295. Find Median from Data Stream

Hard | Design | Two Pointers | Data Stream | Sorting | Heap (Priority Queue)

Problem Description

The problem requires designing a class, `MedianFinder`, that can handle a stream of integers and provide a way to find the median at any given time. Median, by definition, is the middle value in an ordered list of numbers. If the list has an odd number of elements, the median is simply the middle element. If the list has an even number of elements, there is no single middle element, so the median is calculated by taking the average of the two middle numbers.

Intuition

To find the median efficiently, we need a data structure that allows quick access to the middle elements. Utilizing two heaps is an elegant solution: a max heap to store the smaller half of the numbers and a min heap to store the larger half. This way, the largest number in the smaller half or the smallest number in the larger half can easily give us the median.

In Python, the default `heapq` module provides a min heap implementation. To get a max heap behavior, we insert negatives of the numbers into a heap.

By balancing the heaps so that their size differs by at most one, we ensure that we either have a single middle element when the combined size is odd (this will be at the top of the larger heap), or two middle elements when the combined size is even (the top of each heap).

The `addNum` method works by adding a new number to the max heap (smaller half) first, by pushing its negative value. We then pop the top value from the max heap and push it to the min heap (larger half) to maintain the order and balance. If the larger half has more than one extra element compared to the smaller half, we move the top element from the larger half to the smaller half.

`findMedian` checks the current size of the heaps. If the heaps are of the same size, the median is the average of the top values of both heaps. If the sizes differ, the median is the top element of the larger heap.

Solution Approach

The solution is based on maintaining two heaps (`h1` and `h2`), which are used to simulate a max heap and a min heap respectively.

- `h1` (max heap) stores the smaller half of the numbers, and we simulate it by pushing negatives of the numbers into a min heap.
- `h2` (min heap) stores the larger half of the numbers as they are, since the `heapq` library already implements a min heap.

Here's how the methods of the `MedianFinder` class work:

- The `__init__` method simply initializes two empty lists, `h1` and `h2`, that we'll use as heaps.
- `addNum` adds a new number to the data structure:
  1. We first add the number to `h1` (which is simulating a max heap) by pushing its negative.
  2. Then, we balance the max heap by popping an element from `h1` and pushing it onto `h2` (min heap).
  3. We check if `h2` (larger half) has more than one element than `h1` (smaller half). If it does, we balance the heaps by moving the top element from `h2` to `h1`.
- `findMedian` computes the median based on the current elements:
  1. If `h2` has more elements than `h1`, the median is just the top element of `h2` (the smallest number in the larger half).
  2. If `h1` and `h2` have the same number of elements, the median is the average of the top element of `h1` (the largest number in the smaller half, remember `h1` is storing negatives) and the top element of `h2` (the actual smallest number in the larger half).

The efficiency of adding numbers is `O(log n)` due to the heap operations, and finding the median is `O(1)` since it involves only accessing the tops of the heaps. This solution is quite efficient and handles the streaming data well.

Example Walkthrough

Let's walk through a small example to illustrate the implementation of the `MedianFinder` class using the given solution approach:

1. Initialize the `MedianFinder` class, so we start with two empty heaps, `h1` (max heap) and `h2` (min heap).
2. We execute `addNum(3)`. Here's how it works:
   - We insert -3 into `h1` to keep track of the max heap.
   - We pop -3 from `h1` (which is 3 in its original form) and push it to `h2`.
   - Since `h1` is now empty and `h2` has only one element, no balancing is needed.
3. Next, we execute `addNum(1)`:
   - We insert -1 into `h1`.
   - We pop -1 from `h1` (which is 1 in its original form) and push it to `h2`.
   - Now `h2` has two elements (1 and 3), we pop 1 from `h2` and push its negative into `h1` to balance the heaps.
4. At this point, `h1` has the single element -1, and `h2` has the single element 3.
5. We add another number by calling `addNum(5)`:
   - We insert -5 into `h1`.
   - We pop -5 from `h1` (which is 5 in its original form) and push it into `h2`.
   - `h2` now having 3 and 5, is larger than `h1` which only has -1. No additional balancing is needed.
6. Now, when calling `findMedian()`:
   - Since we have a total of 3 elements and `h1` has 1 element and `h2` has 2 elements, the median is the top of `h2` which is 3.
7. Let's add another number by executing `addNum(2)`:
   - We insert -2 into `h1`.
   - We pop -2 from `h1` (which is 2 in its original form) and push it to `h2`.
   - Since `h2` (with elements 2, 3, and 5) now has more elements, we pop 2 from `h2` and push its negative into `h1` to balance the heaps. Now `h1` has -2 and -1, and `h2` has 3 and 5.
8. Calling `findMedian()` now gives us:
   - Since `h1` and `h2` have the same number of elements (2 each), the median is the average of the top element of `h1` (-1, which is 1 when we negate it) and the top element of `h2` (which is 3), giving us a median of (1 + 3) / 2 = 2.

