



**Indian Institute of Information Technology Allahabad**

# Data Structures and Algorithms

## Binary Search Tree



**Dr. Shiv Ram Dubey**

Associate Professor

Department of Information Technology

Indian Institute of Information Technology, Allahabad

Email: [srdubey@iiita.ac.in](mailto:srdubey@iiita.ac.in)

Web: <https://profile.iiita.ac.in/srdubey/>

# DISCLAIMER

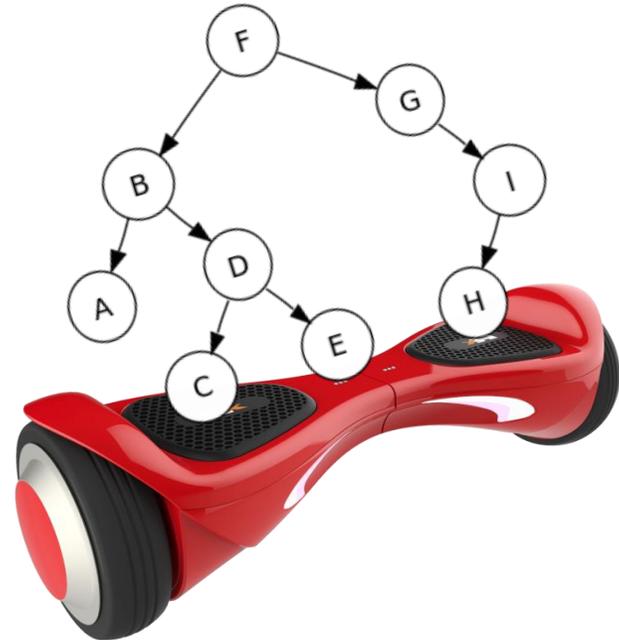
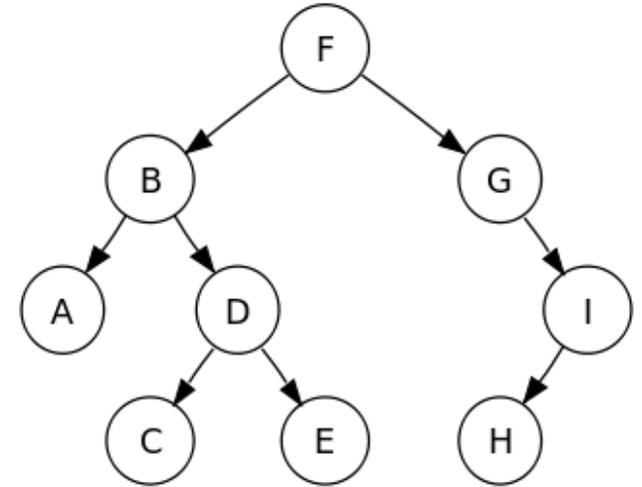
The content (text, image, and graphics) used in this slide are adopted from many sources for academic purposes. Broadly, the sources have been given due credit appropriately. However, there is a chance of missing out some original primary sources. The authors of this material do not claim any copyright of such material.

# Tree

- Binary search trees
  - They are better when they're balanced.

this will lead us to...

- Self-Balancing Binary Search Trees
  - **AVL Tree**
  - **2-3 Tree**
  - **Red-Black trees.**



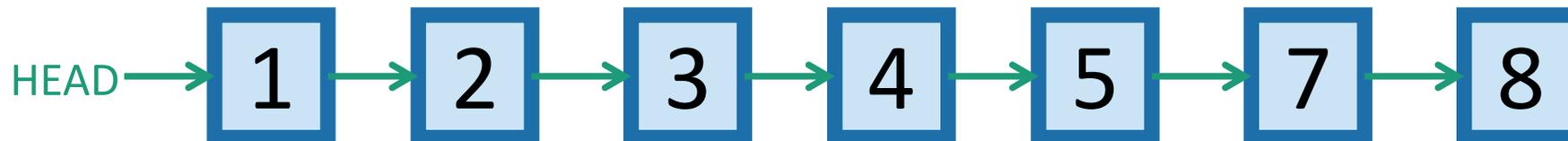
# Some data structures

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

- (Sorted) arrays:



- (Sorted) linked lists:



- Some basic operations:

- **INSERT, DELETE, SEARCH**

# Sorted Arrays



- $O(n)$  INSERT/DELETE:

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



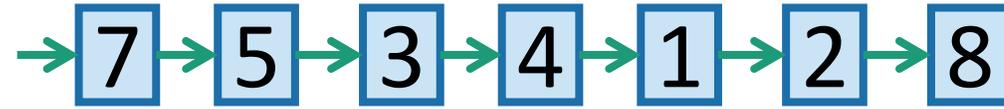
eg, insert 4.5

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



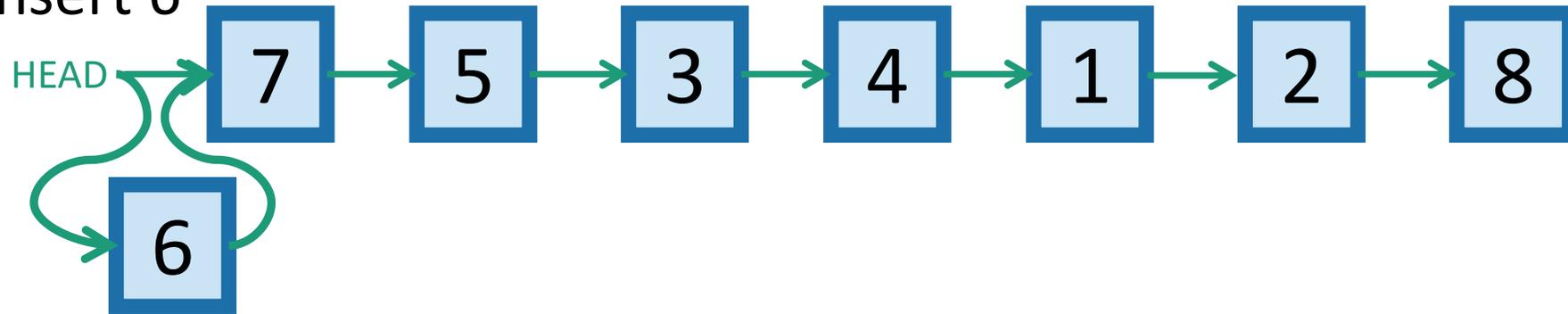
eg, Binary search to see if 3 is in A.

# UNSorted linked lists

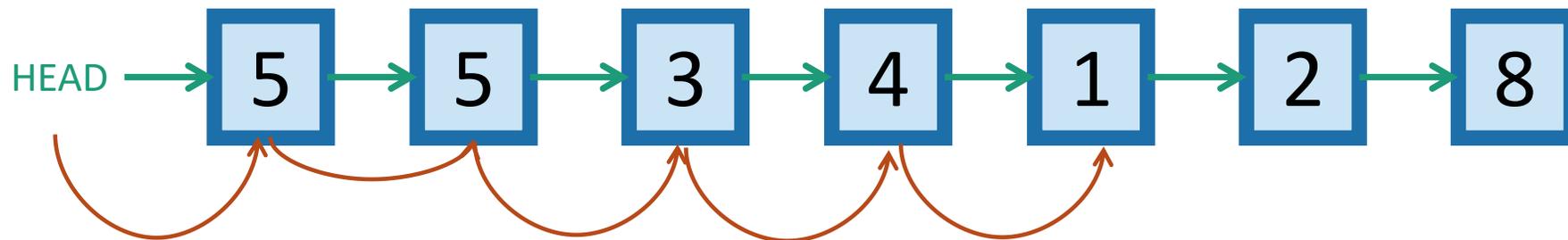


- $O(1)$  INSERT:

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

|        | 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 | $O(n)$        | $O(1)$       | $O(\log(n))$         |

# Motivation for Binary Search Trees

|        | 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 | $O(n)$         | $O(1)$ 😊     | $O(\log(n))$         |

# Motivation for Binary Search Trees

|        | 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 | $O(n)$ 😞       | $O(1)$ 😊     | $O(\log(n))$         |

# 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 | $O(n)$ 😞       | $O(1)$ 😊     | $O(\log(n))$ 😊       |

# Binary tree terminology

For today all keys are distinct.

Each node has at most two children.

The left child of 3 is 2

The right child of 3 is 4

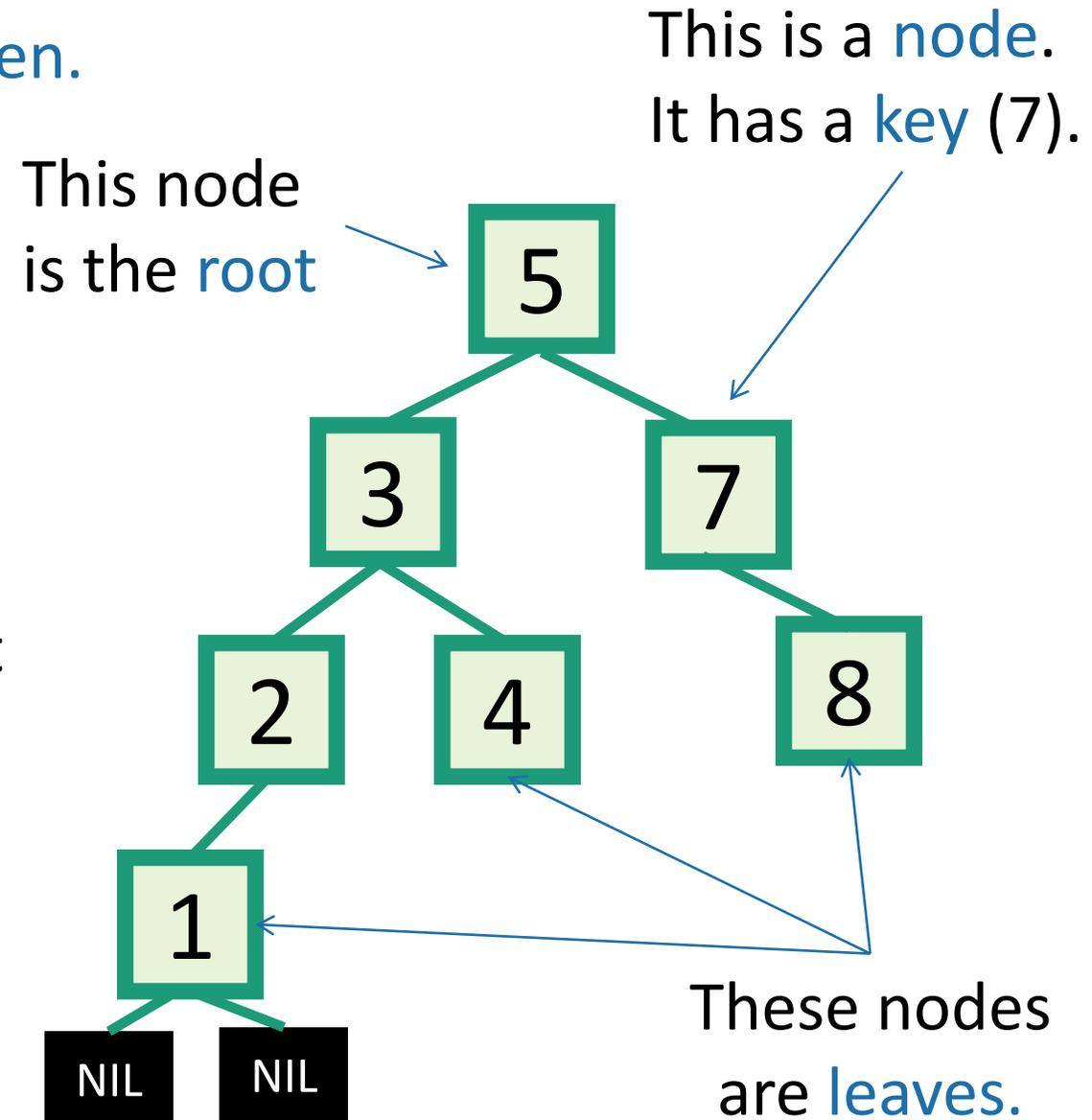
The parent of 3 is 5

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.  
(Not usually drawn).

