ICSI 403
DESIGN AND ANALYSIS OF ALGORITHMS

Lecture 11 – More Height-Balanced Trees:
Red-Black Trees (Inserting)

J Marques de Carvalho

Review

- In BSTs (Binary Search Trees), the dynamic set operators (INSERT, DELETE, SEARCH, PREDECESSOR, SUCCESSOR, MAX, and MIN) all (can) take $O(h)$ time, where h is the height of the tree.
- In a height balanced tree, $h \approx lg\ N$
- In a degenerate tree, $h = N$
- We need to have a way to keep the tree balanced as we perform repeated insertions and deletions.

A New Kind of Balanced Tree

- Red-Black Trees (1972 – Rudolph Bayer)
  - It's still a BST, and we keep it "mostly balanced"
  - Doesn't use "Balance Factor" like AVL
  - Instead, uses a single bit for "color" ("red" or "black") and some very special rules.
    - By constraining the way nodes can be colored on any path from the root to a leaf, red-black trees (RBT's) ensure that no such path is more than twice as long as any other, so that the tree is always ***approximately*** balanced.

Red-Black Trees (RBT's)

- Slightly different tree structure
- In the Huffman tree, data (characters) were only meaningful at a leaf.
- We're used to storing data in ANY node in a BST, and the only thing different about a leaf node was that it happened to not have any children (two NULL pointers).
- In an RBT, the leaves are ***always*** empty – *our data resides in the internal nodes*. Furthermore, all leaves are the SAME empty node, which Cormen calls $T.NIL$. The root's parent is also $T.NIL$

Red-Black Trees (RBT's)

- The "Ground Rules":
  1) Every node is colored either "Red" or "Black"
  2) The root is black
  3) Every leaf, $T.NIL$, is always black
  4) If a node is red, then both of its children are black (hence, no two consecutive red nodes on a path from root to leaf)
  5) For EVERY node, the number of black nodes between that node and the leaves is the same

RBT Example

- Example:

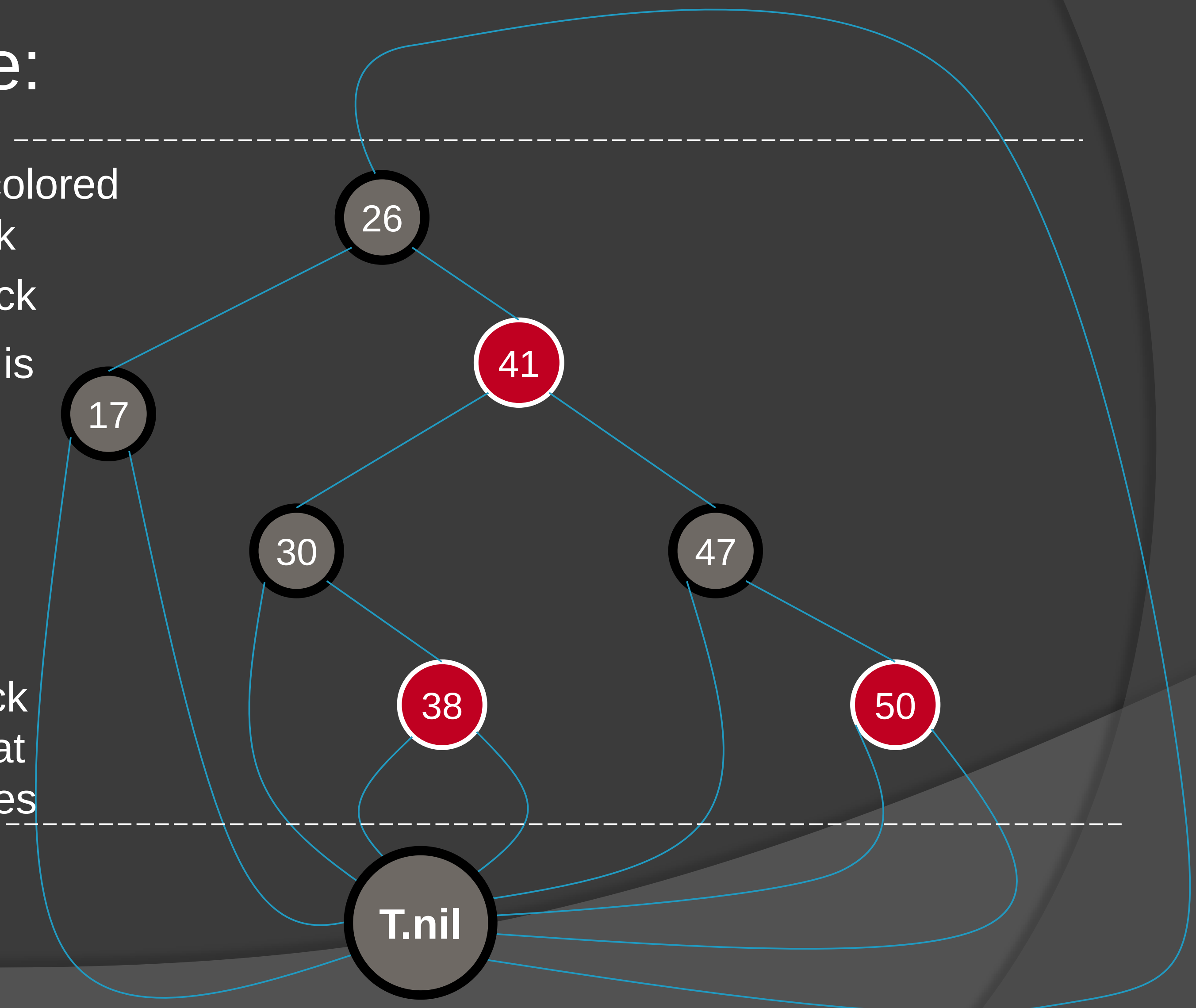
1) Every Node is colored either Red or Black

