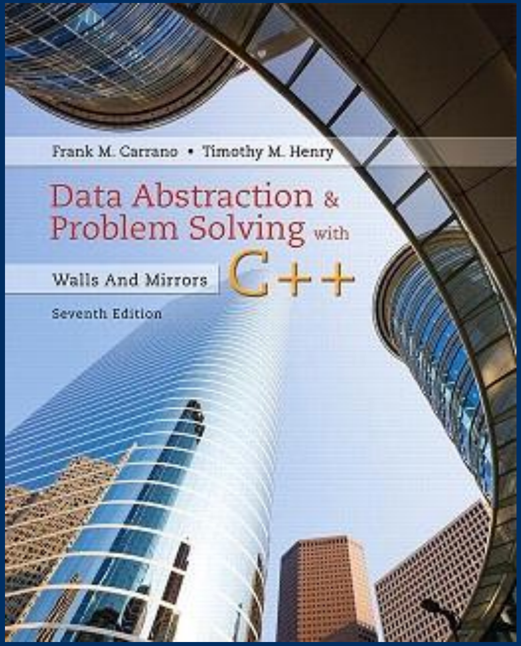
Chapter 19

Balanced Search Trees

CS 302 - Data Structures

M. Abdullah Canbaz



Modified from authors’ slides

Reminders

- Assignment 7 is available
  - Due Wednesday, May 7th at 2pm
  - TA
    - Shehryar Khattak,
      Email: *shehryar [at] nevada {dot} unr {dot} edu*,
      Office Hours: Friday, 11:00 am - 1:00 pm at ARF 116
- Assignment 8 is available
  - Due Wednesday, May 16th at 2pm
  - TA
    - Athanasia Katsila,
      Email: *akatsila [at] nevada {dot} unr {dot} edu*,
      Office Hours: Thursdays, 10:30 am - 12:30 pm at SEM 211

Red-Black Trees

Modified from Dr Monica Nicolescu

Red-Black-Trees

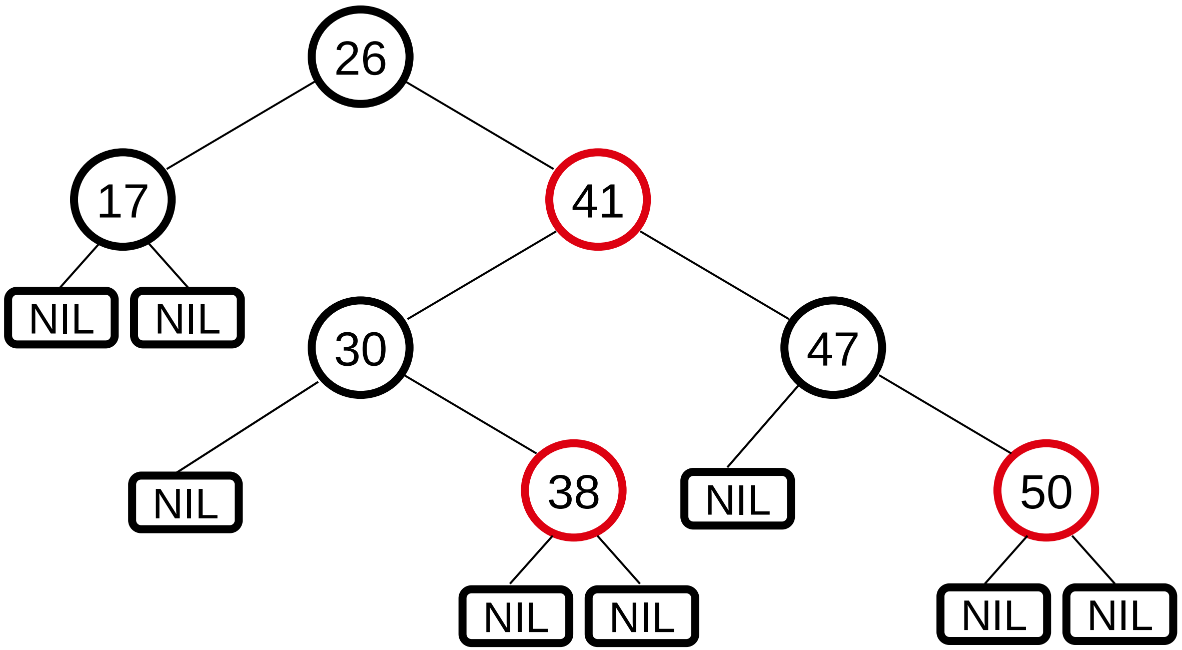
- A **red–black tree** is
  - a kind of self-balancing binary search tree.
  - Each node of the binary tree has an extra bit, and that bit is often interpreted as the color (red or black) of the node.
  - These color bits are used to ensure the tree remains approximately balanced during insertions and deletions.