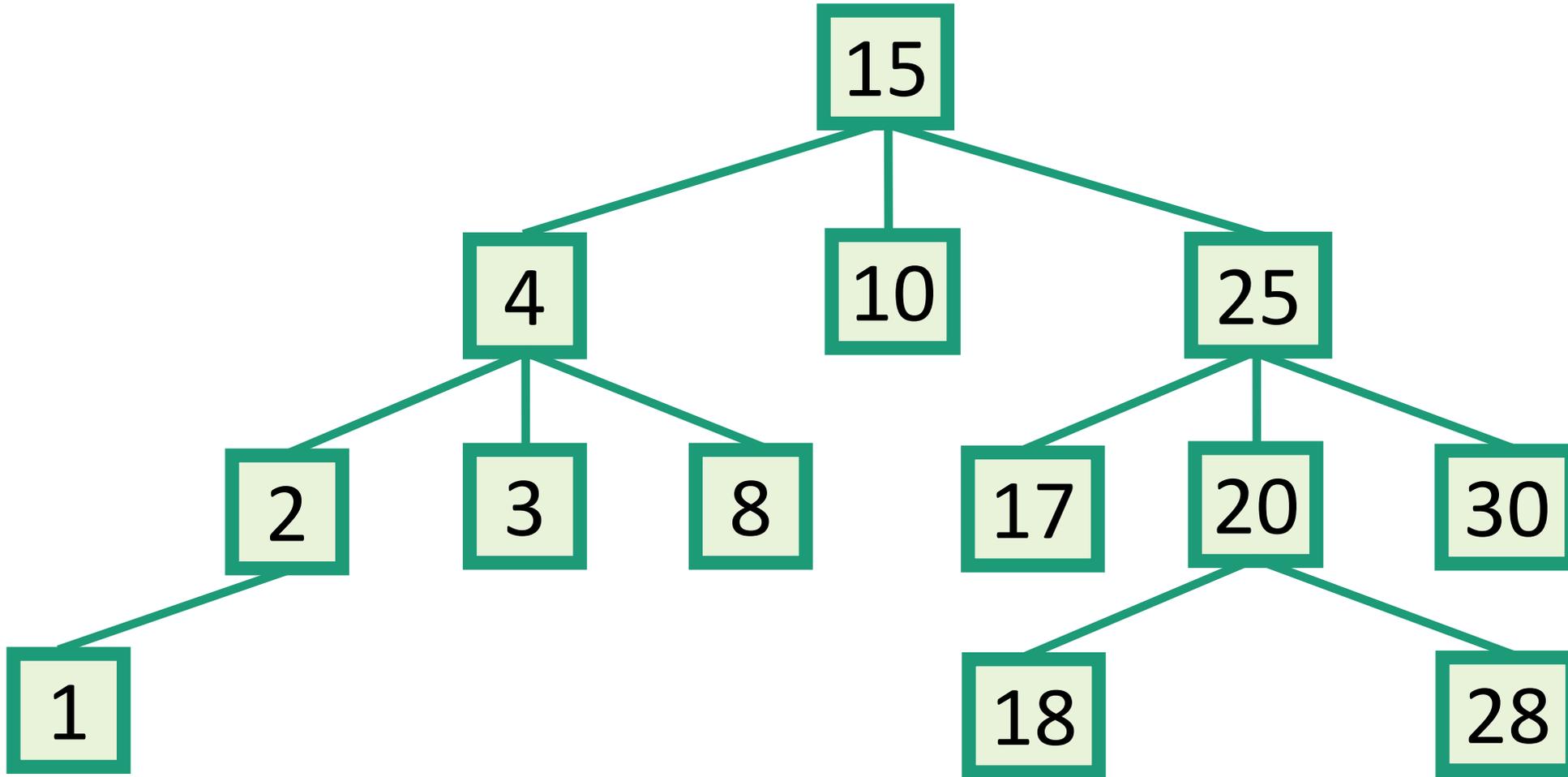
The height of this tree is 3. (Max number of edges from the root to a leaf).



# Definition: k-ary trees

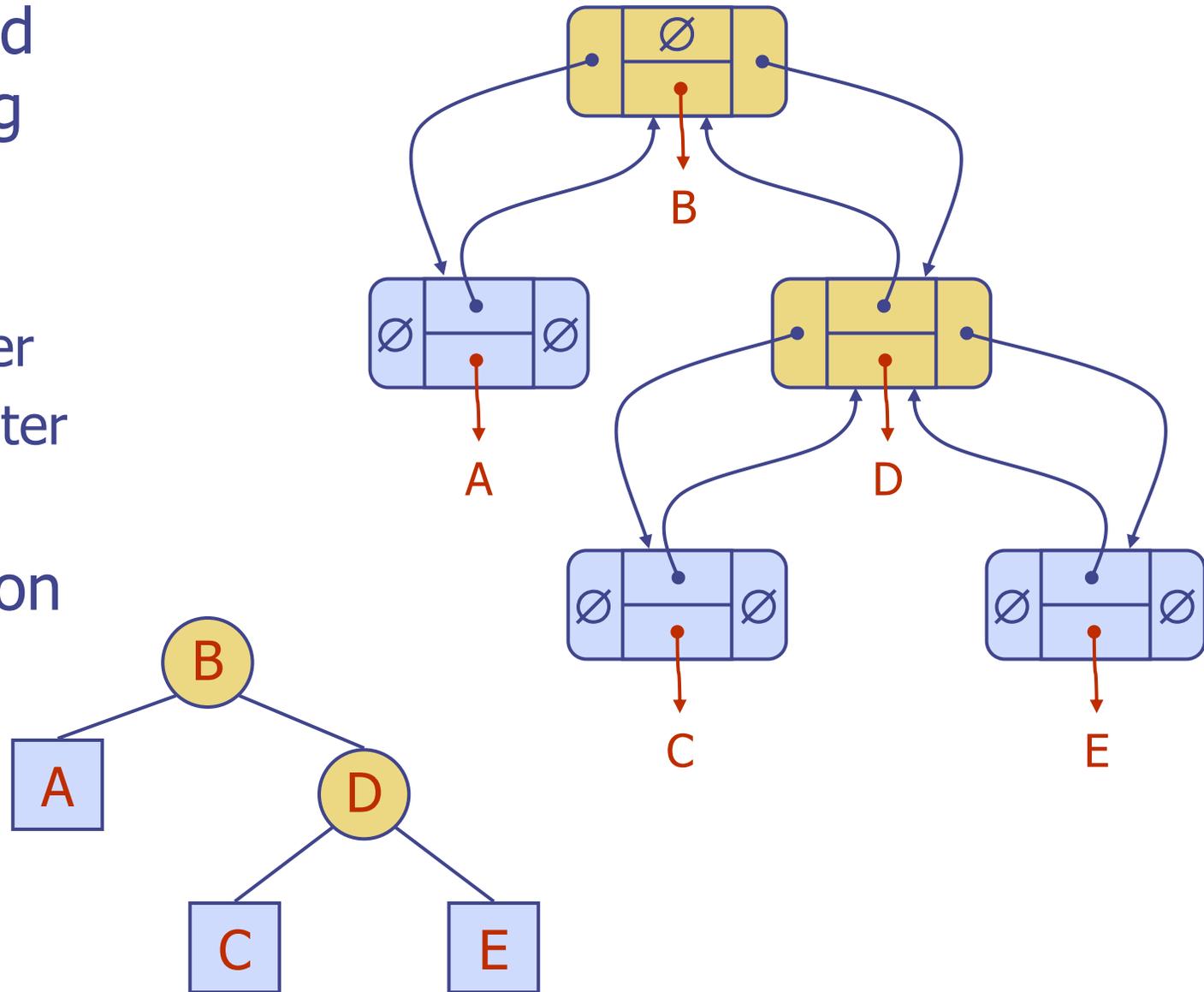
- Rooted tree where every vertex has no more than 'k' children
- Full k-ary if every internal vertex has exactly 'k' children (i.e., except leaf/external vertices).
- $k=2$  gives a binary tree
- $k=3$  gives a ternary tree

# Example: 3-ary tree



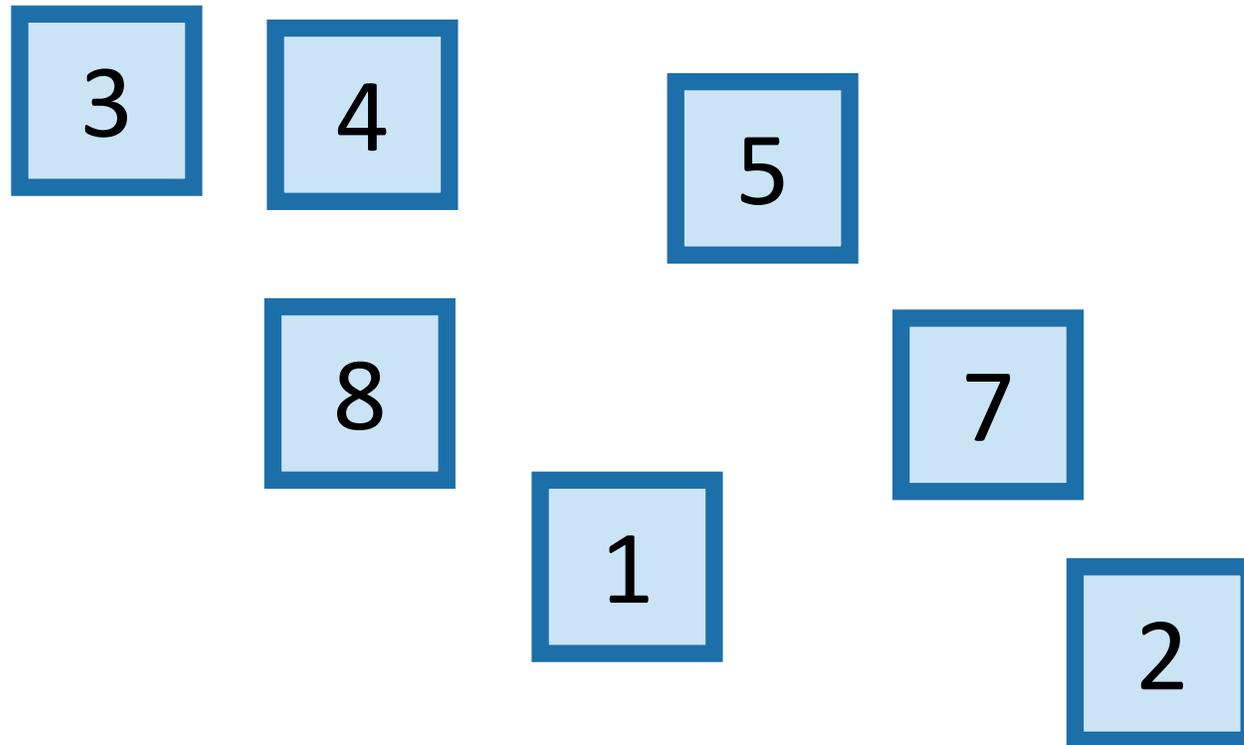
# Linked Structure for Binary Trees

- ◆ A node is represented by a structure storing
  - Element
  - Parent node pointer
  - Left child node pointer
  - Right child node pointer
- ◆ Node structure implement the Position ADT



