**[树状数组](http://kmplayer.javaeye.com/blog/562119)**

**文章分类:**[**C++编程**](http://www.javaeye.com/blogs/category/cpp)

**1,用途**   
树状数组是一种非常优雅的数据结构.当要频繁的对数组元素进行修改,同时又要频繁的查询数组内任一区间元素之和的时候,可以考虑使用树状数组.   
换句话说,树状数组最基本的应用:   
对于一个数组，如果有多次操作，每次的操作有两种：1、修改数组中某一元素的值，2、求和，求数组元素a[1]+a[2]+…a[num]的和。   
**2,复杂度**   
最直接的算法可以在O(1)时间内完成一次修改,但是需要O(n)时间来进行一次查询.而树状数组的修改和查询均可在O(log(n))的时间内完成.   
**3,生成**   
设a[1...N]为原数组,定义c[1...N]为对应的树状数组:   
c[i] = a[i - 2^k + 1] + a[i - 2^k + 2] + ... + a[i]   
其中k为i的二进制表示末尾0的个数,所以2^k即为i的二进制表示的最后一个1的权值.   
所以2^k可以表示为n&(n^(n-1))或更简单的n&(-n).

Cpp代码

int lowbit(int n)

{

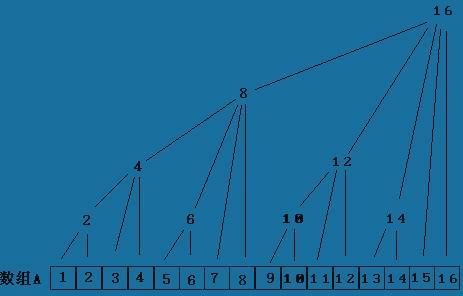
return n& (-n);

//or return n&(n^(n-1));

}

也就是说，把k表示成二进制1\*\*\*10000，那么c[k]就是1\*\*\*00001 + 1\*\*\*00010 + ... + 1\*\*\*10000这一段数的和。

举例



可以看出:设节点编号为x，那么这个节点管辖的区间为2^k个元素。（其中k为x二进制末尾0的个数）   
C1 = A1   
C2 = A1 + A2   
C3 = A3   
C4 = A1 + A2 + A3 + A4   
C5 = A5   
C6 = A5 + A6   
C7 = A7   
C8 = A1 + A2 + A3 + A4 + A5 + A6 + A7 + A8   
...   
C16 = A1 + A2 + A3 + A4 + A5 + A6 + A7 + A8 + A9 + A10 + A11 + A12 + A13 + A14 + A15 + A16   
**4,修改**   
修改一个节点，必须修改其所有祖先，最坏情况下为修改第一个元素，最多有log(n)的祖先。   
对a[n]进行修改后,需要相应的修改c数组中的p1, p2, p3...等一系列元素   
其中p1 = n,  pi+1 = pi + lowbit(pi)   
所以修改原数组中的第n个元素可以实现为:

Cpp代码

void Modify(int n, int delta)

{

while(n <= N)

{

c[n] += delta;

n += lowbit(n);

}

}

5,求和

当要查询a[1],a[2]...a[n]的元素之和时,需要累加c数组中的q1, q2, q3...等一系列元素

其中q1 = n,qi+1 = qi - lowbit(qi)

所以计算a[1] + a[2] + .. a[n]可以实现为:

Cpp代码

int Sum(int n)

{

int result = 0;

while(n != 0)

{

result += c[n];

n -= lowbit(n);

}

return result;

}

为什么是效率是log(n)的呢？以下给出证明：

n = n – lowbit(n)这一步实际上等价于将n的二进制的最后一个1减去。而n的二进制里最多有log(n)个1，所以查询效率是log(n)的。

换句话说:

若需改变a[i]，则c[i]、c[i+lowbit(i)]、c[i+lowbit(i)+lowbit(i+lowbit(i)]……就是需要改变的 c数组中的元素。

若需查询s[i]，则c[i]、c[i-lowbit(i)]、c[i-lowbit(i)-lowbit(i- lowbit(i))]……就是需要累加的c数组中的元素。

6,与线段树的比较

树状数组是一个可以很高效的进行区间统计的数据结构。在思想上类似于线段树，比线段树节省空间，编程复杂度比线段树低，但适用范围比线段树小。

7,应用

(1)http://acm.pku.edu.cn/JudgeOnline/problem?id=2155

首先对于每个数A

定义集合up(A)表示{A, A+lowestbit(A), A+lowestbit(A)+lowestbit(A+lowestbit(A))...}

定义集合down(A)表示{A, A-lowestbit(A), A-lowestbit(A)-lowestbit(A-lowestbit(A)) ... , 0}。

可以发现对于任何A<B，up(A)和down(B)的交集有且仅有一个数。

于是对于这道题目来说，翻转一个区间[A,B]（为了便于讨论先把原问题降为一维的情况），我们可以把down(B)的所有元素的翻转次数+1，再把down(A-1)的所有元素的翻转次数-1。而每次查询一个元素C时，只需要统计up(C)的所有元素的翻转次数之和，即为C实际被翻转的次数。

(2)http://acm.pku.edu.cn/JudgeOnline/problem?id=3321

一棵树上长了苹果，每一个树枝节点上有长苹果和不长苹果两种状态，两种操作，一种操作能够改变树枝上苹果的状态，另一种操作询问某一树枝节点一下的所有的苹果有多少。具体做法是做一次dfs，记下每个节点的开始时间low[i]和结束时间high[i]，那么对于i节点的所有子孙的开始时间和结束时间都应位于low[i]和high[i]之间，另外用一个数组c[i]记录附加在节点i上的苹果的个数，然后用树状数组统计low[i]到high[i]之间的附加苹果总数。这里用树状数组统计区间可以用Sum(high[i])-Sum(low[i]-1)来计算。

