Red Black Trees

Ira A. Fulton Schools of Engineering
Arizona State University

Red Black Trees, Part 1

- Definition and Properties of Red-Black Trees
- Rotations, RBT-Insert Preparation
- RBT-Insert-Fixup (3 cases)
- Examples, Running time of RBT-Insertion
- RBT-Deletion Preparation
- RBT-Deletion-Fixup (4 cases)
- Examples and Summary

Problems with Binary Search Trees

- $\boldsymbol{O(h)}$ time operations
- However, h could be as large as $n-1$
- Can we make h upper-bounded by $O(\log n)$?
- If so, can we use $O(\log n)$ time for each operation?

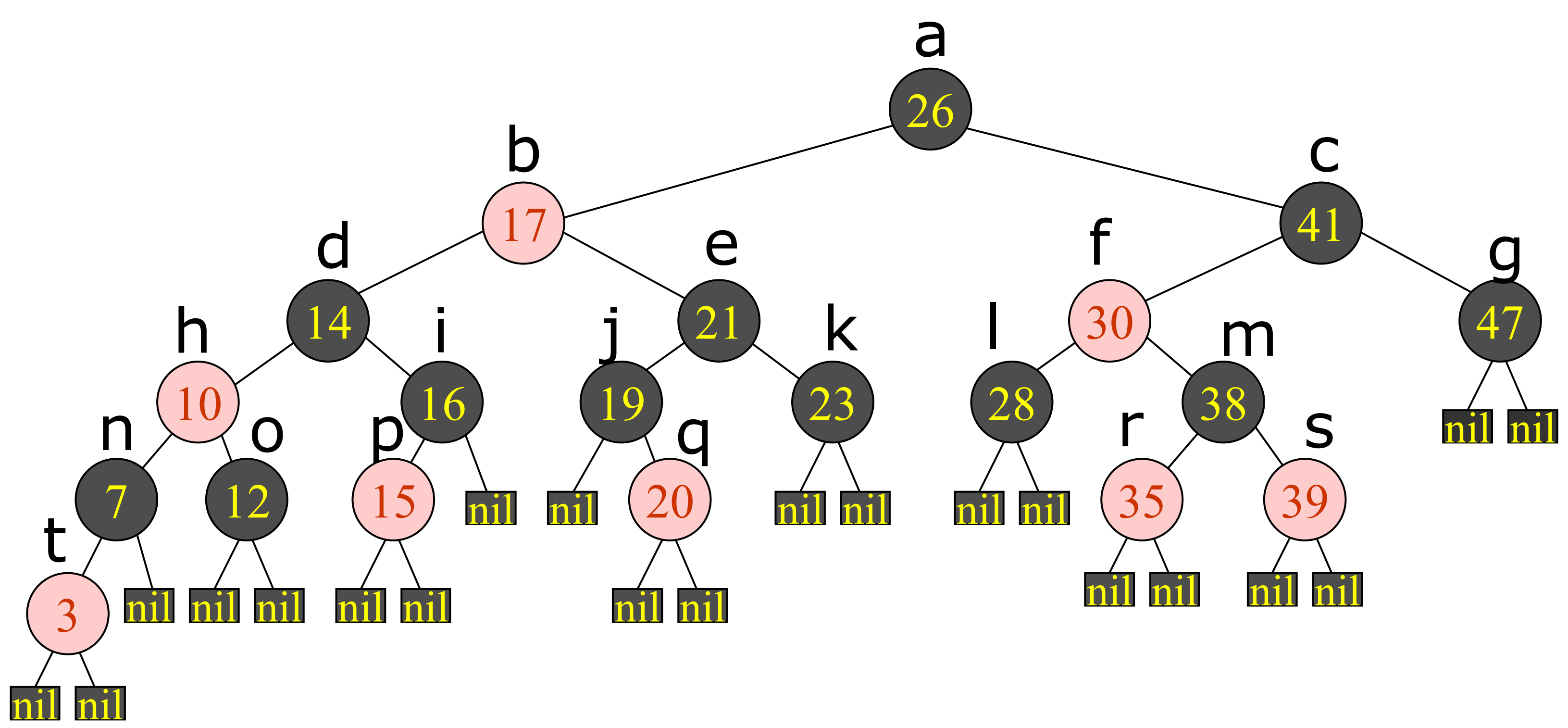
- Yes!

Red-Black Trees: Definition

A binary search tree is a red-black tree if each non-nil node has a color (red or black) and it satisfies the following five properties

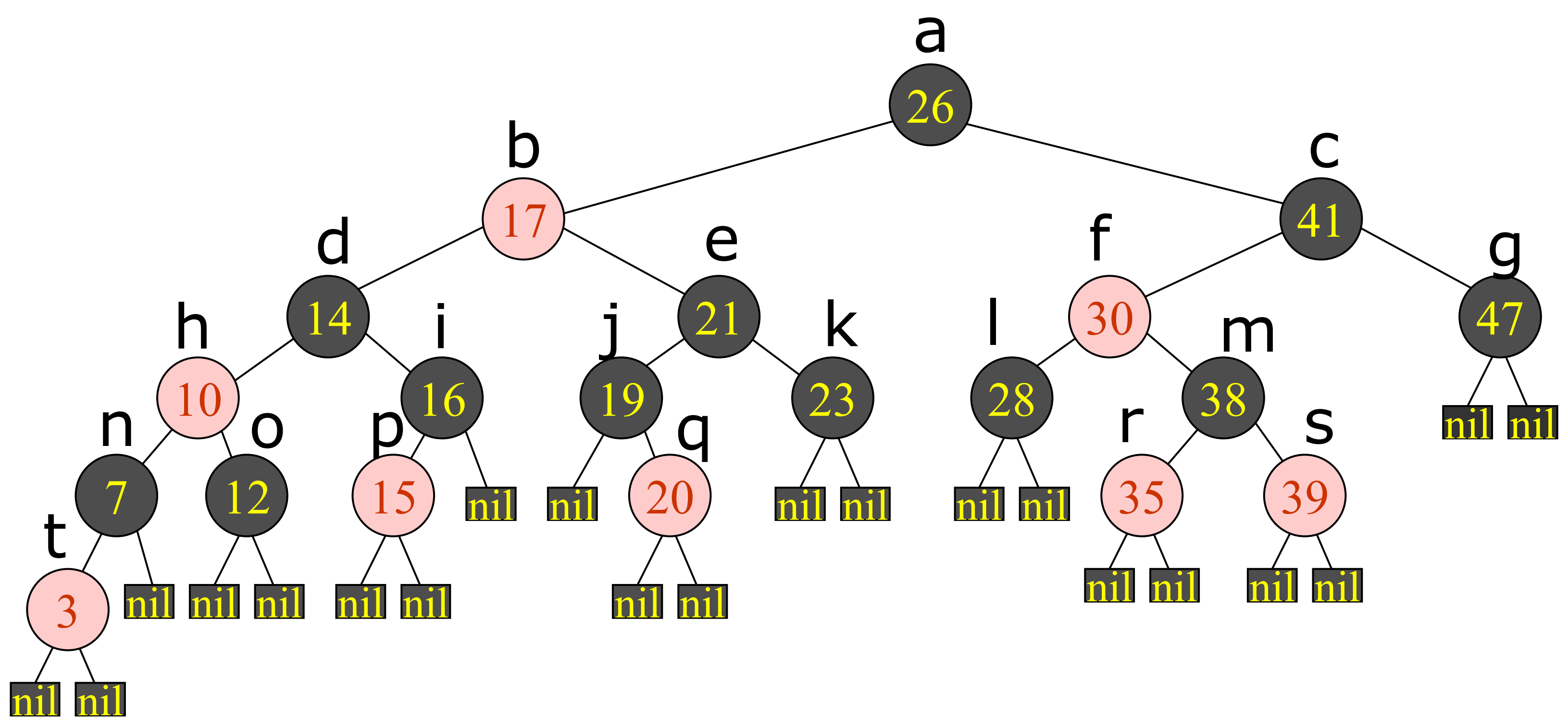
1. **Every node is either red or black;**
2. **The root is black;**
3. **Every leaf (nil) is (considered) black;**
4. **If a node is red, then both its children are black;**
5. **Every simple path from a node to a descendant leaf (nil) contains the same number of black nodes.**

Red-Black Trees: Example



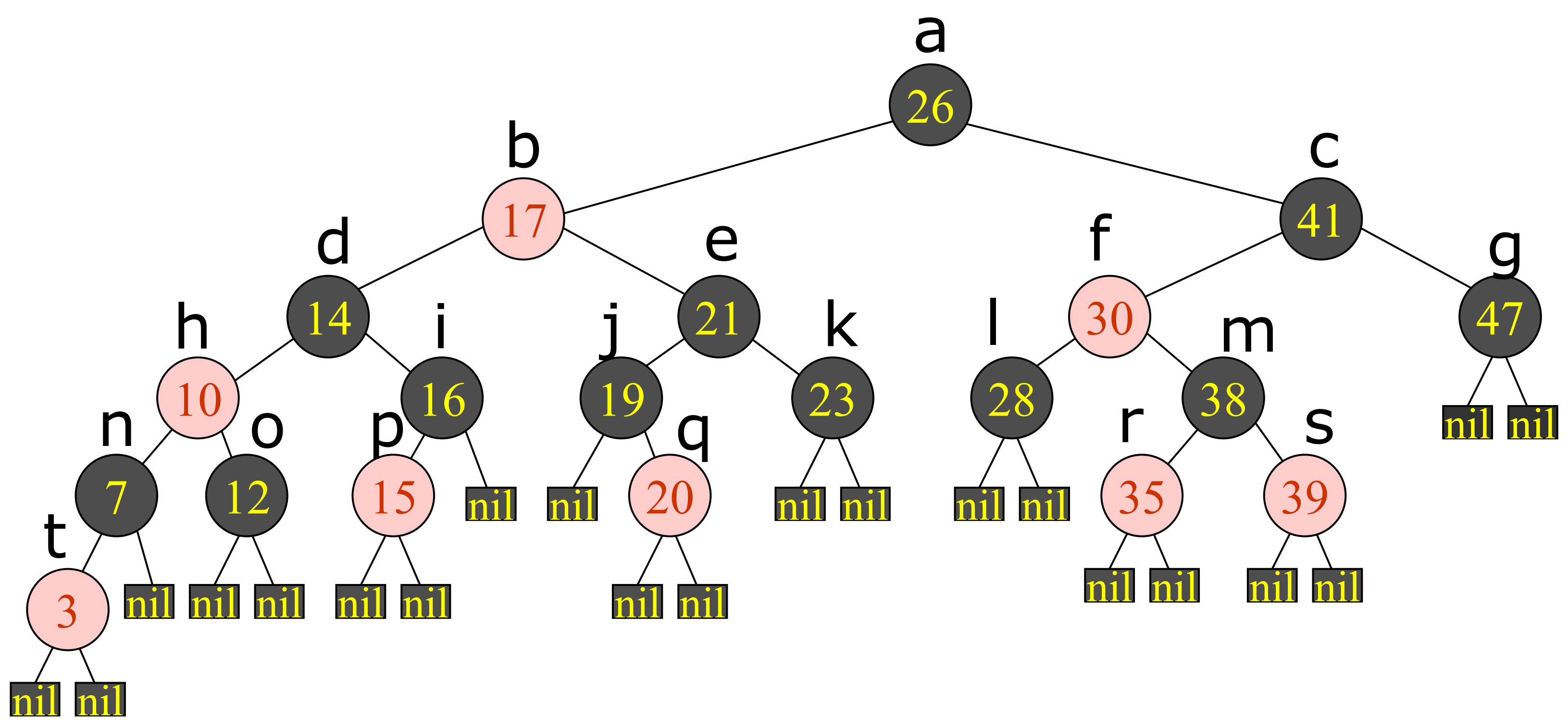
Is it a binary search tree?

Red-Black Trees: Example



Is it a binary search tree? YES

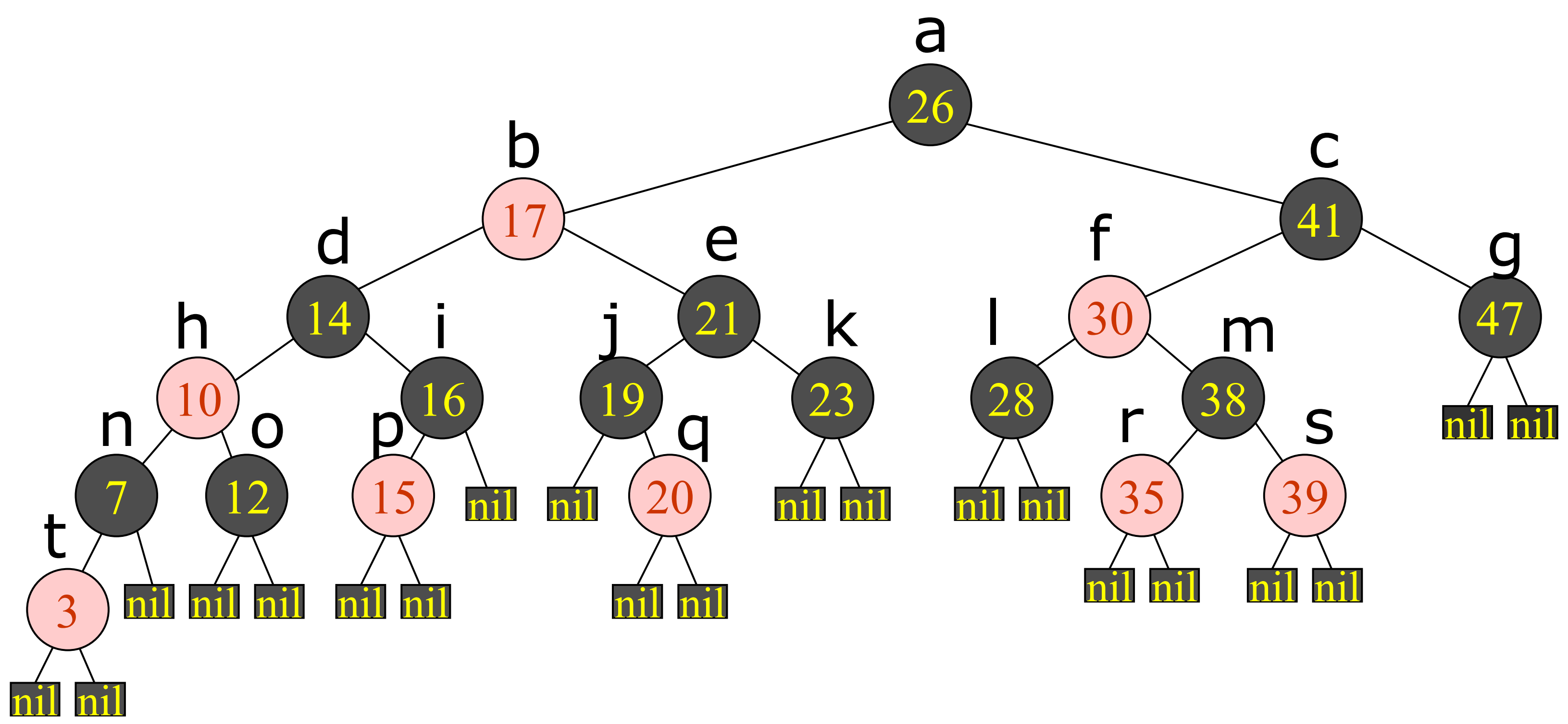
Red-Black Trees: Example



Is it a binary search tree? YES

Does it satisfy the five properties of RB-Trees?

Red-Black Trees: Example



Is it a binary search tree? YES

Does it satisfy the five properties of RB-Trees? YES

Internal nodes and leaf nodes

- We will call the non-leaf nodes in a red-black tree internal nodes or data-bearing nodes.

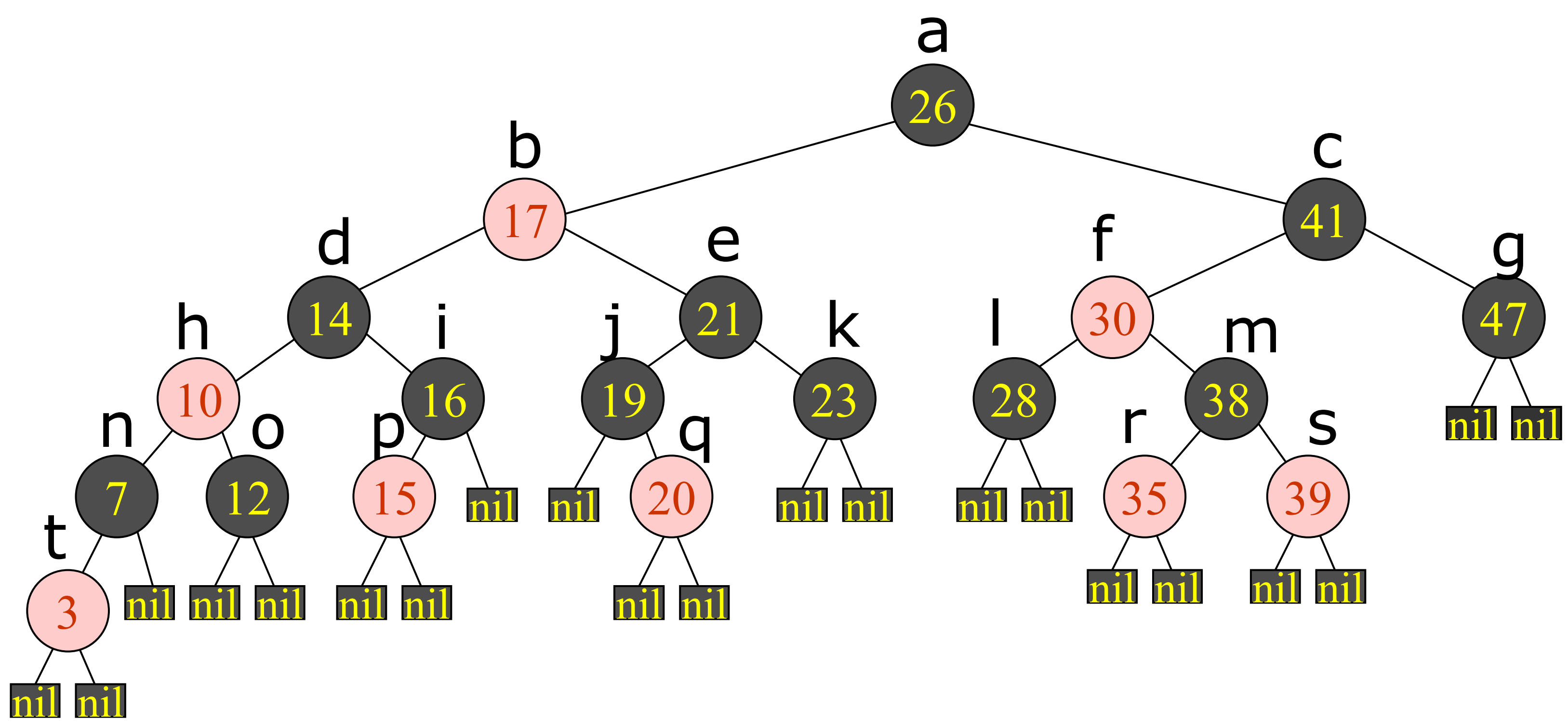
- The internal nodes in a red-black tree form a binary search tree.

- The nil/leaf nodes are all black.

- We call the number of black nodes on any path from, but not including, a node x down to a leaf the black height of the node, denoted by bh(x).

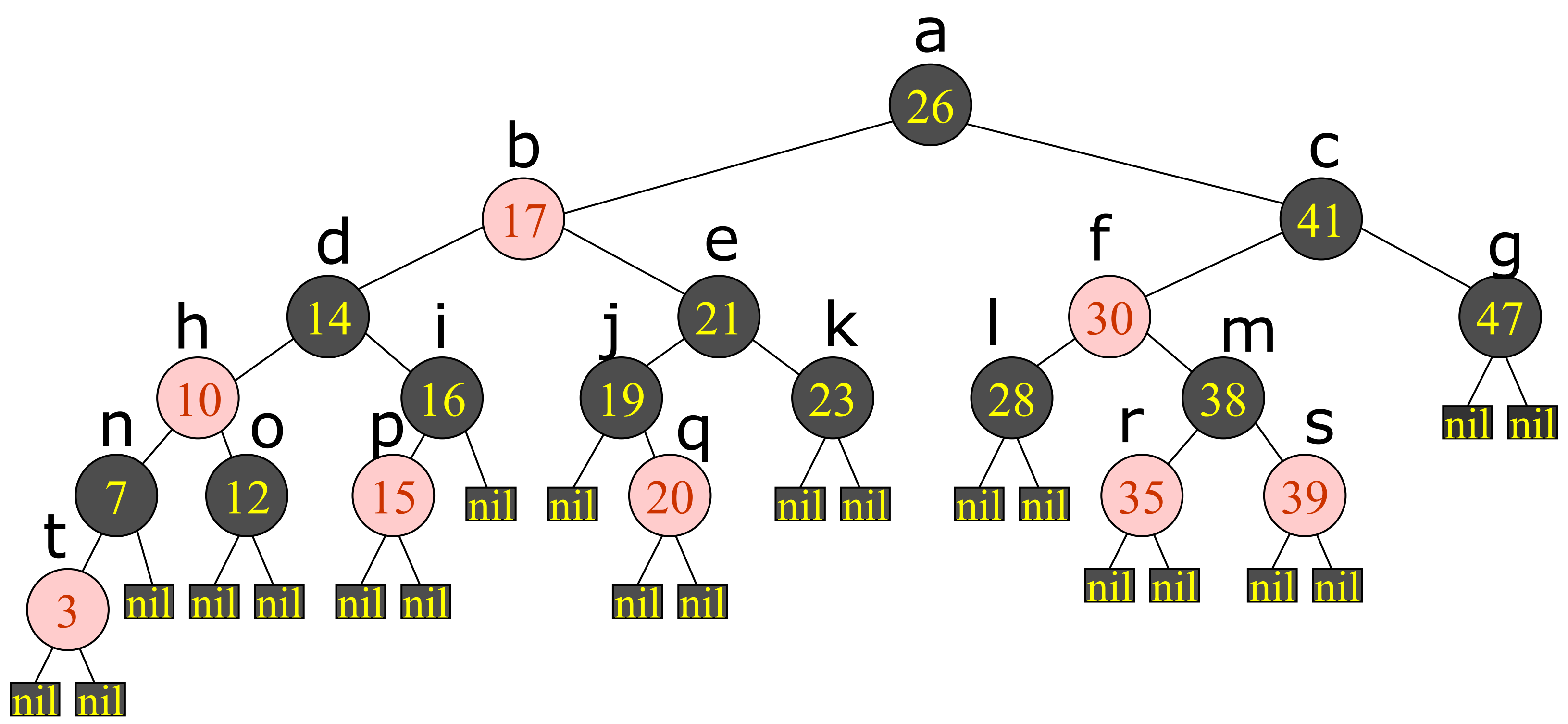
- The black height of the root is the black height of the red-black tree.

Black Height of a Node (Tree)



The black height of a node x is the number of black nodes from x to a descendent leaf node of x, not counting x

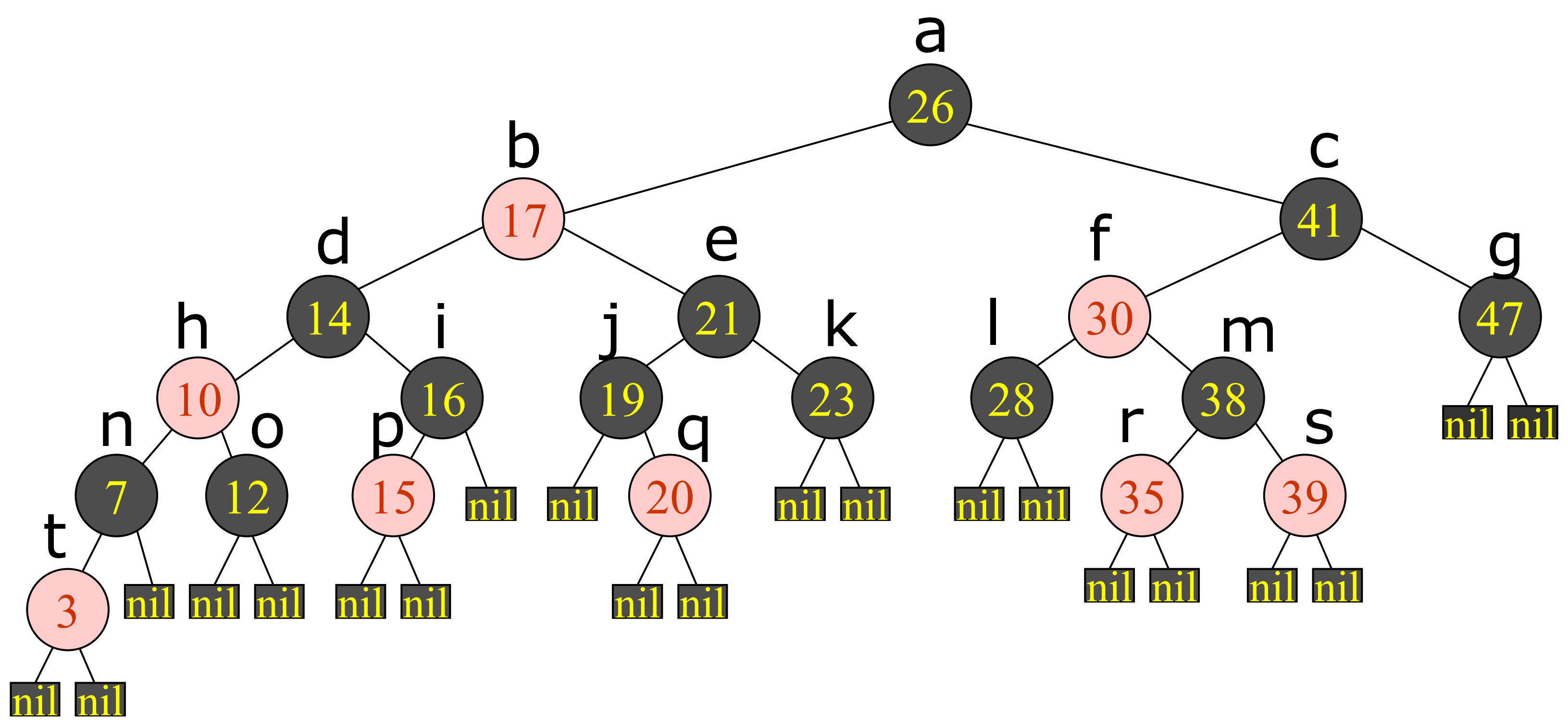
Black Height of a Node (Tree)



The black height of a node x is the number of black nodes from x to a descendent leaf node of x, not counting x

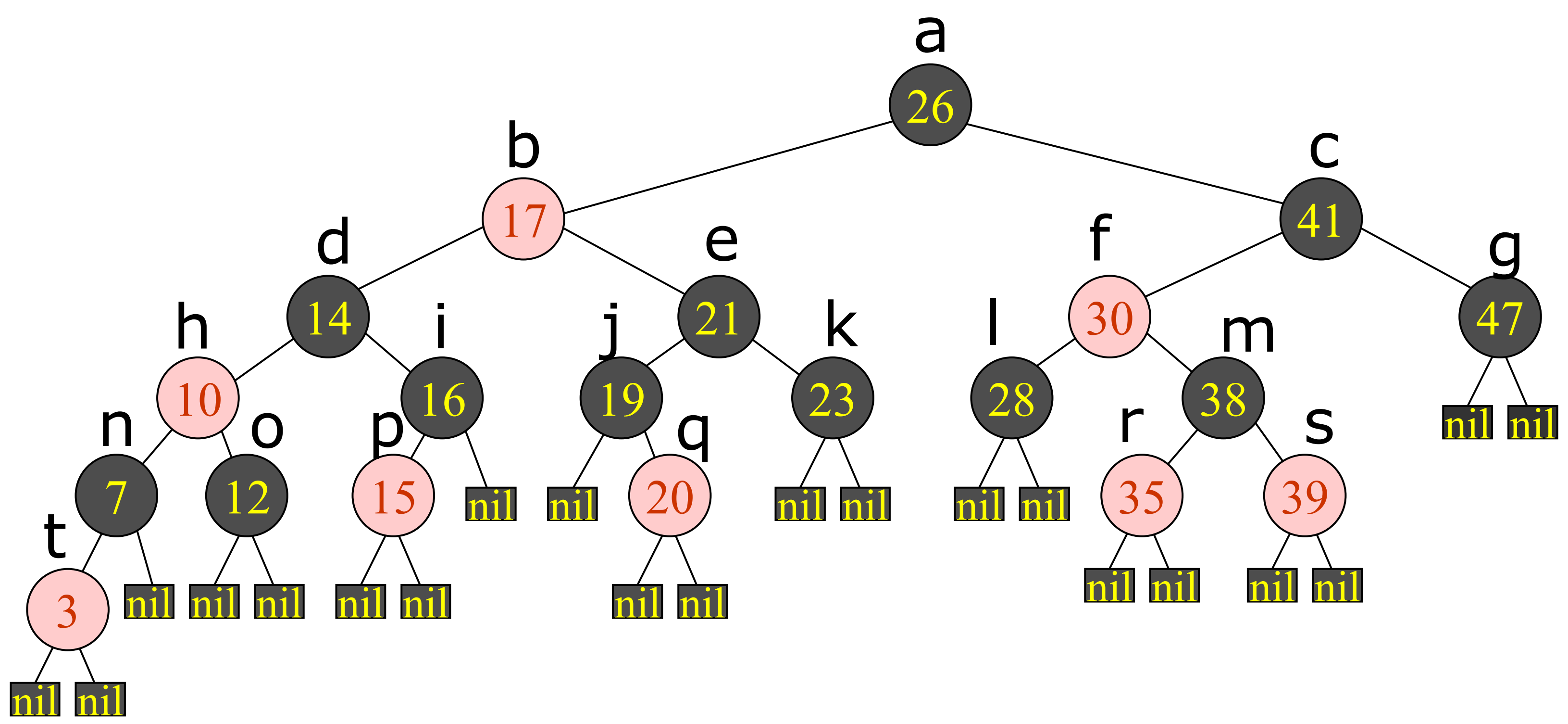
$bh(d) = bh(h) = 2, bh(i) = 1.$

Black Height of a Node (Tree)



