# Question

Given a stream of integers and a window size, calculate the moving average of all integers in the sliding window.

Implement the MovingAverage class:

* MovingAverage(int size) Initializes the object with the size of the window size.
* double next(int val) Returns the moving average of the last size values of the stream.

**Example 1:**

**Input**

["MovingAverage", "next", "next", "next", "next"]

[[3], [1], [10], [3], [5]]

**Output**

[null, 1.0, 5.5, 4.66667, 6.0]

**Explanation**

MovingAverage movingAverage = new MovingAverage(3);

movingAverage.next(1); // return 1.0 = 1 / 1

movingAverage.next(10); // return 5.5 = (1 + 10) / 2

movingAverage.next(3); // return 4.66667 = (1 + 10 + 3) / 3

movingAverage.next(5); // return 6.0 = (10 + 3 + 5) / 3

**Constraints:**

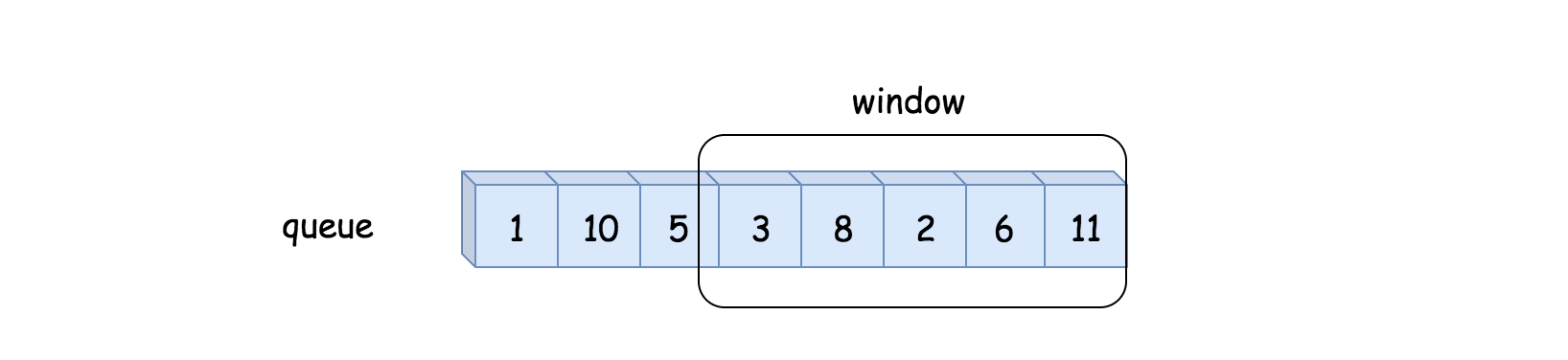
* 1 <= size <= 1000
* -105 <= val <= 105
* At most 104 calls will be made to next.

# Solution

#### **Approach 1: Array or List**

**Intuition**

Following the description of the problem, we could simply keep track of all the incoming values with the data structure of Array or List. Then from the data structure, later we retrieve the necessary elements to calculate the average.



**Algorithm**

* First, we initialize a variable queue to store the values from the data stream, and the variable n for the size of the moving window.
* At each invocation of next(val), we first append the value to the queue. We then retrieve the last n values from the queue, in order to calculate the average.

|  |
| --- |
| class MovingAverage {  int size;  List queue = new ArrayList<Integer>();  public MovingAverage(int size) {  this.size = size;  }  public double next(int val) {  queue.add(val);  // calculate the sum of the moving window  int windowSum = 0;  for(int i = Math.max(0, queue.size() - size); i < queue.size(); ++i)  windowSum += (int)queue.get(i);  return windowSum \* 1.0 / Math.min(queue.size(), size);  }  } |

**Complexity**

* Time Complexity: \mathcal{O}(N)O(*N*) where N*N* is the size of the moving window, since we need to retrieve N*N* elements from the queue at each invocation of next(val) function.
* Space Complexity: \mathcal{O}(M)O(*M*), where M*M* is the length of the queue which would grow at each invocation of the next(val) function.

