

Algoritmer og Datastrukturer 1

Gerth Stølting Brodal

Binære Søgetræer [CLRS, kapitel 12]



Abstrakt Datastruktur: Ordbog

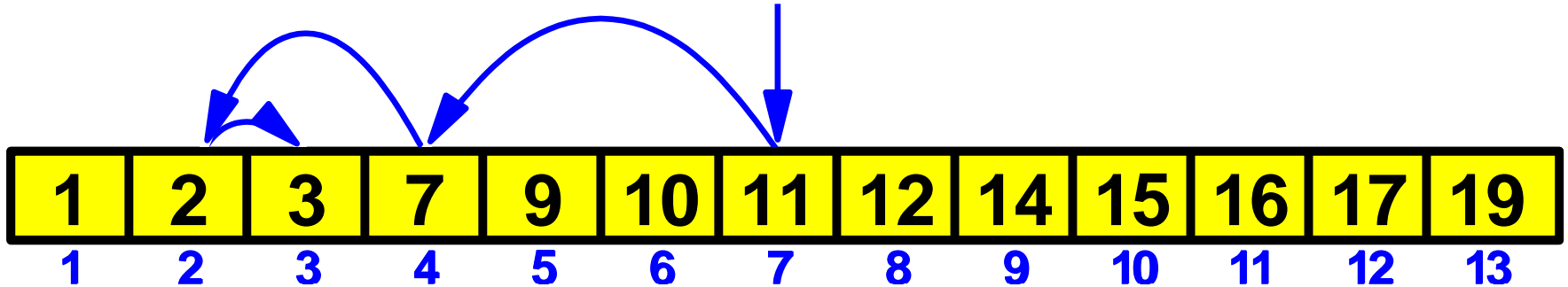
Search(S, x)

Insert(S, x)

Delete(S, x)

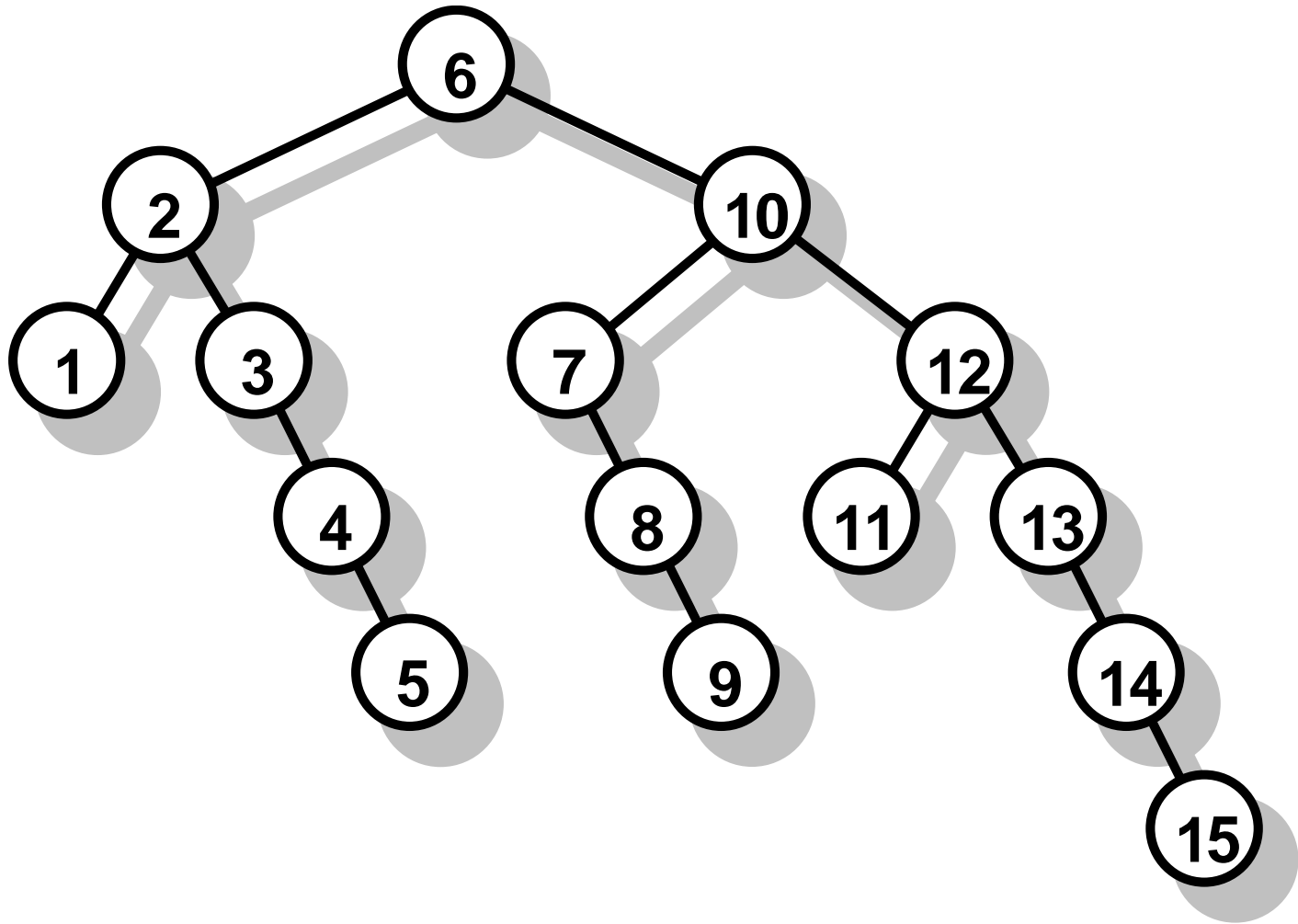
(Statisk) Ordbog : Sorteret Array

Search(4)



Search(S, x)	$O(\log n)$
Insert(S, x)	$O(n)$
Delete(S, x)	

Søgetræ



Invariant For alle knuder er elementerne i venstre (højre) undertræ mindre (større) end eller lig med elementet i knuden

Søgetræs søgninger

TREE-SEARCH(x, k)

