Design and Analysis of Algorithms

CSE 5311

Lecture 11 Red-Black Trees

Junzhou Huang, Ph.D.

Department of Computer Science and Engineering

Reviewing: Binary Search Trees

- ***Binary Search Trees* (BSTs) are an important data structure for dynamic sets**
  - Each node has at most two children
- Each node contains:
  - key and data
  - left: points to the left child
  - right: points to the right child
  - p(parent): point to parent
- Binary-search-tree property:
  - y is a node in the left subtree of x: $y.key \leq x.key$
  - y is a node in the right subtree of x: $y.key \geq x.key$
  - Height: $\boldsymbol{h}$

Review: Inorder Tree Walk

- An *inorder walk* prints the set in sorted order:

  TreeWalk(x)

  TreeWalk(left[x]);

  print(x);

  TreeWalk(right[x]);

  - Easy to show by induction on the BST property
  - *Preorder tree walk*: print root, then left, then right
  - *Postorder tree walk*: print left, then right, then root

Review: BST Search

```
TreeSearch(x, k)
    if (x = NULL  or  k = key[x])
        return x;
    if (k < key[x])
        return TreeSearch(left[x], k);
    else
        return TreeSearch(right[x], k);
```

Review: Sorting With BSTs

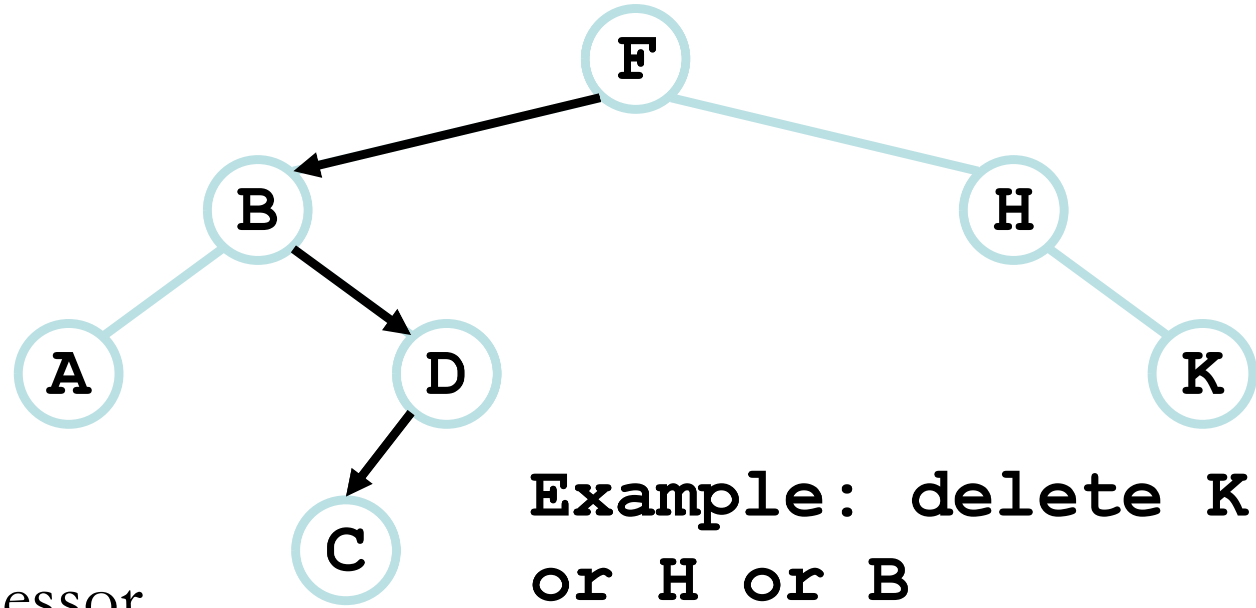
- **Basic algorithm:**
  - Insert elements of unsorted array from 1..*n*
  - Do an inorder tree walk to print in sorted order
- **Running time:**
  - Best case: $\Omega(n \lg n)$ (it's a comparison sort)
  - Worst case: $O(n^2)$
  - Average case: $O(n \lg n)$ (it's a quicksort!)

Review: More BST Operations

- **Minimum:**
  - Find leftmost node in tree
- **Successor:**
  - x has a right subtree: successor is minimum node in right subtree
  - x has no right subtree: successor is first ancestor of x whose left child is also ancestor of x
    - Intuition: As long as you move to the left up the tree, you're visiting smaller nodes.
- **Predecessor: similar to successor**

Review: More BST Operations

- **Delete:**
  - x has no children:
    - ➢Remove x
  - x has one child:
    - ➢Splice out x
  - x has two children:
    - ➢Swap x with successor
    - ➢Perform case 1 or 2 to delete it



Example: delete K or H or B

Red-Black Trees

- ***Red-black trees*:**
  - Binary search tree with an additional attribute for its nodes: color which can be **red** or **black**
  - "Balanced" binary search trees guarantee an O(lgn) running time
  - Constrains the way nodes can be colored on any path from the root to a leaf:

Ensures that no path is more than twice as long as any other path

$\Rightarrow$ the tree is balanced

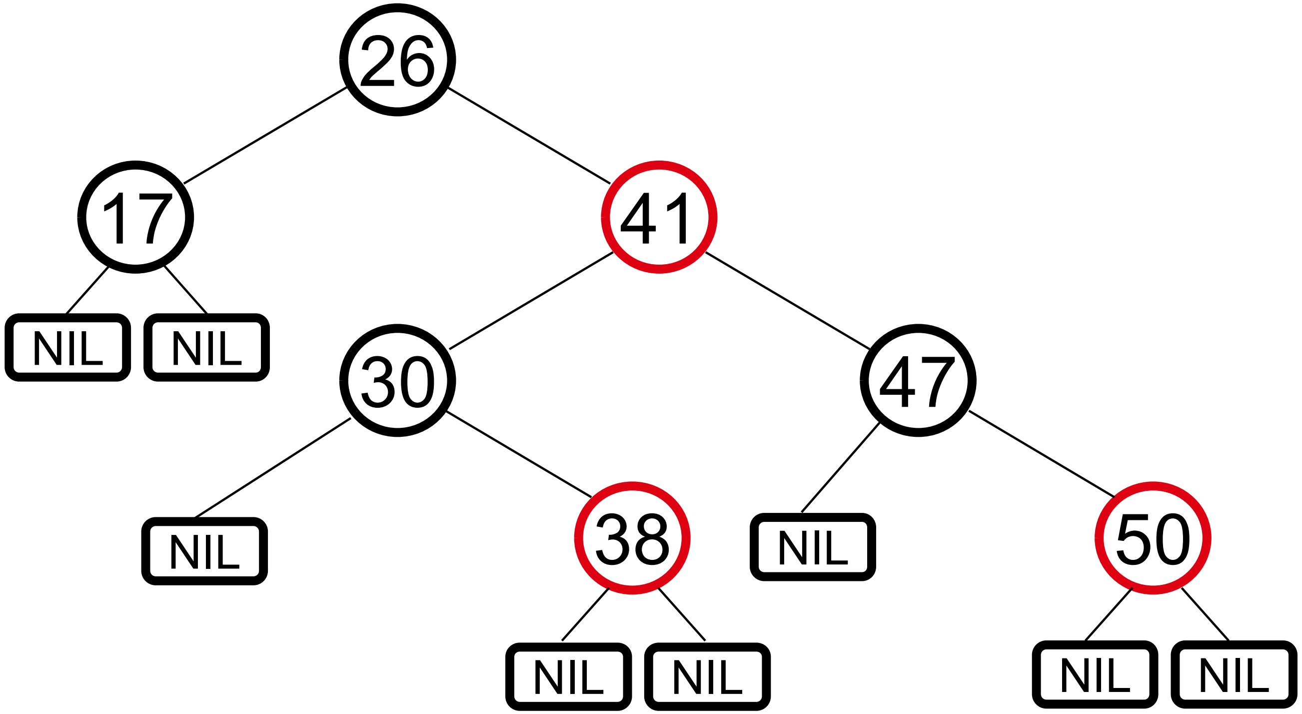
Red-Black Properties (**Satisfy the binary search tree property**)

