Red-Black Trees

- What is a red-black tree?
  - node color: red or black
  - *nil*[*T*] and black height
- Subtree rotation
- Node insertion
- Node deletion

Red-black trees: Overview

- Red-black trees are a variation of binary search trees to ensure that the tree is ***balanced***.
  - » Height is $O(\lg n)$, where n is the number of nodes.
- Operations take $O(\lg n)$ time in the worst case.
- A red-black tree is normally not perfectly balanced, but satisfying:

  The length of the longest path from a node to a leaf is less than two times of the length of the shortest path from that node to a leaf.

Red-black Tree

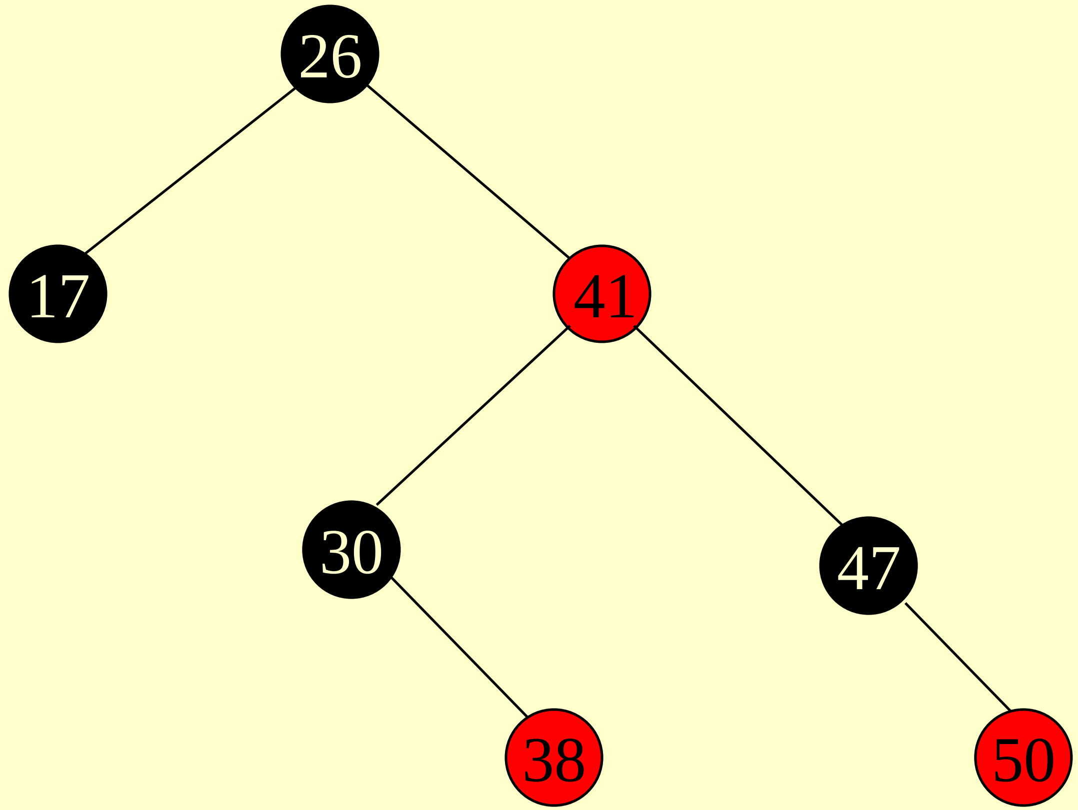
- Every node is a red-black tree is associated with a bit: the attribute *color*, which is either **red** or **black**.
- All other attributes of BSTs are inherited:
  - » *key*, *left*, *right*, and *p*.
- If a child or the parent of a node does not exist, the corresponding pointer field of the node contains the value *nil*.
- Sentinel - *nil*[*T*], representing all the *virtual nil* nodes.
  - A node, if it has only one child, a virtual nil child will be created. If it has no children (i.e., it is a leaf node), two virtual nil children will be created.
  - For the tree root, a virtual nil parent will be created.

Red-black Properties

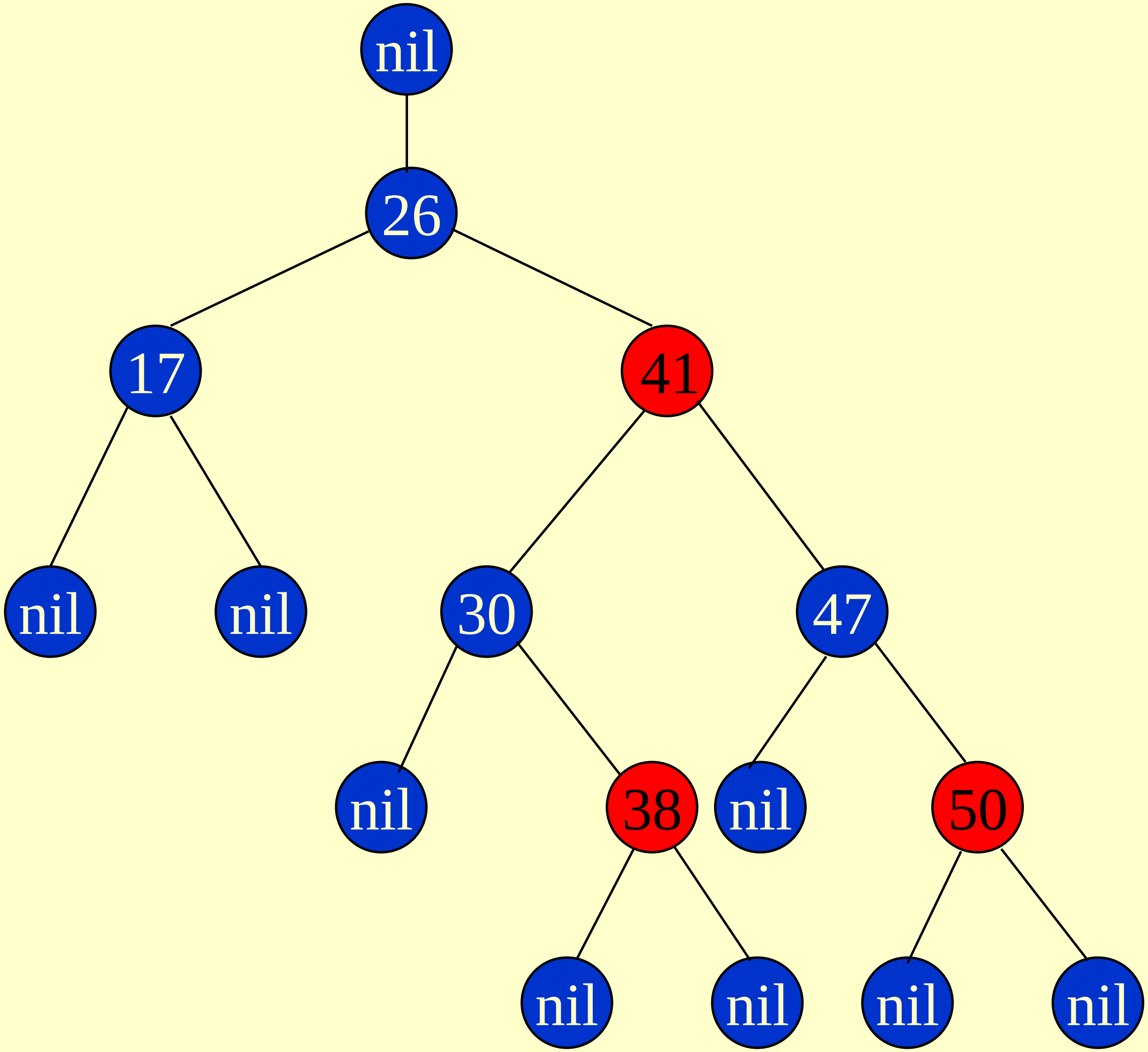
1. Every node is either red or black.
2. The root is black.
3. Every *virtual* node (*nil*) is black.
4. If a node is red, then both its children are black.

