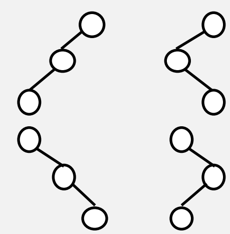
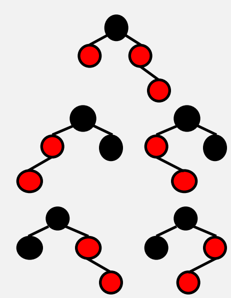
Red-Black Trees

COMP 2210 – Dr. Hendrix

AUBURN
UNIVERSITY

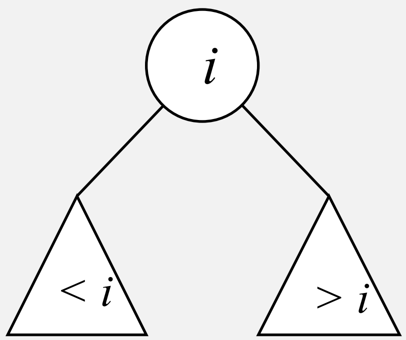
SAMUEL GINN
COLLEGE OF ENGINEERING

Compared to what we already know …

	Max height	Insertion	If unbalanced	Repair
AVL	~$1.44\log_2 N$	Insert according to value, then check balance with a reverse walk. bf = ±2?	Identify a 3-node neighborhood 	Rotations -At most one repair per insertion -Many repairs possible for deletion
RB	$2\log_2(N+1)$	Insert according to value, then check balance with a reverse walk. red-red?	Identify a 4-node neighborhood 	Rotations and recolorings -Many repairs possible for insertion and deletion

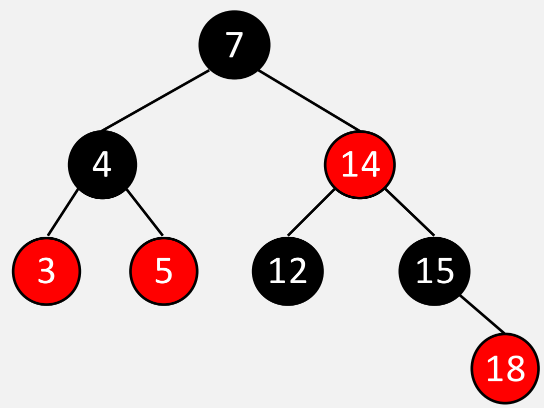
Red-Black Trees

A red-black tree is a **binary search tree** with the following node color rules.



1. Each node is either red or black.
2. The root and all empty trees are black.
3. All paths from the root to an empty tree contain the same number of black nodes.
4. A red node can't have a red child.

Example Red-Black tree:



I know what you're thinking …

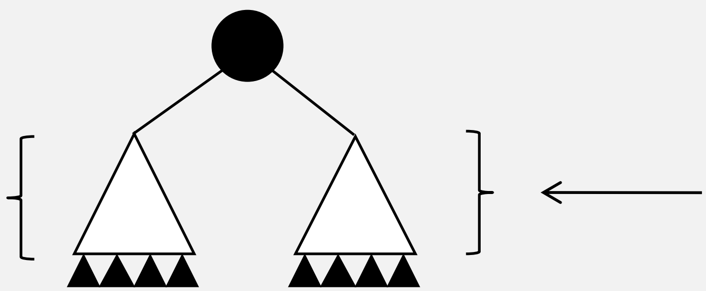
WTF??

Why The Four??

A closer look at the rules…

Rule 1 tells us what types of nodes are legal: red ones and black ones.

Rule 2 specifies the root must be black and, since empty trees are valid trees, it gives them a color (black). We now know what the "boundaries" of a red-black tree looks like.



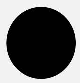
We don't know what this level looks like yet. That's what the other two rules are for.

Rule 3 + Rule 4 = Balance

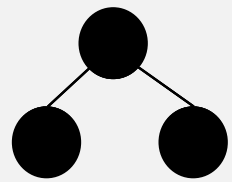
Rule 3 is half of the balance requirement. It makes a statement about the height of the tree in terms of black nodes. This is often called the tree's **black height**.

Applying only rules 1, 2, and 3 would allow the following as red-black trees:

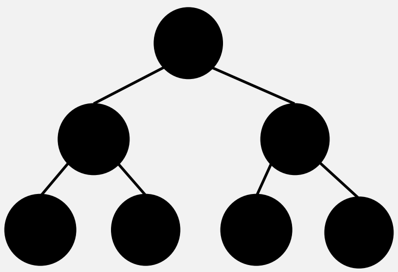
bh=1



bh=2



bh=3



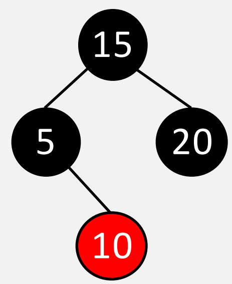
Without red nodes, red-black trees could only be full.

Red nodes

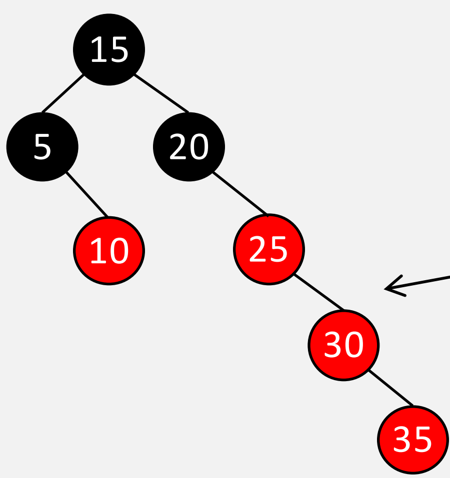
A red node is used like "filler". It allows a red-black tree to obey rules 1, 2, and 3 without being a perfect triangle (full).

This is like the role of the ±1 nodes in AVL trees

For example, using a red node is the only way we can add a new value to this tree:



But … we could take this way too far!



Rule 4 keeps this from happening.

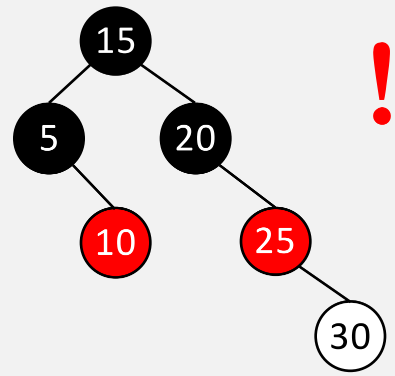
Rule 3 v. Rule 4

Rule 3 puts a constraint on how we use black nodes.

