Red Black Trees

Colored Nodes Definition

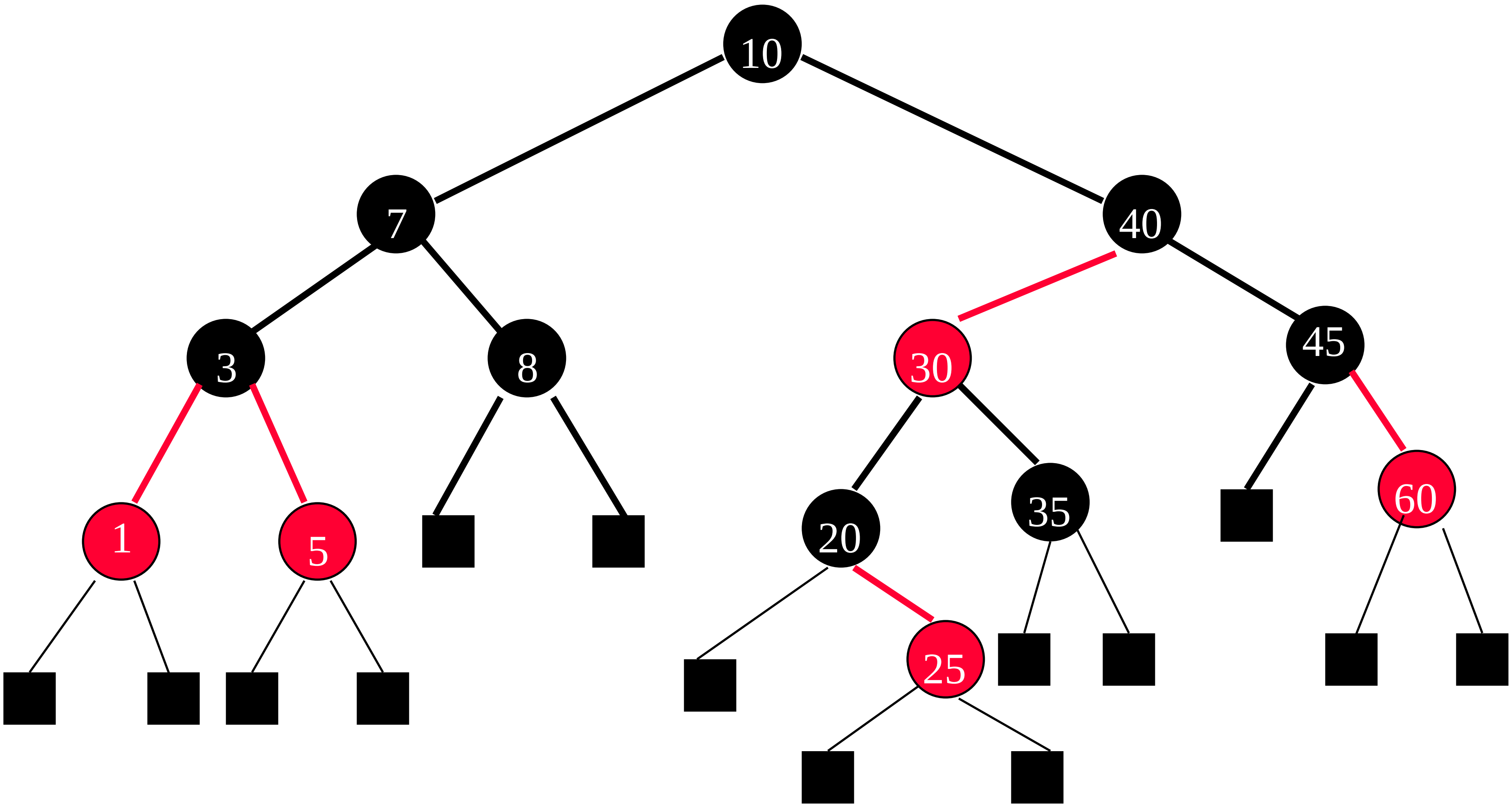
- Binary search tree.
- Each node is colored red or black.
- Root and all external nodes are black.
- No root-to-external-node path has two consecutive red nodes.
- All root-to-external-node paths have the same number of black nodes

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



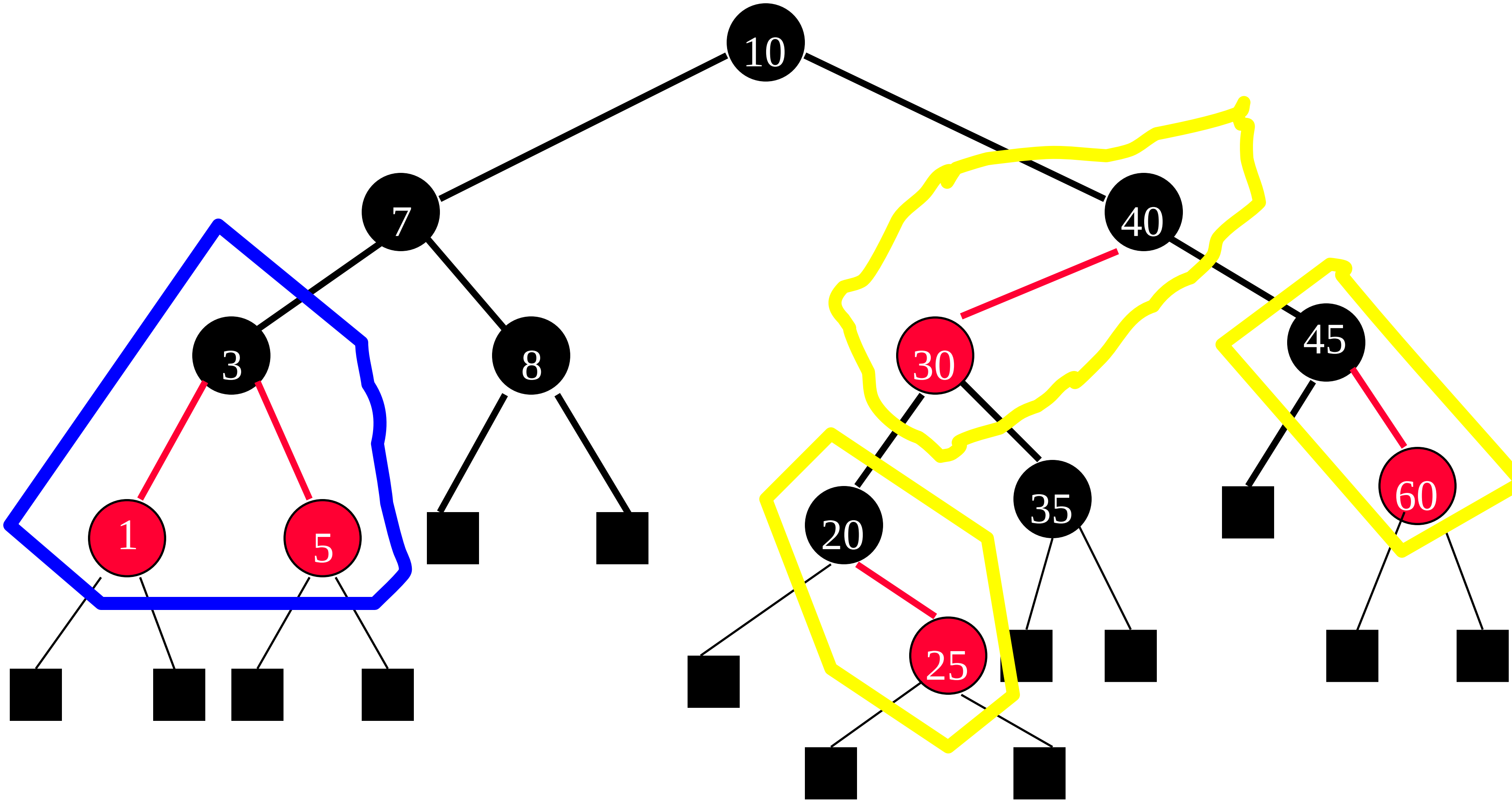
Properties

- 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 $>= h/2$, and all external nodes are at the same level.

Properties



Properties

- Let $h' >= h/2$ be the height of the collapsed tree.
- In worst-case, all internal nodes of collapsed tree have degree 2.
- 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 1 rotation and O(log n) color flips per insert/delete.
- Priority search trees.
  - Two keys per element.
  - Search tree on one key, priority queue on other.
  - Color flip doesn't disturb priority queue property.
  - Rotation disturbs priority queue property.
  - O(log n) fix time per rotation => $O(\log^2 n)$ overall time.

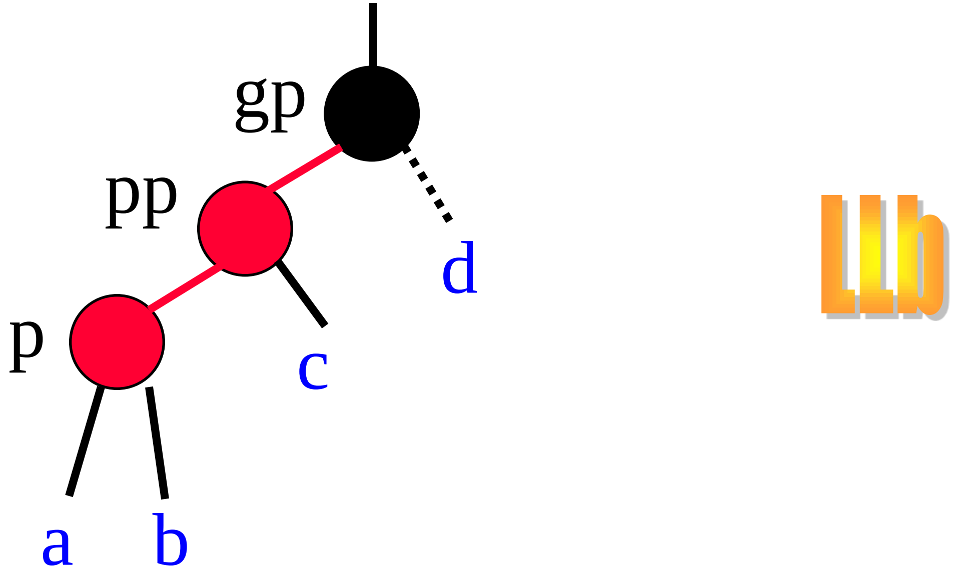
Properties

- O(1) amortized complexity to restructure following an insert/delete.
- C++ STL implementation
- java.util.TreeMap => 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 => 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.

