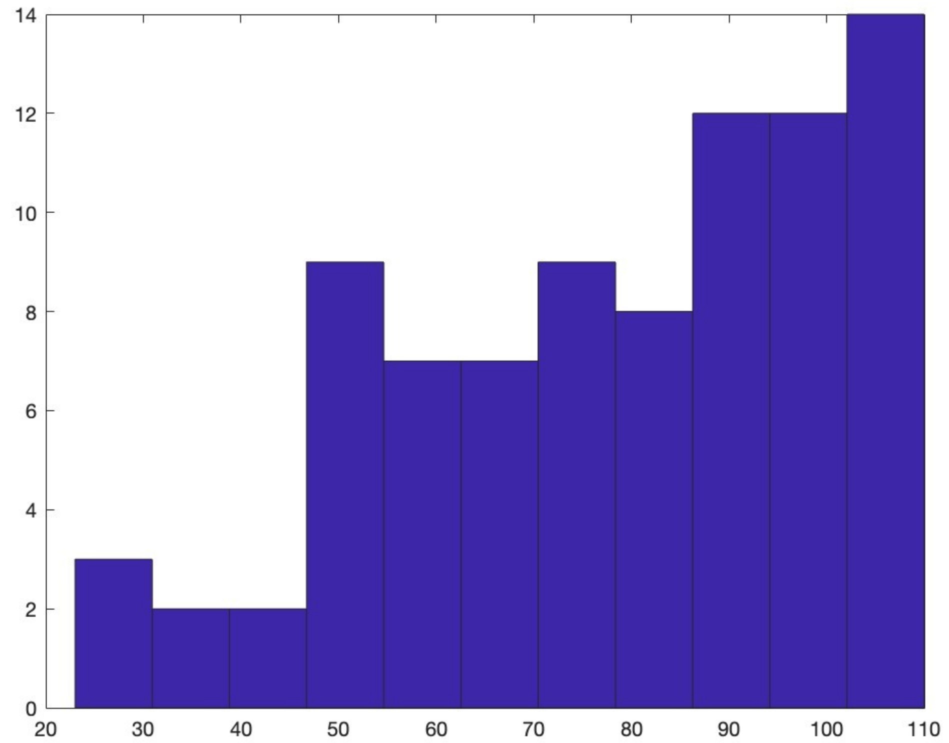
Analysis of Algorithms
CS 477/677

Instructor: Monica Nicolescu

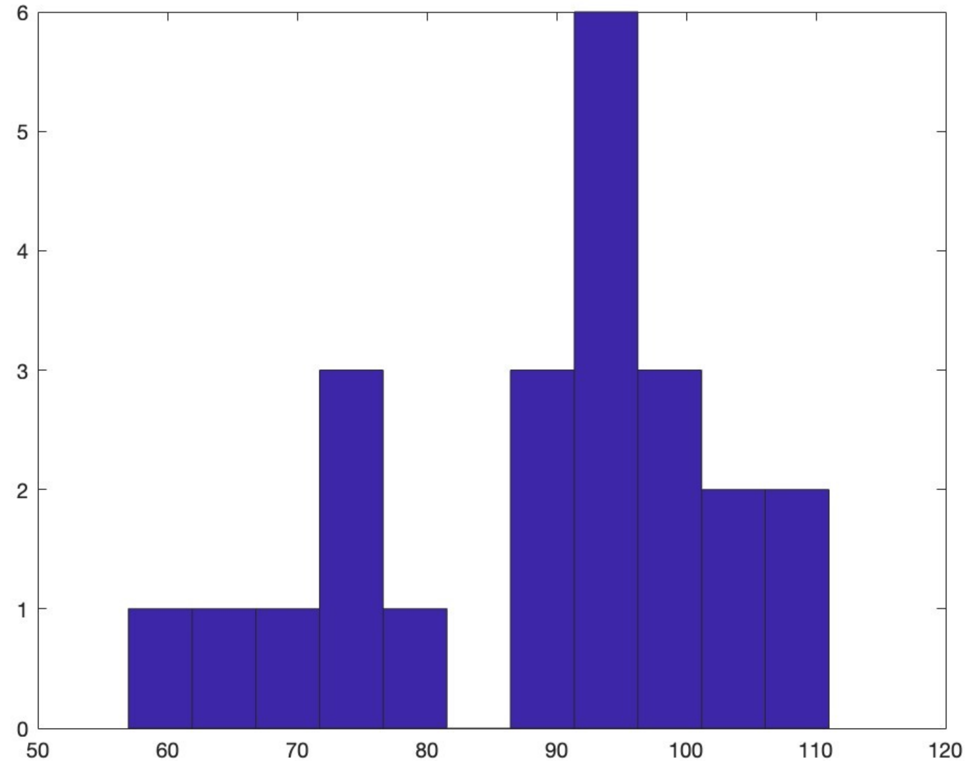
Lecture 13

Midterm Results - CS 477



Min = 23 Max = 110 Average = 78.2

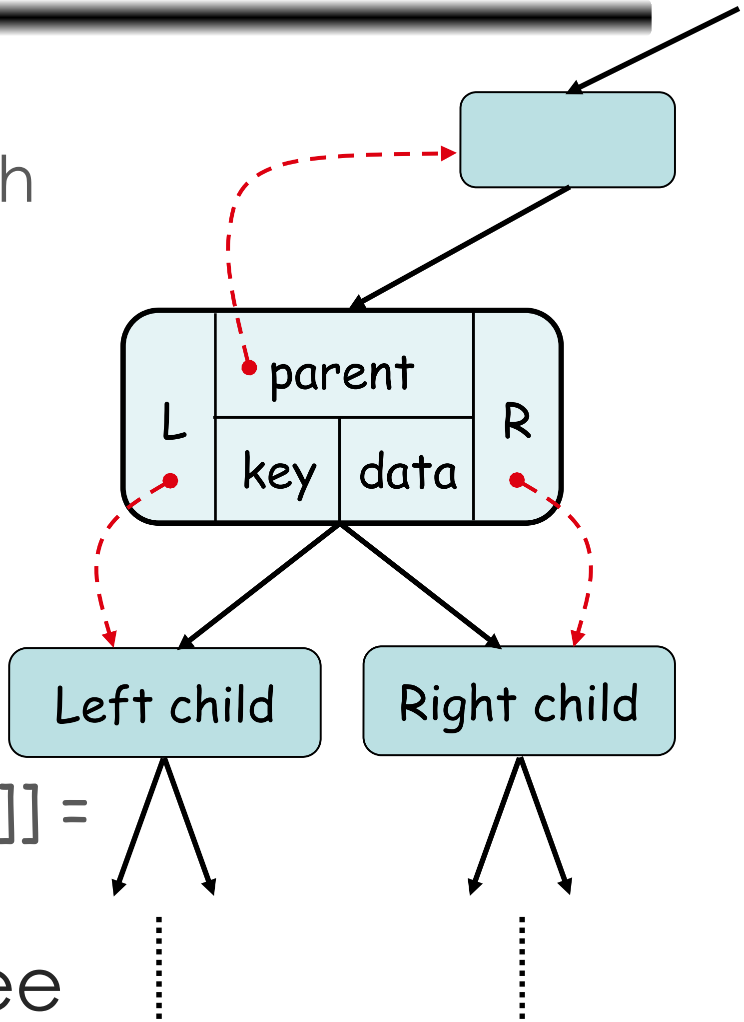
Midterm Results - CS 677



Min = 57 Max = 111 Average = 89.26

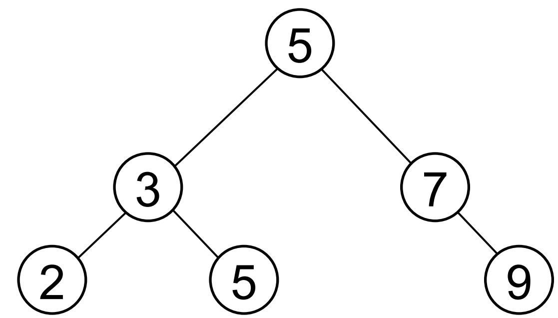
Binary Search Trees

- Tree representation:
  - A linked data structure in which each node is an object
- Node representation:
  - **Key** field
  - Satellite data
  - **Left**: pointer to left child
  - **Right**: pointer to right child
  - **p**: pointer to parent (**p [root [T]] =** **NIL**)
- Satisfies the binary search tree property



Binary Search Tree Example

- Binary search tree property:
  - If **y** is in left subtree of **x**, then **key [y] ≤ key [x]**
  - If **y** is in right subtree of **x**, then **key [y] ≥ key [x]**



Binary Search Trees

- Support many dynamic set operations
  - SEARCH, MINIMUM, MAXIMUM, PREDECESSOR, SUCCESSOR, INSERT, DELETE
- Running time of basic operations on binary search trees
