先来看一个问题

二叉搜索树简介

二叉搜索树是一种二叉树的树形数据结构，其定义如下：

1. 空树是二叉搜索树。
2. 若二叉搜索树的左子树不为空，则其左子树上所有点的附加权值均小于其根节点的值。
3. 若二叉搜索树的右子树不为空，则其右子树上所有点的附加权值均大于其根节点的值。
4. 二叉搜索树的左右子树均为二叉搜索树。

基本操作

数组定义

lc[x]	rc[x]	val[x]	cnt[x]	siz[x]
x节点左孩子	x节点右孩子	x节点存储的值	存储值的数量	x子树的数量总数

遍历二叉搜索树

由二叉搜索树的递归定义可得，二叉搜索树的中序遍历权值的序列为非降的序列。时间复杂度为 $O(n)$ 。

遍历一棵二叉搜索树的代码如下：

```
void print(int o)  //遍历以 o 为根节点的二叉搜索树
{
  if (!o) return;          //遇到空树，返回
  print(lc[o]);            //递归遍历左子树
  printf("%d\n", val[o]);  //输出根节点信息
  print(rc[o]);            //递归遍历右子树
}
```

查找最小/最大值

由二叉搜索树的性质可得，二叉搜索树上的最小值为二叉搜索树左链的顶点，最大值为二叉搜索树右链的顶点。时间复杂度为 $O(h)$ 。

```
int findmin(int o) {
  if (!lc[o]) return val[o];
  return findmin(lc[o]);  //一直向左儿子跳
}
int findmax(int o) {
  if (!rc[o]) return val[o];
  return findmax(rc[o]);  //一直向右儿子跳
}
```

插入一个元素

定义 `insert(o,v)` 为在以 o 为根节点的二叉搜索树中插入一个值为 v 的新节点。

分类讨论如下：

若 o 为空，直接返回一个值为 v 的新节点。

若 o 的权值大于 v ，在 o 的左子树中插入权值为 v 的节点。

若 o 的权值等于 v ，该节点的附加域该值出现的次数自增 1 。

若 o 的权值小于 v ，在 o 的右子树中插入权值为 v 的节点。

时间复杂度为 $O(h)$ 。

```
void insert(int &o, int v) {
  if (!o) {
    val[o = ++sum] = v;
    cnt[o] = siz[o] = 1;
    return;
  }
  siz[o]++;
  if (val[o] > v) insert(lc[o], v);
  if (val[o] == v) {
    cnt[o]++;
    return;
  }
  if (val[o] < v) insert(rc[o], v);
}
```

删除一个元素

定义 `delete(o,v)` 为在以 o 为根节点的二叉搜索树中删除一个值为 v 的节点。

先在二叉搜索树中找到权值为 v 的节点，分类讨论如下：

若该节点的附加 cnt 为 1 ：

若 o 为叶子节点，直接删除该节点即可。

若 o 为链节点，即只有一个儿子的节点，返回这个儿子。

若 o 有两个非空子节点，一般是用它左子树的最小值代替它，然后将它删除。

时间复杂度 $O(h)$ 。

```
int deletemin(int o) {
  if (!lc[o])
    int ret = val[o], o = rc[o], return ret;
  else
    return deletemin(lc[o]);
}
void delete (int& o, int v) {
  siz[o]--;
  if (val[o] == v) {
    if (lc[o] && rc[o])
      o = deletemin(rc[o]);
    else
      o = lc[o] + rc[o];
    return;
  }
```

```
    if (val[o] > v) delete (lc[o], v);
    if (val[o] < v) delete (rc[o], v);
}
```

求元素的排名

排名定义为将数组元素排序后第一个相同元素之前的数的个数 $+1$ 。

维护每个根节点的子树大小 siz 。查找一个元素的排名，首先从根节点跳到这个元素，若向右跳，答案加上左儿子节点个数加当前节点重复的数个数，最后答案加上终点的左儿子子树大小 $+1$ 。

时间复杂度 $O(h)$ 。

```
int queryrnk(int o, int v) {
    if (val[o] == v) return siz[lc[o]] + 1;
    if (val[o] > v) return queryrnk(lc[o], v);
    if (val[o] < v) return queryrnk(rc[o], v) + siz[lc[o]] + cnt[o];
}
```

查找排名为 k 的元素

在一棵子树中，根节点的排名取决于其左子树的大小。

若其左子树的大小大于等于 k ，则该元素在左子树中；

若其左子树的大小在区间 $[k-1, k+cnt-1]$ 中，则该元素为子树的根节点；

若其左子树的大小小于 $k+cnt-1$ ，则该元素在右子树中。

时间复杂度 $O(h)$ 。

```
int querykth(int o, int k) {
    if (siz[lc[o]] >= k) return querykth(lc[o], k);
    if (siz[lc[o]] + cnt[o] < k)
        return querykth(rc[o], k - siz[lc[o]] - cnt[o]);
    return o;
}
```

