Red-Black Trees

CptS 223 - Fall 2017 - Aaron Crandall

Today’s Agenda

- Announcements
- Humor of the day
- Red-Black Trees

Announcements

-

Thing of the day: First hoverbike is tested

Fun times!

Attendance!

bit.ly/2xBcl42

“Bee cee el”

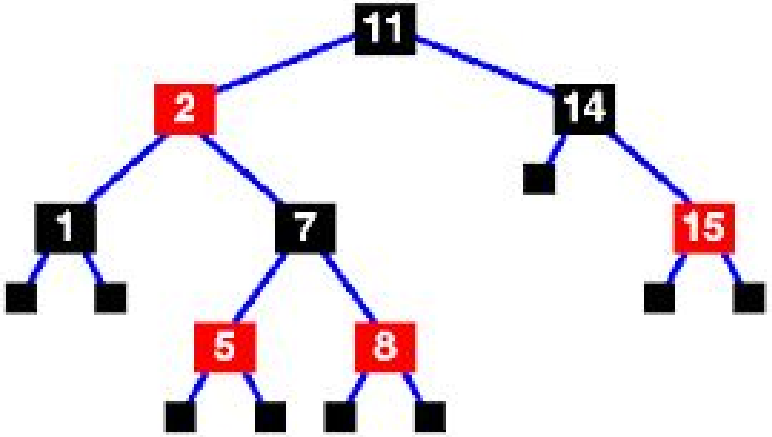
Final questions about:

- B-Trees
- Splay Trees
- AVL Trees
- BST Trees
- Trees Trees?

Red-Black Trees

- Operations take worst case O(log N) time
- Non-recursive inserts are possible (reduces stack overhead)
- It has some interesting properties that generate the kinds of behavior we want. It’s not as obvious why at first.
- Unlike Splay Trees, I definitely see Red-Black trees in use
    - They show up in the C++ STL in at least one place (the SET type)

