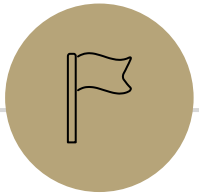
Priority Queues

CSE 332 Spring 2025
Lecture 4

Logistics

	Monday	Tuesday	Wednesday	Thursday	Friday
This Week	TODAY Exercise 0 due Ex 2 available				Exercise 1 due Ex 3 available
Next Week	Ex 2 due				Ex 3 due

Amortized Analysis slides from last lecture will be covered on Wed or Fri. Want to make sure we get through heaps today so you can start on Ex 2.



A new ADT: Priority Queues

A New ADT

Our previous worklists (stacks, queues, etc.) all choose the next element based on the order they were inserted.

That's not always a good idea.

Emergency rooms aren't first-come-first-served.

Sometimes our objects come with a **priority**, that tells us what we need to do next.

An ADT that can handle a line with priorities is a **priority queue.**

Priority Queue ADT

Min Priority Queue ADT

state

Set of comparable values
- Ordered based on "priority"

behavior

insert(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.

Uses:

- Operating System
- Well-designed printers
- Some Compression Schemes (google Huffman Codes)
- Sorting
- Graph algorithms

Keys and Values

In most applications, you'll have two things

A priority and an object with that priority

- On a printer, the priority of the file and the file itself
- At an ER, the priority of the patient, and the patient themselves

We're going to ignore the object and only focus on the priority for the slides (makes it easier to look at), don't forget the other object when you implement things though!

- On the slides, we'll usually use `ints` for priorities
- All you need are comparable values (`doubles` are fine, as would be non-numbers if they can be ordered)
- On Ex2, objects comparable interface on objects give priorities.

Implementing Priority Queues: Take I

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

	Insert	removeMin
Unsorted Array	$\Theta(1)$	$\Theta(n)$
Unsorted Linked List	$\Theta(1)$	$\Theta(n)$
Sorted Linked List	$\Theta(n)$	$\Theta(1)$
Sorted Circular Array	$\Theta(n)$	$\Theta(1)$
Binary Search Tree	$\Theta(\text{height})$	$\Theta(\text{height})$

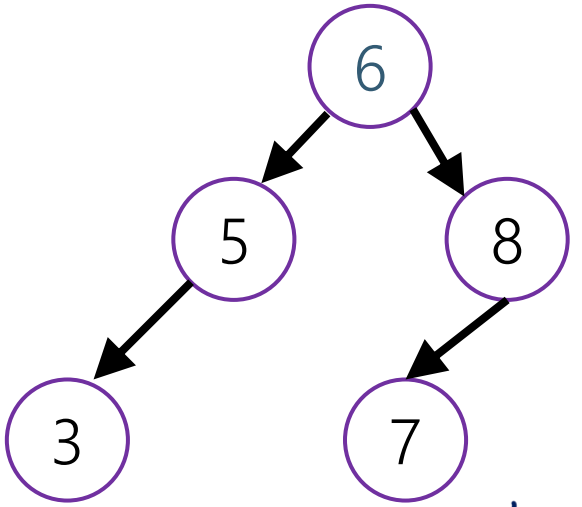
For Array implementations, assume that the array is not yet full.
Other than this assumption, do **worst case** analysis. (amortized bounds will match).

