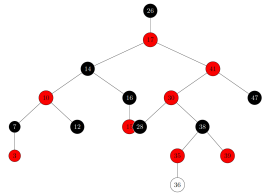


Red-Black Trees 3 - Deletion
Lecture 17

COSC 242 – Algorithms and Data Structures

Today’s outline

1. Delete
2. Properties
3. Delete fixup
4. Cases
5. Examples

Today’s outline

1. Delete
2. Properties
3. Delete fixup
4. Cases
5. Examples

Overview

Like other RBT operations, deletion takes $O(\log_2 n)$ time.

Deleting a node from an RBT is more complicated than insertion.

The procedure for deletion is a modification from `BST_delete` we saw in L14. The rebalancing and enforcement of RBT properties will happen in a new sub-procedure called `RBT_Delete_Fixup`.

This is the same approach as `RBT_Insert`, itself an extension of `BST_Insert` (L12), which used `RBT_Insert_Fixup` to maintain RBT properties.

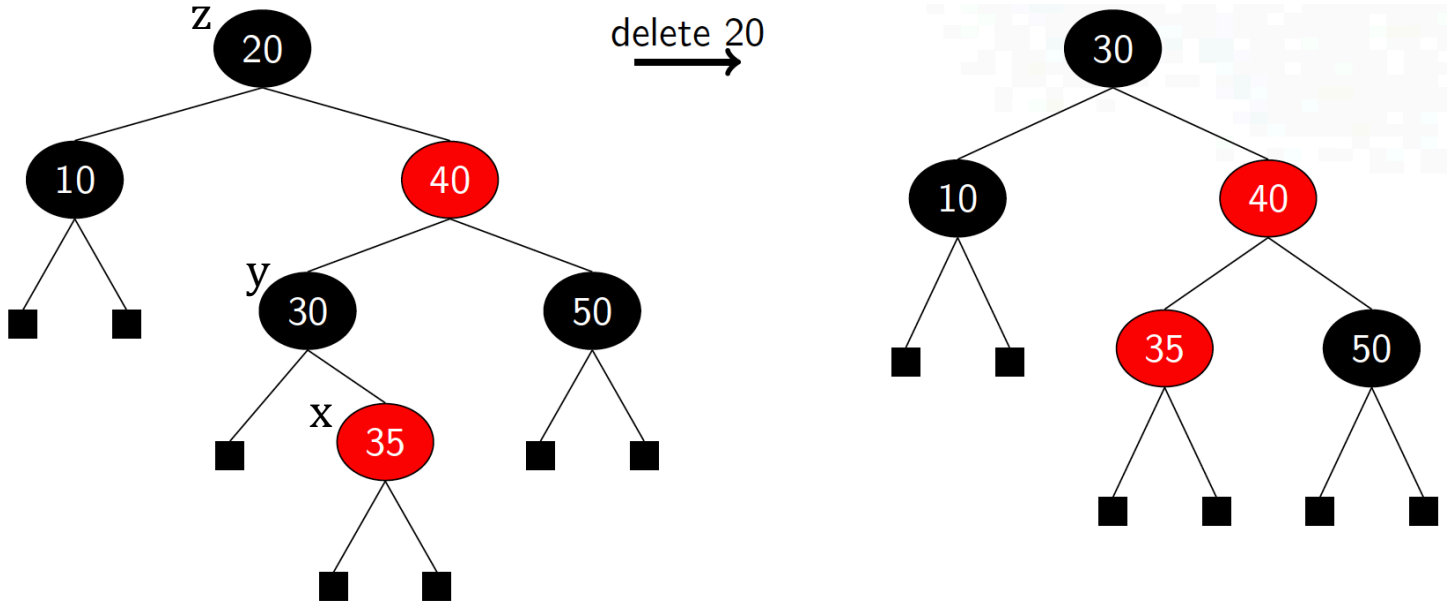
Refresh: BST Delete

To delete a node z, BST_delete recursively searches for z, and then:

1. If z has < 2 children, replace it by a child (possibly nil); or
2. If z has == 2 children, replace it by its successor

Some node, call it y, eventually gets spliced out.

It may be that $y = z$, or y may be z's successor.



Here z has two children, so y is z's successor, 30. y gets spliced out.
x replaces y

RBT deletion

To delete a node z in an RBT:

1. Delete z as for a BST
2. Fix any RBT violations

Call the spliced out node y. The spliced out node is the node that is removed from the tree.

Sometimes y and z point to the same node (e.g., when z has one child).

Sometimes y and z are different, as we saw on the previous slide.


```
function RBT_delete(T, z)
1:   if z→left == NIL or z→right == NIL then
2:       y = z                              // Only one child
3:   else
4:       y = BST_RW_Successor(z)            // Two children, assign successor
5:   if y→left != NIL then                  // Assign x to y's left or right child
6:       x = y→left
7:   else
8:       x = y→right
9:   x→parent = y→parent
10:  if y→parent == NIL then                // y is root, so
11:      T→root = x                         // make root x, otherwise
12:  else if y == y→parent→left then        // Splice out y, assign x to parent subtree
13:          y→parent→left = x              // left child
14:      else                               // or
15:          y→parent→right = x             // right child
16:  if y != z then
17:      z→key = y→key                      // copy y's remaining data into z
18:  if y.colour == BLACK then
19:      RBT_Delete_Fixup(T, x)             // If y was black, then fixup is needed
end function
```

Today’s outline

1. Delete
2. Properties
3. Delete fixup
4. Cases
5. Examples

Refresh: RBT Properties

1. Every node is either **red** or **black**.
2. The root is **black**.
3. Every leaf (nil/null) is **black**.
4. If a node is **red**, then both its children are **black***.
5. For each node, all paths from the node to leaves contain the same number of **black** nodes.

Questions

If y is **red**, can any of those properties be violated?

If y is **black**, can any of those properties be violated?

When y is red

When y is **red**, the RBT properties still hold when y is spliced out:

- No black-heights in the tree have changed
- No red nodes have been made adjacent (reminder: property 4, all children of a red node are black)
- Since y could not have been the root if it was red, the root remains black

When y is **black**

1. Every node is either red or black.
2. The root is black.
3. Every leaf (nil/null) is black.
4. If a node is red, then both its children are black.
5. For each node, all paths from the node to leaves contain the same number of black nodes.

When y is **black**, three problems may arise:

1. If y was root, and a red child of y becomes the root, we have violated property 2.
2. If both x and x→parent are **red**, we have violated property 4.
3. Moving y within the tree causes any simple path that previously contained y to have one fewer black nodes, violating property 5.

Hence, Line 19 call to `RBT_Delete_Fixup` is made only if y was black.

Correction

We can correct the violation of P5 by saying that node x, now occupying y's original position, has an "extra" black (+1BH).

That is, if we add +1 to the count of black nodes on any simple path that contains x, then Property 5 will hold.

In essence, when y is removed, we "push" it's blackness onto node x.

1. Every node is either red or black.
2. The root is black.
3. Every leaf (nil/null) is black.
4. If a node is red, then both its children are black.
5. For each node, all paths from the node to leaves contain the same number of black nodes.

Correction

1. Every node is either red or black.
2. The root is black.
3. Every leaf (nil/null) is black.
4. If a node is red, then both its children are black.
5. For each node, all paths from the node to leaves contain the same number of black nodes.

The problem is that now node x is neither **red** nor **black**, thereby violating property 1.

Instead, x is either "**doubly black**" or "**red-and-black**", and it contributes either 2 or 1, respectively, to the count of black nodes on simple paths containing x.

Correction

The colour attribute of x will still be **RED** (if x is red-black) or **BLACK** (if x is double black). Pay attention to this in the pseudocode.

The +1 black on a node is reflected in x's pointing to the node, rather than in the colour attribute.

That is, you won't see "red-black" or "double black" in the code; it's what x is pointing to that conveys the +1 black.

Today’s outline

1. Delete
2. Properties
3. Delete fixup
4. Cases
5. Examples

Labels

Let's start by defining some labels:

- z is the node to be deleted
- y is the node that gets spliced out (sometimes $y = z$ and sometimes y is z's successor)
- x is the child that replaced y
- w is the new sibling of x

Four cases to handle

1. x's sibling, w, is **red**. Fix then fall to case 2, 3, or 4.
2. w is **black** and has two black children. Fix then traverse up the tree. If fell through from case 1, terminate.
3. w is **black** and w's left child (or "inner" child) is **red** and right child ("outer") is **black**. Fix then fall to case 4.
4. w is **black** and w's right child ("outer") is **red**. Fix and terminate.

Here, "outer" and "inner" refer to w's child, and its position with respect to x.


```
function RBT_delete_fixup(T, x)
1:  while x != root and x.colour == BLACK
2:      if x == x→parent→left                    // Symmetric with right-child, L22 else
3:          w = x→parent→right                   // Set sibling
4:          if w.colour == RED                   // Is x-sibling red?
5:              w.colour = BLACK                 // Case 1
6:              x→parent.colour = RED            // Case 1
7:              Left_rotate(T, x→parent)         // Case 1
8:              w = x→parent→right               // Case 1
9:          if w→left.colour == BLACK and w→right.colour == BLACK // Both children black?
10:             w.colour = RED                   // Case 2
11:             x = x→parent                     // Case 2
12:         else if w→right.colour == BLACK      // Left is red, right is black?
13:                 w→left.colour = BLACK        // Case 3
14:                 w.colour = RED               // Case 3
15:                 Right_rotate(T, w)           // Case 3
16:                 w = x→parent→right           // Case 3
17:             w.colour = x→parent.colour       // Case 4 (fall through)
18:             x→parent.colour = BLACK          // Case 4
19:             w→right.colour = BLACK           // Case 4
20:             Left_rotate(T, x→parent)         // Case 4
21:             x = T→root                       // Case 4
22:     else ... (same as L2 if clause, with “right” and “left” exchanged)
23: x.colour = BLACK
end function
```

Today’s outline

1. Delete
2. Properties
3. Delete fixup
4. Cases
5. Examples

Fixup

The main job of `RBT_Delete_Fixup` is to restore RBT property 1.

