CSCI 2270

Data Structures & Algorithms

Gabe Johnson

Lecture 1 Feb 8, 2013

Red-Black Trees
Sketch It, Make It

Lecture Goals

1. Infinity and RetroGrade
2. Red-Black Trees
3. SIMI

Upcoming Homework Assignment

HW #3 **Due: Friday, Feb 8**

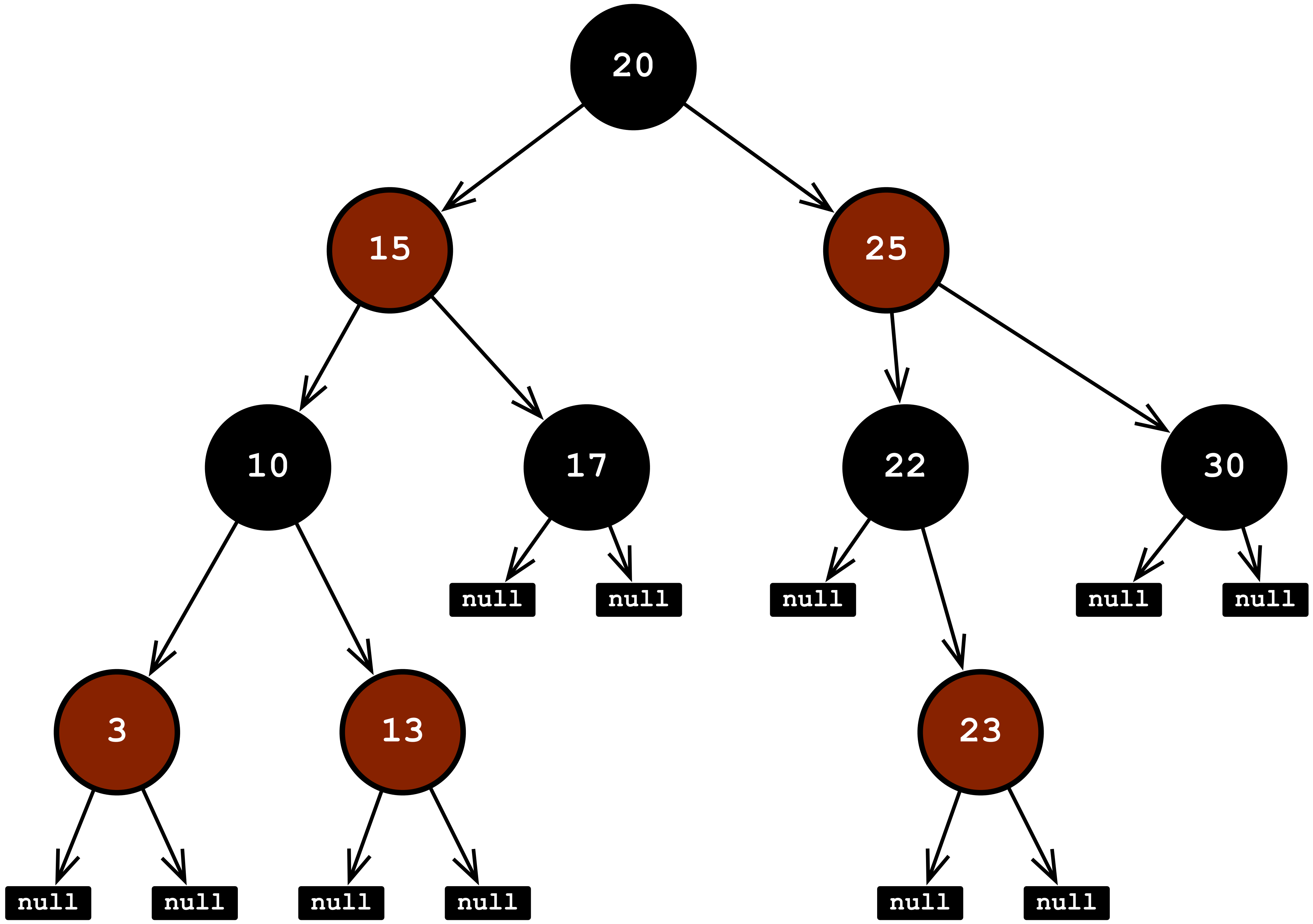
<u>Sorting Algorithms</u>

How’s it going?

RetroGrade getting you down?

Ask your friends to test their code locally before uploading to RG. Also ask them to read about what an infinite loop is. By this point you should not be confused, even slightly, about what these are and how to avoid them.

