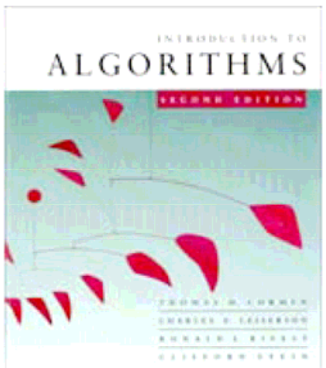
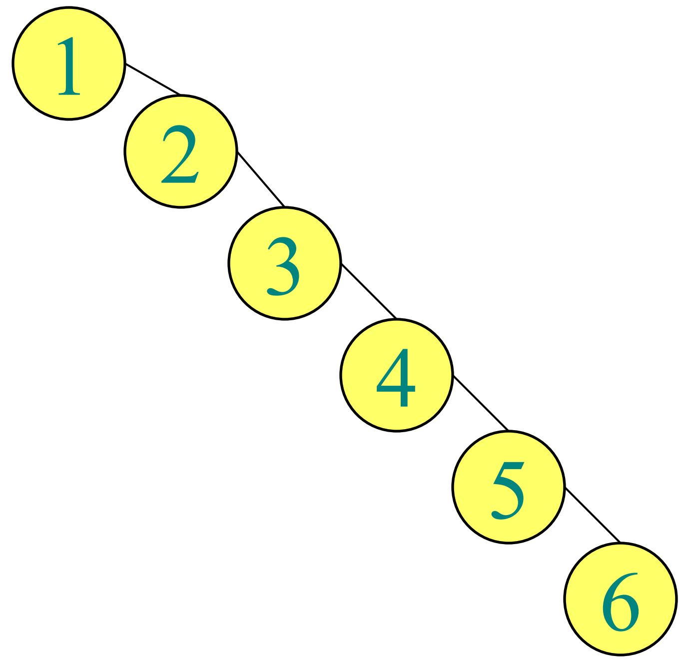
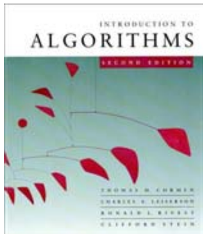



# Red Black Trees

# Today

- Balanced search trees,
  or how to avoid this
  even in the worst case





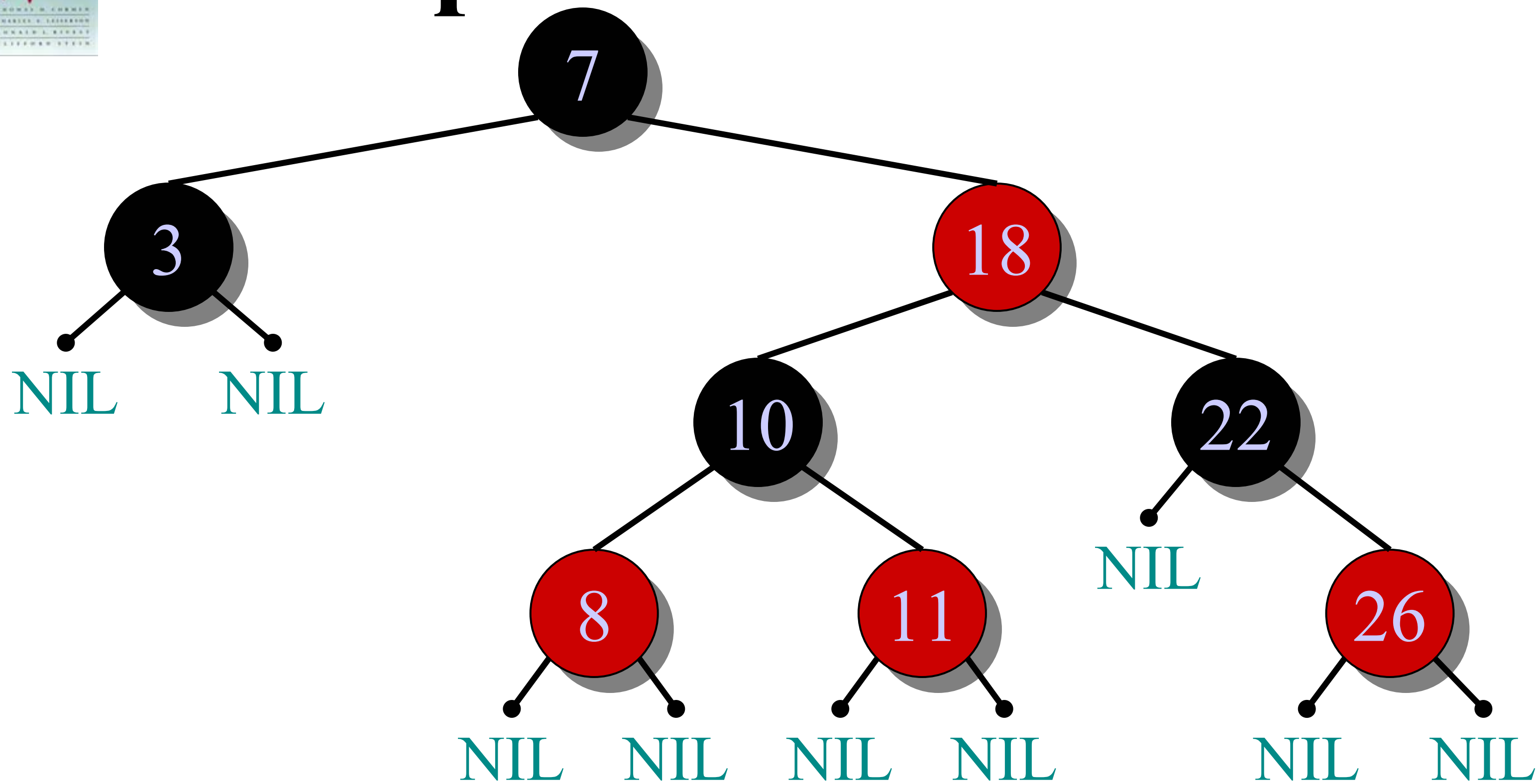

# Red-black trees

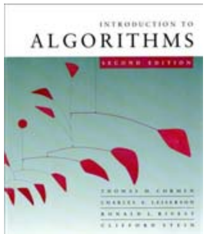
BSTs with an extra one-bit color field in each node.

***Red-black properties:***

1. Every node is either red or black.
2. The root and leaves (NIL's) are black.
3. If a node is red, then its parent is black.
4. All simple paths from any node $x$ to a descendant leaf have the same number of black nodes.

