Technical Document: MedianQueue Data Structure

Title:

MedianQueue: An Order-Preserving Median Maintenance Data Structure

Overview:

`MedianQueue` is a custom-designed data structure that supports efficient median computation over a dynamic data stream with ordered insertion and removal (FIFO). Unlike a traditional priority queue or multiset, this structure maintains the **median** in **O(1)** time, with support for **insertion**, **removal**, and additional statistical functions like **mean**, **mode**, **sum**, **minimum**, and **maximum** in amortized **O(1)** time.

Data Structures Used:

- **multiset<long long> s**: Stores the data in sorted order for median extraction.
- **multiset<long long>::iterator median**: Maintains a pointer to the current median.
- **deque<long long> q**: Preserves the insertion order for FIFO-based removals.

Use Cases for Extended MedianQueue

1. Real-Time Sensor Stream with Sliding-Window Statistics

Scenario:
An industrial machine continuously emits readings—for example, temperature, vibration level, or pressure—every second. The control system needs to monitor these values in real time but must ignore transient spikes or sensor glitches. At any moment, it should report:

- The **minimum** and **maximum** readings over the last *N* seconds,
- The **median** to filter out occasional outliers,

- The **arithmetic mean** for energy or resource balancing,
- The **mode** (most frequent reading) to detect repeating fault patterns,
- The **sum** to compute total resource usage (for example, total gas flow),
- And automatically discard any reading older than *N* seconds.

How MedianQueue Helps:

1. **Queue (FIFO) Behavior:**
    - Every new reading is inserted at the back of the structure.
    - Simultaneously, any reading older than *N* seconds (the oldest element) is removed from the front. This guarantees the container always holds exactly the last *N* data points.
2. **Sliding-Window Analytics in O(1):**
    - `getMin()` and `getMax()` are retrieved in constant time because the structure maintains auxiliary pointers or values.
    - `getMedian()` adjusts automatically as insertions and removals happen, yielding the current median in *O(1)*.
    - `getMean()` is O(1) because the class keeps a running total (`sum`) and a count of elements.
    - `getMode()` remains O(1) on average if a frequency map is maintained alongside insertion/removal.
    - `getSum()` simply returns the stored sum.
3. **Why It Matters:**
    - In a noisy industrial environment, a single sensor glitch (e.g., a sudden but incorrect spike) should not trigger an alarm. The median and minimum/maximum filters avoid false positives.
    - The mean and sum help in energy-budget planning—for instance, estimating total power consumed over that same window.

- Mode detection can immediately point out if a certain sensor reading repeats abnormally (e.g., a vibration level staying at a dangerous threshold).

---

2. Financial Markets: Sliding-Window Price Analysis

Scenario:
A trading platform displays the last ten minutes of transaction prices for a particular stock. Traders and automated algorithms rely on statistics such as:

- The **median** price (to avoid impact of flash spikes),
- The **mean** price (for trend estimation),
- The **maximum** and **minimum** trade values (to detect sudden volatility),
- The **sum** of trade amounts (to gauge total trading volume),
- The **mode** price (to identify a commonly traded price point).

Once a trade is more than ten minutes old, it should no longer influence these statistics.

How MedianQueue Helps:

1. **Queue (FIFO) Behavior & Sliding Window:**
    - Each new trade (price and quantity) is inserted.
    - Every ten-minute mark, the oldest trade "falls out" of the window and is removed.
    - The data structure always holds exactly those trades that occurred within the most recent ten minutes.
2. **Constant-Time Statistical Queries:**
    - `getMin()` and `getMax()` report the lowest and highest price in the most recent ten-minute window instantly.
    - `getMedian()` yields the price at which half of the trades were above and half below. This is essential to mitigate the effect of a single large block trade driving the average.

- `getMean()` computes the weighted or unweighted average price, depending on implementation, in O(1) by keeping a running sum.
- `getMode()` identifies the price level that occurred most frequently during that period, which may signal that many traders are clustering around one price.
- `getSum()` provides total trading volume (sum of quantities) or sum of prices (depending on design), which is crucial for volume-weighted analytics.

3. **Business Benefits:**
   - High-frequency traders and risk-management systems can react within microseconds—no costly recomputation at each query.
   - By maintaining all statistics simultaneously in one container, the platform avoids building separate data structures for median, maximum, or sum, reducing memory overhead and synchronization complexity.

