1825. Finding MK Average

My Submissions (/contest/weekly-contest-236/problems/finding-mk-average/submissions/) Back to Contest (/contest/weekly-contest-236/)

User Accepted:	158
User Tried:	1288
Total Accepted:	174
Total Submissions:	3173
Difficulty:	Hard

You are given two integers, `m` and `k`, and a stream of integers. You are tasked to implement a data structure that calculates the **MKAverage** for the stream.

The **MKAverage** can be calculated using these steps:

1. If the number of the elements in the stream is less than `m` you should consider the **MKAverage** to be `-1`. Otherwise, copy the last `m` elements of the stream to a separate container.
2. Remove the smallest `k` elements and the largest `k` elements from the container.
3. Calculate the average value for the rest of the elements **rounded down to the nearest integer**.

Implement the `MKAverage` class:

- `MKAverage(int m, int k)` Initializes the **MKAverage** object with an empty stream and the two integers `m` and `k`.
- `void addElement(int num)` Inserts a new element `num` into the stream.
- `int calculateMKAverage()` Calculates and returns the **MKAverage** for the current stream **rounded down to the nearest integer**.

Example 1:

```
Input
["MKAverage", "addElement", "addElement", "calculateMKAverage", "addElement", "calculateMKAverage", "addElement", "addElement",
[[3, 1], [3], [1], [], [10], [], [5], [5], [5], []]
Output
[null, null, null, -1, null, 3, null, null, null, 5]

Explanation
MKAverage obj = new MKAverage(3, 1);
obj.addElement(3);        // current elements are [3]
obj.addElement(1);        // current elements are [3,1]
obj.calculateMKAverage(); // return -1, because m = 3 and only 2 elements exist.
obj.addElement(10);       // current elements are [3,1,10]
obj.calculateMKAverage(); // The last 3 elements are [3,1,10].
                          // After removing smallest and largest 1 element the container will be [3].
                          // The average of [3] equals 3/1 = 3, return 3
obj.addElement(5);        // current elements are [3,1,10,5]
obj.addElement(5);        // current elements are [3,1,10,5,5]
obj.addElement(5);        // current elements are [3,1,10,5,5,5]
obj.calculateMKAverage(); // The last 3 elements are [5,5,5].
                          // After removing smallest and largest 1 element the container will be [5].
                          // The average of [5] equals 5/1 = 5, return 5
```

Constraints:

- $3 <= m <= 10^5$
- $1 <= k*2 < m$
- $1 <= num <= 10^5$
- At most 10^5 calls will be made to `addElement` and `calculateMKAverage`.

Discuss (https://leetcode.com/problems/finding-mk-average/discuss)

