295. Find Median From Data Stream

Feb. 6, 2017 | 164.2K views

Average Rating: 4.82 (239 votes)

Median is the middle value in an ordered integer list. If the size of the list is even, there is no middle value. So the median is the mean of the two middle value.

For example,
`[2,3,4]`, the median is `3`

`[2,3]`, the median is `(2 + 3) / 2 = 2.5`

Design a data structure that supports the following two operations:

- void addNum(int num) - Add a integer number from the data stream to the data structure.
- double findMedian() - Return the median of all elements so far.

Example:

```
addNum(1)
addNum(2)
findMedian() -> 1.5
addNum(3)
findMedian() -> 2
```

Follow up:

1. If all integer numbers from the stream are between 0 and 100, how would you optimize it?
2. If 99% of all integer numbers from the stream are between 0 and 100, how would you optimize it?

Solution

Approach 1: Simple Sorting

Intuition

Do what the question says.

Algorithm

Store the numbers in a resize-able container. Every time you need to output the median, sort the container and output the median.

C++ Copy

```cpp
class MedianFinder {
    vector<int> store;

public:
    // Adds a number into the data structure.
    void addNum(int num)
    {
        store.push_back(num);
    }

    // Returns the median of current data stream
    double findMedian()
    {
        sort(store.begin(), store.end());

        int n = store.size();
        return (n & 1 ? store[n / 2] : ((double) store[n / 2 - 1] + store[n / 2]) * 0.5);
    }
};
```

Complexity Analysis

- Time complexity: $O(n \log n) + O(1) \simeq O(n \log n)$.
  - Adding a number takes amortized $O(1)$ time for a container with an efficient resizing scheme.
  - Finding the median is primarily dependent on the sorting that takes place. This takes $O(n \log n)$ time for a standard comparative sort.
- Space complexity: $O(n)$ linear space to hold input in a container. No extra space other than that needed (since sorting can usually be done in-place).

Approach 2: Insertion Sort

Intuition

Keeping our input container always sorted (i.e. maintaining the sorted nature of the container as an *invariant*).

Algorithm

Which algorithm allows a number to be added to a sorted list of numbers and yet keeps the entire list sorted? Well, for one, **insertion sort!**

We assume that the current list is already sorted. When a new number comes, we have to add it to the list while maintaining the sorted nature of the list. This is achieved easily by finding the correct place to insert the incoming number, using a **binary search** (remember, the list is *always sorted*). Once the position is found, we need to shift all higher elements by one space to make room for the incoming number.

This method would work well when the amount of insertion queries is lesser or about the same as the amount of median finding queries.

C++ Copy

```cpp
class MedianFinder {
    vector<int> store; // resize-able container

public:
    // Adds a number into the data structure.
    void addNum(int num)
    {
        if (store.empty())
            store.push_back(num);
        else
            store.insert(lower_bound(store.begin(), store.end(), num), num);     // binary search and insertion combined
    }

    // Returns the median of current data stream
    double findMedian()
    {
        int n = store.size();
        return n & 1 ? store[n / 2] : ((double) store[n / 2 - 1] + store[n / 2]) * 0.5;
    }
```

Complexity Analysis

- Time complexity: $O(n) + O(\log n) \approx O(n)$.
  - Binary Search takes $O(\log n)$ time to find correct insertion position.
  - Insertion can take up to $O(n)$ time since elements have to be shifted inside the container to make room for the new element.

> **Pop quiz:** Can we use a *linear* search instead of a *binary* search to find insertion position, without incurring any significant runtime penalty?

- Space complexity: $O(n)$ linear space to hold input in a container.

Approach 3: Two Heaps

Intuition

The above two approaches gave us some valuable insights on how to tackle this problem. Concretely, one can infer two things:

1. If we could maintain direct access to median elements at all times, then finding the median would take a constant amount of time.
2. If we could find a reasonably fast way of adding numbers to our containers, additional penalties incurred could be lessened.

But perhaps the most important insight, which is not readily observable, is the fact that we *only* need a consistent way to access the median elements. Keeping the *entire* input sorted is **not a requirement.**

> Well, if only there were a data structure which could handle our needs.

