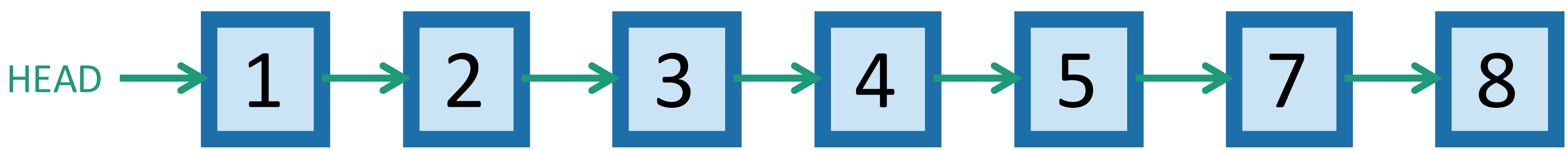
Lecture 7

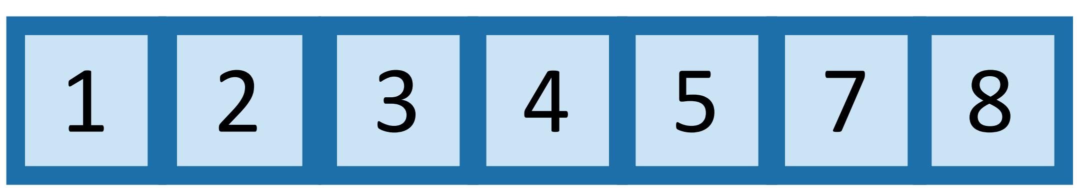
Binary Search Trees and Red-Black Trees

Motivation for binary search trees

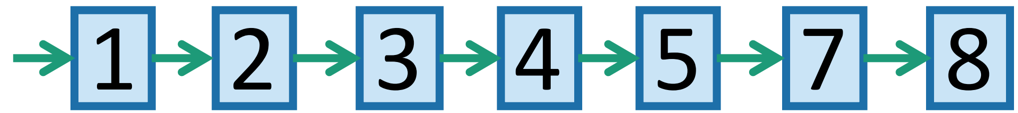
- We've been assuming that we have access to some basic data structures:
  - (Sorted) linked lists



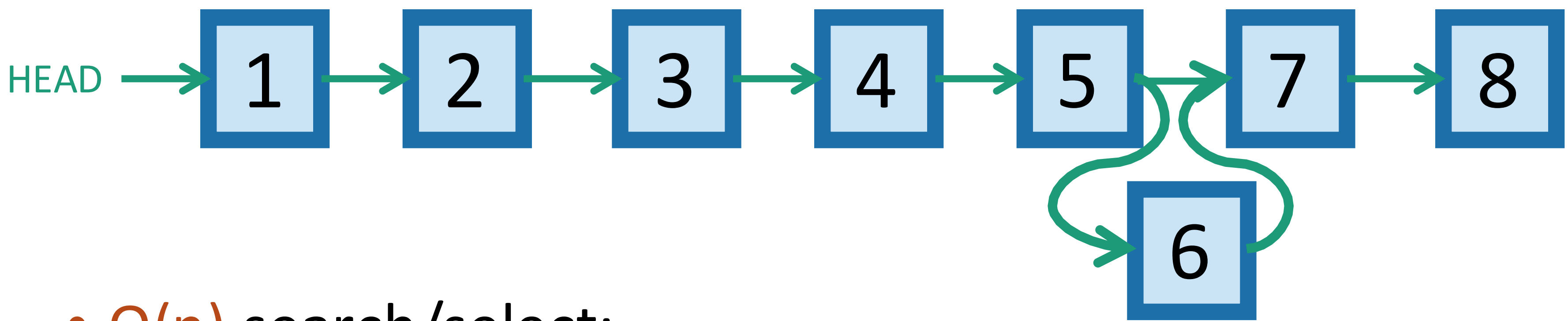
  - (Sorted) arrays



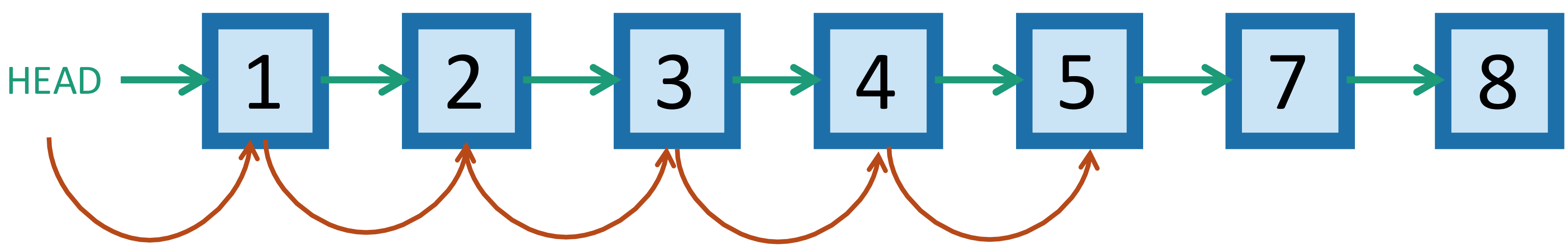
Sorted linked lists



- O(1) insert/delete (assuming we have a pointer to the location of the insert/delete):

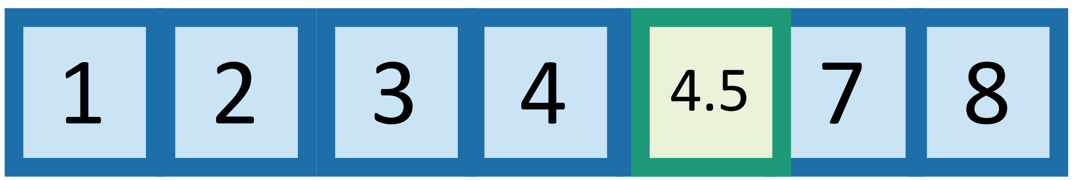


- O(n) search/select:

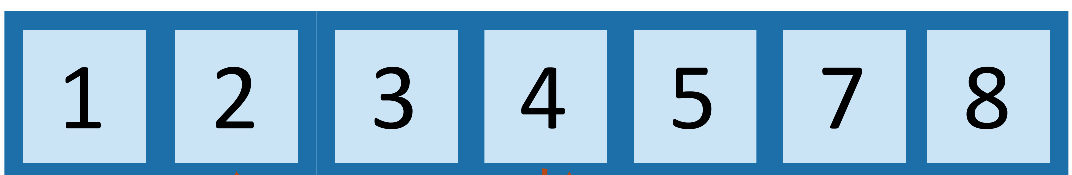


Sorted Arrays

- O(n) insert/delete:



- O(log(n)) search, O(1) select:



Search: Binary search to see if 3 is in A.

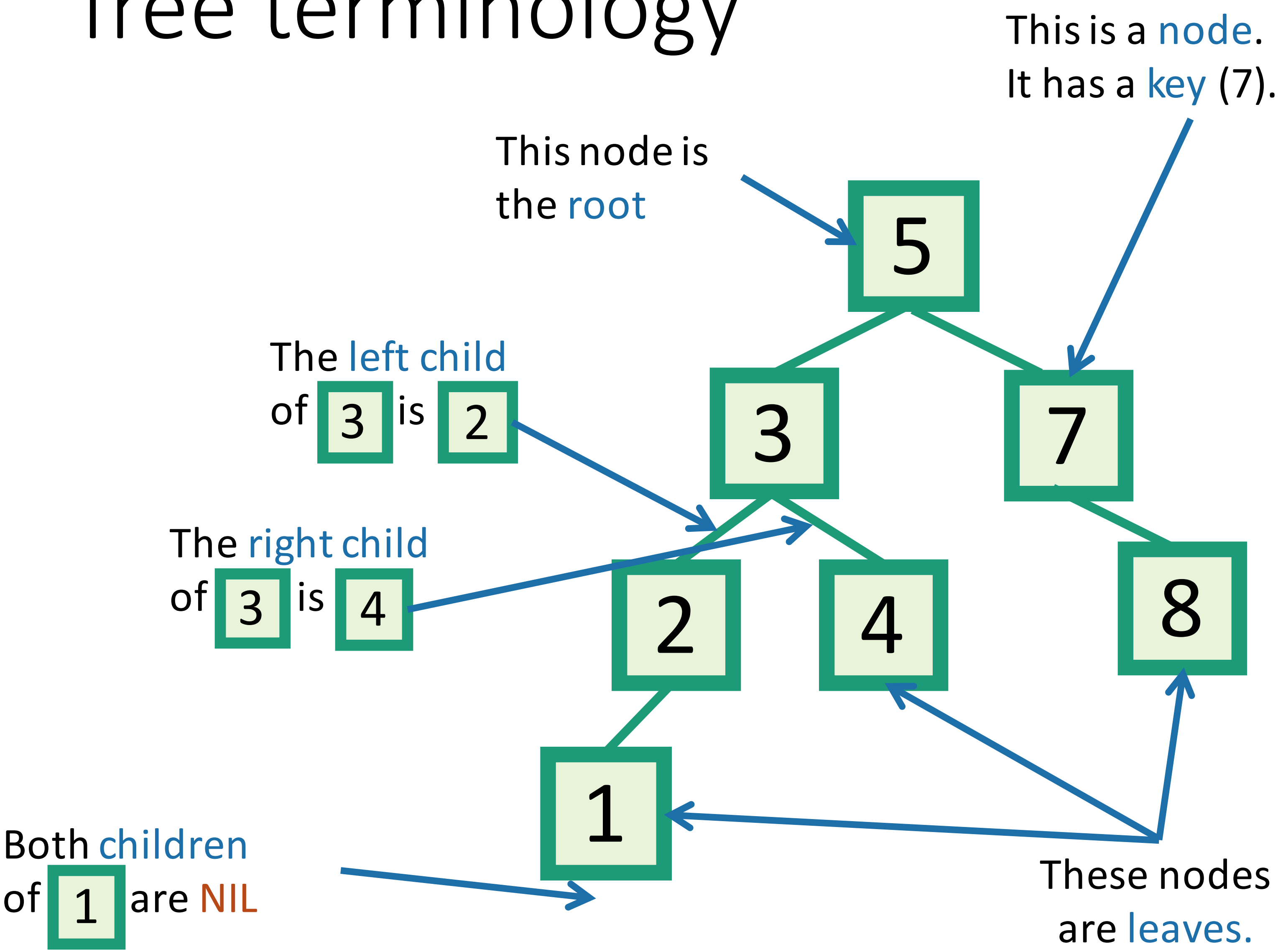
Select: Third smallest is A[3].