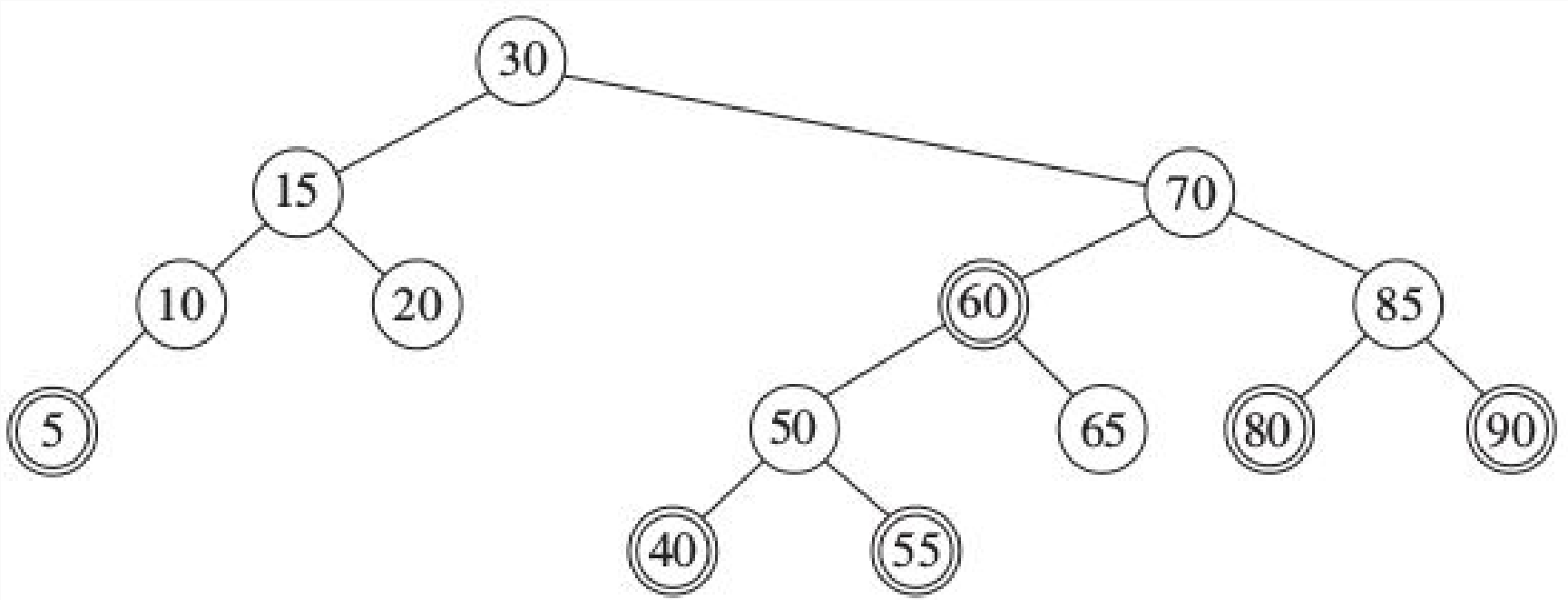
Basic Red-Black Tree Properties



1) Every node is colored either red or black.
2) The root is black.
3) If a node is red, its children must be black.
4) Every path from a node to a null pointer must contain the same number of black nodes.

This creates a height of at most 2 * log(N+1)
This is vs. AVL trees' 1.44 * log(N+2) - 0.328 average height

An example of this tree:
(double circles are red)



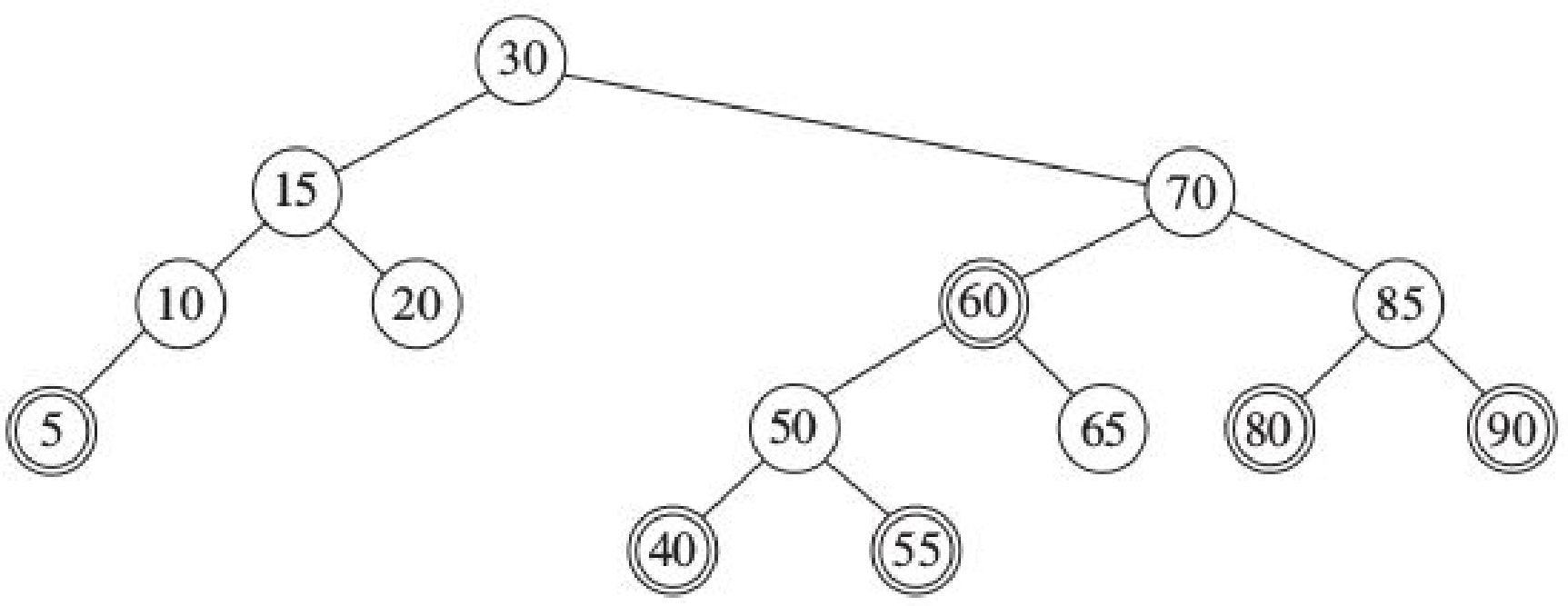
Tree doesn’t keep balance or height info