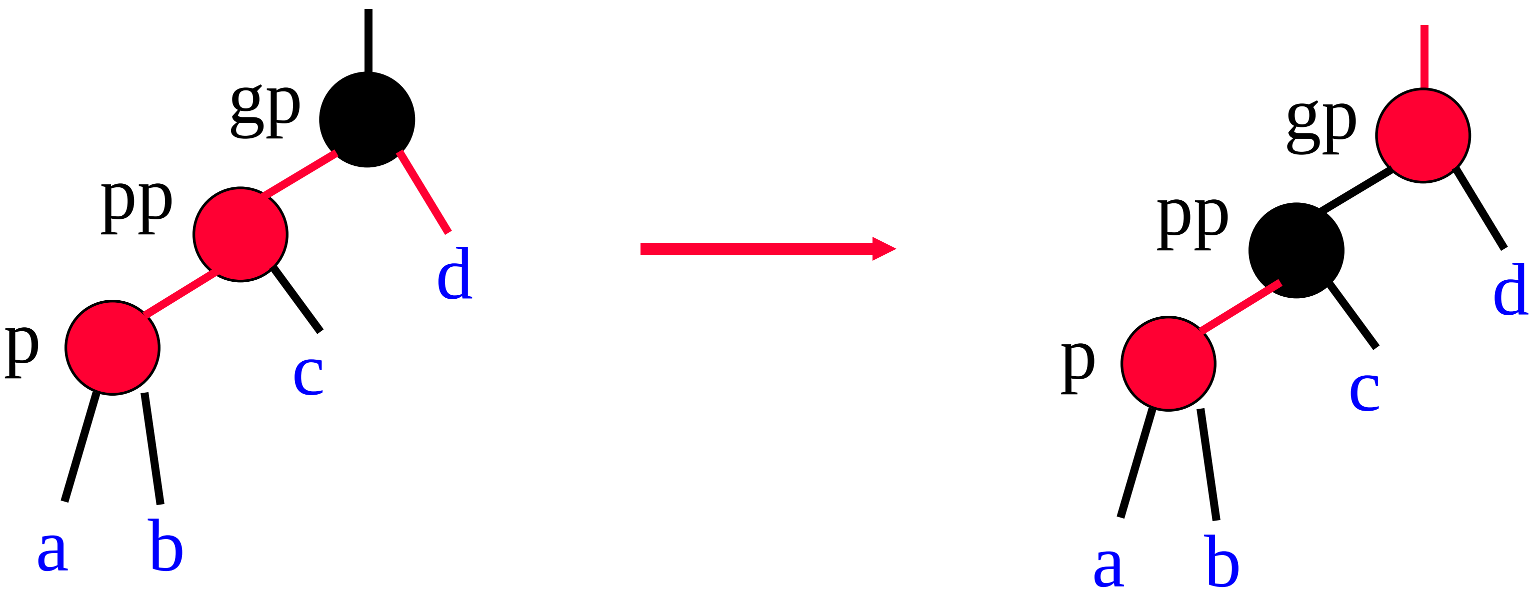
Classification Of 2 Red Nodes/Pointers



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

XYr

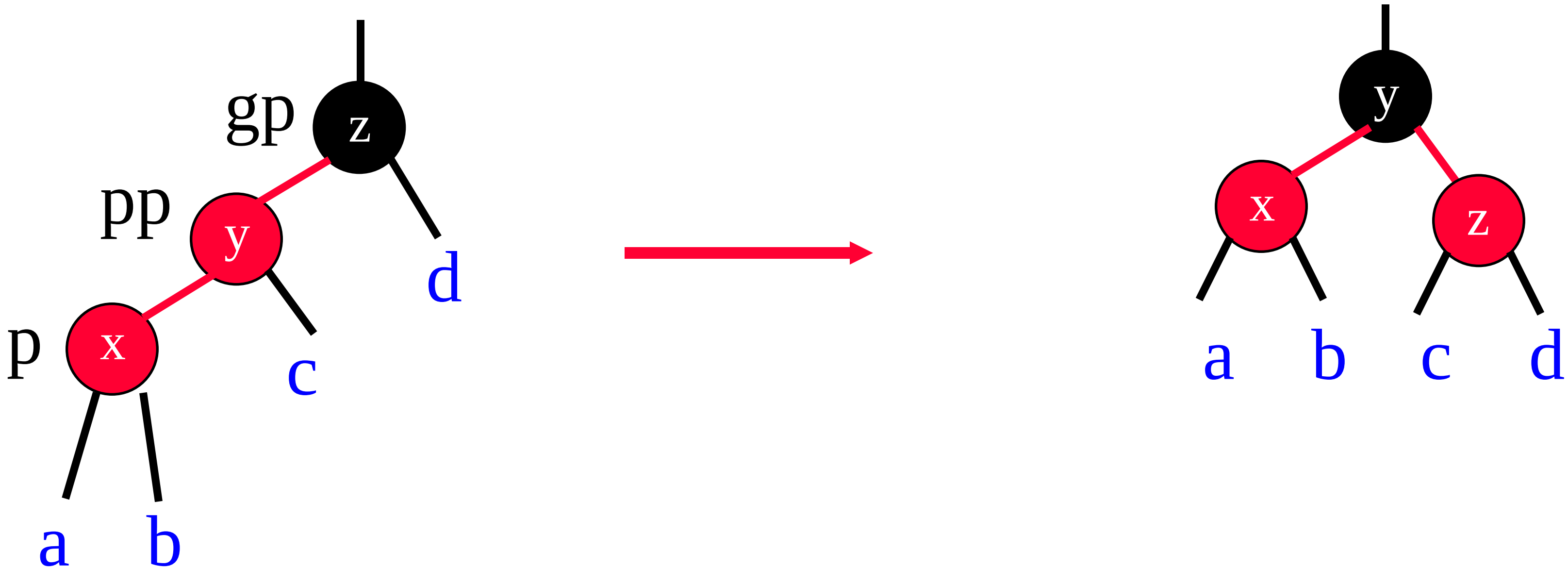
- Color flip.



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

LLb

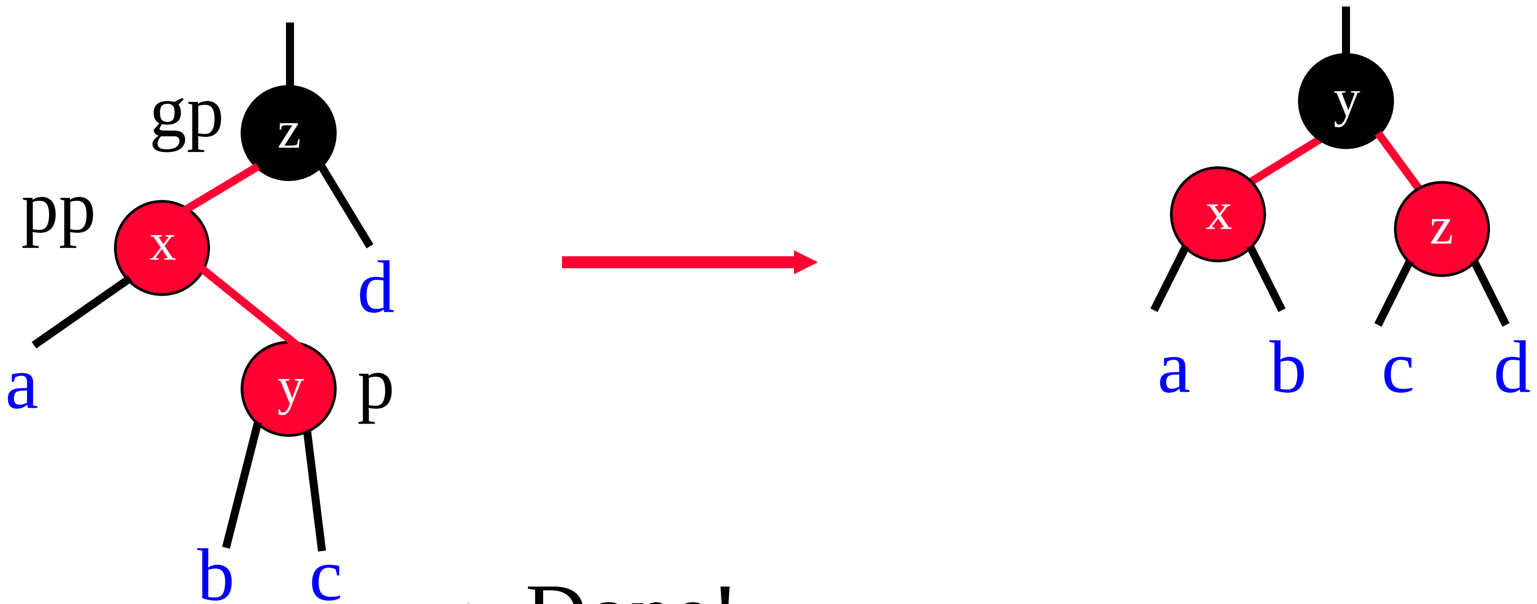
- Rotate.



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

LRb

- Rotate.

