CSE 101-02 5-26-22

- Student Evaluations of Teaching (SETs)

  open: Mon 5/23

  closes: Sun 6/5, 11:59 PM

Priority Queues chap 6

use Heap data structure { max ← cover, min

Binary

- Priority Queues { max ← cover / min

States:

A state in a P.Q. is a finite Set S of 'records' each with a key attribute.

$$x = (\underbrace{\cdot, \cdot, \cdot, \cdot}_{\text{satellite data}}, key) \in S$$

$$S = \{\cdot, \cdot, \cdot, x, \cdot, \cdot\}$$

↑ state

- ADT operations:

- Insert(S, x):
  insert a new record x into S

- Max(S)
  return record with minimum key.

- ExtractMax(S)
  return and delete record with minimum key

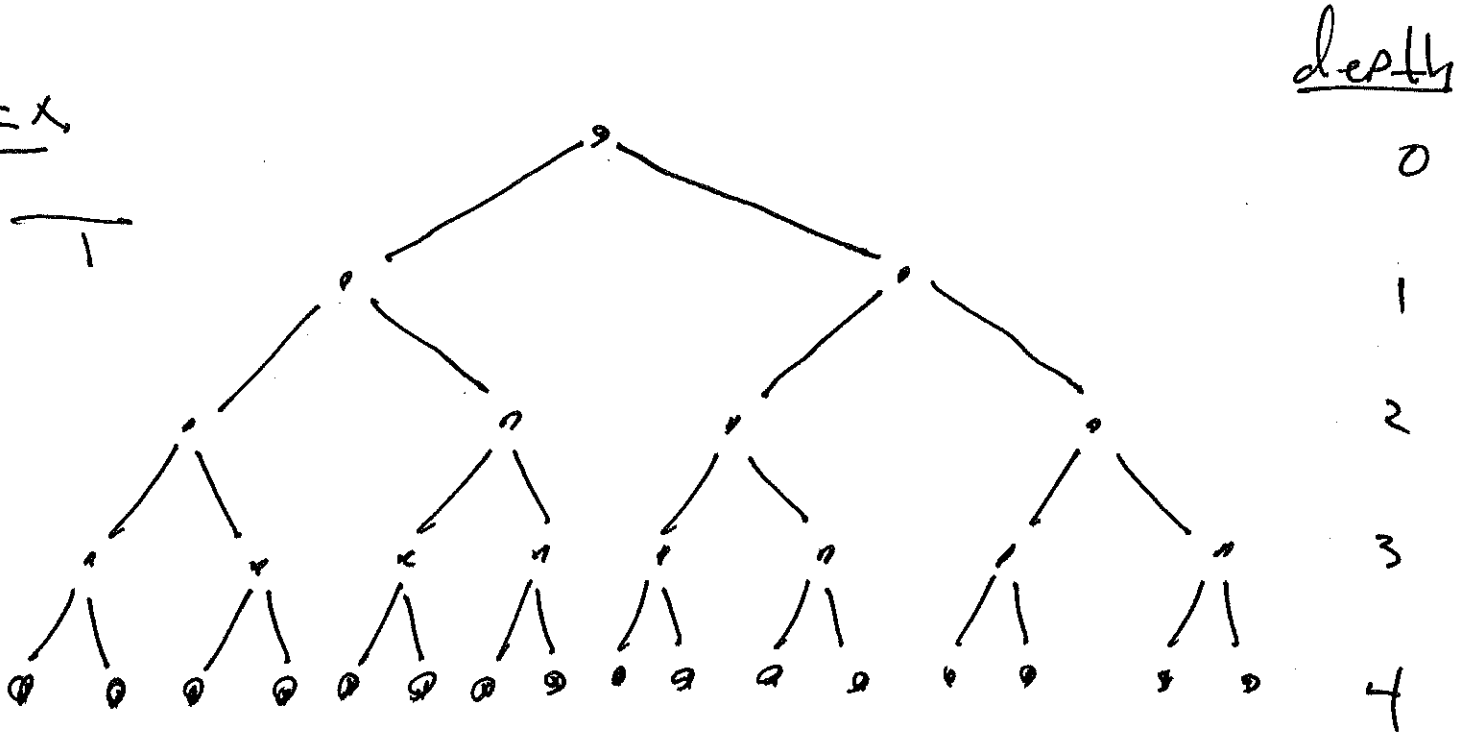
- IncreaseKey(S, x, k):
  change Key[x] to k if k > Key[x], else do nothing.

new data structure: Heap { max / min

depth

A complete binary tree (CBT) is a B.T. in which all leaves have same depth, all internal nodes have 2 children.