  - On average: $\Theta(\lg n)$
    - The expected height of the tree is $\lg n$
  - In the worst case: $\Theta(n)$
    - The tree is a linear chain of n nodes

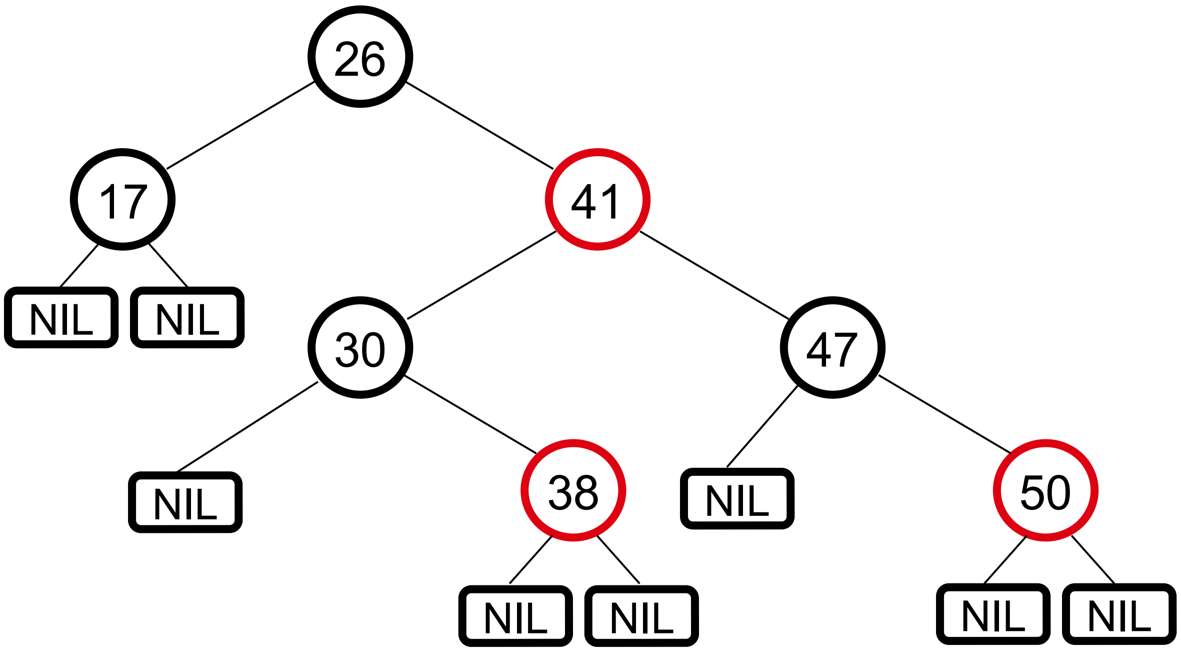
Red-Black Trees

- "Balanced" binary trees guarantee an ***O(lgn)*** running time on the basic dynamic-set operations
- Red-black tree
  - Binary tree with an additional attribute for its nodes: **color** which can be **red** or **black**
  - 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 another ⇒ the tree is balanced
  - The nodes inherit all the other attributes from the binary-search trees: **key**, **left**, **right**, **p**

Red-Black Trees Properties

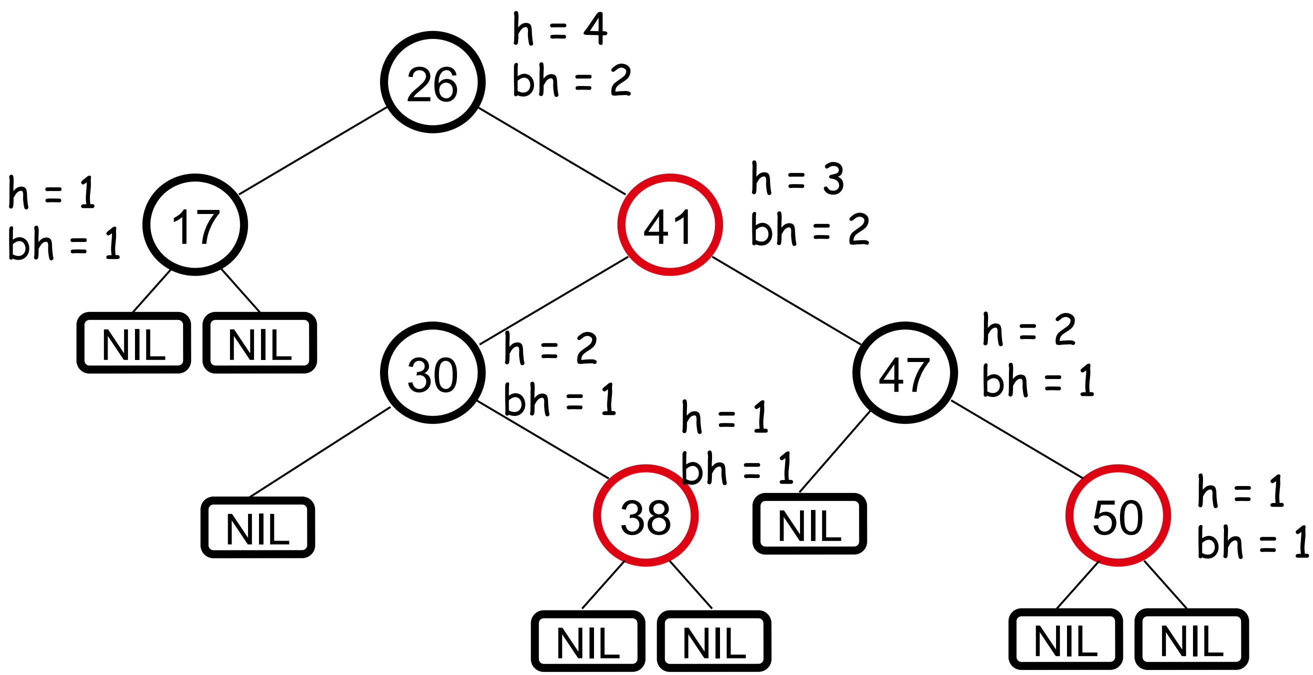
1. Every **node** is either **red** or **black**
2. The **root** is **black**
3. Every *leaf* (NIL) is **black**
4. If a node is red, then both its children are black
   - No two red nodes in a row on a simple path from the root to a leaf
5. For each node, all paths from the node to descendant leaves contain the same number of black nodes