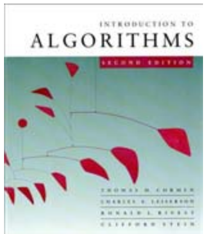
# Example of a red-black tree



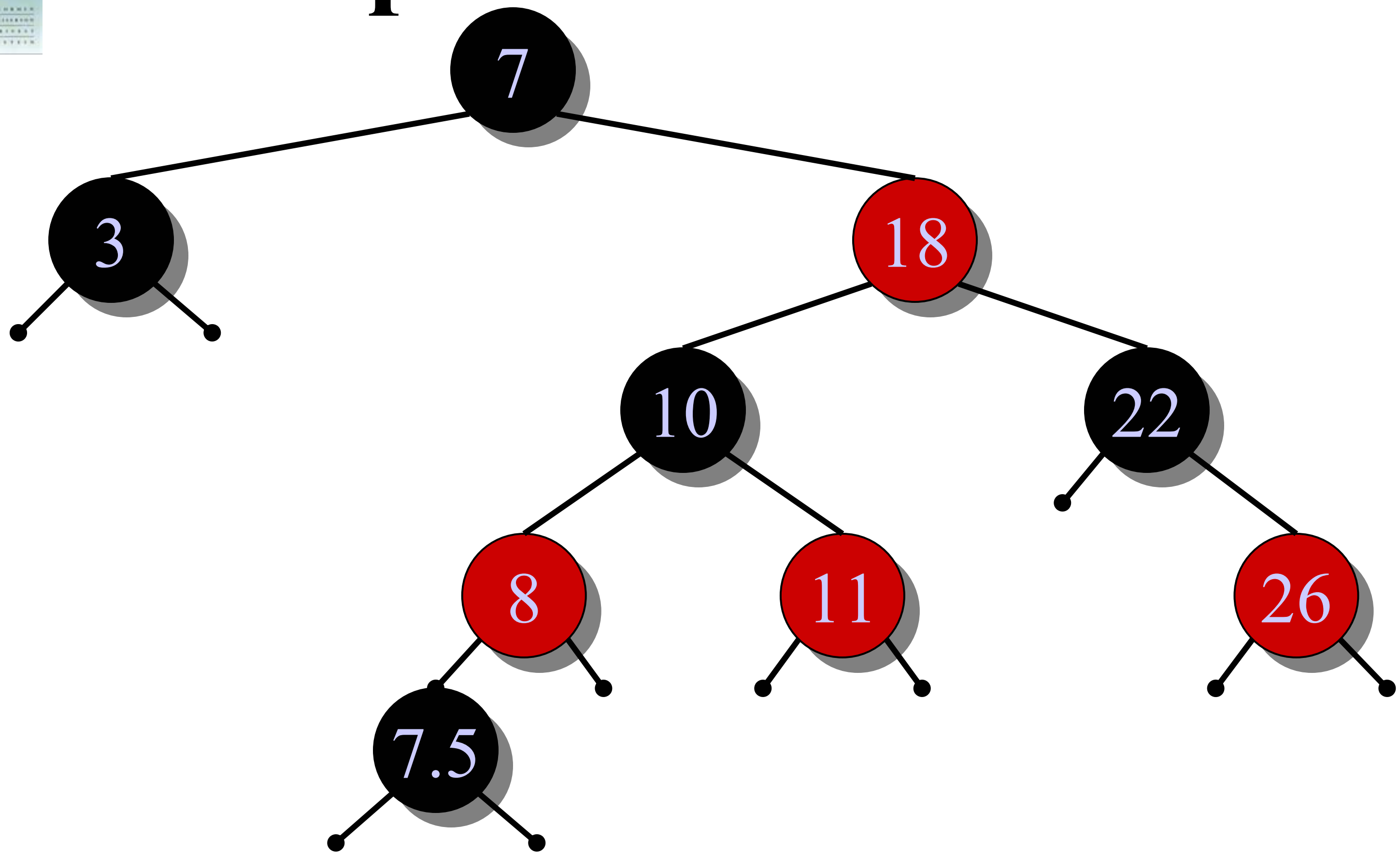



# Use of red-black trees

- What properties would we like to prove about red-black trees ?
  - They **always** have O(log n) height
  - There is an O(log n)–time insertion procedure which preserves the red-black properties
- Is it true that, after we add a new element to a tree (as in the previous lecture), we can always recolor the tree to keep it red-black ?



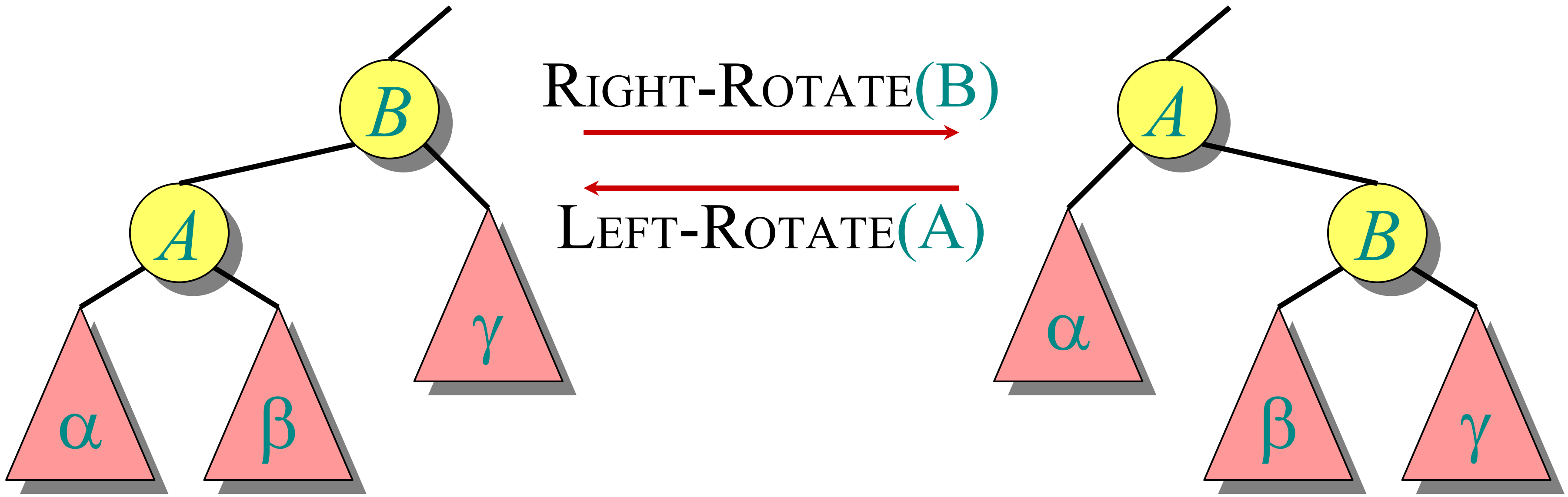
# Example of a red-black tree



# Use of red-black trees

- What properties would we like to prove about red-black trees ?
  - They **always** have O(log n) height
  - There is an O(log n)–time insertion procedure which preserves the red-black properties
- Is it true that, after we add a new element to a tree (as in the previous lecture), we can always recolor the tree to keep it red-black ?
- NO
- After insertions, sometimes we need to juggle nodes around

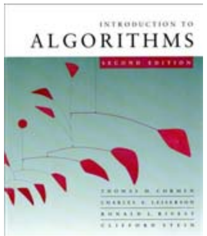
# Rotations



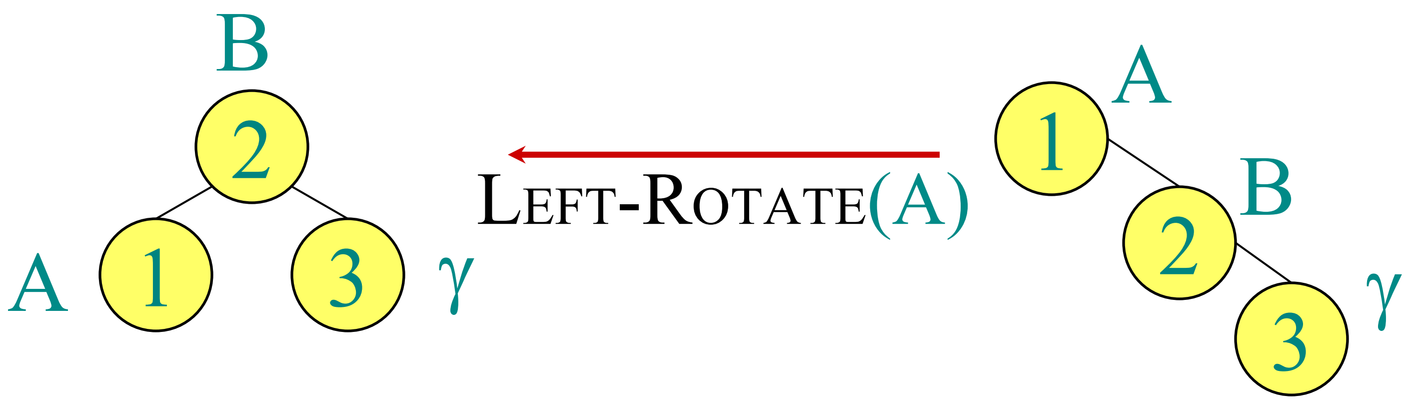

Rotations maintain the inorder ordering of keys:

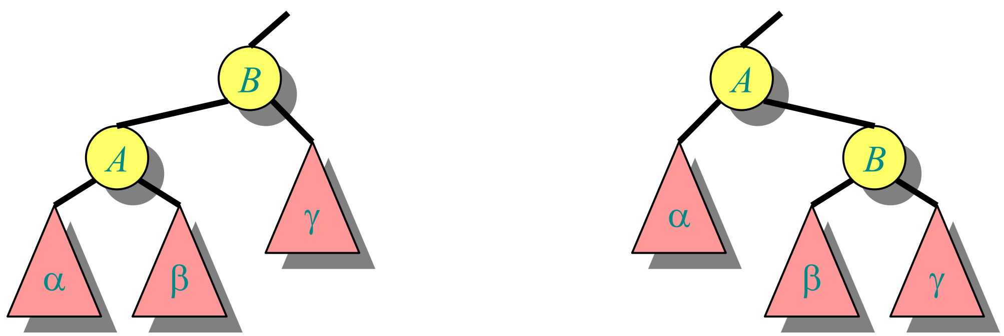
- $a \in \alpha$, $b \in \beta$, $c \in \gamma$ $\Rightarrow$ $a \le A \le b \le B \le c$.

A rotation can be performed in $O(1)$ time.



# Rotations can reduce height

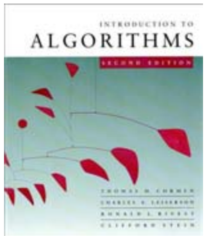


# Red-black tree wrap-up

- Can show how
  - O(log n) re-colorings
  - 1 rotation

  can restore red-black properties after an insertion
- Instead, we will see 2-3 trees (but will come back to red-black trees at the end)

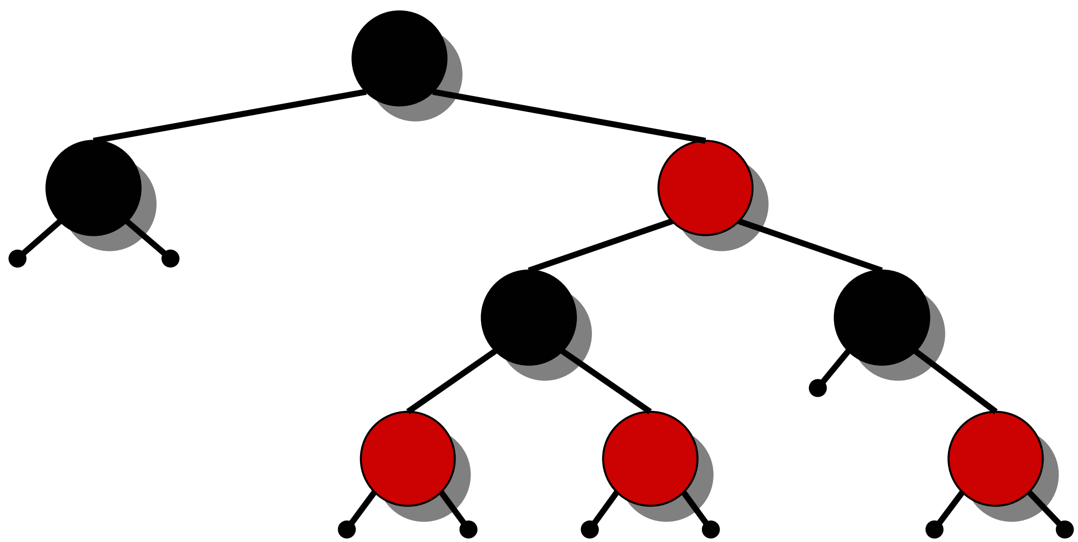


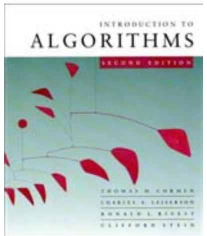

# Height of a red-black tree

**Theorem.** A red-black tree with $n$ keys has height
$$h \le 2 \lg(n + 1).$$

**INTUITION:**

- Merge red nodes into their black parents.





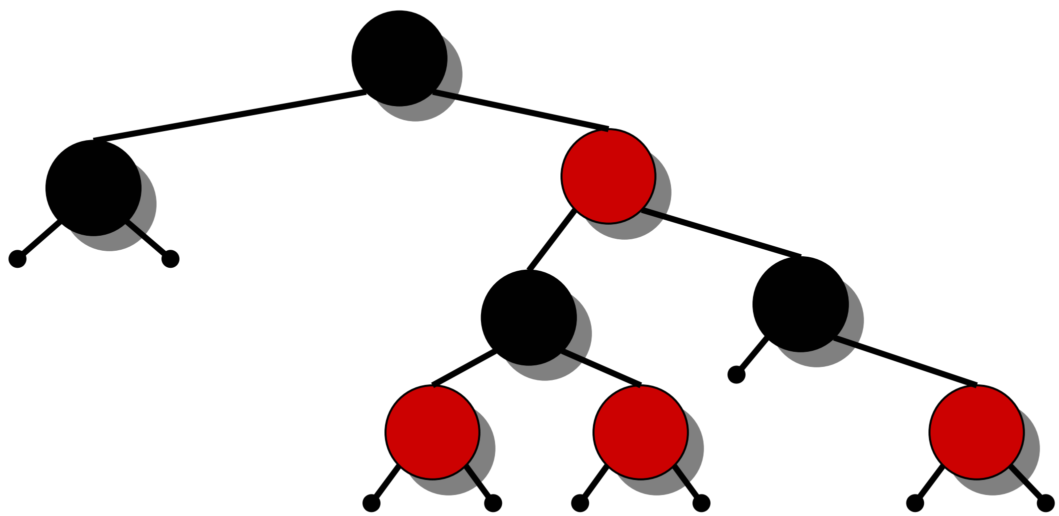
# Height of a red-black tree

**Theorem.** A red-black tree with $n$ keys has height

$$h \le 2\lg(n+1).$$

**INTUITION:**

- Merge red nodes into their black parents.



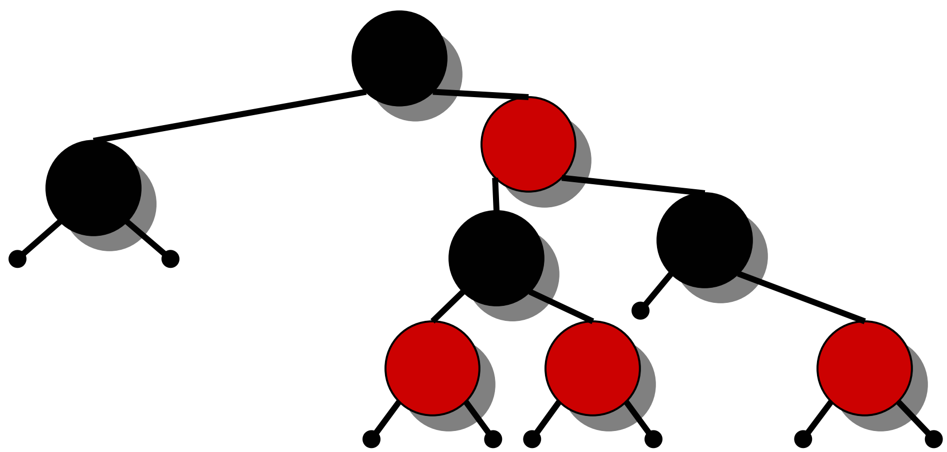
# Height of a red-black tree

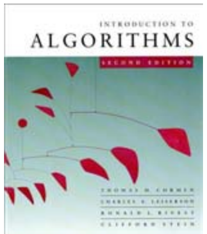
**Theorem.** A red-black tree with $n$ keys has height

$$h \le 2 \lg(n + 1).$$

**INTUITION:**

- Merge red nodes into their black parents.





