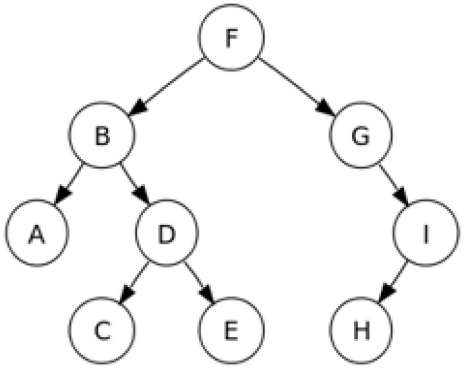
Binary Search Trees

二元查找树

Contents



- Begin a brief foray into data structures!
- Binary search trees

this will lead us to…

数据结构：
链表（串列）

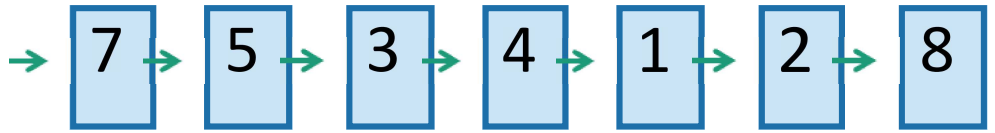
Linked List

UNSorted linked lists

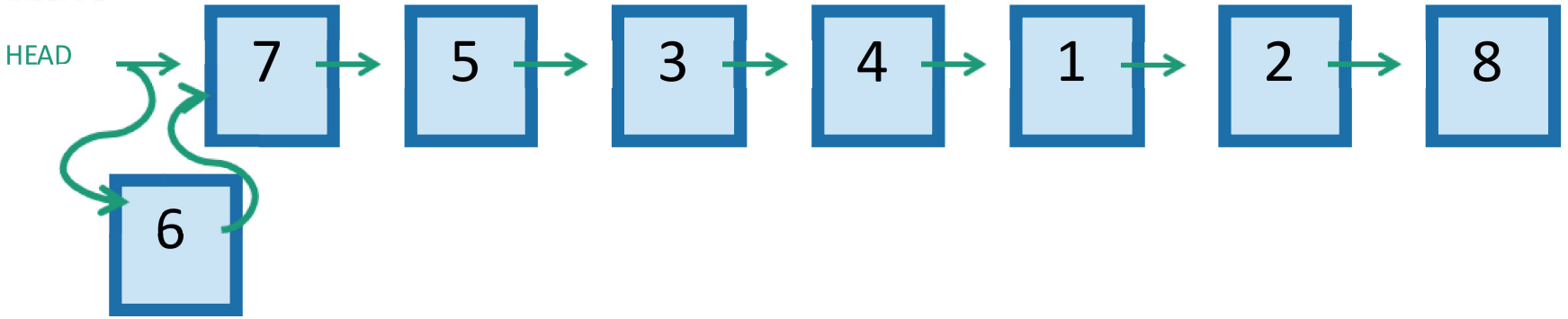
Some data structures

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

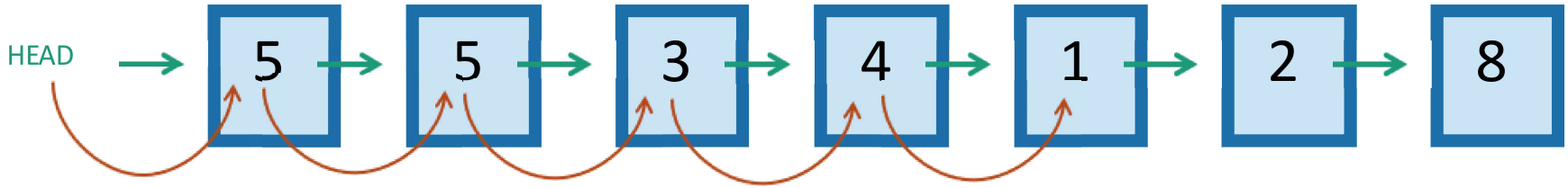
- O(1) INSERT:



eg, insert 6



- O(n) SEARCH/DELETE:



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

想一想，练一练

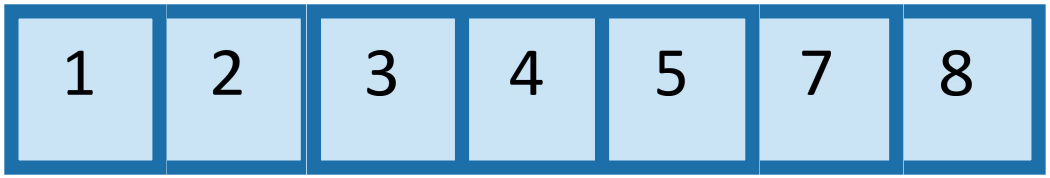
- 如何实现一个串列（linked list）？
- 保存 一组7,5,3,4,1,2, 8
- 增加9，删除4

- 提示：
  - python内置了list类型，但不使用
  - 可利用[*args]添加数组

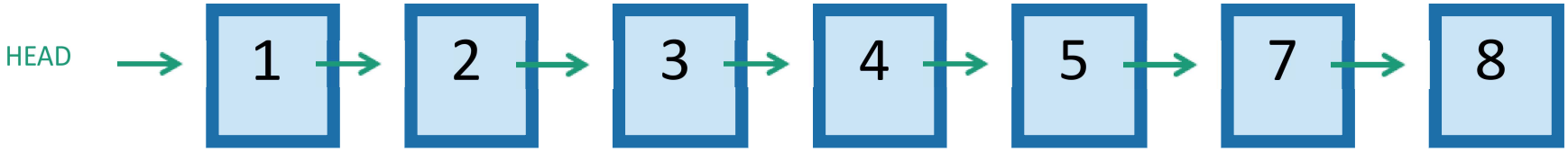
Some data structu 5

for storing objects like (aka, nodes with keys)

- (Sorted) arrays:



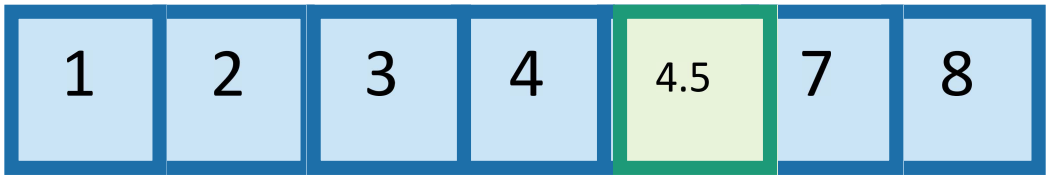
- (Sorted) linked lists:



- Some basic operations:
  - INSERT, DELETE, SEARCH

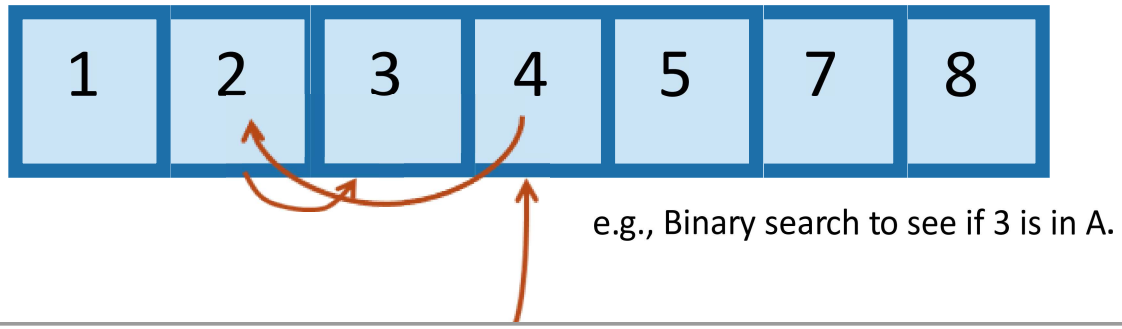
Sorted Arrays

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



e.g., insert 4.5

- O(log(n)) SEARCH:



e.g., Binary search to see if 3 is in A.

Motivation for Binary Search Trees

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

数据结构：二元搜索树

Binary Search Tree

Motivation for Binary Search Trees

TODAY!

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

For today all keys are distinct.

Binary tree terminology

Each node has two children.