5. For each node, all paths from the node to descendant leaves contain the same number of black nodes.

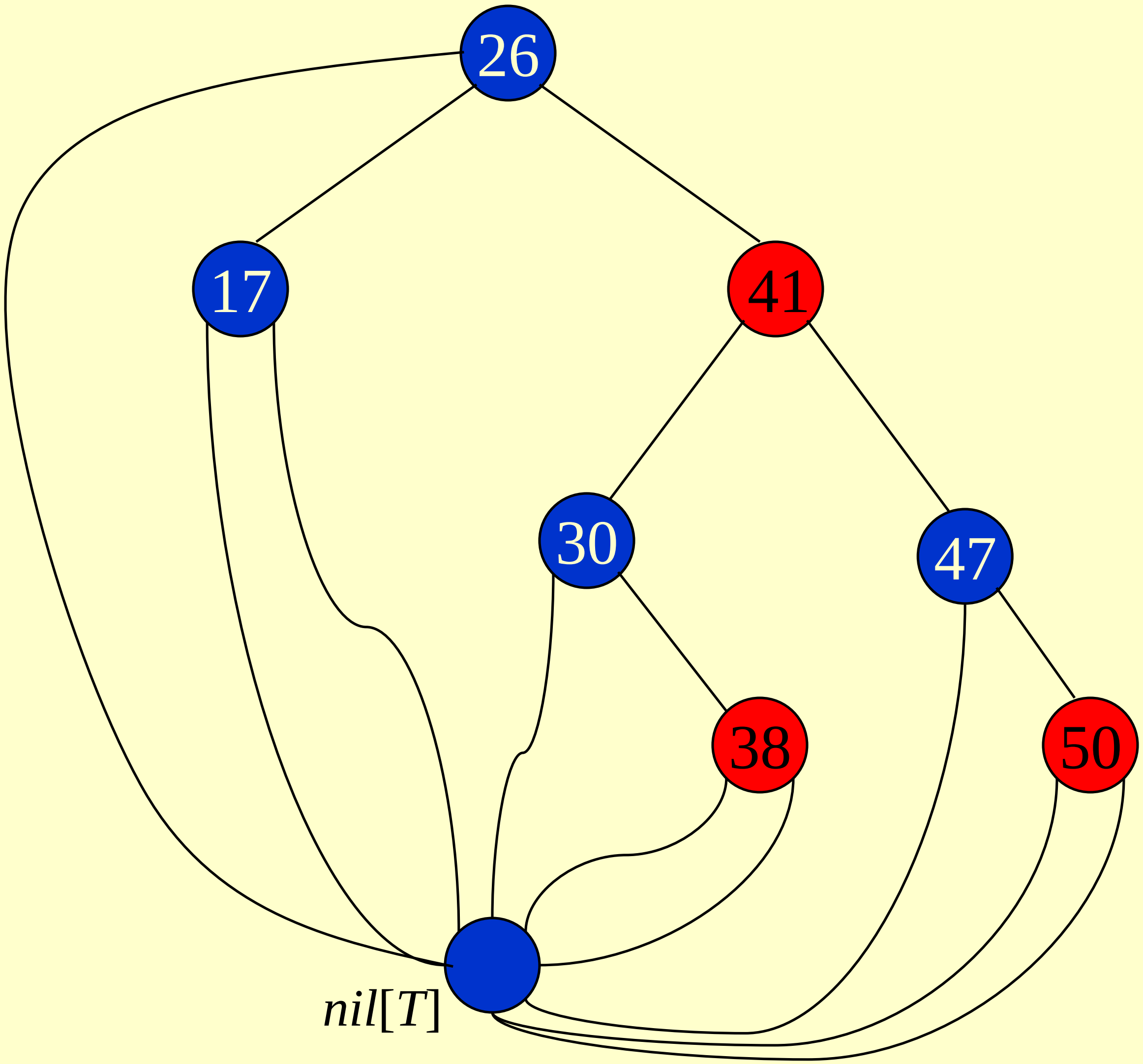
Red-black Tree – Example



Red-black Tree – Example



Red-black Tree – Example



Red-black Properties

1. Every node is either red or black.
2. The root is black.
3. Every *virtual* leaf (*nil*) is black.
4. If a node is red, then both its children are black.

5. For each node, all paths from the node to descendant leaves contain the same number of black nodes.

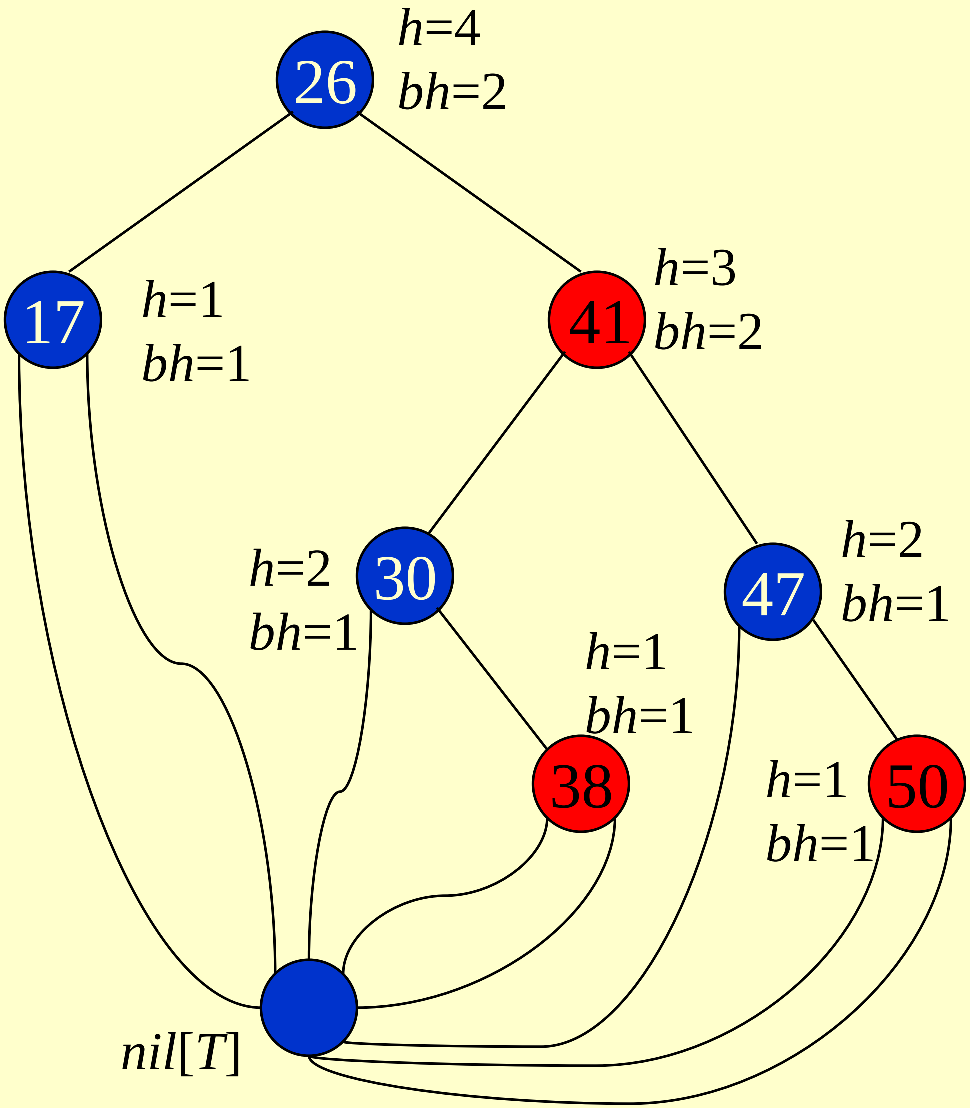
Height of a Red-black Tree

- Height of a node:
  - » $h(x)$ = number of edges in a longest path to a leaf.
- Black-height of a node x, $bh(x)$:
  - » $bh(x)$ = number of black nodes (including $nil[T]$) on the path from x to leaf, not counting x.
- Black-height of a red-black tree is the black-height of its root.
  - » By Property 5, black height is well defined.

Height of a Red-black Tree

- Example:
- Height of a node:

  $h(x)$ = # of edges in a longest path to a leaf.
- Black-height of a node $bh(x)$ = # of black nodes on path from x to leaf, not counting x.
- How are they related?
  - » $bh(x) \leq h(x) \leq 2bh(x)$



Lemma "RB Height"

Consider a node x in an RB tree: The longest descending path from x to a leaf has length $h(x)$, which is at most twice the length of the shortest descending path from x to a leaf.

Proof:

black nodes on any path from $x = bh(x)$ (prop 5)

$\leq$ # nodes on shortest path from x, $s(x)$. (prop 1)

But, there are no consecutive red (prop 4),

and we end with black (prop 3), so $h(x) \leq 2\, bh(x)$.

Thus, $h(x) \leq 2s(x)$. **QED**

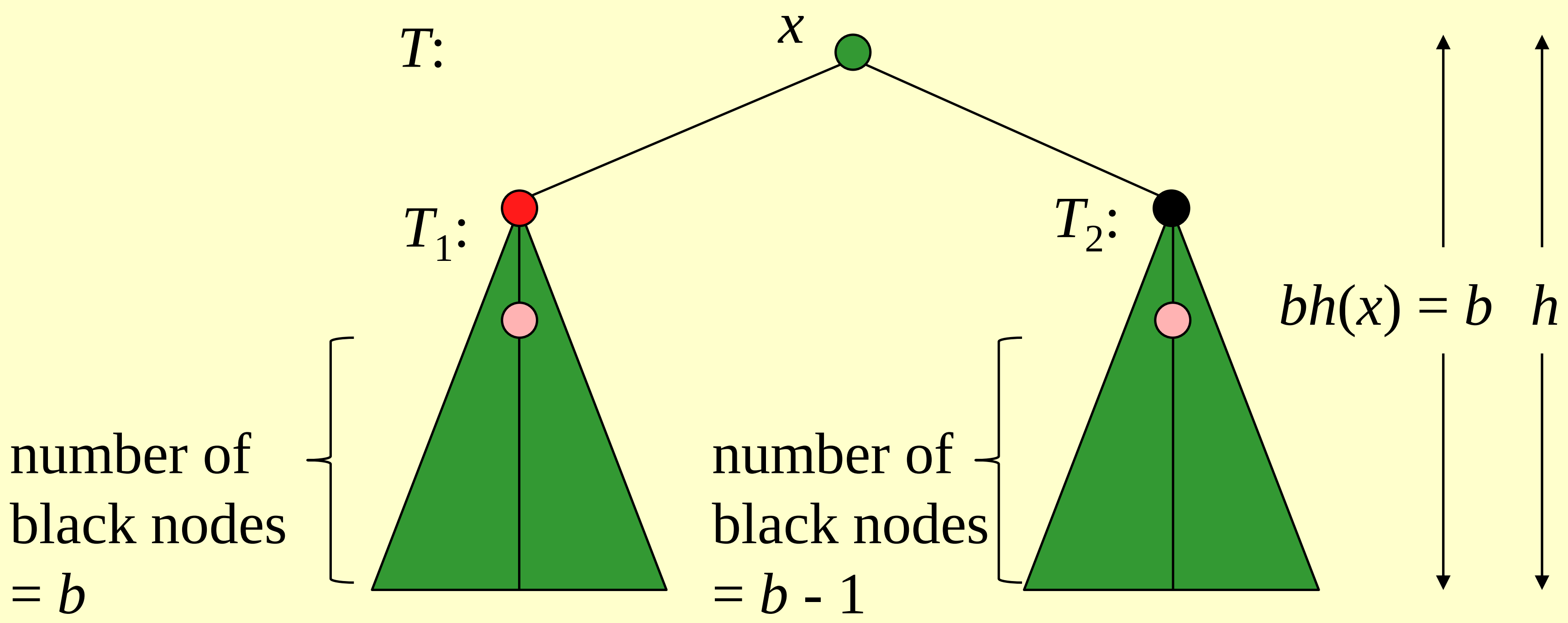
Bound on RB Tree Height

- **Lemma:** The subtree rooted at any node x has $\geq 2^{bh(x)}-1$ internal nodes.
- **Proof:** By induction on height of x, $h(x)$.
  - » **Base Case:** Height $h(x) = 0 \Rightarrow x$ is a leaf $\Rightarrow bh(x) = 0$. Subtree has $2^0-1 = 0$ nodes.
  - » **Induction Step: Assume that for any node with height < *h* the lemma holds.**

    Consider node x with $h(x) = h > 0$ and $bh(x) = b$.

