

# Advanced Data Structures

## Red Black Trees

# Red Black Trees

## Colored Nodes Definition

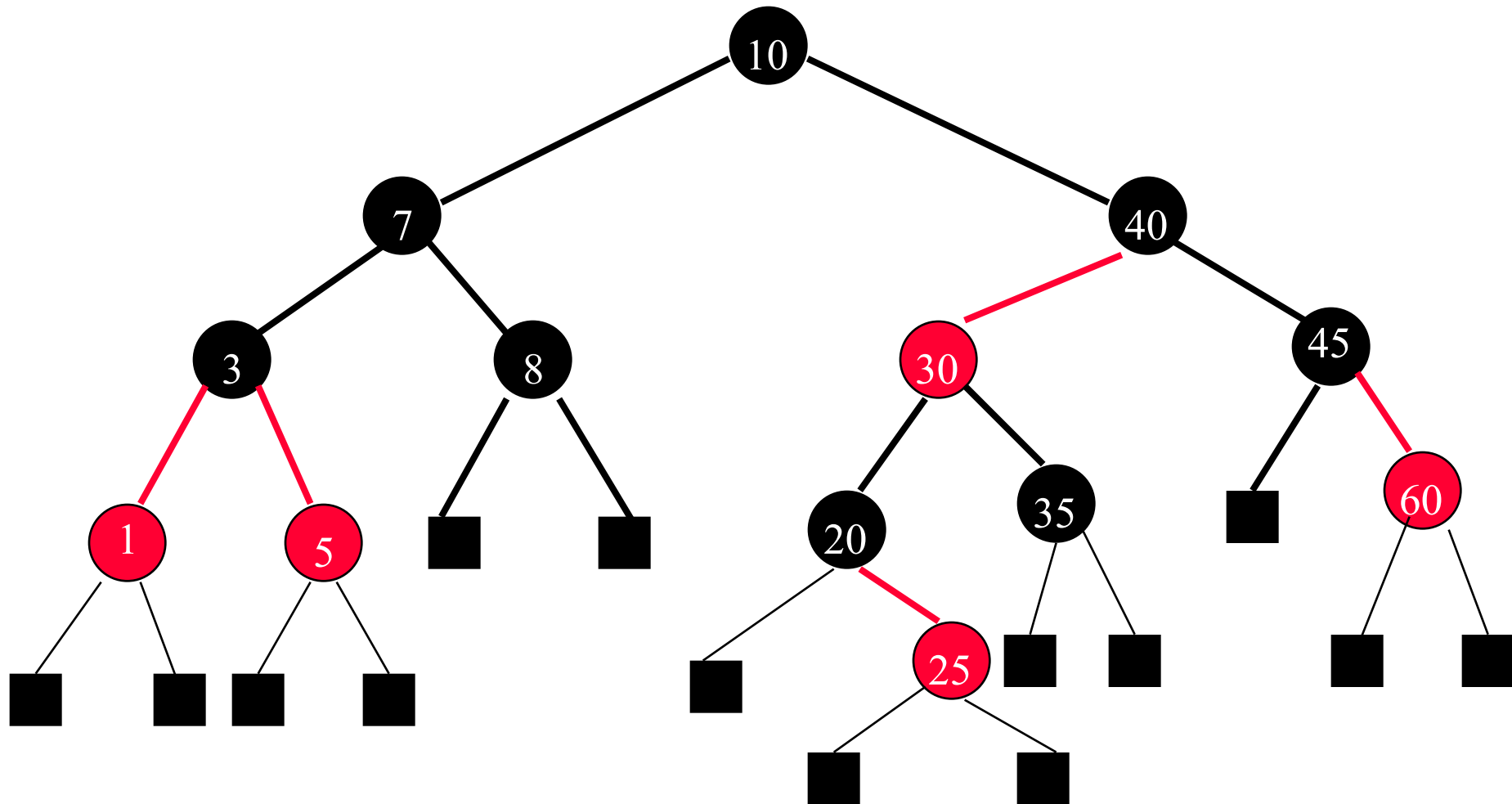
- Binary search tree.
- Each node is colored **red** or black.
- Root and all external nodes are black.
- No root-to-external-node path has two consecutive red nodes.
- All root-to-external-node paths have the same number of black nodes

# Red Black Trees

## Colored Edges Definition

- Binary search tree.
- Child pointers are colored **red** or black.
- Pointer to an external node is black.
- No root to external node path has two consecutive **red** pointers.
- Every root to external node path has the same number of black pointers.

# Example Red-Black Tree



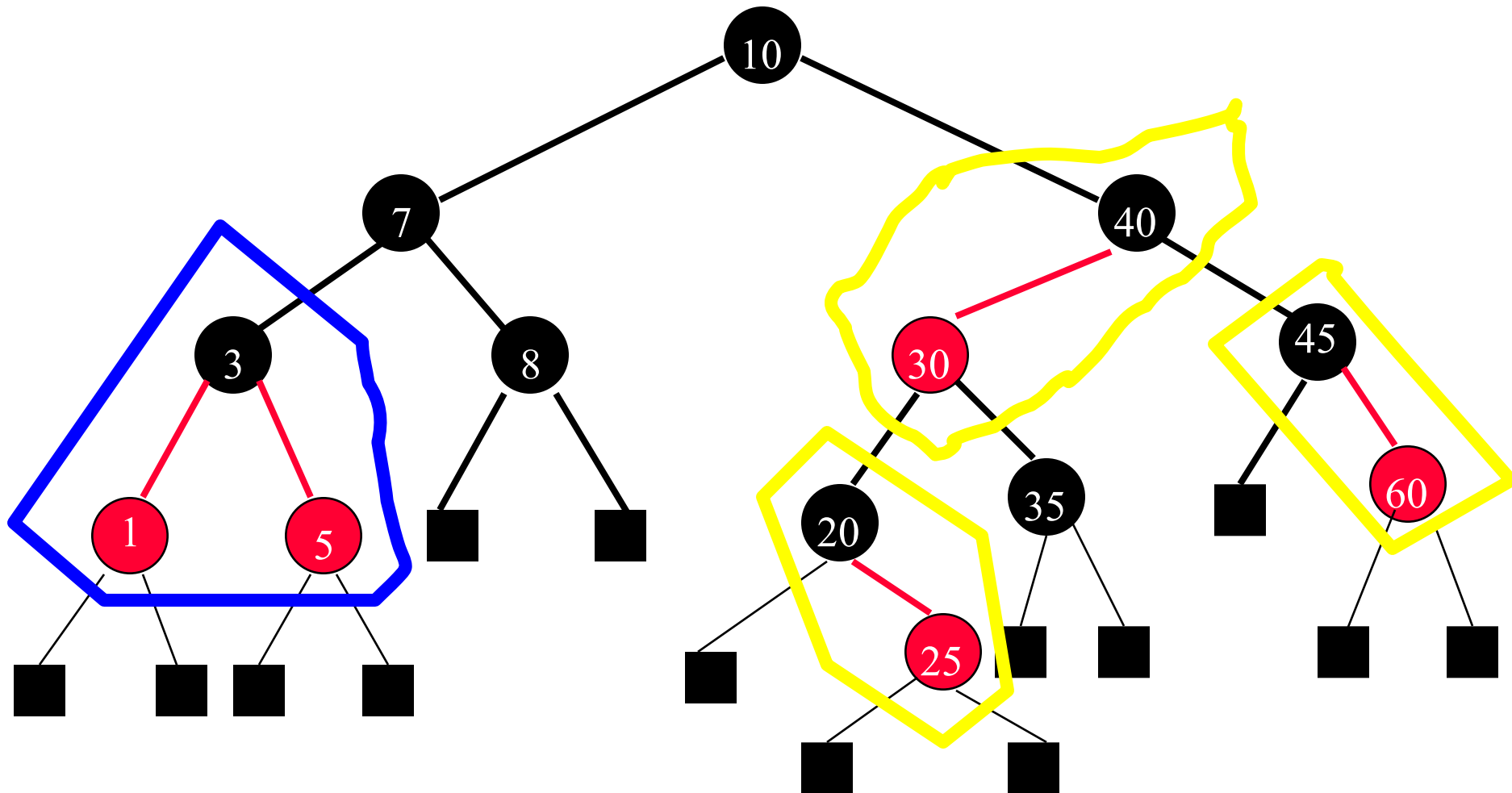
# Properties

- The height of a red black tree that has  $n$  (internal) nodes is between  $\log_2(n+1)$  and  $2\log_2(n+1)$ .

# Properties

- Start with a red black tree whose height is  $h$ ; collapse all red nodes into their parent black nodes to get a tree whose node-degrees are between 2 and 4, height is  $\geq h/2$ , and all external nodes are at the same level.

# Properties



# Properties

- Let  $h' \geq h/2$  be the height of the collapsed tree.
- Internal nodes of collapsed tree have degree between 2 and 4.
- Number of internal nodes in collapsed tree  $\geq 2^{h'} - 1$ .
- So,  $n \geq 2^{h'} - 1$
- So,  $h \leq 2 \log_2 (n + 1)$



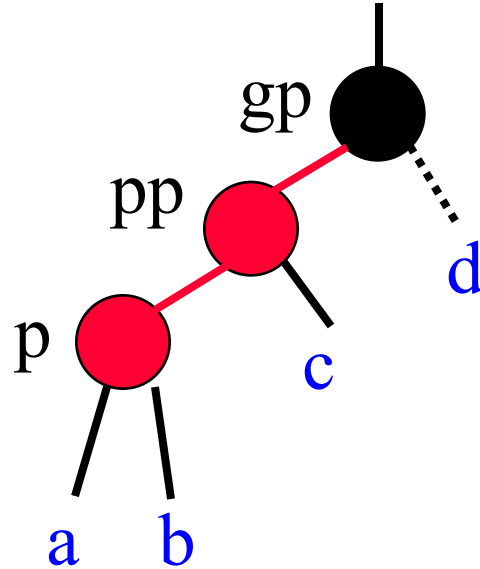
# Properties

- $O(1)$  amortized complexity to restructure following an insert/delete.
- C++ STL implementation
- `java.util.TreeMap`  $\Rightarrow$  red black tree

# Insert

- New pair is placed in a new node, which is inserted into the red-black tree.
- New node color options.
  - Black node  $\Rightarrow$  one root-to-external-node path has an extra black node (black pointer).
    - Hard to remedy.
  - Red node  $\Rightarrow$  one root-to-external-node path may have two consecutive red nodes (pointers).
    - May be remedied by color flips and/or a rotation.

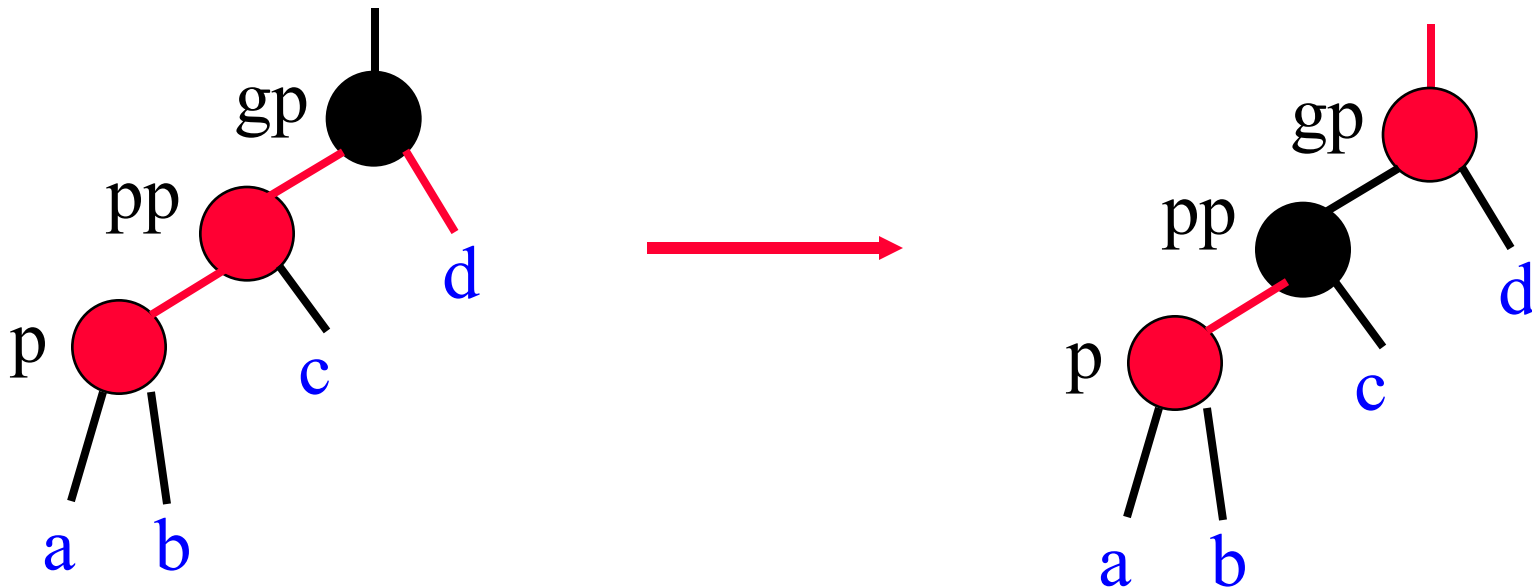
# Classification Of 2 Red Nodes/Pointers



- **XYZ**
  - **X** => relationship between **gp** and **pp**.
    - **pp** left child of **gp** => **X = L**.
  - **Y** => relationship between **pp** and **p**.
    - **p** left child of **pp** => **Y = L**.
  - **z = b** (black) if **d = null** or a black node.
  - **z = r** (red) if **d** is a red node.

# XYr

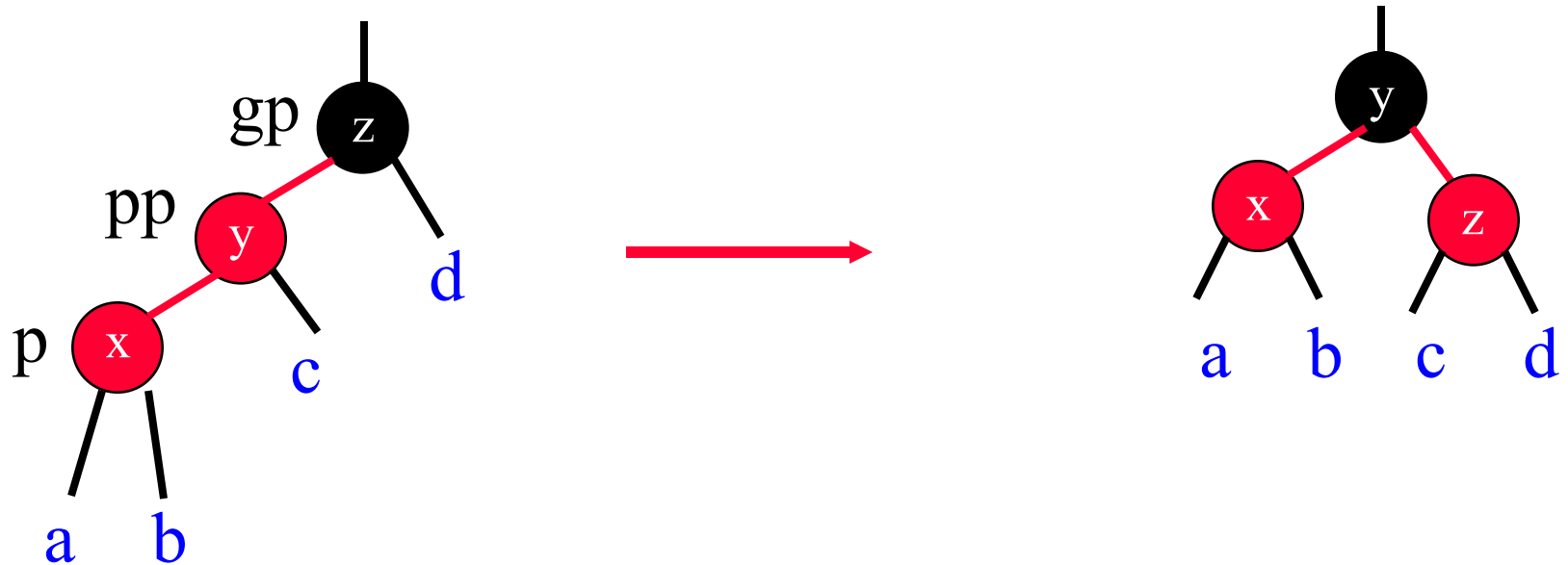
- Color flip.



- Move **p**, **pp**, and **gp** up two levels.
- Continue rebalancing.

# LLb

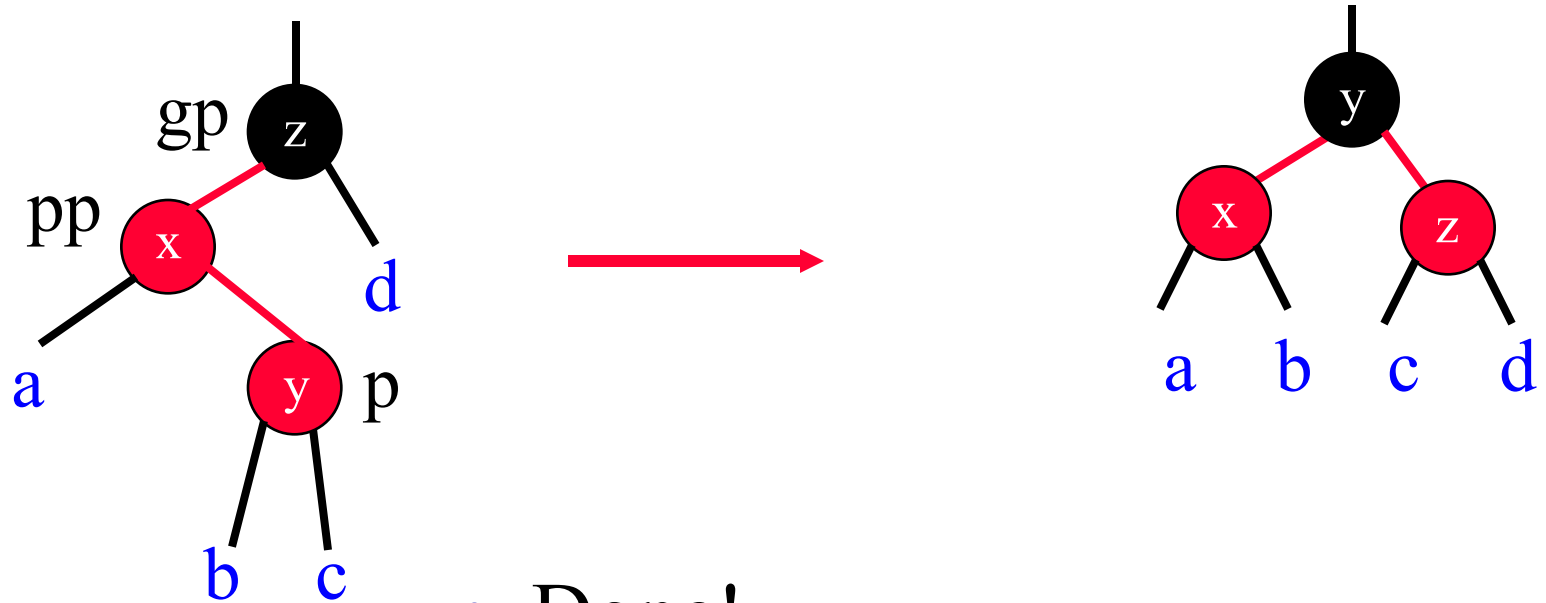
- Rotate.



