Technically, ~~if the heap is "filled out"~~ the max level is between $\lg n$ and $\lg n + 1$

$\lg n$ might not be integer

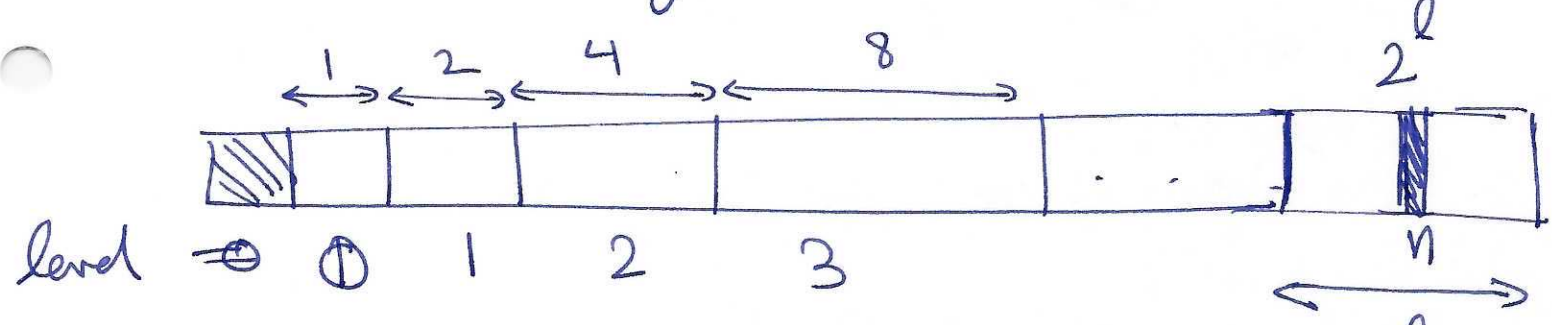
$\Theta(\lg n)$ $\quad$ $\lg n + \Theta(1)$

n = size

$A[n] \to A\left[\left\lfloor \frac{n}{2} \right\rfloor\right] \to A\left[\frac{n}{2^2}\right] \to A\left[\frac{n}{2^3}\right] \cdots A[1]$

parent

$\lg n$



Suppose max level is l

$\int e^x dx = e^x$

2^{2l}

$2^{l+1} - 1$

$\sum_{i=0}^{l} 2^i \geq n$ $\quad$ $\sum_{i=0}^{l-1} 2^i < n$

$= 2^{l+1} - 1 \geq n$ $\quad$ $2^l - 1 < n$

$$\frac{n+1}{2} \leq 2^l < n-1$$

$$\log(n+1) - 1 \leq l \leq \lg_2(n-1)$$

Binary Heap (Max) Invariant/Property

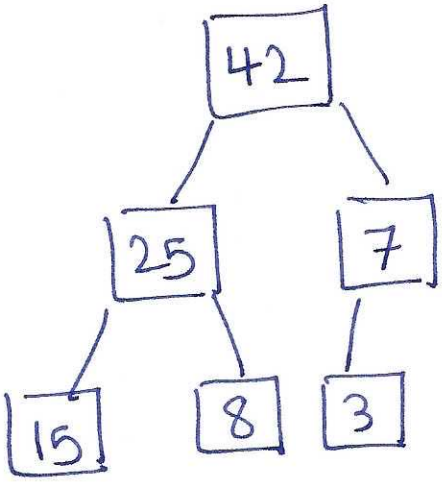
key at parent $\geq$ key at child

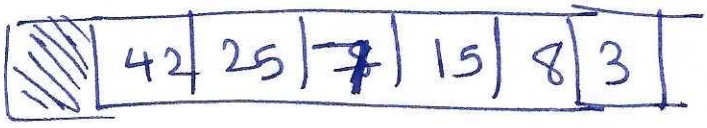
For all i $\begin{cases} A[i].key \geq A[2i].key \\ A[i].key \geq A[2i+1].key \end{cases}$

