Report

a) **Explain how to perform the operation getMedian() in O(1) time given such a representation.**

Check sizes for both of queues and if they are equal, then it means we need to find the average value of top values for both of queues (runs in O(1) since size() and peek() operations run in O(1)). If Q- size is greater than Q+ size, then we need to return top value from the Q- (peek() operation runs in O(1)). Otherwise, return top value from the Q+ (peek() runs in O(1)). As you can see, all required operations run in O(1), so the whole function runs in O(1).

IF QMIN.SIZE == QMAX.SIZE **THEN** RETURN AVG (QMIN.TOP , QMAX.TOP)

ELSE IF QMAX.SIZE > QMIN.SIZE **THEN** RETURN QMAX.TOP

ELSE RETURN QMIN.TOP

b) **Explain how to perform the operation S.add(v) In O(log n) time, where n is the current cardinality of the set, while maintaining such a representation.**

If both of queues are empty, it doesn't matter to what queue the value should be added, for example, Q-. If Q+ is empty, but Q- is not empty, add value to Q+, if Q+ is not empty, but Q- is empty, then add value to Q-. Otherwise (if both of queues are not empty), we need to check whether the value is greater than the current median (call getMedian() function) and if so, add value to Q+, otherwise add value to Q-. So far, we used only isEmpty(), add() and getMedian() operations. According to the stated above, getMedian() takes O(1) time, isEmpty() operation in both of queues only checks whether size equals to zero (size is instance variable), so O(1). Method add() in both of queues adds values to the binary heaps, and in binary heaps every addition operation requires O(log n) time since value is compared only with 1 node at each level + heapify operation (also O(log n)). So far, we have O(log n).

Finally, we need to provide rebalance operation. If Q+ size is less than Q- size more than 1, remove top value from Q+ and add it to Q-. If Q- size is less than Q+ size more than 1, remove top value from Q- and add it to Q+. Here we use size () (O(1)), add() (O(log n)), and remove() (O(log n)), so the total time is O (log n).

IF QMAX.EMPTY AND QMIN.EMPTY **THEN** QMAX.ADD

ELSE IF QMIN.EMPTY AND QMAX.NOTEMPTY **THEN** QMIN.ADD

ELSE IF QMIN.NOTEMPTY AND QMAX.EMPTY **THEN** QMAX.ADD

ELSE

IF V > MEDIAN **THEN** QMIN.ADD

ELSE QMAX.ADD

IF QMIN.SIZE – QMAX.SIZE > 1 **THEN** QMAX.ADD <- QMIN.REMOVE

ELSE IF QMAZ.SIZE – QMIN.SIZE > 1 **THEN** QMIN.ADD <- QMAX.REMOVE

c) **Explain how to perform the operation S.deleteMedian() in O(log n) time, where n is the current cardinality of the set, while maintaining such a representation.**

Check whether sum of sizes of queues are even and if so, then it is even cardinality and we need to remove top value from Q-, otherwise remove top value from Q+. Operation size() requires O(1) time, remove() takes O(log n) time (because of heapify function), so the total time for that operation is O (log n).

IF (QMAX.SIZE + QMIN.SIZE) % 2 == 0 **THEN** QMAX.REMOVE

ELSE QMIN.REMOVE

d) **Design and analyze efficient algorithms to support these three operations when S is stored using an array instead of two priority-queues. Compare the theoretical time complexities of the two solutions. You do not need to implement this solution to find median of a set.**

Add operation

ARRAY[CURRENT_INDEX ++] = V

ARRAY.SORT

As you can see above, this operation is pretty simple, it only requires having current index and increment it every time the new value v is added. After that, the array must be sorted. If for sorting array Quick sort algorithm is used, then adding operation requires O (n log n).

Get median operation

IF ARRAY.LENGTH % 2 == 0 **THEN** RETURN (ARRAY[ARRAY.LENGTH / 2] + ARRAY[ARRAY.LENGTH / 2 - 1]) / 2

ELSE RETURN ARRAY[ARRAY.LENGTH / 2]

Here we need to check only array length and if it's even, then return just average value of two central values; if odd – return central value. This operation is simple and takes constant time (O(1)).

Delete median operation

First, we need to find index of median. It can be done using getMedian operation to find median value, and then using binarySearch operation to find index of this value. So far, we have O(log n) in the worst case (binary search) because getMedian runs in constant time (see explanation above). Then we need to remove index form array, so we need to shift all elements after this index in one position left, it will be n / 2. As it's known, n / 2 > O (log n), so the final running time will be O (n / 2).

