

# Module 4 - Trees

Modified/Inspired from Stanford's CS161 by  
Nayyar Zaidi

# Motivations for BST

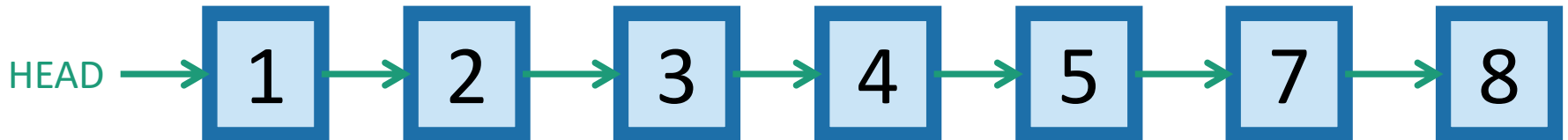
1. The punchline is **important**:
  - A data structure with  $O(\log(n))$   
INSERT/DELETE/SEARCH
2. The idea behind AVL Trees and **Red-Black Trees** is clever
  - It's good to be exposed to clever ideas.
  - Also it's just aesthetically pleasing.

Some data structures  
for storing objects like **5** (aka, **nodes** with **keys**)

- (Sorted) arrays:



- (Sorted) linked lists:



- Some basic operations:
  - **INSERT, DELETE, SEARCH**

# Sorted Arrays

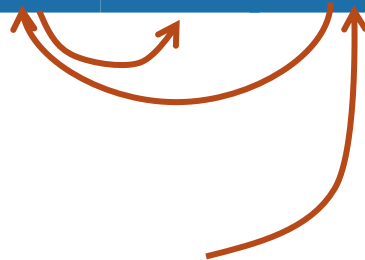
1	2	3	4	5	7	8
---	---	---	---	---	---	---

- $O(n)$  INSERT/DELETE:

1	2	3	4	4.5	7	8
---	---	---	---	-----	---	---

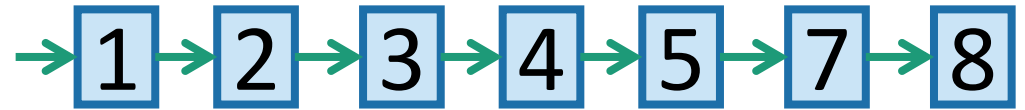
- $O(\log(n))$  SEARCH:

1	2	3	4	5	7	8
---	---	---	---	---	---	---



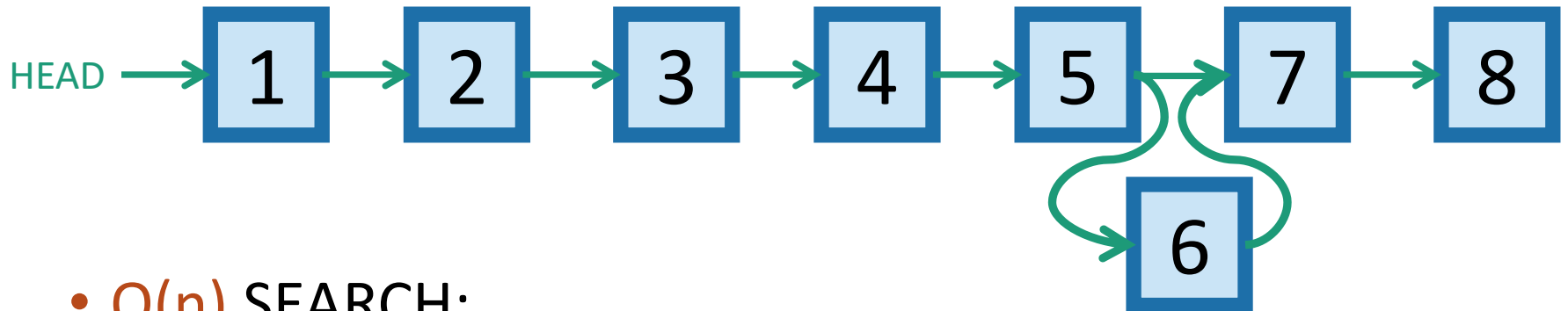
eg, Binary search to see if 3 is in A.

# Sorted linked lists

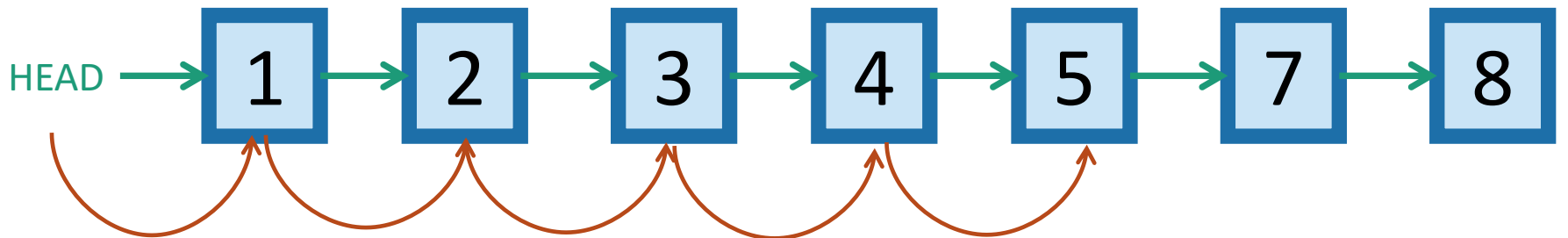


- $O(1)$  INSERT/DELETE:

- (assuming we have a pointer to the location of the insert/delete)









- $O(n)$  SEARCH:



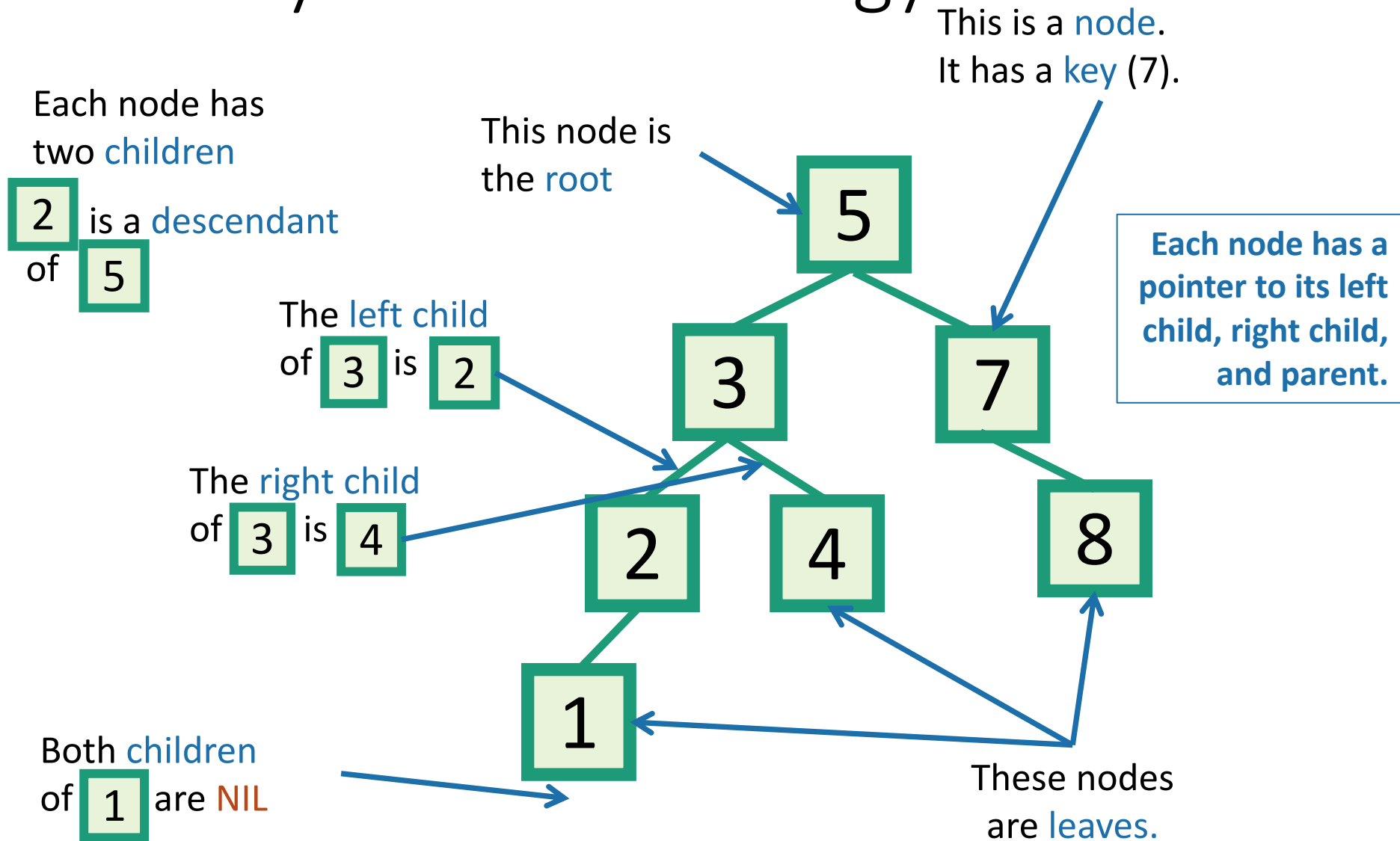
# Motivation for Binary Search Trees

**TODAY!**

	Sorted Arrays	Linked Lists	Binary Search Trees*
Search	$O(\log(n))$ 	$O(n)$ 	$O(\log(n))$ 
Insert/Delete	$O(n)$ 	$O(1)$ 	$O(\log(n))$ 

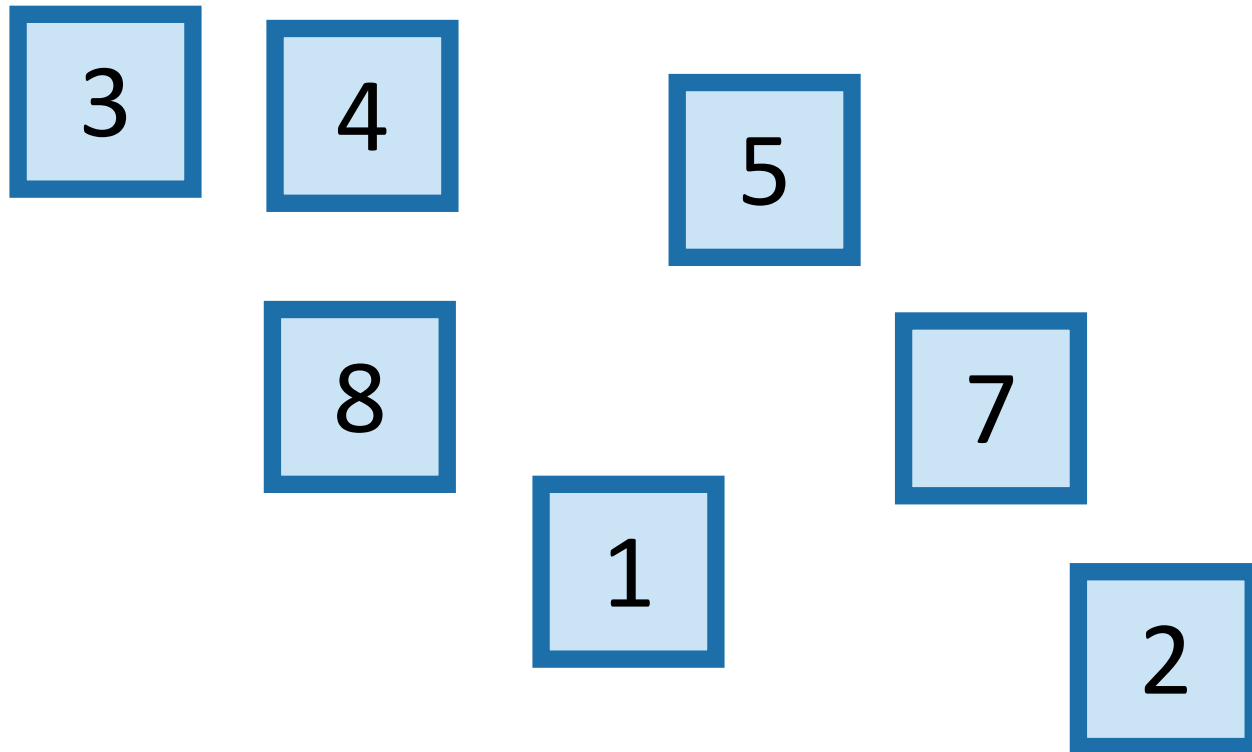
For today all keys are distinct.

# Binary tree terminology



# Binary Search Trees

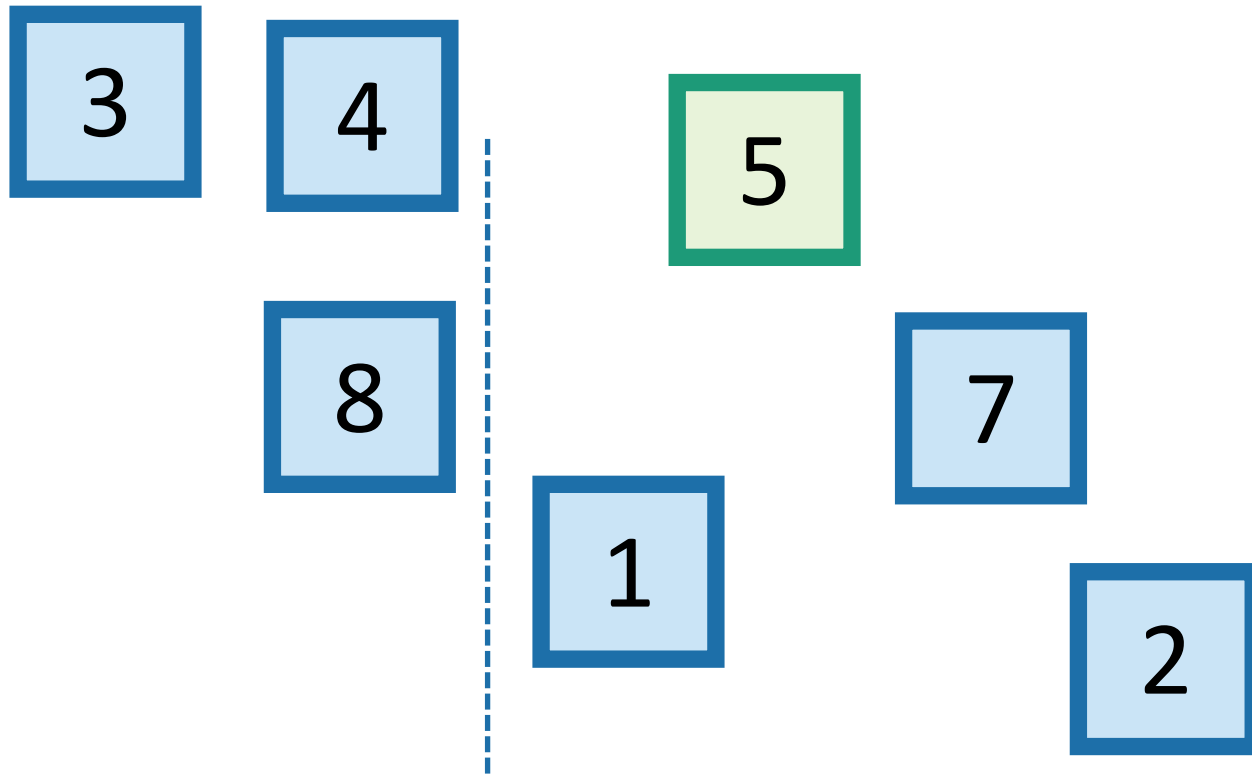
- It's a **binary tree** so that:
  - Every LEFT descendant of a node has key less than that node.
  - Every RIGHT descendant of a node has key larger than that node.
- Example of building a binary search tree:





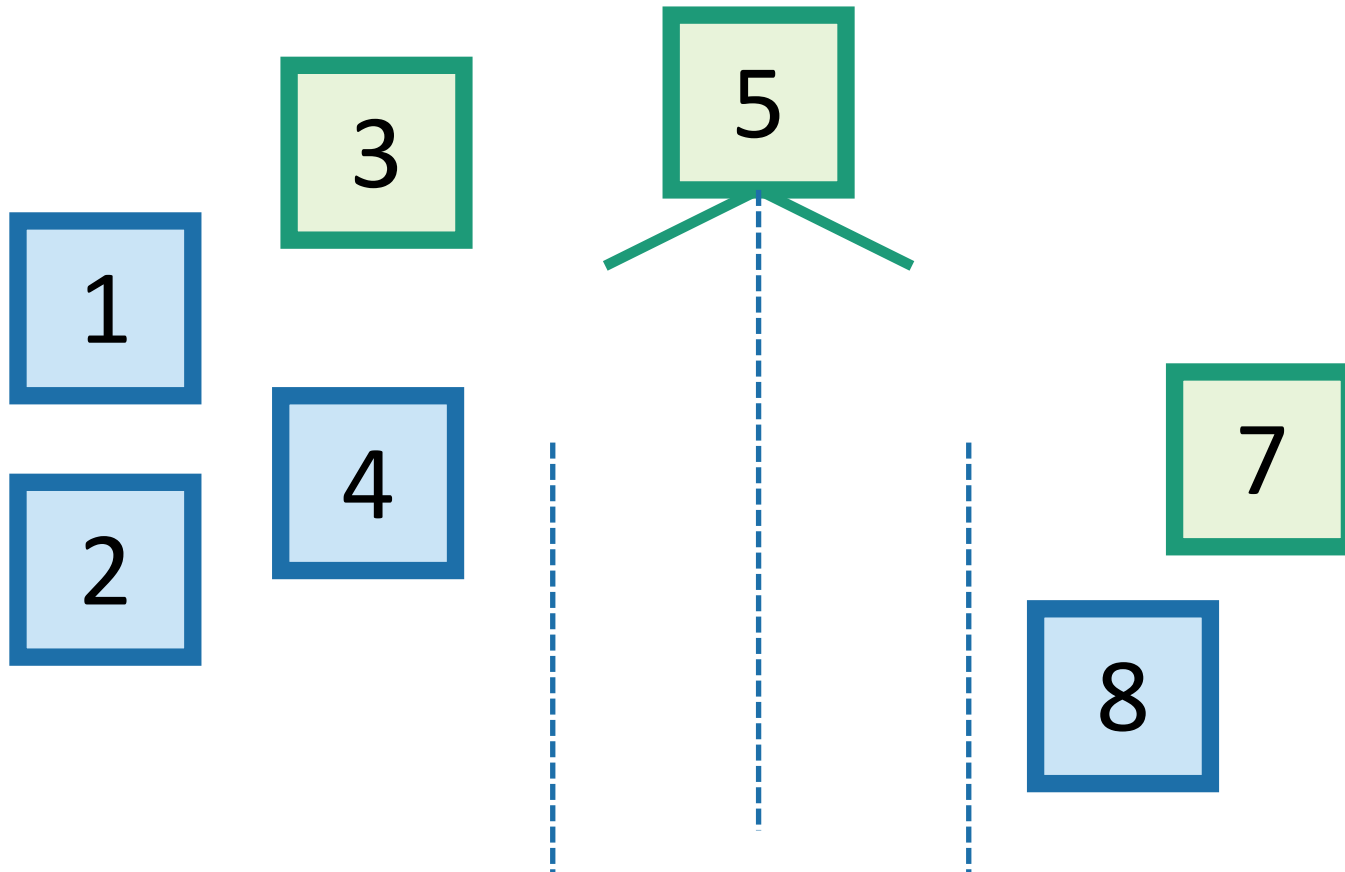
# Binary Search Trees

- It's a **binary tree** so that:
  - Every LEFT descendant of a node has key less than that node.
  - Every RIGHT descendant of a node has key larger than that node.
- Example of building a binary search tree:



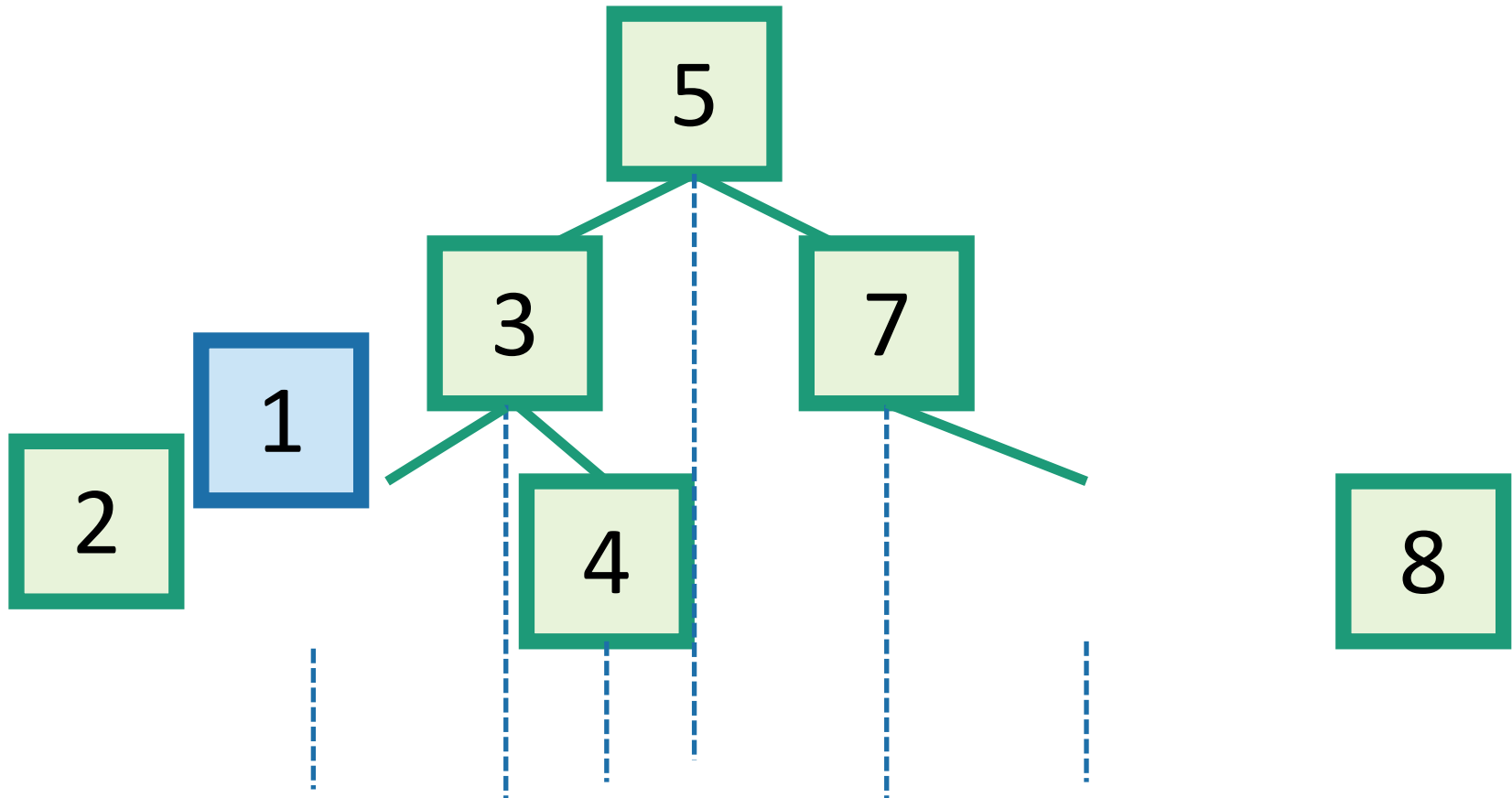
# Binary Search Trees

- It's a **binary tree** so that:
  - Every LEFT descendant of a node has key less than that node.
  - Every RIGHT descendant of a node has key larger than that node.
- Example of building a binary search tree:



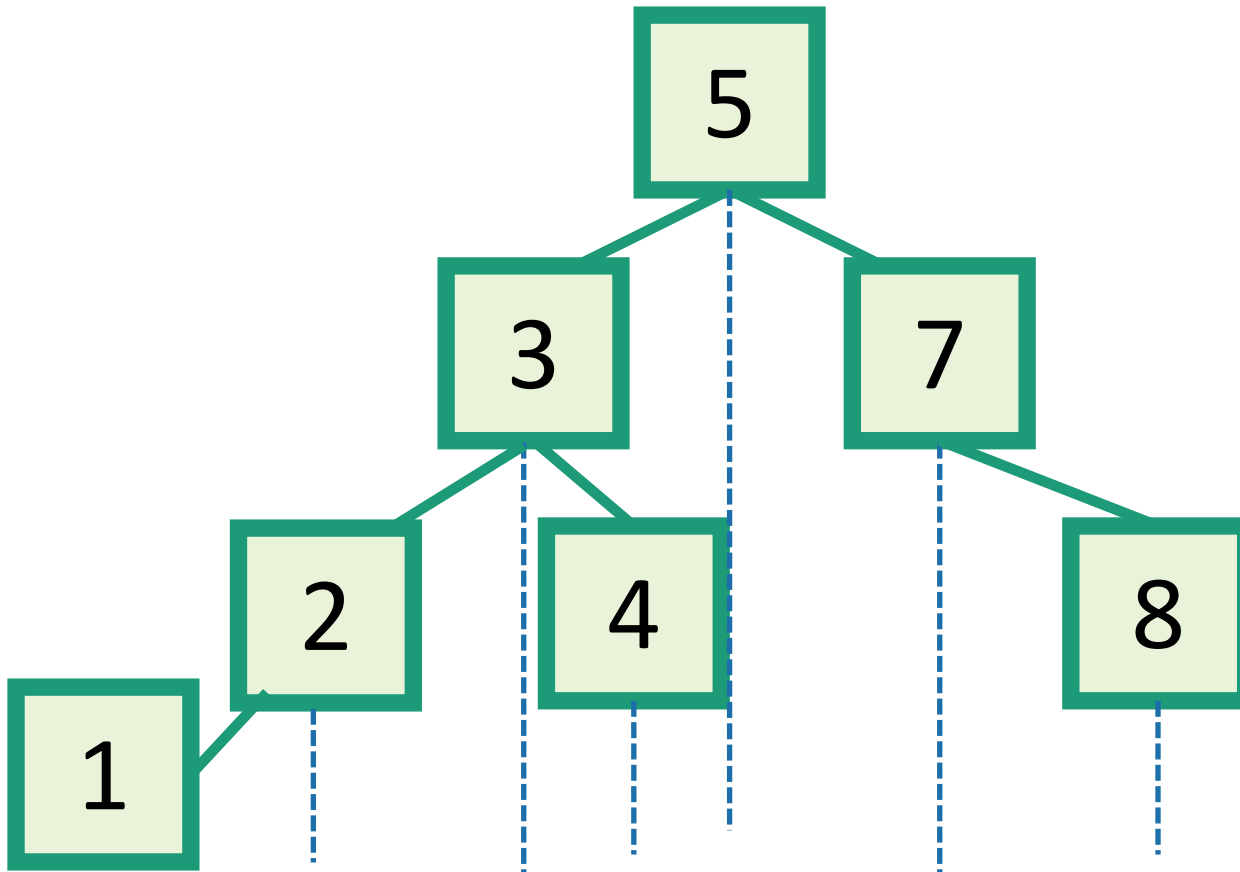
# Binary Search Trees

- It's a **binary tree** so that:
  - Every LEFT descendant of a node has key less than that node.
  - Every RIGHT descendant of a node has key larger than that node.
- Example of building a binary search tree:



# Binary Search Trees

- It's a **binary tree** so that:
  - Every LEFT descendant of a node has key less than that node.
  - Every RIGHT descendant of a node has key larger than that node.
- Example of building a binary search tree:



Q: Is this the only binary search tree I could possibly build with these values?