Red-Black-Trees Properties

(**Binary search tree property is satisfied**)

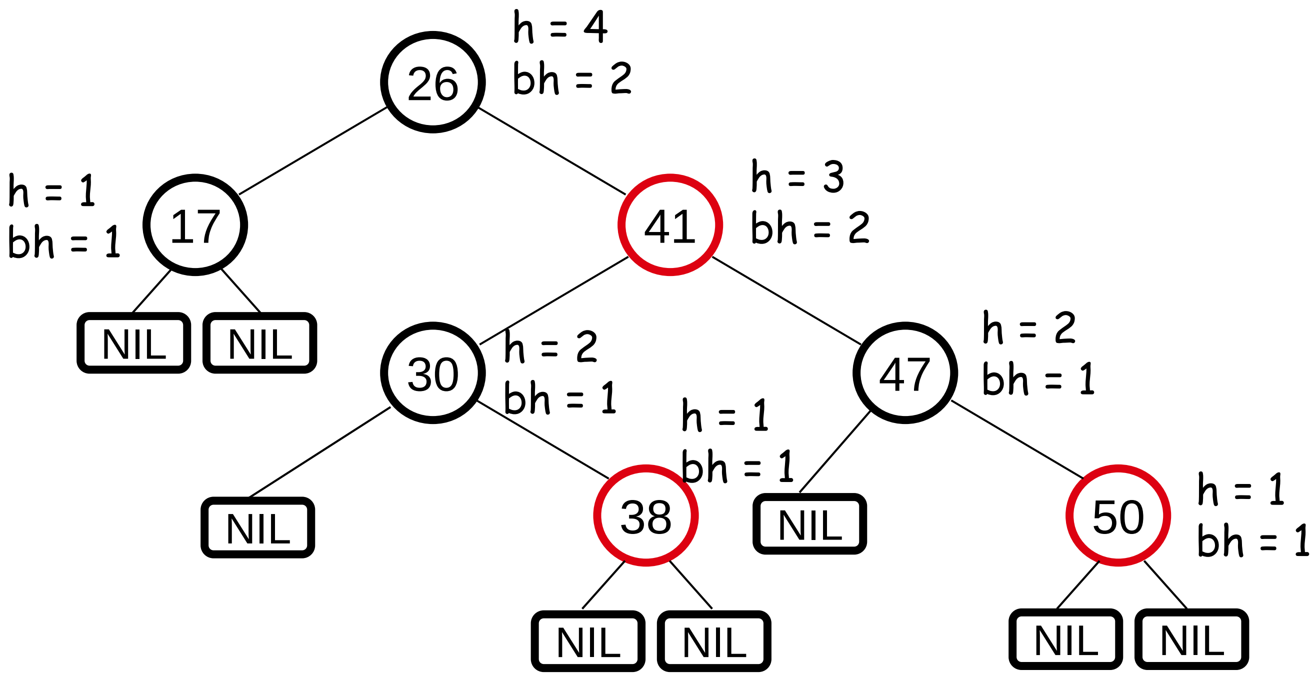
1. Every node is either red or **black**
2. The root is **black**
3. Every leaf (NIL) is **black**
4. If a node is red, then both its children are **black**
   - No two consecutive red nodes on a simple path from the root to a leaf
5. For each node, all paths from that node to a leaf contain the same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

Example: RED-BLACK-TREE



- For convenience, we add NIL nodes and refer to them as the leaves of the tree.
  - Color[NIL] = BLACK

Definitions

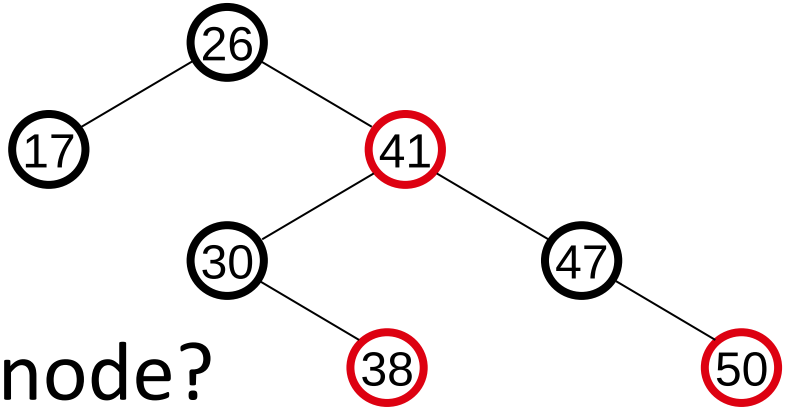


- **Height of a node:** the number of edges in the **longest** path to a leaf
- **Black-height** bh(x) of a node x: the number of black nodes (including NIL) on the path from x to a leaf, <u>not counting x</u>

Height of Red-Black-Trees

A red-black tree with n internal nodes has height <u>at most</u> $2\log(N+1)$

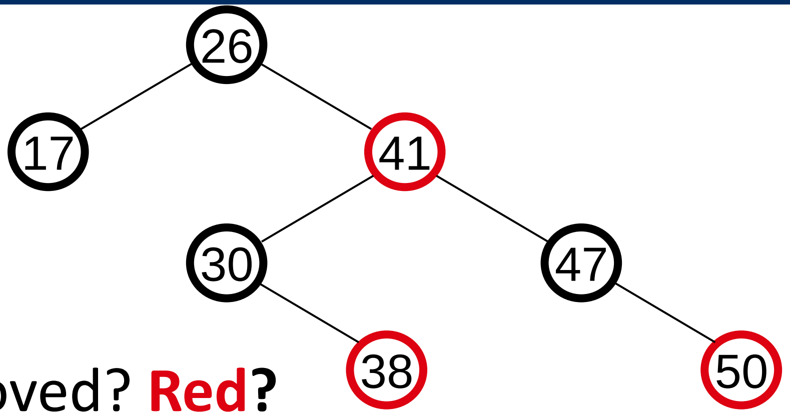
Insert Item



What color to make the new node?

