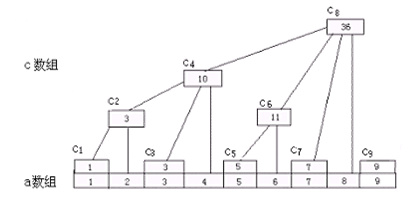
**树状数组和线段树及扩展（上）**

http://simg.sinajs.cn/blog7style/images/common/sg_trans.gif(2009-08-04 00:14:47)

由于达不到创造性的水平，但是为了能够造福大众，决定对树状数组、线段树（关于RMQ看（<http://blog.sina.com.cn/s/blog_4d88e9860100cthl.html> ）做一些总结。与以前较为敷衍的介绍不同，因为以前是一知半解，（现在已经比半解懂一点了），所以决定来一下介绍和总结。废话到此，言归正传。

树状数组，参阅《浅谈信息学竞赛中的“0”和“1”》，树状数组图如下：

[](http://photo.blog.sina.com.cn/showpic.html#blogid=4d88e9860100ewbb&url=http://static4.photo.sina.com.cn/orignal/4d88e986g719fe9fe42b3&690)

C1=a1

C2=a1+a2

C3=a3

C4=a1+a2+a3+a4

…

Cn=a(n-2^k+1)+…+an

它的管辖区域为2^k，其中k为n写成二进制时最后为0的个数。

#define lowbit(x) (x&(-x))

就可以求2^k.

其中update(int x)为增加或减少函数

void update(int x,int delta)

{

       while(x<=max)

       {

              c[x]+(-)=delta;

              x+(-)=lowbit(x);

       }

}

查询函数为：

int query(int x)

{

       int sum=0;

       while(x)

       {

              sum+=c[x];

              x-=lowbit(x);

       }

}

树状数组的查询和插入操作都是log(n)的，证明很显然，对于插入，一个数n,它二进制的1不会超过logn，对于查询，它的祖先不会超过n的所有位数（二进制时），所以也是logn。

树状数组也可以处理区间的整体加减，如果要在[a,b]加c，那么只要update(a,c) update(b+1,-c)，注意此时查询某个值的时候就只要找到当前的x位的值即可，不需要累加。

到这发现了树状数组的神奇了吧，代码及其简单，虽然线段树其实也有这几个函数，但是这个显然更简洁，而且空间也是n的，而线段树则至少是2倍。也许大家还会认为线段树毕竟比较NB，掌握了线段树这个就无足轻重了（我以前也是这么想的），但是现在觉得掌握树状数组是非常必要的。首先在现场的时候是要打代码的，而且要保证正确率，树状数组无疑更胜一筹，但是致命的一点是线段树的空间复杂度实在太高了，在扩展到多维的时候实在力不从心，代码量也大大增加，而树状数组则很不错。

不过，没有十全十美的事情，树状数组在查询区间最值问题时束手无策。

对于两者的比较，我将拿后面的matrix题来说明一下。

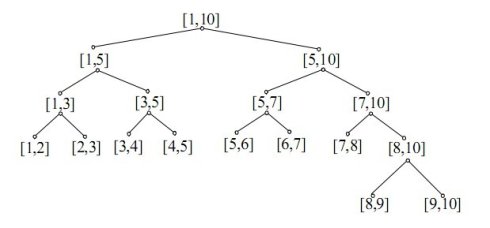
线段树（参阅《解决动态统计问题的两把利刃》）：

首先不得不感慨，线段树是很优美的东西，一般用在查询区间的极值（RMQ一般解决静态的，而线段树则是在线的），其中区间的整体操作还是有一些技巧的，当然还可以用在计算区间的和，实现树状数组实现的功能，还有关于矩形的各种统计（在我以前的博文已经给出例子和代码<http://blog.sina.com.cn/s/blog_4d88e9860100cv63.html>，关于矩形统计，如果是个数少坐标范围大，我推荐用矩形切割法），还有神奇的是线段树加上合并排序就是归并树，一会我会再讲一下。

简要介绍一下线段树：

线段树是一棵二叉树，其结点是一条“线段”——[a,b]，它的左儿子和右儿子分别是这条线段的左半段和右半段，即[a, (a+b)/2 ) ]和[(a+b)/2+1 ,b]。线段树的叶子结点是长度为1的单位线段[a,a+1]

下图

：[](http://photo.blog.sina.com.cn/showpic.html#blogid=4d88e9860100ewbb&url=http://static10.photo.sina.com.cn/orignal/4d88e986g703f2495b119&690)

简要的代码：（查询和修改区间和）

#define l(x) (x<<1)

#define r(x) ((x<<1)+1)

struct segtree

{

       int l,r,v;

};

segtree tree[maxn];

void build(int node,int l,int r)

{

       tree[node].l=l;

       tree[node].r=r;

       if(l>=r)return;

       int mid=(l+r)>>1;

       build(l(node),l,mid);

       build(r(node),mid+1,r);

}

void update(int node,int l,int r,int v)

{

       if(l==tree[node].l&&r==tree[node].r)

       {

              tree[node].v=v;

              return;

       }

       int mid=(tree[node].l+tree[node].r)>>1;

       if(r<=mid)update(l(node),l,r,v);

       else if(l>mid)update(r(node),l,r,v);

       else

       {

              update(l(node),l,mid,v);

              update(r(node),mid+1,r,v);

       }

}

int query(int node,int l,int r)