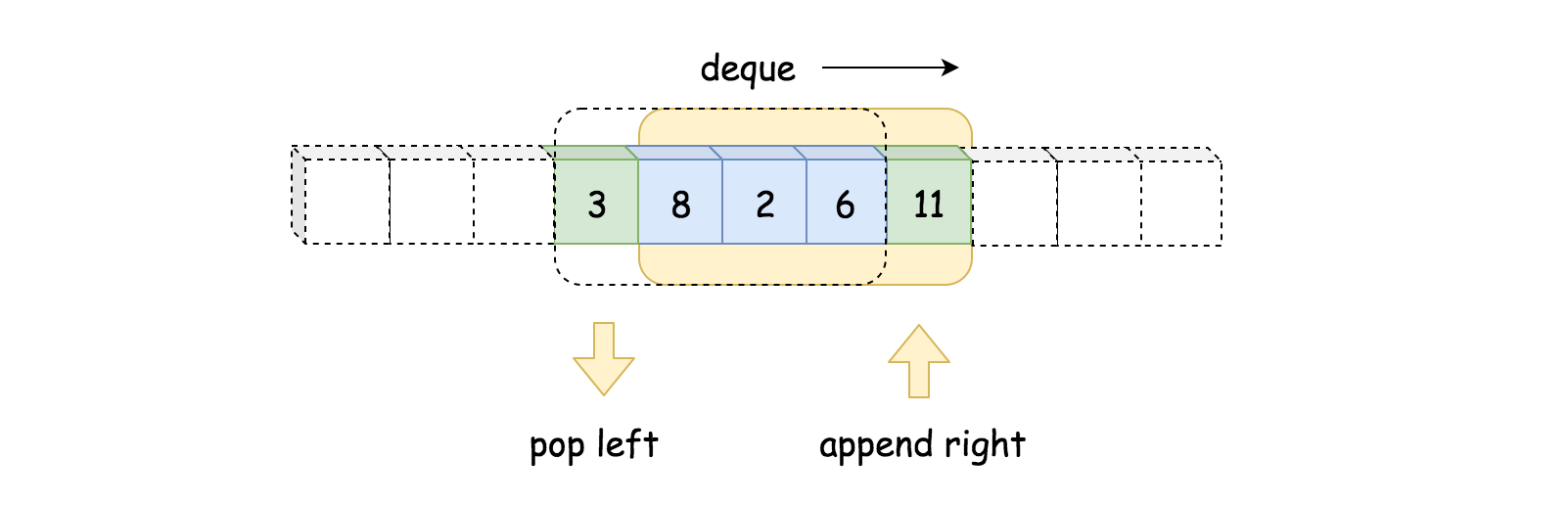
#### **Approach 2: Double-ended Queue**

**Intuition**

We could do better than the first approach in both time and space complexity.

First of all, one might notice that we do not need to keep all values from the data stream, but rather the last n values which falls into the moving window.

By definition of the moving window, at each step, we add a new element to the window, and at the same time we remove the oldest element from the window. Here, we could apply a data structure called double-ended queue (a.k.a deque) to implement the moving window, which would have the constant time complexity (\mathcal{O}(1)O(1)) to add or remove an element from both its ends. With the deque, we could reduce the space complexity down to \mathcal{O}(N)O(*N*) where N*N* is the size of the moving window.



Secondly, to calculate the sum, we do not need to reiterate the elements in the moving window.

We could keep the sum of the previous moving window, then in order to obtain the sum of the new moving window, we simply add the new element and deduce the oldest element. With this measure, we then can reduce the time complexity to constant.

**Algorithm**

Here is the definition of the deque from Python. We have similar implementation of deque in other programming languages such as Java.

Deques are a generalization of stacks and queues (the name is pronounced deck and is short for double-ended queue). Deques support thread-safe, memory efficient appends and pops from either side of the deque with approximately the same O(1) performance in either direction.

