# Streaming Median

Ibrahim Albluwi

# Streaming Median

Assume that you receive an arbitrary stream of numbers. How can we efficiently report the median at any point in time?

# Streaming Median

Assume that you receive an arbitrary stream of numbers. How can we efficiently report the median at any point in time?

**Solution 1.** Maintain a max-heap:

**`insert()`**: $O(\log n)$

**`median()`**: $O(n \log n)$
Remove from the heap the first $\frac{n}{2}$ elements to reach the median and then insert them back.

# Streaming Median

Assume that you receive an arbitrary stream of numbers. How can we efficiently report the median at any point in time?

**Solution 1.** Maintain a max-heap:

`insert()`: $O(\log n)$

`median()`: $O(n \log n)$
Remove from the heap the first $\frac{n}{2}$ elements to reach the median and then insert them back.

**Solution 2.** Maintain an unordered array:

`insert()`: $\Theta(1)$
Add to the end of the list. Note that the array might resize, so the running time is amortized.

`median()`: $\Theta(n)$
Use `Quickselect` to find the median. Note that this is the expected case if the array is shuffled.

# Streaming Median

Assume that you receive an arbitrary stream of numbers. How can we efficiently report the median at any point in time?

**Solution 1.** Maintain a max-heap:

**`insert()`**: $O(\log n)$

**`median()`**: $O(n \log n)$
Remove from the heap the first $\frac{n}{2}$ elements to reach the median and then insert them back.

**Solution 2.** Maintain an unordered array:

**`insert()`**: $\Theta(1)$
Add to the end of the list. Note that the array might resize, so the running time is amortized.

**`median()`**: $\Theta(n)$
Use `Quickselect` to find the median. Note that this is the expected case if the array is shuffled.

**Solution 3.** Maintain a sorted array:

**`insert()`**: $O(n)$
Search for the right position and then shift any elements that come after.

**`median()`**: $\Theta(1)$
The median is always at index $\frac{n}{2}$.

# Streaming Median

**Solution 4.** Use a max-heap to store the lower half of the elements ($\leq$ median) and a min-heap to store the upper half of the elements (> median).

Assume that the max-heap is named **`left`** and the min-heap is named **`right`**. Ensure that:

- Any element in **`left`** is smaller than or equal to all the elements in **`right`**.

# Streaming Median

**Solution 4.** Use a max-heap to store the lower half of the elements (≤ median) and a min-heap to store the upper half of the elements (> median).

Assume that the max-heap is named **`left`** and the min-heap is named **`right`**. Ensure that:

- Any element in **`left`** is smaller than or equal to all the elements in **`right`**.
- **`left`**`.size() - `**`right`**`.size()` is `0` (equal) or `1` (**`left`** is larger by 1).

# Streaming Median

**Solution 4.** Use a max-heap to store the lower half of the elements (≤ median) and a min-heap to store the upper half of the elements (> median).

Assume that the max-heap is named **`left`** and the min-heap is named **`right`**. Ensure that:

- Any element in **`left`** is smaller than or equal to all the elements in **`right`**.
- **`left`**`.size() - `**`right`**`.size()` is `0` (equal) or `1` (**`left`** is larger by 1).

Therefore, **`left`** always has $\lceil \frac{n}{2} \rceil$ elements and the **median** is always the **maximum** element in **`left`**.

`left` `right`
**Example**: `[1 2 `**`3`**` • 4 5 6]`
`median`

`left` `right`
**Example**: `[1 2 3 `**`4`**` • 5 6 7]`
`median`

# Streaming Median

**Solution 4.** Use a max-heap to store the lower half of the elements ($\leq$ median) and a min-heap to store the upper half of the elements ($>$ median).

Assume that the max-heap is named **`left`** and the min-heap is named **`right`**. Ensure that:

- Any element in **`left`** is smaller than or equal to all the elements in **`right`**.
- **`left`**`.size()` - **`right`**`.size()` is `0` (equal) or `1` (**`left`** is larger by 1).

Therefore, **`left`** always has $\lceil \frac{n}{2} \rceil$ elements and the **median** is always the **maximum** element in **`left`**.

