Lecture 14: Complexity & heaps

- review/catch-up for prof. G.
- heap data structure

Announcements:

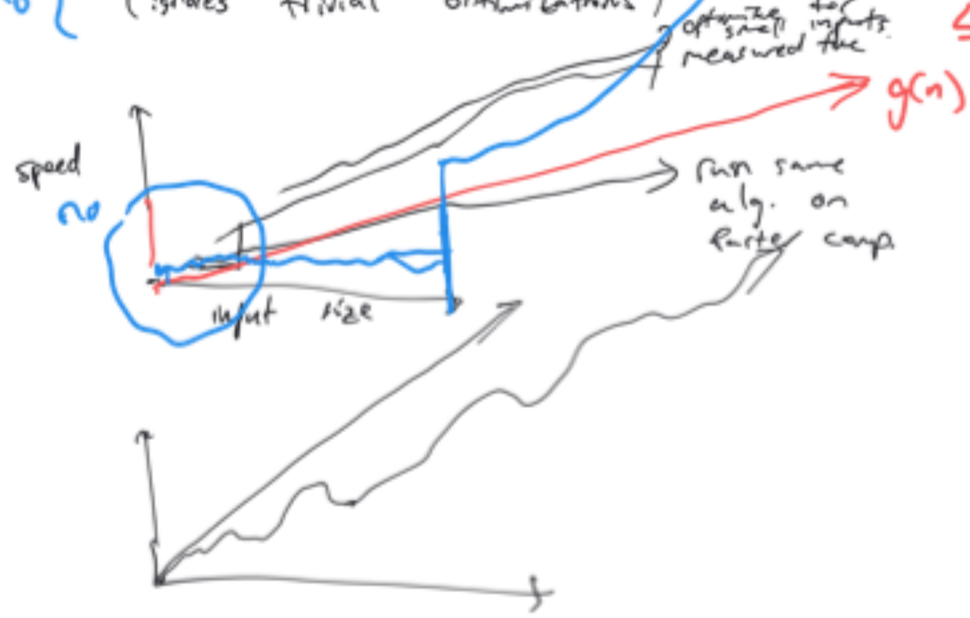
- P3 extension to tomorrow @ 5PM
- Poll/survey coming soon.

f is $O(g)$ means:

c $\left\{ \begin{array}{l} \text{- want to ignore constant speedups} \\ \text{(e.g. run alg. on comp. that's} \\ \text{2x faster)} \end{array} \right.$

n_0 $\left\{ \begin{array}{l} \text{- ignore what happens for small inputs.} \\ \text{(ignores trivial optimizations)} \end{array} \right.$

these are $\le 2 \cdot g(n)$



Consider just upper bounds
(won't take more time than n steps)

f is $O(g)$ means

- there exists constants n_0, c such that
- for any $n > n_0$,

$f(n) \le c\,g(n)$

n:

$\log_2$ not $\log_{10}$

$\uparrow$ # of binary digits $\qquad$ $\uparrow$ # of digits

divide & conquer $\rightarrow$ $\lg n$

n $\leftarrow$ for each input, do something in const. time.

$n \log n$ $\leftarrow$ sorting

n^2

n^3 $\leftarrow$ for each input: for each input: do something

$\vdots$

2^n $\leftarrow$ (prohibitively expensive) search or optimization

$\vdots$

n^n

$8 = 2^3$

input: size 10,000

$\log_{10} 10{,}000$ steps (4 or 5)

- split into 10, recursively handle one (size 1,000)
- split again (size 100)
- split again (size 10)
- base case, size 1

Priority Queue:

- collection:
  - add an item with a priority.
  - remove and return the item with maximum priority.

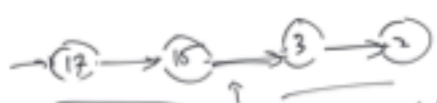
Implementations?

- linked list (store (priority, item) pairs)

  (Item 1, priority 7) -> (Item 2, priority 5)

  - add an item: put @ beginning (constant time)
  - find / remove max priority: search through list. O(n) time.