---

3. Web Server Latency Monitoring (Rolling Window)

Scenario:
A cloud service collects response times for incoming API calls. Every second, the monitoring system records the latency (in milliseconds). To drive an SLA dashboard and automated throttling:

- The **maximum** latency in the past five minutes must be known (to detect outlier requests).
- The **minimum** latency (for baseline tracking).
- The **median** latency (to ignore spikes caused by transient network issues).
- The **mean** latency (to observe overall performance trends).
- The **mode** latency (the latency that occurs most often, which may reveal a bottleneck or caching effect).
- The **sum** of all latencies (to compute average over the window or total latency burden).

Each data point older than five minutes should be evicted automatically.

How MedianQueue Helps:

1. **Automatic Eviction & Order Preservation:**
    - Each API call latency is enqueued as soon as it is measured.
    - After five minutes, the data structure's `pop()` method removes the oldest latency.
    - This ensures precise sliding-window behavior without manual housekeeping.
2. **O(1) Access to All Metrics:**
    - `getMin()` and `getMax()` directly return the smallest and largest value in the current window.
    - `getMedian()` provides the 50th percentile latency.
    - `getMean()` and `getSum()` are updated in constant time by maintaining a running sum and current count.
    - `getMode()` reads from a maintained frequency map to identify the most frequently observed latency.
3. **Practical Outcome:**
    - If the median latency suddenly jumps, the system can flag "steady degradation," even if a few anomalous high-latency requests occur.
    - The maximum latency helps trigger immediate alerts for outliers (e.g., an individual request taking ten seconds).
    - The rolling mean and sum can feed into resource-allocation algorithms (e.g., auto-scaling new server instances).

---

4. Healthcare Wearables: Continuous Vital-Sign Tracking

Scenario:
A wearable device tracks a user's heart rate, blood oxygen saturation (SpO_2), and step count in real time. For patient safety and trend analysis, the device's firmware needs to display or transmit:

- The **minimum** and **maximum** heart rate over the past one minute (to detect bradycardia or tachycardia).
- The **median** SpO_2 reading (to filter out occasional erroneous readings due to motion).
- The **mean** heart rate or SpO_2 (for trending or warning thresholds).
- The **mode** posture or activity level (e.g., the most common step interval).
- The **sum** of steps (total activity count).
- As soon as a data point is more than one minute old, it must be removed so that the displayed statistics always reflect the last sixty seconds of data.

How MedianQueue Helps:

1. **Sliding-Window with Strict Time Bound:**
    - Each new heart-rate sample (or SpO_2 sample or step event) is enqueued immediately.
    - A timer periodically calls `pop()` to evict data older than one minute.
    - This keeps the window size fixed at all times (e.g., 60 heart-rate values if samples arrive once per second).
2. **Instantaneous Vital Statistics:**
    - `getMin()` and `getMax()` detect dangerous deviations: if max > 180 bpm or min < 40 bpm, an alert can trigger.
    - `getMedian()` filters out momentary signal dropouts (e.g., false low SpO_2 reading).
    - `getMean()` smooths the display of real-time vitals, so the user or physician sees stable trends.
    - `getMode()` identifies the most frequent activity classification (e.g., “walking” if step intervals cluster around a particular value).
    - `getSum()` provides cumulative step count for the window, useful for in-device calorie estimates.

3. **User and Clinical Benefits:**

    - Patients receive stable, reliable feedback; caregivers can be alerted as soon as statistics cross predefined thresholds.

    - Battery life improves because there is no need to recompute statistics from scratch on every measurement.

    - The firmware remains lightweight (all O(1) functions are fast and deterministic).

---

5. Online Multiplayer Game Leaderboard Balancing

Scenario:
A game server tracks player scores in a continuously running match. To maintain fair matchmaking and dynamic in-game adjustments, the system requires:

- The **minimum** and **maximum** score among active players (to detect runaway leaders or very low-performing players).

- The **median** score (to set difficulty or adjust spawn rates).

- The **mean** score (for global balance metrics).

- The **mode** score (if many players cluster around the same score tier).

- The **sum** of all scores to compute average score increment per round.

- As each round ends, the scores for players who leave are evicted; new players' scores get inserted.

How MedianQueue Helps:

1. **Queue Behavior for Rounds (FIFO):**

    - Treat each player's score record similarly to a data point in a queue. When a player leaves or a round completes, that record is popped.

    - In team-based modes, you can maintain separate structures per team.

2. **Sliding-Window is Comparable to Active Set:**

- The window corresponds to “currently active players.”
- As soon as a player leaves, `pop()` removes their score; as soon as a new player joins or scores update, `insert()` updates accordingly.

