More Heaps

CSE 332 Spring 2025
Lecture 5

Logistics

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

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

Binary Heaps

A Binary **Min**-Heap is

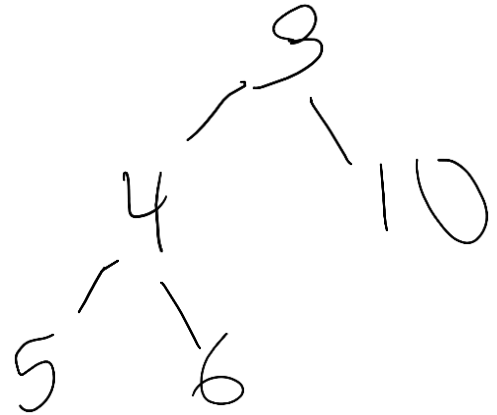
1. A Binary Tree
2. Every node is less than or equal to all of its children
   - In particular, the smallest element must be the root!
3. The tree is **complete**
   - Every level of the tree is completely filled, except possibly the last level, which is filled from left to right.
   - Thus, no Degenerate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

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

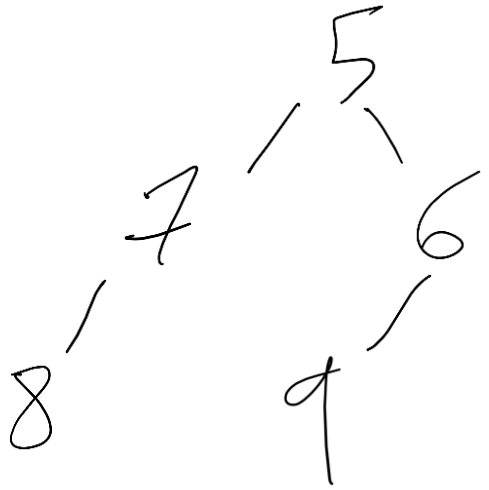
Are These Min-Heaps

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 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?

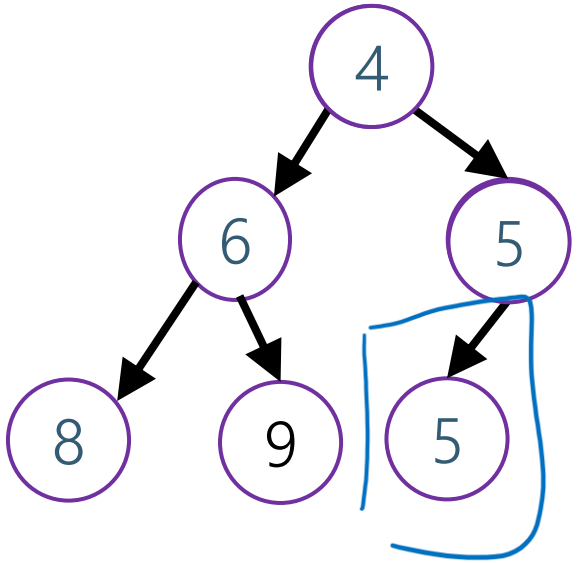
	Insert	removeMin
Unsorted Array	$\Theta(1)$	$\Theta(n)$
Unsorted Linked List	$\Theta(1)$	$\Theta(n)$
Sorted Linked List	$\Theta(n)$	$\Theta(1)$
Sorted Circular Array	$\Theta(n)$	$\Theta(1)$
Binary Search Tree	$\Theta(\text{height})$	$\Theta(\text{height})$