Red-Black Trees



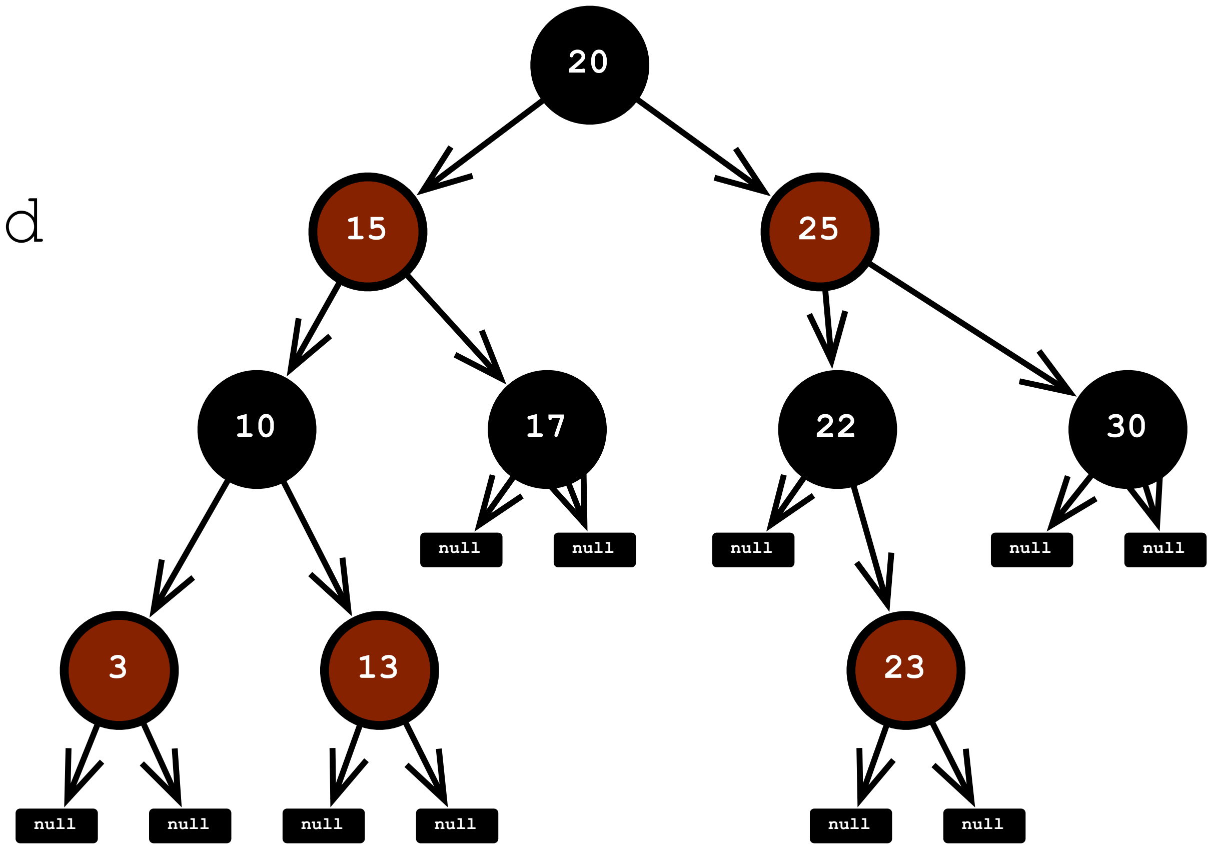
The Rules

Red-black tree rules:

1. Nodes are colored red or black.

2. Root is black. Can ignore this after you get things going.

3. Leaves are null and are considered black.



4. Red nodes have black children.

5. Path from a node to any descendant leaves have the same number of black nodes. *This is the heart of the matter that lets us use coloring to self-balance the tree.*

Red-Black = Self Balancing

The goal is to have a tree that shuffles things around when you add or remove items so the paths from root to leaves is generally O(log n).

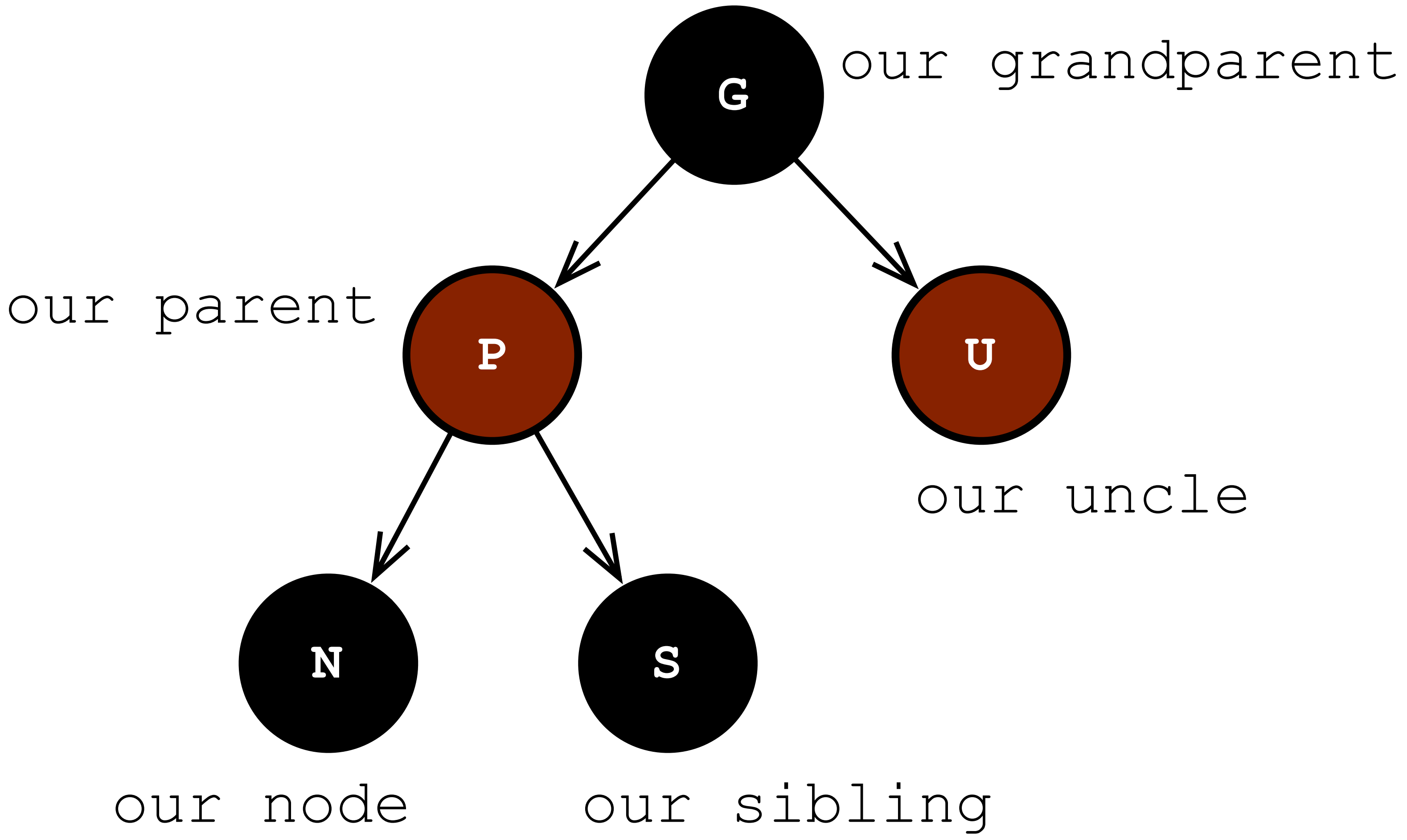
If you have lots of duplicates in your data (maybe somebody inserts nothing but '12') this nice O(log n) dream is quashed. But in general, with normal data that doesn't contain lots of duplicates, we can get pretty close to balanced height.

