Midterm #1 Review

CptS 223 - Fall 2017 - Aaron Crandall

Today’s Agenda

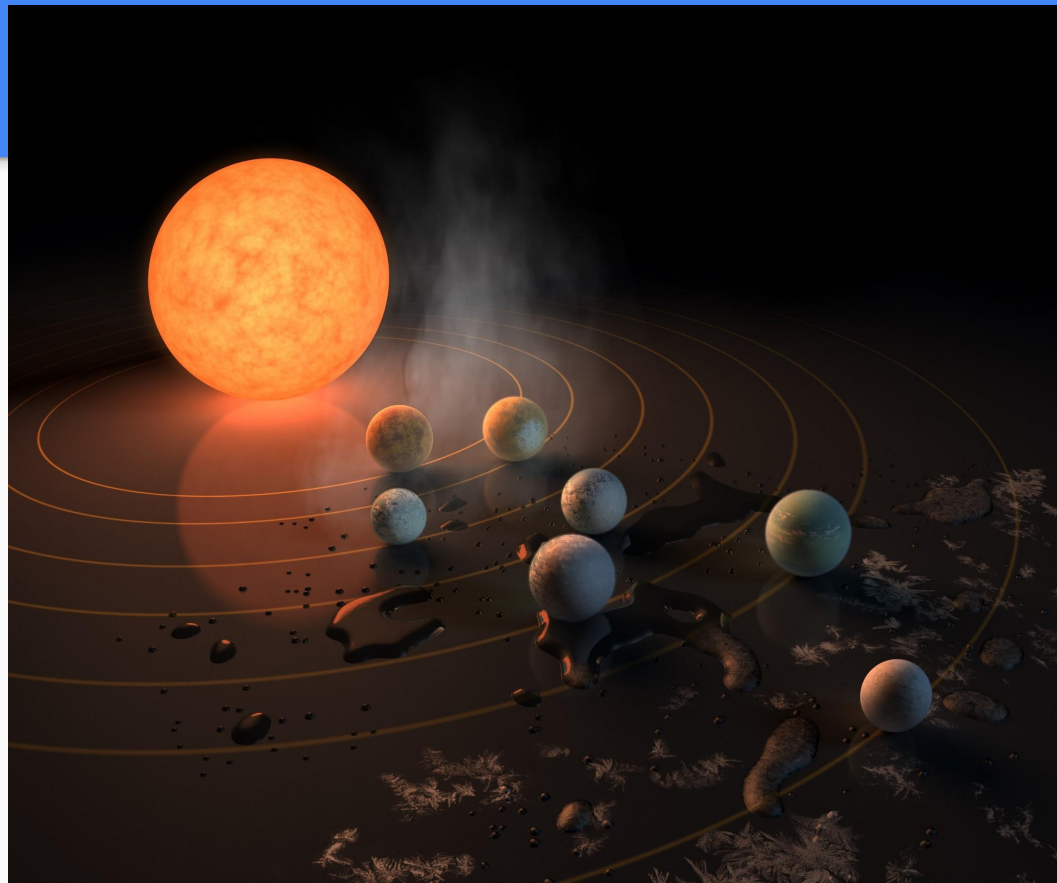
- Announcements
- Humor of the day
- Reviewing for Midterm #1

Announcements

- Midterm #1 is on Monday
  - Only things you have for the exam are:
    - Pencils
    - Calculator (non-Internet connected device)
    - Your brain
    - Hope
- Remember: WSU athletics can apparently kick us all off campus unless we buy tickets to their events. Remember that parking will start ticketing right as this class ends if you're in football parking lots without a permit.

Thing of the day: Solar system of 7 planets discovered!

- Seven earth-sized plants
- Orbiting dwarf star
- 40 light years away
- Some probably have liquid water…
- Thank you TRAPPIST-1 and NASA’s Spitzer Space Telescope
- Too bad James Webb is delayed again!



Hacktoberfest 2017

https://hacktoberfest.digitalocean.com/

Hacktoberfest is a month-long celebration of open source software in partnership with GitHub

To get a shirt, you must make four pull requests between October 1–31 in any timezone. Pull requests can be to any public repo on GitHub, not just the ones we've highlighted.