    - Each child of x has height at most $h - 1$ and black-height either b (child is red) or $b - 1$ (child is black).
    - By ind. hyp., each child has $\geq 2^{bh(x)-1} - 1$ internal nodes.
    - Subtree rooted at x has $\geq 2(2^{bh(x)-1} - 1) + 1$ $= 2^{bh(x)} - 1$ internal nodes. (The +1 is for x itself.)

Bound on RB Tree Height



number of internal nodes of T_1 $\geq 2^b - 1$

number of internal nodes of T_2 $\geq 2^{b-1} - 1$

number of internal nodes of $T \geq |T_1| + |T_2| + 1$

$\geq (2^b - 1) + (2^{b-1} - 1) + 1 \geq 2^b - 1$

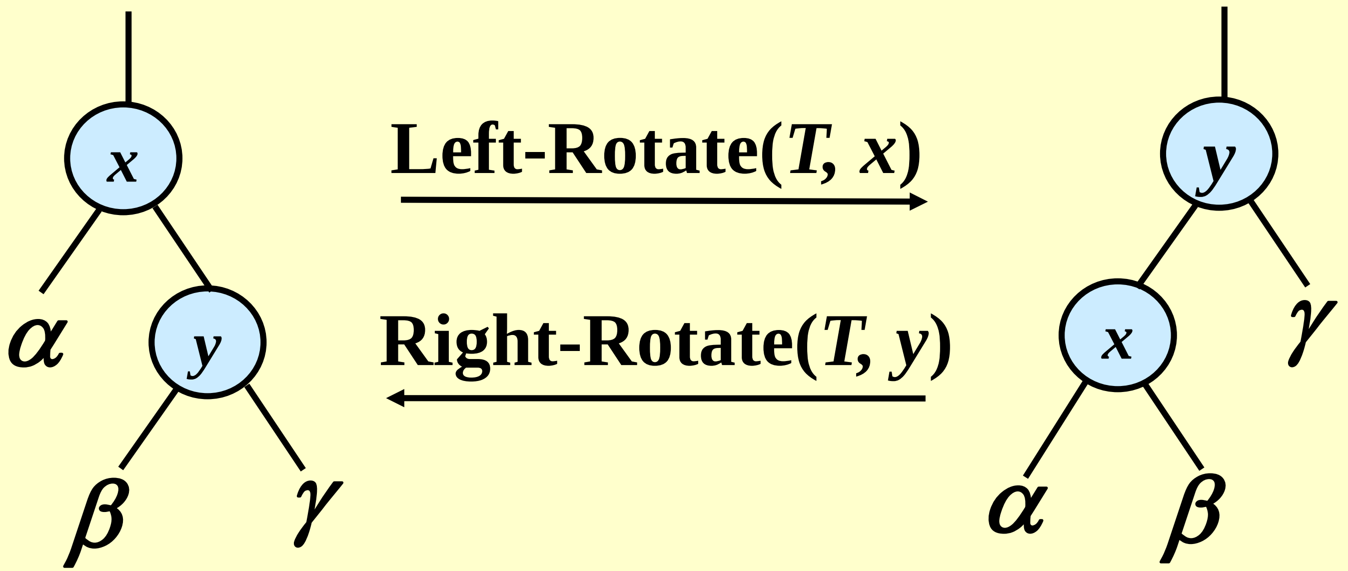
Bound on RB Tree Height

- Lemma: The subtree rooted at any node x has $\geq 2^{bh(x)}-1$ internal nodes.
- **Lemma 13.1:** A red-black tree with n internal nodes has height at most $2\lg(n+1)$.
- **Proof:**
  - » By the above lemma, $n \geq 2^{bh} - 1$,
  - » and since $bh \geq h/2$, we have $n \geq 2^{h/2} - 1$.
  - » $\Rightarrow h \leq 2\lg(n + 1)$.

Operations on RB Trees

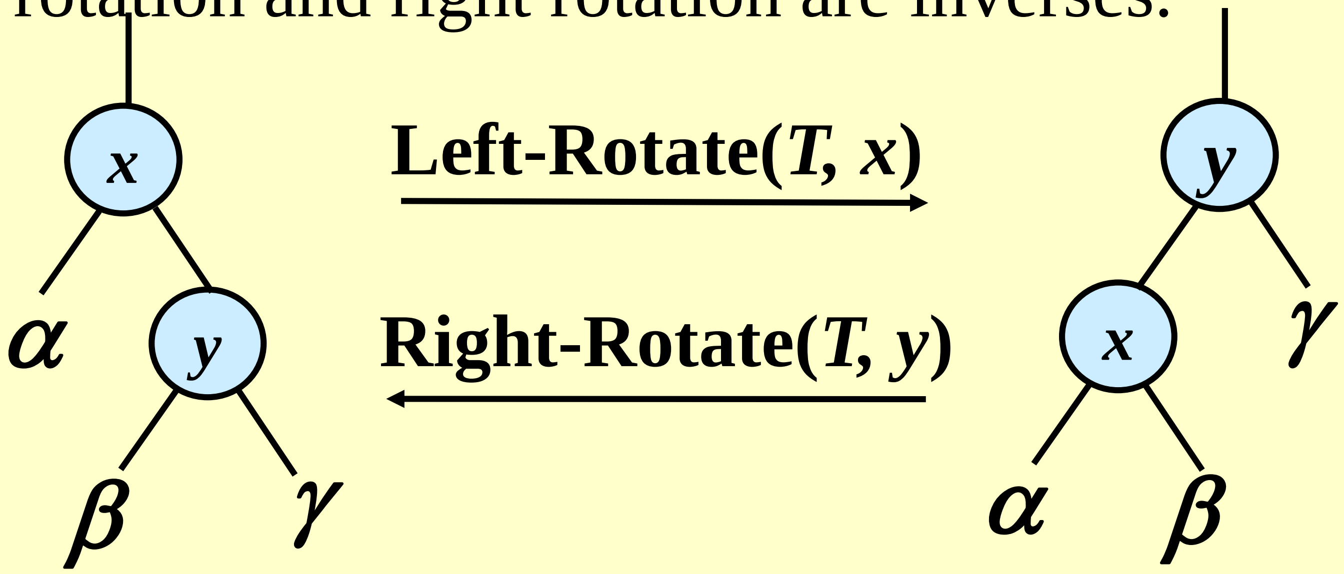
- All operations can be performed in $O(\lg n)$ time.
- The query operations, which don't modify the tree, are performed in exactly the same way as they are in binary search trees.
- Insertion and Deletion are not straightforward. Why?

Rotations



Rotations

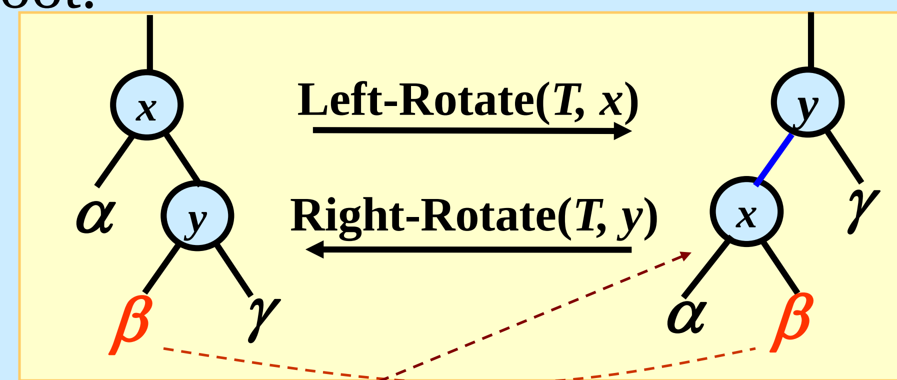
- Rotations are the basic tree-restructuring operation for almost all *balanced* search trees.
- Rotation takes a red-black-tree and a node as the input,
- Change pointers to change the local structure, and
- Won't violate the binary-search-tree property.
- Left rotation and right rotation are inverses.



