Lecture 7

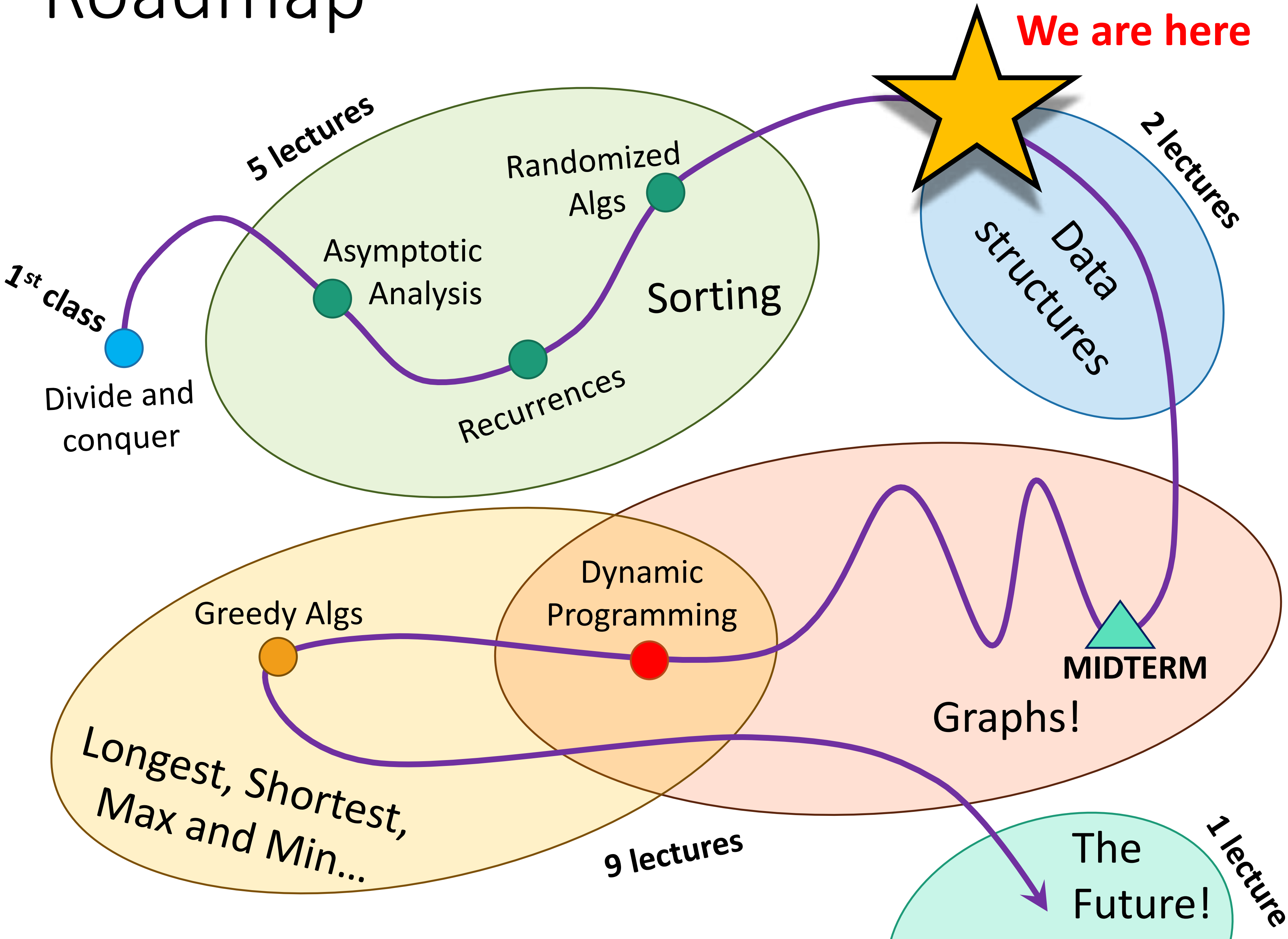
Binary Search Trees and Red-Black Trees

Announcements

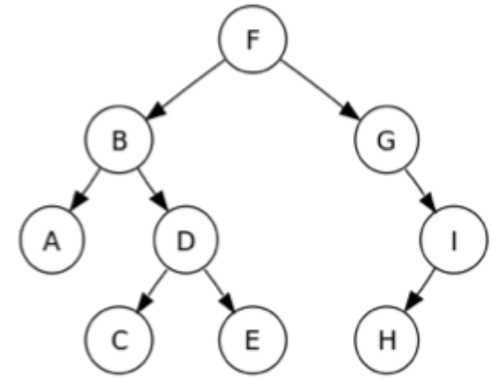
- I won’t be here on Monday, Greg Valiant will be.
- HW3 will be released on Friday
  - Hope you enjoyed your week off ☺
- All OAE letters and exam conflicts are past due!
  - If you have any, please email cs161-win1819-staff@lists.stanford.edu ASAP!
  - (Note about privacy: this list is only read by Mary, Richard and Dana.)

More detailed schedule on the website!

Roadmap

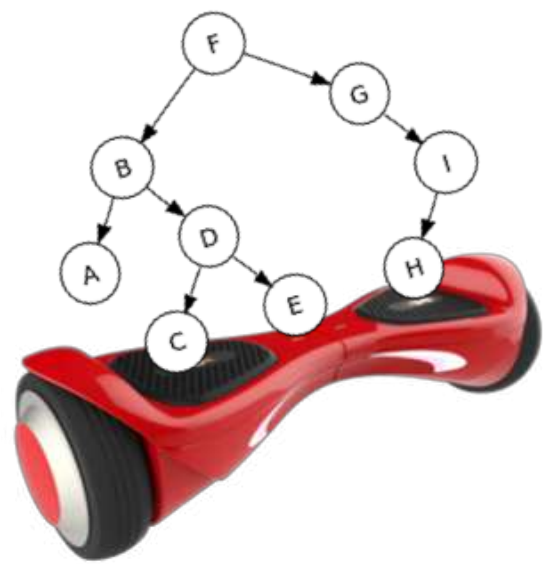


Today



- Begin a brief foray into data structures!
    - See CS 166 for more!
- Binary search trees
    - You may remember these from CS 106B
    - They are better when they're balanced.

this will lead us to…

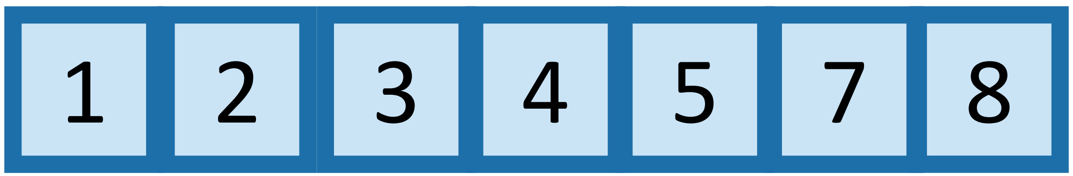


- Self-Balancing Binary Search Trees
    - **Red-Black** trees.

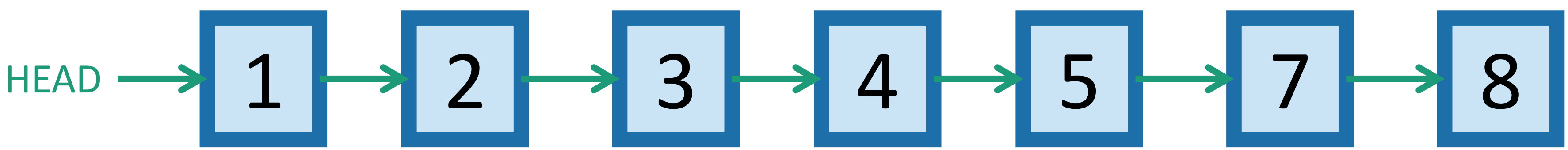
Some data structures

for storing objects like $\boxed{5}$ (aka, nodes with keys)

- (Sorted) arrays:

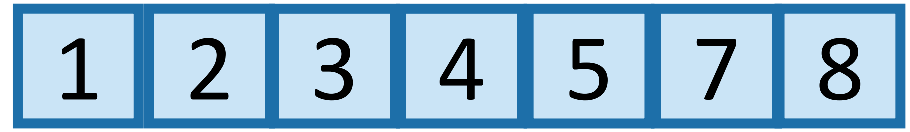


- (Sorted) linked lists:

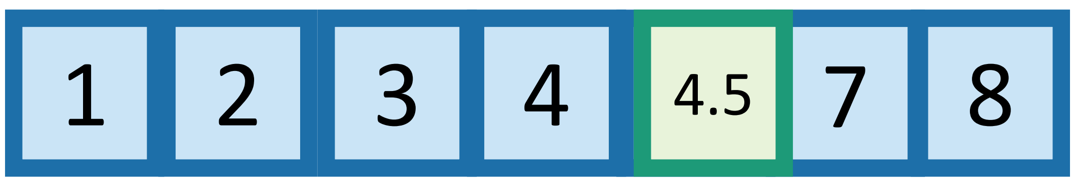


- Some basic operations:
  - INSERT, DELETE, SEARCH

Sorted Arrays

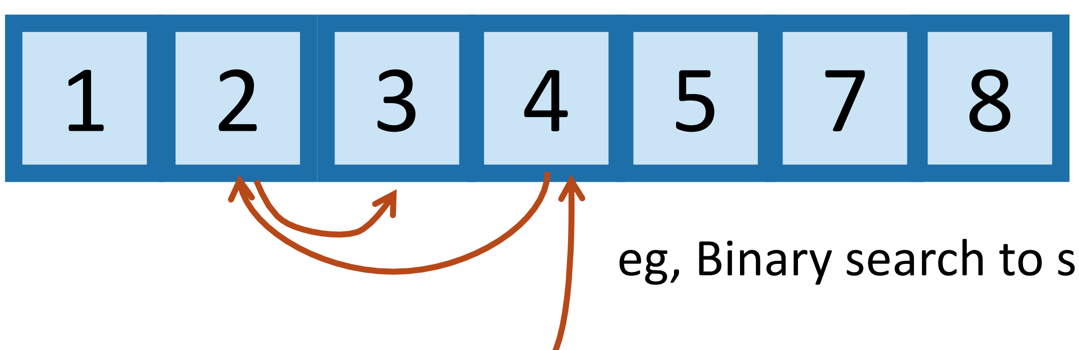


- O(n) INSERT/DELETE:
  - First, find the relevant element (time O(log(n)) as below), and then move a bunch elements in the array:



eg, insert 4.5

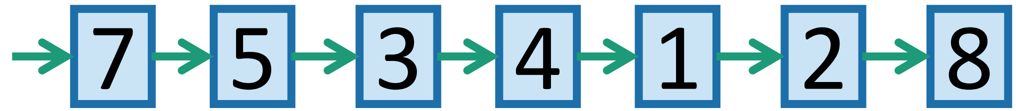
- O(log(n)) SEARCH:



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

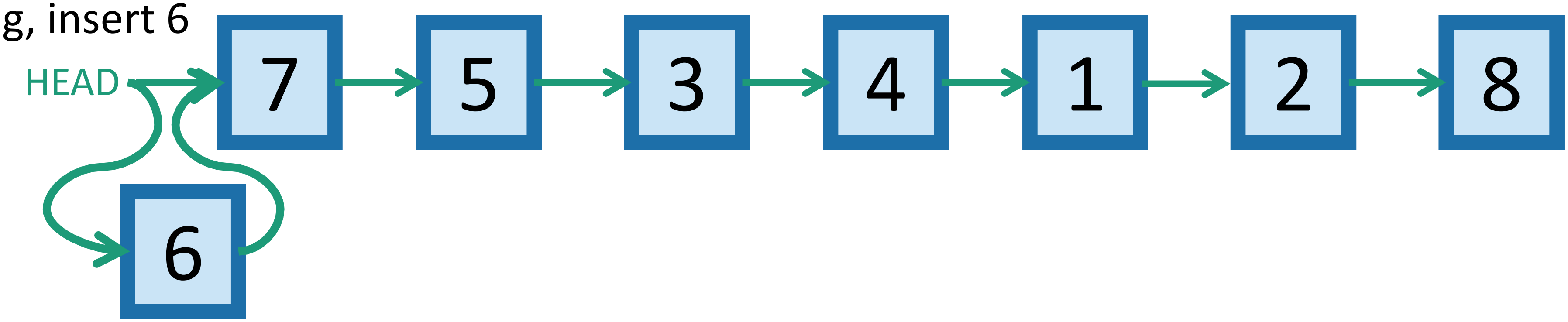
UNSorted linked lists

NOTE: In class I had sorted linked lists as an example, but not-necessarily-sorted is what I meant to put. Not sure what happened…

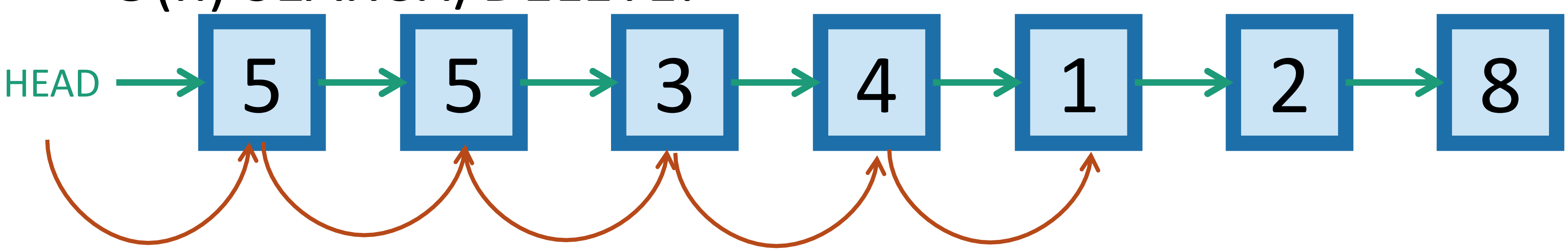


- O(1) INSERT:

eg, insert 6



- O(n) SEARCH/DELETE:



eg, search for 1 (and then you could delete it by manipulating pointers).

Motivation for Binary Search Trees

TODAY!

	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))

For today all keys are distinct.

Binary tree terminology

Each node has two children.

The left child of $\boxed{3}$ is $\boxed{2}$

The right child of $\boxed{3}$ is $\boxed{4}$

The parent of $\boxed{3}$ is $\boxed{5}$

$\boxed{2}$ is a descendant of $\boxed{5}$

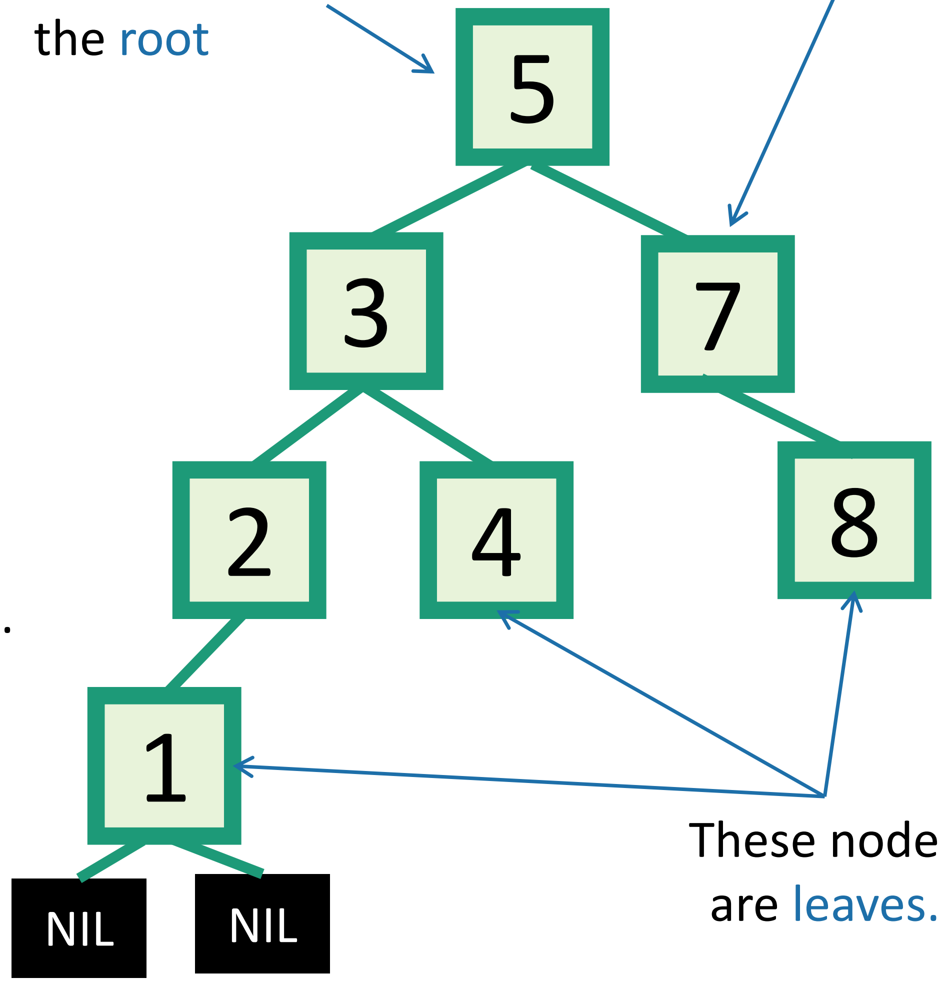
Each node has a pointer to its left child, right child, and parent.

Both children of $\boxed{1}$ are NIL.
(I won't usually draw them).

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

This is a node.
It has a key (7).

This node is the root

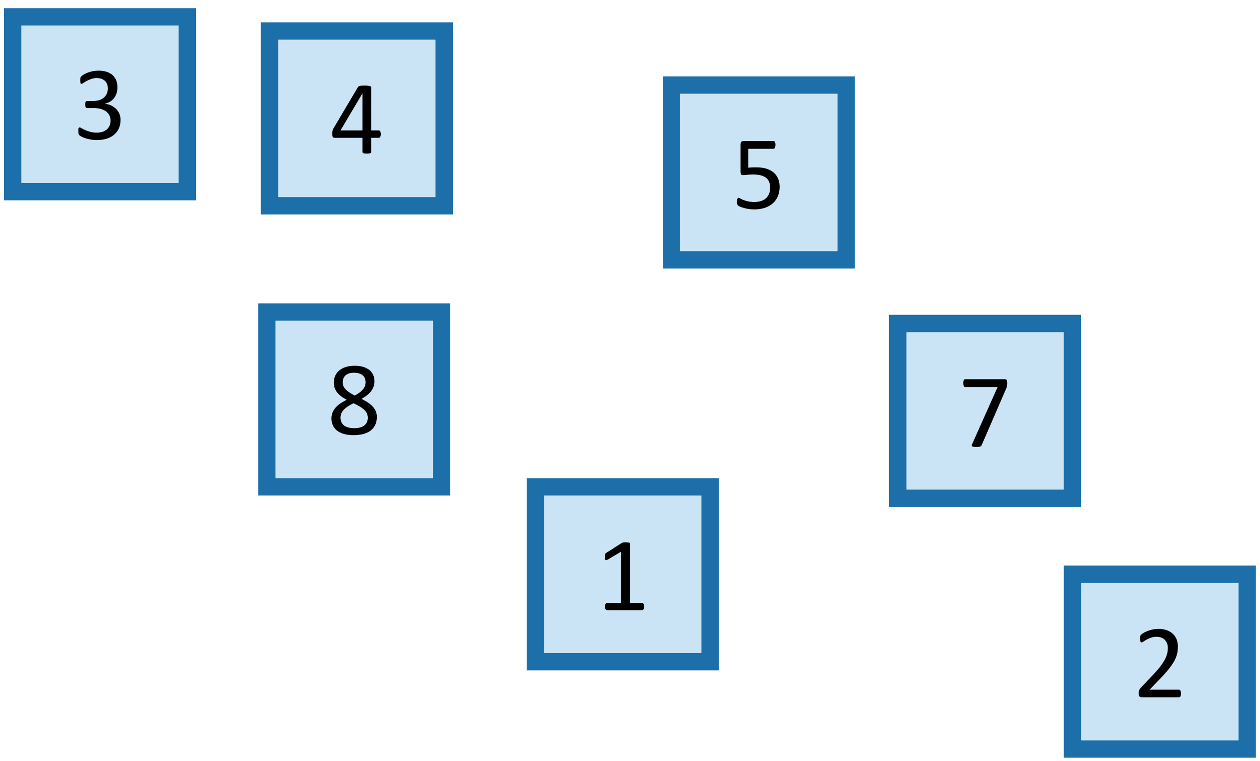


These nodes are leaves.