**Example**: `[1 2` **`3`** `• 4 5 6]` **Example**: `[1 2 3` **`4`** `• 5 6 7]`

### General Idea:

- **Insert** the new element into the left heap if it is less than or equal to the current median and to the right if it is greater than the current median.

- **Rebalance** the heaps by moving an element from the larger heap to the smaller heap if the size invariant is violated.

# Streaming Median

**Solution 4.** Use a max-heap to store the lower half of the elements (≤ median) and a min-heap to store the upper half of the elements (> median).

Assume that the max-heap is named **`left`** and the min-heap is named **`right`**. Ensure that:

- Any element in **`left`** is smaller than or equal to all the elements in **`right`**.
- **`left`**`.size() -` **`right`**`.size()` is `0` (equal) or `1` (**`left`** is larger by 1).

Therefore, **`left`** always has $\lceil \frac{n}{2} \rceil$ elements and the **median** is always the **maximum** element in **`left`**.

**Example**: `[1 2 3 • 4 5 6]` **Example**: `[1 2 3 4 • 5 6 7]`

```
insert(k):

    If k <= left.max(): left.insert(k)
    Else:               right.insert(k)
```

# Streaming Median

**Solution 4.** Use a max-heap to store the lower half of the elements (≤ median) and a min-heap to store the upper half of the elements (> median).

Assume that the max-heap is named **`left`** and the min-heap is named **`right`**. Ensure that:

- Any element in **`left`** is smaller than or equal to all the elements in **`right`**.
- **`left`**`.size() - `**`right`**`.size()` is `0` (equal) or `1` (**`left`** is larger by 1).

Therefore, **`left`** always has $\lceil \frac{n}{2} \rceil$ elements and the **median** is always the **maximum** element in **`left`**.

**Example**: `[1 2 3 • 4 5 6]`  **Example**: `[1 2 3 4 • 5 6 7]`

```
insert(k):

    If k <= left.max(): left.insert(k)
    Else:               right.insert(k)
```

insert $k$ in **`left`** if $k \leq$ median

insert $k$ in **`right`** if $k >$ median

median

# Streaming Median

**Solution 4.** Use a max-heap to store the lower half of the elements (≤ median) and a min-heap to store the upper half of the elements (> median).

Assume that the max-heap is named **`left`** and the min-heap is named **`right`**. Ensure that:

- Any element in **`left`** is smaller than or equal to all the elements in **`right`**.
- **`left`**`.size() - `**`right`**`.size()` is `0` (equal) or `1` (**`left`** is larger by 1).

Therefore, **`left`** always has $\lceil \frac{n}{2} \rceil$ elements and the **median** is always the **maximum** element in **`left`**.

**Example**: `[1 2 `**`3`**` • 4 5 6]`  **Example**: `[1 2 3 `**`4`**` • 5 6 7]`

```
insert(k):

    If k <= left.max(): left.insert(k)
    Else:               right.insert(k)

    If left.size()  > right.size()+1: right.insert(left.delMax()).
```

more than $\lceil \frac{n}{2} \rceil$ elements are in **`left`**

# Streaming Median

**Solution 4.** Use a max-heap to store the lower half of the elements (≤ median) and a min-heap to store the upper half of the elements (> median).

Assume that the max-heap is named **`left`** and the min-heap is named **`right`**. Ensure that:

- Any element in **`left`** is smaller than or equal to all the elements in **`right`**.
- **`left`**`.size() - `**`right`**`.size()` is `0` (equal) or `1` (**`left`** is larger by 1).

Therefore, **`left`** always has $\lceil \frac{n}{2} \rceil$ elements and the **median** is always the **maximum** element in **`left`**.

**Example**: `[1 2 3 • 4 5 6]`  **Example**: `[1 2 3 4 • 5 6 7]`

```
insert(k):

    If k <= left.max(): left.insert(k)
    Else:               right.insert(k)

    If left.size()  > right.size()+1: right.insert(left.delMax()).
```

remove the max from left
and insert it into right

# Streaming Median

**Solution 4.** Use a max-heap to store the lower half of the elements (≤ median) and a min-heap to store the upper half of the elements (> median).