Midterm #1 - Chapters 1-4 + Red-Black trees

- C++11 features (notably, the Big Five)
- Linux commands & environment
- Algorithm analysis: Big-O, Computation and Space complexity
- ADT’s - Stacks, Lists, Queues & their C++11 STL implementations
- Trees: BST, B+, AVL, Red-Black Trees

Midterm review - Linux stuff

- Basic linux commands
  - Working with files & directories
  - What is an operating system (in a general sense)?
- g++ and make use
  - Notably some command line options (like -g and -std=c++11) and what they do
  - Basics of Makefiles - targets, commands, mostly that it's a system to run building/testing
- git and it's use
  - What is it and what do we like it?
  - What's it good for?

Midterm review - Walking through the book

- Chapter 1 - Programming overview
  - C++11 vectors, Big Five
- Chapter 2 - Algorithm Analysis
  - Focus will be on Big-O for worst, best, and average cases.
    - Start at the innermost loop (or recursive loop), and work outwards
  - For a given function, how much memory is allocated
    - Must include the stack space for recursion when doing space complexity
    - Still done by order O(N) notation
    - If you pass in the array by reference, you get O(1) for the reference/pointer
    - Pass by copy: O(N)

Midterm review - Walking through the book

Chapter 3 - Lists, Stacks, Queues

- What they are
- How they’re implemented in C++11 STL
- Big-O for: insert, delete, search
- Which ones to use for example cases and why

Midterm review - Walking through the book

Chapter 4 - Trees

- Definition of a tree (it’s a special graph - a Directed Acyclic Graph)
- Binary trees
- Binary Search Trees
- Splay trees (but they’re not on the exam)
- AVL trees
- B+ Trees
- Red-Black Trees

ADT complexities - by no means a complete table for all operations!

ADT	Function	Optimal	Average	Worst
Stack	Push/Pop	O(1)	O(1)	O(1)
Queue	Enqueue/Dequeue	O(1)	O(1)	O(1)
List	Insert(i)	O(1) - ends	O(N/2)	O(N)
BST	Insert/Delete	O(log N)	O(log N)	O(N)
AVL	Insert/Delete	O(log N)	O(log N)	O(log N)
Splay Tree	Insert/Delete	O(log N)	O(log N)	O(log N)
Red-Black Tree	Insert/Delete	O(log N)	O(log N)	O(log N)

The C++11 Big Five / Rule of Five

- Destructor
- Copy Constructor
- Move Constructor
- Copy Assignment Operator
- Move Assignment Operator

How long does a copy of an N-element structure take?
How long does a move of an N-element structure take?

Why use different structures?

