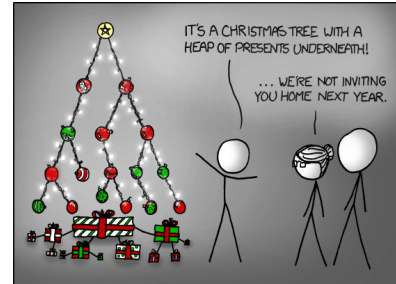


# CSE 332

## Data Abstractions

# Heaps



### Outline

- 1 Reviewing Heap Representation
- 2 Heap Operations, Again
- 3 buildHeap

### A New Data Structure: Heap

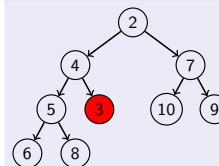
1

#### PriorityQueue ADT

insert(val)	Adds val to the queue.
deleteMin()	Returns the <b>highest priority</b> item not already returned by a deleteMin. (Errors if empty.)
findMin()	Returns the <b>highest priority</b> item not already returned by a deleteMin. (Errors if empty.)
isEmpty()	Returns true if all inserted elements have been returned by a deleteMin.

Heaps give us  $O(\lg n)$  insert and deleteMin:

#### And Now, Heaps



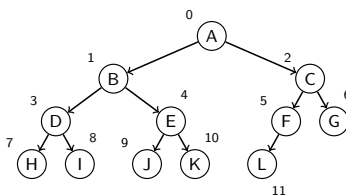
**Heap Property:**  
All Children are larger

**Structure Property:**  
Insist the tree has no "gaps"

### And... how do we implement Heap?

2

We've insisted that the tree be complete to be a valid Heap. Why?



Fill in an array in **level-order** of the tree:

heap: 

A	B	C	D	E	F	G	H	I	J	K	L	0	0	0
<small>h[0]</small>	<small>h[1]</small>	<small>h[2]</small>	<small>h[3]</small>	<small>h[4]</small>	<small>h[5]</small>	<small>h[6]</small>	<small>h[7]</small>	<small>h[8]</small>	<small>h[9]</small>	<small>h[10]</small>	<small>h[11]</small>	<small>h[12]</small>	<small>h[13]</small>	<small>h[14]</small>

parent(n) = (n - 1) / 2  
leftChild(n) = 2n + 1  
rightChild(n) = 2n + 2

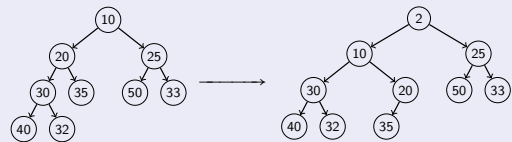
### insert Pseudocode

3

```

1 void insert(val) {
2   if (size == arr.length - 1) {
3     resize();
4   }
5   arr[size] = val;
6   percolateUp(size);
7   size++;
8 }
9
1 void percolateUp(hole) {
2   while (hole > 0 && arr[hole] < arr[parent(hole)]) {
3     swap(hole, parent(hole));
4     hole = parent(hole);
5   }
6 }
    
```

#### Insert 2 into this Heap



Before: 

10	20	25	30	35	50	33	40	32	0	0	0
<small>heap[0]</small>	<small>heap[1]</small>	<small>heap[2]</small>	<small>heap[3]</small>	<small>heap[4]</small>	<small>heap[5]</small>	<small>heap[6]</small>	<small>heap[7]</small>	<small>heap[8]</small>	<small>heap[9]</small>	<small>heap[10]</small>	<small>heap[11]</small>

After: 

2	10	25	30	20	50	33	40	32	35	0	0
<small>heap[0]</small>	<small>heap[1]</small>	<small>heap[2]</small>	<small>heap[3]</small>	<small>heap[4]</small>	<small>heap[5]</small>	<small>heap[6]</small>	<small>heap[7]</small>	<small>heap[8]</small>	<small>heap[9]</small>	<small>heap[10]</small>	<small>heap[11]</small>