The best of both worlds

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))

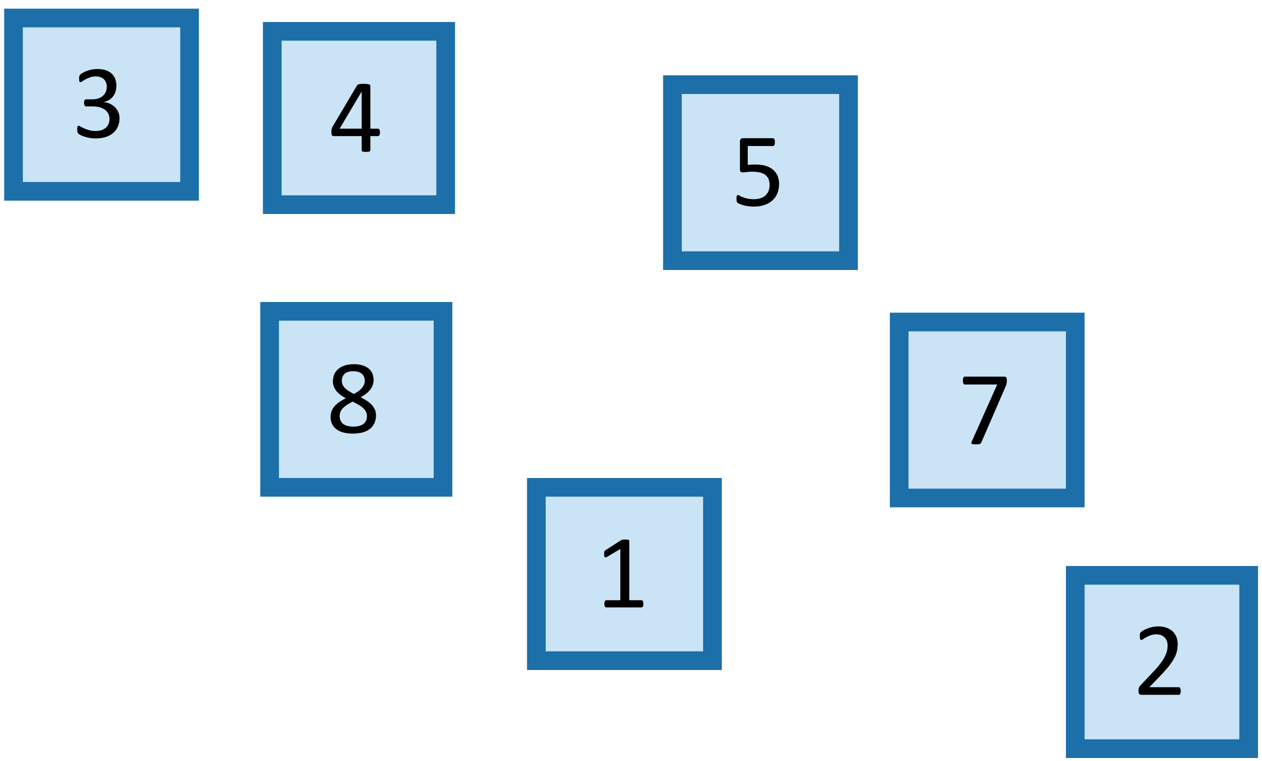
Tree terminology



Today all keys are distinct.

