

# FHQ-treap (无旋treap)

正如其名，这种树的本质上还是一棵treap，关于treap的定义，它是一种tree (BST) 和 heap (堆) 的缝合怪，通过tree的性质来维护数据结构，实现功能，通过堆的性质来维护树的深度，防止其退化成一条链。treap维护深度的方式是通过随机化，对每一个结点随即生成一个堆关键字，使用堆关键字来维护一个堆的性质。当数据完全随机的时候，可以认为其深度永远是 $\log N$ 级别的。

作为一个平衡树，treap也少不了烦人的左旋右旋，而FHQ treap可以说是一个码农福音，它以特殊的方式实现treap的所有功能，避免了烦人的旋转过程，除此之外，它还扩展了treap的功能，让它拥有了splay的区间操作功能。

FHQ-treap的主要两个操作包括split (分裂) 和merge (合并) 。

前置代码：

```
1  #include<bits/stdc++.h>
2  using namespace std;
3  const int maxn = 500005;
4  struct node//fhq treap的一个结点
5  {
6      int val;//存储的数据值
7      int key;//维护堆用的关键字
8      int siz;//树的大小
9      int l, r;//左右儿子
10     node() {}
11     node(int _val) :val(_val) {}
12 }tree[maxn];
13 int tot;//已经开辟的结点数
14 int root;//树的根
15 void pushup(int x)
16 {
17     tree[x].siz = tree[tree[x].l].siz + tree[tree[x].r].siz + 1;//维护子树大小
18     ///////////////////////////////////////////////////
19     //更多需要维护的东西
20     ///////////////////////////////////////////////////
21 }
22 int newnode(int x)//开辟新结点
23 {
24     tree[++tot].val = x;
25     tree[tot].key = rand();
26     tree[tot].siz = 1;
27     tree[tot].l = tree[tot].r = 0;
28     return tot;
29 }
```

split操作有两种，第一种是按权值分裂，第二种是按排名分裂(用于文艺平衡树) 。

分裂，对于FHQ-treap的含义是将整一棵树分裂成两棵树，而分裂的时候遵循了一定的规则。第一种权值分裂就是将树分裂成两棵树，其中一棵的所有结点都小于等于要插入的结点，另一棵树的结点都大于等于要插入的结点。

代码：

```

1 void Split(int now, int k, int& x, int& y)//分裂，参数分别为当前结点，分裂界限k，小
  值树根x，大值树根y
2 {
3     if (!now)x = y = 0;//如果已经没有子树要分裂，就让大小值树全部为0，代表无子结点
4     else
5     {
6         if (tree[now].val <= k)//如果当前结点的值小于等于k，那么左子树必然全部插入小
          值树中，右子树仍然存在小于等于k的数字
7         //因此，将当前直接替换小值树的根，从当前结点的右子树开始继续分裂，而当前结点
          的右子树就作为小值树新的根
8         {
9             x = now;
10            Split(tree[now].r, k, tree[now].r, y);
11        }
12        else//同理，当当前结点的值大于k的时候，右子树可以全部插入大值树，向左子树继续分裂
13        {
14            y = now;
15            Split(tree[now].l, k, x, tree[now].l);
16        }
17        pushup(now);//更新当前点数据
18    }
19 }

```

第二个操作是合并，合并之前肯定保证了小值树的所有值都小于大值树，也就是x的所有子树结点都大于y的子树，在合并的时候只需要考虑堆关键字的大小，确定谁为根即可。

代码：

```

1 int Merge(int x, int y)//合并，x为小值树子树的根，y为大值树子树的根，返回值是这一步合
  并后的根节点
2 {
3     if (!x || !y)return x + y;//如果某一个树没有了，直接把另一个树作为子树新的根返回
4     if (tree[x].key < tree[y].key)
5     {
6         tree[x].r = Merge(tree[x].r, y);//如果小值树当前子树的根节点比大值树当前子
          树的根节点小，就让小值树为根，大值树和小值树的右子树合并
7         pushup(x);//更新数据
8         return x;
9     }
10    else
11    {
12        tree[y].l = Merge(x, tree[y].l);//否则就让大值树当前子树根结点为根，小值树
          和大值树当前结点的左子树合并
13        pushup(y);
14        return y;
15    }
16 }

```

插入操作，插入的时候将原树分裂成大于插入值和小于等于插入值的两部分，将插入值和小的树合并，再将小树和大树合并即可。

```

1 void insert(node val)//插入结点val
2 {
3     int x = 0, y = 0;//存储大小值树
4     Split(root, val.val, x, y);//将原树进行分裂
5     root = Merge(Merge(x, newnode(val.val)), y);//首先合并新插入的节点和小值树，然
    后合并小值树和大值树
6 }