The black height of a RBT T is the maximum black height of nodes in T

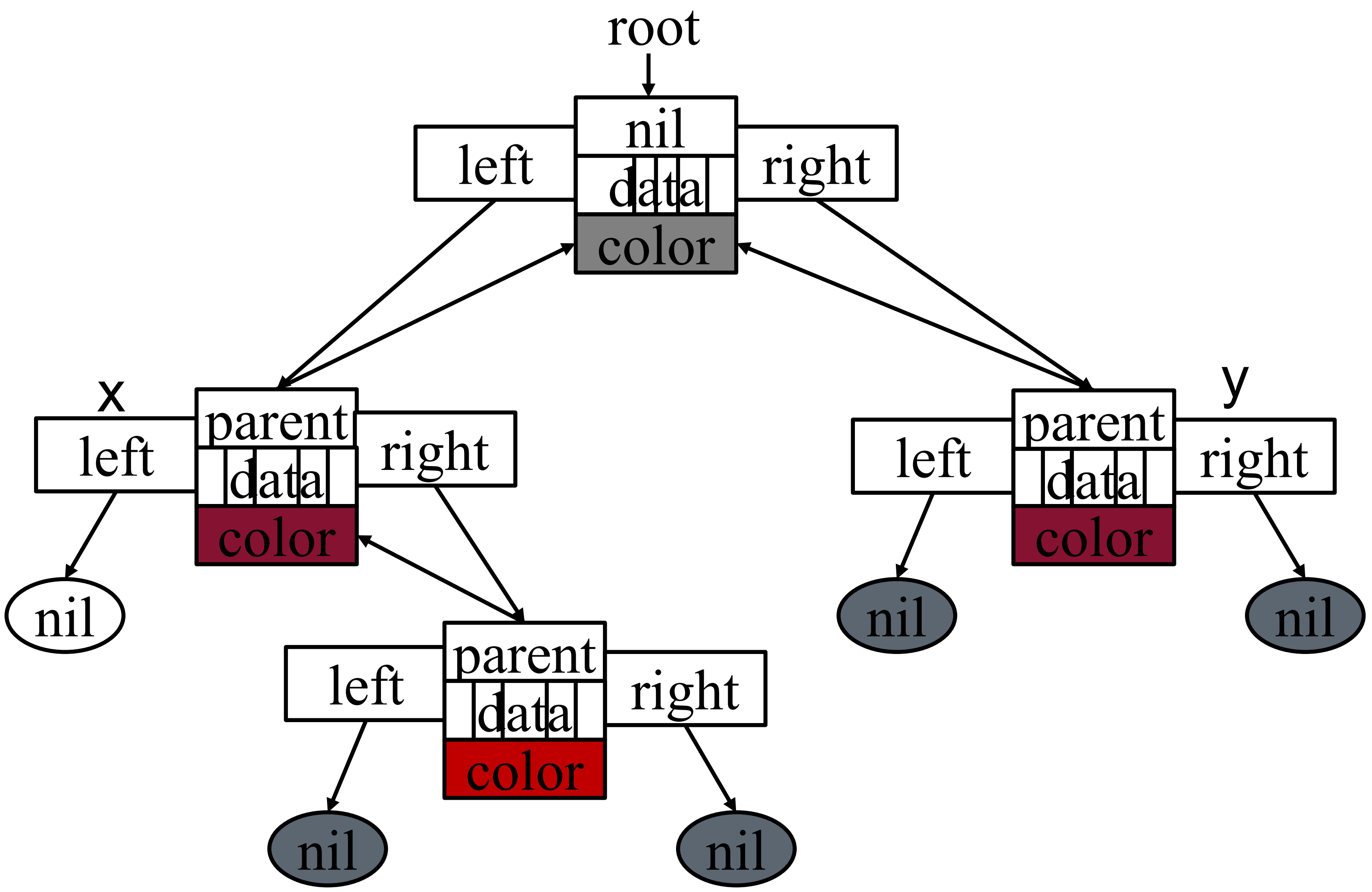
Black Height of a Node (Tree)



The black height of a RBT T is the maximum black height of nodes in T

$$bh(T) = 3$$

Structure of a Tree Node (5 fields, at least)



RB-Tree Height

Lemma: Let x be any node in an RB-Tree. Then the number of internal nodes in the subtree rooted at x is at least $2^{bh(x)} - 1$. Let $size(x)$ denote the number of internal nodes in the subtree rooted at x. The lemma claims that $size(x) \geq 2^{bh(x)} - 1$.

Proof. By induction on the height $h(x)$ of node x.

Base case: $h(x) = 0$.

x is a leaf node. $bh(x) = 0$. $size(x) = 0$ which is also $2^{bh(x)} - 1$.

Base case proved for $h(x) = 0$.

RB-Tree Height

Induction Step:

- **Assume that $h(x) < k$ implies $size(x) \geq 2^{bh(x)} - 1$ for some $k \geq 1$.**

- **Now $h(x) = k$. Then x must be an internal node.**

- **Let y and z be the left child and right child of x, respectively. We have $h(y) \leq h(x) - 1 < k$ and $h(z) \leq h(x) - 1 < k$. Also, $bh(y) \geq bh(x) - 1$ and $bh(z) \geq bh(x) - 1$. Hence**

- $size(y) \geq 2^{bh(y)} - 1 \geq 2^{bh(x)-1} - 1,$

- $size(z) \geq 2^{bh(z)} - 1 \geq 2^{bh(x)-1} - 1.$

- $size(x) = size(y) + size(z) + 1 \geq 2^{bh(x)} - 1.$

RB-Tree Height

Lemma 13.1: A red-black tree with n internal nodes has height at most $2\log(n+1)$.

Proof. Let h be the tree height and bh the black tree height. By property 4, $bh \geq h/2$.

From the previous lemma, we have

$n \geq 2^{bh} - 1 \geq 2^{h/2} - 1.$

Hence

$n + 1 \geq 2^{h/2}.$

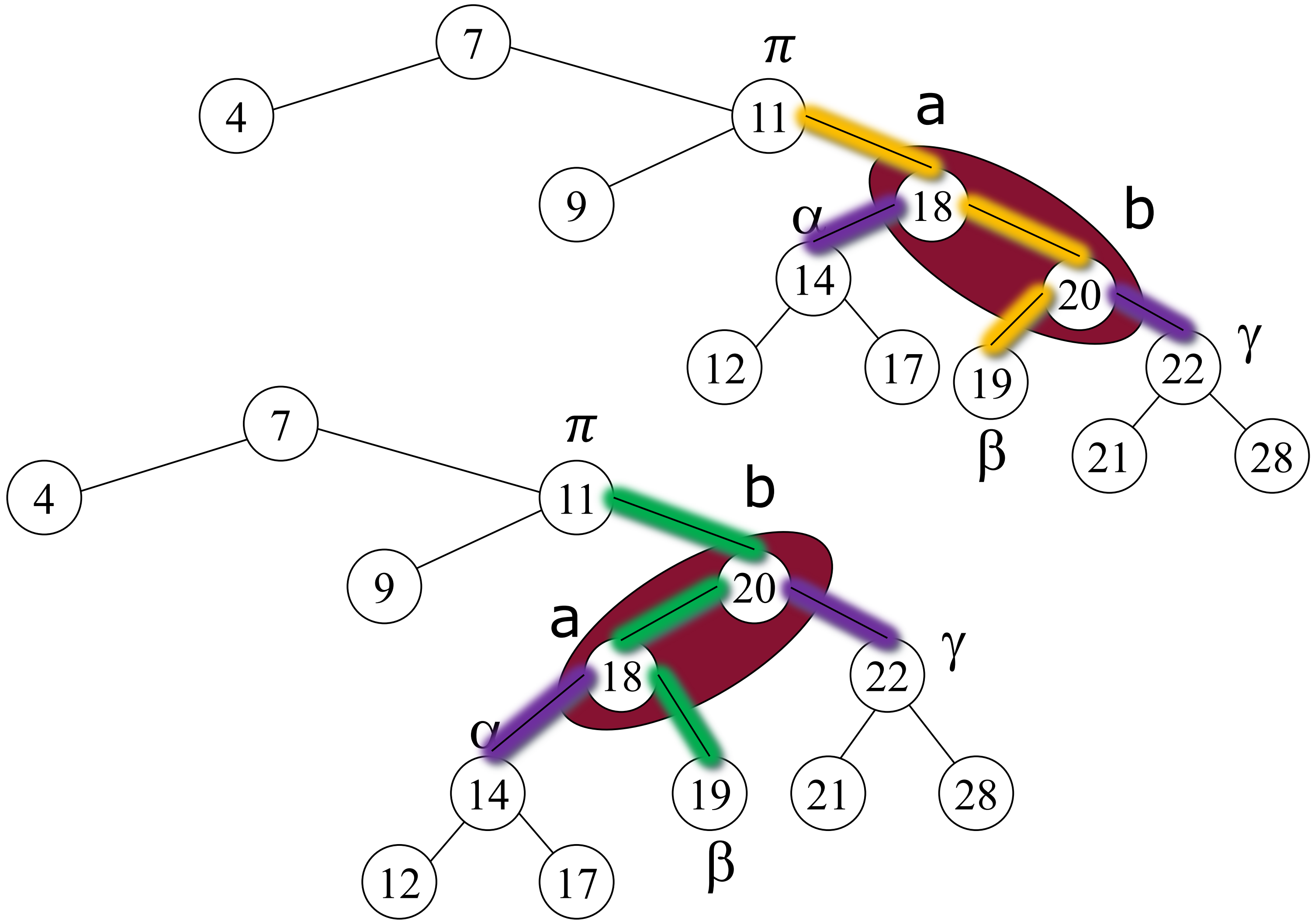
Taking the logarithm, we get the lemma.

Ira A. Fulton Schools of
Engineering
Arizona State University

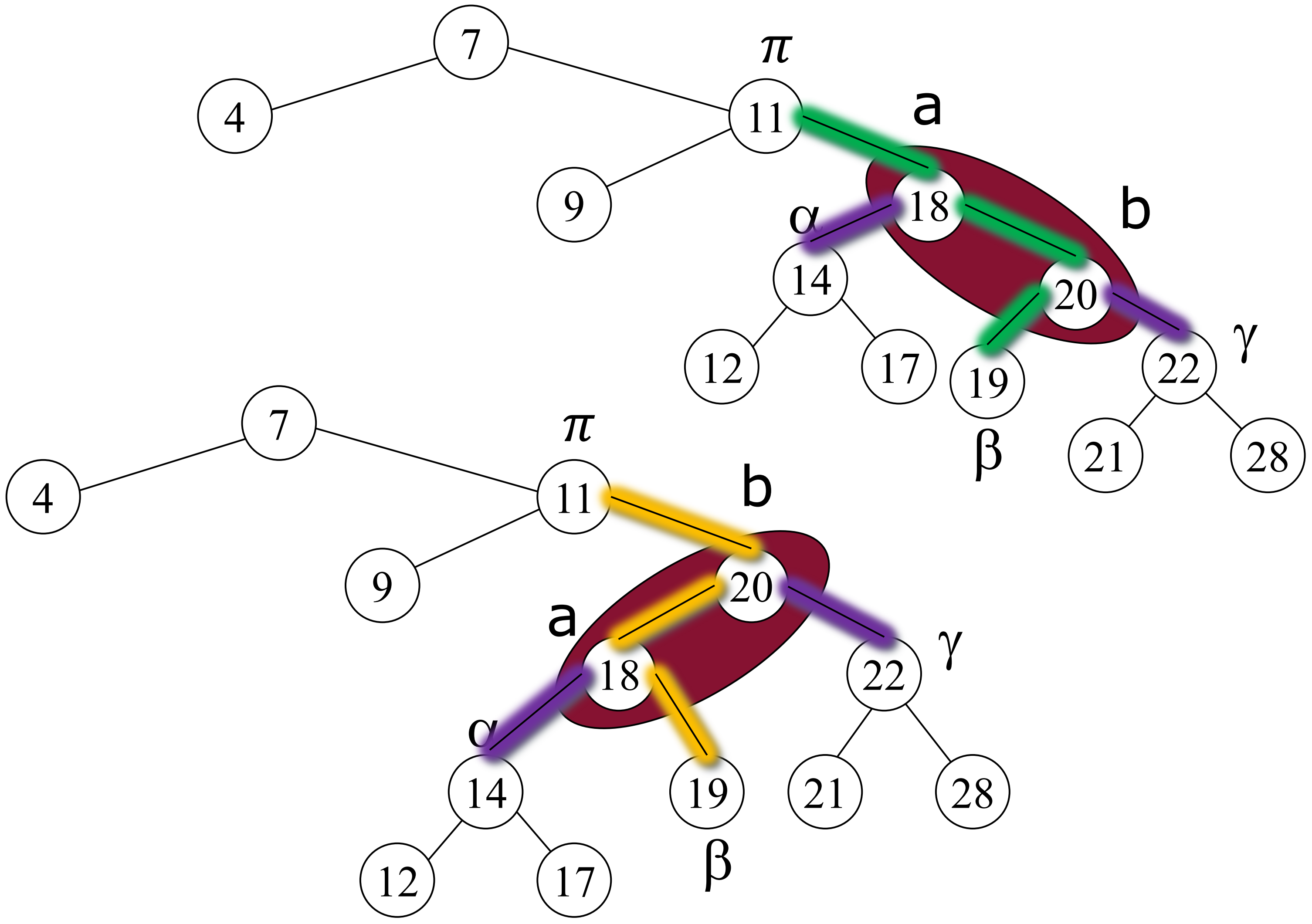
Red Black Trees, Part 2

- Definition and Properties of Red-Black Trees
- Rotations, RBT-Insert Preparation
- RBT-Insert-Fixup (3 cases)
- Examples, Running time of RBT-Insertion
- RBT-Deletion Preparation
- RBT-Deletion-Fixup (4 cases)
- Examples and Summary

Left Rotation: identify the backbone
$a \neq \text{nil}, b \neq \text{nil}$



Right Rotation: identify the backbone $a \neq \mathbf{nil}, b \neq \mathbf{nil}$



Key Things To Remember When Doing Rotations

1. **Identifying the backbone, then $\pi, \alpha, \beta, \gamma$.**
2. **Rotate the backbone**
3. **update up to 6 links highlighted in gold→green.**
4. **If the two nodes on the backbone have different colors, let the colors remain at its vertical levels:**

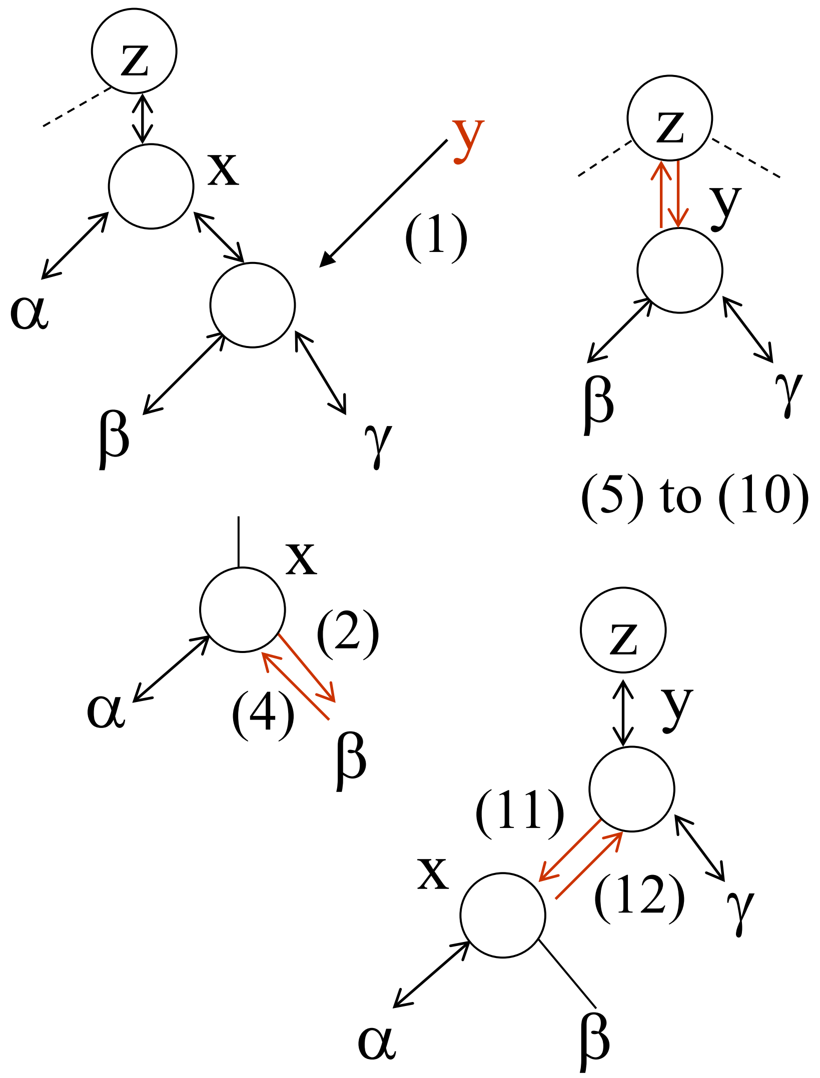
If before the rotation, the upper node is red (black) and the lower node is black (red), then after the rotation, the upper node is still red (black) and the lower node is still black (red). Note that this requires recoloring the nodes.

Key Things To Remember When Doing Rotations

1. **Nodes (a and b) on the backbone should not be nil.**
2. **Node π may be nil.**
3. **Anyone of alpha, beta, and gamma may be nil.**
4. **Update pointers at nodes a and b.**
5. **Update pointers at π and β, if not nil.**

Red-Black Tree Rotation Algorithms

```
Left-rotate (T, x)
1.   y := x.right
2.   x.right := y.left
3.   if y.left ≠ nil
4.       then y.left.parent := x
5.   y.parent = x.parent
6.   if x.parent = nil
7.       then root[T] := y
8.       else if x = x.parent.left
9.           then x.parent.left := y
10.          else x.parent.right := y
11.  y.left := x
12.  x.parent := y
```



Insertion Into A Red-Black Tree

1. We first color the node to be inserted to be red, and insert it as if performing insertion in a binary search tree.

2. Is the resulting tree still a red-black tree? Maybe not…

3. Which of the five properties may be violated?

   A. Property 1: No

   B. Property 2: Yes, if we are inserting to an originally empty tree

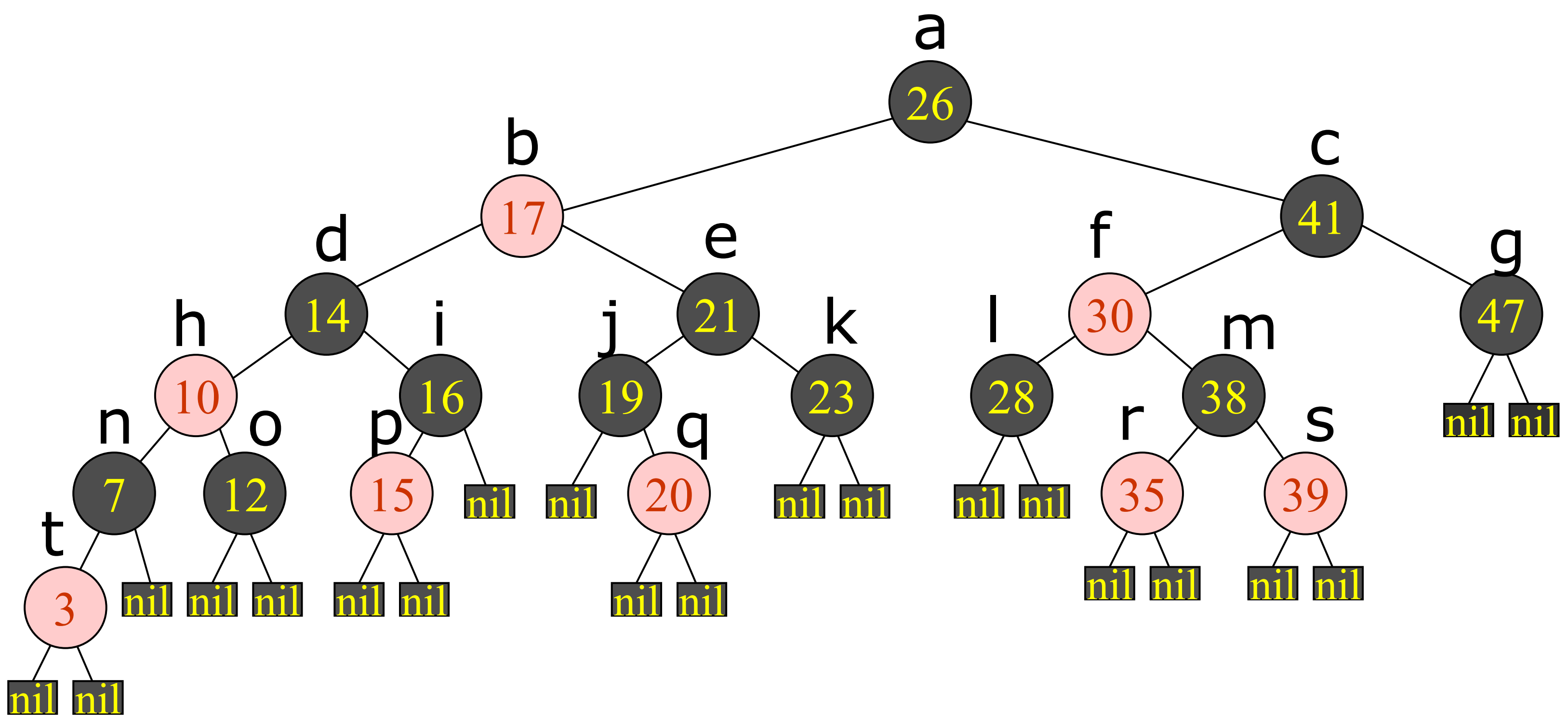
   C. Property 3: No

   D. Property 4: Yes, if the new node becomes a child of a red node

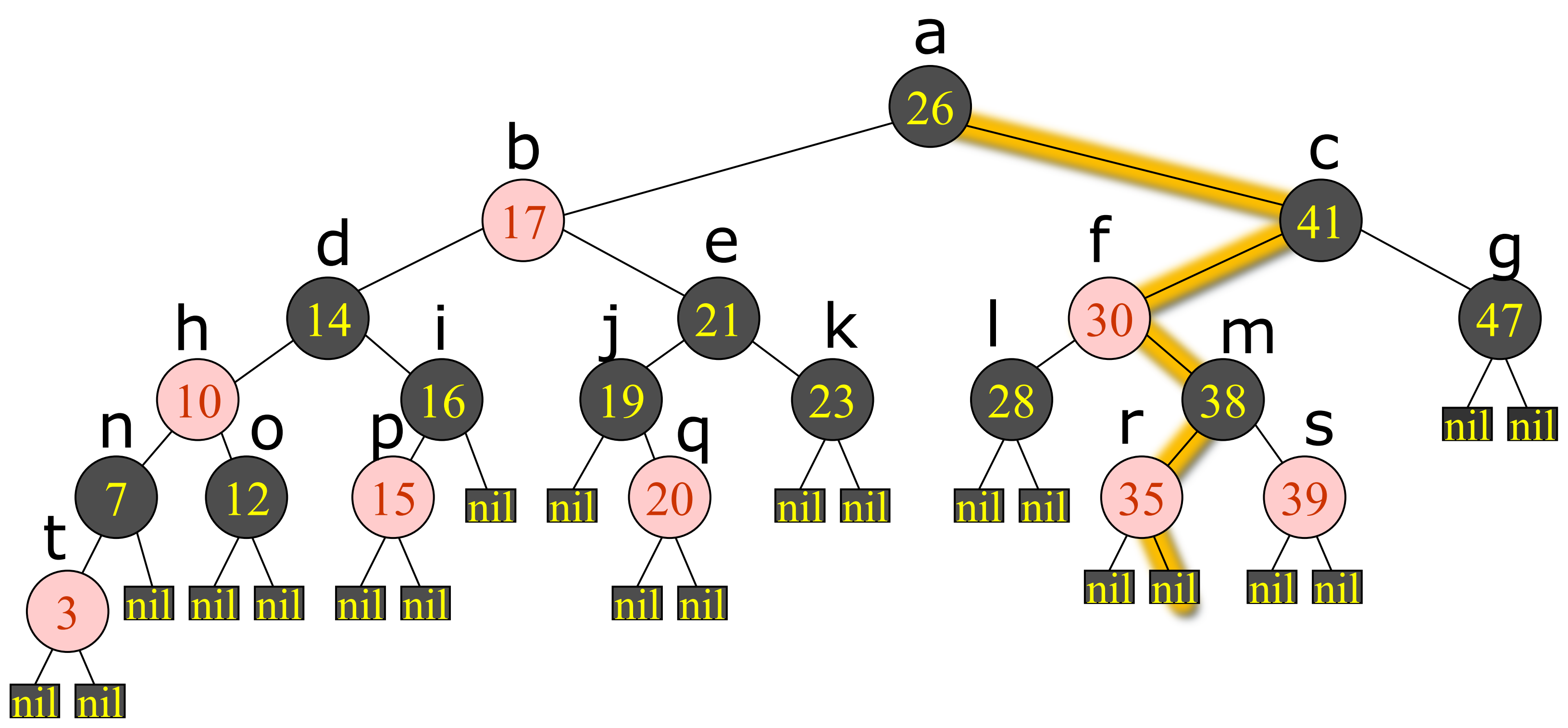
   E. Property 5: No, since we are inserting a red node

4. We can check whether there is a violation in O(1) time. If there is a violation, we need to fix it. Property 2 violation can be fixed by changing the root node to black. Next, we study the case where Property 4 is violated.