From your pre-lecture exercise…

Binary Search Trees

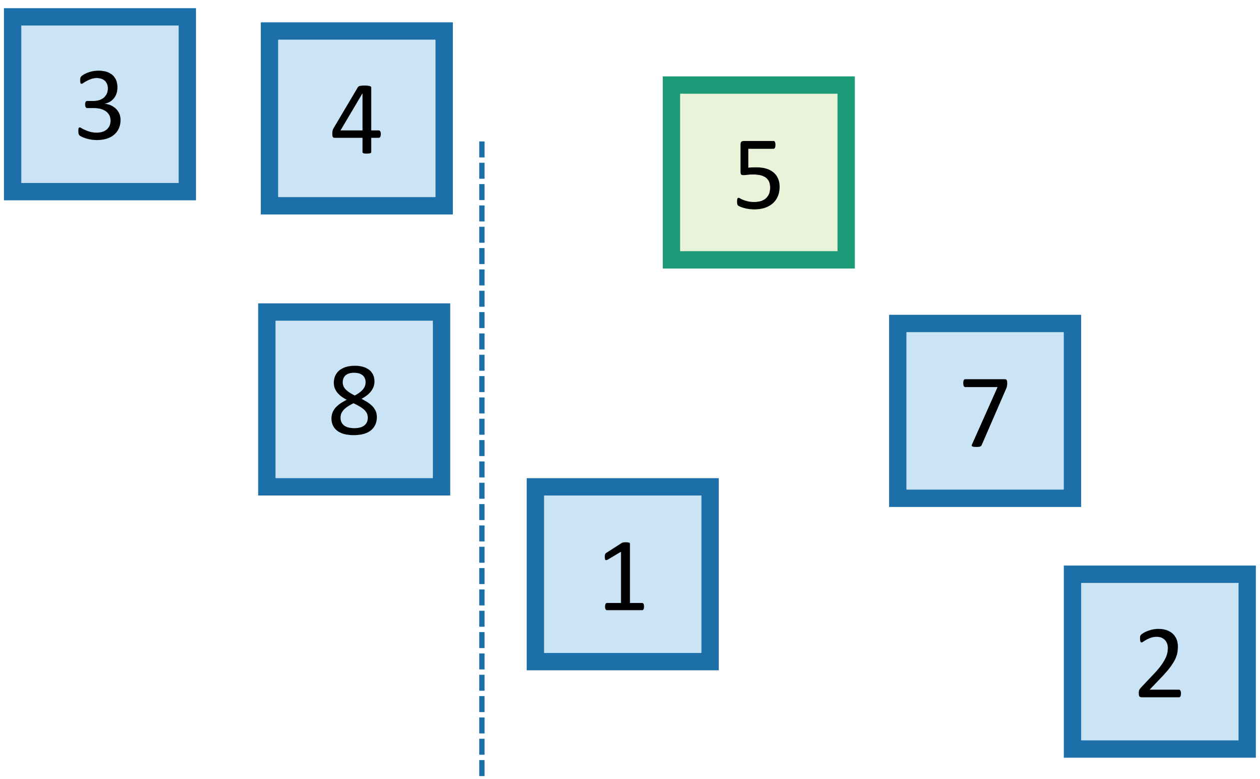
- A BST is a binary tree so that:
  - Every LEFT descendant of a node has key less than that node.
  - Every RIGHT descendant of a node has key larger than that node.
- Example of building a binary search tree:



From your pre-lecture exercise…

Binary Search Trees

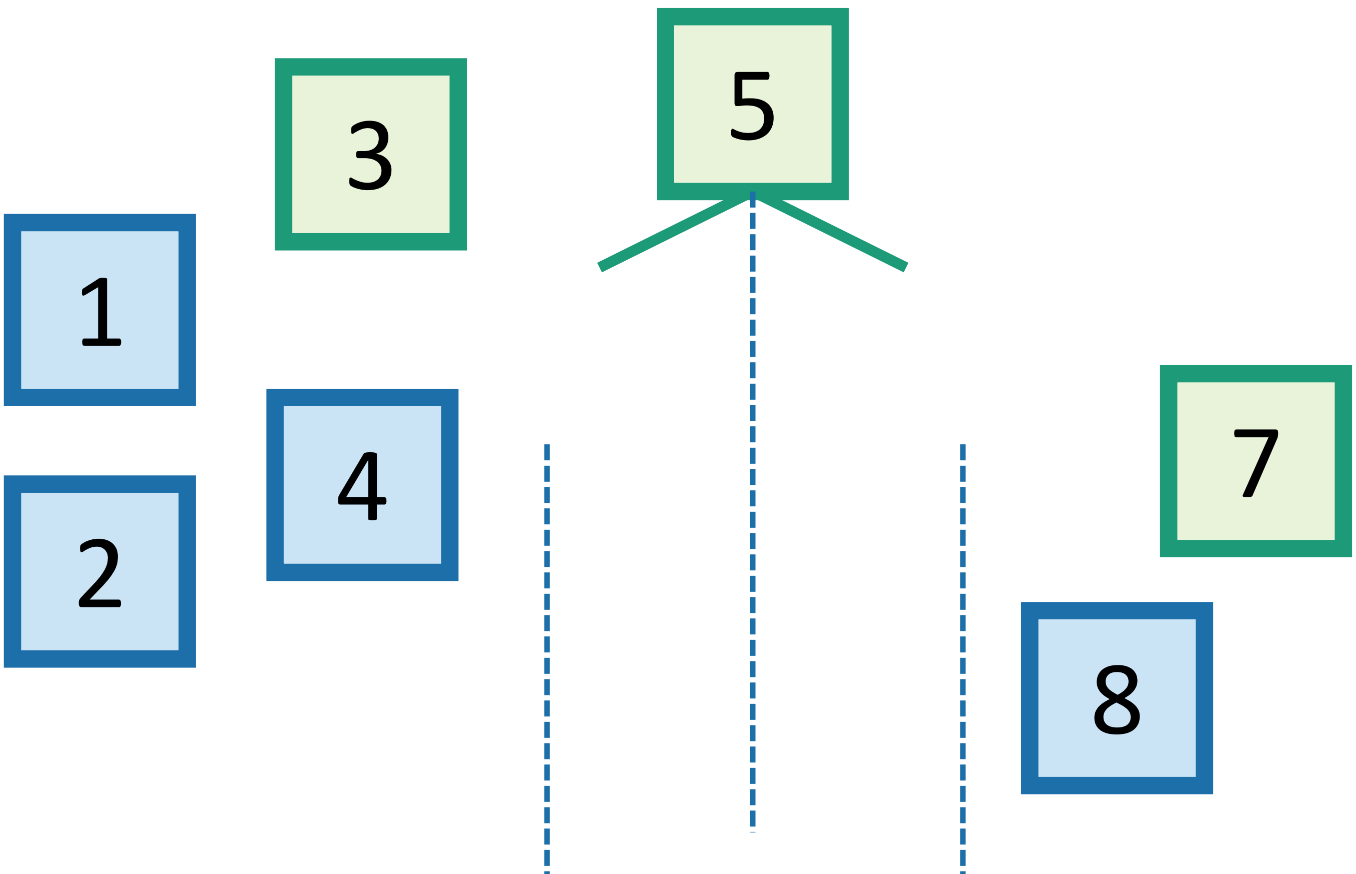
- A BST is a binary tree so that:
  - Every LEFT descendant of a node has key less than that node.
  - Every RIGHT descendant of a node has key larger than that node.
- Example of building a binary search tree:



From your pre-lecture exercise…

Binary Search Trees

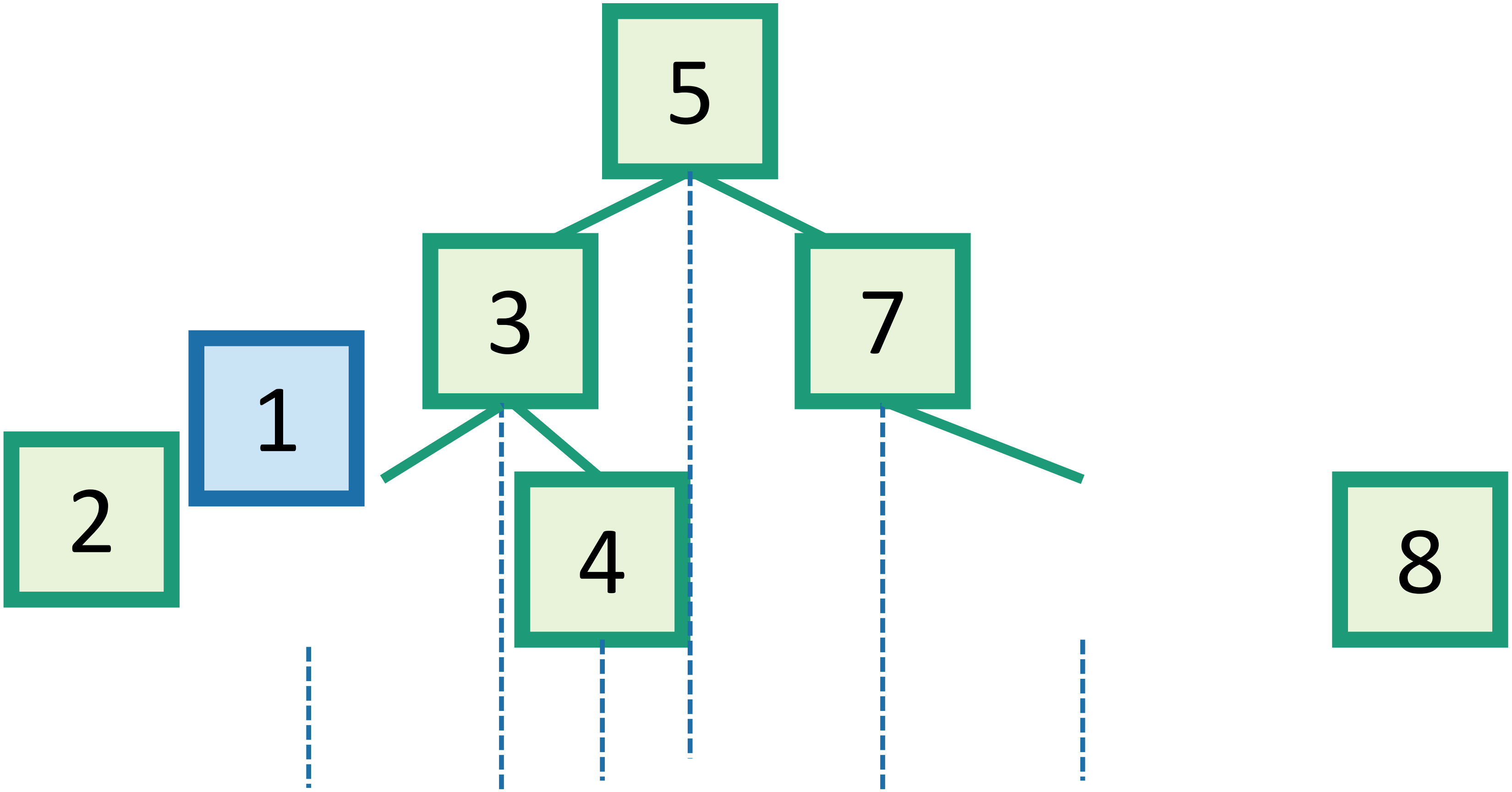
- A BST is a binary tree so that:
  - Every LEFT descendant of a node has key less than that node.
  - Every RIGHT descendant of a node has key larger than that node.
- Example of building a binary search tree:



From your pre-lecture exercise…

Binary Search Trees

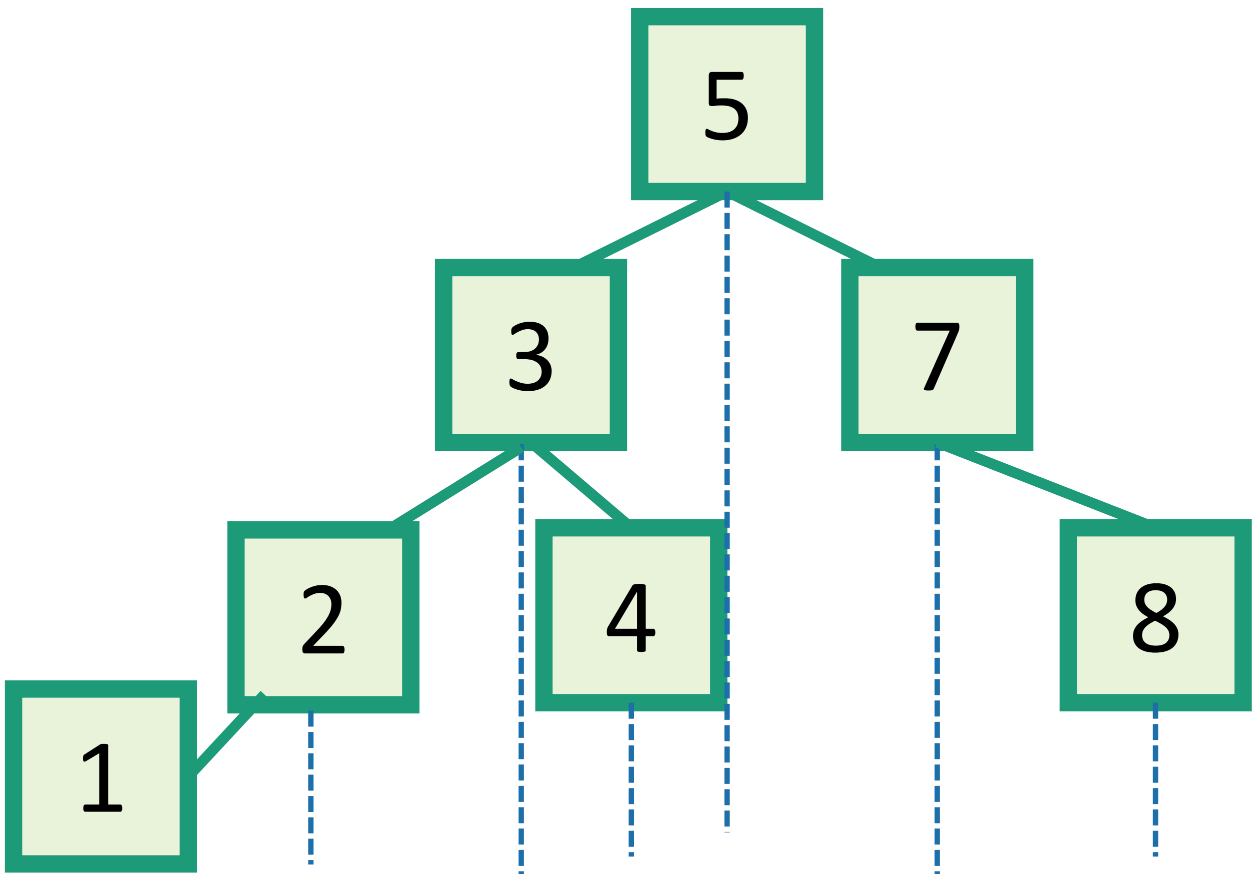
- A BST is a binary tree so that:
  - Every LEFT descendant of a node has key less than that node.
  - Every RIGHT descendant of a node has key larger than that node.
- Example of building a binary search tree:



From your pre-lecture exercise…

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:

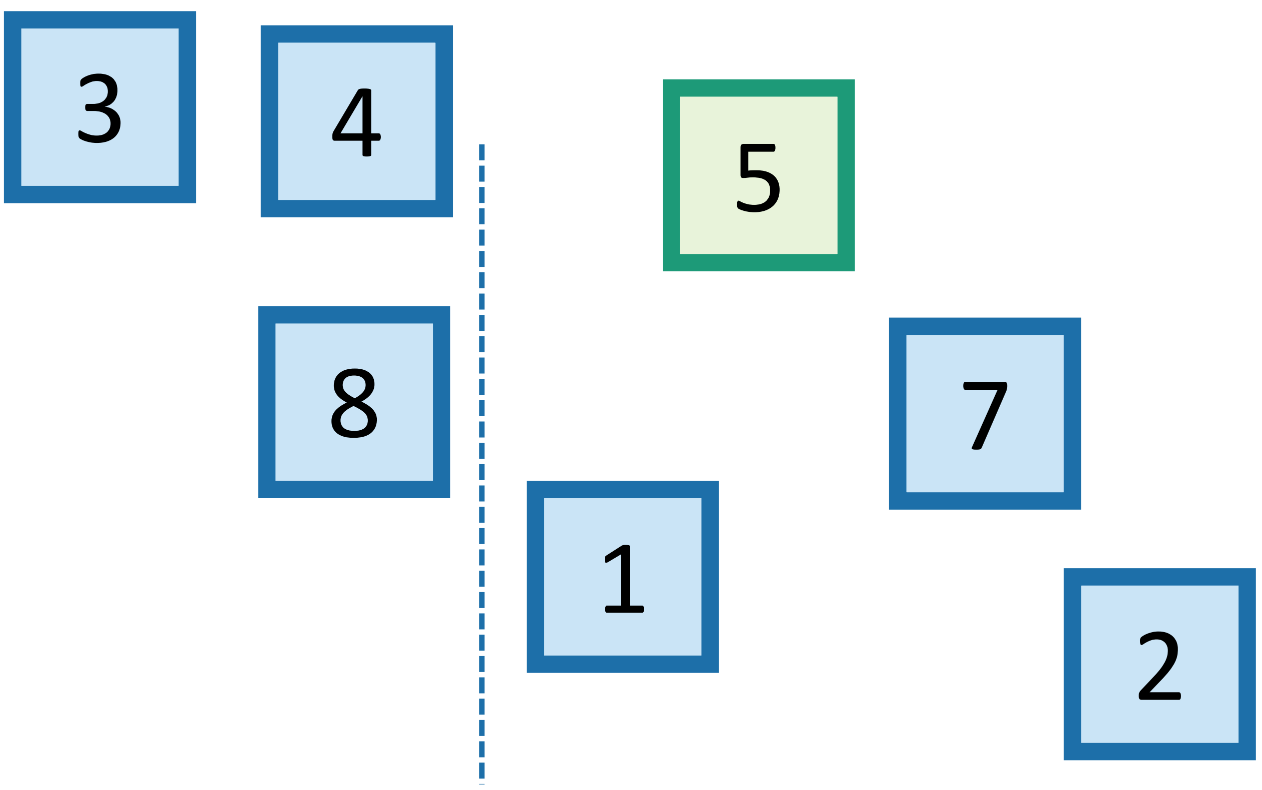


Q: Is this the only binary search tree I could possibly build with these values?

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

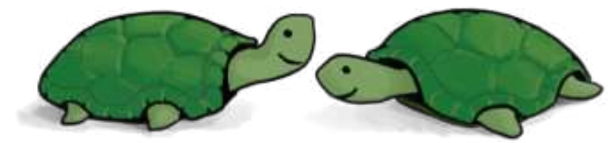
Aside: this should look familiar

kinda like QuickSort

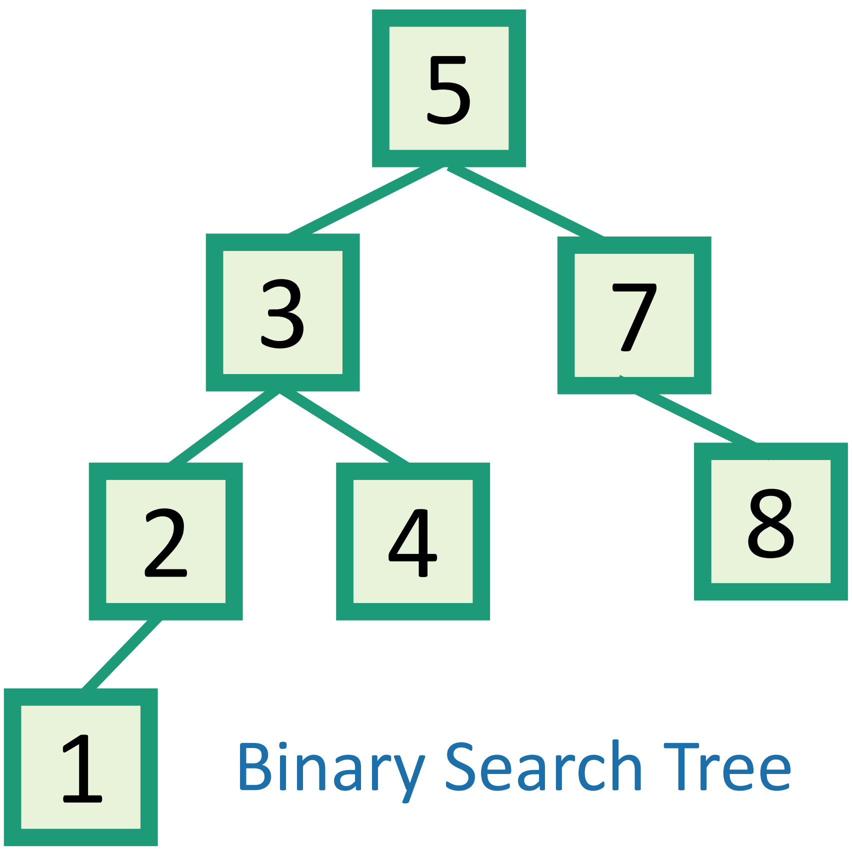


Binary Search Trees

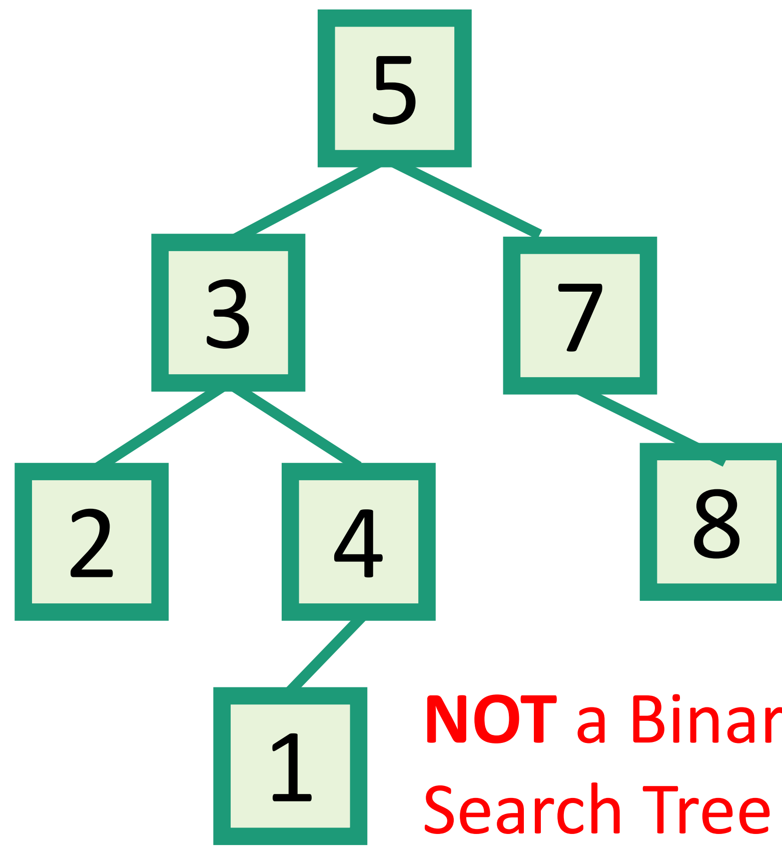
Which of these is a BST?



- 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

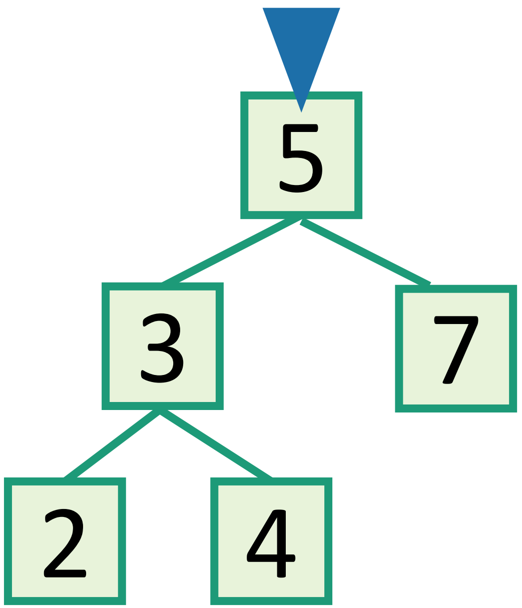


NOT a Binary Search Tree

Aside: In-Order Traversal of BSTs

- Output all the elements in sorted order!

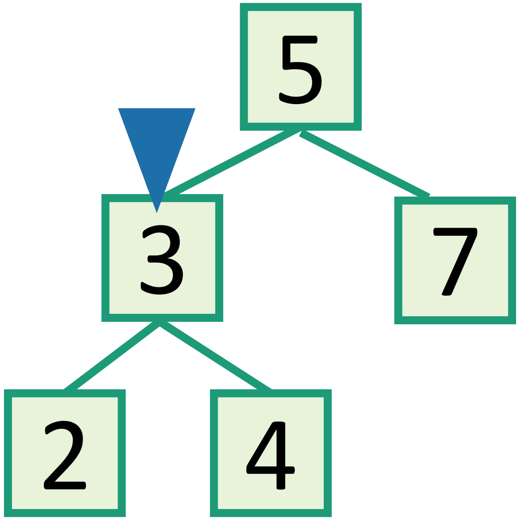
- inOrderTraversal(x):
    - if x!= NIL:
        - inOrderTraversal(x.left)
        - print(x.key)
        - inOrderTraversal(x.right)



Aside: In-Order Traversal of BSTs

- Output all the elements in sorted order!

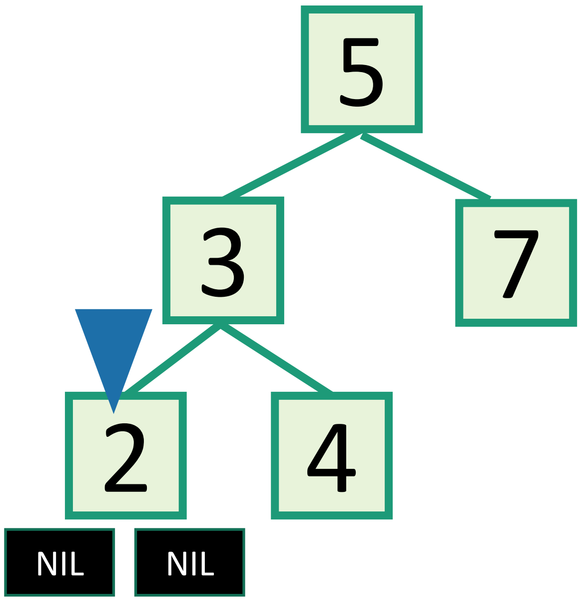
- inOrderTraversal(x):
  - if x!= NIL:
    - inOrderTraversal(x.left)
    - print(x.key)
    - inOrderTraversal(x.right)



Aside: In-Order Traversal of BSTs

- Output all the elements in sorted order!