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

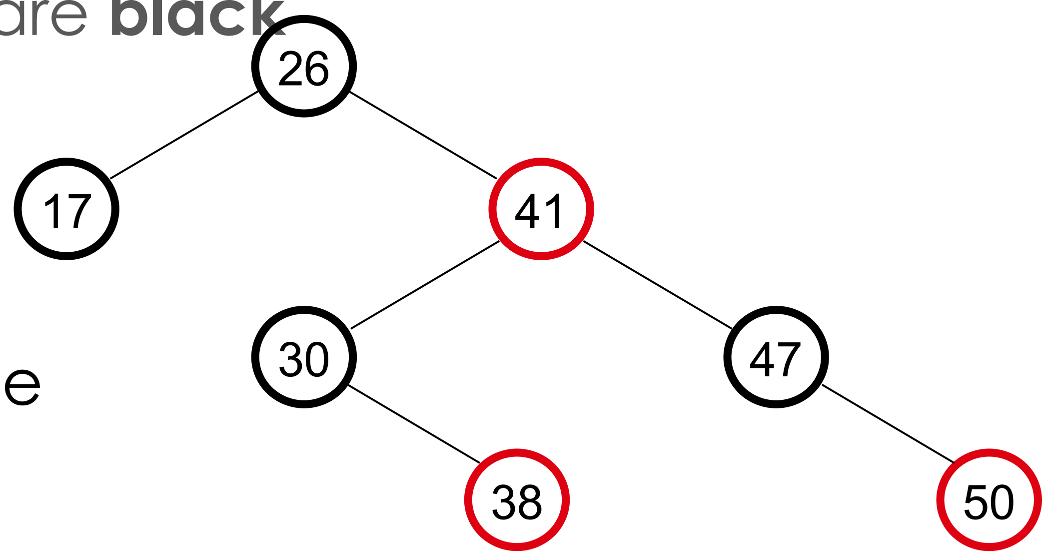
Black-Height of a Node



- **Height of a node:** the number of edges in a longest path to a leaf
- **Black-height** of a node **x**: **bh(x)** is the number of black nodes (including NIL) on a path from **x** to leaf, not counting **x**

Properties of Red-Black Trees

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



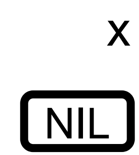
Property 4: if a node is red then both its children are black

Properties of Red-Black Trees

Claim

The subtree rooted at any node **x** contains at least $2^{bh(x)} - 1$ internal nodes

Proof: By induction on height of **x**



Basis: height[x] = 0 ⇒

x is a leaf (NIL[T]) ⇒

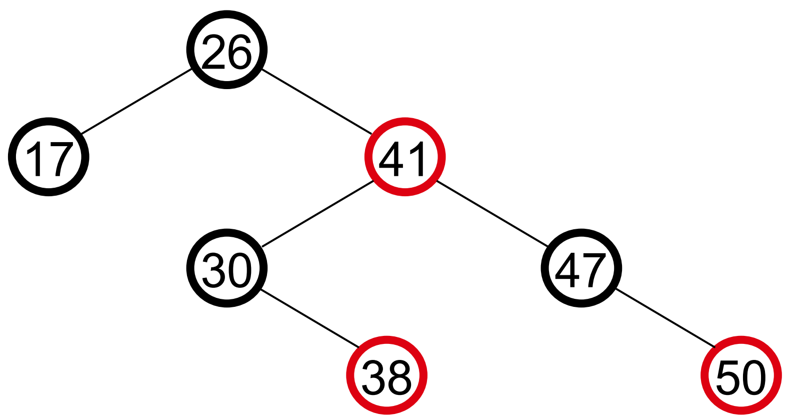
bh(x) = 0 ⇒

# of internal nodes: $2^0 - 1 = 0$

Properties of Red-Black Trees

Inductive step:

- Let **height**(**x**) = **h** and **bh(x)** = **b**
- Any child **y** of **x** has:
  - bh (y) = b (if the child is **red**), or
  - bh (y) = b - 1 (if the child is **black**)



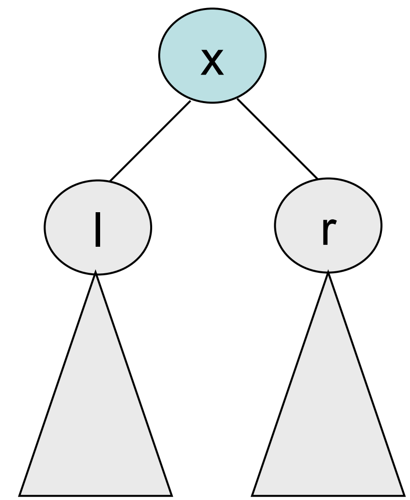
Properties of Red-Black Trees

- Want to prove:
  - The subtree rooted at any node **x** contains at least $2^{bh(x)} - 1$ internal nodes
- Assume true for children of **x**:
  - Their subtrees contain at least $2^{bh(x) - 1} - 1$ internal nodes
- The subtree rooted at **x** contains at least:

  $(2^{bh(x) - 1} - 1) + (2^{bh(x) - 1} - 1) + 1 =$

  $2 \cdot (2^{bh(x) - 1} - 1) + 1 =$

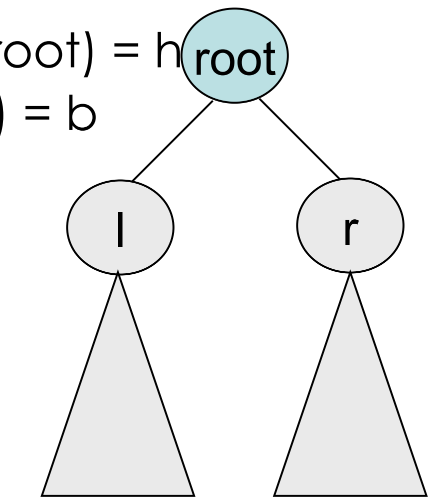
  $2^{bh(x)} - 1$ internal nodes



Properties of Red-Black Trees

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

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



Proof:

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

number n of internal nodes

since $b \geq h/2$

- Add 1 to all sides and then take logs:

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

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

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

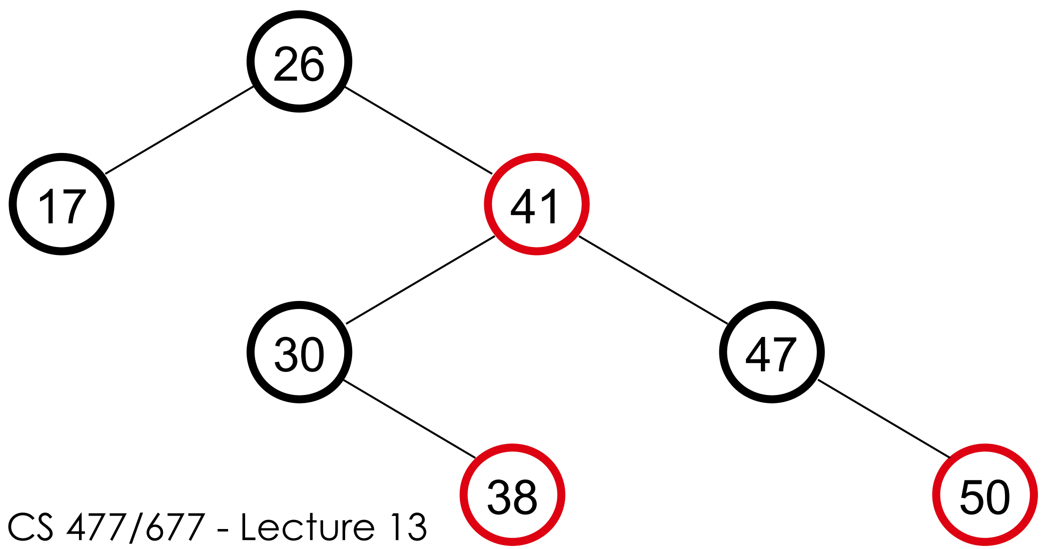
Operations on Red-Black Trees

- The non-modifying binary-search tree operations MINIMUM, MAXIMUM, SUCCESSOR, PREDECESSOR, and SEARCH run in $O(h)$ time
  - They take $O(\lg n)$ time on red-black trees
- What about TREE-INSERT and TREE-DELETE?
  - They will still run in $O(\lg n)$
  - We have to guarantee that the modified tree will still be a red-black tree

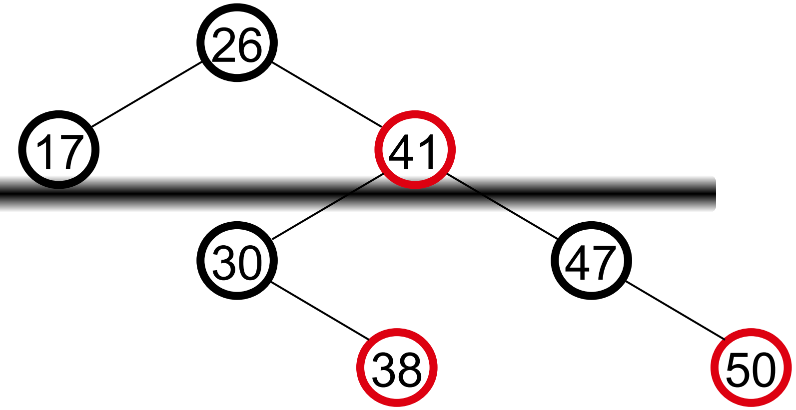
INSERT

INSERT: what color to make the new node?

- Red? Let's insert 35!
  - Property 4: if a node is red, then both its children are black
- Black? Let's insert 14!
  - Property 5: 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? **Red?**

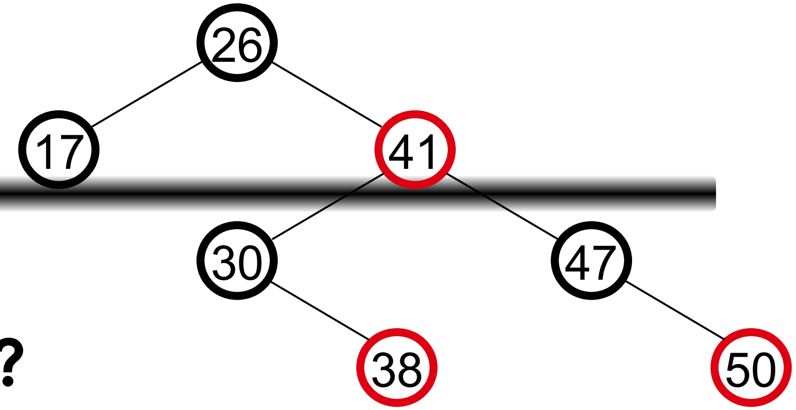
1. Every **node** is either **red** or **black** OK!
2. The **root** is **black** OK!
3. Every *leaf* (NIL) is **black** OK!
4. If a node is red, then both its children are black

OK! Does not change any black heights

OK! Does not 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

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**
3. Every **leaf** (NIL) is **black** OK!
4. If a node is red, then both its children are black

Not OK! If removing the root and the child that replaces it is **red**

Not OK! Could change the black heights of some nodes

Not OK! Could create two red nodes in a row

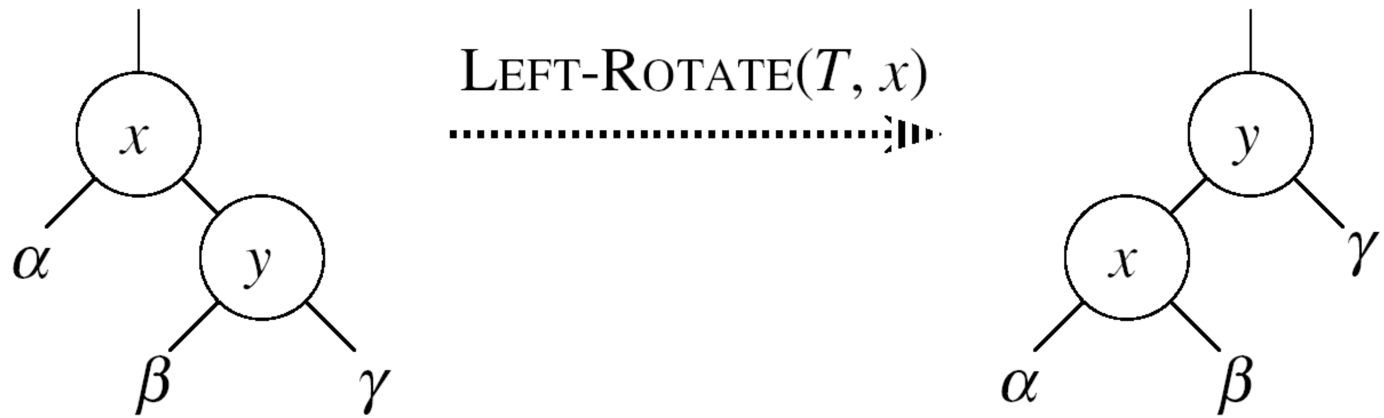
5. For each node, all paths from the node to descendant leaves contain the same number of black nodes