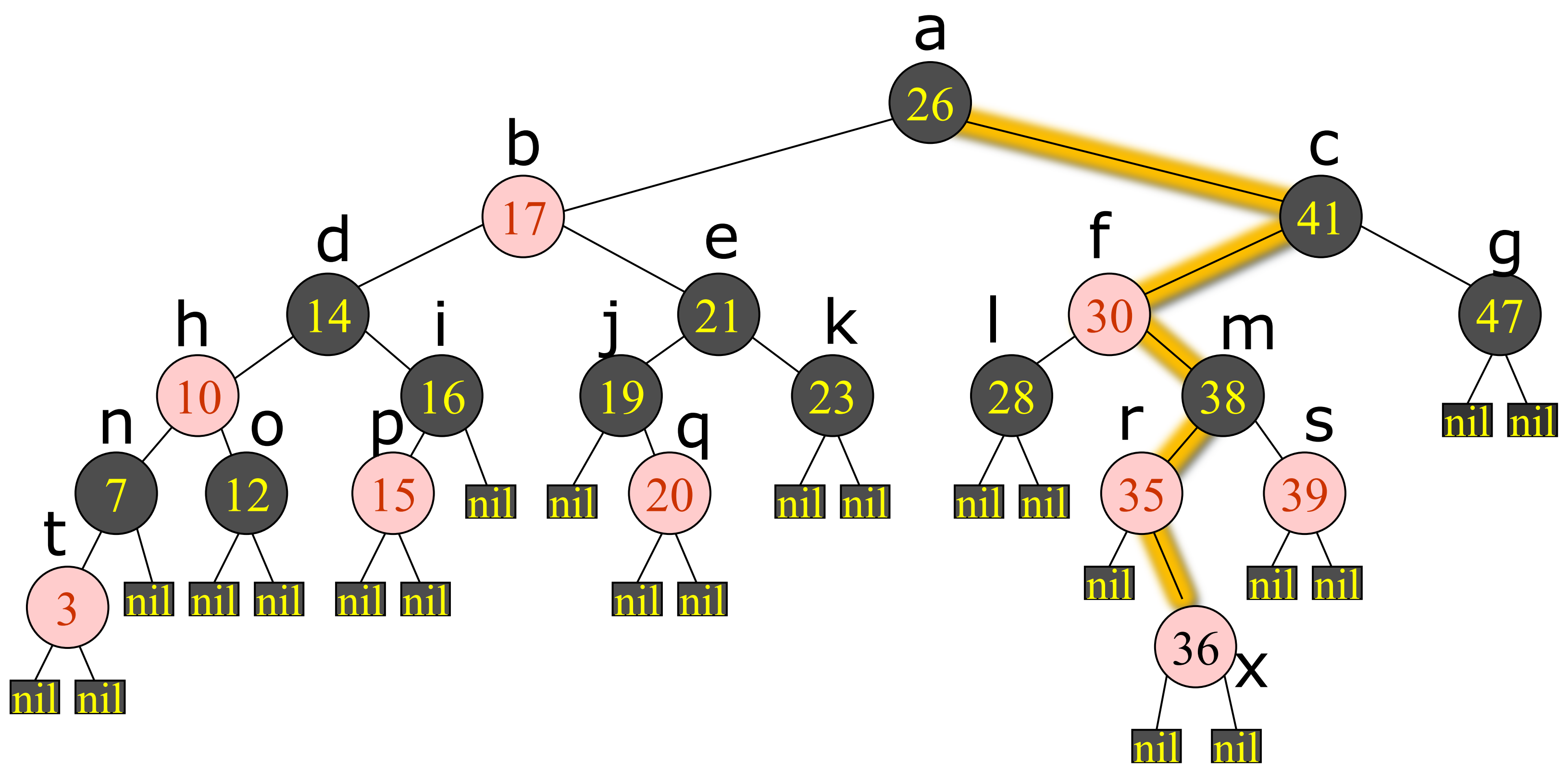
Example: Inserting 36 Violates Property 4



Example: Inserting 36 Violates Property 4



Example: Inserting 36 Violates Property 4



Ira A. Fulton Schools of Engineering

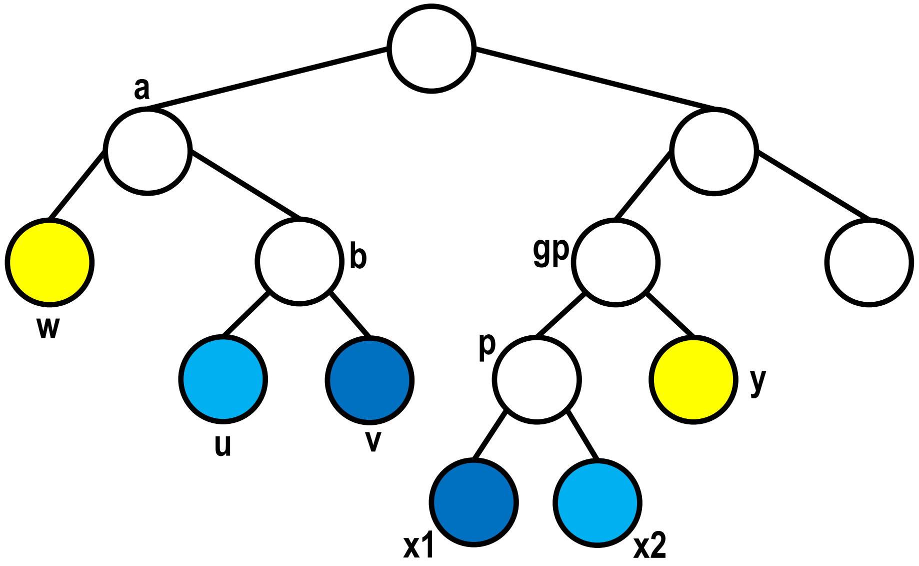
Arizona State University

Red Black Trees, Part 3

- Definition and Properties of Red-Black Trees
- Rotations, RBT-Insert Preparation
- RBT-Insert-Fixup (3 cases)
- Examples, Running time of RBT-Insertion
- RBT-Deletion Preparation
- RBT-Deletion-Fixup (4 cases)
- Examples and Summary

Some Handy Notations

- parent, sibling, grandparent, uncle/aunt, near, far
- u's parent is b;
- u and v are siblings;
- u's grandparent is a;
- w is the uncle/aunt of u and v.
- w is near u, but far from v.
- y is close to x2, but far from x1.



Insertion Into A Red-Black Tree

1. **Let x points to the newly inserted (red) node.**
2. **Since Property 4 is violated, we conclude that**
   a) x is not the root (otherwise we are violating property 2)
   b) the parent of x is red (otherwise we are not violating property 4)
   c) the grandparent of x is black (otherwise property 4 is violated before insertion)
3. **The only violation is a pair of consecutive red nodes: x and its parent.**
4. **The goal is to move the violation up the tree, until it disappears or transform the violation to Property 2.**

Insertion Into A Red-Black Tree

The goal is to move the violation up the tree, until it disappears or reaches the root.

1. If the uncle of x is red (grandparent must be black): We change x's parent and uncle to black and its grandparent to red. Repointing x to its grandpa. Move the violation up!

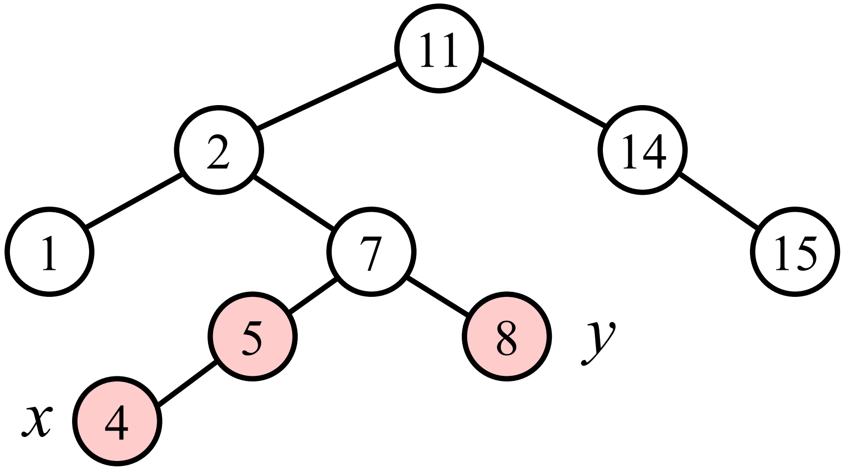
In the rest, we assume that the uncle of x is black.

2. If x is near its uncle, use x and its parent as the backbone to perform a rotation, repointing x to the lower of the two consecutive red nodes after the rotation.

3. If x is far from its uncle, use the parent and the grandparent as the backbone to perform a rotation. Now the violation is removed, because the uncle is black

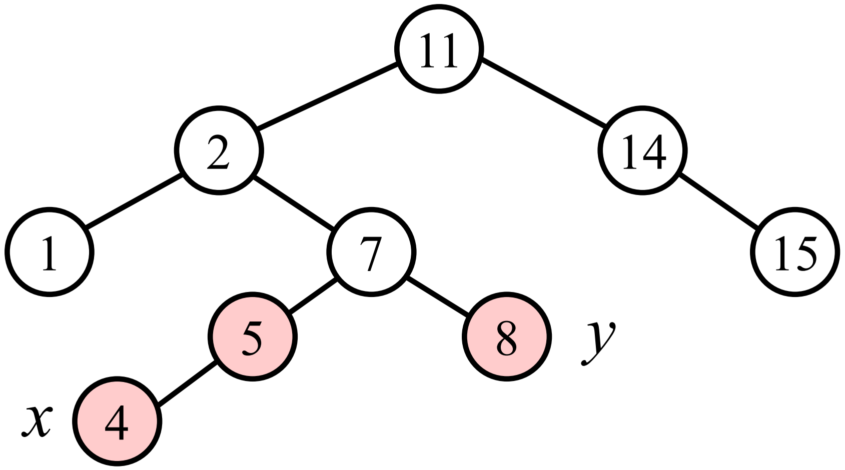
Insertion Fixup Case 1

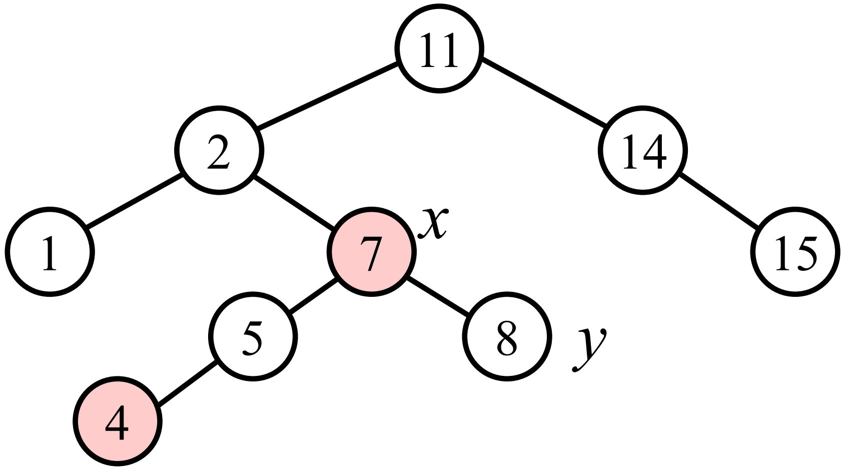
- Case 1: x is red, x's parent is red, x's uncle is red.
- Action: change (blk) grandparent of x to red; change (red) parent and uncle of x to black; re-position x to parent(parent(x)).
- Result: either the violation is resolved, or x is moved up two hops.



Insertion Fixup Case 1

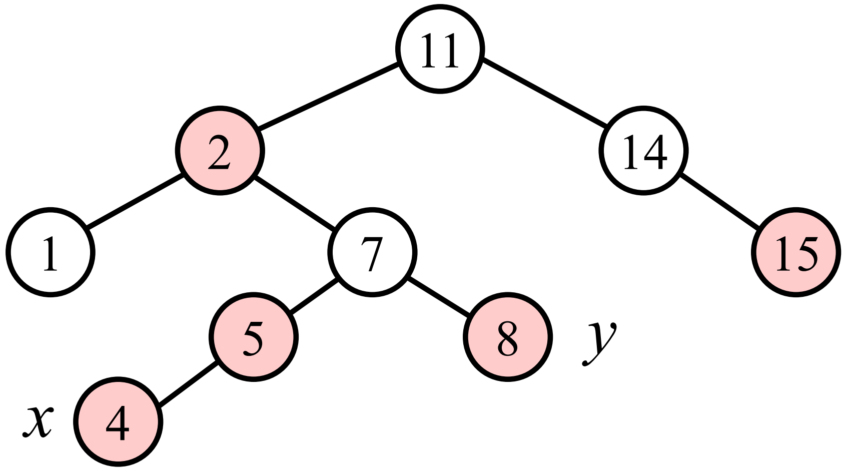
- **Case 1: x is red, x's parent is red, x's uncle is red.**
- **Action: change (blk) grandparent of x to red; change (red) parent and uncle of x to black; re-position x to parent(parent(x)).**
- **Result: either the violation is resolved, or x is moved up two hops.**





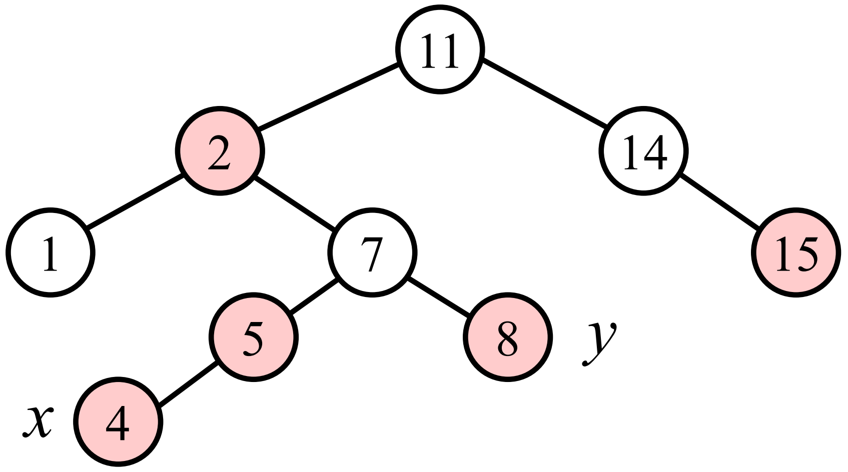
Insertion Fixup Case 1

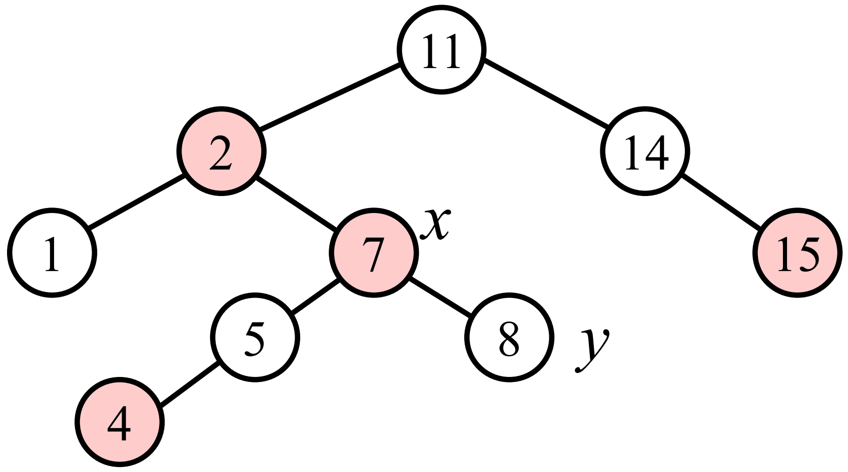
- **Case 1: x is red, x’s parent is red, x’s uncle is red.**
- **Action: change (blk) grandparent of x to red; change (red) parent and uncle of x to black; re-position x to parent(parent(x)).**
- **Result: either the violation is resolved, or x is moved up two hops.**



Insertion Fixup Case 1

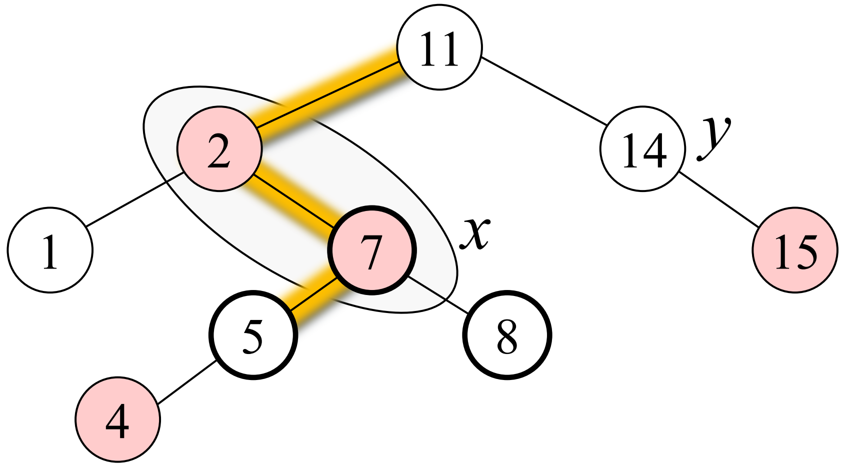
- Case 1: x is red, x's parent is red, x's uncle is red.
- Action: change (blk) grandparent of x to red; change (red) parent and uncle of x to black; re-position x to parent(parent(x)).
- Result: either the violation is resolved, or x is moved up two hops.





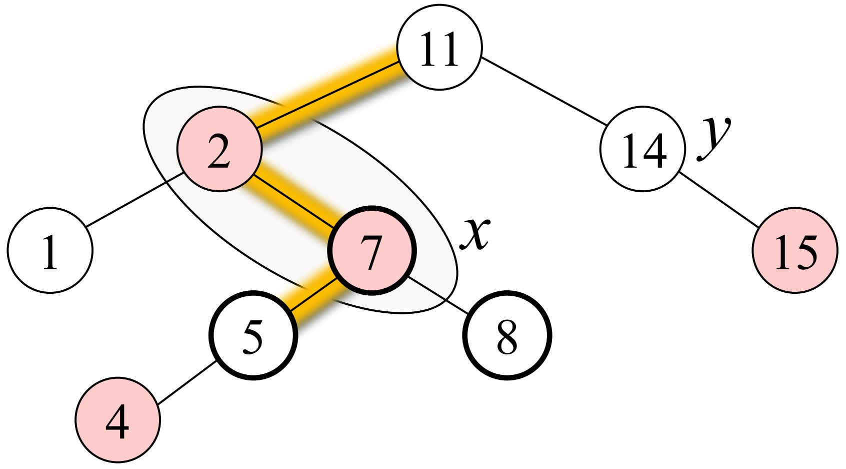
Insertion Fixup Case 2

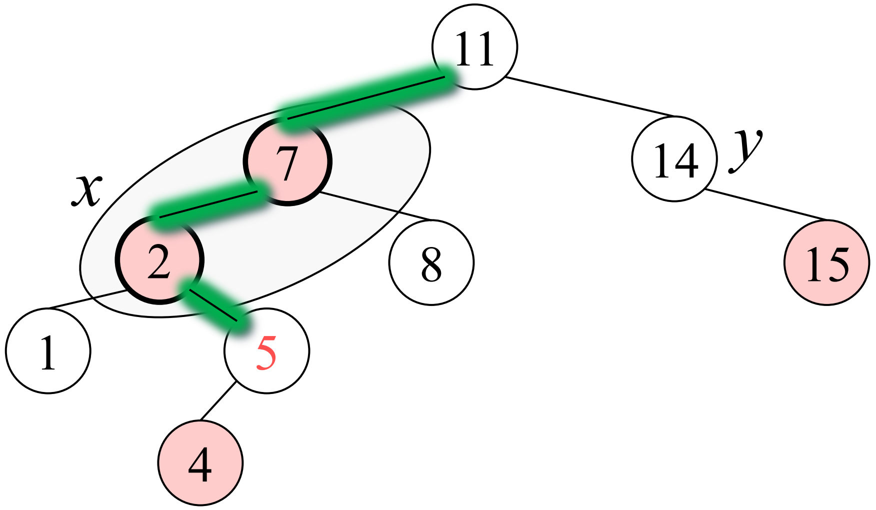
- **Case 2: x is red, x's parent is red, x's uncle is blk, x is near its uncle.**
- **Action: Use x and its parent as the backbone to perform a rotation. re-position x to the lower red node.**
- **Result: transform to case 3.**



Insertion Fixup Case 2

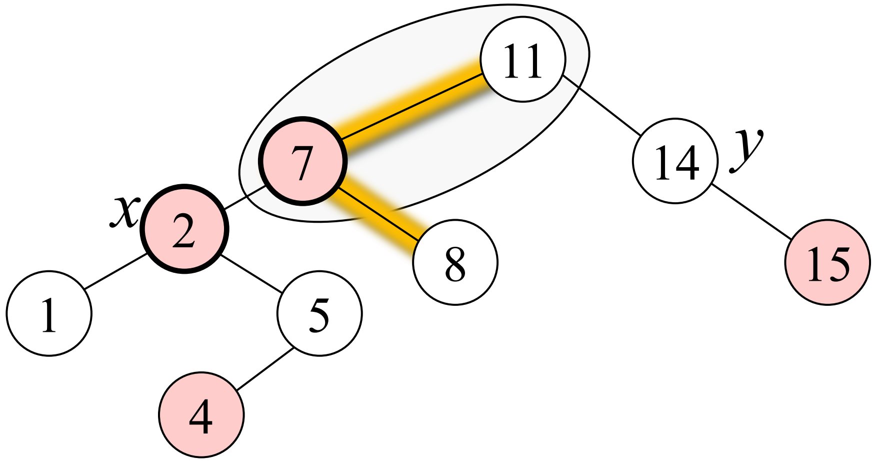
- **Case 2: x is red, x's parent is red, x's uncle is blk, x is near its uncle.**
- **Action: Use x and its parent as the backbone to perform a rotation. re-position x to the lower red node.**
- **Result: transform to case 3.**





Insertion Fixup Case 3

- **Case 3: x is red, x's parent is red, x's uncle is blk, x is far from its uncle.**
- **Action: Use x's parent and grandparent as the backbone to perform a rotation.**
- **Result: violation resolved.**

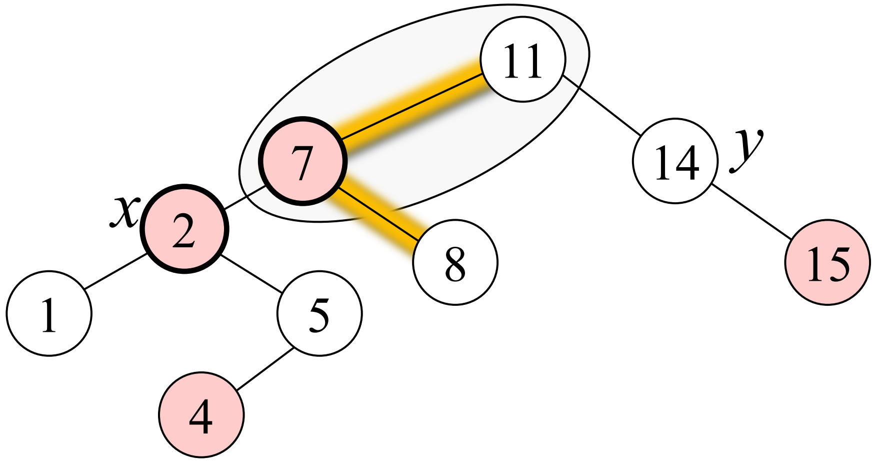


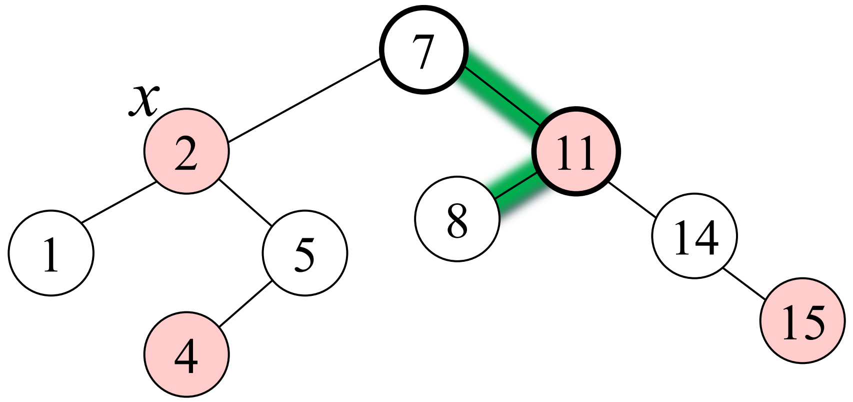
Insertion Fixup Case 3

- Case 3: x is red, x’s parent is red, x’s uncle is blk, x is far from its uncle.

- Action: Use x’s parent and grandparent as the backbone to perform a rotation.

- Result: violation resolved.





Ira A. Fulton Schools of Engineering

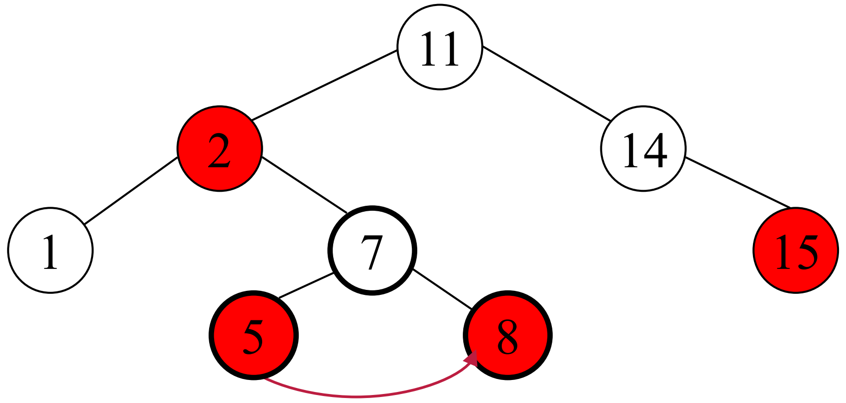
Arizona State University

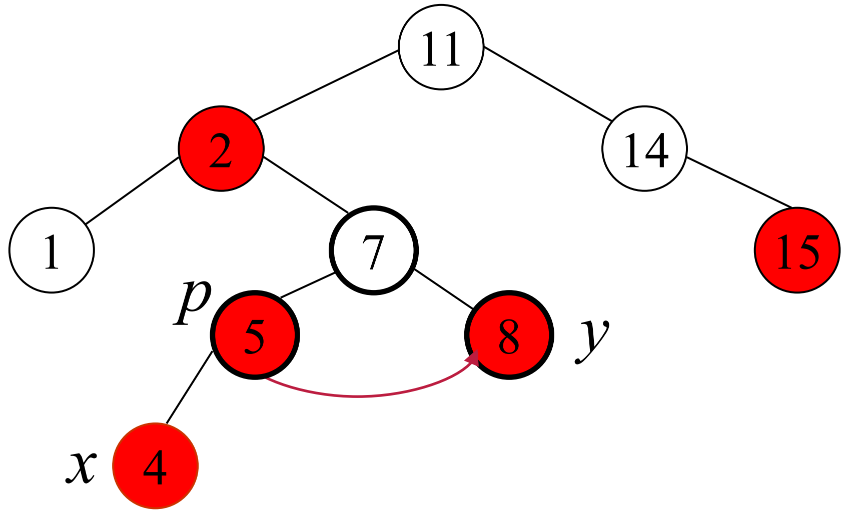
Red Black Trees, Part 4

- Definition and Properties of Red-Black Trees
- Rotations, RBT-Insert Preparation
- RBT-Insert-Fixup (3 cases)
- Examples, Running time of RBT-Insertion
- RBT-Deletion Preparation
- RBT-Deletion-Fixup (4 cases)
- Examples and Summary

RB-Tree Insertion Example

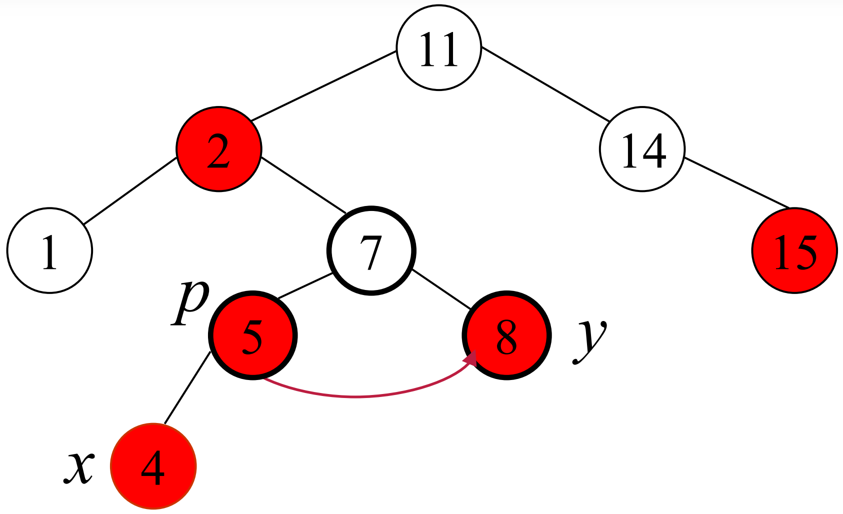
- We have an RB-tree on the right.
- Insert 4.
- After inserting 4, before insertion fixup, we have the situation on the lower-right.
- Property 4 is violated.
- x is red, x's parent is red, x's uncle is red.
- Case 1.