- Done!
- Same as LL rotation of AVL tree.

# LRb

- Rotate.

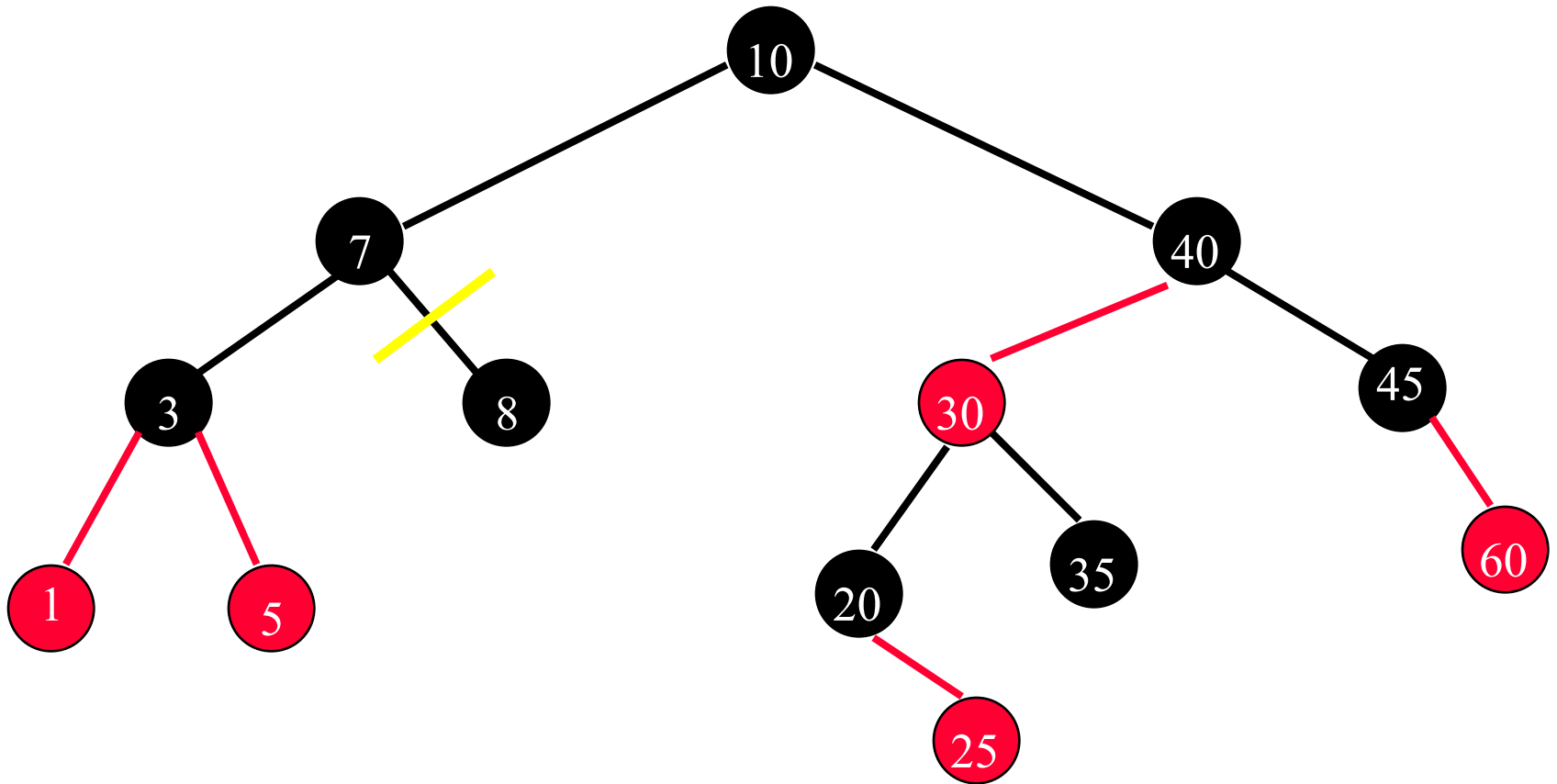


- Done!
- Same as LR rotation of AVL tree.
- RRb and RLb are symmetric.

# Delete

- Delete as for unbalanced binary search tree.
- If red node deleted, no rebalancing needed.
- If black node deleted, a subtree becomes one black pointer (node) deficient.

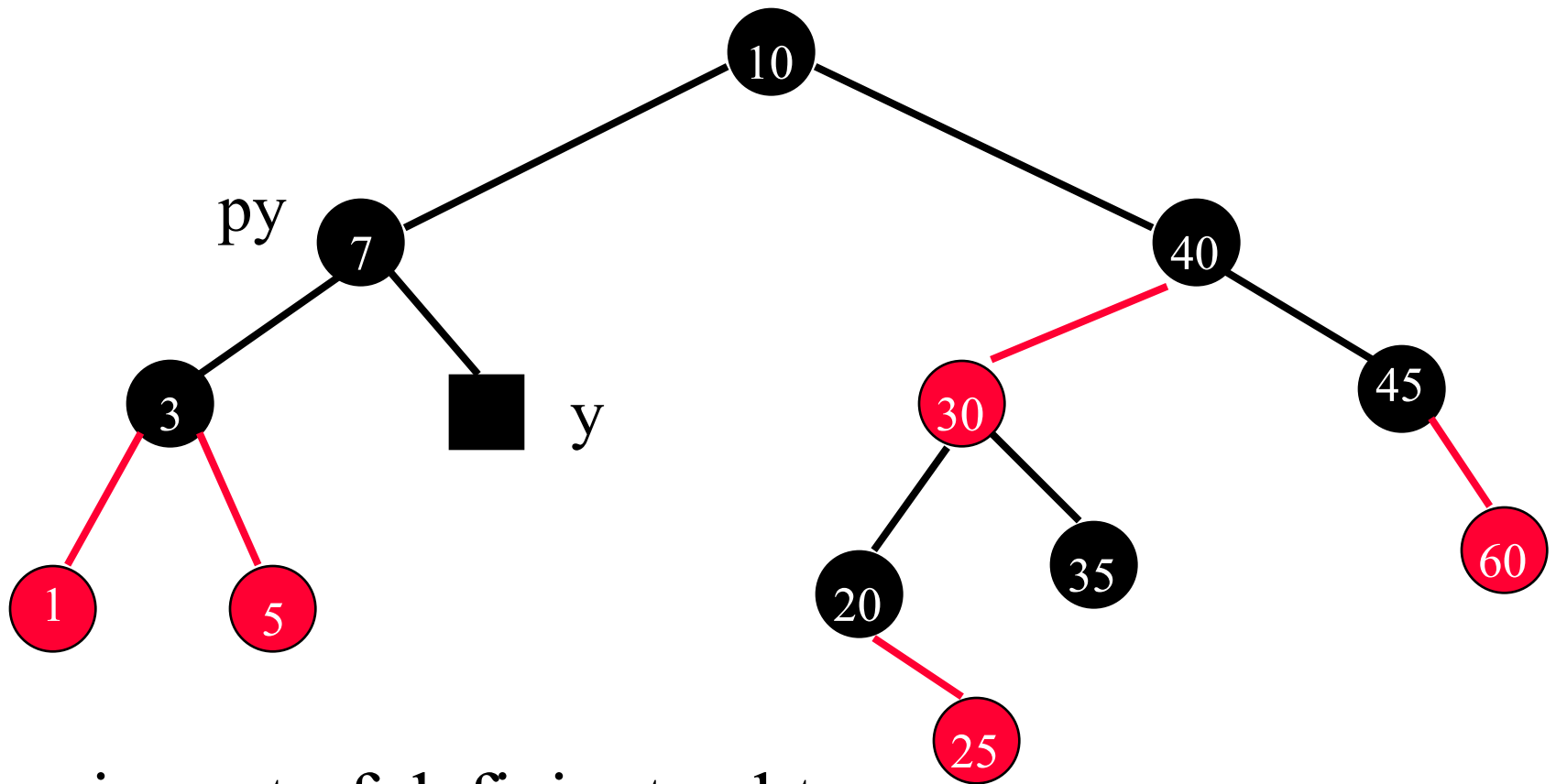
# Delete A Black Leaf



- Delete 8.

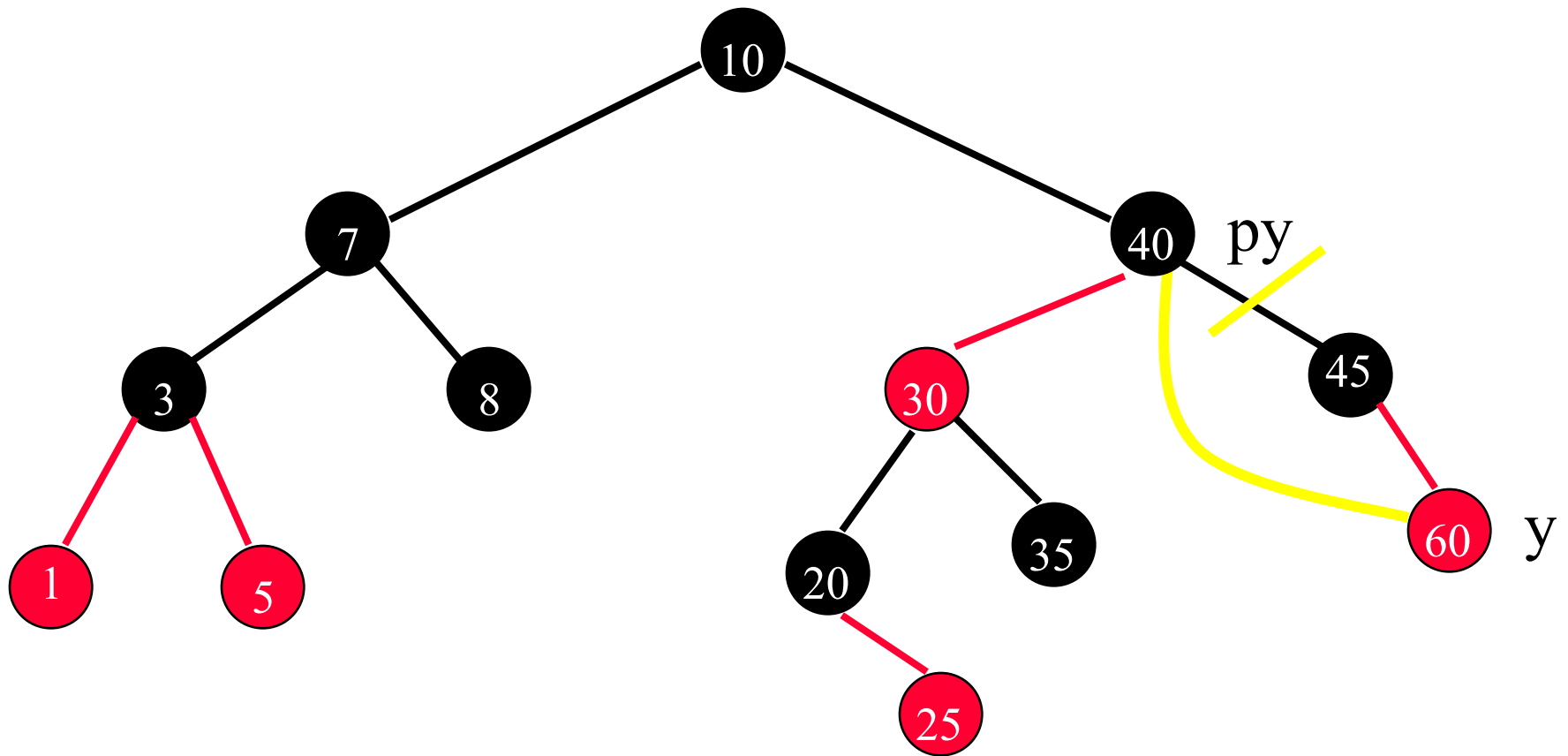


# Delete A Black Leaf



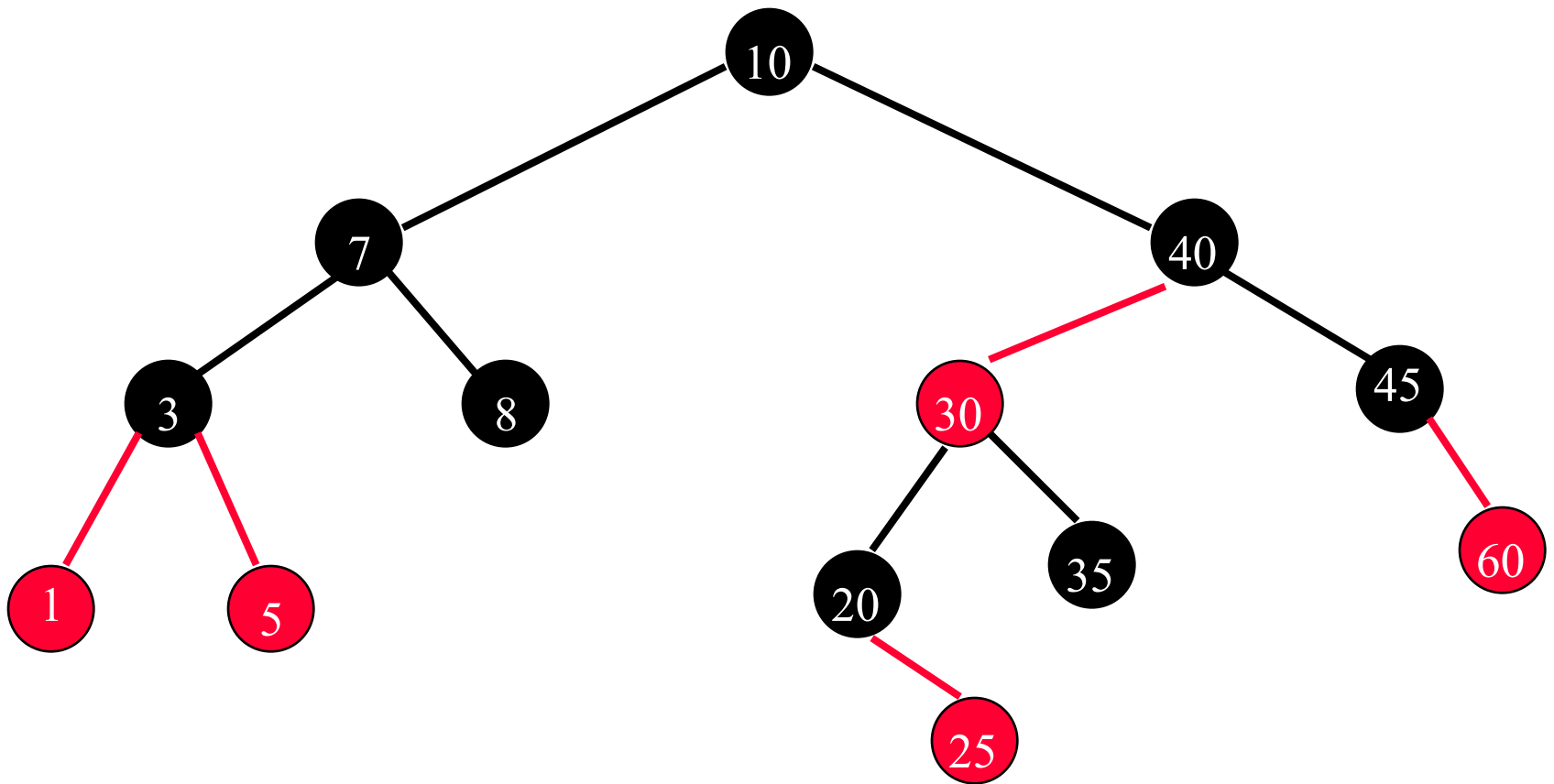
- $y$  is root of deficient subtree.
- $py$  is parent of  $y$ .

# Delete A Black Degree 1 Node



- Delete 45.
- $y$  is root of deficient subtree.