- The ***red-black properties***:
  1. Every node is either red or black
  2. Every leaf (NULL pointer) is black
     - Note: this means every "real" node has 2 children
  3. If a node is red, both children are black
     - Note: can't have 2 consecutive reds on a path
  4. Every path from node to descendent leaf contains the same number of black nodes
  5. The root is always black

***black-height:* # black nodes on path to leaf**

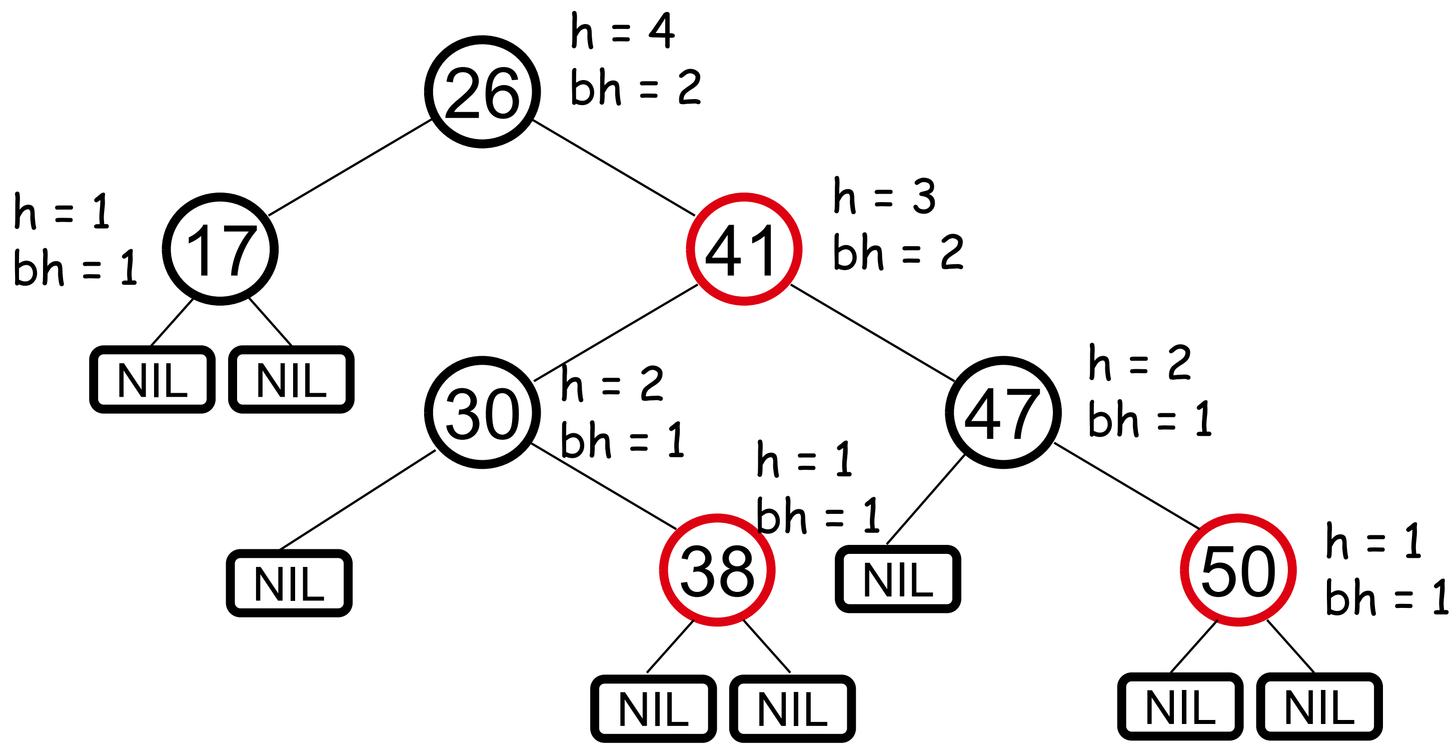
Label example with *h* and bh values

Example: RED-BLACK-TREE



- For convenience we use a sentinel NIL[T] to represent all the NIL nodes at the leafs
  - NIL[T] has the same fields as an ordinary node
  - Color[NIL[T]] = BLACK
  - The other fields may be set to arbitrary values

Black-Height of a Node



- **Height of a node:** the number of edges in the **longest** path to a leaf
- **Black-height** of a node x: bh(x) is the number of black nodes (including NIL) on the path from x to a leaf,

<u>not counting x</u>

Height of Red-Black Trees

- ***What is the minimum black-height of a node with height h?***
- A: a height-h node has black-height $\geq h/2$
- **Theorem:** A red-black tree with n internal nodes has height $h \leq 2 \lg(n + 1)$
- ***How do you suppose we'll prove this?***

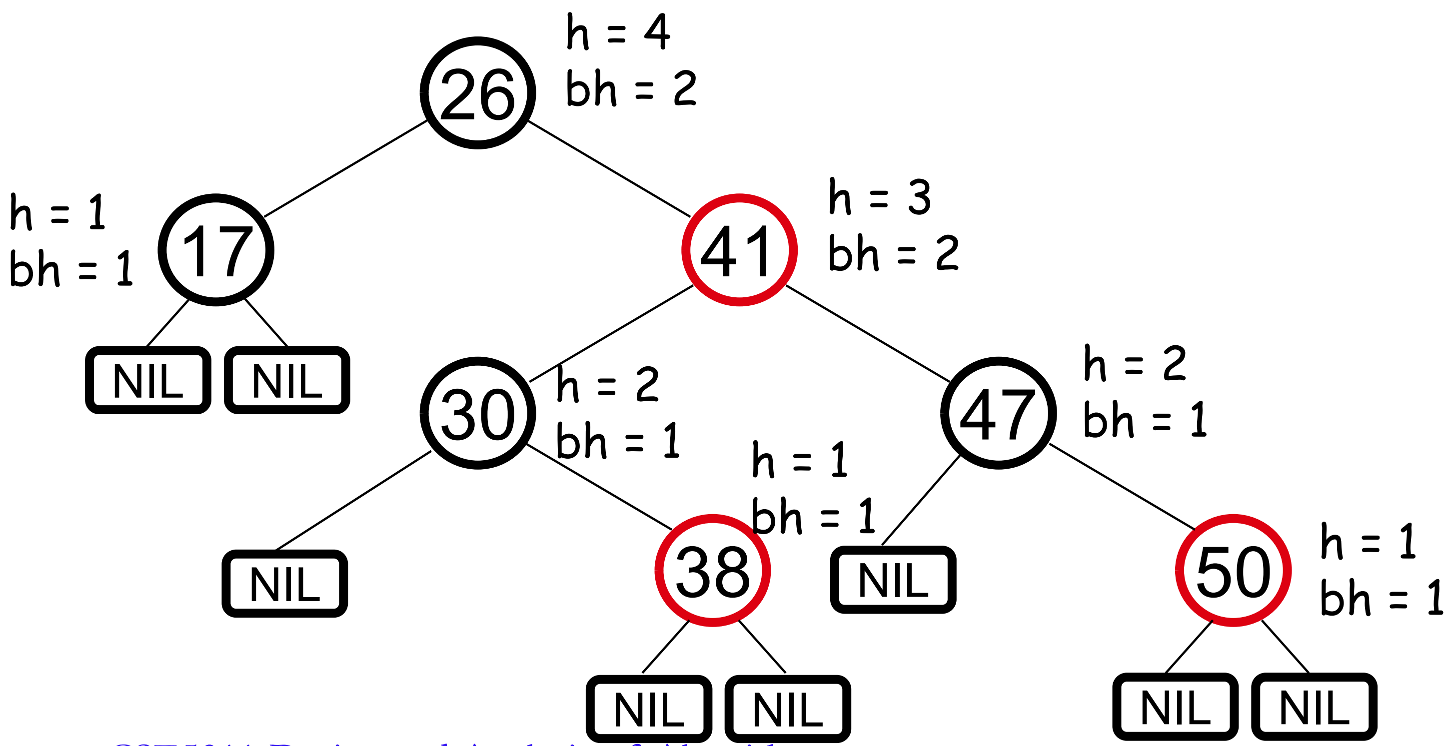
- ***Need to prove two claims first!!!***

