COMP 285

Analysis of Algorithms

Welcome to COMP 285

Lecture 11: BSTs + Self-Balancing Trees

Lecturer: Chris Lucas (cflucas@ncat.edu)

HW2

Grades were released!

HW3

Due Today @ 11:59pm

HW4

Released by EoD! Due 10/13!

Midterm Approaching!

10/06 @ 2pm! Lectures 0-11

Midterm Approaching!

10/06 @ 2pm! Lectures 0-11

Written exam, similar to quizzes, homework assignments, coding

Midterm Approaching!

Review topics?

Written exam, similar to quizzes, homework assignments, coding

Extra Credit Opportunities!

- Technical Interview Prep with Meta (Oct.) +0.5%
- Technical Interview Prep with Chris (Wed) +0.5%, up to 1%
- Midpoint survey (link) +1%

Quiz!

www.comp285-fall22.ml

Recall where we ended last lecture…

Binary tree terminology

Each node has two children.

The left child of 3 is 2

The right child of 3 is 4

The parent of 3 is 5

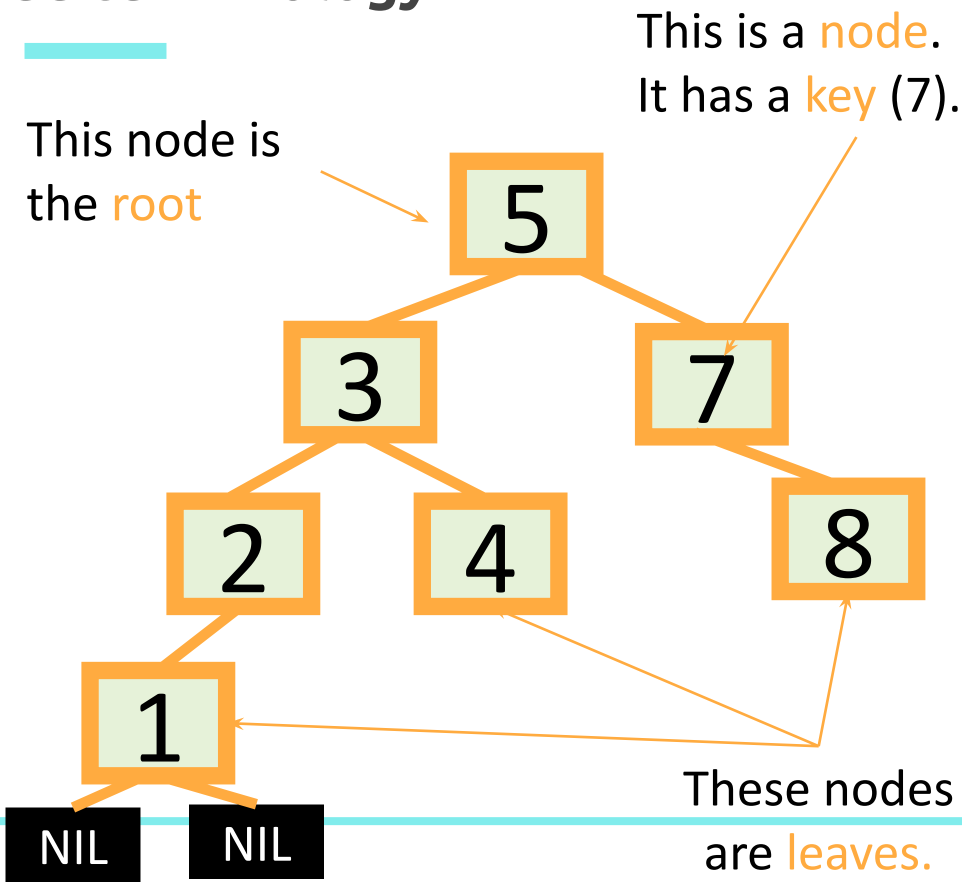
2 is a descendant of 5

Each node has a pointer to its left child, right child, and parent.

Both children of 1 NIL.

(I won't usually draw them).

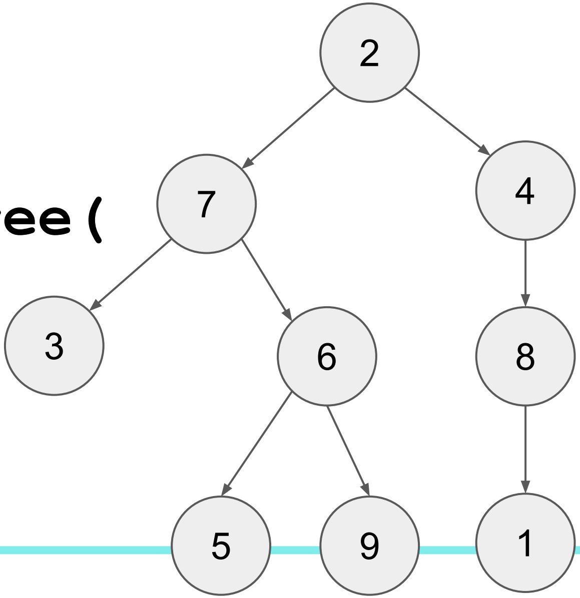
The height of this tree is 3. (Max number of edges from the root to a leaf).



findMaxOfTree

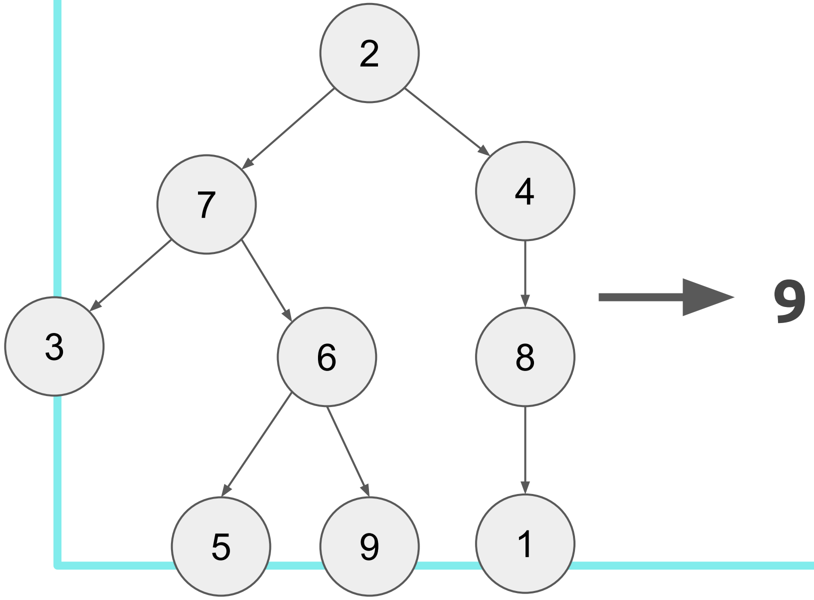
Instructions: *Write an algorithm that takes in a tree of ints, and returns the max of all the values within the tree.*

findMaxOfTree(



) outputs 9

Write an algorithm that takes in a tree of ints, and returns the max of all the values within the tree.

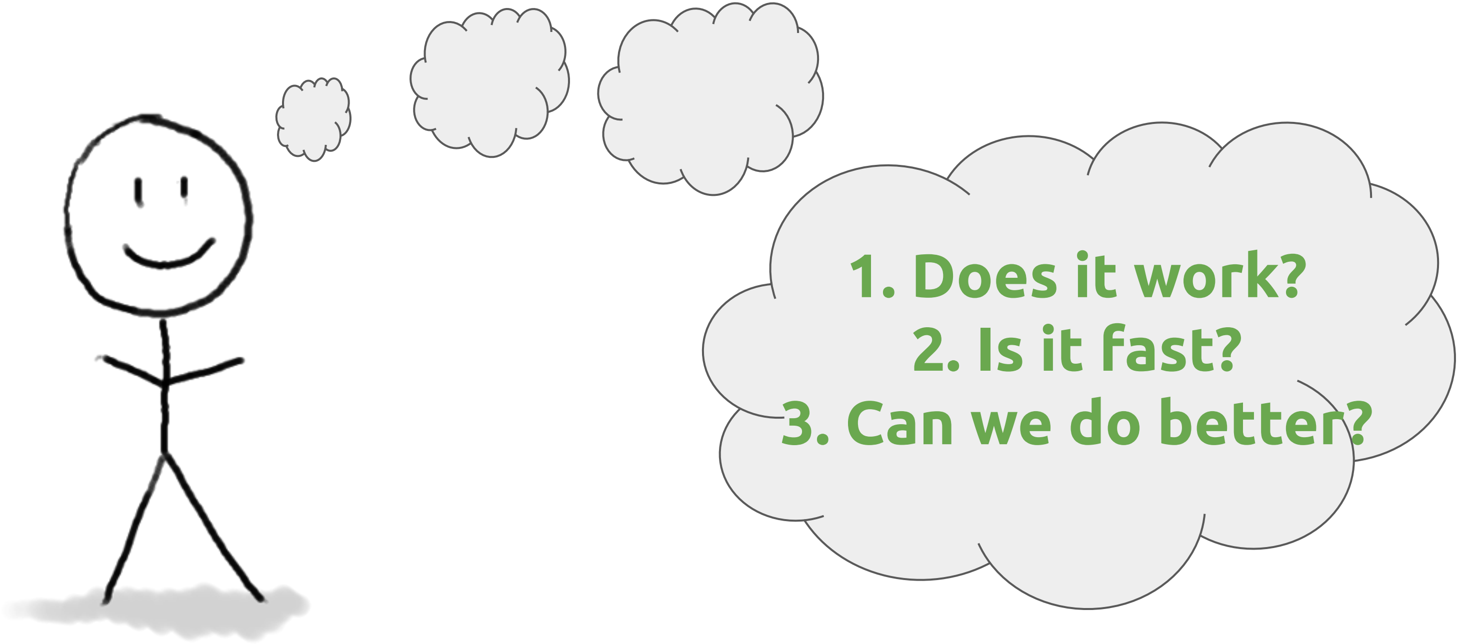



```
int findMaxOfTree(TreeNode<int>* root){
 if (root->isLeaf()) {
   return root->getValue();
 }
 int leftMax = INT_MIN;
 if (root->getLeft()) {
   leftMax = fMOT(root->getLeft());
 }
 int rightMax = INT_MIN;
 if (root->getRight()) {
   rightMax = fMOT(root->getRight());
 }
 return std::max({
             root->getValue(),
             leftMax,
             rightMax
         });
}
```

Big Questions!

- What are/Why binary search trees (BST)?
- Why does balance matter?
- How do we maintain balance?

Our Guiding Questions…



Motivation for Binary Search Trees

	Sorted Arrays	Linked Lists	(balanced) Binary Search Trees
Search			
Delete			
Insert			

Motivation for Binary Search Trees

	Sorted Arrays	Linked Lists	(balanced) Binary Search Trees
Search	O(log(n))		
Delete	O(n)		
Insert	O(n)		

Motivation for Binary Search Trees

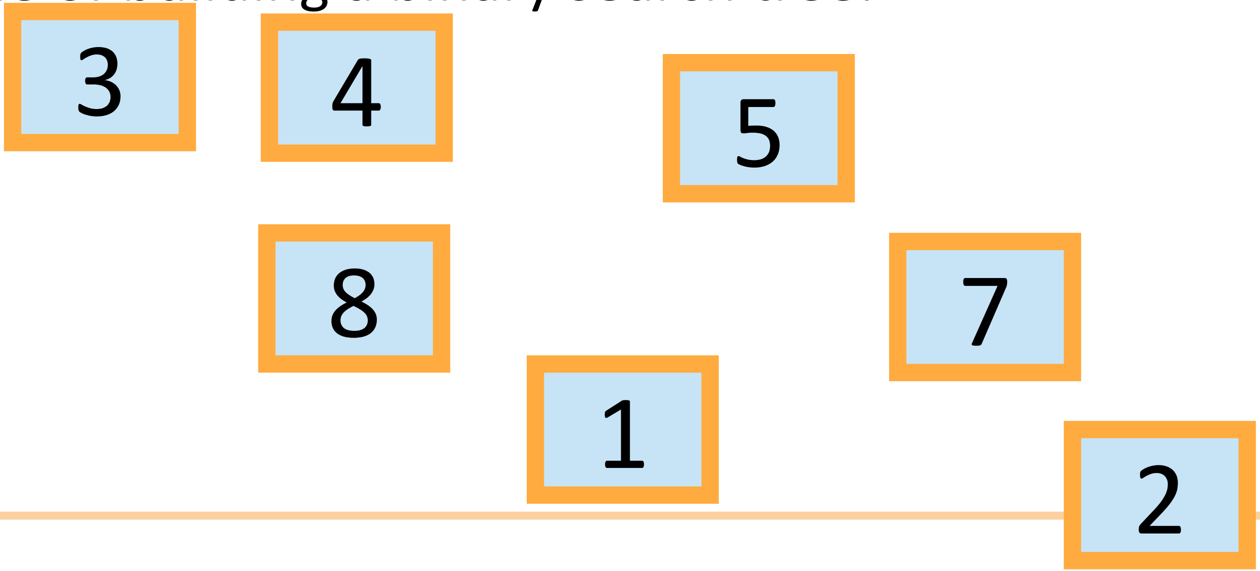
	Sorted Arrays	Linked Lists	(balanced) Binary Search Trees
Search	O(log(n))	O(n)	
Delete	O(n)	O(n)	
Insert	O(n)	O(1)	

Motivation for Binary Search Trees

	Sorted Arrays	Linked Lists	(balanced) Binary Search 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))

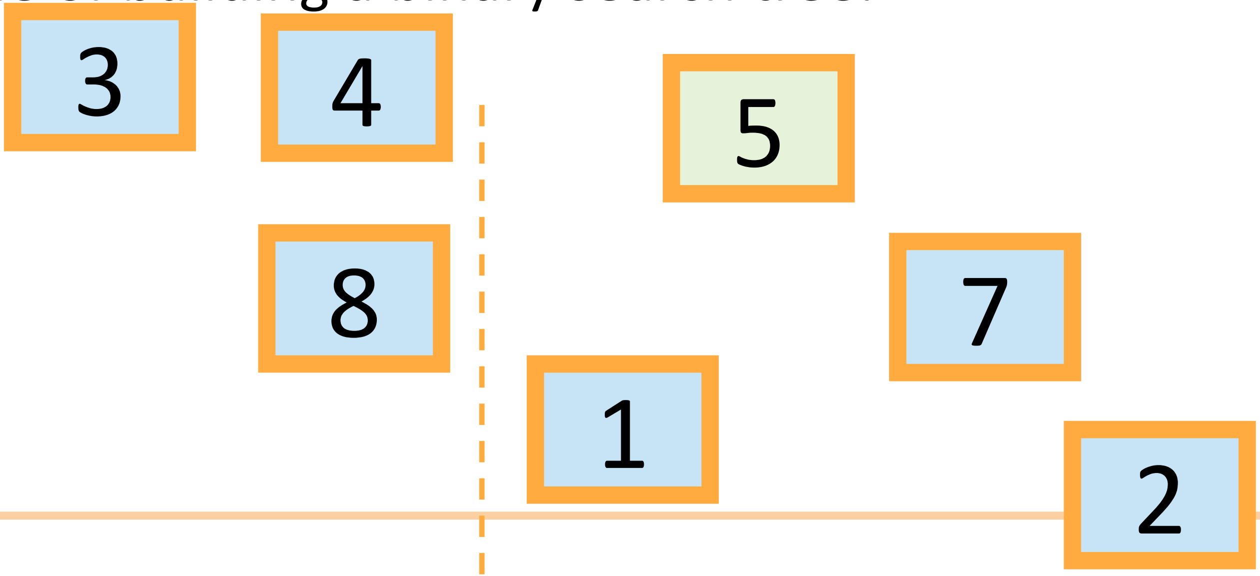
Binary Search Trees

- A BST is 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:



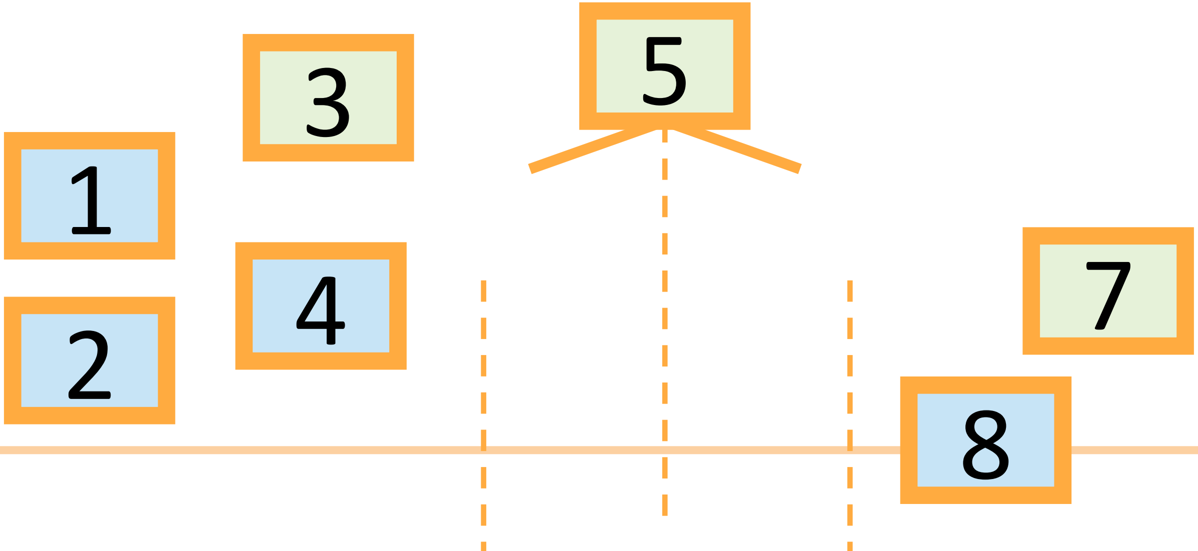
Binary Search Trees

- A BST is 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:



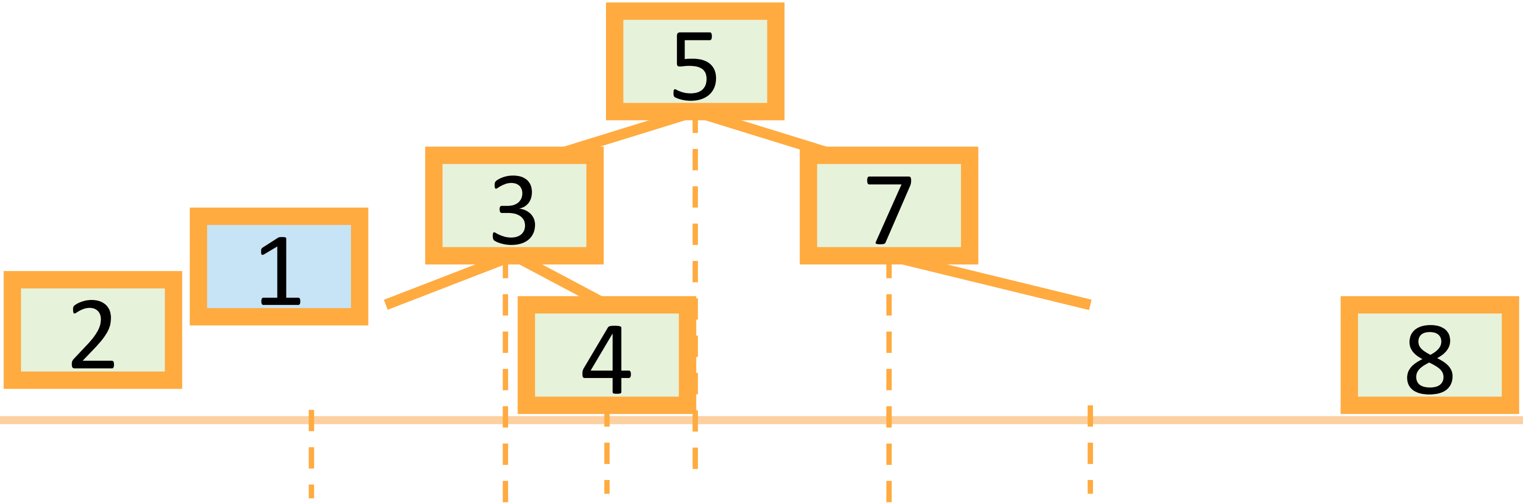
Binary Search Trees

- A BST is 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:



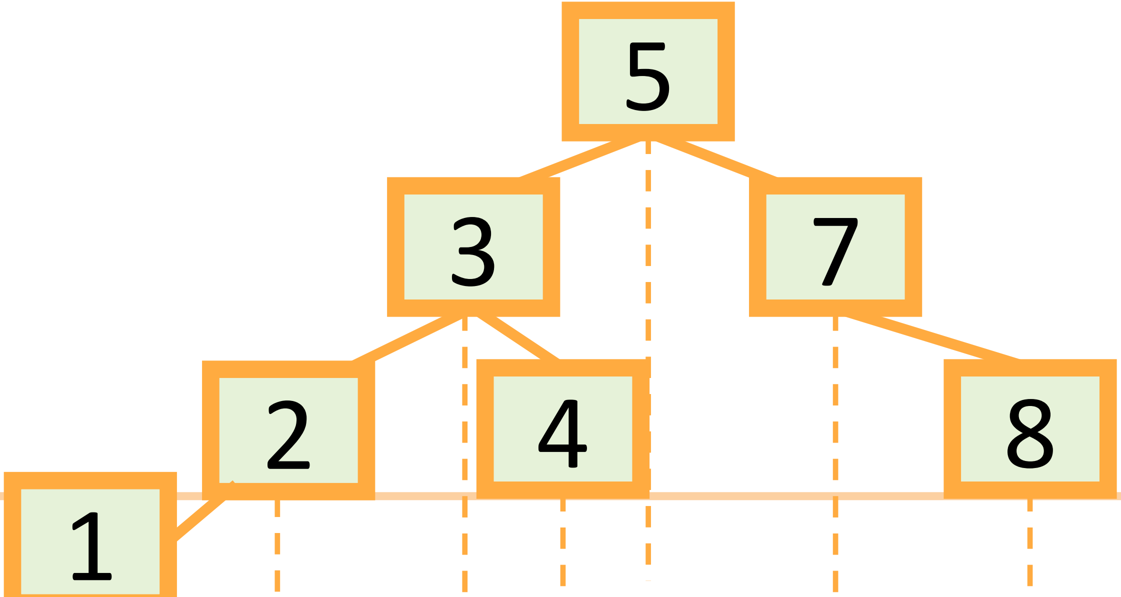
Binary Search Trees

- A BST is 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

- A BST is 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:

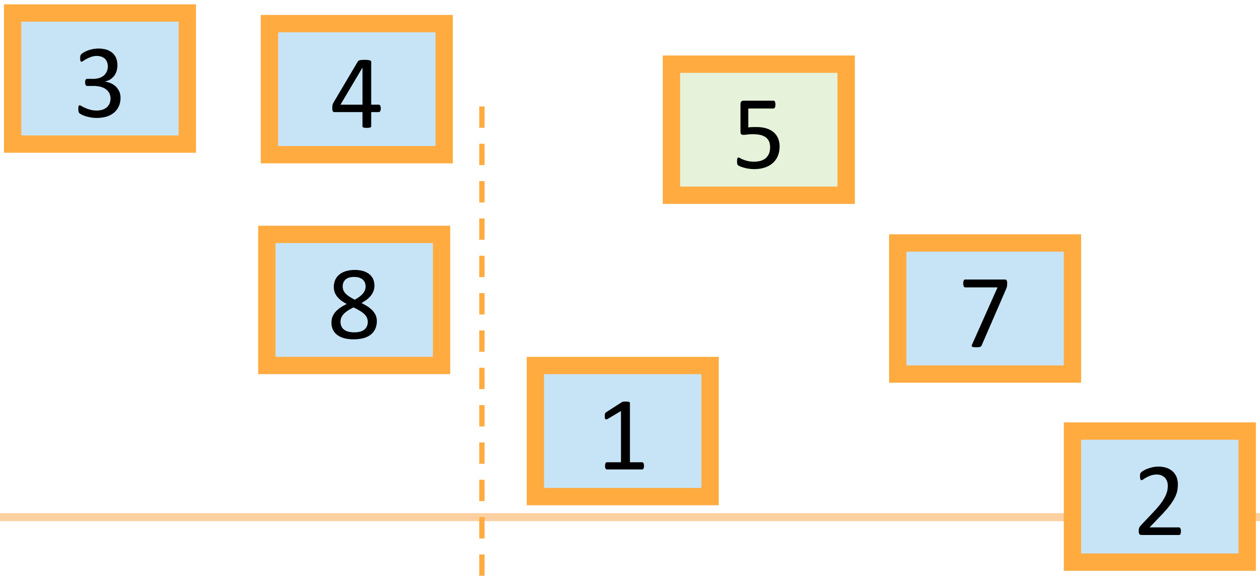


Q: Is this the only binary search tree I could possibly build with these values?

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

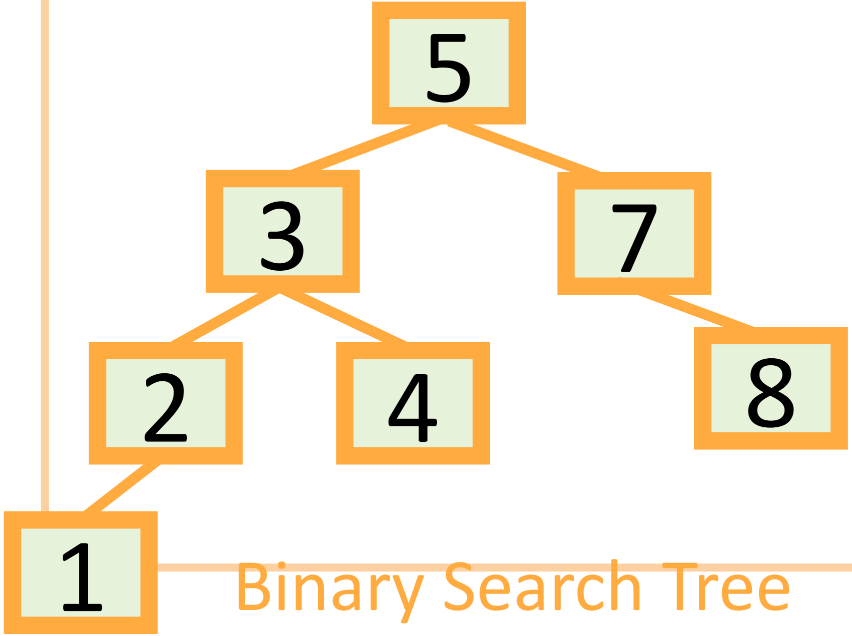
Aside: this should look familiar

kinda like QuickSort

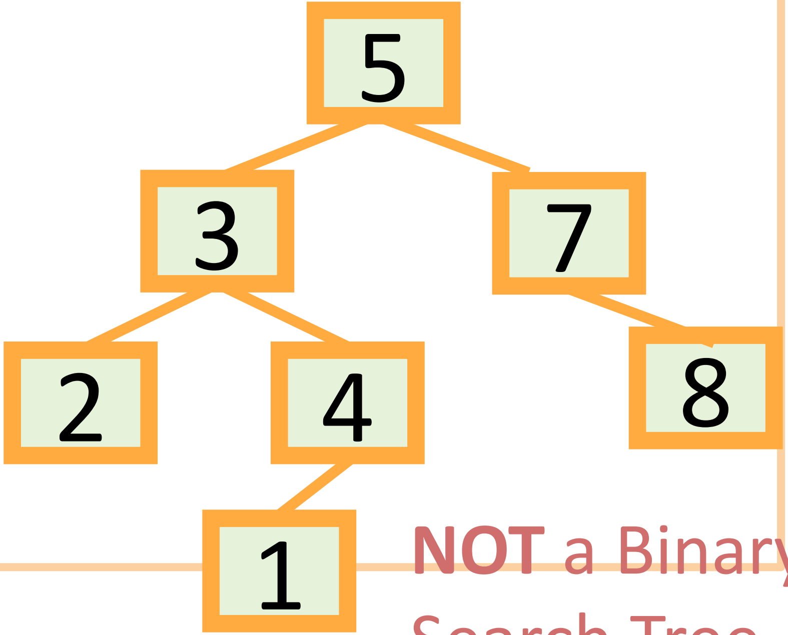


Are these Binary Search Trees? Yes or No?

- A BST is 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.



Binary Search Tree



NOT a Binary Search Tree

Are these Binary Search Trees? Yes or

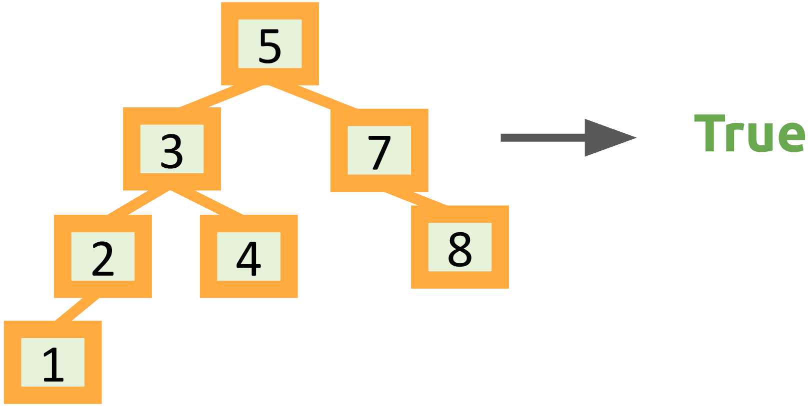
- A BST is a binary tree so that:
  - Every LEFT descendant of
  - Every RIGHT des

Can we do this programmatically?

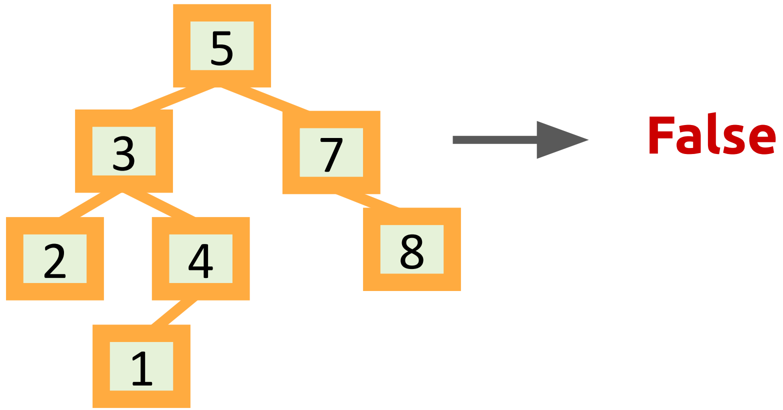
Binary Search Tree

NOT a Binary Search Tree

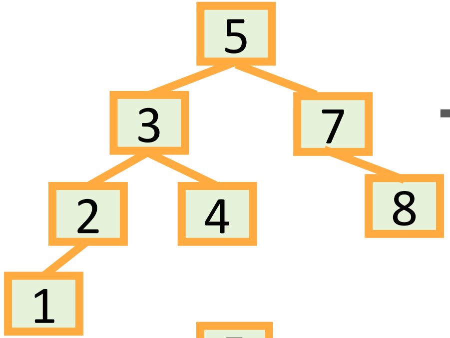
Given a root node, determine if it's a BST and return True/False



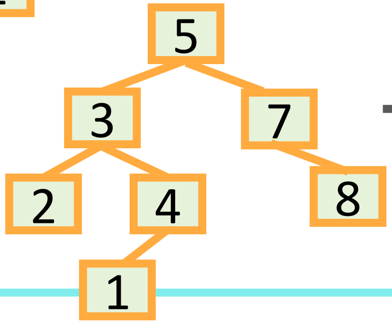
Given a root node, determine if it's a BST and return True/False



Given a root node, determine if it's a BST and return True/False



True



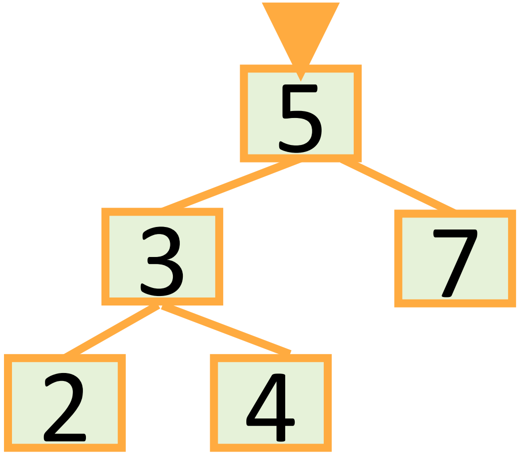
False

Let’s code it!!!

Aside: In-Order Traversal of BSTs

- Output all the elements in sorted order!

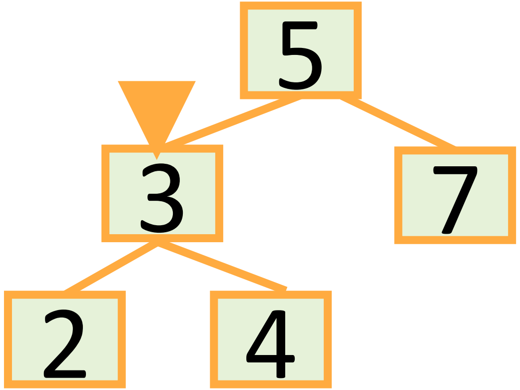
- inOrderTraversal(x):
  - if x!= NIL:
    - inOrderTraversal(x.left)
    - print(x.value)
    - inOrderTraversal(x.right)



Aside: In-Order Traversal of BSTs

- Output all the elements in sorted order!

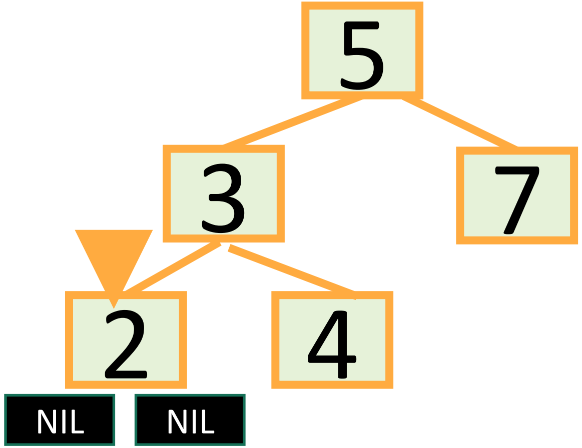
- inOrderTraversal(x):
  - if x!= NIL:
    - inOrderTraversal(x.left)
    - print(x.value)
    - inOrderTraversal(x.right)



Aside: In-Order Traversal of BSTs

- Output all the elements in sorted order!

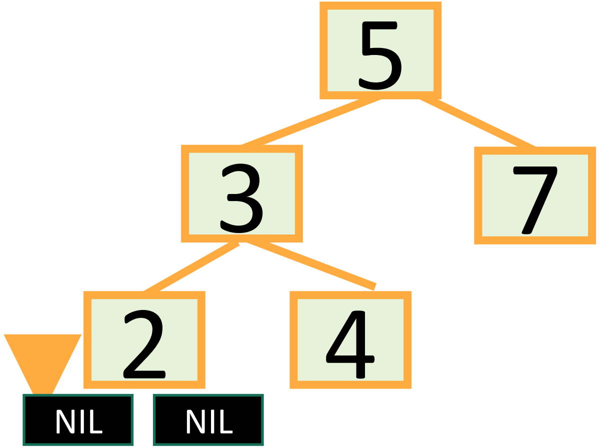
- inOrderTraversal(x):
  - if x!= NIL:
    - inOrderTraversal(x.left)
    - print(x.value)
    - inOrderTraversal(x.right)



Aside: In-Order Traversal of BSTs

- Output all the elements in sorted order!

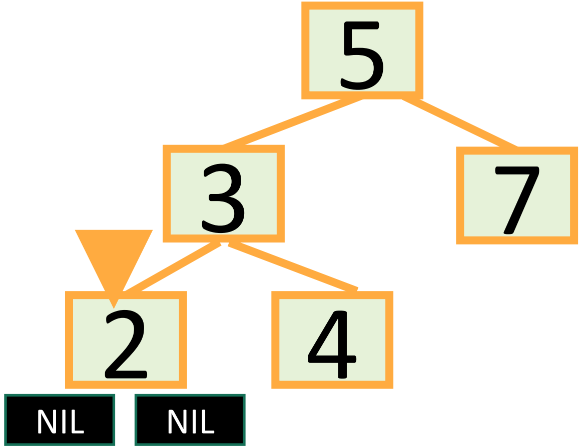
- inOrderTraversal(x):
  - if x!= NIL:
    - inOrderTraversal(x.left)
    - print(x.value)
    - inOrderTraversal(x.right)



Aside: In-Order Traversal of BSTs

- Output all the elements in sorted order!

- inOrderTraversal(x):
  - if x!= NIL:
    - inOrderTraversal(x.left)
    - print(x.value)
    - inOrderTraversal(x.right)

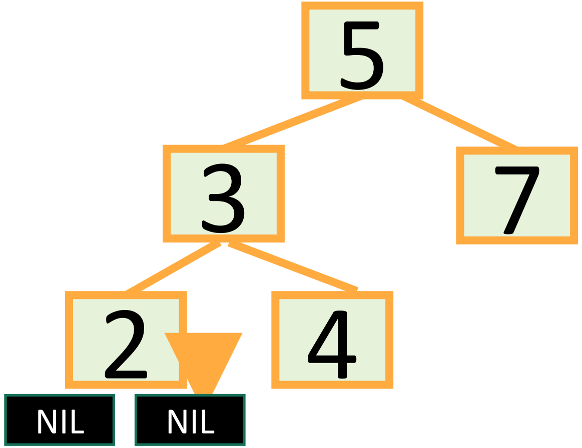


2

Aside: In-Order Traversal of BSTs

- Output all the elements in sorted order!

- inOrderTraversal(x):
  - if x!= NIL:
    - inOrderTraversal(x.left)
    - print(x.value)
    - inOrderTraversal(x.right)

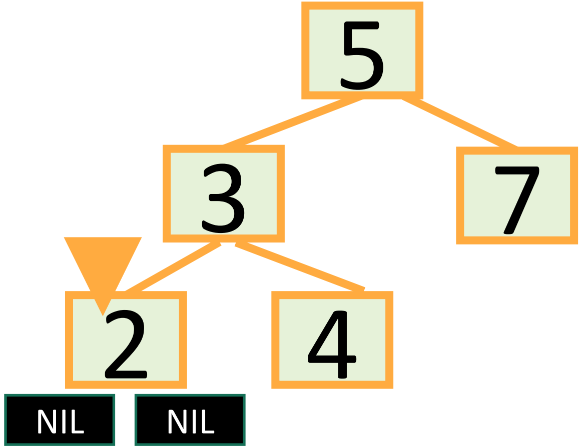


2

Aside: In-Order Traversal of BSTs

- Output all the elements in sorted order!

