Chapter Fifteen

Colored Tree

Red Black Trees

Colored Nodes Definition

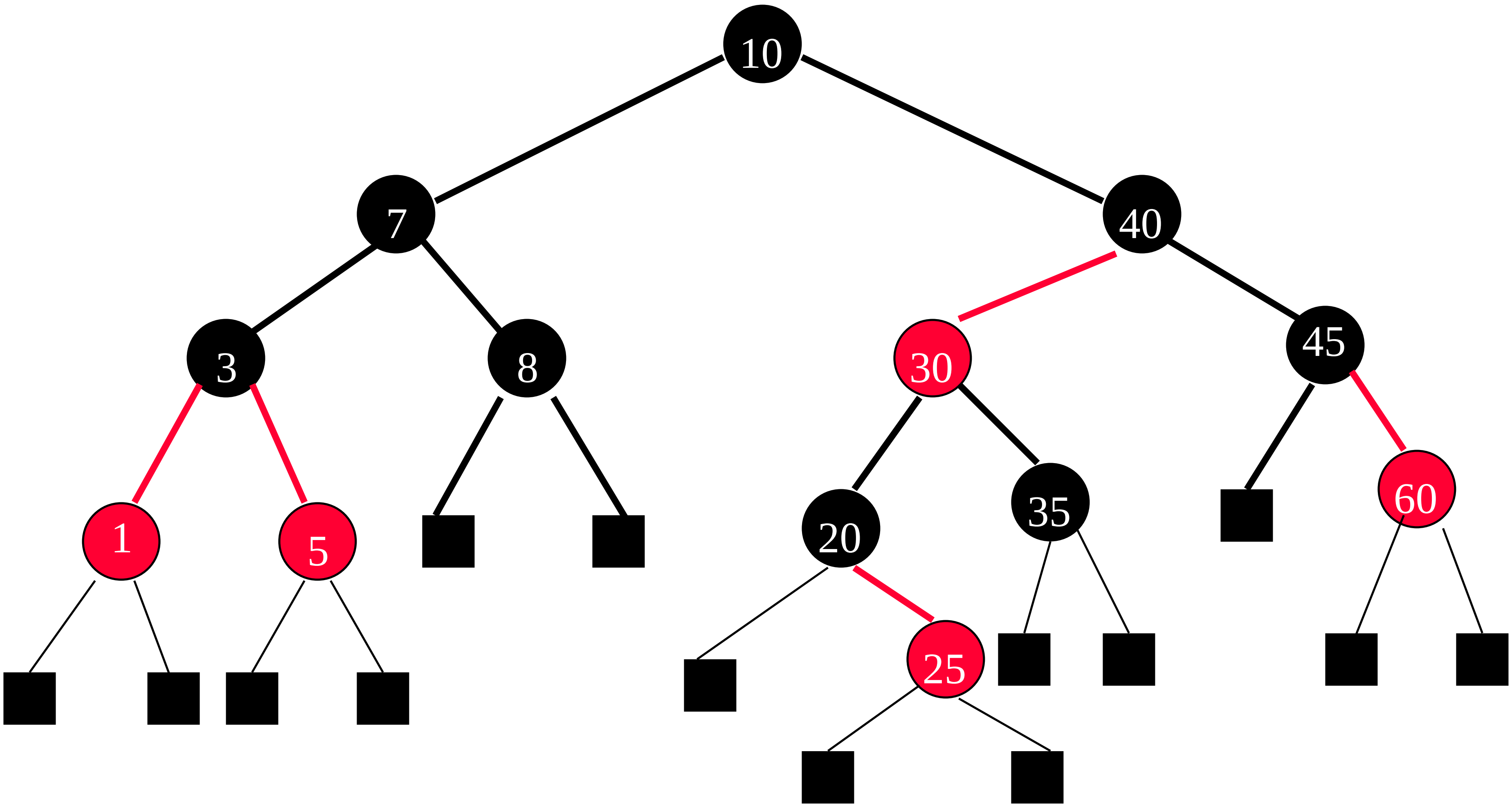
- Binary search tree.
- Each node is colored red or black.
- Root and all external nodes are black.
- No root-to-external-node path has two consecutive red nodes.
- <u>All root-to-external-node paths have the same number of black nodes</u>

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



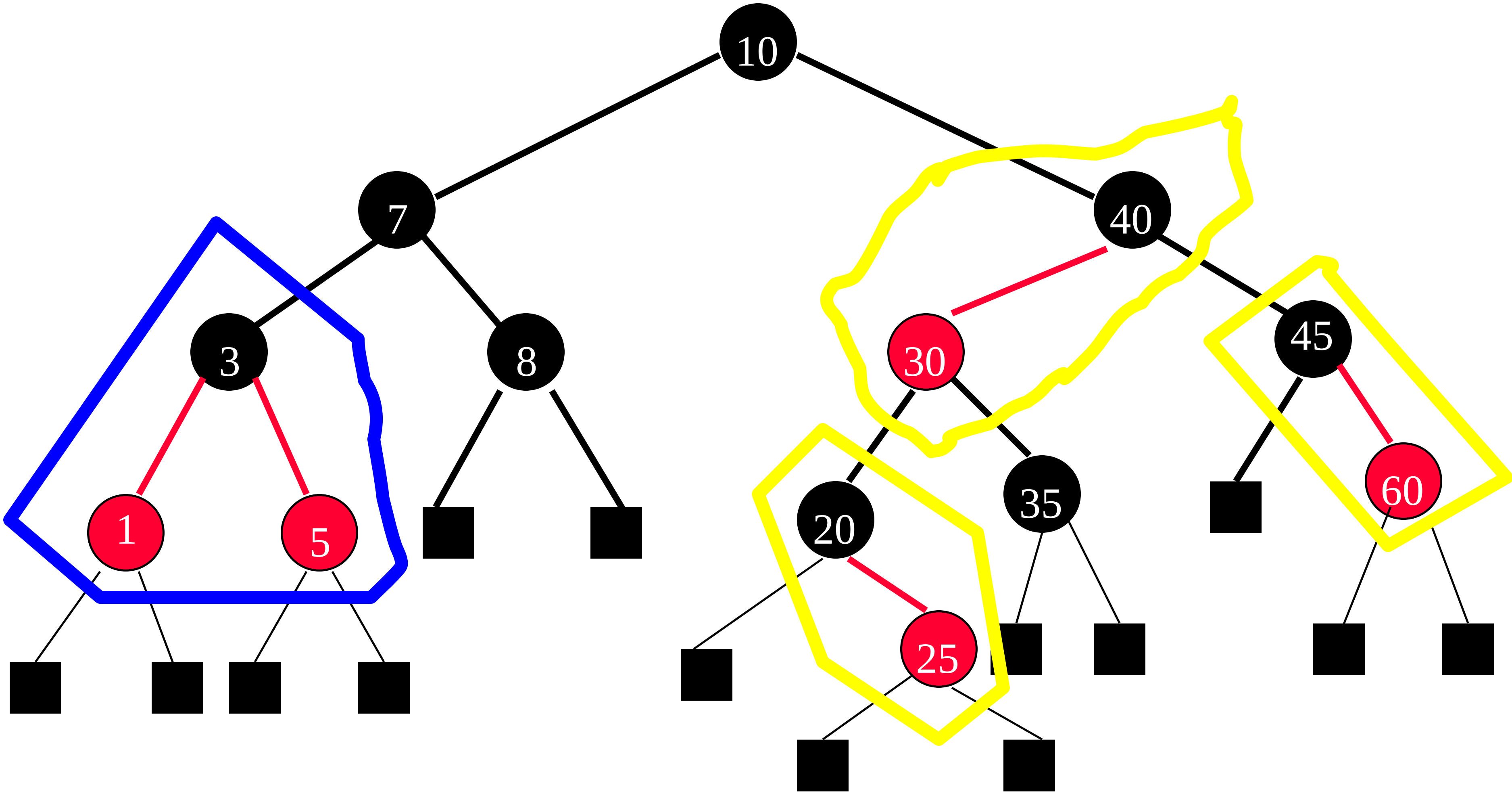
Properties

- 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.

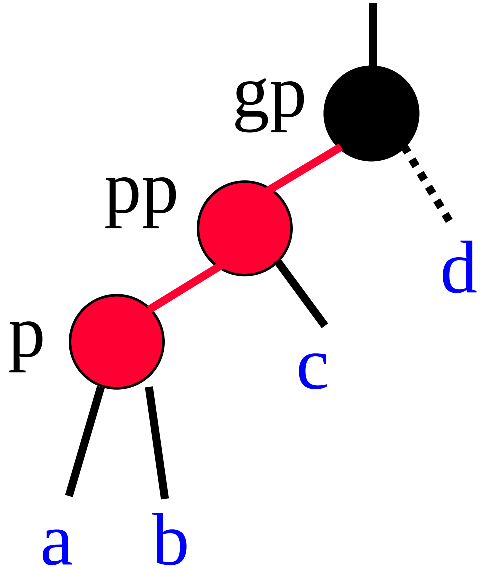
Properties

- 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

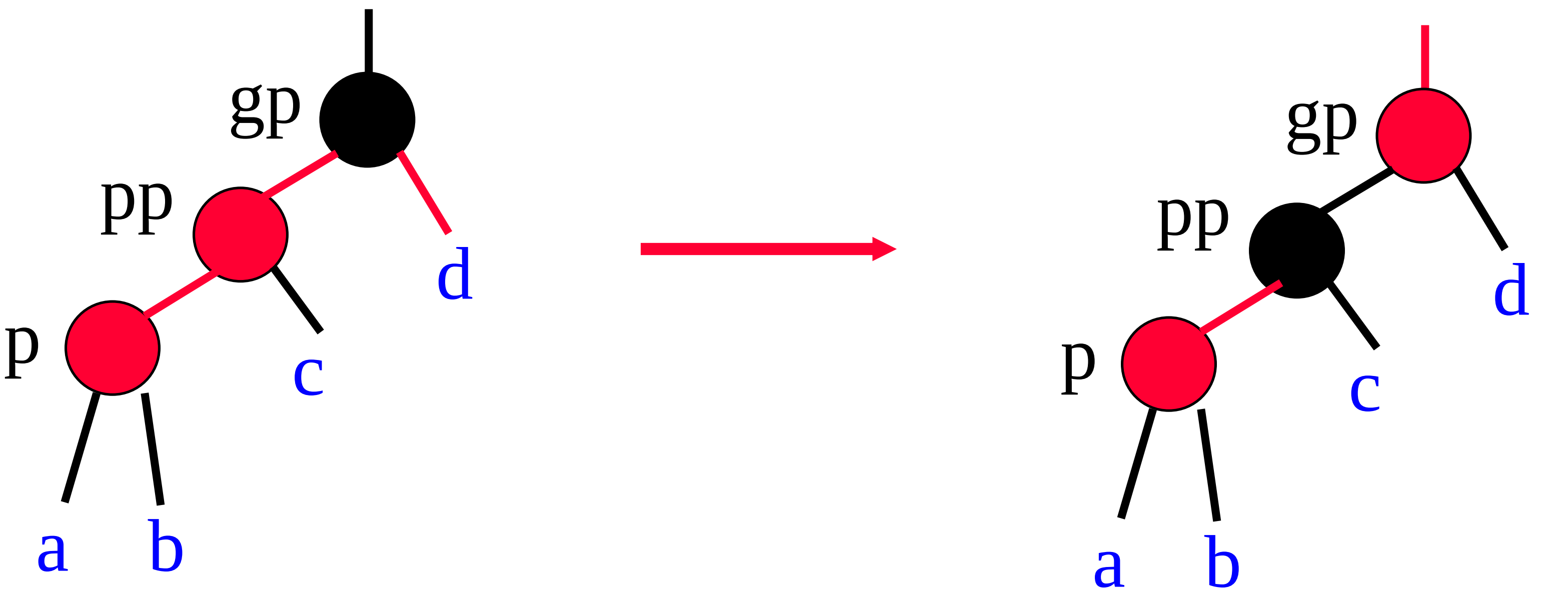


LLb

- 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

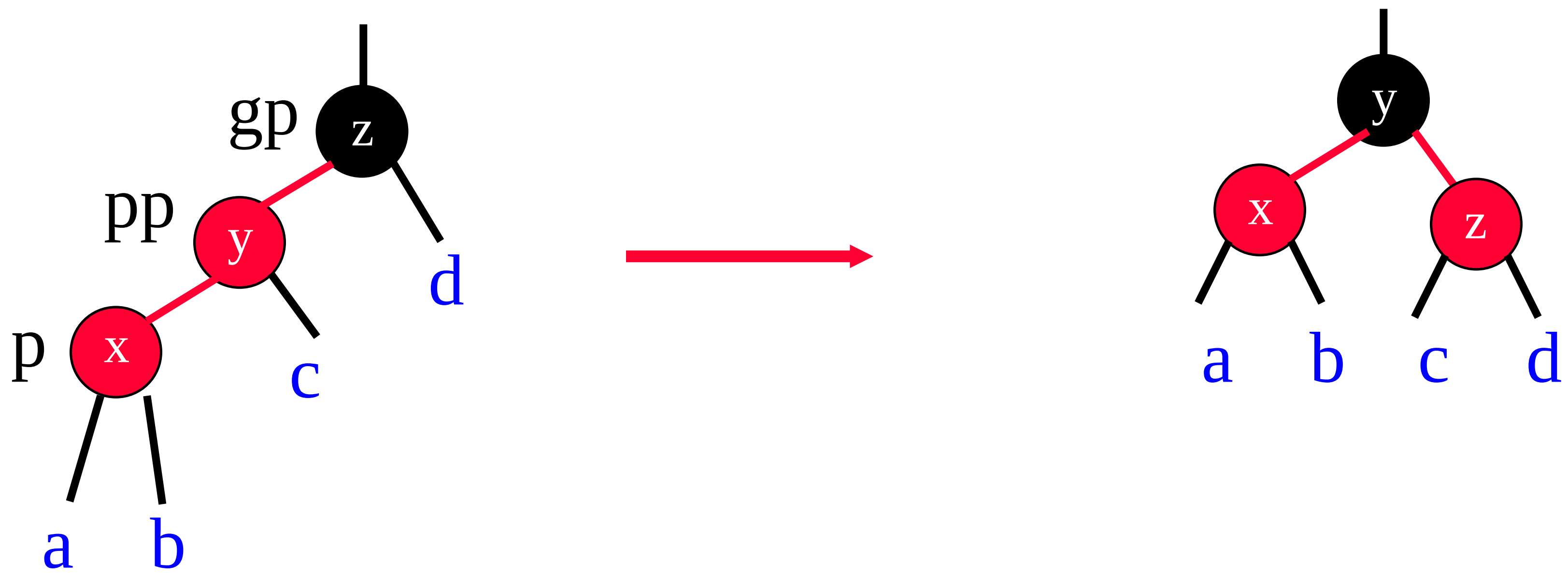
- Color flip.



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

LLb

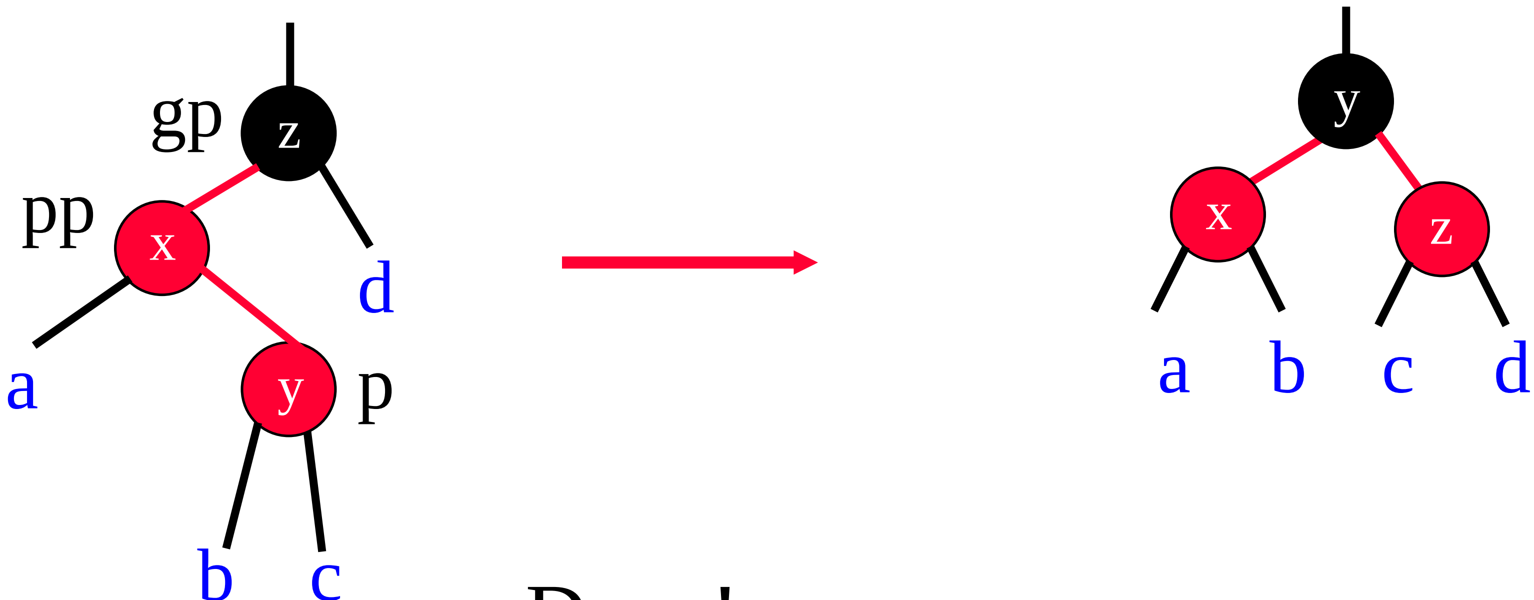
- Rotate.



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

LRb

- Rotate.

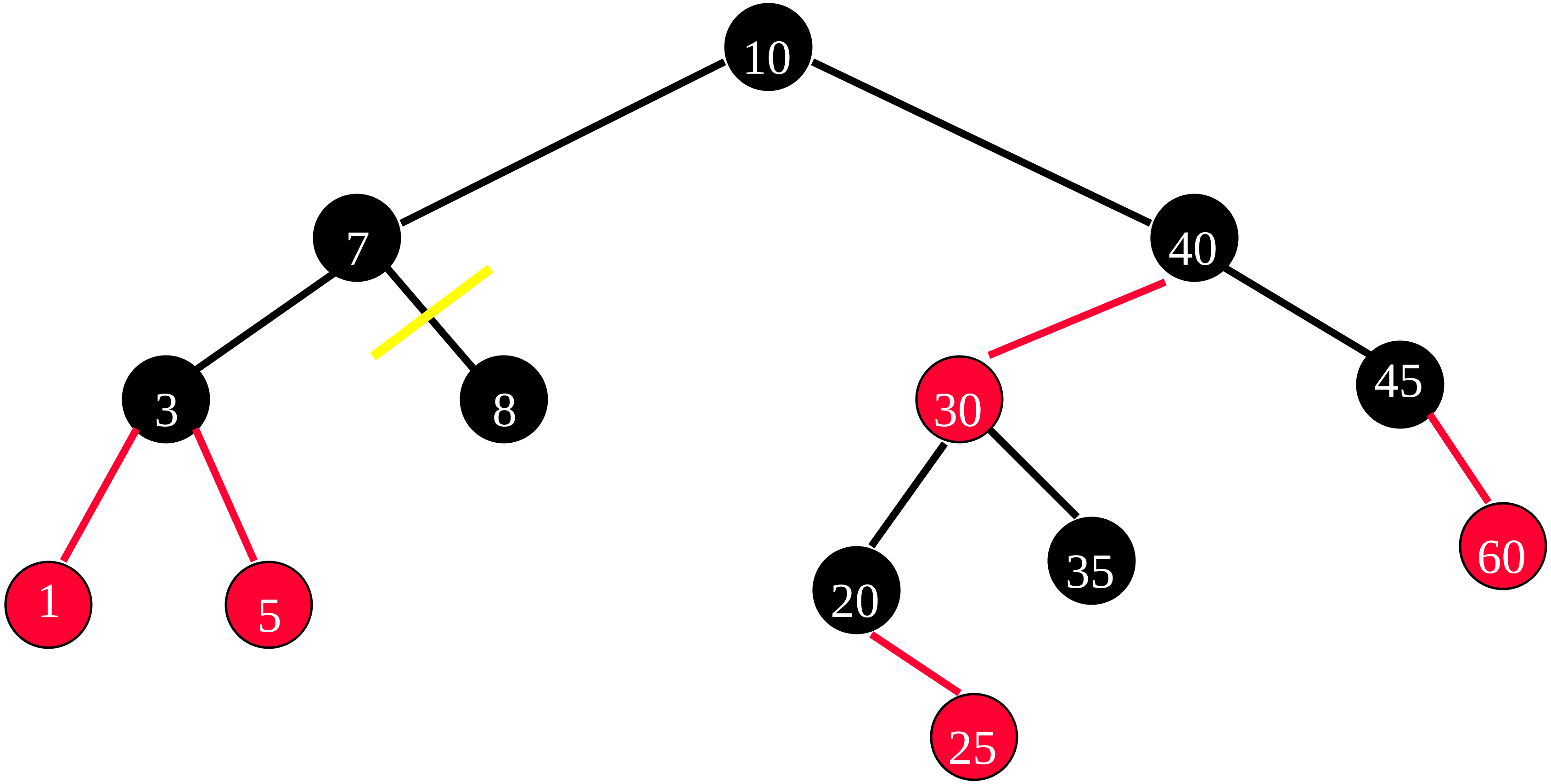


- Done!
- Same as LR rotation of AVL tree.
- RRb and RLb are symmetric.

Delete

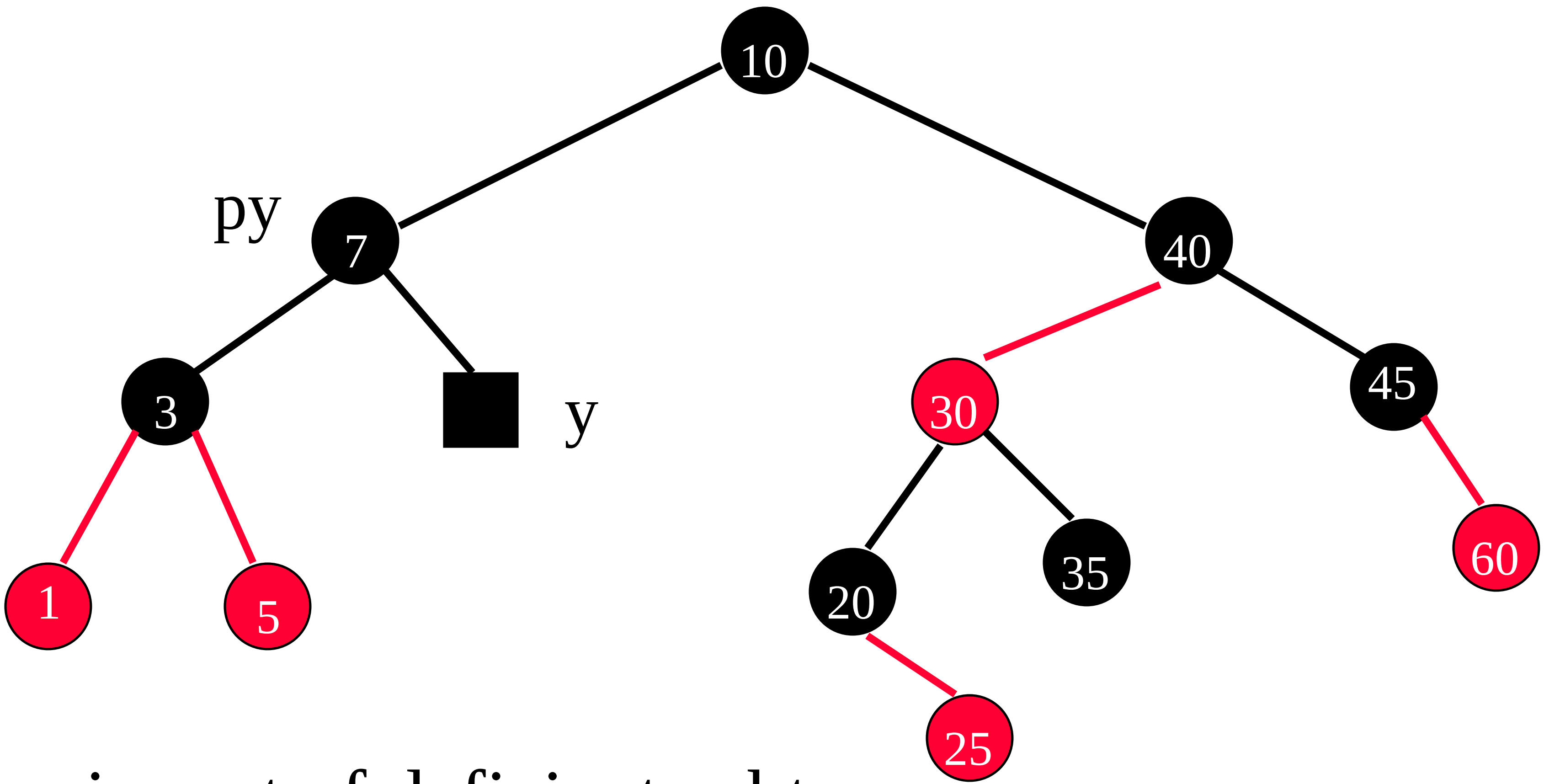
- Delete as for unbalanced binary search tree.
- If red node deleted, no rebalancing needed.
- If black node deleted, a subtree becomes one black pointer (node) deficient.

Delete A Black Leaf



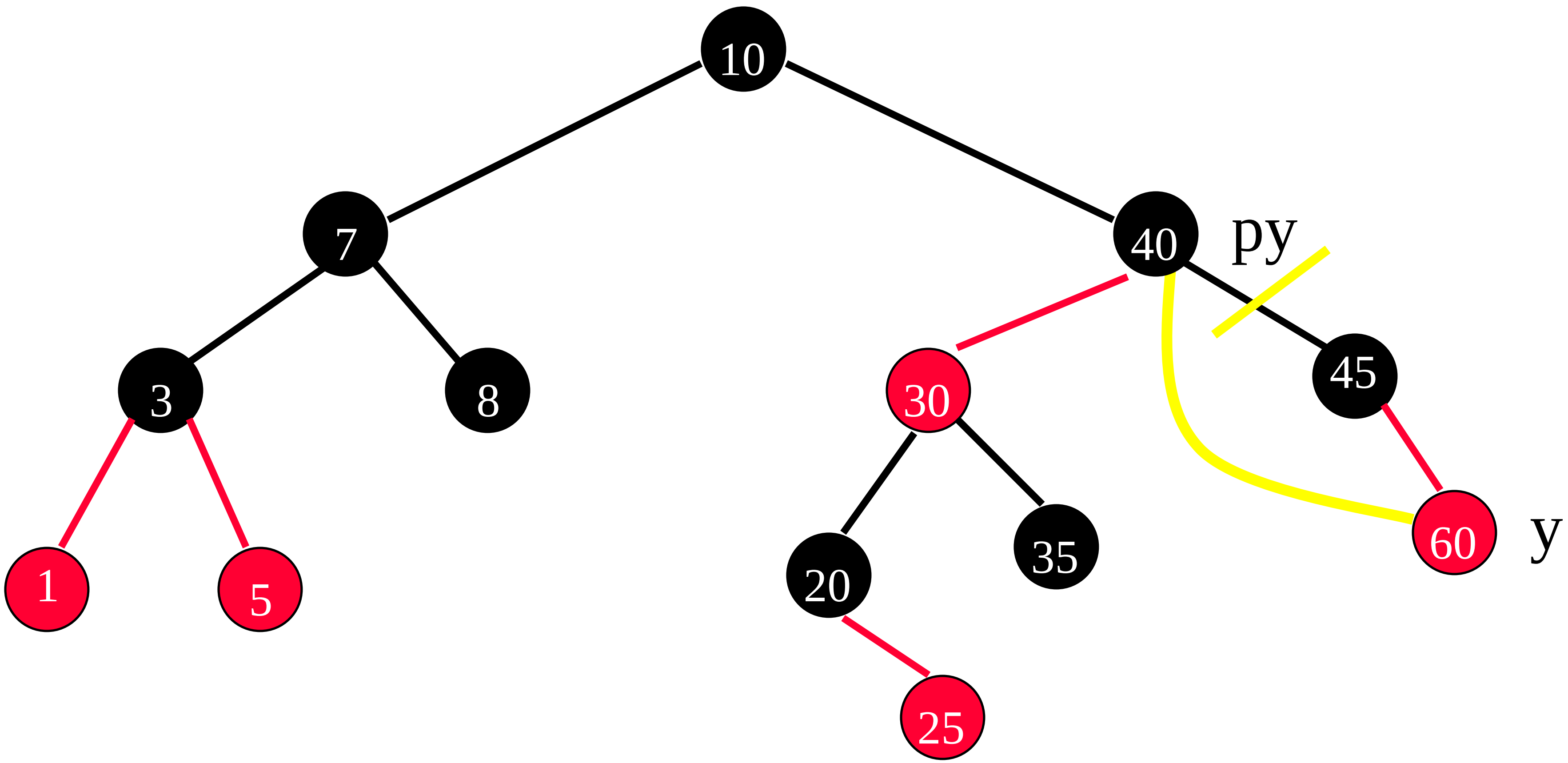
- Delete 8.

