703. Kth Largest Element in a Stream

Easy | Tree | Design | Binary Search Tree | Binary Tree | Data Stream | Heap (Priority Queue) | Leetcode Link

Problem Description

In this problem, we are asked to design a `KthLargest` class that can determine the k^{th} largest element in a stream of numbers. Importantly, when we refer to the k^{th} largest element, we mean according to sorted order, not that the element is distinct from the others. The numbers can be repeated in the stream.

The class should be able to handle two types of actions:

1. Initialization (`__init__`): When an instance of the class is created, it should be initialized with an integer `k` and an array of integers `nums`. The `k` represents the position of the largest element we are interested in, and `nums` is the initial stream of numbers.

2. Adding New Elements (`add`): The class should provide a method to add a new integer `val` to the stream. After adding the new integer, this method should return the current k^{th} largest element from the stream.

Intuition

The straightforward approach to find the k^{th} largest element would be to sort the stream of numbers each time we insert a new element and then pick the k^{th} largest. However, sorting every time we insert a new element leads to a less efficient solution. To solve the problem more optimally, we can use a data structure that does the hard work for us - a Min Heap.

The reason behind choosing a Min Heap is its useful property: the smallest element is always at the root and can be accessed in constant time. If we maintain a Min Heap of exactly `k` largest elements from the current stream, the root of the heap (the smallest element in the heap) is our desired k^{th} largest element.

Here's how we construct our solution using this intuition:

1. Initialize a Min Heap (`self.q`) and a variable `self.size` to store `k`.
2. Insert the initial stream of numbers to the Min Heap using the `add` method.
   - During each insert, we push the new value into the heap.
   - If the heap size exceeds `k`, it means we have more than `k` elements, so we can remove the smallest one (the root).
   - The k^{th} largest element will then be the new root of the heap.
3. The `add` method maintains the Min Heap invariant and ensures it always contains the `k` largest elements after each insertion.
4. Finally, we can return the k^{th} largest element by looking at the root of the Min Heap.

Using this approach, we avoid the need to sort the entire stream every time, thereby improving the efficiency of finding the k^{th} largest element in the stream.

Solution Approach

To implement the solution, we employ Python's `heapq` module, which provides an implementation of the Min Heap data structure.

Here's a step-by-step breakdown of the implementation:

1. When an instance of `KthLargest` class is created, the `__init__` method is called with two parameters: `k` and `nums`.

   - `k` is stored in `self.size` which represents the position of the k^{th} largest element we are interested in.
   - `self.q` is initialized as an empty list, which will be used as the Min Heap.
2. The `nums` list is iterated over and each number is added to the Min Heap using the `add` method.

3. The `add` method handles inserting a new element into the Min Heap and ensures the Min Heap contains only the `k` largest elements.

   - `heappush(self.q, val)` is used to add the new value `val` to the Min Heap.
   - We check if the size of the heap exceeds `k` by comparing `len(self.q)` to `self.size`.
   - If the heap size is greater than `k`, we remove the smallest element (root of the Min Heap) using `heappop(self.q)` to ensure that only the `k` largest elements remain.
   - This keeps the heap size at max `k` elements at all times.
   - The root element, which can be accessed with `self.q[0]`, is the k^{th} largest element because all elements bigger than it are also in the heap.

By following this approach, we ensure that:

- The add operation has a time complexity of O(log k) since it involves heap operations which work in logarithmic time relative to the number of elements in the heap.
- The space complexity of the solution is O(k) because the Min Heap holds at most `k` elements at any given time.

This provides a balanced and efficient way to continuously find the k^{th} largest element in the stream without having to sort the entire array each time a new element is added.

Example Walkthrough

Let us consider an example to illustrate the solution approach.

Suppose `k` is 3, and the initial array of numbers `nums` is `[4, 5, 8, 2]`. We want to design a `KthLargest` class where we can add elements and always be able to find the 3rd largest element in the current stream.

Here is a step-by-step walkthrough of how the `KthLargest` class processes the stream of numbers:

1. We create an instance of `KthLargest` by passing in our values `k=3` and `nums=[4,5,8,2]`. The `__init__` method initializes `self.size` to 3 and `self.q` as an empty Min Heap.

2. We add each element of `nums` into the Min Heap using the `add` method, ensuring that after every addition, the Min Heap holds at most `k` largest elements. Initially, the Min Heap (self.q) is empty, and we proceed as follows:

   - Add 4: self.q becomes [4].
   - Add 5: self.q becomes [4, 5].
   - Add 8: self.q becomes [4, 5, 8].
   - Add 2: self.q remains [4, 5, 8] because 2 is not among the 3 largest elements.