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

Implementing Heaps

Let's start with removeMin.

```
percolateDown(curr)
  while(curr.value > curr.left.value || 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

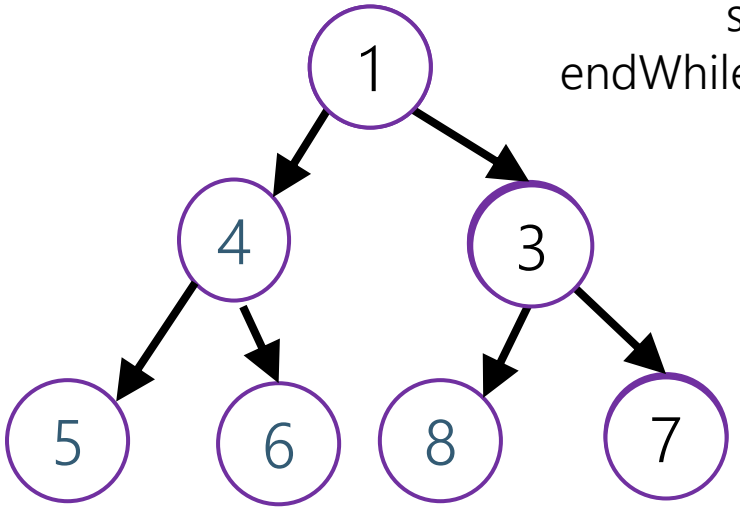
Shape is correct now

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

Implementing Heaps

Insertion

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



What is the shape going to be after the insertion?

Again, plug the hole first.

Might violate the heap property. Percolate it up

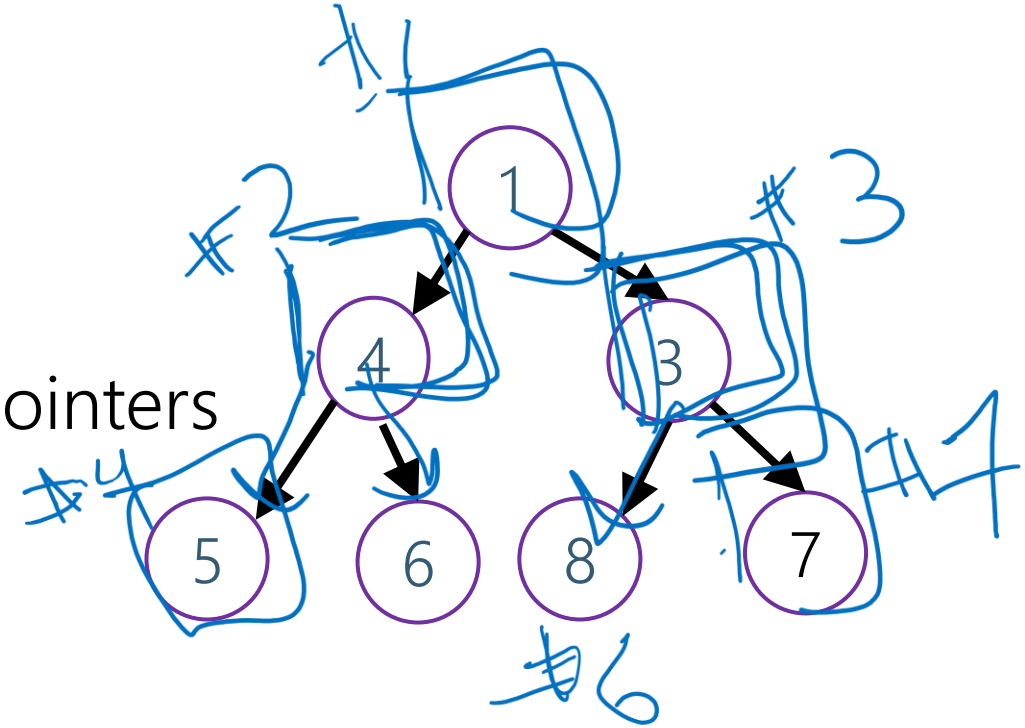
An Optimization

Pointers are annoying.

They're also slow.

Shape is simple—we don't need pointers

We can use an array instead.



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

An Optimization

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

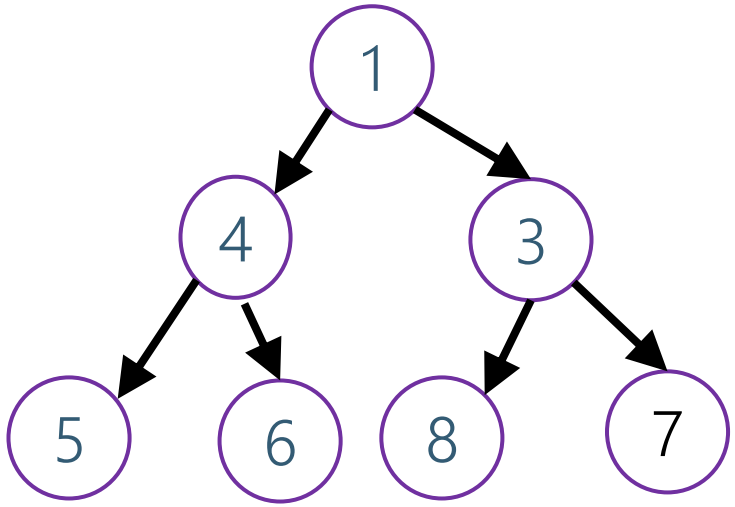
If I'm at index i, what is the index of:

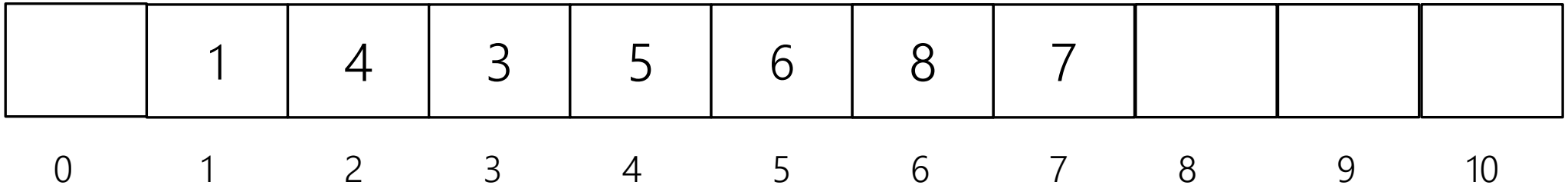
My left child, right child and parent?

My left child:

My right child:

My parent:





An Optimization

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

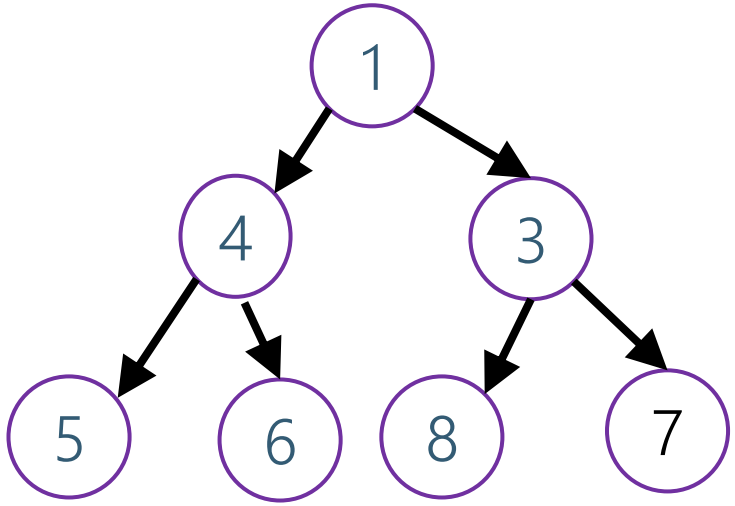
If I'm at index i, what is the index of:

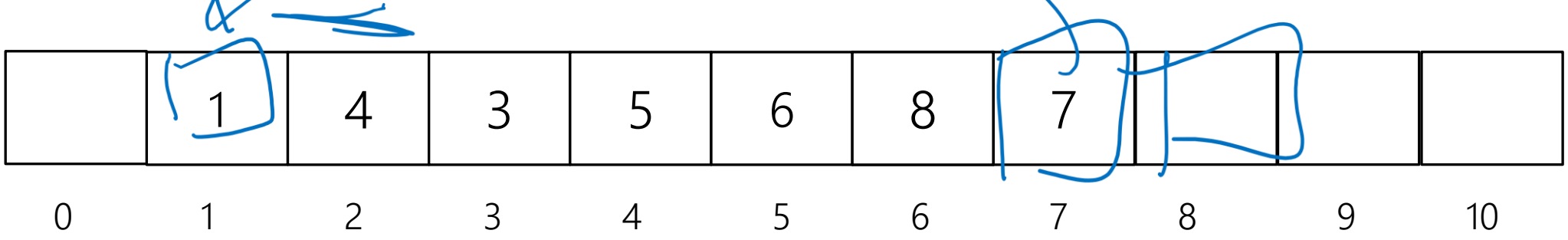
