Heaps and Heapsort

Abstract data type: Priority Queue Q

high # = highest priority

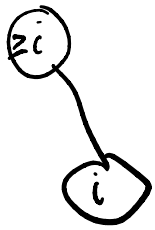
- $Q.insert(key)$ - inserts the element into Q.
- $Q.extractMax()$ - removes and returns the element of Q which has the largest key.
- $Q.findMax()$ - returns the element of Q which has the largest key.

Possible implementations: n elements in queue

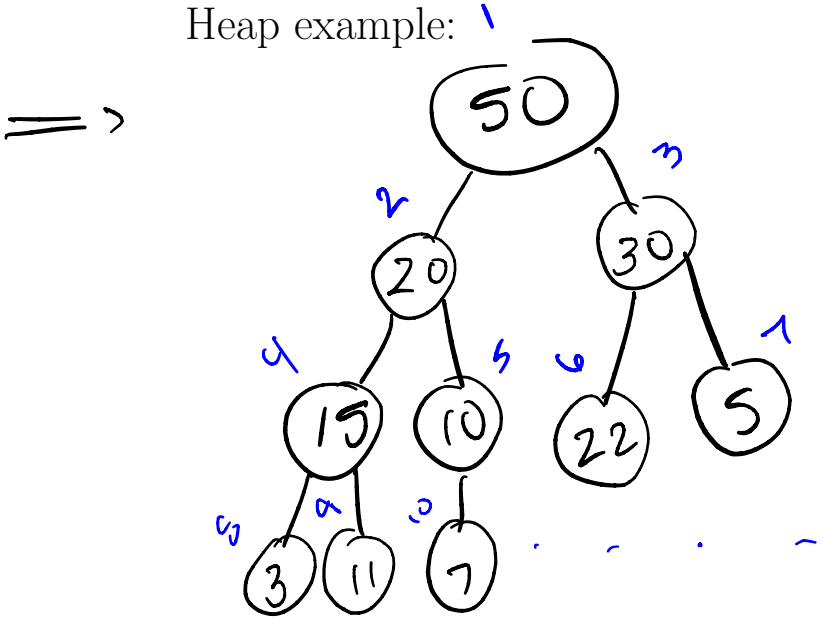
	insert	extractMax (remove)	findMax
Unsorted array:	$\Theta(1)$	$\Theta(n)$	$\Theta(n)$
Sorted array: (// largest # at end of array)	worst case $\Theta(n)$	$\Theta(1)$	$\Theta(1)$
Balanced Binary search trees:	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(\log n)$
Heaps: (// Max)	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(1)$

A max heap is a

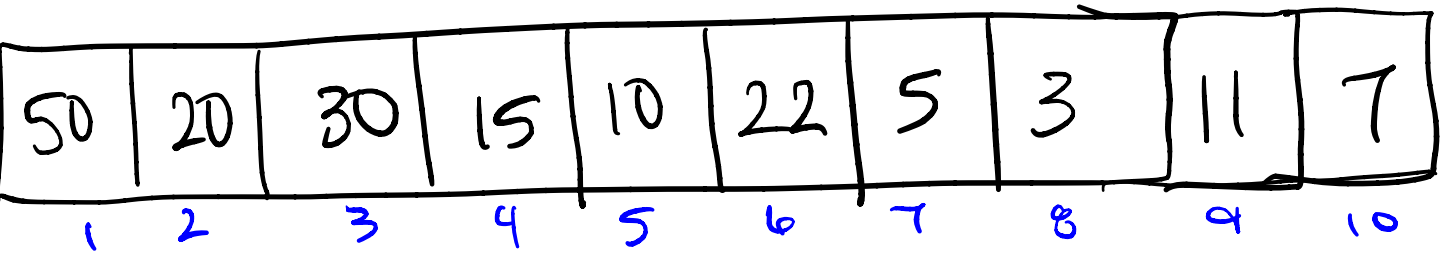
(1) almost complete binary tree (flushed left on last level)
(2) each node stores a key that fufills the
max-heap order property:
for all nodes, i, key of the parent of $i \geq$ the key of i