## deleteMin Pseudocode

4

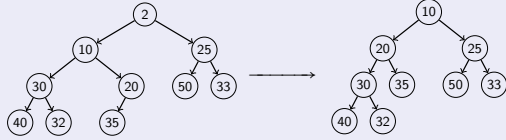
```

1 int deleteMin() {
2   if (isEmpty()) {
3     throw ...;
4   }
5   ans = arr[0];
6   arr[0] = arr[size - 1];
7   size--;
8   percolateDown(0);
9   return ans;
10 }

1 void percolateDown(bad) {
2   target = getSmallestChild(bad);
3   while (arr[target] < arr[bad]) {
4     swap(bad, target);
5     bad = target;
6     target = getSmallestChild(bad);
7   }
8 }

```

### Delete Min



Before: 

2	10	25	30	20	50	33	40	32	35	0	0
heap[0]	heap[1]	heap[2]	heap[3]	heap[4]	heap[5]	heap[6]	heap[7]	heap[8]	heap[9]	heap[10]	heap[11]

After: 

10	20	25	30	35	50	33	40	32	0	0	0
heap[0]	heap[1]	heap[2]	heap[3]	heap[4]	heap[5]	heap[6]	heap[7]	heap[8]	heap[9]	heap[10]	heap[11]

## insert Runtime

5

We know insert is  $\mathcal{O}(\lg n)$ , but...

Just like with BSTs, the **order** of insertion makes a big difference.

With **randomly ordered inputs**, we have:

- an average of **2.6** comparisons per insert
- an element moves up **1.6** levels on average

Unfortunately, we're not so lucky on deleteMin; we usually have to percolate all the way down.

## Analyzing insert's Average Case

6

Suppose a heap has  $n$  nodes.

- How many nodes on the bottom level?  $\frac{n}{2}$
- And the level above?  $\frac{n}{4}$
- etc.

Suppose we have a random value,  $x$ , in the heap.

- How often is  $x$  in the bottom level?  $\frac{1}{2}$  of the time
- And the level above?  $\frac{1}{4}$  of the time
- etc.

So, putting these things together, we see that for a random value  $x$ , there's a  $\frac{1}{2}$  probability we compare once, a  $\frac{1}{4}$  probability we compare twice, etc.

Taking a weighted average (expected value) gives us:

$$\text{Average \# of Comparisons} < \frac{1}{2} + \frac{2}{4} + \frac{3}{8} + \dots = \sum_{i=0}^{\infty} \frac{i}{2^i} = 2$$

This is  $\mathcal{O}(1)$ !

## Evaluating the Array Implementation

7

### Advantages

Minimal amount of wasted space:

- Only unused space on right in the array
- No "holes" due to complete tree property
- No wasted space representing tree edges

Fast lookups:

- Benefit of array lookup speed
- Multiplying and dividing by 2 is extremely fast (can be done through bit shifting (see CSE 351))
- Last used position is easily found by using  $\text{size} - 1$  for the index

### Disadvantages

What if the array gets too full (or wastes space by being too empty)? Array will have to be resized.

**Advantages outweigh Disadvantages: This is how it is done!**

## Changing the ADT...

8

### What else can we do with a heap?

Given a particular index  $i$  into the array...

- decreaseKey( $i$ , newPriority): Change priority, percolate up
- increaseKey( $i$ , newPriority): Change priority, percolate down
- remove( $i$ ): Call decreaseKey( $i$ ,  $-\infty$ ), then deleteMin

### What are the running times of these operations?

They're all worst case  $\mathcal{O}(\lg n)$ , but decreaseKey is **average**  $\mathcal{O}(1)$ .

## Building a Heap

9

### The Easy Way...

```

1 void buildHeap(int[] input) {
2   for (int i = 0; i < input.length; i++) {
3     insert(input[i]);
4   }
5 }

```

**What is the time complexity of buildHeap?**

The worst case is  $\mathcal{O}(n \lg n)$ .

### Can we do better?

With our current ADT, no! But if we have access to the internals of the data structure, we can.

In other words, if we **add a new operation to the ADT**, then we can do better.

