



香港科技大学(广州)  
THE HONG KONG  
UNIVERSITY OF SCIENCE AND  
TECHNOLOGY (GUANGZHOU)

# Design and Analysis of Algorithms

Jing Tang | DSAA 2043 Fall 2024

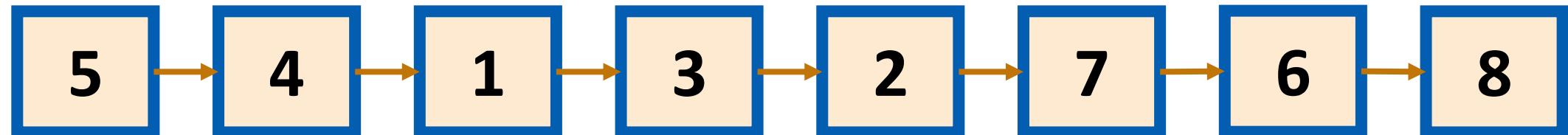
# Binary Search Tree

# Binary Search Tree

Sorted Array:



Linked list (not necessarily sorted):

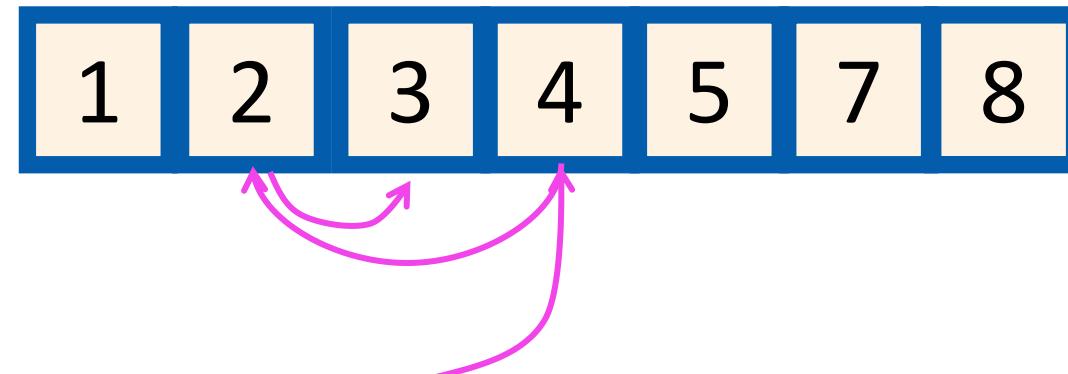


# Array

- $O(n)$  INSERT/DELETE:
  - First, find the relevant element (we'll see how below), and then move a bunch elements in the array:

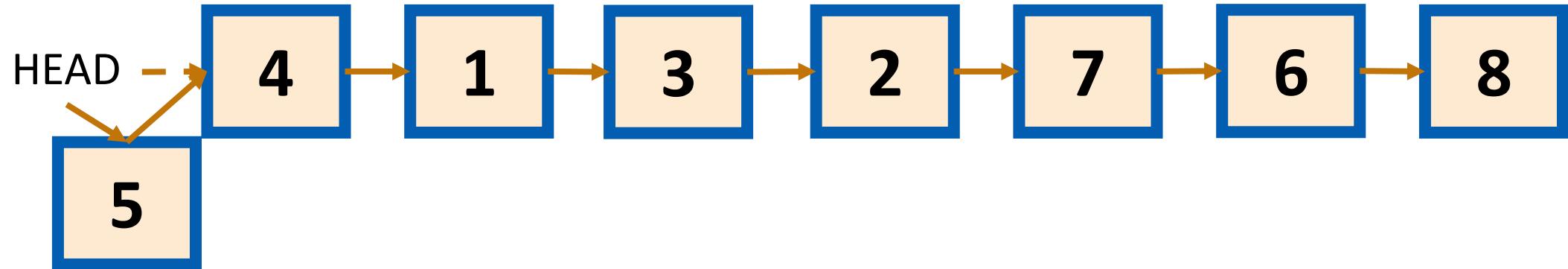


- $O(\log(n))$  SEARCH (if sorted):

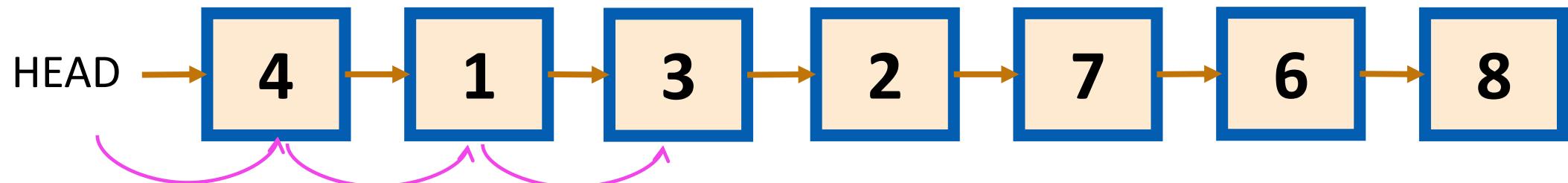


# Linked Lists

- $O(1)$  INSERT (manipulating pointers)



- $O(n)$  SEARCH/DELETE:



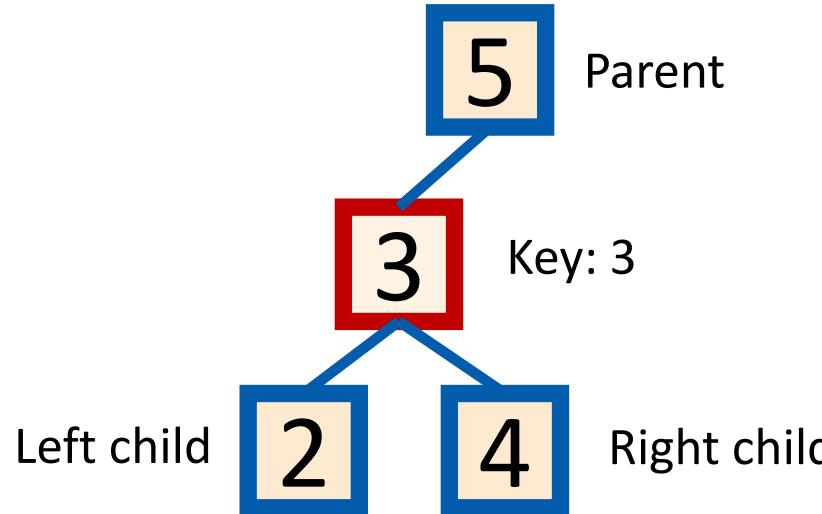
eg, search for 3 (and then you could delete it by manipulating pointers).

# Binary Search Tree

	Arrays	Linked Lists	(Balanced) Binary Search Trees
Search	$O(n)$ $(O(\log n) \text{ if sorted})$	$O(n)$	$O(\log n)$
Delete	$O(n)$	$O(n)$	$O(\log n)$
Insert	$O(n)$	$O(1)$	$O(\log n)$

# Binary Tree Terminology

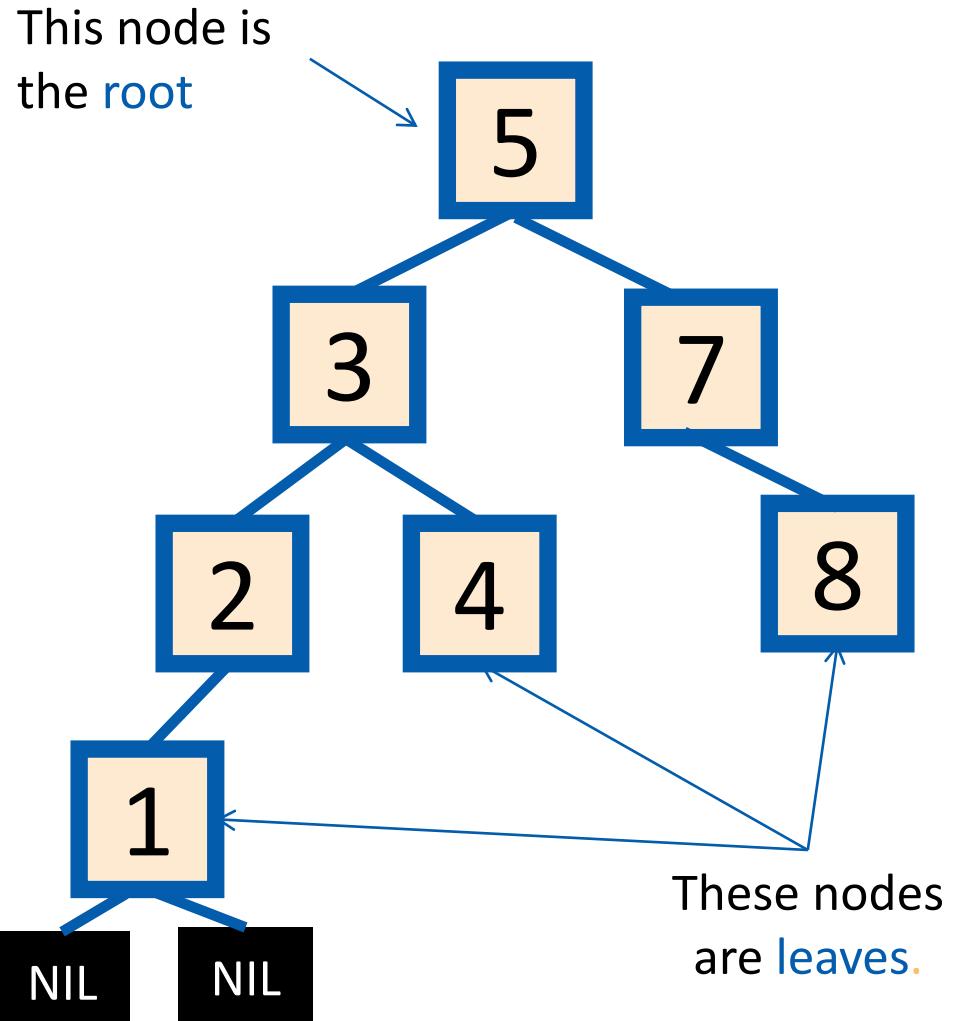
Each node has two **children**



**2** is a **descendant** of **5**

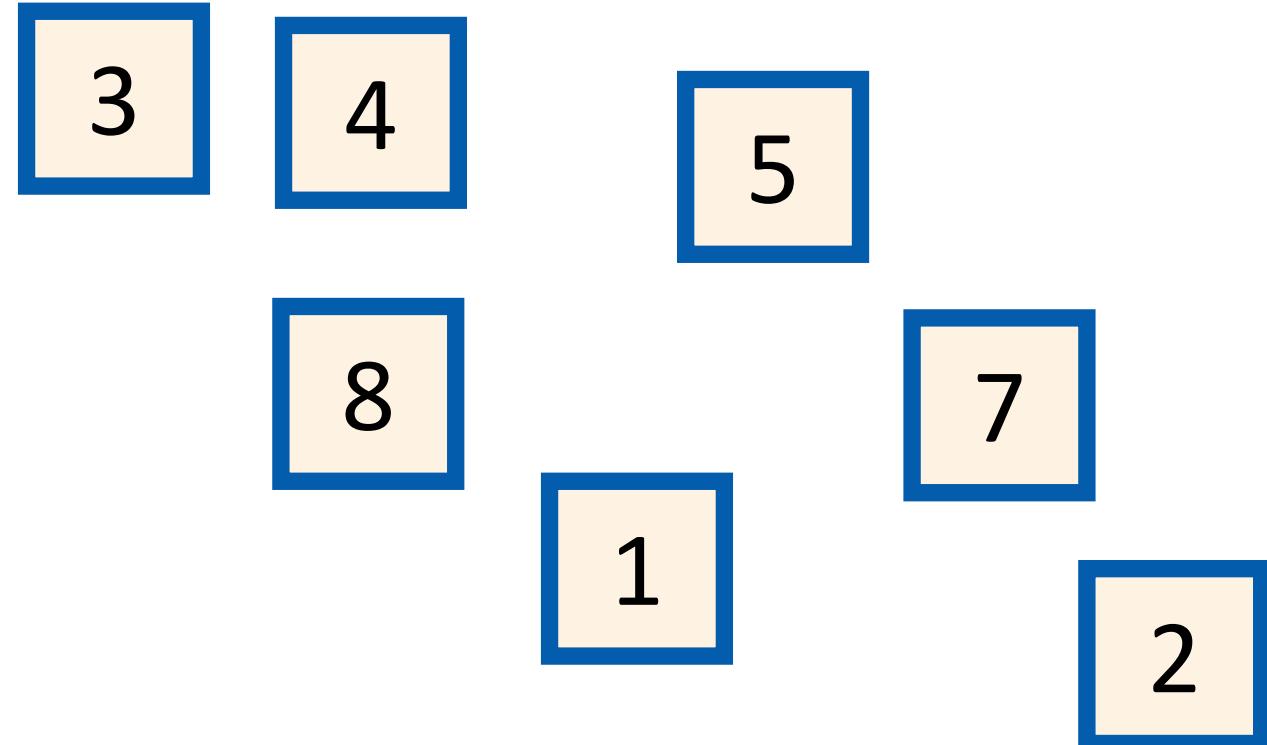
Both **children** of **1** are **NIL**.  
(I won't usually draw them).

The **height** of this tree is 3



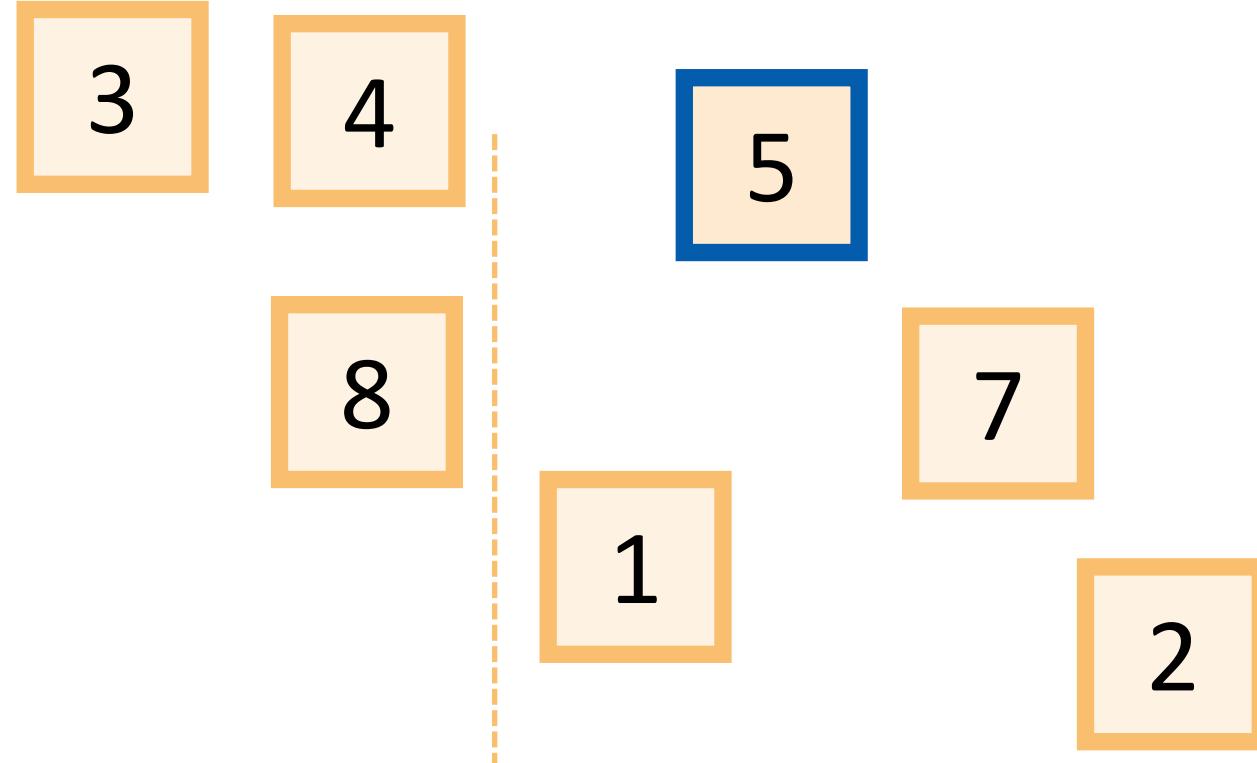
# Binary Search Tree

- 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.



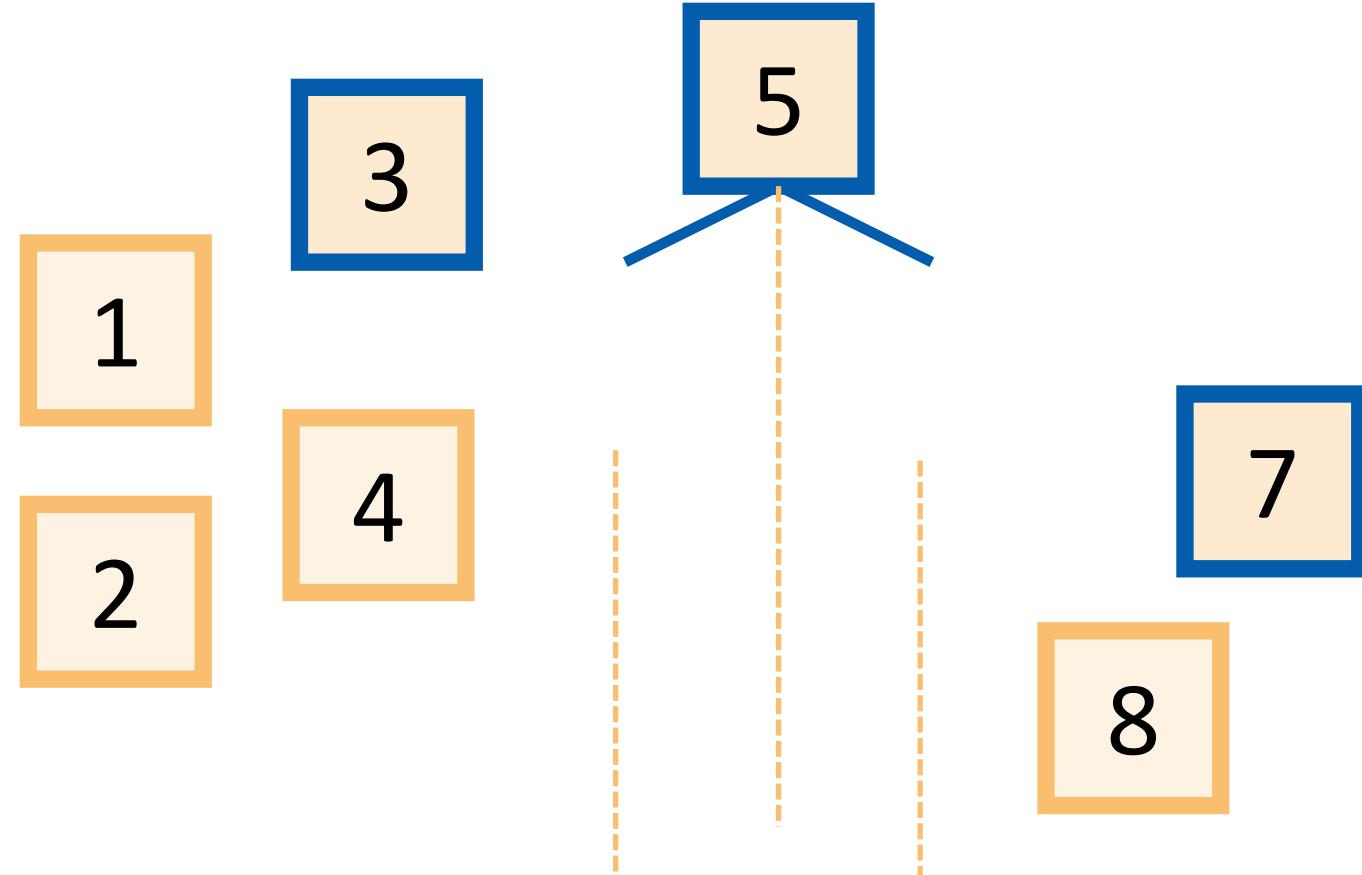
# Binary Search Tree

- 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.