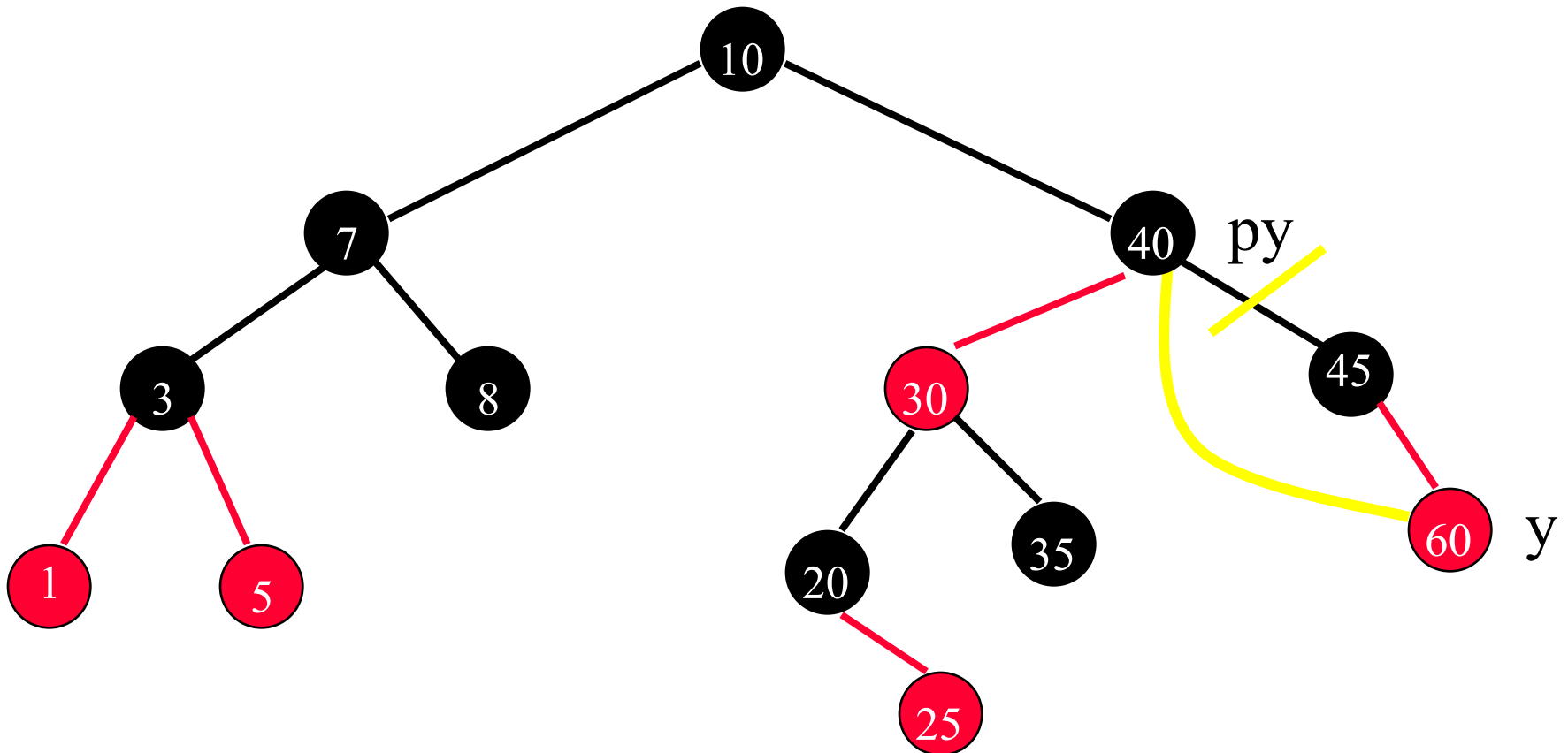
# Delete A Black Degree 2 Node



- Not possible, degree 2 nodes are never deleted.

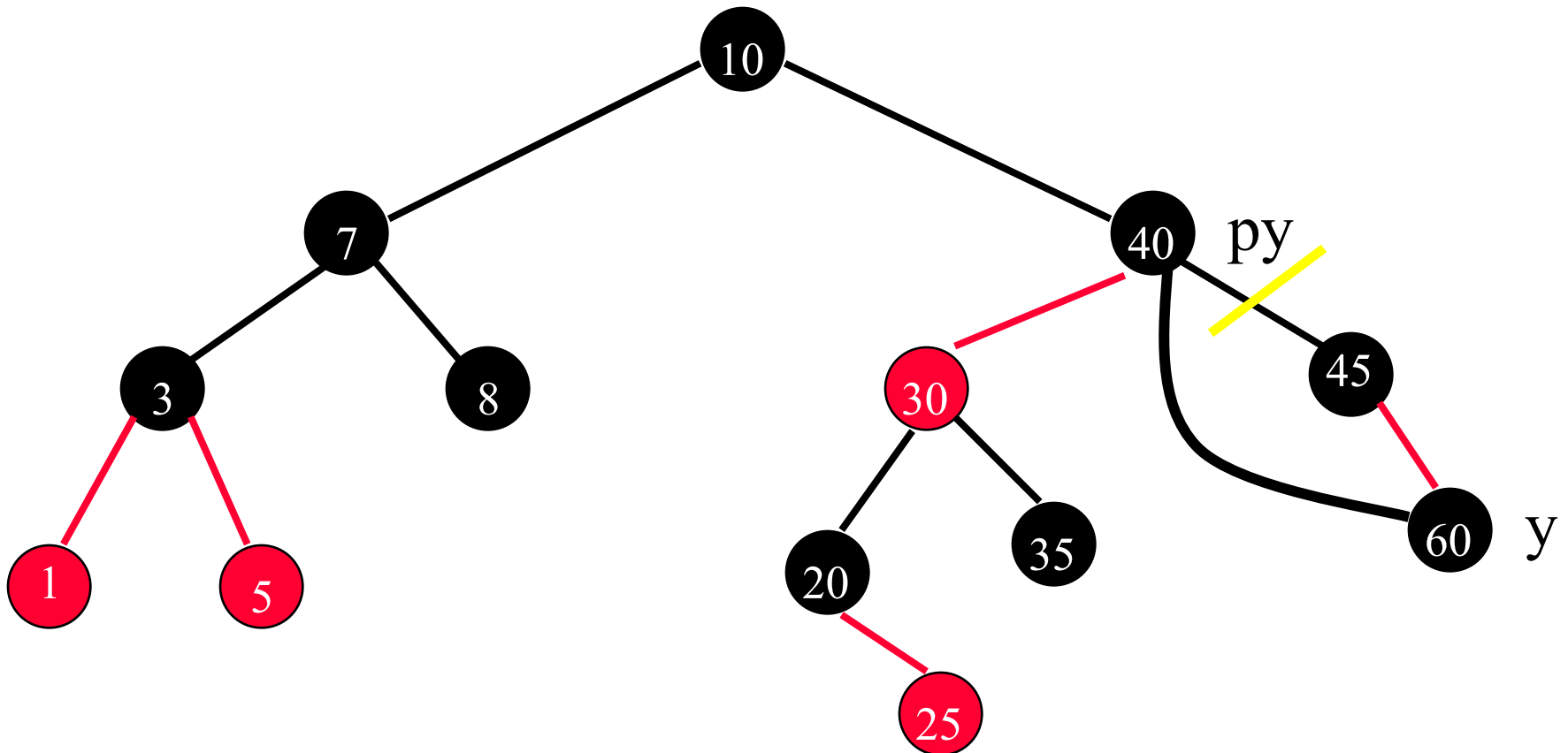
# Rebalancing Strategy

- If  $y$  is a red node, make it black.



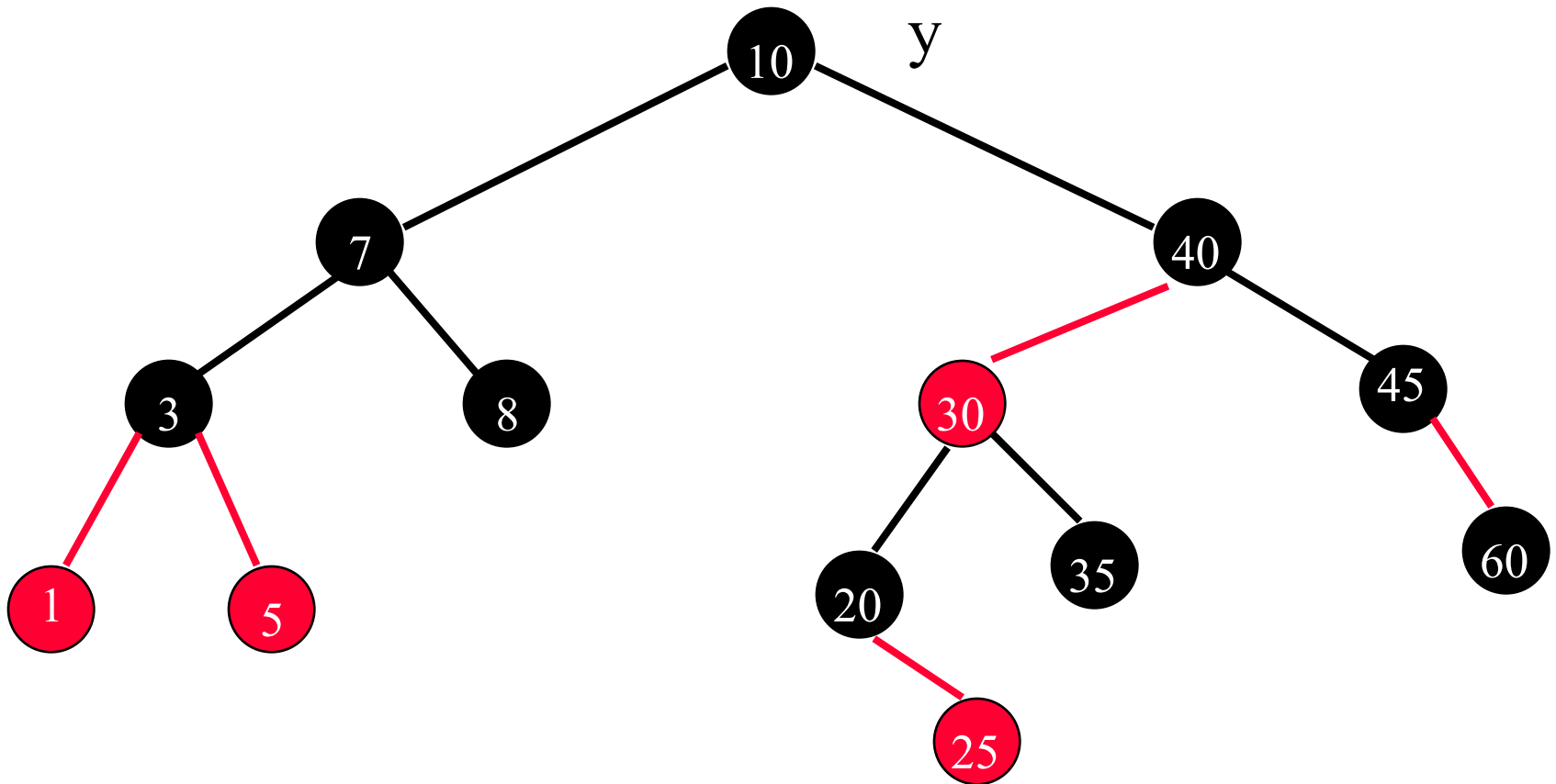
# Rebalancing Strategy

- Now, no subtree is deficient. Done!



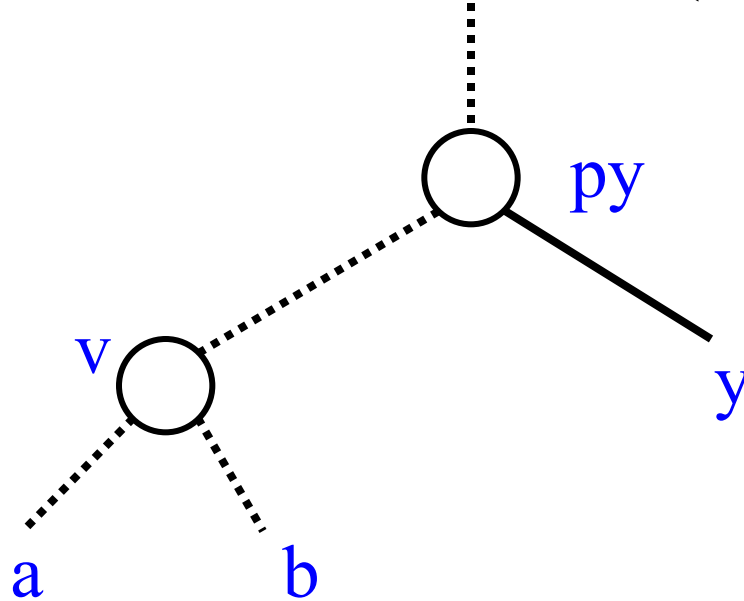
# Rebalancing Strategy

- **y** is a black root (there is no **py**).
- Entire tree is deficient. Done!



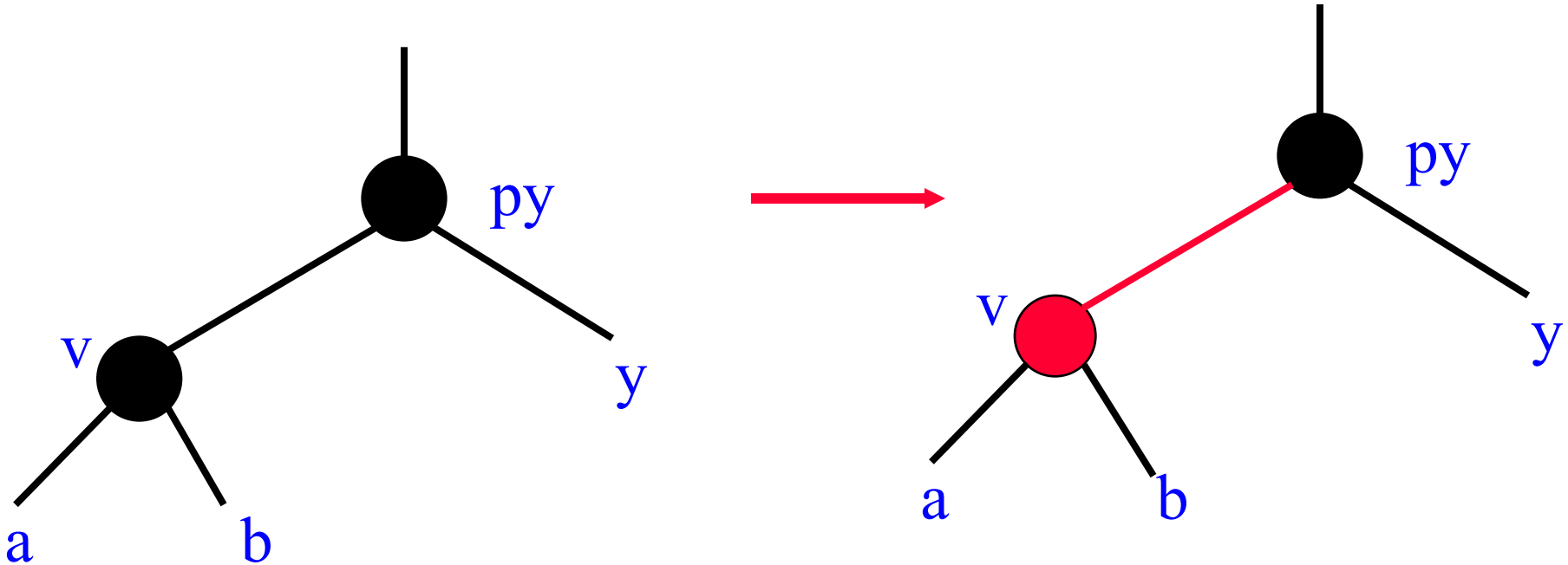
# Rebalancing Strategy

- $y$  is black but not the root (there is a  $py$ ).



- $Xcn$ 
  - $y$  is right child of  $py \Rightarrow X = R$ .
  - Pointer to  $v$  is black  $\Rightarrow c = b$ .
  - $v$  has 1 red child  $\Rightarrow n = 1$ .

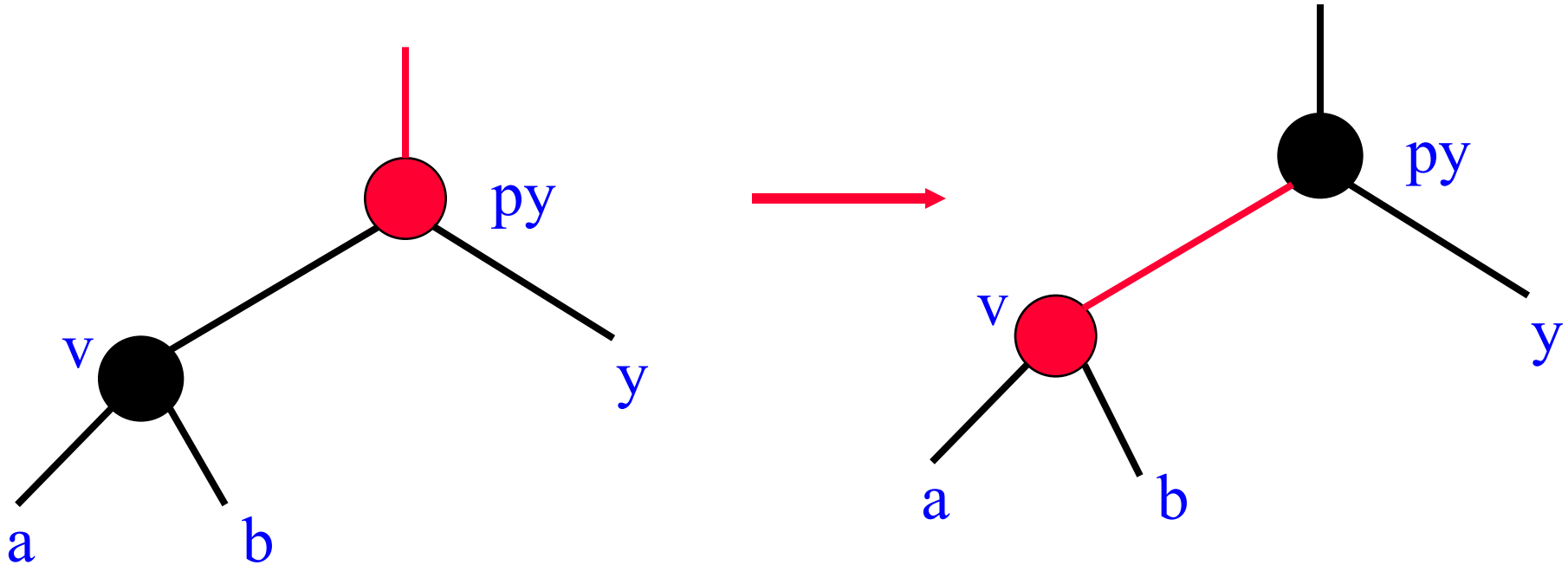
## Rb0 (case 1, py is black)



- Color change.
- Now,  $py$  is root of deficient subtree.
- Continue!

