1244. Design A Leaderboard

Design a Leaderboard class, which has 3 functions:

1. `addScore(playerId, score)`: Update the leaderboard by adding `score` to the given player's score. If there is no player with such id in the leaderboard, add him to the leaderboard with the given `score`.
2. `top(K)`: Return the score sum of the top `K` players.
3. `reset(playerId)`: Reset the score of the player with the given id to 0. It is guaranteed that the player was added to the leaderboard before calling this function.

Initially, the leaderboard is empty.

Input:
["Leaderboard","addScore","addScore","addScore","addScore","addScore","top","reset","reset","addScore","top"]
[[],[1,73],[2,56],[3,39],[4,51],[5,4],[1],[1],[2],[2,51],[3]]
Output:
[null,null,null,null,null,null,73,null,null,null,141]

Explanation:
Leaderboard leaderboard = new Leaderboard ();
leaderboard.addScore(1,73); // leaderboard = [[1,73]];
leaderboard.addScore(2,56); // leaderboard = [[1,73],[2,56]];
leaderboard.addScore(3,39); // leaderboard = [[1,73],[2,56],[3,39]];
leaderboard.addScore(4,51); // leaderboard = [[1,73],[2,56],[3,39],[4,51]];
leaderboard.addScore(5,4); // leaderboard = [[1,73],[2,56],[3,39],[4,51],[5,4]];
leaderboard.top(1); // returns 73;
leaderboard.reset(1); // leaderboard = [[2,56],[3,39],[4,51],[5,4]];
leaderboard.reset(2); // leaderboard = [[3,39],[4,51],[5,4]];
leaderboard.addScore(2,51); // leaderboard = [[2,51],[3,39],[4,51],[5,4]];
leaderboard.top(3); // returns 141 = 51 + 51 + 39;

```python
import collections
import random


class Leaderboard(object):

    def __init__(self):
        self.__lookup = collections.Counter()
    def addScore(self, playerId, score):
        """
        :type playerId: int
        :type score: int
        :rtype: None
        """
        self.__lookup[playerId] += score

    def top(self, K):
        """
        :type K: int
        :rtype: int
        """
        def kthElement(nums, k, compare):
            def PartitionAroundPivot(left, right, pivot_idx, nums, compare):
                new_pivot_idx = left
                nums[pivot_idx], nums[right] = nums[right], nums[pivot_idx]
                for i in xrange(left, right):
                    if compare(nums[i], nums[right]):
                        nums[i], nums[new_pivot_idx] = nums[new_pivot_idx], nums[i]
                        new_pivot_idx += 1

                nums[right], nums[new_pivot_idx] = nums[new_pivot_idx], nums[right]
                return new_pivot_idx

            left, right = 0, len(nums) - 1
            while left <= right:
                pivot_idx = random.randint(left, right)
                new_pivot_idx = PartitionAroundPivot(left, right, pivot_idx, nums, compare)
                if new_pivot_idx == k:
                    return
                elif new_pivot_idx > k:
                    right = new_pivot_idx - 1
                else:  # new_pivot_idx < k.
                    left = new_pivot_idx + 1

        scores = self.__lookup.values()
        kthElement(scores, K, lambda a, b: a > b)
        return sum(scores[:K])

    def reset(self, playerId):
        """
        :type playerId: int
        :rtype: None
        """
        self.__lookup[playerId] = 0
```

Basically, we are doing quickselect in Top K elements.Now, in Quick-select we know that Kth largest element is at its right place. The elements before it or after it is not sorted. But from Kth index to last, we get the answer as we are not required to sort the elements.