Heap example:



Because of property (1) the heap can be stored in an array:



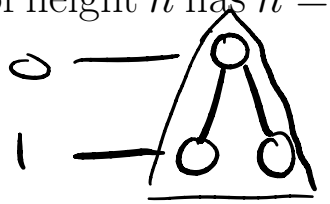
For the node stored at index i of the array

(1) The parent child of i will be at index $\lfloor \frac{i}{2} \rfloor$
(2) The left child of i will be at index $2i$
(3) The right child of i will be at index $2i + 1$

Prove using Induction when you reread notes

Lemma 1:
A complete binary tree of height h has $n = 2^{h+1} - 1$ nodes.

Proof:
By induction.



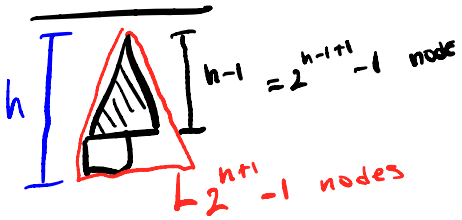
height(h) = 1

$2^{1+1} - 1 = 2^2 - 1 = 3$ nodes

(relatively easy; for the base case, remember a tree with only one node it has height 0)

Lemma 2:
For an almost complete binary tree of height h with n nodes it is the case that $h = \lfloor \log n \rfloor$.

Proof:
Since the tree is almost complete we know from Lemma 1 that $2^h - 1 < n \leq 2^{h+1} - 1$
(b) (a)



a) $n \leq 2^{h+1} - 1 \iff n < 2^{h+1}$ // log both sides

$\iff \log_2 n < \log_2 2^{h+1}$

$= \log_2 n < h+1$

$= h > \log_2 n - 1$ // lower bound

b) $2^h - 1 < n \iff 2^h \leq n$ // log both sides

$= \log_2 2^h \leq \log_2 n$

$= h \leq \log_2 n$ // upper bound

a & b) $\Rightarrow$ $\log_2 n - 1 < h \leq \log_2 n$ (a) (b)