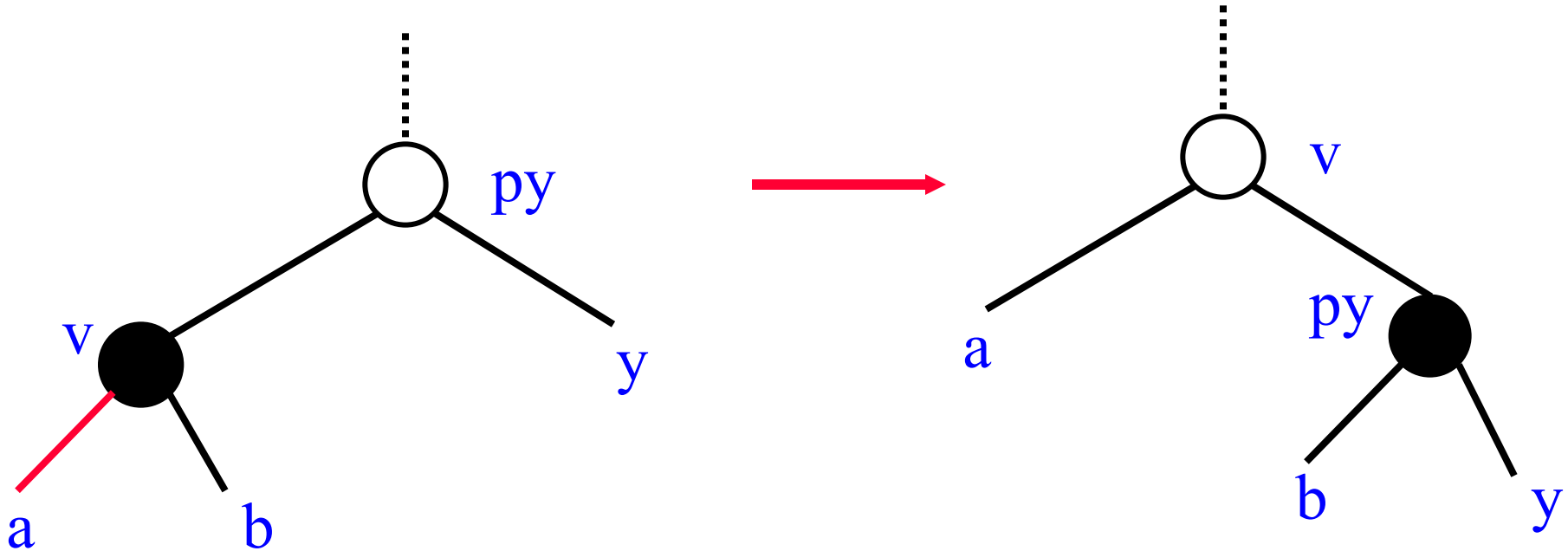


## Rb0 (case 2, py is red)



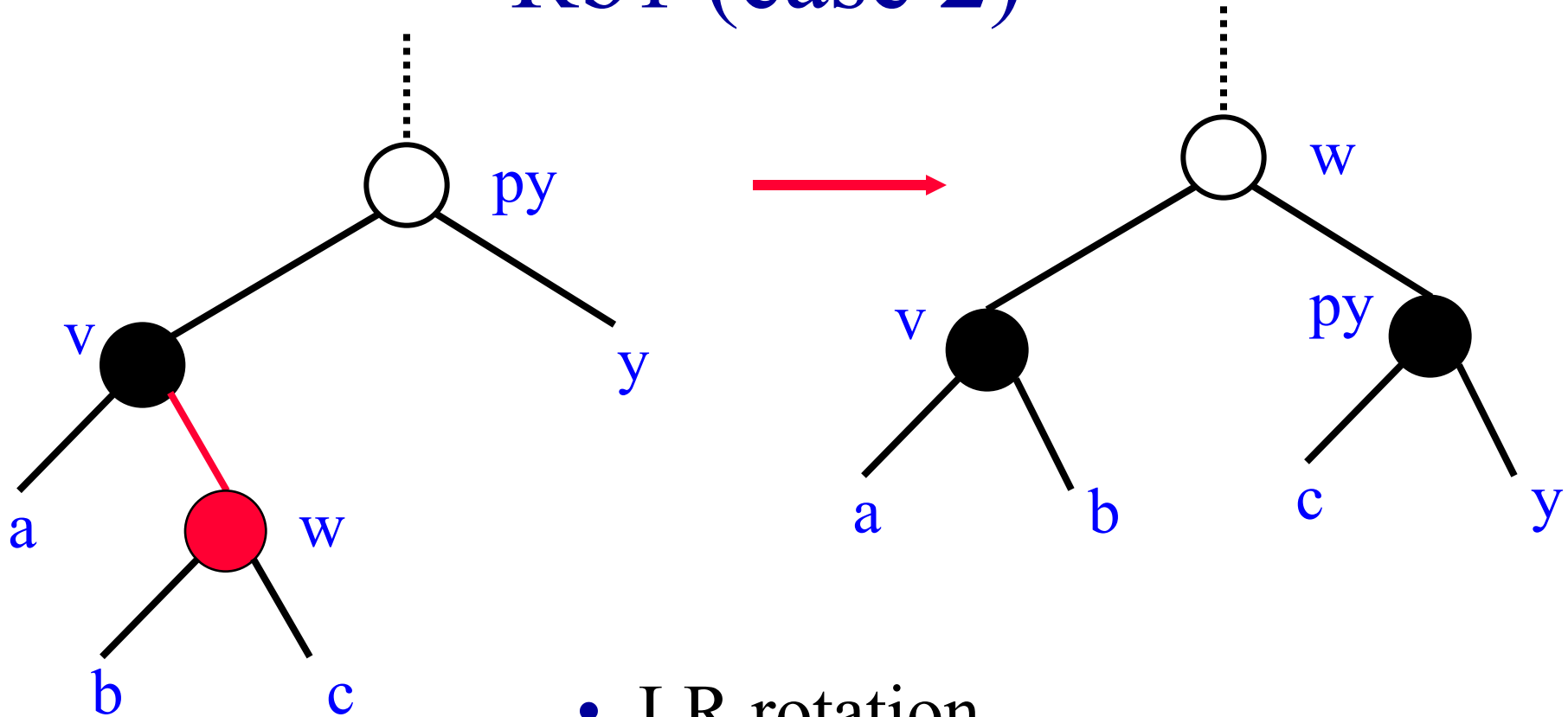
- Color change.
- Deficiency eliminated.
- Done!

# Rb1 (case 1)



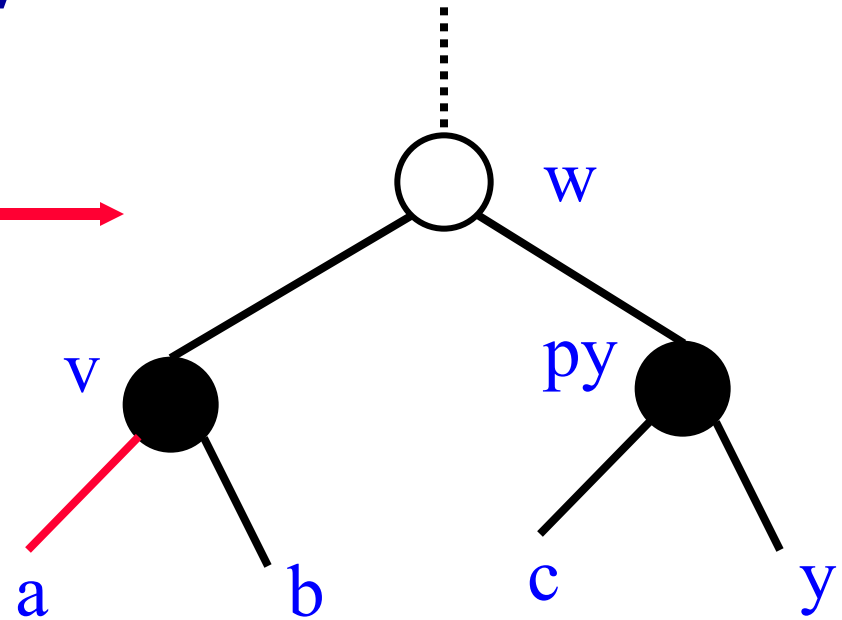
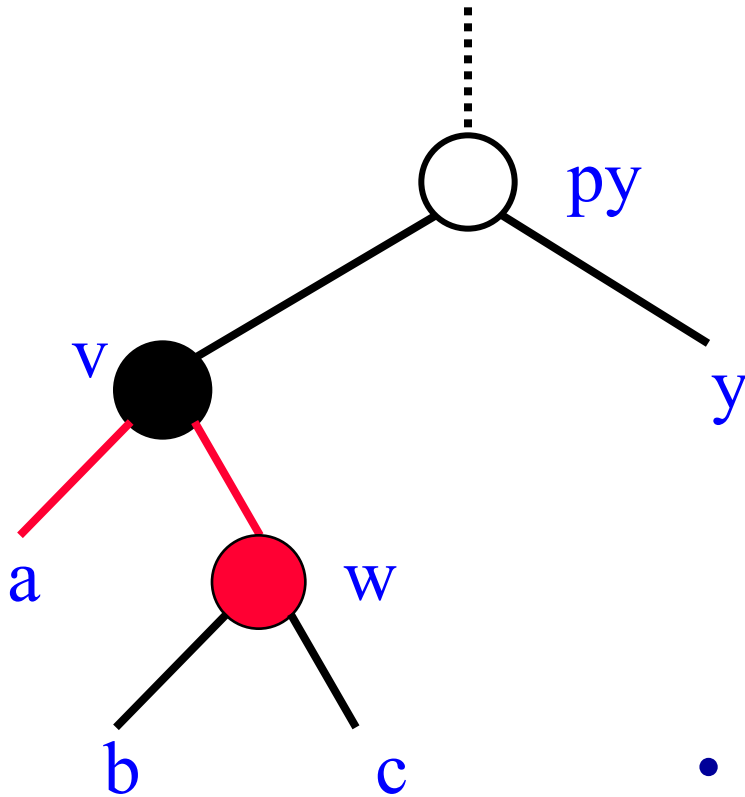
- LL rotation.
- Deficiency eliminated.
- Done!

## Rb1 (case 2)



- LR rotation.
- Deficiency eliminated.
- Done!

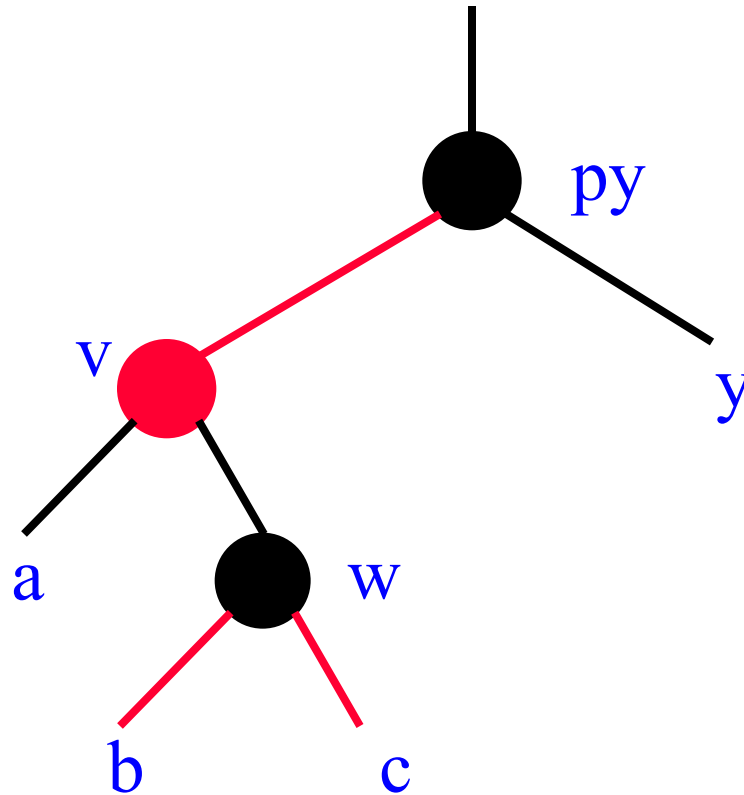
# Rb2



- LR rotation.
- Deficiency eliminated.
- Done!

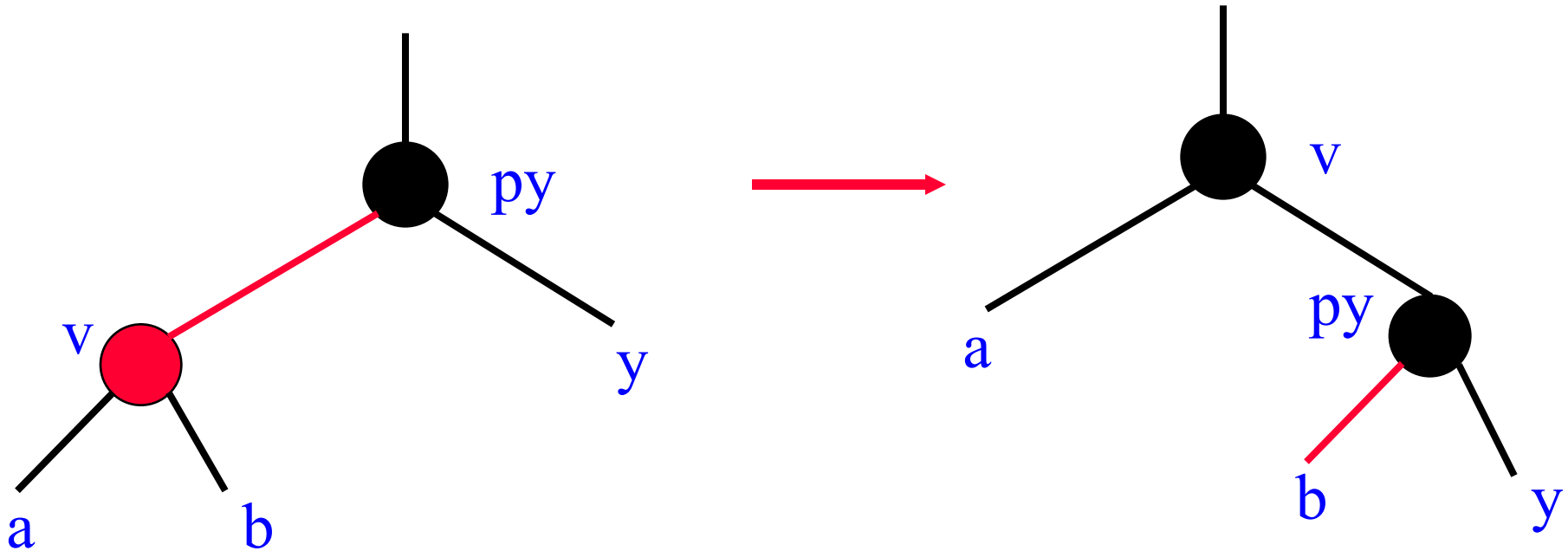
$$Rr(n)$$

- $n$  = # of red children of  $v$ 's right child  $w$ .



