Sorting 7B

N log N Sorts

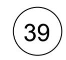
Heap Sort

- We can use a max-heap to sort data.
  - Convert an array to a max-heap.
  - Remove the root from the heap and store it in its proper position in the same array. Repeat until all elements in the array are in sorted order.

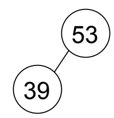
Building the max-heap

ADD NEXT VALUE TO HEAP AND FIX HEAP

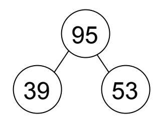
0	1	2	3	4	5	6
39	53	95	72	61	48	83



0	1	2	3	4	5	6
53	39	95	72	61	48	83

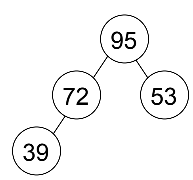


0	1	2	3	4	5	6
95	39	53	72	61	48	83



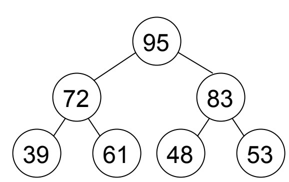
Building the max-heap (cont'd)

0	1	2	3	4	5	6
95	72	53	39	61	48	83



CONTINUE UNTIL THE HEAP IS COMPLETED...

0	1	2	3	4	5	6
95	72	83	39	61	48	53



Sorting from the heap

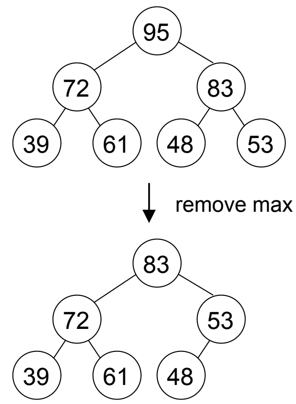
0	1	2	3	4	5	6
95	72	83	39	61	48	53

SWAP THE MAX OF THE HEAP
WITH THE LAST VALUE OF THE HEAP:

0	1	2	3	4	5	6
53	72	83	39	61	48	95

FIX THE HEAP (NOT INCLUDING MAX):

0	1	2	3	4	5	6
83	72	53	39	61	48	95



Sorting from the heap (cont'd)

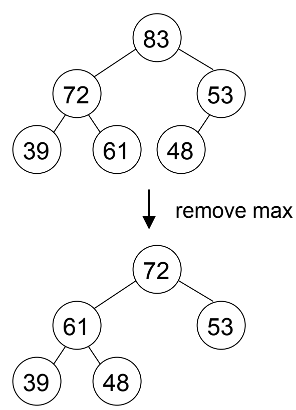
0	1	2	3	4	5	6
83	72	53	39	61	48	95

SWAP THE MAX OF THE HEAP
WITH THE LAST VALUE OF THE HEAP:

0	1	2	3	4	5	6
48	72	53	39	61	83	95

FIX THE HEAP (NOT INCLUDING MAX):

0	1	2	3	4	5	6
72	61	53	39	48	83	95



Sorting from the heap (cont'd)

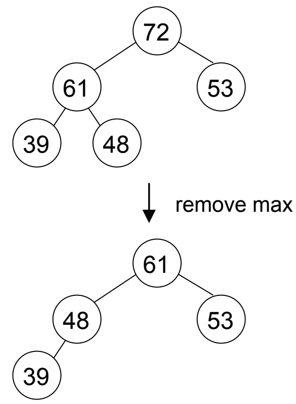
0	1	2	3	4	5	6
72	61	53	39	48	**83**	**95**

SWAP THE MAX OF THE HEAP
WITH THE LAST VALUE OF THE HEAP:

0	1	2	3	4	5	6
48	61	53	39	**72**	**83**	**95**

FIX THE HEAP (NOT INCLUDING THAT MAX):

0	1	2	3	4	5	6
61	48	53	39	**72**	**83**	**95**



REPEAT UNTIL THE HEAP HAS 1 NODE LEFT

Run-Time Analysis

- Building the max-heap
  - Each fix heap is O(log n).
  - There are n elements added to the heap.
  - Building the heap = O(n log n).
- Sorting from the max-heap.
  - Removing max and fixing heap is O(log n).
  - This is done n times.
  - Sorting from the max-heap = O(n log n).

- O(n log n) + O(n log n) = ___________

Divide-and-Conquer Sorts

- Divide the elements to be sorted into two groups of approximately equal size.
- Sort each of these smaller groups.
- Combine the two sorted groups into one large sorted list.

