树状数组

- （BIT, Binary Indexed Trees）

ACM@USC

徐华杰

- 已知一个数组 **a[ ]**，要求完成下列操作：
- **1.**第 **i** 个元素**add data**；
- **2.**查询区间 **[x, y]** 内所有元素的**sum**；
- ……

如何解决？？

一道题目引发的血案…

- 使用原始数组 **a[i]** 存储数据的第 **i** 个元素。那么对于要求的 **2** 种操作：
- **1.**修改数据第 **i** 个元素：直接修改 **a[i]**。时间复杂度为 **O(1)**。
- **2.**查询区间 **[x, y]** 的元素之和：循环 **x** 至 **y** 累加。时间复杂度为 **O(n)**。

设计数据结构.1

- 使用累加数组 **sum[i]** 存储数据的前 **i** 个元素之和。那么对于要求的 **2** 种操作：
- **1.**修改数据第 **i** 个元素：修改 **sum[i] … sum[n]**。时间复杂度为 **O(n)**。
- **2.**查询区间 **[x, y]** 的元素之和：**sum[y]–sum[x-1]** 时间复杂度为 **O(1)**。

设计数据结构.2

数据结构	add data	sum[x , y]	……
a[ ]	O(1)	O(n)	……
sum[ ]	O(n)	O(1)	……

- 假如我们进行 **m** 次操作，最坏的情况下：
- 时间复杂度为 **O(n*m)**。
- 如果**(n,m<=100,000)**,显然这种方法是不行的？

设计数据结构.3

- 在方法 **2** 中，我们需要一个数组的前缀和 **sum[i] = a[1]+a[2]+ … +a[i]**。
- 不难发现，如果我们修改了任意一个 **a[i]**，**sum[i]**、**sum[i+1] … sum[n]**都会发生变化。
- 可以说：每次修改 **a[i]** 后，调整前缀和 **sum** 在最坏的情况下需要 **O(n)**的时间。

该用什么？.1

- 但方法 **2** 的思想已经给了我们启发。对于有关“区间”的问题，如果我们只在单个元素上做文章，可能不会有太大的收获。
- 但是如果对于这些数据元素进行合理的划分（如方法 **2** 将其化为 **n** 个前缀），然后对于整体进行操作，往往会有神奇的功效。