3. **Instantaneous Score Analytics:**
   - `getMin()`/`getMax()` detect outliers—if a player is too far ahead, the game can spawn power-downs or “rubber-band” mechanics.
   - `getMedian()` defines the middle performance level: if the median falls below a threshold, the server may adjust difficulty.
   - `getMean()` yields an overall performance indicator.
   - `getMode()` helps identify “score tiers” where many players land, enabling targeted in-game events (e.g., bonus for those in Tier 3).
   - `getSum()` tracks total team or match score, useful for objectives like “first team to reach 10,000 points.”
4. **Gameplay Advantages:**
   - Real-time balancing becomes trivial without recomputing large sorted lists or using multiple heaps.
   - The server code remains concise—one data structure does all statistical heavy lifting in O(1) per query.
   - Enhanced fairness and responsiveness improve player retention.

Why These Use Cases Benefit from Extended MedianQueue

1. **Single Container for Multiple Metrics:**
   - Instead of building separate min-heap, max-heap, balanced BST, or frequency map for mode, the extended MedianQueue bundles all required statistics into

one cohesive structure.

2. **FIFO Order & Sliding Window:**
    - Many real-world applications are not just “all-time” statistics but require “last *N* seconds/minutes/transactions.”
    - By preserving insertion order in the internal deque and automatically popping the oldest entries, this data structure seamlessly supports a sliding window.
3. **Constant-Time Queries:**
    - When monitoring must happen continuously (e.g., hundreds of sensor readings per second), querying in O(1) for median, mean, min, max, mode, and sum is essential to avoid bottlenecks.
4. **Low Memory Footprint & Simplified Maintenance:**
    - Maintaining only one multiset (for median), one deque (for FIFO), and auxiliary variables/maps (for sum, frequency, min, max) uses less memory than multiple separate structures.
    - Code remains easier to audit and less prone to bugs compared to synchronizing several data structures manually.

By incorporating **queue behavior**, **sliding-window functionality**, and **constant-time** access to `getMin()`, `getMax()`, `getMedian()`, `getMean()`, `getMode()`, and `getSum()`, the extended **MedianQueue** becomes a versatile tool across domains ranging from industrial IoT to finance, healthcare, web services, and gaming.

Method Descriptions and Time Complexities:

Method	Description	Time Complexity
`insert(x)`	Inserts a new value `x` into the structure while adjusting the median.	O(log n)
`pop()`	Removes the oldest inserted element, maintains order and median tracking.	O(log n)
`peek()`	Returns the oldest element inserted (front of queue).	O(1)
`getMedian()`	Returns the current median value from the multiset.	O(1)
`dump()`	Clears the structure, resets internal containers and state.	O(n)
`size()`	Returns the number of elements currently in the structure.	O(1)
`empty()`	Checks if the container is empty.	O(1)

Additional Extensions (To Be Implemented):

The following methods are proposed as part of the extended `MedianQueue` version:

Method	Description	Time Complexity
`getMean()`	Returns the arithmetic mean of all elements. Requires maintaining a sum.	O(1)
`getMode()`	Returns the mode (most frequent element). Needs frequency map maintenance.	O(1) (amortized)
`getSum()`	Returns the sum of all elements inserted. Requires sum tracking.	O(1)
`getMin()`	Returns the minimum element.	O(1)
`getMax()`	Returns the maximum element.	O(1)

Note: These enhancements will require auxiliary data members like `sum`, `frequency_map`, `min`, and `max` to be tracked during insertions and deletions.

Invariants Maintained:

- The median iterator always points to the correct median element.
- When size is odd: median points to the middle element.
- When size is even: median points to the **lower** of the two middle values.

Design Strengths:

- Maintains correct order for `pop()` with consistent median adjustment.
- Efficient due to multiset balancing and iterator tracking.
- Extendable for rich statistical analysis in real-time systems.

Possible Enhancements:

- Fixed window-size version (`MedianSlidingWindowQueue`)
- Support for percentile-based queries
- Thread-safe variant for concurrent environments
- Integration with persistent storage for long-term data tracking

Limitations:

- Internally uses `multiset`, so memory usage can be slightly higher than raw arrays.
- Each `insert()` or `pop()` involves log-time complexity due to multiset operations.

Author Information:

Developer: Shashank Vashistha
Academic Institution: NIT Kurukshetra
Program: MCA
Intellectual Property Type: Software/Data Structure Design
Proposed Protection: Copyright