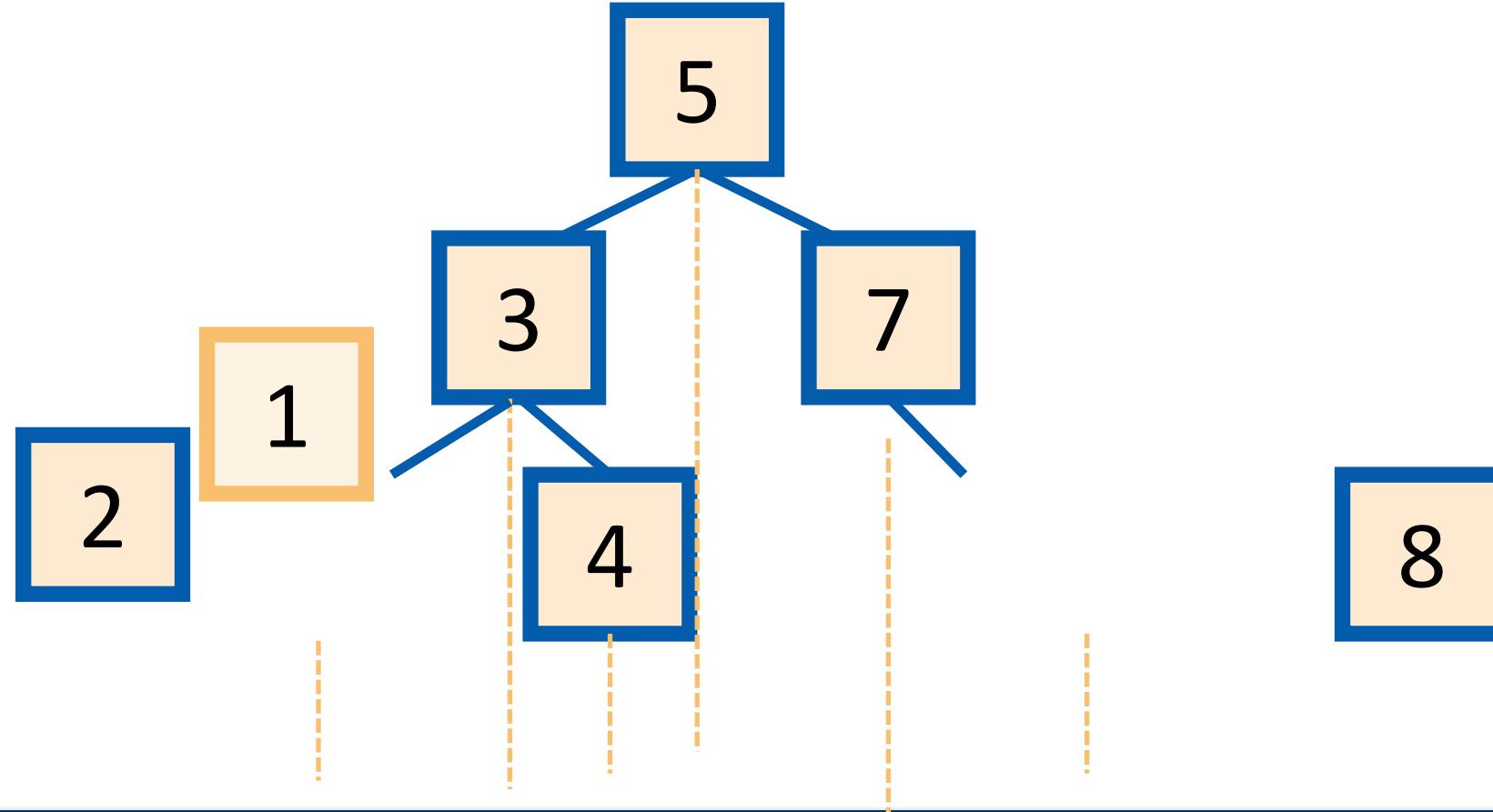
# Binary Search Tree

- 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.



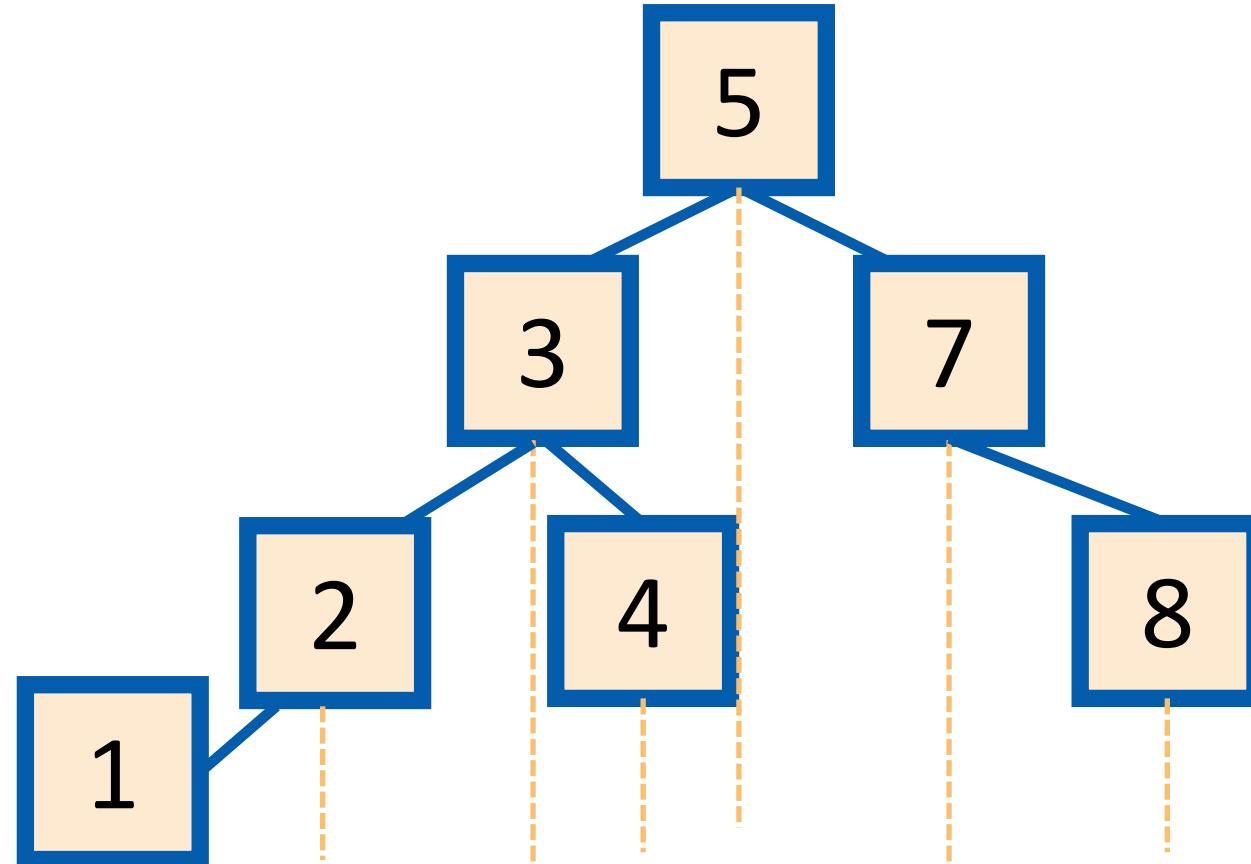
# Binary Search Tree

- 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

- 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.



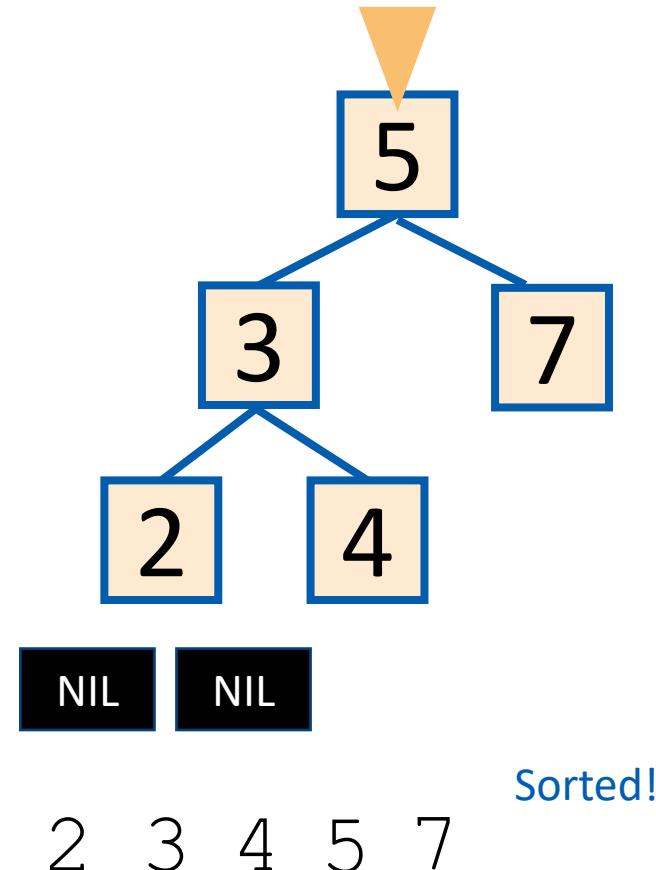
Q: Is this the only  
binary search tree I  
could possibly build  
with these values?

# Traversal

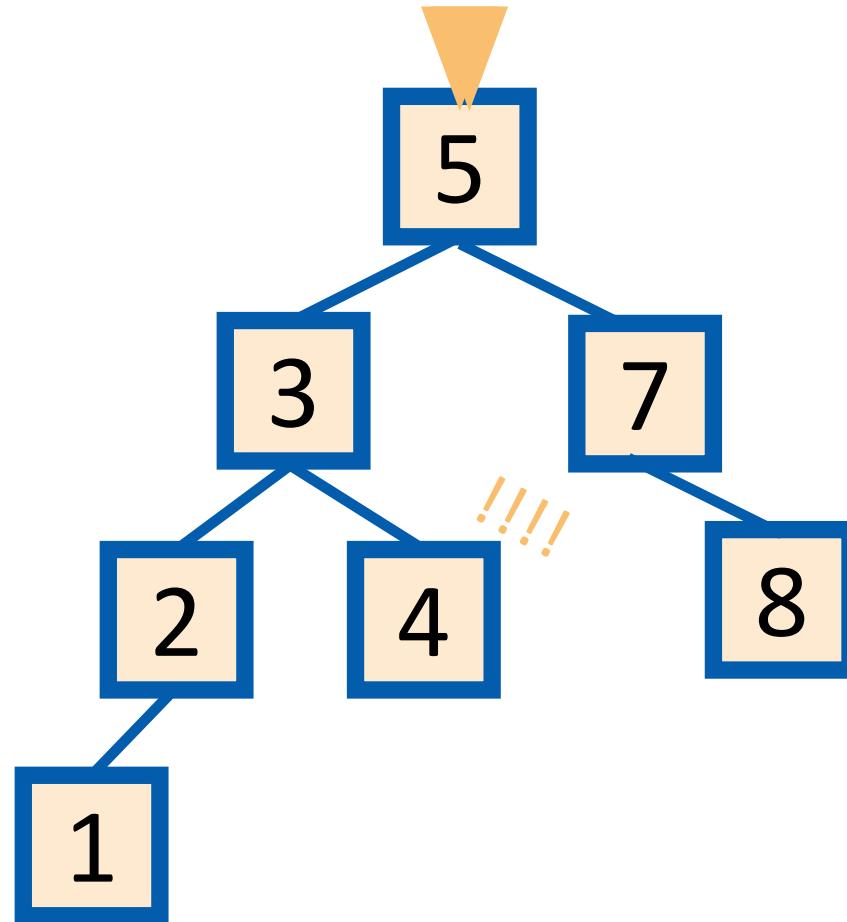
- Output all the elements in sorted order!

- `inOrderTraversal(x):`
  - if  $x \neq \text{NIL}$ :
    - `inOrderTraversal( x.left )`
    - `print( x.key )`
    - `inOrderTraversal( x.right )`

Pre-order / post-order traversal?



# Search

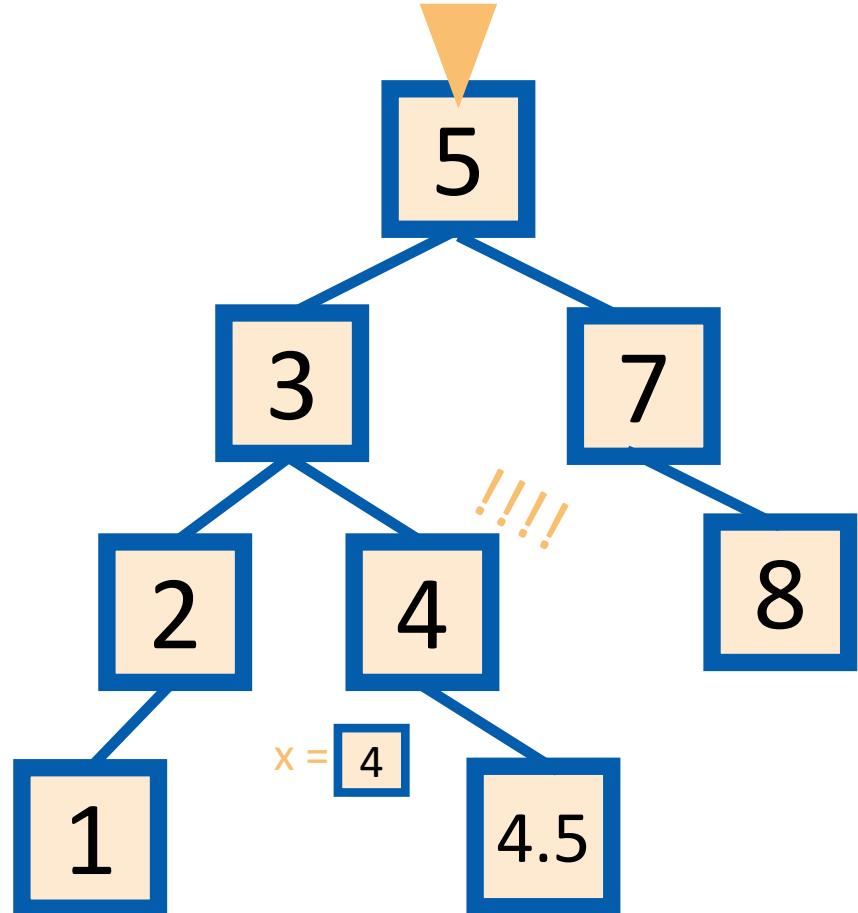


**EXAMPLE:** Search for 4.

**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)

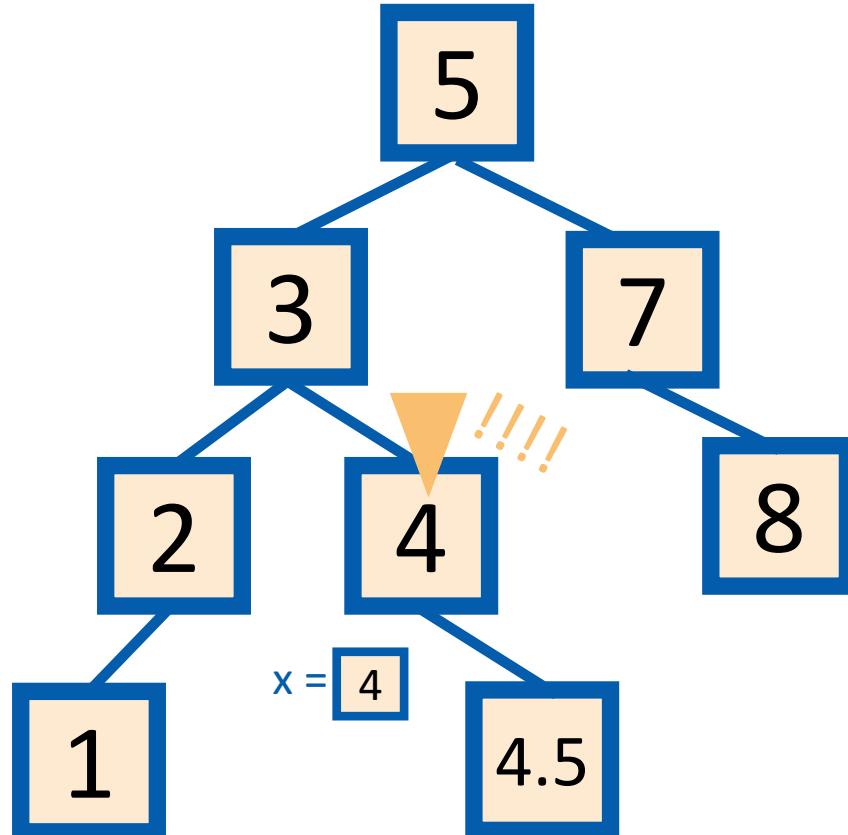
# Insert



## EXAMPLE: Insert 4.5

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

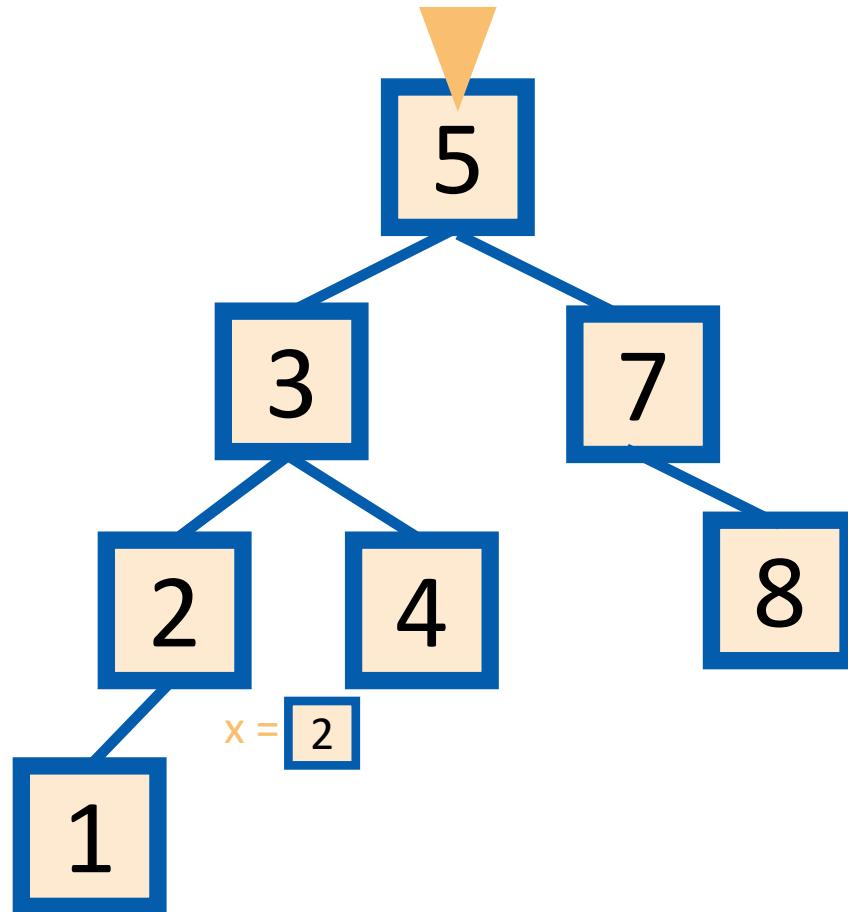
# Insert



## 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

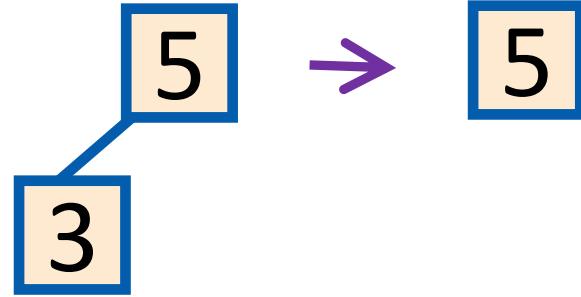


## EXAMPLE: Delete 2

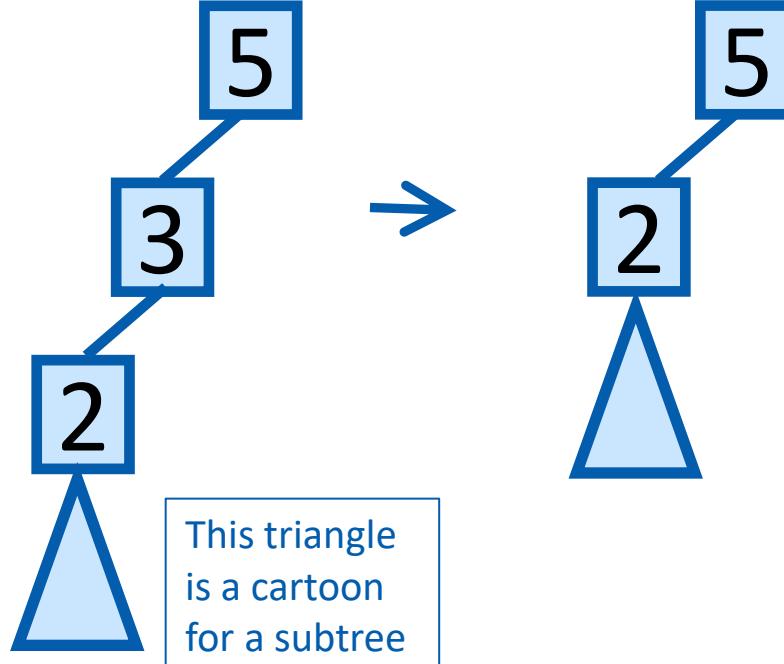
- **DELETE(key):**
  - $x = \text{SEARCH}(\text{key})$
  - **if**  $x.\text{key} == \text{key}:$ 
    - ....delete  $x$ ....