Delete A Black Leaf



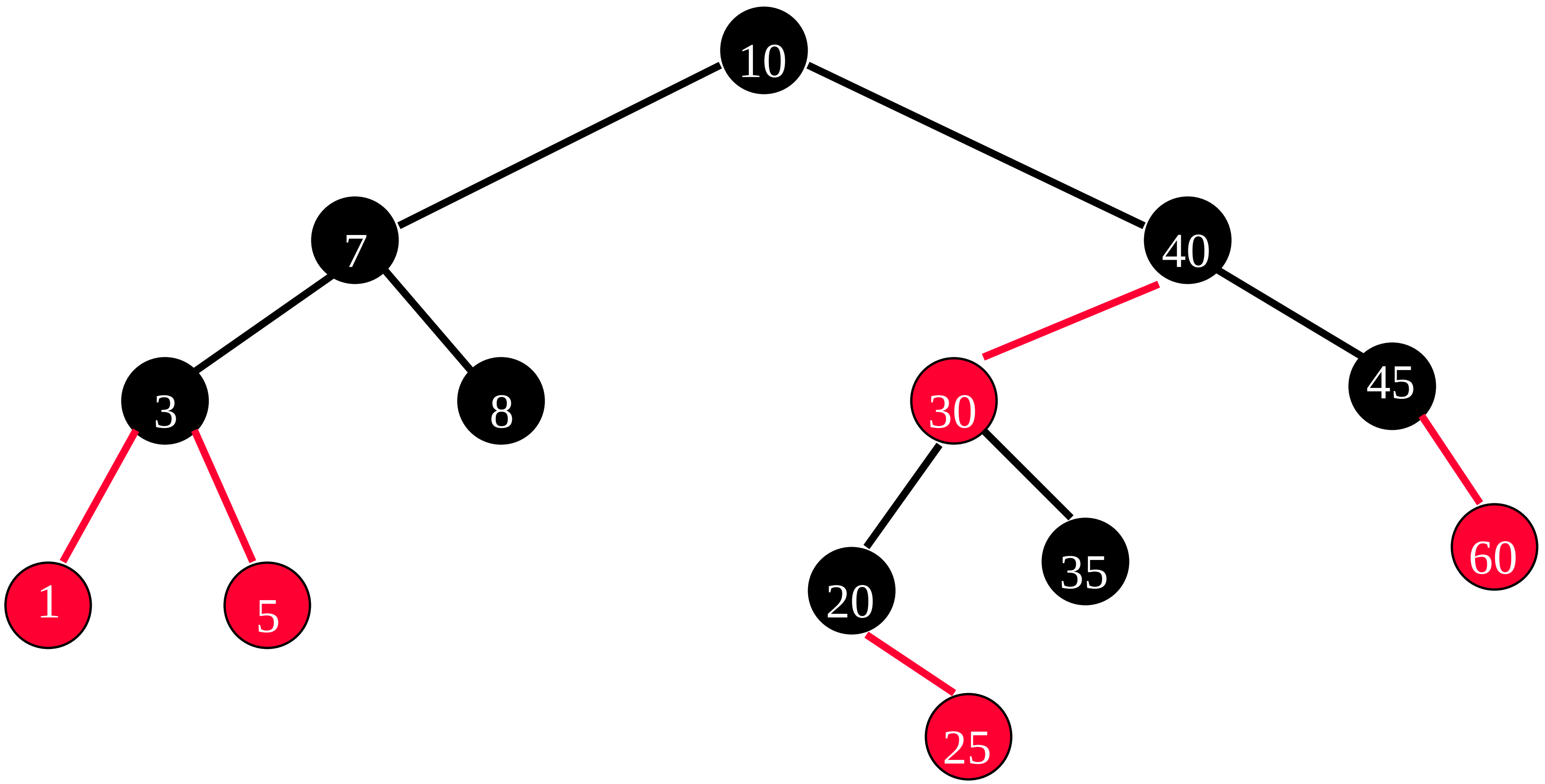
- y is root of deficient subtree.
- py is parent of y.

Delete A Black Degree 1 Node



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

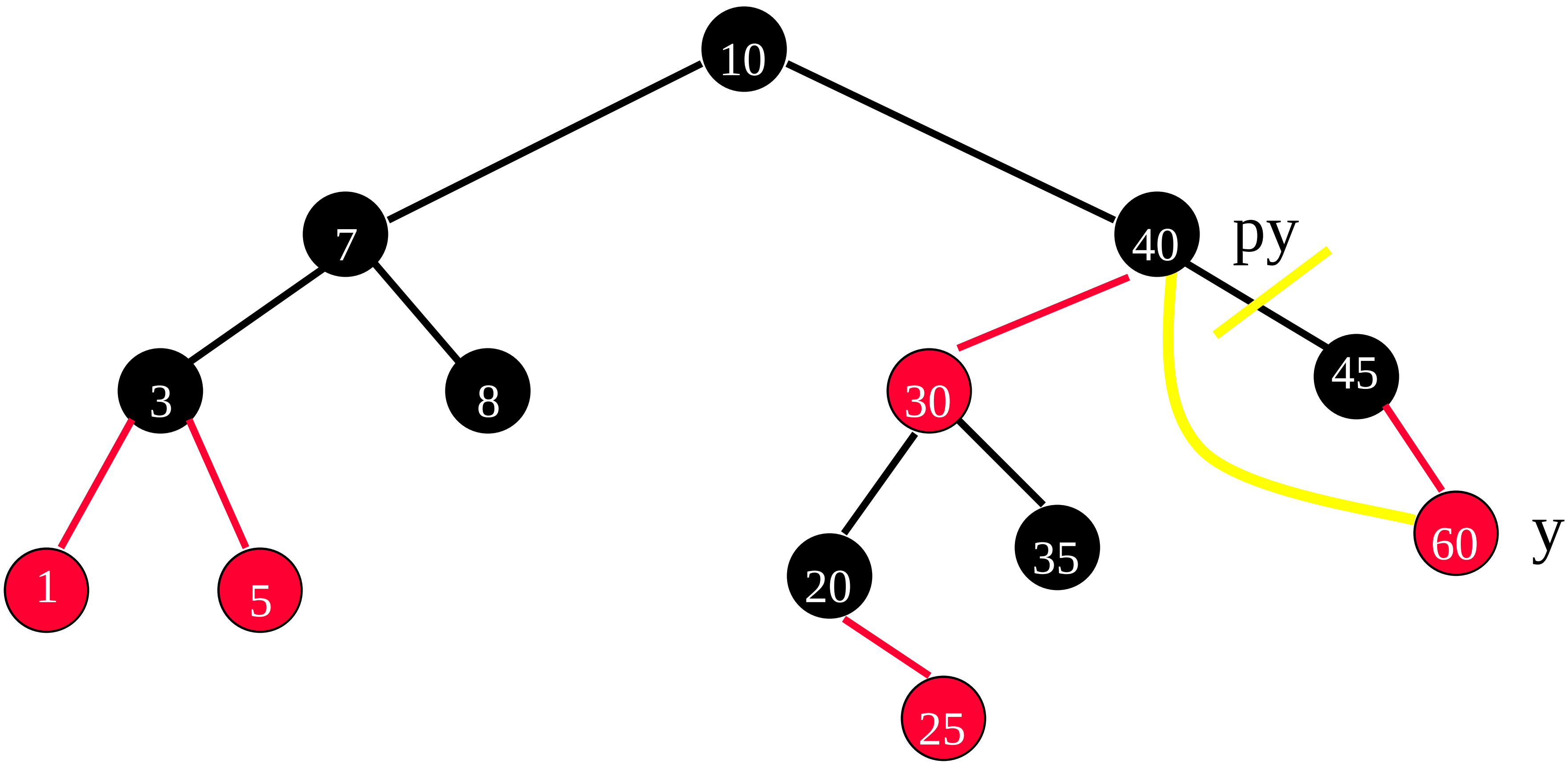
Delete A Black Degree 2 Node



- Not possible, degree 2 nodes are never deleted.

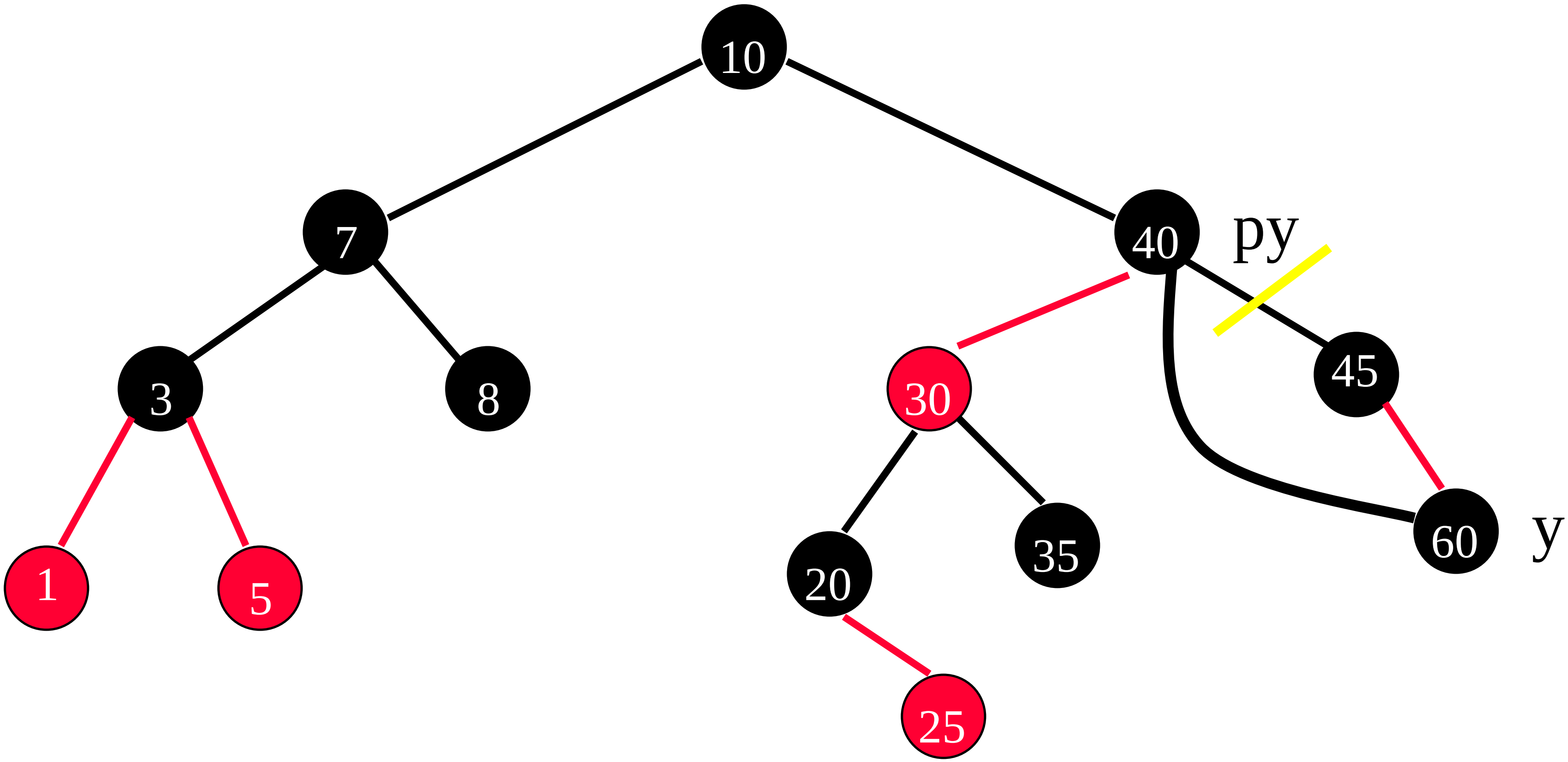
Rebalancing Strategy

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



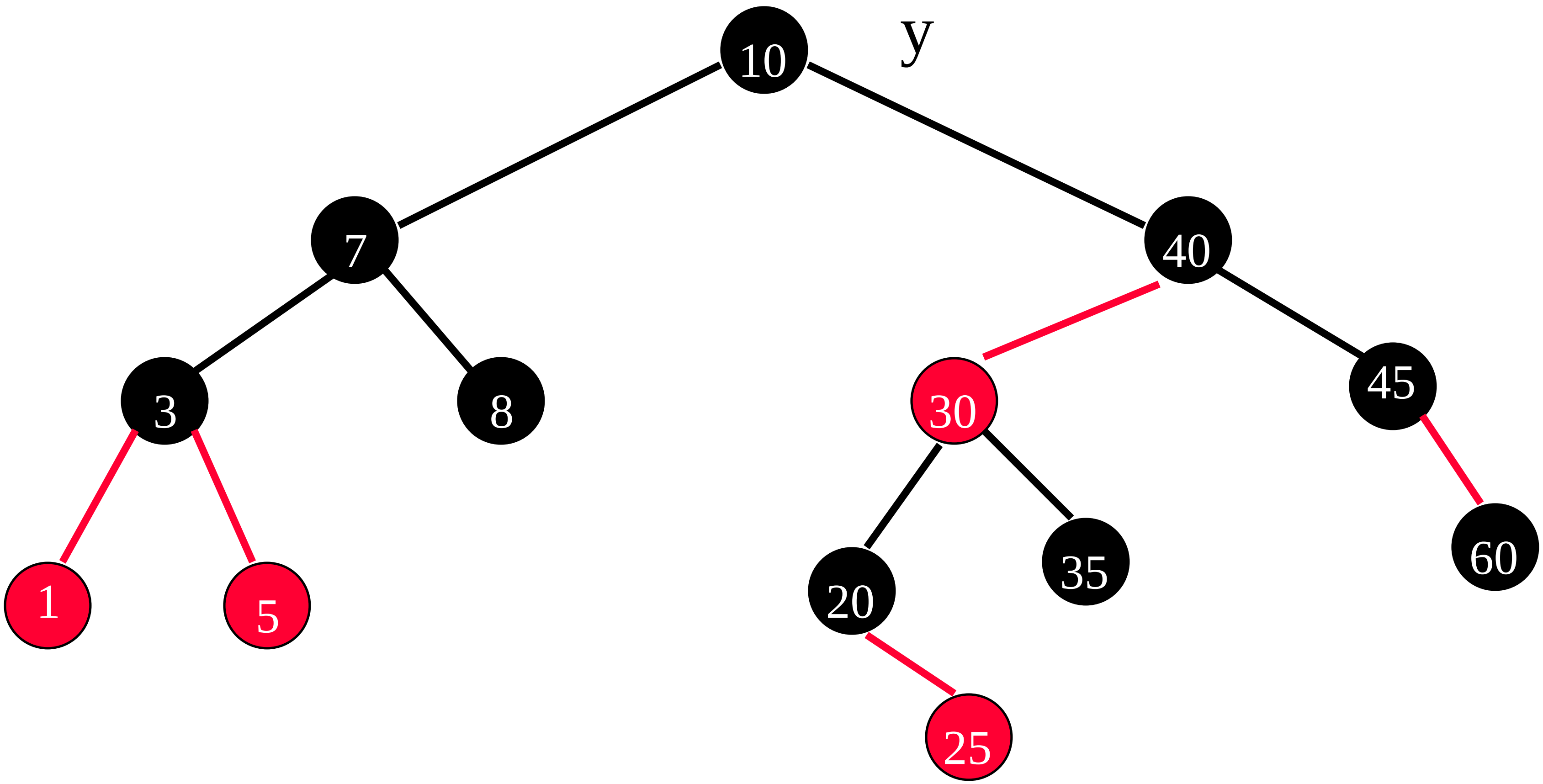
Rebalancing Strategy

- Now, no subtree is deficient. Done!



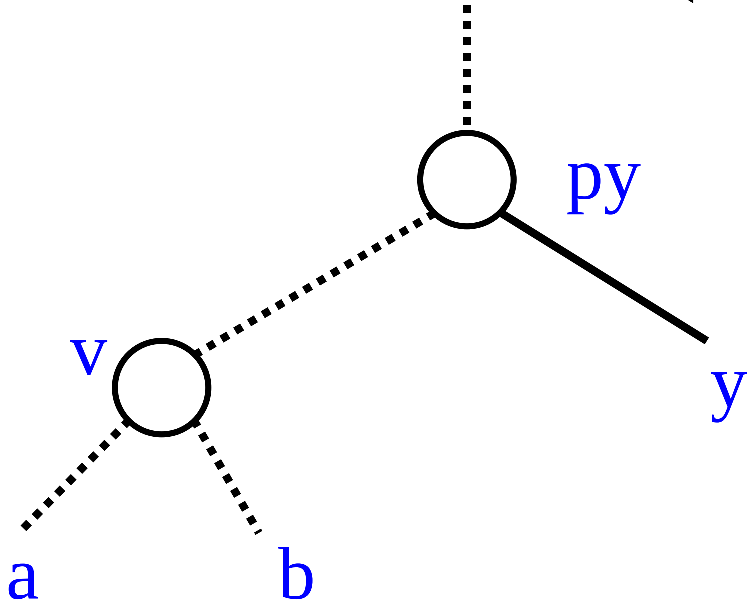
Rebalancing Strategy

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



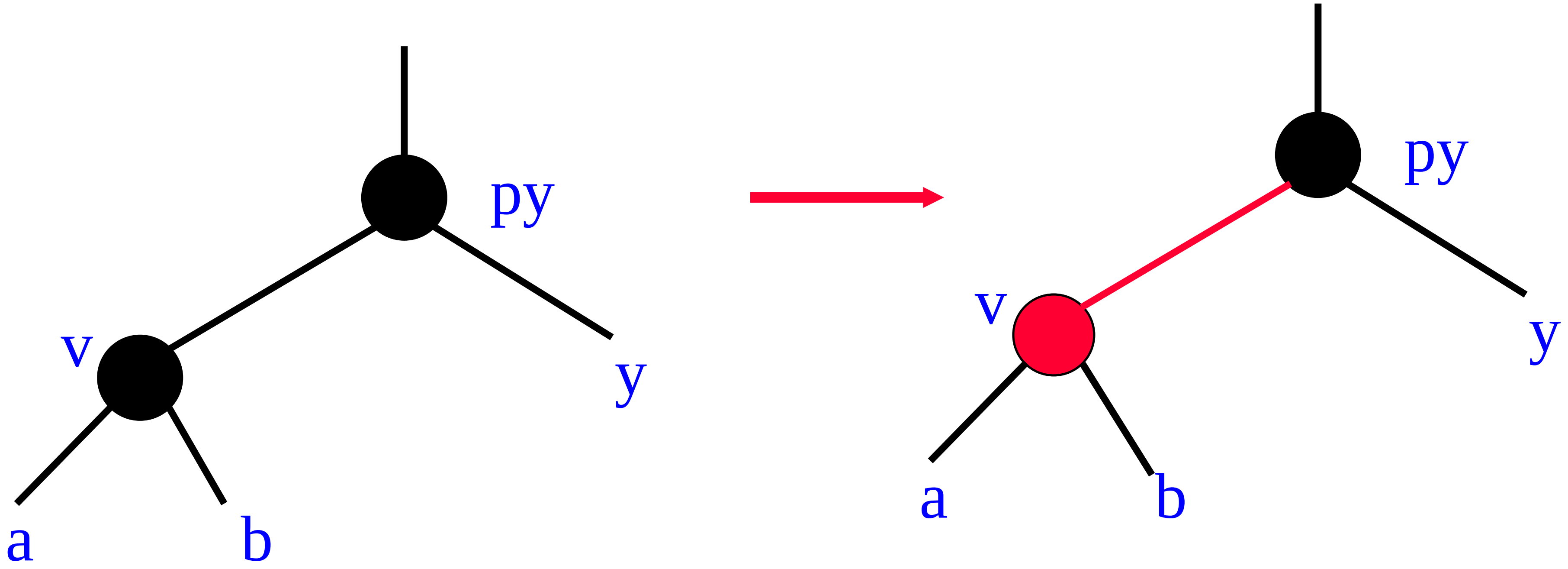
Rebalancing Strategy

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



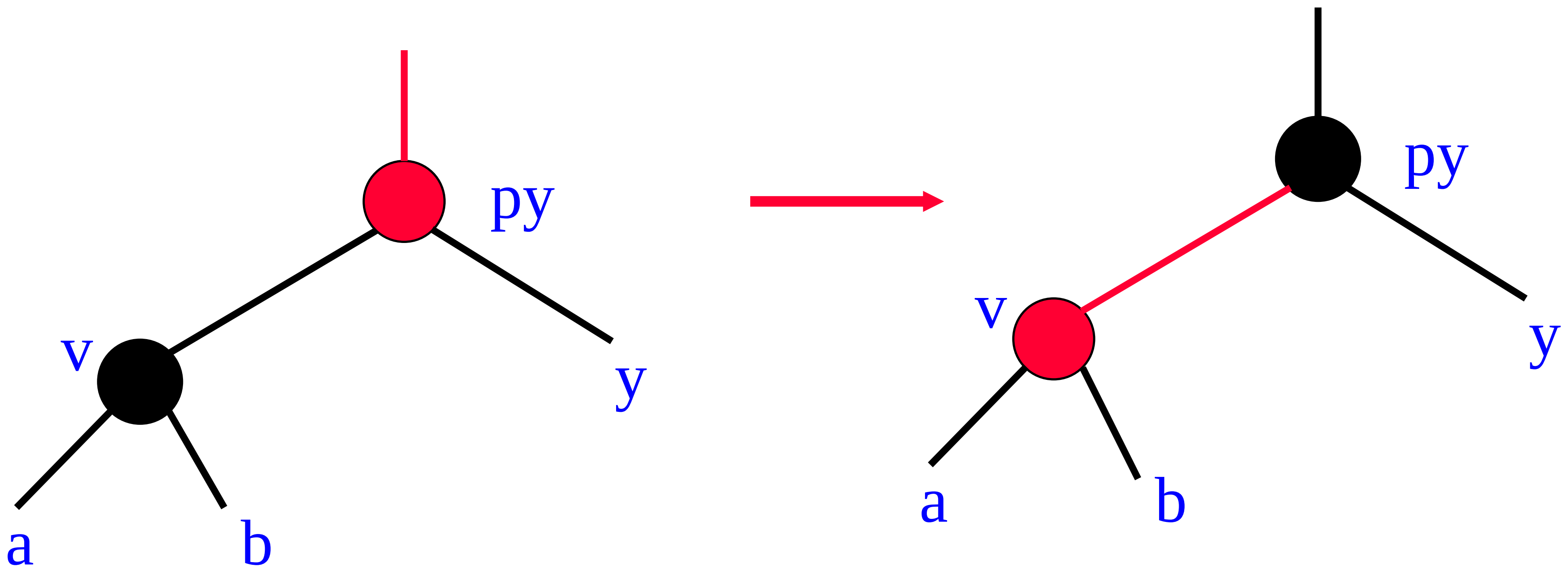
- Xcn
  - y is right child of py => X = R.
  - Pointer to v is black => c = b.
  - v has 1 red child => n = 1.

Rb0 (case 1)



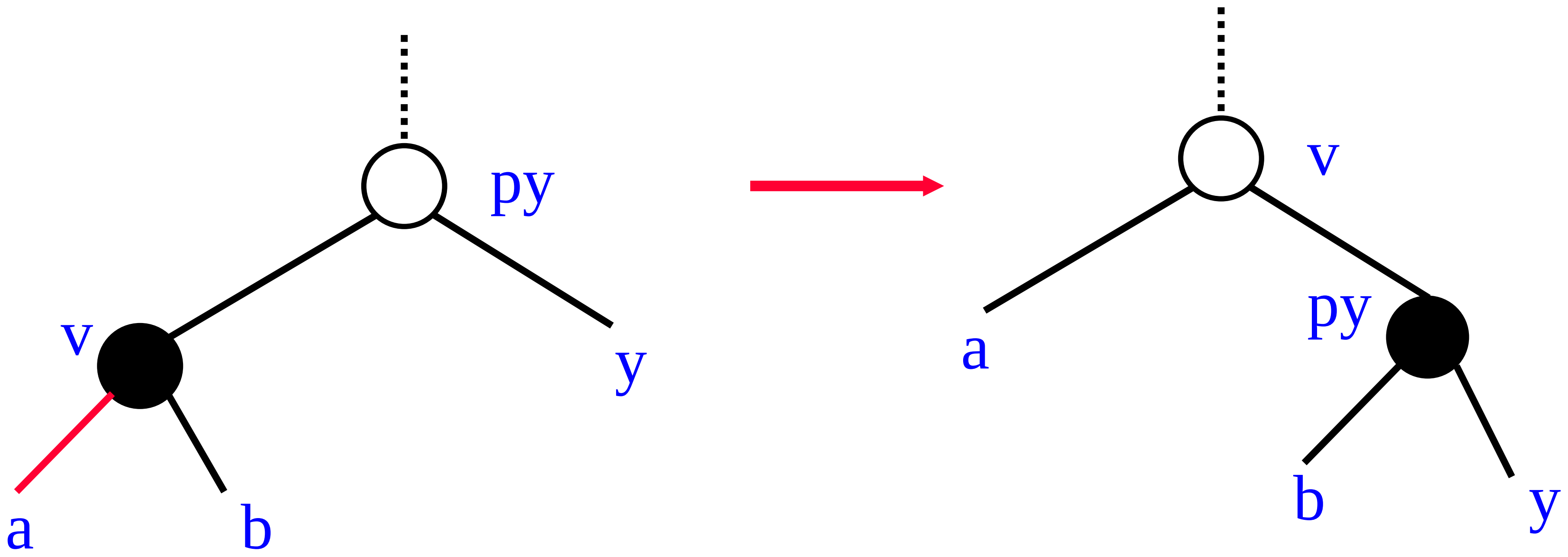
- Color change.
- Now, py is root of deficient subtree.
- Continue!