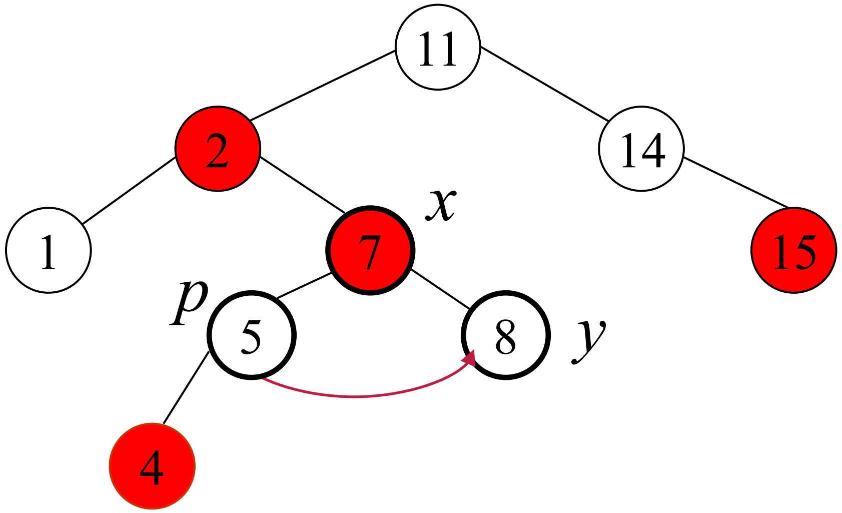




RB-Tree Insertion Example

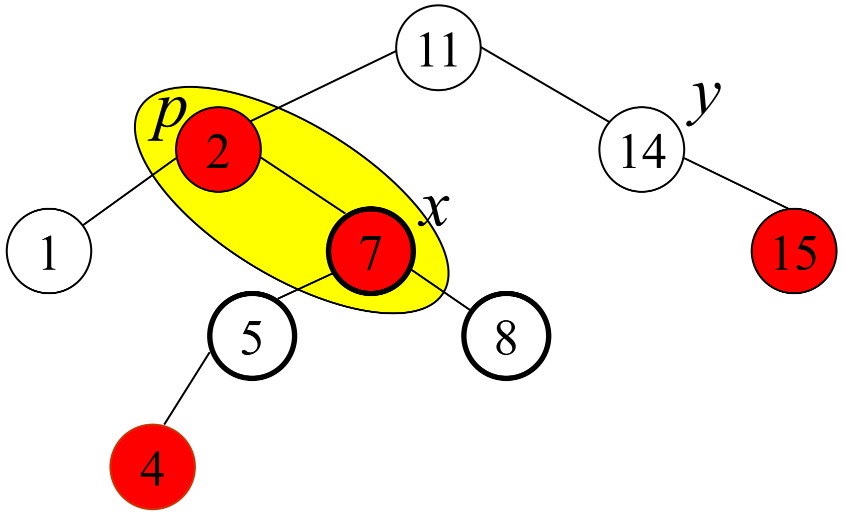
- x is red, x's parent is red, x's uncle is red.
- The grandpa of x must be black.
- Action: change color of x's grandpa from black to red; change color of x's parent and uncle from red to black. Reposition x to its grandpa.
- Result: the problem is moved up two levels.

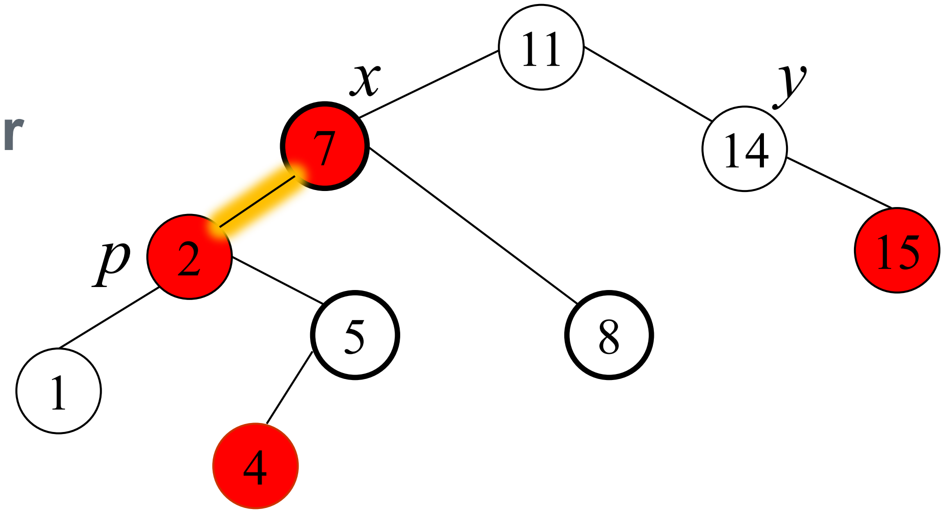




RB-Tree Insertion Example

- x is red, x's parent is red, x's uncle is blk. x is close to its uncle.
- This is Case 2.
- Action: Rotate using (x, p) as the backbone.
- Result: p is red, p's parent is red, p's uncle is black. p is far from its uncle.
- This is Case 3.

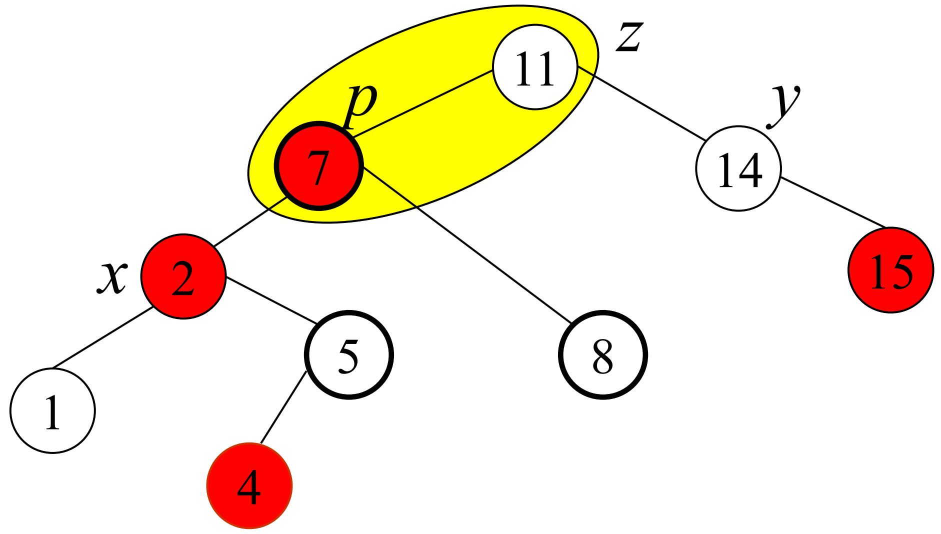


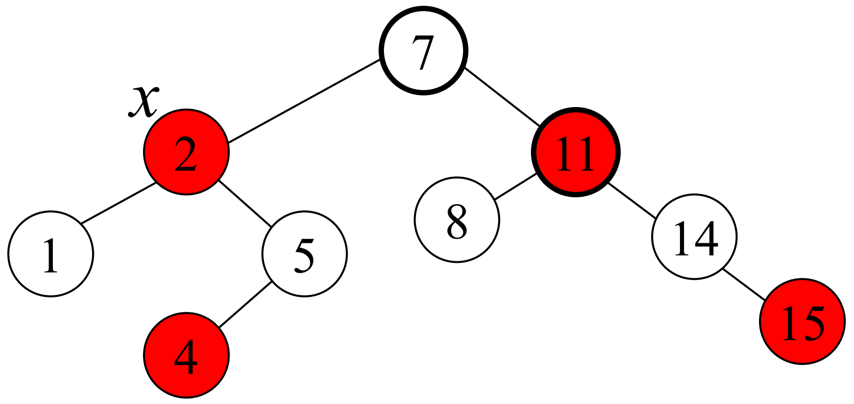


RB-Tree Insertion Example

Relabel:

- x is red, x's parent is red, x's uncle is black. x is far from its uncle.
- This is Case 3.
- Action: Use (p, z) as backbone for rotation.
- Result: Done.





Time Complexity for Insertion Fixup

- Case 3 is resolved in $O(1)$ time
- Case 2 is transformed to Case 3 in $O(1)$ time
- Case 1 takes $O(1)$ time to resolve or to move up two hops. Since the tree height is $O(\log n)$, it takes $O(\log n)$ time in the worst case.

Red-Black Tree Insertion Algorithms

```
RB-Tree-Insert (T, x)
1.   Tree-Insert(T, x)      // Use binary search insert, insert a leaf.
2.   x.color := Red
3.   while x.parent and x.parent.color = Red do  // adjacent red nodes
4.       if x.parent = x.parent.parent.left
5.       then y := x.parent.parent.right
6.           if y.color = Red
7.           then  x.parent.color := Black
8.                 y.color := Black                        case 1
9.                 x.parent.parent.color :=Red
10.                x = x.parent.parent
11.          else  if x = x.parent.right
12.          then  x = x.parent                            case 2
13.                Left-Rotate(T, x)
14.          x.parent.color := black
15.          x.parent.parent.color := Red                  case 3
16.          Right-Rotate(T, x.parent.parent)
17.      else  (symmetric to the then-clause
                      with "right" and "left" exchanged.
18.  Root[T].color := Black
```

Ira A. Fulton Schools of Engineering

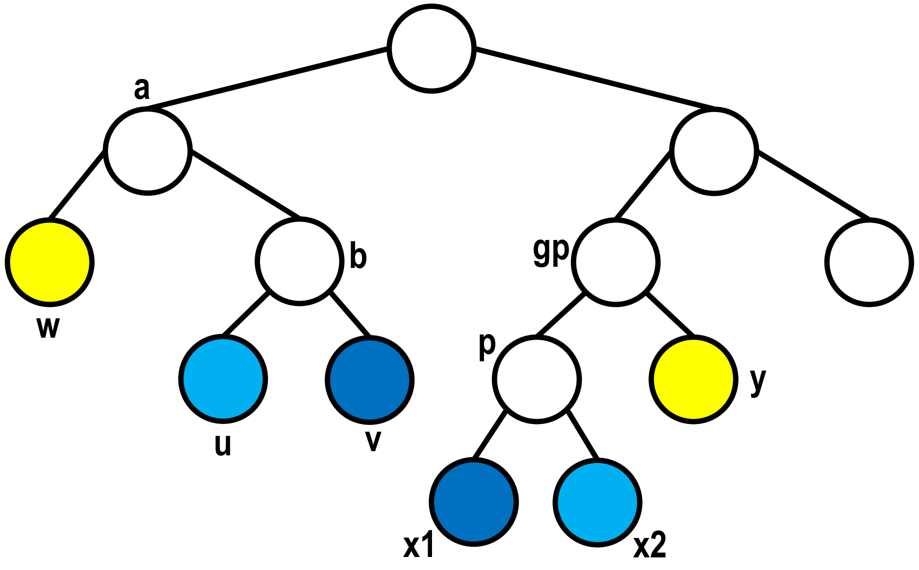
Arizona State University

Red Black Trees, Part 5

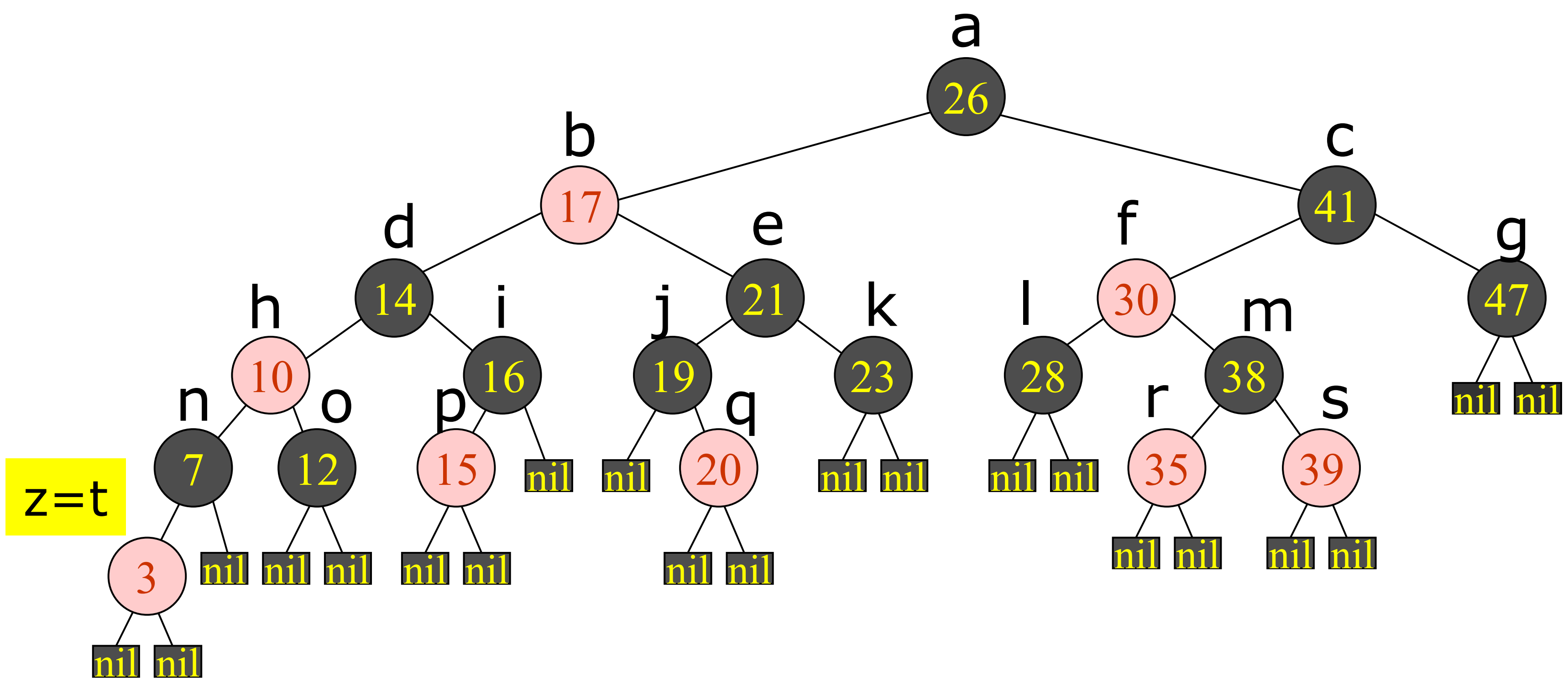
- Definition and Properties of Red-Black Trees
- Rotations, RBT-Insert Preparation
- RBT-Insert-Fixup (3 cases)
- Examples, Running time of RBT-Insertion
- RBT-Deletion Preparation
- RBT-Deletion-Fixup (4 cases)
- Examples and Summary

Relationship between Tree Nodes

- u’s parent is b;
- u and v are siblings;
- u’s grandparent is a;
- w is the uncle/aunt of u and v.
- w is close to u;
- w is far from v.
- For w, NearChild[b] is u.
- For w, FarChild[b] is v.
- For y, NearChild[p] is x2.
- For y, FarChild[p] is x1.



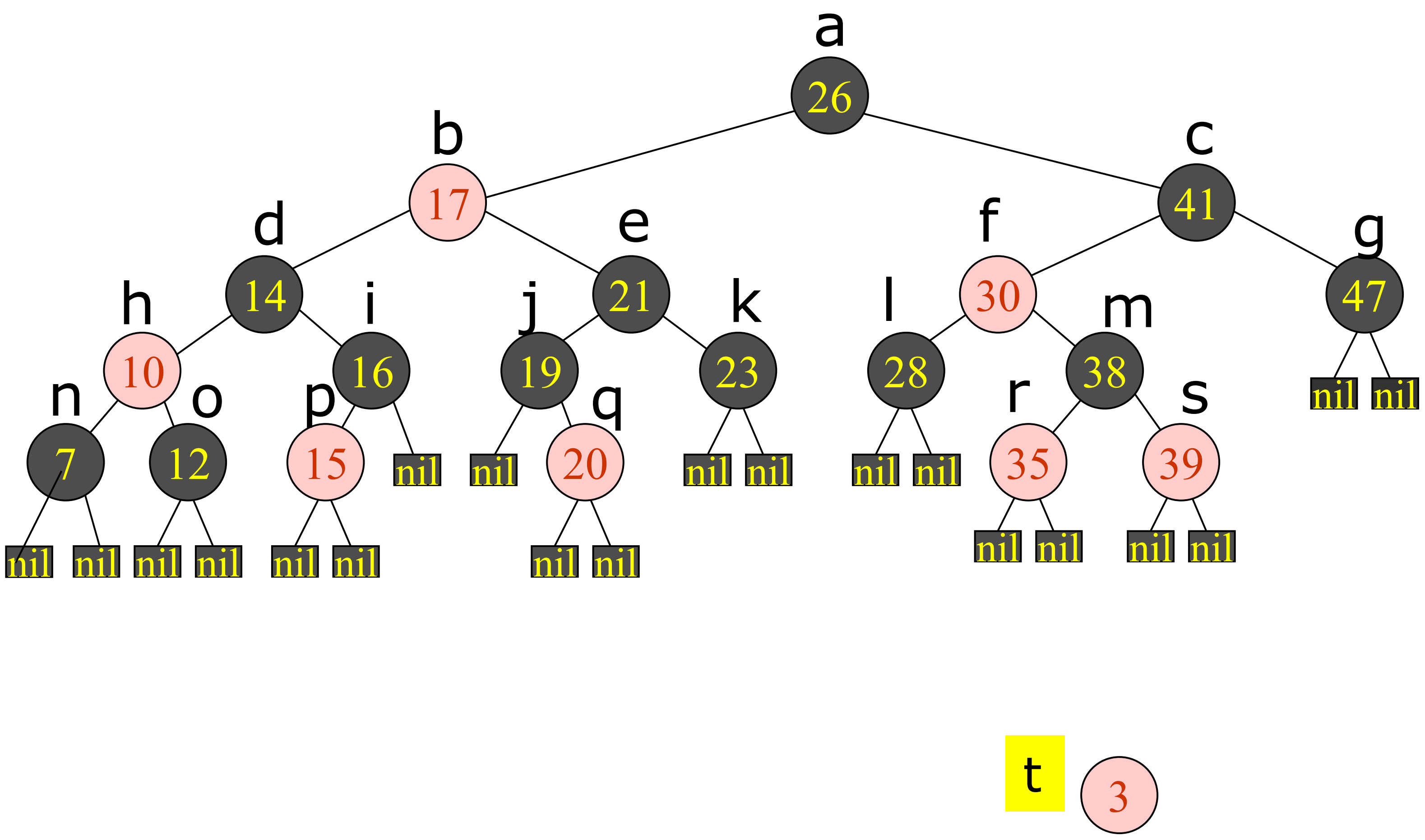
Deleting Node $z = t$ (with key=3): Step 1 of 2



Case-1 BST deletion

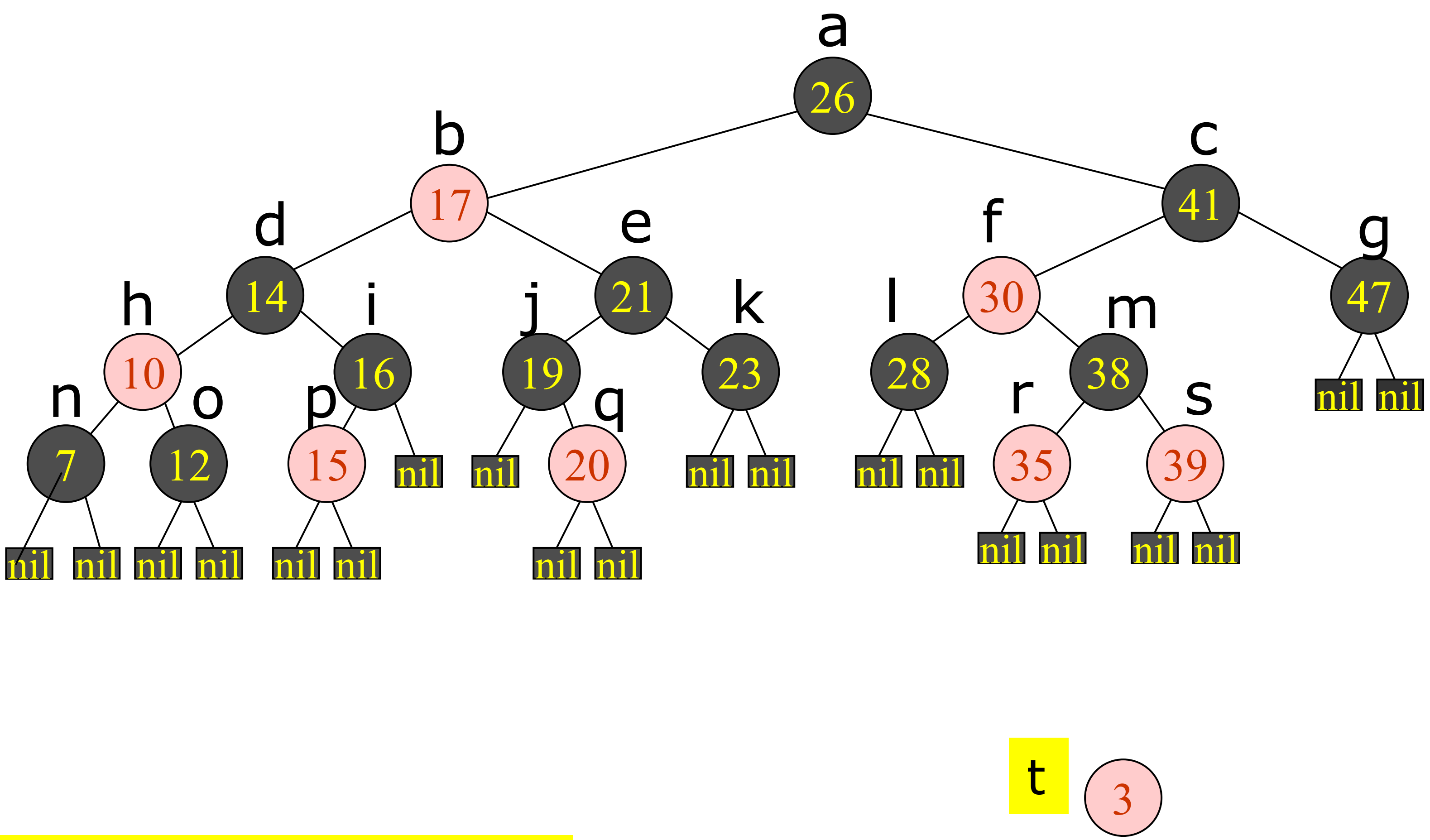
Deleting node t directly

Deleting Node $z = t$ (with key=3): Step 2 of 2



Deleting node t

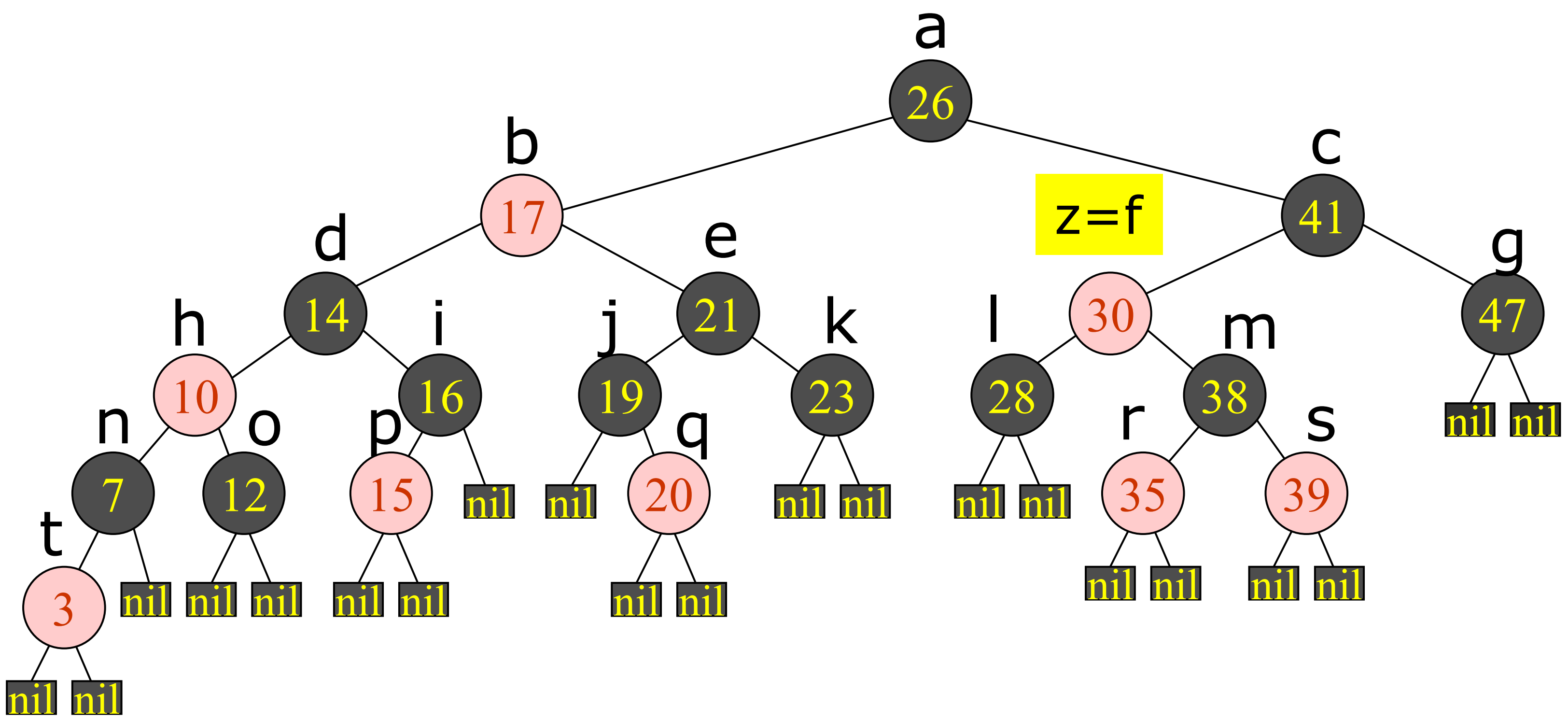
Deleting Node $z = t$ (with key=3): Result



Is the resulting tree an RB-Tree?