As it turns out there are two data structures for the job:

- Heaps (or Priority Queues [1])
- Self-balancing Binary Search Trees (we'll talk more about them in Approach 4)

Heaps are a natural ingredient for this dish! Adding elements to them take logarithmic order of time. They also give direct access to the maximal/minimal elements in a group.

If we could maintain *two* heaps in the following way:

- A max-heap to store the smaller half of the input numbers
- A min-heap to store the larger half of the input numbers

This gives access to median values in the input: they comprise the top of the heaps!

Wait, what? How?

If the following conditions are met:

1. Both the heaps are balanced (or nearly balanced)
2. The max-heap contains all the smaller numbers while the min-heap contains all the larger numbers

then we can say that:

1. All the numbers in the max-heap are smaller or equal to the top element of the max-heap (let's call it x)
2. All the numbers in the min-heap are larger or equal to the top element of the min-heap (let's call it y)

Then x and/or y are smaller than (or equal to) almost half of the elements and larger than (or equal to) the other half. That is *the* definition of **median** elements.

This leads us to a huge point of pain in this approach: **balancing the two heaps!**

Algorithm

- Two priority queues:
  1. A max-heap `lo` to store the smaller half of the numbers
  2. A min-heap `hi` to store the larger half of the numbers
- The max-heap `lo` is allowed to store, at worst, one more element more than the min-heap `hi`. Hence if we have processed k elements:
  - If $k = 2 * n + 1 \quad (\forall n \in \mathbb{Z})$, then `lo` is allowed to hold $n + 1$ elements, while `hi` can hold n elements.
  - If $k = 2 * n \quad (\forall n \in \mathbb{Z})$, then both heaps are balanced and hold n elements each.

  This gives us the nice property that when the heaps are perfectly balanced, the median can be derived from the tops of both heaps. Otherwise, the top of the max-heap `lo` holds the legitimate median.
- Adding a number `num`:
  - Add `num` to max-heap `lo`. Since `lo` received a new element, we must do a balancing step for `hi`. So remove the largest element from `lo` and offer it to `hi`.
  - The min-heap `hi` might end holding more elements than the max-heap `lo`, after the previous operation. We fix that by removing the smallest element from `hi` and offering it to `lo`.

  The above step ensures that we do not disturb the nice little size property we just mentioned.

A little example will clear this up! Say we take input from the stream `[41, 35, 62, 5, 97, 108]`. The run-though of the algorithm looks like this:

```
Adding number 41
MaxHeap lo: [41]           // MaxHeap stores the largest value at the top (index 0)
MinHeap hi: []             // MinHeap stores the smallest value at the top (index 0)
Median is 41
=======================
Adding number 35
MaxHeap lo: [35]
MinHeap hi: [41]
Median is 38
=======================
Adding number 62
MaxHeap lo: [41, 35]
MinHeap hi: [62]
Median is 41
=======================
Adding number 4
MaxHeap lo: [35, 4]
MinHeap hi: [41, 62]
Median is 38
=======================
Adding number 97
MaxHeap lo: [41, 35, 4]
MinHeap hi: [62, 97]
Median is 41
=======================
Adding number 108
MaxHeap lo: [41, 35, 4]
MinHeap hi: [62, 97, 108]
Median is 51.5
```

C++ Copy

```cpp
class MedianFinder {
    priority_queue<int> lo;                              // max heap
    priority_queue<int, vector<int>, greater<int>> hi;   // min heap

public:
    // Adds a number into the data structure.
    void addNum(int num)
    {
        lo.push(num);                                    // Add to max heap

        hi.push(lo.top());                               // balancing step
        lo.pop();

        if (lo.size() < hi.size()) {                     // maintain size property
            lo.push(hi.top());
            hi.pop();
        }
    }

    // Returns the median of current data stream
    double findMedian()
    {
        return lo.size() > hi.size() ? lo.top() : ((double) lo.top() + hi.top()) * 0.5;
    }
};
```

Complexity Analysis

- Time complexity: $O(5 \cdot \log n) + O(1) \approx O(\log n)$.
  - At worst, there are three heap insertions and two heap deletions from the top. Each of these takes about $O(\log n)$ time.
  - Finding the mean takes constant $O(1)$ time since the tops of heaps are directly accessible.
- Space complexity: $O(n)$ linear space to hold input in containers.