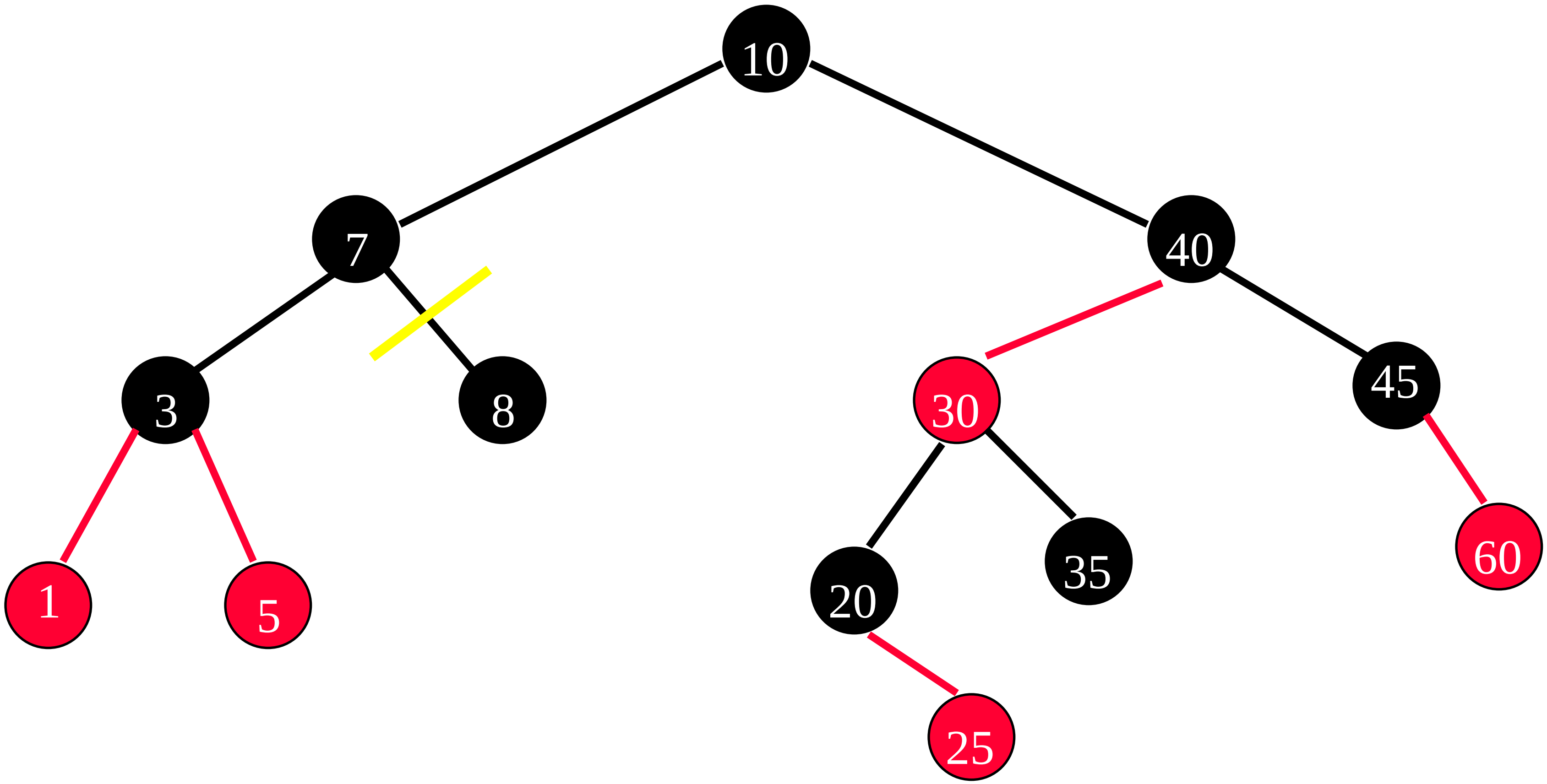


- Done!
- Same as LR rotation of AVL tree.
- RRb and RLb are symmetric.

Delete

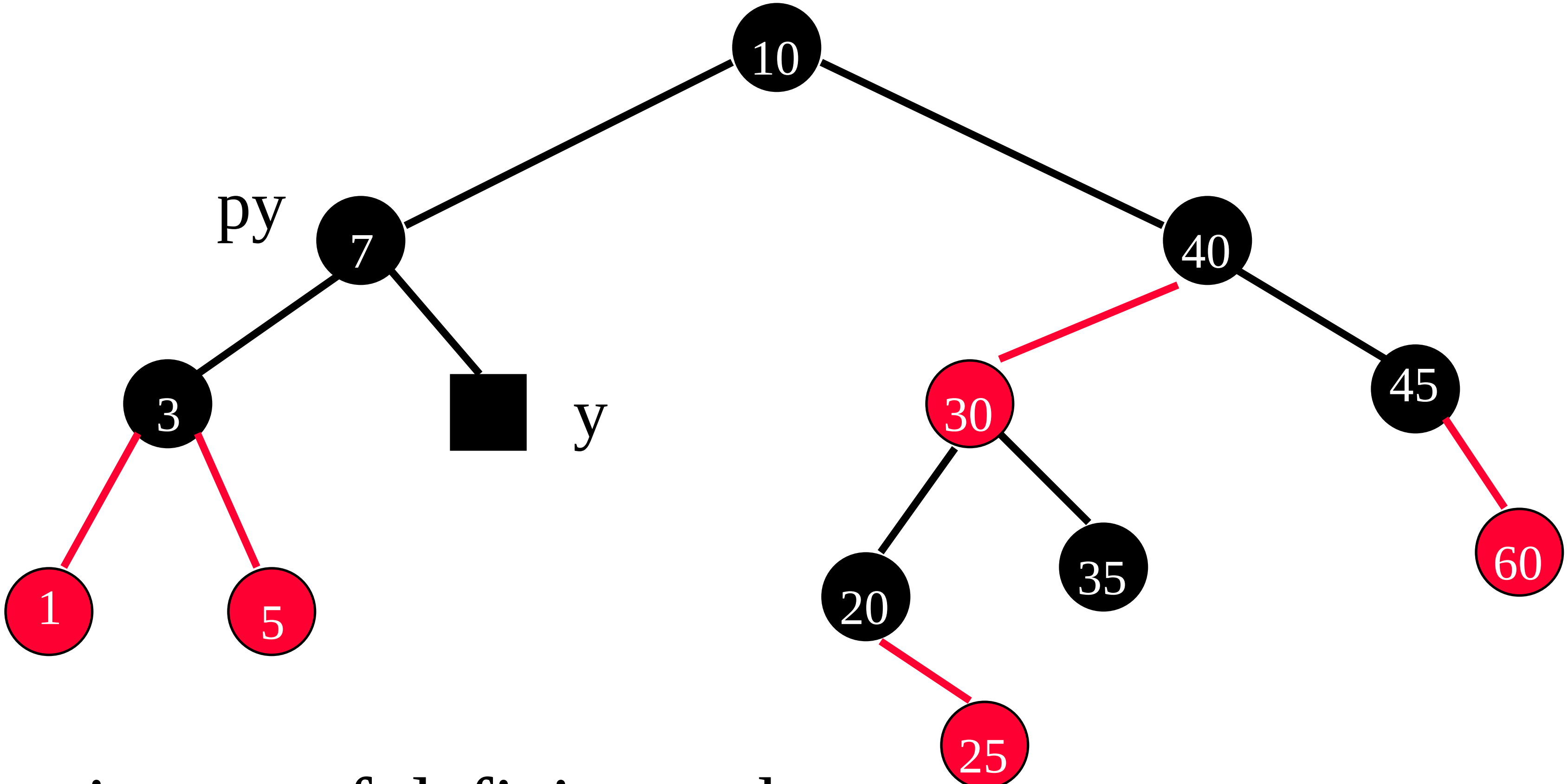
- Delete as for unbalanced binary search tree.
- If red node deleted, no rebalancing needed.
- If black node deleted, a subtree becomes one black pointer (node) deficient.

Delete A Black Leaf



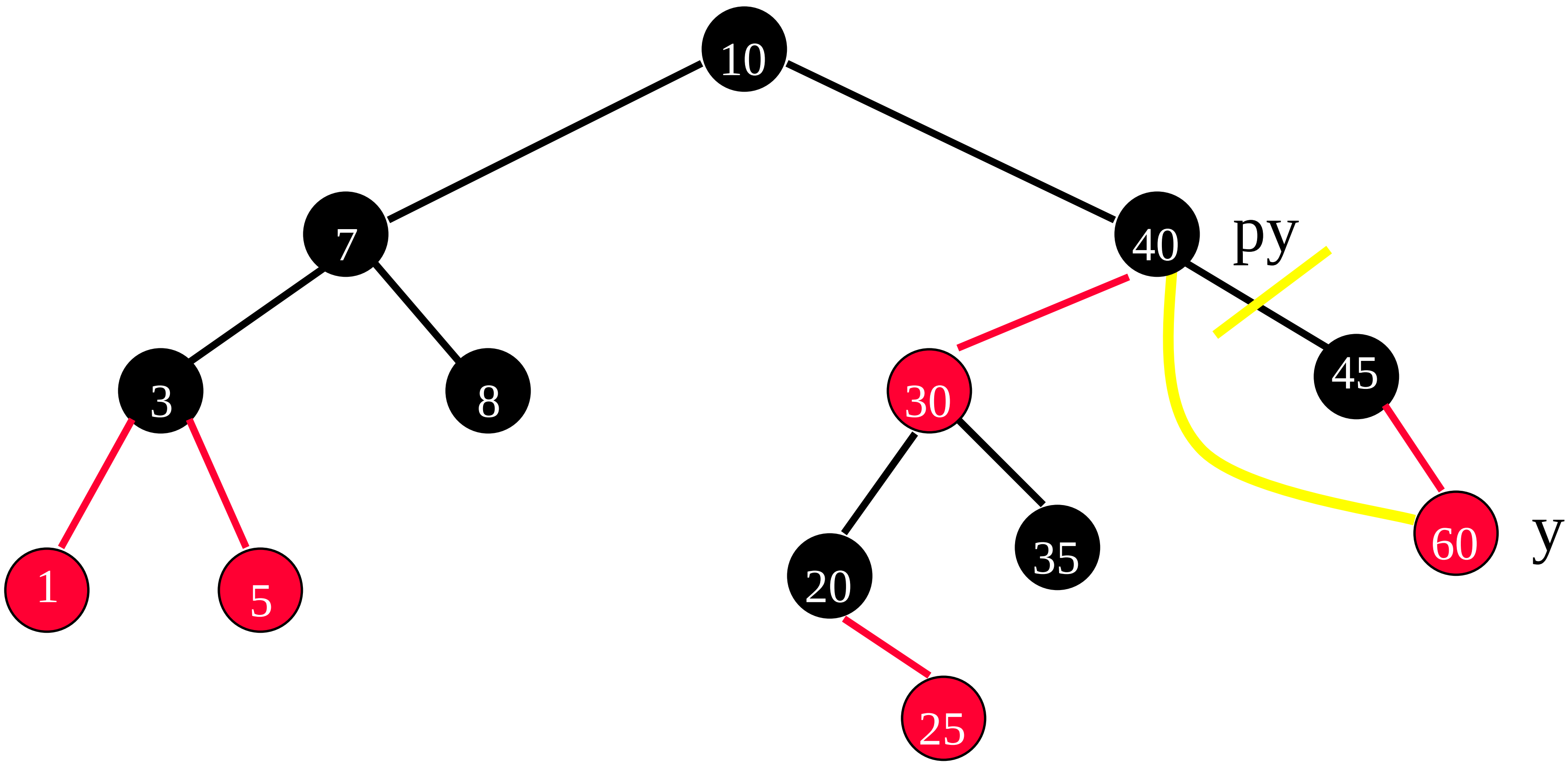
- Delete 8.