Rule 4 puts a constraint on how we use red nodes.

Think about the effect of these two rules as we (*intuitively*) add nodes.

Suppose that we have inserted 3 values and the tree looks like this:



!

This is just to illustrate a point. Real algorithm works a bit differently.

Now suppose we add more values:

Add 10 Must be red

Add 25 Must be red

Add 30 **Can't be red** **Can't be black**

Violates Rule 4 *Violates Rule 3*

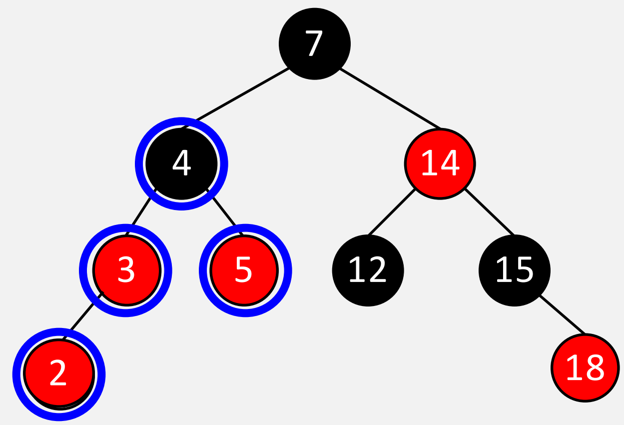
The tension between Rule 3 and Rule 4 forces rotations and recolorings, and thus keeps the tree balanced.

Inserting elements

Use the standard BST insertion algorithm to insert the new node. (Ex: 2)

What color do we make the new node?

Red *Why?*



Beginning with the red node just inserted, walk the reverse path back toward the root, looking for violations of Rule 4. (red-red)

Stop at the first (lowest) red node that has a red parent. This node's grandparent roots the 4-node neighborhood that will be repaired.

Repairs will be a combination of rotations and re-colorings.

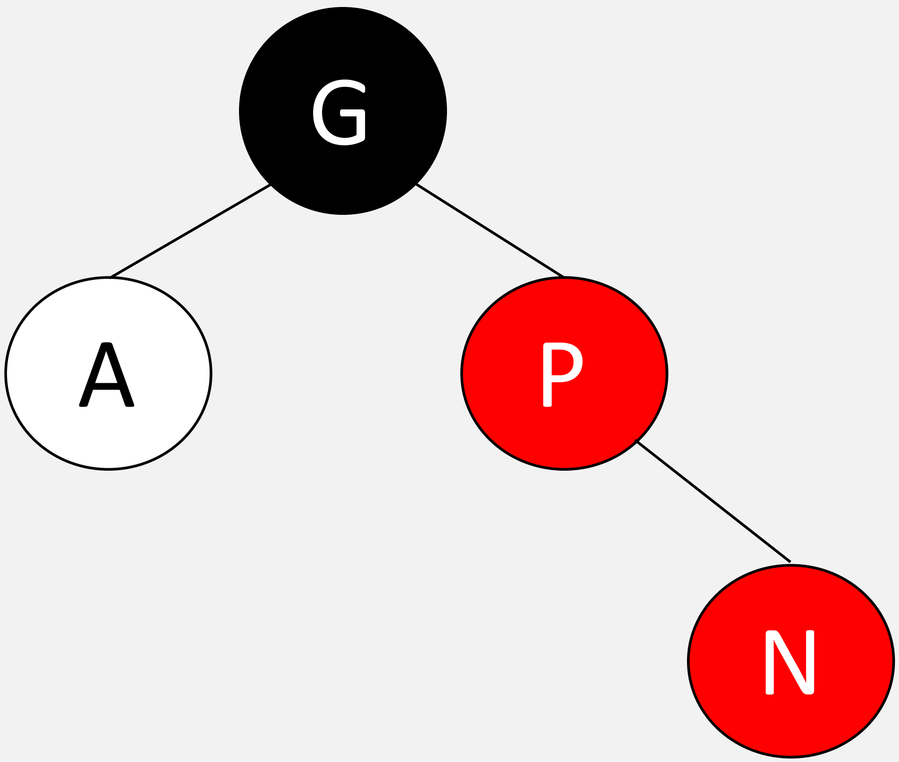
The 4-node neighborhood

The bottom node (N) of the neighborhood is the first red node with a red parent (P).

The grandparent (G) of N is the root of the 4-node neighborhood.

What color is G? **Black**

The ancle* (A) of N is the fourth node.



The repair needed is determined first by A's color and second by the structural configuration of these four nodes.

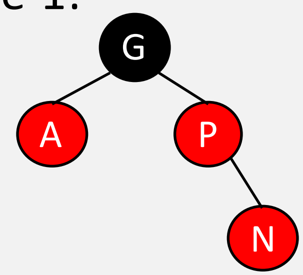
**gender-neutral form for a parent's sibling.*

5 cases for repair

A is red

Repaired by only re-coloring nodes.

Case 1:

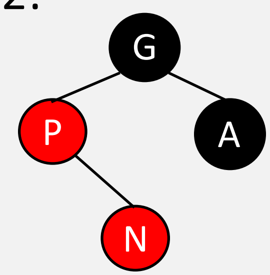


There are 4 structural sub-cases here. It only matters that A is red, however.

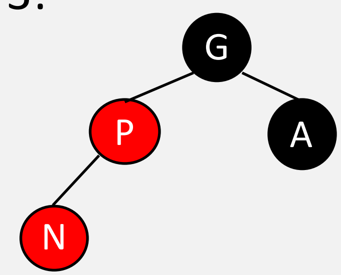
A is black

Repaired by rotations and re-coloring nodes.

