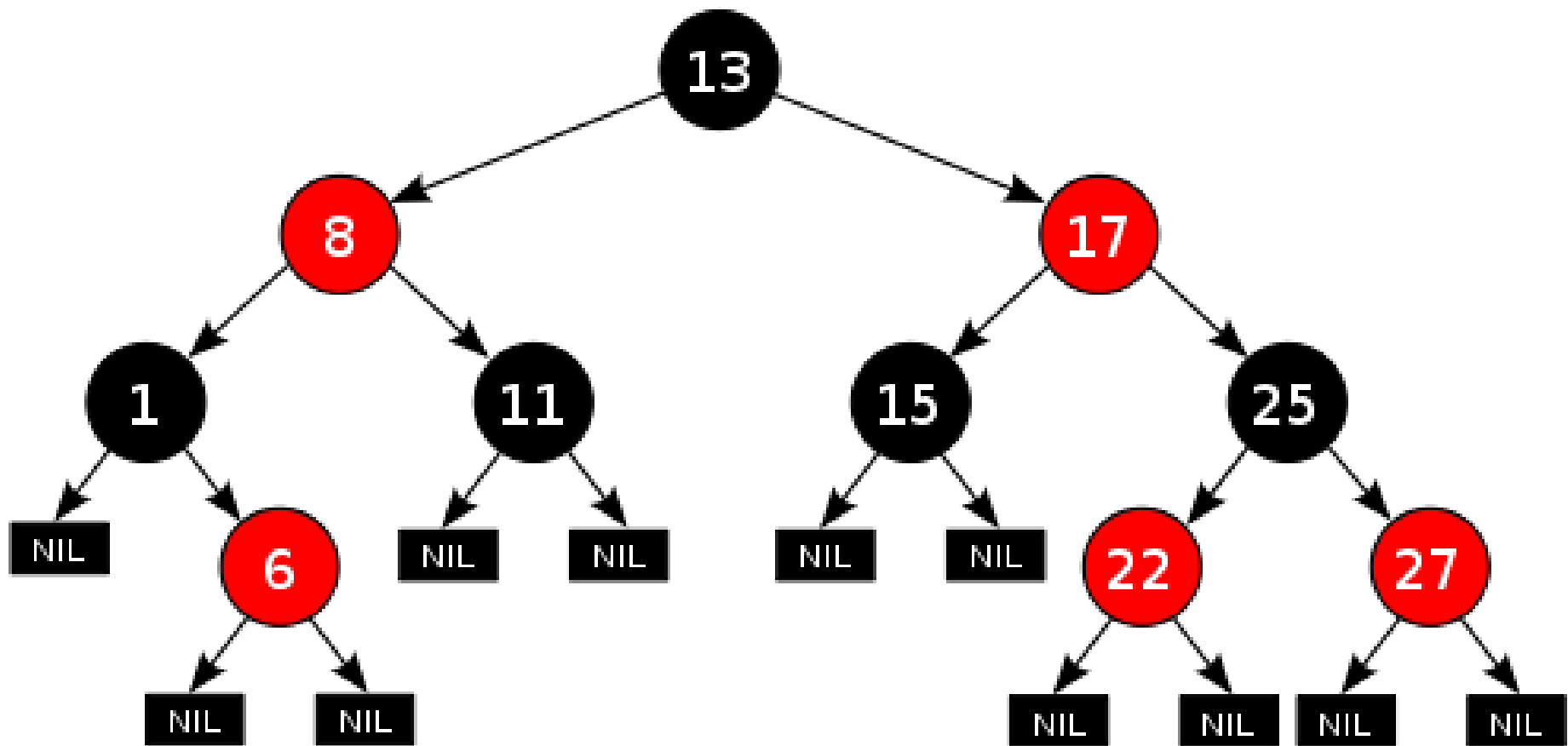


Red-Black Trees



Red-Black Tree Properties

(Definition of RB Trees)

A red-black tree is a BST with following properties:

1. Every node is either **red** or black.
2. The root is black.
3. Every leaf is NIL and black.
4. Both children of each **red** node are black.
5. All root-to-leaf paths contain the same number of black nodes.