This is achieved through the `while` loop Lines 1-22 by moving the extra black up the tree until:

1. x points to a red-and-black node. We colour x (singly) **black** in L23. (Recall SL15: the colour attribute of "red-and-black" is **red**, thus breaking the while loop); or
2. x points to the root (on L11 or L21). We "remove" the extra black; or
3. Having performed suitable rotations and re-colourings, we exit the loop.

Remember: If x is a right-child, flip all left and rights. You may prefer to think of w's children as "outer" and "inner" with respect to x.

Notes about fixup

Within the while loop, x always points to a non-root **doubly black** node.

The code handles 4 cases. The transformation in each case preserves the number of black nodes (including x's extra black) from - and including - the root of the subtree shown to each of the subtrees $\alpha, \beta, \gamma, \delta, \varepsilon, \zeta$.

Thus, if property 5 held prior to transformation, it continues to hold afterwards.

In our next set of figures, c an c' refers to a node that could be either **red** or **black** (either could happen, and neither is required).

Case 1 – x's sibling w is red

Operations

- Switch colours of w and x's parent (L5, 6)
- Left-rotate on x's parent (L7)
- x's new sibling is now black (L8)

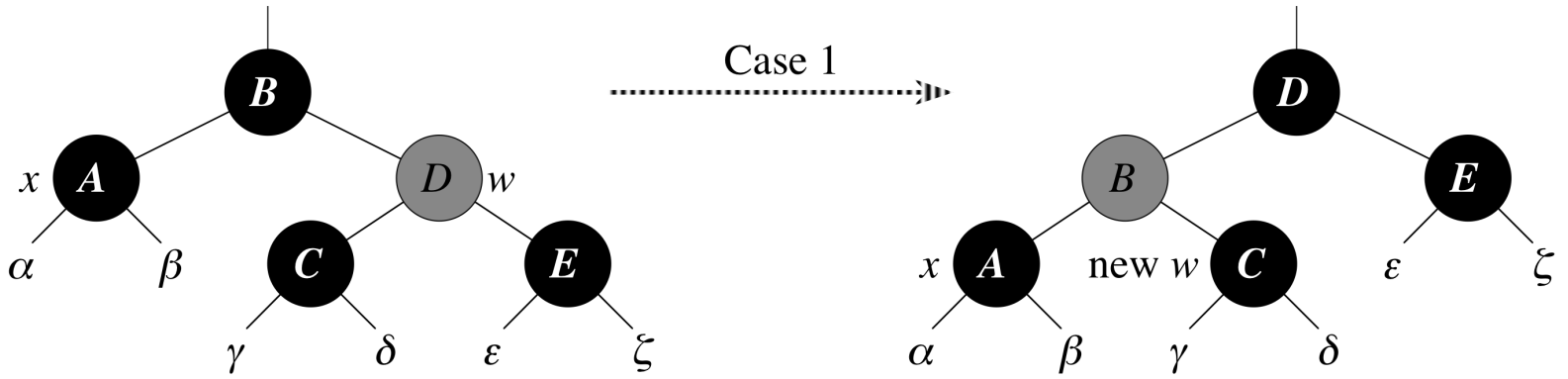
Flow

- Doesn't terminate.
- May fall through to cases 2, 3, or 4.

Children

$\alpha, \beta = 3$ (B + 2A)

$\gamma, \delta, \varepsilon, \zeta = 2$ (B + C or E)




```
procedure RBT_delete_fixup(T, x)
1:   while x != root and x.colour == BLACK
2:       if x == x→parent→left  // Symmetric with right-child
3:           w = x→parent→right
4:           if w.colour == RED              // Is x-sibling red?
5:               w.colour = BLACK            // Case 1
6:               x→parent.colour = RED       // Case 1
7:               Left_rotate(T, x→parent)    // Case 1
8:               w = x→parent→right          // Case 1
```

Case 2 - w is black, both of w's children black

Operations

- Make w red (L10)
- Make x-parent the new x (L11). This pushes +1 black up the tree, seeking balance.

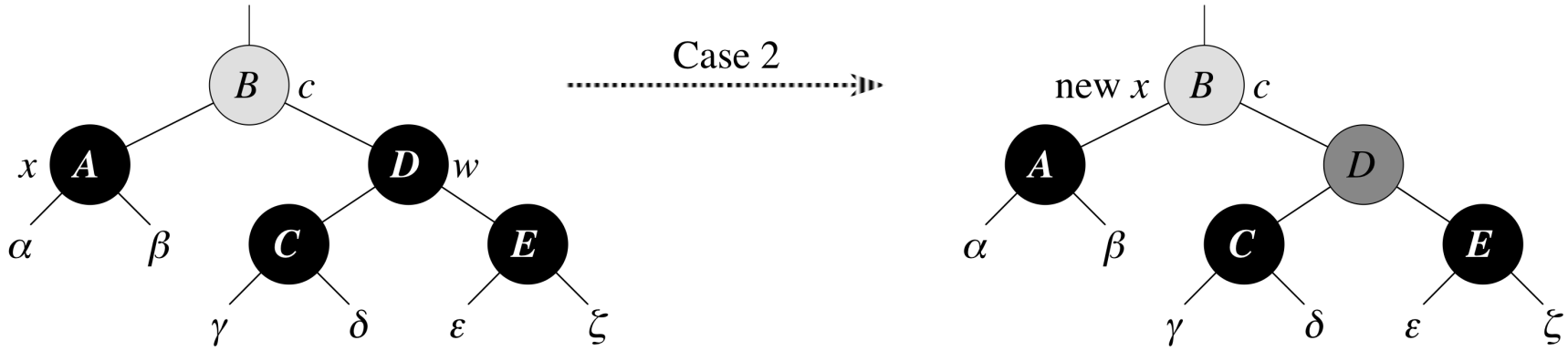
Flow

- If entered from Case 1, terminate (x = red-black)

Children

$\alpha, \beta = 2\ or\ 3$ (B? + 2A)

$\gamma, \delta, \varepsilon, \zeta = 2\ or\ 3$ (B? + D + C or E)



Colouring

If we entered from Case 1, then B must be red. But if we didn't, then B could be either red or black.

```
    // Both children black?
9:  if w→left.colour == BLACK and w→right.colour == BLACK
10:     w.colour = RED                  // Case 2
11:     x = x→parent                    // Case 2
12: else if w→right.colour == BLACK
```

Case 3 - w is black, w's left child (inner) is red, and w's right child (outer) is black

Operations

- Make w's left (inner) child black (L13)
- Make w red (L14)
- Right-rotate on w (L15)
- Update w (x-sibling) (L16)

Flow

- Will fall into Case 4

Children

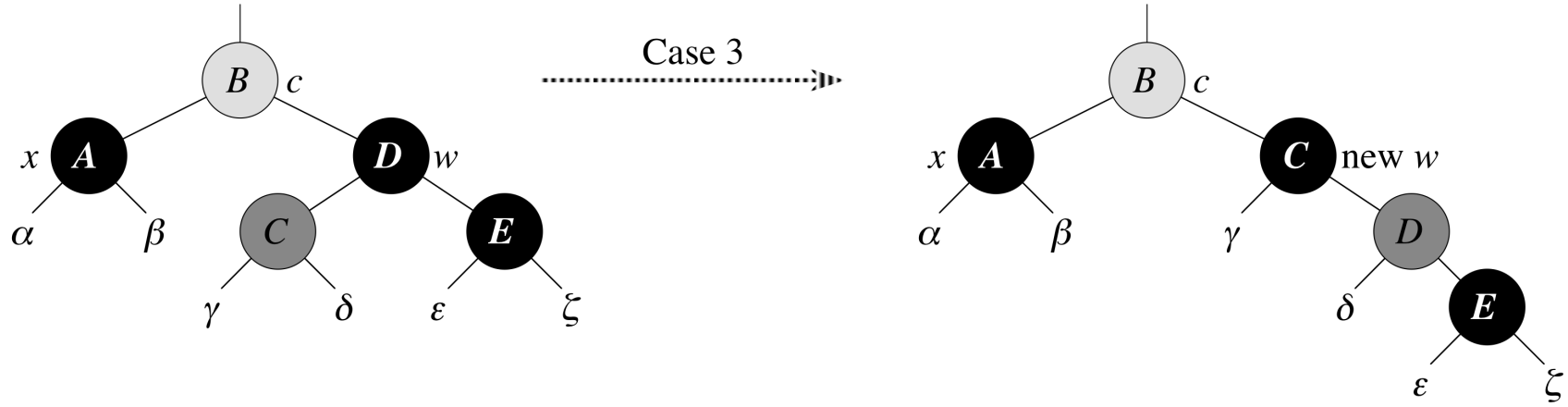
$\alpha, \beta = 2\ or\ 3$ (B? + 2A)