Delete A Black Leaf



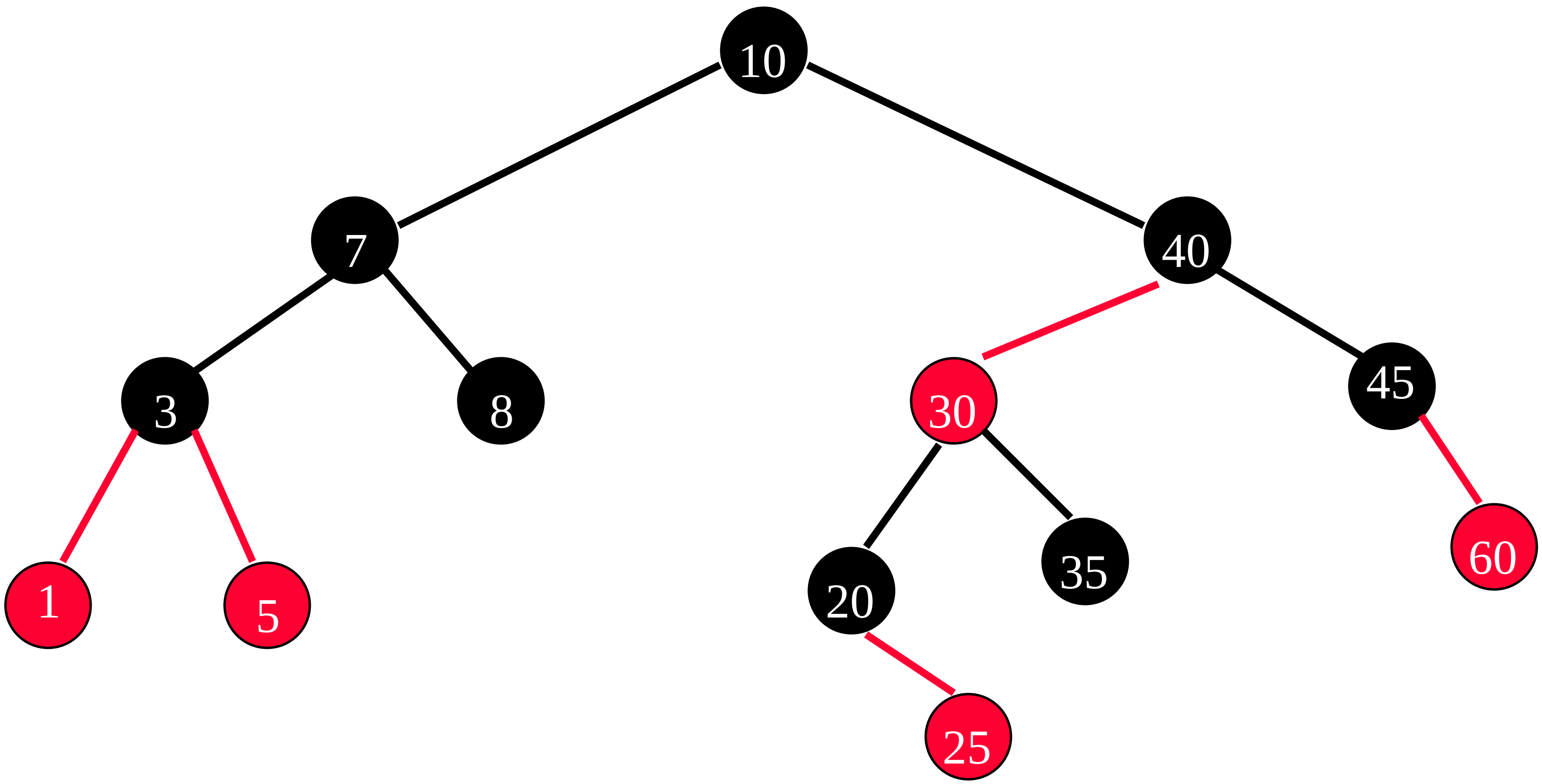
- y is root of deficient subtree.
- py is parent of y.

Delete A Black Degree 1 Node



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

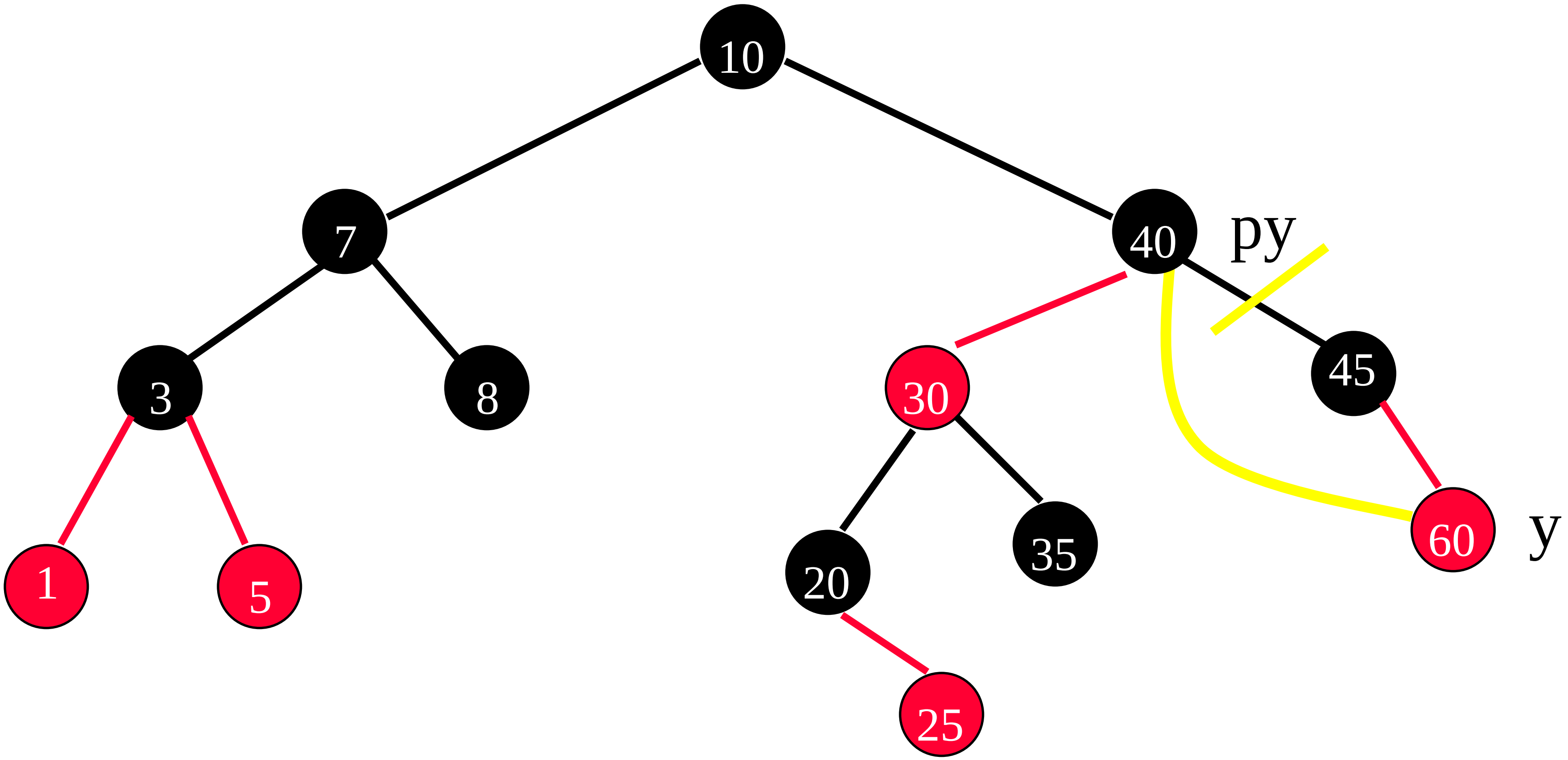
Delete A Black Degree 2 Node



- Not possible, degree 2 nodes are never deleted.