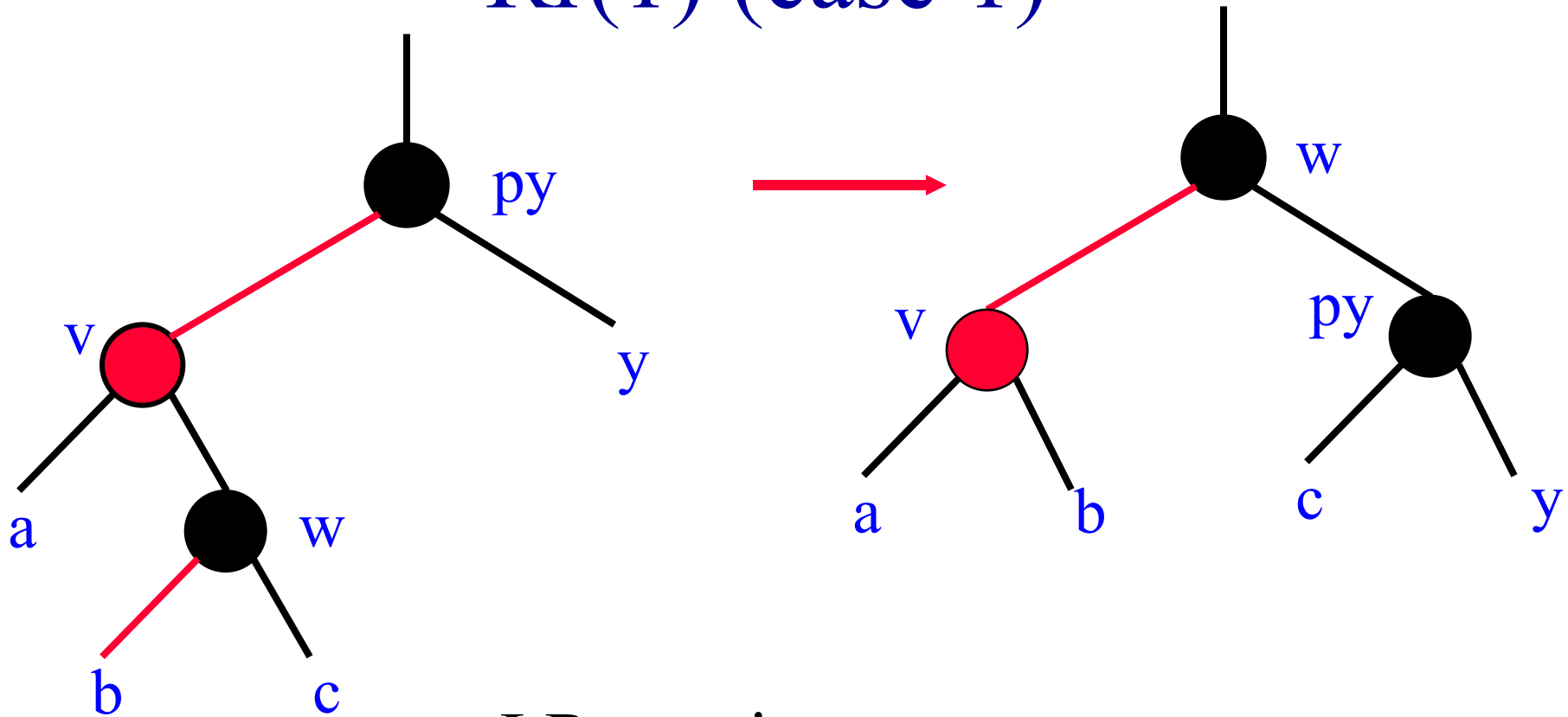
**Rr(2)**

$Rr(0)$



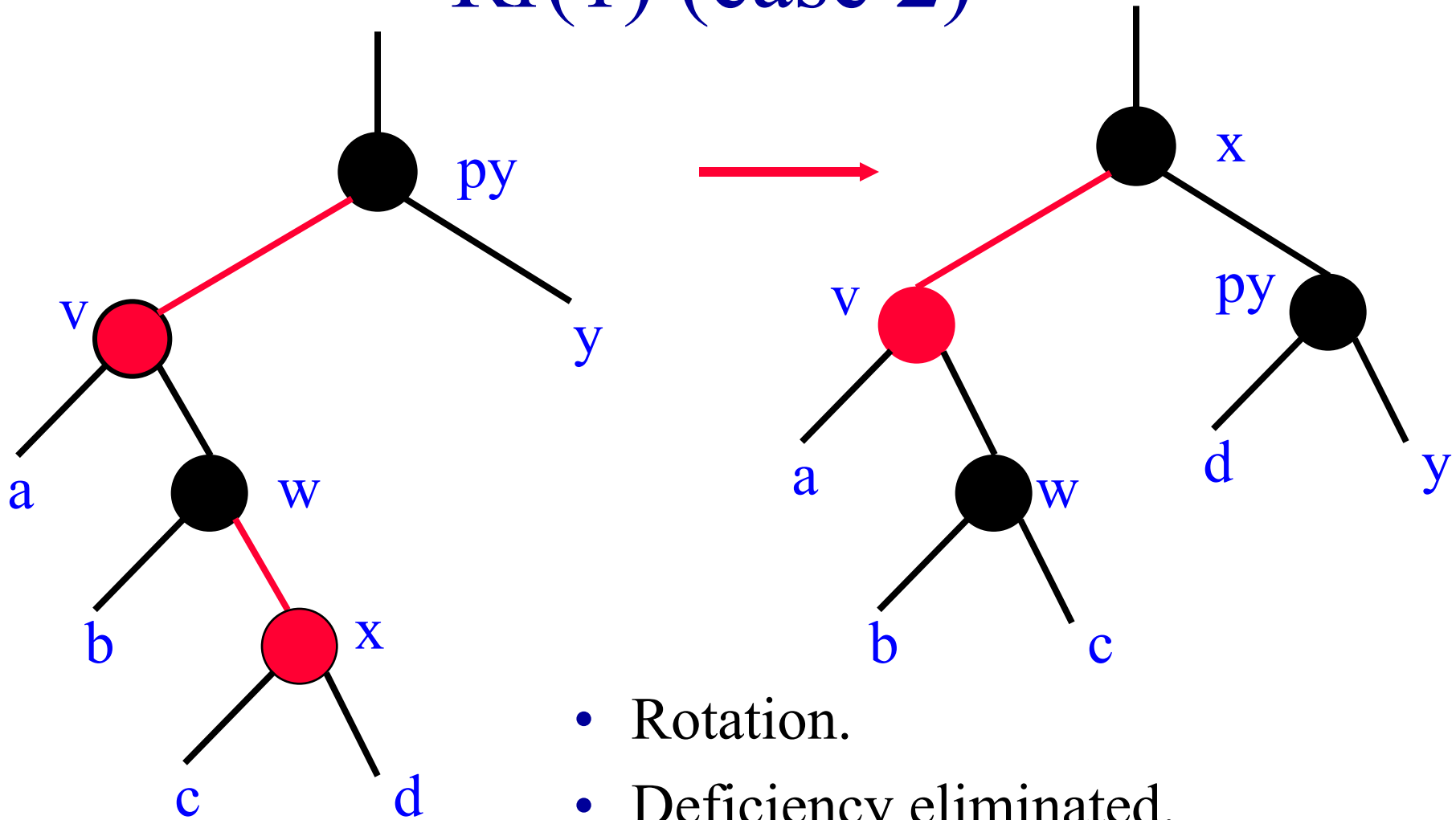
- LL rotation.
- Done!

## Rr(1) (case 1)



- LR rotation.
- Deficiency eliminated.
- Done!

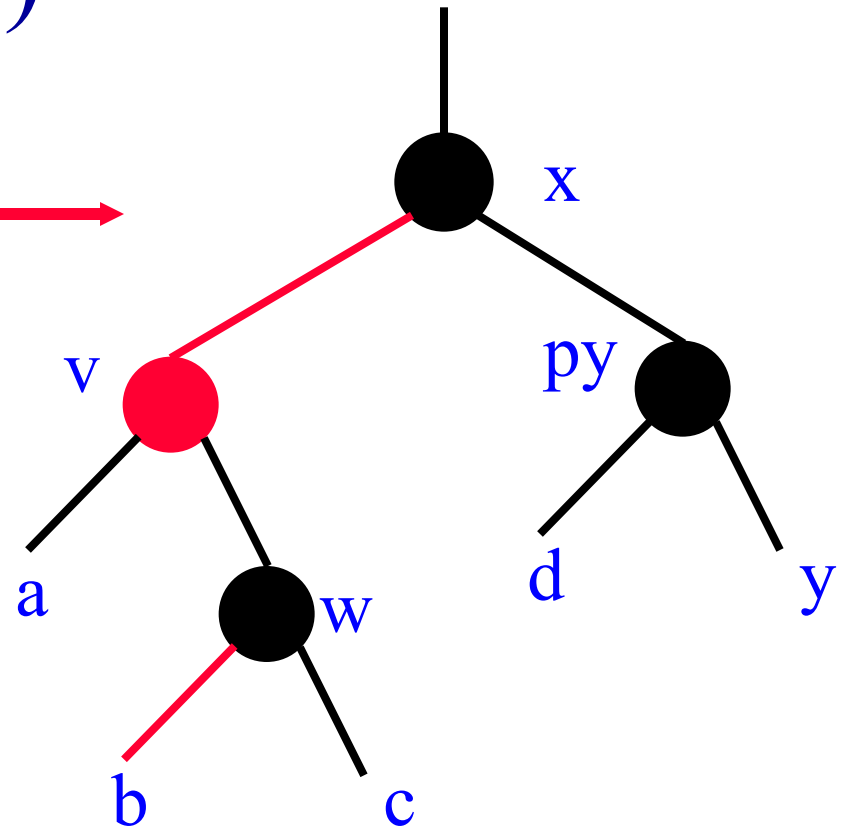
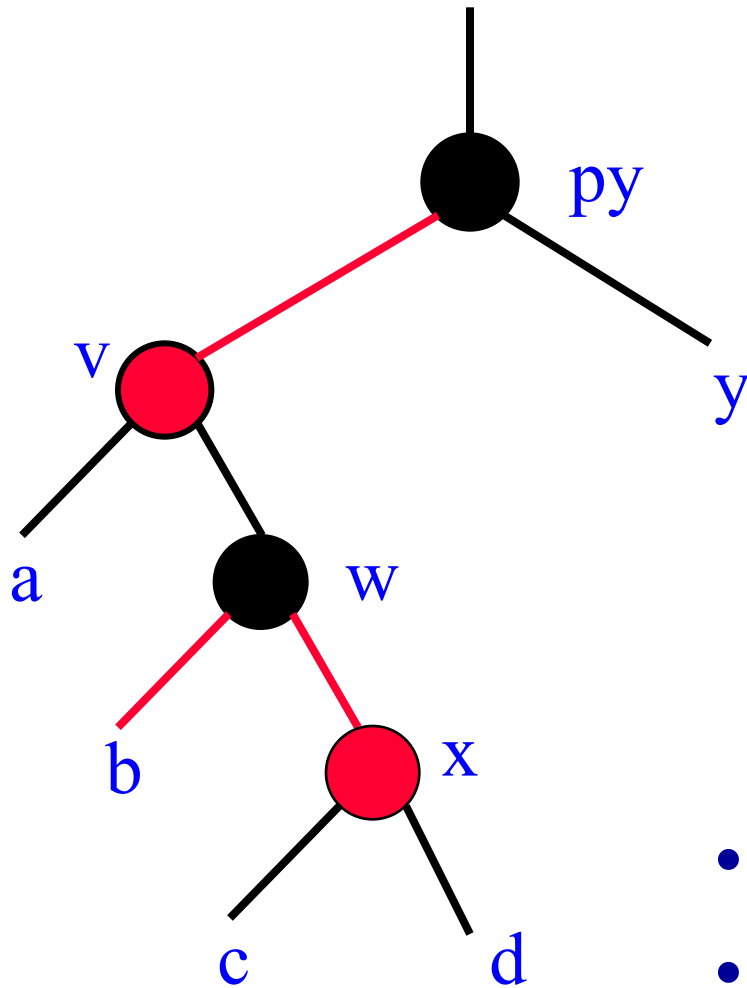
## Rr(1) (case 2)



- Rotation.
- Deficiency eliminated.
- Done!



Rr(2)

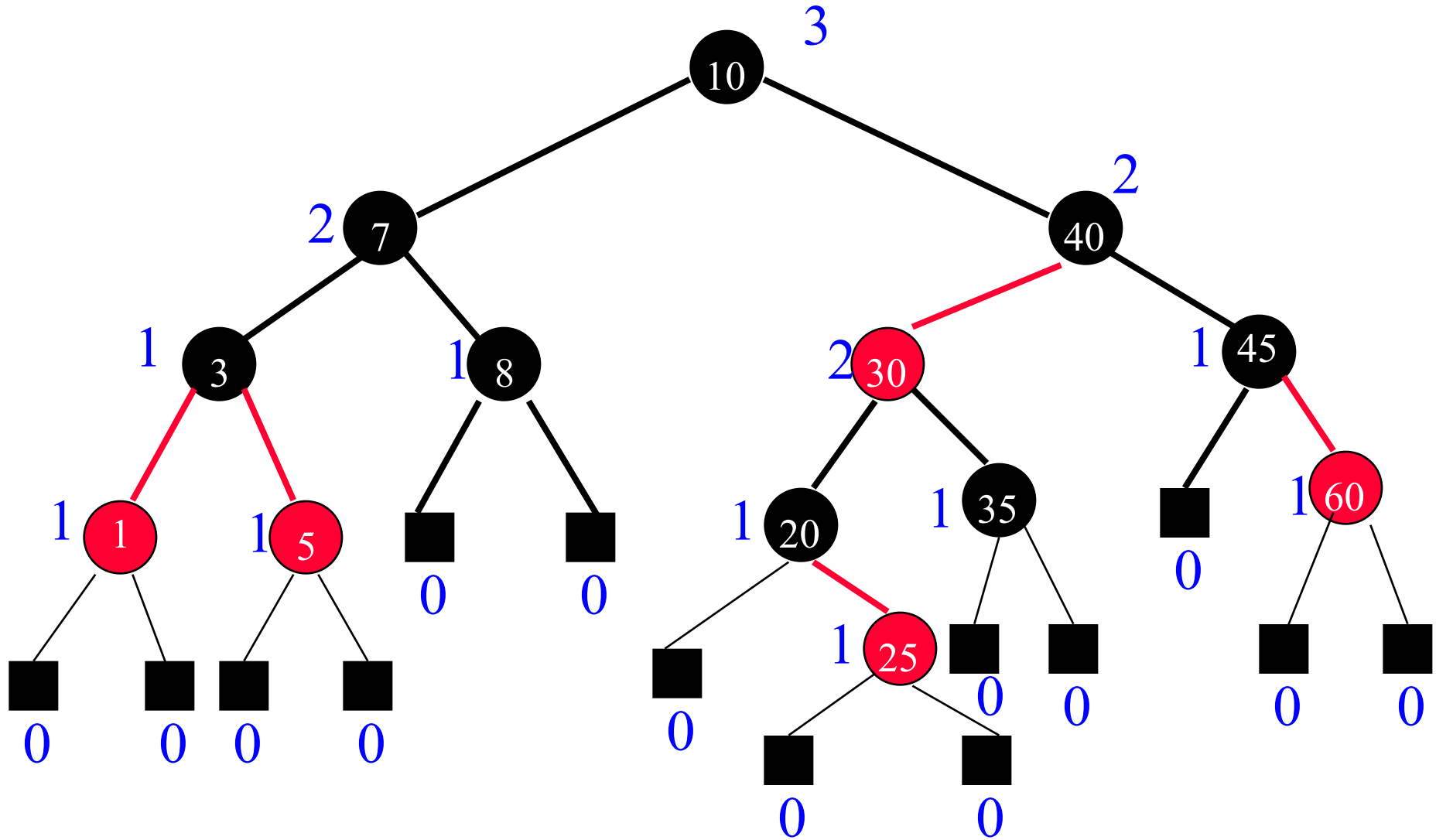


- Rotation.
- Deficiency eliminated.
- Done!

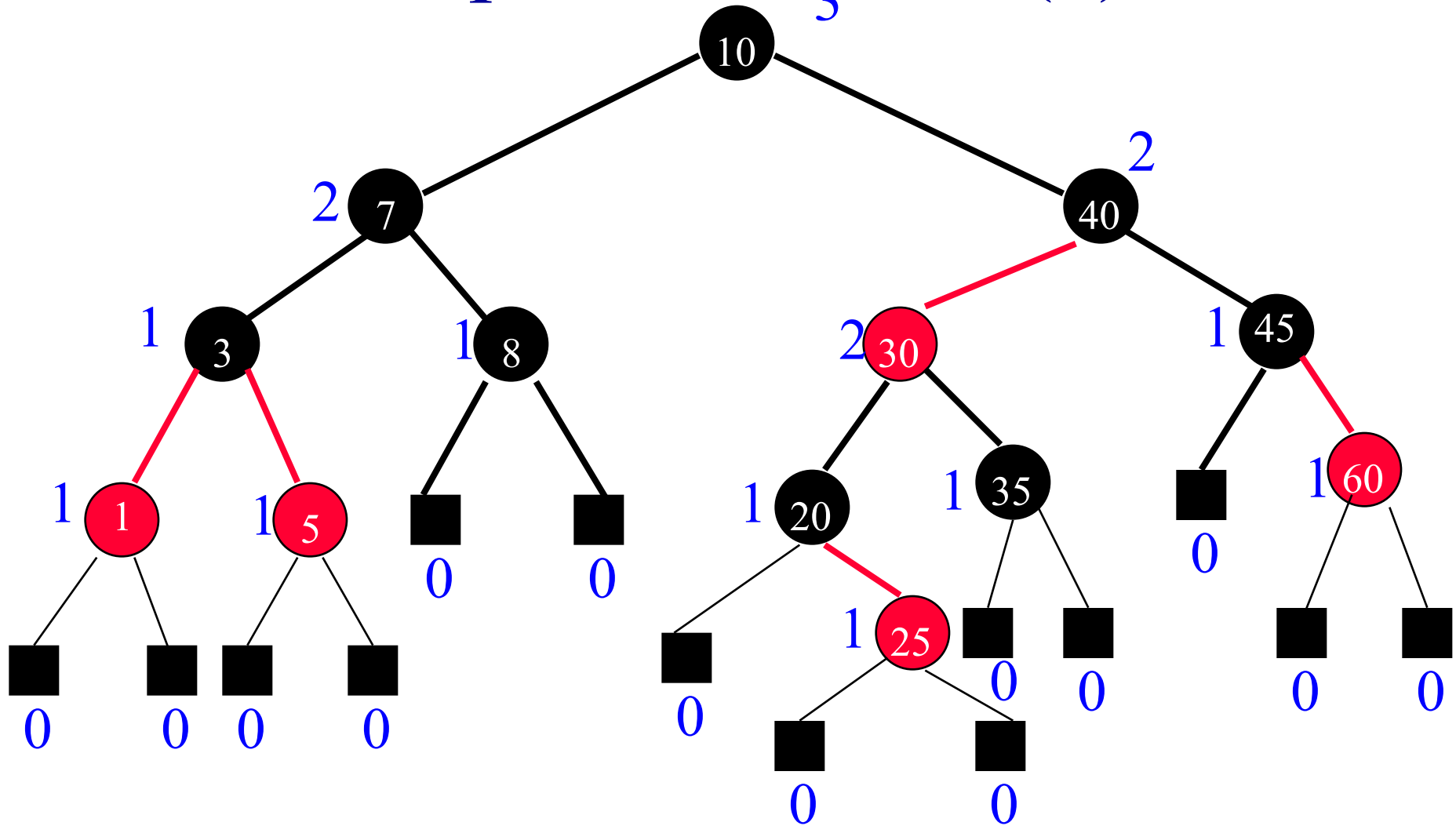
# Red-Black Trees—Rank

- $\text{rank}(x) = \# \text{ black pointers on path from } x \text{ to an external node.}$
- Same as  $\# \text{ black nodes (excluding } x) \text{ from } x \text{ to an external node.}$
- $\text{rank}(\text{external node}) = 0.$

# An Example

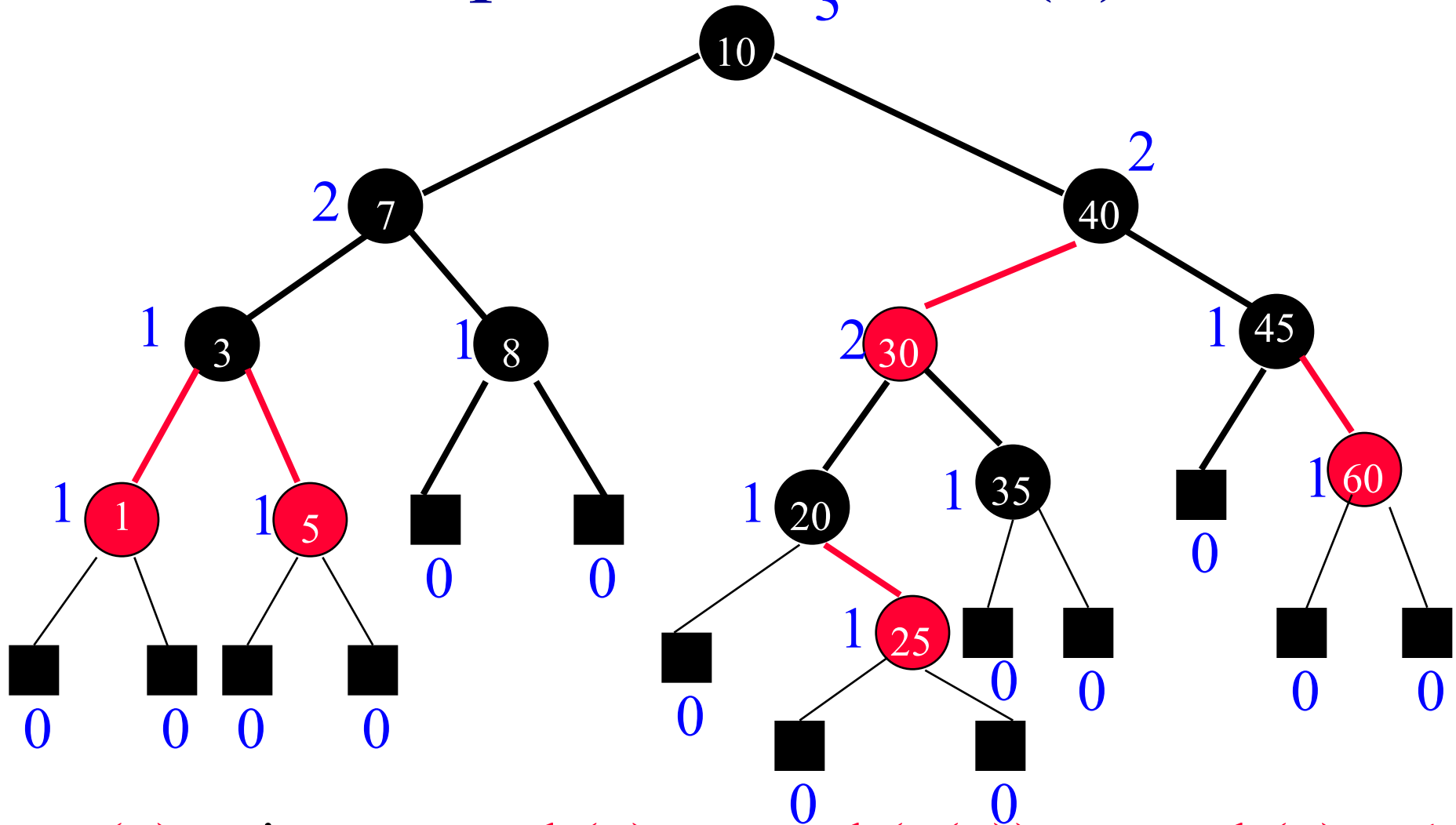


# Properties Of $\text{rank}(x)$



- $\text{rank}(x) = 0$  for  $x$  an external node.
- $\text{rank}(x) = 1$  for  $x$  parent of external node.