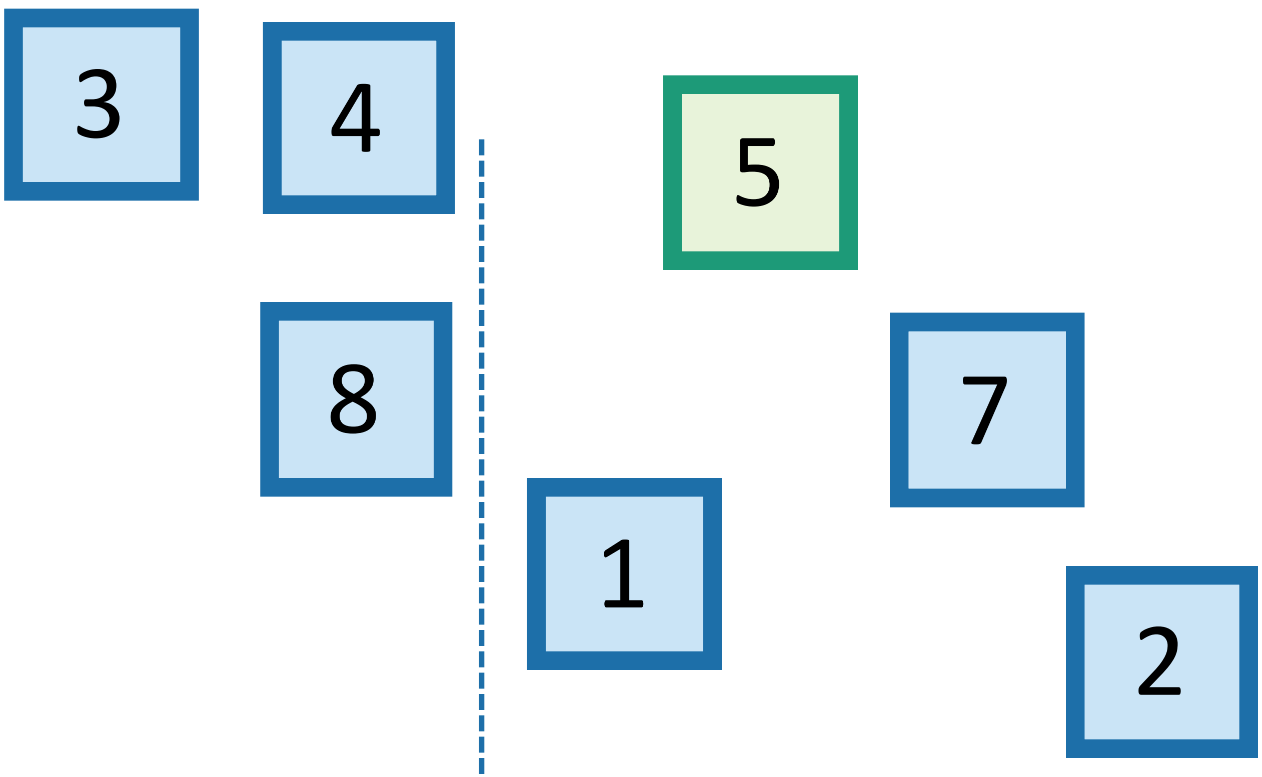
Binary Search Trees

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



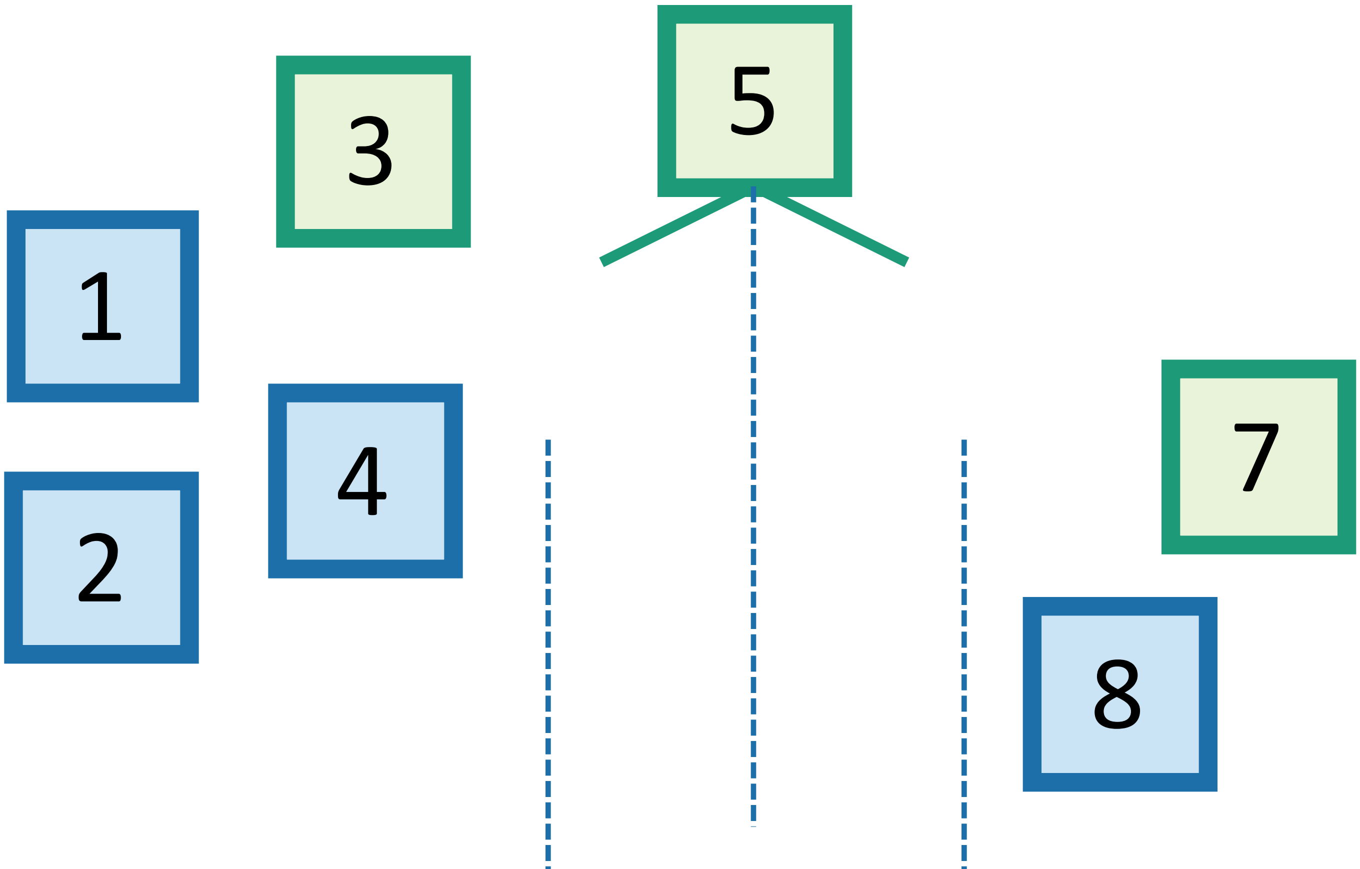
Binary Search Trees

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



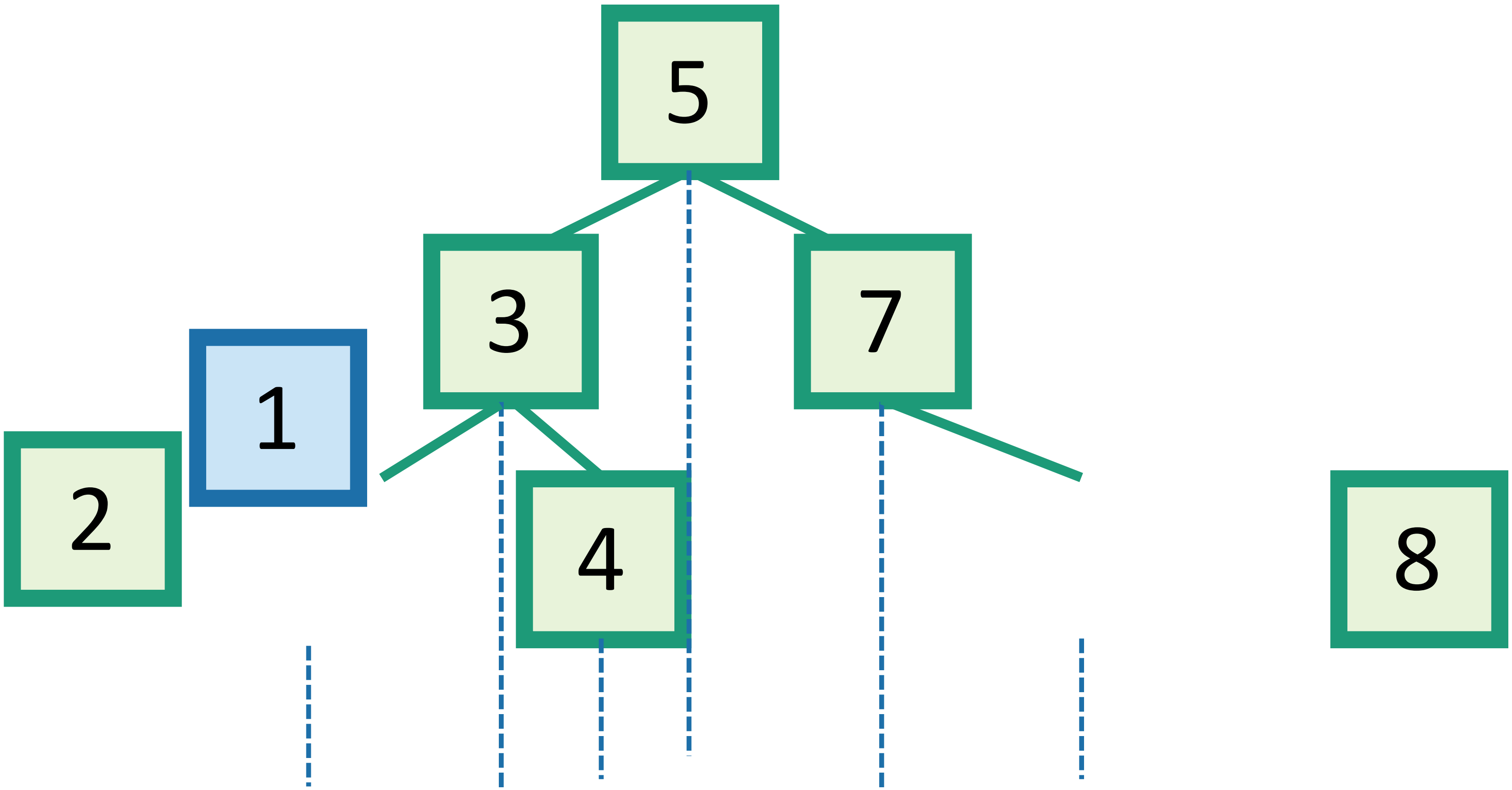
Binary Search Trees

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



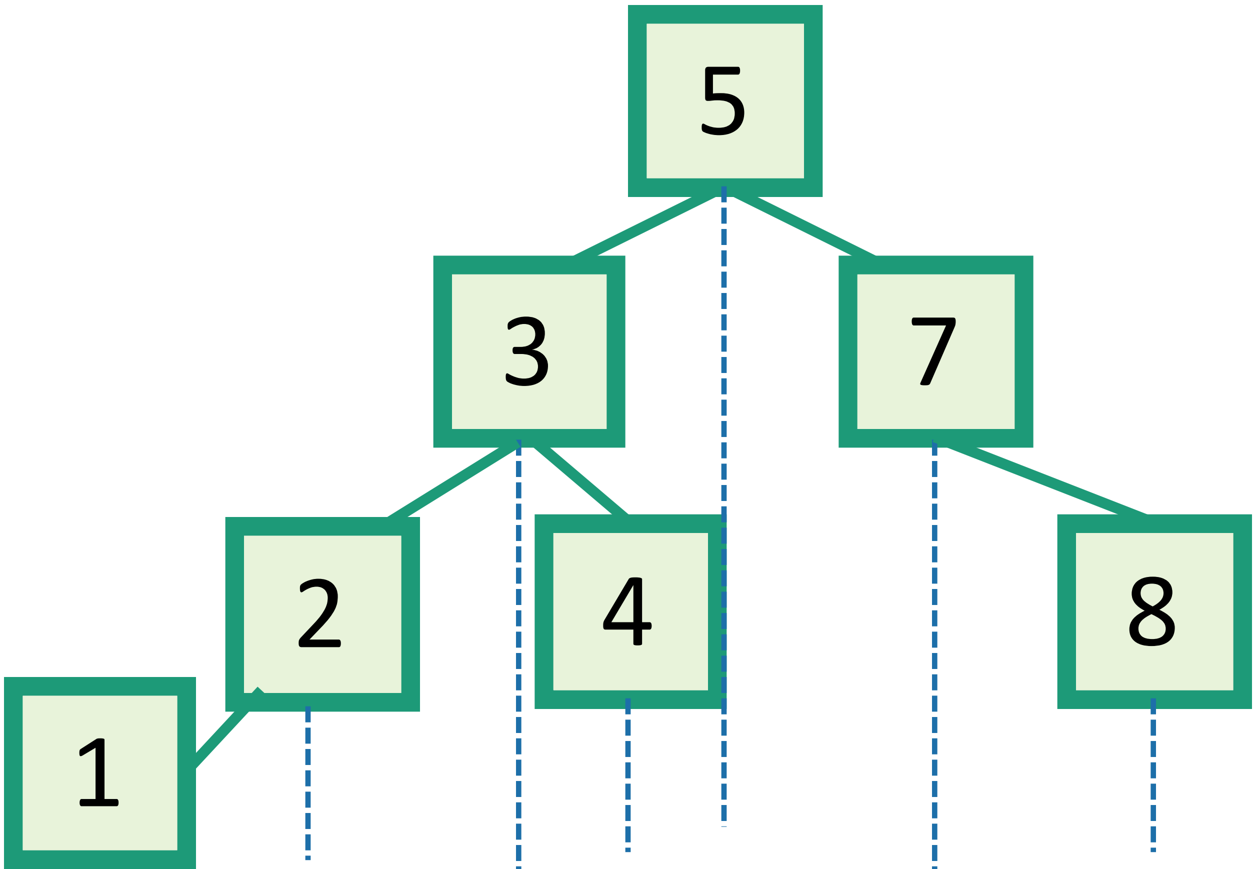
Binary Search Trees

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



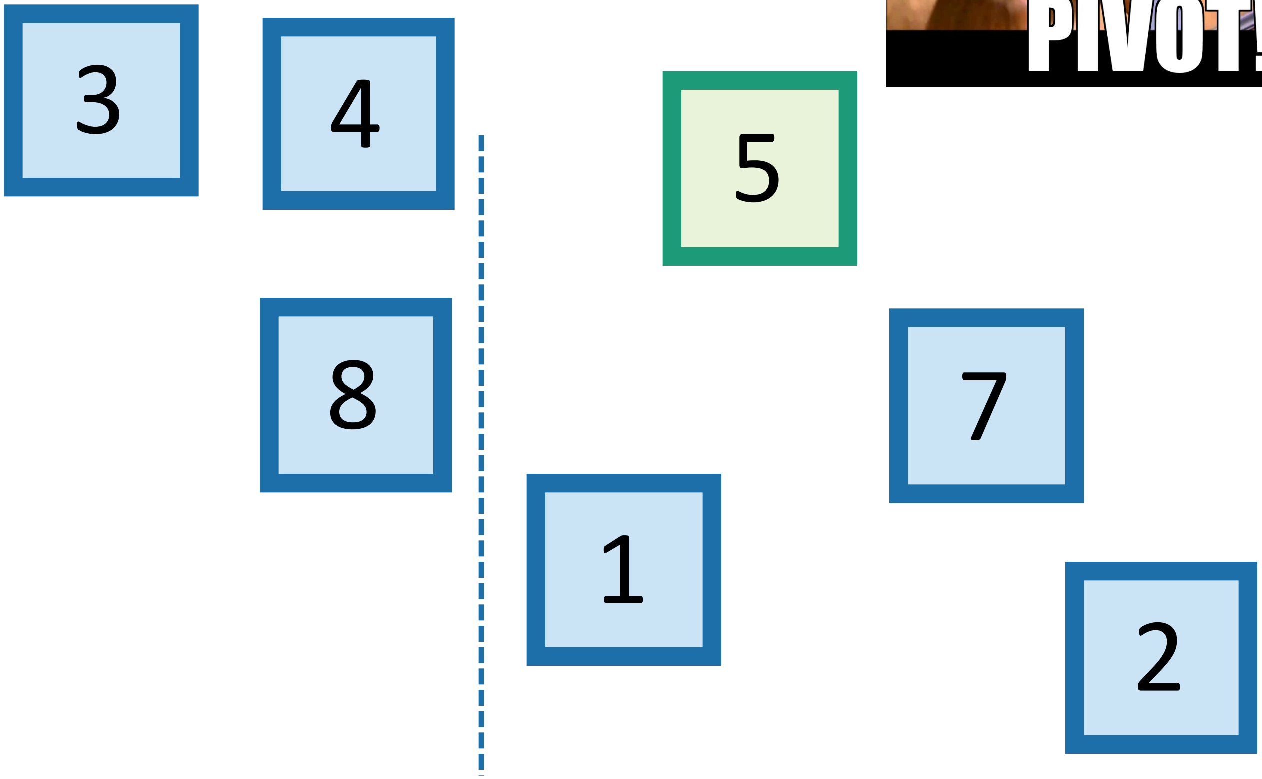
Binary Search Trees