Approach 4: Multiset and Two Pointers

Intuition

Self-balancing Binary Search Trees (like an AVL Tree) have some *very* interesting properties. They maintain the tree's height to a logarithmic bound. Thus inserting a new element has reasonably good time performance. The median **always** winds up in the root of the tree and/or one of its children. Solving this problem using the same approach as Approach 3 but using a Self-balancing BST seems like a good choice. Except the fact that implementing such a tree is not trivial and prone to errors.

Why reinvent the wheel? Most languages implement a `multiset` class which emulates such behavior. The only problem remains keeping track of the median elements. That is easily solved with **pointers!** [2]

We maintain two pointers: one for the lower median element and the other for the higher median element. When the total number of elements is odd, both the pointers point to the same median element (since there is only one median in this case). When the number of elements is even, the pointers point to two consecutive elements, whose mean is the representative median of the input.

Algorithm

- Two iterators/pointers `lo_median` and `hi_median`, which iterate over the `data` multiset.
- While adding a number `num`, three cases arise:
  1. The container is currently **empty.** Hence we simply insert `num` and set both pointers to point to this element.
  2. The container currently holds an **odd** number of elements. This means that both the pointers currently point to the same element.
     - If `num` is not equal to the current median element, then `num` goes on either side of it. Whichever side it goes, the size of that part increases and hence the corresponding pointer is updated. For example, if `num` is less than the median element, the size of the lesser half of input increases by 1 on inserting `num`. Thus it makes sense to decrement `lo_median`.
     - If `num` is equal to the current median element, then the action taken is dependent on how `num` is inserted into `data`. **NOTE:** In our given C++ code example, `std::multiset::insert` inserts an element *after* all elements of equal value. Hence we increment `hi_median`.
  3. The container currently holds an **even** number of elements. This means that the pointers currently point to consecutive elements.
     - If `num` is a number between both median elements, then `num` becomes the new median. Both pointers must point to it.
     - Otherwise, `num` increases the size of either the lesser or higher half of the input. We update the pointers accordingly. It is important to remember that both the pointers ***must*** point to the same element now.
- Finding the median is easy! It is simply the **mean** of the elements pointed to by the two pointers `lo_median` and `hi_median`.

C++ Copy

```cpp
class MedianFinder {
    multiset<int> data;
    multiset<int>::iterator lo_median, hi_median;

public:
    MedianFinder()
        : lo_median(data.end())
        , hi_median(data.end())
    {
    }

    void addNum(int num)
    {
        const size_t n = data.size();   // store previous size

        data.insert(num);               // insert into multiset

        if (!n) {
            // no elements before, one element now
            lo_median = hi_median = data.begin();
        }
        else if (n & 1) {
            // odd size before (i.e. lo == hi), even size now (i.e. hi = lo + 1)

            if (num < *lo_median)       // num < lo
                lo_median--;
            else                        // num >= hi
```

A much shorter (but harder to understand), ***one*** *pointer* version [3] of this solution is given below:

C++ Copy

```cpp
class MedianFinder {
    multiset<int> data;
    multiset<int>::iterator mid;

public:
    MedianFinder()
        : mid(data.end())
    {
    }

    void addNum(int num)
    {
        const int n = data.size();
        data.insert(num);

        if (!n)                                 // first element inserted
            mid = data.begin();
        else if (num < *mid)                    // median is decreased
            mid = (n & 1 ? mid : prev(mid));
        else                                    // median is increased
            mid = (n & 1 ? next(mid) : mid);
    }

    double findMedian()
    {
        const int n = data.size();
        return ((double) *mid + *next(mid, n % 2 - 1)) * 0.5;
```

Complexity Analysis

- Time complexity: $O(\log n) + O(1) \approx O(\log n)$.
  - Inserting a number takes $O(\log n)$ time for a standard `multiset` scheme. [4]
  - Finding the mean takes constant $O(1)$ time since the median elements are directly accessible from the two pointers.
- Space complexity: $O(n)$ linear space to hold input in container.