Yes, by luck…

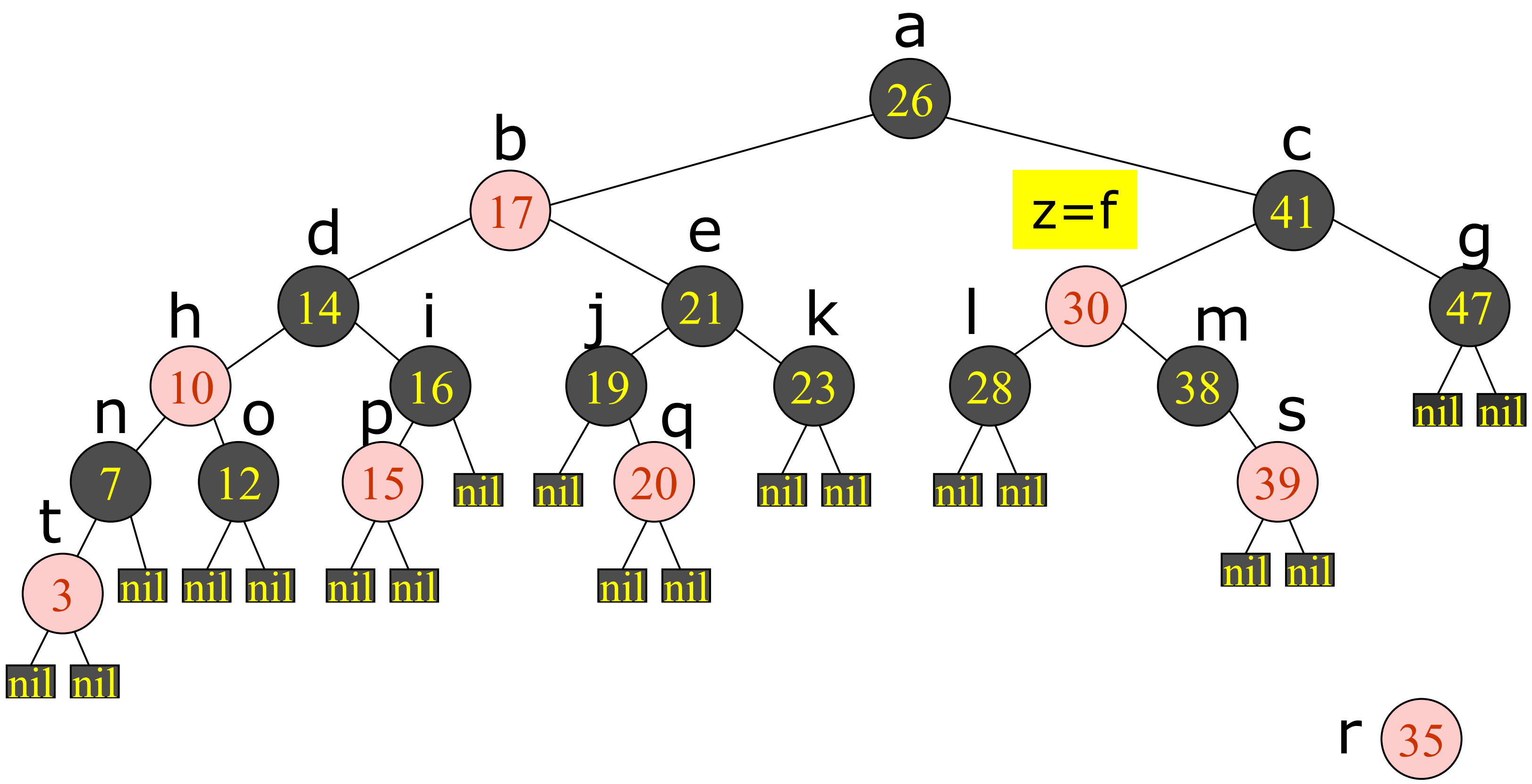
Deleting Node $z = f$ (with key=30): Step 1 of 3



Case-3 BST deletion

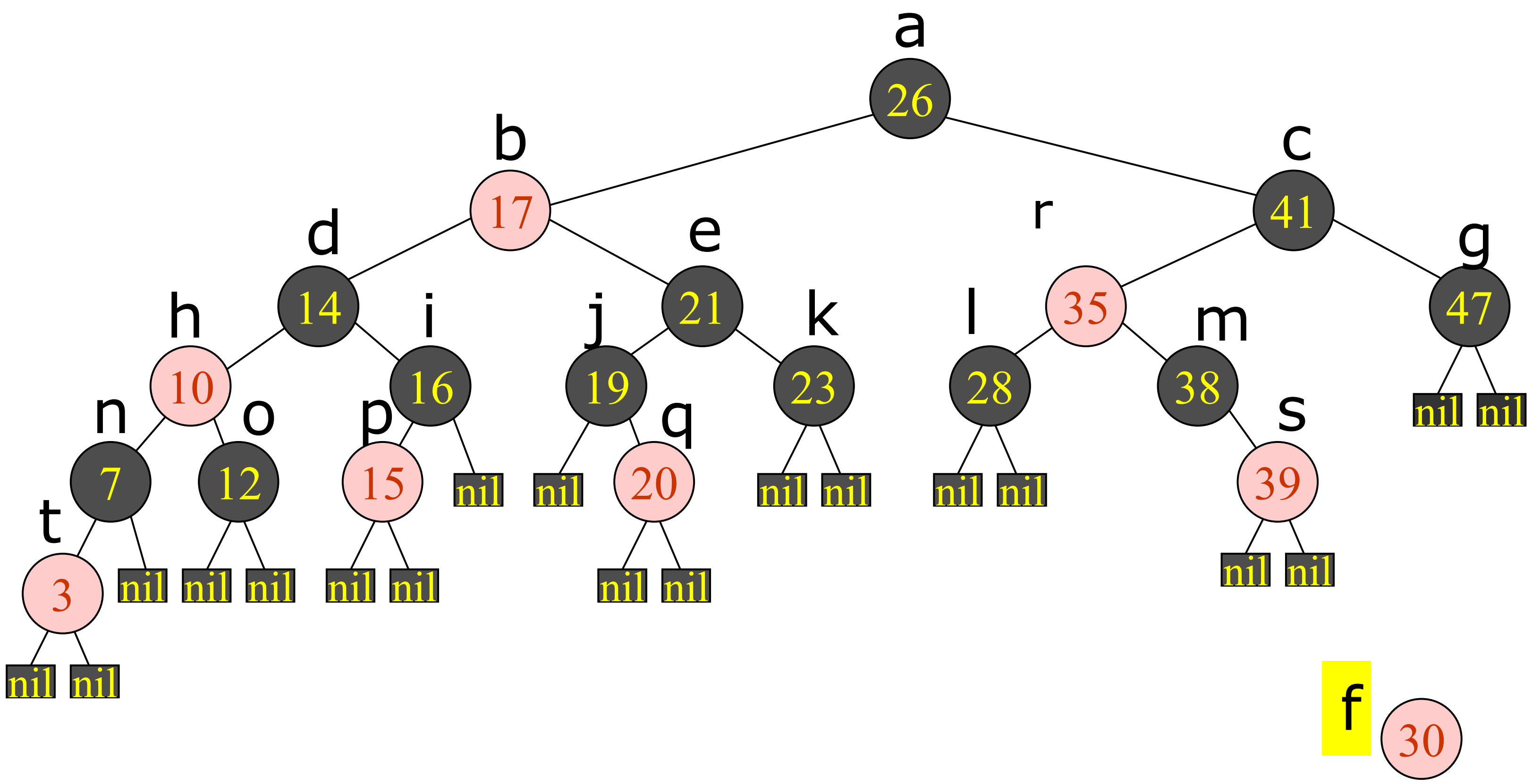
Splicing out node f's successor (r), then replacing f with r

Deleting Node $z = f$ (with key=30): Step 2 of 3



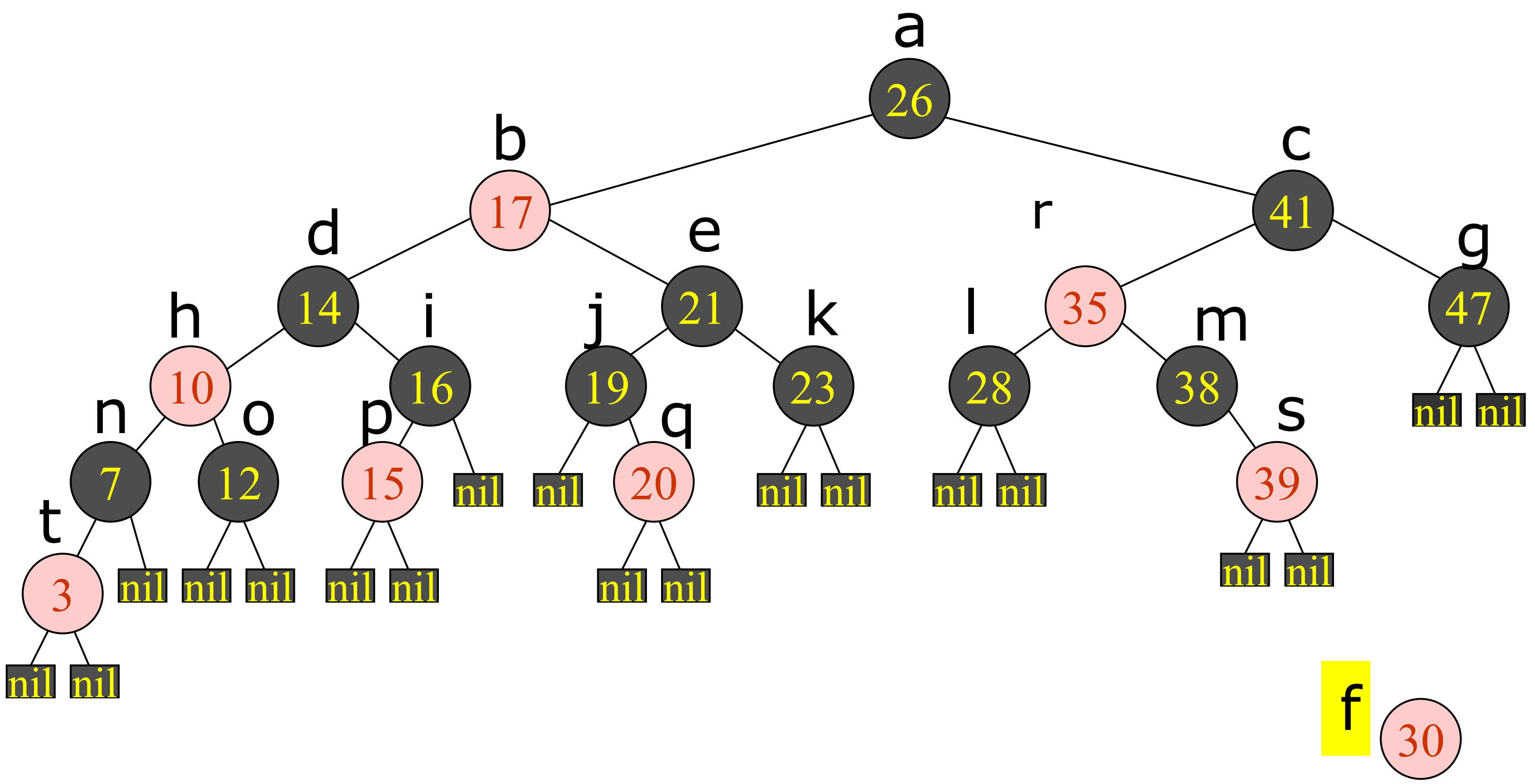
Splicing out node r

Deleting Node $z = f$ (with key=30): Step 3 of 3



Replacing node f with node r

Deleting Node $z = f$ (with key=30): Result

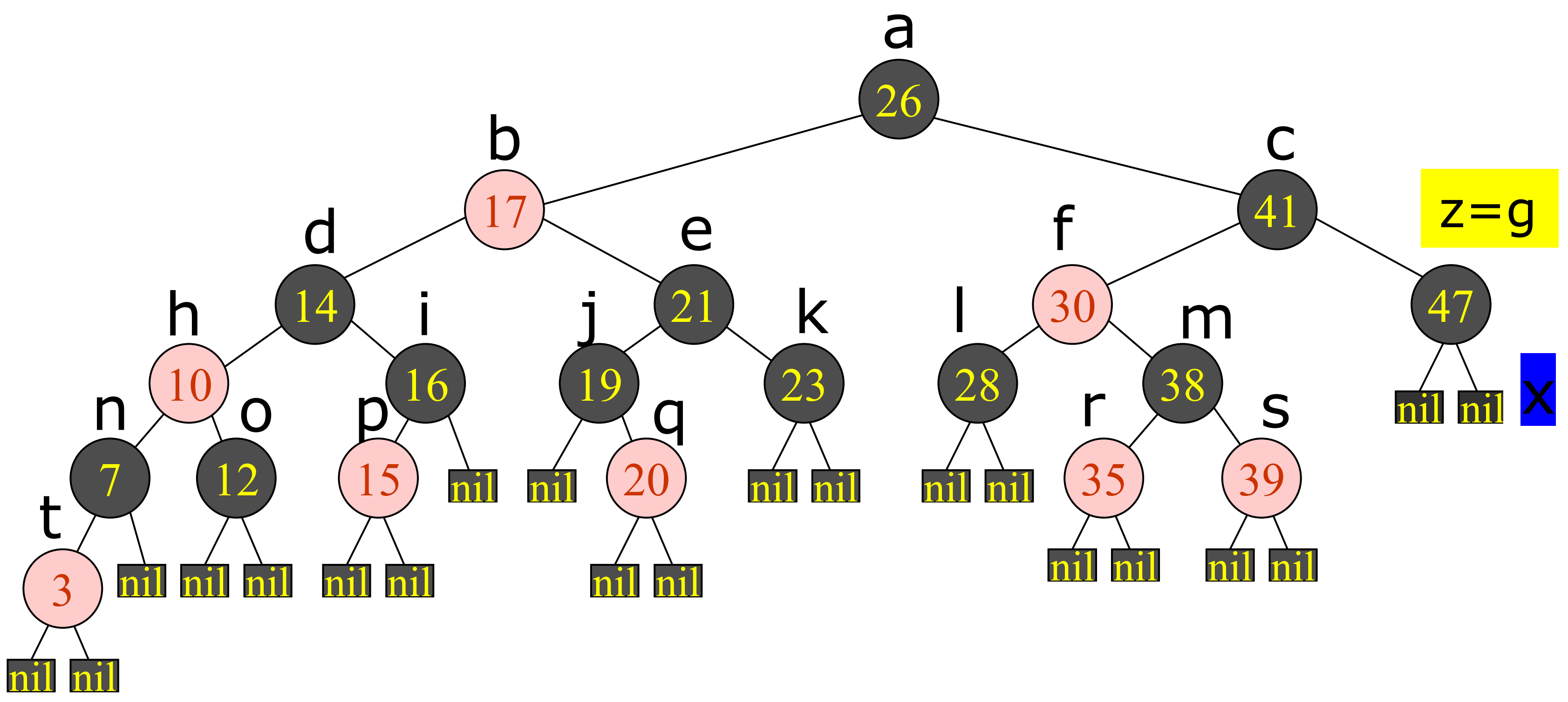


Is the resulting tree an RB-Tree?

Yes, by luck…again…

Cannot Always Rely on Luck…

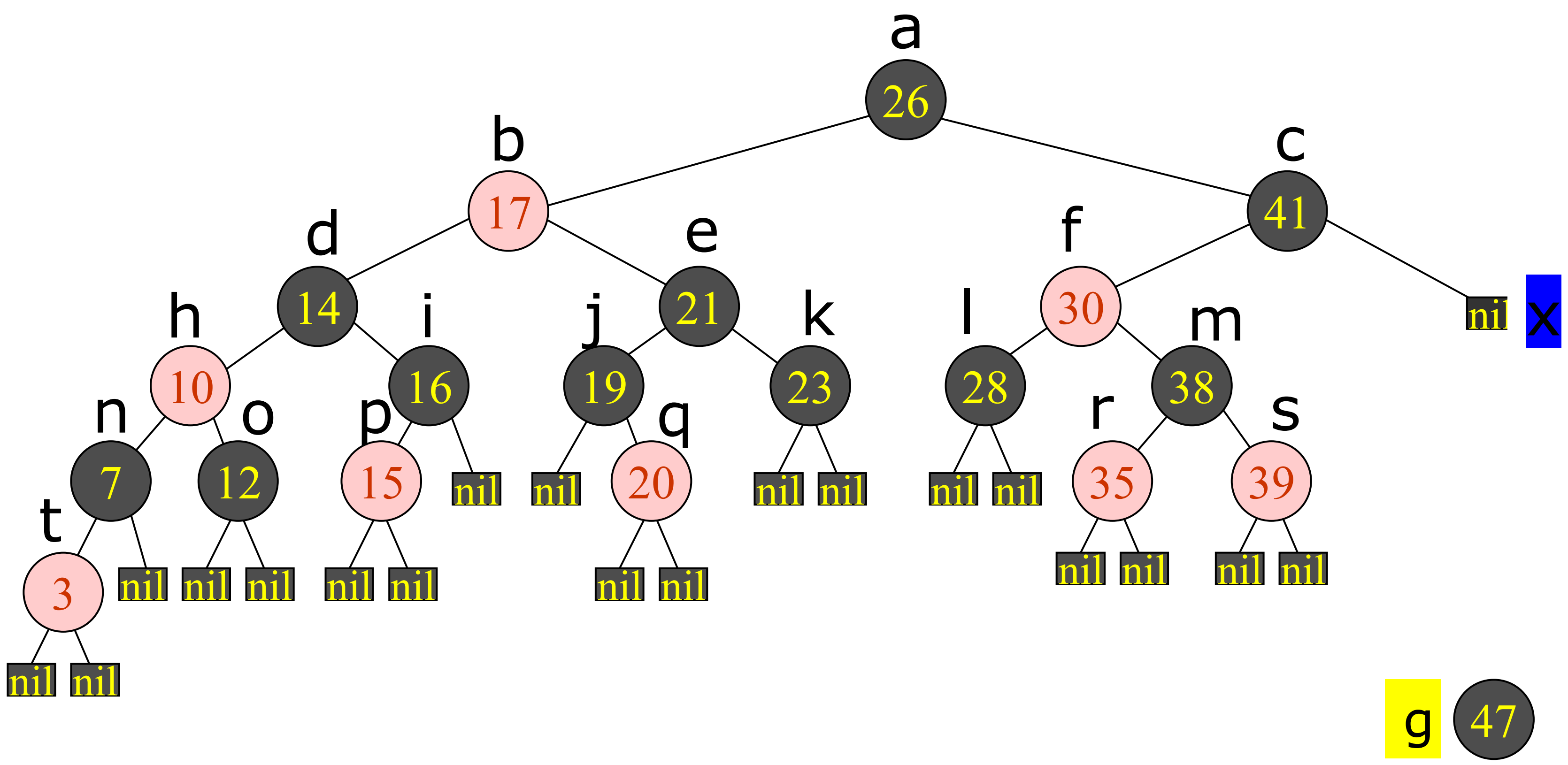
Deleting Node $z = g$ (with key=47): Step 1 of 2



Case-1 BST deletion

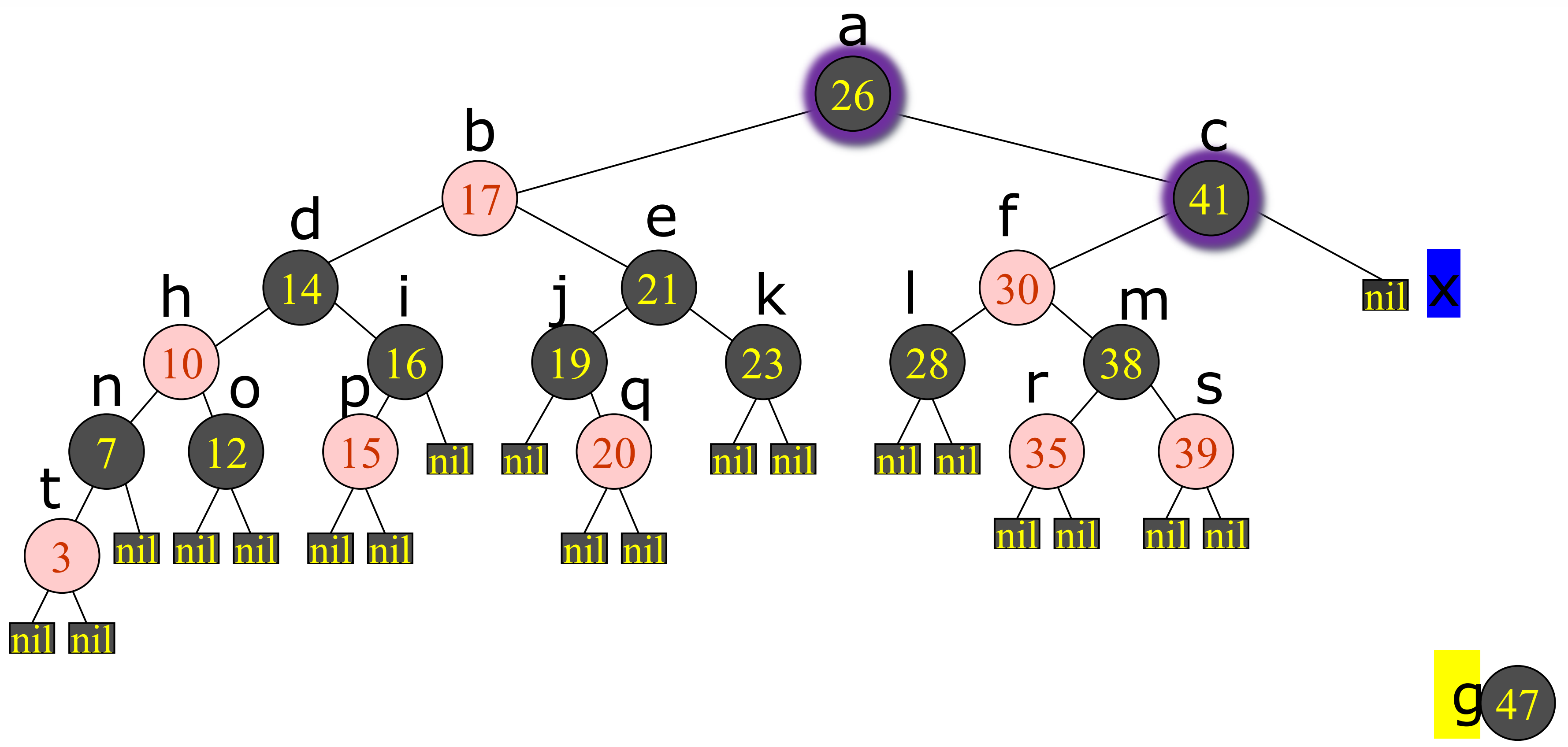
Deleting node g directly

Deleting Node $z = g$ (with key=47): Step 2 of 2



Deleting node g

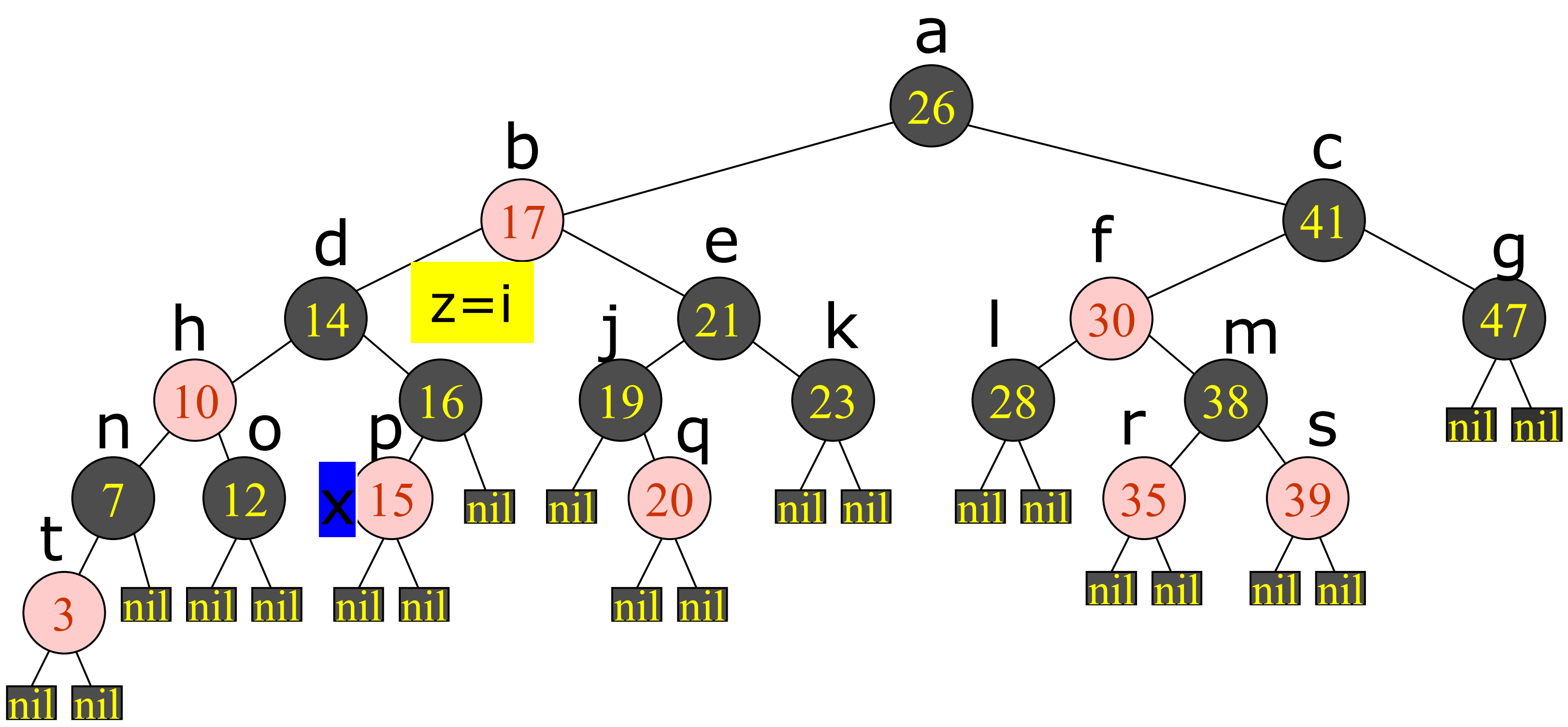
Deleting Node $z = g$ (with key=47): Result



Is the resulting tree still an RB-tree?

NO... Property 5 is violated...at all ancestors of x.

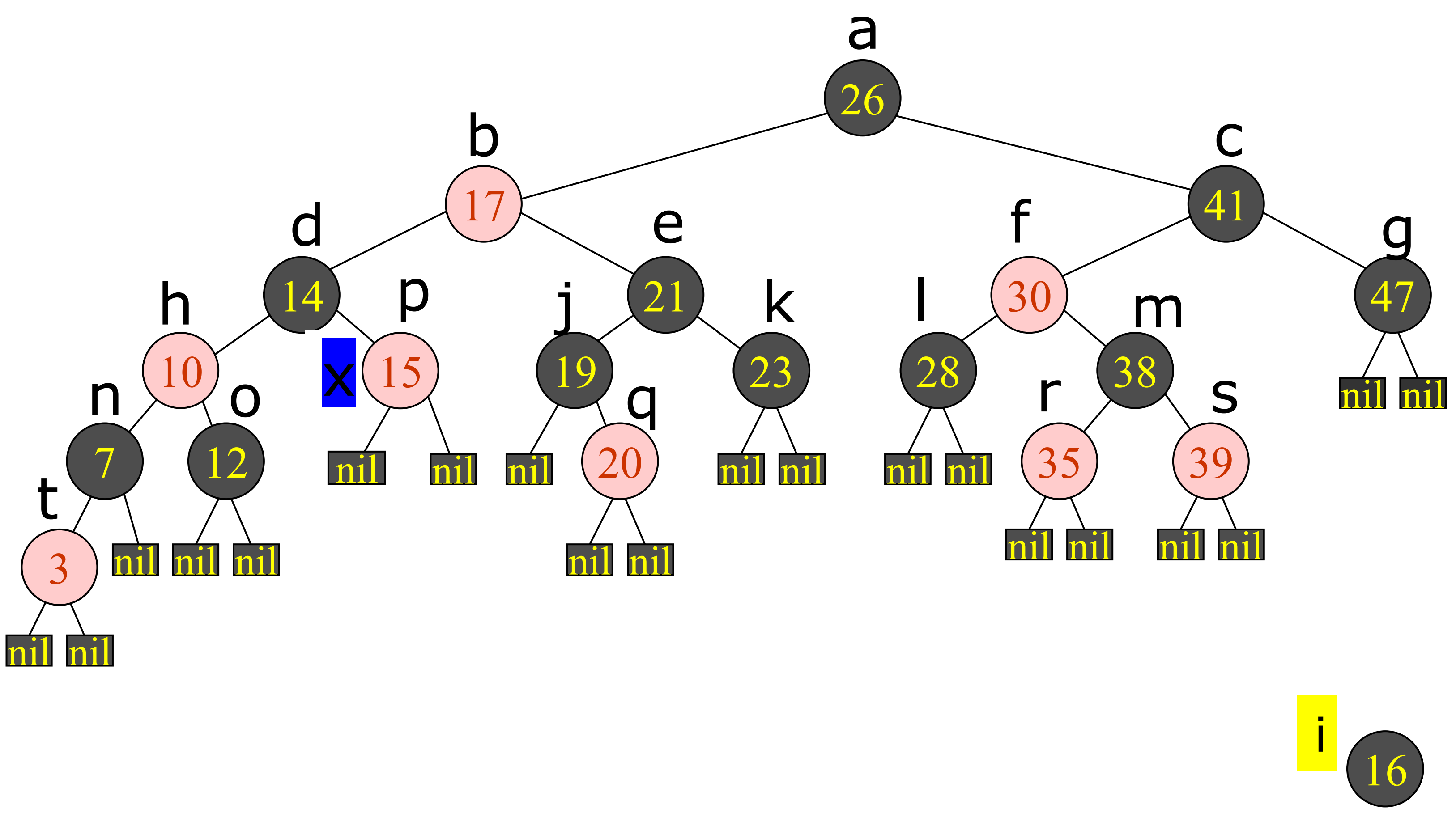
Deleting Node $z = i$ (with key=16): Step 1 of 2