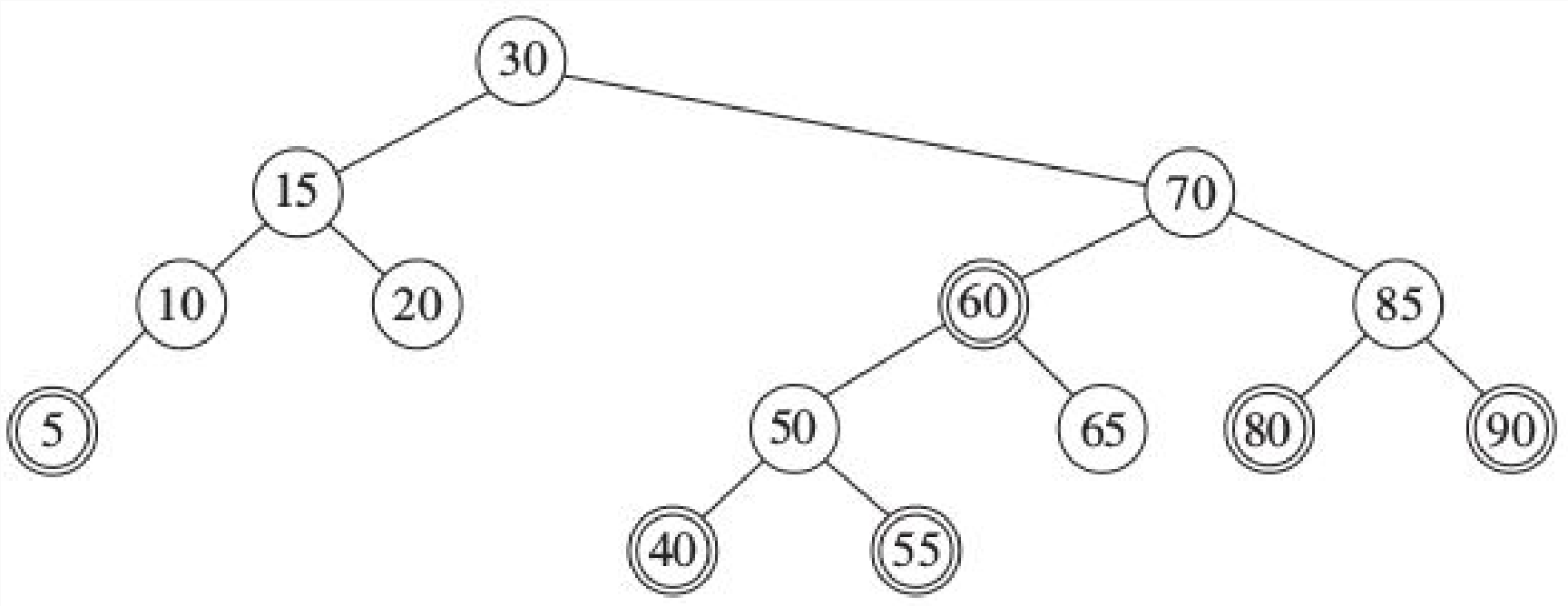
- Be able to decide which structures are right for given situations:
  - Stacks vs. queues
  - Vectors vs. lists
  - BST vs AVL vs Red-Black vs. B+ Trees (sorry Splay trees, you're not in this one)
    - Guaranteed access times and implementation
      - AVL - recursive vs. red-black iterative (same Big-O, but different wall clock)
    - Why AVL vs. B+ Trees? - what happens when they get too big for RAM?
      - What is the memory hierarchy?

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) - which is O(log N)

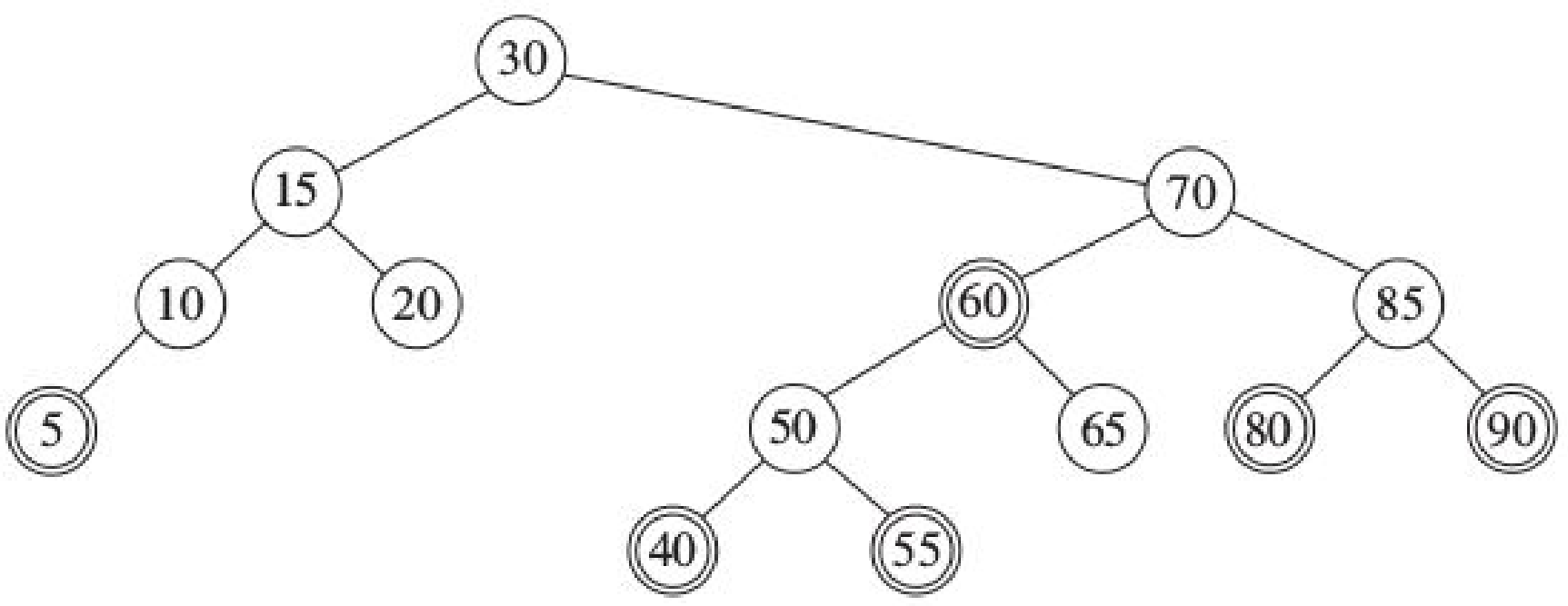
An example of this kind of tree
Double circles are red nodes, singles black



