CSE100: Design and Analysis of Algorithms
Lecture 14 – Binary Search Trees (cont.)

Mar 8th 2022

Binary Search Trees and Red-Black Trees

For today all keys are distinct.

Binary tree terminology (review)

Each node has two children.

The left child of 3 is 2

The right child of 3 is 4

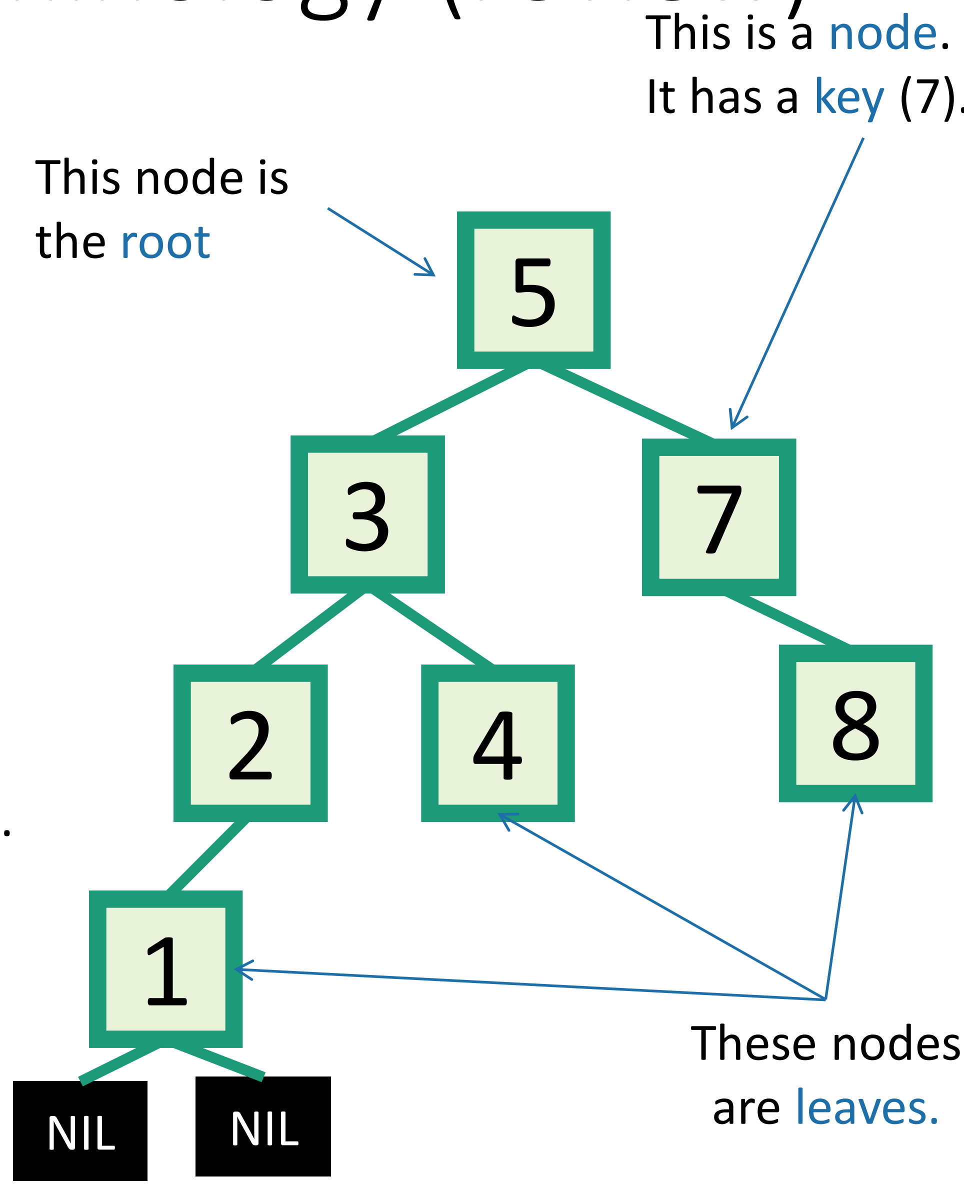
The parent of 3 is 5

2 is a descendant of 5

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

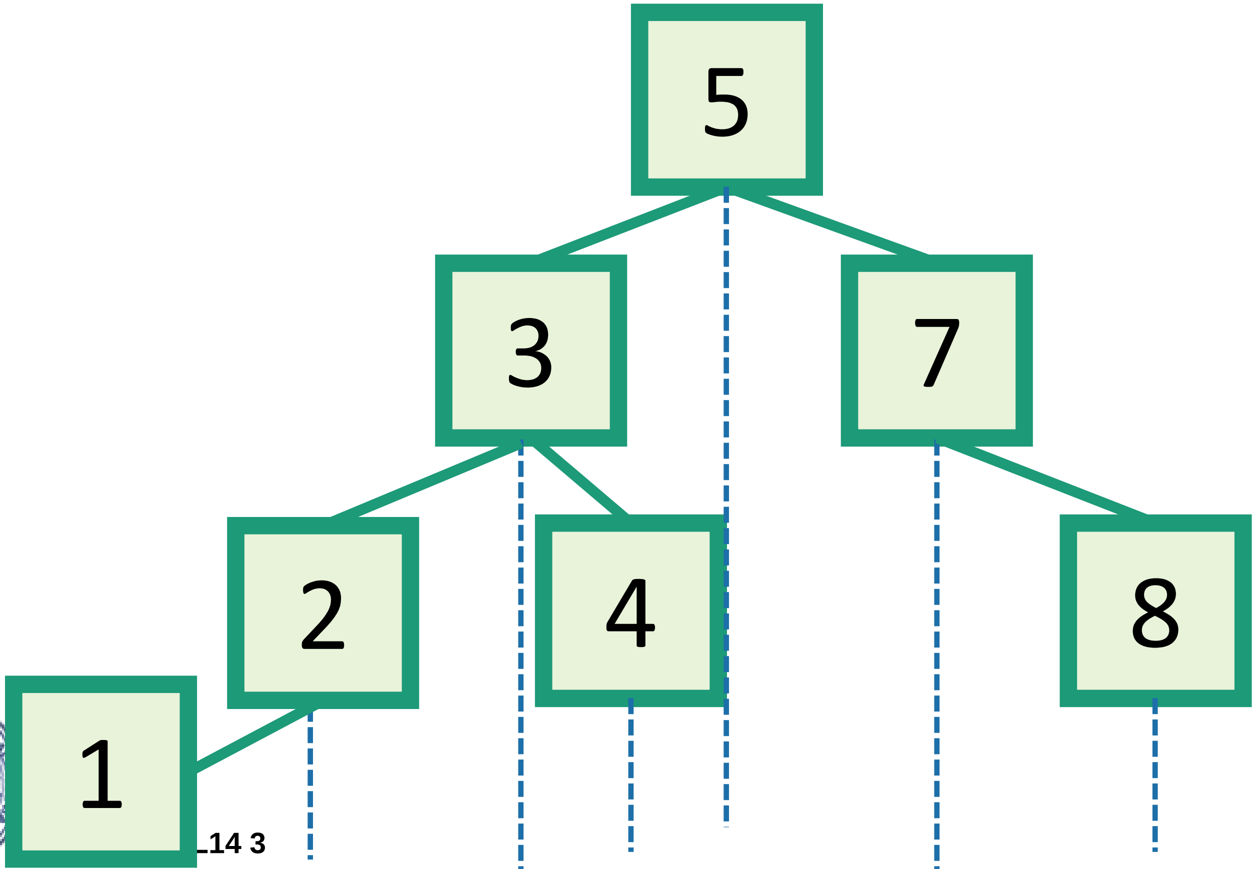
Both children of 1 are NIL.
(We won't usually draw them).

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



Binary Search Trees (review)

- 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:



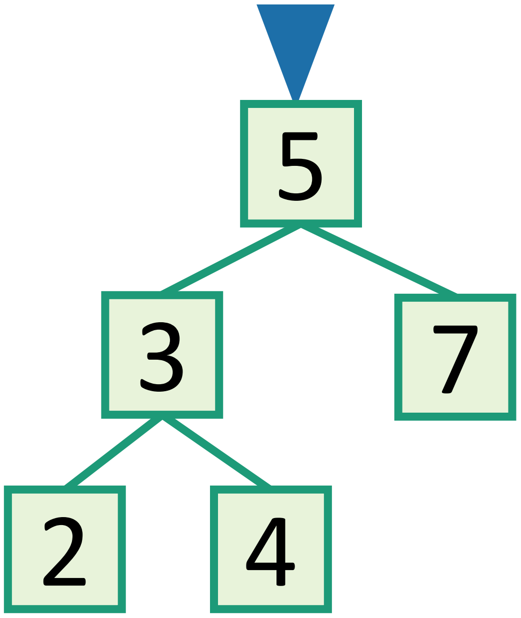
Q: Is this the only binary search tree I could possibly build with these values?

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

Aside: In-Order Traversal of BSTs (review)

- Output all the elements in sorted order!

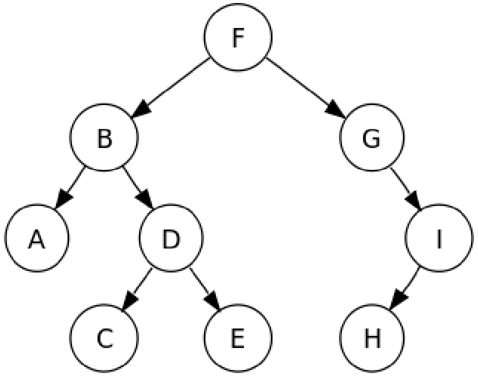
- inOrderTraversal(x):
  - if x!= NIL:
    - inOrderTraversal(x.left)
    - print(x.key)
    - inOrderTraversal(x.right)



2 3 4 5 7 Sorted!

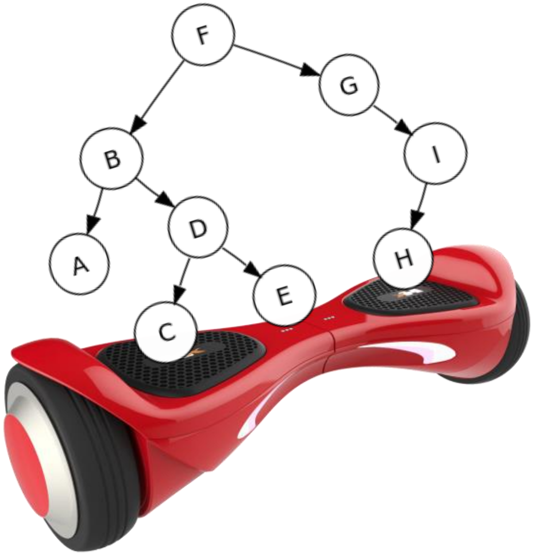
- Runs in time O(n).

Today



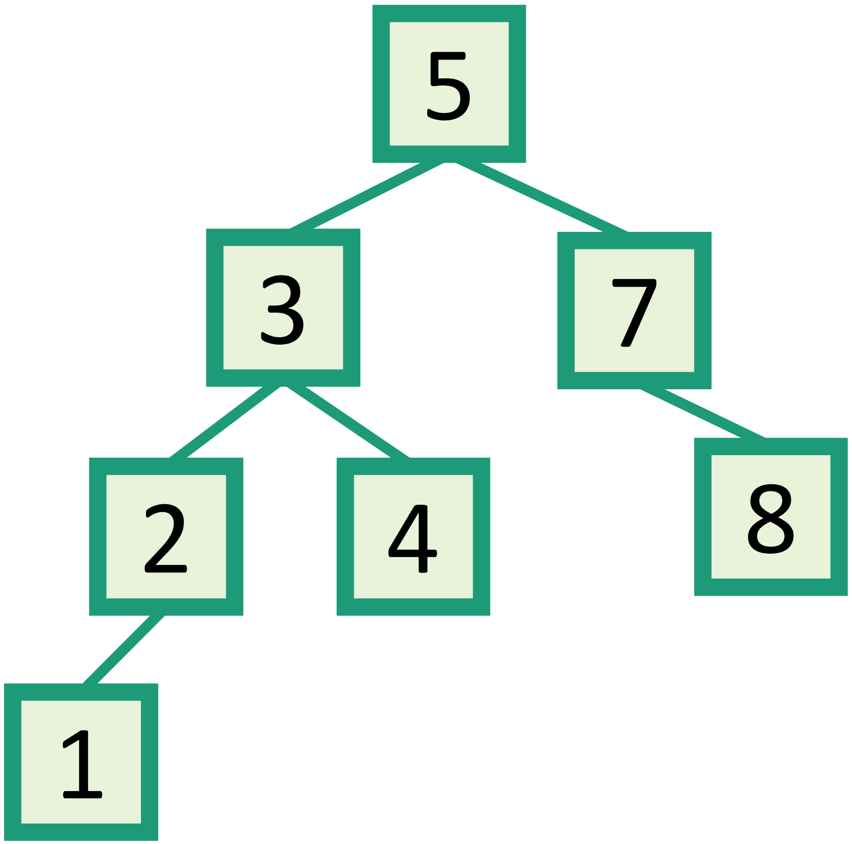
- Begin a brief foray into data structures!
- Binary search trees (cont.)
    - You may remember these from CSE 30
    - They are better when they're balanced.

this will lead us to…

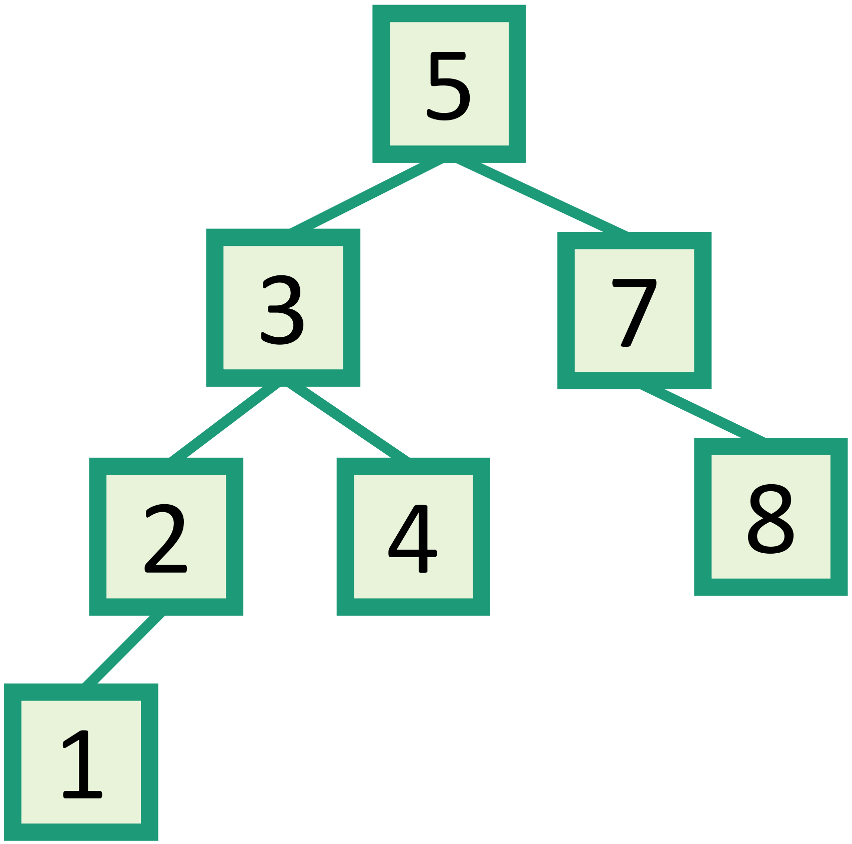


- Self-Balancing Binary Search Trees
    - **Red-Black** trees.

SEARCH in a Binary Search Tree
definition by example

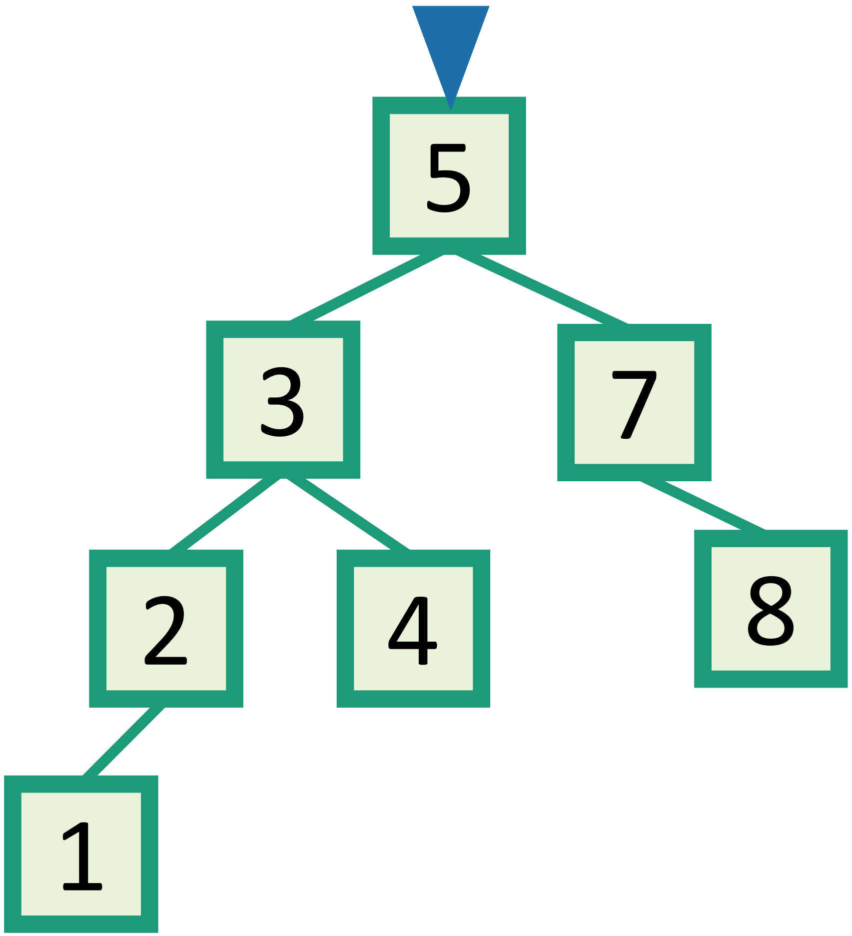


SEARCH in a Binary Search Tree
definition by example



EXAMPLE: Search for 4.

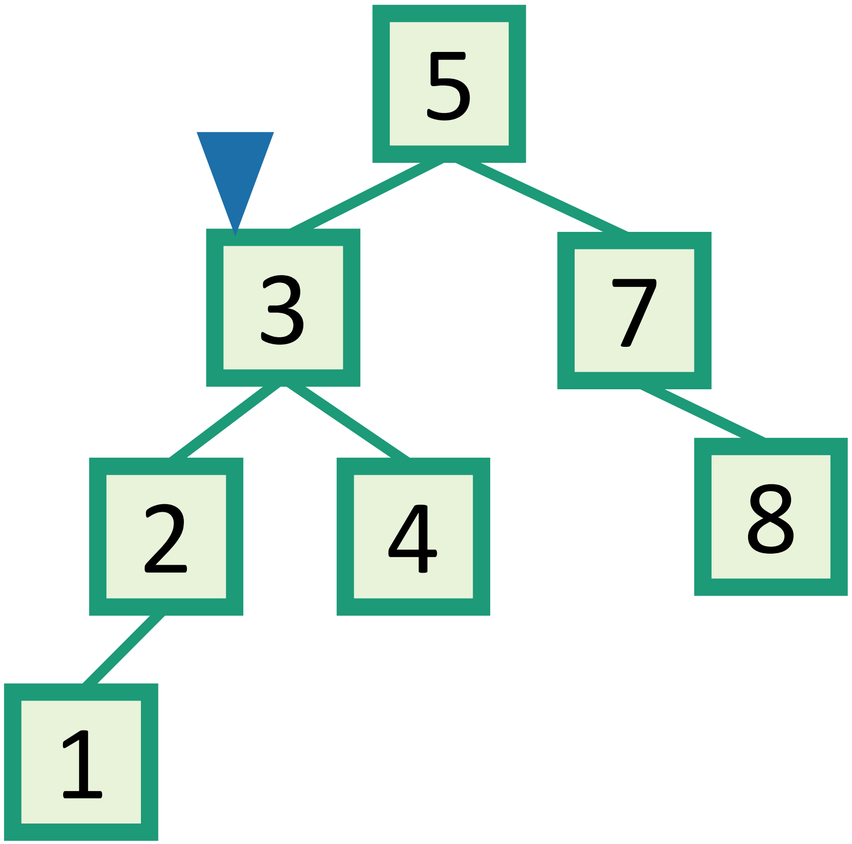
SEARCH in a Binary Search Tree
definition by example



EXAMPLE: Search for 4.