Theoretically, the array based solution is quite simple to implement. All that is needed is array with the specific size. Two queue based solution is much more complicated since we need firstly to implement heap-based queues. But as per the time complexities, the queues –based solution is efficient since O (log n) < O (n / 2).

e) Implement Q+ using size-balanced Linked-List based heaps.

See MinQueue.java file

f) Implement Q-using array-based heaps.

See MaxQueue.java file

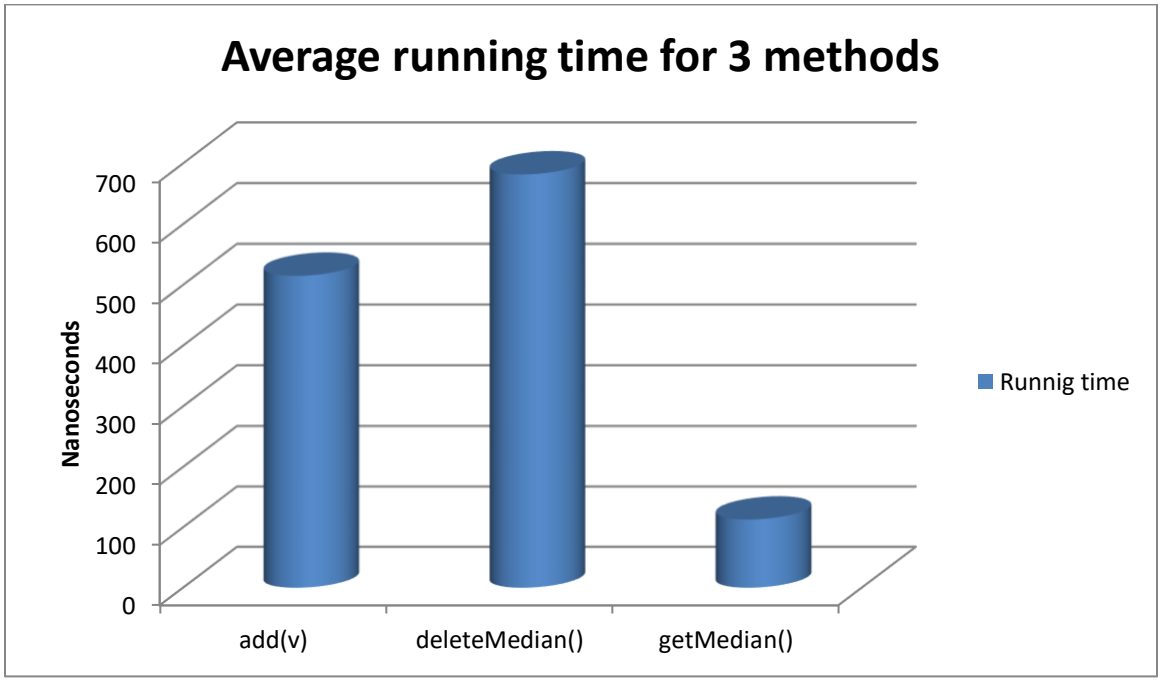
g) Test your implementation of the above "median-finding" data structure using interactive prompts from console, as shown in the Sample Inputs and Outputs below.

```
A - add?
C - check median?
D - delete median?
E - exit?
A
ok, specify the value v to be inserted into S
3
done, value 3 is inserted in Q-,
Inorder traversal of Q- is: 1, 3
Inorder traversal of Q+ is: 15, 5
A - add?
C - check median?
D - delete median?
E - exit?
C
the median is 4
Inorder traversal of Q- is: 1, 3
Inorder traversal of Q+ is: 15, 5
A - add?
C - check median?
D - delete median?
E - exit?
D
lower median 3 is deleted from Q-
Inorder traversal of Q- is: 1
Inorder traversal of Q+ is: 15, 5
A - add?
C - check median?
D - delete median?
E - exit?
D
median 5 is deleted from Q+
Inorder traversal of Q- is: 1
Inorder traversal of Q+ is: 15
A - add?
C - check median?
D - delete median?
E - exit?
E
exit program.
```

h) A large sequence of add()/getMedian()/deleteMedian() is given in an input file hw4input.txt,measure the average time for add(), getMedian() and deleteMedian() operations from this sequence. Do these timings concur with the theoretical time complexities? If not, can you explain why there is discrepancy? For this and the remaining, turn-off the interactive prompts from the console and execute the operations listed in the hw4input.txt file.

