CSE 101-01 5-26-22

- Student Evaluation of Teaching (SETs)

  open mon. 5/23

  close sun. 6/5 11:59 PM

new ADT: Priority Queue { min / max

state: a finite set S of records, with a key field

$x = (\cdot, \cdot, \cdot, \cdot, \text{key})$

(satellite data)

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

Operations: (max priority queue)

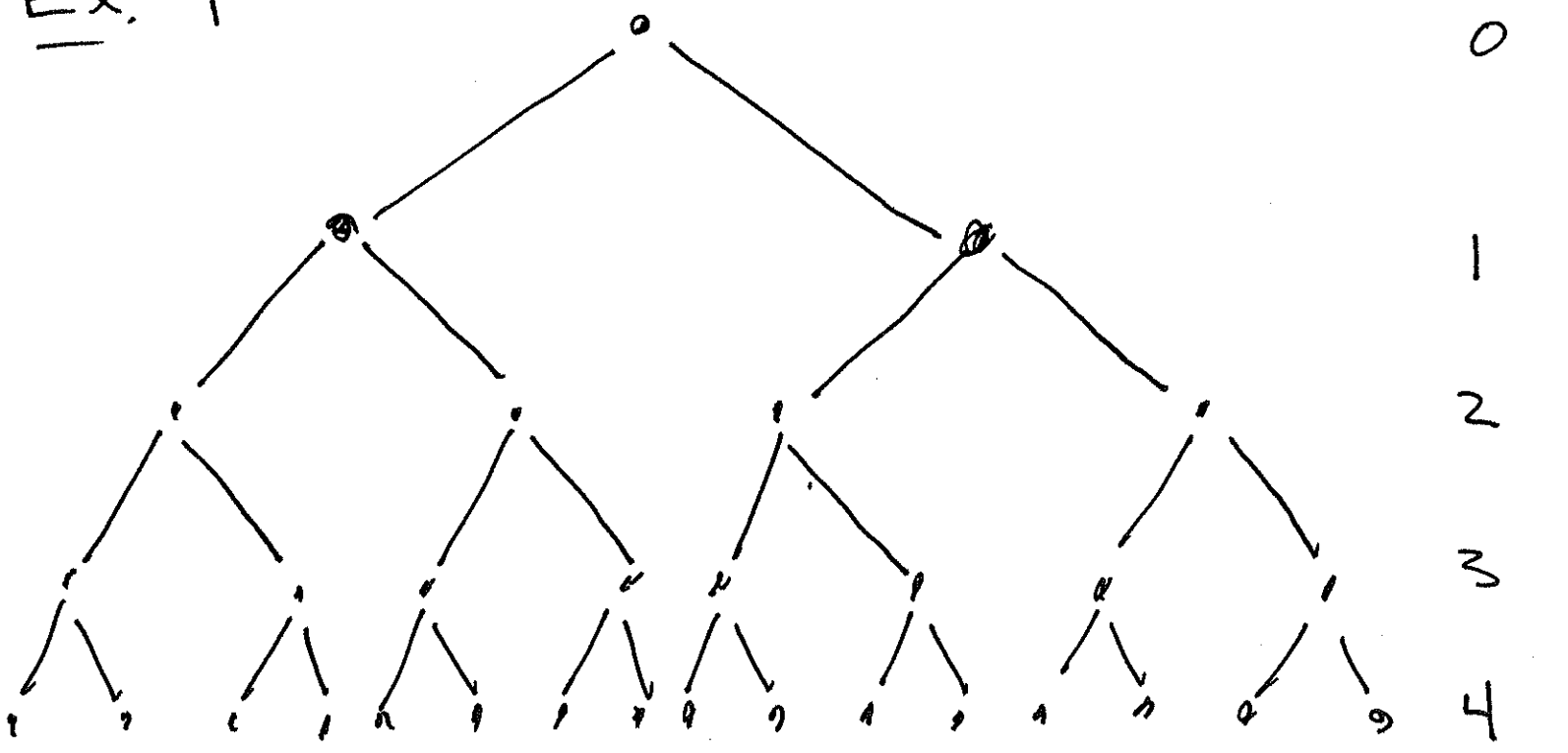
- Insert(S, x)
  adds a new record x to S.
- Max(S)
  returns record with max key
- ExtractMax(S)
  returns and deletes record with max key.
- IncreaseKey(S, x, k)
  change key[x] to k if k > key[x], else does nothing.