- Red?
  - Let's insert 35!
    - Property 4 is violated: if a node is red, then both 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 Item

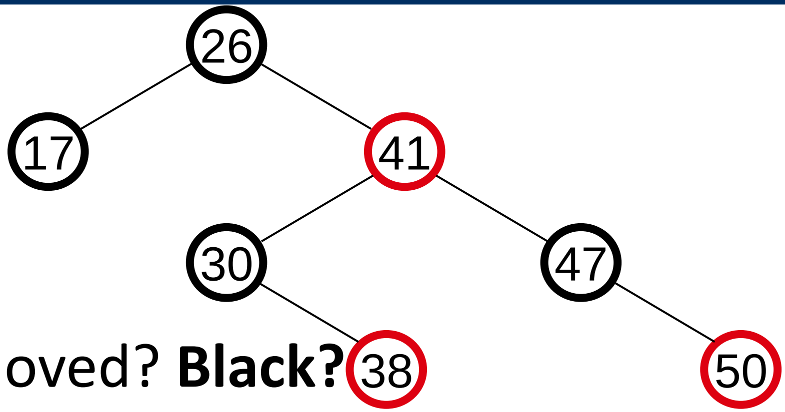


What color was the node that was removed? **Red?**

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!

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

Delete Item



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 change the black heights of some nodes

Not OK! Could create two red nodes in a row

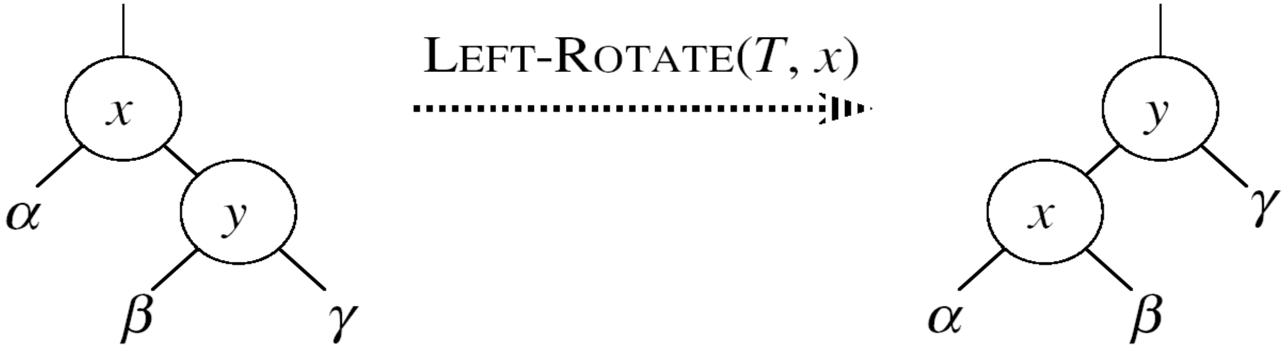
5. For each node, all paths from the node to descendant leaves contain the same number of black nodes

Rotations

- Operations for re-structuring the tree after insert and delete operations
  - Together with some node re-coloring, they help restore the red-black-tree property
  - Change some of the pointer structure
  - **Preserve** the binary-search tree property
- Two types of rotations:
  - Left & right rotations

