Binary Indexed Trees

分类： C++ 算法 数据结构 2013-06-02 15:17 224人阅读 评论(0) 收藏 举报

目录(?) [-]

Algorithm Tutorials

Binary Indexed Trees

Archive
Printable view
Discuss this article
Write for TopCoder

By boba5551
TopCoder Member

Introduction
We often need some sort of data structure to make our algorithms faster. In this article we will discuss the **Binary Indexed Trees** structure. According to Peter M. Fenwick, this structure was first used for data compression. Now it is often used for storing frequencies and manipulating cumulative frequency tables.

Let's define the following **problem**: We have n boxes. Possible queries are
1. add marble to box **i**
2. sum marbles from box **k** to box **l**

The naive solution has time complexity of O(1) for query 1 and O(n) for query 2. Suppose we make **m** queries. The worst case (when all queries are 2) has time complexity O(n * m). Using some data structure (i.e. RMQ) we can solve this problem with the worst case time complexity of O(m log n). Another approach is to use Binary Indexed Tree data structure, also with the worst time complexity O(m log n) -- but Binary Indexed Trees are much easier to code, and require less memory space, than RMQ.

Notation
BIT - **B**inary **I**ndexed **T**ree
MaxVal - maximum value which will have non-zero frequency
f[i] - frequency of value with index **i**, **i** = 1 .. MaxVal
c[i] - cumulative frequency for index **i** (f[1] + f[2] + ... + f[i])
tree[i] - sum of frequencies stored in **BIT** with index **i** (latter will be described what index means); sometimes we will write *tree frequency* instead *sum of frequencies stored in BIT*
num¯ - complement of integer **num** (integer where each binary digit is inverted: 0 -> 1; 1 -> 0)
NOTE: Often we put f[0] = 0, c[0] = 0, tree[0] = 0, so sometimes I will just ignore index 0.

Basic idea
Each integer can be represented as sum of powers of two. In the same way, cumulative frequency can be represented as sum of sets of subfrequencies. In our case, each set contains some successive number of non-overlapping frequencies.

idx is some index of **BIT**. **r** is a position in **idx** of the last digit 1 (from left to right) in binary notation. **tree[idx]** is sum of frequencies from index (**idx** - 2^**r** + 1) to index **idx** (look at the Table 1.1 for clarification). We also write that **idx** is **responsible** for indexes from (**idx** - 2^**r** + 1) to **idx** (note that responsibility is the key in our algorithm and is the way of manipulating the tree).

	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

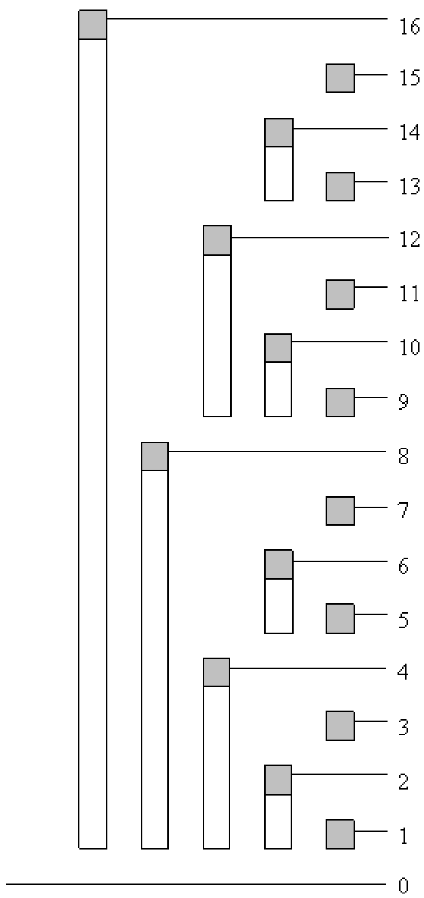

Image 1.3 - tree of responsibility for indexes (bar shows range of frequencies accumulated in top element)