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)

Easy case: insert 25



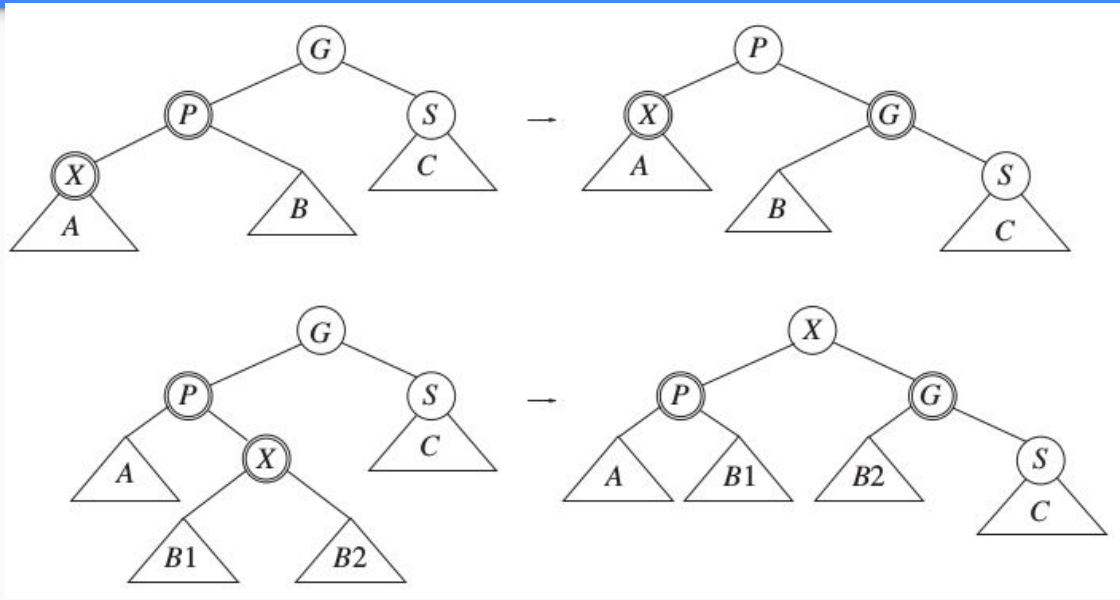
Tree uses Splay tree rotates

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

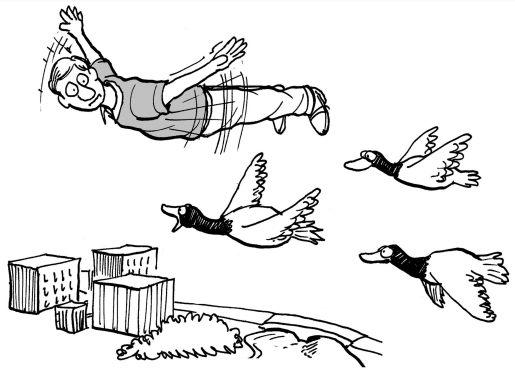


Note the changes to the tree: coloring

- 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

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)
- Results in an iterative algorithm instead of recursive



