8.3 用树状数组统计子树权和的实验范例

有时，从现实生活抽象出的树模型中节点被赋予了一个权值，而解题目标是动态统计子树的权和。8.1 给出了通过树的后序遍历求解的方法，时间花费为 O(n)。如果节点的权值发生变化，则“牵一发而动全身”，相关子树的权和随之发生了变化，就需要再次通过后序遍历统计子树的权和。显然，这种“蛮力搜索”的方法并不适合。

树的后序遍历实质上是按照自下而上的顺序将非线性结构的树转化为一个线性序列，每棵子树对应这个线性序列的一个连续的子区间。这就提醒了我们，是否可以用线性数据结构的方法解决动态统计子树权和的问题呢？有的，这就是树状数组。

树状数组（Binary Indexed Tree (B.I.T)，Fenwick Tree），也被称为二叉索引树，是一个查询和修改复杂度都为 $\log_2 n$ 的数据结构，主要用于查询任意两位之间的所有元素之和。定义相关数据结构如下：

设数组 $a[\ ]$，元素个数为 n，存储在 $a[1]$到 $a[n]$中；

子区间的权和数组为 sum，其中数组 a 中从 i 到 j 区间内的权和 $sum[i,j]=\sum_{k=i}^{j} a[k]$；

前缀的权和数组为 s，其中数组 a 中长度为 i 的前缀的权和 $s[i]=\sum_{k=1}^{i} a[k]$；显然，$sum[i,j]=s[j]-s[i-1]$；

$lowbit(k)$为整数 k 的二进制表示中右边第一个 1 所代表的数，在程序实现时，$lowbit(k)=k\&(-k)$；例如，12 的二进制是 1100，右边第一个 1 所代表的数字是 4；$-k$ 则是将 k 按位取反，然后末尾加 1；$k\&(-k)$则是 k 与$-k$ 按位与；例如，1100 按位取反，然后末尾加 1 的结果是 0100，两者按位与的结果 100，所代表的数是 4。

树状数组 c，其中 $c[k]$存储从 $a[k]$开始向前数 $lowbit(k)$个元素之和，即 $c[k]=\sum_{i=k-lowbit(k)+1}^{k} a[i]$，显然，树状数组采用了分块的思想（如图 8.3-1）。