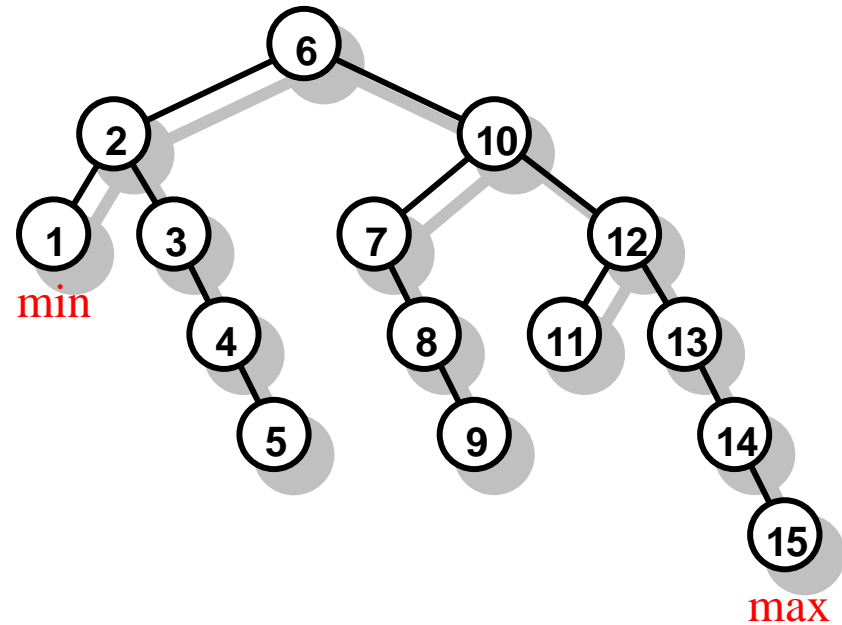
```
1  if  $x == \text{NIL}$  or  $k == x.\text{key}$ 
2      return  $x$ 
3  if  $k < x.\text{key}$ 
4      return TREE-SEARCH( $x.\text{left}, k$ )
5  else return TREE-SEARCH( $x.\text{right}, k$ )
```

ITERATIVE-TREE-SEARCH(x, k)

```
1  while  $x \neq \text{NIL}$  and  $k \neq x.\text{key}$ 
2      if  $k < x.\text{key}$ 
3           $x = x.\text{left}$ 
4      else  $x = x.\text{right}$ 
5  return  $x$ 
```

TREE-SUCCESSOR(x)

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



INORDER-TREE-WALK(x)

```
1  if  $x \neq \text{NIL}$ 
2      INORDER-TREE-WALK( $x.\text{left}$ )
3      print  $x.\text{key}$ 
4      INORDER-TREE-WALK( $x.\text{right}$ )
```

TREE-MINIMUM(x)

```
1  while  $x.\text{left} \neq \text{NIL}$ 
2       $x = x.\text{left}$ 
3  return  $x$ 
```

TREE-MAXIMUM(x)

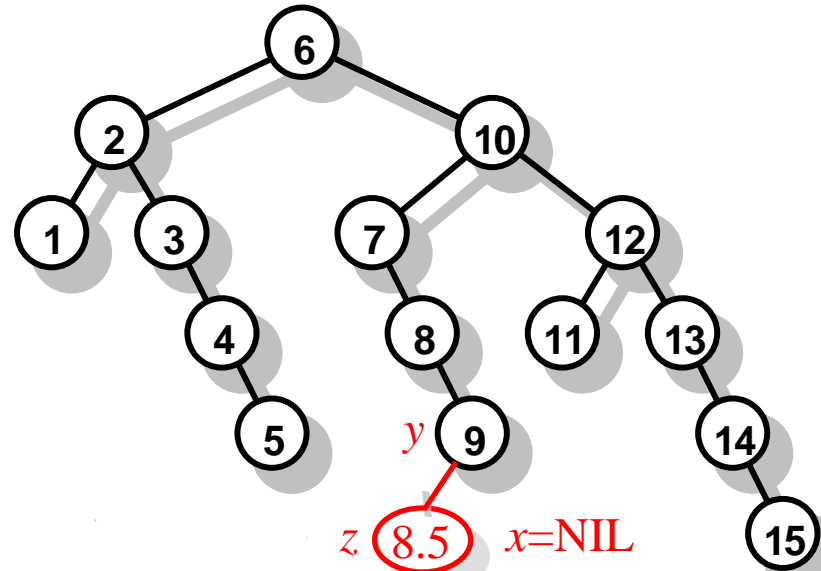
```
1  while  $x.\text{right} \neq \text{NIL}$ 
2       $x = x.\text{right}$ 
3  return  $x$ 
```

Indsættelse i Søgetræ

TREE-INSERT(T, z)

Iterativ søgning

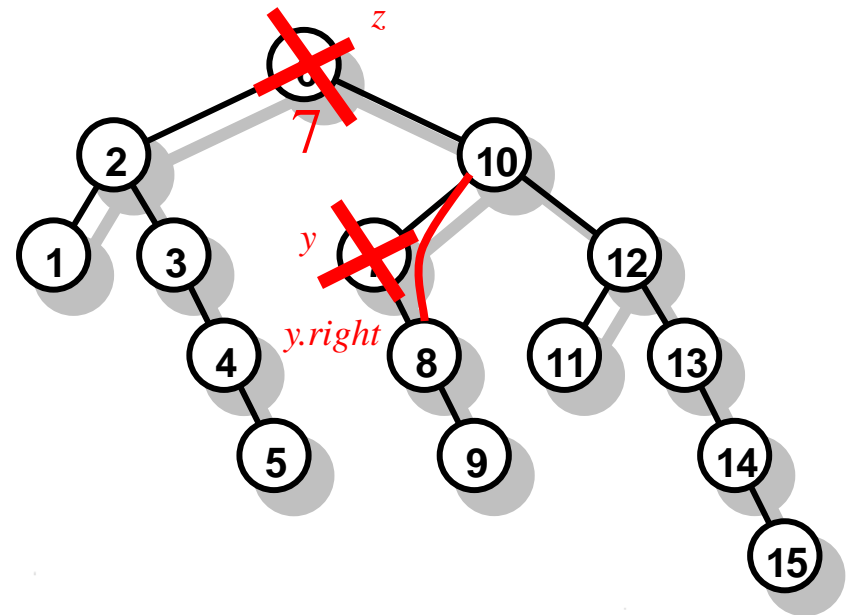
```
1   $y = \text{NIL}$ 
2   $x = T.\text{root}$ 
3  while  $x \neq \text{NIL}$ 
4       $y = x$ 
5      if  $z.\text{key} < x.\text{key}$ 
6           $x = x.\text{left}$ 
7      else  $x = x.\text{right}$ 
8   $z.p = y$ 
9  if  $y == \text{NIL}$ 
10      $T.\text{root} = z$  // tree  $T$  was empty
11  elseif  $z.\text{key} < y.\text{key}$ 
12      $y.\text{left} = z$ 
13  else  $y.\text{right} = z$ 
```



Slettelse fra Søgetræ [CLRS, Ed. 3]

TREE-DELETE(T, z)

```
1  if  $z.left == \text{NIL}$ 
2      TRANSPLANT( $T, z, z.right$ )
3  elseif  $z.right == \text{NIL}$ 
4      TRANSPLANT( $T, z, z.left$ )
5  else  $y = \text{TREE-MINIMUM}(z.right)$ 
6      if  $y.p \neq z$ 
7          TRANSPLANT( $T, y, y.right$ )
8           $y.right = z.right$ 
9           $y.right.p = y$ 
10     TRANSPLANT( $T, z, y$ )
11      $y.left = z.left$ 
12      $y.left.p = y$ 
```



