Announcements

- **Midterm on Thursday, Nov 3, 2005**
  - **8.5”x11” notes (2-sided) allowed**
  - **Calculators OK**
  - **Closed book**
  - **Partial credits**

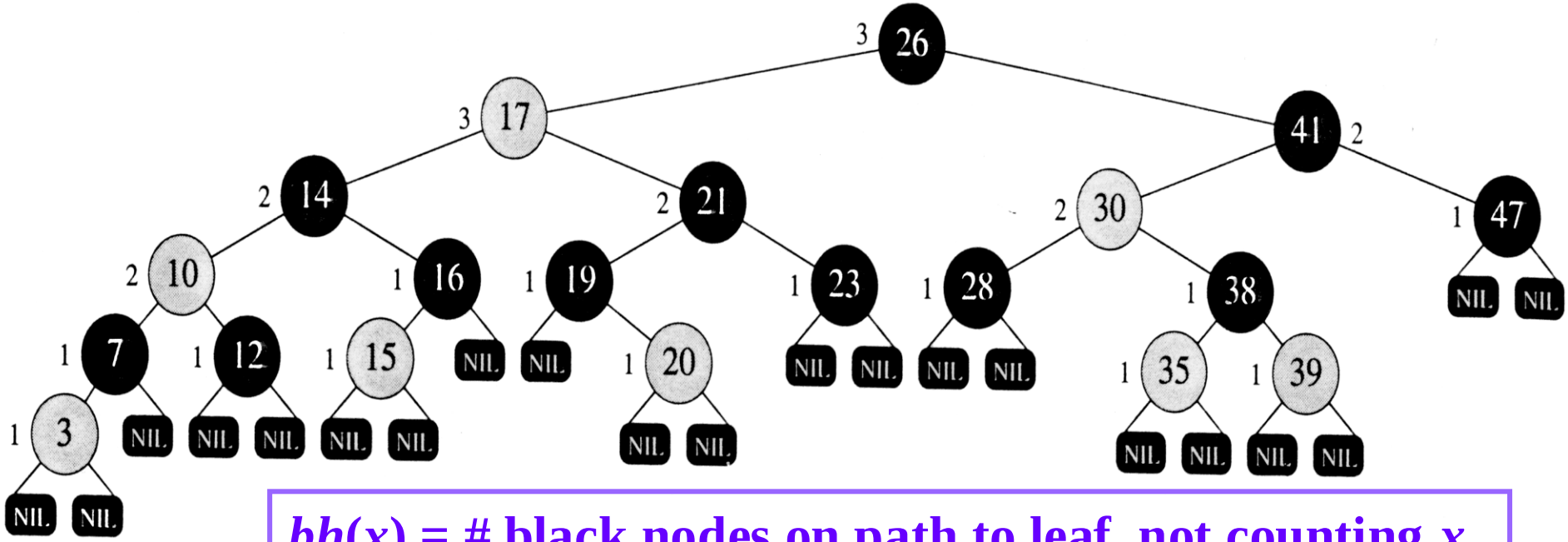
Red-Black Tree

- A binary search tree with 1 extra bit of storage per node, its color (Red or Black). By constraining the way nodes can be colored on any path from the root to a leaf, it ensures that *no path is more than twice as long as any other*--the tree is *approximately balanced*.

Red-Black Properties

- Every node is either red or black.
- Every leaf (NIL) is black
- If a node is red, then both its children are black
- Every simple path from a node to a descendant leaf contains the same number of black nodes

Example of Red-Black Tree



***bh*(*x*) = # black nodes on path to leaf, not counting *x*.**

Figure 14.1 A red-black tree with black nodes darkened and red nodes shaded. Every node in a red-black tree is either red or black, every leaf (NIL) is black, the children of a red node are both black, and every simple path from a node to a descendant leaf contains the same number of black nodes. Each non-NIL node is marked with its black-height; NIL's have black-height 0.

Lemma "R-B Height"

- The longest simple path from a node in a R-B tree to a descendent leaf has length at most twice that of the shortest simple path from this node to a descendent leaf.
  - In the longest path, at least every other node is black (R-B tree property 3) and it has height $h(x)$. Only half of the nodes in the longest path are black. In the shortest path, at most every node is black and has height $bh(x)$. Since the two paths contain equal numbers of black nodes by R-B tree property 4, the length of the longest path is at most twice the length of the shortest path, i.e. $bh(x) \geq h(x)/2 \Rightarrow h(x) \leq 2*bh(x)$

Lemma 14.1

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

Proof :

First, we show that any subtree rooted at node x contains at least $2^{bh(x)} - 1$ internal nodes by induction.

- If the height of x is 0, then x must be a leaf (NIL), and subtree rooted at x contains at least $2^{bh(x)} - 1 = 2^0 - 1 = 0$ internal nodes.
- Now, consider a node that has positive height and is an internal node with 2 children. Each child has a black-height of either $bh(x)$ or $bh(x) - 1$, depending on whether its color is R or B respectively. Since the height of a child of x is less than the height of x itself, we can apply the inductive hypothesis to state that each child has $\geq 2^{bh(x)-1} - 1$ internal nodes. So, the subtree rooted at x has $(2^{bh(x)-1} - 1)+(2^{bh(x)-1} - 1)+1= 2^{bh(x)} - 1$ internal nodes.

Lemma 14.1

Proof (cont):

Next, let h be the height of the tree. According to R-B Tree Property 3, at least half of the nodes on any simple path from the root to a leaf, not including the root, must be black. Consequently the black height of the root must be at least $h/2$; thus for x = root

$$n \geq 2^{bh(x)} - 1 \geq 2^{h/2} - 1$$

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

$$\Rightarrow \lg(n+1) \geq h/2$$

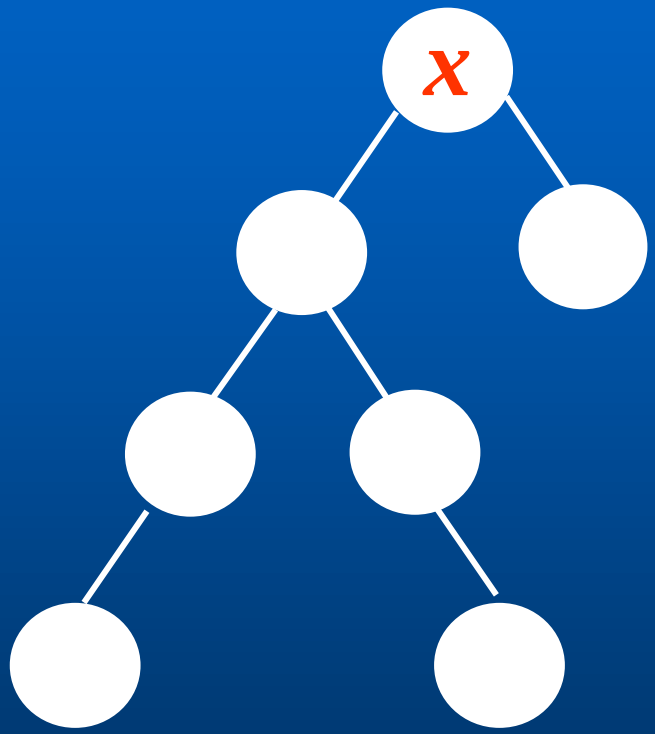
Therefore, $h \leq 2 \lg(n+1)$

$\Rightarrow$ All tree operations run in $O(\lg n)$ worst-case time

Points to Remember

- Every internal node has 2 children
- But leaf nodes are typically not shown
- Leaves carry no data