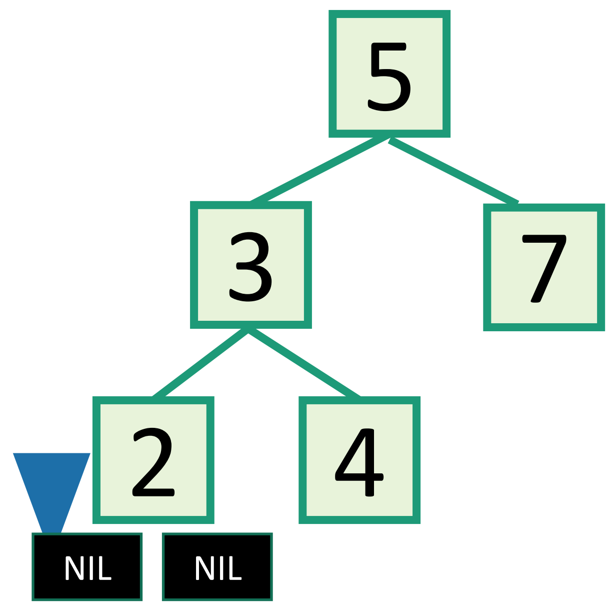
- inOrderTraversal(x):
  - if x!= NIL:
    - inOrderTraversal(x.left)
    - print(x.key)
    - inOrderTraversal(x.right)



Aside: In-Order Traversal of BSTs

- Output all the elements in sorted order!

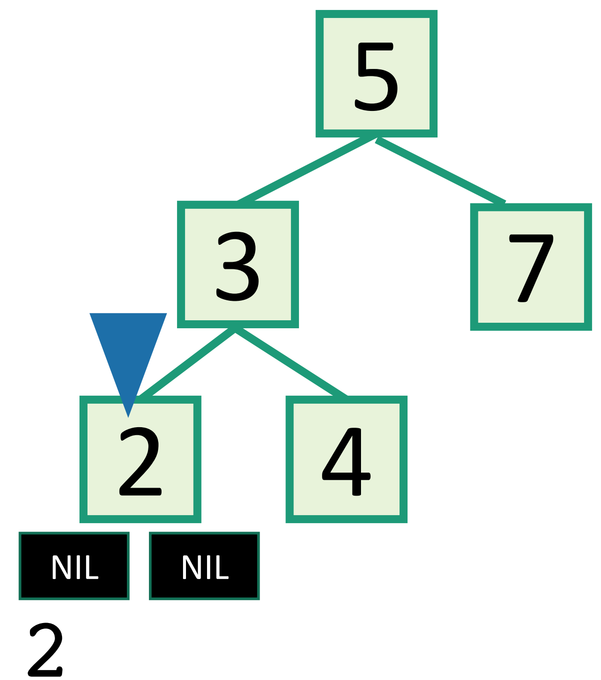
- inOrderTraversal(x):
  - if x!= NIL:
    - inOrderTraversal(x.left)
    - print(x.key)
    - inOrderTraversal(x.right)



Aside: In-Order Traversal of BSTs

- Output all the elements in sorted order!

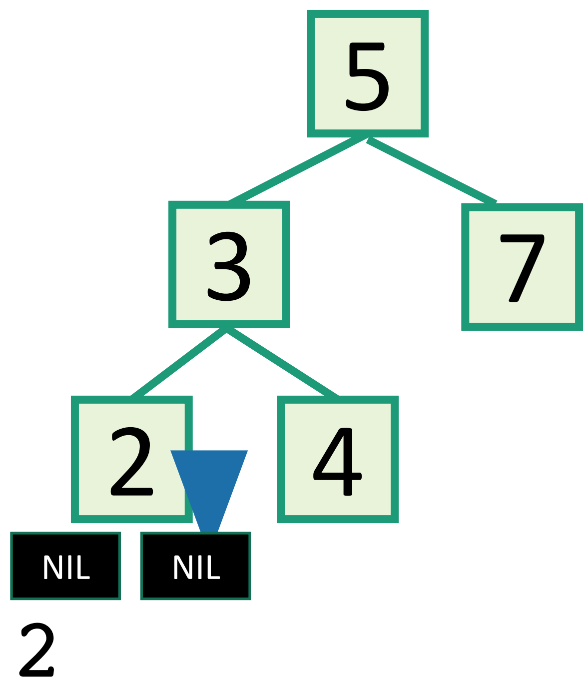
- inOrderTraversal(x):
  - if x!= NIL:
    - inOrderTraversal(x.left)
    - print(x.key)
    - inOrderTraversal(x.right)



Aside: In-Order Traversal of BSTs

- Output all the elements in sorted order!

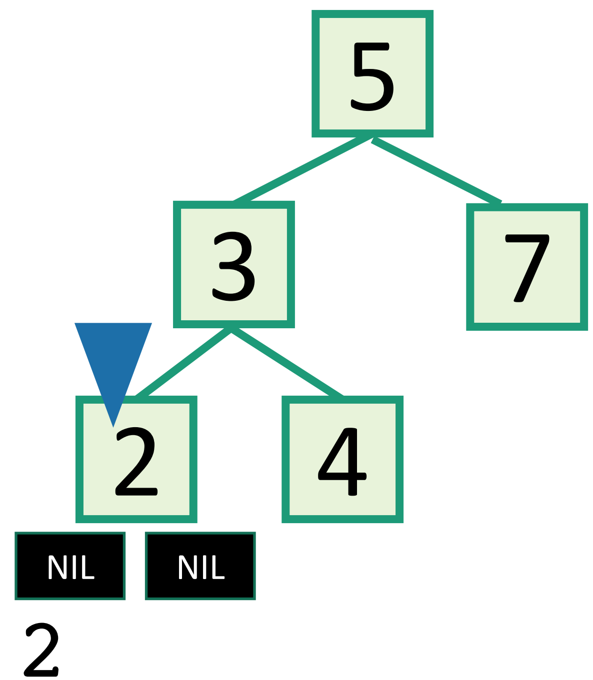
- inOrderTraversal(x):
  - if x!= NIL:
    - inOrderTraversal(x.left)
    - print(x.key)
    - inOrderTraversal(x.right)



Aside: In-Order Traversal of BSTs

- Output all the elements in sorted order!

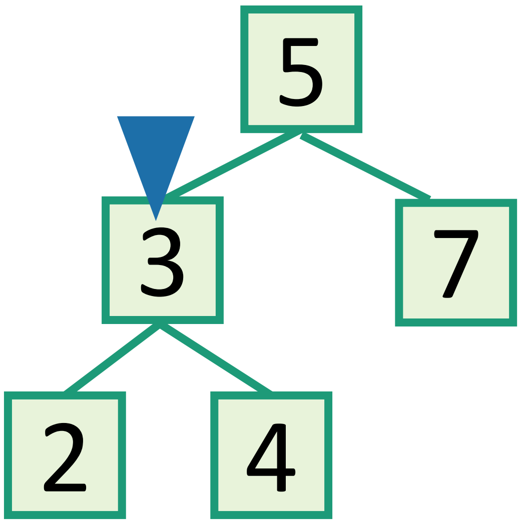
- inOrderTraversal(x):
  - if x!= NIL:
    - inOrderTraversal(x.left)
    - print(x.key)
    - inOrderTraversal(x.right)



Aside: In-Order Traversal of BSTs

- Output all the elements in sorted order!

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

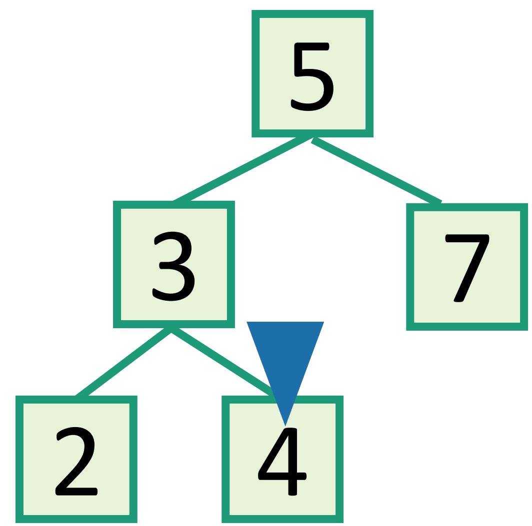


2 3

Aside: In-Order Traversal of BSTs

- Output all the elements in sorted order!

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

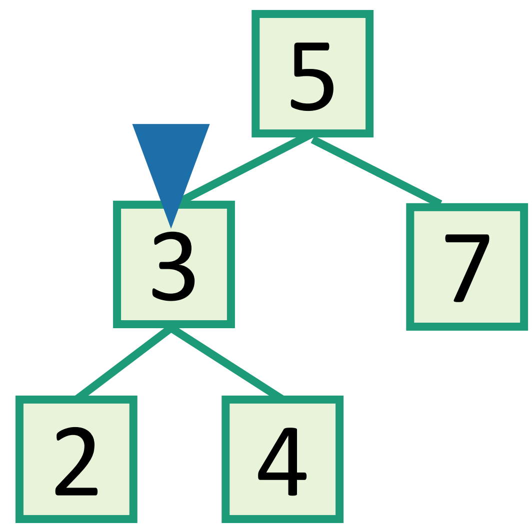


2 3 4

Aside: In-Order Traversal of BSTs

- Output all the elements in sorted order!

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

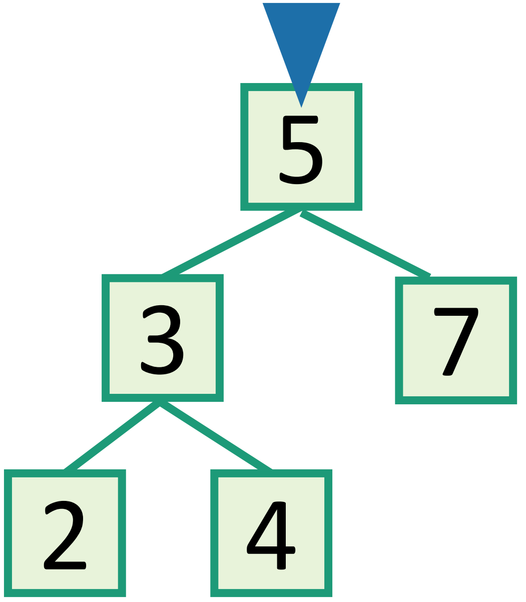


2 3 4

Aside: In-Order Traversal of BSTs

- Output all the elements in sorted order!

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

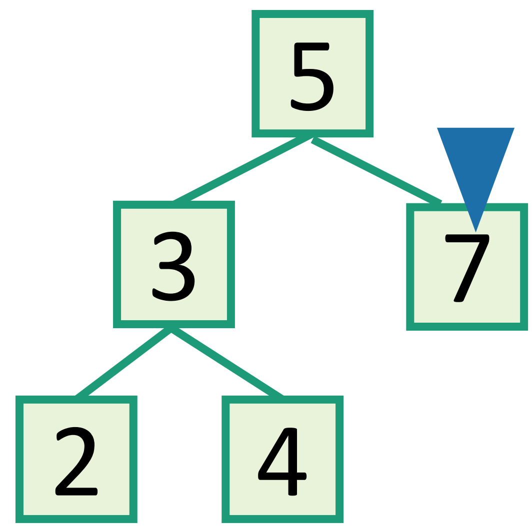


2 3 4 5

Aside: In-Order Traversal of BSTs

- Output all the elements in sorted order!

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

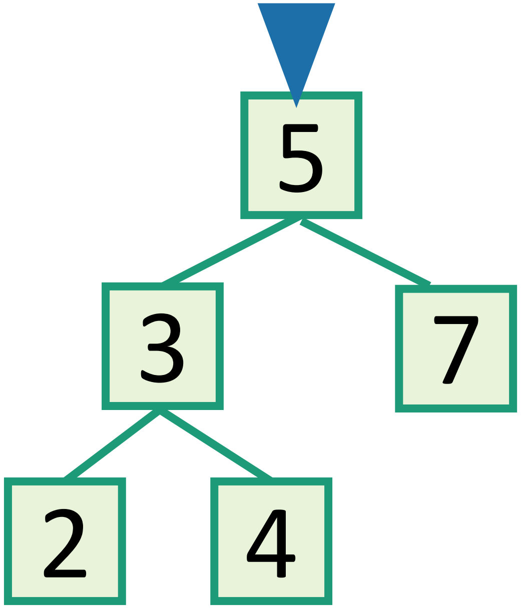


2 3 4 5 7

Aside: In-Order Traversal of BSTs

- Output all the elements in sorted order!

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



- Runs in time O(n).

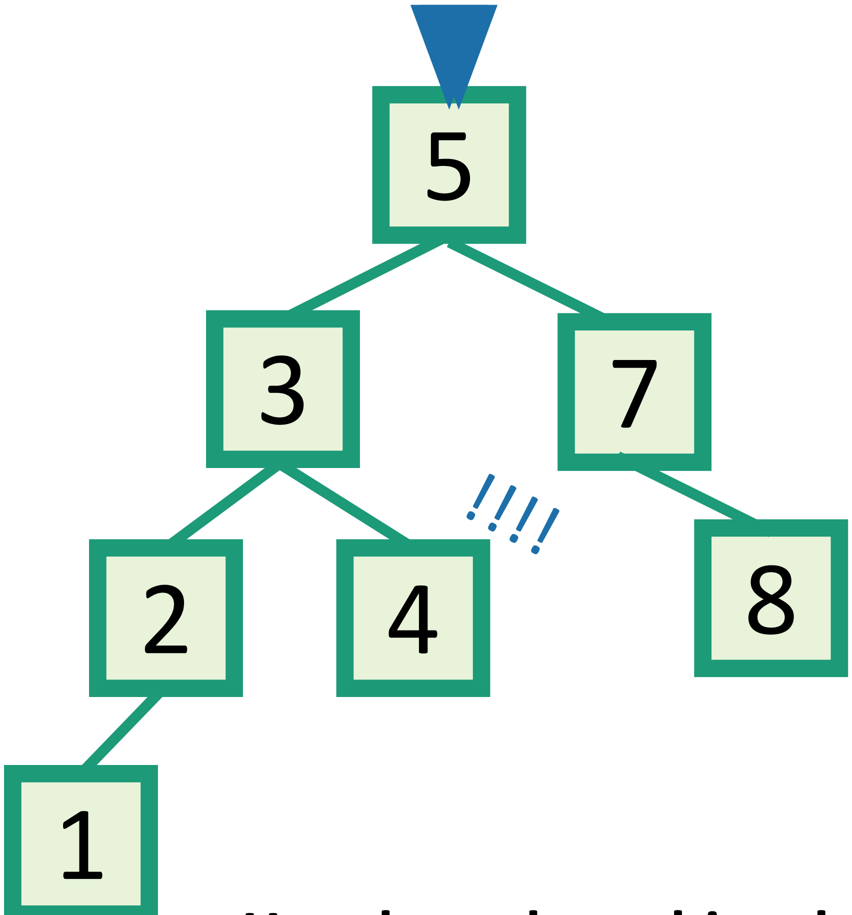
2 3 4 5 7 Sorted!

Back to 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)

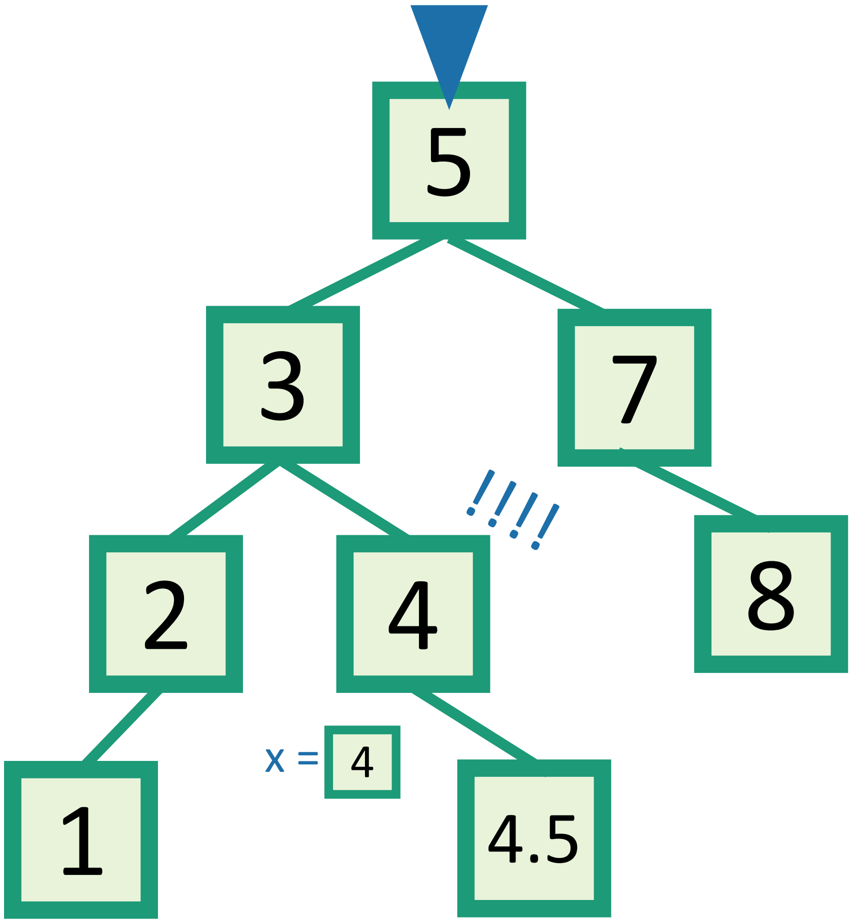
How long does this take?

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

Write pseudocode (or actual code) to implement this!

Ollie the over-achieving ostrich

INSERT in a Binary Search Tree



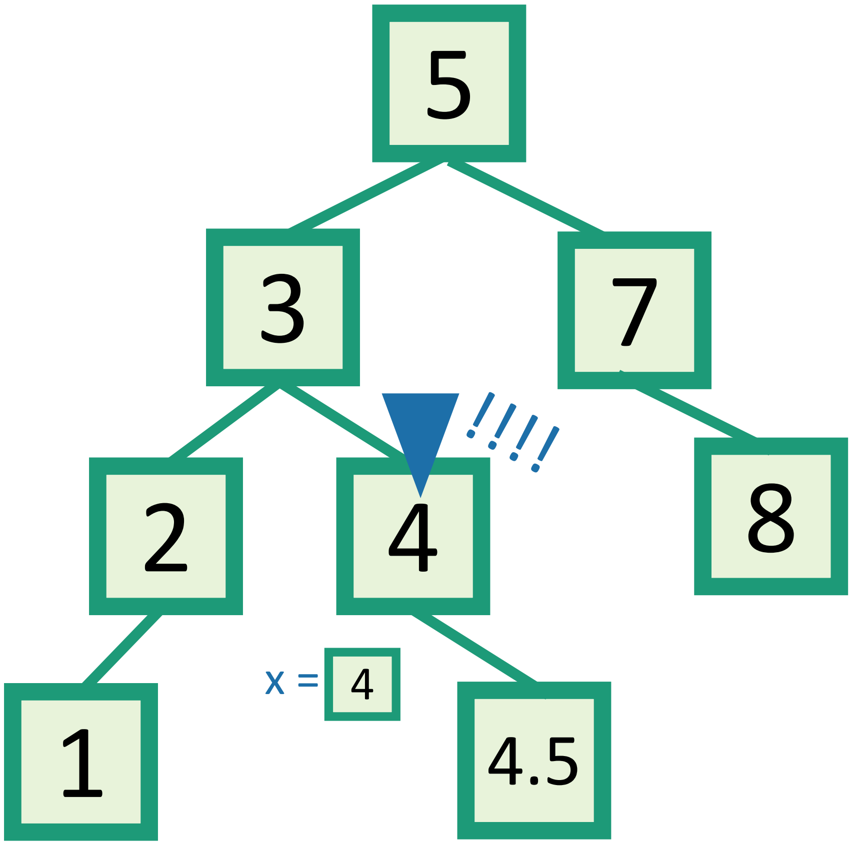
EXAMPLE: Insert 4.5

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

You thought about this on your pre-lecture exercise! (See skipped slide for pseudocode.)

INSERT in a Binary Search Tree

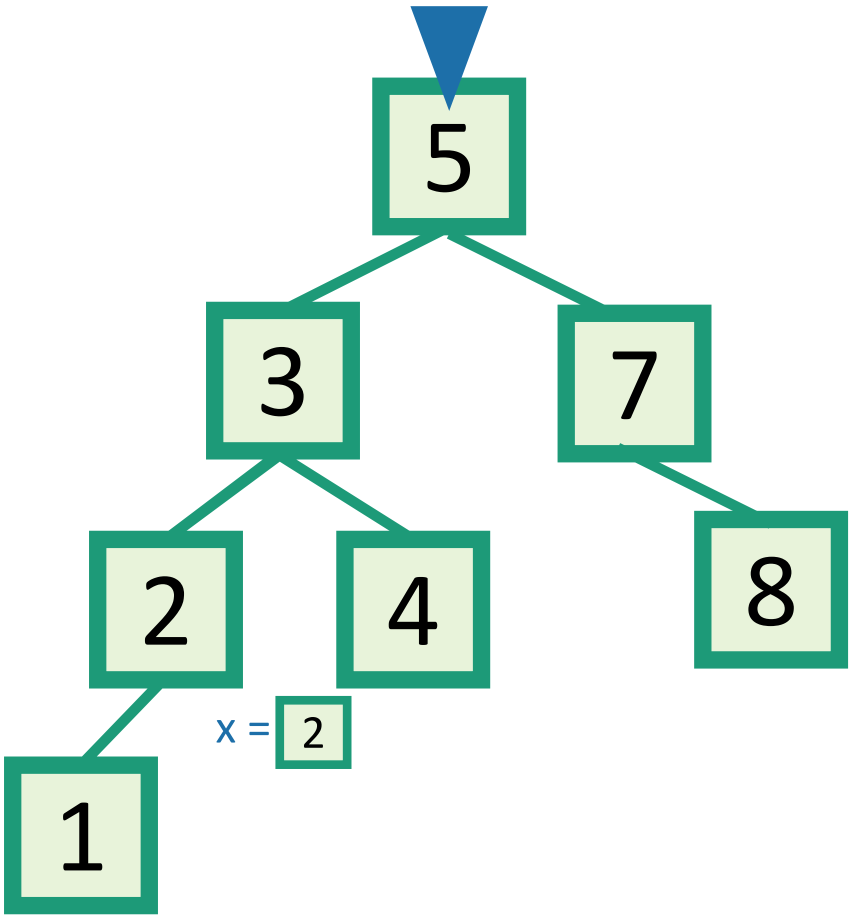
This slide skipped in class – here for reference



EXAMPLE: Insert 4.5

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

DELETE in a Binary Search Tree



EXAMPLE: Delete 2

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

You thought about this in your pre-lecture exercise too!

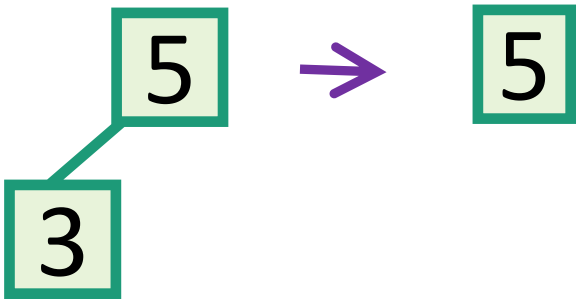
This is a bit more complicated...see the skipped slides for some pictures of the different cases.

DELETE in a Binary Search Tree

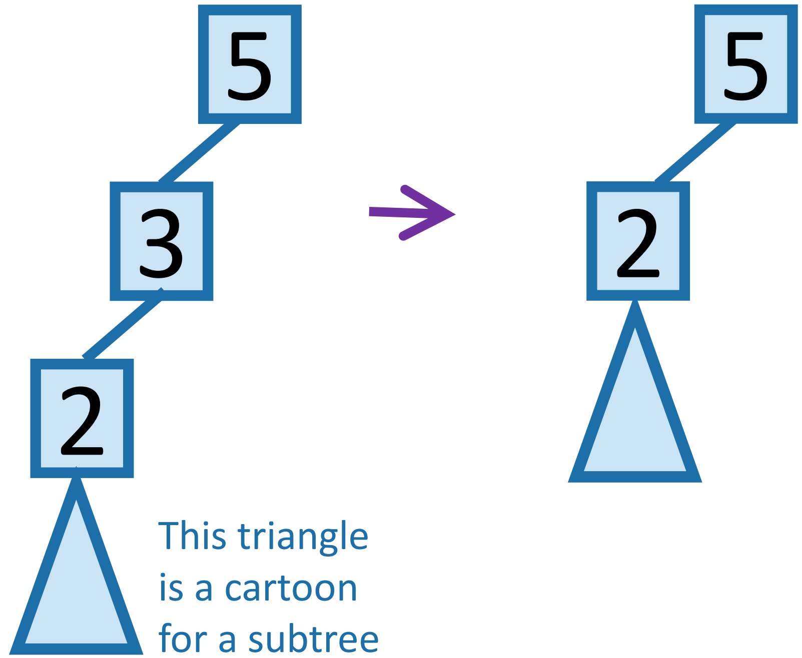
several cases (by example)

say we want to delete 3

This slide skipped in class – here for reference!



