CSCE 750
9/14/2023

Priority Queues / Heaps

But first: Change of variables in recurrences

Change-of-variables can turn a strange recurrence into a more familiar recurrence (that is equivalent)

$$T(n) = 2T(\sqrt{n}) + \lg n$$

Let $k := \lg n$, so $n = 2^k$

Define

$$S(k) = T(n) = T(2^k)$$

Find a recurrence satisfied by $S(k)$:

$$S(k) = T(2^k) = 2T(\sqrt{2^k}) + \lg(2^k)$$

$$= 2T(2^{k/2}) + k$$

$$= 2S(k/2) + k$$

$\therefore S(k) = \Theta(k \lg k)$ $\quad k = \lg n$

$T(n) = \Theta(\lg n \lg\lg n)$

Priority Queue

Abstract def: A **max-priority queue** is a collection of items each including a priority, drawn from some totally ordered set.
(Java: Comparable interface)
supporting the following basic operations:

core ops

- Insert(H, x) – insert item x into max-priority queue H
- FindMax(H) – return the max-priority item in H (nonempty). H is unaltered.
- DeleteMax(H) – remove (& return) the max-priority item in H. H is altered.
- IncreaseKey(H, x, k) – increase priority of item x in H to k
  (undef if priority is already > k)

A min-priority queue is the same but with priority ordering reversed:

FindMin, DeleteMin, DecreaseKey } instead of { FindMax, DeleteMax, IncreaseKey

Implementations (max-heaps)

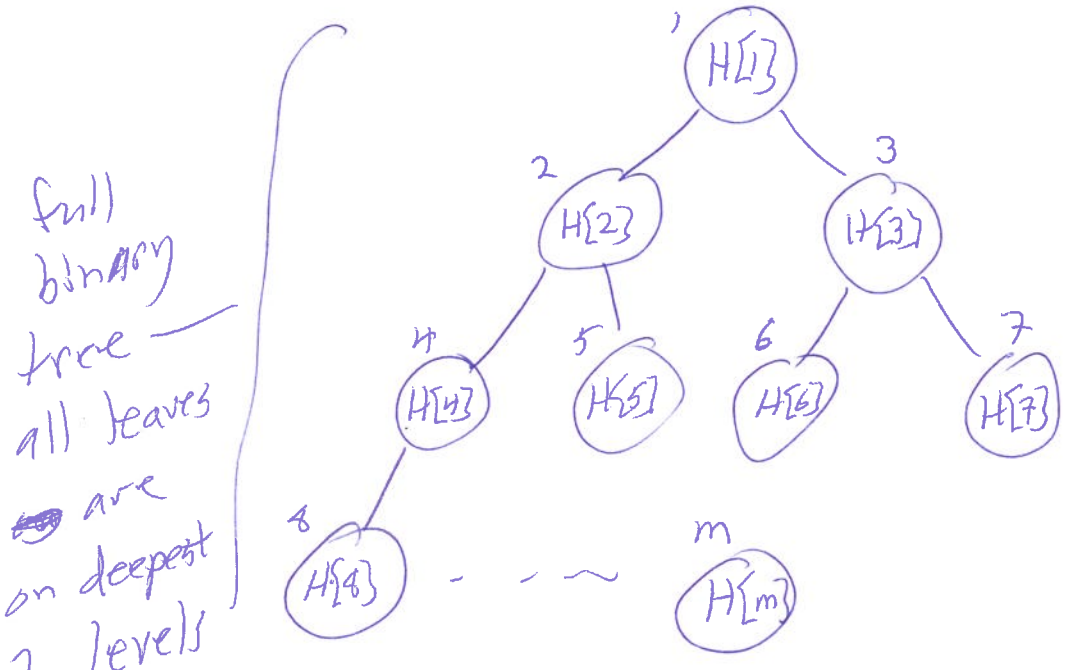
- Binary heaps — (first) — arrays
- Fibonacci heaps — (won't discuss) } linked structs
- Binomial Heaps

Binary Heap

$H[1..n]$ can hold n items (just numbers in this example)

A heap of size m is stored in $H[1], \ldots, H[m]$ $(m \leq n)$

Picture as a full binary tree:



For $2 \leq i \leq n$,

$parent(i) = \lfloor \frac{i}{2} \rfloor$

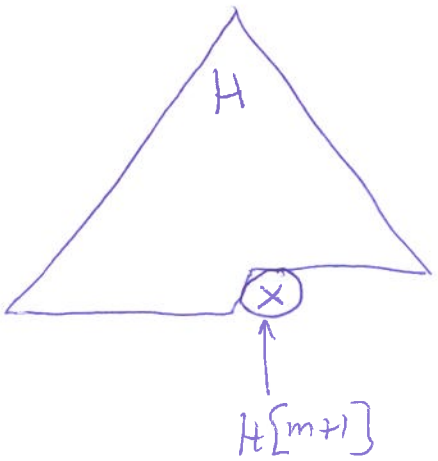
$left(i) = 2i$

$right(i) = 2i+1$

Items are stored in max-heap order: For any $2 \leq i \leq m$, $H[parent(i)] \geq H[i]$.

