CS 315 - Day 26, Red-Black Trees 2

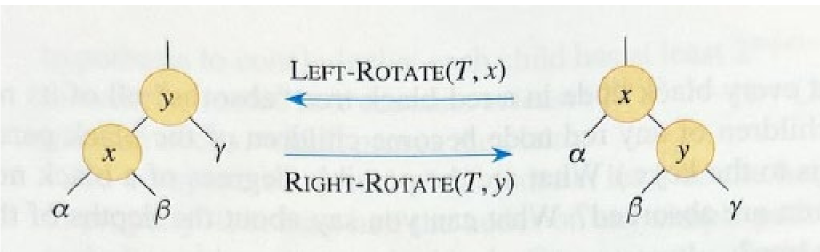
Previously we introduced RB Trees, with the goal being to balance the height of a BST Tree, so that $h = O(\lg N)$. There are other balanced trees, we're just focusing on one type, RB Trees.

Red-Black 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**.
5. For each node, all simple paths from the node to descendant leaves contain the same number of **black** nodes.

We are also just going to focus on insertion. With this foundation, learning other trees or routines should be straightforward.

Our key helper is ***rotations***.



Start by doing regular binary-search-tree insertion:

```
RB-INSERT(T, z)
  x = T.root                 // node being compared with z
  y = T.nil                  // y will be parent of z
  while x ≠ T.nil            // descend until reaching the sentinel
      y = x
      if z.key < x.key
          x = x.left
      else x = x.right
  z.p = y                    // found the location—insert z with parent y
  if y == T.nil
      T.root = z             // tree T was empty
  elseif z.key < y.key
      y.left = z
  else y.right = z
  z.left = T.nil             // both of z's children are the sentinel
  z.right = T.nil
  z.color = RED              // the new node starts out red
  RB-INSERT-FIXUP(T, z)      // correct any violations of red-black properties
```

- RB-INSERT ends by coloring the new node z red.
- Then it calls RB-INSERT-FIXUP because it could have violated a red-black property.

Which property might be violated?

1. OK. (Every node is still either red or black.)
2. If z is the root, then there's a violation. (The root must be black.) Otherwise, OK.
3. OK. (All leaves are still black.)
4. If $z.p$ is red, there's a violation: both z and $z.p$ are red. (Not allowed to have two red nodes in a row.)
5. OK. (Adding a red node doesn't change any black-heights.)

Remove the violation by calling RB-INSERT-FIXUP:

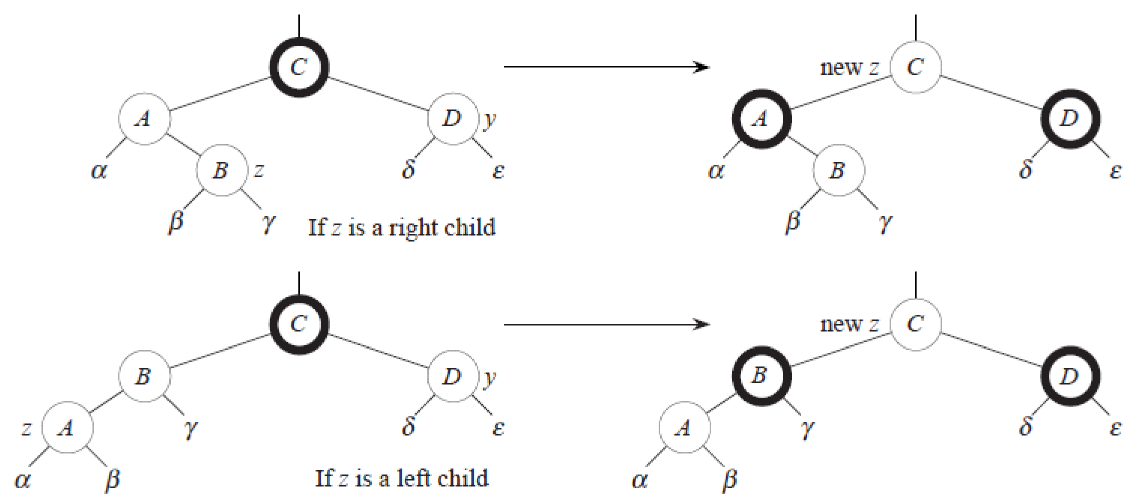
First, some notation:

```
            z.p.p
    z.p              z.p.p.right (uncle)
z
```

We will only encounter examples and problems where* `z.p` *is a <u>left</u> child.