- inOrderTraversal(x):
  - if x!= NIL:
    - inOrderTraversal(x.left)
    - print(x.value)
    - inOrderTraversal(x.right)

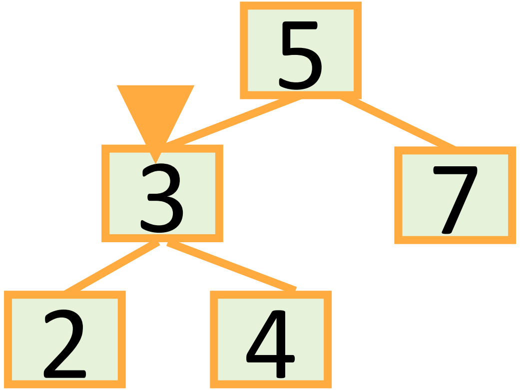


2

Aside: In-Order Traversal of BSTs

- Output all the elements in sorted order!

- inOrderTraversal(x):
  - if x!= NIL:
    - inOrderTraversal(x.left)
    - print(x.value)
    - inOrderTraversal(x.right)

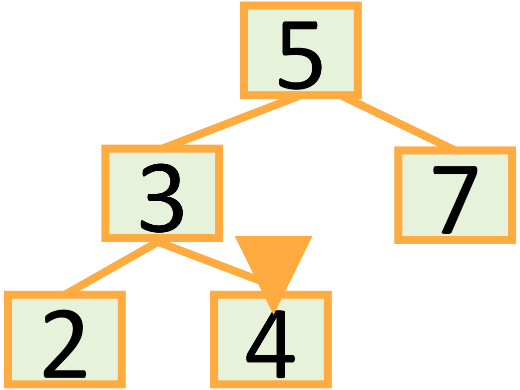


2 3

Aside: In-Order Traversal of BSTs

- Output all the elements in sorted order!

- inOrderTraversal(x):
  - if x!= NIL:
    - inOrderTraversal(x.left)
    - print(x.value)
    - inOrderTraversal(x.right)

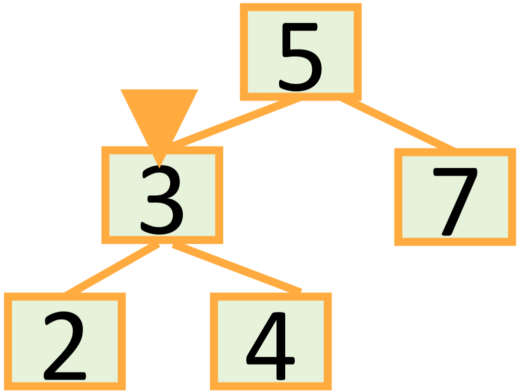


2 3 4

Aside: In-Order Traversal of BSTs

- Output all the elements in sorted order!

- inOrderTraversal(x):
  - if x!= NIL:
    - inOrderTraversal(x.left)
    - print(x.value)
    - inOrderTraversal(x.right)

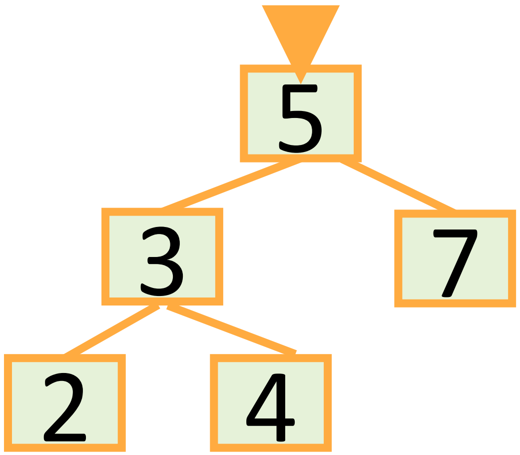


2 3 4

Aside: In-Order Traversal of BSTs

- Output all the elements in sorted order!

- inOrderTraversal(x):
  - if x!= NIL:
    - inOrderTraversal(x.left)
    - print(x.value)
    - inOrderTraversal(x.right)

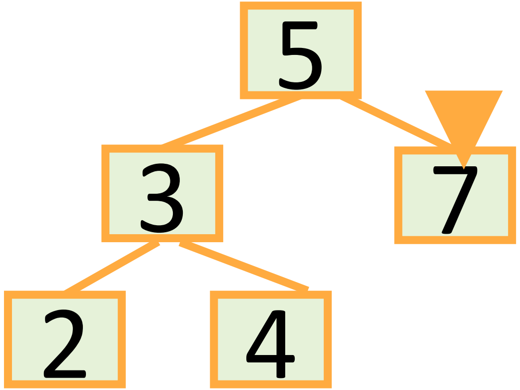


2 3 4 5

Aside: In-Order Traversal of BSTs

- Output all the elements in sorted order!

- inOrderTraversal(x):
  - if x!= NIL:
    - inOrderTraversal(x.left)
    - print(x.value)
    - inOrderTraversal(x.right)

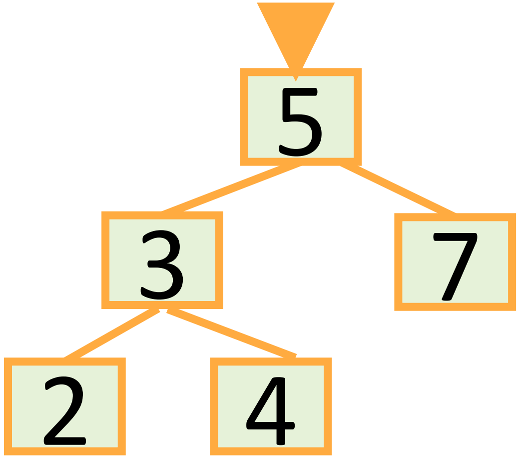


2 3 4 5 7

Aside: In-Order Traversal of BSTs

- Output all the elements in sorted order!

- inOrderTraversal(x):
  - if x!= NIL:
    - inOrderTraversal(x.left)
    - print(x.value)
    - inOrderTraversal(x.right)



- Runs in time O(n).

2 3 4 5 7 Sorted!

Big Questions!

- What are/Why binary search trees?
- Why does balance matter?
- How do we maintain balance?

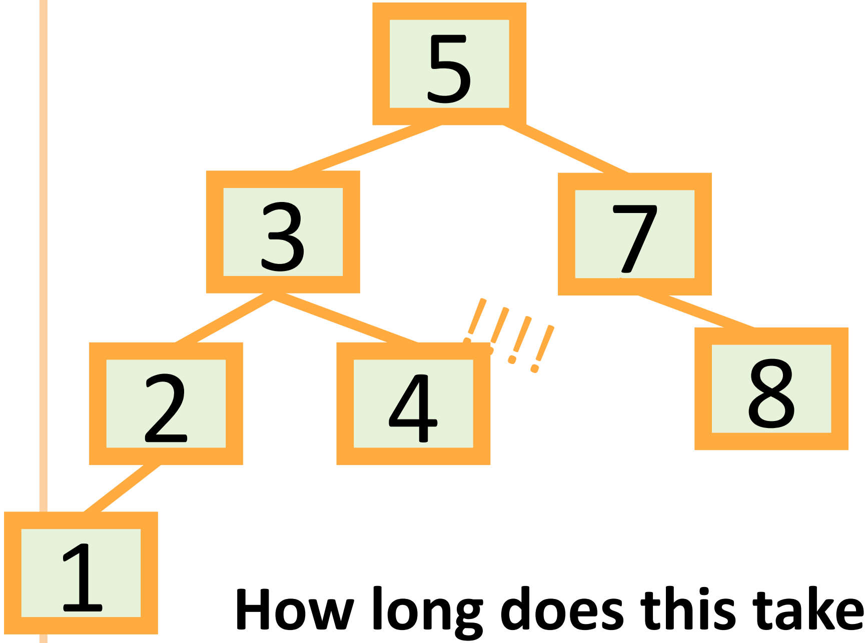
Back to the goal

Fast SEARCH/INSERT/DELETE

Can we do these?

	Sorted Arrays	Linked Lists	Binary Search 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)) 😃

SEARCH in a Binary Search Tree



EXAMPLE: Search for 4.

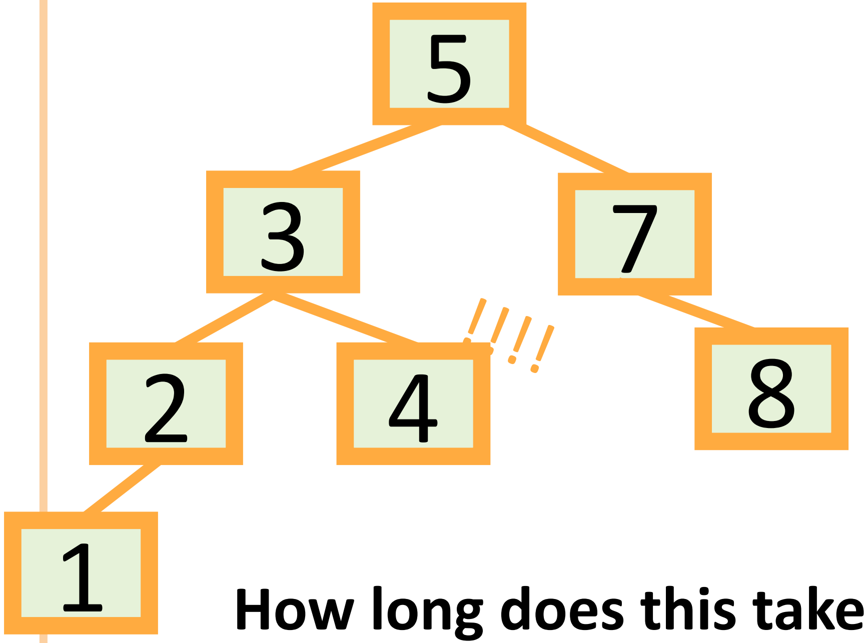
EXAMPLE: Search for 4.5

- 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(length of longest path) = O(height)

SEARCH in a Binary Search Tree



EXAMPLE: Search for 4.

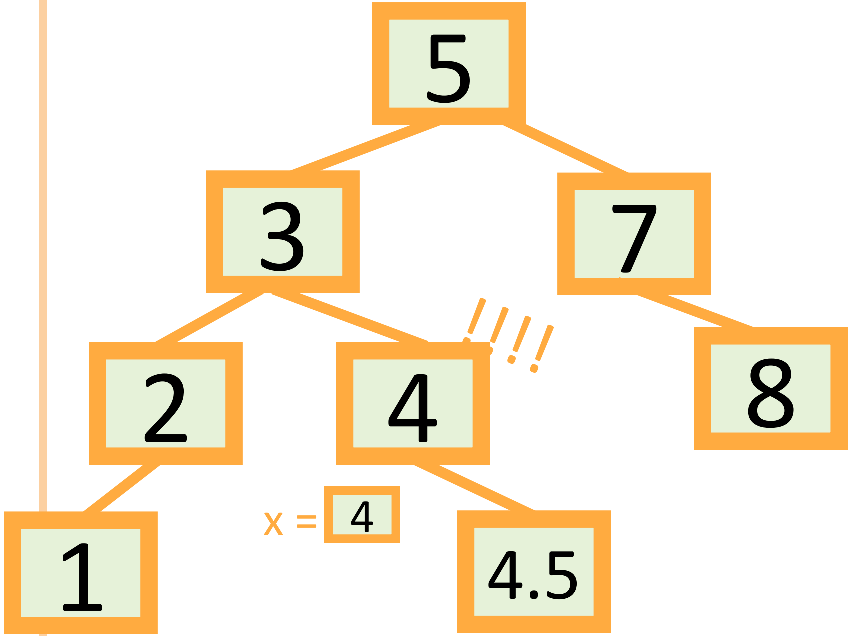
EXAMPLE: Search for 4.5

- if value > x.value:
  - Recurse on x.rightChild
- if value < x.value:
  - Recurse on x.leftChild
- if x.value == value:
  - **Return x**

How long does this take?

O(length of longest path) = O(height)

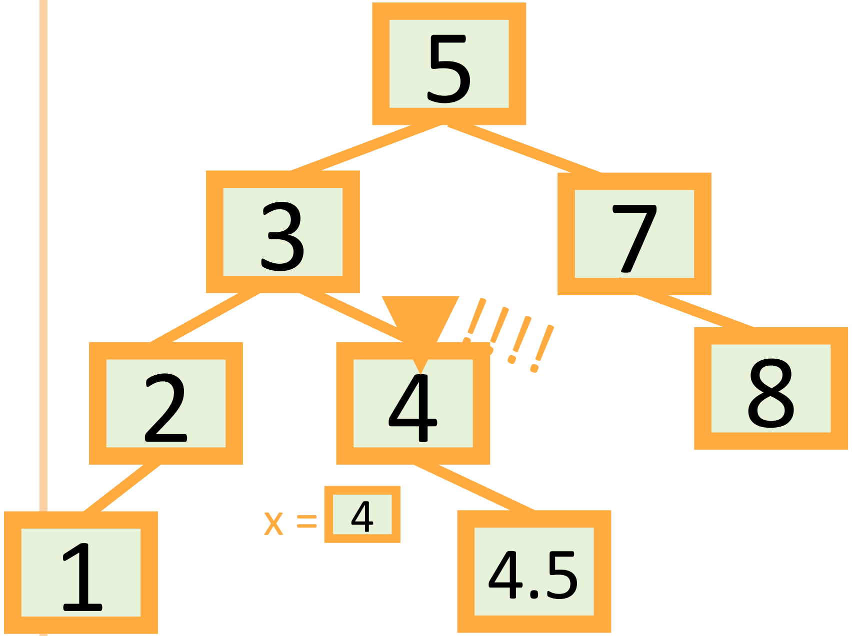
INSERT in a Binary Search Tree



EXAMPLE: Insert 4.5

- INSERT(value):
  - x = SEARCH(value)
  - **Insert** a new node with desired value at x…

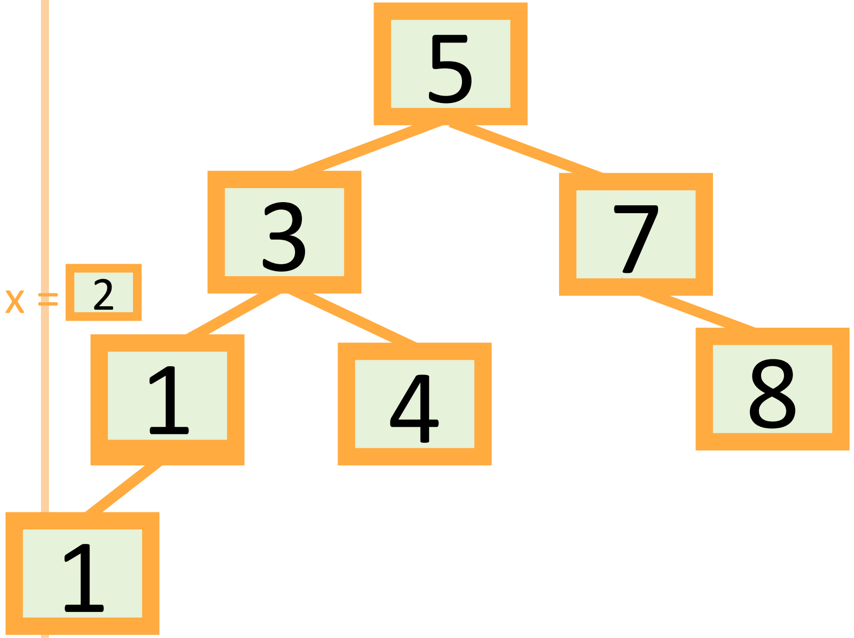
INSERT in a Binary Search Tree



EXAMPLE: Insert 4.5

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

DELETE in a Binary Search Tree

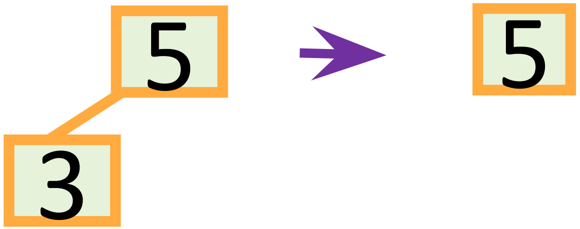


EXAMPLE: Delete 2

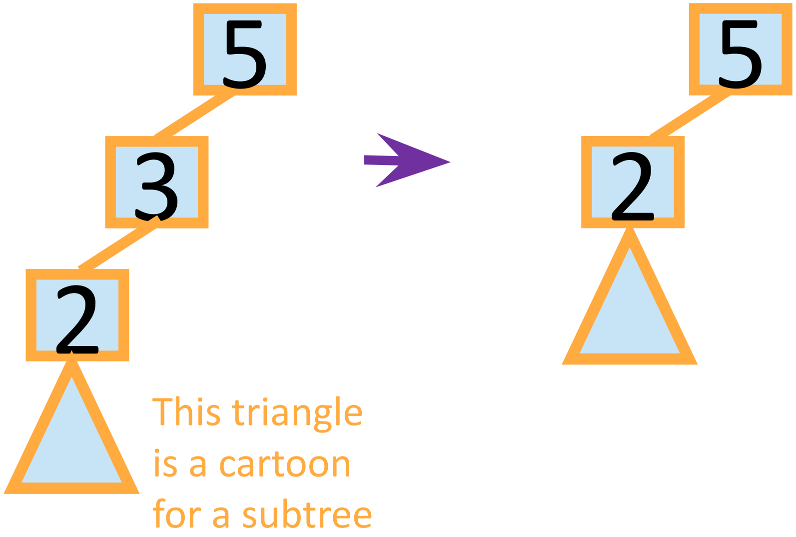
- DELETE(value):
  - x = SEARCH(value)
  - **if** x.value == value:
    - ….delete x….

DELETE in a Binary Search Tree

say we want to delete 3



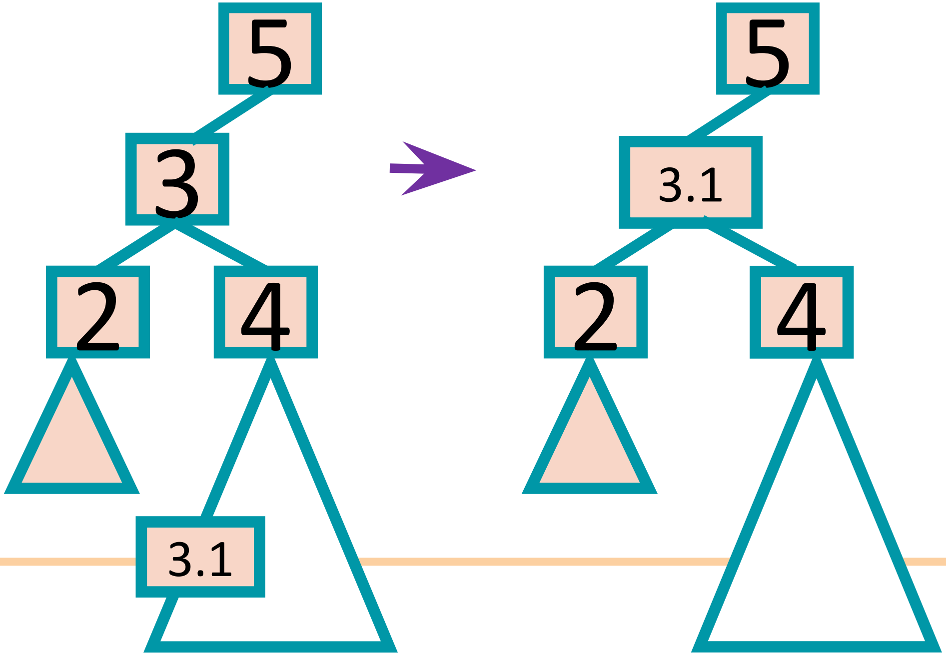
Case 1: if 3 is a leaf,
just delete it.