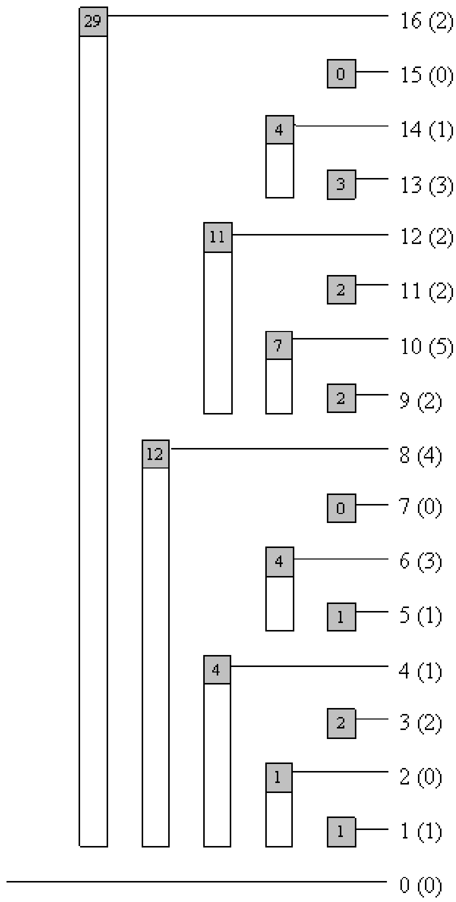

Image 1.4 - tree with tree frequencies

Suppose we are looking for cumulative frequency of index 13 (for the first 13 elements). In binary notation, 13 is equal to 1101. Accordingly, we will calculate**c[1101] = tree[1101] + tree[1100] + tree[1000]** (more about this later).

Isolating the last digit
NOTE: Instead of "the last non-zero digit," it will write only "the last digit."

There are times when we need to get just the last digit from a binary number, so we need an efficient way to do that. Let **num** be the integer whose last digit we want to isolate. In binary notation **num** can be represented as **a1b**, where **a** represents binary digits before the last digit and **b** represents zeroes after the last digit.

Integer **-num** is equal to $\mathbf{\overline{(a1b)} + 1 = \overline{a}0\overline{b} + 1}$. **b** consists of all zeroes, so $\mathbf{\overline{b}}$ consists of all ones. Finally we have

$$\mathbf{-num = \overline{(a1b)} + 1 = \overline{a}0\overline{b} + 1 = \overline{a}0\overline{(0...0)} + 1 = \overline{a}0(1...1) + 1 = \overline{a}1(0...0) = \overline{a}1b}.$$

Now, we can easily isolate the last digit, using bitwise operator **AND** (in C++, Java it is **&**) with **num** and **-num**:

$$\begin{array}{rl} & a1b \\ \& & \overline{a}1b \\ \hline & = (0...0)1(0...0) \end{array}$$

Read cumulative frequency
If we want to read cumulative frequency for some integer **idx**, we add to **sum tree[idx]**, substract last bit of **idx** from itself (also we can write - remove the last digit; change the last digit to zero) and repeat this while **idx** is greater than zero. We can use next function (written in C++)

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

Example for **idx** = 13; **sum** = 0:

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	---	---	---

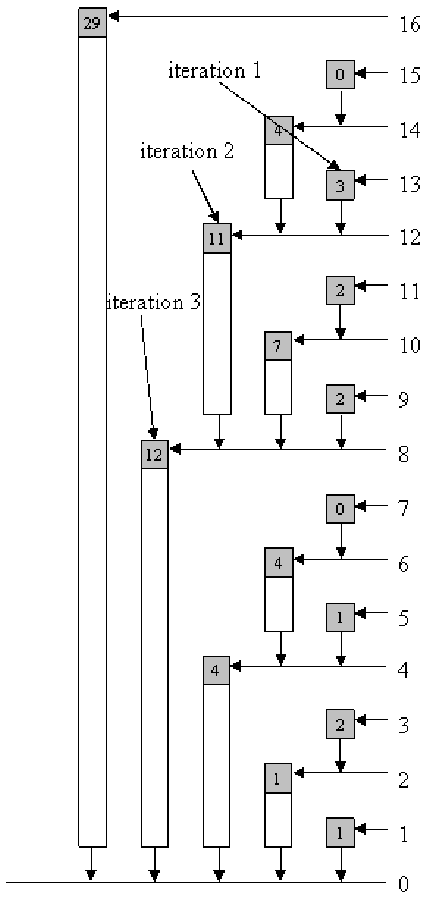

Image 1.5 - arrows show path from index to zero which we use to get sum (image shows example for index 13)

So, our result is 26. The number of iterations in this function is number if bits in **idx**, which is at most **log MaxVal**.

> *Time complexity:* O(log MaxVal).
> *Code length:* Up to ten lines.