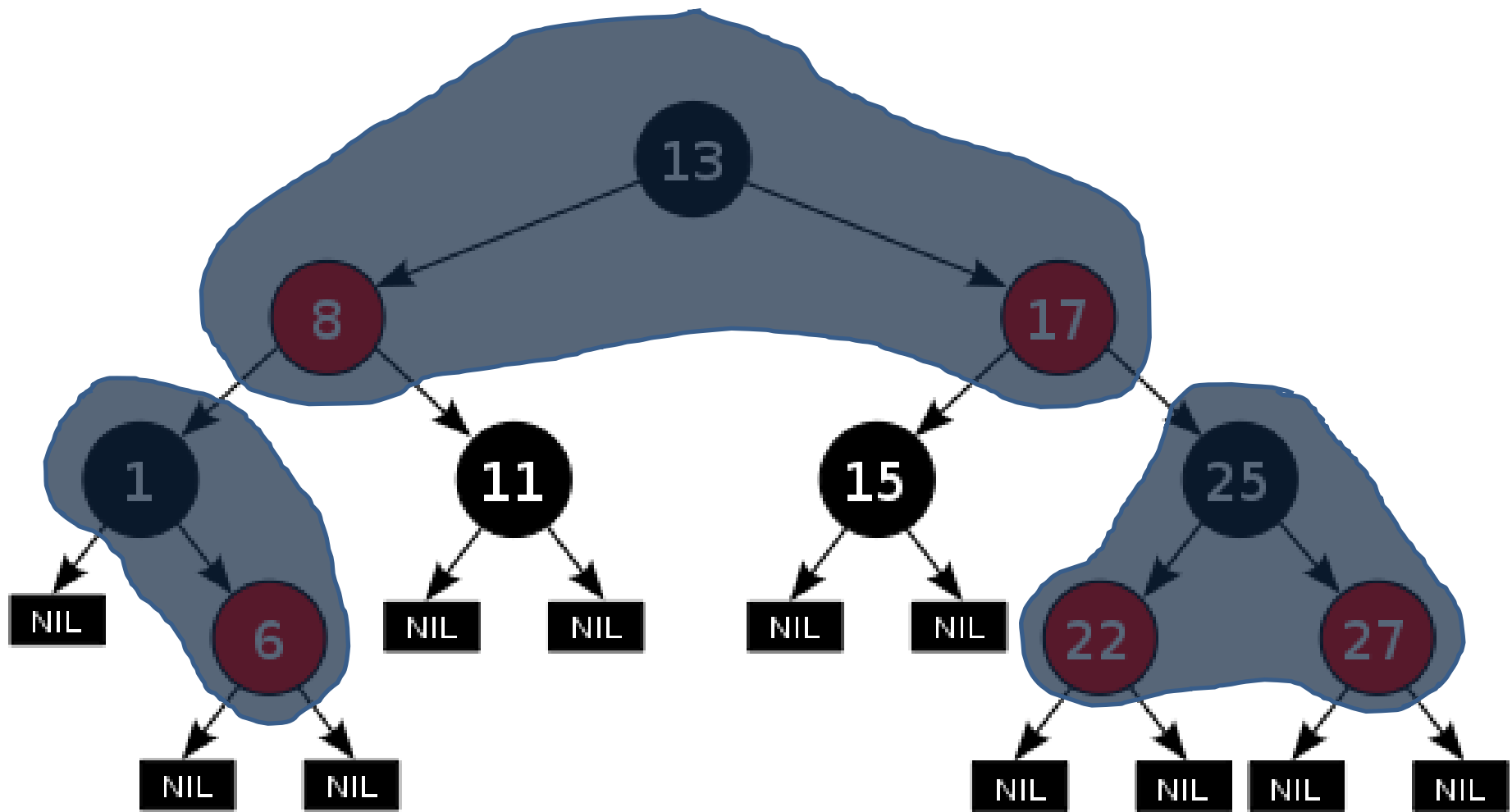
More Properties

1. No root-to-leaf path contains two consecutive red nodes.
2. For each node x , all paths from x to descendant leaves contain the same number of black nodes. This number, not counting x , is the black height of x , denoted $bh(x)$.
3. No root-to-leaf path is more than twice as long as any other.

Theorem. A red-black tree with n internal nodes has height $\leq 2 \log(n + 1)$.

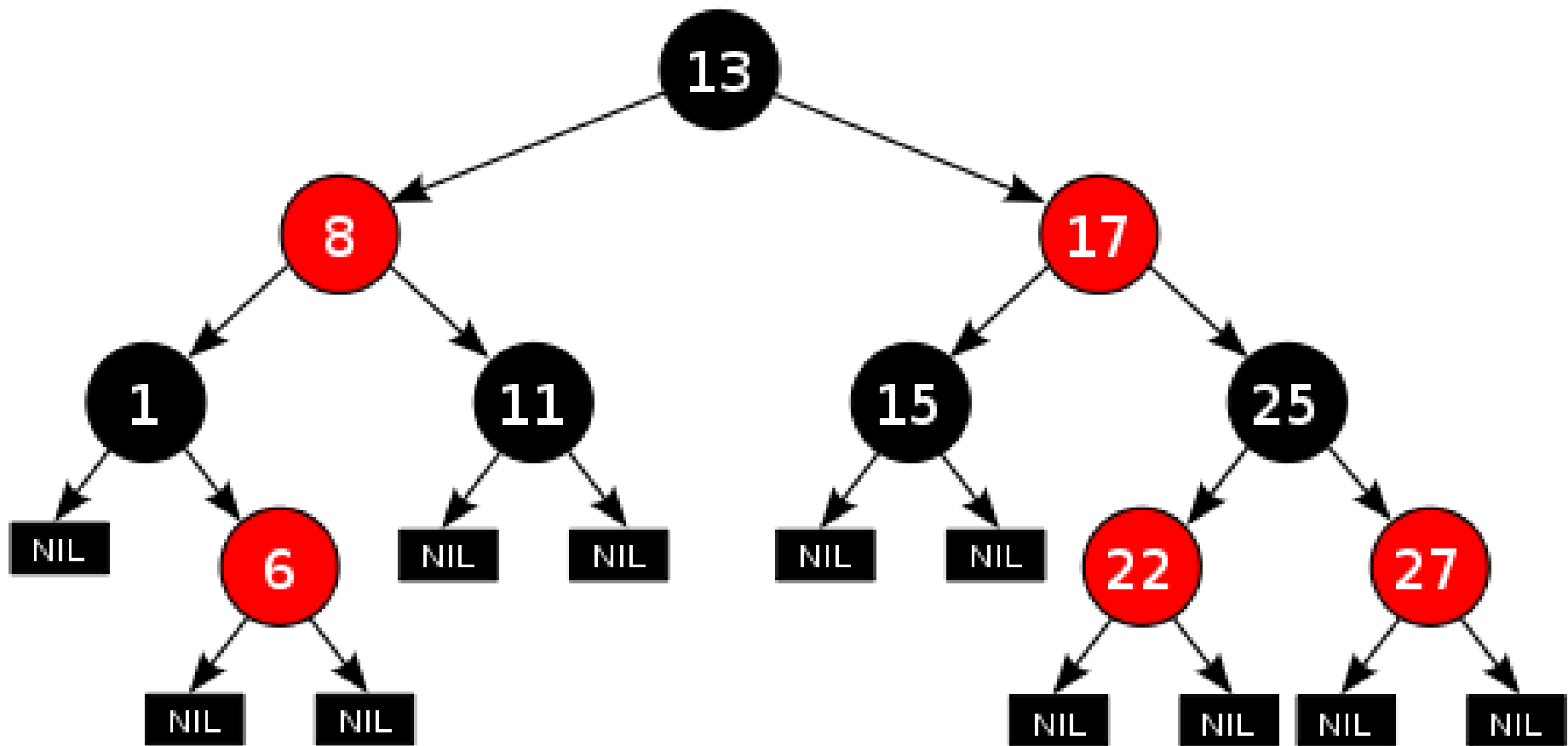
Proof of Theorem

- Consider a red black tree with height h .
- Collapse all red nodes into their (black) parent nodes to get a tree with all black nodes.
- Each internal node has 2 to 4 children.
- The height of the collapsed tree is $h' \geq h / 2$, and all external nodes are at the same level.
- Number of internal nodes in collapsed tree is
$$n \geq 1 + 2 + 2^2 + \dots + 2^{h'-1} = 2^{h'} - 1 \geq 2^{h/2} - 1.$$
- So, $h \leq 2\log_2(n + 1)$.



