树状数组(Binary Indexed Trees)

November 15, 2012
作者：Hawstein

出处：http://hawstein.com/posts/binary-indexed-trees.html

前言

本文翻译自TopCoder上的一篇文章： Binary Indexed Trees ，并非严格逐字逐句翻译，其中加入了自己的一些理解。水平有限，还望指摘。

目录

简介

我们常常需要某种特定的数据结构来使我们的算法更快，于是乎这篇文章诞生了。 在这篇文章中，我们将讨论一种有用的数据结构：数状数组(Binary Indexed Trees)。 按 Peter M. Fenwich (链接是他的论文，TopCoder上的链接已坏)的说法，这种结构最早是用于数据压缩的。现在它常常被用于存储频率及操作累积频率表。

定义问题如下：我们有n个盒子，可能的操作为：

1. 往第i个盒子增加石子(对应下文的update函数)
2. 计算第k个盒子到第l个盒子的石子数量(包含第k个和第l个)

原始的解决方案中(即用普通的数组进行存储，box[i]存储第i个盒子装的石子数)， 操作1和操作2的时间复杂度分别是O(1)和O(n)。假如我们进行m次操作，最坏情况下， 即全为第2种操作，时间复杂度为O(n*m)。使用某些数据结构(如 RMQ) ，最坏情况下的时间复杂度仅为O(m log n)，比使用普通数组为快许多。 另一种方法是使用数状数组，它在最坏情况下的时间复杂度也为O(m log n)，但比起RMQ， 它更容易编程实现，并且所需内存空间更少。

符号含义

- BIT: 树状数组
- MaxVal: 具有非0频率值的数组最大索引，其实就是问题规模或数组大小n
- f[i]: 索引为i的频率值，即原始数组中第i个值。i=1…MaxVal
- c[i]: 索引为i的累积频率值，c[i]=f[1]+f[2]+…+f[i]
- tree[i]: 索引为i的BIT值(下文会介绍它的定义)
- num^- : 整数num的补，即在num的二进制表示中，0换为1，1换成0。如：num=10101，则 num^- =01010

注意: 一般情况下，我们令f[0]=c[0]=tree[0]=0，所以各数组的索引都从 1 开始。 这样会给编程带来许多方便。

基本思想

每个整数都能表示为一些2的幂次方的和，比如13，其二进制表示为1101，所以它能表示为： $13 = 2^0 + 2^2 + 2^3$.类似的，累积频率可表示为其子集合之和。在本文的例子中， 每个子集合包含一些连续的频率值，各子集合间交集为空。比如累积频率c[13]= f[1]+f[2]+…+f[13]，可表示为三个子集合之和(数字3是随便举例的， 下面的划分也是随便举例的)，c[13]=s1+s2+s3， 其中s1=f[1]+f[2]+…+f[4]，s2=f[5]+f[6]+…+f[12]，s3=f[13]。

idx记为BIT的索引，r记为idx的二进制表示中最右边的1后面0的个数， 比如idx=1100(即十进制的12)，那么r=2。tree[idx]记为f数组中， 索引从(idx-2^r +1)到idx的所有数的和，包含f[idx-2^r +1]和f[idx]。即： tree[idx]=f[idx-2^r +1]+…+f[idx]，见表1.1和1.2，你就会一目了然。 我们也可称idx对索引(idx-2^r +1)到索引idx负责。(We also write that idx is responsible for indexes from (idx-2^r +1)to idx)

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
f	1	0	2	1	1	3	0	4	2	5	2	2	3	1	0	2
c	1	1	3	4	5	8	8	12	14	19	21	23	26	27	27	29
tree	1	1	2	4	1	4	0	12	2	7	2	11	3	4	0	29

Table 1.1

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
tree	1	1..2	3	1..4	5	5..6	7	1..8	9	9..10	11	9..12	13	13..14	15	1..16