The left child of 3 is 2

The right child of 3 is 4

The parent of 3 is 5

2 a descendant of 5

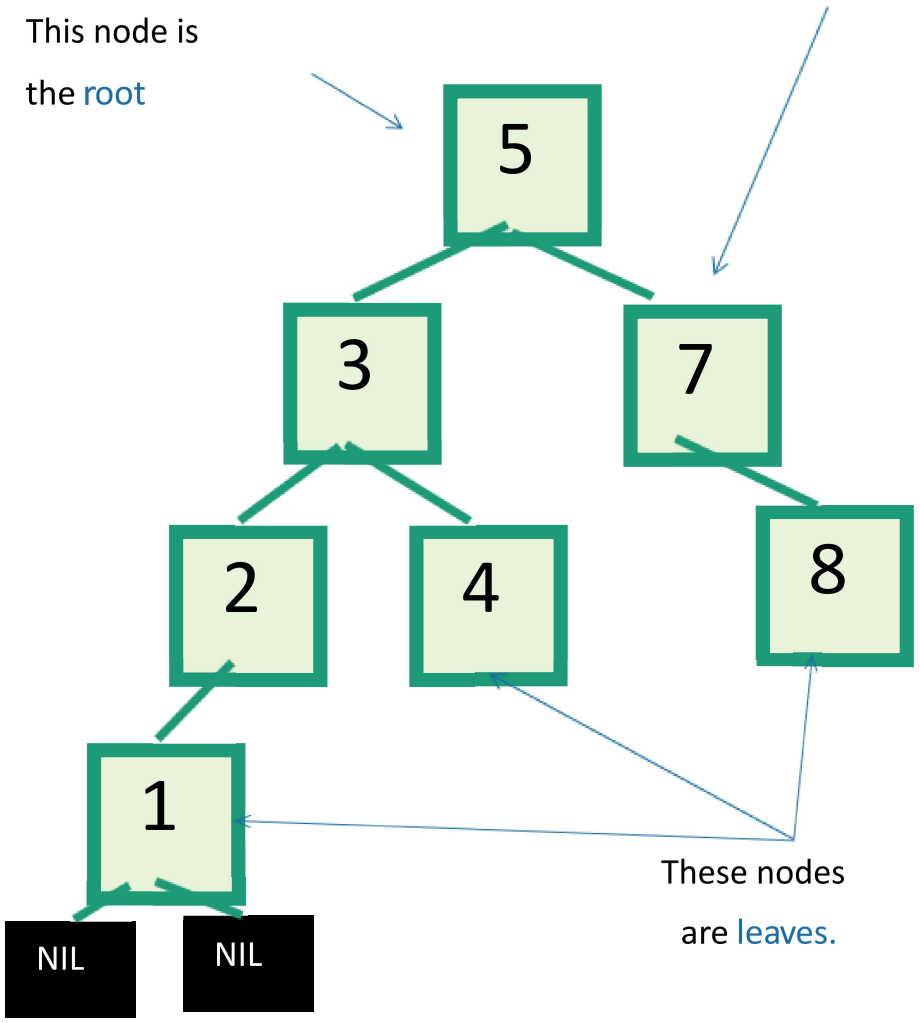
Each node has a pointer to its left child, right child, and parent.

Both children of 1 are NIL

(I won't usually draw them).

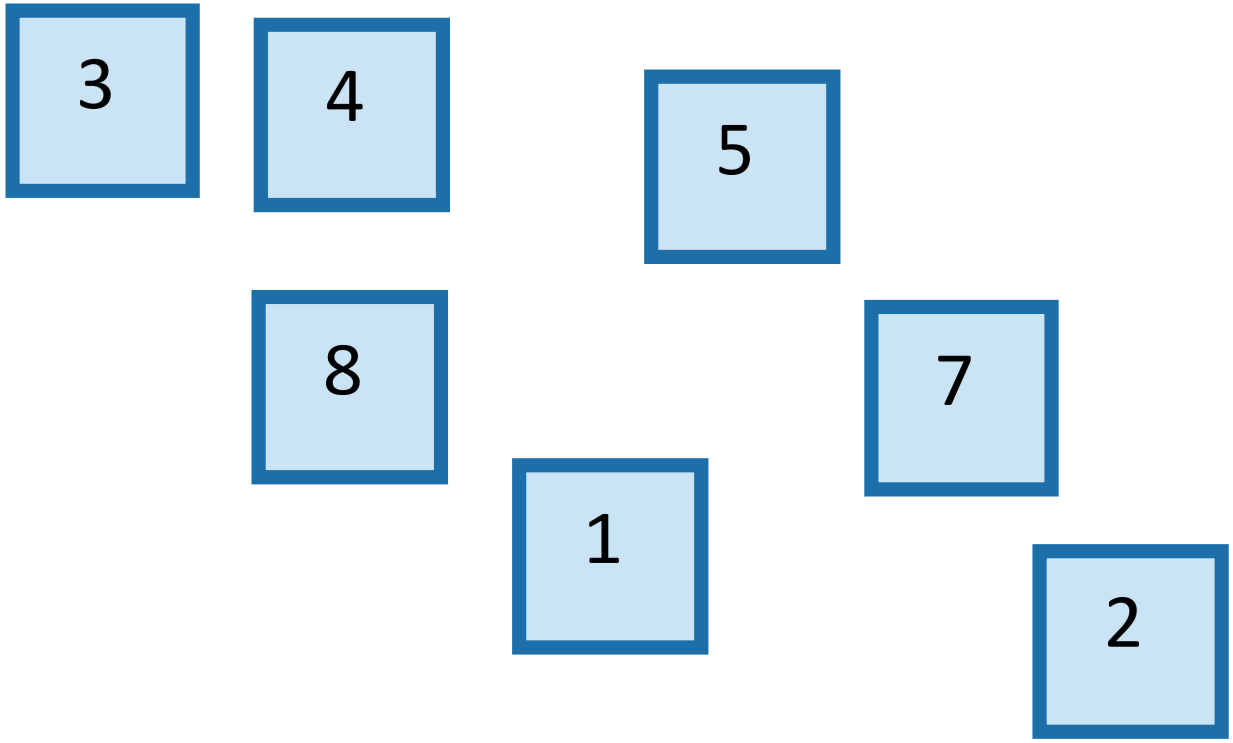
The height of this tree is 3.

(Max number of edges from the root to a leaf).