Rb0 (case 2)



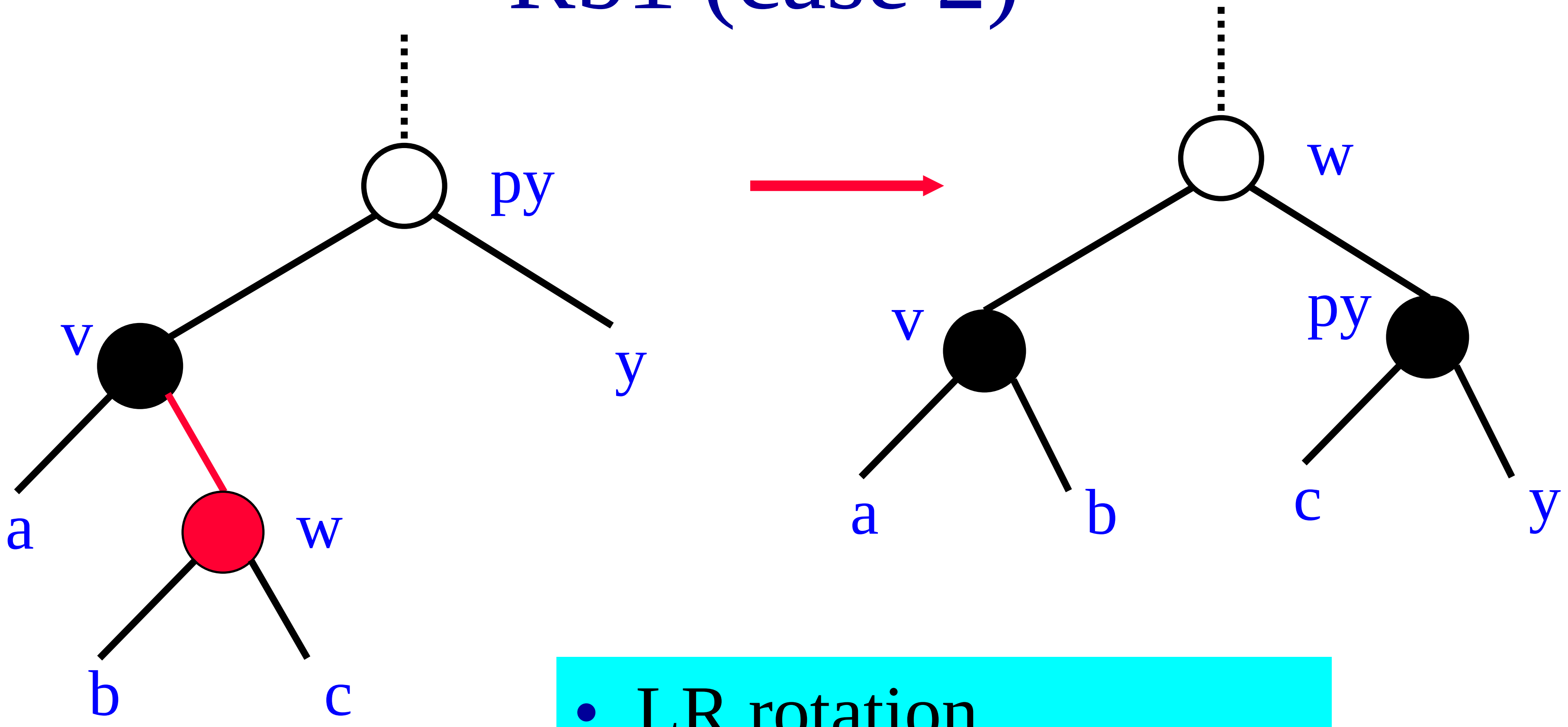
- Color change.
- Deficiency eliminated.
- Done!

Rb1 (case 1)



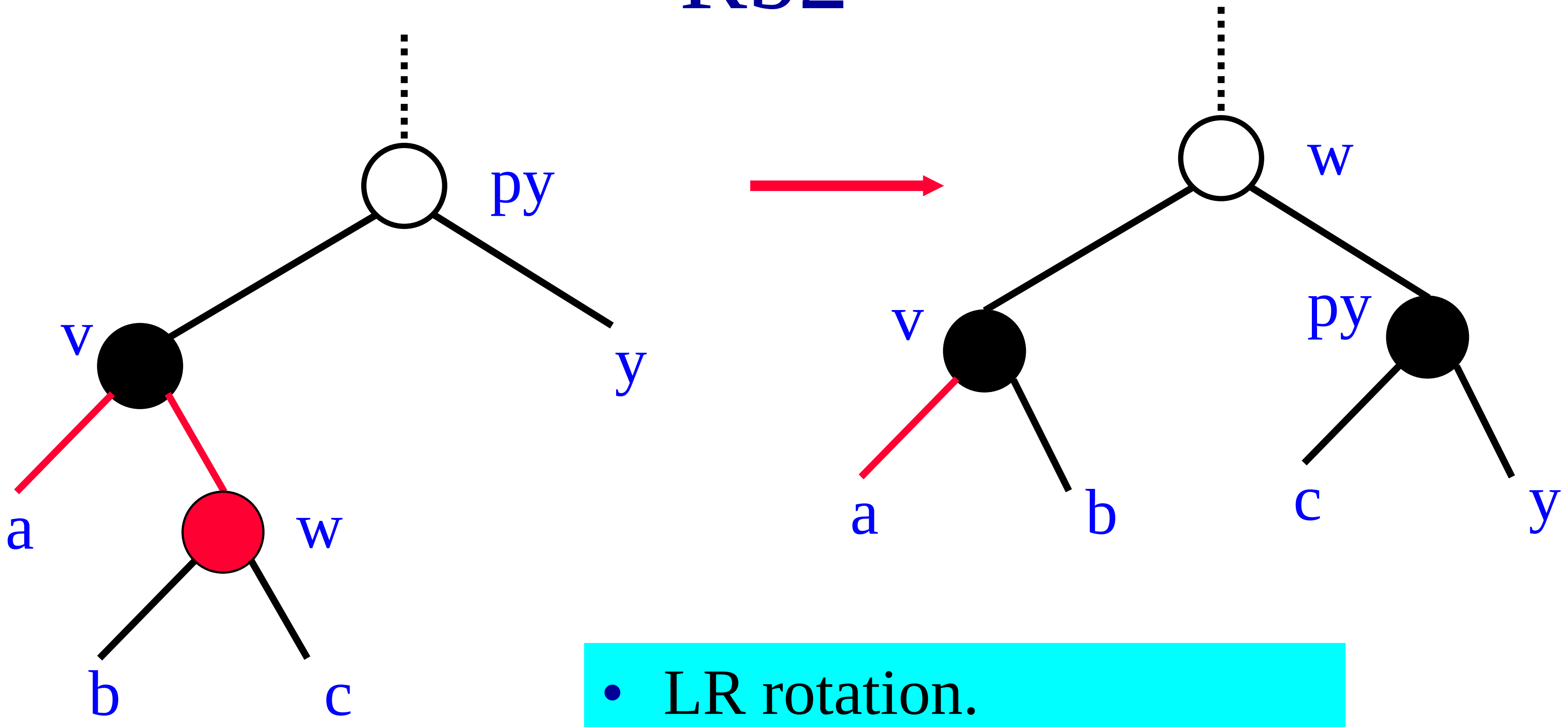
- LL rotation.
- Deficiency eliminated.
- Done!

Rb1 (case 2)



- LR rotation.
- Deficiency eliminated.
- Done!

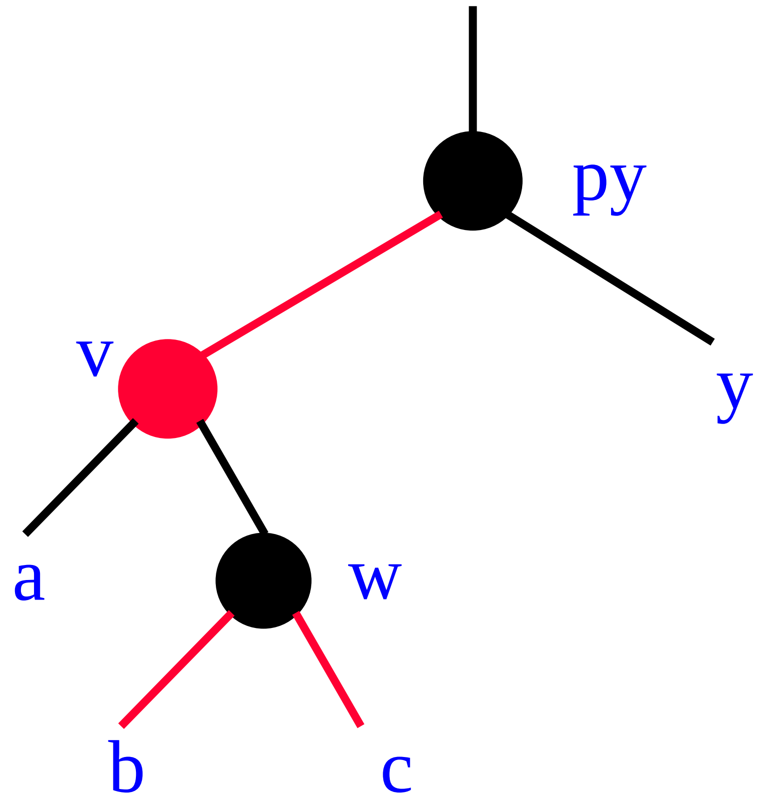
Rb2



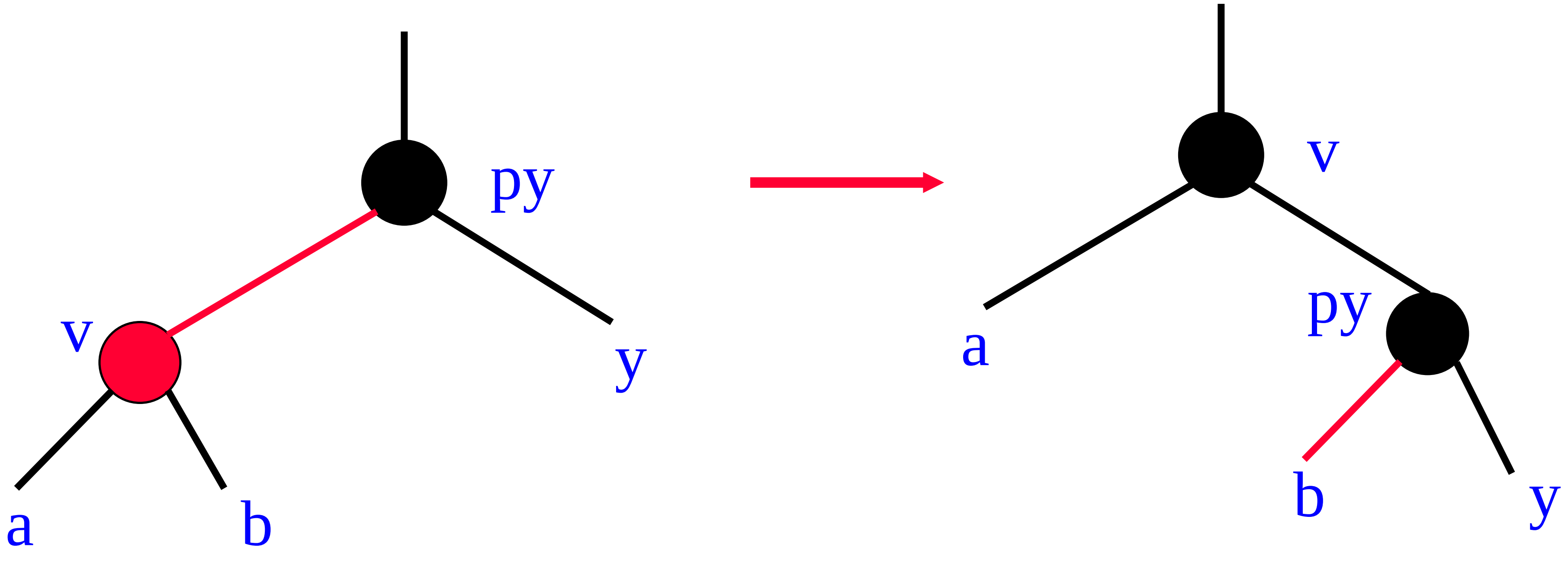
- LR rotation.
- Deficiency eliminated.
- Done!

Rr(n)

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

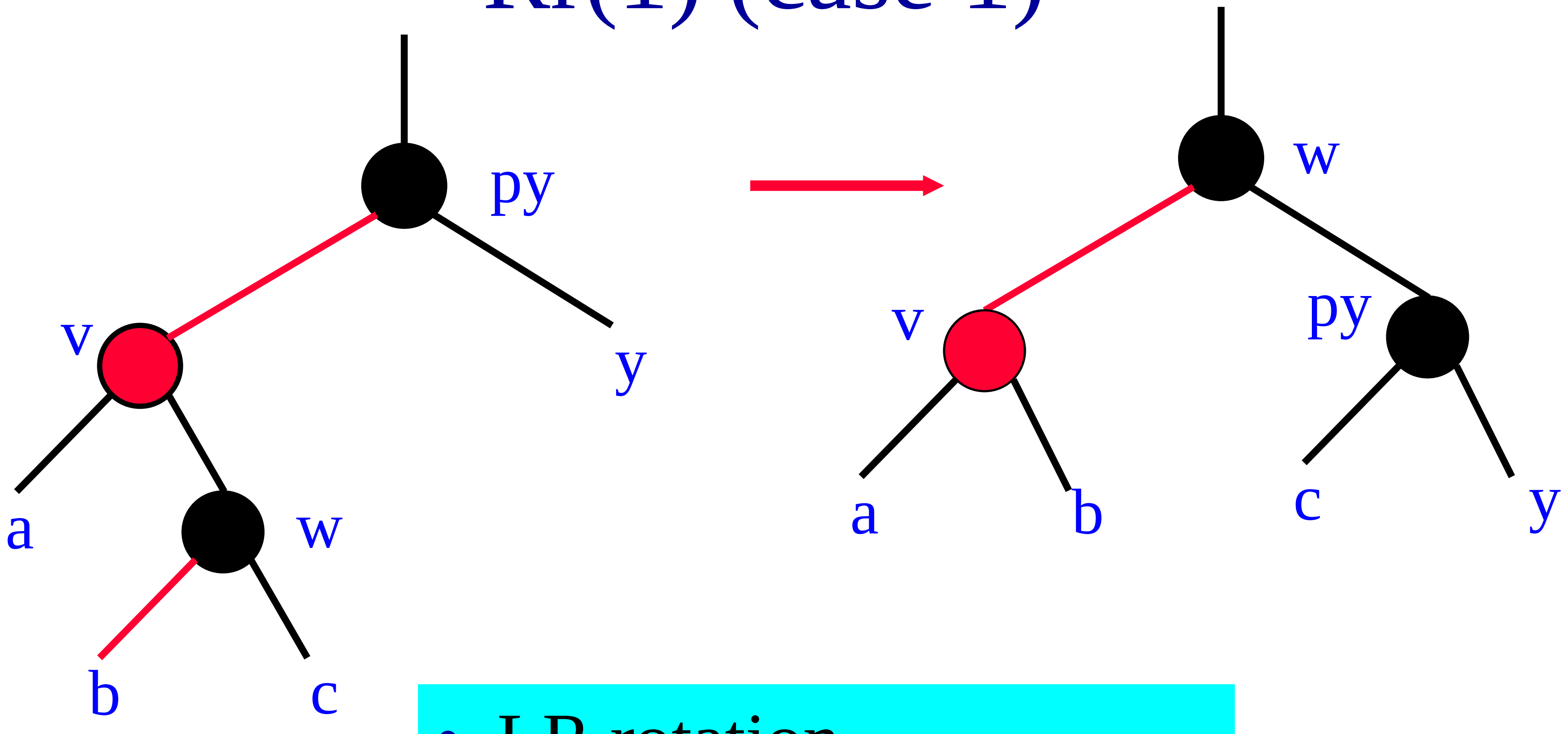


Rr(0)



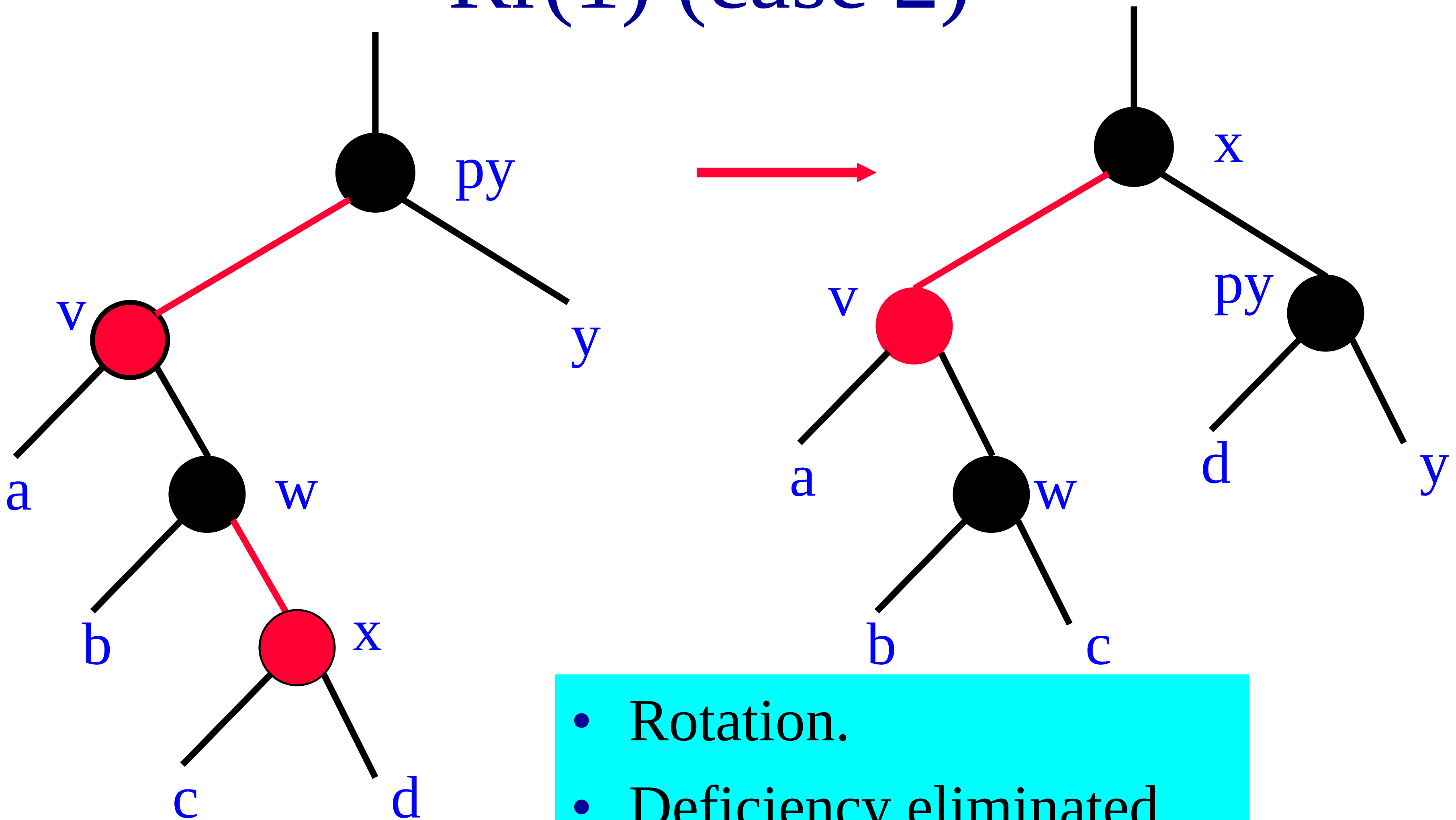
- LL rotation.
- Done!

Rr(1) (case 1)



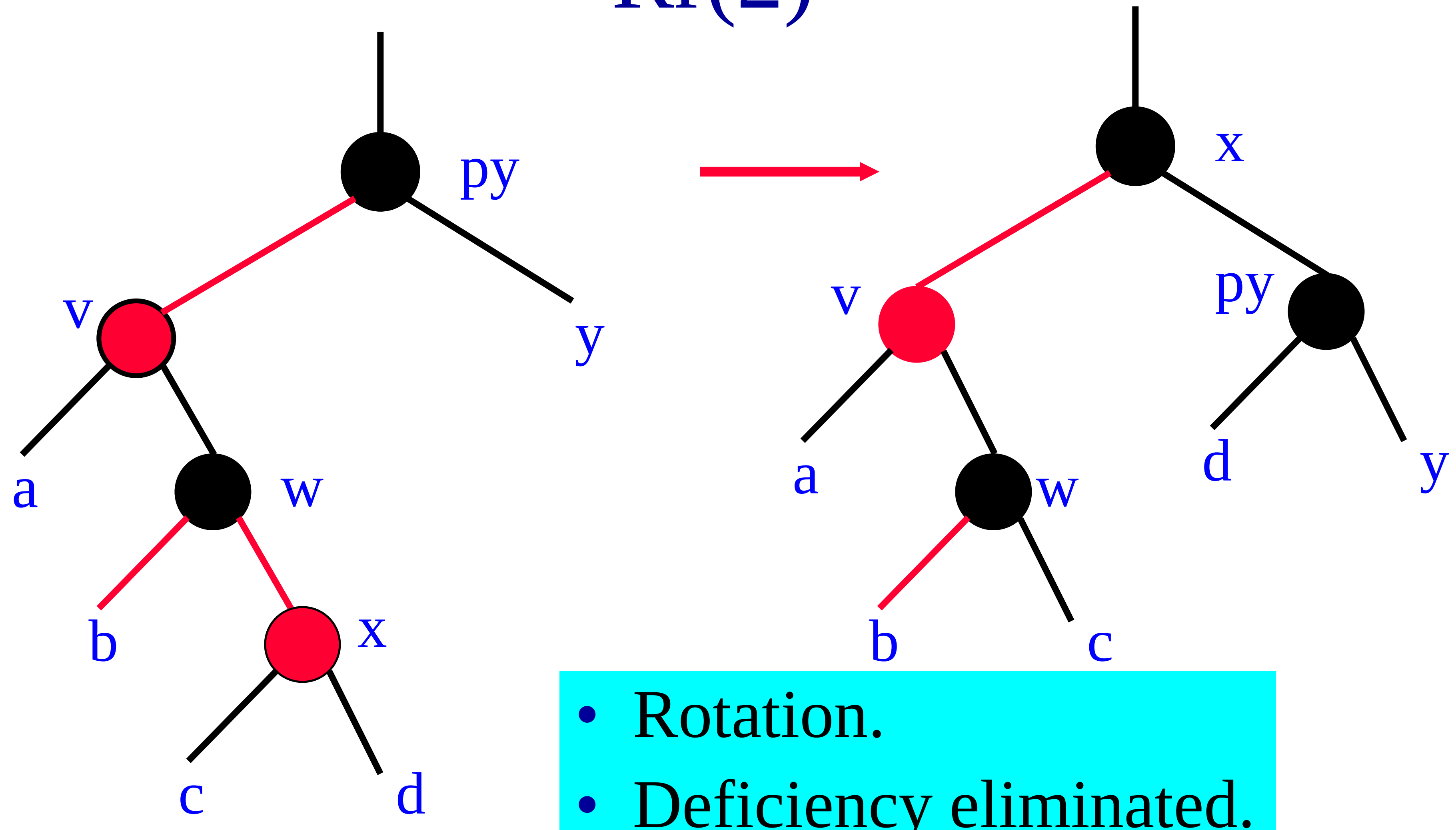
- LR rotation.
- Deficiency eliminated.
- Done!

Rr(1) (case 2)



- Rotation.
- Deficiency eliminated.
- Done!

Rr(2)

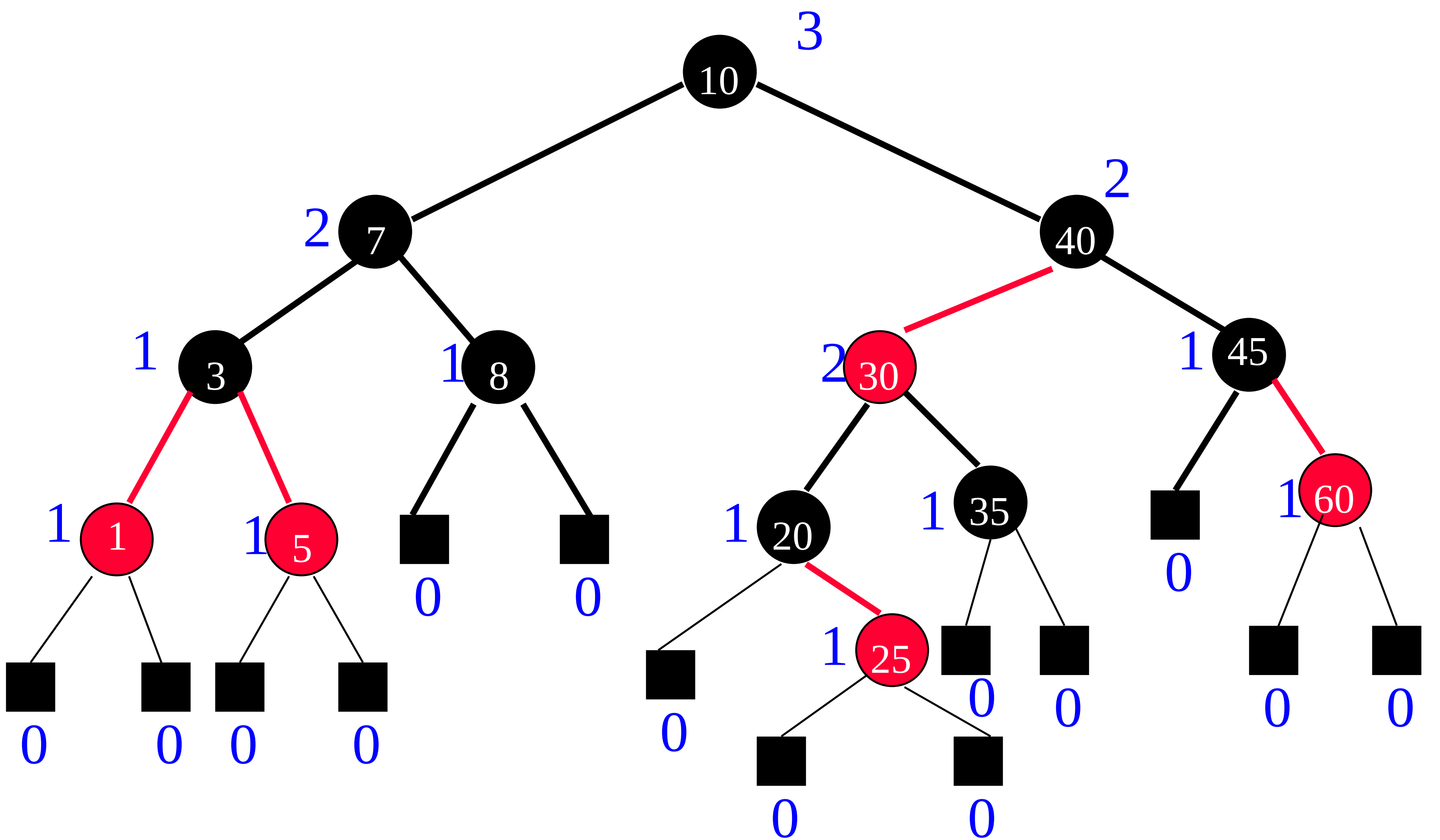


- Rotation.
- Deficiency eliminated.
- Done!

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



3
10
2
7
2
40
1
3
1
8
2
30
1
45
1
1
1
5
0
0
1
20
1
35
0
1
60
1
25
0
0
0
0
0
0
0
0
0
0
0
0
0

Properties Of rank(x)

- rank(x) = 0 for x an external node.
- rank(x) = 1 for x parent of external node.
- p(x) exists => rank(x) <= rank(p(x)) <= rank(x) + 1.
- g(x) exists => rank(x) < 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.

(* Below not covered in this year *)
Relationship Between rank() And Color

- $(p(x),x)$ is a red pointer iff $rank(x) = rank(p(x))$.
- $(p(x),x)$ is a black pointer iff $rank(x) = rank(p(x)) – 1$.
- Red node iff pointer from parent is red.
- Root is black.
- Other nodes are black iff pointer from parent is black.
- Given $rank(root)$ and node/pointer colors, remaining ranks may be computed on way down.