Table 1.2 - table of responsibility

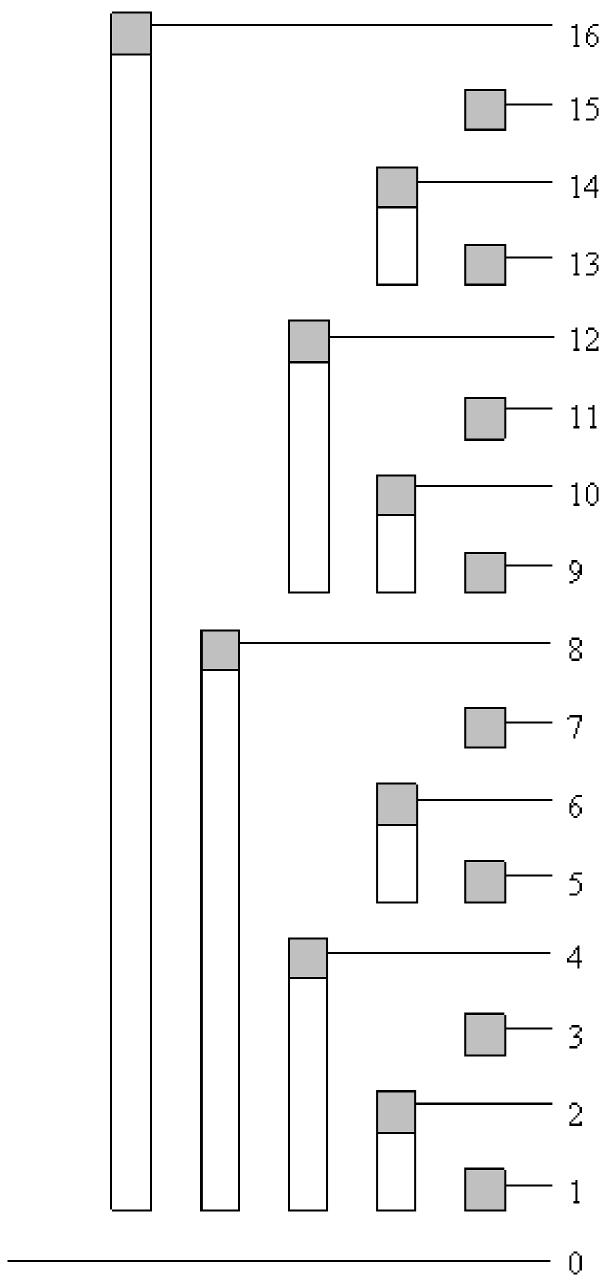

Image 1.3 - tree of responsibility for indexes (bar shows range of frequencies accumulated in top element)

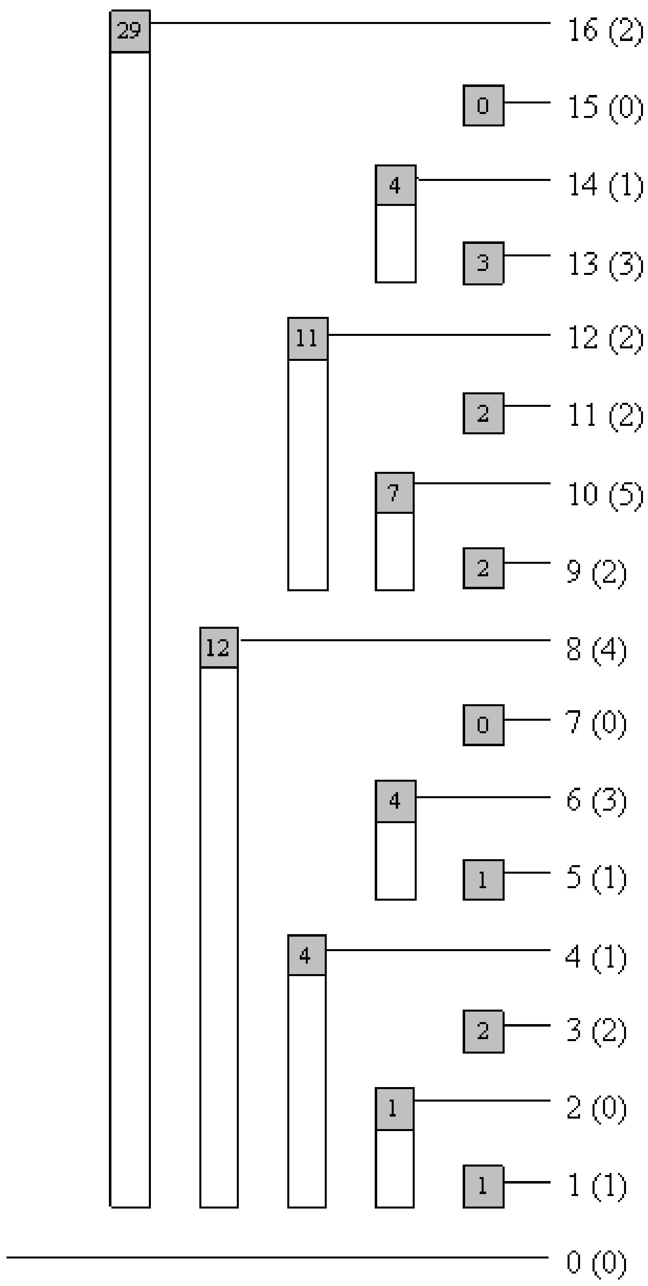

Image 1.4 - tree with tree frequencies

假设我们要得到索引为13的累积频率(即c[13])，在二进制表示中，13=1101。因此， 我们可以这样计算：c[1101]=tree[1101]+tree[1100]+tree[1000]，后面将详细讲解。

分离出最后的1

注意：最后的1表示一个整数的二进制表示中，从左向右数最后的那个1。

由于我们经常需要将一个二进制数的最后的1提取出来，因此采用一种高效的方式来做这件 事是十分有必要的。令num是我们要操作的整数。在二进制表示中，num可以记为a1b，a代表最后的1前面的二进制数码，由于a1b中的1代表的是从左向右的最后一个1， 因此b全为0，当然b也可以不存在。比如说13=1101，这里最后的1右边没有0，所以b不存在。

