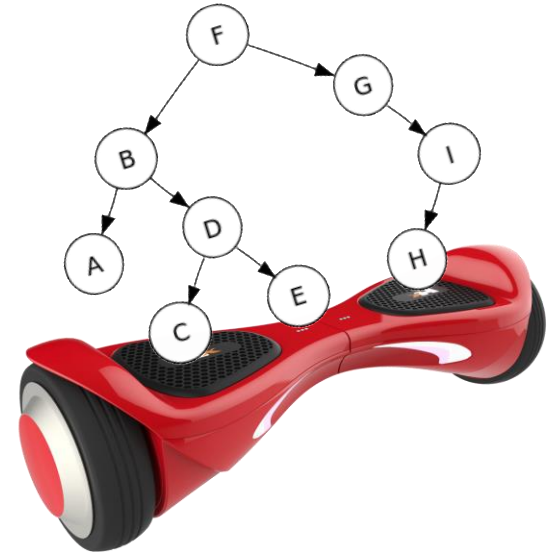


Advanced Data Structure and Algorithm

Red-Black Trees

Today

- Self-Balancing Binary Search Trees
 - **Red-Black** trees.



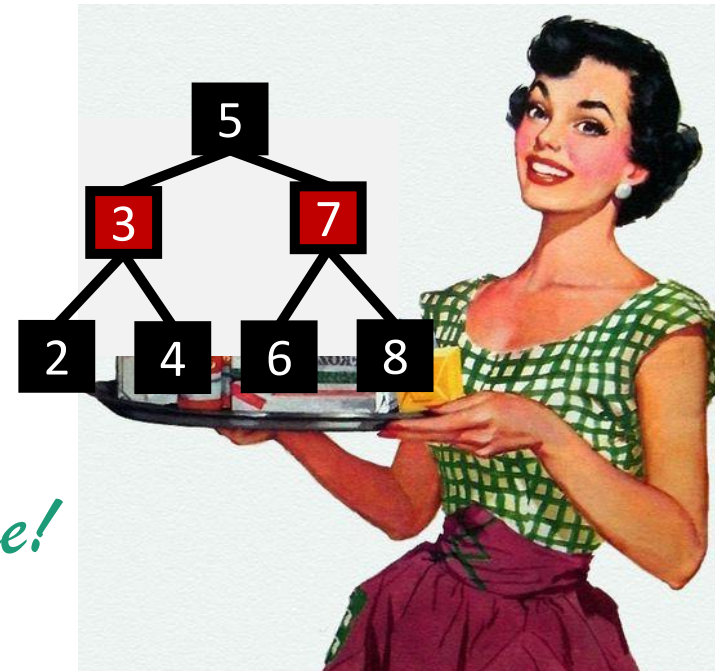
Red-Black Trees

- A Binary Search Tree that balances itself!
- No more time-consuming by-hand balancing!
- Be the envy of your friends and neighbors with the time-saving...

Red-Black tree!

*Maintain balance by stipulating that **black nodes** are balanced, and that there aren't too many **red nodes**.*

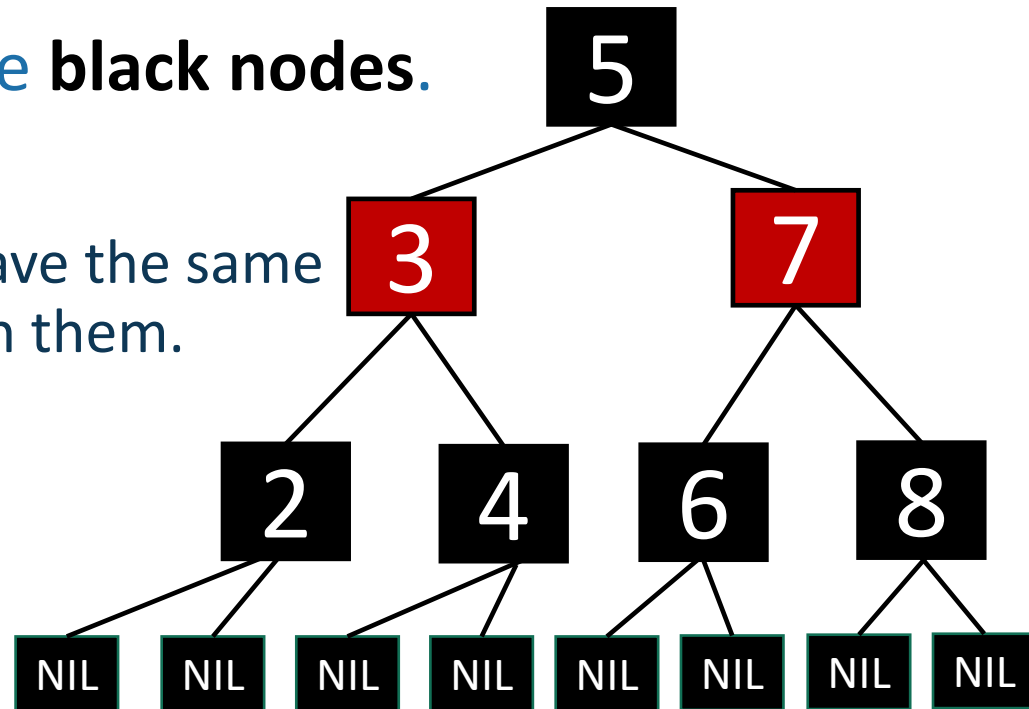
It's just good sense!



Red-Black Trees

obey the following rules (which are a proxy for balance)

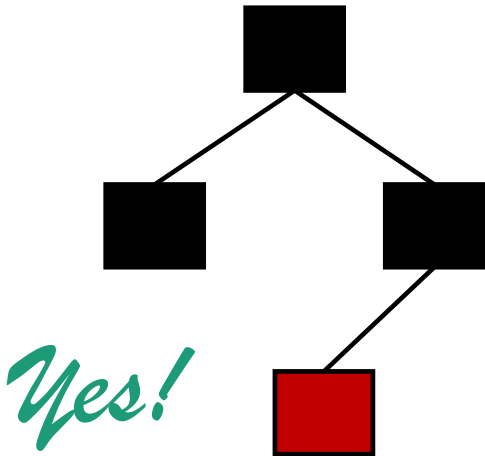
- Every node is colored **red** or **black**.
- The root node is a **black node**.
- NIL children count as **black nodes**.
- Children of a **red node** are **black nodes**.
- For all nodes x:
 - all paths from x to NIL's have the same number of **black nodes** on them.



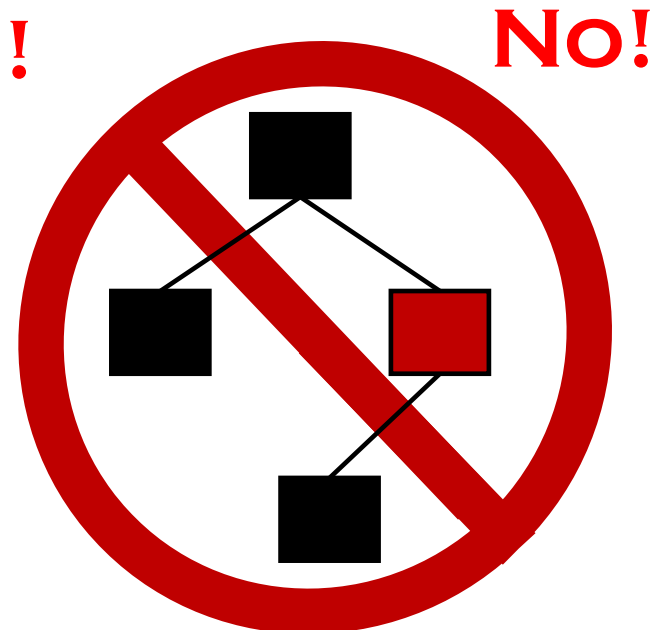
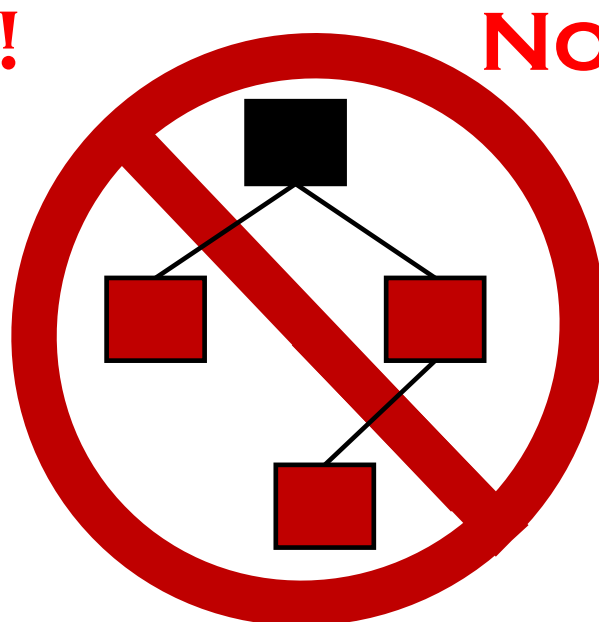
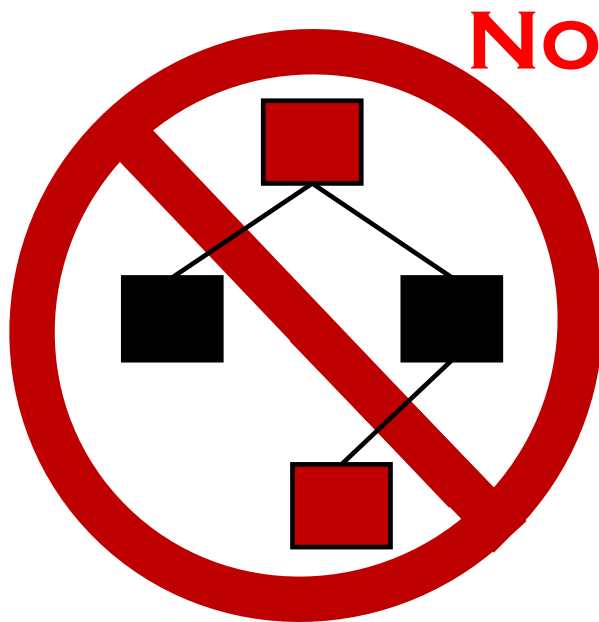
I'm not going to draw the NIL children in the future, but they are treated as black nodes.

Examples(?)

- Every node is colored **red** or **black**.
- The root node is a **black node**.
- NIL children count as **black nodes**.
- Children of a **red node** are **black nodes**.
- For all nodes x:
 - all paths from x to NIL's have the same number of **black nodes** on them.

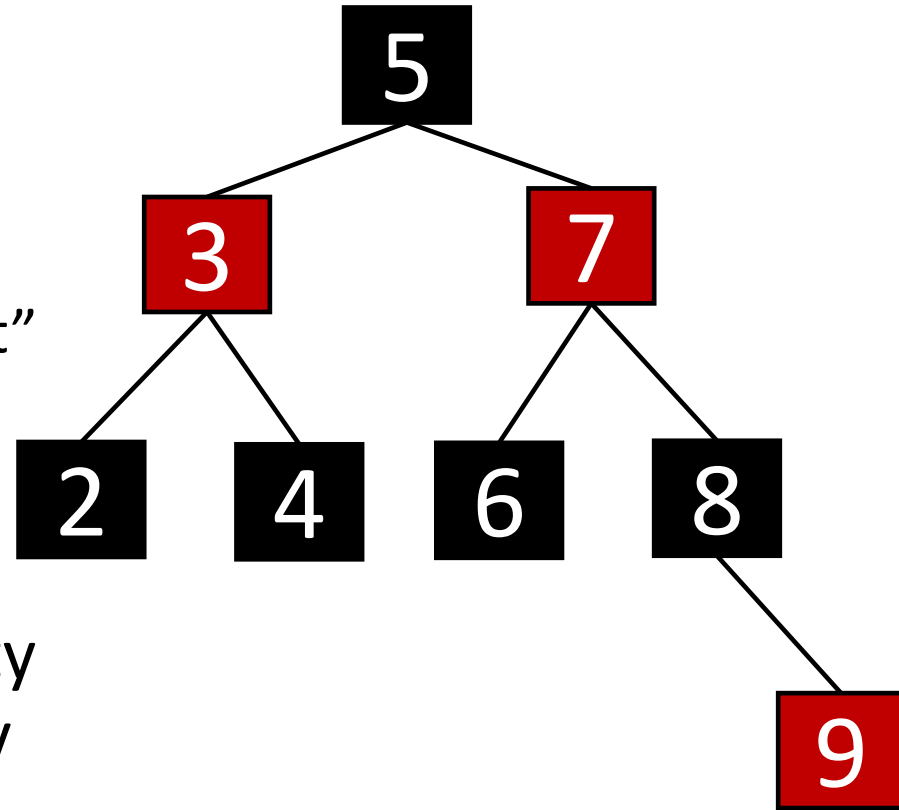


Which of these
are red-black trees?
(NIL nodes not drawn)



Why these rules??????