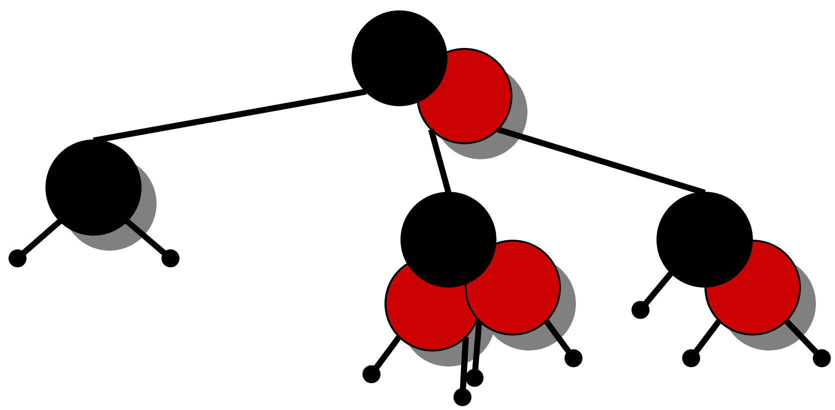

# Height of a red-black tree

**Theorem.** A red-black tree with $n$ keys has height

$$h \le 2 \lg(n + 1).$$

**INTUITION:**

- Merge red nodes into their black parents.



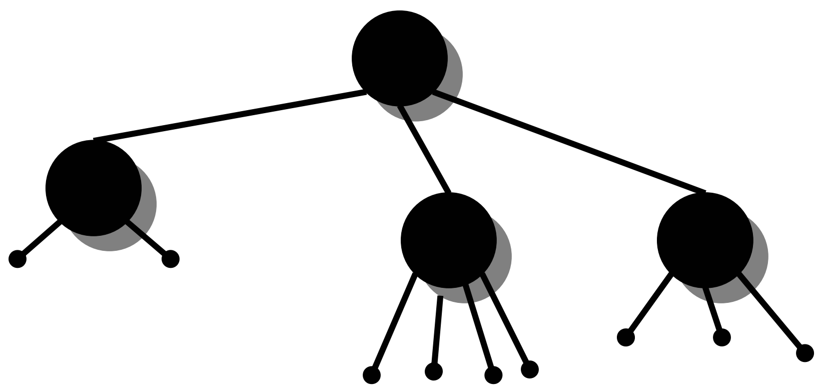
# Height of a red-black tree

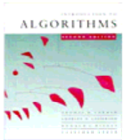
**Theorem.** A red-black tree with $n$ keys has height

$$h \leq 2 \lg(n + 1).$$

**INTUITION:**

- Merge red nodes into their black parents.






# Red-Black Trees

## Red-black trees

- A variation of binary search trees.
- ***Balanced***: height is $O(\lg n)$, where $n$ is the number of nodes.
- Operations will take $O(\lg n)$ time in the worst case.

# Red-Black Trees

A ***red-black tree*** is a binary search tree + 1 bit per node: an attribute *color*, which is either red or black.

All leaves are empty (nil) and colored black.

• We use a single sentinel, *nil*[*T* ], for all the leaves of red-black tree *T* .

• *color*[*nil*[*T* ]] is black.

• The root's parent is also *nil*[*T* ].

All other attributes of binary search trees are inherited by red-black trees (*key*, *left*, *right*, and *p*). We don't care about the key in *nil*[*T* ].

# Red-Black Trees

**Red-black properties**

1. Every node is either red or black.

2. The root is black.

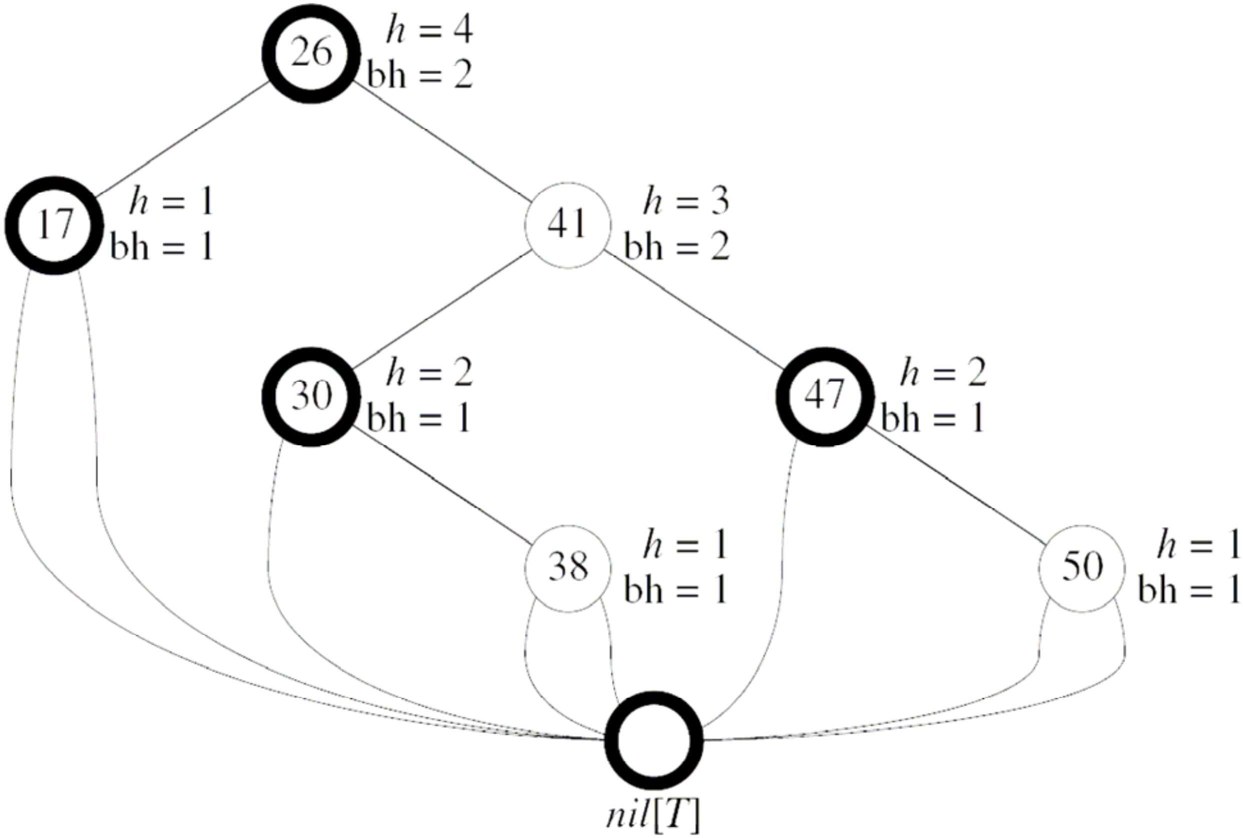
3. Every leaf (*nil*[*T* ]) is black.