A: **No.** I made choices about which nodes to choose when. Any choices would have been fine.

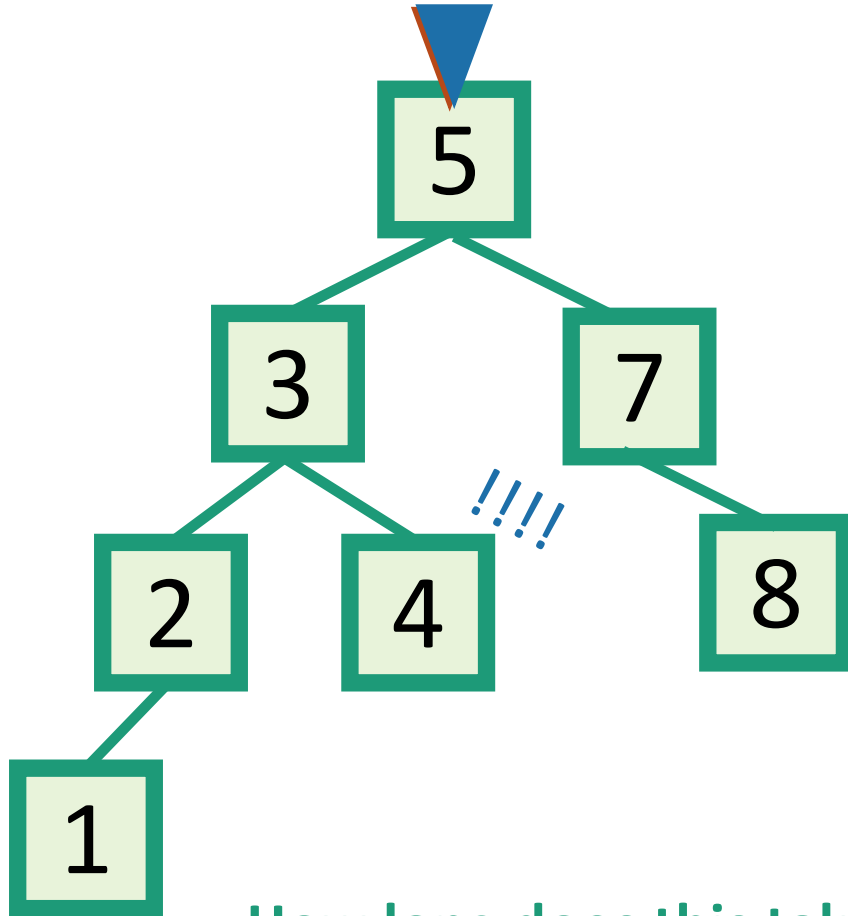
# Remember the goal

Fast **SEARCH/INSERT/DELETE**

Can we do these?

# SEARCH in a Binary Search Tree

definition by example



**EXAMPLE:** Search for 4.

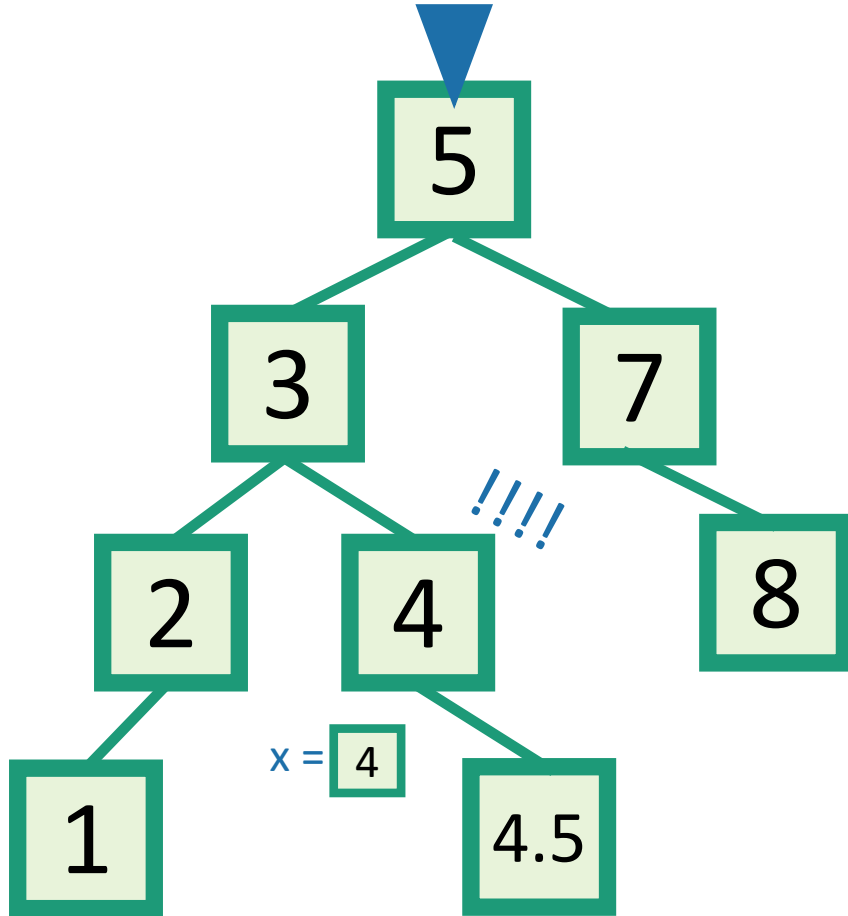
**EXAMPLE:** Search for 4.5

- It turns out it will be convenient to **return 4** in this case
- (that is, **return** the last node before we went off the tree)

How long does this take?

$O(\text{length of longest path}) = O(\text{height})$

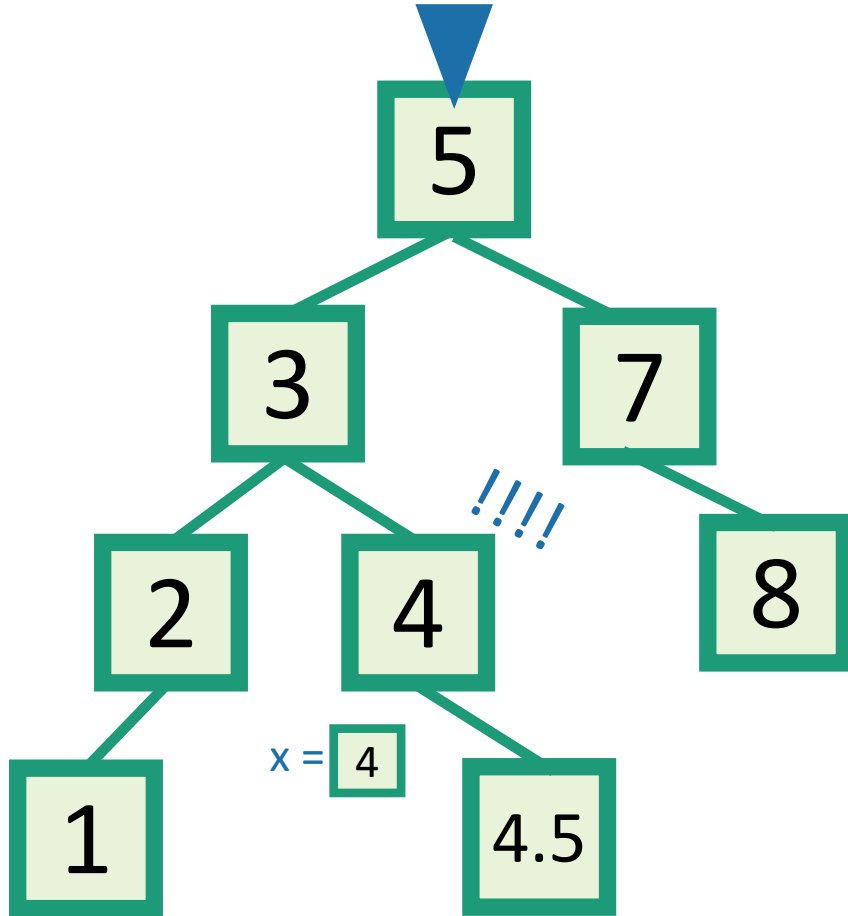
# INSERT in a Binary Search Tree



## EXAMPLE: Insert 4.5

- **INSERT**(key):
  - $x = \text{SEARCH}(\text{key})$
  - **Insert** a new node with desired key at  $x$ ...

# INSERT in a Binary Search Tree

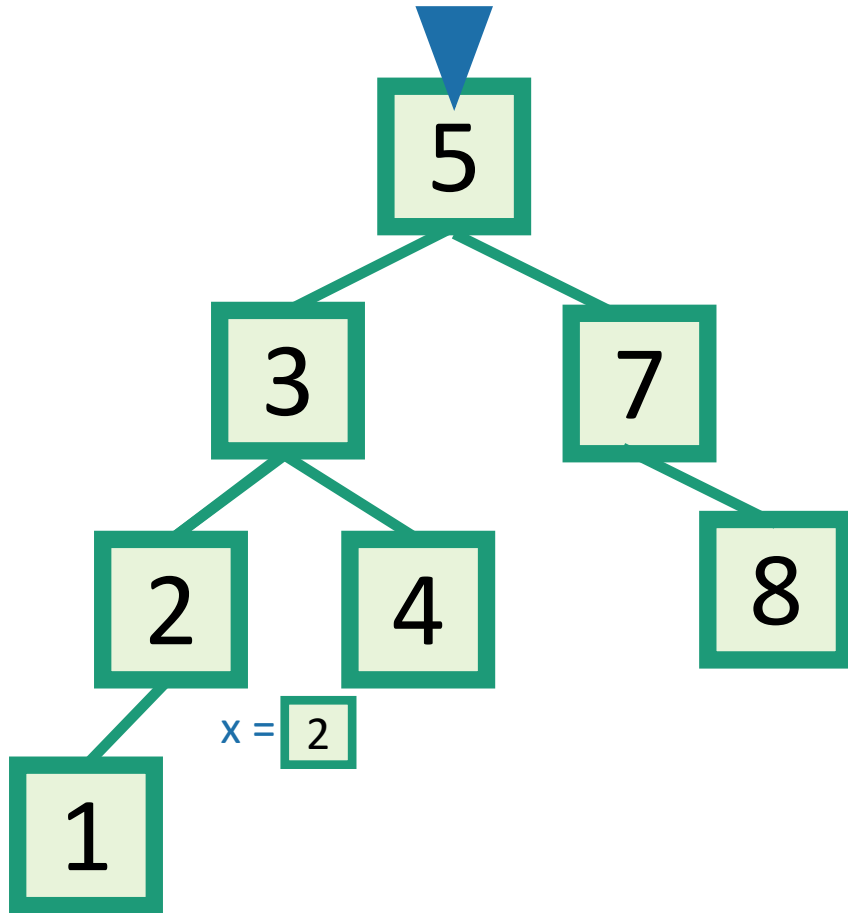


## EXAMPLE: Insert 4.5

- **INSERT**(key):
  - $x = \text{SEARCH}(\text{key})$
  - **if**  $\text{key} > x.\text{key}$ :
    - Make a new node with the correct key, and put it as the right child of  $x$ .
  - **if**  $\text{key} < x.\text{key}$ :
    - Make a new node with the correct key, and put it as the left child of  $x$ .
  - **if**  $x.\text{key} == \text{key}$ :
    - **return**



# DELETE in a Binary Search Tree



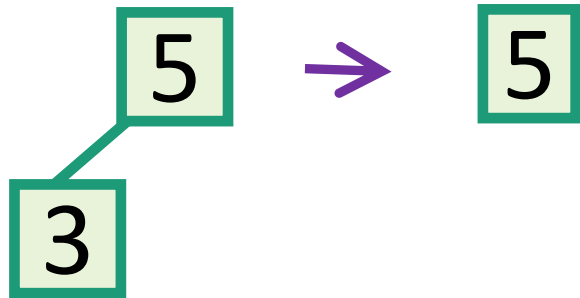
## EXAMPLE: Delete 2

- **DELETE**(key):
  - $x = \text{SEARCH}(\text{key})$
  - **if**  $x.\text{key} == \text{key}$ :
    - ....delete  $x$ ....

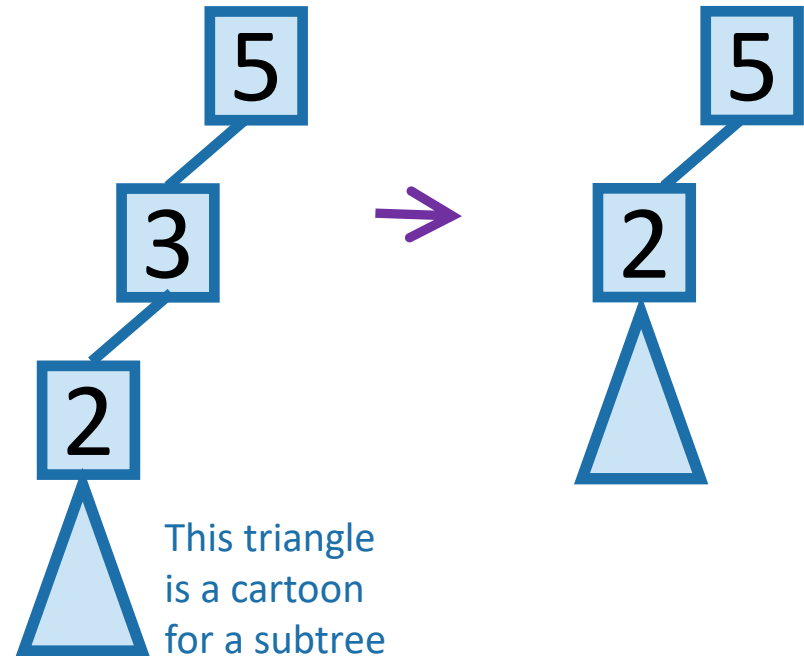
# DELETE in a Binary Search Tree

several cases (by example)

say we want to delete 3



**Case 1:** if 3 is a leaf,  
just delete it.

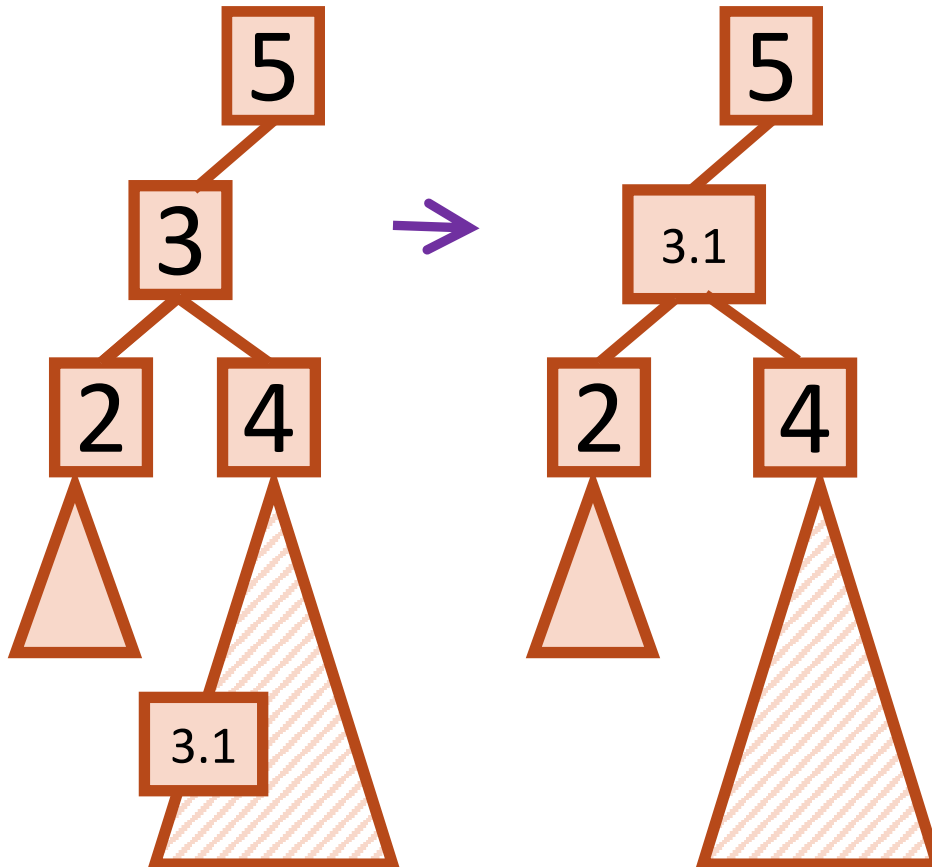


**Case 2:** if 3 has just one child,  
move that up.

# DELETE in a Binary Search Tree

ctd.

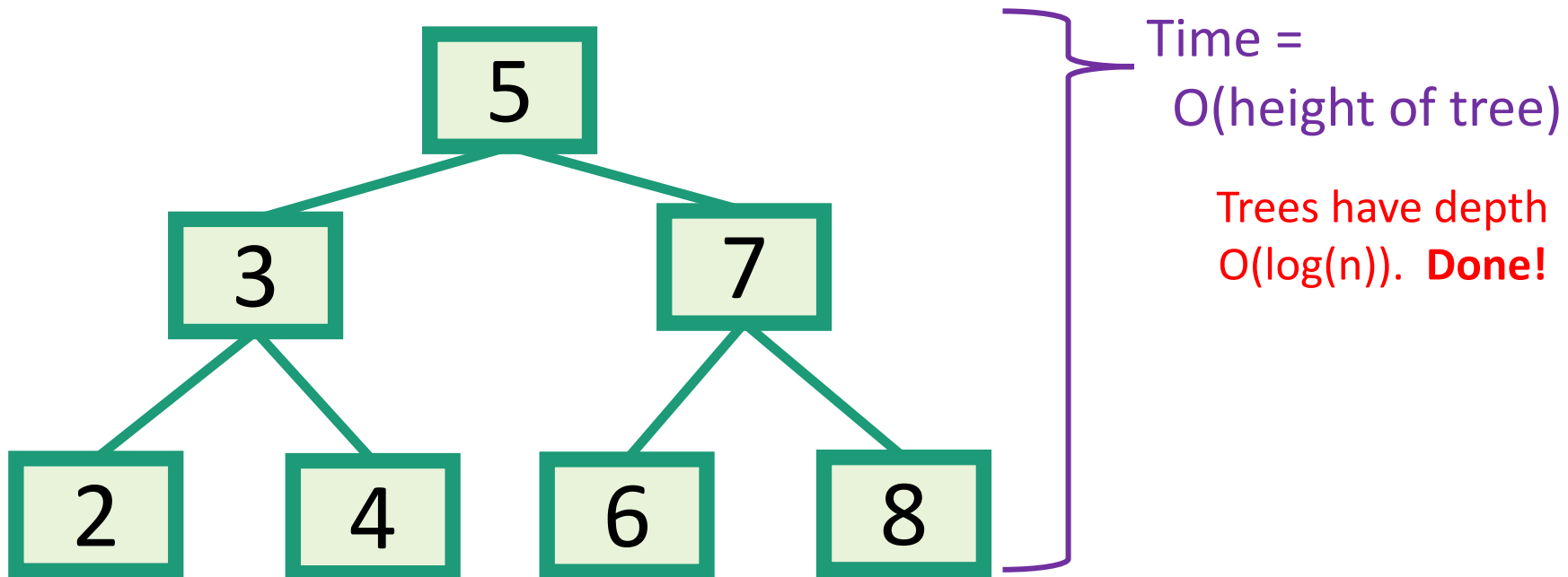
**Case 3:** if 3 has two children,  
replace 3 with its **immediate successor**.  
(aka, next biggest thing after 3)



- Does this maintain the BST property?
  - Yes.
- How do we find the immediate successor?
  - SEARCH for 3 in the subtree under 3.right
- How do we remove it when we find it?
  - If [3.1] has 0 or 1 children, do one of the previous cases.
- What if [3.1] has two children?
  - It doesn't.

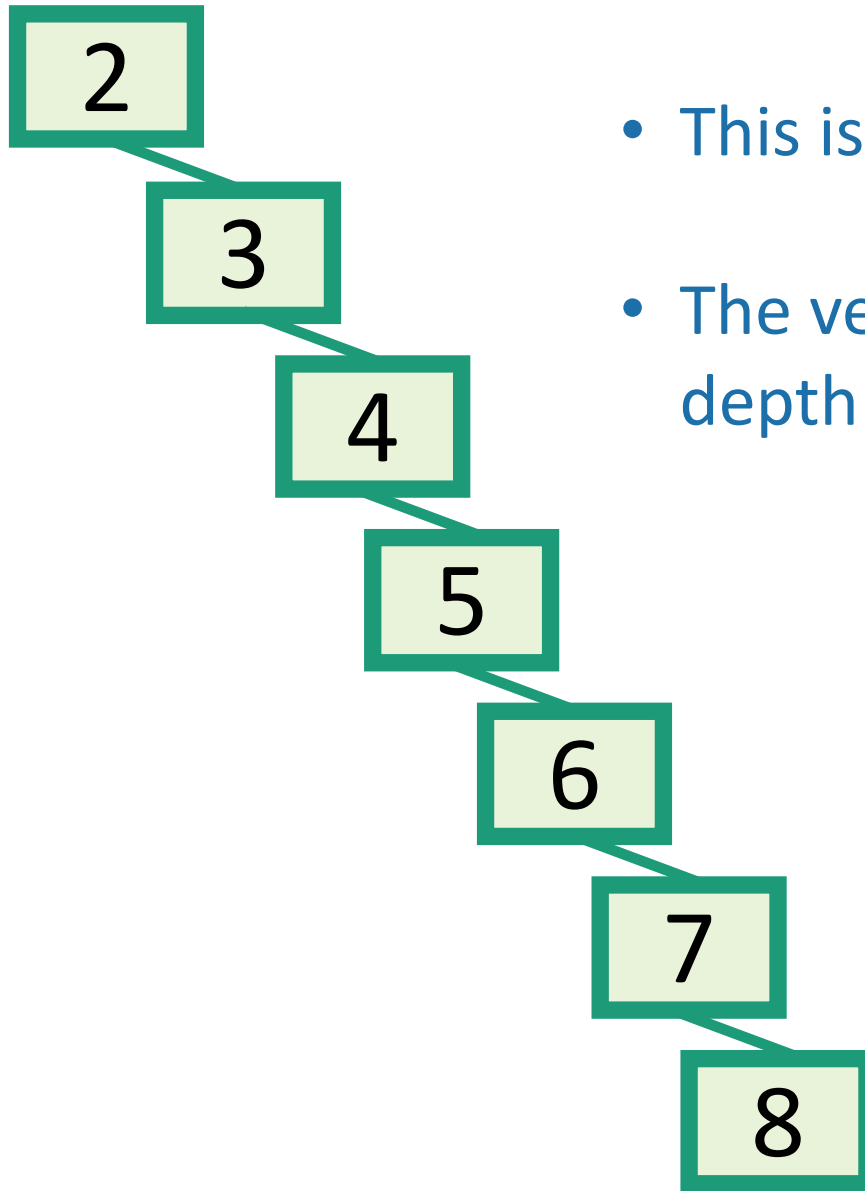
# How long do these operations take?