This means that given reasonable data, the worst case complexity of search and remove is O(log n).

(It might help to recall that big-oh refers to the time an algorithm takes *in proportion to* the input size n.)

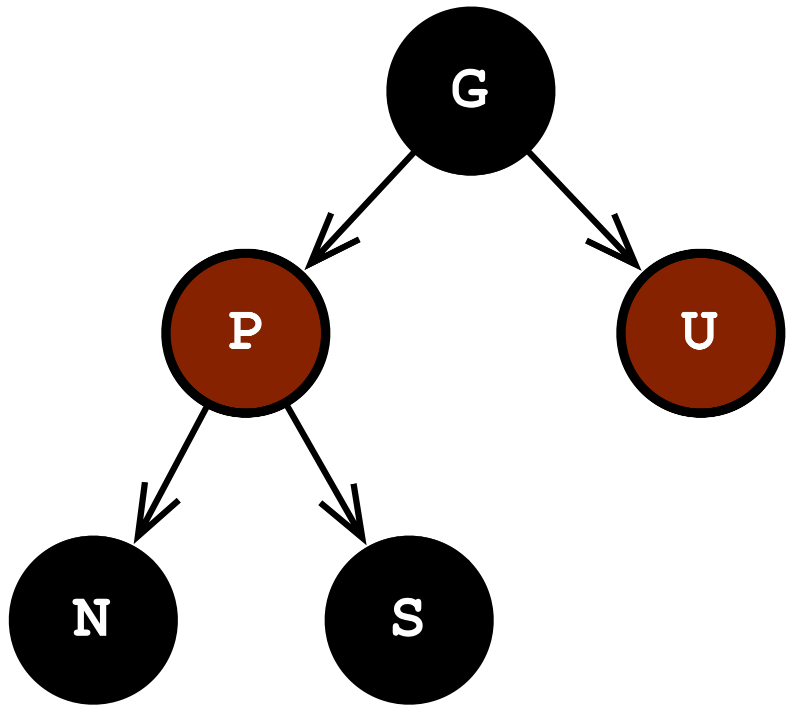
Red-Black Family Tree

Red-black trees keep things in the family. Here's the anatomy of a red-black tree node (n, at the bottom left) and its relationship with the parent, sibling, uncle, and grandparent nodes. These are all relevant in the various insert/delete/rotate operations.



Code For RB Family

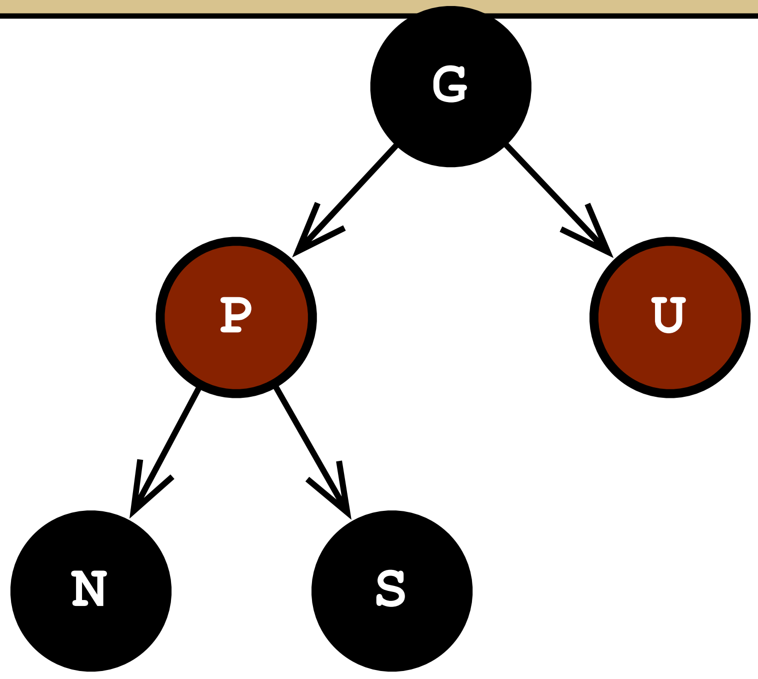
```
RED = 0
BLACK = 1

class red_black_node:
    color = BLACK
    parent = None
    left = None
    right = None
    value = 0
```



```
# returns grandparent of node, or None if not available
def grampa(n):
    if n is not None and n.parent is not None:
        return n.parent.parent
    else:
        return None
```

