Please find enclosed my C++ program in answer to your test. There is an automatically generated Makefile in the Debug folder (I'm using Eclipse CDT).

After reading some articles online I decided to use the heap structure to store the numbers as they arrived. The algorithm to find the 10-th smallest number is quite straightforward and is as follows:

1. Store the observed numbers in a max-heap structure until it contains 10 numbers.
2. The 10-th smallest number (at this point the maximum) will be the root of the max-heap.
3. Continue observing numbers, numbers greater than or equal to the current 10-th smallest number can be immediately disregarded, numbers less than the current 10-th smallest number are added to the heap. After a new number is added to the heap, the root (which will be the current 11-smallest number) can be removed.

This algorithm is implemented in the more general class `KthSmallestNumber` - where on construction 'k' is chosen (of course here k = 10). Using templates the class also allows for a general type of data to be entered, `double`, `int`, etc.

The algorithm to calculate the median uses two heaps, a max-heap and a min-heap, and is a little more complex. The algorithm is as follows:

1. Store the first value in the max-heap.
2. We then alternate between adding a number to the max-heap (even current total observations) and min-heap (odd current total observations) so that the heaps are the same size, or the max-heap is 1 bigger.
3. If the current total observations are odd, the min-heap is going to increase in size by 1. However, if the new observation is less than the root of the max-heap, it must go into the max-heap, the root of the max-heap is then moved to the min-heap. Otherwise, the new observation is simply placed directly in the min-heap.
4. If the current total observations are even, the max-heap is going to increase in size by 1. However, if the new observation is bigger than the root of the min-heap, it must go into the min-heap, the root of the min-heap is then moved to the max-heap. Otherwise, the new observation is simply placed directly in the max-heap.
5. If the two heaps are the same size (total number of observations even)

the median is the mean of the root of each of the two heaps. If the max-heap is bigger (odd total number of observations) the median is the root of the max-heap.

This algorithm is implemented in the class `Median`, which used templated in a similar fashion to `KthSmallestNumber`.

The algorithm is probably best viewed via an example as in Figure 1 (source: `http://allenlipeng47.com/PersonalPage/index/view/84/nkey`).

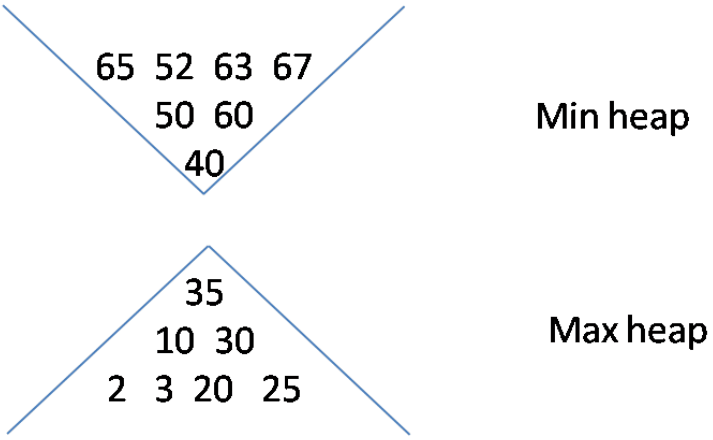

Figure 1: Example of the min-heap and max-heap when the total number of observations is even

It should be noted that if the input is an integer, the k-th smallest number will also be an integer, however, the median should be stored as a double, at least in the usual case when the median is the mean of the two most central numbers when the number of observations are even.

For me the main design issues in this task were simplicity and speed. After reading about the heap structure (which I was completely unaware of before yesterday), and the various C++ methods (of which I used `push_heap`, `pop_heap`, `pop_back` and `front`) I knew the algorithm would be fairly simple to implement, and I hope the code is quite readable. I also found some benchmarks showing the speed of the algorithms as in Figure 2 (source: `https://www.daniweb.com/software-development/cpp/threads/469821/fast-insertion-into-a-sorted-list`).

I am sure there are some more efficient, state-of-the-art algorithms for calculating the median, but implementing them is probably out of scope for this test (please correct me if I am wrong). As far as I can tell this is a very ef-

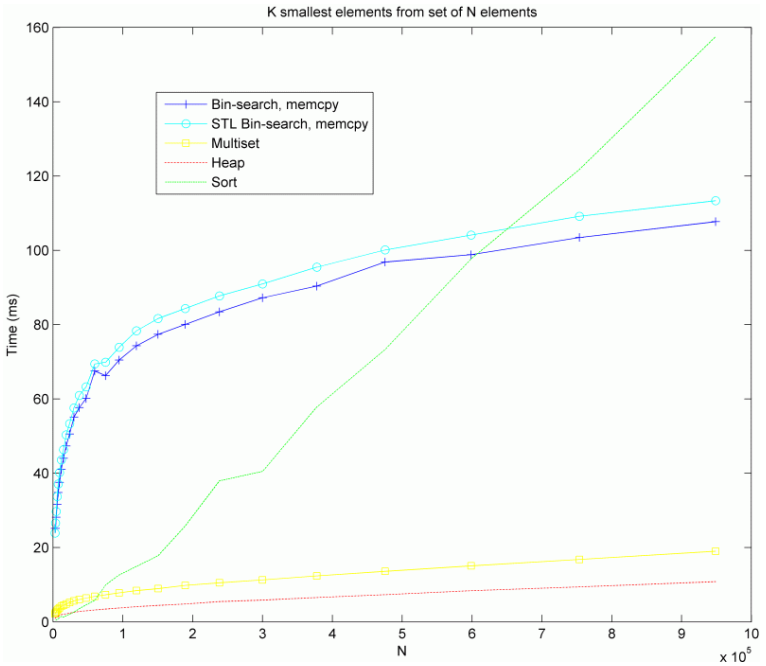

Figure 2: Benchmarks of the k smallest elements from a set of N elements

ficient algorithm (some more here: `http://www.cs.umd.edu/~meesh/351/mount/lectures/lect14-heapsort-analysis-part.pdf`).

Besides passing the usual optimisation flag (`-O3`), I didn't make any specific alterations to the algorithm in order to increase performance. I have tried to be as efficient as I could in the code I have written, however. If we were only calculating the statistics after a block of numbers, a block insert to the heaps may be more efficient than one-by-one as is current. Of course there would be some speed increase in that we simply calculate the statistics less and print to the console less.

Most interesting for me, would be to approximate the median. I have glanced at some algorithms for doing this for example count-min sketch (`https://sites.google.com/site/countminsketch/`) which seems to be highly praised. As a statistician however, I'd try to exploit anything I could about the numbers which were being input. For example if the numbers followed a distribution that was symmetric, the mean is an approximation to the median. Alternatively, the input values may approximately follow some other distribution, we could estimate/update the distribution parameters with each new addition of data, and then calculate the theoretical distribution median.