- **SEARCH** is the big one.
  - Everything else just calls **SEARCH** and then does some small  $O(1)$ -time operation.



How long does search take?

# Wait...



- This is a valid binary search tree.
- The version with  $n$  nodes has depth  $n$ , **not**  $O(\log(n))$ .

Could such a tree show up?  
In what order would I have to  
insert the nodes?

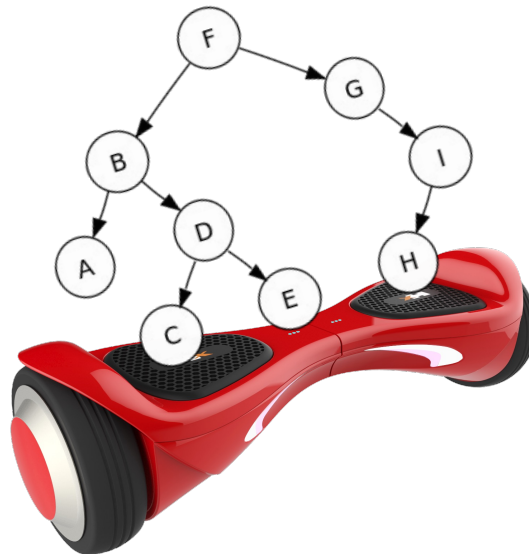
Inserting in the order  
2,3,4,5,6,7,8 would do it.

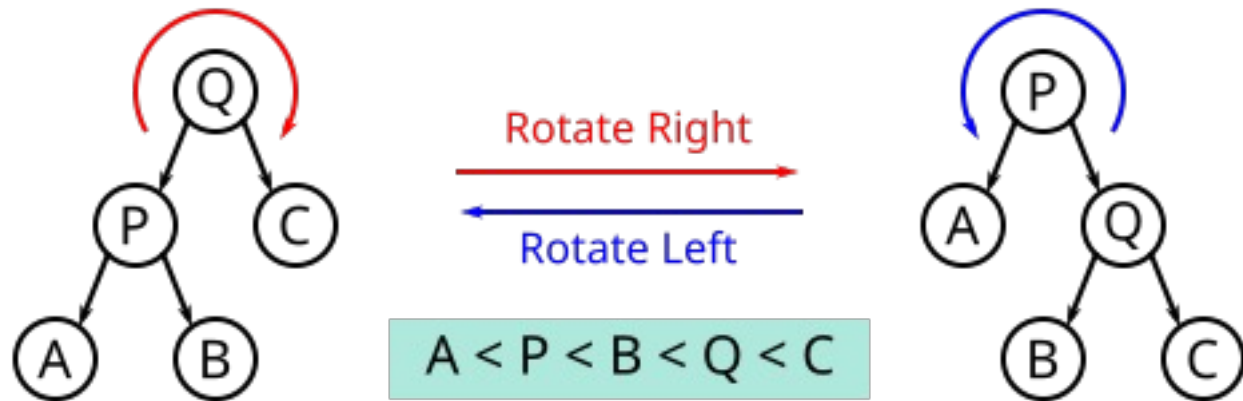
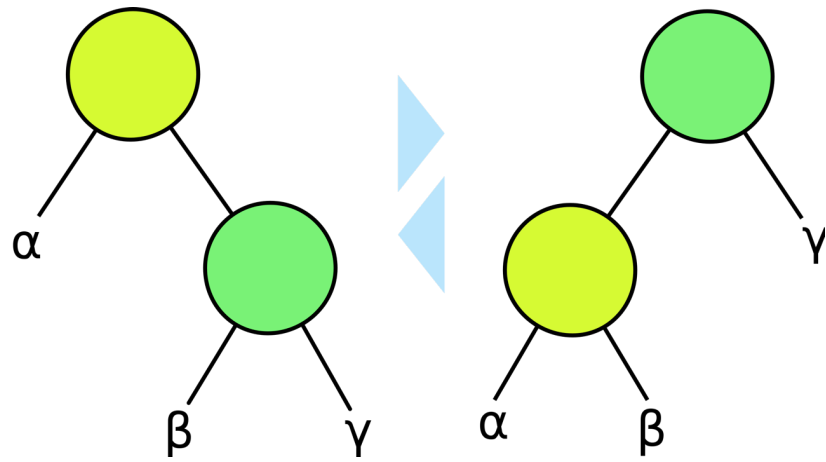
So this **could** happen.

# What to do?

- Goal: Fast SEARCH/INSERT/DELETE
- All these things take time  $O(\text{height})$
- And the height might be big!!! ☹️
- Idea 0:
  - Keep track of how deep the tree is getting.
  - If it gets too tall, re-do everything from scratch.
- Turns out that's not a great idea. Instead we turn to...

# Self-Balancing Binary Search Trees



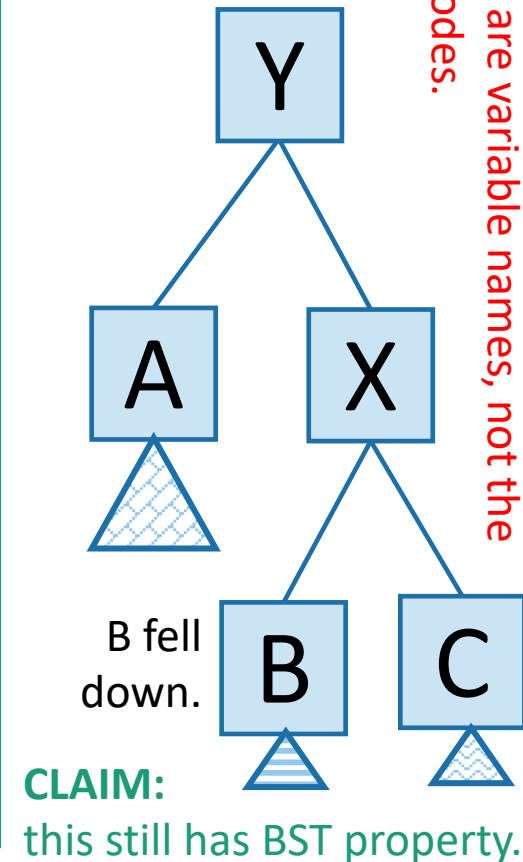
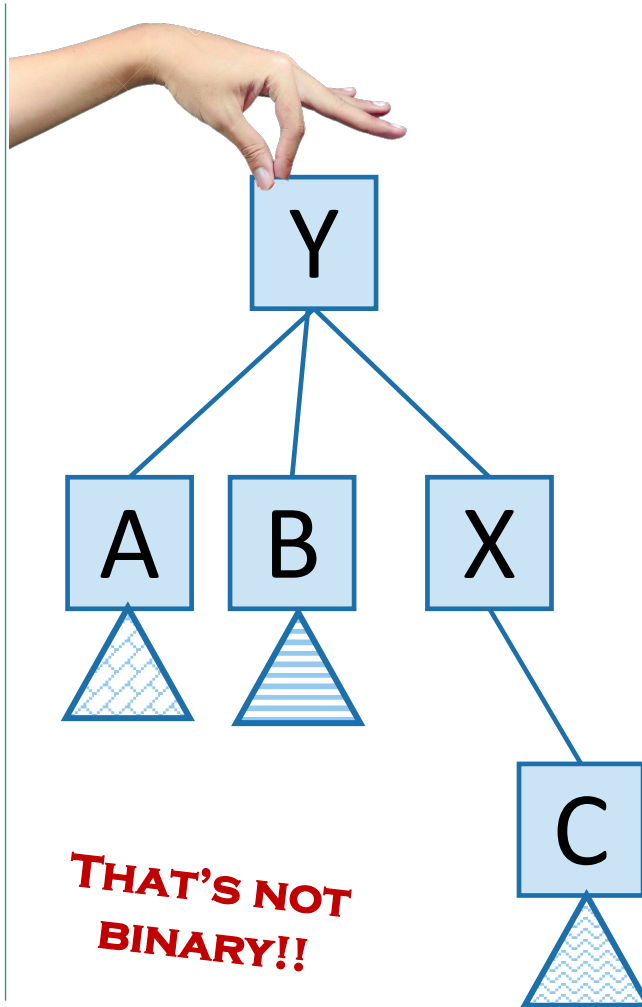
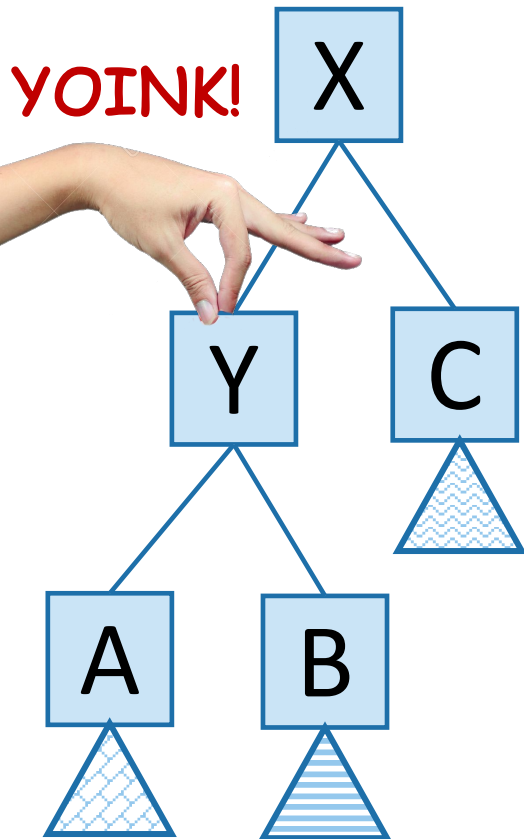




# Idea 1: Rotations

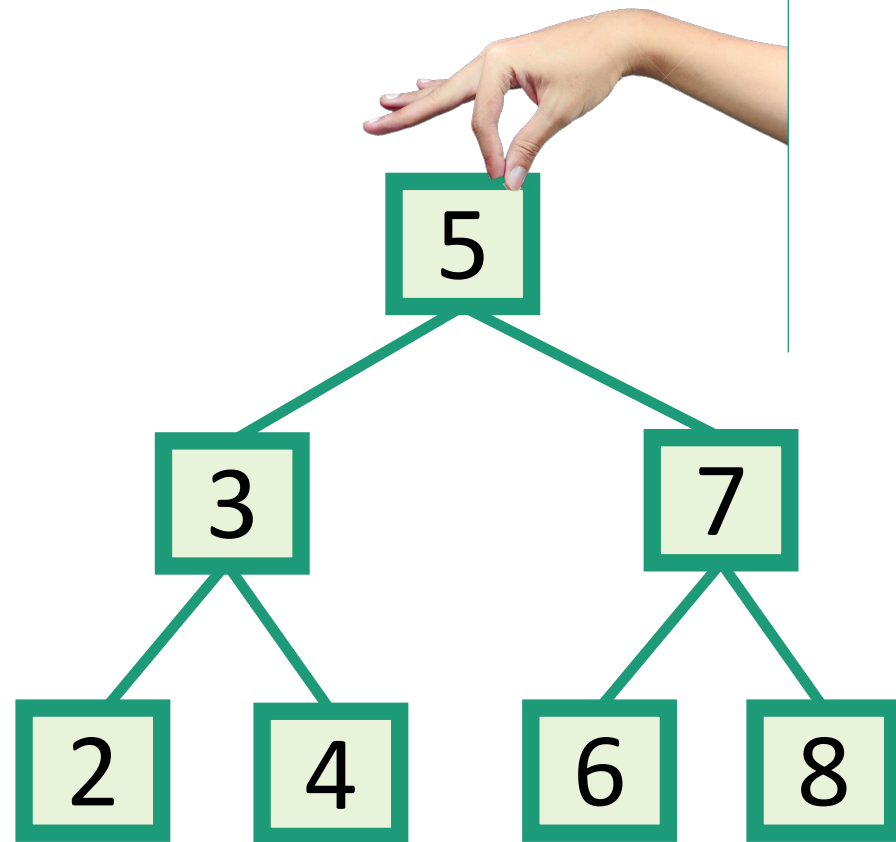
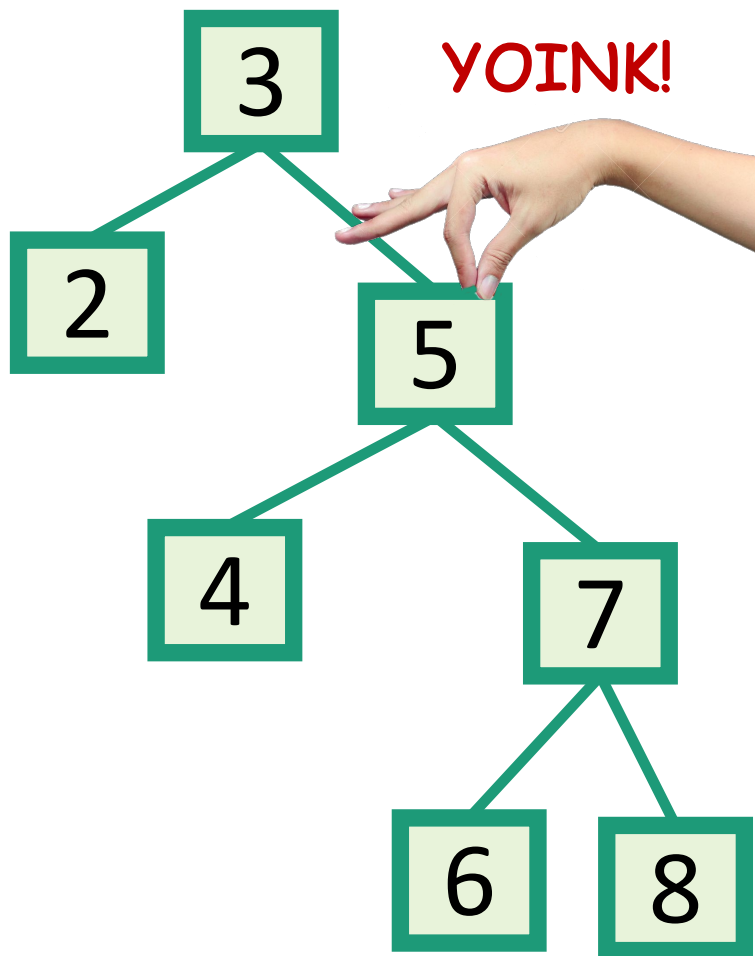
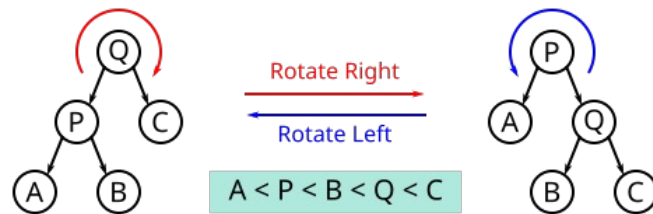
No matter what lives underneath A,B,C,  
this takes time  $O(1)$ . (Why?)

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



Note: A, B, C, X, Y are variable names, not the contents of the nodes.

This seems helpful

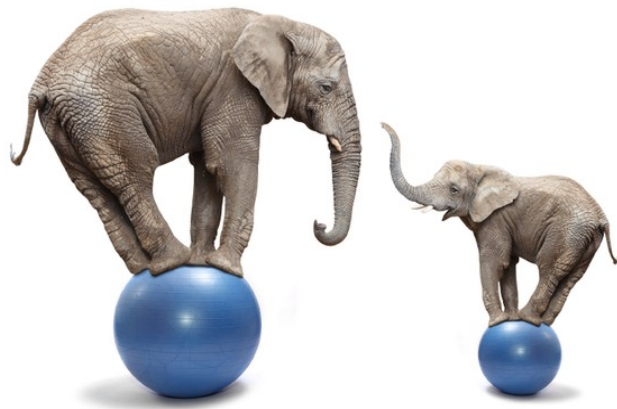


# Does this work?

- Whenever something seems unbalanced, do rotations until it's okay again.

# Idea 2: have some proxy for balance

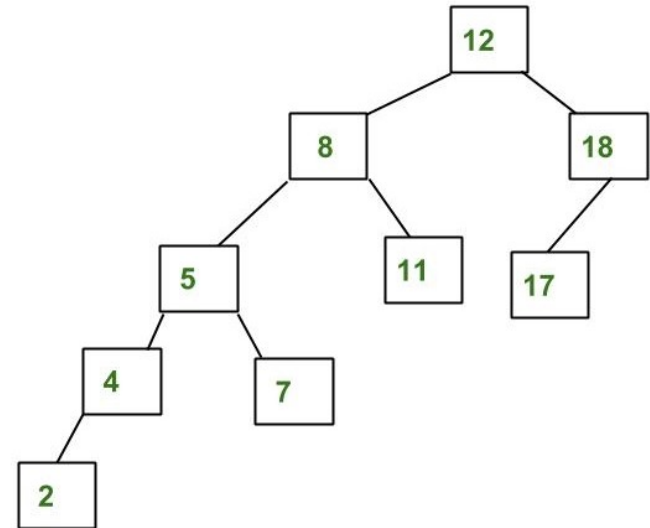
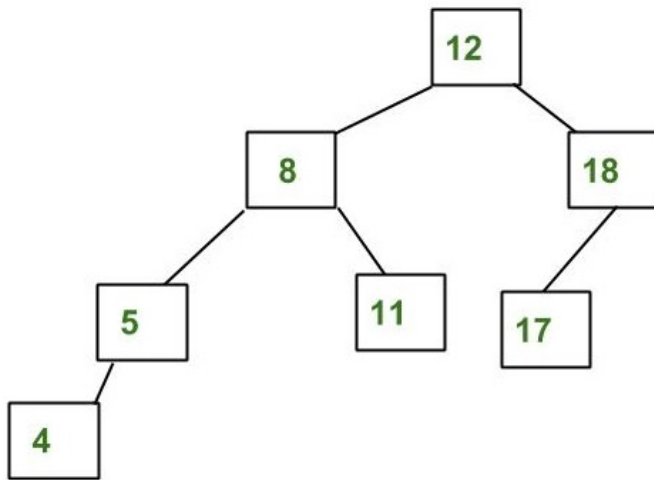
- Maintaining **perfect balance** is too hard.
- Instead, come up with some **proxy for balance**:
  - If the tree satisfies **[SOME PROPERTY]**, then it's pretty balanced.
  - We can maintain **[SOME PROPERTY]** using rotations.

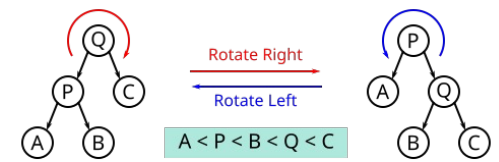
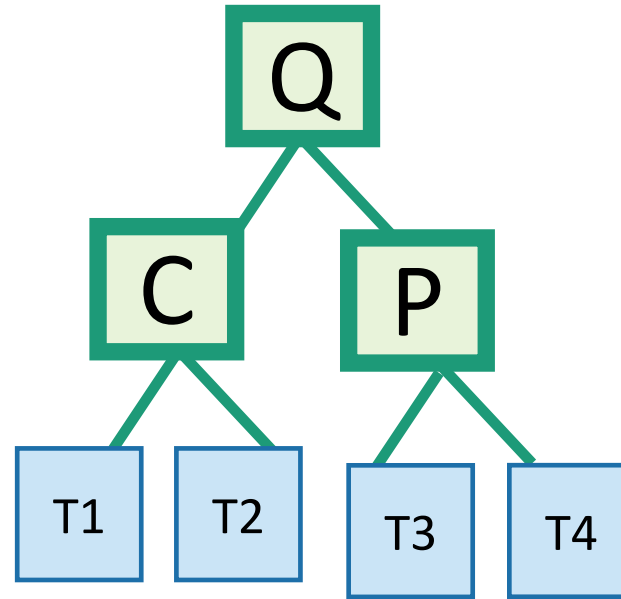
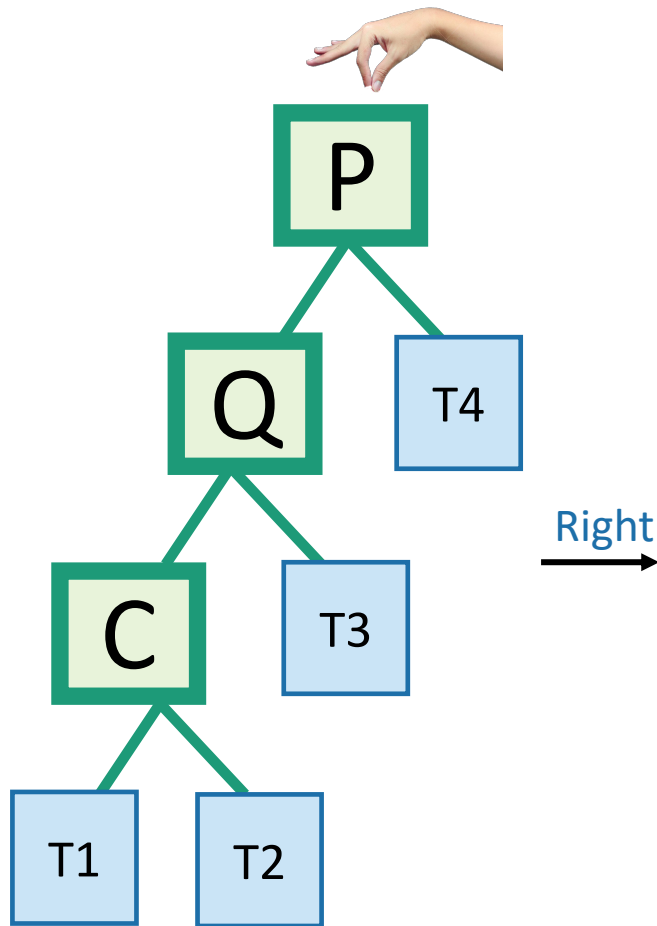


There are actually several ways to do this, but today we'll see...