Q. What is the element with largest key?

~~(R) A[1]~~ (G) A[n] (B) Not sure

n = size





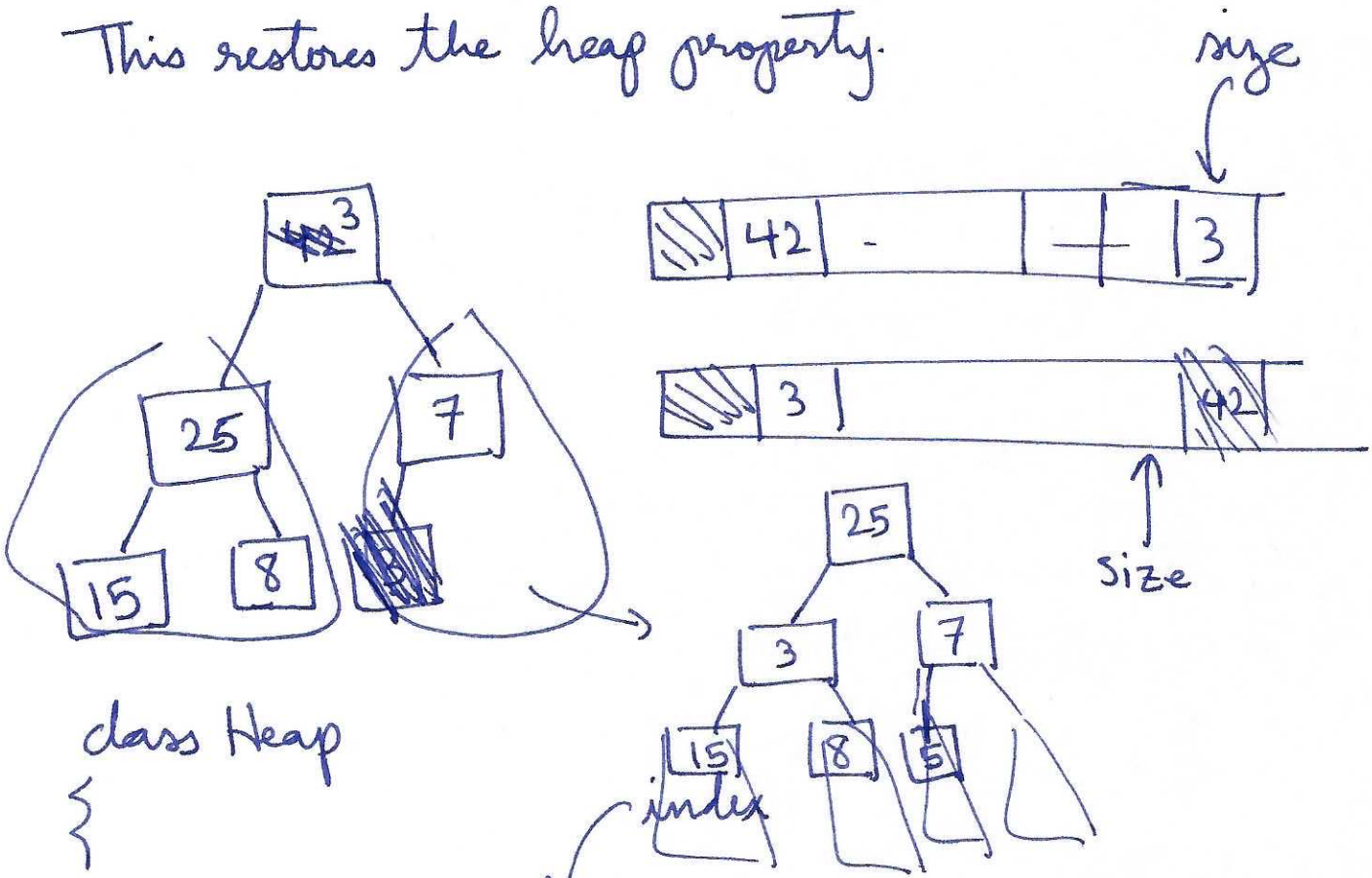
FindMax can be done in $\Theta(1)$.

ExtractMax

Insert

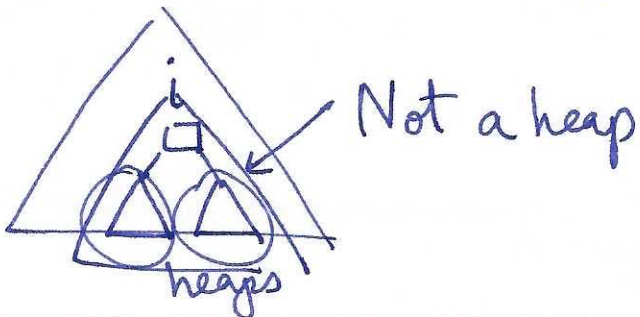
In a binary heap, inserts and Extract Max change the heap and affect the Heap Property. So we need to fix the invariants.

This restores the heap property.




```
class Heap
{
    Heapify(int i)
```

Heapify assumes that the child subheaps satisfy the heap property, and it will fix the heap so that subheap rooted at i also satisfies the heap property.