Left Rotation – Pseudo-code

Left-Rotate (*T, x*)

```
y ← right[x]   // Set y.
right[x] ← left[y]   //Turn y's left subtree β into x's right subtree.
if left[y] ≠ nil[T]
    then p[left[y]] ← x //Set x to be the parent of left[y] = β.
p[y] ← p[x]          //Link x's parent to y.
if p[x] = nil[T]     //If x is the root.
    then root[T] ← y
    else if x = left[p[x]]
        then left[p[x]] ← y
        else right[p[x]] ← y
left[y] ← x          // Put x as y's left child.
p[x] ← y
```



Rotation

- The pseudo-code for Left-Rotate assumes that
  - » *right*[*x*] $\neq$ *nil*[*T*], and
  - » root's parent is *nil*[*T*].
- Left Rotation on *x*, makes *x* the left child of *y*, and the left subtree of *y* into the right subtree of *x*.
- Pseudocode for Right-Rotate is symmetric: exchange *left* and *right* accordingly.
- ***Time:*** $O(1)$ for both Left-Rotate and Right-Rotate, since a constant number of pointers are modified.

Reminder: Red-black Properties

1. Every node is either red or black.
2. The root is black.
3. Every leaf (*nil*) is black.
4. If a node is red, then both its children are black.

5. For each node, all paths from the node to descendant leaves contain the same number of black nodes.

Insertion in RB Trees

- Insertion must preserve all red-black properties.
- Should an inserted node be colored Red? Black?
- Basic steps:
  - » Use Tree-Insert from BST (slightly modified) to insert a node *z* into *T*.
    - Procedure **RB-Insert(z)**.
  - » Color the node *z* red.
  - » Fix the modified tree by re-coloring nodes and performing rotation to preserve RB tree property.
    - Procedure **RB-Insert-Fixup**.

Insertion

RB-Insert(*T*, *z*)

```
y ← nil[T]
x ← root[T]
while x ≠ nil[T]
    do y ← x
        if key[z] < key[x]
            then x ← left[x]
            else x ← right[x]
p[z] ← y
if y = nil[T]
    then root[T] ← z
    else if key[z] < key[y]
        then left[y] ← z
        else right[y] ← z
```

RB-Insert(*T*, *z*) Contd.

```
left[z] ← nil[T]
right[z] ← nil[T]
color[z] ← RED
RB-Insert-Fixup(T, z)
```

How does it differ from the Tree-Insert procedure of BSTs?

Which of the RB properties might be violated?

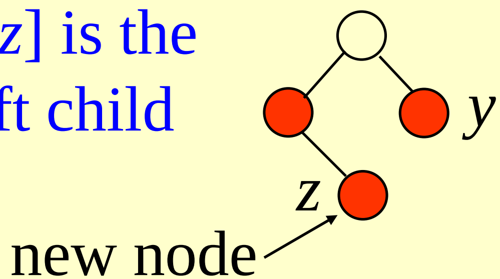
Fix the violations by calling RB-Insert-Fixup.

Insertion – Fixup

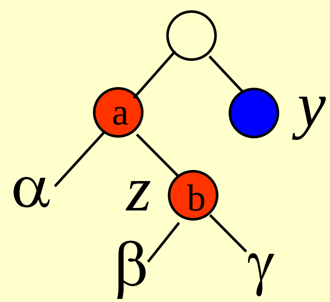
- Problem: we may have a pair of consecutive reds where we did the insertion.
- Solution: rotate it up the tree and away… Six cases have to be handled:

p[*z*] is the left child

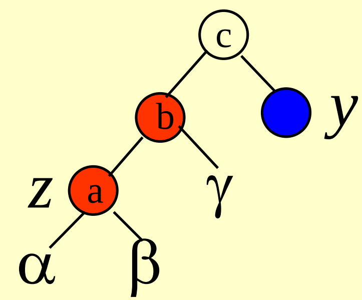
Case 1:



Case 2:

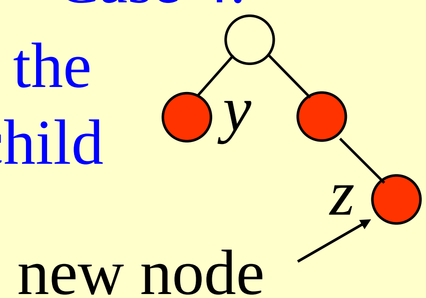


Case 3:

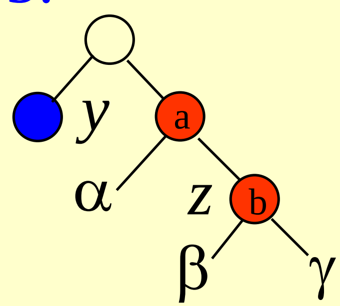


p[*z*] is the right child

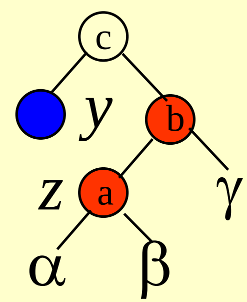
Case 4:



Case 5:



Case 6:

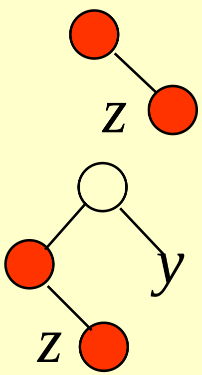


Insertion – Fixup

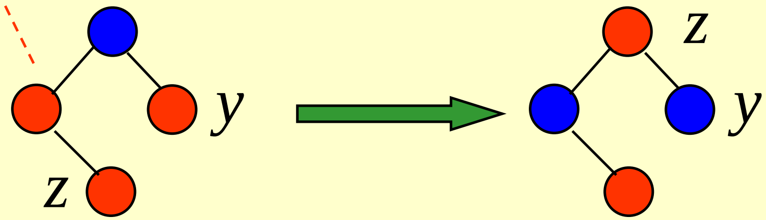
RB-Insert-Fixup (*T*, *z*)

z's parent is the left child of its own parent

```
1.  while color[p[z]] = RED
2.      do if p[z] = left[p[p[z]]]   //for cases 1 – 3
             then y ← right[p[p[z]]]
3.                if color[y] = RED
4.                   then color[p[z]] ← BLACK        // Case 1
5.                        color[y] ← BLACK           // Case 1
6.                        color[p[p[z]]] ← RED       // Case 1
7.                        z ← p[p[z]]                // Case 1
```



Case 1:



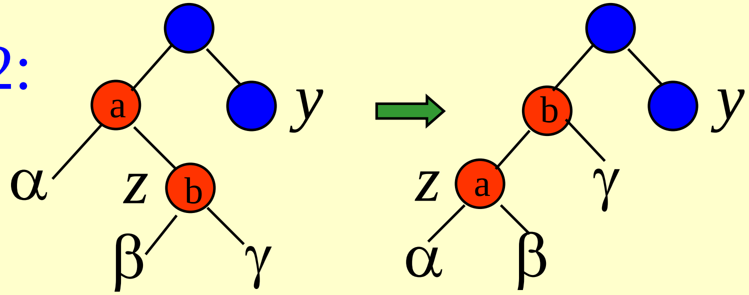
Change this node to red to keep the number of black nodes not increased

Insertion – Fixup

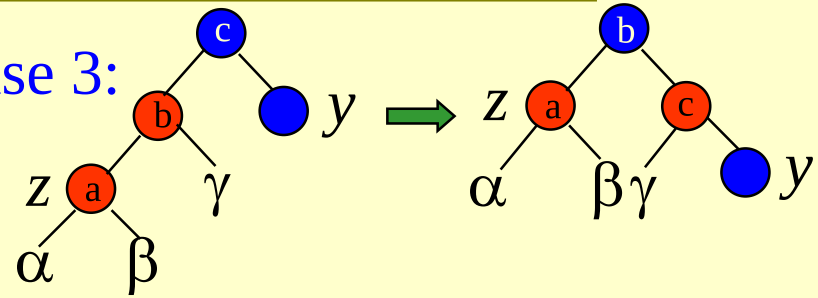
RB-Insert-Fixup(*T*, *z*) (Contd.)

```
9.          else if z = right[p[z]]  // color[y] ≠ RED
10.              then z ← p[z]          // Case 2
11.                    LEFT-ROTATE(T, z)  // Case 2
12.              color[p[z]] ← BLACK       // Case 3
13.              color[p[p[z]]] ← RED       // Case 3
14.              RIGHT-ROTATE(T, p[p[z]]) // Case 3
15.      else (if p[z] = right[p[p[z]]]) (for cases 4 – 6, same
16.           as 3-14 with "right" and "left" exchanged)
17. color[root[T ]] ← BLACK
```

Case 2:



Case 3:



Correctness

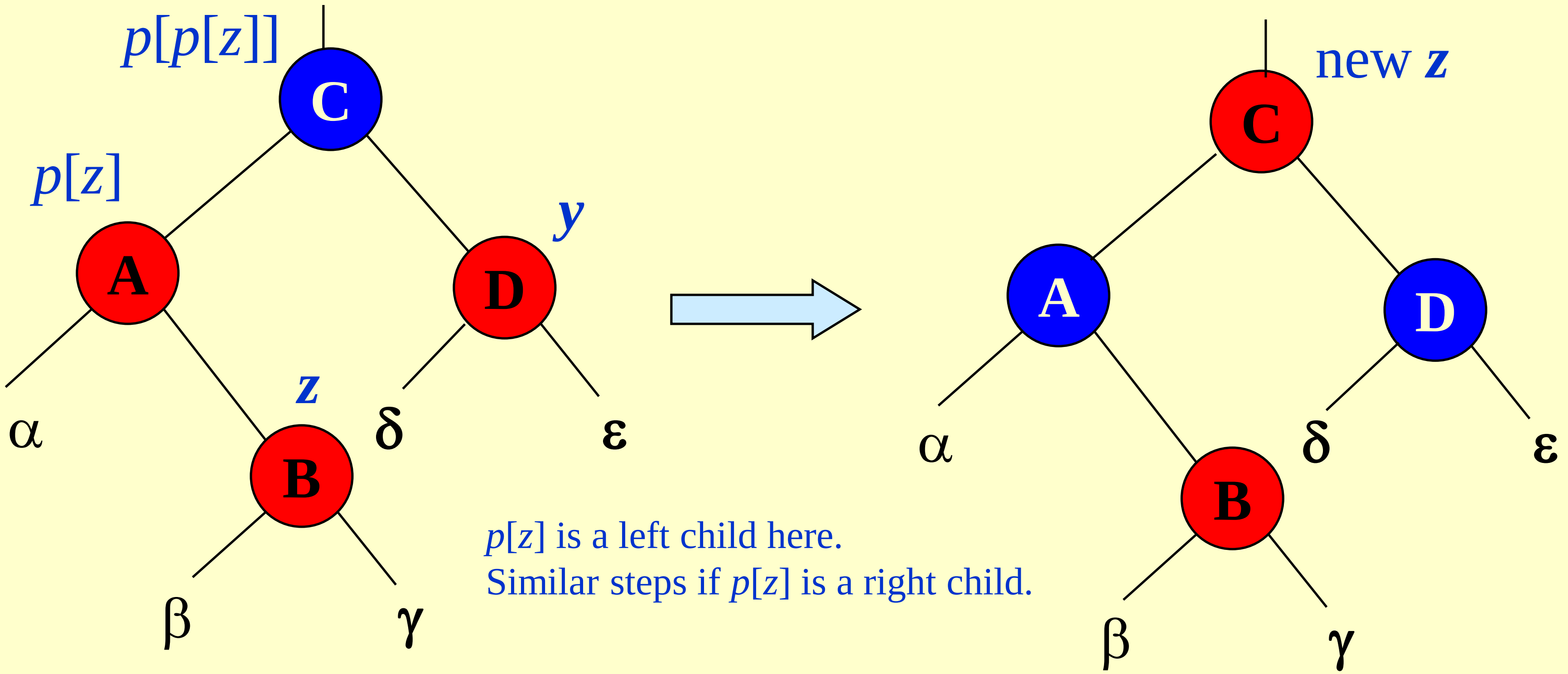
Loop invariant:

- At the start of each iteration of the **while** loop,
  - » z is red.
  - » If $p[z]$ is the root, then $p[z]$ is black.
  - » There is at most one red-black violation:
    - Property 2: z is a red root, or
    - Property 4: z and $p[z]$ are both red.

Correctness – Contd.

- **Initialization:** OK.
- **Termination:** The loop terminates only if $p[z]$ is black.
  Hence, property 4 is OK.
  The last line ensures property 2 always holds.
- **Maintenance:** We drop out when z is the root (since then $p[z]$ is sentinel $nil[T]$, which is black). When we start the loop body, the only violation is of property 4.
  - » There are 6 cases, 3 of which are symmetric to the other 3. We consider cases in which $p[z]$ is a left child.
  - » Let y be z's uncle ($p[z]$'s sibling).

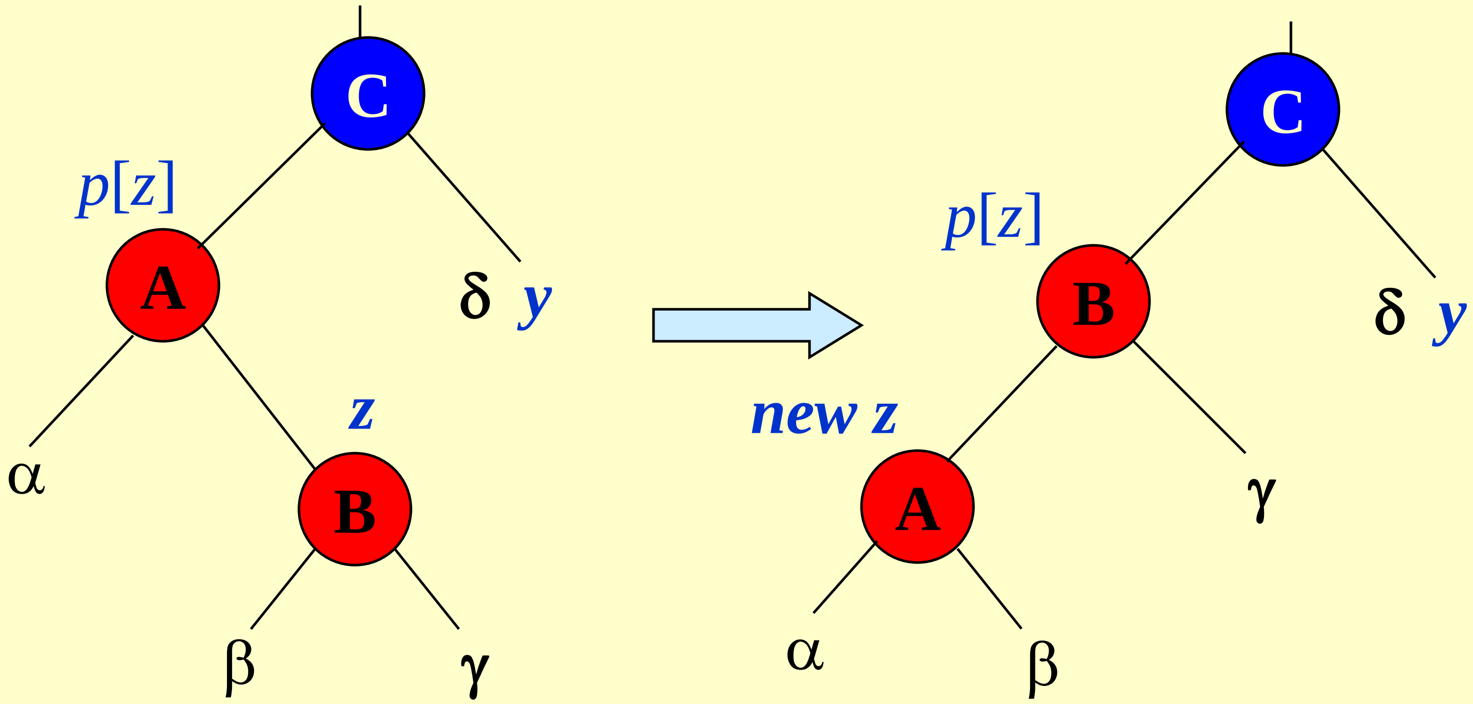
Case 1 – uncle *y* is red



$p[z]$ is a left child here.
Similar steps if $p[z]$ is a right child.

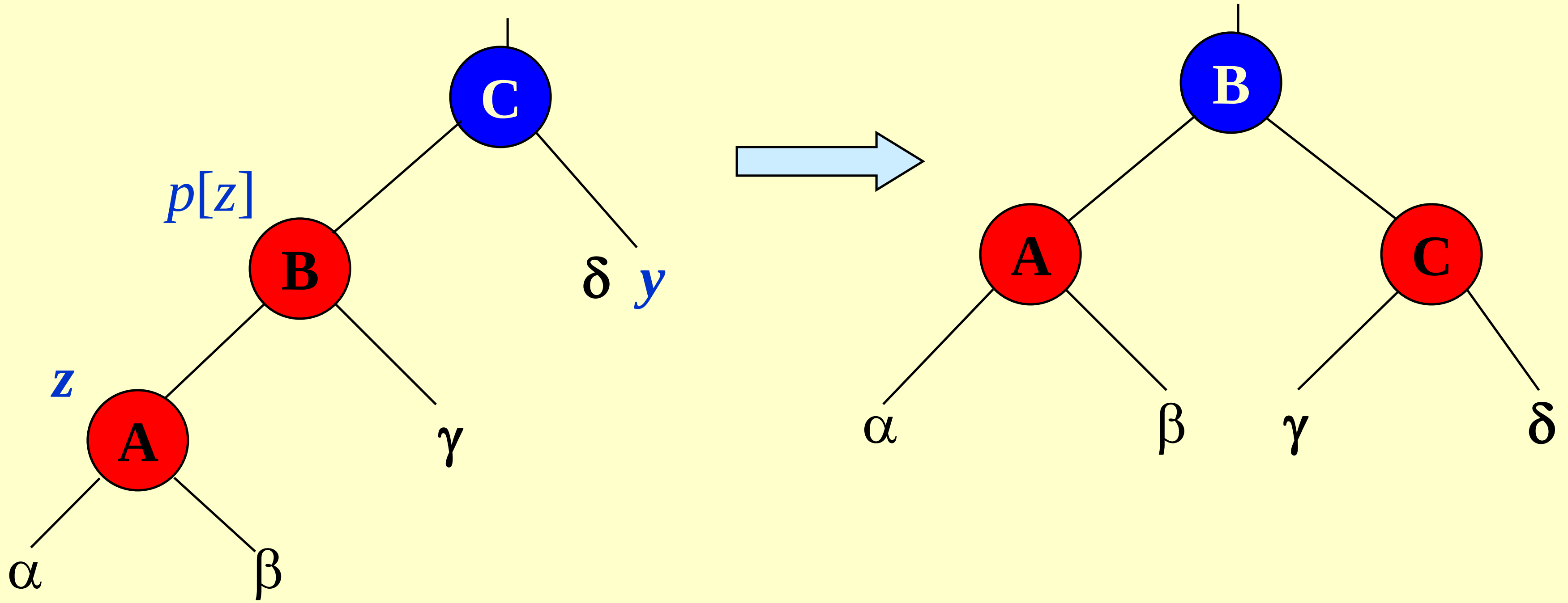
- $p[p[z]]$ (z's grandparent) must be black, since z and $p[z]$ are both red and there are no other violations of property 4.
- Make $p[z]$ and y black $\Rightarrow$ now z and $p[z]$ are not both red. But property 5 might now be violated.
- Make $p[p[z]]$ red $\Rightarrow$ restores property 5.
- The next iteration has $p[p[z]]$ as the new z (i.e., z moves up 2 levels).