TRANSPLANT(T, u, v)

```
1  if  $u.p == \text{NIL}$ 
2       $T.root = v$ 
3  elseif  $u == u.p.left$ 
4       $u.p.left = v$ 
5  else  $u.p.right = v$ 
6  if  $v \neq \text{NIL}$ 
7       $v.p = u.p$ 
```

Søgetræer

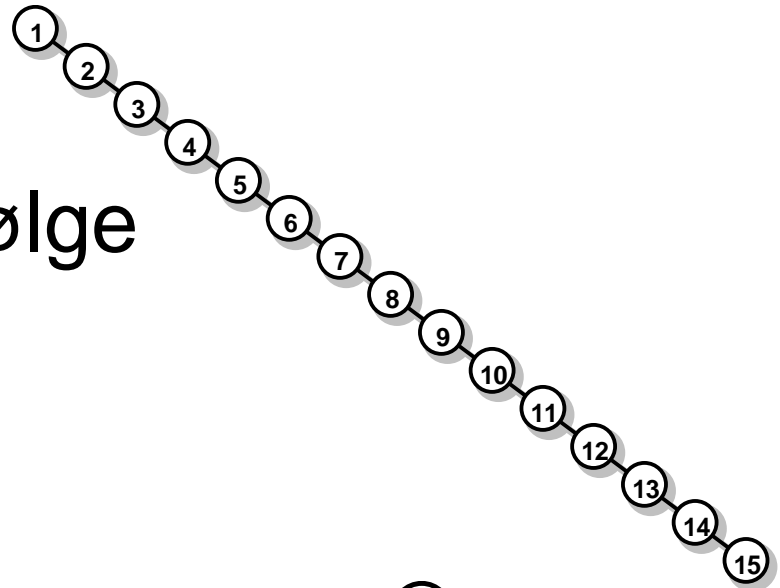
Inorder-Tree-Walk	$O(n)$
Tree-Search Iterative-Tree-Search	$O(h)$
Tree-Minimum Tree-Maximum	$O(h)$
Tree-Insert	$O(h)$
Tree-Delete	$O(h)$

h = højden af træet, n = antal elementer

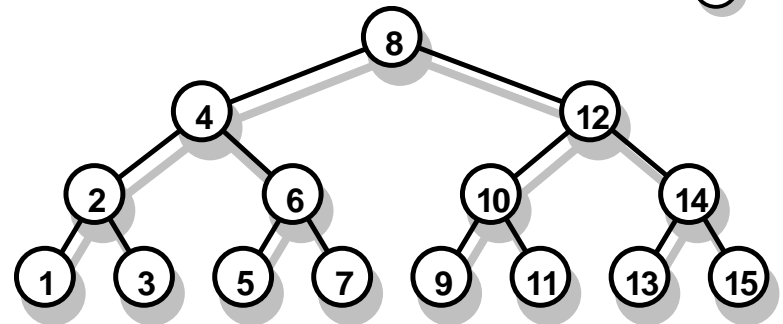
Højden af et Søgetræ ?

Største og Mindste Højde

Indsæt i stigende rækkefølge
- Højde n



Perfekt balanceret
(mindst mulig højde)
- Højde $1 + \lfloor \log n \rfloor$



Tilfældige Indsættelser i et Søgetræ

Algoritme:

Indsæt n elementer i tilfældige rækkefølge

- Ligesom ved QuickSort argumenteres at den **forventede dybde** af et element er $O(\log n)$
- Den forventede **højde af træet** er $O(\log n)$, dvs. *alle knuder* har forventet dybde $O(\log n)$ [CLRS, Kap. 12.4]

Balancerede Søgetræer

Ved **balancerede søgetræer** omstruktureres træet løbende så søgetiderne forbliver $O(\log n)$

- AVL-træer
- BB[α]-træer
- Splay træer
- Lokalt balancerede træer
- Rød-sorter træer
- Randomized trees
- (2,3)-træer
- (2,4)-træer
- B-træer
- Vægtbalancerede B-træer
- ...

Algoritmer og Datastrukturer 1

Gerth Stølting Brodal

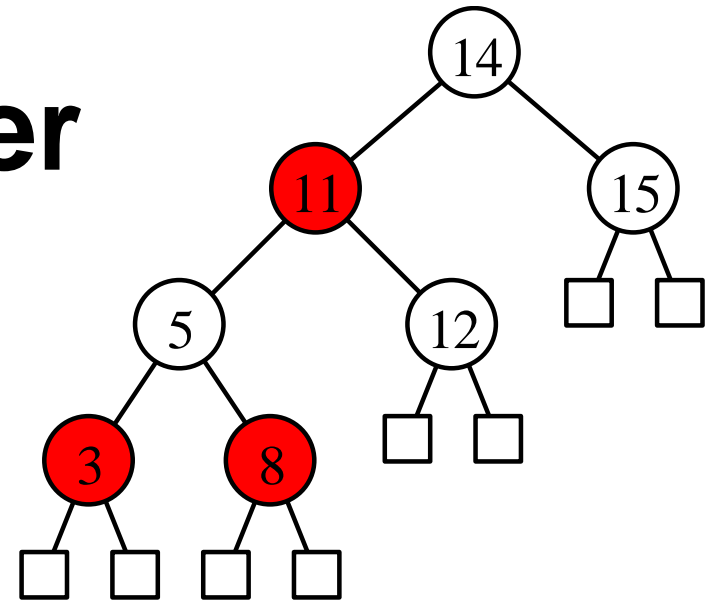
Rød-Sorte Søgetræer [CLRS, kapitel 13]



Rød-Sorte Søgetræer

Mål

Søgetræer med dybde $O(\log n)$



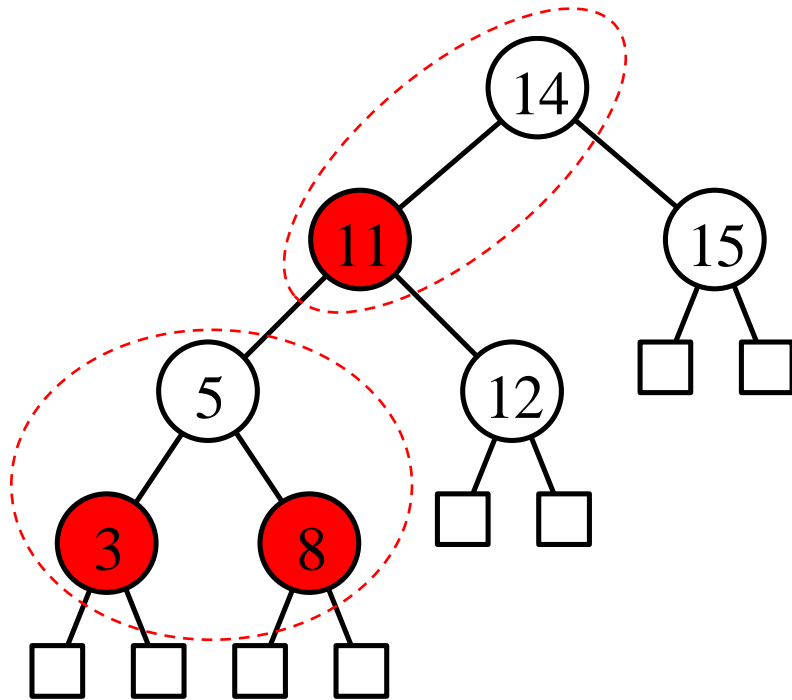
Invarianter

- Hver knude er enten **RØD** eller **SORT**
- Hver **RØD** knude har en **SORT** far
- Alle stier fra roden til et eksternt blad har **samme antal sorte knuder**