Assume that the max-heap is named **`left`** and the min-heap is named **`right`**. Ensure that:

- Any element in **`left`** is smaller than or equal to all the elements in **`right`**.
- **`left`**`.size() - `**`right`**`.size()` is `0` (equal) or `1` (**`left`** is larger by 1).

Therefore, **`left`** always has $\lceil \frac{n}{2} \rceil$ elements and the **median** is always the **maximum** element in **`left`**.

**Example**: `[1 2 3 • 4 5 6]` **Example**: `[1 2 3 4 • 5 6 7]`

```
insert(k):

    If k <= left.max(): left.insert(k)
    Else:               right.insert(k)

    If left.size()  > right.size()+1: right.insert(left.delMax()).
    If right.size() > left.size():    left.insert(right.delMin()).
```

If **`right`** is larger than **`left`** remove the min from **`right`** and insert it into **`left`**.

# Streaming Median

**Solution 4.** Use a max-heap to store the lower half of the elements (≤ median) and a min-heap to store the upper half of the elements (> median).

Assume that the max-heap is named **`left`** and the min-heap is named **`right`**. Ensure that:

- Any element in **`left`** is smaller than or equal to all the elements in **`right`**.
- **`left`**`.size() - `**`right`**`.size()` is `0` (equal) or `1` (**`left`** is larger by 1).

Therefore, **`left`** always has $\lceil \frac{n}{2} \rceil$ elements and the **median** is always the **maximum** element in **`left`**.

**Example**: `[1 2 `**`3`**` • 4 5 6]` **Example**: `[1 2 3 `**`4`**` • 5 6 7]`

**insert** into the correct heap

**rebalance** the heaps if necessary

```
insert(k):
    If k <= left.max(): left.insert(k)
    Else:               right.insert(k)

    If left.size()  > right.size()+1: right.insert(left.delMax()).
    If right.size() > left.size():    left.insert(right.delMin()).
```

# Streaming Median

**Solution 4.** Use a max-heap to store the lower half of the elements (≤ median) and a min-heap to store the upper half of the elements (> median).

Assume that the max-heap is named **`left`** and the min-heap is named **`right`**. Ensure that:

- Any element in **`left`** is smaller than or equal to all the elements in **`right`**.
- **`left`**`.size() - `**`right`**`.size()` is `0` (equal) or `1` (**`left`** is larger by 1).

Therefore, **`left`** always has $\lceil \frac{n}{2} \rceil$ elements and the **median** is always the **maximum** element in **`left`**.

**Example**: `[1 2 3 • 4 5 6]` **Example**: `[1 2 3 4 • 5 6 7]`

```
insert(k):
    If k <= left.max(): left.insert(k)
    Else:               right.insert(k)

    If left.size()  > right.size()+1: right.insert(left.delMax()).
    If right.size() > left.size():    left.insert(right.delMin()).
```

**insert** into the correct heap — $O(\log n)$

**rebalance** the heaps if necessary — $O(\log n)$

# Streaming Median

**Solution 4.** Use a max-heap to store the lower half of the elements ($\leq$ median) and a min-heap to store the upper half of the elements ($>$ median).

Assume that the max-heap is named **`left`** and the min-heap is named **`right`**. Ensure that:

- Any element in **`left`** is smaller than or equal to all the elements in **`right`**.
- **`left`**`.size() - `**`right`**`.size()` is `0` (equal) or `1` (**`left`** is larger by 1).

Therefore, **`left`** always has $\lceil \frac{n}{2} \rceil$ elements and the **median** is always the **maximum** element in **`left`**.

**Example**: `[1 2 `**`3`**` • 4 5 6]` **Example**: `[1 2 3 `**`4`**` • 5 6 7]`

Running Time:

**`insert()`**: $O(\log n)$
Inserting into the left or the right heaps is $O(\log n)$ and rebalancing is $O(\log n)$.

**`median()`**: $\Theta(1)$
The median is always `left.max()`.