More Code

```
def sibling(n):
    if n == n.parent.left:
        return n.parent.right
    else:
        return n.parent.left
```



```
# return the uncle node of n (parent's sibling).
def uncle(n):
    g = grampa(n)
    if g is None:
        return None
    if n.parent == g.left:
        return g.right
    else:
        return g.left
```

Insertion Cleanup Chain

Inserting: There are *five* cases you need to consider when inserting a new node. The new node is painted red. Insert it as you would any other binary search tree, then repair any damage you did to the red-black invariant by invoking the cleanup routine chain.

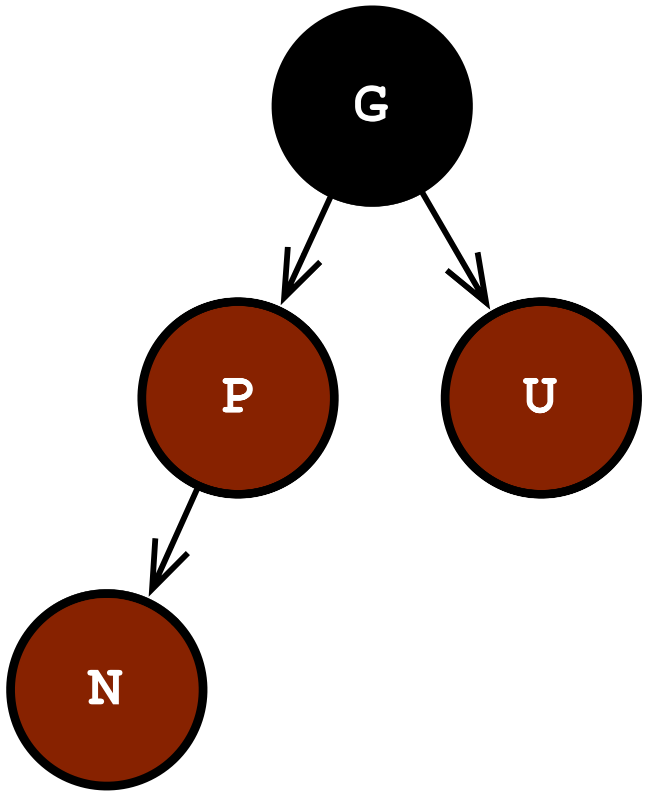
Case 1 is where the new node is the first one, the tree root. Make it black. If the tree already has a root, call case 2.

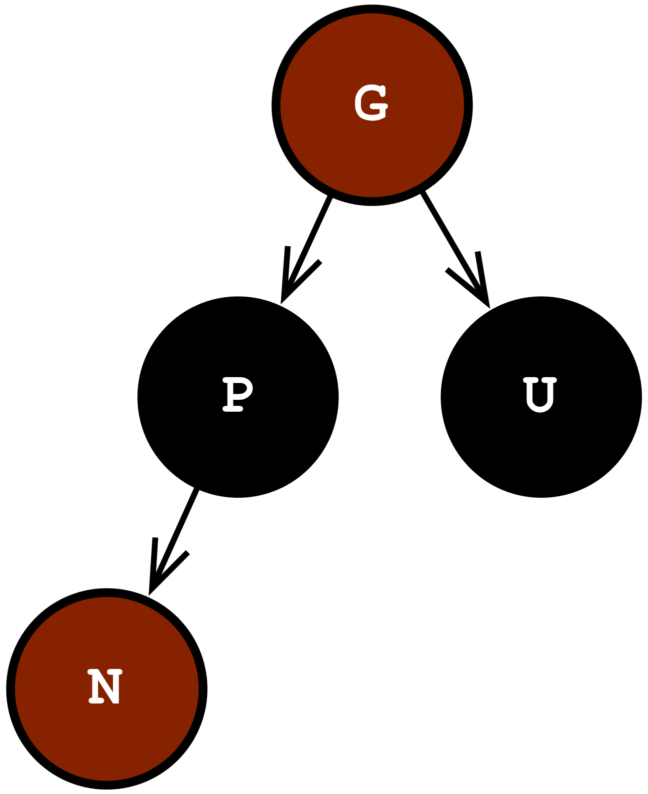
Case 2 is where P is black, so we're done. If P is red, call case 3.

Cases 3, 4, and 5 are trickier and deserve diagrams to explain.

Case 3

Case 3 is when P and U are red.