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



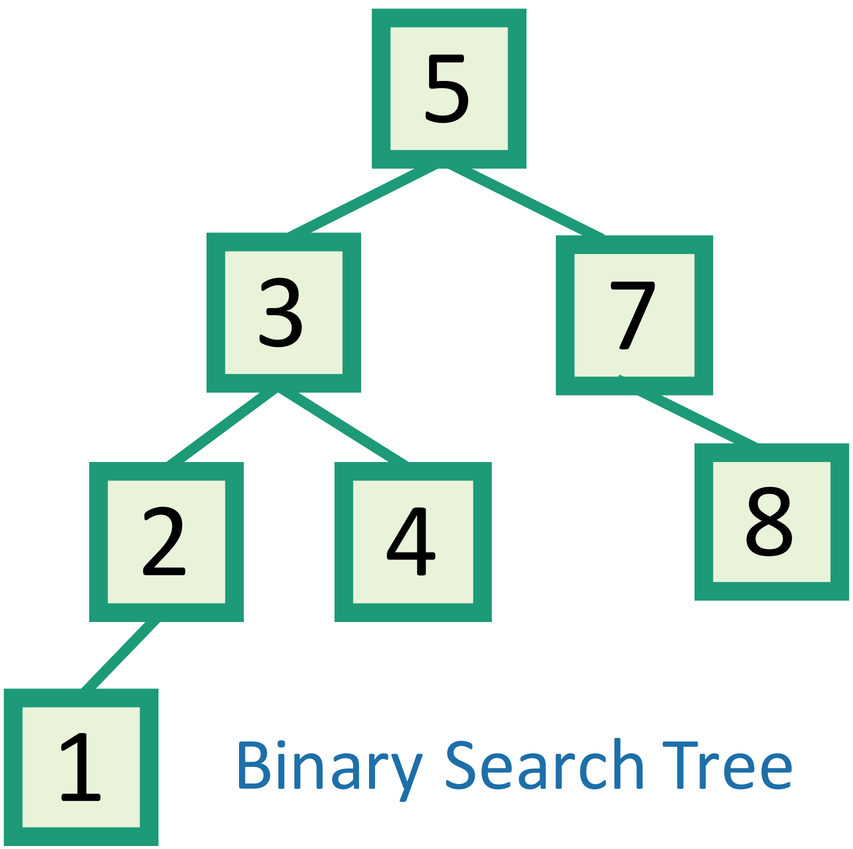
Aside: this should look familiar

kinda like QuickSort

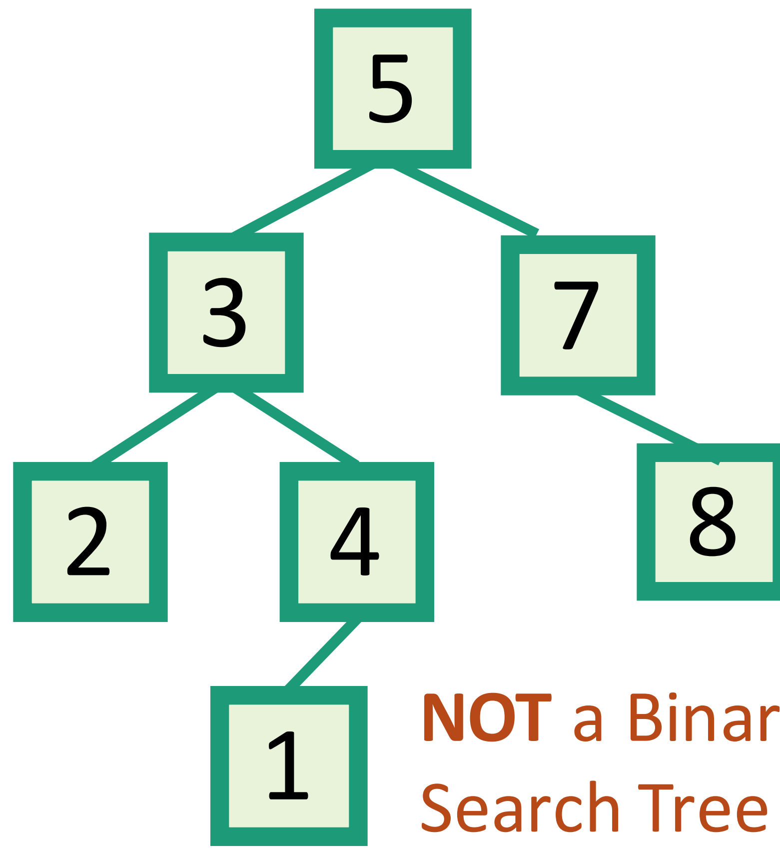


Binary Search Trees

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



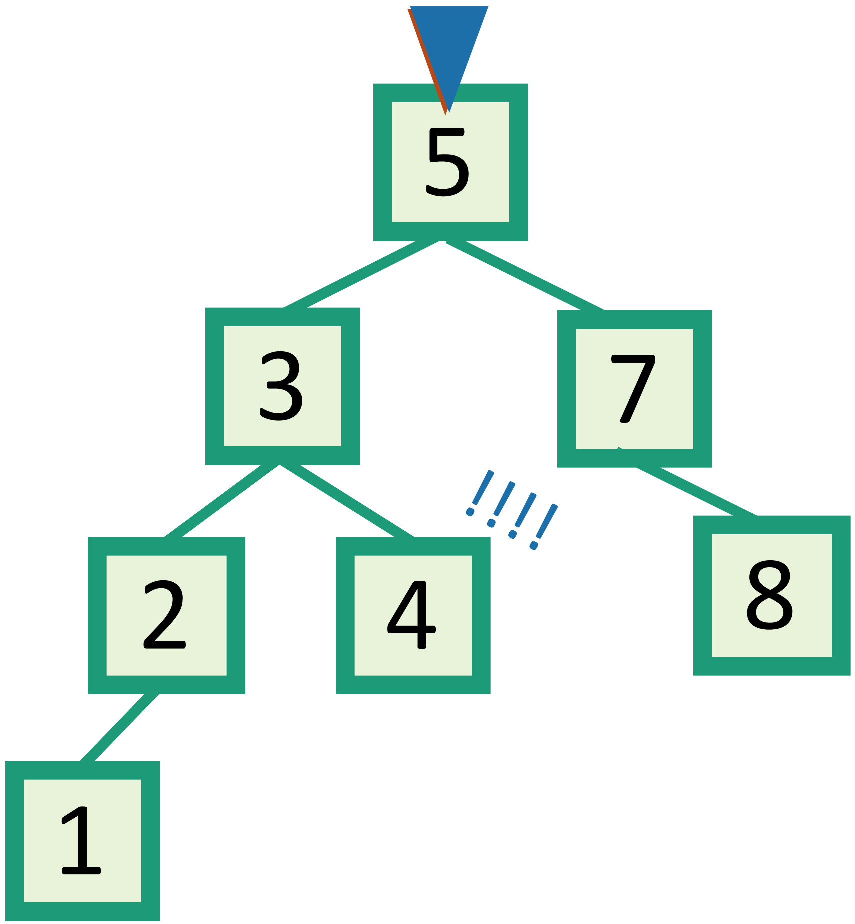
Binary Search Tree



NOT a Binary Search Tree

SEARCH in a Binary Search Tree

definition by example

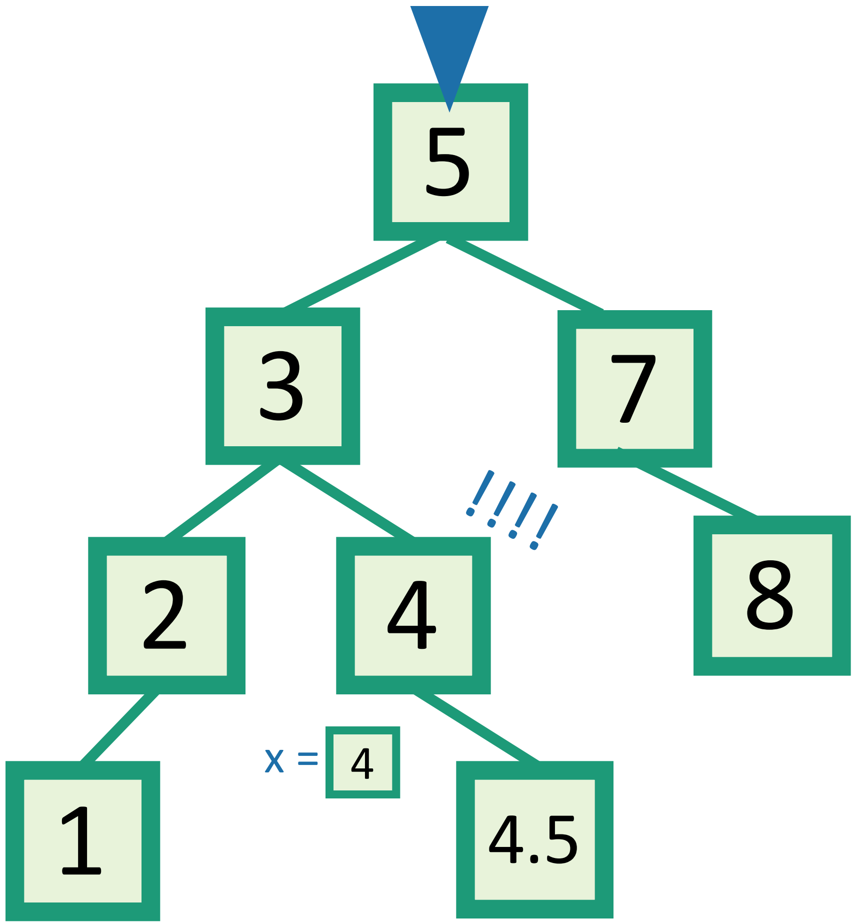


EXAMPLE: Search for 4.

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)