Insert a node z

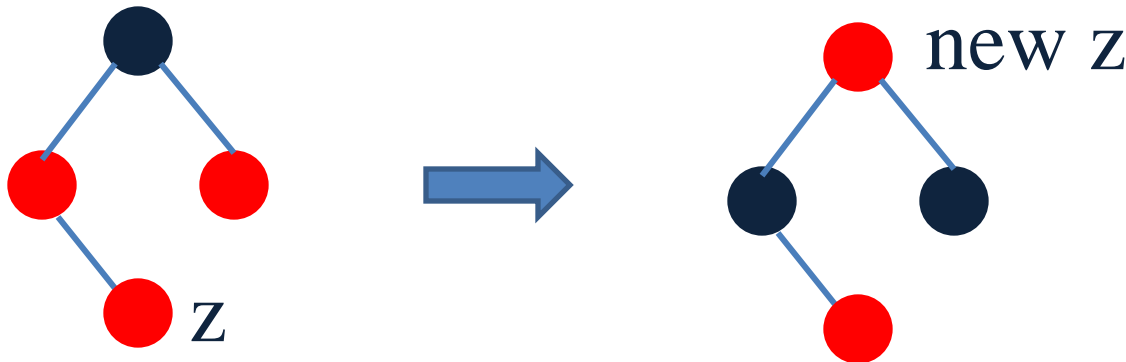
- Insert z as in a regular BST; **color it red**.
- If any violation to RB properties, fix it.
- Possible violations:
 - **The root is red. (Case 0)**
To fix up, make it black.
 - **Both z and z 's parent are red.**
To fix up, consider three cases. (Actually, six cases: I, II, III, I', II', III')



Insert Fixup: Case I

The parent and “uncle” of z are both red:

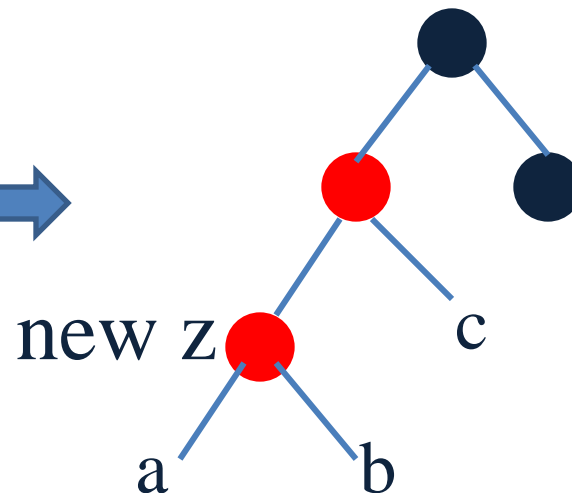
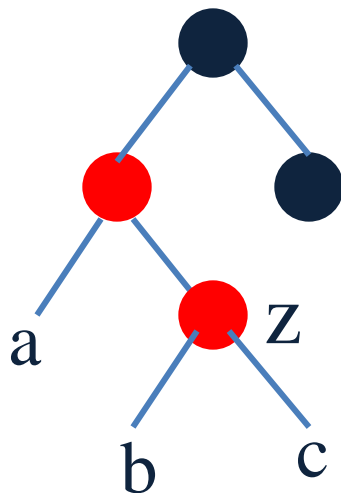
- Color the parent and uncle of z black;
- Color the grandparent of z red;
- Repeat on the grandparent of z.



Insert Fixup: Case II

The parent of z is red, the uncle of z is black, z's parent is a left child, z is a right child:

- Left Rotate on z's parent;
- Make z's left child the new z; it becomes Case III.

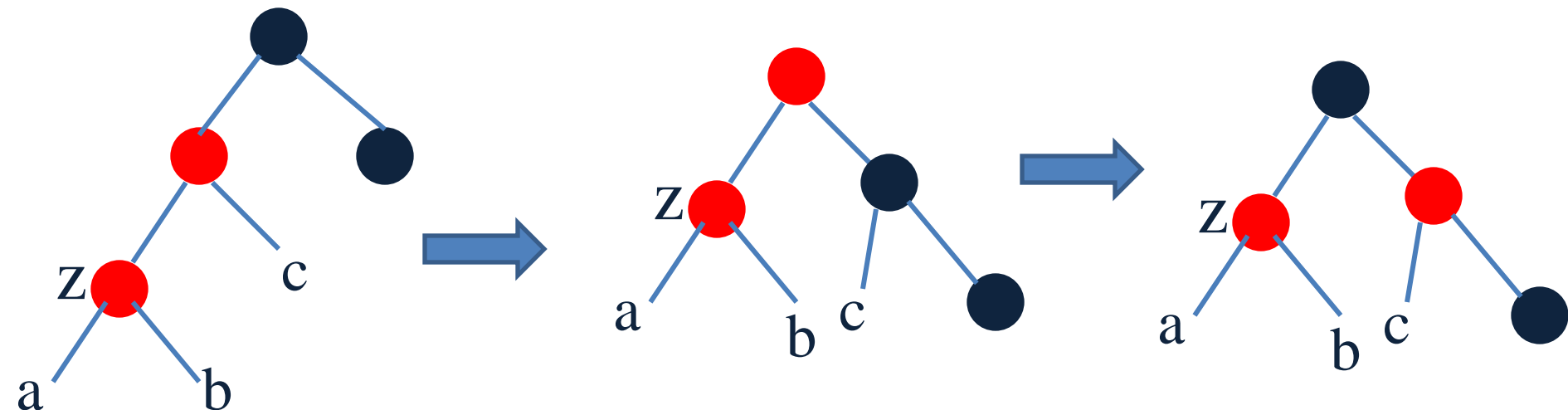


Case III

Insert Fixup: Case III

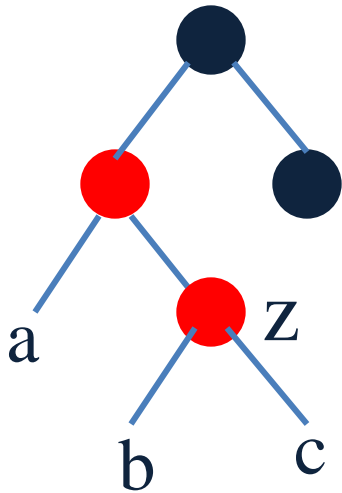
The parent of z is Red and the “uncle” is Black, z is a left child, and its parent is a left child:

- Right Rotate on the grandparent of z .
- Switch colors of z 's parent and z 's sibling.
- Done!

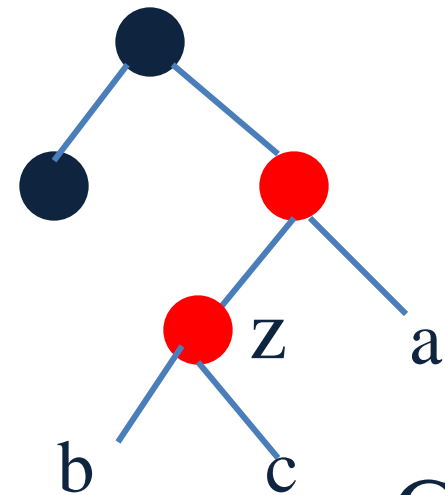


Case II'

