \geq 2^{\frac{h}{2}} - 1$ , risolvendo per  $h \rightarrow h \leq 2 \log(n + 1)$  ■

## Conseguenze

- Le operazioni che restano invariate tra RB-trees e BST (RICERCA, MAX, MIN, SUCCESSORE) sono  $\mathcal{O}(2 \log(n + 1))$
- Se riesco a riparare le violazioni in maniera efficiente ho anche INSERISCI e CANCELLA in  $\mathcal{O}(2 \log(n + 1))$

# Alberi rosso-neri

## Rotazione

Operazione locale a due nodi di un BST che cambia il livello a cui sono situati due nodi senza violare la proprietà BST

LEFTROTATE( $T, x$ )  $\rightarrow$

$$l(\rho) = i - 1$$

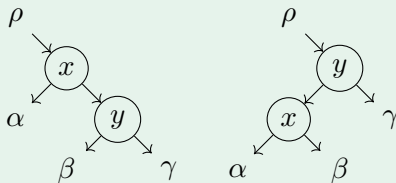
$$l(x) = i$$

$$l(\alpha) = i + 1$$

$$l(y) = i + 1$$

$$l(\beta) = i + 2$$

$$l(\gamma) = i + 2$$



$$l(\rho) = i - 1$$

$$l(x) = i + 1$$

$$l(\alpha) = i + 2$$

$$l(y) = i$$

$$l(\beta) = i + 2$$

$$l(\gamma) = i + 1$$

$\leftarrow$  RIGHTROTATE( $T, y$ )

# Rotazioni

## Pseudocodice

LEFTROTATE( $T, x$ )

```
1   $y \leftarrow x.right$  // Calcola  $y$ 
2   $x.right \leftarrow y.left$  // Sposta il sottoalbero  $\beta$ 
3  if  $y.left \neq T.nil$  // Sistema il riferimento a  $p$  della radice di  $\beta$ 
4       $y.left.p \leftarrow x$ 
5   $y.p \leftarrow x.p$  // Sistema il riferimento al padre del nodo  $y$ 
6  if  $x.p = T.nil$  // Sistema il riferimento al figlio del padre di  $y$ 
7       $T.root \leftarrow y$ 
8  elseif  $x = x.p.left$ 
9       $x.p.left \leftarrow y$ 
10 else  $x.p.right \leftarrow y$ 
11  $y.left \leftarrow x$  // Aggancia  $x$  a sinistra di  $y$ 
12  $x.p \leftarrow y$ 
```

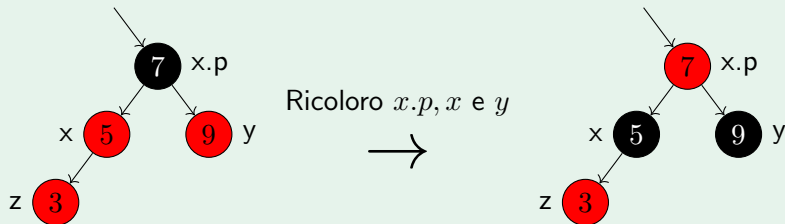
# Alberi rosso-neri

## Inserimento

- L'inserimento procede ad inserire il nuovo elemento come se l'albero fosse un semplice BST salvo:
  - Assegnare il valore dei sottoalberi del nodo a  $T.nil$  al posto di  $NIL$  se viene inserito come una foglia
  - Assegnare il valore del genitore del nodo a  $T.nil$  al posto di  $NIL$  se il nodo è inserito come radice
  - Colorare il nodo appena inserito di rosso
- Possono essere violate la proprietà 4 (i figli di un nodo rosso sono entrambi neri) e la 2 (la radice è nera)
- RIPARARBINISCI( $z$ ), dato il nodo inserito  $z$ :
  - Con  $z.p$  figlio sx del nonno di  $z$ : 3 casi a seconda del colore dello "zio" e della posizione di  $z$  rispetto a  $z.p$
  - Simmetrica per il caso in cui il padre è figlio dx del nonno

# RIPARARBINERISCI( $z$ )

Caso 1 - lo "zio"  $y$  è rosso, posizione di  $z$  irrilevante,  $z.key = 3$

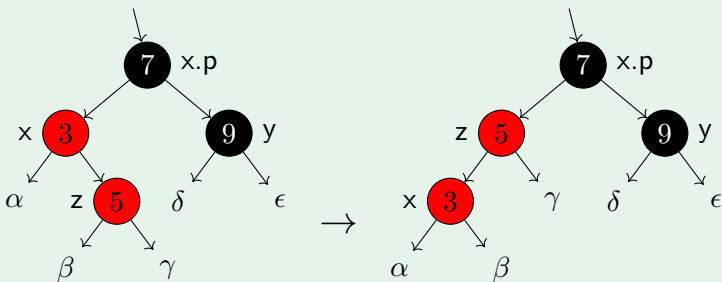


- Successivamente chiamo  $RIPARARBINERISCI(x.p)$ :  $x.p$  potrebbe avere un padre rosso
- Se  $x.p$  è la radice, posso colorarla di nero senza problemi