Defn

A complete binary tree (CBT)

is a B.T. in which all leaves have same depth, and all internal nodes have 2 children.

Ex. T



Observe

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

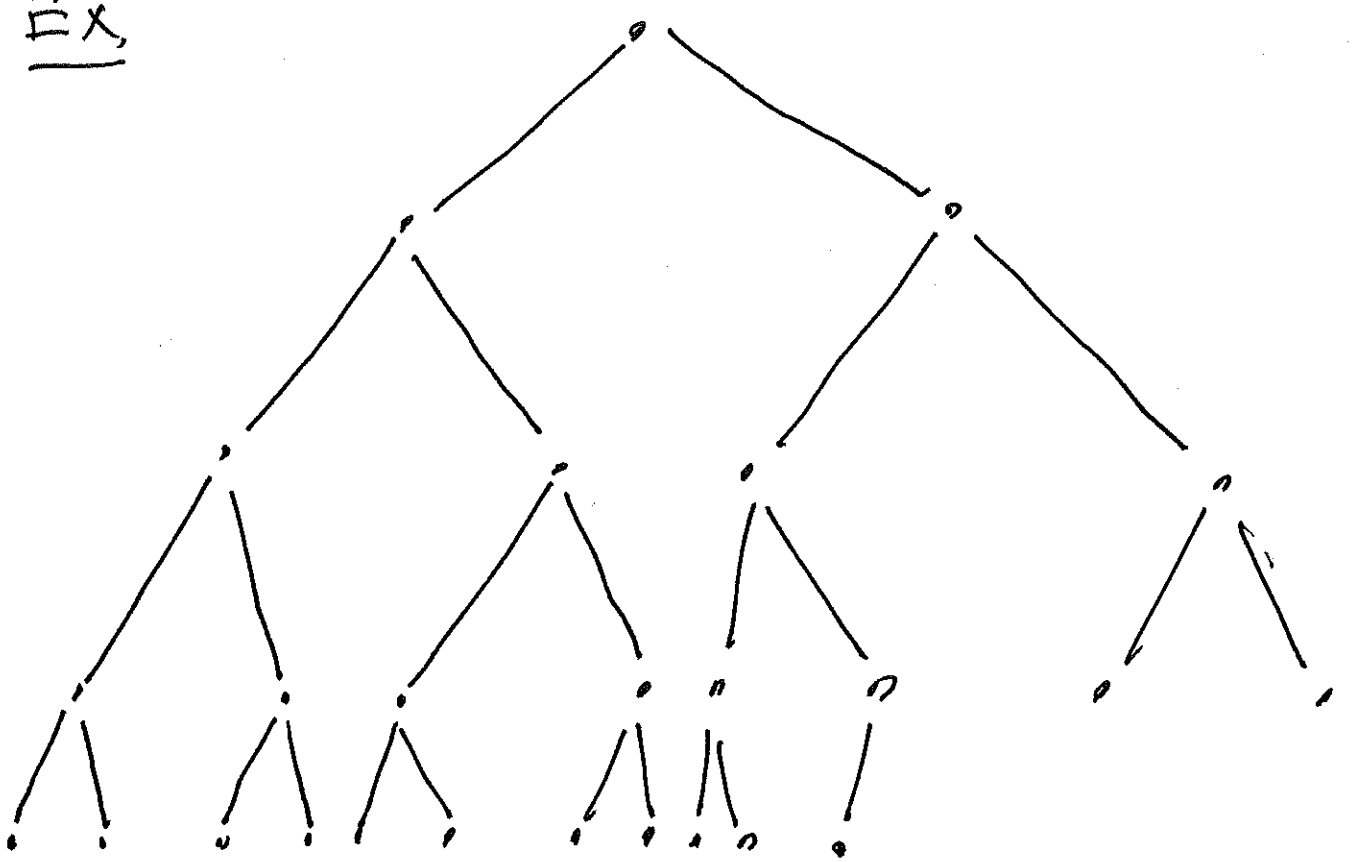
the total # of nodes n is

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

Defn

an <u>almost complete binary tree</u> (ACBT) is a B.T. filled at all levels, except possibly the last (deepest), which is filled left to right.

Ex.