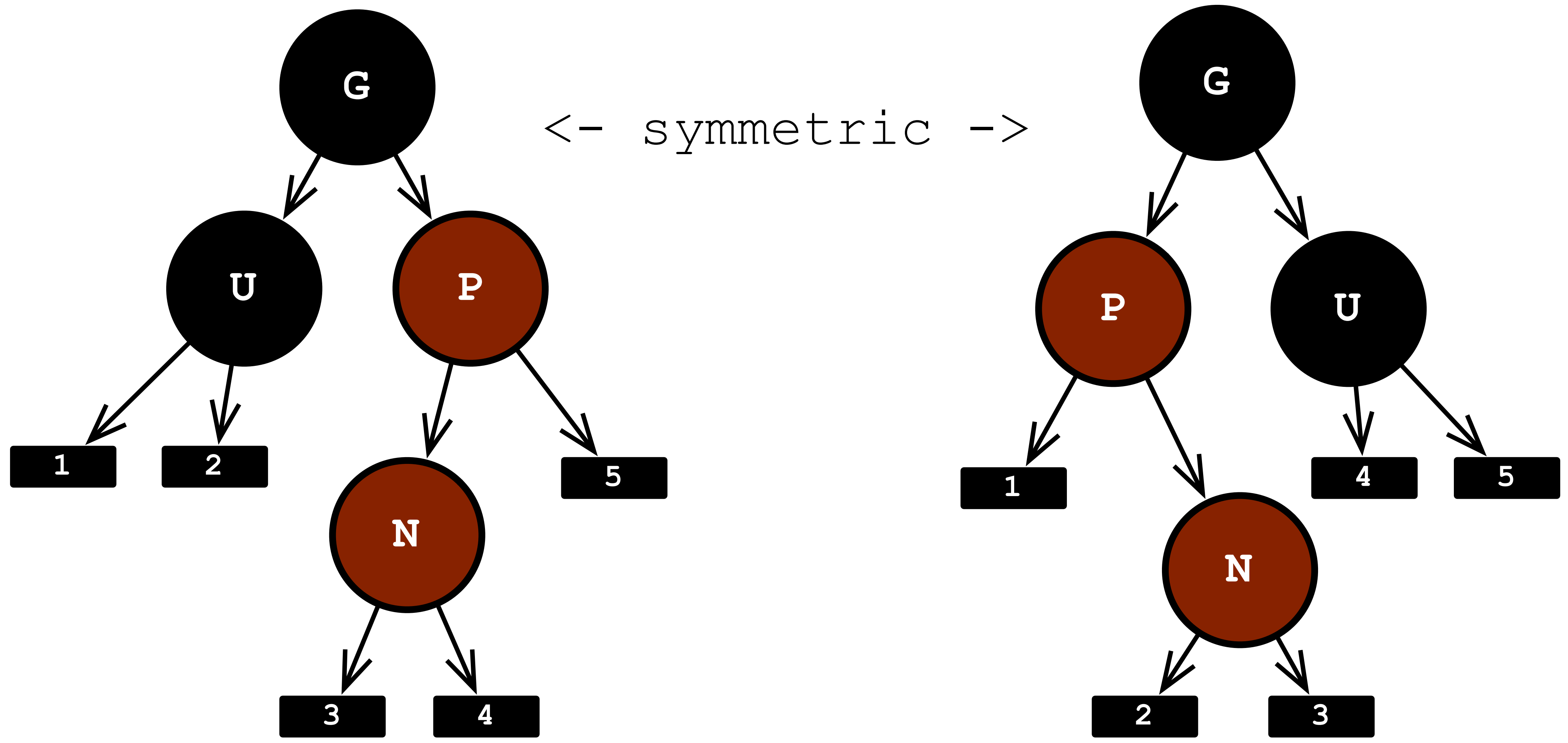


Change the colors as shown here, then call insert case 1 using G.

As always, if case 3 isn't appropriate, call the next insertion case.