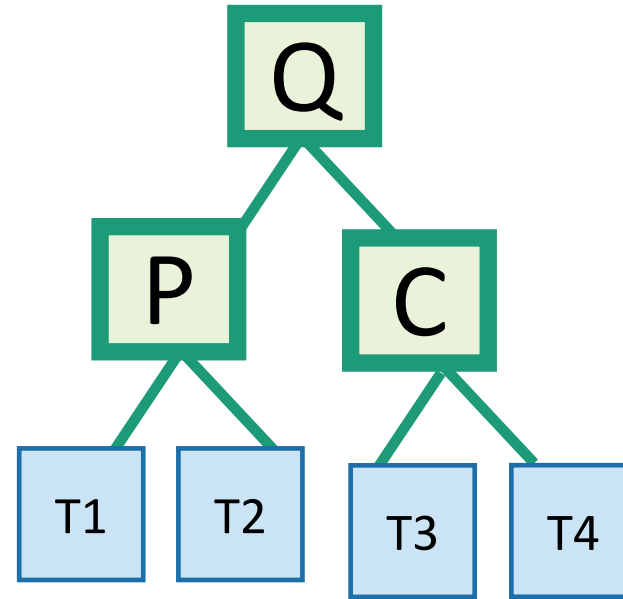
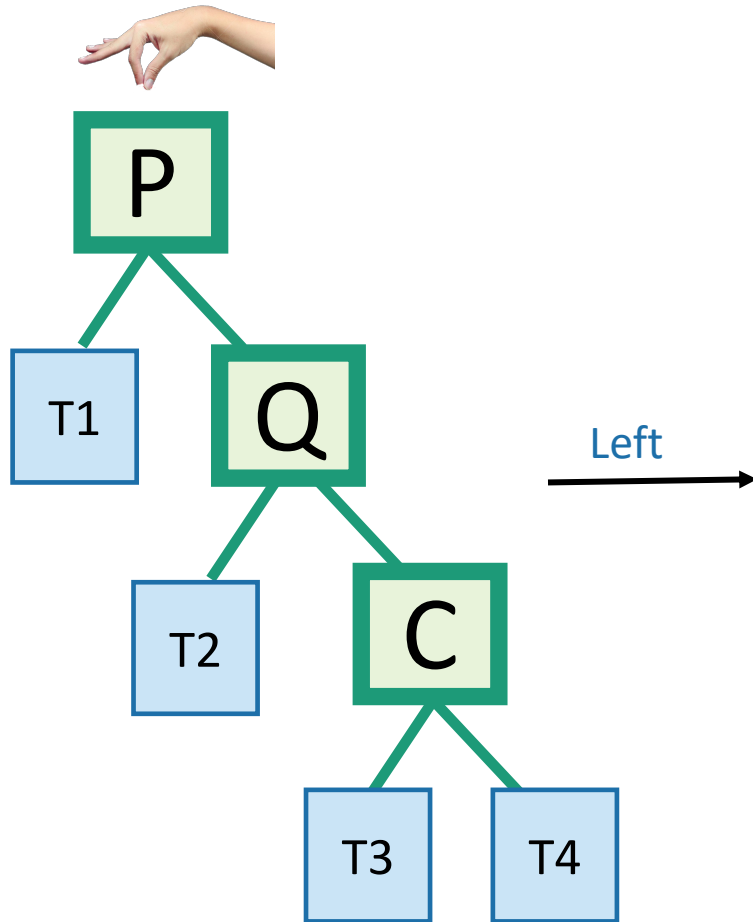
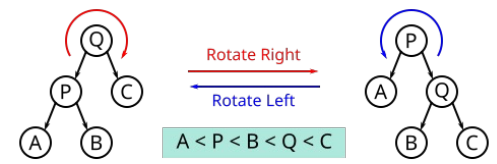
# AVL Trees

- Height-Balancing Property



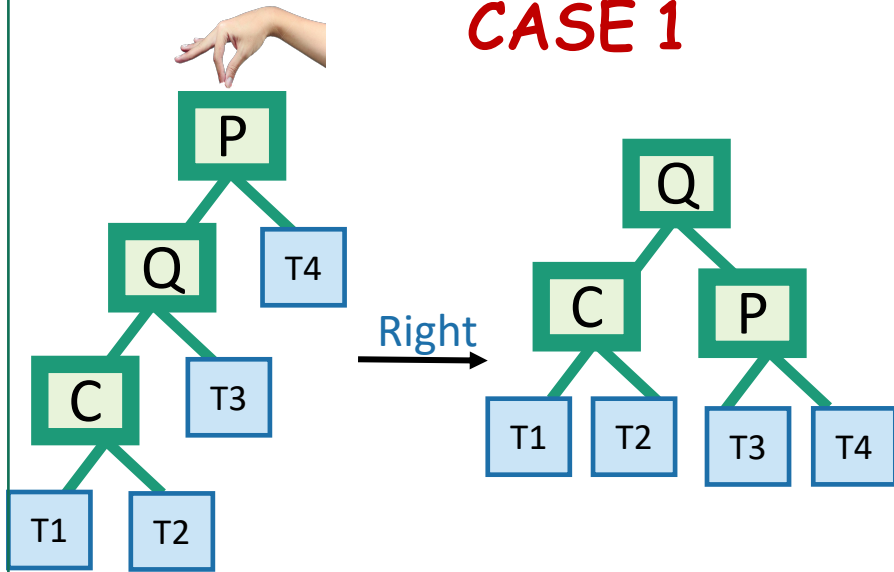


**CASE 1**

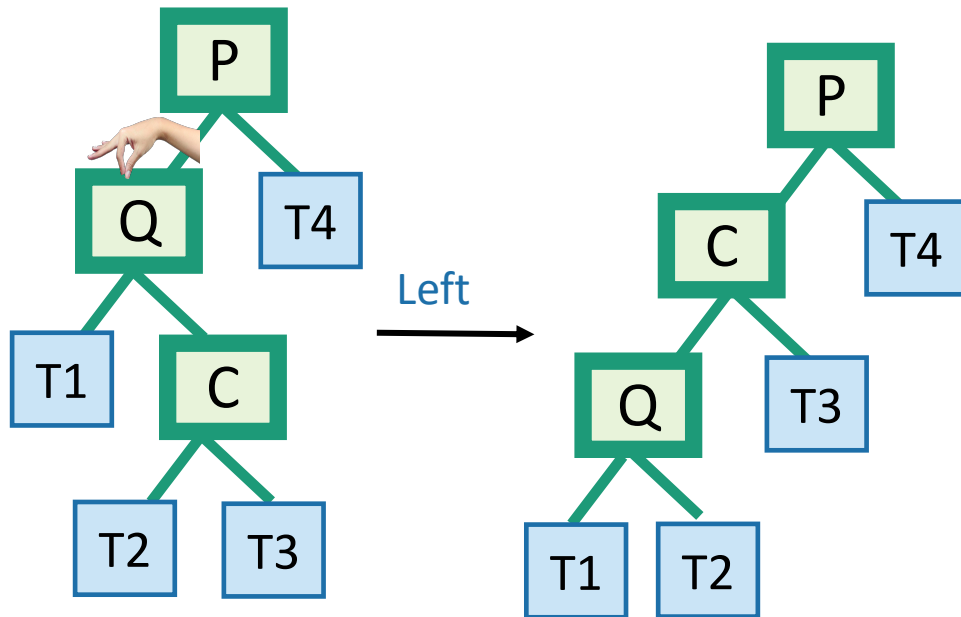
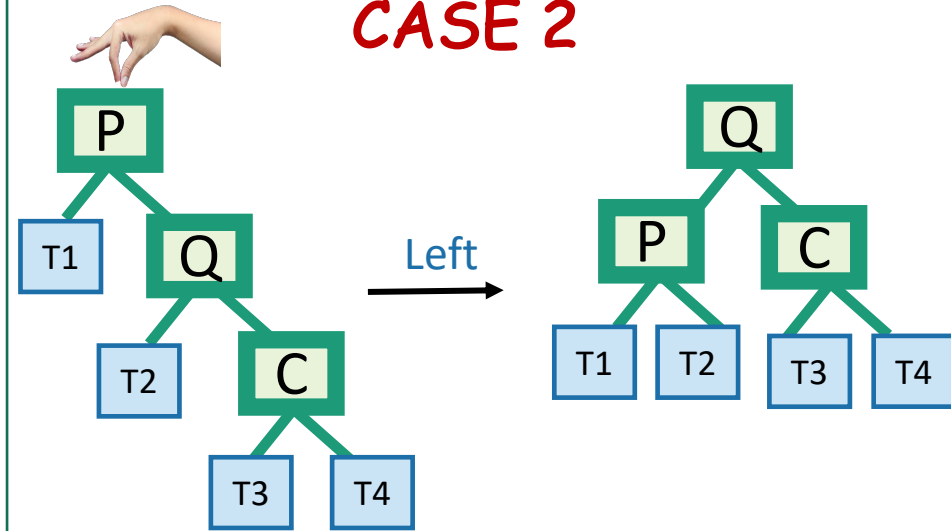


**CASE 2**

## CASE 1



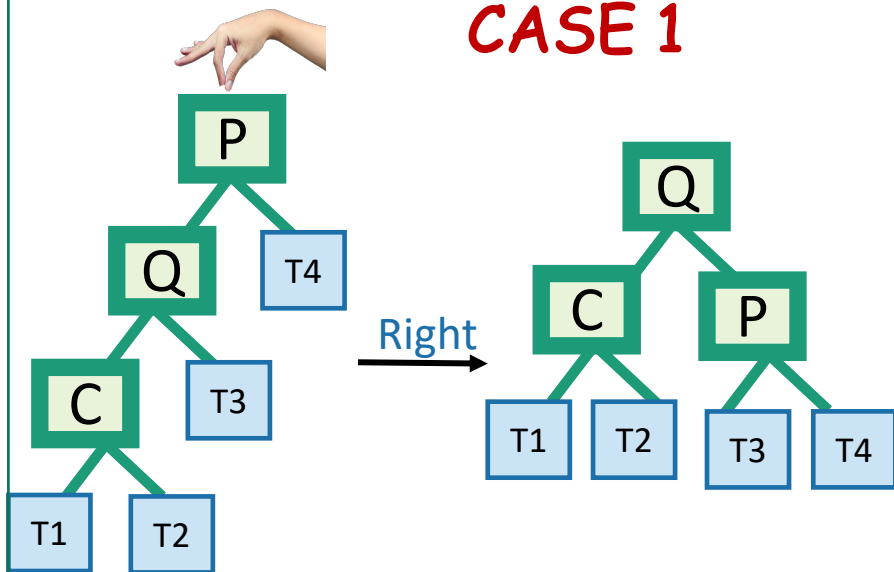
## CASE 2



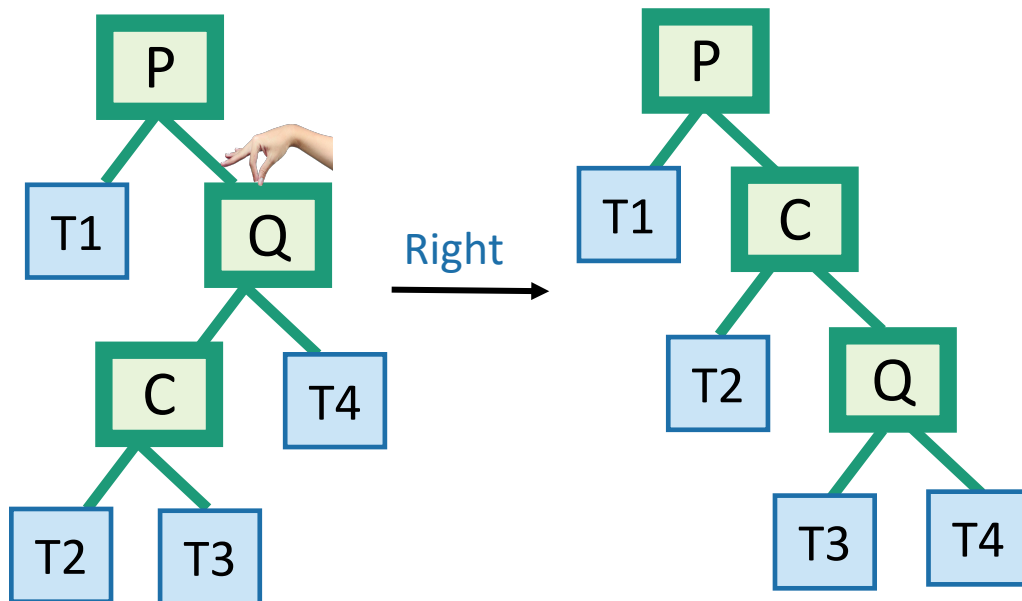
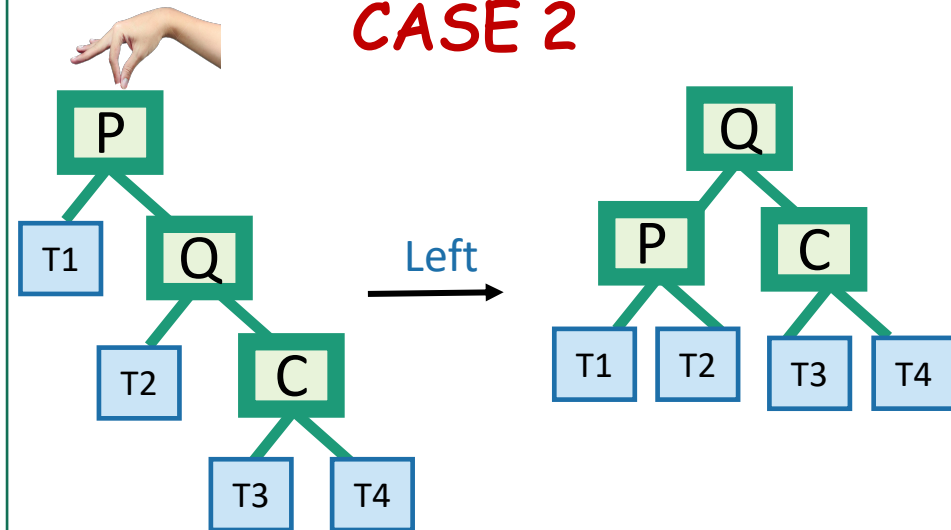
## CASE 3



## CASE 1

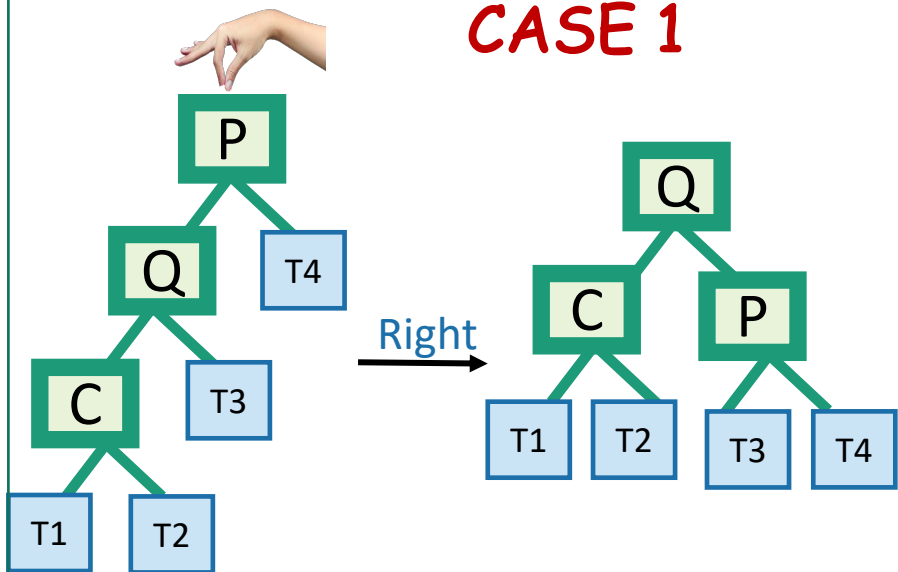


## CASE 2

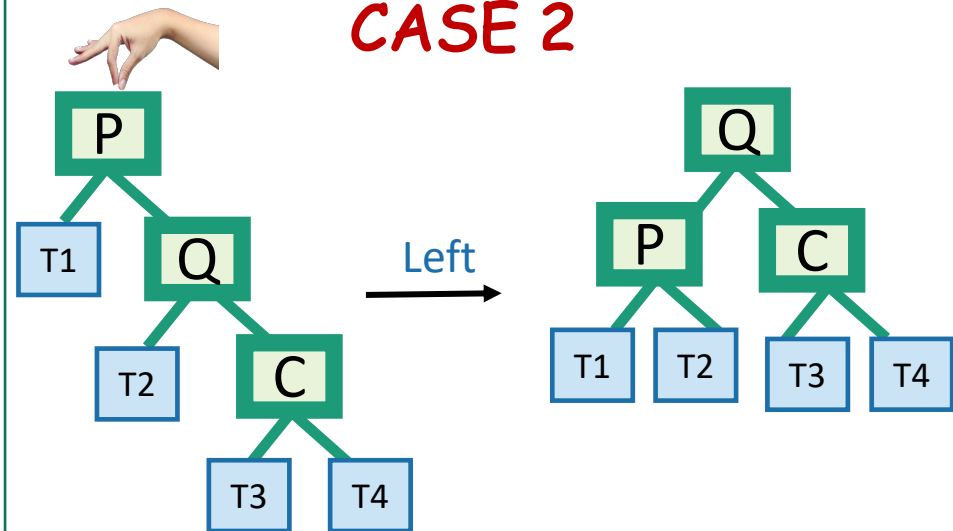


## CASE 4

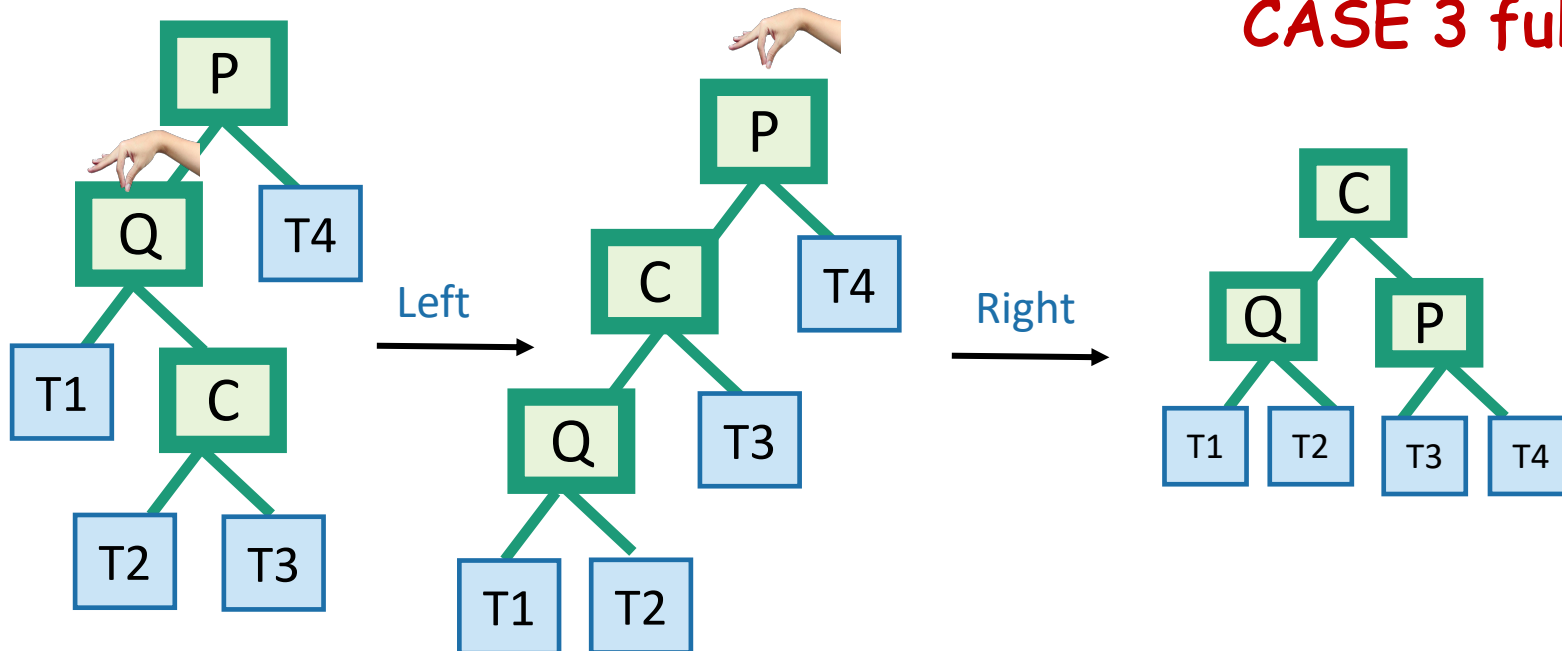
## CASE 1



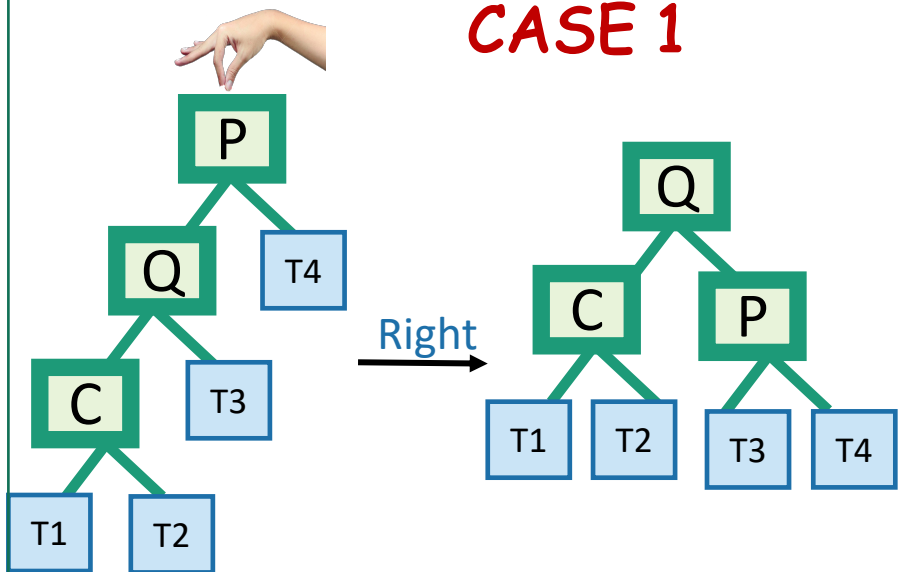
## CASE 2



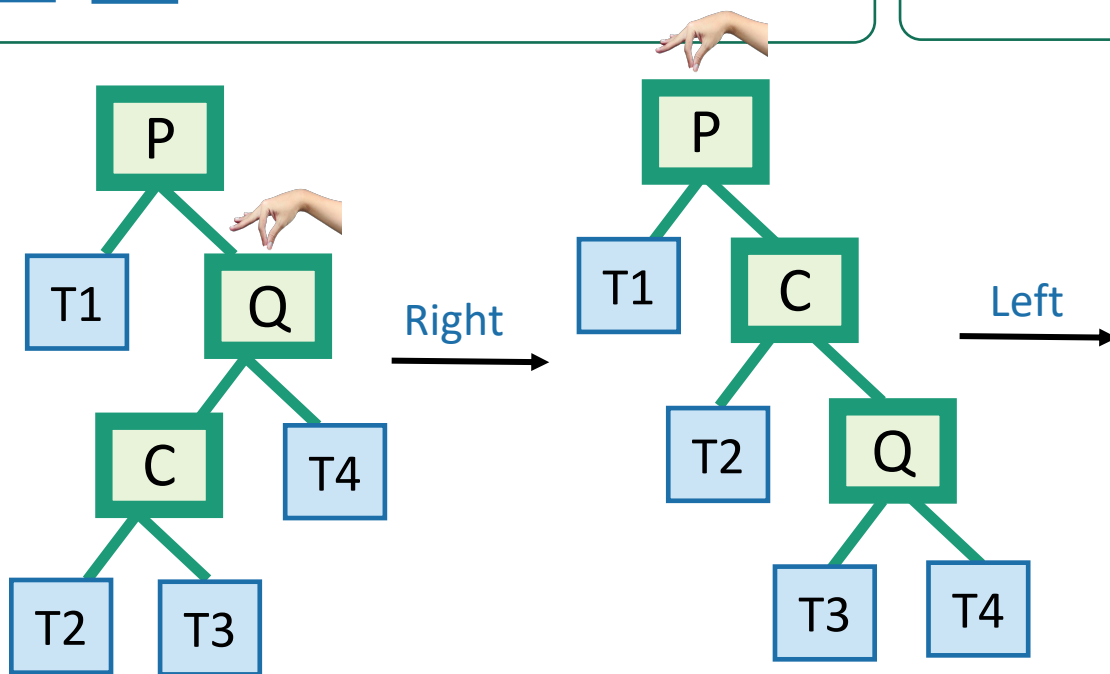
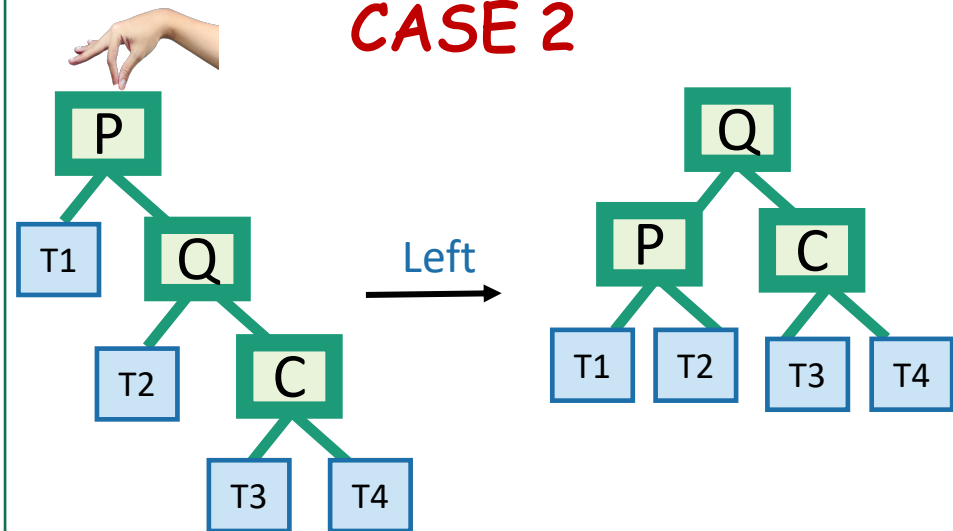
## CASE 3 full



