Binary Heaps

- binary trees
- complete trees: perfectly balanced except bottom level

height is $H$

$2^H \leq N \leq 2^{H+1} - 1$

$H = 3$ 8  $\times$  $H = 3$ 15

$log(n+1) - 1 \leq H \leq log N$

height is $\Theta(log N)$

(full tree $2^{H+1} - 1$)
- heap-order property
  1. max-heap: parent ≥ child
  2. min-heap: parent ≤ child

→ today we will deal with max-heaps

max-heap representation

with arrays of length N+1 if heap has N elements

T
S
R
P
N
0 1 2 3 4 5
level order
\[ \text{parent}(k) = \frac{k}{2} \]
\[ \text{left}(k) = 2k \]
\[ \text{right}(k) = 2k+1 \]

\[ \text{We do not need links!} \]

---

**operations on wax-heaps**

1) for maintaining heap order

\[ \text{T} \]

\[ \text{H} \]
\[ \text{I} \]
\[ \text{S} \]
\[ \text{R} \]
\[ \text{D} \]
\[ \text{A} \]

\[ \text{E} \]
\[ \text{I} \]
\[ \text{N} \]
\[ \text{G} \]

**how?** we swap with largest child and repeat!

"sink" "demon"
max-heapify (A, i) \[ N = A.length() \]
\[ l = \text{left}(i); r = \text{right}(i) \]
\[
\text{if } A[l] > A[i] \&\& l \leq N
\]
\[ \text{largest} = l; \]
\[
\text{else}
\]
\[ \text{largest} = i; \]
\[
\text{if } A[r] > A[\text{largest}] \&\& r \leq N
\]
\[ \text{largest} = r; \]
\[
\text{if } \text{largest} \neq i
\]
\[ \text{swap}(A, i, \text{largest}) \]
\[ \text{max-heapify}(A, \text{largest}) \]

Example: \( i = 2 \)

max-heapify (A, 2)

\[ \text{largest} = 5 \Rightarrow 5 \]

after swap
max-heap(A, 5)
largest = 10 \rightarrow n'
after swap

running time of max-heapify
(intuitively: smallest element at root)

recurrence formula

\[ T(n) = T\left(\frac{2}{3}n\right) + C \]

why not \( \frac{1}{2} \)
correct case \( \Theta(\log n) \)
why do we insert in max-heap?
we need the opposite of max-heapity or sick/deadness
we can call it swim/promote

\[
\text{swim}(A, k, \ell)
\]

while \((k > 1) \&\& A[\ell] < A[k]\)

\[
\text{swap}(A, k, \ell)
\]

\[
k = k / 2
\]
\text{insert} \ (A, x) \quad \text{key}
\begin{align*}
&N++;
&A[N] = x; \quad \text{put at root}
&\text{swim} \ (A, N); \quad \text{promote}
\end{align*}

\text{insert} \ (27)
\begin{align*}
31 & \quad 31 \\
15 & \quad 25 \\
9 & \quad 20 & \quad 17 \\
8 & \quad 15 & \quad 7 & \quad 20 & \quad 12 \\
N = 8 & \quad 8
\end{align*}
extract Max(A)

\[ \text{max} \left( A[1] \right) ; \]
\[ \text{swap}(A,1,N) ; \]
\[ N-- ; \]
\[ \text{max-heapify}(A,1) ; \]
\[ \text{return } \text{max} \]

\[
\begin{array}{cccccccc}
39 & 1 & 27 & 25 & 1 & 1 & 1 & 1 \\
& 8 & 15 & 7 & 20 & 17 & 8 & 25
\end{array}
\]
\[
N-- ; \quad N = 7
\]
max-heapify (sink/dequeue)
swim/padicle

Θ(\text{log}_2 n)

\rightarrow

priority queue
Heap sort

heap sort(A)

build-max-heap(A)

for i = N down to 1
    swap(A, 1, i)
    max-heapify(A, 1)

27
  /   
15   25
  /     
8 7 20 17

17

15 25
  /   
8 7 20 27
Finish it later!
build-max-heap$(A)$

for $k = N/2$ down to 1

max-heapify$(A, k)$

why

if $k = N$ down 1 it would mean put each element at the right place

N=9 $k = N/2$ means that we start from the first non-leaf.

This is definitely $O(n \log n)$

?
Observe that max-heapify runs at different height

\[ h = \frac{N}{2} \rightarrow h = \frac{N}{2} \text{ (2 worst case)} \]

Running time of building a heap is rather

**Sum of the heights** instead of \( \frac{N}{2} \cdot \log N \)

Height of heap is \( h \).

Assume worst case of full tree.

- \( 1 \) node with height \( h \)
- \( 2^{h-2} \) nodes with height \( 2 \)
- \( 2^{h-1} \) nodes with height \( 1 \)

\[ \text{Sum} = 1 \]
\[ \begin{align*}
2^{h-1} &+ 1 \\
2^{h-2} &+ 2 \\
2^{h-3} &+ 2 \\
1 &+ 1 + \ldots + 1
\end{align*} \]
\[ = 2^{h+1} - h + 1 \]
\[ = N - h + 2 \]

build-max-heap is \( \Theta(N) \)

heapsort worst case \( \Theta(N \log N) \)