Case 2 – *y* is black, *z* is a right child



- Left rotate around $p[z]$, $p[z]$ and z switch roles $\Rightarrow$ now z is a left child, and both z and $p[z]$ are red.
- Takes us immediately to case 3.

Case 3 – *y* is black, *z* is a left child



- Make $p[z]$ black and $p[p[z]]$ red.
- Then right rotate on $p[p[z]]$. Ensures property 4 is maintained.
- No longer have 2 reds in a row.
- $p[z]$ is now black $\Rightarrow$ no more iterations.

Algorithm Analysis

- $O(\lg n)$ time to get through RB-Insert up to the call of RB-Insert-Fixup.
- Within RB-Insert-Fixup:
  - » Each iteration takes $O(1)$ time.
  - » Each iteration but the last moves z up 2 levels.
  - » $O(\lg n)$ levels $\Rightarrow$ $O(\lg n)$ time.
  - » Thus, insertion in a red-black tree takes $O(\lg n)$ time.
  - » Note: there are at most 2 rotations overall.

Deletion

- Deletion, like insertion, should preserve all the RB properties.
- The properties that may be violated depends on the color of the deleted node.
  - » Red – OK. Why?
  - » Black?
- Steps:
  - » Do regular BST deletion.
  - » Fix any violations of RB properties that may be caused by a deletion.

Deletion

RB-Delete(*T*, *z*)

```
if left[z] = nil[T] or right[z] = nil[T]
    then y ← z
    else y ← TREE-SUCCESSOR(z)
if left[y] ≠ nil[T]
    then x ← left[y]
    else x ← right[y]
p[x] ← p[y]   // Do this, even if x is nil[T]
```

Deletion

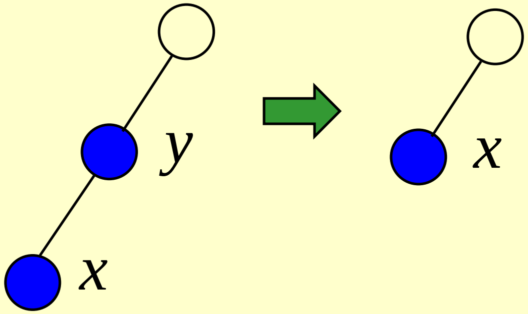
RB-Delete (*T, z*) (Contd.)

```
if p[y] = nil[T]
    then root[T] ← x
    else if y = left[p[y]] (*if y is a left child.*)
            then left[p[y]] ← x
            else right[p[y]] ← x  (*if y is a right child.*)
if y ≠ z
    then key[z] ← key[y]
         copy y's satellite data into z
if color[y] = BLACK
    then RB-Delete-Fixup(T, x)
return y
```

The node passed to the fixup routine is the only child of the spliced up node, or the sentinel.

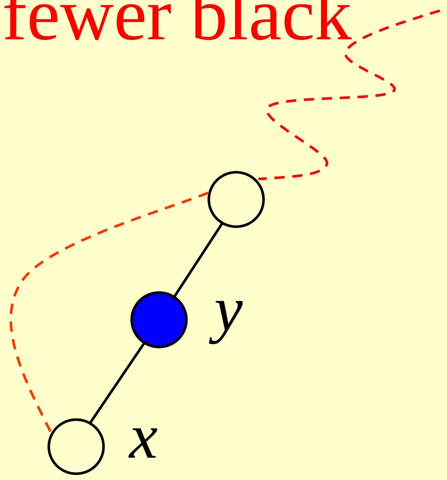
RB Properties Violation

- If *y* is black, we could have violations of red-black properties:
  - » Prop. 1. OK.
  - » Prop. 2. If *y* is the root and *x* is red, then the root has become red.
  - » Prop. 3. OK.
  - » Prop. 4. Violation if *p*[*y*] and *x* are both red.
  - » Prop. 5. Any path containing *y* now has 1 fewer black node.



RB Properties Violation

- Prop. 5. Any path containing *y* now has 1 fewer black node.
  - » Correct by giving *x* an "extra black."
  - » Add 1 to the count of black nodes on paths containing *x*.
  - » Now property 5 is OK, but property 1 is not.
  - » *x* is either ***doubly black*** (if *color*[*x*] = BLACK) or ***red & black*** (if *color*[*x*] = RED).
  - » The attribute *color*[*x*] is still either RED or BLACK. No new values for *color* attribute.
  - » In other words, the extra blackness on a node is by virtue of "*x* pointing to the node". (If a node is pointed to by *x*, it has an extra black.)



- Remove the violations by calling RB-Delete-Fixup.

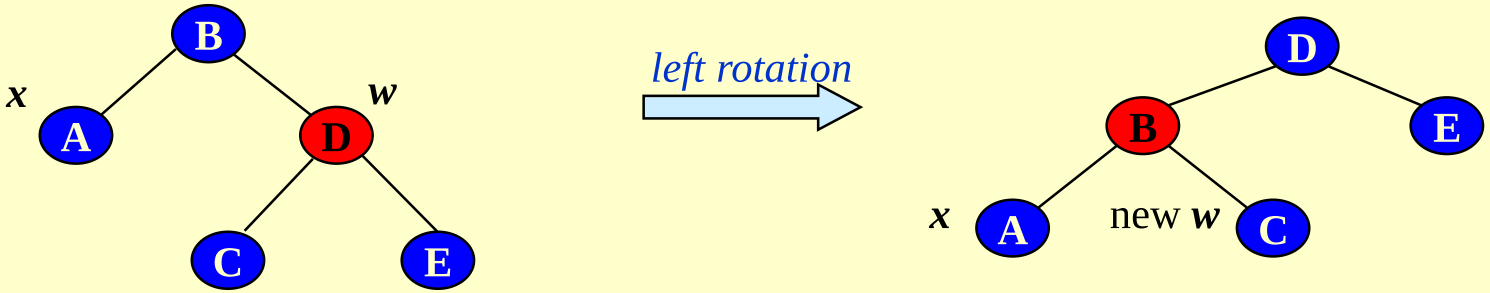
Deletion – Fixup

RB-Delete-Fixup(*T*, *x*)

```
while x ≠ root[T] and color[x] = BLACK
    do if x = left[p[x]]        //for cases 1 - 4
        then w ← right[p[x]]
            if color[w] = RED                      // Case 1
                then color[w] ← BLACK              // Case 1
                    color[p[x]] ← RED              // Case 1
                    LEFT-ROTATE(T, p[x])           // Case 1
                    w ← right[p[x]]                // Case 1
```

not necessary

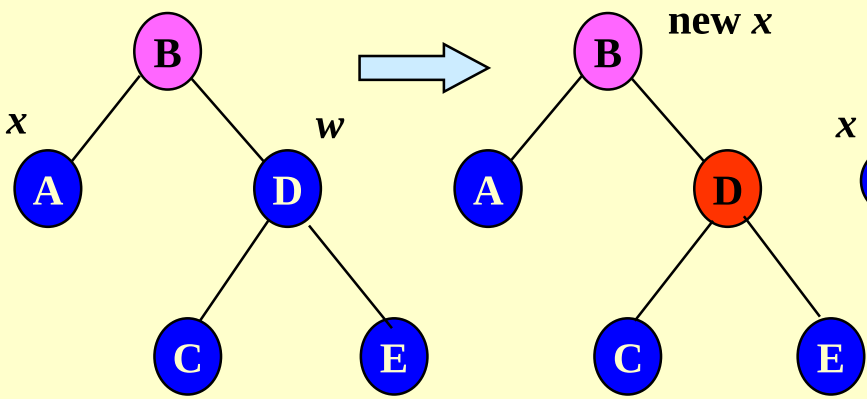
Case 1:



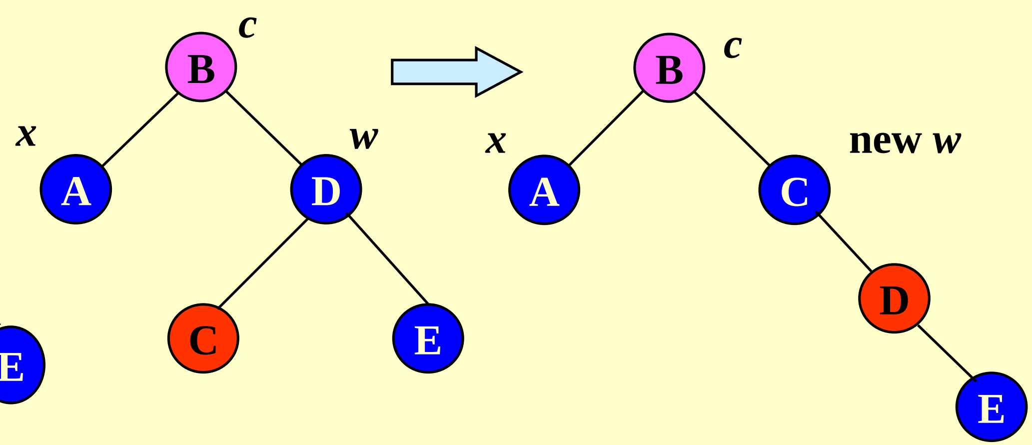
RB-Delete-Fixup(*T, x*) (Contd.)

```
        /* x is still left[p[x]] */
        if color[left[w]] = BLACK and color[right[w]] = BLACK
            then color[w] ← RED                                  // Case 2
                 x ← p[x]                                        // Case 2
            else if color[right[w]] = BLACK                      // Case 3
                    then color[left[w]] ← BLACK                  // Case 3
                         color[w] ← RED                          // Case 3
                         RIGHT-ROTATE(T, w)                      // Case 3
                         w ← right[p[x]]                         // Case 3
```