2) The Root is Black

3) Every leaf, *T.nil* is *always* black

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

5) For every node, the number of black nodes between that node and the leaves is the same

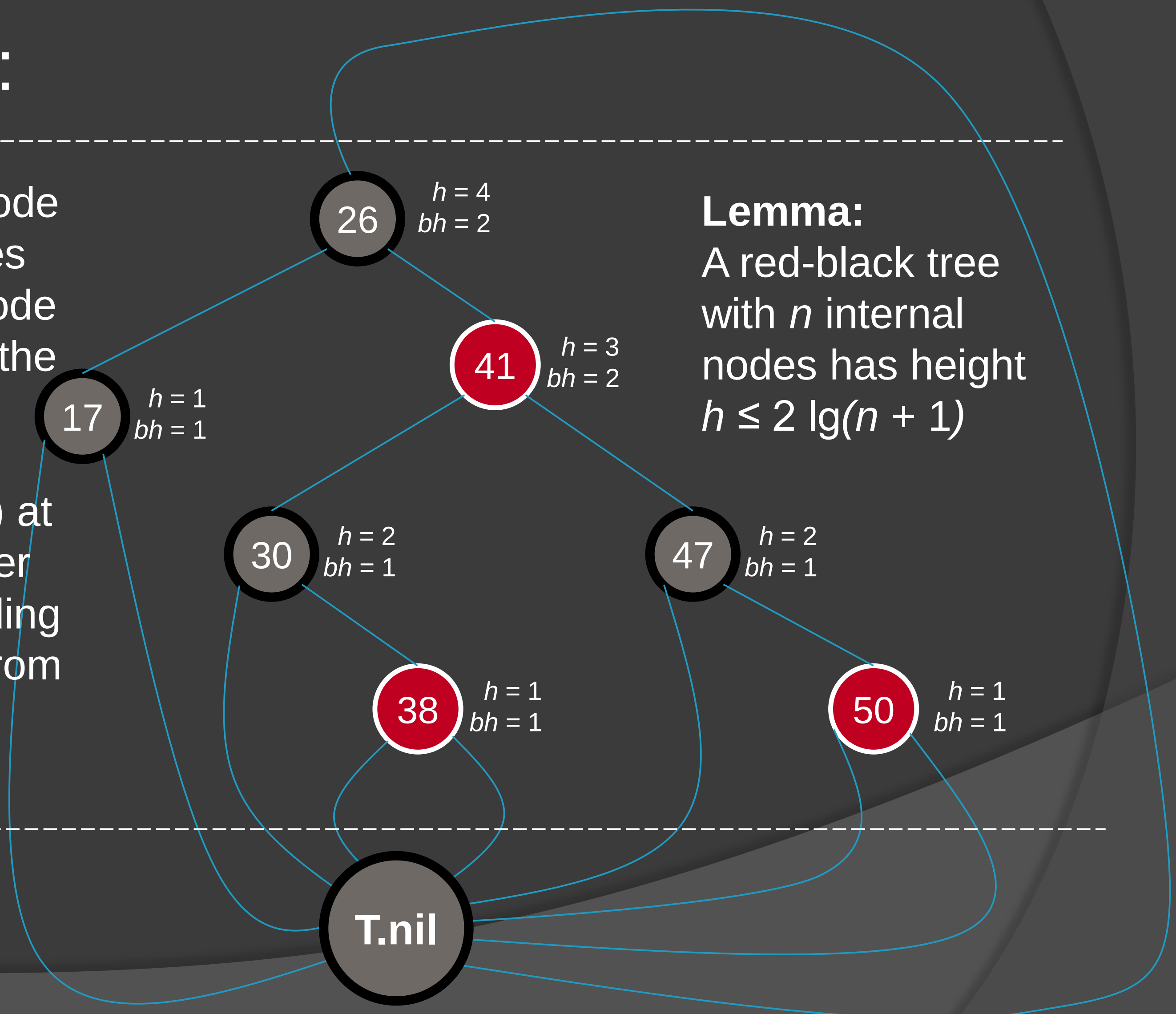


RBT Example

- Example:

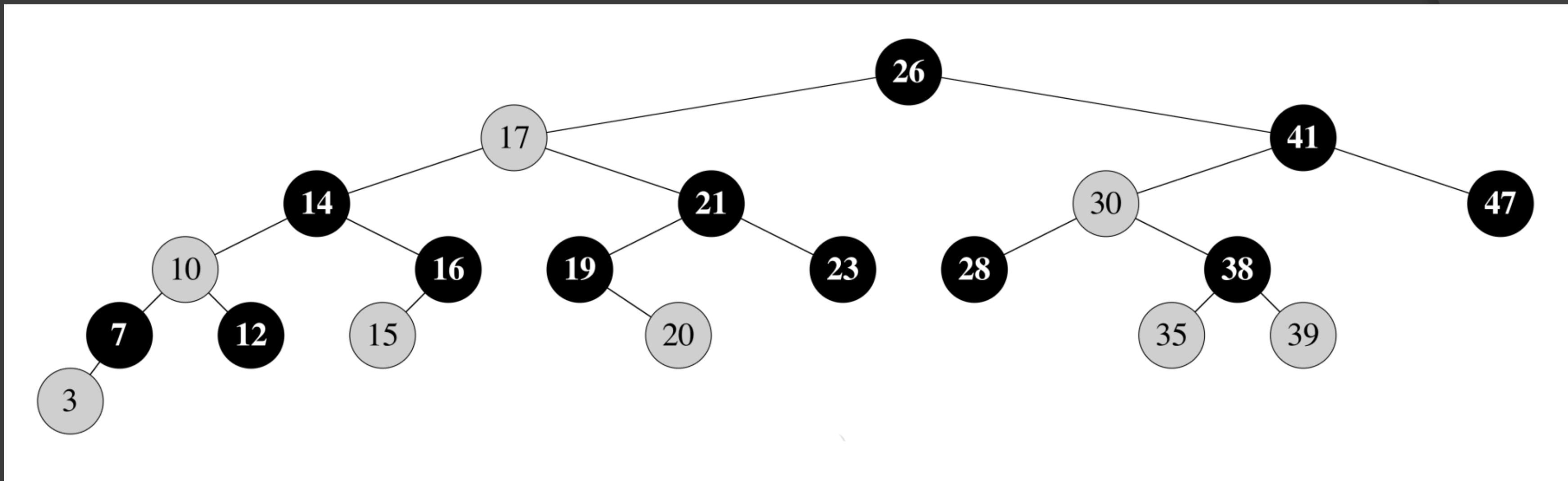
The *height* (h) at a node is the number of edges (links) between the node and the leaves along the longest path