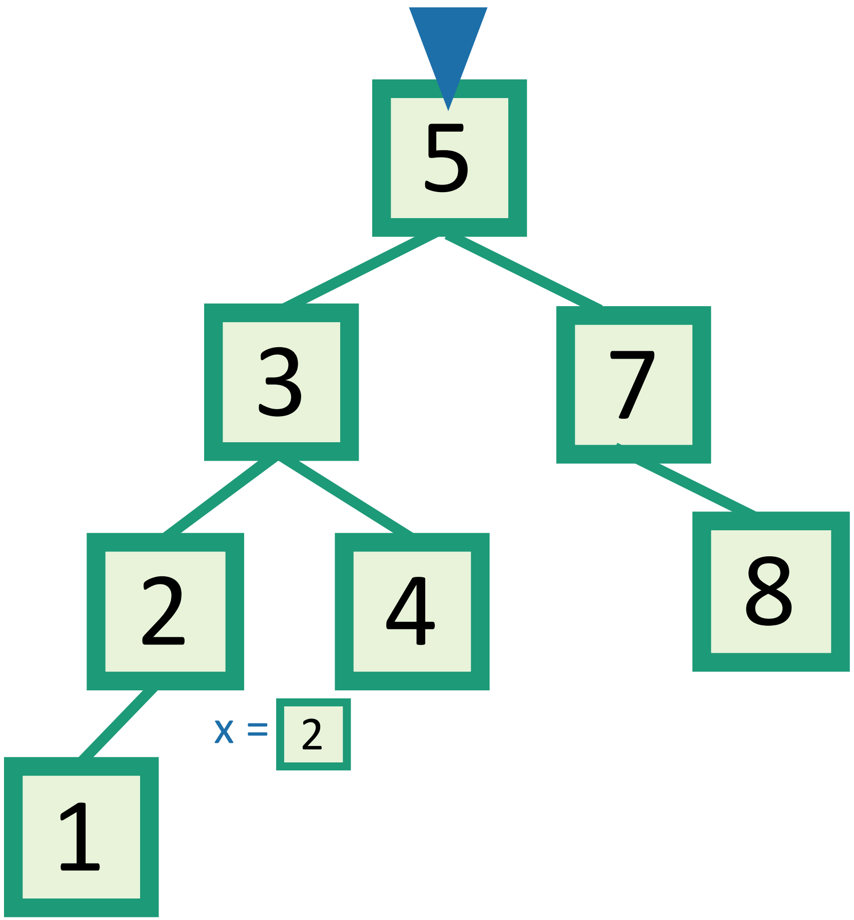
INSERT in a Binary Search Tree



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



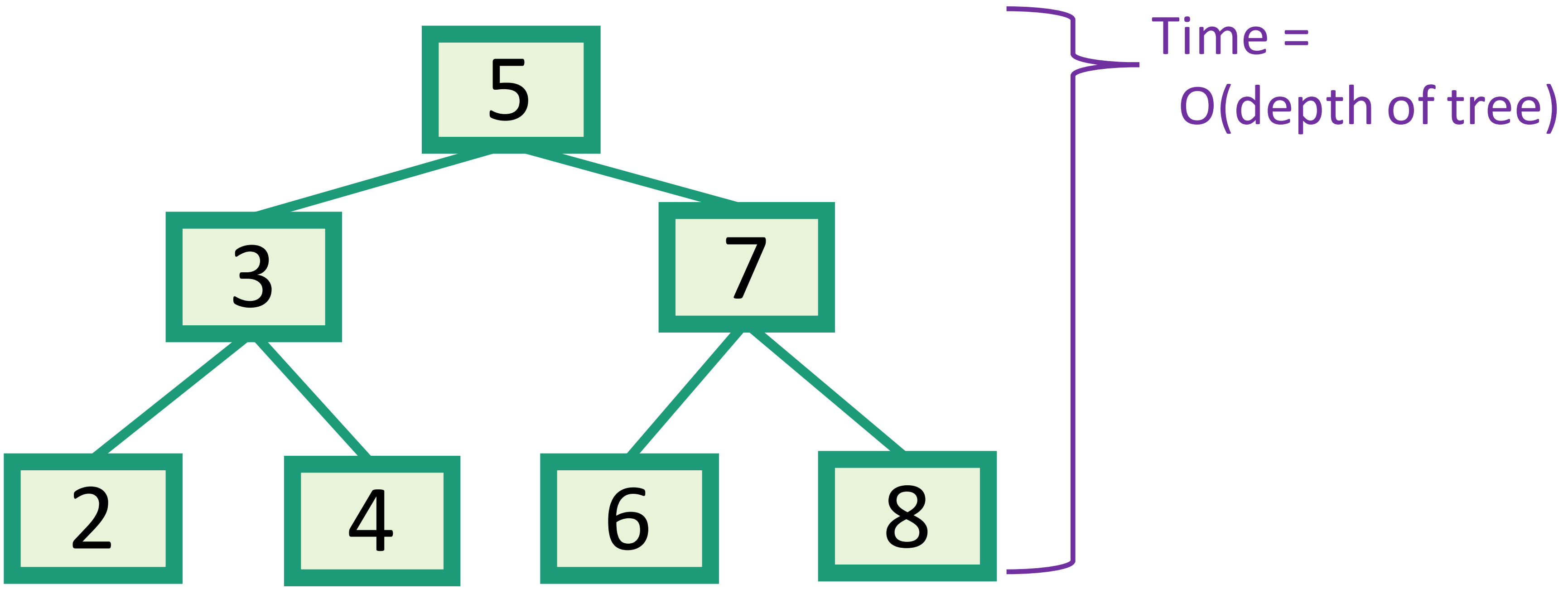
EXAMPLE: Delete 2

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

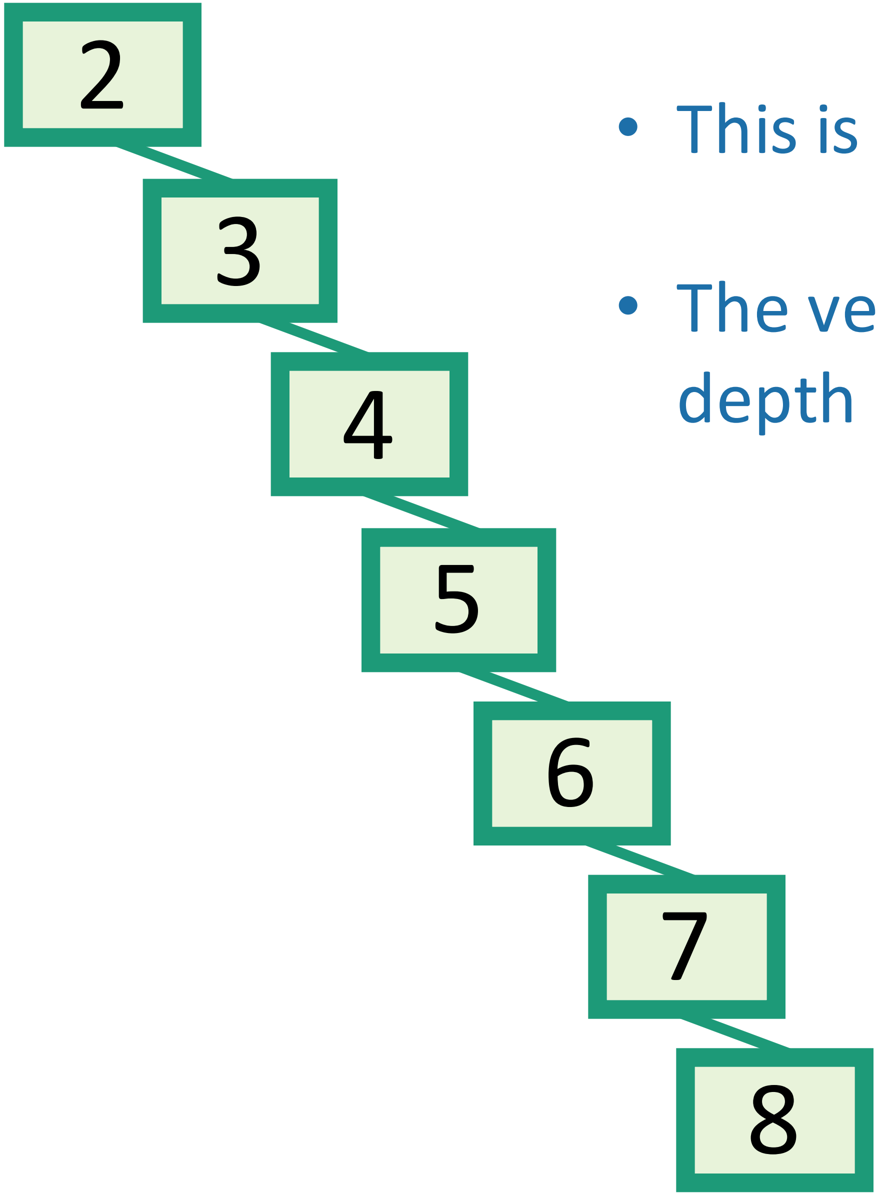
This is a bit more complicated…

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.



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.

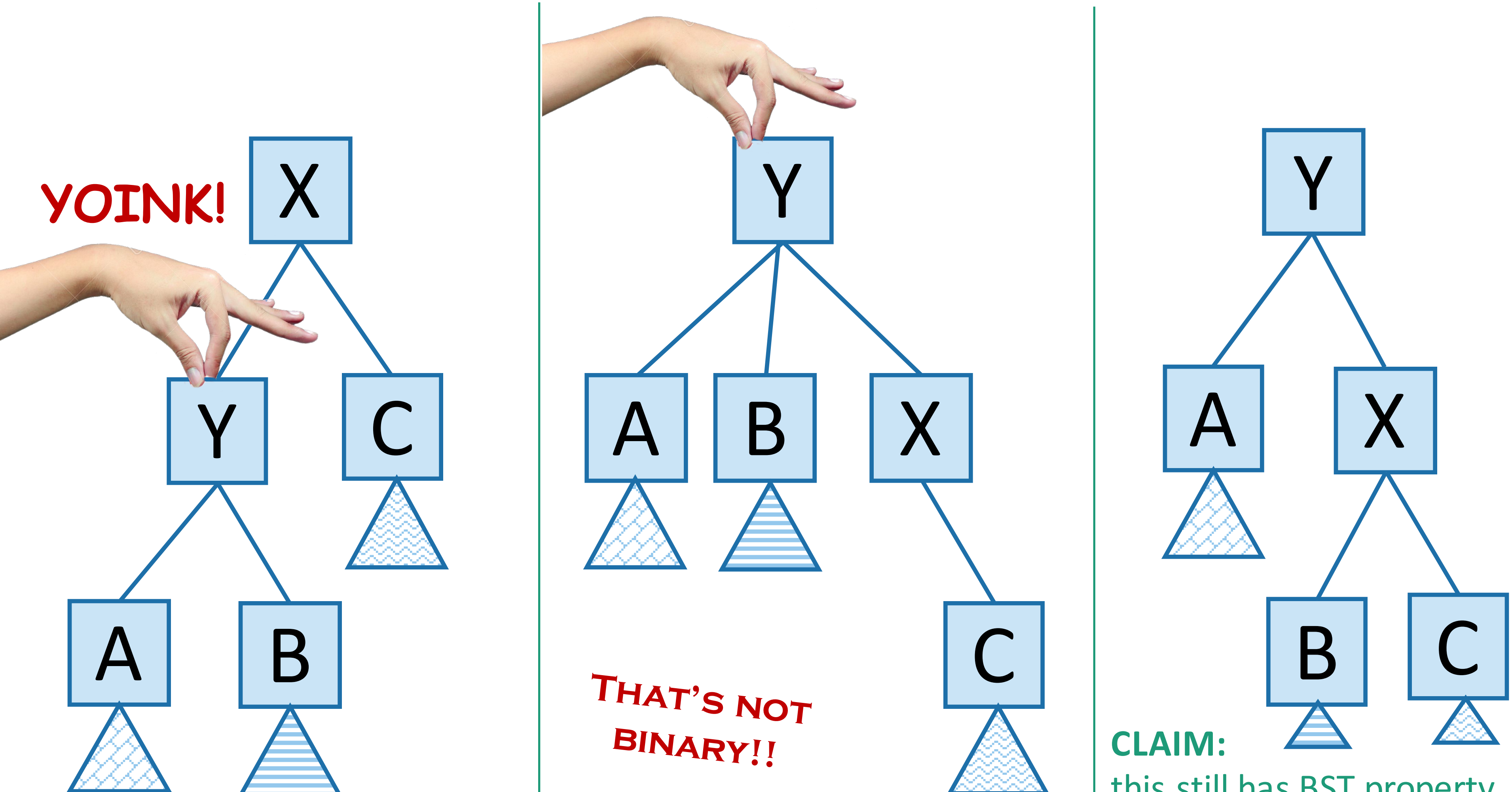
What to do?

- Keep track of how deep the tree is getting.

Idea 1: Rotations

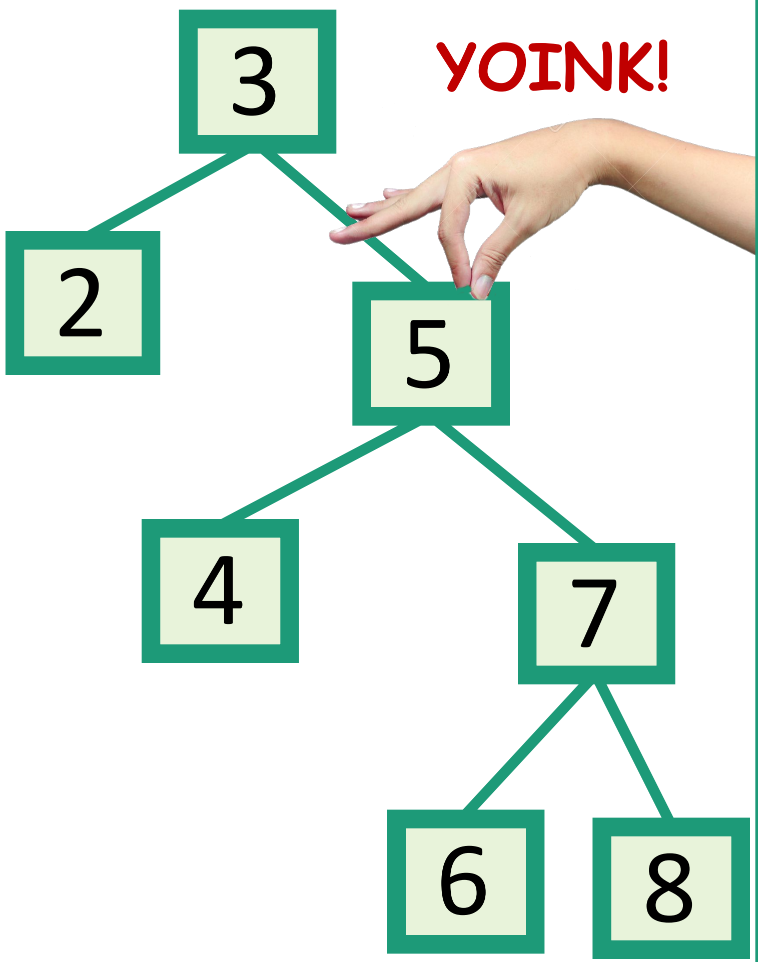
No matter what lives underneath A,B,C,
this takes time O(1). (Why?)