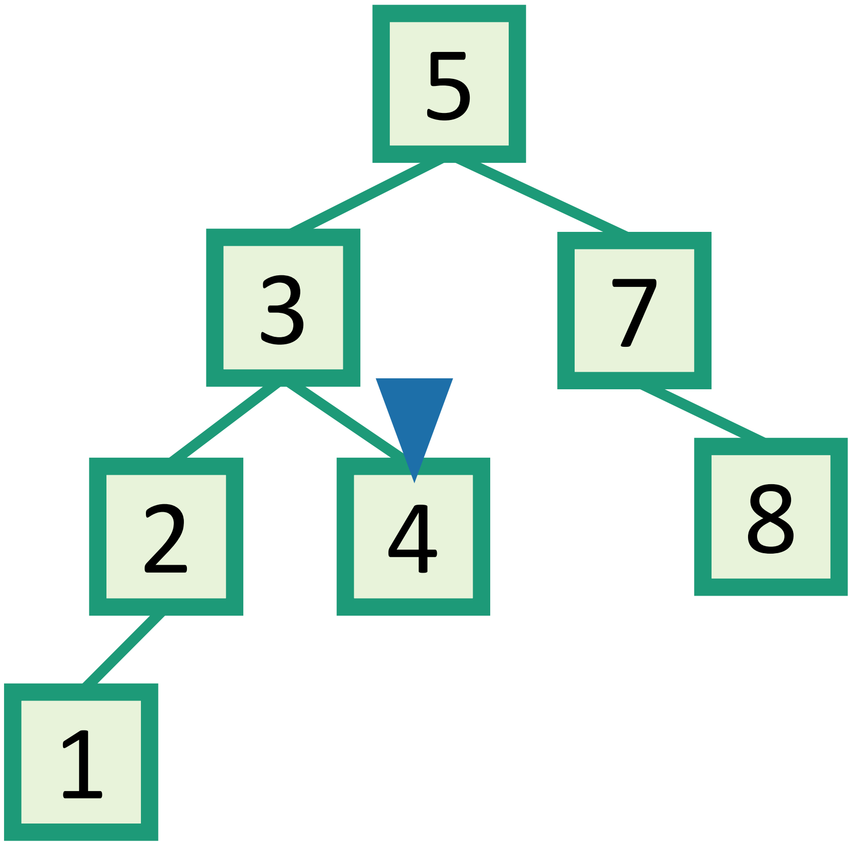
SEARCH in a Binary Search Tree

definition by example



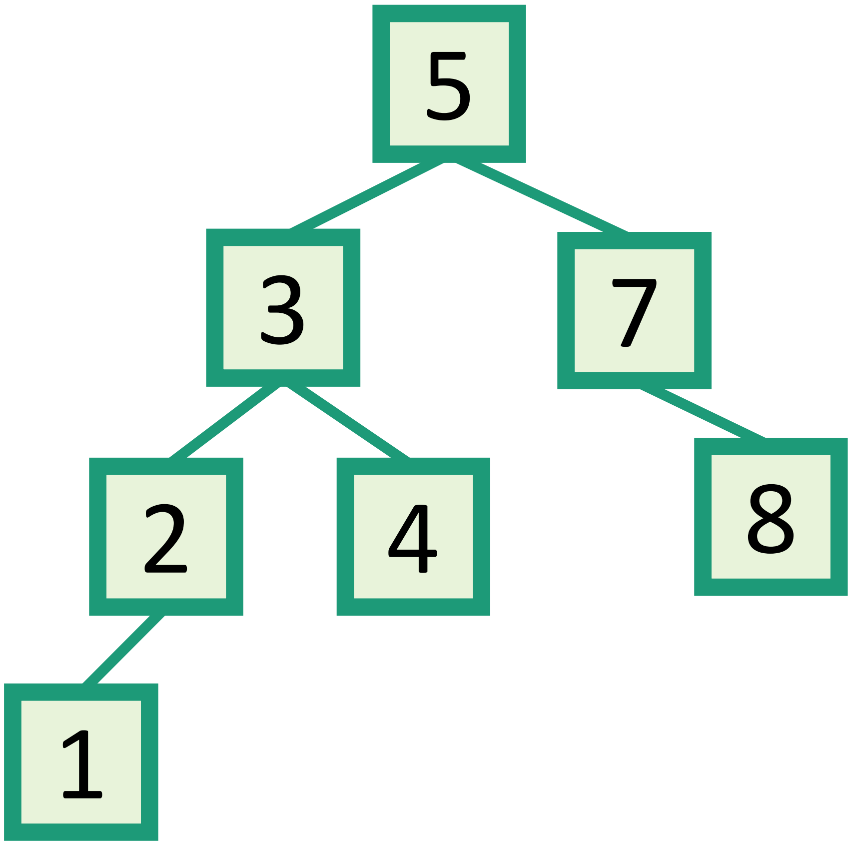
EXAMPLE: Search for 4.

SEARCH in a Binary Search Tree
definition by example



EXAMPLE: Search for 4.

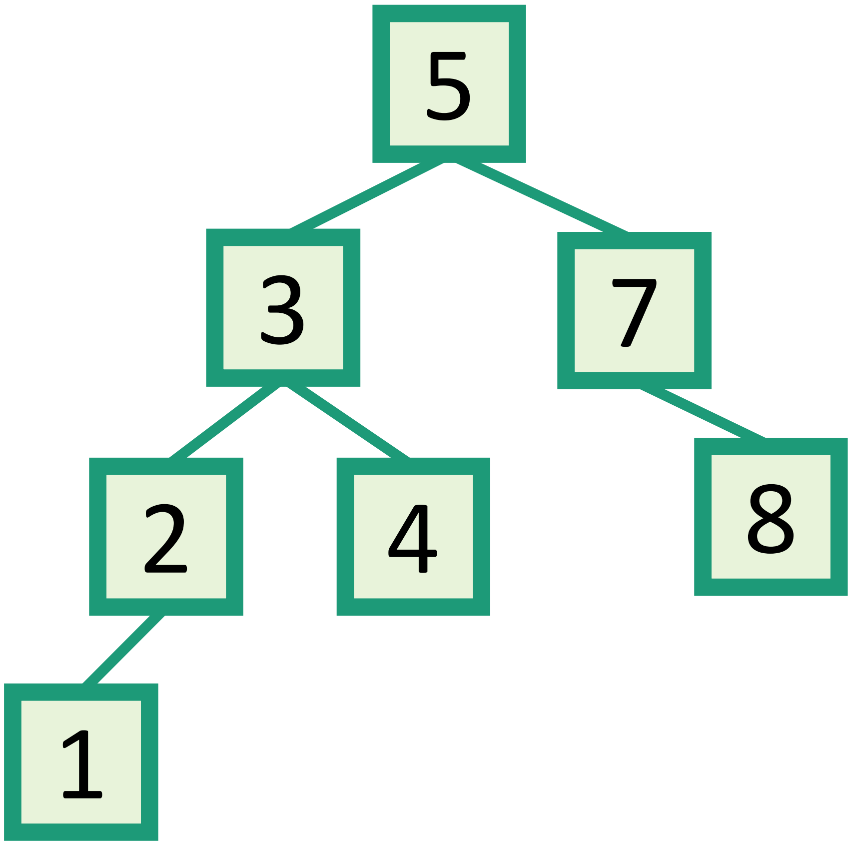
SEARCH in a Binary Search Tree
definition by example



EXAMPLE: Search for 4.

SEARCH in a Binary Search Tree

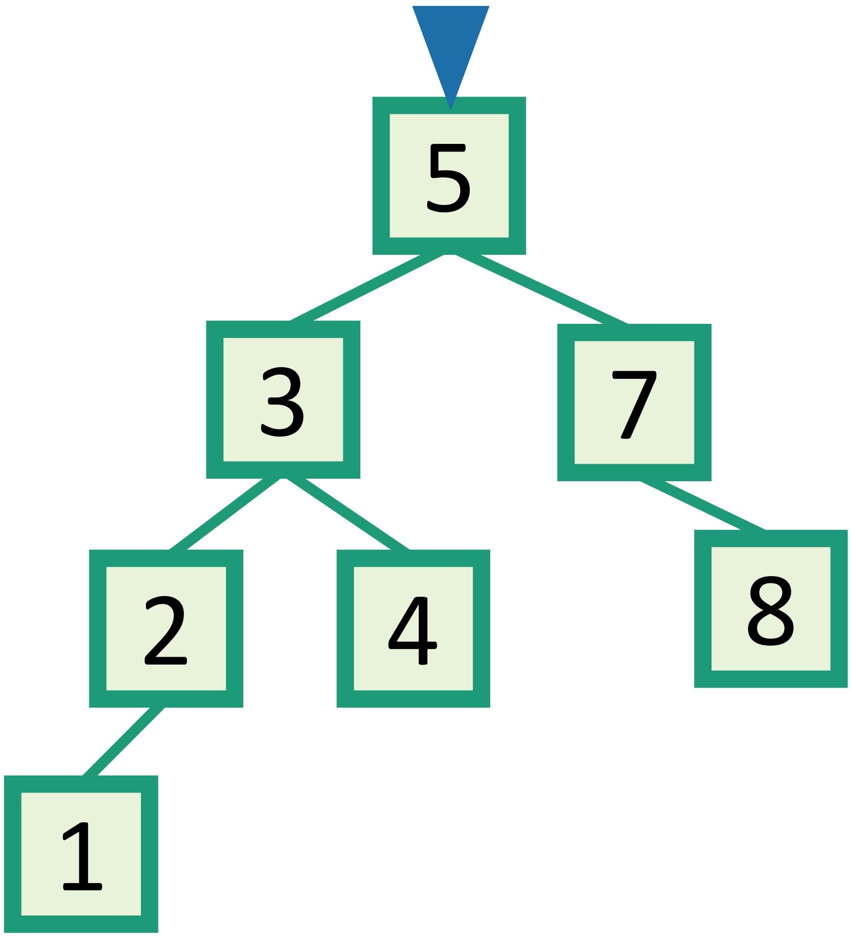
definition by example



EXAMPLE: Search for 4.

EXAMPLE: Search for 4.5

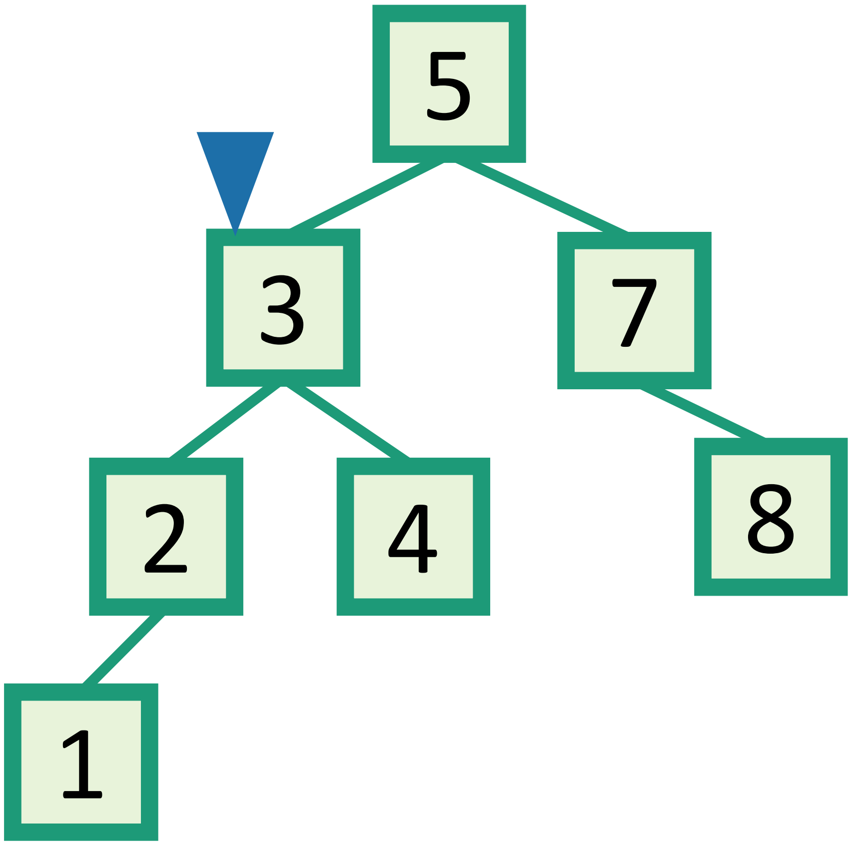
SEARCH in a Binary Search Tree
definition by example



EXAMPLE: Search for 4.

EXAMPLE: Search for 4.5

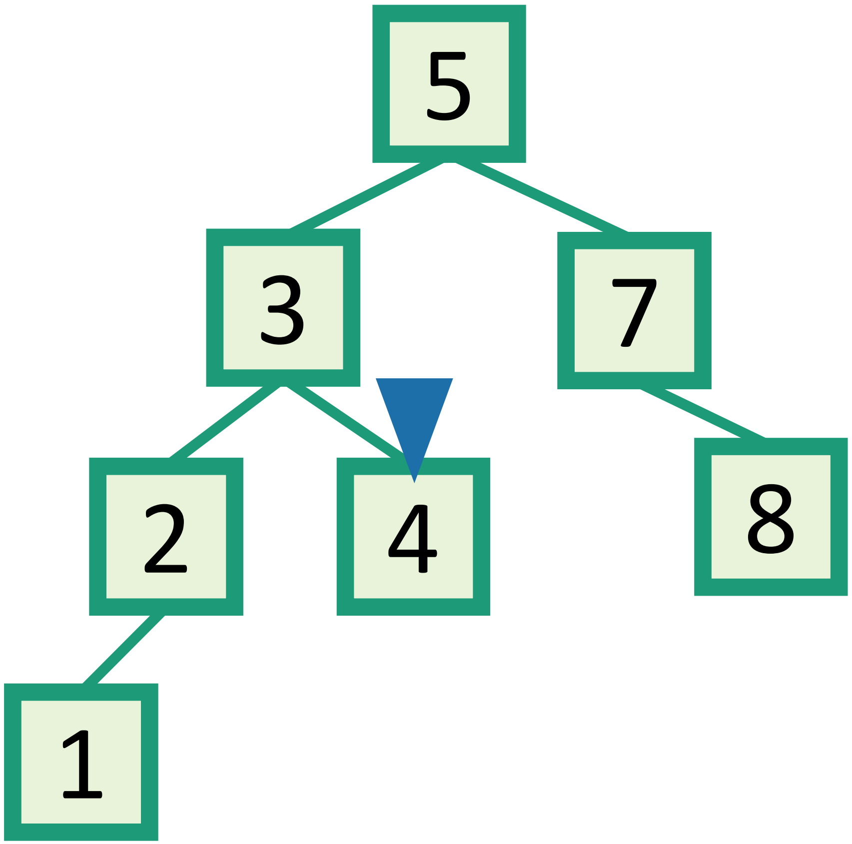
SEARCH in a Binary Search Tree
definition by example



EXAMPLE: Search for 4.

EXAMPLE: Search for 4.5

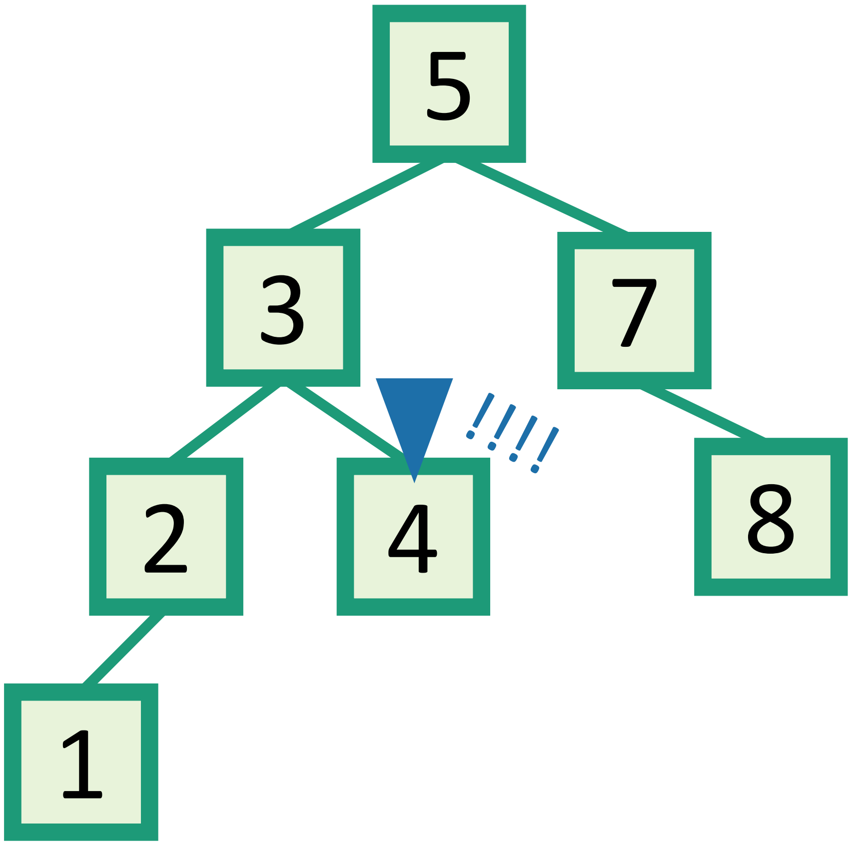
SEARCH in a Binary Search Tree
definition by example



EXAMPLE: Search for 4.

EXAMPLE: Search for 4.5

SEARCH in a Binary Search Tree
definition by example

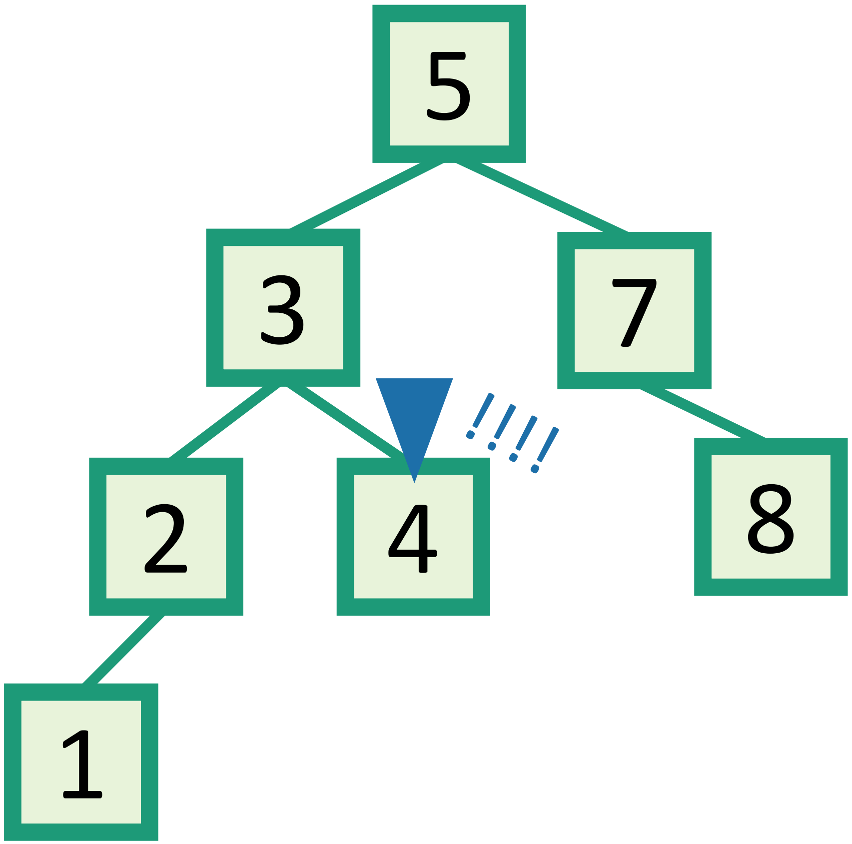


EXAMPLE: Search for 4.

EXAMPLE: Search for 4.5

SEARCH in a Binary Search Tree

definition by example



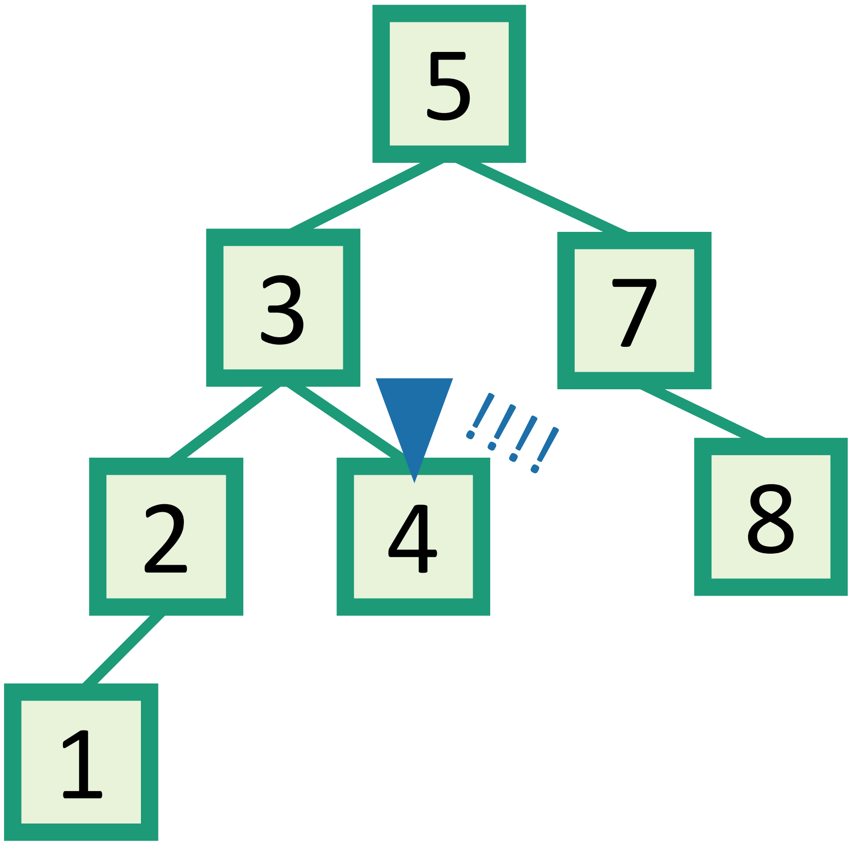
EXAMPLE: Search for 4.

EXAMPLE: Search for 4.5

- It turns out it will be convenient to **return 4** in this case

SEARCH in a Binary Search Tree

definition by example



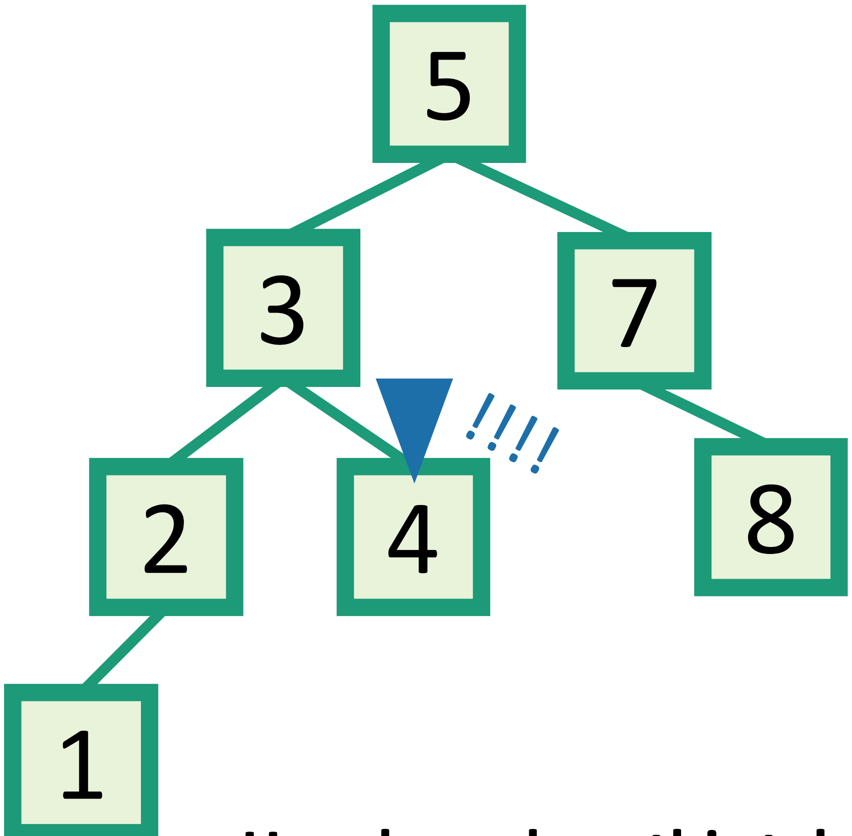
EXAMPLE: Search for 4.

