More Heaps

Data Structures and Parallelism

Logistics

The spec for stack says operations should take "amortized O(1) time"
You will meet this requirement if you:

- double the size of the array when it fills up
- Take O(n) time to resize
- And take O(1) time when the array isn't full

Lecture on Friday will explain "amortized" time fully.

There's a "handouts" tab on the webpage.

You should be added to gradescope (submit exercise 1 there by Friday)

Outline

More Heaps
- Inserting and RemoveMin
- Some more operations
- Building a heap all at once

More Algorithm Analysis!

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

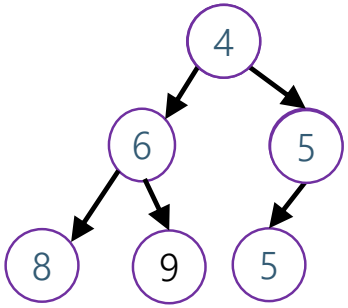
Uses:

- Operating System
- Well-designed printers
- Huffman Codes (in 143)
- Sorting (in Project 2)
- Graph algorithms

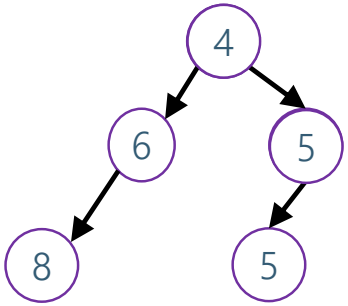
Tree Words

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

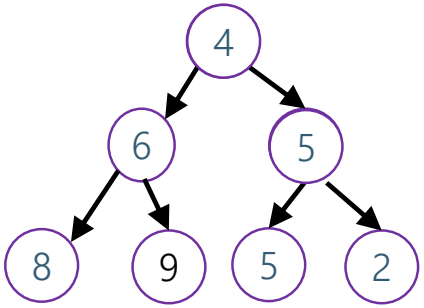
Perfect – every row is completely filled



Complete, but not perfect



Neither



Both Perfect and Complete

Binary Heaps

A Binary Heap is

1. A Binary Tree

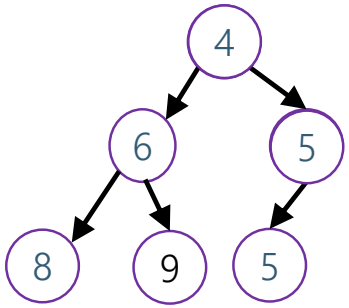
2. Every node is less than or equal to all of its children
   - In particular, the smallest element must be the root!

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

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

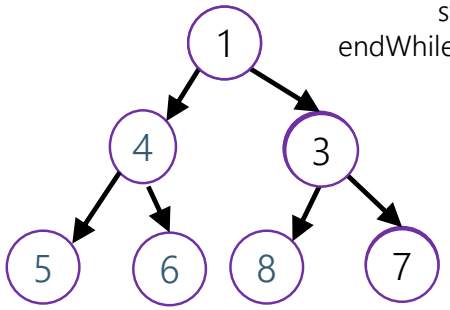
Shape is correct now

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

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.

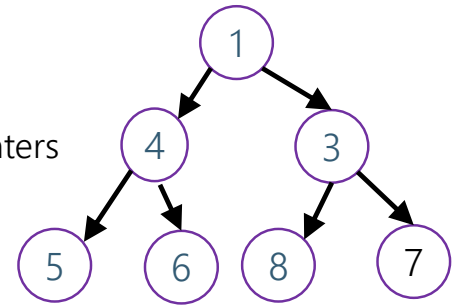
An Optimization

Pointers are annoying.

They're also slow.

Our tree is so nicely shaped, 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

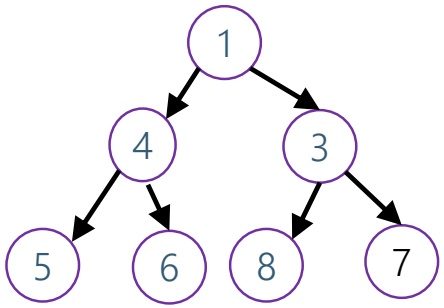
If I'm at index i, what is the index of:

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

More Operations

Later in the quarter, we'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.