JavaScript

```
class SplayNode {
    constructor(value) {
        this.parent = null;
        this.left = null;
        this.right = null;
        this.val = value;
        this.sum = value;
        this.sz = 1;
```

```
    }
    Update() {
        this.sz = (this.left != null ? this.left.sz : 0) + (this.right != null ? this.right.sz : 0) + 1;
        this.sum = (this.left != null ? this.left.sum : 0) + (this.right != null ? this.right.sum : 0) + this.val;
    }
    IsLeft() {
        return this.parent != null && this.parent.left == this;
    }
    IsRight() {
        return this.parent != null && this.parent.right == this;
    }
    IsRoot(guard = null) {
        return this.parent == guard;
    }
}

class SplayTree {
    constructor() {
        this.root = null;
        this.cmp = (x, y) => x >= y ? 0 : 1;
    }
    Zig(x) { // right rotation
        let y = x.parent;
        if (x.right != null) x.right.parent = y;
        y.left = x.right;
        x.right = y;
        if (y.IsLeft()) {
            y.parent.left = x;
        } else if (y.IsRight()) { // Special attention here for Link-Cut Trees.
            y.parent.right = x;
        }
        x.parent = y.parent;
        y.parent = x;
        y.Update();
        x.Update();
    }

    // Zag:
    //   y               x
    //  / \             / \
    // A   x    --.    y   C
    //    / \          / \
    //   B   C        A   B
    Zag(x) { // left rotation
        let y = x.parent;
        if (x.left != null) x.left.parent = y;
        y.right = x.left;
        x.left = y;
        if (y.IsLeft()) {
            y.parent.left = x;
        } else if (y.IsRight()) { // Special attention here for Link-Cut Trees.
            y.parent.right = x;
        }
        x.parent = y.parent;
        y.parent = x;
        y.Update();
        x.Update();
    }
    ZigZig(x) { // RR
        this.Zig(x.parent);
        this.Zig(x);
    }
    ZigZag(x) { // RL
        this.Zig(x);
        this.Zag(x);
    }
    ZagZag(x) { // LL
        this.Zag(x.parent);
        this.Zag(x);
    }
    ZagZig(x) { // LR
        this.Zag(x);
        this.Zig(x);
    }

    // Splay a "node" just under a "guard", which is default to splay to the "root".
    Splay(node, guard = null) {
```

```
        while (!node.IsRoot(guard)) {
            if (node.parent.IsRoot(guard)) {
                if (node.IsLeft()) {
                    this.Zig(node);
                } else {
                    this.Zag(node);
                }
            } else {
                if (node.parent.IsLeft()) {
                    if (node.IsLeft()) {
                        this.ZigZig(node);
                    } else {
                        this.ZagZig(node);
                    }
                } else {
                    if (node.IsRight()) {
                        this.ZagZag(node);
                    } else {
                        this.ZigZag(node);
                    }
                }
            }
        }
        if (guard == null) this.root = node; // reset "root" to "node".
    }
    LastNode(x) {
        this.Splay(x);
        let node = x.left;
        if (node == null) return null;
        while (node.right != null) node = node.right;
        this.Splay(node);
        return node;
    }
    NextNode(x) {
        this.Splay(x);
        let node = x.right;
        if (node == null) return null;
        while (node.left != null) node = node.left;
        this.Splay(node);
        return node;
    }
    Find(value) {
        return this.FindFirstOf(value);
    }
    FindFirstOf(value) {
        let node = this.root, res = null, last_visited = null;
        while (node != null) {
            last_visited = node;
            if (this.cmp(value, node.val)) {
                node = node.left;
            } else if (this.cmp(node.val, value)) {
                node = node.right;
            } else {
                res = node;
                node = node.left;
            }
        }
        if (last_visited != null) this.Splay(last_visited);
        return res;
    }
    FindLastOf(value) {
        let node = this.root, res = null, last_visited = null;
        while (node != null) {
            last_visited = node;
            if (this.cmp(value, node.val)) {
                node = node.left;
            } else if (this.cmp(node.val, value)) {
                node = node.right;
            } else {
                res = node;
                node = node.right;
            }
        }
        if (last_visited != null) this.Splay(last_visited);
        return res;
    }
    FindRankOf(node) {
```

```
        this.Splay(node);
        return node.left == null ? 0 : node.left.sz;
    }
    FindSuccessorOf(value) {
        let node = this.root, res = null, last_visited = null;
        while (node != null) {
            last_visited = node;
            if (this.cmp(value, node.val)) {
                res = node;
                node = node.left;
            } else {
                node = node.right;
            }
        }
        if (last_visited != null) this.Splay(last_visited);
        return res;
    }
    FindPrecursorOf(value) {
        let node = this.root, res = null, last_visited = null;
        while (node != null) {
            last_visited = node;
            if (this.cmp(node.val, value)) {
                res = node;
                node = node.right;
            } else {
                node = node.left;
            }
        }