$\gamma, \delta = 1\ or\ 2$ (B? + D)

$\varepsilon, \zeta = 2\ or\ 3$ (B? + D + E)

Colouring

If we enter from Case 1, then B must be red.




```
12:  else if w→right.colour == BLACK   // Left is red, right is black?
13:           w→left.colour = BLACK    // Case 3
14:           w.colour = RED           // Case 3
15:           Right_rotate(T, w)       // Case 3
16:           w = x→parent→right       // Case 3
17:       w.colour = x→parent.colour   // Case 4 (fall through)
```

Case 4 - w is black, w's right child (outer) is red

Operations

- Make w x-parent colour (L17)
- Make x-parent black (L18)
- Make w's right (outer) child black
- Left-rotate x-parent (L20)
- Make x the root (L21)

Flow

- Always terminate

Children

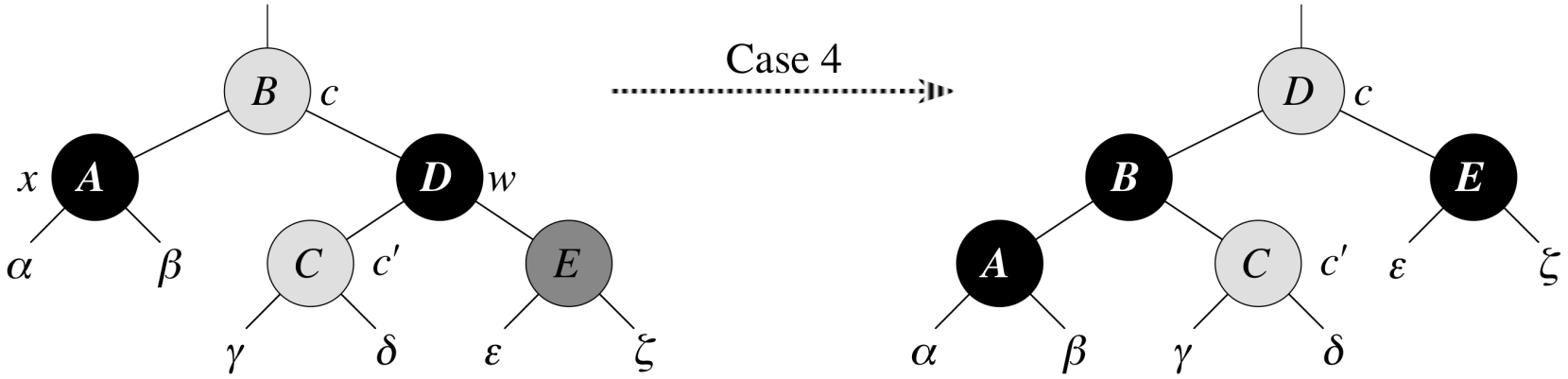
$\alpha, \beta = 1\ or\ 2$ (B? + 2A)

$\gamma, \delta = 1, 2, or\ 2$ (B? + D + C?)

$\varepsilon, \zeta = 1\ or\ 2$ (B? + D)

Colouring

If we enter from Case 1, then B must be red.
C can be either black or red, as either is allowed/possible.



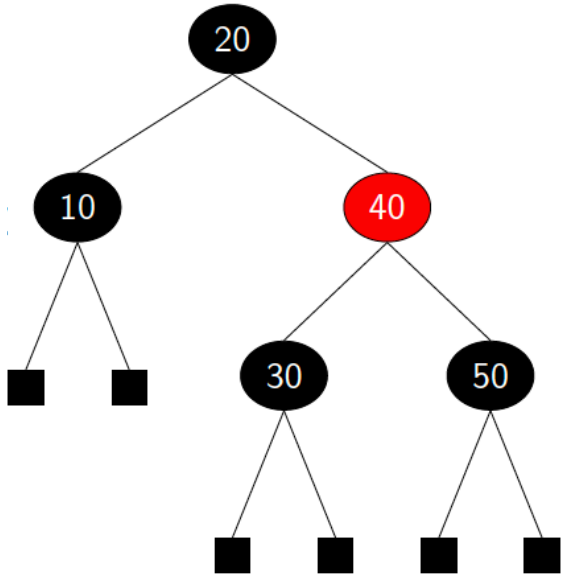

```
16:             w = x→parent→right      // Case 3
17:         w.colour = x→parent.colour  // Case 4
18:         x→parent.colour = BLACK     // Case 4
19:         w→right.colour = BLACK      // Case 4
20:         Left_rotate(T, x→parent)    // Case 4
21:         x = T→root                  // Case 4
22: else ...
```

Today’s outline

1. Delete
2. Properties
3. Delete fixup
4. Cases
5. Examples

Case 1 - Example

Delete 10

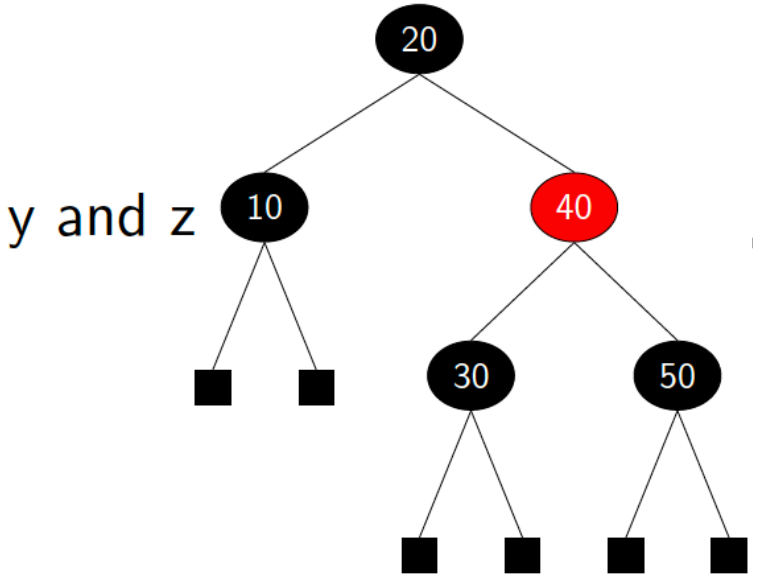


1. Identify y and z.
2. Delete z as for a BST
3. Identify w, w's children, and case
4. Fix any RBT violations

Y = spliced out node (removed from tree)
Z = Key value to remove
W = x's sibling

Case 1 - Example

Delete 10



1. **Identify y and z.**
2. Delete z as for a BST
3. Identify w, w's children, and case
4. Fix any RBT violations

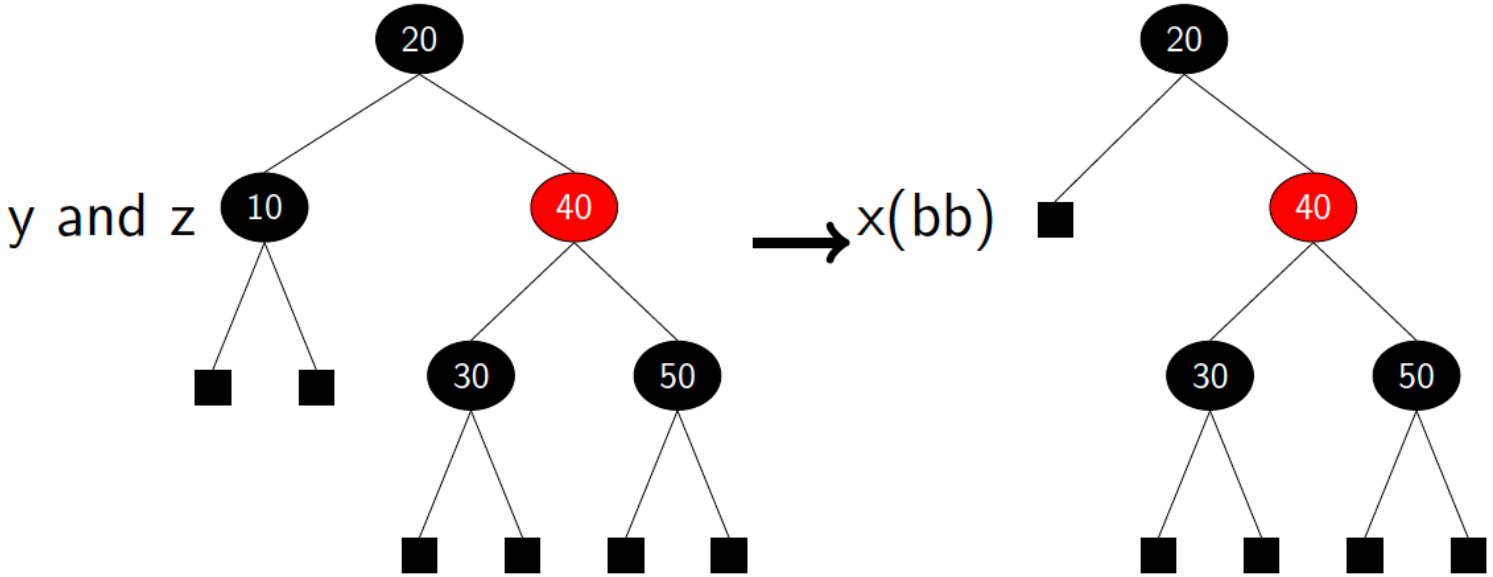
Y = spliced out node (removed from tree)
Z = Key value to remove
W = x's sibling