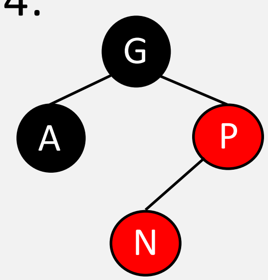
Case 2:



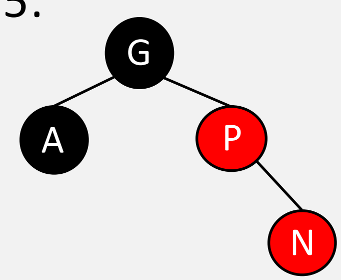
Case 3:



Case 4:



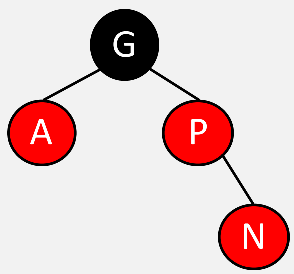
Case 5:



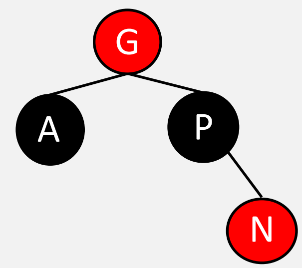
Case 1 repair

Re-color only

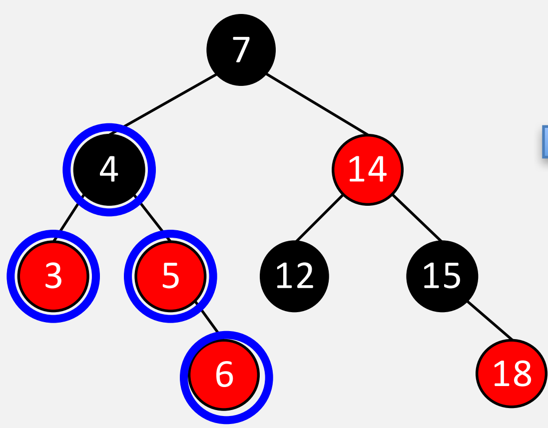
A is red

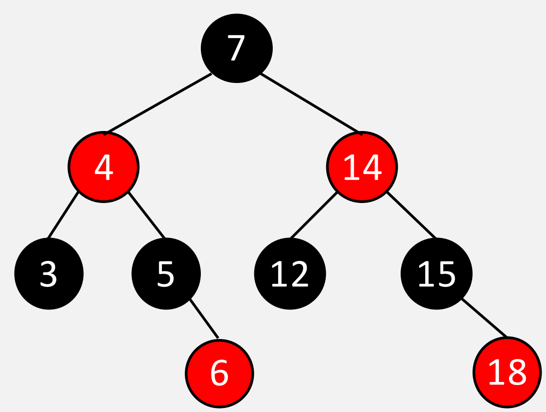


Color flip P
Color flip A
Color flip G

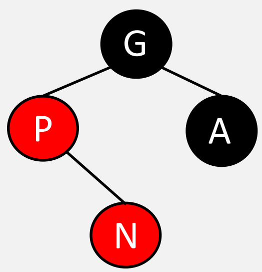


Example: Add 6



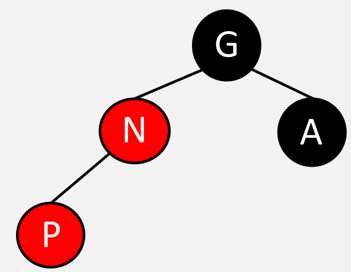


Case 2 repair



Rotate only

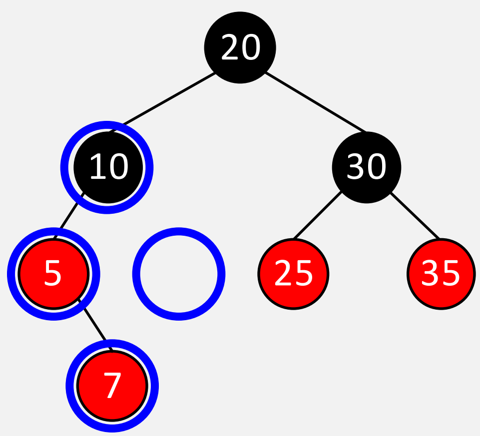
Rotate **left** around P

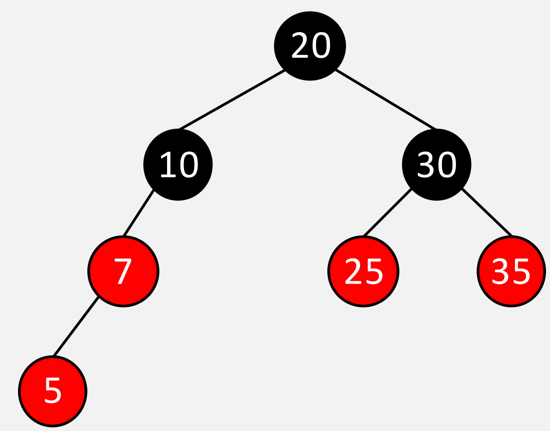


Still violates Rule 4!

The rotation transforms this into Case 3 for final repairs.