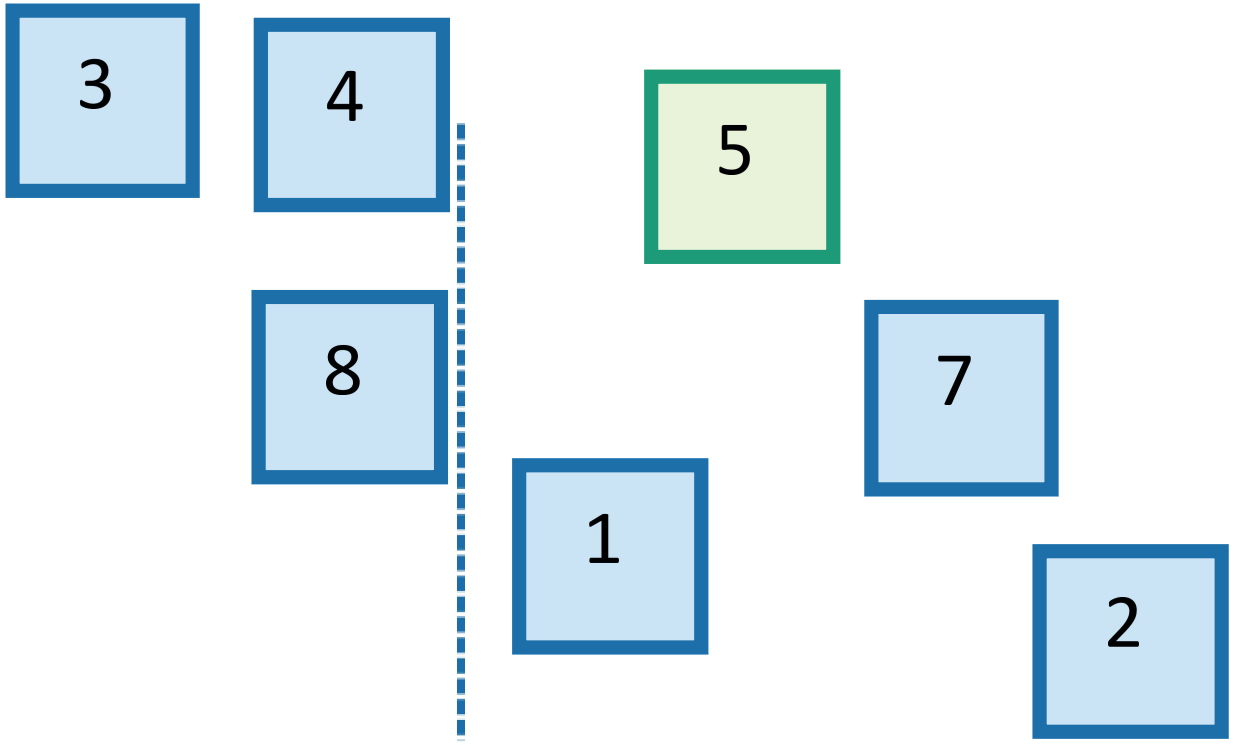
Binary Search Trees

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



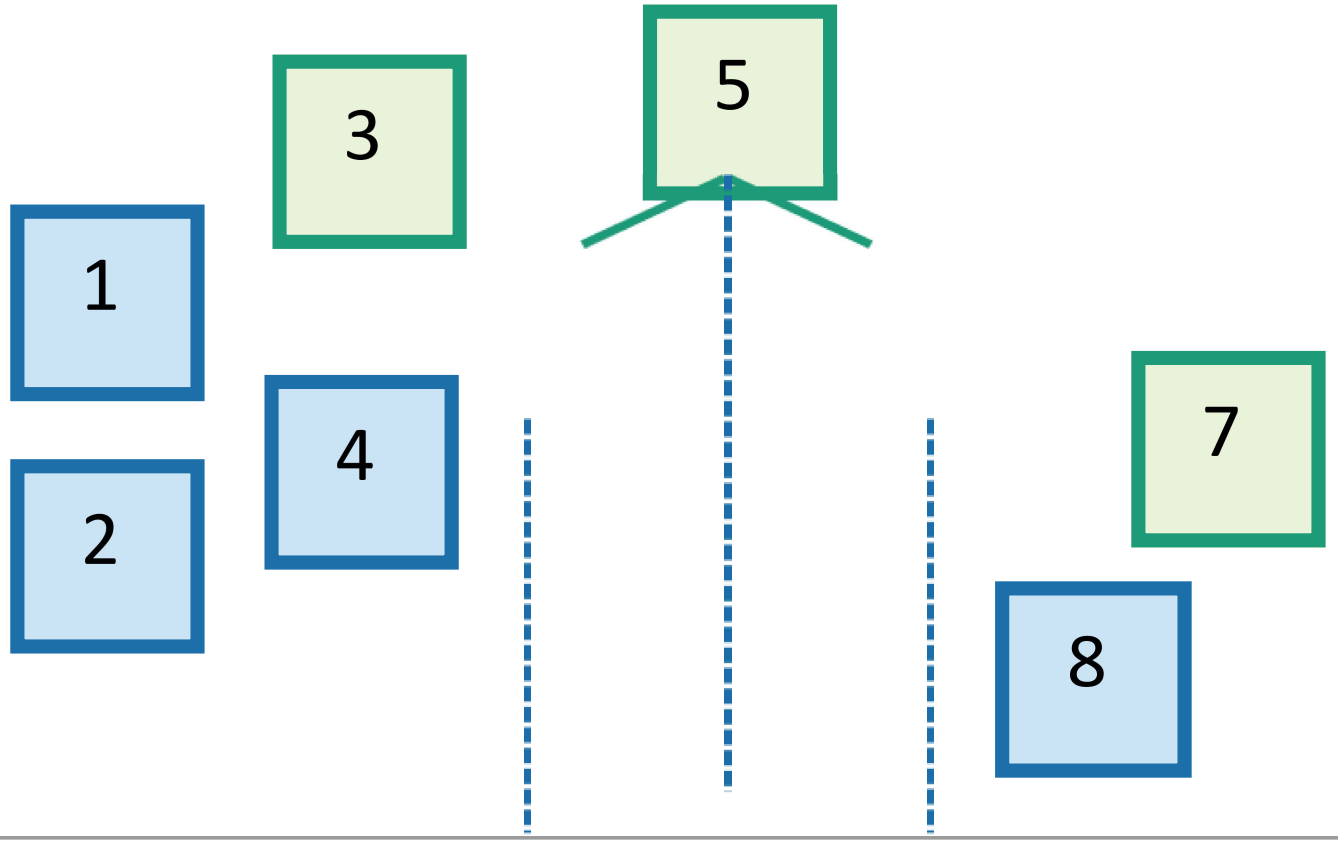
Binary Search Trees

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



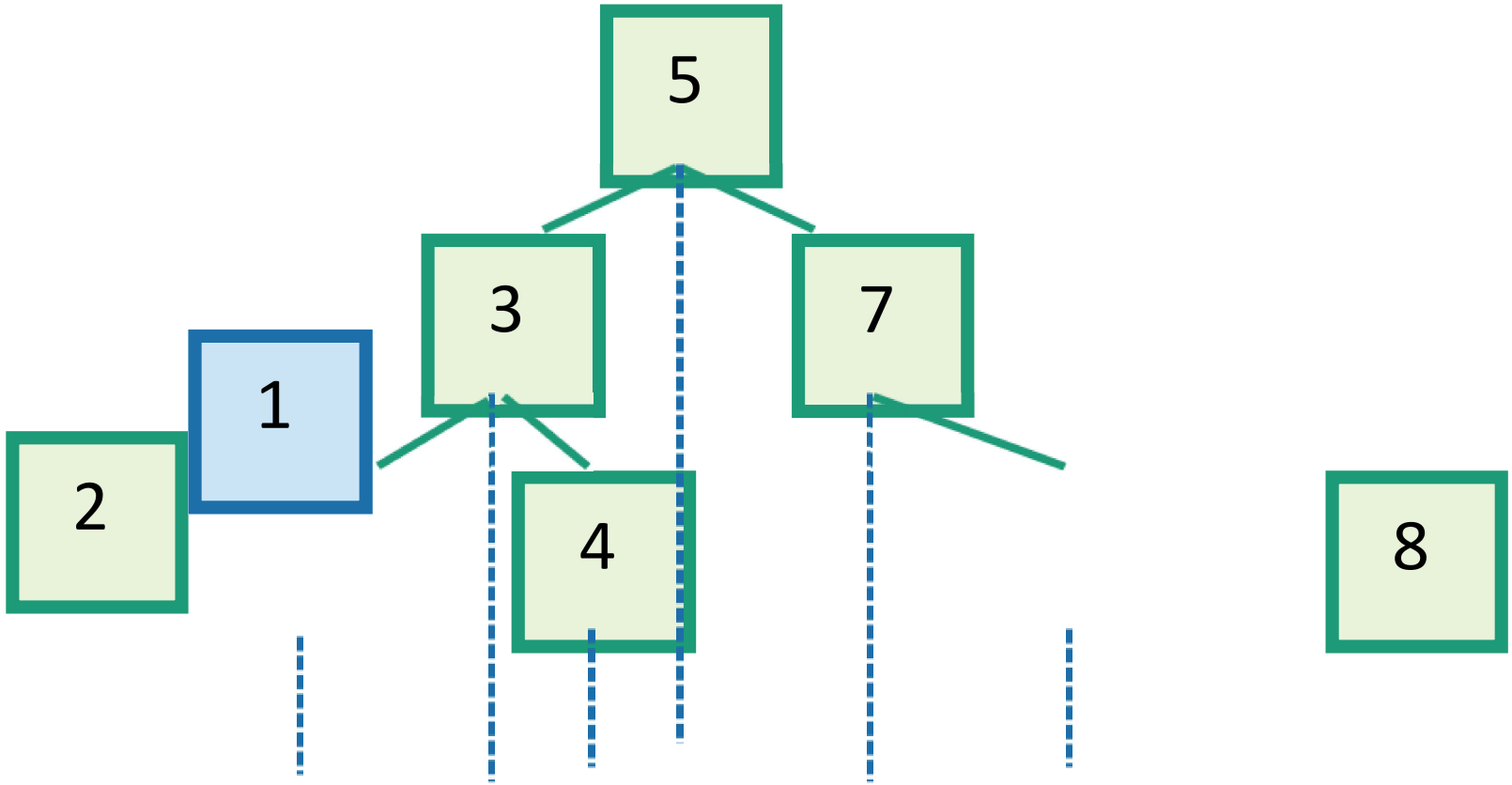
Binary Search Trees

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



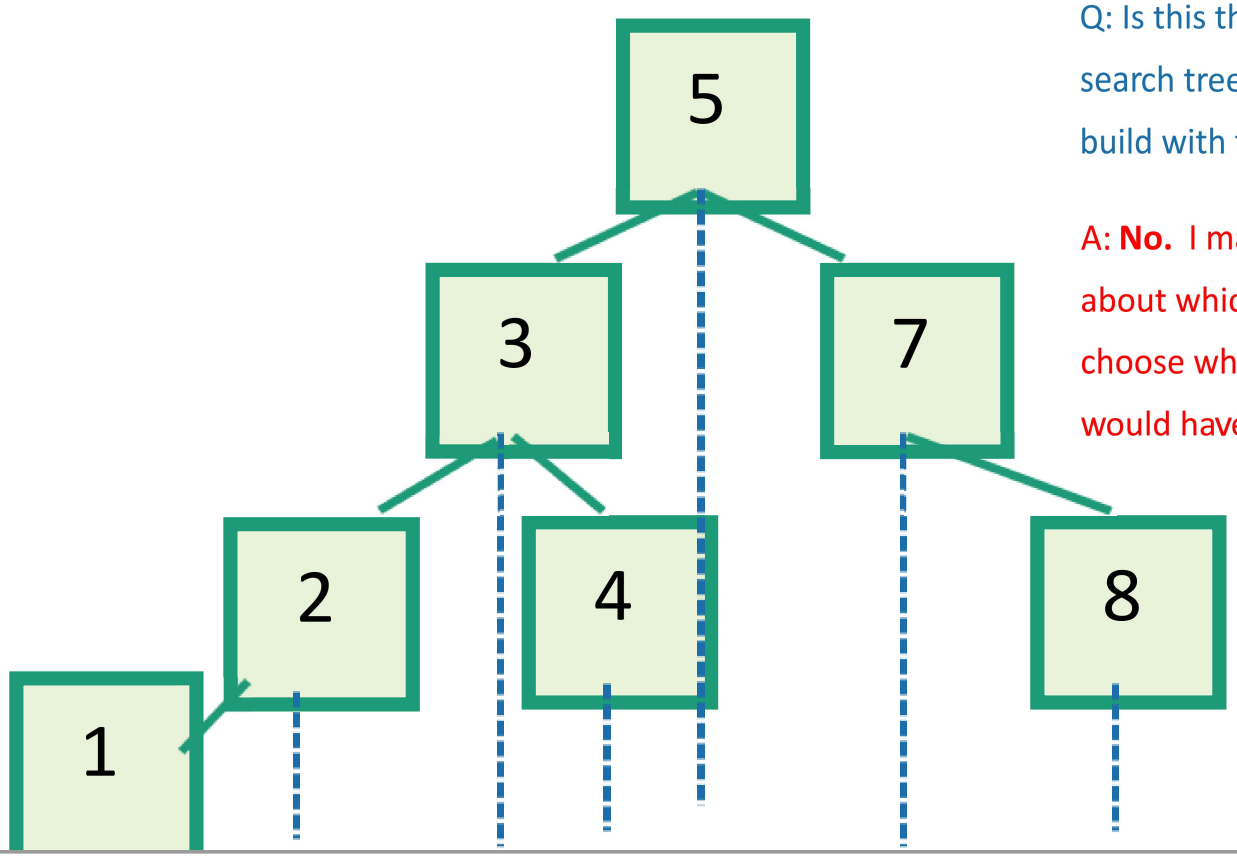
Binary Search Trees

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