- The only information added to a BST is the color of a node
- No height or balance information is kept
- This isn’t a tree that enforces balance like an AVL tree, but it ends up behaving in a way that’s just as good (almost as good)

Insert - Always the big question

- Inserted at leaf location, per a BST
- If it’s colored black, then it violates condition #4
  - So, it’s colored red
  - If parent is black, then you’re done
  - If parent is red… it violates condition #3, and we need to adjust the tree
- Basic operations are color changes and tree rotations
- FYI: Null nodes are always considered to be black

Easy case: insert 25



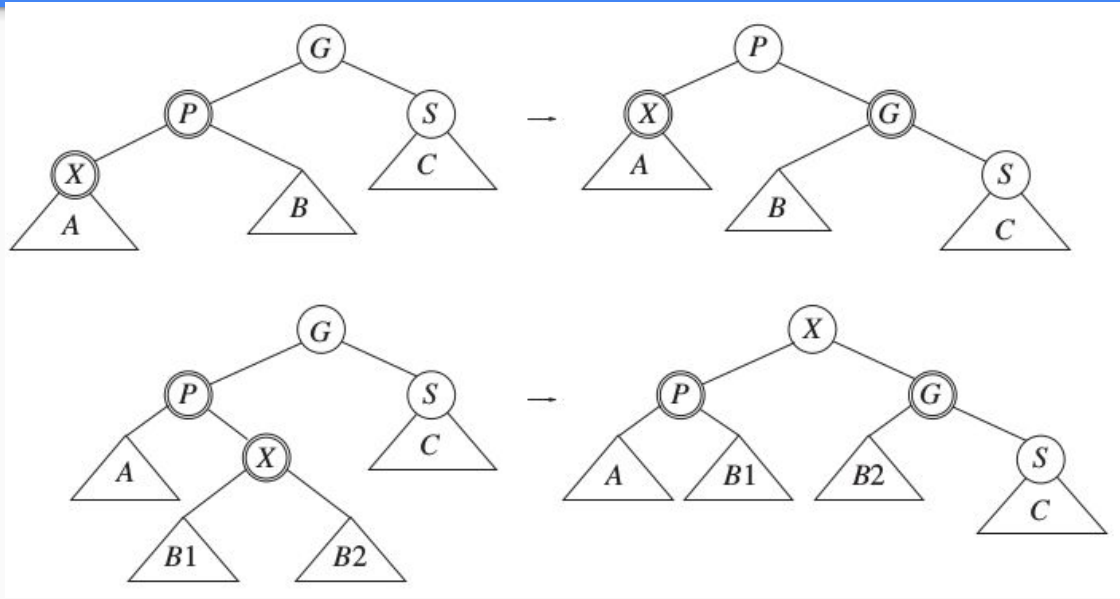
Other cases of bottom-up insertion

Case #1: Parent is red.

If: sibling of parent is black (see insert 3 or 8):

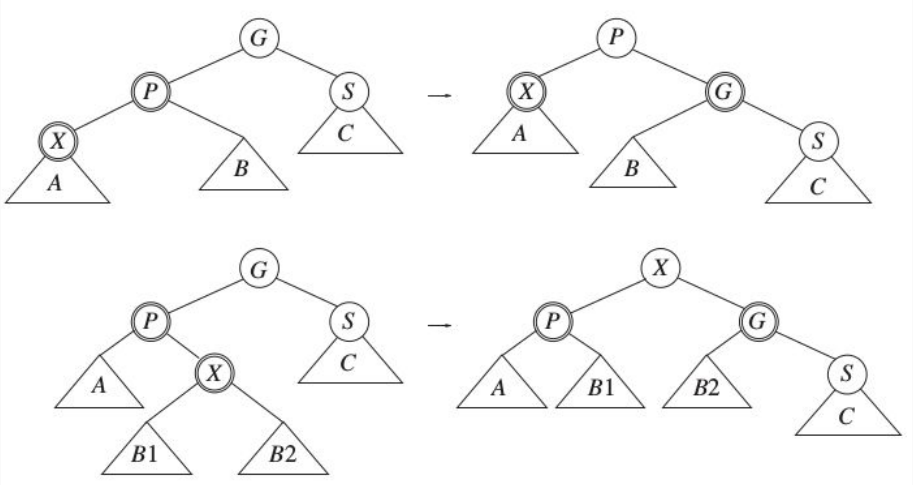
either zig-zag or zig-zig

X is new added leaf
P is parent
S is sibling to parent (aunt)
G is grandparent



Note the changes to the tree: coloring

- When the zig-zag and zig-zig operations are performed, the nodes at the end need their colors fixed
- This will preserve the rules about red nodes having no red children and the number of black nodes higher up in the tree