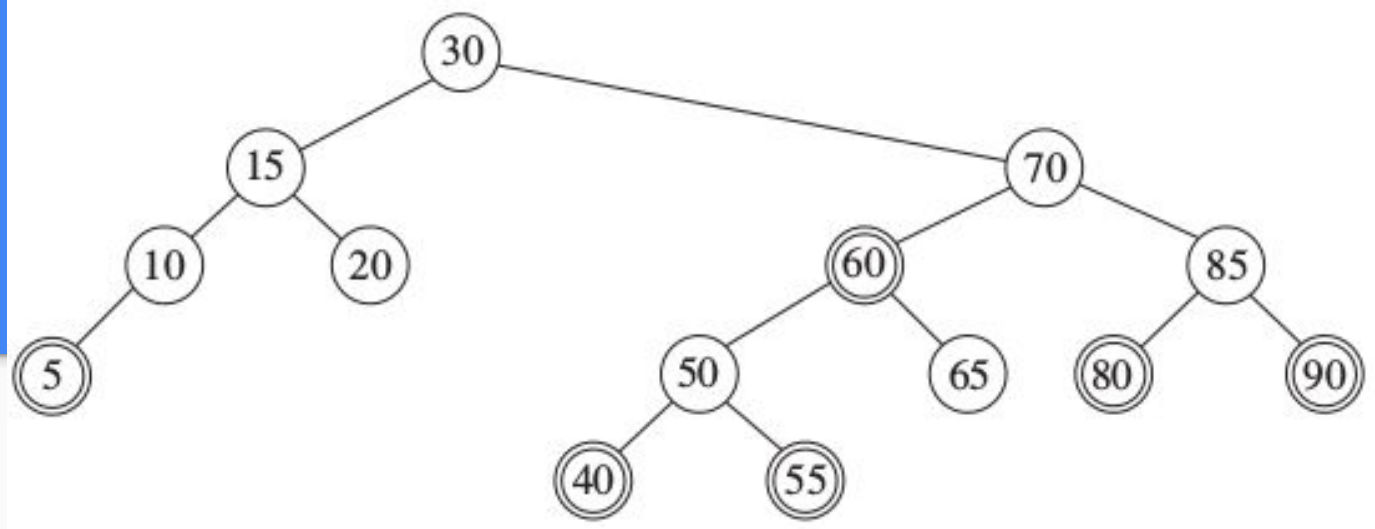
"Sometimes it's good to get a different perspective."

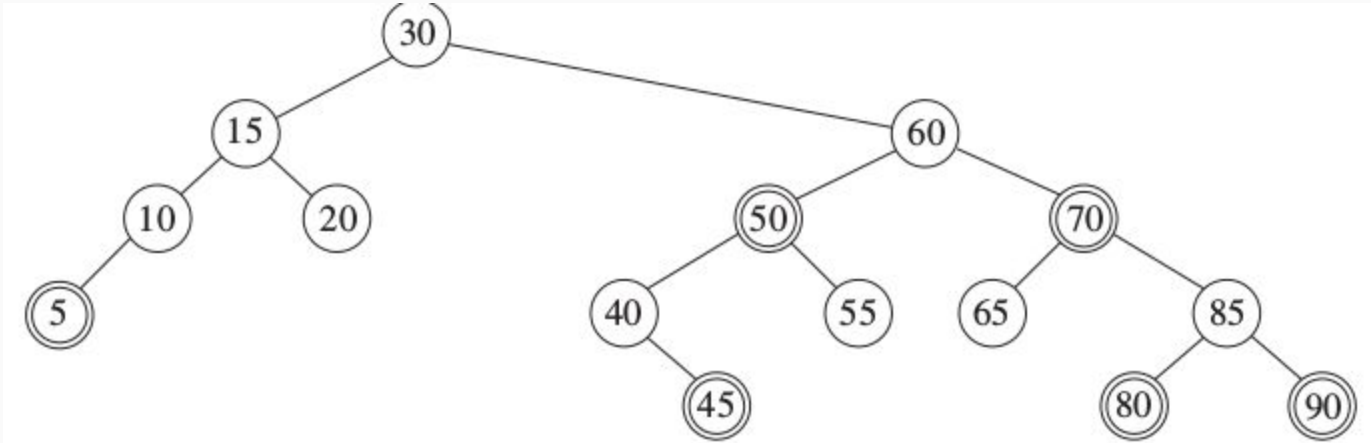
Color FLIP

- 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
    - The key aspect is the balanced number of black nodes from root to leaves
    - This forces only so many nodes in each branch as the red nodes move around
    - The migration of the red nodes essentially keeps a relative balance
- Biggest advantage is the low overhead of insertions
  - Secondarily 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?