This is a bit more complicated...

# Delete



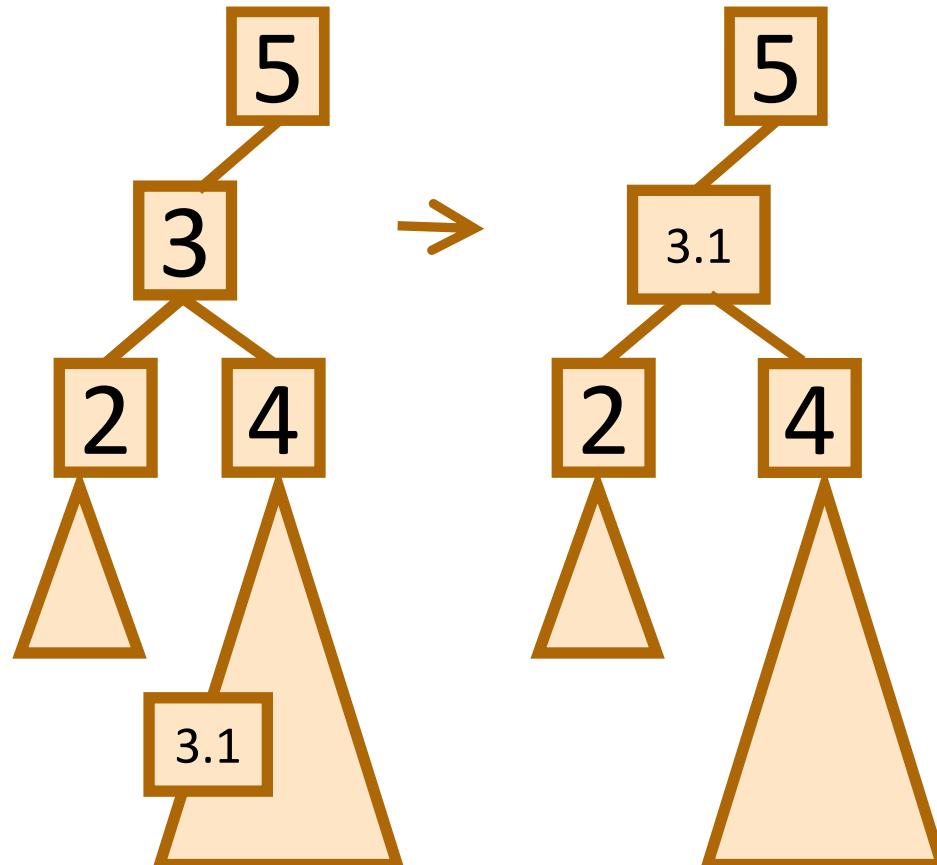
**Case 1:** if 3 is a leaf,  
just delete it.



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

# Delete

**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.

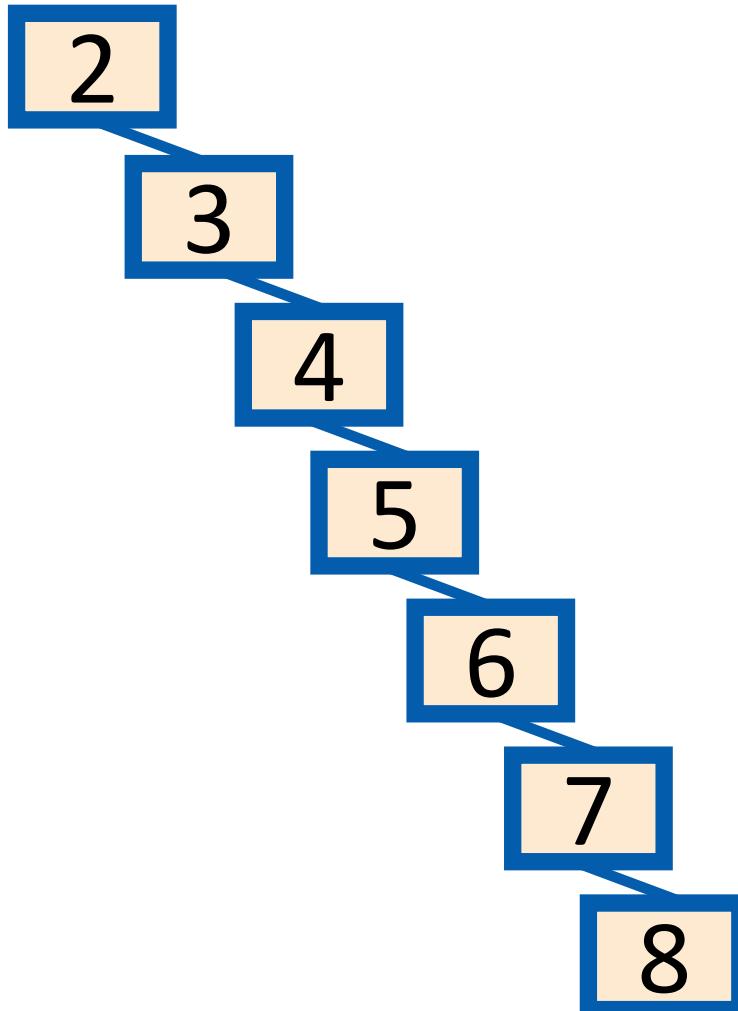
# More Operations

- `findmin(x)`: finds the minimum of the tree rooted at  $x$
- `findmax(x)`: finds the max of the tree rooted at  $x$
- `deletemin()`: finds the minimum of the tree and delete it

Time complexities of them?

# The Importance of Being Balanced

- This is a valid binary search tree.
- The version with  $n$  nodes has depth  $n$ , **not**  $\Theta(\log(n))$ .



# Balanced BST strategy

- Augment every node with some property
- Define a local invariant on property
- Show (prove) that invariant guarantees  $\Theta(\log n)$  height
- Design algorithms to maintain property and the invariant

# AVL Trees

# AVL Trees

An AVL (Adelson-Velskii and Landis) tree is a binary search tree that also meets the following rule

**AVL condition:** For every node, the height of its left subtree and right subtree differ by at most 1.

Height of a tree:

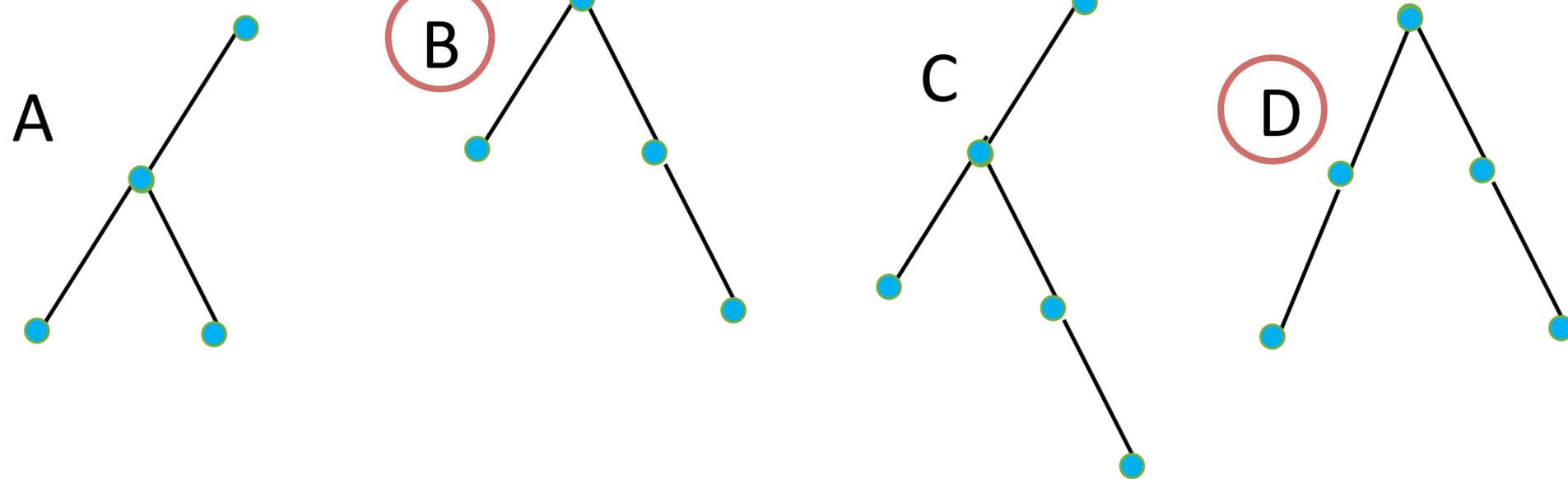
Maximum number of edges on a path from the root to a leaf.

A tree with one node has height 0.

A null tree (no nodes) has height -1.

# AVL Trees

Which one(s) is balanced according to AVL's definition?



# AVL Trees

An AVL tree is a binary search tree that also meets the following rule

**AVL condition:** For every node, the height of its left subtree and right subtree differ by at most 1.

This will avoid the  $\Theta(n)$  behavior! We have to check:

1. We must be able to maintain this property when inserting/deleting.
2. Such a tree must have height  $\Theta(\log n)$ .

# Bounding the Height

- Let  $n(h)$  be the minimum number of nodes in an AVL tree of height  $h$ .
- If we can say  $n(h)$  is big, we'll be able to say that a tree with  $n$  nodes has a small height.
- So...what's  $n(h)$ ?

$$n(h) = \begin{cases} 1, & \text{if } h = 0 \\ 2, & \text{if } h = 1 \\ n(h - 1) + n(h - 2) + 1, & \text{otherwise} \end{cases}$$

# Bounding the Height

- Hey! That's a recurrence!
- Recurrences can describe any kind of function, not just running time of code!

$$\bullet n(h) = \begin{cases} 1, & \text{if } h = 0 \\ 2, & \text{if } h = 1 \\ n(h - 1) + n(h - 2) + 1, & \text{otherwise} \end{cases}$$

- We could use tree method, but it's a little...weird.
- It'll be easier if we change things just a bit:

$$\bullet n(h) \geq \begin{cases} 1, & \text{if } h = 0 \\ 2, & \text{if } h = 1 \\ n(h - 2) + n(h - 2) + 1, & \text{otherwise} \end{cases}$$

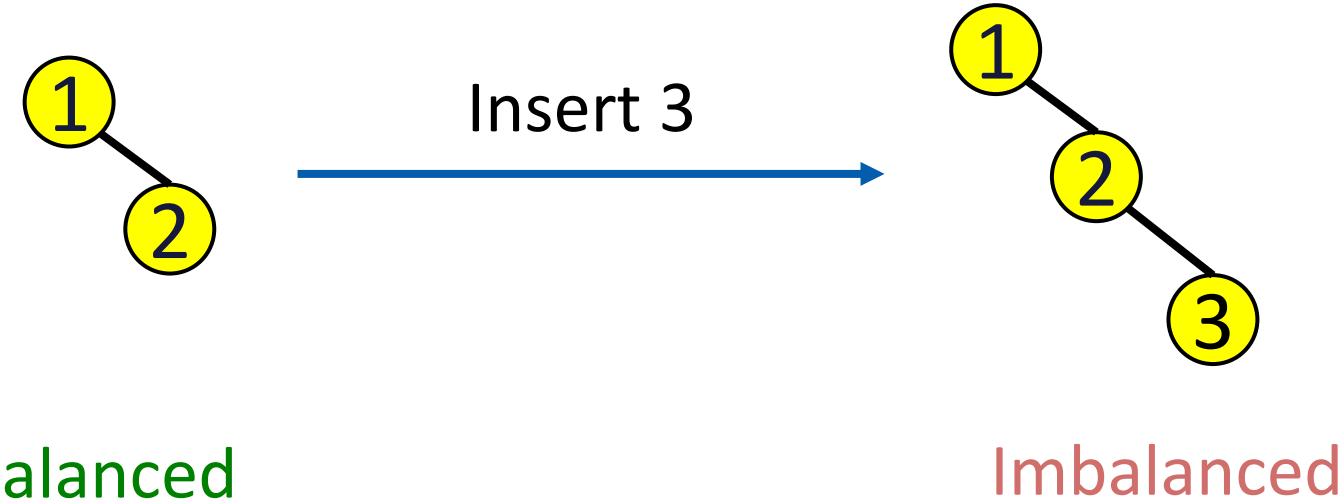
# Bounding the Height

$$\begin{aligned}n(h) &= n(h-1) + n(h-2) + 1 \\&> 2n(h-2) \\&> 2 \times 2n(h-4) \\&> 2^{\frac{h}{2}} \\h &< 2\log n(h)\end{aligned}$$

Hence,  $h = \Theta(\log n)$ .

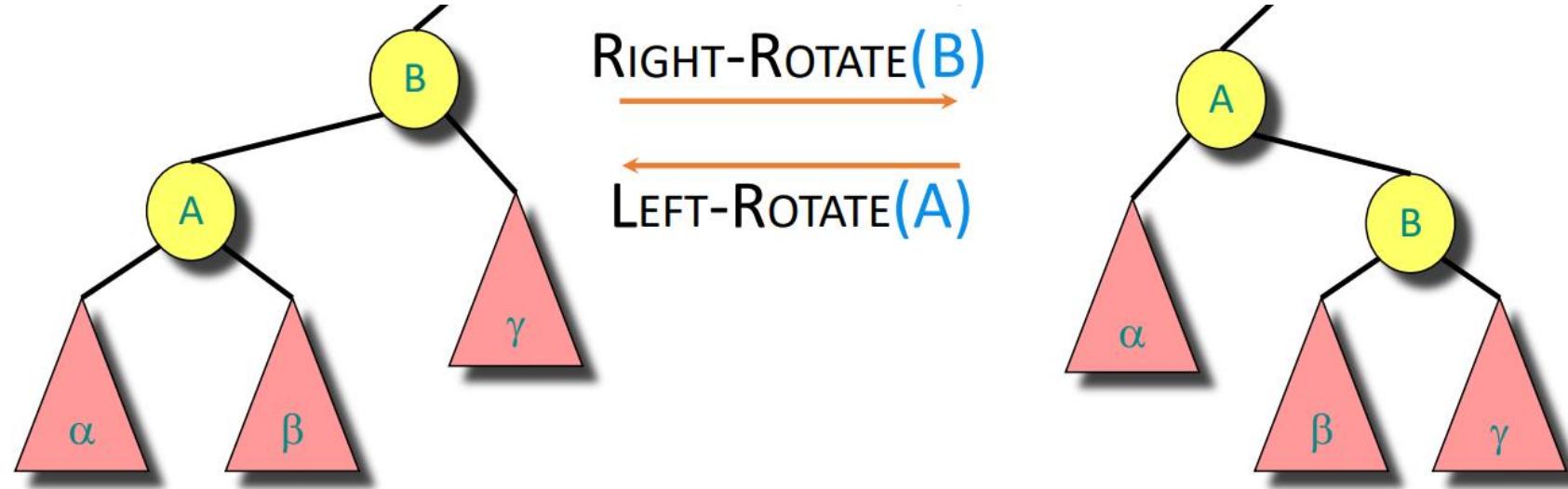
# Insertion

What happens if when we the AVL condition is violated after insertion?

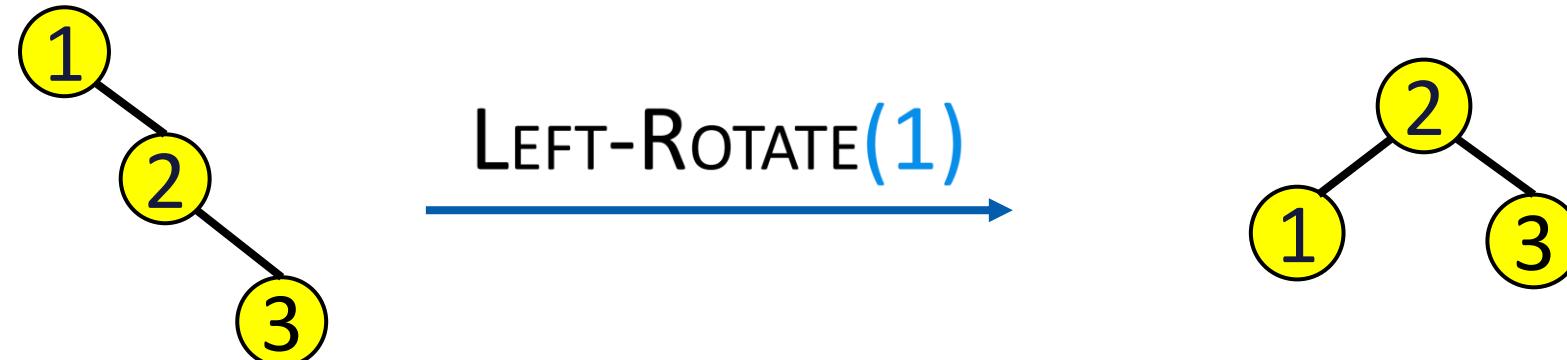


# Insertion

Rotations!

