P5076 搜索二叉树（bst.cpp）

【题目描述】

您需要写一种数据结构，来维护一些数（都是 10^9 以内的整数）的集合，最开始时集合是空的。现在需要提供以下操作，操作次数 M 不超过 10^4。

操作分为以下 5 种类型：

1．查询 X 数的排名（排名定义为比当前数小的数的个数+1，若有多个相同的数，应输出最小的排名）。

2．查询排名为 X 的数。

3．求 X 的前驱（前驱定义为小于 X，且最大的数）。若未找到则输出-2147483647。

4．求 X 的后继（后继定义为大于 X，且最小的数）。若未找到则输出 2147483647。

5．插入一个数 X。

【输入格式】

输入一个正整数 M，表示操作次数。

接下来 M 行，每行两个整数 T（$1 \leqslant T \leqslant 5$）和 X（$0 \leqslant X \leqslant 10^9$）：其中 T 表示 5 种操作类型之一，X 是操作对象。

【输出格式】

输出操作 1～4 的结果。

【输入样例】

```
7
5 1
5 3
5 5
1 3
2 2
3 3
4 3
```

【输出样例】

```
2
3
1
5
```

二叉搜索树 简介

我们平常所说的"平衡树"（伸展树 Splay，替罪羊树等）实际上都属于"平衡二叉搜索树"，也就是既满足"平衡树"又满足"二叉搜索树"。二叉搜索树的效率比平衡二叉搜索树的效率低很多，但是在学习平衡二叉搜索树之前也要理解二叉搜索树的实现原理，此文就是来帮助理解的。

BST（Binary Search Tree），二叉搜索树，又叫二叉排序树。

BST 是一棵空树或具有以下几种性质的树：

1. 若左子树不空，则左子树上所有结点的值均小于它的根结点的值
2. 若右子树不空，则右子树上所有结点的值均大于它的根结点的值
3. 左、右子树也分别为二叉排序树
4. 没有权值相等的结点

看到第 4 条，我们会有一个疑问，在数据中遇到多个相等的数该怎么办呢，显然我们可以多加一个计数器，记录当前这个值出现的次数。

那么我们的每一个节点都记录了以下几个信息：

1. 当前节点的权值，也就是序列里的数
2. 左孩子的下标和右孩子的下标，如果没有则为 0
3. 计数器，记录当前这个值出现的次数
4. 子树大小和自己的大小的和

至于为什么要有 4，我们放到后面讲。

节点是这样的:

```
struct node
{
    int val,ls,rs,cnt,siz;
}tree[500010];
```

其中 val 是权值，ls/rs 是左/右孩子的下标，cnt 是当前的权值出现了几次，siz 是子树大小和自己的大小的和。

以下操作函数均以递归方式呈现：

插入操作:

x 是当前节点的下标，v 是要插入的值。要在树上插入一个 v 的值，就要找到一个合适 v 的位置，如果本身树的节点内有代表 v 的值的节点，就把该节点的计数器加 1，否则一直向下寻找，直到找到叶子节点，这个时候就可以从这个叶子节点连出一个儿子，代表 v 的节点。具体向下寻找该走左儿子还是右儿子是根据二叉搜索树的性质来的。

```
void add(int x,int v)
{
    tree[x].siz++;        //如果查到这个节点，说明这个节点的子树里面肯定是有 v 的，所以 siz++
    if(tree[x].val==v)   //如果恰好有重复的数，就把 cnt++，退出即可，因为我们要满足第四条性质
    {
        tree[x].cnt++;
        return ;
    }
    if(tree[x].val>v)   //如果 v<tree[x].val，说明 v 实在 x 的左子树里
    {
        if(tree[x].ls!=0)
            add(tree[x].ls,v);    //如果 x 有左子树，就去 x 的左子树
        else                      //如果不是，v 就是 x 的左子树的权值
        {
            cont++;               //cont 是目前 BST 一共有几个节点
            tree[cont].val=v;
            tree[cont].siz=tree[cont].cnt=1;
            tree[x].ls=cont;
        }
    }
    else    //右子树同理
    {
        if(tree[x].rs!=0)
          add(tree[x].rs,v);
        else
        {
            cont++;
            tree[cont].val=v;
            tree[cont].siz=tree[cont].cnt=1;
            tree[x].rs=cont;
        }
    }
}
```

找前驱操作：

x 是当前的节点的下标，val 是要找前驱的值，ans 是目前找到的比 val 小的数的最大值。

找前驱的方法也是不断的在树上向下爬找具体节点，具体爬的方法可以参考代码注释部分。

```
int queryfr(int x,int val,int ans)
{
    if (tree[x].val>=val)        //如果当前值大于 val，就说明查的数大了，所以要往左子树找
    {
        if (tree[x].ls==0)       //如果没有左子树就直接返回找到的 ans
            return ans;
        else                     //如果不是的话，去查左子树
            return queryfr(tree[x].ls,val,ans);
    }
    else       //如果当前值小于 val，就说明我们找比 val 小的了
    {
    //如果没有右孩子，就返回 tree[x].val，因为走到这一步时，我们后找到的一定比先找到的大(参考第二条性质)
        if (tree[x].rs==0)
            return (tree[x].val<val) ? tree[x].val : ans
       //如果有右孩子，我们还要找这个节点的右子树，因为万一右子树有比当前节点还大并且小于要找的 val 的话，ans 需要更新
        if (tree[x].cnt!=0)          //如果当前节数的个数不为 0，ans 就可以更新为 tree[x].val
            return queryfr(tree[x].rs,val,tree[x].val);
        else                         //反之 ans 不需要更新
            return queryfr(tree[x].rs,val,ans);
    }
}
```