我们知道，对一个数取负等价于对该数的二进制表示取反加1。所以-num等于(a1b)^- +1= a^- 0b^- +1。由于b全是0，所以b^- 全为1。最后，我们得到：

-num=(a1b)^- +1=a^- 0b^- +1=a^- 0(1…1)+1=a^- 1(0…0)=a^- 1b

现在，我们可以通过与操作(在C++,java中符号为&)将num中最后的1分离出来：

num & -num = a1b & a^- 1b = (0…0)1(0…0)

读取累积频率

给定索引idx，如果我们想获取累积频率即c[idx]，我们只需初始化sum=0，然后当idx>0时，重复以下操作：sum加上tree[idx]，然后将idx最后的1去掉。 (C++代码如下)

```
int read(int idx){
        int sum = 0;
        while (idx > 0){
                sum += tree[idx];
                idx -= (idx & -idx);
        }
        return sum;
}
```

为什么可以这么做呢？关键是tree数组设计得好。我们知道，tree数组是这么定义的： tree[idx] = f[idx-2^r +1] +…+ f[idx]. 上面的程序sum加上tree[idx]后， 去掉idx最后的1，假设变为idx1，那么有idx1 = idx-2^r ， sum接下来加上tree[idx1] = f[idx1-2^{r1} +1] +…+ f[idx1] = f[idx1-2^{r1} +1] +…+ f[idx-2^r]， 我们可以看到tree[idx1]表达示的最右元素为f[idx-2^r]，这与tree[idx]表达式的最左元素f[idx-2^r +1]无缝地连接了起来。所以，只需要这样操作下去，即可求得f[1]+…+ f[idx]，即c[idx]的结果。

来看一个具体的例子，当idx=13时，初始sum=0:

```
tree[1101]=f[13]
tree[1100]=f[9]+...+f[12]
tree[1000]=f[1]+...+f[8]
c[1101]=f[1]+...+f[13]=tree[1101]+tree[1100]+tree[1000]
```

iteration	idx	position of the last digit	idx & -idx	sum
1	13 = 1101	0	1 (2 ^0)	3
2	12 = 1100	2	4 (2 ^2)	14
3	8 = 1000	3	8 (2 ^3)	26
4	0 = 0	---	---	---

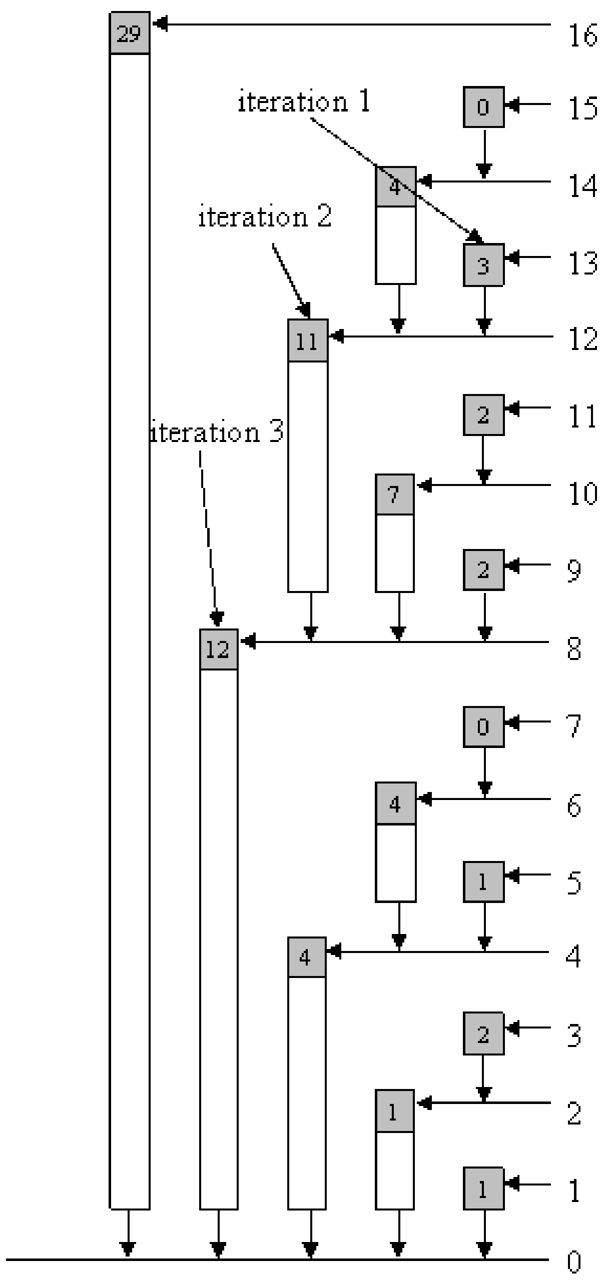

Image 1.5 - arrows show path from index to zero which we use to get sum (image shows example for index 13)