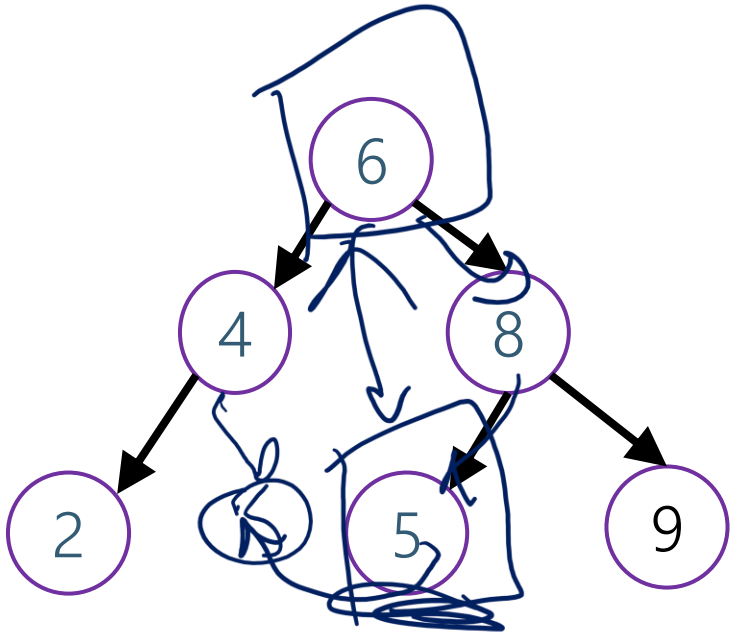
Review: Binary Search Trees

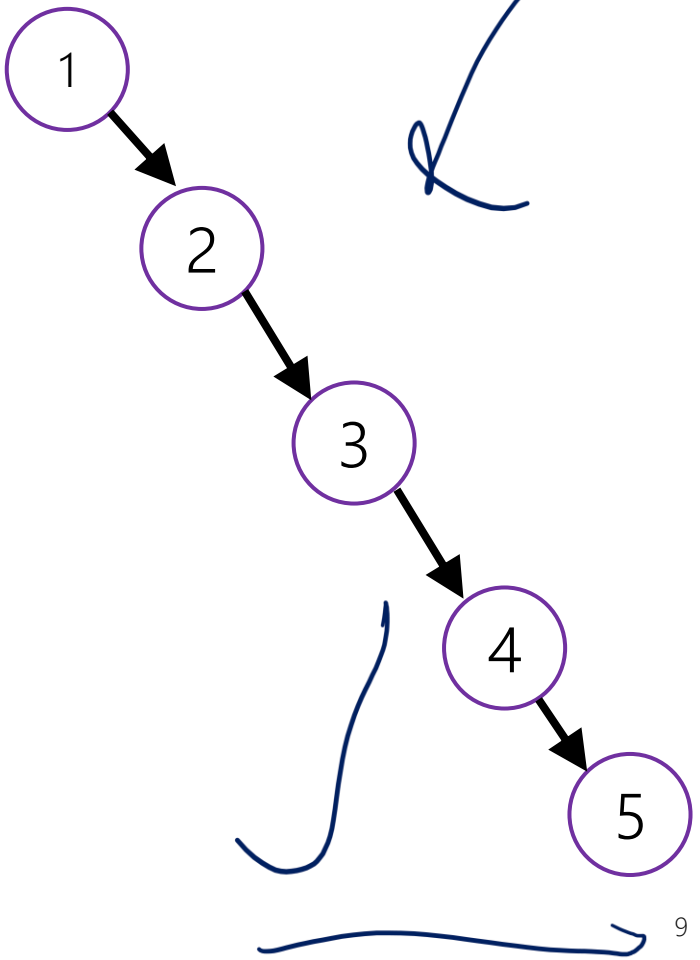
A BST is:

1. A binary tree
2. For each node, everything in its left subtree is smaller than it and everything in its right subtree is larger than it.

Are These BSTs?







Implementing Priority Queues: Take I

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

	Insert	removeMin
Unsorted Array	$\Theta(1)$	$\Theta(n)$
Unsorted Linked List	$\Theta(1)$	$\Theta(n)$
Sorted Linked List	$\Theta(n)$	$\Theta(1)$
Sorted Circular Array	$\Theta(n)$	$\Theta(1)$
Binary Search Tree		

For Array implementations, assume that the array is not yet full.
Other than this assumption, do **worst case** analysis. (amortized bounds will match).

Implementing Priority Queues: Take I

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

	Insert	removeMin
Unsorted Array	$\Theta(1)$	$\Theta(n)$
Unsorted Linked List	$\Theta(1)$	$\Theta(n)$
Sorted Linked List	$\Theta(n)$	$\Theta(1)$
Sorted Circular Array	$\Theta(n)$	$\Theta(1)$
Binary Search Tree	$\Theta(\text{height})$	$\Theta(\text{height})$

For Array implementations, assume that the array is not yet full.
Other than this assumption, do **worst case** analysis. (amortized bounds will match).

Implementing Priority Queues: Take II

BSTs have really bad behavior in the **worst** case, but is it actually a common problem?

Worst case, both of those operations are $\Theta(n)$.

But often BSTs have height $\log(n)$

Can we somehow get that behavior in the **worst case** for priority queues?

BST Properties

A BST is:

1. A binary tree

2. For each node, everything in its left subtree is smaller than it and everything in its right subtree is larger than it.

Point 2 is what causes the really bad behavior in the worst case.

We probably don't want exactly that requirement for implementing a priority queue.

Maybe we can explicitly enforce that we don't get a degenerate tree.

Binary Heaps

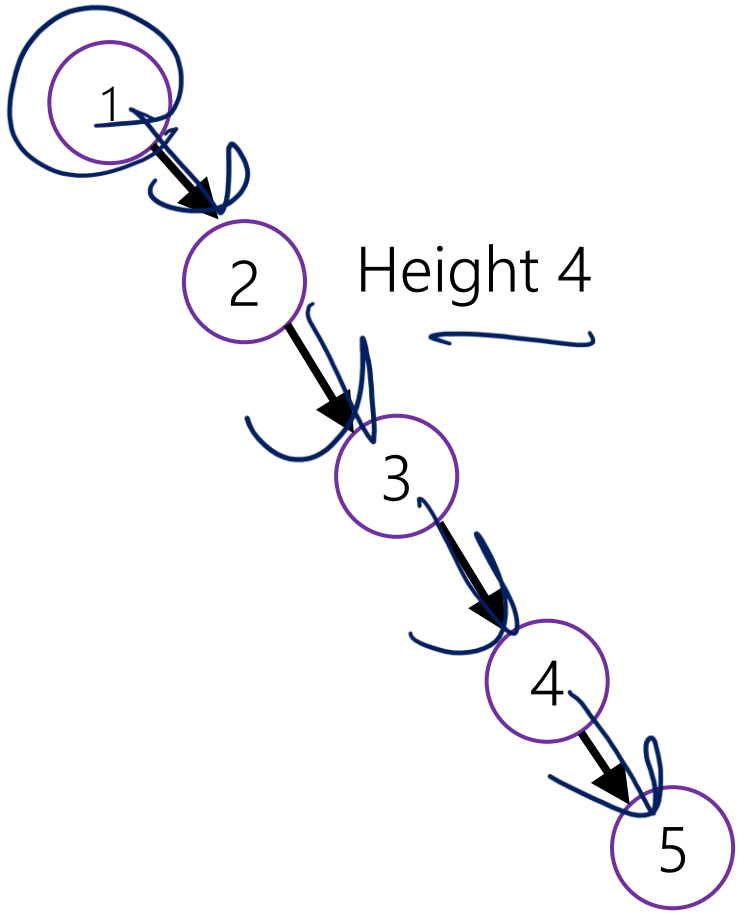
A Binary **Min**-Heap is

1. A Binary Tree
2. Every node is less than or equal to all of its children
   - In particular, the smallest element must be the root!
3. The tree is **complete**
   - Every level of the tree is completely filled, except possibly the last level, which is filled from left to right.
   - Thus, no Degenerate trees!

