**<http://cse.iitkgp.ac.in/~abhij/course/theory/Algo1/Autumn13/lab/A5.html>**

**Implementation of Median Heaps**

A max-heap (or a max-priority queue, to be more precise) supports constant-time max finding, and logarithmic-time insertion and deleteMax. A min-heap, on the other hand, supports constant-time min finding, and logarithmic-time insertion and deleteMin. In this exercise, you build a heap (let us call it a **med-heap**) that should support the following three operations:

* Constant-time median finding
* Logarithmic-time insertion
* Logarithmic-time deletion of the median

Using a med-heap, you are supposed to write a sorting function analogous to heap sort.

Let *N* be the current number of elements in a collection (a med-heap). We make the following convention. If *N* = 2*k* + 1 (that is, odd), then the median is the (*k* + 1)-st smallest element in the collection. If *N* = 2*k* (even), then we call the *k*-th smallest element the median of the collection.

A med-heap is realized by maintaining two heaps: a max-heap consisting of the smaller half of the values, and a min-heap consisting of the larger half of the values. The following figure demonstrates how a single array can be used for the contiguous representation of both the heaps back to back.

|  |  |  |
| --- | --- | --- |
| Max-heap of smaller values | Empty space | Min-heap of larger values |
| Direction of growth → | (for future insertions) | ← Direction of growth |

Let *N*1 be the number of elements stored in the max-heap, and *N*2 the number of elements stored in the min-heap. We have *N*1 + *N*2 = *N*. We maintain the convention *N*1 = ⌊*N* / 2⌋ and *N*2 = ⌈*N* / 2⌉. Notice that the max-heap grows from the left, that is, the maximum in this heap can be found at the zeroth index of the array. On the other hand, the min-heap grows from the right end, that is, the minimum in this heap can be found at the (*n* − 1)-st index of the array, where *n* is the total capacity of the array (including the empty space).

Now, write the following functions.

* Write the insert and deleteMax functions for the max-heap, and the insert and deleteMin functions for the min-heap.
* Write the findMedian function for the med-heap. Distinguish between the two cases *N*2 = *N*1 and *N*2 = *N*1 + 1.
* Write the insert function for the med-heap. Suppose that you want to insert *x* in a non-full med-heap. Let *m* be the current median in the med-heap. If *x* ≤ *m*, then you should insert *x* in the max-heap. But then if we already have *N*1 = *N*2, then this insertion will violate the above conventions regarding the new sizes of the two heaps. You therefore shift an element (which one is it?) from the max-heap to the min-heap, and then insert *x* in the max-heap. On the other hand, if *x* > *m*, then insert *x* in the min-heap after a size-adjusting transfer, if necessary.
* Write the deleteMedian function for the med-heap. This boils down to a deletion in the max-heap or in the min-heap depending upon whether *N*2 = *N*1 or *N*2 = *N*1 + 1.
* Write a medHeapSort function that starts with a med-heap filled to the capacity (that is, with no empty space between the max-heap and the min-heap). The function iteratively deletes the current median. Each deletion creates an empty cell. The deleted median is copied there. After all of the *n* elements are deleted, the array should store the sorted list.
* Write a main function which first reads *n* (the capacity of the array) from the user. It first initializes the array to an empty med-heap. Subsequently, it inserts *n* random elements in the med-heap. After each insertion, it prints the current median. After all the elements are inserted, the array is filled to the capacity. Now, the medHeapSort function is called. When it returns, the (sorted) array is to be printed.

**Sample Output**

n = 20

+++ MedHeap initialized

+++ Going to insert elements one by one in MedHeap

Insert(3064) done. Current median = 3064.

Insert( 545) done. Current median = 545.

Insert(2978) done. Current median = 2978.

Insert(5176) done. Current median = 2978.

Insert(7432) done. Current median = 3064.

Insert(2687) done. Current median = 2978.

Insert(9903) done. Current median = 3064.

Insert(7991) done. Current median = 3064.

Insert(7963) done. Current median = 5176.

Insert(6184) done. Current median = 5176.

Insert(5426) done. Current median = 5426.

Insert(9981) done. Current median = 5426.

Insert(8838) done. Current median = 6184.

Insert(8053) done. Current median = 6184.

Insert(1069) done. Current median = 6184.

Insert(2950) done. Current median = 5426.

Insert(3625) done. Current median = 5426.

Insert(9130) done. Current median = 5426.

Insert(8458) done. Current median = 6184.

Insert(9070) done. Current median = 6184.

+++ Median Heap Sort done

545 1069 2687 2950 2978 3064 3625 5176 5426 6184 7432 7963 7991 8053 8458

8838 9070 9130 9903 9981

**Remark:** If our ultimate goal is the medHeapSort function, then we could have done better than inserting *n* elements one by one in the med-heap, which takes O(*n* log *n*) time. What we instead do is just starting with an array fully populated with elements. We then convert the array to a med-heap using a linear-time makeMedHeap function. Such a function can, for example, be implemented if we can *quickly* locate the median in the initial array. A linear-time median-finding algorithm does exist, and will be covered later in the theory course. We then partition the array using the median as pivot as in the quick sort algorithm. Finally, we call makeMaxHeap to convert the smaller half to a max-heap, and makeMinHeap to convert the larger half to a min-heap. The subsequent sequence of *n* deletions would anyway take O(*n* log *n*) time. But it is interesting to know that a linear-time makeHeap algorithm can be designed for med-heaps too.