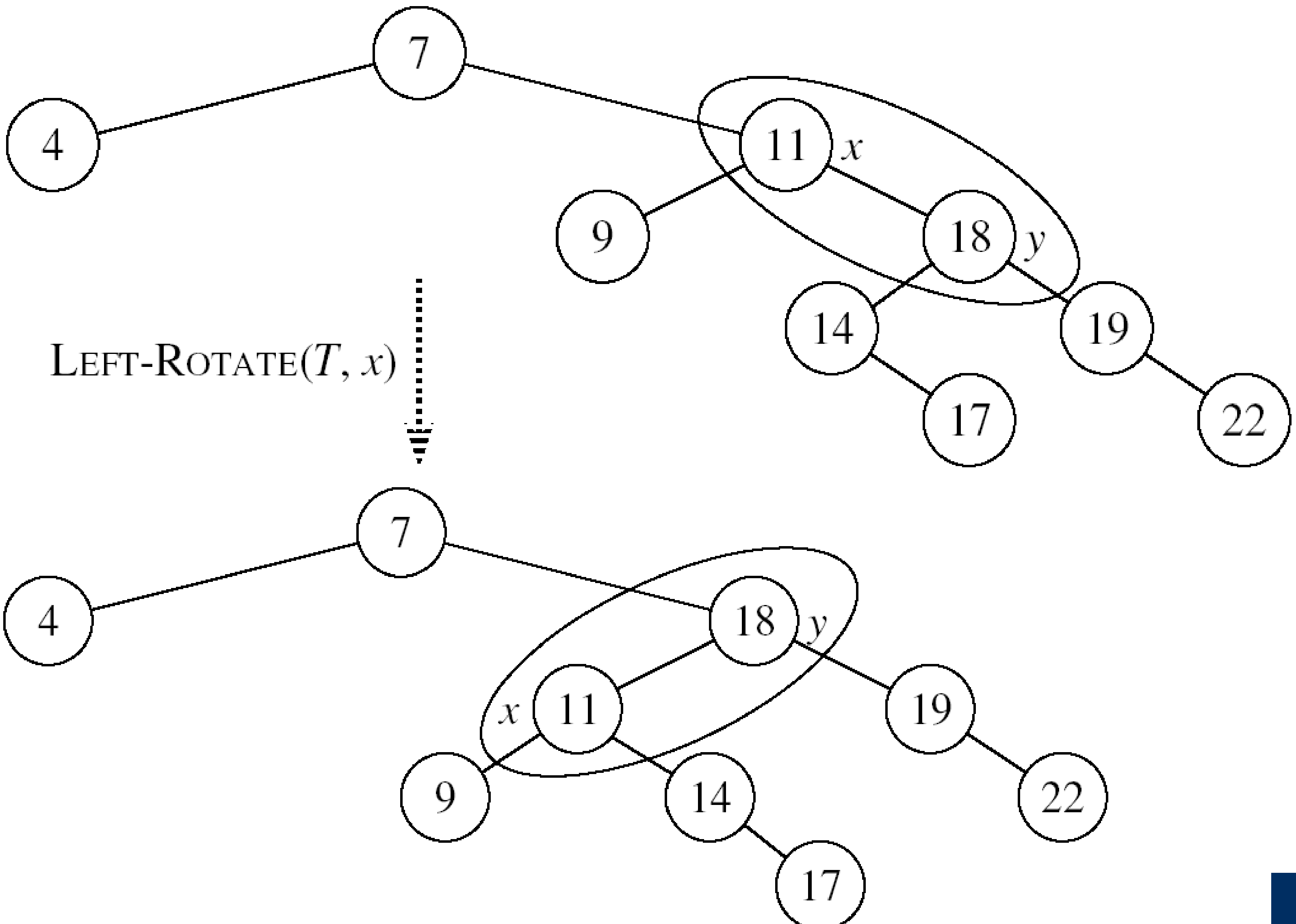
Left Rotations

- Assumptions for a left rotation on a node x:
  - The right child y of x 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)

$y \leftarrow right[x]$ ► Set y

$right[x] \leftarrow left[y]$ ► y' s left subtree becomes x' s right subtree

if $left[y] \neq NIL$

  then $p[left[y]] \leftarrow x$ ► Set the parent relation from left[y] to x

$p[y] \leftarrow p[x]$ ► The parent of x becomes the parent of y

if $p[x] = NIL$

  then $root[T] \leftarrow y$

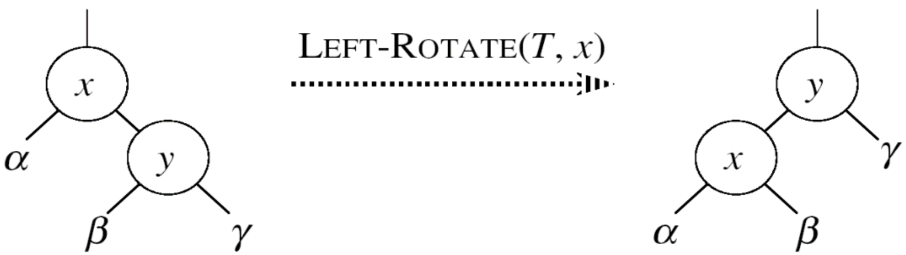
  else if $x = left[p[x]]$

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

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

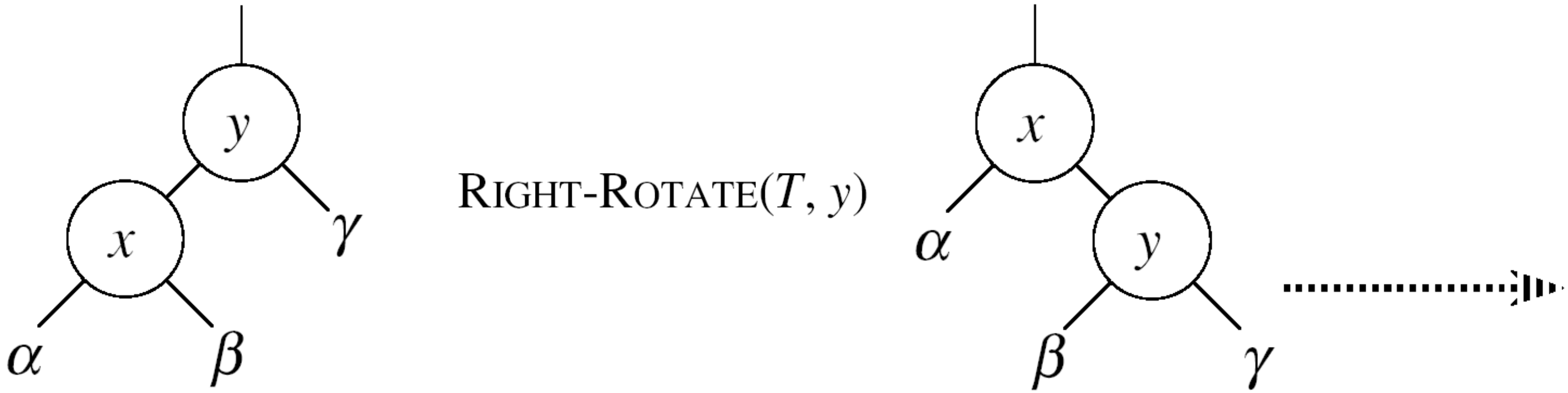
$left[y] \leftarrow x$ ► Put x on y' s left

