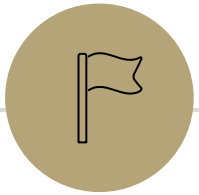

# Lecture 11: Intro to Heaps

CSE 373 Data Structures and Algorithms



# Priority Queue *ADT*

## Binary Heap

## Binary Heap Methods

# Motivation

Some motivation for today's lecture:

- Priority Queues are a staple of Java's built-in data structures, commonly used for sorting needs
- Using Priority Queues and knowing their implementations are common technical interview subjects
- You're implementing one in the next project – so everything you get out of today should be useful for that!

# A new ADT!

Imagine you're managing a queue of food orders at a restaurant, which normally takes food orders first-come-first-served.

Suddenly, Ana Mari Cauce walks into the restaurant!

You realize that you should serve her as soon as possible (to gain political influence or so that she leaves the restaurant as soon as possible), and realize other celebrities (CSE 373 staff) could also arrive soon. Your new food management system should rank customers and let us know which food order we should work on next (the most prioritized thing).

# Priority Queue ADT

## Min Priority Queue ADT

**state**

Set of comparable values
- Ordered based on "priority"

**behavior**

**add(value)** – add a new element to the collection

**removeMin()** – returns the element with the smallest priority, removes it from the collection

**peekMin()** – find, but do not remove the element with the smallest priority

Perfect for our Ana Mari Cauce food order situation

Other uses:

- Well-designed printers
- Huffman Coding (CSE 143/123 last hw)
- Sorting algorithms
- Graph algorithms

# Priority Queue ADT

If a Queue is "First-In-First-Out" (FIFO) Priority Queues are "Most-Important-Out-First"

Items in Priority Queue must be comparable – The data structure will maintain some amount of internal sorting, in a sort of similar way to BSTs/AVLs

**Min Priority Queue ADT**

**state**

Set of comparable values – Ordered based on "priority"

**behavior**

**removeMin()** – returns the element with the smallest priority, removes it from the collection

**peekMin()** – find, but do not remove the element with the smallest priority

**add(value)** – add a new element to the collection

**Max Priority Queue ADT**

**state**

Set of comparable values – Ordered based on "priority"

**behavior**

**removeMax()** – returns the element with the largest priority, removes it from the collection

**peekMax()** – find, but do not remove the element with the largest priority

**add(value)** – add a new element to the collection

# Implementing Priority Queues: Take I

Maybe we already know how to implement a priority queue.
How long would removeMin and peek take with these data structures?

| Implementation | add | removeMin | Peek |
|---|---|---|---|
| Unsorted Array | | | |
| Linked List (sorted) | | | |
| AVL Tree | | | |

For Array implementations, assume you do not need to resize.
Other than this assumption, do **worst case** analysis.

# Implementing Priority Queues: Take I

Maybe we already know how to implement a priority queue.
How long would removeMin and peek take with these data structures?

| Implementation | add | removeMin | Peek |
|---|---|---|---|
| Unsorted Array | $\Theta(1)$ | $\Theta(n)$ | $\Theta(n)$ |
| Linked List (sorted) | $\Theta(n)$ | $\Theta(1)$ | $\Theta(1)$ |
| AVL Tree | $\Theta(\log n)$ | $\Theta(\log n)$ | $\Theta(\log n)$ |

For Array implementations, assume you do not need to resize.
Other than this assumption, do **worst case** analysis.

# Implementing Priority Queues: Take I

Maybe we already know how to implement a priority queue.

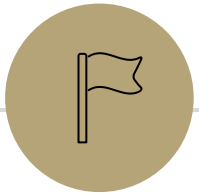
How long would removeMin and peek take with these data structures?

| Implementation | add | removeMin | Peek |
|---|---|---|---|
| Unsorted Array | $\Theta(1)$ | $\Theta(n)$ | ~~$\Theta(n)$~~ $\Theta(1)$ |
| Linked List (sorted) | $\Theta(n)$ | $\Theta(1)$ | $\Theta(1)$ |
| AVL Tree | $\Theta(\log n)$ | $\Theta(\log n)$ | ~~$\Theta(\log n)$~~ $\Theta(1)$ |

Add a field to keep track of the min.

Update on every insert or remove.