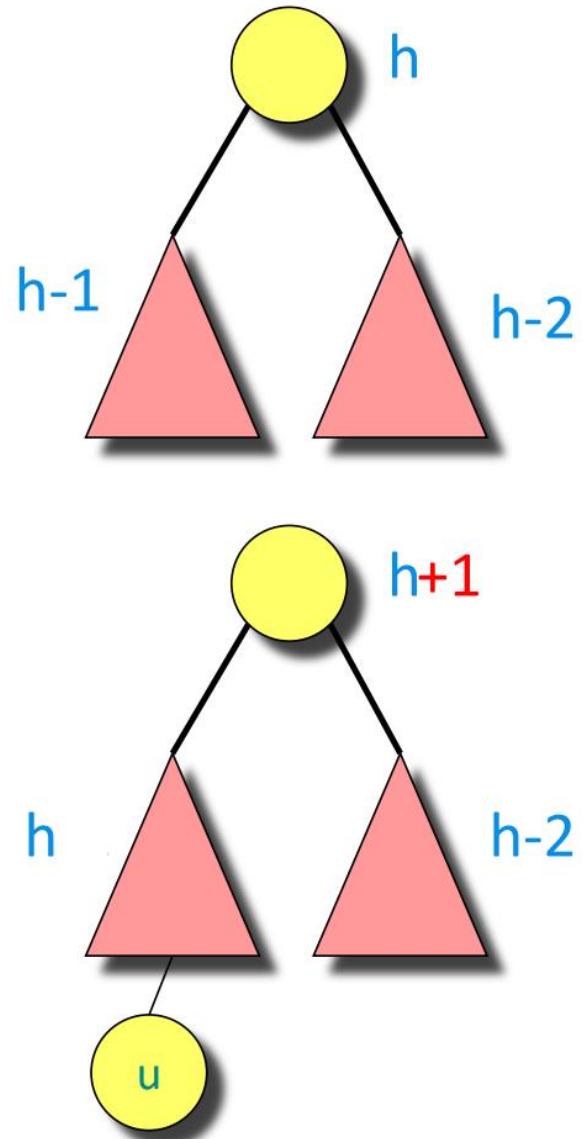


Rotations can reduce the height!



# Insertion / Deletion

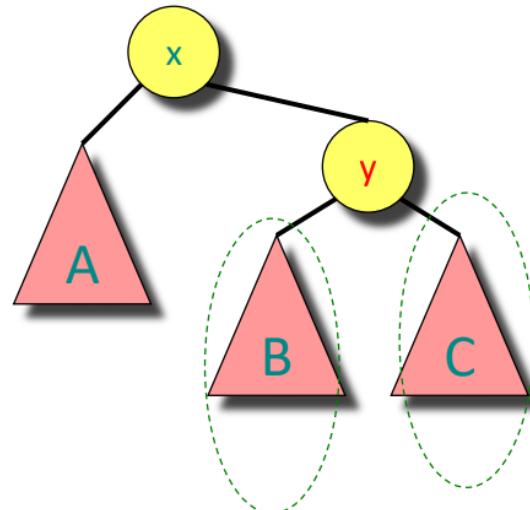
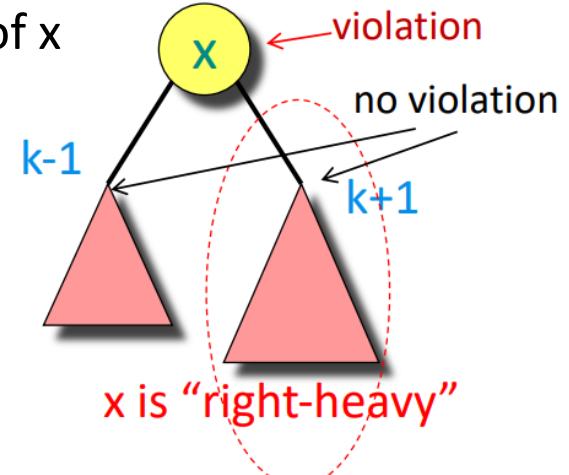
- Insert new node  $u$  as in the simple BST
  - Can create imbalance
- Work your way up the tree, restoring the balance
- Similar issue/solution when deleting a node



# Balancing

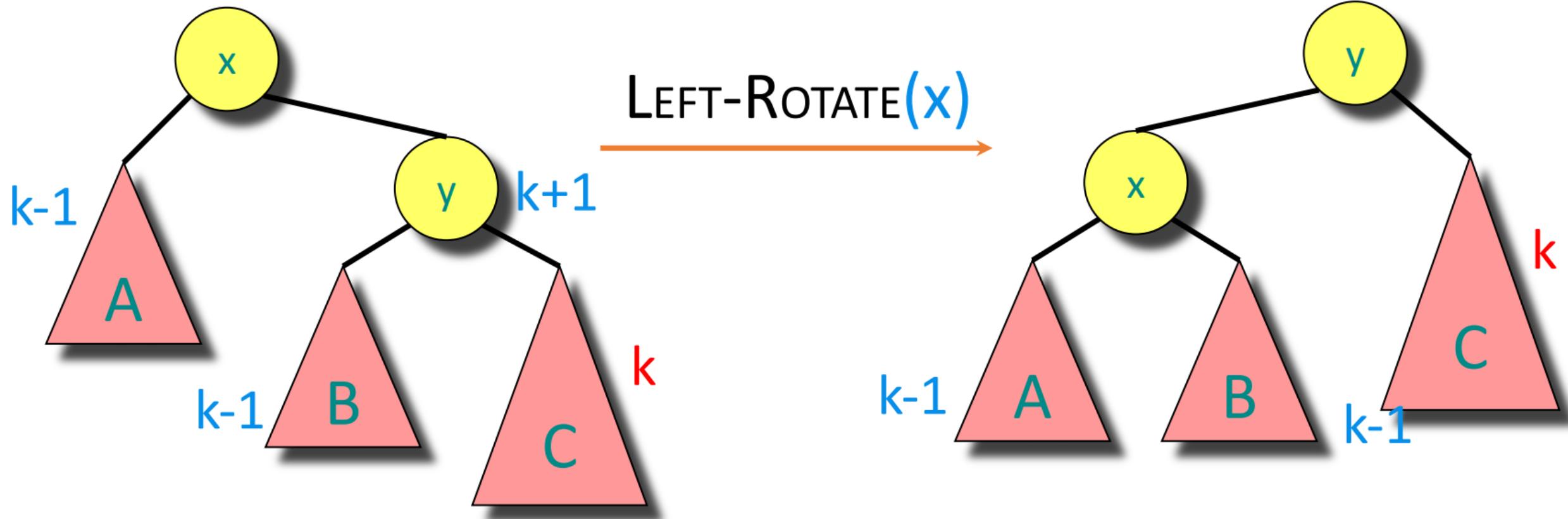
- Let  $x$  be the lowest “violating” node
  - we will try to correct that and move up the tree
- Assume that  $x$  is “right-heavy”
  - we analyze more the right subtree of  $x$
  - $y$  is the right child of  $x$
- Scenarios
  - Case 1:  $y$  is right-heavy / balanced
  - Case 2:  $y$  is left-heavy

The right child of  $x$  has +2 height than the left child of  $x$



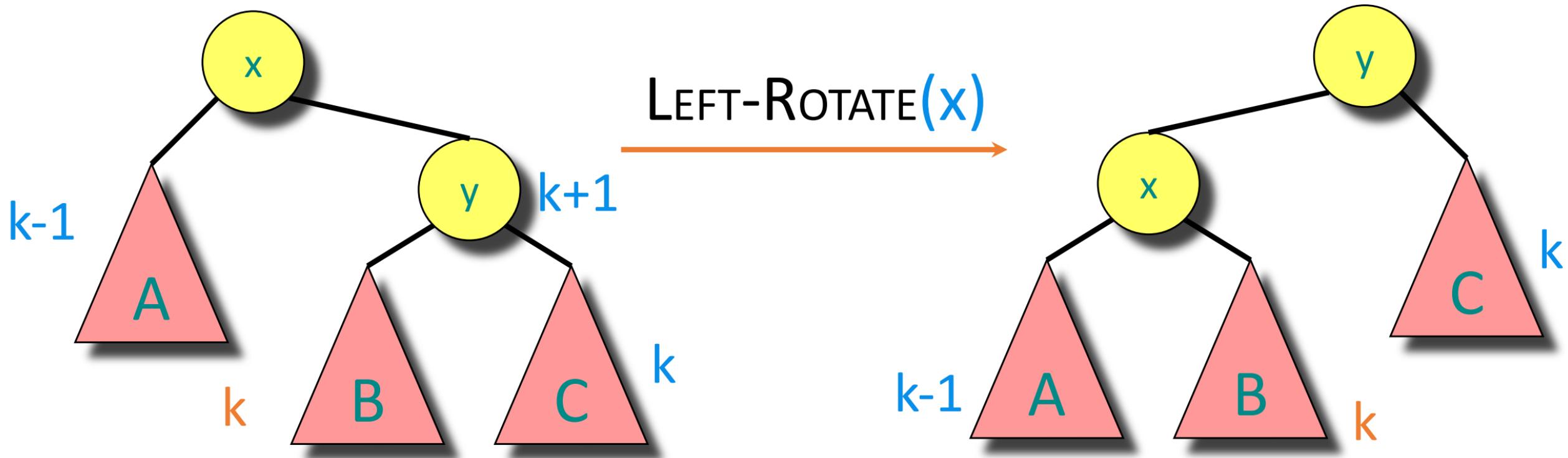
# Balancing

Case 1.1:  $y$  is right-heavy



# Balancing

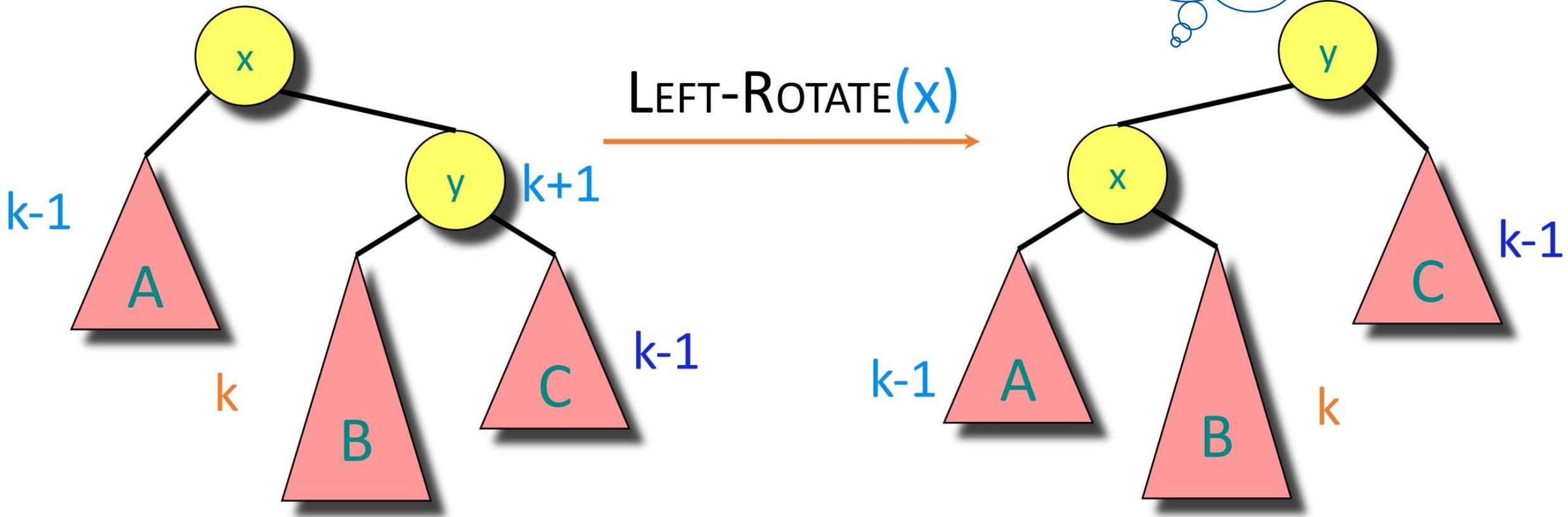
Case 1.2:  $y$  is balanced



Same as Case 1.1

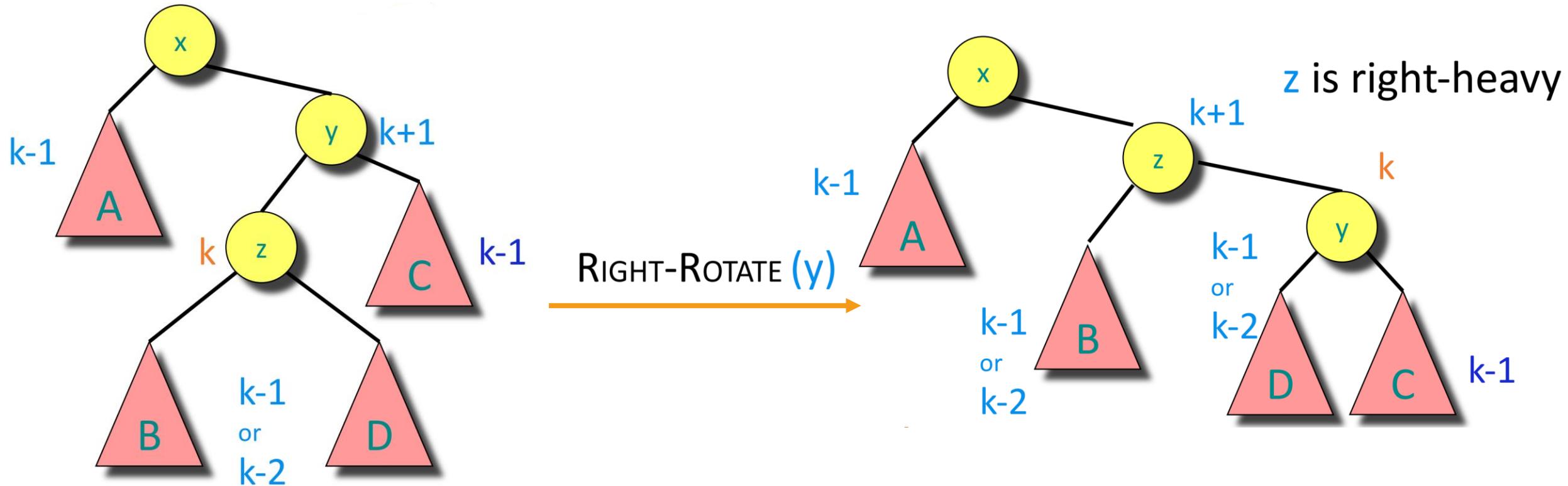
# Balancing

Case 2: y is left-heavy



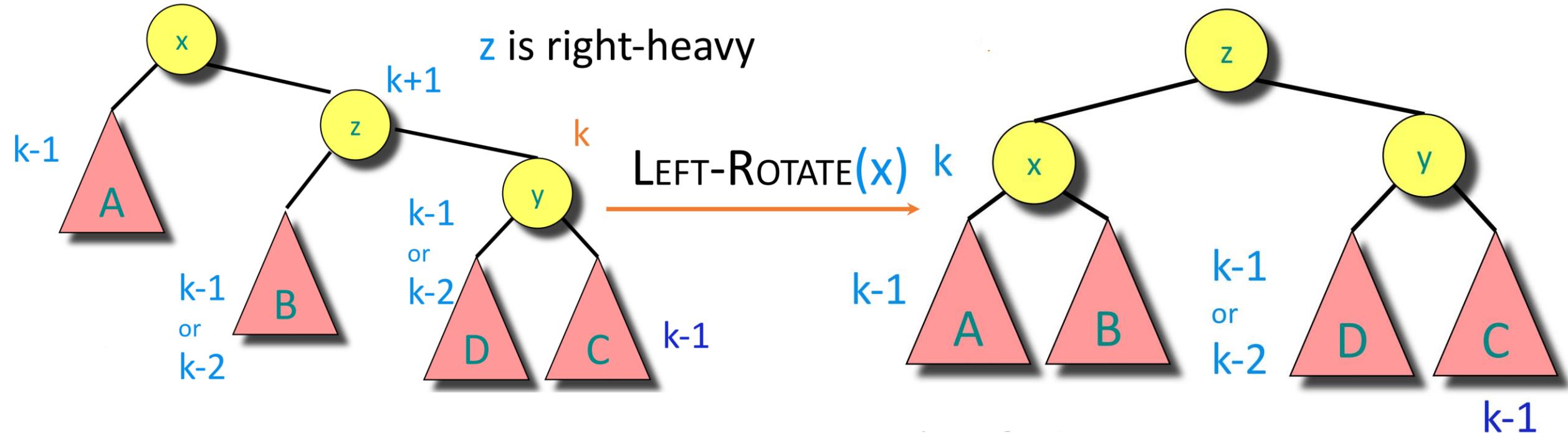
# Balancing

Case 2: y is left-heavy



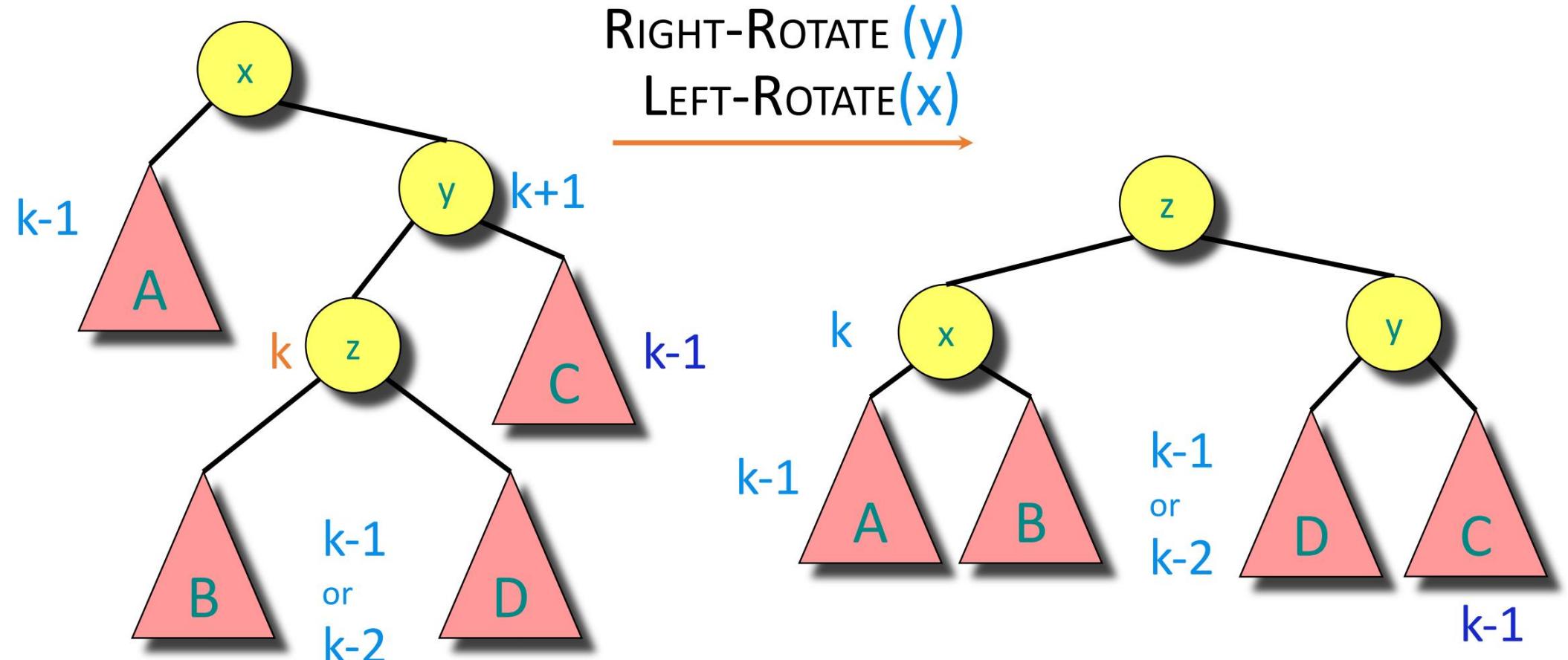
# Balancing

Case 2: y is left-heavy



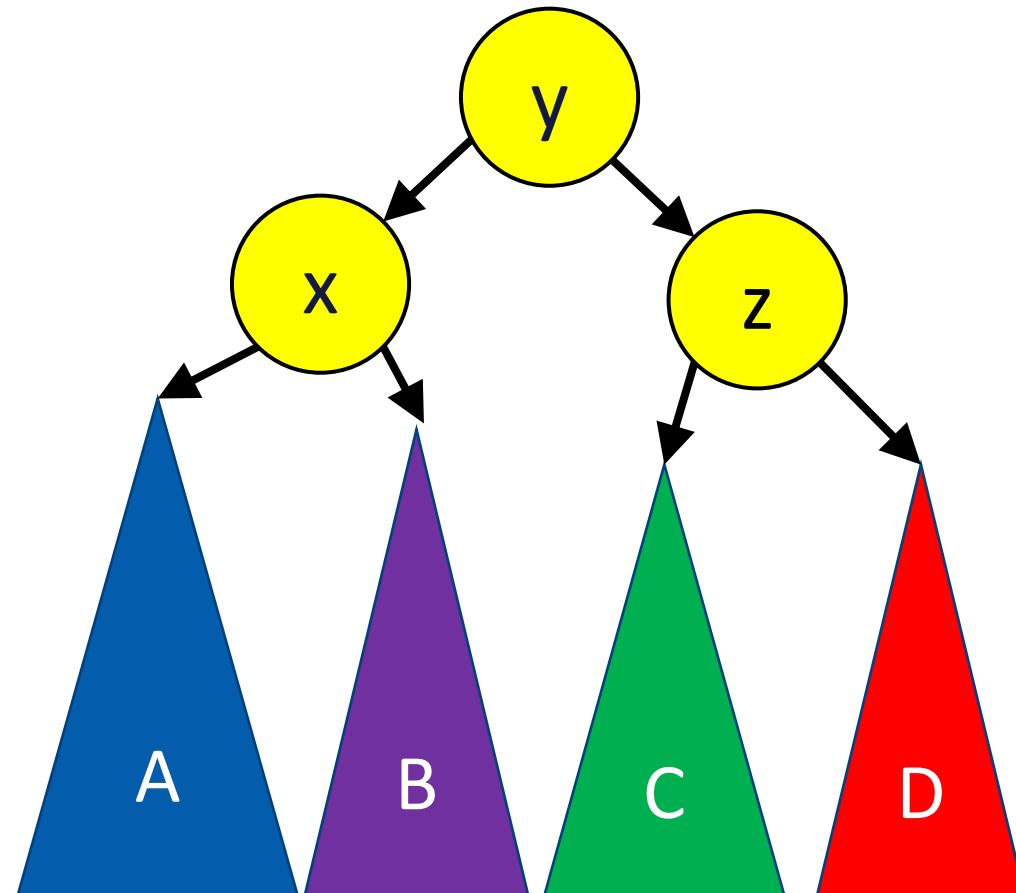
# Balancing

Case 2: y is left-heavy



# Four Types of Rotations

To summarize



Insert location	Solution
Left subtree of left child (A)	Single right rotation
Right subtree of left child (B)	Double (left-right) rotation
Left subtree of right child (C)	Double (right-left) rotation
Right subtree of right child(D)	Single left rotation