$p[x] \leftarrow y$ ► y becomes x' s parent



Right Rotations

- Assumptions for a right rotation on a node x:
  - The left child x of y 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

Insert Item

- 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

RB-INSERT(T, z)

```
y ← NIL
x ← root[T]
```

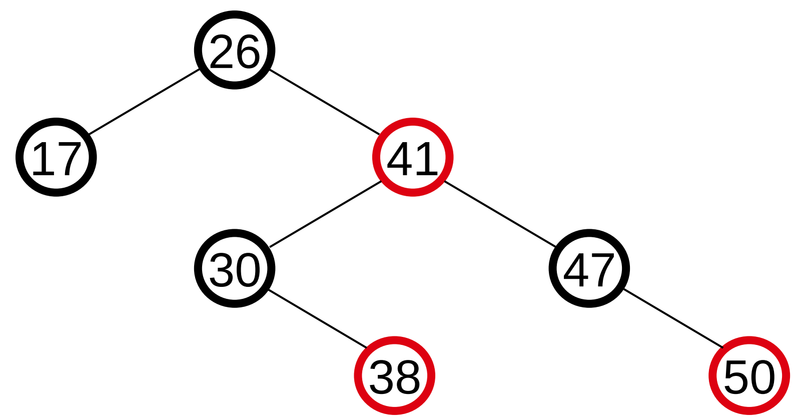
- Initialize nodes x and y
- Throughout the algorithm y points to the parent of x

```
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