Rotations

- Operations for restructuring 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

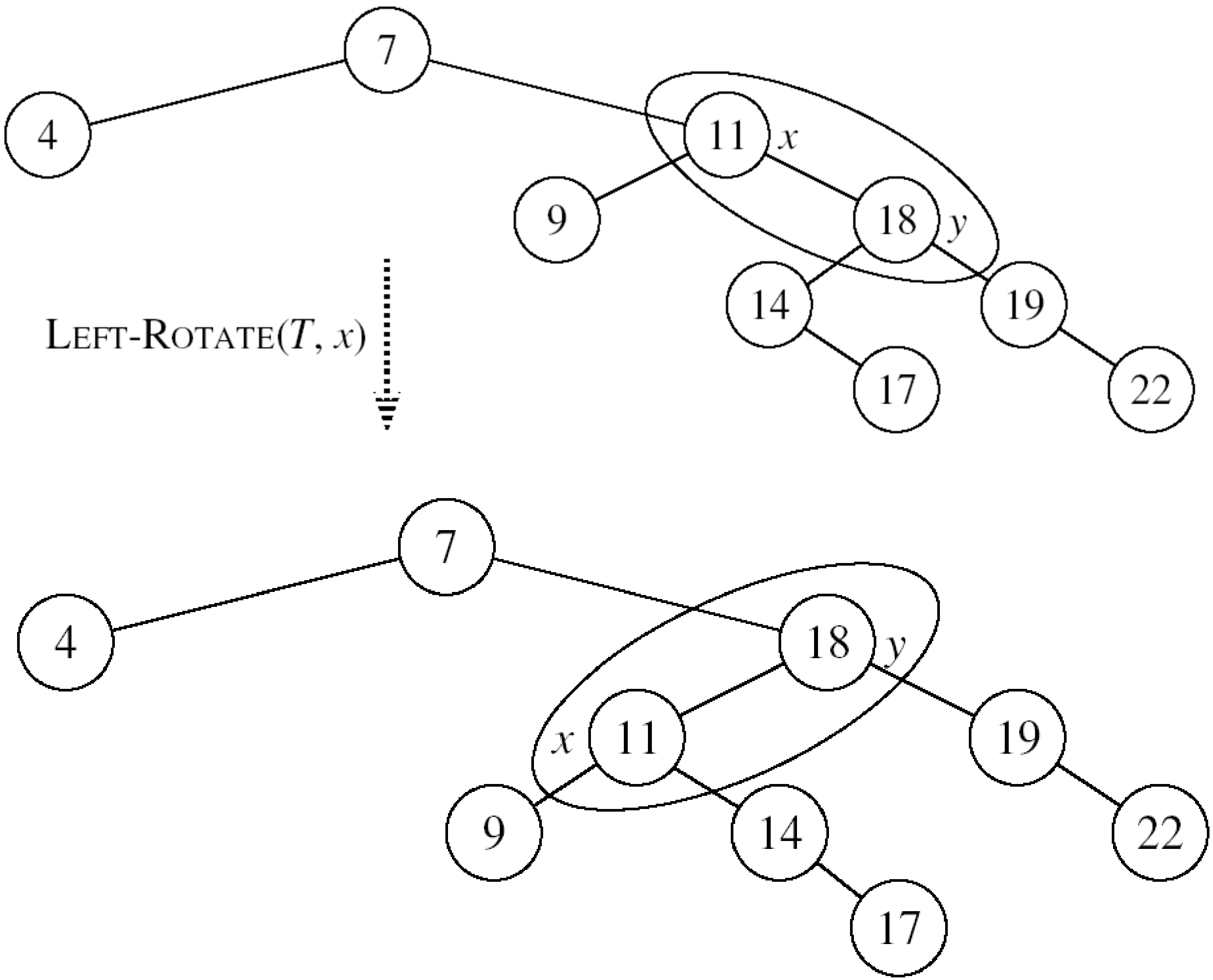
Left Rotations

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



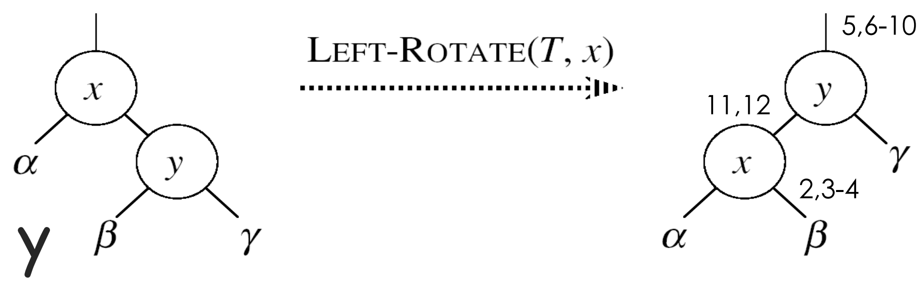
- Idea:
  - Pivots around the link from **x** to **y**
  - Makes **y** the new root of the subtree
  - **x** becomes **y**'s left child
  - **y**'s left child becomes **x**'s right child

Example: LEFT-ROTATE



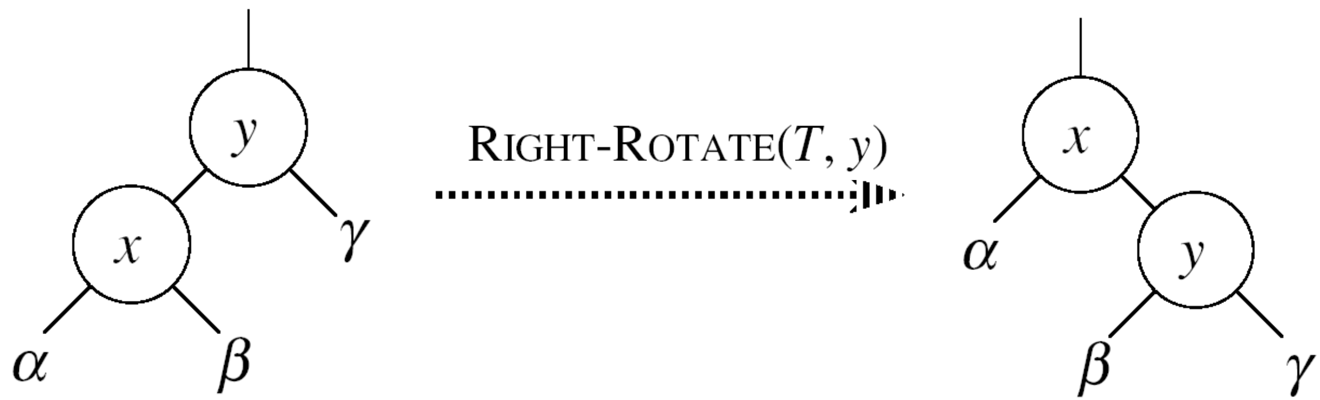
LEFT-ROTATE(T, x)