Binary Search Trees

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

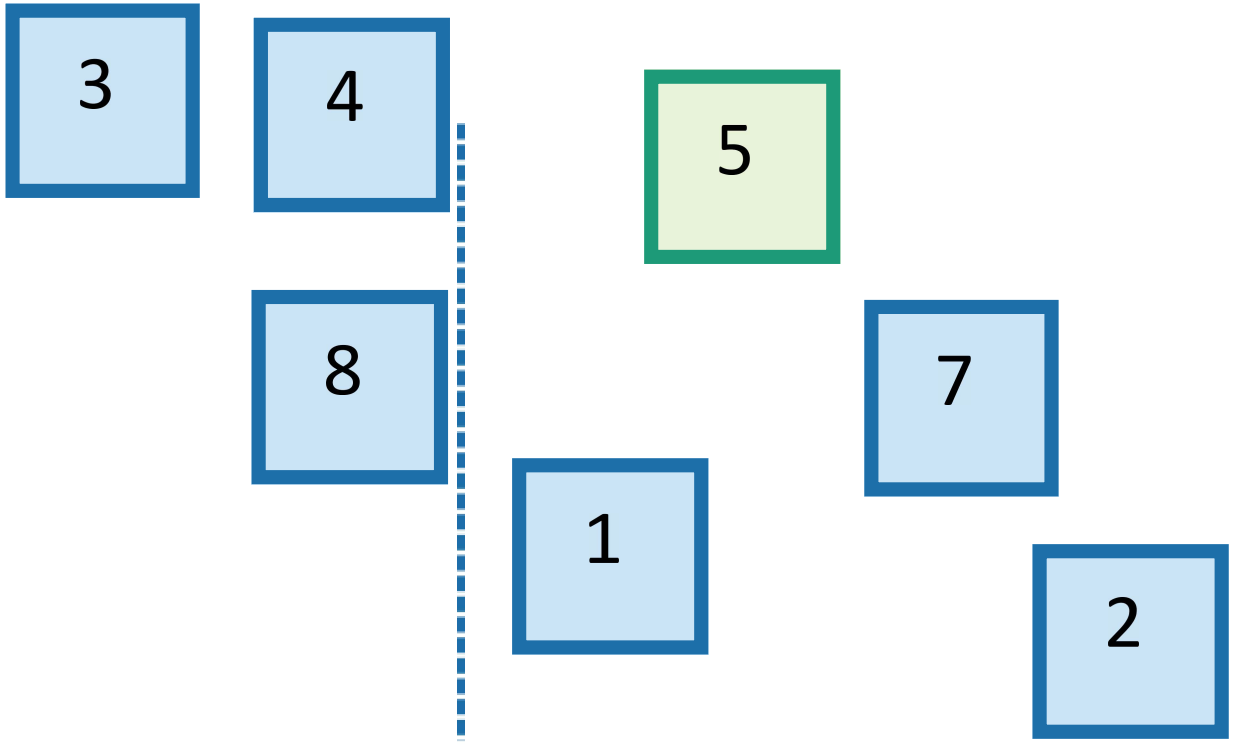


Q: Is this the only binary search tree I could possibly build with these values?

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

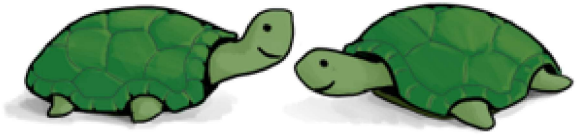
Aside: this should look familiar

kinda like QuickSort

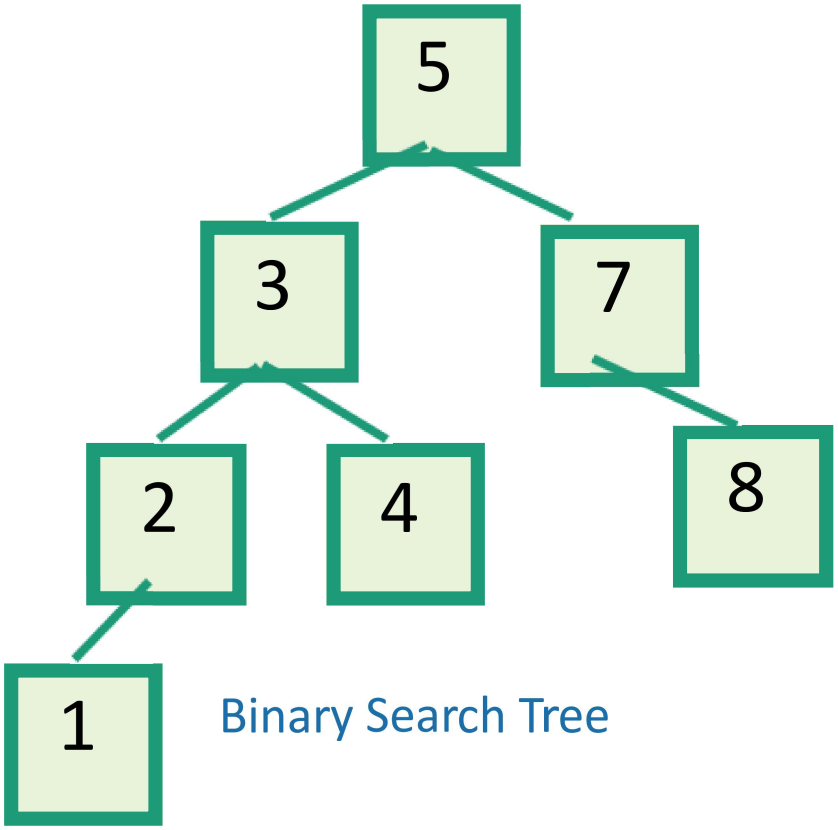


Binary Search Trees

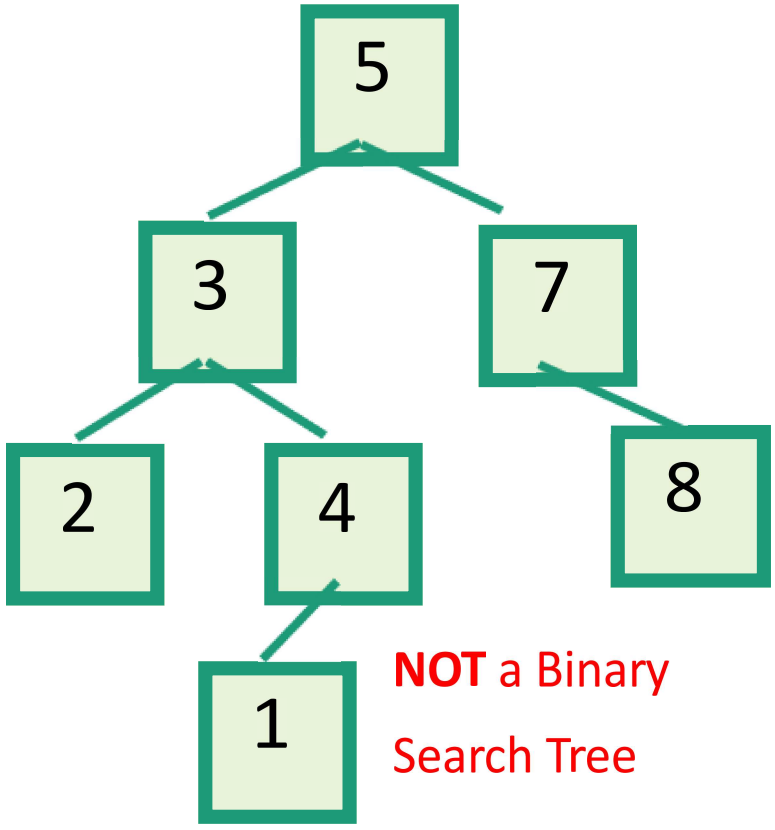
Which of these is a BST?