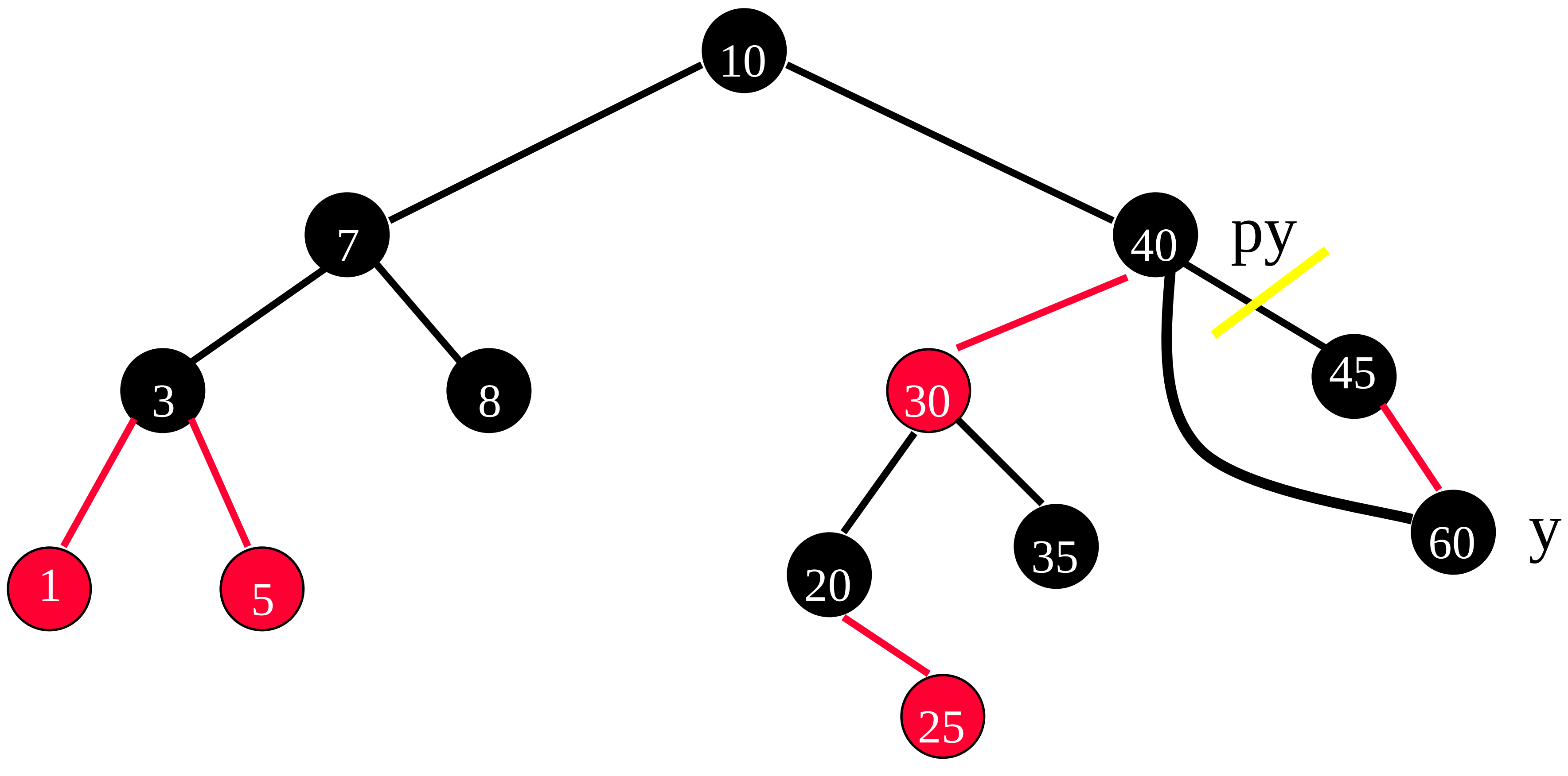
Rebalancing Strategy

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



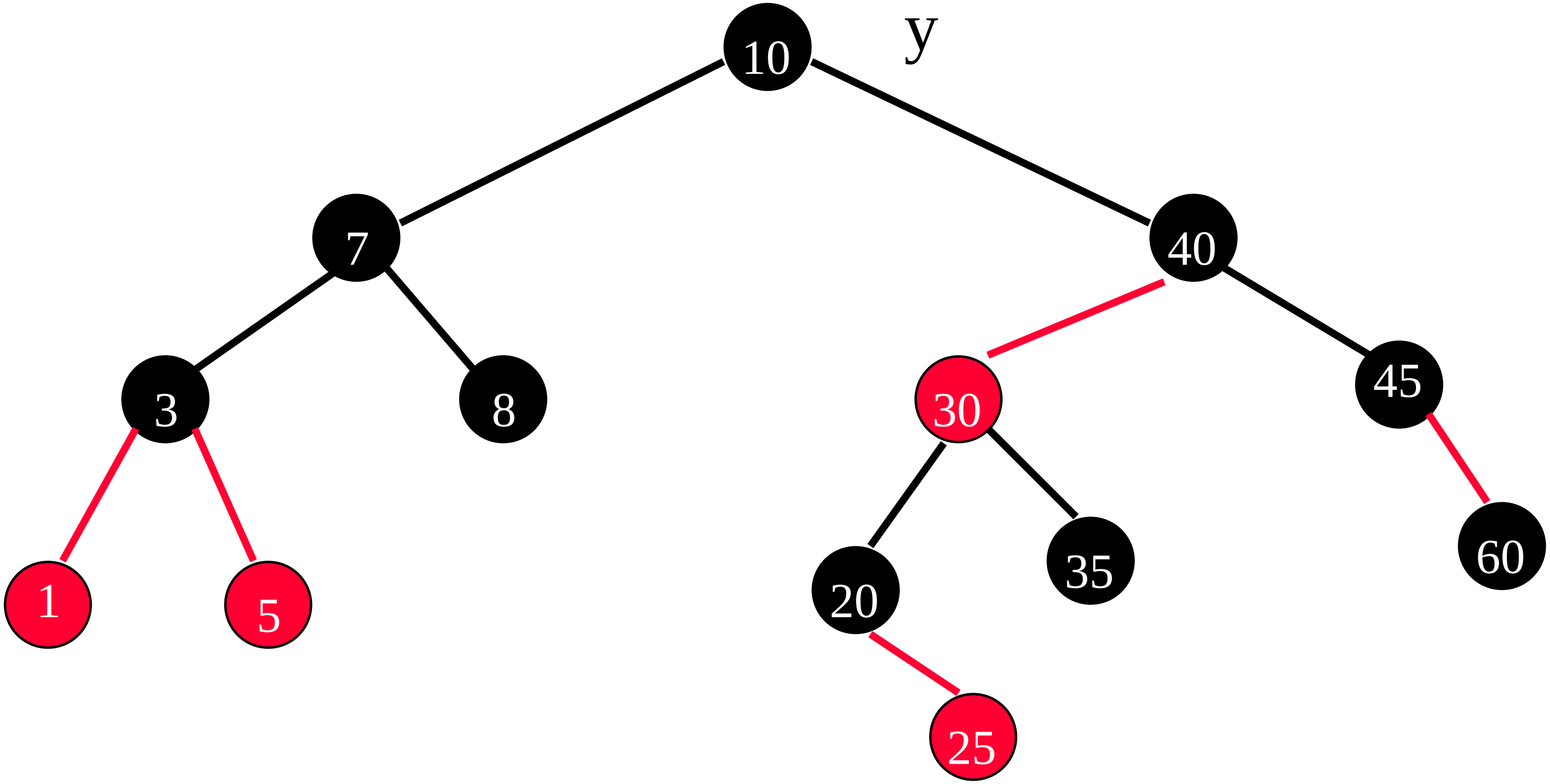
Rebalancing Strategy

- Now, no subtree is deficient. Done!



