

1825. Finding MK Average

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

Back to Contest (/contest/weekly-contest-236/)

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**.

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

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:

- $3 \leq m \leq 10^5$
- $1 \leq k \cdot 2 < m$
- $1 \leq \text{num} \leq 10^5$
- At most  $10^5$  calls will be made to `addElement` and `calculateMKAverage`.

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

JavaScript

📄

↺

⚙️

```
1 class SplayNode {
2   constructor(value) {
3     this.parent = null;
4     this.left = null;
5     this.right = null;
6     this.val = value;
7     this.sum = value;
8     this.sz = 1;
```

```

9      }
10     Update() {
11         this.sz = (this.left != null ? this.left.sz : 0) + (this.right != null ? this.right.sz : 0) + 1;
12         this.sum = (this.left != null ? this.left.sum : 0) + (this.right != null ? this.right.sum : 0) + this.val;
13     }
14     IsLeft() {
15         return this.parent != null && this.parent.left == this;
16     }
17     IsRight() {
18         return this.parent != null && this.parent.right == this;
19     }
20     IsRoot(guard = null) {
21         return this.parent == guard;
22     }
23 }
24
25 class SplayTree {
26     constructor() {
27         this.root = null;
28         this.cmp = (x, y) => x >= y ? 0 : 1;
29     }
30     Zig(x) { // right rotation
31         let y = x.parent;
32         if (x.right != null) x.right.parent = y;
33         y.left = x.right;
34         x.right = y;
35         if (y.IsLeft()) {
36             y.parent.left = x;
37         } else if (y.IsRight()) { // Special attention here for Link-Cut Trees.
38             y.parent.right = x;
39         }
40         x.parent = y.parent;
41         y.parent = x;
42         y.Update();
43         x.Update();
44     }
45
46     // Zag:
47     //   y           x
48     //  / \         / \
49     // A  x  --.  y  C
50     //  / \       / \
51     // B  C       A  B
52     Zag(x) { // left rotation
53         let y = x.parent;
54         if (x.left != null) x.left.parent = y;
55         y.right = x.left;
56         x.left = y;
57         if (y.IsLeft()) {
58             y.parent.left = x;
59         } else if (y.IsRight()) { // Special attention here for Link-Cut Trees.
60             y.parent.right = x;
61         }
62         x.parent = y.parent;
63         y.parent = x;
64         y.Update();
65         x.Update();
66     }
67     ZigZig(x) { // RR
68         this.Zig(x.parent);
69         this.Zig(x);
70     }
71     ZigZag(x) { // RL
72         this.Zig(x);
73         this.Zag(x);
74     }
75     ZagZag(x) { // LL
76         this.Zag(x.parent);
77         this.Zag(x);
78     }
79     ZagZig(x) { // LR
80         this.Zag(x);
81         this.Zig(x);
82     }
83
84     // Splay a "node" just under a "guard", which is default to splay to the "root".
85     Splay(node, guard = null) {

```

```

86 while (!node.IsRoot(guard)) {
87     if (node.parent.IsRoot(guard)) {
88         if (node.IsLeft()) {
89             this.Zig(node);
90         } else {
91             this.Zag(node);
92         }
93     } else {
94         if (node.parent.IsLeft()) {
95             if (node.IsLeft()) {
96                 this.ZigZig(node);
97             } else {
98                 this.ZagZig(node);
99             }
100        } else {
101            if (node.IsRight()) {
102                this.ZagZag(node);
103            } else {
104                this.ZigZag(node);
105            }
106        }
107    }
108 }
109 if (guard == null) this.root = node; // reset "root" to "node".
110 }
111 LastNode(x) {
112     this.Splay(x);
113     let node = x.left;
114     if (node == null) return null;
115     while (node.right != null) node = node.right;
116     this.Splay(node);
117     return node;
118 }
119 NextNode(x) {
120     this.Splay(x);
121     let node = x.right;
122     if (node == null) return null;
123     while (node.left != null) node = node.left;
124     this.Splay(node);
125     return node;
126 }
127 Find(value) {
128     return this.FindFirstOf(value);
129 }
130 FindFirstOf(value) {
131     let node = this.root, res = null, last_visited = null;
132     while (node != null) {
133         last_visited = node;
134         if (this.cmp(value, node.val)) {
135             node = node.left;
136         } else if (this.cmp(node.val, value)) {
137             node = node.right;
138         } else {
139             res = node;
140             node = node.left;
141         }
142     }
143     if (last_visited != null) this.Splay(last_visited);
144     return res;
145 }
146 FindLastOf(value) {
147     let node = this.root, res = null, last_visited = null;
148     while (node != null) {
149         last_visited = node;
150         if (this.cmp(value, node.val)) {
151             node = node.left;
152         } else if (this.cmp(node.val, value)) {
153             node = node.right;
154         } else {
155             res = node;
156             node = node.right;
157         }
158     }
159     if (last_visited != null) this.Splay(last_visited);
160     return res;
161 }
162 FindRankOf(node) {