- 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

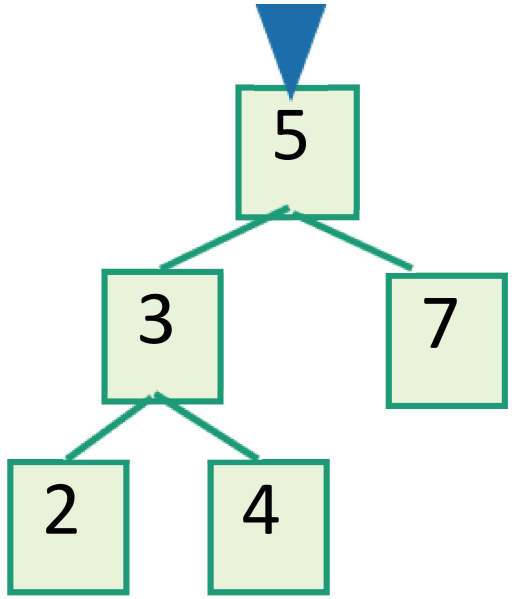
想一想，练一练

- 保存 一组7,5,3,4,1,2, 8到一棵二元搜索树

Aside: In-Order Traversal of BSTs

- Output all the elements in sorted order!

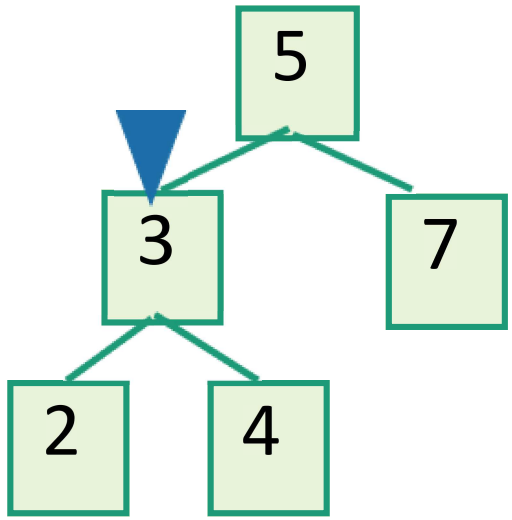
- inOrderTraversal(x):
  - if x!= NIL:
    - inOrderTraversal(x.left)
    - print(x.key)
    - inOrderTraversal(x.right)



Aside: In-Order Traversal of BSTs

- Output all the elements in sorted order!

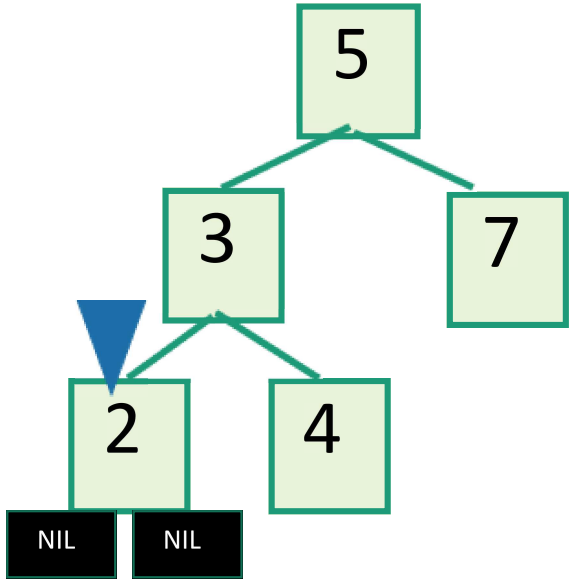
- inOrderTraversal(x):
  - if x!= NIL:
    - inOrderTraversal(x.left)
    - print(x.key)
    - inOrderTraversal(x.right)



Aside: In-Order Traversal of BSTs

- Output all the elements in sorted order!

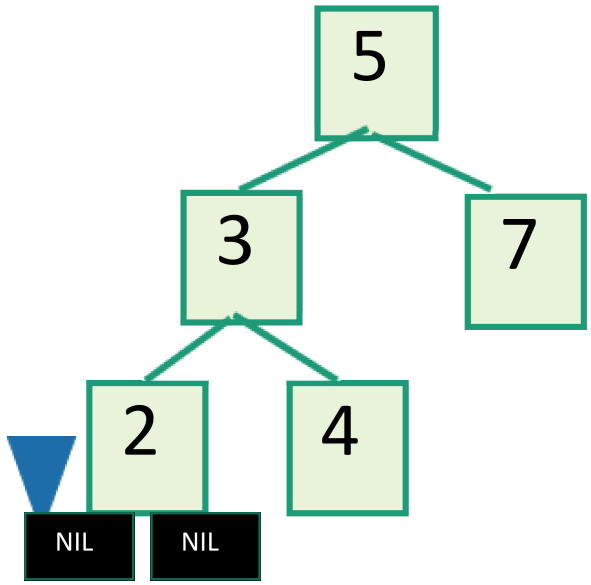
- inOrderTraversal(x):
  - if x!= NIL:
    - inOrderTraversal(x.left)
    - print(x.key)
    - inOrderTraversal(x.right)



Aside: In-Order Traversal of BSTs

- Output all the elements in sorted order!

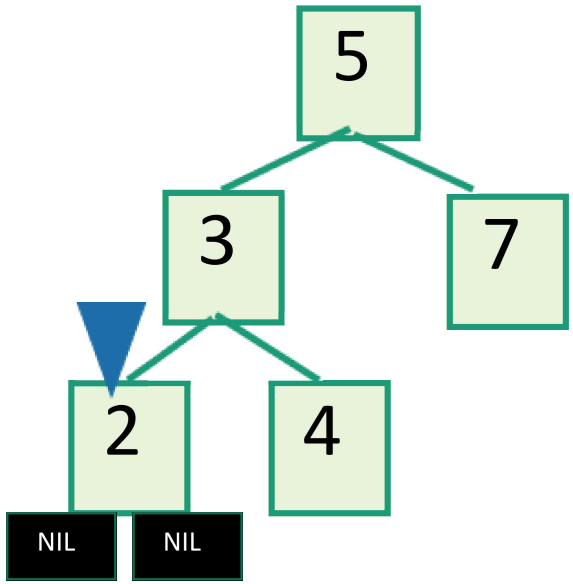
- inOrderTraversal(x):
  - if x!= NIL:
    - inOrderTraversal(x.left)
    - print(x.key)
    - inOrderTraversal(x.right)



Aside: In-Order Traversal of BSTs

- Output all the elements in sorted order!

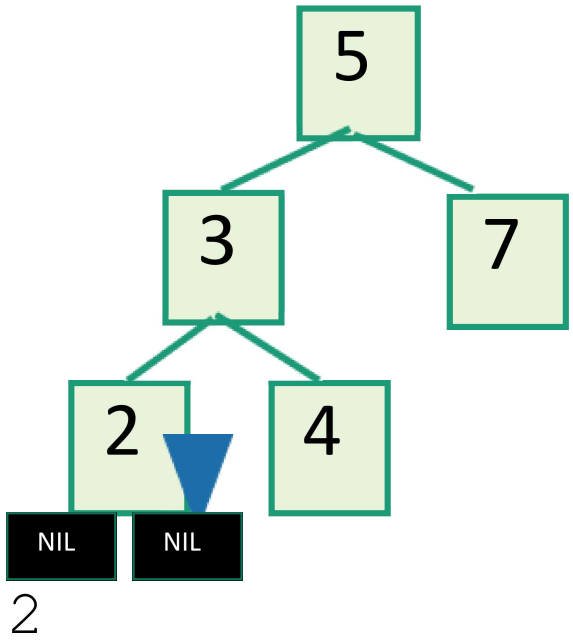
- inOrderTraversal(x):
  - if x!= NIL:
    - inOrderTraversal(x.left)
    - print(x.key)
    - inOrderTraversal(x.right)



Aside: In-Order Traversal of BSTs

- Output all the elements in sorted order!

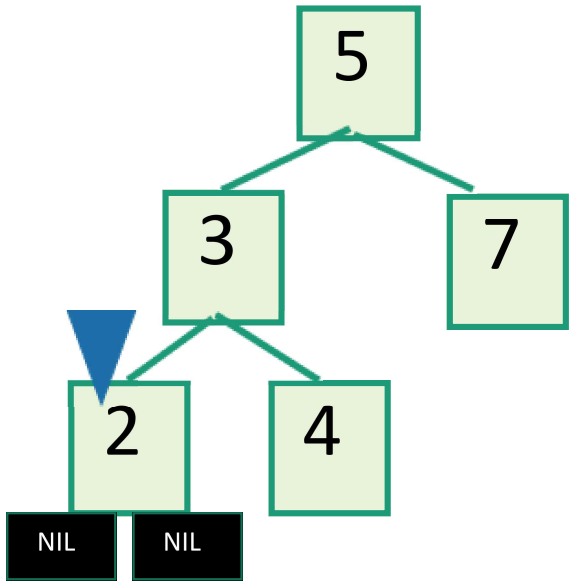
- inOrderTraversal(x):
  - if x!= NIL:
    - inOrderTraversal(x.left)
    - print(x.key)
    - inOrderTraversal(x.right)



Aside: In-Order Traversal of BSTs

- Output all the elements in sorted order!