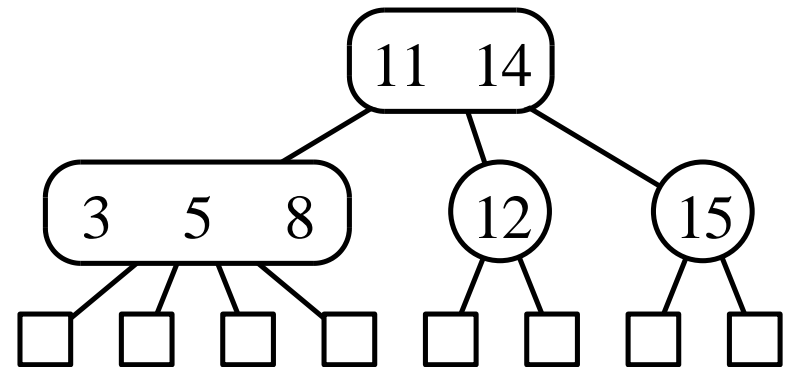
Sætning

Et rød-sort træ har højde $\leq 2 \cdot \log(n+1)$

Rød-Sorte vs (2-4)-træer



Rød-sort træ

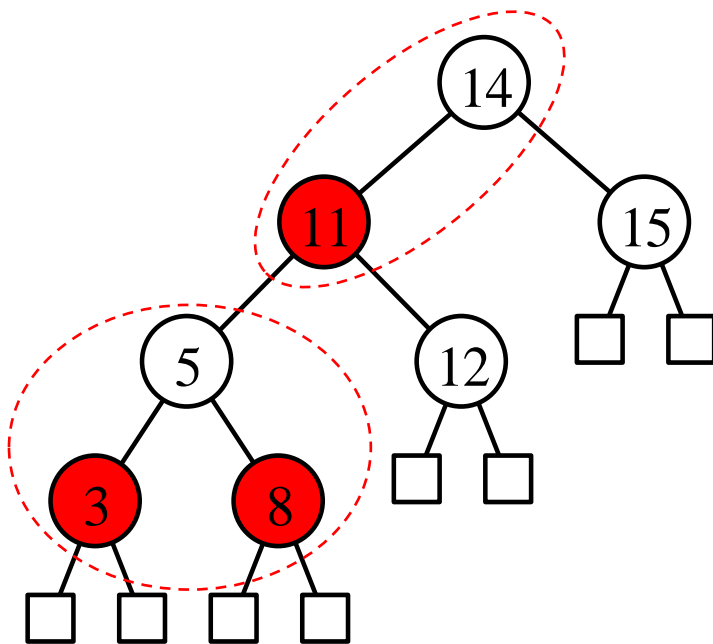


(2,4)-træ

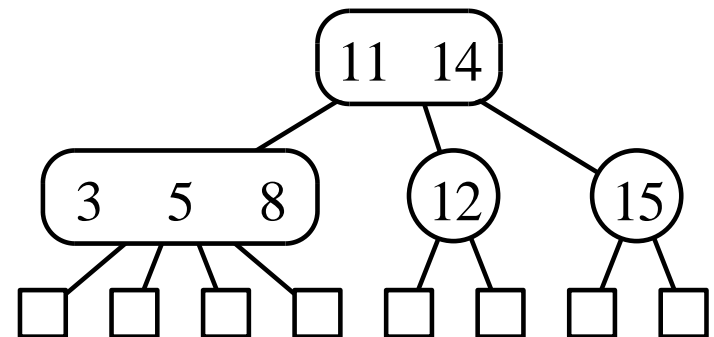
(2,4)-træ \equiv **rød-sort træ** hvor alle røde knuder er slået sammen med faderen

Egenskaber ved (2,4)-træer

- Alle interne knuder har mellem **2 og 4 børn**
- Knude med i **børn** gemmer $i-1$ **elementer**
- Stier fra roden til et eksternt blad er lige lange