The # of nodes n in an ACBT of height h must satisfy

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

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

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

so $\boxed{h = \lfloor \lg n \rfloor}$

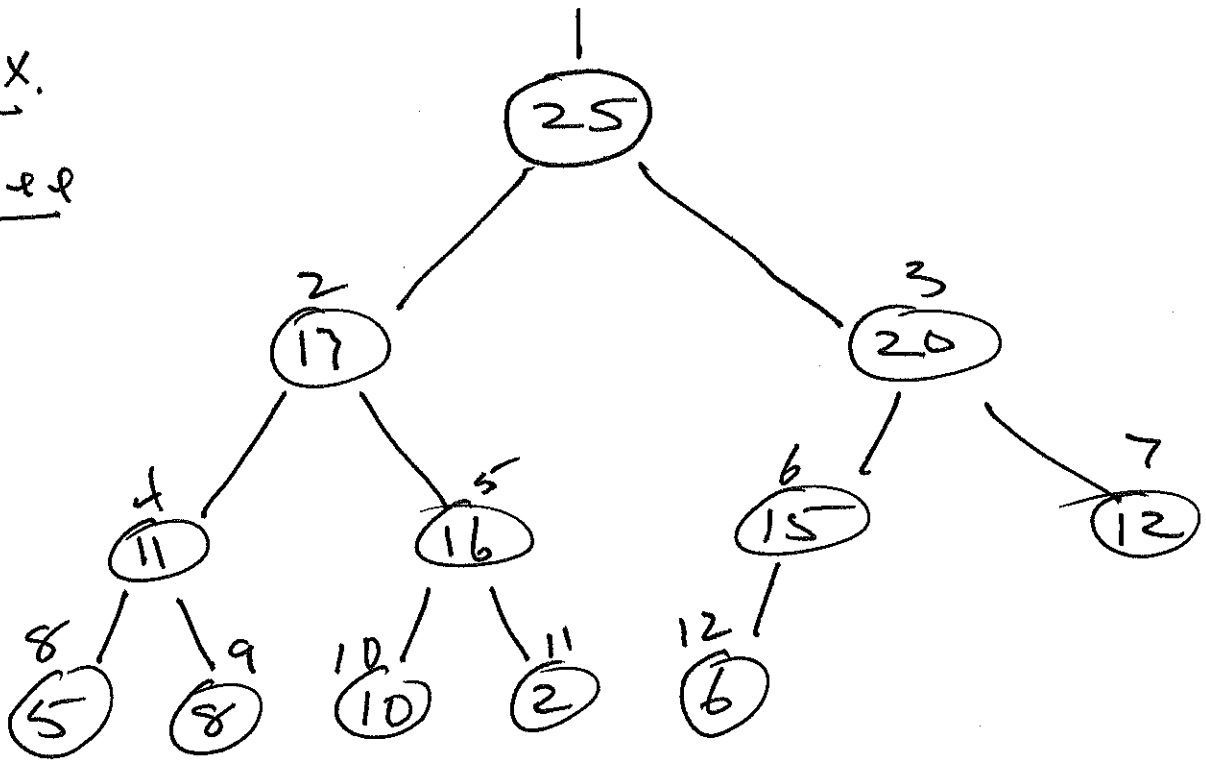
6.1 Heaps

new data struct: Binary heap { min / max *

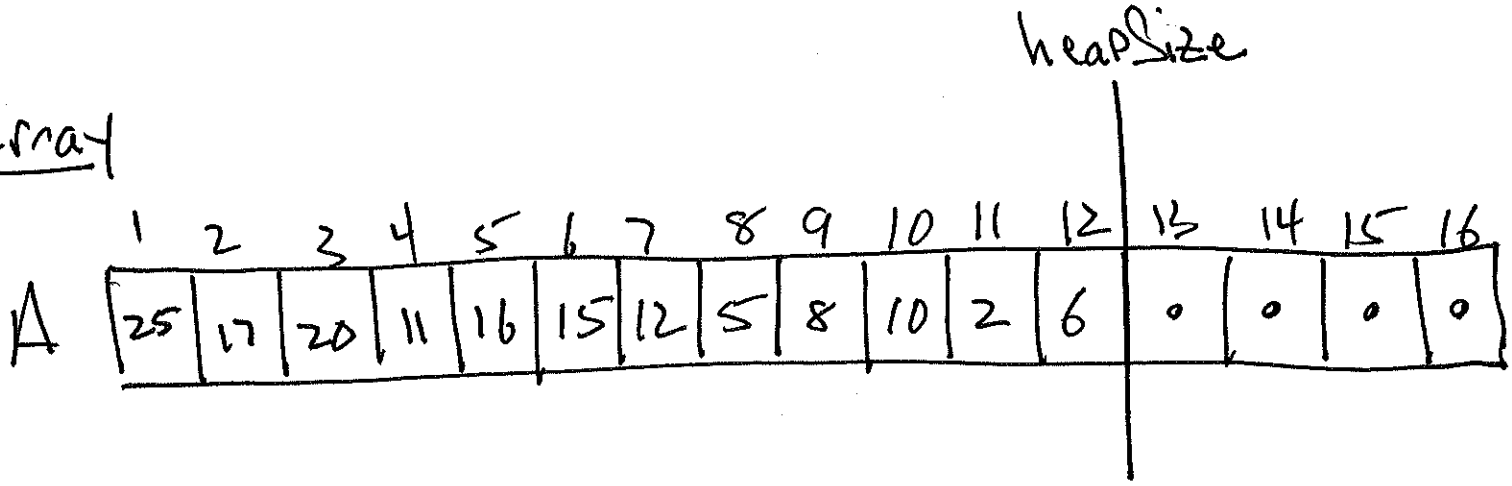
it is an array used to represent an ACBT.