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



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

Write pseudocode for all of these!

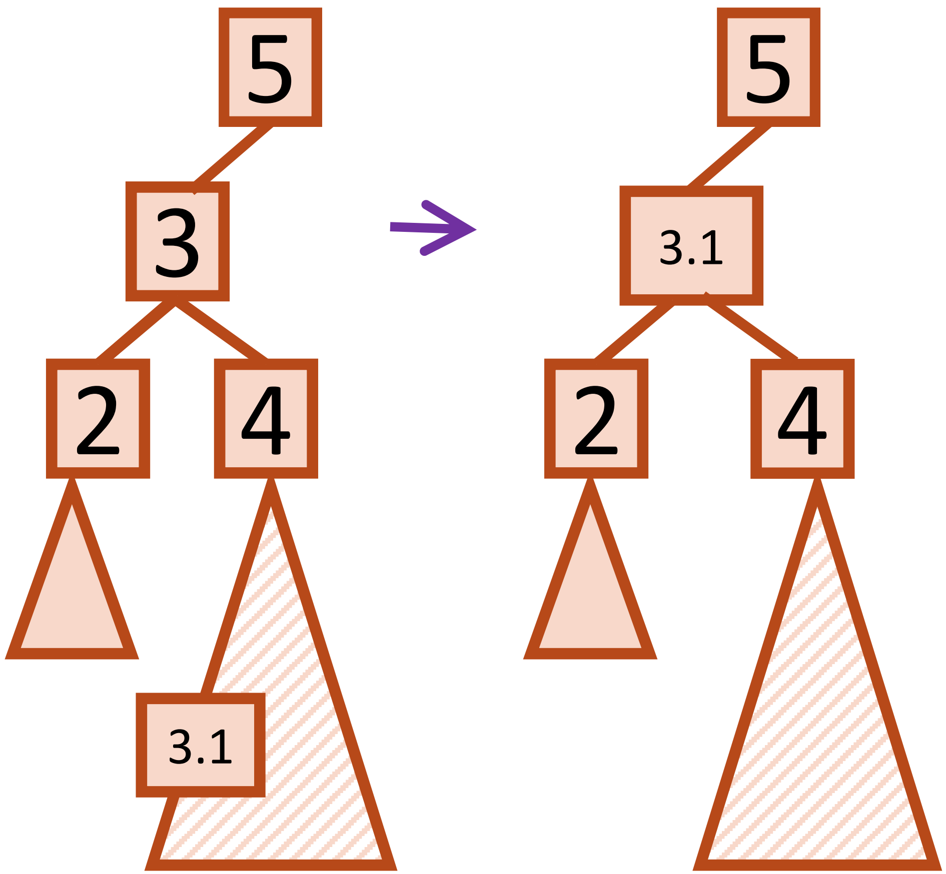
Siggi the Studious Stork

DELETE in a Binary Search Tree

ctd.

This slide skipped in class – here for reference!

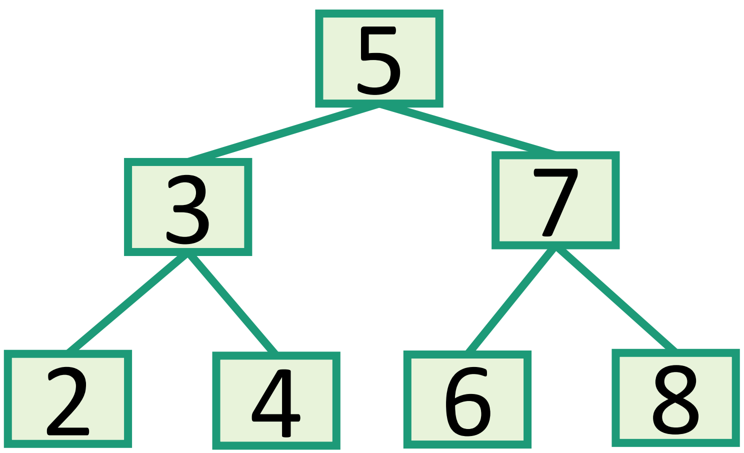
Case 3: if 3 has two children,
replace 3 with it's immediate successor.
(aka, next biggest thing after 3)



- 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)

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

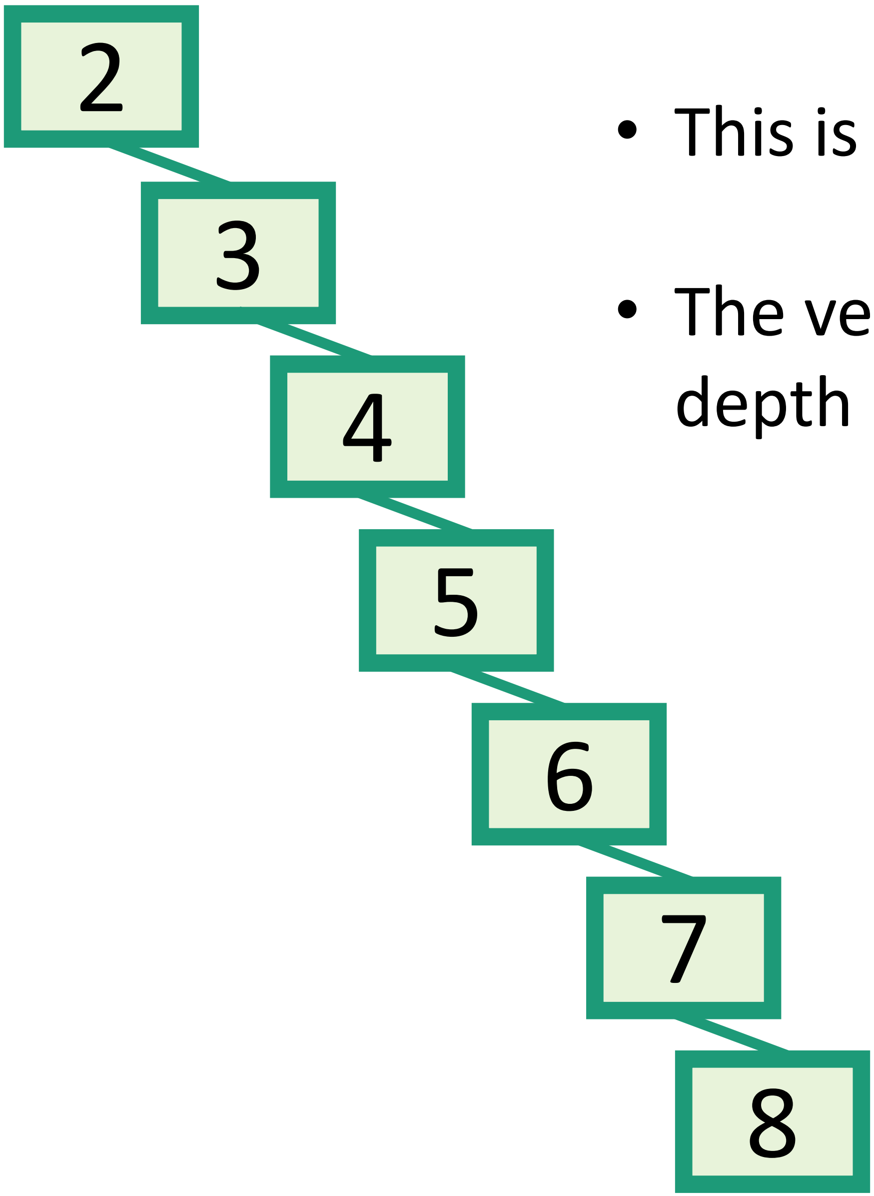
Lucky the lackadaisical lemur.

Wait a second…

Plucky the Pedantic Penguin

How long does search take?

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?

How often is "every so often" in the worst case? It's actually pretty often!

Ollie the over-achieving ostrich

- Goal: Fast SEARCH/INSERT/DELETE
- All these things take time O(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.
    - At least $\Omega(n)$ every so often….

- Turns out that's not a great idea. Instead we turn to…

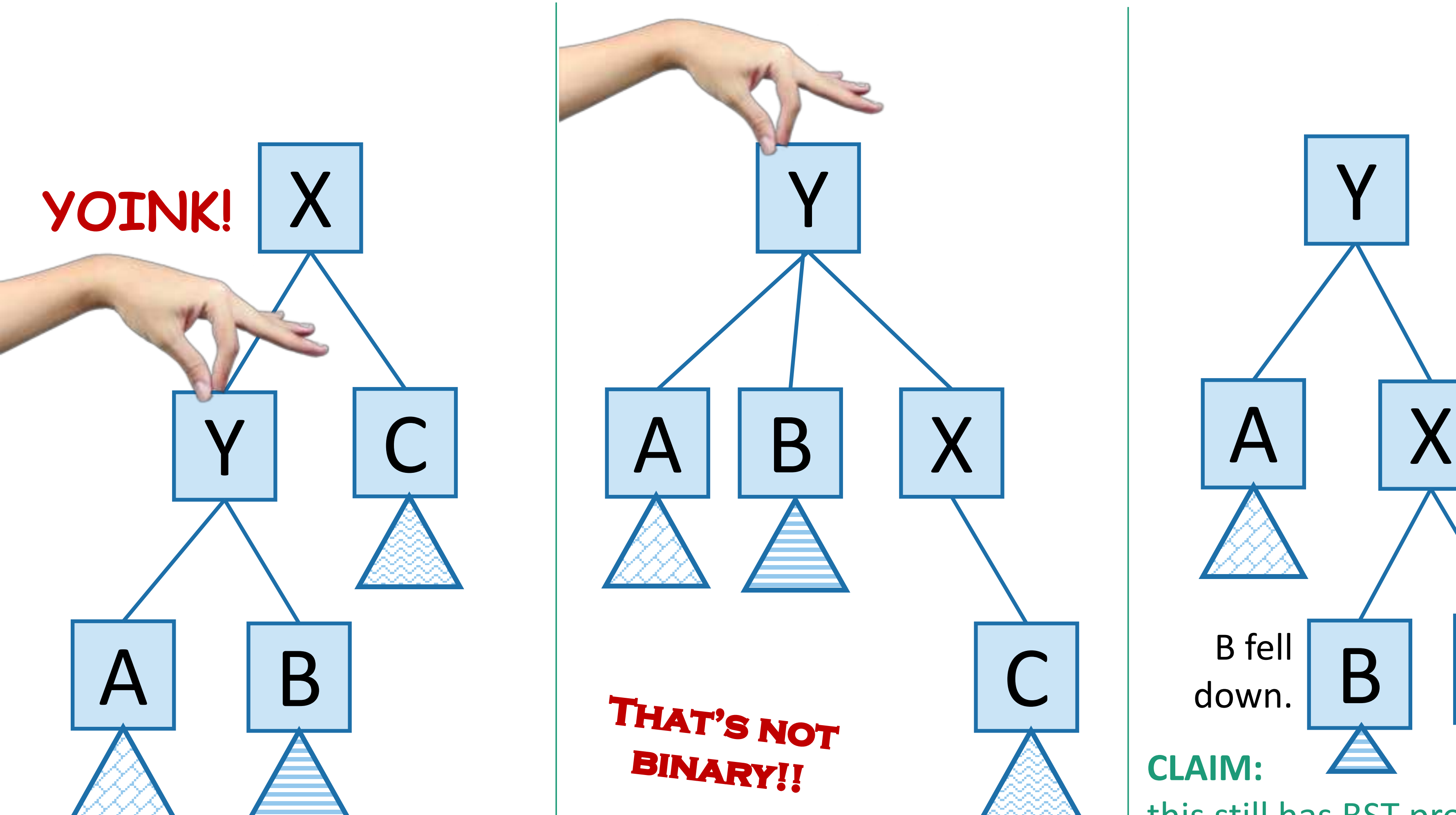
Self-Balancing Binary Search Trees

Idea 1: Rotations

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

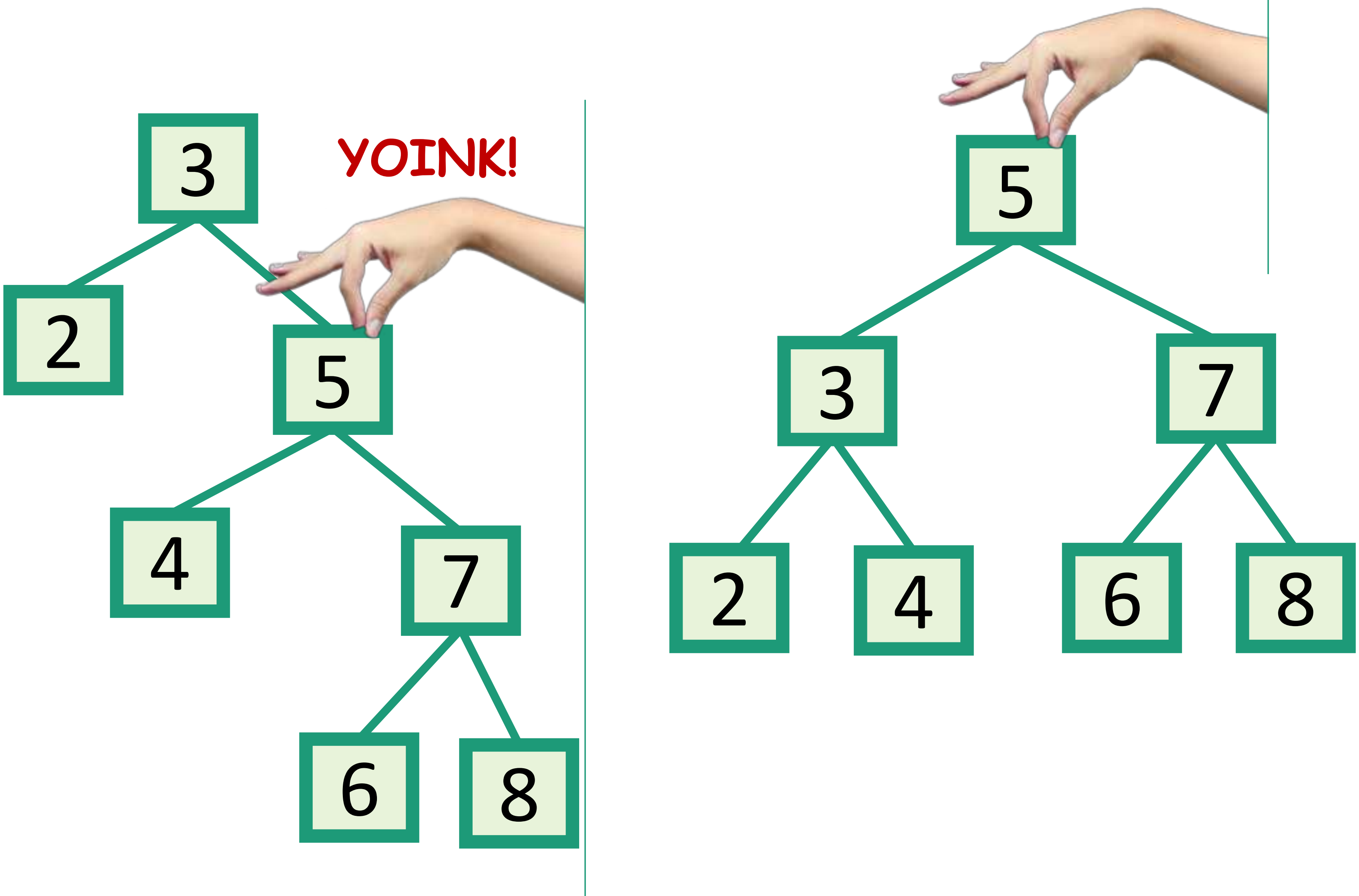
- Maintain Binary Search Tree (BST) property, while moving stuff around.

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



CLAIM:
this still has BST property.

This seems helpful



Strategy?

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

Lucky the Lackadaisical Lemur

Even for Lucky this is pretty vague. What do we mean by "seems unbalanced"? What's "okay"?

Idea 2: have some proxy for balance

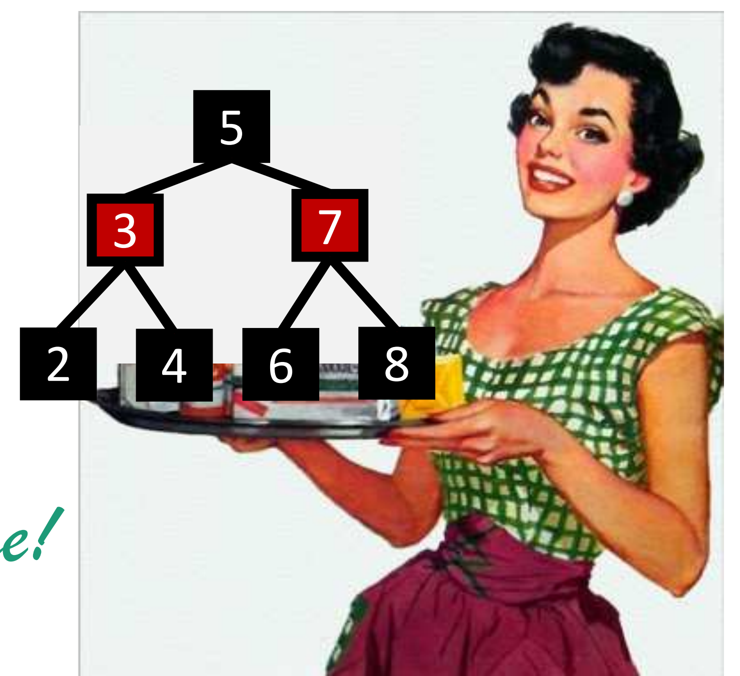
- Maintaining perfect balance is too hard.
- Instead, come up with some proxy for balance:
  - If the tree satisfies [SOME PROPERTY], then it's pretty balanced.
  - We can maintain [SOME PROPERTY] using rotations.

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

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!



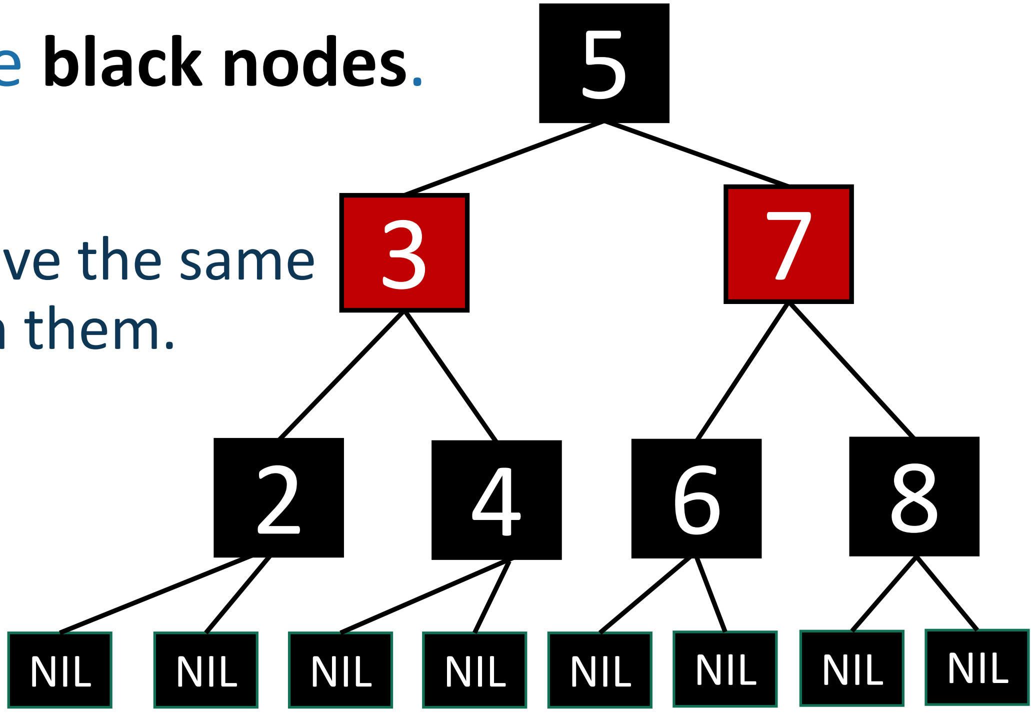
Maintain balance by stipulating that ***black nodes*** are balanced, and that there *aren't too many* **red nodes.**

It's just good sense!

Red-Black Trees

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

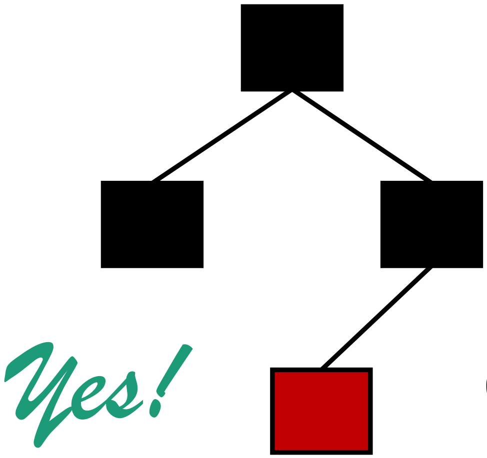
- Every node is colored **red** or **black.**
- The root node is a **black node**.
- NIL children count as **black nodes**.
- Children of a **red node** are **black nodes**.
- For all nodes x:
  - all paths from x to NIL's have the same number of **black nodes** on them.



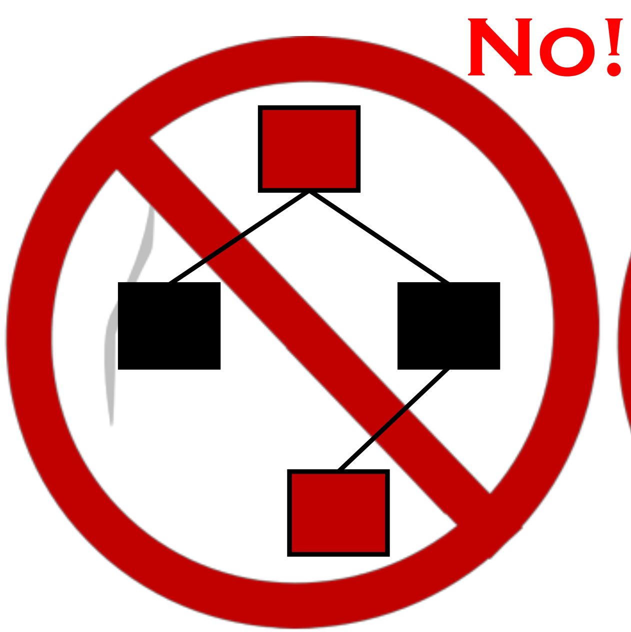
I'm not going to draw the NIL children in the future, but they are treated as black nodes.

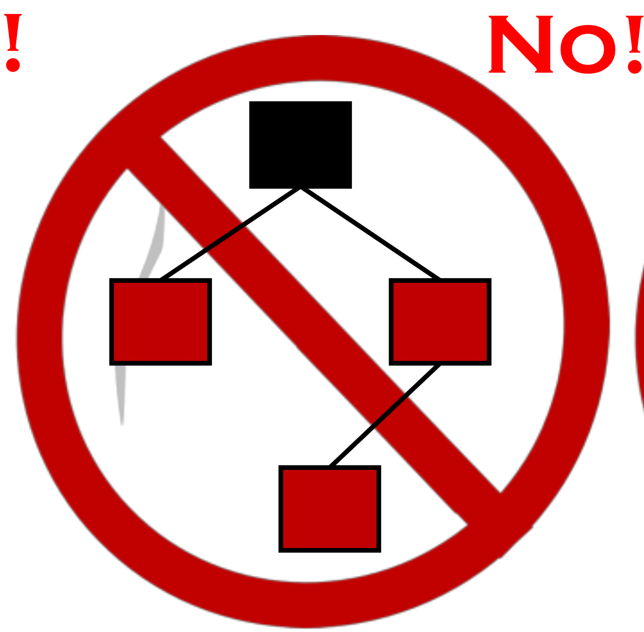
Examples(?)