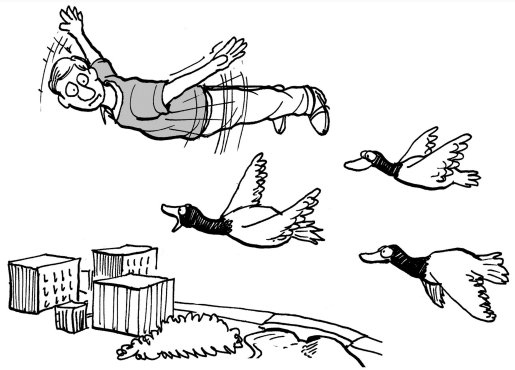


What happens if the S (aunt) node is red?

- Do the same thing, but color:
  - S red
  - New root red
  - G black
- If the great-grandparent is also red… we can’t have two red nodes in a row (root and great-grandparent), so you recursively percolate the rotations up the tree until you no longer have two consecutive red nodes or we reach the root.
- Yeah, that’s complex.

An alternative to bottom-up insertion!

- Enter the Top-Down Red-Black Tree
- Similar to a splay tree that kind of uses a top-down procedure
- If we use color flipping on the way down, we can avoid the percolation process (which is stack and pointer complex)



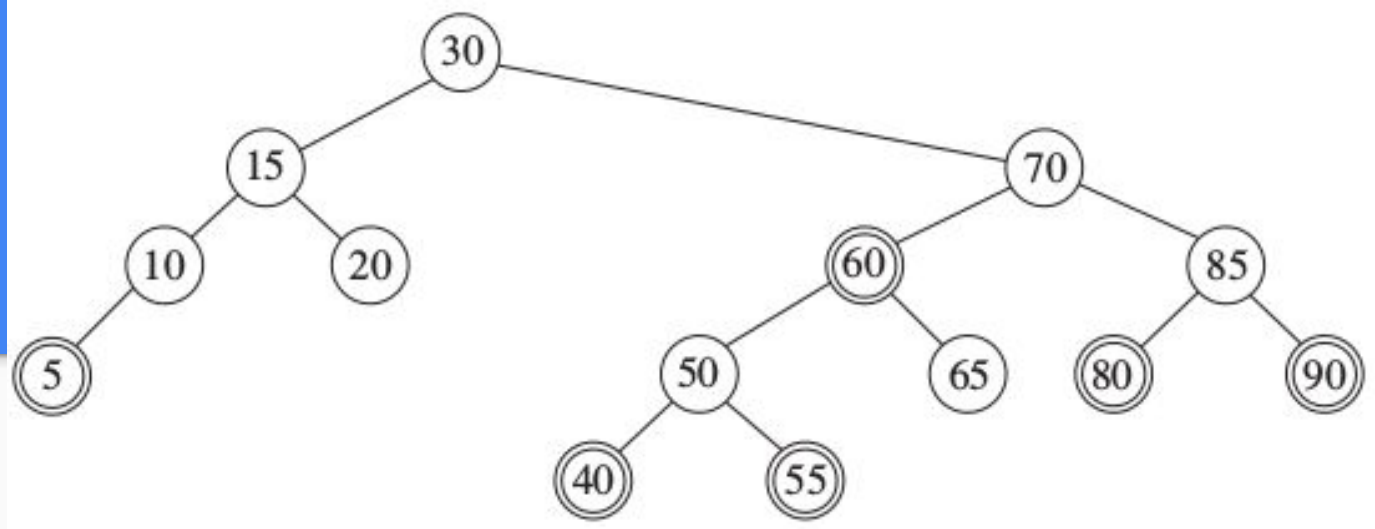
"Sometimes it's good to get a different perspective."

