

```

1. class Treap {
2.     static minstd_rand generator;
3.
4.     struct Node {
5.         int priority, size, value, min_value, add = 0, rev = 0;
6.         Node *l = nullptr, *r = nullptr;
7.         Node (int value): value(value), priority(generator()), size(1), min_value(value) {}
8.     } *root = nullptr;
9.
10.    static int get_size(Node *n){
11.        return n ? n -> size : 0;
12.    }
13.
14.    static int get_min_value(Node *n){
15.        return n ? n -> min_value + n -> add : INF;
16.    }
17.
18.    static void push(Node *n){
19.        if (n){
20.            if (n -> add){
21.                n -> value += n -> add;
22.                n -> min_value += n -> add;
23.                if (n -> l){
24.                    n -> l -> add += n -> add;
25.                }
26.                if (n -> r){
27.                    n -> r -> add += n -> add;
28.                }
29.                n -> add = 0;
30.            }
31.            if (n -> rev){
32.                swap(n -> l, n -> r);
33.                n -> l -> rev ^= 1;
34.                n -> r -> rev ^= 1;
35.                n -> rev = 0;
36.            }
37.        }
38.    }
39.
40.    static void update(Node *n){
41.        if (n){
42.            n -> size = get_size(n -> l) + 1 + get_size(n -> r);
43.            n -> min_value = min(get_min_value(n -> l), min(n -> value, get_min_value(n -> r)));
44.        }
45.    }
46.
47.    static Node *merge(Node *a, Node *b){
48.        push(a);
49.        push(b);
50.        if (!a || !b){
51.            return a ? a : b;
52.        }
53.        if (a->priority > b->priority){
54.            a->r = merge(a->r, b);
55.            update(a);
56.            return a;
57.        }
58.        else {
59.            b->l = merge(a, b->l);
60.            update(b);
61.            return b;
62.        }
63.    }
64.
65.    static void split(Node *n, int k, Node *&a, Node *&b){
66.        push(n);
67.        if (!n){
68.            a = b = nullptr;
69.            return ;
70.        }
71.        if (get_size(n -> l) < k){
72.            split(n->r, k - get_size(n -> l) - 1, n->r, b);
73.            a = n;
74.        }
75.        else {
76.            split(n->l, k, a, n->l);
77.            b = n;
78.        }
79.        update(a);
80.        update(b);
81.    }
82.
83.    void show_tree(Node *n){
84.        if (!n)
85.            return ;
86.        cout << n -> value << endl;
87.        show_tree(n -> l);
88.        show_tree(n -> r);

```

```

89.     }
90.
91. public:
92.
93.     int get(int index){
94.         Node *greater, *equal, *less;
95.         split(root, index, less, greater);
96.         split(greater, 1, equal, greater);
97.         int result = equal -> value;
98.         root = merge(merge(less, equal), greater);
99.         return result;
100.    }
101.
102.    void push_back(int value){
103.        root = merge(root, new Node(value));
104.    }
105.
106.    void push_front(int value){
107.        root = merge(new Node(value), root);
108.    }
109.
110.    void insert(int index, int value){
111.        Node *greater, *less;
112.        split(root, index, less, greater);
113.        root = merge(merge(less, new Node(value)), greater);
114.    }
115.
116.    void erase(int index){
117.        Node *greater, *equal, *less;
118.        split(root, index, less, greater);
119.        split(greater, 1, equal, greater);
120.        root = merge(less, greater);
121.    }
122.
123.    void erase(int l, int r){
124.        Node *greater, *equal, *less;
125.        split(root, l, less, greater);
126.        split(greater, r - l + 1, equal, greater);
127.        root = merge(less, greater);
128.    }
129.
130.    void movetofront(int l, int r){
131.        Node *greater, *equal, *less;
132.        split(root, l, less, greater);
133.        split(greater, r - l + 1, equal, greater);
134.        root = merge(merge(equal, less), greater);
135.    }
136.
137.    void revolve(int l, int r, int x){
138.        Node *greater, *equal, *less;
139.        split(root, l, less, greater);
140.        split(greater, r - l + 1, equal, greater);
141.        int len = get_size(equal);
142.        x %= len;
143.        // переставляем x последних элементов в начало
144.        Node *left, *right;
145.        split(equal, len - x, left, right);
146.        equal = merge(right, left);
147.        root = merge(merge(less, equal), greater);
148.    }
149.
150.    int get_min(int l, int r){
151.        Node *greater, *equal, *less;
152.        split(root, l, less, greater);
153.        split(greater, r - l + 1, equal, greater);
154.        int result = get_min_value(equal);
155.        root = merge(merge(less, equal), greater);
156.        return result;
157.    }
158.
159.    void range_add(int l, int r, int value){
160.        Node *greater, *equal, *less;
161.        split(root, l, less, greater);
162.        split(greater, r - l + 1, equal, greater);
163.        equal -> add += value;
164.        root = merge(merge(less, equal), greater);
165.    }
166.
167.    void reverse(int l, int r){
168.        Node *greater, *equal, *less;
169.        split(root, l, less, greater);
170.        split(greater, r - l + 1, equal, greater);
171.        equal -> rev ^= 1;
172.        root = merge(merge(less, equal), greater);
173.    }
174.
175.    void show_tree(){
176.        show_tree(root);

```

```
177.     }
178.
179.     int size(){
180.         return get_size(root);
181.     }
182. };
183.
184. minstd_rand Treap::generator
```