Symmetric to Case II



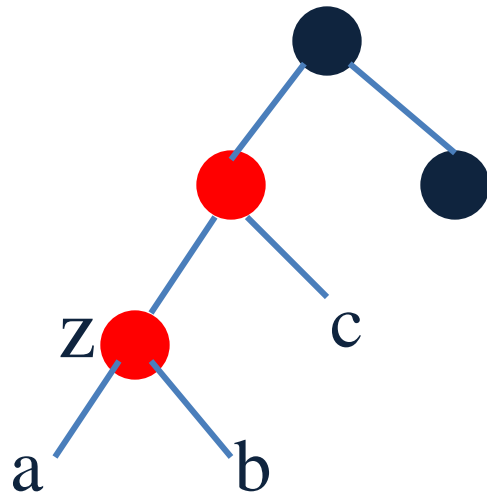
Case II



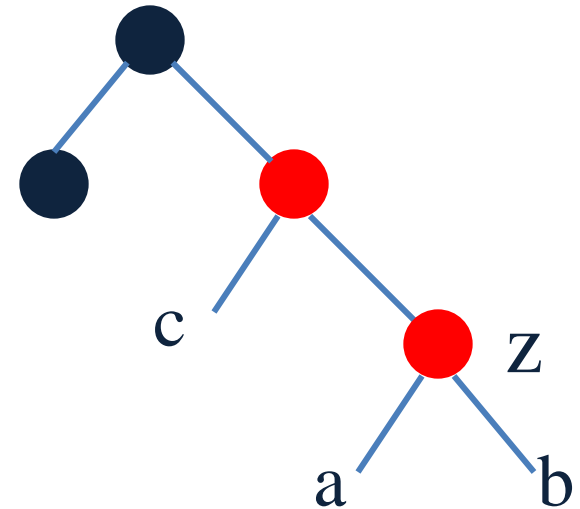
Case II'

Case III'

Symmetric to Case III



Case III



Case III'

Demonstration

- <http://www.ece.uc.edu/~franco/C321/html/RedBlack/redblack.html>

BST Deletion Revisited

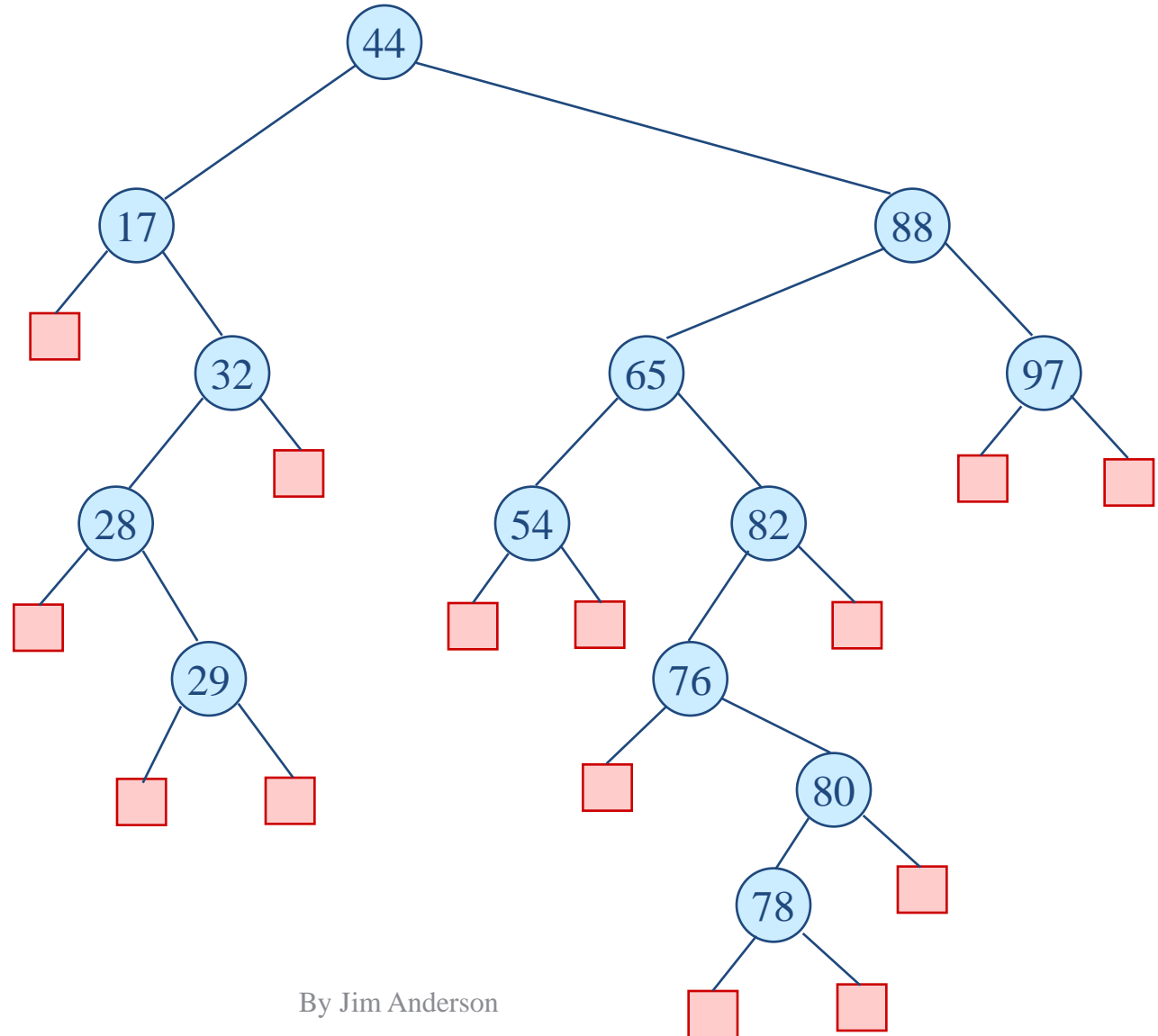
Delete z

- If z has no children, we just remove it.
- If z has only one child, we splice out z.
- If z has two children, we splice out its successor y, and then replace z's key and satellite data with y's key and satellite data.
- Which physical node is deleted from the tree?