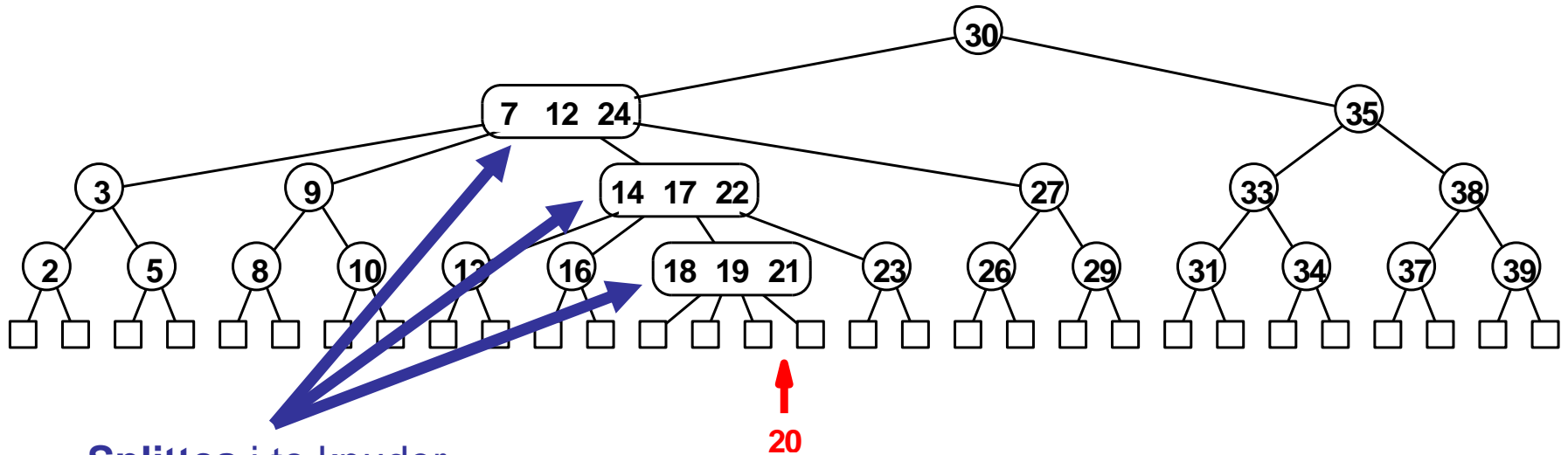


Rød-sort træ

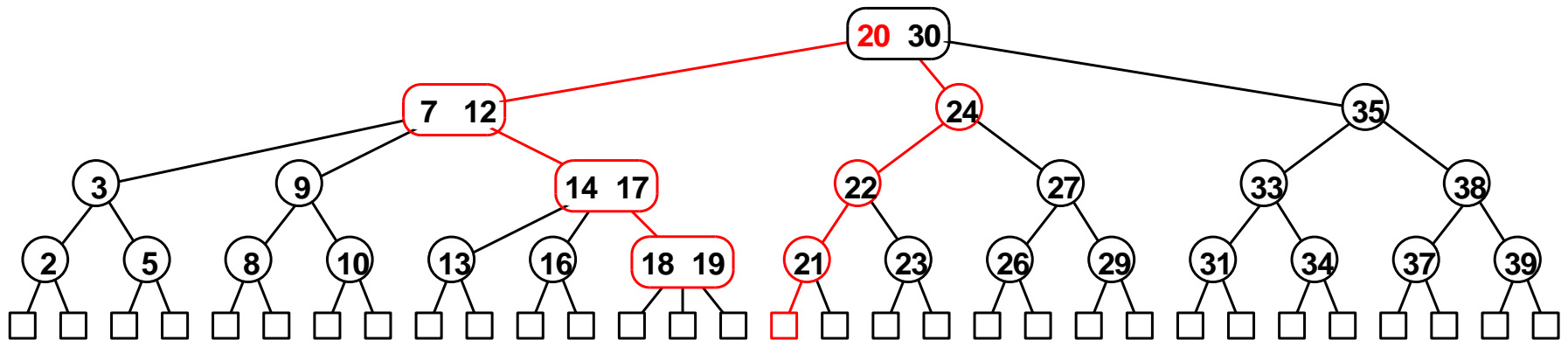


(2,4)-træ

Indsættelse i et (2,4)-træ

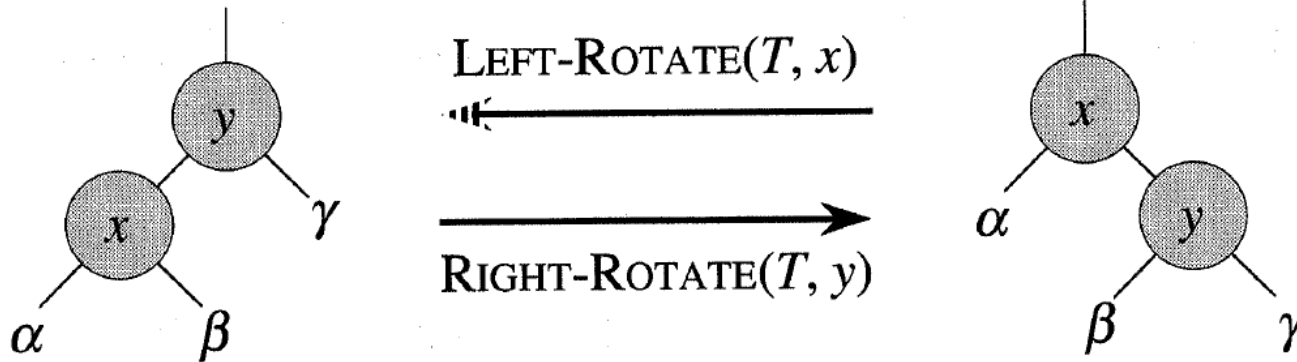


Splittes i to knuder
og midterste nøgle
flyttes et niveau op



Opdateringer af Rød-Sorte Træer

Rotationer



$\text{LEFT-ROTATE}(T, x)$

```
1   $y = x.\text{right}$            // set  $y$ 
2   $x.\text{right} = y.\text{left}$       // turn  $y$ 's left subtree into  $x$ 's right subtree
3  if  $y.\text{left} \neq T.\text{nil}$ 
4       $y.\text{left}.p = x$ 
5   $y.p = x.p$                // link  $x$ 's parent to  $y$ 
6  if  $x.p == T.\text{nil}$ 
7       $T.\text{root} = y$ 
8  elseif  $x == x.p.\text{left}$ 
9       $x.p.\text{left} = y$ 
10 else  $x.p.\text{right} = y$ 
11  $y.\text{left} = x$            // put  $x$  on  $y$ 's left
12  $x.p = y$ 
```

Insert

RB-INSERT(T, z)

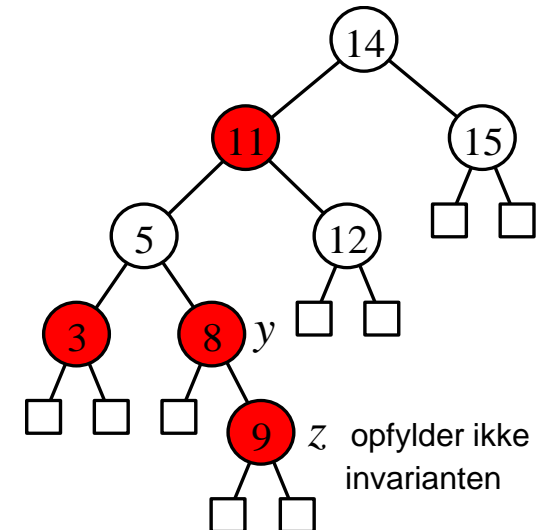
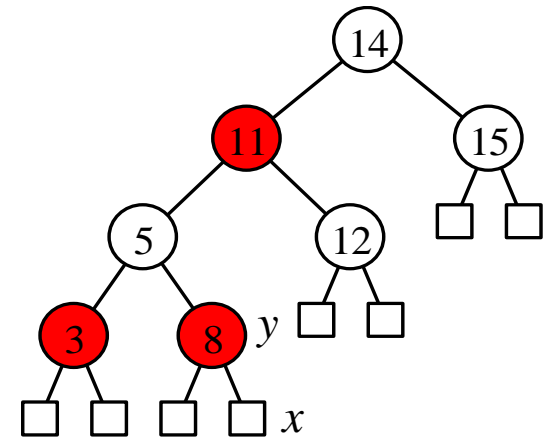
```
1   $y = T.nil$ 
2   $x = T.root$ 
3  while  $x \neq T.nil$ 
4       $y = x$ 
5      if  $z.key < x.key$ 
6           $x = x.left$ 
7      else  $x = x.right$ 
8   $z.p = y$ 
9  if  $y == T.nil$ 
10      $T.root = z$ 
11  elseif  $z.key < y.key$ 
12      $y.left = z$ 
13  else  $y.right = z$ 
14   $z.left = T.nil$ 
15   $z.right = T.nil$ 
16   $z.color = RED$ 
17  RB-INSERT-FIXUP( $T, z$ )
```