Claim 1

4. If a node is **red**, then both its children are **black**

- **No two consecutive red nodes on a simple path from the root to a leaf**

- Any node x with height h(x) has bh(x) ≥ h(x)/2
- **Proof**
  - By property 4, at most h/2 **red** nodes on the path from the node to a leaf
  - Hence at least h/2 are **black**

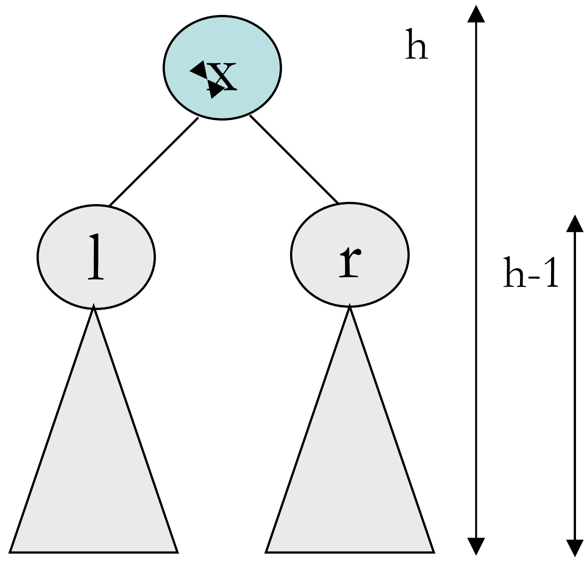


Claim 2

- **A subtree rooted at a node *x* contains at least $2^{bh(x)} - 1$ internal nodes**
- **Proof:**
  - Proof by induction on height *h*
  - Base step: *x* has height 0 (i.e., NULL leaf node)
    - ***What is bh(x)?***
    - A: 0
    - So…subtree contains $2^{bh(x)} - 1$
      $= 2^0 - 1$
      $= 0$ internal nodes (TRUE)

Claim 2: cont'd

- **Inductive proof that subtree at node *x* contains at least $2^{bh(x)}$ - 1 internal nodes**
  - Inductive step: *x* has positive height and 2 children
    - Each child has black-height of bh(*x*) (if the child is **red**) or bh(*x*)-1 (**(**if the child is **black**)
    - The height of a child = (height of *x*) - 1
    - So the subtrees rooted at each child contain at least $2^{bh(x) - 1}$ - 1 internal nodes
    - Thus subtree at *x* contains $(2^{bh(x) - 1} - 1) + (2^{bh(x) - 1} - 1) + 1$ $= 2 \bullet 2^{bh(x)-1} - 1 = 2^{bh(x)} - 1$ nodes



bh(l)≥bh(x)-1

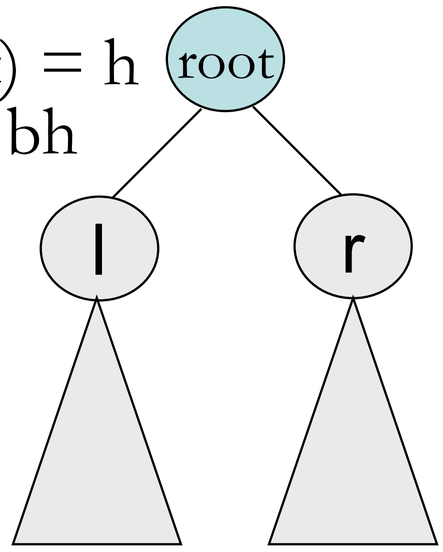
bh(r)≥bh(x)-1

Height of Red-Black-Trees

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

Proof:

height(root) = h
bh(root) = bh



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

number **n** of internal nodes

since bh ≥ h/2

- Add 1 to both sides and then take logs:

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

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

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

RB Trees: Worst-Case Time

- **So we've proved that a red-black tree has O(lg *n*) height**
- **Corollary: These operations take O(lg *n*) time:**
  - Minimum(), Maximum()
  - Successor(), Predecessor()
  - Search()
- **Insert() and Delete():**
  - Will also take O(lg *n*) time
  - But will need special care since they modify tree
  - We have to guarantee that the modified tree will still be a red-black tree

Red-Black Tree

- **Recall binary search tree**
  - Key values in the left subtree <= the node value
  - Key values in the right subtree >= the node value
- **Operations:**
  - insertion, deletion
  - Search, maximum, minimum, successor, predecessor.
  - O(h), h is the height of the tree.

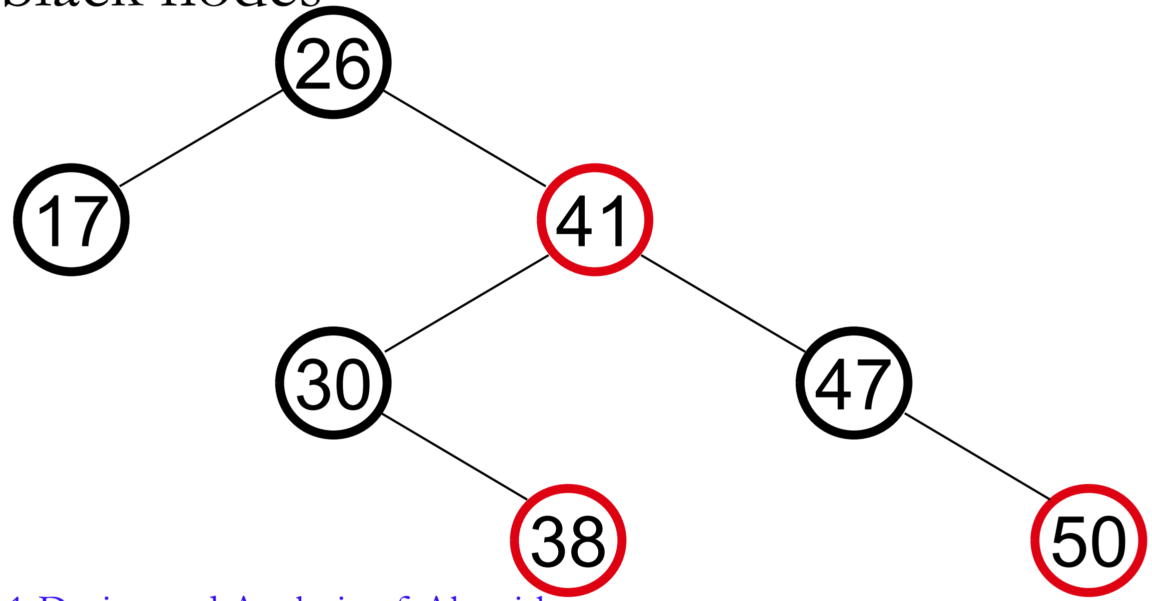
Red-black trees

- **Definition: a binary tree, satisfying:**
  1. Every node is red or black
  2. The root is black
  3. Every leaf is NIL and 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.
- **Purpose: keep the tree balanced.**
- **Other balanced search tree:**
  - AVL tree, 2-3-4 tree, Splay tree, Treap