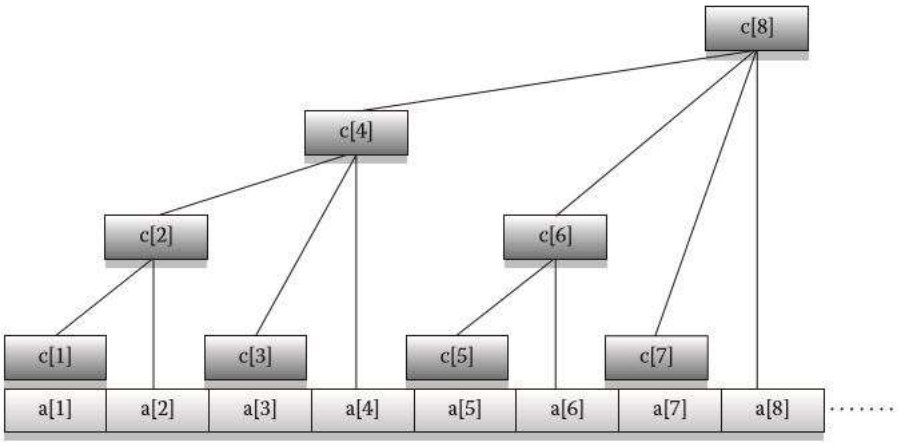


图 8.3-1

更改和查询数组 a 的元素直接在树状数组 c 中进行。例如，如果要更改元素 $a[2]$，影响到数组 c 中的元素有 $c[2]$、$c[4]$和 $c[8]$，我们只需一层一层往上修改就可以了，而这个过程的时间复杂度是 $O(\log_2 n)$。又例如，查找 $s[7]$，7 的二进制表示为 0111，右边的第一个 1 出现在第 0 位上，也就是说要从 $a[7]$开始数 1 个元素（$a[7]$），即 $c[7]$；然后将这个 1 舍掉，得到 6，二进制表示为 0110，右边第一个 1 出现在第 1 位上，也就是说要从 $a[6]$开始向前数 2 个元素($a[6]$，$a[5]$)，即 $c[6]$；最后舍掉用过的 1，得到 4，二进制表示为 0100,右边第一个 1 出现在第 2 位上，也就是说要从 $a[4]$开始向前数 4 个元素($a[4]$，$a[3]$，$a[2]$，$a[1]$)，即 $c[4]$，显然，$s[7]=c[7]+c[6]+c[4]$。

$a[x]$增加 k 后，树状数组 c 的调整过程如下：

```
for(i=x; i<cnt; i+=lowbit(i)) c[i]+=k;
```

$s[x]=\sum_{k=1}^{x} a[k]$ 的计算过程如下：

```
s[x]=0;
for(i=x; i>0; i-=lowbit(i)) s[x]+=c[i];
```

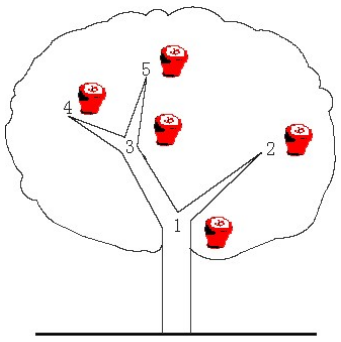
显然，计算 $s[x]$的复杂度是 $\log_2 n$，计算 $sum[x,y]=\sum_{i=x}^{y} a[i]=s[y]-s[x-1]$仅需花费时间 $2\log_2 n$。动态维护树状数组以及求和过程的复杂度通过数组 c 的定义都降到了 $\log_2 n$。

实际上，树状数组本来是用于一维序列的“动态统计”的，即作为统计对象的数据需要被频繁更新；被推广至统计子树权和问题，说明应用线性数据结构可以用于解决非线性问题。

【8.3.1 Apple Tree】

在 Kaka 的房子外面有一棵苹果树。每年秋天，树上要结出很多苹果。Kaka 很喜欢吃苹果，所以他一直精心培育这棵苹果树。

这棵苹果树有 N 个分岔点，连接着各个分枝。Kaka 将这些分岔点从 1 到 N 进行编号，根被编号为 1。苹果在分岔点生长，在一个分岔点不会有两只苹果。因为 Kaka 要研究苹果树的产量，所以他想知道在一棵子树上会有多少苹果（图 8.3-2）。



-

图 8.3-2

问题在于，在一个空的分岔点上，过些时间可能会长出新苹果；而 Kaka 也可能会从树上摘苹果用来作为他的甜点心。

输入

第一行给出整数 N（N≤100,000），是树的分岔点的数目。

后面的 $N-1$ 行，每行给出两个整数 u 和 v，表示分岔点 u 和 v 是由树枝连接的。

下一行给出整数 M（M≤100,000）。

后面的 M 行，每行包含了一条信息，或者是

“C x” 表示在分岔点 x 存在的苹果已经发生变化，即，如果在分岔点 x 原来有苹果，则苹果被 Kaka 摘了吃了；或者是新苹果在空的分岔点长出来了；或者是

“Q x” 表示查询在分岔点 x 上面的苹果的数量，包括分岔点 x 的苹果（如果有苹果的话）。

开始的时候，树上是长满苹果的。

输出

对于每个查询，在一行中输出相应的回答。

样例输入	样例输出
3 1 2 1 3 3 Q 1 C 2 Q 1	3 2

试题来源：POJ Monthly

在线测试：POJ 3321

试题解析

苹果树就是一棵树，分岔点是树节点，分岔点上的苹果数为节点权值。指令“Q x”是计算以节点x为根的子树的权值和；指令“C x”表示节点节点x的权值发生变化，由1变成0（分岔点x上的苹果被摘吃），或者由0变成1（空的分岔点x长出新苹果）。为了方便快捷地统计子树的权值和，我们引入了树状数组c[ ]，以访问时间为顺序，通过在后序遍历的过程中给节点加盖时间戳的办法，将苹果树的非线性结构转化为线性序列。设

