(1)

6.006 Recitation 9

Agenda
- heaps review.
- ADT vs. implementatio

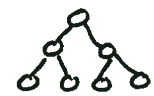
Heaps

idea: a semi organized data structure (typically tree-based)
- organized enough to give decent runtimes
- unorganized enough to make maintainence easy.

ex: lazy laundry - separate underwear, pants, and shirts.

many types - binary, binomial, fibonacci.

→ Binary Heaps : use a binary tree structure



2 types : min-heap, max-heap

invarient: (1) parent ≥ children.

- because this invarient is loose we conforce another

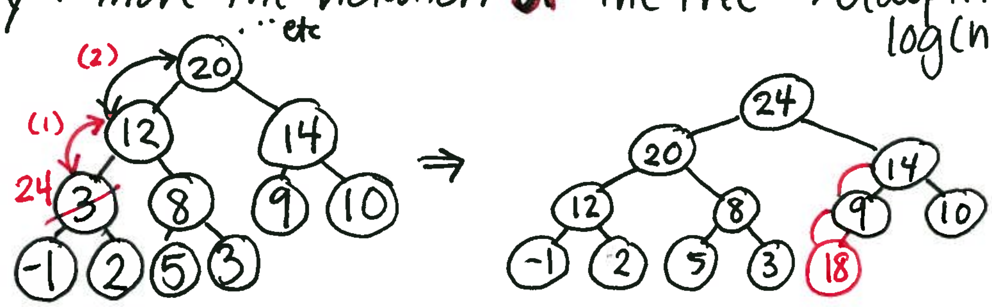
(2) tree must be complete

how is this semi-organized? 2 many valid locations to insert.

→ operations

- extract-max : return root
  heapify(root) $\Rightarrow \Theta(1)$ + heapify
- heapify : move the violation down the tree. $\Rightarrow O(\text{depth} = \log(n))$ b/c complete.
- * increase-key : move the violation UP the tree $\Rightarrow O(\text{depth} = \log(n))$

ex



- insert : add to bottom of heap
  increase-key(new element) $\Rightarrow \Theta(1)$ + incr-key.

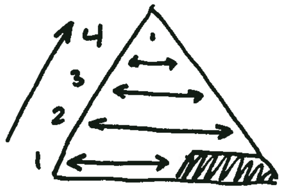
ex above: add 18.

* assumes you know the location of the changing node.

- * decrease-key : change, then heapify node
- * delete : exchange w/ bottom of heap, then heapify

(2)

- build : call heapify on each node, starting at leaves + progressing to the root.



build(node):
    if node is a leaf : return
    build(left-child)
    build(right-child)
    heapify(node)
    return.

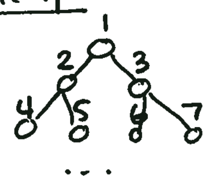
} ensure subtree @ node has only 1 violation: the root.

runtime: every node is heapified
heapify takes $O(\lg n)$
so this is $O(n\lg n)$
lowerbound? i duno.

$T(n) = 2T(n/2) + \lg(n)$
try: $cn - d$ → No $d \geq \lg(n)$
$cn - d\lg n$ → ok if $d \geq 1$
not a lower bound.

Implementing a Binary Heap.

→ array indicies



!! very space efficient and time efficient: can find children/parent fast.

since complete if $i \in$ heap then $j < i$ means $j \in$ heap.

→ doubly linked tree
- just here to show that ADT ≠ implementation.

· array convenient for heapsort.

Priority Queue

ops: insert(x), max(), extract-max(), increase-key(x new)

implementations:

	ins()	max()	extr()	incr()
– sorted list	lg(n)	1	1	lgn
– heaps	lg(n)	1	1	lgn
– unsorted list	1	n	n	1

- to use it you need not know the implementation only the interface.
  - like the ADTs provided in python: list, dict, etc.

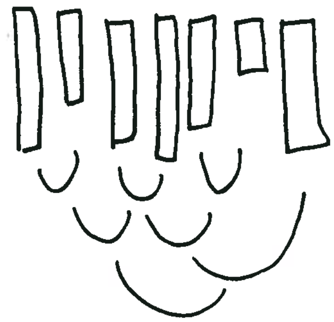
ADT : need to know to **USE**
implement: need to **DO**.

Merging K-lists

Problem: you have k sorted lists and you want 1 sorted list. the total # of elements is n

- popular interview question

IDEA 1: merge pairs of lists



a. merging 2 lists takes $\Theta(m)$
   m = # elts in the lists.
b. you will need $\log(k)$ merges stages
c. each stage touches (roughly) every element.

$O(n \log k)$

⚠: $O(n)$ extra space needed.

IDEA 2: merge all k simultaneously using a min heap.

- take the top element from each list (along w/ keeping track of which list it came from)
- place the elements into a min heap
- remove-min + add to 'sorted' region [look a selection sort!]
- add the next element from the list ↑ came from.

time: building heap $O(k \lg k)$
remove min $O(\lg k)$
add element $O(\lg k)$
– each element will be added and removed so: $n(\lg k)2$

space: $O(k)$

– wait – huh?? well turns out w/ some smartness we can store our sorted list in the sublists free space.

- does having unbalanced lists affect the methods?
- other methods?
- what benefits do each have?
- why a heap rather than a sorted list? since really we're creating a priority Q...

this worked well. acessible problem students had lots of good ideas.