Exercise: Red-Black Tree

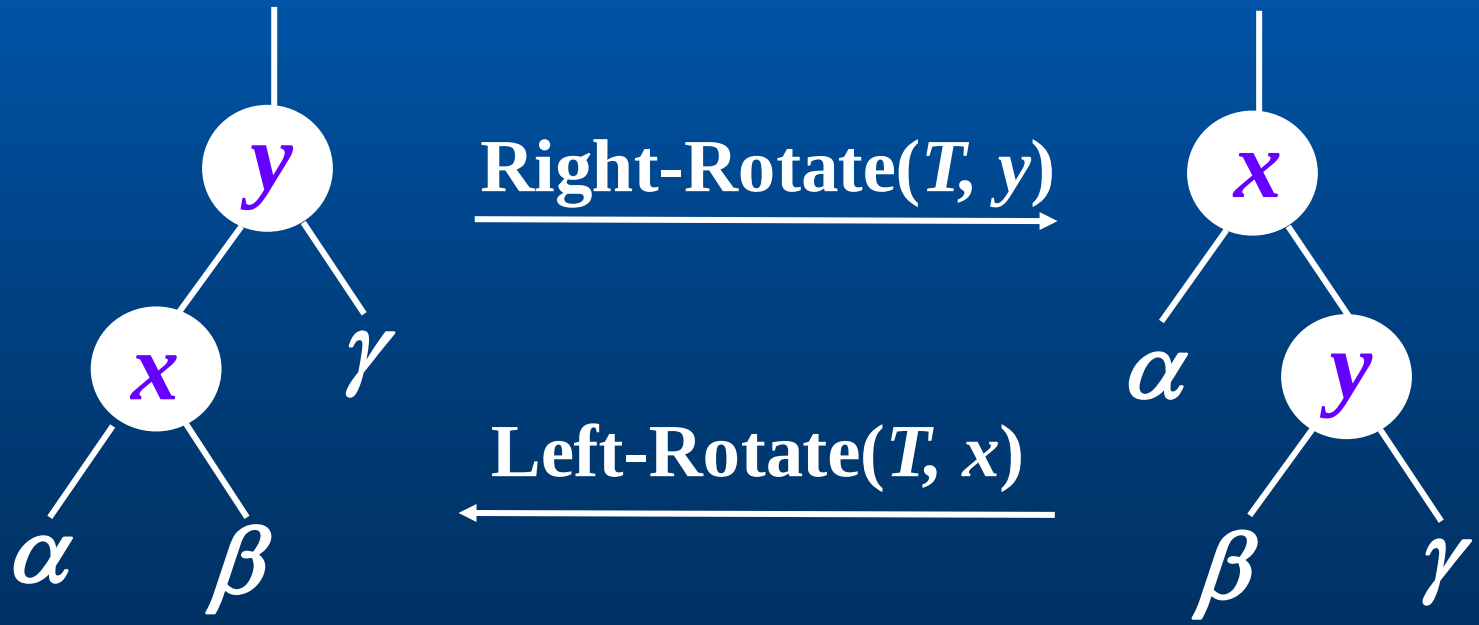


bh(x) =

h(x) =

Rotations

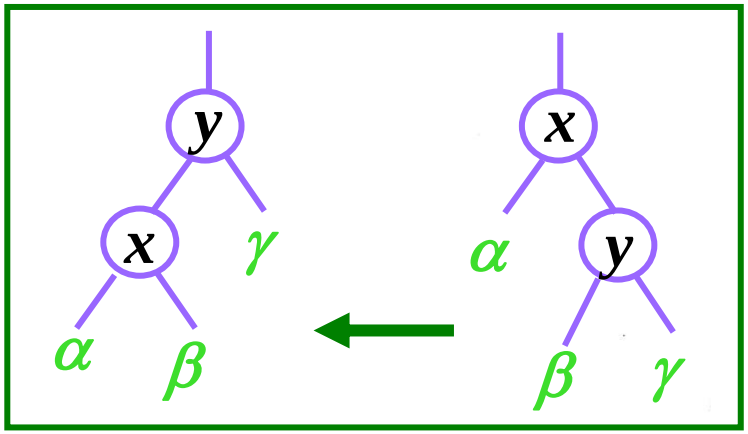
- We change the pointer structure through *rotation*, a local operation in a search tree that preserves the inorder key ordering.



LEFT-ROTATE(T, x)

- **This routine runs in $O(1)$ time.**

```
 1  y ← right[x]                ▷ Set y.
 2  right[x] ← left[y]          ▷ Turn y's left subtree into x's right subtree.
