Red Black Trees

Red Black Trees

Colored Nodes Definition

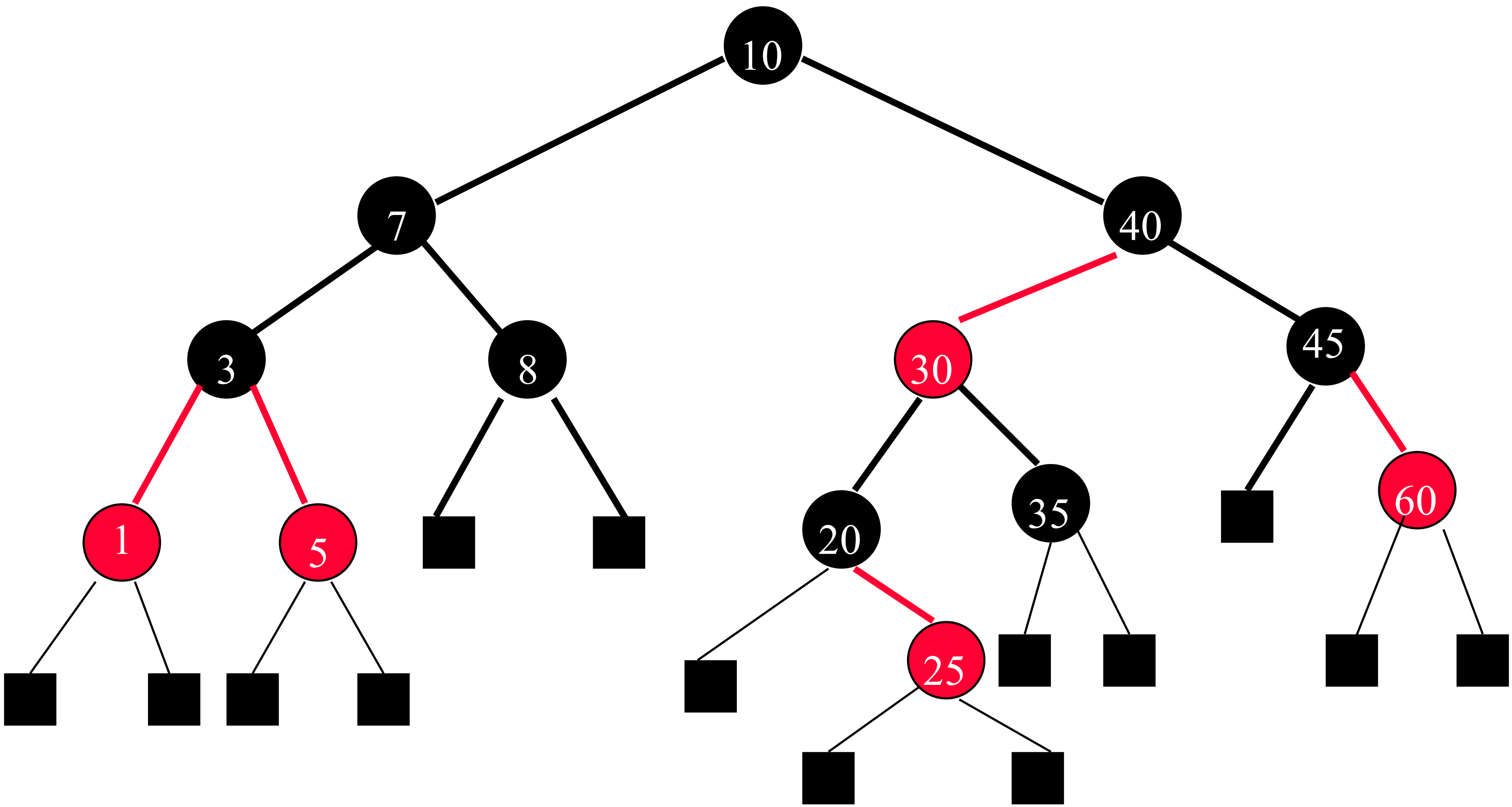
- Binary search tree.
- Each node is colored **red** or **black**.
- Root and all external nodes are **black**.
- No root-to-external-node path has two consecutive **red** nodes.
- All root-to-external-node paths have the same number of **black** nodes

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

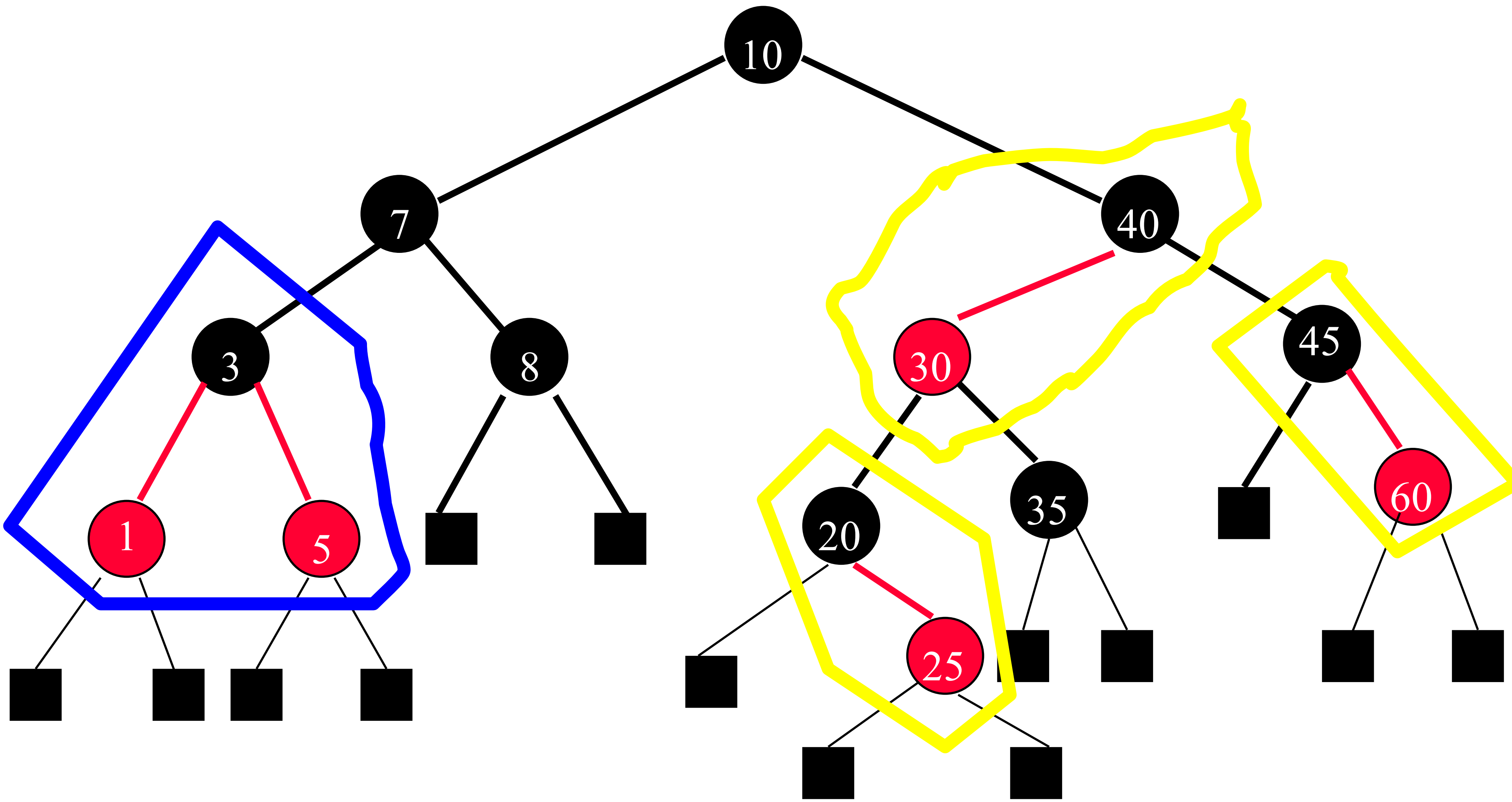
- The height of a red black tree that has n (internal) nodes is between $\log_2(n+1)$ and $2\log_2(n+1)$.

How can we prove this?

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 $h' >= h/2$ 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 $>= 2^{h'}-1$.
- So, $n >= 2^{h'}-1$
- So, $h <= 2 \log_2 (n + 1)$

Properties

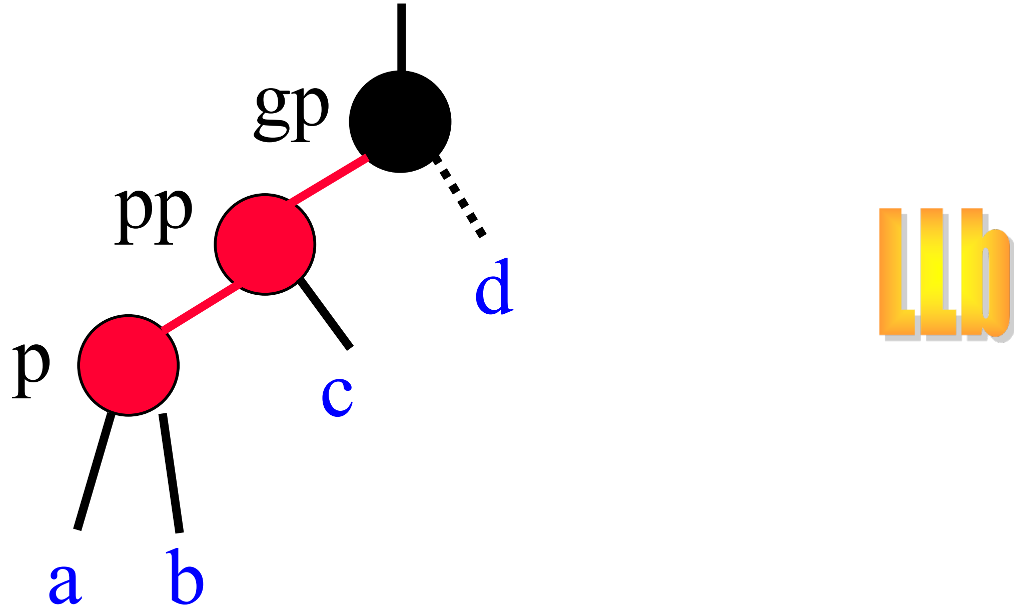
- At most $O(1)$ rotation and $O(\log n)$ color flips per insert/delete.

- Red-black trees are a clever binary representation of 2-3-4 trees.

Insert

- The new node is inserted just like binary search tree→ color to have red-black tree property
- New node color options.
  - **Black** node => one root-to-external-node path has an extra black node (black pointer).
    - Hard to remedy.
  - **Red** node => one root-to-external-node path may have two consecutive red nodes (pointers).
    - May be remedied by color flips and/or a rotation.

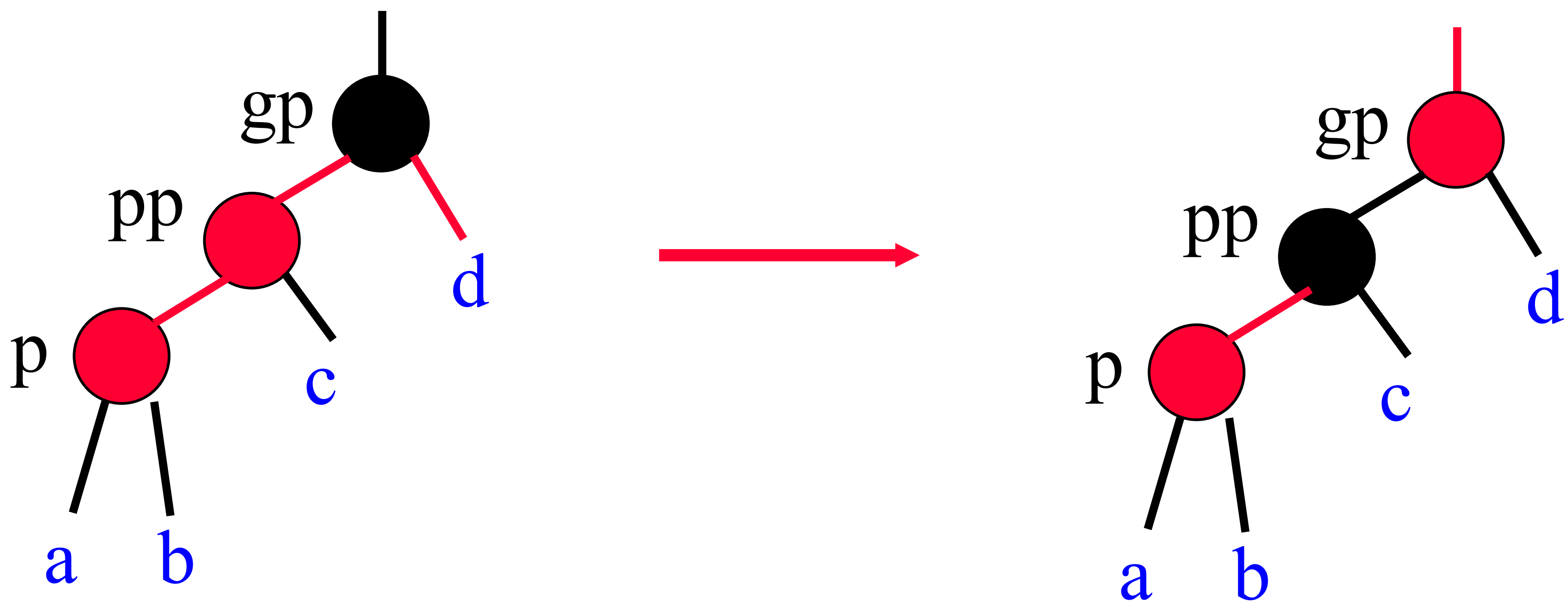
Classification Of 2 Red Nodes/Pointers



- 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

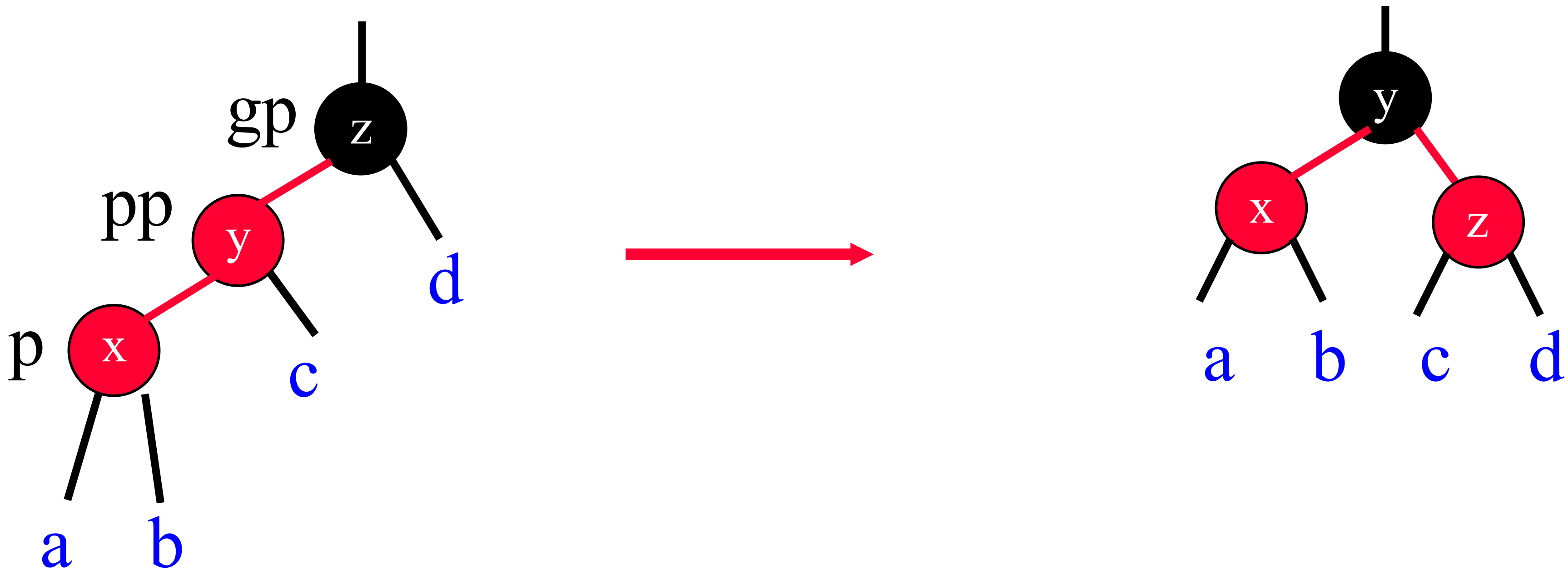
- Color flip.