找后继操作：

与找前驱同理，只不过反过来了，在这里就不多赘述了。

```
int queryne(int x, int val, int ans)
{
    if (tree[x].val<=val)
    {
        if (tree[x].rs==0)
            return ans;
        else
            return queryne(tree[x].rs,val,ans);
    }
    else
    {
        if (tree[x].ls==0)
            return (tree[x].val>val)? tree[x].val : ans;
        if (tree[x].cnt!=0)
            return queryne(tree[x].ls,val,tree[x].val);
        else
            return queryne(tree[x].ls,val,ans);
    }
}
```

按值找排名操作：

这里我们就要用到 siz 了，排名就是比这个值要小的数的个数再+1，所以我们按值找排名，就可以看做找比这个值小的数的个数，最后加上 1 即可。

```
int queryval(int x,int val)
{
    if(x==0) return 0;     //没有排名
    if(val==tree[x].val) return tree[tree[x].ls].siz;
                    //如果当前节点值=val，则我们加上现在比 val 小的数的个数，也就是它左子树的大小
    if(val<tree[x].val) return queryval(tree[x].ls,val);
                //如果当前节点值比 val 大了，我们就去它的左子树找 val，因为左子树的节点值一定是小的
    return queryval(tree[x].rs,val)+tree[tree[x].ls].siz+tree[x].cnt;
        //如果当前节点值比 val 小了，我们就去它的右子树找 val，同时加上左子树的大小和这个节点的值出现次数
                        //因为这个节点的值小于 val，这个节点的左子树的各个节点的值一定也小于 val
}
//注:这里最终返回的是排名-1，也就是比 val 小的数的个数，在输出的时候记得+1
```

按排名找值操作：

因为性质 1 和性质 2，我们发现排名为 n 的数在 BST 上是第 n 靠左的数。或者说排名为 n 的数的节点在 BST 中，它的左子树的 siz 与它的各个祖先的左子树的 siz 相加恰好等于 n（这里相加是要减去重复部分）。

所以问题又转化成上一段 **或者说** 的后面的部分

rk 是要找的排名

```
int queryrk(int x,int rk)
{
    if(x==0) return INF;
    if(tree[tree[x].ls].siz>=rk)            //如果左子树大小>=rk 了，就说明答案在左子树里
        return queryrk(tree[x].ls,rk);      //查左子树
    if(tree[tree[x].ls].siz+tree[x].cnt>=rk)
                                //如果左子树大小加上当前的数的多少恰好>=k，说明我们找到答案了
        return tree[x].val;     //直接返回权值
    return queryrk(tree[x].rs,rk-tree[tree[x].ls].siz-tree[x].cnt);
                                //否则就查右子树，同时减去当前节点的次数与左子树的大小
}
```

删除操作：

具体就是利用二叉搜索树的性质在树上向下爬找到具体节点，把计数器-1。与上文同理就不放代码了。

BST 的弊端：时间复杂度最坏为 $O(n^2)$。

看完上文，你一定理解了二叉搜索树的具体实现原理和方法，但是如果构建出的一棵 BST 是个链的话，时间复杂度就会退化到 $O(n^2)$级别，因为如果每次都查找链最低端的叶子节点的复杂度是 O(n)的。而去保持这个树是个平衡树，就可以防止出现这个错误的复杂度，这个时候就有了平常所说的平衡树（这是 NOI 学习内容）。