Change frequency at some position and update tree
The concept is to update tree frequency at all indexes which are responsible for frequency whose value we are changing. In reading cumulative frequency at some index, we were removing the last bit and going on. In changing some frequency **val** in tree, we should increment value at the current index (the starting index is always the one whose frequency is changed) for **val**, add the last digit to index and go on while the index is less than or equal to **MaxVal**. Function in C++:

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

Let's show example for **idx** = 5:

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	---	---

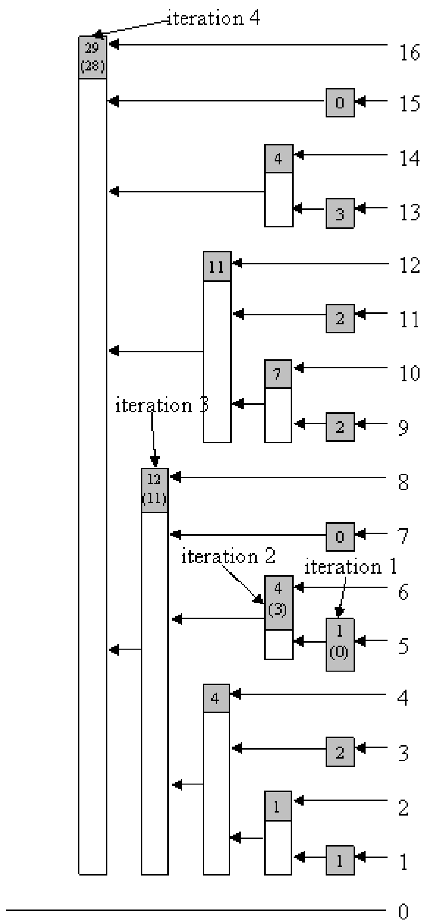

*Image 1.6 - Updating tree (in brackets are tree frequencies before updating); arrows show path while we update tree from index to **MaxVal** (image shows example for index 5)*

Using algorithm from above or following arrows shown in Image 1.6 we can update **BIT**.

> *Time complexity:* O(log MaxVal).
> *Code length:* Up to ten lines.

Read the actual frequency at a position
We've described how we can read cumulative frequency for an index. It is obvious that we can not read just **tree[idx]** to get the actual frequency for value at index**idx**. One approach is to have one aditional array, in which we will seperately store frequencies for values. Both reading and storing take O(1); memory space is linear. Sometimes it is more important to save memory, so we will show how you can get actual frequency for some value without using aditional structures.

Probably everyone can see that the actual frequency at a position **idx** can be calculated by calling function **read** twice -- **f[idx] = read(idx) - read(idx - 1)** -- just by taking the difference of two adjacent cumulative frequencies. This procedure always works in 2 * O(log n) time. If we write a new function, we can get a bit faster algorithm, with smaller const.

If two paths from two indexes to root have the same part of path, then we can calculate the sum until the paths meet, substract stored sums and we get a sum of frequencies between that two indexes. It is pretty simple to calculate sum of frequencies between adjacent indexes, or read the actual frequency at a given index.

Mark given index with **x**, its predecessor with **y**. We can represent (binary notation) **y** as **a0b**, where **b** consists of all ones. Then, **x** will be $\mathbf{a1\bar{b}}$ (note that $\mathbf{\bar{b}}$ consists all zeros). Using our algorithm for getting **sum** of some index, let it be **x**, in first iteration we remove the last digit, so after the first iteration **x** will be$\mathbf{a0\bar{b}}$, mark a new value with **z**.

Repeat the same process with **y**. Using our function for reading **sum** we will remove the last digits from the number (one by one). After several steps, our **y** will become (just to remind, it was **a0b**) $\mathbf{a0\bar{b}}$, which is the same as **z**. Now, we can write our algorithm. Note that the only exception is when **x** is equal to 0. Function in C++:

```
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;
}
```

Here's an example for getting the actual frequency for index 12:

First, we will calculate **z = 12 - (12 & -12) = 8**, **sum = 11**

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	---	---	---

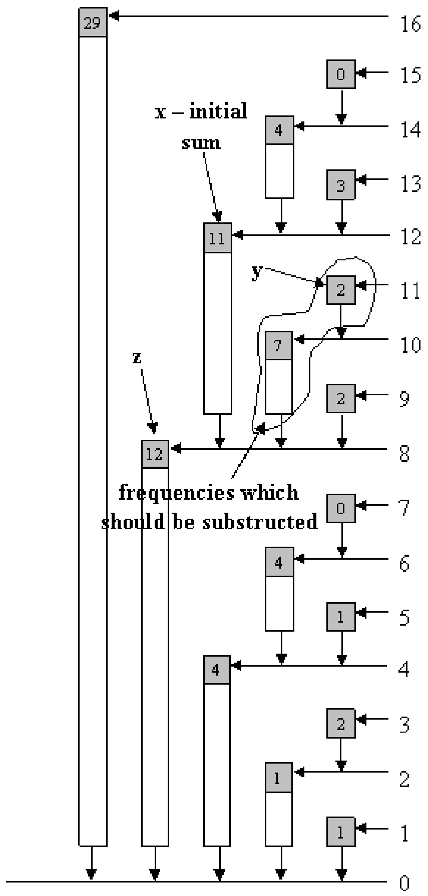

Image 1.7 - read actual frequency at some index in BIT
(image shows example for index 12)