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

LLb

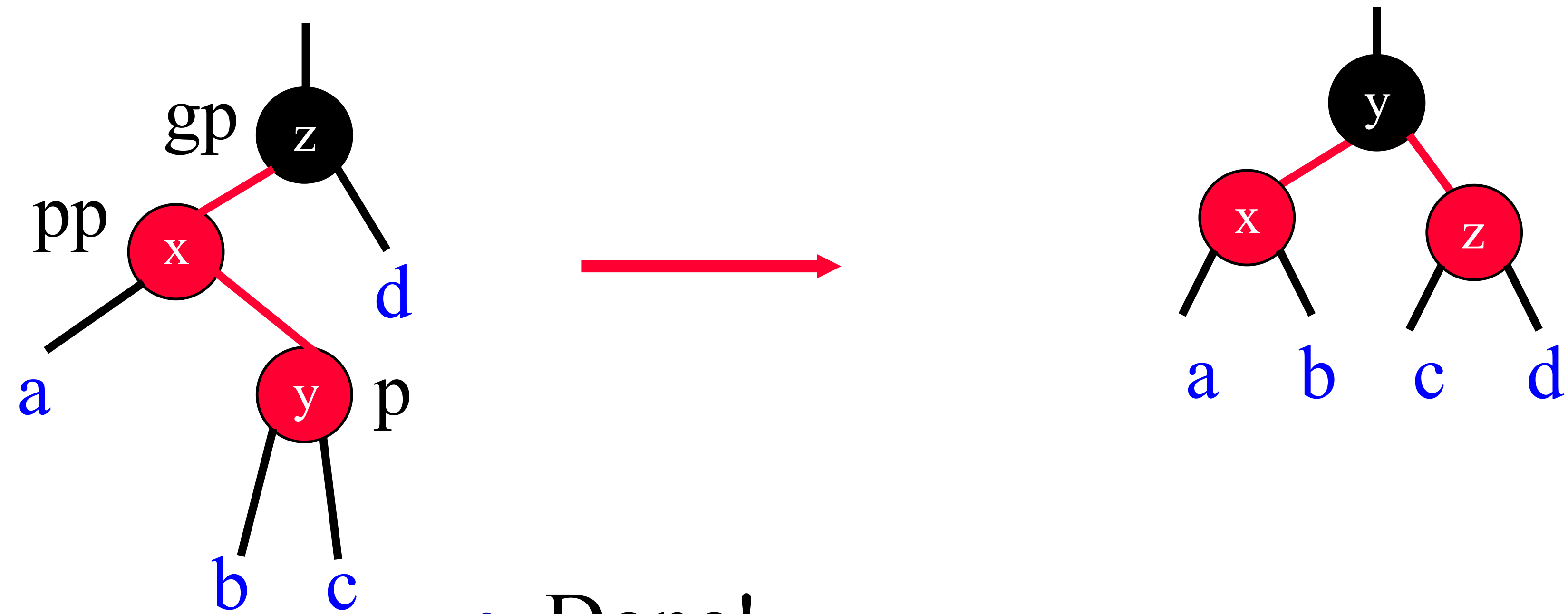
- Rotate.



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

LRb

- Rotate.

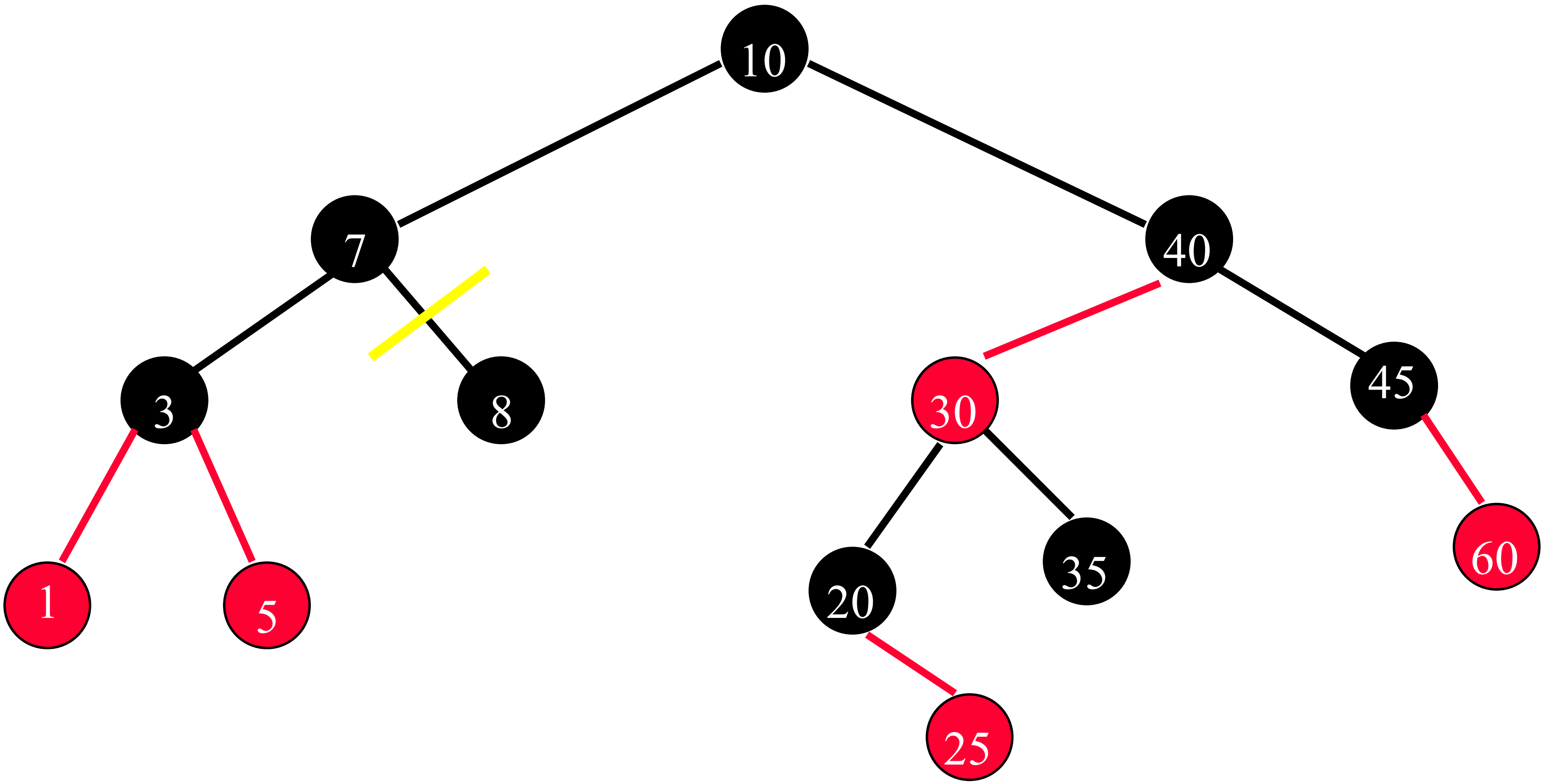


- Done!
- Same as LR rotation of AVL tree.
- RRb and RLb are symmetric.

Delete

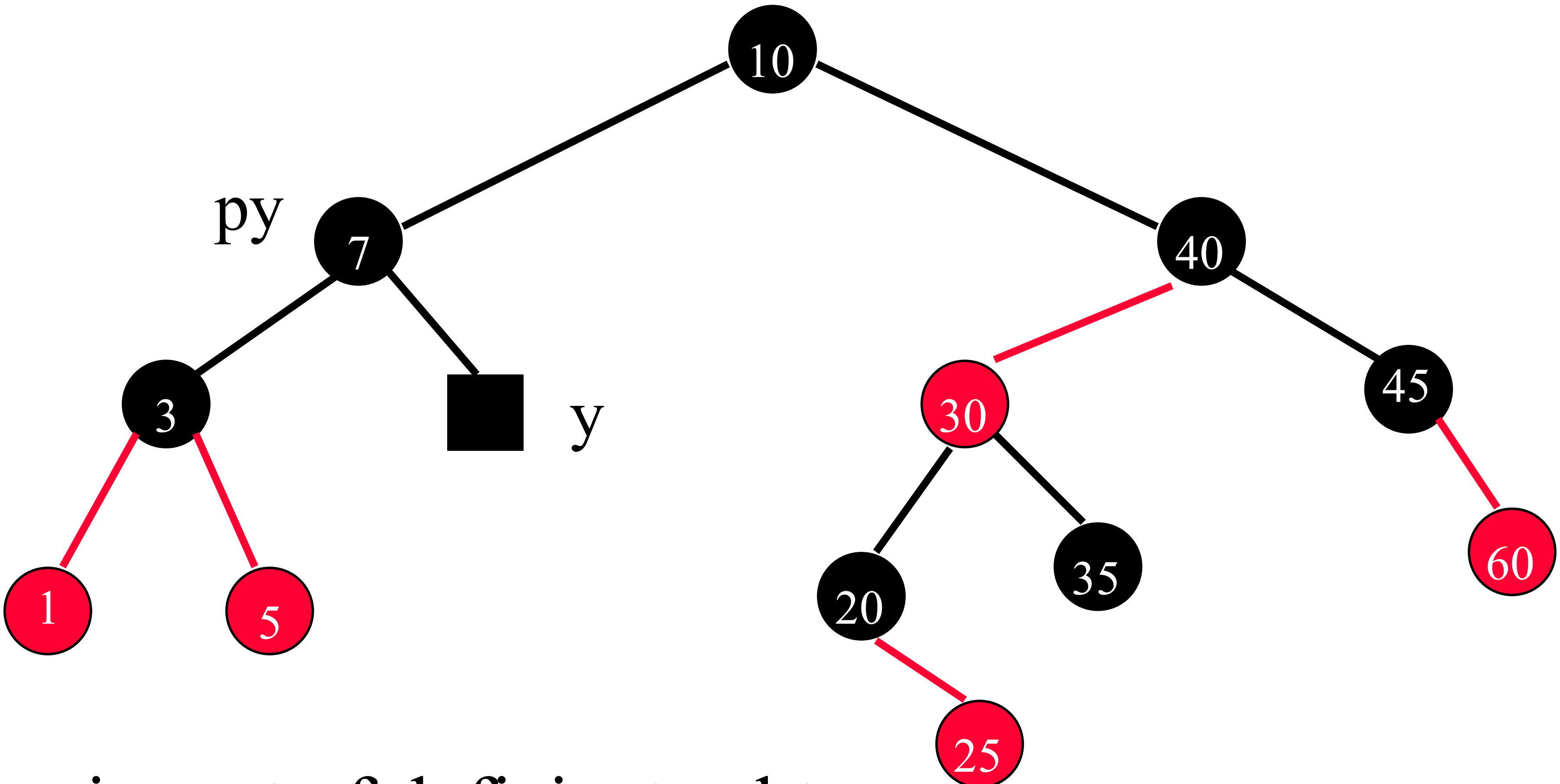
- First delete like unbalanced binary search tree.
  - We need to consider only case 0 (node with 0 child) and case 1 (node with 1 child) as case 2 reduces to case 0 or 1.
- If red node deleted, no rebalancing needed.
- If black node deleted, a subtree becomes one black pointer (node) deficient.

Delete A Black Leaf



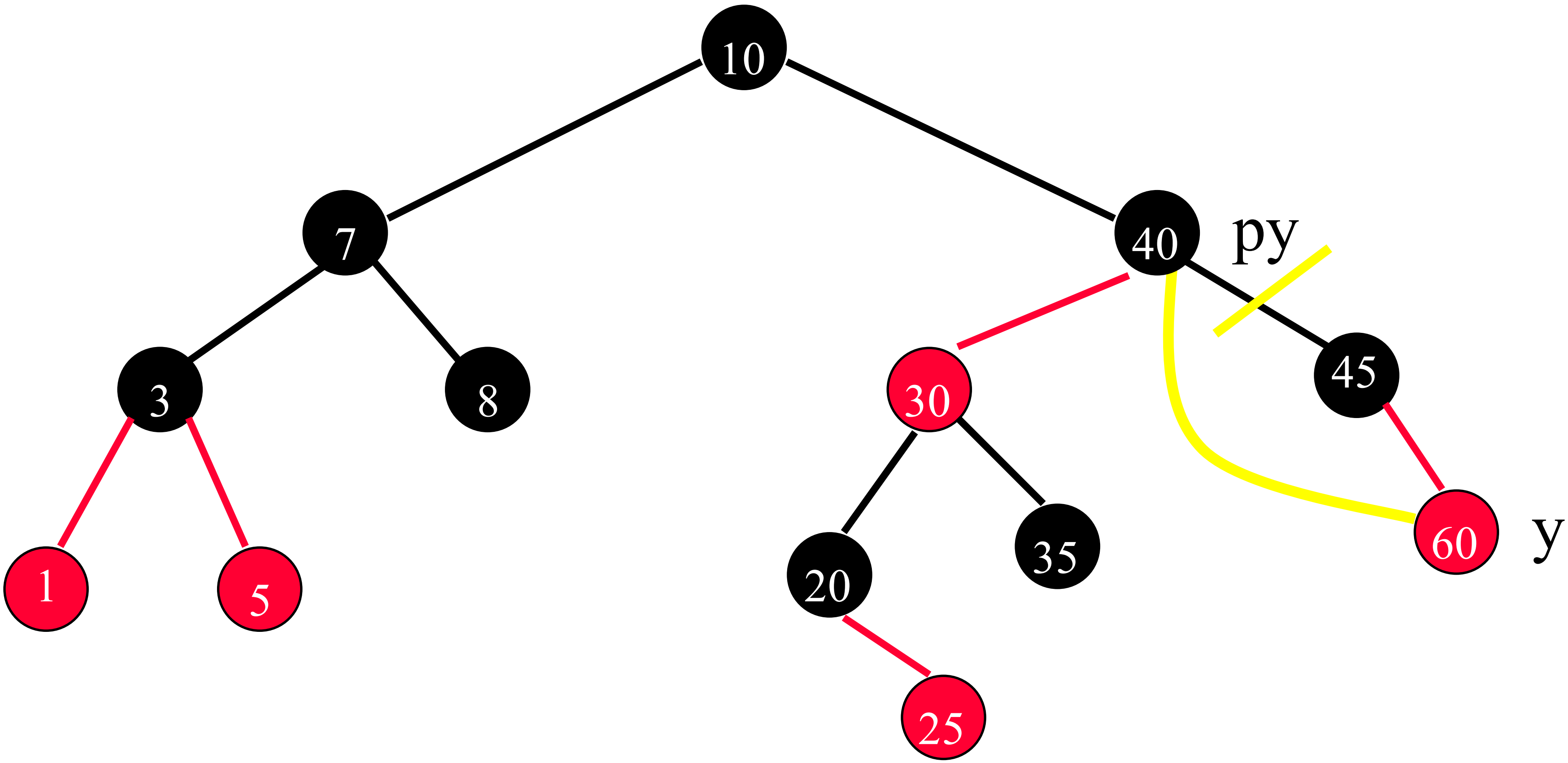
- Delete 8.