# Other Self-Balancing Trees

- “Red-black trees” work on a similar principle to AVL trees.
- “Splay trees”: Get  $O(\log n)$  amortized bounds for all operations.
- “Scapegoat trees”: worst case  $O(\log n)$  search complexity. Others are same as splay trees.
- “Treaps” – a BST and heap in one (!)

Similar tradeoffs to AVL trees.

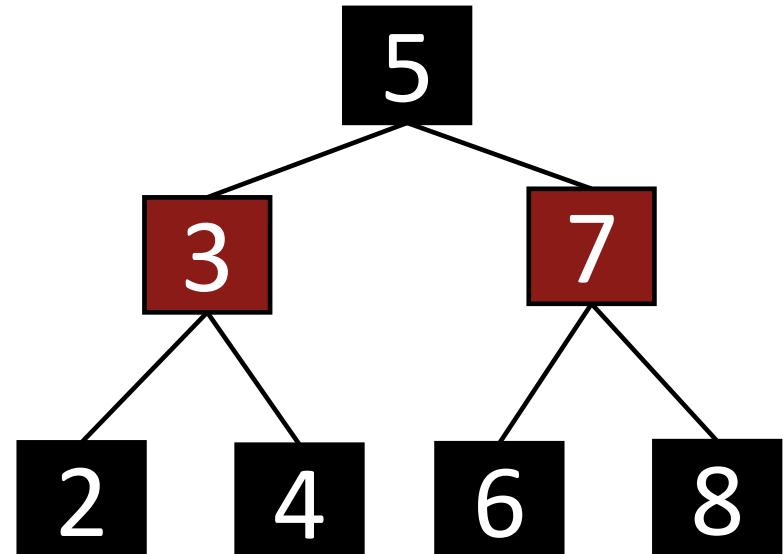
# Red-Black Trees

# Red-Black Trees

- AVL trees requires more rotations during insertion/deletion due to relatively strict balancing.
- What if we relax the constraint a bit and use some [proxy](#) of balancing?

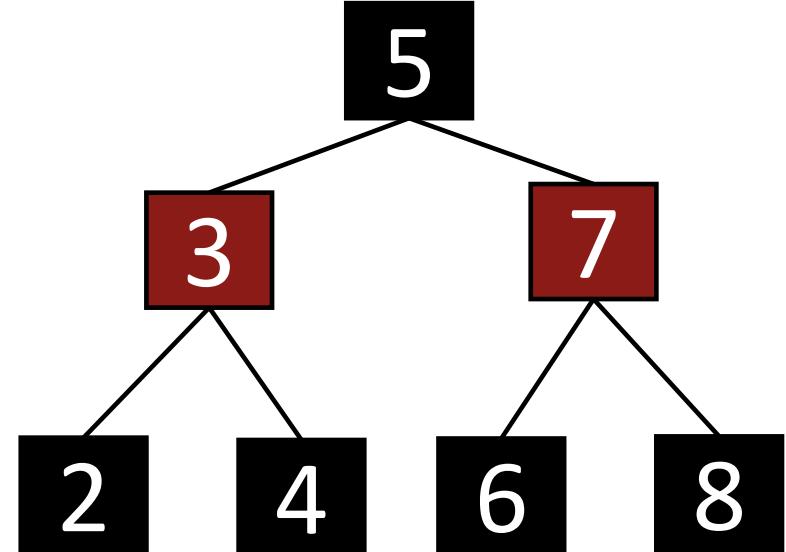
*Red-Black tree!*

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



# Red-Black Trees

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

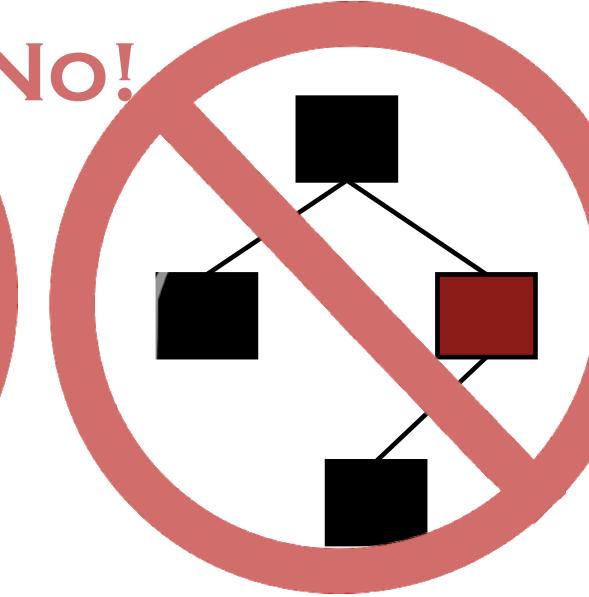
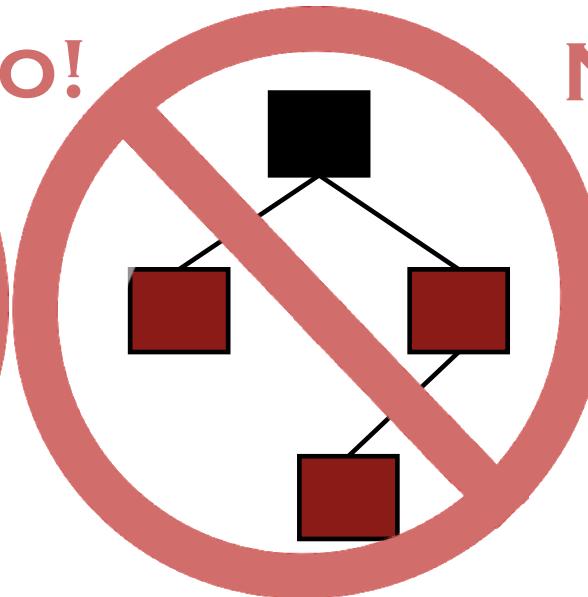
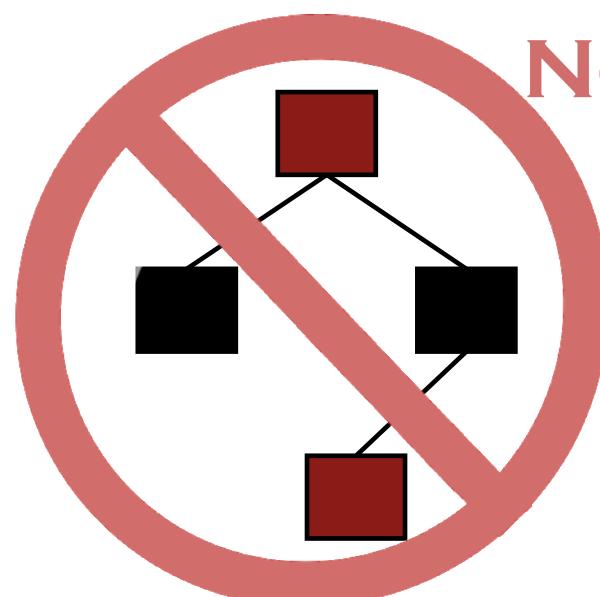


# Red-Black Trees

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

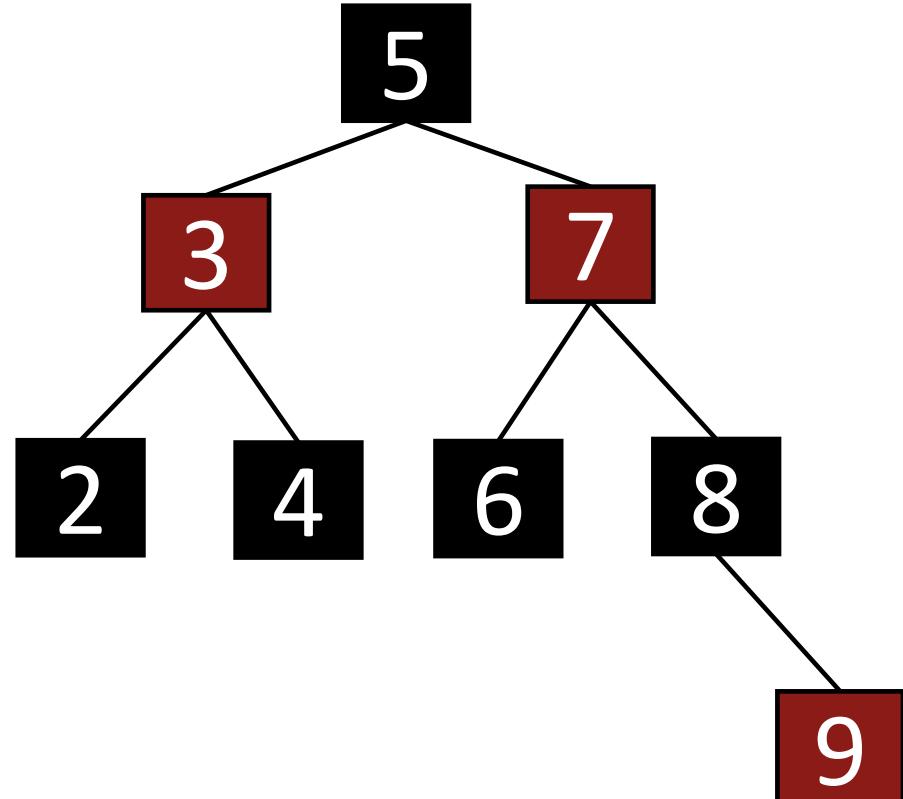
Which of these  
are red-black trees?  
(NIL nodes not drawn)

1 minute think  
1 minute share



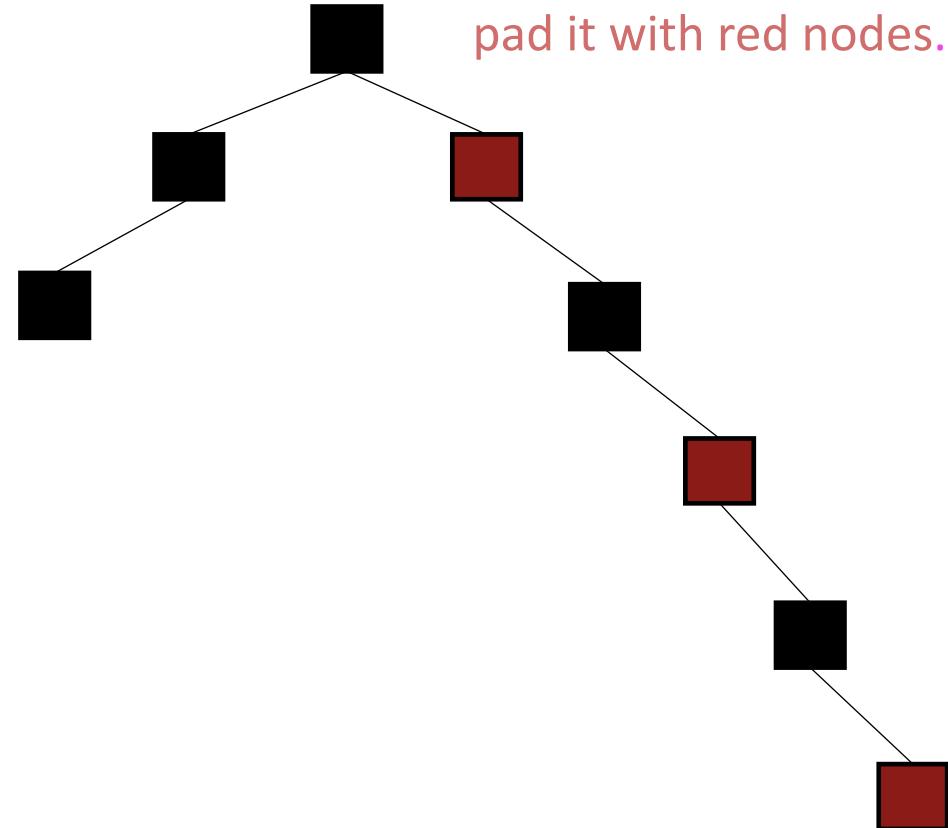
# Why These Rules?

- This is pretty balanced.
  - The **black nodes** are balanced
  - The **red nodes** don't mess things up too much.
- We can maintain this property as we insert/delete nodes, by using rotations or color flipping.



# Why These Rules?

- This is “pretty balanced”.
- Conjecture:
  - the height of a **red-black** tree with  $n$  nodes is at most  $2 \log(n)$



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

# Why These Rules?

The height of a RB-tree with  $n$  non-NIL nodes is at most  $2\log_2(n + 1)$ .

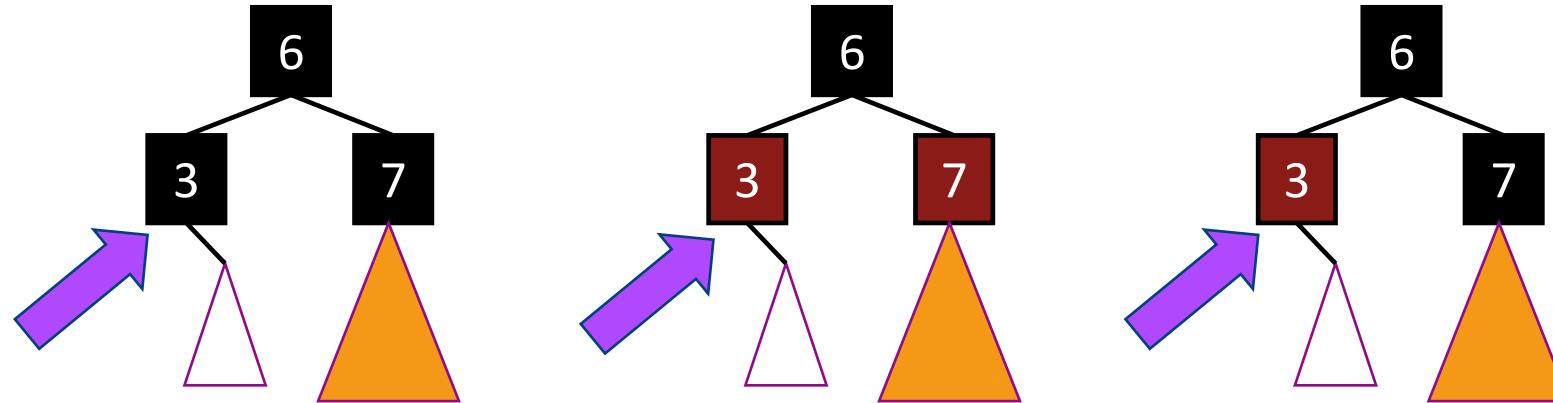
- Prove it?

# Insert / Delete

- Since the insertion and deletion in RB Trees are complicated, you don't need to master the details of them.
  - You should know what the “proxy for balance” property is and why it ensures approximate balance.
  - You should know **that** this property can be efficiently maintained, but you do not need to know the details of how.