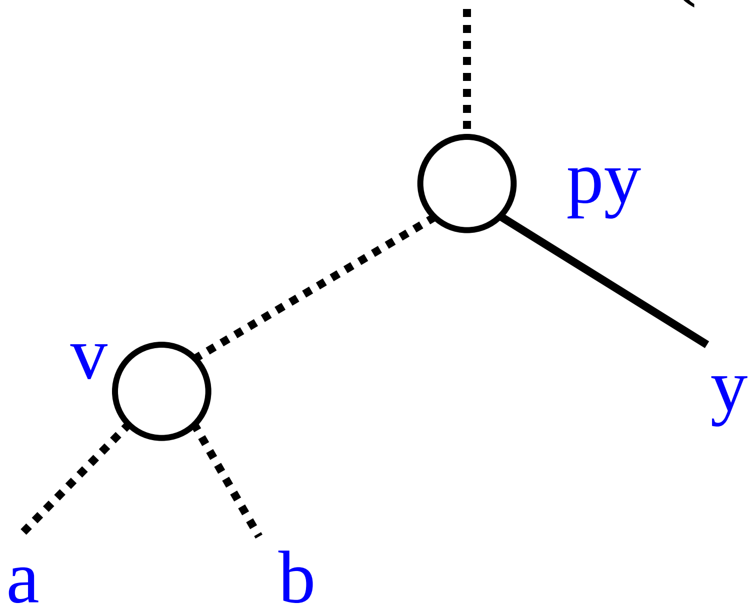
Rebalancing Strategy

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



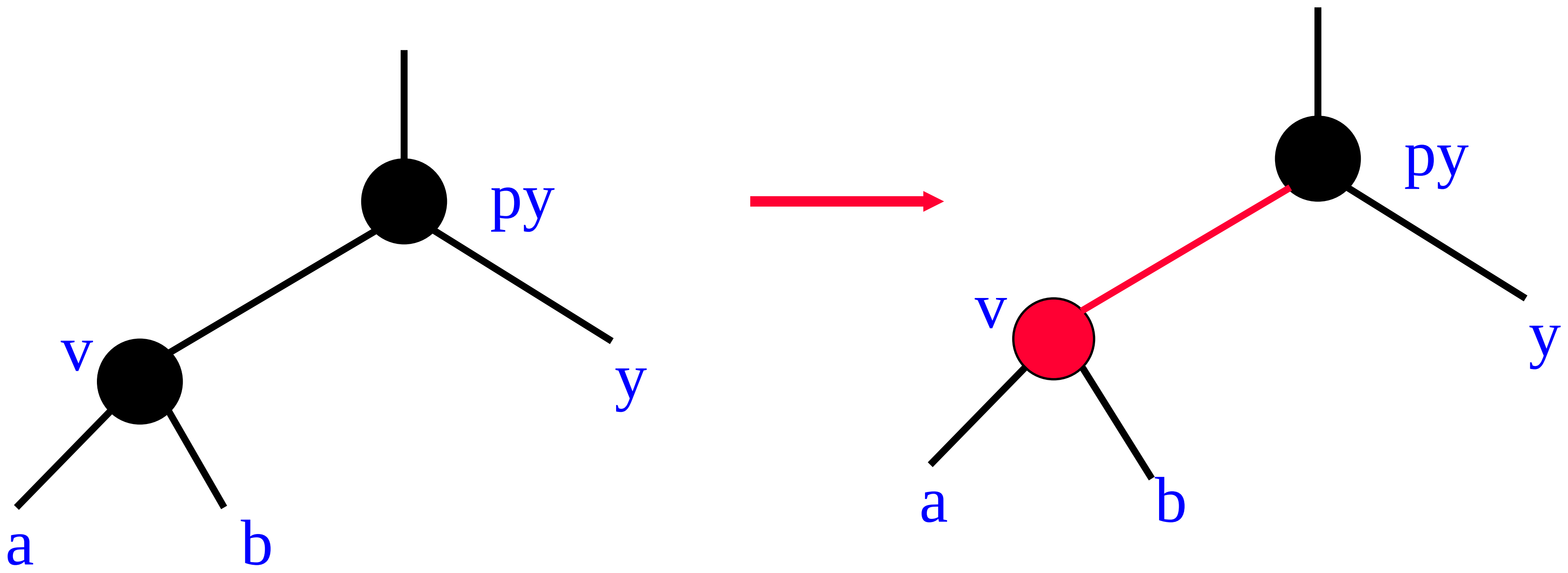
Rebalancing Strategy

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



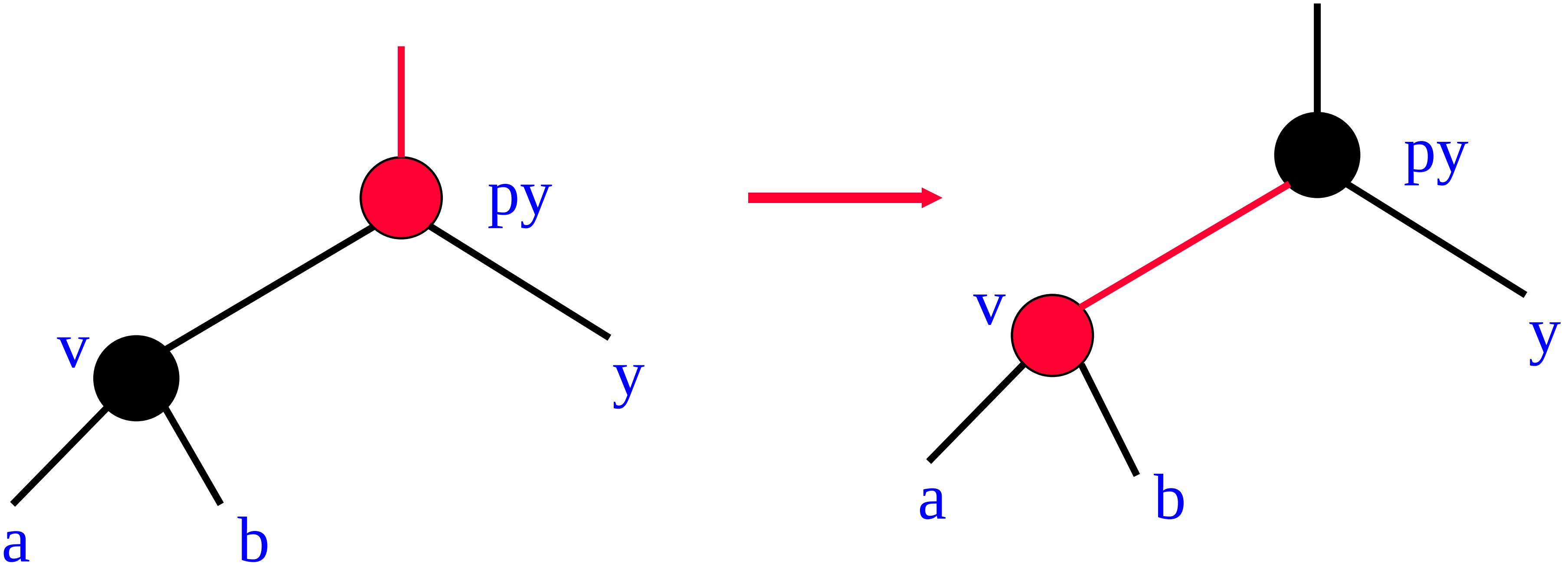
- Xcn
  - y is right child of py => X = R.
  - Pointer to v is black => c = b.
  - v has 1 red child => n = 1.

Rb0 (case 1)



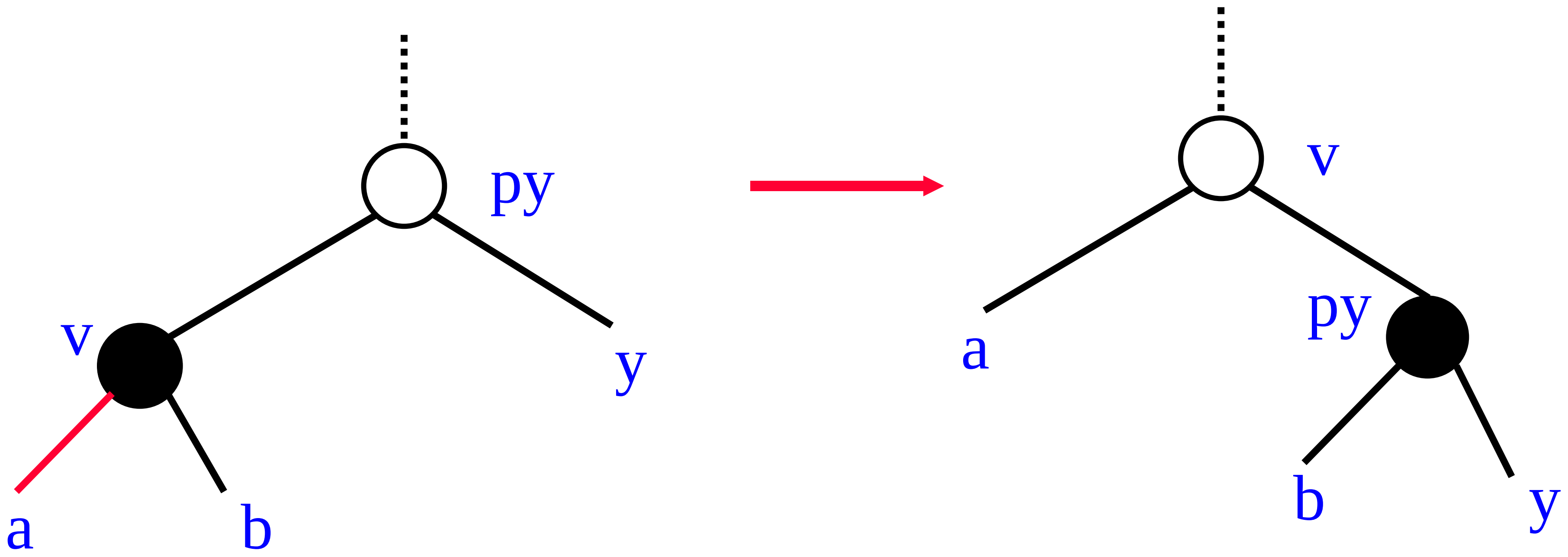
- Color change.
- Now, py is root of deficient subtree.
- Continue!

Rb0 (case 2)



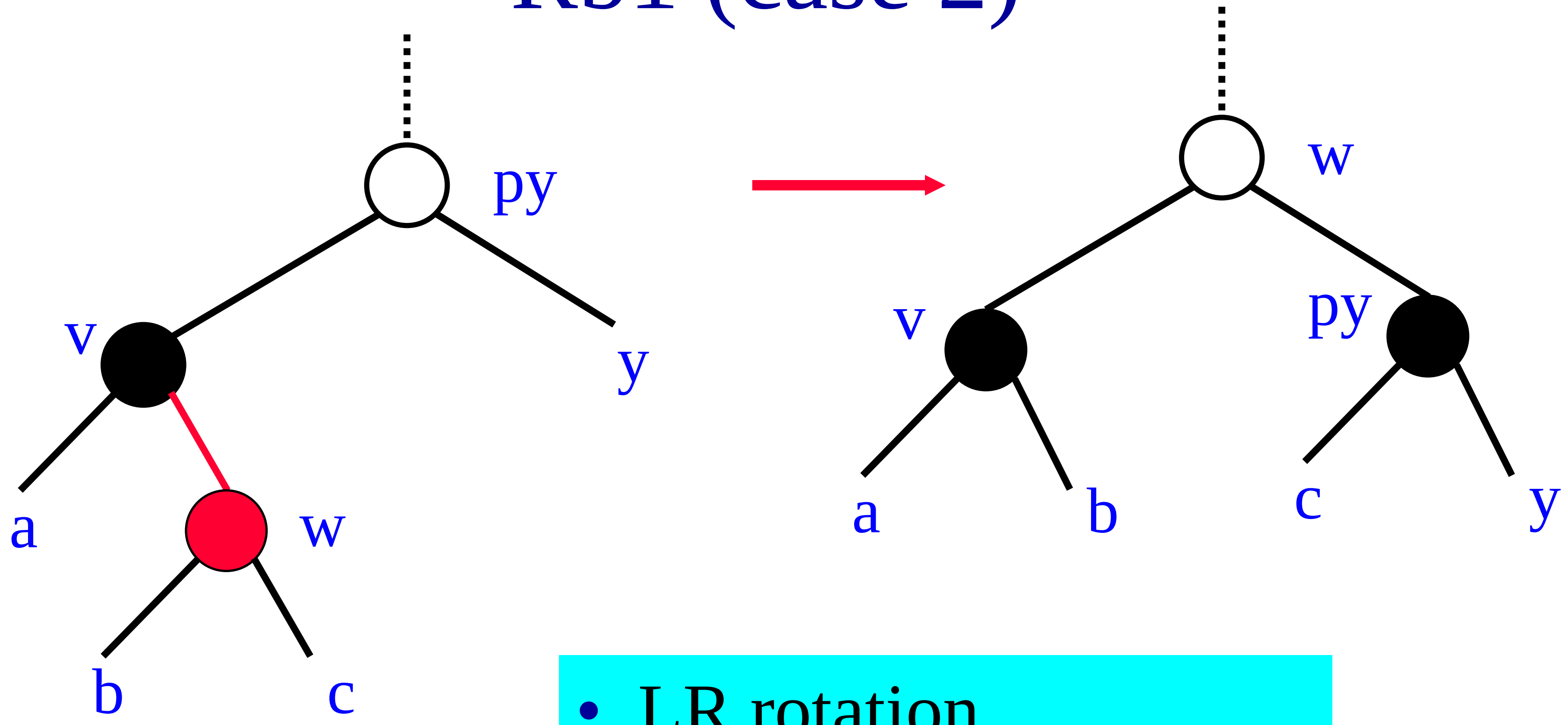
- Color change.
- Deficiency eliminated.
- Done!