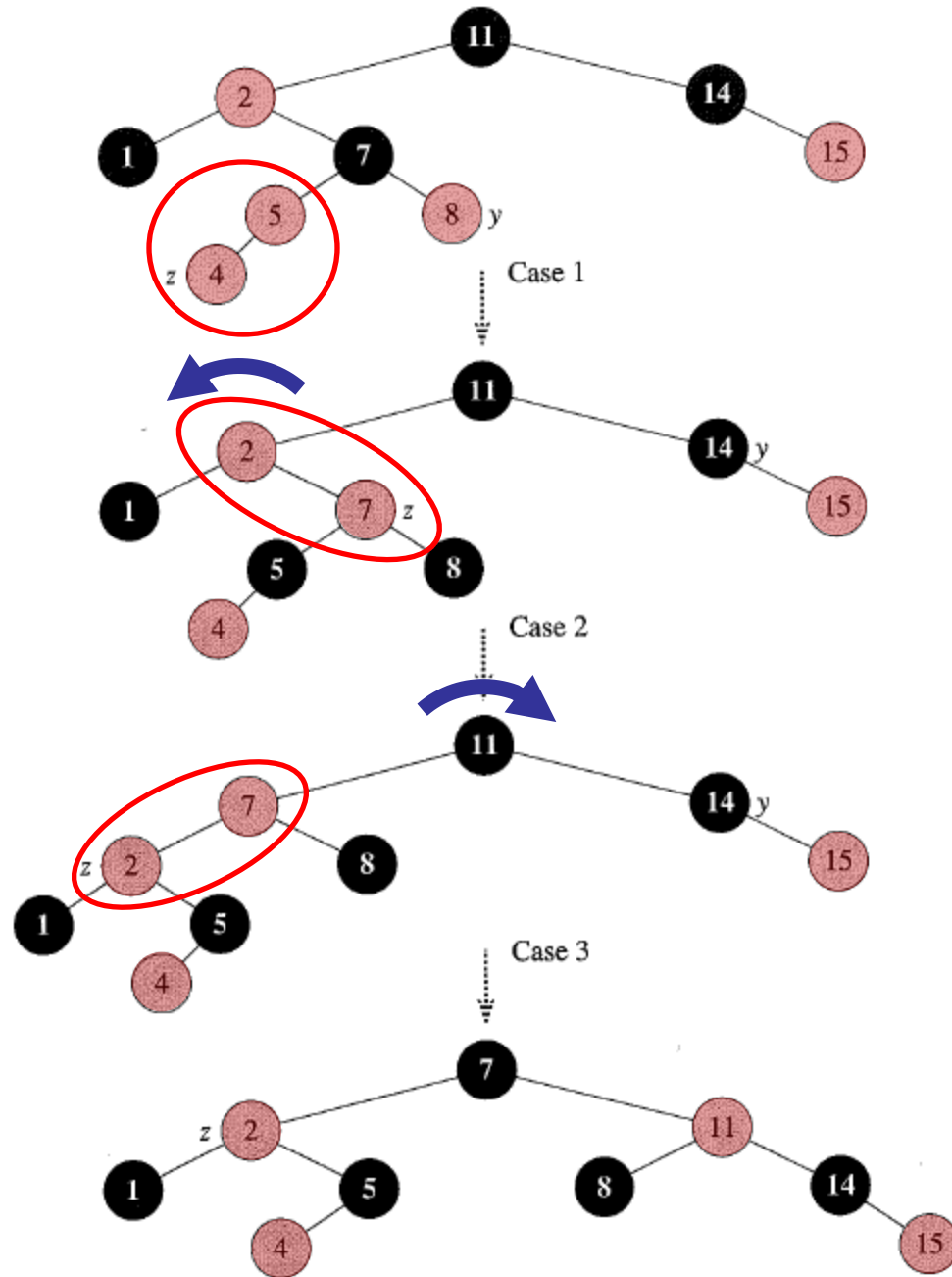
Ubalanceret
indsættelse

søg

opret z

Insert(9)

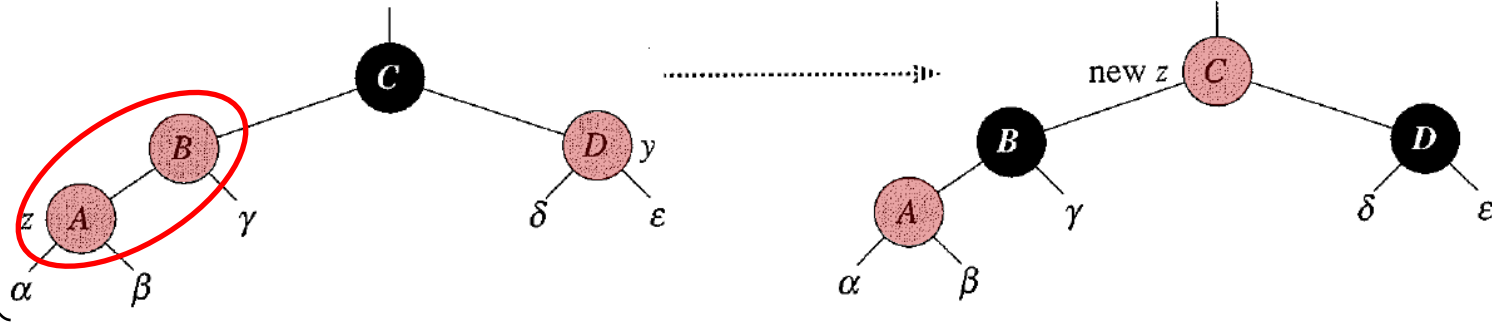
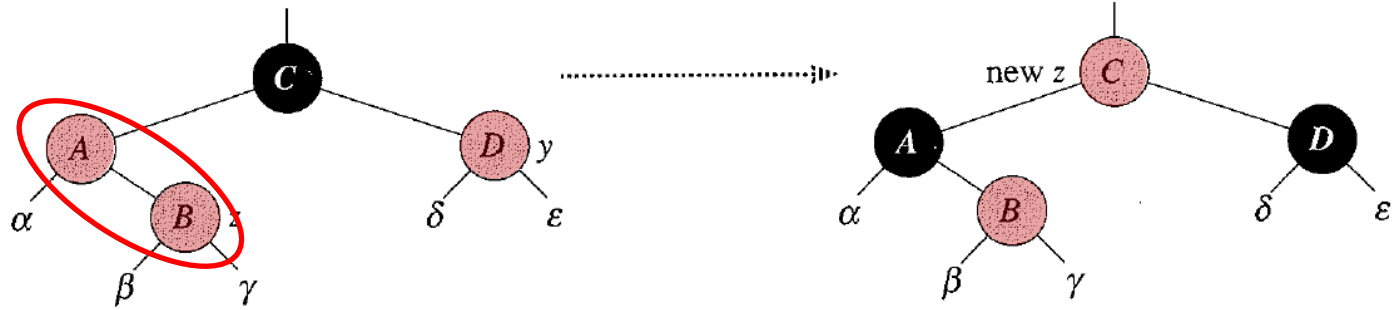