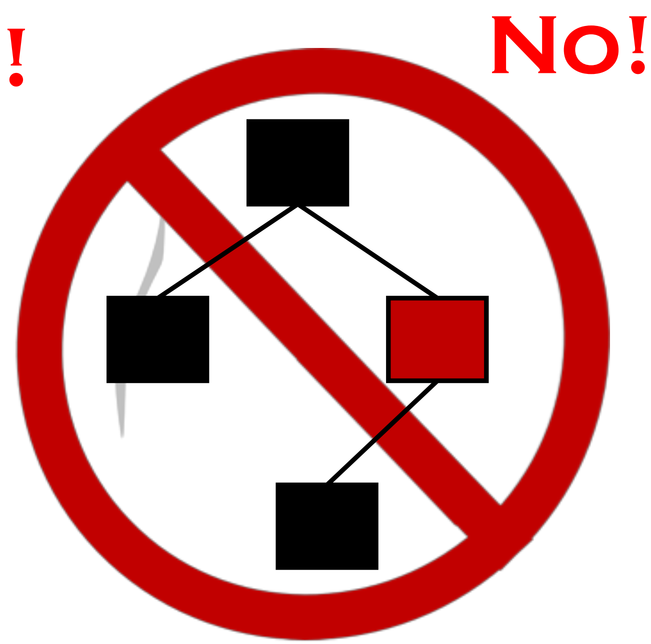
- Every node is colored **red** or **black.**
- The root node is a **black node**.
- NIL children count as **black nodes**.
- Children of a **red node** are **black nodes**.
- For all nodes x:
  - all paths from x to NIL's have the same number of **black nodes** on them.



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

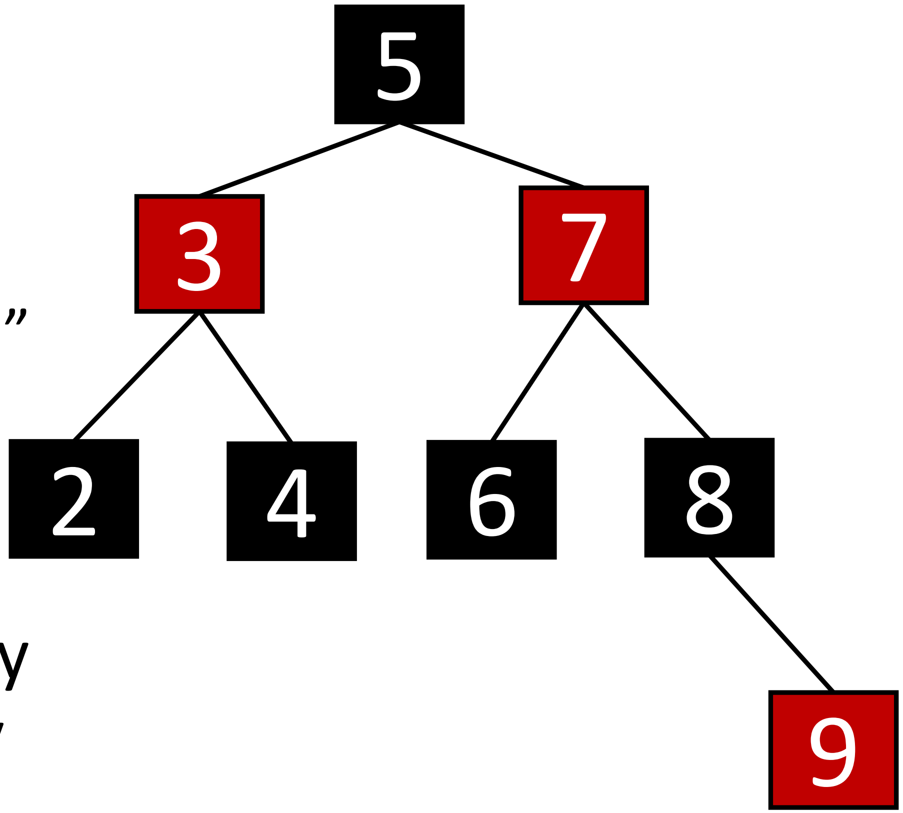






Why **these** rules???????

- 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.

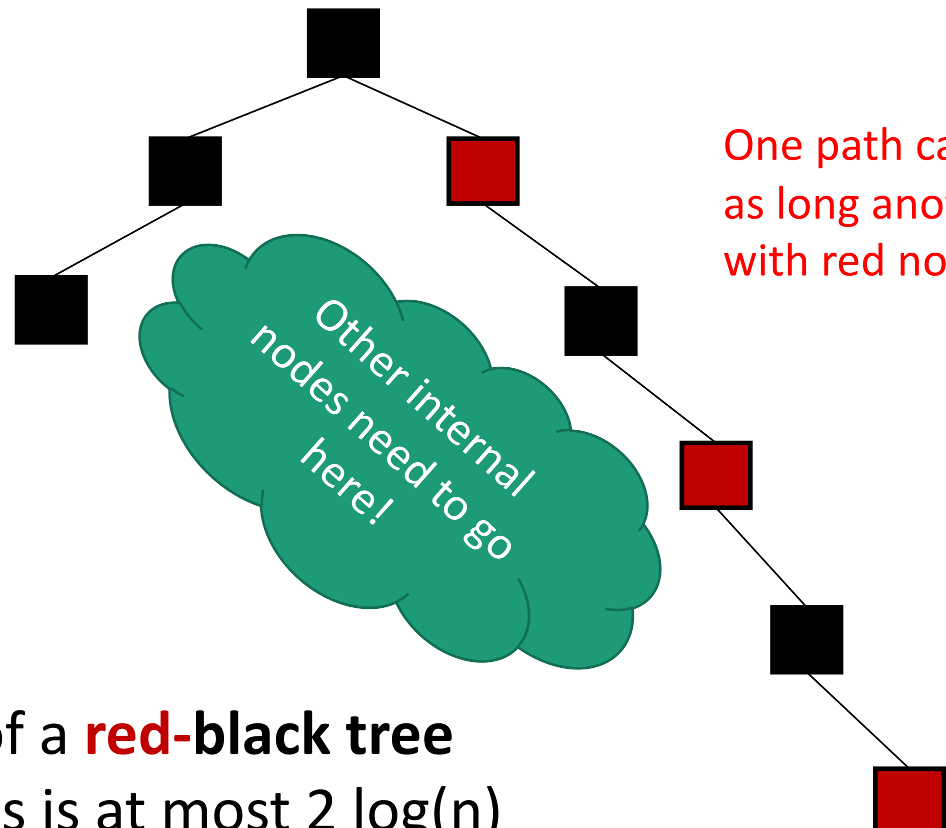
It's just a smidge weaker than perfect balance, but we can actually maintain it!

Let’s build some intuition!

Lucky the lackadaisical lemur

This is “pretty balanced”

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

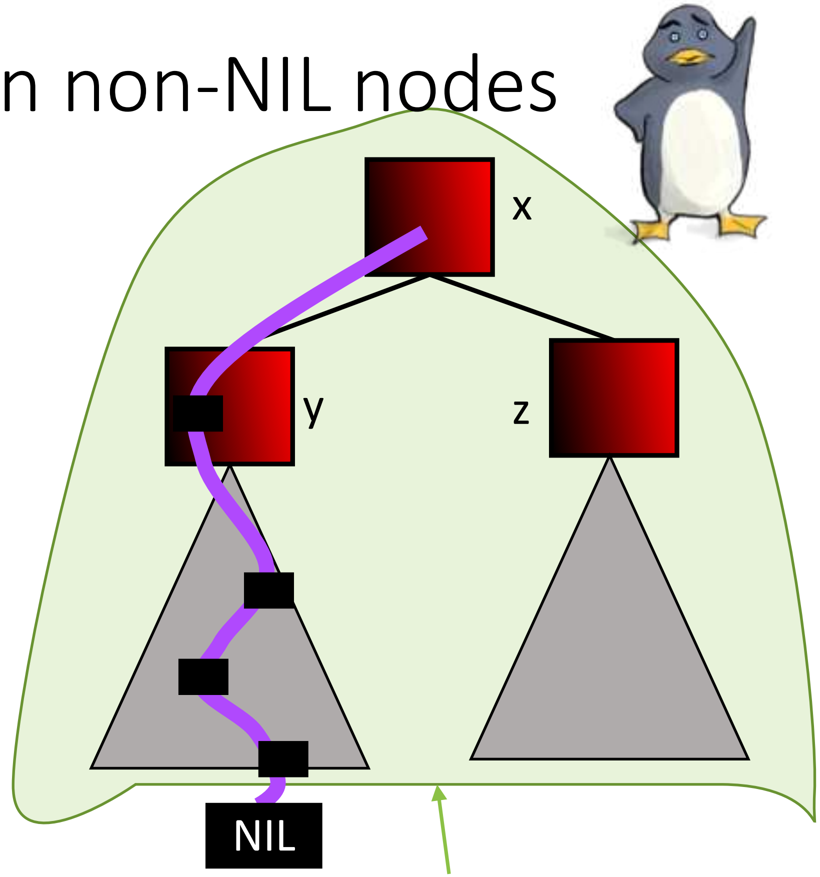


One path can be at most twice as long another if we pad it with red nodes.

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

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 – on board if time]



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

Then:

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

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

NOTE: I goofed up badly in lecture on this slide – the green blob was all wrong and I didn't notice for way too long. Sorry!

Rearranging:

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

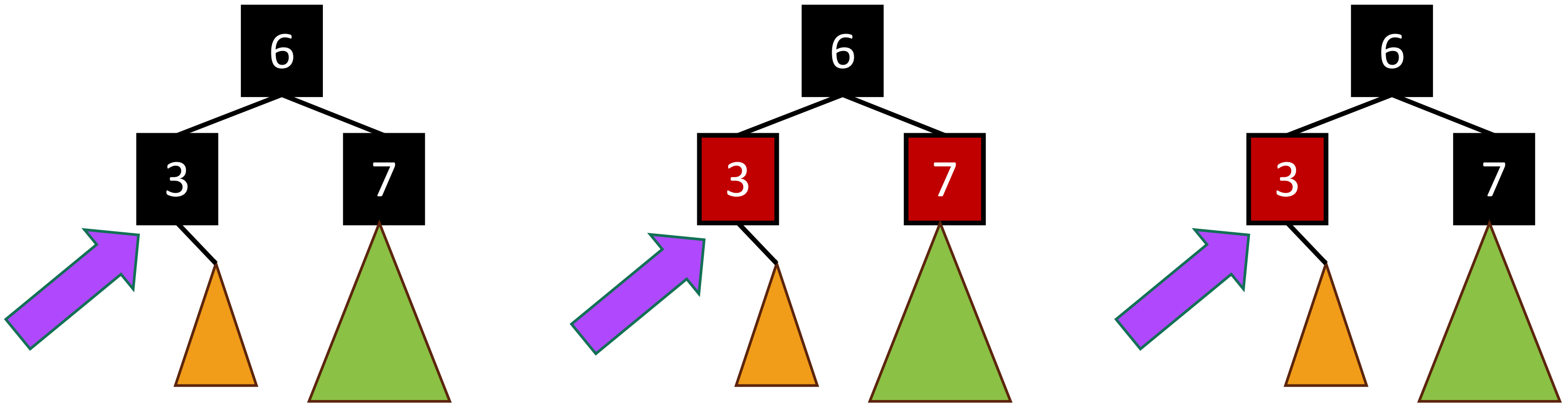
This is great!

- 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.

INSERT/DELETE

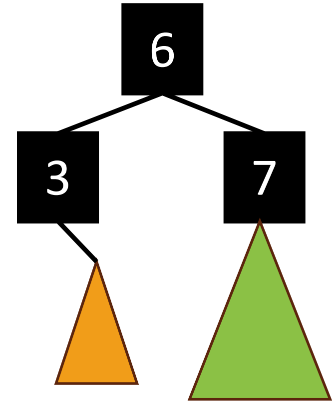
- For the rest of lecture [if time], we'll sketch how to do INSERT/DELETE for RBTrees.
  - See CLRS for more details if you are interested.
- You are **not responsible** for the details of INSERT/DELETE for RBTrees for this class.
  - You should know what the "proxy for balance" property is and why it 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



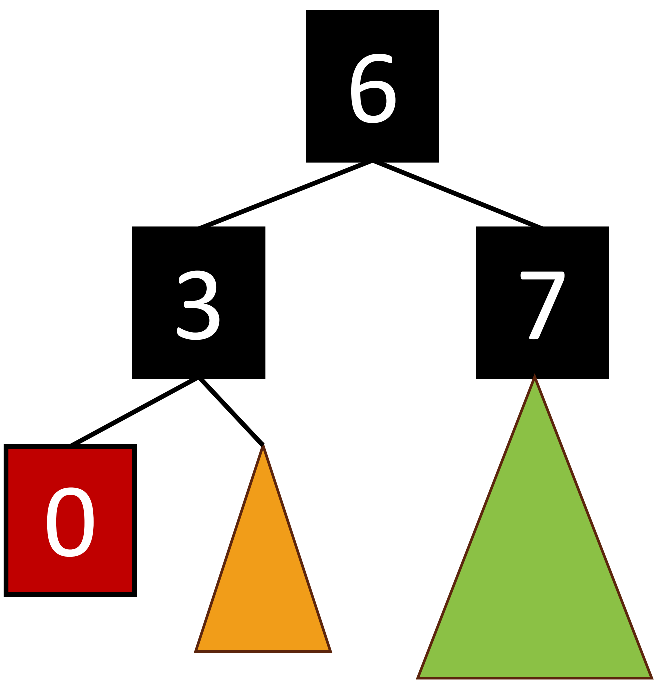
- Suppose we want to insert 0 **here**.
- There are 3 “important” cases for different colorings of the existing tree, and there are 9 more cases for all of the various symmetries of these 3 cases.

INSERT: Case 1



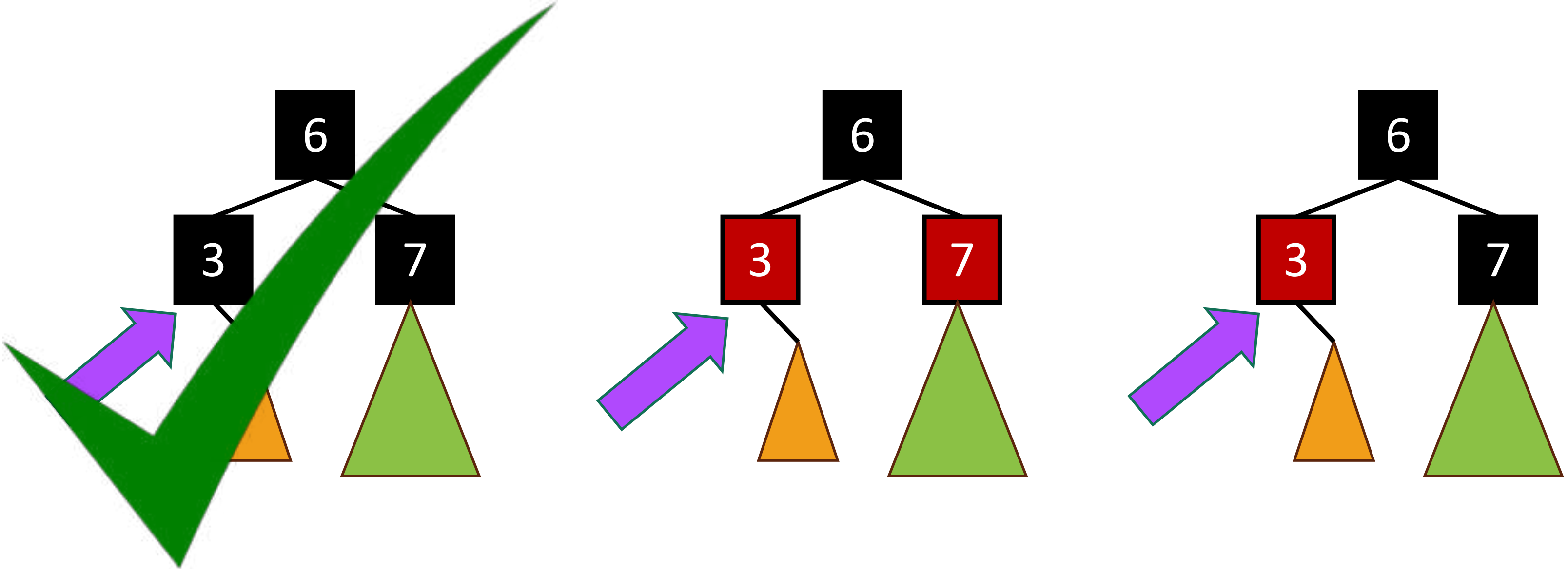
What if it looks like this?

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



Example: insert 0

INSERT: Many cases

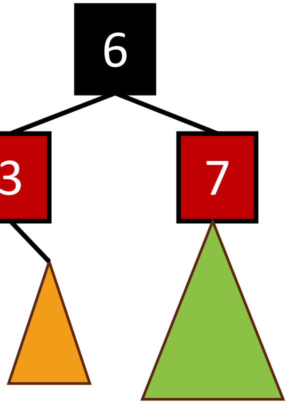


- Suppose we want to insert 0 **here**.

- 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.

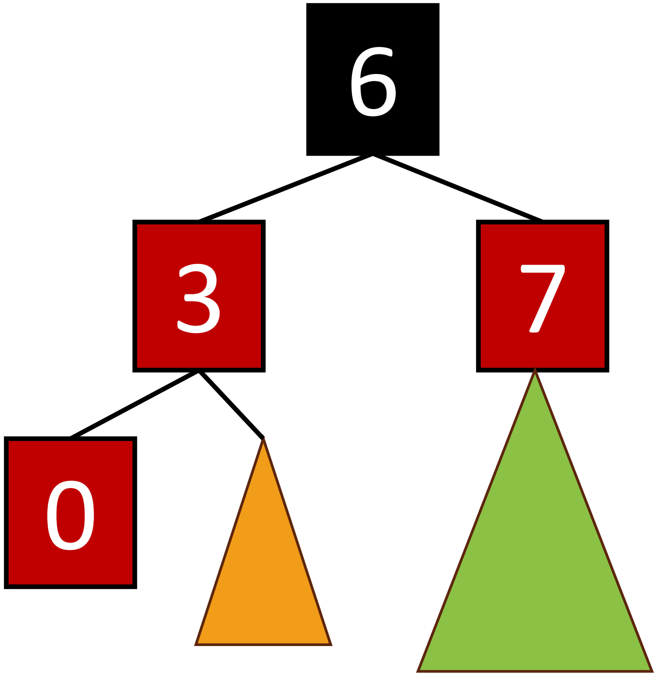
INSERT: Case 2

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

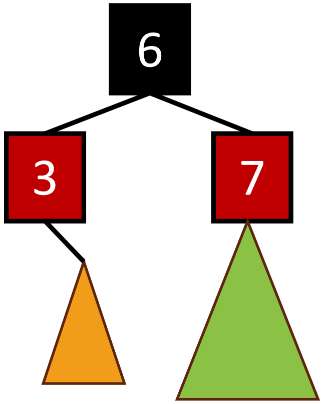


What if it looks like this?

Example: insert 0

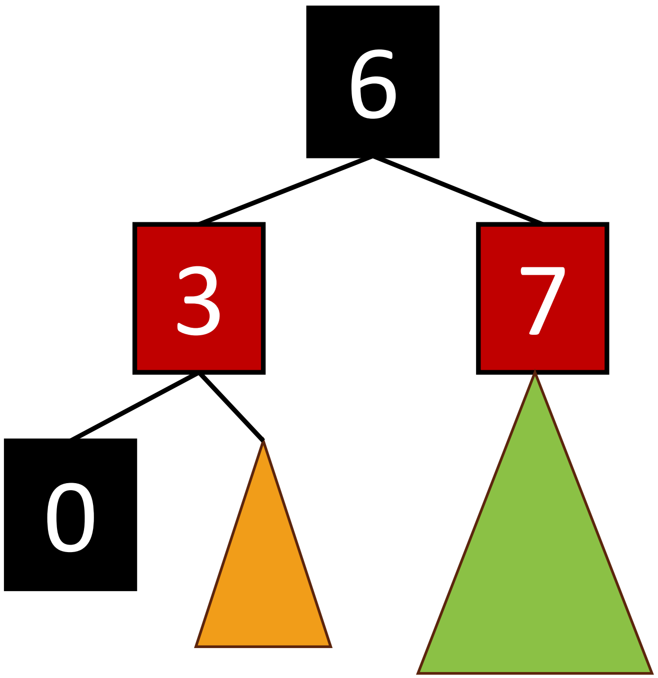


INSERT: Case 2



What if it looks like this?

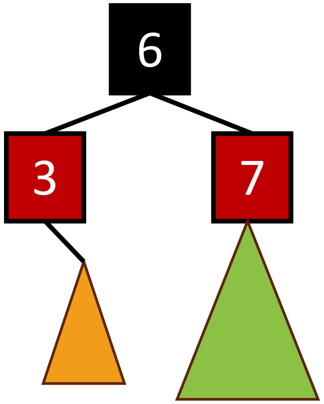
- Make a new **red node.**
- Insert it as you would normally.
- Fix things up if needed.



Example: insert 0

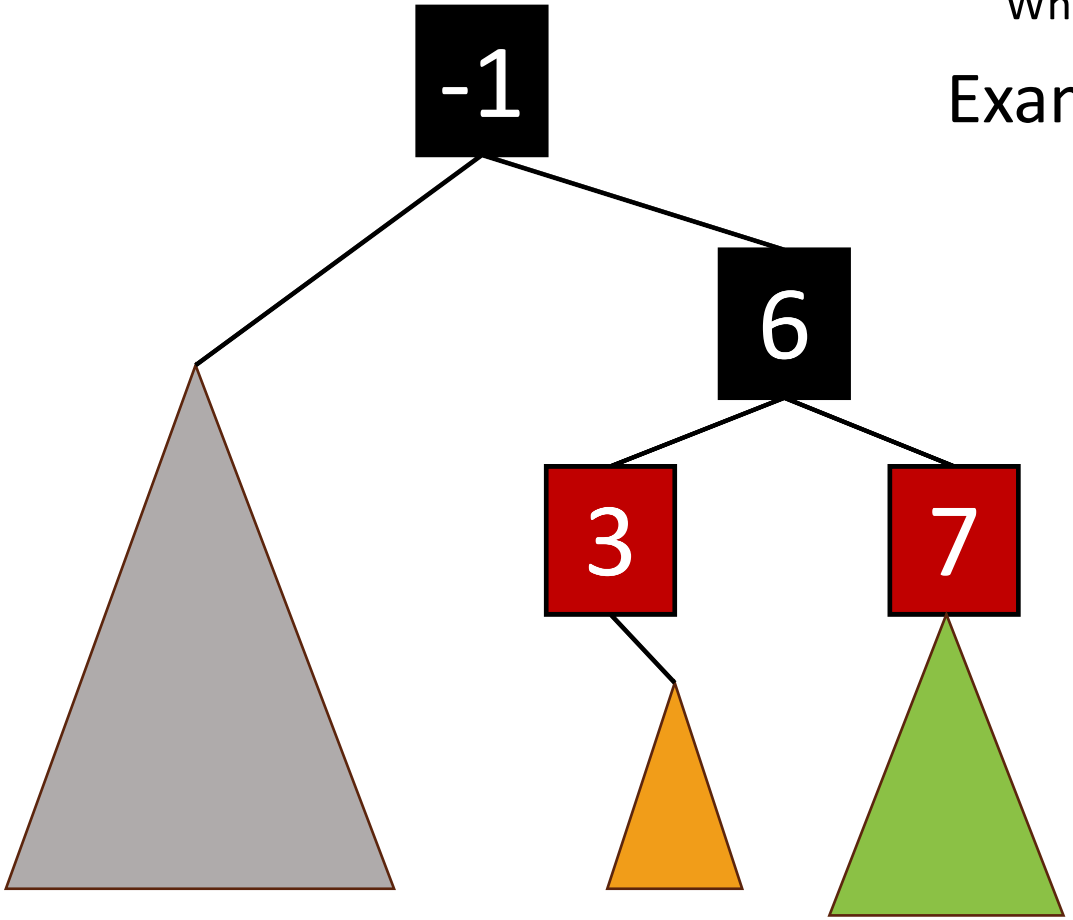
Can’t we just insert 0 as a **black node?**

We need a bit more context



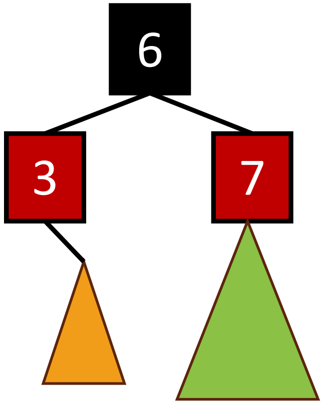
What if it looks like this?