Delete A Black Leaf



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

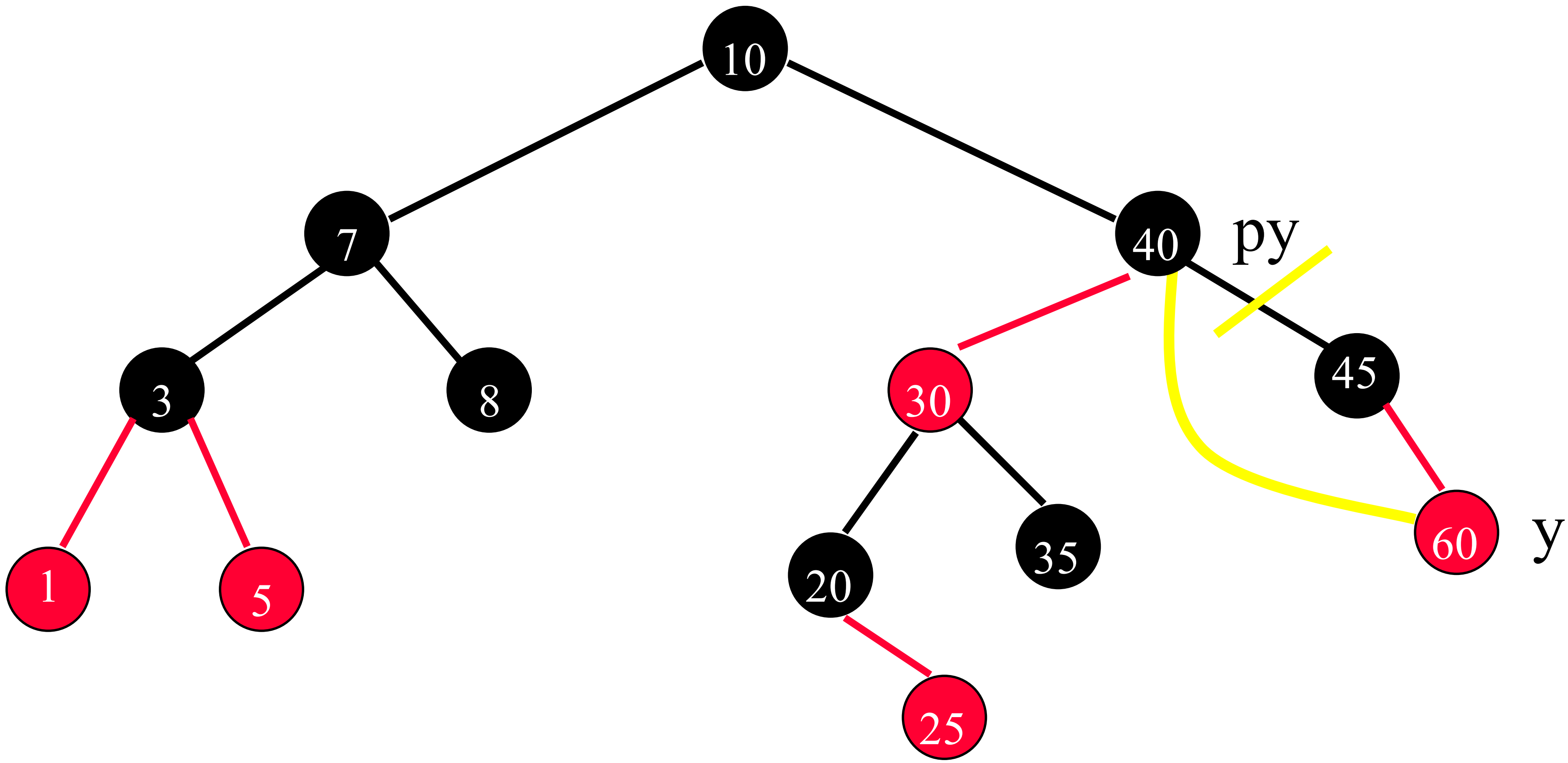
Delete A Black Degree 1 Node



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

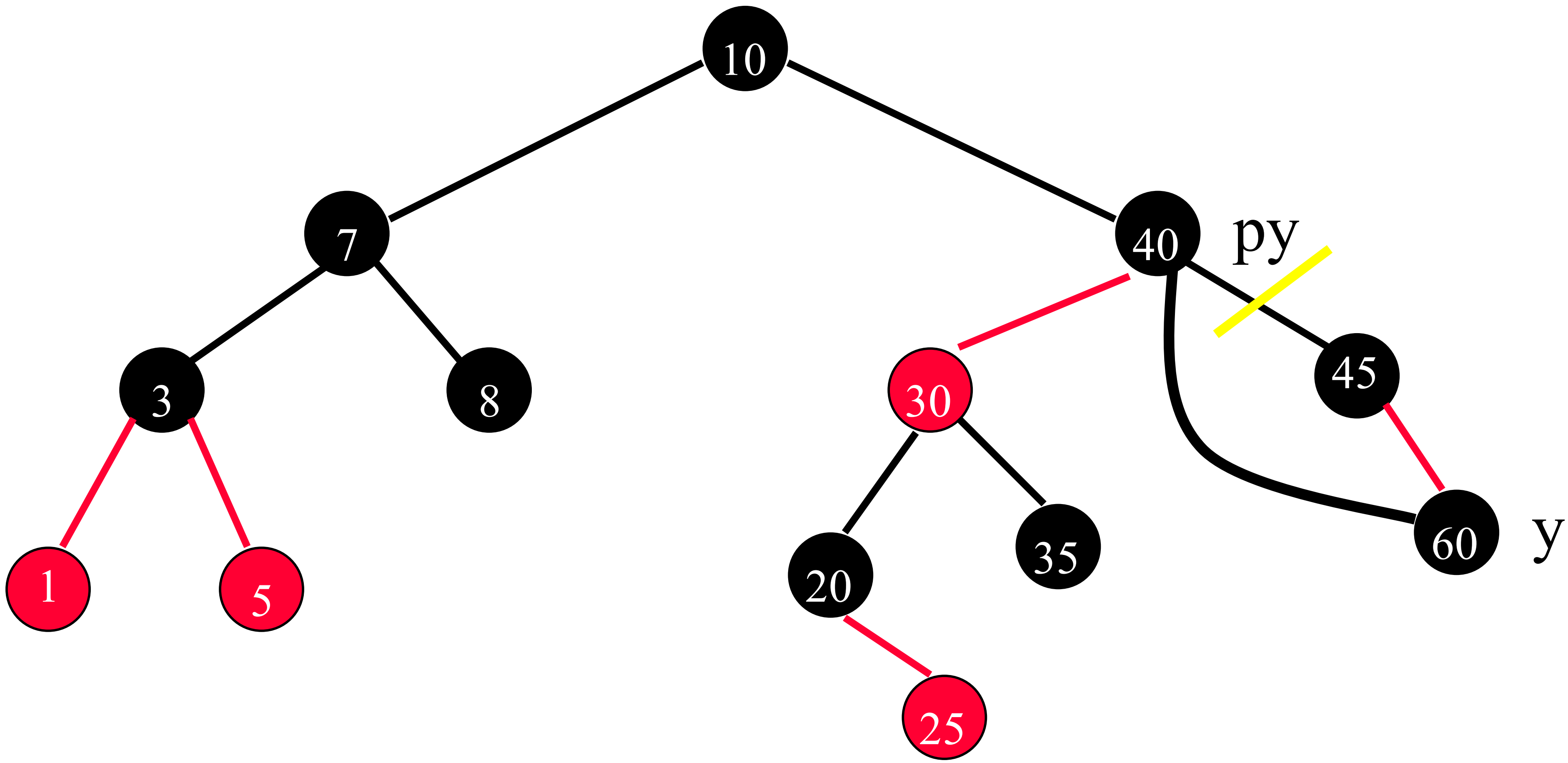
Rebalancing Strategy

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



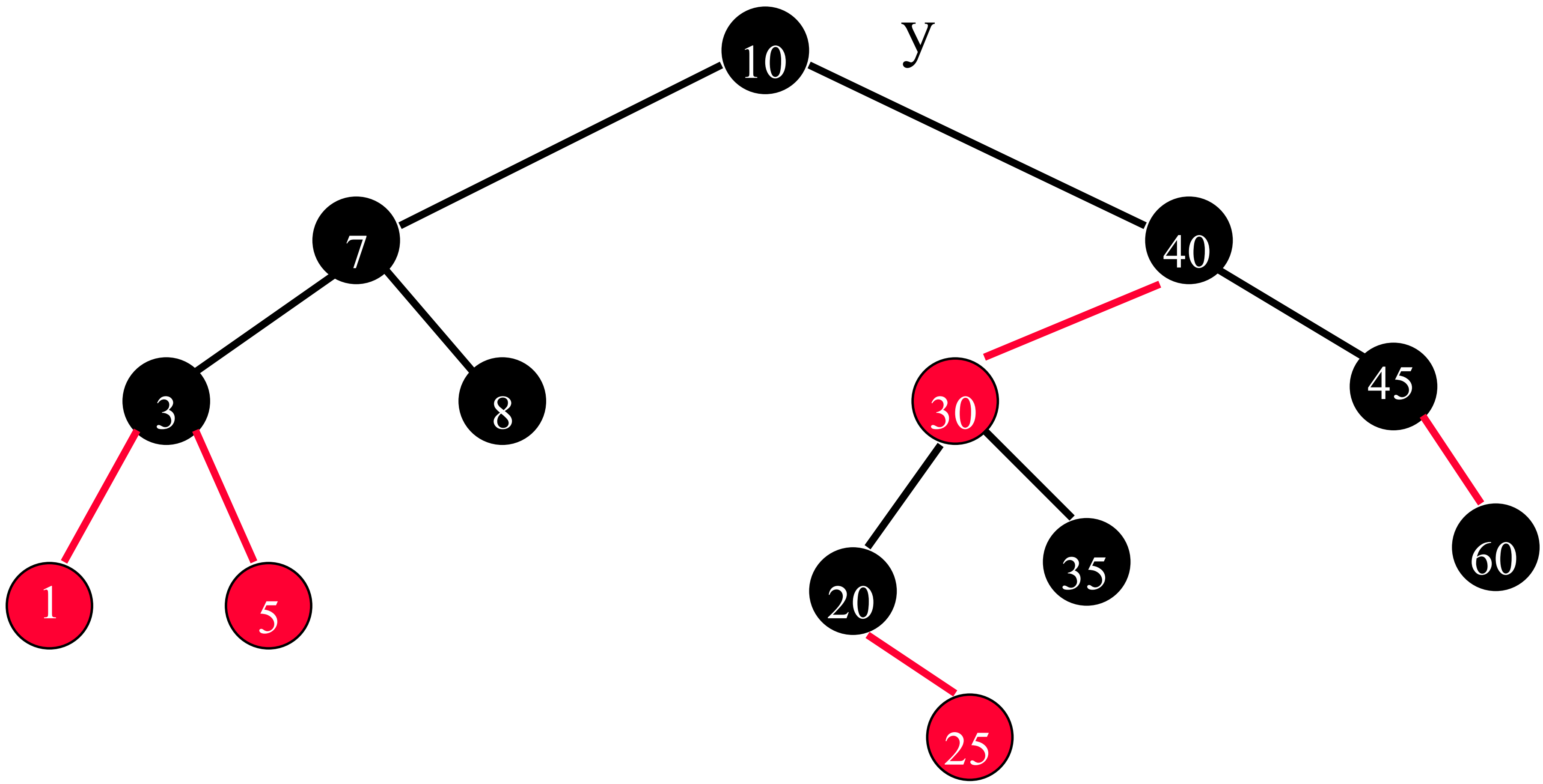
Rebalancing Strategy

- Now, no subtree is deficient. Done!