My left child, right child and parent?

My left child: $2i$

My right child: $2i + 1$

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





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?

$1 + 2 + 4 + 8 + \cdots + 2^k$

$2^{k+1} - 1$

n

$\log(n) \approx h$

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

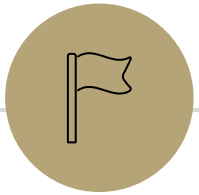
- Extract() instead of removeMin()
- One updatePriority() method instead of separate IncreaseKey(), DecreaseKey()

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.

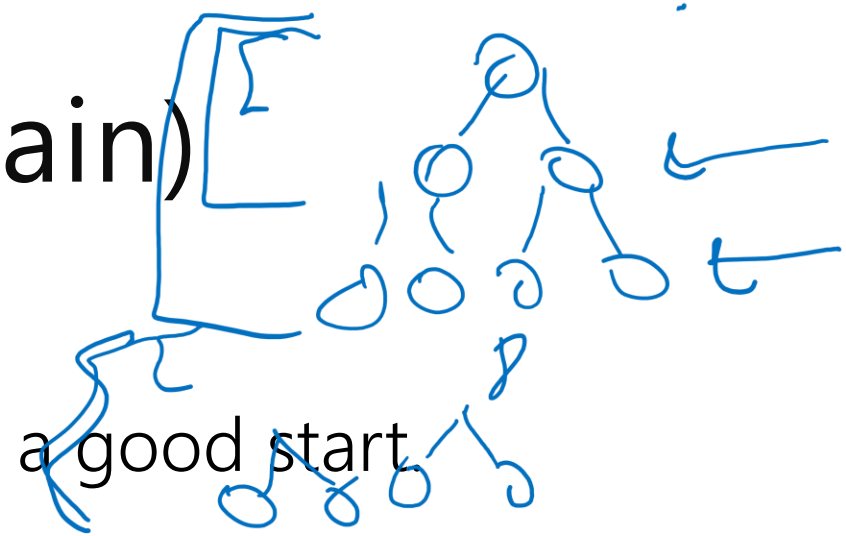
It's not clear that you can make each insert, one right after the other, hit the worst-case behavior.

- 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.

Is It Really Faster?

How long does it take to percolate everything down?

Each element at level i will do $h - i$ operations (up to some constant factor)

Total operations?

$$\sum_{i=0}^{h} 2^i(h-i) = \sum_{i=0}^{\log n} 2^i(\log n - i) =$$

$$\Theta(n)$$

Floyd's BuildHeap

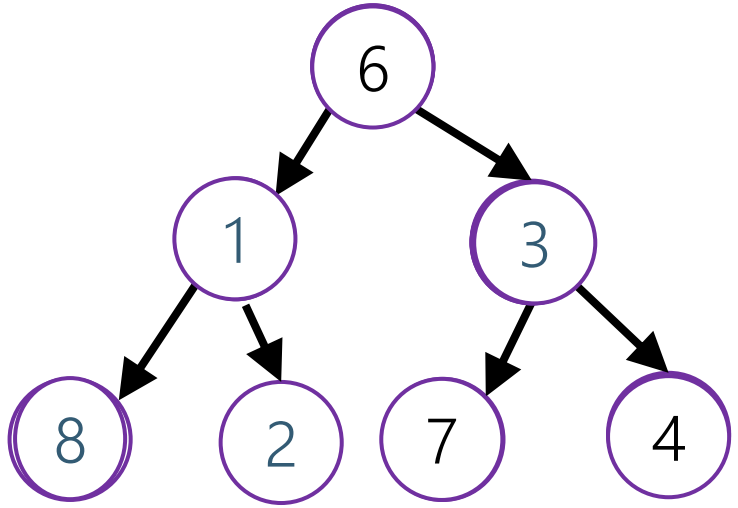
Ok, it's really faster.
But can we make it **work**?

It's not clear what order to call the percolateDown's in.

Should we start at the top or bottom?

Two Possibilities