- This is pretty balanced.
 - The **black nodes** are balanced
 - The **red nodes** are “spread out” so they don’t mess things up too much.
- We can maintain this property as we insert/delete nodes, by using rotations.



This is the really clever idea!

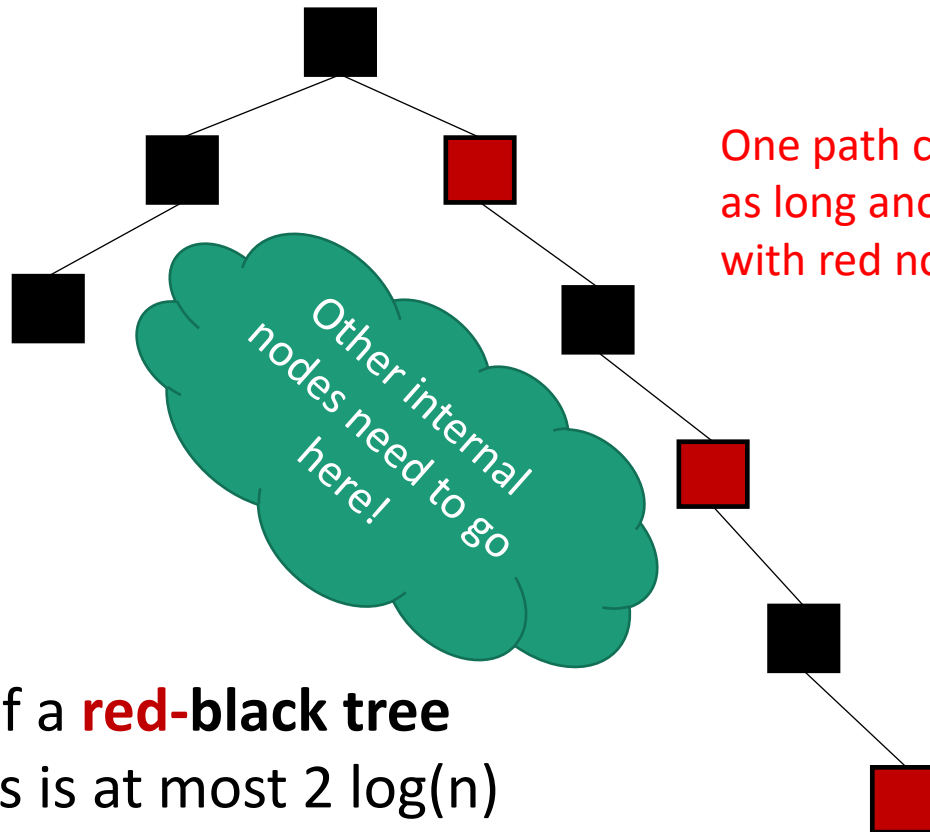
This **Red-Black** structure is a **proxy for balance**.

It’s just weaker than perfect balance, but we can actually maintain it!



This is “pretty balanced”

- To see why, intuitively, let's try to build a Red-Black Tree that's unbalanced.



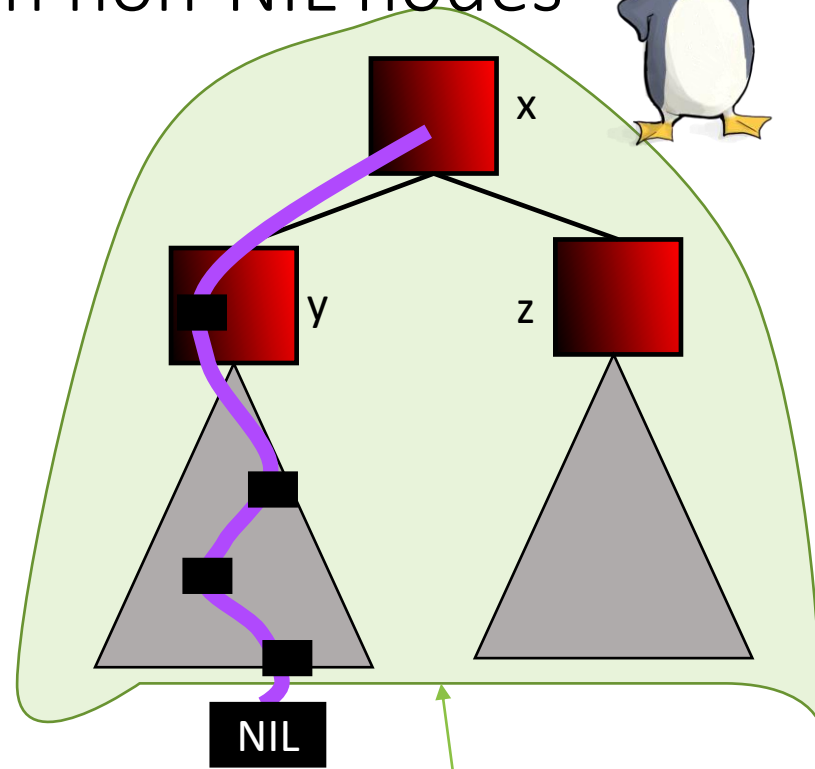
Conjecture:
the height of a **red-black tree**
with n nodes is at most $2 \log(n)$



The height of a RB-tree with n non-NIL nodes is at most $2\log(n + 1)$



- Define $b(x)$ to be the number of black nodes in any path from x to NIL.
 - (excluding x , including NIL).
- Claim:
 - There are at least $2^{b(x)} - 1$ non-NIL nodes in the subtree underneath x . (Including x).
- [Proof by induction]



Claim: at least $2^{b(x)} - 1$ nodes in this WHOLE subtree (of any color).

Then:

$$n \geq 2^{b(\text{root})} - 1 \quad \text{using the Claim}$$

$$\geq 2^{\text{height}/2} - 1 \quad b(\text{root}) \geq \text{height}/2 \text{ because of RBTree rules.}$$

Rearranging:

$$n + 1 \geq 2^{\text{height}/2} \Rightarrow \text{height} \leq 2\log(n + 1)$$

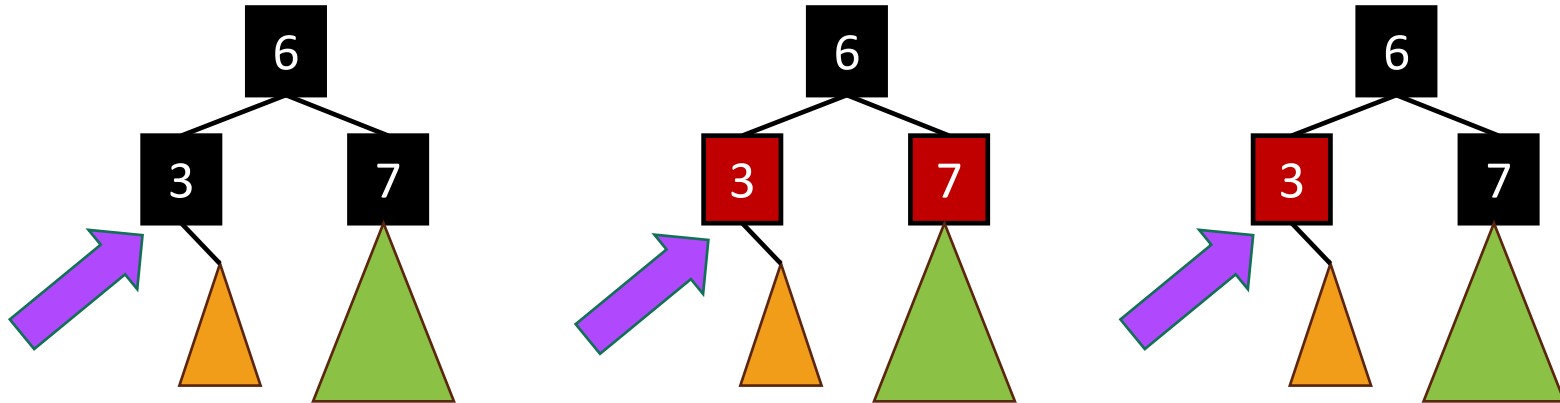
This is great!

- SEARCH in an RBTree is immediately $O(\log(n))$, since the depth of an RBTree is $O(\log(n))$.
- What about INSERT/DELETE?
 - Turns out, you can INSERT and DELETE items from an RBTree in time $O(\log(n))$, while maintaining the RBTree property.

INSERT/DELETE

- INSERT/DELETE for RBTrees
 - You should know what the “proxy for balance” property is and why it ensures approximate balance.

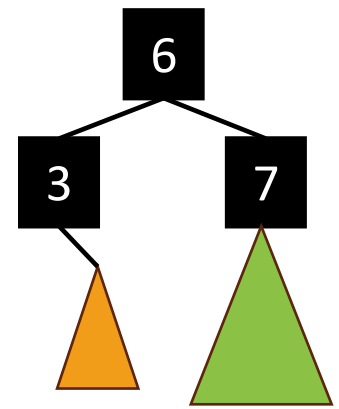
INSERT: Many cases



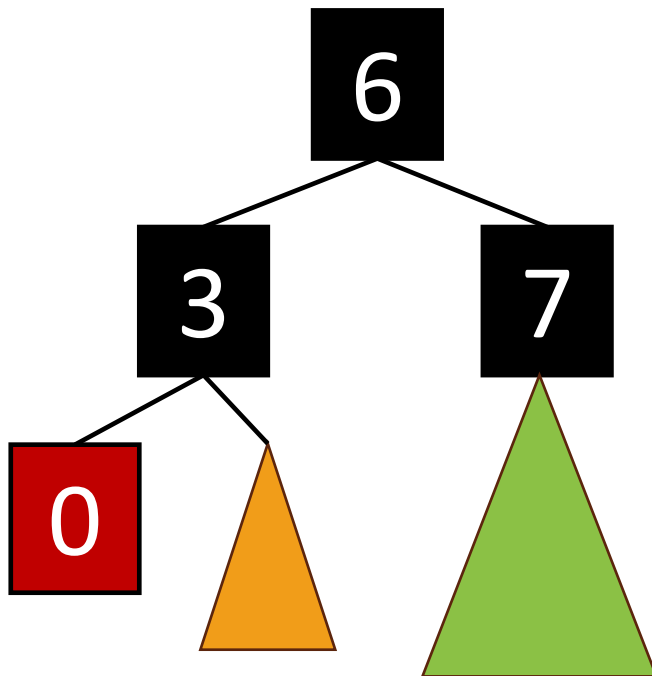
- Suppose we want to insert 0 **here**.
- There are 3 “important” cases for different colorings of the existing tree, and there are 9 more cases for all of the various symmetries of these 3 cases.

INSERT: Case 1

- Make a new **red node**.
- Insert it as you would normally.



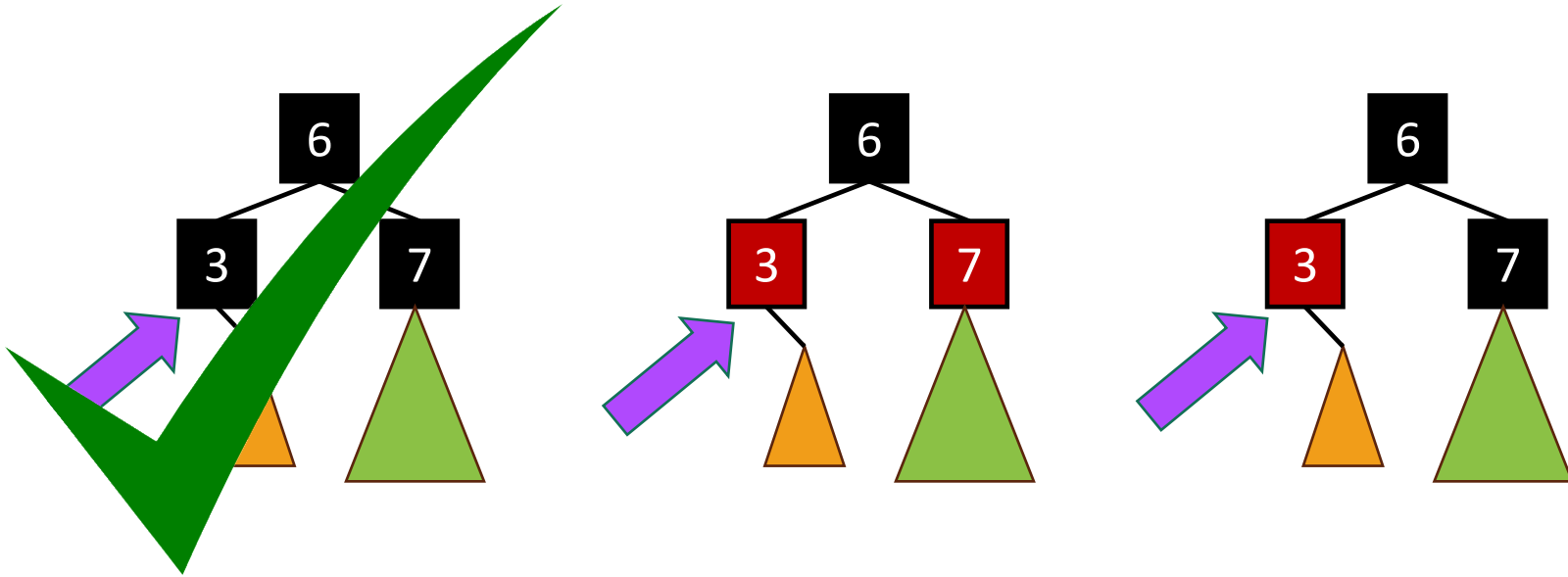
What if it looks like this?