read函数迭代的次数是idx二进制表示中位的个数，其最大值为log(MaxVal)。 在本文中MaxVal=16。

时间复杂度：O(log MaxVal)
代码长度：不到10行

改变某个位置的频率并且更新数组

当我们改变f数组中的某个值，比如f[idx]，那么tree数组中哪些元素需要改变呢？ 在读取累积频率一节，我们每累加一次tree[idx]，就将idx最后一个1移除， 然后重复该操作。而如果我们改变了f数组，比如f[idx]增加val，我们则需要为当前索引的 tree数组增加val：tree[idx] += val。然后idx更新为idx加上其最后的一个1， 当idx不大于MaxVal时，不断重复上面的两个操作。详情见以下C++函数：

```
void update(int idx ,int val){
        while (idx <= MaxVal){
                tree[idx] += val;
                idx += (idx & -idx);
        }
}
```

接下来让我们来看一个例子，当idx=5时：

iteration	idx	position of the last digit	idx & -idx
1	5 = 101	0	1 (2 ^0)
2	6 = 110	1	2 (2 ^1)
3	8 = 1000	3	8 (2 ^3)
4	16 = 10000	4	16 (2 ^4)
5	32 = 100000	---	---

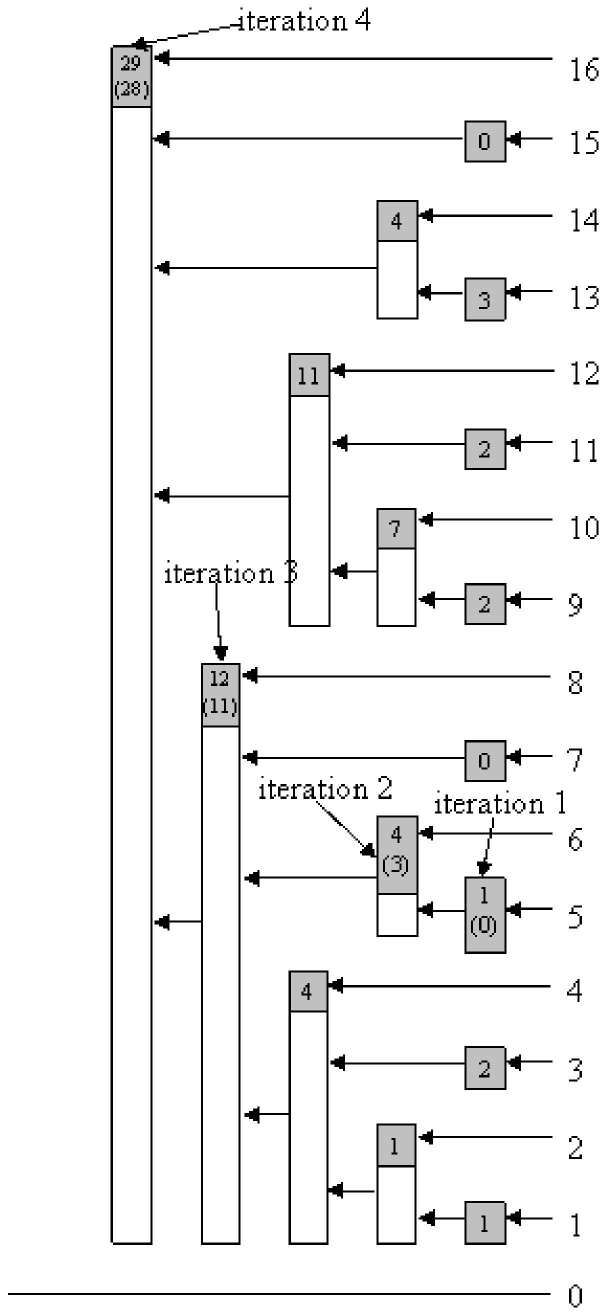

*Image 1.6 - Updating tree (in brackets are tree frequencies before updating); arrows show path while we update tree from index to **MaxVal** (image shows example for index 5)*

使用上面的算法或者按照图1.6的箭头所示去操作，我们即可更新BIT。

时间复杂度：O(log MaxVal)
代码长度：不到10行

读取某个位置的实际频率

上面我们已经讨论了如何读取指定索引的累积频率值(即c[idx])，很明显我们无法通过 tree[idx]直接读取某个位置的实际频率f[idx]。有人说，我们另外再开一个数组来存储f数 组不就可以了。这样一来，读和存f[idx]都只需要O(1)的时间，而空间复杂度则是O(n)的。 不过如果考虑到节约内存空间是更重要的话，我们就不能这么做了。接下来我们将展示在不 增加内存空间的情况下，如何读取f[idx]。(事实上，本文所讨论的问题都是基于我们只维 护一个tree数组的前提)