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

DELETE in a Binary Search Tree (cont.)

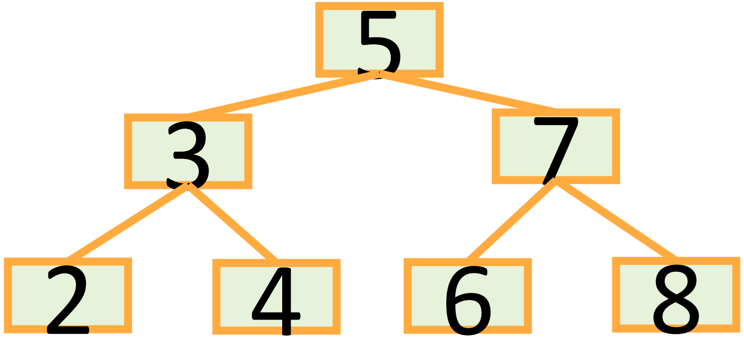
Case 3: if 3 has two children,
replace 3 with it's immediate successor.
(aka, next biggest thing after 3)



- 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.



Time = O(height of tree)

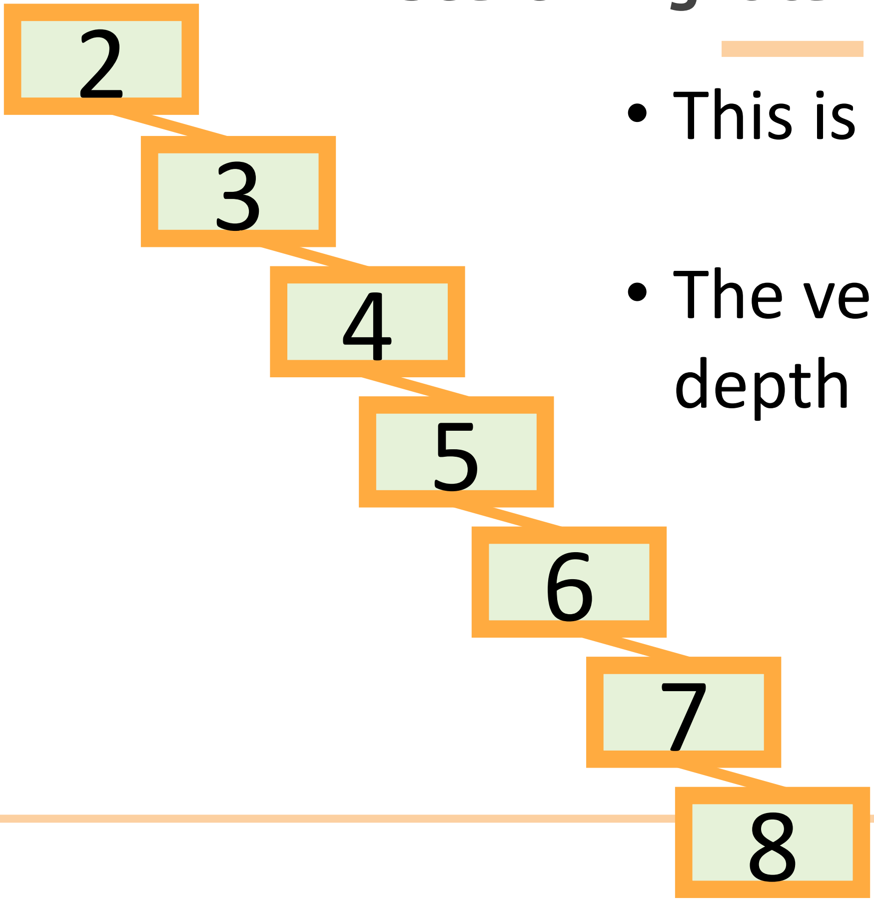
How long does search take?

1 minute think; 1 minute pair+share

Trees have depth O(log(n)). **Done!**

Wait a second…

Search might take time O(n).



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

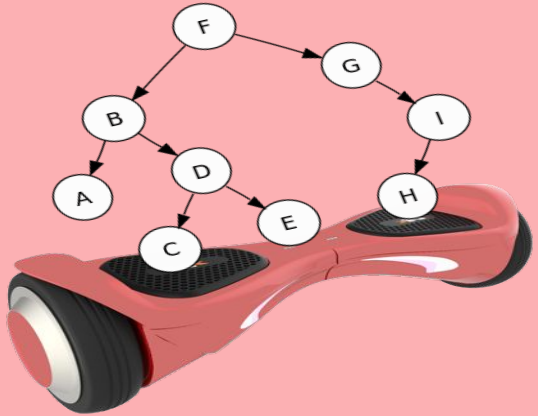
What to do?

- Goal: Fast SEARCH/INSERT/DELETE
- All these things take time O(height)
- And the height might be big!!! ☹️
- Idea!
  - Keep track of how deep the tree is getting.
  - If it gets too tall, re-do everything from scratch.
    - At least $\Omega(n)$ every so often….
- Turns out that's not a great idea, but we're onto something…

Big Questions!

- What are/Why binary search trees?
- Why does balance matter?
- How do we maintain balance?

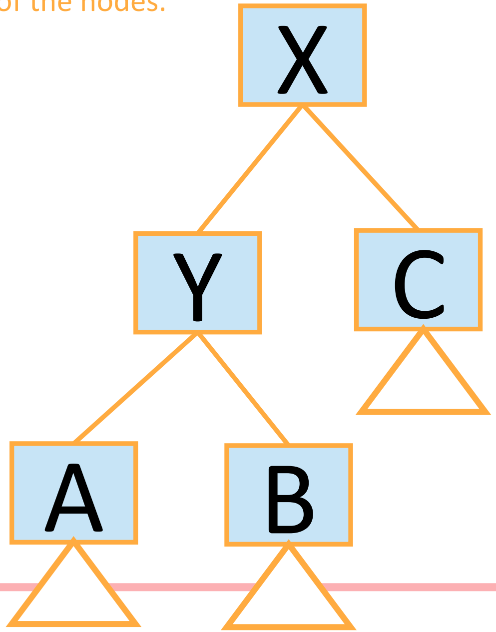
Self-Balancing Binary Search Trees



Idea! Rotations

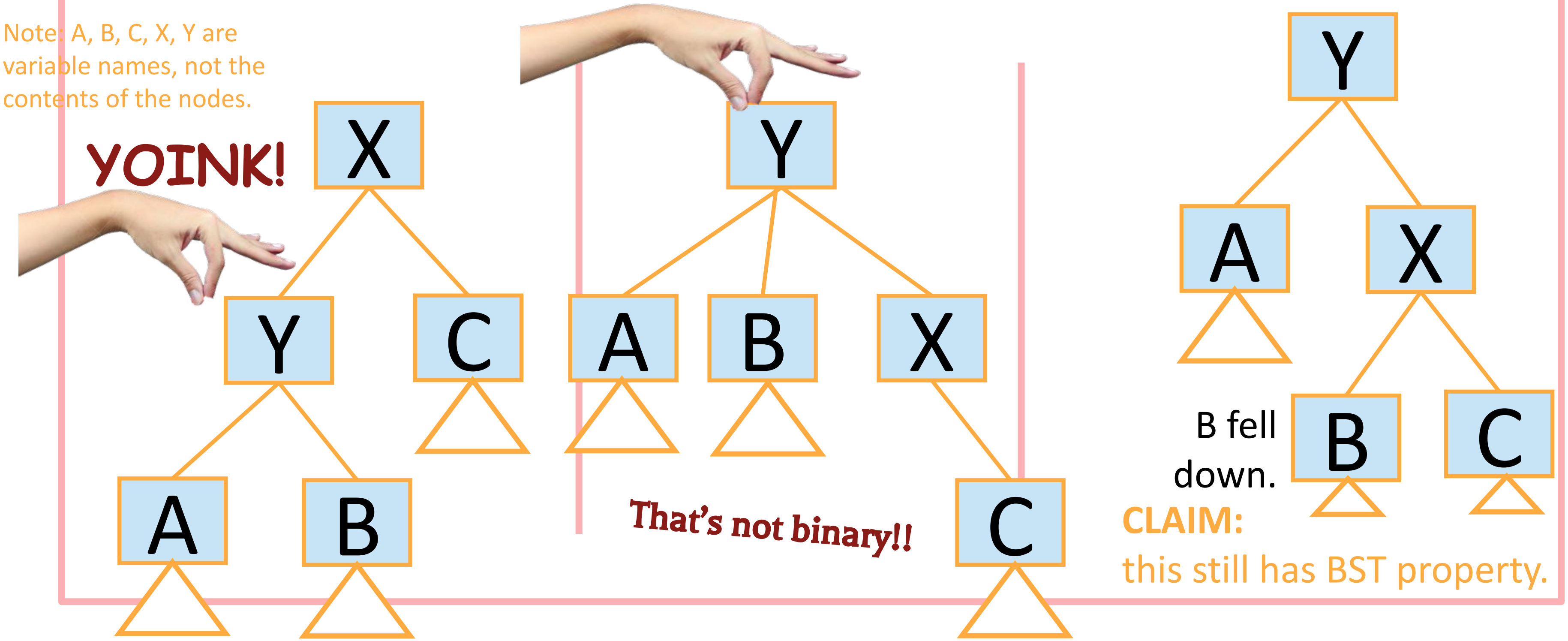
- Maintain Binary Search Tree (BST) property, while moving stuff around.

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

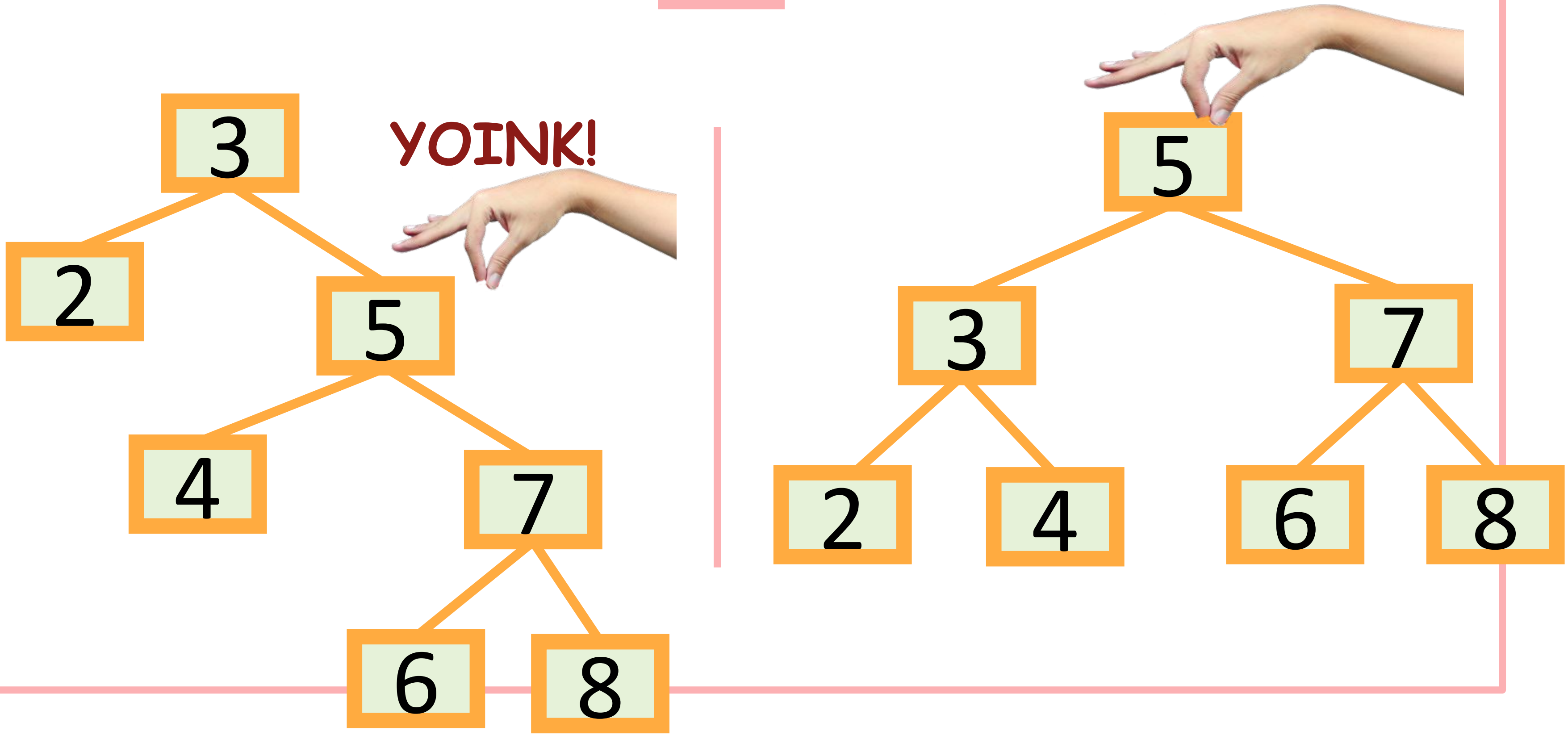


Idea! Rotations

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



This seems helpful



Next idea! Have some proxy for balance

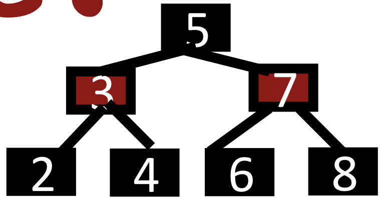
- Maintaining perfect balance is too hard.
- Instead, come up with some proxy for balance:
  - If the tree satisfies [SOME PROPERTY], then it's pretty balanced.
  - We can maintain [SOME PROPERTY] using rotations.

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

Red-Black Trees

- A Binary Search Tree that balances itself!
- No more time-consuming by-hand balancing!

Red-Black tree!

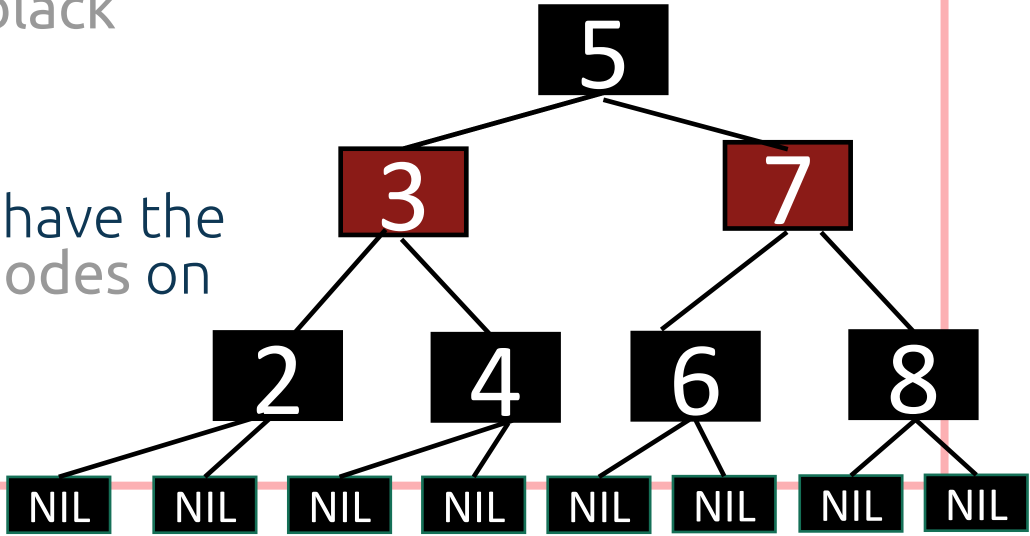


Maintain balance by stipulating that **black nodes** are balanced, and that there *aren't too many* **red nodes.**

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.

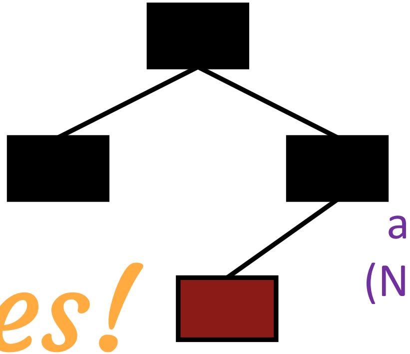
I'm not going to draw the NIL children in the future, but they are treated as black nodes.



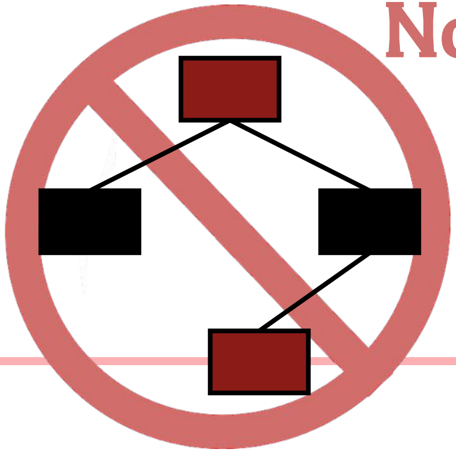
Examples(?)

- 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.

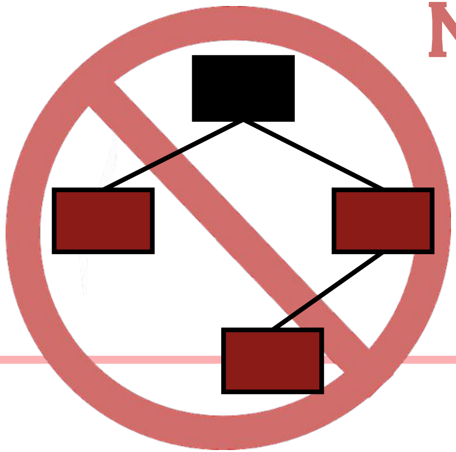
Which of these are red-black trees? (NIL nodes not drawn)



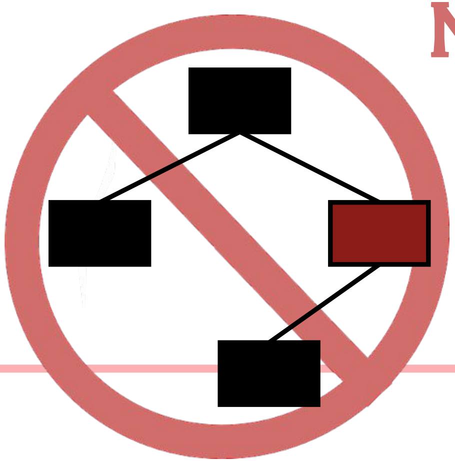
Yes!



No!