AVL Trees are our baseline – let's look at what computer scientists came up with as an alternative, analyze that, and then come back to AVL Tree as an option later



Priority Queue ADT

Binary Heap

Binary Heap Methods

# Heaps

Idea:

In a BST, we organized the data to find anything quickly. (go left or right to find a value deeper in the tree)

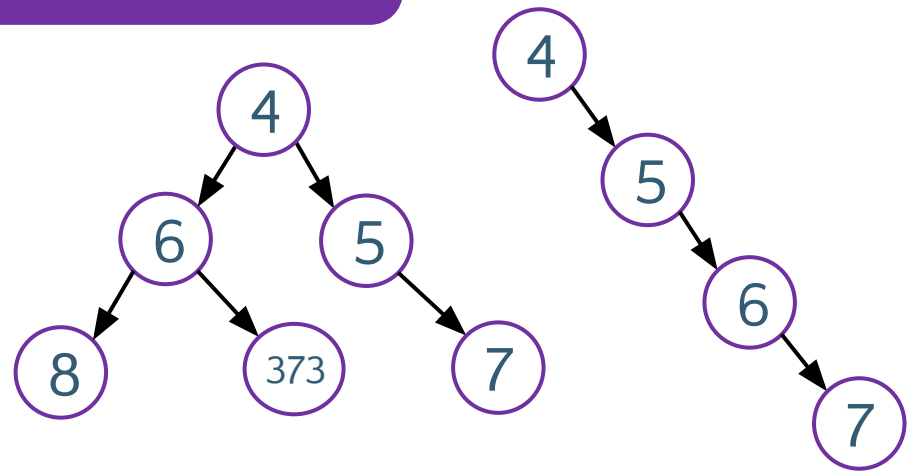
Now we just want to find the smallest things fast, so let's write a different invariant:

**Heap invariant**
Every node is less than or equal to both of its children.

In particular, the smallest node is at the root!

- Super easy to peek now!

Do we need more invariants?

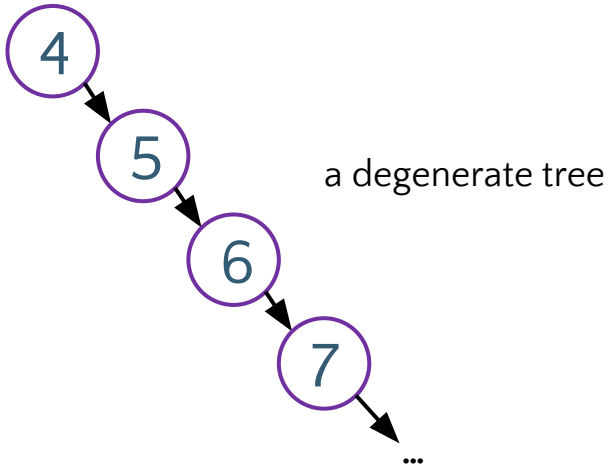


# Heaps

With the current definition we could still have degenerate trees.
From our BST / AVL intuition, we know that degenerate trees take a long time to traverse from root to leaf, so we want to avoid these tree structures.

The BST invariant was a bit complicated to maintain.

- Because we had to make sure when we inserted we could maintain the exact BST structure where nodes to the left are less than, nodes to the right are greater than…
- The heap invariant is looser than the BST invariant.
- Which means we can make our structure invariant stricter.



a degenerate tree

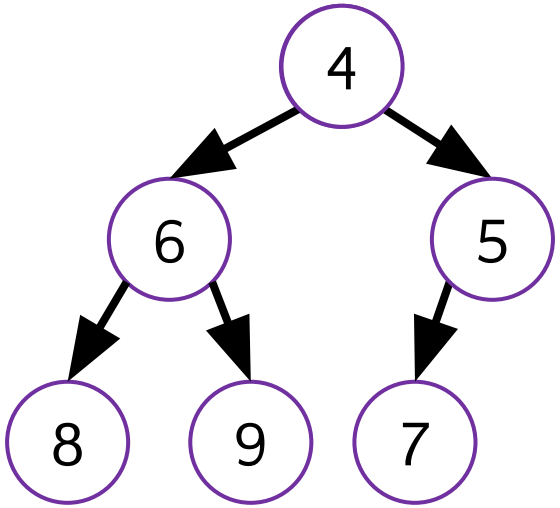
# Heaps

**Heap structure invariant:**
A heap is always a **complete** tree.

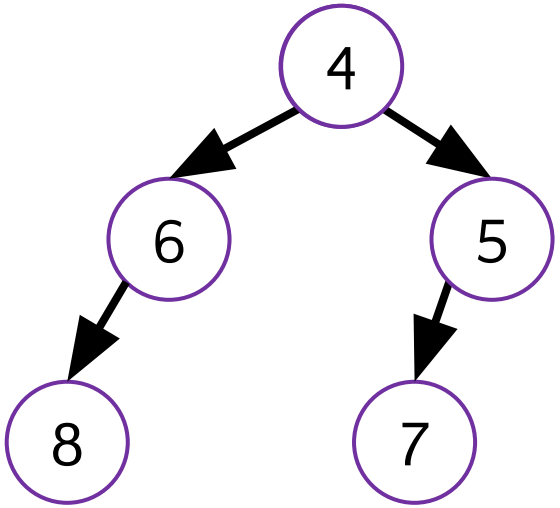
→ helps us avoid degenerate trees

A tree is complete if:

- Every row, except potentially the last, is completely full
- The last row is filled from left to right (no "gap")




complete



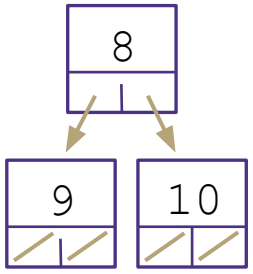

not complete

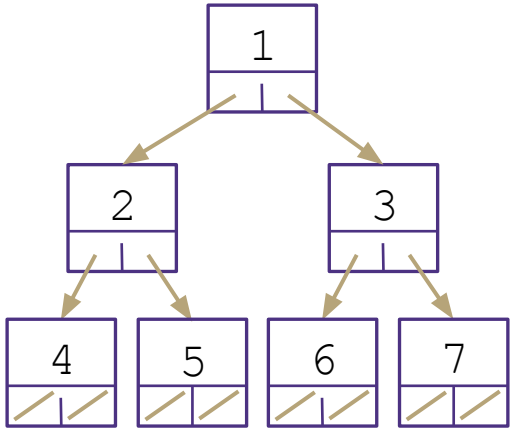
# Binary Heap invariants summary

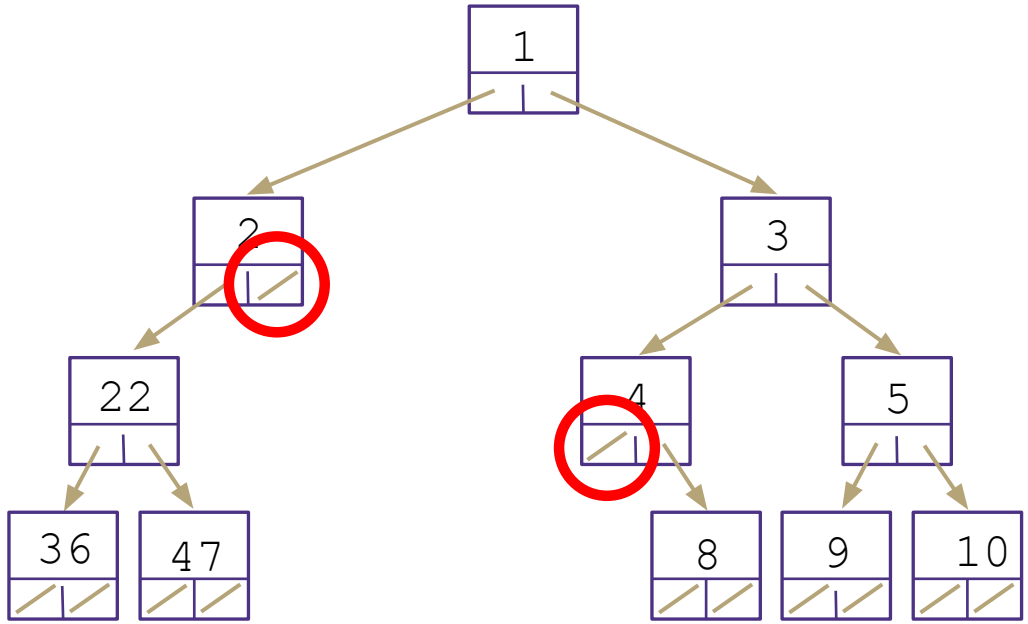
This is a big idea! (heap invariants!)

One flavor of heap is a **binary** heap.

1. **Binary Tree**: every node has at most 2 children
2. **Heap invariant**: every node is smaller than (or equal to) its children
3. **Heap structure invariant**: each level is "complete" meaning it has no "gaps"
   a. Heaps are filled up left to right