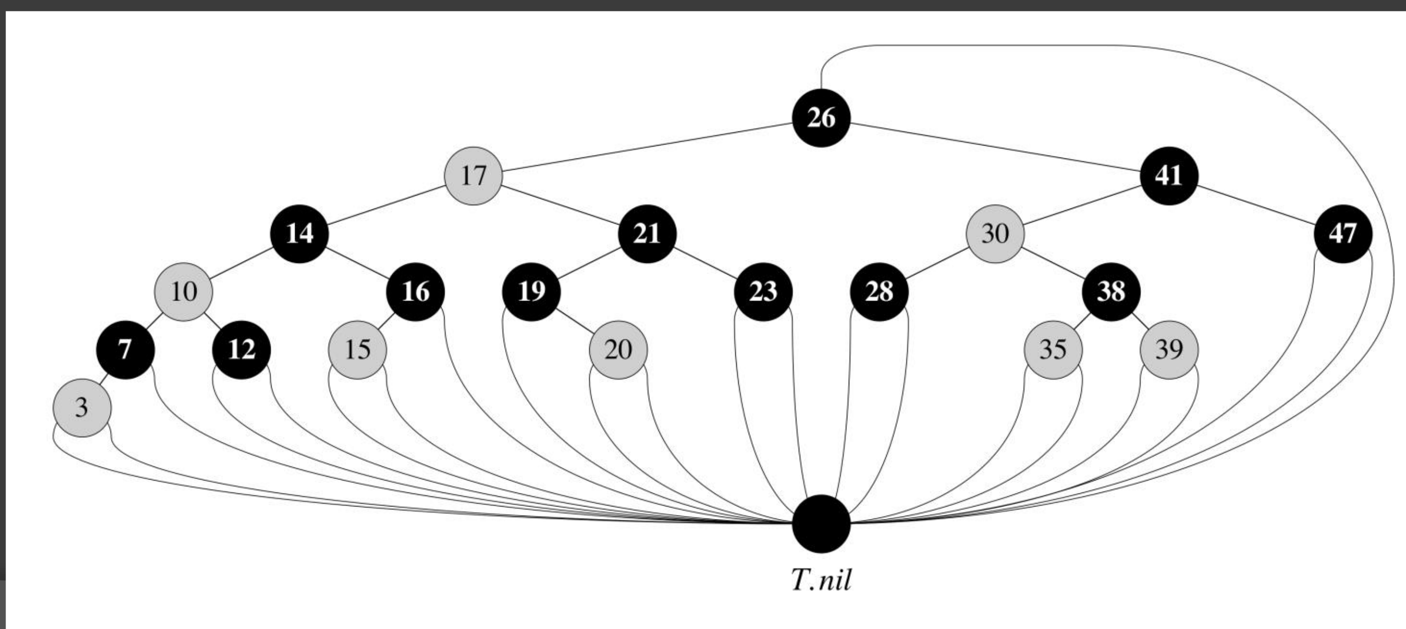
The *black height* (bh) at a node X is the number of black nodes (including $T.nil$) along the path from X to the leaves, not counting X.



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

Example From the Text (1)





Operations on RBTs

- SEARCH, MIN, MAX, SUCCESSOR, and PREDECESSOR work just like any other BST, and are all ***O(h)* → *O(lg n)***

- INSERT and DELETE – not so easy.

Operations on RBTs

- INSERT: What color to make the new node?
  - If we make it red, it might violate #4 (its parent might be red, and we can't have two successive red nodes)
    - 4) If a node is red, then ***both*** of its children are black
  - If we make it black, it might violate #5 (it might change the number of black nodes between some ancestor and the leaves)
    - 5) For every node, the number of black nodes between that node and the leaves is the same

Operations on RBTs

- DELETE: What color was the node we removed?
  - If Red, it's 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.
  - If 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.

RBT Operations

- The BST tree algorithms INSERT and DELETE run in $O(\lg n)$ time, but do not guarantee that the modified binary search tree will be a red-black tree.
  - They give NO balance-related guarantees
- To restore these properties, we must change the colors of some of the nodes in the tree and also change the pointer structure.
- We change the pointer structure through ***rotation***

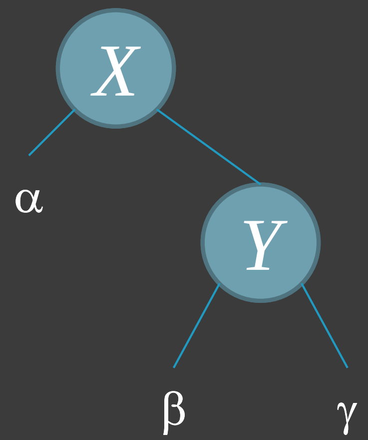
RBT Rotations

- *Rotation* is a local operation in a search tree that preserves the BST property (as in AVL).
- When we do a left rotation on a node X, we assume that its right child Y is not $T.nil$; X may be any node in the tree whose right child is not $T.nil$.
- The left rotation "pivots" around the link from X to Y. It makes Y the new root of the subtree, with X as Y's left child and Y's left child as X's right child.