Further Thoughts

There are so many ways around this problem, that frankly, it is scary. Here are a few more that I came across:

- **Buckets!** If the numbers in the stream are statistically distributed, then it is easier to keep track of buckets where the median would land, than the entire array. Once you know the correct bucket, simply sort it find the median. If the bucket size is significantly smaller than the size of input processed, this results in huge time saving. @mitbbs8080 has an interesting implementation here.
- **Reservoir Sampling.** Following along the lines of using buckets: if the stream is statistically distributed, you can rely on Reservoir Sampling. Basically, if you could maintain just one good bucket (or *reservoir*) which could hold a representative sample of the entire stream, you could estimate the median of the entire stream from just this one bucket. This means good time and memory performance. Reservoir Sampling lets you do just that. Determining a **"good"** size for your reservoir? *Now, that's a whole other challenge.* A good explanation for this can be found in this StackOverflow answer.
- **Segment Trees** are a great data structure if you need to do a lot of insertions or a lot of read queries over a limited range of input values. They allow us to do all such operations *fast* and in roughly the *same amount of time*, **always.** The only problem is that they are far from trivial to implement. Take a look at my introductory article on Segment Trees if you are interested.
- **Order Statistic Trees** are data structures which seem to be tailor-made for this problem. They have all the nice features of a BST, but also let you find the k^{th} order element stored in the tree. They are a pain to implement and no standard interview would require you to code these up. But they are fun to use if they are already implemented in the language of your choice. [5]

1. Priority Queues queue out elements based on a predefined priority. They are an abstract concept and can, as such, be implemented in many different ways. Heaps are an efficient way to implement Priority Queues.
2. Shout-out to @pharese for this approach.
3. Inspired from this post by @StefanPochmann.
4. Hinting can reduce that to amortized constant $O(1)$ time.
5. **GNU** `libstdc++` users are in luck! Take a look at this StackOverflow answer.

Rate this article:

Previous Next

Comments: 53 Sort By

Type comment here... (Markdown is supported)

Preview Post

yuxiong 1049 November 16, 2018 5:28 AM
I'm so glad that some of the solutions (like the ones in this post) weren't written by awice. He is pretty smart and good at coding. But he failed to make solutions/explanations understandable.
115 Share Reply
SHOW 3 REPLIES

lnmlv 68 October 26, 2018 2:29 AM
any thoughts about the follow-ups?
35 Share Reply
SHOW 14 REPLIES

itsiashu 16 July 29, 2019 8:37 AM
Understand and implemented the two heaps approach very well. My only concern, if it's really a data stream, can we really use heap methods what if servers start running out of memory? or think of this how scalable heaps solution is? And if it's not what's the point of having a question for datastream and solving it with Heaps rather than just use vectors/list to store, sort and find the mean?
16 Share Reply
SHOW 4 REPLIES

andy_d 30 February 14, 2019 1:09 AM
Would be nice to see some comments on the follow ups mentioned in the description. A very nice write up otherwise especially when considering some of the more recent ones from contributors
14 Share Reply

JustGoCrazy 15 February 23, 2019 1:08 AM
Can someone tell me why the median is always the root and/or one of its children in a balanced BST?
I can find an anti-example: [4, 2, 5, 1, 3] where the median 3 is neither the root or one of its children
10 Share Reply
SHOW 3 REPLIES

amby_leet_code 9 July 3, 2018 5:47 AM
Thank you so much for the article. Sharing my Java implementation for the Two Heaps approach described here.

```
class MedianFinder {
```

Read More
9 Share Reply
SHOW 2 REPLIES

dd2233 223 June 20, 2018 4:17 AM
Very nice question and analysis @babhishek21

I would like to point out a data structure that allows you to do the "Insertion Sort" algorithm with O(log n): https://en.wikipedia.org/wiki/Skip_list

Read More
7 Share Reply
SHOW 1 REPLY

kremebrulee 52 October 22, 2018 12:11 AM
There's a bug in the first solution:
return (n & 1 ? (store[n / 2 - 1] + store[n / 2]) * 0.5 : store[n / 2]);

The if else statements are flipped.
5 Share Reply
SHOW 1 REPLY

luce 296 October 3, 2019 4:23 AM
Approach 1 is actually O(N * Nlog(N)). Worst case is if I call find median after each new inserted number.
You need to insert via. binary search during the addNum function to achieve O(Nlog(N).
4 Share Reply

kolarsur 20 March 6, 2019 9:55 PM
Any inputs on the follow up mentioned in the problem description?
3 Share Reply
SHOW 1 REPLY

‹ 1 2 3 4 5 6 ›