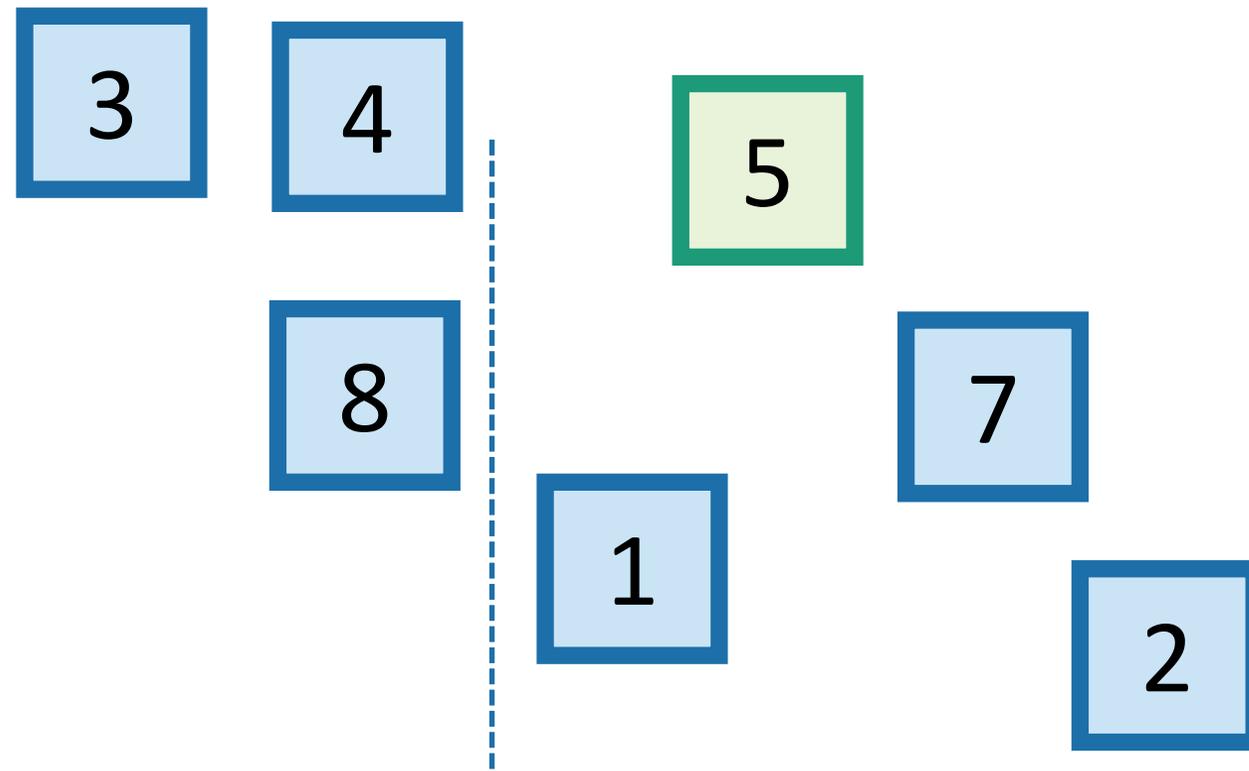
# Binary Search Trees

- 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:



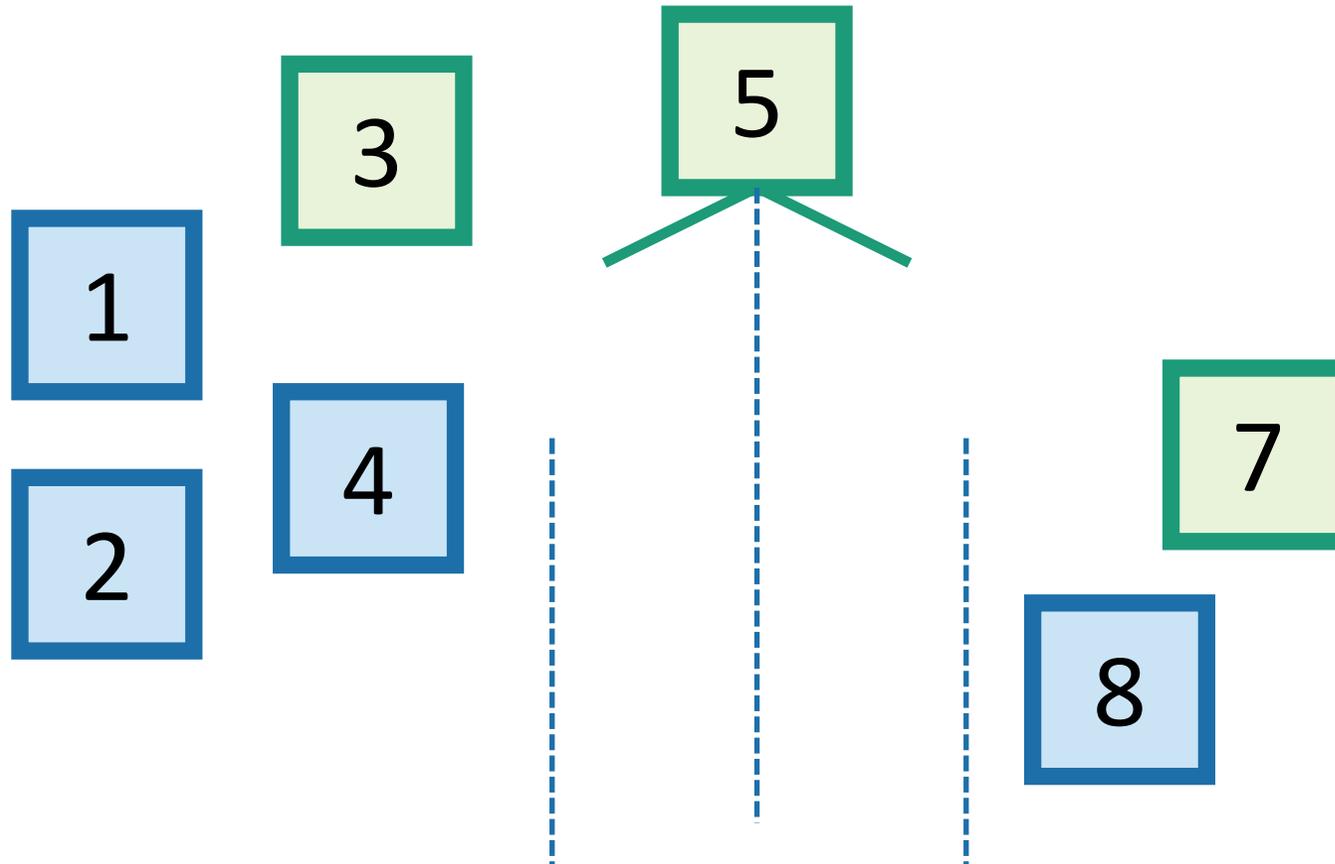
# Binary Search Trees

- 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:



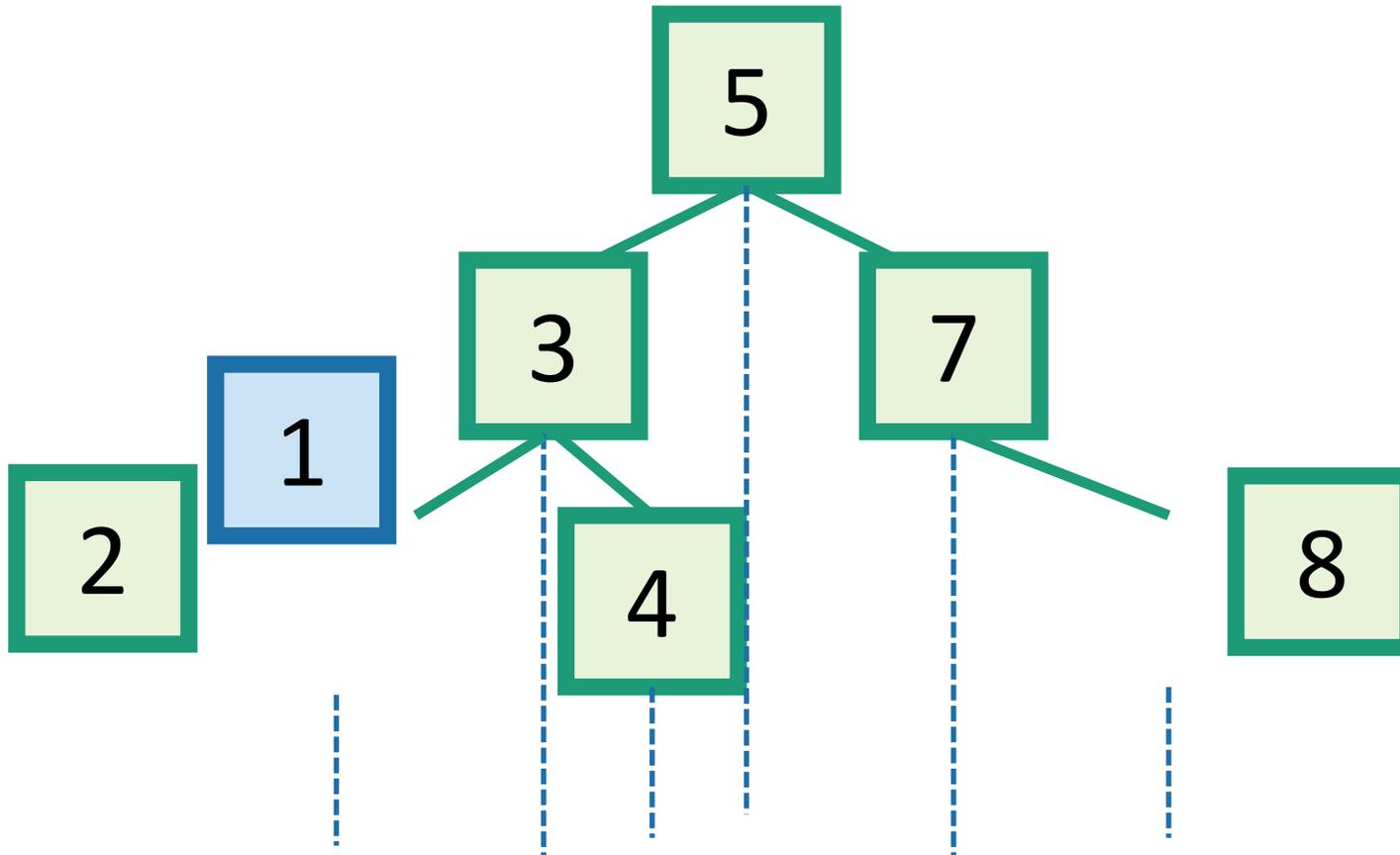
# Binary Search Trees

- 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:



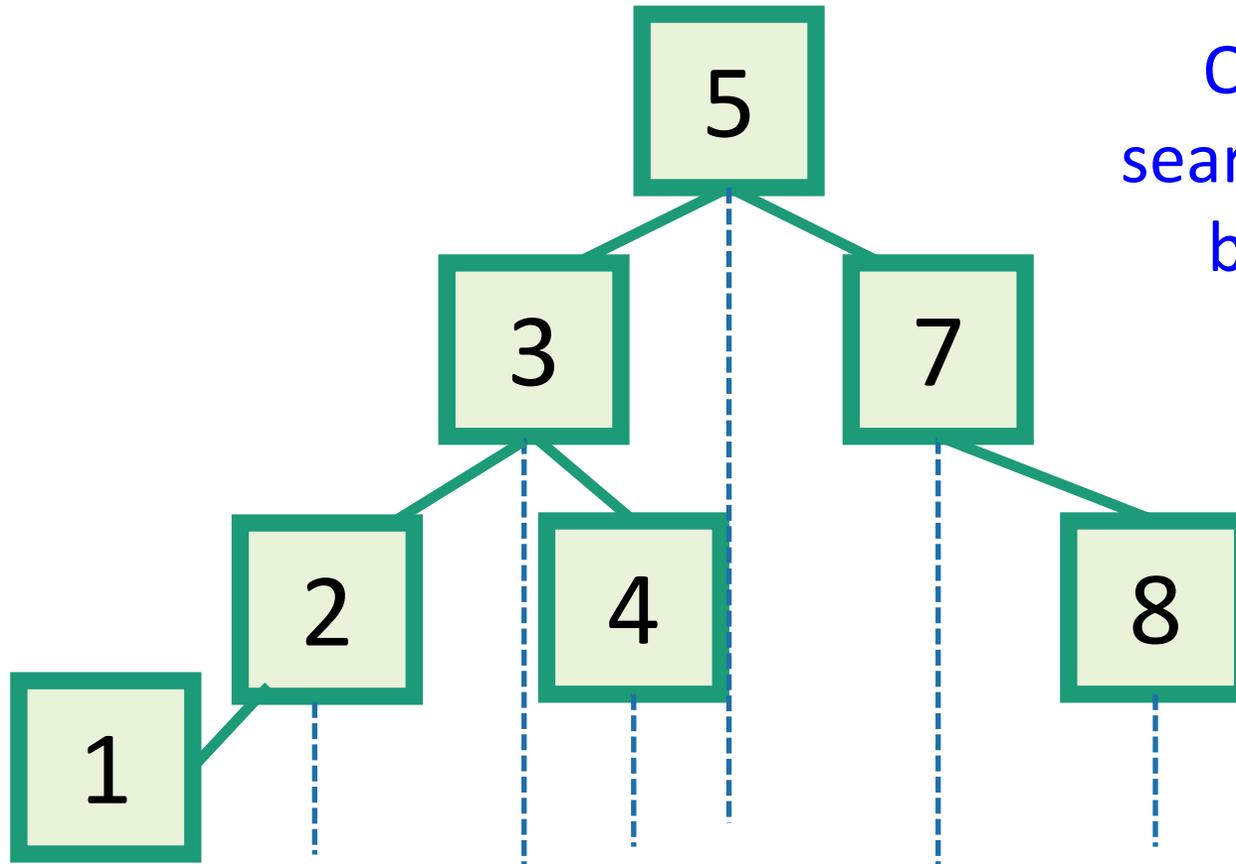
# Binary Search Trees

- 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:

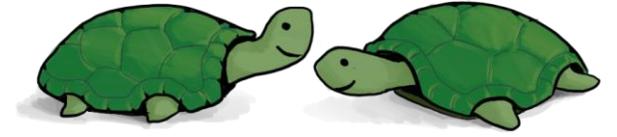


Q: Is this the only binary search tree I could possibly build with these values?

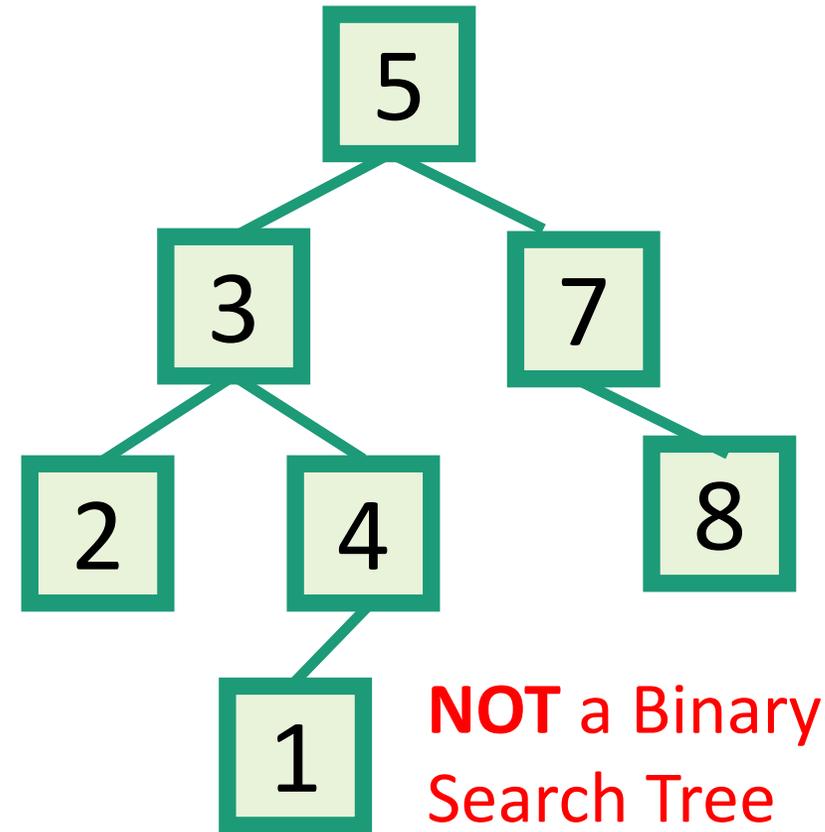
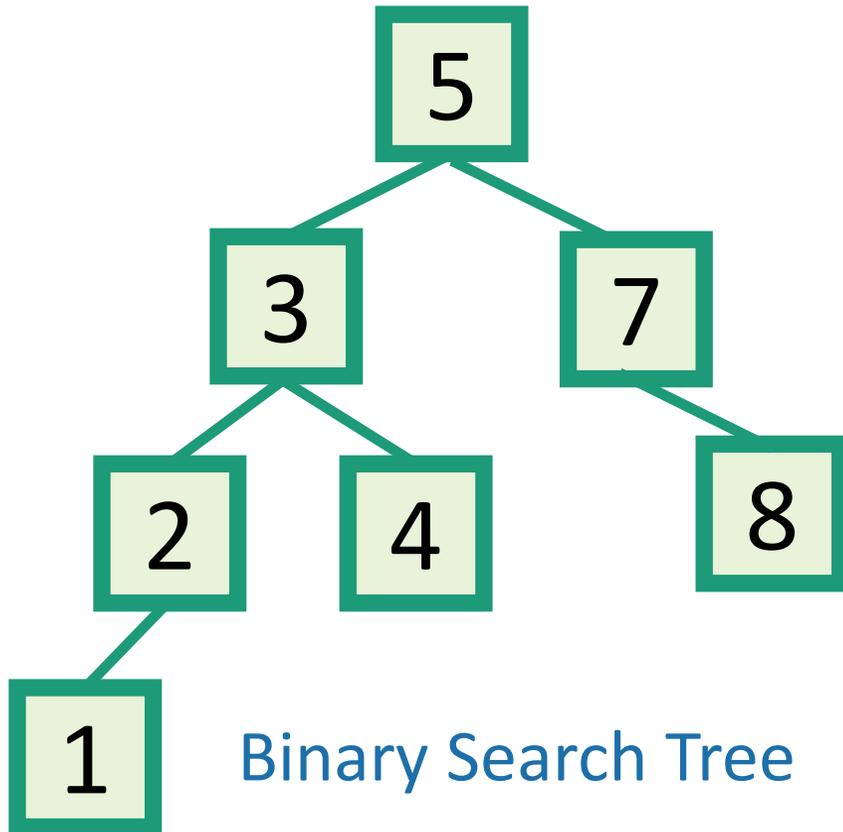
A: **No.** I made choices about which nodes to choose when. Any choices would have been fine.

# Binary Search Trees

Which of these is a BST?

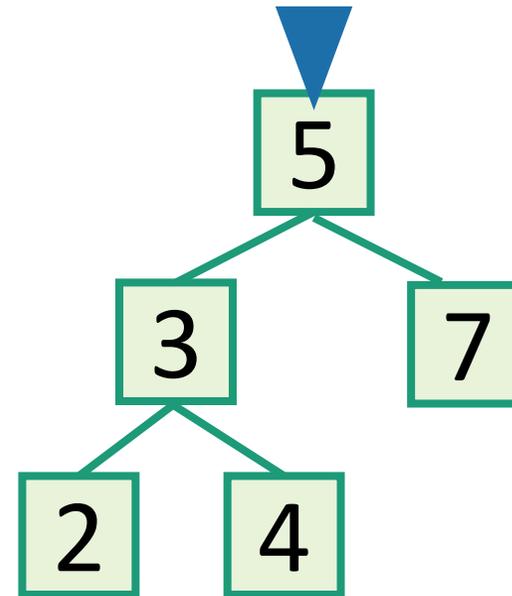


- A BST is a binary tree so that:
  - Every LEFT descendant of a node has key less than that node.
  - Every RIGHT descendant of a node has key larger than that node.



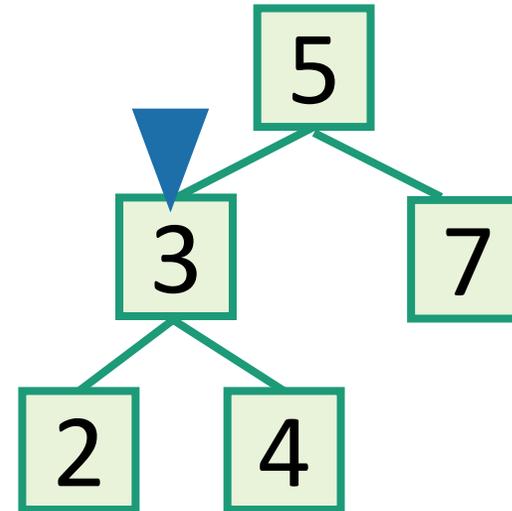
# Aside: In-Order Traversal of BSTs

- Output all the elements in sorted order!
- inOrderTraversal(x):
  - if  $x \neq \text{NIL}$ :
    - inOrderTraversal( x.left )
    - print( x.key )
    - inOrderTraversal( x.right )



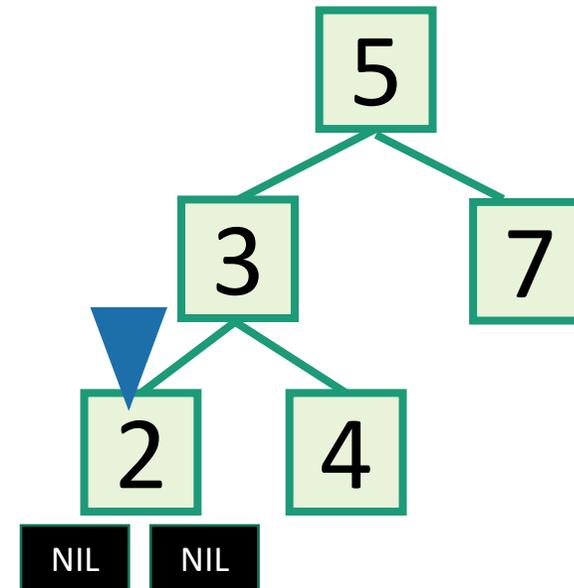
# Aside: In-Order Traversal of BSTs

- Output all the elements in sorted order!
- inOrderTraversal(x):
  - if  $x \neq \text{NIL}$ :
    - inOrderTraversal( x.left )
    - print( x.key )
    - inOrderTraversal( x.right )