$d[u]$—节点 u 的初访时间；

$f[u]$—节点 u 的结束时间，即访问了以 u 为根的子树后回溯至 u 的时间。

显然，区间[$d[u]$，$f[u]$]反映了以 u 为根的子树结构。例如，图 8.3-3 给出一棵二叉树

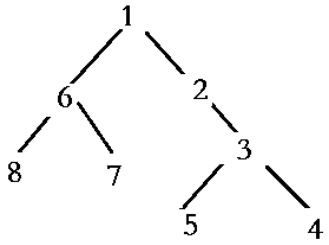


图 8.3-3

后序遍历二叉树，*d*[*u*]和*f*[*u*]如下表所示：

节点访问顺序	1	6	8	7	6	2	3	5	4	3	2	1
节点区间[*d*[*u*],*f*[*u*]]	[1，]	[1，]	[1，1]	[2，2]	[1，3]	[4，]	[4，]	[4，4]	[5，5]	[4，6]	[4，7]	[1，8]
说明	初访节点 1	初访节点 6	终访节点 8	终访节点 7	终访节点 6	初访节点 2	初访节点 3	终访节点 5	终访节点 4	终访节点 3	终访节点 2	终访节点 1

按照后序遍历的节点顺序，其*f*[ ]值正好递增（*f*[8]=1，*f*[7]=2，*f*[6]=3，*f*[5]=4，*f*[4]=5，*f*[3]=6，*f*[2]=7），*f*[1]=8）。若[*d*[*v*]，*f*[*v*]]是[*d*[*u*]，*f*[*u*]]的子区间，则以 *u* 为根的子树包含了以 *v* 为根的子树。因此可用*f*[*u*]标志 *c*[ ]的指针。计算[*d*[*u*]，*f*[*u*]]的方法十分简单：

```
void DFS(int u);
{
  d[u]=time;                    //初次访问u的时间设为区间左指针
  依次对u引出的每条出边的另一端点v进行DFS(v);
  f[u]=time++;                  //遍访了u的所有后代后的时间为区间右指针,访问时间+1
}
```

若命令为‘C *x*’，即节点 *x* 的权值 *a*[*x*]发生变化（0 变 1 或者 1 变 0），则 *a*[*f*[*x*]]取反（（*a*[*f*[*x*]]=(*a*[*f*[*x*]]+1)%2），从 *c*[*f*[*x*]]出发向上调整树状数组 *c*[ ]。调整的方法如下：

```
void change(int x)
{
  int i;
  if(a[x]) for(i=x; i<cnt; i+=lowbit(i)) c[i]++;
    else for(i=x; i<cnt; i+=lowbit(i)) c[i]--;
}
```

由于最初时树上长满苹果，因此可按照如下方法构造最初的树状数组*c*[ ]:

```
for(i=1; i<=n; i++){
    a[i]设为1；
    change(i);
 }
```

若命令为‘Q *x*’，则以节点*x*为根的子树的权值和为*sum*(*f*[*x*])-*sum*(*d*[*x*]-1)。其中*sum*(*x*)为前*x*个访问时间的前缀和。计算方法如下：

```
int sum(int x)
{
    int i, res=0;
    for (i=x; i>0; i-=lowbit(i))    res+=c[i];
    return res;
}
```