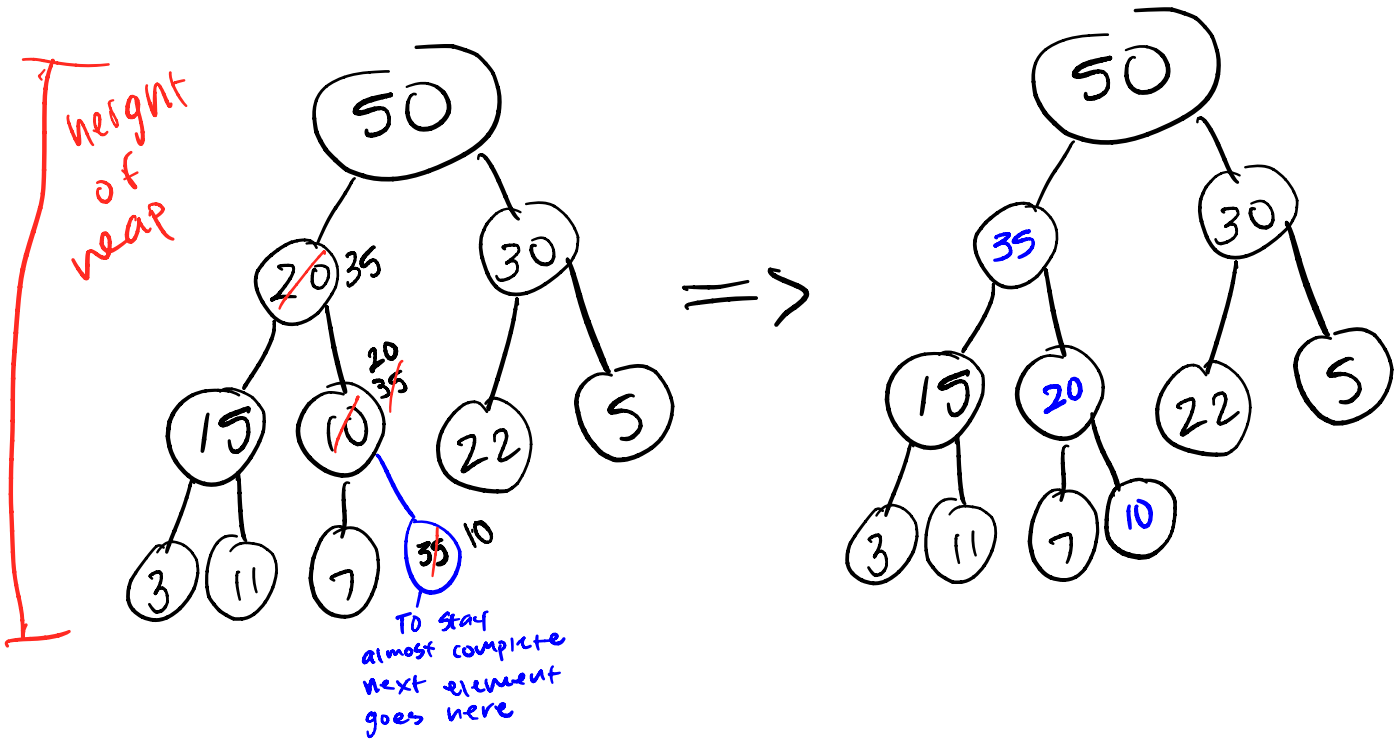
only differ by 1

$\Rightarrow \lfloor \log_2 n \rfloor \Rightarrow h$ is the whole # in this range $\Rightarrow h = \lfloor \log_2 n \rfloor$

Algorithm 1 void insert(int $A[1 \ldots n]$, int key)

```
//Heap is stored in A[1...n]
//Find a spot for key where it does not violate heap order property
i = n + 1;   //go to end of Array
while i > 1 and key > A[parent(i)] do
    A[i] = A[parent(i)]; // move parent down
    i = parent(i); //Find parent of parent
end while
A[i] = key; // insert i at correct location
```

Insert example: Insert(35)



Worst case runtime: $O(h) = O(\log n)$

=> we have to go all the way to top of heap

Best case runtime: $O(1)$

Algorithm 2 void heapifyDown(int $A[1 \ldots n]$, int i)

```
//Heap is stored in A[1...n]
//Push A[i] down the tree while it violates heap order property
while (left(i) ≤ n and A[i] < A[left(i)])
      or (right(i) ≤ n and A[i] < A[right(i)]) do
   //A[i] violates heap order property
   swap A[i] with its largest child node
   place index of largest child node in i
end while
```