Let's compare algorithm for reading actual frequency at some index when we twice use function **read** and the algorithm written above. Note that for each odd number, the algorithm will work in const time O(1), without any iteration. For almost every even number **idx**, it will work in c * O(log idx), where c is strictly less than 1, compare to **read(idx) - read(idx - 1),** which will work in c1 * O(log idx), where c1 is **always** greater than 1.

> *Time complexity:* c * O(log MaxVal), where c is less than 1.
> *Code length:* Up to fifteen lines.

Scaling the entire tree by a constant factor
Sometimes we want to scale our tree by some factor. With the procedures described above it is very simple. If we want to scale by some factor **c**, then each index **idx** should be updated by **-(c - 1) * readSingle(idx) / c** (because **f[idx] - (c - 1) * f[idx] / c = f[idx] / c**). Simple function in C++:

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

This can also be done more quickly. Factor is linear operation. Each tree frequency is a linear composition of some frequencies. If we scale each frequency for some factor, we also scaled tree frequency for the same factor. Instead of rewriting the procedure above, which has time complexity O(MaxVal * log MaxVal), we can achieve time complexity of O(MaxVal):

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

Time complexity: O(MaxVal).
Code length: Just a few lines.

Find index with given cumulative frequency
The naive and most simple solution for finding an index with a given cumultive frequency is just simply iterating through all indexes, calculating cumulative frequency, and checking if it's equal to the given value. In case of negative frequencies it is the only solution. However, if we have only non-negative frequencies in our tree (that means cumulative frequencies for greater indexes are not smaller) we can figure out logarithmic algorithm, which is modification of binary search. We go through all bits (starting with the highest one), make the index, compare the cumulative frequency of the current index and given value and, according to the outcome, take the lower or higher half of the interval (just like in binary search). Function in C++:

```
// 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;
}
```

Example for cumulative frequency 21 and function **find**:

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)

Time complexity: O(log MaxVal).
Code length: Up to twenty lines.

2D BIT

BIT can be used as a multi-dimensional data structure. Suppose you have a plane with dots (with non-negative coordinates). You make three queries:

1. set dot at (x , y)
2. remove dot from (x , y)
3. count number of dots in rectangle (0 , 0), (x , y) - where (0 , 0) if down-left corner, (x , y) is up-right corner and sides are parallel to x-axis and y-axis.

If **m** is the number of queries, **max_x** is maximum x coordinate, and **max_y** is maximum y coordinate, then the problem should be solved in O(m * log (max_x) * log (max_y)). In this case, each element of the tree will contain array - (**tree[max_x][max_y]**). Updating indexes of x-coordinate is the same as before. For example, suppose we are setting/removing dot (**a** , **b**). We will call **update(a , b , 1)/update(a , b , -1)**, where **update** is:

```
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);
        }
}
```

The function **updatey** is the "same" as function **update**:

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

It can be written in one function/procedure:

```
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);
        }
}
```