Ex



T

observe

$$(\#\text{nodes at depth } d) = 2^d$$

so if T has n nodes, height h

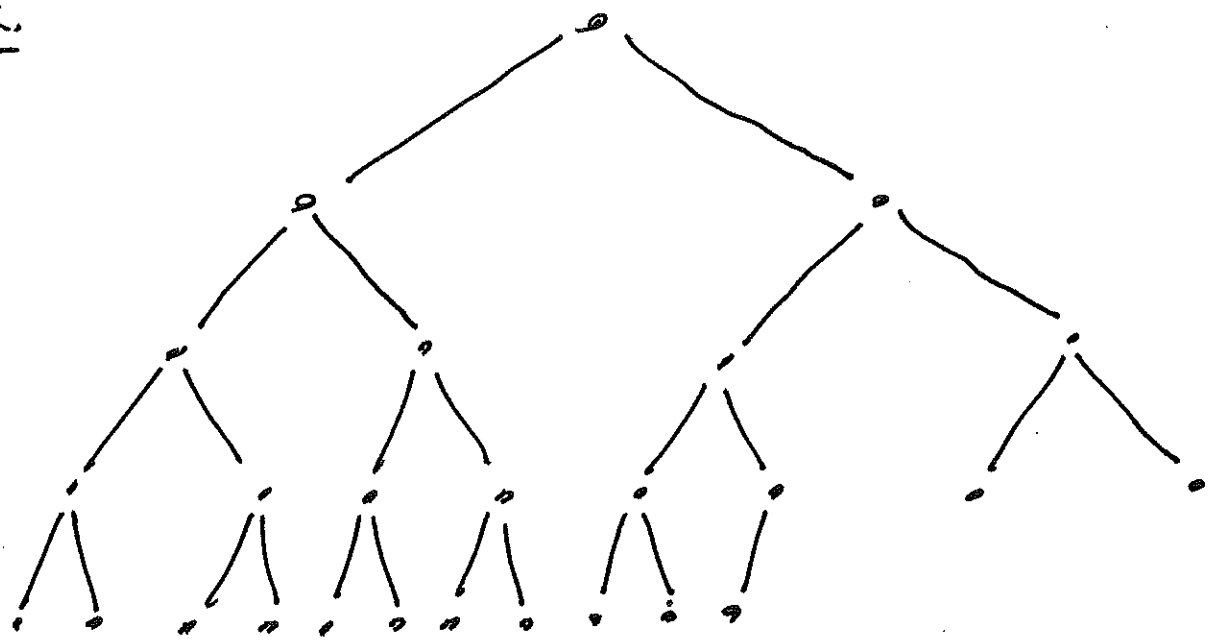
$$n = \sum_{d=0}^{h} 2^d = \frac{2^{h+1} - 1}{2 - 1} = 2^{h+1} - 1$$

Defn

An Almost Complete Binary Tree (ACBT)

is filled at all levels, except possibly the last (deepest), which is filled left to right.

Ex. T



The # of nodes n in an ACBT with height h satisfies

$$2^h - 1 < n \leq 2^{h+1} - 1$$

$$\therefore \quad 2^h \leq n < 2^{h+1}$$

$$\therefore \quad h \leq \lg(n) < h+1$$

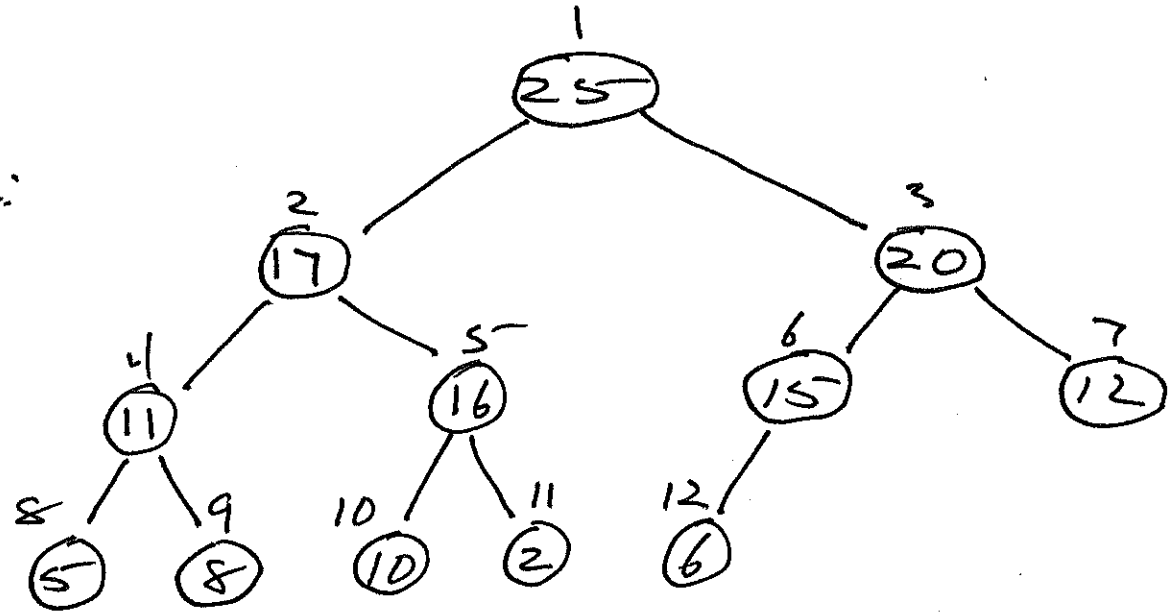
$$\therefore \quad \boxed{h = \lfloor \lg(n) \rfloor}$$