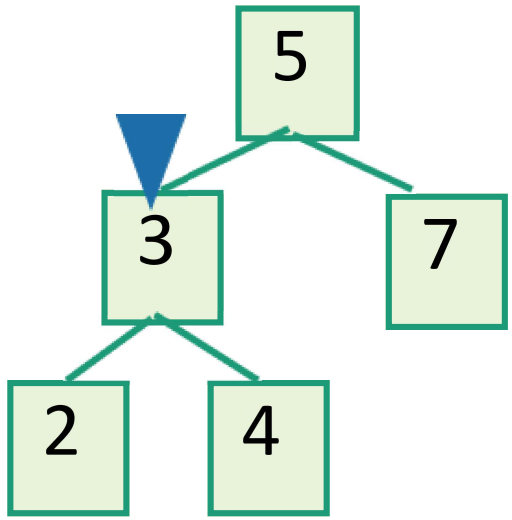
- inOrderTraversal(x):
  - if x!= NIL:
    - inOrderTraversal(x.left)
    - print(x.key)
    - inOrderTraversal(x.right)



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)

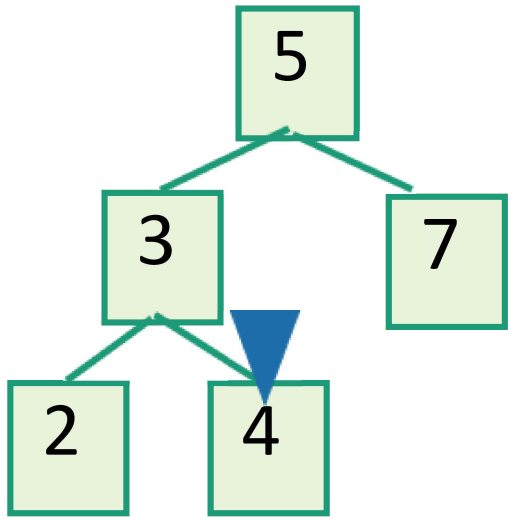


2 3

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)

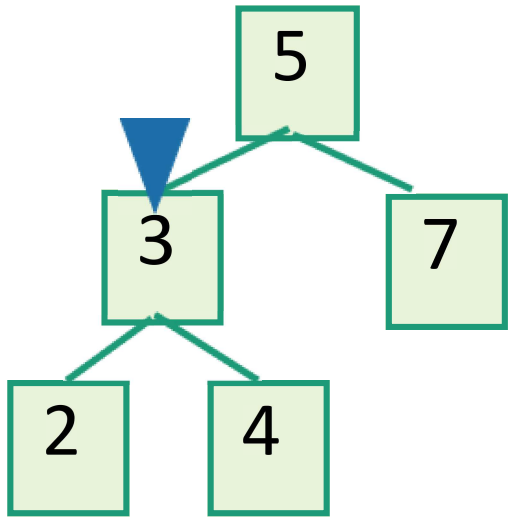


2 3 4

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)

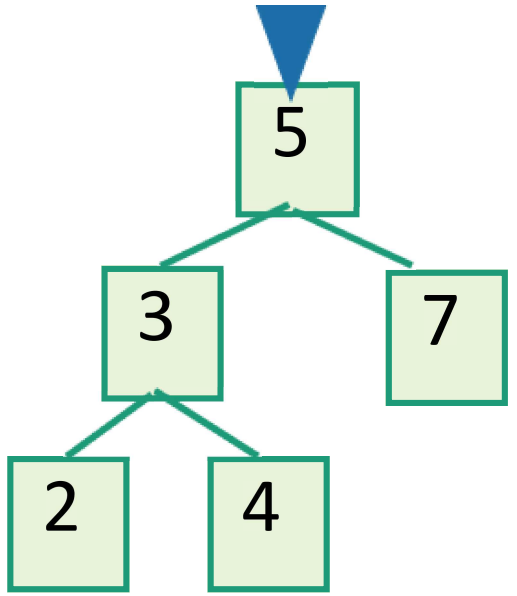


2 3 4

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)

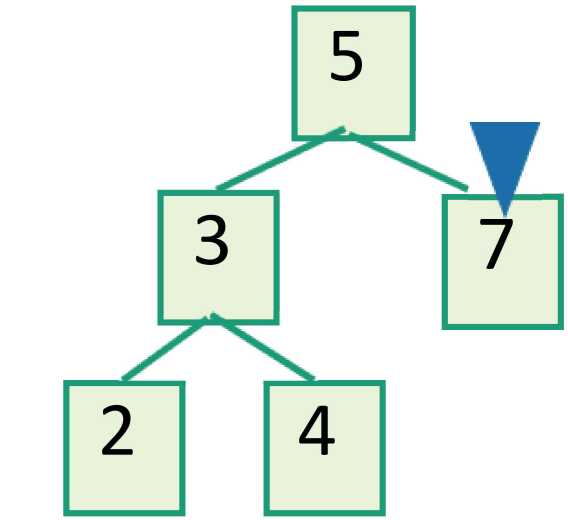


2 3 4 5

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)

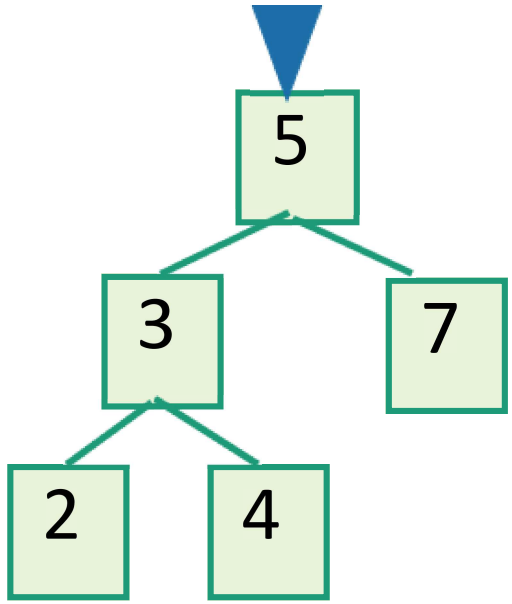


2 3 4 5 7

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!

想一想，练一练

遍历一棵树?

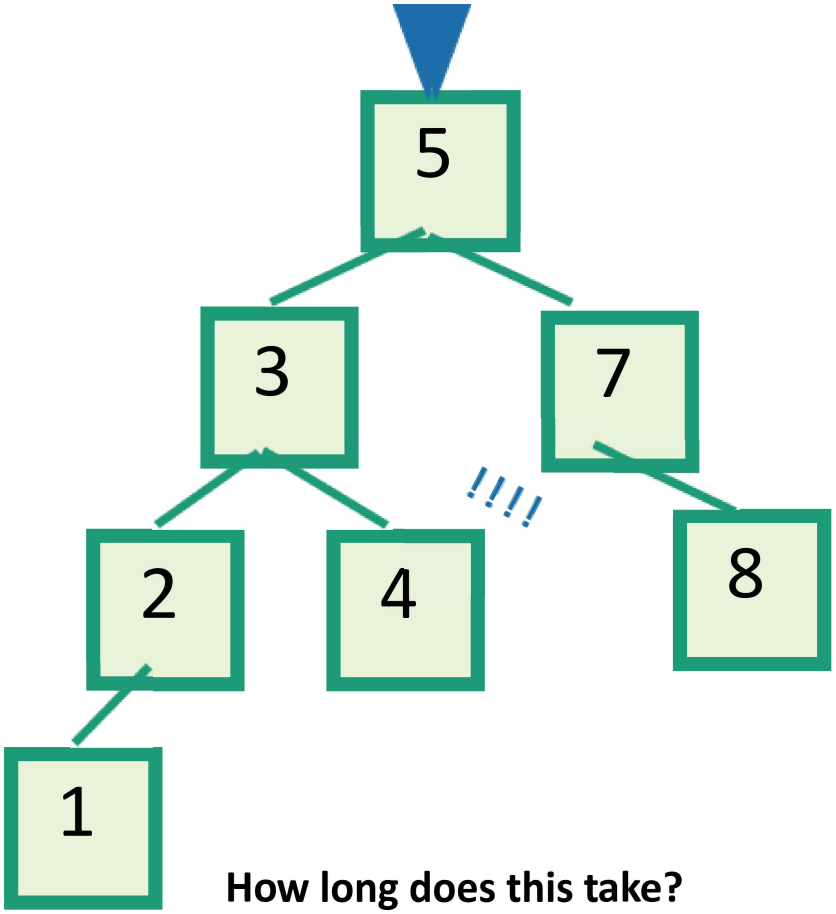
Back to the goal

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)