This is a trade-off: convenience, efficiency, simplicity

## Building a Heap, Take 2

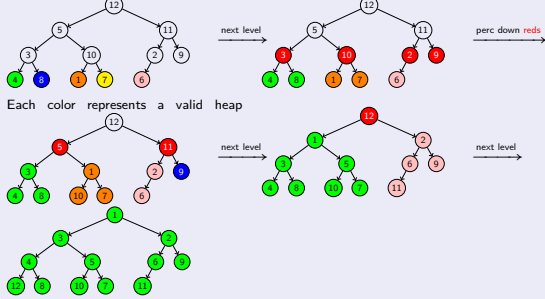
10

### Floyd's buildHeap Idea

Our previous attempt added a node, then fixed the heap, then added a node, then fixed the heap, etc.  
What if we added all the nodes and then **fixed the heap all at once!**

### Floyd's buildHeap

Each highlighted node is a valid heap! Percolate down red nodes until at top

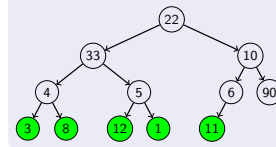


## Floyd's buildHeap

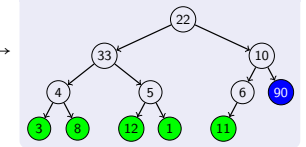
11

```
1 void buildHeap(int[] input) {
2     for (i = (size + 1)/2; i >= 0; i--) {
3         percolateDown(i);
4     }
5 }
```

The leaves begin as valid heaps



Now, we begin percolating down

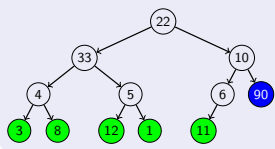


## Floyd's buildHeap

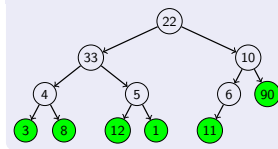
12

```
1 void buildHeap(int[] input) {
2     for (i = (size + 1)/2; i >= 0; i--) {
3         percolateDown(i);
4     }
5 }
```

percolateDown(6)



No changes to make!

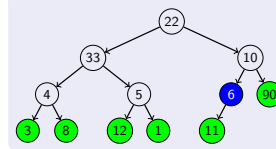


## Floyd's buildHeap

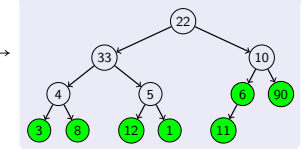
13

```
1 void buildHeap(int[] input) {
2     for (i = (size + 1)/2; i >= 0; i--) {
3         percolateDown(i);
4     }
5 }
```

percolateDown(5)



No changes to make!

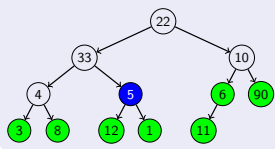


## Floyd's buildHeap

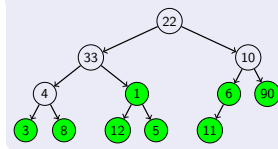
14

```
1 void buildHeap(int[] input) {
2     for (i = (size + 1)/2; i >= 0; i--) {
3         percolateDown(i);
4     }
5 }
```

percolateDown(4)



Swap 5 and 1

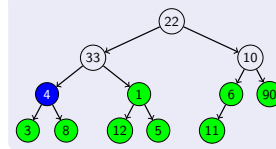


## Floyd's buildHeap

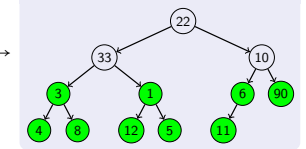
15

```
1 void buildHeap(int[] input) {
2     for (i = (size + 1)/2; i >= 0; i--) {
3         percolateDown(i);
4     }
5 }
```

percolateDown(3)

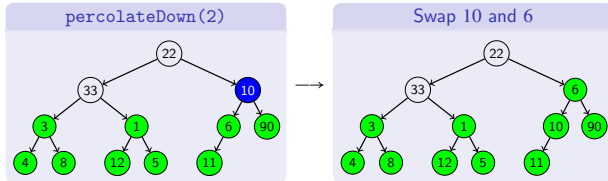


Swap 4 and 3



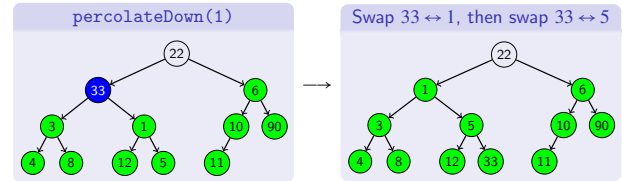
```

1 void buildHeap(int[] input) {
2   for (i = (size + 1)/2; i >= 0; i--) {
3     percolateDown(i);
4   }
5 }
    
```