事实上，有了前面的讨论，要得到f[idx]是一件非常容易的事： f[idx] = read[idx] - read[idx-1]。即前idx个数的和减去前idx-1个数的和， 然后就是f[idx]了。这种方法的时间复杂度是2*O(log n)。下面我们将重新写一个函数， 来得到一个稍快一点的版本，但其本质思想其实和read[idx]-read[idx-1]是一样的。

假如我们要求f[12]，很明显它等于c[12]-c[11]。根据上文讨论的规律，有如下的等式: (为了方便理解，数字写成二进制的表示)

```
c[12]=c[1100]=tree[1100]+tree[1000]
c[11]=c[1011]=tree[1011]+tree[1010]+tree[1000]
f[12]=c[12]-c[11]=tree[1100]-tree[1011]-tree[1010]
```

从上面3个式子，你发现了什么？没有错，c[12]和c[11]中包含公共部分，而这个公共部分 在实际计算中是可以不计算进来的。那么，以上现象是否具有一般规律性呢？或者说， 我怎么知道，c[idx]和c[idx-1]的公共部分是什么，我应该各自取它们的哪些tree元素来做 差呢？下面将进入一般性的讨论。

让我们来考察相邻的两个索引值idx和idx-1。我们记idx-1的二进制表示为a0b(b全为1)， 那么idx即a0b+1=a1b^- .(b^- 全为0)。使用上文中读取累积频率的算法(即read函数) 来计算c[idx]，当sum加上tree[idx]后(sum初始为0)，idx减去最后的1得a0b^- ， 我们将它记为z。

用同样的方法去计算c[idx-1]，因为idx-1的二进制表示是a0b(b全为1)，那么经过一定数量 的循环后，其值一定会变为a0b^- ,(不断减去最后的1)，而这个值正是上面标记的z。那么， 到这里已经很明显了，z往后的tree值是c[idx]和c[idx-1]都共有的， 相减只是将它们相互抵

消，所以没有必要往下再计算了。

也就是说，c[idx]-c[idx-1]等价于取出tree[idx]，然后当idx-1不等于z时，不断地减去 其对应的tree值，然后更新这个索引(减去最后的1)。当其等于z时停止循环(从上面的分析 可知，经过一定的循环后，其值必然会等于z)。下面是C++函数：

```
int readSingle(int idx){
int sum = tree[idx]; // sum will be decreased
if (idx > 0){ // special case
        int z = idx - (idx & -idx); // make z first
        idx--; // idx is no important any more, so instead y, you can use idx
        while (idx != z){ // at some iteration idx (y) will become z
                sum -= tree[idx];
// substruct tree frequency which is between y and "the same path"
                idx -= (idx & -idx);
        }
}
return sum;
}
```

下面我们来看看根据这个算法，f[12]是怎么计算出来的：

首先，计算z值：z = 12 - (12 & -12) = 8，sum = tree[12] = 11(见表1.1)

iteration	y	position of the last digit	y & -y	sum
1	11 = 1011	0	1 (2 ^0)	9
2	10 = 1010	1	2 (2 ^1)	2
3	8 = 1000	---	---	---

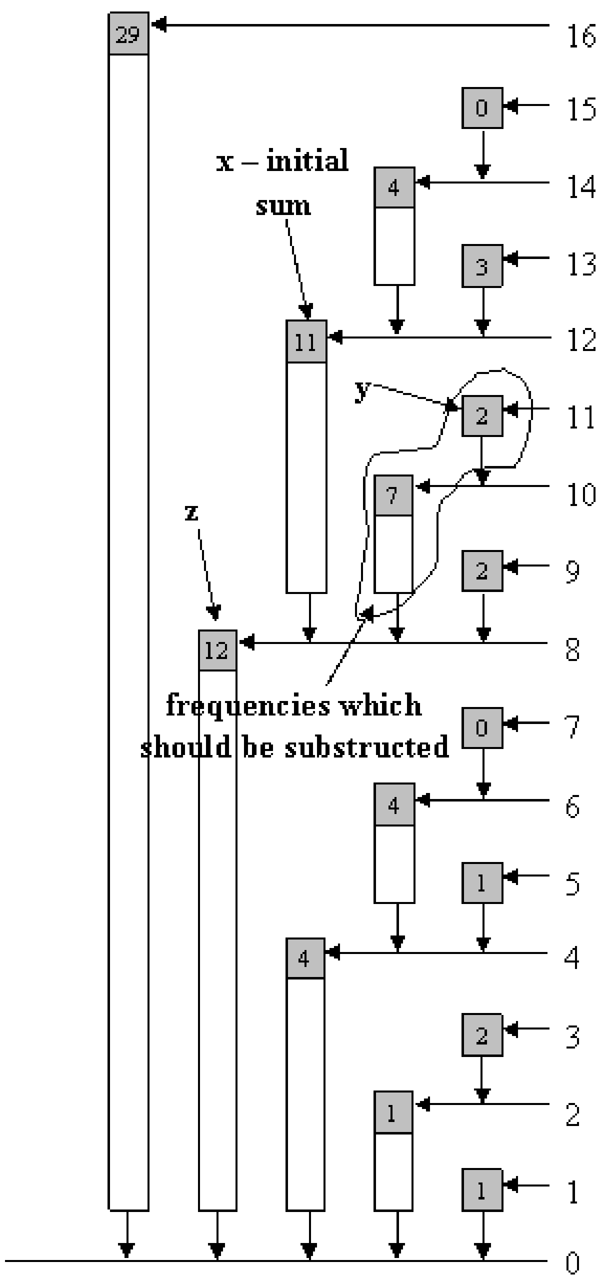

Image 1.7 - read actual frequency at some index in BIT
(image shows example for index 12)