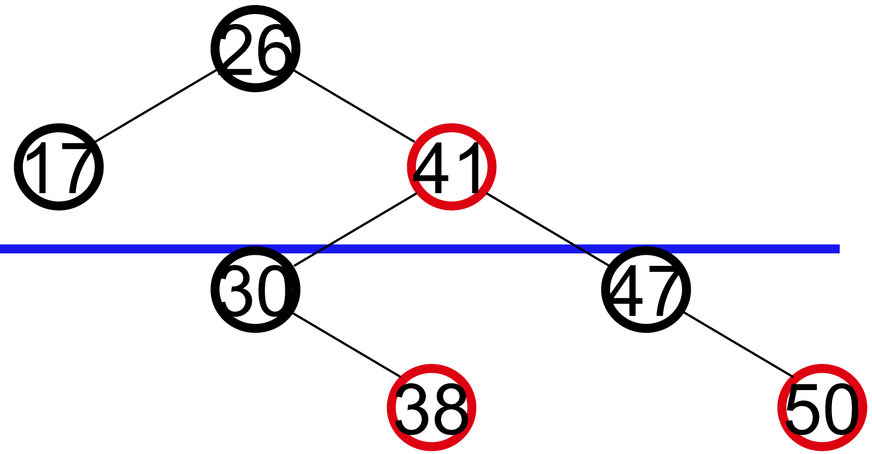
INSERT

INSERT: what color to make the new node?

- Red? Let's insert 35!
  - Property 4 is violated: if a node is red, then both its children are black
- Black? Let's insert 14!
  - Property 5 is violated: all paths from a node to its leaves contain the same number of black nodes



DELETE



DELETE: what color was the node that was removed? **Black?**

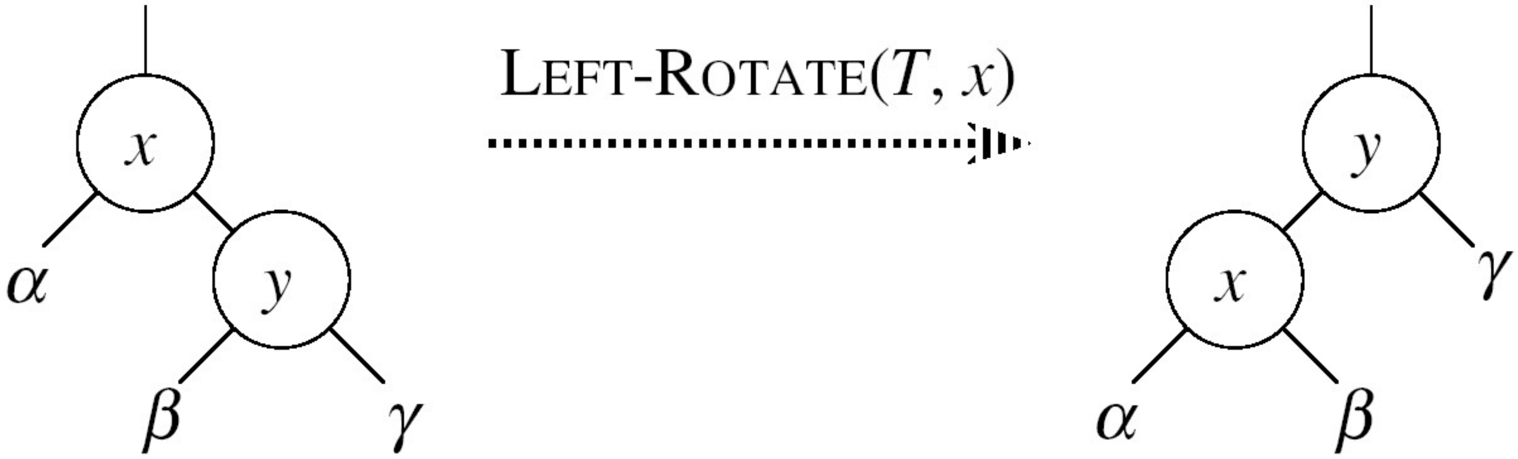
1. Every node is either **red** or **black** — OK!
2. The root is **black** — Not OK! If removing the root and the child that replaces it is **red**
3. Every leaf (NIL) is **black** — OK!
4. If a node is red, then both its children are black — Not OK! Could create two red nodes in a row
5. For each node, all paths from the node to descendant leaves contain the same number of black nodes — Not OK! Could change the black heights of some nodes

Rotations

- Operations for re-structuring the tree after insert and delete operations on red-black trees
- **Rotations take a red-black-tree and a node within the tree and:**
  - Together with some node re-coloring they help restore the red-black-tree property
  - Change some of the pointer structure
  - **Do not** change the binary-search tree property
- **Two types of rotations:**
  - Left & right rotations

