(binary) Max-heap

Rules:
1. last row filled from left
2. other rows full

Rule 1 (structure) is not always required in all applications

E.g. “delete only”

We will assume it’s required.
(binary) MAX-heap

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

parent(i) = ⌊i/2⌋
left-child(i) = 2i
right-child(i) = 2i + 1

Notice every subtree is also a heap

We can use an array to store a heap. No pointers.

16  14  10  16  3  9  7  2  4  1  8

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

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

(we want to preserve the structure)
Try again:
Modify extraction
- extract MAX
- place rightmost leaf at top

(strange move... it's small!)
- 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)
- extract MAX
- place rightmost leaf at top
  (strange move... it's small!)
- swap top with 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
- 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? $\Rightarrow \Theta(\log n)$

Ready to extract new MAX

- 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

$14 \rightarrow 16$
To work in-place when extracting max we can swap it w/ leaf.

One more time:
More space-efficient
To work in-place when extracting max, we can swap it with 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.
Extract max

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

$\leq$ do this $n$ times $\Rightarrow O(n \log n)$

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

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

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

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

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

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

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

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.

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

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

\[
\begin{array}{c}
14 \\
1 \\
9
\end{array}
\]

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

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

$\uparrow$ do this $n$ times $\rightarrow O(n\log n)$

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

$\downarrow$ do this $n$ times $\rightarrow O(n \log n)$

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 $\rightarrow O(n \log n)$

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 $\Rightarrow O(n \log n)$

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 \( n \) times } \rightarrow O(n \log n) \]

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

"Forward" method

read input array left to right

Also works "online": (streaming data)
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)
\( (R > P > L) \)
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$(R)$ $\quad (R > P > L)$

...but we may have a new problem iff $R > G$

then $R \leftrightarrow G. \quad G > P > L : \text{OK} \quad \& \quad R > G > X : \text{OK}$
R will move up some path until smaller than node above.

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

\[ O(n \log n) \text{ time per node and in-place: iterate on array. Only swaps used.} \]
“Reverse” heap-build

(also works in-place)
Iterate from end of array from right to left on each level, starting at bottom.

Heapify each node \( x \), i.e., subtree at \( x \).
Heapify $x$:

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

Compare two children, determine max.
Then compare $X$ with max, swap if needed.
Heapify x:

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

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

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

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

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

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

[CLRS 1148]

use \[\sum_0^{\infty} kx^k\]