Ex.

tree



Array



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

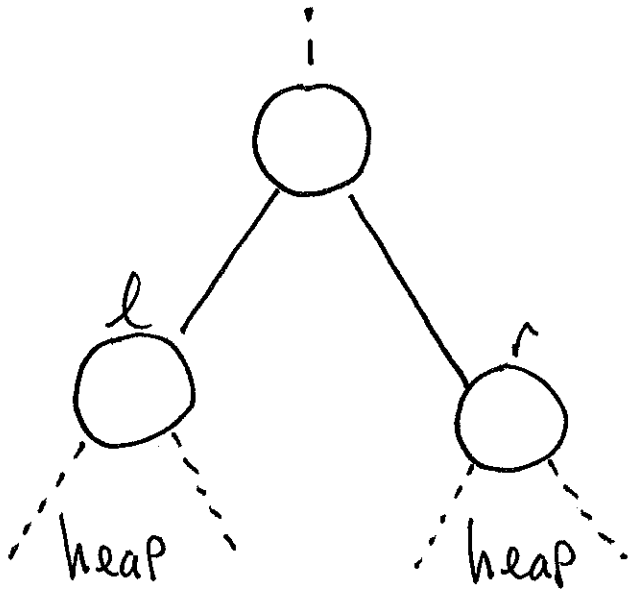
min: $A[Parent(i)] \leq A[i]$

operations on heaps:

- Heapify
- BuildHeap
- Heapsort
- P.Q. OPS: .....

6.2 Heapify (max)

establishes heap property at one node



where $l = left(i)$, $r = right(i)$

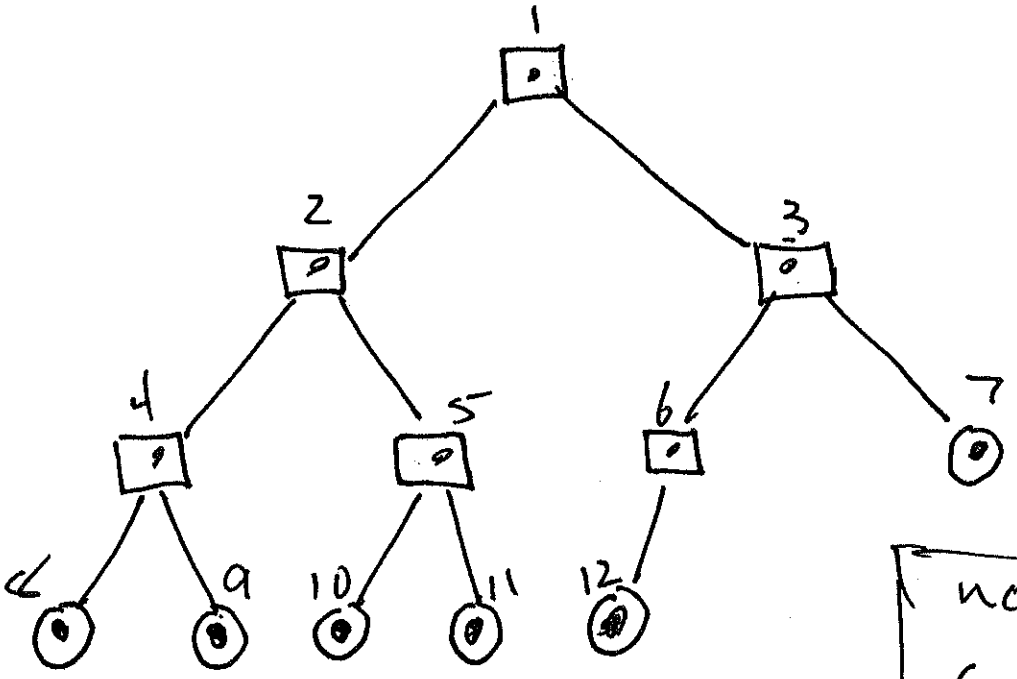
Pre: left and right subtrees are valid heaps