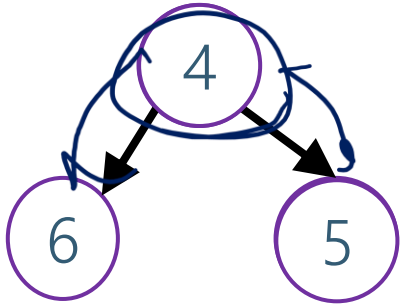
Called **min**-heap, because most important element has smallest priority. A **max**-heap follows the same principles but puts bigger elements on top.

Tree Words

Height – the number of edges on the longest path from the root to a leaf.



Height 4

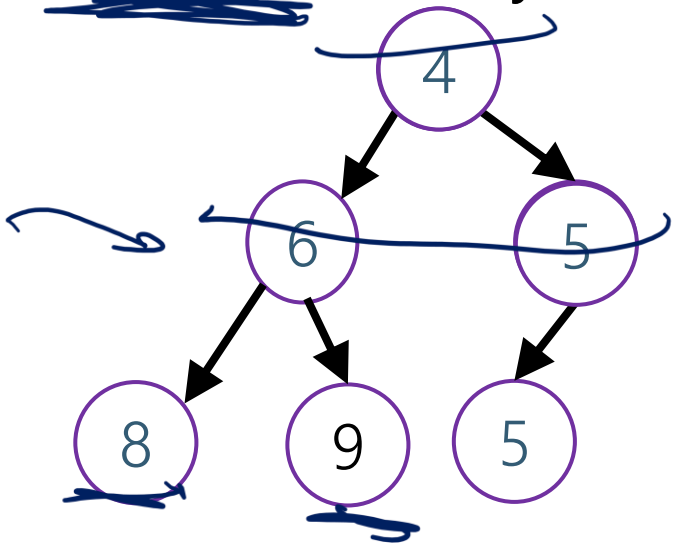


Height 1

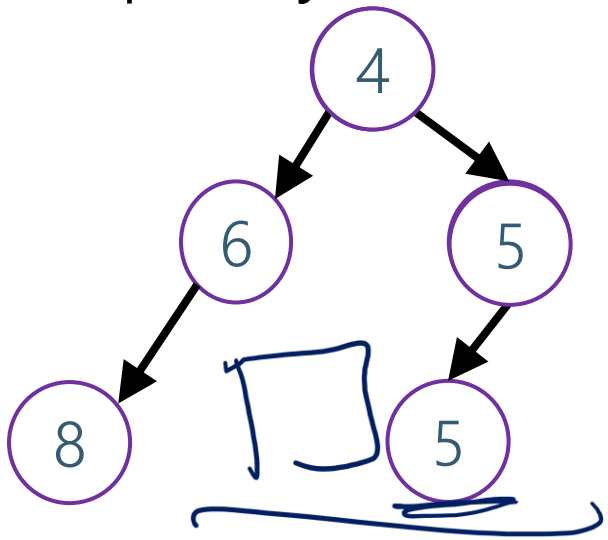
Tree Words

Complete – every row is completely filled, except possibly the last row, which is filled from left to right.

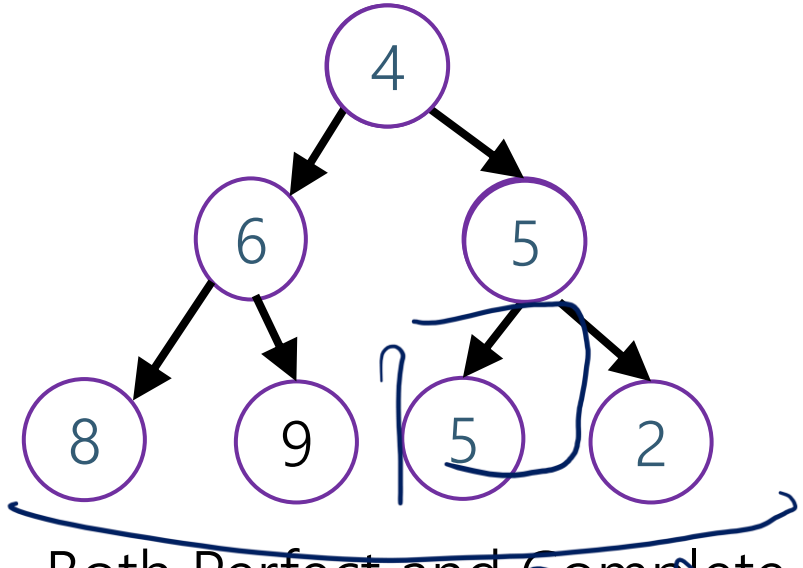
Perfect – every row is completely filled



