Priority Queues

A priority queue is a collection of items, each item has a priority (numerical value).

Max-priority queue — items with highest value removed first

Min-priority queue — ----- lowest --, -----

Let H be a max-priority queue. H supports the following basic operations:

FindMax(H) — returns item in H with highest value (H is not altered)

DeleteMax(H) — removes the highest item

Insert(H, x) — insert item x into H

IncreaseKey(H, x, k) — increase the key (i.e., the priority) of item x in H to k

---

Implementations via heaps

3 ways of implementing a (max-)heap

1. Binary heap — array
2. Binomial heap — linked struct
3. Fibonacci heap — " "

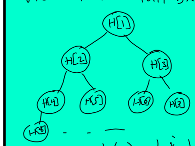
(2 and 3: mergeable)

(max-) Binary heap H

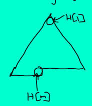
H[1...n] array for some n
n is the capacity of H

H has m items, stored in indices 1, ..., m.

View H as a full binary tree:



$parent(i) := \lfloor \frac{i}{2} \rfloor$
(index of H[i]'s parent for $i \geq 2$)
$left(i) := 2i$ — index of H[i] left child
$right(i) := 2i+1$ — ---- ---right...

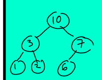


H.size = # items in H

~~FindMax(...)~~
~~return~~

Items in H are in max-heap order meaning, for all $2 \leq i \leq m$,

$H[parent(i)] \geq H[i]$



example

Max-heap order implies max item is H[1].

FindMax(H) — return H[1].

Insert(H, x) —

H.size ++;
H[H.size] := x
i := H.size
while $i \geq 2$ and H[i] > H[parent(i)] :
  Swap H[i] with H[parent(i)]
  i := parent(i)

"cascade up"

$d = depth = \lg n \pm O(1)$

$Time = (\#iterations)(time\ for\ an\ iteration) + O(1)$, with #iterations $= \Theta(\lg n)$ and time for an iteration $= O(1)$

$= \Theta(\lg n)$ worst-case for a heap of size n.

Min-heap ops: FindMin, DeleteMin, Insert, DecreaseKey } { FindMax, ⋮, IncreaseKey

---

DeleteMax(H) — uses a subroutine MaxHeapify

MaxHeapify(H, i)

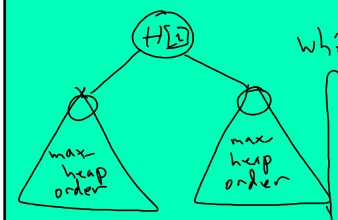
H[1.. H.size] array of keys

$1 \leq i \leq H.size$

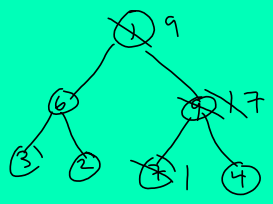
Precondition: subtrees rooted at left(i) and right(i) are in max-heap order

Post-condition — tree rooted at i is in maxheap order

MaxHeapify only rearranges items in the subtree rooted at i.



$\Theta(\lg H.size)$



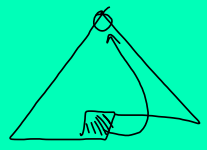
"cascade down"

While $i < H.size$ and at least one of i's children is bigger than $H[i]$ ($H[left(i)] > H[i]$ or $H[right(i)] > H[i]$)
- Swap $H[i]$ with the larger of its children
- Change i to the index of that swapped child

DeleteMax(H)

$O(1)$ $H[1] := H[H.size]$; $H.size--$

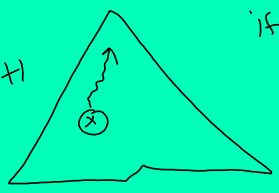
$O(\lg n)$ MaxHeapify(H, 1)



n = size of H

Time is $\Theta(\lg n)$ worst case

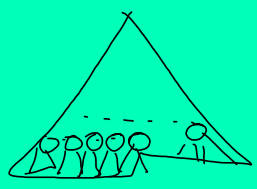
IncreaseKey(H, x, k) — given its location as an index



if $k > x$ then change x to k, cascade up from this point

Time = $\Theta(\lg n)$ worst-case.

BuildMaxHeap(H, n)
// Turns H into a (binary) max-heap of size n



H' empty max heap

for i := 1 to n
- Insert(H', H[i])

H := H'

BuildMaxHeap(H, n)
- i := $\lfloor n/2 \rfloor$
- while i ≥ 1:
  - MaxHeapify(H, i)
  - i--