 3  if left[y] ≠ NIL
 4     then p[left[y]] ← x
 5  p[y] ← p[x]                 ▷ Link x's parent to y.
 6  if p[x] = NIL
 7     then root[T] ← y
 8     else if x = left[p[x]]
 9             then left[p[x]] ← y
10             else right[p[x]] ← y
11  left[y] ← x                 ▷ Put x on y's left.
12  p[x] ← y
```



Example of Left-Rotate Operation

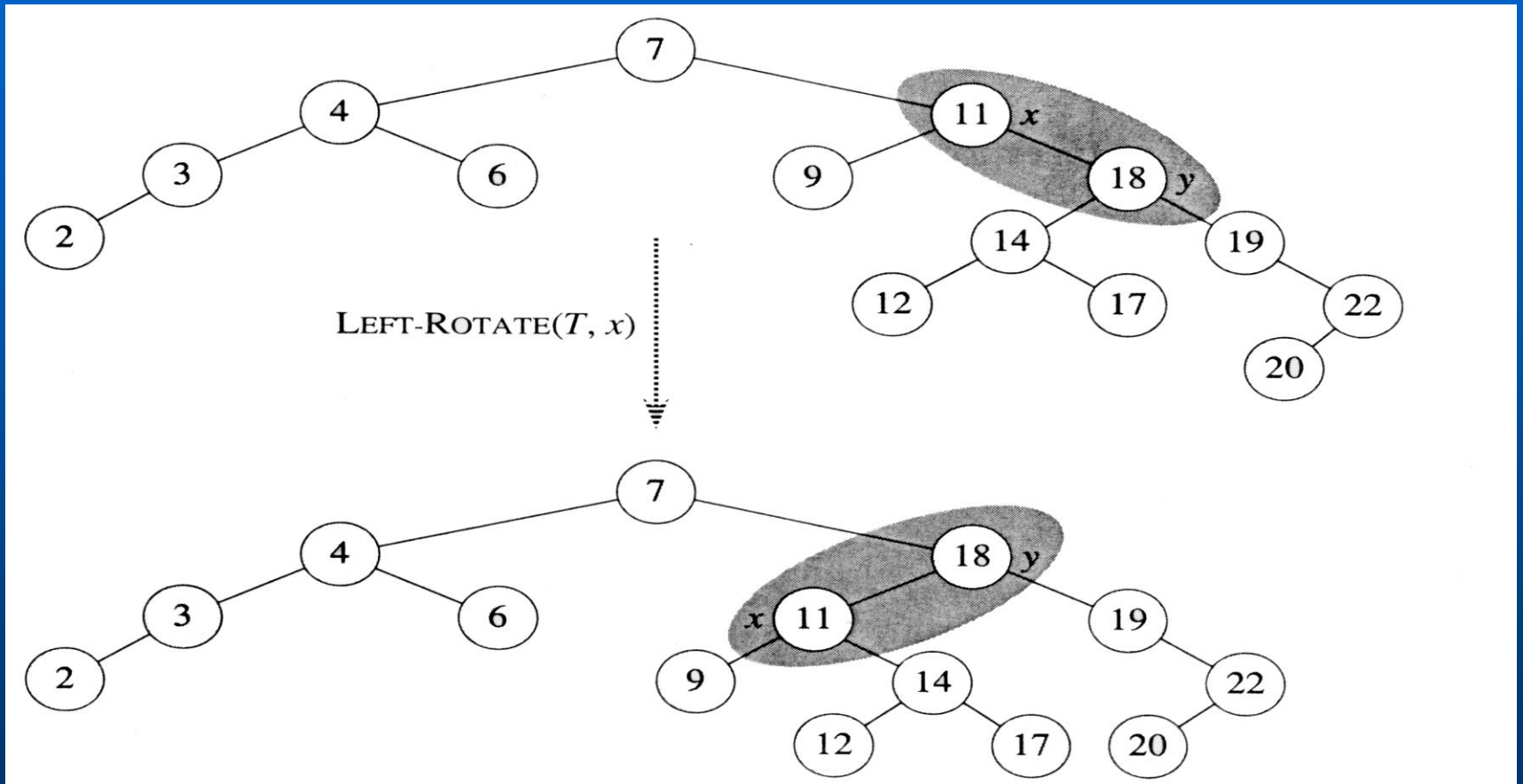
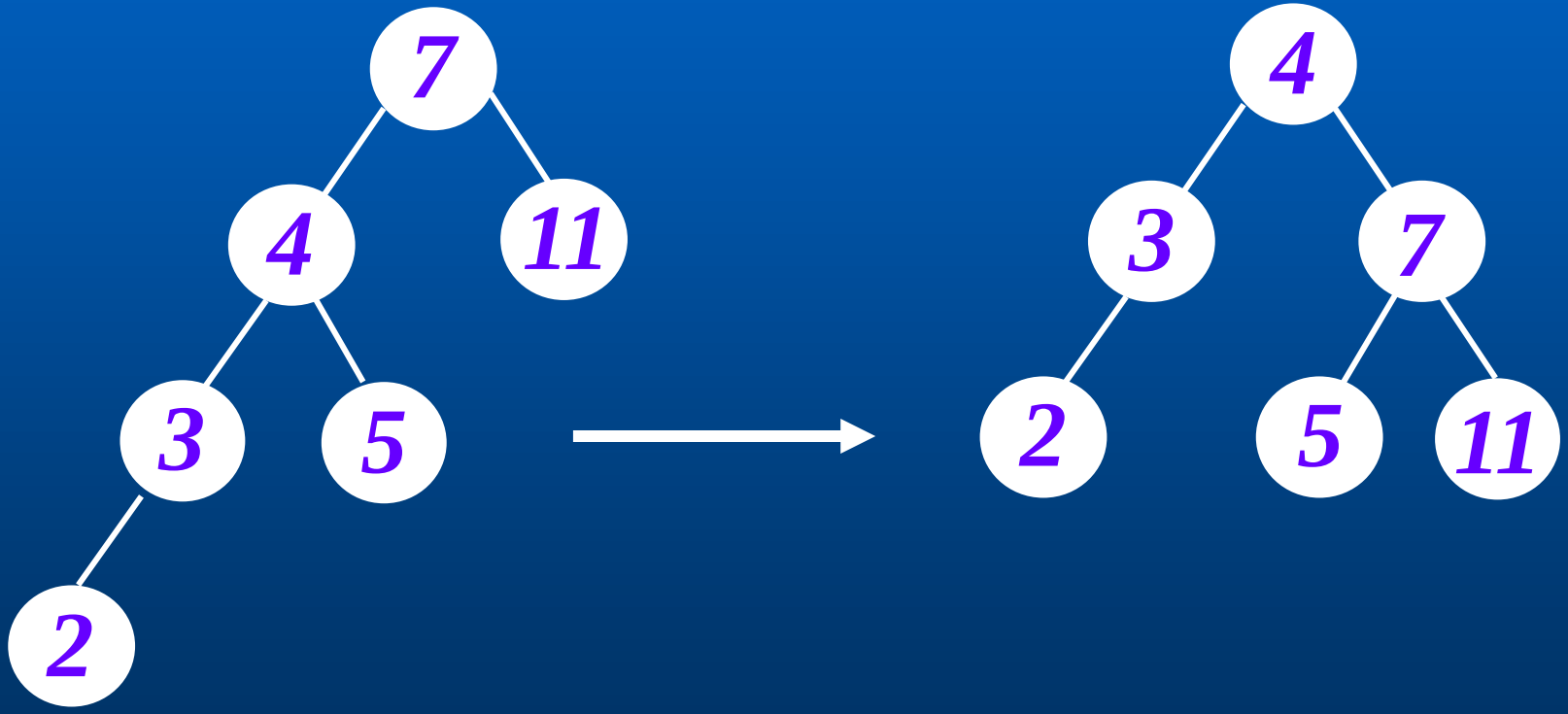