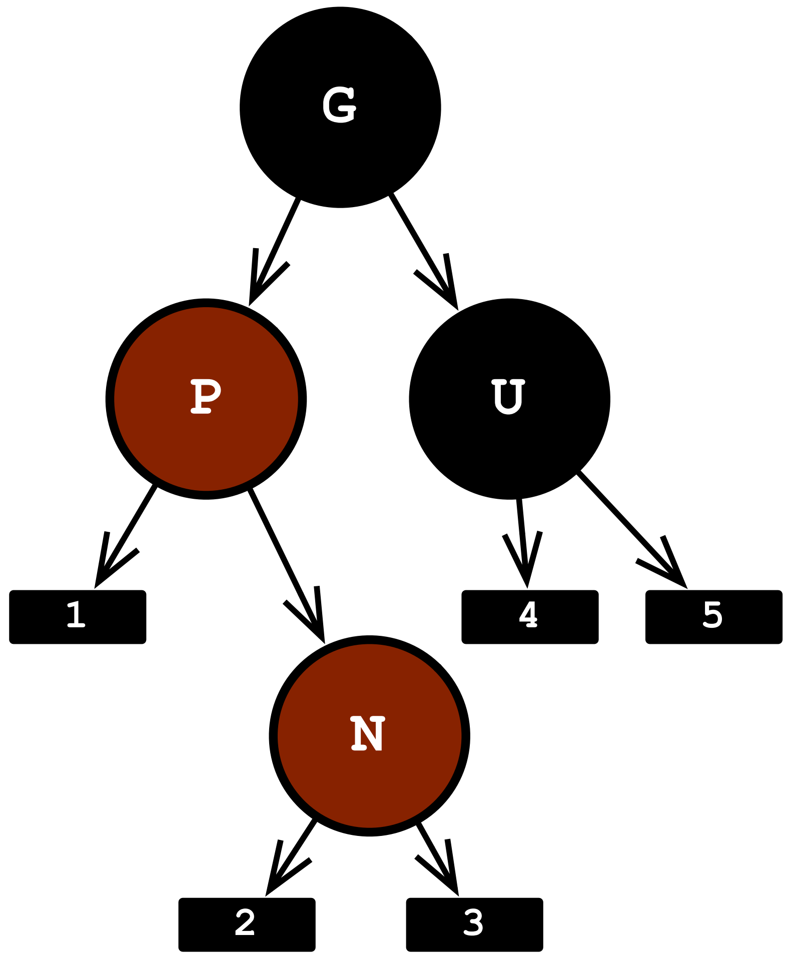
Symmetry for Cases 4, 5

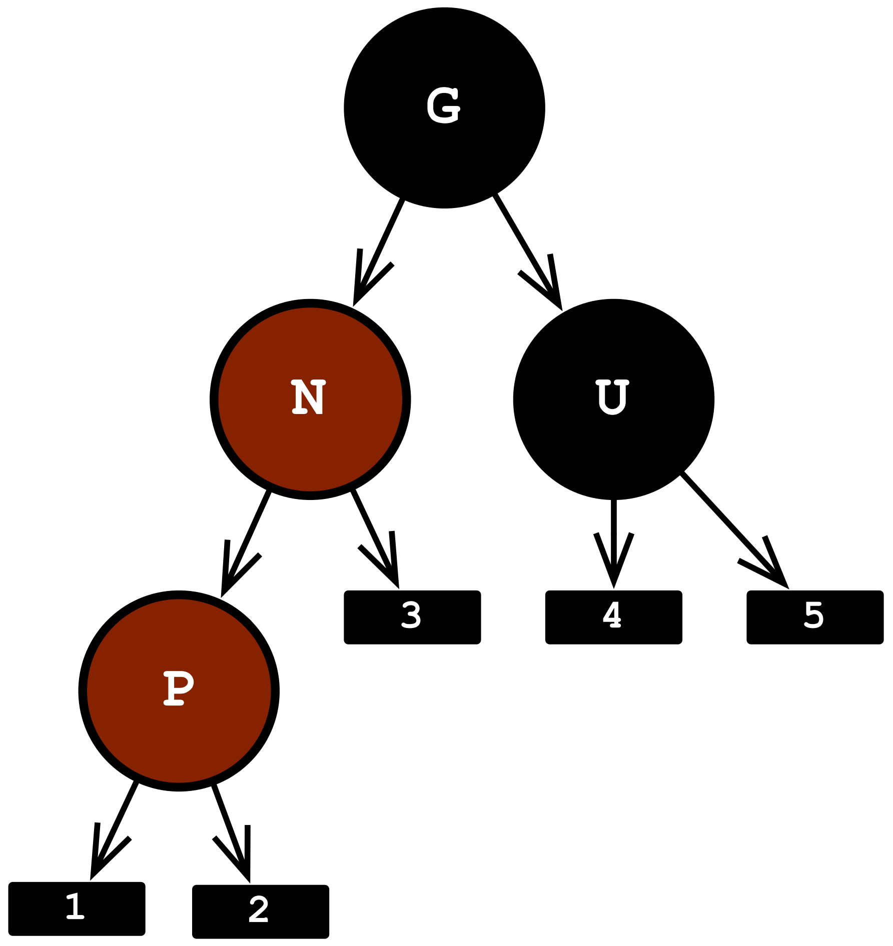
For **cases 4 and 5**, each has a symmetric 'left' and 'right' case. I'm only showing one of them, where we rotate left. But for each, you should have two cases that are symmetric. Just swap 'left' with 'right' and you'll be good.



Case 4

Case 4 is when P is red but U is black, and N is the right child of P, and P is the left child of G.