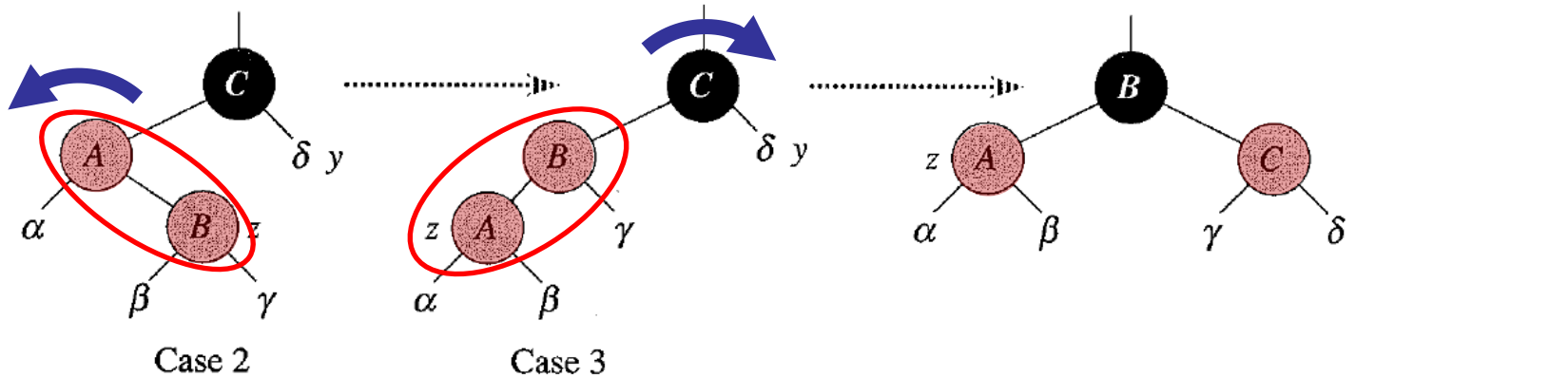
Indsættelse i rød-sort træer: rebalancering

Omfarv C
og dens to
røde børn

Case 1



Afslut

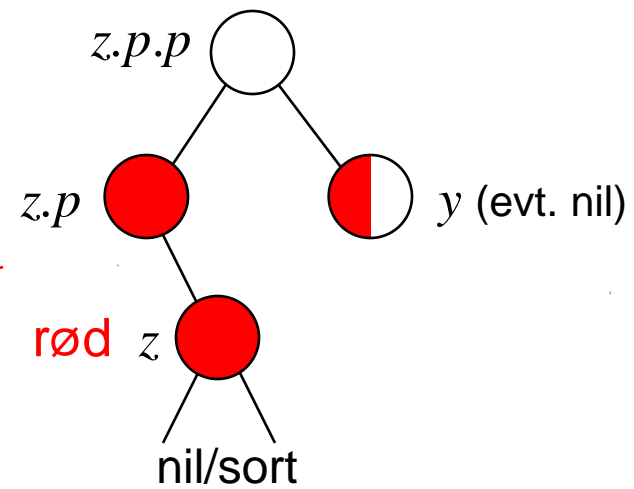


RB-INSERT-FIXUP(T, z)

```

1  while  $z.p.color == RED$ 
2      if  $z.p == z.p.p.left$ 
3           $y = z.p.p.right$ 
4          if  $y.color == RED$ 
5               $z.p.color = BLACK$  // case 1
6               $y.color = BLACK$  // case 1
7               $z.p.p.color = RED$  // case 1
8               $z = z.p.p$  // case 1
9          else if  $z == z.p.right$ 
10              $z = z.p$  // case 2
11             LEFT-ROTATE( $T, z$ ) // case 2
12              $z.p.color = BLACK$  // case 3
13              $z.p.p.color = RED$  // case 3
14             RIGHT-ROTATE( $T, z.p.p$ ) // case 3
15         else (same as then clause
                with “right” and “left” exchanged)
16      $T.root.color = BLACK$ 

```



Delete

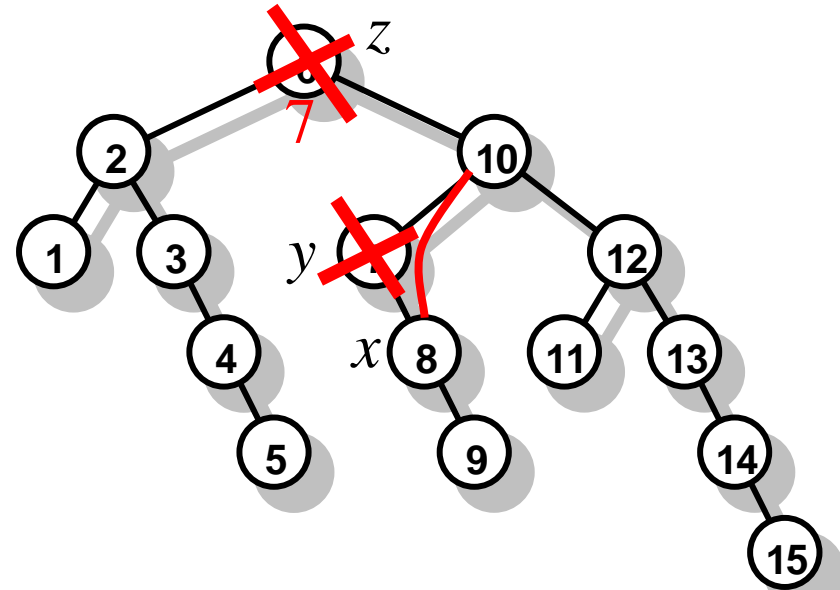
RB-DELETE(T, z)

Ubalanceret slettelse

```

1   $y = z$ 
2   $y\text{-original-color} = y.\text{color}$ 
3  if  $z.\text{left} == T.\text{nil}$ 
4       $x = z.\text{right}$ 
5      RB-TRANSPLANT( $T, z, z.\text{right}$ )
6  elseif  $z.\text{right} == T.\text{nil}$ 
7       $x = z.\text{left}$ 
8      RB-TRANSPLANT( $T, z, z.\text{left}$ )
9  else  $y = \text{TREE-MINIMUM}(z.\text{right})$ 
10      $y\text{-original-color} = y.\text{color}$ 
11      $x = y.\text{right}$ 
12     if  $y.p == z$ 
13          $x.p = y$ 
14     else RB-TRANSPLANT( $T, y, y.\text{right}$ )
15          $y.\text{right} = z.\text{right}$ 
16          $y.\text{right}.p = y$ 
17     RB-TRANSPLANT( $T, z, y$ )
18      $y.\text{left} = z.\text{left}$ 
19      $y.\text{left}.p = y$ 
20      $y.\text{color} = z.\text{color}$ 
21 if  $y\text{-original-color} == \text{BLACK}$ 
22     RB-DELETE-FIXUP( $T, x$ )

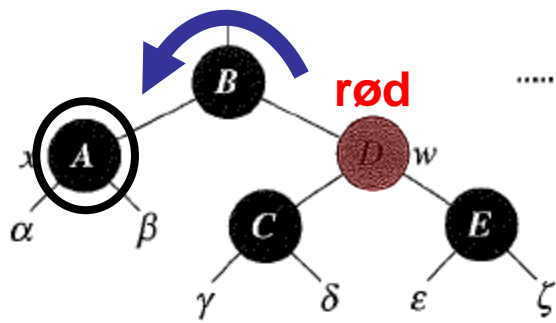
```