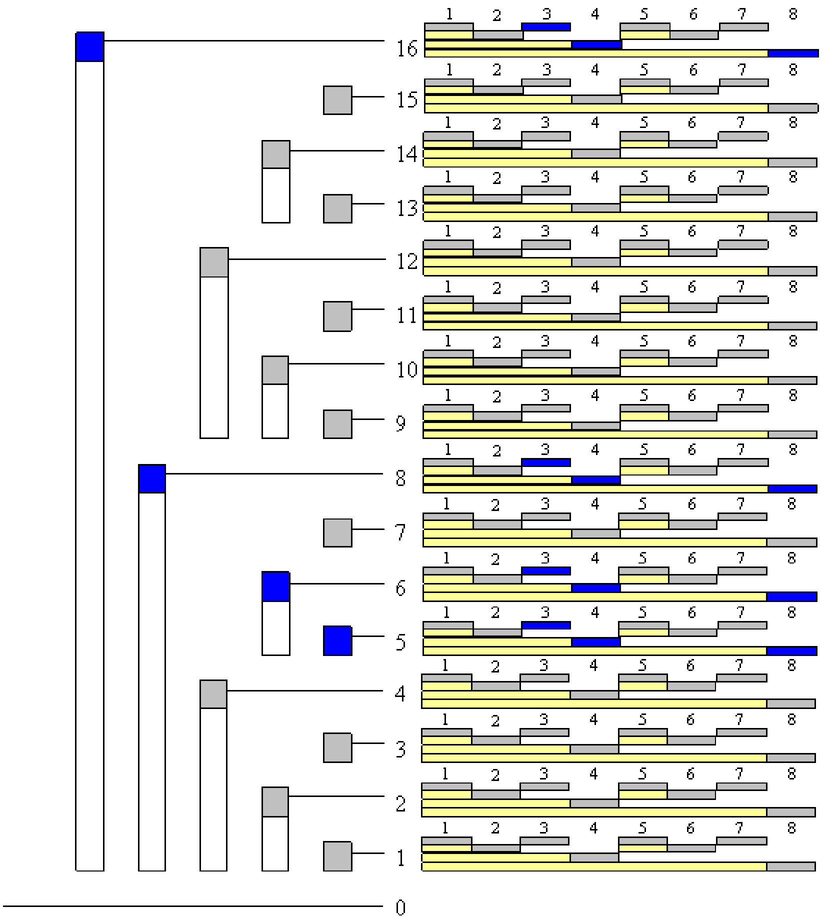

Image 1.8 - BIT is array of arrays, so this is two-dimensional BIT (size 16 x 8).
Blue fields are fields which we should update when we are updating index (5 , 3).

The modification for other functions is very similar. Also, note that BIT can be used as an n-dimensional data structure.

Sample problem

- SRM 310 - FloatingMedian
- Problem 2:
  Statement:
  There is an array of **n** cards. Each card is putted face down on table. You have two queries:
  1. T i j (turn cards from index i to index j, include i-th and j-th card - card which was face down will be face up; card which was face up will be face down)
  2. Q i (answer 0 if i-th card is face down else answer 1)

  Solution:
  This has solution for each query (and 1 and 2) has time complexity O(log n). In array **f** (of length **n + 1**) we will store each query **T (i , j)** - we set **f[i]++** and **f[j + 1]--**. For each card **k** between **i** and **j** (include **i** and **j**) sum **f[1] + f[2] + ... + f[k]** will be increased for 1, for all others will be same as before (look at the image 2.0 for clarification), so our solution will be described sum (which is same as cumulative frequency) module 2.

  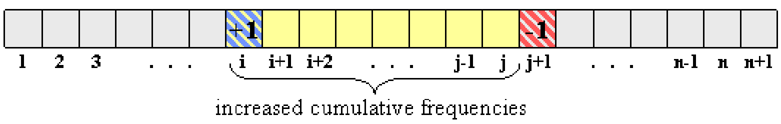

  Image 2.0

  Use **BIT** to store (increase/decrease) frequency and read cumulative frequency.

Conclusion

- Binary Indexed Trees are very easy to code.
- Each query on Binary Indexed Tree takes constant or logarithmic time.
- Binary Indexeds Tree require linear memory space.
- You can use it as an n-dimensional data structure.

References
[1] RMQ
[2] Binary Search
[3] Peter M. Fenwick

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)

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)

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]
```

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：

接下来让我们来看一个例子，当idx=5时：

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：

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)

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：

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：

```
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)

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)

```
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：

其中updatey函数与update函数是相似的：

以上两个函数可以整合成一个函数：

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。

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

- 15 Apr 2013 » 如何用Python写一个贪吃蛇AI

- 31 Mar 2013 » Pyglet教程
- 30 Mar 2013 » Linux Mint 12下的GLX问题
- 26 Mar 2013 » 动态规划：从新手到专家
- 14 Mar 2013 » Cracking the coding interview--问题与解答

上一篇 解题报告（A，B，C）
下一篇 如何用Python写一个贪吃蛇AI

主题推荐 解决方案 数据结构 动态规划 二分查找 二维数组

猜你在找

数据结构之动态规划之最优二叉查找树
数据结构C#--动态规划法解决两个字符串中寻找最长公
数据结构－用二维数组构造列表
数据结构和算法------有序数组和二分查找
韩顺平_轻松搞定网页设计html+css+javascript_第28讲
剑指offer二分查找二维数组
算法LeetCodeSearch a 2D Matrix二维数组的二分查找
CC++学院3二维数组二分查找法指针模块注射
二维数组的二分查找 解题报告
已排序二维数组中的二分查找

查看评论

暂无评论

您还没有登录,请[登录]或[注册]

* 以上用户言论只代表其个人观点，不代表CSDN网站的观点或立场