```
void StartTop(){
    for(int i=0; i < n;i++){
        percolateDown(i)
    }
}
```

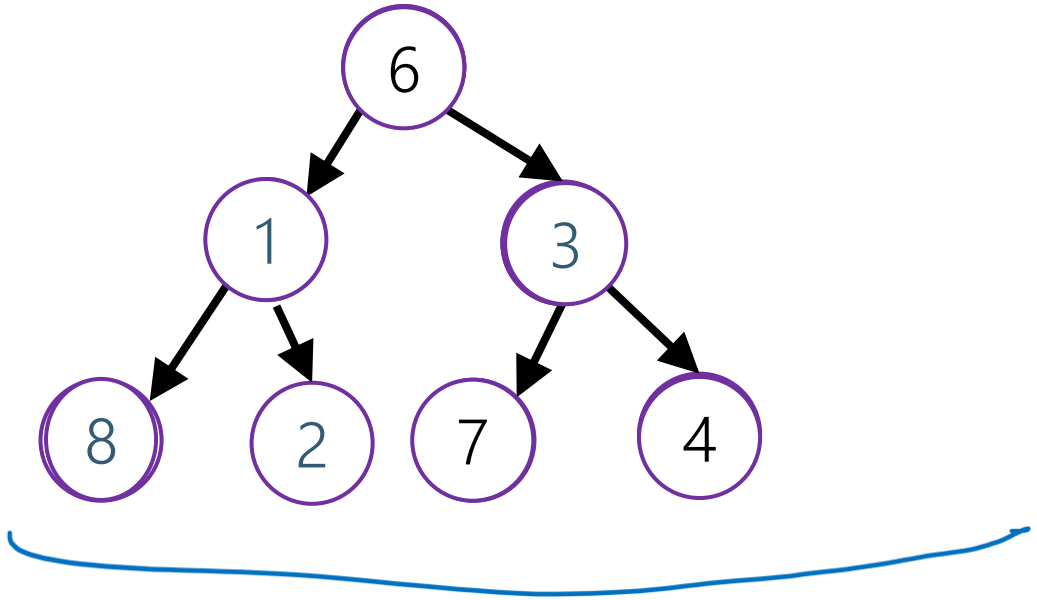


Try both of these on some trees. Is either of them possibly an ok algorithm?