4. If a node is red, then both its children are black. (Hence no two reds in a row on a simple path from the root to a leaf.)

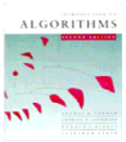
5. For each node, all paths from the node to descendant leaves contain the same number of black nodes.

# Red-Black Trees

Example:




*[Nodes with bold outline indicate black nodes. Don't add heights and black-heights yet. We won't bother with drawing nil[T] any more.]*

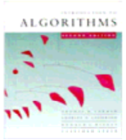



# Red-Black Trees

## Height of a red-black tree

• ***Height of a node*** is the number of edges in a longest path to a leaf.

• ***Black-height*** of a node *x*: bh*(x)* is the number of black nodes (including *nil*[*T* ]) on the path from *x* to leaf, not counting *x*. By property 5, black-height is well defined.




# Red-Black Trees

***Claim***

Any node with height $h$ has black-height $\geq h/2$.

***Proof*** By property 4, $\leq h/2$ nodes on the path from the node to a leaf are red. Hence $\geq h/2$ are black. (claim)

***Claim***

The sutree rooted at any node $x$ contains $\geq 2^{\text{bh}(x)} - 1$ internal nodes.

***Proof*** By induction on height of $x$.

**Basis**: Height of $x = 0 \Rightarrow x$ is a leaf $\Rightarrow \text{bh}(x) = 0$.

The sutree rooted at $x$ has 0 internal nodes. $2^0 - 1 = 0$.

**Inductive step**: Let the height of $x$ be $h$ and $\text{bh}(x) = b$. Any child of $x$ has height $h - 1$ and black-height either $b$ (if the child is red) or $b - 1$ (if the child is black). By the inductive hypothesis, each child has $\geq 2^{\text{bh}(x)-1} - 1$ internal nodes. Thus, the subtree rooted at $x$ contains $\geq 2 \cdot (2^{\text{bh}(x)-1} - 1) + 1 = 2^{\text{bh}(x)} - 1$ internal nodes. (The +1 is for $x$ itself.) (claim)

# Red-Black Trees

***Lemma***

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

***Proof*** Let $h$ and $b$ be the height and black-height of the root, respectively. By the above two claims,

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

Adding 1 to both sides and then taking logs gives $\lg(n+1) \geq h/2$, which implies that $h \leq 2\lg(n+1)$. (theorem)

# Red-Black Trees

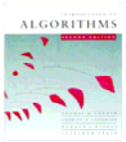
## Operations on red-black trees

The non-modifying binary-search-tree operations MINIMUM, MAXIMUM, SUCCESSOR, PREDECESSOR, and SEARCH run in $O(\text{height})$ time. Thus, they take $O(\lg n)$ time on red-black trees.

Insertion and deletion are not so easy.

If we **insert**, what color to make the new node?

- Red? Might violate property 4.
- Black? Might violate property 5.




# Red-Black Trees

If we **delete**, thus removing a node, what color was the node that was removed?

- Red? OK, since we won't have changed any black-heights, nor will we have created two red nodes in a row. Also, cannot cause a violation of property 2, since if the removed node was red, it could not have been the root.

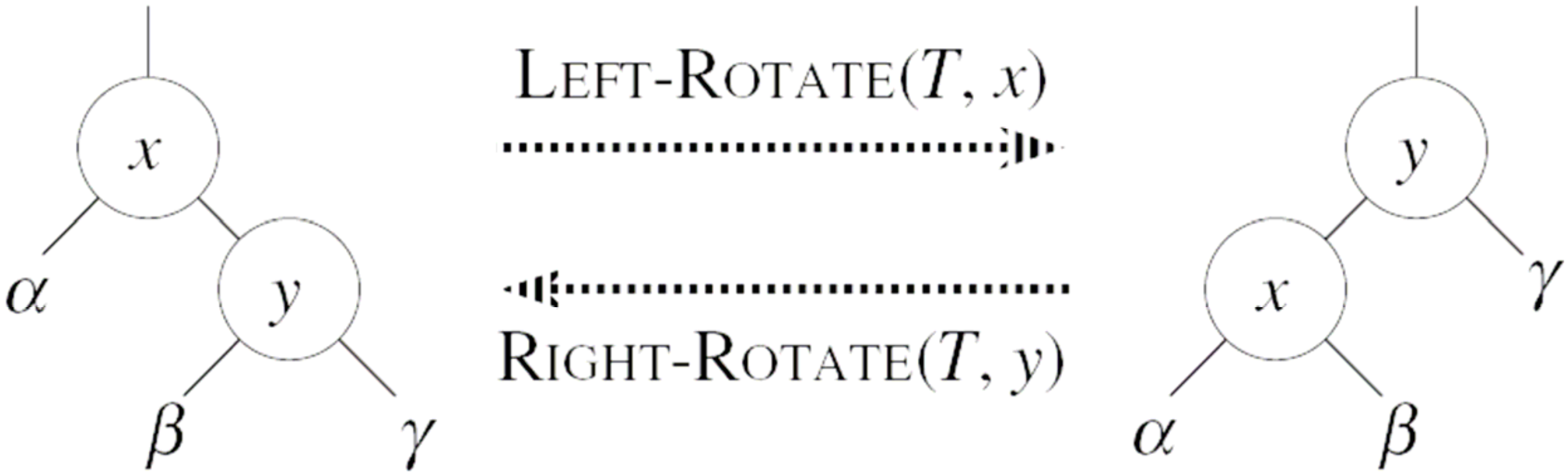
- Black? Could cause there to be two reds in a row (violating property 4), and can also cause a violation of property 5. Could also cause a violation of property 2, if the removed node was the root and its child—which becomes the new root—was red.

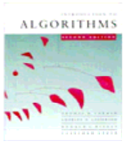
# Rotations

## Rotations

- The basic tree-restructuring operation.
- Needed to maintain red-black trees as balanced binary search  trees.
- Changes the local pointer structure. (Only pointers are changed.)
- Won’t upset the binary-search-tree property.
- Have both left rotation and right rotation.
  They are inverses of each other.
- A rotation takes a red-black-tree and a node within the tree.

# Rotations






# Rotations

LEFT-ROTATE(*T*, *x*)

$y \leftarrow right[x]$ ▷ Set *y*.

$right[x] \leftarrow left[y]$ ▷ Turn *y*'s left subtree into *x*'s right subtree.

**if** $left[y] \neq nil[T]$

  **then** $p[left[y]] \leftarrow x$

$p[y] \leftarrow p[x]$ ▷ Link *x*'s parent to *y*.

**if** $p[x] = nil[T]$

  **then** $root[T] \leftarrow y$

  **else if** $x = left[p[x]]$

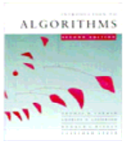
    **then** $left[p[x]] \leftarrow y$

    **else** $right[p[x]] \leftarrow y$

$left[y] \leftarrow x$ ▷ Put *x* on *y*'s left.