Complete, but not perfect



Neither



Both Perfect and Complete

Binary Heaps

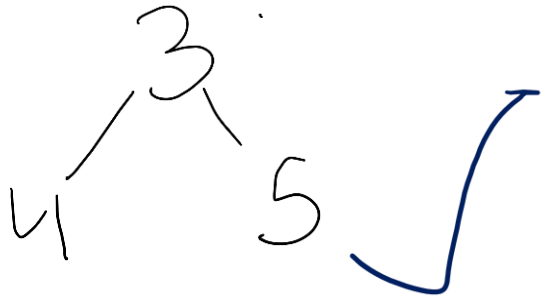
A Binary Min-Heap is

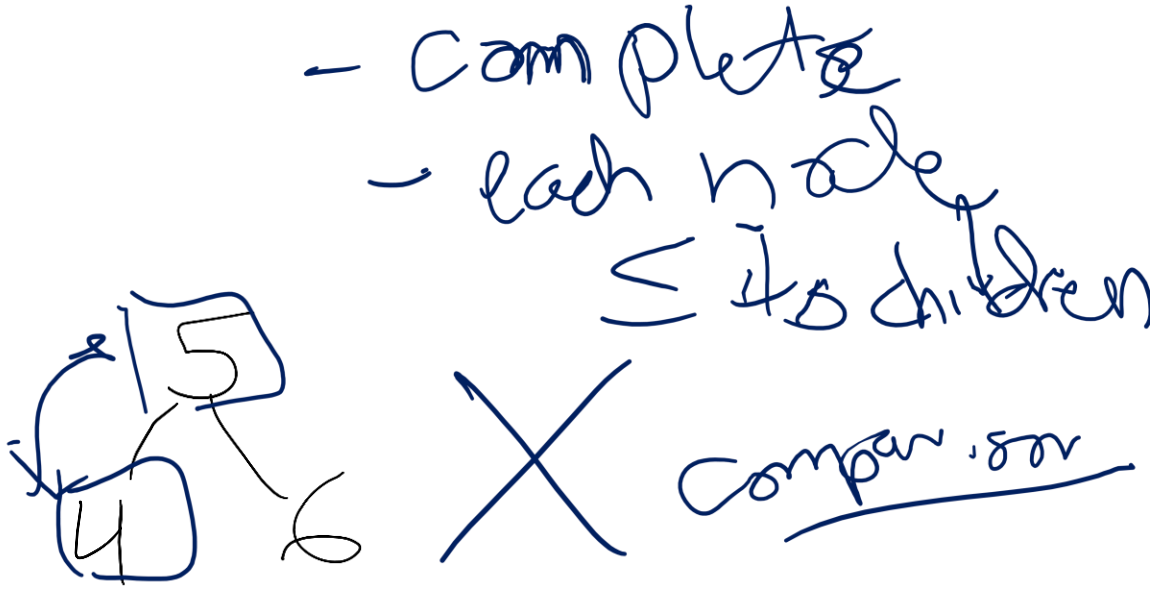
1. A Binary Tree

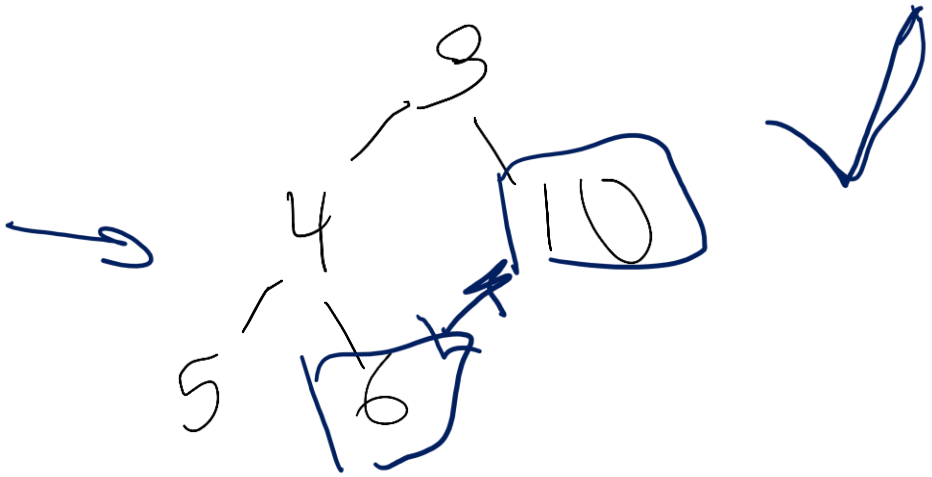
2. Every node is less than or equal to all of its children
   - In particular, the smallest element must be the root!

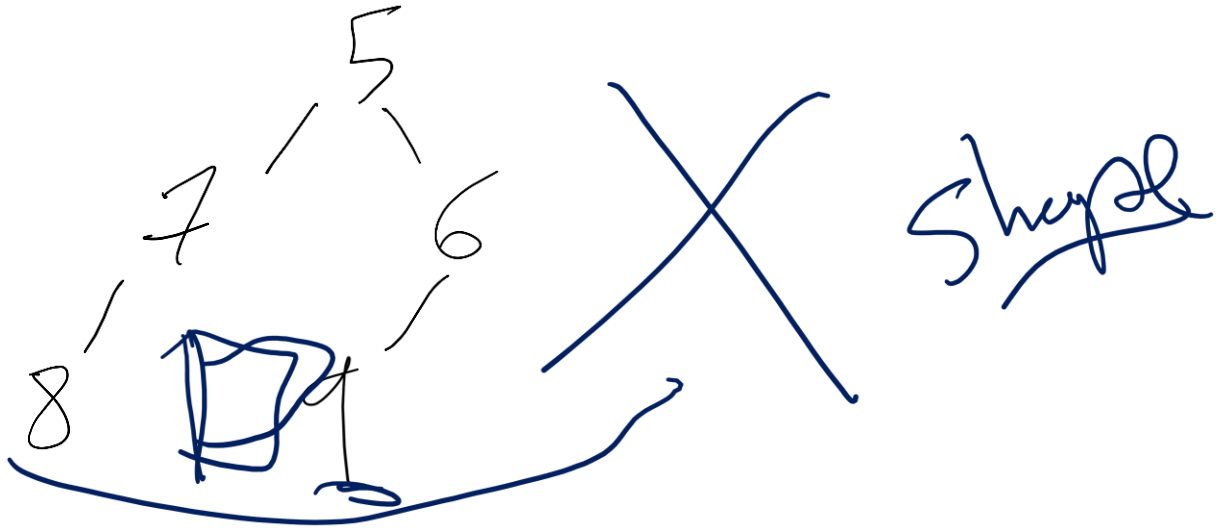
3. The tree is **complete**
   - Every level of the tree is completely filled, except possibly the last level, which is filled from left to right.
   - Thus, no degenerate trees!