# Properties Of $\text{rank}(x)$

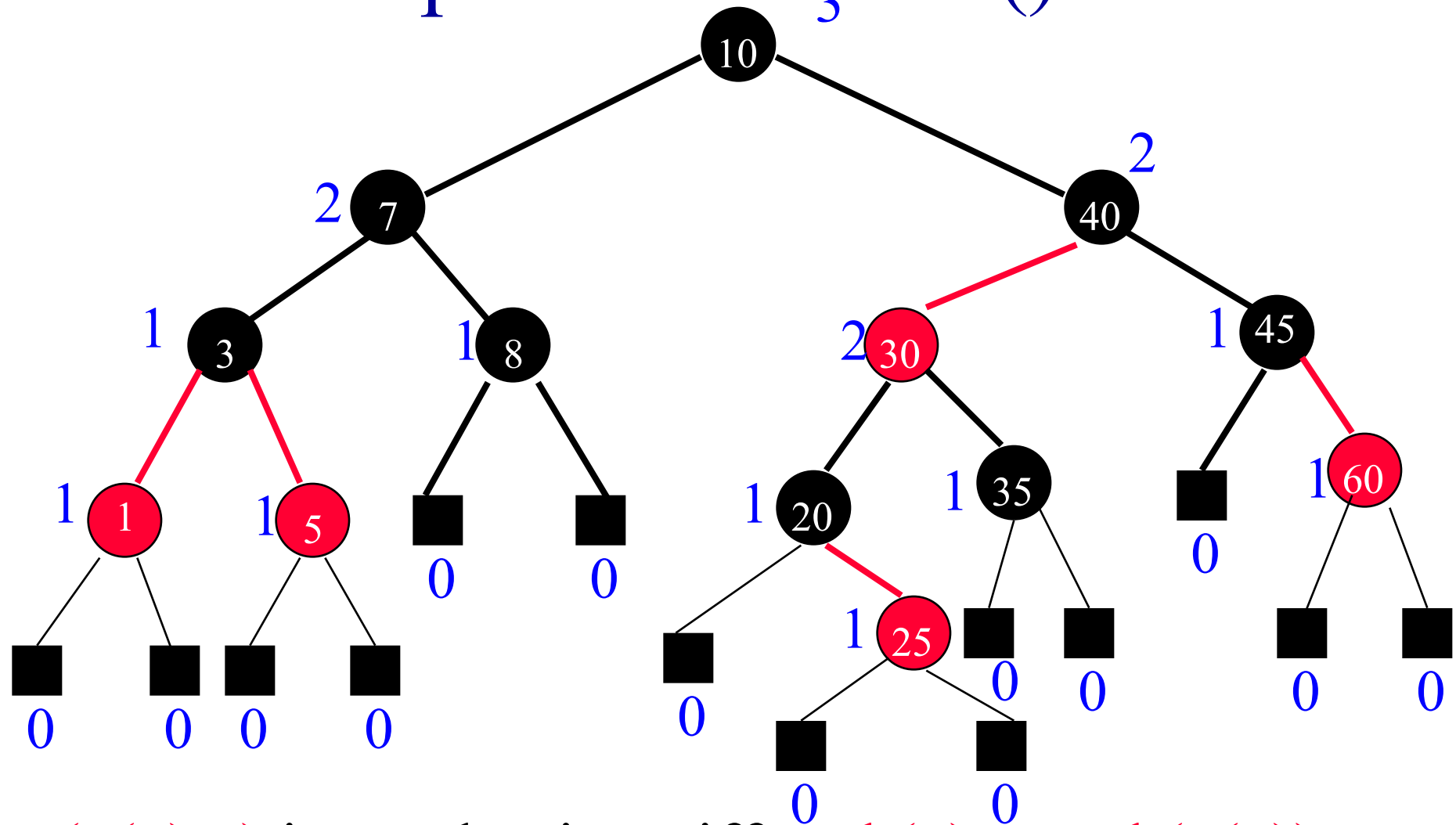


- $p(x)$  exists  $\Rightarrow \text{rank}(x) \leq \text{rank}(p(x)) \leq \text{rank}(x) + 1$ .
- $g(x)$  exists  $\Rightarrow \text{rank}(x) < \text{rank}(g(x))$ .

# Red-Black Tree

- A binary search tree is a red-black tree iff integer ranks can be assigned to its nodes so as to satisfy the stated 4 properties of rank.

# Relationship Between $\text{rank}()$ And Color



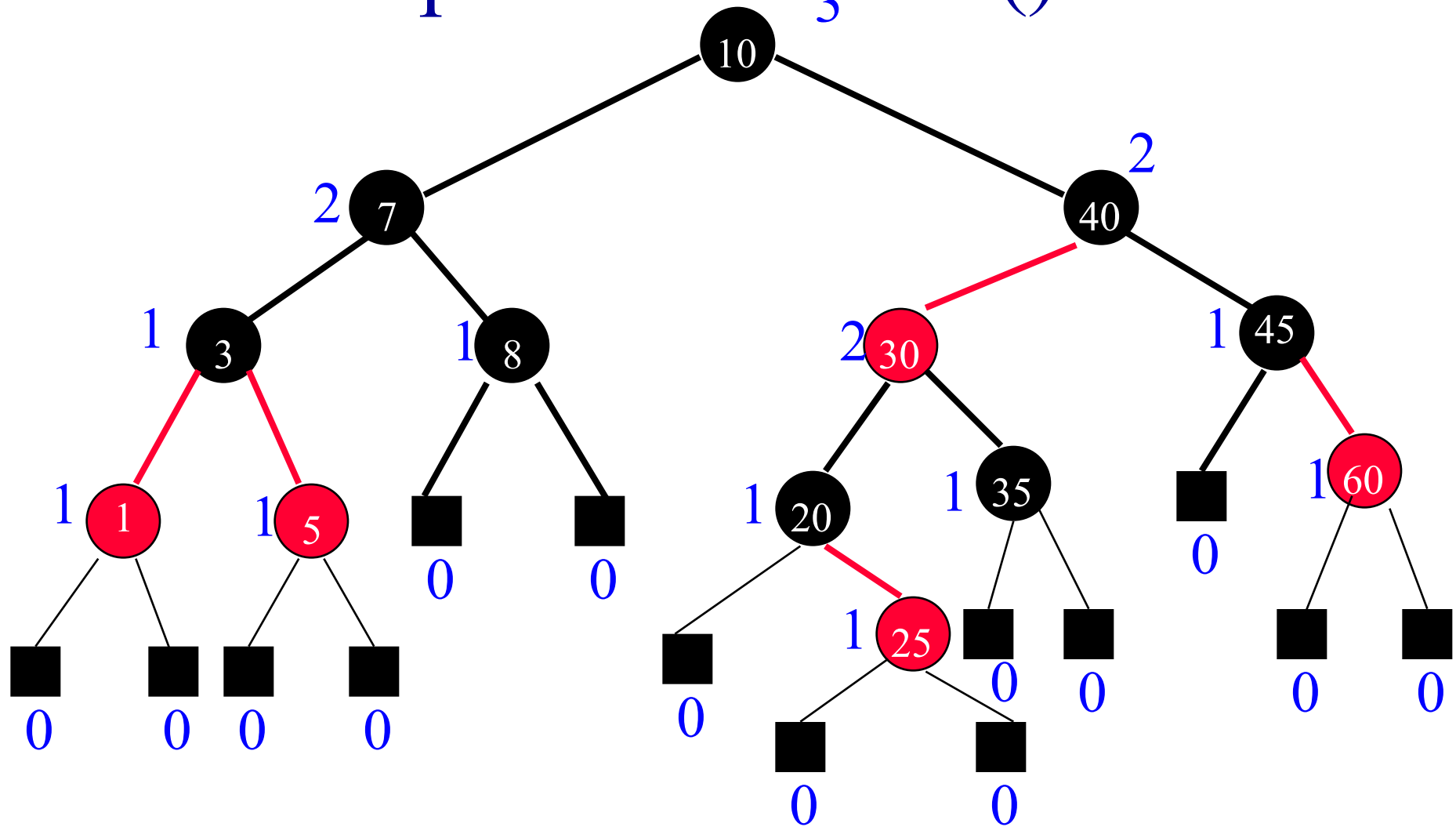
- $(p(x), x)$  is a red pointer iff  $\text{rank}(x) = \text{rank}(p(x))$ .
- $(p(x), x)$  is a black pointer iff  $\text{rank}(x) = \text{rank}(p(x)) - 1$ .

# Relationship Between rank() And Color

- Root is black.
- Other nodes:
  - Red iff pointer from parent is red.
  - Black iff pointer from parent is black.

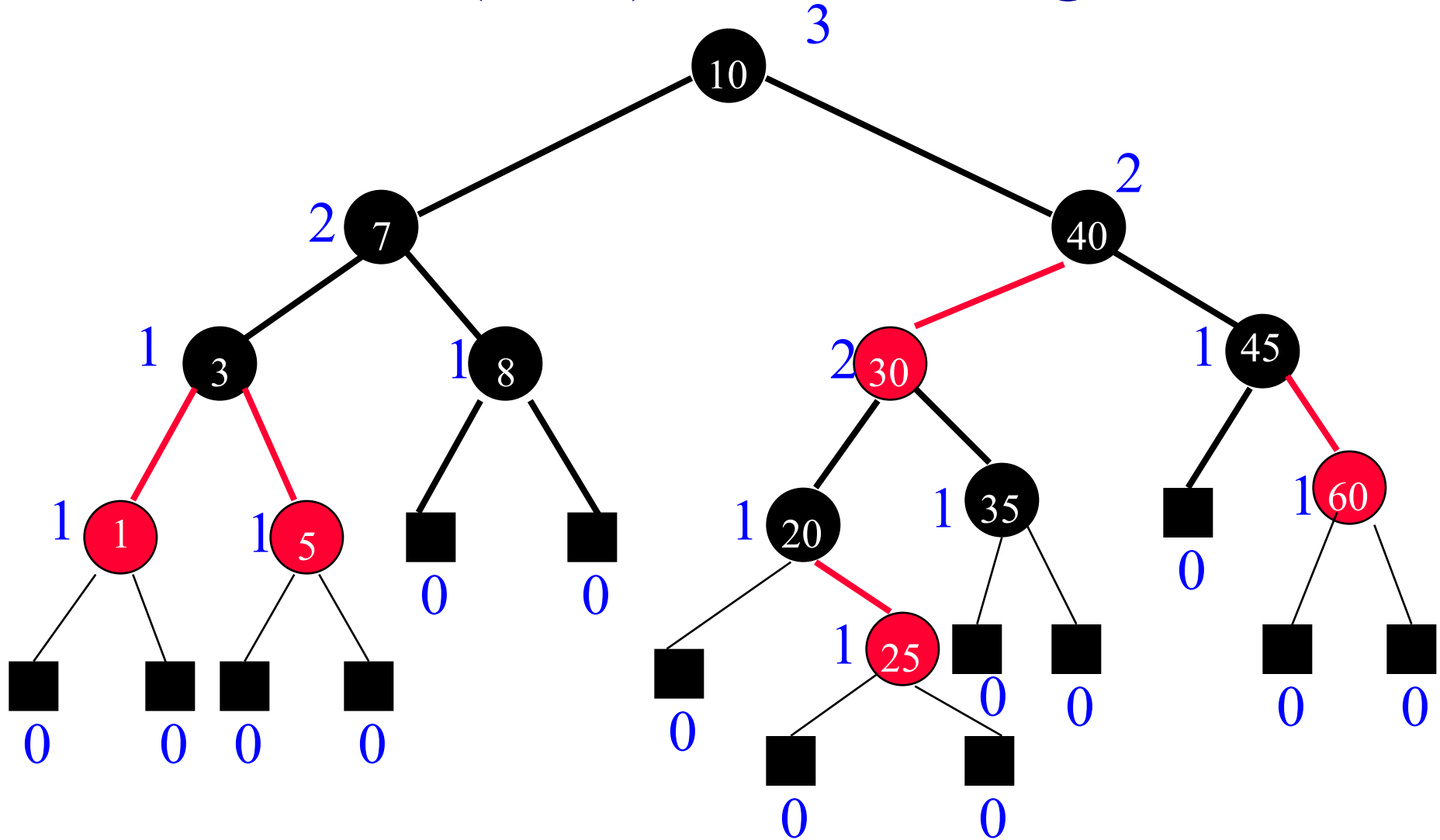


# Relationship Between $\text{rank}()$ And Color



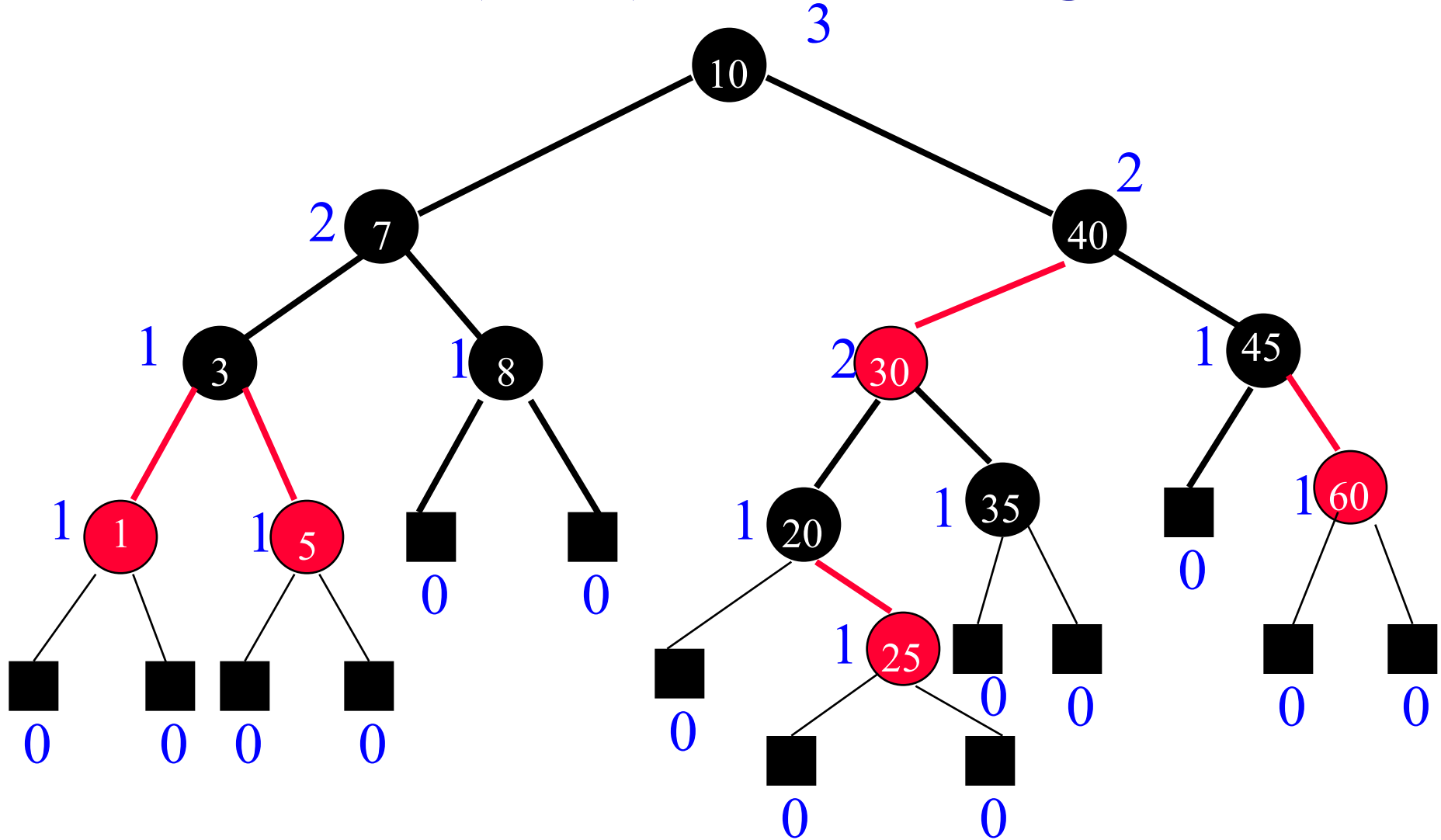
- Given  $\text{rank}(\text{root})$  and node/pointer colors, remaining ranks may be computed on way down.

# rank(root) & tree height



- Height  $\leq 2 * \text{rank}(\text{root})$ .

# rank(root) & tree height



- No external nodes at levels 1, 2, ..., rank(root).