{

       if(l==tree[node].l&&r==tree[node].r)return tree[node].v;

       int mid=(tree[node].l+tree[node].r)>>1;

       if(r<=mid)return query(l(node),l,r);

       else if(l>mid)return query(r(node),l,r);

       else return query(l(node),l,mid)+query(r(node),mid+1,r);

}

线段树既然如此神奇，它的功效肯定不止是这样，有时候，我们会有一种操作，就是在区间[l,r]上修改v，看上面的代码，它是某点（l=r）修改，查询[l,r]区间的和，但是要整个区间改变的时候，不是这么简单，这一点为什么大家可以思考一下。

那么怎么解决这个问题呢，首先其实问题就是我们如果修改的是[l,r],但是当下次询问时与此区间不同，那么上次在[l,r]上的修改就不会反映在这次的查询上，比如在[1,5]上都加上v,如果我查询[1,3]就不会有效果，因为我们在update时，找到相应的l,r后就return了。当然有人会说那一个一个的修改，把整个区间的操作转化为单个的操作，那样就是以前的代码了。但是这样做以后，修改[l,r] len=r-l,那么时间复杂度变成了len\*log(len)，还不如直接数组存储然后直接修改，那样只要len,显然这不是我们希望看到的。

那么有没有办法来做到快速的修改呢，我们所做的其实并不是真正的去修改，而是换了一种思维，因为刚才的[1,5]修改后，并不一定每个子区间都会被访问到，于是我们采用标记法，当前在[l,r]上只是标记一下修改了，并在[l,r]的子区间“做手脚”。

代码：

void build(int node,int l,int r)

{

       tree[node].l=l;

       tree[node].r=r;

       tree[node].bj=0;

       if(l>=r)return;

       int mid=(l+r)>>1;

       build(l(node),l,mid);

       build(r(node),mid+1,r);

}

void modify(int node)

{

       tree[node].v+=tree[node].bj;

       int tt=tree[node].bj;

       tree[node].bj=0;

       if(tree[node].l==tree[node].r)return;

       tree[l(node)].bj+=tt;

       tree[r(node)].bj+=tt;

}

void update(int node,int l,int r,int v)

{

       if(l==tree[node].l&&r==tree[node].r)

       {

              tree[node].v+=v;

              if(tree[node].bj)modify(node);

          return;

       }

       int mid=(tree[node].l+tree[node].r)>>1;

       if(r<=mid)update(l(node),l,r,v);

       else if(l>mid)update(r(node),l,r,v);

       else

       {

              update(l(node),l,mid,v);

              update(r(node),mid+1,r,v);

       }

}

int query(int node,int l,int r)

{

       if(l==tree[node].l&&r==tree[node].r)

       {

              if(tree[node].bj)modify(node);

              return tree[node].v;

       }

       int mid=(tree[node].l+tree[node].r)>>1;

       if(tree[node].bj)modify(node);

       if(r<=mid)return query(l(node),l,r);

       else if(l>mid)return query(r(node),l,r);

       else return query(l(node),l,mid)+query(r(node),mid+1,r);

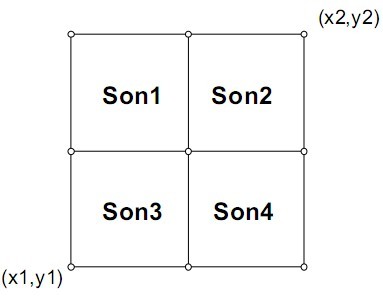
}

红色为修改后的，这样就可以处理线段修改了，通过每次修改的时候标记，并在下次访问到的时候进行处理，并将标记传到当前节点的儿子。

线段树不但可以处理线性的统计问题，还可以处理平面和空间的统计问题。比如要处理空间的问题，就要“方块树”，一种实现方式就是树中放树，这样的二位线段树即有2层，这种树其实操作起来比较麻烦，而且失去了刚才成段操作的性质，也就是说这种时候，一般要修改值时，第一维一般是一个数，第二维才是区间的（第二维就相当于一个普通的线段树，可以用标记的方法），例题可以看<http://blog.sina.com.cn/s/blog_4d88e9860100cud8.html>

这里还要介绍一种方法就是在原来的线段中直接变成矩形，这时候就是四分树。即一个矩形(x1,y1,x2,y2)四个儿子为(x1,y1,(x1+x2)/2,(y1+y2)/2),((x1+x2)/2,y1,x2,(y1+y2)/2),(x1,(y1+y2)/2,(x1+x2)/2,y2),((x1+x2)/2,(y1+y2)/2,x2,y2);

如图：

[](http://photo.blog.sina.com.cn/showpic.html#blogid=4d88e9860100ewbb&url=http://static16.photo.sina.com.cn/orignal/4d88e986g703f24495f0f&690)

不过这种方法在时间上相对“树套树”有劣势，空间上一样。在扩展到多维的时候，树套树基本无法接受，太繁琐了，而第二种方法就是变成2^n分法。

想一个问题，查询区间上第k大的数是多少？好像也可以用线段树做，那么怎么做呢。。于是发现一个很神奇的事情，把合并排序和线段树放在一起，于是形成了归并树。为什么可以放在一起呢，想一下关于线段树的构造，它是一个递归，叶子节点是长度为1的，再上一层是2，再上是4…而关于合并排序呢，显示以2为单位比较，然后是4… 可以说这两个过程完美的复合了。关于归并树的实现可以参照<http://blog.sina.com.cn/s/blog_4d88e9860100d0j6.html>

POJ 2155 matrix <http://acm.pku.edu.cn/JudgeOnline/problem?id=2155> 代码我在下一篇放出来。<http://blog.sina.com.cn/s/blog_4d88e9860100ewbc.html>