Example: insert 0



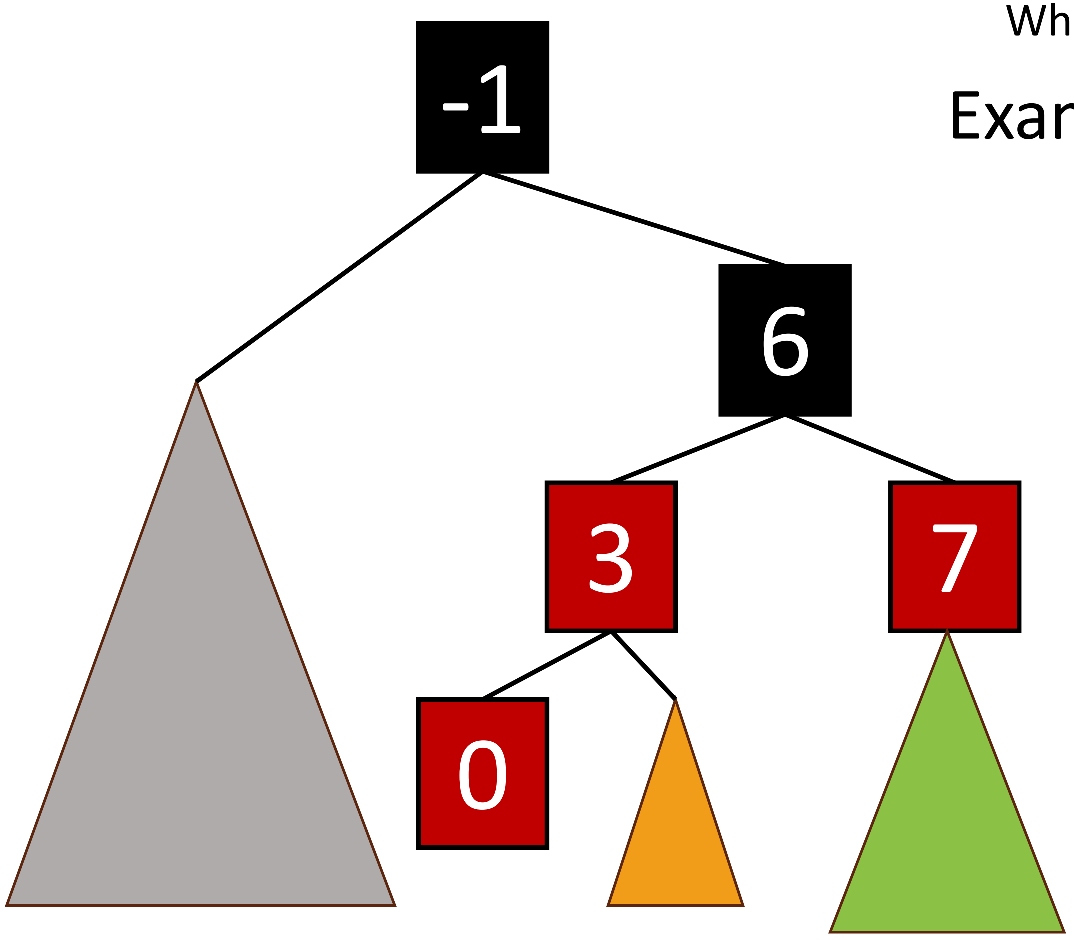
We need a bit more context

- Add 0 as a red node.



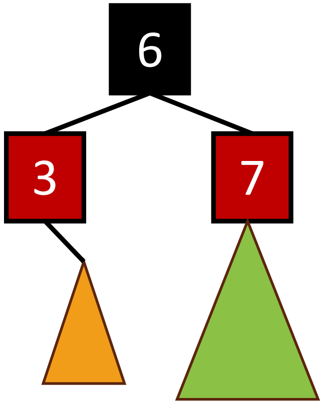
What if it looks like this?

Example: insert 0



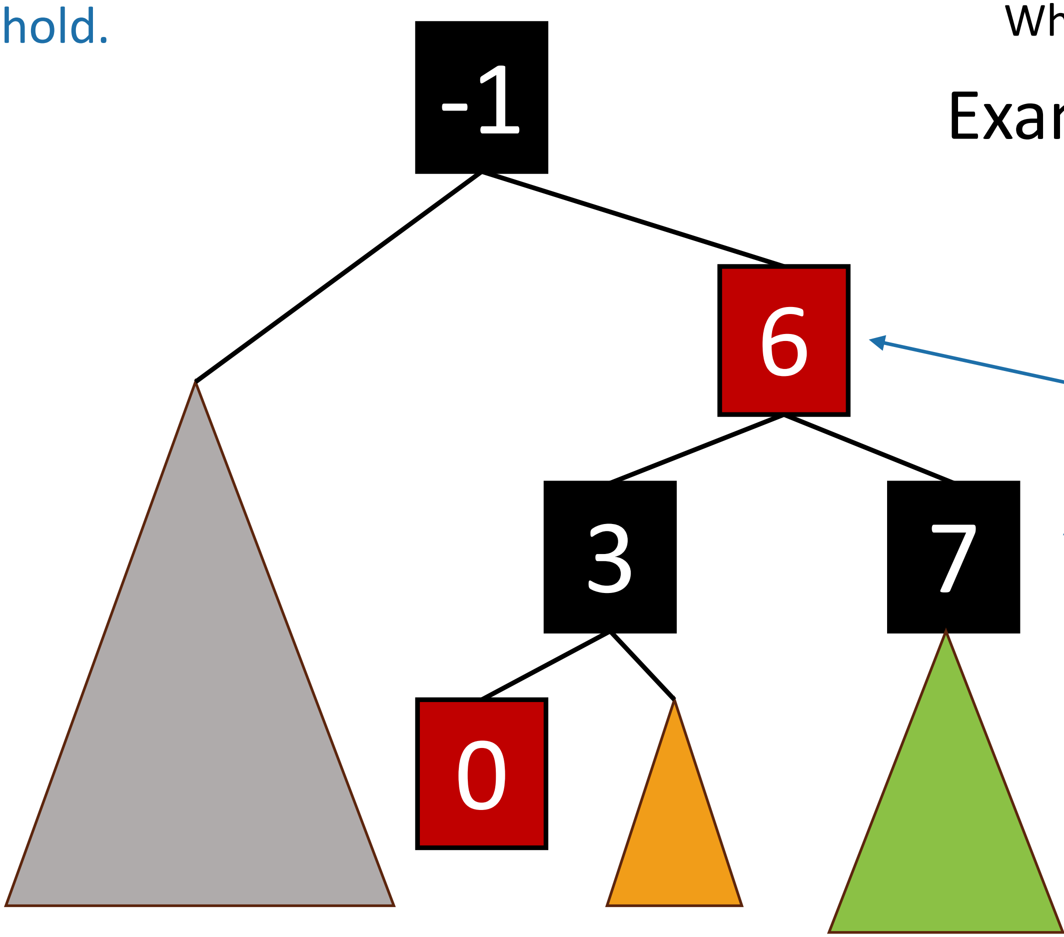
We need a bit more context

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



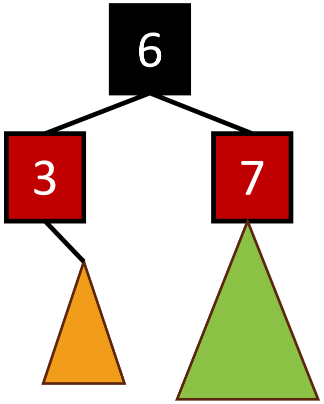
What if it looks like this?

Example: insert 0



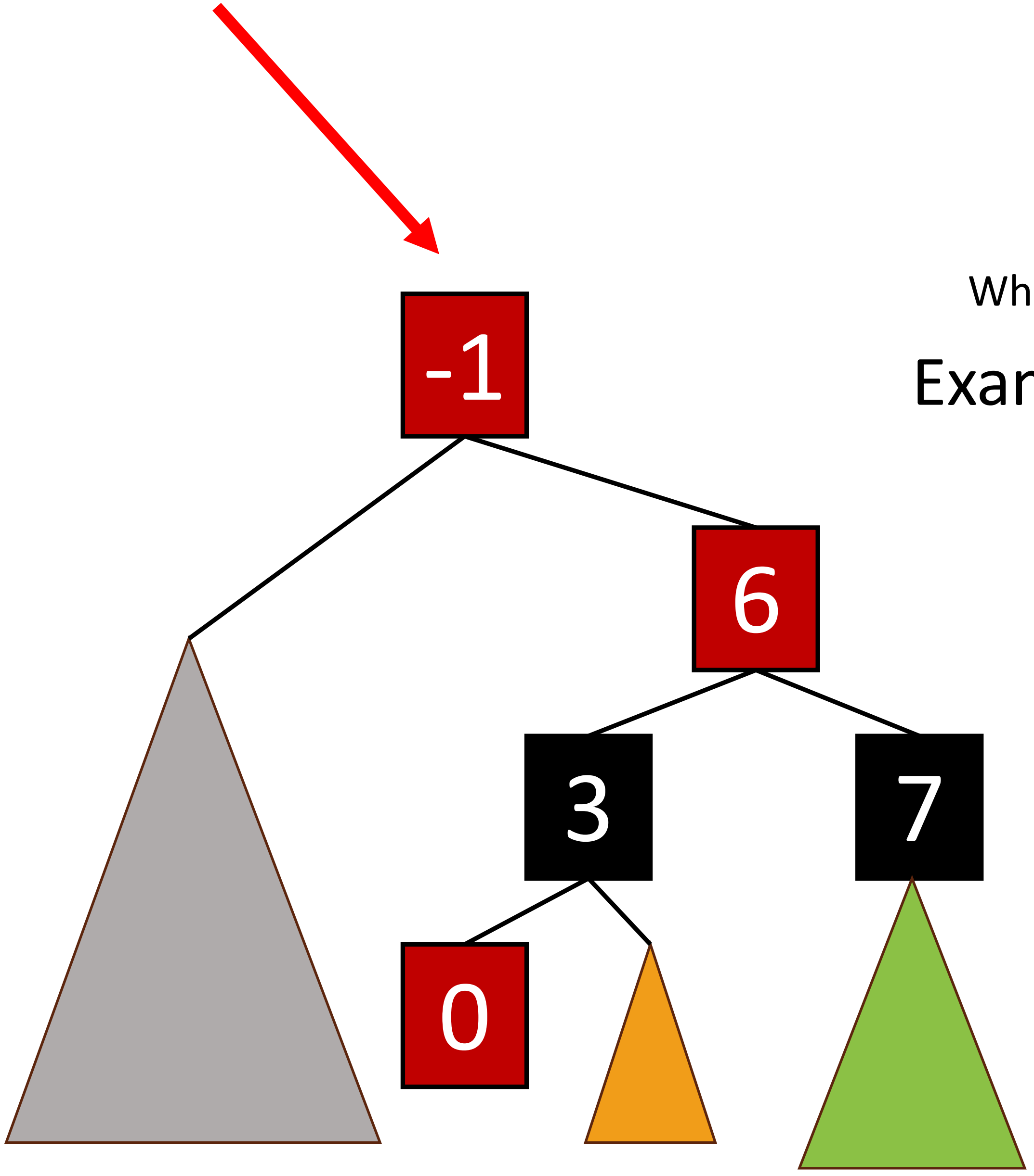
Flip colors!

But what if that was red?

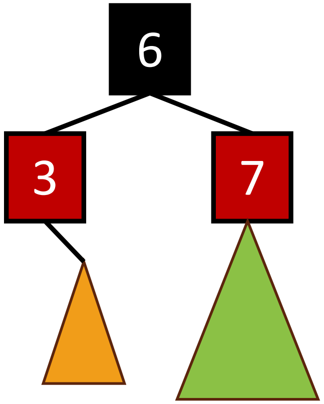


What if it looks like this?

Example: insert 0

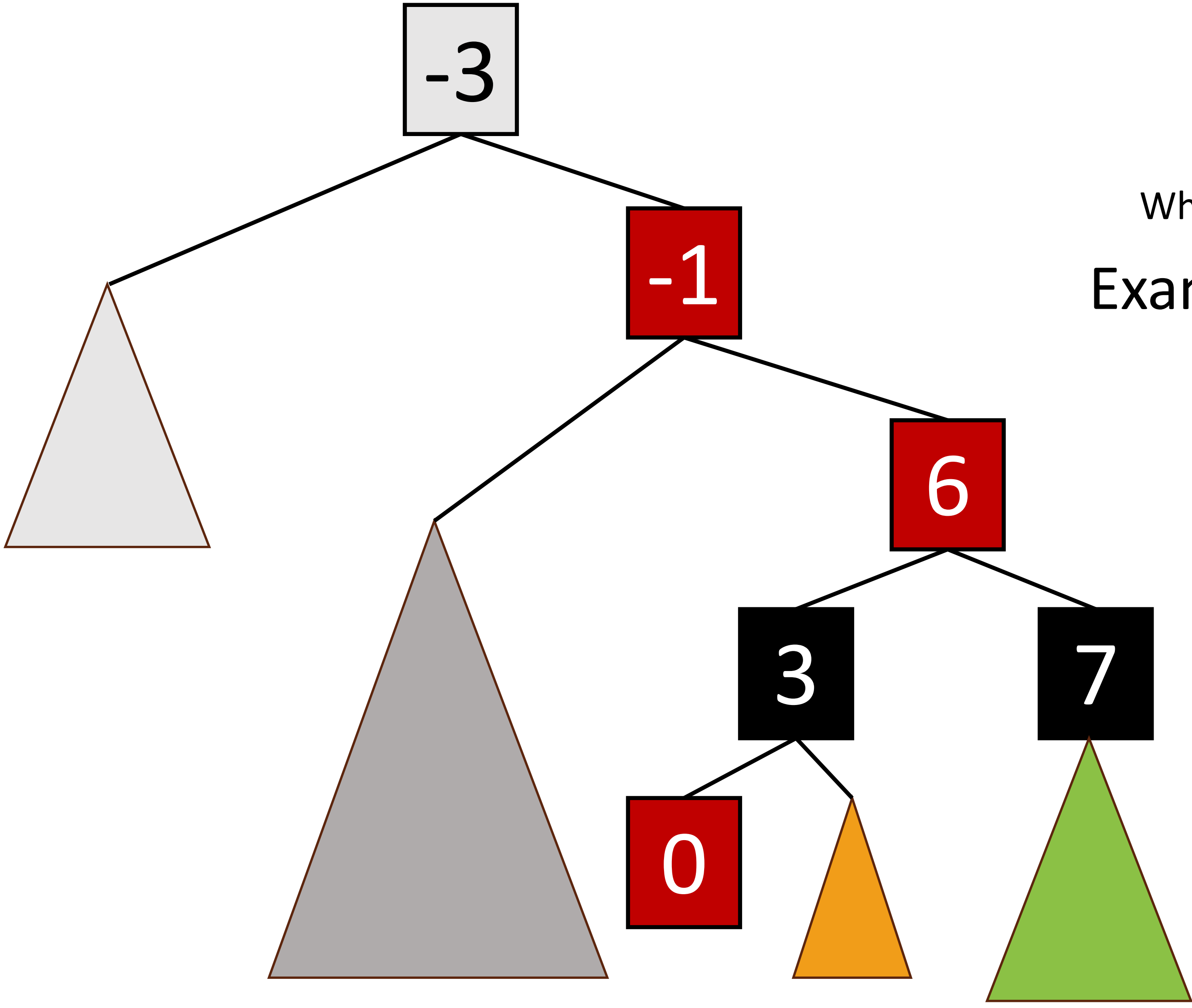


More context…

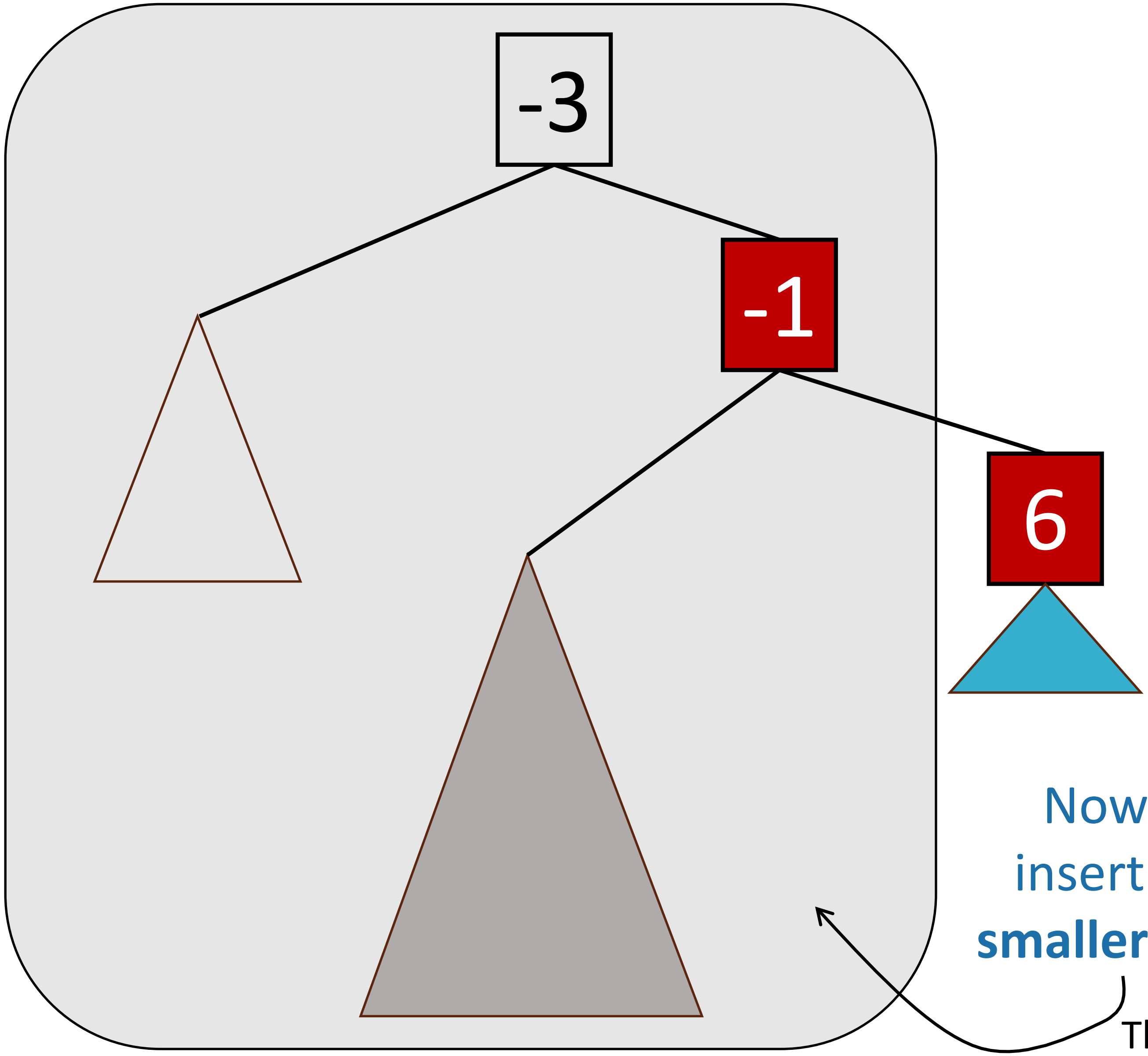


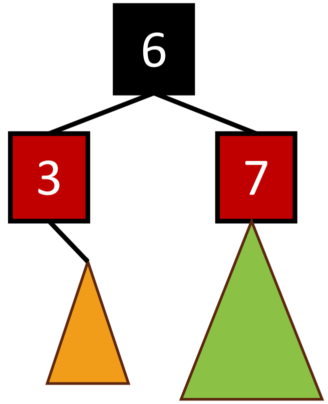
What if it looks like this?

Example: insert 0



More context…





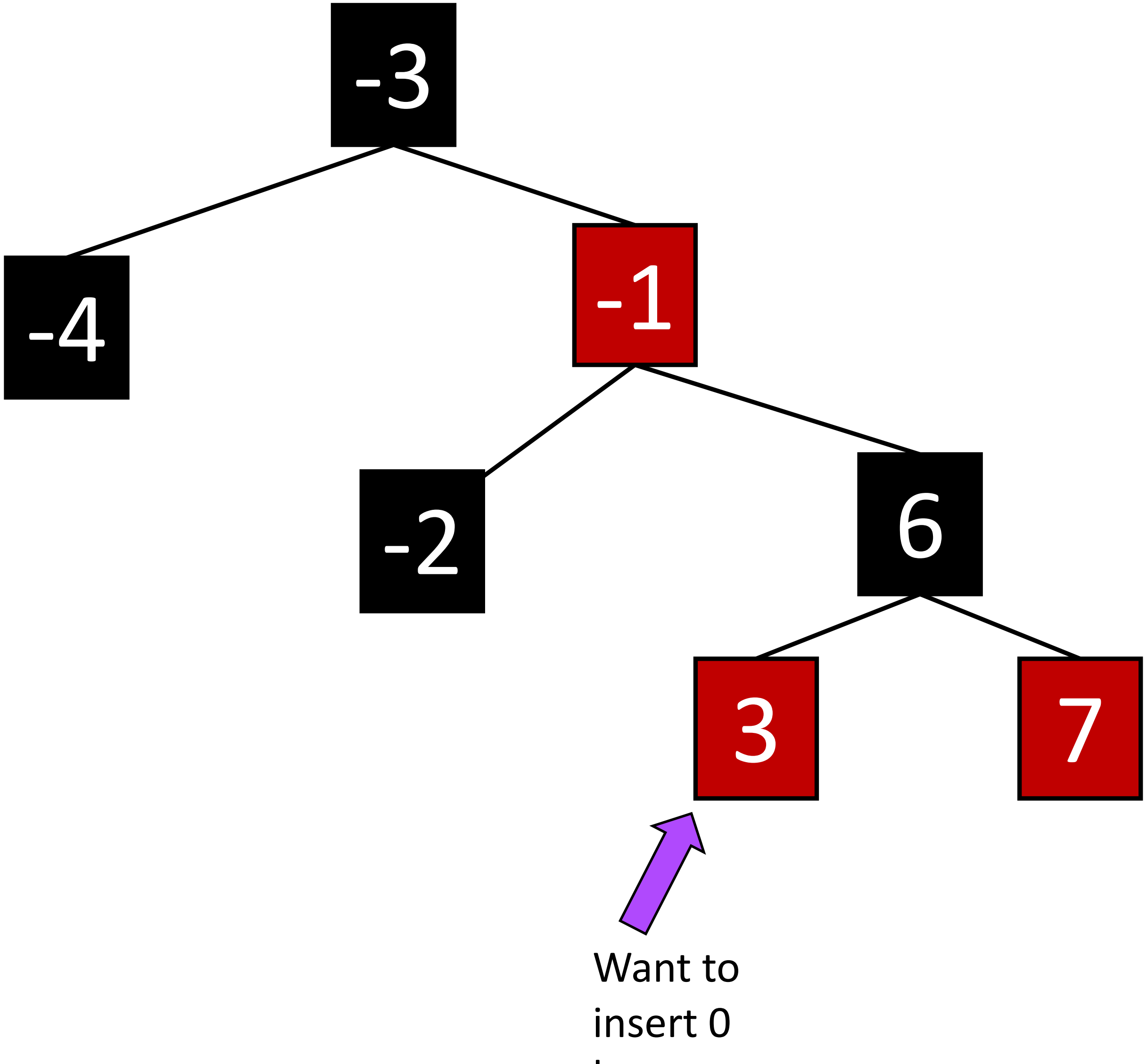
What if it looks like this?