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
- There are both left and right rotations. They are inverses of each other (symmetric)
- A rotation takes a red-black-tree and a node within the tree as arguments

Left-Right Rotation



$\alpha < X < \beta < Y < \gamma$ $\alpha < X < \beta < Y < \gamma$

Note: α stays the left child of X, and γ stays the right child of Y
Only X, Y, and β change their relative positions

Left-Rotate Pseudocode (p.313)

LEFT-ROTATE(T, x)

```
y = x.right                    // y is x's right child
x.right = y.left               //Turn y's left subtree into
if y.left != T.nil             // x's right subtree
  y.left.p = x
y.p = x.p                      // Link x's parent to y
if x.p == T.nil
    T.root = y
else if x == x.p.left
          x.p.left  = y
     else x.p.right = y
y.left = x                     // put x on y's left, which…
x.p = y                        // …makes x's parent be y
```

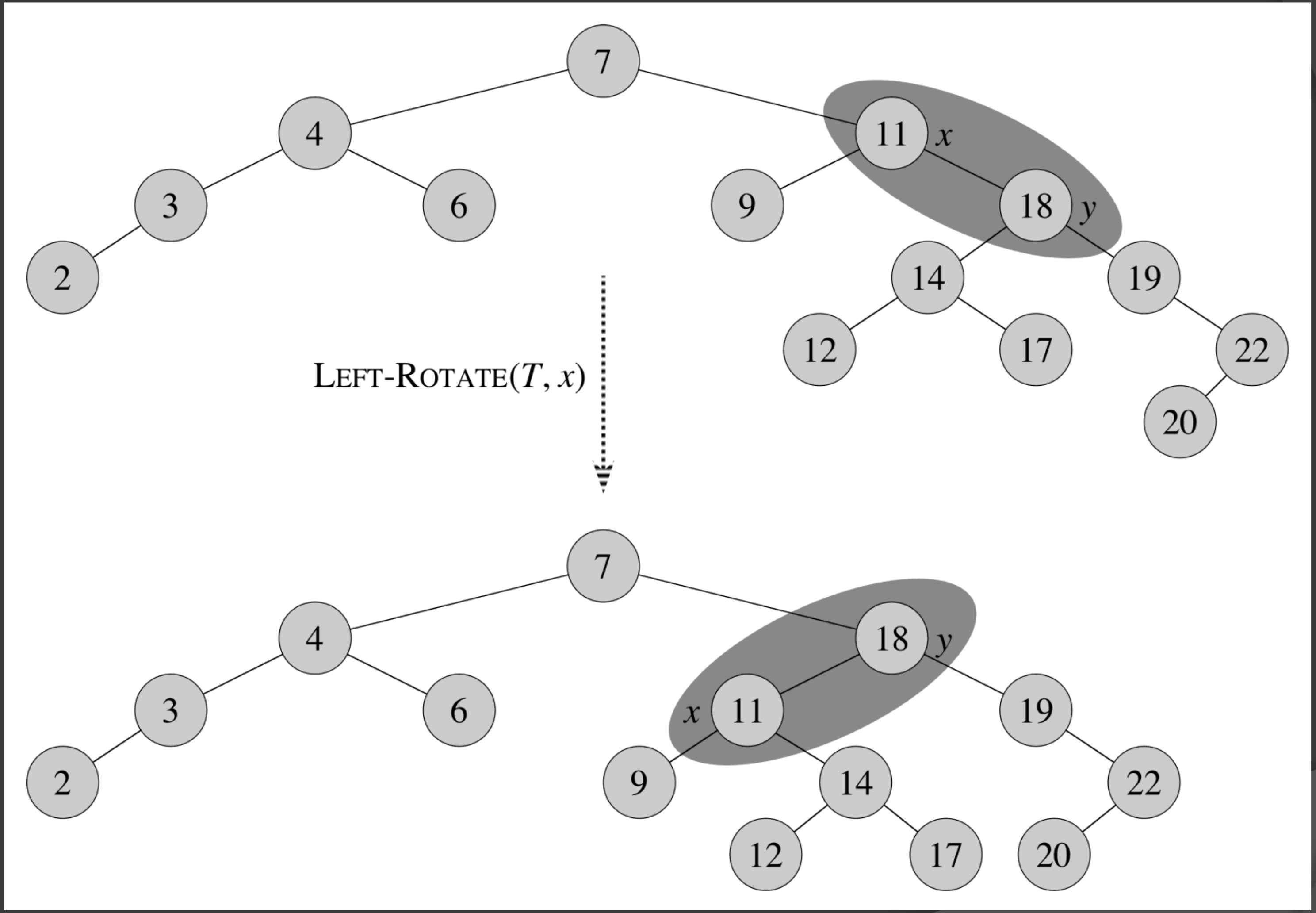
This all assumes T.root = T.nil and x.right != T.nil

Right-Rotate Pseudocode

RIGHT-ROTATE(T, x)

    // Everything is symmetric.
    // Exchange all occurrences of “left” and “right”

Left-Rotate Example



Rotation preserves the BST property

Inserting Into a RBT

- Same (high-level) approach as AVL Insert:
  - Search the tree to find where this node belongs
    - Just like any “regular” BST insert
  - If the tree is empty, then this node becomes the root
    - Just like any “regular” BST insert
  - Otherwise, make the new node the appropriate child of the appropriate node
    - Just like any “regular” BST insert
  - Clean up any problems the insertion created
    - UNLIKE “regular” BST insert, but like AVL

Inserting Into a RBT (p. 315)

```
RB-INSERT(T, z)          Insert node with key z into RBT T
  y = T.nil              x searches for insertion point
  x = T.root             y lags behind x (is its parent)
  while x != T.nil
     y = x
     if z.key < x.key x = x.left else x = x.right
  z.p = y
  if y == T.nil
     T.root = z
  else if z.key < y.key
            y.left = z
       else y.right = z
  z.left = T.nil         Our new node is at the bottom …
  z.right = T.nil        .. of the tree, so its children are T.nil
  z.color = RED              Insert this node as a red one
  RBT-INSERT-FIXUP(T, z)     Fix anything we broke
```

What Might We Have Messed Up?