Rb1 (case 1)



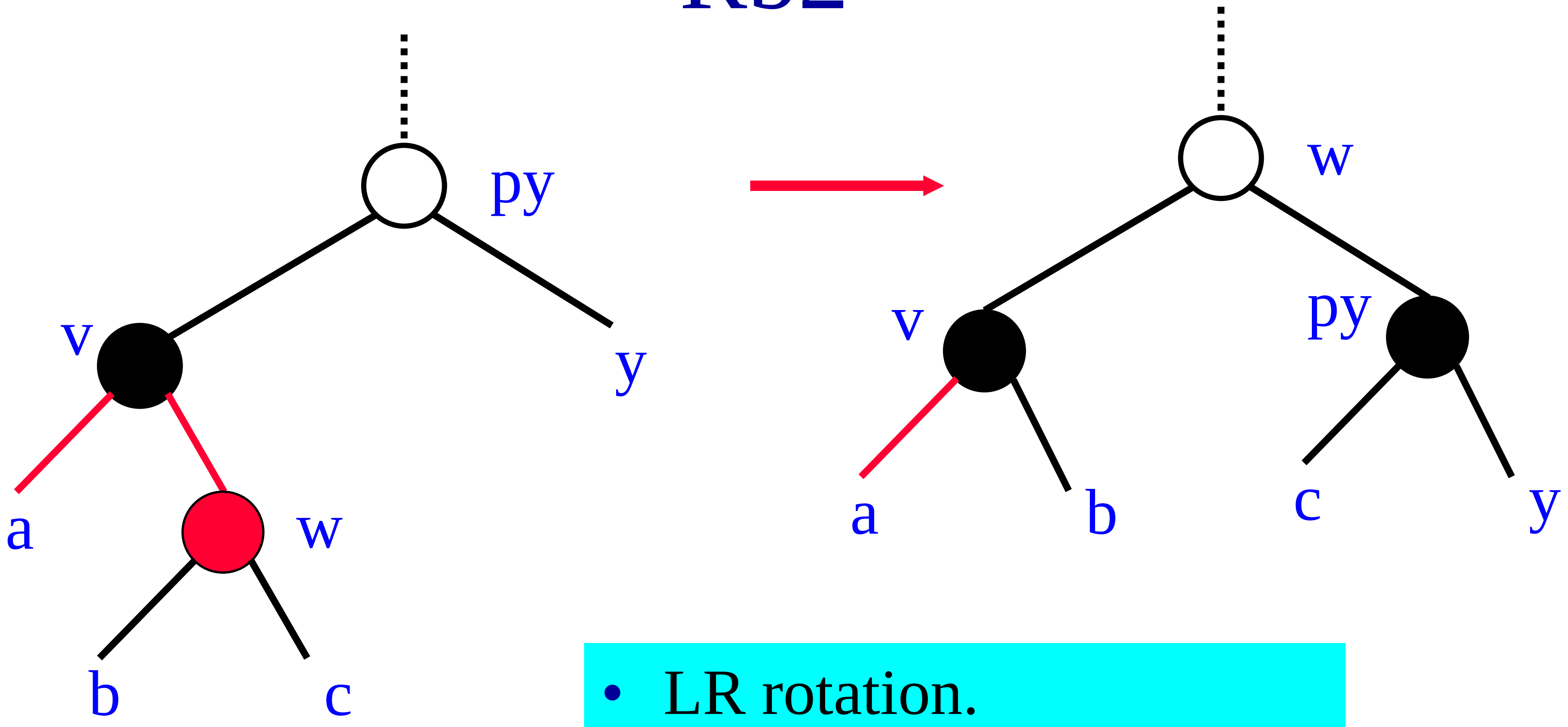
- LL rotation.
- Deficiency eliminated.
- Done!

Rb1 (case 2)



- LR rotation.
- Deficiency eliminated.
- Done!

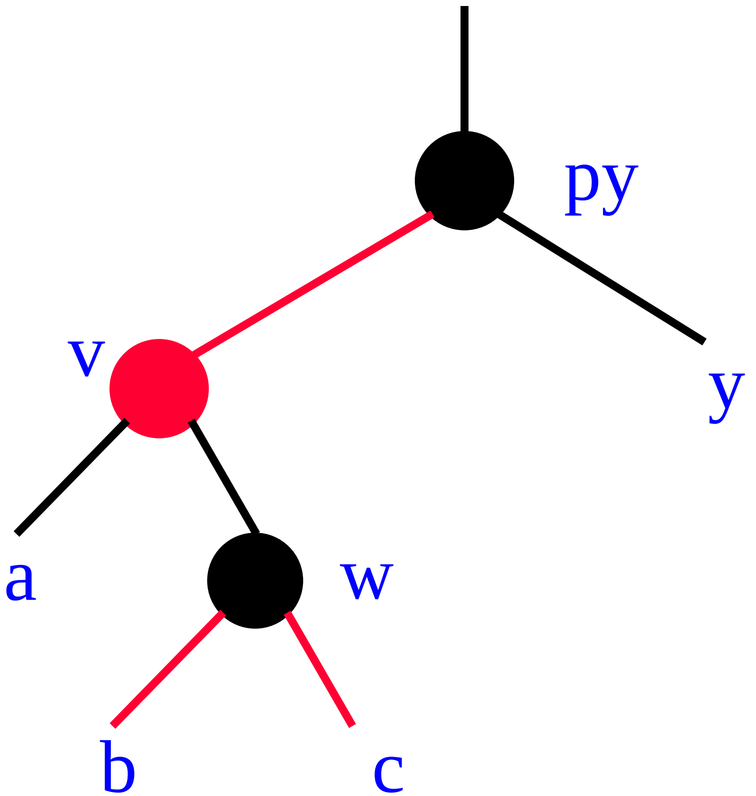
Rb2



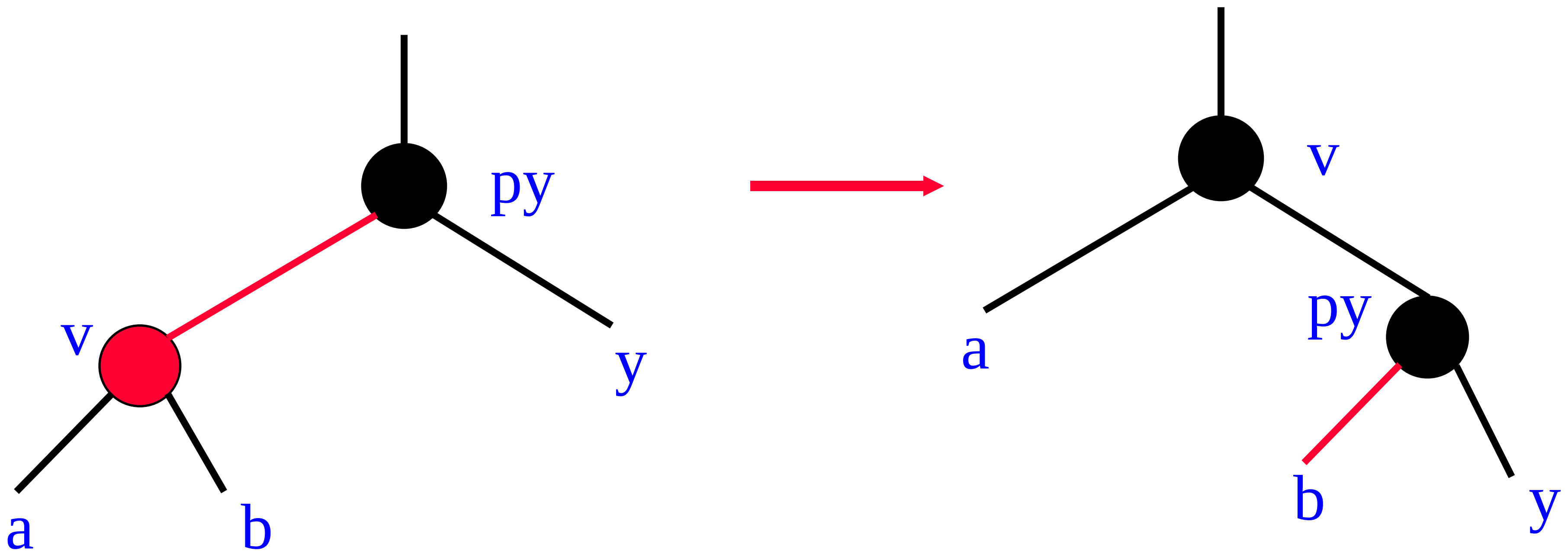
- LR rotation.
- Deficiency eliminated.
- Done!