1. y ← right[x] ►Set y
2. right[x] ← left[y] ► y's left subtree becomes x's right subtree
3. **if** left[y] ≠ NIL
4.   **then** p[left[y]] ← x ► Set the parent relation from left[y] to x
5. p[y] ← p[x] ► The parent of x becomes the parent of 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 ► y becomes x's parent



Right Rotations

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

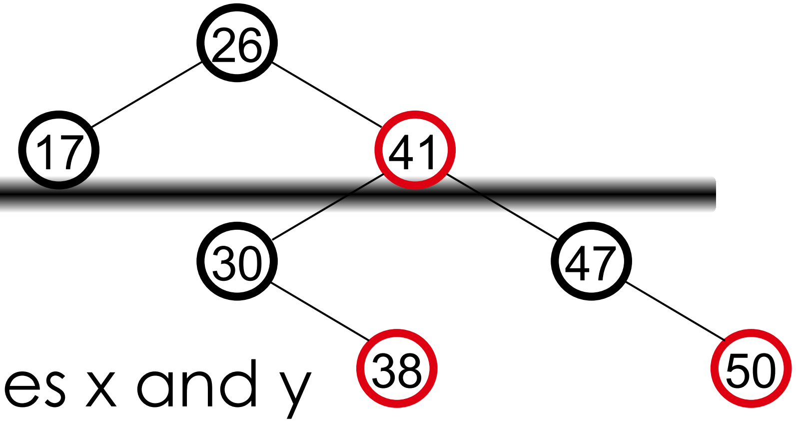


- Idea:
  - Pivots around the link from **y** to **x**
  - Makes **x** the new root of the subtree
  - **y** becomes **x**'s right child
  - **x**'s right child becomes **y**'s left child

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-INSERT(T, z)




```
y ← NIL
x ← root[T]
```

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

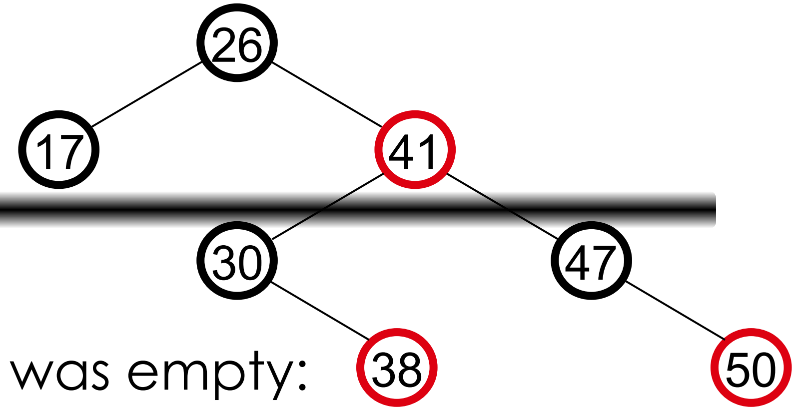
```
while x ≠ NIL
        do y ← x
              if key[z] < key[x]
                then x ← left[x]
                else x ← right[x]
```