runtime = $\Theta(\log n)$

where n = # nodes in subtree at i

6.3 BuildHeap

Ex.



note: $6 = \lfloor \frac{12}{2} \rfloor$

$A_1, A_2 \ldots A_6 \quad A_7 \ldots A_{12}$

internal nodes | leaves

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

runtime: $\Theta(n)$

6.4 Heapsort

start with an unordered array A.

- Call Heapify(A)
- swap A[1] ⇔ A[heapSize]
- decrament heapsize
- Call Heapify on root.

⋮

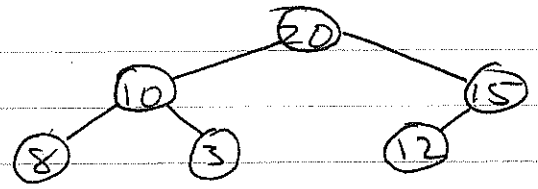
repeat

runtime: $\Theta(n \log n)$

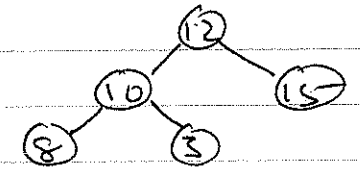
Start: 10 8 12 20 3 15

Ex. After Build-Heap(A):

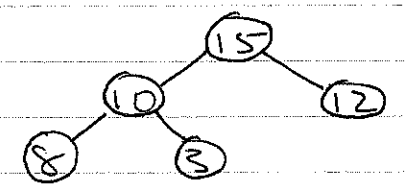
20 10 15 8 3 12 |



12 10 15 8 3 | 20

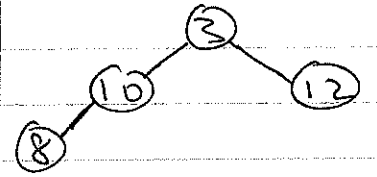


Heapify →

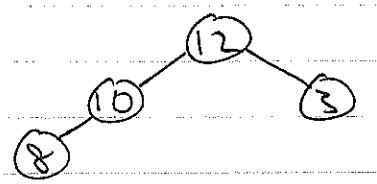


15 10 12 8 3 | 20

3 10 12 8 | 15 20



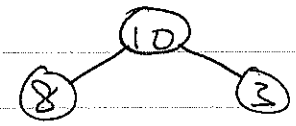
Heapify →



12 10 3 8 | 15 20

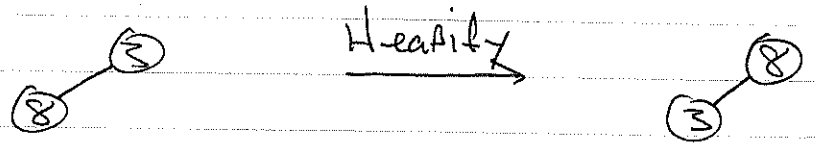
8 10 3 | 12 15 20

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

- HeapMaximum(A)

  cost: $\Theta(1)$

- HeapDeleteMax(A)

  cost: $\Theta(\log n)$

- HeapExtractMax(A)

  cost: $\Theta(\log n)$

- HeapIncreaseKey(A, i, k)

  cost: $\Theta(\log n)$