Case 1 - Example

1. Identify y and z.
2. **Delete z as for a BST**
3. Identify w, w's children, and case
4. Fix any RBT violations

Y = spliced out node (removed from tree)
Z = Key value to remove
W = x's sibling

Delete 10

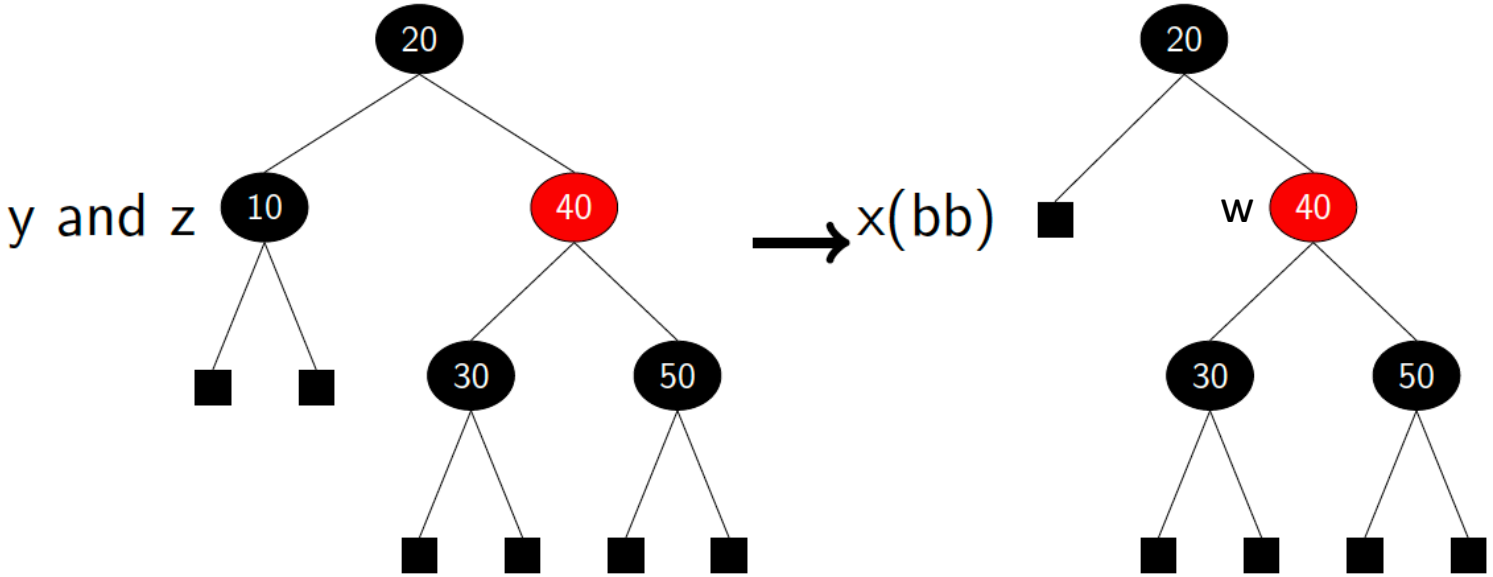


Case 1 - Example

1. Identify y and z.
2. Delete z as for a BST
3. **Identify w, w's children, and case**
4. Fix any RBT violations

Y = spliced out node (removed from tree)
Z = Key value to remove
W = x's sibling

Delete 10



Case 1 – x's sibling w is red

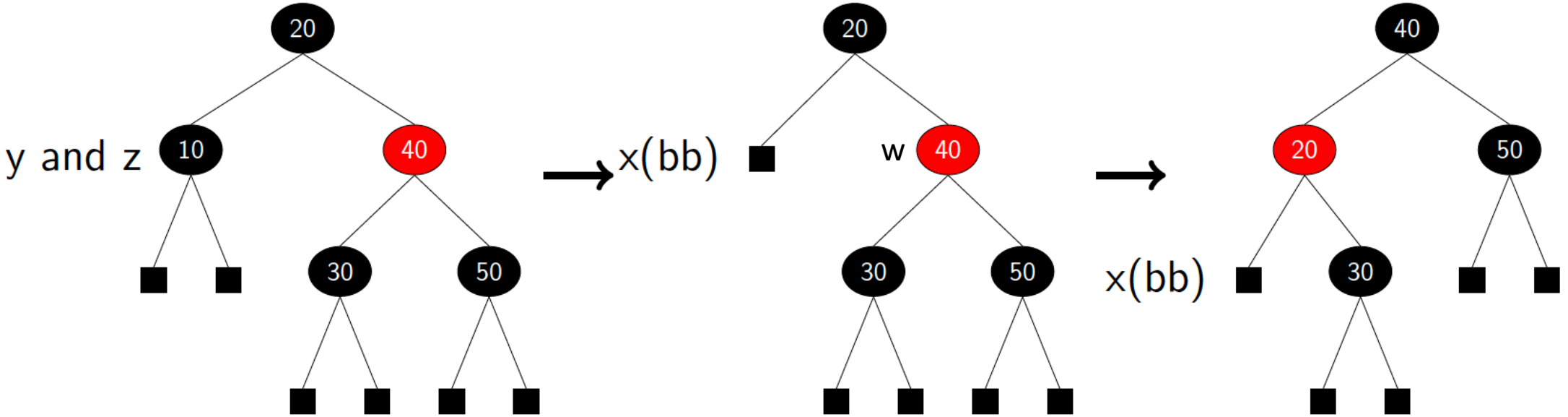
- Switch colours of w and x's parent
- Left-rotate on x's parent

Case 1 - Example

1. Identify y and z.
2. Delete z as for a BST
3. Identify w and w's children
4. **Fix any RBT violations**

Y = spliced out node (removed from tree)
Z = Key value to remove
W = x's sibling

Delete 10



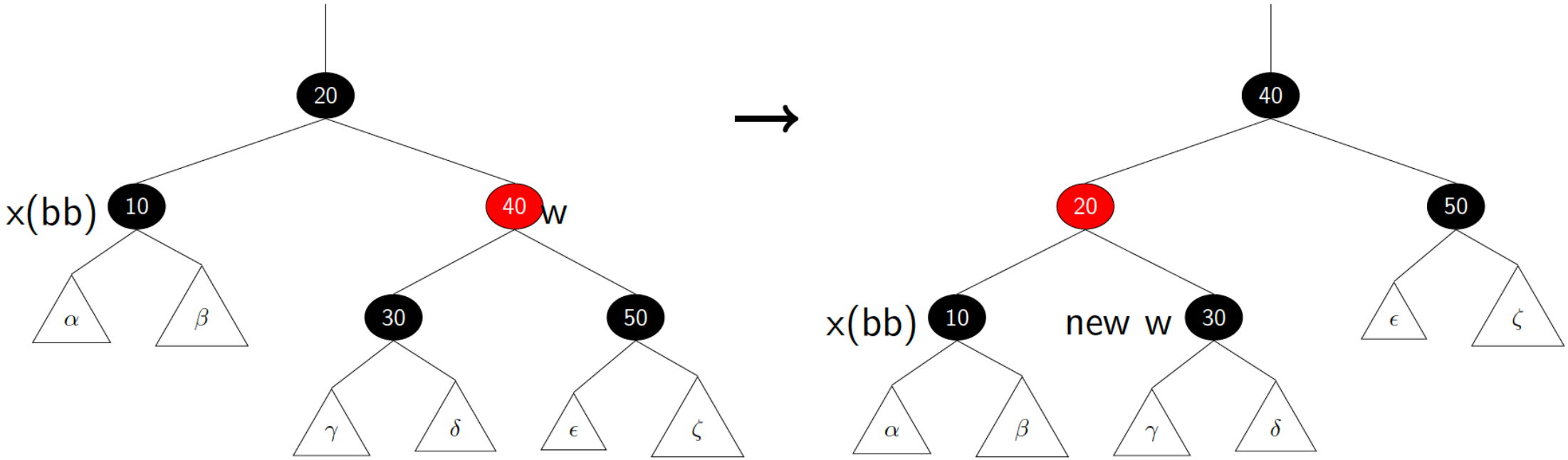
Case 1 – x's sibling w is red

- Switch colours of w and x's parent
- Left-rotate on x's parent

Case 1 ➔ Case 2, 3, or 4.
In this example, we have **Case 2**

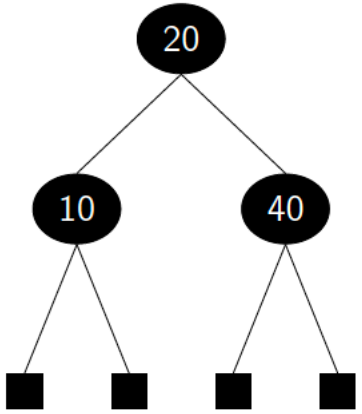
Case 1 – Example

Same tree, without deleting 10, showing fixup actions and only relevant nodes.