- Which property might z's insertion violate?

1. *Every node is either red or black*. OK.
2. *The root is black*. If z is the root, then there's a violation. Otherwise, OK.
3. *Every leaf ($T.nil$) is black*. OK.
4. *If a node is red, then both its children are black.*

   If $z.p$ is red, there's a violation, because both z and $z.p$ are red.
5. *For each node, all paths from the node to descendant leaves contain the same number of black nodes.* OK.

What Might We Have Messed Up?

- Notation for the next couple of slides:
  - $z.p$ is z's <u>*parent*</u>; $z.p.p$ is z's <u>*grandparent*</u>
  - y points to the "right uncle" of z (the right child of z's grandparent)
  - If there is a violation of the red-black properties, there is at most one violation, and it is of either property 2 or property 4.
    - If there is a violation of #2, it occurs because z is the root and is red.
    - If there is a violation of #4, it occurs because both z and $p.z$ are red.

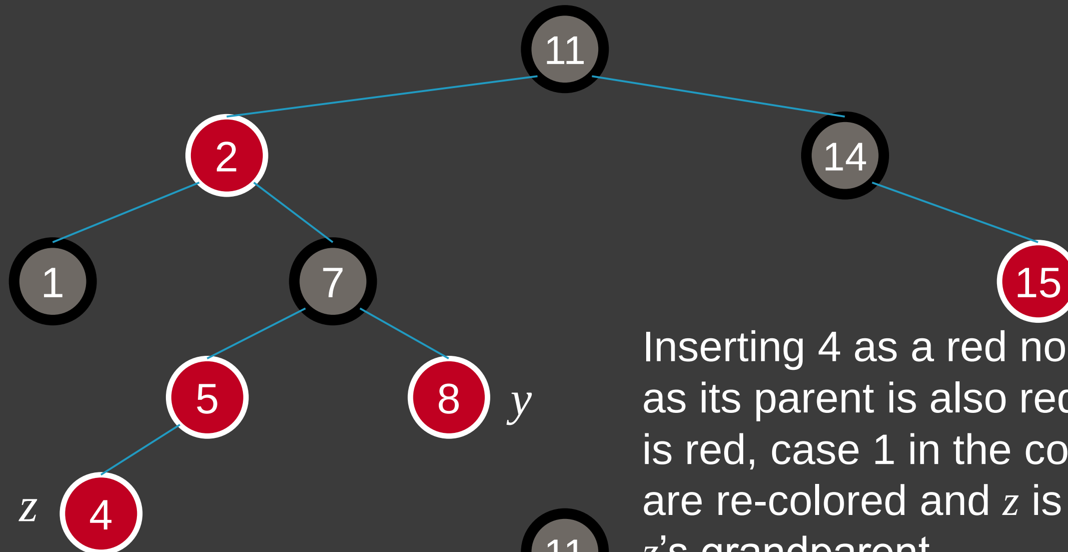
What Might We Have Messed Up?

- There are three cases (1, 2, 3)
- Fixing one problem may create another,
- If it does, however, it will not create any more than one problem (zero or one).
- Eventually, we will solve a problem and not create a new one. At this point, we have re-balanced the tree.

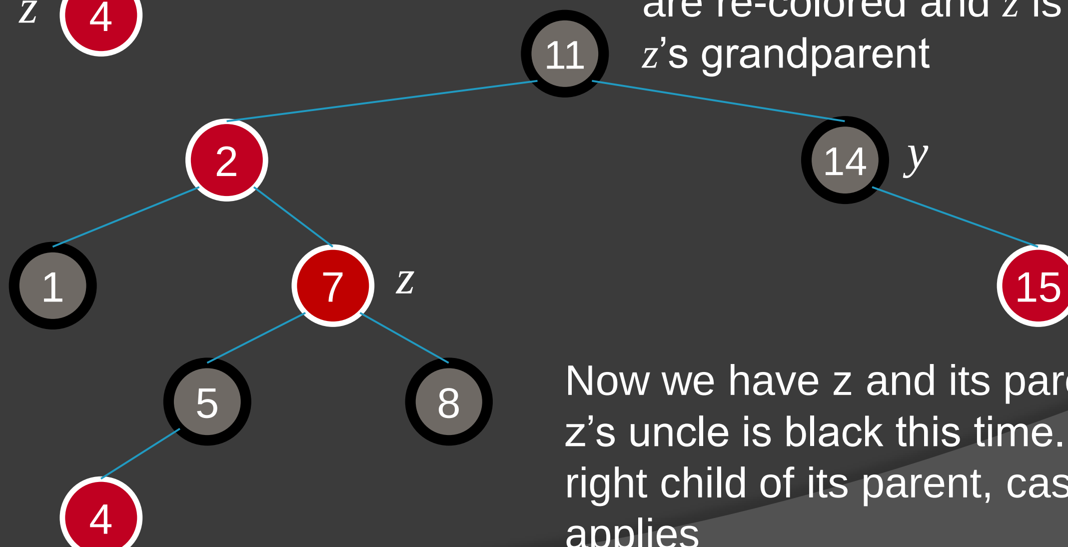
The Fixup Code (p. 316)

```
RBT-INSERT-FIXUP (T, z)
while z.p.color == red
    if z.p == z.p.p.left then
        y = z.p.p.right
        if y.color == RED then
            z.p.color = BLACK               Case 1
            y.color = BLACK                 Case 1
            z.p.p.color = RED               Case 1
            z = z.p.p                       Case 1
        else if z == z.p.right
                z = z.p                         Case 2
                LEFT-ROTATE (T, z)              Case 2
            z.p.color =  BLACK                      Case 3
            z.p.p.color = RED                       Case 3
            RIGHT-ROTATE (T, z.p.p)                 Case 3
    else  < “else” clause symmetric to “then”.  Swap “left” and “right” >
T.root.color = BLACK
```