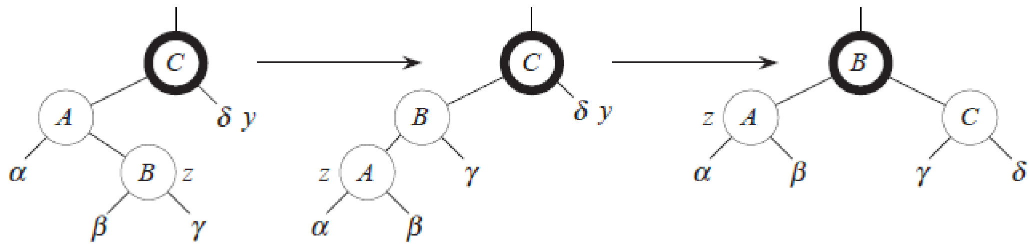
There are three cases for `RB-Insert-Fixup` to consider

Case 1: y is red



- $z.p.p$ (z's grandparent) must be black, since z and $z.p$ are both red and there are no other violations of property 4.
- Make $z.p$ and y black $\Rightarrow$ now z and $z.p$ are not both red. But property 5 might now be violated.
- Make $z.p.p$ red $\Rightarrow$ restores property 5.
- The next iteration has $z.p.p$ as the new z (i.e., z moves up 2 levels).

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



Case 2 Case 3

- Move z up one level (make $z.p$ the new z), then left rotate around the new z $\Rightarrow$ now z is a left child, and both z and $z.p$ are red.
- Falls through into case 3.

Case 3: 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 $\Rightarrow$ no more iterations.

Note how Case 2 turns right into Case 3.

```
RB-INSERT-FIXUP(T, z)
  while z.p.color == RED
      if z.p == z.p.p.left              // is z's parent a left child?
          y = z.p.p.right               // y is z's uncle
          if y.color == RED             // are z's parent and uncle both red?
              z.p.color = BLACK         ⎫
              y.color = BLACK           ⎬ case 1
              z.p.p.color = RED         ⎪
              z = z.p.p                 ⎭
          else
              if z == z.p.right
                  z = z.p               ⎫ case 2
                  LEFT-ROTATE(T, z)     ⎭
              z.p.color = BLACK         ⎫
              z.p.p.color = RED         ⎬ case 3
              RIGHT-ROTATE(T, z.p.p)    ⎭
      else (same as then part, but with "right" and "left" exchanged)
  T.root.color = BLACK
```