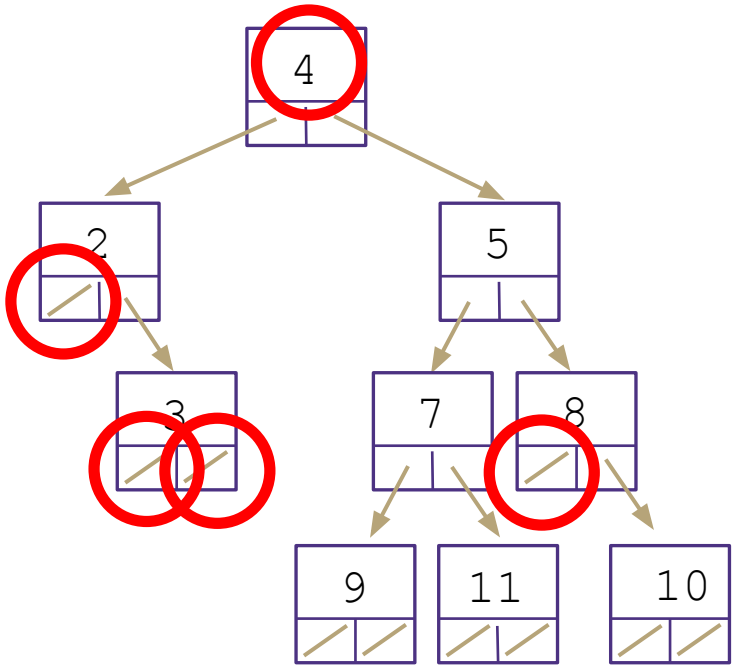



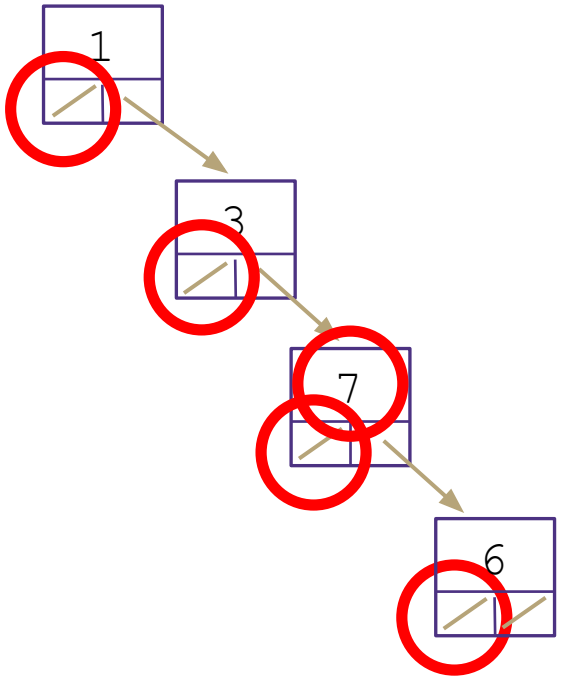
# Self Check – Are these valid heaps?

Binary Heap Invariants:
1. Binary Tree
2. Heap
3. Complete

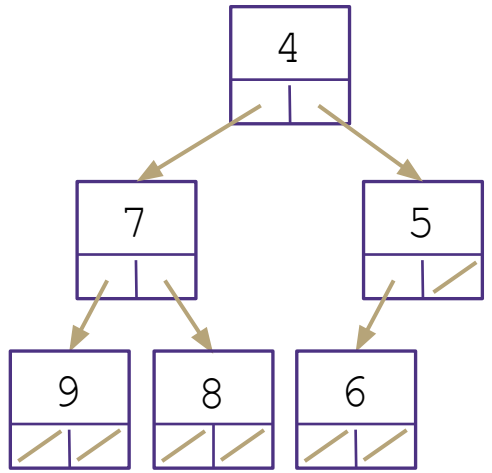
INVALID




INVALID



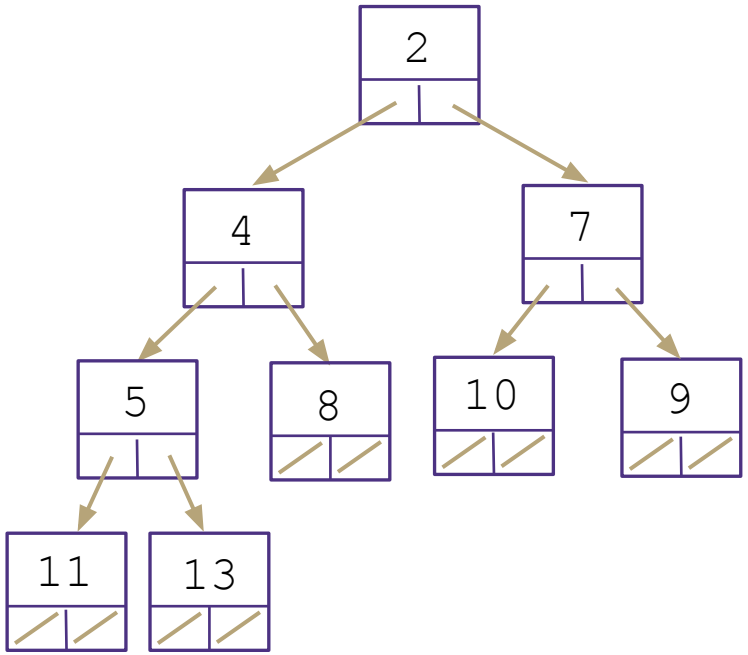

VALID

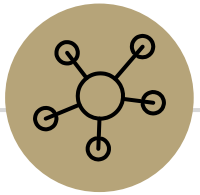


# Heap heights

**A binary heap bounds our height at Theta(log(n)) because it's complete** – and it's actually a little stricter and better than AVL.

This means the runtime to traverse from root to leaf or leaf to root will be log(n) time.