Now the Min Heap contains `k` largest elements, the k^{th} largest (3rd largest) among them is 4 (the root of the Min Heap).

3. If we call the `add` method to insert a new element, let's say `3`, we again ensure that the Min Heap only contains the `k` largest elements.

   - Add 3: self.q remains [4, 5, 8] because 3 is not among the 3 largest elements. The 3rd largest element is still 4.
4. Let's insert a larger element, `9`. The Min Heap needs to accommodate this change, and the smallest element in the heap should be removed if the heap's size exceeds `k`.

   - Add 9: self.q becomes [5, 8, 9] after removing 4 because the size exceeded `k`. Now the 3rd largest element is 5.

So after adding the element 9, our Min Heap has elements [5, 8, 9], and the 3rd largest (k^{th} largest) element is 5.

The class now effectively and efficiently maintains the 3rd largest element as we add new values into the stream. The `add` method updates the Min Heap in logarithmic time, making the operation swift compared to the linear time complexity of sorting the numbers upon each insertion.

Python Solution

```python
from heapq import heappush, heappop

class KthLargest:
    def __init__(self, k: int, nums: List[int]):
        # Initialize a min heap to store the k largest elements
        self.min_heap = []
        # Store the size k to know the kth largest value
        self.k = k
        # Add the initial elements to the heap
        for num in nums:
            self.add(num)

    def add(self, val: int) -> int:
        # Add the new value to the min heap
        heappush(self.min_heap, val)
        # If the size of the heap exceeds k, remove the smallest element
        if len(self.min_heap) > self.k:
            heappop(self.min_heap)
        # The root of the min heap is the kth largest value
        return self.min_heap[0]

# Example usage:
# kthLargest = KthLargest(3, [4, 5, 8, 2])
# print(kthLargest.add(3))   # returns 4
# print(kthLargest.add(5))   # returns 5
# print(kthLargest.add(10))  # returns 5
# print(kthLargest.add(9))   # returns 8
# print(kthLargest.add(4))   # returns 8

```

Java Solution

```java
import java.util.PriorityQueue;

// Class to find the kth largest element in a stream of numbers
class KthLargest {

    // Priority queue to maintain the smallest 'k' elements seen so far
    private PriorityQueue<Integer> minHeap;

    // The kth position we are interested in for the largest element
    private int k;

    /**
     * Constructor to initialize the data structure and populate with initial elements.
     *
     * @param k    The kth position to track in the list of largest elements.
     * @param nums An array of initial numbers to be added to the kth largest tracker.
     */
    public KthLargest(int k, int[] nums) {
        // Initialize a min-heap with the capacity to hold 'k' elements
        this.minHeap = new PriorityQueue<>(k);
        this.k = k;

        // Add the initial elements to the kth largest tracker
        for (int num : nums) {
            add(num);
        }
    }

    /**
     * Adds a new number into the stream and returns the kth largest element.
     *
     * @param val The new number to be added to the stream.
     * @return The kth largest element after adding the new number.
     */
    public int add(int val) {
        // Always add the new value to the min-heap
        minHeap.offer(val);

        // If the size of the min-heap exceeds 'k', remove the smallest element
        if (minHeap.size() > k) {
            minHeap.poll();
        }

        // The root of the min-heap represents the kth largest element
        return minHeap.peek();
    }
}

/* Usage example:
 * KthLargest kthLargest = new KthLargest(3, new int[]{4, 5, 8, 2});
 * kthLargest.add(3);   // returns 4
 * kthLargest.add(5);   // returns 5
 * kthLargest.add(10);  // returns 5
 * kthLargest.add(9);   // returns 8
 * kthLargest.add(4);   // returns 8
 */

```

C++ Solution

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

// Define a class to find kth largest element in a stream of numbers.
class KthLargest {
public:
    // Declare a min heap to keep track of the k largest elements.
    std::priority_queue<int, std::vector<int>, std::greater<>> minHeap;
    // Store the value of 'k' to know which largest element to keep track of.
    int kthSize;

    // Constructor for initializing the KthLargest class.
    // k - The kth position to track.
    // nums - Initial list of numbers from which we find the kth largest element.
    KthLargest(int k, std::vector<int>& nums) {
        kthSize = k;
        // Add initial numbers to the heap
        for (int num : nums) {
            add(num);
        }
    }

    // Function to add a number to the stream and return the kth largest element.
    int add(int val) {
        // Add the new number to the min heap.
        minHeap.push(val);
        // If the heap size is greater than k, remove the smallest element.
        if (minHeap.size() > kthSize) {
            minHeap.pop();
        }
        // Return the kth largest element (top of the min heap).
        return minHeap.top();
    }
};