# Insert

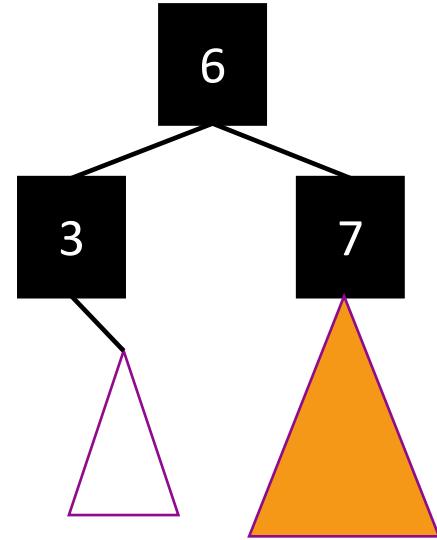
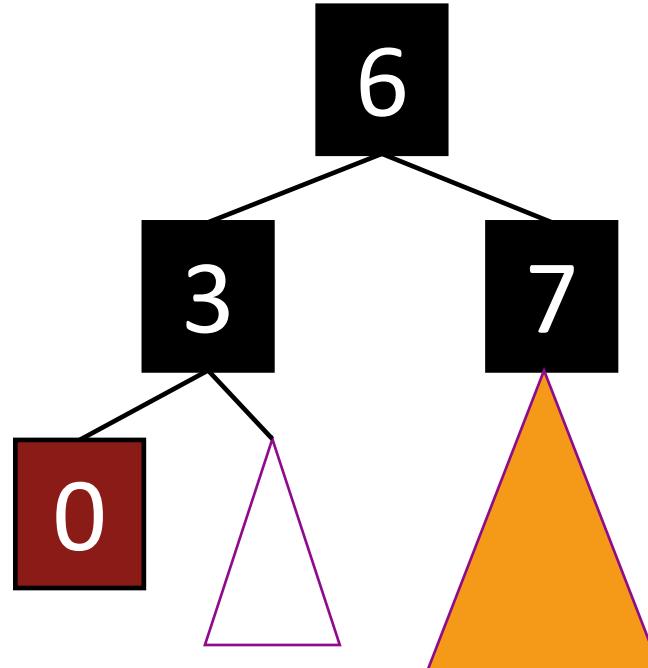
## Many cases



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

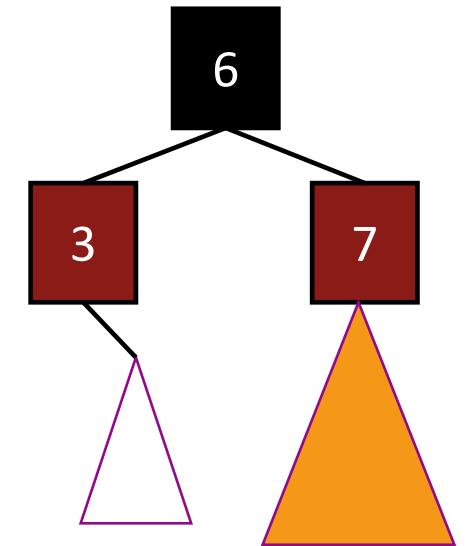
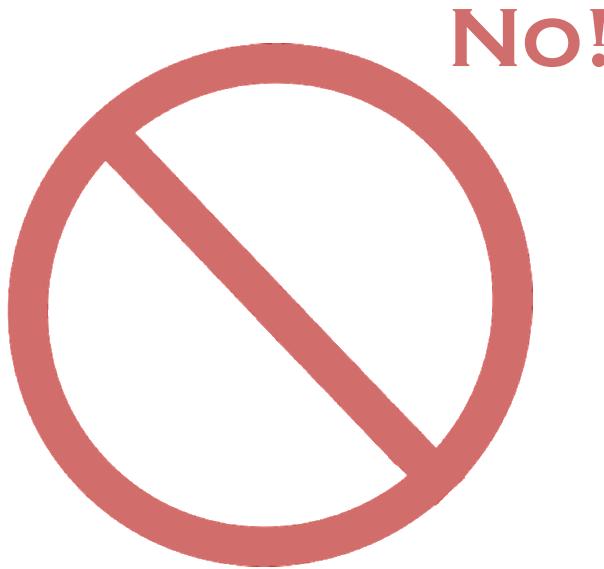
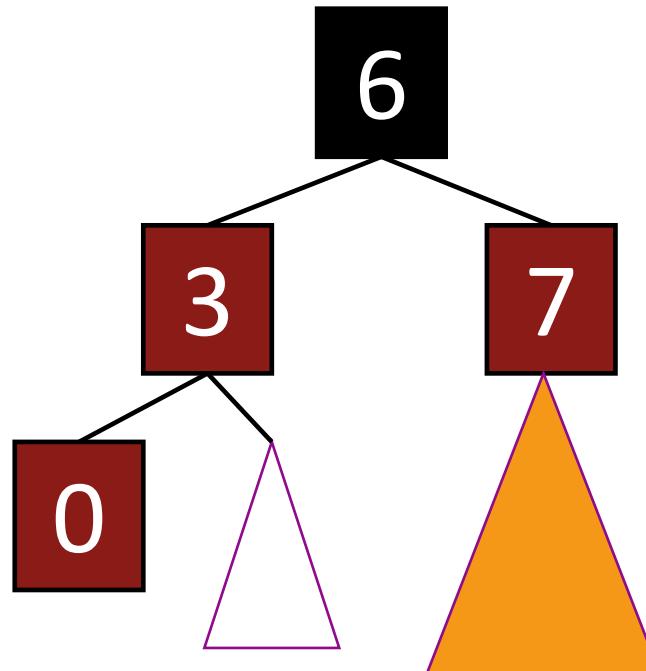
# Insert: Case 1

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



# Insert: Case 2

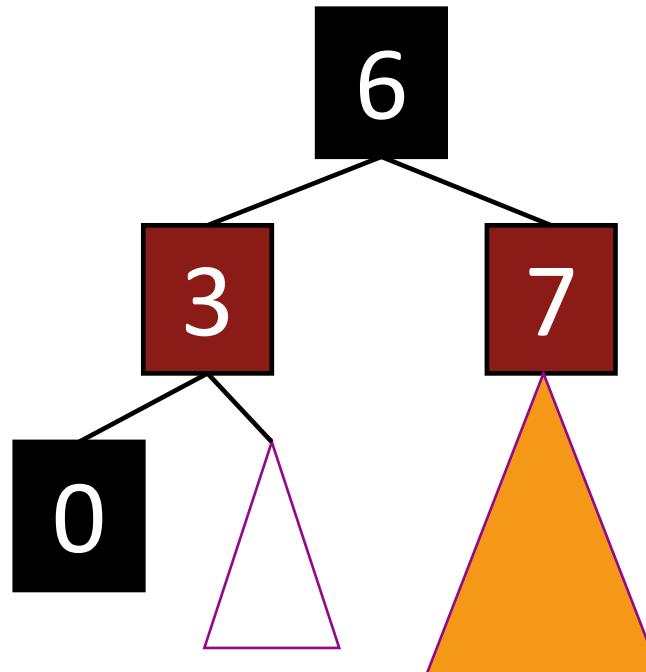
- Make a new **red node**.
- Insert it as you would normally?
- Fix things up if needed.



What if it looks like this?

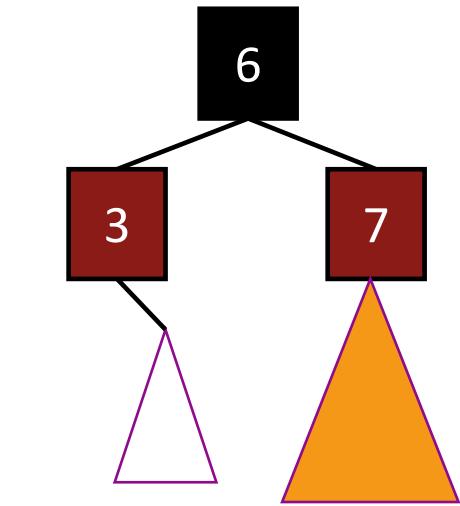
# Insert: Case 2

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



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

No!

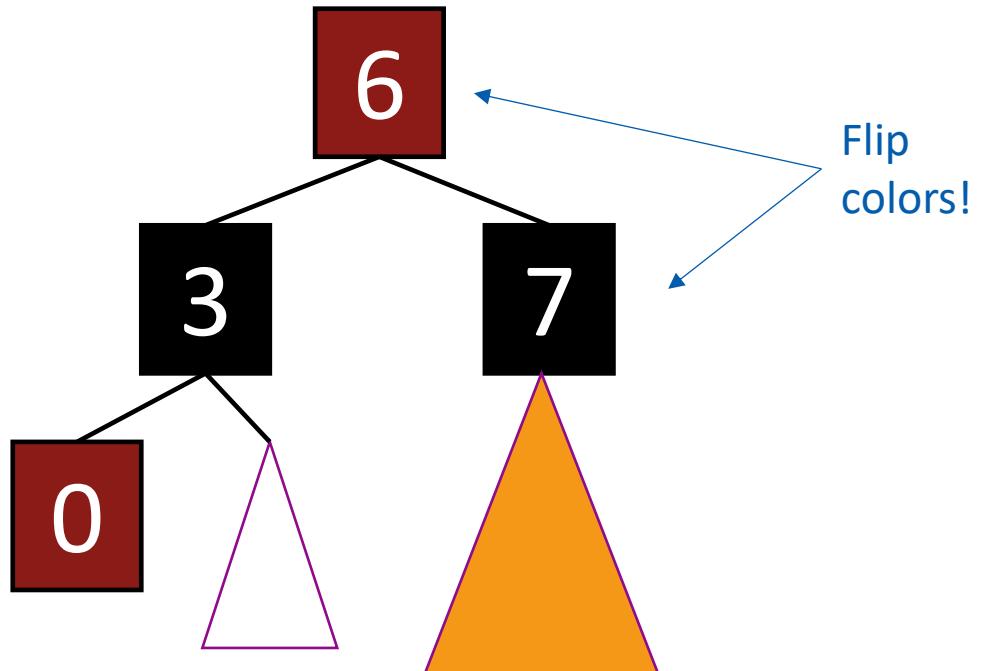


What if it looks like this?

One more black  
node in this path!

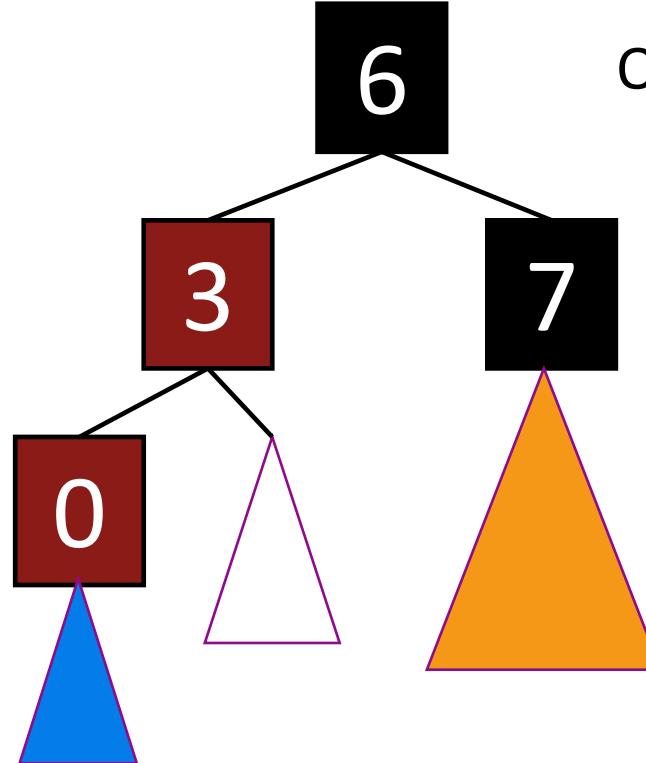
# Insert: Case 2

- An important observation: The root can be switched from red to black without violating any rule.
- Add 0 as a **red** node.
- Flip the colors of its parent and uncle.
- Pass the **red** to the top.
- Flip the color of the root from **red** to **black**.



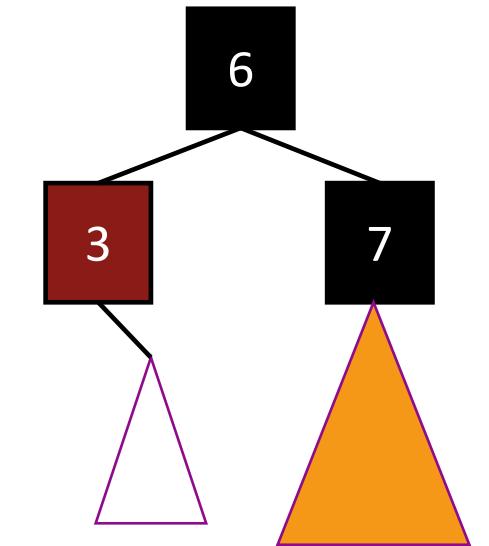
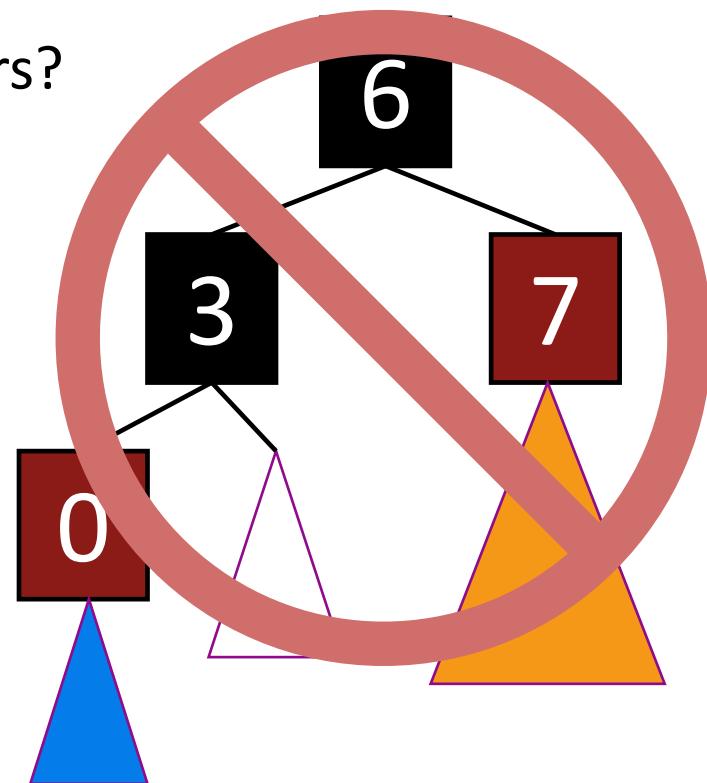
# Insert: Case 3

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



Only flip colors?

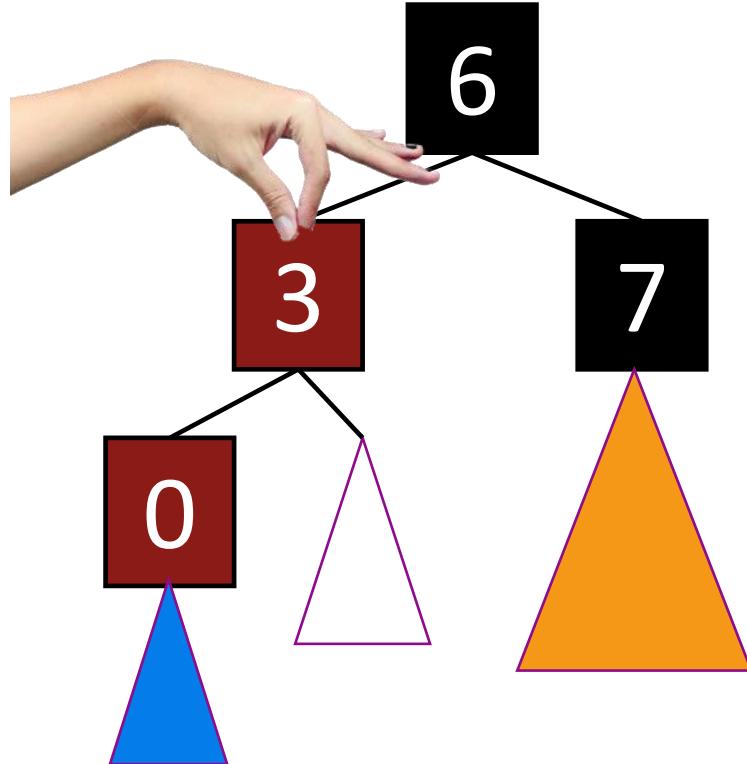
No!



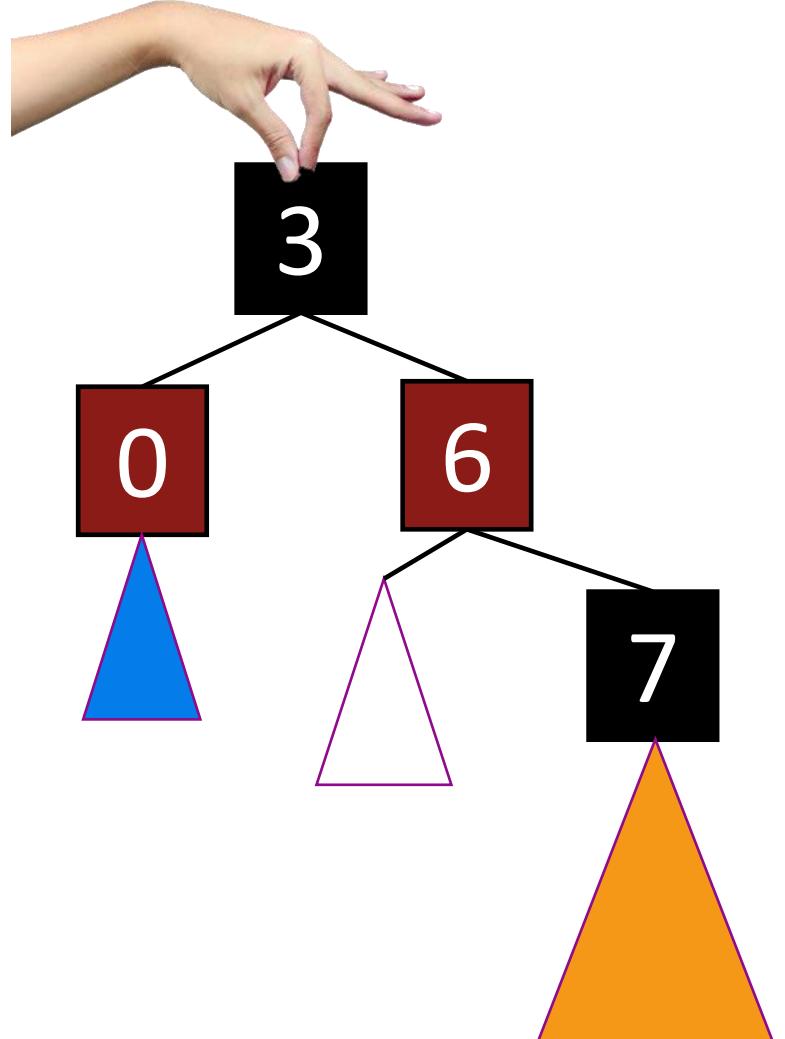
What if it looks like this?

# Insert: Case 3

- Recall Rotations:

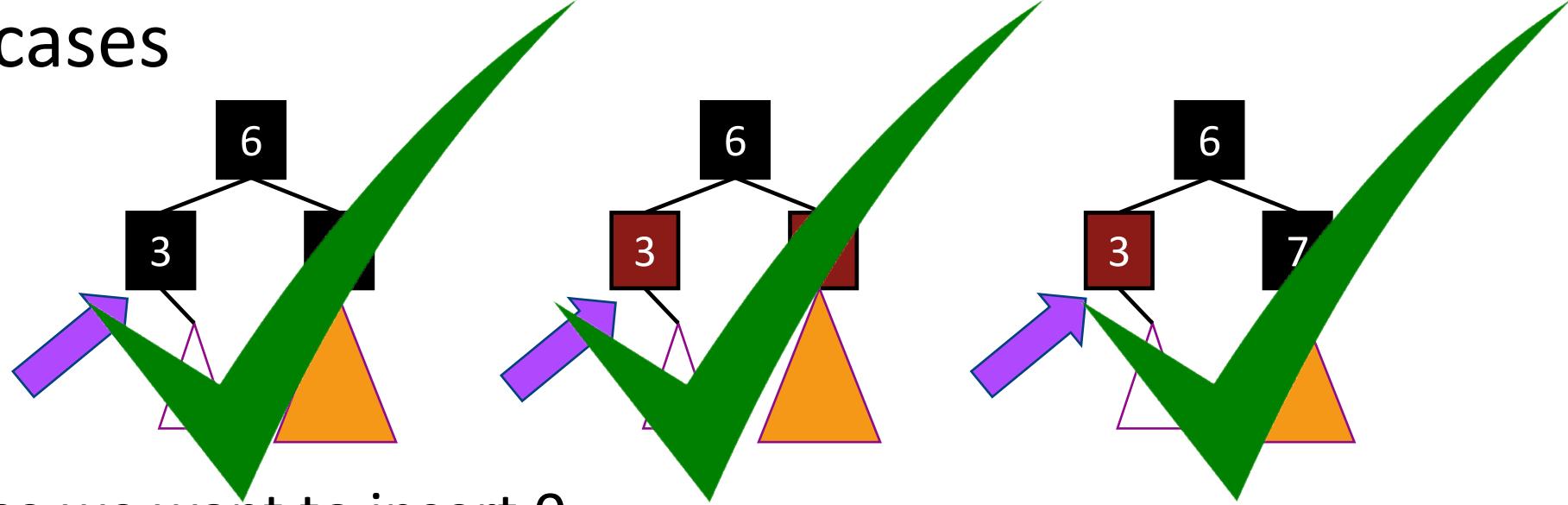


Rotate  
+  
Flip color



# Insert

Many cases



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

# (Binary) Heaps

# Revisiting FindMin

- Application: Find the smallest (or highest priority) item quickly
  - **Operating system** needs to schedule jobs according to priority instead of FIFO
  - **Event simulation** (bank customers arriving and departing, ordered according to when the event happened)
  - **Find** student with highest grade, employee with highest salary etc.