Example: insert 0



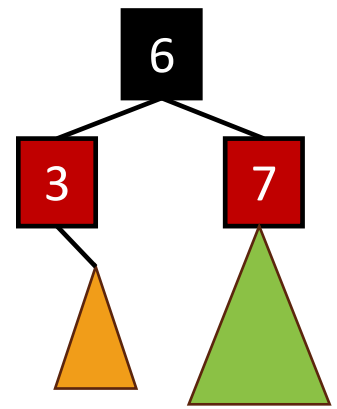
INSERT: Many cases



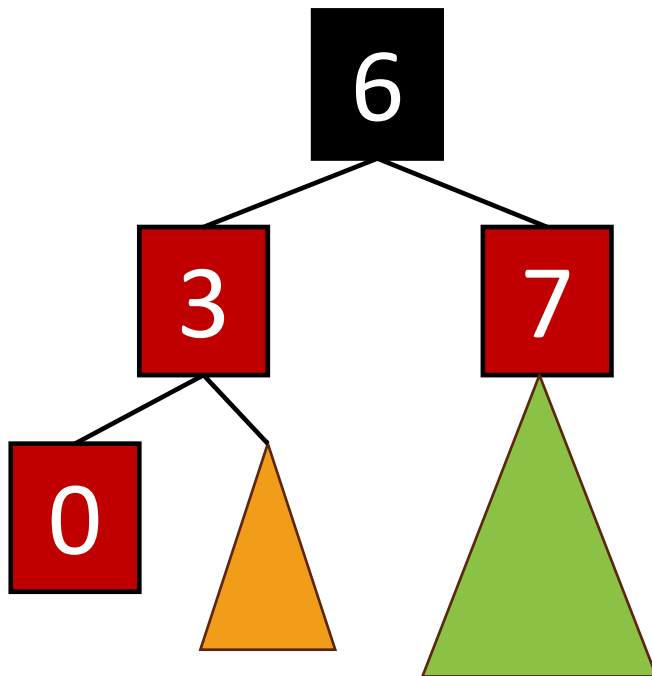
- Suppose we want to insert 0 **here**.
- There are 3 “important” cases for different colorings of the existing tree, and there are 9 more cases for all of the various symmetries of these 3 cases.

INSERT: Case 2

- Make a new **red node**.
- Insert it as you would normally.
- Fix things up if needed.



What if it looks like this?

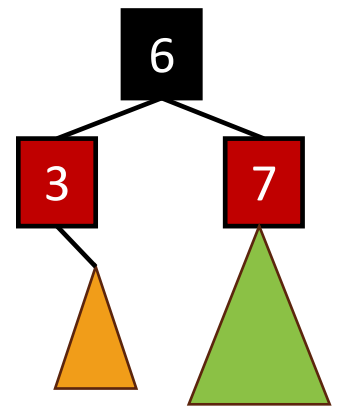
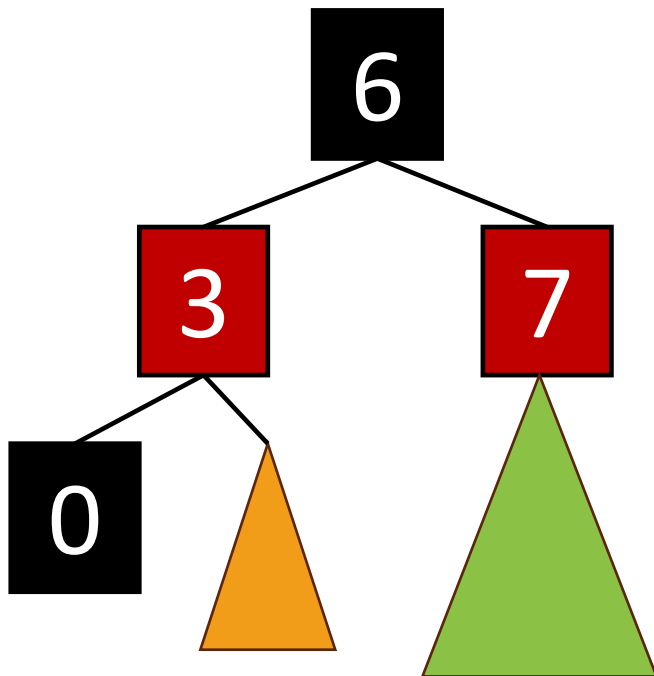


Example: insert 0



INSERT: Case 2

- Make a new **red node**.
- Insert it as you would normally.
- Fix things up if needed.



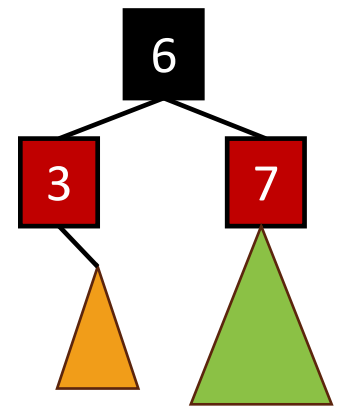
What if it looks like this?

Example: insert 0

Can't we just insert 0 as a **black node**?

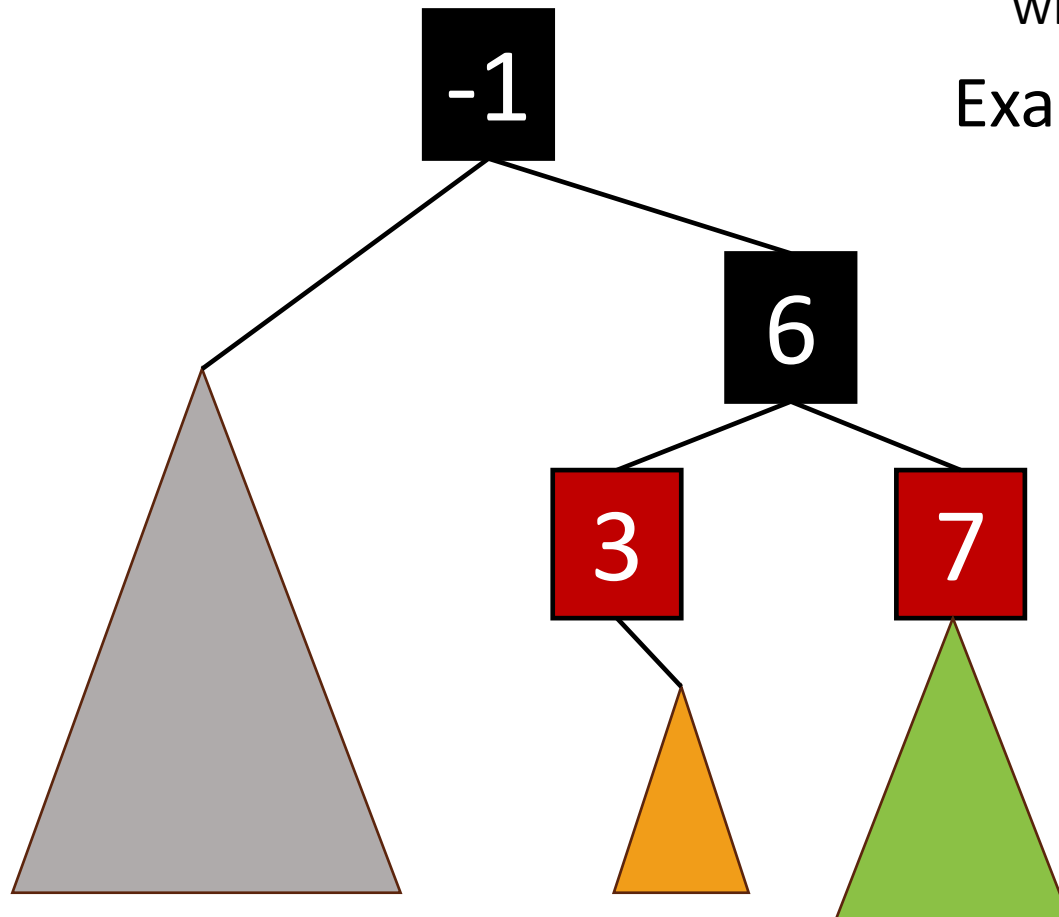


We need a bit more context



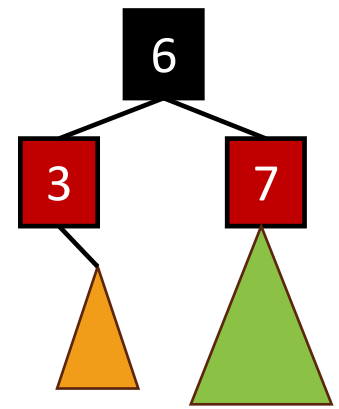
What if it looks like this?

Example: insert 0



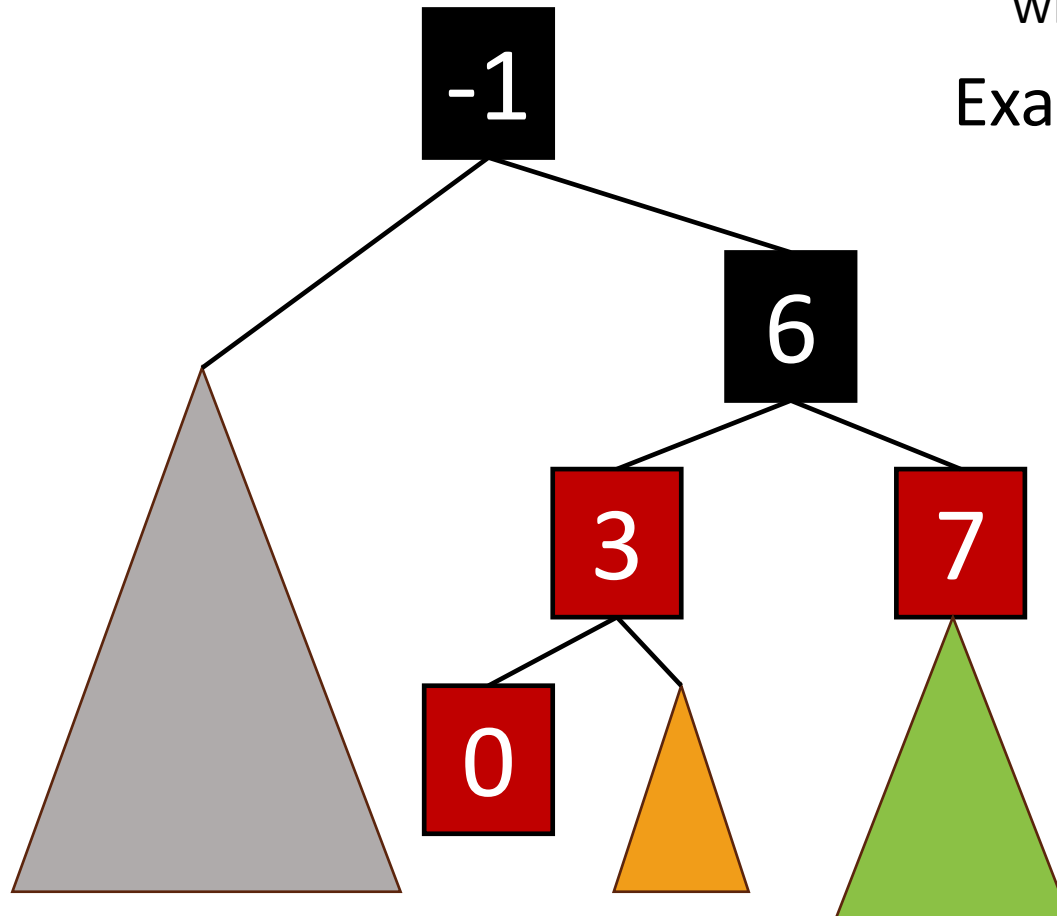
We need a bit more context

- Add 0 as a red node.



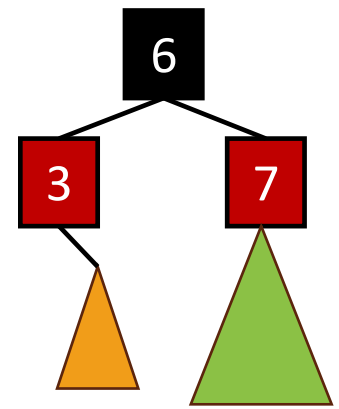
What if it looks like this?