Case 2 - Example

Delete 10

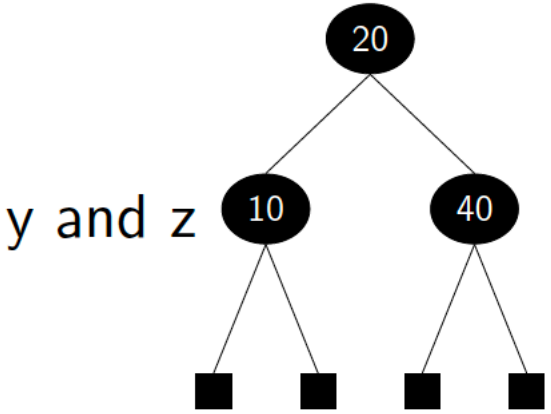


1. Identify y and z.
2. Delete z as for a BST
3. Identify w, w's children, and case
4. Fix any RBT violations

Y = spliced out node (removed from tree)
Z = Key value to remove
W = x's sibling

Case 2 - Example

Delete 10

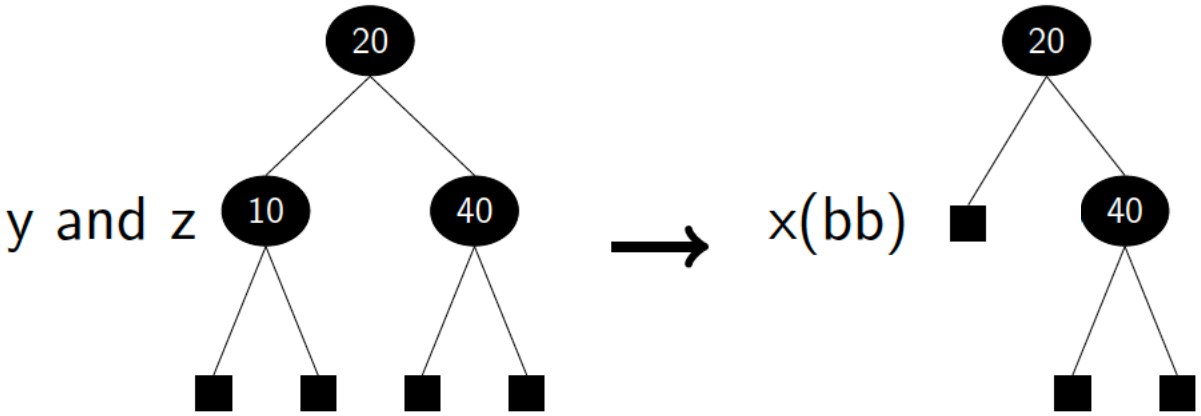


1. **Identify y and z.**
2. Delete z as for a BST
3. Identify w, w's children, and case
4. Fix any RBT violations

Y = spliced out node (removed from tree)
Z = Key value to remove
W = x's sibling

Case 2 - Example

Delete 10



1. Identify y and z.
2. **Delete z as for a BST**
3. Identify w, w's children, and case
4. Fix any RBT violations

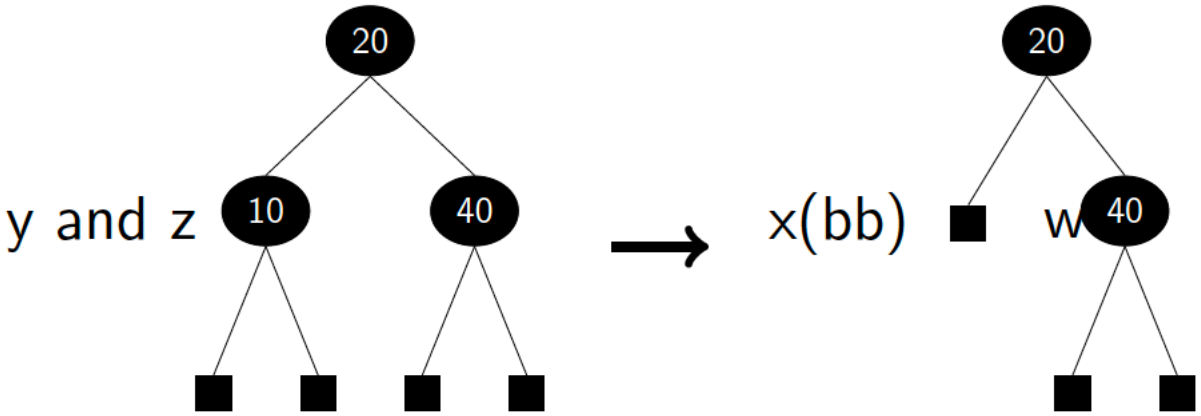
Y = spliced out node (removed from tree)
Z = Key value to remove
W = x's sibling

Case 2 - Example

1. Identify y and z.
2. Delete z as for a BST
3. **Identify w, w's children, and case**
4. Fix any RBT violations

Y = spliced out node (removed from tree)
Z = Key value to remove
W = x's sibling

Delete 10



Case 2 – w is black, both w's children are black

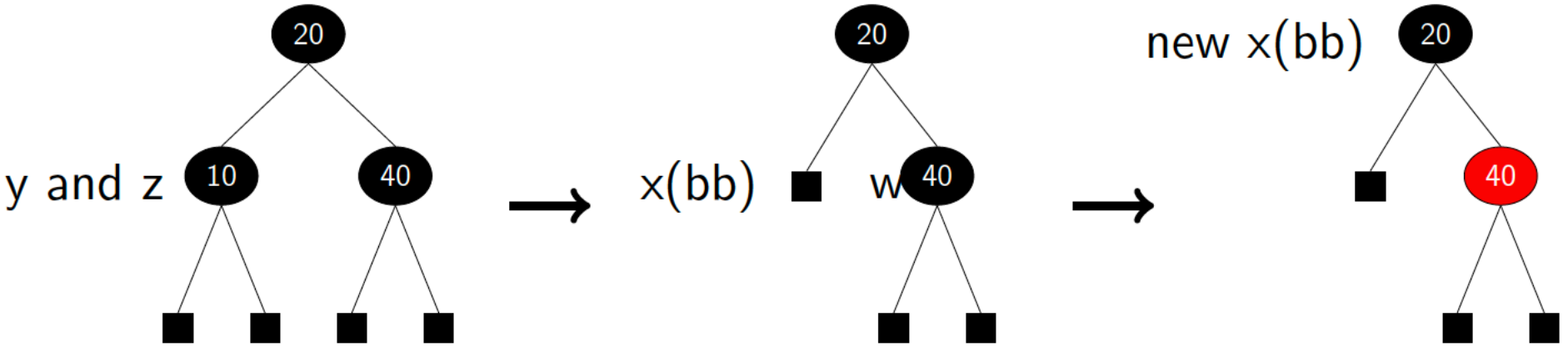
- Make w red
- Make x-parent the new x
- If came from case 1 then terminate

Case 2 - Example

1. Identify y and z.
2. Delete z as for a BST
3. Identify w, w's children, and case
4. **Fix any RBT violations**

Y = spliced out node (removed from tree)
Z = Key value to remove
W = x's sibling

Delete 10



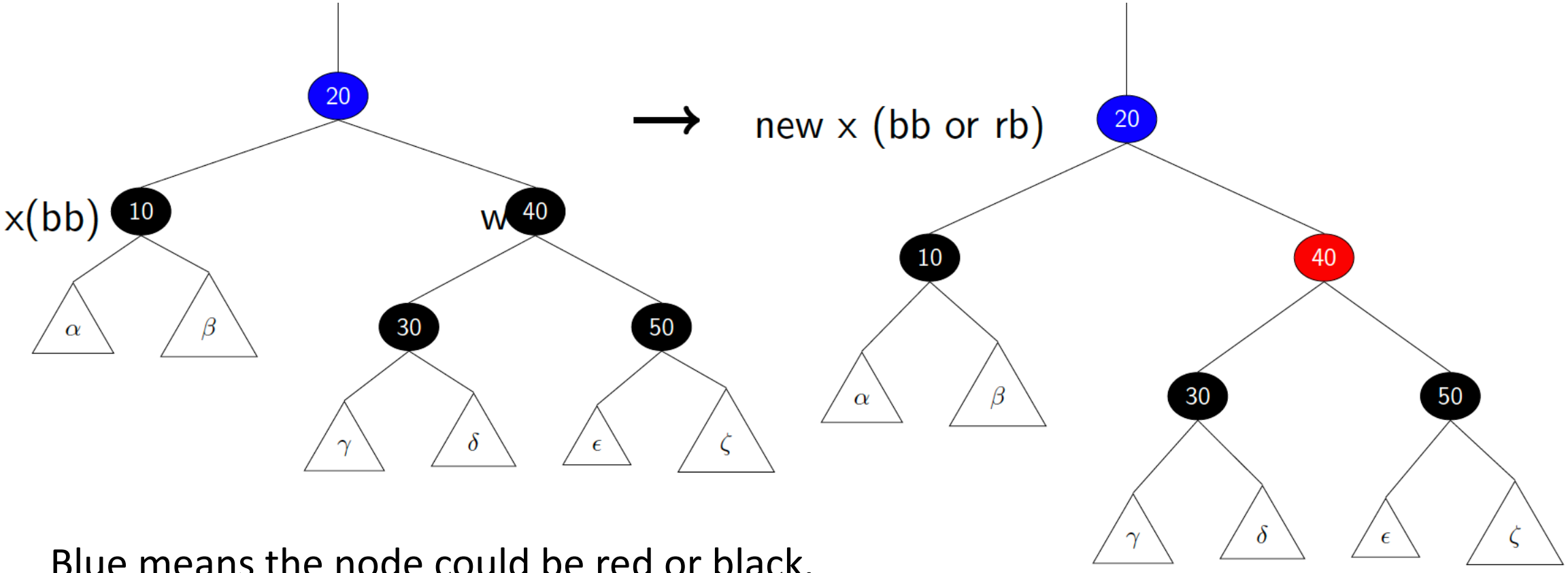
Case 2 – w is black, both w's children are black

- Make w red
- Make x-parent the new x
- If came from case 1 then terminate

Either we are done, as new-x is now red-black (seen as "red" by while loop, and exits), or we head up the tree with new x.