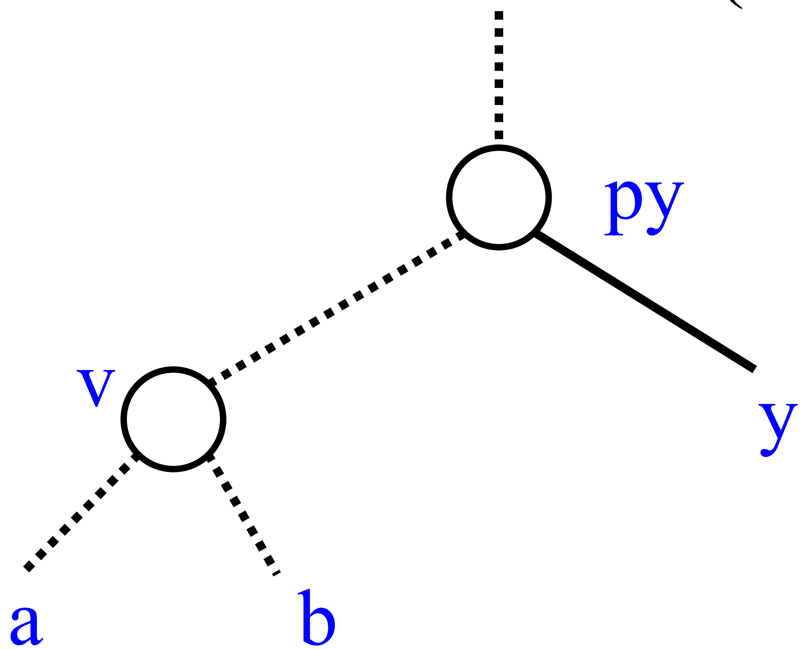
Rebalancing Strategy

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



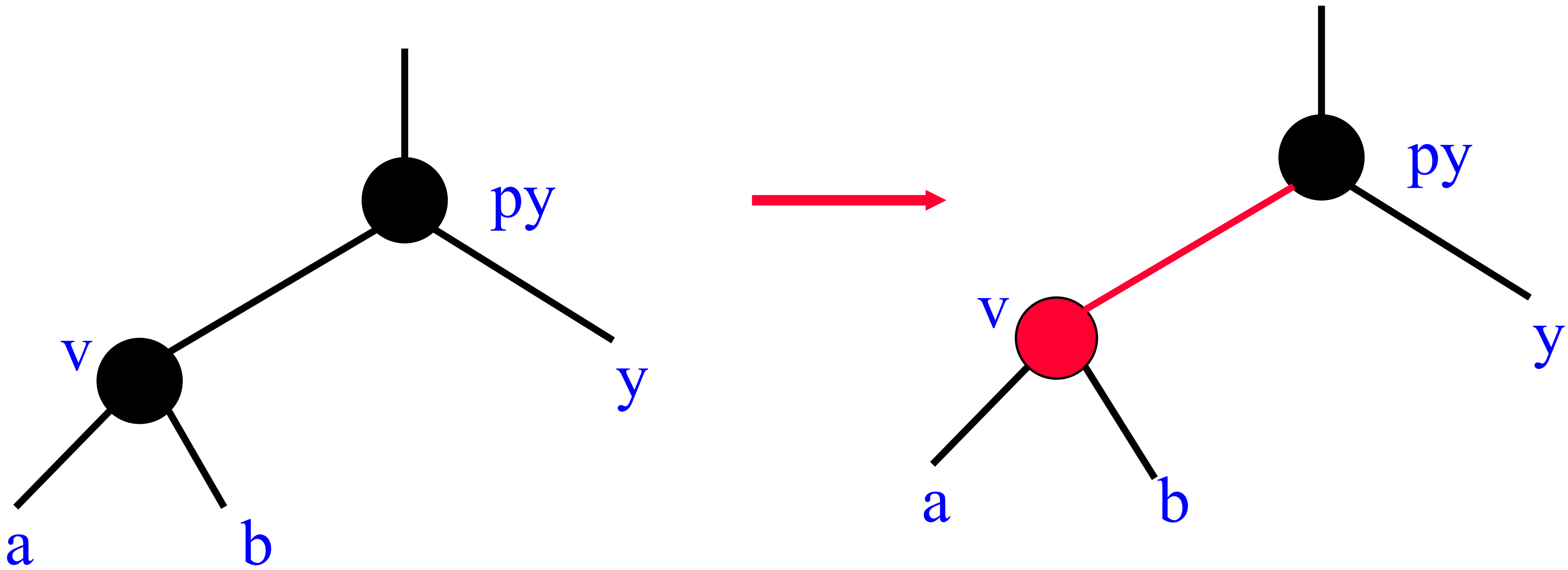
Rebalancing Strategy

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



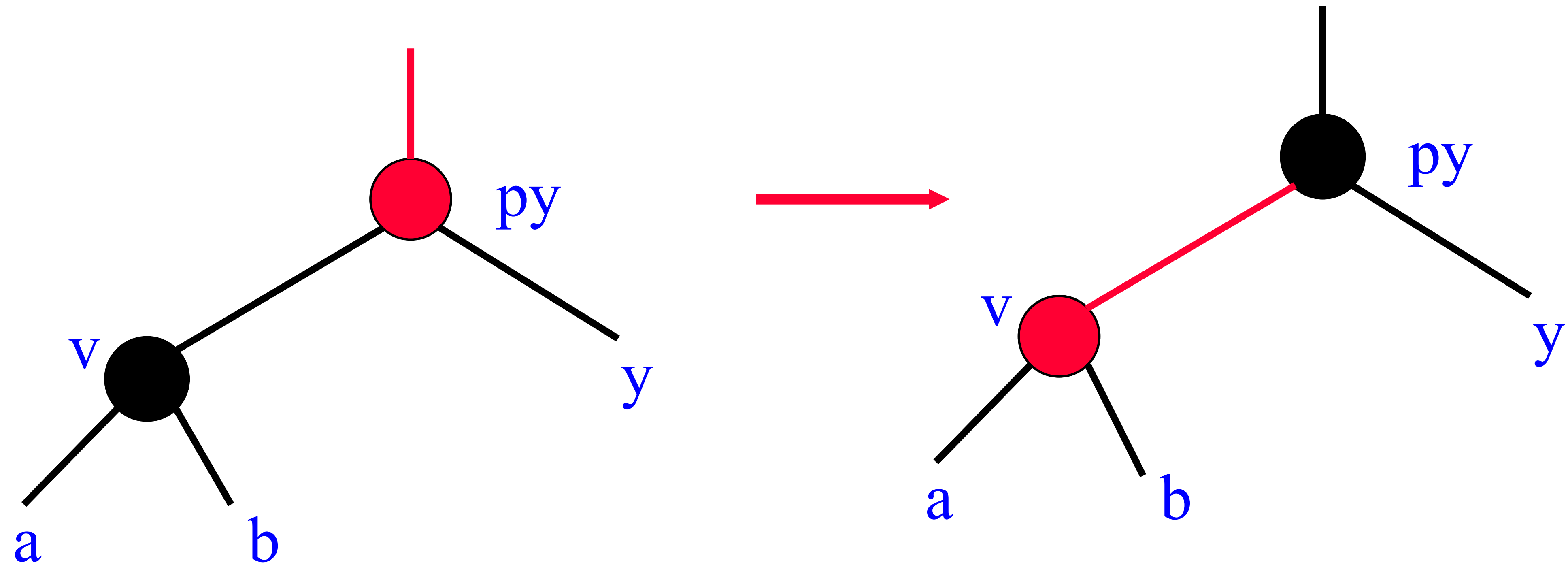
- Xcn
  - y is right child of py => X = R.
  - Pointer to v is black => c = b.
  - v has 1 red child => n = 1.

Rb0 (case 1, py is black)



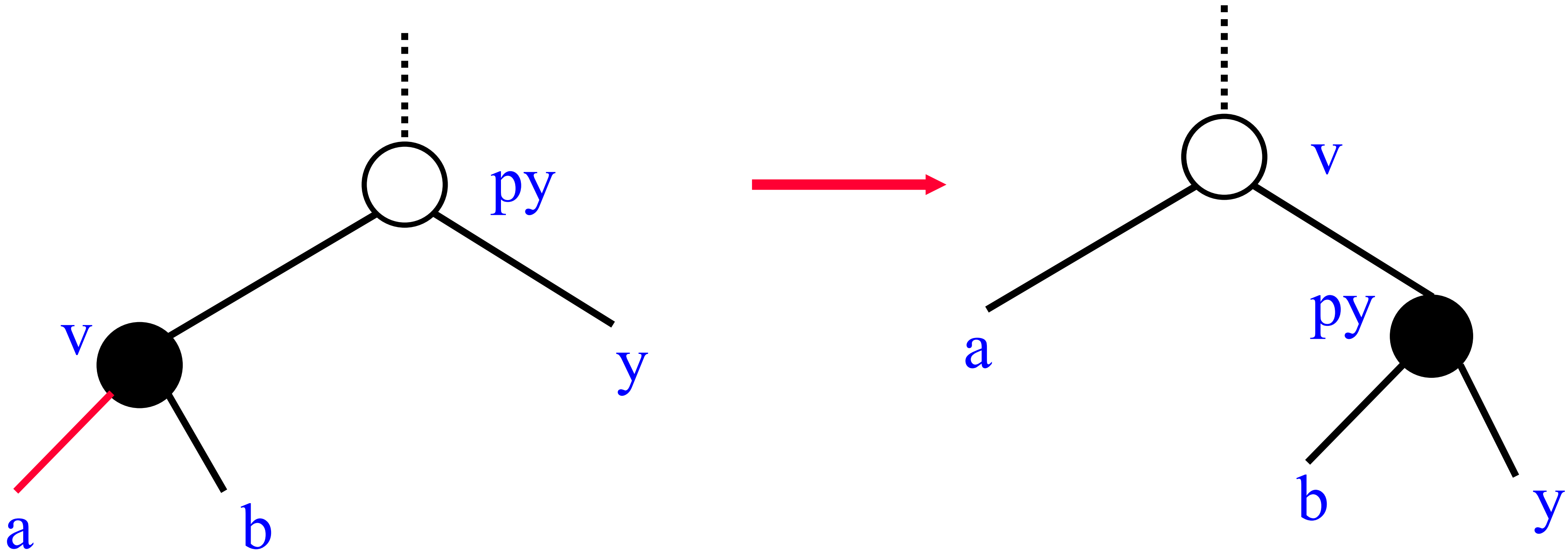
- Color change.
- Now, py is root of deficient subtree.
- Continue!

Rb0 (case 2, py is red)



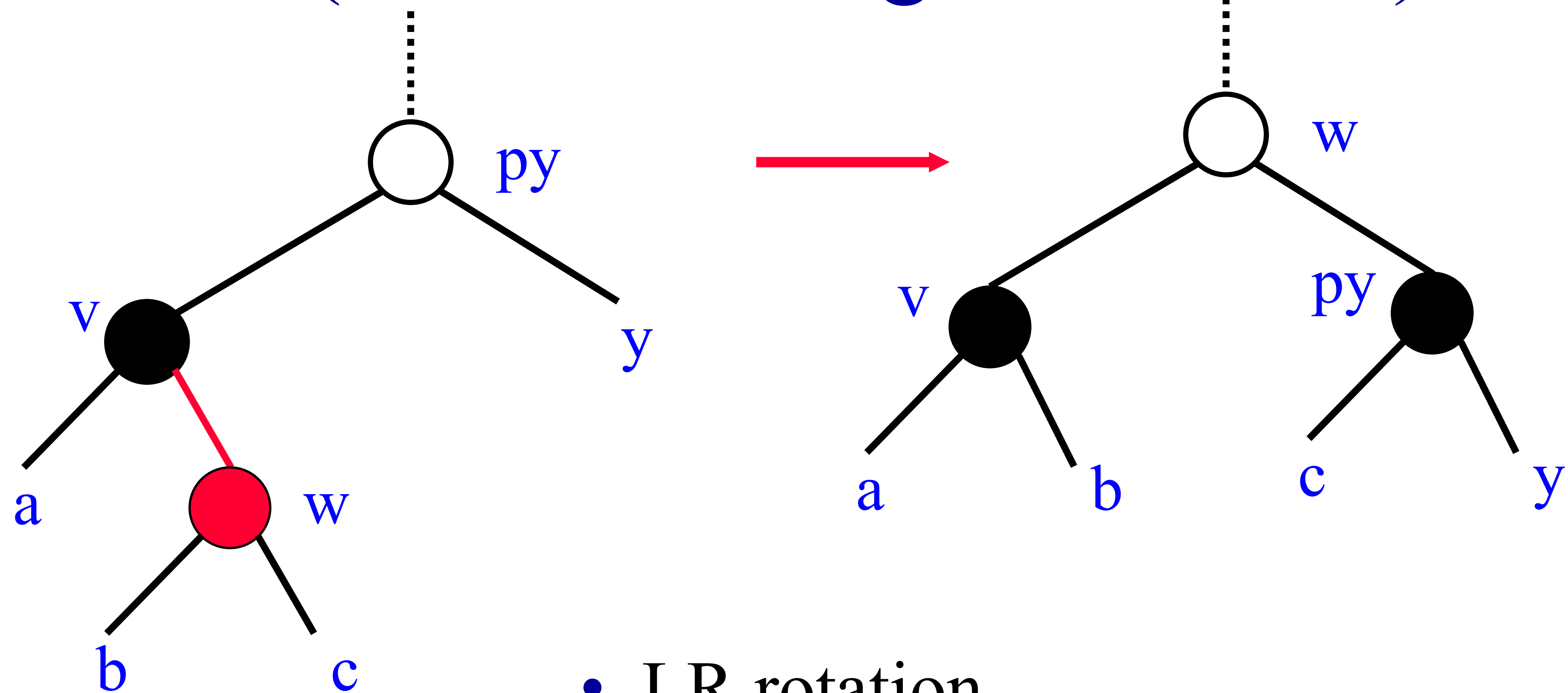
- Color change.
- Deficiency eliminated.
- Done!

Rb1 (case 1: v's left child red)



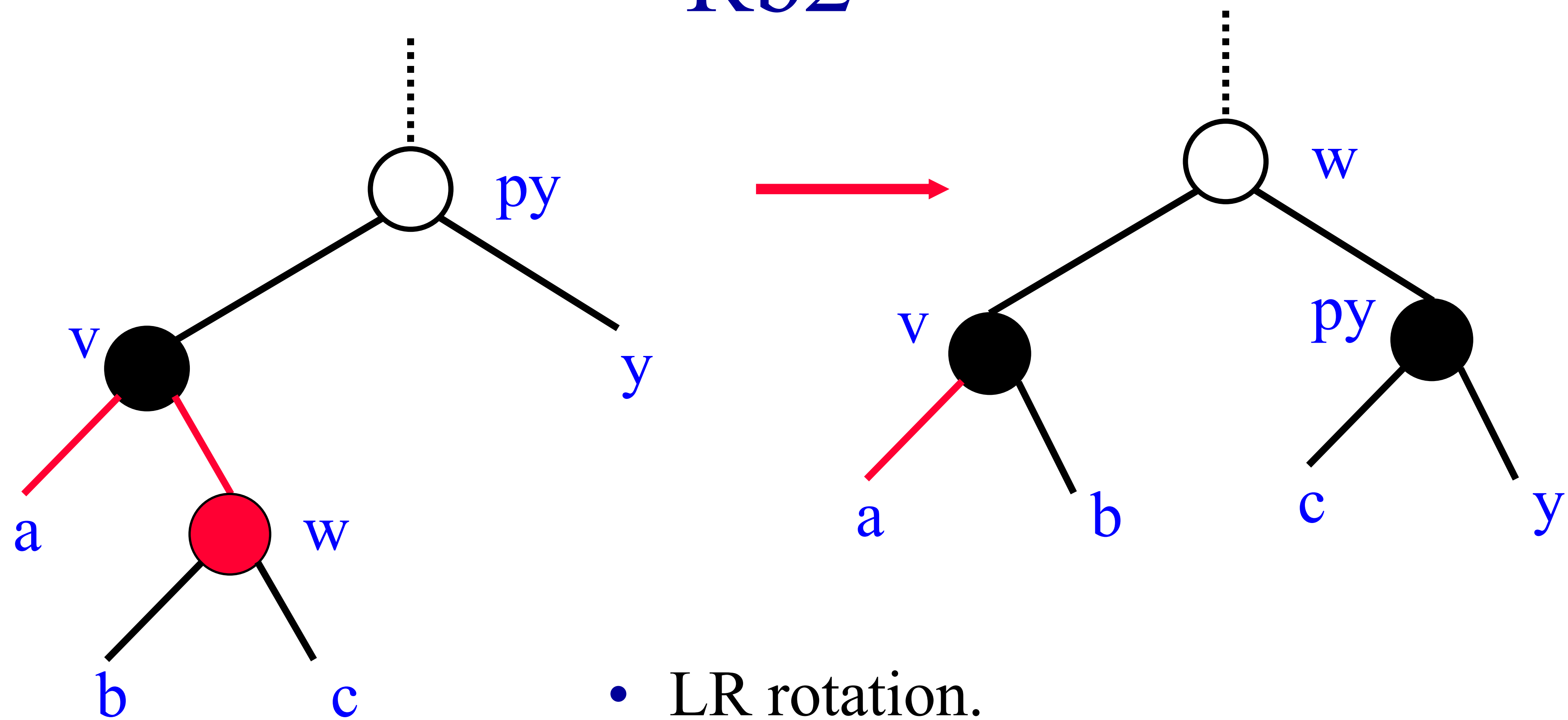
- LL rotation.
- Deficiency eliminated.
- Done!

Rb1 (case 2 : v's right child red)



- LR rotation.
- Deficiency eliminated.
- Done!