FindMax(H)
    return H[1]

Insert(H, x)

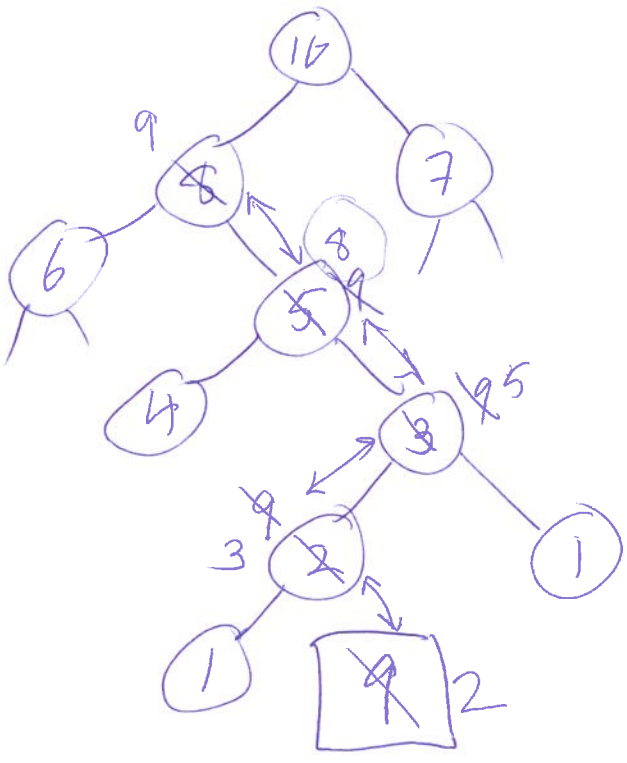


O(1)

1. H[m+1] := x

2. "Cascade up" to restore max heap order:

O(1) i := m+1
while i ≥ 2 and \
    H[parent(i)] < H[i]:
O(1) swap(H[parent(i)], H[i])
    i := parent(i)



Time = Θ(# of while-loop iterations)

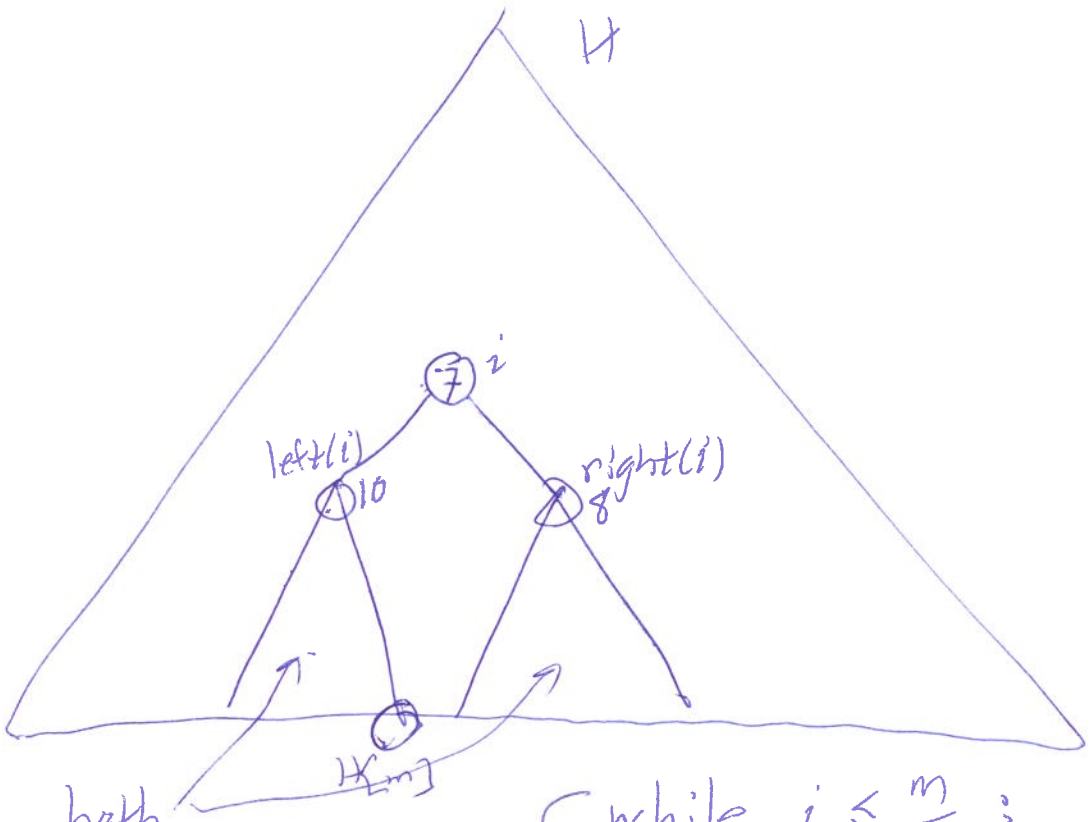
= Θ(lg m)