/**
 * The provided code snippet illustrates the use of the KthLargest class.
 * It demonstrates the instantiation of a KthLargest object and subsequent calls to its add method.
 */

```

Typescript Solution

```typescript
type ComparatorFunction = (a: number, b: number) => number;

let kthLargestSize: number;
let minHeap: MinHeap;

interface MinHeap {
    data: number[];
    comparator: ComparatorFunction;
    heapify: () => void;
    peek: () => number | null;
    offer: (value: number) => void;
    poll: () => number | null;
    bubbleUp: (index: number) => void;
    bubbleDown: (index: number) => void;
    swap: (index1: number, index2: number) => void;
    size: () => number;
}

// Initializing the Kth largest element finder
function KthLargestInit(k: number, nums: number[]): void {
    kthLargestSize = k;
    minHeap = createMinHeap();
    nums.forEach(num => {
        kthLargestAdd(num);
    });
}

// Function to add a new value to the data collection
function kthLargestAdd(val: number): number {
    minHeap.offer(val);
    if (minHeap.size() > kthLargestSize) {
        minHeap.poll();
    }
    return minHeap.peek()!;
}

// Create a MinHeap with default values and a comparator
function createMinHeap(): MinHeap {
    const heap: MinHeap = {
        data: [],
        comparator: (a, b) => a - b,
        heapify() {
            if (this.size() < 2) return;
            for (let i = 1; i < this.size(); i++) {
                this.bubbleUp(i);
            }
        },
        peek() {
            return this.size() === 0 ? null : this.data[0];
        },
        // Adds a new value and bubbles it up to maintain heap property
        offer(value) {
            this.data.push(value);
            this.bubbleUp(this.size() - 1);
        },
        // Removes and returns the root value of the heap
        poll() {
            if (this.size() === 0) {
                return null;
            }
            const result = this.data[0];
            const last = this.data.pop();
            if (this.size() !== 0) {
                this.data[0] = last!;
                this.bubbleDown(0);
            }
            return result;
        },
        // Bubbles a value up the heap until the heap property is restored
        bubbleUp(index) {
            while (index > 0) {
                const parentIndex = (index - 1) >> 1;
                if (this.comparator(this.data[index], this.data[parentIndex]) < 0) {
                    this.swap(index, parentIndex);
                    index = parentIndex;
                } else {
                    break;
                }
            }
        },
        // Bubbles a value down the heap until the heap property is restored
        bubbleDown(index) {
            const lastIndex = this.size() - 1;
            while (true) {
                let findIndex = index;
                const leftIndex = index * 2 + 1;
                const rightIndex = index * 2 + 2;
                if (
                    leftIndex <= lastIndex &&
                    this.comparator(this.data[leftIndex], this.data[findIndex]) < 0
                ) {
                    findIndex = leftIndex;
                }
                if (
                    rightIndex <= lastIndex &&
                    this.comparator(this.data[rightIndex], this.data[findIndex]) < 0
                ) {
                    findIndex = rightIndex;
                }
                if (index !== findIndex) {
                    this.swap(index, findIndex);
                    index = findIndex;
                } else {
                    break;
                }
            }
        },
        // Swaps two values in the heap
        swap(index1, index2) {
            [this.data[index1], this.data[index2]] = [this.data[index2], this.data[index1]];
        },
        // Returns the number of elements in the heap
        size() {
            return this.data.length;
        }
    };

    heap.heapify();
    return heap;
}

// Example of usage
KthLargestInit(3, [4, 5, 8, 2]);
console.log(kthLargestAdd(3)); // Should return the kth largest element after adding 3
console.log(kthLargestAdd(5)); // Should return the kth largest element after adding 5
console.log(kthLargestAdd(10)); // Should return the kth largest element after adding 10
console.log(kthLargestAdd(9)); // Should return the kth largest element after adding 9
console.log(kthLargestAdd(4)); // Should return the kth largest element after adding 4

```

Time and Space Complexity

Time Complexity:

- The `__init__(self, k: int, nums: List[int])` method has a time complexity of `O(n * log(k))` where `n` is the length of the `nums` list. This is because for every element in nums, the `add` operation is called which takes `O(log(k))` time due to the heap operation and we perform this `n` times.

- The `add(self, val: int) -> int` method has a time complexity of `O(log(k))`. This is because the `heappush` operation could take up to `O(log(k))` time to maintain the heap properties when adding a new element, and similarly `heappop` operation, which is called only when the heap size exceeds `k`, takes `O(log(k))`.

Space Complexity:

- The space complexity of the whole class is `O(k)` since the heap that is maintained by the class never contains more than `k` elements at any time.