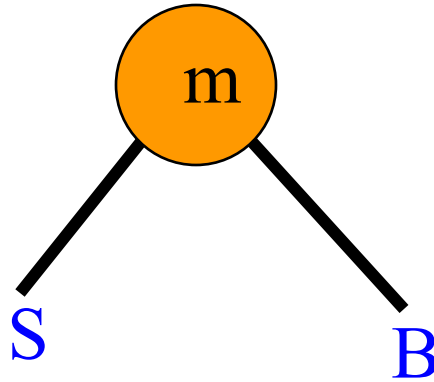
# rank(root) & tree height

- No external nodes at levels 1, 2, ..., rank(root).
  - So, #nodes  $\geq \sum_{1 \leq i \leq \text{rank}(\text{root})} 2^{i-1} = 2^{\text{rank}(\text{root})} - 1$ .
  - So, rank(root)  $\leq \log_2(n+1)$ .
- So, height(root)  $\leq 2\log_2(n+1)$ .

# Join(S,m,B)

- Input
  - Dictionary **S** of pairs with small keys.
  - Dictionary **B** of pairs with big keys.
  - An additional pair **m**.
  - All keys in **S** are smaller than **m.key**.
  - All keys in **B** are bigger than **m.key**.
- Output
  - A dictionary that contains all pairs in **S** and **B** plus the pair **m**.
  - Dictionaries **S** and **B** may be destroyed.

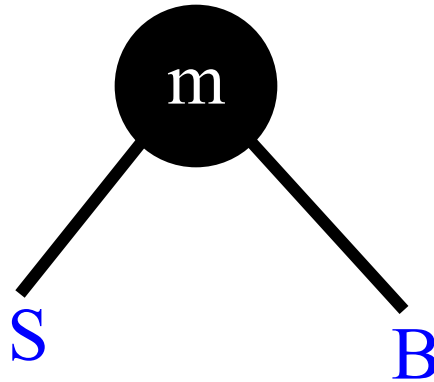
# Join Binary Search Trees



- $O(1)$  time.

# Join Red-black Trees

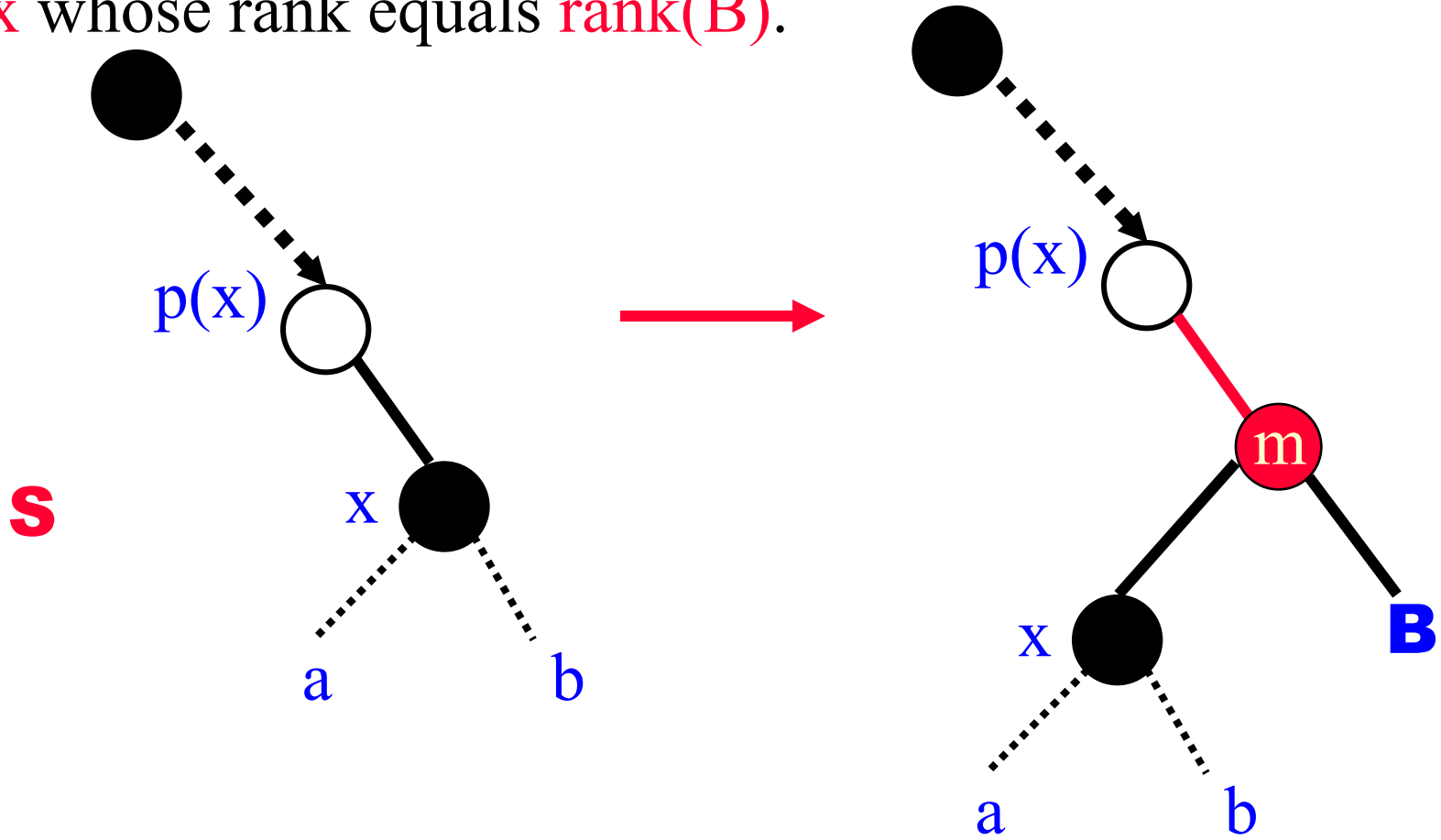
- When  $\text{rank}(S) = \text{rank}(B)$ , use binary search tree method.



- $\text{rank}(\text{root}) = \text{rank}(S) + 1 = \text{rank}(B) + 1.$

$$\text{rank}(S) > \text{rank}(B)$$

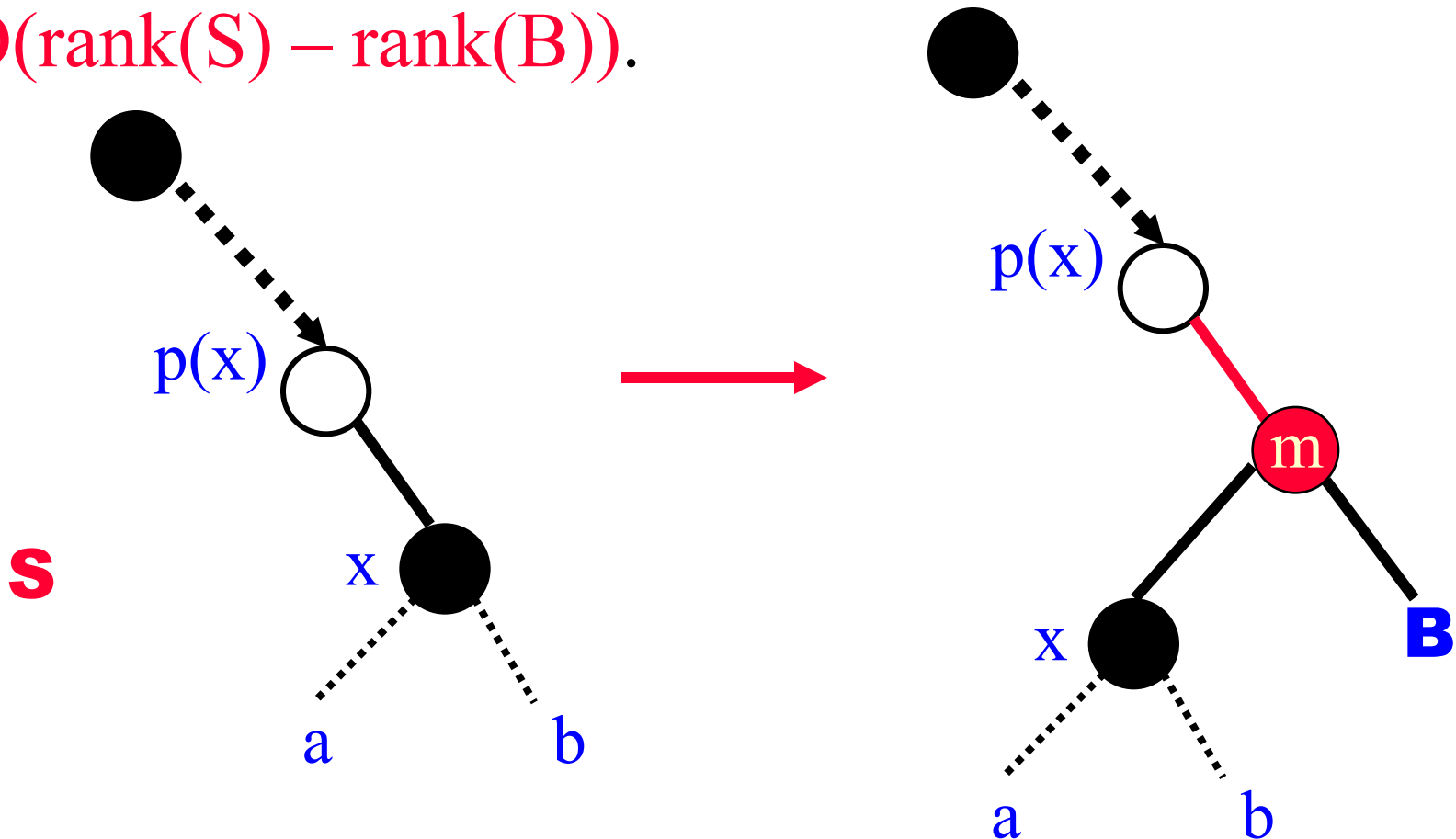
- Follow right child pointers from root of **S** to first node **x** whose rank equals **rank(B)**.





$$\text{rank}(S) > \text{rank}(B)$$

- If there are now **2** consecutive red pointers/nodes, perform bottom-up rebalancing beginning at **m**.
- $O(\text{rank}(S) - \text{rank}(B))$ .



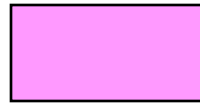
$$\text{rank}(S) < \text{rank}(B)$$

- Follow left child pointers from root of **B** to first node **x** whose rank equals  $\text{rank}(S)$ .
- Similar to case when  $\text{rank}(S) > \text{rank}(B)$ .

# Split(k)

- Inverse of join.
- Obtain
  - **S** ... dictionary of pairs with key  $< k$ .
  - **B** ... dictionary of pairs with key  $> k$ .
  - **m** ... pair with key  $= k$  (if present).

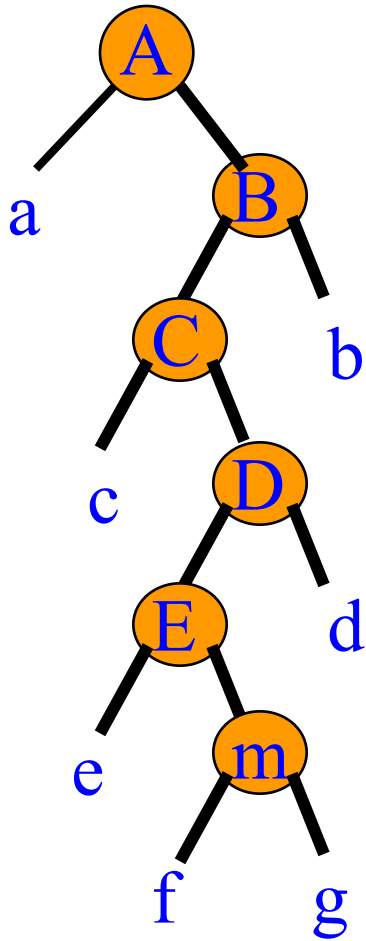
# Split A Binary Search Tree



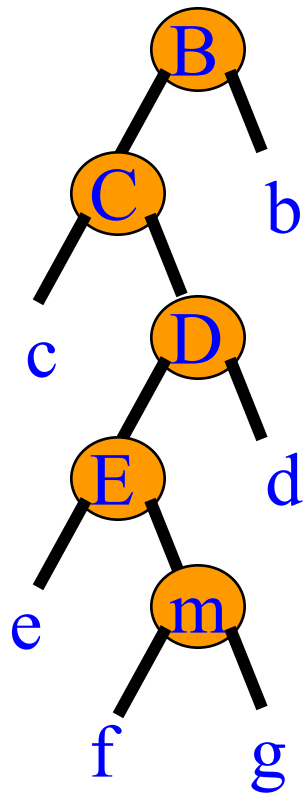
S



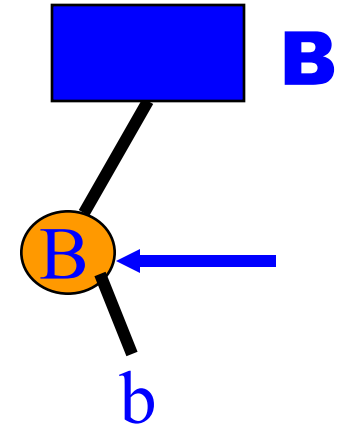
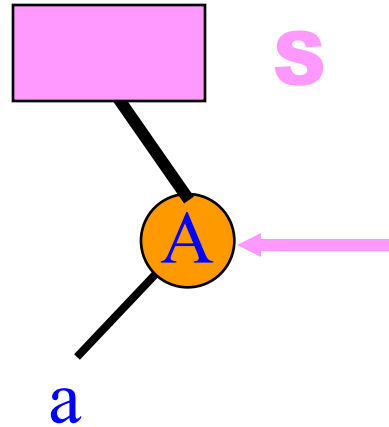
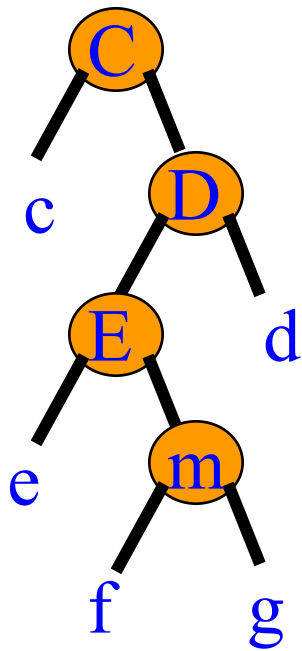
B