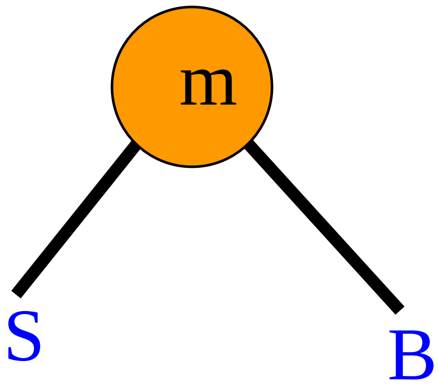
rank(root)

- Height $<= 2 * \text{rank(root)}$.
- No external nodes at levels 1, 2, …, rank(root).
  - So, #nodes $>= \Sigma_{1 <= i <= \text{rank(root)}} 2^{i-1} = 2^{\text{rank(root)}} - 1$.
  - So, $\text{rank(root)} <= \log_2(n+1)$.
- So, $\text{height(root)} <= 2\log_2(n+1)$.

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.

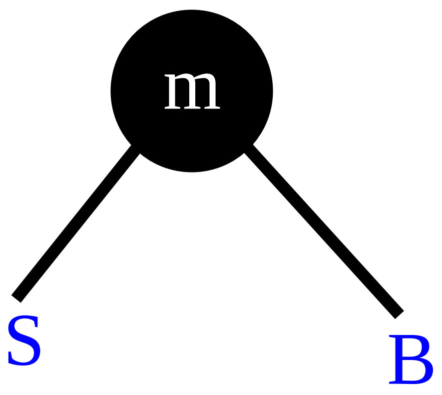
Join Binary Search Trees



- O(1) time.

Join Red-black Trees

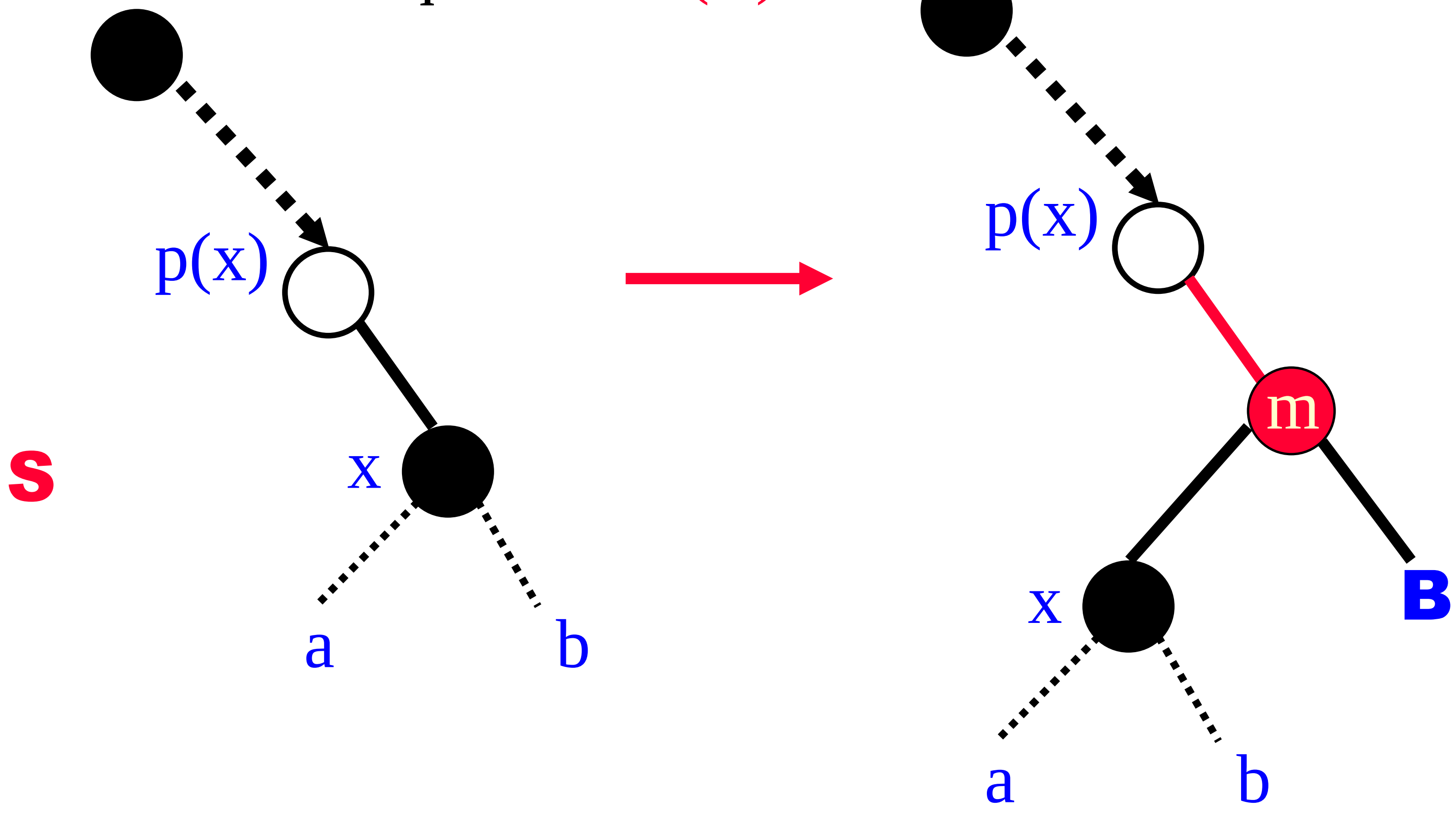
- When rank(S) = rank(B), use binary search tree method.



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

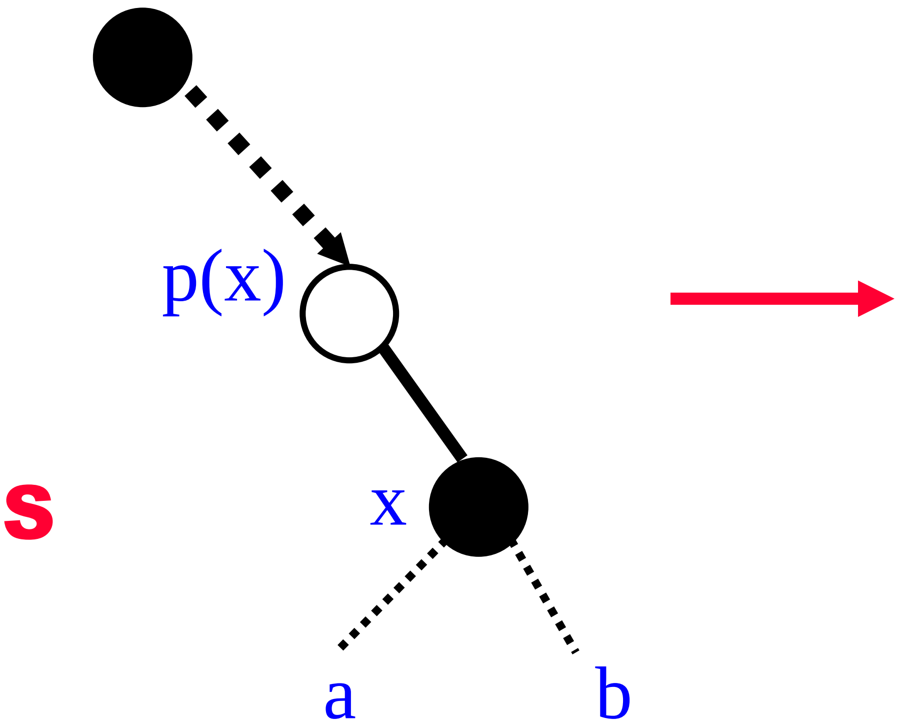
rank(S) > rank(B)

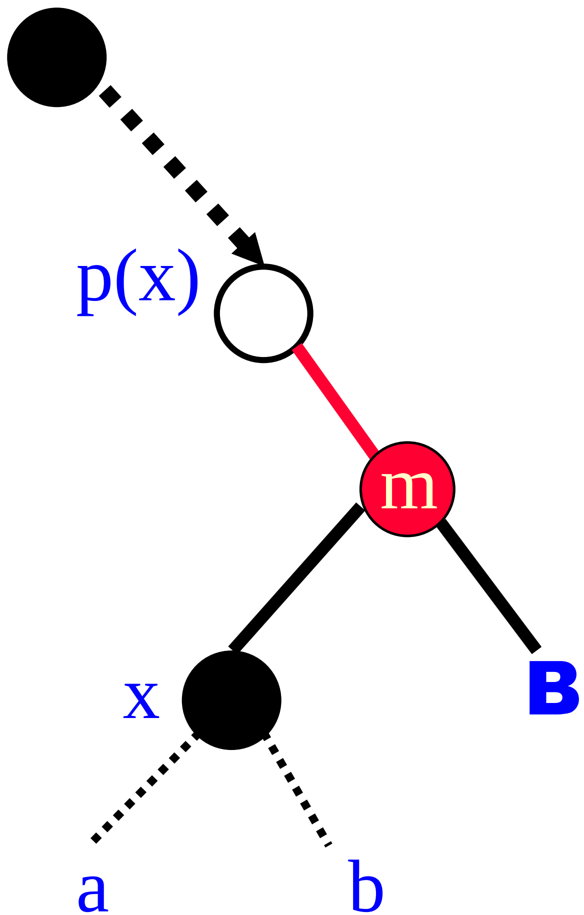
- Follow right child pointers from root of S to first node x whose rank equals rank(B).



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)).