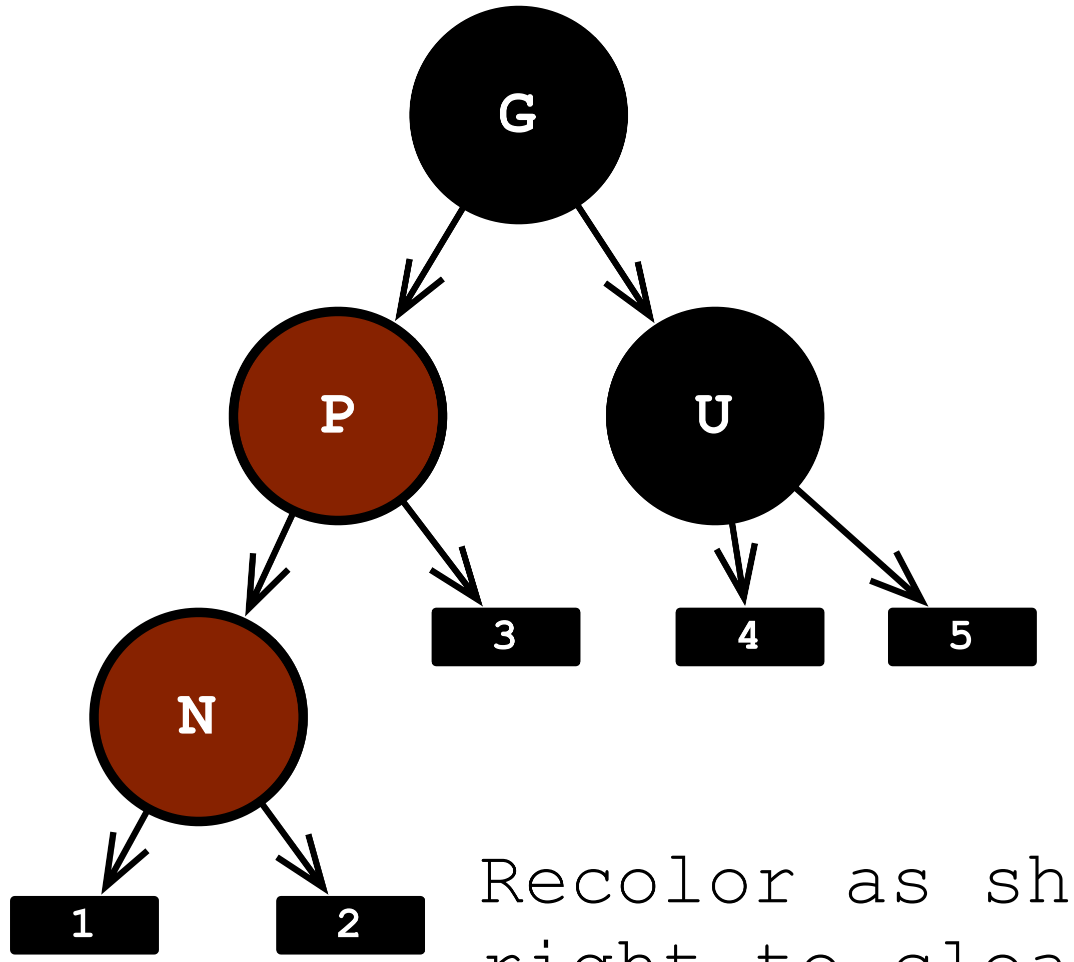




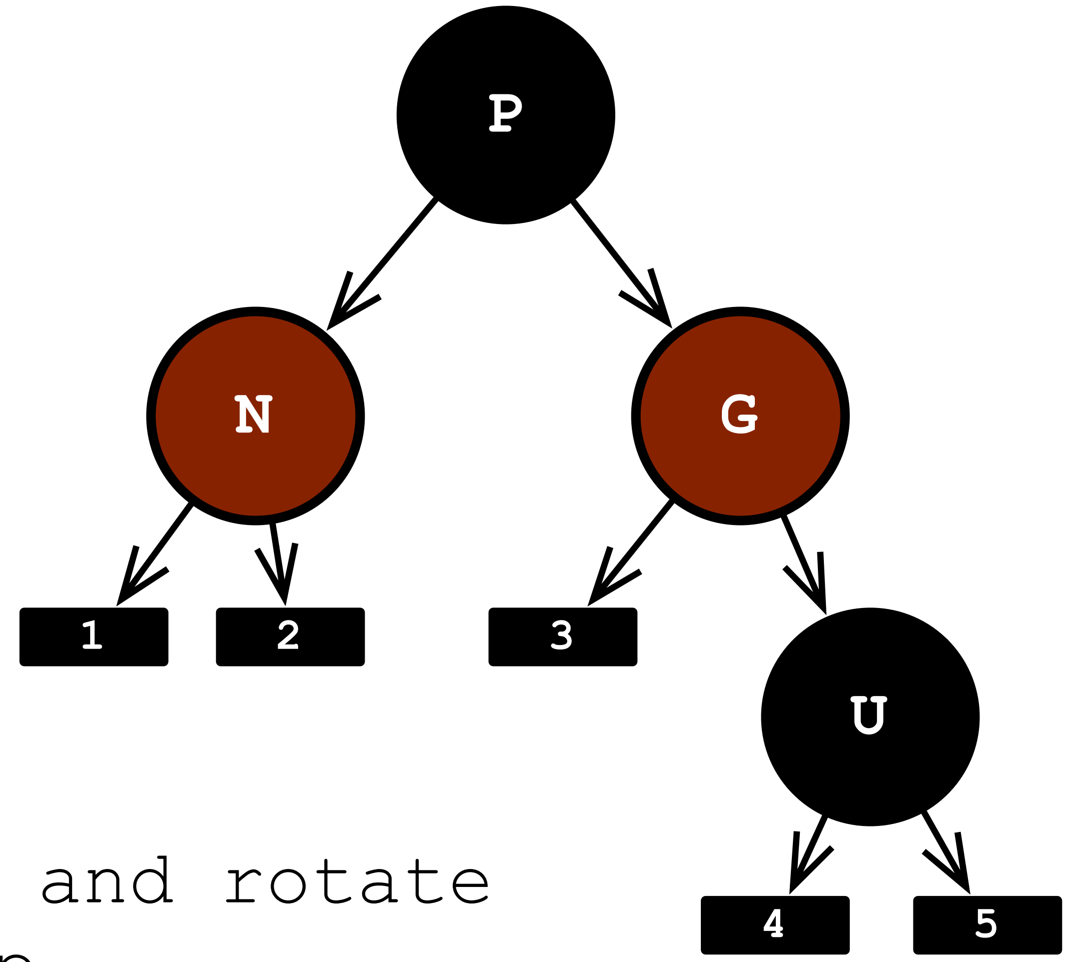
'Rotate Left' as shown here, then cleaning up by invoking case 5 on P.