该用什么？.2

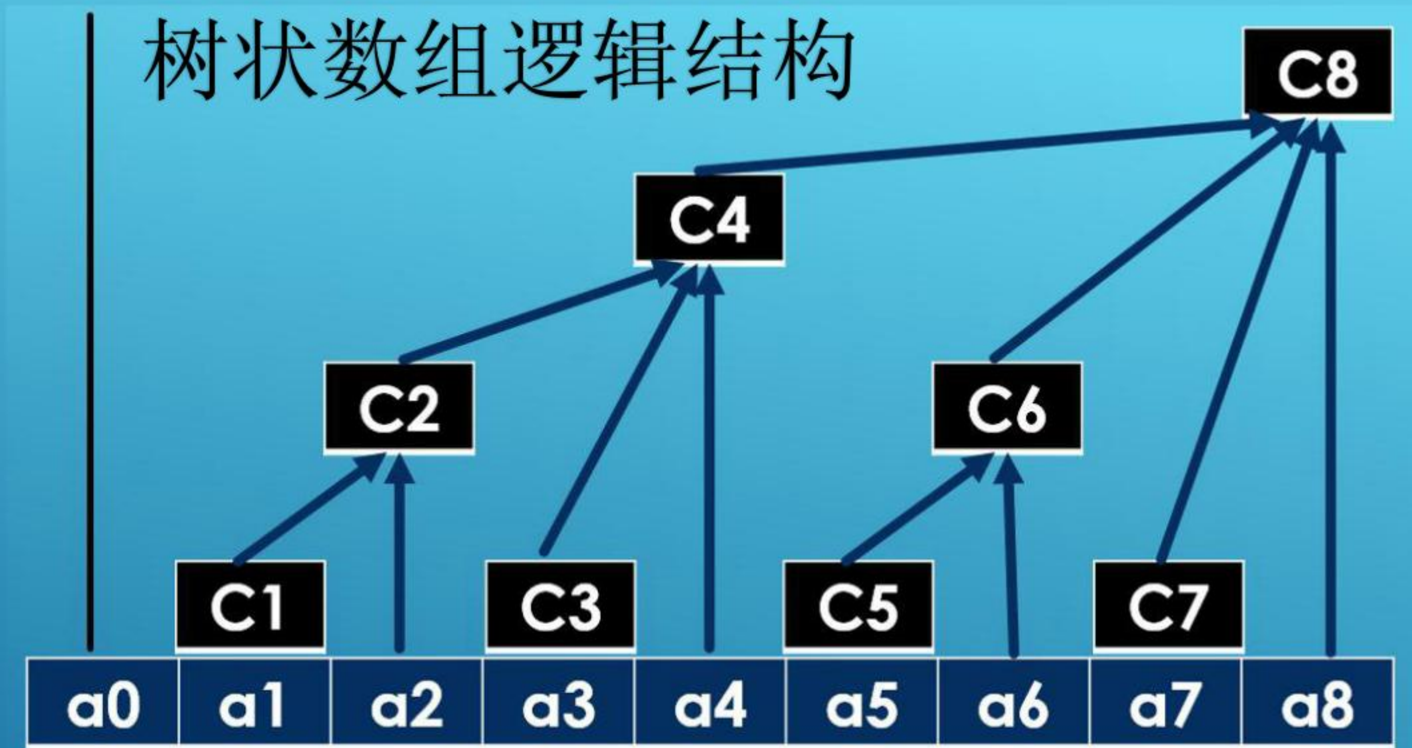
数据结构	add data	sum[x , y]	......
a[ ]	O(1)	O(n)	......
sum[ ]	O(n)	O(1)	......
BIT[ ]	O(log_{2^N})	O(log_{2^N})	......