Example: insert 0



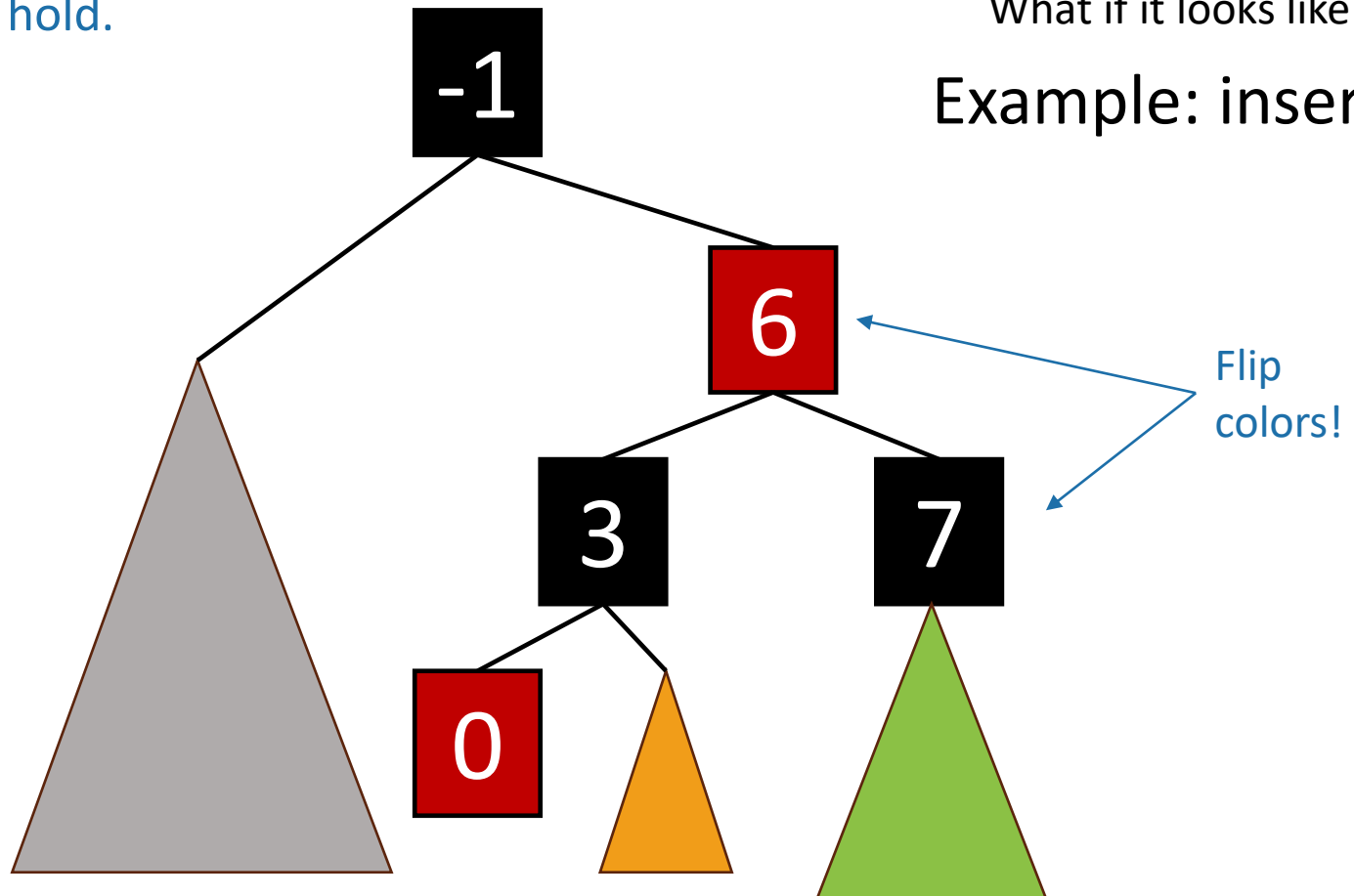
We need a bit more context

- Add 0 as a red node.
- **Claim:** RB-Tree properties still hold.

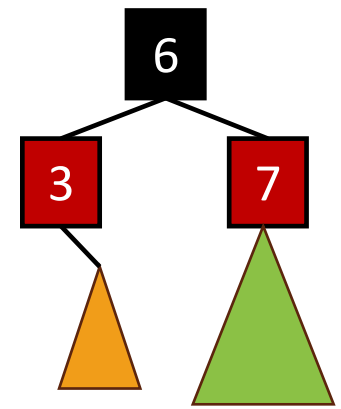
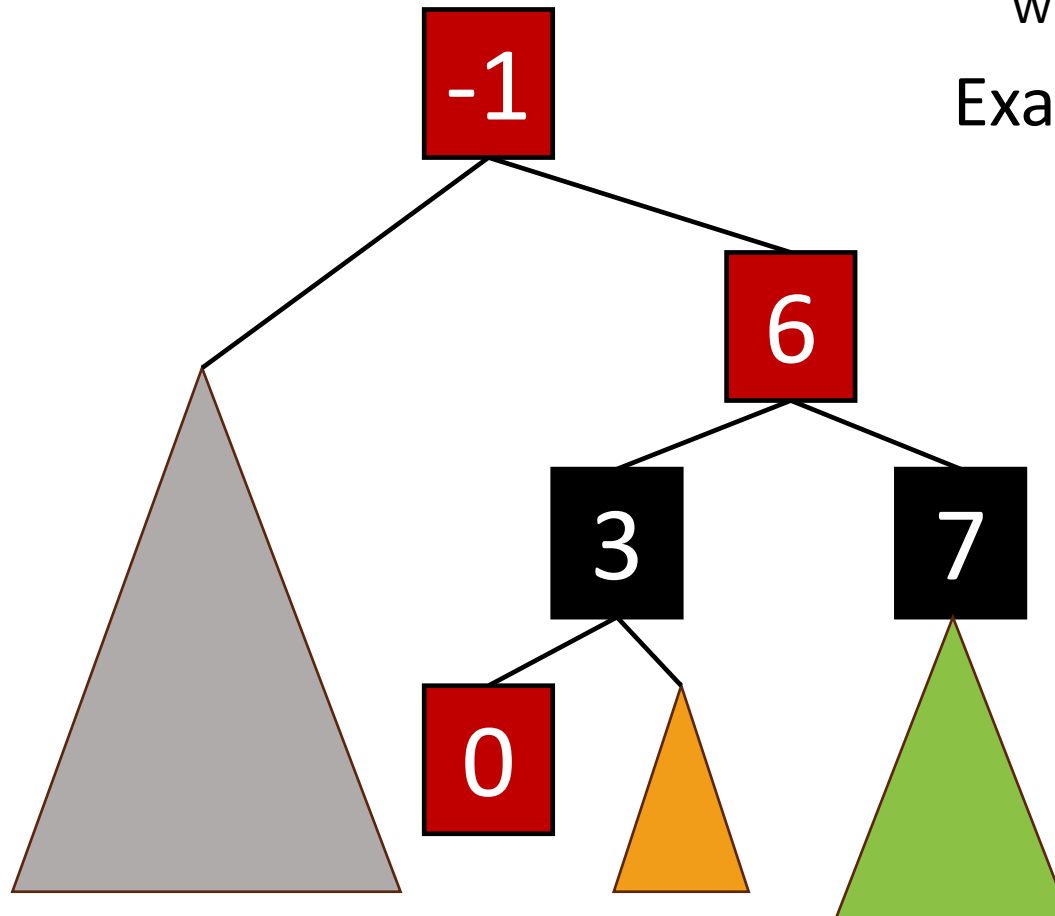
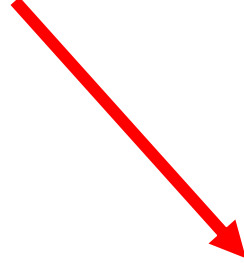


What if it looks like this?

Example: insert 0



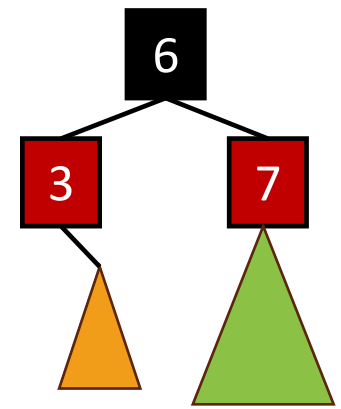
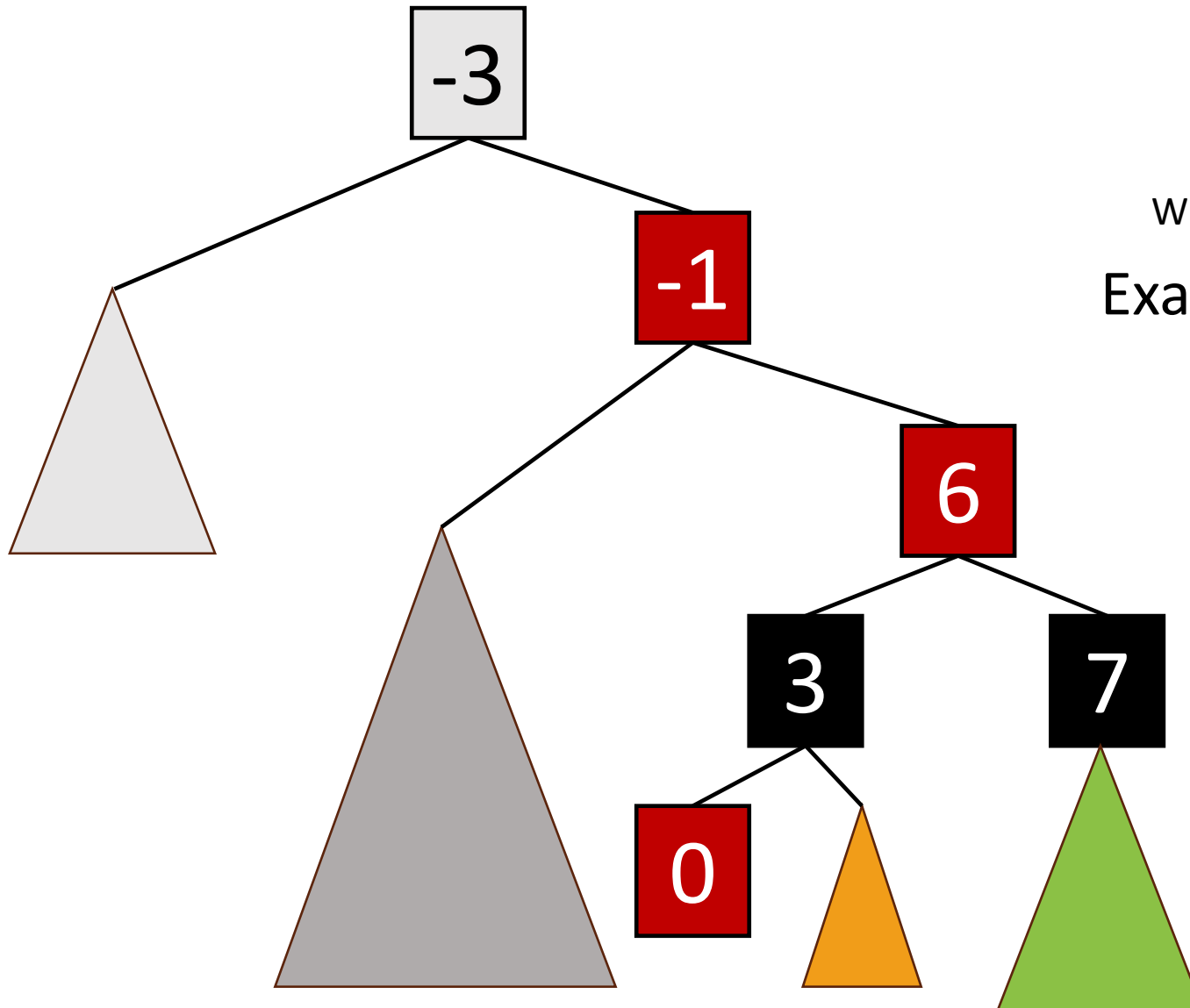
But what if **that** was red?



What if it looks like this?