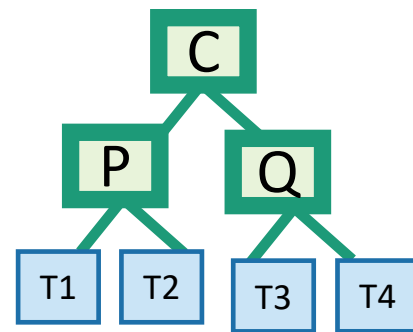
## CASE 1



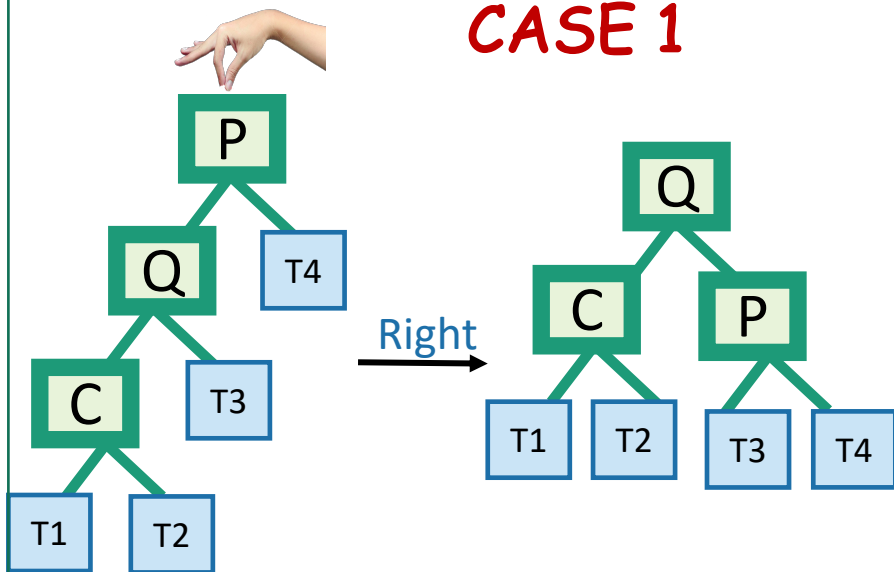
## CASE 2



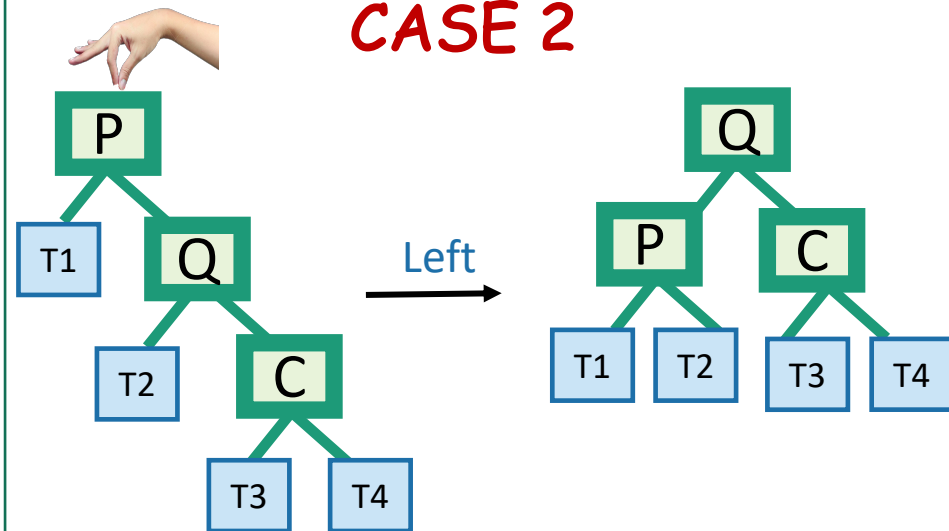
## CASE 4 full



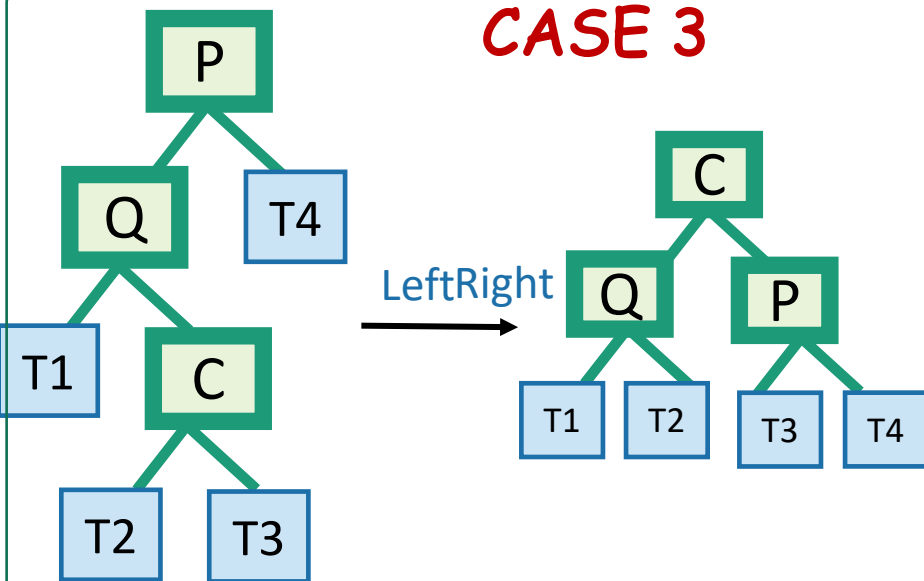
## CASE 1



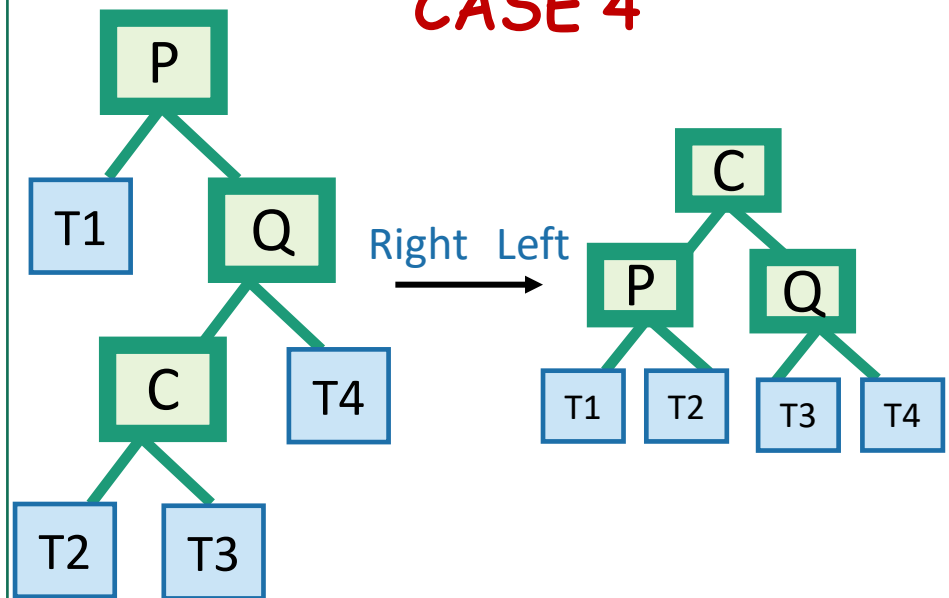
## CASE 2



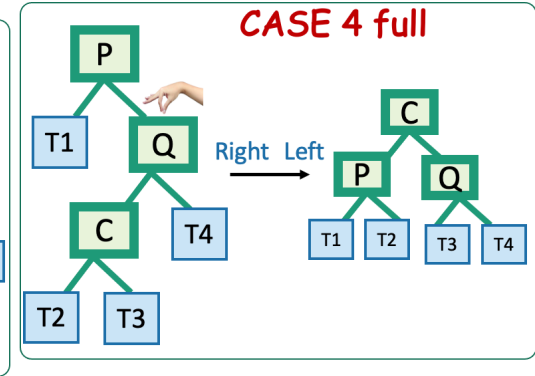
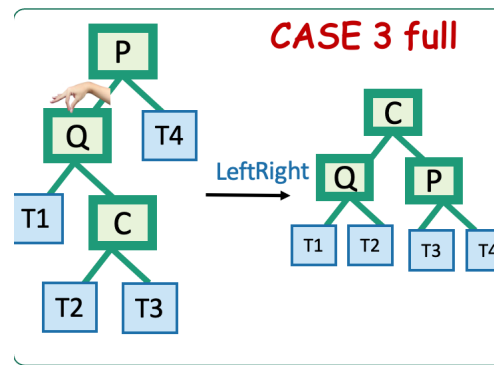
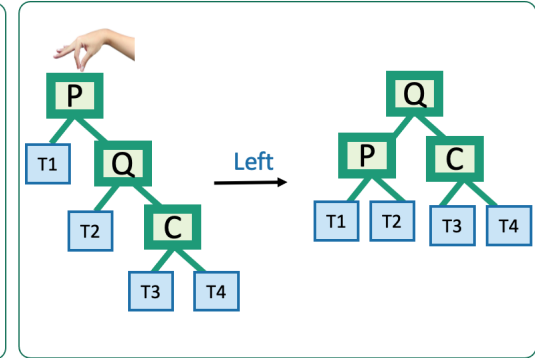
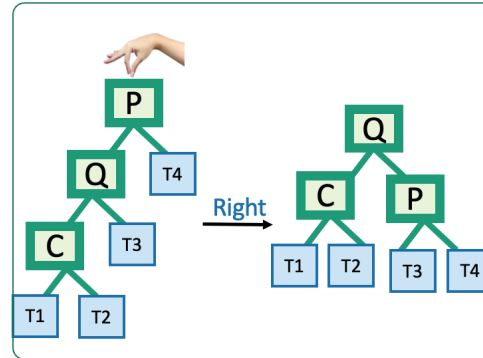
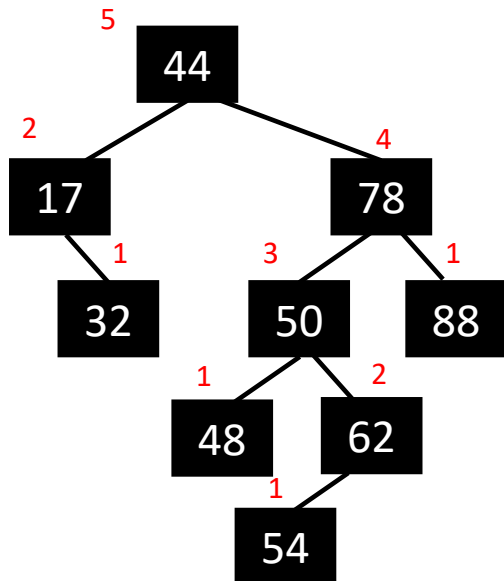
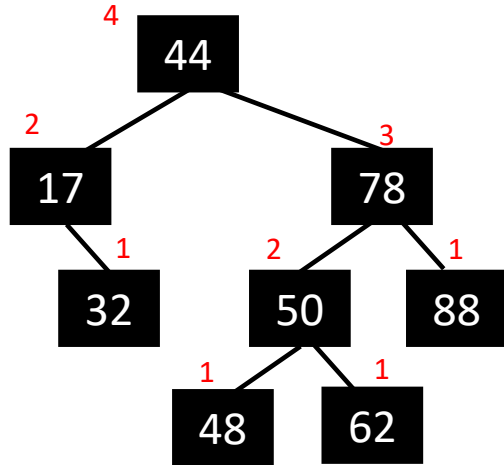
## CASE 3



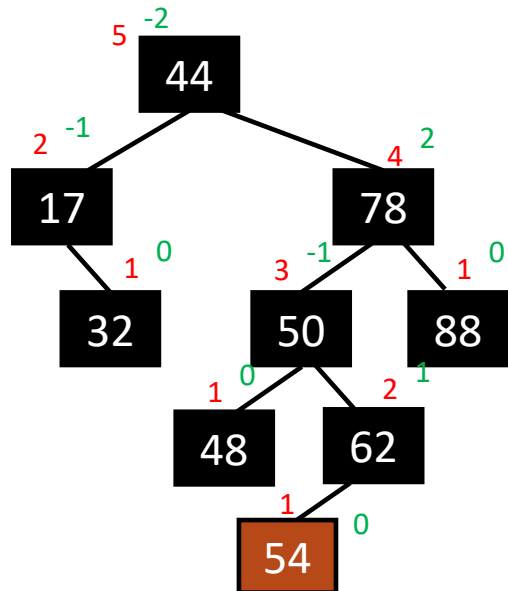
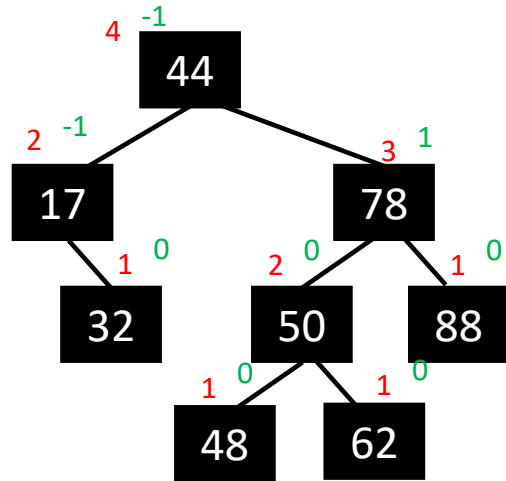
## CASE 4



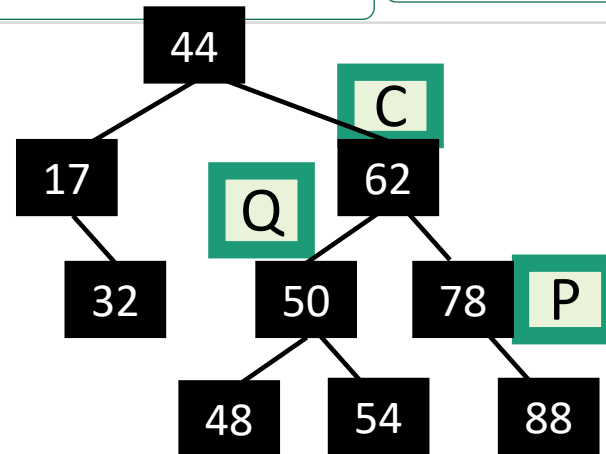
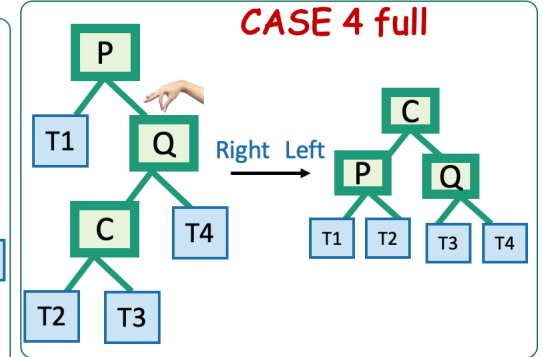
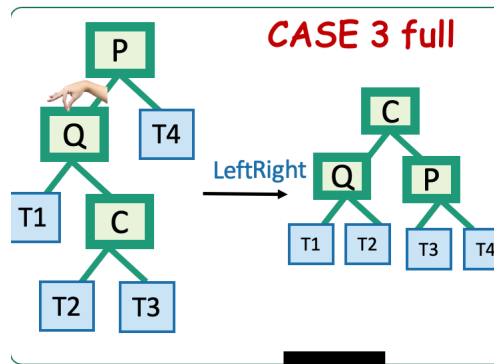
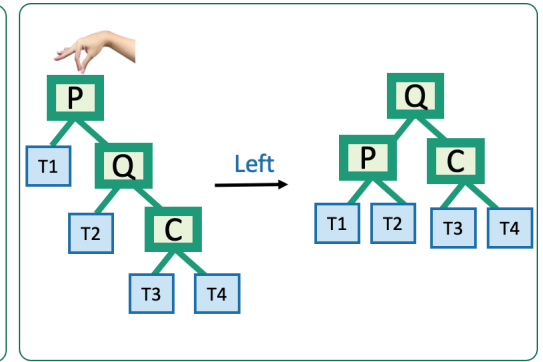
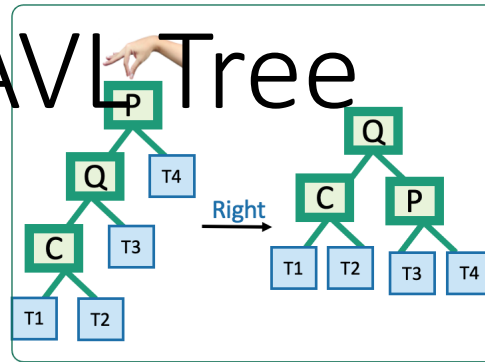
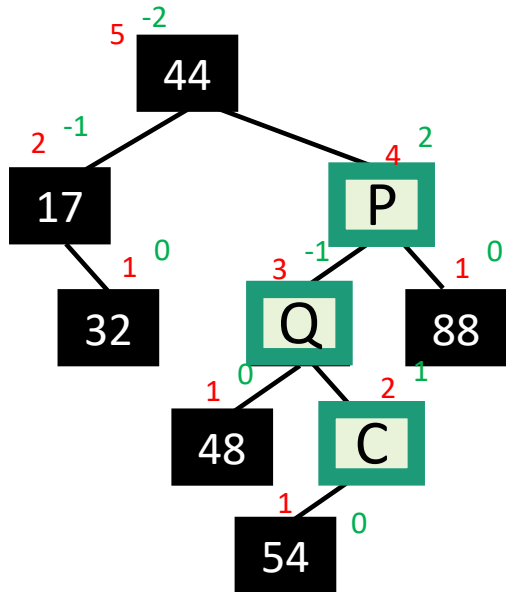
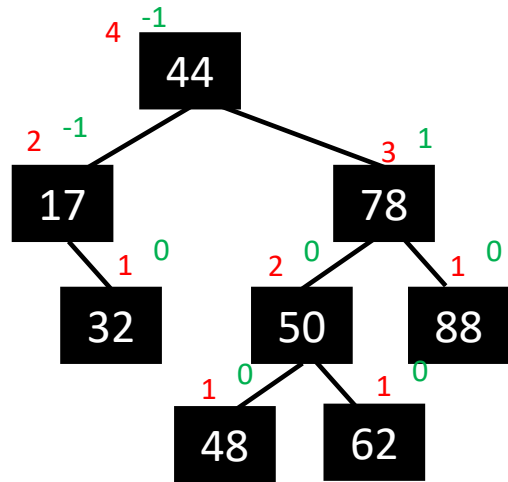
# Inserting into a AVL Tree



# Inserting into a AVL Tree



# Inserting into a AVL Tree



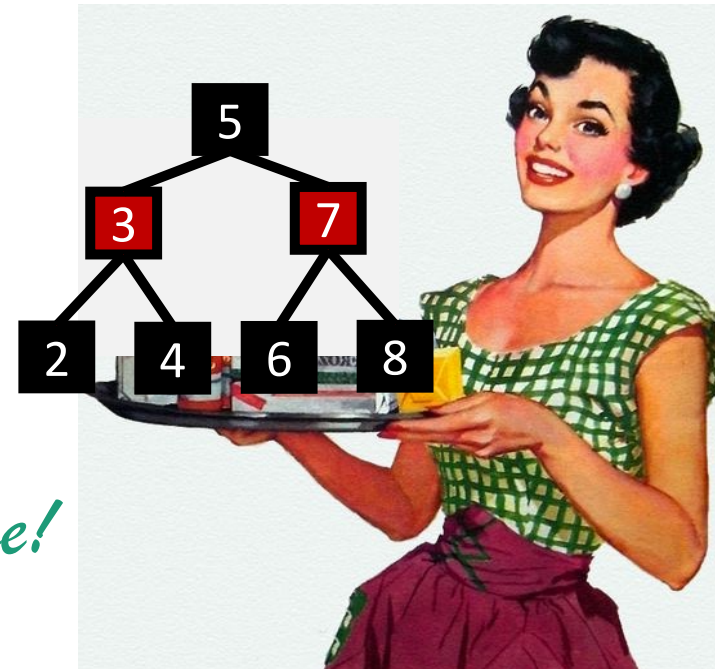
# Red-Black Trees

- 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!*

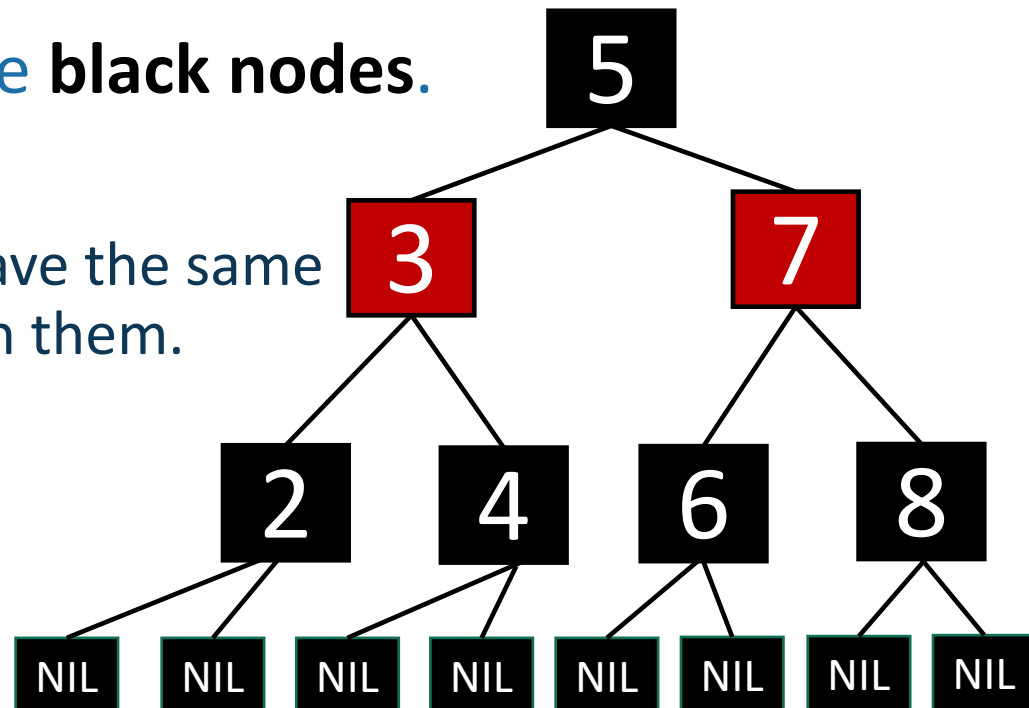




# Red-Black Trees

these rules are the proxy for balance

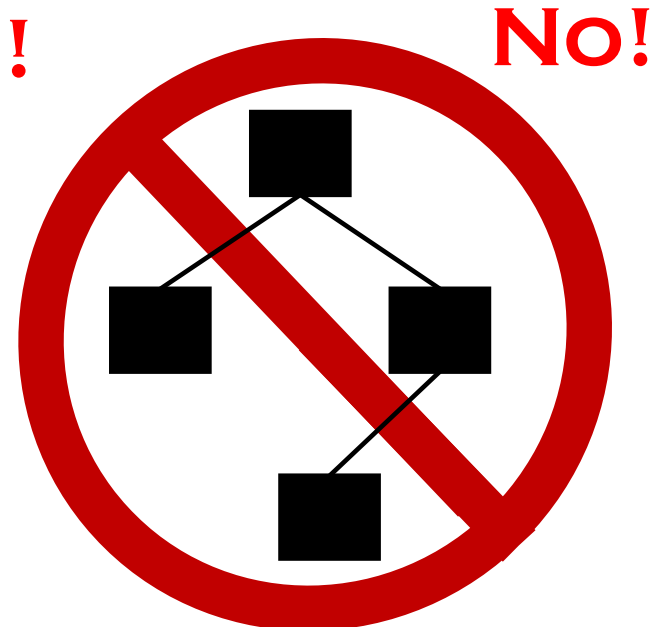
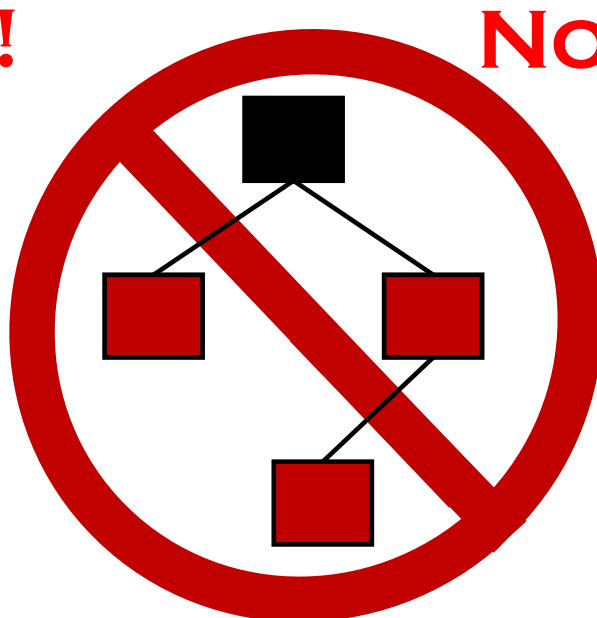
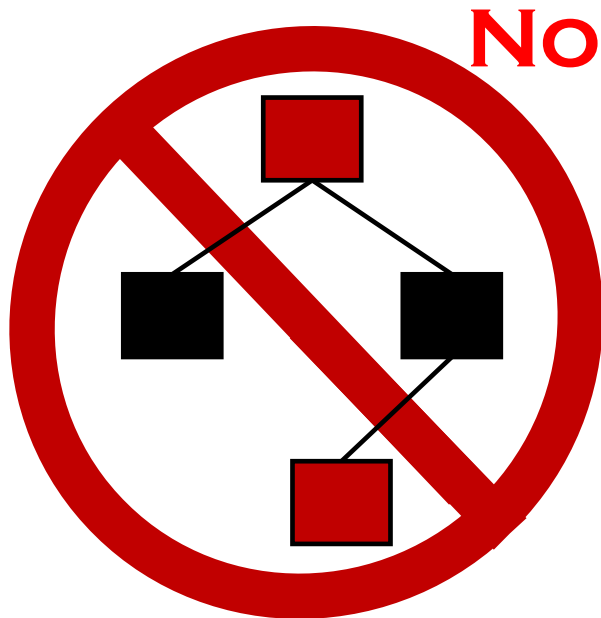
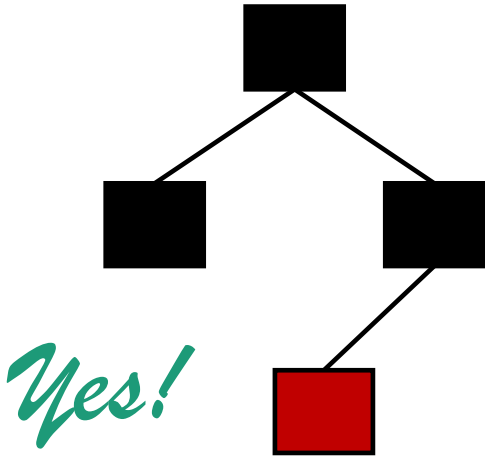
- Every node is colored **red** or **black**.
- The root node is a **black node**.
- NIL children count as **black nodes**.
- Children of a **red node** are **black nodes**.
- For all nodes x:
  - all paths from x to NIL's have the same number of **black nodes** on them.



