# Sorting #2 - Shell and Heap
# The Two Sorted Towers

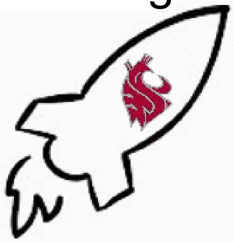
CptS 223 - Fall 2017 - Aaron Crandall

# Today’s Agenda

- Announcements
- Thing of the day
- Sorting is fun! - Doing a more formal heapsort
  - fginnorSstu!

# Announcements

- Feedback from the Google hiring team that visited:
  - They were also quite vehement about the need for code reviews, version co git, and exposure to UML.  Students who don't understand graph traversal, r hashes, and heaps are no-gos.  Apparently they see a lot of students who g these things!
  - So… notice that this is essentially our course (except the engineering aspects)?
- Hardware Hackathon is Saturday. Even if you don't participate, try to swing through and see what people are doing.



# BONUS TOD:
# Detecting Meter Maids with a RPi

- Guy parks on SF 2 hour limited street
- Camera & RPi to photo cars
- Motion detect for photo of cars & stuff
- Uses Tensor flow lib to image recog
- If 75%+ likely it's a meter maid:
    - Send SMS to phone via Twilio
    - Start clock on getting a ticket
- Bonus long term parking via tech
    - [http://peoplesparking.space/](http://peoplesparking.space/)

# Why N^2 lower bound for simple sorts?

- First, define inversions:

  an array of numbers is any ordered pair (i, j) having the property that:

  i < j but a[i] > a[j].
- Sort the example from last section: {34, 8, 64, 51, 32, 21}
  - I: (34, 8), (34, 32), (34, 21), (64, 51), (64, 32), (64, 21), (51, 32), (51, 21), and (32, 21)
  - Exactly the number of swaps needed by insertion sort to sort the list
  - This is *always* the case since each swap fixes one inversion
  - A sorted list has no inversions, so it just needs checking: O(N)

# So... given I inversions in N elements...

- There is always O(N) work involved to test for an ordered list
- Then you add in I inversions, you get O(I + N) for insertion sort
- Thus, you get O(N) running time if there are no inversions
- How many inversions can you expect (on average)?

# Calculating the average number of inversions in a list

- Assuming no duplicates
  - Otherwise, we need a way to estimate duplicates to subtract from total swaps
- Input becomes a permutation of the first N integers, all equally likely
  - Reverse our input list to get: 21, 32, 51, 64, 8, 34 = $L_r$
  - Consider every pair (x,y) with y>x
    - In exactly one of L and $L_r$, the ordered pair is an inversion
  - The total number of pairs in a list L and the reverse becomes: N(N-1)/2
  - The average is half of the total, so N(N-1)/4 inversions
- So… O(N) + O([N^2-N]/4) -> Ω(N^2) min total swaps are required to sort

# Simple sorting algorithm minimum bound

Theorem:
Any algorithm that sorts by exchanging adjacent elements requires (N^2) time on average.

Proof:
The average number of inversions is initially N(N−1)/4 = (N^2). Each swap removes only one inversion, so (N^2) swaps are required.

# Why is this cool?

- Example of a lower bound proof (it’s a minimum work needed proof)
- Valid not only for insertion sort, but the entire class of sorting algorithms
  - Ones that only swap adjacent values
  - Also proves all future algorithms as yet undiscovered
- This proof is relatively simple, but in general lower bounds proofs are tough to do
- Shows that in order for a sorting algorithm to run subquadratic o(N^2), it must do comparisons and exchanges between far apart elements.
  - Means it must eliminate more than one inversion per exchange

# Shell sort (we might do it later in detail...)

- Named after Donald Shell
- Relatively simple algorithm using groups of numbers as sorting blocks
  - First to break the O(N^2) barrier, with a o(N^2) algorithm
  - Does far away comparisons, then shrinks down to insertion sort
  - Highly sensitive to your choices for how to shrink down the distance of the comparisons.
  - Proofs are complex, especially with the compound behavior
- Works great for 10's of thousands of elements
- Easy to code iteratively