RB-TRANSPLANT(T, u, v)

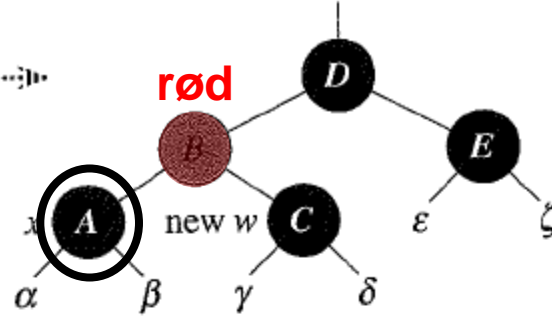
```

1  if  $u.p == T.\text{nil}$ 
2       $T.\text{root} = v$ 
3  elseif  $u == u.p.\text{left}$ 
4       $u.p.\text{left} = v$ 
5  else  $u.p.\text{right} = v$ 
6   $v.p = u.p$ 

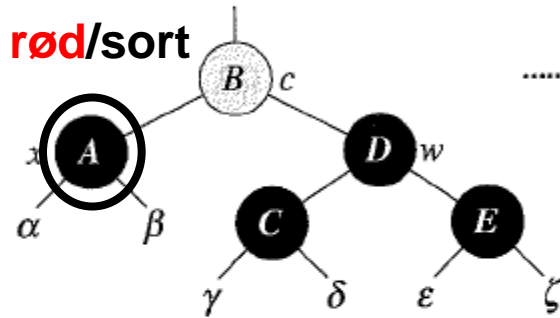
```



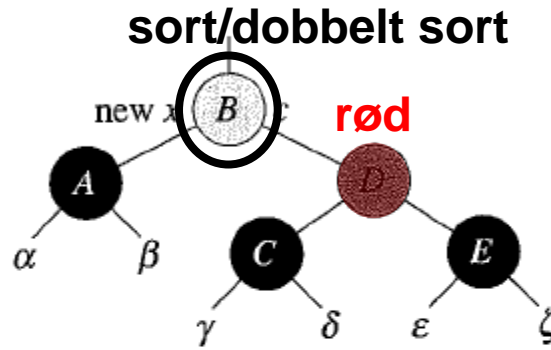
Case 1



Case 1 → 2,3,4

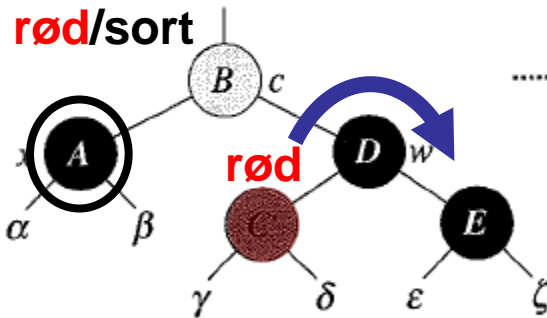


Case 2

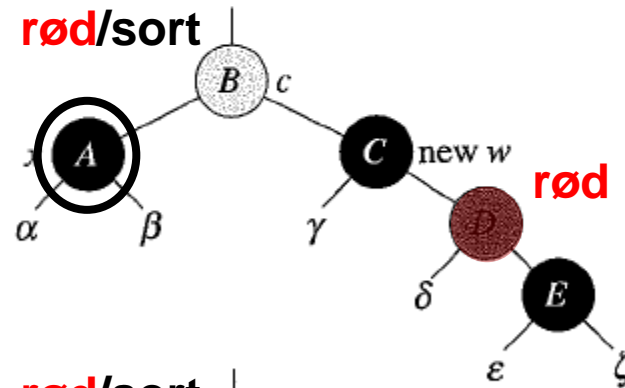


sort/dobbelt sort

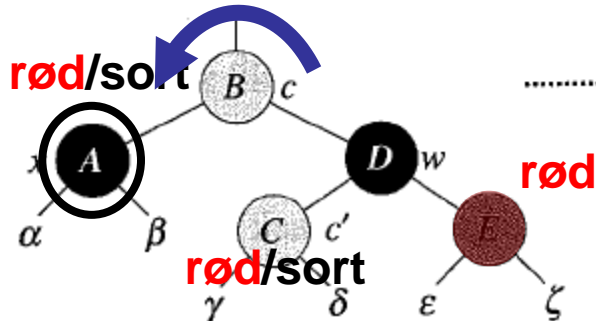
Case 2 →
Problemet x flyttet
tættere på roden
eller færdig



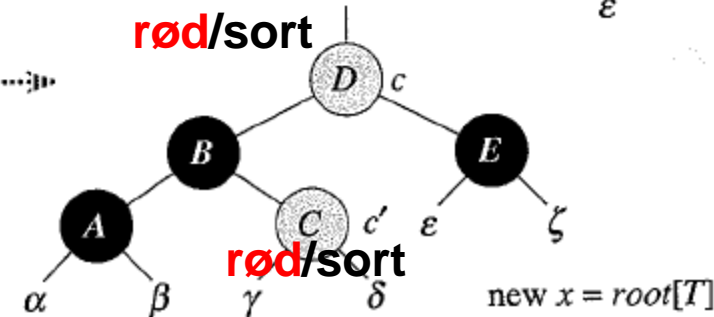
Case 3



Case 3 → 4
(og færdig)



Case 4



Case 4 → Færdig

RB-DELETE-FIXUP(T, x)

```

1  while  $x \neq T.root$  and  $x.color == BLACK$ 
2      if  $x == x.p.left$ 
3           $w = x.p.right$ 
4          if  $w.color == RED$ 
5               $w.color = BLACK$  // case 1
6               $x.p.color = RED$  // case 1
7              LEFT-ROTATE( $T, x.p$ ) // case 1
8               $w = x.p.right$  // case 1
9          if  $w.left.color == BLACK$  and  $w.right.color == BLACK$ 
10              $w.color = RED$  // case 2
11              $x = x.p$  // case 2
12         else if  $w.right.color == BLACK$ 
13              $w.left.color = BLACK$  // case 3
14              $w.color = RED$  // case 3
15             RIGHT-ROTATE( $T, w$ ) // case 3
16              $w = x.p.right$  // case 3
17              $w.color = x.p.color$  // case 4
18              $x.p.color = BLACK$  // case 4
19              $w.right.color = BLACK$  // case 4
20             LEFT-ROTATE( $T, x.p$ ) // case 4
21              $x = T.root$  // case 4
22         else (same as then clause with “right” and “left” exchanged)
23      $x.color = BLACK$ 

```

Dynamisk Ordbog :

Rød-Sorte Træer

Search(S, x)	$O(\log n)$
Insert(S, x)	
Delete(S, x)	