Through these steps, we can see how the `MedianFinder` class handles the addition of numbers and maintains a structure that allows us to easily find the median at any point. The use of two heaps is instrumental in efficiently managing the median calculation for a stream of data.

Python Solution

```python
from heapq import heappush, heappop

class MedianFinder:
    def __init__(self):
        """
        Initialize your data structure here.
        """
        self.small = []  # Max heap (simulated with negative values)
        self.large = []  # Min heap

    def addNum(self, num: int) -> None:
        """
        Adds a number into the data structure.
        """
        heappush(self.small, -num)  # Add to max heap
        # Move the largest element of small (max heap) to large (min heap)
        heappush(self.large, -heappop(self.small))

        # If large heap has more elements, move the smallest element of large to small
        if len(self.large) > len(self.small):
            heappush(self.small, -heappop(self.large))

    def findMedian(self) -> float:
        """
        Returns the median of current data stream
        """
        # If the heaps have equal size, the median is the average of the two heap's top elements
        if len(self.small) == len(self.large):
            return (-self.small[0] + self.large[0]) / 2.0
        # If the small heap has more elements, return the top element of the small heap (which is the max element)
        elif len(self.small) > len(self.large):
            return -self.small[0]
        # Otherwise, return the top element of the large heap (which is the min element)
        else:
            return self.large[0]

```

Java Solution

```java
import java.util.PriorityQueue;
import java.util.Collections;

class MedianFinder {
    private PriorityQueue<Integer> minHeap = new PriorityQueue<>(); // Min-heap to store the larger half of the numbers
    private PriorityQueue<Integer> maxHeap = new PriorityQueue<>(Collections.reverseOrder()); // Max-heap to store the smaller half c

    /** Initialize MedianFinder. */
    public MedianFinder() {
        // The constructor is kept empty as there's nothing to initialize outside the declarations.
    }

    /** Adds a number into the data structure. */
    public void addNum(int num) {
        minHeap.offer(num); // Add the number to the min-heap
        maxHeap.offer(minHeap.poll()); // Balance the heaps by moving the smallest number of min-heap to max-heap

        // Ensure max-heap has equal or one more element than the min-heap
        if (maxHeap.size() > minHeap.size() + 1) {
            minHeap.offer(maxHeap.poll()); // Move the maximum number of max-heap to min-heap
        }
    }

    /** Returns the median of current data stream. */
    public double findMedian() {
        if (maxHeap.size() > minHeap.size()) {
            // If max-heap has more elements, the median is the top of the max-heap
            return maxHeap.peek();
        }
        // Otherwise, the median is the average of the tops of both heaps
        return (minHeap.peek() + maxHeap.peek()) / 2.0;
    }
}

/**
 * Example of usage:
 * MedianFinder medianFinder = new MedianFinder();
 * medianFinder.addNum(num); // Add a number
 * double median = medianFinder.findMedian(); // Get the current median
 */