Case 2 - Example

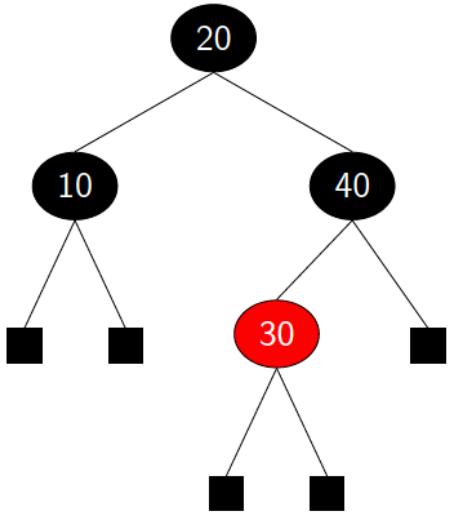
Same tree, without deleting 10, showing fixup actions only.



Blue means the node could be red or black.

Case 3 - Example

Delete 10

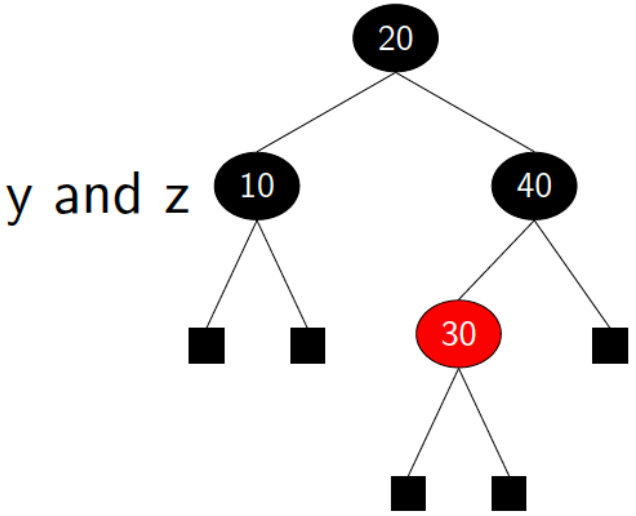


1. Identify y and z.
2. Delete z as for a BST
3. Identify w, w's children, and case
4. Fix any RBT violations

Y = spliced out node (removed from tree)
Z = Key value to remove
W = x's sibling

Case 3 - Example

Delete 10



1. **Identify y and z.**
2. Delete z as for a BST
3. Identify w, w's children, and case
4. Fix any RBT violations

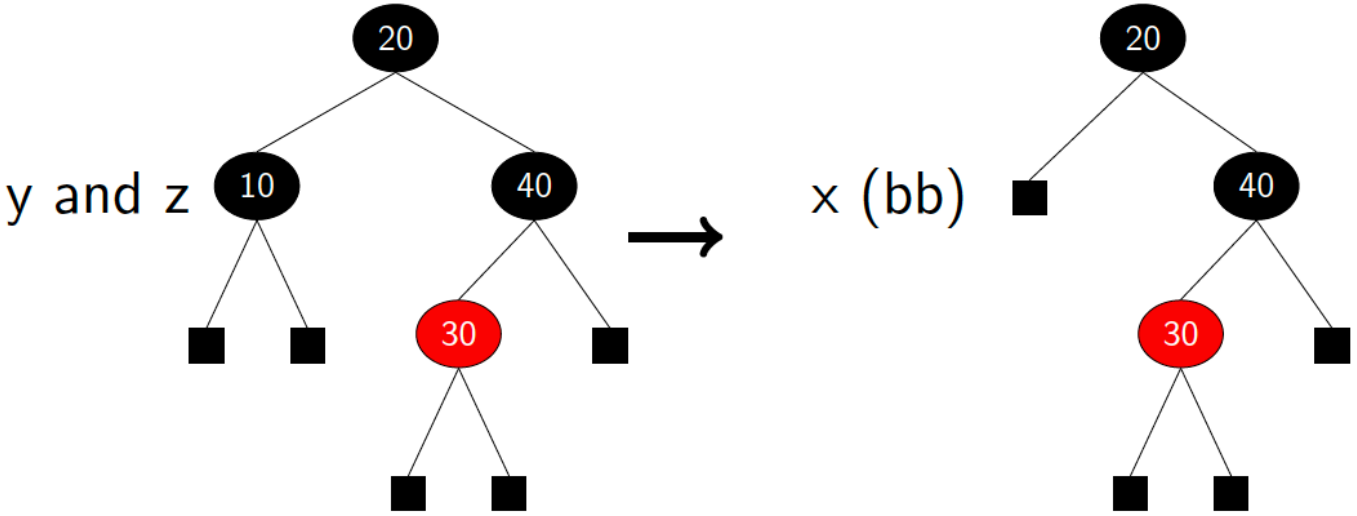
Y = spliced out node (removed from tree)
Z = Key value to remove
W = x's sibling

Case 3 - Example

1. Identify y and z.
2. **Delete z as for a BST**
3. Identify w, w's children, and case
4. Fix any RBT violations

Y = spliced out node (removed from tree)
Z = Key value to remove
W = x's sibling

Delete 10

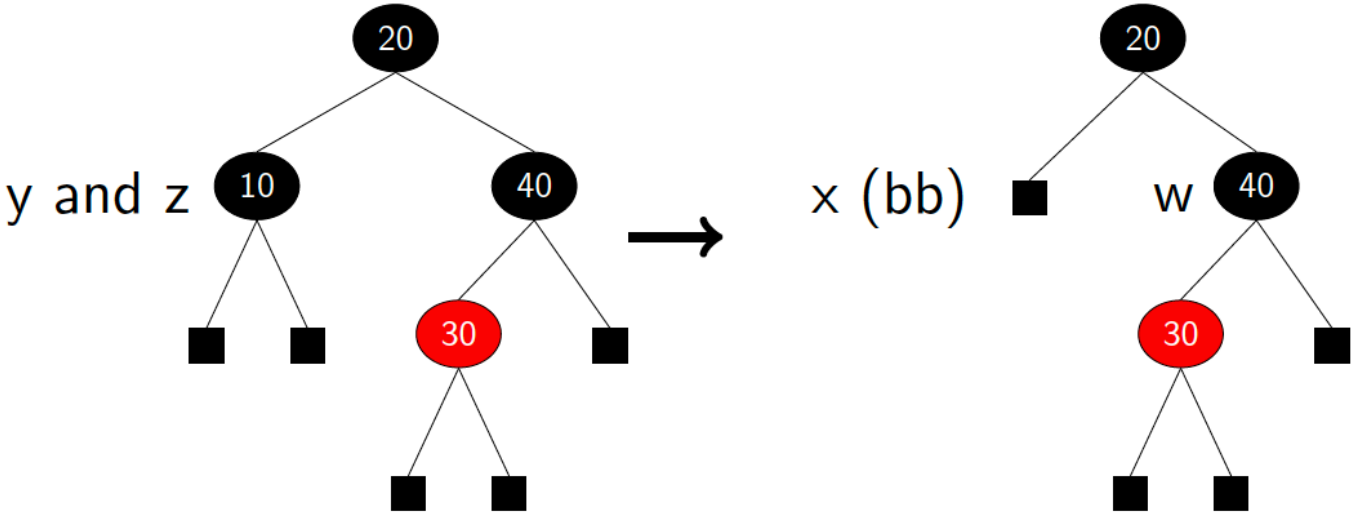


Case 3 - Example

1. Identify y and z.
2. Delete z as for a BST
3. **Identify w, w's children, and case**
4. Fix any RBT violations

Y = spliced out node (removed from tree)
Z = Key value to remove
W = x's sibling

Delete 10



Case 3 – w is black, left child (inner) is red

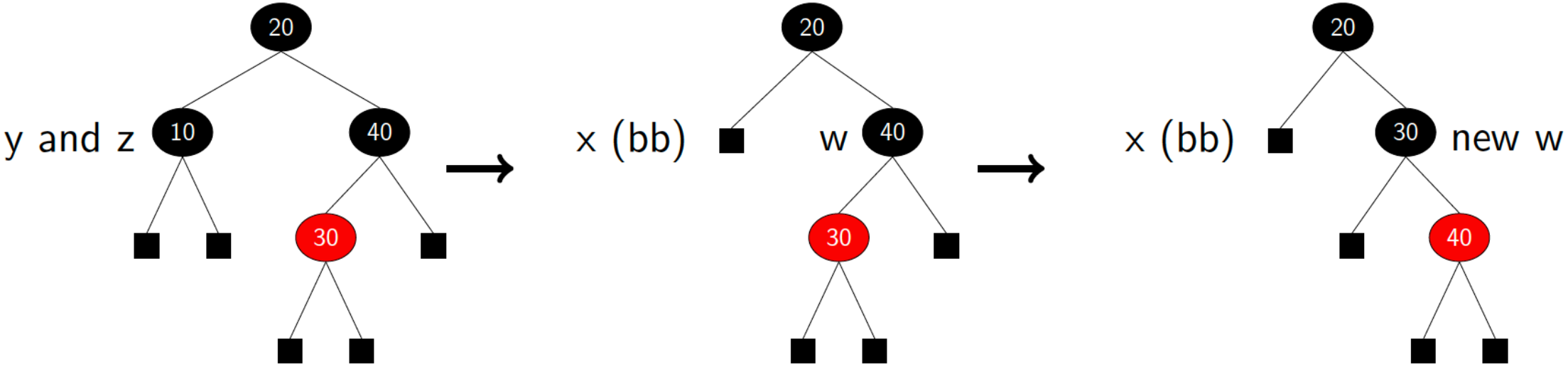
- Make w's left-child black
- Make w red
- Right-rotate on w
- Update w (x-sibling)

