树形基础数据结构的应用

安徽师大附中　　叶国平

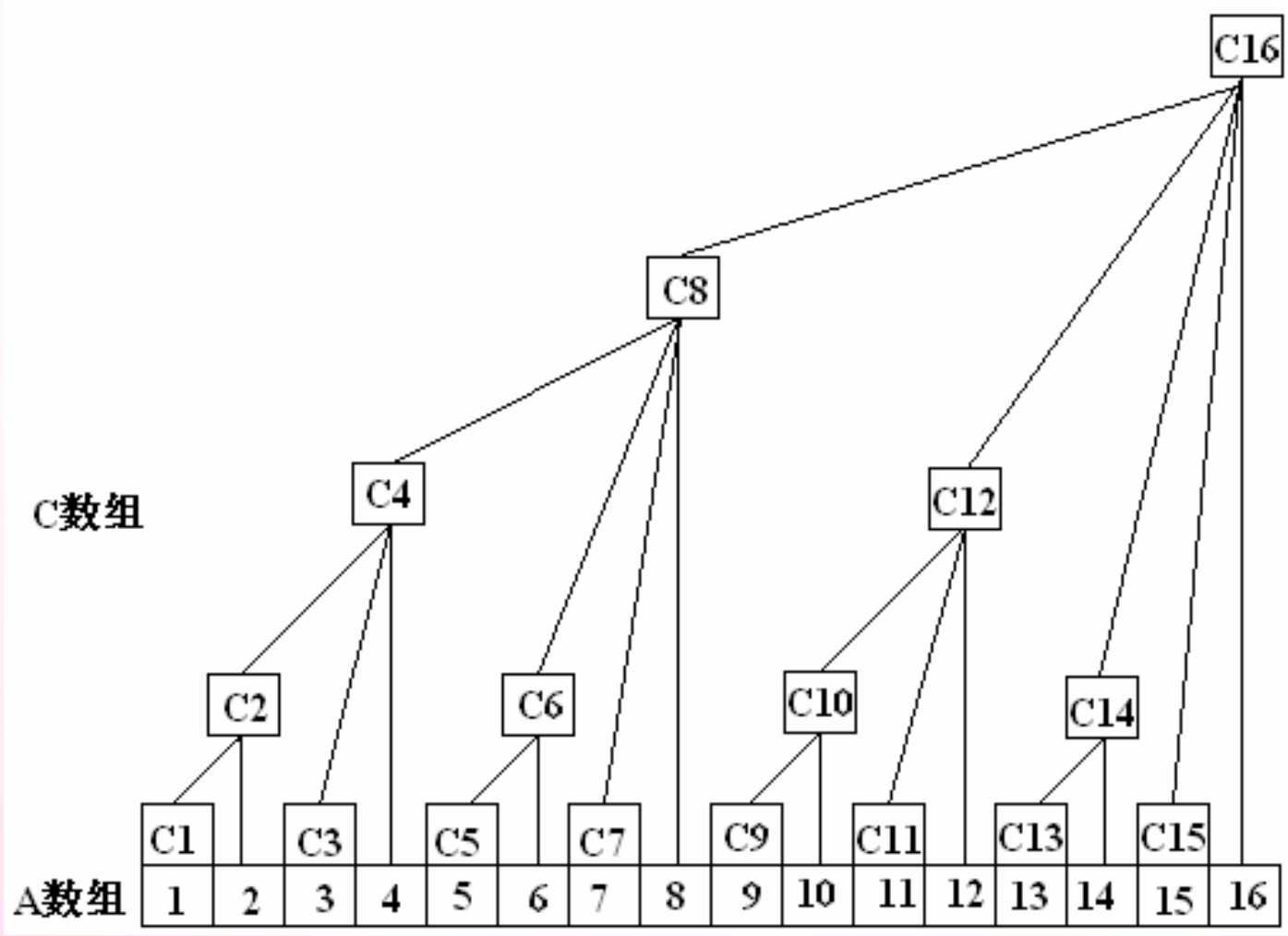
树状数组

线段树的时空效率都非常高—时间复杂度$O(\log^n)$。空间复杂度为O(n)，但系数却很大。比如线段树的每个结点都存两个指针，指向左右孩子，还有别的信息等等。对于长度为n的线段，线段树有2n个结点。种种这些都加重了内存的负担。如果题目的空间有严格限制，那么使用线段树就会很有可能会丢分。