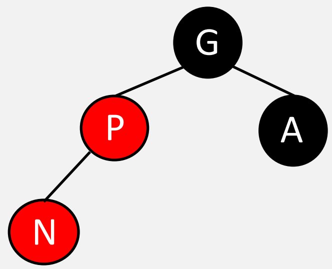
Example: Add 7





Case 3 repair

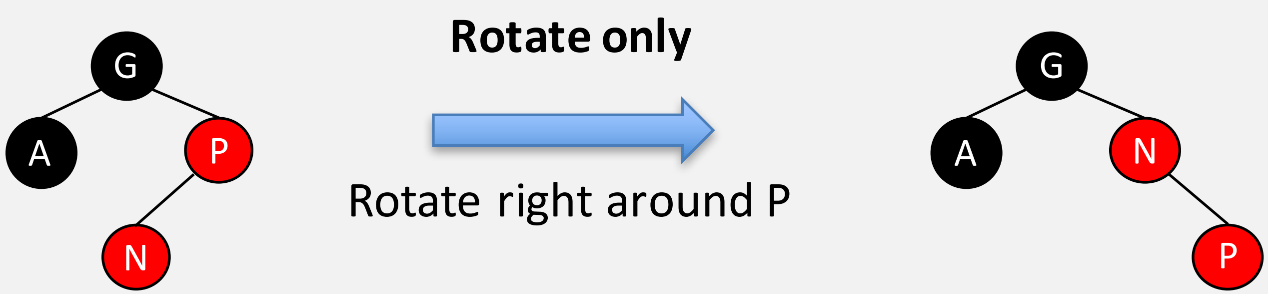
Rotate and re-color



Color flip P
Color flip G
Rotate right around G

Example: Add 5 *(or come from the previous Case 2)*

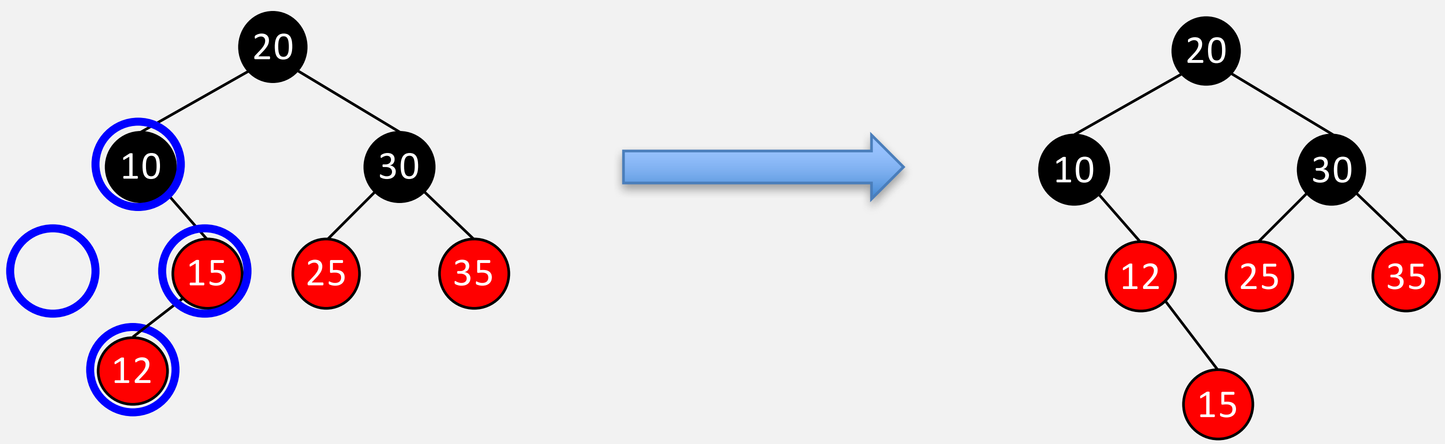
Case 4 repair



Still violates Rule 4!

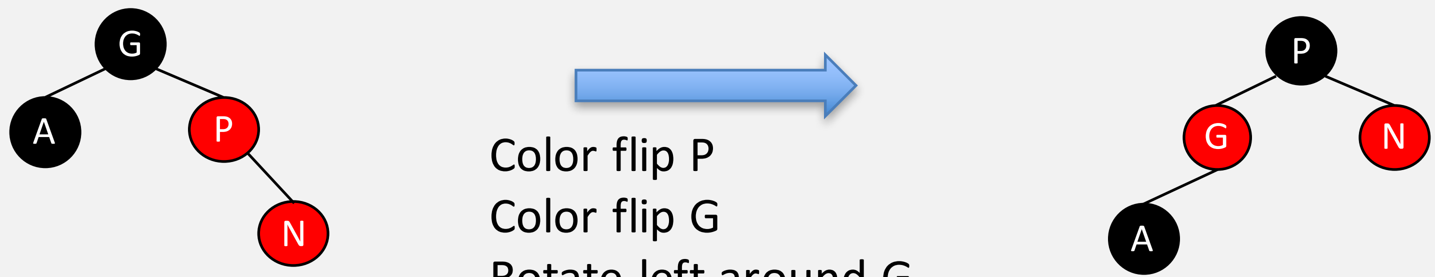
The rotation transforms this into Case 5 for final repairs.

Example: Add 12



Case 5 repair

Rotate and re-color

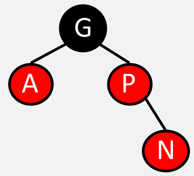


Color flip P
Color flip G
Rotate left around G

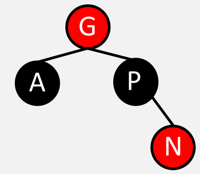
Example: Add 15 *(or come from the previous Case 4)*

Repair case summary

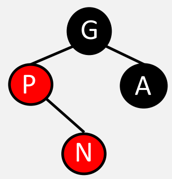
Case 1



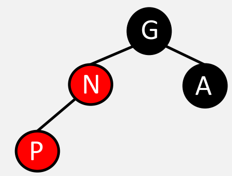
Color flip P
Color flip A
Color flip G



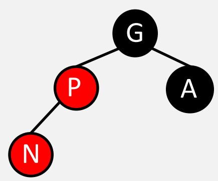
Case 2



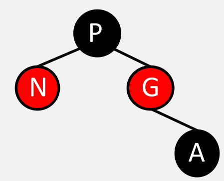
Rotate left around P



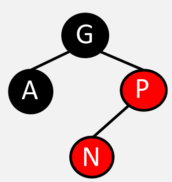
Case 3



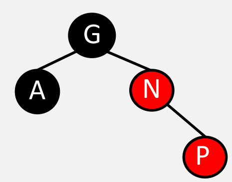
Color flip P
Color flip G
Rotate right around G