- 假如我们进行 **m** 次操作，最坏的情况下：
-
- 时间复杂度为 **O(m*log_{2^N})**。

using - 树状数组

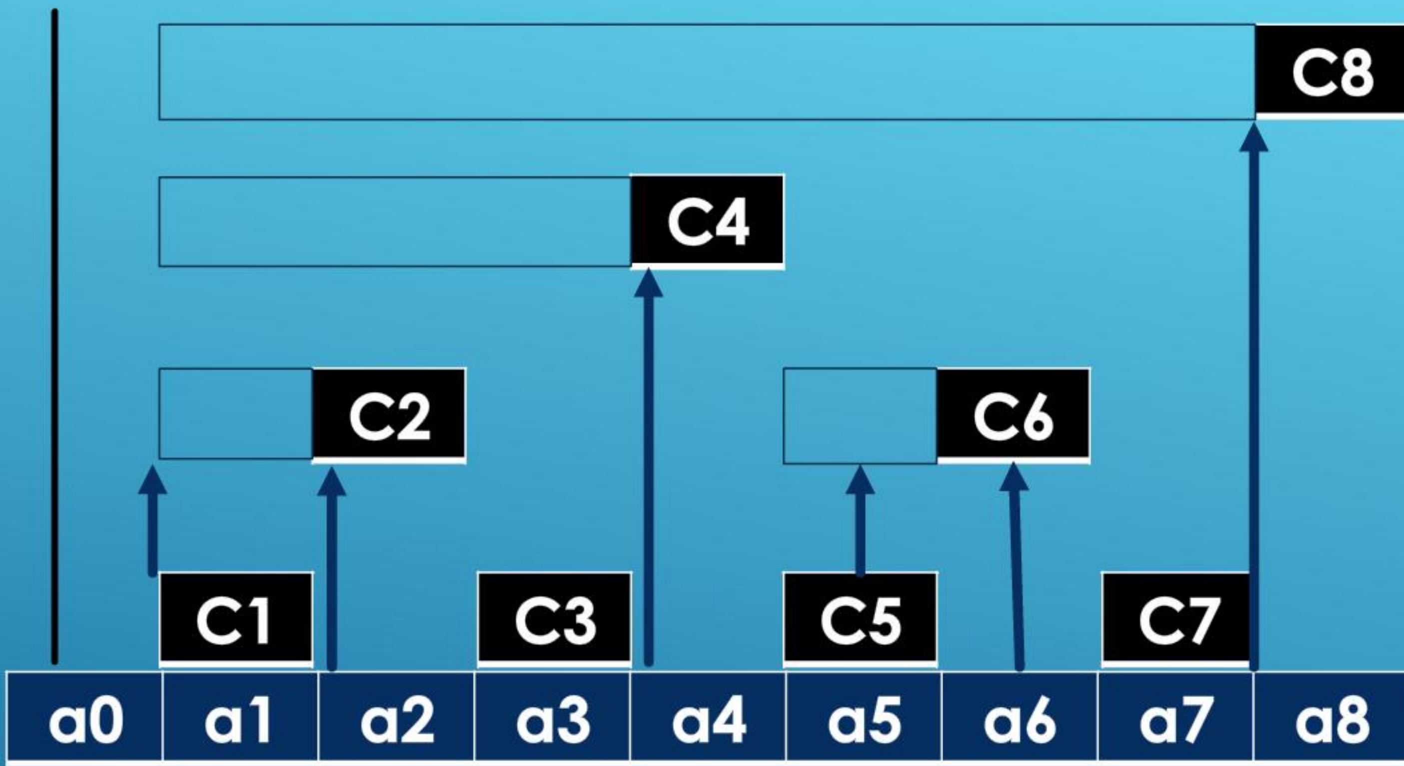
- 树状数组(**Binary Indexed Trees**)，最早用于数据压缩。
- 现在它常常被用于存储频率及操作累积频率表。
- 特性：信息更新 时间复杂度 **O**(log_{2^N})；
- 求和查询 时间复杂度 **O**(log_{2^N})；
- 定义：**C[ ]** (设为树状数组)