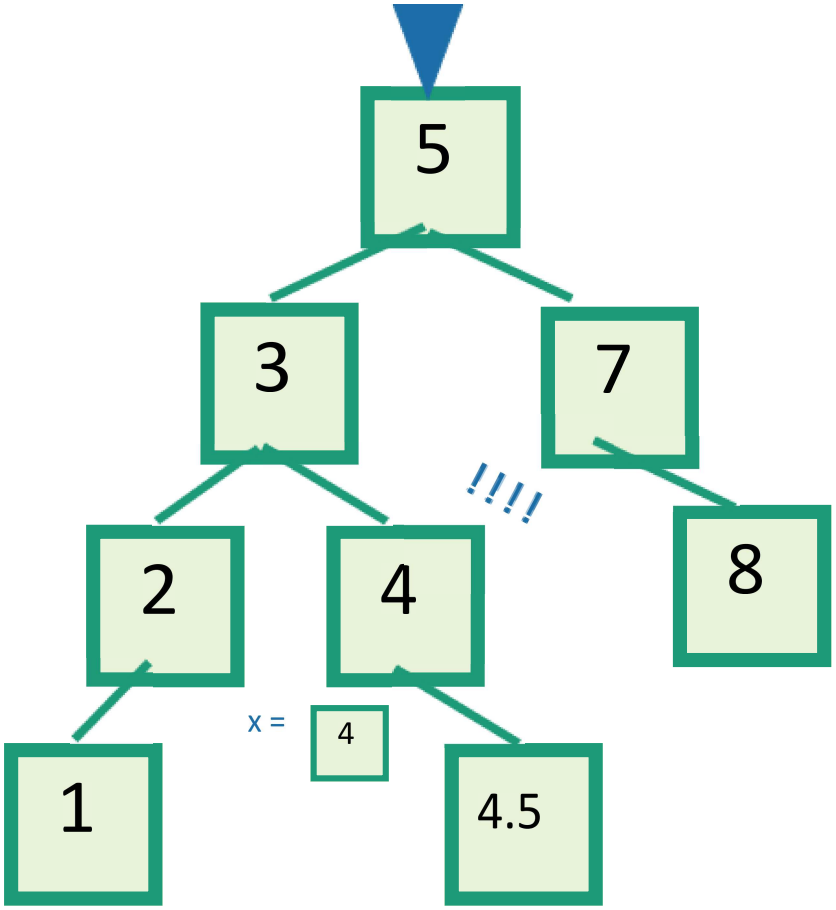
How long does this take?

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

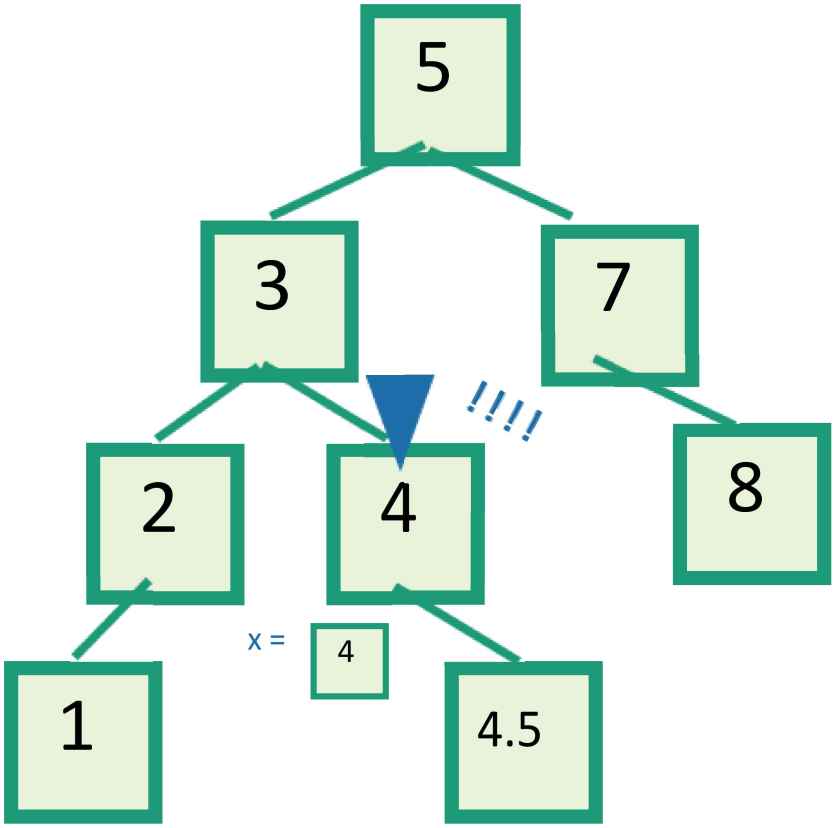


EXAMPLE: Insert 4.5

- 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.)

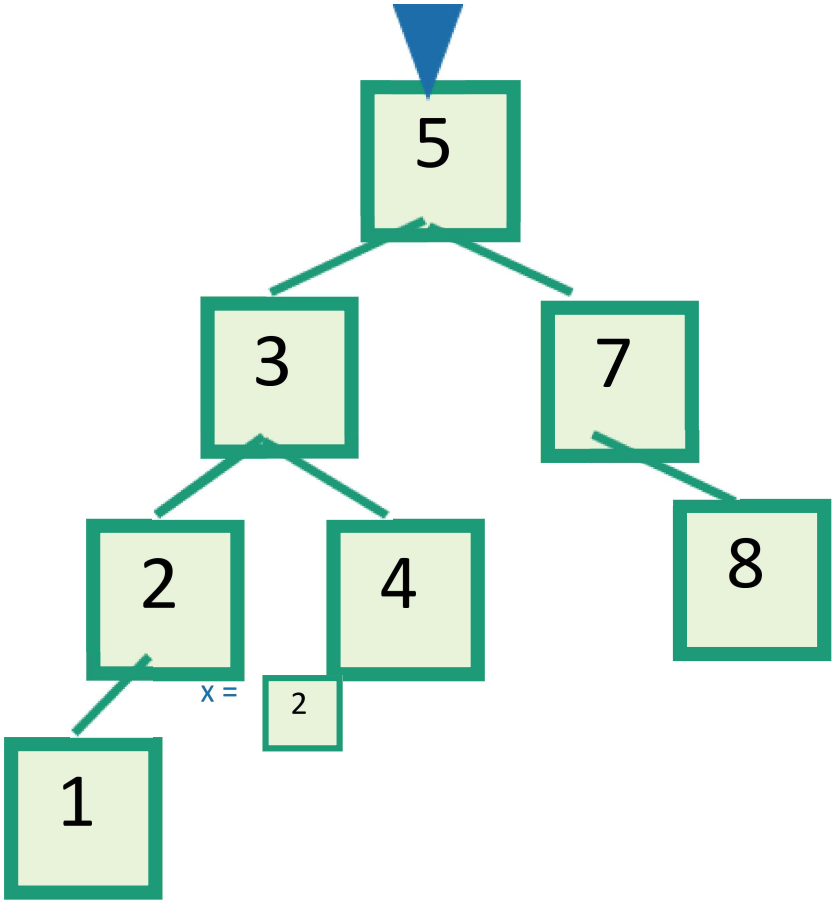
INSERT in a Binary Search Tree



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....

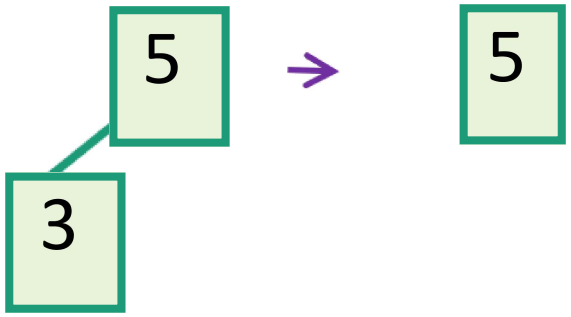
You thought about this in your pre-lecture exercise too!