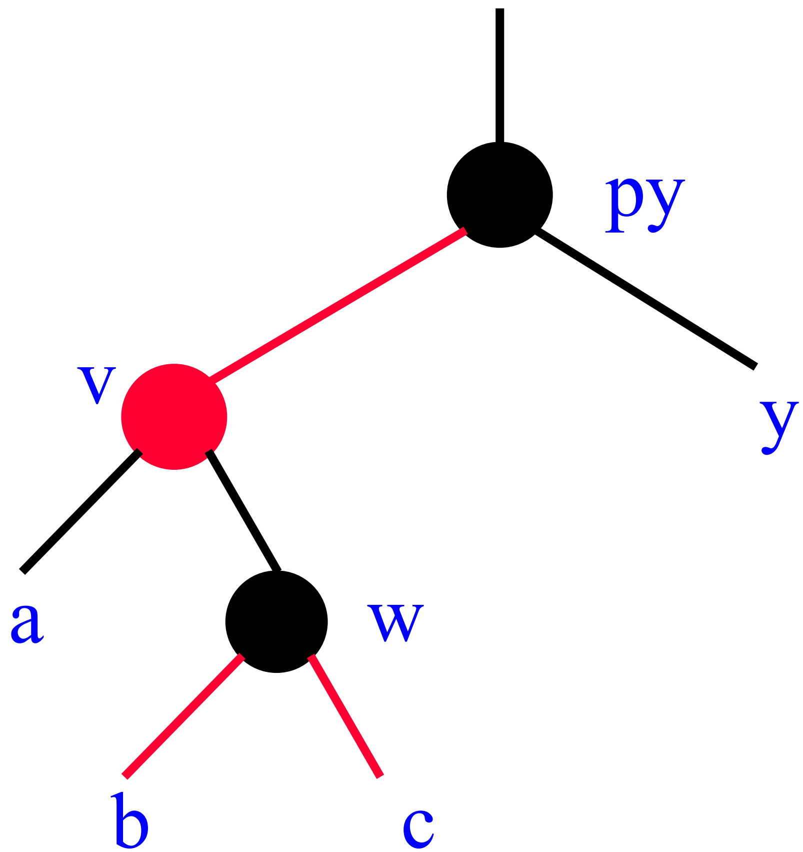
Rb2



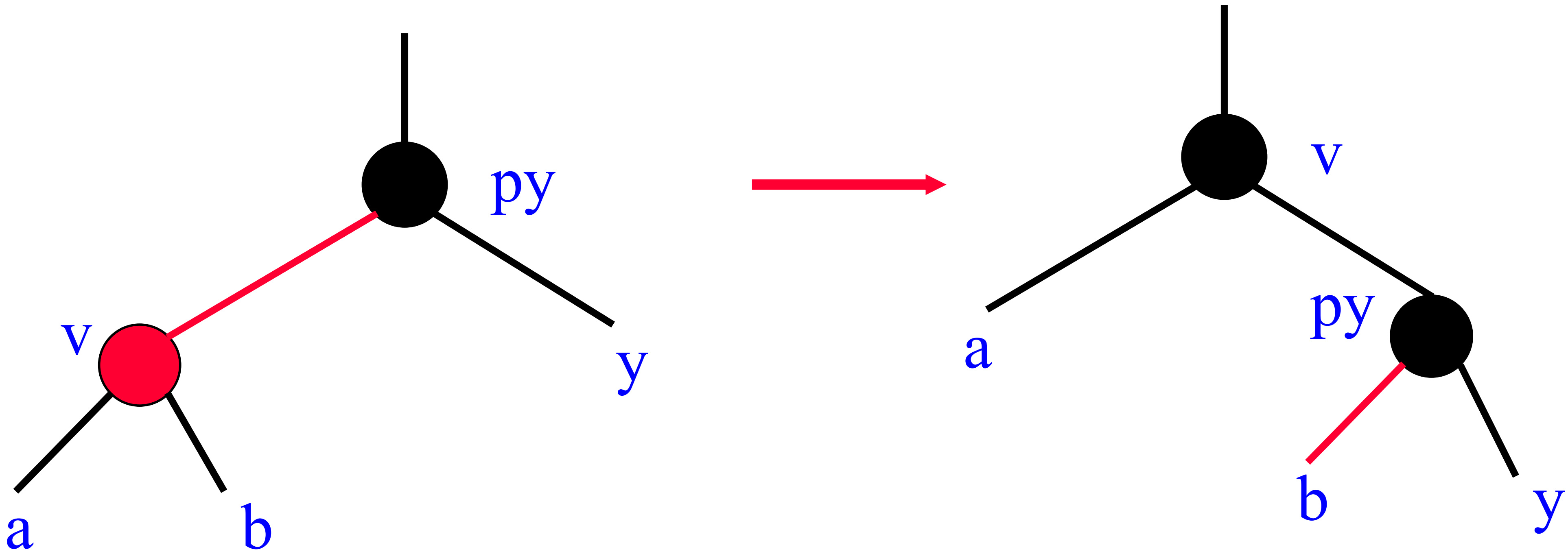
- LR rotation.
- Deficiency eliminated.
- Done!

Rr(n)

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

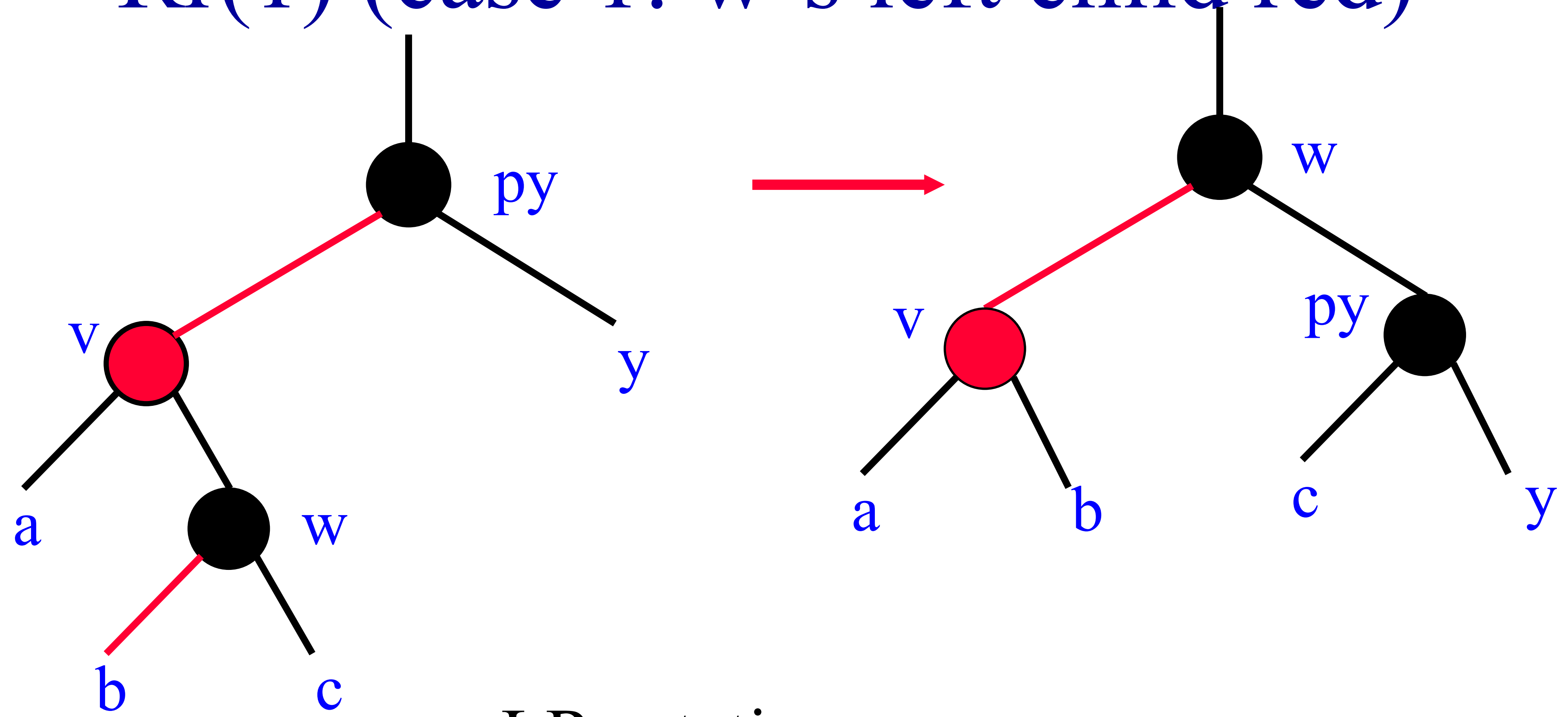


Rr(0)



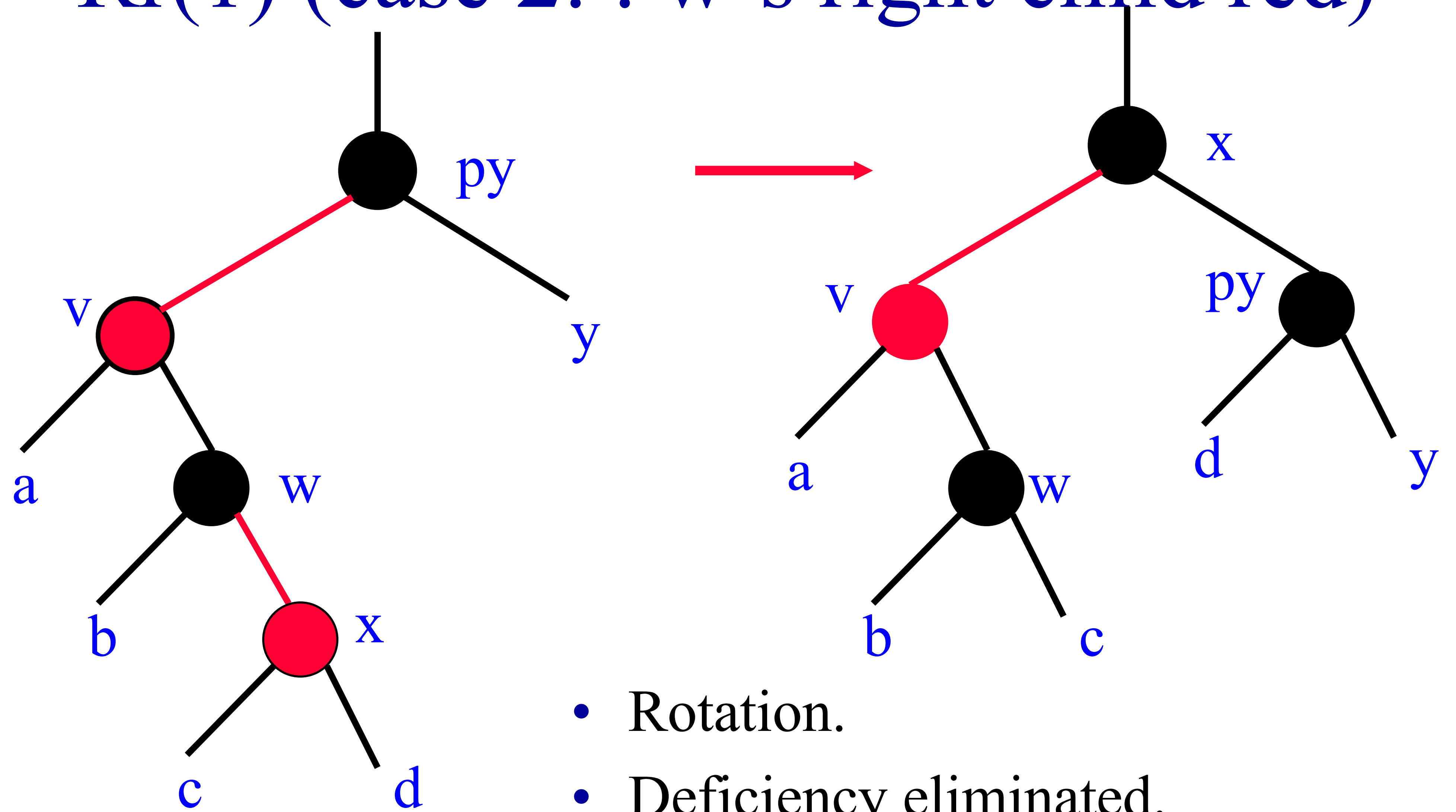
- LL rotation.
- Done!

Rr(1) (case 1: w's left child red)



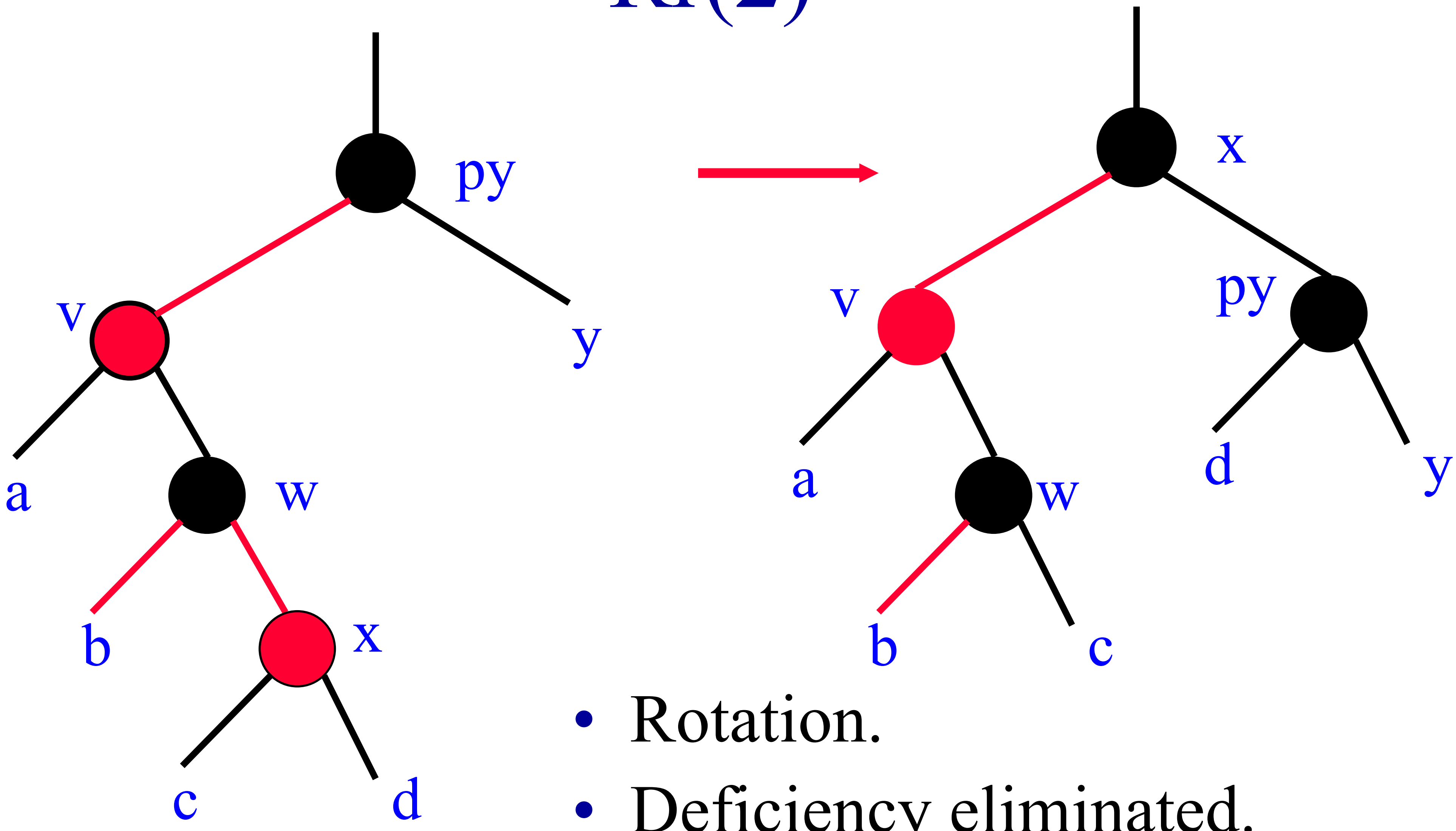
- LR rotation.
- Deficiency eliminated.
- Done!

