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：

```
#include<bits/stdc++.h>
using namespace std;
const int maxn = 500005;
struct node//fhq treap的一个结点
{
    int val;//存储的数据值
    int key;//维护堆用的关键字
    int siz;//树的大小
    int l, r;//左右儿子
    node() {}
    node(int _val) :val(_val) {}
}tree[maxn];
int tot;//已经开辟的结点数目
int root;//树的根
void pushup(int x)
{
    tree[x].siz = tree[tree[x].l].siz + tree[tree[x].r].siz + 1;//维护子树大小
    ////////////////////////////////////////////////////////////////////
    //更多需要维护的东西
    ////////////////////////////////////////////////////////////////////
}
int newnode(int x)//开辟新结点
{
    tree[++tot].val = x;
    tree[tot].key = rand();
    tree[tot].siz = 1;
    tree[tot].l = tree[tot].r = 0;
    return tot;
}
```

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：

```
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
{
    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
    else
    {
        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，右子树仍然可能存在小于等于k的数字
            //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
        {
            x = now;
            Split(tree[now].r, k, tree[now].r, y);
        }
        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
        {
            y = now;
            Split(tree[now].l, k, x, tree[now].l);
        }
        pushup(now);//更新当前点数据
    }
}
```

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：

```
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
{
    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
    if (tree[x].key < tree[y].key)
    {
        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
        pushup(x);//更新数据
        return x;
    }
    else
    {
        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
        pushup(y);
        return y;
    }
}
```

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。

```
void insert(node val)//插入结点val
{
    int x = 0, y = 0;//存储大小值树
    Split(root, val.val, x, y);//将原树进行分裂
    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
}
```

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：

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

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：

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

获取第k小，和其他平衡树获取第k小相同。

代码：

```
node getkth(int _root, int k)//获取第k小
{
    if (k == 0)return node(0);
    int tmp = _root;
    while (1)
    {
        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
        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
        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
    }
}
node getkth(int k)//单参数求第k小
{
```

```
    return getkth(root, k);
}
```

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：

```
node getpre(node k)//求前驱
{
    int x = 0, y = 0;
    Split(root, k.val - 1, x, y);//将树以k-1为关键字拆分
    node ans = getkth(x, tree[x].siz);//求左子树的最大值就是前驱
    root = Merge(x, y);
    return ans;
}

node getnxt(node k)//求后继
{
    int x = 0, y = 0;
    Split(root, k.val, x, y);//将树以k为关键字拆分
    node ans = getkth(y, 1);//求右子树的最小值就是后继
    root = Merge(x, y);
    return ans;
}
```

FHQ-treap文艺平衡树

前置代码：

```
struct node//fhq treap的一个结点
{
    int val;//存储的数据值
    int key;//维护堆用的关键字
    int siz;//树的大小
    int l, r;//左右儿子
    ////////////////////////////////////////////////////////////////////
    //文艺平衡树
    int tag;
    ////////////////////////////////////////////////////////////////////
    node() {}
    node(int _val) :val(_val) {}
}tree[maxn];

void pushdown(int x)
{
    if (x && tree[x].tag)
    {
        tree[x].tag ^= 1;
        swap(tree[x].l, tree[x].r);
        if (tree[x].l)tree[tree[x].l].tag ^= 1;
        if (tree[x].r)tree[tree[x].r].tag ^= 1;
    }
}
```

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：

```
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
{
    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
    else
    {
        pushdown(now);
        if (k <= tree[tree[now].l].siz)
        {
            y = now;
            split(tree[now].l, k, x, tree[now].l);
        }
        else
        {
            x = now;
            split(tree[now].r, k - tree[tree[now].l].siz - 1, tree[now].r, y);
        }
        pushup(now);//更新当前点数据
    }
}
```

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：

```
void rever(int l, int r)//翻转
{
    int a, b, c, d;
    split(root, r, a, b);//分裂r
    split(a, l - 1, c, d);//分裂l，得到l到r的区间
    tree[d].tag ^= 1;//给区间的根节点更新翻转标记
    root = Merge(Merge(c, d), b);//合并
}
void print(int x)//中序遍历
{
    if (!x)return;
    pushdown(x);
    print(tree[x].l);
    printf("%d ", tree[x].val);
    print(tree[x].r);
}
```