```

```

163         this.Splay(node);
164         return node.left == null ? 0 : node.left.sz;
165     }
166     FindSuccessorOf(value) {
167         let node = this.root, res = null, last_visited = null;
168         while (node != null) {
169             last_visited = node;
170             if (this.cmp(value, node.val)) {
171                 res = node;
172                 node = node.left;
173             } else {
174                 node = node.right;
175             }
176         }
177         if (last_visited != null) this.Splay(last_visited);
178         return res;
179     }
180     FindPrecursorOf(value) {
181         let node = this.root, res = null, last_visited = null;
182         while (node != null) {
183             last_visited = node;
184             if (this.cmp(node.val, value)) {
185                 res = node;
186                 node = node.right;
187             } else {
188                 node = node.left;
189             }
190         }
191         if (last_visited != null) this.Splay(last_visited);
192         return res;
193     }
194     FindKth(rank) {
195         if (rank < 0 || rank >= this.Size()) return null;
196         let node = this.root;
197         while (node != null) {
198             let leftsize = node.left == null ? 0 : node.left.sz;
199             if (leftsize == rank) break;
200             if (leftsize > rank) {
201                 node = node.left;
202             } else {
203                 rank -= leftsize + 1;
204                 node = node.right;
205             }
206         }
207         this.Splay(node);
208         return node;
209     }
210     NewNode(value) {
211         return new SplayNode(value);
212     }
213     DeleteNode(node) {
214         node = null;
215     }
216
217     // ----- Public Usage -----
218     Size() {
219         return this.root == null ? 0 : this.root.sz;
220     }
221     IsEmpty() {
222         return this.root == null;
223     }
224
225     // Insert an element into the container O(log(n))
226     // Insert an element into the container O(log(n))
227     Insert(value) {
228         // pr("insert begin111")
229         if (this.root == null) {
230             this.root = this.NewNode(value);
231             return this.root;
232         }
233         // pr("insert begin222")
234         let node = this.root;
235         while (node != null) {
236             if (this.cmp(value, node.val)) {
237                 if (node.left == null) {
238                     node.left = this.NewNode(value);
239                     node.left.parent = node;

```

```

240         node = node.left;
241         break;
242     }
243     node = node.left;
244 } else {
245     if (node.right == null) {
246         node.right = this.NewNode(value);
247         node.right.parent = node;
248         node = node.right;
249         break;
250     }
251     node = node.right;
252 }
253 }
254 // pr("insert end, prepare splay", node)
255 this.Splay(node);
256 // pr("splay end")
257 return node;
258 }
259
260 // Delete an element from the container if it exists O(log n)
261 Delete(value) {
262     let node = this.Find(value);
263     if (node == null) return false;
264     this.Splay(node);
265     if (node.left == null) {
266         this.root = node.right;
267         if (node.right != null) node.right.parent = null;
268         this.DeleteNode(node);
269         return true;
270     }
271     if (node.right == null) {
272         this.root = node.left;
273         if (node.left != null) node.left.parent = null;
274         this.DeleteNode(node);
275         return true;
276     }
277     let last_node = this.LastNode(node);
278     let next_node = this.NextNode(node);
279     this.Splay(last_node);
280     this.Splay(next_node, last_node);
281     // After the above operations, the tree becomes:
282     //      last_node
283     //      /      \
284     //   A        next_node
285     //      /      \
286     //     node      B
287     // Then "next_node.left" is "node".
288     this.DeleteNode(next_node.left);
289     next_node.left = null;
290     next_node.Update();
291     last_node.Update();
292     return true;
293 }
294
295 // Whether the splay tree contains value O(log n).
296 Contains(value) {
297     return this.CountOf(value) > 0;
298 }
299
300 // The number of occurrences of value O(log n)
301 CountOf(value) {
302     let x = this.FindFirstOf(value);
303     if (x == null) return 0;
304     let rank_x = this.FindRankOf(x);
305     let y = this.FindLastOf(value);
306     let rank_y = this.FindRankOf(y);
307     return rank_y - rank_x + 1;
308 }
309
310 // The number of elements strictly less than value O(log n)
311 RankOf(value) {
312     let x = this.FindPrecursorOf(value);
313     return x == null ? 0 : this.FindRankOf(x) + 1;
314 }
315
316 // Get the k-th element (0-indexed) O(log n).

```

```

317 Kth(rank) {
318     let x = this.FindKth(rank);
319     return x == null ? null : (x.val);
320 }
321
322 // Find the smallest element that is strictly greater than value > , if it exists O(log n).
323 SuccessorOf(value) {
324     let node = this.FindSuccessorOf(value);
325     return node == null ? null : (node.val);
326 }
327
328 // Find the largest element that is strictly less than value < , if it exists O(log n).
329 PrecursorOf(value) {
330     let node = this.FindPrecursorOf(value);
331     return node == null ? null : (node.val);
332 }
333
334 // Get sorted values in the splay tree O(n).
335 show() {
336     let res = [];
337     const dfs = (x) => {
338         if (x == null) return;
339         dfs(x.left);
340         res.push(x.val);
341         dfs(x.right);
342     };
343     dfs(this.root);
344     return res;
345 }
346 }
347
348 function MKAverage(m, k) {
349     let a = [], tree = new SplayTree();
350     return { addElement, calculateMKAverage }
351     function addElement(x) {
352         a.push(x);
353         tree.Insert(x);
354         if (a.length > m) tree.Delete(a[a.length - m - 1]);
355     }
356     function calculateMKAverage() {
357         if (a.length < m) return -1;
358         let res = tree.root.sum;
359         let node = tree.FindKth(k);
360         if (node.left) res -= node.left.sum;
361         node = tree.FindKth(m - k - 1);
362         if (node.right) res -= node.right.sum;
363         return (res / (m - 2 * k)) >> 0;
364     }
365 }

```

☐ Custom Testcase[Use Example Testcases](#)[Run](#)[Submit](#)Submission Result: **Accepted** (/submissions/detail/877819447/) ?[More Details](#) (/submissions/detail/877819447/)

Share your acceptance!

Copyright © 2023 LeetCode

[Help Center](#) (/support) | [Jobs](#) (/jobs) | [Bug Bounty](#) (/bugbounty) | [Online Interview](#) (/interview/) | [Students](#) (/student) | [Terms](#) (/terms) | [Privacy Policy](#) (/privacy) [United States](#) (/region)