参考程序

```
#include<iostream>              //预编译命令
#include<cstring>
#define max 100002              //定义节点数的上限
using namespace std;
struct node1                    //边表为 edge，其中第 i 条边相连的节点为 edge[i].tail；连接的下条边的序号为 edge[i].next
{
    int next,tail;
}edge[max];
struct node2      //苹果树为 apple，以节点 i 为根的子树在后序序列中的区间为[apple[i].l，apple[i].r]
{
    int r,l;
}apple[max];
int s[max],cnt,c[max],a[max];   //后序遍历中第 i 个节点的权值为 a[i]；后序遍历序号为 cnt；树状数组为 c；节点 i 相连的第 i 条边的序号为 s[i]
void DFS(int u)         //从节点 u 出发，计算每个节点为根的子树区间[apple[].l，apple[].r]
{
    int i;
    apple[u].l=cnt;
    for(i=s[u];i!=-1;i=edge[i].next)
        DFS(edge[i].tail);
    apple[u].r=cnt++;
}
inline int lowbit(int x)              //计算二进制数 x 右方的第 1 位 1 对应的权
{
    return x&(-x);
}
```

```
void change(int x)                  //从 a[x]出发，调整树状数组
{
    int i;
    if(a[x])                        //若 a 序列的第 x 个元素非零，则树状数组的相关元素值+1；否则树状数组的相关元素值-1
        for(i=x;i<cnt;i+=lowbit(i))
            c[i]++;
    else                            //若权值和为 x 的子树根上的苹果被吃掉，则其通往根的路径上每棵子树的权值和-1
        for(i=x;i<cnt;i+=lowbit(i))
            c[i]--;
}

int sum(int x)                      //计算 sum_{k=1}^{x} a[k]

{
    int i,res=0;
    for(i=x;i>0;i-=lowbit(i))
        res+=c[i];
    return res;
}
int main()                                          //主函数
{
    int i,n,m,t1,t2,t;
    char str[3];
    scanf("%d",&n);                                 //读树的节点数
    memset(s,-1,sizeof(s[0])*(n+1));                //s 的每个元素初始化为-1
    memset(c,0,sizeof(c[0])*(n+1));                 //c 的每个元素初始化为 0
    memset(apple,0,sizeof(apple[0])*(n+1));         // apple 的每个元素初始化为 0
    for(i=0;i<n-1;i++){
        scanf("%d%d",&t1,&t2);                      //读第 i 条边（t1, t2）
        edge[i].tail=t2;                //第 i 条边连接 t2，其后继指针指向 t1 连接的上一条边
        edge[i].next=s[t1];
        s[t1]=i;                                    //设节点 t1 连接的边序号 i
    }
    cnt=1;
    DFS(1);                 //从节点 1 出发进行 DFS，计算每个节点的后序值，节点权值设 1
```

```
    scanf("%d",&m);       //读信息数
    for(i=1;i<=n;i++){    //构造长满苹果的树上对应的树状数组 c
        a[i]=1;           //设 a[i]为 1，由此出发调整树状数组
        change(i);
    }
    while(m--){
        scanf("%s%d",&str,&t);    //读命令标志 str 和节点序号 t
        if(str[0]=='Q')           //输出节点 t 上的苹果数
            printf("%d\n",sum(apple[t].r)-sum(apple[t].l-1));
        else{                     //计算节点 t 上苹果的变化情况
            a[apple[t].r]=(a[apple[t].r]+1)%2;  //节点 t 上的苹果数由 1 变成 0 或由 0 变成 1
            change(apple[t].r);
        }
    }
    return 0;
}
```