# Heapsort!

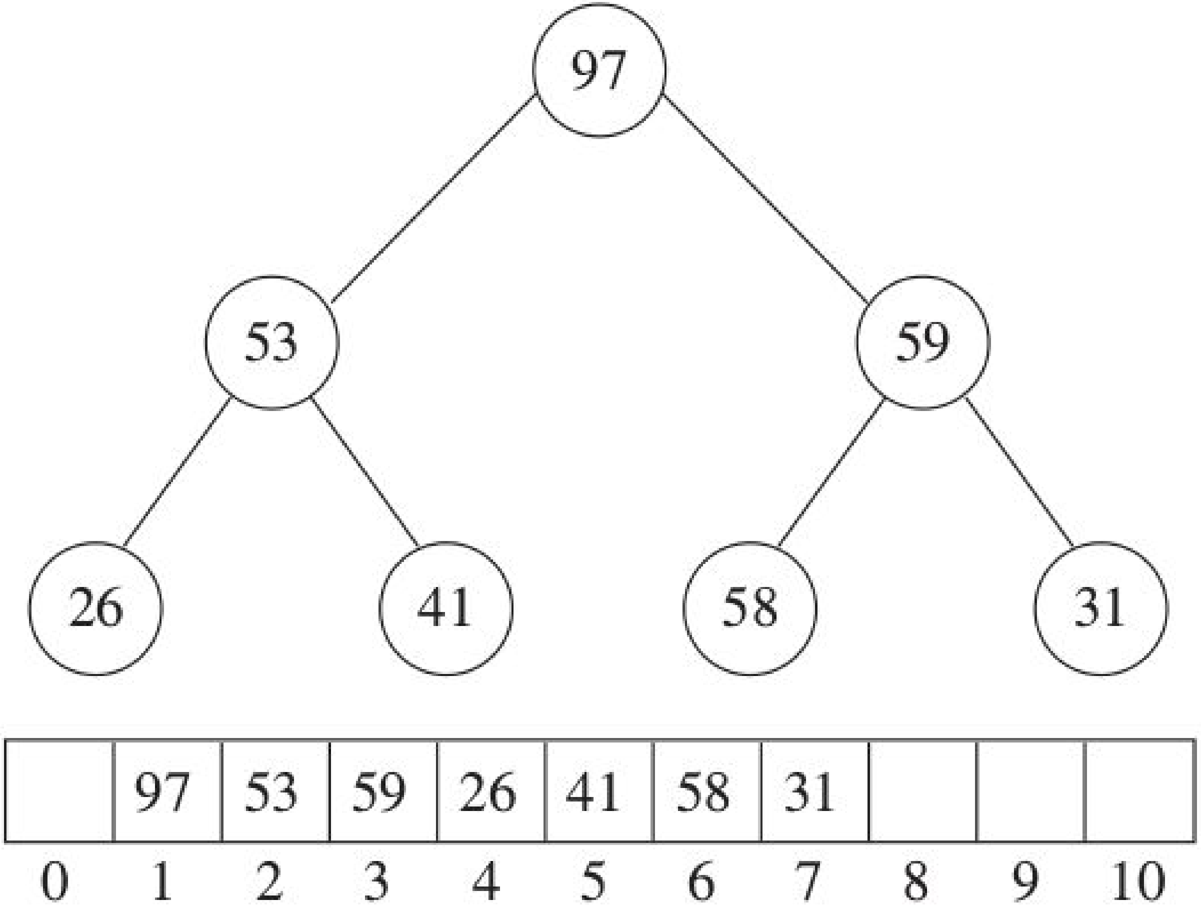
- Basically put your array in a heap, then pull it out at the root
- Works in O(N log N) time
- BuildHeap works in O(N) time
- deleteMin() works in O(log N) time
    - We do this N times
- Total is: BuildHeap + N * deleteMin -> O(N) + O(N * log(N))
    - O(N log N)
- Why not just roll with this and be done seeking more algorithms?