# RIPARARBINERISCI( $z$ )

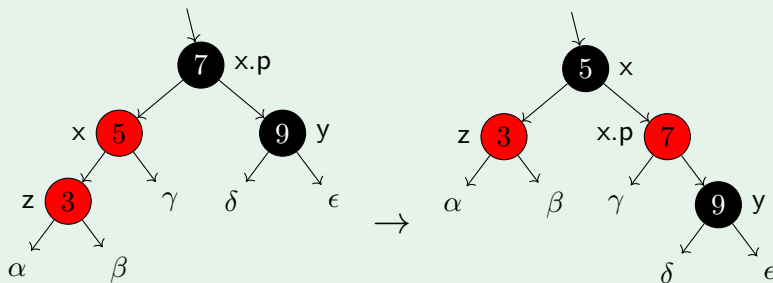
Caso 2 - lo "zio"  $y$  è nero,  $z = x.right$ ,  $z.key = 5$



- Effettuiamo  $\text{LEFTROTATE}(T, x)$ : ora la riparazione va effettuata su  $z$ , con  $z.left = x$  rosso (caso successivo)

# RIPARABINSERISCI( $z$ )

Caso 3 - lo "zio"  $y$  è nero,  $z = x.left$ ,  $z.key = 3$



- Scambiamo i colori di  $x$  e  $x.p$  ed eseguiamo  $\text{RIGHTROTATE}(T, p.x)$

# Operazioni su alberi RB

## RiparaRBInserisci – Pseudocodice

RIPARARBINISERISCI( $T, z$ )

```
1  while  $z.p.color = red$ 
2      if  $z.p \leftarrow z.p.left$ 
3           $y \leftarrow z.p.p.right$ 
4          if  $y.color = red$  // Caso 1
5               $z.p.color \leftarrow black$ 
6               $y.color \leftarrow black$ 
7               $z.p.p.color \leftarrow red$ 
8               $z \leftarrow z.p.p$ 
9          else if  $z = z.p.right$  // Caso 2, ruotare  $z$ 
10              $z \leftarrow z.p$ 
11             LEFTROTATE( $z$ )
12              $z.p.color \leftarrow black$  // Caso 3
13              $z.p.p.color \leftarrow red$ 
14             RIGHTROTATE( $z.p.p$ )
15         else // come le righe 3–14, scambiando right con left
16      $T.root.color \leftarrow black$ 
```

# Alberi rosso-neri - Inserimento

## Analisi di complessità

- Nei casi 2 e 3 la procedura  $\text{RIPARARBINISERISCI}(z)$  deve solo effettuare un cambio di colori locale e 2 o 1 rotazioni
  - Tutte queste operazioni sono  $\mathcal{O}(k)$
- Nel caso 1 continua analizzando il nonno del nodo corrente
  - Caso pessimo: il ciclo itera numero di volte pari a metà dell'altezza dell'intero albero
- L'intera riparazione prende al più  $\mathcal{O}(\log(n)) \rightarrow$  l'inserimento, comprensivo di riparazione, in alberi RB è  $\mathcal{O}(\log(n))$

# Alberi rosso-neri

## Cancellazione

- La cancellazione procede a cancellare il nuovo elemento come se l'albero fosse un semplice BST salvo:
  - l' uso di  $T.nil$  al posto di  $NIL$
  - invocare la procedura che ripara eventuali violazioni delle proprietà RB
- Nel caso sia eliminato un nodo rosso, non è necessario alcun cambiamento (non sono possibili violazioni delle proprietà)
- Se il nodo eliminato è nero,  $RIPARARBCANCELLA(x)$ , dato il nodo  $x$  presente al posto di quello cancellato  $z$  ripristina le proprietà

# Alberi rosso-neri

## Struttura di RIPARARBCANCELLA( $x$ )

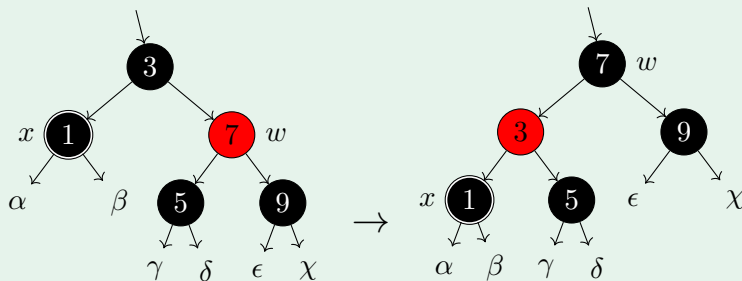
- I 5 casi della procedura RIPARARBCANCELLA( $x$ ), con  $x$  figlio sx di  $x.p$  (procedura simmetrica se figlio dx) sono:
- Caso 0:  $x$  è rosso
- Se  $x$  è nero
  - Caso 1: Il suo fratello è rosso
  - Se il suo fratello è nero
    - Caso 2:  $x$  ha entrambi i nipoti neri
    - Caso 3:  $x$  ha entrambi il nipote sinistro rosso
    - Caso 4:  $x$  ha entrambi il nipote destro rosso
- La riparazione da effettuare deve sistemare il fatto che  $x$  ha un “colore aggiuntivo oltre al proprio”: il nero che ha “ereditato” dal nodo cancellato