Case 3 - Example

1. Identify y and z.
2. Delete z as for a BST
3. Identify w, w's children, and case
4. **Fix any RBT violations**

Y = spliced out node (removed from tree)
Z = Key value to remove
W = x's sibling

Delete 10



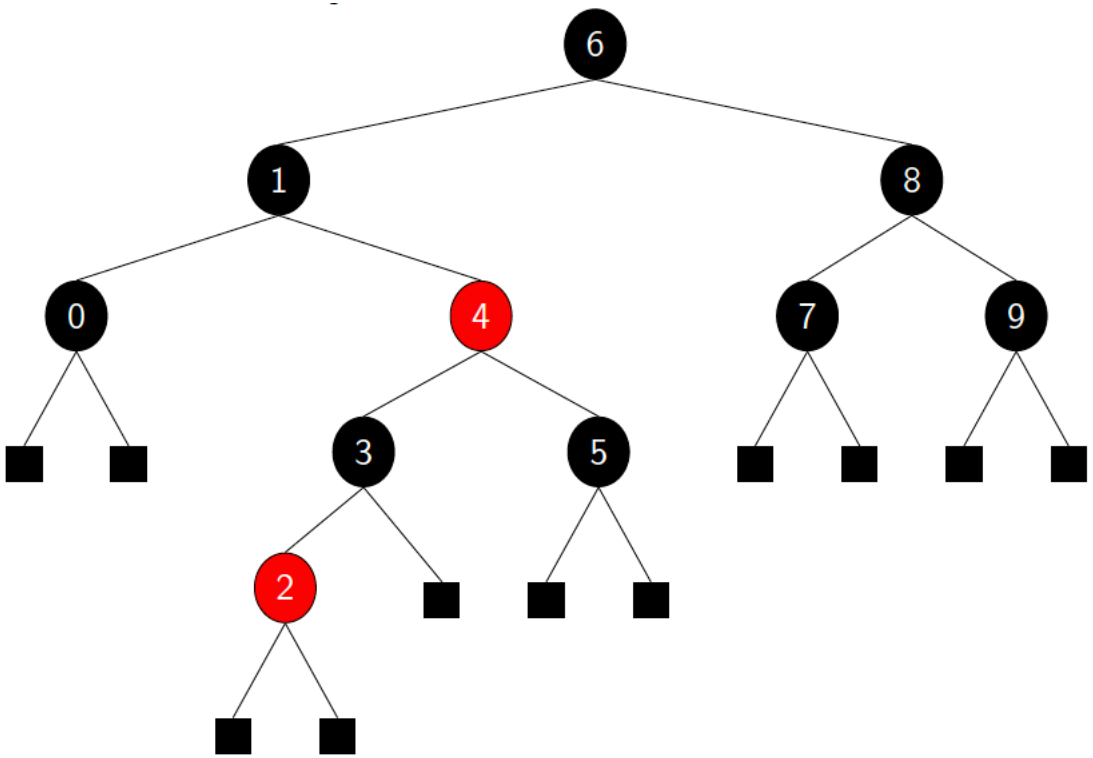
Case 3 – w is black, left child (inner) is red

- Make w's left-child black
- Make w red
- Right-rotate on w
- Update w (x-sibling)

We now fall into ➔ **Case 4**

Case 4 - Example

Delete 5

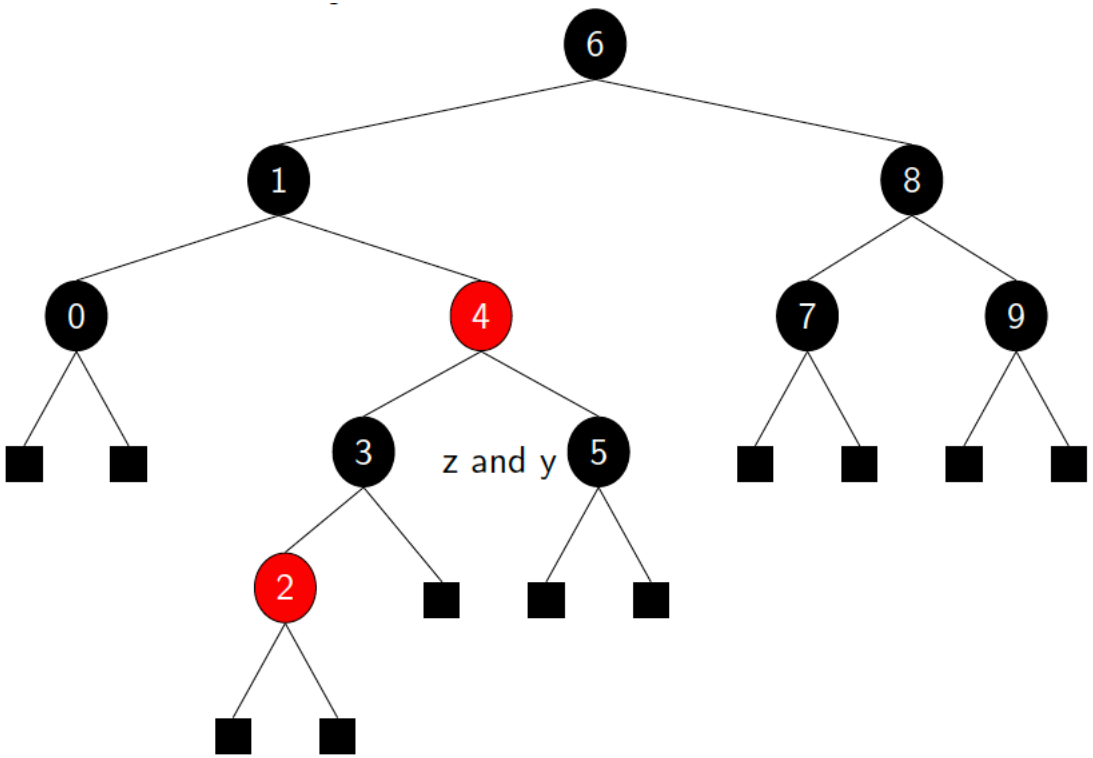


1. Identify y and z.
2. Delete z as for a BST
3. Identify w, w's children, and case
4. Fix any RBT violations

Y = spliced out node (removed from tree)
Z = Key value to remove
W = x's sibling

Case 4 - Example

Delete 5

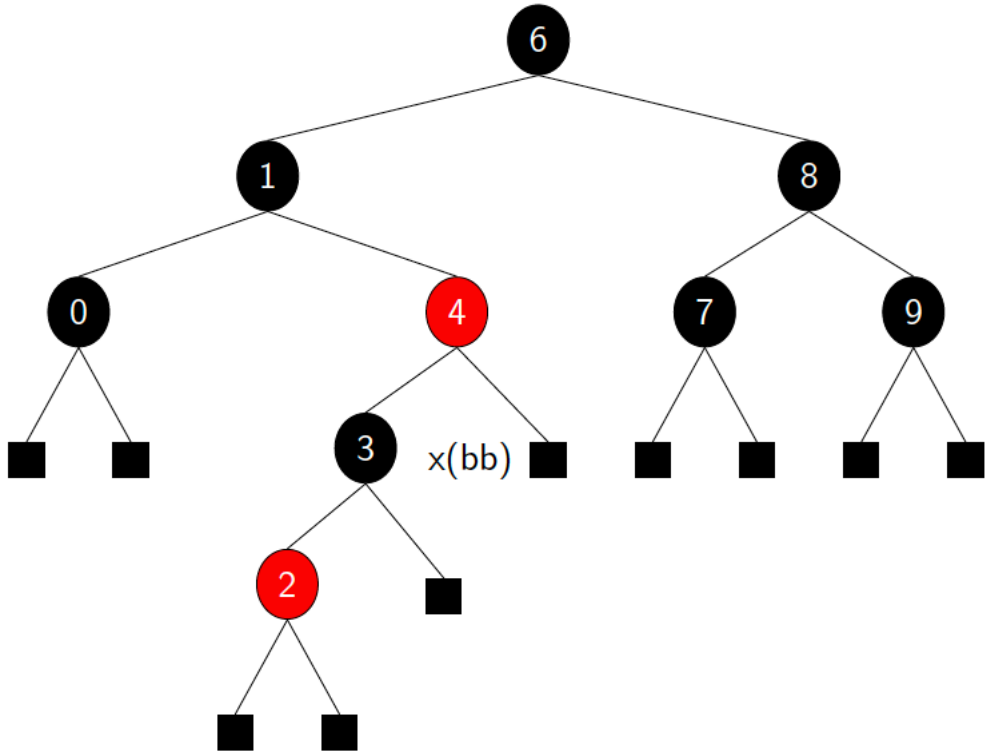


1. **Identify y and z.**
2. Delete z as for a BST
3. Identify w, w's children, and case
4. Fix any RBT violations