Follow the intuition, we replace the queue with the deque and add a new variable window\_sum in order to calculate the sum of moving window in constant time.

|  |
| --- |
| class MovingAverage {  int size, windowSum = 0, count = 0;  Deque queue = new ArrayDeque<Integer>();  public MovingAverage(int size) {  this.size = size;  }  public double next(int val) {  ++count;  // calculate the new sum by shifting the window  queue.add(val);  int tail = count > size ? (int)queue.poll() : 0;  windowSum = windowSum - tail + val;  return windowSum \* 1.0 / Math.min(size, count);  }  } |

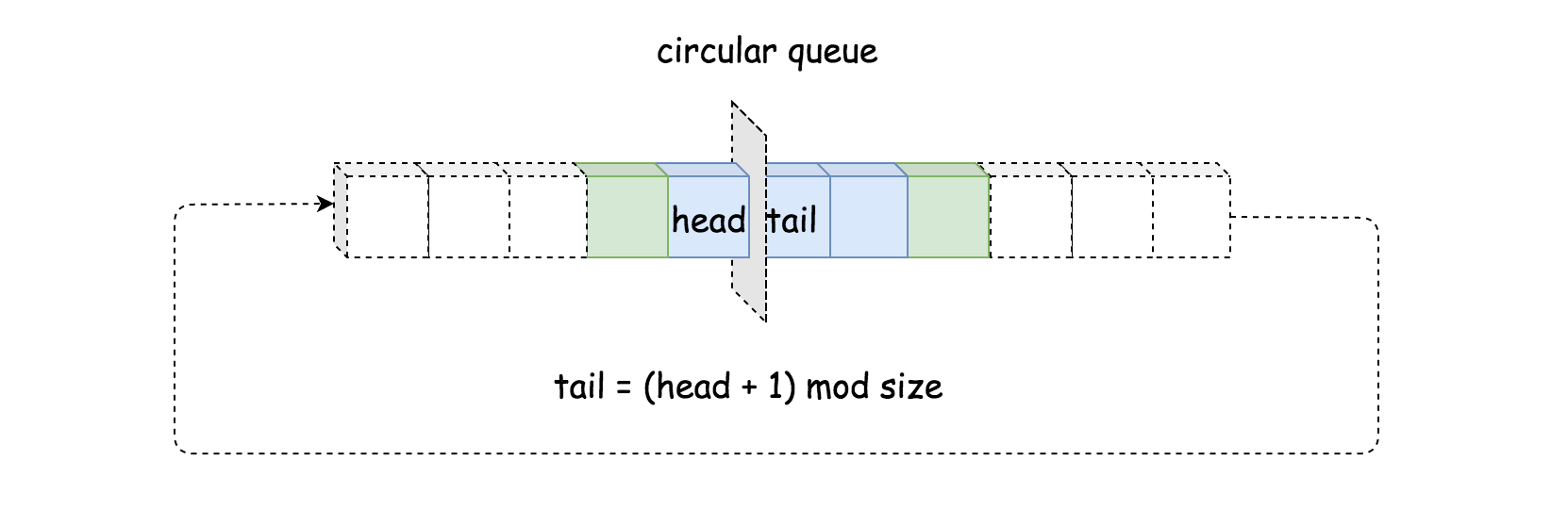
**Complexity**

* Time Complexity: \mathcal{O}(1)O(1), as we explained in intuition.
* Space Complexity: \mathcal{O}(N)O(*N*), where N*N* is the size of the moving window.

#### **Approach 3: Circular Queue with Array**

**Intuition**

Other than the deque data structure, one could also apply another fun data structure called circular queue, which is basically a queue with the circular shape.