Use recursion to sort the smaller groups.

Merge Sort

- Split the array into two “halves”.
- Sort each of the halves recursively using merge sort.
- Merge the two sorted halves into a new sorted array.
  - Merge sort does <u>not</u> sort in place.

- Example:

66 33 77 55 / 11 99 22 88 44

sort the halves recursively...

33 55 66 77 / 11 22 44 88 99

Merge Sort (cont'd)

Then merge the two sorted halves into a new array:

33 55 66 77 / 11 22 44 88 99
__ __ __ __ __ __ __ __ __

33 55 66 77 / 11 22 44 88 99
11 __ __ __ __ __ __ __ __

33 55 66 77 / 11 22 44 88 99
11 22 __ __ __ __ __ __ __

Merge Sort (cont'd)

33 55 66 77 / 11 22 44 88 99
11 22 33 __ __ __ __ __ __

33 55 66 77 / 11 22 44 88 99
11 22 33 44 __ __ __ __ __

33 55 66 77 / 11 22 44 88 99
11 22 33 44 55 __ __ __ __

Merge Sort (cont'd)

33	55	66	<u>77</u>	/	11	22	44	<u>88</u>	99
11	22	33	44		55	66	__	__	__

44	55	66	77	/	11	22	33	<u>88</u>	99
11	22	33	44		55	66	77	__	__

Once one of the halves has been merged into the new array,
copy the remaining element(s) of the other half into the new array:

44	55	66	77	/	11	22	33	88	99
11	22	33	44		55	66	77	88	99

Run-Time Analysis

- Let T(N) = number of comparisons to sort N elements using merge sort.
  - How many comparisons does it take to sort half of the array? T(N/2)
  - How many comparisons does it take to merge the two halves? N-1 (max.)
- T(N) = 2*T(N/2) + N-1 (a recurrence relation)
- What is the stopping case? T(1) = 0
- Solve for T(N)
  - You will see how to do this in 15-211.
- $T(N) = N \log_2 N - N + 1 = O(N \log N)$

Quick Sort

- Choose a pivot element of the array.
- Partition the array so that
  - the pivot element is in the correct position for the sorted array
  - all the elements to the left of the pivot are less than or equal to the pivot
  - all the elements to the right of the pivot are greater than the pivot
- Sort the subarray to the left of the pivot and the subarray to the right of the pivot recursively using quick sort

Partitioning the array

Arbitrarily choose the first element as the pivot.

66 44 99 55 11 88 22 77 33

Search from the left end for the first element that is greater than the pivot.

66 44 99 55 11 88 22 77 33

Search from the right end for the first element that is less than (or equal to) the pivot.

66 44 99 55 11 88 22 77 33

Now swap these two elements.

66 44 33 55 11 88 22 77 99

Partitioning the array (cont'd)

66 44 33 55 11 88 22 77 99

From the two elements just swapped, search again from the left and right ends for the next elements that are greater than and less than the pivot, respectively.

66 44 33 55 11 88 22 77 99

Swap these as well.

66 44 33 55 11 22 88 77 99

Continue this process until our searches from each end meet.

Partitioning the array (cont'd)

At this point, the array has been partitioned into two subarrays, one with elements less than (or equal to) the pivot, and the other with elements greater than the pivot.

66 44 33 55 11 22 88 77 99

Finally, swap the pivot with the last element in the first subarray section (the elements that are less than the pivot).

22 44 33 55 11 66 88 77 99

Now sort the two subarrays on either side of the pivot using quick sort recursively.

Run-Time Analysis

- Assume the pivot ends up in the center position of the array every time (recursively too).
- Then, quick sort runs in O(N log N) time just like merge sort.
- However, what if the pivot doesn't end up in the center during partitioning?

  Example: Pivot is smallest element. Then we get two subarrays, one of size 0, and the other of size n-1 (instead of n/2 for each).
- Then, quick sort can perform as poorly as $O(n^2)$.

Some Improvements to Quick Sort

- Choose three values from the array, and use the middle element of the three as the pivot.

<u>66</u> 44 99 55 **<u>11</u>** 88 22 77
<u>33</u>

Of 11, 33, 66, use 33 as the pivot.

- As quick sort is called recursively, if a subarray is of "small size", use insertion sort instead of quick sort to complete the sorting to reduce the number of recursive calls.