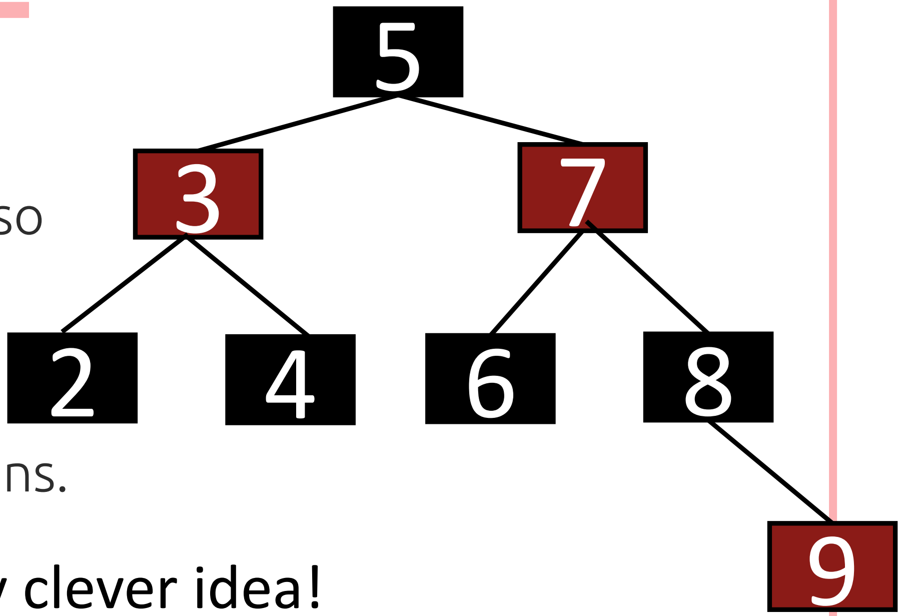
No!



No!

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!

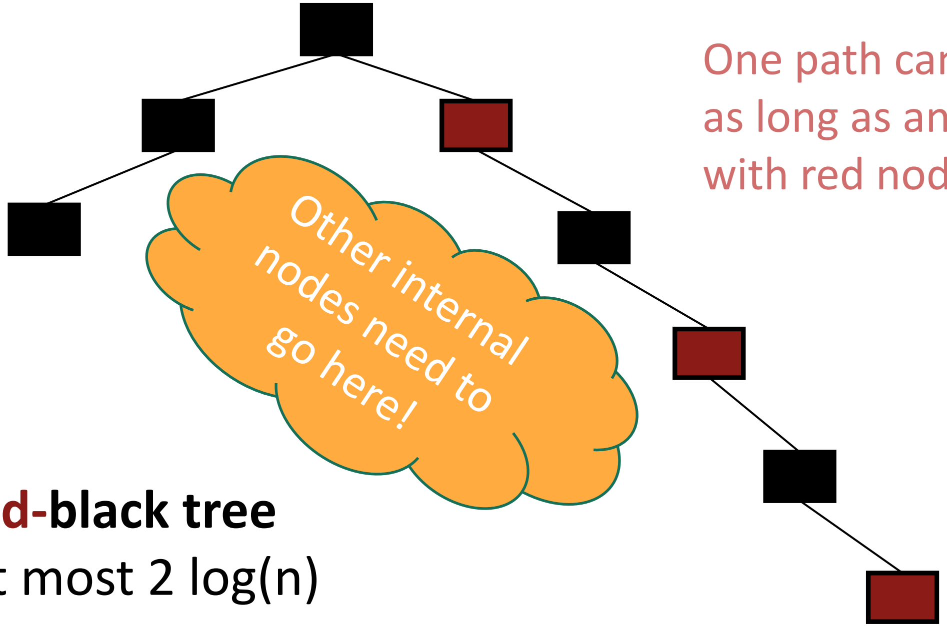
This **Red-Black** structure is a proxy for balance.

It's just a smidge 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.

Note, this is just a conjecture to build intuition! Rigorous proof on the next, skipped slide.

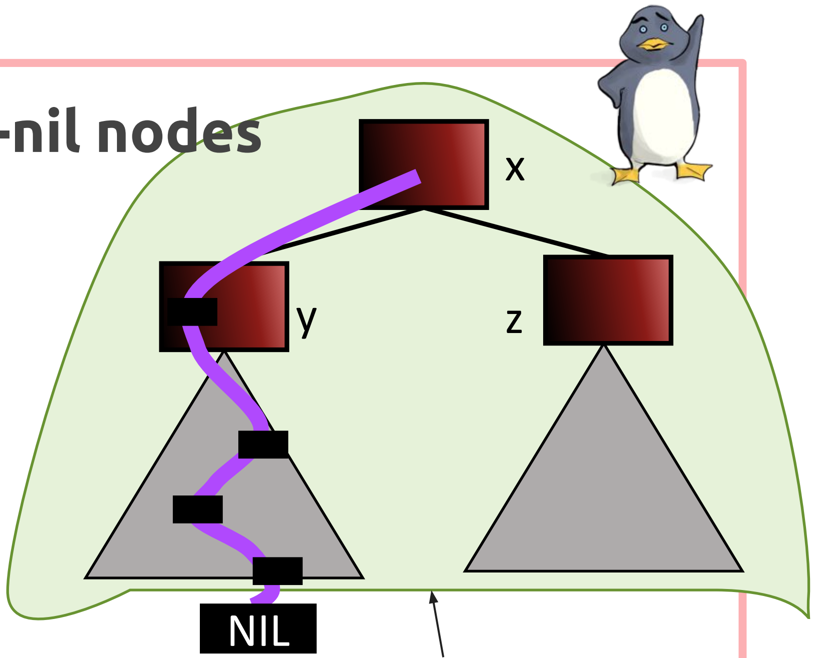


One path can be at most twice as long as another if we pad it with red nodes.

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

RBTree with N non-nil nodes

- 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).
- [You can proof by induction]



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

Then:

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

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

Rearranging:

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

This is great!

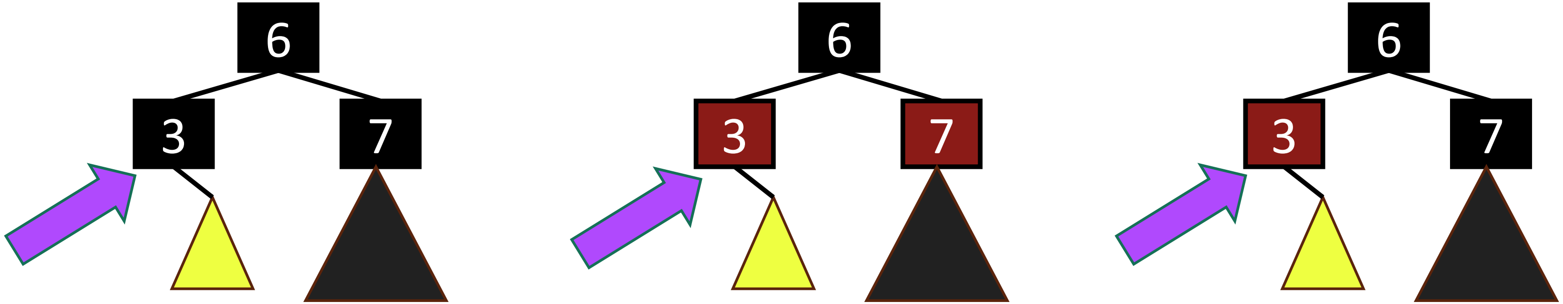
- SEARCH in an RBTree is immediately O(log(n)), since the depth of an RBTree is O(log(n)).

- 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.
  - That's why this is a good property!

INSERT/DELETE

- I expect we are out of time…
  - There are some slides which you can check out to see how to do INSERT/DELETE in RBTrees if you are curious.
  - See CLRS Ch 13. for even more details.
- You are **not responsible** for the details of INSERT/DELETE for RBTrees for this class.
  - You should know what the “proxy for balance” property is and why it ensures approximate balance.
  - You should know **that** this property can be efficiently maintained, but you do not need to know the details of how.

INSERT: Many cases

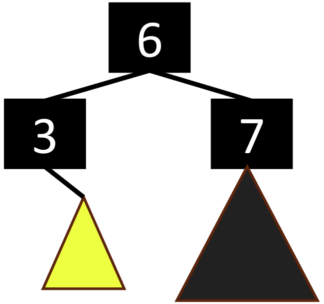


- Suppose we want to insert 0 **here**.

- 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.

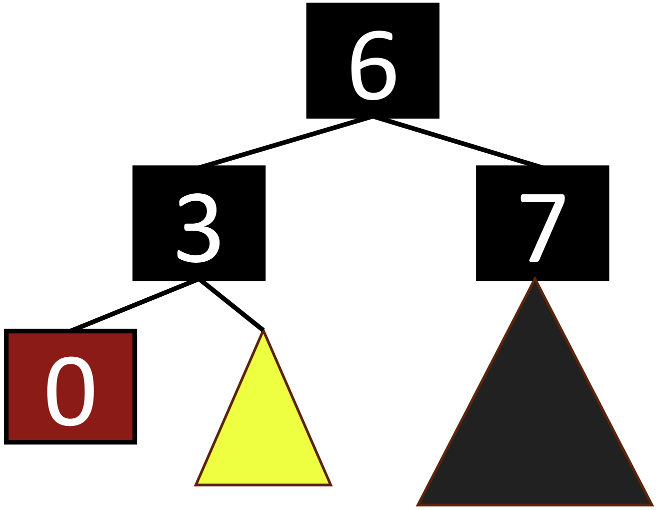
INSERT: Case 1

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

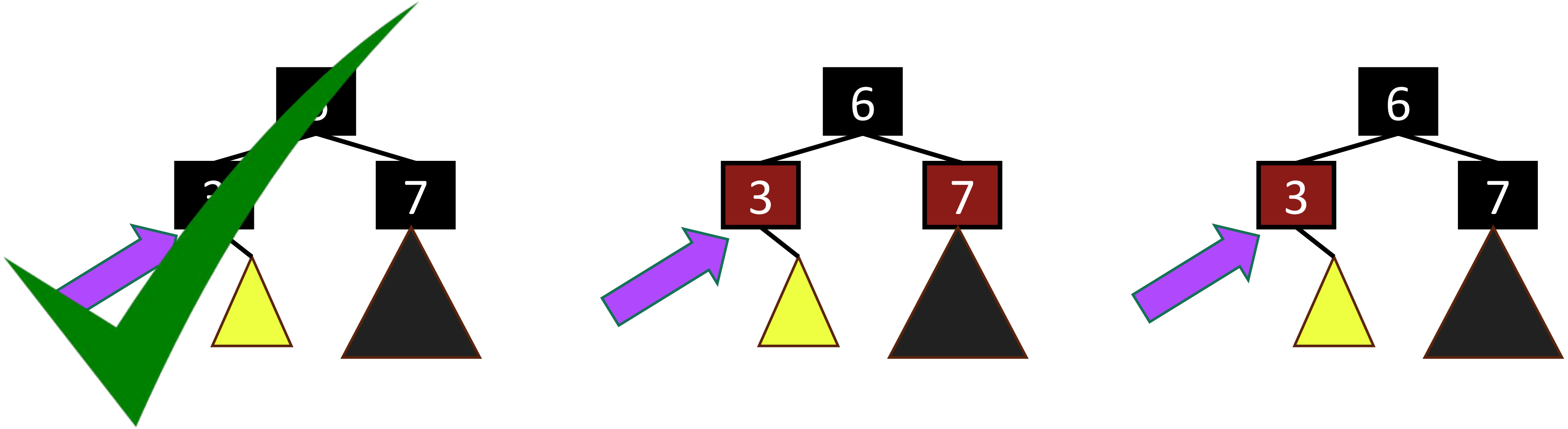


What if it looks like this?

Example: insert 0



INSERT: Many cases

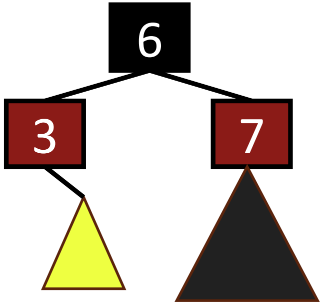


- Suppose we want to insert 0 **here**.

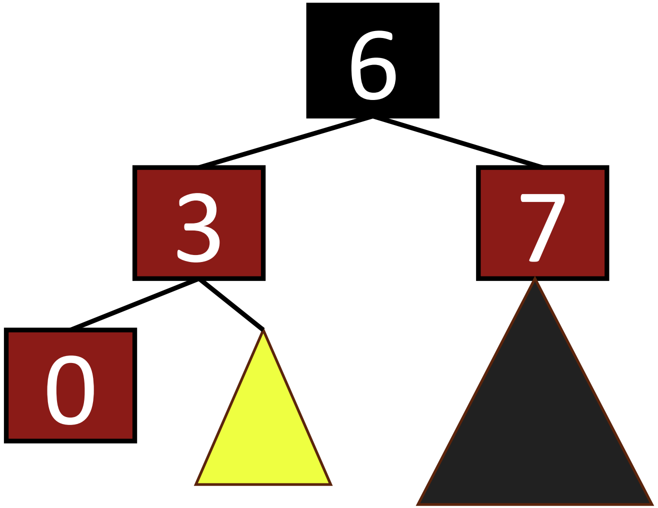
- There are 3 “important” cases for different colorings of the existing tree, and there are 9 more cases for all of the various symmetries of these 3 cases.

INSERT: Case 2

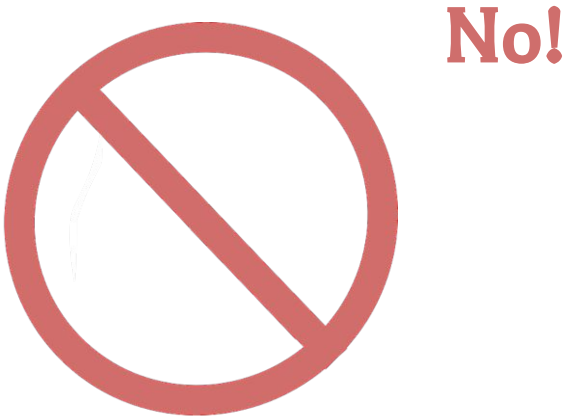
- Make a new **red node.**
- Insert it as you would normally.
- Fix things up if needed.



What if it looks like this?

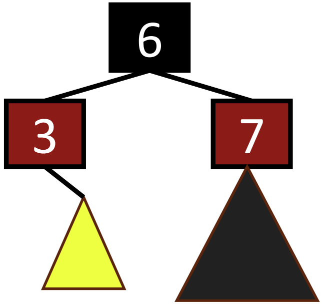


Example: insert 0

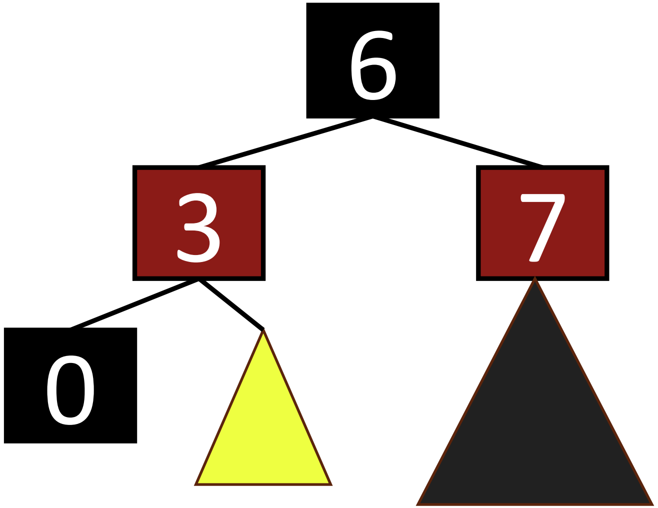


INSERT: Case 2

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



What if it looks like this?

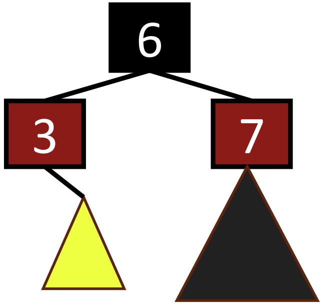


Example: insert 0

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

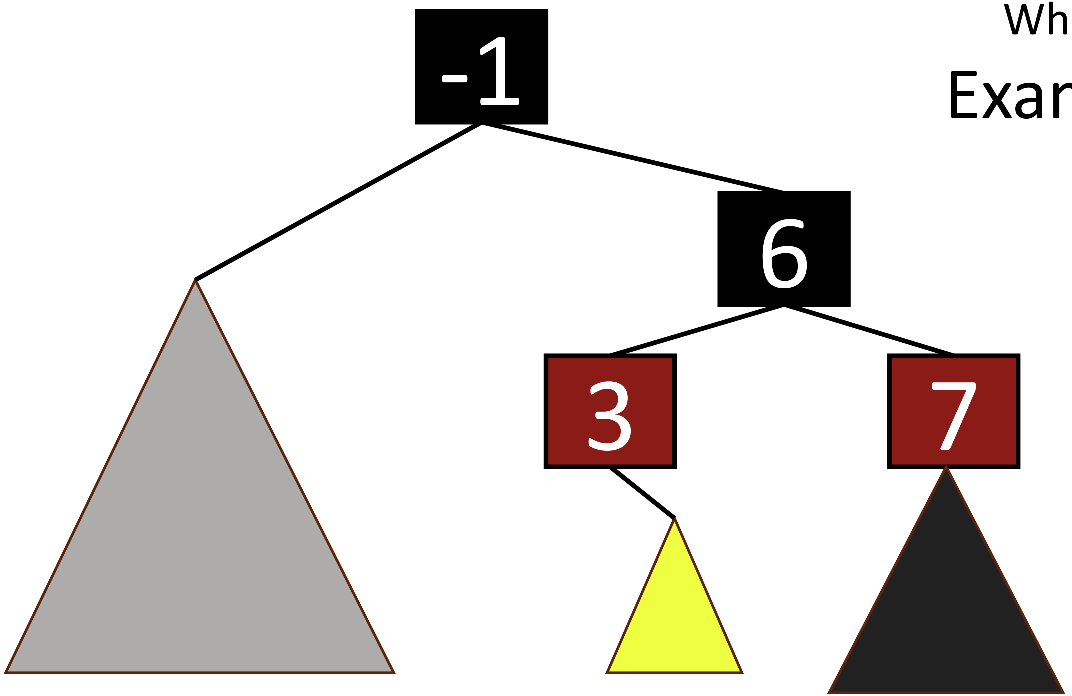
No!

We need a bit more context



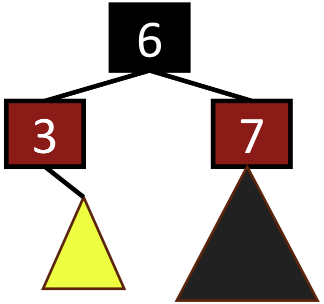
What if it looks like this?