```
void StartBottom(){
    for(int i=n; i >= 0;i--){
        percolateDown(i)
    }
}
```

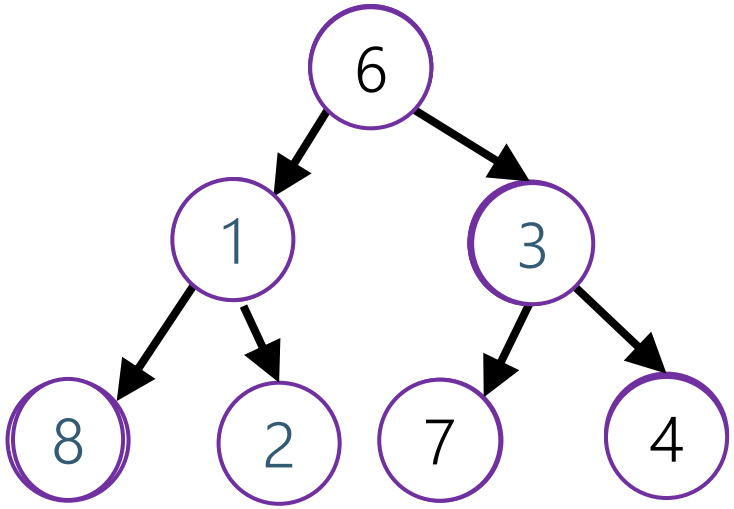
Only One Possiblity

If you run `StartTop()` on this heap, it will fail.



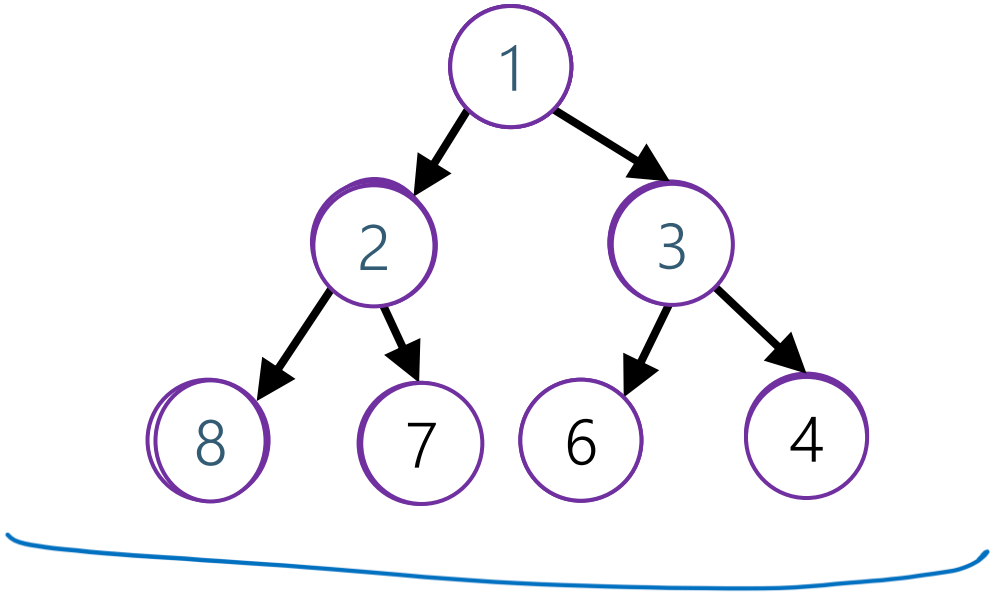
Only One Possiblity

If you run `StartTop()` on this heap, it will fail.



Only One Possibility

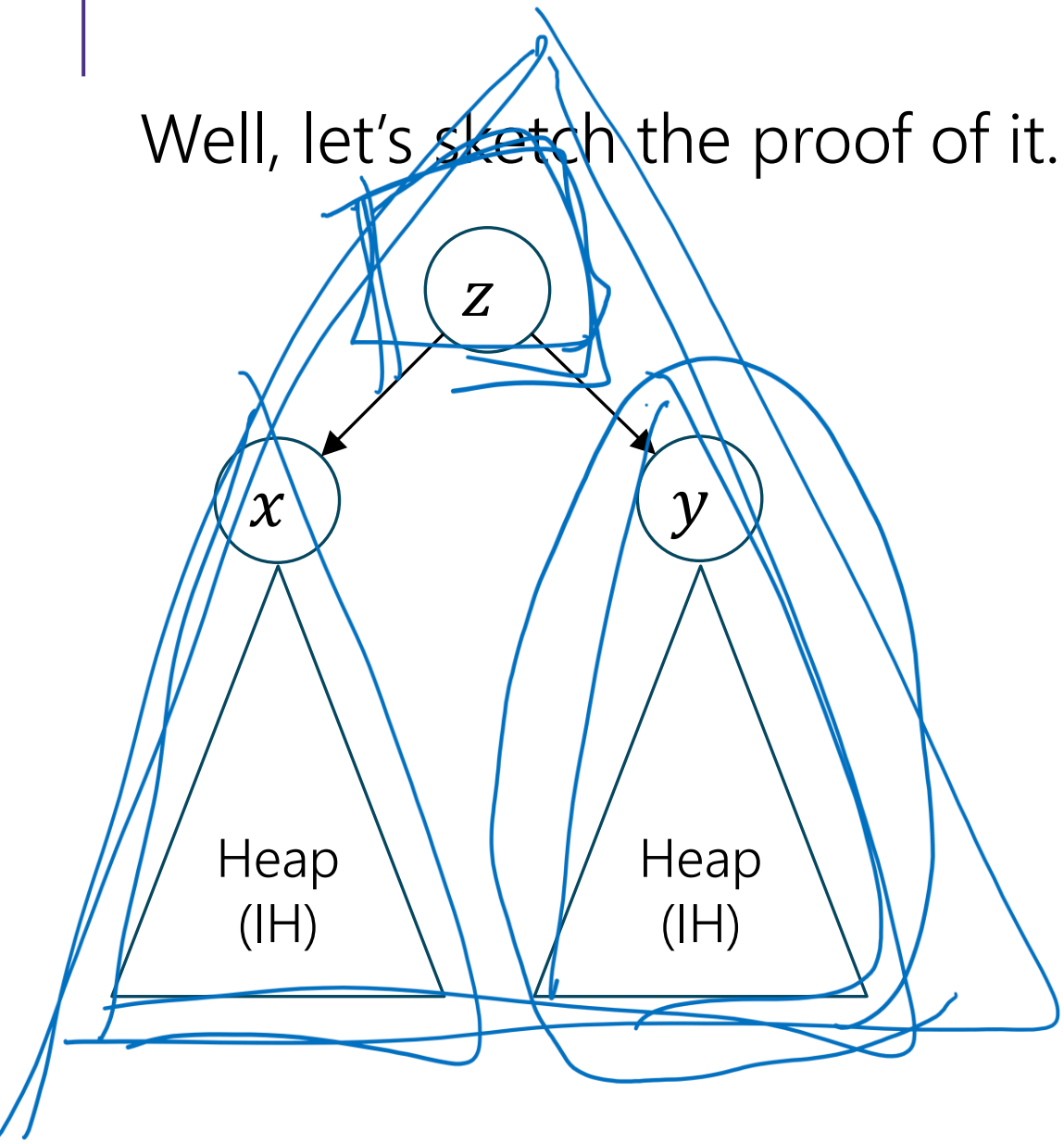
But `StartBottom()` seems to work.



Does it always work?

Let's Prove It!

Well, let's sketch the proof of it.



BC: Single node

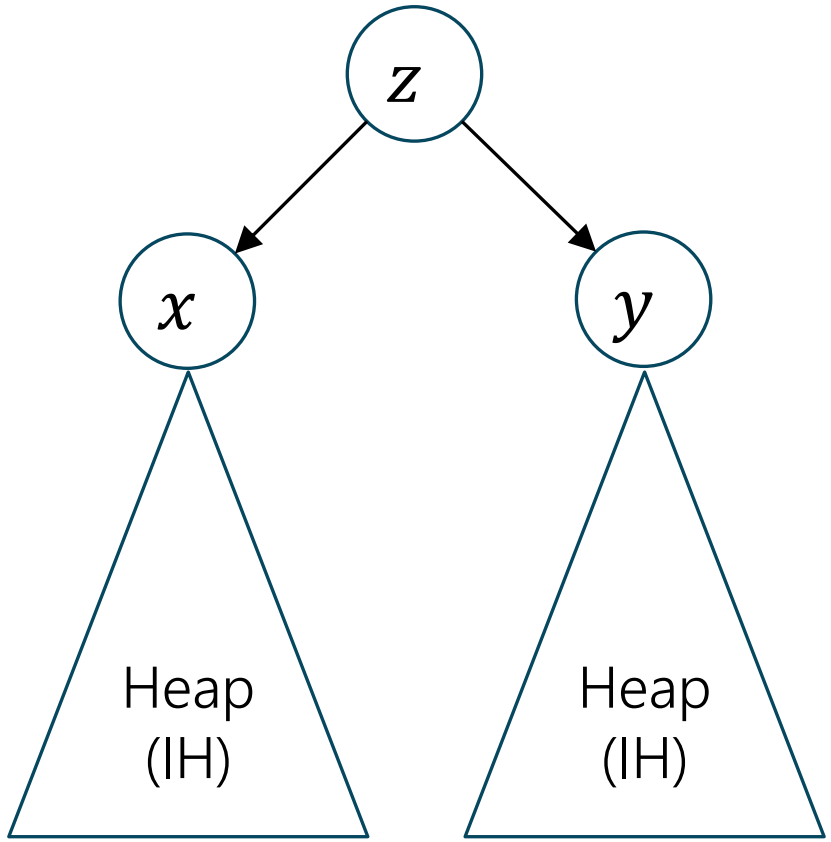
IH

IS

O(n)

Let's Prove It!

Well, let's sketch the proof of it.



Base Case: Leaf node is always a heap.
IH: Suppose x, y are roots of heaps
IS: We percolateDown, when you percolateDown and the children are already heaps, the whole thing is a heap!

More Operations

Let's 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.

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

Alternative Options

Binary heaps, implemented with an array are the default priorityQueue implementation (it's the only one we discuss here).

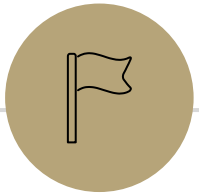
There are alternatives, but unusual use cases.

d-heaps, work like our heaps, but d children, not 2 children.

- Shallower which can be helpful for **very** large heaps for cache reasons.

Leftist heaps, skew heaps, binomial queues (see Weiss 6.6-6.8)

- Trees, but not nice enough for the array implementation trick (i.e., really pointers)
- Useful mainly if you frequently need a `merge` operation (given two priority queues, create a combined priority queue).
- Fun for-your-own-thinking, how would you merge binary heaps? (it won't be super fast, but think about how your strategy might change for different sizes)



Amortization

Amortization

How much does housing cost per day in Seattle?

Well, it depends on the day.

The day rent is due, it's $1800.

The other days of the month it's free.

Amortization

Amortization is an **accounting analysis**. It's a way to reflect the fact that even though the "first of the month" is very expensive, the reason that it's very expensive is that it's taking on responsibility for all the other days.

If we distributed the cost equally across the days, (because all days *should* be equally responsible), we "amortize" the cost.

Amortization

AMORTIZED

It costs $1800/month (which we pay once)

So the cost per day is $\frac{1800}{30} = 60$.

Good answer if the question is "what does my daily pay need to be to afford housing?"

UNAMORTIZED

On the first it costs $1800.

Every other day of the month it costs $0

Good answer if the question is "how much do I need to keep in my bank account so it doesn't get overdrawn?"

Amortization

What's the worst case for `enqueue` into an array-based queue?

- The running time is $O(n)$ when we need to resize, and $O(1)$ otherwise.

Is $O(n)$ a good description of the worst-case behavior?

Amortization

AMORTIZED

It takes $O(n)$ time to resize once, the next $n-1$ calls take $O(1)$ time each.

So the cost per operation is $\frac{O(n)+[n-1]O(1)}{n} = O(1)$

Good answer if the question is "what will happen when I do many insertions in a row?"