树状数组



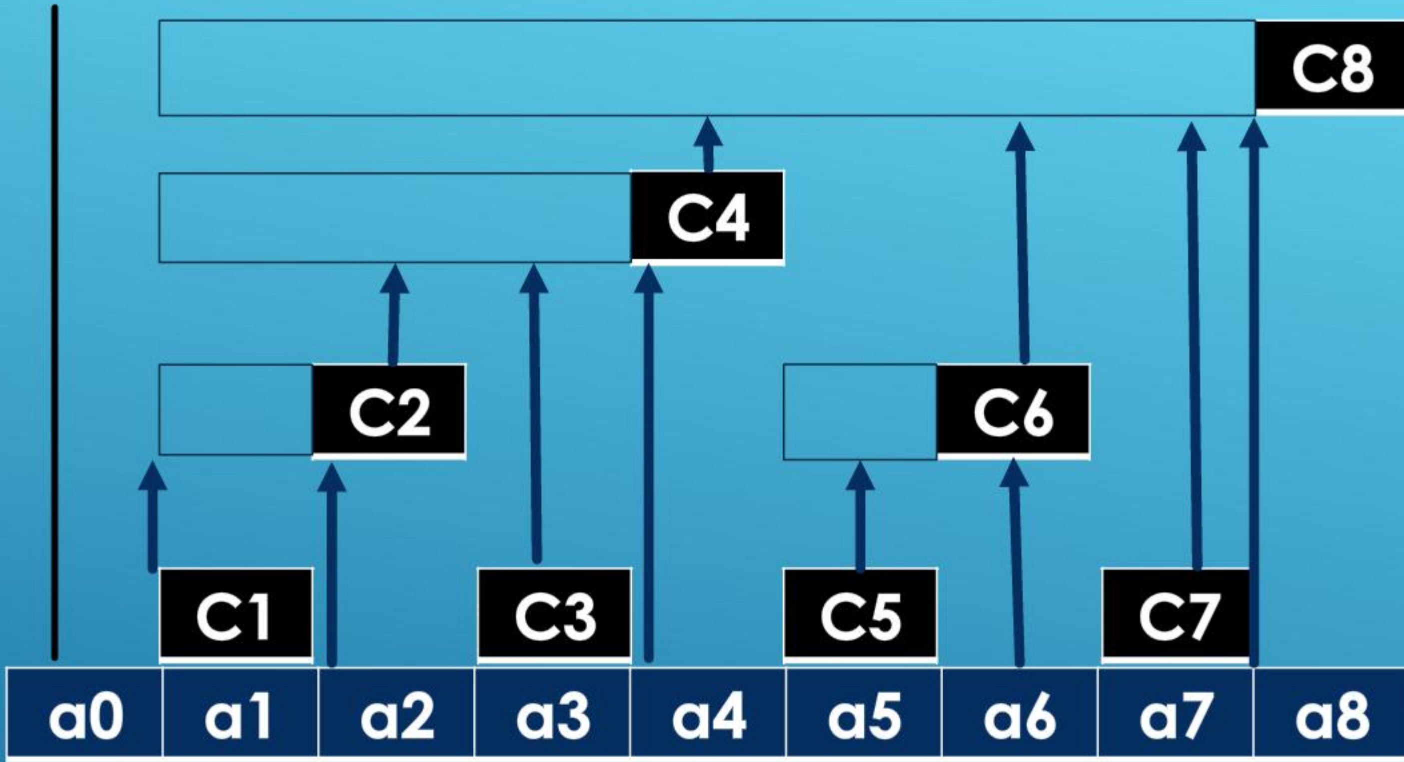
树状数组逻辑结构图

- 原始数组a与树状数组C的关系?



树状数组逻辑结构图

- 原始数组**a**与树状数组**C**的关系?



树状数组逻辑结构图

- 原始数组**a**与树状数组**C**的关系？

- C1 = a1
- C2 = a1 + a2
- C3 = a3
- C4 = a1+a2+a3+a4
- C5 = a5
- C6 = a5+a6
- C7 = a7
- C8 = a1+a2+...+a8

- C[i] = a[xx]+...+a[i] ?

找不到规律?

树状数组 与 原始数组.1

- C0001 = a0001

once again!

- C0010 = a0001 + a0010
- C0011 = a0011
- C0100 = a0001 + … + a0100
- C0101 = a0101
- C0110 = a0101 + a0110
- C0111 = a0111
- C1000 = a0001 + … + a1000

树状数组 与 原始数组.2

树状数组 与 原始数组.3

- C0001 = a0001
- C0010 = a0001 + a0010
- C0011 = a0011
- C0100 = a0001 + ... + a0100
- C0101 = a0101
- C0110 = a0101 + a0110
- C0111 = a0111
- C1000 = a0001 + ... + a1000

once again!

- **C[i]**中的初始位置**a[xx]**是:
- 去掉最低位的**'1'**后，加上**1**。
- 到 **a[i]** 的累加

- 起点的下标如何计算？
- 减去最右边的**'1'**，然后加上**1**。
- 例：**40=(00101000)2**
- **-40=(11011000)2**
- 所以：**40&(-40)=(001000)2**
- 于是，**C[40]**的起点下标**xx=40-[40&(-40)]+1**
- 所以：**C[i]**的起点下标**xx=i-[i&(-i)]+1**

- **lowbit(i)**

树状数组 与 原始数组.4

- 树状数组之所以高效简洁的原因就是能够利用位运算直接求出 i 对应的 **lowbit**

- **#define lowbit(x) ((x)&(-x))**
- **/* OR */**
- **Int lowbit(int i)   { return i&(-i); }**

树状数组 lowbit(x)

- 换个角度看：

- a[i]∈C[i], C[i=i+lowbit(i)], C[i=i+lowbit(i)], ...

- a[i]的值，在树状数组C中那些元素中？
- a[1]∈C[1], C[2], C[4], C[8], ...
- (a0001 ∈ C0001, C0010, C0100, C1000,...)
- a[3]∈C[3], C[4], C[8], ...
- (a0011 ∈ C0011, C0100, C1000, ...)