对比该算法及调用两次read函数的方法，当idx为奇数时，该算法的时间复杂度仅为O(1)， 迭代次数为0。而对于几乎所有的偶数idx，其时间复杂度为c*O(log idx)， 其中c严格小于1。而read(idx)-read(idx-1)的时间复杂度为c1*O(log idx)， 其中c1总是大于1.

时间复杂度：c*O(log MaxVal),c严格小于1
代码长度：不到15行

缩放整个数状数组

有时候我们需要缩放整个f数组，然后更新tree数组。利用上面讨论的结论，我们可以轻松 地达到这个目的。比如，我们要将f[idx]变为f[idx]/c，我们只需要调用上面的update 函数，然后把除以c转变为加上-(c-1)*readSingle(idx)/c即可。这个很容易理解， f[idx]-(c-1)*f[idx]/c = f[idx]/c。用一个for循环即可将所有的tree元素更新。 代码如下：

```
void scale(int c){
        for (int i = 1 ; i <= MaxVal ; i++)
                update(-(c - 1) * readSingle(i) / c , i);
}
```

上面的方法似乎有点绕，其实，我们有更快的方法。除法是线性操作，而tree数组中的元素 又是f数组元素的线性组合。因此，如果我们用一个因子去缩放f数组，我们就可以用该因子去 直接缩放tree数组，而不必像上面程序那样麻烦。上面程序的时间复杂度为 O(MaxVal*log MaxVal)，而下面的程序只需要O(MaxVal)的时间：

```
void scale(int c){
        for (int i = 1 ; i <= MaxVal ; i++)
                tree[i] = tree[i] / c;
}
```

时间复杂度：O(MaxVal)
代码长度：几行

返回指定累积频率的索引

问题可描述为：给你一个累积频率值cumFre，如果存在c[idx]=cumFre，则返回idx； 否则返回-1。该问题最朴素及最简单的解决方法是求出依次求出c[1]到c[MaxVal]， 然后与给出的cumFre对比，如果存在c[idx]=cumFre，则返回idx;否则返回-1。 如果f数组中存在负数，那么该方法就是唯一的解决方案。但如果f数组是非负的， 那么c数组一定是非降的。即如果i>=j，则c[i]>=c[j]。这种情况下，利用二分查找的思想， 我们可以写出时间复杂度为O(log n)的算法。我们从MaxVal的最高位开始(比如本文中 MaxVal是16,所以tIdx从二进制表示10000即16开始)，比较cumFre和tree[tIdx] 的值，根据其比较结果，决定在大的一半区间还是在小的一半区间继续进行查找。 C++函数如下：(如果c数组中存在多个cumFre，find函数返回任意其中一个，findG返回最大 的idx值)

```
// if in tree exists more than one index with a same
// cumulative frequency, this procedure will return
// some of them (we do not know which one)

// bitMask - initialy, it is the greatest bit of MaxVal
// bitMask store interval which should be searched
int find(int cumFre){
        int idx = 0; // this var is result of function

        while ((bitMask != 0) && (idx < MaxVal)){ // nobody likes overflow :)
                int tIdx = idx + bitMask; // we make midpoint of interval
                if (cumFre == tree[tIdx]) // if it is equal, we just return idx
                        return tIdx;
                else if (cumFre > tree[tIdx]){
                        // if tree frequency "can fit" into cumFre,
                        // then include it
                        idx = tIdx; // update index
                        cumFre -= tree[tIdx]; // set frequency for next loop
                }
                bitMask >>= 1; // half current interval
        }
        if (cumFre != 0) // maybe given cumulative frequency doesn't exist
                return -1;
        else
                return idx;
}



// if in tree exists more than one index with a same
// cumulative frequency, this procedure will return
// the greatest one
int findG(int cumFre){
        int idx = 0;

        while ((bitMask != 0) && (idx < MaxVal)){
                int tIdx = idx + bitMask;
                if (cumFre >= tree[tIdx]){
                        // if current cumulative frequency is equal to cumFre,
                        // we are still looking for higher index (if exists)
                        idx = tIdx;
                        cumFre -= tree[tIdx];
                }
                bitMask >>= 1;
        }
        if (cumFre != 0)
                return -1;
        else
                return idx;
}
```

来看一个例子，当要查找的累积频率是21时，下面的过程将展示算法是如何进行的： (这里我就不翻译了，偷个懒)