Figure 14.3 An example of how the procedure LEFT-ROTATE(T, x) modifies a binary search tree. The NIL leaves are omitted. Inorder tree walks of the input tree and the modified tree produce the same listing of key values.

Example of Right Rotate Operation

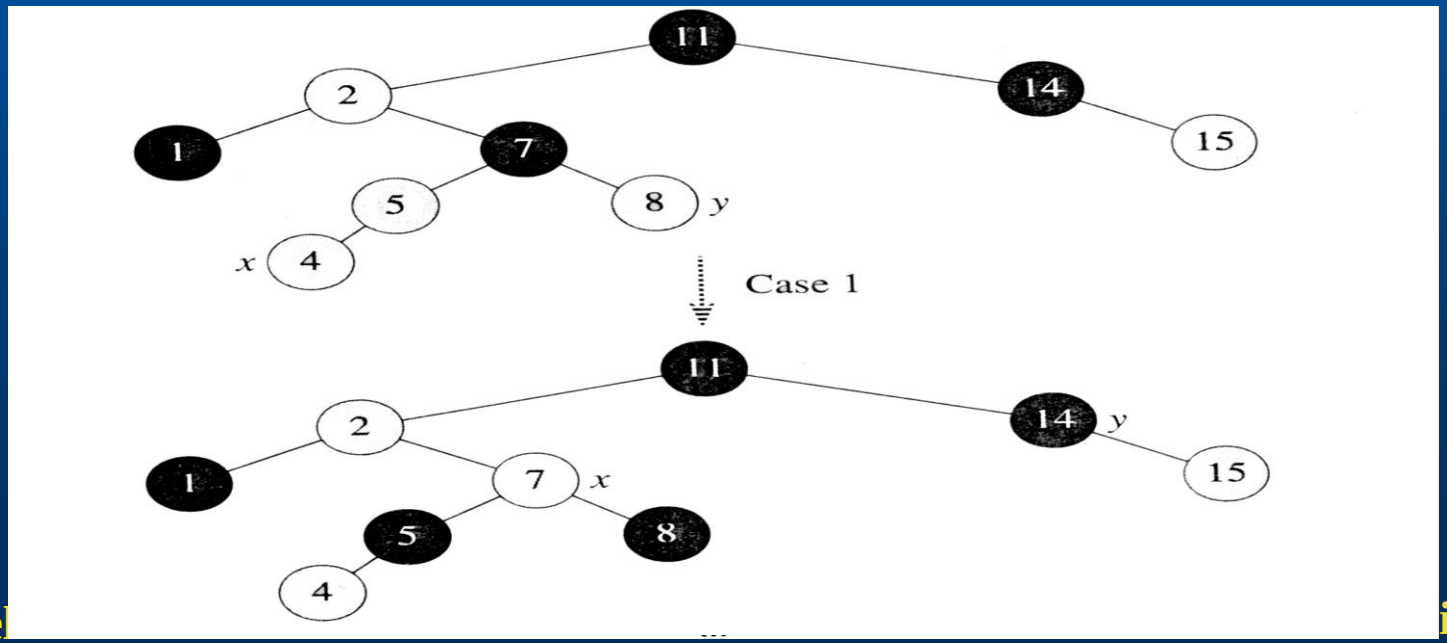


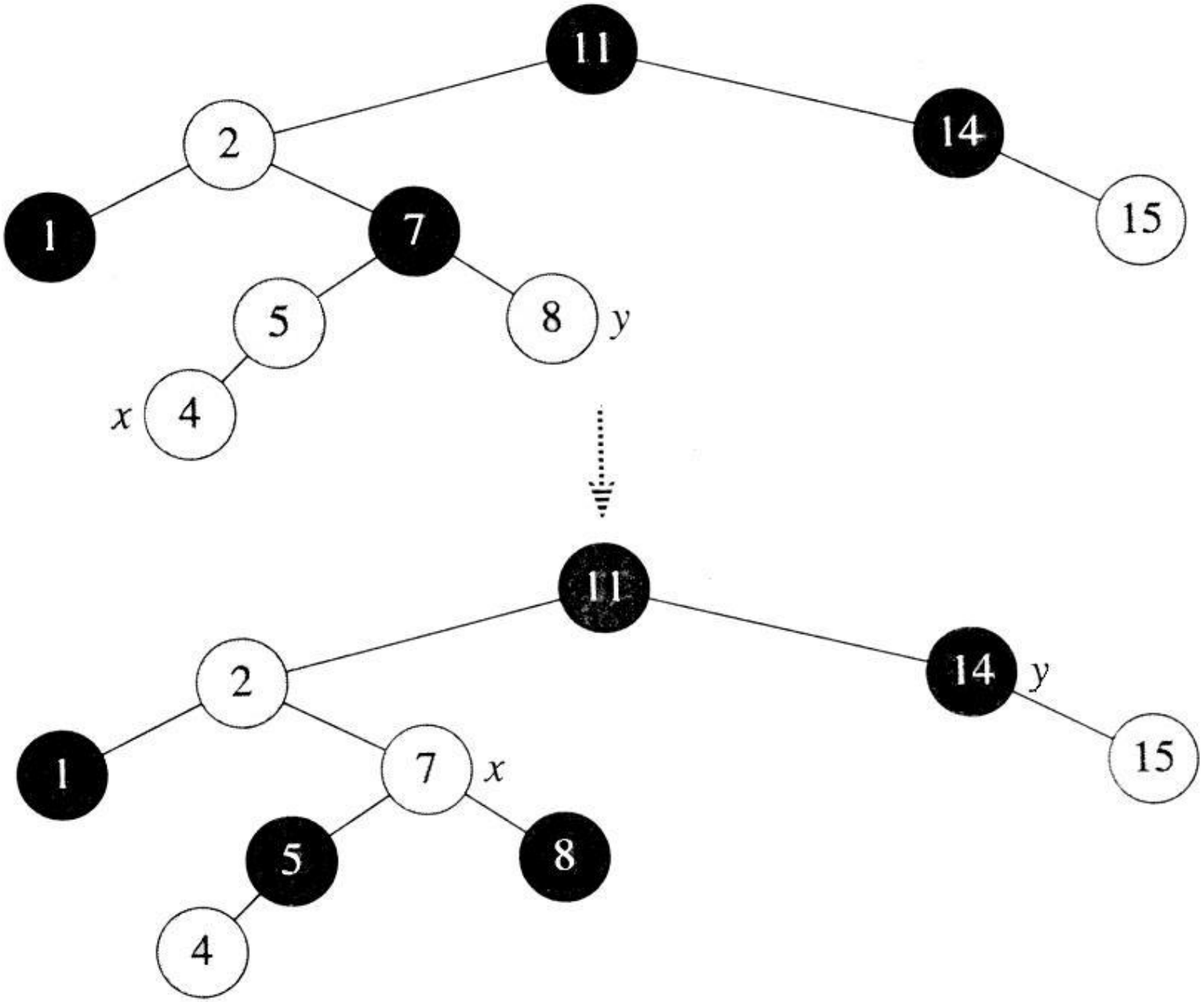
Insertion in R-B Trees

- It can be done in $O(\lg n)$ time.
- Basic steps:
  - Use Tree-Insert from BST to insert a node x into T
  - Color the node x red.
  - Fix up the modified tree by re-coloring nodes and performing rotation to preserve R-B tree property.

RB-Insert

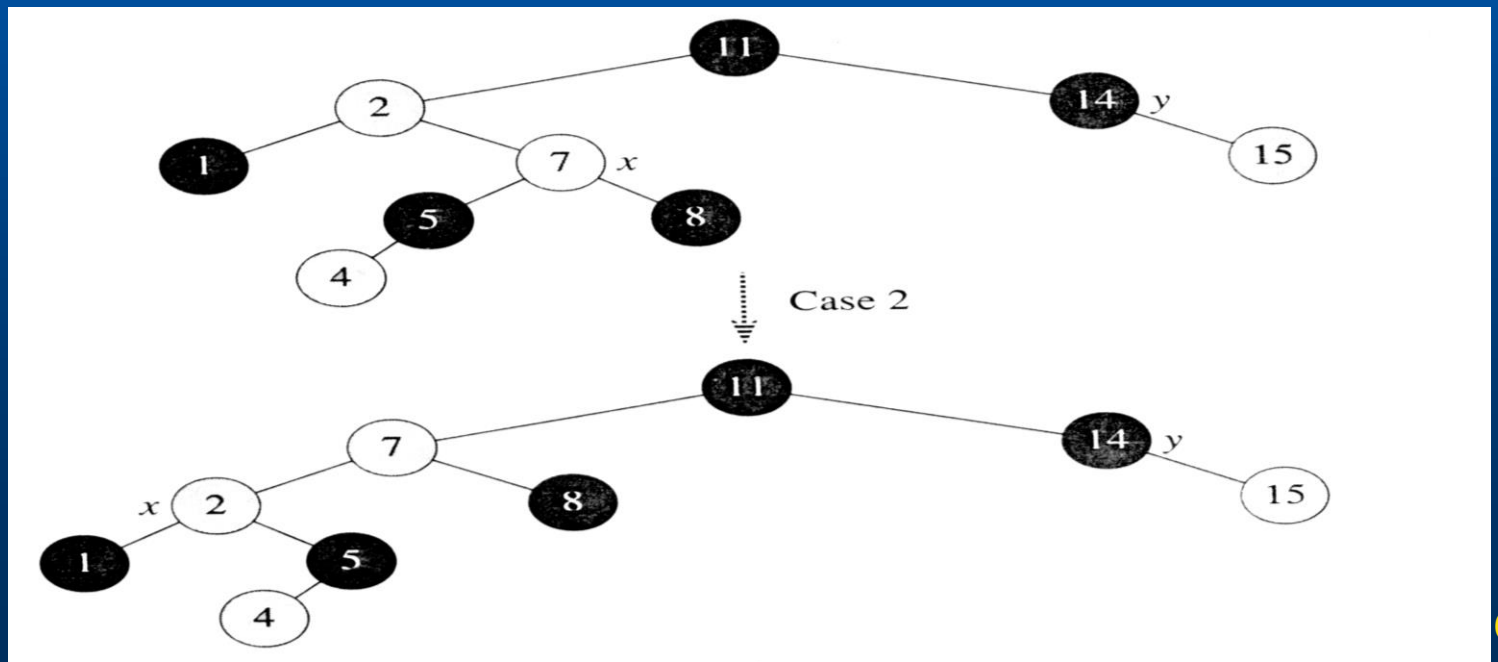
- Case 1 (the child x, its parent and its uncle y are red):
  - Change parent and uncle to black
  - Change grandparent to red (preserves black height)
  - Make grandparent new x, grandparent's uncle new y