Example: insert 0

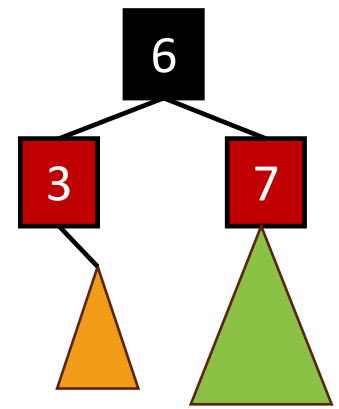
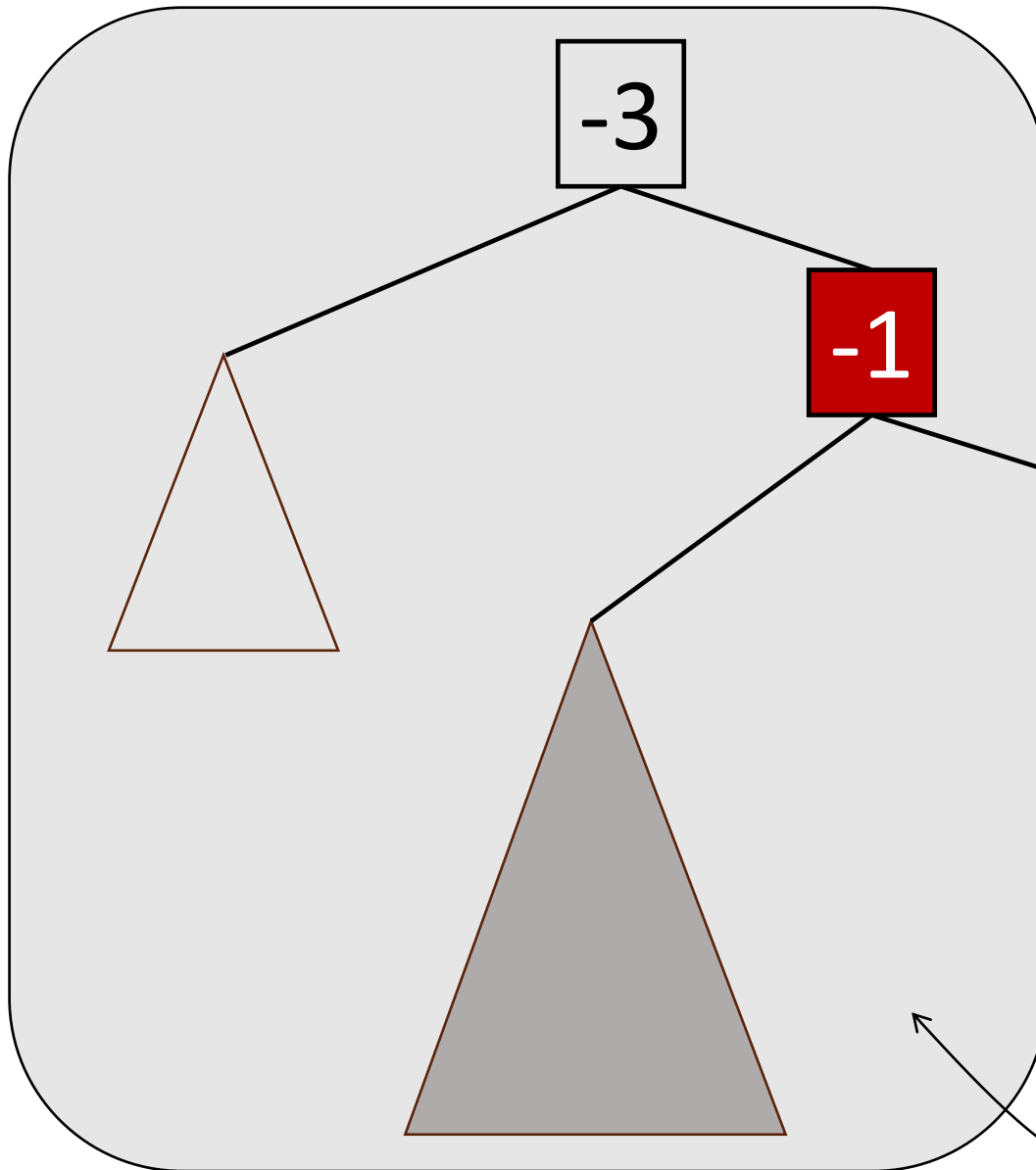
More context...



What if it looks like this?

Example: insert 0

More context...



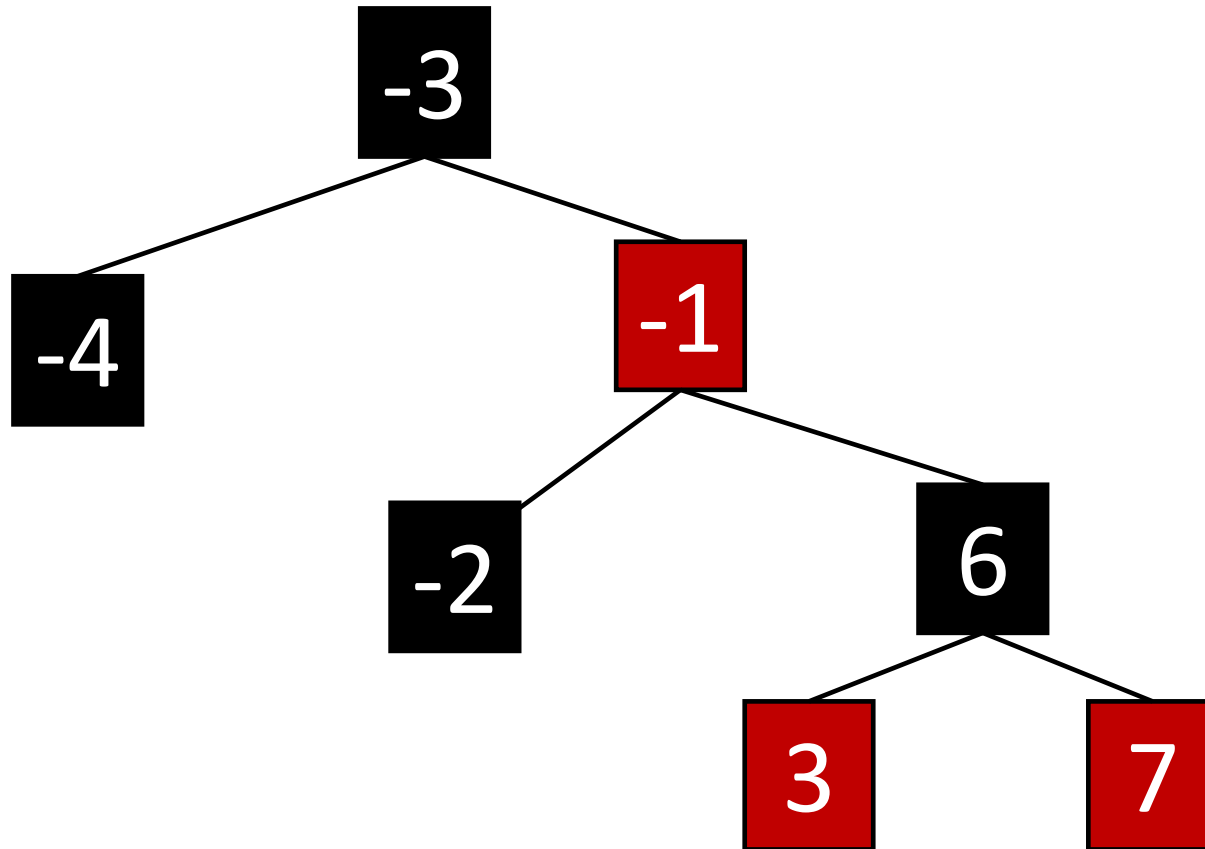
What if it looks like this?

Example: insert 0

Now we're basically
inserting 6 into some
smaller tree. Recurse!

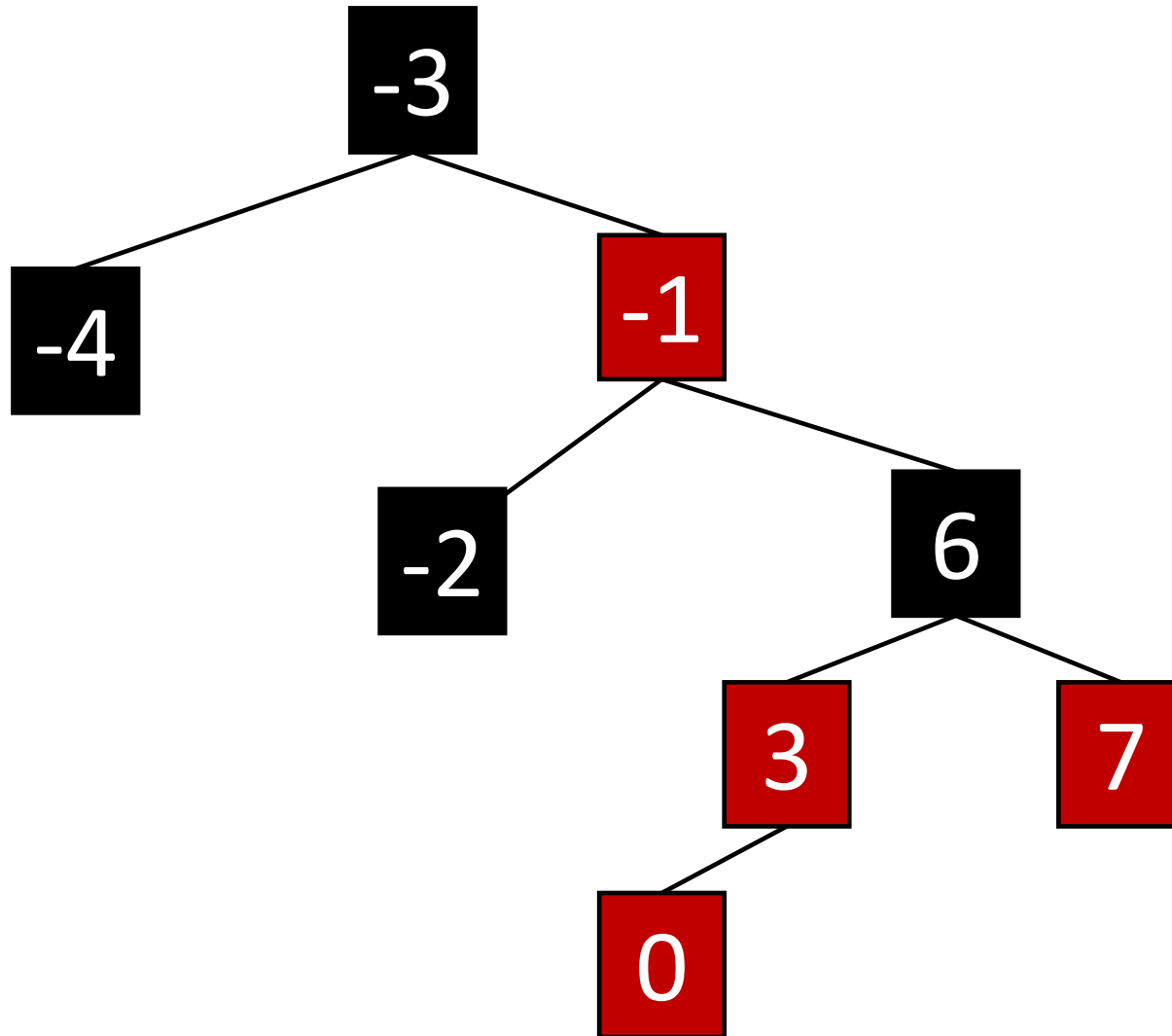
This one!

Example, part I

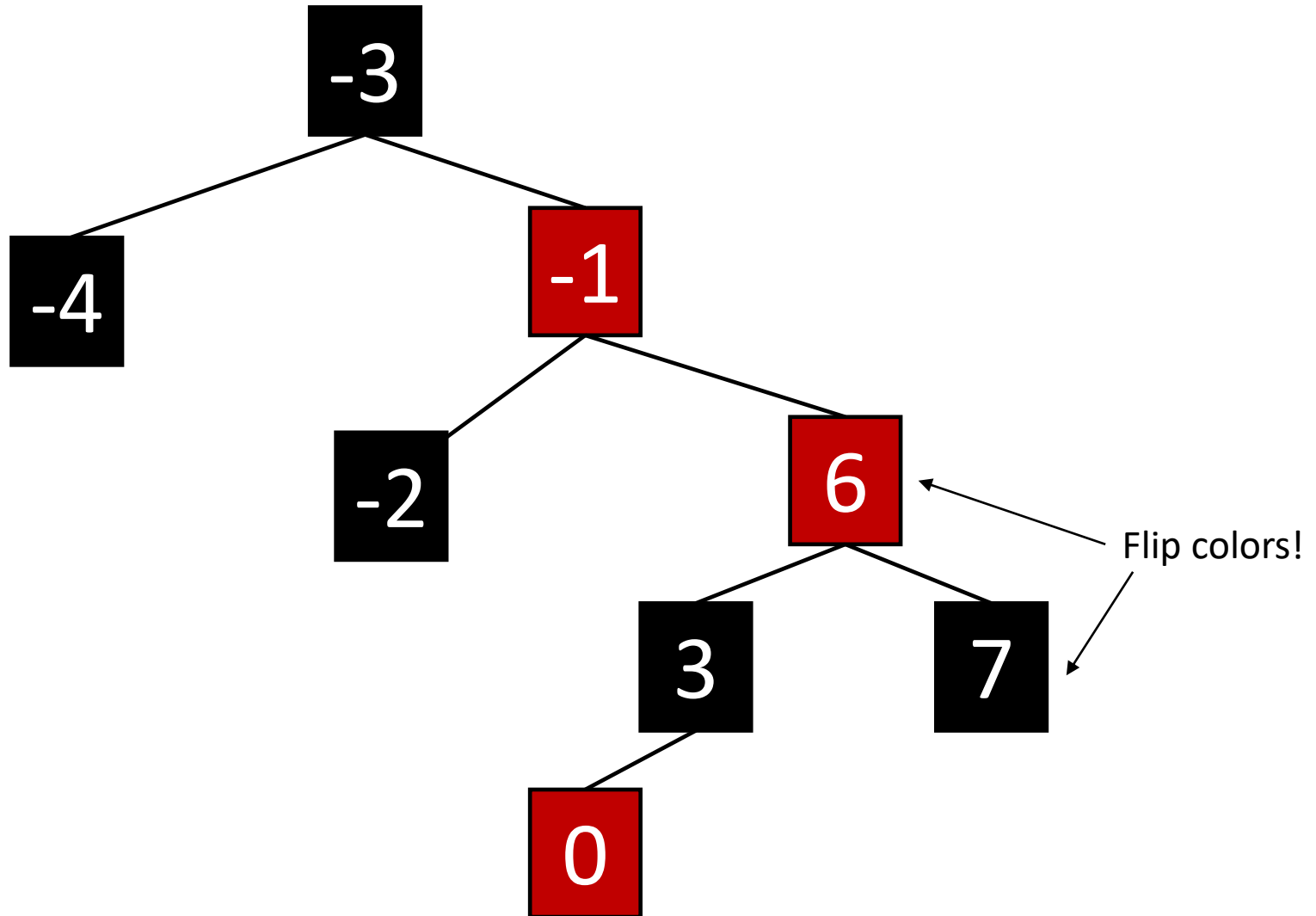


Want to
insert 0
here.