Example: insert 0



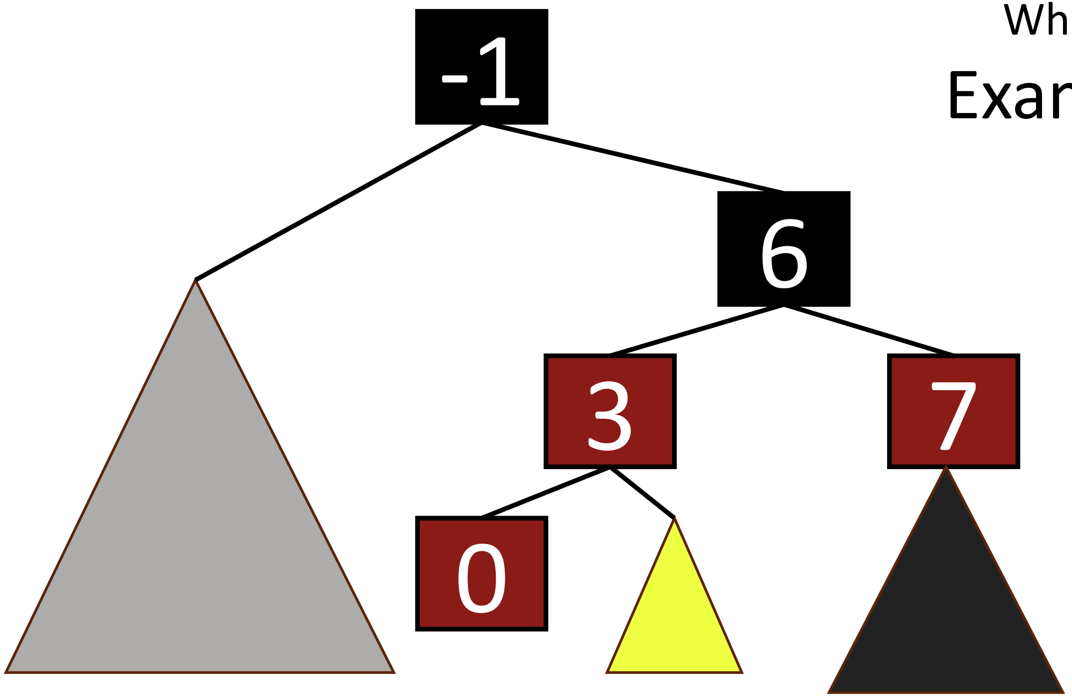
We need a bit more context

- Add 0 as a red node.



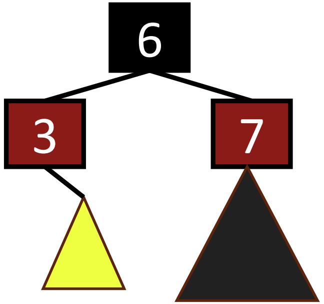
What if it looks like this?

Example: insert 0



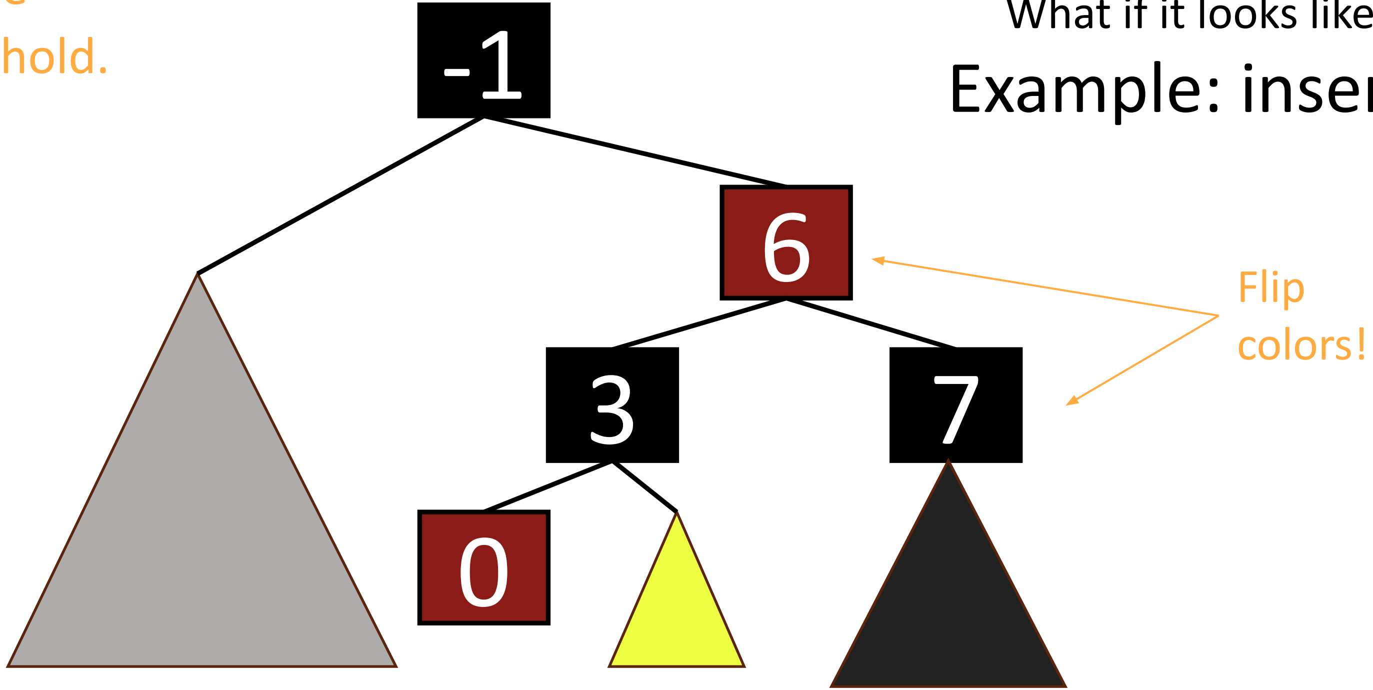
We need a bit more context

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

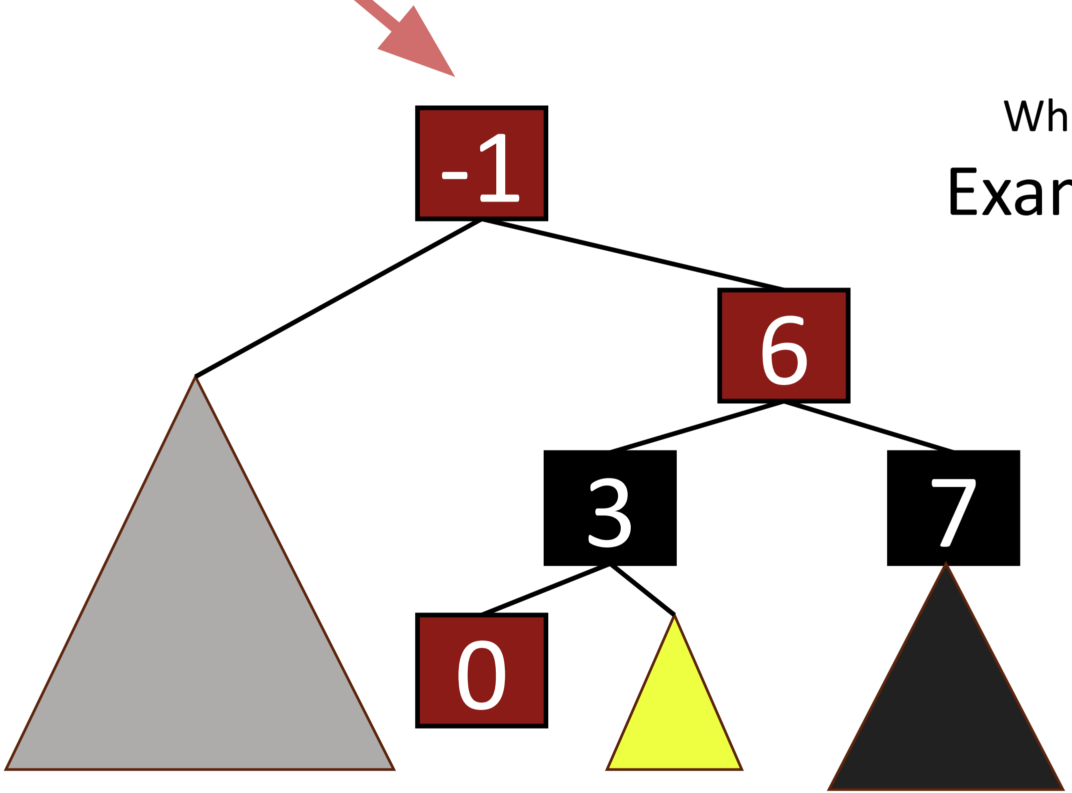


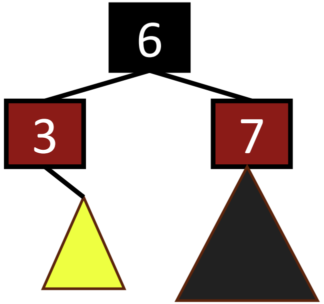
What if it looks like this?

Example: insert 0



But what if that was red?

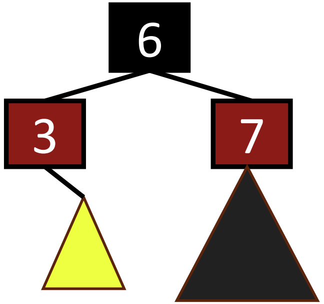




What if it looks like this?

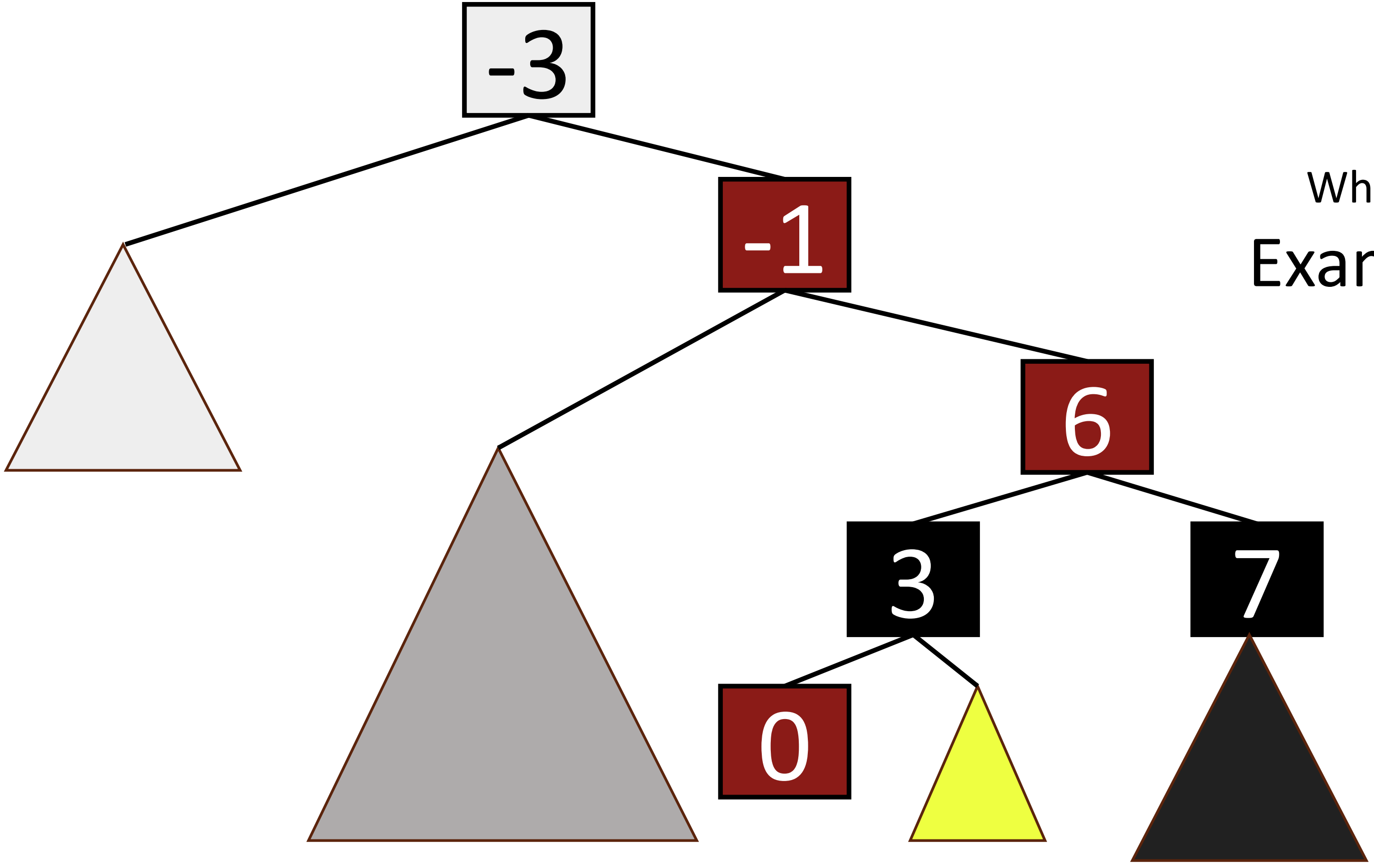
Example: insert 0

More context...

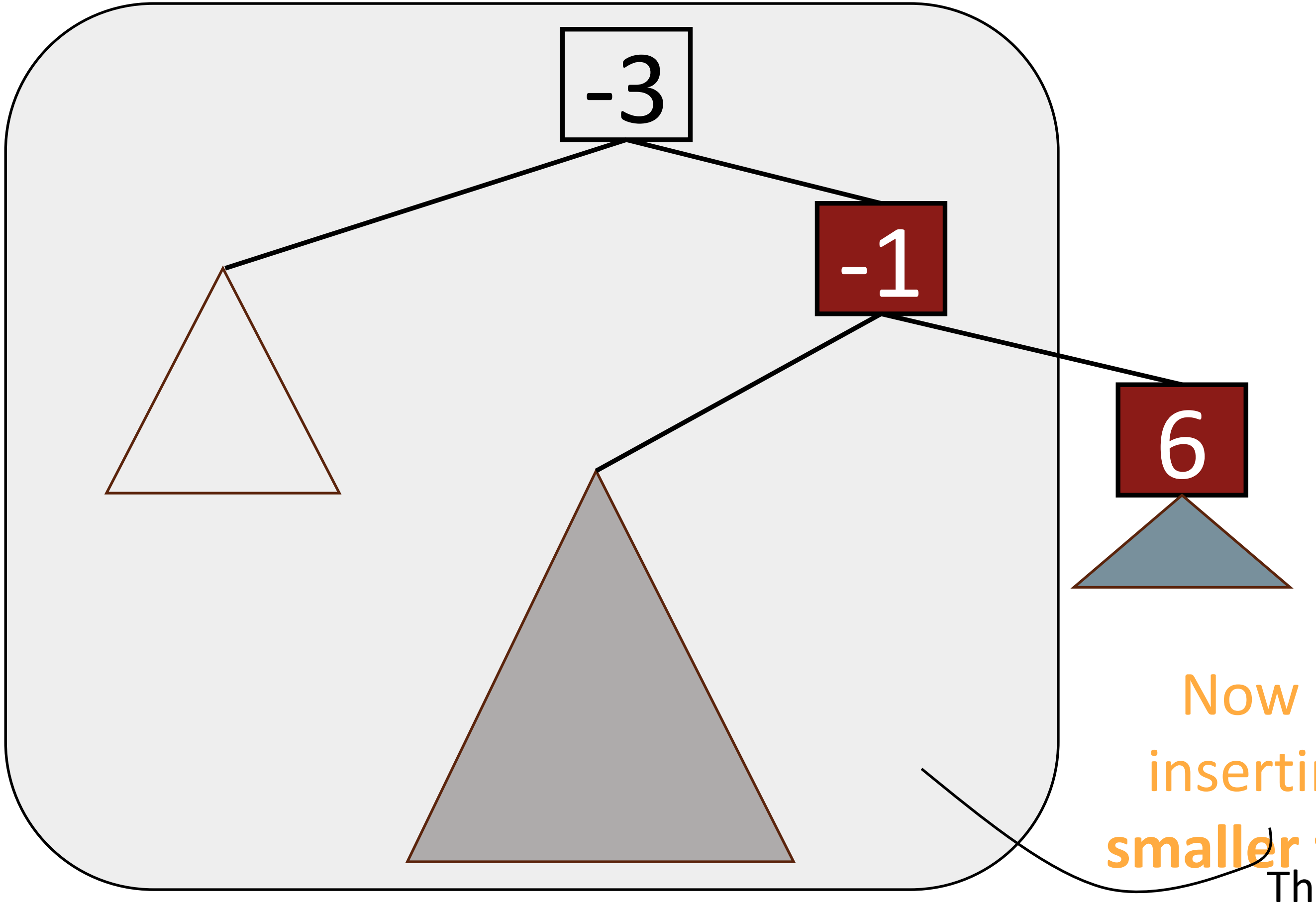


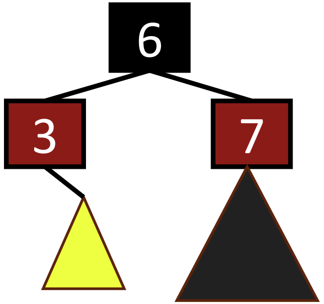
What if it looks like this?

Example: insert 0



More context...





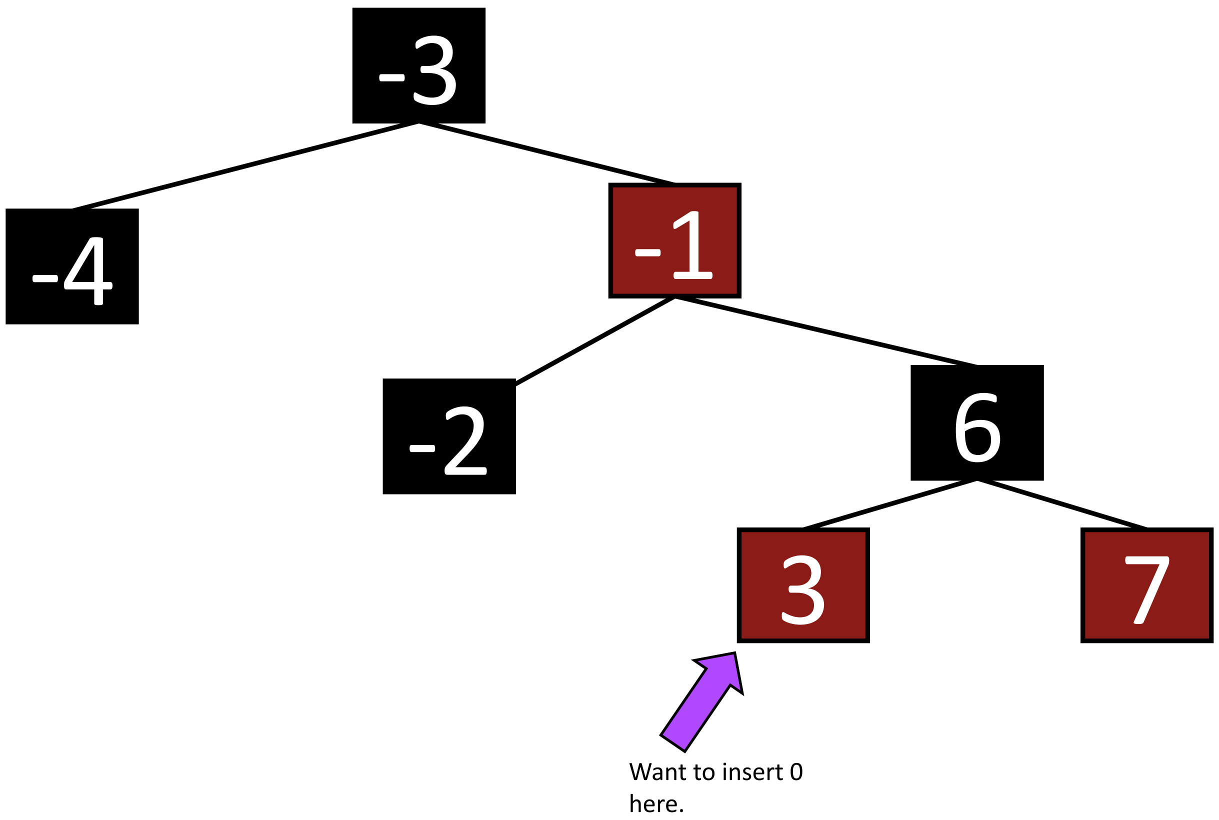
What if it looks like this?