树状数组不仅可以统计子树权和，而且可以用于逆序对的计算。如果 $i>j$ && $a[i]<a[j]$，则 $a[i]$和 $a[j]$就为一对逆序对。

逆序对的计算，就是对序列中的每个数，找出排在其前面有多少个比自己大的数。显然，树状数组可以优化这种“需要遍历”的情况。方法如下：

首先，递增排序原序列。建立一个元素类型为结构体的数组 *a*[]，其中 *a*[*i*].*val* 为输入的数，*id* 为输入顺序。然后，按 *val* 域值为第一关键字，*id* 域值为第二关键字递增排序 *a*[]。如果没有逆序的话，递增序列 *a*[]中每个元素的下标 *i* 与 *id* 域值是相同；如果有逆序数，那么必然存在元素下标 *i* 与域值 *id* 不同的情况。所以利用树状数组的特性，我们可以方便的算出逆序数的个数。例如输入 4 个数：9，-1，18，5。设初始时的结构型数组 *a*[]为：

a[1].*val*=9，*a*[1].*id*=1；*a*[2].*val*=-1，*a*[2].*id*=2；*a*[3].*val*=18；*a*[3].*id*=3；*a*[4].*val*=5；*a*[4].*id*=4。

按 *val* 域值递增顺序给数组 *a* 排序，则数组 *a*[]为：

a[1].*val*=-1，*a*[1].*id*=2；*a*[2].*val*=5，*a*[2].*id*=4；*a*[3].*val*=9，*a*[3].*id*=1；*a*[4].*val*=18，*a*[4].*id*=3。

所以，*a*[]的域值 *id* 组成序列（简称域值 *id* 序列）2，4，1，3，该序列的逆序数是 3。而且，3 也是原序列 9，-1，18，5 的逆序数。

下面，我们利用树状数组的特性求域值 *id* 序列的逆序数。

依次插入域值 *id* 序列的元素 *x*（即原序列中第 *x* 个元素设访问标志 1），通过 *Modify*(*x*)调整树状数组：

```
void Modify(x)                //访问 x 位置，调整树状数组 c[]
{
    for(int i=x; i<=n; i+=lowbit(i)) c[i]+=1;
```

}

通过 *getsum*(*x*)查询区间[1, *x*]的和，即得出前面有多少个不大于它的数（包括自己）。

int *getsum* (int *x*)　　//计算和返回 $ans = \sum_{k=1}^{p} a[k]$

{

LL *ans*=0;

for(int *i*=*x*; *i*>0; *i*-=*lowbit*(*i*)) *ans*+=*c*[*i*];

return *ans*;

}

再用已插入数的个数减去 *getsum*(*x*)，就算出了前面有多少个数比他大了。由此得出计算序列 *a*[]的逆序对的个数 *sum* 的方法：

递增排序 *a*[1].*val*‥*a*[*n*].*val*，得出排序后的域值 *id* 序列 *a*[1].*id*‥*a*[*n*].*id*。

sum = 0;

for(*i*=1; *i*<=*n*; *i*++)

{

Modify(*a*[*i*].*id*);

sum+=(*i*-*getsum*(*a*[*i*].*id*));

}

【8.3.2 Japan】

日本计划迎接 ACM-ICPC 世界总决赛，在比赛场地必须修建许多条道路。日本是一个岛国，东海岸有 *N* 座城市，西海岸有 *M* 座城市（*M*≤1000，*N*≤1000），将建造 *K* 条高速公路。从北到南，每个海岸的城市编号为 1, 2, ...。每条高速公路都是直线，连接东海岸城市和西海岸城市。建设资金由 ACM 提供担保。其中很大一部分是由高速公路之间的交叉口数量决定的。两条高速公路最多在一个地方交叉。请您编写一个计算高速公路之间交叉口数量的程序。

输入

输入首先给出 *T*，表示测试用例的数量。每个测试用例首先给出三个数字，*N*，*M* 和 *K*。接下来的 *K* 行每行给出两个数字：高速公路连接的城市编号，其中，第一个是东海岸的城市编号，第二个是西海岸的城市编号。

输出

对于每个测试用例，标准输出一行：

Test case (用例编号): (交叉口的数量)