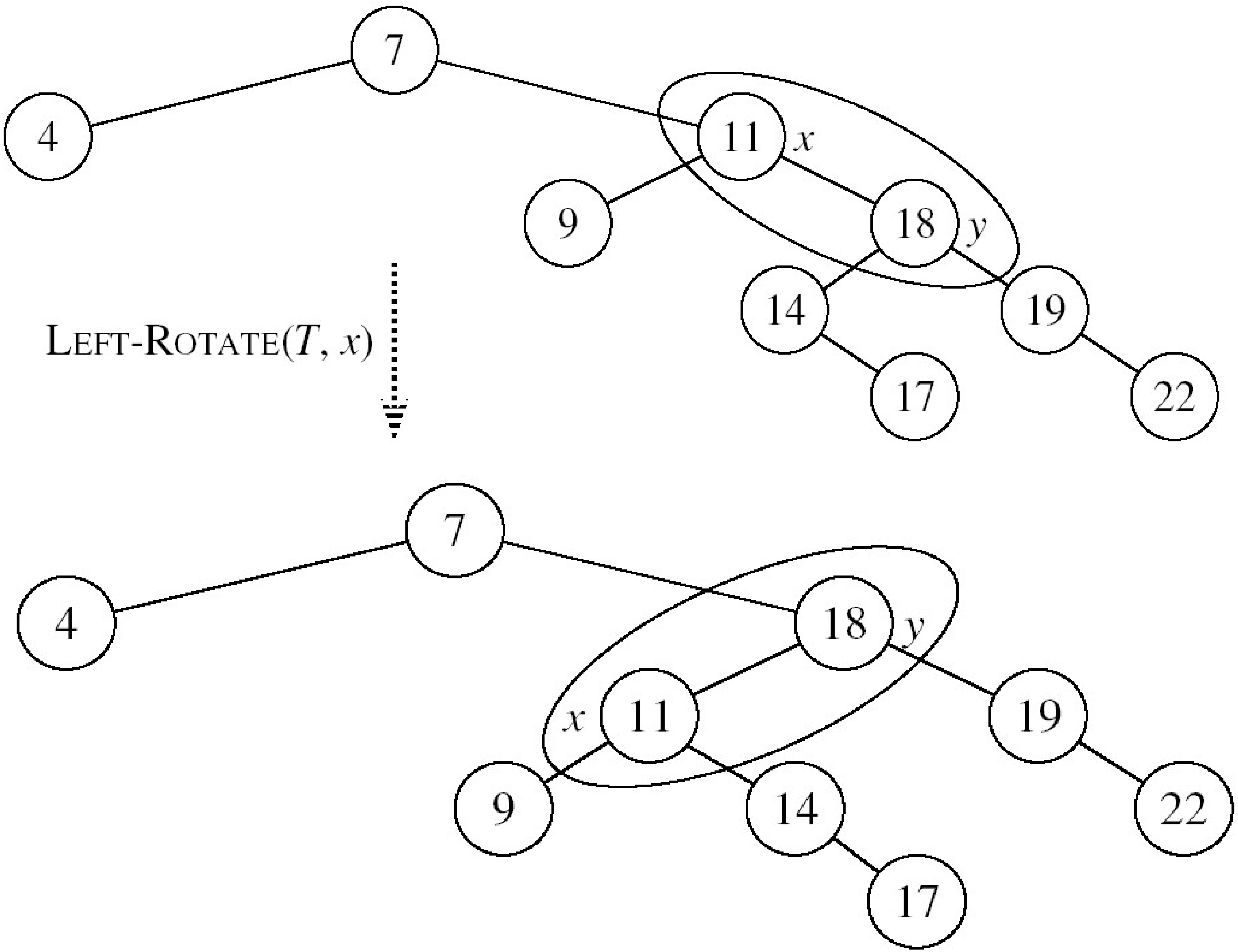
Left Rotations

- **Assumptions for a left rotation on a node x:**
  - The right child of x (y) is not NIL



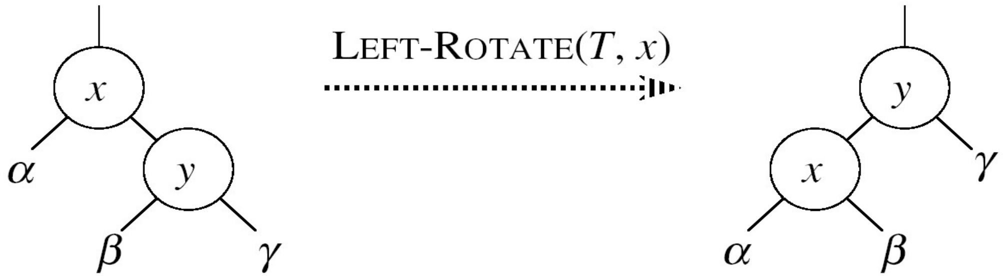
- **Idea:**
  - Pivots around the link from x to y
  - Makes y the new root of the subtree
  - x becomes y's left child
  - y's left child becomes x's right child

Example: LEFT-ROTATE



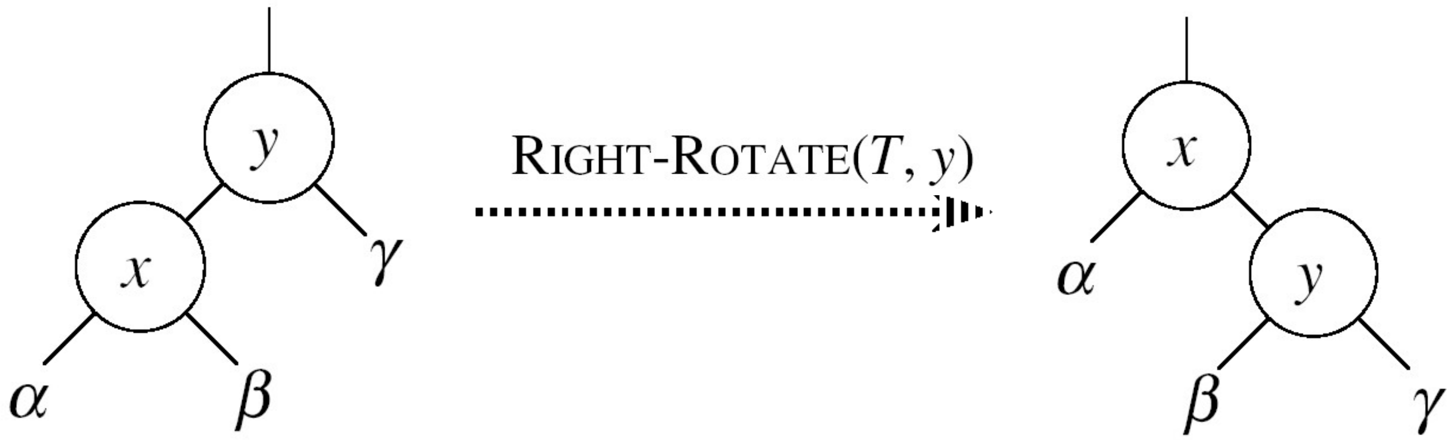
LEFT-ROTATE(T, x)

```
y ← right[x]                          ▸ Set y
right[x] ← left[y]                    ▸ y's left subtree becomes x's right subtree
if left[y] ≠ NIL
    then p[left[y]] ← x               ▸ Set the parent relation from left[y] to x
p[y] ← p[x]                           ▸ The parent of x becomes the parent of y
if p[x] = NIL
    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 on y's left
p[x] ← y                              ▸ y becomes x's parent
```



Right Rotations

- **Assumptions for a right rotation on a node x:**
  - The left child of y (x) is not NIL



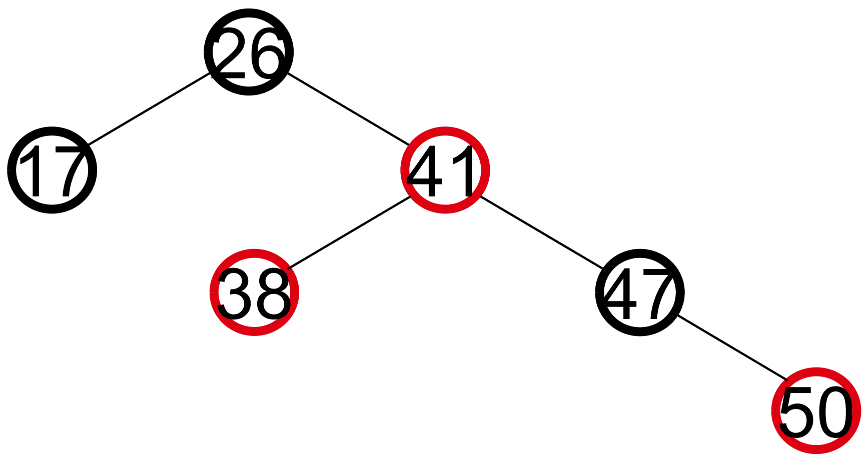
- **Idea.**
  - Pivots around the link from y to x
  - Makes x the new root of the subtree
  - y becomes x's right child
  - x's right child becomes y's left child

Insertion

- **Goal:**
  - Insert a new node z into a red-black-tree
- Idea:
  - Insert node z into the tree as for an ordinary binary search tree
  - Color the node **red**
  - Restore the red-black-tree properties
    - ➢Use an auxiliary procedure RB-INSERT-FIXUP

RB Properties Affected by Insert

1. Every node is either **red** or **black** — OK!
2. The root is **black** — If z is the root ⇒ not OK
3. Every leaf (NIL) is **black** — OK!
4. If a node is red, then both its children are black — If p(z) is red ⇒ not OK: z and p(z) are both red
5. For each node, all paths from the node to descendant leaves contain the same number of black nodes — OK!



