树状数组

Knight 叶

例：求连续和

求长度为n的数组A在A[a]…A[b]（1=<a<b<=n<10^5）之间的总和。

```
int c = 0;
for (int i = a; i <= b; i
++)
{
        c += A[i];
}
```

时间复杂度为O(n)

最大执行次数n_{max} = 10^5

求前缀和

```
int sum[MAXN];                    //用sum[i]表示A[1] + … + A[i] 的和
for (int i = 1; i <= n; i ++)
{
        sum[i] = sum[i - 1] + A[i];
}printf ("%d\n", sum[b] - sum[a - 1]);
```

单次查询时间复杂度O(1)，q次查询为O(q+n)

如果频繁的修改A[i] (0<i<n)?

此时前缀和sum[i]…sum[n]将被修改!

如果对sum[i]…sum[n]的值都进行修改！时间复杂度为O(n)！此时的问题将转化为如何**维护前缀和**?

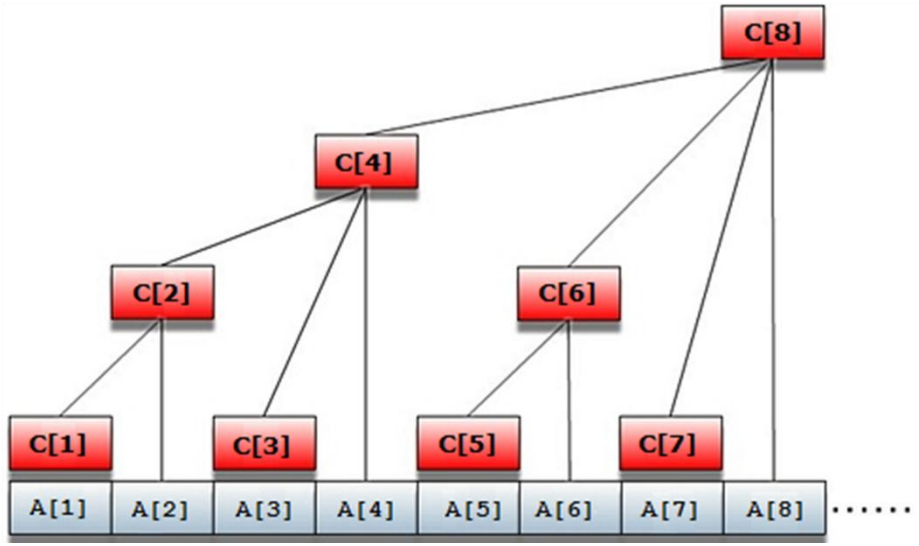
树状的介绍

- Binary Indexed Tree(**树状数组**)是一种**树型数据结构**，用于动态维护一个序列的前缀和。全部是由数组实现.
- 树状数组要用到二进制的知识,还有运算符操作的知识.
- 树状数组的更新,查询,求和和删除的时间复杂度都是(logN);

树状数组的模型

- 原数组是A，C是A的树状数组

- 令这棵树的结点编号为C[1]，C[2]...C[n]。令每个结点的值为这棵树的值的总和，那么容易发现：



Ci = A(i - 2^k+1) + ... + Ai

C[1] = A[1]
C[2] = A[1] + A[2]
C[3] = A[3]
C[4] = A[1] + A[2]+ A[3]+ A[4]
C[5] = A[5]
C[6] = A[5] + A[6]
C[7]= A[7]
C[8] = A[1] + A[2] + A[3] + A[4] + A[5] + A[6] + A[7]+ A[8]
...
C[16] = A[1] + A[2] + A[3] + A[4] + A[5] + A[6] + A[7] + A[8]+
A[9] + A[10] + A[11]+ A[12] + A[13] + A[14] + A[15] + A[16]

树状数组的实现

- 对于序列A，我们设一个数组C
  ① $C[i] = A[i - 2^k + 1] + \cdots + A[i]$
  ② k为i在二进制下末尾连续0的个数 k表示第k层
- C即为A的树状数组
- 如何求k?

这里有一个简便的方法:

2^k = i&(-i);

-i的二进制数是i的补码(反码+1).举个例子:

i = $(1000010)_2$

-i= $(0111101)_2$+ $(1)_2$ = $(0111110)_2$

i&(-i) = $(10)_2$ 所以2^k = 2

树状数组的更新

- 当我们改变A[x]的值时,我们只需对c[x],c[x+lowbit(x)],c[x+lowbit(x)+lowbit(x+lowbit(x))]……进行修改。
- {注：lowbit(x)=2^k}

- 如果是求和呢？ 暴力O(N).如果是M次求和呢？ O(M*N)

树状数组求和

- 我们要求数组A[1] 到A[x] 之间的和 sum[x]=A[1]+…+A[x].
- 我们只要求c[x]+c[x-lowbit(x)]+c[i-lowbit(x)-lowbit(x-lowbit(x))]……。其中：lowbit(x)=2^k;
- 这种处理方式等效于每次将x的二进制数的末尾1删去，进行分段计数
- 求x和y之间的和Sum(x,y)=A[x]+..+A[y]呢?
- Sum(x,y)=sum(y)-sum(x);

树状数组

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。

树状数组

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.

树状数组

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).

代码:

```
int lowbit(int n){
        return n& (-n); //or return n&(n^(n-1));
}
```

树状数组

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， p(i+1) = pi + lowbit(pi) 所以修改原数组中的第n个元素可以实现为:

```
void Modify(int n, int delta){
        while(n <= N)     {
                c[n] += delta;
                n += lowbit(n);
        }
}
```

树状数组

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。

练习题:

HDU 1541 1556 2352 2155

POJ 2299