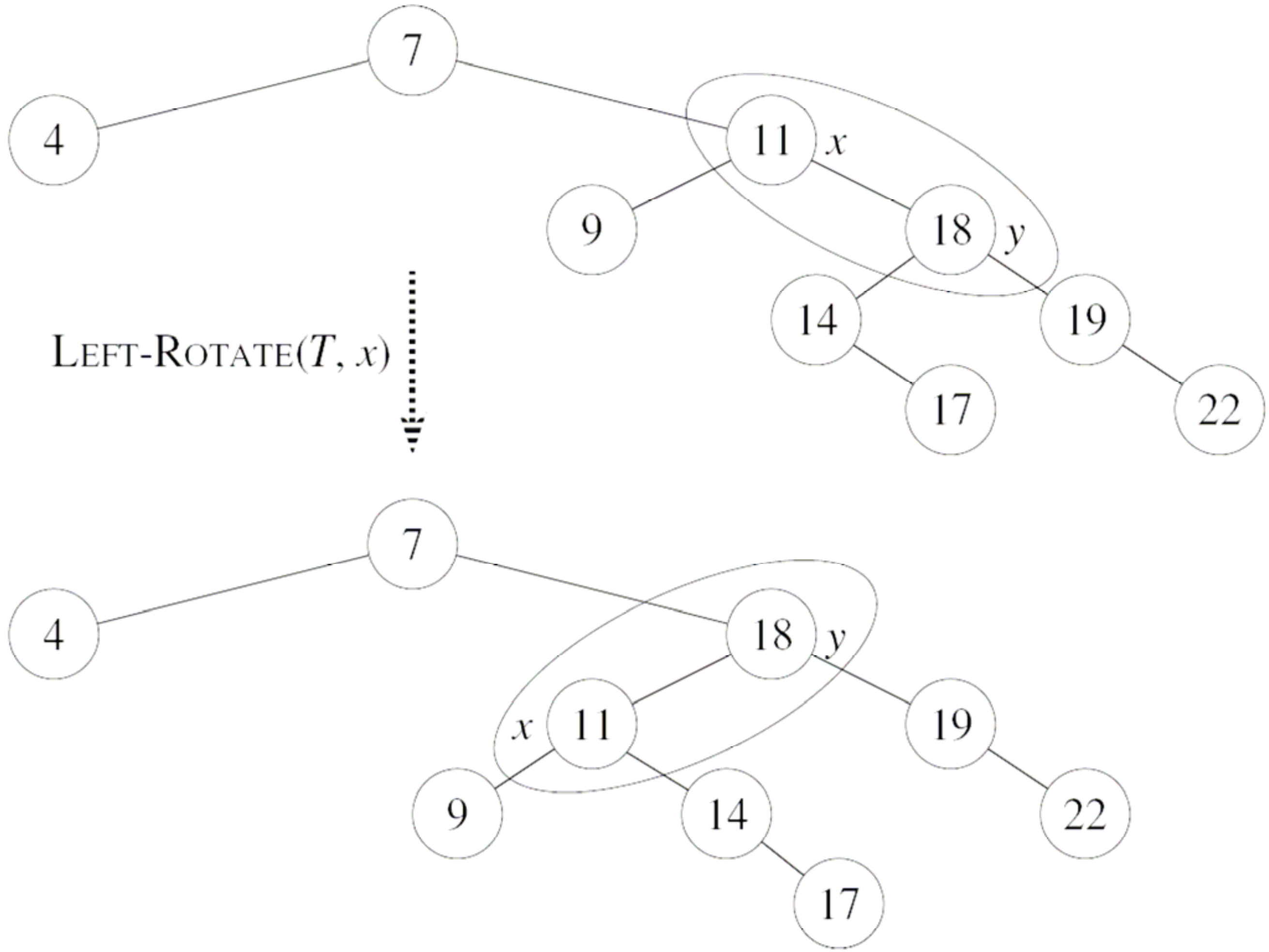
$p[x] \leftarrow y$

Pseudocode for RIGHT-ROTATE is symmetric: exchange *left* and *right* everywhere.

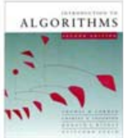


# Rotations

***Example:*** [Use to demonstrate that rotation maintains inorder ordering of keys. Node colors omitted.]




***Time:*** $O(1)$ for both LEFT-ROTATE and RIGHT-ROTATE, since a constant number of pointers are modified.




# Insertion

**Insertion** Start by doing regular binary-search-tree 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
left[z]← nil[T ]
right[z]← nil[T ]
color[z]← RED
RB-INSERT-FIXUP(T, z)
```

# Insertion

• RB-INSERT ends by coloring the new node *z* red.

• Then it calls RB-INSERT-FIXUP because we could have violated a red-black property.

Which property might be violated?

1. OK.

2. If *z* is the root, then there's a violation. Otherwise, OK.

3. OK.

4. If *p*[*z*] is red, there's a violation: both *z* and *p*[*z*] are red.

5. OK.

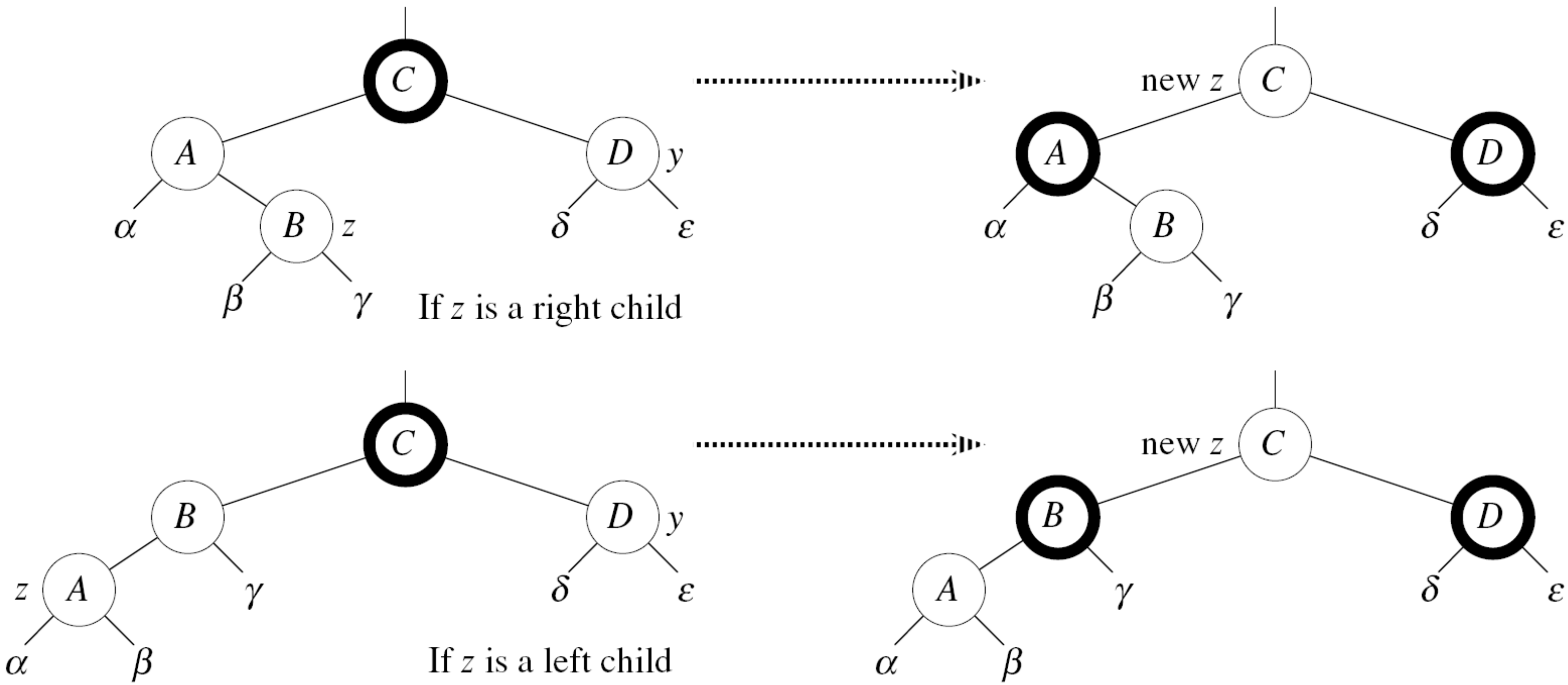
Remove the violation by calling RB-INSERT-FIXUP:

# Insertion

