(binary) MAX-heap

Rules:
- last row filled from left
- other rows full
- parent > children
(binary) MAX-heap

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

parent > children

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

Notice every subtree is also a heap
A binary MAX-heap is a complete binary tree where the value of each node is greater than or equal to the values of its children. The heap property ensures that the maximum value is at the root of the tree.

Rules:
- Last row filled from left
- Other rows full
- Parent > children

Notice every subtree is also a heap.

How can we identify the indices of the children of a node?
(binary) MAX-heap

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

left-child(i) = 2i
right-child(i) = 2i+1

Notice every subtree is also a heap
(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
(binary) MAX-heap

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

1 2 3 4 5 6 7 8 9 10

16 14 10 8 7 9 3 2 4 1 . . . .

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

\[
\text{parent}(i) = \lfloor i/2 \rfloor
\]

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

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

Notice every subtree is also a heap.
How does this relate to sorting?
How does this relate to sorting?

Largest element is on top.
How does this relate to sorting?

Largest element is on top.
2nd largest is in level 2.
How does this relate to sorting?

Largest element is on top.
2nd largest is in level 2.
3rd largest is in level 2.
How does this relate to sorting?

Largest element is on top.

2nd largest is in level 2.

3rd largest is

- 1 in level 2
- 4 in level 3
How does this relate to sorting?

Largest element is on top.

2nd largest is in level 2.

3rd largest is

- 4 in level 2
- 4 in level 3
- & child of 2nd
How does this relate to sorting?

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

(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 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
- 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

Time?
height of heap?

- 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
time?
height of heap? $\Rightarrow \Theta(\log n)$

- 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

- time?
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 with largest child
  (locally restore parent > child)
- repeat downward while needed

- time?
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 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.
Extract max

To work in-place when extracting \textit{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.
Restore heap

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 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 \texttt{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 \texttt{max} we can swap it w/ 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 $\Rightarrow O(n \log n)$

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?

Start with unsorted elements

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

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

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

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

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

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 $\rightarrow O(n \log n)$

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 → $O(n \log n)$

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

Then repeat swapping while required.
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?

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

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

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?
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.

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)

\[ \text{etc} \]
Correctness

When inserting a new leaf (wlog $R$) there is a problem iff $R > P$
When inserting a new leaf (wlog R), there is a problem iff $R > P$.

By swapping $R \leftrightarrow P$, we have a heap in $\text{subtree}(R)$ ($R > P > L$).
When inserting a new leaf (wlog R) there is a problem iff $R > P$ 

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

...but we may have a new problem iff $R > G$
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.
R will move up some path until smaller than node above.

That path will "shift down" so every subtree has a larger root.
R will move up some path until smaller than node above.

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

$O(n \log n)$ 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.
Iterate from end of array right to left on each level, starting at bottom

heapify each node \( x \), i.e., subtree at \( x \).
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.

\( \downarrow \) already heaps
Heapify $X$:

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

Might have to swap
x w/ 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 & further down levels.

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

\[ \text{Overall } O(n \log n) \]
Time $= O(h(x))$
Time = $O(h(x))$

Total time: $O\left(\sum_{all \ x} h(x)\right)$
\[ \sum \leq \frac{n}{2} \cdot 1 + \frac{n}{4} \cdot 2 + \frac{n}{8} \cdot 3 + \ldots + 2 \cdot ((\log n) - 1) + 1 \cdot \log n \]

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

\[ \text{Total time} : O \left( \sum_{\text{all } x} h(x) \right) \]
\[
\sum \leq \frac{n}{2} \cdot 1 + \frac{n}{4} \cdot 2 + \frac{n}{8} \cdot 3 + \cdots + 2 \cdot (\log n) - 1 + 1 \cdot \log n
\]

\[
= \sum_{h=1}^{\log n} \frac{n}{2^h} \cdot h
\]

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

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

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

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

\[
= \sum_{h=1}^{\log n} \frac{n}{2^h} \cdot h = n \cdot \sum_{h=1}^{\log n} \frac{h}{2^h} \leq n \cdot \frac{1/2}{(1-1/2)^2}
\]
\[ \sum \leq \frac{n}{2} \cdot 1 + \frac{n}{4} \cdot 2 + \frac{n}{8} \cdot 3 + \ldots + 2 \cdot (\log n - 1) + 1 \cdot \log n \]

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

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

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

[CLRS 1148]

\[ \text{use } \sum_{k=0}^{n} k^2 \]