样例输入	样例输出
1 3 4 4 1 4 2 3	Test case 1: 5

3 2 3 1	

试题来源：ACM Southeastern Europe 2006

在线测试：POJ 3067，UVA 2926

试题解析

在日本岛东海岸和西海岸分别有 N 和 M 座城市，建造 K 条高速公路连接东、西海岸的城市，本题请您求交点个数。

因为连接东、西海岸城市的 K 条高速公路都是直线；所以东、西海岸城市用点表示，如果在东海岸的第 x 个城市与西海岸的第 y 个城市之间建造了一条高速公路，则在相应的第 x 个点与第 y 个点之间连一直线。那么，如果在第 x_1 个点与第 y_1 个点之间有一直线，在第 x_2 个点与第 y_2 个点之间有一直线，并且$(x_1- x_2)*(y_1- y_2)<0$，则对应的两条高速公路有交点。

以东海岸的N座城市的编号为第一关键字、以西海岸的M座城市的编号为第二关键字，对连接东、西海岸的 K 条高速公路进行排序。

设第 i 条高速公路的端点分别为 x_i 和 y_i。则对前 i-1 条高速公路的端点 x_k 和 y_k，$1≤k≤i$-1，$x_k≤x_i$，如果 $x_k<x_i$，$y_k>y_i$，则相应的高速公路和第 i 条高速公路相交。也就是说，在前 i-1 条边中，与第 i 条边相交的边的 y_k 值必然大于 y_i 的值，所以此时我们只需要求出在前 i-1 条边中有多少条边的 y_k 值在区间$[y_i+1, M]$中即可，也就是求 y_i 的逆序数。这样，就将问题转化成区间求和的问题，用树状数组解决。

参考程序

```
#include <iostream>
#include <cstring>
#include <cstdio>
#include <algorithm>
#define LL long long
using namespace std;
const int M=1003;
LL C[M];                          //树状数组 C[]
int n,m,k;                        //n 和 m 为东海岸与西海岸的城市数，k 为及高速公路数
struct Node                       //高速路序列的元素为结构体类型
{
  int x,y;                        //连接高速路两端的城市序号
}edge[1003*1002];                 //高速路序列
bool cmp(Node a,Node b)           //比较函数：按照 x 域为第一关键字、y 域为第二关键字定义高速公路 a 和 b 的大小关系
{
    if(a.x==b.x)return a.y<=b.y;
    else return a.x<b.x;
```

```
}
int lowbit(int a)                          //计算二进制数 a 右方的第 1 位 1 对应的权
{
    return a&(-a);
}
void Modify(int p,int c)                   //a[p]增加 c，调整树状数组 C[]
{
    for(int i=p;i<=m;i+=lowbit(i))
      C[i]+=c;
}

int getsum(int p)                          //计算和返回 ans = \sum_{k=1}^{p} a[k]

{
    LL ans=0;
    for(int i=p;i>0;i-=lowbit(i))
    {
       ans+=C[i];
    }
    return ans;
}
int main()
{
    int T=0;                               //测试用例编号初始化
    int cas;
    scanf("%d",&cas);                      //输入测试用例数
    while(cas--)                           //依次处理每个测试用例
    {
       scanf("%d%d%d",&n,&m,&k);     //输入东海岸与西海岸的城市数 n 和 m 以及高速公
路数 k
       for(int i=0;i<k;i++)                //输入每条高速公路连接的两个城市
         scanf("%d%d",&edge[i].x,&edge[i].y);
      memset(C,0,sizeof(C));               //树状数组初始化
      sort(edge,edge+k,cmp);               //递增排序 k 条高速公路
      LL ans=0;                            //高速公路间的交叉口数量初始化
      for(int i=0;i<k;i++)                 //使用树状数组统计高速公路间的交叉口数量
      {
```

```
            Modify(edge[i].y,1);                    //将前 i-1 条边中 y 域值在区间[第 i 条边的 y 域值,
M]中的高速路数计入 ans
            ans+=(getsum(m)-getsum(edge[i].y));
        }
        T++;                                        //计算和输出测试用例编号
        printf("Test case %d: %lld\n",T,ans);       //输出高速公路间的交叉口数量
    }
    return 0;
}
```