First iteration	tIdx is 16; tree[16] is greater than 21; half bitMask and continue
Second iteration	tIdx is 8; tree[8] is less than 21, so we should include first 8 indexes in result, remember idx because we surely know it is part of result; subtract tree[8] of cumFre (we do not want to look for the same cumulative frequency again - we are looking for another cumulative frequency in the rest/another part of tree); half bitMask and contiue
Third iteration	tIdx is 12; tree[12] is greater than 9 (there is no way to overlap interval 1-8, in this example, with some further intervals, because only interval 1-16 can overlap); half bitMask and continue
Forth iteration	tIdx is 10; tree[10] is less than 9, so we should update values; half bitMask and continue
Fifth iteration	tIdx is 11; tree[11] is equal to 2; return index (tIdx)

时间复杂度：O(log MaxVal)
代码长度：不到20行

2D BIT(Binary Indexed Trees)

BIT可被扩展到多维的情况。假设在一个布满点的平面上(坐标是非负的)。 你有以下三种查询：

1. 将点(x, y)置1
2. 将点(x, y)置0
3. 计算左下角为(0, 0)右上角为(x, y)的矩形内有多少个点(即有多少个1)

如果m是查询次数，max_x和max_y分别是最大的x坐标和最大的y坐标，那么解决该问题的 时间复杂度为O(m*log(max_x)*log(max_y))。在这个例子中，tree是个二维数组。 对于tree[x][y]，当固定x坐标时，更新y坐标的过程与一维情况相同。 如果我们想在点(a, b)处置1/0，我们可以调用函数update(a,b,1)/update(a,b,-1)， 其中update函数如下：

```
void update(int x , int y , int val){
        while (x <= max_x){
                updatey(x , y , val);
                // this function should update array tree[x]
                x += (x & -x);
        }
}
```

其中updatey函数与update函数是相似的：

```
void updatey(int x , int y , int val){
        while (y <= max_y){
                tree[x][y] += val;
                y += (y & -y);
        }
}
```

以上两个函数可以整合成一个函数：

```
void update(int x , int y , int val){
        int y1;
        while (x <= max_x){
                y1 = y;
                while (y1 <= max_y){
                        tree[x][y1] += val;
                        y1 += (y1 & -y1);
                }
                x += (x & -x);
        }
}
```