(m = size of H)

Subroutine MaxHeapify(H, m, i)

takes an array $H[1..m]$ and index $1 \le i \le m$ and it assumes (precondition) that the subtrees rooted at left(i) and right(i) are in max-heap order.

Returns with the subtree rooted at i in max-heap order (with the same items as before)



both in max heap order

"cascade down"

while $i \le \frac{m}{2}$:

$O(1)$

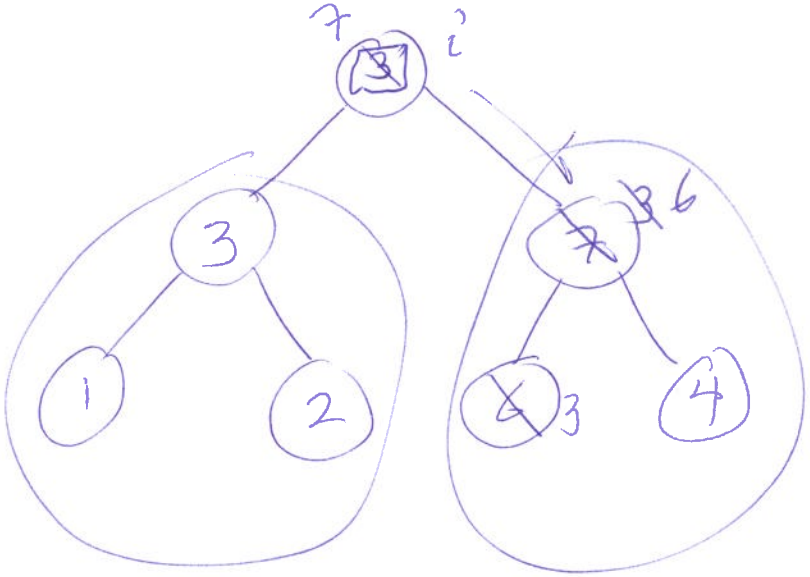
if H[i] is less than one of its children (H[left(i)], H[right(i)])

swap H[i] with the bigger of its children

i := left(i) or right(i) which ever is the bigger child

else break

Ex



Running time of MaxHeapify(H, i, m)

(Worst-case)

Time = Θ(# loop iterations)

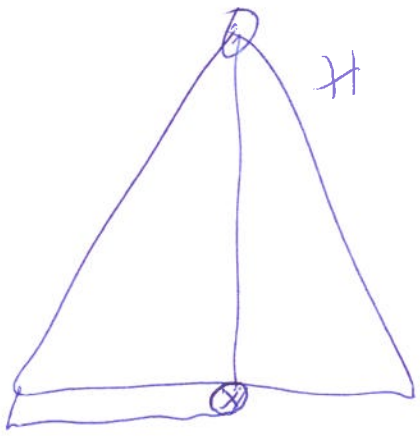
$= \Theta(\lg m - \lg i) = \Theta(\lg \frac{m}{i})$

DeleteMax(H)

save H[1] (to return it)

O(1) | H[1] := H[m]

$\Theta(\lg m)$ | MaxHeapify(H, m-1, 1)



1st method: Do m DeleteMax / Insert pairs to build a new max heap.

Worst-case time is $\Theta(m \lg m)$

Faster way takes $\Theta(m)$ time

```
BuildMaxHeap(H, m)
// Puts H[1..m] into max heap order
for i := ⌊m/2⌋ downto 1 do
    MaxHeapify(H, m, i)
```