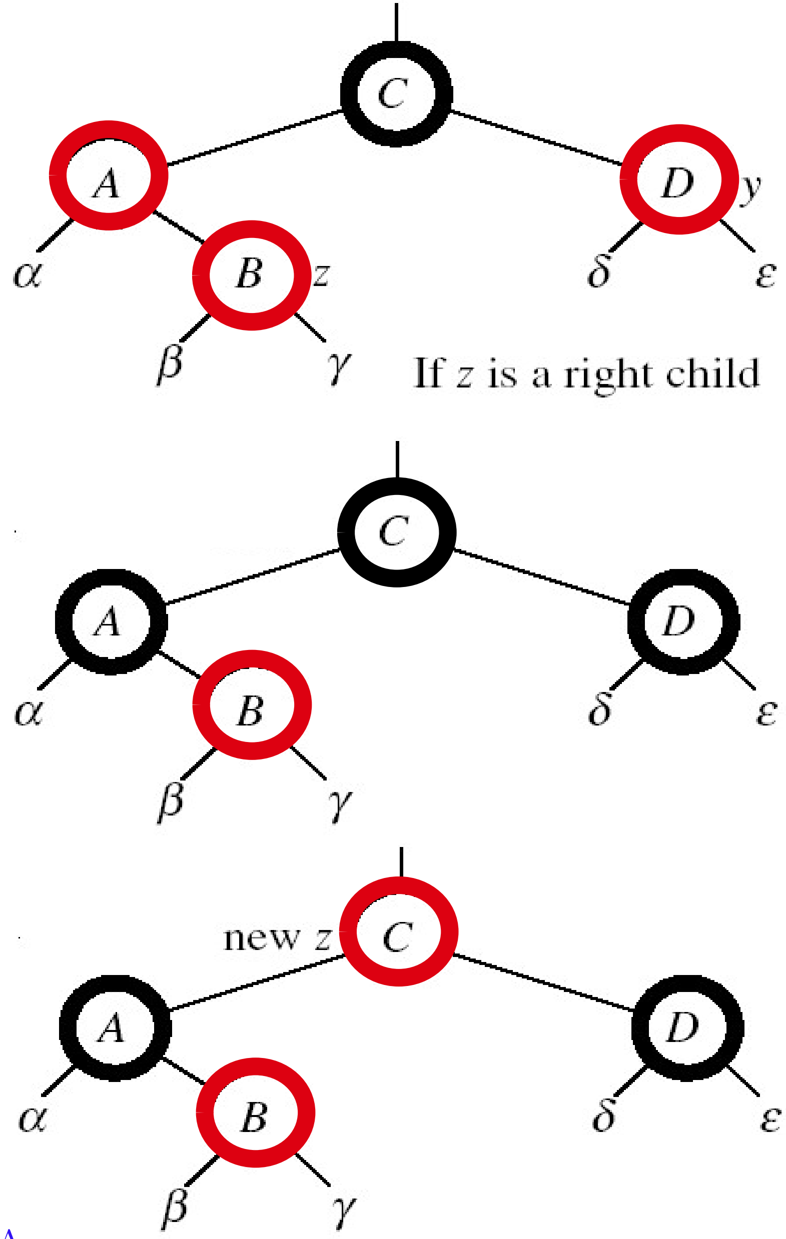
RB-INSERT-FIXUP – Case 1

z's "uncle" (y) is **red**

Idea: (z is a right child)

- p[p[z]] (z's grandparent) must be black: z and p[z] are both red

- Color p[z] **black**
- Color y **black**
- Color p[p[z]] **red**
- z = p[p[z]]
  - Push the **"red"** violation up the tree



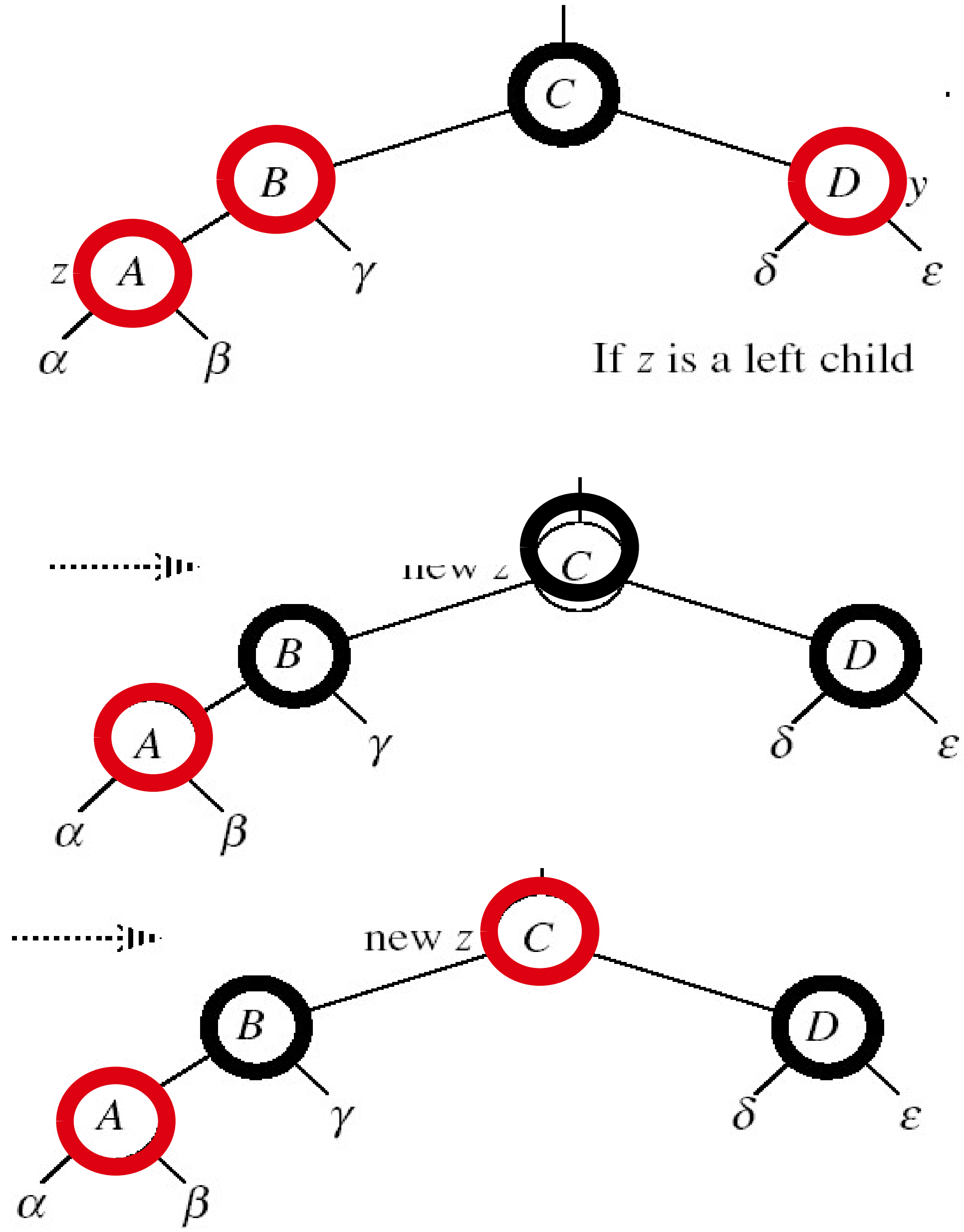
RB-INSERT-FIXUP – Case 1

z's "uncle" (y) is **red**

Idea: (z is a left child)