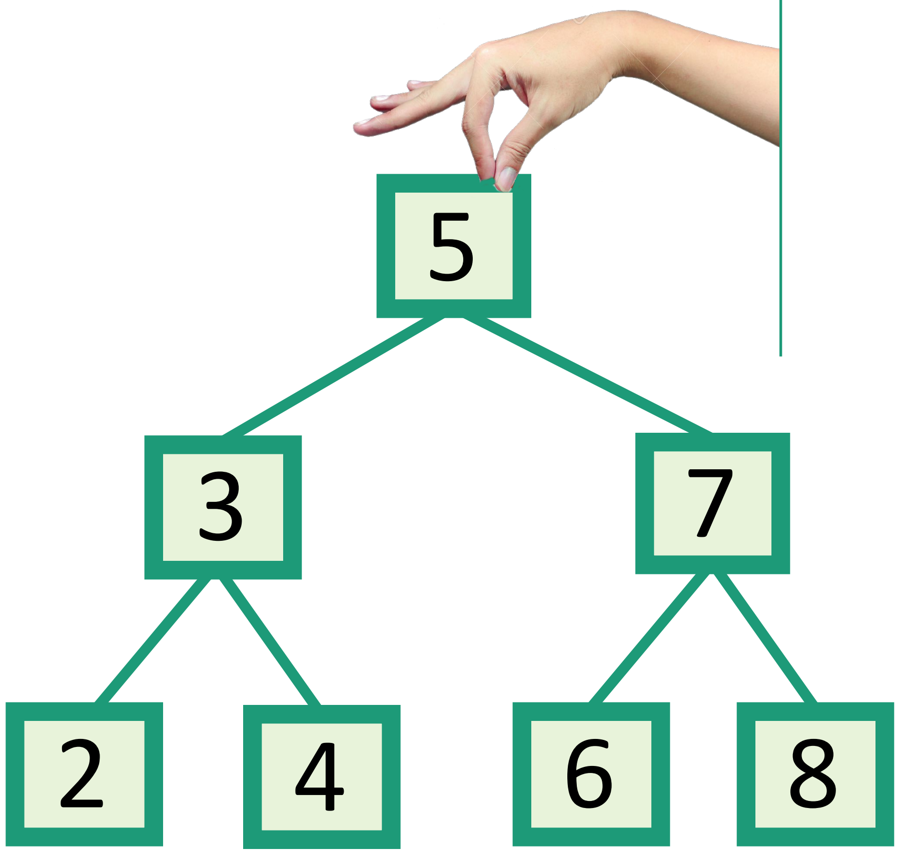
- Maintain Binary Search Tree (BST) property, while moving stuff around.



CLAIM:
this still has BST property.

This seems helpful





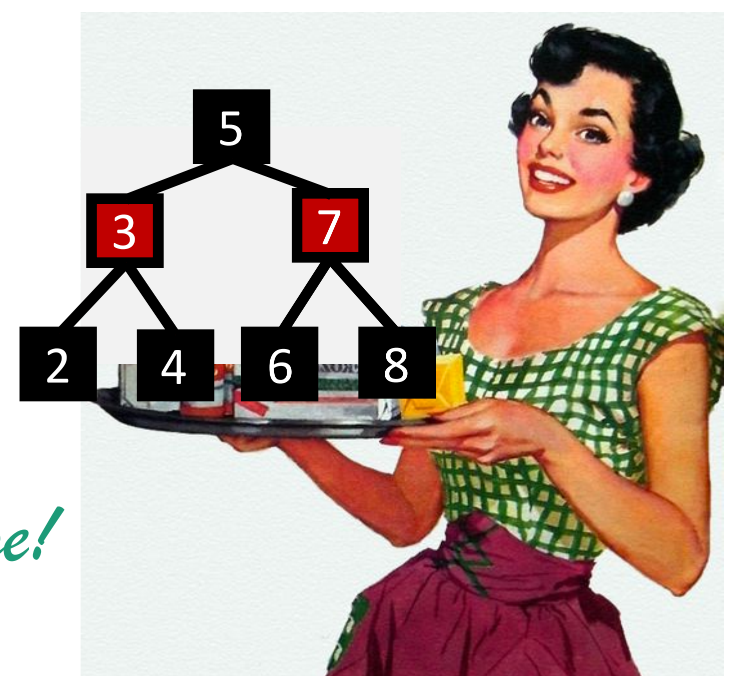
Idea 2: have some proxy for balance

- 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.

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!



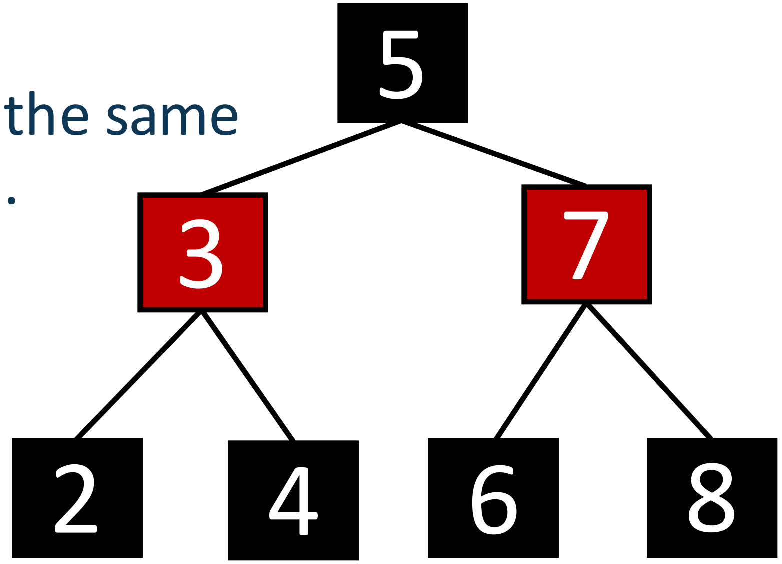
Maintain balance by stipulating that **black nodes** are balanced, and that there *aren't too many* **red nodes**.

It's just good sense!

Red-Black Trees

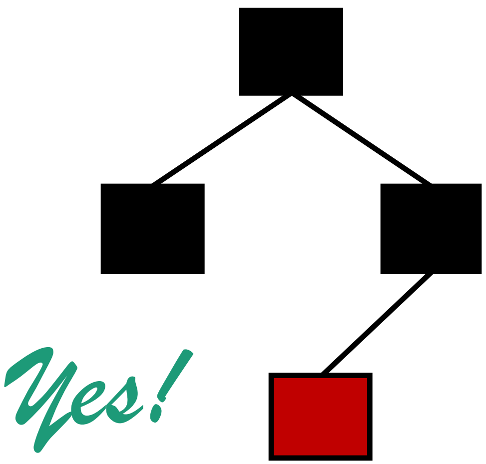
these rules are the proxy for balance

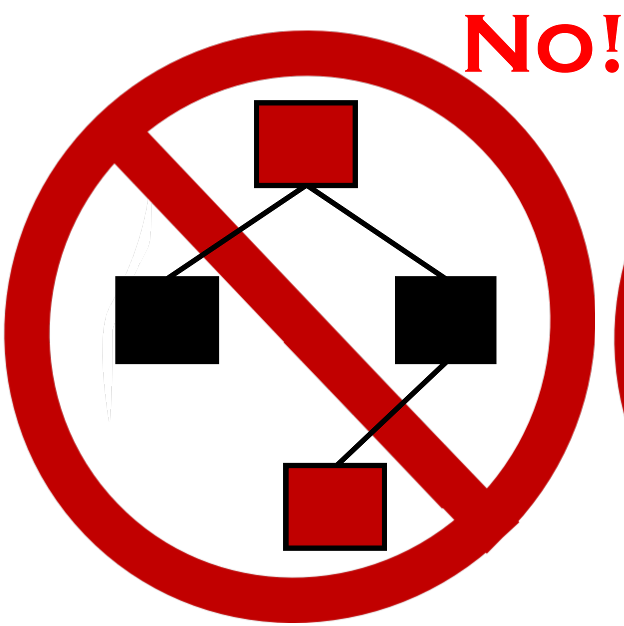
- Every node is colored **red** or **black.**
- The root node is a **black node**.
- NIL children count as **black nodes**.
- Children of a **red node** are **black nodes**.
- For all nodes x:
  - all paths from x to NONE's have the same number of black nodes on them.

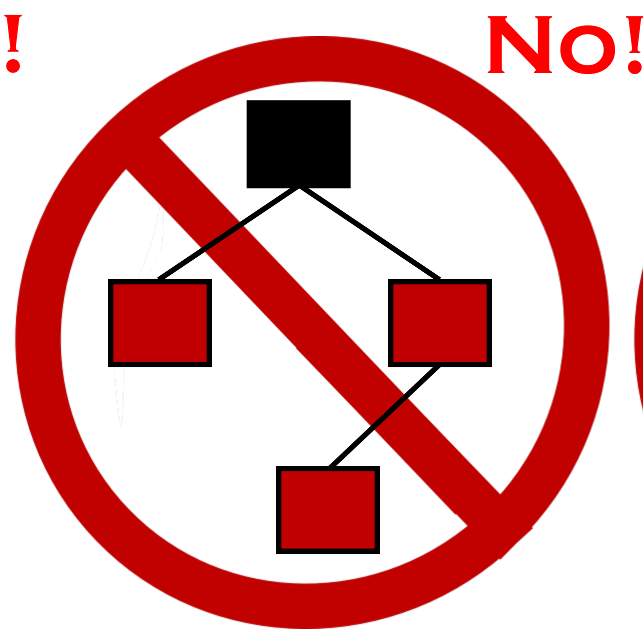


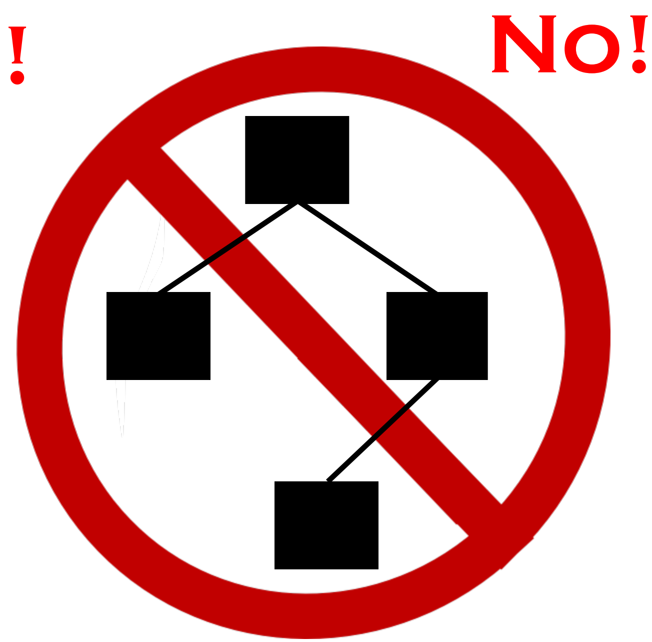
Examples(?)

- Every node is colored **red** or **black.**
- The root node is a **black node**.
- NONE children count as **black nodes**.
- Children of a **red node** are **black nodes**.
- For all nodes x:
  - all paths from x to NONE's have the same number of black nodes on them.