Rr(1) (case 2: : w's right child red)



- Rotation.
- Deficiency eliminated.
- Done!

Rr(2)

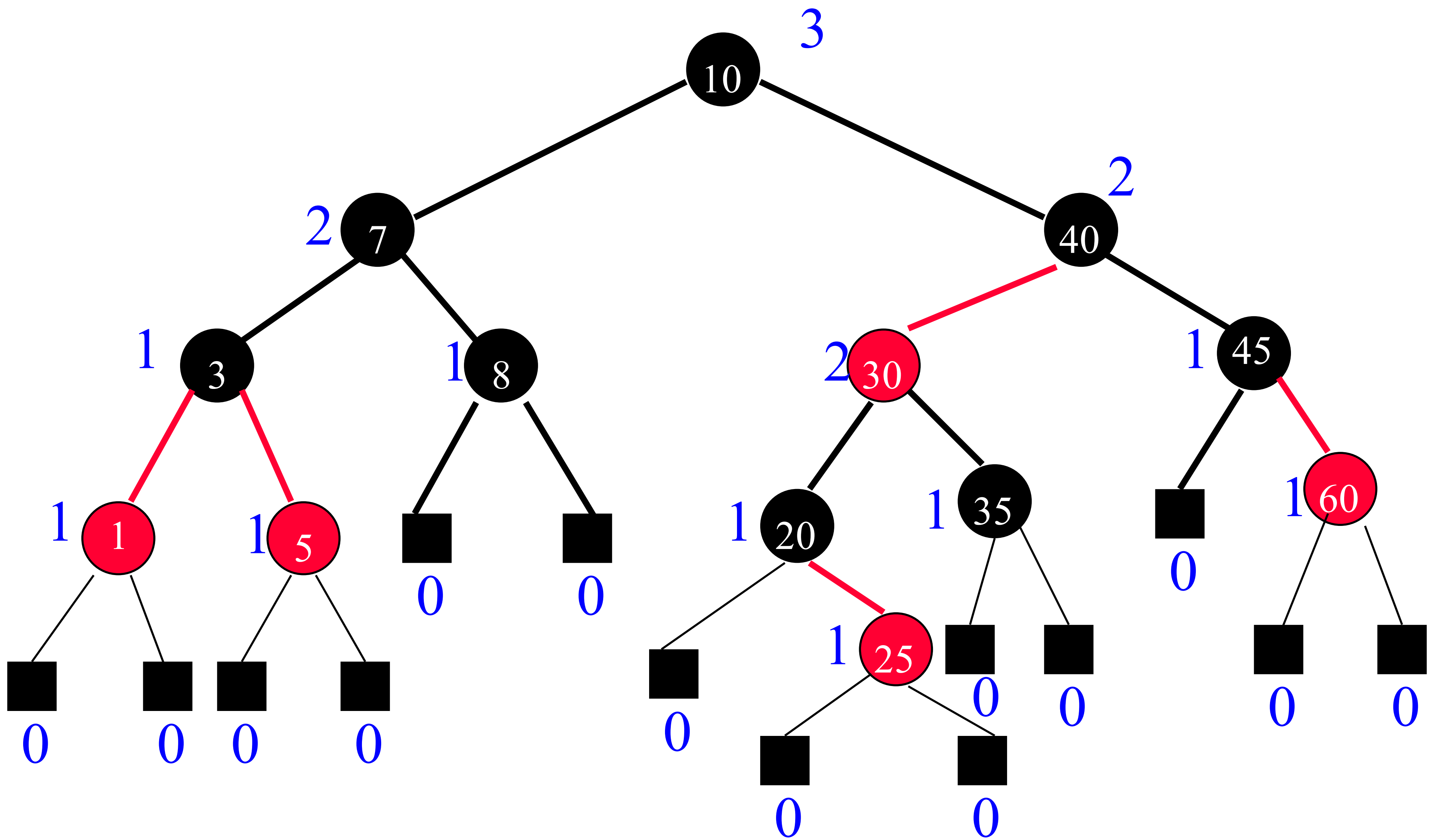


- Rotation.
- Deficiency eliminated.
- Done!

Red-Black Trees—Again

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

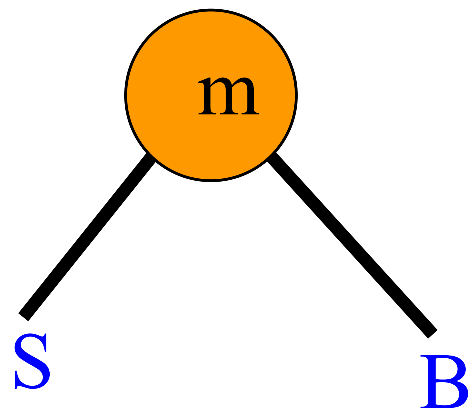
An Example



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.

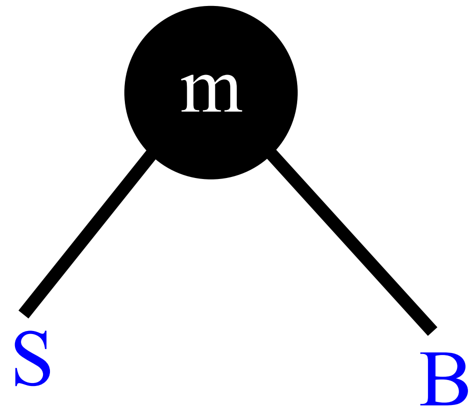
Join Binary Search Trees



- O(1) time.

Join Red-black Trees

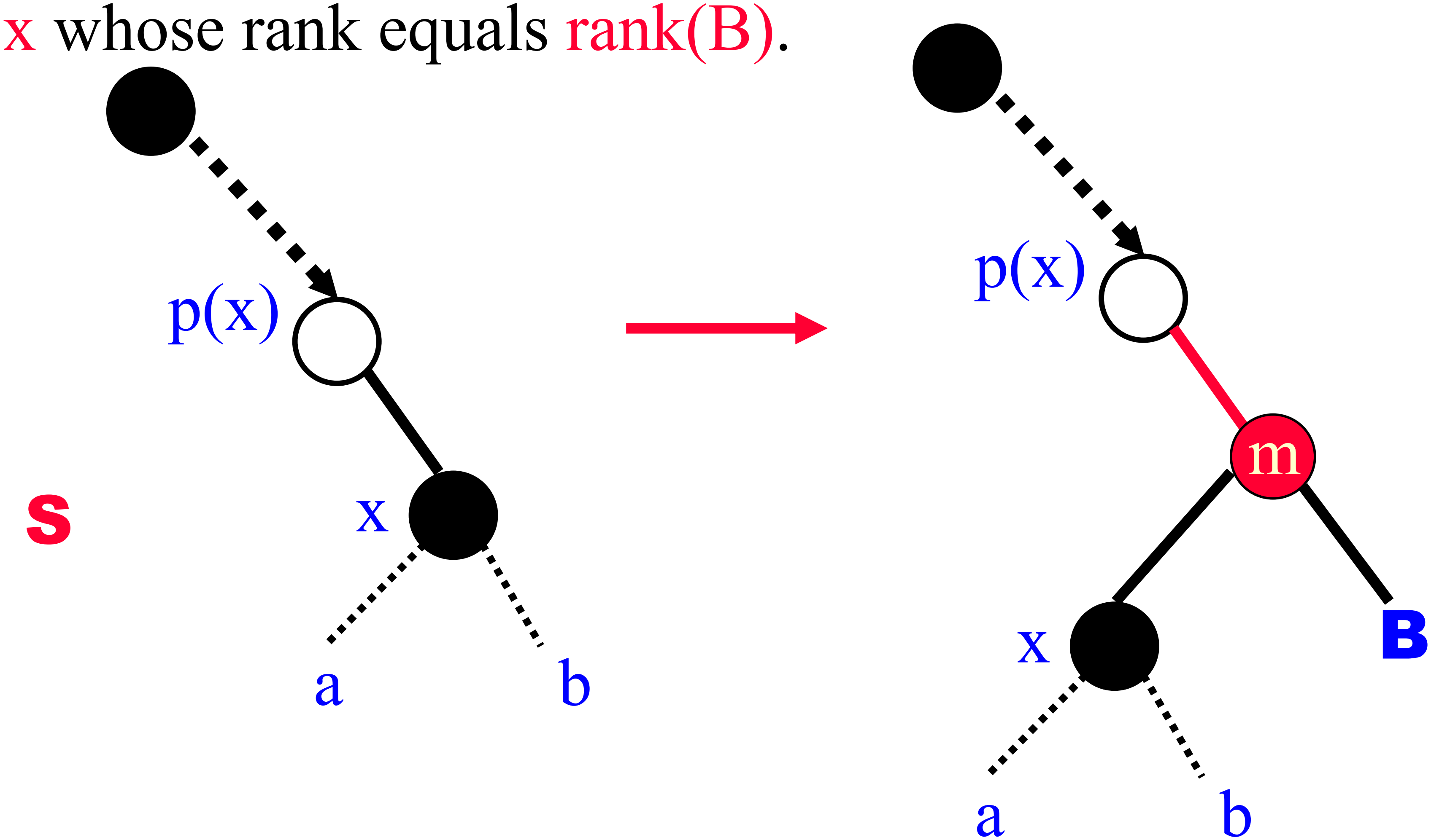
- When rank(S) = rank(B), use binary search tree method.



- rank(root) = rank(S) + 1 = rank(B) + 1.

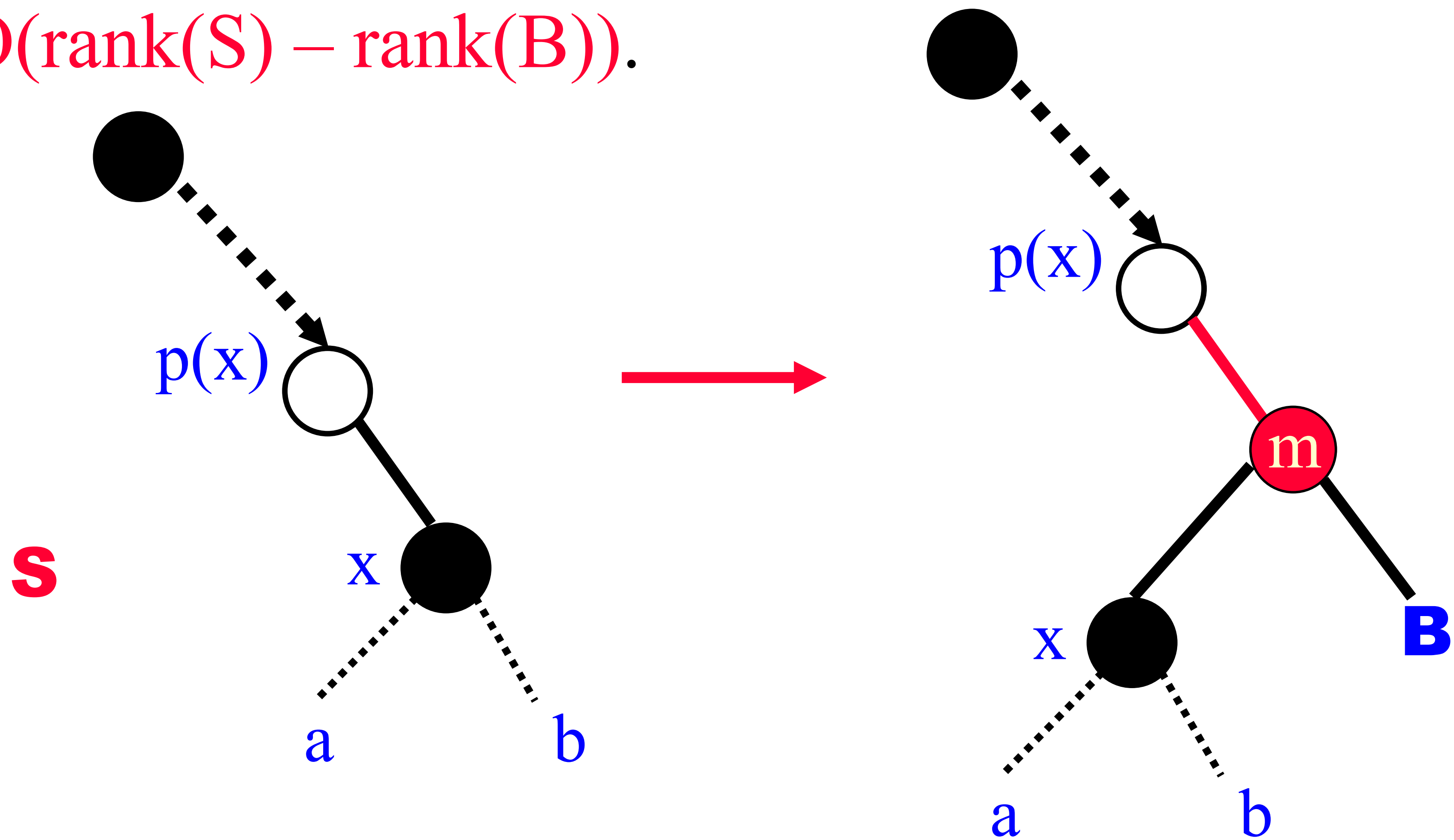
rank(S) > rank(B)

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



rank(S) > rank(B)

- If there are now 2 consecutive red pointers/nodes, perform bottom-up rebalancing beginning at m.
- O(rank(S) – rank(B)).