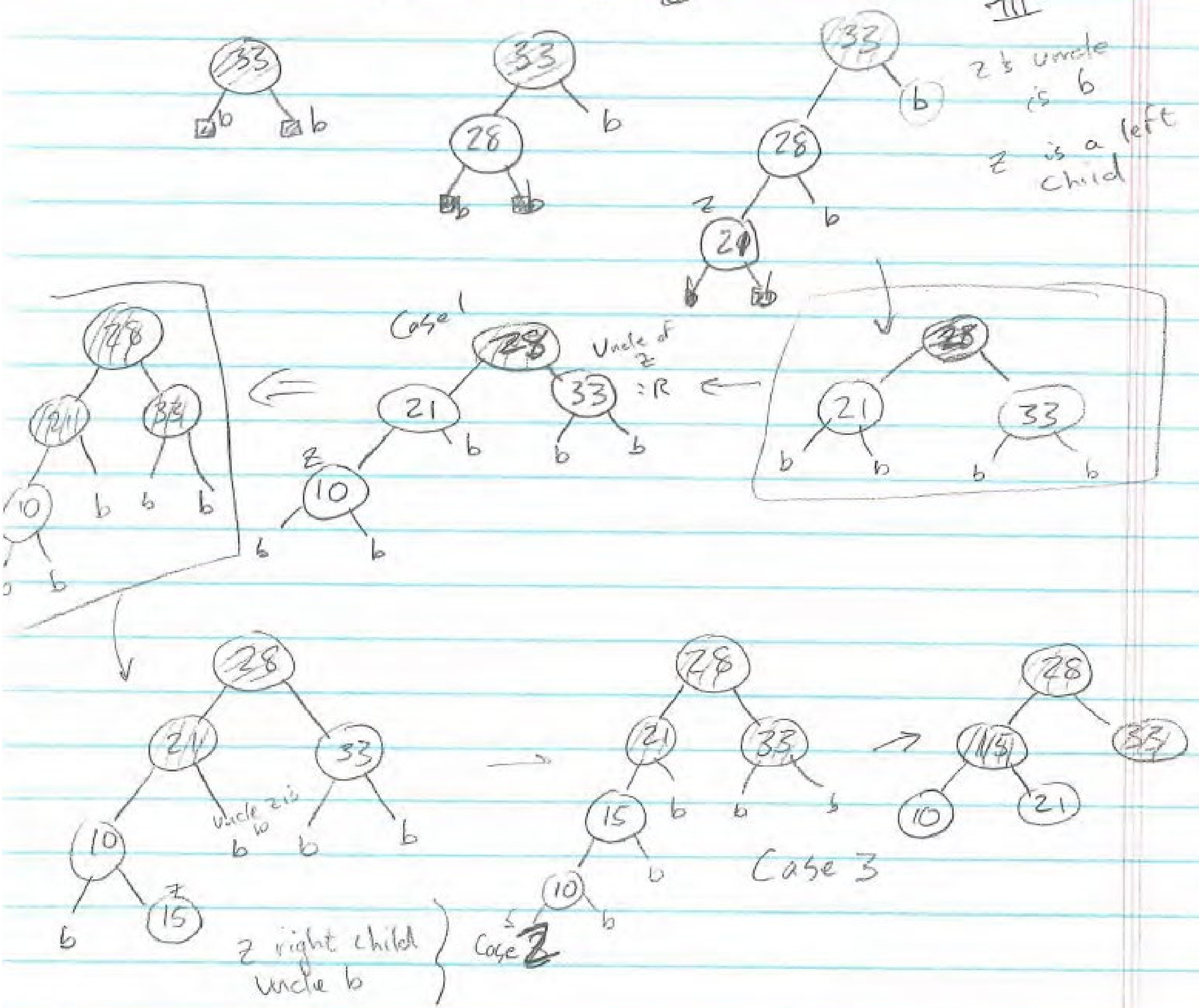
Figure 13.4 The operation of RB-INSERT-FIXUP. **(a)** A node z after insertion. Because both z and its parent $z.p$ are red, a violation of property 4 occurs. Since z's uncle y is red, case 1 in the code applies. Node z's grandparent $z.p.p$ must be black, and its blackness transfers down one level to z's parent and uncle. Once the pointer z moves up two levels in the tree, the tree shown in **(b)** results. Once again, z and its parent are both red, but this time z's uncle y is black. Since z is the right child of $z.p$, case 2 applies. Performing a left rotation results in the tree in **(c)**. Now z is the left child of its parent, and case 3 applies. Recoloring and right rotation yield the tree in **(d)**, which is a legal red-black tree.

<u>In-class w/***assigned groups of four (4)***, write 0. and 1. on board as ground rule(s):</u>

0. No devices.
1. Intro: Name, POE, Icebreaker.
2. Pick someone on the group who will report out.
3. Work through RB-Insert and `RB-Fixup` and insert the following nodes into an `RB-Tree` in the order given:

33, 28, 21, 10, 15

33, 28, 21, 10, 15


III
z's uncle is b
z is a left child
Case 1
Uncle of z = R
uncle z is b
z right child uncle b
Case 2
Case 3