Are These Min-Heaps





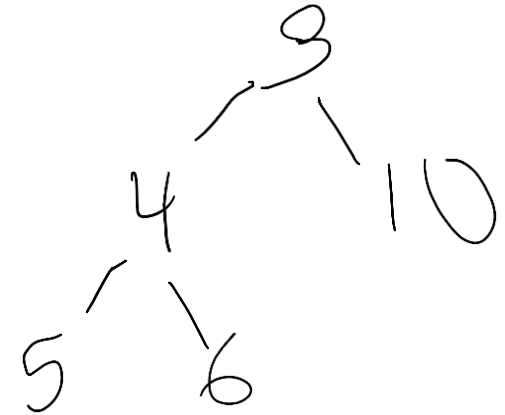




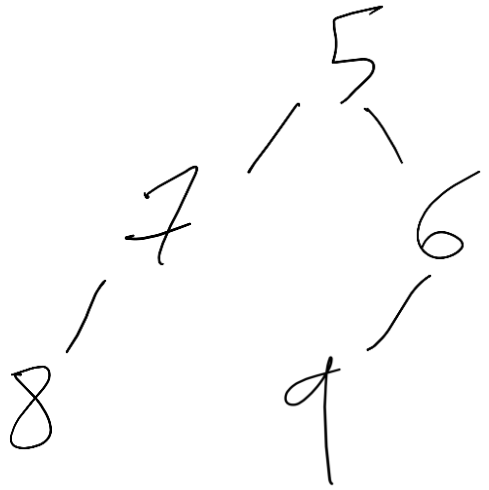
Are These Min-Heaps

Valid heap!

5 has a smaller child.



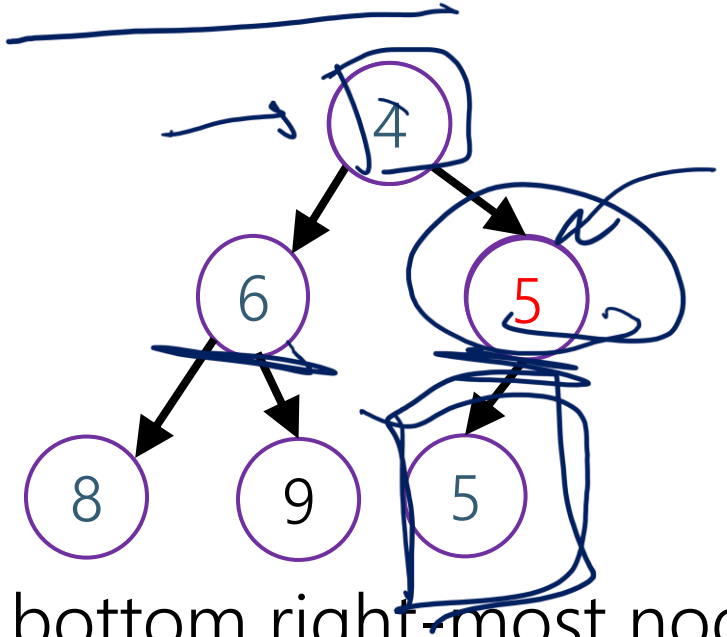
Valid heap!
5 is smaller than 10, but 10 isn't an ancestor so not a violation.



Wrong shape!

Implementing Heaps

Let's start with **removeMin**.



Idea: take the bottom right-most node and use it to plug the hole

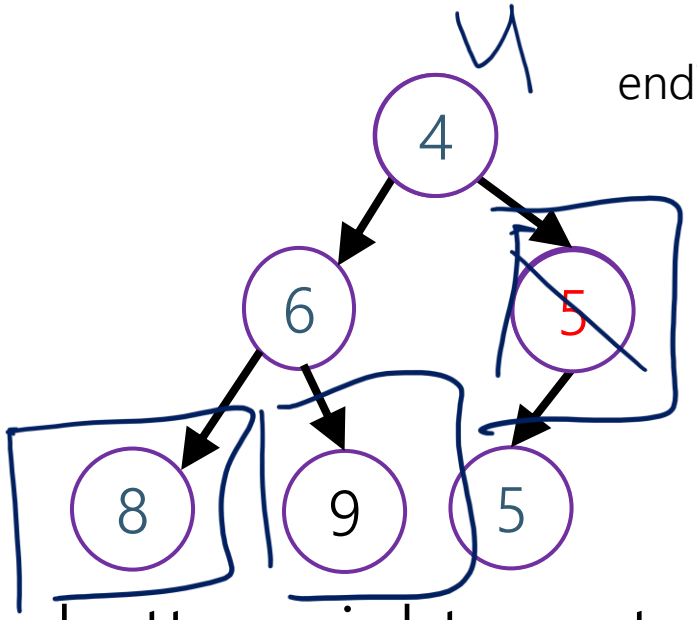
Shape is correct now

But that value might be to big. We need to "percolate it down"

Implementing Heaps

Let's start with **removeMin**.

```
percolateDown(curr)
  while(curr.value > curr.left.value or curr.value > curr.right.value)
        swap curr with min of left and right
  endWhile
```