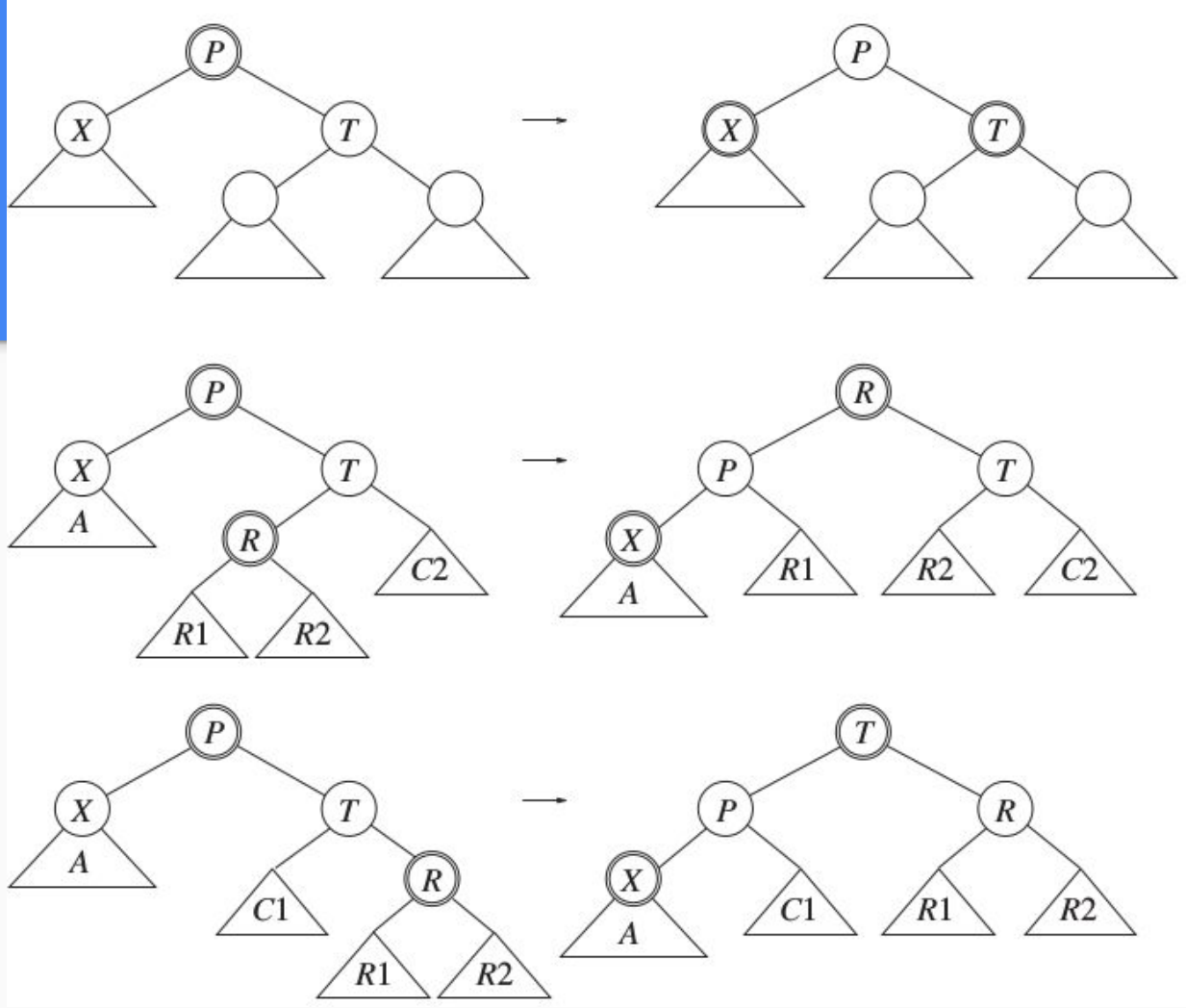
- If leaf node we delete is red, then no worries, just nuke it.
- If node is black… that would violate property #4 since the tree would lose a black node in the root->nullNode count
- So, we just ensure the top-down pass makes the leave node red
  - Simple, right?

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 that leaf (the current 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.

I’ll see you for the exam on Monday

- I’ll post answers to HW2 at 12:01am on Monday