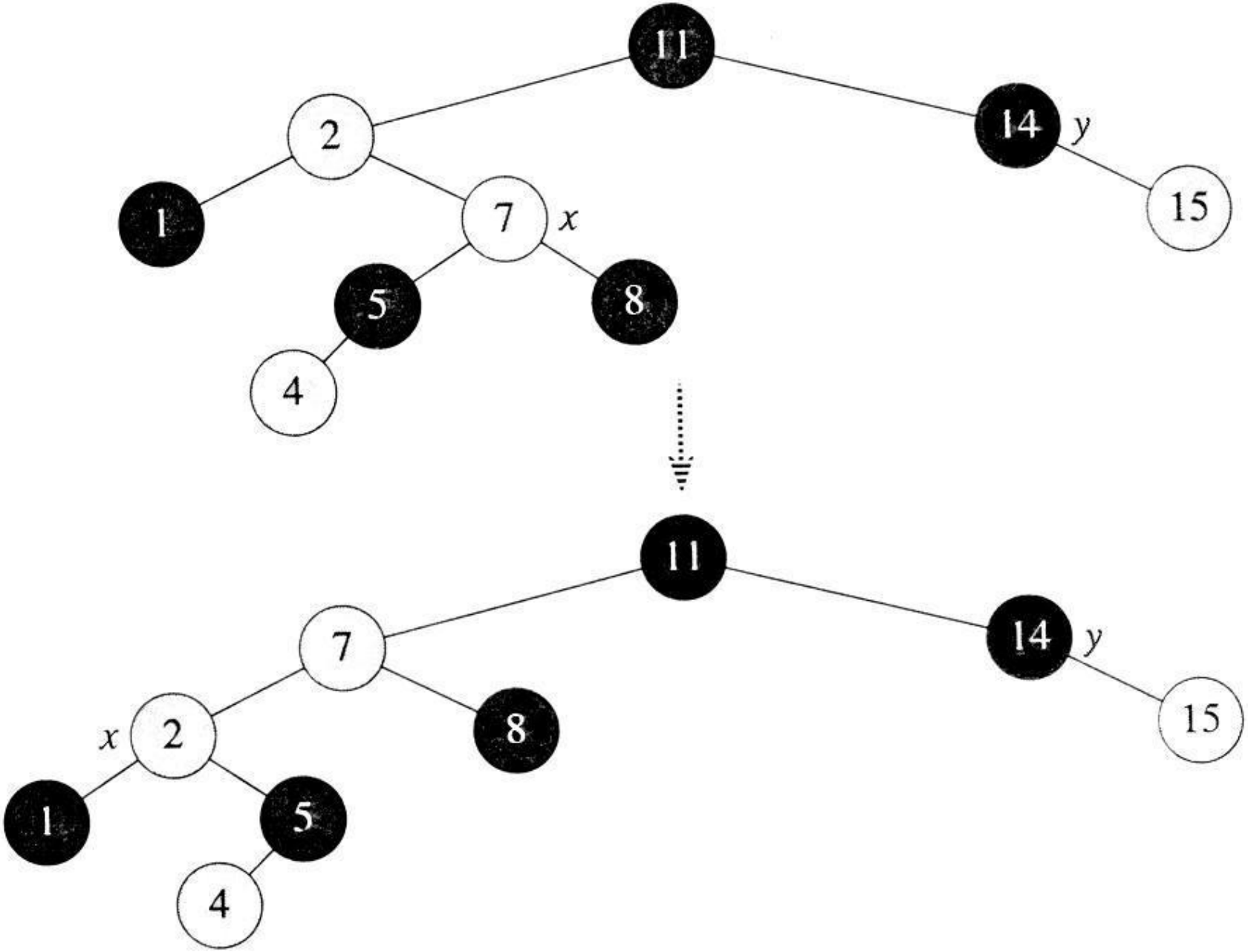



11
2
14
1
7
15
5
8
y
x
4
11
2
14
y
1
7
15
x
5
8
4

RB-Insert

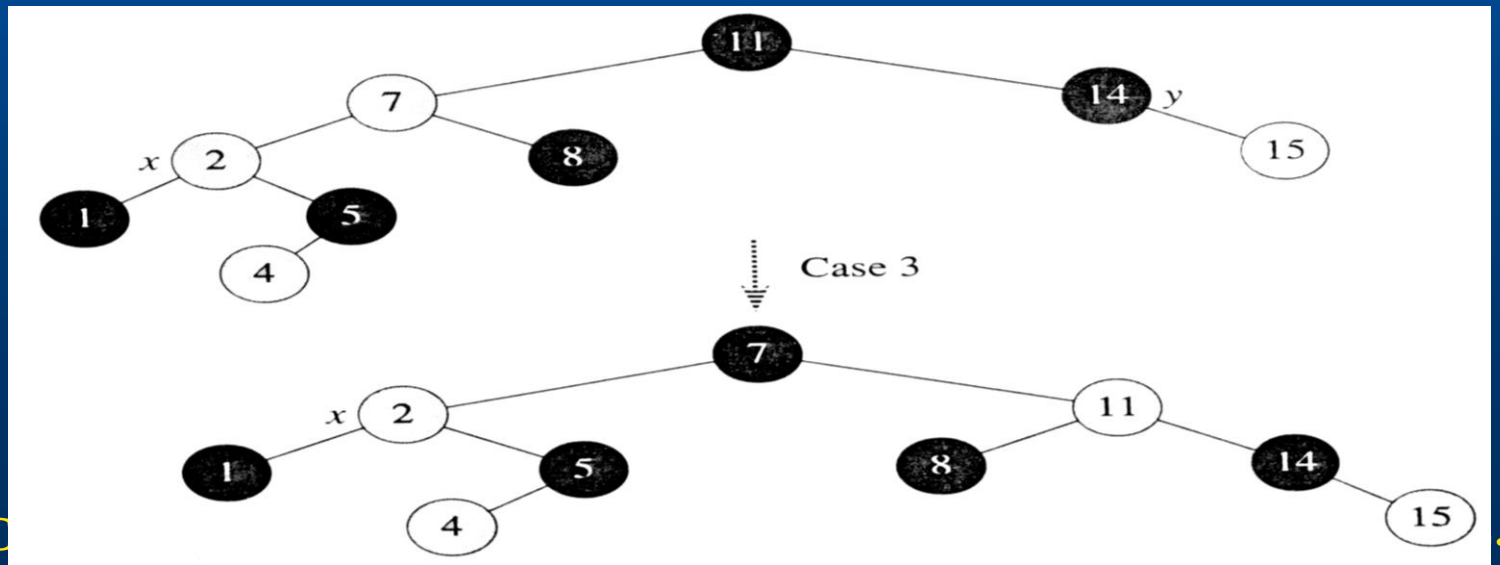
- Case 2 (x and its parent are red, y is black, and x is an "inside child"):
  - Rotate outwards on x's parent
  - Former parent becomes new x
  - Old x becomes parent

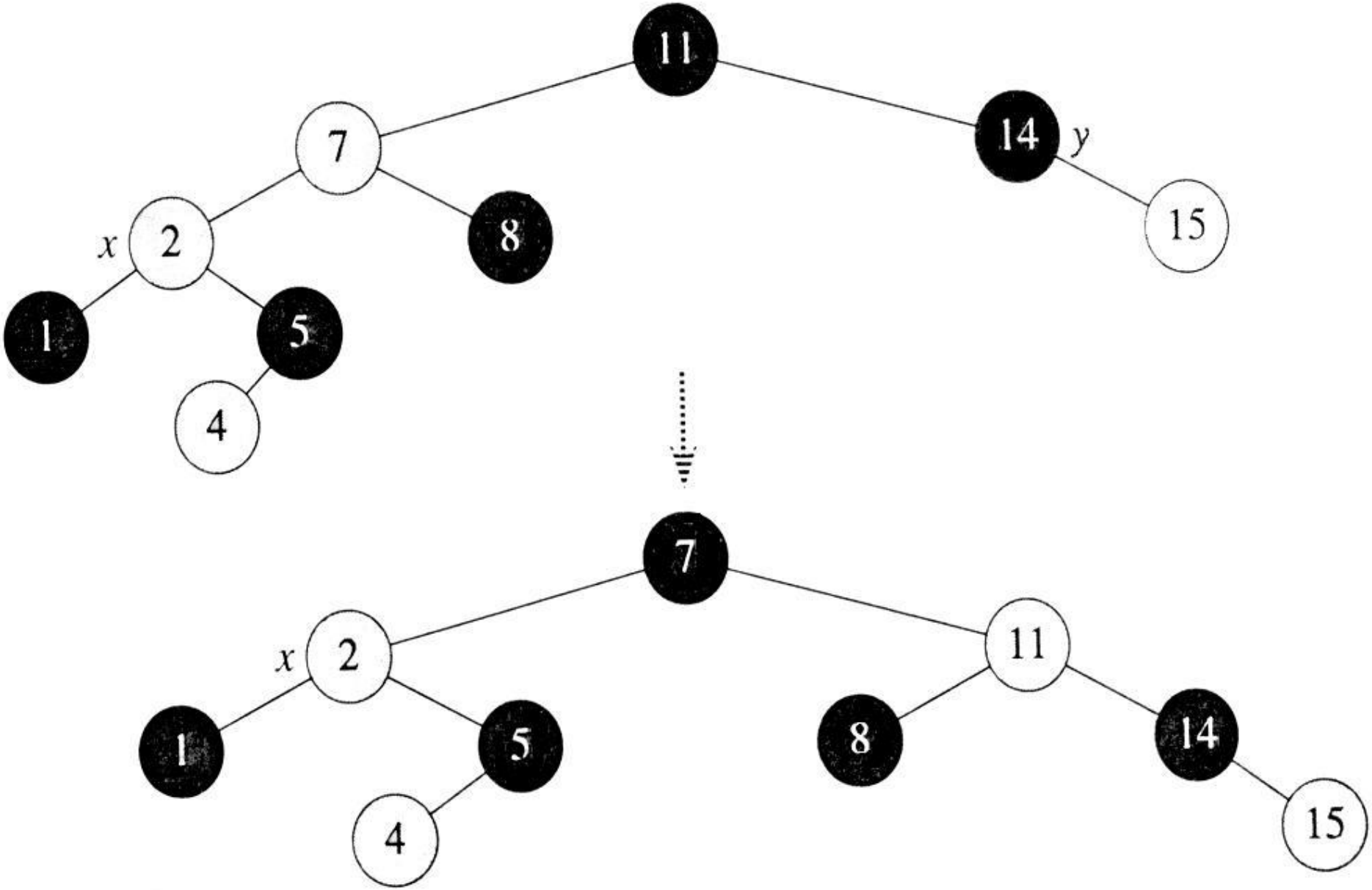



11
2
14
y
1
7
x
15
5
8
4
11
7
14
y
x
2
8
15
1
5
4

RB-Insert

- Case 3 (x and its parent are red, y is black, and x is an "outside child"):
  - Rotate "away from x" around x's grandparent
  - Color x's parent black
  - Now x's old grandparent is x's sibling; color it red to maintain the right number of black nodes down each path