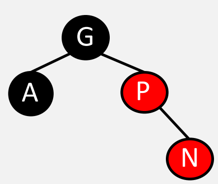
Case 4



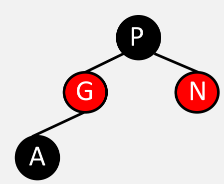
Rotate right around P



Case 5

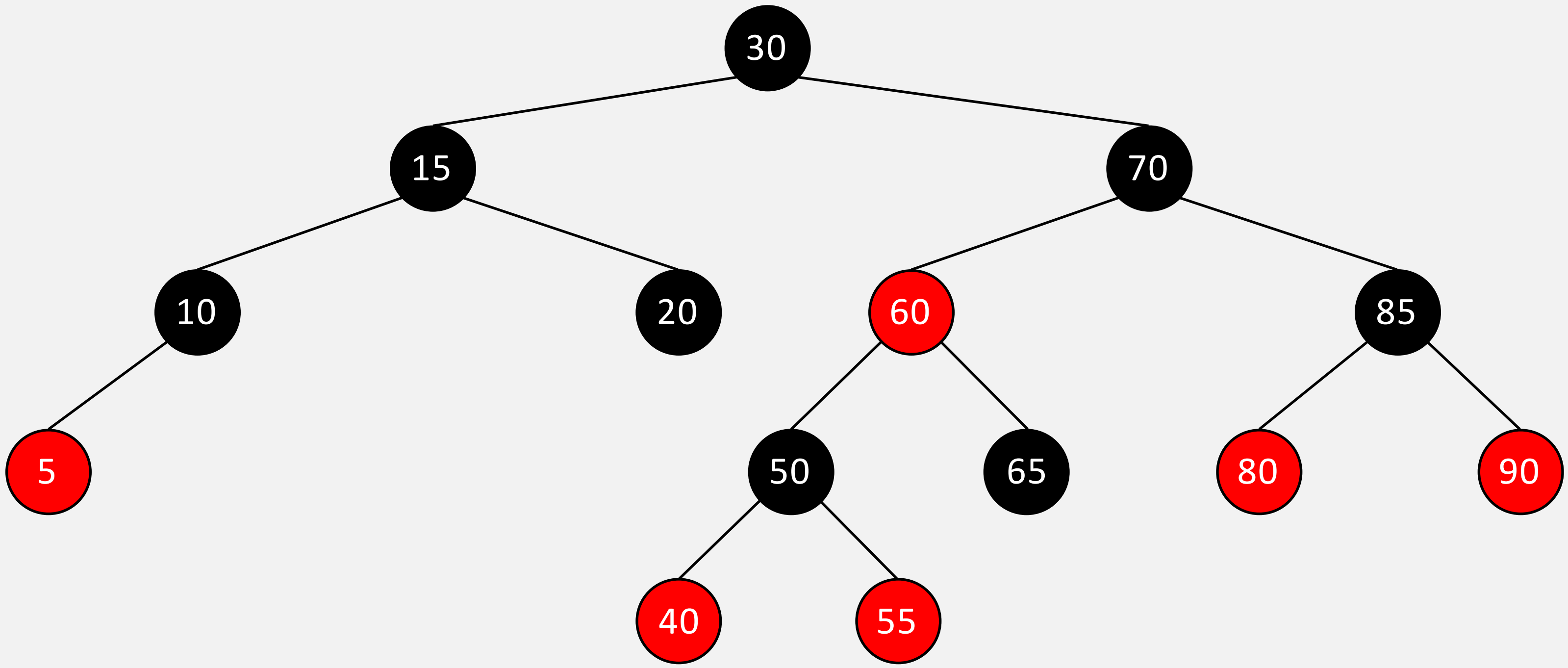


Color flip P
Color flip G
Rotate left around G

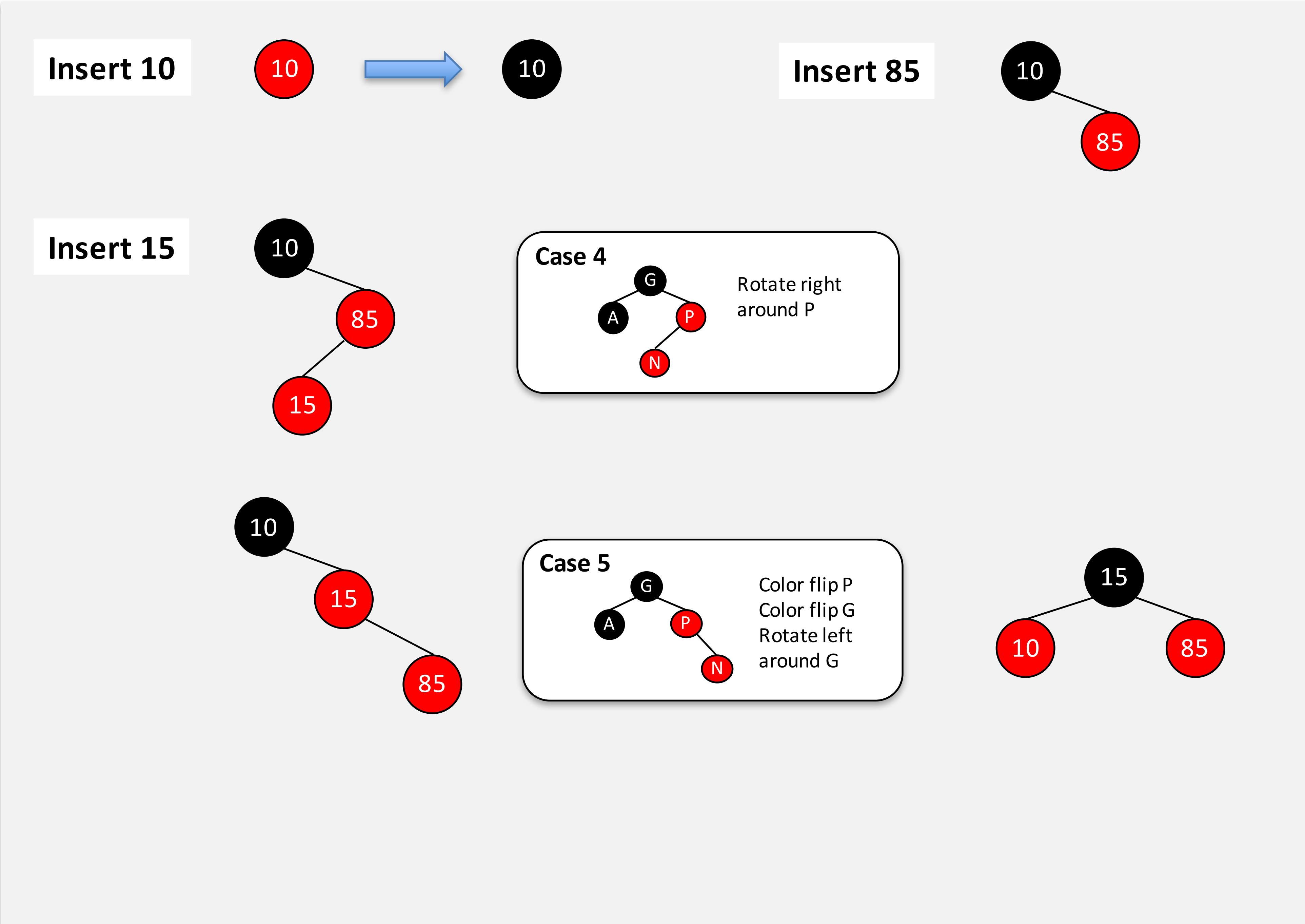


Building a red-black tree

Insert: 10, 85, 15, 70, 20, 60, 30, 50, 65, 80, 90, 40, 5, 55

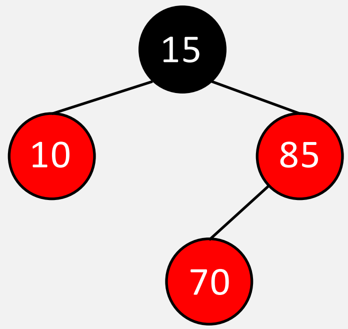


Building a red-black tree

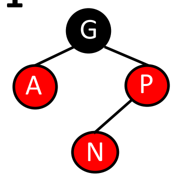


Building a red-black tree

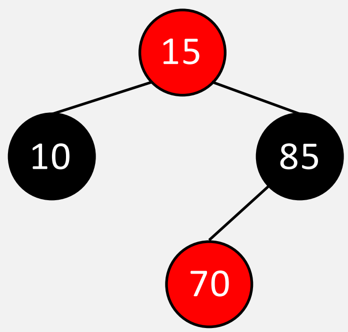
Insert 70



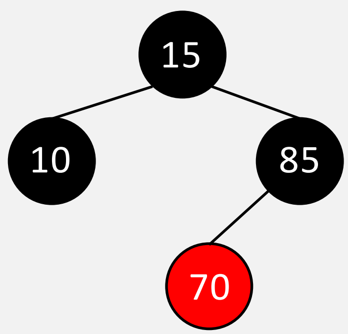