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)

SEARCH in a Binary Search Tree

definition by example



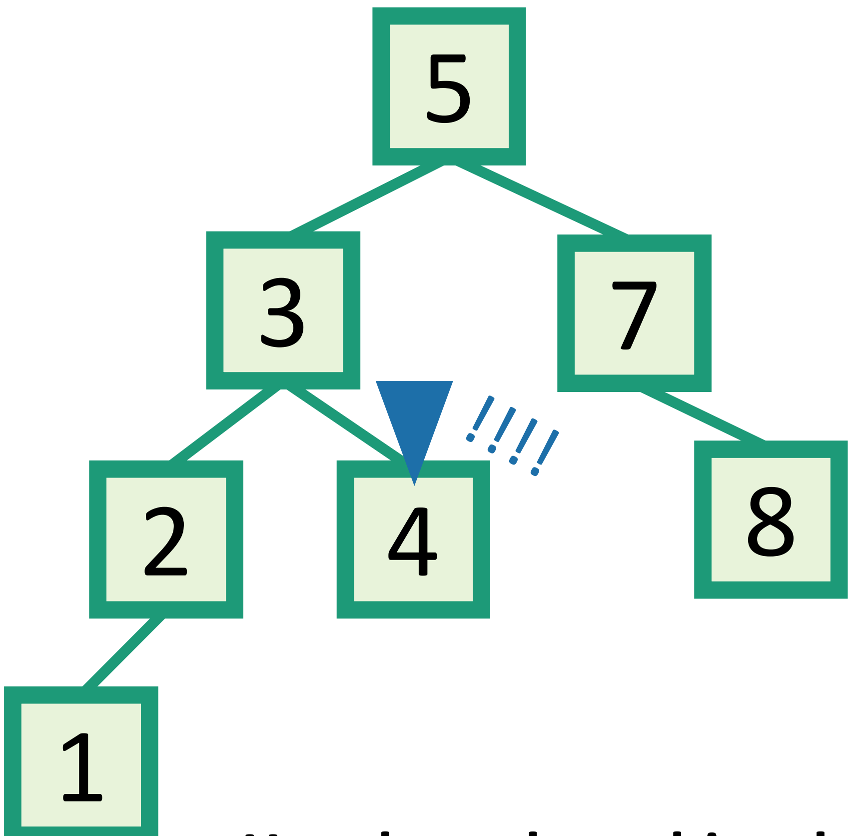
EXAMPLE: Search for 4.

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?

SEARCH in a Binary Search Tree
definition by example



EXAMPLE: Search for 4.

EXAMPLE: Search for 4.5

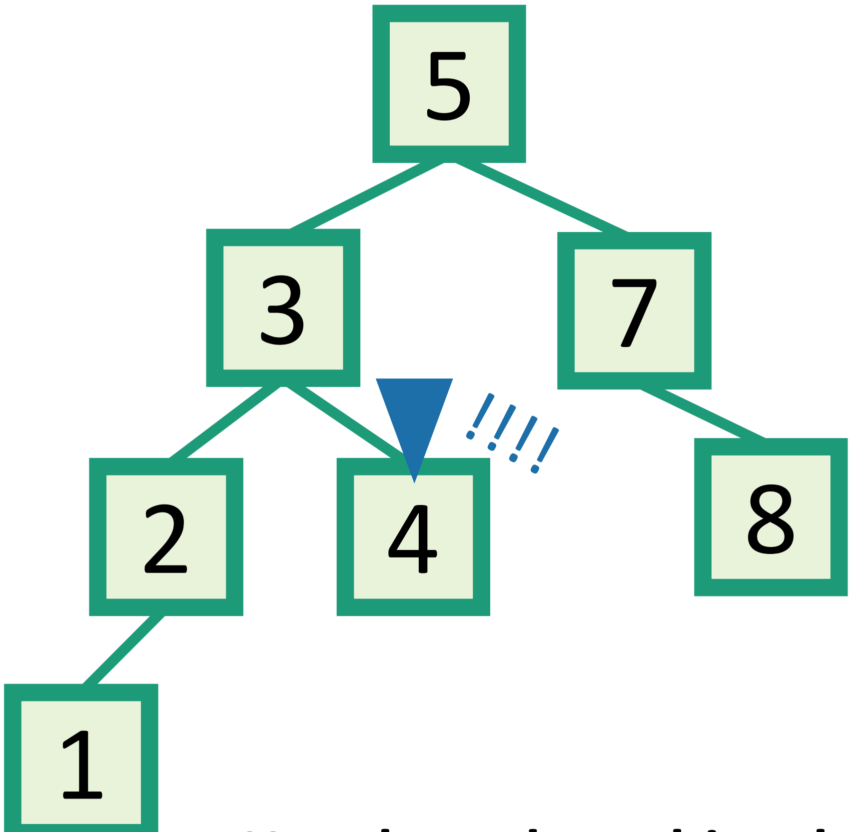
- It turns out it will be convenient to **return 4** in this case
- (that is, **return** the last node before we went off the tree)

How long does this take?

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

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)

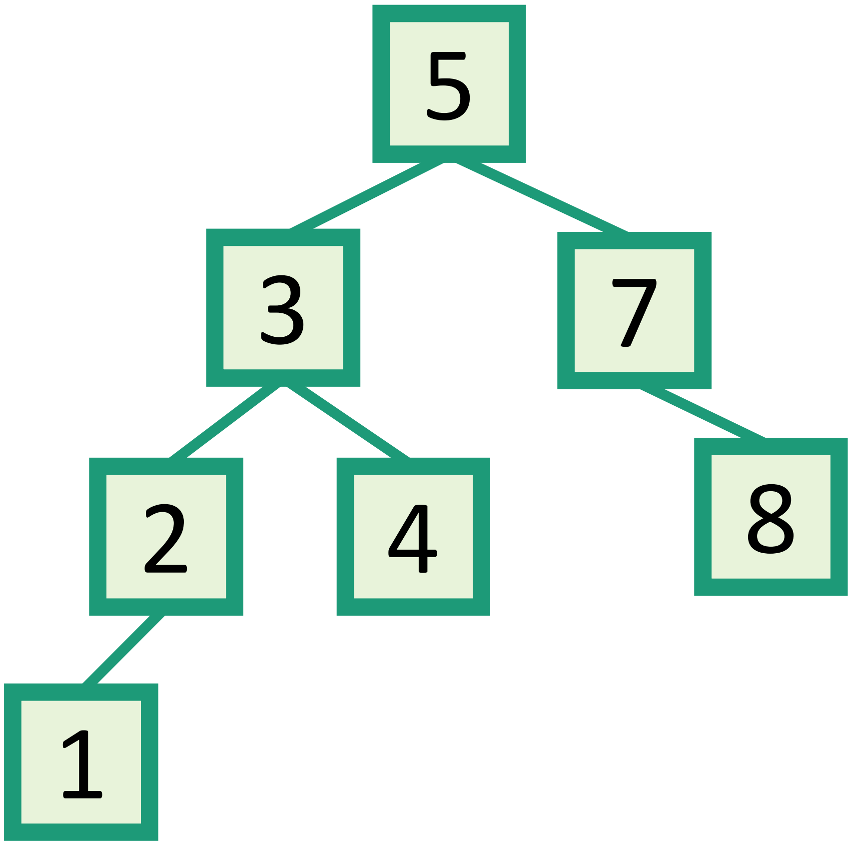
How long does this take?

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

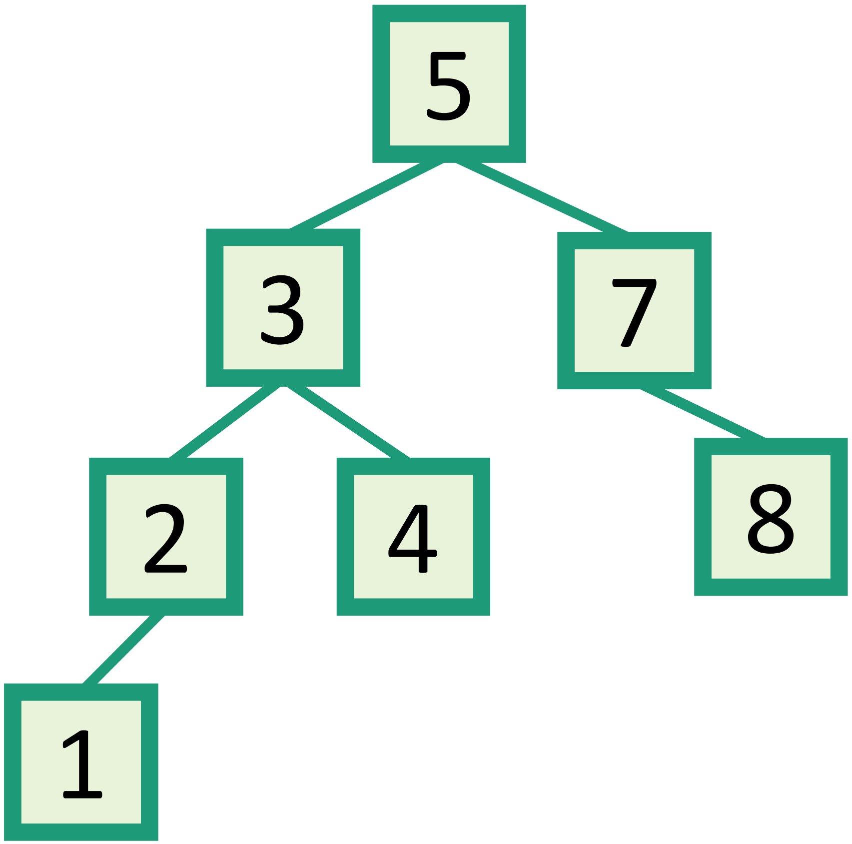
Write pseudocode (or actual code) to implement this!

Ollie the over-achieving ostrich

INSERT in a Binary Search Tree

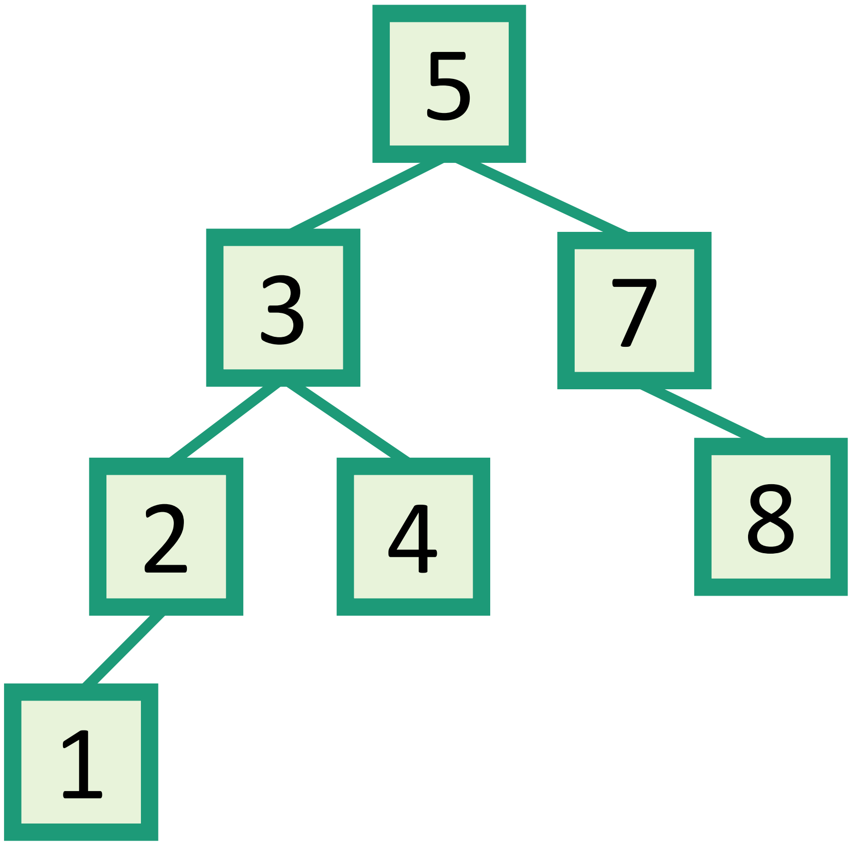


INSERT in a Binary Search Tree



EXAMPLE: Insert 4.5

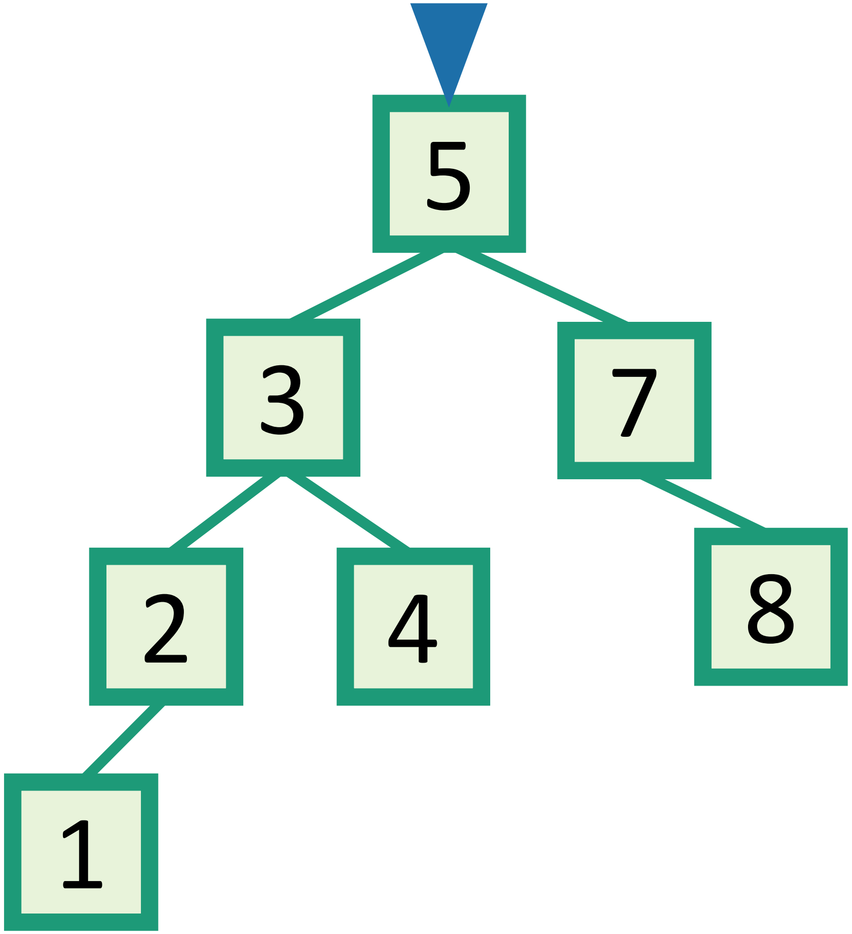
INSERT in a Binary Search Tree



EXAMPLE: Insert 4.5

- INSERT(key):

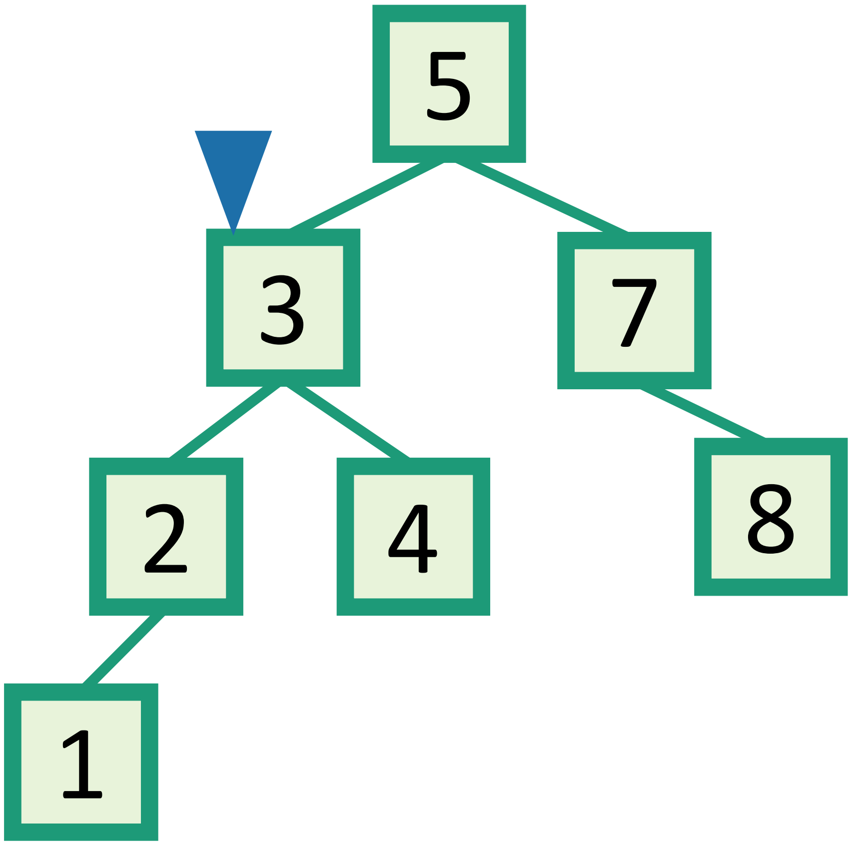
INSERT in a Binary Search Tree



EXAMPLE: Insert 4.5

- INSERT(key):
  - x = SEARCH(key)

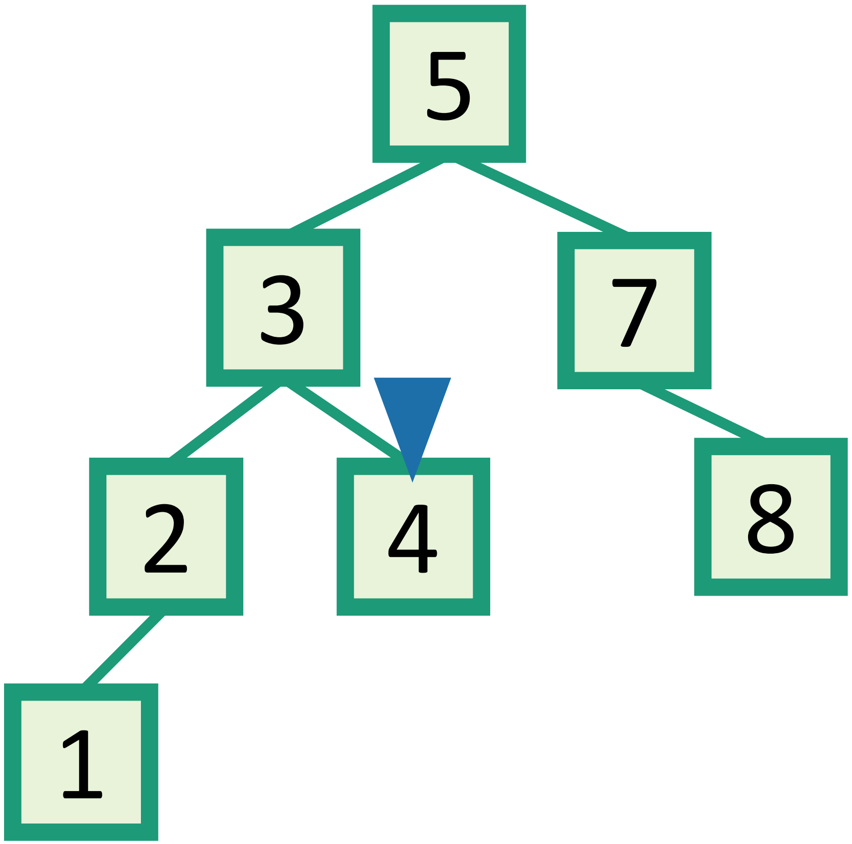
INSERT in a Binary Search Tree



EXAMPLE: Insert 4.5

- INSERT(key):
  - x = SEARCH(key)

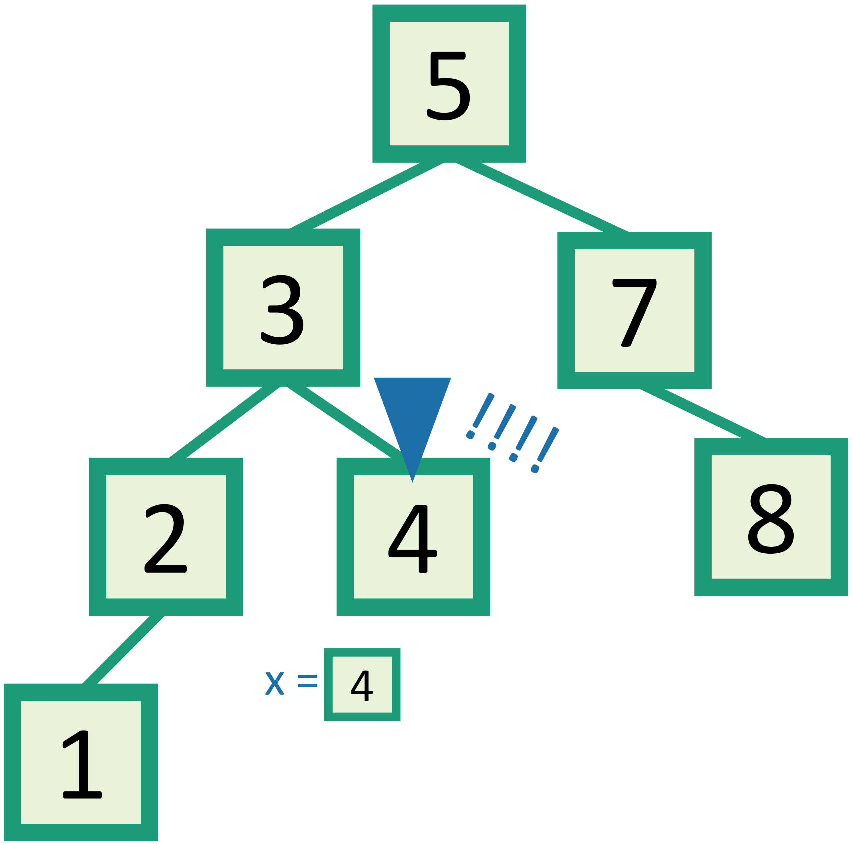
INSERT in a Binary Search Tree



EXAMPLE: Insert 4.5

- INSERT(key):
  - x = SEARCH(key)