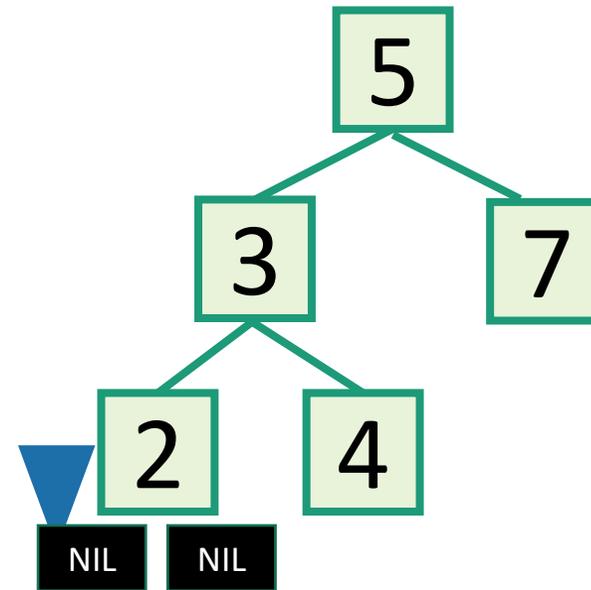
# Aside: In-Order Traversal of BSTs

- Output all the elements in sorted order!
- inOrderTraversal(x):
  - if  $x \neq \text{NIL}$ :
    - inOrderTraversal( x.left )
    - print( x.key )
    - inOrderTraversal( x.right )



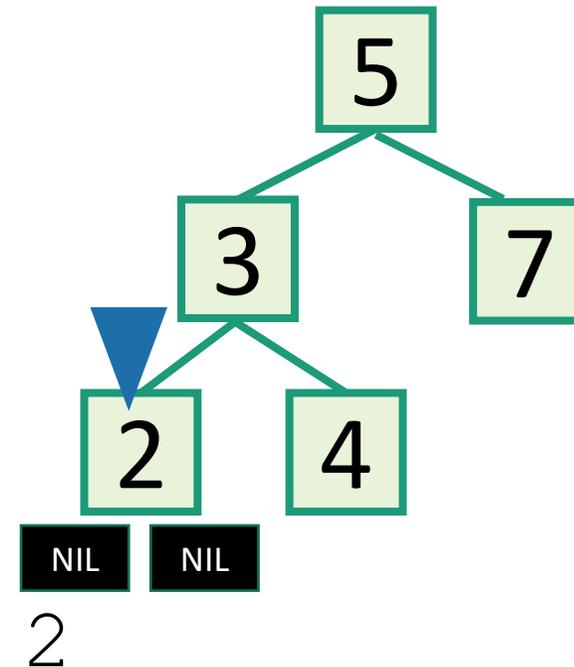
# Aside: In-Order Traversal of BSTs

- Output all the elements in sorted order!
- `inOrderTraversal(x)`:
  - if `x != NIL`:
    - `inOrderTraversal( x.left )`
    - `print( x.key )`
    - `inOrderTraversal( x.right )`



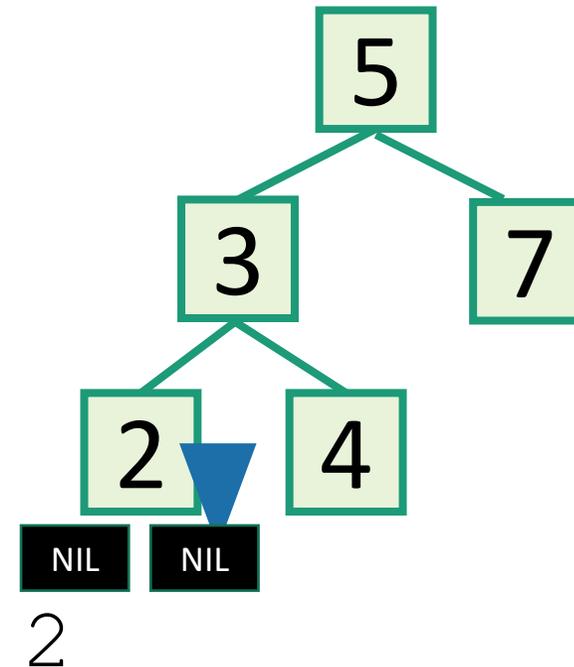
# Aside: In-Order Traversal of BSTs

- Output all the elements in sorted order!
- inOrderTraversal(x):
  - if  $x \neq \text{NIL}$ :
    - inOrderTraversal( x.left )
    - print( x.key )
    - inOrderTraversal( x.right )



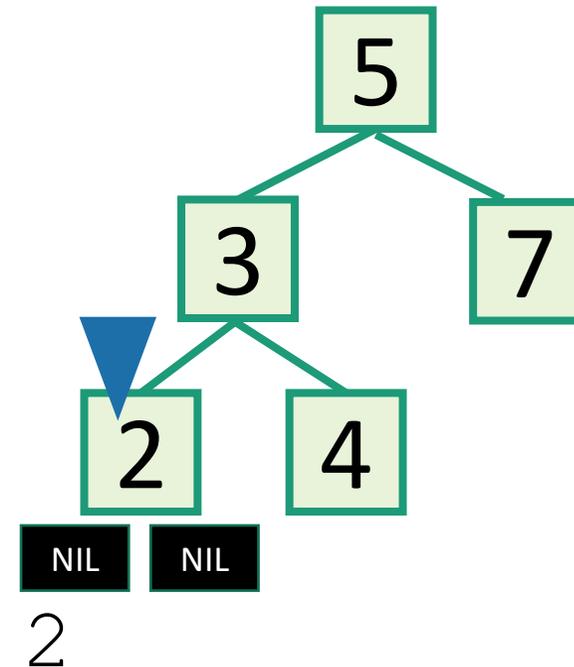
# Aside: In-Order Traversal of BSTs

- Output all the elements in sorted order!
- inOrderTraversal(x):
  - if  $x \neq \text{NIL}$ :
    - inOrderTraversal( x.left )
    - print( x.key )
    - inOrderTraversal( x.right )



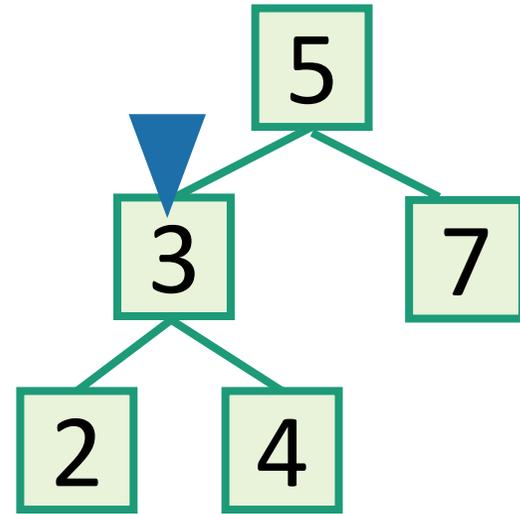
# Aside: In-Order Traversal of BSTs

- Output all the elements in sorted order!
- inOrderTraversal(x):
  - if  $x \neq \text{NIL}$ :
    - inOrderTraversal( x.left )
    - print( x.key )
    - inOrderTraversal( x.right )



# Aside: In-Order Traversal of BSTs

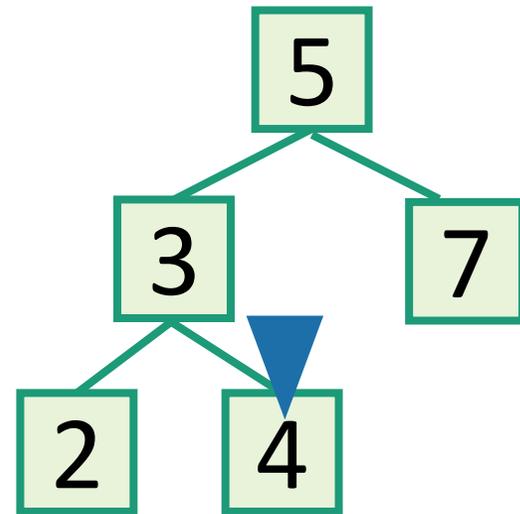
- Output all the elements in sorted order!
- `inOrderTraversal(x)`:
  - if `x != NIL`:
    - `inOrderTraversal( x.left )`
    - `print( x.key )`
    - `inOrderTraversal( x.right )`



2 3

# Aside: In-Order Traversal of BSTs

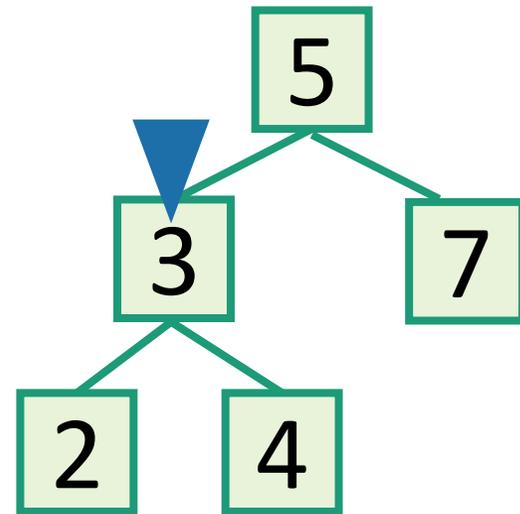
- Output all the elements in sorted order!
- `inOrderTraversal(x)`:
  - if `x != NIL`:
    - `inOrderTraversal( x.left )`
    - `print( x.key )`
    - `inOrderTraversal( x.right )`



2 3 4

# Aside: In-Order Traversal of BSTs

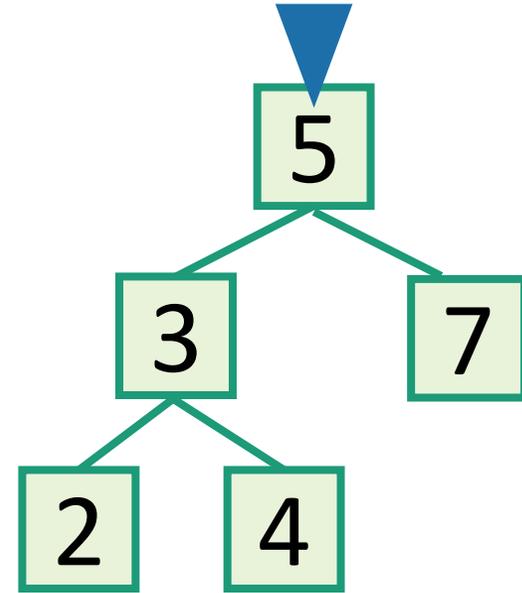
- Output all the elements in sorted order!
- `inOrderTraversal(x)`:
  - if `x != NIL`:
    - `inOrderTraversal( x.left )`
    - `print( x.key )`
    - `inOrderTraversal( x.right )`



2 3 4

# Aside: In-Order Traversal of BSTs

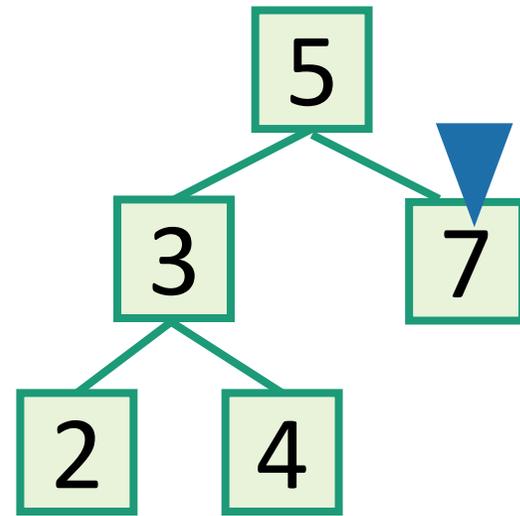
- Output all the elements in sorted order!
- `inOrderTraversal(x)`:
  - if `x != NIL`:
    - `inOrderTraversal( x.left )`
    - `print( x.key )`
    - `inOrderTraversal( x.right )`



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)`:

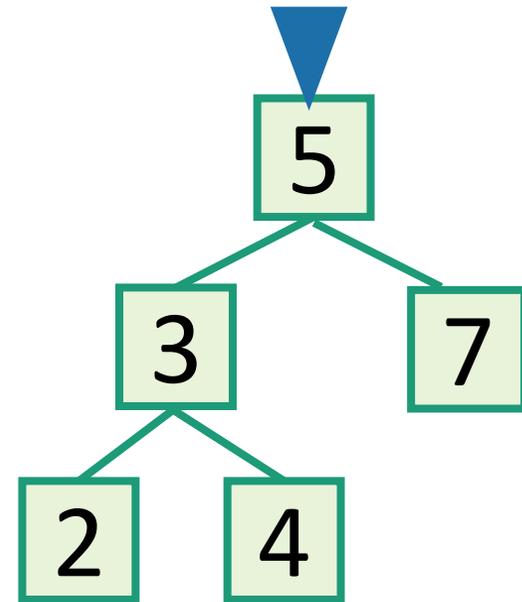
- if `x != NIL`:

- `inOrderTraversal( x.left )`

- `print( x.key )`

- `inOrderTraversal( x.right )`

- Runs in time  $O(n)$ .