Case 1



Color flip P
Color flip A
Color flip G

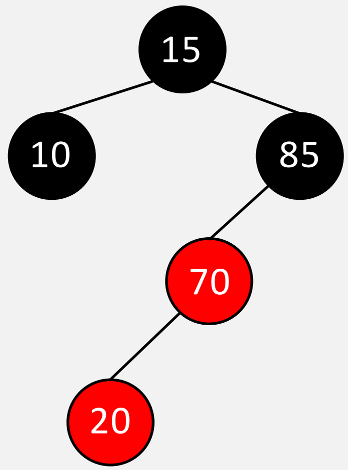


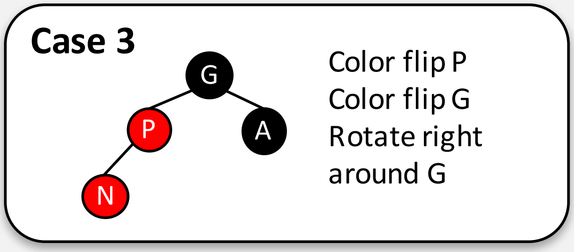
Root must always be black.

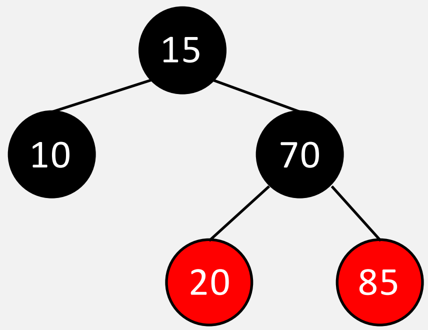


Building a red-black tree

Insert 20

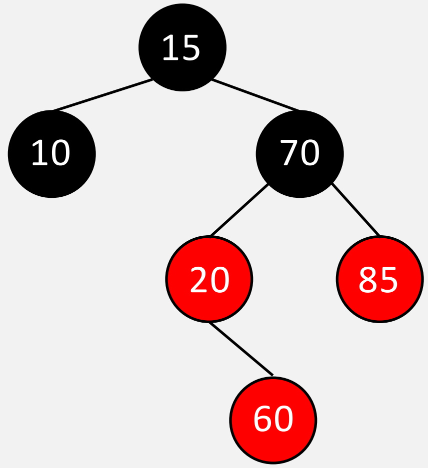




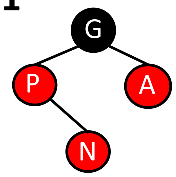


Building a red-black tree

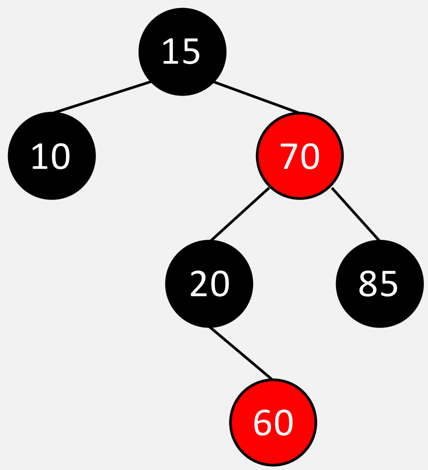
Insert 60



Case 1



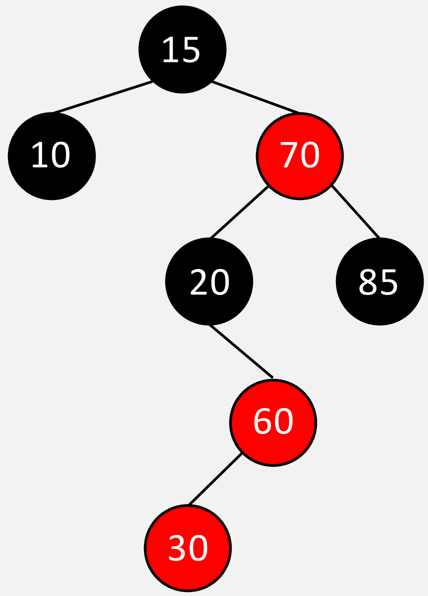
Color flip P
Color flip A
Color flip G



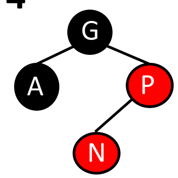
Taller than the corresponding AVL tree.