图例说明 写的很详细的博客

注意点

二叉搜索树上的基本操作所花费的时间与这棵树的高度成正比。对于一个有 n 个结点的二叉搜索树中，这些操作的最优时间复杂度为 $O(\log_2 n)$ ，最坏为 $O(n)$ 。随机构造这样一棵二叉搜索树的期望高度为 $O(\log_2 n)$ 。

Treap

treap 是一种弱平衡的二叉搜索树。treap 这个单词是 tree 和 heap 的组合，表明 treap 是一种由树和堆组合形成的数据结构。treap 的每个结点上要额外储存一个值 $priority$ 。treap 除了要满足二叉搜索树的性质之外，还需满足父节点的 $priority$ 大于等于两个儿子的 $priority$ 。而 $priority$ 是每个结点建立时随机生成的，因此 treap 是期望平衡的。

treap 分为旋转式和无旋式两种。两种 treap 都易于编写，但无旋式 treap 的操作方式使得它天生支持维护序列、可持久化等特性。后文以重新实现 `set<int>`（不可重集合）为例，介绍无旋式 treap。

旋转 treap

旋转 treap 在做普通平衡树题的时候，是所有平衡树中常数较小的

维护平衡的方式为旋转。性质与普通二叉搜索树类似

因为普通的二叉搜索树会被递增或递减的数据卡，用 treap 对每个节点定义一个权值，由 rand 得到，从而防止特殊数据卡。

每次删除/插入时通过 rand 值决定要不要旋转即可，其他操作与二叉搜索树类似

[写的很详细的博客]

以下是 bzoj 普通平衡树模板代码

```
#include <algorithm>
#include <cstdio>
#include <iostream>

#define maxn 100005
#define INF (1 << 30)

int n;

struct treap {
  int l[maxn], r[maxn], val[maxn], rnd[maxn], size[maxn], w[maxn];
  int sz, ans, rt;
  inline void pushup(int x) { size[x] = size[l[x]] + size[r[x]] + w[x]; }
  void lrotate(int &k) {
    int t = r[k];
    r[k] = l[t];
    l[t] = k;
    size[t] = size[k];
    pushup(k);
    k = t;
  }
  void rrotate(int &A) {
    int B = l[A];
    l[A] = r[B];
    r[B] = A;
    size[B] = size[A];
    pushup(A);
    A = B;
  }
  void insert(int &k, int x) {
    if (!k) {
      sz++;
      k = sz;
      size[k] = 1;
      w[k] = 1;
      val[k] = x;
      rnd[k] = rand();
      return;
    }
    size[k]++;
```

```
    if (val[k] == x) {
      w[k]++;
    } else if (val[k] < x) {
      insert(r[k], x);
      if (rnd[r[k]] < rnd[k]) lrotate(k);
    } else {
      insert(l[k], x);
      if (rnd[l[k]] < rnd[k]) rrotate(k);
    }
  }
//小根堆treap
  void del(int &k, int x) {
    if (!k) return;
    if (val[k] == x) {
      if (w[k] > 1) {
        w[k]--;
        size[k]--;
        return;
      }
      if (l[k] == 0 || r[k] == 0)
        k = l[k] + r[k];
      else if (rnd[l[k]] < rnd[r[k]]) {
        rrotate(k);
        del(k, x);
      } else {
        lrotate(k);
        del(k, x);
      }
    } else if (val[k] < x) {
      size[k]--;
      del(r[k], x);
    } else {
      size[k]--;
      del(l[k], x);
    }
  }

  int queryrank(int k, int x) {
    if (!k) return 0;
    if (val[k] == x)
      return size[l[k]] + 1;
    else if (x > val[k]) {
      return size[l[k]] + w[k] + queryrank(r[k], x);
    } else
      return queryrank(l[k], x);
  }

  int querynum(int k, int x) {
    if (!k) return 0;
    if (x <= size[l[k]])
      return querynum(l[k], x);
    else if (x > size[l[k]] + w[k])
      return querynum(r[k], x - size[l[k]] - w[k]);
    else
      return val[k];
  }

  void querypre(int k, int x) {
```

```
    if (!k) return;
    if (val[k] < x)
      ans = k, querypre(r[k], x);
    else
      querypre(l[k], x);
  }

  void querysub(int k, int x) {
    if (!k) return;
    if (val[k] > x)
      ans = k, querysub(l[k], x);
    else
      querysub(r[k], x);
  }
} T;

int main() {
  srand(123);
  scanf("%d", &n);
  int opt, x;
  for (int i = 1; i <= n; i++) {
    scanf("%d%d", &opt, &x);
    if (opt == 1)
      T.insert(T.rt, x);
    else if (opt == 2)
      T.del(T.rt, x);
    else if (opt == 3) {
      printf("%d\n", T.queryrank(T.rt, x));
    } else if (opt == 4) {
      printf("%d\n", T.querynum(T.rt, x));
    } else if (opt == 5) {
      T.ans = 0;
      T.querypre(T.rt, x);
      printf("%d\n", T.val[T.ans]);
    } else if (opt == 6) {
      T.ans = 0;
      T.querysub(T.rt, x);
      printf("%d\n", T.val[T.ans]);
    }
  }
  return 0;
}
```