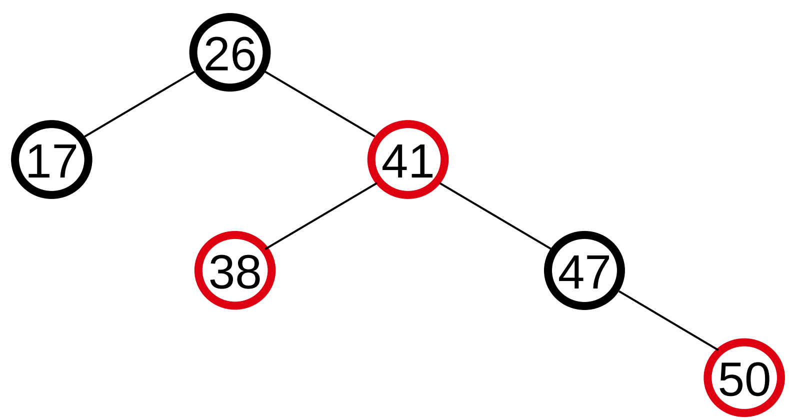
RB Properties Affected by Insert

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

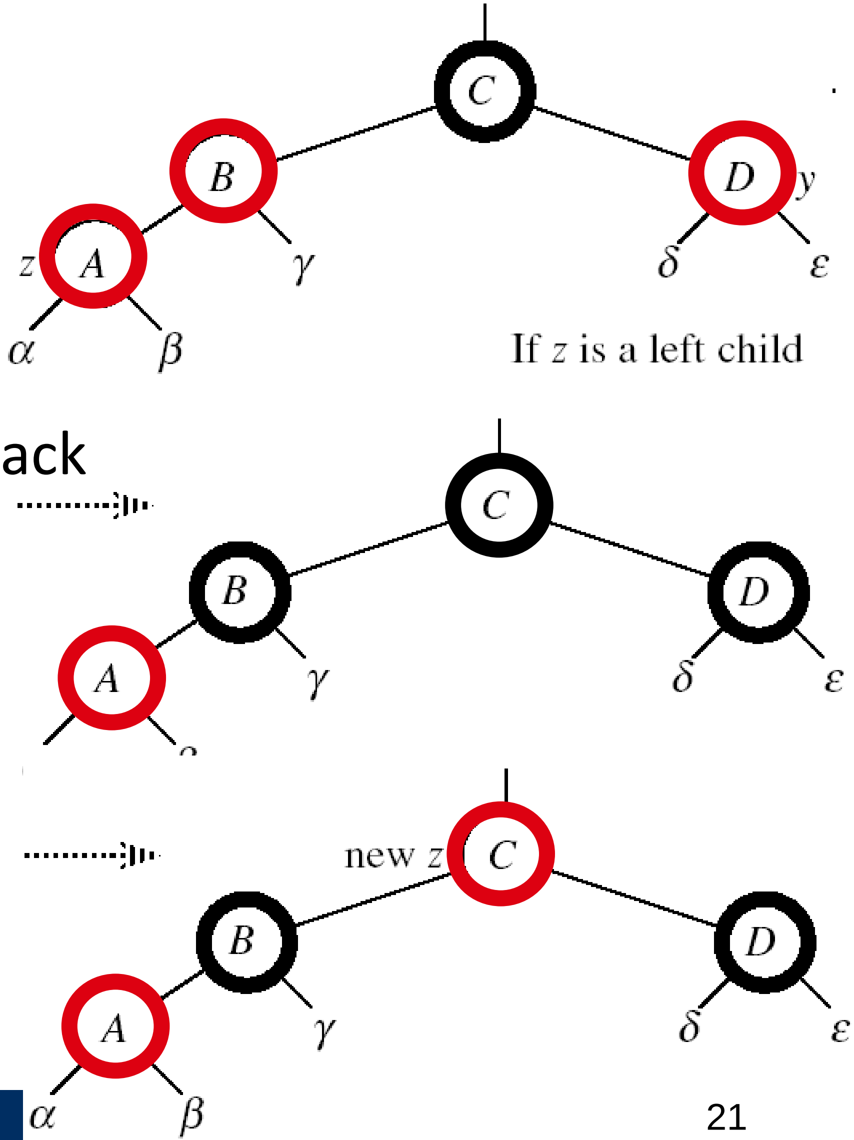
If p(z) is red ⇒ not OK
z and p(z) are both red

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**

(z could be either left or right child)

Idea:

- p[p[z]] (z's grandparent) must be black
- color p[z] ← **black**
- color y ← **black**
- color p[p[z]] ← **red**
- z = p[p[z]]