6.1 Heaps

A Binary Heap is an array used to represent an ACBT

Ex.

Tree:



Array:

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
A	25	17	20	11	16	15	12	5	8	10	2	6	·	·	·	·

2 attributes: length[A] = 16

heapSize[A] = 12

helper functions:

$$\begin{cases} Parent(i) = \lfloor \frac{i}{2} \rfloor & (i \geq 2) \\ left(i) = 2i \\ right(i) = 2i + 1 \end{cases}$$

heap properties:

* max : $A[Parent(i)] \geq A[i]$

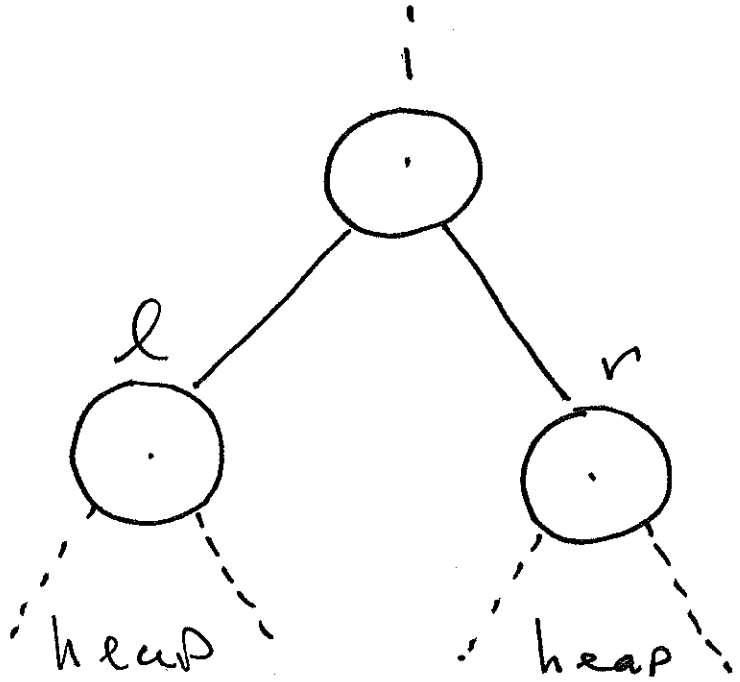
min : $A[Parent(i)] \leq A[i]$

operations on a heap:

- Heapify()
- BuildHeap()
- HeapSort()
- P.Q. operations .....

6.2 Heapify

establishes (max) heap property at index i



Pre: subtrees at l, r are valid heaps, where

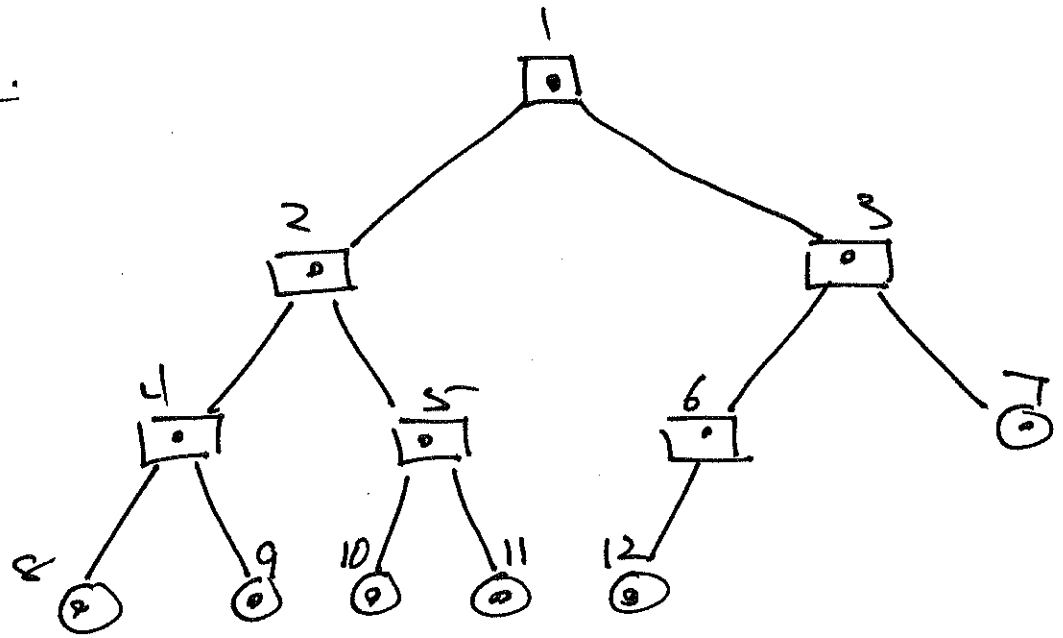
$l = left(i)$

$r = right(i)$

runtime = $\Theta(\log n)$

6.2 Build Heap

Ex.