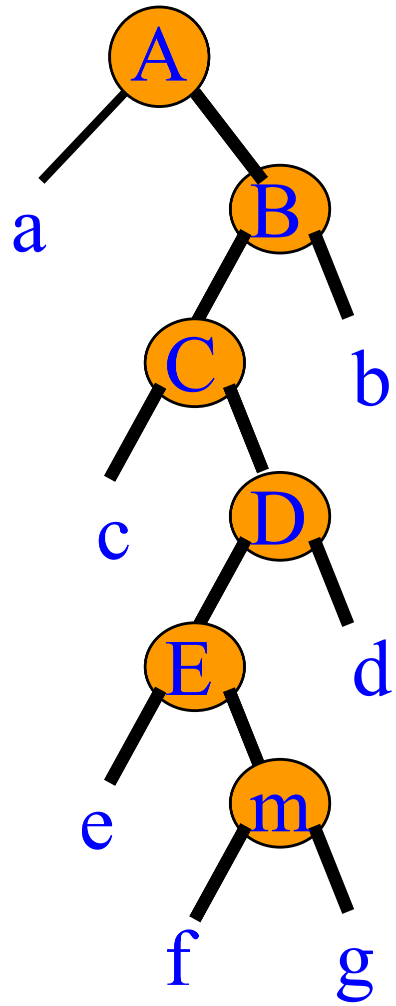
rank(S) < rank(B)

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

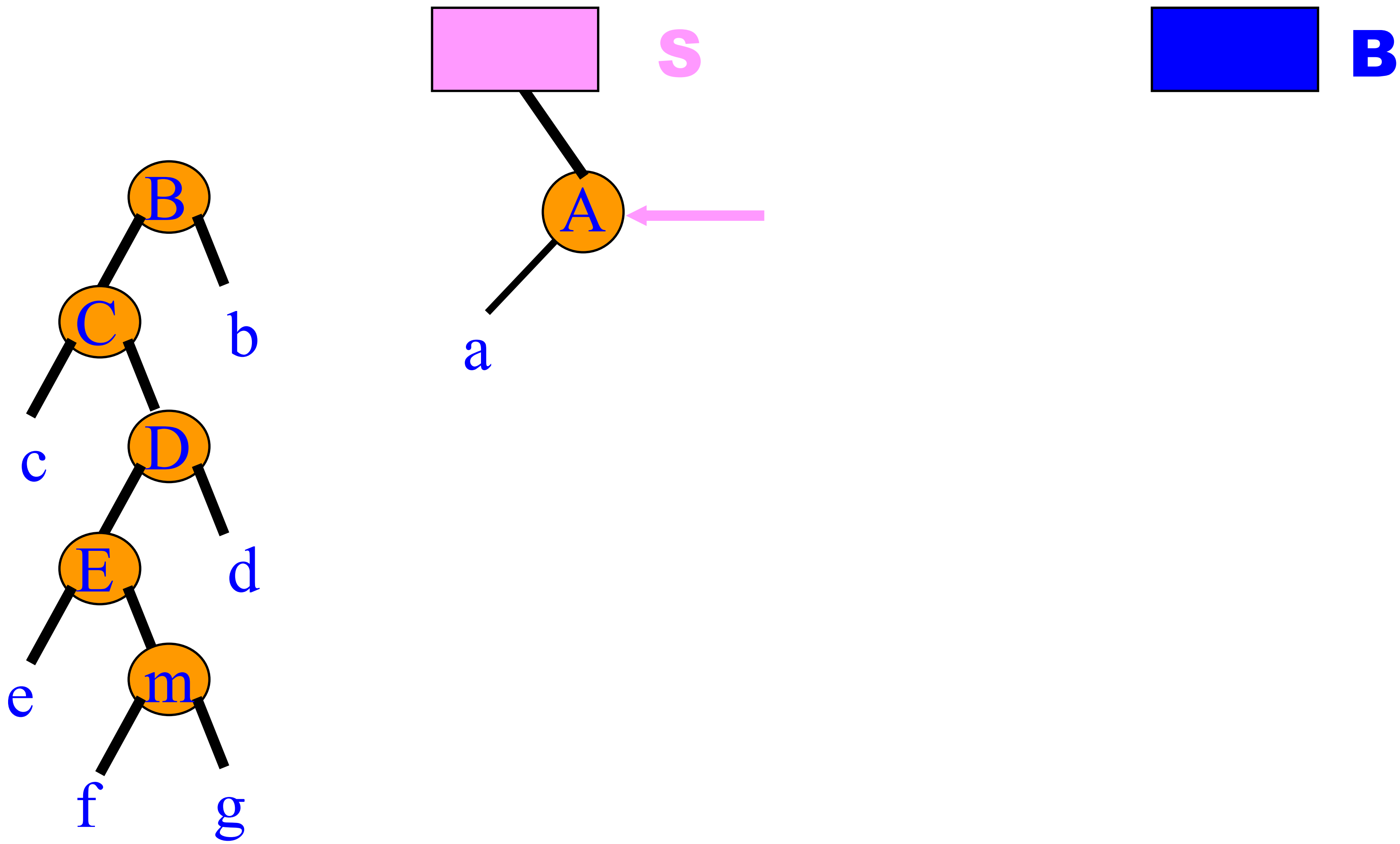
Split(k)

- 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).