  - Push the **"red"** violation up the tree

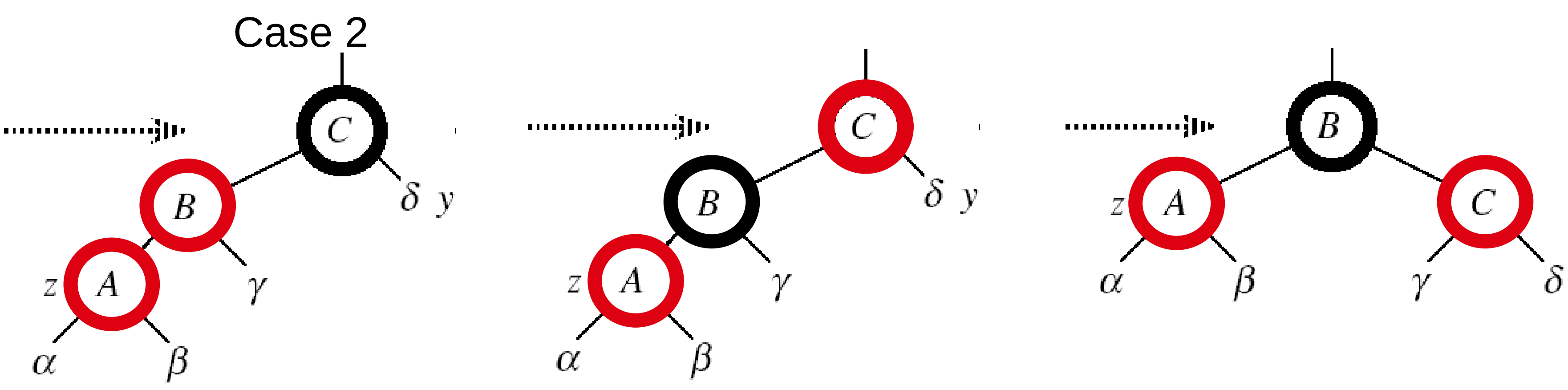
RB-INSERT-FIXUP

Case 2:

- 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

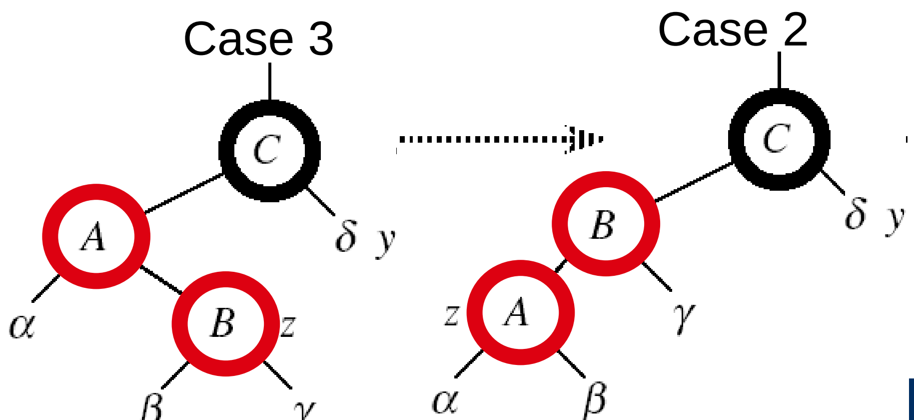
<u>Case 3:</u>

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

Idea:

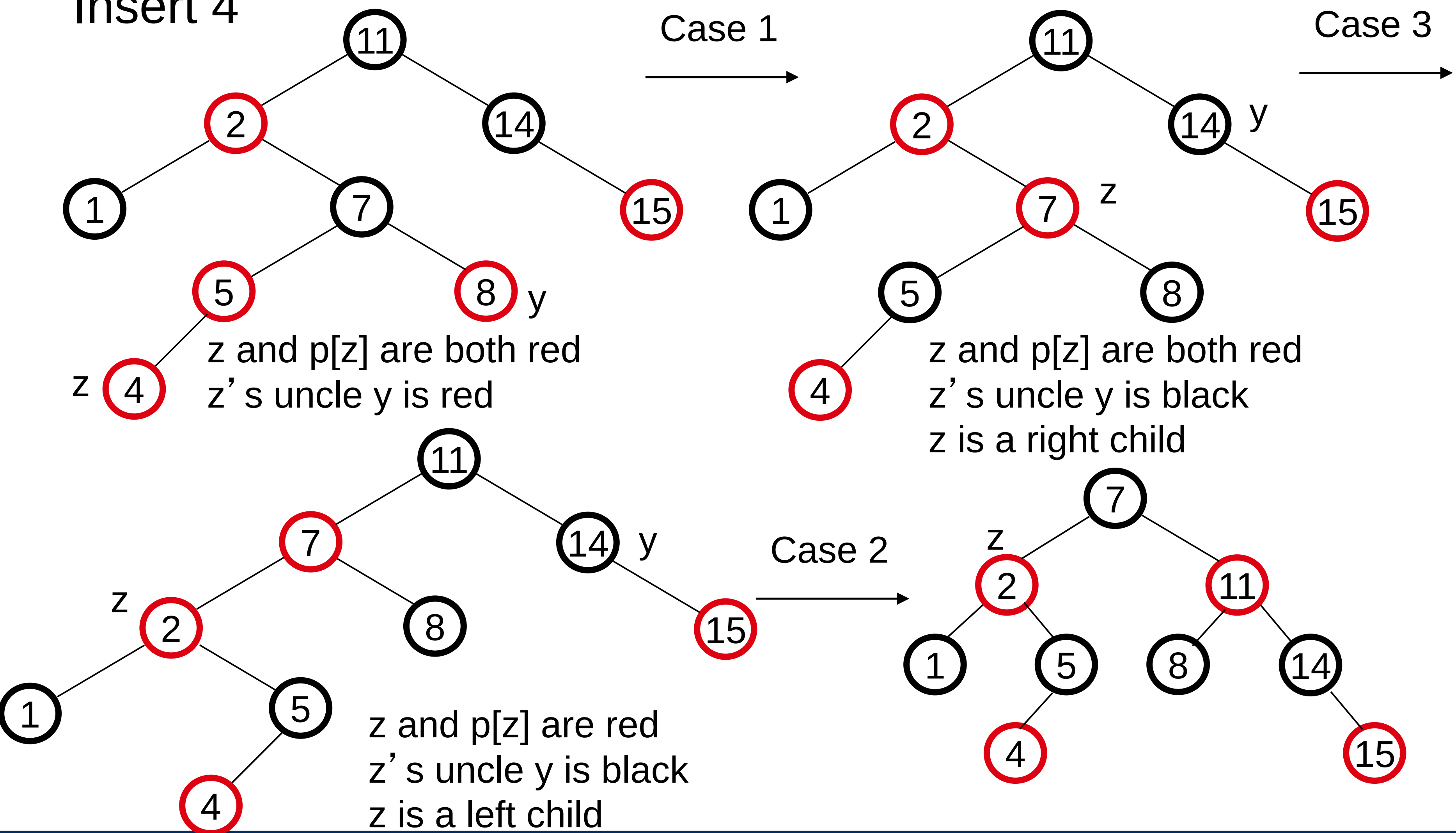
- z $\leftarrow$p[z]
- LEFT-ROTATE(T, z)

$\Rightarrow$ now z is a left child, and both z and p[z] are red $\Rightarrow$ case 2



Example

Insert 4



RB-INSERT-FIXUP(T, z)