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

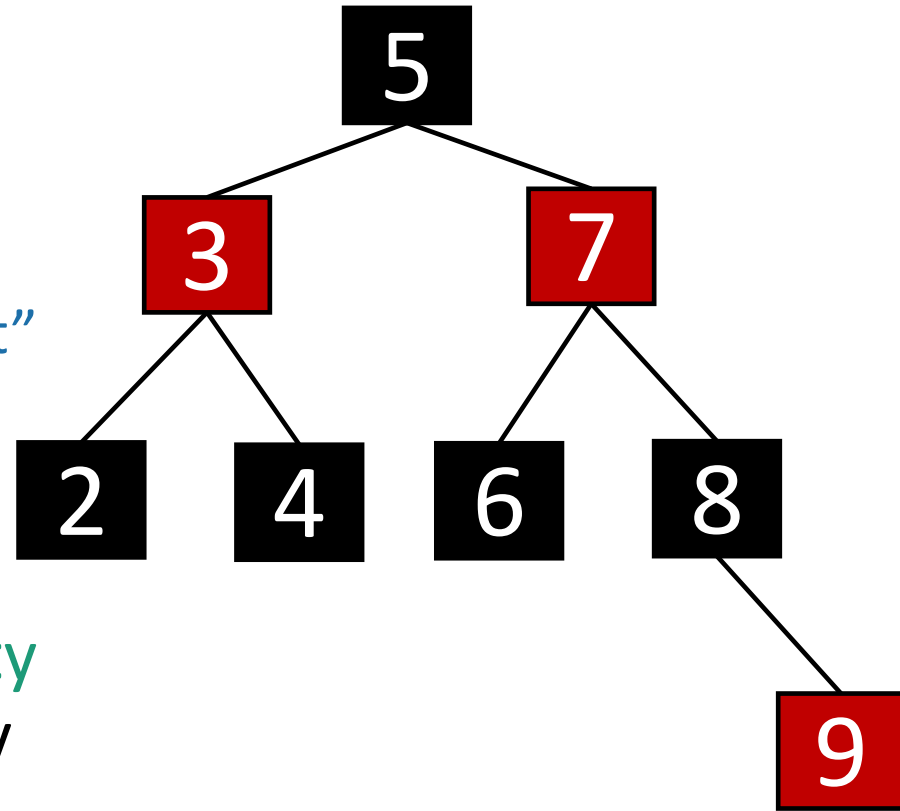
# Examples(?)

- 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.



# Why??????

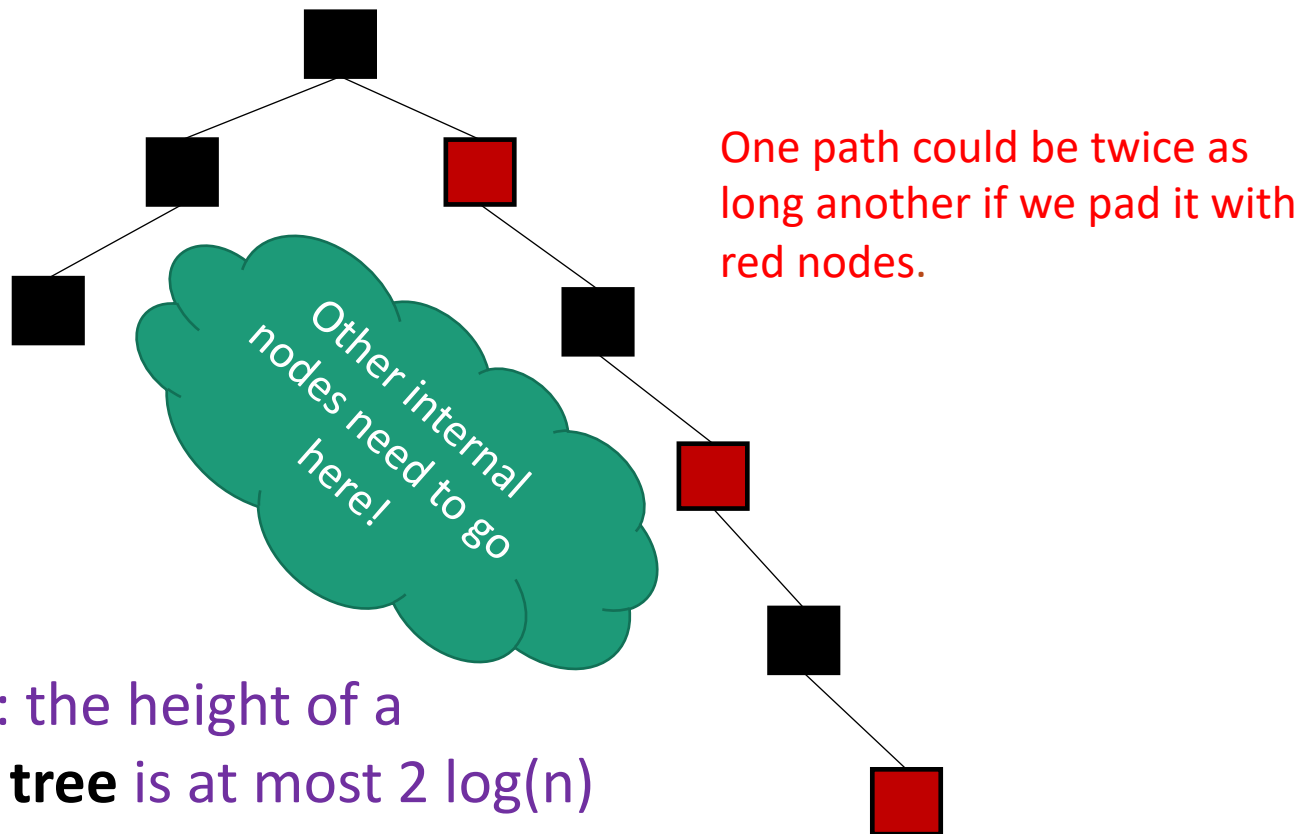
- This is pretty balanced.
  - The **black nodes** are balanced
  - The **red nodes** are “spread out” so they don’t mess things up too much.
- We can maintain this property as we insert/delete nodes, by using rotations.



This is the really clever idea!  
This **Red-Black** structure is a proxy for balance.

# 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** is at most  $2 \log(n)$

Okay, so it's balanced...

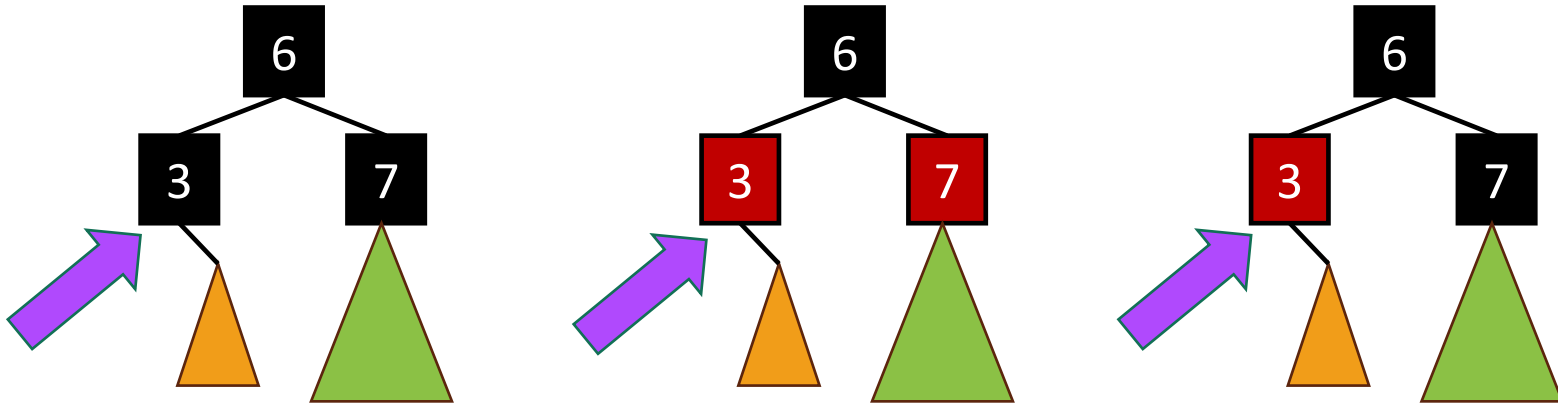
...but can we maintain it?

- Yes!

- See CLRS for more details.

- (You are not responsible for the details for this class – but you should understand the main ideas).

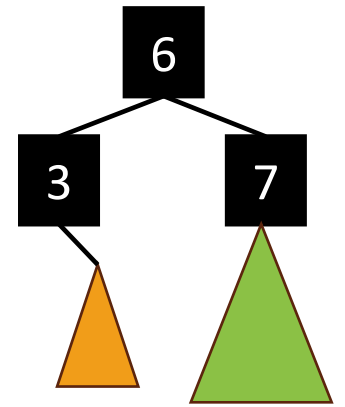
# Many cases



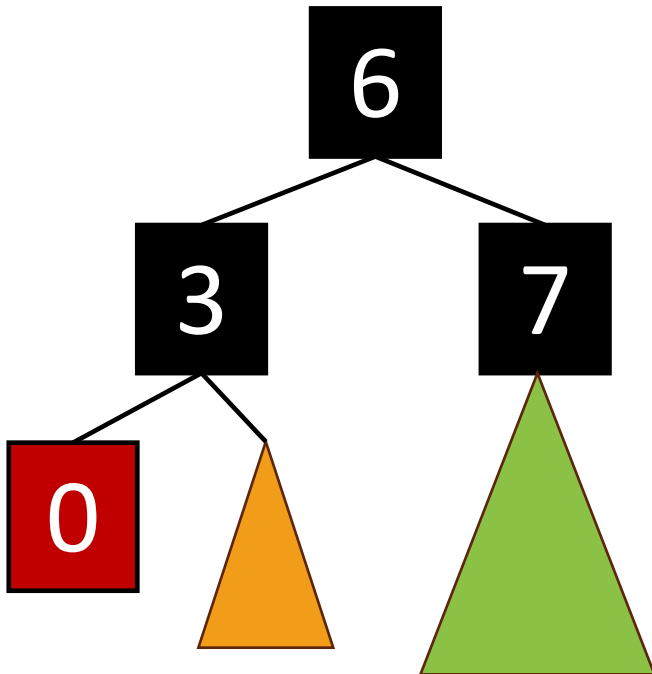
- Suppose we want to insert **here**.
  - eg, want to insert 0.

# Inserting into a Red-Black Tree

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



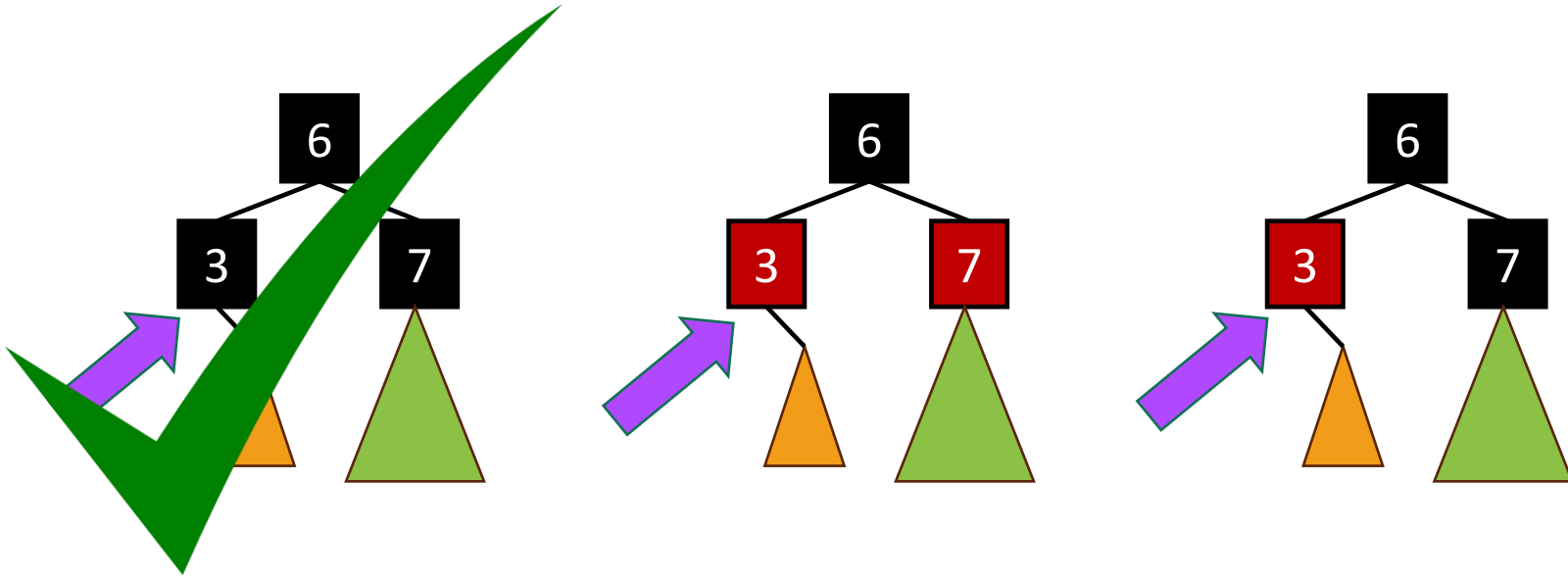
What if it looks like this?



Example: insert 0



# Many cases

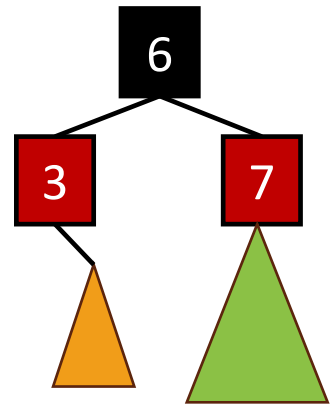


- Suppose we want to insert **here**.
  - eg, want to insert 0.



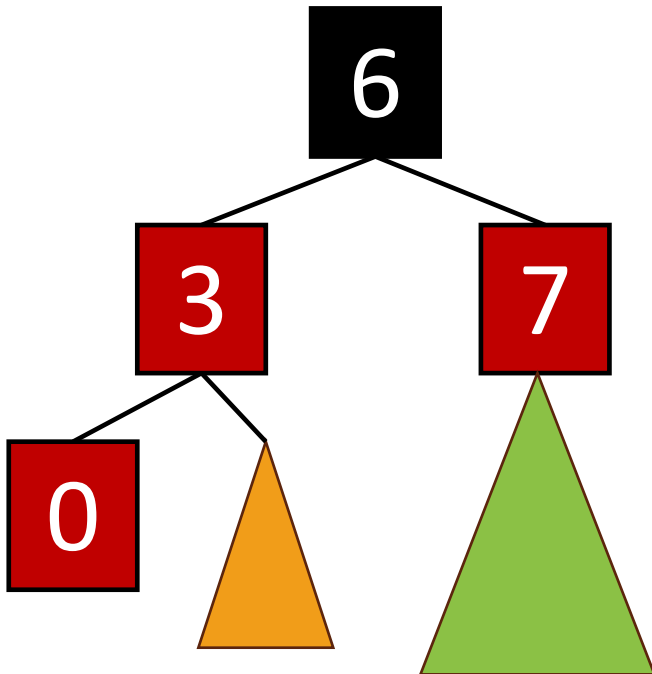
# Inserting into a Red-Black Tree

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



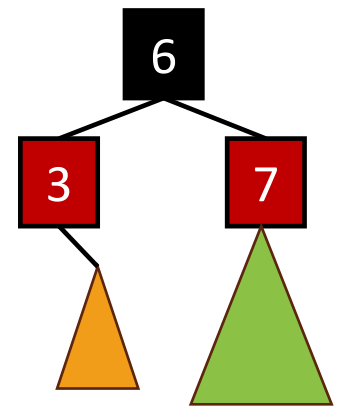
What if it looks like this?

Example: insert 0

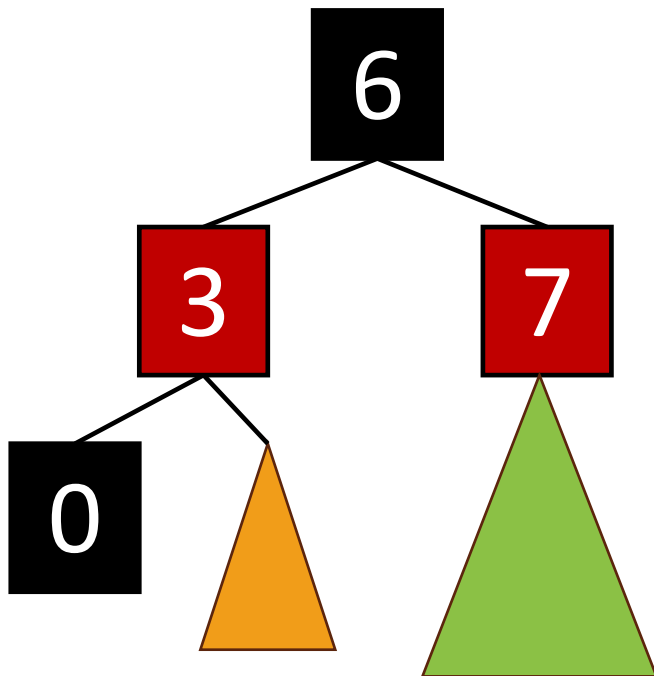


# Inserting into a Red-Black Tree

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



What if it looks like this?

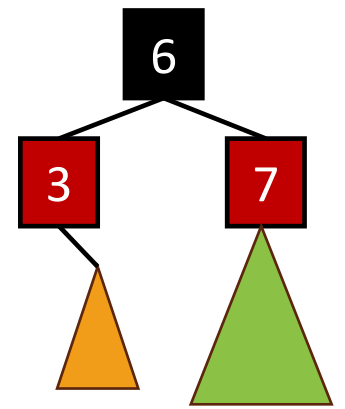


Example: insert 0

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

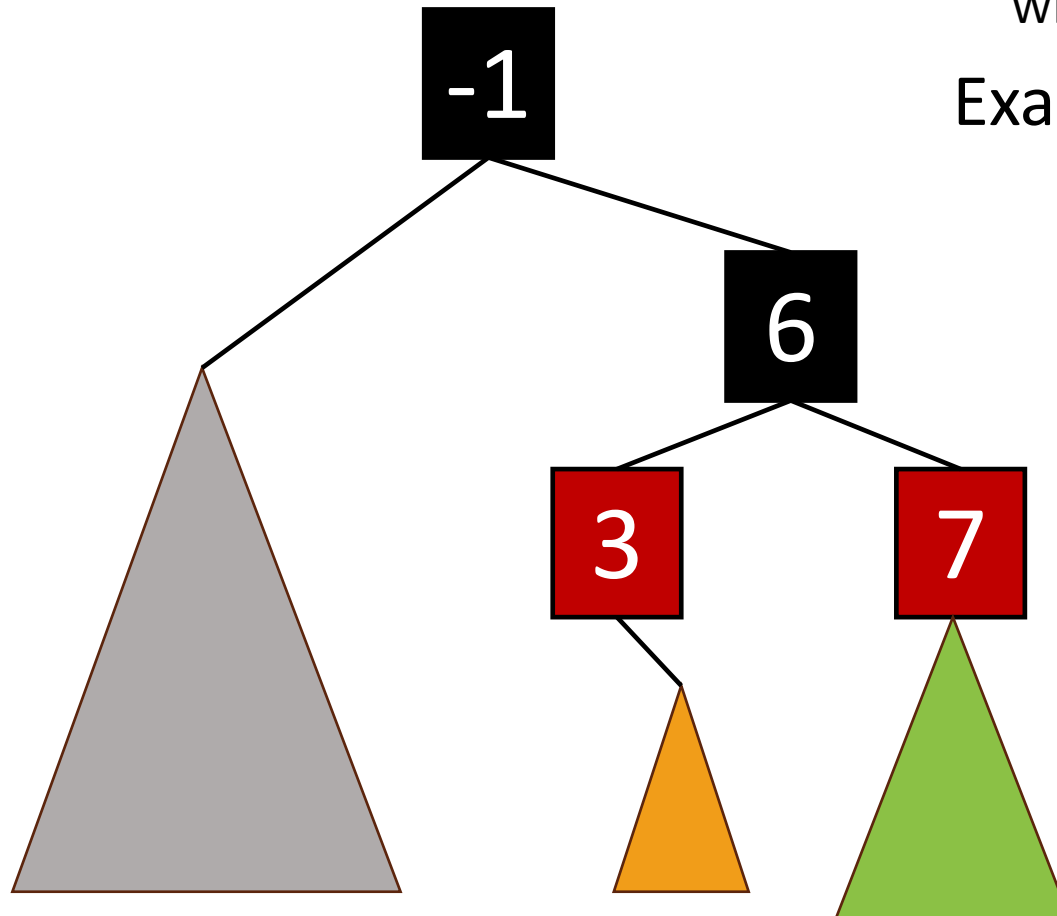


# We need a bit more context



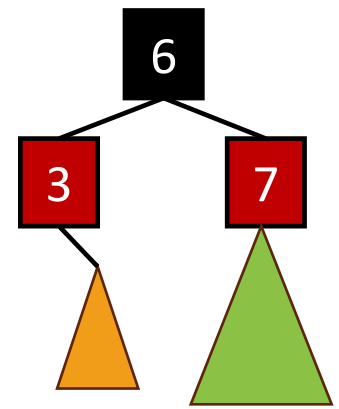
What if it looks like this?