BST: Delete

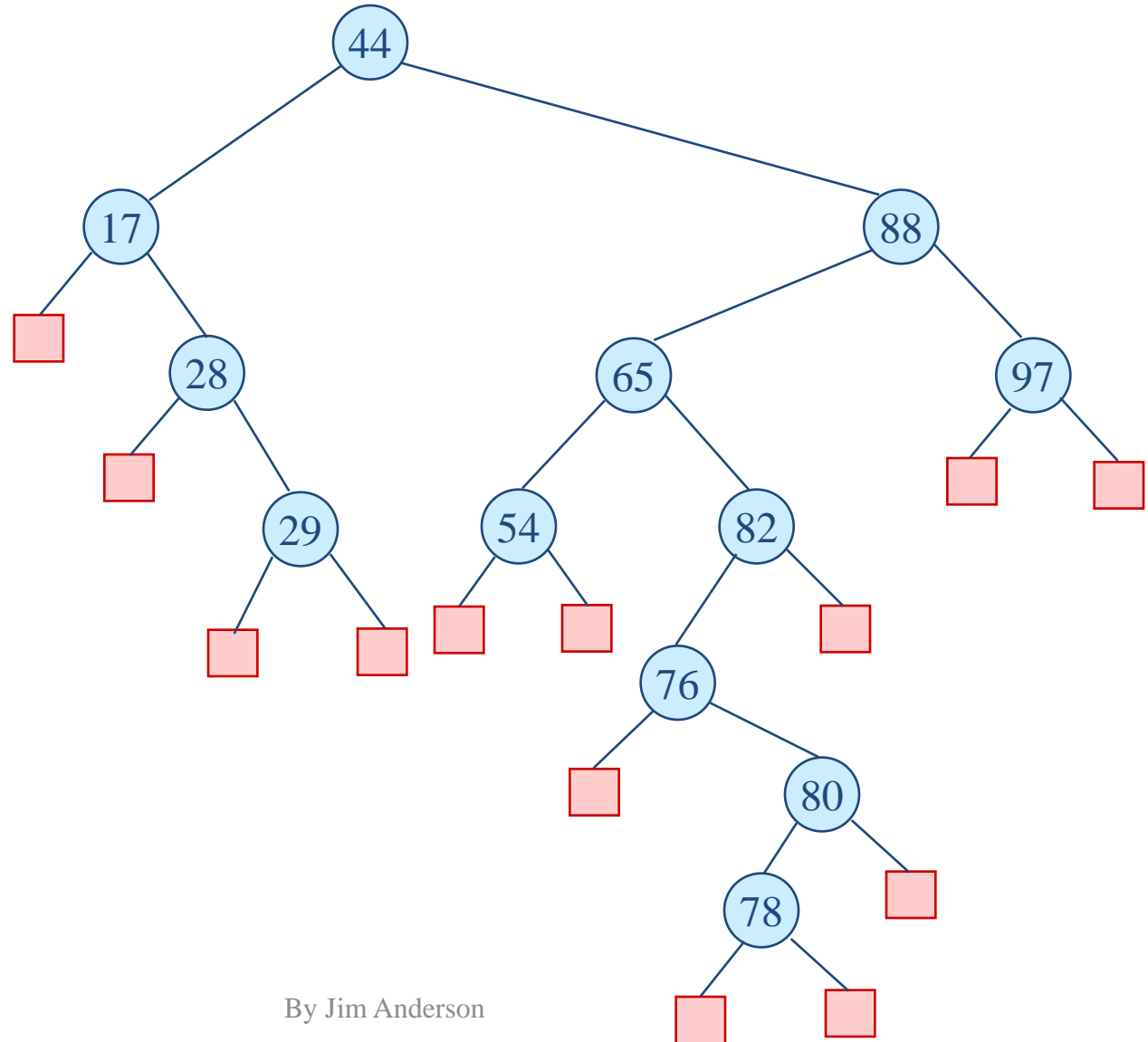
Delete(32)

Has only one
child: just splice
out 32.



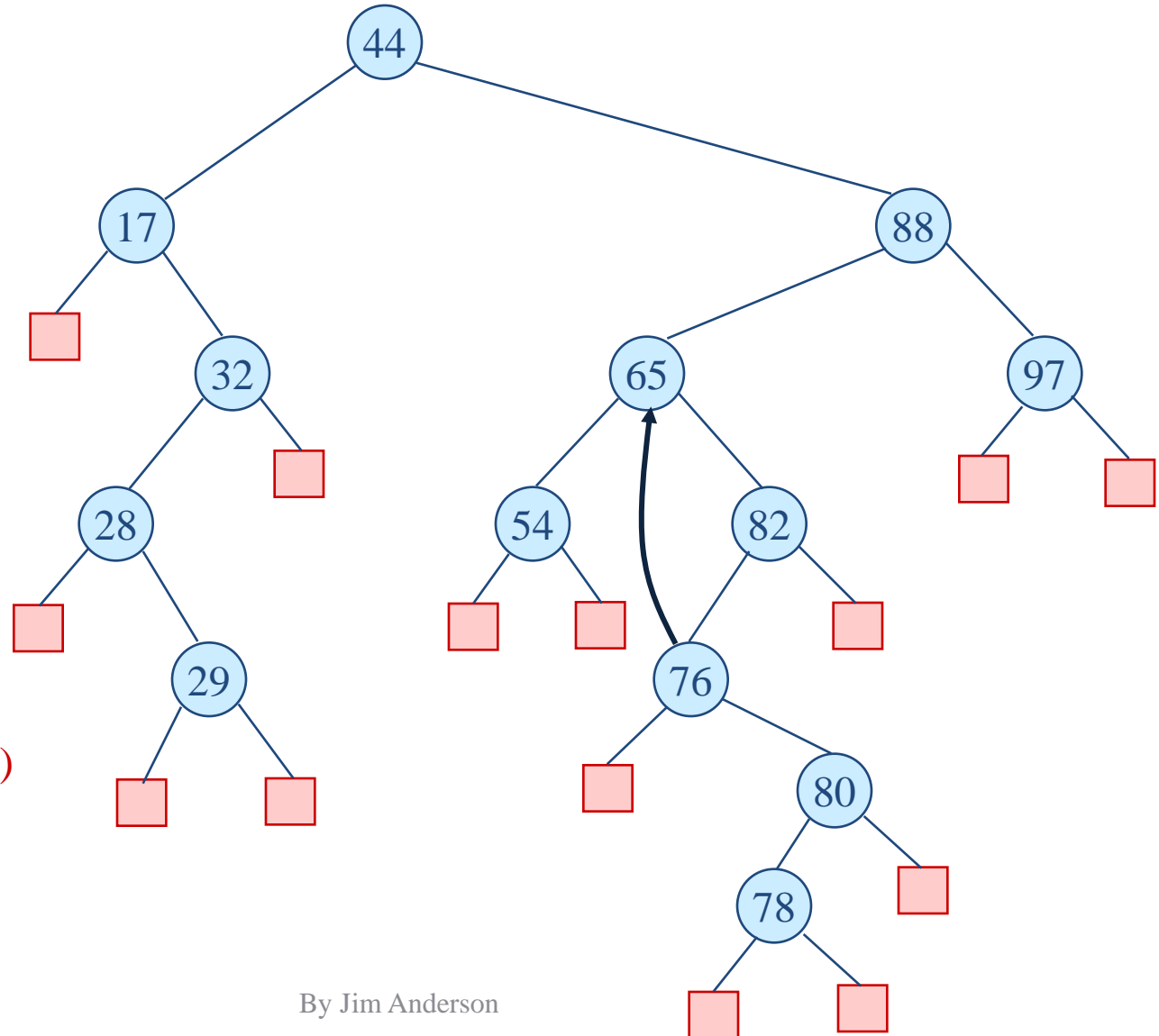
BST: Delete

Delete(32)



BST: Delete

Delete(65)