Case-2 BST deletion

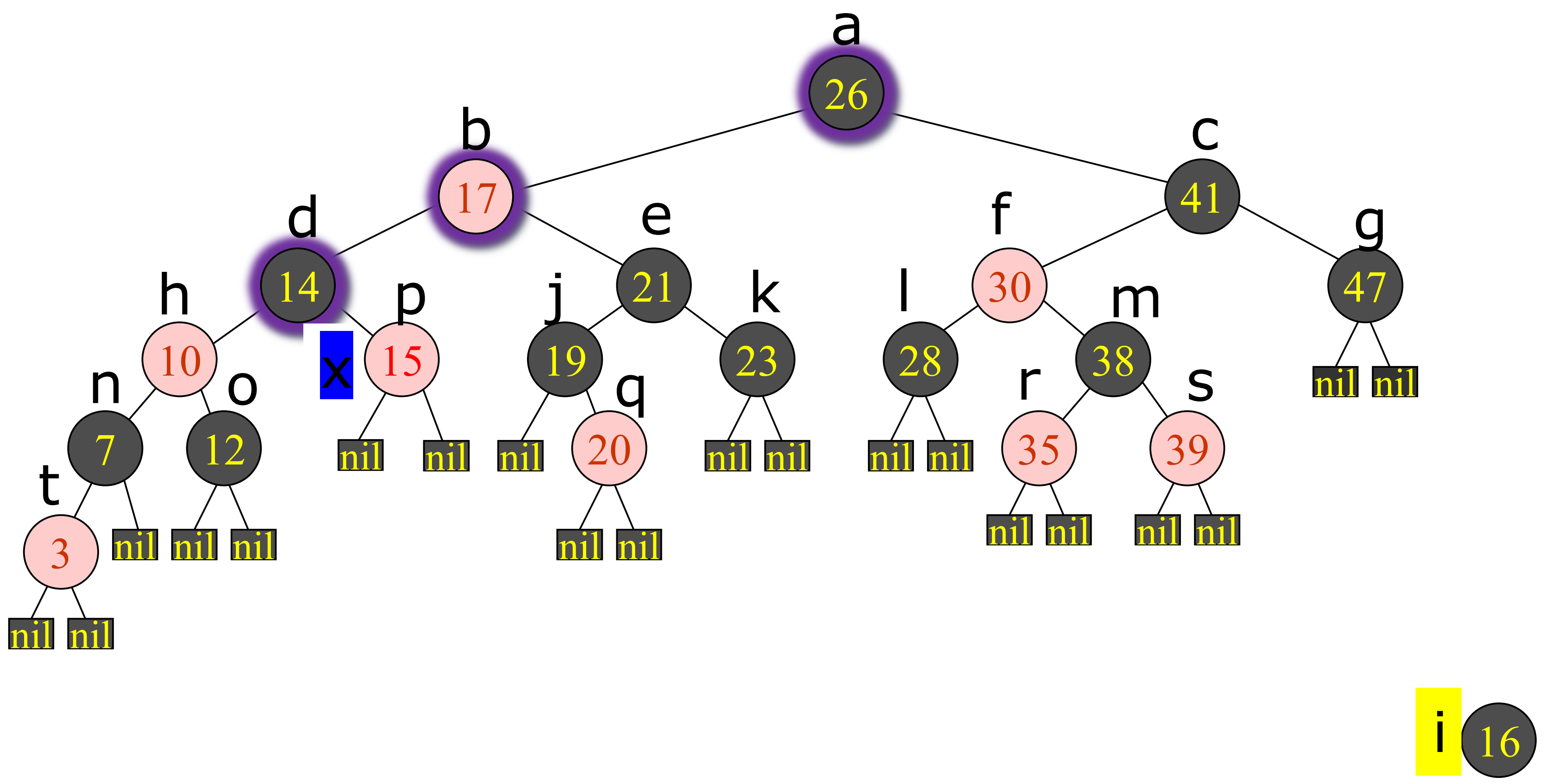
Deleting node i and connecting neighbors

Deleting Node $z = i$ (with key=16): Step 2 of 2



Deleting node i and connecting neighbors

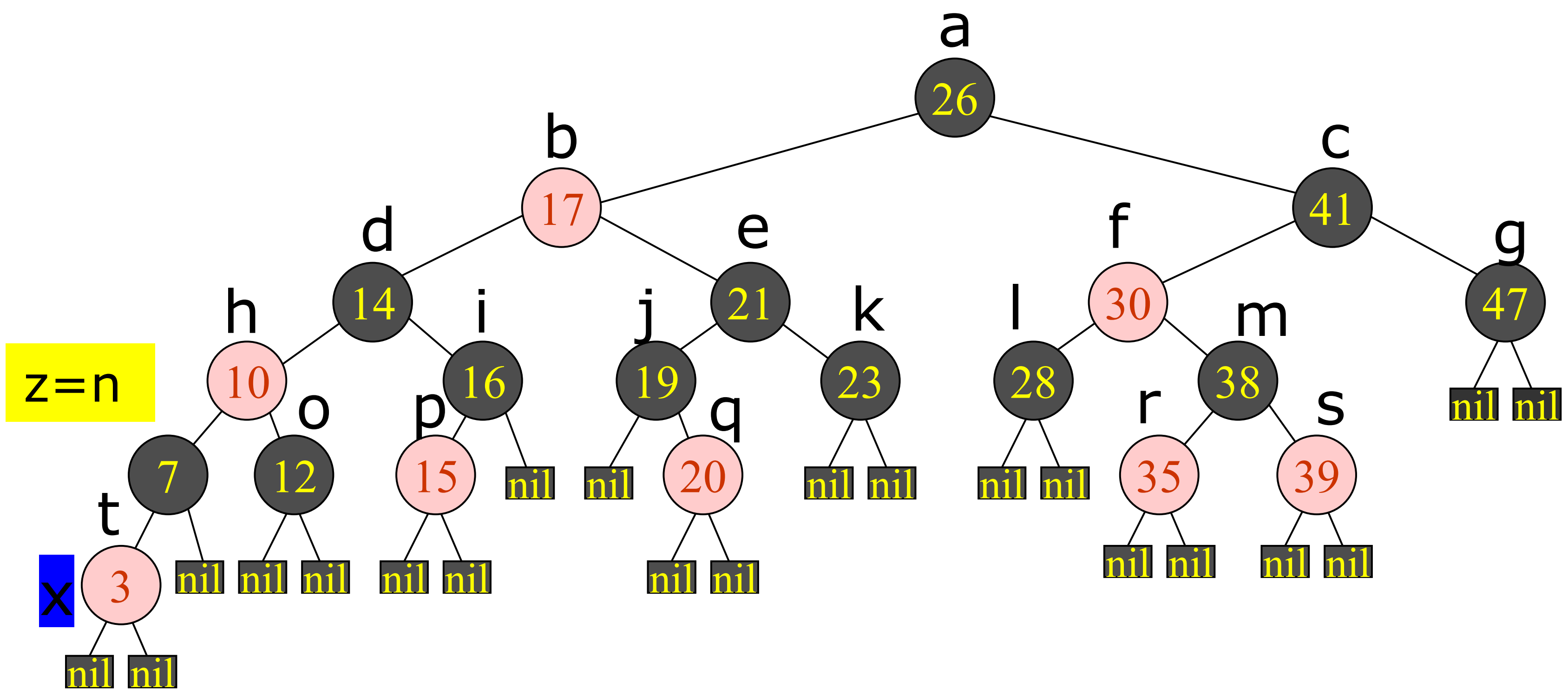
Deleting Node $z = i$ (with key=16): Result



Is the resulting tree still an RB-tree?

NO... Property 5 is violated...at all ancestors of x.

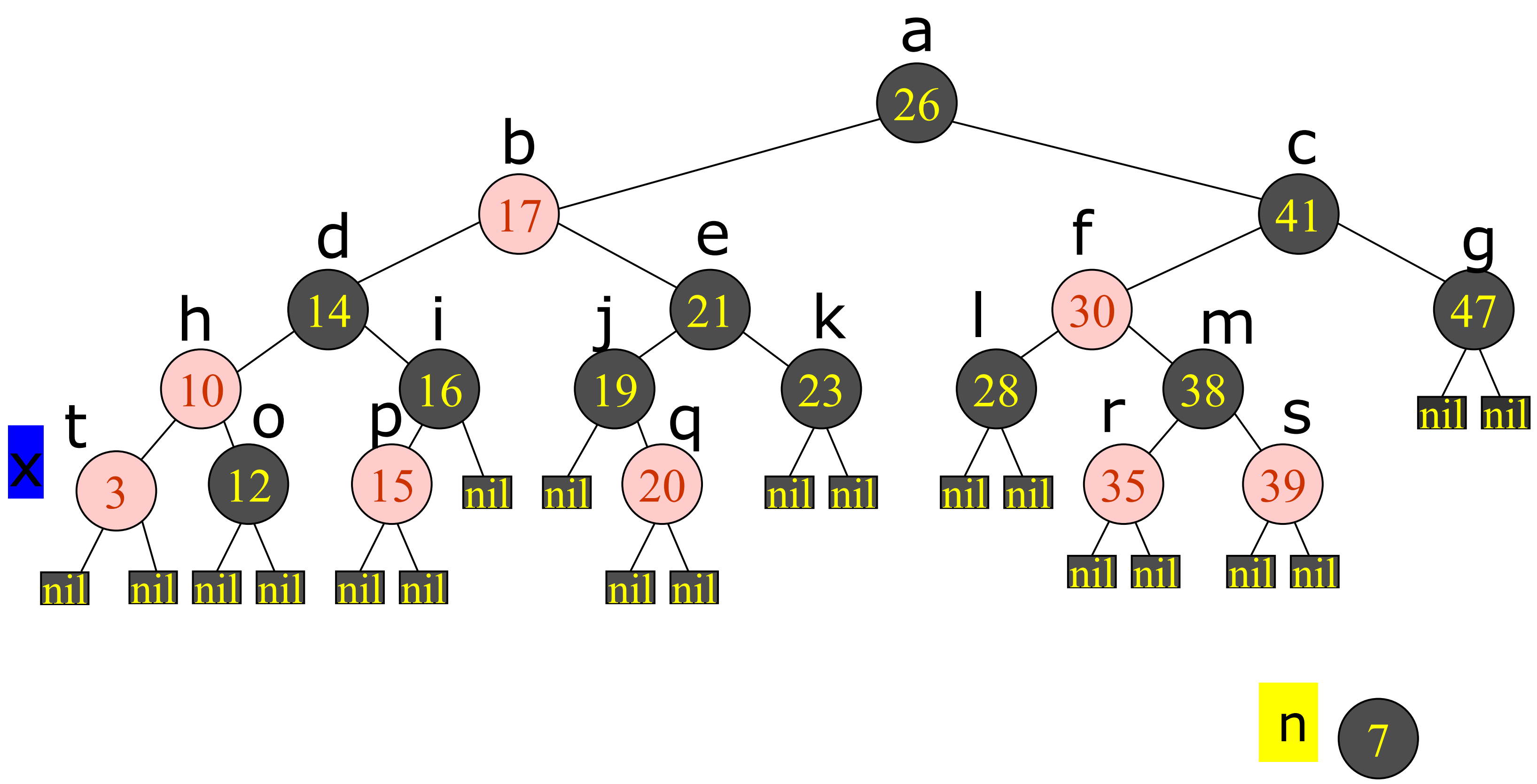
Deleting Node $z = n$ (with key=7): Step 1 of 2



Case-2 BST deletion

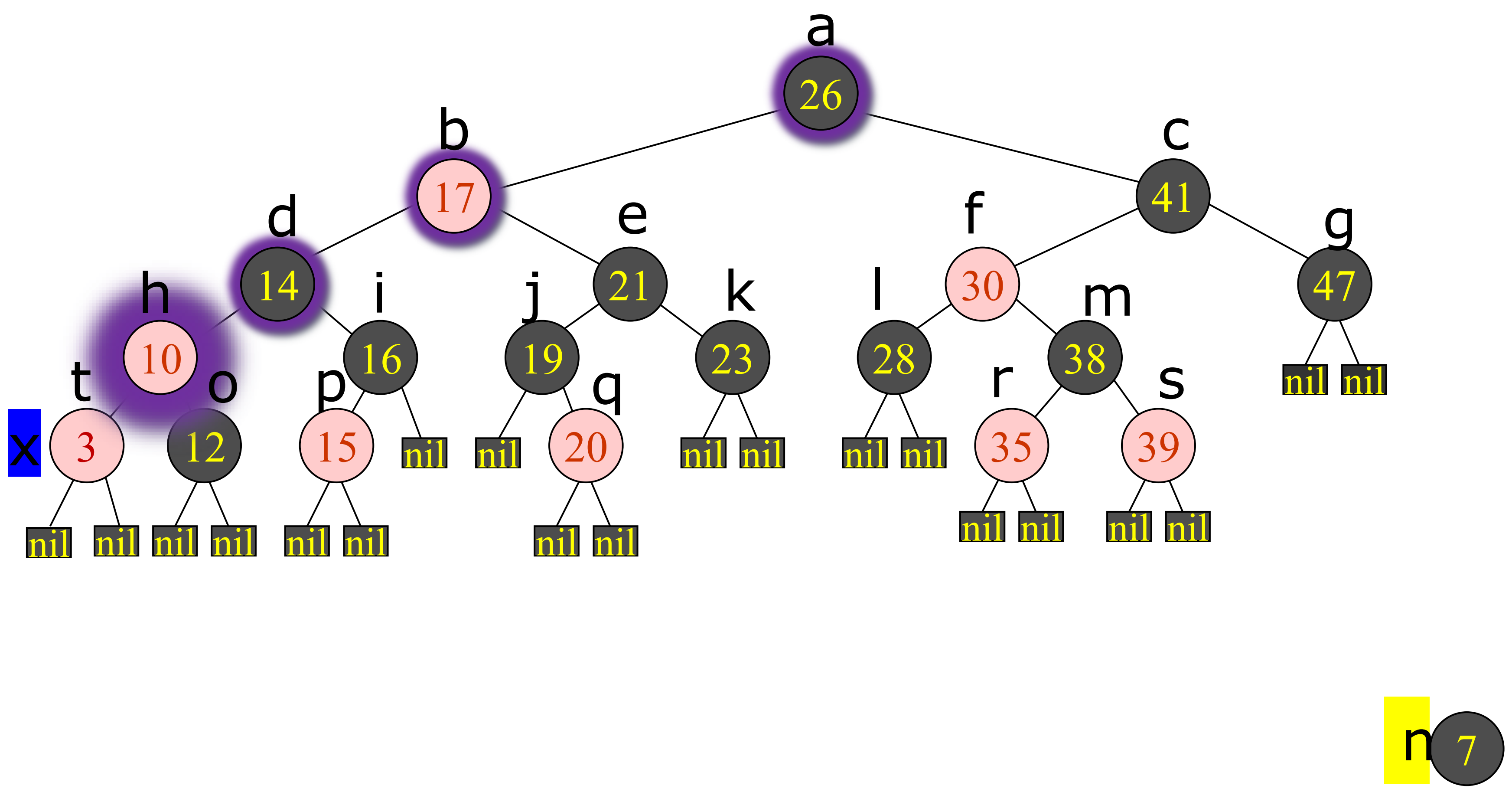
Deleting node n and connecting neighbors

Deleting Node $z = n$ (with key=7): Step 2 of 2



Deleting node n and connecting neighbors

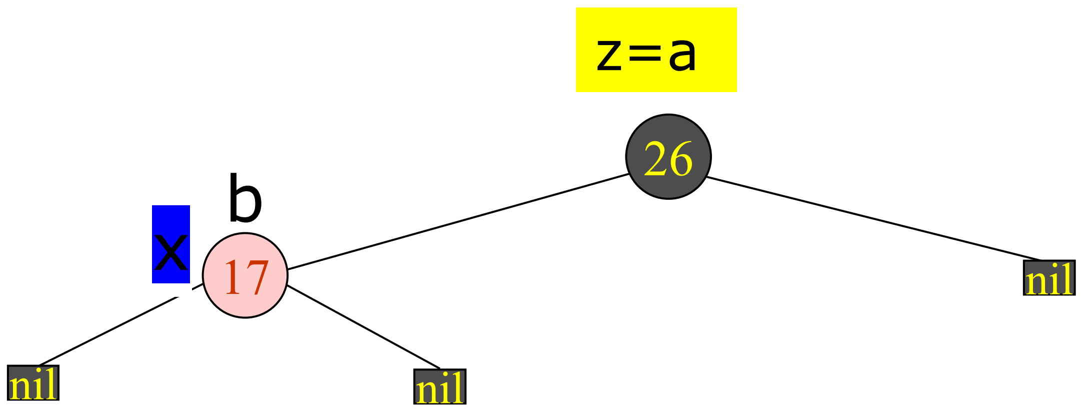
Deleting Node $z = n$ (with key=7): Result



Is the resulting tree still an RB-tree?

NO... Both property 4 and property 5 are violated

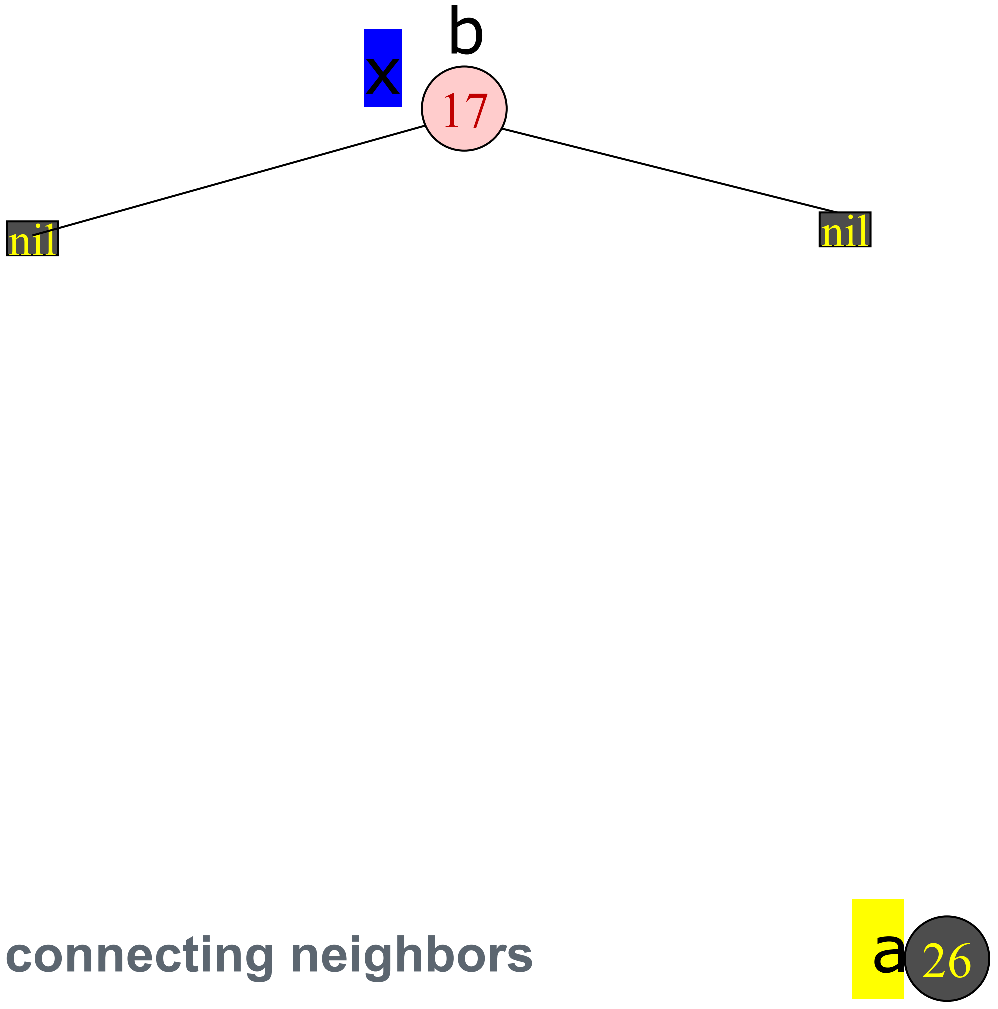
Deleting Node $z = a$ (with key=26): Step 1 of 2



Case-2 BST deletion

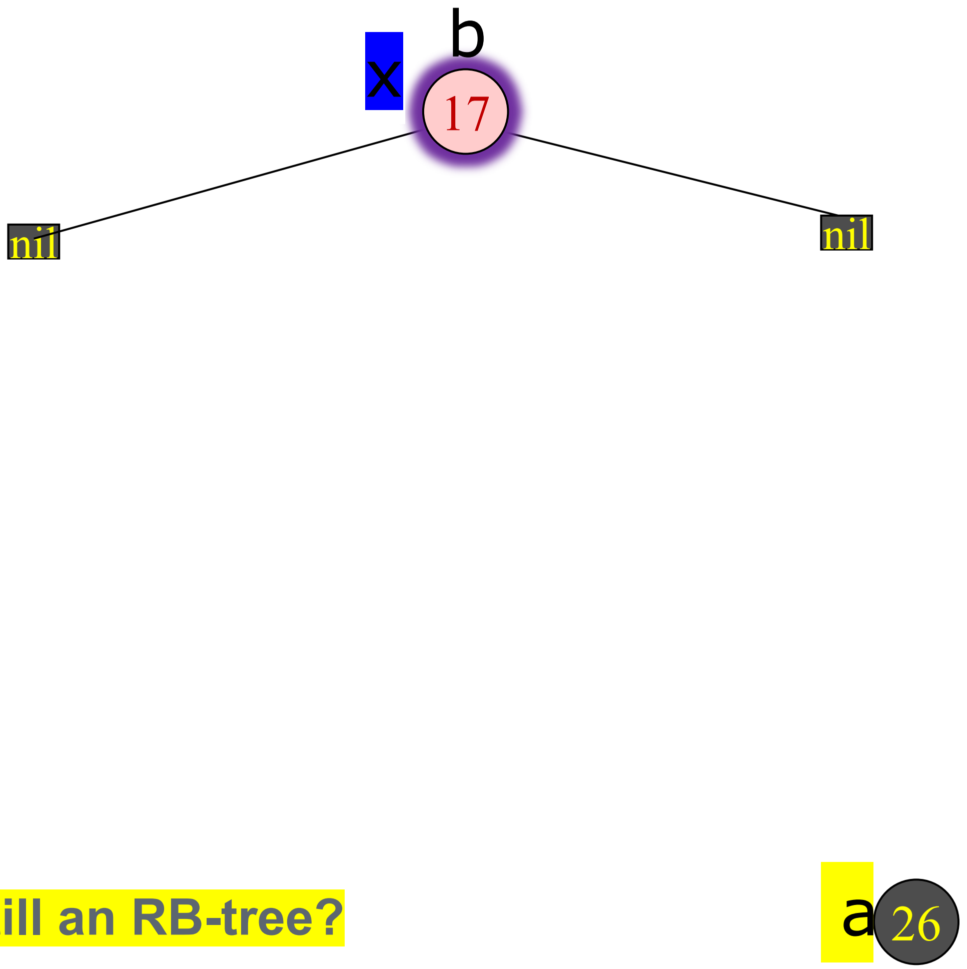
Deleting node a and connecting neighbors

Deleting Node $z = a$ (with key=26): Step 2 of 2



Deleting node a and connecting neighbors

Deleting Node $z = a$ (with key=26): Step 2 of 2



Is the resulting tree still an RB-tree?

NO... Property 2 is violated

Ira A. Fulton Schools of Engineering

Arizona State University

Red Black Trees, Part 6

- Definition and Properties of Red-Black Trees
- Rotations, RBT-Insert Preparation
- RBT-Insert-Fixup (3 cases)
- Examples, Running time of RBT-Insertion
- RBT-Deletion Preparation
- RBT-Deletion-Fixup (4 cases)
- Examples and Summary

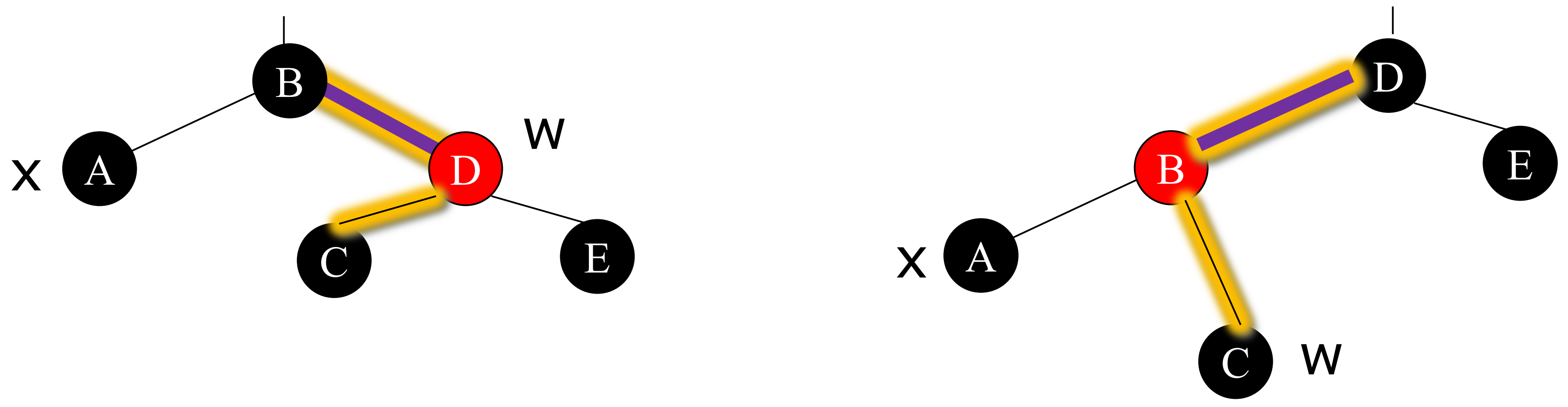
RB-Tree Deletion: First Step

- **Want to delete node z. Perform BST deletion first.**
- **In BST deletion cases 1 and 2, the node y to be spliced out is identical to node z.**
- **In BST deletion case 3, the node y to be spliced out is the successor of node z.**
- **In all cases, node y has at most one non-nil child.**
- **If y has 1 non-nil child, let x point to that child.**
- **If y has 0 non-nil child, let x point to one of the nils.**
- **When replacing z using y, change y to z's color.**

RB-Tree Deletion: First Step

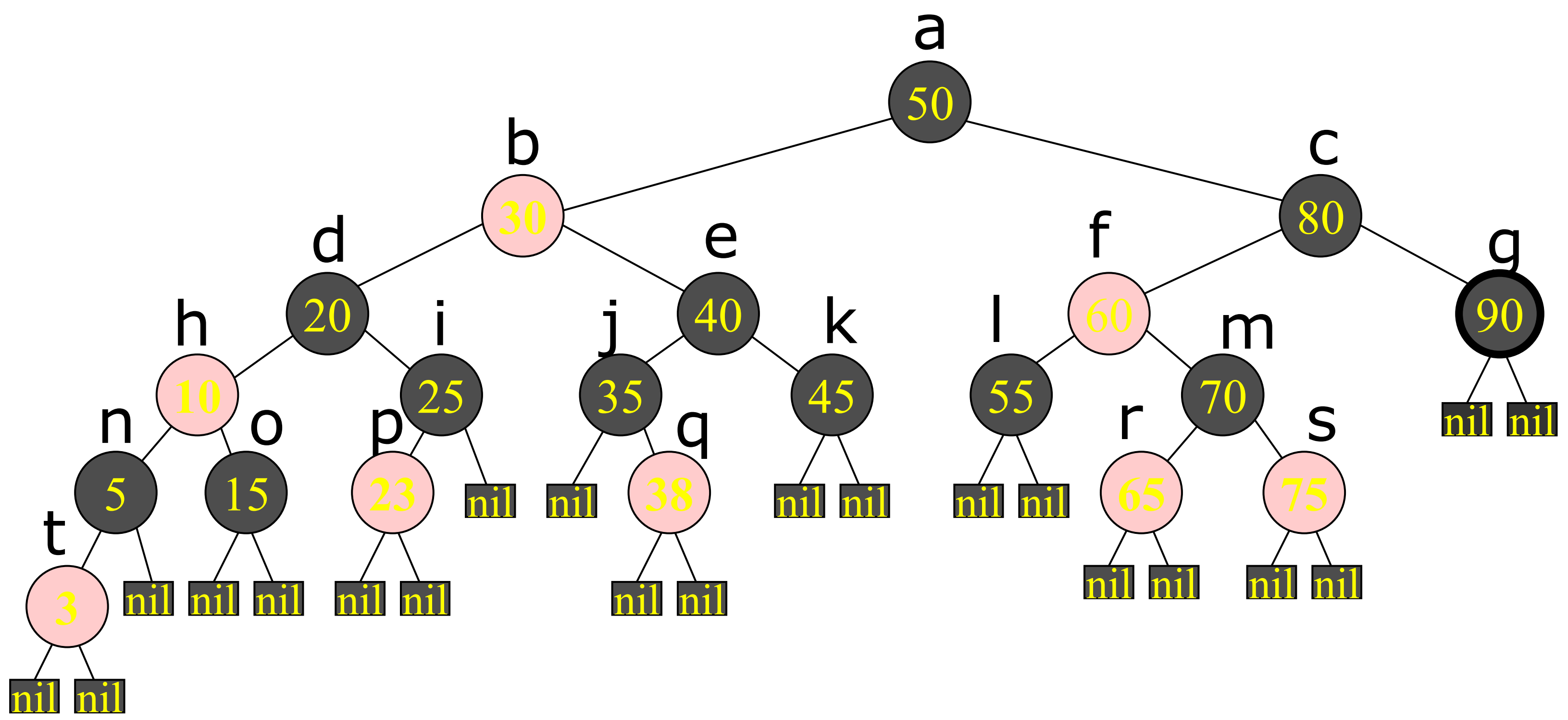
- If y’s original color is red, the resulting tree is still an RB-Tree.
- If y’s original color is black, the resulting tree is not an RB-Tree. In this case, we let the pointer x carry y’s original black color.
- If x points to a red node, make it blk, we are done.
- If x points to a blk node, we need to do a deletion fixup. Let w be the sibling of x.
- We deal with 4 different cases.