# Priority Queue ADT

- Priority Queue can efficiently do:
  - FindMin (and DeleteMin)
  - Insert
- What if we use...
  - Lists: If sorted, what is the run time for Insert and FindMin? Unsorted?
  - Binary Search Trees: What is the run time for Insert and FindMin?
  - Hash Tables (Maybe next lecture): What is the run time for Insert and FindMin?

# Less Flexibility → More Speed

- **Lists**
  - If sorted: FindMin is  $O(1)$  but Insert is  $O(N)$
  - If not sorted: Insert is  $O(1)$  but FindMin is  $O(N)$
- **Balanced Binary Search Trees (BSTs)**
  - Insert is  $O(\log N)$  and FindMin is  $O(\log N)$
- **Hash Tables**
  - Insert  $O(1)$  but  $O(N)$  for FindMin
- BSTs look good but...
  - **BSTs are efficient for all Finds, not just FindMin**
  - **We only need FindMin**

# Better than a speeding BST

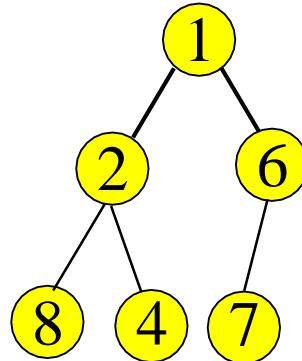
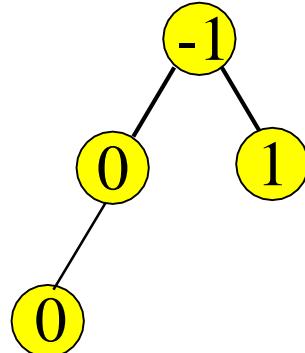
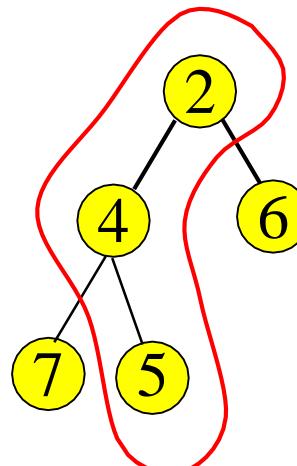
- Can we do better than Balanced Binary Search Trees?
  - Very limited requirements: Insert, FindMin, DeleteMin
  - The goals are:
    - FindMin is  $O(1)$
    - Insert is  $O(\log N)$
    - DeleteMin is  $O(\log N)$

# Binary Heaps

- A binary heap is a binary tree (NOT a BST) that is:
  - Complete: the tree is completely filled except possibly the bottom level, which is filled from left to right
  - Satisfies the heap order property
    - every node is less than or equal to its children (MinHeap, the default)
    - or every node is greater than or equal to its children (for MaxHeap)
- The root node is always the smallest node
  - or the largest, depending on the heap order (for MaxHeap)

# Heap order property

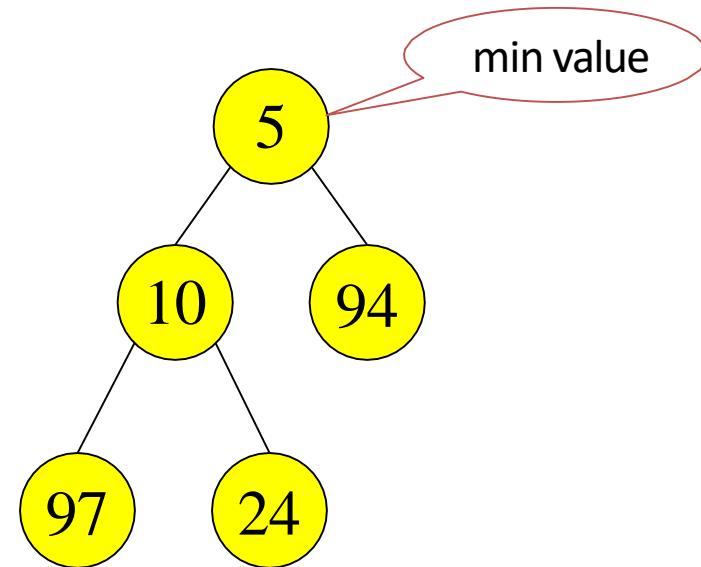
- A heap provides limited ordering information
- Each *path* is sorted, but the subtrees are not sorted relative to each other
  - A binary heap is NOT a binary search tree



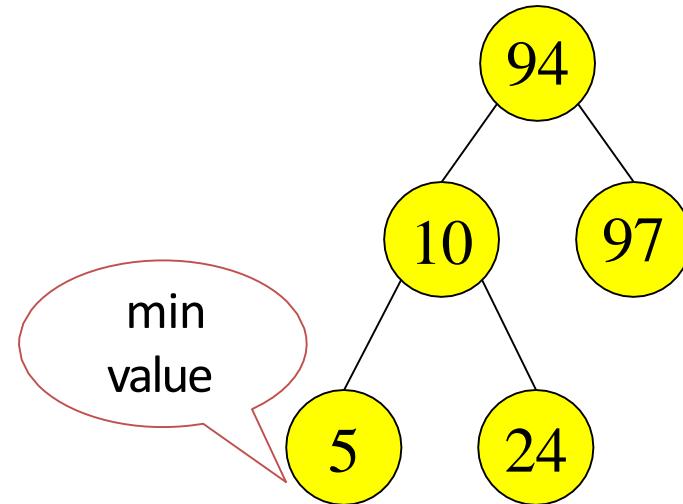
These are all valid binary min heaps

# Binary Heap vs Binary Search Tree

Binary Heap



Binary Search Tree

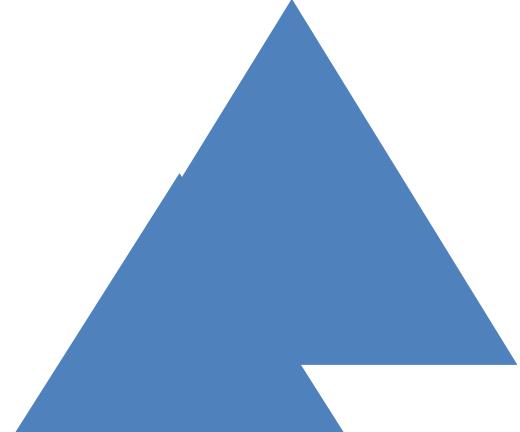


Parent is less than both  
left and right children

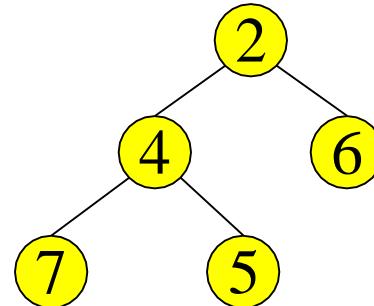
Parent is greater than left  
child, less than right child

# Structure Property

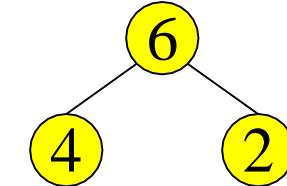
- A binary heap is a complete tree.
  - All nodes are in use except for possibly the right end of the bottom row.



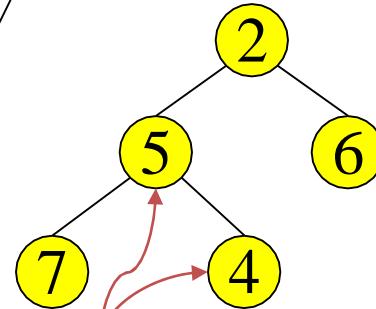
# Examples



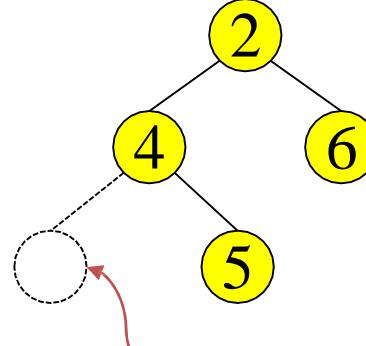
complete tree,  
heap order is  
"min"



complete tree,  
heap order is "max"



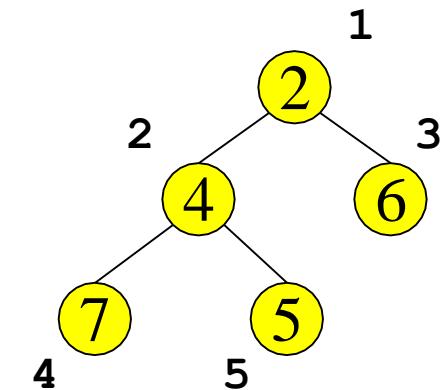
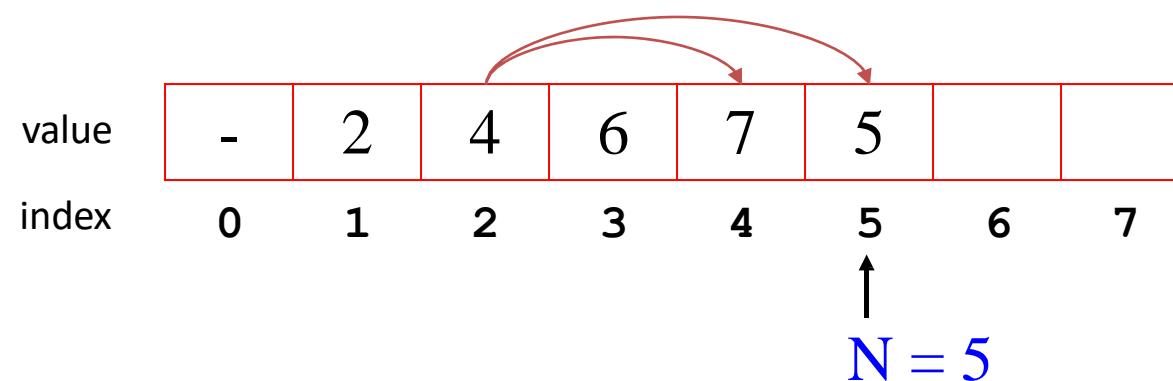
complete tree, but  
min heap order is  
broken



not  
complete

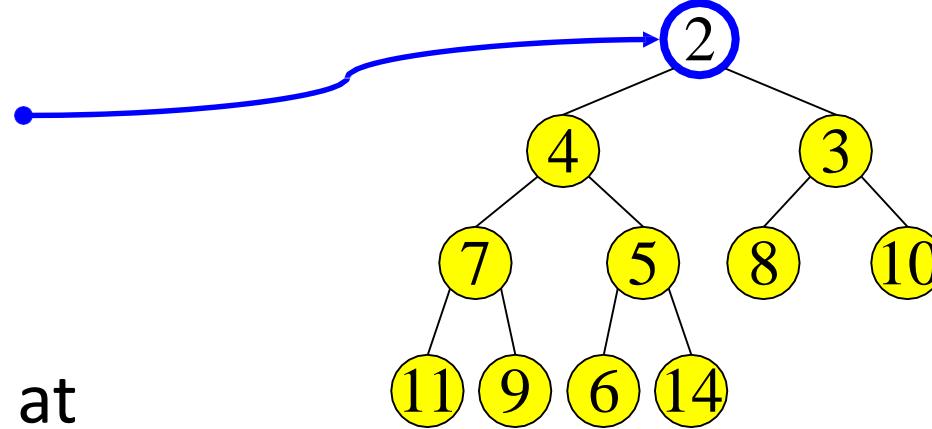
# Array Implementation (Implicit Pointers)

- Root node =  $A[1]$
- Children of  $A[i]$  =  $A[2i], A[2i + 1]$
- Parent of  $A[j]$  =  $A[j // 2]$
- Keep track of current size  $N$  (number of nodes)



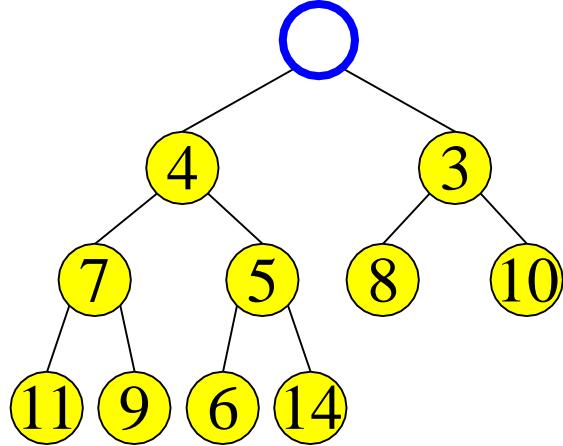
# FindMin and DeleteMin

- FindMin: Easy!
  - Return root value A[1]
  - Run time = ?
- DeleteMin:
  - Delete (and return) value at root node?



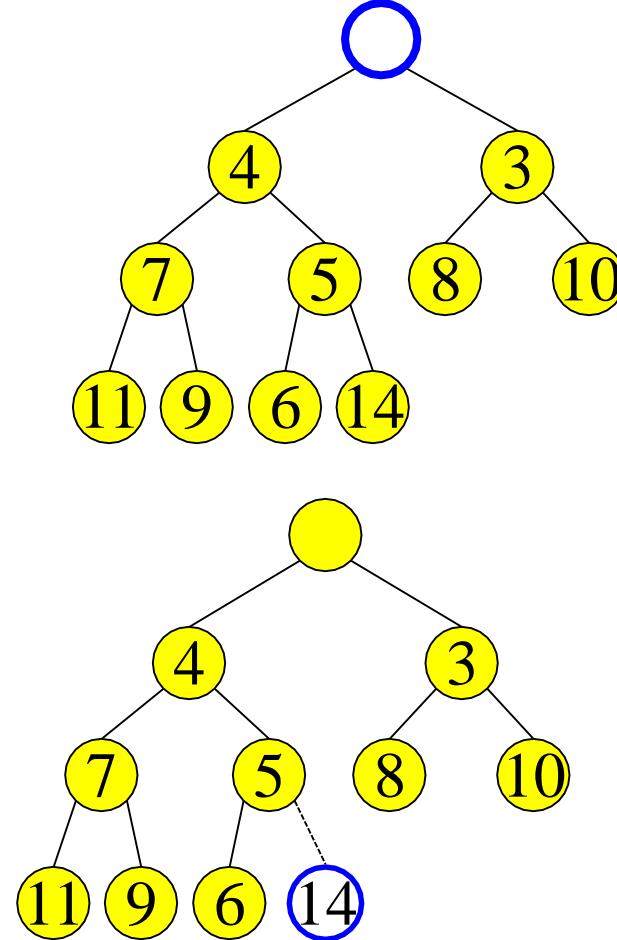
# Maintain the Structure Property

- Delete (and return) value at root node



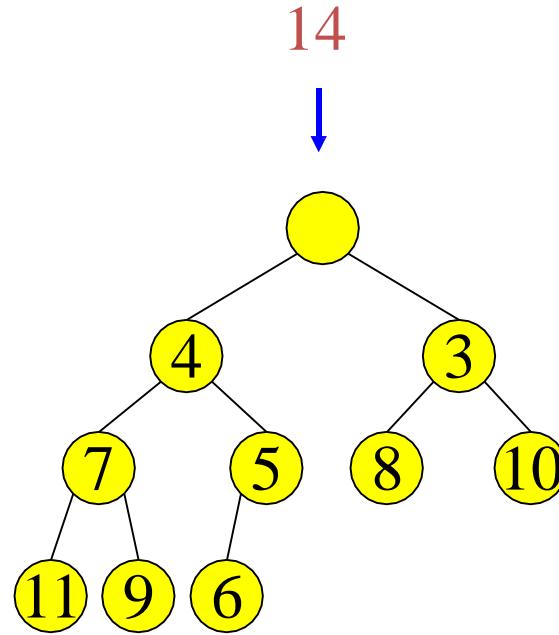
# Maintain the Structure Property

- We now have a “Hole” at the root
  - Need to fill the hole with another value
- When we get done, the tree will have one less node and **must still be complete.**

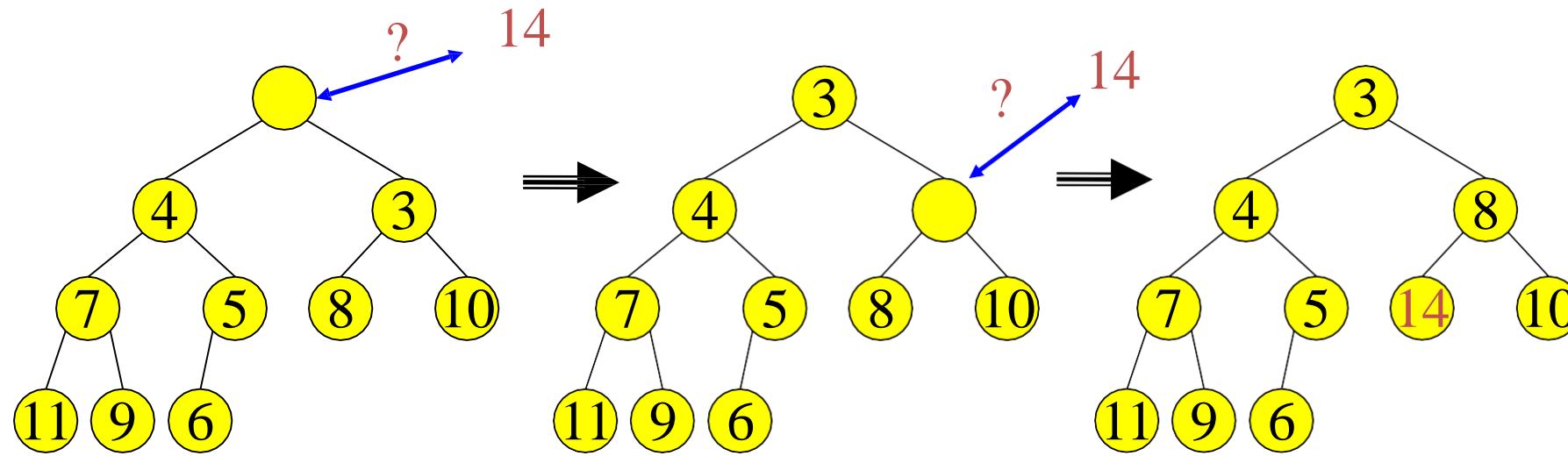


# Maintain the Heap Property

- The last value has lost its node
  - we need to find a new place for it



# DeleteMin: Percolate Down



- Keep comparing with children  $A[2i]$  and  $A[2i + 1]$
- Copy smaller child up and go down one level
- Done if both children are  $\geq$  item or reached a leaf node
- What is the run time?