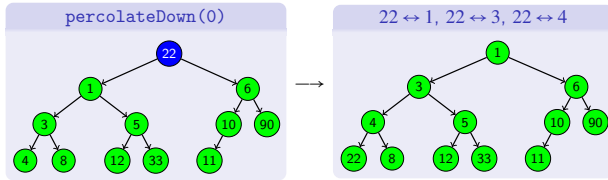
```

1 void buildHeap(int[] input) {
2   for (i = (size + 1)/2; i >= 0; i--) {
3     percolateDown(i);
4   }
5 }
    
```



```

1 void buildHeap(int[] input) {
2   for (i = (size + 1)/2; i >= 0; i--) {
3     percolateDown(i);
4   }
5 }
    
```



```

1 void buildHeap(int[] input) {
2   for (i = (size + 1)/2; i >= 0; i--) {
3     percolateDown(i);
4   }
5 }
    
```

The algorithm seems to work. Let's **prove it**:  
To prove that it works, we'll prove the following:

Before loop iteration  $i$ , all  $arr[j]$  where  $j > i$  have the heap property

Formally, we'd do this by induction. Here's a sketch of the proof:

- Base Case: All  $j > (size + 1) / 2$  **have no children**.
- Induction Step:  
We know that percolateDown **preserves the heap property** and makes its argument also have the heap property. So, after the  $(i+1)$ st iteration, we know  $i$  is less than all its children and by the IH, we know that all of the children past  $arr[i]$  already had the heap property (and percolateDown didn't break it).

So, since the loop ends with index 0, once we're done all the elements of the array will have the heap property.

```

1 void buildHeap(int[] input) {
2   for (i = (size + 1)/2; i >= 0; i--) {
3     percolateDown(i);
4   }
5 }
    
```

Was this even worth the effort?

The loop runs  $n/2$  iterations and each one is  $\mathcal{O}(\lg n)$ ; so, the algorithm is  $\mathcal{O}(n \lg n)$ .

This is certainly true, but it's **not**  $\Omega(n \lg n)$ ...

A Tighter Analysis

- On the second lowest level there are  $\frac{n}{2^2}$  elements and each one can percolate **at most** 1 time
- On the third lowest level there are  $\frac{n}{2^3}$  elements and each one can percolate **at most** 2 times
- ...

Putting this together, the **largest possible number of swaps** is:

$$\sum_{i=1}^k \frac{ni}{2^{i+1}} < \frac{n}{2} \left( \sum_{i=1}^{\infty} \frac{i}{2^i} \right) = \frac{2n}{2} = n$$

ADT?

- Without buildHeap, our ADT already let clients implement their own in  $\Omega(n \lg n)$  worst case
- By providing a specialized operation internally (with access to the data structure), we can do  $\mathcal{O}(n)$  worst case

Our Analyses!

- Correctness: Non-trivial inductive proof using loop invariant
- Efficiency:
  - First analysis easily proved it was  $\mathcal{O}(n \lg n)$
  - A tighter analysis shows the same algorithm is  $\mathcal{O}(n)$

**More Complicated Heaps**

- Leftist heaps, skew heaps, binomial queues (Weiss 6.6-6.8)
- Different data structures for priority queues that support a logarithmic time `merge` operation (impossible with binary heaps)
- Intuition: We already saw `merge` for the amortized array dictionary
- `insert` & `deleteMin` defined in terms of `merge`

***d*-heaps**

We can have heaps with  $d$  children instead of just 2 (see Weiss 6.5)

- Makes heaps shallower, useful for heaps too big for memory
- How does this affect the asymptotic run-time (for small  $d$ 's)?