Cpp代码

#include <stdio.h>

#include <string.h>

#include <vector>

using namespace std;

//vector<int> g[100005];

struct Node

{

int v;

struct Node \*next;

}g[100005];

int n,m,cnt,low[100005],high[100005],c[100005],flag[100005];

bool mark[100005];

void dfs(int v)

{

struct Node \*p=g[v].next;

mark[v]=true;

cnt++;

low[v]=cnt;

while(p)

{

if(!mark[p->v])

dfs(p->v);

p=p->next;

}

high[v]=cnt;

}

int lowbit(int k)

{

return k&(-k);

}

void Modify(int num, int v)

{

while(num <= n)

{

c[num]+=v;

num+=lowbit(num);

}

}

int Sum(int num)

{

int ans=0;

while(num > 0)

{

ans+=c[num];

num-=lowbit(num);

}

return ans;

}

int main()

{

int i,a,b,ans;

char temp[10];

struct Node \*p;

//freopen("in.txt","r",stdin);

scanf("%d",&n);

memset(g,0,sizeof(g));

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

{

scanf("%d%d",&a,&b);

p=new Node;

p->next=g[a].next;

p->v=b;

g[a].next=p;

p=new Node;

p->next=g[b].next;

p->v=a;

g[b].next=p;

}

memset(mark,false,sizeof(mark));

memset(c,0,sizeof(c));

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

flag[i]=1;

cnt=0;

dfs(1);

scanf("%d",&m);

while(m--)

{

scanf("%s",temp);

if(temp[0] == 'Q')

{

scanf("%d",&a);

ans=high[a]-low[a]+1+Sum(high[a])-Sum(low[a]-1);

printf("%d\n",ans);

}

else

{

scanf("%d",&a);

if(flag[a]) Modify(low[a],-1);

else Modify(low[a],1);

flag[a]^=1;

}

}

return 0;

}

(3)http://acm.pku.edu.cn/JudgeOnline/problem?id=2481

给n个区间[Si,Ei]，区间[Sj,Ej]< [Si,Ei] 有 Si <= Sj and Ej <= Ei and Ei - Si > Ej – Sj。按y坐标从小到达，x坐标从大到小的顺序排序，然后从后往前扫描，记录i之前所有的j区间Sj<Si的个数，这个用树状数组实现。扫描一遍可得出结果。

Cpp代码

#include <stdio.h>

#include <string>

#include <algorithm>

using namespace std;

struct P

{

int x,y,id;

}p[100005];

int n,a[100005],max\_n,b[100005];

int lowbit(int k)

{

return k&(-k);

}

void Modify(int num, int v)

{

while(num <= max\_n)

{

a[num]+=v;

num+=lowbit(num);

}

}

int Sum(int num)

{

int ans=0;

if(num <= 0) return 0;

while(num)

{

ans+=a[num];

num-=lowbit(num);

}

return ans;

}

bool operator <(const P a, const P b)

{

if(a.y == b.y) return a.x > b.x;

return a.y < b.y;

}

int main()

{

int i;

//freopen("in.txt","r",stdin);

while(scanf("%d",&n), n)

{

max\_n=0;

for(i=0; i<n; i++)

{

scanf("%d%d",&p[i].x,&p[i].y);

p[i].id=i;

p[i].x++;

p[i].y++;

if(p[i].y > max\_n) max\_n=p[i].y;

}

sort(p,p+n);

memset(a,0,sizeof(a));

for(i=n-1; i>=0; i--)

{

if(i != n-1 && p[i].y == p[i+1].y && p[i].x == p[i+1].x)

b[p[i].id]=b[p[i+1].id];

else

b[p[i].id]=Sum(p[i].x);

Modify(p[i].x,1);

}

for(i=0; i<n; i++)

{

if(i) printf(" ");

printf("%d",b[i]);

}

printf("\n");

}

return 0;

}

(4)用树状数组求区间第K小元素

算法的时间复杂度是O(log(n))的，如果要求在线计算的话显然很有优势。

基本思路是：

先开一个数组，其中记录某个数出现次数，每输入一个树，相当于将该数出现次数加1，对应到树状数组中就相当于insert(t, 1),统计的时候，可以利用树状数组的求和，既可以二分枚举，也可以利用数的二进制表示，下面的代码有效地利用了数的二进制表示。

Cpp代码

#include <iostream>

using namespace std;

#define maxn 1<<20

int n,k;

int c[maxn];

int lowbit(int x){

return x&-x;

}

void insert(int x,int t){

while(x<maxn){

c[x]+=t;

x+=lowbit(x);

}

}

int find(int k){

int cnt=0,ans=0;

for(int i=20;i>=0;i--){

ans+=(1<<i);

if(ans>=maxn || cnt+c[ans]>=k)ans-=(1<<i);

else cnt+=c[ans];

}

return ans+1;

}

void input(){

memset(c,0,sizeof(c));

int t;

scanf("%d%d",&n,&k);

for(int i=0;i<n;i++){

scanf("%d",&t);

insert(t,1);

}

printf("%d\n",find(k));

}

int main(){

int cases;

scanf("%d",&cases);

while(cases--){

input();

}

return 0;

}