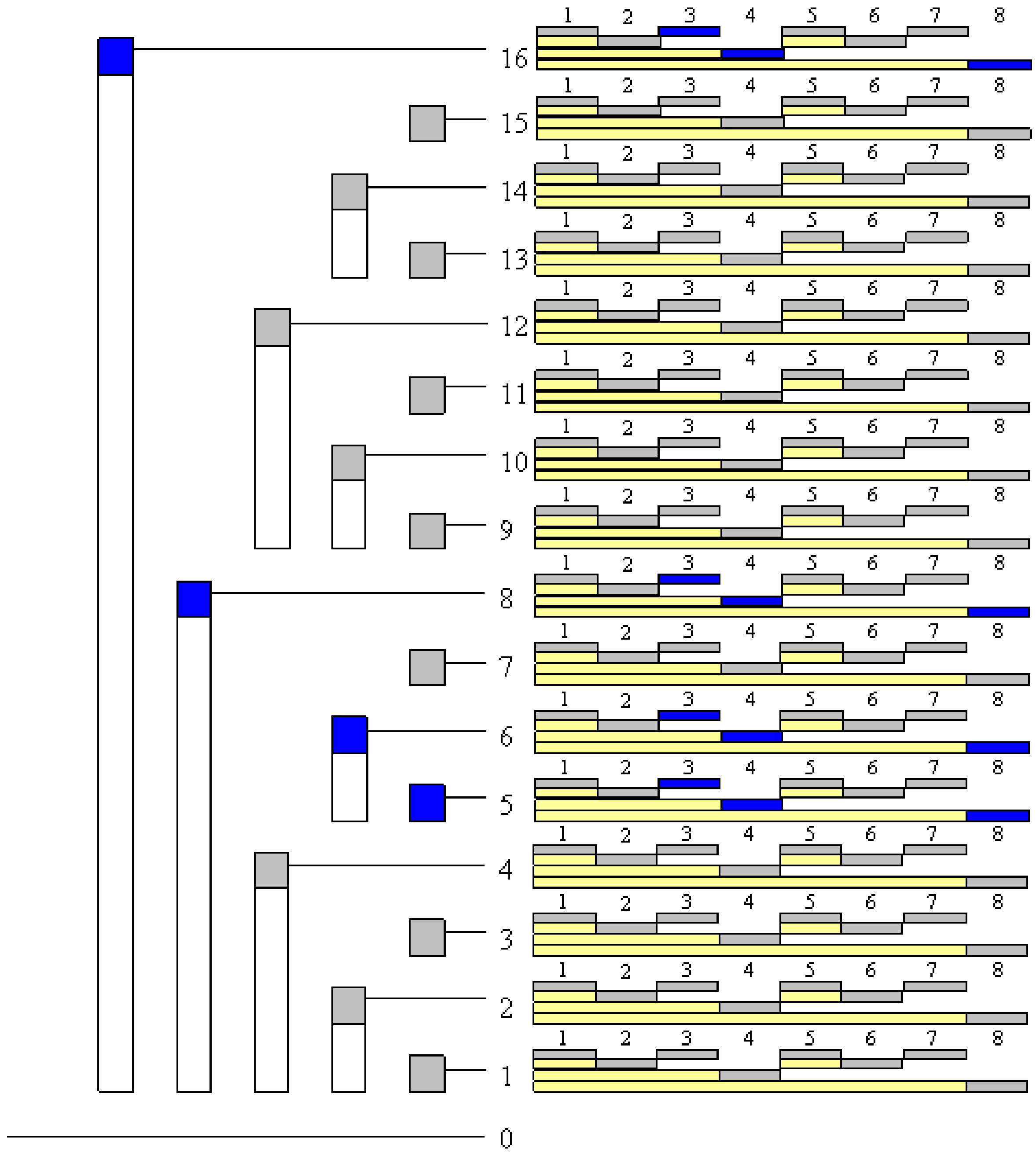

Image 1.8 - BIT is array of arrays, so this is two-dimensional BIT (size 16 x 8).
Blue fields are fields which we should update when we are updating index (5 , 3).

其它函数的修改也非常相似，这里就不一一写出来了。此外，BIT也可被扩展到n维的情况。

问题样例

- SRM 310-FloatingMedian
- 问题2:

  描述:

  n张卡片摆成一排，分别为第1张到第n张，开始时它们都是下面朝下的。你有两种操作:

  1. T(i,j):将第i张到第j张卡片进行翻转，包含i和j这两张。(正面变反面，反面变正面)
  2. Q(i):如果第i张卡片正面朝下，返回0；否则返回1.

  解决方案:

  操作1和操作2都有O(log n)的解决方案。设数组f初始全为0，当做一次T(i, j)操作后， 将f[i]加1，f[j+1]减1.这样一来，当我们做一次Q(i)时，只需要求f数组的前i项和c[i] ，然后对2取模即可。结合图2.0，当我们做完一次T(i, j)后，f[i]=1，f[j+1]=-1。 这样一来，当k<i时，c[k]%2=0，表明正面朝下；当i<=k<=j时，c[k]%2=1，表明正面朝 上(因为这区间的卡片都被翻转了！)；当k>j时，c[k]%2=0，表示卡片正面朝下。 Q(i)返回的正是我们要的判断。

  注意：这里我们使用BIT结构，所以只维护了一个tree数组，并没有维护f数组。 所以，虽然做一次T(i, j)只需要使f[i]加1，f[j+1]减1，但更新tree数组还是需要 O(log n)的时间；而读取c[k]的时间复杂度也是O(log n)。这里其实只用到了一维BIT 的update函数和read函数。

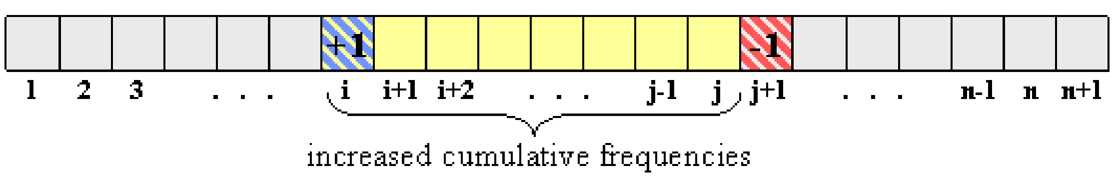

Image 2.0

总结

- 树状数组十分容易进行编程实现

- 树状数组的每个操作花费常数时间或是(log n)的时间
- 数状数组需要线性的存储空间(O(n)，只维护tree数组)
- 树状数组可扩展成n维的情况

参考资料

[1] RMQ

[2] Binary Search

[3] Peter M. Fenwick

Random Posts

- 06 Mar 2014 » 把《把时间当作朋友》读薄
- 20 Jan 2014 » Google Java编程风格指南
- 11 Aug 2013 » 把《编程珠玑》读薄
- 23 Jul 2013 » 如何用C++实现一个LRU Cache
- 18 Jul 2013 » 微信公众平台：程序员的面试吧

gaotong • 8 months ago

高水平文章，可以发表论文了

• Reply • Share ›

biaobiaoqi • 10 months ago

精彩。搜其他资料，大都不明所以。

• Reply • Share ›

Qiang Qin • a year ago

讲的非常清晰，大师级！

• Reply • Share ›

greedydaam • a year ago

超级超级棒！

• Reply • Share ›

刘俊 • a year ago

非常谢谢！ 终于让我对树状数组有了个比较清晰的理解！

• Reply • Share ›

Hawstein Mod → 刘俊 • a year ago

很高兴能对你有所帮助:-)

• Reply • Share ›