RB-Tree Deletion Fixup Case-1



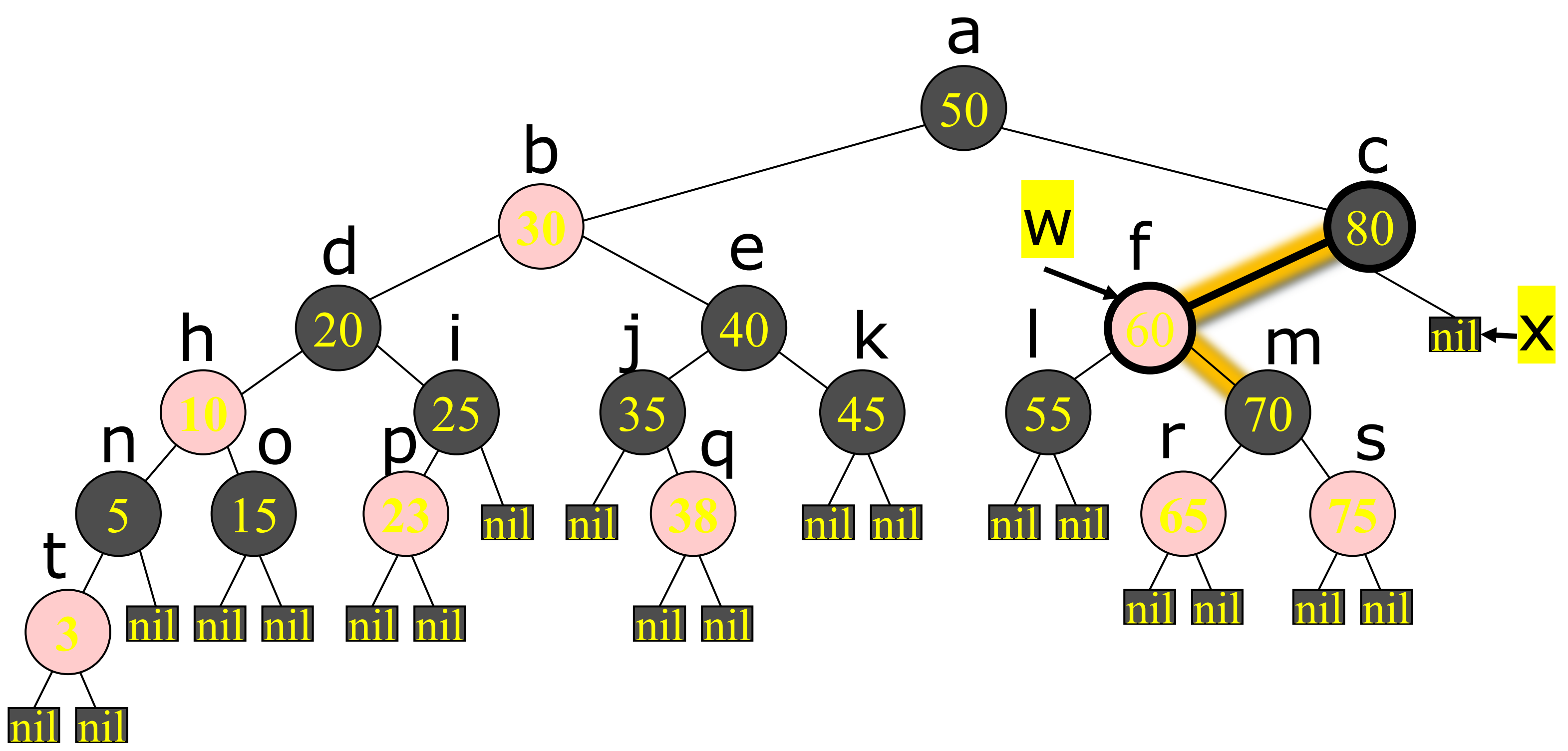
- Case-1: sibling w is red.
- Since x is double black, none of w's children is nil.
- Since w is red, w's parent and children are all black.
- Action: rotate using (p[x], w) as backbone.
- Result: x's new sibling (after rotation) is black. This transforms to cases 2, 3, or 4.

Deletion Fixup Case-1 Example



Suppose we delete 90 (node g). Node g (blk) is spliced out.

Deletion Fixup Case-1 Example

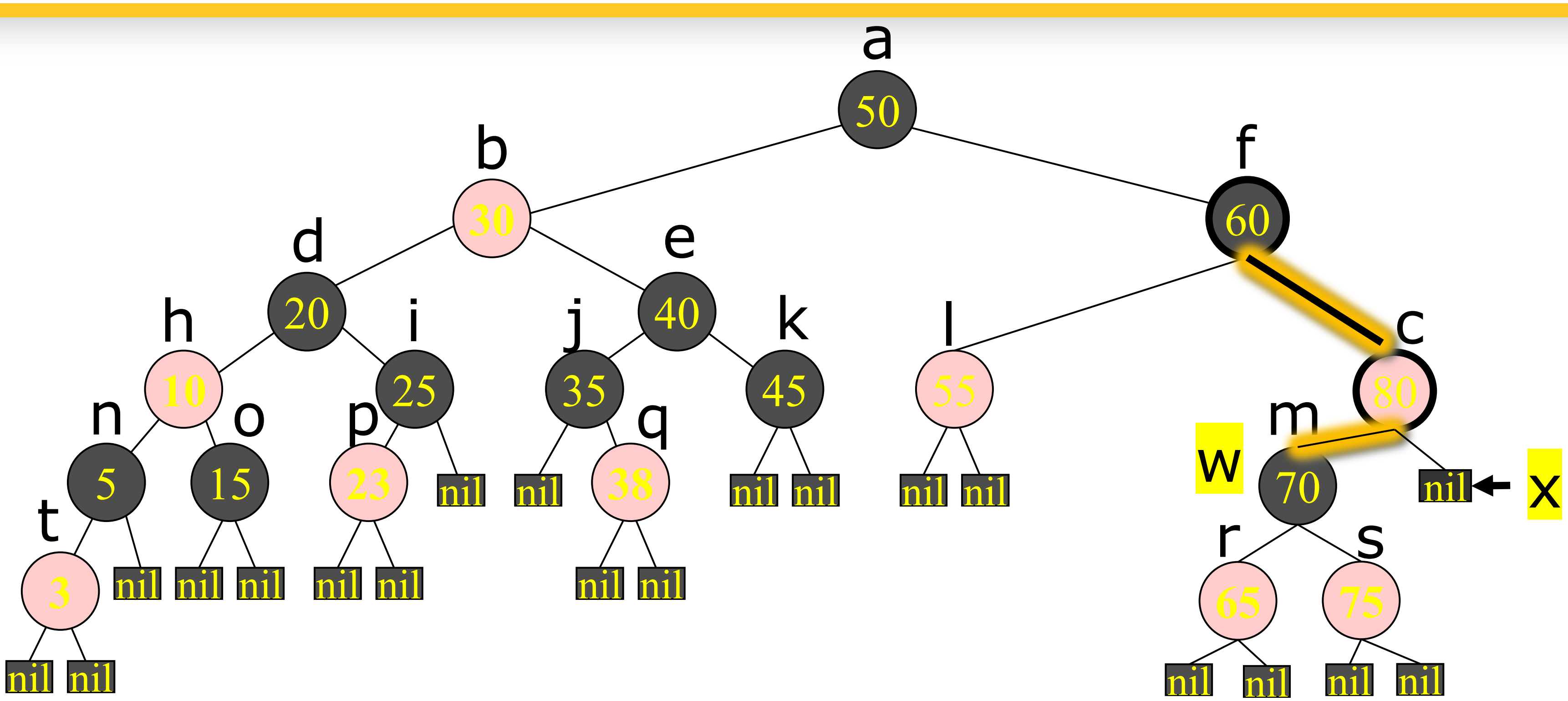


After deleting 90, x is “double-black”.

The **sibling w is red**.

We have **Case 1**: Use parent and sibling as backbone to rotate.

Deletion Fixup Case-1 Example



After deleting 90, x is “double-black”.

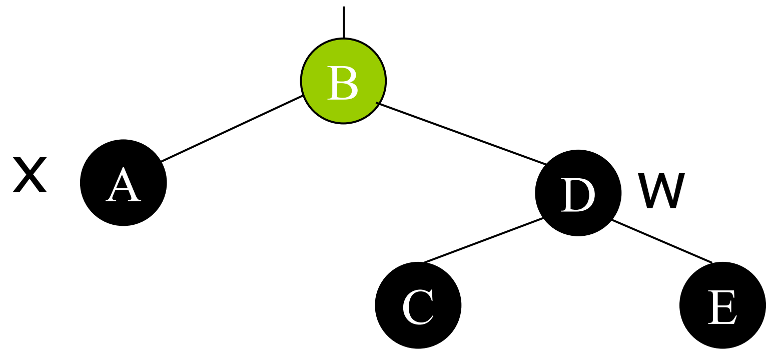
The sibling w is red.

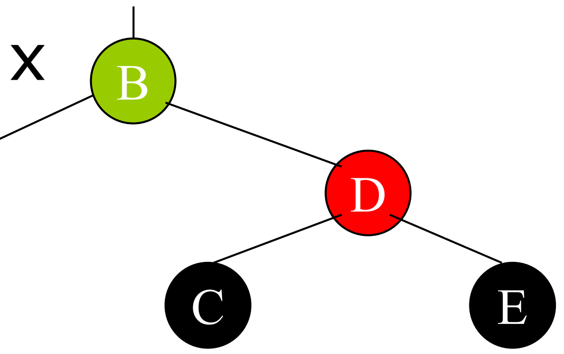
We have Case 1: Use parent and sibling as backbone to rotate.

The new sibling of x is guaranteed to be black.

Not done yet. Just illustrating Case 1 fixup.

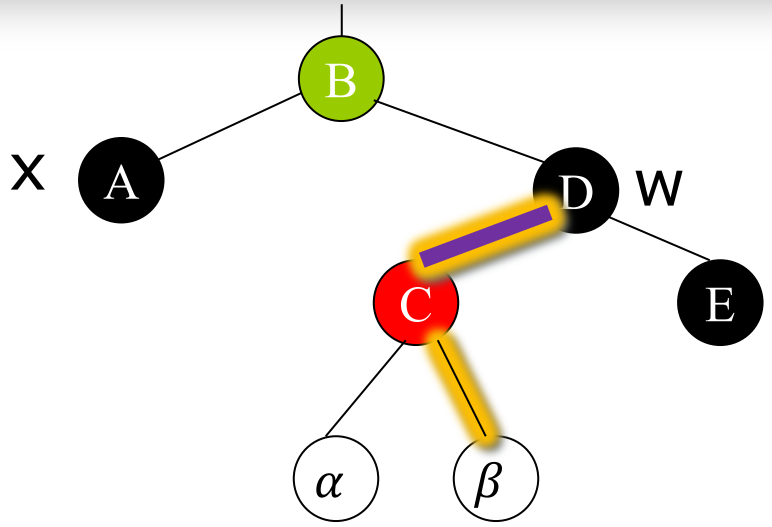
RB-Tree Deletion Fixup Case-2

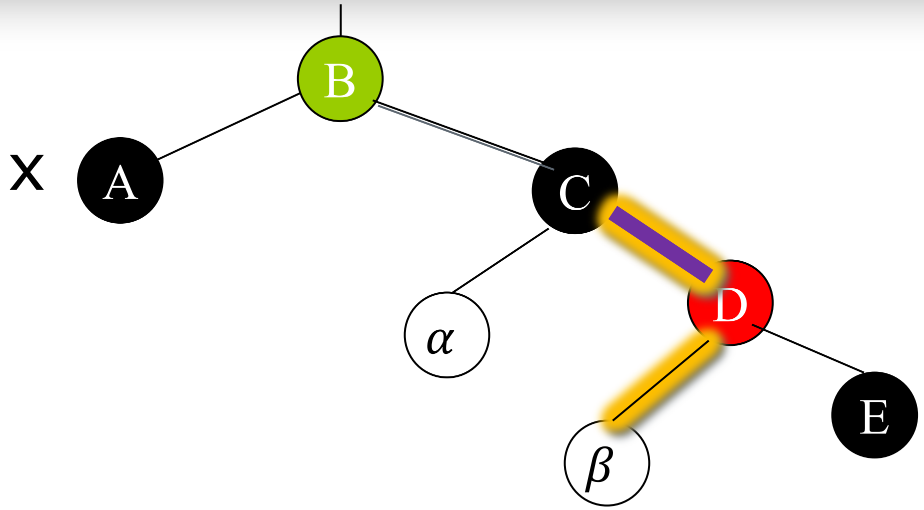




- Case-2: w is blk; w has two blk children.
- The green color (of B) may be either red or blk.
- Action: w.color ←red; x ←p[x].
- Result: If B was red, make it blk, and we are done.
- If B was blk, it is now double-blk.
- If B is root, make it single-blk, and we are done.

RB-Tree Deletion Fixup Case-3





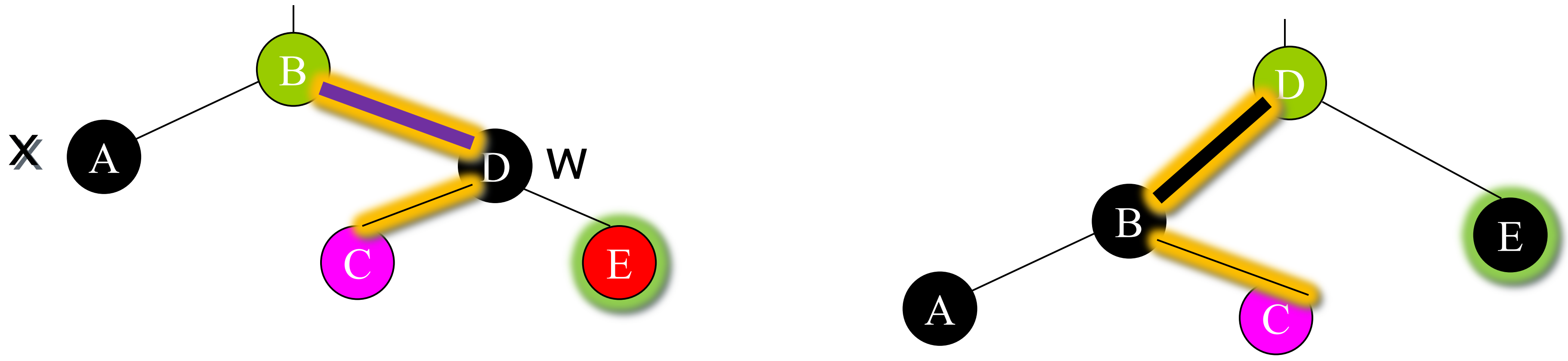
- Case-3: w is blk, with NearChild[w] (near x) red and the other child blk.

- Note. Both α and β must be blk, since C is red.

- Action: Rotate using (w, NearChild[w]) as backbone.

- Result: Transforms to Case-4.

RB-Tree Deletion Fixup Case-4



- Case-4: w is blk, with FarChild[w] (far from x) red.
- Note. Green and Pink are independent colors.
- Action: Rotate using (p[x], w) as backbone; transfer the extra blk color of x to FarChild[w]
- Result: Violation resolved

Summary of RB-Tree Deletion Fixup Cases

- **Case-4 can be resolved in $\boldsymbol{O}(1)$ time.**
- **Case-3 can be transformed to Case-4 in $\boldsymbol{O}(1)$ time.**
- **In Case-2, we either resolve the violation in $\boldsymbol{O}(1)$ time or move up the tree by one hop.**
- **In Case-1, we spend $\boldsymbol{O}(1)$ time to transform to one of the other cases.**
  - If we transform to Case-2, the violation will be resolved in $\boldsymbol{O}(\mathbf{1})$ time.
  - If we transform to Case-3 or Case-4, the violation will be resolved in $\boldsymbol{O}(\mathbf{1})$ time.
- **Therefore the worst-case time complexity is $\boldsymbol{O}(\log \boldsymbol{n})$**
- **Case-4: w is blk, with FarChild[w] (far from x) red.**
- **Note. Green and Pink are independent colors.**
- **Action: Rotate using (p[x], w) as backbone; transfer the extra blk color of x to FarChild[w]**
- **Result: Violation resolved**

ASU®

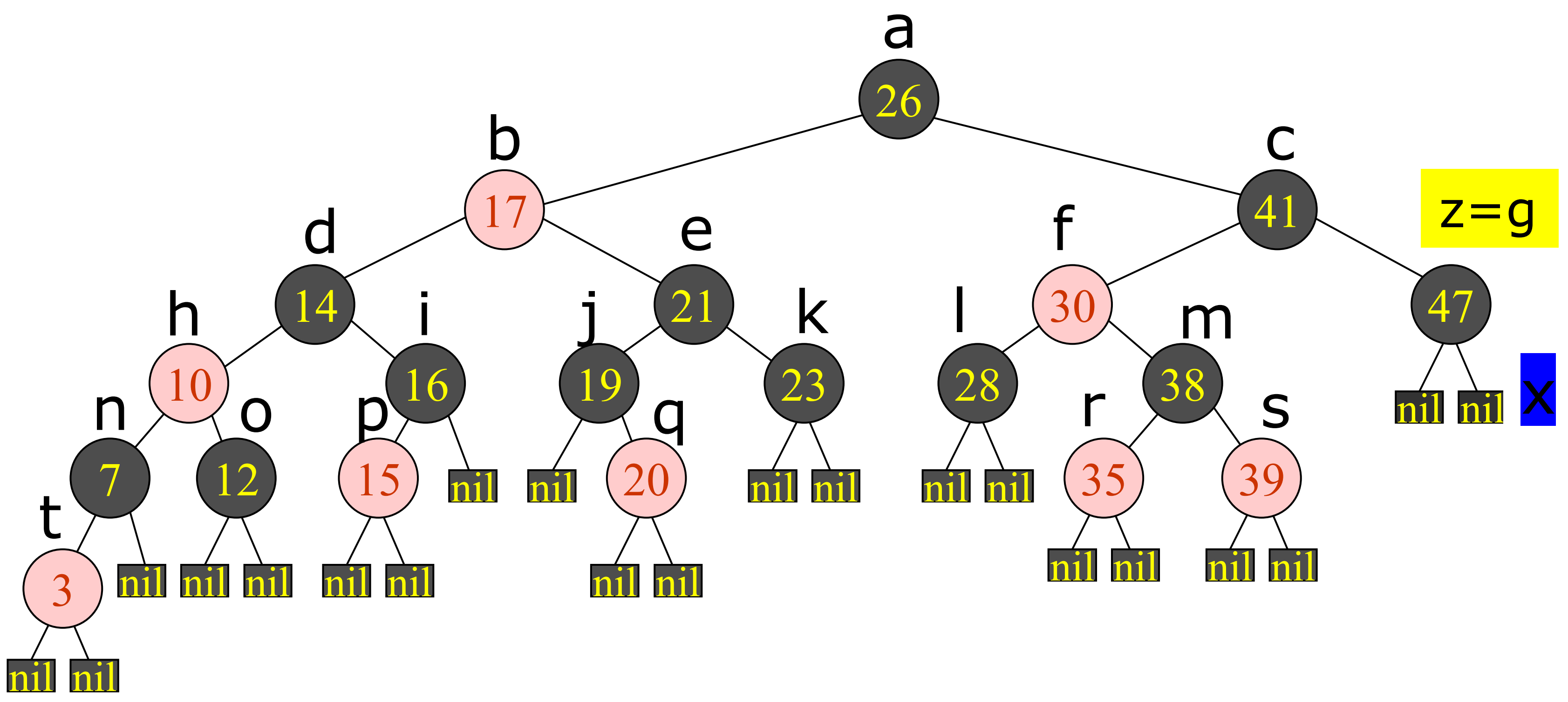
Ira A. Fulton Schools of Engineering

Arizona State University

Red Black Trees, Part 7

- Definition and Properties of Red-Black Trees
- Rotations, RBT-Insert Preparation
- RBT-Insert-Fixup (3 cases)
- Examples, Running time of RBT-Insertion
- RBT-Deletion Preparation
- RBT-Deletion-Fixup (4 cases)
- Examples and Summary

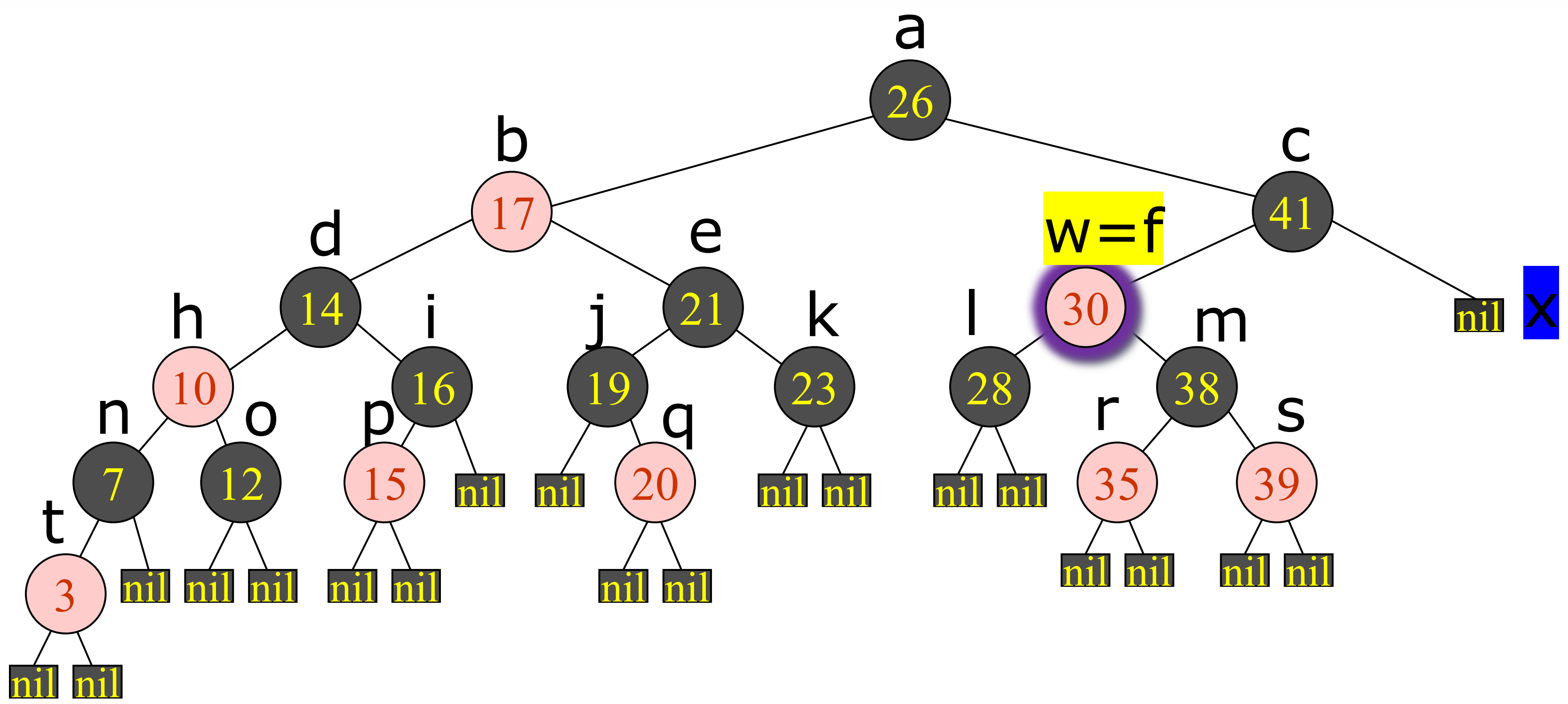
Example: Deleting Node $z = g$ (with key=47)



Case-1 BST deletion

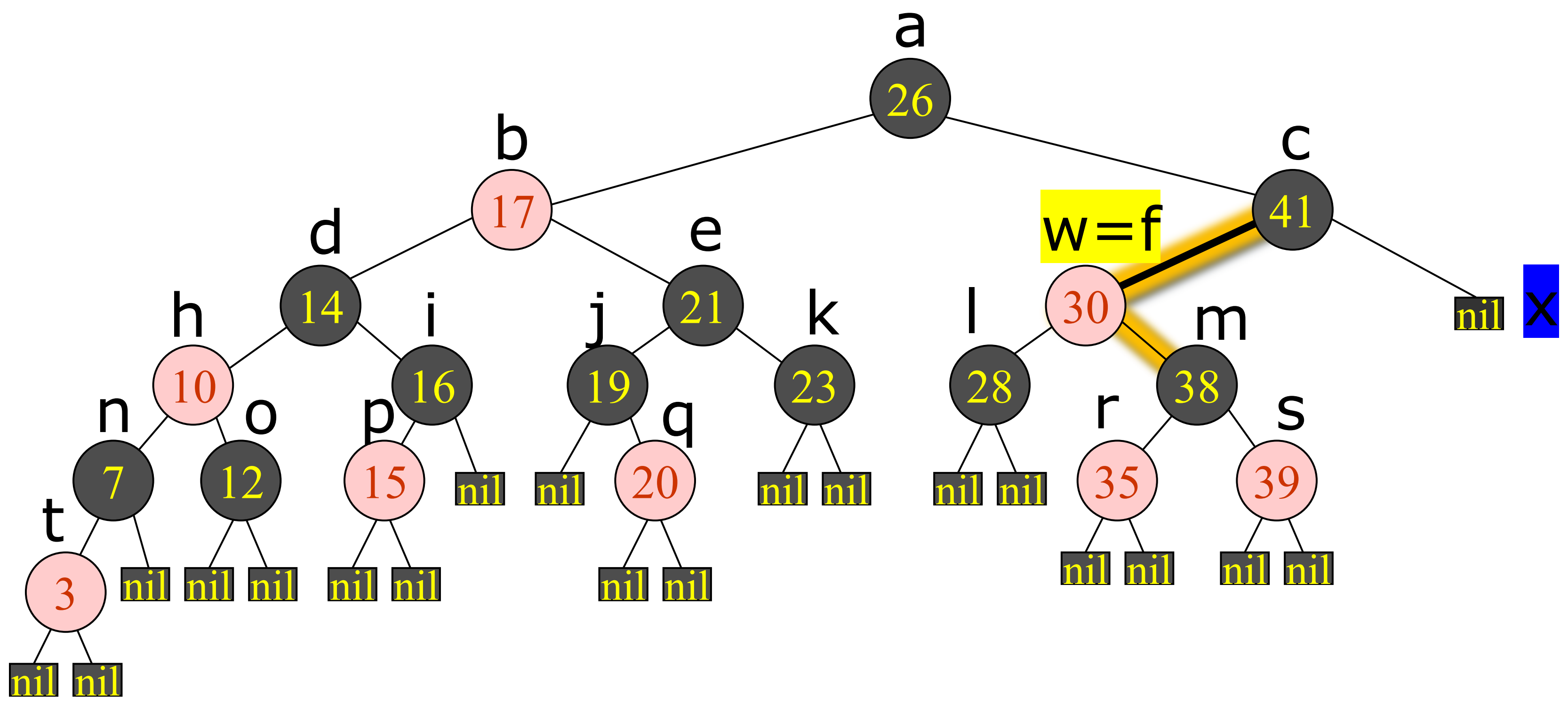
Deleting node g directly

Example: Deleting Node $z = g$ (with key=47)



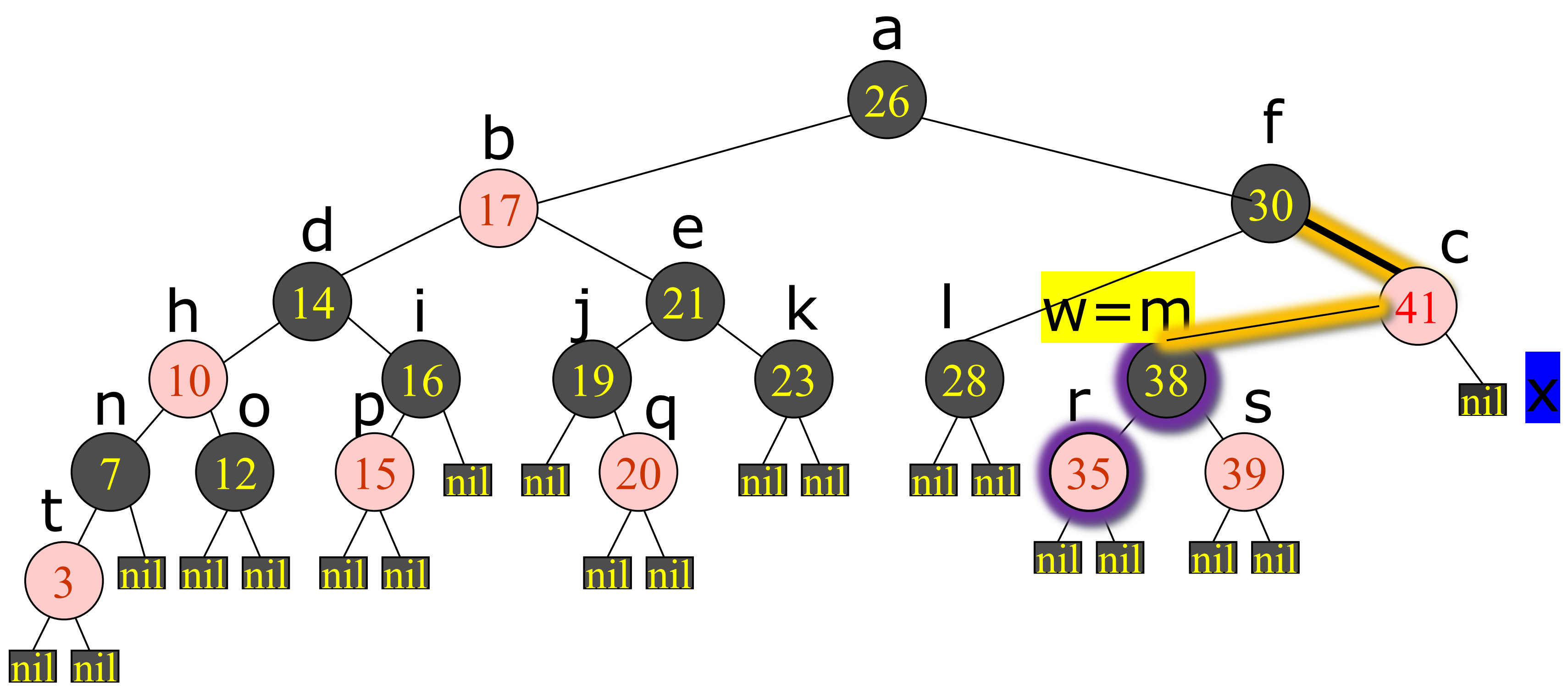
x is double-blk. sibling w is red. We have Case-1.