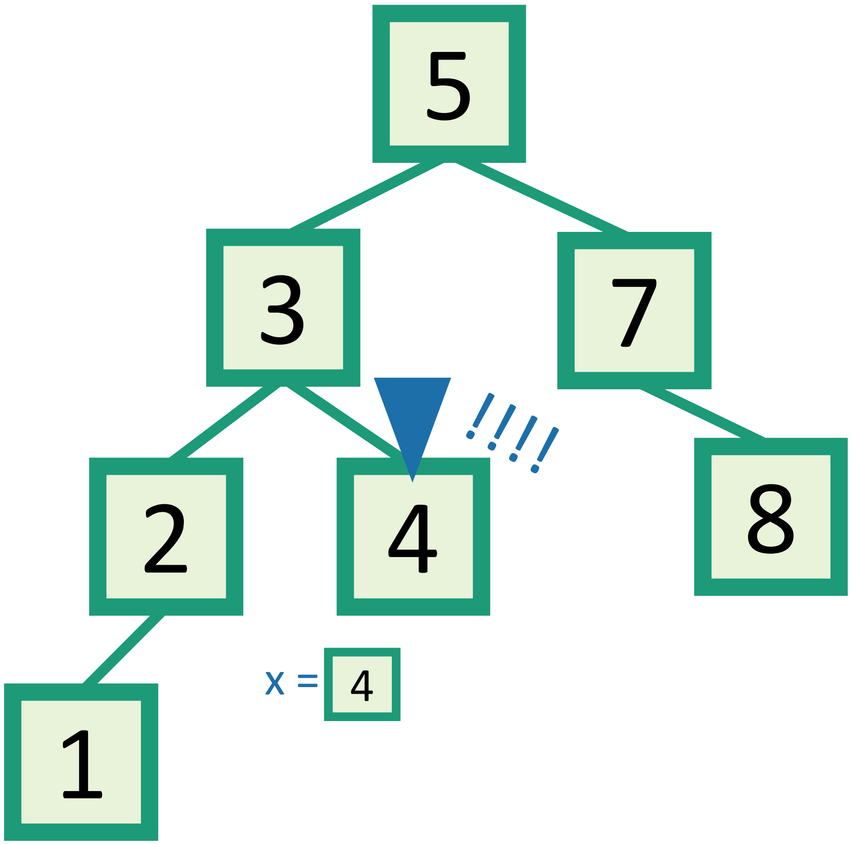
INSERT in a Binary Search Tree



EXAMPLE: Insert 4.5

- INSERT(key):
  - x = SEARCH(key)

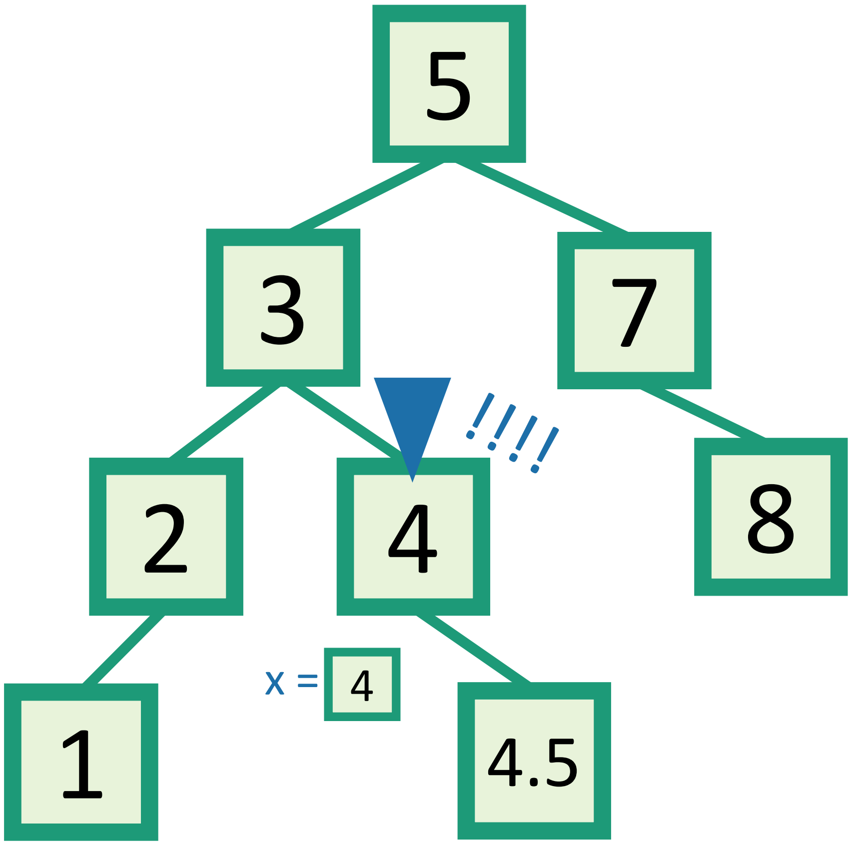
INSERT in a Binary Search Tree



EXAMPLE: Insert 4.5

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

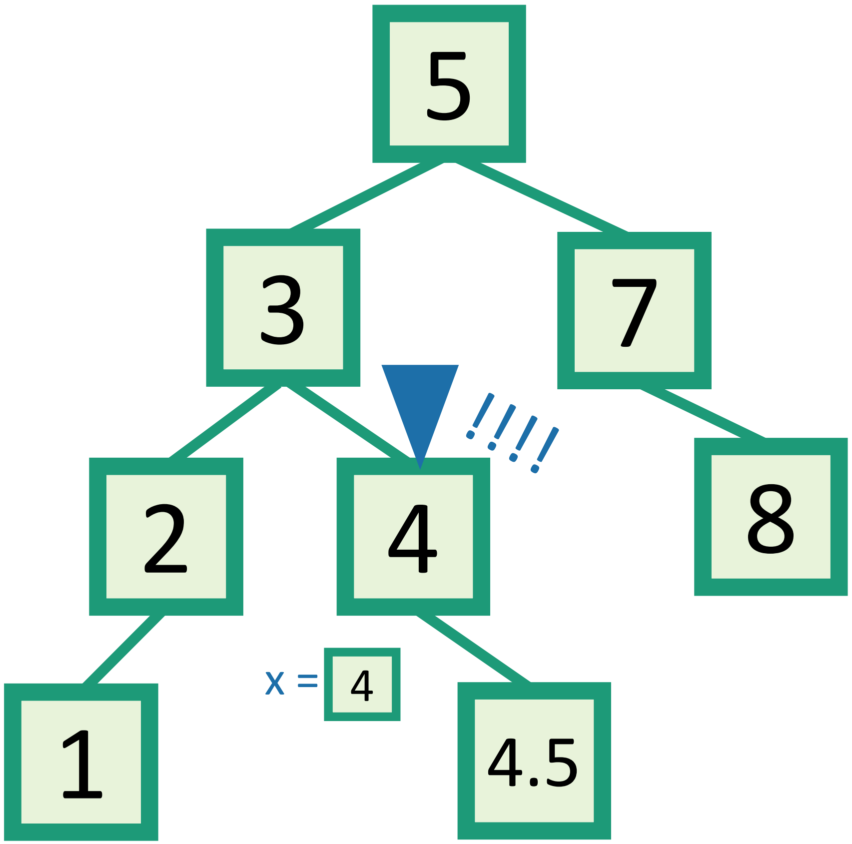
INSERT in a Binary Search Tree



EXAMPLE: Insert 4.5

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

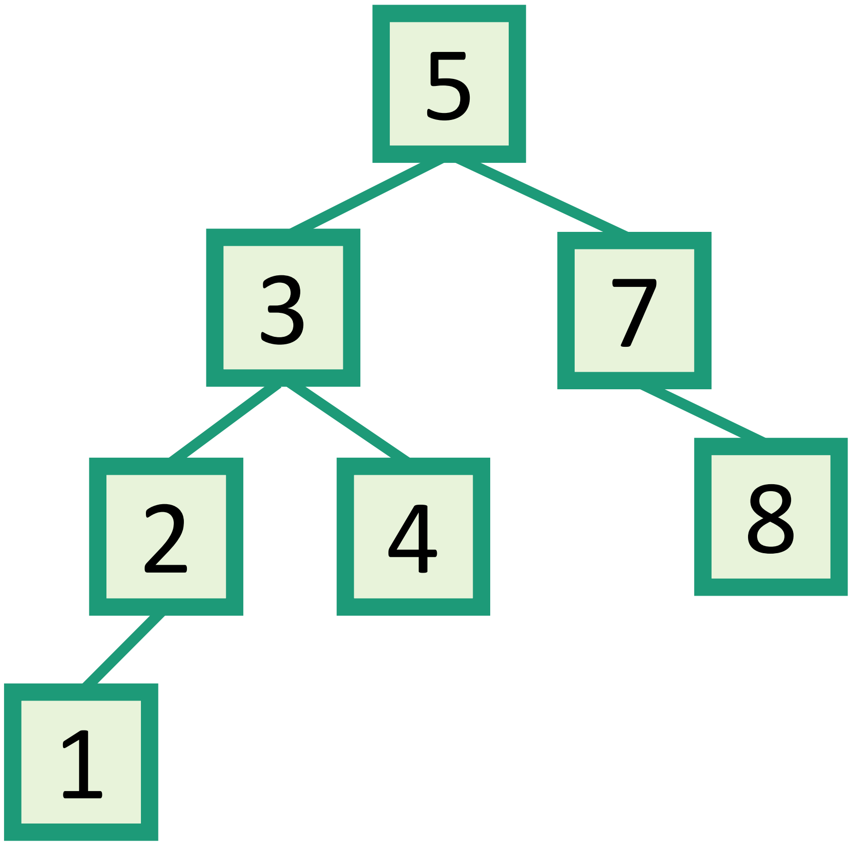
INSERT in a Binary Search Tree



EXAMPLE: Insert 4.5

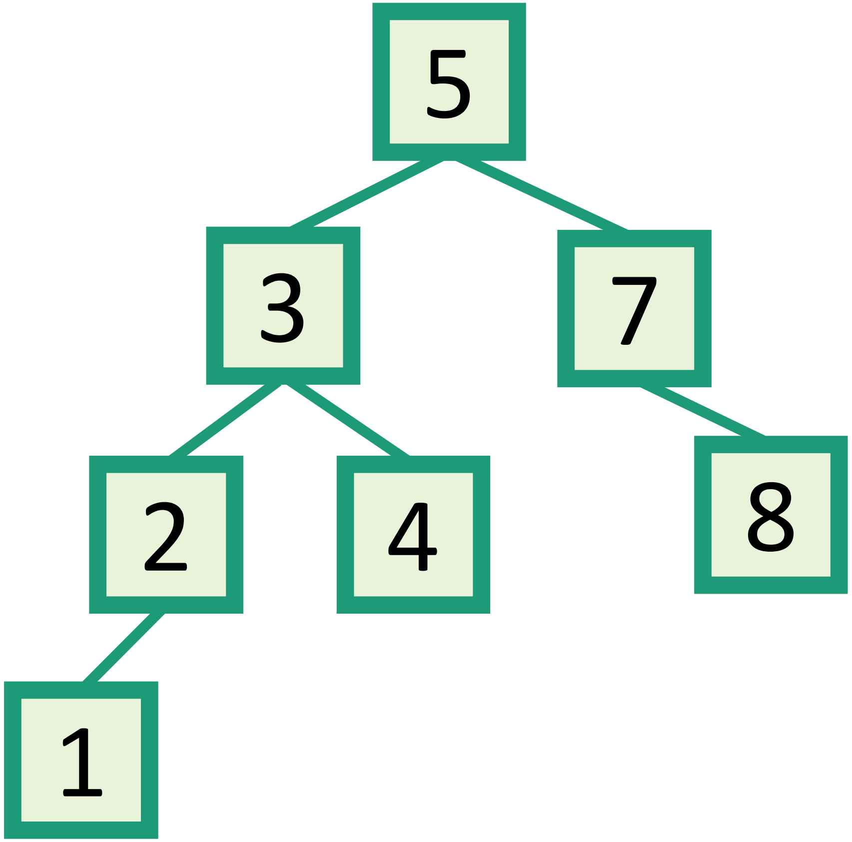
- INSERT(key):
  - x = SEARCH(key)
  - **if** key > x.key:
    - Make a new node with the correct key, and put it as the right child of x.
  - **if** key < x.key:
    - Make a new node with the correct key, and put it as the left child of x.
  - **if** x.key == key:
    - **return**

DELETE in a Binary Search Tree

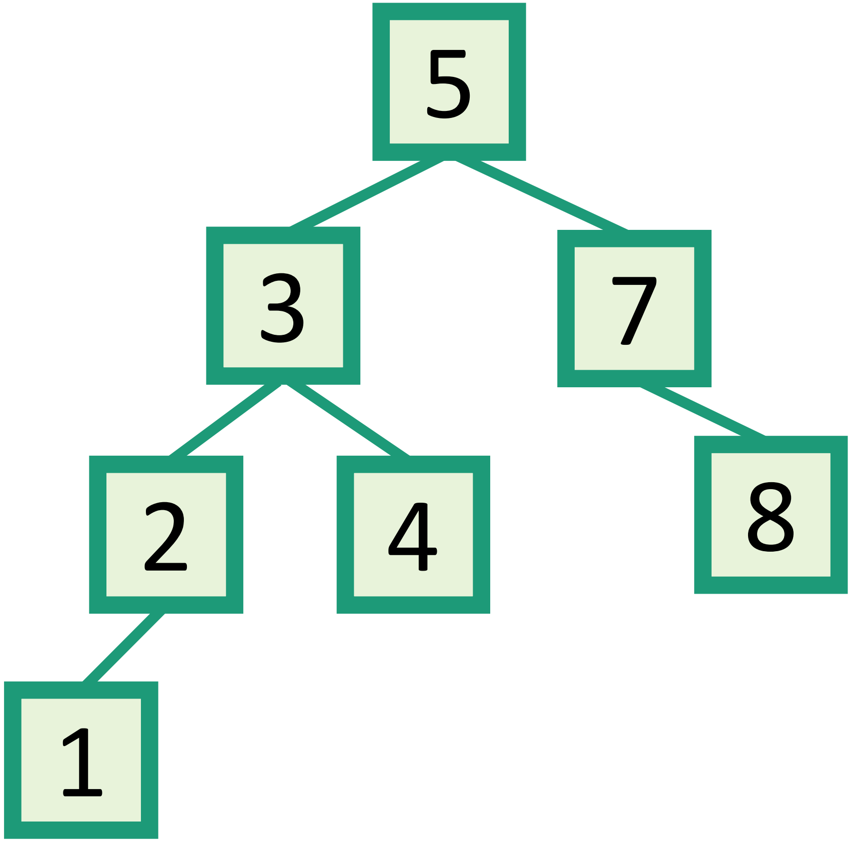


DELETE in a Binary Search Tree

EXAMPLE: Delete 2



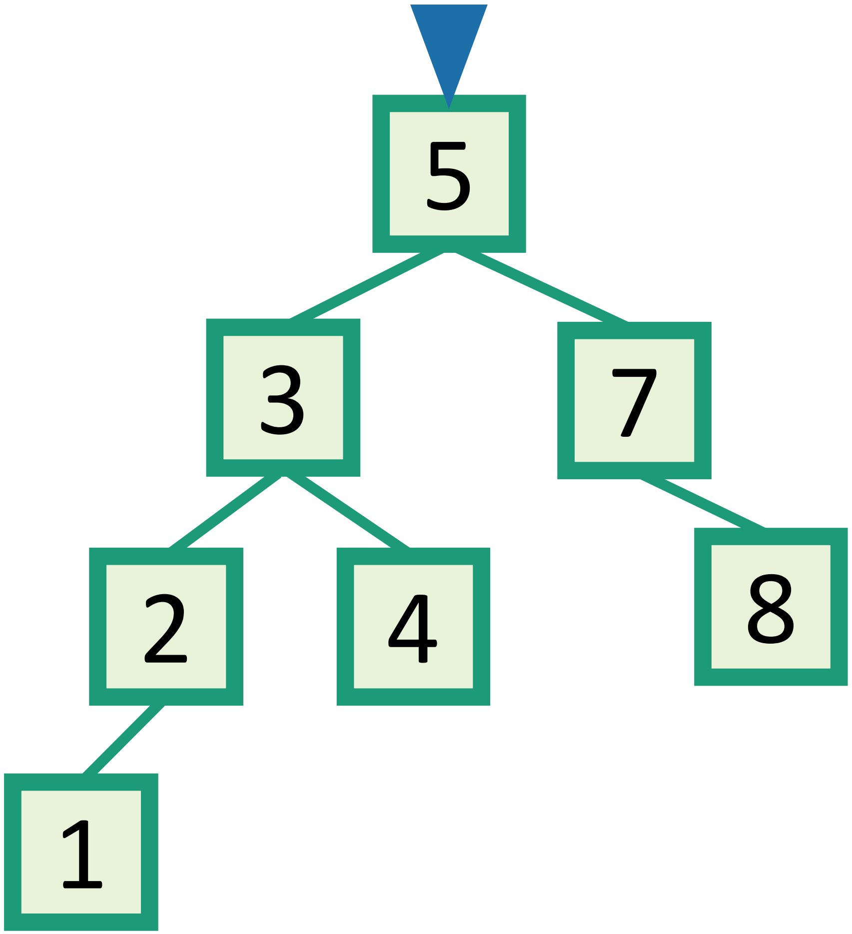
DELETE in a Binary Search Tree



EXAMPLE: Delete 2

- DELETE(key):

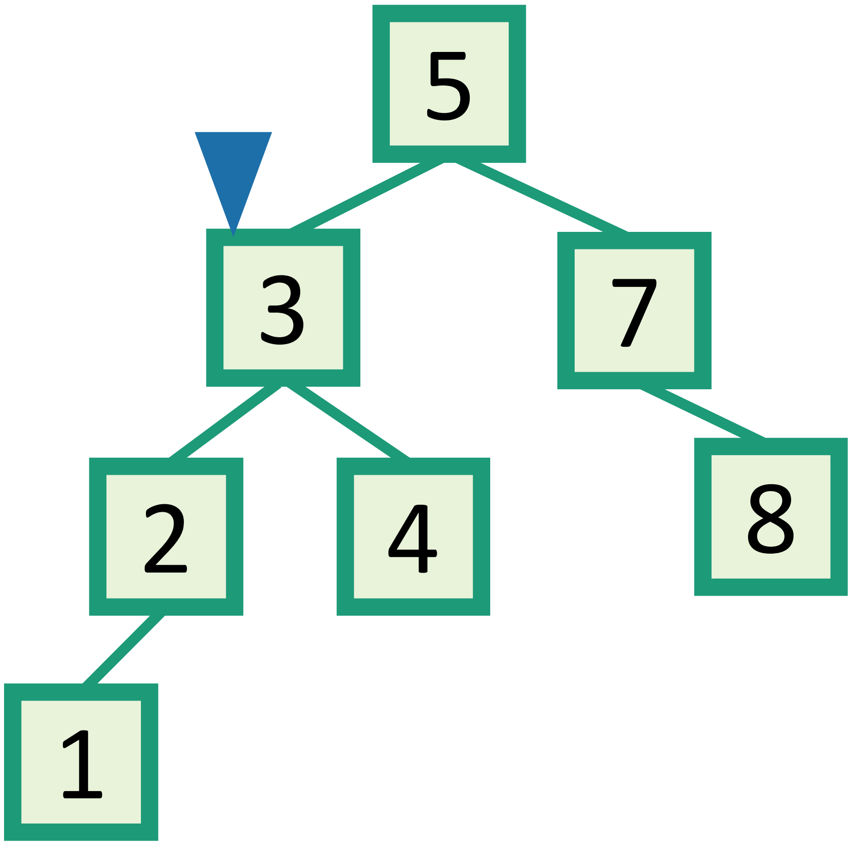
DELETE in a Binary Search Tree



EXAMPLE: Delete 2

- DELETE(key):
  - x = SEARCH(key)

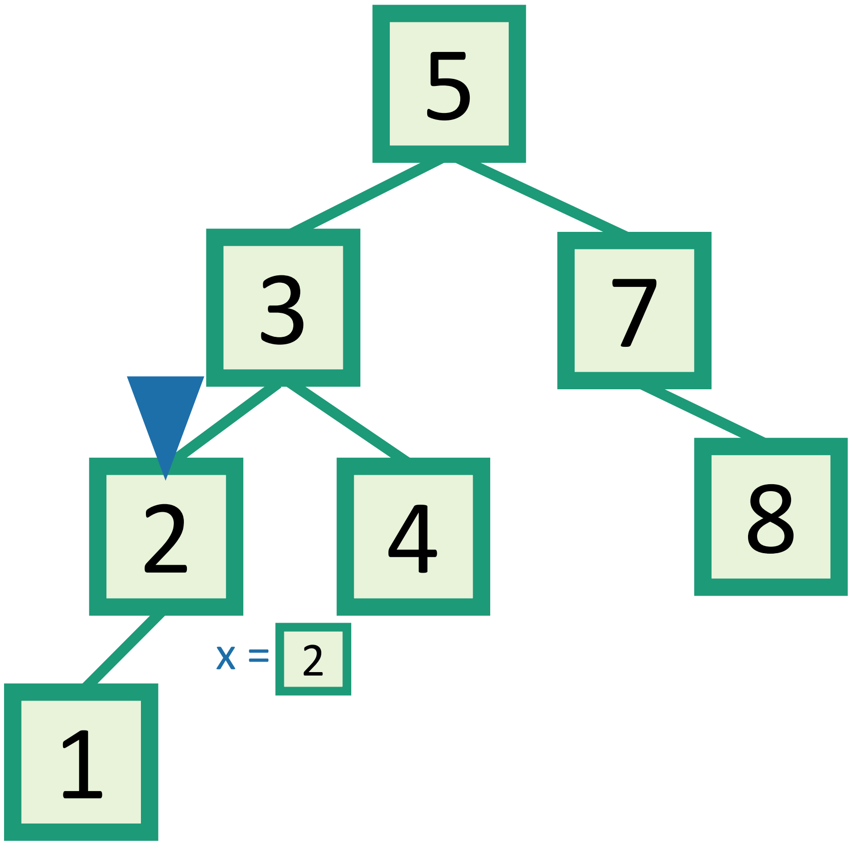
DELETE in a Binary Search Tree



EXAMPLE: Delete 2

- DELETE(key):
  - x = SEARCH(key)