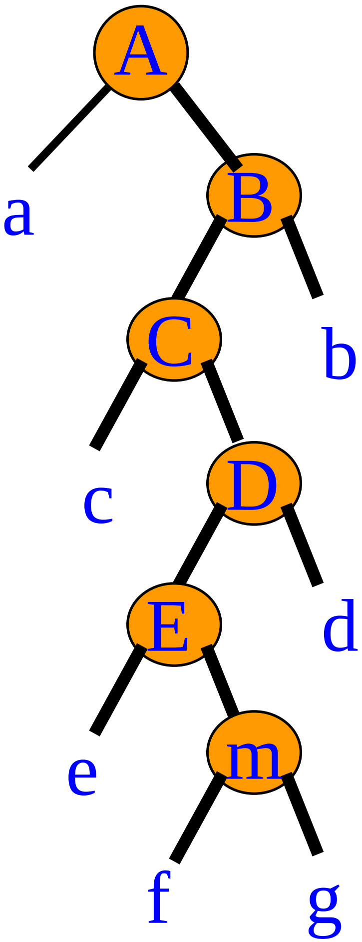
$\mathrm{rank}(S) < \mathrm{rank}(B)$

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

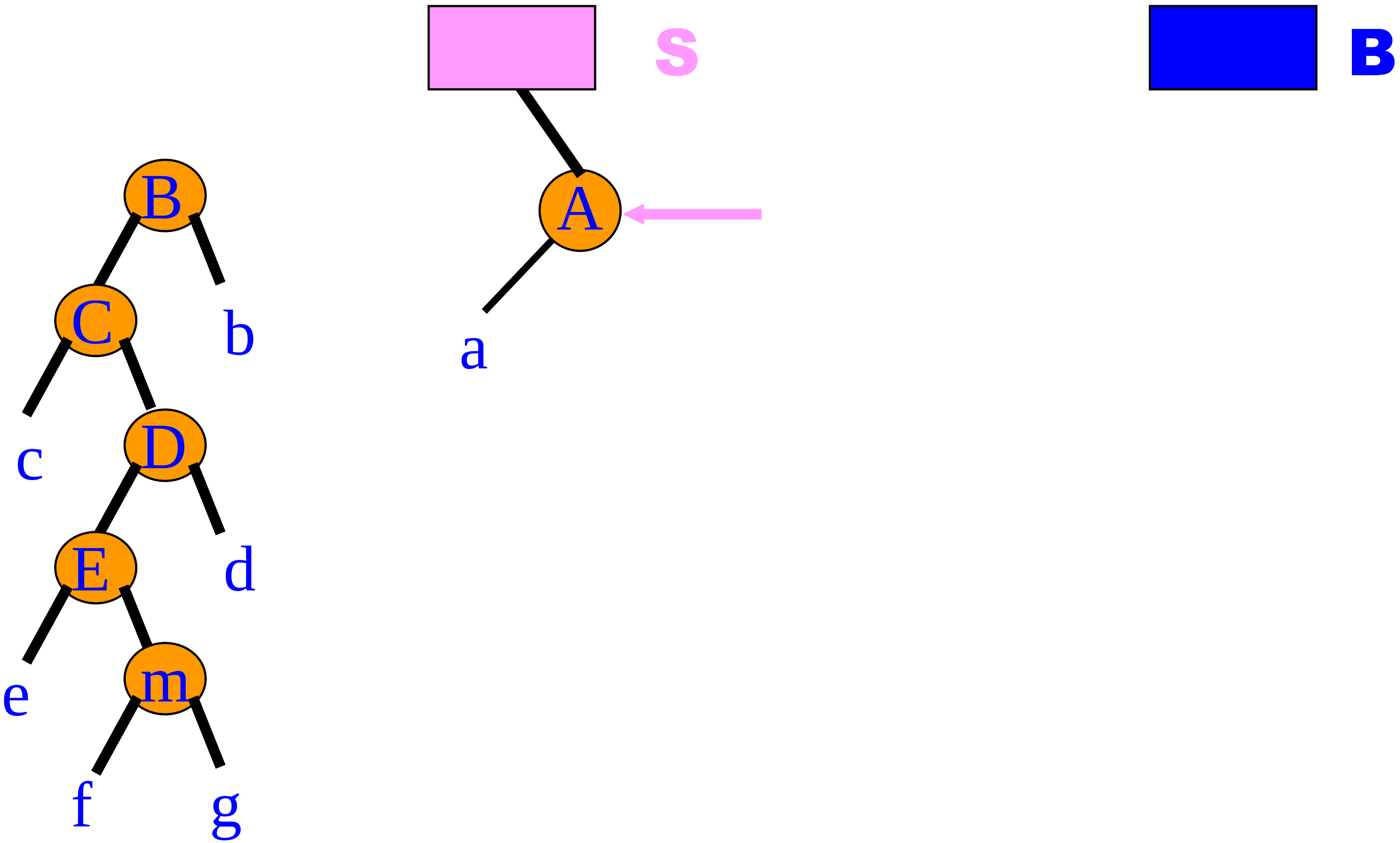
Split A Binary Search Tree



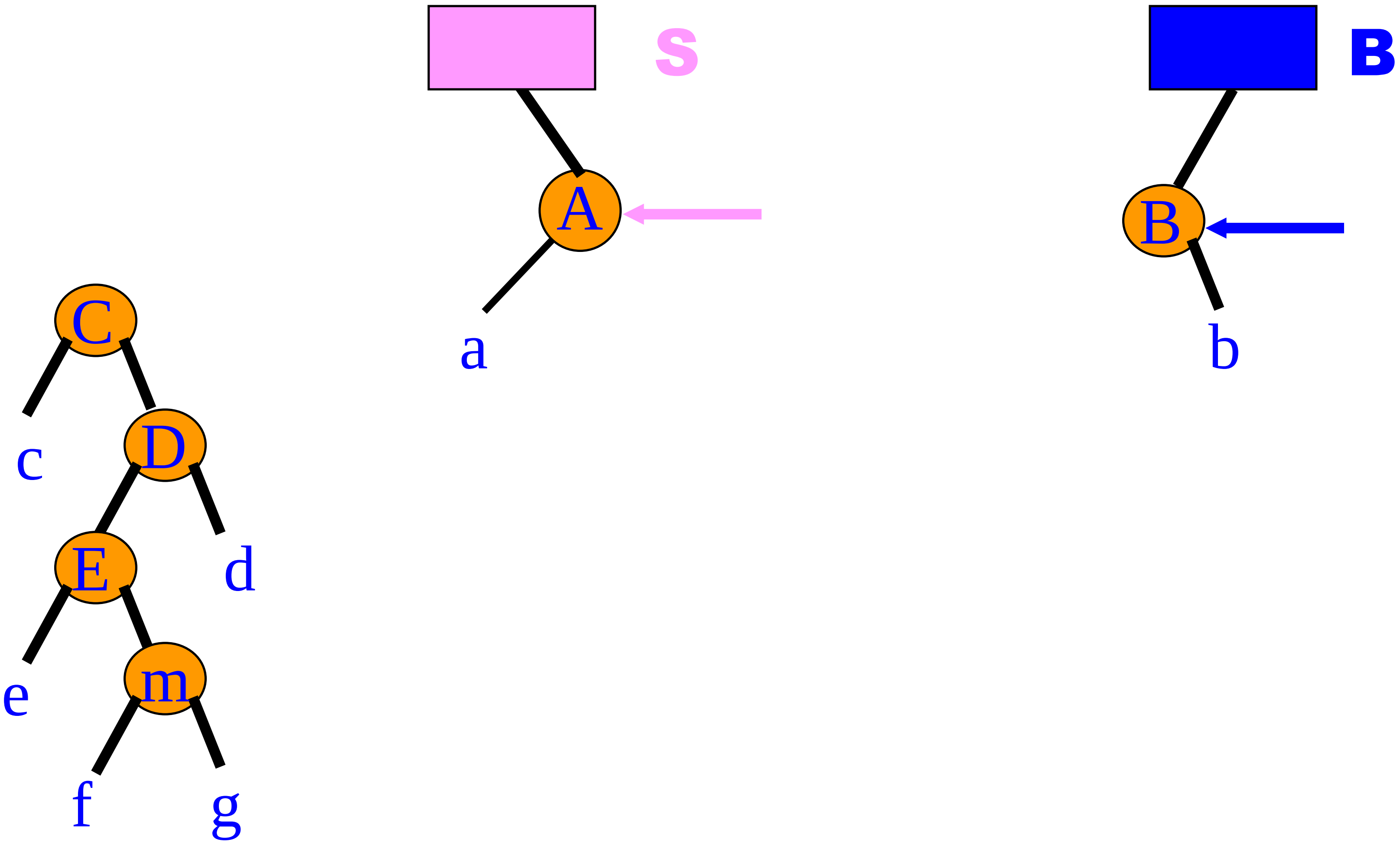
S

B

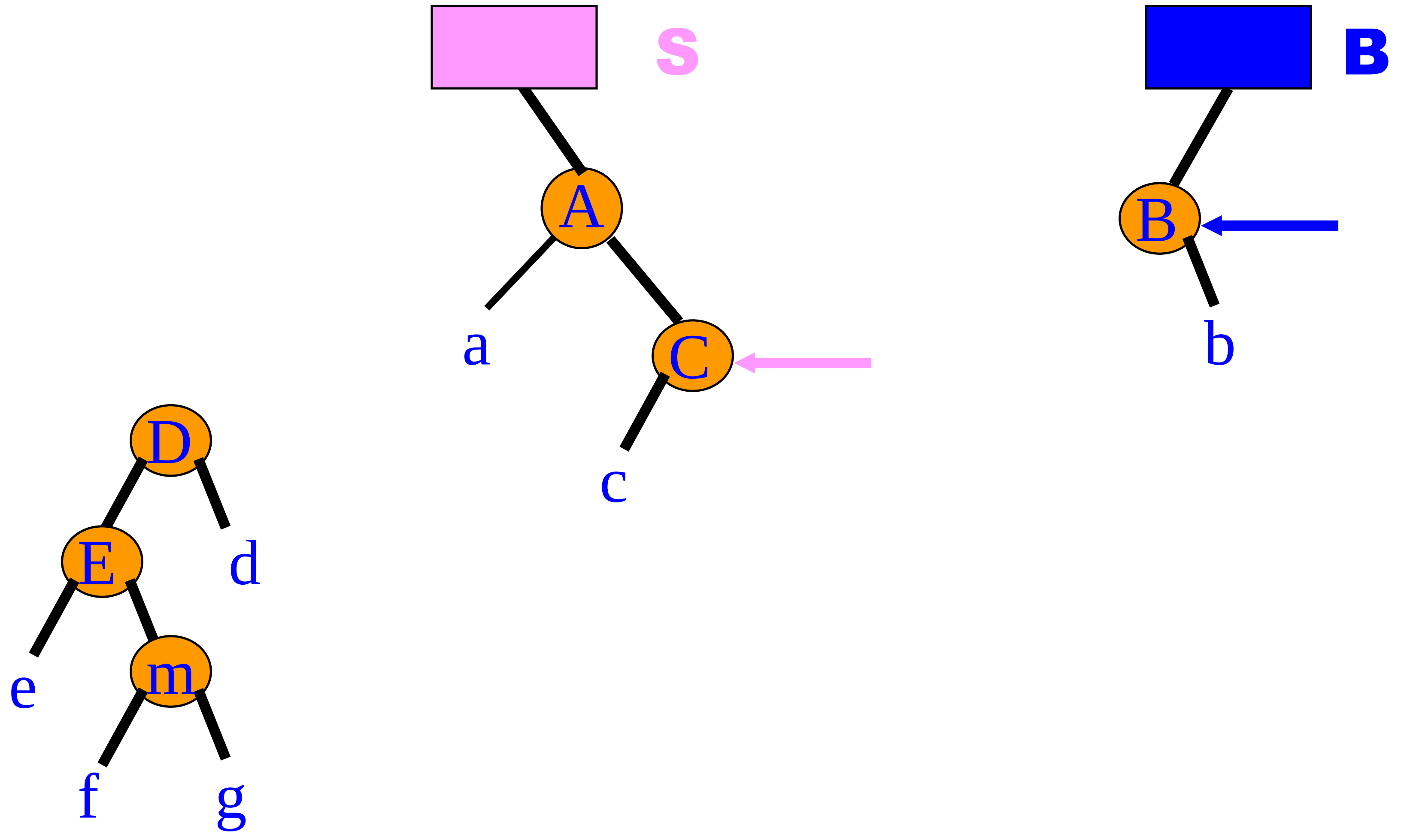
Split A Binary Search Tree



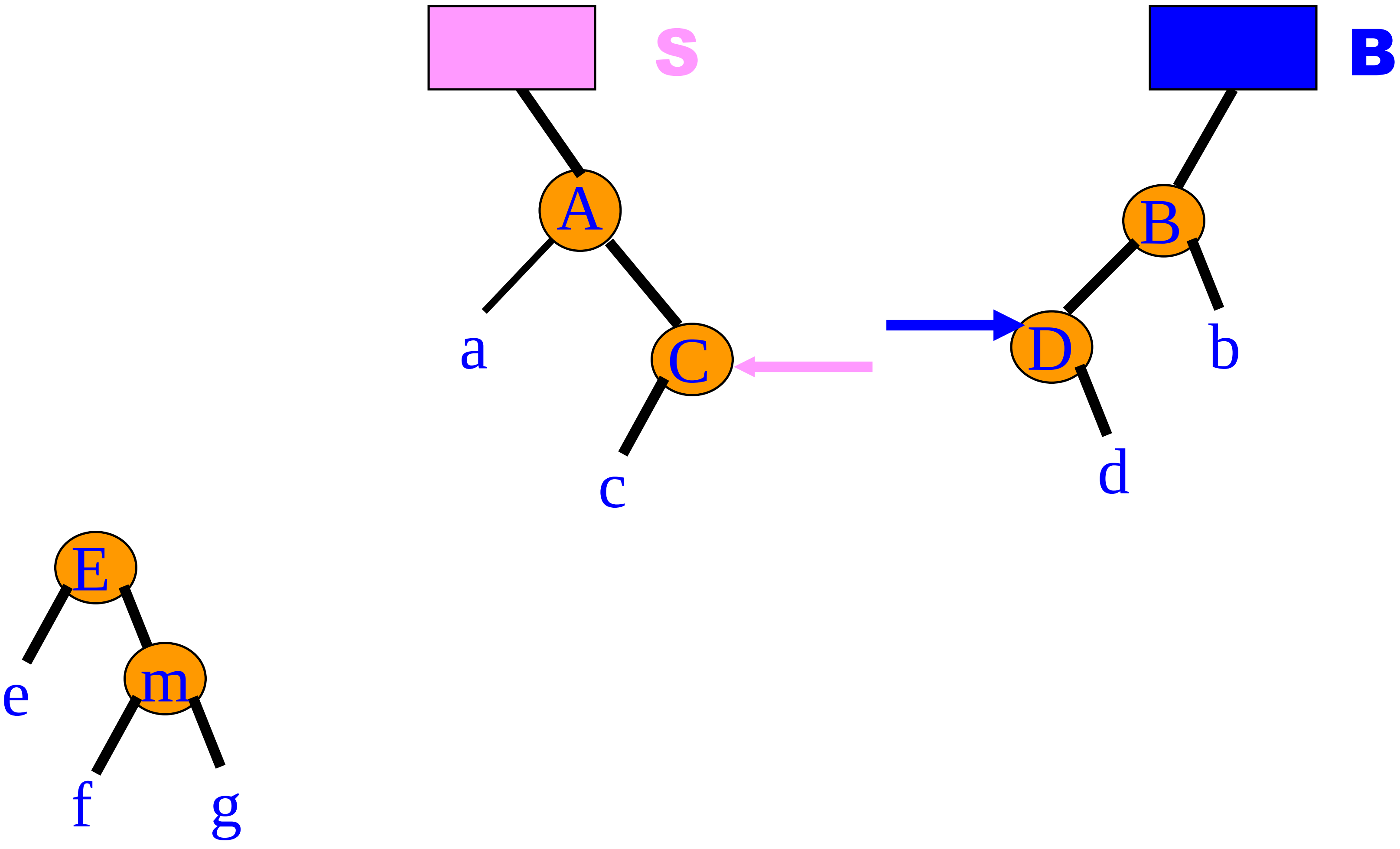
Split A Binary Search Tree



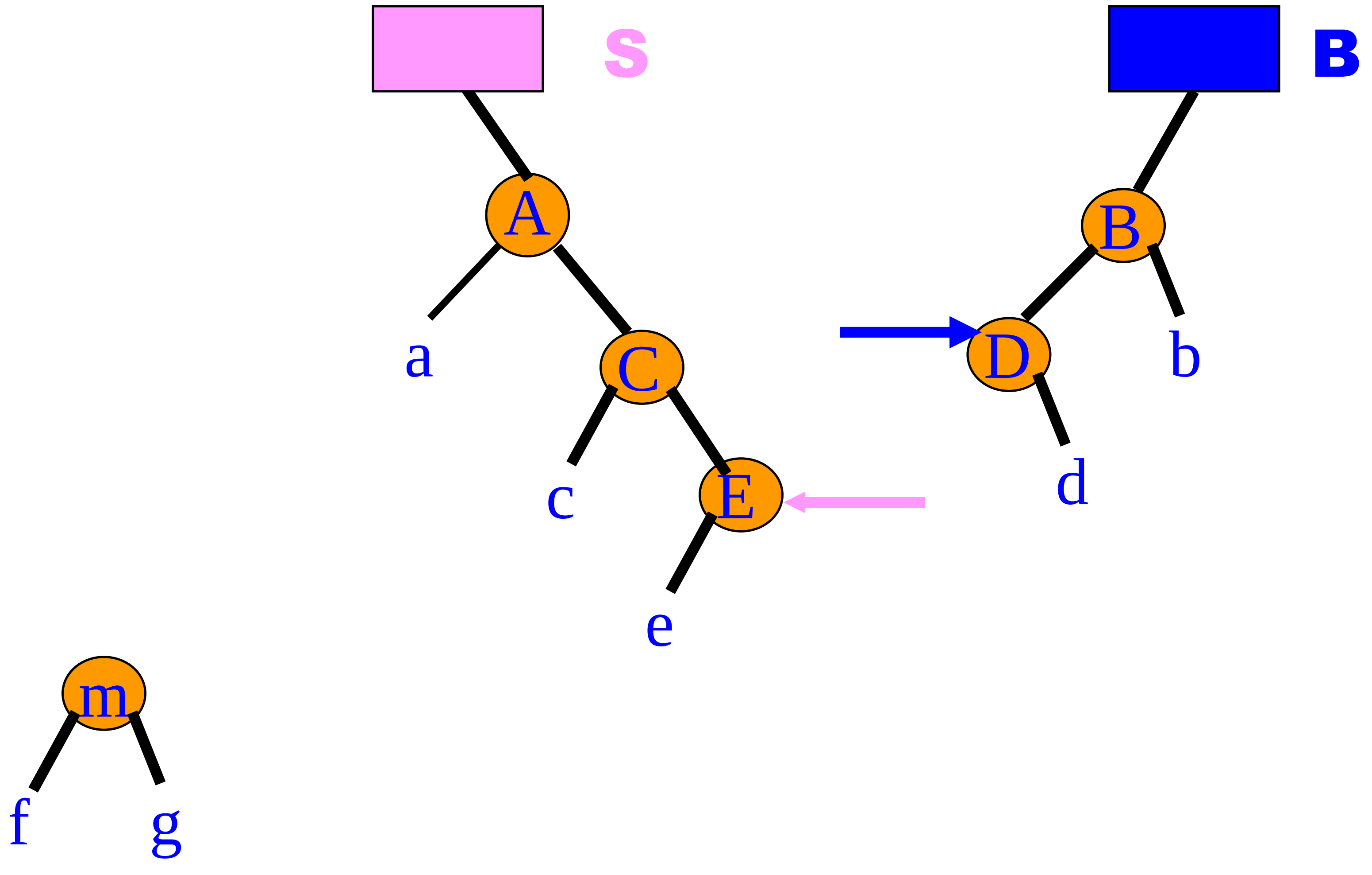
Split A Binary Search Tree



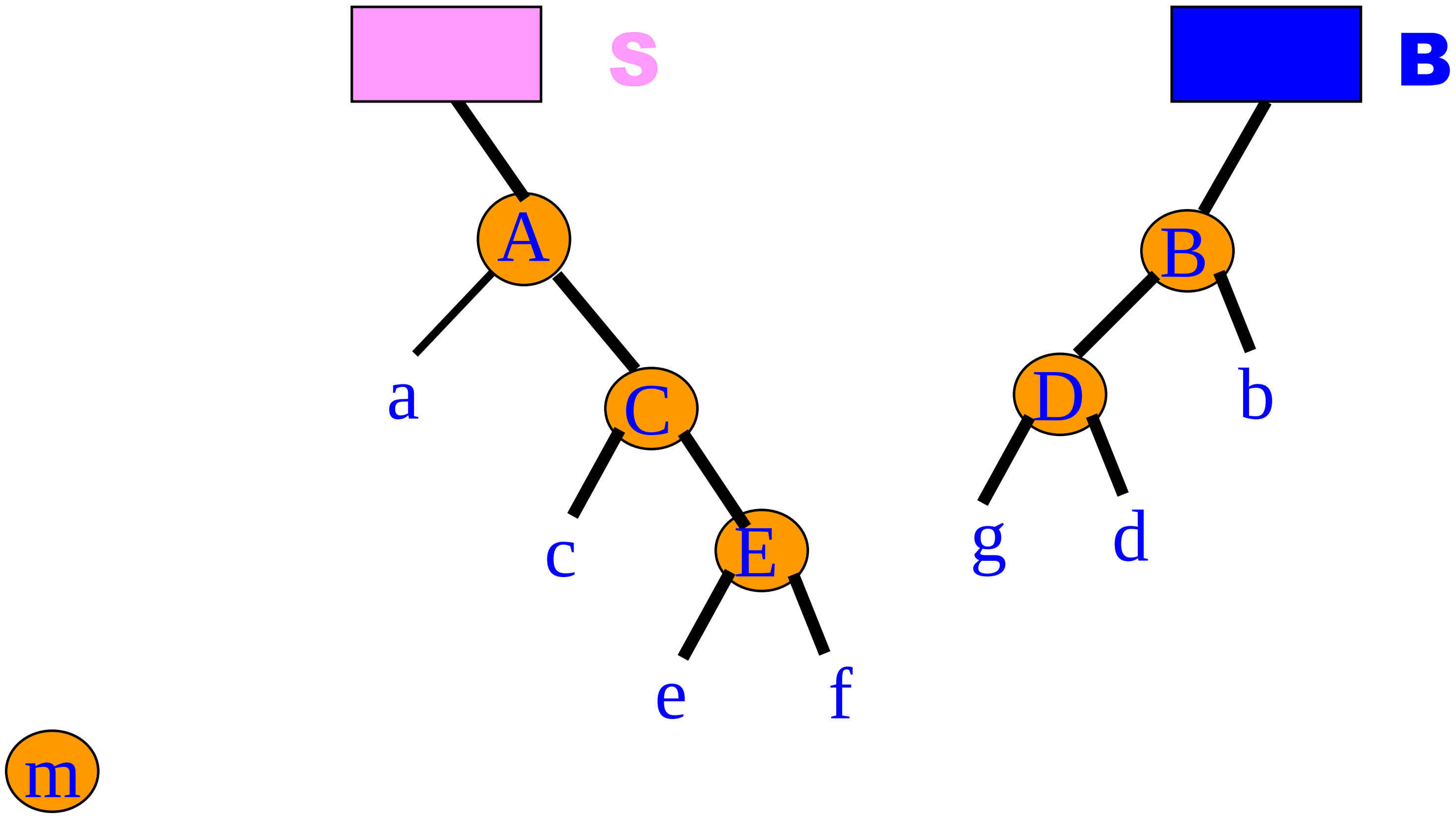
Split A Binary Search Tree



Split A Binary Search Tree



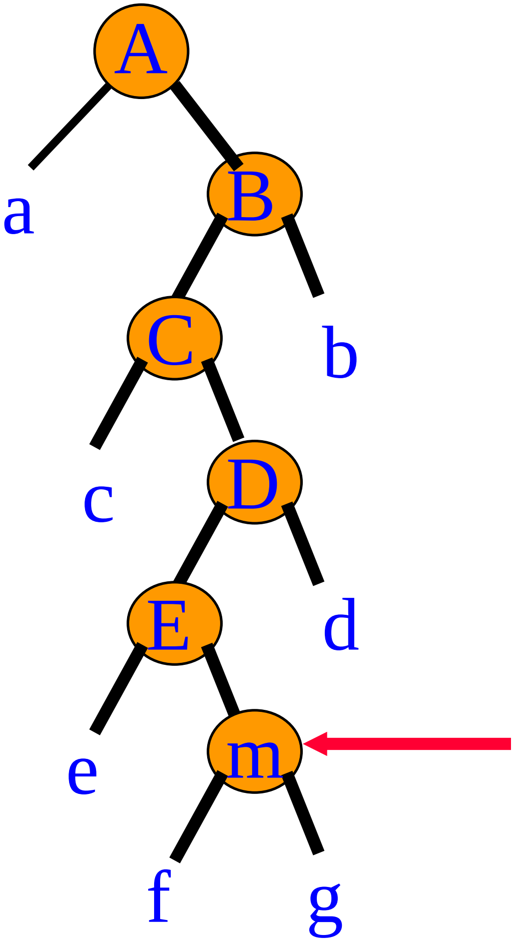
Split A Binary Search Tree



Split A Red-Black Tree

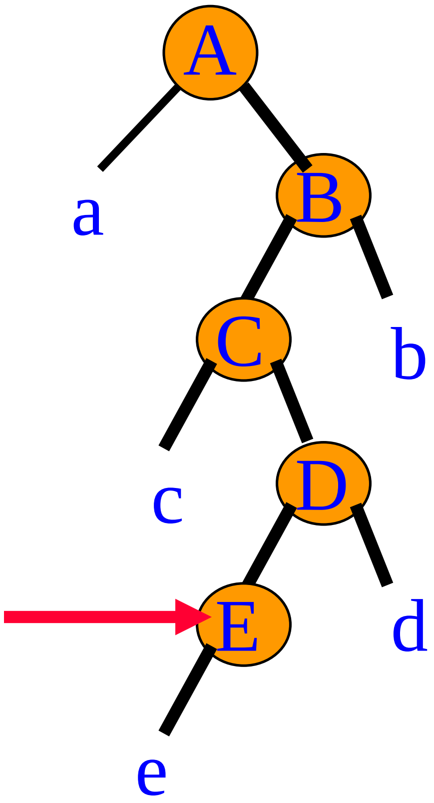
- Previous strategy does not split a red-black tree into two red-black trees.
- Must do a search for m followed by a traceback to the root.
- During the traceback use the join operation to construct S and B.

Split A Red-Black Tree



S = f **B =** g

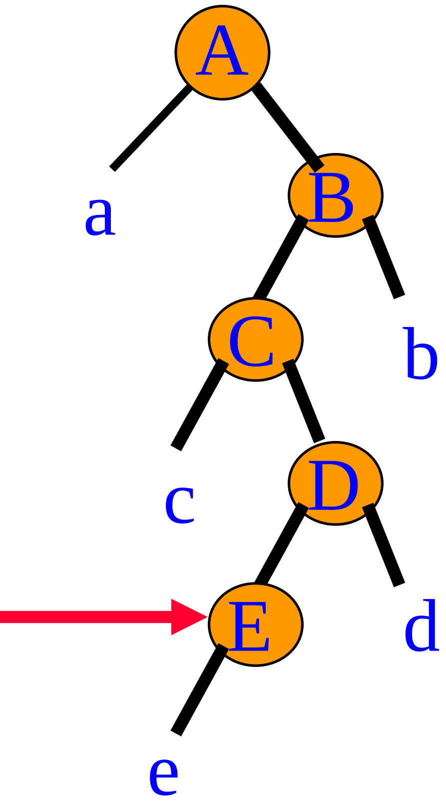
Split A Red-Black Tree



S = f

B = g

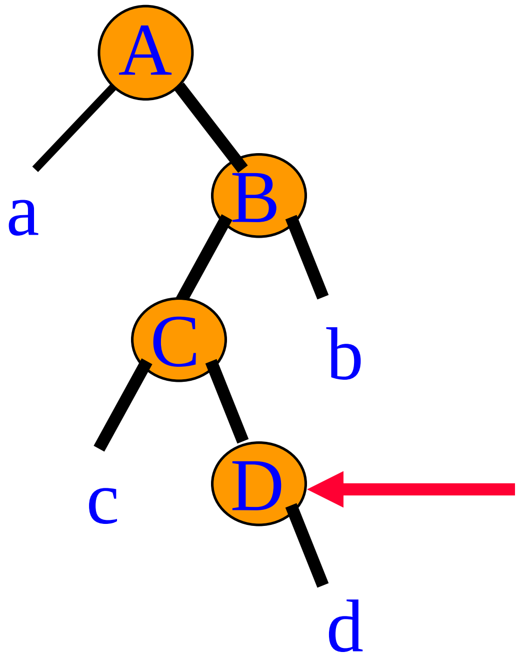
Split A Red-Black Tree



S = f　　　　**B** = g

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

Split A Red-Black Tree

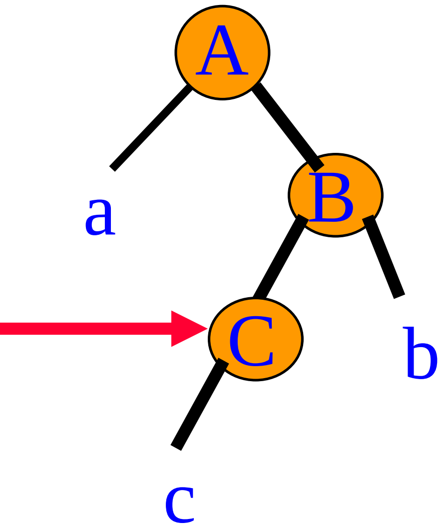


S = f **B** = g

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

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

Split A Red-Black Tree



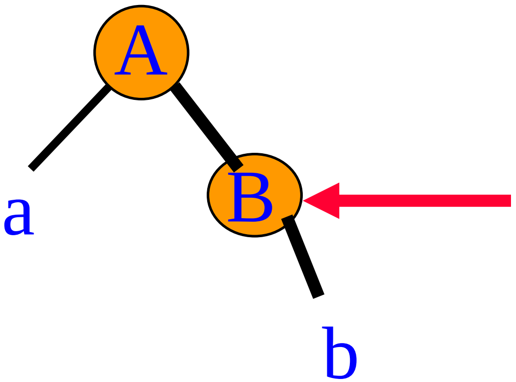
S = f **B** = g

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

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

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

Split A Red-Black Tree



S = f **B** = g

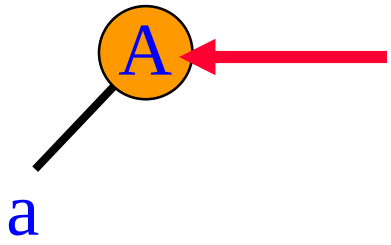
S = join(e, E, **S**)

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**)

Complexity Of Split

- O(log n)
- See text.

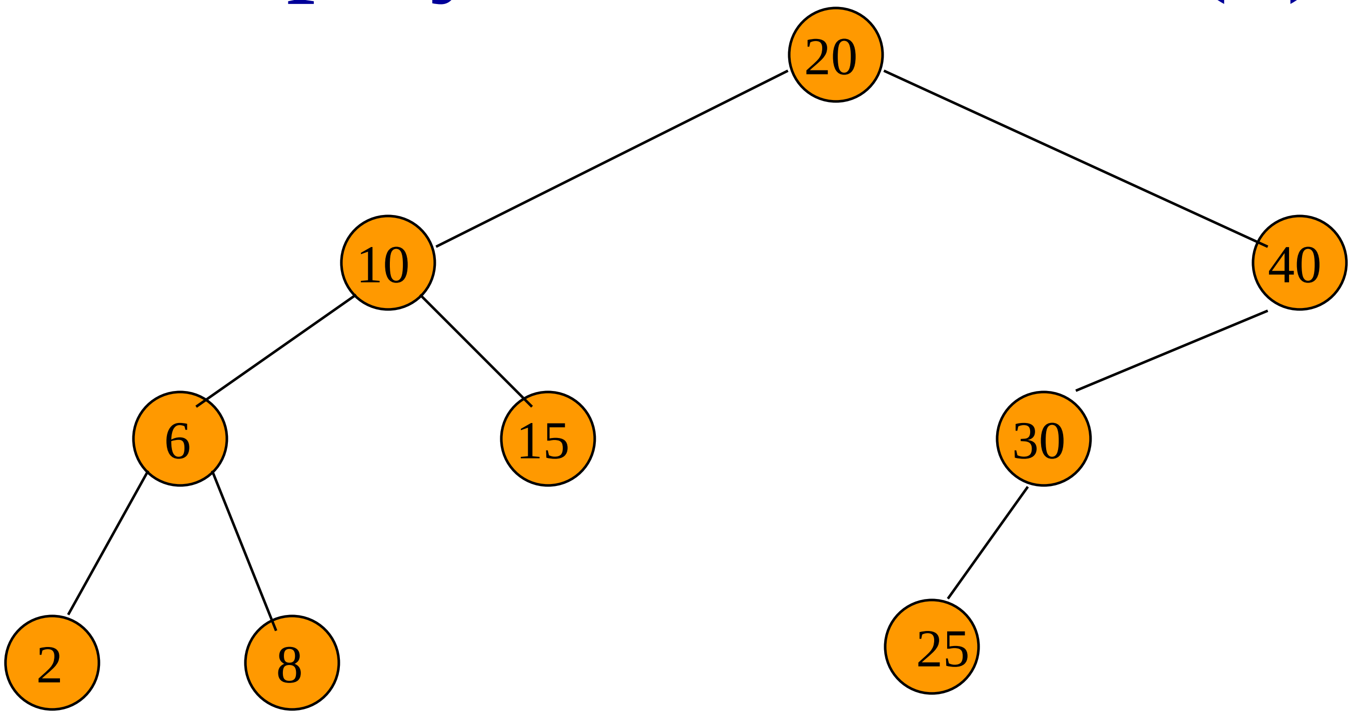
(** Splay Trees not covered in class**)

- Binary search trees.
- Search, insert, delete, and split have amortized complexity O(log n) & actual complexity O(n).
- Actual and amortized complexity of join is O(1).
- Priority queue and double-ended priority queue versions outperform heaps, deaps, etc. over a sequence of operations.
- Two varieties.
  - Bottom up.
  - Top down.

Bottom-Up Splay Trees

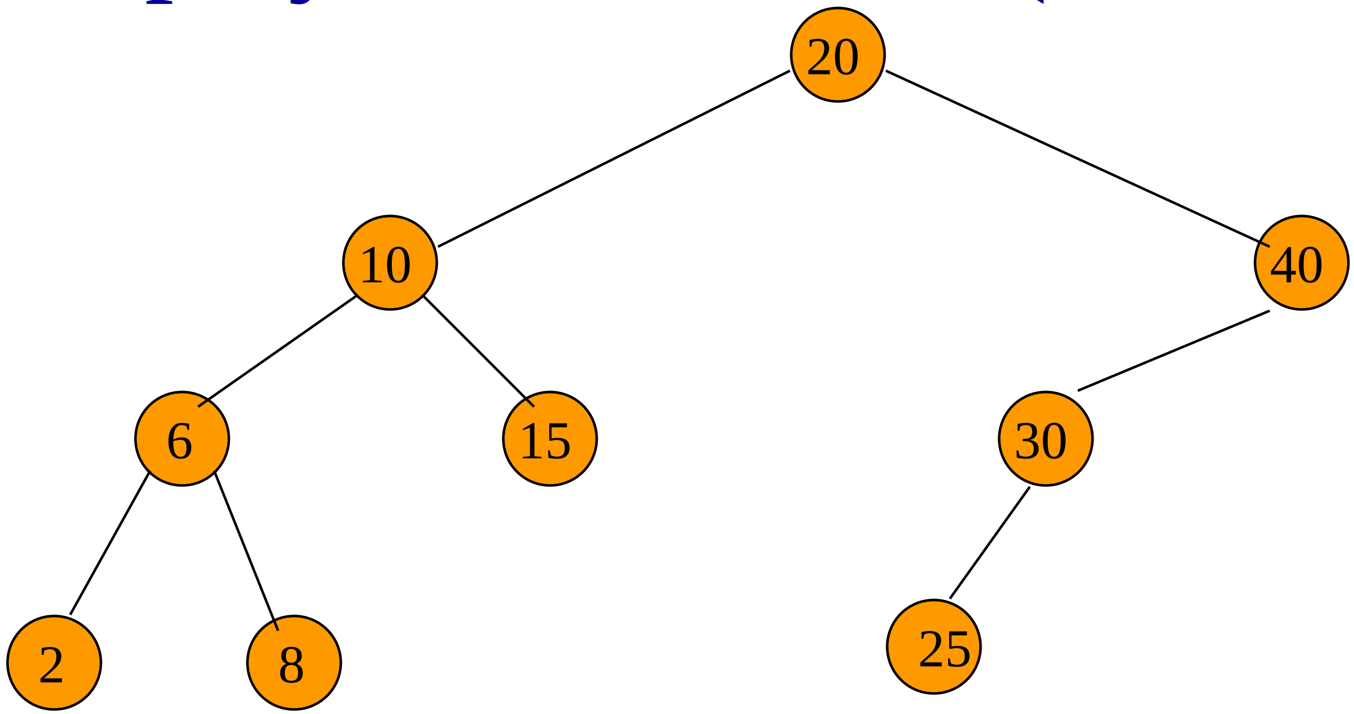
- Search, insert, delete, and join are done as in an unbalanced binary search tree.
- Search, insert, and delete are followed by a splay operation that begins at a splay node.
- When the splay operation completes, the splay node has become the tree root.
- Join requires no splay (or, a null splay is done).
- For the split operation, the splay is done in the middle (rather than end) of the operation.

Splay Node – search(k)



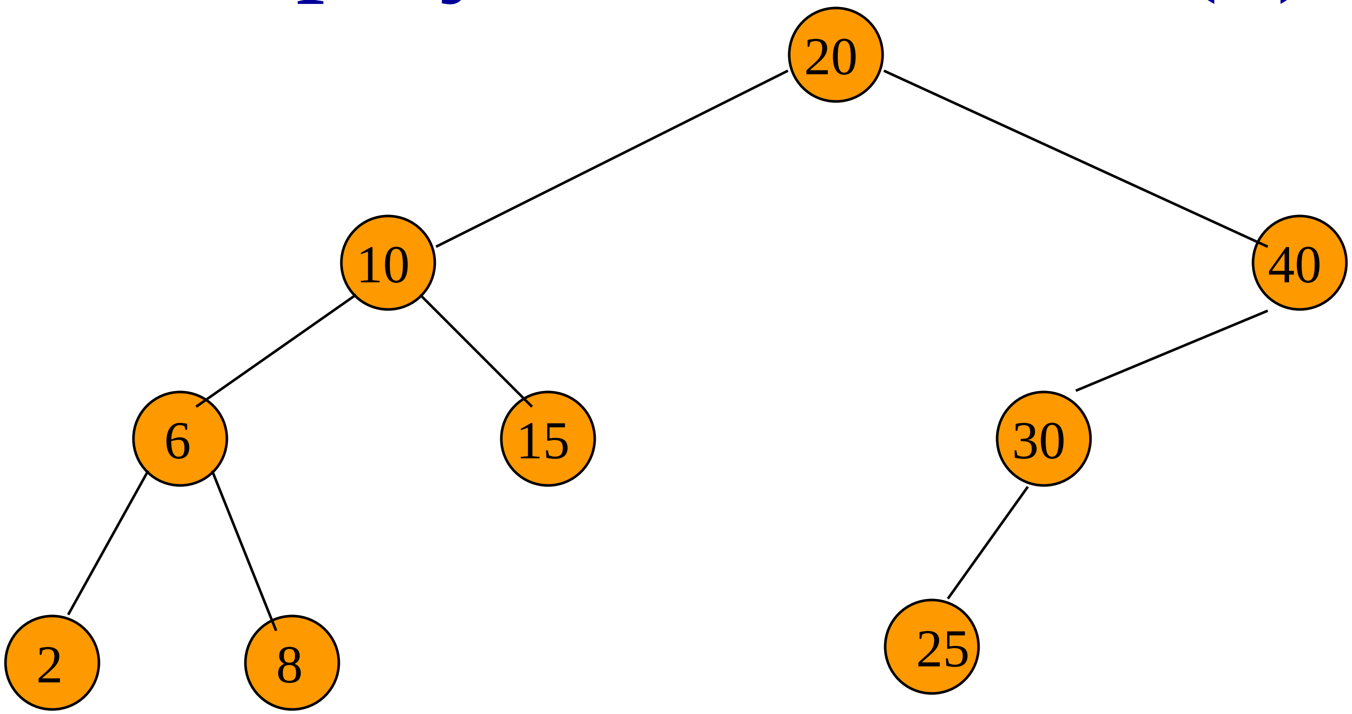
- If there is a pair whose key is k, the node containing this pair is the splay node.
- Otherwise, the parent of the external node where the search terminates is the splay node.

Splay Node – insert(newPair)



- If there is already a pair whose key is newPair.key, the node containing this pair is the splay node.
- Otherwise, the newly inserted node is the splay node.

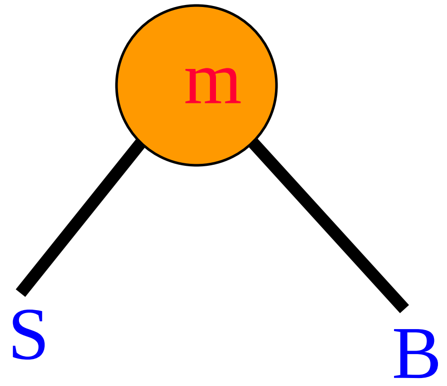
Splay Node – delete(k)



- If there is a pair whose key is k, the parent of the node that is physically deleted from the tree is the splay node.
- Otherwise, the parent of the external node where the search terminates is the splay node.

Splay Node – split(k)

- Use the unbalanced binary search tree insert algorithm to insert a new pair whose key is k.
- The splay node is as for the splay tree insert algorithm.
- Following the splay, the left subtree of the root is S, and the right subtree is B.



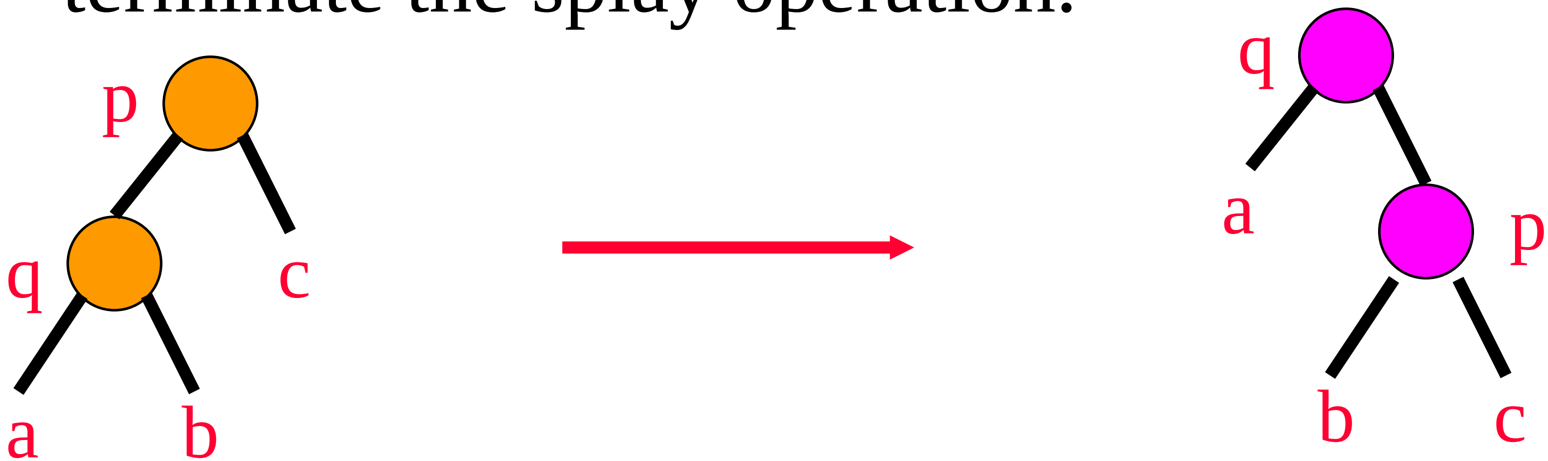
- m is set to null if it is the newly inserted pair.

Splay

- Let q be the splay node.
- q is moved up the tree using a series of splay steps.
- In a splay step, the node q moves up the tree by 0, 1, or 2 levels.
- Every splay step, except possibly the last one, moves q two levels up.

