Agenda
- heaps review.
- ADT vs. implementation

Heaps
- Idea: a semi-organized data structure (typically tree-based)
  - organized enough to give decent runtimes
  - unorganized enough to make maintenance 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
  - invariant: parent $\geq$ children.
    - because this invariant is loose we can force another
    - tree must be complete

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

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

  * assumes you know
    - location of the changing node.

  * above: add 18.
  - decrease-key: change, then heapify node
  - delete: exchange w/ bottom of heap, then heapify
build: call heapify on each node, starting at leaves and progressing to the root.

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

runtime: every node is heapified
heapify takes $O(\log n)$
so this is $O(n\log n)$
lower bound? i dunno.

Implementing a Binary Heap:

- array indices
- 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:
- sorted list
  - ins(): $\log(n)$
  - max(): $1$
  - extr(): $1$
  - incr(): $\log(n)$

- heaps
  - ins(): $\log(n)$
  - max(): $1$
  - extr(): $1$
  - incr(): $\log(n)$

- unsorted list
  - ins(): $1$
  - max(): $n$
  - extr(): $n$
  - incr(): $1$

- to use if 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

- A popular interview question

**IDEA 1:** merge pairs of lists

- \( \Theta(m) \) for merging 2 lists
- \( \log(k) \) merges stages
- Each stage touches roughly every element

\[ \Theta(n \log k) \]

Extra space needed: \( O(n) \)

**IDEA 2:** merge all \( k \) simultaneously using a min heap

- Take the top element from each list (along with keeping track of which list it came from)
- Place the elements into a min heap
- Remove-min + add to 'sorted' region [look at selection sort!]
- Add the next element from the list it came from.

Time: building heap \( O(k \log k) \)

- Remove min \( O(\log k) \)
- Add element \( O(\log k) \)

- Each element will be added and removed so: \( n \log k \)

Space: \( O(k) \)

- Wait - huh?? well turns out with 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. An accessible problem students had lots of good ideas.