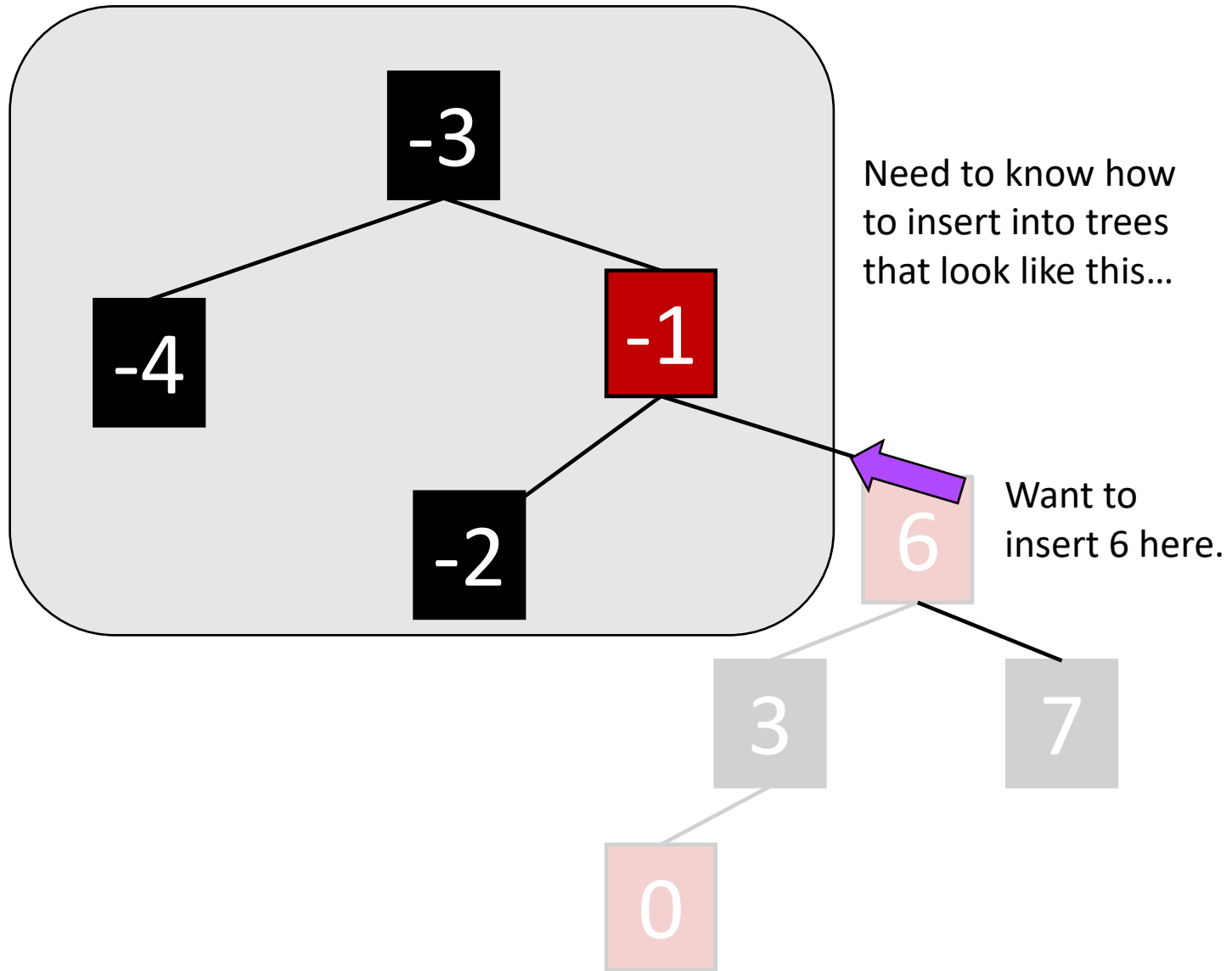
Example, part I



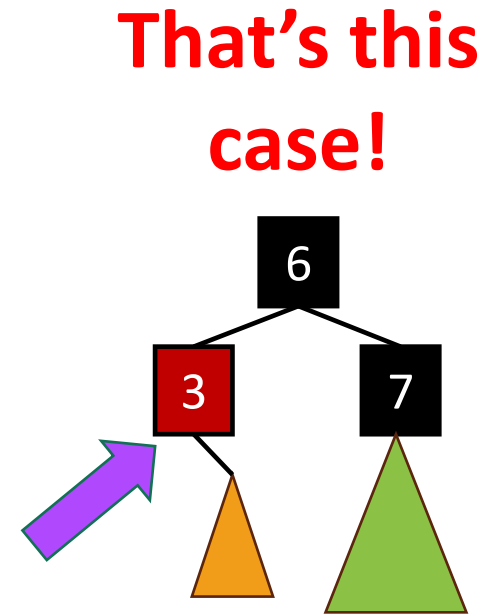
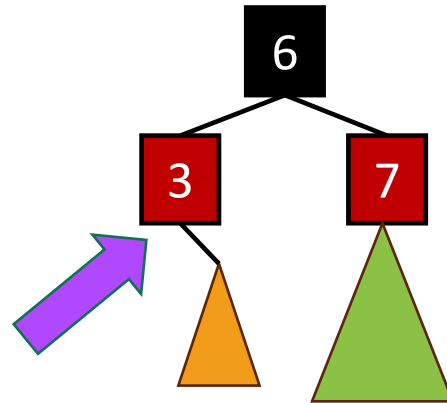
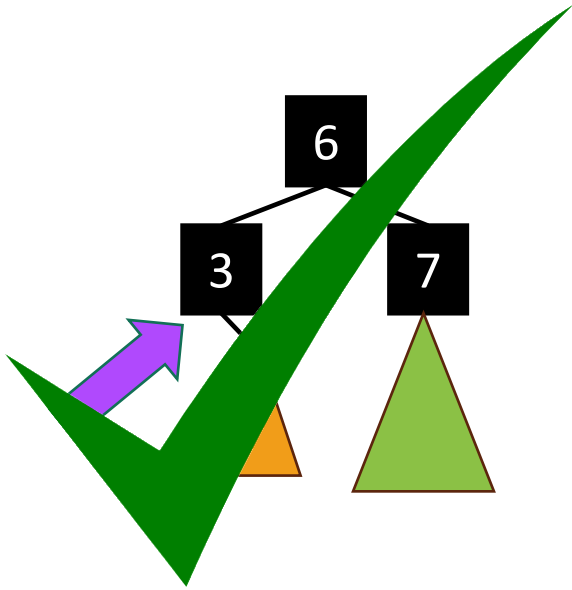
Example, part I



Example, part I



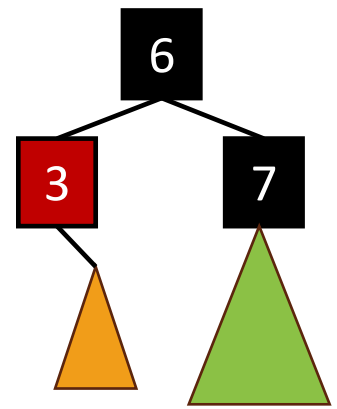
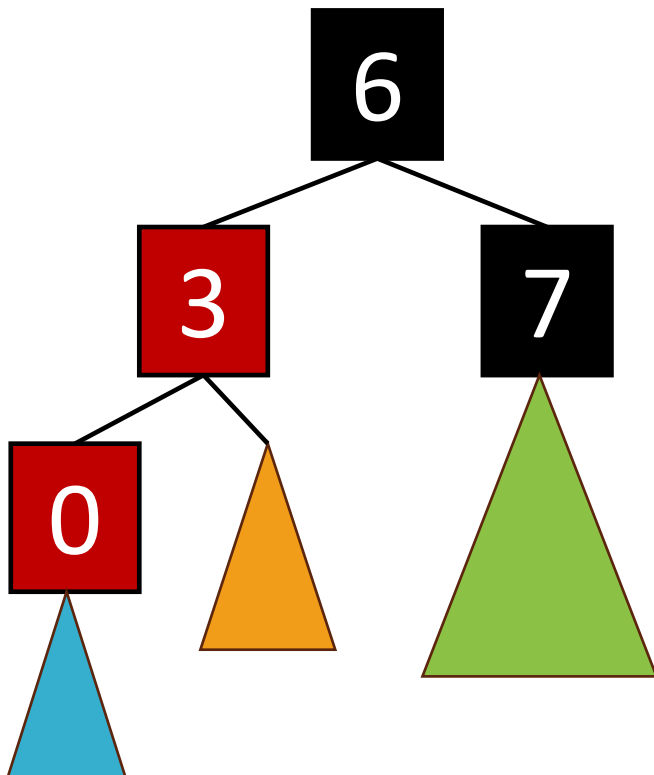
INSERT: Many cases



- Suppose we want to insert 0 **here**.
- There are 3 “important” cases for different colorings of the existing tree, and there are 9 more cases for all of the various symmetries of these 3 cases.

INSERT: Case 3

- Make a new **red node**.
- Insert it as you would normally.
- Fix things up if needed.



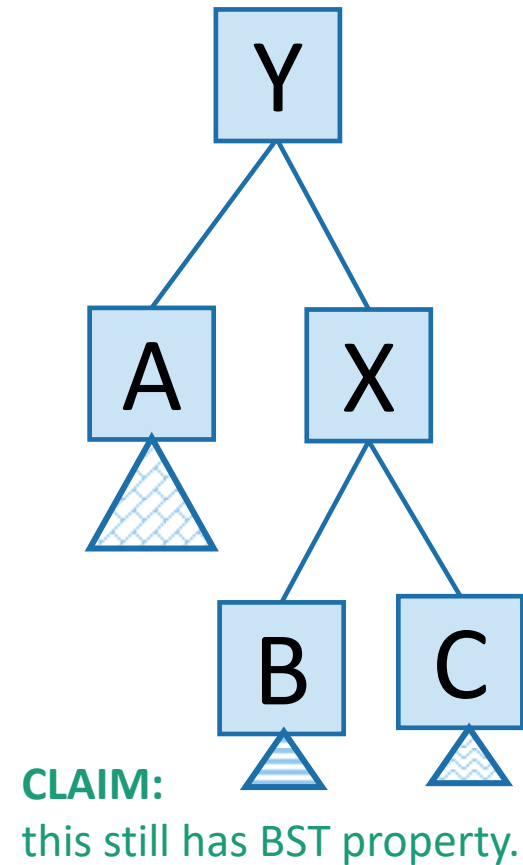
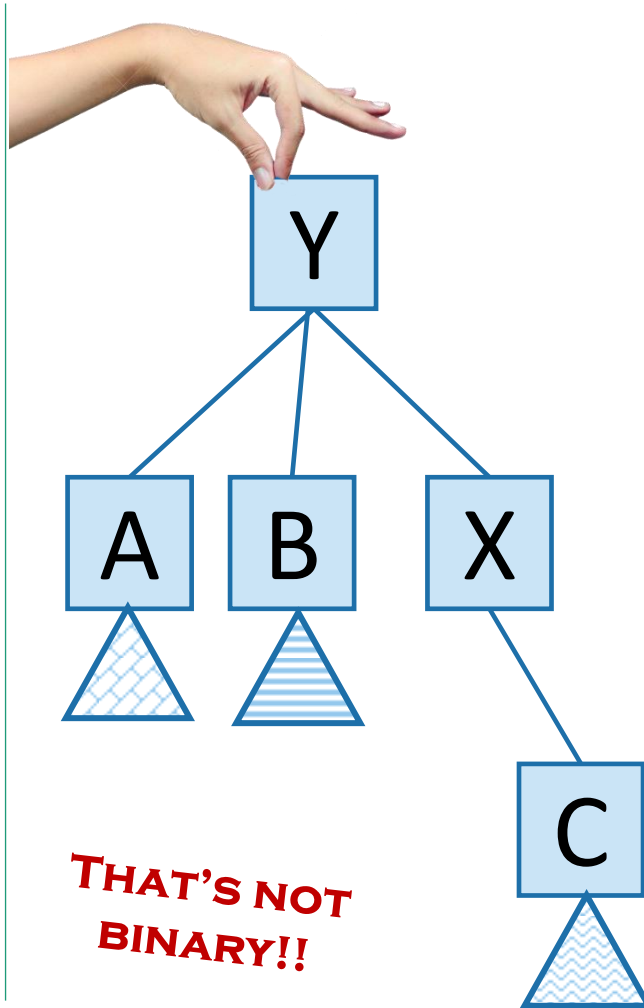
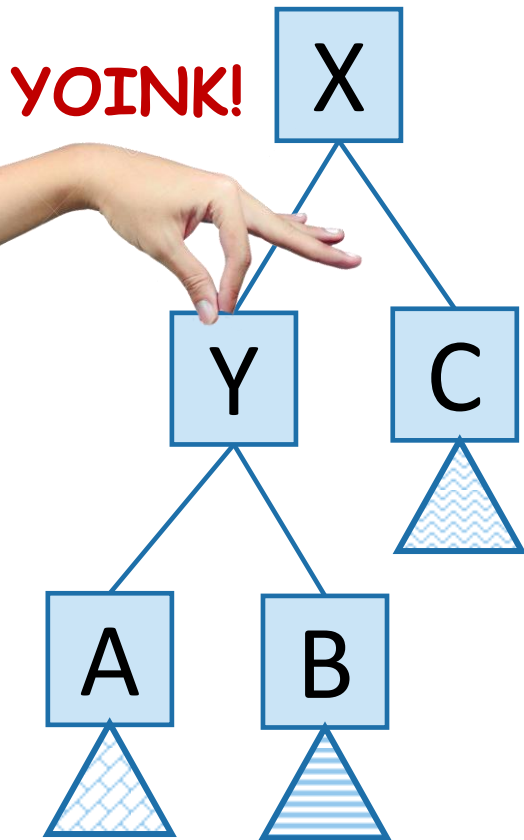
What if it looks like this?