Splay Step

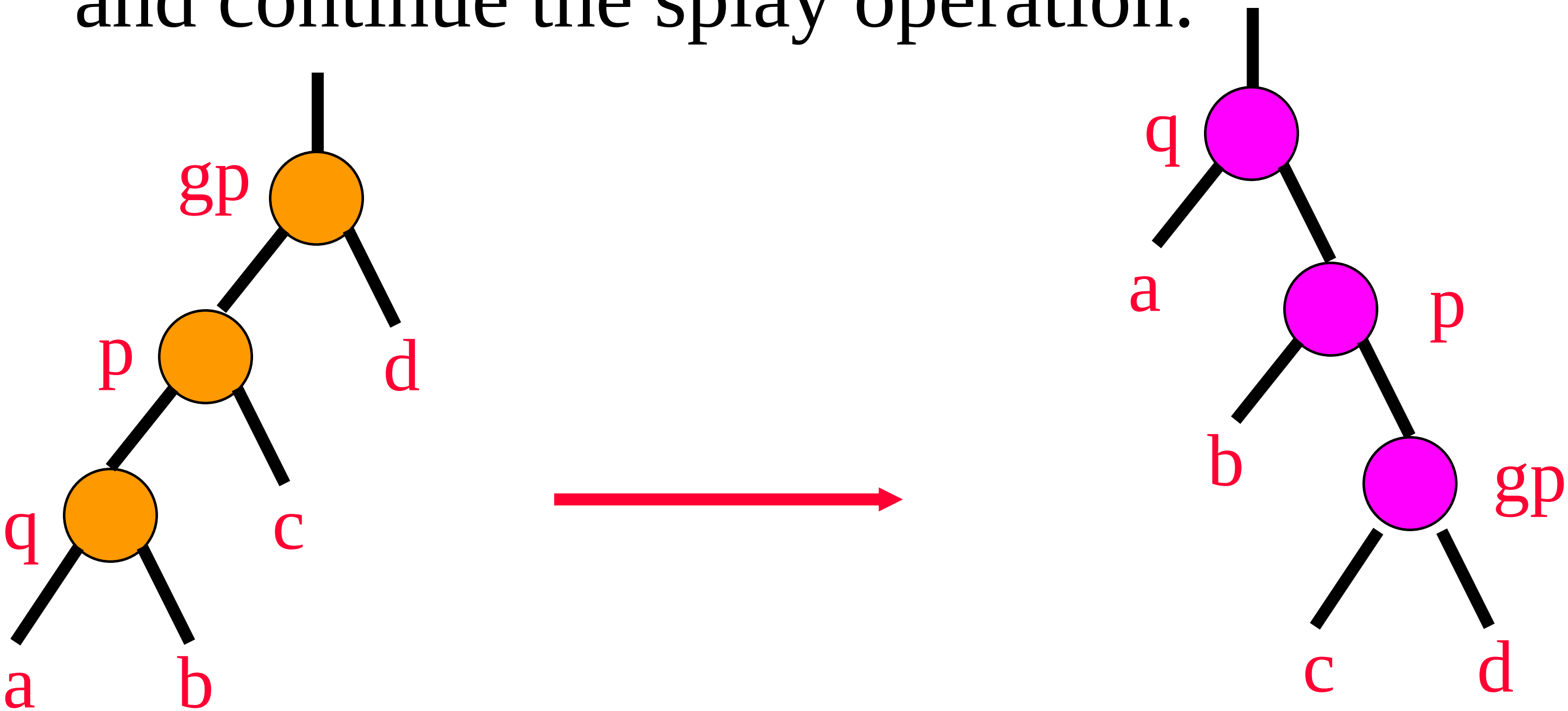
- If q = null or q is the root, do nothing (splay is over).
- If q is at level 2, do a one-level move and terminate the splay operation.



- q right child of p is symmetric.

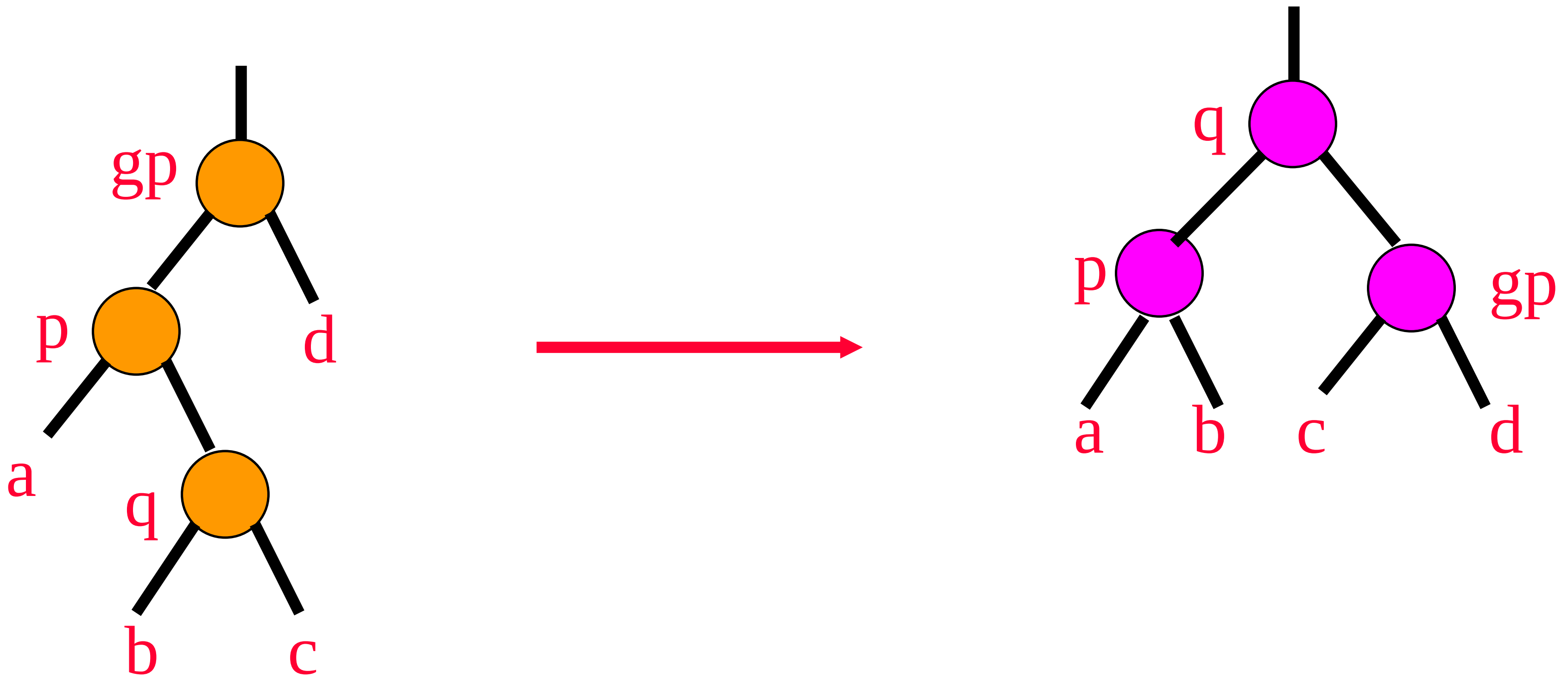
Splay Step

- If q is at a level > 2, do a two-level move and continue the splay operation.



- q right child of right child of gp is symmetric.

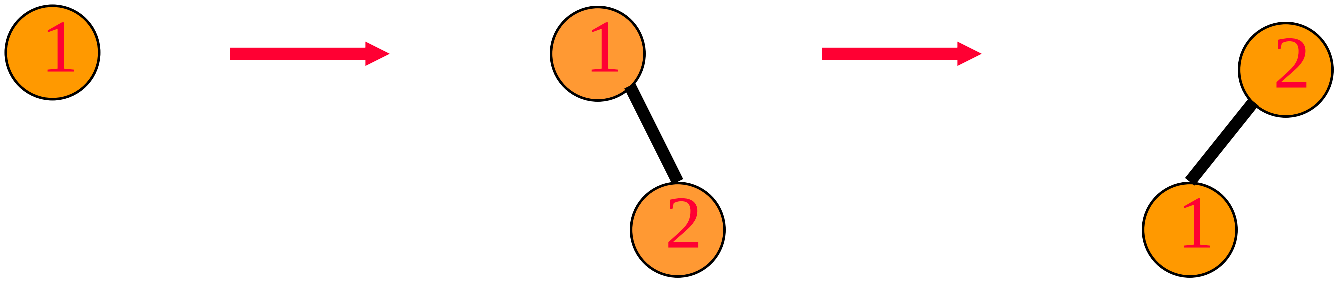
2-Level Move (case 2)



- q left child of right child of gp is symmetric.

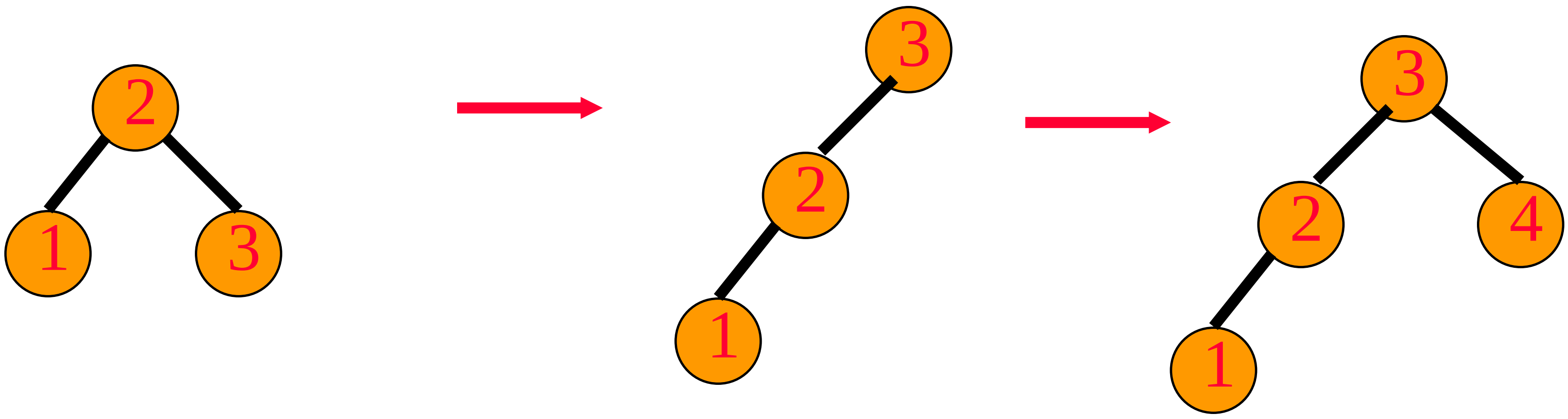
Per Operation Actual Complexity

- Start with an empty splay tree and insert pairs with keys 1, 2, 3, …, in this order.



Per Operation Actual Complexity

- Start with an empty splay tree and insert pairs with keys 1, 2, 3, …, in this order.



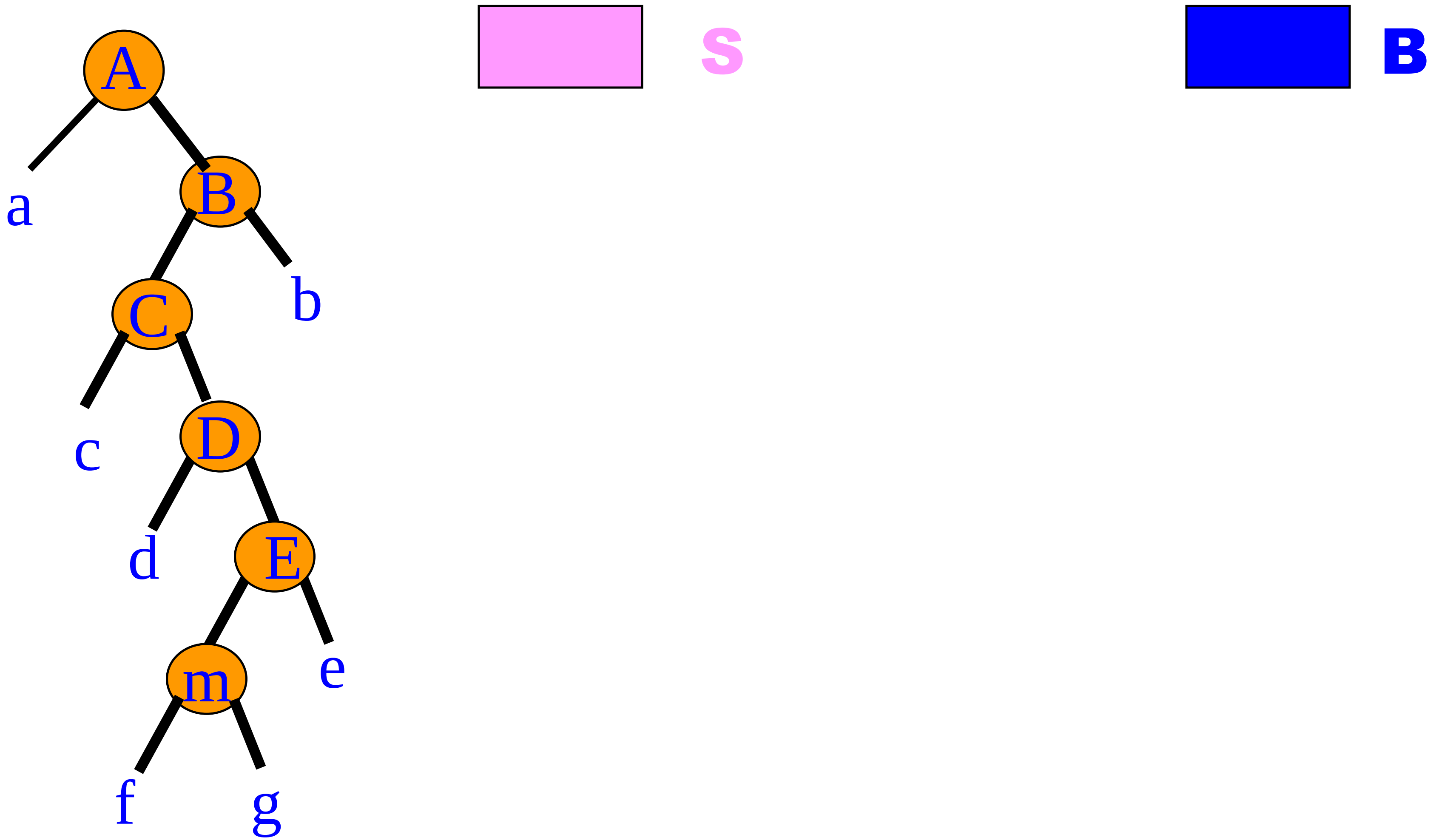
Per Operation Actual Complexity

- Worst-case height = n.
- Actual complexity of search, insert, delete, and split is O(n).

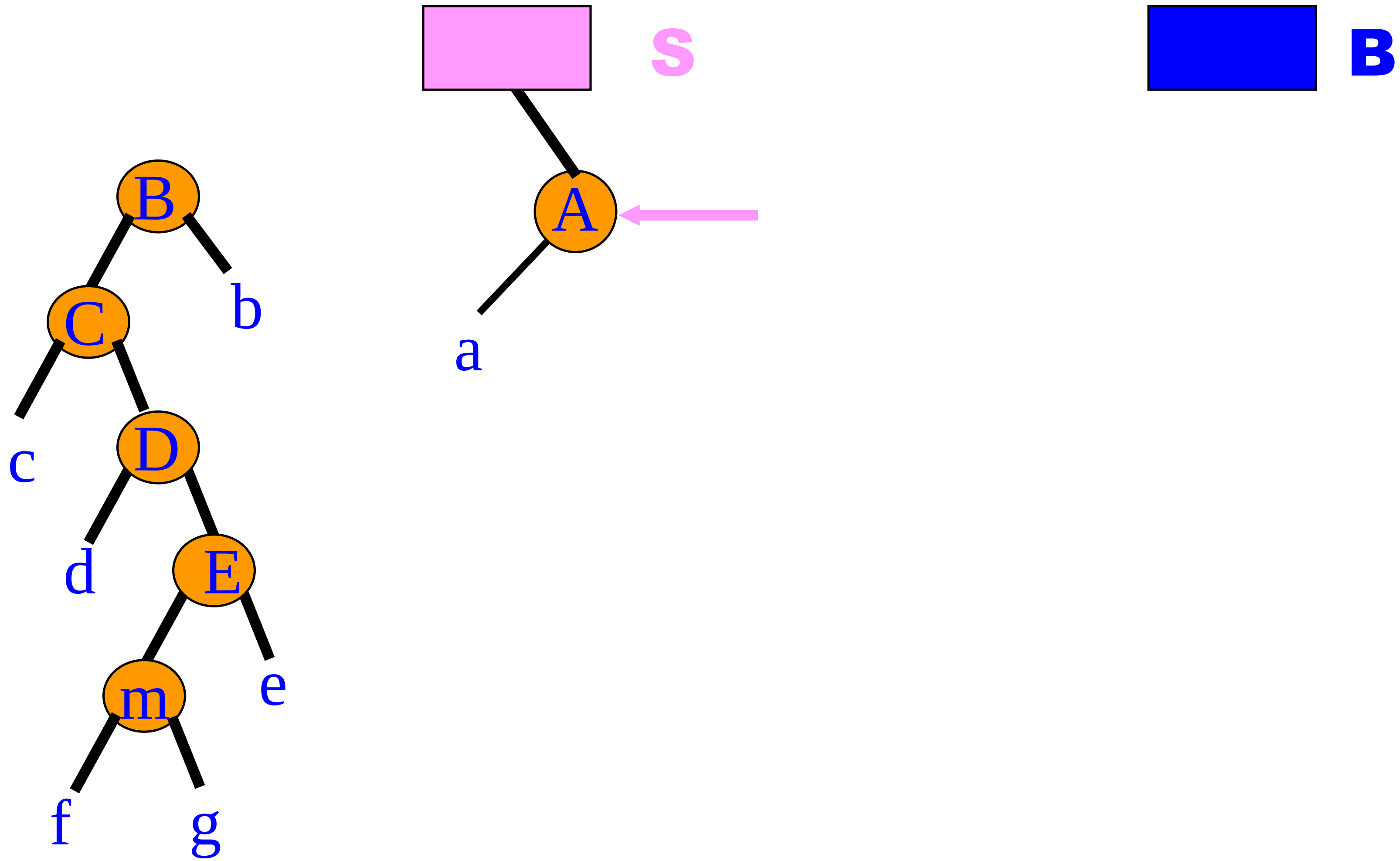
Top-Down Splay Trees

- On the way down the tree, split the tree into the binary search trees **S** (small elements) and **B** (big elements).
  - Similar to split operation in an unbalanced binary search tree.
  - However, a rotation is done whenever an LL or RR move is made.
  - Move down 2 levels at a time, except (possibly) in the end when a one level move is made.
- When the splay node is reached, **S**, **B**, and the subtree rooted at the splay node are combined into a single binary search tree.