UNAMORTIZED

The resize will take $O(n)$ time. That's the worst thing that could happen.

Good answer if the question is "how long might one (unlucky) user need to wait on a single insertion?"

Amortization

The most common application of amortized bounds is for insertions/deletions and data structure resizing.

Let's see why we always do that doubling strategy.

How long in total does it take to do m insertions?

We might need to double a bunch, but the total resizing work is at most $O(m)$

And the regular insertions are at most $\mathrm{m} \cdot O(1) = O(m)$

So m insertions take $O(m)$ work total

Or amortized $\frac{O(m)}{m} = O(1)$ time.

Total Resizing work

For m insertions, the biggest the array could be is $2m$ (if m is arbitrarily large). So resizing will make arrays of size

$2m, m, \frac{m}{2}, \frac{m}{4}, \ldots$ down to whatever the starting point was.

Work is $c2m + cm + \frac{cm}{2} + \frac{cm}{4} + \cdots$ down to $c \cdot$(starting size)

Total work?

$$\sum_{i=0}^{\log(m)} c \cdot \frac{2m}{2^i} = 2mc \cdot \sum_{i=0}^{\log(m)} 2^{-i} \leq 2mc \cdot \sum_{i=0}^{\infty} 2^{-i} = 4mc = O(m)$$

Total Resizing work

For m insertions, the biggest the array could be is $2m$ (if m is arbitrarily large). So resizing will make arrays of size

$2m, m, \frac{m}{2}, \frac{m}{4}, \ldots$ down to whatever the starting point was.

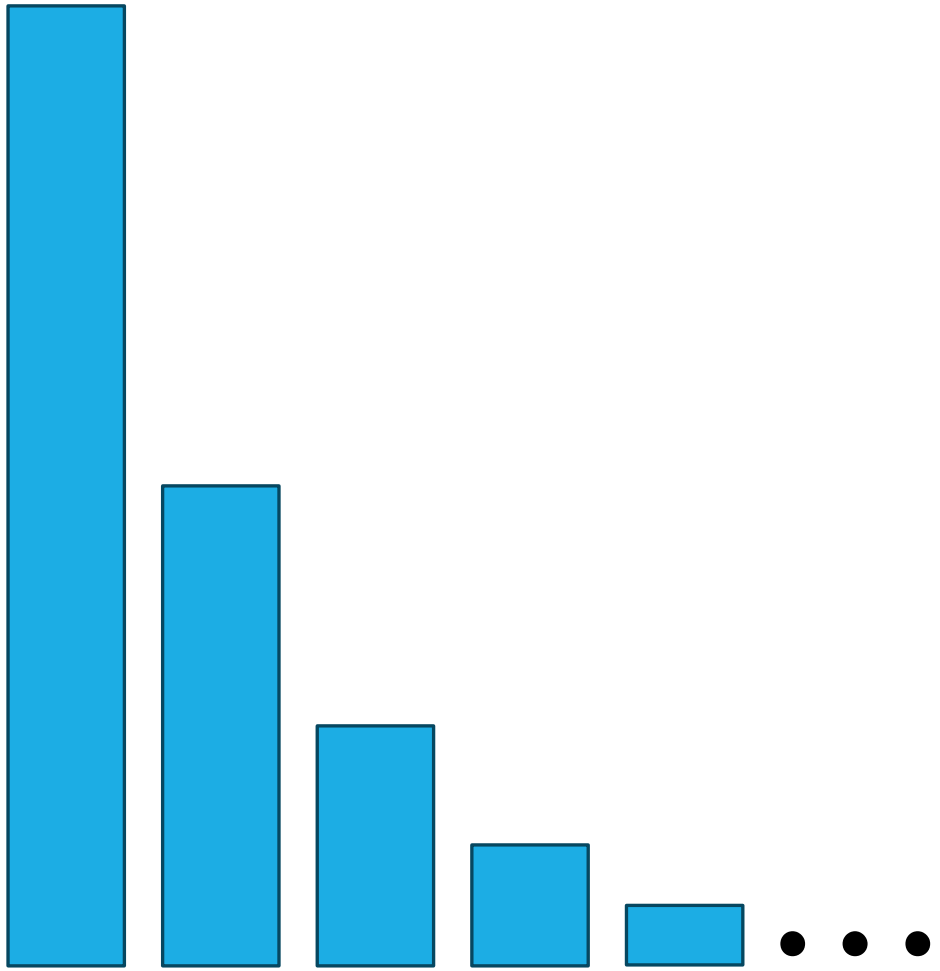
Work is $c2m + cm + \frac{cm}{2} + \frac{cm}{4} + \cdots$ down to $c \cdot$(starting size)

Total work?

$$\sum_{i=0}^{\log(m)} c \cdot \frac{2m}{2^i} = 2mc \cdot \sum_{i=0}^{\log(m)} 2^{-i} \leq 2mc \cdot \sum_{i=0}^{\infty} 2^{-i} = 4mc = O(m)$$

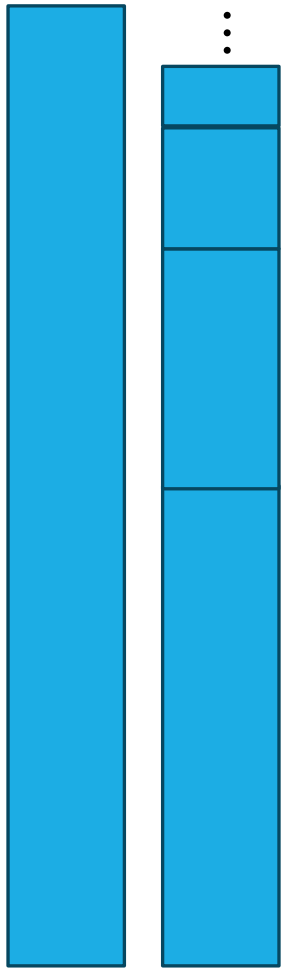
Summation Intuition

We'll see the summation $\sum_{i=0}^{\max} \frac{\text{constant}}{2^i}$ a lot. Why does it converge?



Summation Intuition

We'll see the summation $\sum_{i=0}^{\max} \frac{\text{constant}}{2^i}$ a lot. Why does it converge?



Every term in the summation fills half of the gap and leaves half the gap (because it's half as big).
but then the next term will fill only half the gap again (because it's half as big).

Half the total is in the first term, half the remaining total is in the next, …

Amortization

Why do we double? Why not increase the size by 10,000 each time we fill up?

How much work is done on resizing to get the size up to m?

Will need to do work on order of current size every 10,000 inserts

$$\sum_{i=0}^{\frac{m}{10000}} 10000i \approx 10{,}000 \cdot \frac{m^2}{10{,}000^2} = O(m^2)$$

The other inserts do $O(m)$ work total.

The amortized cost to insert is $O\left(\frac{m^2}{m}\right) = O(m)$.

Much worse than the $O(1)$ from doubling!

Amortization vs. Average-Case

Amortization and “average/best/worst” case are independent properties (you can have un-amortized average-case, or amortized worst-case, or un-amortized worst-case, or …).

Average case asks “if I selected a possible input on random, how long would my code take” (compare to worst-case: “if I select the worst value”)