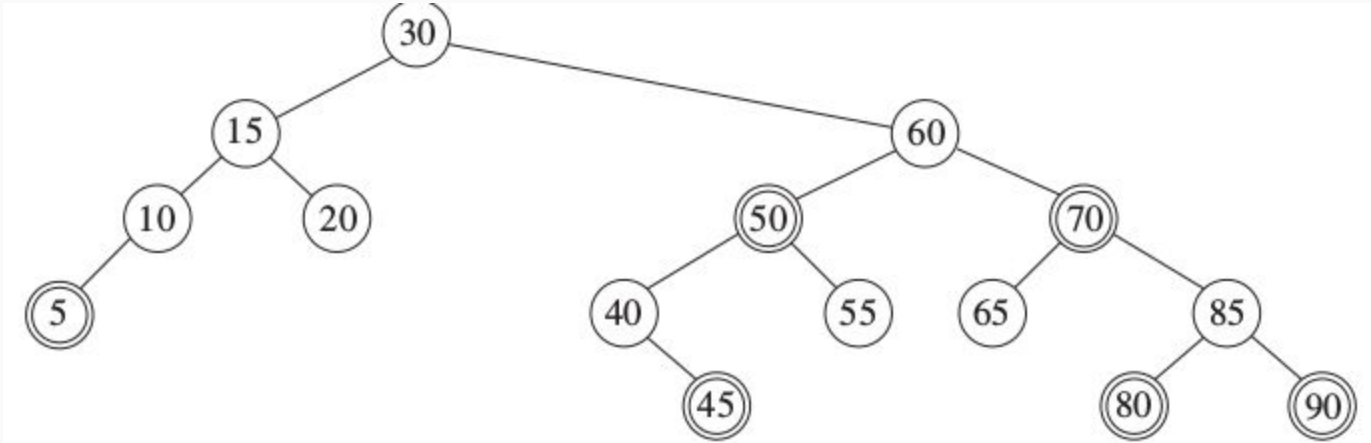
Color FLIP

- On the way *down* during the insert, if we see a node X with two red children, we make X red and the children black. (if X is root, fix at end)
- Induces a property violation if X's parent P is also red
  - Fix it with a rotation (zig-zag or zig-zig)
  - CANNOT get a percolation since we fixed this with the color flip on the way down
    - S (Aunt) cannot be red

Figure 12.11 Color flip: only if *X*'s parent is red do we continue with a rotation

Example: insert 45





The net result of this behavior...

- A tree with a depth about that of an AVL tree
- Is it balanced? No guarantee is the AVL sense
  - But the rotations to keep the Red-Black properties generally keep the tree in good order
- Biggest advantage is the low overhead of insertions
  - Secondary is that in practice, rotations are relatively rare

Overall, it’s more complicated to implement

- Book uses a nullNode (not just nullptr) to help with null leaves
- Also a root sentinel with
  - Key == -INF
  - Color == red
  - *Right == Real root
- These allow for easier testing of Red-Black properties at the leaves and root:
  - nullNode is always black
  - Root’s parent is red so root must be black, even if it was flipped during an insert

Top-down deletion

- Notice, no bottom up deletion? Most implementations are top-down red-black trees for symmetry, just like B-Trees are really B+ trees now
- Key is being able to delete a leaf
- Similar to a BST delete, but we need to ensure the Red-Black properties are maintained if we delete a black leaf node (thereby violating condition #4).

Basic outline of deletion

- Node has two children: replace with smallest in right subtree
  - That node must have one child, which can then be deleted
- Node only has a right child: ditto
- Node with a left child: replace with largest node in left subtree
  - That node must have one child, so it can be deleted

So… what about their colors?