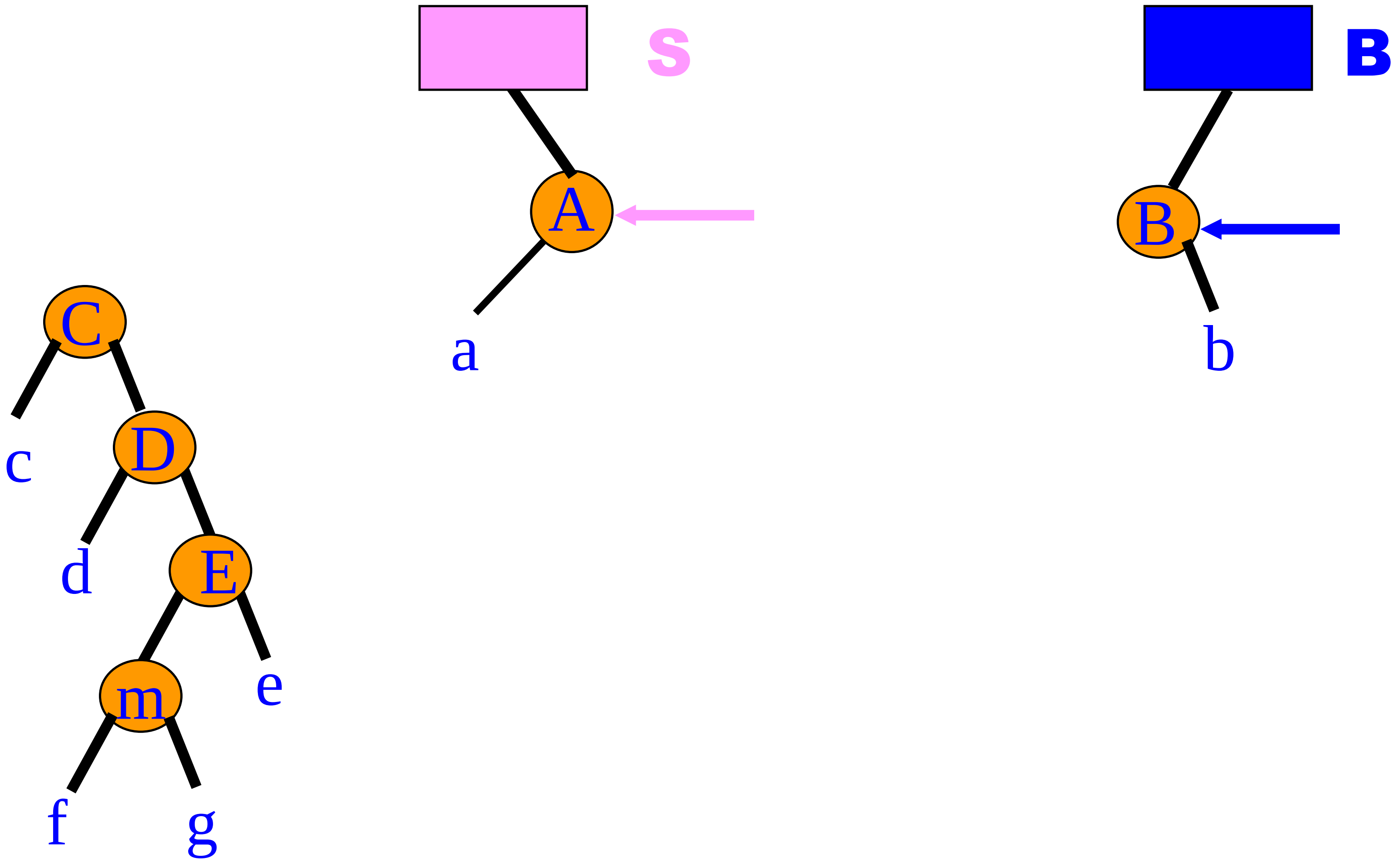
Split A Binary Search Tree



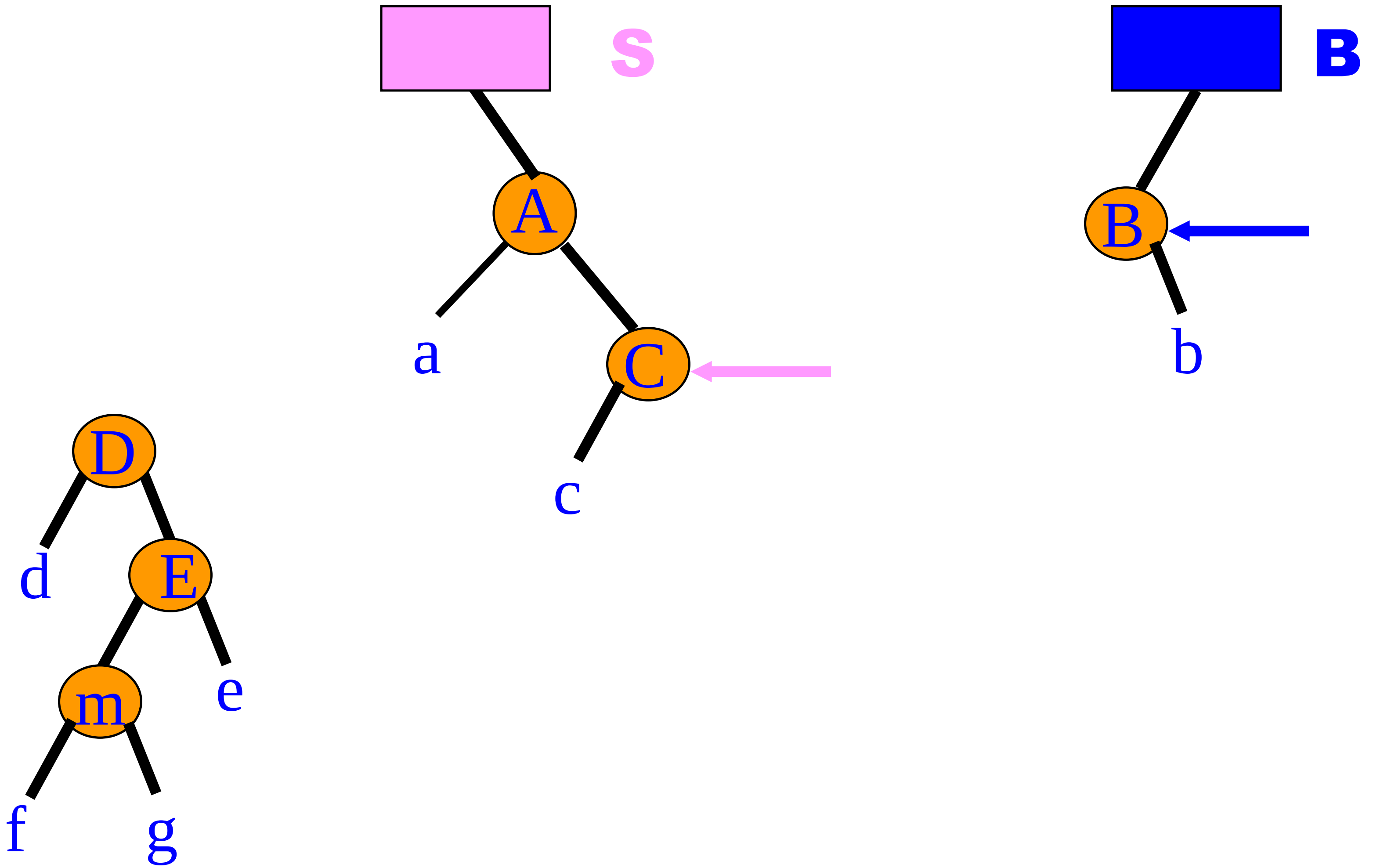
Split A Binary Search Tree



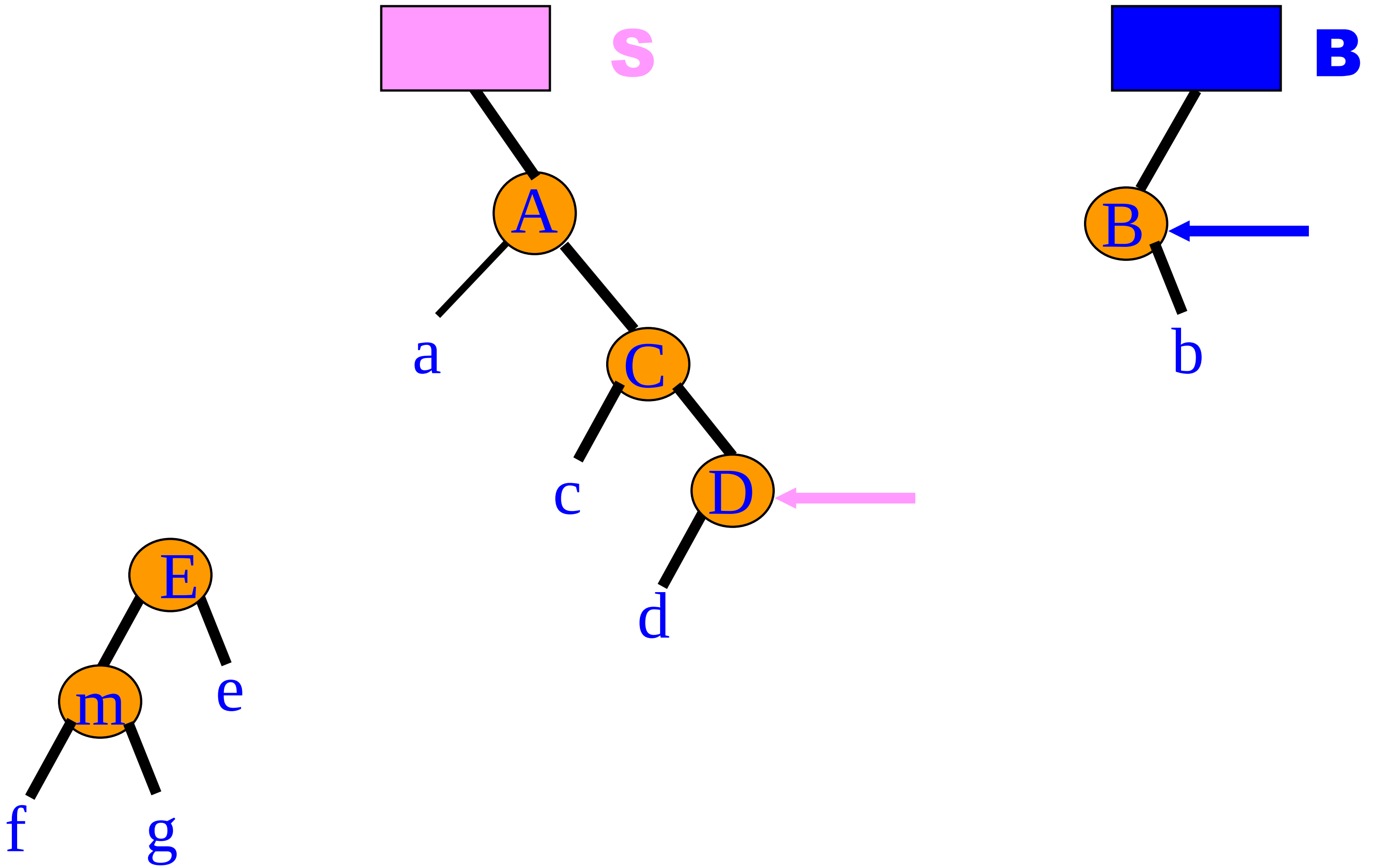
Split A Binary Search Tree



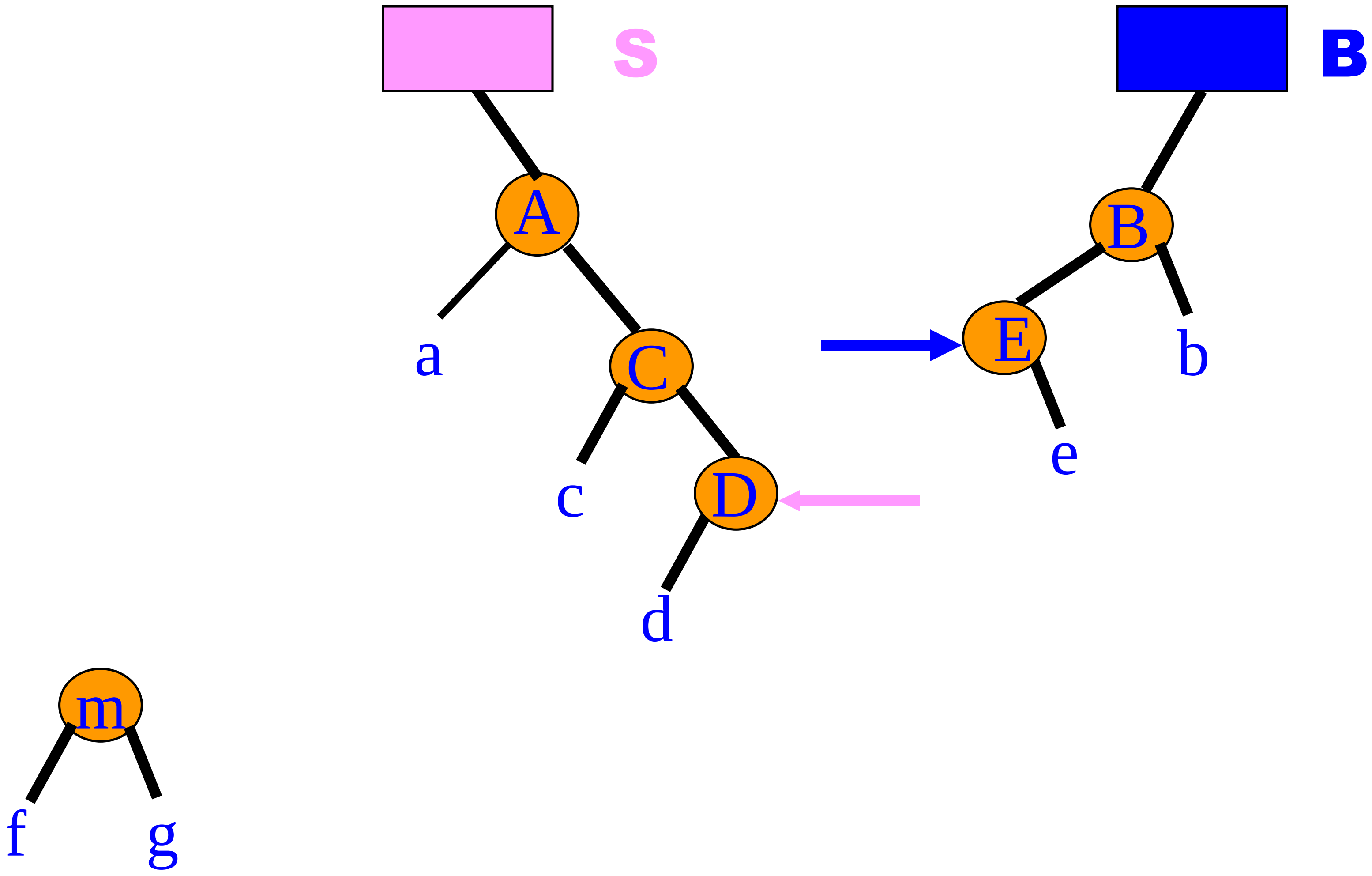
Split A Binary Search Tree



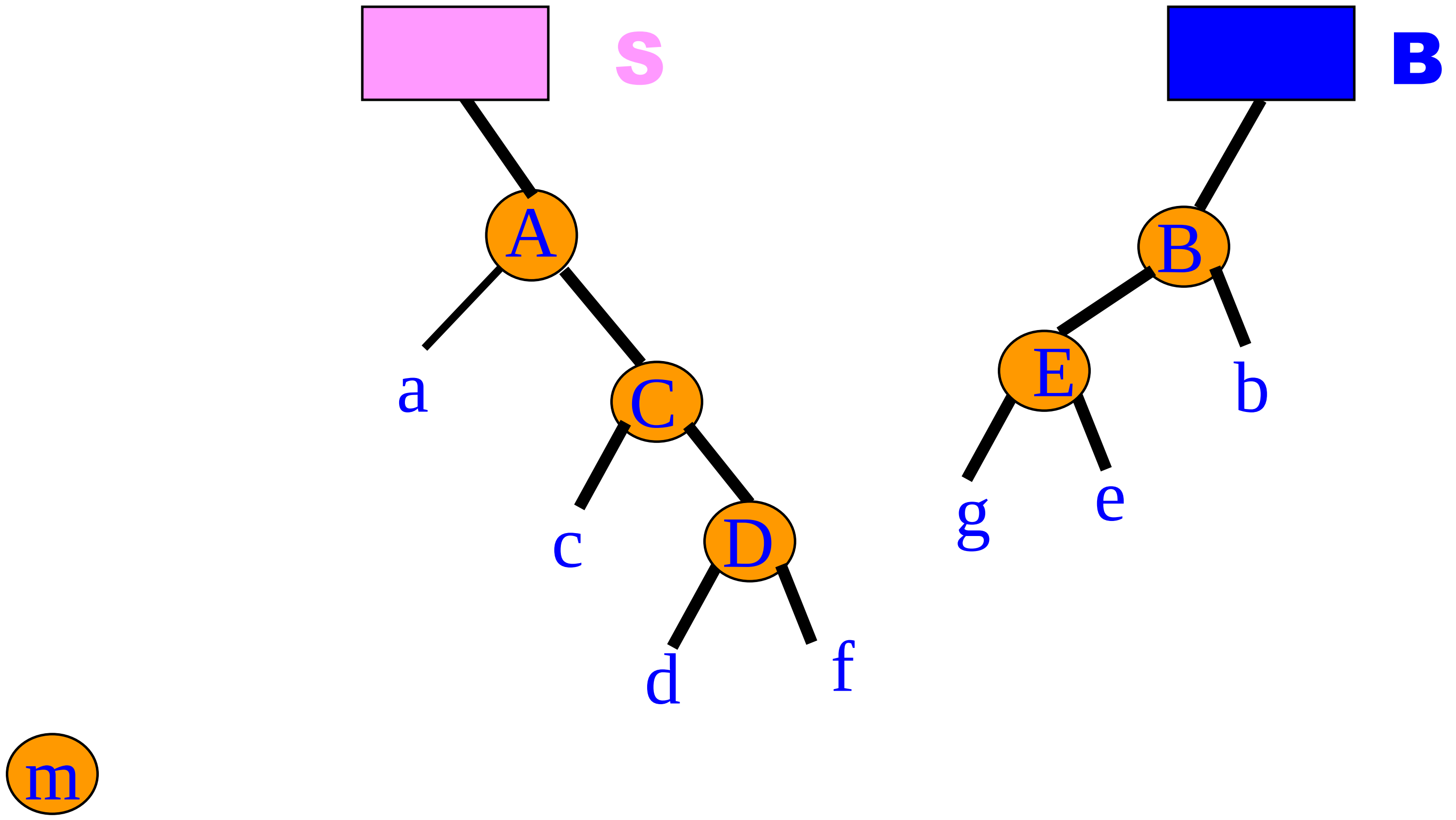
Split A Binary Search Tree



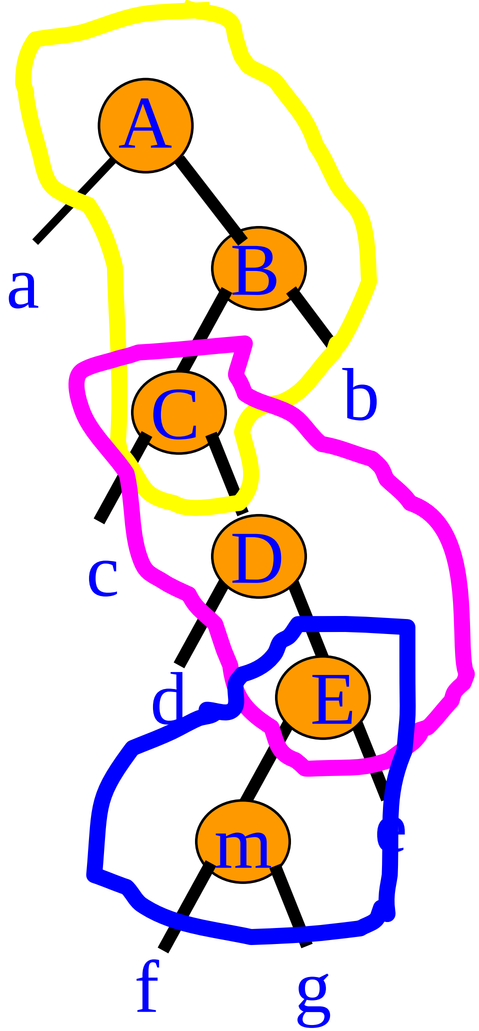
Split A Binary Search Tree



Split A Binary Search Tree

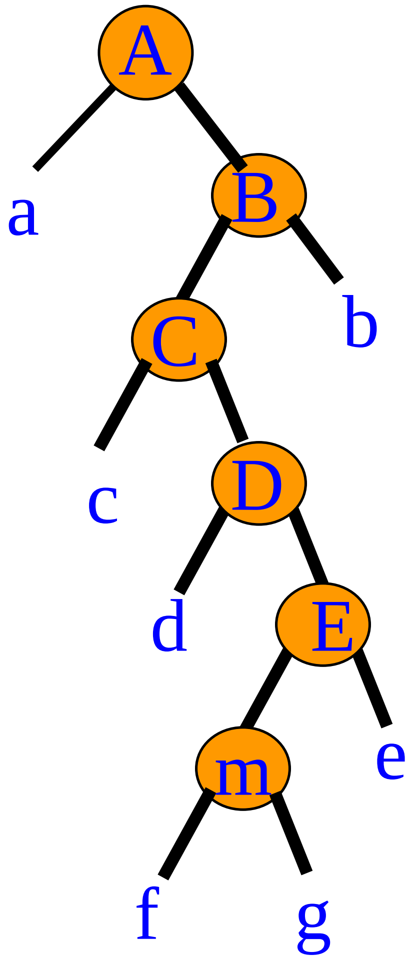


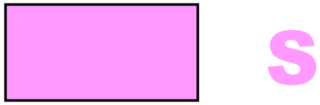
Two-Level Moves

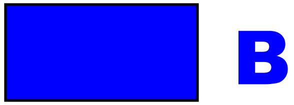


- Let m be the splay node.
- RL move from A to C.
- RR move from C to E.
- L move from E to m.

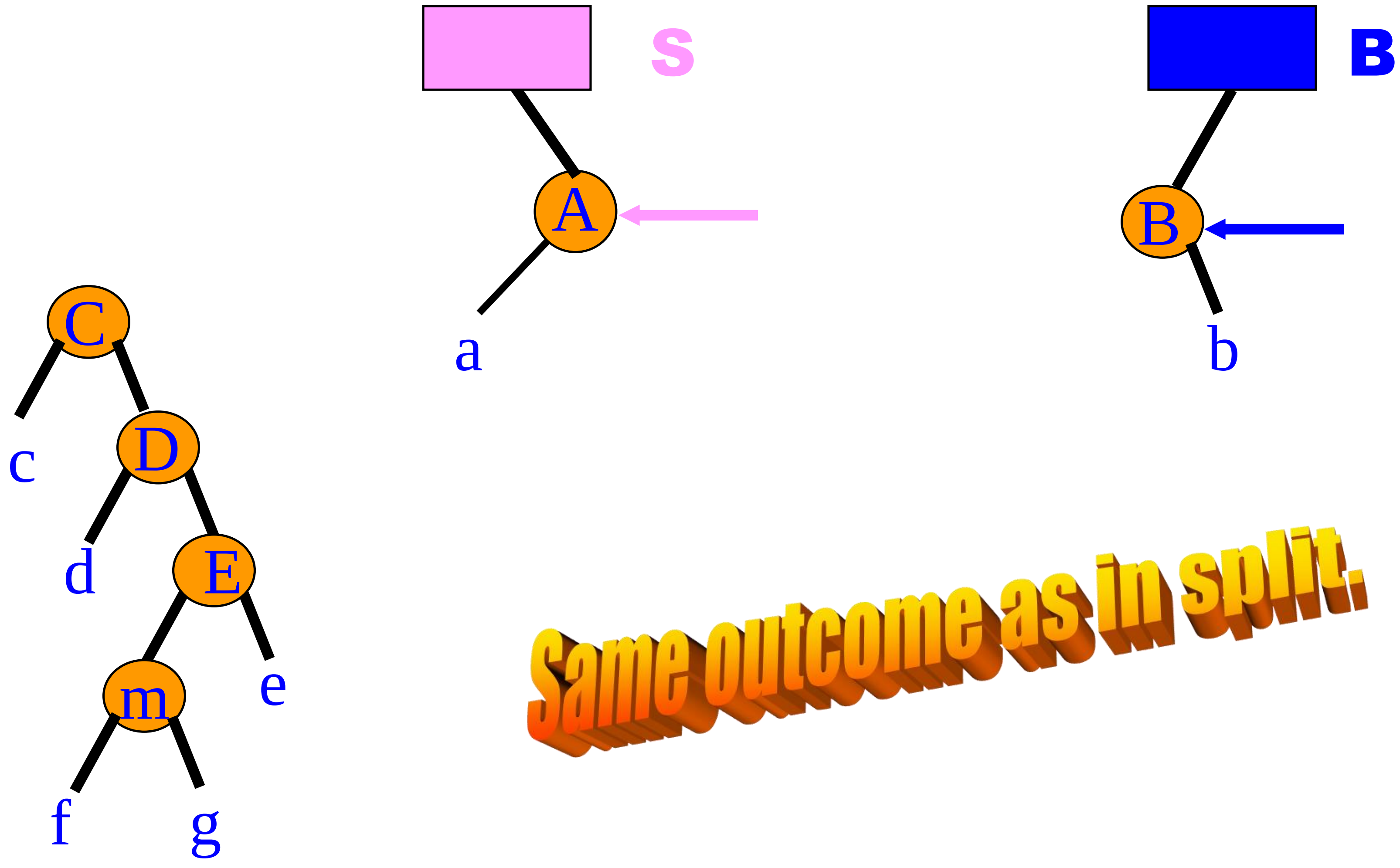
RL Move





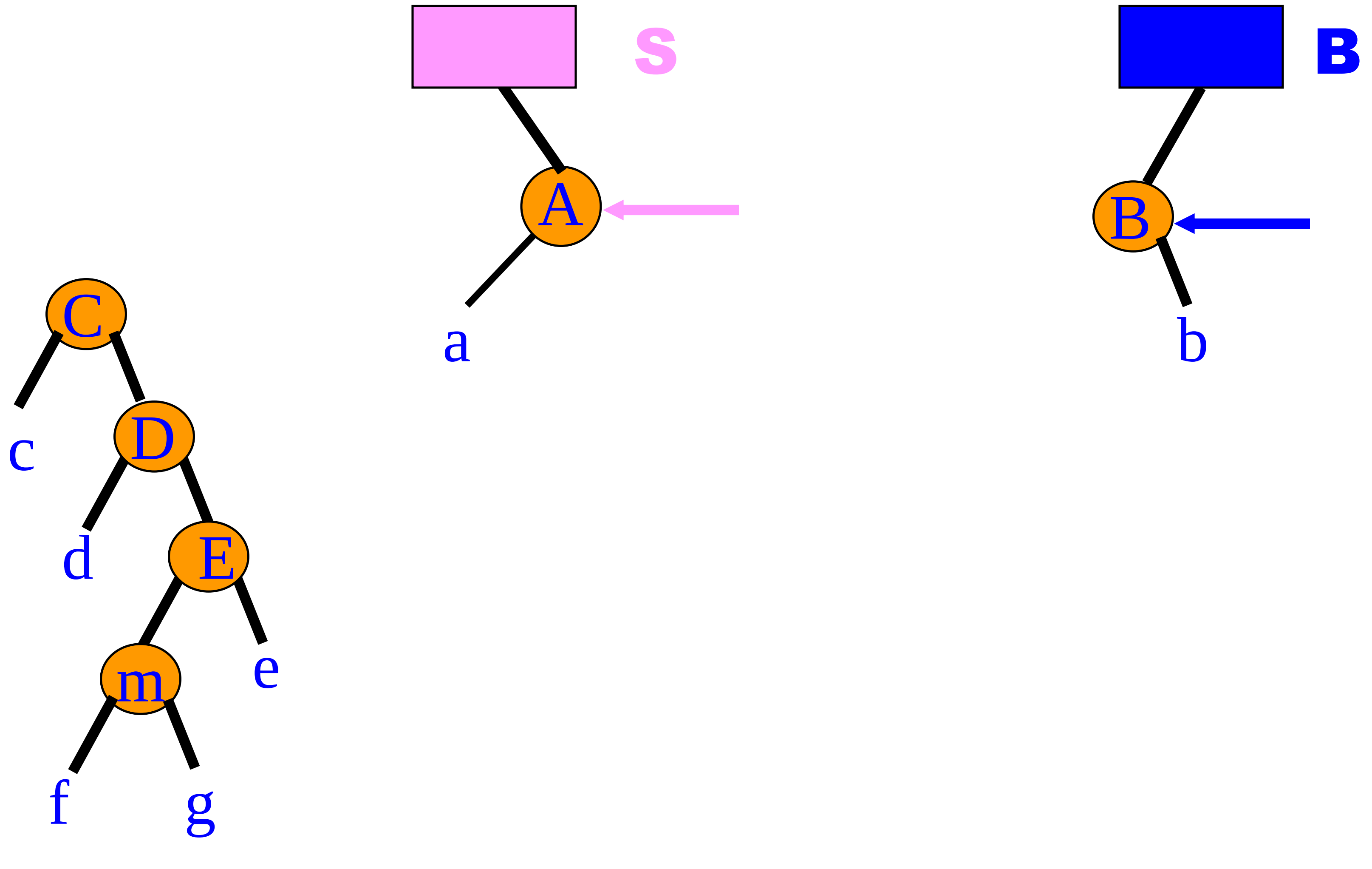


RL Move

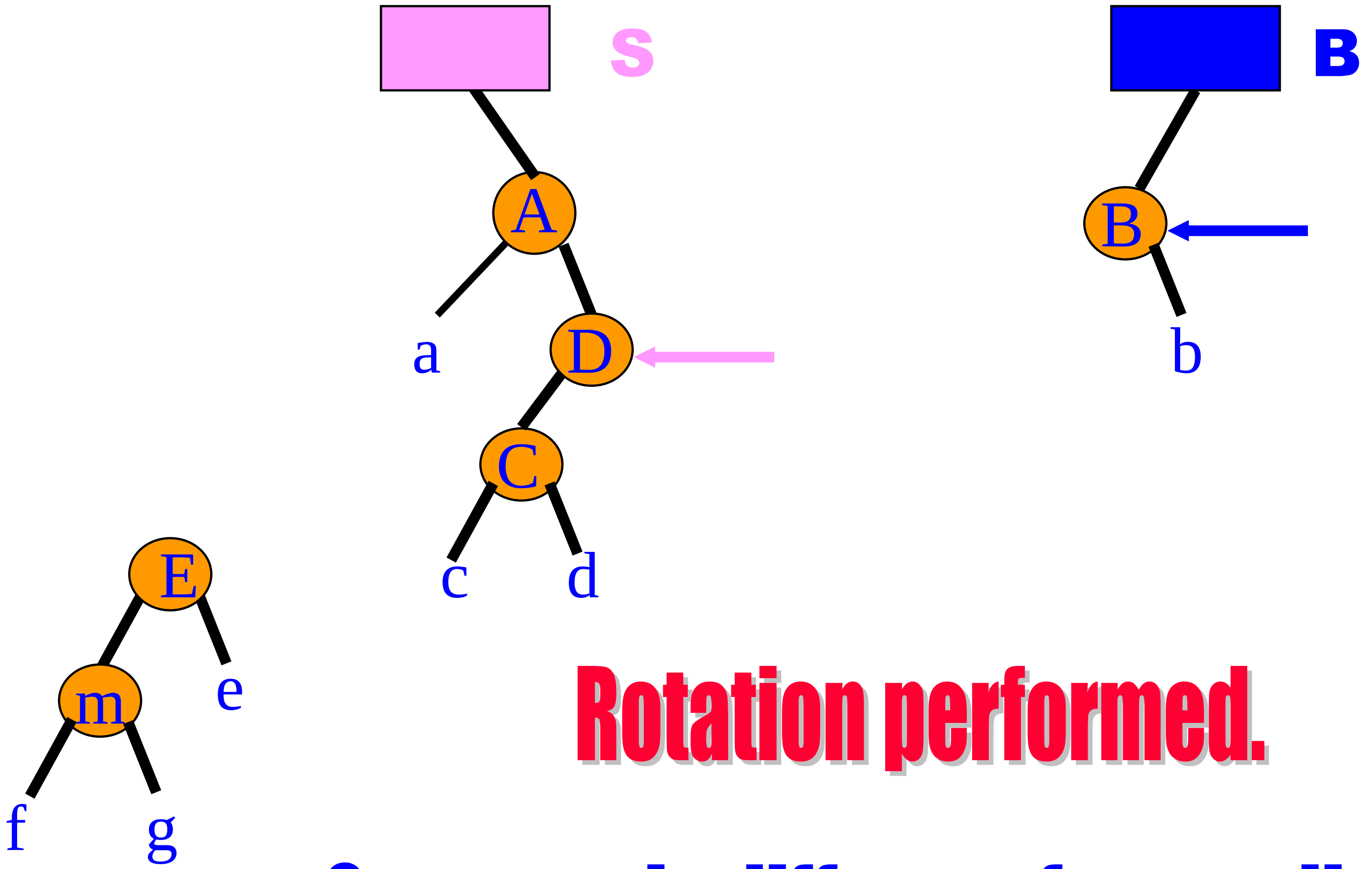


Same outcome as in split.

RR Move



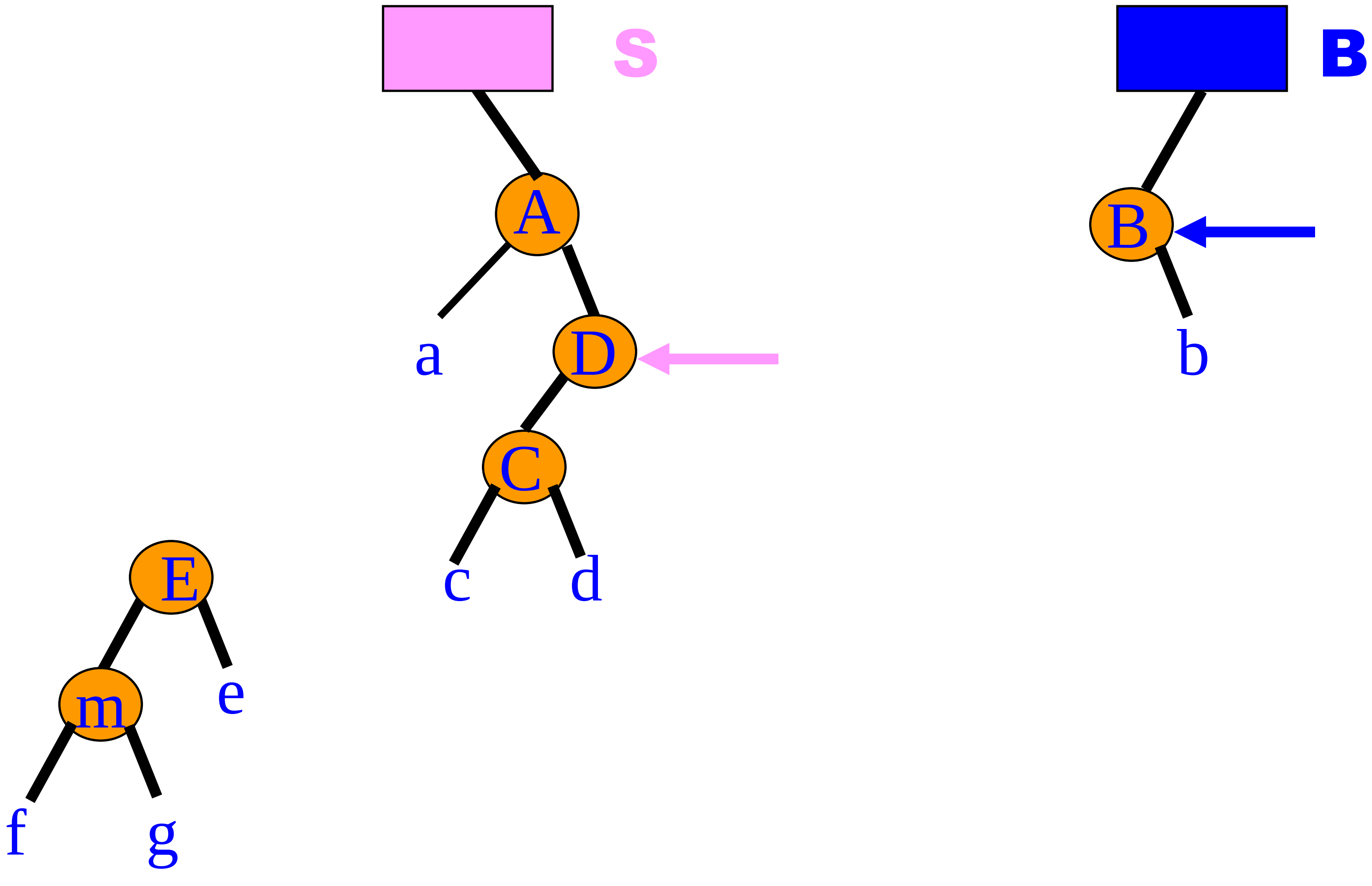
RR Move



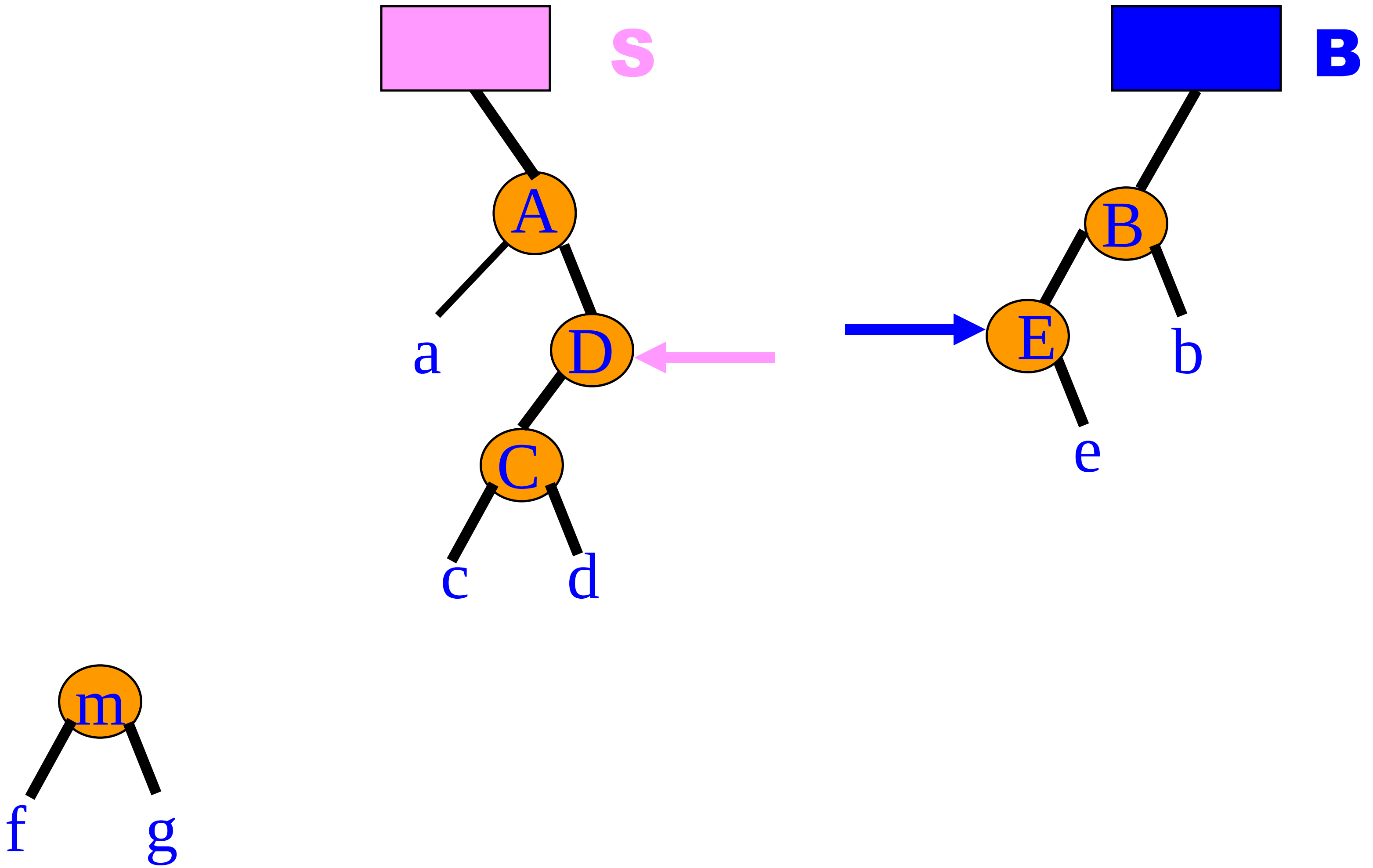
Rotation performed.