Case 2:



Case 3:

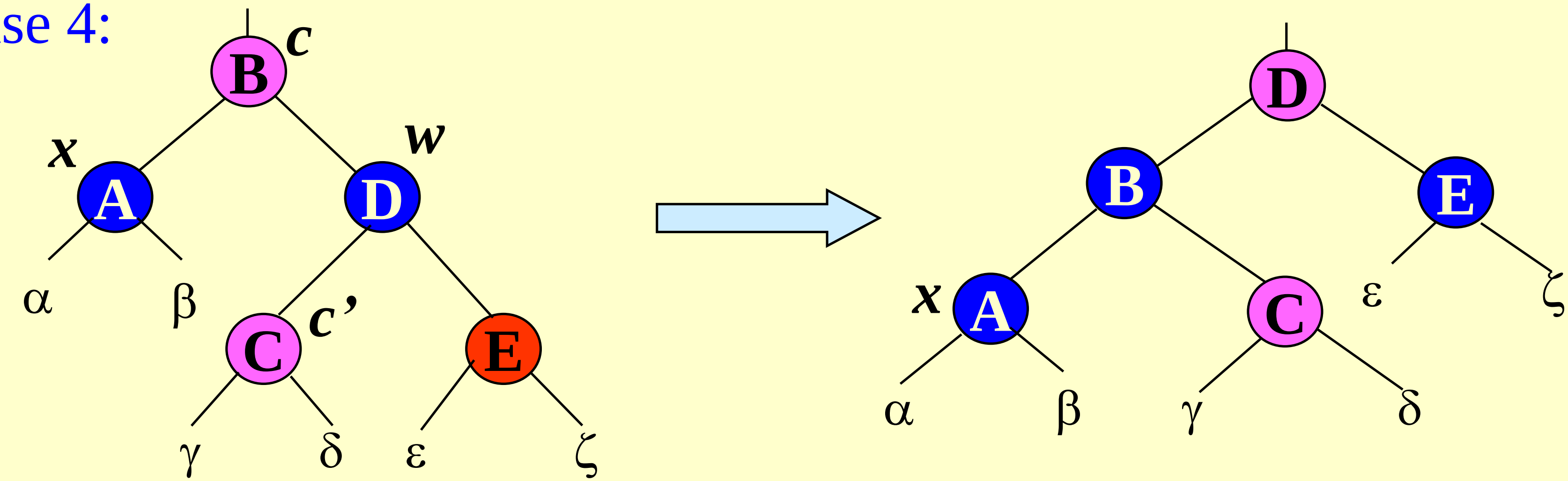


RB-Delete-Fixup(*T, x*) (Contd.)

```
        /* x is still left[p[x]] */
17.            color[w] ← color[p[x]]                 // Case 4
18.            color[p[x]] ← BLACK                    // Case 4
19.            color[right[w]] ← BLACK                // Case 4
20.            LEFT-ROTATE(T, p[x])                   // Case 4
21.            x ← root[T]                            // Case 4
22.      else (for cases 5 – 8, same as lines 3 - 21 with "right" and "left"
         exchanged)
23.  color[x] ← BLACK
```

to go out the while-loop (← line 21)

Case 4:



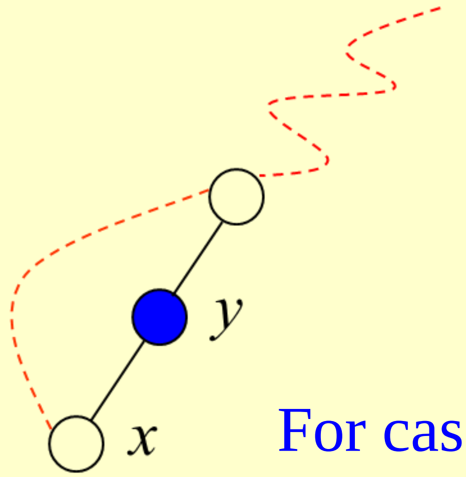
Deletion – Fixup

- ***Idea:*** Move the extra black (represented by *x*) up the tree until
- *x* points to a red node (this node is considered to be a red & black node since "*x* points to" means an extra black) ⇒ turn it into a black node,
- *x* points to the root ⇒ just remove the extra black, or
- We can do certain rotations and recoloring and finish.
- 8 cases in all, 4 of which are symmetric to the other. (4 cases for the situation that *x* is the left child of *p*[*x*]; 4 cases for the situation that *x* is the right child of *p*[*x*].)
- Within the **while** loop:
  - » *x* always points to a nonroot doubly black node.
  - » *w* is *x*'s sibling.
  - » *w* cannot be *nil*[*T*]. Otherwise, it would violate property 5 at *p*[*x*].

begin → case 1, case 2, case 3 → case 4

case 1 → case 2 — do something then terminate or repeat

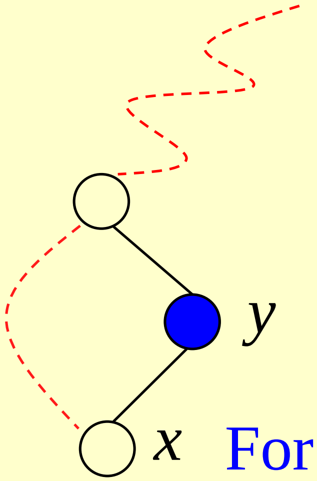
case 3 → case 4 — rotate, terminate



For cases 1 - 4, *x* is the left child of *p*[*x*] after *y* is deleted.

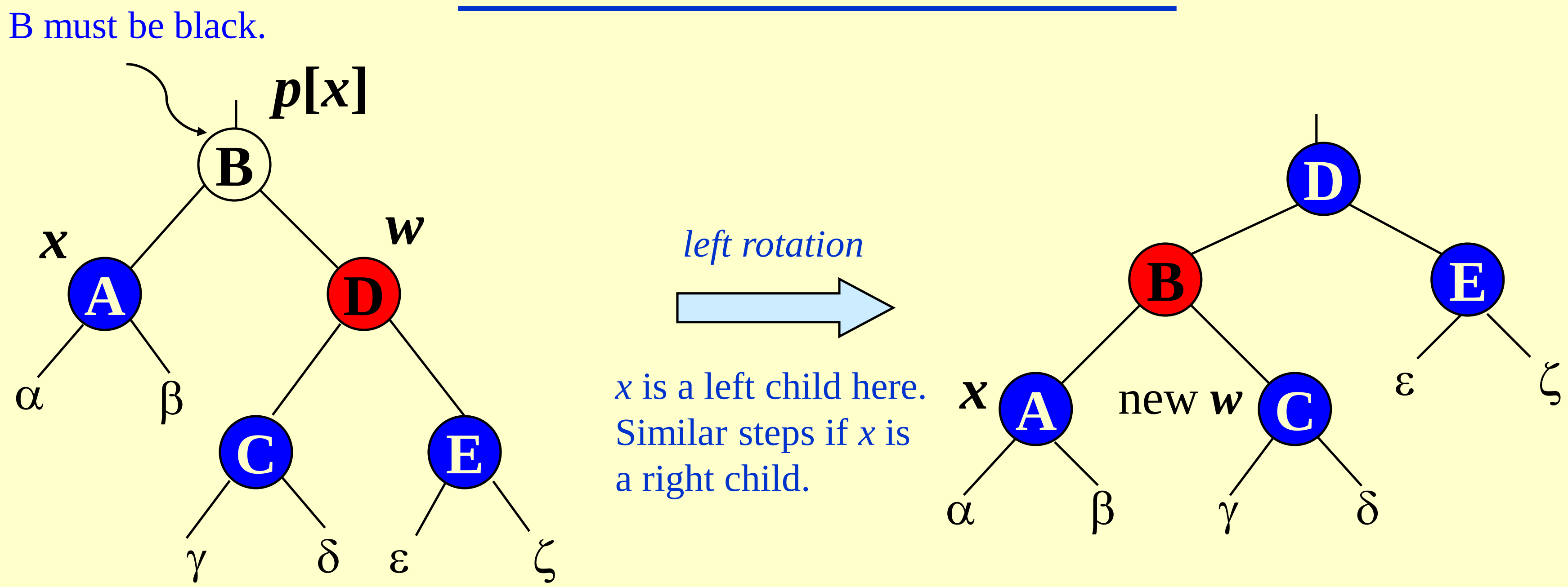
begin

- case 5
  - case 6 — do something then terminate or repeat
  - case 7
  - case 8 — rotate, terminate
- case 6
- case 7
- case 8