Here is the result of average time in nanoseconds (System.nanoTime()):

```
Average time for add(v): 515
Average time for deleteMedian(): 682
Average time for getMedian(): 113
```



As we can see above, the results concur with the theoretical time complexities. Methods add(v) and deleteMedian() take O(log n), whereas getMedian() takes constant time (O(1)), so we see the expected results where time for getMedian() is much less than for other 2 methods.

i) Use the above data structure to implement sortUsingMedians() which returns a sorted array of length n, where n is the current cardinality of the set S as maintained above. Measure the time for sortUsingMedians() –does it concur with O(n log n)

As you can see above, I used two indexes (j and k) to populate left half of sorted array (index j in case of even cardinality), and right half of sorted array (index k in case of even cardinality). For odd cardinality the indexes swapped. Every time I used deleteMedianSort() method (the same method as deleteMedian(), but in this case it returns actual median value, not a string representation how I used it in console program) two times: first time to populate one half of array, second time to populate another half of array. It is obvious that this method iterates over half of elements n and uses deleteMedian for every element, in other words it has O (n / 2 log n) ≈ O (n log n) running time complexity.

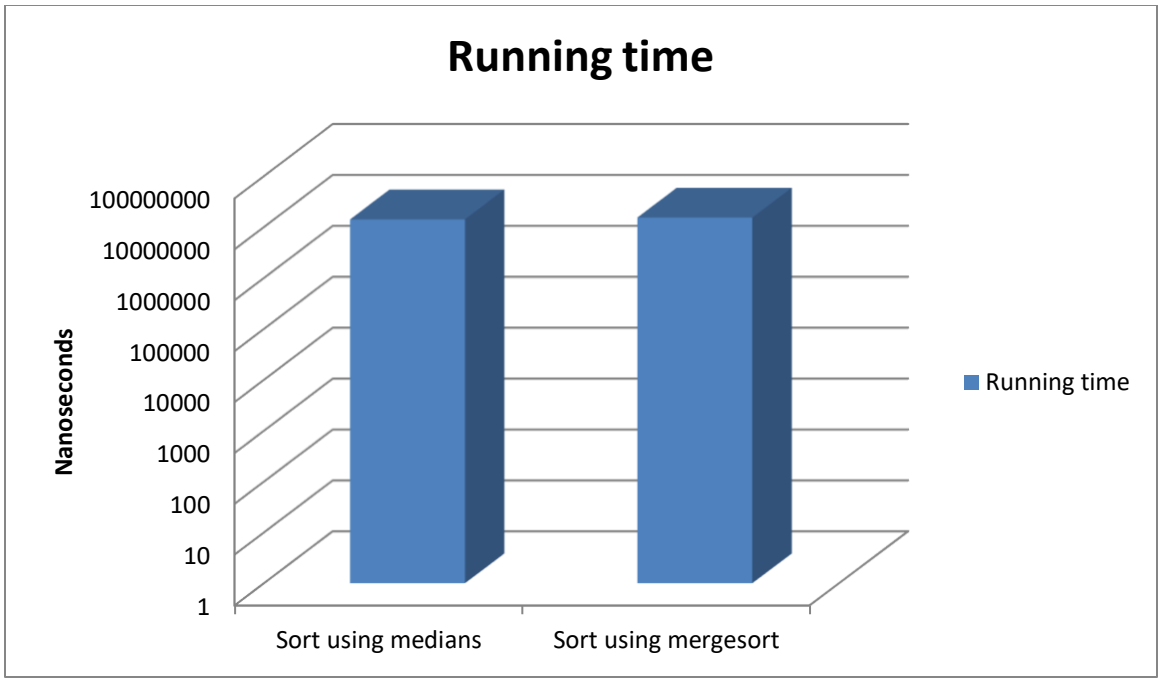
Theoretical	**Empirical**
O (n log n)	O (n / 2 log n)

j) (10 points extra credit): Using the same data values as in hw4input.txtfile, use merge-sort to return a sorted array of length n. Compare this timing with sortUsingMedians()? What do you observe? Compare these two timings with merge-sort's theoretical time complexity O(n log n). What do you observe?

Here is the result:

```
Sort using medians: [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, C
Time: 14120025

Sort using merge sort: [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, C
Time: 15322342
```



As you can see, the time values are almost the same, so this confirms the theoretical running times for both of algorithms (O(n log n).

Theoretical	**Empirical**
O (n log n)	O (n log n)