Outcome is different from split.

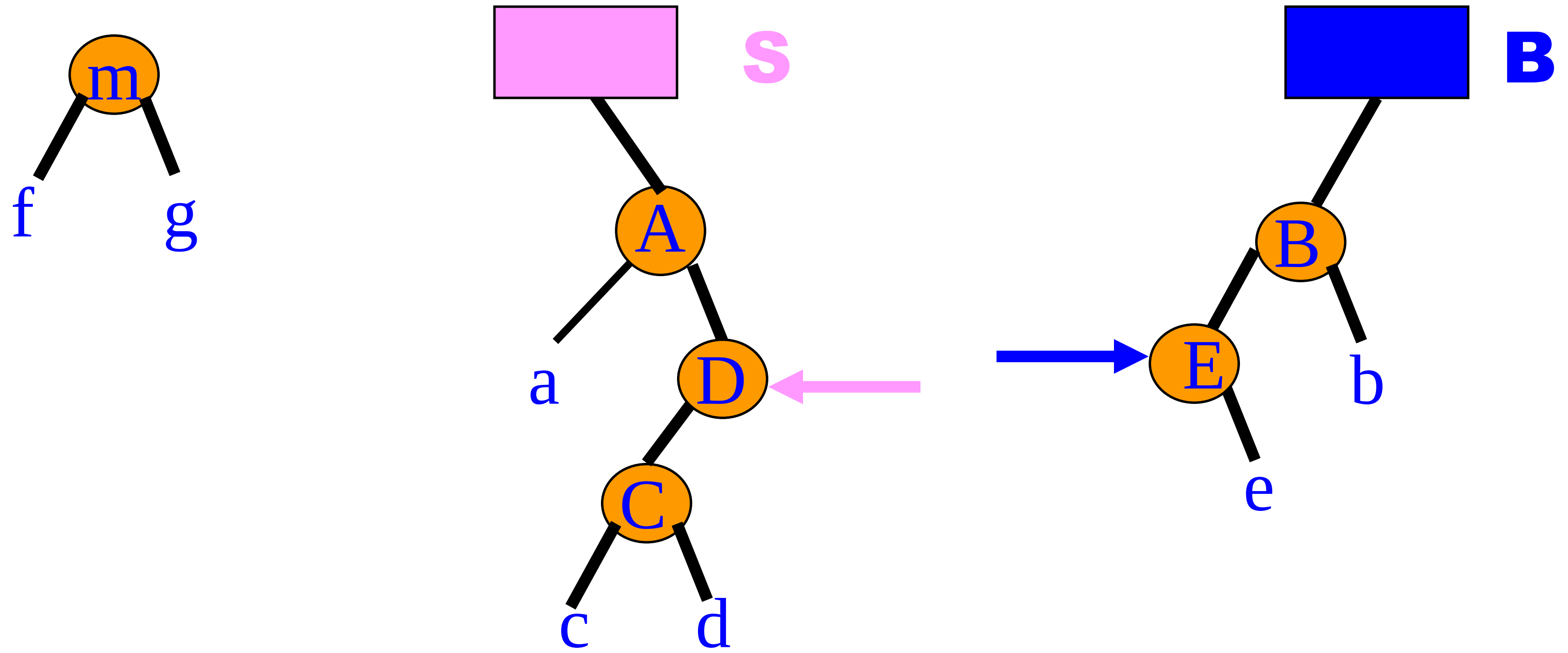
L Move



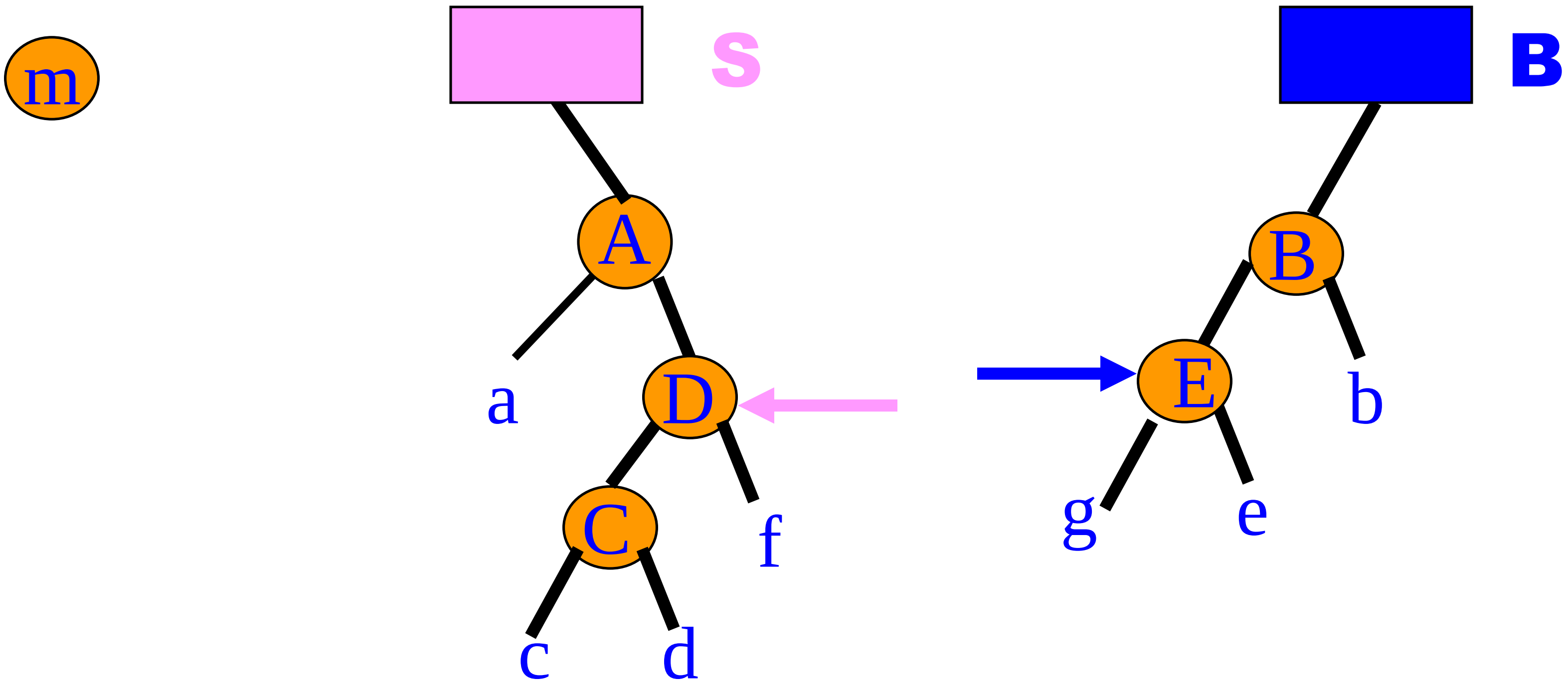
L Move



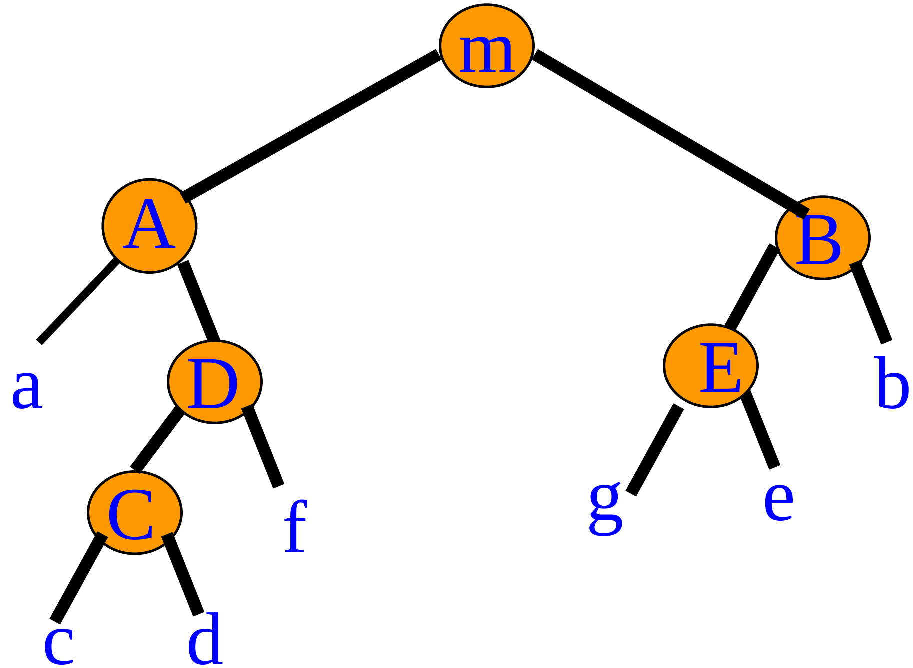
Wrap Up



Wrap Up



Wrap Up



Bottom Up vs Top Down

- Top down splay trees are faster than bottom up splay trees.

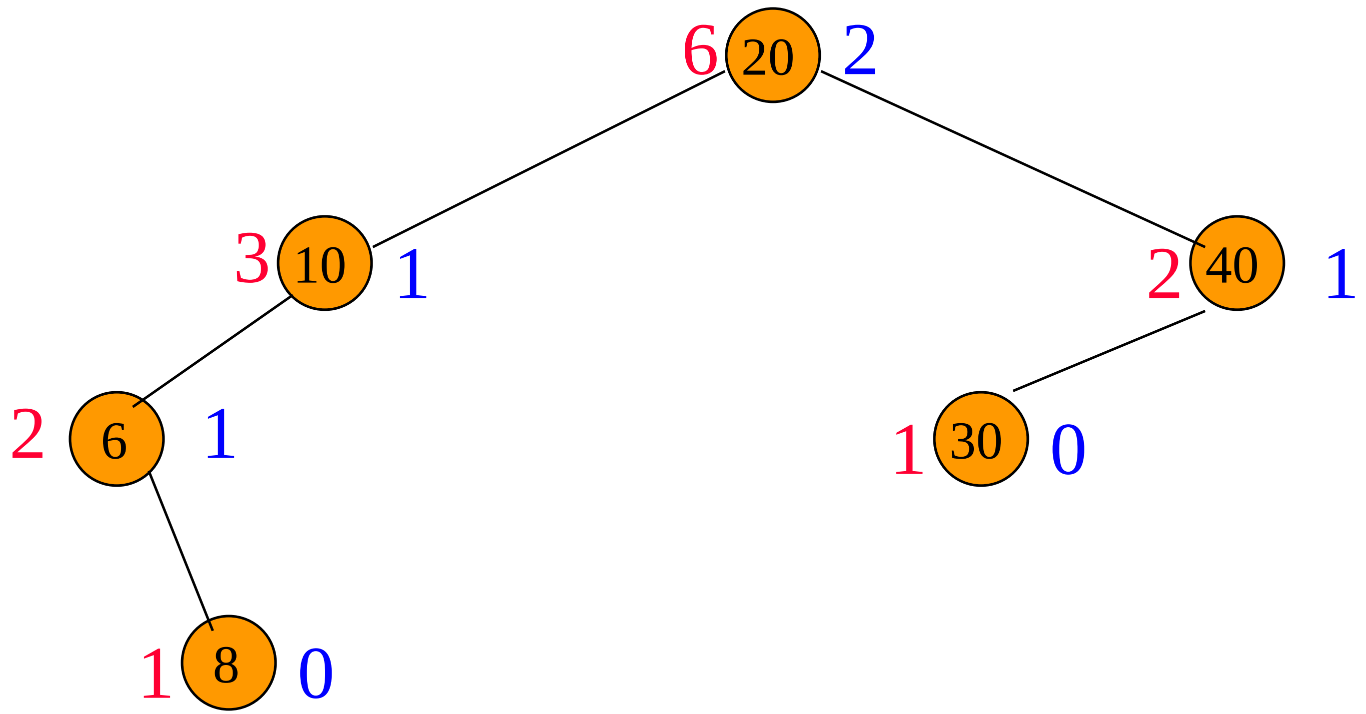
Bottom-Up Splay Trees–Analysis

- Actual and amortized complexity of join is O(1).
- Amortized complexity of search, insert, delete, and split is O(log n).
- Actual complexity of each splay tree operation is the same as that of the associated splay.
- Sufficient to show that the amortized complexity of the splay operation is O(log n).

Potential Function

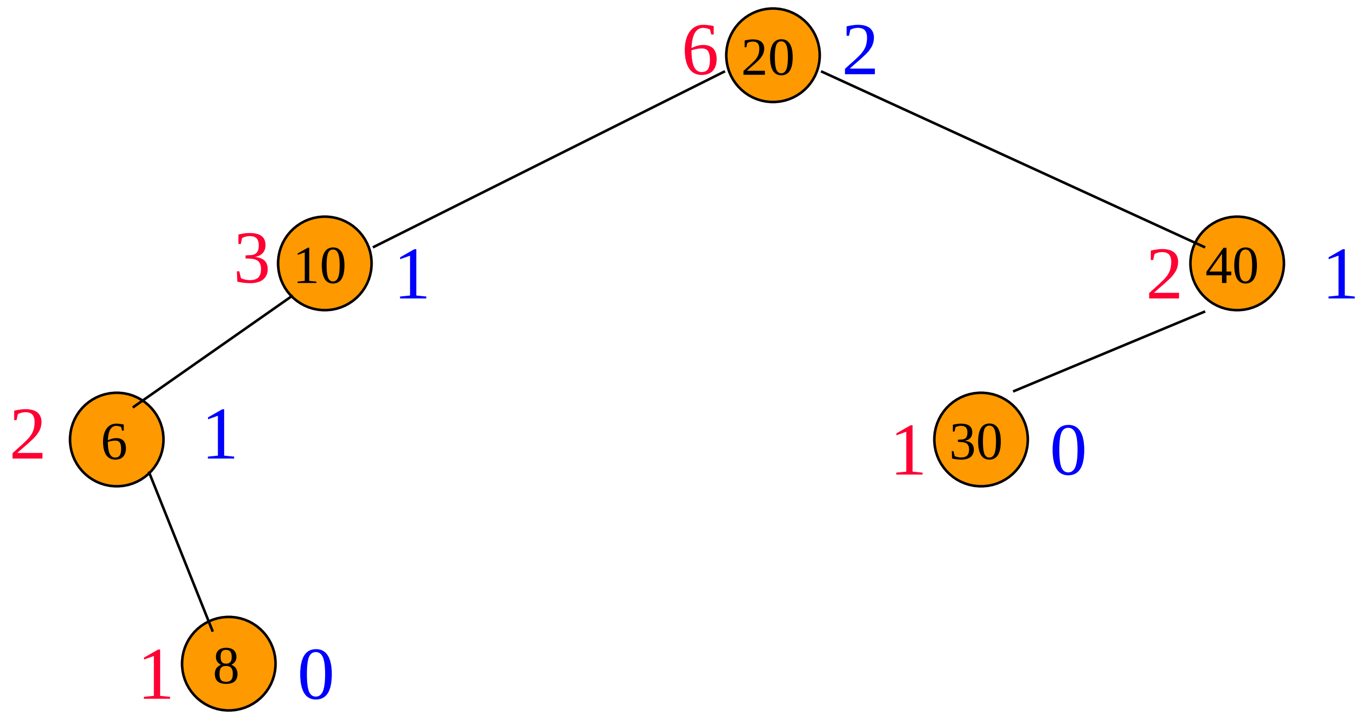
- size(x) = #nodes in subtree whose root is x.
- rank(x) = floor($\log_2$ size(x)).
- P(i) = $\Sigma_{x \text{ is a tree node}}$ rank(x).
  - P(i) is potential after i'th operation.
  - size(x) and rank(x) are computed after i'th operation.
  - P(0) = 0.
- When join and split operations are done, number of splay trees > 1 at times.
  - P(i) is obtained by summing over all nodes in all trees.

Example



- size(x) is in red.
- rank(x) is in blue.
- Potential = 5.

Example



- $\text{rank(root)} = \text{floor}(\log_2 n)$.
- When you insert, potential may increase by $\text{floor}(\log_2 n)+1$.

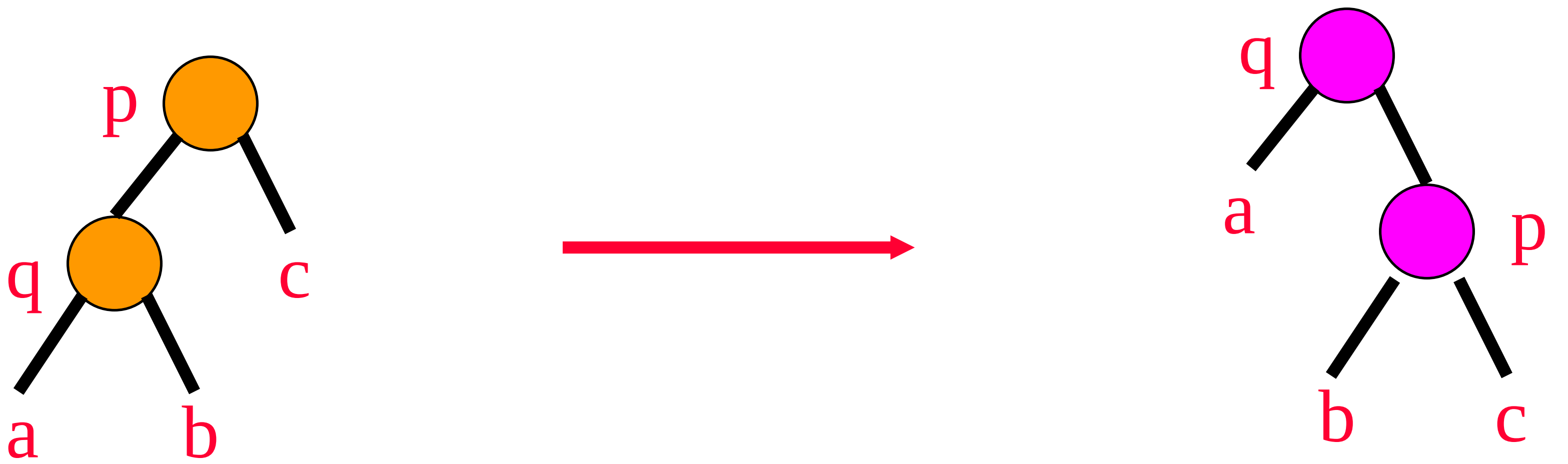
Splay Step Amortized Cost

- If q = null or q is the root, do nothing (splay is over).
- $\Delta P = 0$.
- amortized cost $=$ actual cost $+ \Delta P$

  $= 0$.

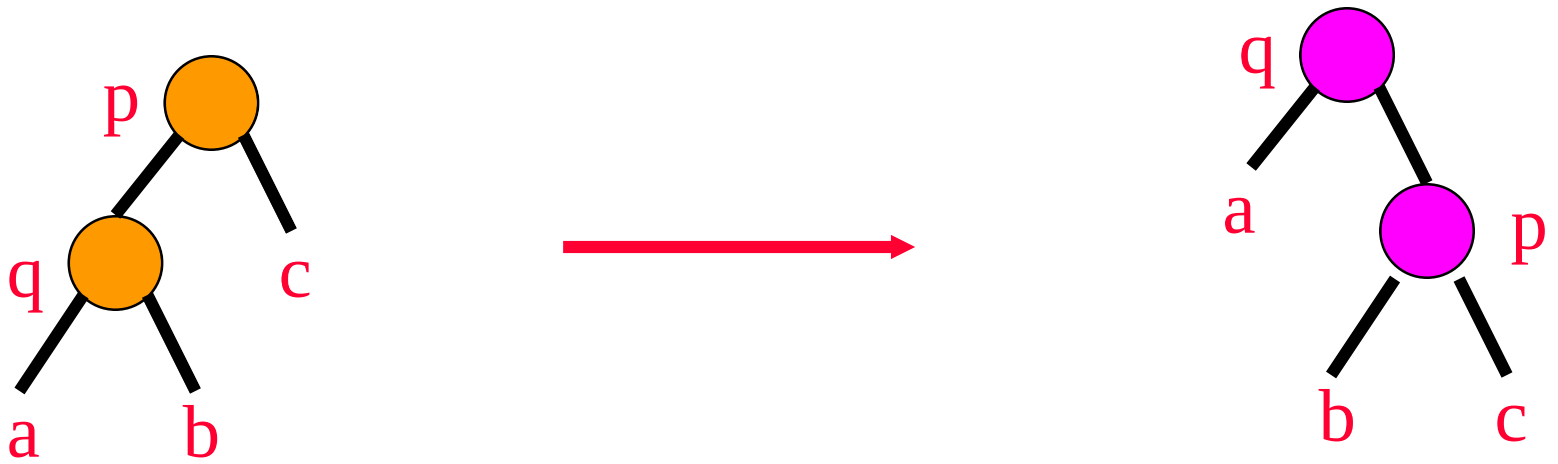
Splay Step Amortized Cost

- If q is at level 2, do a one-level move and terminate the splay operation.



- $r(x)$ = rank of x before splay step.
- $r'(x)$ = rank of x after splay step.

Splay Step Amortized Cost



- $\Delta P = r'(p) + r'(q) - r(p) - r(q)$

  $\quad\quad <= r'(q) - r(q).$

- amortized cost = actual cost + ΔP

  $\quad\quad <= 1 + r'(q) - r(q).$

2-Level Move (case 1)

- $\Delta P = r'(gp) + r'(p) + r'(q) - r(gp) - r(p) - r(q)$

2-Level Move (case 1)

- r’(q) = r(gp)
- r’(p) <= r’(q)
- r’(gp) <= r’(q)
- r (q) <= r(p)

2-Level Move (case 1)

- $\Delta P = r'(gp) + r'(p) + \cancel{r'(q)} - \cancel{r(gp)} - r(p) - r(q)$
- $r'(q) = r(gp)$
- $r'(gp) <= r'(q)$
- $r'(p) <= r'(q)$
- $r(q) <= r(p)$
- $\Delta P <= r'(q) + r'(q) - r(q) - r(q)$

  $= 2(r'(q) - r(q))$

2-Level Move (case 1)

A more careful analysis reveals that

$\Delta P <= 3(r'(q) - r(q)) - 1$ (see text for proof)

2-Level Move (case 1)

- amortized cost = actual cost + ΔP

$$\le 1 + 3(r'(q) - r(q)) - 1$$

$$= 3(r'(q) - r(q))$$

2-Level Move (case 2)

- Similar to Case 1.

Splay Operation

- When q != null and q is not the root, zero or more 2-level splay steps followed by zero or one 1-level splay step.
- Let r''(q) be rank of q just after last 2-level splay step.
- Let r'''(q) be rank of q just after 1-level splay step.

Splay Operation

- Amortized cost of all 2-level splay steps is $<= 3(r''(q) - r(q))$
- Amortized cost of splay operation

$<= 1 + r'''(q) - r''(q) + 3(r''(q) - r(q))$

$<= 1 + 3(r'''(q) - r''(q)) + 3(r''(q) - r(q))$

$= 1 + 3(r'''(q) - r(q))$

$<= 3(\text{floor}(\log_2 n) - r(q)) + 1$

Actual Cost Of Operation Sequence

- Actual cost of an n operation sequence

  = O(actual cost of the associated n splays).
- actual_cost_splay(i) = amortized_cost_splay(i) – ΔP

  $\le 3(\text{floor}(\log_2 i) - r(q)) + 1 + P'(i) - P(i)$
- P'(i) = potential just before i'th splay.
- P(i) = potential just after i'th splay.
- $P'(i) \le P(i - 1) + \text{floor}(\log_2 i)$

Actual Cost Of Operation Sequence

- actual_cost_splay(i) = amortized_cost_splay(i) – ΔP

  $<= 3(\text{floor}(\log_2 i) – r(q)) + 1 + P'(i) – P(i)$

  $<= 3 * \text{floor}(\log_2 i) + 1 + P'(i) – P(i)$

  $<= 4 * \text{floor}(\log_2 i) + 1 + P(i – 1) – P(i)$
- $P(0) = 0$ and $P(n) >= 0$.
- Σ_i actual_cost_splay(i)

  $<= 4n * \text{floor}(\log_2 n) + n + P(0) – P(n)$

  $<= 5n * \text{floor}(\log_2 n)$

  $= O(n \log n)$