(binary) MAX-heap

Rules:
- last row filled from left
- other rows full
- parent > children

\[ \text{parent}(i) = \left\lfloor \frac{i}{2} \right\rfloor \]

\[ \text{left-child}(i) = 2i \]

\[ \text{right-child}(i) = 2i + 1 \]

Notice every subtree is also a heap

1 2 3 4 5 6 7 8 9 10
16 14 10 8 7 9 3 2 4 1 ....
How does this relate to sorting?

Largest element is on top.
2nd largest is in level 2.
3rd largest is

4 in level 2
OR
4 in level 3 & child of 2nd

getting messy
extract MAX
attempt to extract MAX & restore heap
attempt to extract MAX & restore heap
attempt to extract MAX & restore heap
attempt to extract MAX & restore heap
attempt to extract MAX & restore heap
attempt to extract MAX & restore heap

now what? .... failed
attempt to extract MAX & restore heap

if we can do this in $O(\log n)$ time, we have a sorting algorithm
- extract MAX
- place rightmost leaf at top
  (strange move... it's small!)
- extract MAX
- place rightmost leaf at top
  (strange move... it's small!)
- swap top w/ largest child
  (locally restore parent->child)
- extract MAX
- place rightmost leaf at top
  (strange move... it’s small!)
- swap top w/ largest child
  (locally restore parent > child)
- repeat downward while needed
- extract MAX
- place rightmost leaf at top (strange move... it's small!)
- swap top w/ largest child (locally restore parent>child)
- repeat downward while needed
height of heap? \( \Theta(\log n) \) 

Ready to extract new \text{MAX} 

- extract \text{MAX} 
- place rightmost leaf at top 
  (strange move... it's small!) 
- swap top w/ largest child 
  (locally restore parent \text{\&} child) 
- repeat downward while needed 

\text{time?}
To work in-place when extracting max we can swap it w/ leaf.

Instead of deleting this node, just ignore it.

Notice max is stored at the max index of our array.
To work in-place when extracting max we can swap it w/ leaf.
To work in-place when extracting MAX we can swap it w/ leaf.
To work in-place when extracting max we can swap it w/ leaf.
To work in-place when extracting max we can swap it with leaf.
To work in-place when extracting max we can swap it w/ leaf.
To work in-place when extracting max we can swap it w/ leaf.
To work in-place when extracting max we can swap it w/ leaf.
To work in-place when extracting max we can swap it w/ leaf.
To work in-place when extracting max we can swap it w/ leaf.
To work in-place when extracting max we can swap it w/ leaf.
To work in-place when extracting \text{MAX} we can swap it with a leaf.
To work in-place when extracting max we can swap it w/ leaf.
To work in-place when extracting MAX we can swap it w/ leaf.
To work in-place when extracting MAX we can swap it w/ leaf.
To work in-place when extracting max we can swap it with leaf.
To work in-place when extracting max we can swap it w/ leaf.
To work in-place when extracting MAX we can swap it w/ leaf.
To work in-place when extracting \( \text{MAX} \) we can swap it with a leaf.
To work in-place when extracting max we can swap it with leaf.
So we can extract MAX & maintain a heap, in $O(\log n)$ time.

Do this $n$ times $\implies O(n \log n)$ sorting

But how did we have a heap in the first place?

Start with unsorted elements
So we can extract MAX & maintain a heap, in $O(\log n)$ time.

Do this $n$ times $\rightarrow O(n \log n)$ sorting

But how did we have a heap in the first place?

```
\begin{array}{cccc}
  & 2 & & \\
 4 & 10 & 3 & \\
 7 & 9 & 1 & \\
 8 & 16 & 14 & 1
\end{array}
```

make heap of size 1
So we can extract MAX & maintain a heap, in $O(\log n)$ time.

Do this $n$ times $\rightarrow O(n \log n)$ sorting

But how did we have a heap in the first place?

trivially get heap of size 2, possibly w/ a swap
So we can extract MAX & maintain a heap, in $O(\log n)$ time.

Do this $n$ times $\rightarrow O(n \log n)$ sorting

But how did we have a heap in the first place?

generally, insert new element as rightmost leaf in lowest level
So we can extract MAX & maintain a heap, in $O(\log n)$ time.

Do this $n$ times $\Rightarrow O(n \log n)$ sorting

But how did we have a heap in the first place?

\[ \begin{array}{c}
 2 \\
 4 \quad 10 \quad 3 \\
 7 \quad 8 \quad 16 \\
\end{array} \quad \begin{array}{c}
 14 \\
 1 \quad 9 \\
\end{array} \]

then repeat swapping while required
So we can extract MAX & maintain a heap, in $O(\log n)$ time.

Do this $n$ times $\rightarrow O(n \log n)$ sorting

But how did we have a heap in the first place?
So we can extract MAX & maintain a heap, in $O(\log n)$ time.

\[ \text{do this } n \text{ times} \rightarrow O(n \log n) \text{ sorting} \]

But how did we have a heap in the first place?
So we can extract MAX & maintain a heap, in $O(\log n)$ time.

Do this $n$ times $\rightarrow O(n \log n)$ sorting

But how did we have a heap in the first place?
So we can extract MAX & maintain a heap, in $O(\log n)$ time.

Do this $n$ times $\rightarrow O(n \log n)$ sorting

But how did we have a heap in the first place?
So we can extract MAX & maintain a heap, in $O(\log n)$ time.

"do this \( n \) times $\Rightarrow O(n \log n)$ sorting

But how did we have a heap in the first place?

e tc
When inserting a new leaf (wlog R) there is a problem iff \( R > P \)

By swapping \( R \leftrightarrow P \) we have a heap in subtree(R) \( (R > P > L) \)

...but we may have a new problem iff \( R > G \) then \( R \leftrightarrow G \). \( G > P > L : \text{OK} \) & \( R > G > X : \text{OK} \)
R will move up some path until smaller than node above.

That path will "shift down" so every subtree has a larger root.

\[ O(n \log n) \text{ time per node} \]

and in-place: iterate on array. Only swaps used.
(another) In-place heap-build

Iterate from end of array from right to left on each level, starting at bottom
In-place heap-build

Iterate from end of array

from right to left

on each level, starting at bottom

heapify each node \( x \)

4, i.e., subtree at \( x \).
Heapify x:

// can assume each child is a heap.

Might have to swap x w/ one of its children & further down levels
Heapify $x$:

- Can assume each child is a heap.
- Might have to swap $x$ with one of its children & further down levels.

Time for $x = O(\text{height}(x))$

Overall $O(n \log n)$
Time = \( O(h(x)) \)

Total time: \( O\left(\sum_{\text{all } x} h(x)\right) \)

\[ \sum \leq \frac{n}{2} \cdot 1 + \frac{n}{4} \cdot 2 + \frac{n}{8} \cdot 3 + \cdots + 2 \cdot ((\log n) - 1) + 1 \cdot \log n \]
\[ \sum \leq \frac{n}{2} \cdot 1 + \frac{n}{4} \cdot 2 + \frac{n}{8} \cdot 3 + \ldots + 2 \cdot (\log n - 1) + 1 \cdot \log n \]

\[ = \sum_{h=1}^{\log n} \frac{n}{2^h} \cdot h = n \cdot \sum \frac{h}{2^h} \leq n \frac{1/2}{(1-1/2)^2} = O(n) \]

\[ \text{Time} = O(h(x)) \]

\[ \text{Total time: } O \left( \sum_{\text{all } x} h(x) \right) \]