# Alberi rosso-neri

## Cancellazione - Caso 0: $x$ è rosso

- Viene colorato di nero: ripristina i valori di  $bh$  senza violazioni

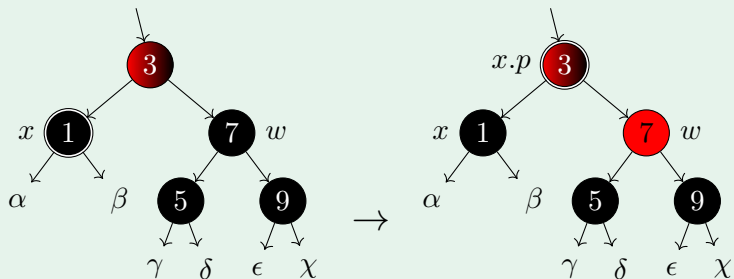
## Cancellazione - Caso 1: $x$ è nero, con fratello $w$ rosso ( $\Rightarrow x.p$ nero)



Scambio i colori di  $w$  e  $w.p$ , LEFTROTATE( $x.p$ ): ora  $x$  ha un fratello  $w$  nero: diventa uno tra i casi 2,3,4

# Alberi rosso-neri

Cancellazione - Caso 2:  $x$  è nero, con fratello  $w$  nero, e nipoti entrambi neri (n.b.  $x.p$  può essere nero o rosso)

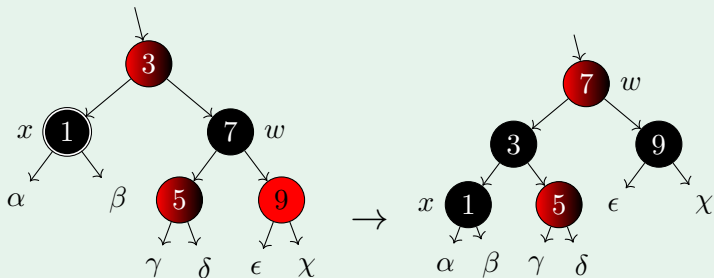


Coloro  $w$  di rosso (i.e. rimuovo un nero dal sottoalbero) e richiamo `RIPARARBCANCELLA` su  $x.p$



# Alberi rosso-neri

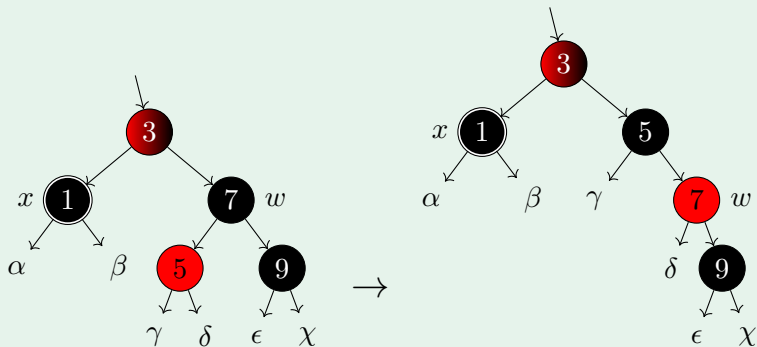
Cancellazione - Caso 4:  $x$  è nero, con fratello  $w$  nero, e nipote dx rosso (n.b.  $x.p$  e  $w.left$  possono essere neri o rossi)



$w$  prende il colore di  $w.p$ ,  $w.right$  diventa nero. Invoco LEFTROTATE su  $w.p$

# Alberi rosso-neri

Cancellazione - Caso 3:  $x$  è nero, con fratello  $w$  nero, e nipote dx nero (n.b.  $x.p$  può essere nero o rosso)



Scambio di colore  $w$  e  $w.left$ , RIGHTROTATE su  $w \rightarrow$  Caso 4

# Operazioni su alberi RB

## RIPARARBCANCELLA – Analisi di complessità

- I casi 0,1,3 e 4 di RIPARARBCANCELLA effettuano un numero costante di rotazioni e scambi di colore  $\rightarrow$  sono  $\mathcal{O}(k)$
- L'unica chiamata ricorsiva è quella che avviene nel caso 2 sul padre di  $x$ 
  - Nel caso il padre ricada nei casi 0,1,3 o 4, la nuova chiamata termina in  $\mathcal{O}(k)$
  - In caso contrario, viene ri-invocata RIPARARBCANCELLA
- Ad ogni chiamata ricorsiva si risale di un livello verso la radice  $\rightarrow$  al massimo effettuiamo  $\mathcal{O}(\log(n))$  chiamate
- La procedura complessiva di cancellazione da alberi RB è quindi  $\mathcal{O}(\log(n))$  come tutte le altre azioni