```
RB-INSERT-FIXUP(T, z)
while color[p[z]] = RED
    do if p[z] = left[p[p[z]]]
          then y ← right[p[p[z]]]
               if color[y] = RED
                 then color[p[z]] ← BLACK                ▷ Case 1
                      color[y] ← BLACK                   ▷ Case 1
                      color[p[p[z]]] ← RED               ▷ Case 1
                      z ← p[p[z]]                        ▷ Case 1
                 else if z = right[p[z]]
                        then z ← p[z]                    ▷ Case 2
                             LEFT-ROTATE(T, z)           ▷ Case 2
                      color[p[z]] ← BLACK                ▷ Case 3
                      color[p[p[z]]] ← RED               ▷ Case 3
                      RIGHT-ROTATE(T, p[p[z]])           ▷ Case 3
          else (same as then clause
                     with "right" and "left" exchanged)
color[root[T]] ← BLACK
```

# Insertion

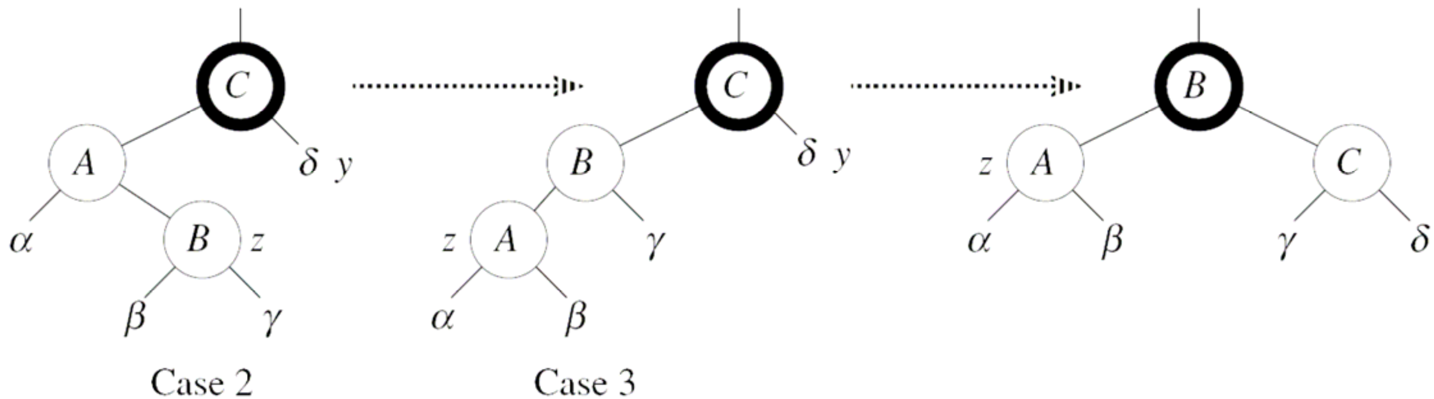
**Case 1:** $y$ is red




- $p[p[z]]$ ($z$'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).

# Insertion

**Case 2:** $y$ is black, $z$ is a right child




- Left rotate around $p[z] \Rightarrow$ 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 $p[p[z]]$.
- No longer have 2 reds in a row.
- $p[z]$ is now black $\Rightarrow$ no more iterations.

# Insertion

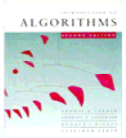
## Analysis

$O(\lg n)$ 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 is either the last one or it moves $z$ up 2 levels.
- $O(\lg n)$ levels $\Rightarrow$ $O(\lg n)$ time.
- Also note that there are at most 2 rotations overall.

Thus, insertion into a red-black tree takes $O(\lg n)$ time.




# Deletion

Start by doing regular binary-search-tree 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]
if p[y] = nil[T]
   then root[T] ← x
   else  if y = left[p[y]]
            then left[p[y]] ← x
            else  right[p[y]] ← x
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
```

- ***y*** **is the node that was actually spliced out.**
- ***x*** **is either**
  - ***y*'s sole non-sentinel child before *y* was spliced out, or**
  - **the sentinel, if *y* had no children.**

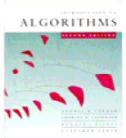
  **In both cases, *p*[*x*] is now the node that was previously *y*'s parent.**

# Deletion

If $y$ is black, we could have violations of red-black properties:

1. OK.
2. If $y$ is the root and $x$ is red, then the root has become red.
3. OK.
4. Violation if $p[y]$ and $x$ are both red.
5. Any path containing $y$ now has 1 fewer black node.
   - Correct by giving $x$ an "extra black."
   - Add 1 to 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.

Remove the violations by calling RB-DELETE-FIXUP:




# Deletion

```
RB-DELETE-FIXUP(T, x)
while x ≠ root[T] and color[x] = BLACK
    do if x = left[p[x]]
        then w ← right[p[x]]
            if color[w] = RED
              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
            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
                      then color[left[w]] ← BLACK            ▷ Case 3
                           color[w] ← RED                    ▷ Case 3
                           RIGHT-ROTATE(T, w)                ▷ Case 3
                           w ← right[p[x]]                   ▷ Case 3
                   color[w] ← color[p[x]]                    ▷ Case 4
                   color[p[x]] ← BLACK                       ▷ Case 4
                   color[right[w]] ← BLACK                   ▷ Case 4
                   LEFT-ROTATE(T, p[x])                      ▷ Case 4
                   x ← root[T]                               ▷ Case 4
        else (same as then clause with "right" and "left" exchanged)
color[x] ← BLACK
```

# Deletion

***Idea:*** Move the extra black up the tree until

- *x* points to a red & black node ⇒turn it into a black node,
- *x* points to the root ⇒just remove the extra black, or
- we can do certain rotations and recolorings and finish.

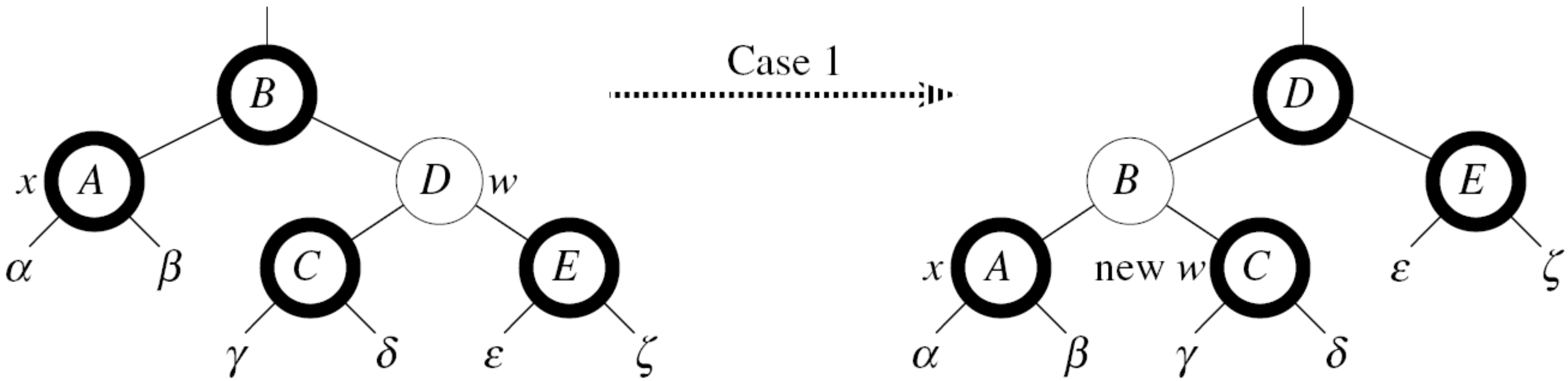
Within the **while** loop:

- *x* always points to a nonroot doubly black node.
- *w* is *x*'s sibling.
- *w* cannot be *nil*[*T* ], since that would violate property 5 at *p*[*x*].