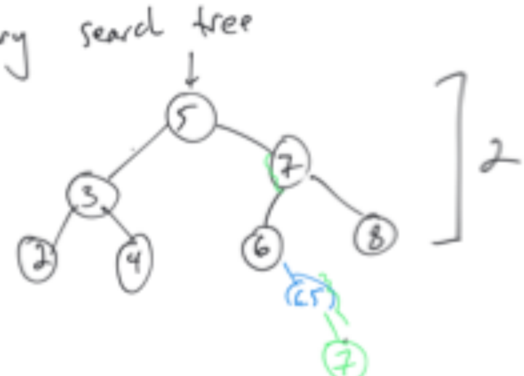
- ordered linked list
  - maximum priority element is always at head, find / remove max is constant time.



  - insert a new item: O(n) in worst case.

  (can't do binary search, can't find middle)

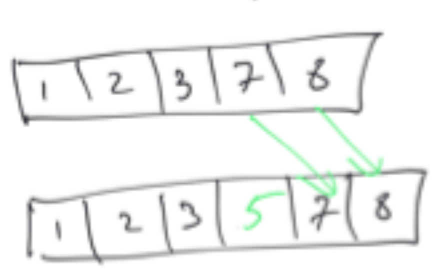
- binary search tree



linear time in worst case (unbalanced)

- hard (expensive) to maintain balance.

- Sorted array
  - Same issues w/ insertion
    need to move elements over to make space for new elt



linear time insert, but can use binary search.

no relation to the Java heap.

MAX-heap
(MIN-heap has smallest @ top)

A **heap** is a binary ~~search~~ tree with the following invariants:

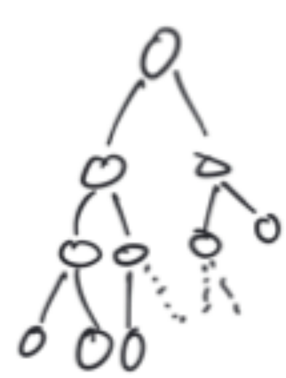
1. the value of every node is greater than (or equal to) the priority of both of its children.

(note: no requirement between siblings).

2. the tree is full.

perfect: all paths have same length

full: all levels are full, except possibly the last, which is filled left-to-right.



How to

- find largest element in a heap?
  it's at the top!

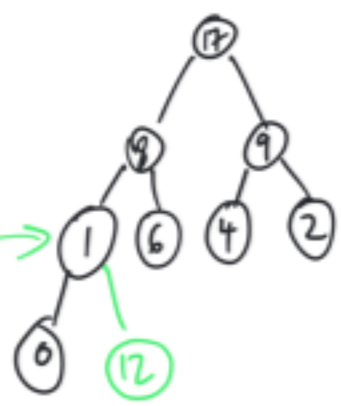
- remove largest element in a heap?

- insert a new element?

① node larger than its children

② heap is full.

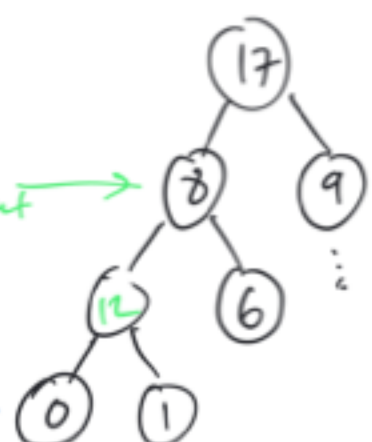
invariant fails!



insert value 12
(ignore 1st invariant)

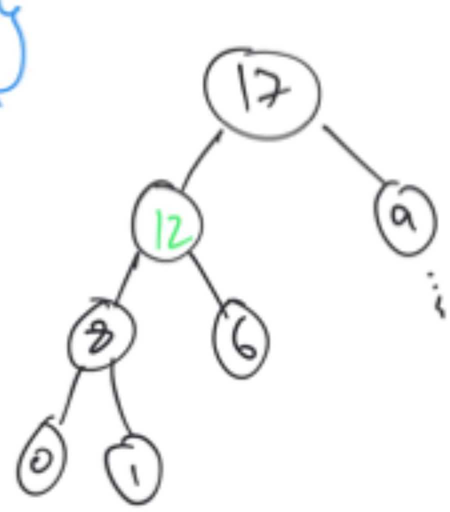
Swap parent of new node & it's two children so biggest is at top.

invariant fails here



swap to maintain invariant

Can't possibly have broken invariant



Since $12 < 17$, both invariants satisfied.