2 3 4 5 7 Sorted!

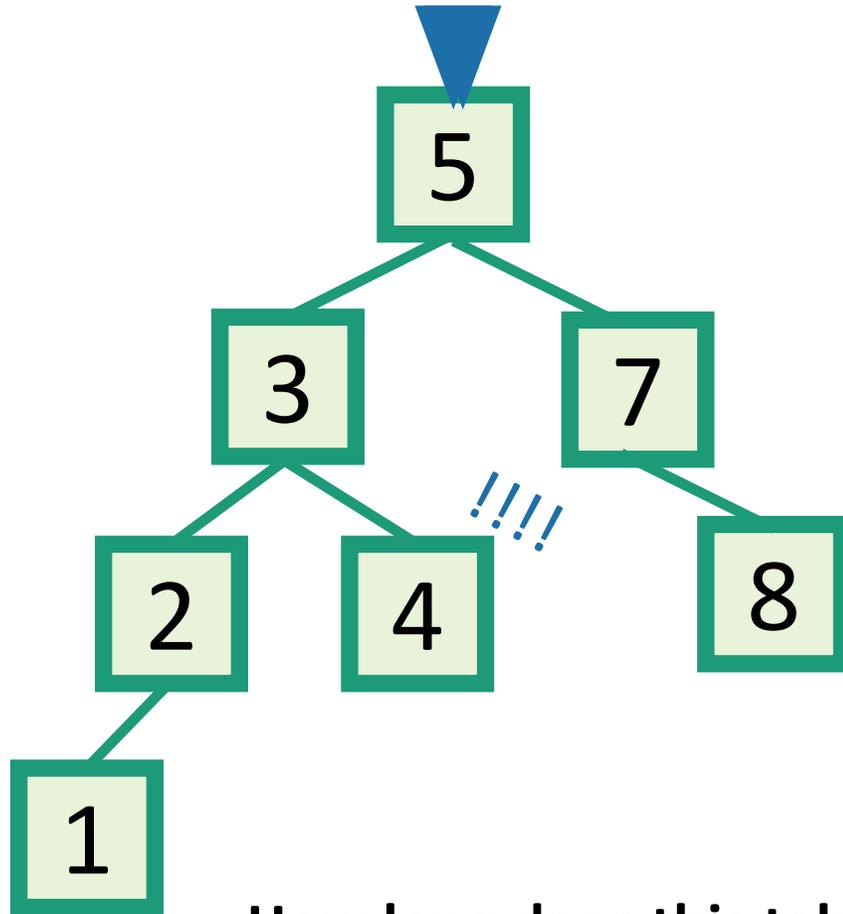
# Back to the goal

Fast **SEARCH**/**INSERT**/**DELETE**

Can we do these?

# SEARCH in a Binary Search Tree

definition by example



**How long does this take?**

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

**EXAMPLE:** Search for 4.

**EXAMPLE:** Search for 4.5

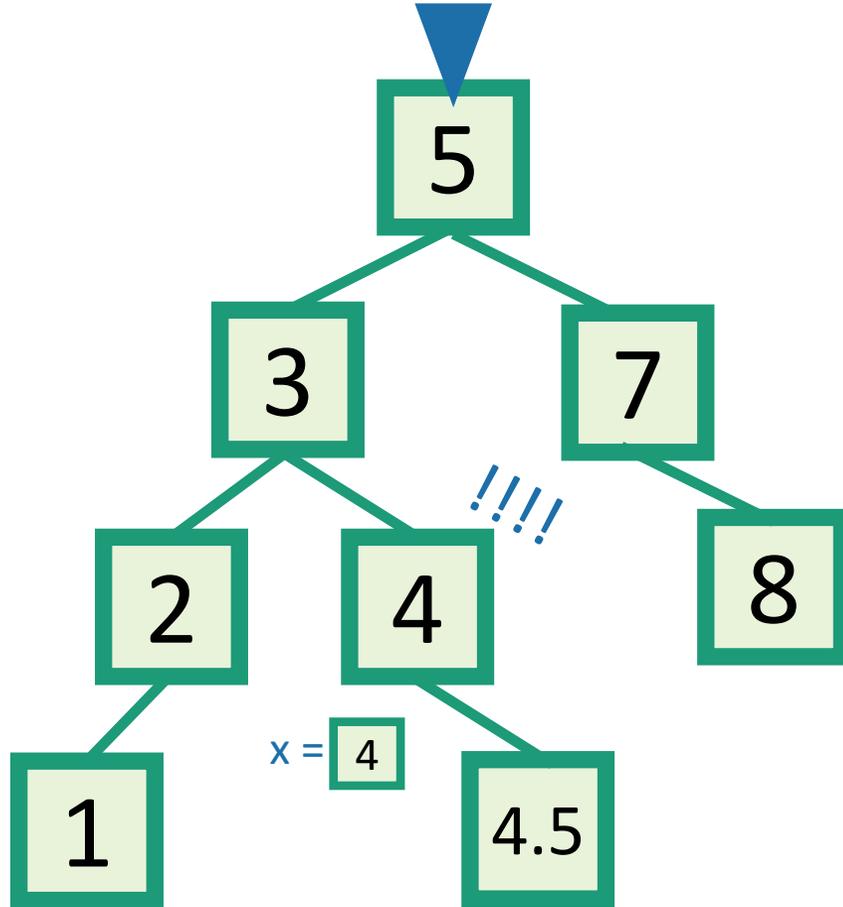
- It turns out it will be convenient to **return 4** in this case
- (that is, **return** the last node before we went off the tree)

Write pseudocode  
(or actual code) to  
implement this!