- p[p[z]] (z's grandparent) must be black: z and p[z] are both red

- color p[z] ← **black**
- color y ← **black**
- color p[p[z]] ← **red**
- z = p[p[z]]
  - Push the **"red"** violation up the tree



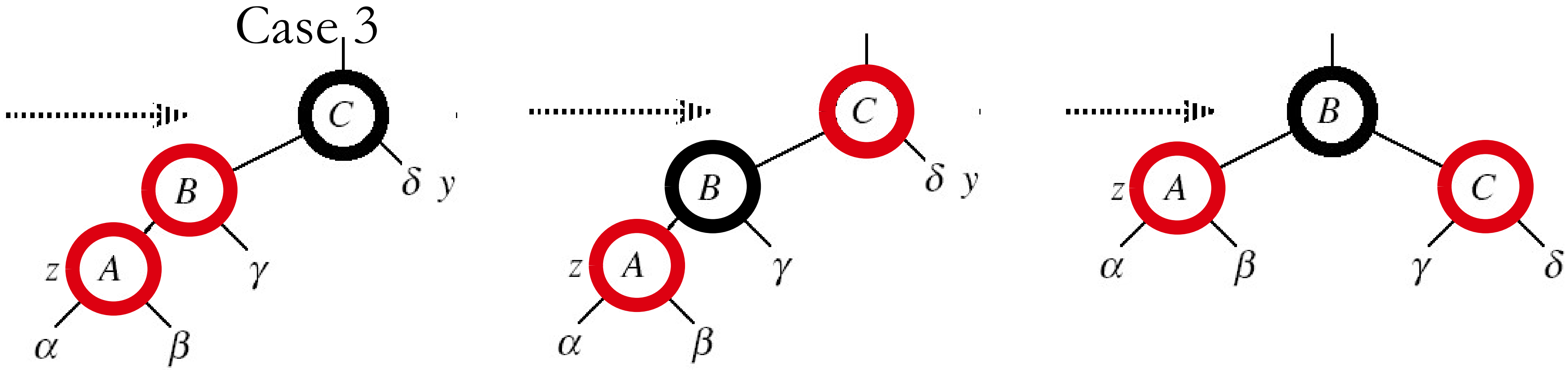
RB-INSERT-FIXUP – Case 3

Case 3:

- z's "uncle" (y) is **black**
- z is a left child

Idea:

- color p[z] ← **black**
- color p[p[z]] ← **red**
- RIGHT-ROTATE(T, p[p[z]])
- No longer have 2 reds in a row
- p[z] is now black



RB-INSERT-FIXUP – Case 2

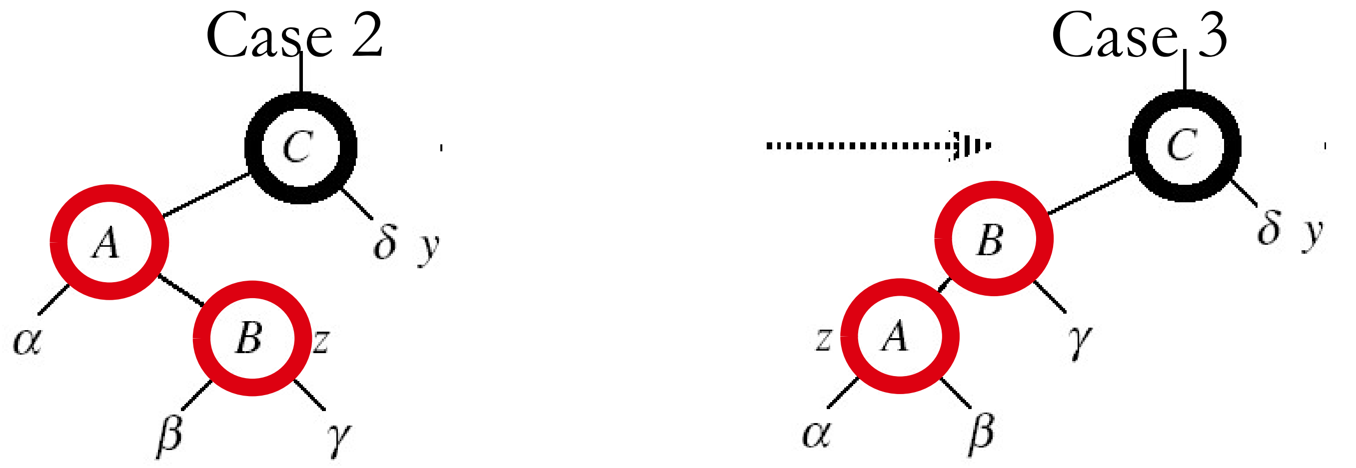
Case 2:

- z's "uncle" (y) is **black**
- z is a right child

Idea:

- z ← p[z]
- LEFT-ROTATE(T, z)

⇒ now z is a left child, and both z and p[z] are red ⇒ case 3



RB-INSERT-FIXUP(T, z)