树状数组

树状数组是一种特殊的高效数据结构：时空复杂度和线段树类似。但是它的系数要小得多。树状数组是一个查询和修改复杂度都为log(n)的数据结构，假设数组a[1]...a[n]，那么查询a[1]+a[2]+...+a[i]和的时间是log级别的，而且是一个在线的数据结构，支持随时修改某个元素a[i]的值，复杂度也为log级别。

树状数组


C16
C8
C4
C12
C数组
C2
C6
C10
C14
C1
C3
C5
C7
C9
C11
C13
C15
A数组
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16

树状数组

容易发现：

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[16]

树状数组

可以推出：

- C[i]有2^k(k是i的二进制数末尾0的个数)个元素;

算2^k有一个很快捷的方法，定义一个如下函数即可：

```
int  lowbit(int  i){
return i&(i^(i-1)); //等价于return i&(-i);
}
```

- C[i]最后一个元素必是A[i];
- 所以：C[i]=A[i-2^k+1]+…+A[i]

树状数组

- 当想要查询一个sum(n),即查询a[1]+a[2]+…+a[n]的和时，可以依据如下算法即可：

  step1: 令tot = 0，转第二步；

  step2: 假如n <= 0，算法结束，返回tot值，
  否则tot = tot + c[n]，转第三步；

  step3: 令n = n – lowbit(n)，转第二步。

- 可以看出，这个算法就是将这一个个区间的和全部加起来，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。

树状数组

- 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。 所以修改算法如下（给某个结点i加上x）：

  step1: 当i > n时，算法结束，否则转第二步；

  step2: C[i] = C[i] + x， i = i + lowbit(i)转第一 步。

- i = i +lowbit(i)这个过程实际上也只是一个把末尾1补为0(在末尾1上加1)的过程。

- **特别注意：** 树状数组的下标不能从0开始，因为lowbit(0)是死循环。

树状数组

- 整个程序如下：

```
int lowbit(int t)
{return t&(t^(t-1));
}
int sum(int end)
{int tot=0;
  while(end>0)
   {tot+=c[end];
    end-=lowbit(end);
   }
  return tot;
}
```

```
void modify(int pos,int num)
{while(pos<=n)
    {c[pos]+=num;
     pos+=lowbit(pos);
    }
}
```

- 初始化

```
memset(c,0,sizeof(c));
for (i=1;i<=n;i++)
    modify(i,a[i]);
```

应用一：单点修改、区间求和

- 题目描述:

已知一个数列，你需要进行下面两种操作：

1. 将某一个数加上k;

2. 求出某区间每一个数的和。

操作1： 格式：1 x k 含义：将第x个数加上k;

操作2： 格式：2 x y 含义：输出区间[x, y]内每个数的和。

例题：求逆序对

- 题目描述:

  输入一个长度为n的序列，求序列中逆序对的个数。 逆序对即 Ai>Aj（i<j）

- 数据范围：n≤150000，0≤Ai≤10^8。

- **例如：**

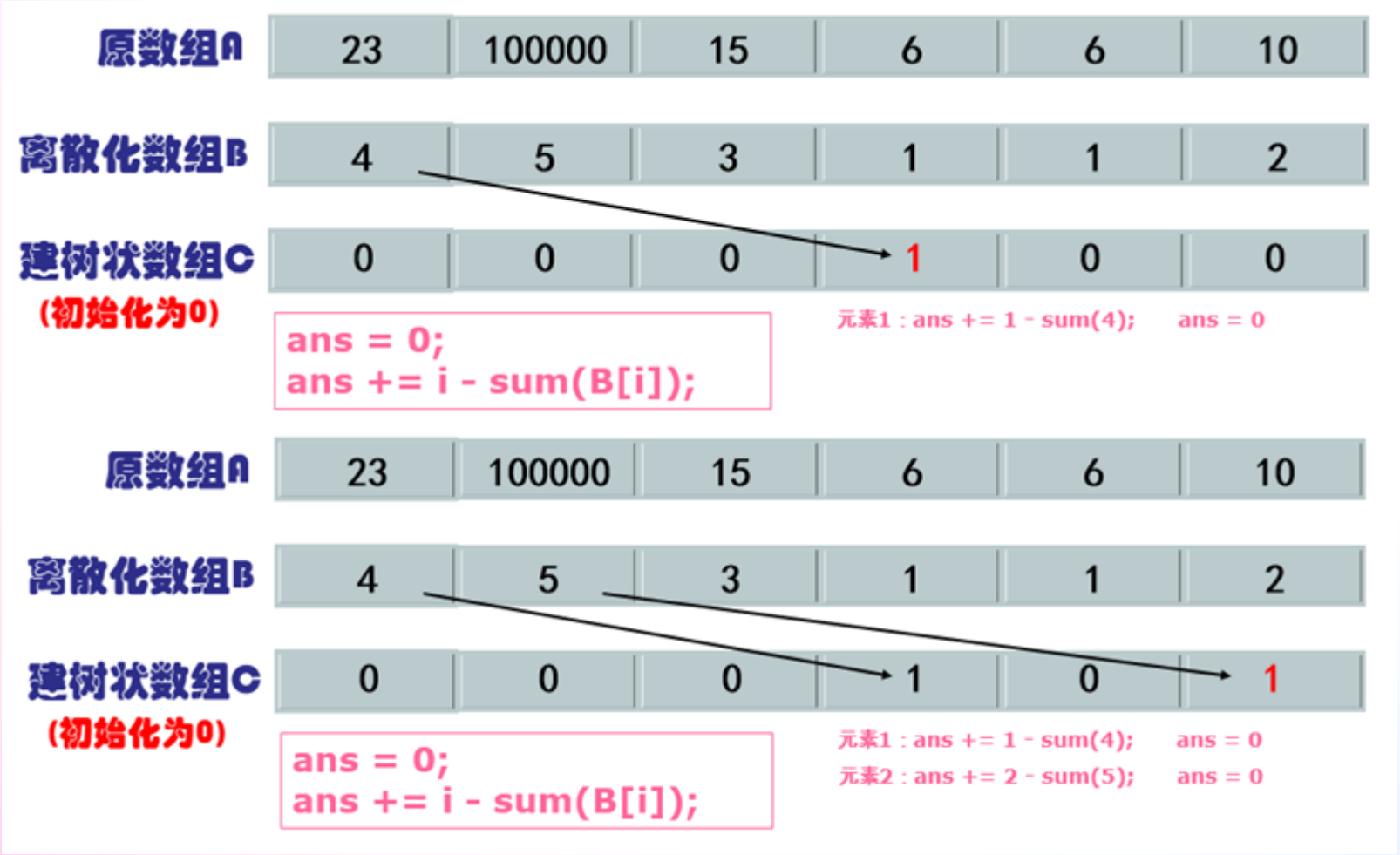
  n=6

  23 10^5 15 6 6 10

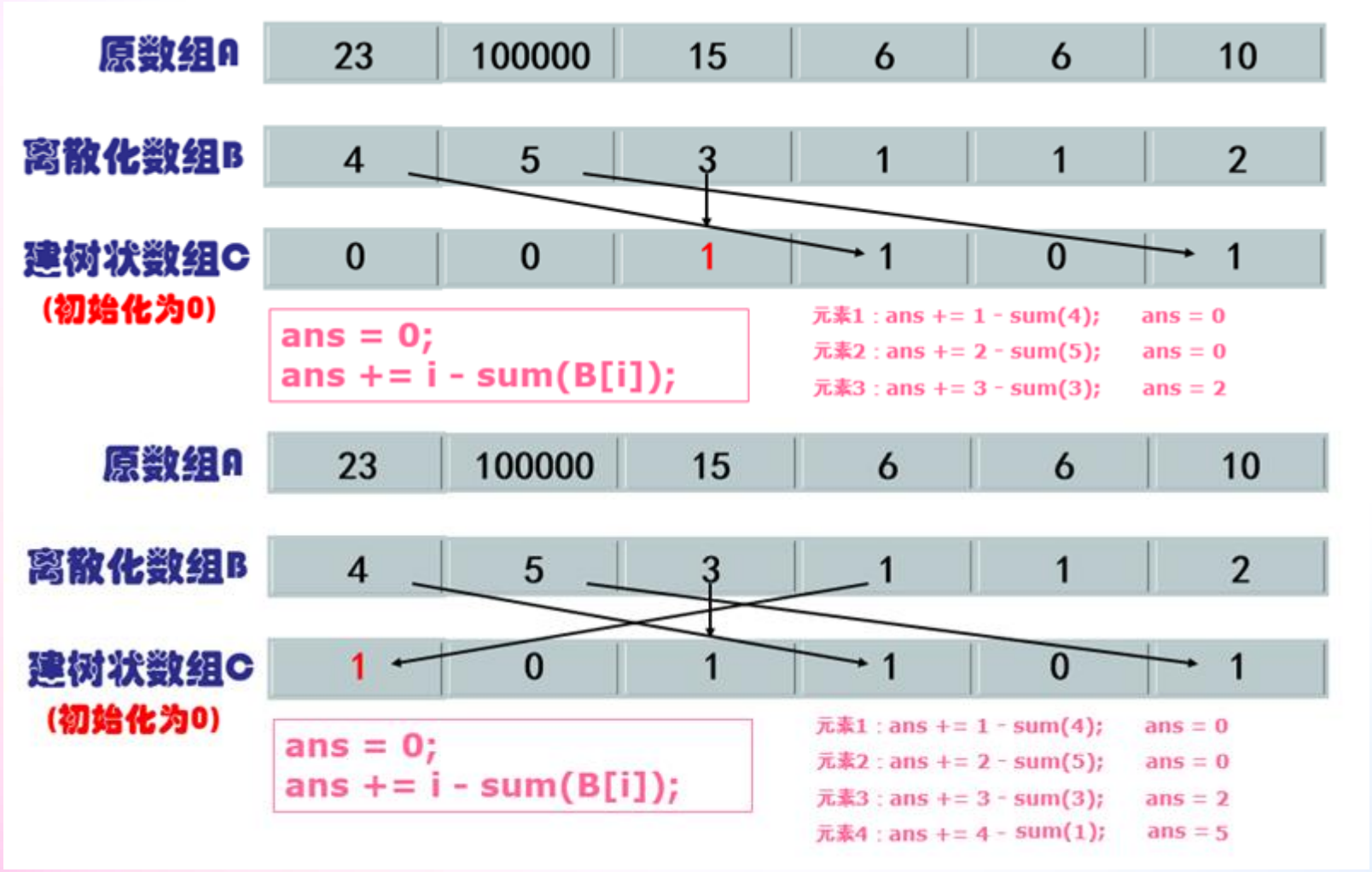
例题：逆序对

- 先将A[ ]离散化，我们可以先开一个大小为n的数组C[ ]，每当读入一个数A[i]，假设离散化后为X，将C[X]加上1，然后我们统计C[1]~C[X]的和sum，sum - 1（除掉这个数本身）就是在这个数前面有多少个数比它小。我们只要用i-sum就可以得出前面有多少数比它大，也就是这个数与前面的数构成了逆序对的数量。