$A_1 \dots A_6$ $A_7 \dots A_{12}$

internal nodes leaves

call Heapify on: 6, 5, 4, 3, 2, 1

rightmost internal node has index:

$$\left\lfloor \frac{n}{2} \right\rfloor$$

runtime $= \Theta(n)$

6.4 Heapsort

take an unordered array A

- call BuildHeap(A)
- swap leftmost with rightmost
- decrement heapSize
- call Heapify on root (ind = 1)

⋮

repeat.

$$\text{runtime} = \Theta(n) + n \cdot \Theta(\log n)$$

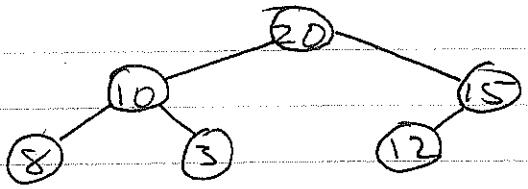
$$= \Theta(n + n\log n)$$

$$= \Theta(n \log n).$$

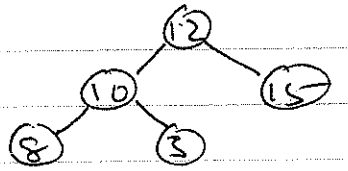
start: 10 8 12 20 3 15

Ex. AFTER Build-Heap(A):

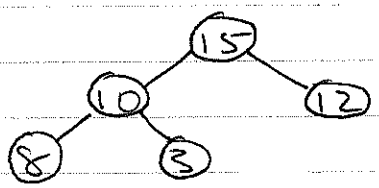
20 10 15 8 3 12 |



12 10 15 8 3 | 20

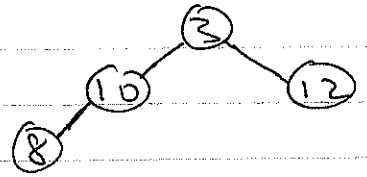


Heapify →

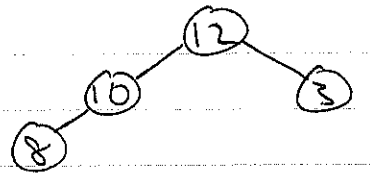


15 10 12 8 3 | 20

3 10 12 8 | 15 20

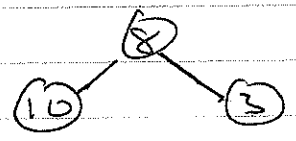


Heapify →

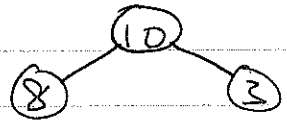


12 10 3 8 | 15 20

8 10 3 | 12 15 20

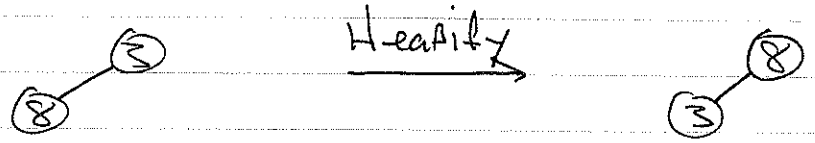


Heapify →



10 8 3 | 12 15 20

3 8 | 10 12 15 20



8 3 | 10 12 15 20

3 | 8 10 12 15 20

Run Time

The call to Build-Heap takes $\Theta(n)$ time in worst case. Each of the $n-1$ calls to Heapify takes $\Theta(\lg n)$ time in worst case.

Therefore the worst case Run Time of Heapsort is

$$T(n) = \Theta(n) + (n-1)\Theta(\lg n) = \Theta(n \lg n)$$

Note this is as good as MergeSort (asymptotically speaking), but Heapsort sorts in-place, while MergeSort requires extra memory.

6.5 Priority Queues { max + min

see Pseudo-code

- HeapMaximum(A)

  cost: $\Theta(1)$

- HeapDeleteMax(A)

  cost: $\Theta(\log n)$

- HeapExtractMax(A)

  cost: $\Theta(\log n)$

- HeapIncreaseKey(A, i, k)

  cost: $\Theta(\log n)$