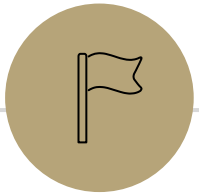


# Questions?

Priority Queue ADT
Priority Queue possible implementations
Heap invariants
Heap height

Priority Queue ADT

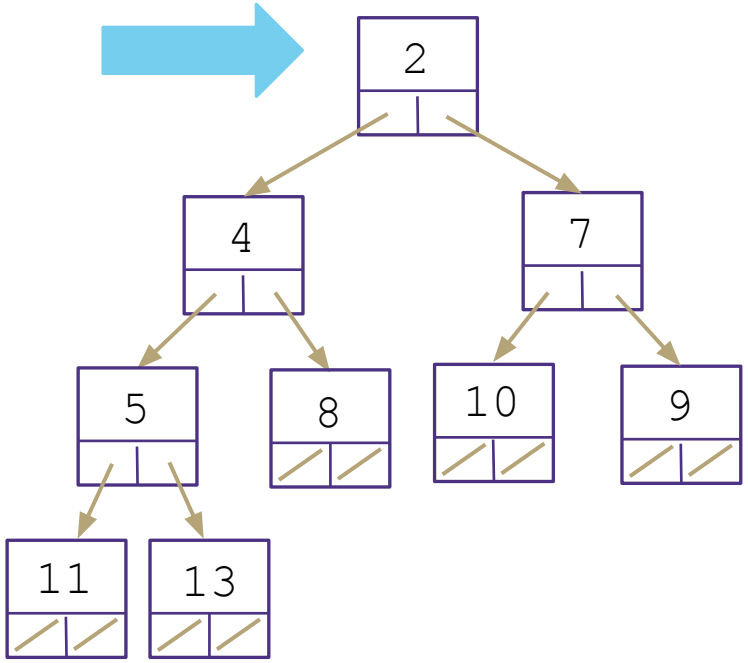
Binary Heap



# Binary Heap Methods

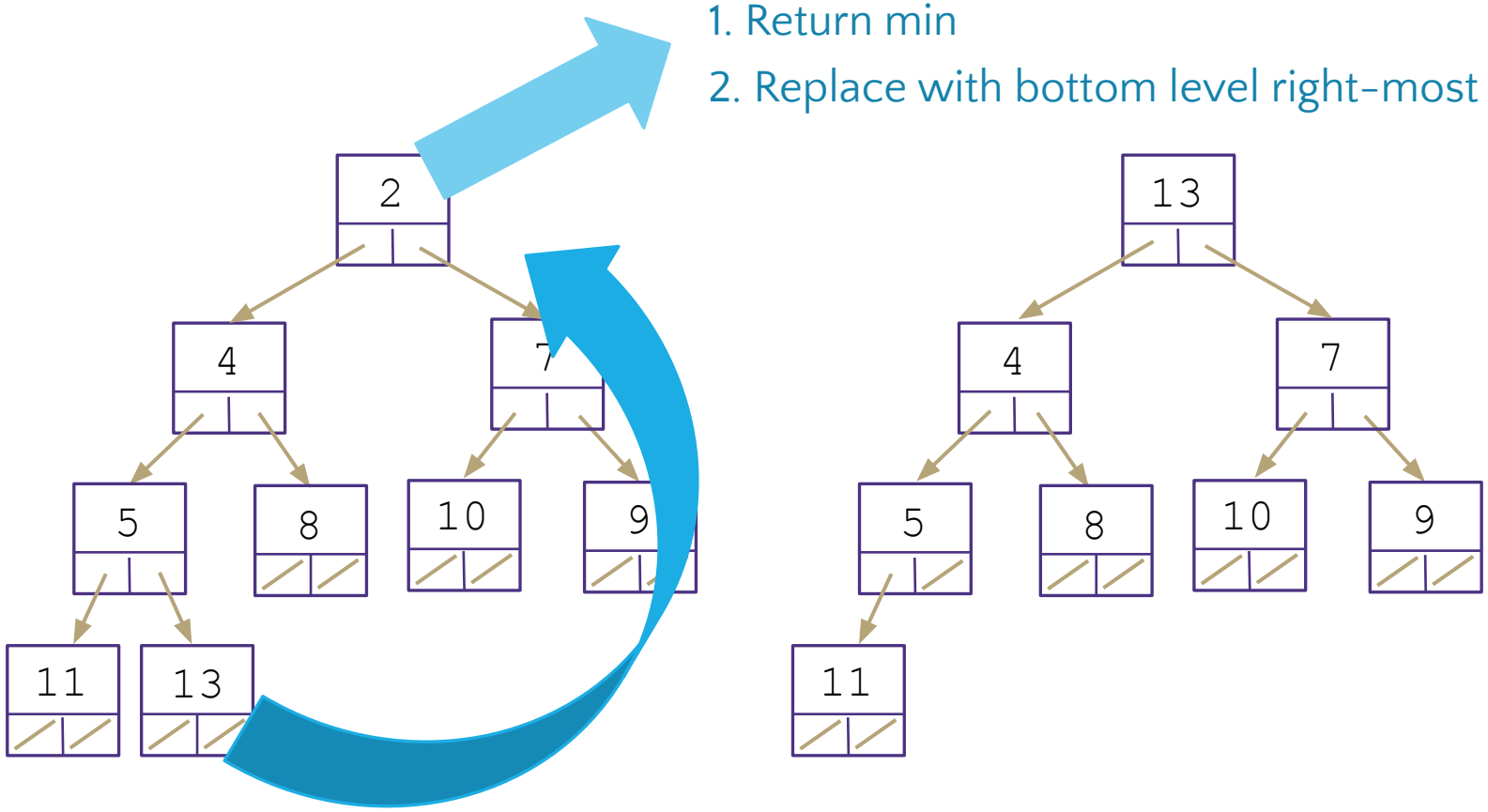
# Implementing peekMin()