# INSERT in a Binary Search Tree

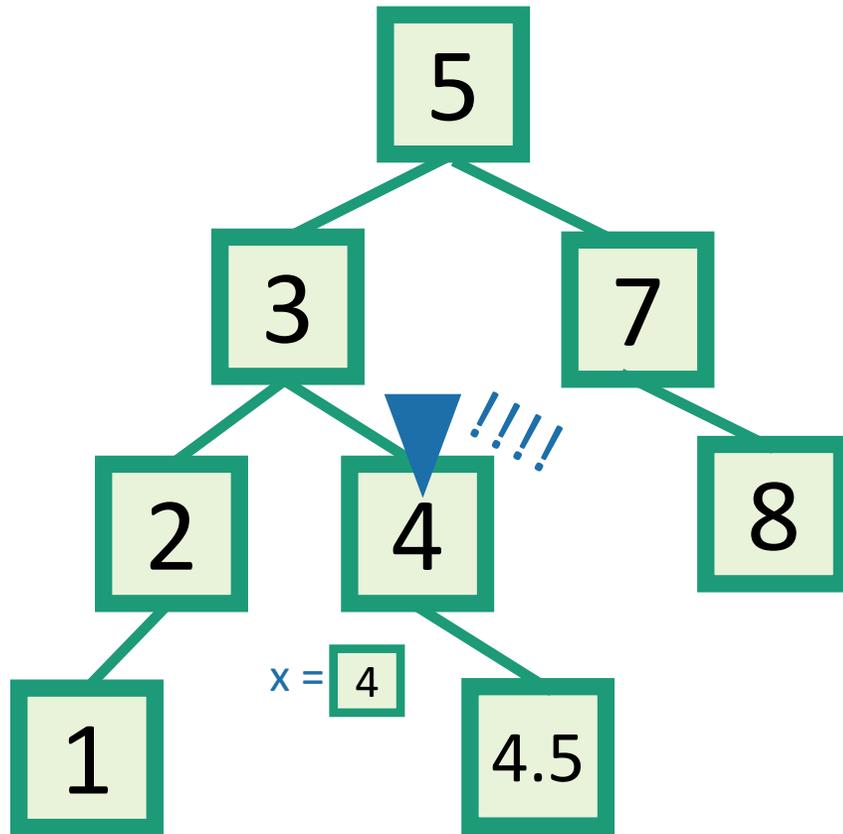
EXAMPLE: Insert 4.5



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

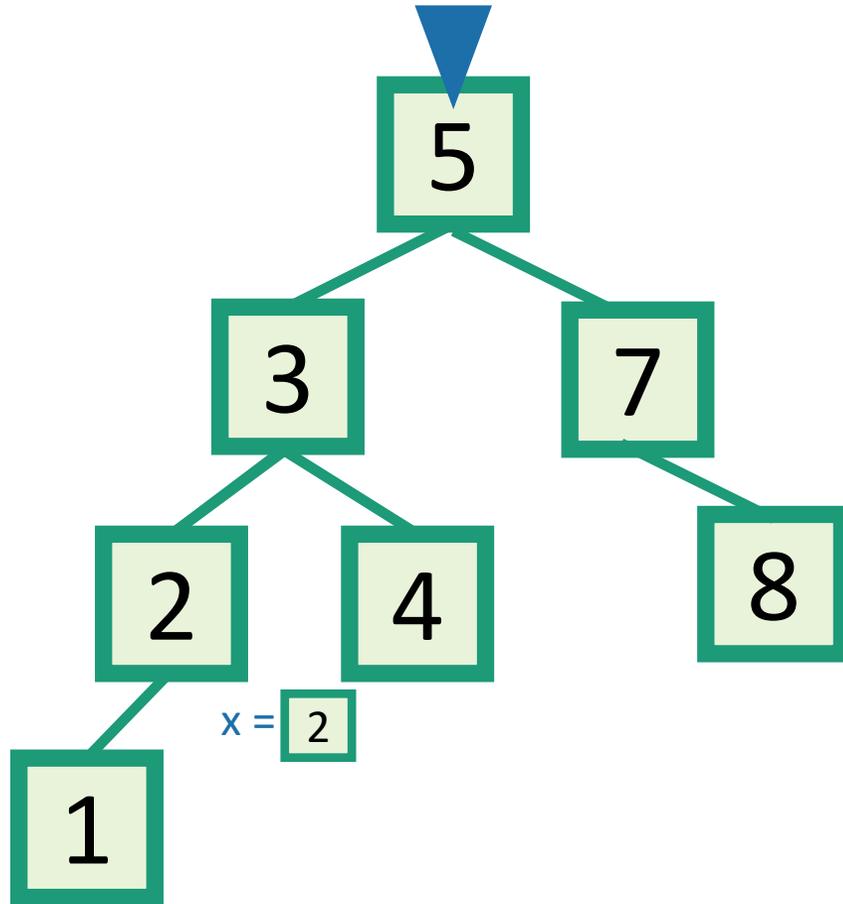
# INSERT in a Binary Search Tree

**EXAMPLE:** Insert 4.5



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

# DELETE in a Binary Search Tree



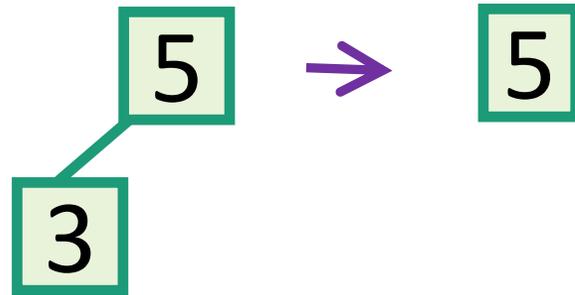
EXAMPLE: Delete 2

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

# DELETE in a Binary Search Tree

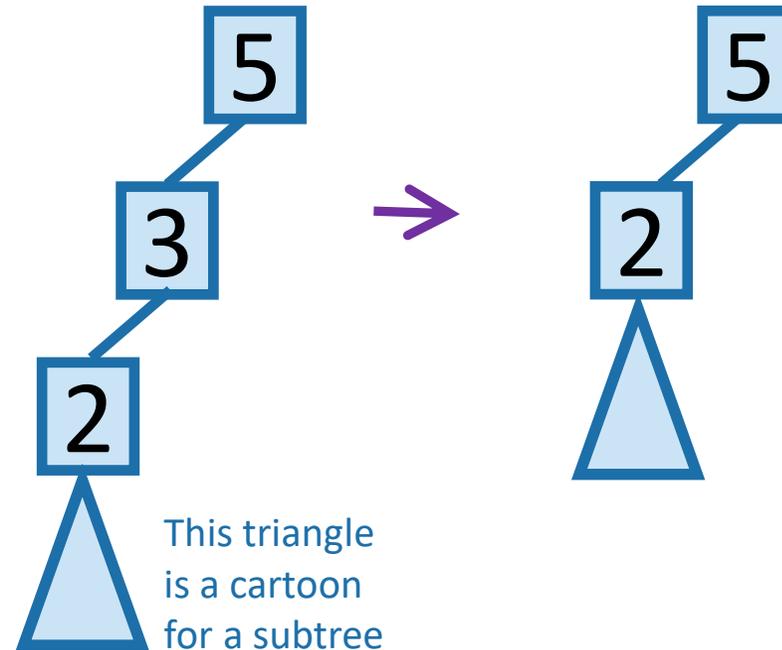
several cases (by example)

say we want to delete 3



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

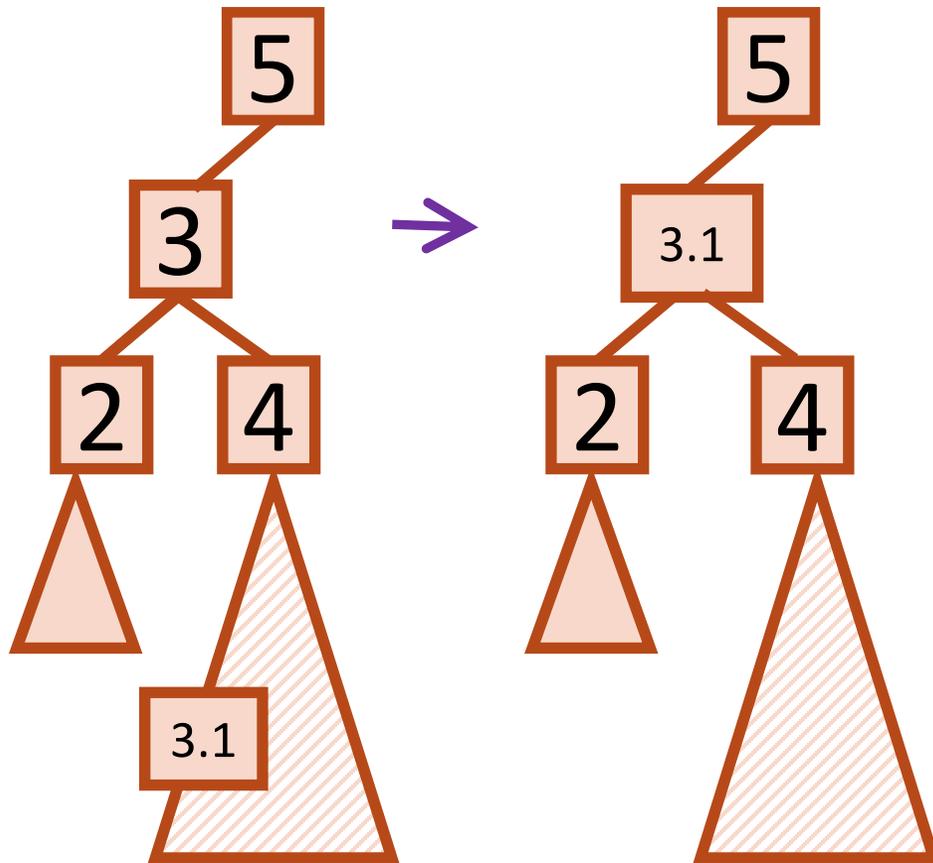
Write pseudocode for all of these!



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

# DELETE in a Binary Search Tree

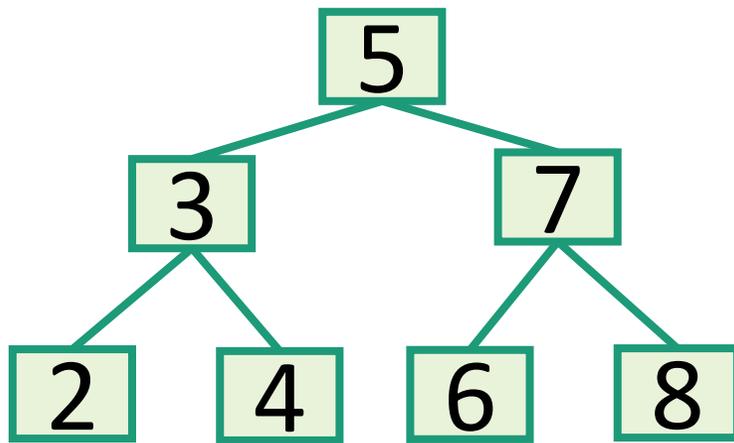
**Case 3:** if 3 has two children, replace 3 with its **immediate successor**. (aka, next biggest thing after 3)



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

# 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 =  $O(\text{height of tree})$

Trees have depth  $O(\log(n))$ .

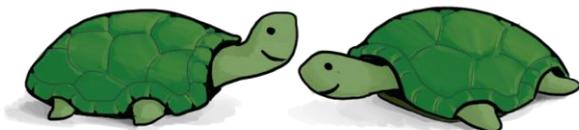
**Done!**



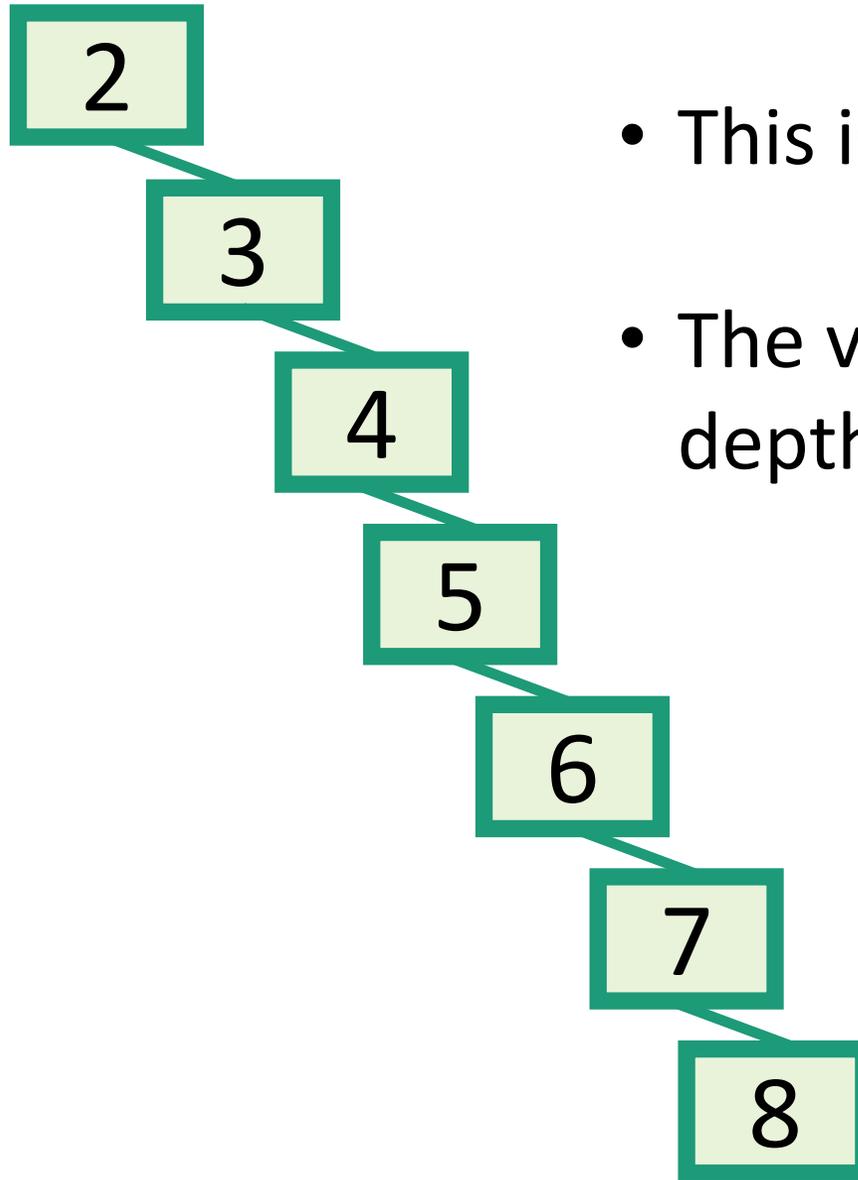
Wait a second...



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**  $O(\log(n))$ .

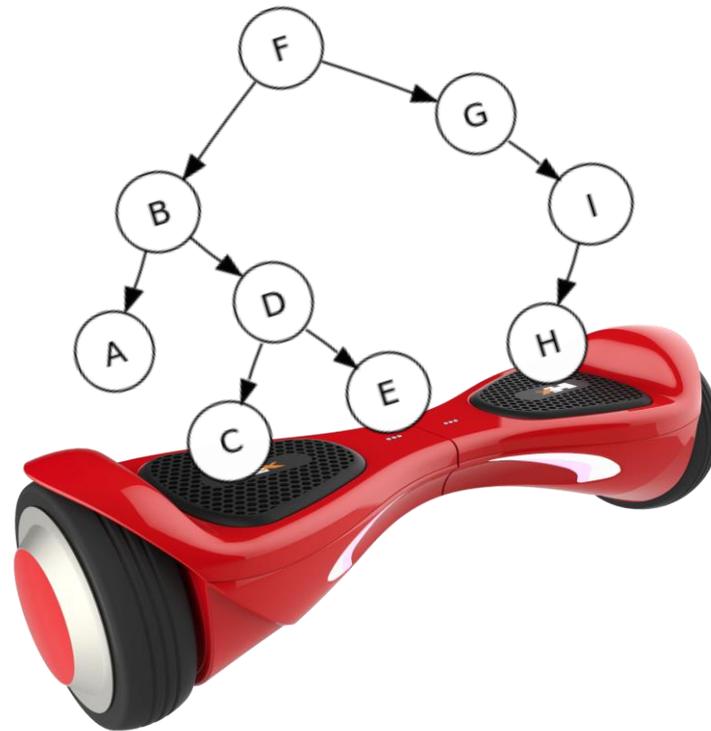
# What to do?