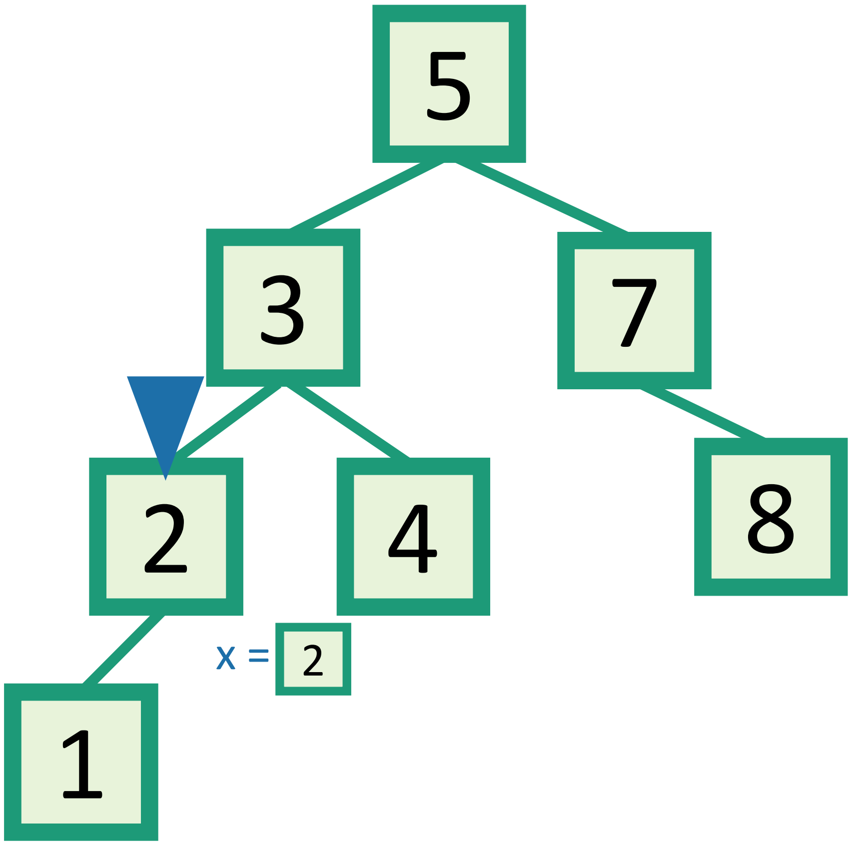
DELETE in a Binary Search Tree



EXAMPLE: Delete 2

- DELETE(key):
  - x = SEARCH(key)

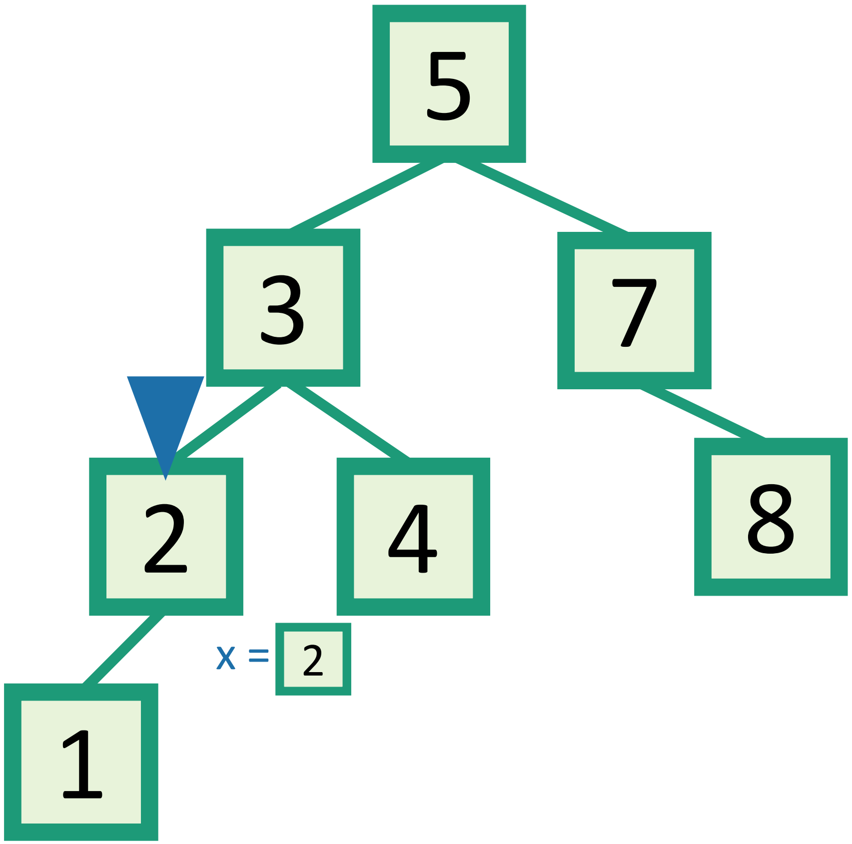
DELETE in a Binary Search Tree



EXAMPLE: Delete 2

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

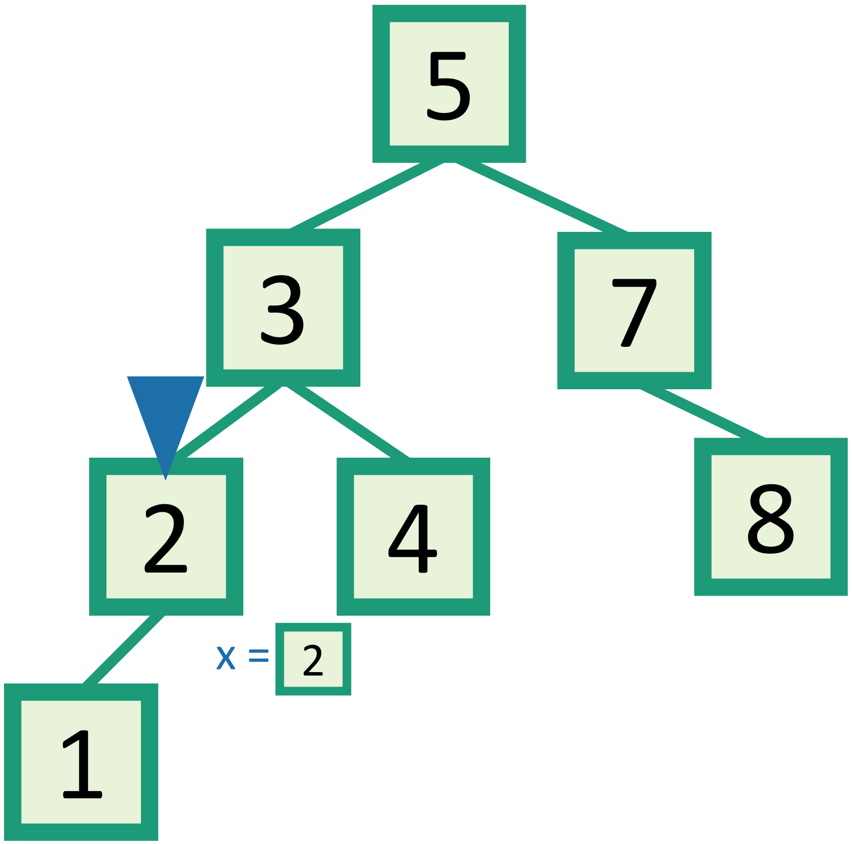
DELETE in a Binary Search Tree



EXAMPLE: Delete 2

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

DELETE in a Binary Search Tree

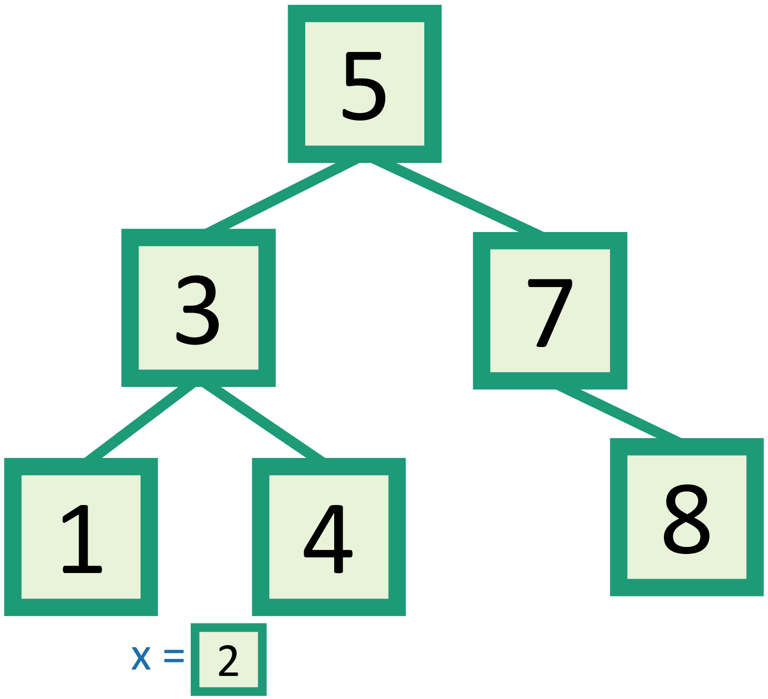


EXAMPLE: Delete 2

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

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

DELETE in a Binary Search Tree



EXAMPLE: Delete 2

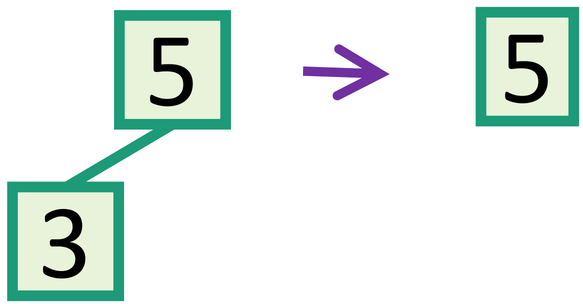
- DELETE(key):
  - x = SEARCH(key)
  - **if** x.key == key:
    - ….delete x….

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

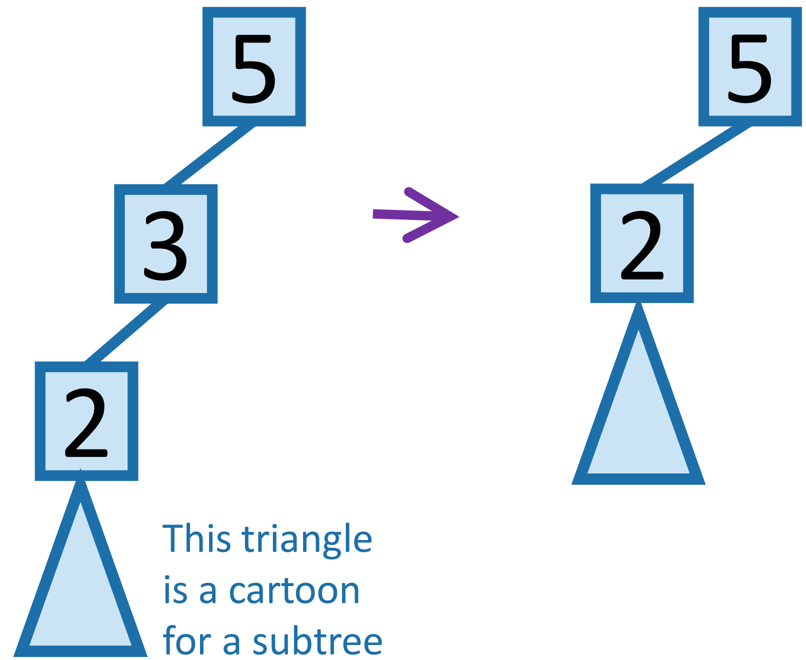
DELETE in a Binary Search Tree

several cases (by example)

say we want to delete 3



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



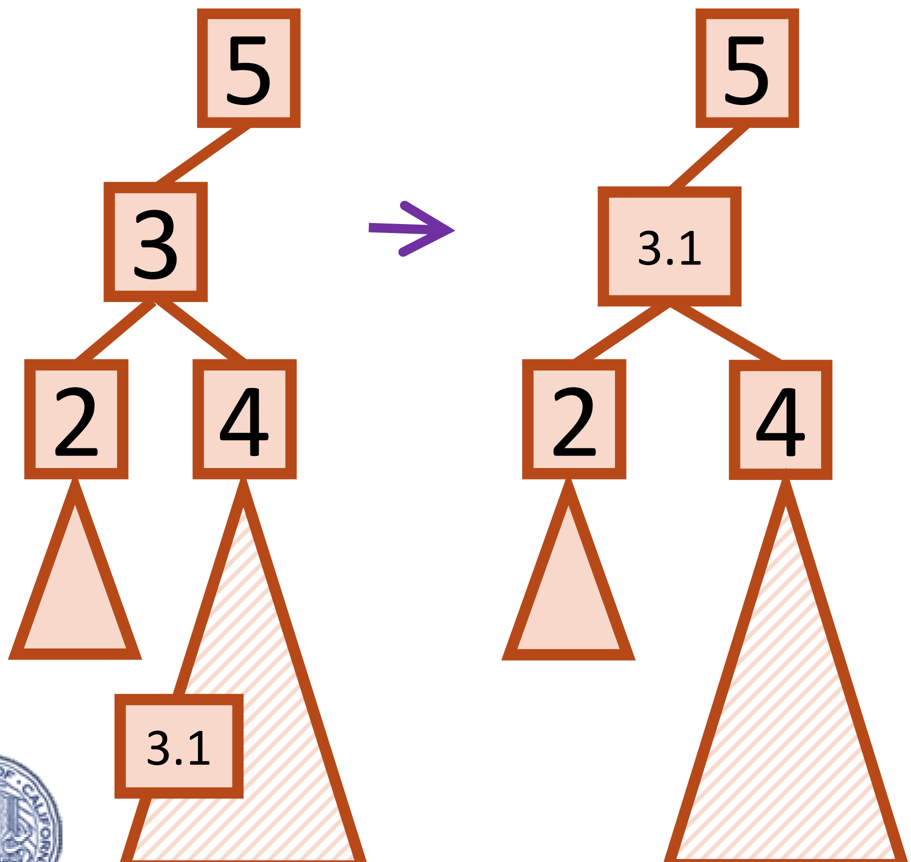
Case 2: if 3 has just one child, move that up.

Write pseudocode for all of these!

Siggi the Studious Stork

DELETE in a Binary Search Tree

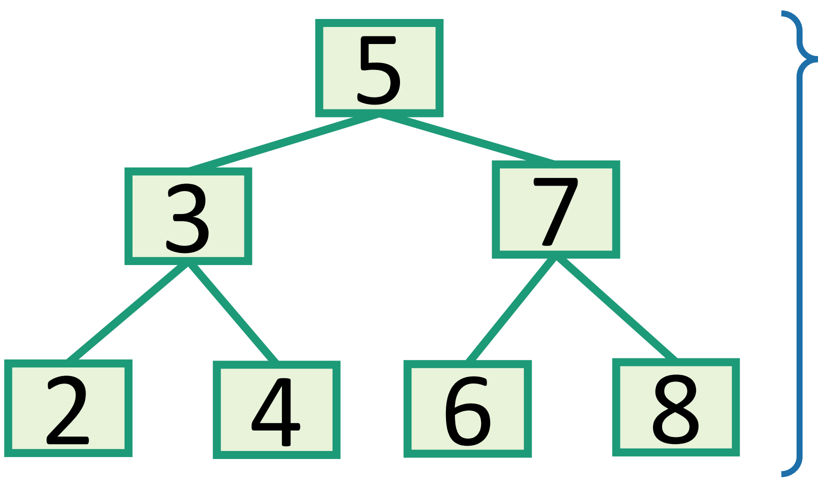
Case 3: if 3 has two children,
replace 3 with it's immediate successor.
(aka, next biggest thing after 3)



- 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)

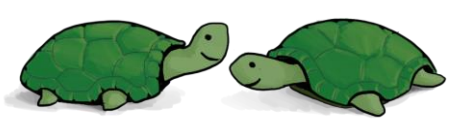
Trees have depth O(log(n)). **Done!**

Lucky the Lackadaisical Lemur

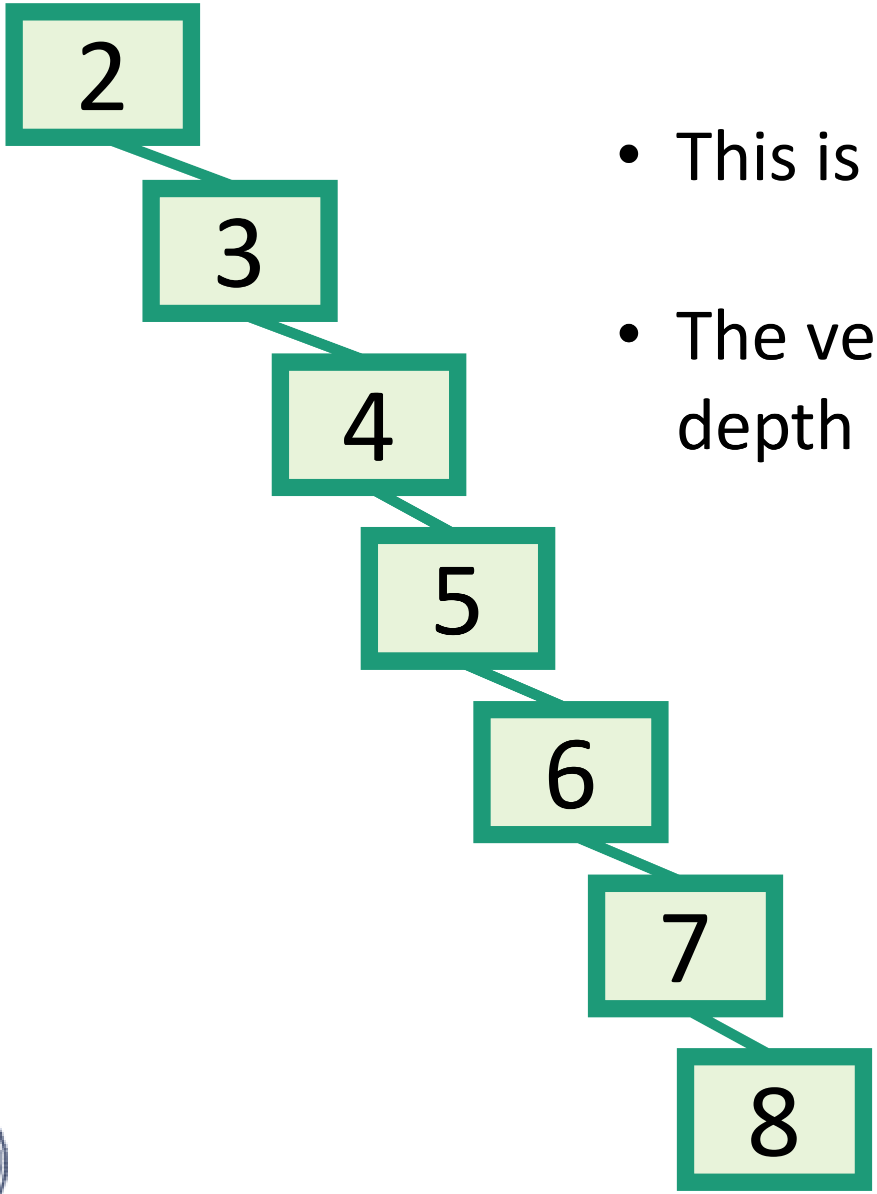
Wait a second…

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!

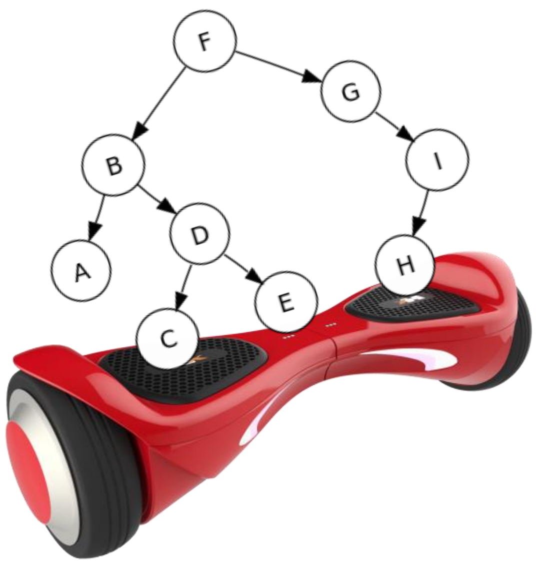
Ollie the over-achieving ostrich

- 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....