Case 1

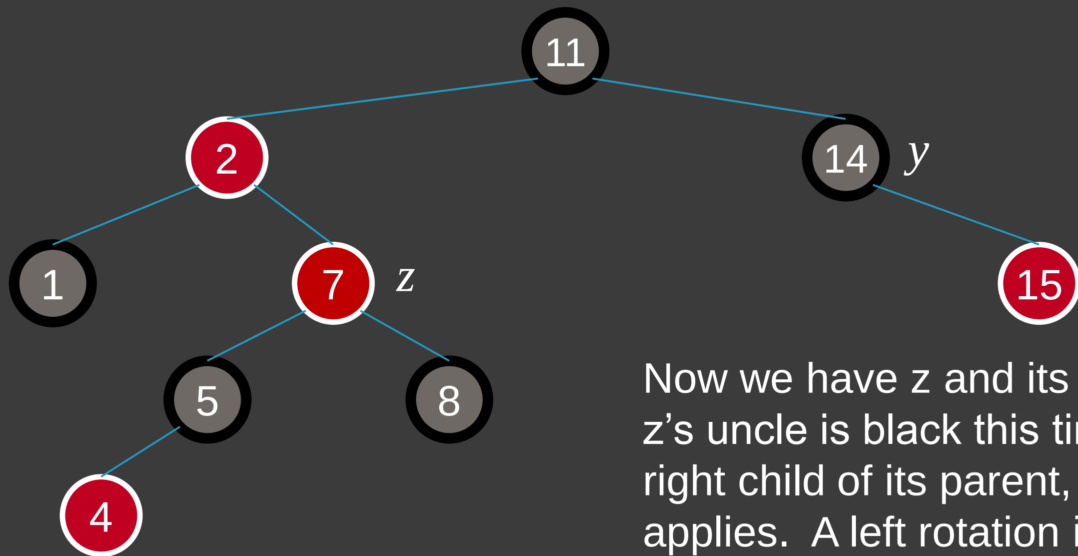


Inserting 4 as a red node (*z*) violates rule 4, as its parent is also red. Since *z*'s uncle (*y*) is red, case 1 in the code applies. Nodes are re-colored and *z* is moved up to point to *z*'s grandparent

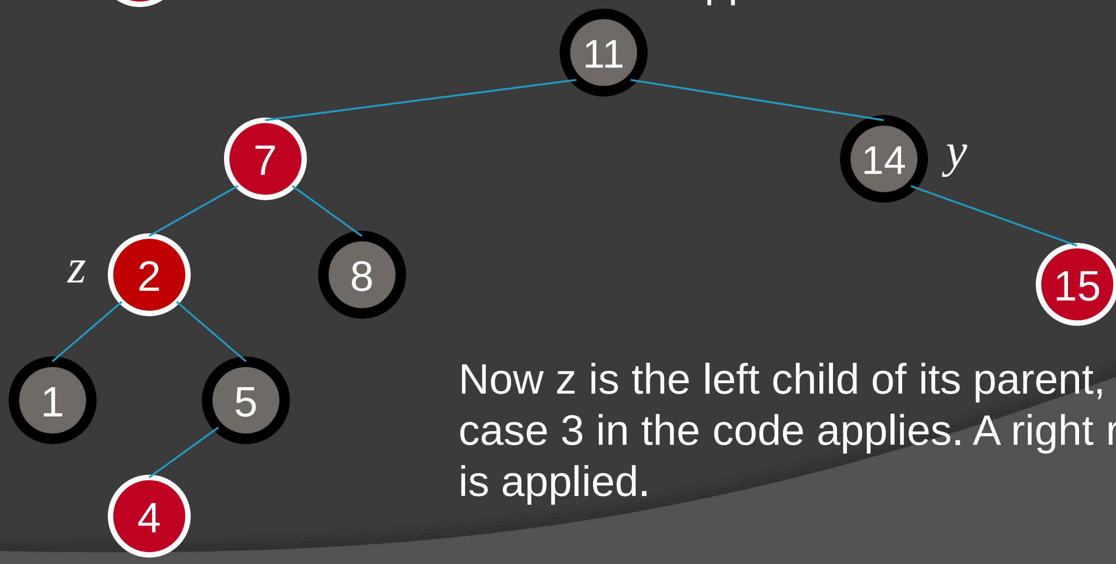


Now we have z and its parent both red, but z's uncle is black this time. Since z is the right child of its parent, case 2 in the code applies