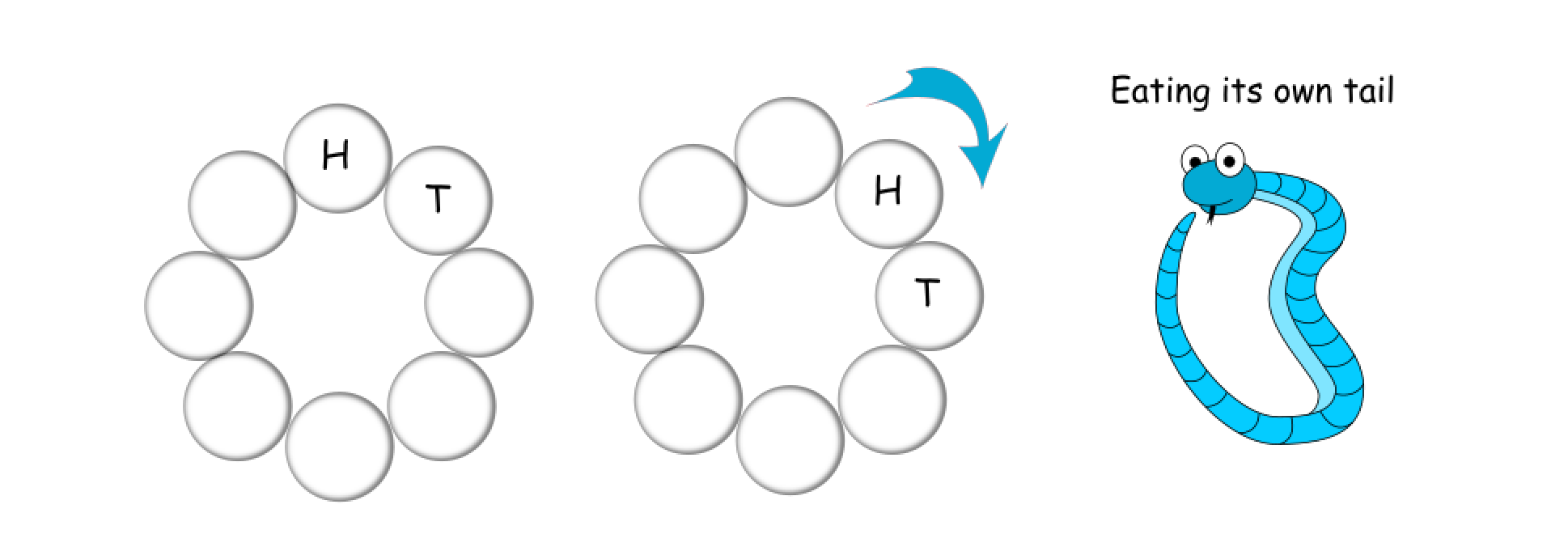
* The major advantage of circular queue is that by adding a new element to a full circular queue, it automatically discards the oldest element. Unlike deque, we do not need to explicitly remove the oldest element.
* Another advantage of circular queue is that a single index suffices to keep track of both ends of the queue, unlike deque where we have to keep a pointer for each end.

**Algorithm**

No need to resort to any library, one could easily implement a circular queue with a fixed-size array. The key to the implementation is the correlation between the index of head and tail elements, which we could summarize in the following formula:

tail=(head+1)modsize

In other words, the tail element is right next to the head element. Once we move the head forward, we would overwrite the previous tail element.



|  |
| --- |
| **class MovingAverage {**  **int size, head = 0, windowSum = 0, count = 0;**  **int[] queue;**  **public MovingAverage(int size) {**  **this.size = size;**  **queue = new int[size];**  **}**  **public double next(int val) {**  **++count;**  **// calculate the new sum by shifting the window**  **int tail = (head + 1) % size;**  **windowSum = windowSum - queue[tail] + val;**  **// move on to the next head**  **head = (head + 1) % size;**  **queue[head] = val;**  **return windowSum \* 1.0 / Math.min(size, count);**  **}**  **}** |

**Complexity**

* Time Complexity: \mathcal{O}(1)O(1), as we can see that there is no loop in the next(val) function.
* Space Complexity: \mathcal{O}(N)O(*N*), where N*N* is the size of the circular queue.