Heapify (i) (Sinking)

1. If A[i] is a leaf (no children), done. // Base case

2. If A[i].key $\geq$ A[2i].key
   AND A[i].key $\geq$ A[2i+1].key, done.
   // The subheap rooted at i is already heap.

3. Find child of A[i] with larger key
   j

4. Swap A[i] and A[j]

5. Heapify (j)

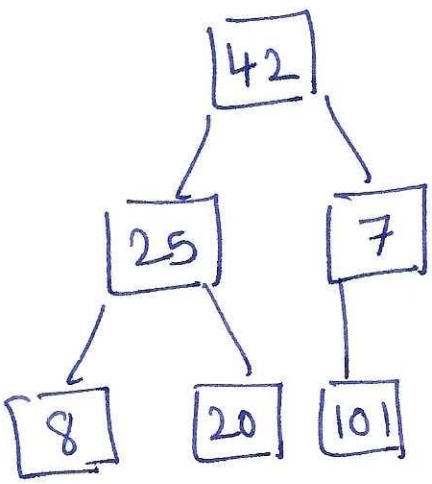
What is running time? $O(\lg n)$

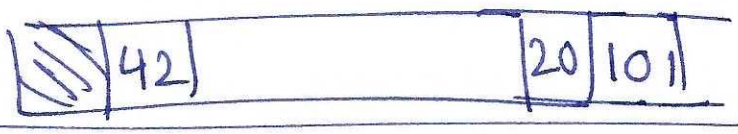
Extract Max

1. Swap A[1] with A[size]
2. Decrement size
3. Heapify (1)

~~(R)~~ 1 (G) Size (B) Not sure

Inserts





swim(i) ← index

1. If $i = 1$, return. (Base case)
2. If $A[i].key \leq A[\lfloor \frac{i}{2} \rfloor].key$, return. (No problem with heap)
3. Swap $A[i]$ with $A[\lfloor \frac{i}{2} \rfloor]$
4. swim$(\lfloor \frac{i}{2} \rfloor)$

insert (obj)

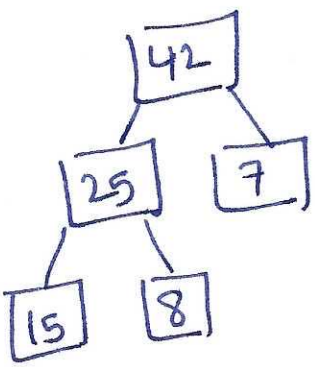
1. ~~A[s~~ Increment size.
2. A[size] = obj
3. swim(size)

Running time of insert is $O(\lg n)$.

HeapSort (A)

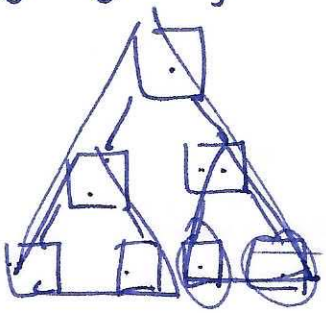
1. Insert all elements of A into heap (convert A into max heap) → n inserts
2. For i = 0 to n-1, ~~Extract~~ ExtractMax. → n extract max operations

$O(n \lg n)$ sorting algorithm



Starting with array A, convert A into max heap.

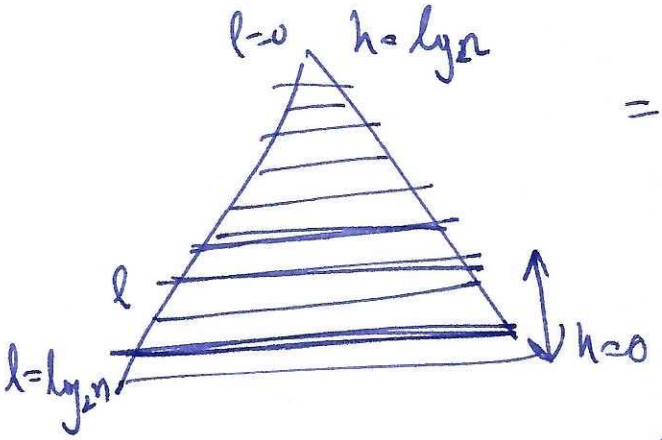
Imagine A as a heap (but failing heap property).



BuildHeap(A) (A has elements from A[1] to A[n])

1. For i from n to 1,
   Heapify(i).

Running time $= \sum_{i=1}^{n}$ Running time of Heapify(i)



$= \sum_{l=0}^{\log_2 n}$ (Running time of Heapify at depth l) $\times$ [# nodes at depth l]

$\leq \sum_{l=0}^{\log_2 n} 2^l \times$ (Running time of Heapify at depth l)

$= \sum_{h=0}^{\lg n} 2^{(\lg n - h)} \times O(h)$

(Running time of Heapify at depth l) $= O(\lg n - l)$

$= O\left(\sum_{h=0}^{\lg n} \frac{n}{2^h} \times h\right) = O\left(n \left[\sum_{h=0}^{\lg n} \frac{h}{2^h}\right]\right) = O(n)$

$\sum_{h=0}^{\lg n} \frac{h}{2^h} \leq 2$