More Heaps

Data Structures and Parallelism

Logistics

Gitlab should run the tests and a "static analysis" every time you push your P1 code.

- Let us know if that's not happening.

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.

Generics, Debugging, "out of memory" info on "handouts" webpage

Outline

More Logistics:

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

Ben Jones will lecture Wednesday and Friday
- Robbie will still be checking email.

Today's Outline:

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

More Algorithm Analysis!

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.

Needing a pointer to the element isn't normal.

Exercise02 will have you think more about why we need it here.

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

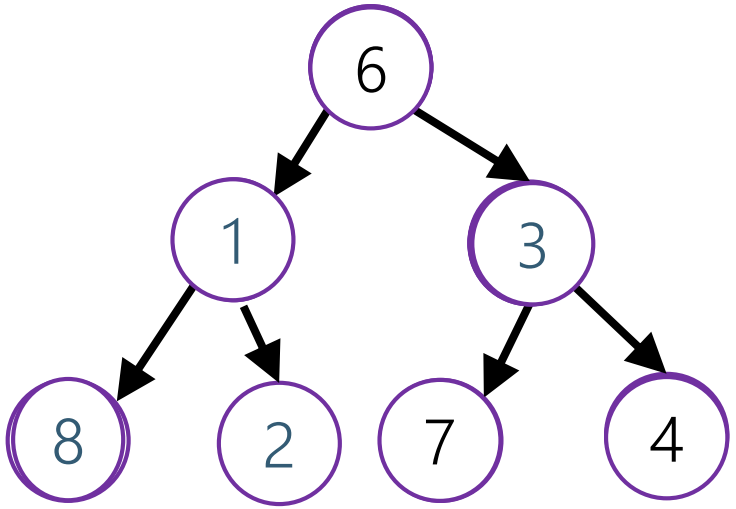
```
void StartTop(){
    for(int i=0; i < n;i++){
        percolateDown(i)
    }
}
```

```
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?

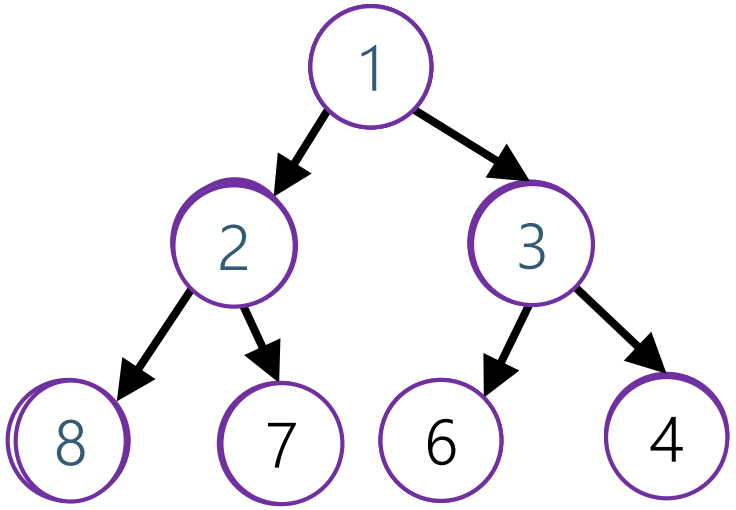
Only One Possiblity

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



Only One 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.

One Last Operation

Merge(heap1, heap2) given two heaps, combine them into one valid heap.

We can use buildheap for this. It runs in $O(n)$ time, where n is the size of the new, combined heap.

Can we do better?

Better Merge?

There are alternative implementations of minimum priority queues that merge better than regular heaps.

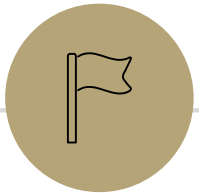
More details in the textbook, leftist heaps, skew heaps, binomial queues.

Main idea: don't have a rigid complete tree, allow for a more complicated tree structure.

Benefit: can merge in $O(\log n)$ time

Drawback: need to use actual pointers – no more array implementation

IT'S A TRADEOFF!!!



Analyzing Recursive Code

Calculating Running Times

Here's some code for calculating the length of a linked list:

What's its running time?

```
Length(Node curr){
    if(curr.next == null)
        return 1;
    return 1 + Length(curr.next);
}
```

We can analyze all the "non-recursive" work like usual

What about the recursive work?

Writing a Recurrence

If the function runs recursively, our formula for the running time should probably be recursive as well.

Such a formula is a **recurrence**.

$$T(n) = \begin{cases} T(n-1) + 2 & \text{if } n > 1 \\ 1 & \text{otherwise} \end{cases}$$

What does this say?

The input to T is the size of the input to the Length.

If the input to T() is large, the running time depends on the recusive call.

If not we can just use the base case.

Another example

```
Mystery(int n){
    if(n == 1)
        return 1;
    for(int i=0; i < n*n; i++){
        for(int j = 0; j < n; j++){
            System.out.println("hi!");
        }
    }
    return Mystery(n/2)
}
```

$$T(n) = \begin{cases} T(n/2) + n^3 + n^2 + 1 & \text{if } n > 1 \\ 1 & \text{otherwise} \end{cases}$$

Try It On Your Own

```
Mystery(int n){
    if(n <= 4)
        return 1;
    for(int i=0; i < n; i++){
        if(i % 3 == 2)
            break;
    }
    return Mystery(n - 5)
}
```

$$T(n) = \begin{cases} T(n-5)+3 & \text{if } n > 4 \\ 1 & \text{otherwise} \end{cases}$$

What Do We Do With That

That's nice. So what's the big-Θ bound.

Ben will teach you how to find the big-Θ on Wednesday!