Runtime: $\Theta(1)$



# Implementing removeMin()

1. Return min
2. Replace with bottom level right-most node







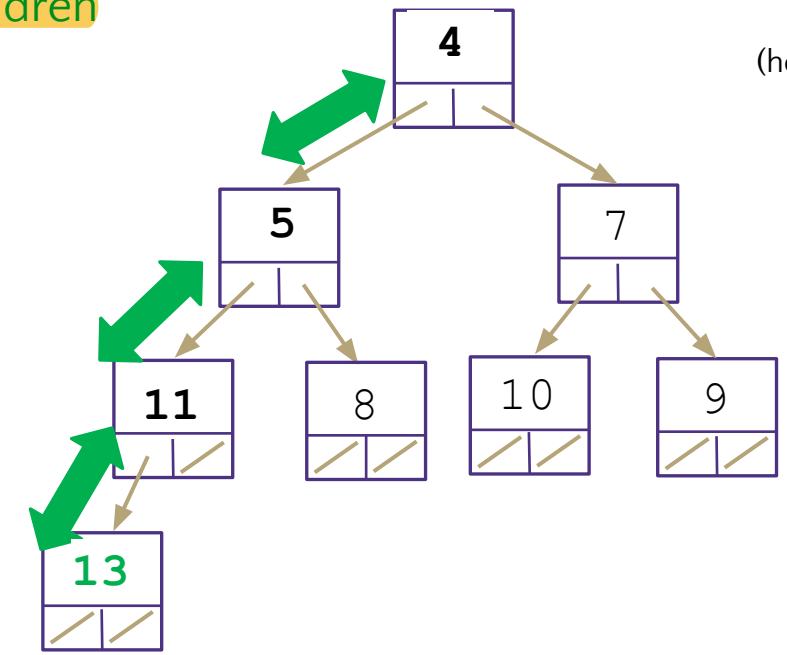
Structure invariant restored

Heap invariant broken

# Implementing removeMin() – percolateDown

1. Return min
2. Replace with bottom level right-most node
3. percolateDown()

   Recursively swap parent with **smallest** child until parent is smaller than both children (or we're at a leaf).




Structure invariant restored
Heap invariant restored

What's the worst-case running time?

Have to:

- Find last element
- Move it to top spot
- Swap until invariant restored

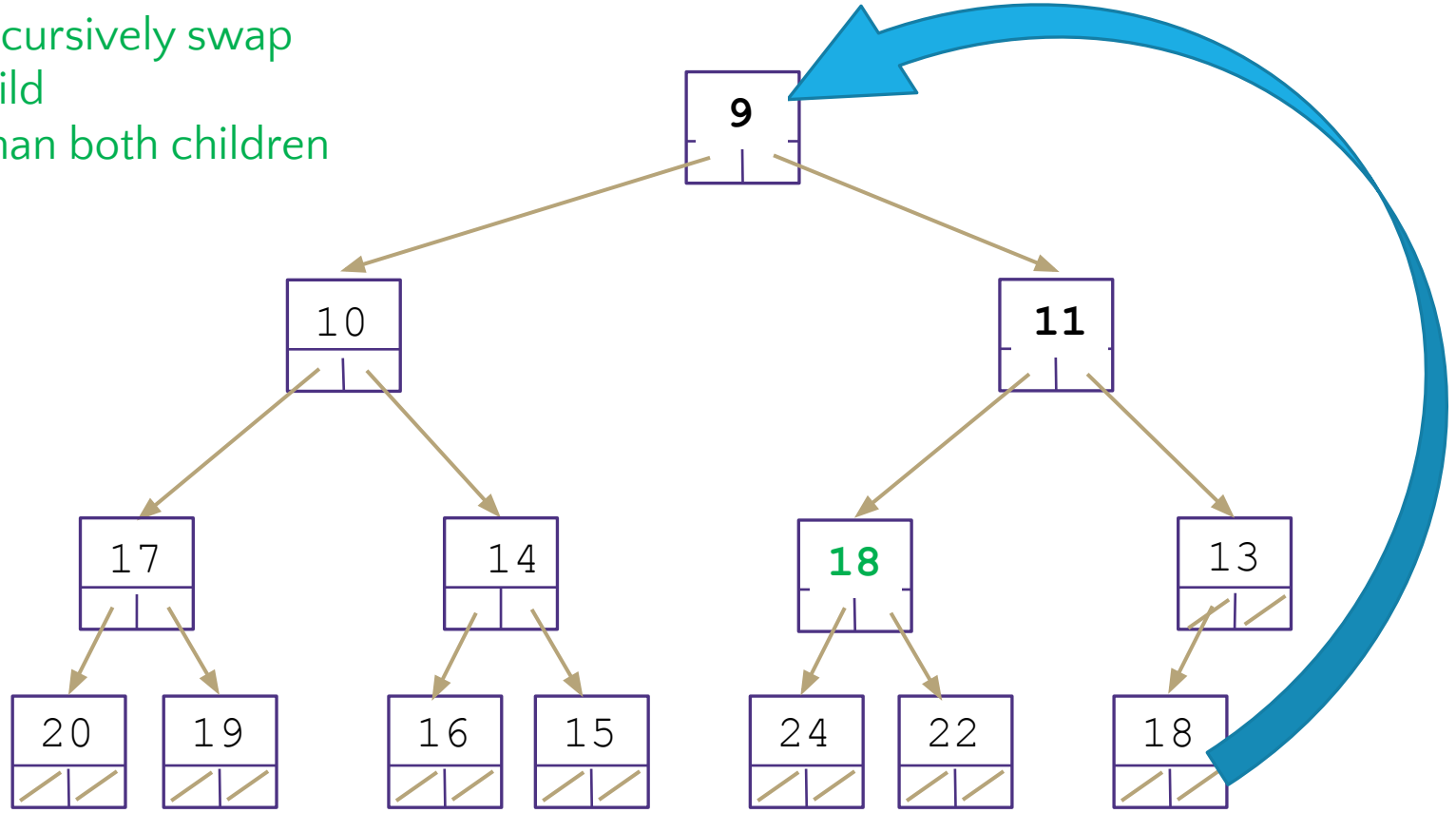
(how many times do we have to swap?)

This is why we want to keep the height of the tree small! The height of these tree structures (BST, AVL, heaps) directly correlates with the worst case runtimes

This is a big idea! (height of all these tree DS correlates w worst case runtimes – we want to design our trees to have reasonably small height!)

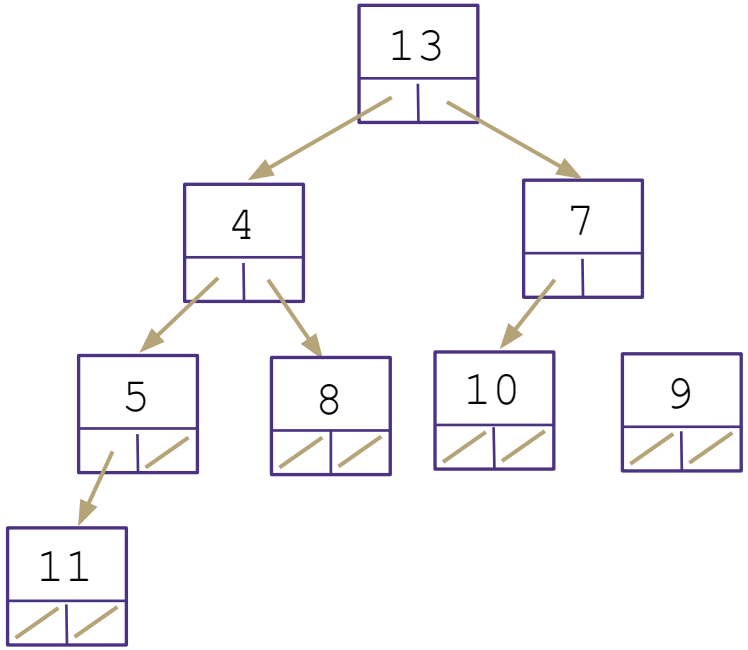
# *Practice:* removeMin()

1.) Remove min node
2.) replace with bottom level right-most node
3.) percolateDown – Recursively swap parent with **smallest** child
until parent is smaller than both children (or we're at a leaf).



# percolateDown()

Why does `percolateDown` swap with the smallest child instead of just any child?




If we swap 13 and 7, the heap invariant isn't restored!