# Percolate Down

```

PercDown(i: integer, x: integer): {
    // N is the number elements, i is the hole, x is the value to insert
    Case {
        No child    2i > N: A[i] := x;           // At bottom
        One child at the end 2i = N: if A[2i] < x then A[i]:= A[2i]; A[2i] := x
                                         else A[i] := x
        Two Children 2i < N: if A[2i] < A[2i+1] then j := 2i
                               else j := 2i+1
                               if A[j] < x then
                                   A[i]:= A[j]; PercDown(j, x);
                               else A[i] := x
    }
}

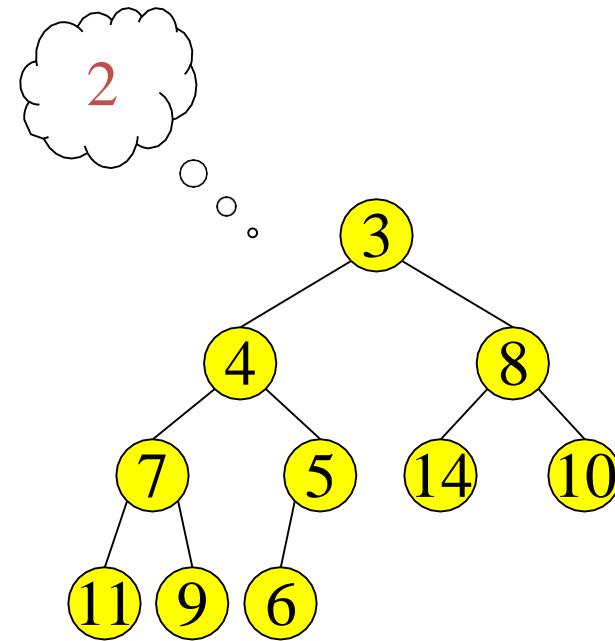
```

# DeleteMin: Run Time Analysis

- Run time is  $O(\text{depth of heap})$
- A heap is a complete binary tree
- Depth of a complete binary tree of  $N$  nodes?
  - depth =  $\log(N)$
- Run time of DeleteMin is  $O(\log N)$

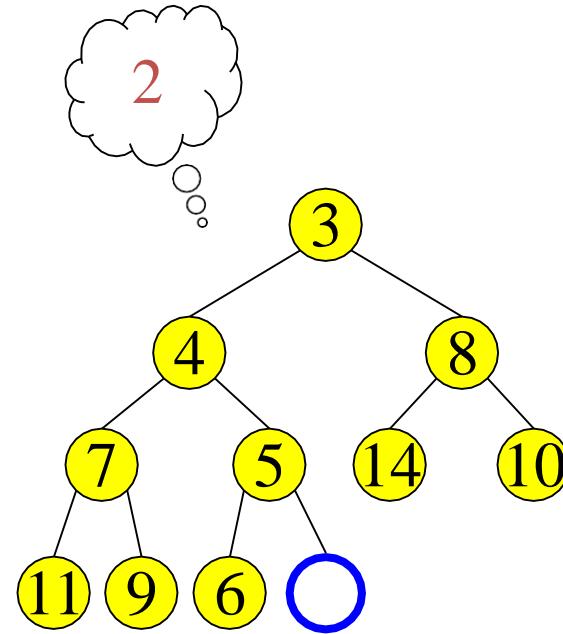
# Insert

- Add a value to the tree
- Structure and heap order properties must still be correct when we are done



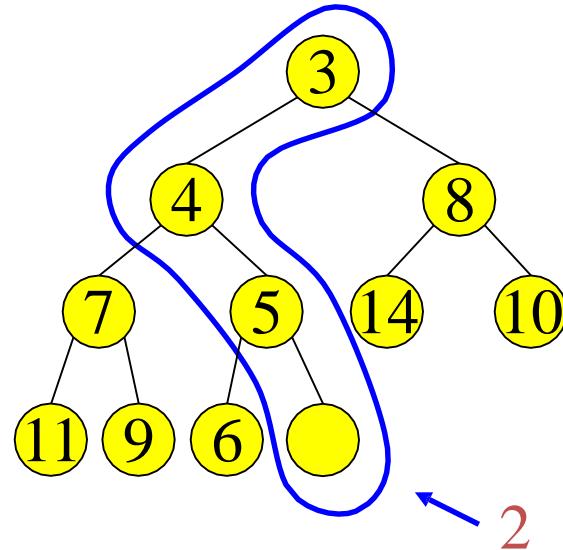
# Maintain the Structure Property

- The only valid place for a new node in a complete tree is at the end of the array
- We need to decide on the correct value for the new node, and adjust the heap accordingly

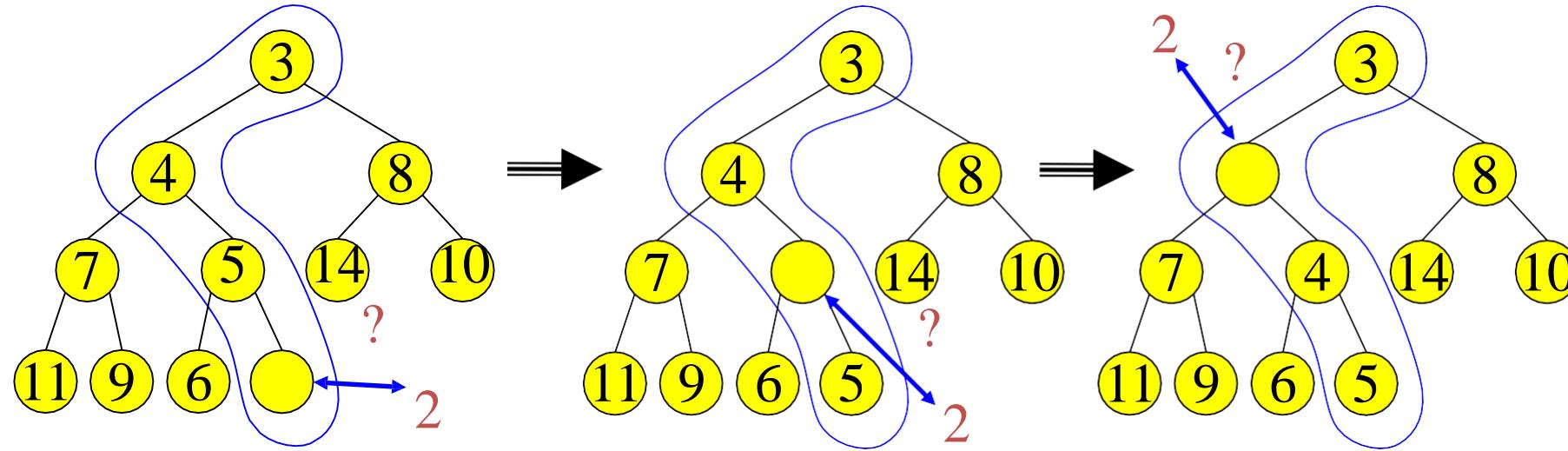


# Maintain the Heap Property

- The new value goes where?

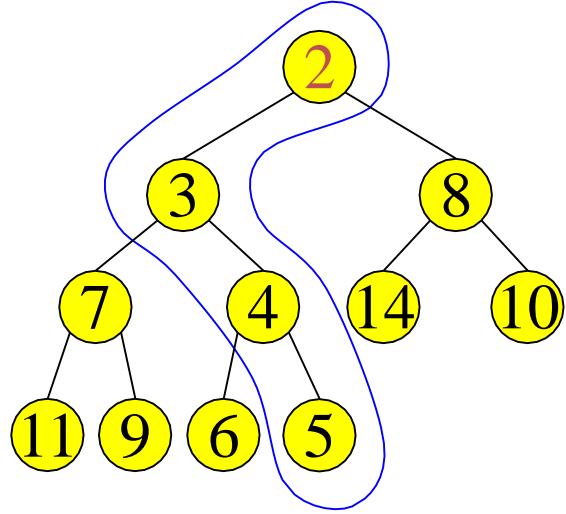


# Insert: Percolate Up



- Start at last node and keep comparing with parent  $A[i/2]$
- If parent larger, copy parent down and go up one level
- Done if parent  $\leq$  item or reached top node  $A[1]$

# Insert: Done



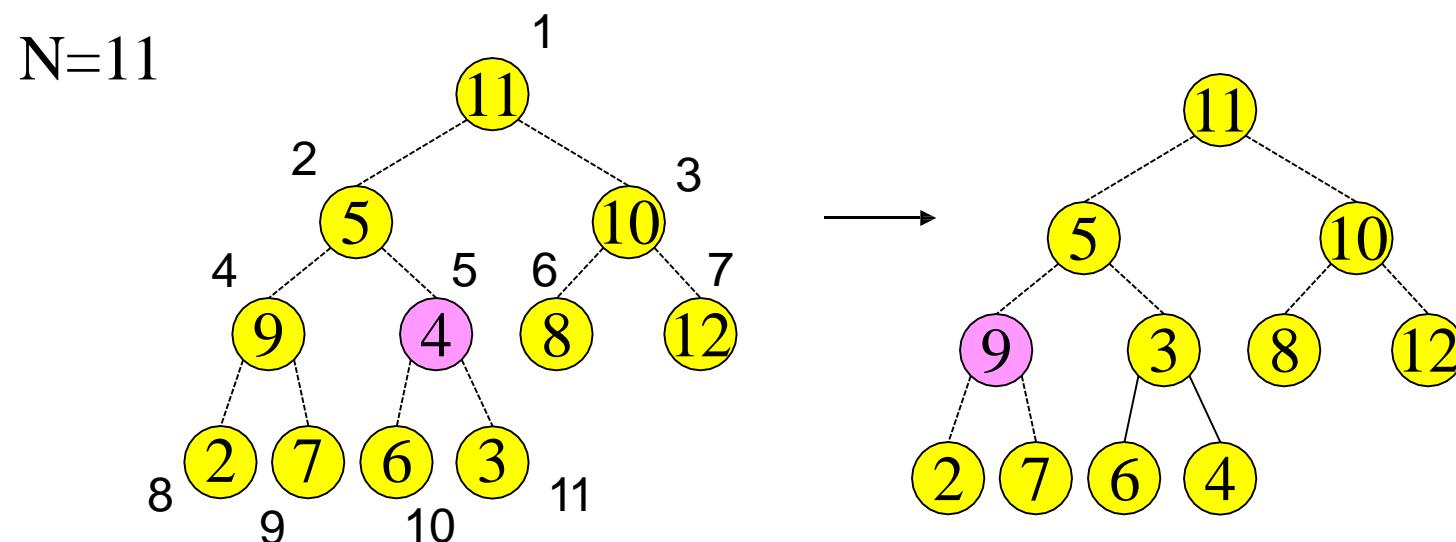
- Run time?

# Binary Heap Analysis

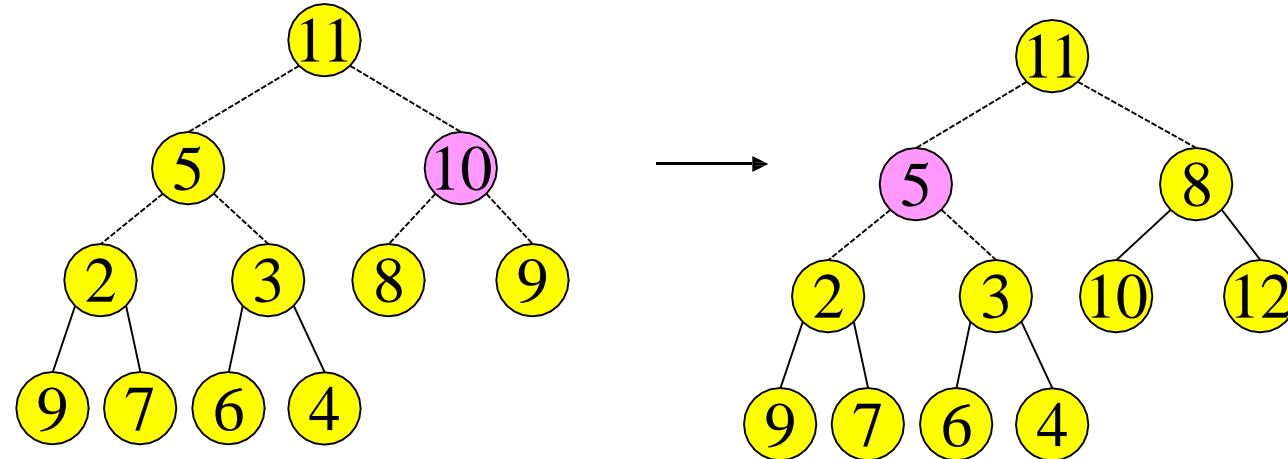
- **Space** needed for heap of N nodes:  $O(\text{MaxN})$ 
  - An array of size MaxN, plus a variable to store the size N
- **Time**
  - FindMin:  $O(1)$
  - DeleteMin and Insert:  $O(\log N)$
  - BuildHeap from N inputs ???

# Build Heap

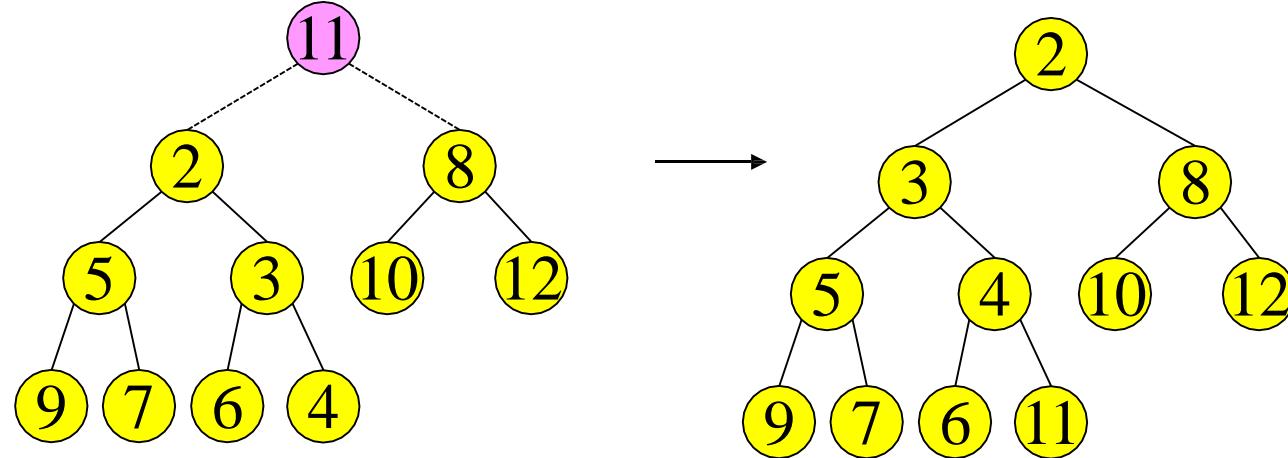
```
BuildHeap {
    for i = N/2 to 1
        PercDown(i, A[i])
}
```



# Build Heap



# Build Heap



# Time Complexity

- Naïve considerations:
  - $n/2$  calls to PercDown, each takes  $c \cdot \log(n)$
  - Total:  $c \cdot n \cdot \log(n)$
- More careful considerations:
  - Only  $\mathcal{O}(n)$

# Analysis of Build Heap

Assume  $n = 2^{h+1} - 1$  where  $h$  is height of the tree

- Thus, level  $h$  has  $2^h$  nodes but there is nothing to PercDown
- At level  $h - 1$  there are  $2^{h-1}$  nodes, each might percolate down 1 level
- At level  $h - j$ , there are  $2^{h-j}$  nodes, each might percolate down  $j$  levels

$$T(n) = \sum_{j=0}^h j 2^{h-j} = \sum_{j=0}^h j \frac{2^h}{2^j}$$

Total Time =  $O(n)$

# Other Heap Operations

- **Find( $X, H$ ):** Find the element  $X$  in heap  $H$  of  $N$  elements
  - What is the running time?  $O(N)$
- **FindMax( $H$ ):** Find the maximum element in  $H$
- Where FindMin is  $O(1)$ 
  - What is the running time?  $O(N)$
- We sacrificed performance of these operations in order to get  $O(1)$  performance for FindMin

# Other Heap Operations

- DecreaseKey( $P, \Delta, H$ ): Decrease the key value of node at position  $P$  by a positive amount  $\Delta$ , e.g., to increase priority
  - First, subtract  $\Delta$  from current value at  $P$
  - Heap order property may be violated
  - so percolate up to fix
  - Running Time:  $O(\log N)$

# Other Heap Operations

- Delete( $P, H$ ): E.g. Delete a job waiting in queue that has been preemptively terminated by user
  - Use DecreaseKey( $P, \Delta, H$ ) followed by DeleteMin
  - Running Time:  $O(\log N)$
- Merge( $H_1, H_2$ ): Merge two heaps  $H_1$  and  $H_2$  of size  $O(N)$ .  $H_1$  and  $H_2$  are stored in two arrays.
  - Can do  $O(N)$  Insert operations:  $O(N \log N)$  time
  - Better: Copy  $H_2$  at the end of  $H_1$  and use BuildHeap.  
Running Time:  $O(N)$

# Other Heap Operations

- Merge( $H_1, H_2$ ): Merge two heaps  $H_1$  and  $H_2$  of size  $O(N)$ .  
 $H_1$  and  $H_2$  are stored in two arrays.
  - Can do  $O(N)$  Insert operations:  $O(N \log N)$  time
  - Better: Copy  $H_2$  at the end of  $H_1$  and use BuildHeap.  
Running Time:  $O(N)$

# Heap Sort

- Idea: buildHeap then call deleteMin  $n$  times

```
input = buildHeap(...);  
output = new E[n];  
for (int i = 0; i < n; i++) {  
    output[i] = deleteMin(input);  
}
```

- Runtime?
  - Best-case \_\_\_\_\_
  - Worst-case \_\_\_\_\_
  - Average-case \_\_\_\_\_
- Stable? \_\_\_\_\_
- In-place? \_\_\_\_\_

# Heap Sort

- Idea: buildHeap then call deleteMin  $n$  times

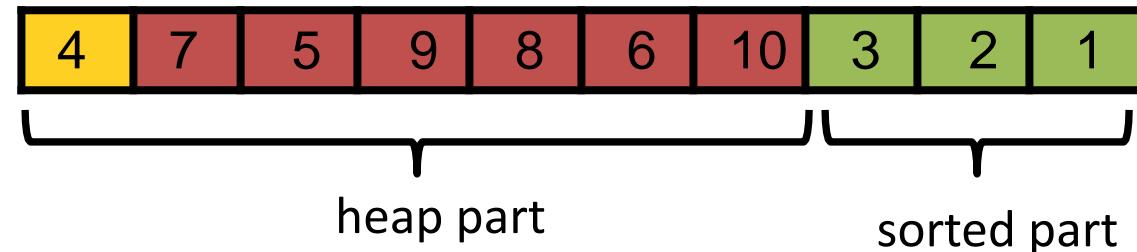
```
input = buildHeap(...);  
output = new E[n];  
for (int i = 0; i < n; i++) {  
    output[i] = deleteMin(input);  
}
```

- Runtime?
  - Best-case, Worst-case, and Average-case:  $O(n \log(n))$
- Stable? No.
- In-place? No. But it could be, with a slight trick...

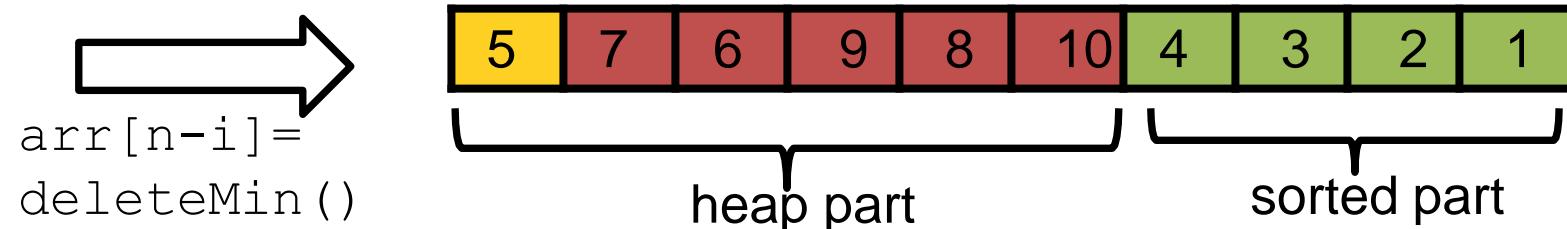
# In-place Heap Sort

- Treat the initial array as a heap (via buildHeap)
- When you delete the  $i^{\text{th}}$  element, put it at  $\text{arr}[n-i]$ 
  - That array location isn't needed for the heap anymore!

But this reverse sorts  
– how would you fix  
that?



put the min at the end of the heap data



# AVL Sort?

Sure, we can also use an AVL tree to:

- Insert each element: total time  $O(n \log n)$
- Repeatedly deleteMin: total time  $O(n \log n)$ 
  - Better: in-order traversal  $O(n)$ , but still  $O(n \log n)$  overall
- But this **cannot be done in-place** and has **worse constant factors** than heap sort.

# The End