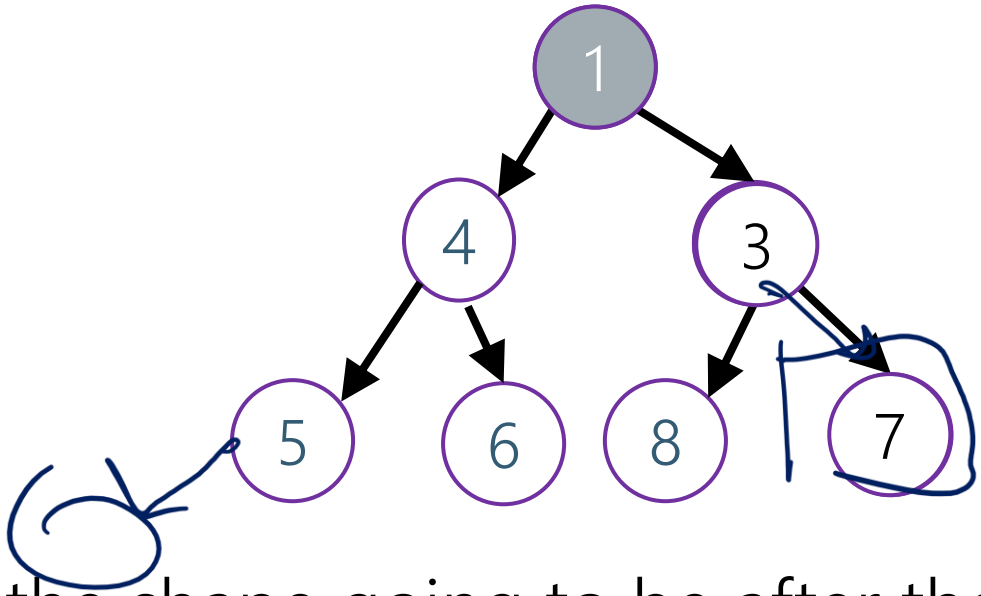
Idea: take the bottom right-most node and use it to plug the hole

Shape is correct now

But that value might be to big. We need to "percolate it down"

Implementing Heaps

Insertion



What is the shape going to be after the insertion?

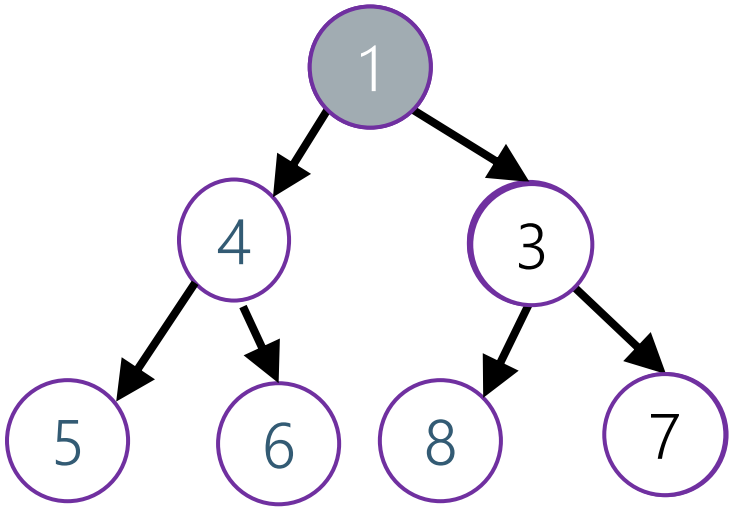
Again, plug the hole first.

Might violate the heap property. Percolate it up

Implementing Heaps

```
percolateUp(curr)
  while(curr.value < curr.parent.value)
        swap curr and parent
  endWhile
```

Insertion



What is the shape going to be after the insertion?

Again, plug the hole first.

Might violate the heap property. Percolate it up

Summary

1. When adding/removing items, “plug the hole” to maintain the shape property (or add at end if no hole).

2. Whatever was moved might be in the wrong spot. percolateUp or percolateDown as appropriate
   - i.e. move one step in the right direction via a swap.