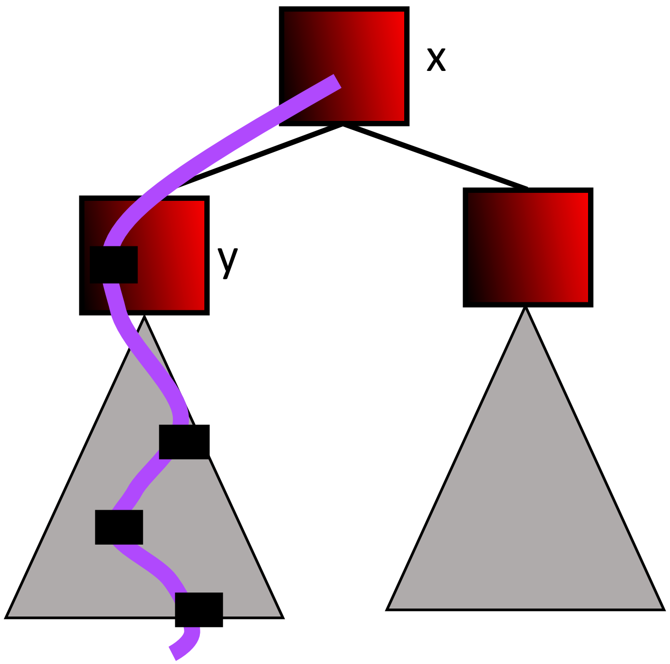






That turns out the be basically right.

[proof sketch]



- Say there are b(x) black nodes in any path from x to NONE.
  - (including x).
- **Claim:**
  - Then there are at least $2^{b(x)} - 1$ nodes in the subtree underneath x.
- [Proof by induction – on board if time]

Then:

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

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

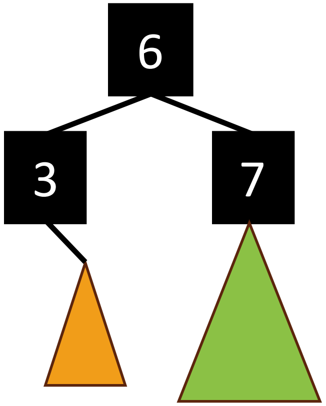
Rearranging:

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

Okay, so it's balanced…
…but can we maintain it?

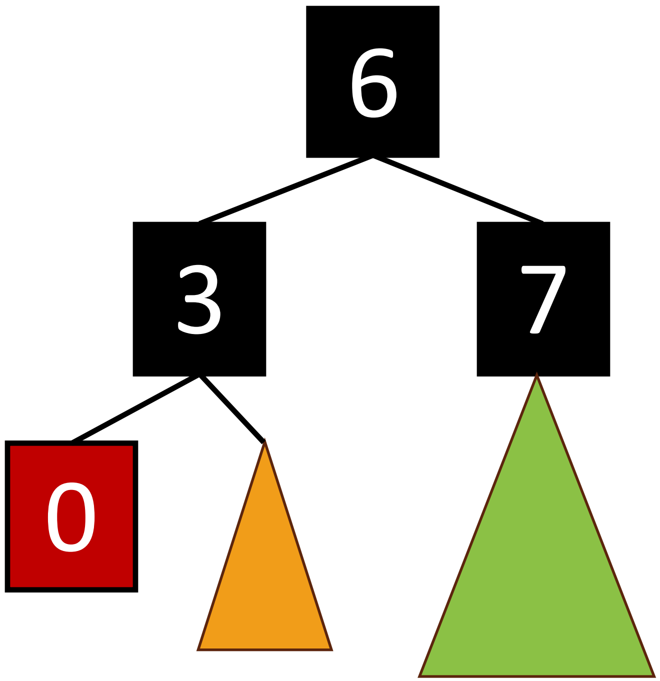
- Yes!

Inserting into a Red-Black Tree



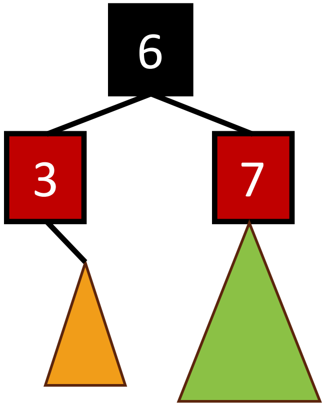
What if it looks like this?

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



Example: insert 0

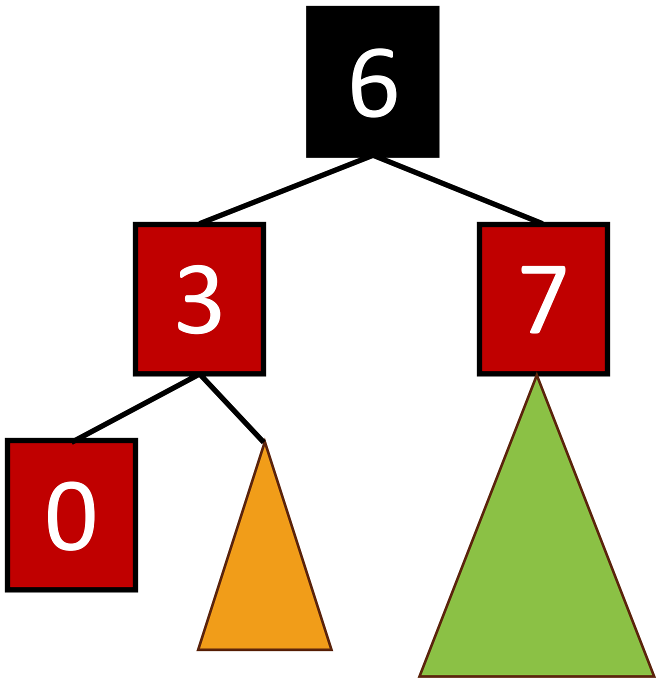
Inserting into a Red-Black Tree



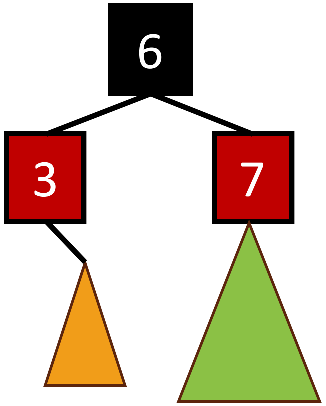
What if it looks like this?

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

Example: insert 0

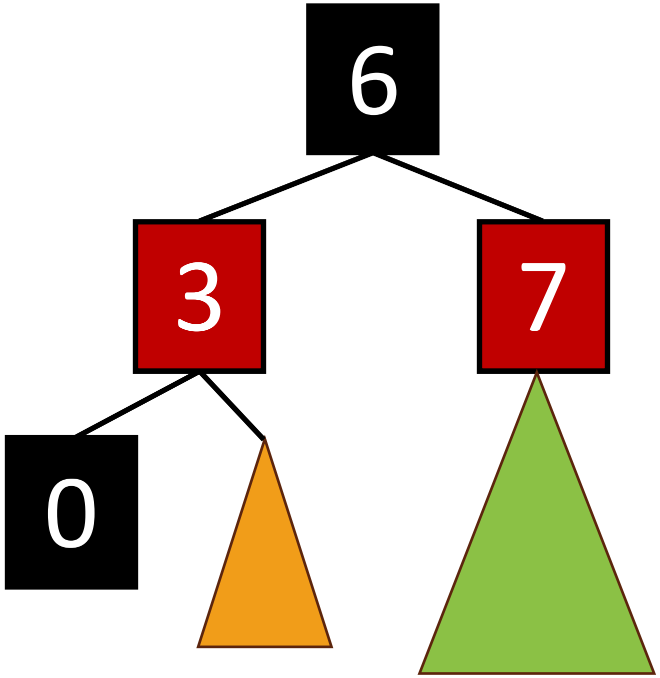


Inserting into a Red-Black Tree



What if it looks like this?

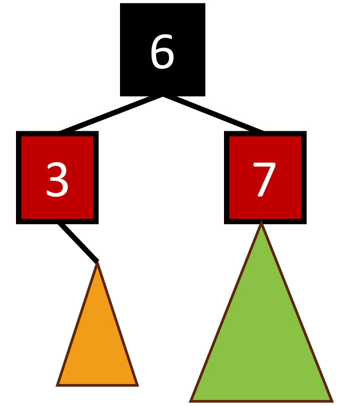
- Make a new **red node.**
- Insert it as you would normally.
- Fix things up if needed.



Example: insert 0

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

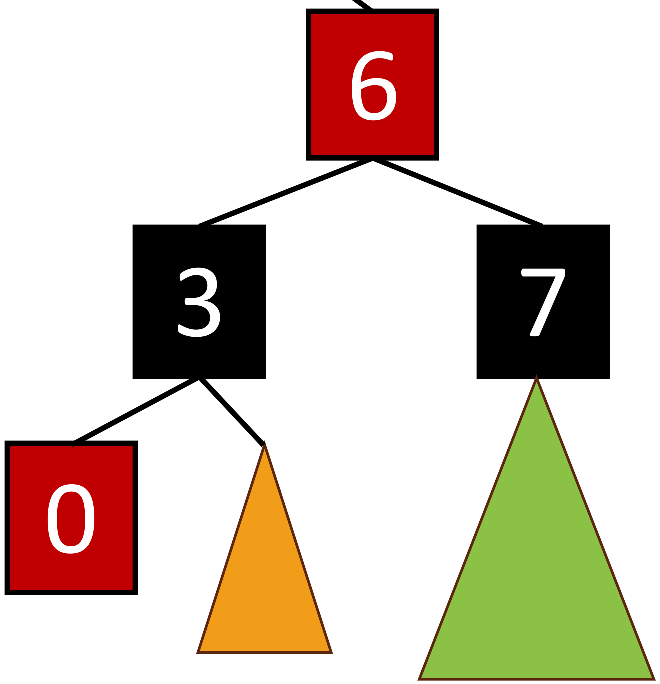
Inserting into a Red-Black Tree



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

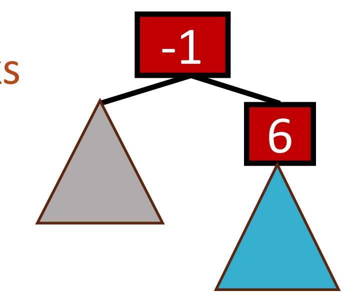


- Instead recolor like this.
- Need to argue:
  - RB-Tree properties still hold.

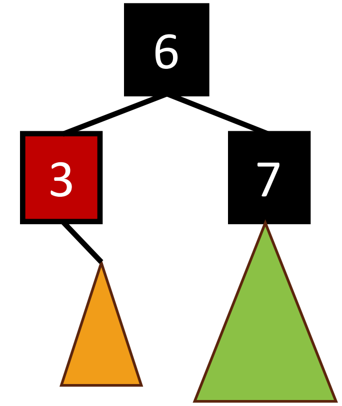
ish

- What about the red root?
  - if 6 is actually the root, color it black.
  - Else, recursively re-color up the tree.