Case 2

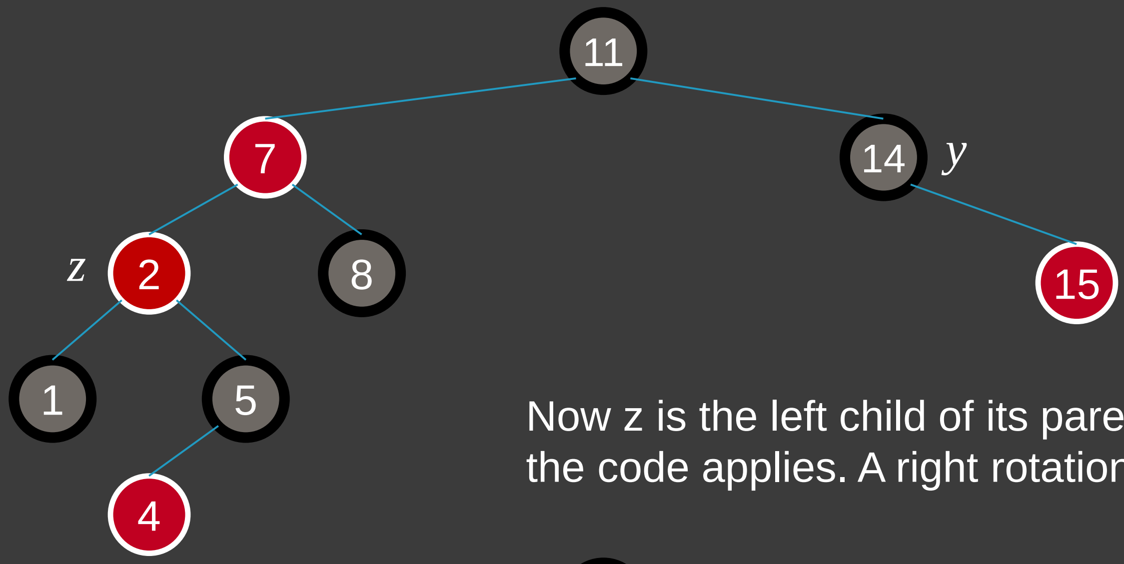


Now we have z and its parent both red, but z's uncle is black this time. Since z is the right child of its parent, case 2 in the code applies. A left rotation is applied

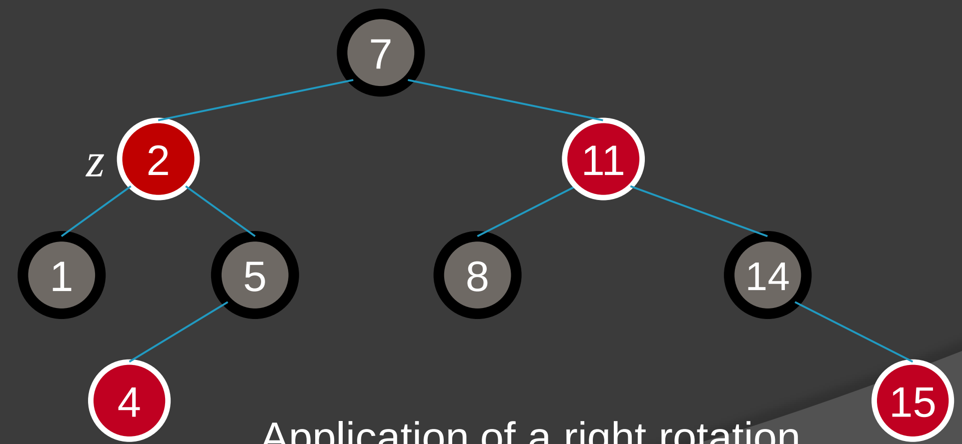


Now z is the left child of its parent, and case 3 in the code applies. A right rotation is applied.

Case 3



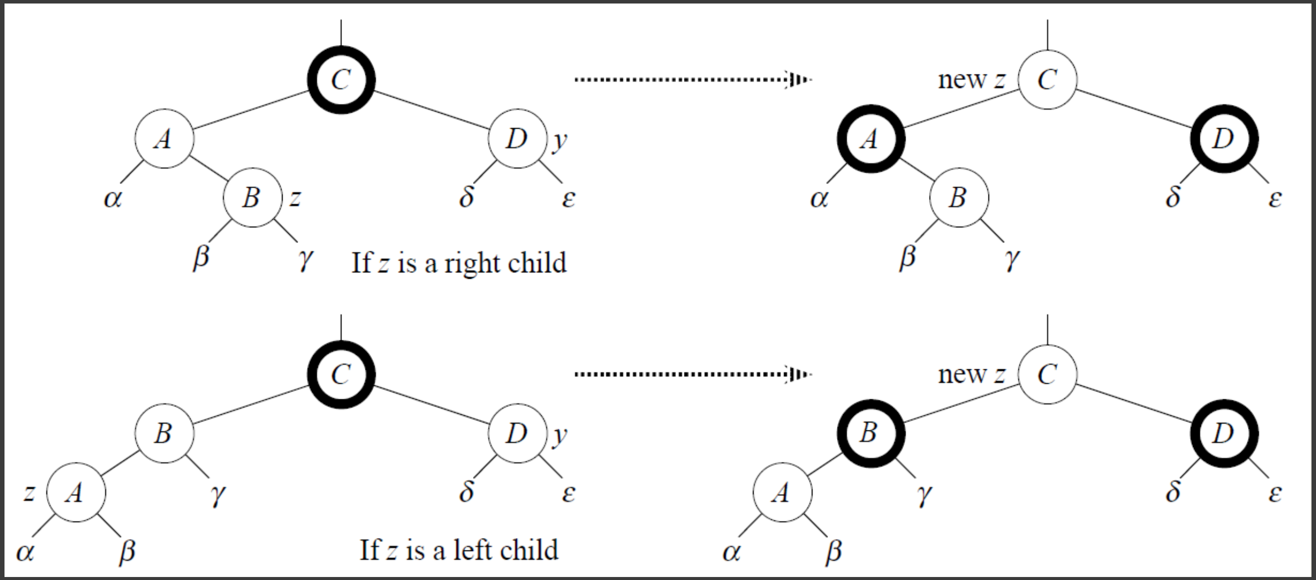
Now z is the left child of its parent, and case 3 in the code applies. A right rotation is applied.