```

删除操作，将原树以删除值为界分开，再将小值树分开，分界是删除值-1。分离出来的第三棵树就是以要删除的结点为根的，将要删除的树左右子树合并成一棵新树，再合并到小值树中，最后和大值树合并即可完成删除。（对于重复值只会删除一个）

代码：

```

1 void del(int val)//删除排序关键字为val的结点
2 {
3     int x = 0, y = 0, z = 0;//存储三棵树
4     Split(root, val, x, y);//分裂原树为大小值树
5     Split(x, val - 1, x, z);//将小值树分裂两部分，val-1为界限
6     z = Merge(tree[z].l, tree[z].r);//合并分裂出的要删除点的左右儿子
7     root = Merge(Merge(x, z), y);//合并三棵树
8 }

```

获取排名，将原树以查询值为界限分裂，小值树的大小就是排名。

代码：

```

1 int Rank(node val)//获得数据val的排名
2 {
3     int x = 0, y = 0;
4     Split(root, val.val - 1, x, y);
5     int ans = tree[x].siz + 1;
6     root = Merge(x, y);
7     return ans;
8 }

```

获取第k小，和其他平衡树获取第k小相同。

代码：

```

1 node getkth(int _root, int k)//获取第k小
2 {
3     if (k == 0)return node(0);
4     int tmp = _root;
5     while (1)
6     {
7         if (k == tree[tmp].l.siz + 1)return tree[tmp];//如果当前结点的左
            子树大小+1等于k，说明当前结点就是第k小
8         if (tree[tmp].l && k <= tree[tmp].l.siz)tmp = tree[tmp].l;//左
            子树存在且k小于等于左子树大小，说明第k小在左子树
9         else k -= (tree[tmp].l ? tree[tmp].l.siz : 0) + 1, tmp =
            tree[tmp].r;//否则第k小在右子树，在右子树中查找第(k-左子树大小-1)小的数字
10    }
11 }
12 node getkth(int k)//单参数求第k小
13 {

```

```

14     return getkth(root, k);
15 }

```

获取前驱后继，分别以查询值和查询值-1为界限分裂，分别查找小值树最大和大值树最小就是前驱和后继。

代码：

```

1  node getpre(node k)//求前驱
2  {
3      int x = 0, y = 0;
4      Split(root, k.val - 1, x, y);//将树以k-1为关键字拆分
5      node ans = getkth(x, tree[x].siz);//求左子树的最大值就是前驱
6      root = Merge(x, y);
7      return ans;
8  }
9
10 node getnxt(node k)//求后继
11 {
12     int x = 0, y = 0;
13     Split(root, k.val, x, y);//将树以k为关键字拆分
14     node ans = getkth(y, 1);//求右子树的最小值就是后继
15     root = Merge(x, y);
16     return ans;
17 }

```

FHQ-treap文艺平衡树

前置代码：

```

1  struct node//fhq treap的一个结点
2  {
3      int val;//存储的数据值
4      int key;//维护堆用的关键字
5      int siz;//树的大小
6      int l, r;//左右儿子
7      ///////////////////////////////////////////////////
8      //文艺平衡树
9      int tag;
10     ///////////////////////////////////////////////////
11     node() {}
12     node(int _val) :val(_val) {}
13 }tree[maxn];
14
15 void pushdown(int x)
16 {
17     if (x && tree[x].tag)
18     {
19         tree[x].tag ^= 1;
20         swap(tree[x].l, tree[x].r);
21         if (tree[x].l)tree[tree[x].l].tag ^= 1;
22         if (tree[x].r)tree[tree[x].r].tag ^= 1;
23     }
24 }

```

这里需要用到按排名进行分裂，大体思路和之前的分裂是相同的。

代码：

```
1 void split(int now, int k, int& x, int& y)//按排名分裂，参数分别为当前结点，分裂界
   限k，小值树根x，大值树根y
2 {
3     if (!now)x = y = 0;//如果已经没有子树要分裂，就让大小值树全部为0，代表无子结点
4     else
5     {
6         pushdown(now);
7         if (k <= tree[tree[now].l].siz)
8         {
9             y = now;
10            split(tree[now].l, k, x, tree[now].l);
11        }
12        else
13        {
14            x = now;
15            split(tree[now].r, k - tree[tree[now].l].siz - 1, tree[now].r,
y);
16        }
17        pushup(now);//更新当前点数据
18    }
19 }
```