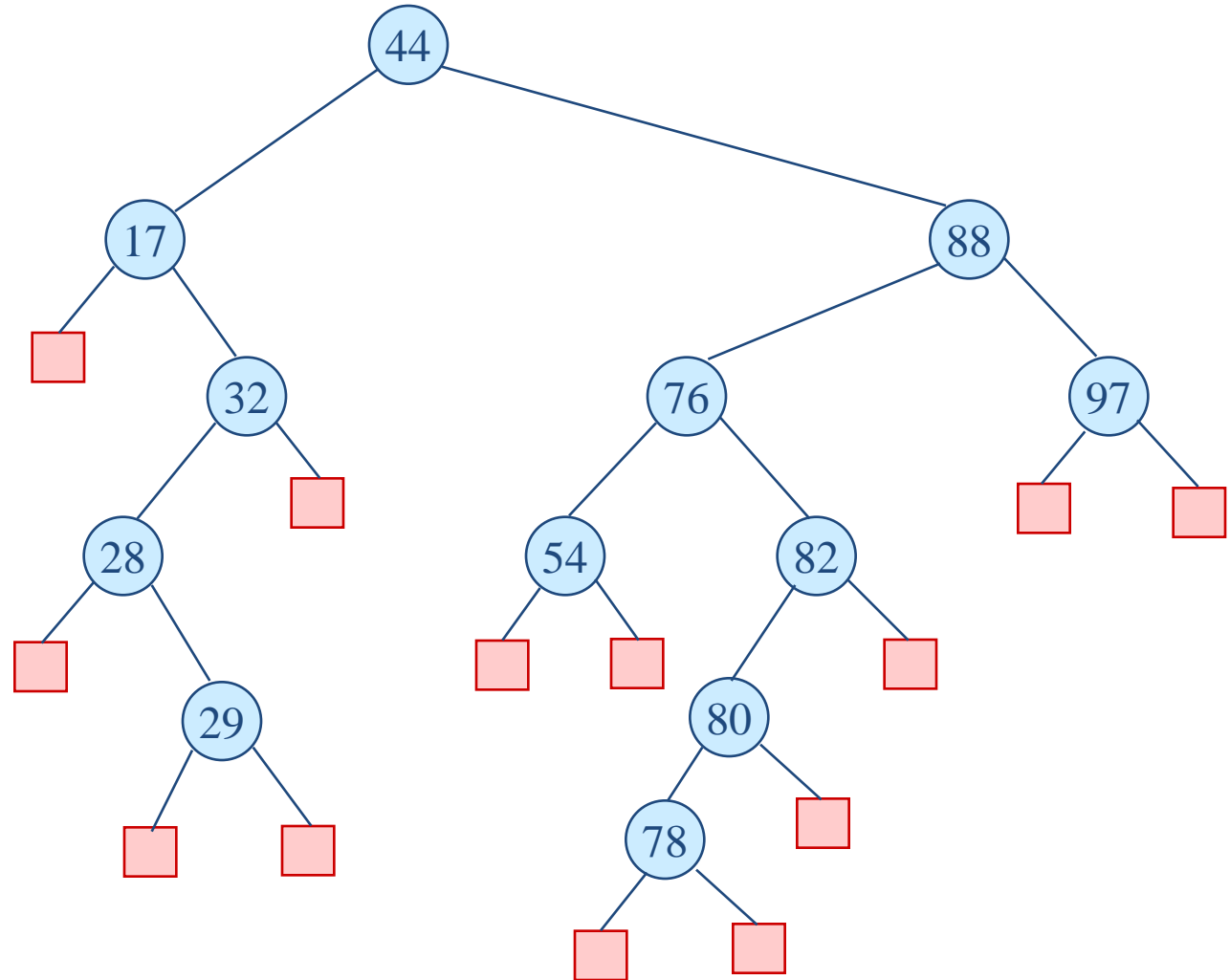


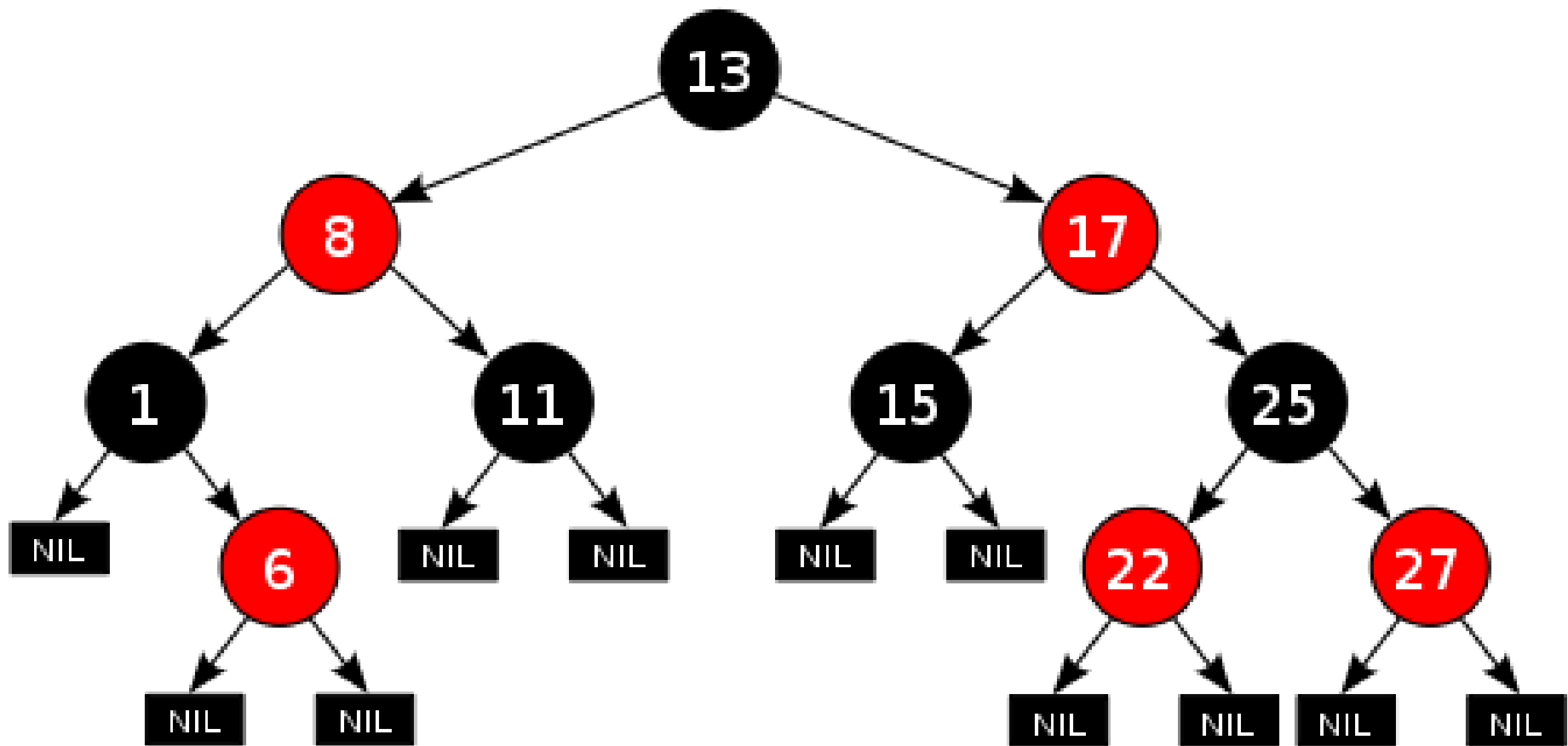
Has two children:
Replace 65 by successor,
76, and splice out
successor.