# Because we sometimes forget memory complexity of an algorithm!

- Why is it a problem here?
- You need to store the sorted data in a second array (in the naive implementation), which doubles your memory needs. Space of: O(2N)
- Can be fixed by storing the sorted data in the same array as the heap shrinks down during deleteMin
  - Result is a reverse sorted array
  - Use a minHeap for x>y sorting or a maxHeap for y<x sorting

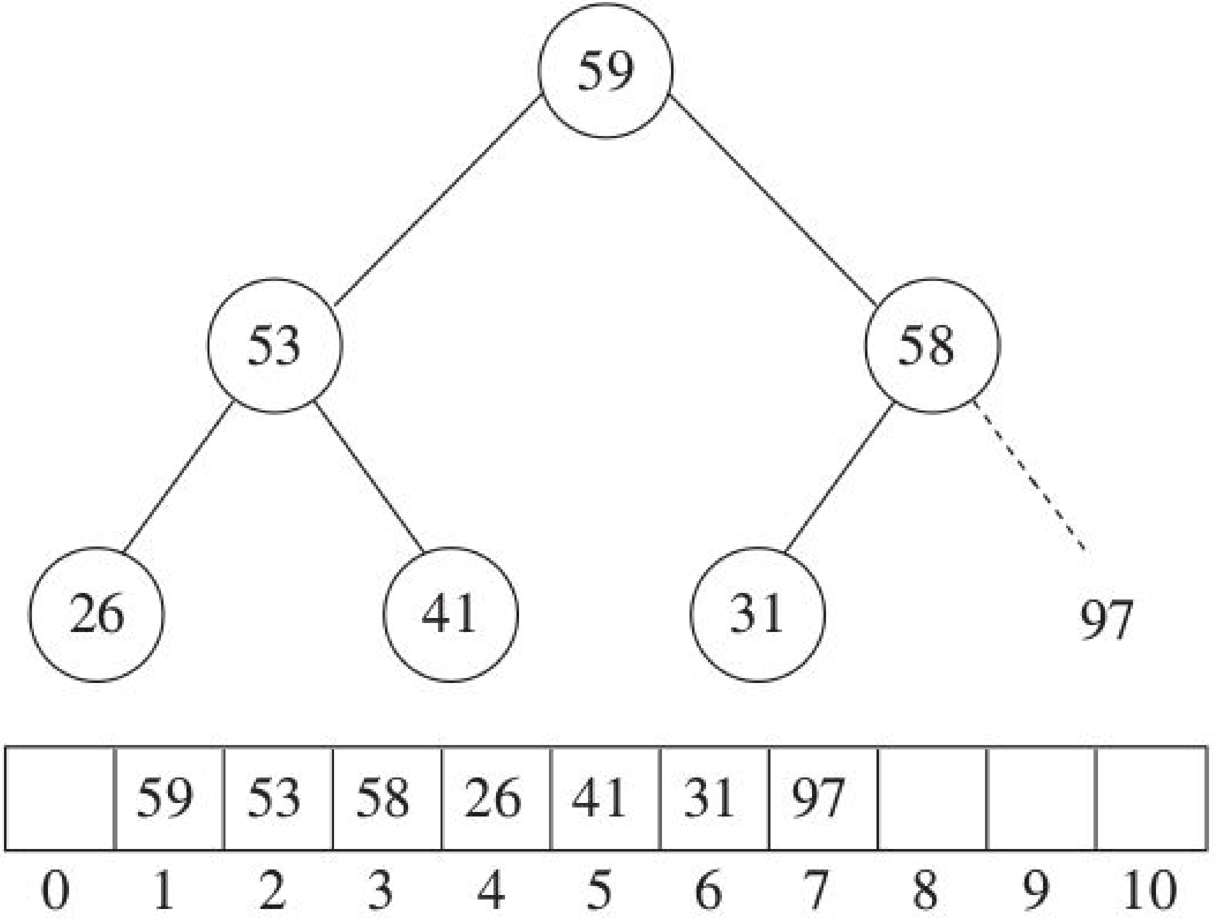
# Example heap

- MaxHeap
- Took O(N) to heapify



# First iteration

- Took O(log N)



# Look at some code!

Note: it’s starting at element 0, not 1.

Their visual example doesn’t do this? Good one.

# Monday: Mergesort

- Mergesort and Quicksort are the two most often used sorts in “real” implementations.
    - Mergesort is the standard in Java
    - Quicksort is the standard in C++