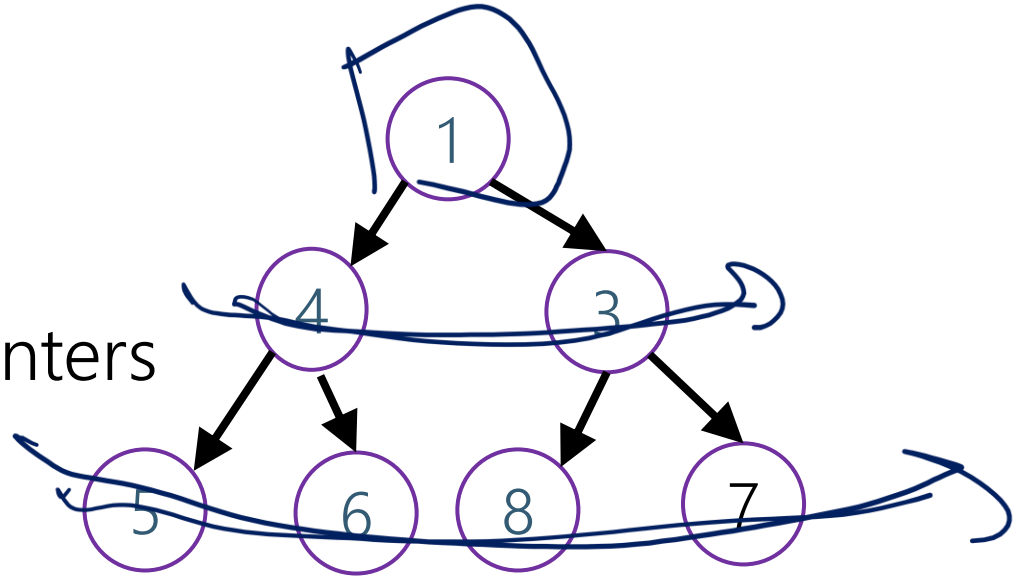
An Optimization

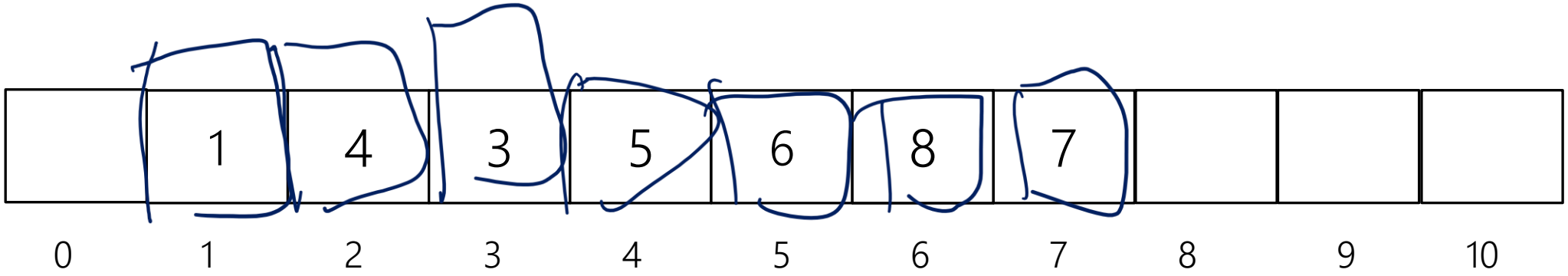
Pointers are annoying.

They're also slow.

Shape is simple—we don't need pointers

We can use an array instead.





An Optimization

On Exercise 2, you'll index from 0 rather than 1. Details are different!

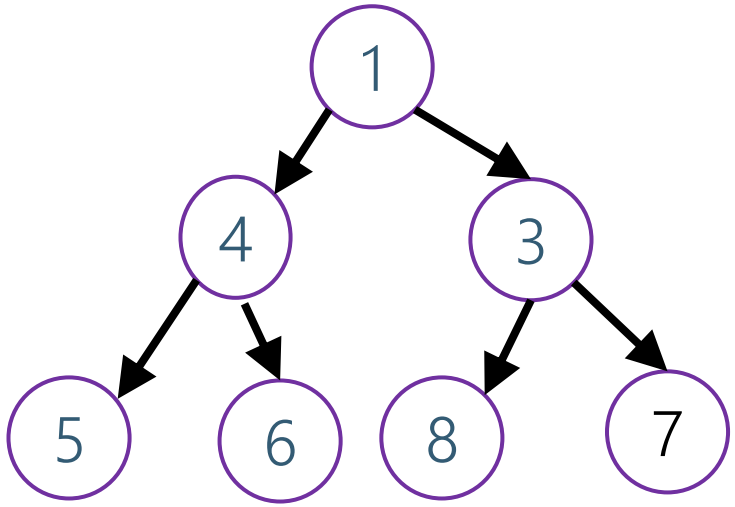
If I'm at index i, what is the index of:

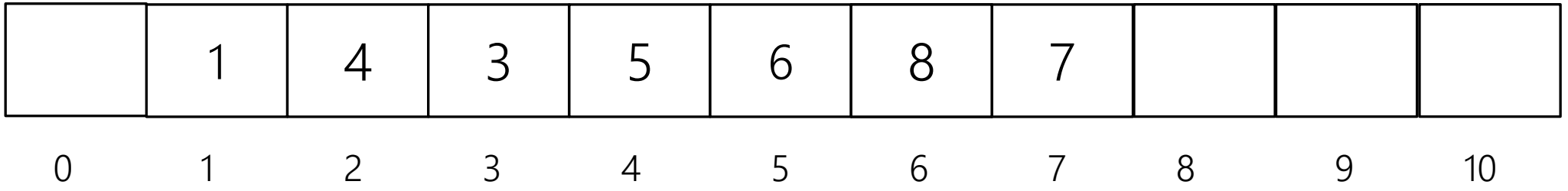
My left child, right child and parent?

My left child:

My right child:

My parent:





An Optimization

On Exercise 2, you'll index from 0 rather than 1. Details are different!

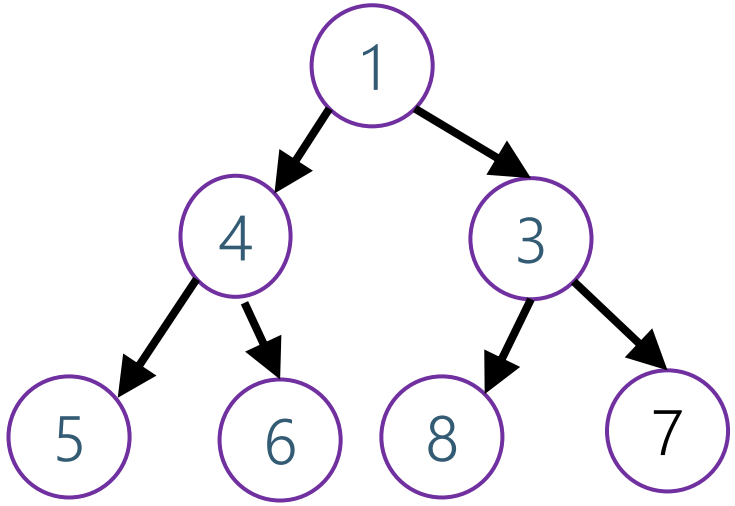
If I'm at index i, what is the index of:

My left child, right child and parent?