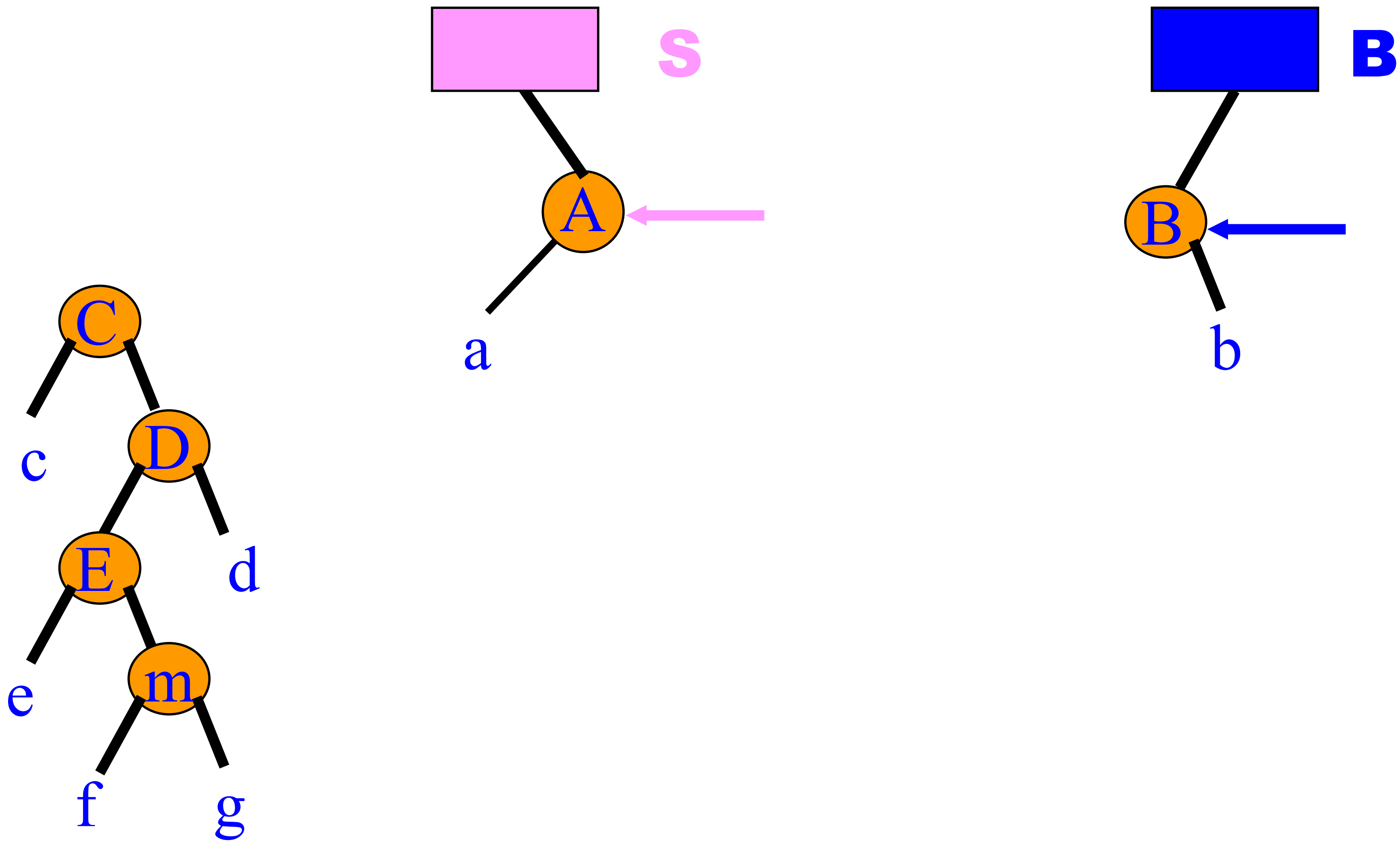
Split A Binary Search Tree



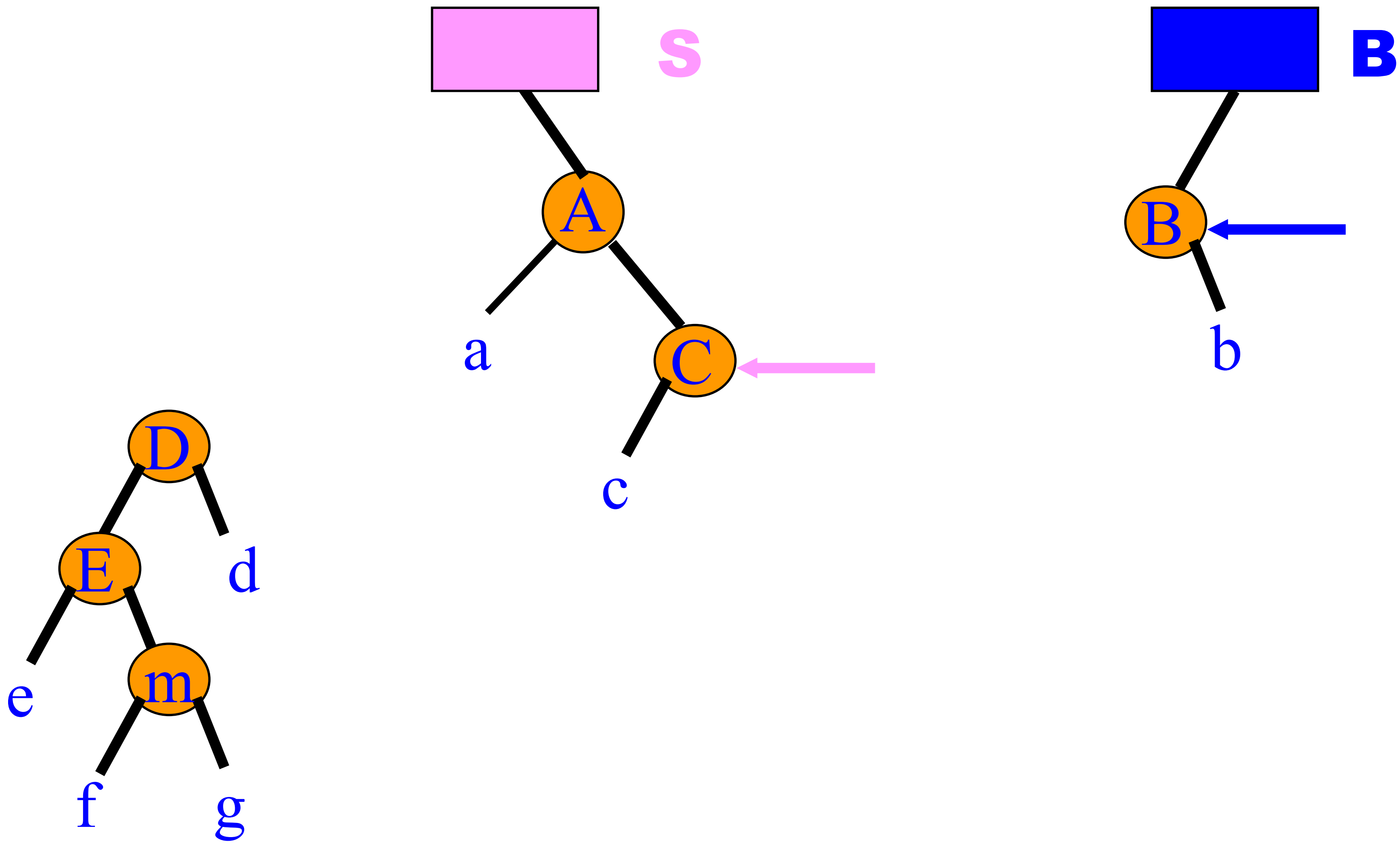
Split A Binary Search Tree



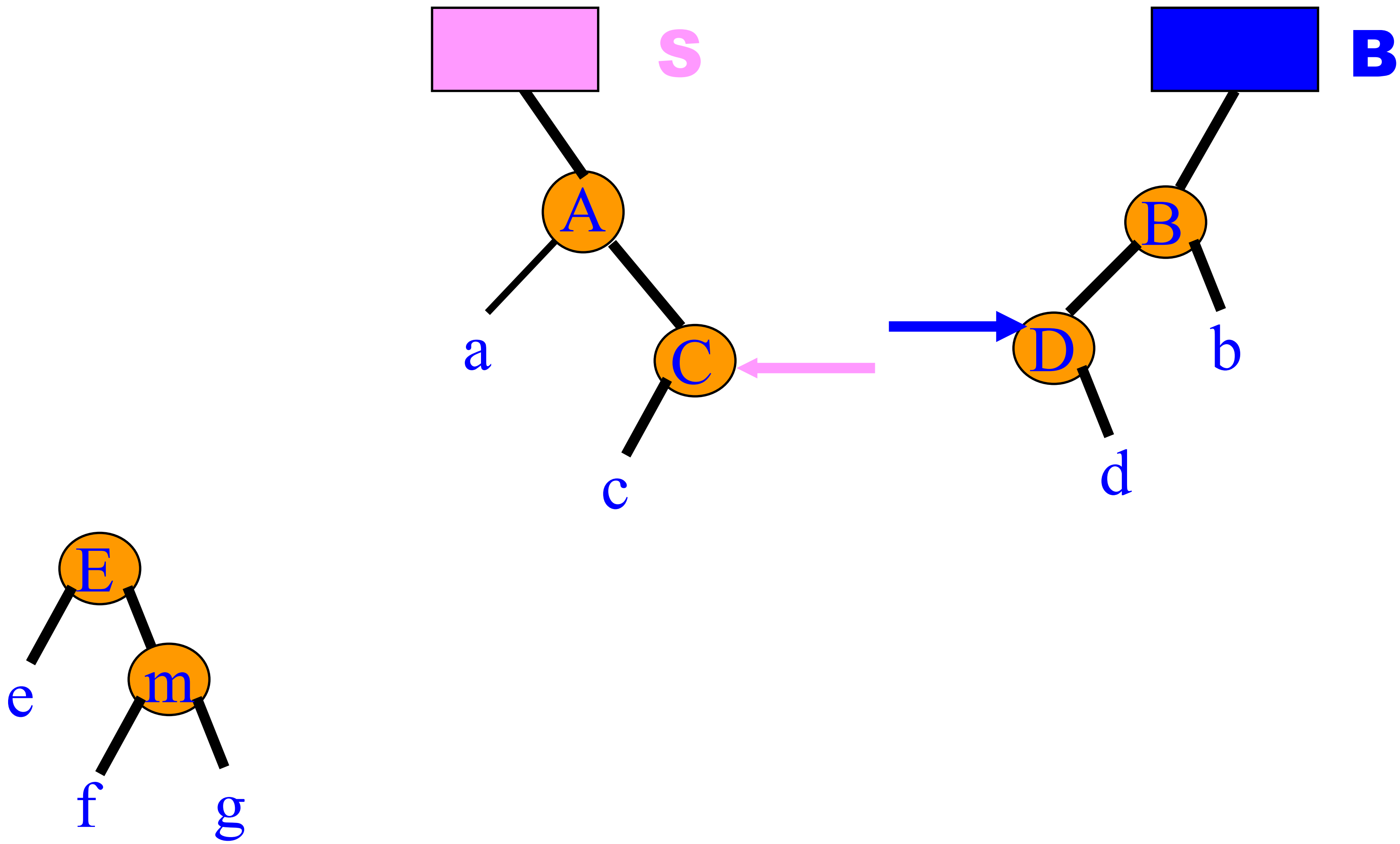
Split A Binary Search Tree



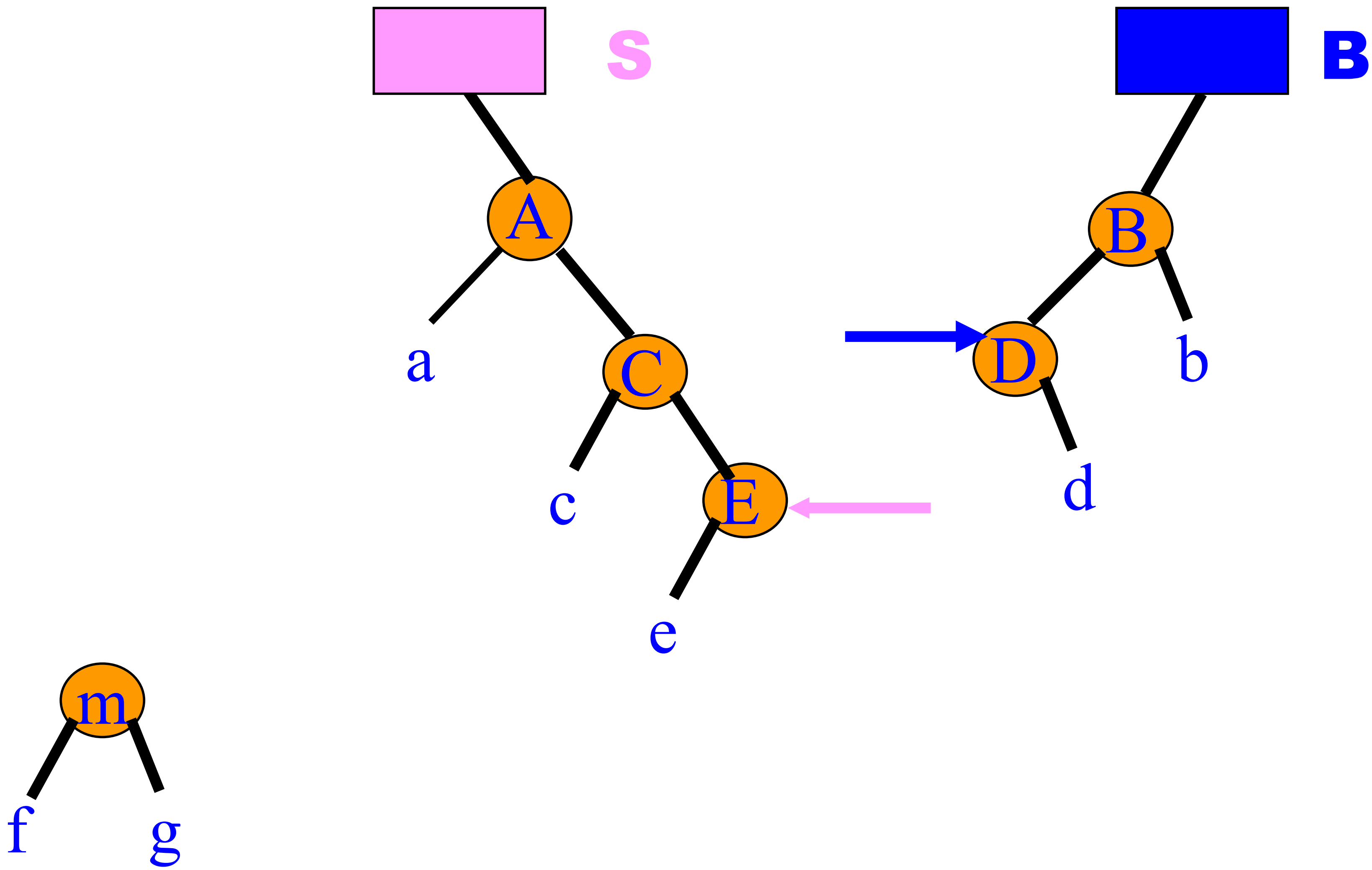
Split A Binary Search Tree



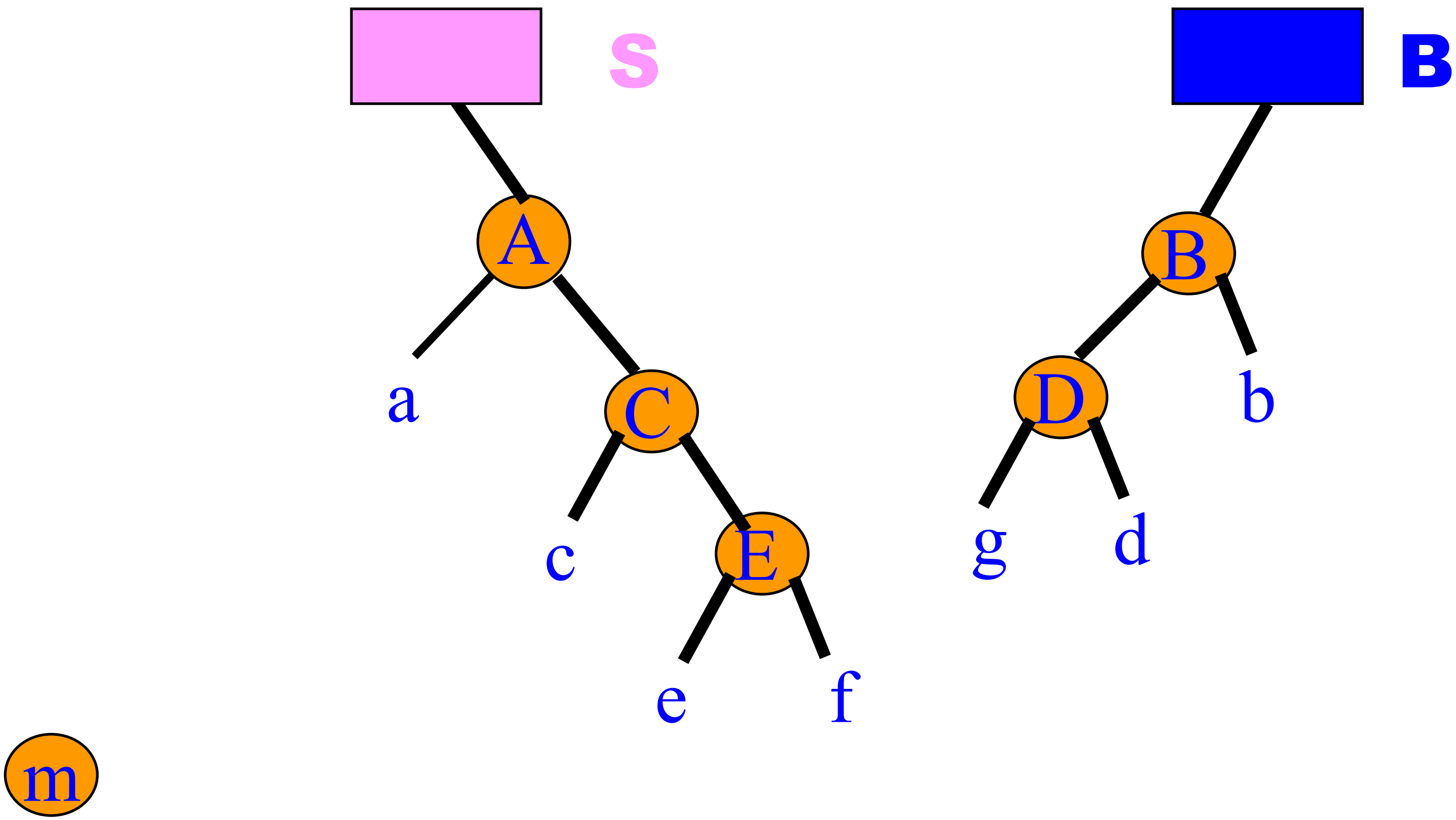
Split A Binary Search Tree



Split A Binary Search Tree



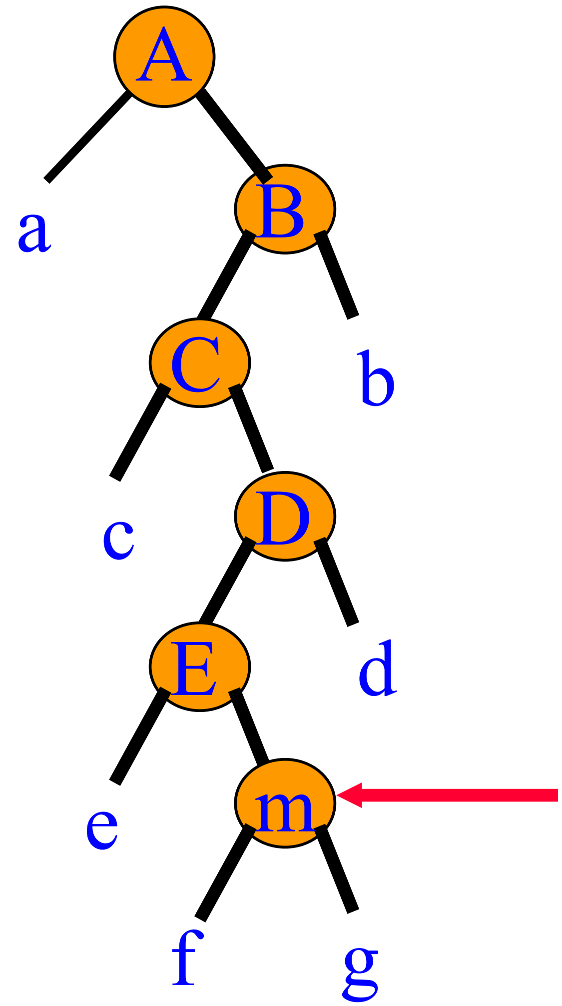
Split A Binary Search Tree



Split A Red-Black Tree

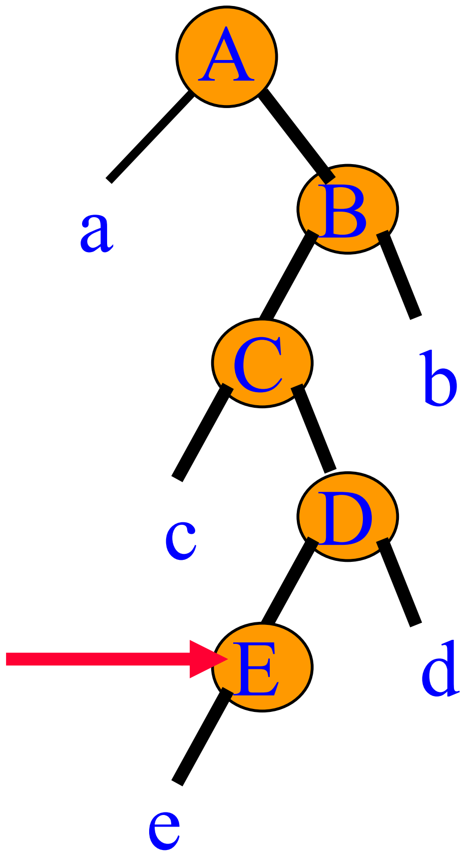
- Previous strategy does not split a red-black tree into two red-black trees.
- Must do a search for m followed by a traceback to the root.
- During the traceback use the join operation to construct S and B.

Split A Red-Black Tree



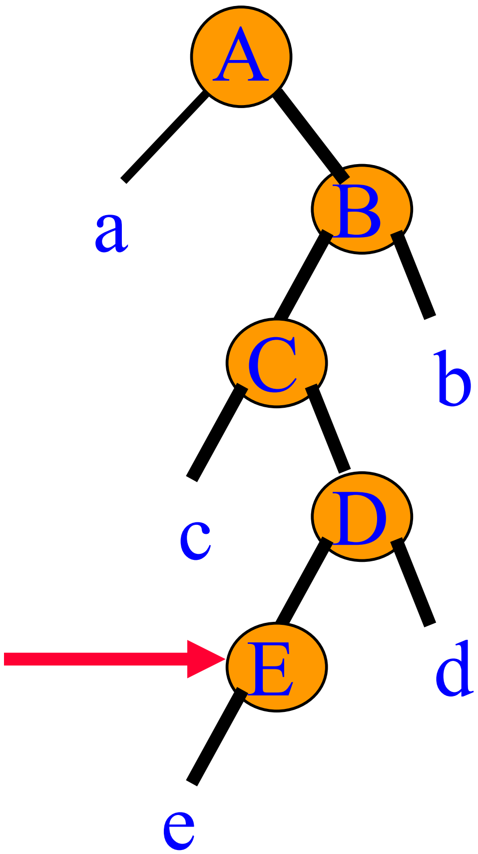
S = f **B** = g

Split A Red-Black Tree



S = f **B** = g

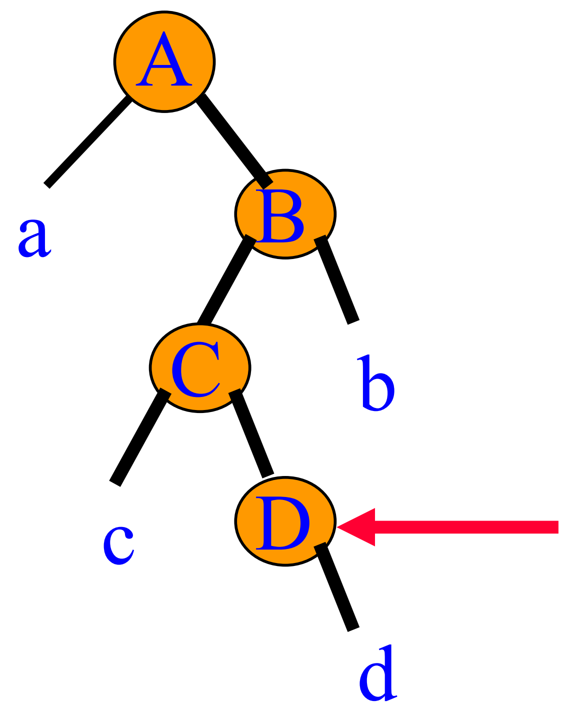
Split A Red-Black Tree



S = f   **B** = g

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

Split A Red-Black Tree

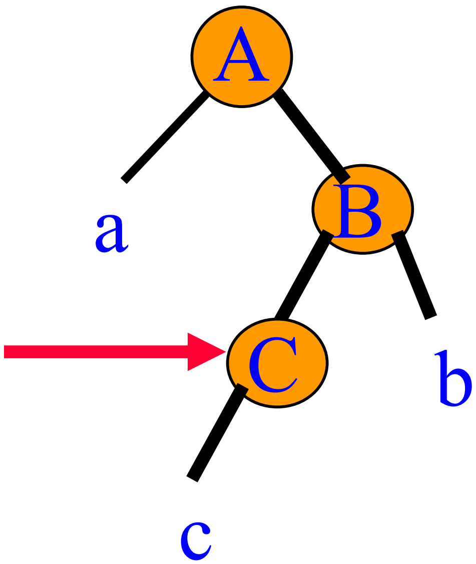


S = f   **B** = g

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

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

Split A Red-Black Tree



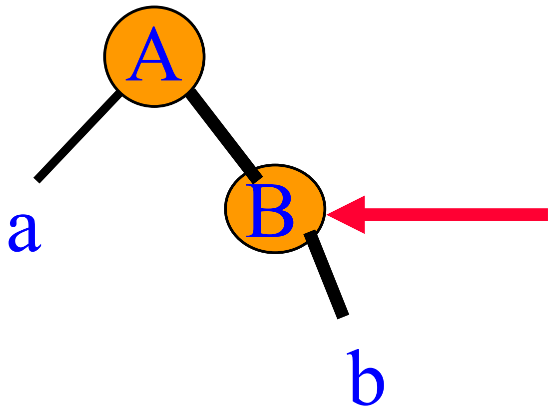
S = f **B** = g

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

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

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

Split A Red-Black Tree



S = f **B** = g

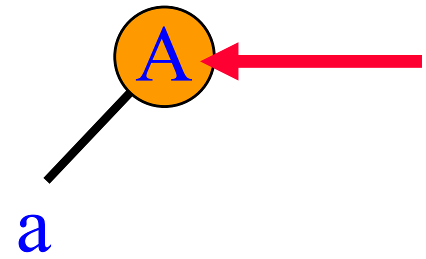
S = join(e, E, **S**)

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**)

Complexity Of Split

- O(log n)