11
7
14
y
x
2
8
15
1
5
4
7
x
2
11
1
5
8
14
4
15

RB-INSERT(T, x)

♦ **This routine runs in $O(lg\ n)$ time.**

```
 1  TREE-INSERT(T, x)
 2  color[x] ← RED
 3  while x ≠ root[T] and color[p[x]] = RED
 4        do if p[x] = left[p[p[x]]]
 5              then y ← right[p[p[x]]]
 6                   if color[y] = RED
 7                      then color[p[x]] ← BLACK                ▷ Case 1
 8                           color[y] ← BLACK                   ▷ Case 1
 9                           color[p[p[x]]] ← RED               ▷ Case 1
10                           x ← p[p[x]]                        ▷ Case 1
11                      else if x = right[p[x]]
12                              then x ← p[x]                   ▷ Case 2
13                                   LEFT-ROTATE(T, x)          ▷ Case 2
14                           color[p[x]] ← BLACK                ▷ Case 3
15                           color[p[p[x]] ← RED                ▷ Case 3
16                           RIGHT-ROTATE(T, p[p[x]])           ▷ Case 3
17              else (same as then clause
                      with "right" and "left" exchanged)
18  color[root[T]] ← BLACK
```

Example of RB-Insert Operation

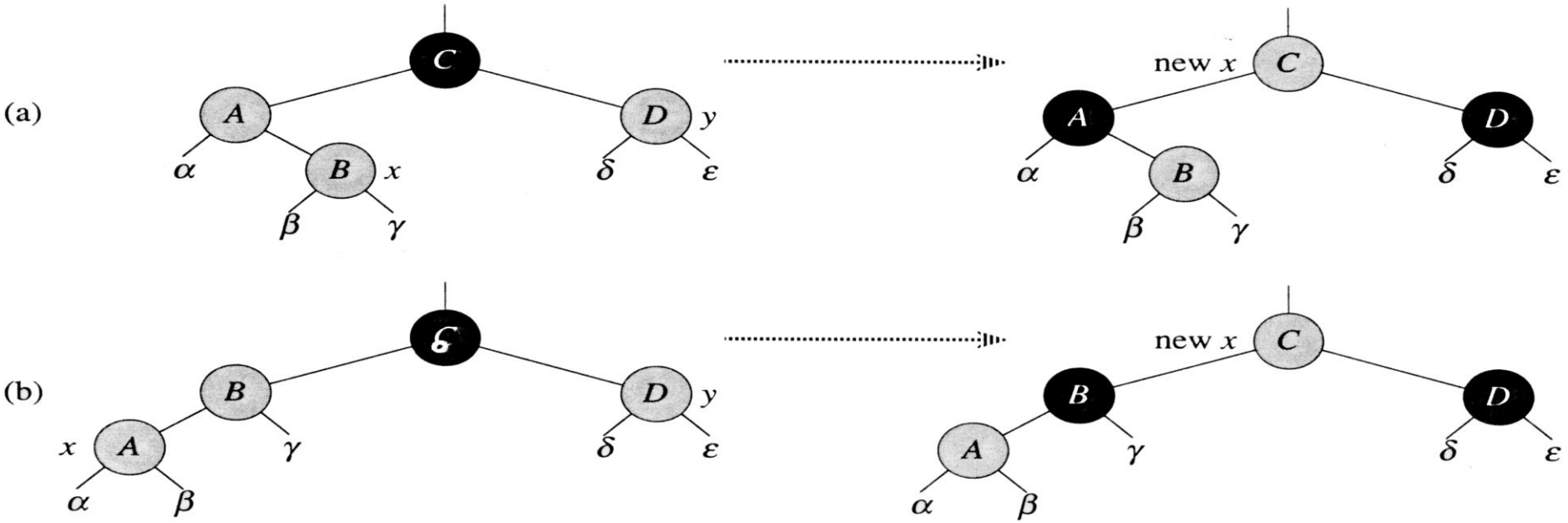


Figure 14.5 Case 1 of the procedure RB-INSERT. Property 3 is violated, since x and its parent $p[x]$ are both red. The same action is taken whether (**a**) x is a right child or (**b**) x is a left child. Each of the subtrees α, β, γ, δ, and ε has a black root, and each has the same black-height. The code for case 1 changes the colors of some nodes, preserving property 4: all downward paths from a node to a leaf have the same number of blacks. The **while** loop continues with node x's grandparent $p[p[x]]$ as the new x. Any violation of property 3 can now occur only between the new x, which is red, and its parent, if it is red as well.

RB-Insert

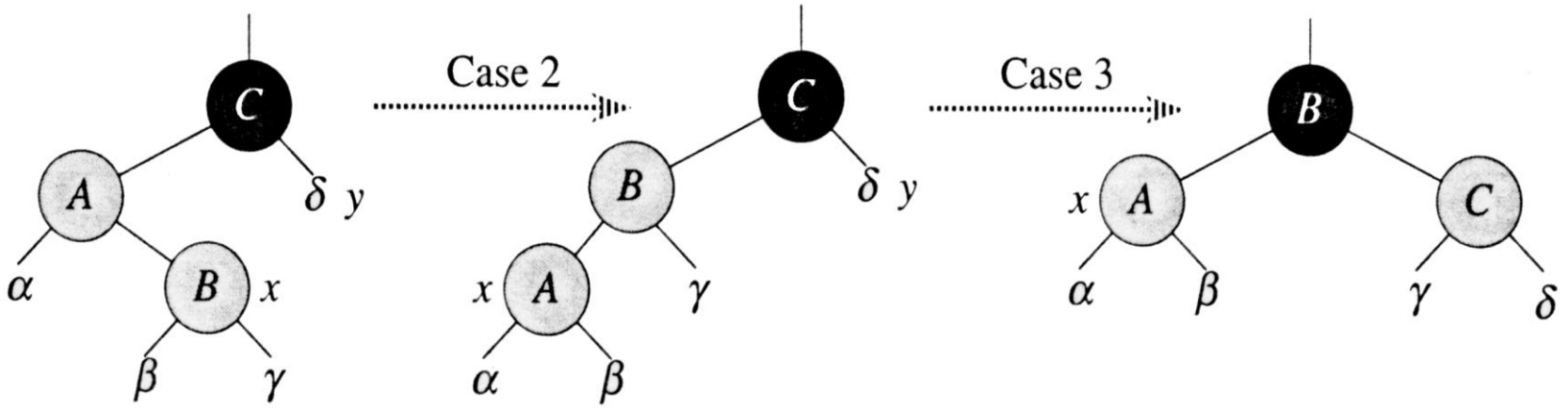


Figure 14.6 Cases 2 and 3 of the procedure RB-INSERT. As in case 1, property 3 is violated in either case 2 or case 3 because x and its parent $p[x]$ are both red. Each of the subtrees α, β, γ, and δ has a black root, and each has the same black-height. Case 2 is transformed into case 3 by a left rotation, which preserves property 4: all downward paths from a node to a leaf have the same number of blacks. Case 3 causes some color changes and a right rotation, which also preserve property 4. The **while** loop then terminates, because property 3 is satisfied: there are no longer two red nodes in a row.

Algorithm Analysis

- While loop only repeats if case 1 is executed, then the pointer x moves up the tree. The total number of times the while loop can be executed is $O(h)$.
- The height of a R-B tree on n nodes is $O(\lg n)$, so Tree-Insert takes $O(\lg n)$ time.