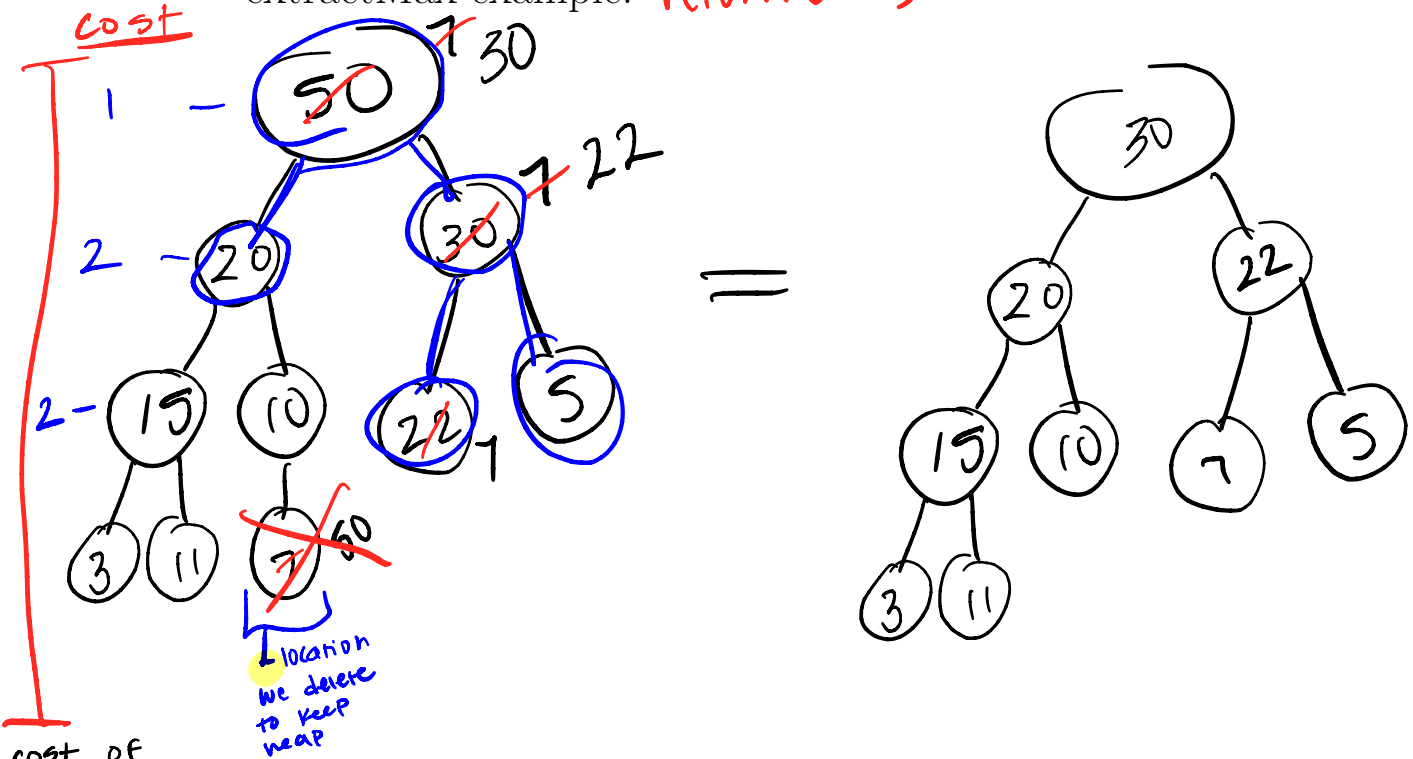
left(i) = 2i

right(i) = 2i + 1

Algorithm 3 int extractMax(int $A[1 \ldots n]$)

```
//Heap is stored in A[1...n]
//[illegible] violates heap order property
max = A[1];
A[1] = A[n];                          //fill hole at top of the heap
heapifyDown(A[1...(n − 1)], 1); //heapify remaining elements
return max;
```

extractMax example: return (50)

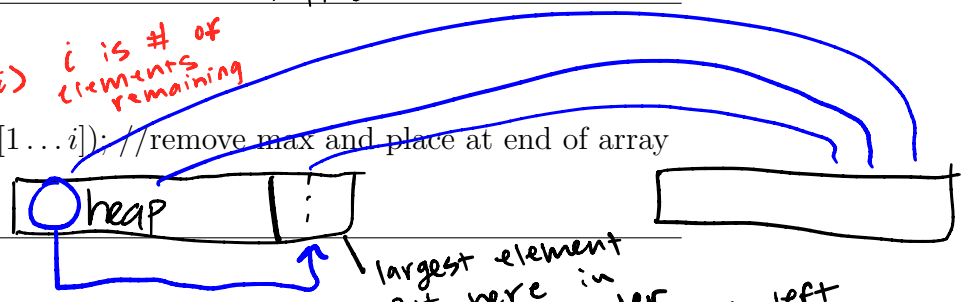


cost of 2 per level despite root = height of tree

Runtime of heapifyDown: $O(h) = O(\log n)$

Algorithm 4 void heapSort(int $A[1 \dots n]$) In-place

buildHeap(A);
$i = n$;
while $i \geq 2$ **do** O(log i) i is # of elements remaining
  $A[i]$ = extractMax($A[1 \dots i]$); //remove max and place at end of array
  $i--$;
end while



Runtime: $\Theta(n \log n)$

use Stirling's formula: $n! = \sqrt{2\pi n} \cdot (\frac{n}{e})^n \cdot (1 + \Theta(\frac{1}{n}))$