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

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



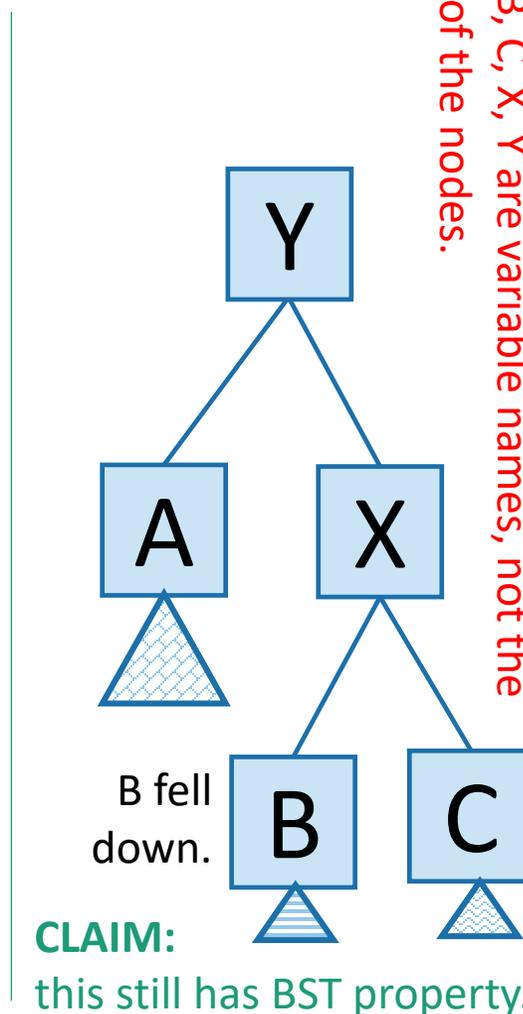
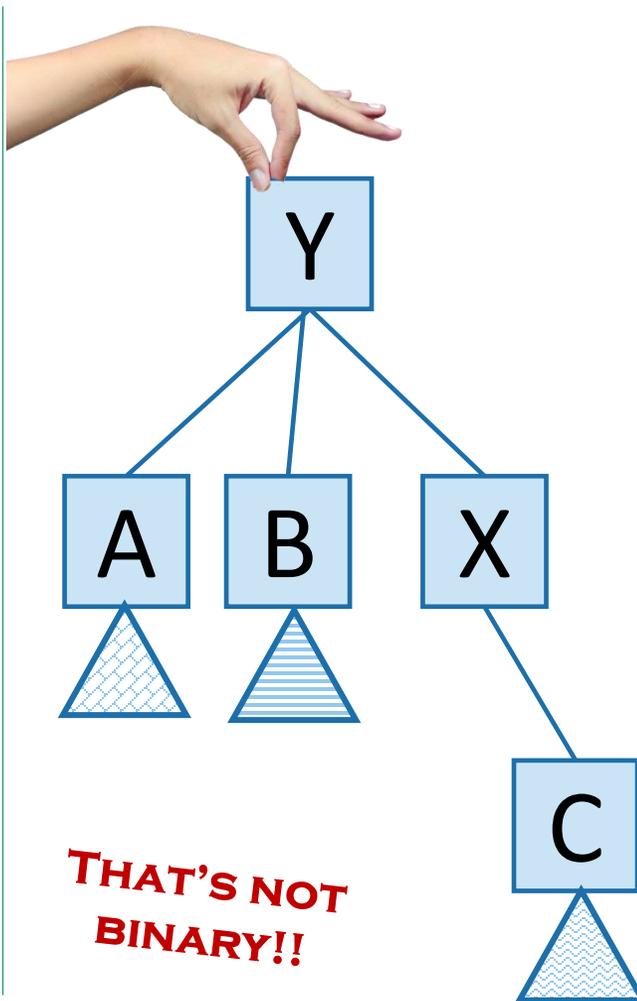
# Self-Balancing Binary Search Trees



# Idea 1: Rotations

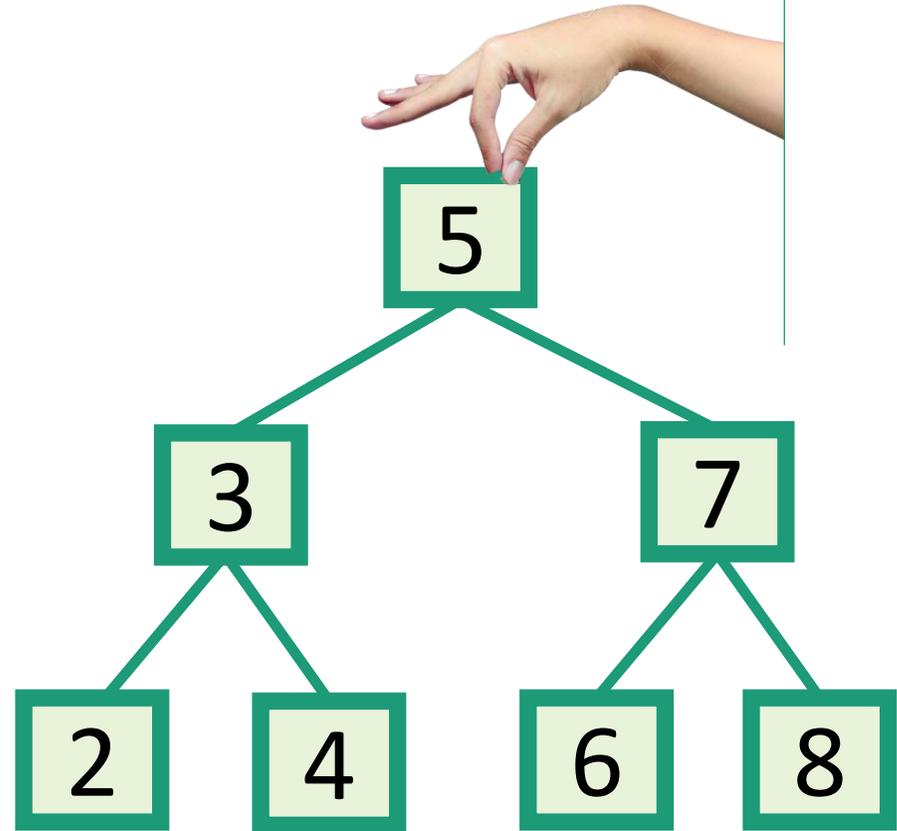
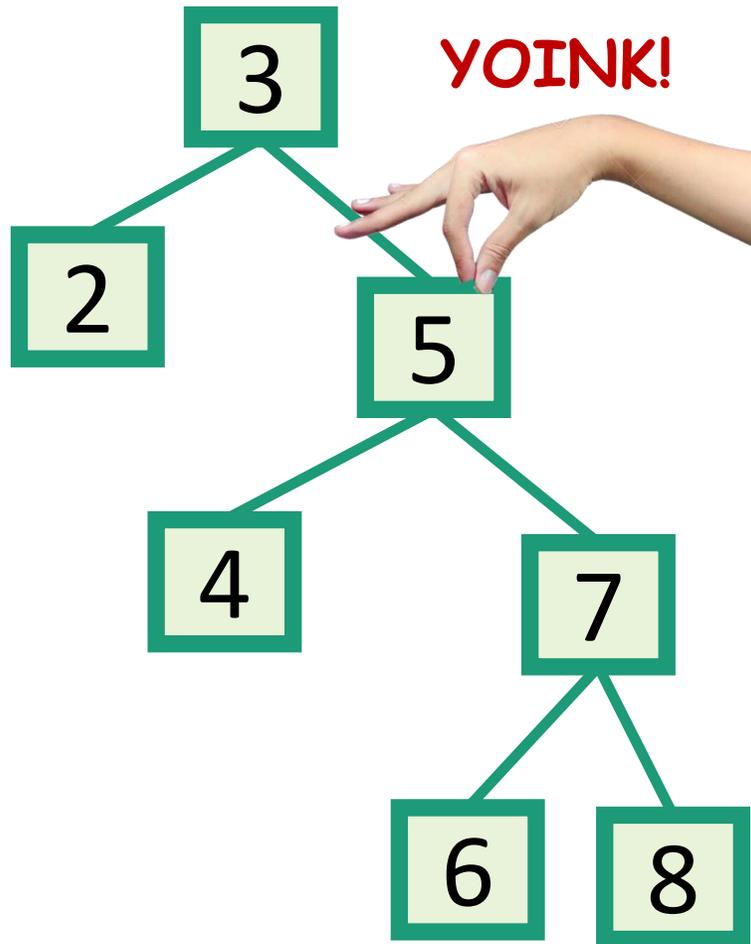
No matter what lives underneath A,B,C, this takes time  $O(1)$ . (Why?)

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



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

This seems helpful



# Strategy?

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



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.

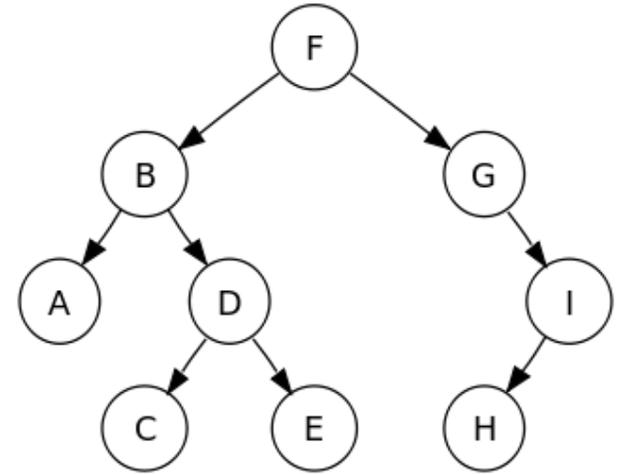


There are actually several ways to do this, but we'll see:

1. AVL Tree (In this course)
2. Multiway-Search Tree (2-4 Tree)
3. Red-Black Tree

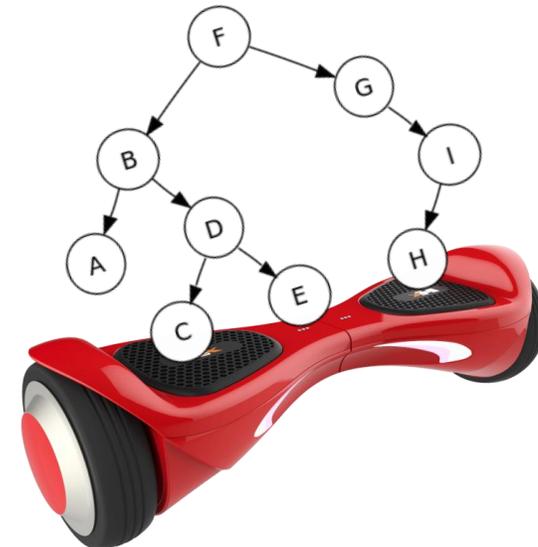
# Recap

- Begin a brief foray into data structures!
- Binary search trees
  - They are better when they're balanced.



this will lead us to...

- Self-Balancing Binary Search Trees
  - AVL Tree
  - Multiway-Search Tree
  - **Red-Black Tree**



# Acknowledgement

- Stanford University

Thank You