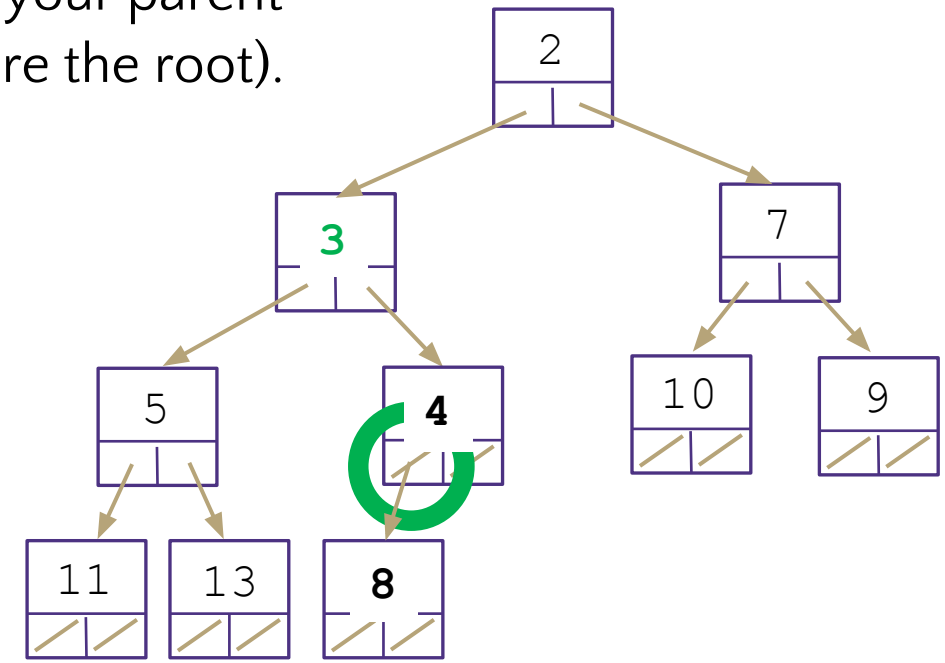
7 is greater than 4 (it's not the smallest child!) so it will violate the invariant.

# Implementing add()

add() Algorithm:

1. Insert a node on the bottom level that ensure no gaps
2. Fix heap invariant by percolate **UP**

i.e. swap with parent, until your parent is smaller than you (or you're the root).

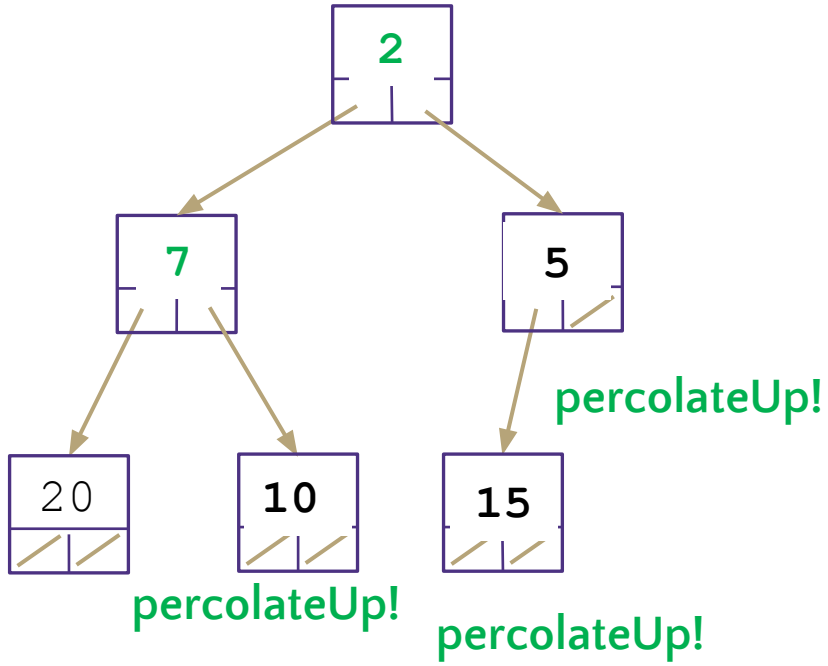



Worst case runtime is similar to `removeMin` and `percolateDown` – might have to do log(n) swaps, so the worst-case runtime is Theta(log(n))

# *Practice:* Building a minHeap

Construct a Min Binary Heap by adding the following values in this order:

- 5, 10, 15, 20, 7, 2




add() Algorithm:

1. Insert a node on the bottom level that ensure no gaps
2. Fix heap invariant by percolate **UP**

i.e. swap with parent, until your parent is smaller than you (or you're the root).

**Min Binary Heap Invariants**

1. Binary Tree – each node has at most 2 children
2. Min Heap – each node's children are larger than itself
3. Level Complete – new nodes are added from left to right completely filling each level before creating a new one

# minHeap runtimes

removeMin():

- remove root node
- find last node in tree and swap to top level
- percolate down to fix heap invariant

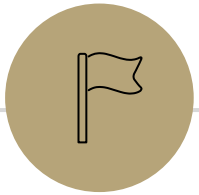
add()

- insert new node into next available spot
- percolate up to fix heap invariant

Finding the last node/next available spot is the hard part.
You can do it in $\Theta(\log n)$ time on complete trees, with some extra class variants

But it's NOT fun

And there's a much better way (that we'll talk about Wednesday)!



# That’s all!