对之前的合并函数进行小小的修改，在合并操作之前，先对x和y两个结点都进行一次pushdown操作。

对于翻转，需要用分裂函数将l到r这个区间的树分裂出来，然后更新分裂出来的树根的翻转标记。

输出的时候只需要从树根开始中序遍历即可。

代码：

```
1 void rever(int l, int r)//翻转
2 {
3     int a, b, c, d;
4     split(root, r, a, b);//分裂r
5     split(a, l - 1, c, d);//分裂l，得到l到r的区间
6     tree[d].tag ^= 1;//给区间的根节点更新翻转标记
7     root = Merge(Merge(c, d), b);//合并
8 }
9 void print(int x)//中序遍历
10 {
11     if (!x)return;
12     pushdown(x);
13     print(tree[x].l);
14     printf("%d ", tree[x].val);
15     print(tree[x].r);
16 }
```

全部代码：

```

1  #include<bits/stdc++.h>
2  using namespace std;
3  const int maxn = 500005;
4  struct node//fhq treap的一个结点
5  {
6      int val;//存储的数据值
7      int key;//维护堆用的关键字
8      int siz;//树的大小
9      int l, r;//左右儿子
10     ///////////////////////////////////////////////////
11     //文艺平衡树
12     int tag;
13     ///////////////////////////////////////////////////
14     node() {}
15     node(int _val) :val(_val) {}
16 }tree[maxn];
17 int tot;//已经开辟的结点数目
18 int root;//树的根
19 void pushup(int x)
20 {
21     tree[x].siz = tree[tree[x].l].siz + tree[tree[x].r].siz + 1;//维护子树大
22     小
23     ///////////////////////////////////////////////////
24     //更多需要维护的东西
25     ///////////////////////////////////////////////////
26 }
27 ///////////////////////////////////////////////////
28 //文艺平衡树
29 void pushdown(int x)
30 {
31     if (x && tree[x].tag)
32     {
33         tree[x].tag ^= 1;
34         swap(tree[x].l, tree[x].r);
35         if (tree[x].l)tree[tree[x].l].tag ^= 1;
36         if (tree[x].r)tree[tree[x].r].tag ^= 1;
37     }
38 }
39 ///////////////////////////////////////////////////
40
41 int newnode(int x)//开辟新结点
42 {
43     tree[++tot].val = x;
44     tree[tot].key = rand();
45     tree[tot].siz = 1;
46     tree[tot].l = tree[tot].r = 0;
47     return tot;
48 }
49
50 ///////////////////////////////////////////////////
51 //文艺平衡树
52 void split(int now, int k, int& x, int& y)//按排名分裂，参数分别为当前结点，分裂
53     界限k，小值树根x，大值树根y
54 {
55     if (!now)x = y = 0;//如果已经没有子树要分裂，就让大小值树全部为0，代表无子结点
56     {

```

```

57     pushdown(now);
58     if (k <= tree[tree[now].l].siz)
59     {
60         y = now;
61         split(tree[now].l, k, x, tree[now].l);
62     }
63     else
64     {
65         x = now;
66         split(tree[now].r, k - tree[tree[now].l].siz - 1, tree[now].r,
y);
67     }
68     pushup(now); //更新当前点数据
69 }
70 }
71 ///////////////////////////////////////////////////
72
73 void split(int now, int k, int& x, int& y) //分裂，参数分别为当前结点，分裂界限k，
小值树根x，大值树根y
74 {
75     if (!now) x = y = 0; //如果已经没有子树要分裂，就让大小值树全部为0，代表无子结点
76     else
77     {
78         if (tree[now].val <= k) //如果当前结点的值小于等于k，那么左子树必然全部插入
小值树中，右子树仍然存在小于等于k的数字
79             //因此，将当前直接替换小值树的根，从当前结点的右子树开始继续分裂，而当前结
点的右子树就作为小值树新的根
80             {
81                 x = now;
82                 split(tree[now].r, k, tree[now].r, y);
83             }
84         else //同理，当当前结点的值大于k的时候，右子树可以全部插入大值树，向左子树继续分
裂
85             {
86                 y = now;
87                 split(tree[now].l, k, x, tree[now].l);
88             }
89         pushup(now); //更新当前点数据
90     }
91 }
92
93 int Merge(int x, int y) //合并，x为小值树子树的根，y为大值树子树的根，返回值是这一步合
并后的根节点
94 {
95     if (!x || !y) return x + y; //如果某一个树没有了，直接把另一个树作为子树新的根返
回
96     ///////////////////////////////////////////////////
97     //文艺平衡树
98     //pushdown(x);
99     //pushdown(y);
100    ///////////////////////////////////////////////////
101    if (tree[x].key < tree[y].key)
102    {
103        tree[x].r = Merge(tree[x].r, y); //如果小值树当前子树的根节点比大值树当前
子树的根节点小，就让小值树为根，大值树和小值树的右子树合并
104        pushup(x); //更新数据
105        return x;
106    }