Amortized or not is “do we care about how much our bank account changes on one day or over the entire month?” (do we care about the running time of individual calls or only what happens over a sequence of them?)

Why use (or don't use) amortized analysis?

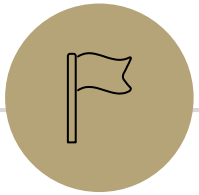
The appropriate analysis depends on your situation (and often it's worth knowing both).

A common use of data structures is as part of an algorithm.

- E.g., I'm trying to process everything in a data set, I insert everything into the data structure, remove them one at a time.
- In that case, we almost always want amortized analysis (we care about when the full analysis is done, not when we go from 49% done to 50% done).

But sometimes you care about individual calls

- Your data structure is feeding another process that the user is watching in real-time.



O, Omega, Theta vs. best/worst

O, Ω, Θ vs. Best, Worst, Average

It's a common misconception that $\Omega()$ is "best-case" and $O()$ is "worst-case". This is a misconception!!

$O()$ says "the complexity of this algorithm is at most" (think $\leq$)

$\Omega()$ says "the complexity of this algorithm is at least" (think $\geq$)

You can use $\leq$ on worst-case or best case; you can use $\geq$ on worst-case or best-case.

Best/Worst/Average say "what function f am I analyzing?"

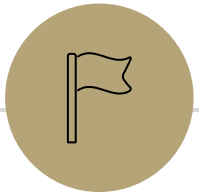
O, Ω, Θ say "let me summarize what I know about f, it's $\leq, \geq, =$..."

Some Example Sentences

	O	Ω	Θ
Best-Case Analysis	In the best-case, linear search will take at most as much time as binary search ever takes. That is, it's $O(\log n)$.	In the best case, linear search still has to look at array index 0; those operations still take time, so you will take at least $\Omega(1)$ time.	In the best case, linear-search is both $O(1)$ and $\Omega(1)$ so it is $\Theta(1)$.
Worst-Case Analysis	In the worst-case, binary search will take at most as much time as linear search ever takes. That is, it's $O(n)$.	In the worst-case, linear search will check at least as many locations in the array as binary search does in the worst-case, so it will take at least $\Omega(\log n)$ time	In the worst-case, binary search takes $\Theta(\log n)$ time. In the worst-case, linear search takes $\Theta(n)$ time.

Why Might you use it?

	O	Ω	Θ
Best-Case Analysis	In the best case, my algorithm is pretty good, it takes at most $O(time)$	Even in the best-case, this algorithm still takes a while; it takes at least $\Omega(time)$	In the best case, my algorithm takes exactly $\Theta(time)$
Worst-Case Analysis	Even in the worst-case my algorithm, isn't that bad! It takes at most $O(time)$ time in the worst-case	In the worst-case, there's still a lot of work the algorithm has to do; it takes at least $\Omega(time)$	In the worst case, my algorithm takes exactly $\Theta(time)$



Extra Amortization Example

A Contrived Example

A `MakesYouWaitList` operates as follows:

When you call `find()`, it does a linear search through an array of n elements to find i. If the index is odd, it spins for $O(n)$ time, if the index is even it spins for $O(n^2)$ time. Additionally, every n^{th} call to find it spins for $O(n^{2.5})$ time. It looks like this:

```
class MakesYouWaitList{
      int callsToFind=0; Object arr[]

find(Object o){
      n=arr.length
      int indexOfI=LinearSearch(o);
      if(indexOfI % 2 == 1)
            for(int k=0; k<n; k++) { }
      else
            for(int k=0; k<n*n; k++) { }
      callsToFind++;
      if(callsToFind == n) {
            callsToFind = 0;
             for(int k=0; k<Math.pow(n,2.5); k++) { }
      }
}
```

All the running times

	Best	Worst	Average
Amortized	$O(n^{1.5})$ Every n operations trigger the last $n^{2.5}$ spin time. No matter what elements are chosen. The best choices (all at even indices) will add n work each, which is a lower order term.	$O(n^2)$ On an odd input we take $O(n^2)$. The $O(n^{1.5})$ we want to assign to each for the big spin-time at the end is a lower order term.	$O(n^2)$On average, half the inputs will be at odd indices and half even, so we'll have: $\frac{n}{2}O(n) + \frac{n}{2}O(n^2) + O(n^{2.5})$ work, which is $O(n^3)$ total. Giving each of the n operations its share we get $O(n^2)$
Unamortized	$O(n)$ as long as the element is stored at an even index and doesn't trigger the resize, we'll get $O(n)$ time.	$O(n^2)$ The worst-case total work for n operations is for all to be odd $n \cdot O(n^2)$, and one to trigger the resize $O(n^{2.5})$. The total is $O(n^3)$, which we distribute equally among the n inserts.	$O(n^2)$ We have a $1/n$ chance of causing the $O(n^{2.5})$ spin, a ½ chance of getting $O(n)$ and ½ of $O(n^2)$, this gives: $\frac{1}{n} \cdot O(n^{2.5}) + \frac{1}{2}O(n) + \frac{1}{2}O(n^2)$ Which gives $O(n^2)$