- 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
    else if key[z] < key[y]
            then left[y] ← z
            else right[y] ← z
left[z] ← NIL
right[z] ← NIL
color[z] ← RED
RB-INSERT-FIXUP(T, z)
```

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

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

Set the fields of the newly added node

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

RB Properties Affected by Insert

1. Every **node** is either **red** or **black** OK!

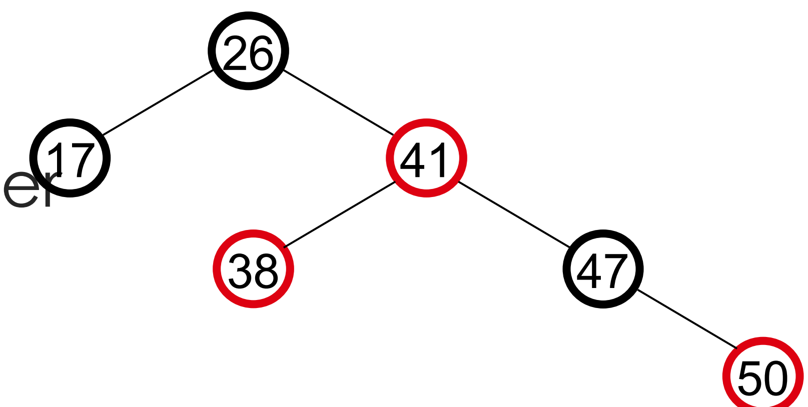
If z is the root ⇒ not OK

2. The **root** is **black**
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

OK!

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

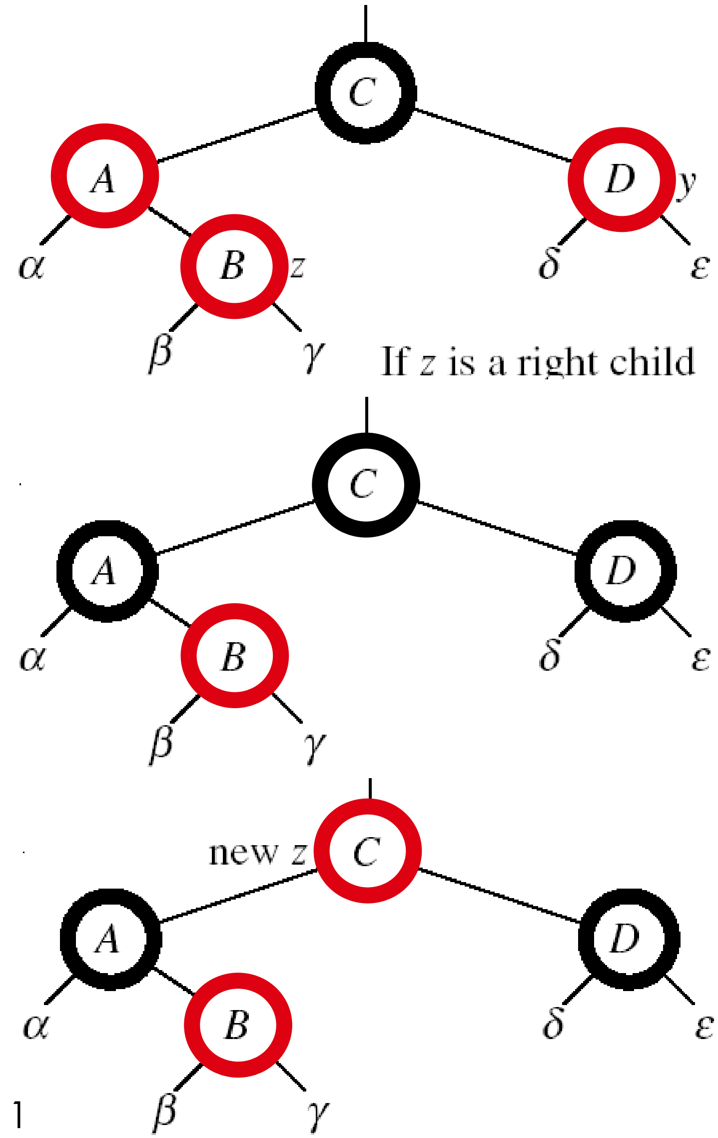


RB-INSERT-FIXUP – Case 1

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**
  - Push the **red** node up the tree
- Make **z = p[p[z]]**



RB-INSERT-FIXUP – Case 1

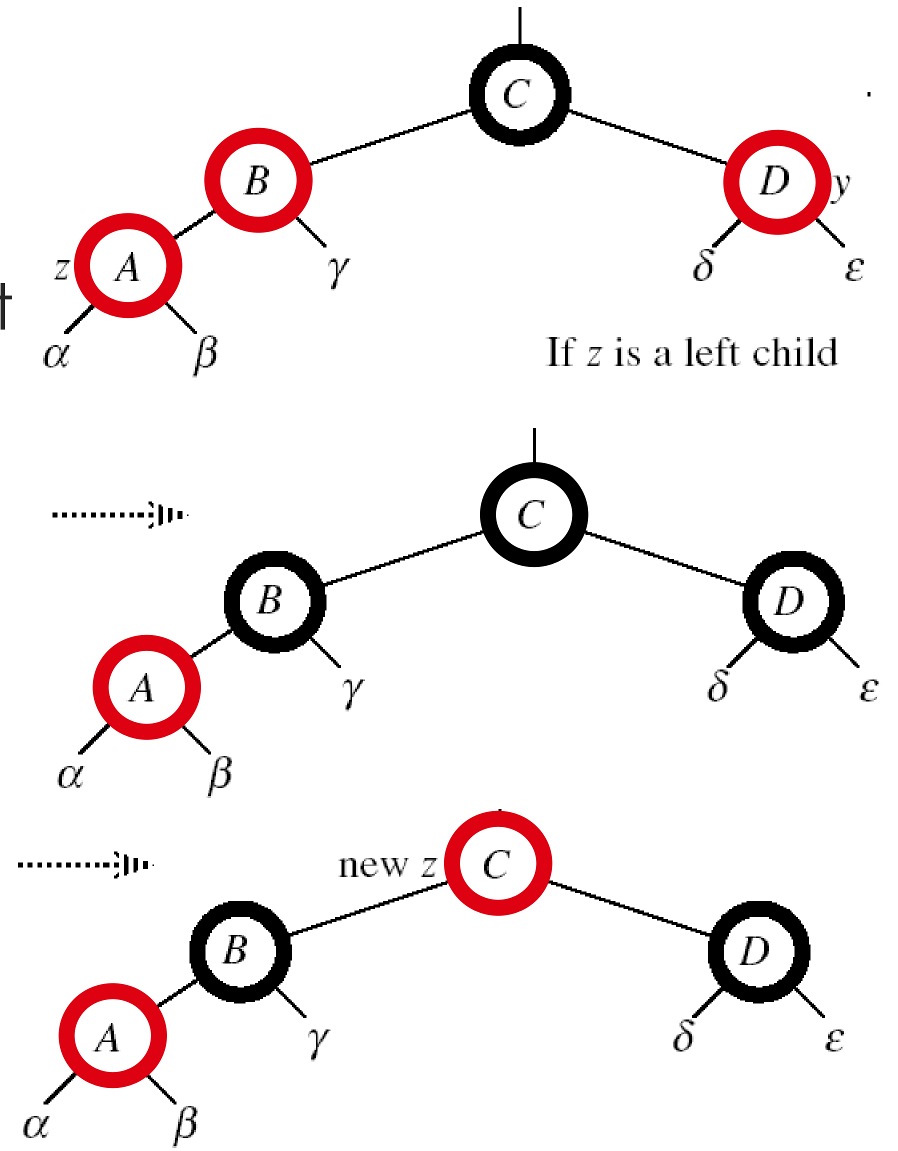
z's "uncle" (y) is **red**

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]]**

  Case1

  – Push the **red** node up the tree



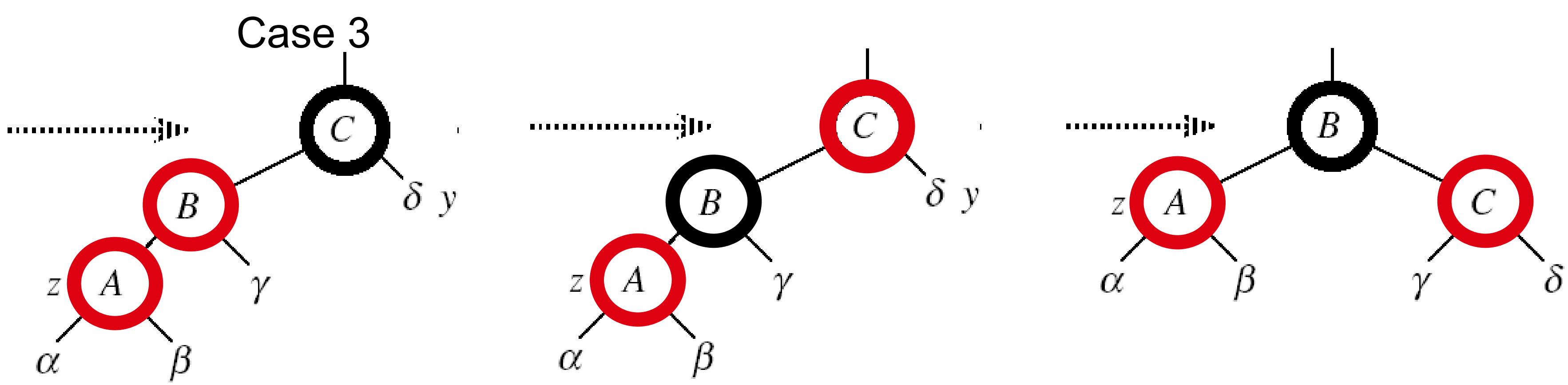
RB-INSERT-FIXUP – Case 3

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]]) Case3
- No longer have 2 reds in a row
- p[z] is now black



RB-INSERT-FIXUP – Case 2

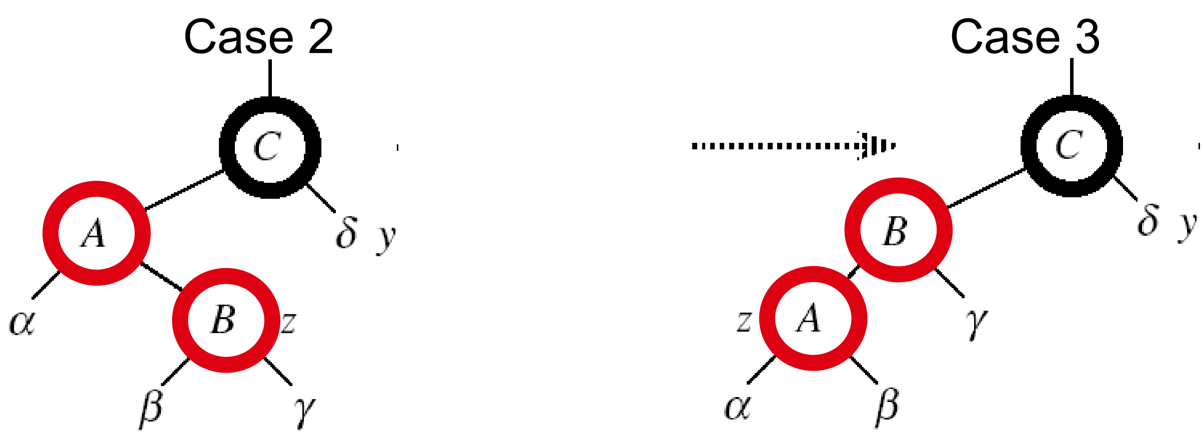
Case 2:

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

Idea:

- **z** ← **p[z]**
- LEFT-ROTATE(T, **z**) Case2

⇒ 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
```