Case 5

Case 5 is when P is red but U is black, N is the left child of P, and P is the left child of G.



Recolor as shown and rotate right to clean up.



Removing a Node

Removing a node from a red-black tree has a similar process of performing an action and repairing any predictable damage it might have caused.

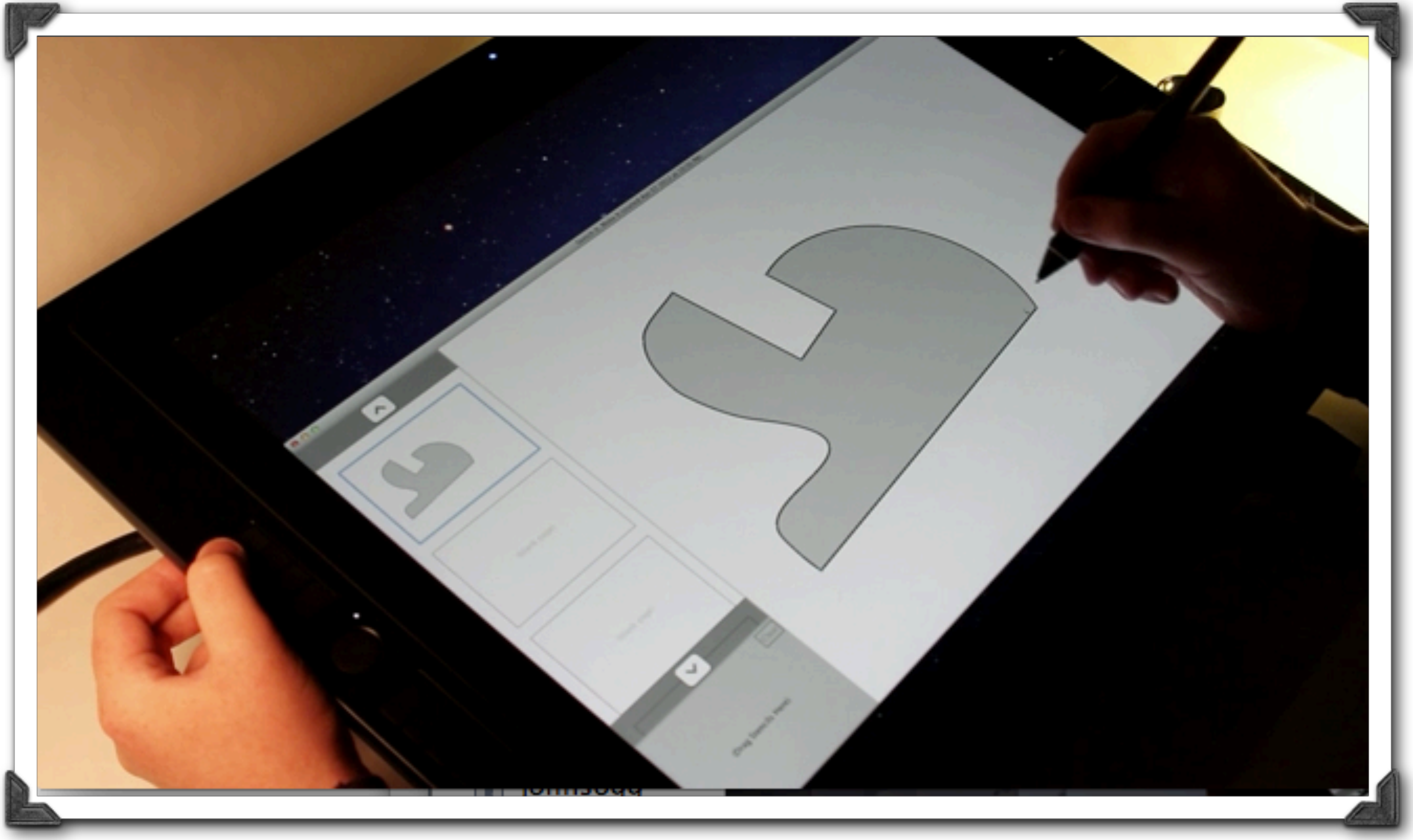
This is a bit trickier than inserting, but the same red-black invariants hold.

Removing a node involves re-painting various nodes and rotating left or right.

We're not going to be tested on removing nodes of red-black trees (but insertion is fair game).

After the test we'll do B-Trees, which are an even more complex balanced tree type. We'll implement all the operations for those. Study how red-black trees work (including remove) before attempting B-Trees, unless you like melty brain.

Sketch It, Make It



(switching over to my thesis defense slides)

The main point here is that you may not ever need to make your own red-black tree, but you will *certainly* run into situations where you need to design, implement, debug, and optimize your own data structures for weird and awesome circumstances.

So it pays to know how to do this stuff.