```

C++ Solution

```cpp
#include <queue>
#include <vector>

class MedianFinder {
public:
    MedianFinder() {
        // Constructor doesn't need any code,
        // as the priority queues are automatically initialized.
    }

    // Inserts a number into the data structure.
    void addNum(int num) {
        // Add the new number to the max heap.
        maxHeap.push(num);

        // Now balance the heaps by always having the top of the max heap
        // ready to move to the min heap to maintain the ordering.
        minHeap.push(maxHeap.top());
        maxHeap.pop();

        // If the min heap has more elements than the max heap,
        // move the top element back to the max heap to maintain
        // size property of heaps.
        if (minHeap.size() > maxHeap.size()) {
            maxHeap.push(minHeap.top());
            minHeap.pop();
        }
    }

    // Finds and returns the median of all numbers inserted.
    double findMedian() {
        // If the max heap is larger, the median is at the top of the max heap.
        if (maxHeap.size() > minHeap.size()) {
            return maxHeap.top();
        }
        // If both heaps have the same size, the median is the average of
        // the tops of both heaps.
        return (double) (minHeap.top() + maxHeap.top()) / 2;
    }

private:
    // Max heap for the lower half of the numbers.
    std::priority_queue<int> maxHeap;
    // Min heap for the upper half of the numbers
    // (using greater<> to make it a min heap).
    std::priority_queue<int, std::vector<int>, std::greater<int>> minHeap;
};

/**
 * The MedianFinder object will be instantiated and called as such:
 * MedianFinder* obj = new MedianFinder();
 * obj->addNum(num); // Method to add a number into the MedianFinder.
 * double median = obj->findMedian(); // Method to find and return the median.
 */

```

Typescript Solution

```typescript
class MedianFinder {
    private small: number[]; // Simulated Max Heap (we'll use Min Heap logic but with negated values)
    private large: number[]; // Min Heap

    constructor() {
        this.small = [];
        this.large = [];
    }

    addNum(num: number): void {
        // Add to max heap (simulated by adding negative number to min heap)
        this.heappush(this.small, -num);
        // Ensure the smallest of the max heap moves to the min heap
        this.heappush(this.large, -this.heappop(this.small));

        // Balance the heaps so that either both have the same size or small has one more element
        if (this.large.length > this.small.length) {
            this.heappush(this.small, -this.heappop(this.large));
        }
    }

    findMedian(): number {
        if (this.small.length > this.large.length) {
            // If the small heap is larger, the median is its top element (remember to negate it back)
            return -this.small[0];
        } else if (this.small.length === this.large.length) {
            // If the heaps are of equal size, the median is the average of the tops of both heaps
            return (-this.small[0] + this.large[0]) / 2;
        } else {
            // If the large heap somehow ends up with more elements, this will handle that case
            return this.large[0];
        }
    }

    private heappush(heap: number[], val: number): void {
        heap.push(val);
        heap.sort((a, b) => a - b); // This simulates the heap push by sorting
    }

    private heappop(heap: number[]): number {
        return heap.shift()!; // This simulates the heap pop by removing the first element
    }
}

/**
 * Your MedianFinder object will be instantiated and called as such:
 * var obj = new MedianFinder()
 * obj.addNum(num)
 * var param_2 = obj.findMedian()
 */

```

Time and Space Complexity

The provided class `MedianFinder` has two methods: `addNum` and `findMedian`, used for adding numbers and finding the median, respectively. The class maintains two heaps: `h1` is the min-heap that stores the larger half of the numbers, and `h2` is the max-heap that stores the smaller half of the numbers (as negatives).

`addNum` Method

Time Complexity:

- Inserting an element into a heap (`heappush`) has a time complexity of `O(log n)`, where `n` is the number of elements in the heap.
- Removing the smallest element from a min heap (`heappop` on `h1`) or the largest element from a max heap (`heappop` on `h2`) has a time complexity of `O(log n)`.
- The `addNum` method does at most two push and pop operations each time it is called:
  1. `heappush(self.h1, num)`: Insert `num` into the min-heap `h1`.
  2. `heappush(self.h2, -heappop(self.h1))`: Pop the minimum value from `h1`, negate it (to simulate a max-heap), and push it into `h2`.
  3. `heappush(self.h1, -heappop(self.h2))`: This happens only if the size of `h2` exceeds the size of `h1` by more than one, ensuring the difference in the number of elements in the two heaps never exceeds one.

Considering the above steps, the time complexity for `addNum` is `O(log n)` because of the heap operations.

Space Complexity:

- The space complexity of the `MedianFinder` class is `O(n)`, where `n` is the number of elements inserted into the `MedianFinder`. This is due to the storage of all elements across two heaps.

`findMedian` Method

Time Complexity:

- Accessing the top element of a heap (minimum value in `h1` and maximum value in `h2`) has a time complexity of `O(1)`.
- `findMedian` performs at most one arithmetic operation, which is done in constant time.

Therefore, the time complexity for `findMedian` is `O(1)`.

Space Complexity:

- The `findMedian` method does not use any additional space that depends on the input size, so its space complexity is `O(1)`.