全部代码：

```cpp
#include<bits/stdc++.h>
using namespace std;
const int maxn = 500005;
struct node//fhq treap的一个结点
{
    int val;//存储的数据值
    int key;//维护堆用的关键字
    int siz;//树的大小
    int l, r;//左右儿子
    ////////////////////////////////////////////////////////////////////
    //文艺平衡树
    int tag;
    ////////////////////////////////////////////////////////////////////
    node() {}
    node(int _val) :val(_val) {}
}tree[maxn];
int tot;//已经开辟的结点数目
int root;//树的根
void pushup(int x)
{
    tree[x].siz = tree[tree[x].l].siz + tree[tree[x].r].siz + 1;//维护子树大小
    ////////////////////////////////////////////////////////////////////
    //更多需要维护的东西
    ////////////////////////////////////////////////////////////////////
}

////////////////////////////////////////////////////////////////////
//文艺平衡树
void pushdown(int x)
{
    if (x && tree[x].tag)
    {
        tree[x].tag ^= 1;
        swap(tree[x].l, tree[x].r);
        if (tree[x].l)tree[tree[x].l].tag ^= 1;
        if (tree[x].r)tree[tree[x].r].tag ^= 1;
    }
}
////////////////////////////////////////////////////////////////////

int newnode(int x)//开辟新结点
{
    tree[++tot].val = x;
    tree[tot].key = rand();
    tree[tot].siz = 1;
    tree[tot].l = tree[tot].r = 0;
    return tot;
}

////////////////////////////////////////////////////////////////////
//文艺平衡树
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
{
    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
    else
    {
```

```
        pushdown(now);
        if (k <= tree[tree[now].l].siz)
        {
            y = now;
            split(tree[now].l, k, x, tree[now].l);
        }
        else
        {
            x = now;
            split(tree[now].r, k - tree[tree[now].l].siz - 1, tree[now].r,
y);
        }
        pushup(now);//更新当前点数据
    }
}
////////////////////////////////////////////////////////////////////////

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
{
    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
    else
    {
        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，右子树仍然可能存在小于等于k的数字
            //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
        {
            x = now;
            Split(tree[now].r, k, tree[now].r, y);
        }
        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
        {
            y = now;
            Split(tree[now].l, k, x, tree[now].l);
        }
        pushup(now);//更新当前点数据
    }
}

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
{
    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
    ////////////////////////////////////////////////////////////////////
    //文艺平衡树
    //pushdown(x);
    //pushdown(y);
    ////////////////////////////////////////////////////////////////////
    if (tree[x].key < tree[y].key)
    {
        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
        pushup(x);//更新数据
        return x;
    }
```

```
    else
    {
        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
        pushup(y);
        return y;
    }
}
////////////////////////////////////////////////////////////////////
//文艺平衡树
void rever(int l, int r)//翻转
{
    int a, b, c, d;
    split(root, r, a, b);//分裂r
    split(a, l - 1, c, d);//分裂l，得到l到r的区间
    tree[d].tag ^= 1;//给区间的根节点更新翻转标记
    root = Merge(Merge(c, d), b);//合并
}
void print(int x)//中序遍历
{
    if (!x)return;
    pushdown(x);
    print(tree[x].l);
    printf("%d ", tree[x].val);
    print(tree[x].r);
}
////////////////////////////////////////////////////////////////////
void insert(node val)//插入结点val
{
    int x = 0, y = 0;//存储大小值树
    Split(root, val.val, x, y);//将原树进行分裂
    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
}

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

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

node getkth(int _root, int k)//获取第k小
{
    if (k == 0)return node(0);
    int tmp = _root;
    while (1)
```

```
    {
        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
        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
        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
    }
}

node getkth(int k)//单参数求第k小
{
    return getkth(root, k);
}

node getpre(node k)//求前驱
{
    int x = 0, y = 0;
    Split(root, k.val - 1, x, y);//将树以k-1为关键字拆分
    node ans = getkth(x, tree[x].siz);//求左子树的最大值就是前驱
    root = Merge(x, y);
    return ans;
}

node getnxt(node k)//求后继
{
    int x = 0, y = 0;
    Split(root, k.val, x, y);//将树以k为关键字拆分
    node ans = getkth(y, 1);//求右子树的最小值就是后继
    root = Merge(x, y);
    return ans;
}
int main()
{
    ////////////////////////////////////////////////////////////////////////
    //文艺平衡树
    //int n, m;
    //cin >> n >> m;
    //for (int i = 1; i <= n; i++)
    //{
    //  insert(i);
    //}
    //while (m--)
    //{
    //  int l, r;
    //  scanf("%d%d", &l, &r);
    //  rever(l, r);
    //}
    //print(root);
    ////////////////////////////////////////////////////////////////////////
    int n;
    cin >> n;
    while (n--)
    {
        int opt, x;
        scanf("%d%d", &opt, &x);
        switch (opt)
        {
```

```
        case 1:insert(node(x)); break;
        case 2:del(x); break;
        case 3:printf("%d\n", Rank(node(x))); break;
        case 4:printf("%d\n", getkth(x).val); break;
        case 5:printf("%d\n", getpre(node(x)).val); break;
        case 6:printf("%d\n", getnxt(node(x)).val); break;
        default:
            break;
        }
    }
    return 0;
}
```