My left child: $2i$

My right child: $2i + 1$

My parent: $\left\lfloor \frac{i}{2} \right\rfloor$



	1	4	3	5	6	8	7			
0	1	2	3	4	5	6	7	8	9	10

Running times?

Worst case: looks like $O(h)$ where h is the height of the tree.

That's true, but it's not a good answer. To understand it, your user needs to understand how you've implemented your priority queue. They should only need to know how many things they put in.

Let's find a formula for h in terms of n.

Heights of Perfect Trees

How many nodes are there in level i of a perfect binary tree?

Heights of Perfect Trees

How many nodes are there in level i of a perfect binary tree?

On the whiteboard we derived that the number of nodes on level i of a binary tree was 2^i.

Thus the total number of nodes in a perfect binary tree of height h is

$\sum_{i=0}^{h} 2^i = 2^{h+1} - 1$.

So if we have n nodes in a perfect tree, we can use the formula

$n = 2^{h+1} - 1$ to conclude that $h = O(\log n)$,so

A perfect tree with n nodes has height $O(\log n)$.

A similar argument can show the same statement for complete trees.

More Operations

On Ex 2, you'll do more things with heaps!

IncreaseKey(element,priority) Given a pointer to an element of the heap and a new, larger priority, update that object's priority.

DecreaseKey(element,priority) Given a pointer to an element of the heap and a new, smaller priority, update that object's priority.

Remove(element) Given a pointer to an element of the heap, remove that element.

Needing a pointer to the element is a bit unusual – it makes maintaining the data structure more complicated.

- Heap doesn't have BST property—it's hard to find things in there!!

Some Exercise 2 Notes

You know everything you need to do Exercise 2. Some minor differences from lecture:

You'll implement both a min-heap and a max-heap.

Some of the method names are different

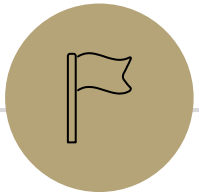
- Extract() instead of removeMin()
- One updatePriority() instead of separate IncreaseKey(), DecreaseKey()

Index from 0 instead of 1.

updatePriority() and some other methods, require pointers to the location in the heap. You'll use a (given, java built-in) hashmap to do that. Be sure you're keeping that map up-to-date!

Remember we omit edge cases in lecture sample code.

- (e.g., what if percolateUp gets all the way to root?)



More Priority Queue Operations

Even More Operations

BuildHeap(elements $e_1, \ldots, e_n$) – Given n elements, create a heap containing exactly those n elements.

Try 1: Just call insert n times.

Worst case running time?

n calls, each worst case $\Theta(\log n)$. So it's $\Theta(n \log n)$ right?

That proof isn't valid.

There are at least two distinct problems (bugs or gaps that need much more explanation), can you find them?

Two Issues

Try 1: Just call insert n times.

Worst case running time?

n calls, each worst case $\Theta(\log n)$. So it's $\Theta(n \log n)$ right?

It's not clear that you can make each insert, one right after the other, hit the worst-case behavior.

- Imagine you said "when operating a [standard] Queue, inserting takes $\Theta(n)$ time in the worst case. So n consecutive inserts take $\Theta(n^2)$ time." That's false!

n changes as you do the insertions!

Fixing the Bugs/Gaps

If you put O in for Θ the proof would work as written.

- Remember O is an upper-bound.
- "The worst thing right now $\leq$ the worst thing ever"
- $O(h) \leq O(\log n)$ where h is current height, and n is final height.

It's not clear that you can make each insert, one right after the other, hit the worst-case behavior.

- You can force this with a heap! Inserting elements in decreasing order will mean every inserted element goes at the leaf location and needs to percolateUp to the root (since minimum needs to be at root).

The size isn't n the whole time.

- But big-O doesn't care about constant factors. And half the time, it's $n/2$ or more.

BuildHeap Running Time (again)

Let's try once more.

Saying the worst case was decreasing order was a good start.

What are the actual running times?

It's $\Theta(h)$, where h is the current height.

But most nodes are inserted in the last two levels of the tree.

- For most nodes, h is $\Theta(\log n)$. (starting from the second half, h is at least $\log\left(\frac{n}{2}\right) = \log(n) - 1 \in \Theta(\log n)$

So the number of operations is at least

$\frac{n}{2} \cdot \Omega(\log n) = \Omega(n \log n)$.

Fixed Proof (Sketch only)

Claim: Inserting n times has a worst case running time of $\Theta(n \log n)$

Proof:

Each of the n calls, has worst case $O(\log n)$. So it's certainly $O(n \log n)$.

For an Omega bound, note that for most elements the height of the data structure is already close to the final height. Considering only the last $n/2$ operations, inserting elements in decreasing order will produce h swaps, which gives $\frac{n}{2} \cdot h \leq \frac{n}{2}(\log(n) - 1) \in \Omega(n \log n)$ swaps, and therefore that many steps.

Thus our running time is $\Theta(n \log n)$.

Where Were We?

We were trying to design an algorithm for:

BuildHeap(elements $e_1, \ldots, e_n$ **)** – Given n elements, create a heap containing exactly those n elements.

Just inserting leads to a $\Theta(n \log n)$ algorithm in the worst case.

Can we do better?

Can We Do Better?

What's causing the n insert strategy to take so long?

Most nodes are near the bottom, and we can make them all go all the way up.

What if instead we tried to percolate things down?

Seems like it might be faster

- The bottom two levels of the tree have $\Omega(n)$ nodes, the top two have 3 nodes.