Note: Successor can have
at most one child. (Why?)

BST: Delete

Delete(65)

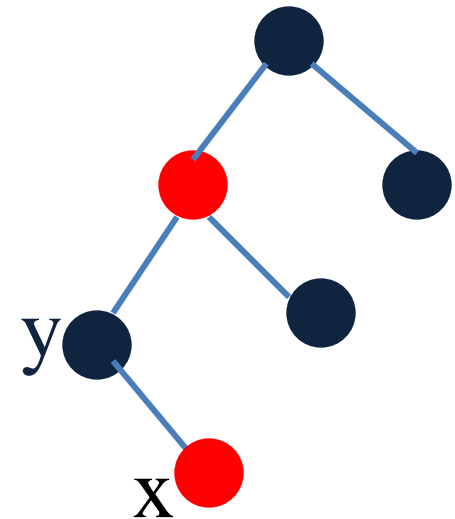
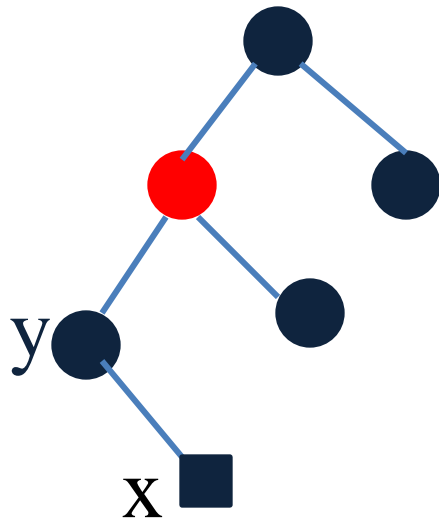




Delete z

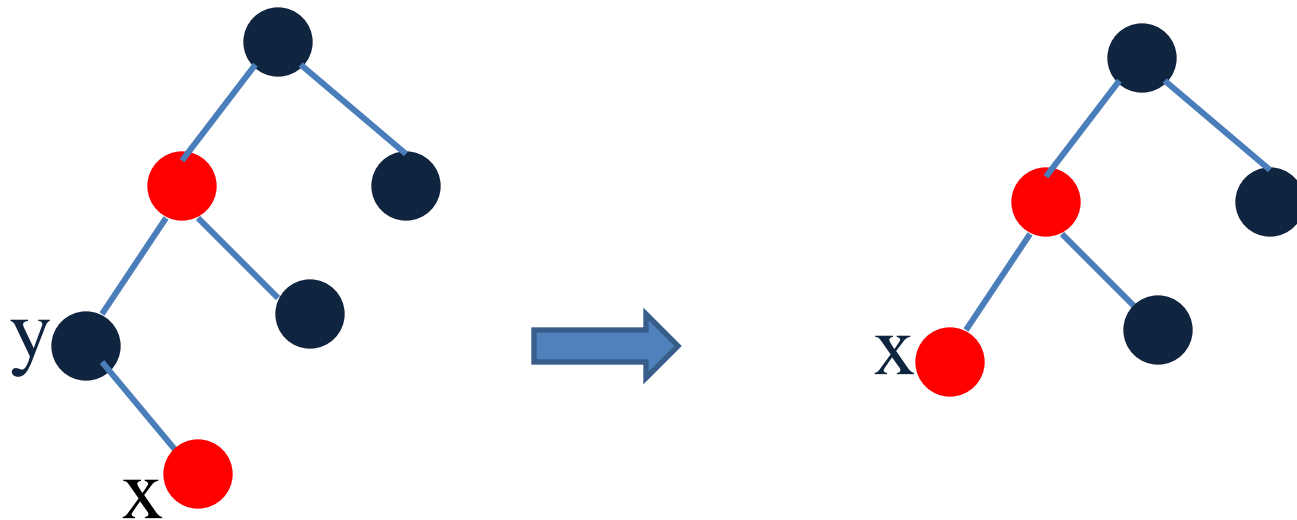
- Delete z as in a regular BST.
- If z had two (non-nil) children, when copying y 's key and satellite data to z , do not copy the color, (i.e., **keep z 's color**).
- Let y be the node being removed or spliced out.
(Note: either $y = z$ or $y = \text{successor}(z)$.)
- If **y is red**, no violation to the red-black properties.
- If **y is black**, then one or more violations may arise and we need to restore the red-black properties.

- Let x denote the child of y before it was spliced out.
- x is either nil (leaf) or was the only non-nil child of y .

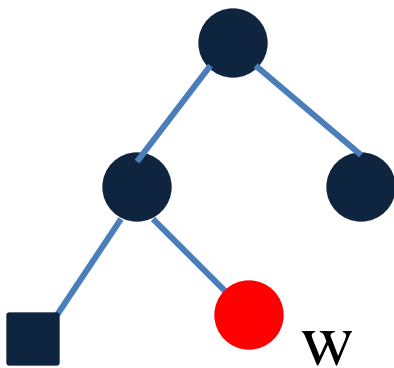
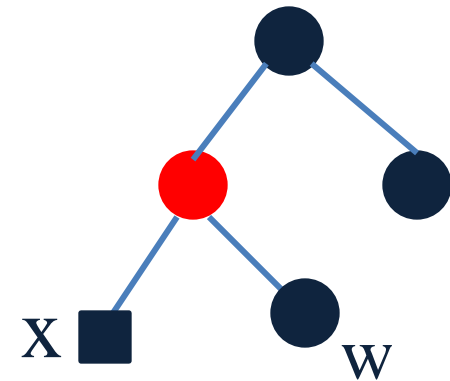
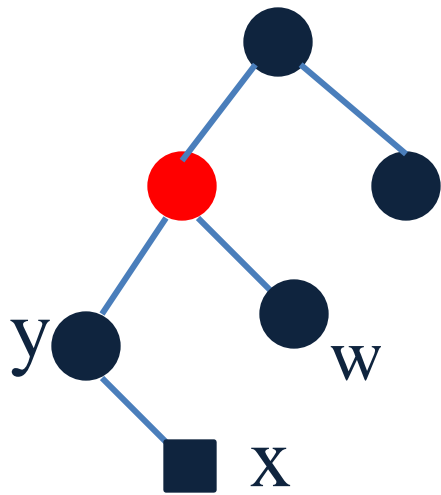


Restoring RB Properties

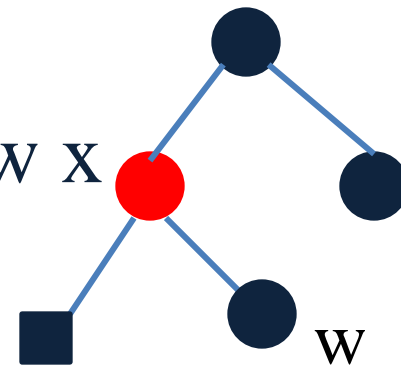
- Easy: If x is red, just change it to black.
- More involved: If x is black.