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

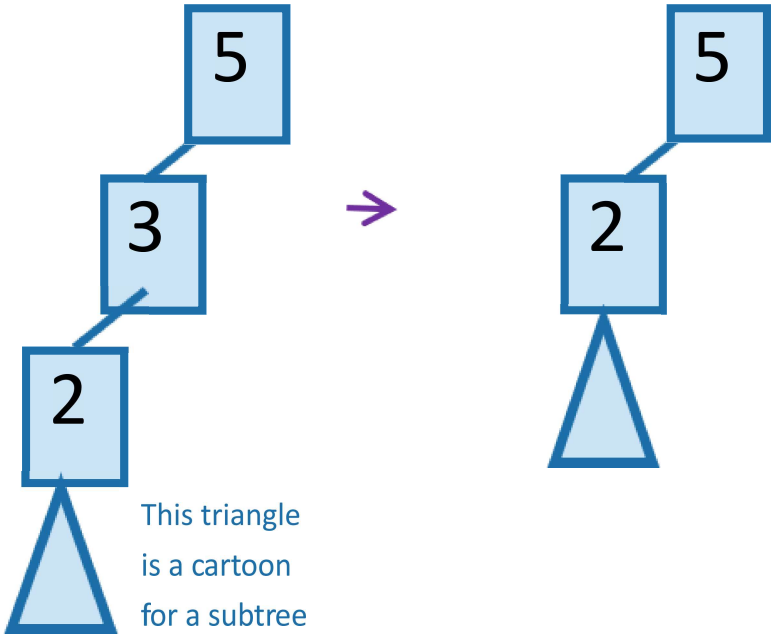


Case 1: if 3 is a leaf,

just delete it.

Write pseudocode for all of these!

Siggi the Studious Stork



This triangle is a cartoon for a subtree

Case 2: if 3 has just one child,

move that up.

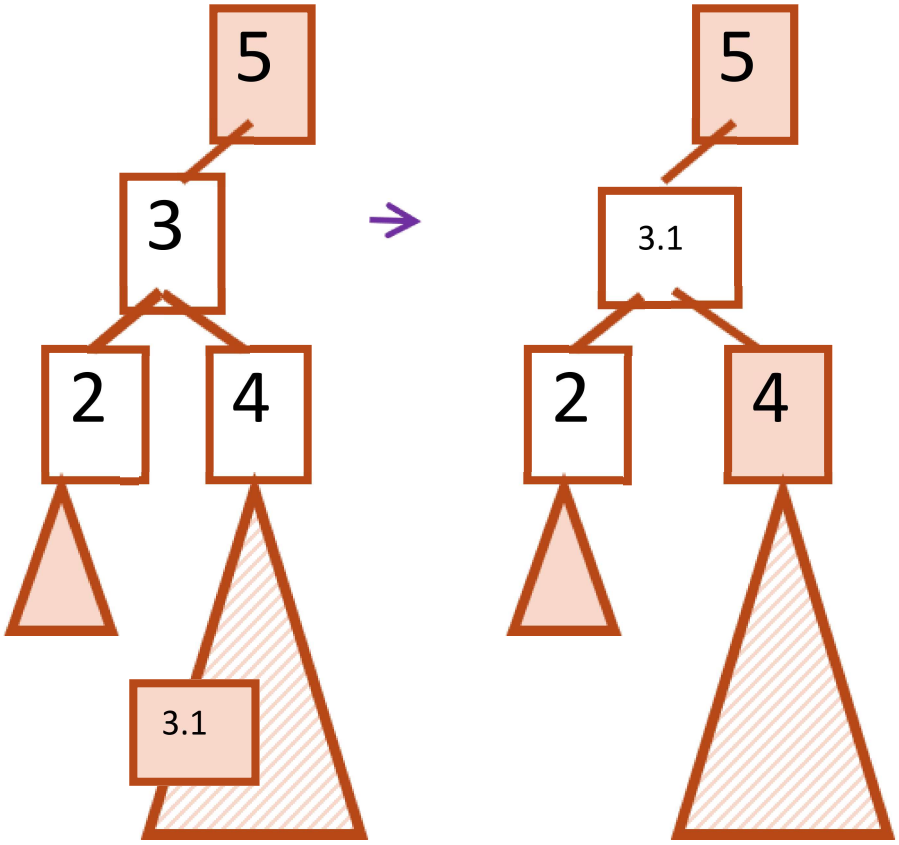
DELETE in a Binary Search Tree

ctd.

Case 3: if 3 has two children,

replace 3 with it' s immediate successor.

(aka, next biggest thing after 3)



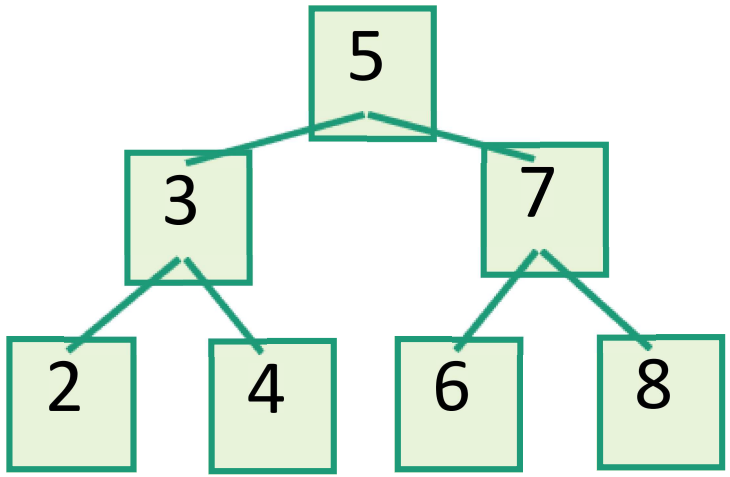
- Does this maintain the BST property?
  - Yes.
- How do we find the immediate successor?
  - SEARCH for 3 in the subtree under 3.right
- How do we remove it when we find it?
  - If [3.1] has 0 or 1 children, do one of the previous cases.
- What if [3.1] has two children?
  - It doesn' t.

想一想，练一练

- 写一下第三种情况的代码

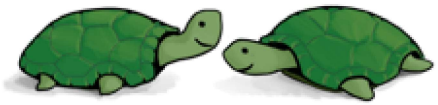
How long do these operations take?

- SEARCH is the big one.
  - Everything else just calls SEARCH and then does some small O(1)-time operation.



Time = O(height of tree)

How long does search take?



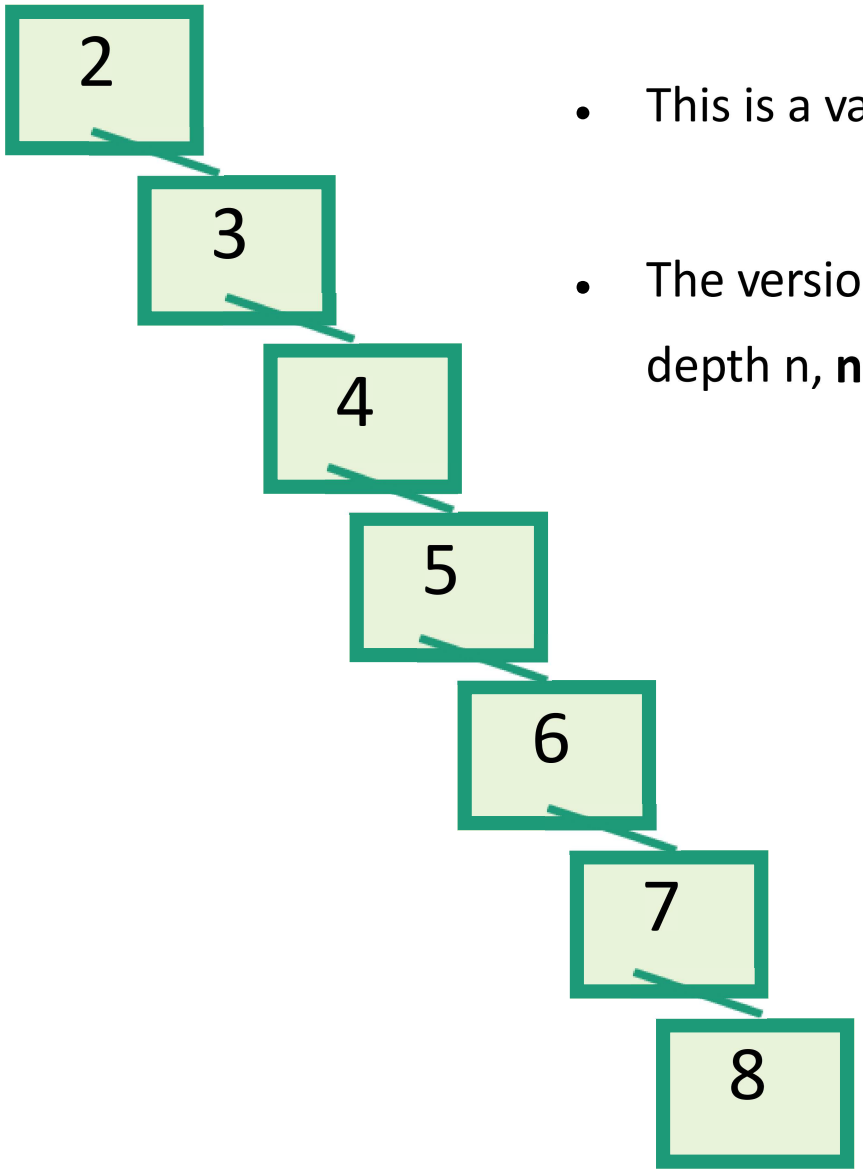
Trees have depth O(log(n)). **Done!**

Lucky the lackadaisical lemur.

Wait a second...

Plucky the Pedantic Penguin

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 Ω(n) every so often….

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

Recap

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