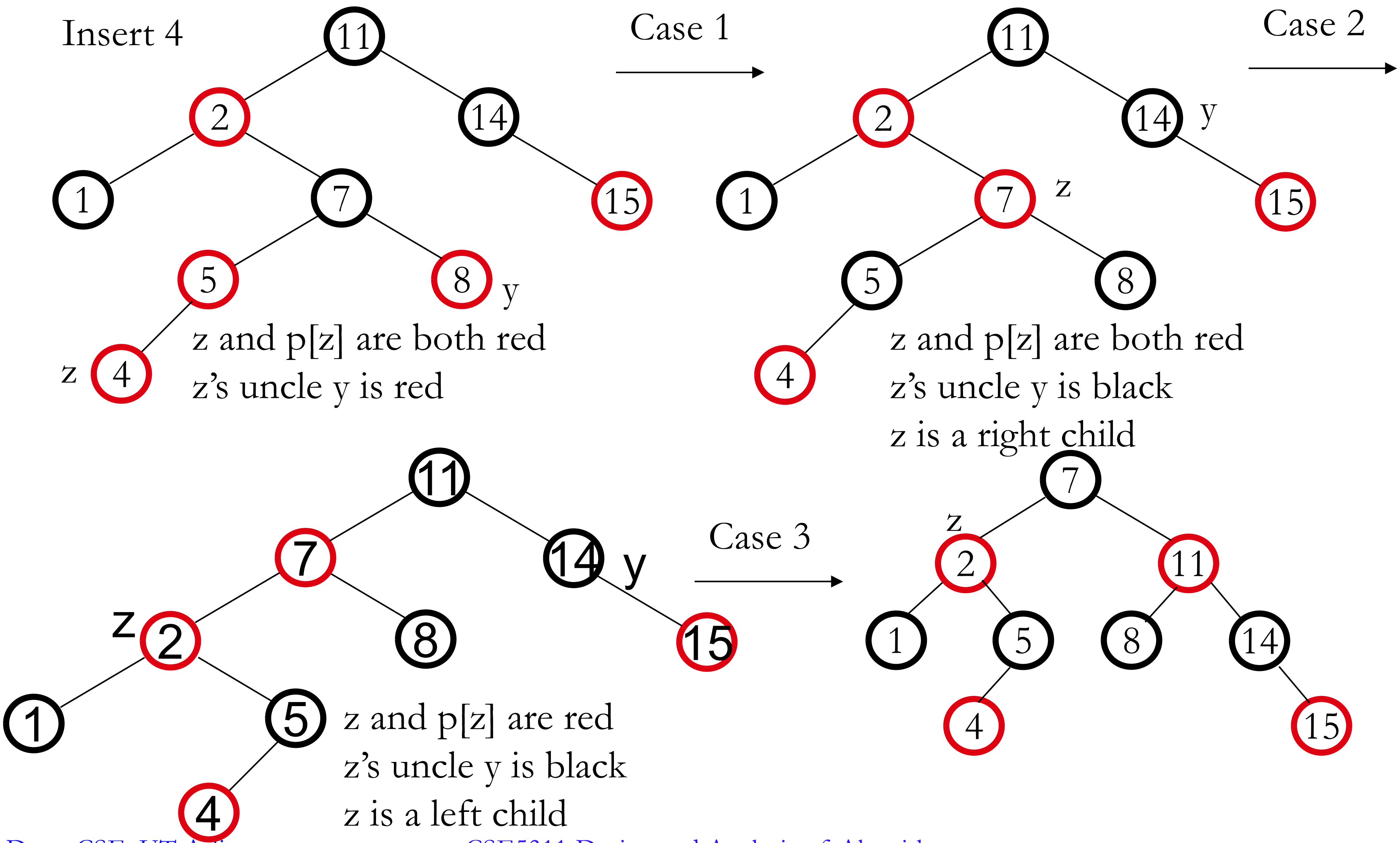
```
while color[p[z]] = RED
    do if p[z] = left[p[p[z]]]
        then y ← right[p[p[z]]]
            if color[y] = RED
                then Case1
            else if z = right[p[z]]
                then Case2
                Case3
    else (same as then clause with "right" and "left" exchanged)
color[root[T]] ← BLACK
```

The while loop repeats only when case1 is executed: O(lgn) times

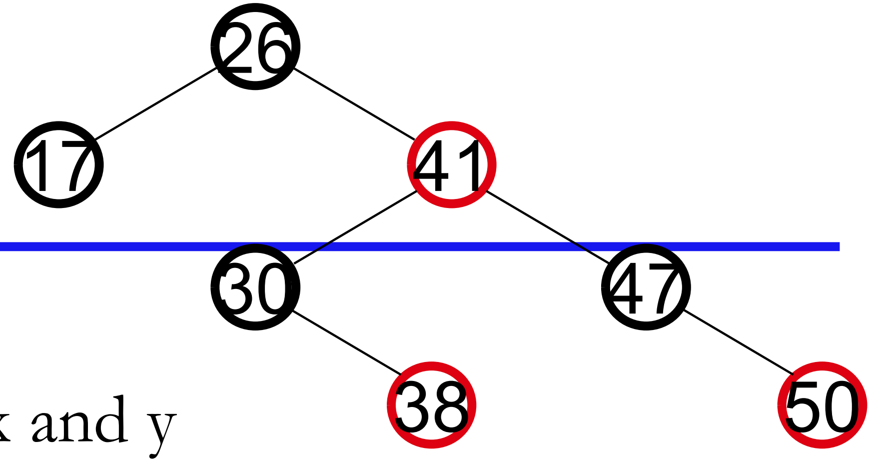
Set the value of x's "uncle"

We just inserted the root, or The red violation reached the root

Example


Insert 4
11
2
14
1
7
15
5
8
y
z
4
z and p[z] are both red
z's uncle y is red
Case 1
11
2
14
y
1
7
z
15
5
8
4
z and p[z] are both red
z's uncle y is black
z is a right child
Case 2
11
7
14
y
z
2
8
15
1
5
4
z and p[z] are red
z's uncle y is black
z is a left child
Case 3
7
z
2
11
1
5
8
14
4
15

RB-INSERT(T, z)




```
y ← NIL
x ← root[T]
```

- Initialize nodes x and y
- Throughout the algorithm y points to the parent of x

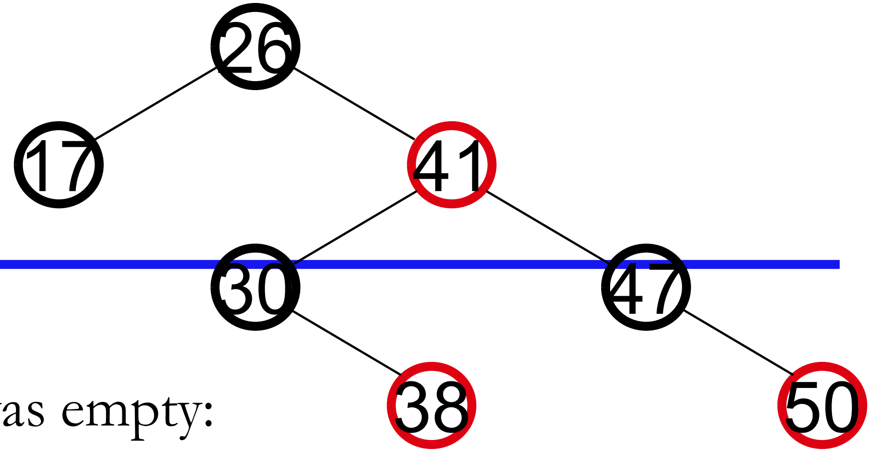
```
while x ≠ NIL
        do y ← x
            if key[z] < key[x]
                then x ← left[x]
                else x ← right[x]
```

- Go down the tree until reaching a leaf
- At that point y is the parent of the node to be inserted

```
p[z] ← y
```

- Sets the parent of z to be y

RB-INSERT(T, z)



if y = NIL
then root[T] ← z

The tree was empty: set the new node to be the root

else if key[z] < key[y]
then left[y] ← z
else right[y] ← z

Otherwise, set z to be the left or right child of y, depending on whether the inserted node is smaller or larger than y's key

left[z] ← NIL
right[z] ← NIL
color[z] ← RED

Set the fields of the newly added node

RB-INSERT-FIXUP(T, z)

Fix any inconsistencies that could have been introduced by adding this new red node

Analysis of RB-INSERT

- Inserting the new element into the tree O(lgn)
- RB-INSERT-FIXUP
  - The while loop repeats only if CASE 1 is executed
  - The number of times the while loop can be executed is O(lgn)
- Total running time of RB-INSERT: O(lgn)

Red-Black Trees - Summary

- **Operations on red-black-trees:**
  - SEARCH O(h)
  - PREDECESSOR O(h)
  - SUCCESOR O(h)
  - MINIMUM O(h)
  - MAXIMUM O(h)
  - INSERT O(h)
  - DELETE O(h)
- **Red-black-trees guarantee that the height of the tree will be O(lgn)**

Problems

- What is the ratio between the longest path and the shortest path in a red-black tree?

  - The shortest path is at least bh(root)
  - The longest path is equal to h(root)
  - We know that $h(root) \leq 2bh(root)$
  - Therefore, the ratio is ≤ 2

Problems

- What red-black tree property is violated in the tree below? How would you restore the red-black tree property in this case?
  - Property violated: if a node is red, both its children are black
  - Fixup: color 7 black, 11 red, then right-rotate around 11