Example: insert 0



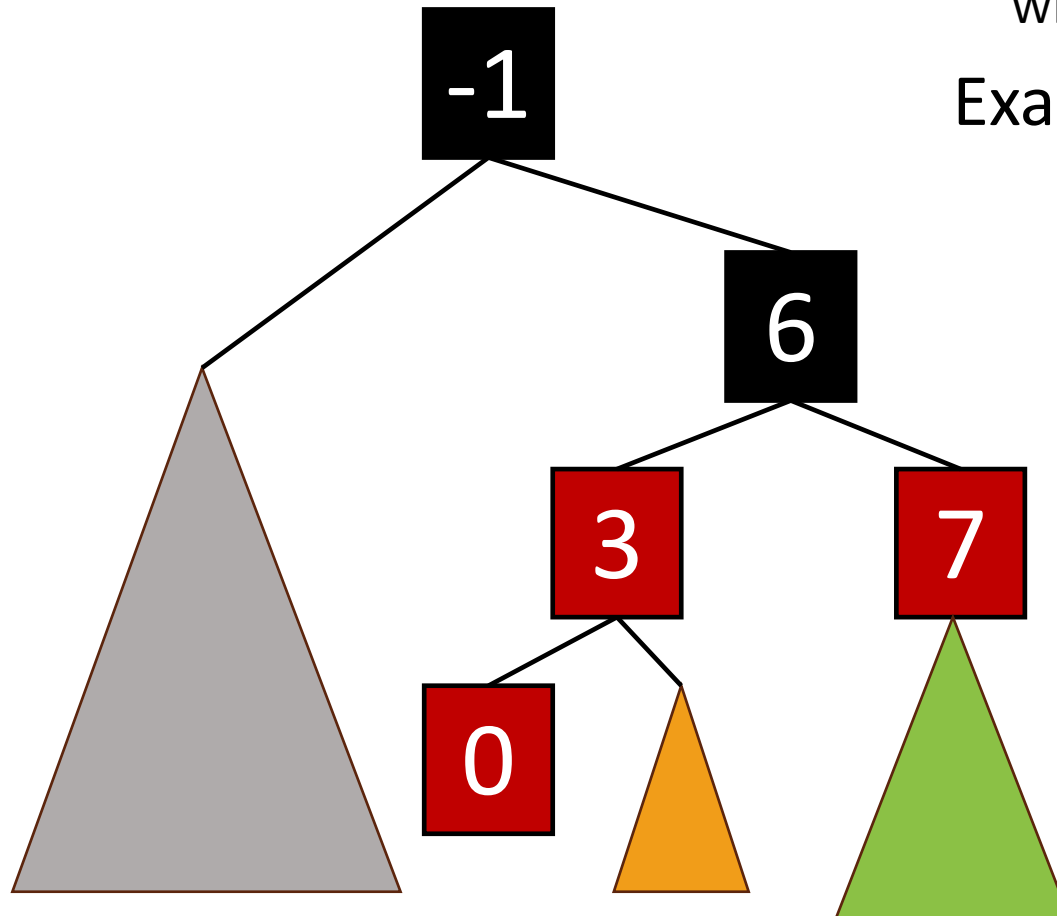
# We need a bit more context

- Add 0 as a red node.



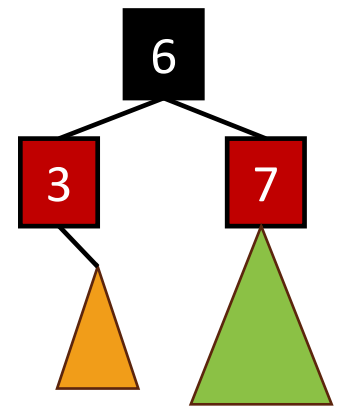
What if it looks like this?

Example: insert 0



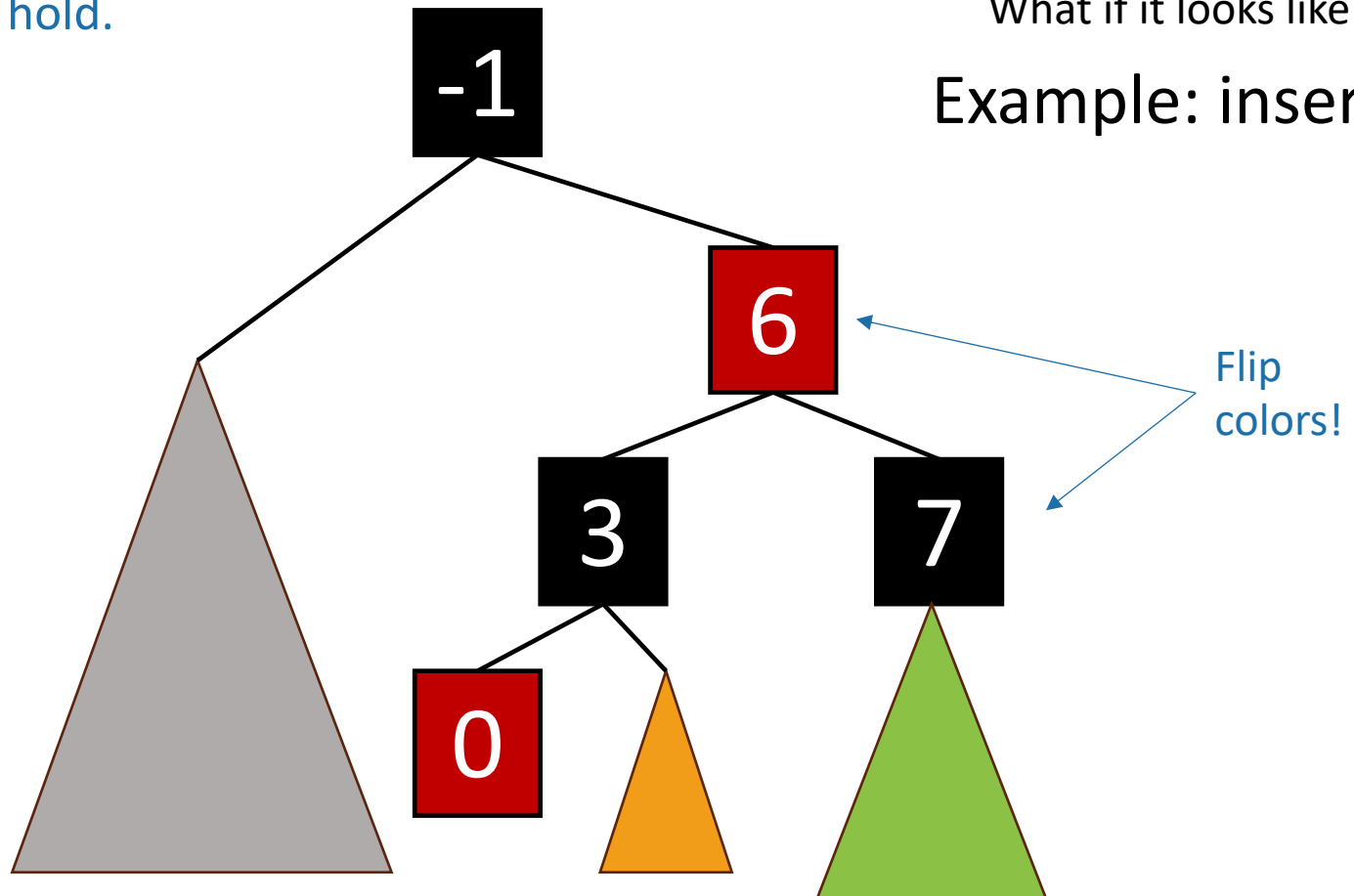
# We need a bit more context

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

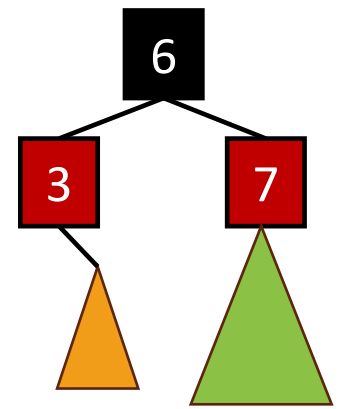
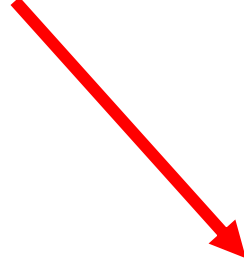


What if it looks like this?

Example: insert 0

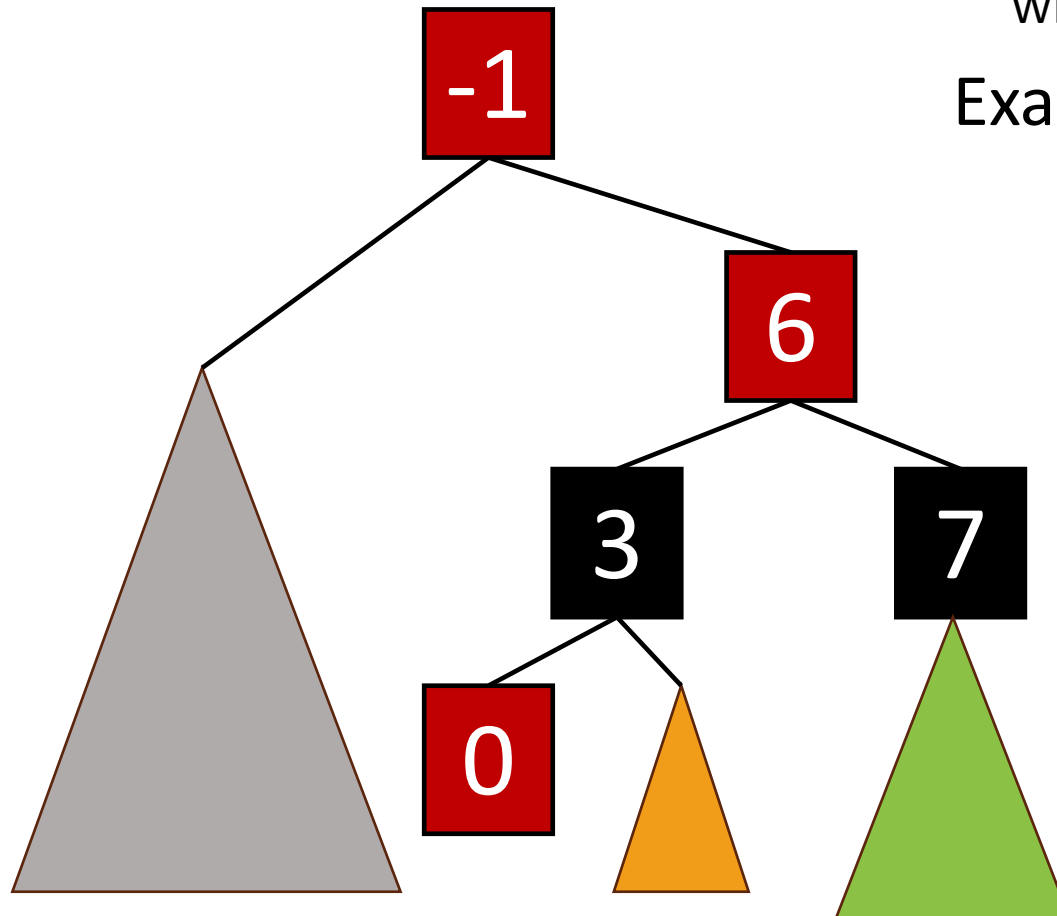


But what if **that** was red?

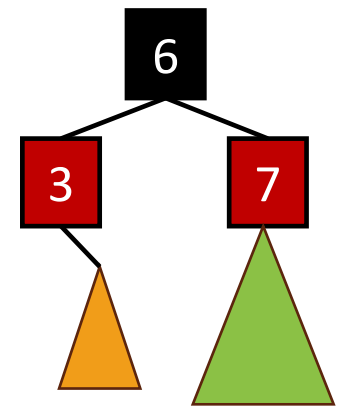
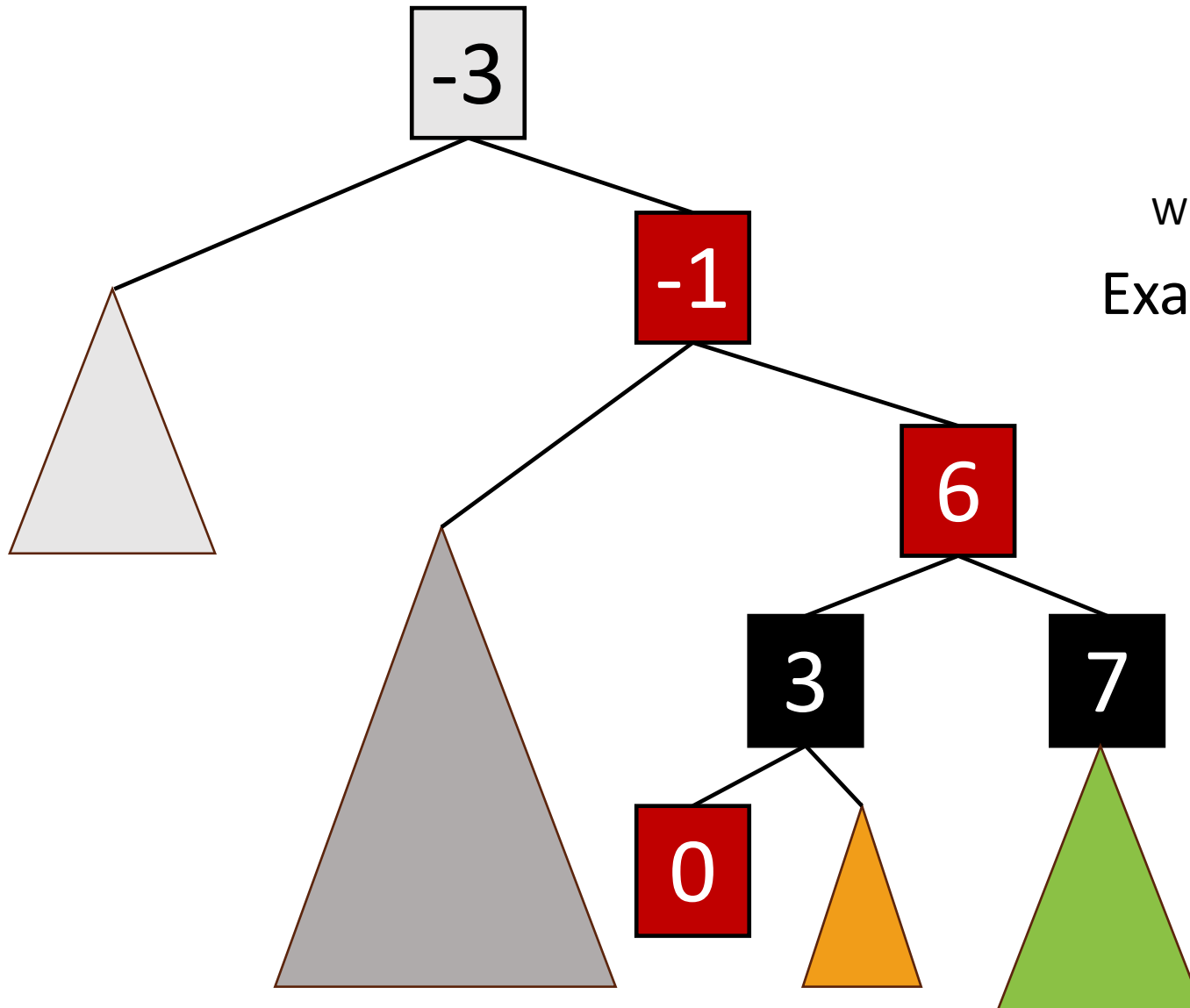


What if it looks like this?

Example: insert 0



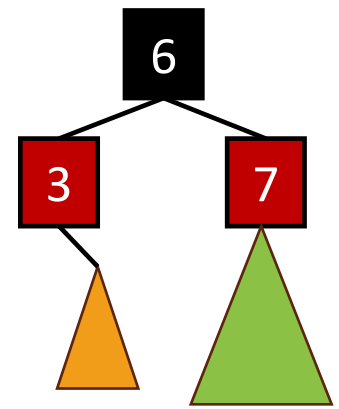
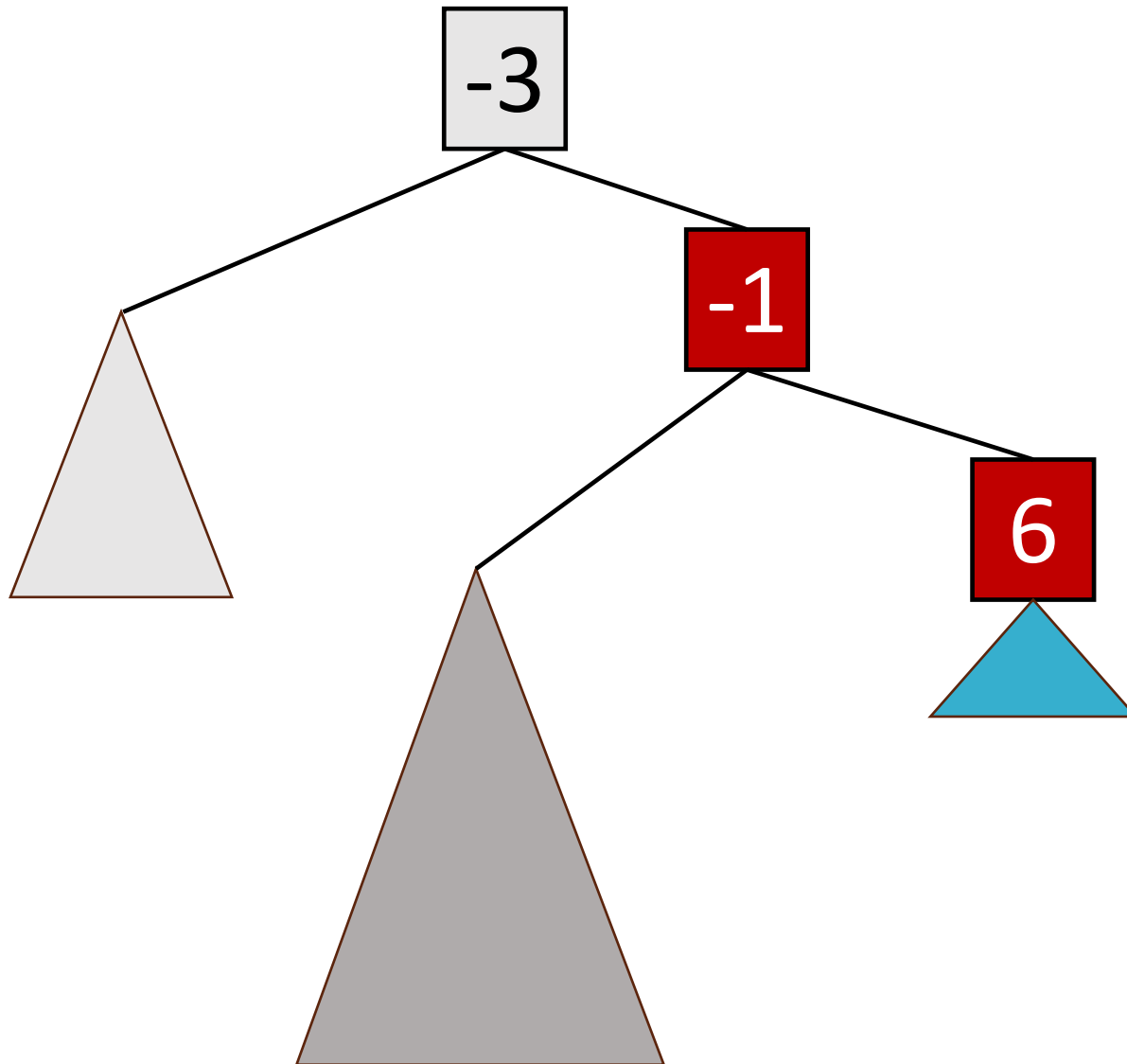
# More context...



What if it looks like this?

Example: insert 0

# More context...



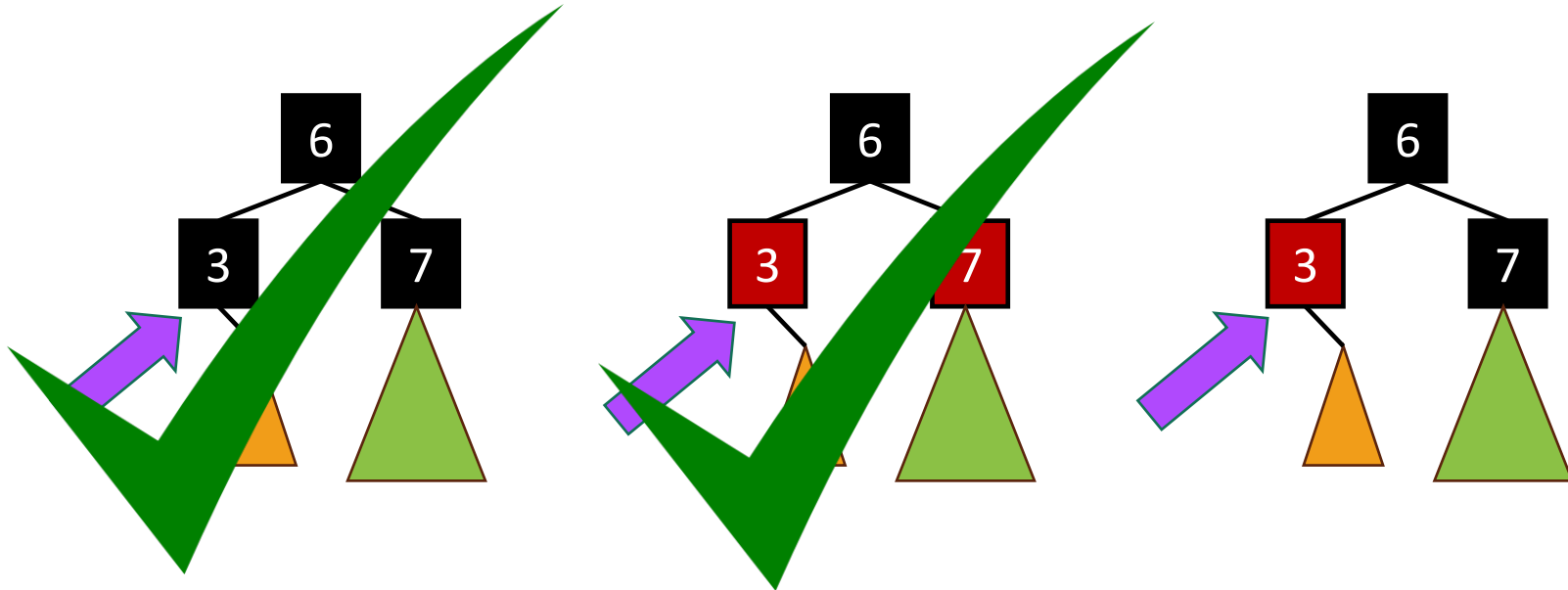
What if it looks like this?

Example: insert 0

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



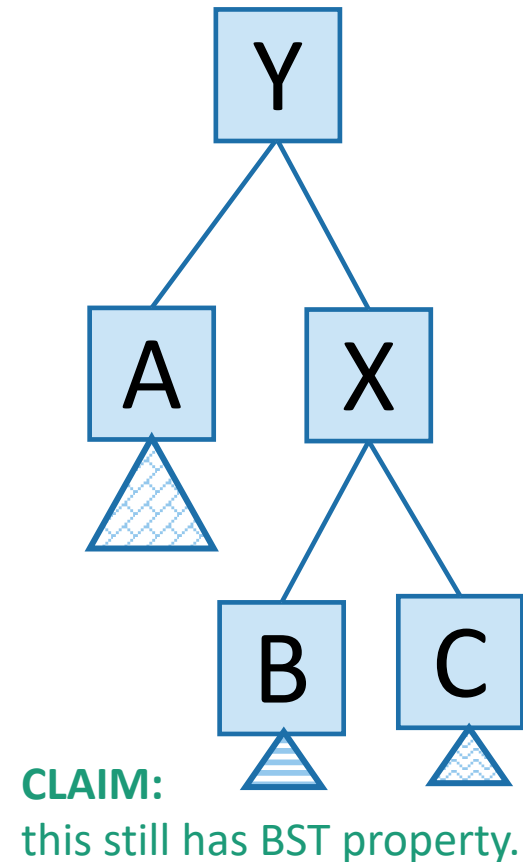
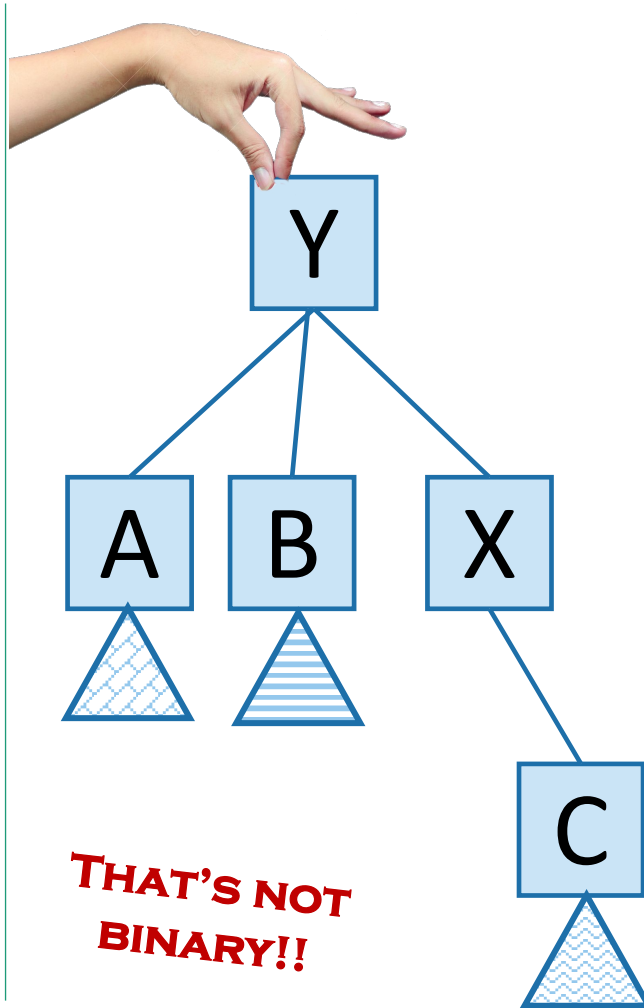
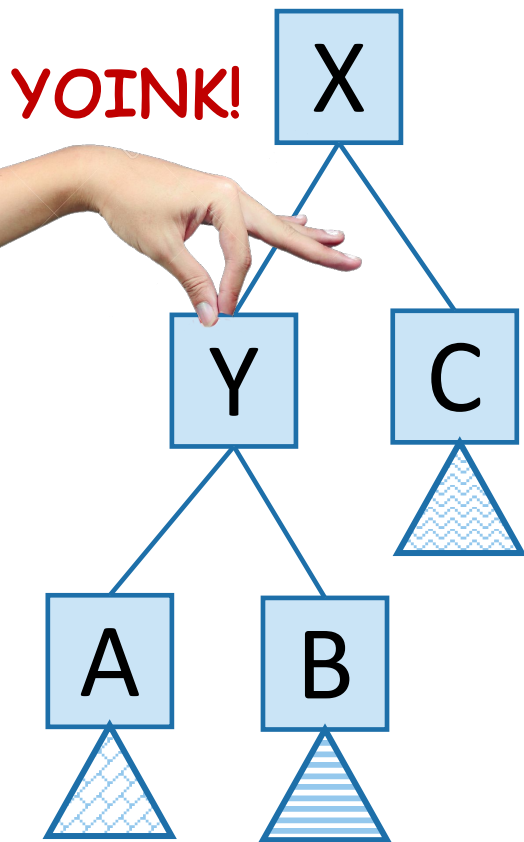
# Many cases



- Suppose we want to insert **here**.
  - eg, want to insert 0.