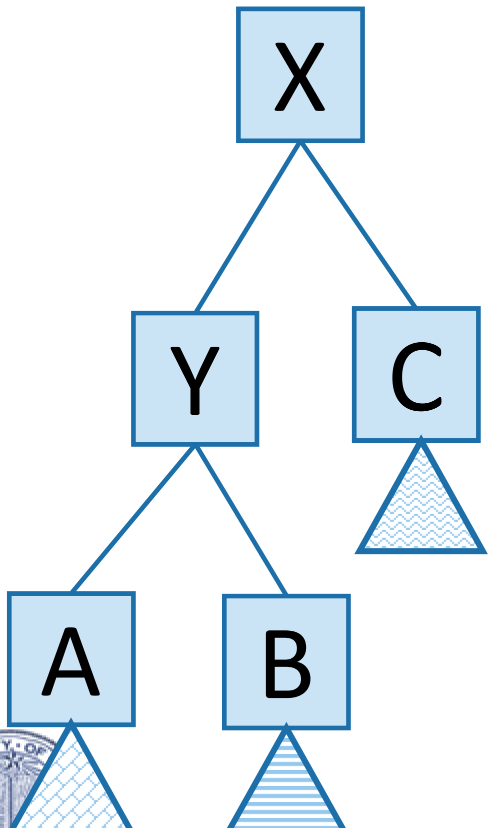
- Turns out that's not a great idea. Instead we turn to…

Self-Balancing Binary Search Trees



Idea 1: Rotations

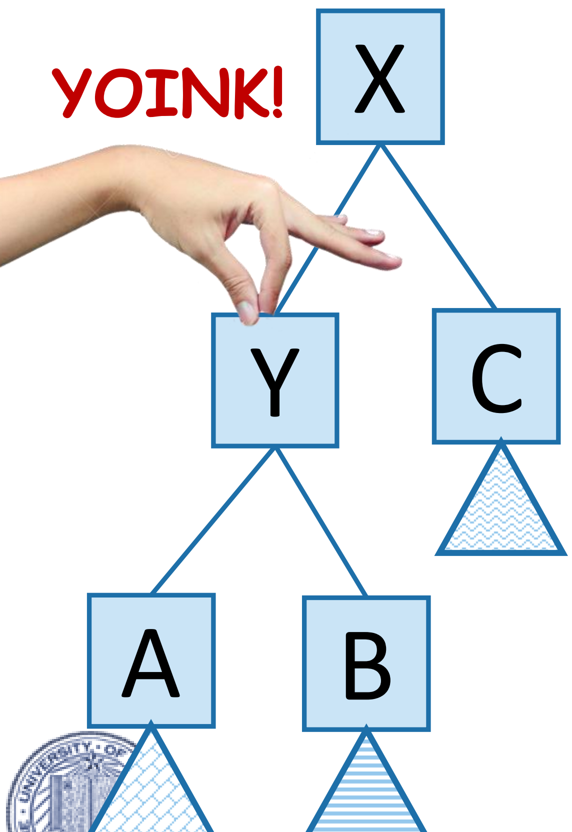
- Maintain Binary Search Tree (BST) property, while moving stuff around.



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

Idea 1: Rotations

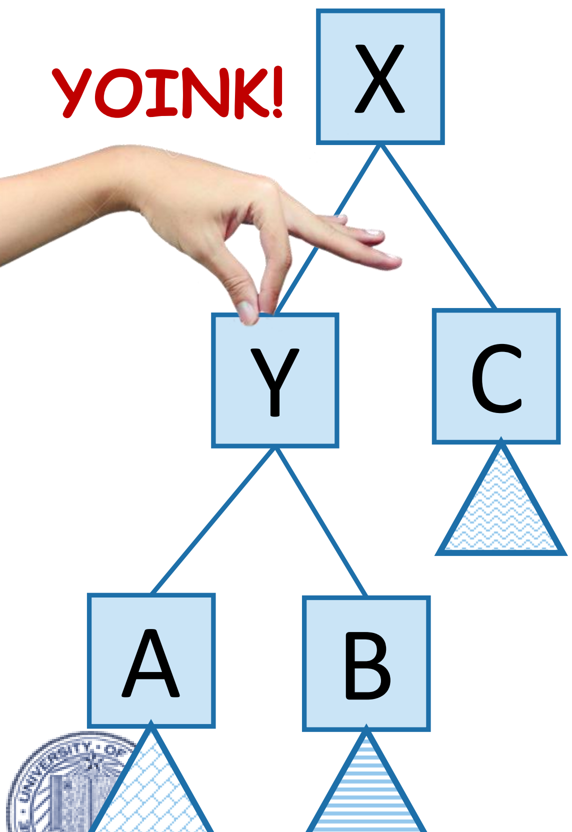
- Maintain Binary Search Tree (BST) property, while moving stuff around.



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

Idea 1: Rotations

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

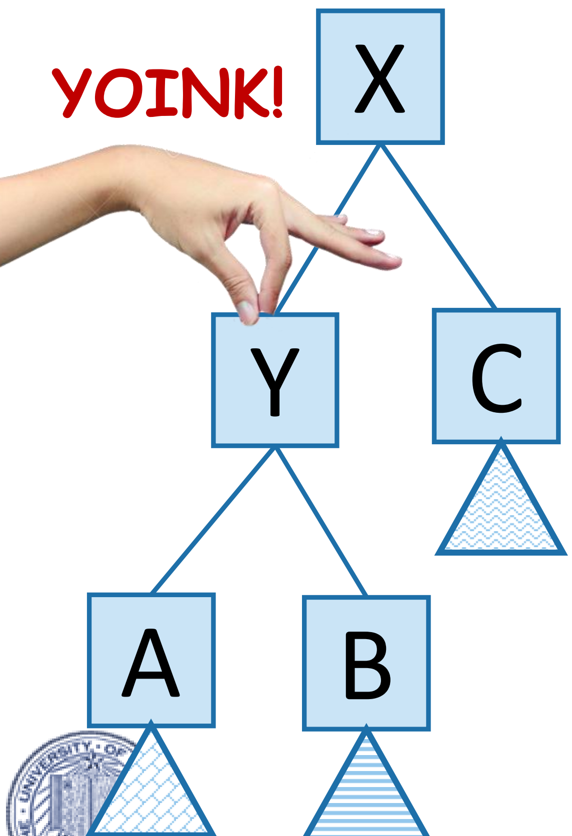


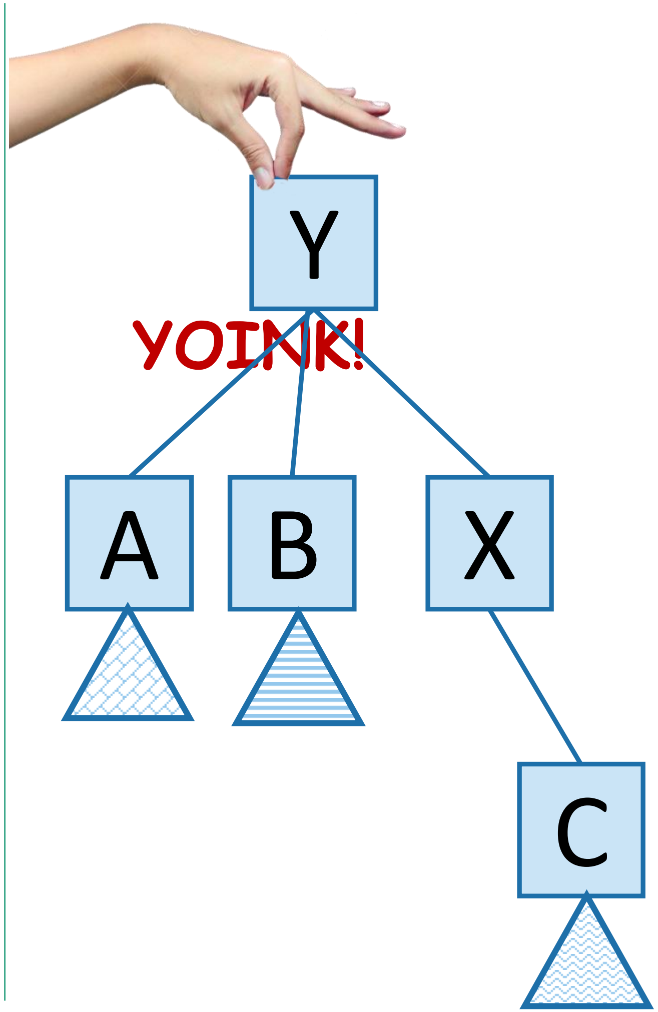
YOINK!

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

Idea 1: Rotations

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

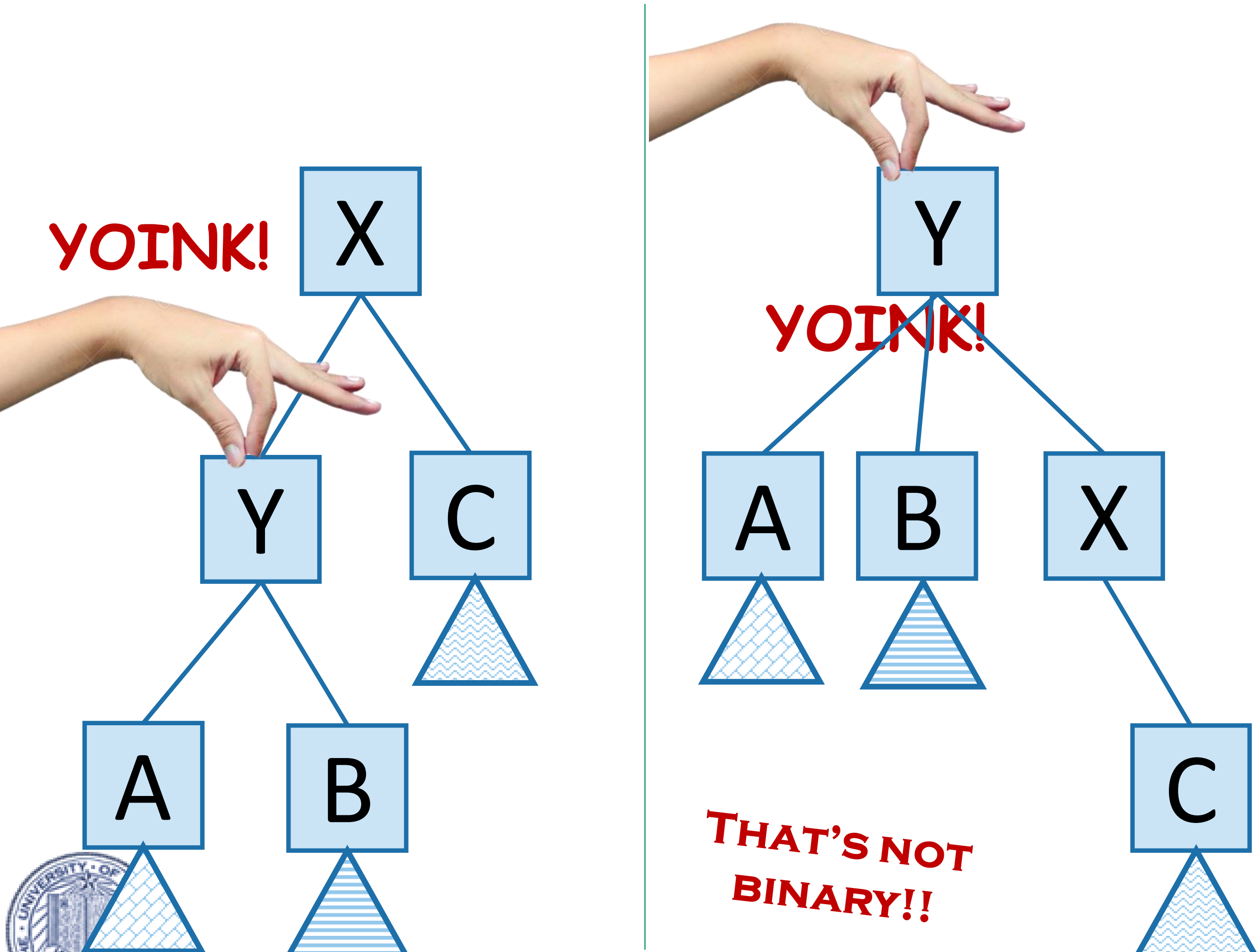




Note: A, B, C, X, Y are variable names, not the contents of the nodes.

Idea 1: Rotations

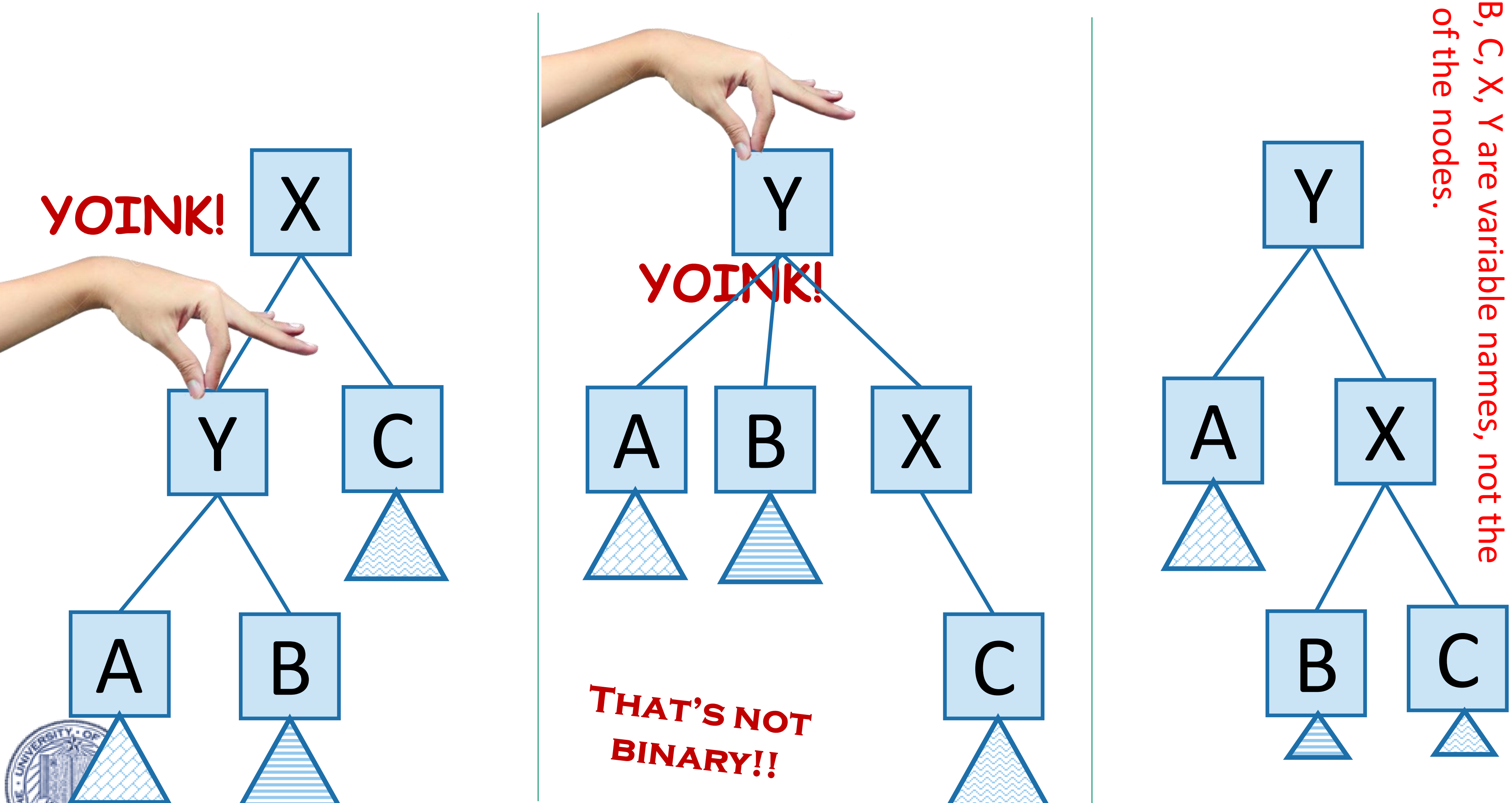
- Maintain Binary Search Tree (BST) property, while moving stuff around.



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

Idea 1: Rotations

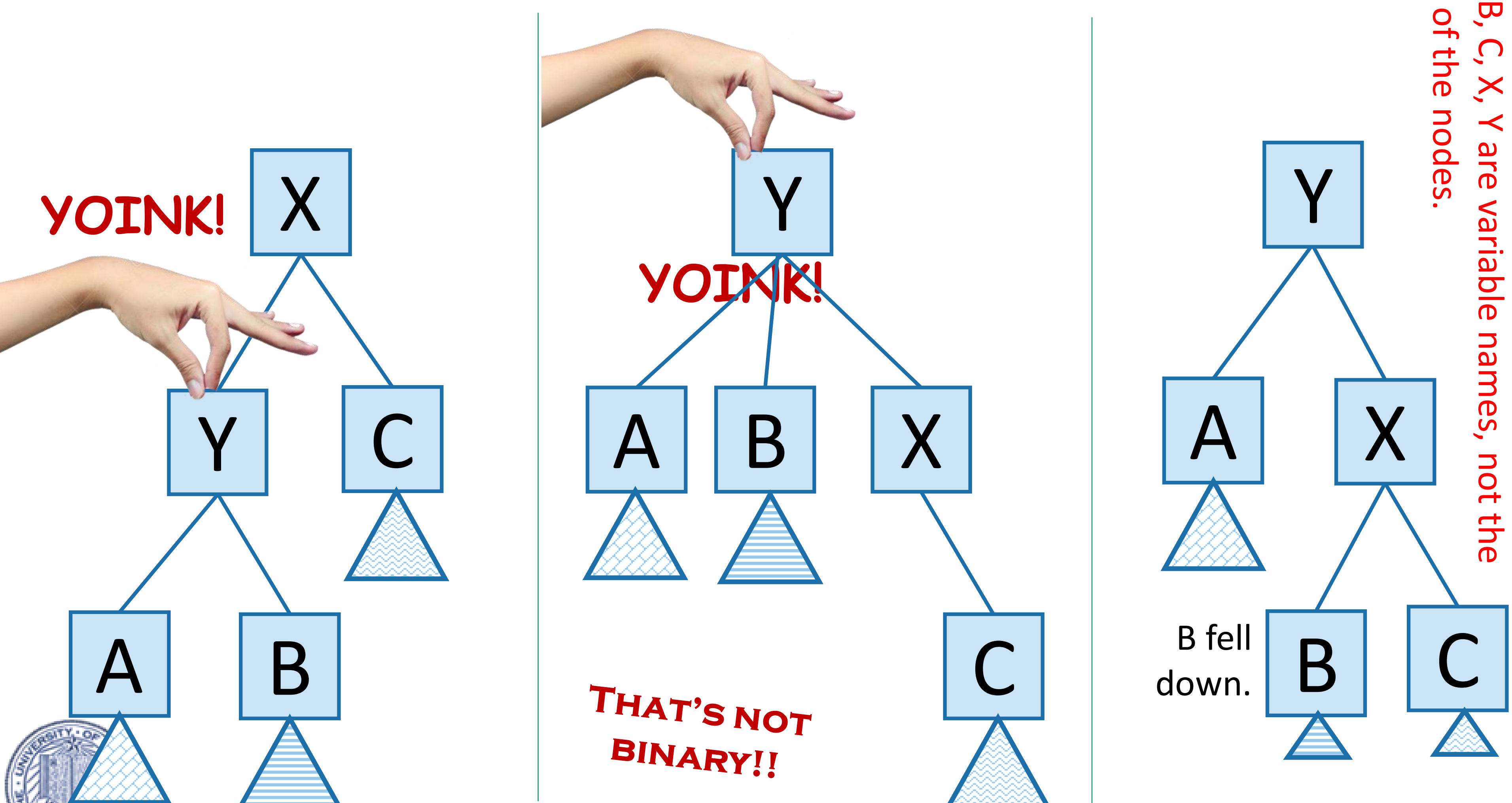
- Maintain Binary Search Tree (BST) property, while moving stuff around.



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

Idea 1: Rotations

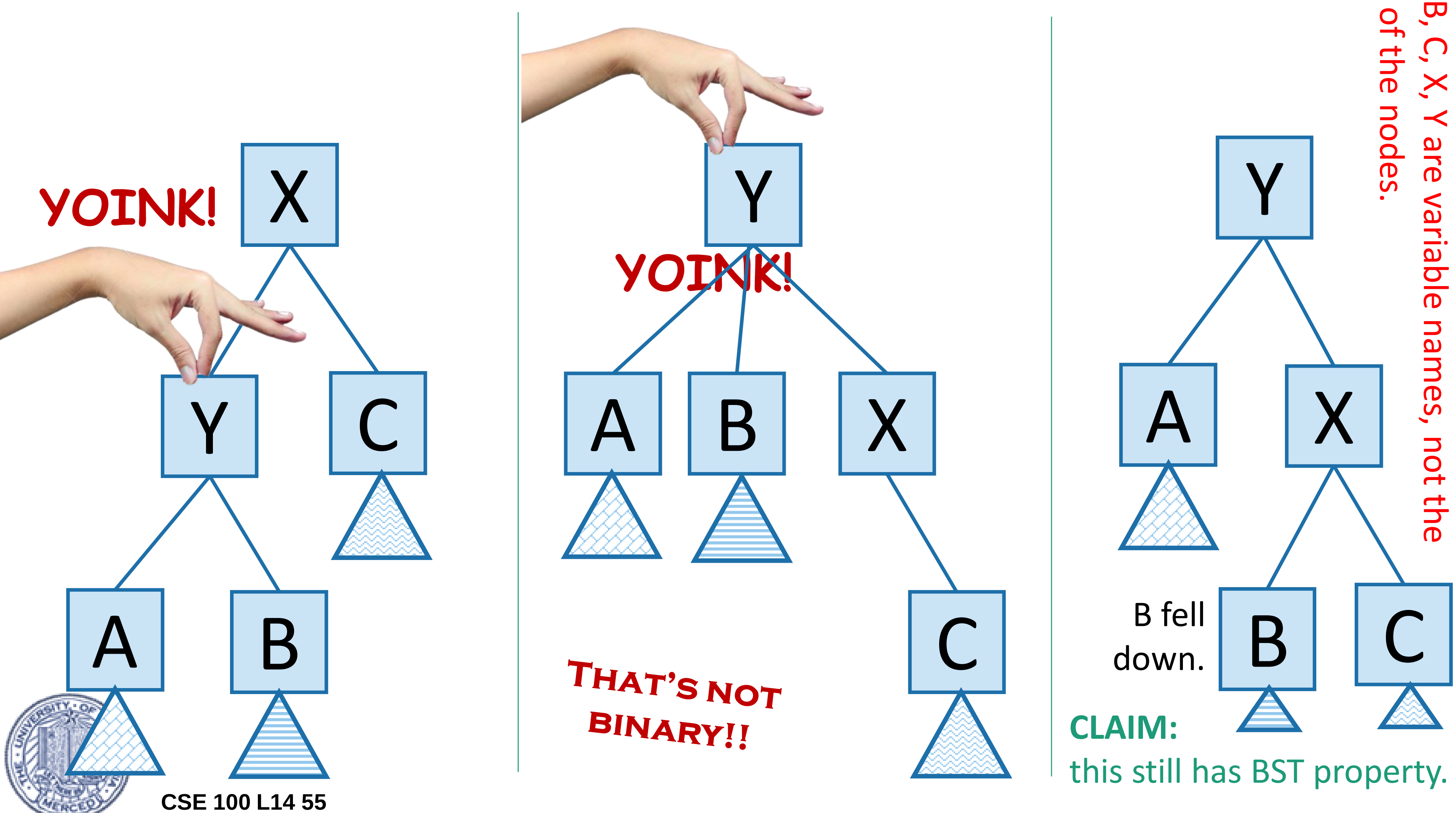
- Maintain Binary Search Tree (BST) property, while moving stuff around.