Example: Deleting Node $z = g$ (with key=47)



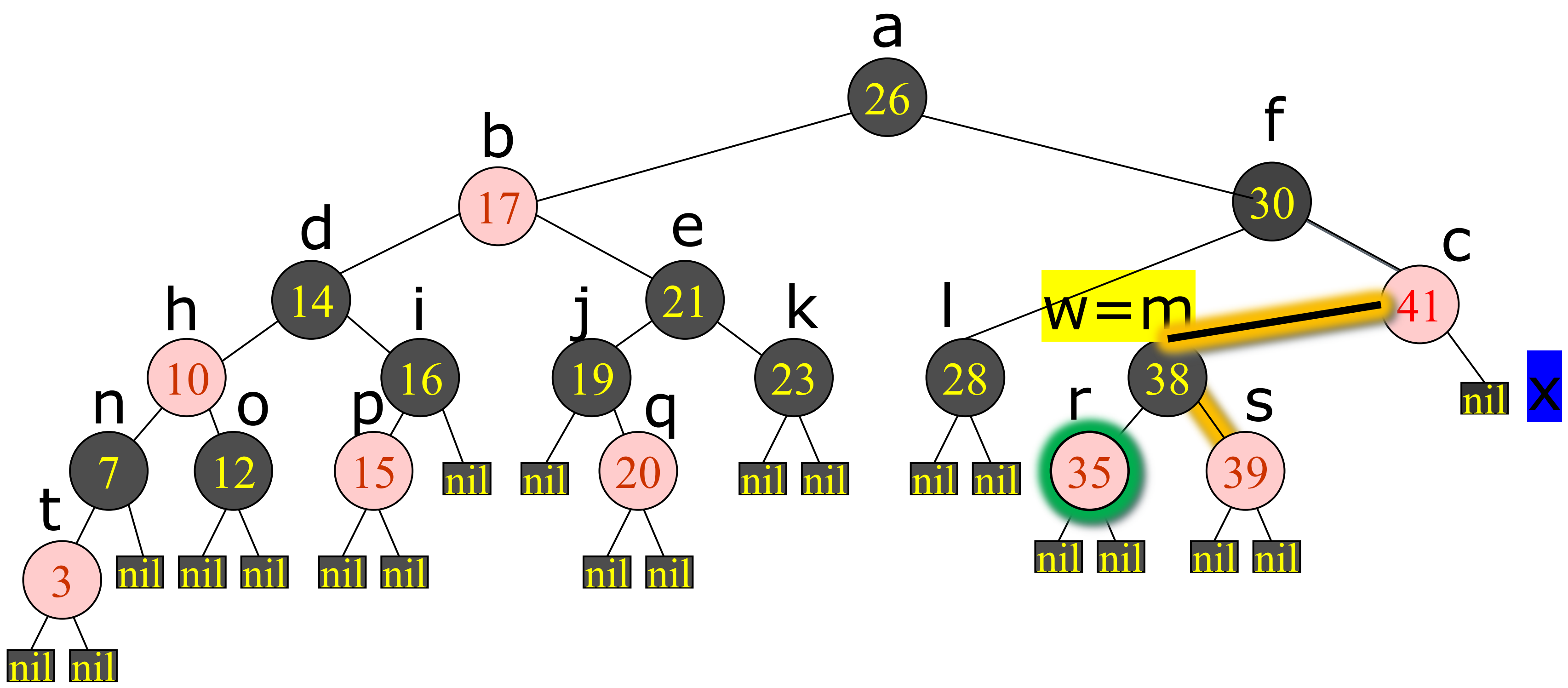
Rotate using (w, p[x]) as backbone

Example: Deleting Node $z = g$ (with key=47)



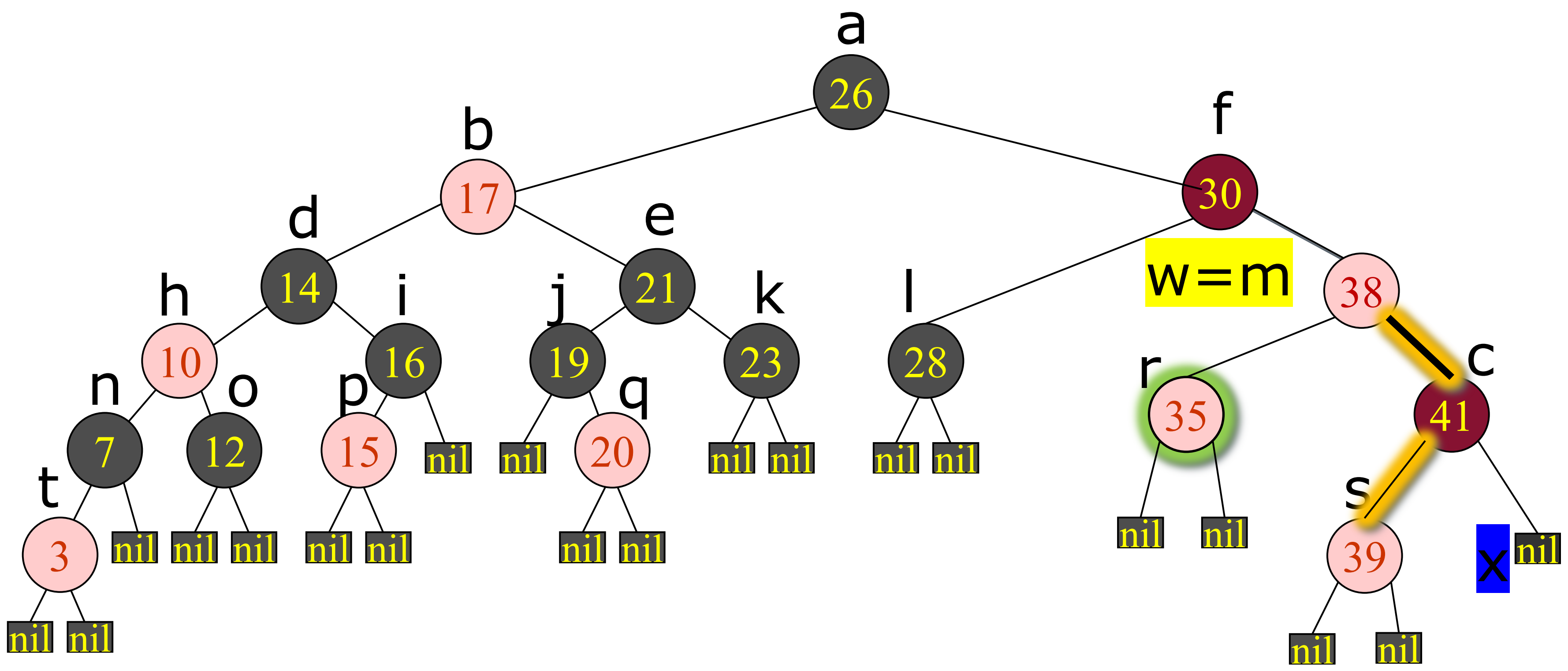
Now we have Case-4. FarChild[w] is r.

Example: Deleting Node $z = g$ (with key=47)



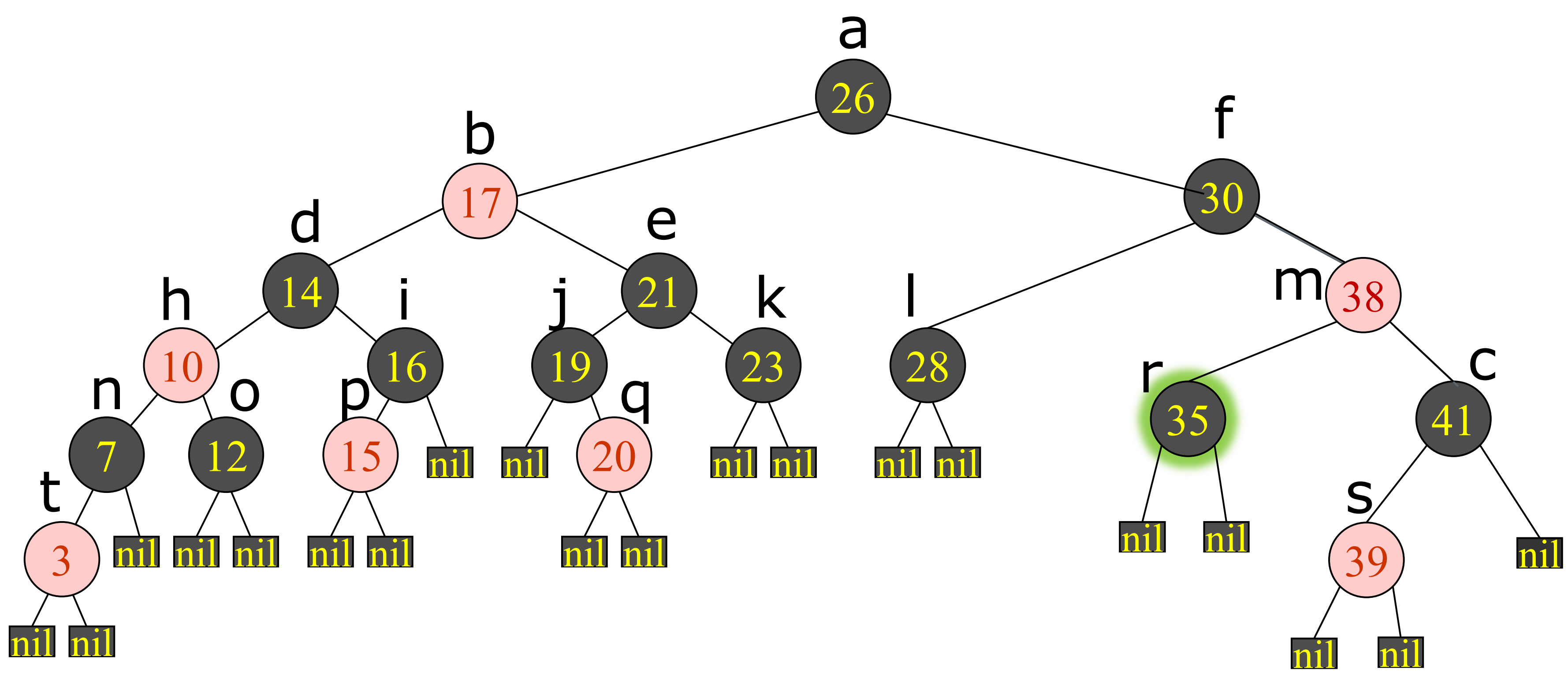
Rotate using (w, p[x]) as backbone.

Example: Deleting Node $z = g$ (with key=47)



Rotate using (w, p[x]) as backbone.

Example: Deleting Node $z = g$ (with key=47)



Transfer the extra blk color of x to r.

We are done.

Time Complexity

In an RB-Tree, insertion, deletion, search can all be done in $O(\log n)$ time.

Ira A. Fulton Schools of Engineering

Arizona State University

Insertion Into A Red-Black Tree

The goal is to move the violation up the tree, until it disappears or reaches the root.

1. <u>If the uncle of x is red</u> (grandparent must be black): We change x’s parent and uncle to black and its grandparent to red. Move the violation up!

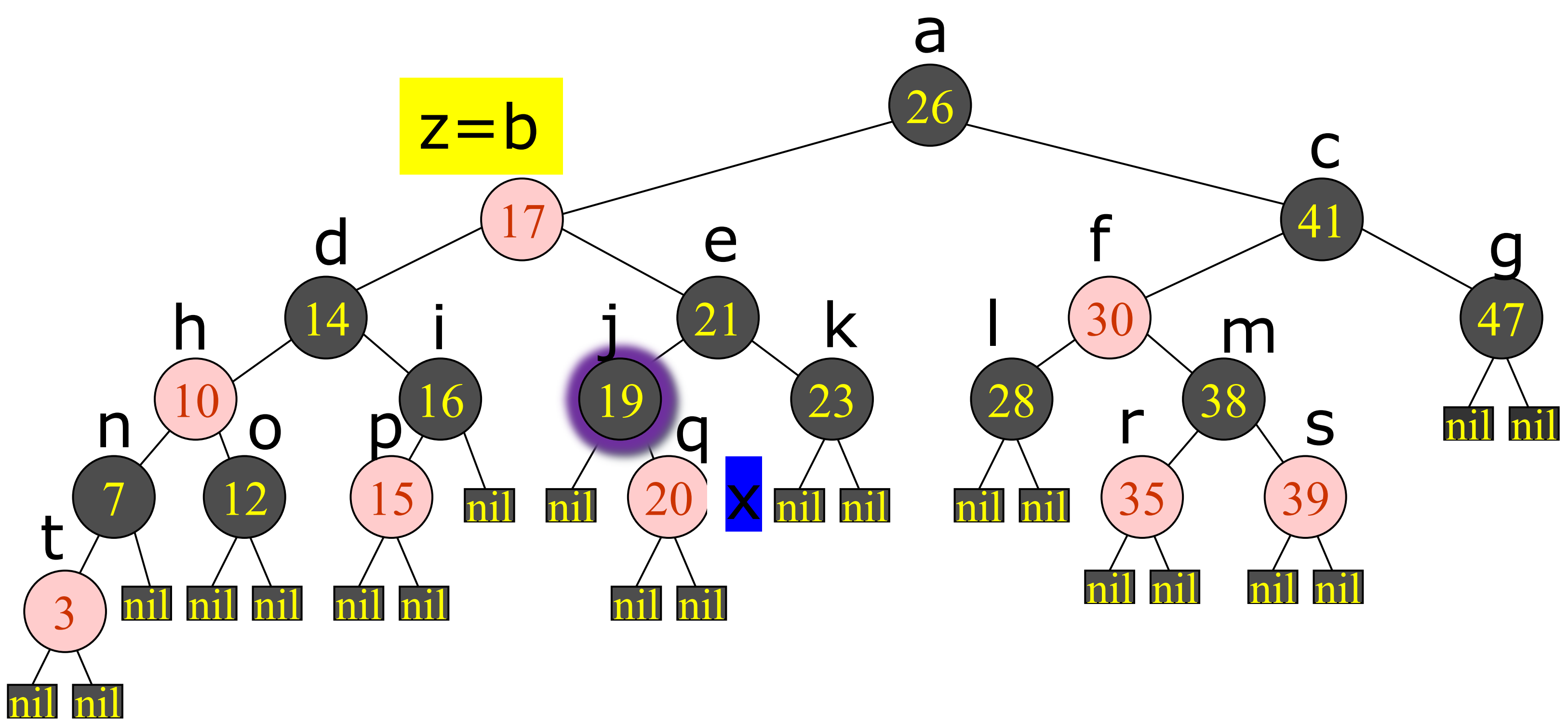
2. <u>Now assume that the uncle of x is black</u>:

a) If x is near its uncle, use x and its parent as the backbone to perform a rotation, repointing x to the lower of the two consecutive red nodes after the rotation.

b) If x is far from its uncle, use the parent and the grandparent as the backbone to perform a rotation. Now the violation is removed, because the uncle is black

More Examples…

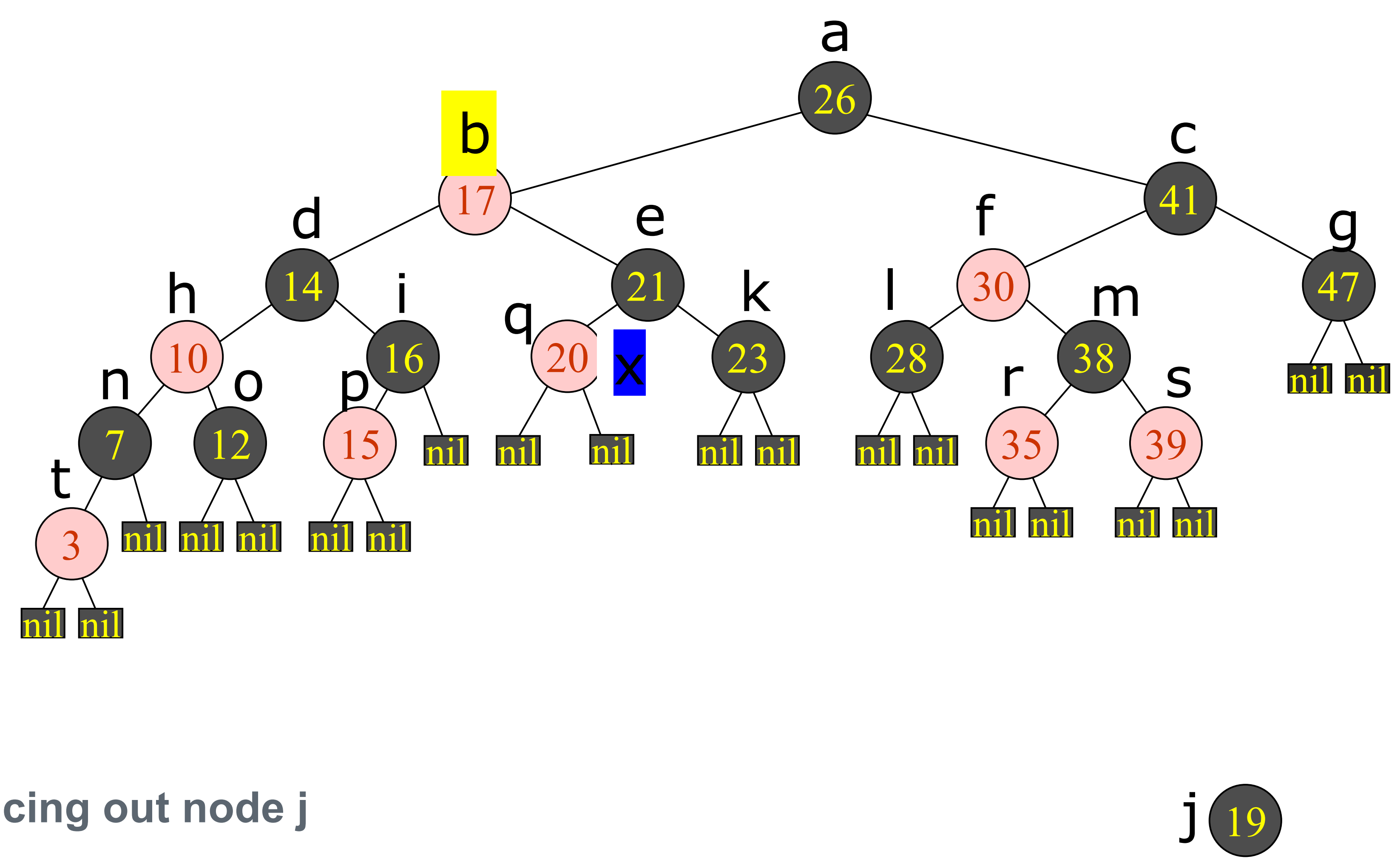
Deleting Node $z = b$ (with key=17), Step 1 of 3



Case-3 BST deletion

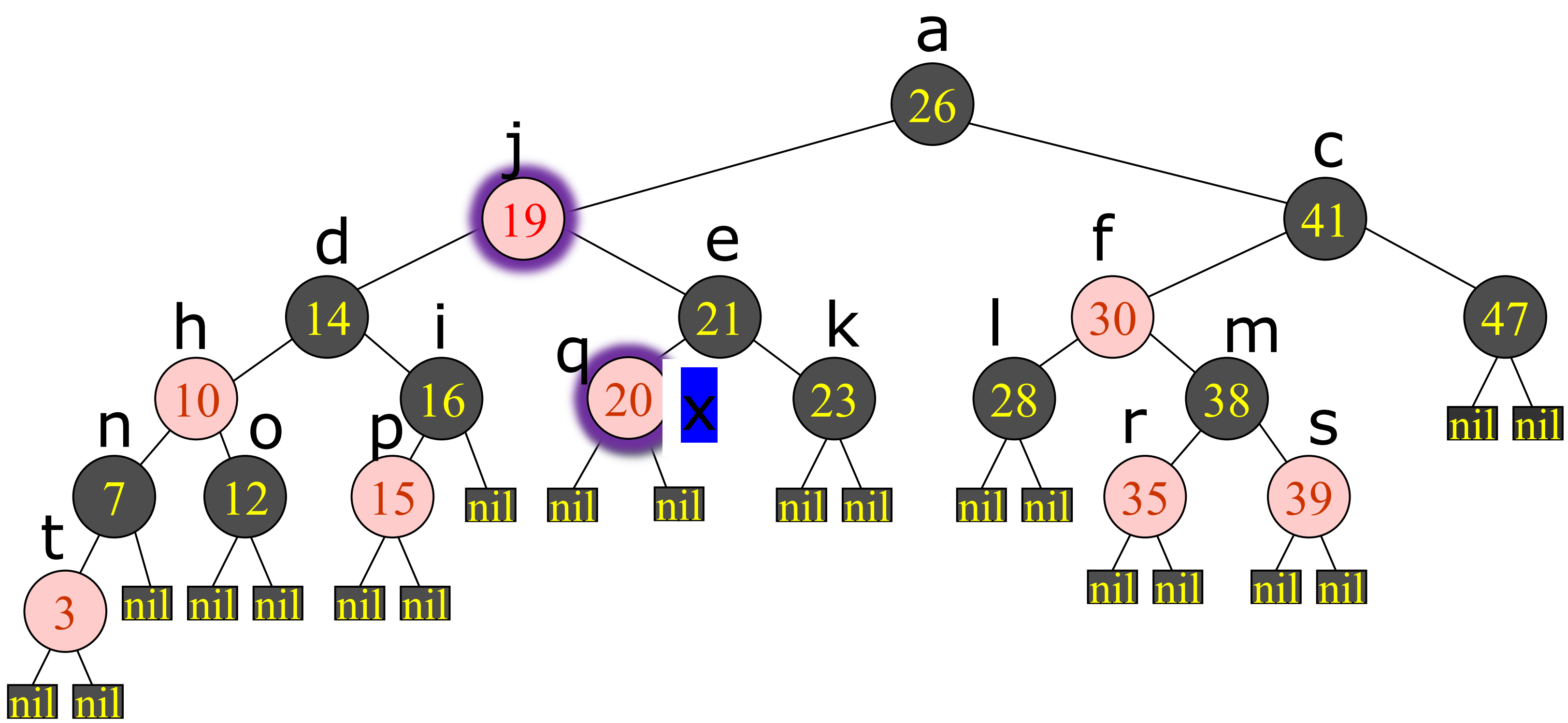
Node *b*'s successor (*j*) is sliced out, then replace node *b*

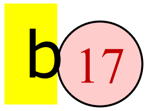
Deleting Node $z = b$ (with key=17), Step 2 of 3



Slicing out node j

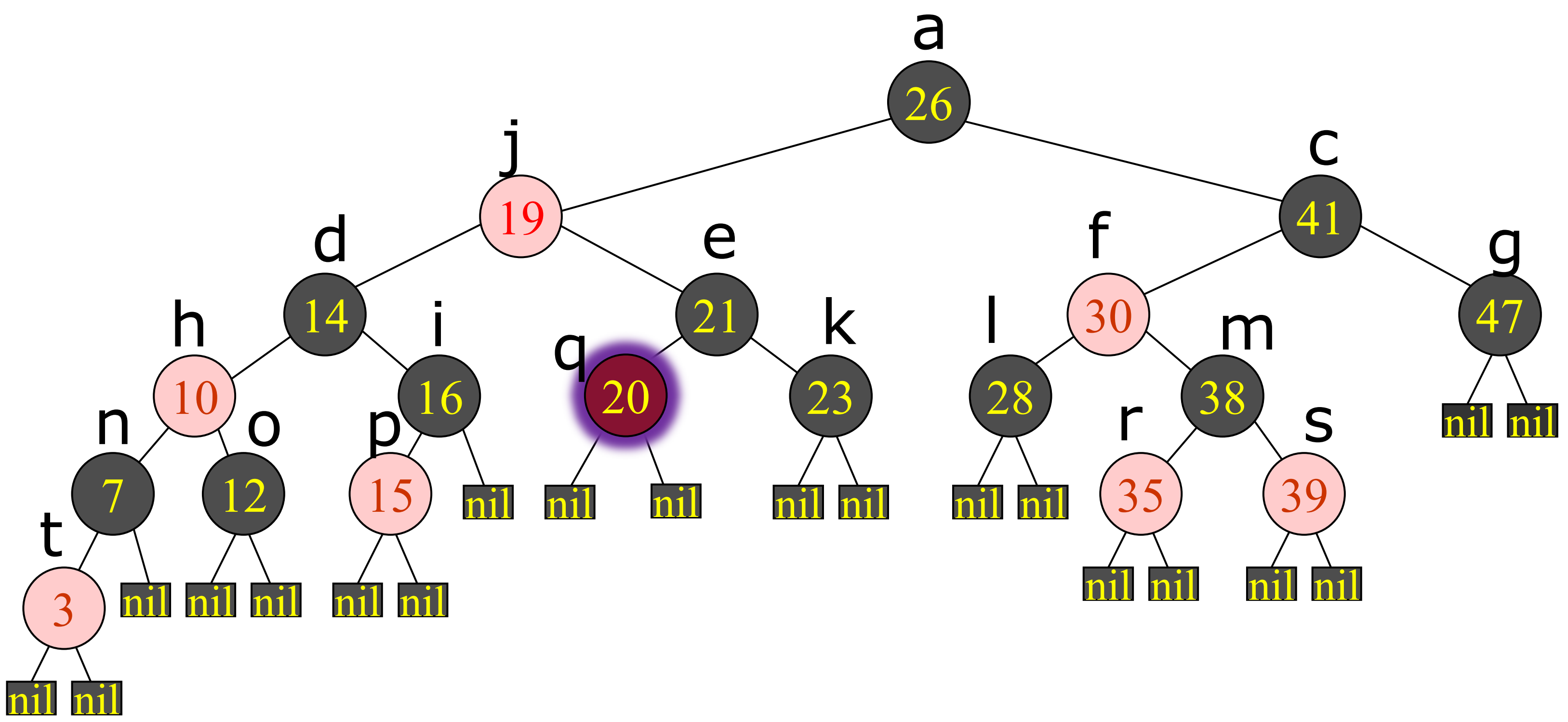
Deleting Node *b* (with key==17), Step 3 of 4





Replacing node *b* with node *j*

Deleting Node *b* (with key==17), Step 4 of 4



Node q is red-black, which is black

We are done.