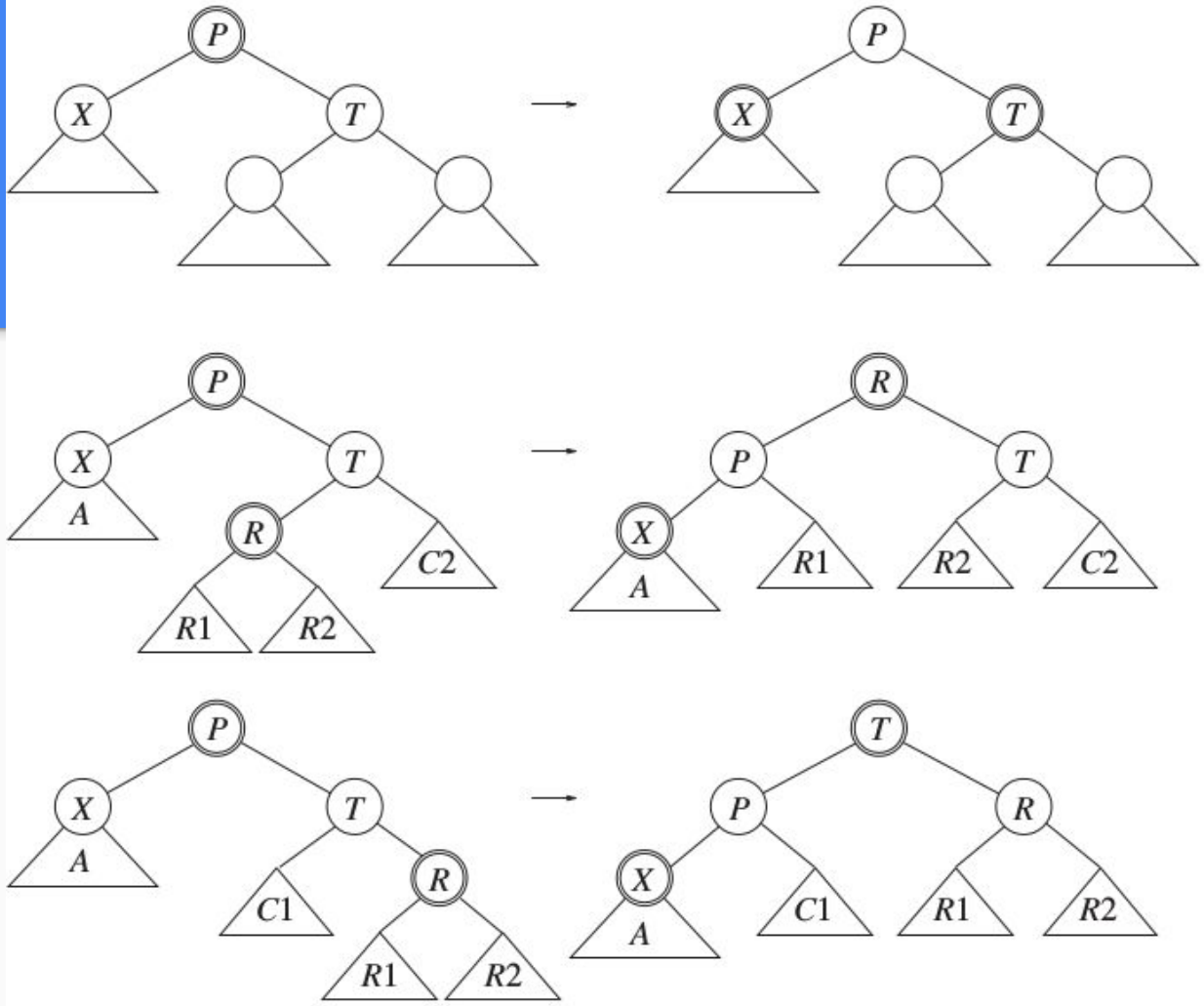
- If leaf node we delete is red, then no worries, just nuke it.
- If node is black… that would violate property #4 since the tree would lose a black node in the root->nullNode count
- So, we just ensure the top-down pass makes the leave node red
  - Simple, right?

To do the top-down delete pass

- Color sentinel red (it’s always red, but this is being pedantic)
- Assume:
  - X == Current node
  - T == Sibling
  - P == Parent of both
- As we traverse down the tree, we attempt to ensure X is red
- When we get to X, we *know* P is red, X is black, and T is black

Case #1: X two black kids

This case has 3 subcases (geez!)



Case #2: X has one red child

- Here, we just fall down to the next level and continue the algorithm
- This will cause a rotation with the next set of X, T, & P nodes
  - We will know the color of the prior set of nodes we worked with, so it will ensure each rotation doesn't need to reach higher than P to keep the properties
- The final rotation will be at the leaf and it will set X red, at which point it can be deleted

Why Red-Black trees?

Despite the complex cases, node color management, and lots of logic, they’re still very efficient.

Looking through the code in the book, it’s not that complex in the end. You just need to be careful to handle the various cases during the top-down insert and delete.

Friday’s class: all review for Exam #1

- Midterm is on Chapters 1-4 + Red-Black trees
  - Anything in the slides or homeworks/programming is up for grabs