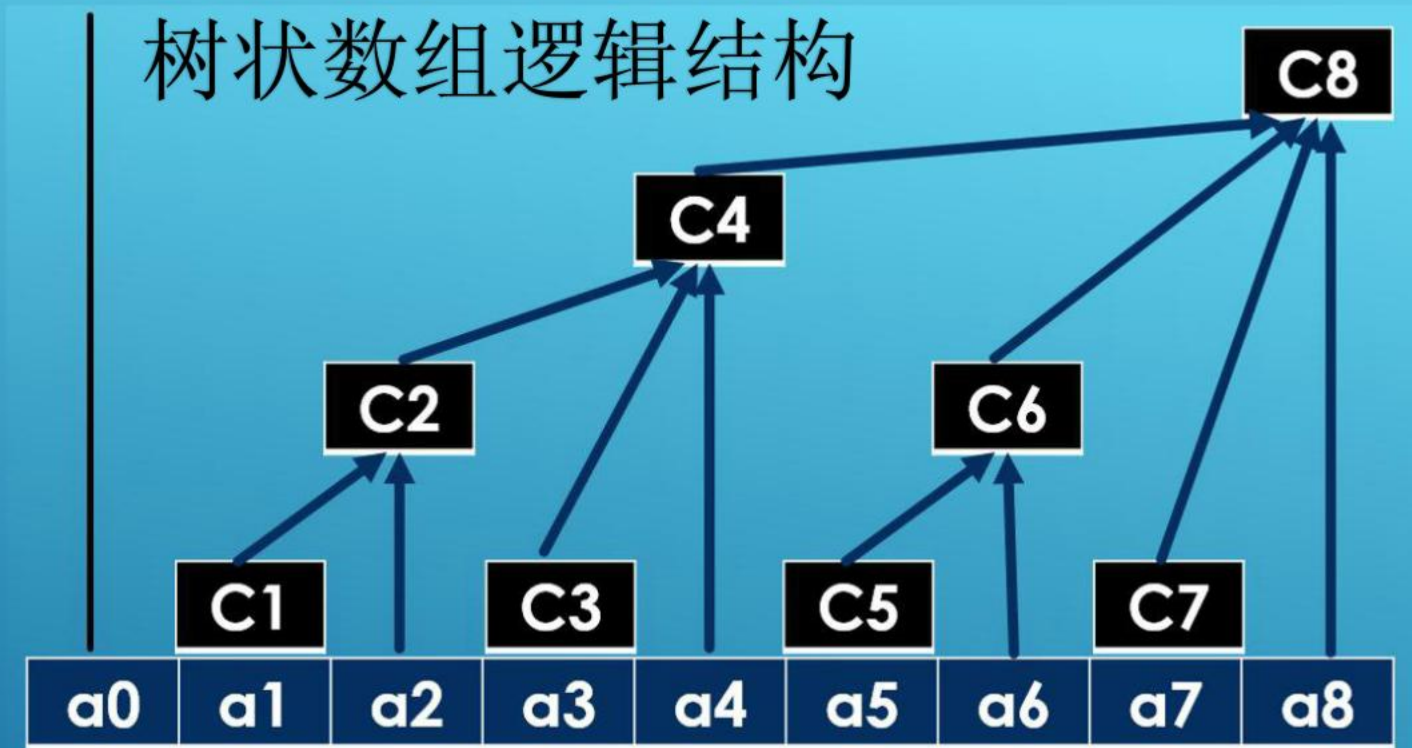
原始数组 与 树状数组

```
void update(int i, int data){
        while( i <= n ){
                C[i] += data;
                i += lowbit(i);
        }
} /* updata 数据插入 O( log_2^N ) */
```

树状数组 update(i, data)

- 由上，我们知道 **C[i] = a[i-lowbit(i)+1]+…+a[i]**。
- 以及**a[i]**∈**C[i]**, **C[i=i+lowbit(i)]**, **C[i=i+lowbit(i)]**, …
- 即：树状数组 **C** 与 原始数组 **a** 的关系。
- **Now**，我们需要知道：
- 累加数组 **sum** 与 树状数组 **C** 的关系。

树状数组的求和查询？



树状数组逻辑结构图

- sum1 = C1
- sum2 = C2
- sum3 = C3+C2
- sum4 = C4
- sum5 = C5+C4
- sum6 = C6+C4
- sum7 = C7+C6+C4
- sum8 = C8

- sum[i] = C[i]+…+C[yy]?