Building a red-black tree

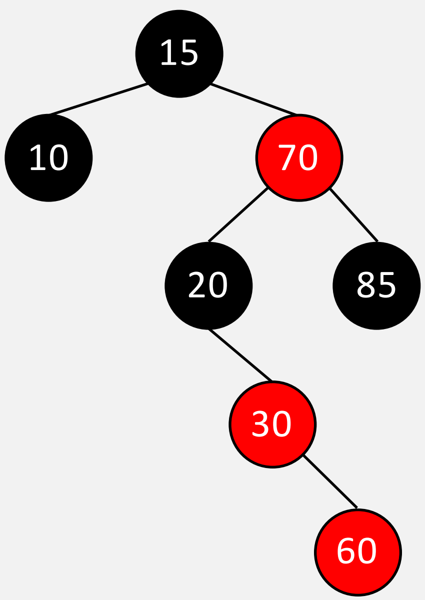
Insert 30



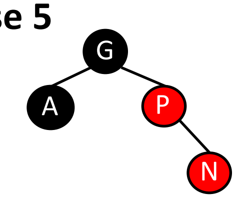
Case 4



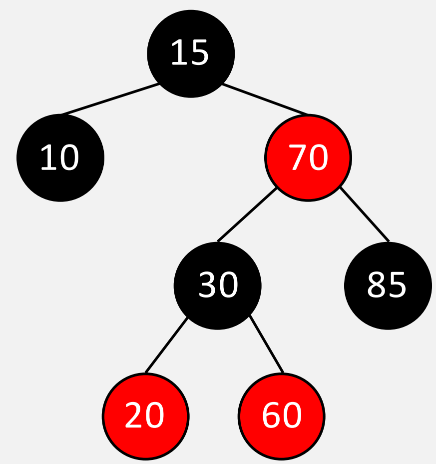
Rotate right
around P



Case 5

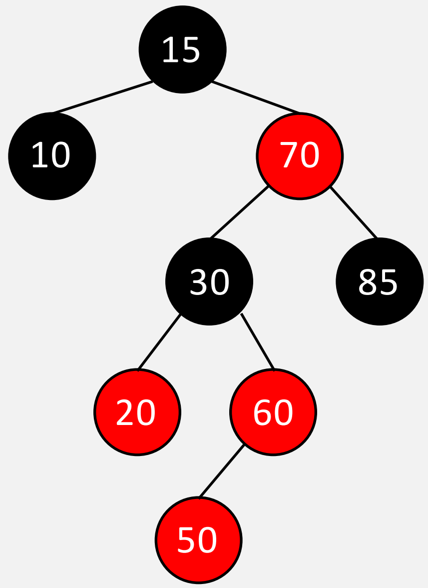


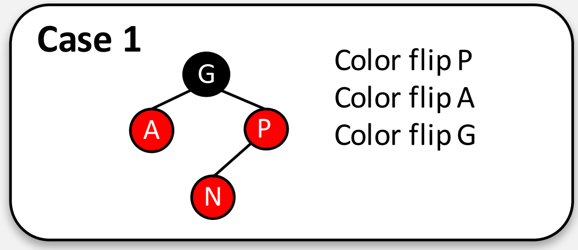
Color flip P
Color flip G
Rotate left
around G