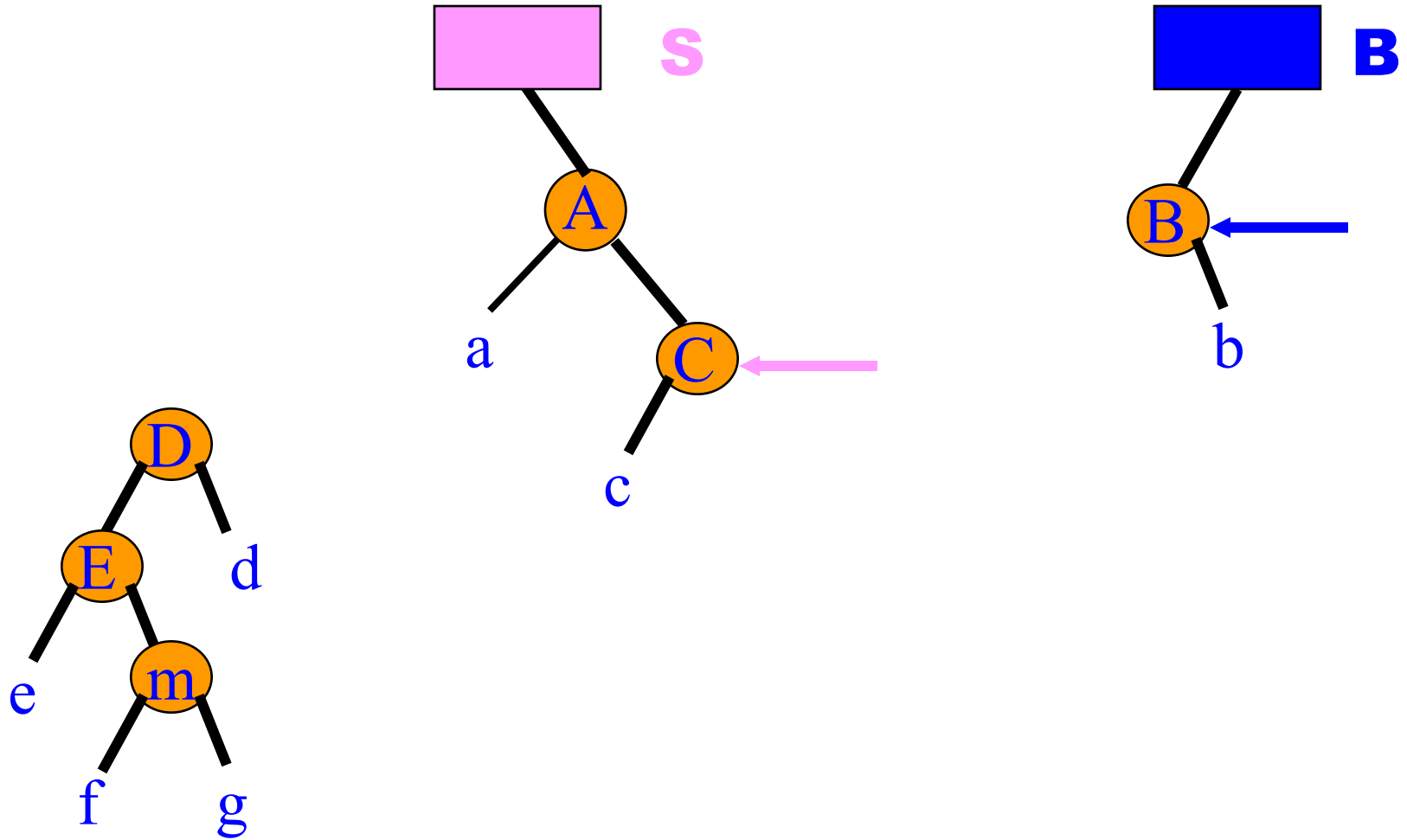
# Split A Binary Search Tree



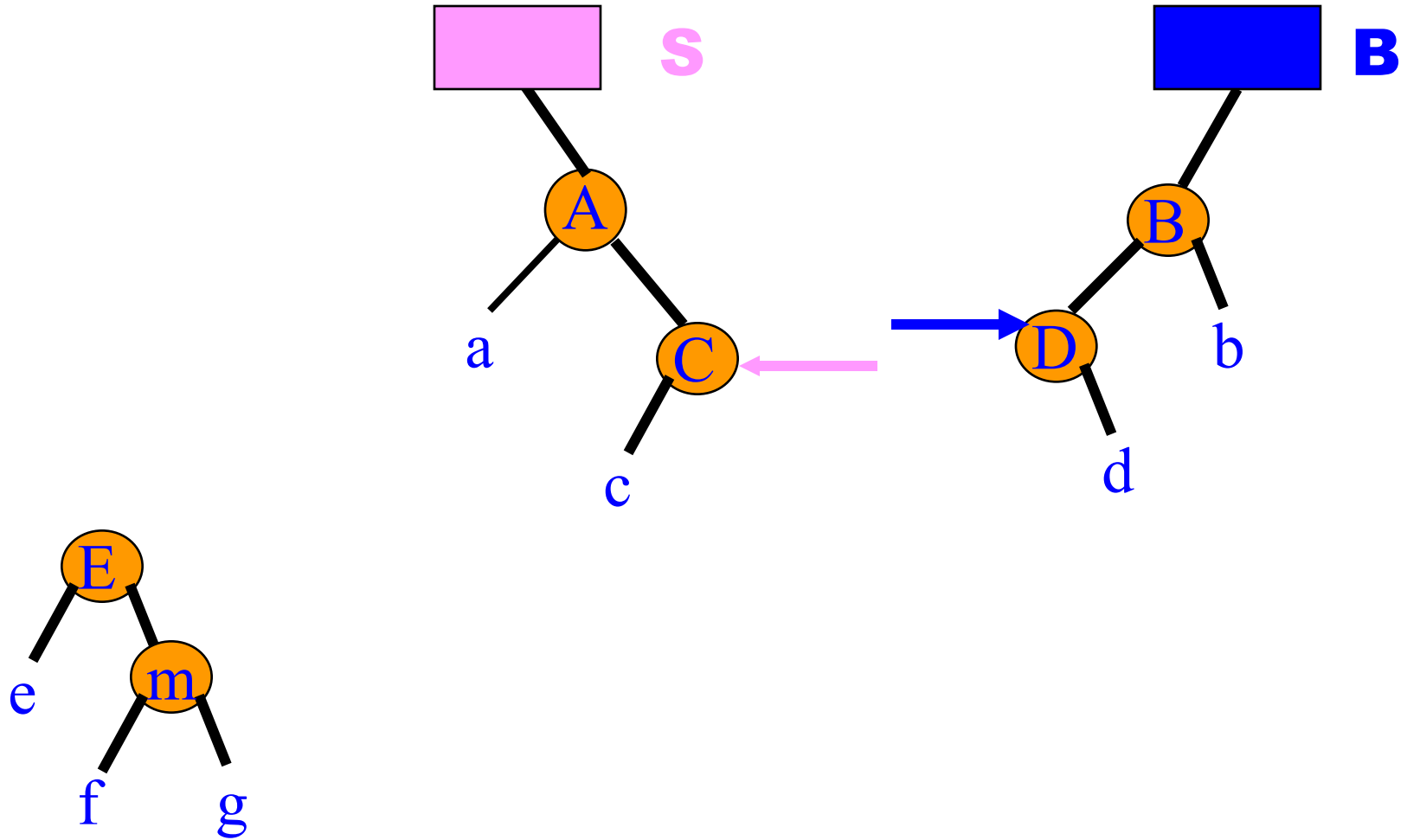
# Split A Binary Search Tree



# Split A Binary Search Tree

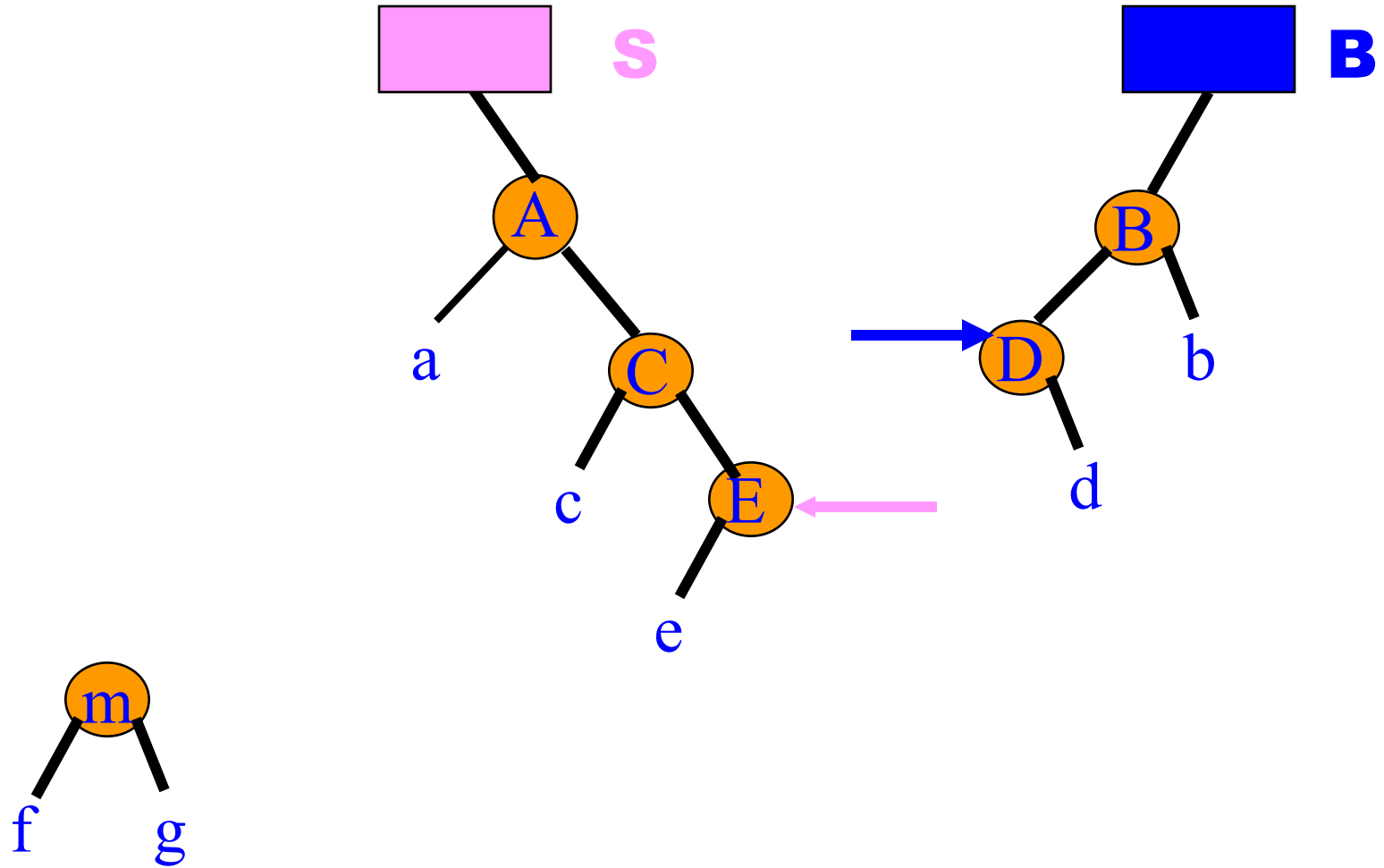


# Split A Binary Search Tree

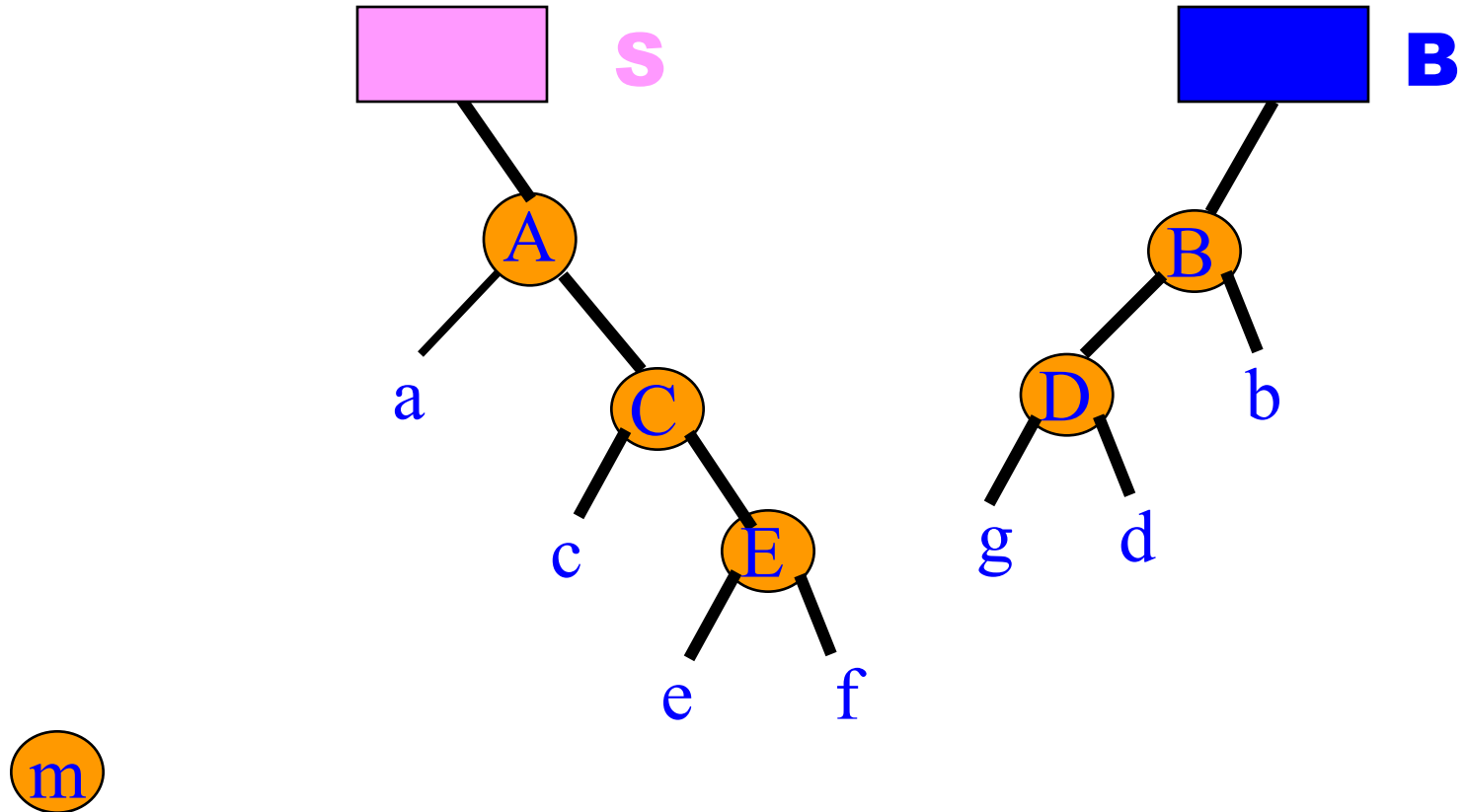




# Split A Binary Search Tree



# Split A Binary Search Tree



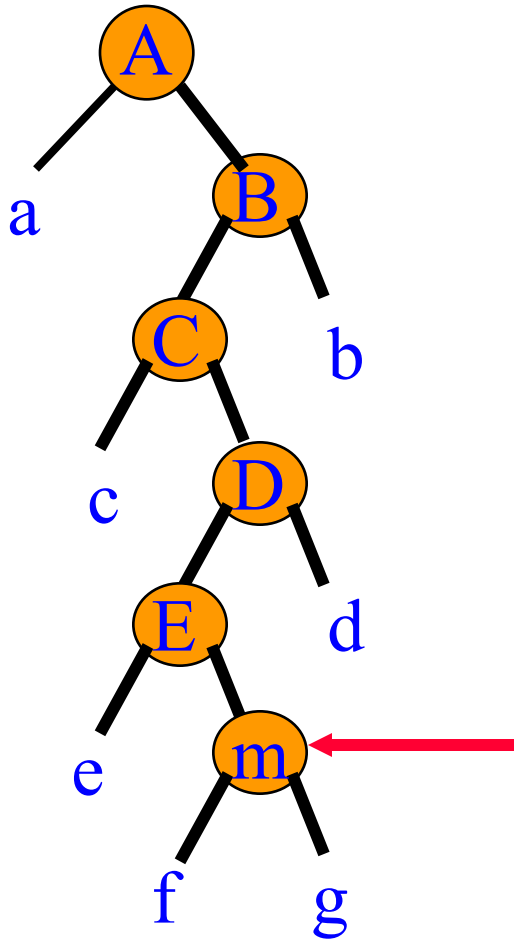
# Split A Red-Black Tree

- Previous strategy does not split a red-black tree into two red-black trees.
- Must do a search for **m** followed by a traceback to the root.
- During the traceback use the join operation to construct **S** and **B**.

# Split A Red-Black Tree

**S** = f

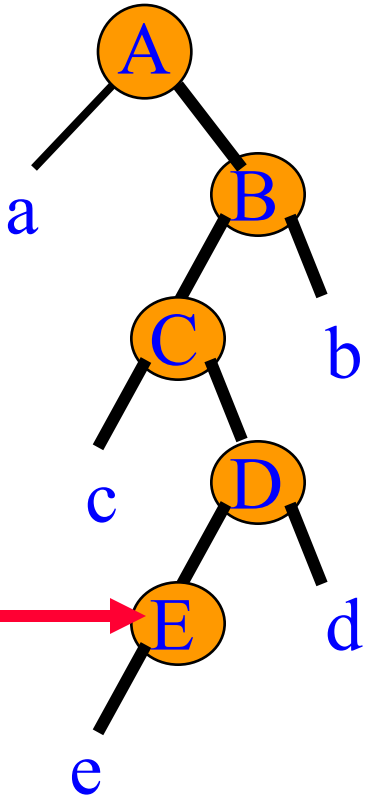
**B** = g



# Split A Red-Black Tree

**S** = f

**B** = g

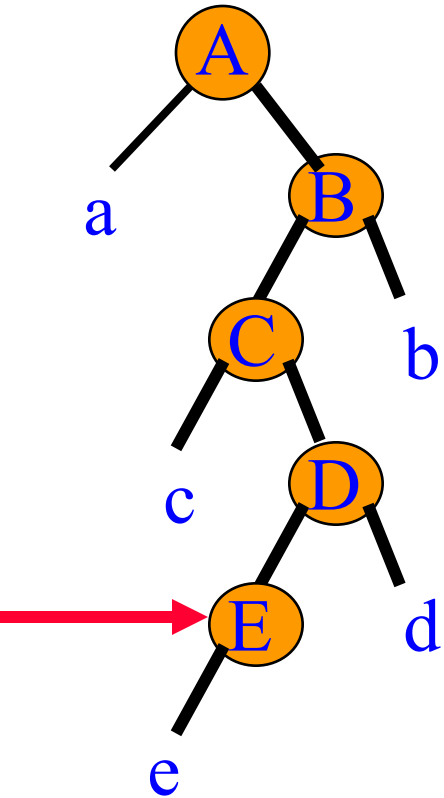


# Split A Red-Black Tree

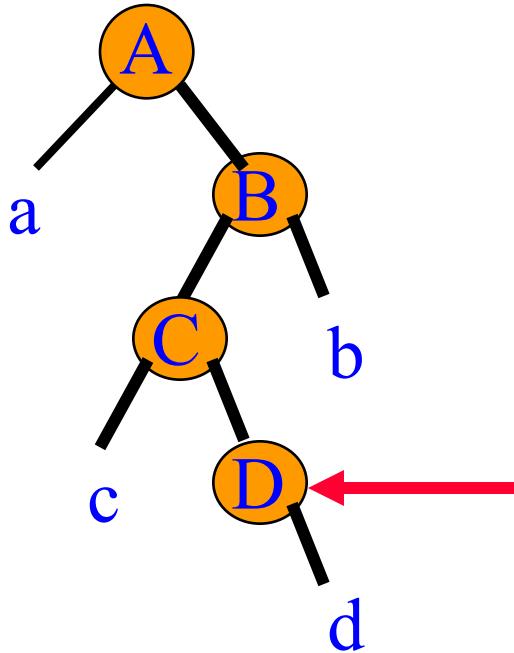
**S** = f

**B** = g

**S** = join(e, E, **S**)



# Split A Red-Black Tree



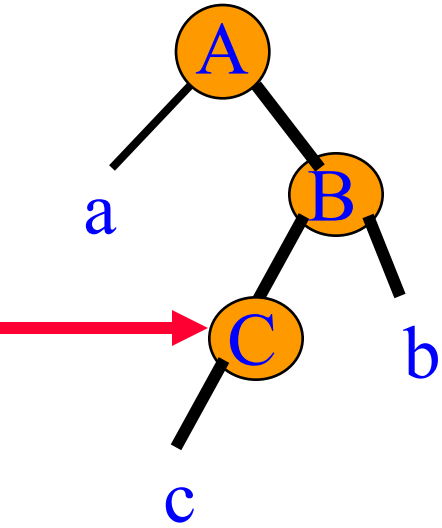
**S** = f

**B** = g

**S** = join(e, E, **S**)

**B** = join(**B**, D, d)

# Split A Red-Black Tree



**S** = f

**B** = g

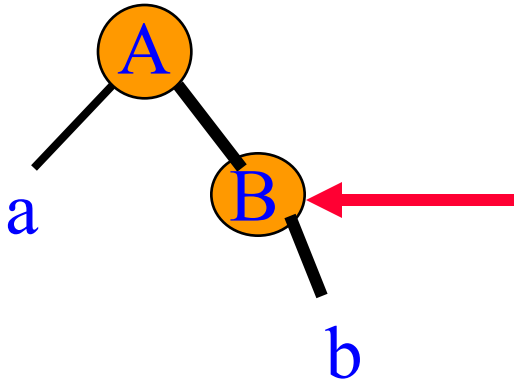
**S** = join(e, E, **S**)

**B** = join(**B**, D, d)

**S** = join(c, C, **S**)



# Split A Red-Black Tree



**S** = f

**B** = g

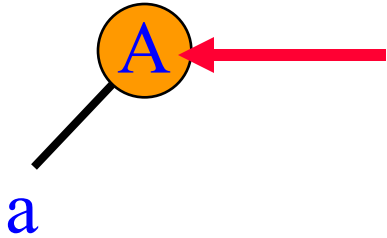
**S** = join(e, E, **S**)

**B** = join(**B**, D, d)

**S** = join(c, C, **S**)

**B** = join(**B**, B, b)

# Split A Red-Black Tree



**S** = f

**B** = g

**S** = join(e, E, **S**)

**B** = join(**B**, D, d)

**S** = join(c, C, **S**)

**B** = join(**B**, B, b)

**S** = join(a, A, **S**)