Example: insert 0

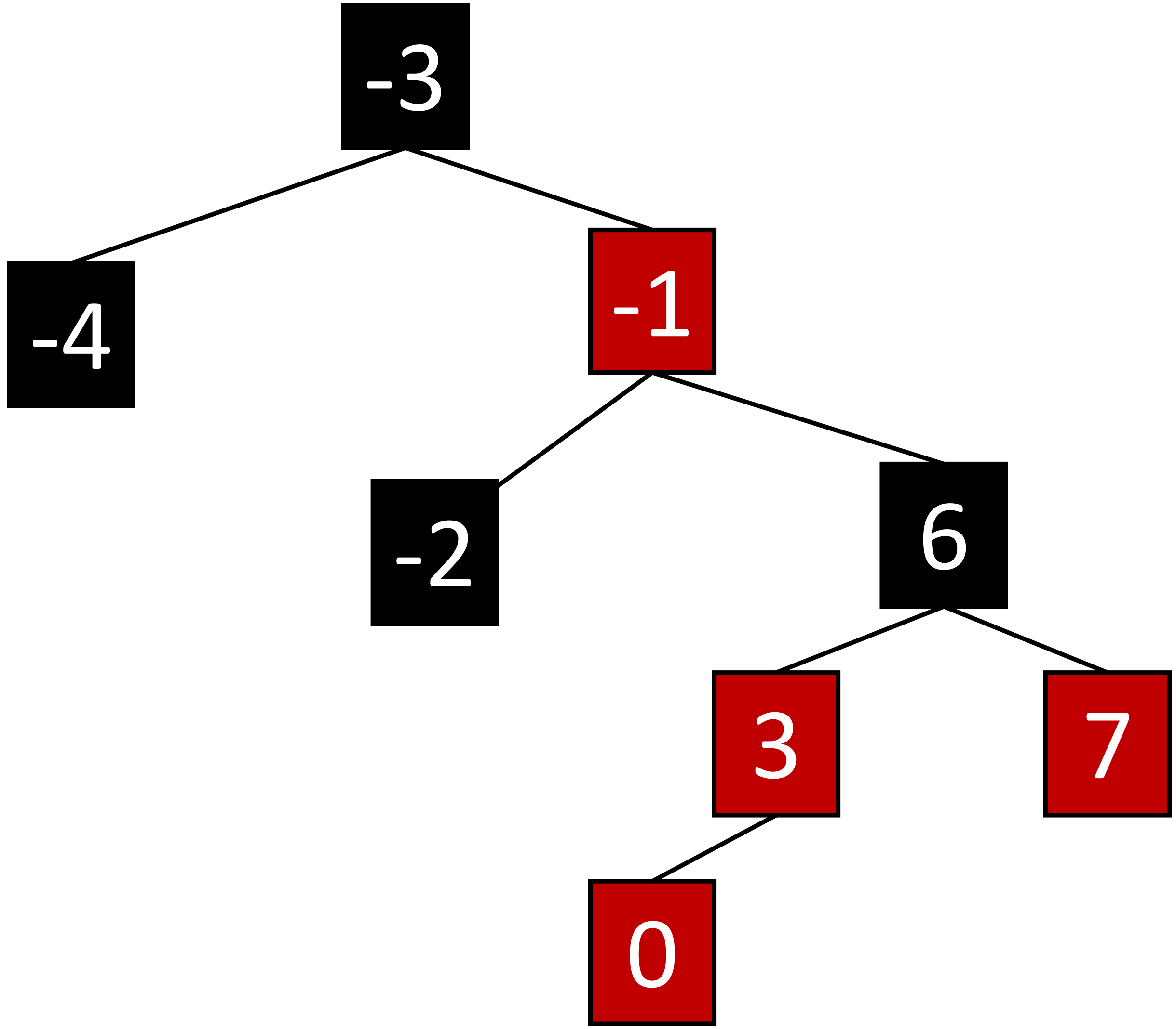
Now we're basically inserting 6 into some **smaller tree**. Recurse!

This one!

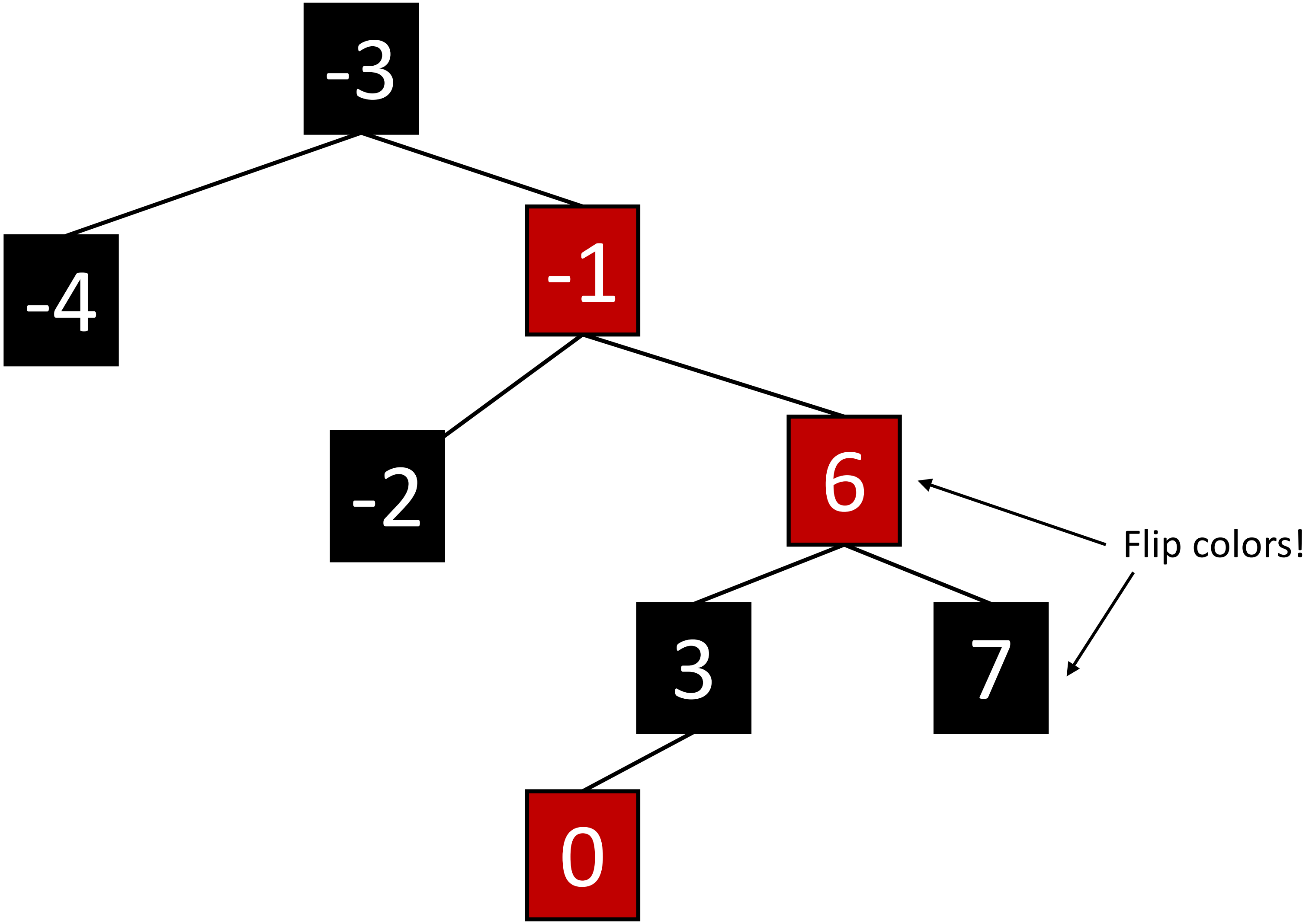
Example, part I



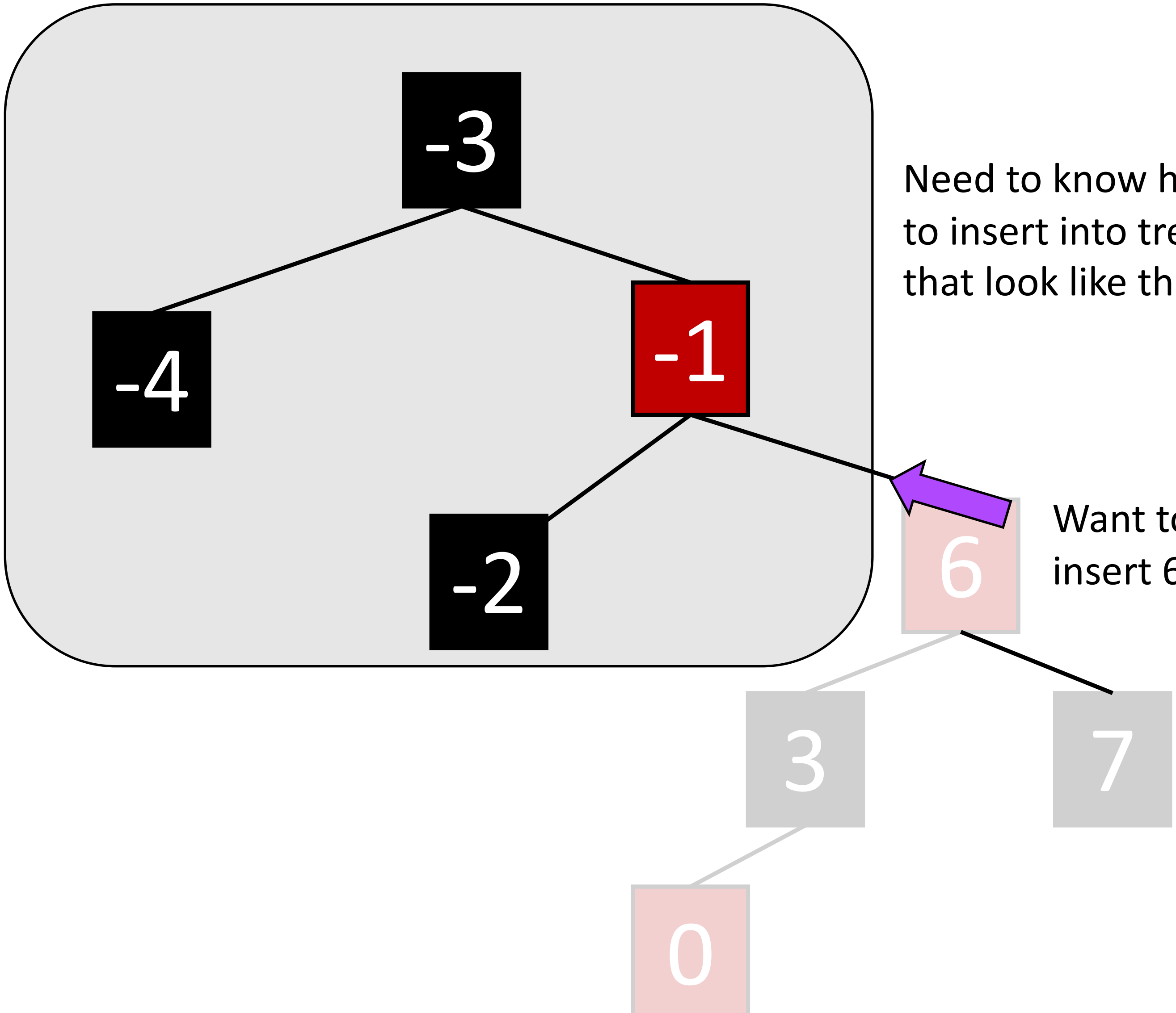
Example, part I



Example, part I



Example, part I

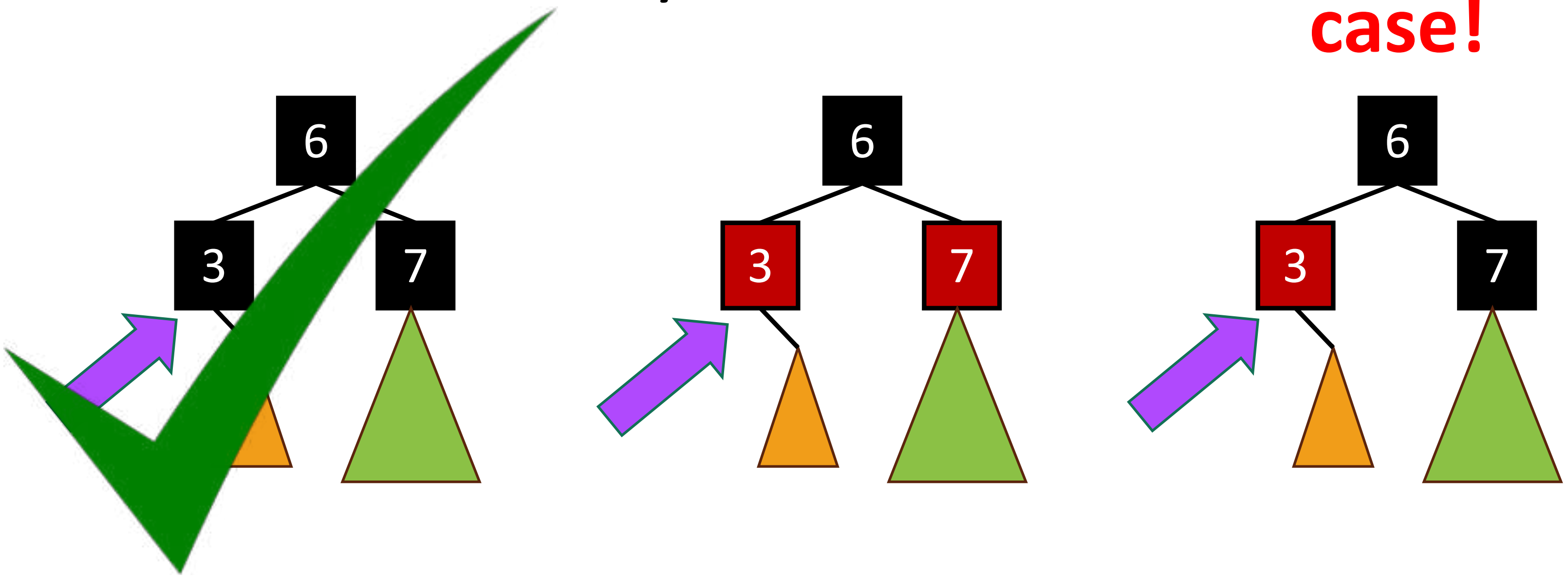


Need to know how to insert into trees that look like this…

Want to insert 6 here.

INSERT: Many cases

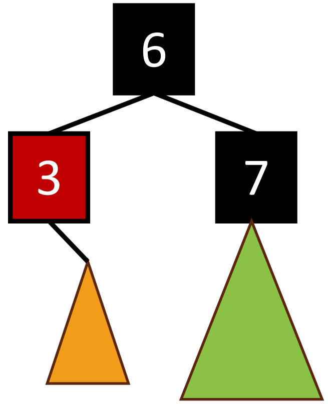
That's this case!



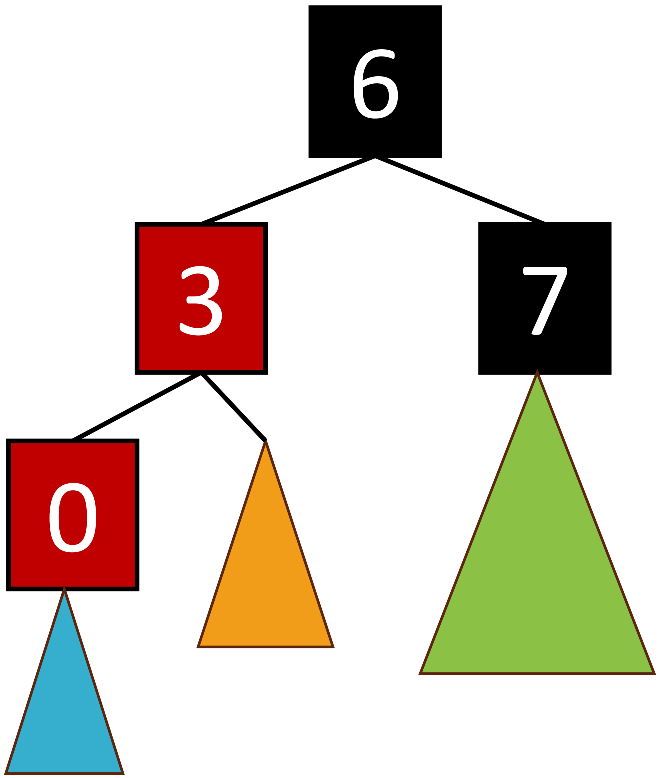
- Suppose we want to insert 0 **here**.
- There are 3 "important" cases for different colorings of the existing tree, and there are 9 more cases for all of the various symmetries of these 3 cases.

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?

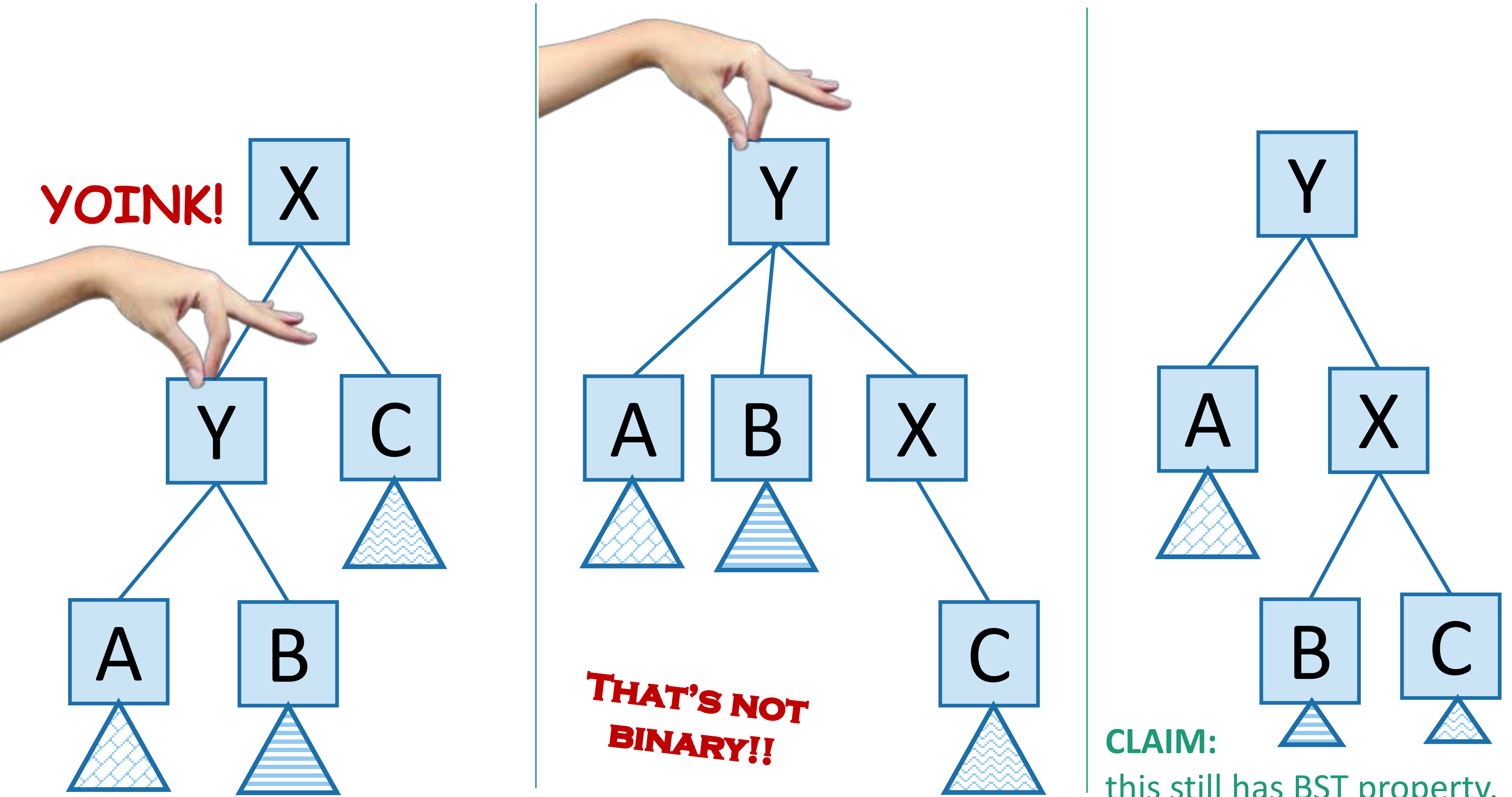


Example: Insert 0.

- Maybe with a subtree below it.