```

```

107     else
108     {
109         tree[y].l = Merge(x, tree[y].l); //否则就让大值树当前子树根结点为根，小值
树和大值树当前结点的左子树合并
110         pushup(y);
111         return y;
112     }
113 }
114 ///////////////////////////////////////////////////////////////////
115 //文艺平衡树
116 void rever(int l, int r) //翻转
117 {
118     int a, b, c, d;
119     split(root, r, a, b); //分裂r
120     split(a, l - 1, c, d); //分裂l，得到l到r的区间
121     tree[d].tag ^= 1; //给区间的根节点更新翻转标记
122     root = Merge(Merge(c, d), b); //合并
123 }
124 void print(int x) //中序遍历
125 {
126     if (!x) return;
127     pushdown(x);
128     print(tree[x].l);
129     printf("%d ", tree[x].val);
130     print(tree[x].r);
131 }
132 ///////////////////////////////////////////////////////////////////
133 void insert(node val) //插入结点val
134 {
135     int x = 0, y = 0; //存储大小值树
136     split(root, val.val, x, y); //将原树进行分裂
137     root = Merge(Merge(x, newnode(val.val)), y); //首先合并新插入的节点和小值树，
然后合并小值树和大值树
138 }
139
140 void del(int val) //删除排序关键字为val的结点
141 {
142     int x = 0, y = 0, z = 0; //存储三棵树
143     split(root, val, x, y); //分裂原树为大小值树
144     split(x, val - 1, x, z); //将小值树分裂两部分，val-1为界限
145     z = Merge(tree[z].l, tree[z].r); //合并分裂出的要删除点的左右儿子
146     root = Merge(Merge(x, z), y); //合并三棵树
147 }
148
149 int Rank(node val) //获得数据val的排名
150 {
151     int x = 0, y = 0;
152     split(root, val.val - 1, x, y);
153     int ans = tree[x].siz + 1;
154     root = Merge(x, y);
155     return ans;
156 }
157
158 node getkth(int _root, int k) //获取第k小
159 {
160     if (k == 0) return node(0);
161     int tmp = _root;
162     while (1)

```



```

163     {
164         if (k == tree[tree[tmp].l].siz + 1) return tree[tmp]; //如果当前结点的
//左子树大小+1等于k，说明当前结点就是第k小
165         if (tree[tmp].l && k <= tree[tree[tmp].l].siz) tmp = tree[tmp].l; //
//左子树存在且k小于等于左子树大小，说明第k小在左子树
166         else k -= (tree[tmp].l ? tree[tree[tmp].l].siz : 0) + 1, tmp =
tree[tmp].r; //否则第k小在右子树，在右子树中查找第(k-左子树大小-1)小的数字
167     }
168 }
169
170 node getkth(int k) //单参数求第k小
171 {
172     return getkth(root, k);
173 }
174
175 node getpre(node k) //求前驱
176 {
177     int x = 0, y = 0;
178     Split(root, k.val - 1, x, y); //将树以k-1为关键字拆分
179     node ans = getkth(x, tree[x].siz); //求左子树的最大值就是前驱
180     root = Merge(x, y);
181     return ans;
182 }
183
184 node getnxt(node k) //求后继
185 {
186     int x = 0, y = 0;
187     Split(root, k.val, x, y); //将树以k为关键字拆分
188     node ans = getkth(y, 1); //求右子树的最小值就是后继
189     root = Merge(x, y);
190     return ans;
191 }
192 int main()
193 {
194     //////////////////////////////////////
195     //文艺平衡树
196     //int n, m;
197     //cin >> n >> m;
198     //for (int i = 1; i <= n; i++)
199     //{
200     //    insert(i);
201     //}
202     //while (m--)
203     //{
204     //    int l, r;
205     //    scanf("%d%d", &l, &r);
206     //    rever(l, r);
207     //}
208     //print(root);
209     //////////////////////////////////////
210     int n;
211     cin >> n;
212     while (n--)
213     {
214         int opt, x;
215         scanf("%d%d", &opt, &x);
216         switch (opt)
217         {

```

```
218         case 1:insert(node(x)); break;
219         case 2:del(x); break;
220         case 3:printf("%d\n", Rank(node(x))); break;
221         case 4:printf("%d\n", getkth(x).val); break;
222         case 5:printf("%d\n", getpre(node(x)).val); break;
223         case 6:printf("%d\n", getnxt(node(x)).val); break;
224         default:
225             break;
226     }
227 }
228 return 0;
229 }
```