Delete(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.

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's no guarantee that we're getting the worst case every time!

BuildHeap Running Time

Let's try again for a Theta bound.

The problem last time was making sure we always hit the worst case.

If we insert the elements in decreasing order **we will!**

So we really have n $\Theta(\log n)$ operations. QED.

There's still a bug with this proof!

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 $\Theta(\log n)$.

So the number of operations is at least

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

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

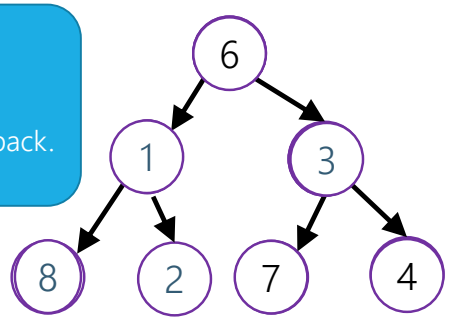
Ok, it's really faster.
But can we make it **work**?

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

Should we start at the top or bottom?

Two Possibilities

Fill out the question on pollev.com/robbie so I know when to bring us back.



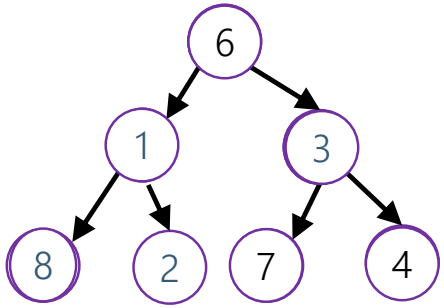

```
void StartTop(){
    for(int i=0; i < n;i++){
        percolateDown(i)
    }
}
```

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

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

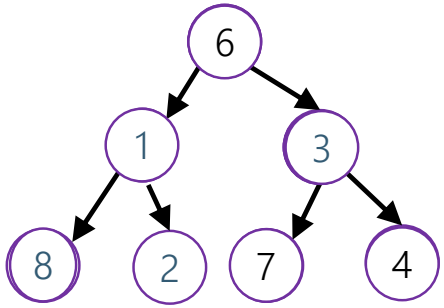
Only One Possiblity

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



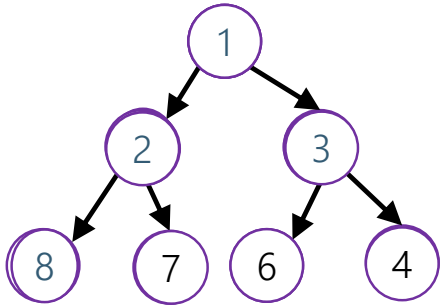
Only One Possiblity

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.
On the whiteboard.

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.

Delete(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.

Amortization

Amortization

How much does housing cost per day in Seattle?
Well it depends on the day.

The day rent is due, it's $1200.
The other days of the month it's free.

So on the worst-case day it's $1200.

Amortization

Amortization is an **accounting trick**. It's a way to reflect the fact that the "first of the month" is really responsible for the other days of the week, and each day should be assigned it's "fair share."

An amortized worst-case running time asks "for any sequence of k operations what is the worst-case of $O\left(\frac{\text{total work}}{k}\right)$"

Amortization

What's the worst case for inserting into an ArrayList?

- $O(n)$. If the array is full.

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

- If you're worried about a single insertion, maybe.
- If you're worried about doing, say, n insertions in a row. NO!

Amortized bounds let us study the behavior of a bunch of consecutive calls.

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 n insertions?

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

And the regular insertions are at most $n \cdot O(1) = O(n)$

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

Or amortized $O(1)$ time.

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 n?

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

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

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

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

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

Amortization vs. Average-Case

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

Average/best/worst asks "over all possible

Amortized or not is just "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?)