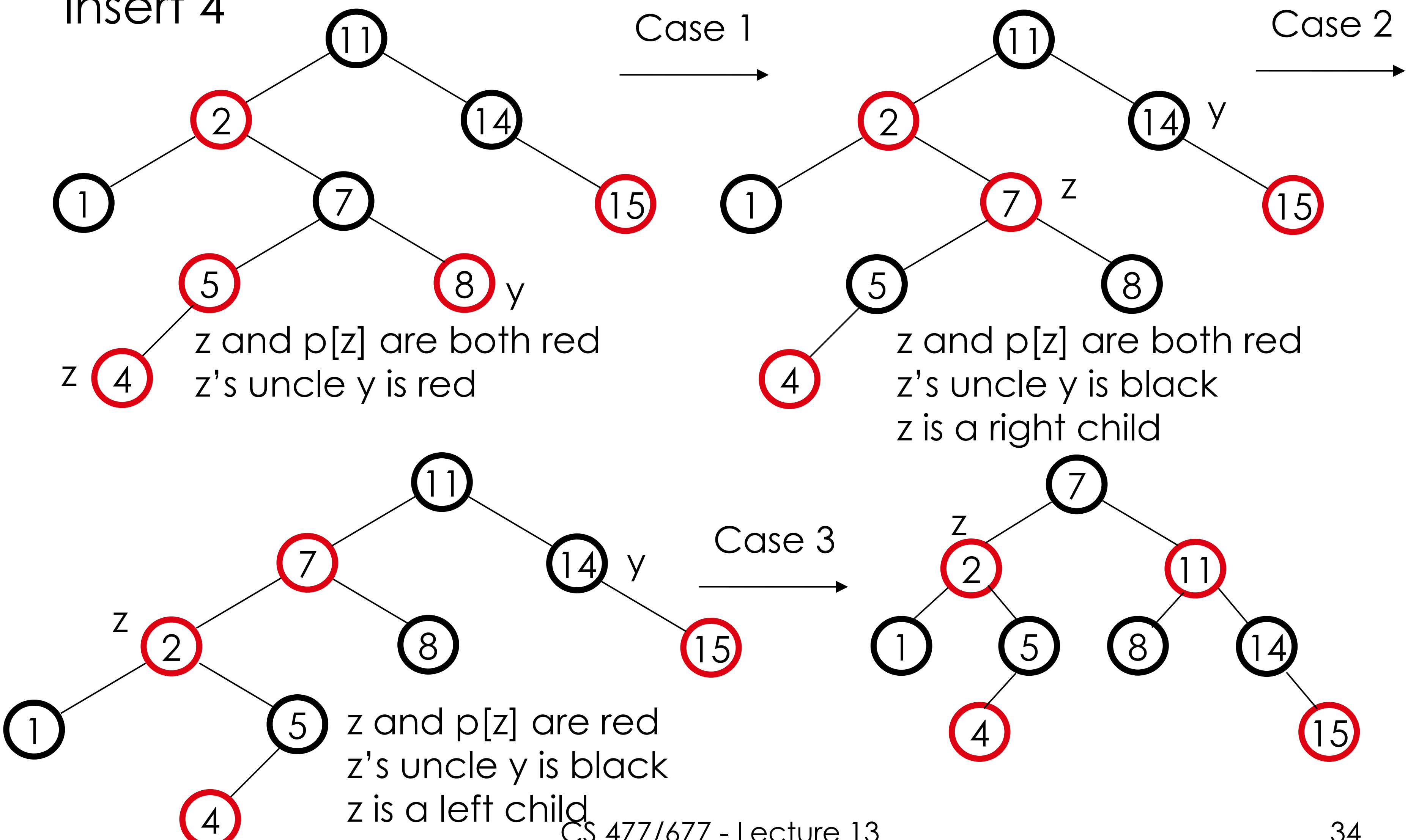
The while loop repeats only when case1 is executed: O(lgn) times

Set the value of x's "uncle"

We just inserted the root, or the red node reached the root

Example

Insert 4



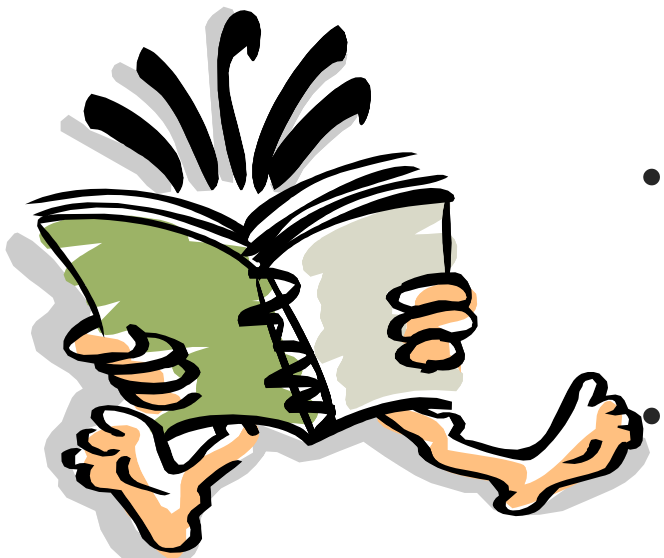
Analysis of RB-INSERT

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

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 $\boldsymbol{O(\lg n)}$

Readings



- For this lecture
  - Sections 6.3, 6,5
  - Chapter 13
- Coming next
  - Chapter 17