$\Rightarrow$ TOTAL: $O(\lg n)$

Review on Tree-Delete (T, x)

if x has no children ♦ case 0

then remove x

if x has one child ♦ case 1

then make child's parent be $p[x]$

if x has two children ♦ case 2

then swap x with its successor

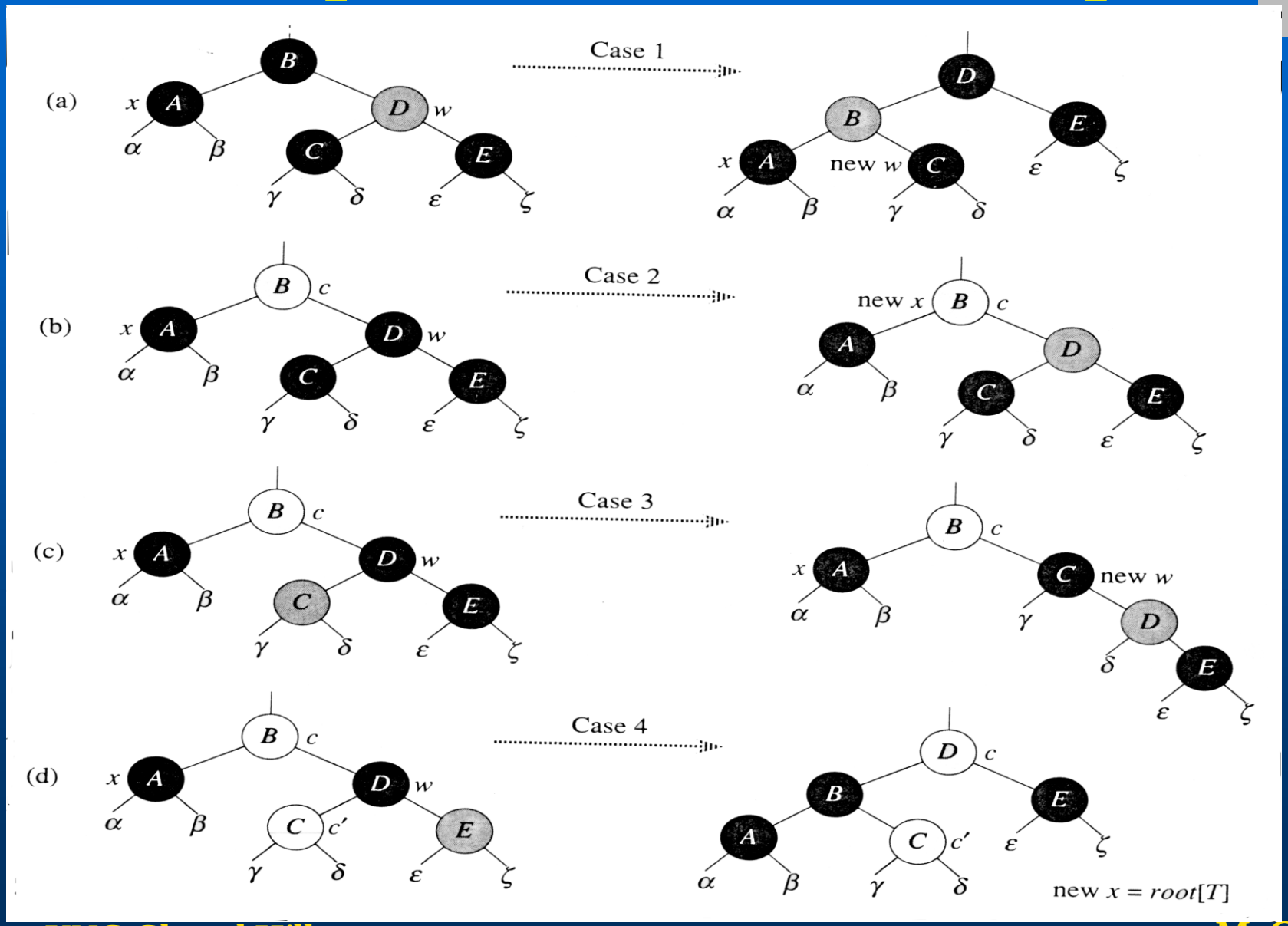
perform case 0 or case 1 to delete it

RB-Delete

- Implementation: use sentinels to treat a NIL child of a node x as an ordinary node whose parent is x. (Sentinel is a dummy object that helps simplify boundary conditions.)
- A minor modification of Tree-Delete: after splicing out a node, it calls RB-Delete-Fixup that changes colors and performs rotations to restore red-black properties.
  - Replace "NIL" by "nil[T]"
  - line 7: p[x]← p[y] is performed unconditionally
  - RB-Delete-Fixup(T, x)

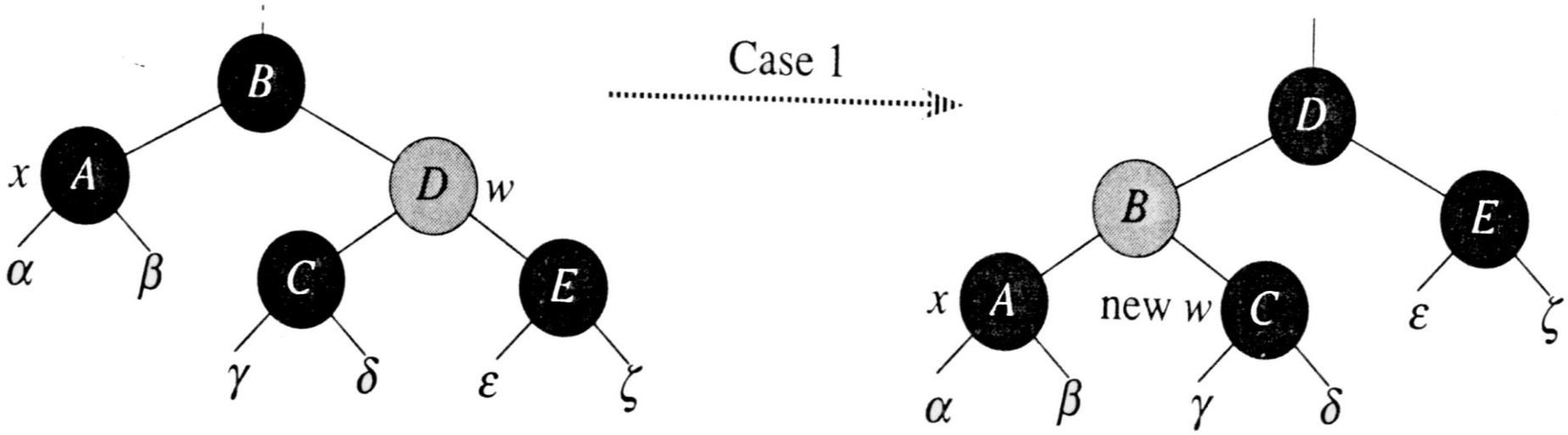
```
RB-DELETE(T, z)
 1  if left[z] = nil[T] or right[z] = nil[T]
 2      then y ← z
 3      else y ← TREE-SUCCESSOR(z)
 4  if left[y] ≠ nil[T]
 5      then x ← left[y]
 6      else x ← right[y]
 7  p[x] ← p[y]
 8  if p[y] = nil[T]
 9      then root[T] ← x
10      else if y = left[p[y]]
11              then left[p[y]] ← x
12              else right[p[y]] ← x
13  if y ≠ z
14      then key[z] ← key[y]
15           ▷ If y has other fields, copy them, too.
16  if color[y] = BLACK
17      then RB-DELETE-FIXUP(T, x)
18  return y
```