Rr(n)

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

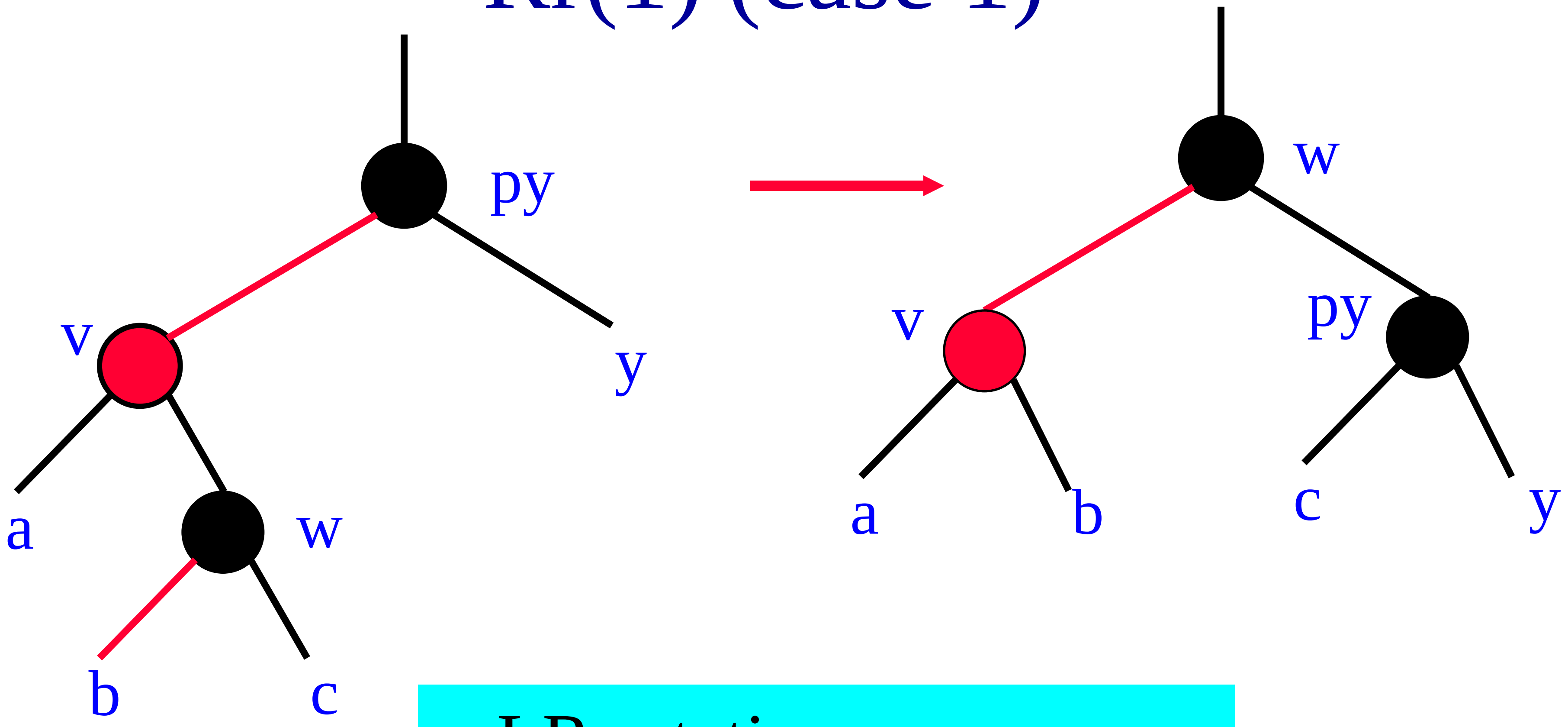


Rr(0)



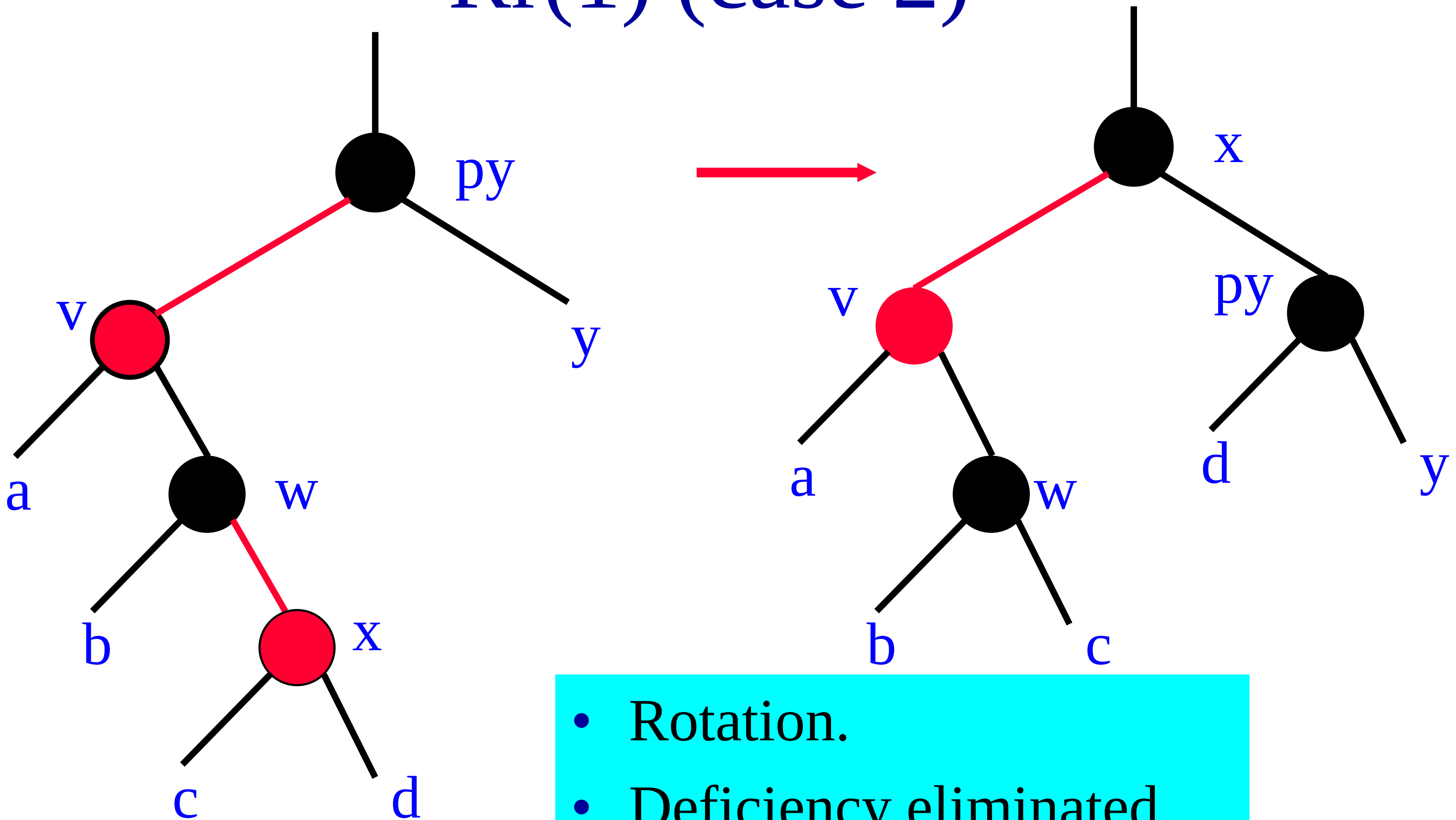
- LL rotation.
- Done!

Rr(1) (case 1)



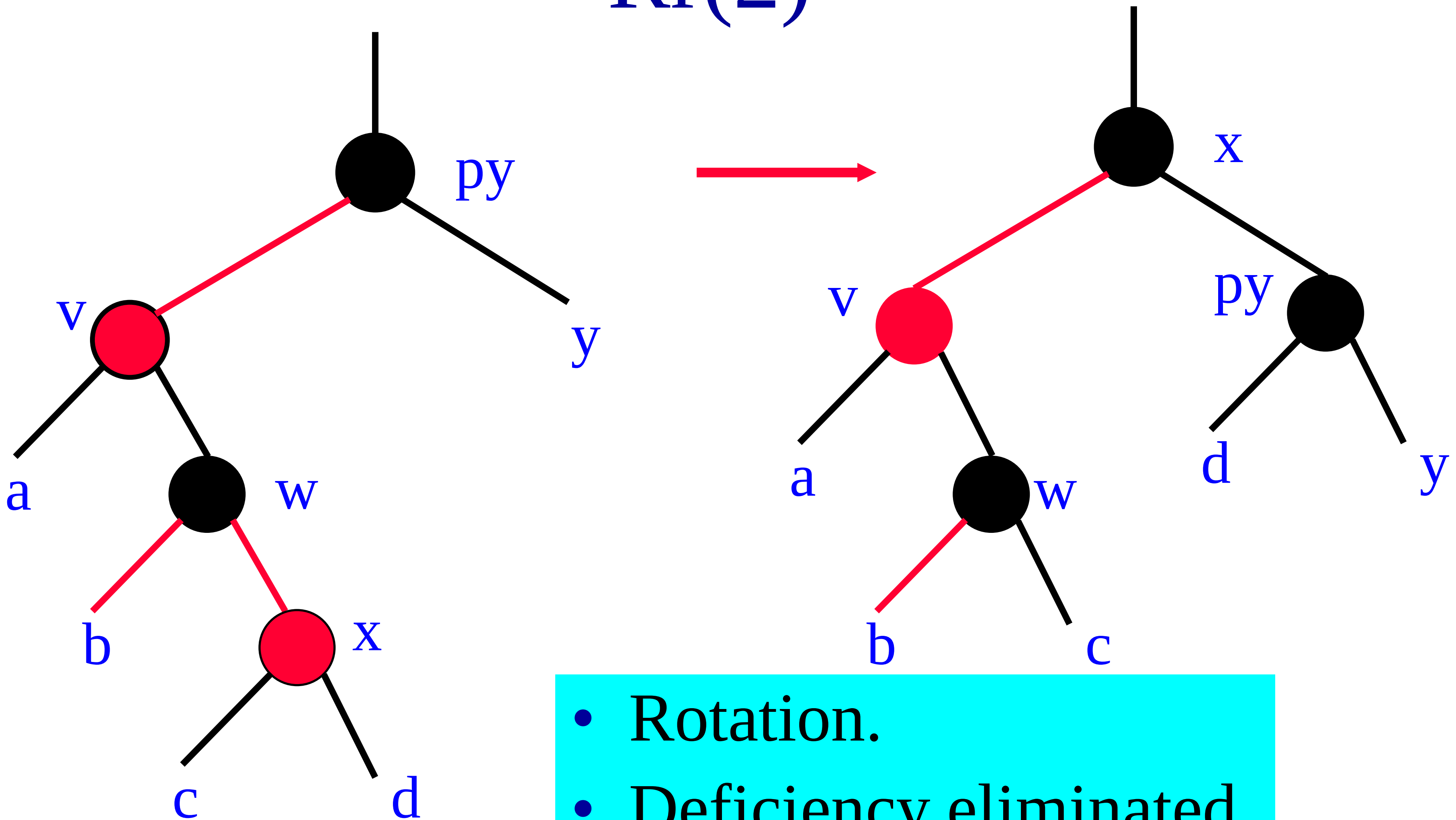
- LR rotation.
- Deficiency eliminated.
- Done!

Rr(1) (case 2)



- Rotation.
- Deficiency eliminated.
- Done!

Rr(2)



- Rotation.
- Deficiency eliminated.
- Done!