【递归实现 BST 的完整版代码】

```
#include<bits/stdc++.h>
using namespace std;

const int INF=2147483647;
int cont,n,opt,xx;
struct node
{
    int val,siz,cnt,ls,rs;
}tree[1000010];

inline void add(int x,int v)
{
    tree[x].siz++;
    if(tree[x].val==v)
    {
        tree[x].cnt++;
        return ;
    }
    if(tree[x].val>v)
    {
        if(tree[x].ls!=0)
          add(tree[x].ls,v);
        else
        {
            cont++;
            tree[cont].val=v;
            tree[cont].siz=tree[cont].cnt=1;
            tree[x].ls=cont;
        }
    }
    else
    {
        if(tree[x].rs!=0)
          add(tree[x].rs,v);
        else
        {
            cont++;
            tree[cont].val=v;
            tree[cont].siz=tree[cont].cnt=1;
            tree[x].rs=cont;
        }
    }
}
```

```
int queryfr(int x, int val, int ans)
{
    if(tree[x].val>=val)
    {
        if(tree[x].ls==0)
            return ans;
        else
            return queryfr(tree[x].ls,val,ans);
    }
    else
    {
        if(tree[x].rs==0)
            return tree[x].val;
        return queryfr(tree[x].rs,val,tree[x].val);
    }
}

int queryne(int x, int val, int ans)
{
    if(tree[x].val<=val)
    {
        if(tree[x].rs==0)
            return ans;
        else
            return queryne(tree[x].rs,val,ans);
    }
    else
    {
        if(tree[x].ls==0)
            return tree[x].val;
        return queryne(tree[x].ls,val,tree[x].val);
    }
}

int queryrk(int x,int rk)
{
    if(x==0) return INF;
    if(tree[tree[x].ls].siz>=rk)
        return queryrk(tree[x].ls,rk);
    if(tree[tree[x].ls].siz+tree[x].cnt>=rk)
        return tree[x].val;
    return queryrk(tree[x].rs,rk-tree[tree[x].ls].siz-tree[x].cnt);
}
```

```
int queryval(int x,int val)
{
    if(x==0) return 0;
    if(val==tree[x].val) return tree[tree[x].ls].siz;
    if(val<tree[x].val) return queryval(tree[x].ls,val);
    return queryval(tree[x].rs,val)+tree[tree[x].ls].siz+tree[x].cnt;
}

int main()
{
    scanf("%d",&n);
    while(n--)
    {
        scanf("%d %d",&opt,&xx);
        if(opt==1) printf("%d\n",queryval(1,xx)+1);
        else if(opt==2) printf("%d\n",queryrk(1,xx));
        else if(opt==3) printf("%d\n",queryfr(1,xx,-INF));
        else if(opt==4) printf("%d\n",queryne(1,xx,INF));
        else
        {
            if(cont==0)
            {
                cont++;
                tree[cont].cnt=tree[cont].siz=1;
                tree[cont].val=xx;
            }
            else add(1,xx);
        }
    }
    return 0;
}
```

【非递归实现 BST 的完整版代码】

```
#include<bits/stdc++.h>
using namespace std;
```

```
#define pb push_back
#define ls tree[x].son[0]
#define rs tree[x].son[1]

const int N = 10010;
const int INF = 2147483647;
int n,root,tot,opt,x;

struct Node
{
    int val, siz, cnt, son[2];
}tree[N];

void add(int v)
{
    if(!tot)
    {
        root = ++tot;
        tree[tot].cnt = tree[tot].siz = 1;
        tree[tot].son[0] = tree[tot].son[1] = 0;
        tree[tot].val = v;
        return ;
    }
    int x = root, last = 0;
    do
    {
        ++tree[x].siz;
        if(tree[x].val == v)
        {
            ++tree[x].cnt;
            break;
        }
        last = x;
        x = tree[last].son[v > tree[last].val];
        if(!x)
        {
            tree[last].son[v > tree[last].val] = ++tot;
            tree[tot].son[0] = tree[tot].son[1] = 0;
            tree[tot].val = v;
            tree[tot].cnt = tree[tot].siz = 1;
            break;
        }
    }while(true);
}
```

```
int queryfr(int val)
{
    int x = root, ans = -INF;
    do
    {
        if(x == 0) return ans;
        if(tree[x].val >= val)
        {
            if(ls == 0) return ans;
            x = ls;
        }
        else
        {
            if(rs == 0) return tree[x].val;
            ans = tree[x].val;
            x = rs;
        }
    }while(true);
}

int queryne(int v)
{
    int x = root, ans = INF;
    do
    {
        if(x == 0) return ans;
        if(tree[x].val <= v)
        {
            if(rs == 0) return ans;
            x = rs;
        }
        else
        {
            if(ls == 0) return tree[x].val;
            ans = tree[x].val;
            x = ls;
        }
    }while(true);
}
```

```
int queryrk(int rk)
{
    int x = root;
    do
    {
        if(x == 0) return INF;
        if(tree[ls].siz >= rk) x = ls;
        else if(tree[ls].siz + tree[x].cnt >= rk) return tree[x].val;
        else rk -= tree[ls].siz + tree[x].cnt, x = rs;
    }while(true);
}

int queryval(int v)
{
    int x = root, ans = 0;
    do
    {
        if(x == 0) return ans;
        if(tree[x].val == v) return ans + tree[ls].siz;
        else if(tree[x].val > v) x = ls;
        else ans += tree[ls].siz + tree[x].cnt, x = rs;
    }while(true);
}

int main()
{
    scanf("%d",&n);
    while(n--)
    {
        scanf("%d %d",&opt,&x);
        if(opt == 1) printf("%d\n", queryval(x) + 1);
        if(opt == 2) printf("%d\n", queryrk(x));
        if(opt == 3) printf("%d\n", queryfr(x));
        if(opt == 4) printf("%d\n", queryne(x));
        if(opt == 5) add(x);
    }
    return 0;
}
```