Example of RB-Delete-Fixup



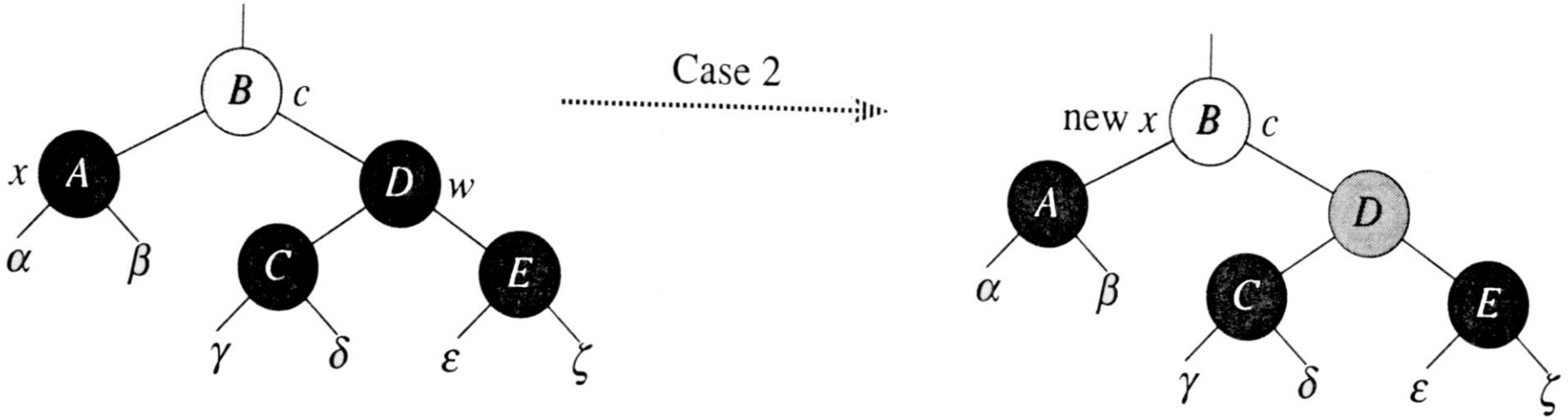
Case 1

- x's sibling w is red
  - Rotate toward x around x's parent
  - Now x's sibling is black (why?) and we are in Case 2, 3, or 4



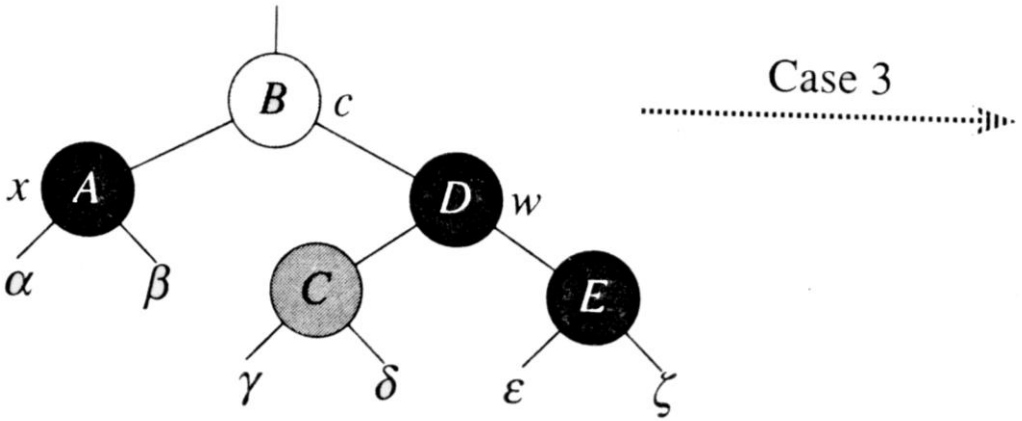
Case 2

- w is black, as are both w's children
  - Make w red, make x point to parent
  - Preserves consistency of black height, counting x
  - Moves problem closer to root
  - Terminates if came from Case 1

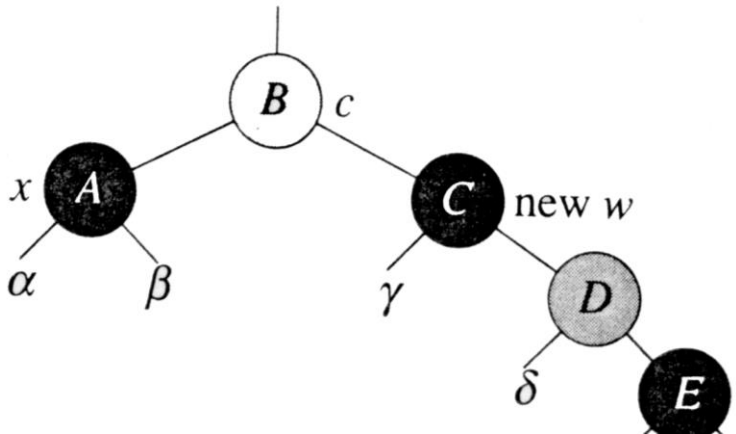


Case 3

- w is black, and its inside child (toward x) is red
  - Rotate outward around w
  - Make x's new sibling black, sibling's new right child red
  - Does this preserve black height consistency?
  - What would happen if we just made C black and E red?

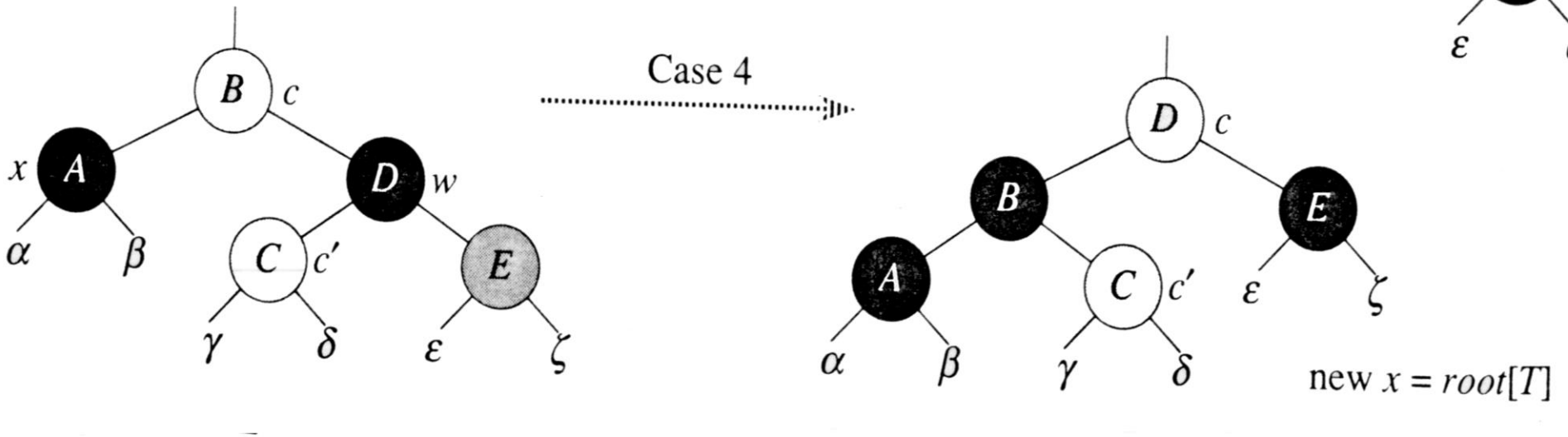


Case 3



Example of RB-Delete-Fixup

- w is black, and its outside child is red
  - Intuition: Not enough blacks on right side
  - Move E to root of right side, color it black--imbalance removed
  - Do this by rotating towards x, adjusting colors as shown



```
RB-DELETE-FIXUP(T, x)
 1  while x ≠ root[T] and color[x] = BLACK
 2        do if x = left[p[x]]
 3              then w ← right[p[x]]
 4                   if color[w] = RED
 5                      then color[w] ← BLACK                          ▷ Case 1
 6                           color[p[x]] ← RED                         ▷ Case 1
 7                           LEFT-ROTATE(T, p[x])                      ▷ Case 1
 8                           w ← right[p[x]]                           ▷ Case 1
 9                   if color[left[w]] = BLACK and color[right[w]] = BLACK
10                      then color[w] ← RED                            ▷ Case 2
11                           x ← p[x]                                  ▷ Case 2
12                      else if color[right[w]] = BLACK
13                              then color[left[w]] ← BLACK            ▷ Case 3
14                                   color[w] ← RED                    ▷ Case 3
15                                   RIGHT-ROTATE(T, w)                ▷ Case 3
16                                   w ← right[p[x]]                   ▷ Case 3
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 (same as then clause
                      with "right" and "left" exchanged)
23  color[x] ← BLACK
```

Analysis on RB-Delete

- Total procedure without RB-Delete-Fixup takes $O(\lg n)$ time
- RB-Delete-Fixup
  - cases 1, 3 & 4 takes constant time
  - case 2 takes at most $O(\lg n)$ time

$\Rightarrow$ RB-Delete takes $O(\lg n)$ time

$\Rightarrow$ All RB-tree operations take $O(\lg n)$ time