Example: Insert 0.

- Maybe with a subtree below it.

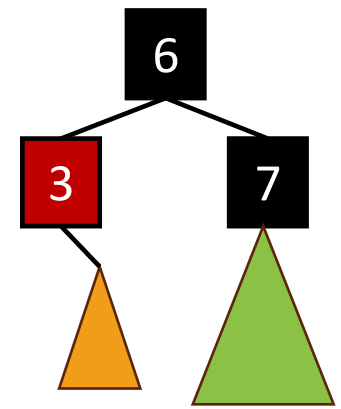
Recall Rotations

- Maintain Binary Search Tree (BST) property, while moving stuff around.



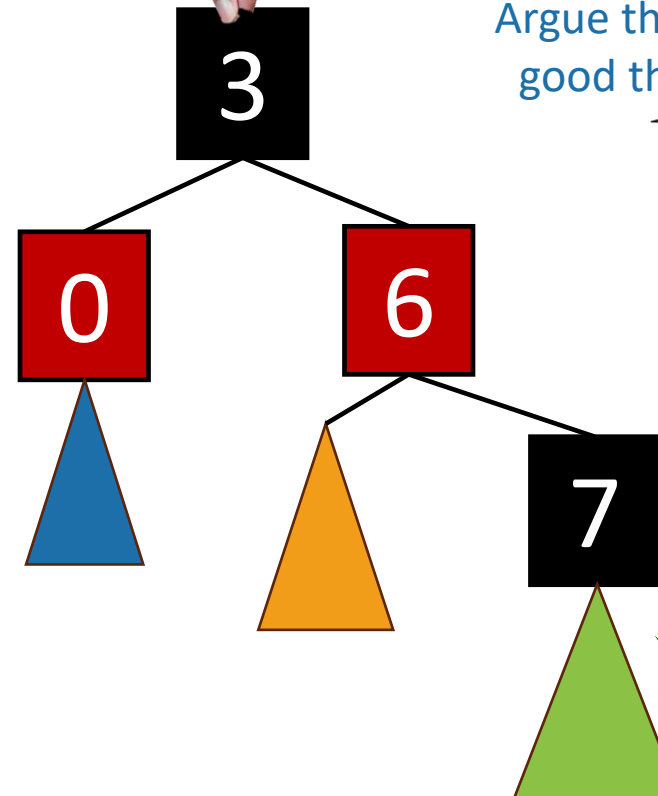
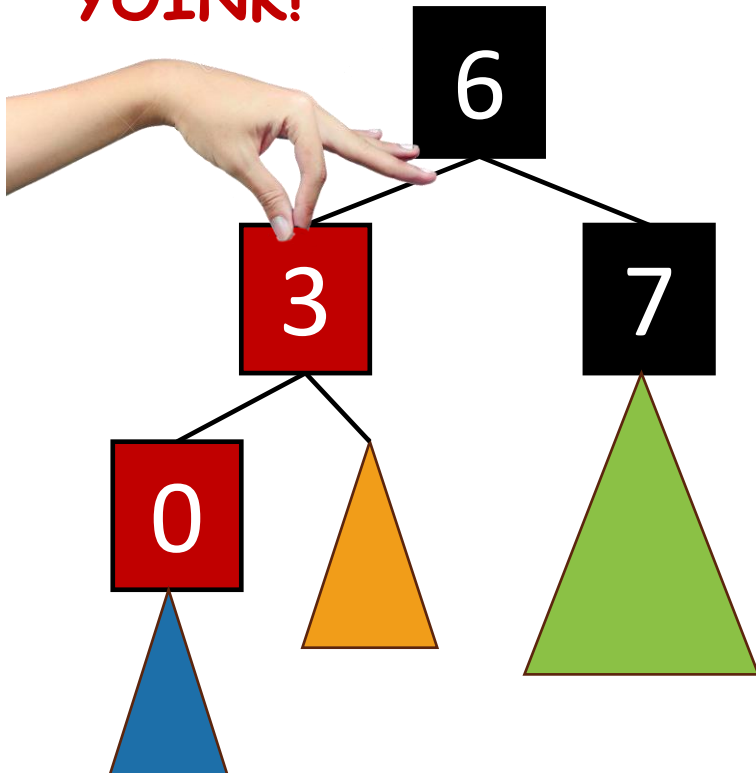
Inserting into a Red-Black Tree

- Make a new **red node**.
- Insert it as you would normally.
- Fix things up if needed.



What if it looks like this?

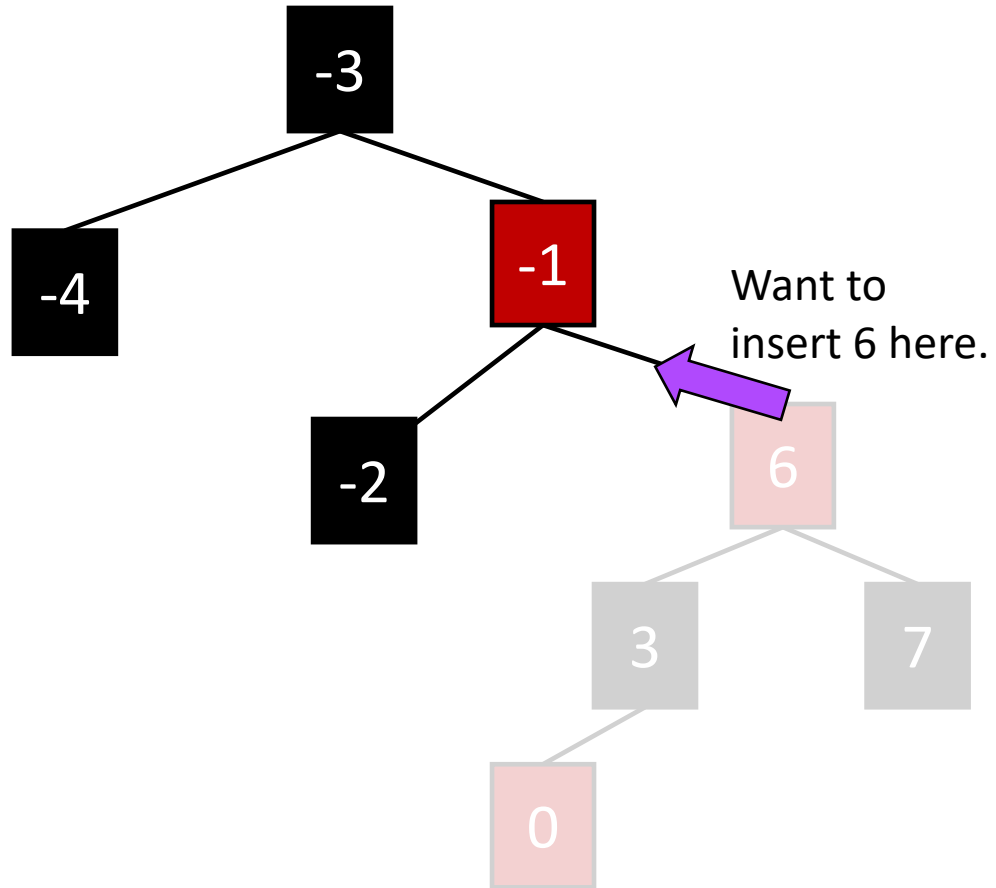
YOINK!



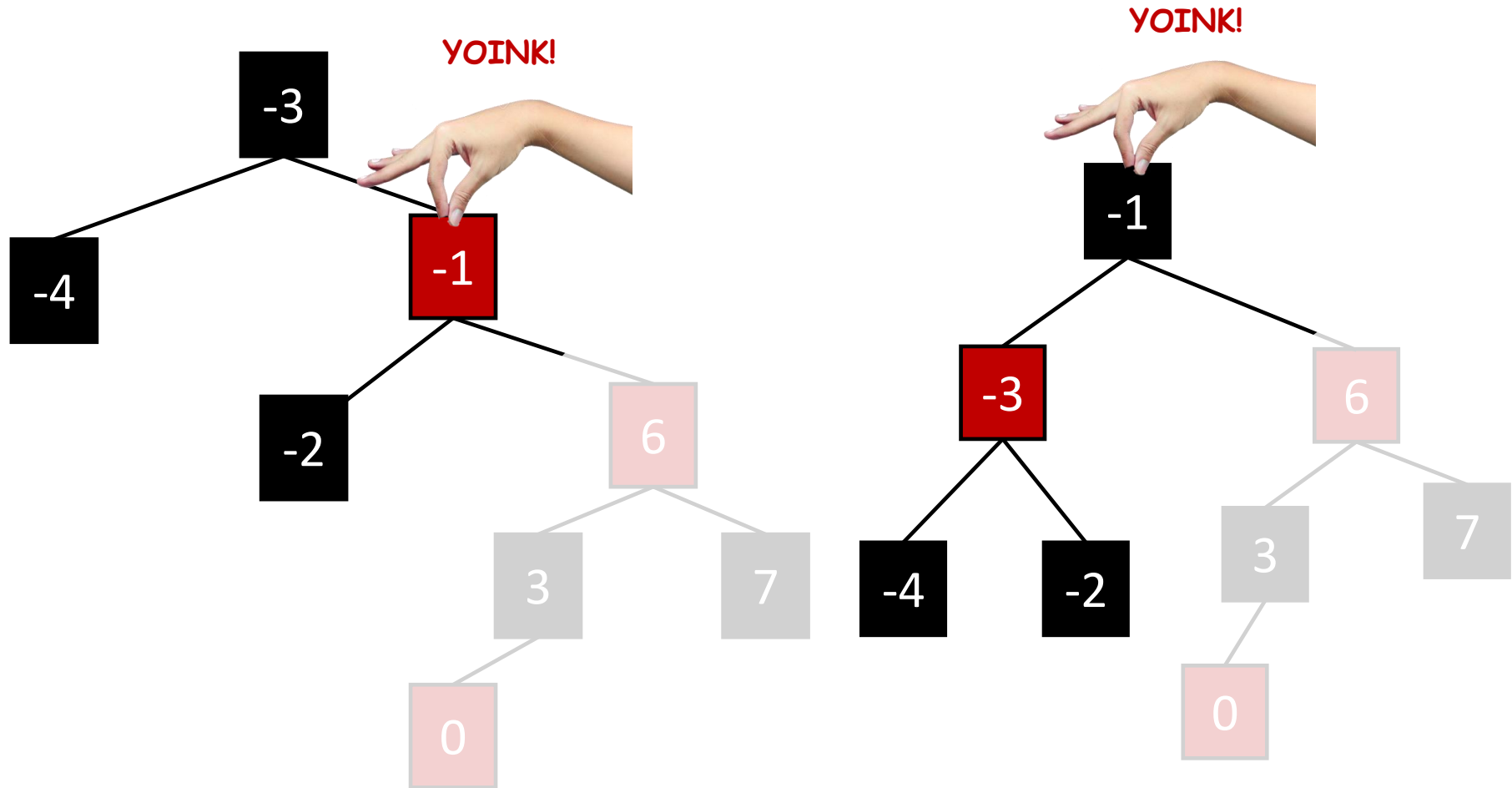
Argue that this is a good thing to do!