Y = spliced out node (removed from tree)
Z = Key value to remove
W = x's sibling

Case 4 - Example

Delete 5



1. Identify y and z.
2. **Delete z as for a BST**
3. Identify w, w's children, and case
4. Fix any RBT violations

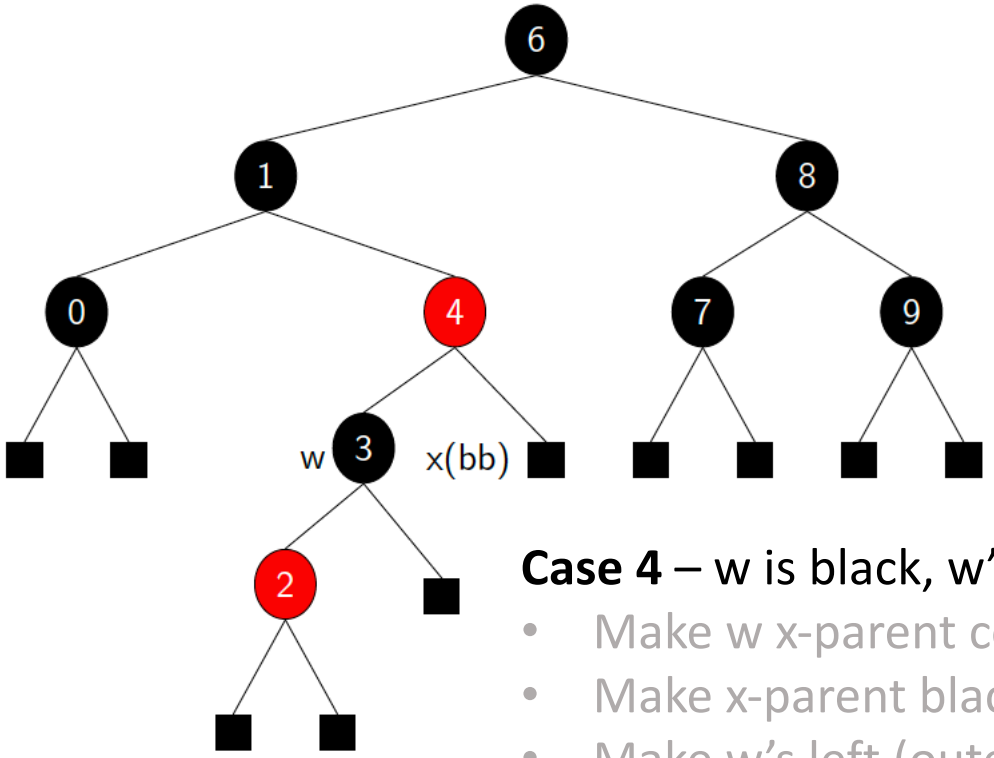
Y = spliced out node (removed from tree)
Z = Key value to remove
W = x's sibling

Case 4 - Example

1. Identify y and z.
2. Delete z as for a BST
3. **Identify w, w's children, and case**
4. Fix any RBT violations

Y = spliced out node (removed from tree)
Z = Key value to remove
W = x's sibling

Delete 5



Note: here we have x as a right-child. Therefore, we flip all left and rights. These are shown underlined.

Case 4 – w is black, w's left (outer) child is black

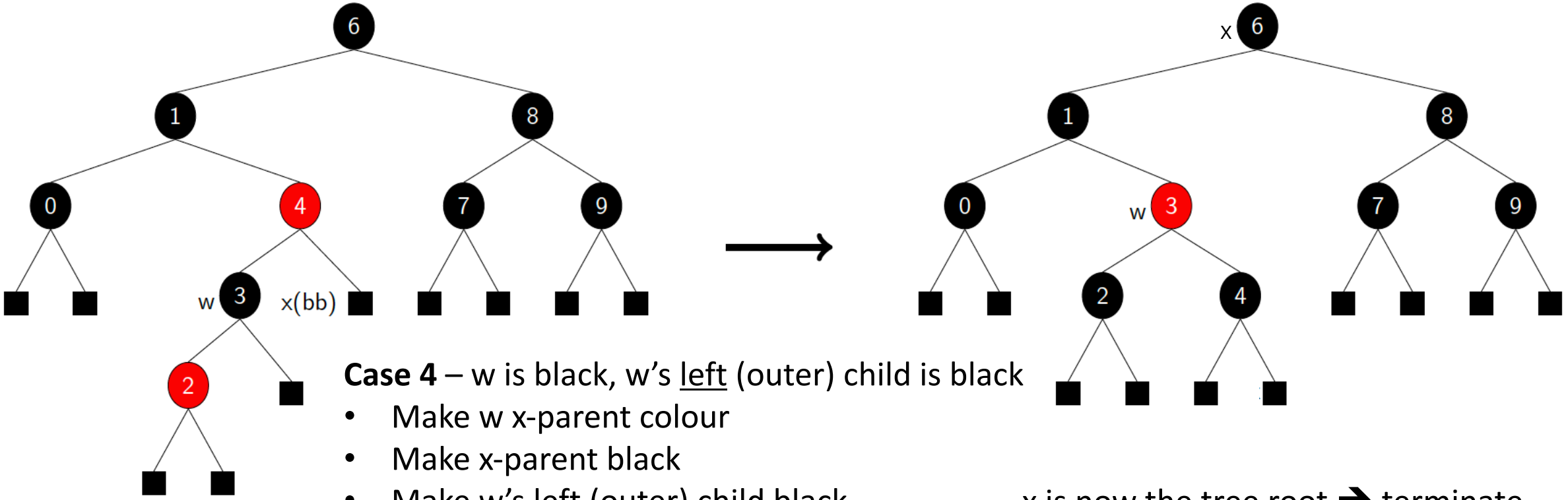
- Make w x-parent colour
- Make x-parent black
- Make w's left (outer) child black
- Right-rotate x-parent
- Make x the root

Case 4 - Example

1. Identify y and z.
2. Delete z as for a BST
3. Identify w, w's children, and case
4. **Fix any RBT violations**

Y = spliced out node (removed from tree)
Z = Key value to remove
W = x's sibling

Delete 5



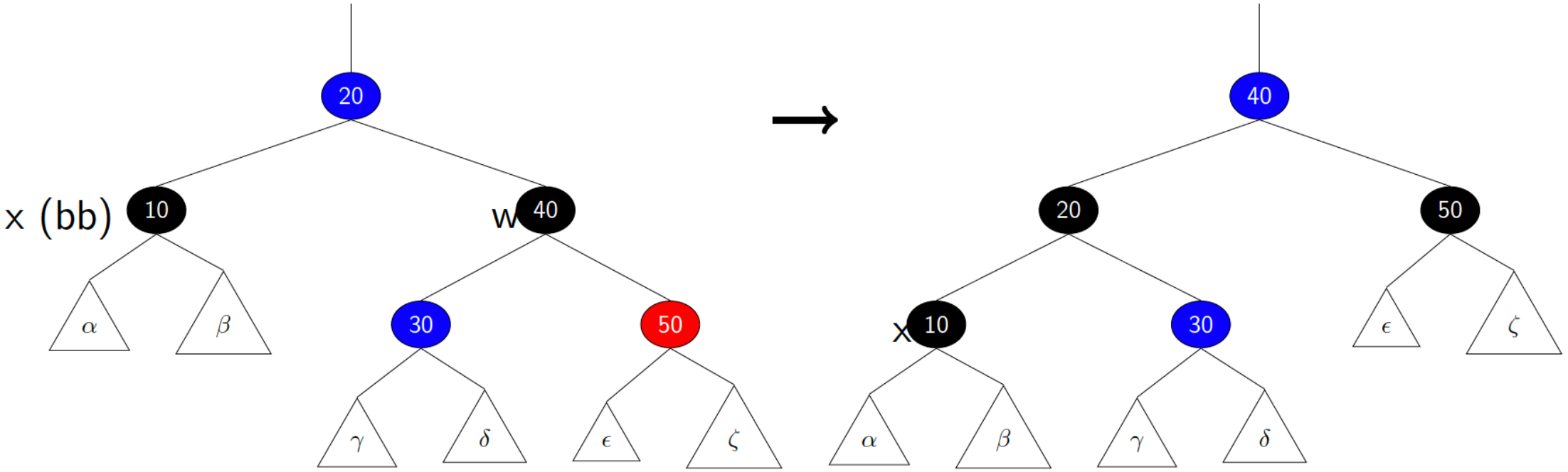
Case 4 – w is black, w's left (outer) child is black

- Make w x-parent colour
- Make x-parent black
- Make w's left (outer) child black
- Right-rotate x-parent
- Make x the root

x is now the tree root ➔ terminate

Case 4 - Example

Modified tree, with x now a left child, without deleting 10, showing fixup actions only. This time x is a left-child.



Blue means the node could be red or black.

Suggested reading

Today's lecture covered section 13.3.

The code for deletion and fixup won't be implemented in the labs due to its high complexity. Pseudocode was included in lectures to make it easier to see how re-balancing was achieved.