Application of a right rotation results in a legal RBT

The Three Cases – More Detail

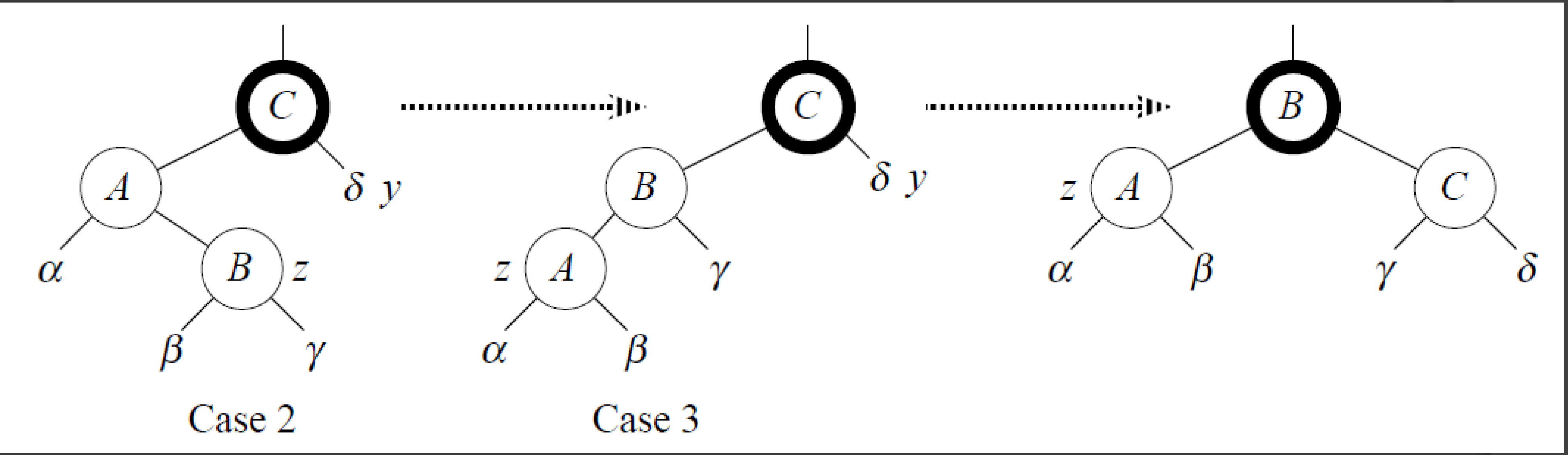
- Case 1: z is red, $z.p$ is red, & z's uncle y is red



- Since $z.p.p$ is black, we can color both $z.p$ and y black (fixing the problem of both z and $z.p$ being red), and color $z.p.p$ red (satisfying rule #5). Move z two levels up the tree, and go through the `while` loop again

The Three Cases – More Detail

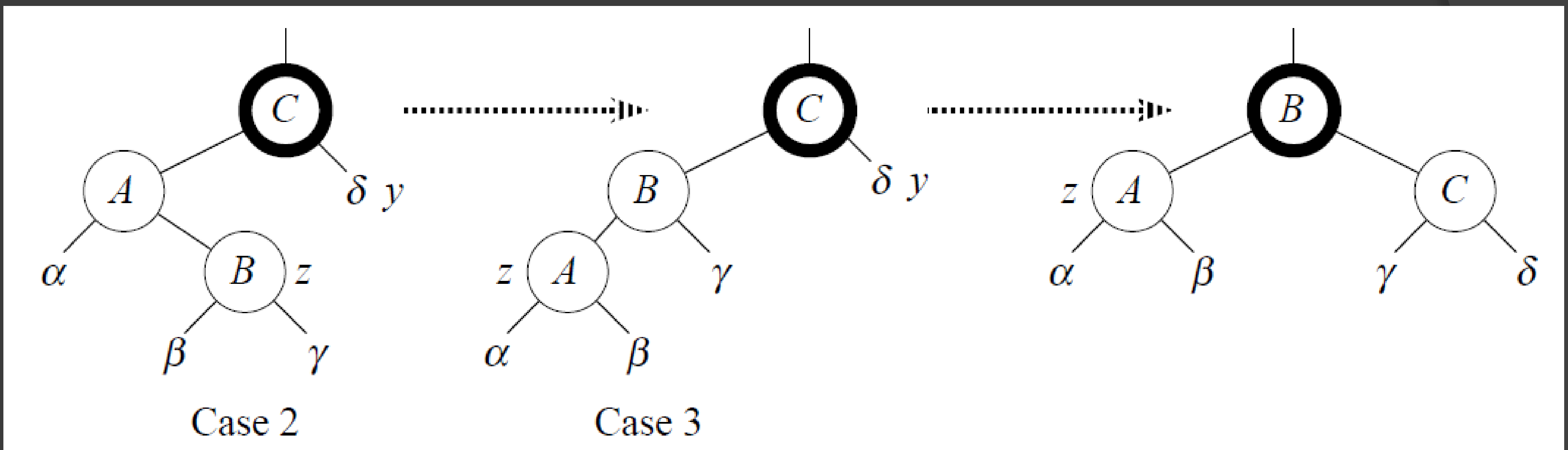
- Case 2: z's uncle y is black, z is a right child



- Left rotate around $z.p$. Now z is a left child, and both z and $z.p$ are red. Takes us immediately to case 3.

The Three Cases – More Detail

- Case 3: z's uncle y is black, z is a left child



- Make $z.p$ black and $z.p.p$ red
- Then right rotate on $z.p.p$
- No longer have 2 reds in a row
- $z.p$ is now black → no more iterations

RBT Insertion Fixup - Summary

- RBT Insertion – The Three cases:
  - 1: z is red, $z.p$ is red, & z's uncle y is red
    - Re-color z, z's parent, and z's uncle (no rotation)
  - 2: z's uncle y is black, z is a right child
    - Do a left rotation around z. Takes us immediately to
  - 3: z's uncle y is black, z is a left child
    - Recolor z's parent and grandparent
    - Do a right rotation around z's grandparent
  - Move z up two levels ($z \leftarrow z$'s parent's parent), and try again as long as z's parent is red

RBT Analysis – Insert Operation

- $O(\lg n)$ time to get through RBT-INSERT up to the call of RBT-INSERT-FIXUP
- Within RBT-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. $O(\lg n)$ time.
  - Also note that there are at most 2 rotations overall.
- Thus, insertion into a red-black tree takes $O(\lg n)$ time

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 $O(\lg n)$

Next Time

- RBT Deletion (pp. 323 ff.)
- Similar to insertion:
  - Start with “regular” BST deletion
  - Then run a routine to repair anything the deletion did to violate the RBT rules.
    - THAT Fix-up has 4 cases.

? Questions ?

? Questions ?