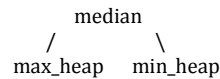


COS 226 – Data Structures and Algorithms
Fall 2014 – Flipped Lecture Section
Group Worksheet week 4 – 10.02.14
25 minutes

- 1. Dynamic median.** Design a data type that supports the following operations
- insert in logarithmic time
 - find-the-median in constant time
 - remove-the-median in logarithmic time.

You may use any ADTs we've discussed in class. This problem may be harder than any you'll find on a 226 exam.

One idea is to create a structure as follows:



Where median is at the root or one of the root elements in max or min heap.

1. insert – if the element is more than median, then insert to min_heap else insert to max_heap ($\lg N$ time)
2. find_median – just return the root element (or one of the children) – constant time
3. remove median – delete the median and promote the next candidate to median (from max or min heap). Adjust the heap in $\lg N$ time

2. Somewhat Leaky Stack.

A leaky stack is a generalization of a stack that supports

- adding a string – $\lg N$ time
- removing the most-recently added string - $\lg N$
- deleting a random string – find a random string and delete in linear time or less

Design a data structure that can perform all operations to meet the performance goals as listed above

insert keys to a PQ in the order they come (most recently will have the highest priority and hence the root element of the heap).

1. when you insert a string, you need to give a priority and insert. Insert takes $\lg N$ worst case time
2. removing the most-recently added will be $\lg N$ time (equivalent to del_max)
3. A random number is generated, but we cannot know where the key is and hence we have to look at all strings (we will discuss a better implementation next week)

3. Heapification.

What is the run-time required to perform a bottom-up sink-based heapification? That is you would sink elements starting from level $(h-1)$ up to level 0 maintaining the heap order property

Bottom up heapification works as follows.

1. assume the binary heap is of height h
2. there are 2^h elements in the lowest level, 2^{h-1} in the level above etc.
3. there are 2^0 or 1 node (root) at level 0
4. Drop the 2^{h-1} elements by at most 1 level to maintain the heap invariant
5. Drop the 2^{h-2} elements by at most 2 level to maintain the heap invariant
6. Continue this until the last element (root) is heapified
7. Total operations required: $2^{h-1} \cdot 1 + 2^{h-2} \cdot 2 + 2^{h-3} \cdot 3 + \dots + 2^{h-h} \cdot h$
8. Show that above sum is roughly $\sim N$ using an integral as an approximation to the discrete sum, where $N = \text{floor}(2^{h+1} - 1)$

4. Largest Common Item

Given an N -by- N matrix of real numbers, find the largest number that appears (at least) once in each row (or report that no such number exists).

9	6	3	8	5
3	5	1	6	8
0	7	5	3	5
3	5	7	8	6
4	3	5	7	9

The running time of your algorithm should be proportional to $N^2 \log N$ in the worst case. You may use extra space proportional to N^2 .

1. Sort each row using heapsort. $(N \lg N) \cdot N$
2. For each number in row 0, from largest to smallest, use binary search to check if it appears in the other $N - 1$ rows. $(\lg N \cdot N)$
3. Return the first number that appears in all N rows.

The order of growth of the running time is $N^2 \log N$, with the bottleneck being steps 1 and 2. Correctness follows because the largest common number must appear in row 0. Scanning the numbers in row 0 from largest to smallest ensures that we find the largest common number.