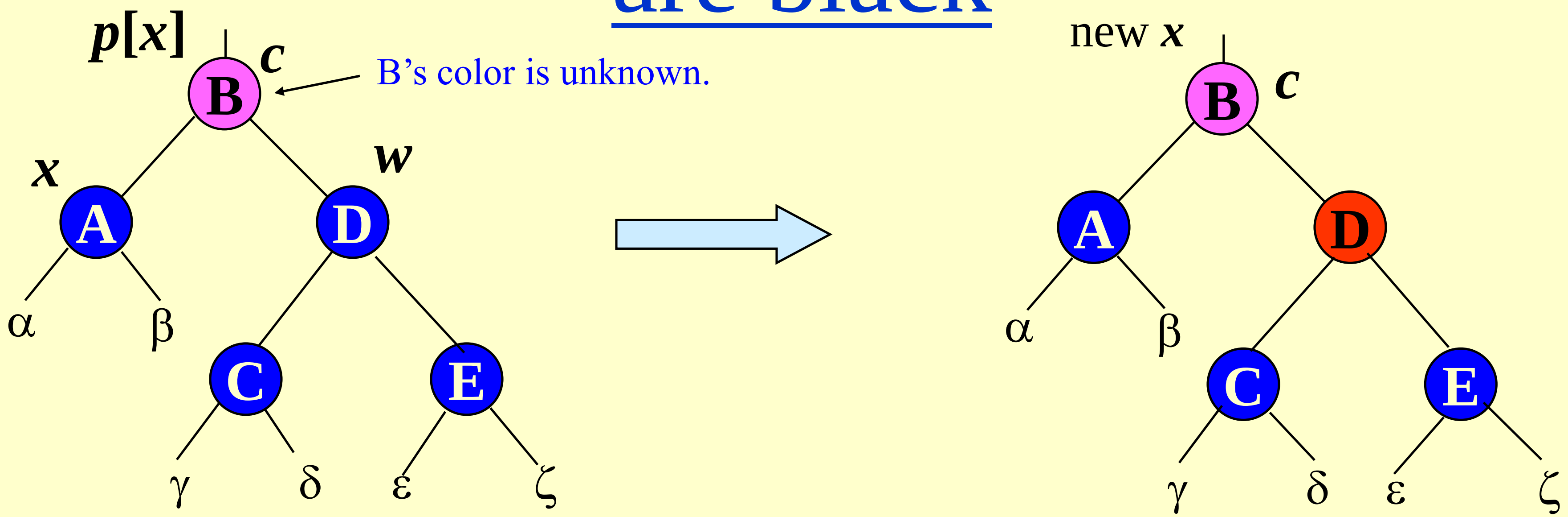
For cases 5 - 8, x is the right child of $p[x]$ after y is deleted.

Case 1 – *w* is red



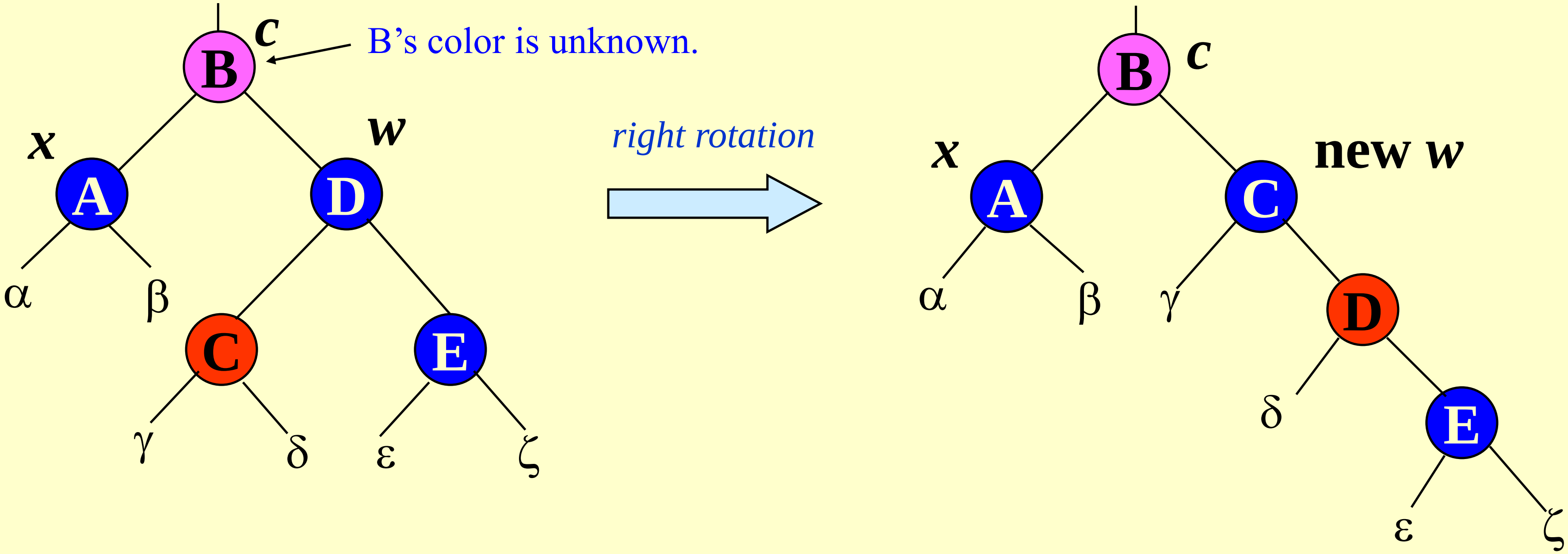
- *w* must have black children.
- Make *w* black and *p*[*x*] red (because *w* is red *p*[*x*] cannot be red).
- Then left rotate on *p*[*x*].
- New sibling of *x* was a child of *w* before rotation $\Rightarrow$ it must be black.
- Go immediately to case 2, case 3, or case 4.

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



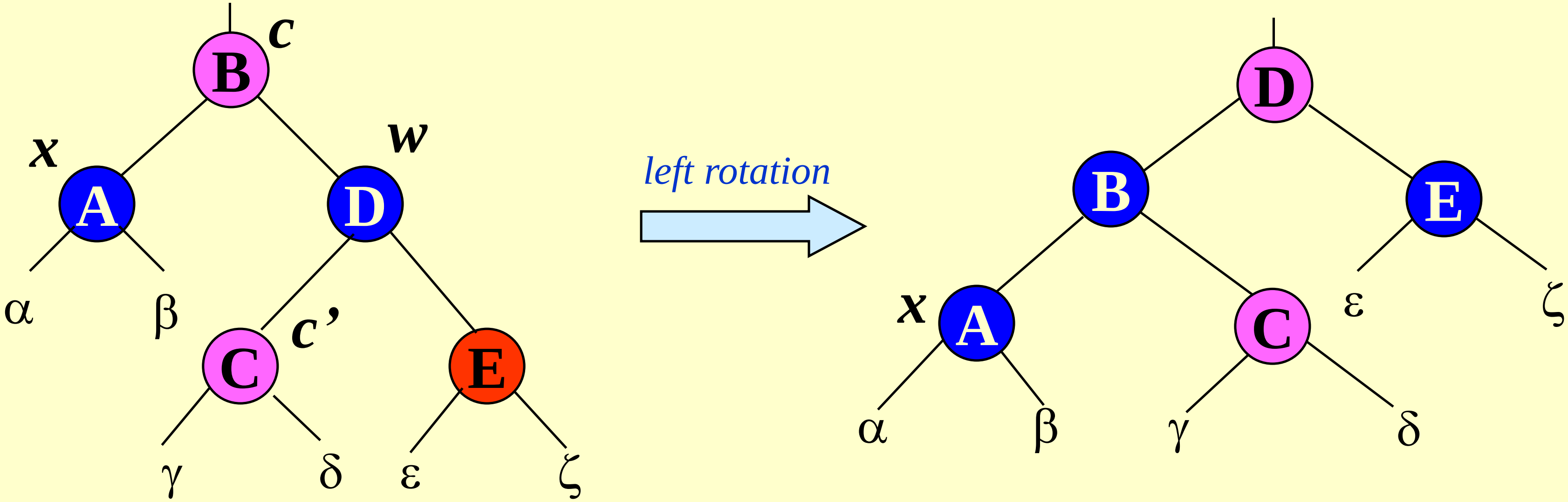
- Take 1 black off *x* (⇒ singly black) and 1 black off *w* (⇒ red).
- Move that black to *p*[*x*].
- Do the next iteration with *p*[*x*] as the new *x*.
- If entered this case from case 1, then *p*[*x*] was red ⇒ new *x* is red & black ⇒ color attribute of new *x* is RED ⇒ loop terminates. Then new *x* is made black in the last line of the algorithm.

Case 3 – *w* is black, *w*'s left child is red, *w*'s right child is black



- Make *w* red and *w*'s left child black.
- Then right rotate on *w*.
- New sibling *w* of *x* is black with a red right child ⇒ case 4.

Case 4 – *w* is black, *w*'s right child is red



- Make *w* be *p*[*x*]'s color (*c*).
- Make *p*[*x*] black and *w*'s right child black.
- Then left rotate on *p*[*x*].
- Remove the extra black on *x* (⇒ *x* is now singly black) without violating any red-black properties.
- All done. Setting *x* to root (see line 21 in the algorithm) causes the loop to terminate.

Analysis

- $O(\lg n)$ time to get through RB-Delete up to the call of RB-Delete-Fixup.
- Within RB-Delete-Fixup:
  - » Case 2 is the only case in which more iterations occur.
    - x moves up 1 level.
    - Hence, $O(\lg n)$ iterations.
  - » Each of cases 1, 3, and 4 has 1 rotation $\Rightarrow \leq 3$ rotations in all.
  - » Hence, $O(\lg n)$ time.

Hysteresis : or the value of lazyness

- The red nodes give us some slack – we don’t have to keep the tree perfectly balanced.
- The colors make the analysis and code much easier than some other types of balanced trees.
- Still, these aren’t free – balancing costs some time on insertion and deletion.