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

Idea 1: Rotations

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



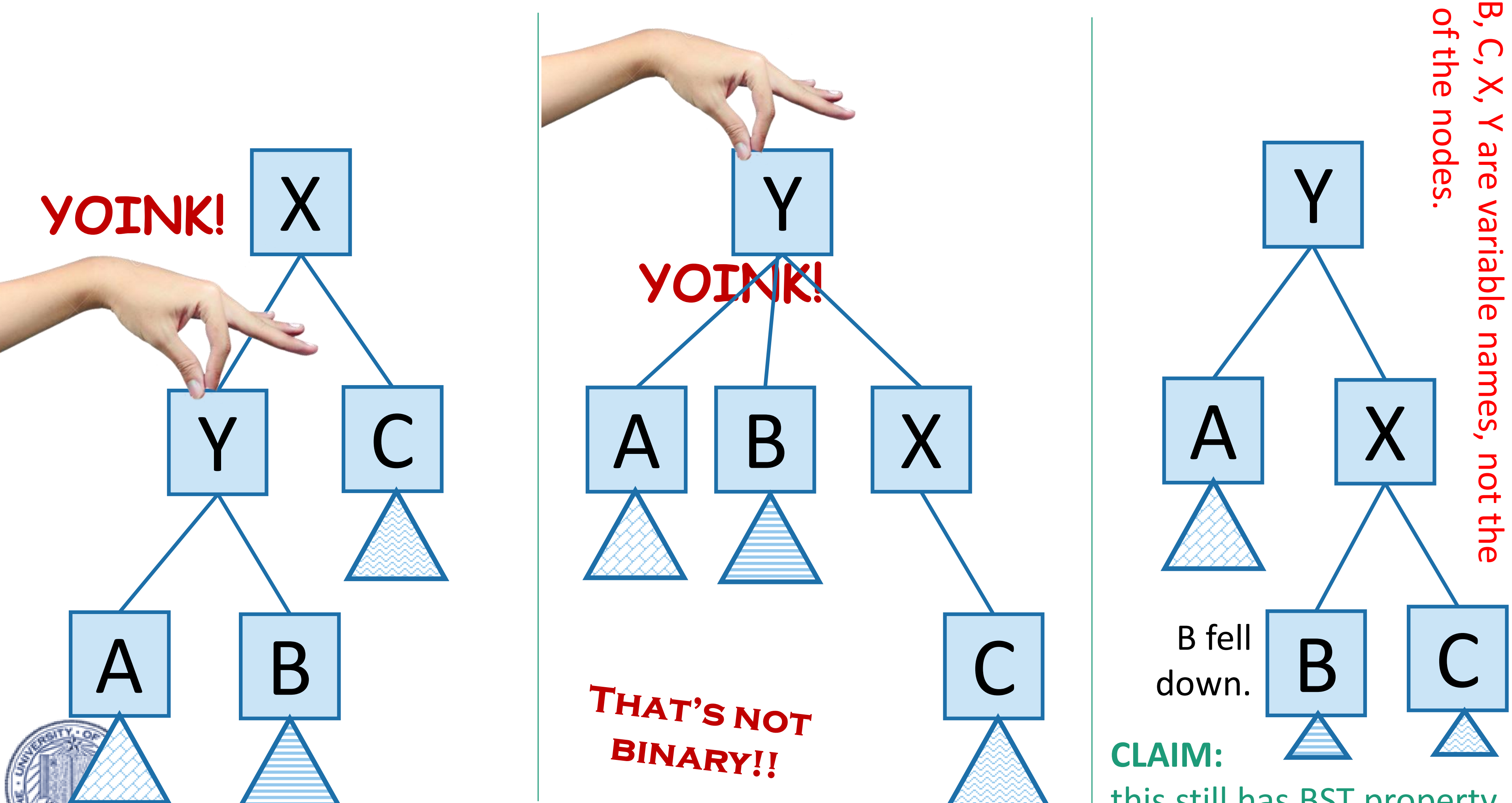
Note: A, B, C, X, Y are variable names, not the contents of the nodes.

Idea 1: Rotations

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

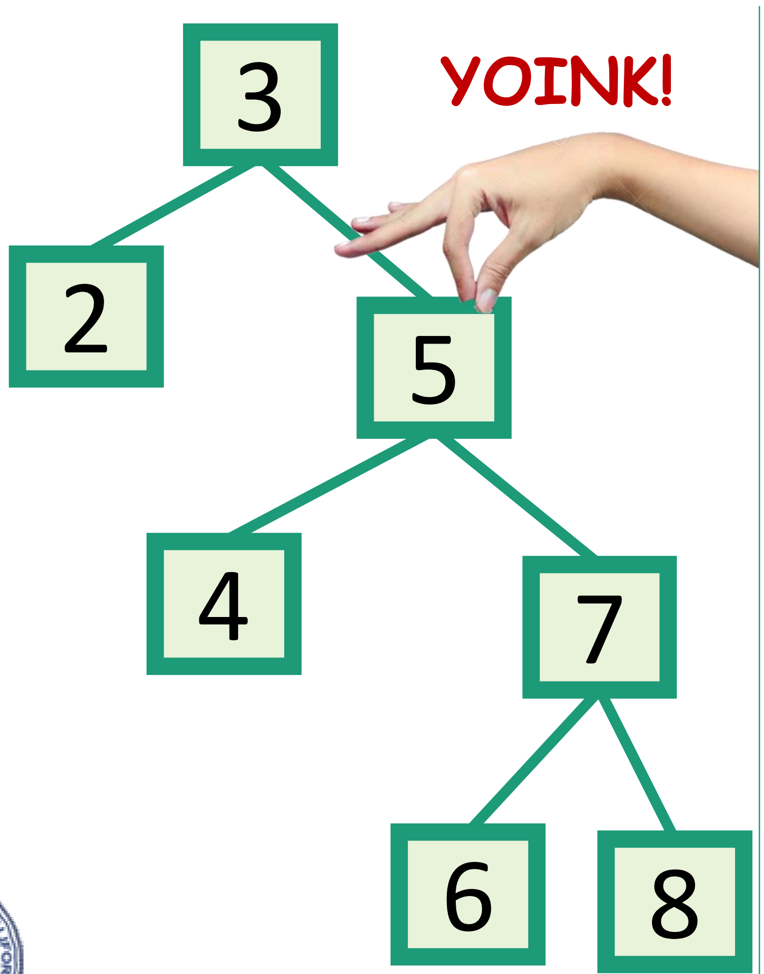
- Maintain Binary Search Tree (BST) property, while moving stuff around.

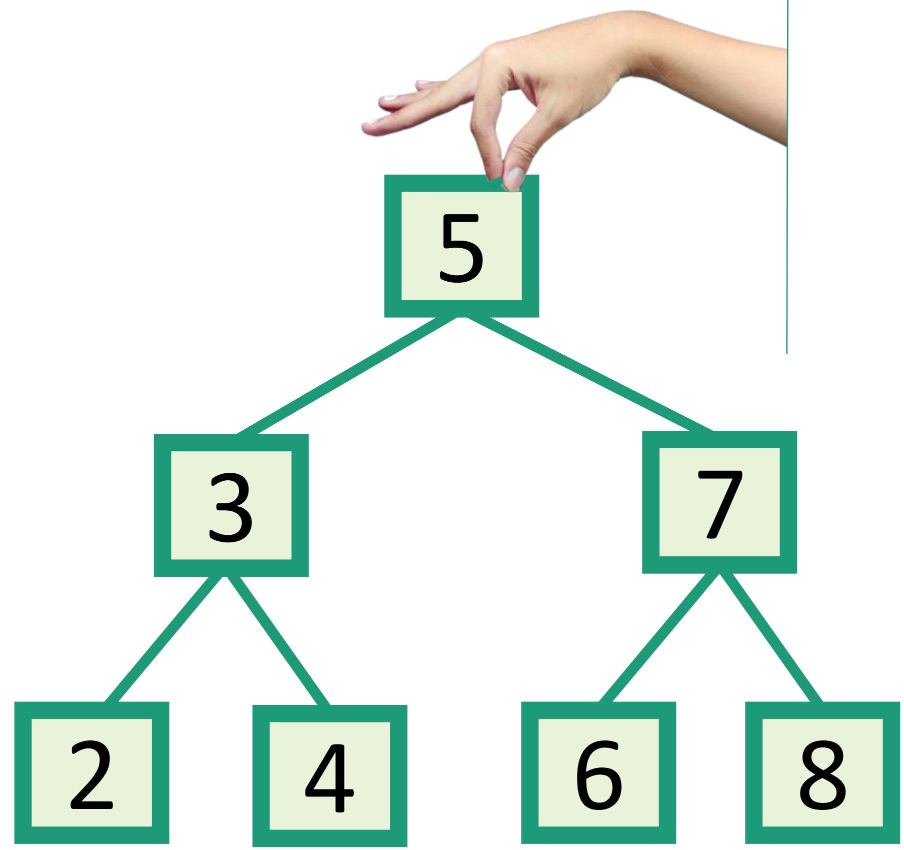
Note: A, B, C, X, Y are variable names, not the contents of the nodes.



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…

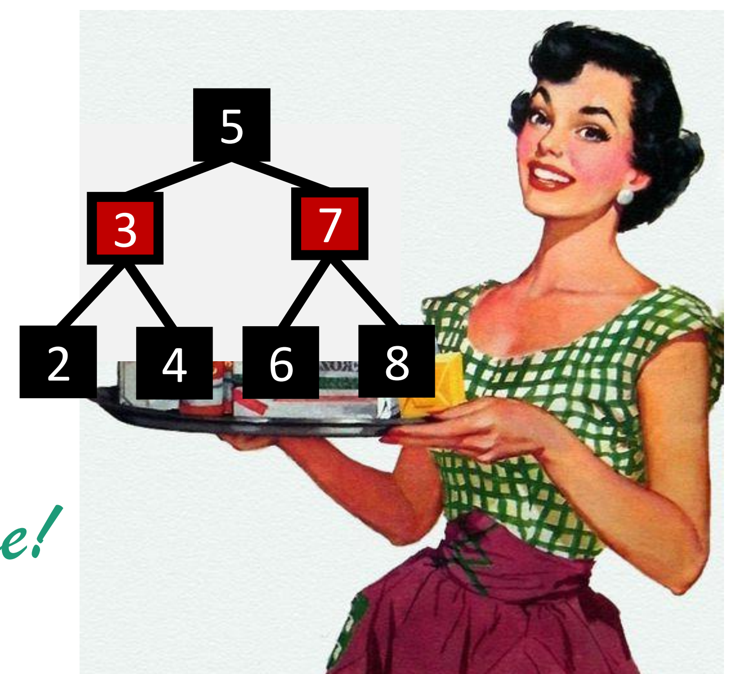
Red-Black Trees

- 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.**

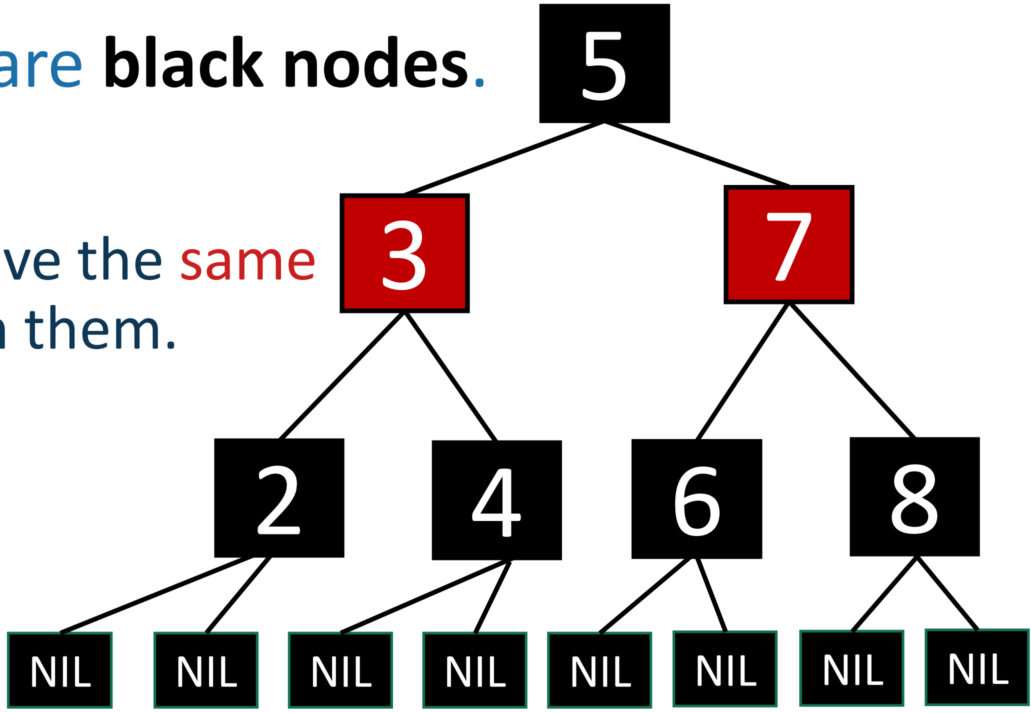
It's just good sense!



Red-Black Trees

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

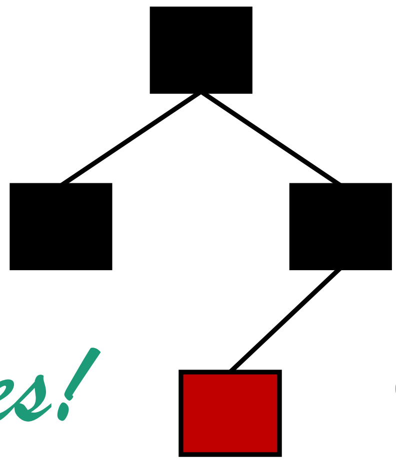
1. Every node is colored **red** or **black.**
2. The root node is a **black node**.
3. NIL children count as **black nodes**.
4. Children of a **red node** are **black nodes**.
5. For all nodes x:
   - all paths from x to NIL's have the same number of **black nodes** on them.



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

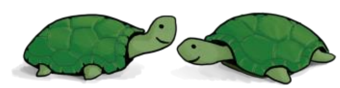
Examples(?)

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

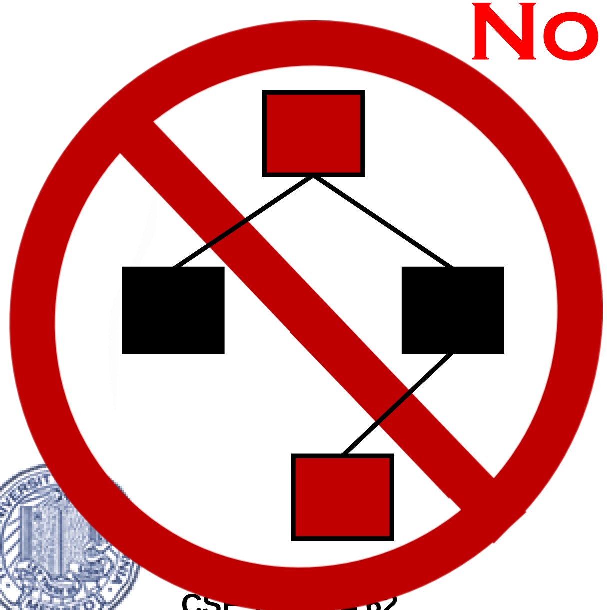


Yes!

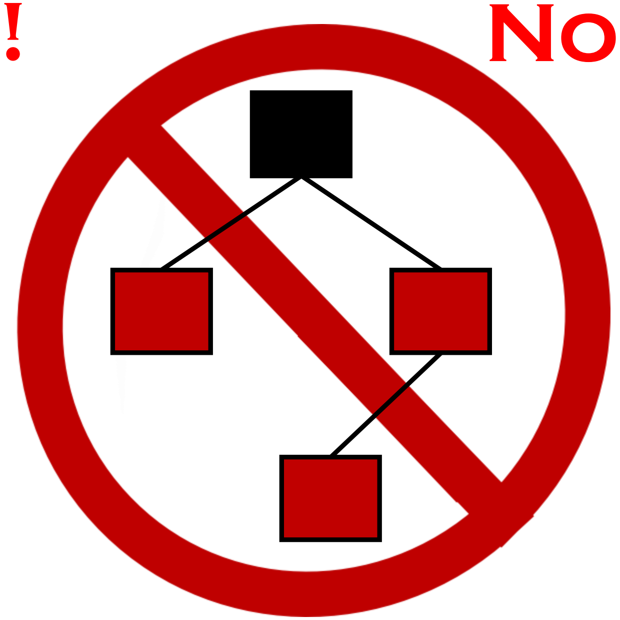
Which of these are red-black trees? (NIL nodes not drawn)



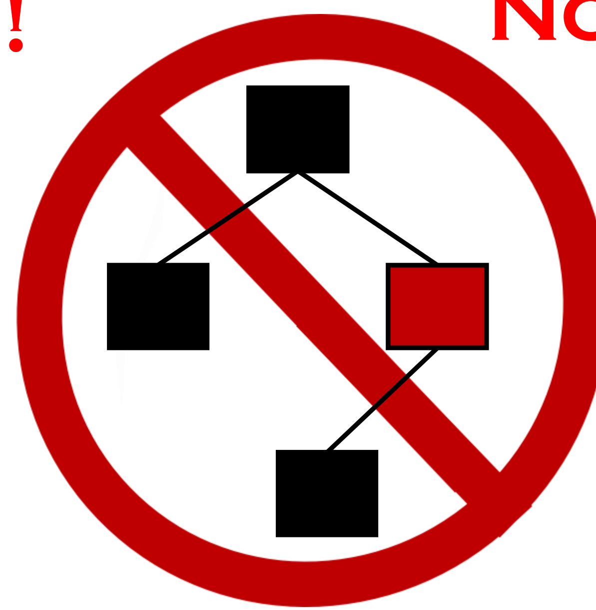
No!



No!

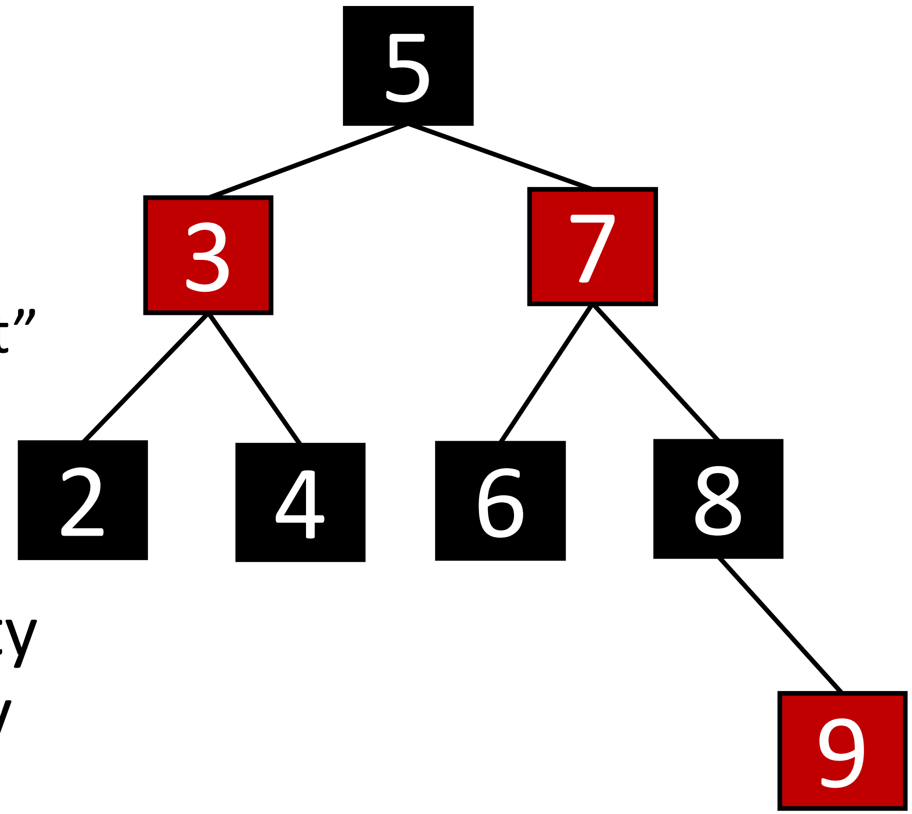


No!



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.

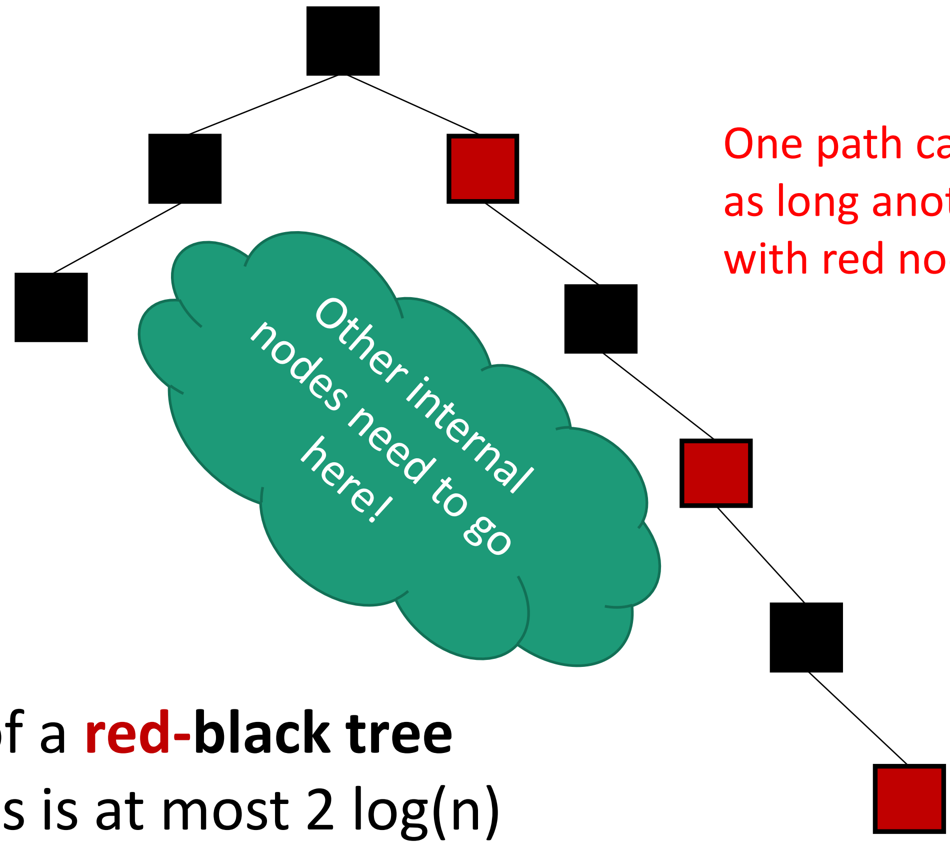
It's just a smidge weaker than perfect balance, but we can actually maintain it!