Example: insert 0

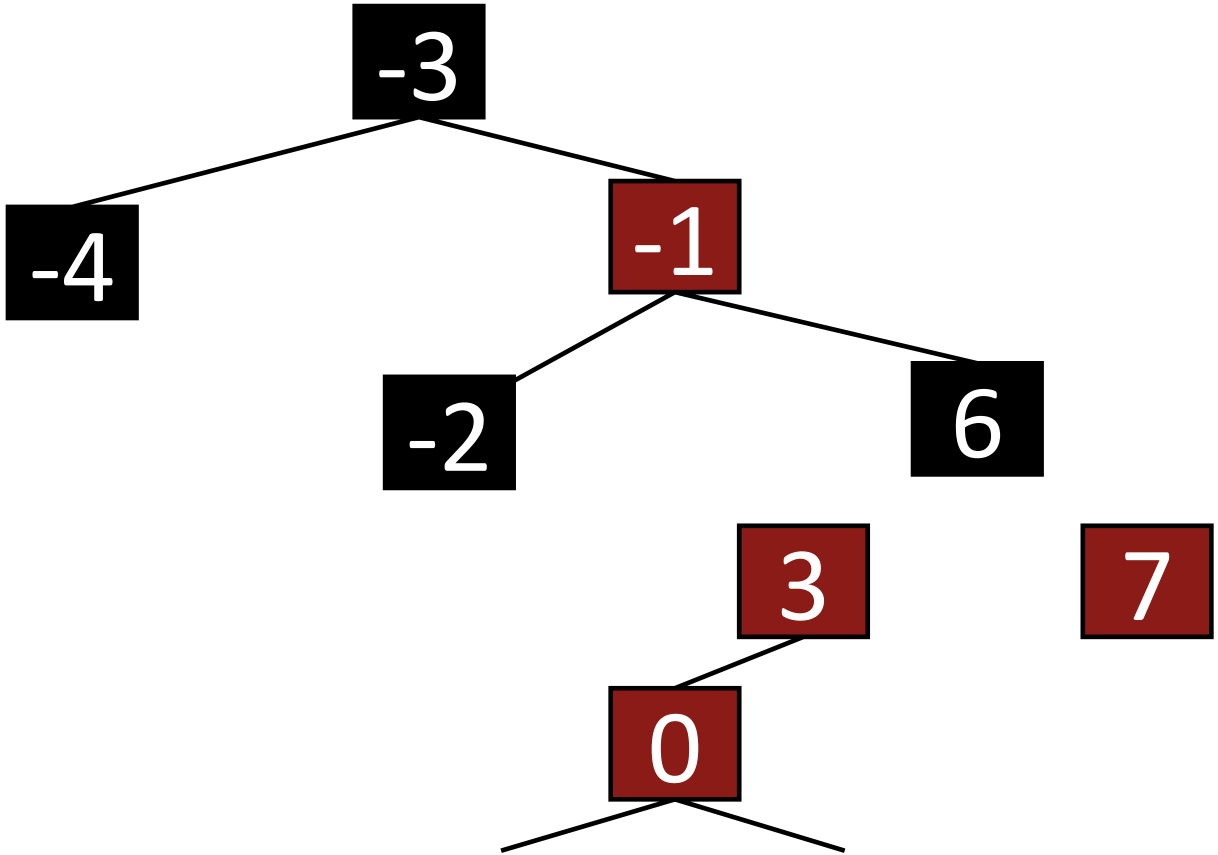
Now we're basically inserting 6 into some **smaller tree**. Recurse!

This one!

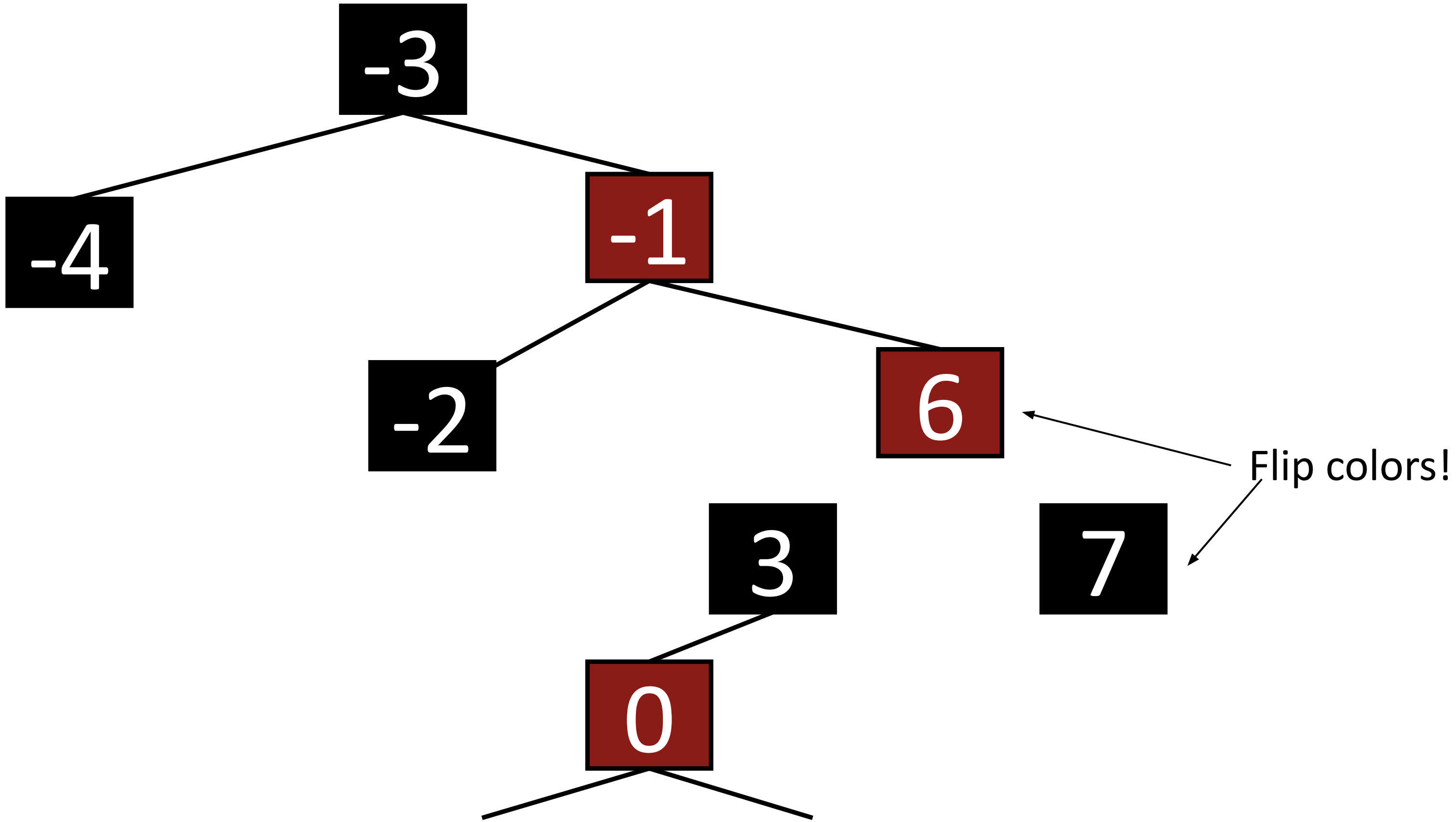
Example, part I



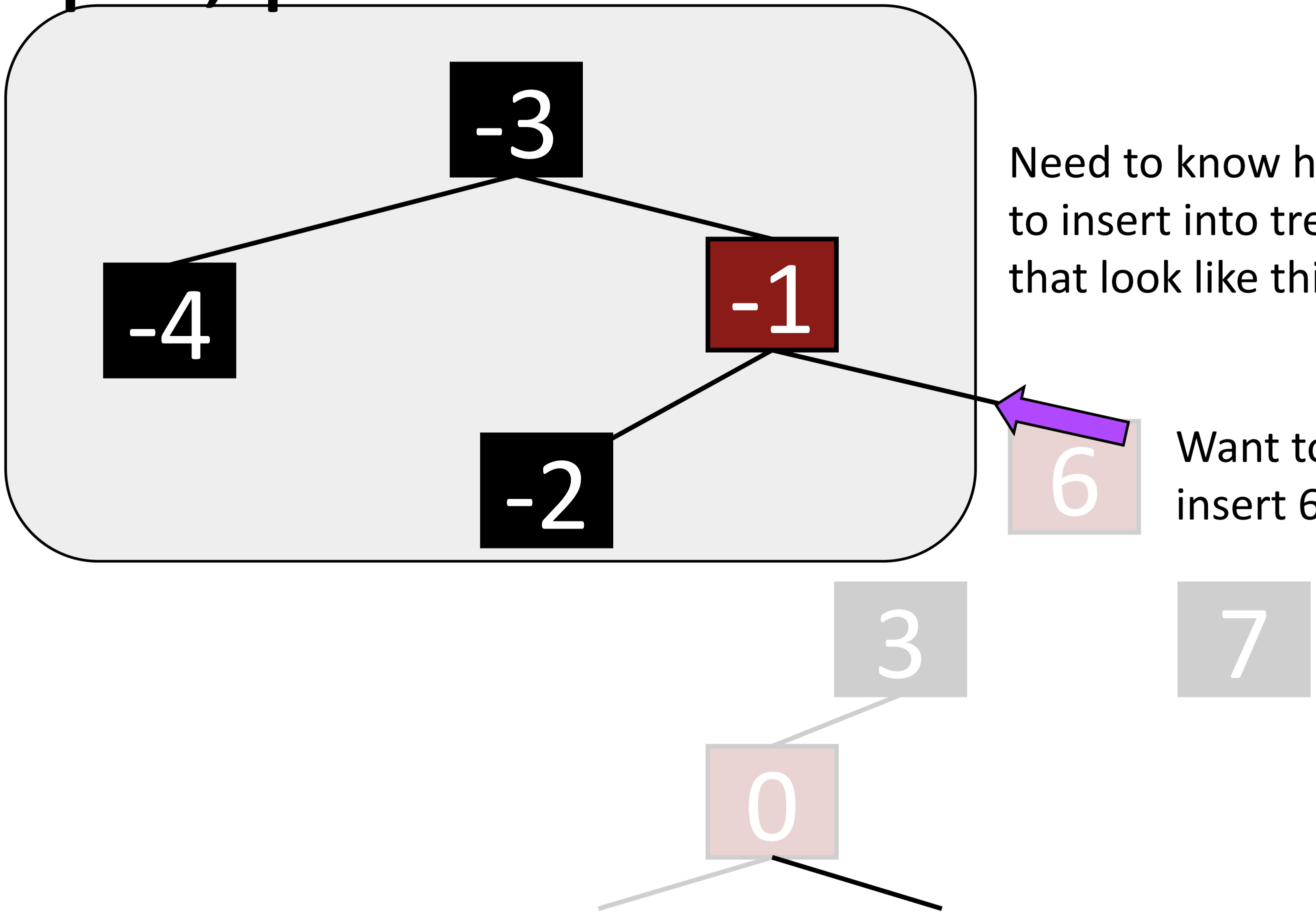
Example, part I



Example, part I



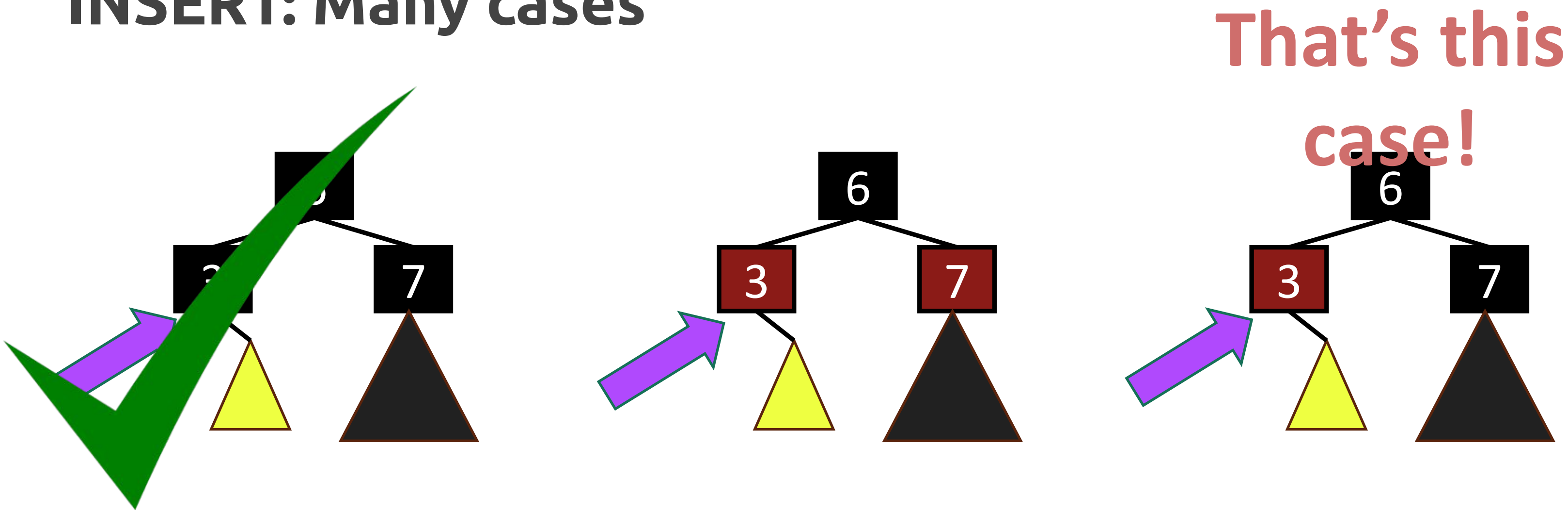
Example, part I



Need to know how to insert into trees that look like this…

Want to insert 6 here.

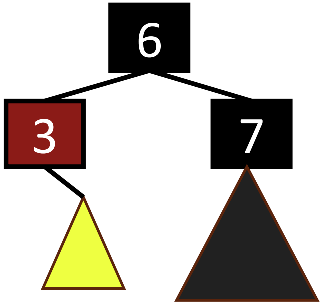
INSERT: Many cases



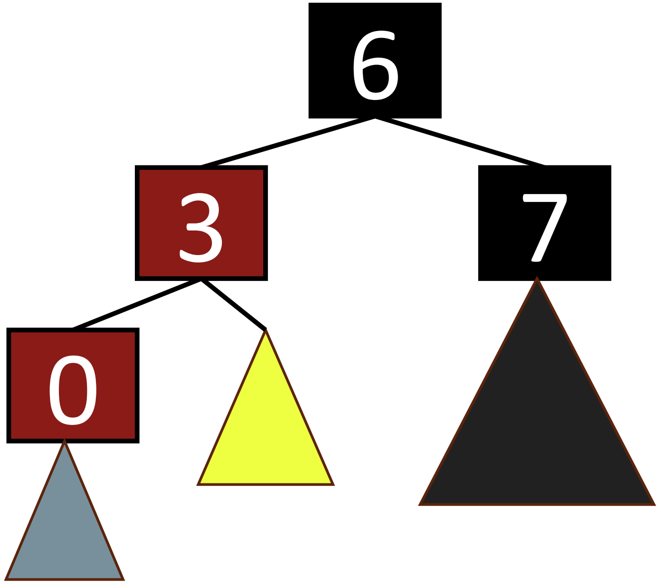
- Suppose we want to insert 0 **here**.
- There are 3 "important" cases for different colorings of the existing tree, and there are 9 more cases for all of the various symmetries of these 3 cases.

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?

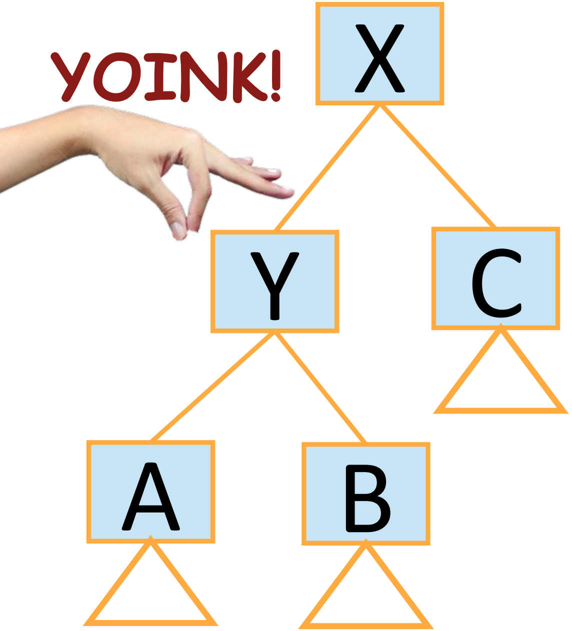


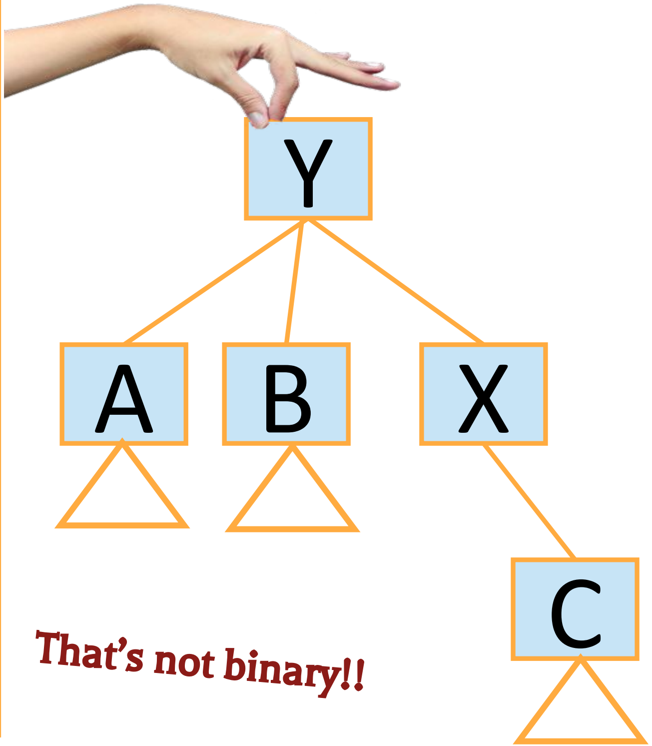
Example: Insert 0.