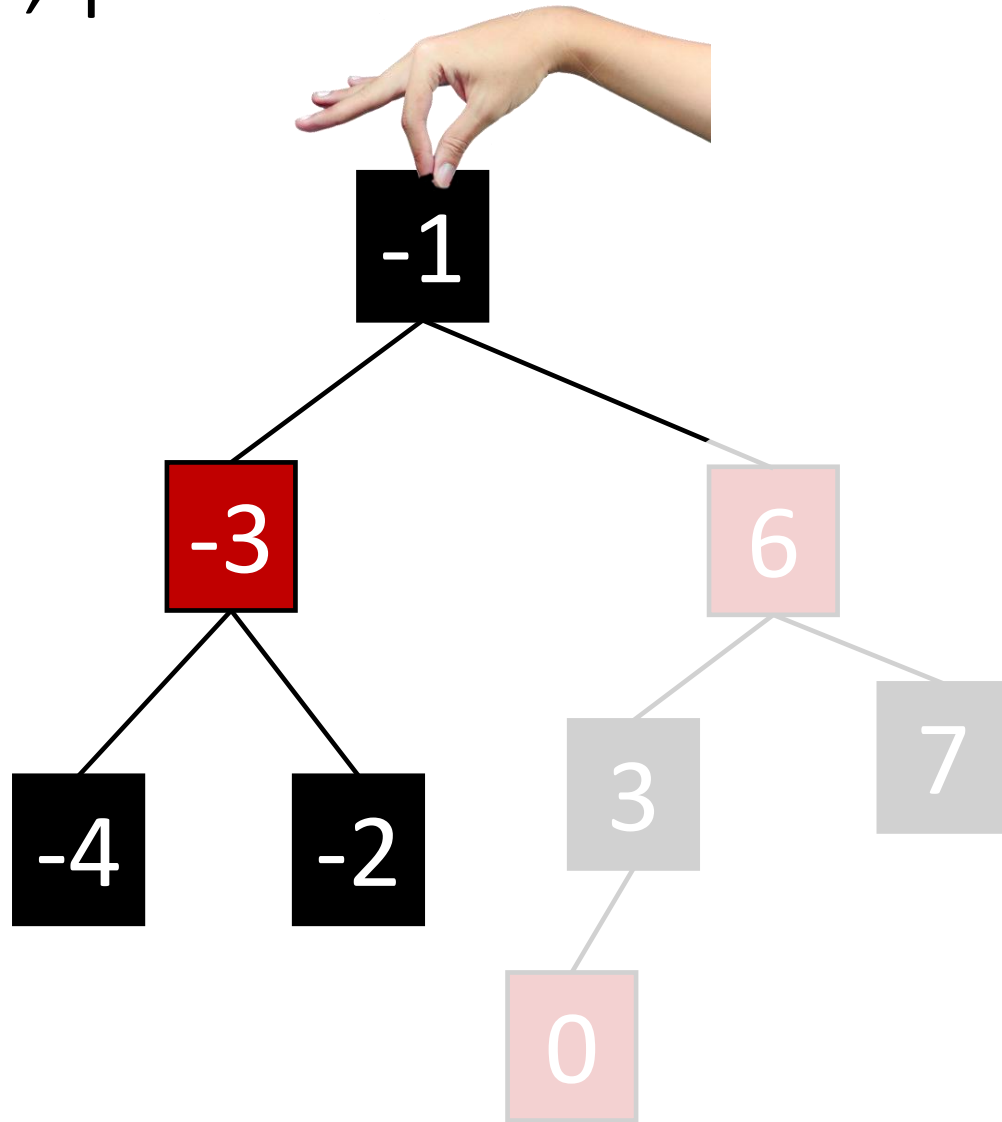
Example, part 2



Example, part 2

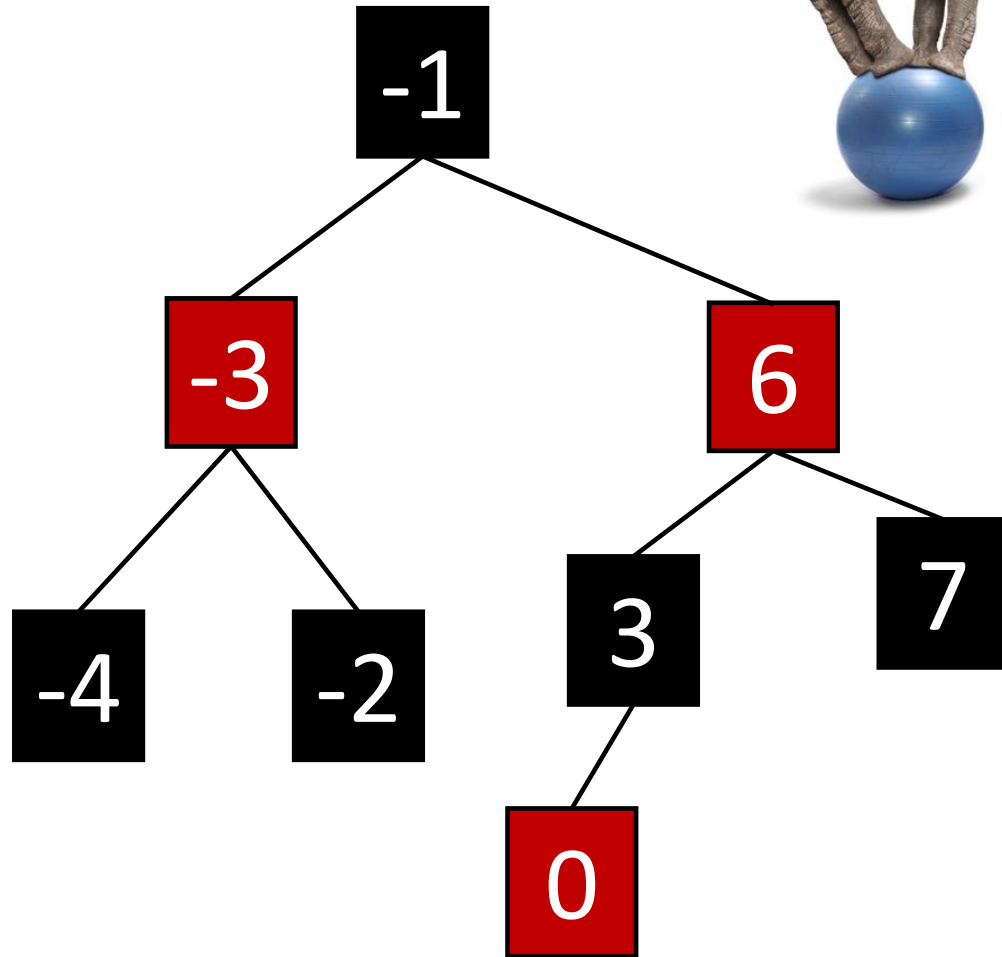


Example, part 2 **YOINK!**

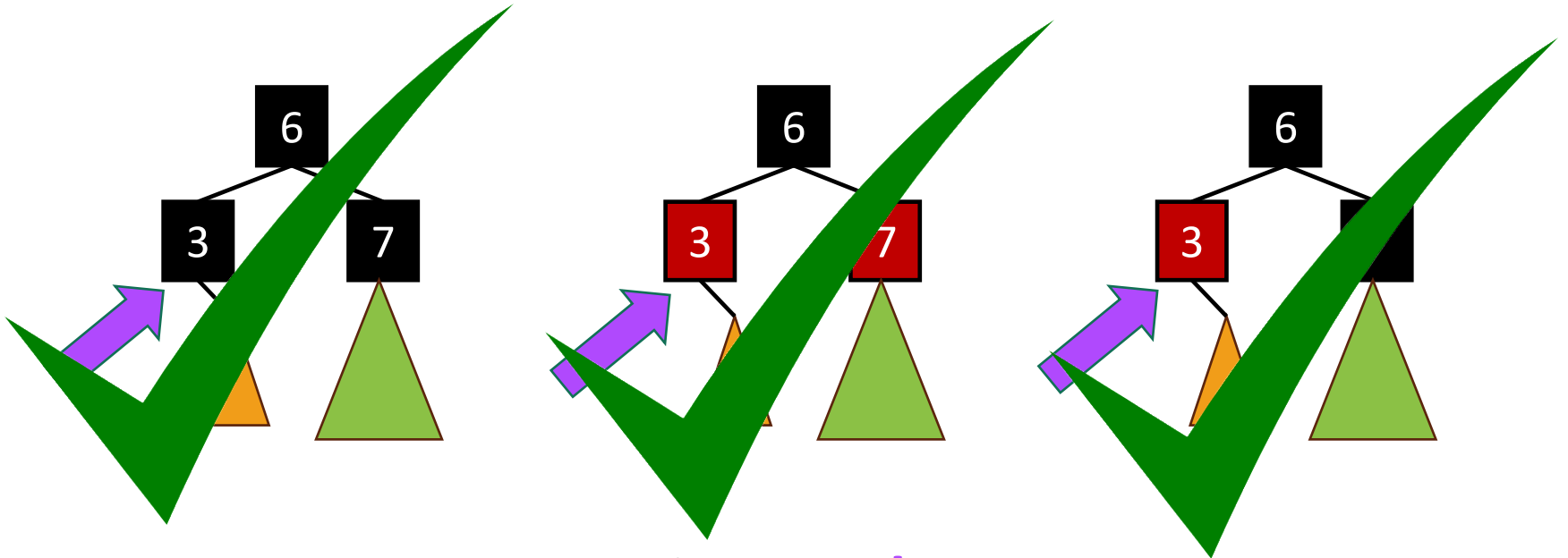


Example, part 2

TA-DA!



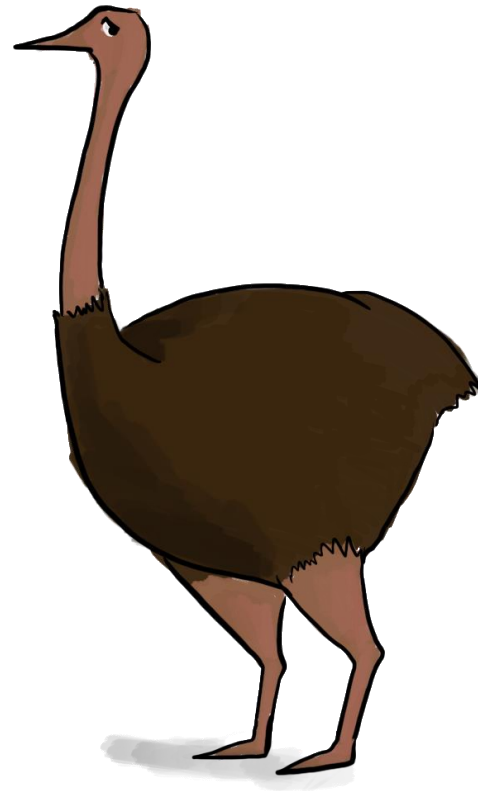
Many cases



- Suppose we want to insert 0 **here**.
- There are 3 “important” cases for different colorings of the existing tree, and there are 9 more cases for all of the various symmetries of these 3 cases.

Deleting from a Red-Black tree

Fun exercise!



That's a lot of cases!

What have we learned?

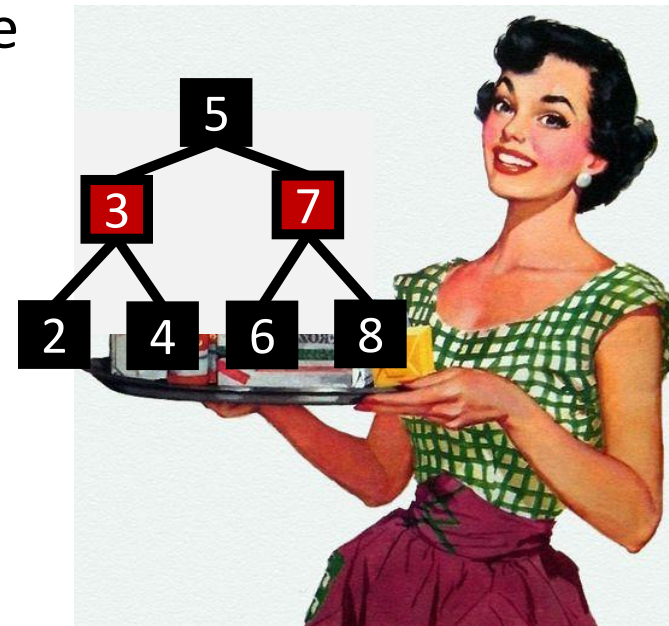
- Red-Black Trees always have height at most $2\log(n+1)$.
- As with general Binary Search Trees, all operations are $O(\text{height})$
- So all operations with RBTrees are $O(\log(n))$.

Conclusion: The best of both worlds

	Sorted Arrays	Linked Lists	Red Black Trees
Search	$O(\log(n))$ 😊	$O(n)$ 😞	$O(\log(n))$ 😊
Delete	$O(n)$ 😞	$O(n)$ 😞	$O(\log(n))$ 😊
Insert	$O(n)$ 😞	$O(1)$ 😊	$O(\log(n))$ 😊

Recap

- Balanced binary trees are the best of both worlds!
- But we need to keep them balanced.
- **Red-Black Trees** do that for us.
 - We get $O(\log(n))$ -time INSERT/DELETE/SEARCH
 - Clever idea: have a proxy for balance



Acknowledgement

- Stanford University