Let’s build some intuition!

This is “pretty balanced”

Lucky the lackadaisical lemur

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

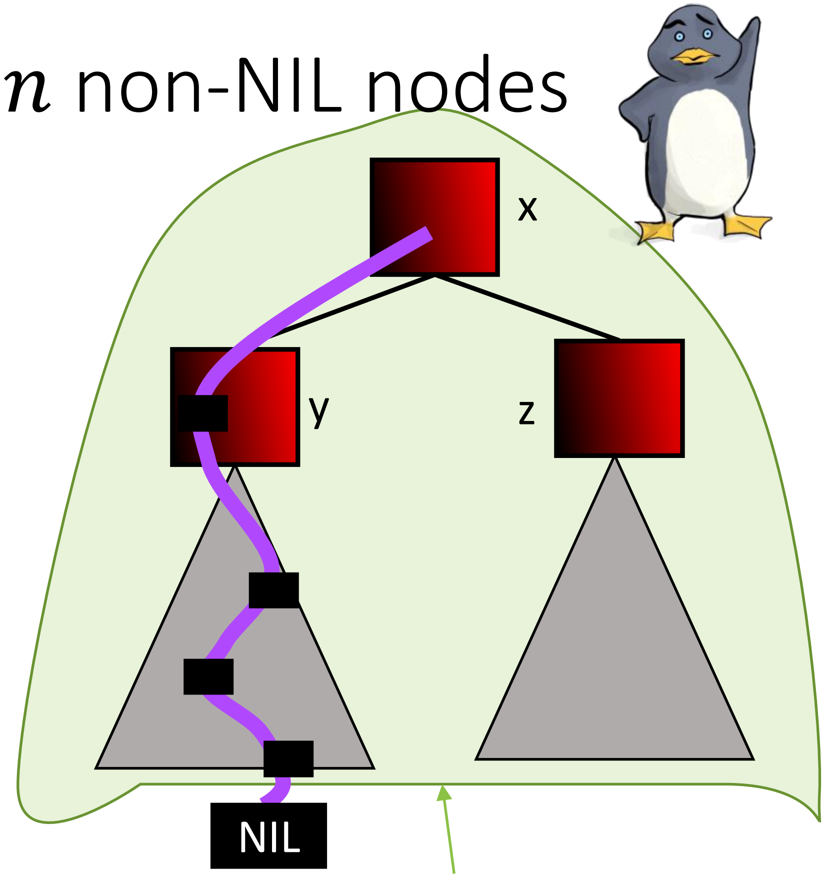


One path can be at most twice as long another if we pad it with red nodes.

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 $2\log(n+1)$

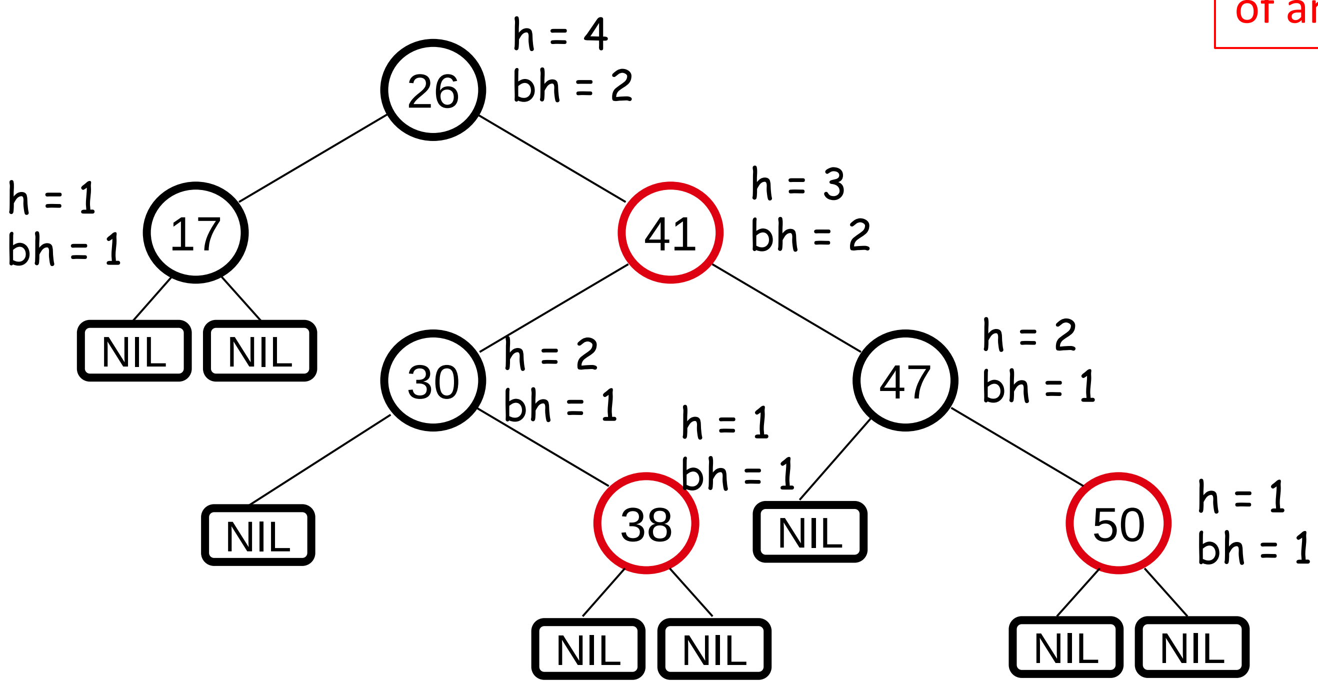
- Define bh(x) to be the number of black nodes in any path from x to NIL.
  - (excluding x, including NIL).
- Claim:
  - There are at least $2^{bh(x)} - 1$ non-NIL nodes in the subtree underneath x. (including x).
- [For a formal Proof – see next slides]



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

Definitions

Aside: Formal Proof on the maximum height of an RB Tree



- **Height of a node** $h(x)$ **:** the number of edges in the **longest** path to a leaf (including the edge from leaf to NIL).
- **Black-height** $bh(x)$ of a node x: the number of black nodes (including NIL) on the path from x to a leaf, <u>not counting x.</u>

Height of Red-Black-Trees

Aside: Formal Proof on the maximum height of an RB Tree

A red-black tree with n internal nodes

has height <u>at most</u> $2\log(n+1)$

- Need to prove two claims first …

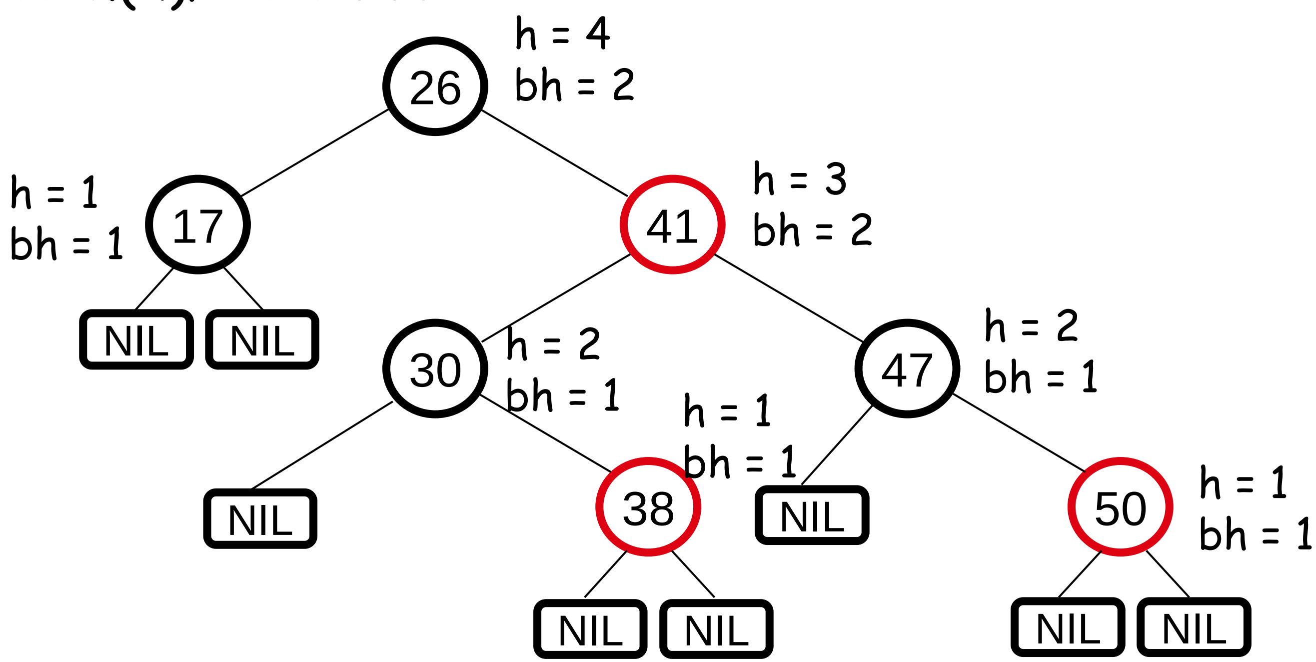
Claim 1

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

- Any node x with height h(x) has

$$\boxed{bh(x) \geq h(x)/2}$$

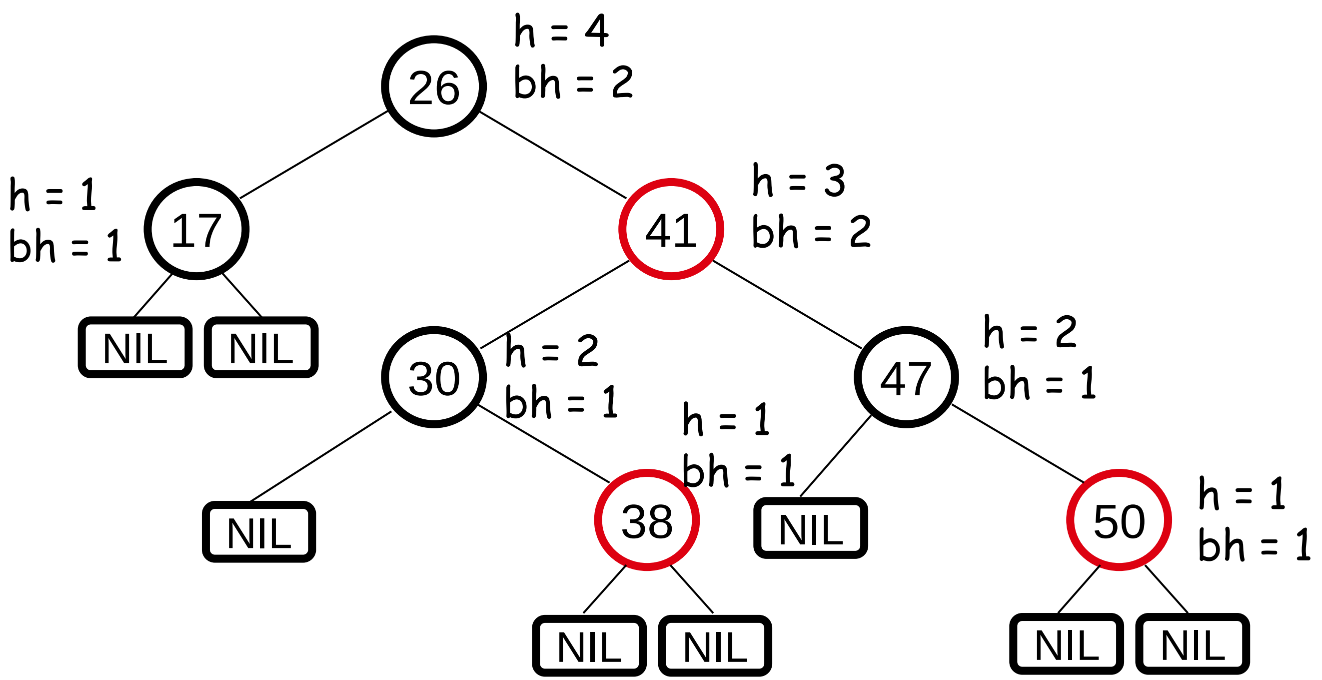
- **Proof**
  - By property 4, at most h(x)/2 **red** nodes on the path from the node x to a leaf
  - Hence at least h(x)/2 are **black**



Claim 2

Aside: Formal Proof on the maximum height of an RB Tree

- The subtree rooted at any node x contains **at least** $2^{bh(x)} - 1$ internal nodes



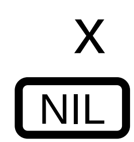
Claim 2 (cont'd)

Aside: Formal Proof on the maximum height of an RB Tree

Proof: By induction on **h(x)**

Basis: $h(x) = 0 \Rightarrow$

x is a leaf (NIL) $\Rightarrow$



$bh(x) = 0 \Rightarrow$

# of internal nodes: $2^0 - 1 = 0$

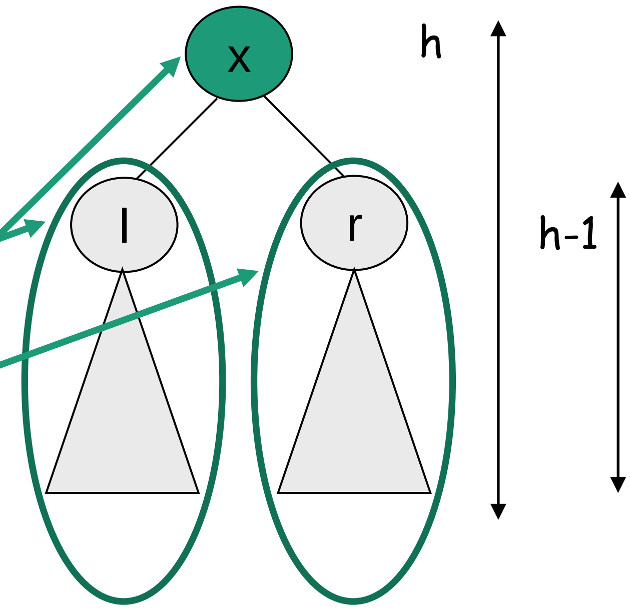
Inductive Hypothesis: assume it is true for $h(x) = h-1$

Claim 2 (cont'd)

Aside: Formal Proof on the maximum height of an RB Tree

Inductive step:

- Prove it for $h(x)=h$



internal nodes at x=

internal nodes at l +

internal nodes at r + 1

Using inductive hypothesis:

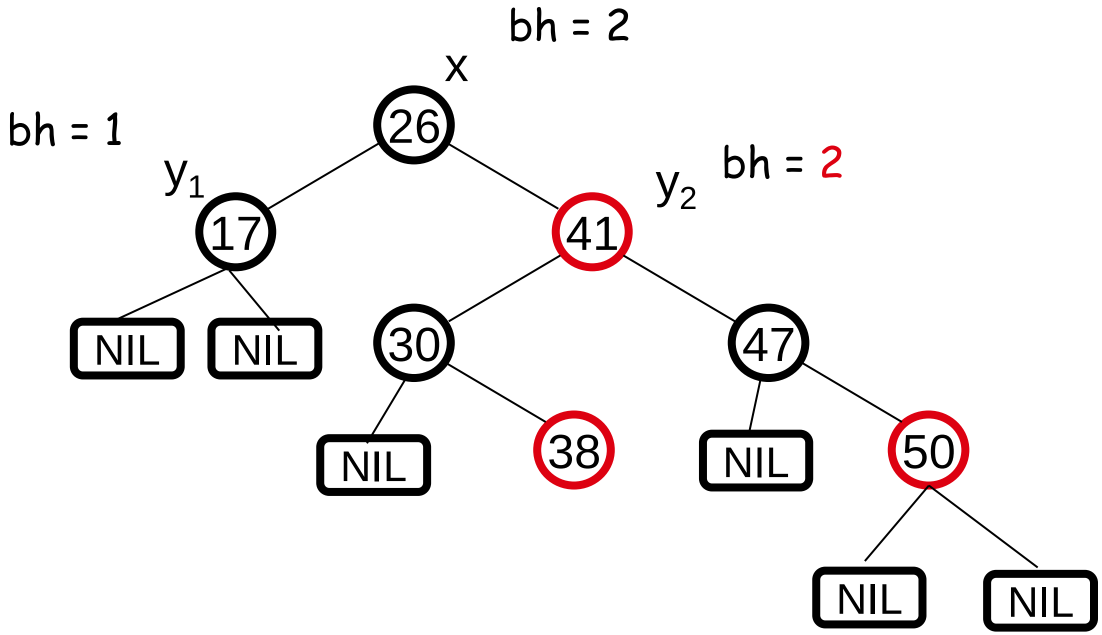
internal nodes at x $\geq (2^{bh(l)} - 1) + (2^{bh(r)} - 1) + 1$

Claim 2 (cont'd)

Aside: Formal Proof on the maximum height of an RB Tree

- Let bh(x) = b, then any child y of x has:
  - bh (y) = b (if the child is **red**), or
  - bh (y) = b - 1 (if the child is **black**)

$bh(y) \geq bh(x)-1$



Claim 2 (cont'd)

Aside: Formal Proof on the maximum height of an RB Tree

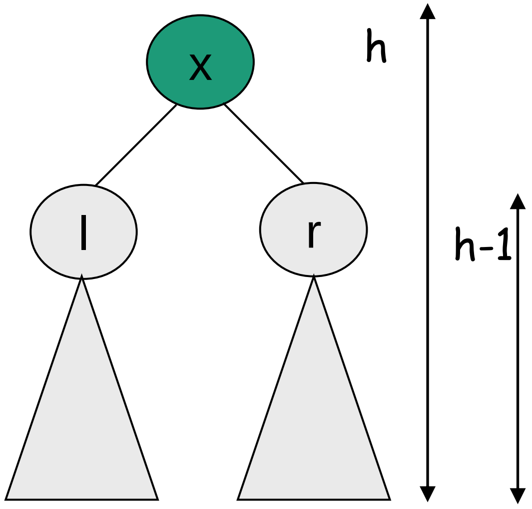
- So, back to our proof:

internal nodes at x $\geq (2^{bh(l)} - 1) + (2^{bh(r)} - 1) + 1$

$$\geq (2^{bh(x)-1} - 1) + (2^{bh(x)-1} - 1) + 1 =$$

$$2 \cdot (2^{bh(x)-1} - 1) + 1 =$$

$2^{bh(x)} - 1$ internal nodes



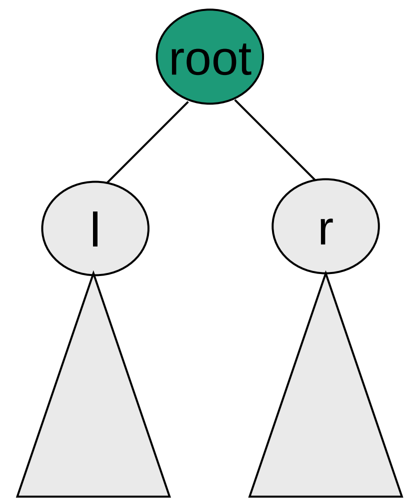
Height of Red-Black-Trees

Aside: Formal Proof on the maximum height of an RB Tree

A red-black tree with **n** internal nodes has height at most $2\log(n+1)$.

Proof:

n	**bh(root)** = b	**height(root)** = h
	$\geq 2^b - 1$	$\geq 2^{h/2} - 1$
number of internal nodes	Claim 2	Claim 1



- Solve for **h**:

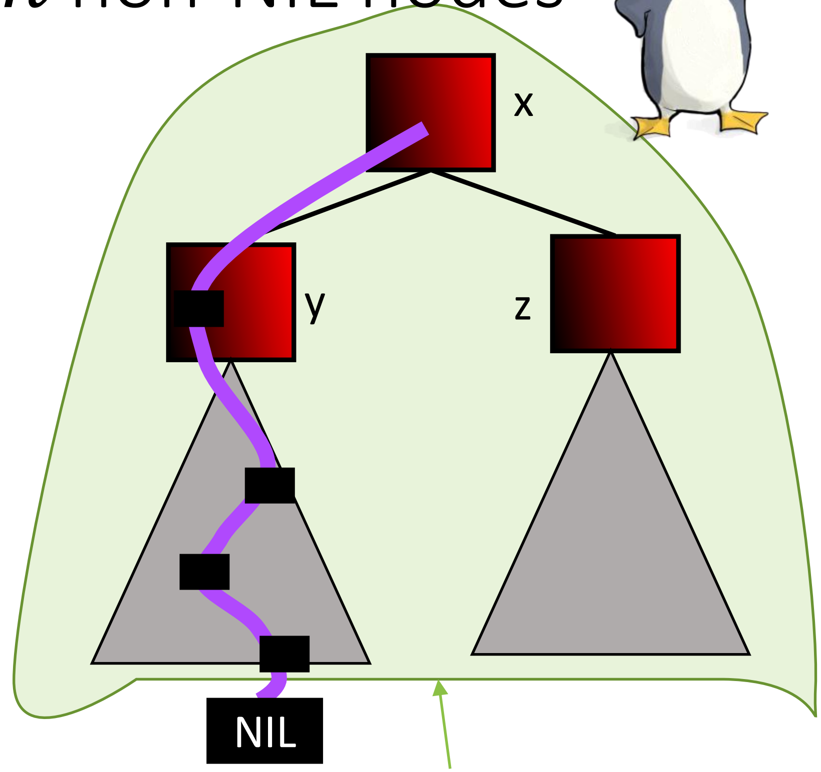
$$n + 1 \geq 2^{h/2}$$

$$\log(n + 1) \geq h/2 \Rightarrow$$

$$\boxed{h \leq 2\log(n + 1)}$$

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

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



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

Then:

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

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

Rearranging:

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