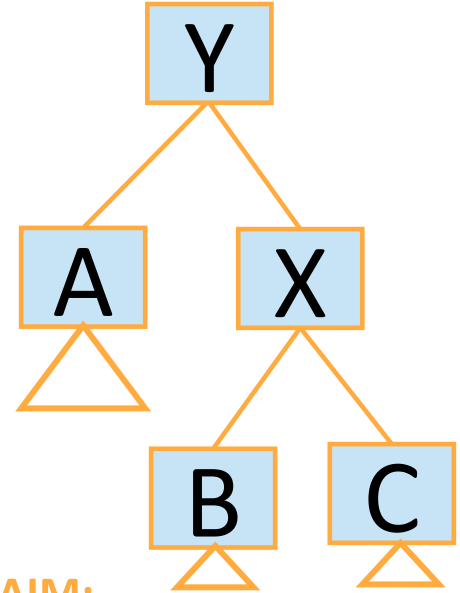
- Maybe with a subtree below it.

Recall Rotations

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



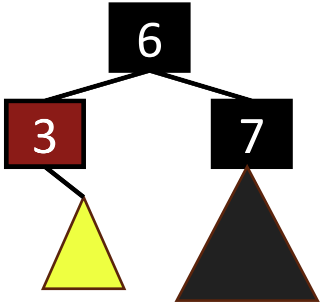




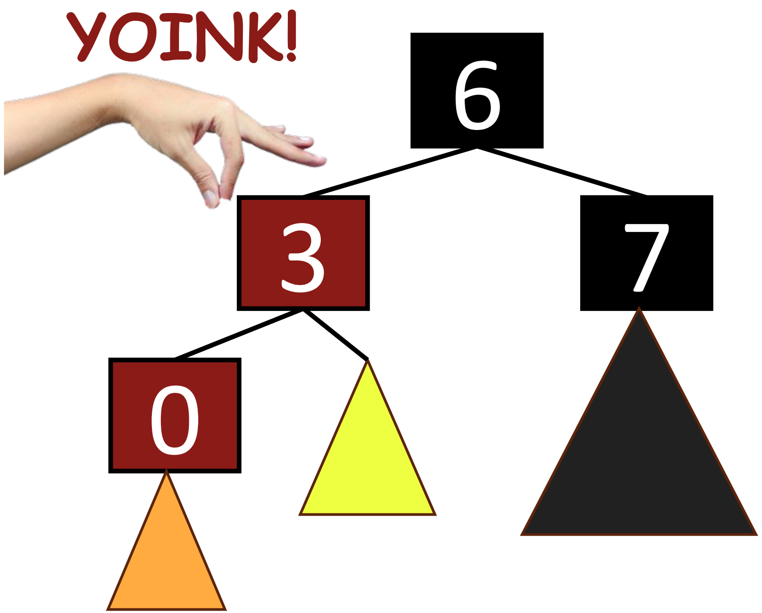
CLAIM:
this still has BST property.

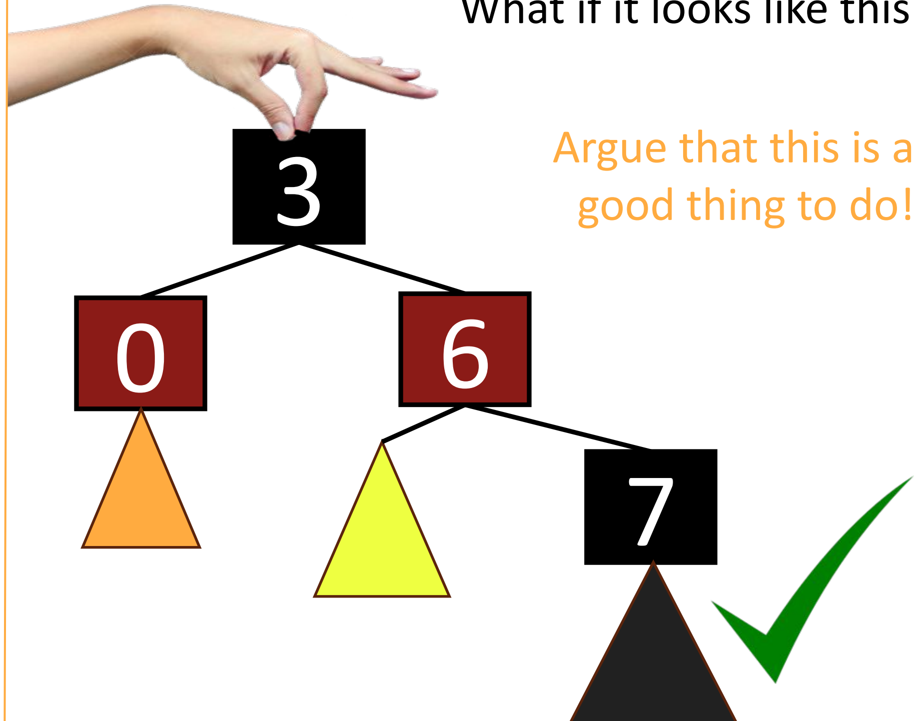
Inserting into a Red-Black Tree

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



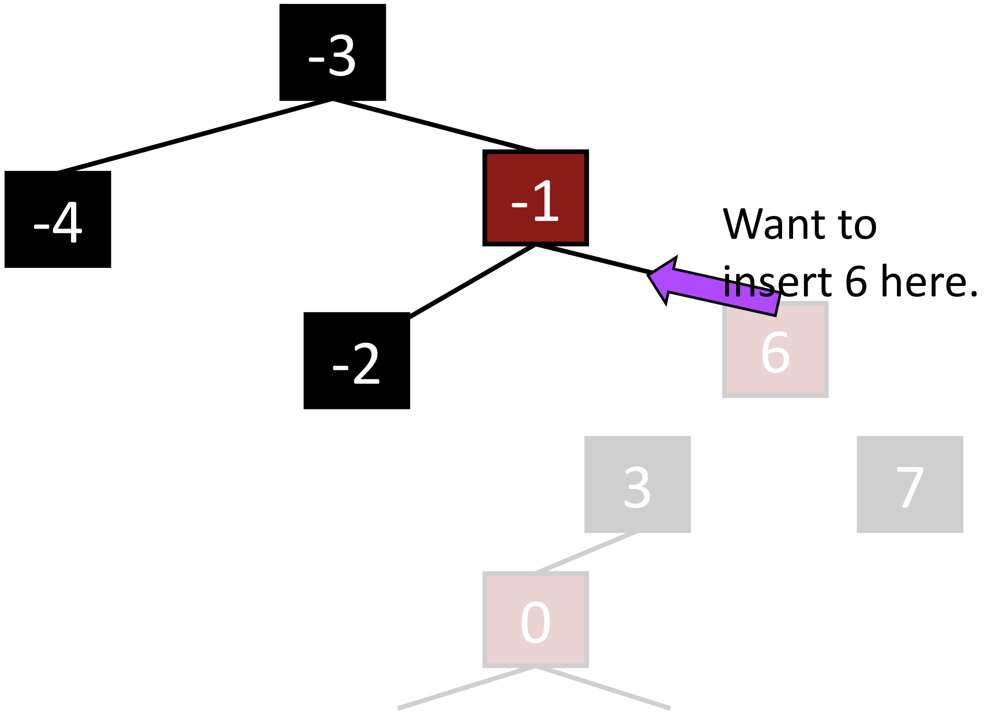
What if it looks like this?



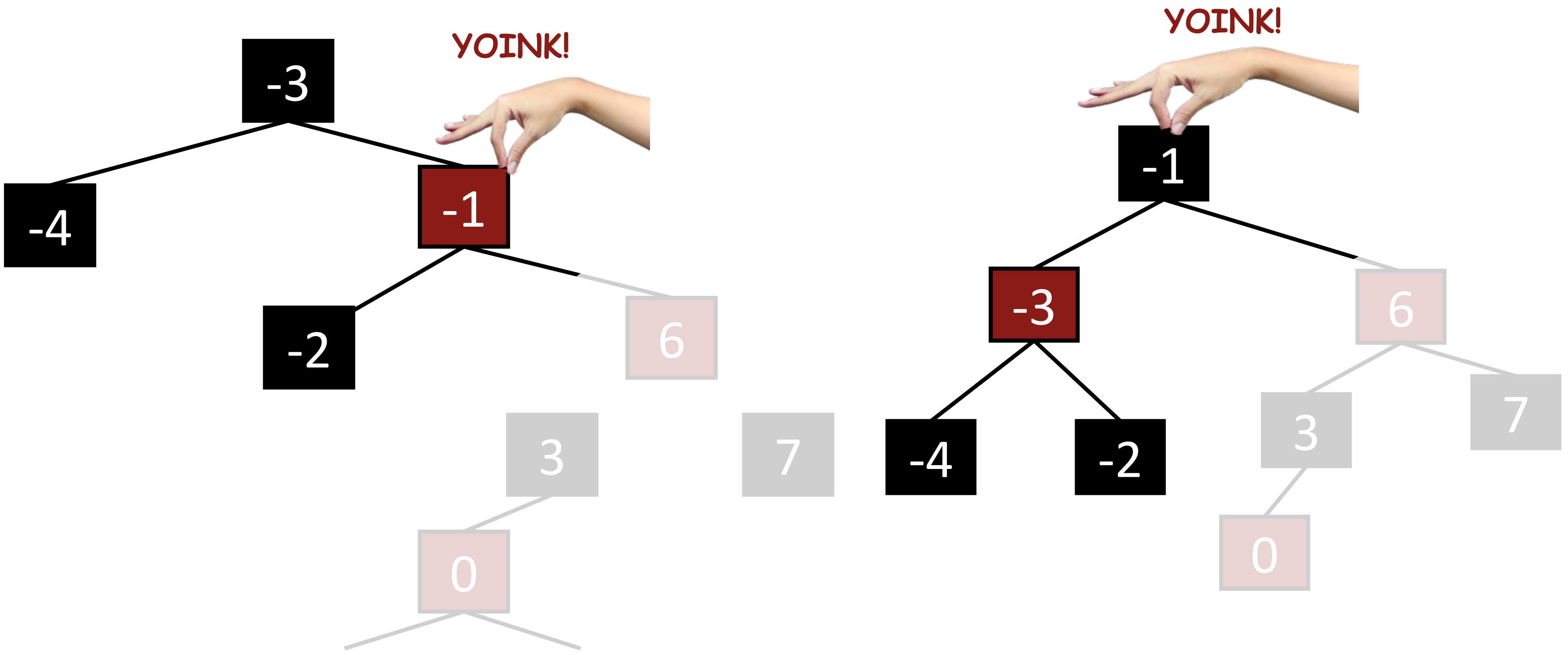


Argue that this is a good thing to do!

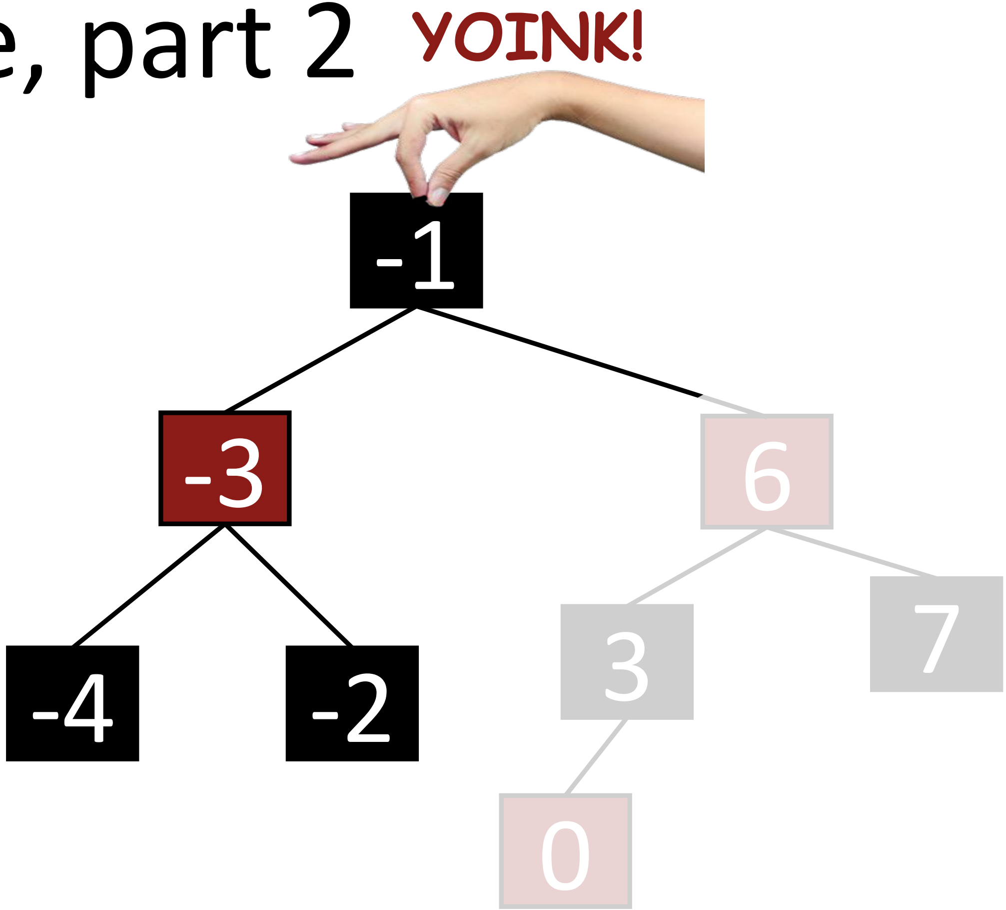
Example, part 2



Example, part 2

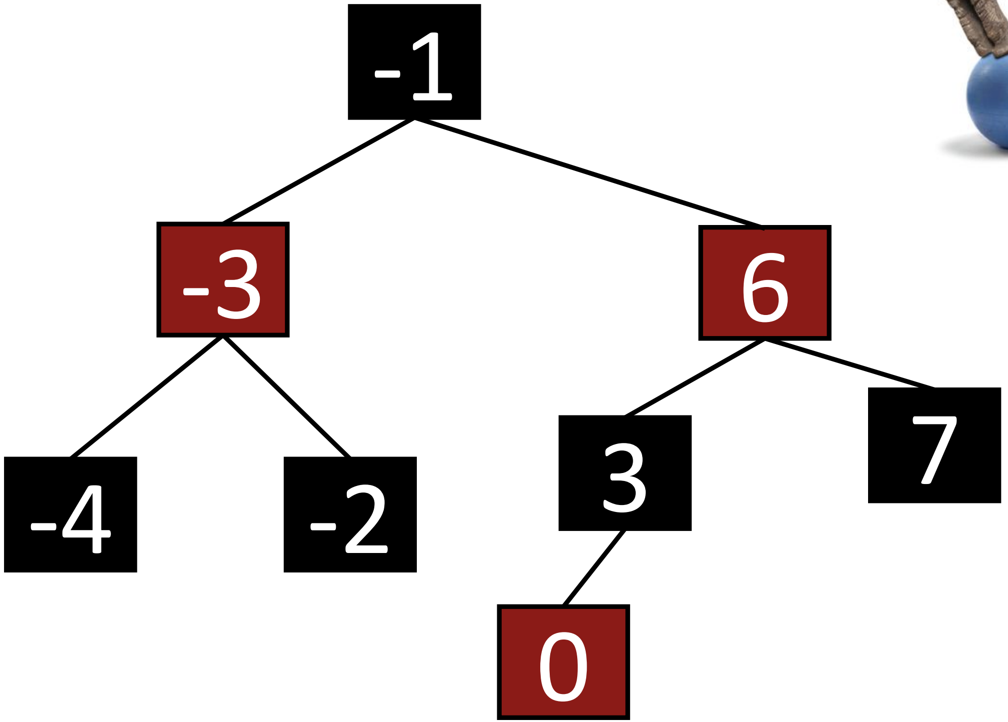


Example, part 2

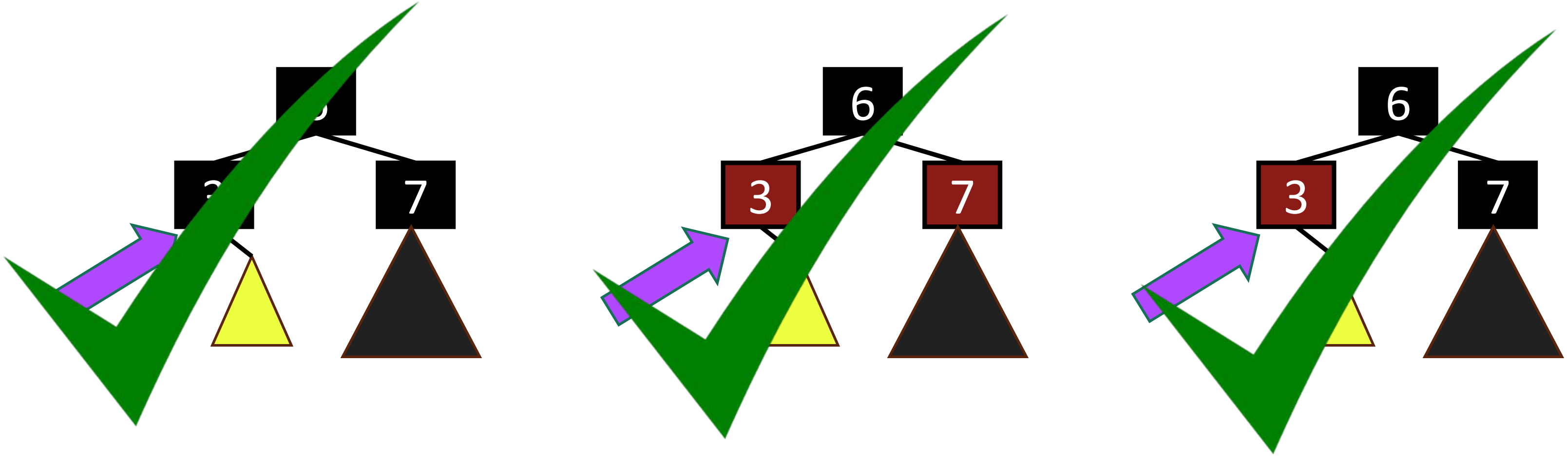


Example, part 2

TA-DA!



Many cases

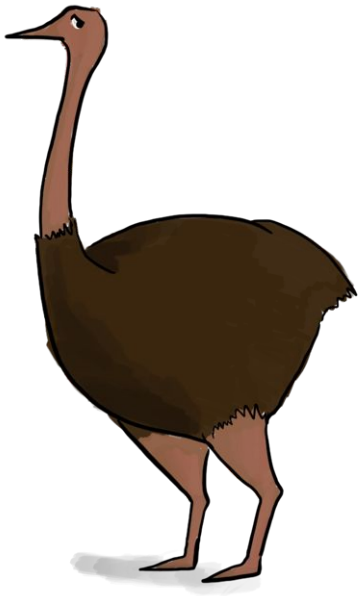


- Suppose we want to insert 0 **here**.

- 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!

- 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 have we learned?

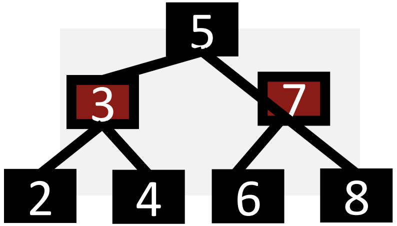
- Red-Black Trees always have height at most 2log(n+1).
- As with general Binary Search Trees, all operations are O(height)
- So all operations with RBTrees are O(log(n)).

Conclusion: The best of both worlds

	Sorted Arrays	Linked Lists	Binary Search 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



How was the pace today?

COMP 285
Analysis of Algorithms

Welcome to COMP 285

Lecture 11: BSTs + Self-Balancing Trees

Lecturer: Chris Lucas (cflucas@ncat.edu)