        if (last_visited != null) this.Splay(last_visited);
        return res;
    }
    FindKth(rank) {
        if (rank < 0 || rank >= this.Size()) return null;
        let node = this.root;
        while (node != null) {
            let leftsize = node.left == null ? 0 : node.left.sz;
            if (leftsize == rank) break;
            if (leftsize > rank) {
                node = node.left;
            } else {
                rank -= leftsize + 1;
                node = node.right;
            }
        }
        this.Splay(node);
        return node;
    }
    NewNode(value) {
        return new SplayNode(value);
    }
    DeleteNode(node) {
        node = null;
    }

    // ------------------------------- Public Usage -------------------------------------
    Size() {
        return this.root == null ? 0 : this.root.sz;
    }
    IsEmpty() {
        return this.root == null;
    }

    // Insert an element into the container O(log(n))
    // Insert an element into the container O(log(n))
    Insert(value) {
        // pr("insert begin111")
        if (this.root == null) {
            this.root = this.NewNode(value);
            return this.root;
        }
        // pr("insert begin222")
        let node = this.root;
        while (node != null) {
            if (this.cmp(value, node.val)) {
                if (node.left == null) {
                    node.left = this.NewNode(value);
                    node.left.parent = node;
```

```
                    node = node.left;
                    break;
                }
                node = node.left;
            } else {
                if (node.right == null) {
                    node.right = this.NewNode(value);
                    node.right.parent = node;
                    node = node.right;
                    break;
                }
                node = node.right;
            }
        }
        // pr("insert end, prepare splay", node)
        this.Splay(node);
        // pr("splay end")
        return node;
    }

    // Delete an element from the container if it exists O(log n)
    Delete(value) {
        let node = this.Find(value);
        if (node == null) return false;
        this.Splay(node);
        if (node.left == null) {
            this.root = node.right;
            if (node.right != null) node.right.parent = null;
            this.DeleteNode(node);
            return true;
        }
        if (node.right == null) {
            this.root = node.left;
            if (node.left != null) node.left.parent = null;
            this.DeleteNode(node);
            return true;
        }
        let last_node = this.LastNode(node);
        let next_node = this.NextNode(node);
        this.Splay(last_node);
        this.Splay(next_node, last_node);
        // After the above operations, the tree becomes:
        //      last_node
        //     /         \
        //    A       next_node
        //           /         \
        //         node         B
        // Then "next_node.left" is "node".
        this.DeleteNode(next_node.left);
        next_node.left = null;
        next_node.Update();
        last_node.Update();
        return true;
    }

    // Whether the splay tree contains value O(log n).
    Contains(value) {
        return this.CountOf(value) > 0;
    }

    // The number of ocurrences of value O(log n)
    CountOf(value) {
        let x = this.FindFirstOf(value);
        if (x == null) return 0;
        let rank_x = this.FindRankOf(x);
        let y = this.FindLastOf(value);
        let rank_y = this.FindRankOf(y);
        return rank_y - rank_x + 1;
    }

    // The number of elements strictly less than value O(log n)
    RankOf(value) {
        let x = this.FindPrecursorOf(value);
        return x == null ? 0 : this.FindRankOf(x) + 1;
    }

    // Get the k-th element (0-indexed) O(log n).
```

```
    Kth(rank) {
        let x = this.FindKth(rank);
        return x == null ? null : (x.val);
    }

    // Find the smallest element that is strictly greater than value > , if it exists O(log n).
    SuccessorOf(value) {
        let node = this.FindSuccessorOf(value);
        return node == null ? null : (node.val);
    }

    // Find the largest element that is strictly less than value < , if it exists O(log n).
    PrecursorOf(value) {
        let node = this.FindPrecursorOf(value);
        return node == null ? null : (node.val);
    }

    // Get sorted values in the splay tree O(n).
    show() {
        let res = [];
        const dfs = (x) => {
            if (x == null) return;
            dfs(x.left);
            res.push(x.val);
            dfs(x.right);
        };
        dfs(this.root);
        return res;
    }
}

function MKAverage(m, k) {
    let a = [], tree = new SplayTree();
    return { addElement, calculateMKAverage }
    function addElement(x) {
        a.push(x);
        tree.Insert(x);
        if (a.length > m) tree.Delete(a[a.length - m - 1]);
    }
    function calculateMKAverage() {
        if (a.length < m) return -1;
        let res = tree.root.sum;
        let node = tree.FindKth(k);
        if (node.left) res -= node.left.sum;
        node = tree.FindKth(m - k - 1);
        if (node.right) res -= node.right.sum;
        return (res / (m - 2 * k)) >> 0;
    }
}
```

Custom Testcase | Use Example Testcases

Run | Submit

Submission Result: Accepted (/submissions/detail/877819447/) | More Details > (/submissions/detail/877819447/)

Share your acceptance!