There are 8 cases, 4 of which are symmetric to the other 4. As with insertion, the cases are not mutually exclusive. We'll look at cases in which *x* is a left child.

# Deletion

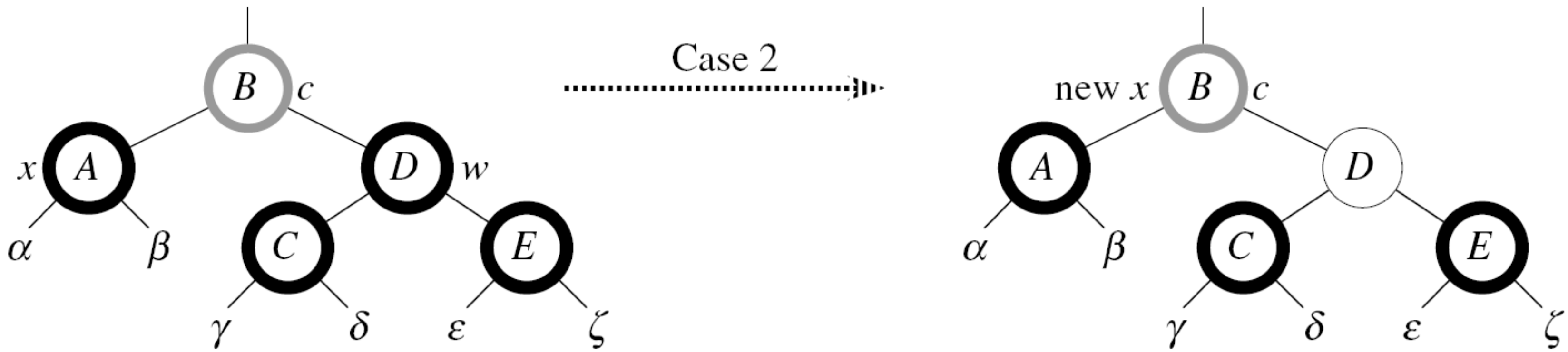
**Case 1:** $w$ is red




- $w$ must have black children.
- Make $w$ black and $p[x]$ red.
- Then left rotate on $p[x]$.
- New sibling of $x$ was a child of $w$ before rotation $\Rightarrow$ must be black.
- Go immediately to case 2, 3, or 4.

# Deletion

**Case 2:** $w$ is black and both of $w$'s children are black

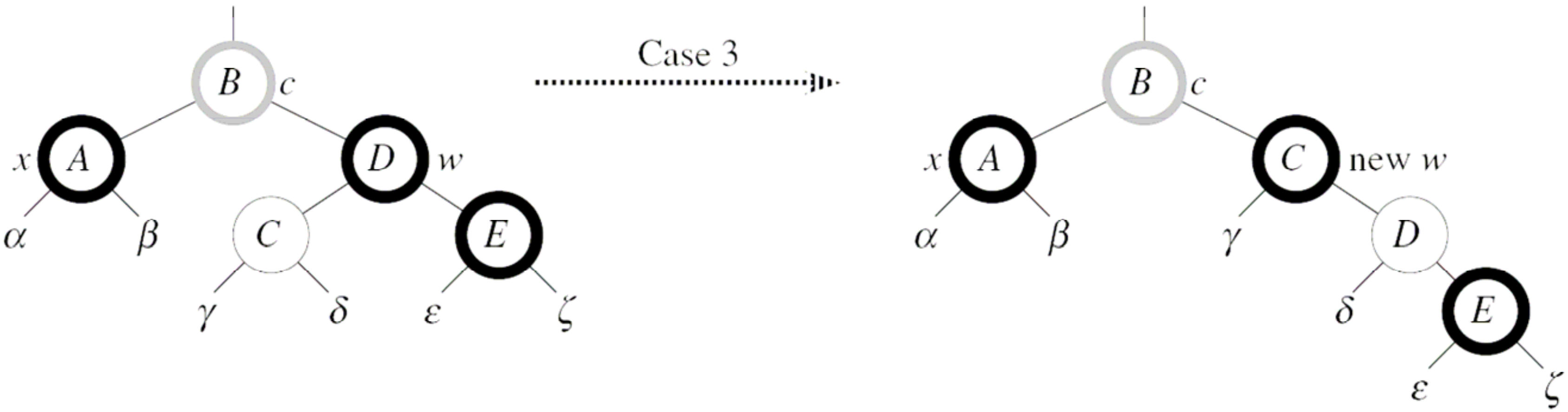



*[Node with gray outline is of unknown color, denoted by c.]*

- Take 1 black off $x$ ($\Rightarrow$ singly black) and off $w$ ($\Rightarrow$ 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 $\Rightarrow$ new $x$ is red & black $\Rightarrow$ color attribute of new $x$ is RED $\Rightarrow$ loop terminates. Then new $x$ is made black in the last line.

# Deletion

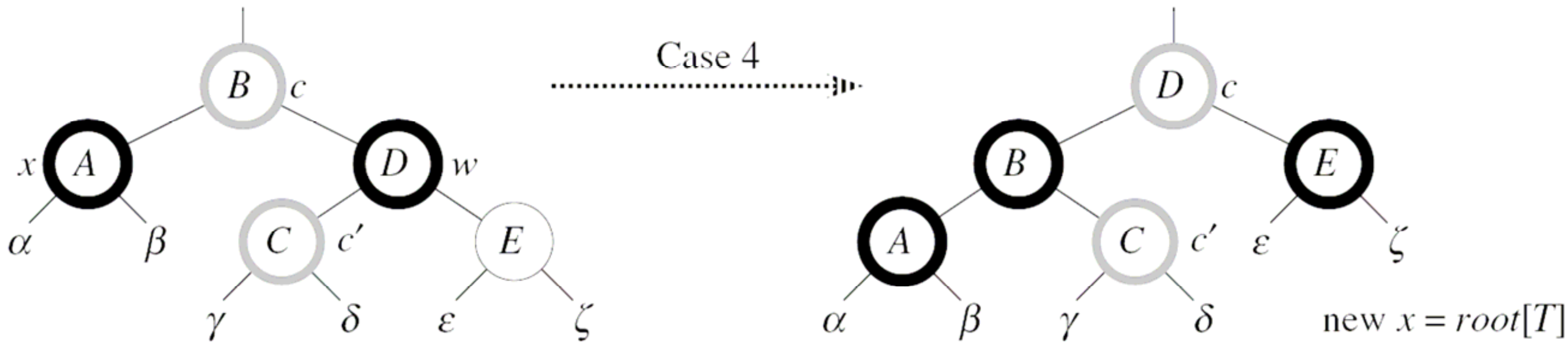
**Case 3:** $w$ is black, $w$'s left child is red, and $w$'s right child is black




- 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 $\Rightarrow$ case 4.

# Deletion

**Case 4:** $w$ is black, $w$'s left child is black, and $w$'s right child is red




*[Now there are two nodes of unknown colors, denoted by $c$ and $c'$.]*

- 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 $x$ ($\Rightarrow x$ is now singly black) without violating any red-black properties.
- All done. Setting $x$ to root causes the loop to terminate.

# Deletion

## 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.