Example: x is black



new x



Restoring RB Properties

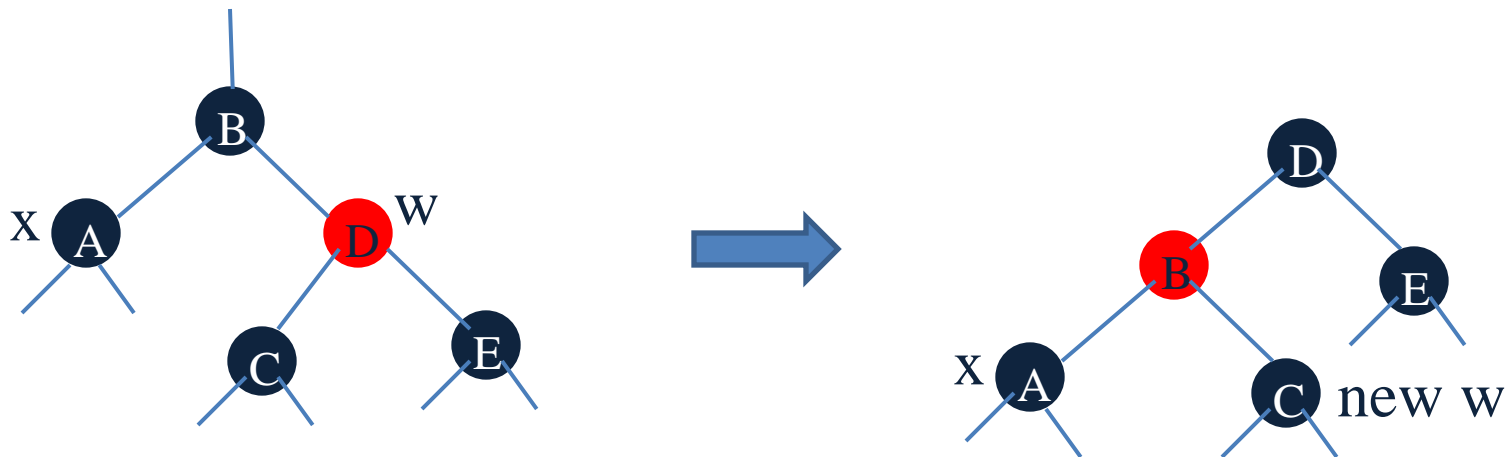
- Assume: x is black and is a left child.
- The case where x is black and a right child is similar (symmetric).
- Four cases:
 1. x 's sibling w is red.
 2. x 's sibling w is black; both children of w are black.
 3. x 's sibling w is black; left child of w is red, right child black.
 4. x 's sibling w is black; right child of w is red.

Main idea

- Regard the **pointer x** itself as black.
- Counting x, the tree satisfies RB properties.
- Transform the tree and move x up until:
 - x points to a red node, or
 - x is the root, or
 - RB properties are restored.
- At any time, maintain RB properties, with x counted as black.

x is a black left child: Case 1

- x' sibling w is red.
- Left rotate on B; change colors of B and D.
- Transform to Case 2, 3, or 4 (where w is black).



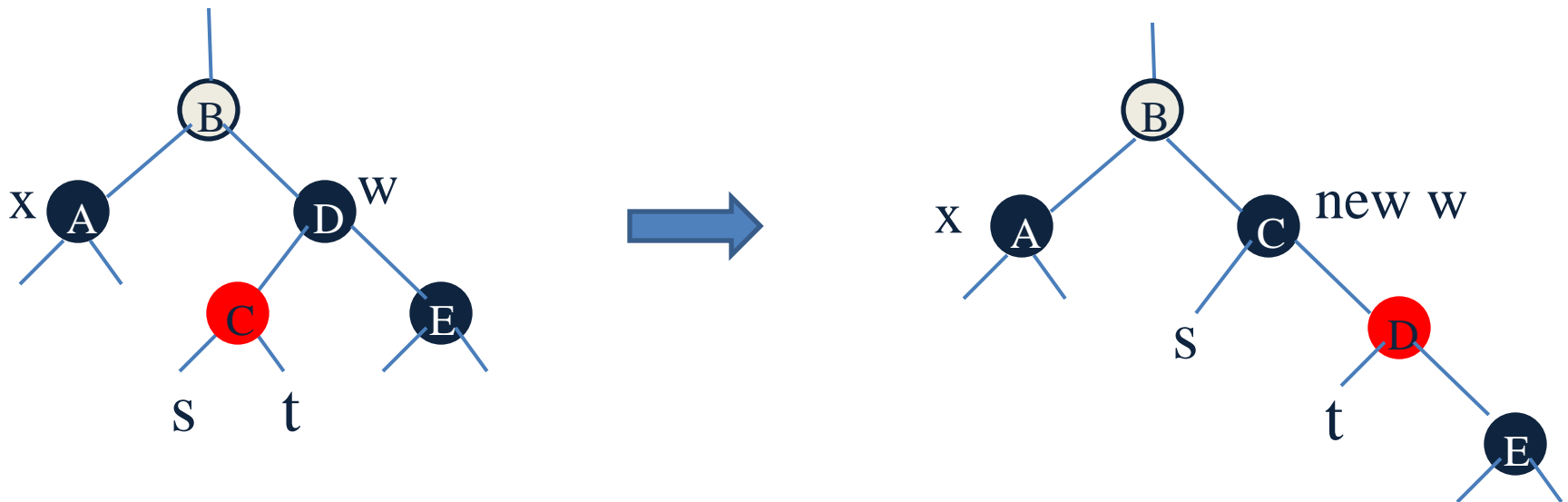
x is a black left child : Case 2

- x's sibling w is black; both children of w are black.
- Move x up, and change w's color to red.
- If new x is red, change it to black; else, repeat.



x is a black left child : Case 3

- x's sibling w is black; w's left child is red, right child black.
- Right rotate on w (D); switch colors of C and D.
- C becomes the new w.
- Transform to Case 4.



X is a black left child : Case 4

- x's sibling w is black; w's right child is red.
- Left rotate on B; switch colors of B and D; change E to black.
- Done!