Now the problem looks like this, where I'm inserting



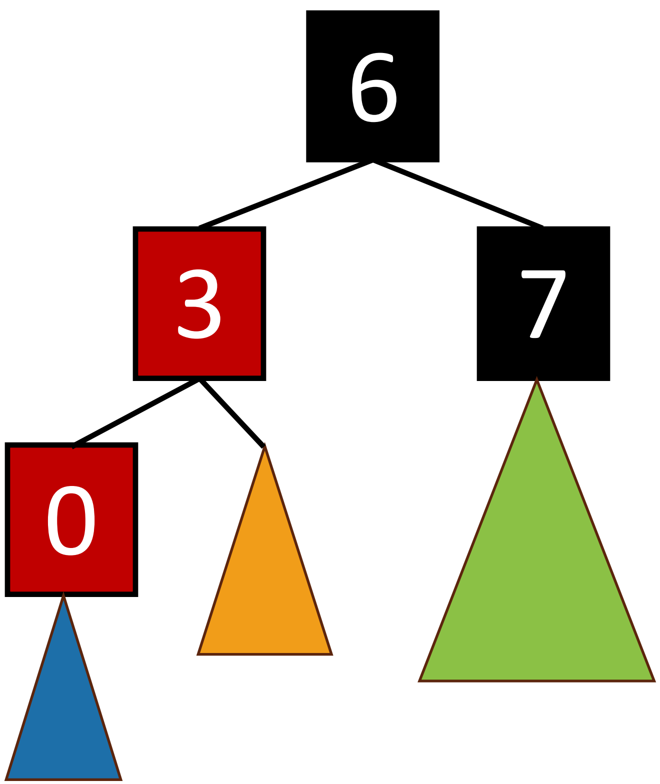
Inserting into a Red-Black Tree



What if it looks like this?

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

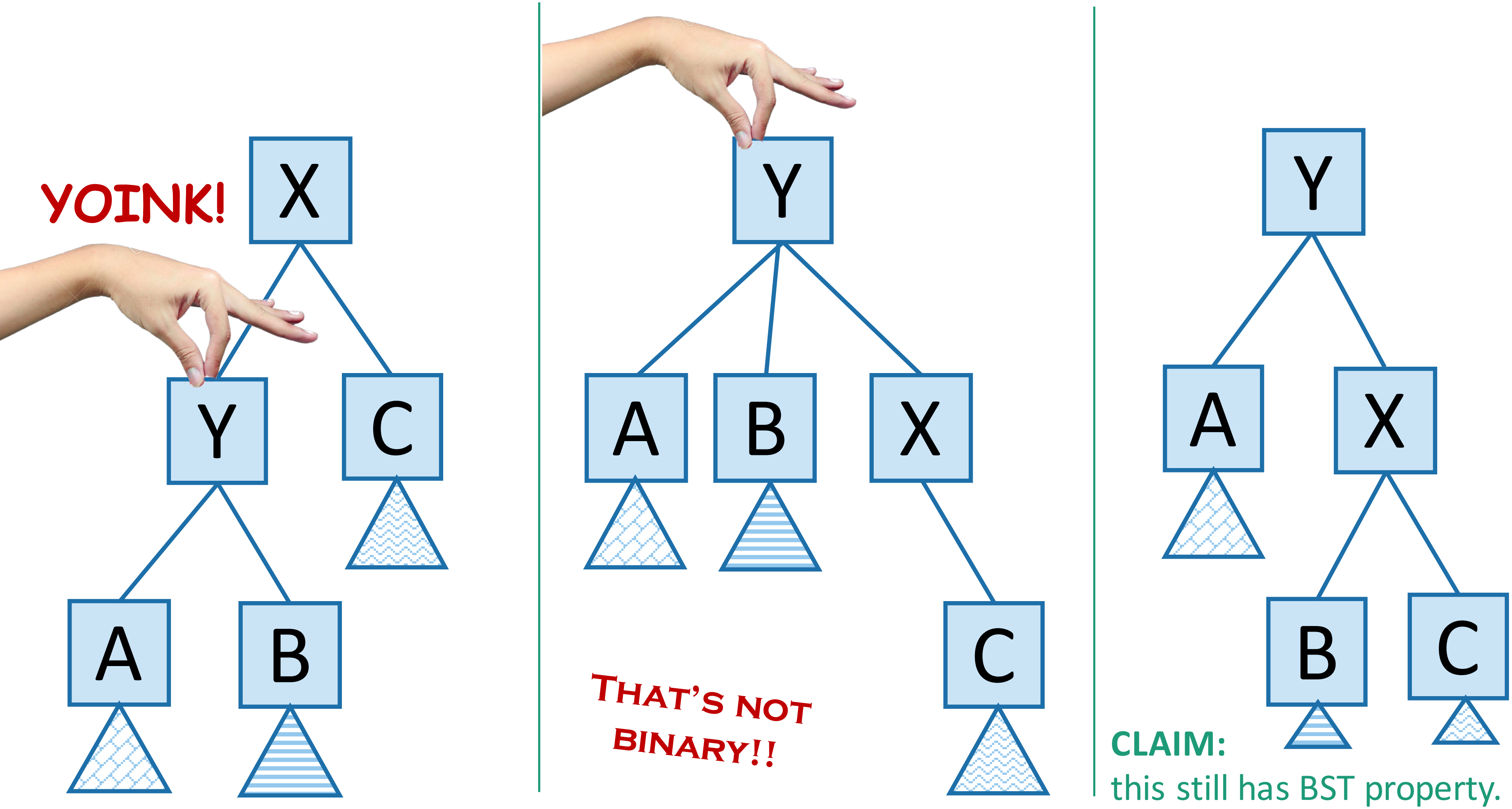
Example: Insert 0.



- Actually, **this can't happen?**
- It might happen that we just turned 0 red from the previous step.
- Or it could happen if 7 is actually NIL.

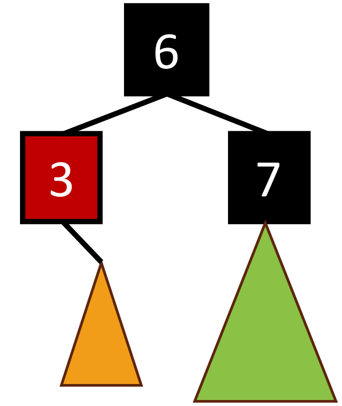
Recall Rotations

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

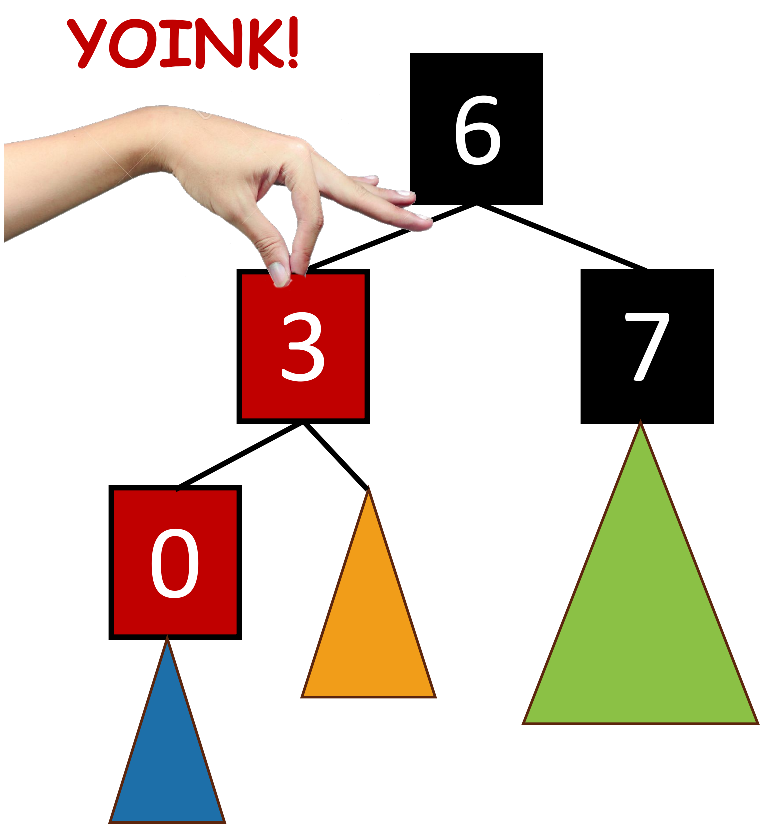


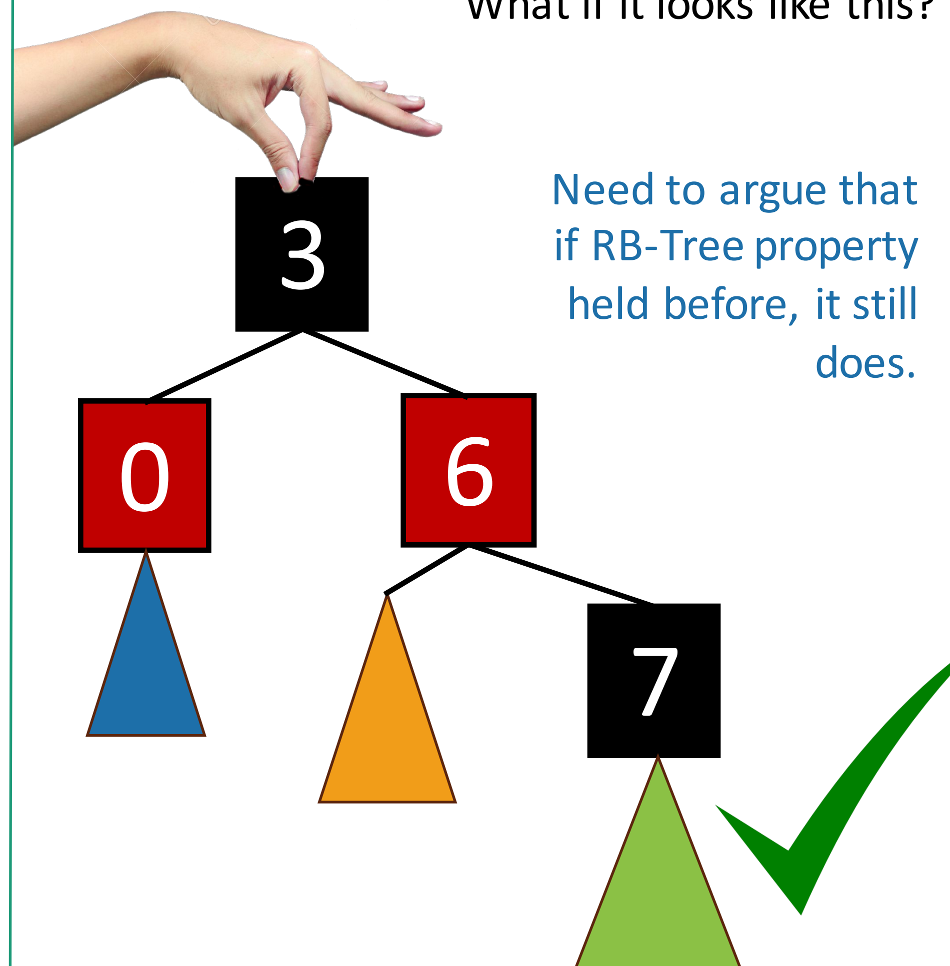
Inserting into a Red-Black Tree

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



What if it looks like this?





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

That’s basically it

- A few things still left to check for INSERT!
  - Anything else that might happen looks basically like what we just did.
  - Formally dealing with the recursion.
  - You check these! (or see CLRS)
- DELETE is similar.

The punchline:

- 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 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))

Summary

- 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