> 1 > 1

$\sum_{i=1}^{n} \log_2 i = \log_2(n) + \log_2(n-1) + \dots + \log_2(2) + \log_2(1)$

$\in \Theta(n \log n)$ $= \log_2(n \cdot (n-1) \cdot (n-2) \cdot \dots \cdot 2 \cdot 1)$

$= \log_2(n!)$ $\Rightarrow n! \approx (\frac{n}{e})^n$

$\Rightarrow (\frac{n}{e})^n \leq n! \leq n^n$

$\Rightarrow \log_2((\frac{n}{e})^n) \leq \log_2(n!) \leq \log_2((n)^n)$

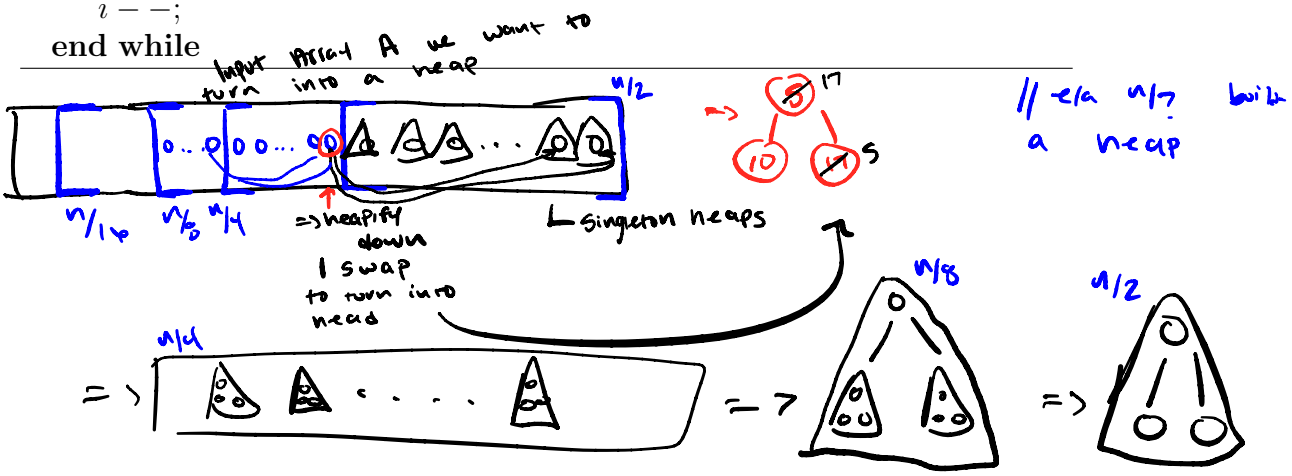
$= n \log_2(\frac{n}{e})$ $= n \log_2 n \in \Theta(n \log n)$

$= n \log n - n \log e$ $= O(1)$ $\in \Theta(n \log n)$

$\Rightarrow \log_2(n!) \in \Theta(n \log n)$ □

Algorithm 5 void buildHeap(int $A[1 \ldots n]$)

```
i = ⌊n/2⌋;
while i ≥ 1 do
    heapifyDown(A, i);
    i − −;
end while
```



$\frac{n}{2}$ 0 operations $\quad \frac{n}{4}$ 1 operation $\quad \frac{n}{8}$ 2 operations $\quad \frac{n}{16}$ 3 operations

$\Rightarrow$ Max # operations $\leq$ height of subtree $+1$

$$\Rightarrow \sum_{h=0}^{\log_2 n} \left(\begin{array}{l}\#\text{ elements} \\ \text{on level } h \\ \text{from bottom}\end{array}\right) \cdot \left(\begin{array}{l}\text{Max} \\ \text{cost} \\ \text{in operations}\end{array}\right)$$

height from bottom of heap

$$\Rightarrow \sum_{h=0}^{\log n} \left\lceil \frac{n}{2^{h+1}} \right\rceil \cdot h \quad \text{linear increase} \quad = ???$$

decrease exponentially

$\Theta(n)$ to build a heap