```
1.  while color[p[z]] = RED
2.        if p[z] = left[p[p[z]]]
3.            then y ← right[p[p[z]]]
4.                 if color[y] = RED
5.                    then Case1
6.                    else if z = right[p[z]]
7.                            then Case3
8.                         Case2
9.        else (same as then clause with "right" and "left"
             exchanged for lines 3-4)
10. color[root[T]] ← BLACK
```

The while loop repeats only when case1 is executed: $O(\log N)$ times

Set the value of x' s "uncle"

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

Analysis of InsertItem

- Inserting the new element into the tree $O(\log N)$

- RB-INSERT-FIXUP
  - The while loop repeats only if CASE 1 is executed
  - The number of times the while loop can be executed is $O(\log N)$

- Total running time of Insert Item: $O(\log N)$

Delete Item

- Delete as usually, then re-color/rotate
- A bit more complicated though …
- Demo
  - http://gauss.ececs.uc.edu/RedBlack/redblack.html

Discussion 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)

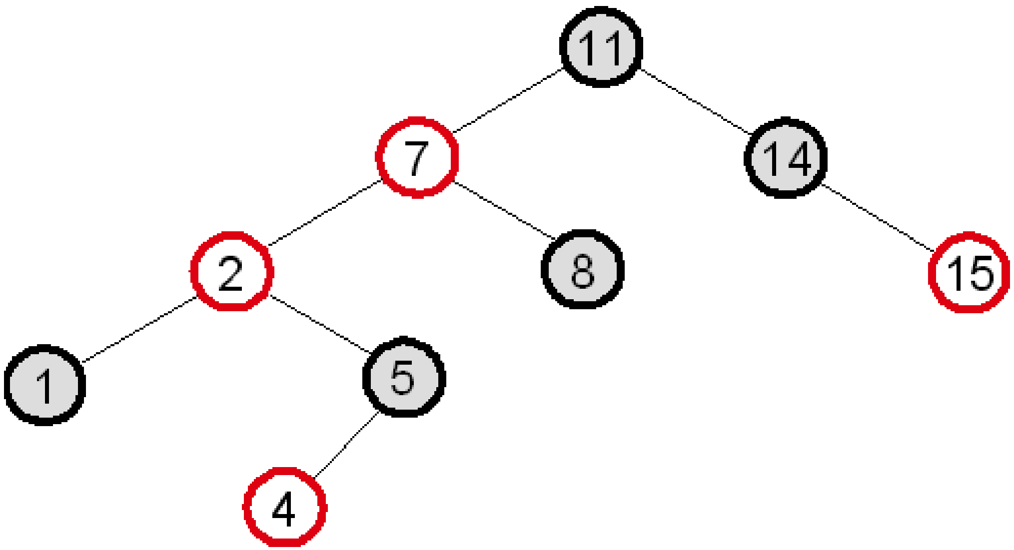
  - The longest path is equal to h(root)

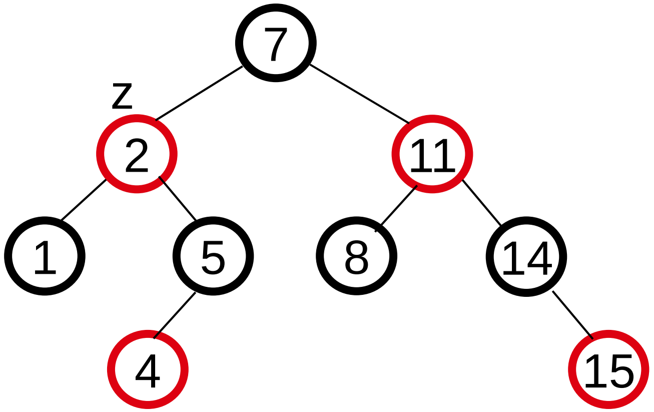
  - From Claim 1, $bh(root) \geq h(root)/2$
    or $h(root) \leq 2\, bh(root)$

  - Therefore, the ratio is ≤ 2

Discussion Problems

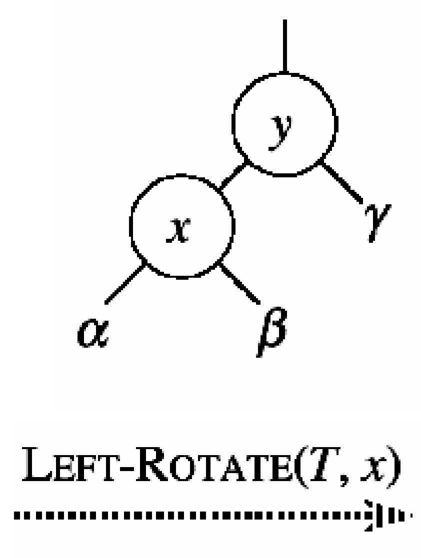
- What red-black tree property is violated in the tree below? How would you restore the red-black tree property in this case?
  - **Property violated**: if a node is red, both its children are black
  - **Fixup**: color 7 black, 11 red, then right-rotate around 11

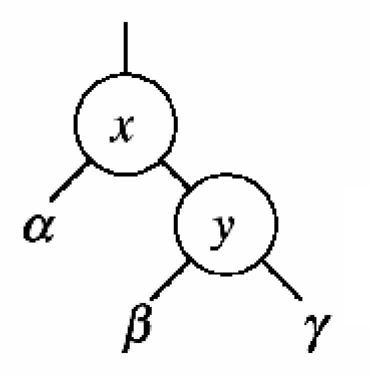




Discussion Problems

- Let a, b, c be arbitrary nodes in subtrees α, β, γ in the tree below.
- How do the depths of a, b, c change when a left rotation is performed on node x?
  - a: increases by 1
  - b: stays the same
  - c: decreases by 1





Discussion Problems

- When we insert a node into a red-black tree, we initially set the color of the new node to red.

  Why didn't we choose to set the color to black?

- Would inserting a new node to a red-black tree and then immediately deleting it, change the tree?