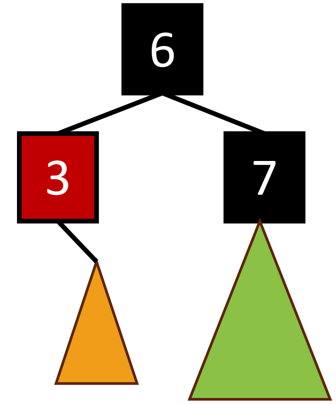
Recall Rotations

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

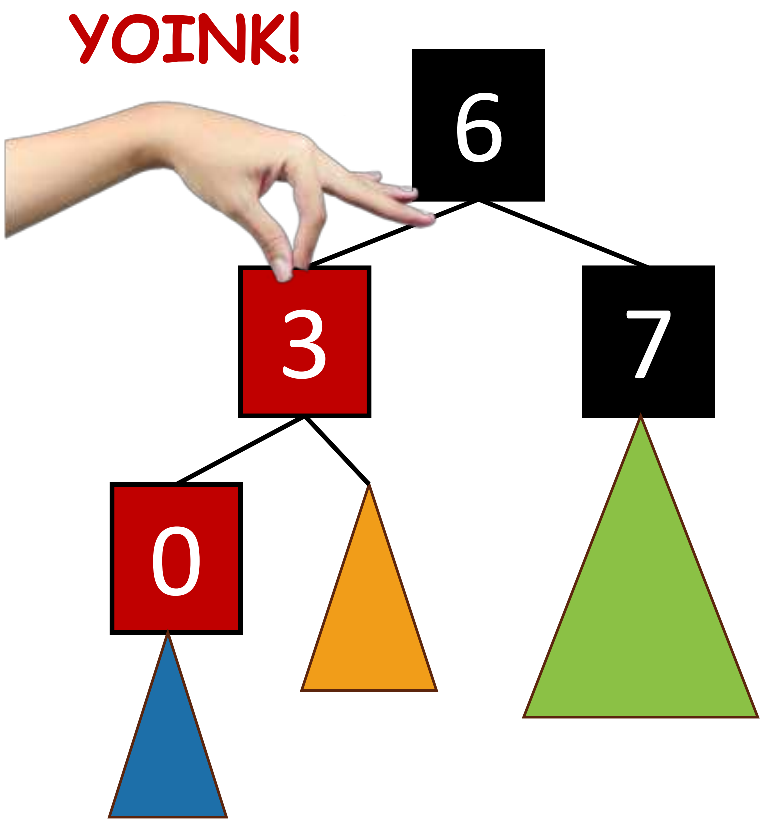


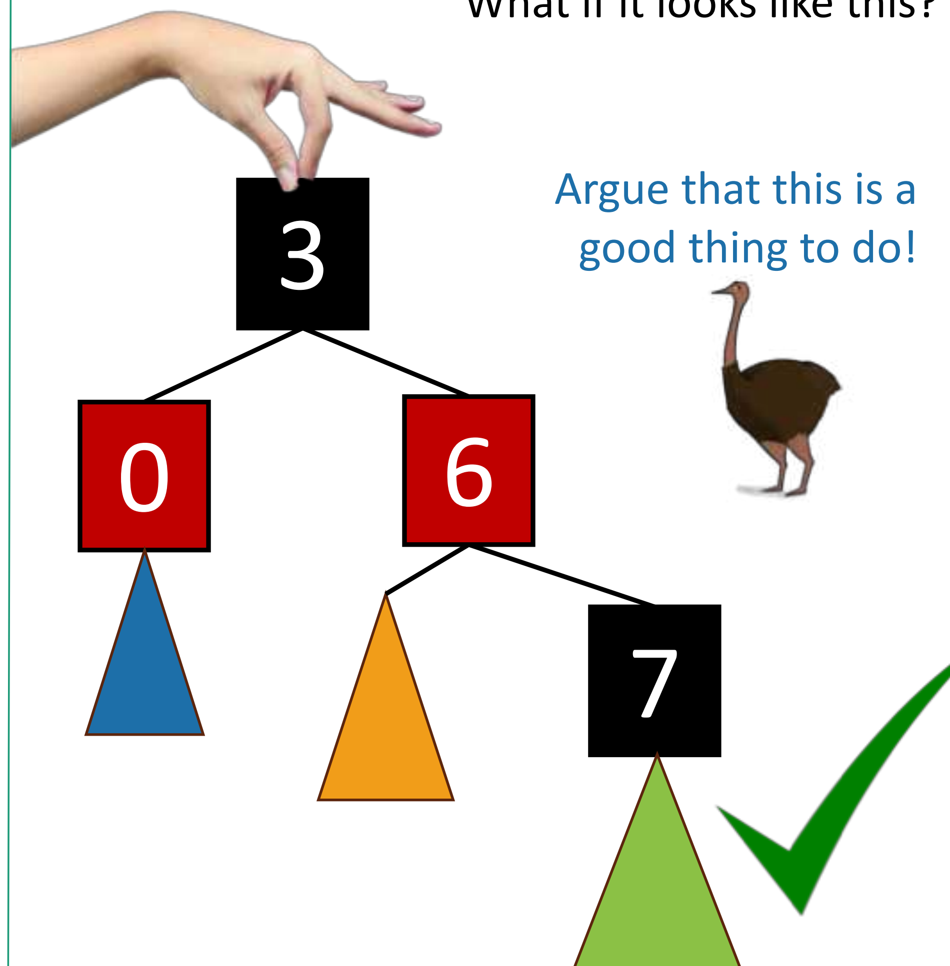
Inserting into a Red-Black Tree

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



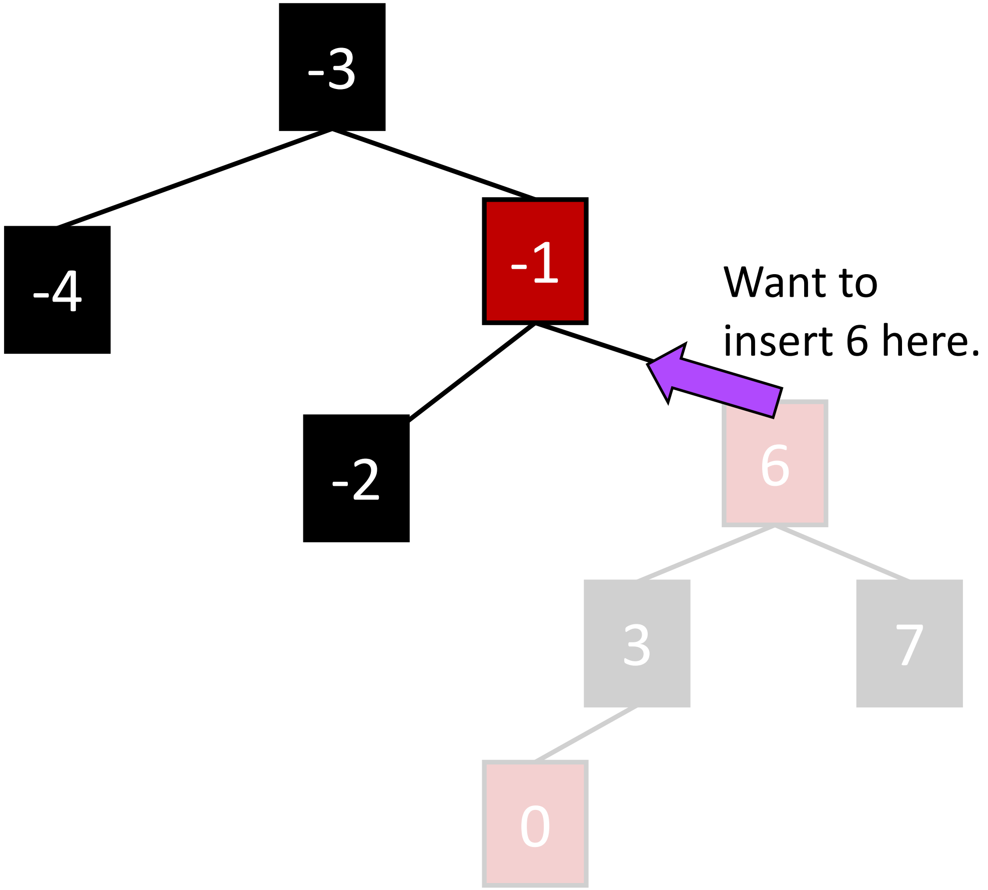
What if it looks like this?



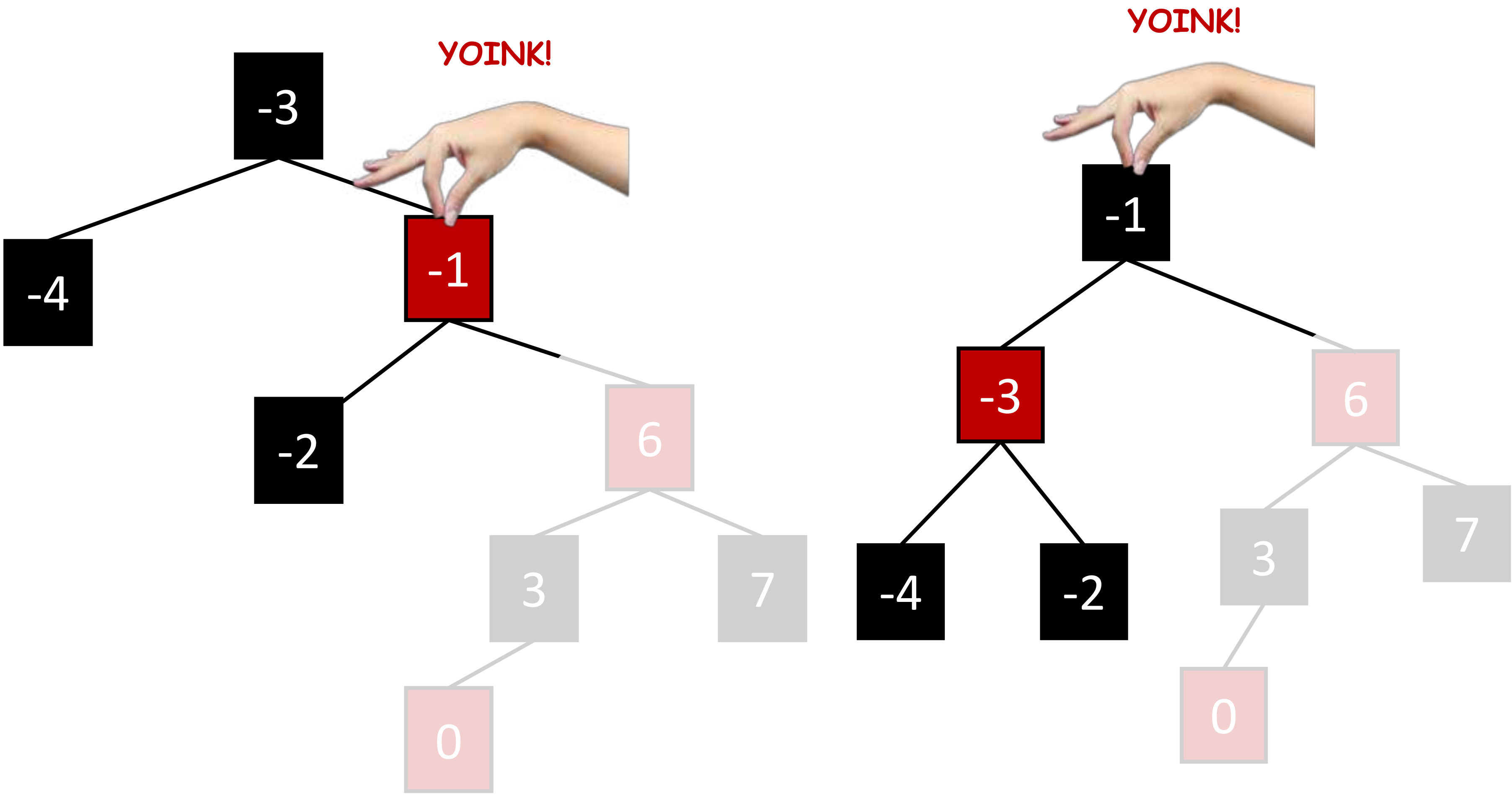


Argue that this is a good thing to do!

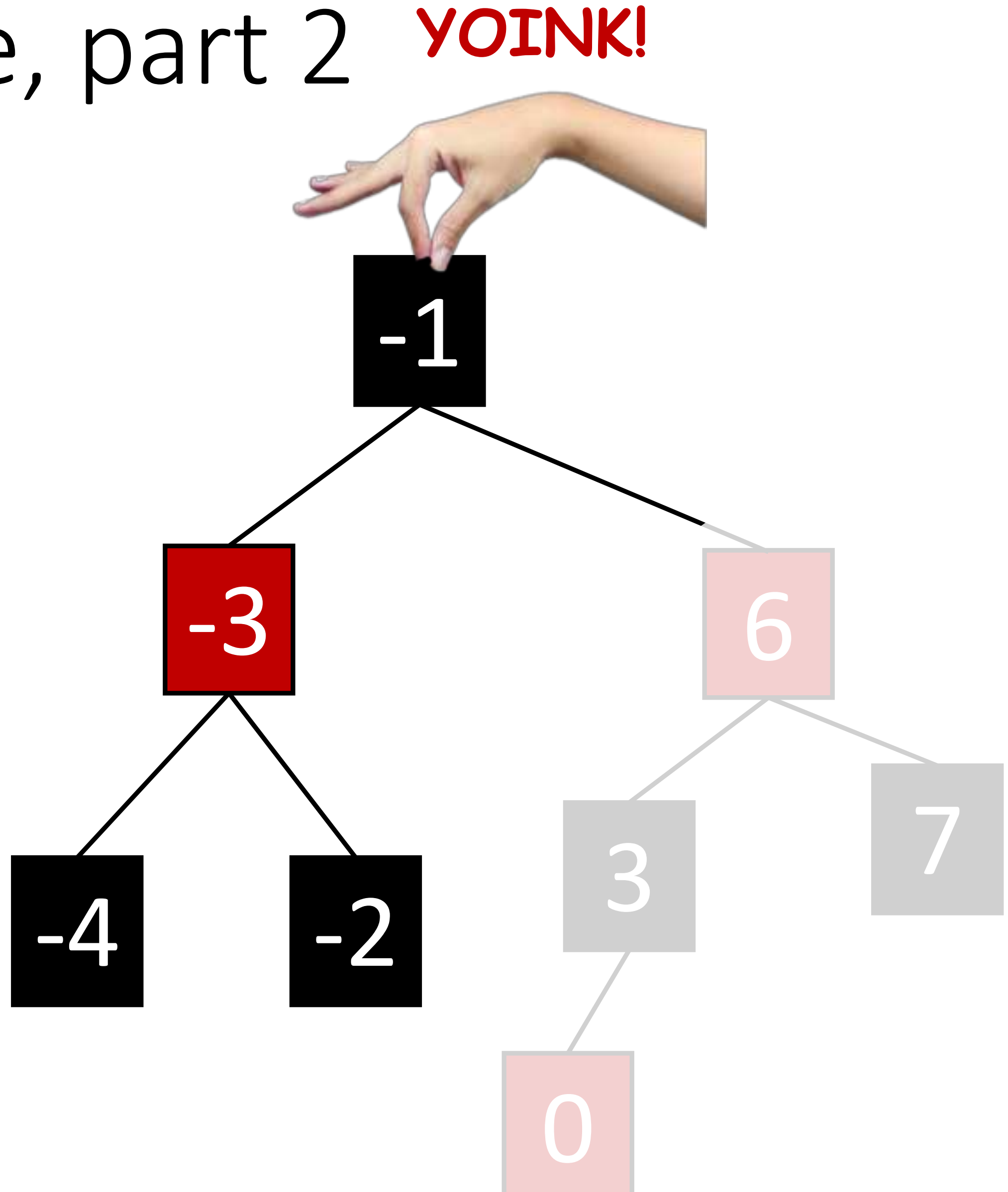
Example, part 2



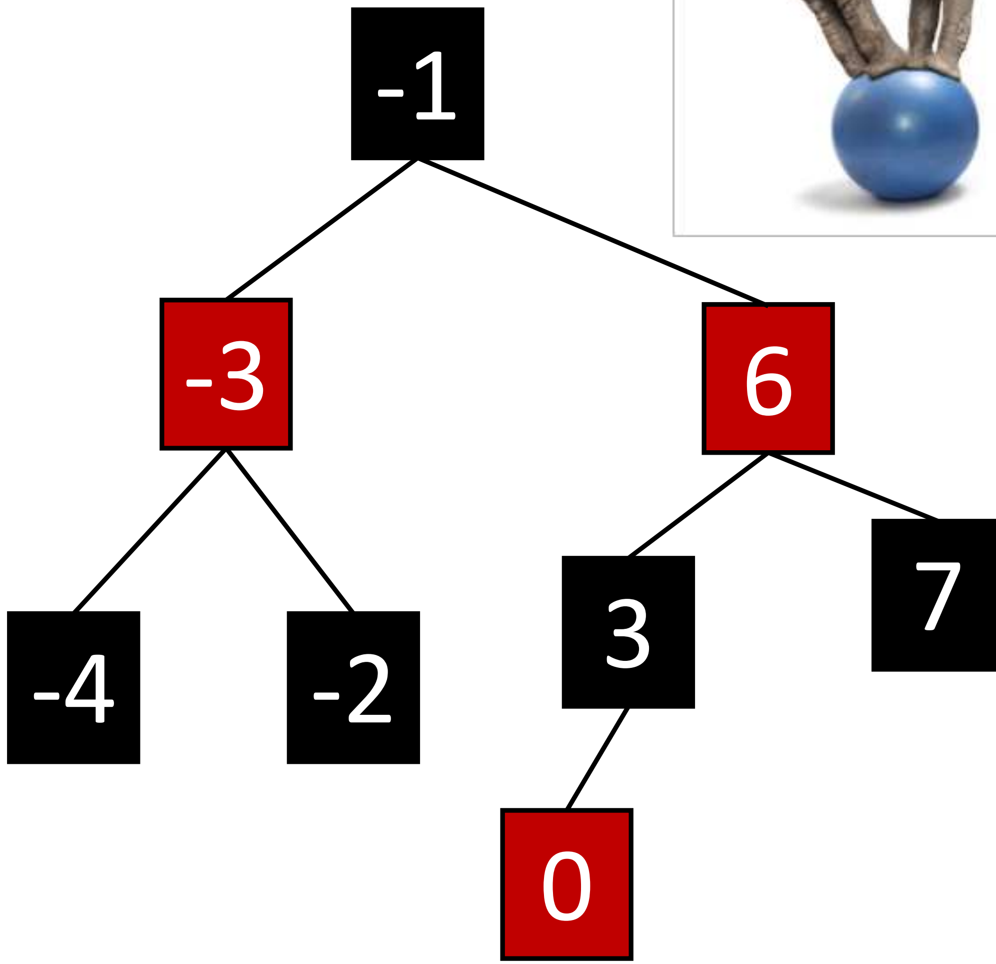
Example, part 2



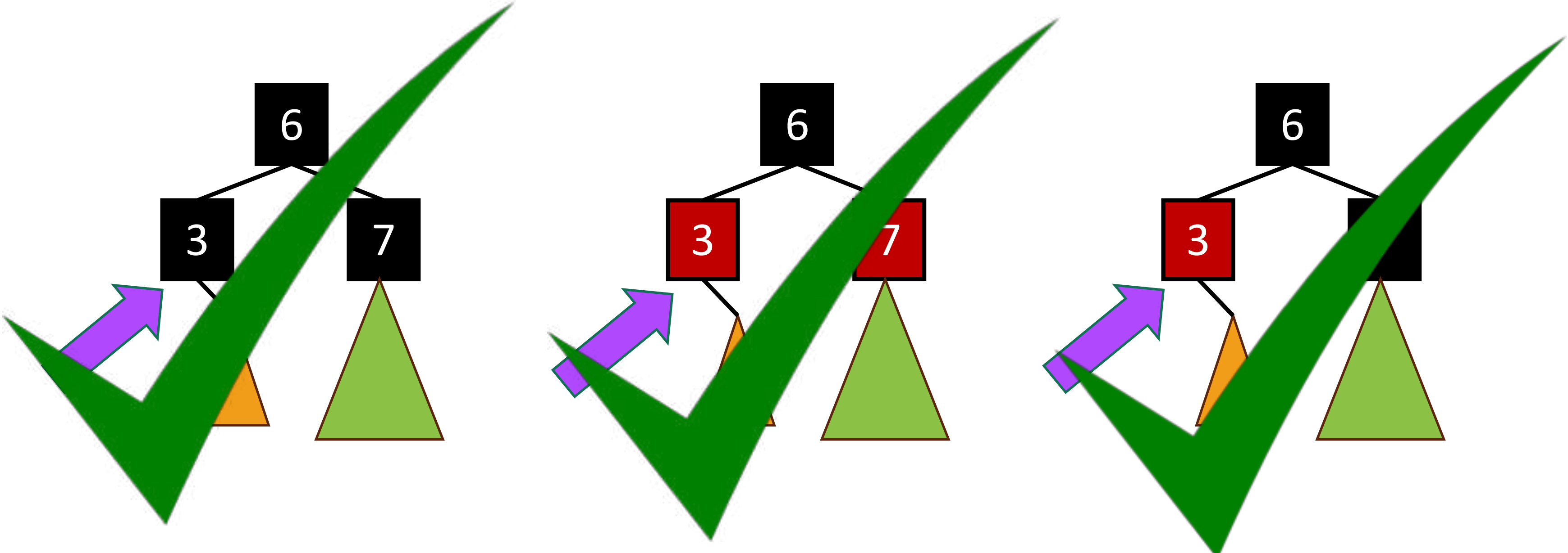
Example, part 2



Example, part 2



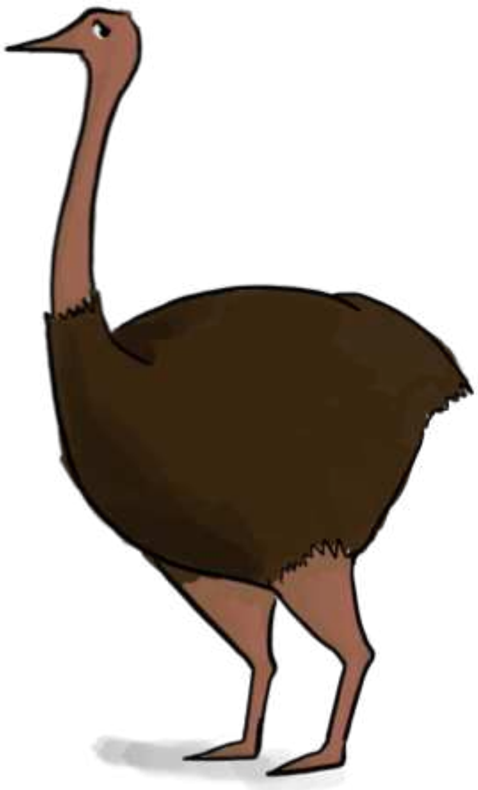
Many cases



- 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!



Ollie the over-achieving ostrich

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?

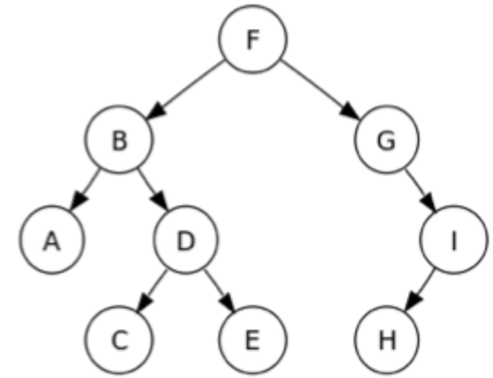
What have we learned?

- 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))

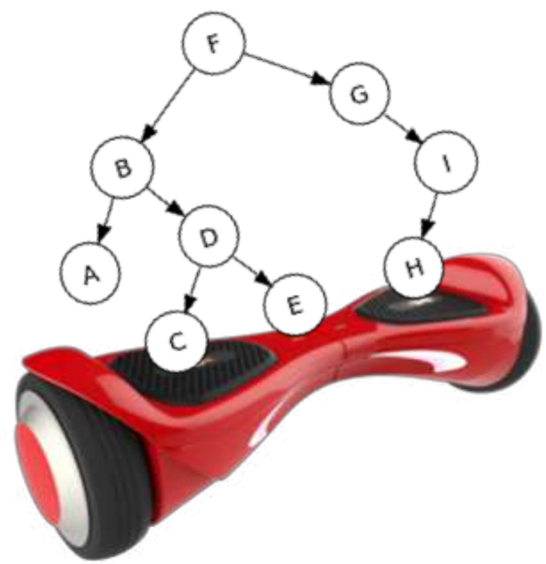
Today



- Begin a brief foray into data structures!
  - See CS 166 for more!
- Binary search trees
  - You may remember these from CS 106B
  - They are better when they're balanced.

this will lead us to…

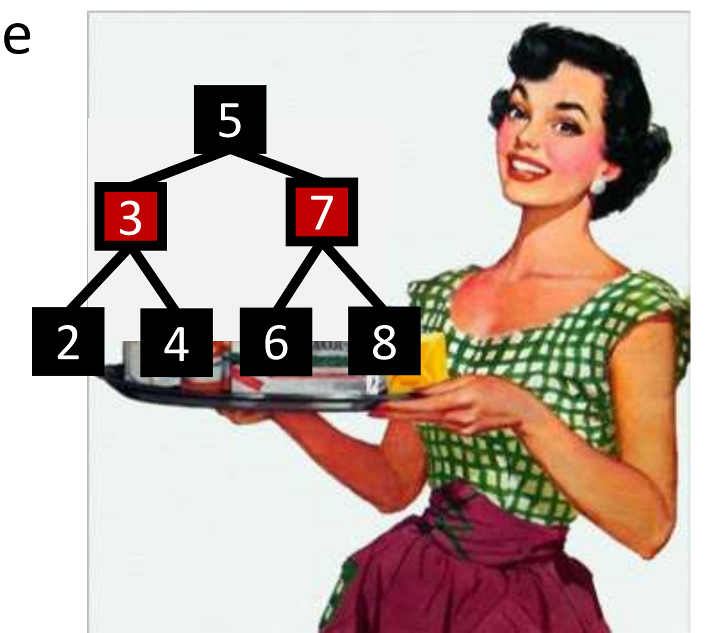
- Self-Balancing Binary Search Trees
  - **Red-Black** trees.



Recap

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



Next time

- **Hashing!**
- I won’t be here, but Greg Valiant will be!