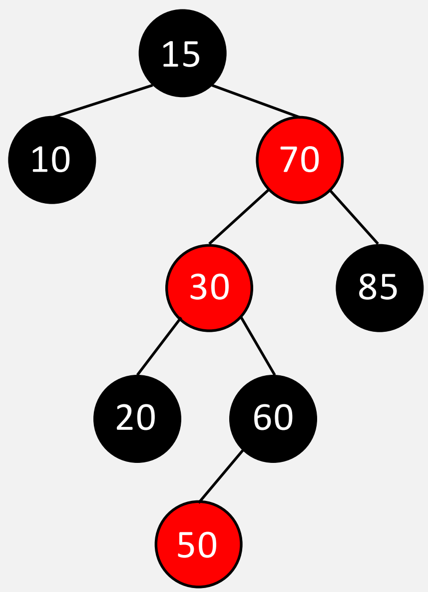


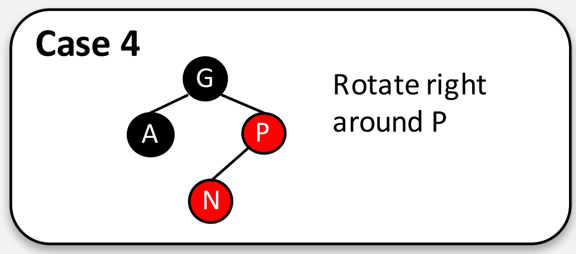
Building a red-black tree

Insert 50

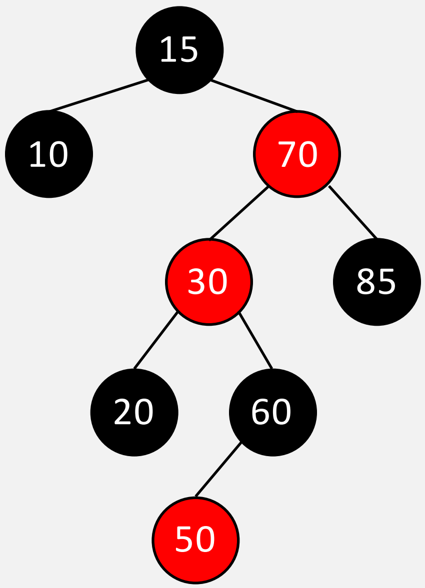


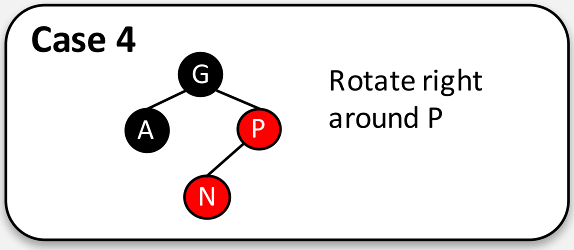


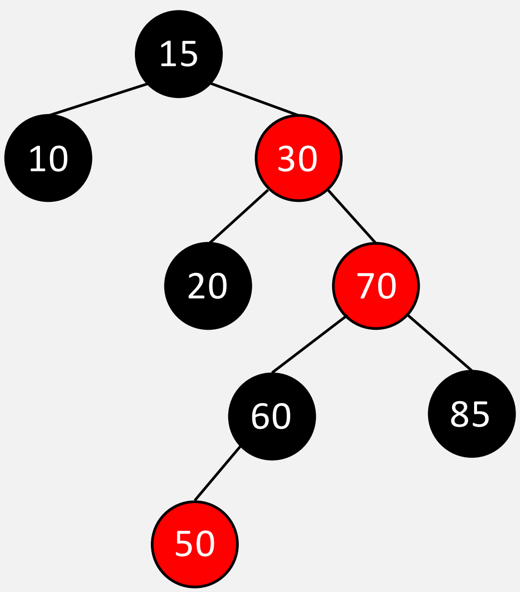


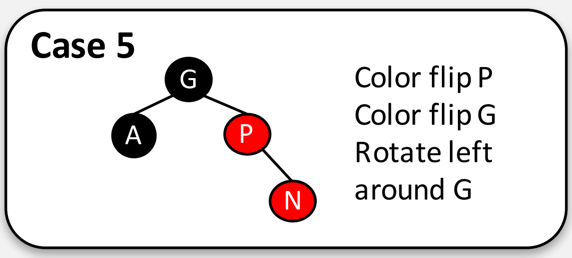


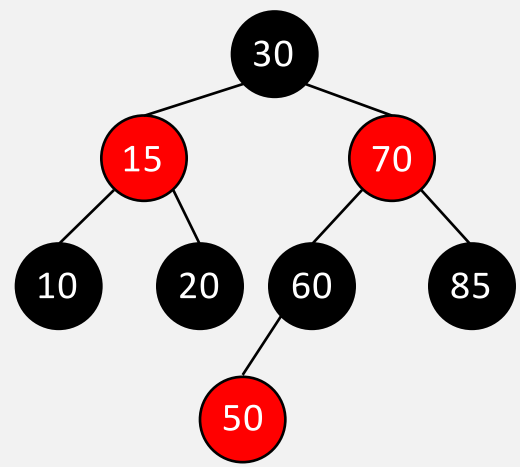
Building a red-black tree





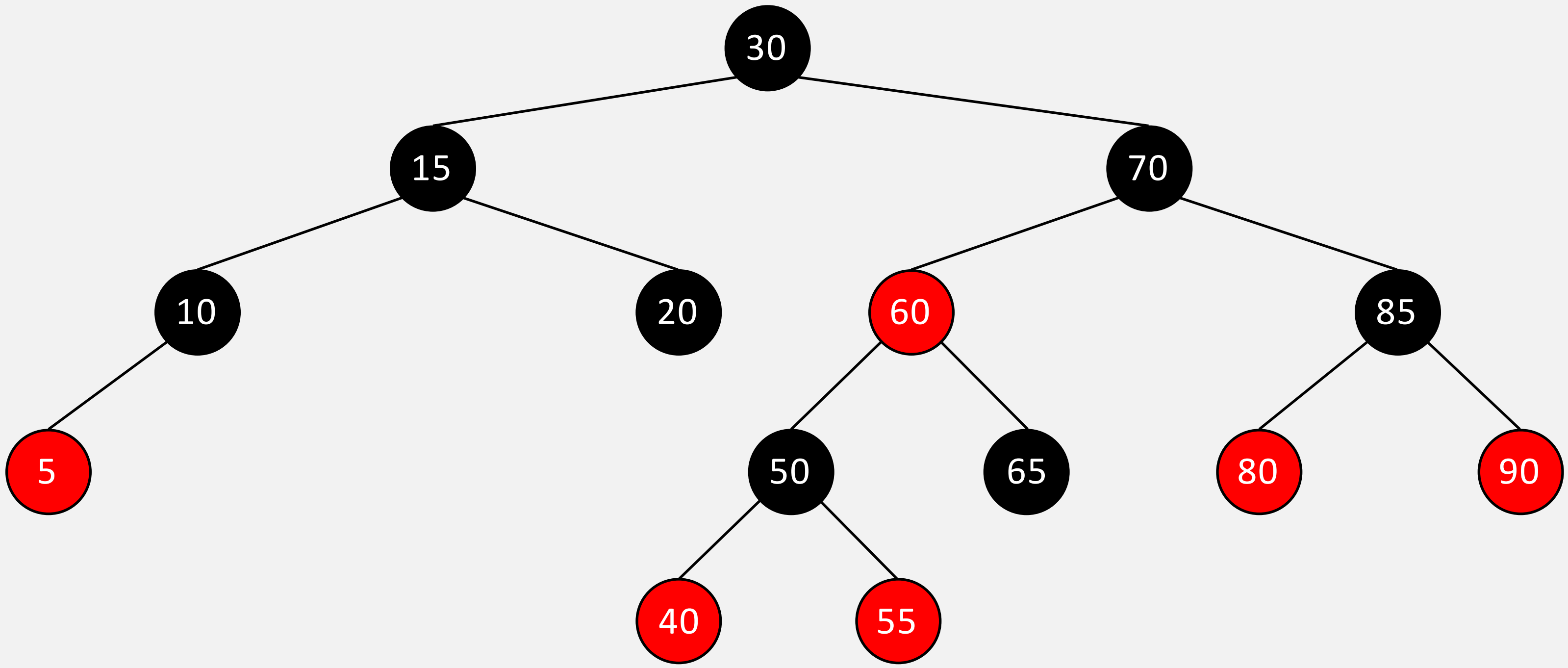






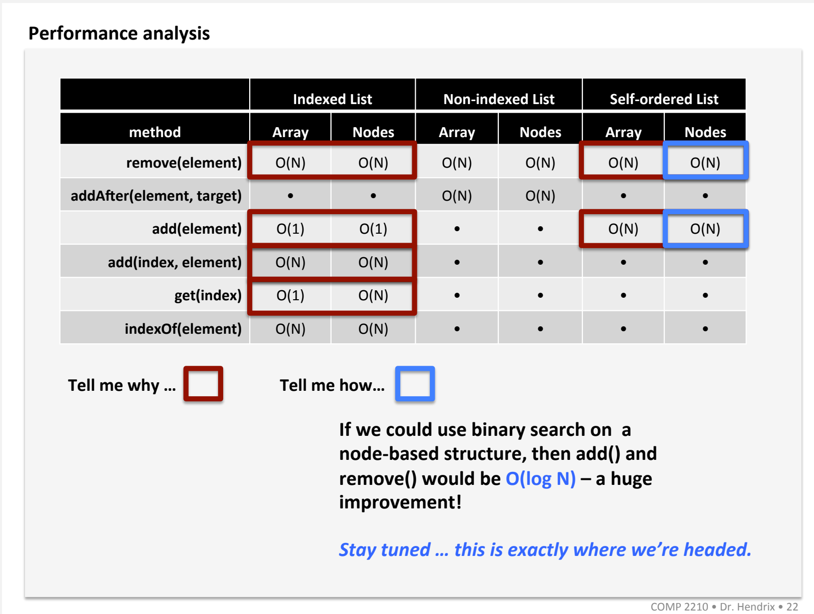
Final tree

Insert: 10, 85, 15, 70, 20, 60, 30, 50, 65, 80, 90, 40, 5, 55



Remind me: what's the point of all this?

Performance analysis of lists …



Performance analysis

method	Indexed List		Non-indexed List		Self-ordered List	
	Array	Nodes	Array	Nodes	Array	Nodes
remove(element)	O(N)	O(N)	O(N)	O(N)	O(N)	O(N)
addAfter(element, target)	•	•	O(N)	O(N)	•	•
add(element)	O(1)	O(1)	•	•	O(N)	O(N)
add(index, element)	O(N)	O(N)	•	•	•	•
get(index)	O(1)	O(N)	•	•	•	•
indexOf(element)	O(N)	O(N)	•	•	•	•

Tell me why … Tell me how…

If we could use binary search on a node-based structure, then add() and remove() would be O(log N) – a huge improvement!

Stay tuned … this is exactly where we're headed.

COMP 2210 • Dr. Hendrix • 22

Balanced binary search trees are like a structural implementation of the binary search algorithm.

So, now we can use binary search on a structure built with linked nodes.

Red-Black trees offer guaranteed O(log N) performance on all three major collection operations: add, remove, and search.

	Self-Ordered Lists		
	Array	Linked List	Red-Black Tree
add(element)	O(N)	O(N)	O(log N)
remove(element)	O(N)	O(N)	O(log N)
search(element)	O(log N)	O(N)	O(log N)

Simple BST v. AVL v. Red-Black