又找不到规律？

树状数组 与 累加数组.1

- sum0001 = C0001

once again too!

- sum0010 = C0010
- sum0011 = C0011+C0010
- sum0100 = C0100
- sum0101 = C0101+C0100
- sum0110 = C0110+C0100
- sum0111 = C0111+C0110+C0100
- sum1000 = C1000

树状数组 与 累加数组. 2

- sum0001 = C0001
- sum0010 = C0010
- sum0011 = C0011+C0010
- sum0100 = C0100
- sum0101 = C0101+C0100
- sum0110 = C0110+C0100
- sum0111 = C0111+C0110+C0100
- sum1000 = C1000

once again too!

- sum[i]=C[i]+
- C[i=i-lowbit(i)]+
- C[i=i-lowbit(i)]+
- …

树状数组 与 累加数组. 3

```
int query(int i){
        int ans = 0;
        while( i>0 ){
                ans += C[i];
                i -= lowbit(i);
        }       return ans;
} /* query 求和查询 O( log_2N ) */
```

树状数组 query(i)

树状数组的多维形式

- **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**)**。
- 你有两种查询方式：
- **I.**将点 **(x, y)** 的数值增加 **data**
- **II.**计算左下角为 **(0, 0)** 右上角为 **(x, y)** 的矩形内的数值总和
- 【设：所有数据**(x, y)**满足**(x<=n && y<=m)**。
- 二维树状数组为 **C[n+1][m+1]** 】

```
void update(int x, int y, int data){
        for( ; x <= n; x += lowbit(x) )
                for(int j=y; j <= m; j += lowbit(j) )
                        C[x][j] += data;
} /* updata 数据插入 O( log_2^N ) */
```

二维树状数组.1

```
int query(int x, int y){
        int ans = 0;
        for( ; x > 0; x -= lowbit(x) )
                for(int j=y; j > 0; j -= lowbit(j) )
                        ans += C[x][j];
        return ans;
} /* query 求和查询 O( log_2N ) */
```

二维树状数组.2

- 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 $O(\log_2 N)$ 的时间
-
- 树状数组需要线性的存储空间 **O(n)**，只维护 C 数组
- 树状数组可以扩展成 **n 维**的情况

树状数组 - 总结

- 1. 树状数组和线段树一样，是一种很高效的数据结构。它的优势在于空间耗费较小。
  - 【对于长度在 n 的线段(区间)，线段树需要 2n 的空间，而树状数组只需要 n 即可。】
- 2. 树状数组的另一个优点是编程简单。
  - 【只要开一个数组，所有的操作都可以在10行之内实现。没有了繁琐的指针和乱七八糟的二分，编写这样的程序不容易出错。】

树状数组PK线段树-优

- **1.** 树状数组的致命缺点是无法记录一些附加信息。
  - 【比如“不相交区间的个数”，就无法用树状数组维护。】
- **2.** 树状数组的应用范围是很窄的。
  - 【求和的问题可以用树状数组。】
  - 【如果求最大值操作，而且没有删除操作的话，那也能够用树状数组。】

树状数组PK线段树-劣

- 如果还有不懂的，可以浏览 **Hawstein** 的博客（有一篇翻译的树状数组资料）。
- http://hawstein.com/posts/binary-indexed-trees.html
- 如果你的英语好，可以浏览英文原版
- （作者：**boba5551**）
- http://community.topcoder.com/tc?module=Static&d1=tutorials&d2=binaryIndexedTrees

END