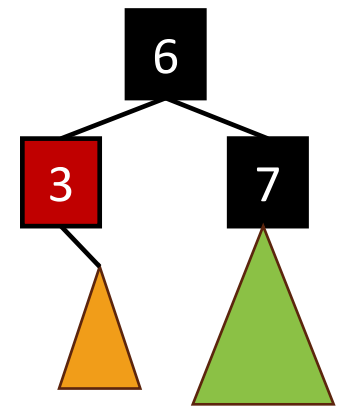
# Recall Rotations

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



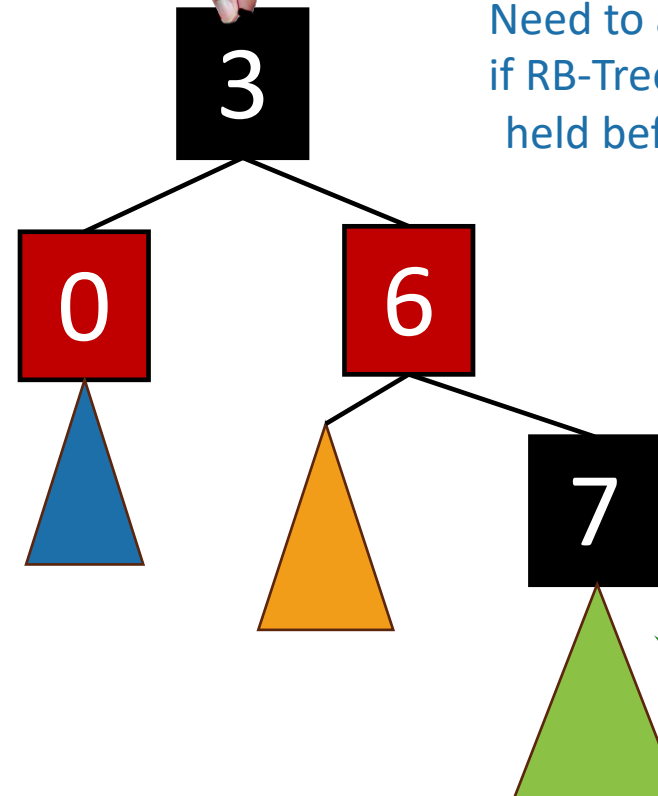
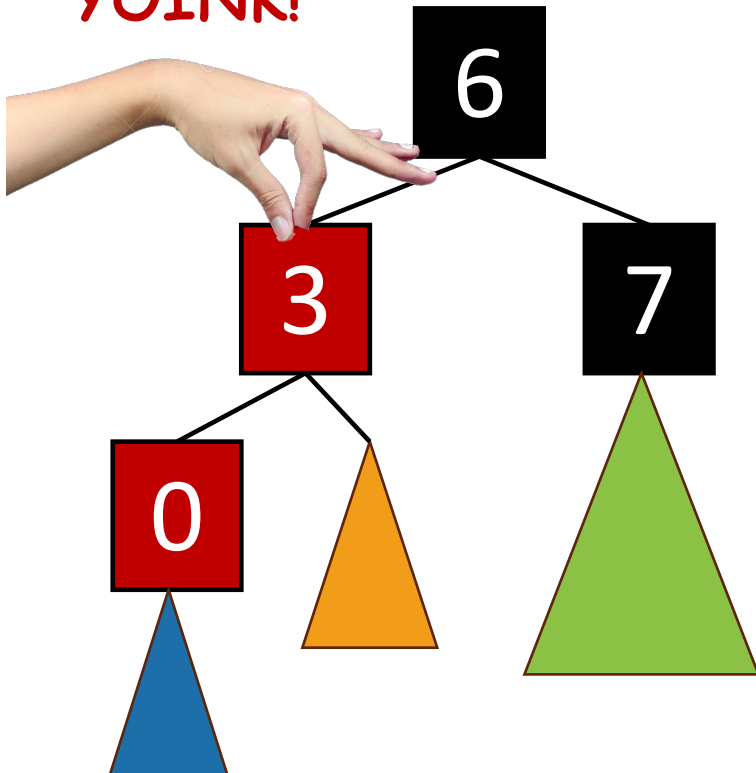
# Inserting into a Red-Black Tree

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



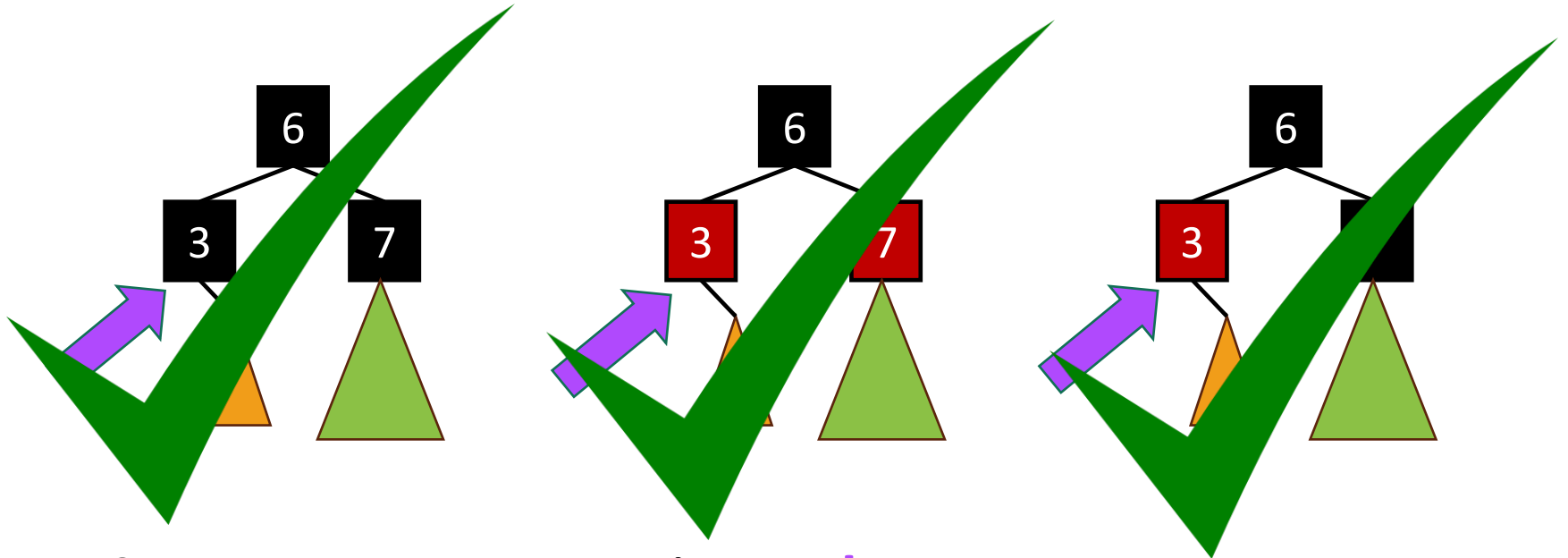
What if it looks like this?

**YOINK!**



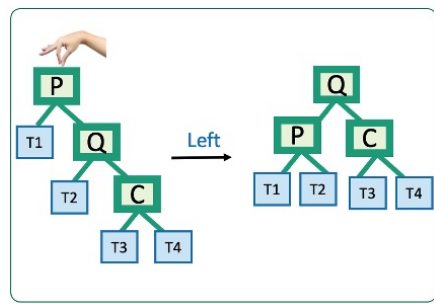
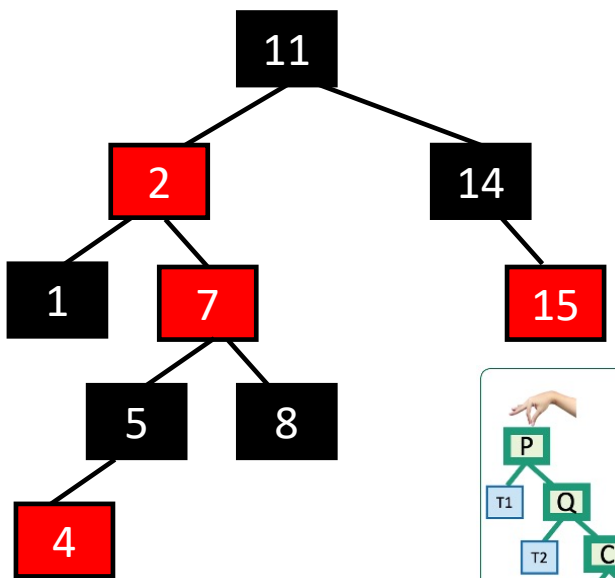
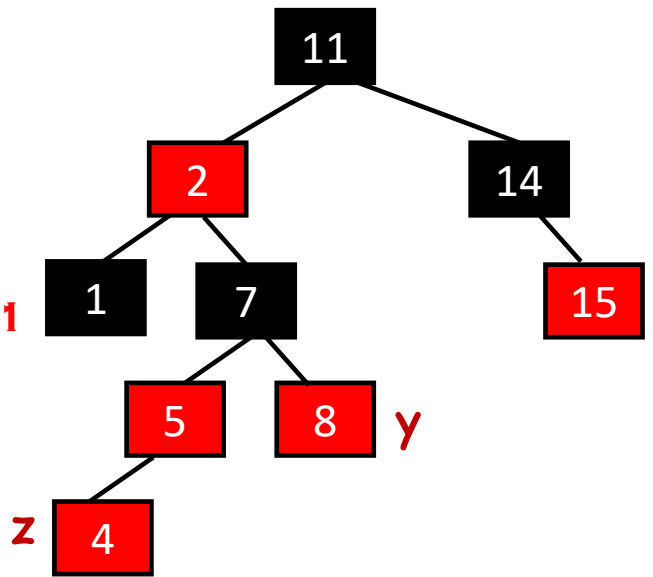
Need to argue that if RB-Tree property held before, it still does.

# Many cases

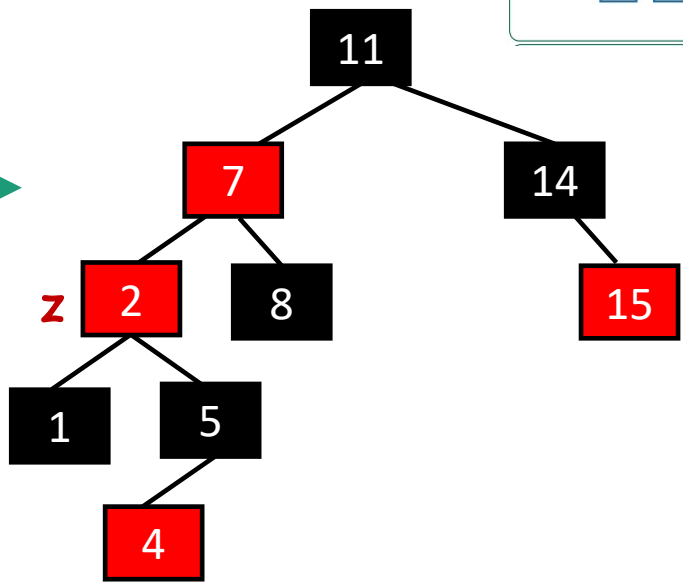
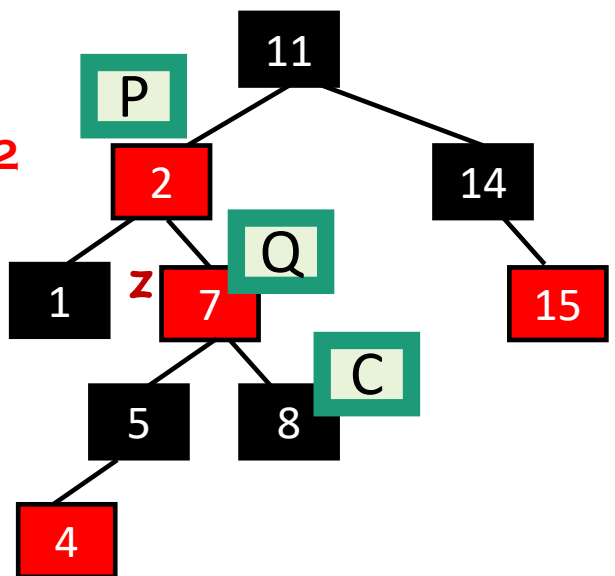


- Suppose we want to insert **here**.
  - eg, want to insert 0.

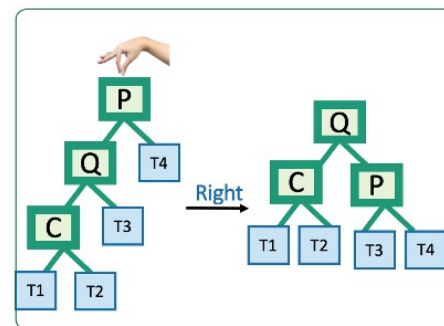
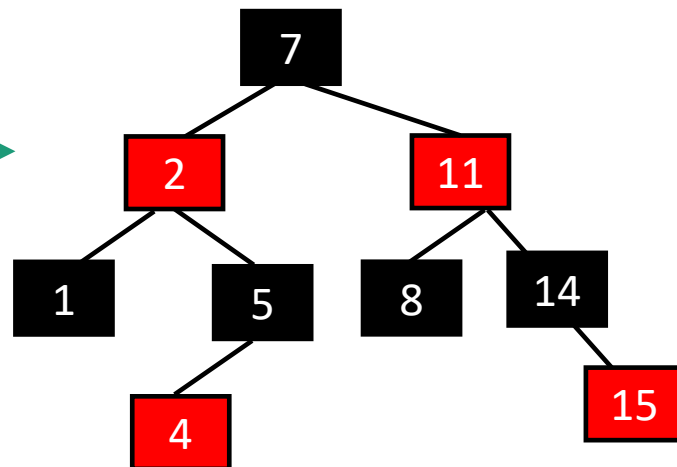
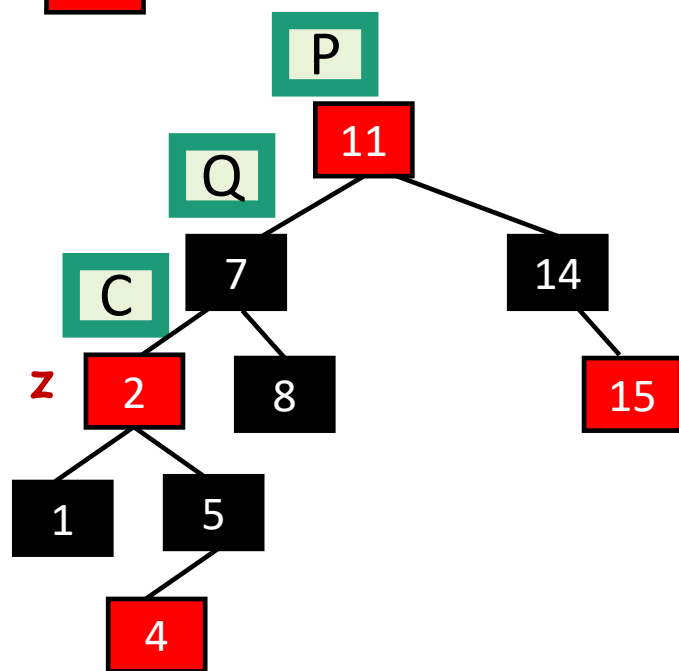
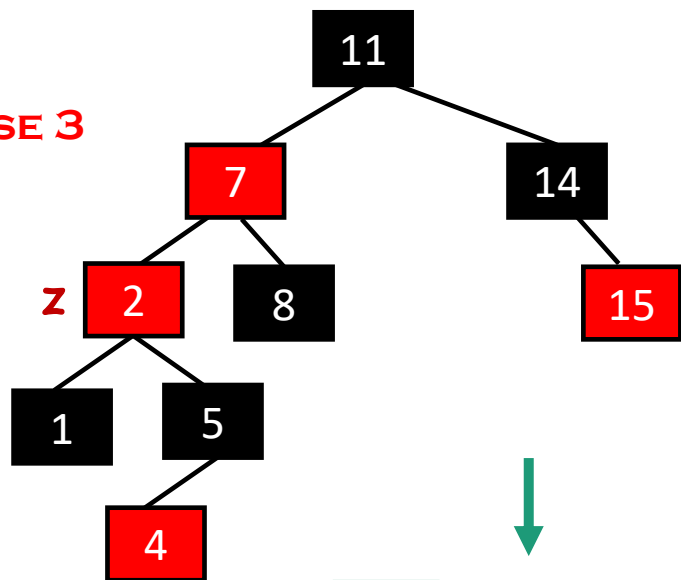
CASE 1



CASE 2



# CASE 3

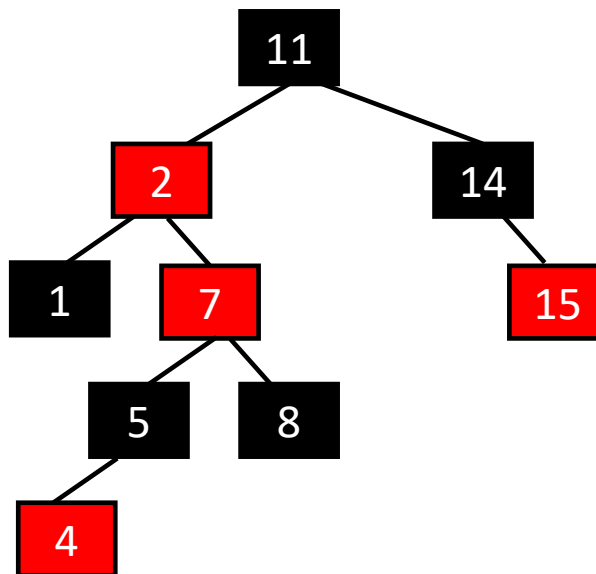
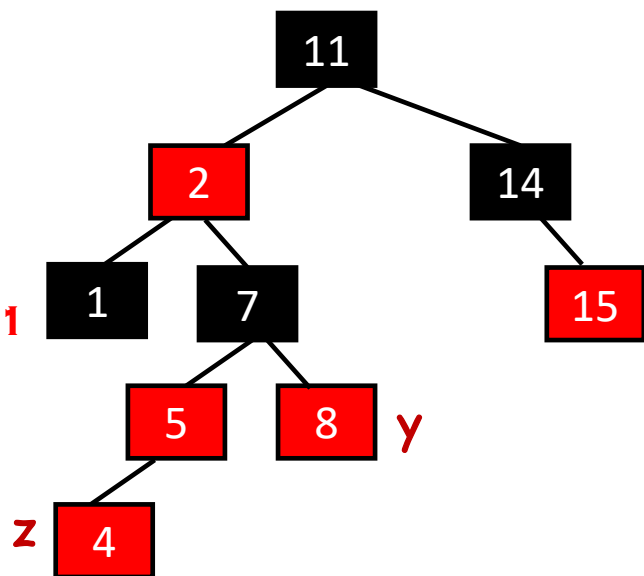


# RB-INSERT-FIXUP( $T, z$ )

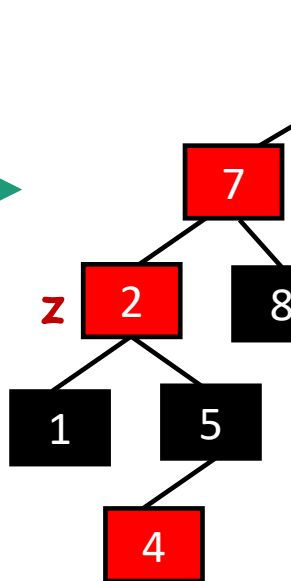
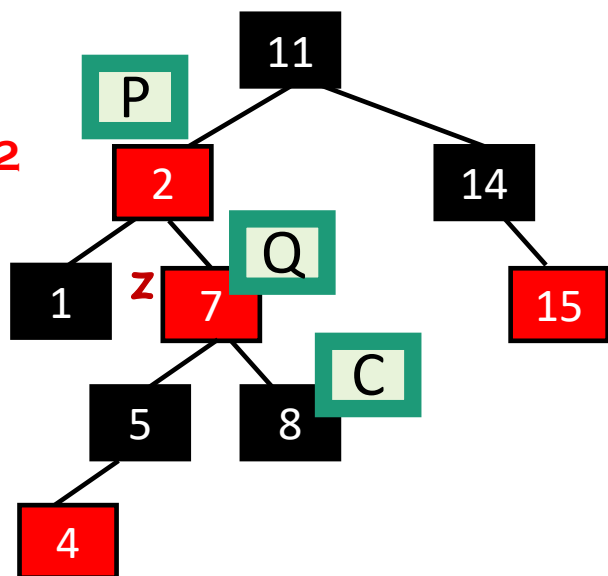
```
1  while  $z.p.color == \text{RED}$ 
2      if  $z.p == z.p.p.left$ 
3           $y = z.p.p.right$ 
4          if  $y.color == \text{RED}$ 
5               $z.p.color = \text{BLACK}$  // case 1
6               $y.color = \text{BLACK}$  // case 1
7               $z.p.p.color = \text{RED}$  // case 1
8               $z = z.p.p$  // case 1
9          else if  $z == z.p.right$ 
10              $z = z.p$  // case 2
11             LEFT-ROTATE( $T, z$ ) // case 2
12              $z.p.color = \text{BLACK}$  // case 3
13              $z.p.p.color = \text{RED}$  // case 3
14             RIGHT-ROTATE( $T, z.p.p$ ) // case 3
15         else (same as then clause
16             with “right” and “left” exchanged)
```

```
16   $T.root.color = \text{BLACK}$ 
```

SE 1



SE 2



RB-INSERT-FIXUP( $T, z$ )

```

1  while  $z.p.color == RED$ 
2      if  $z.p == z.p.p.left$ 
3           $y = z.p.p.right$ 
4          if  $y.color == RED$ 
5               $z.p.color = BLACK$  // case 1
6               $y.color = BLACK$  // case 1
7               $z.p.p.color = RED$  // case 1
8               $z = z.p.p$  // case 1
9          else if  $z == z.p.right$ 
10              $z = z.p$  // case 2
11             LEFT-ROTATE( $T, z$ ) // case 2
12              $z.p.color = BLACK$  // case 3
13              $z.p.p.color = RED$  // case 3
14             RIGHT-ROTATE( $T, z.p.p$ ) // case 3
15         else (same as then clause
16             with "right" and "left" exchanged)
17      $T.root.color = BLACK$ 

```



# Deleting from a Red-Black tree

**Fun exercise!**







# That's a lot of cases

- You are **not responsible** for the nitty-gritty details of Red-Black Trees. (For this class)
  - Though implementing them is a great exercise!
- You should know:
  - What are the properties of an RB tree?
  - And (more important) why does that guarantee that they are balanced?

# What was the point again?

- 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 are  $O(\log(n))$ .

# Conclusion: The best of both worlds

	Sorted Arrays	Linked Lists	Balanced Binary Search Trees
Search	$O(\log(n))$ 	$O(n)$ 	$O(\log(n))$ 
Insert/Delete	$O(n)$ 	$O(1)$ 	$O(\log(n))$ 

# 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**