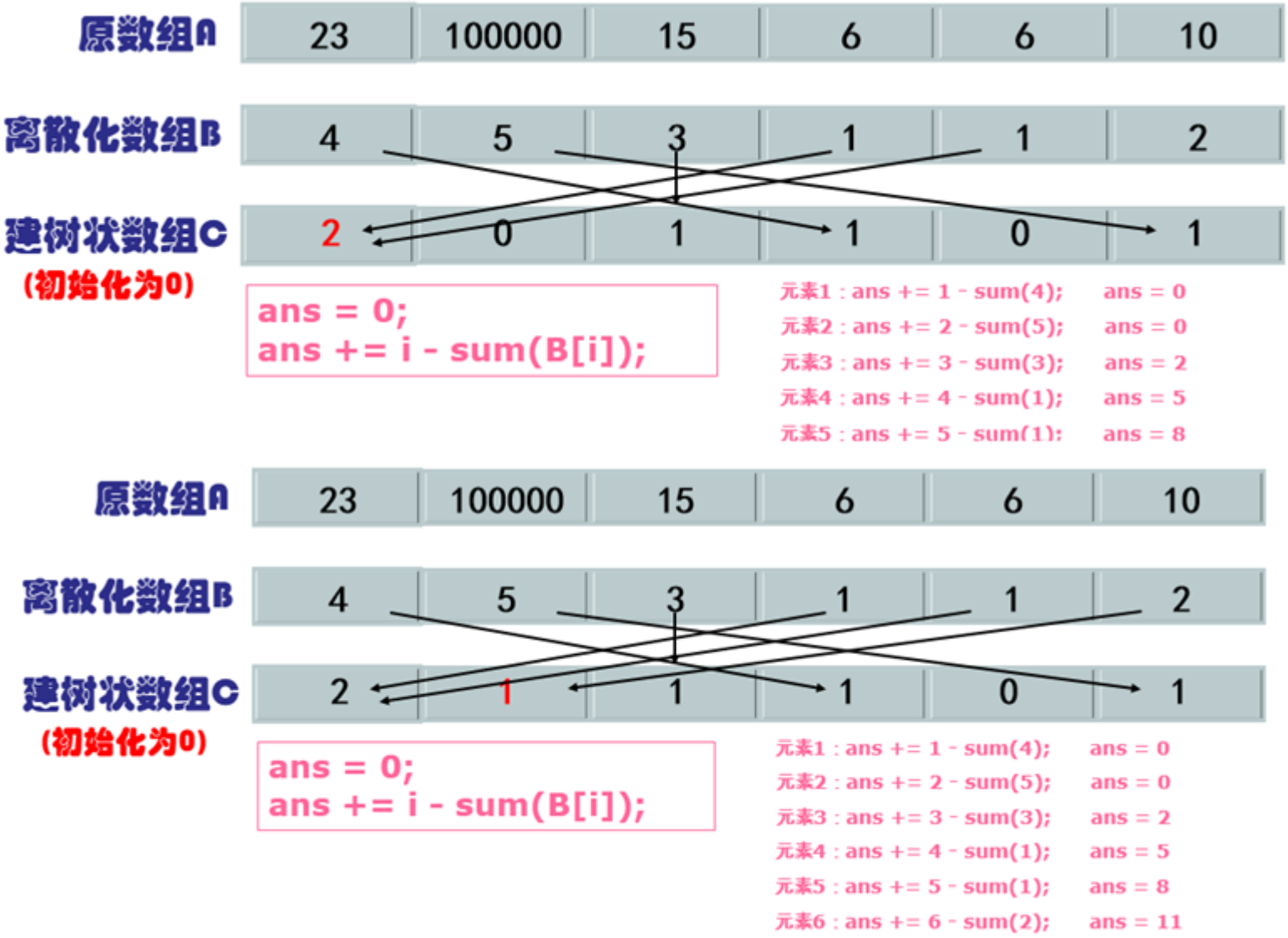
例题：逆序对


原数组A
23
100000
15
6
6
10
离散化数组B
4
5
3
1
1
2
建树状数组C
(初始化为0)
0
0
0
1
0
0
ans = 0;
ans += i - sum(B[i]);
元素1 : ans += 1 - sum(4); ans = 0
原数组A
23
100000
15
6
6
10
离散化数组B
4
5
3
1
1
2
建树状数组C
(初始化为0)
0
0
0
1
0
1
ans = 0;
ans += i - sum(B[i]);
元素1 : ans += 1 - sum(4); ans = 0
元素2 : ans += 2 - sum(5); ans = 0

例题：逆序对


原数组A
23
100000
15
6
6
10
离散化数组B
4
5
3
1
1
2
建树状数组C
(初始化为0)
0
0
1
1
0
1
ans = 0;
ans += i - sum(B[i]);
元素1 : ans += 1 - sum(4); ans = 0
元素2 : ans += 2 - sum(5); ans = 0
元素3 : ans += 3 - sum(3); ans = 2
原数组A
23
100000
15
6
6
10
离散化数组B
4
5
3
1
1
2
建树状数组C
(初始化为0)
1
0
1
1
0
1
ans = 0;
ans += i - sum(B[i]);
元素1 : ans += 1 - sum(4); ans = 0
元素2 : ans += 2 - sum(5); ans = 0
元素3 : ans += 3 - sum(3); ans = 2
元素4 : ans += 4 - sum(1); ans = 5

例题：逆序对


原数组A
23
100000
15
6
6
10
离散化数组B
4
5
3
1
1
2
建树状数组C
(初始化为0)
2
0
1
1
0
1
ans = 0;
ans += i - sum(B[i]);
元素1 : ans += 1 - sum(4); ans = 0
元素2 : ans += 2 - sum(5); ans = 0
元素3 : ans += 3 - sum(3); ans = 2
元素4 : ans += 4 - sum(1); ans = 5
元素5 : ans += 5 - sum(1); ans = 8
原数组A
23
100000
15
6
6
10
离散化数组B
4
5
3
1
1
2
建树状数组C
(初始化为0)
2
1
1
1
0
1
ans = 0;
ans += i - sum(B[i]);
元素1 : ans += 1 - sum(4); ans = 0
元素2 : ans += 2 - sum(5); ans = 0
元素3 : ans += 3 - sum(3); ans = 2
元素4 : ans += 4 - sum(1); ans = 5
元素5 : ans += 5 - sum(1); ans = 8
元素6 : ans += 6 - sum(2); ans = 11

应用二：区间修改、单点求值

- 题目描述:

已知一个数列，你需要进行下面两种操作：

1.将某区间每一个数数加上k

2.求出该数列某个数的值

操作1： 格式：1 x y k 含义：将区间[x,y]内每个数加上k

操作2： 格式：2 k 含义：输出第k项的值。

应用二：区间修改、单点求值

- 首先设定一个a[ ]，我们能得到一个与之相对应的差分数组b[ ]，使b[i]=a[i]-a[i-1]，简单可证，a[i]=b[1]+b[2]+....+b[i]
- 设将a[i]…a[j]这个区间每个数+k，我们只需将b[i]+k，b[j+1]-k即可
- 查询a[i], 因a[i]=b[1]+b[2]+....+b[i],b[1]+…+b[i]求和太慢，于是我们搞一个树状数组优化一下。

应用三：区间修改、区间求和

- 题目描述:

已知一个数列，你需要进行下面两种操作：

1. 将某区间每一个数数加上d

2. 求出该数列某个区间所有数的和

操作1： 格式：update(s, t, d)把区间a[s]...a[t]都增加d

操作2： 格式：query(s, t)，求a[s]...a[t]的区间和。

应用三：区间修改、区间求和

- 首先，看更新操作update(s, t, d)把区间a[s]...a[t]都增加d，我们引入一个数组delta[i]，表示a[i]...a[n]的共同增量，n是数组的大小。那么update操作可以转化为：
  - 1)令delta[s] = delta[s] + d，表示将a[s]...a[n]同时增加d，但这样a[t+1]...a[n]就多加了d;
  - 2)再令delta[t+1] = delta[t+1] - d，表示将a[t+1]... a[n]同时减d。

应用三：区间修改、区间求和

- 然后来看查询操作query(s，t)，求a[s]...a[t]的区间和，转化为求前缀和，设sum[i]=a[1]+..+a[i]，则a[s]+...+a[t] = sum[t] - sum[s-1]
- 那么前缀和sum[x]又如何求呢？它由两部分组成，一是数组的原始和，二是该区间内的累计增量和，把数组a的原始值保存在数组org中，并且delta[i]对sum[x]的贡献值为delta[i]*(x+1-i).

应用三：区间修改、区间求和

- sum[x]=org[1]+..+org[x]+delta[1]*x+delta[2]*(x-1)+delta[3]*(x-2)+...+delta[x]*1

  $= \sum(org[i]) + \sum(delta[i]*(x+1-i))$

  $= \sum(org[i]) + (x+1)*\sum(delta[i]) - \sum(delta[i]*i)$

  $= \sum(org[i]-delta[i]*i) + (x+1)* \sum delta[i] (1 \le i \le x)$

- 这其实就是三个数组org[i]，delta[i]和delta[i]*i的前缀和，org[i]的前缀和保持不变，事先就可以求出来，delta[i]和delta[i]*i的前缀和是不断变化的，可以用两个树状数组来维护。

多维树状数组

- 一维的树状数组的每个操作的时间复杂度都是O(logn)，非常的高效。它可以扩充为m维，这样每个操作的复杂度就变成了$O(\log^m n)$，在m不大的时候，时间复杂度是可以接受的。扩充的方法就是将原来的修改和查询函数中的一个循环，改成m个循环m维数组c[]中的操作。例如：有n×m的二维数组a[]，树状数组为c[]，那么修改和查询操作为：

树状数组

- 将(x,y)的值加上z

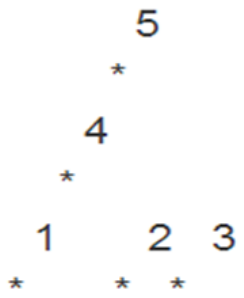
```
void modify(int x, int y, int z)
 {int i=x;
  while(i<=n)
     {int j=y;
       while (j<=m)
           {c[i][j]+=z;
            j+=lowbit(j);
           }
       i+=lowbit(i);
     }
 }
```

- 求(1,1)到(x,y)的和

```
int sum(int x, int y)
{int res=0, i=x;
  while(i>0)
    {j=y;
      while (j>0)
          {res+=c[i][j];
           j-=lowbit(j);
          }
     i-=lowbit(i);
    }
  return res; }
```

例题：Pku2352 Stars

- 天文学家经常要检查星星的地图，每个星星用平面上的一个点来表示，每个星星都有坐标。我们定义一个星星的“级别”为给定的星星中不高于它并且不在它右边的星星的数目。天文学家想知道每个星星的“级别”



- 例如上图，5号星的“级别”是3（1，2，4这三个星星），2号星和4号星的“级别”为1。给你一个地图，你的任务是算出每个星星的“级别”

例题：Pku2352 Stars

- 输入的第一行是星星的数目N(1≤N≤60000),接下来的N行描述星星的坐标(每一行是用一个空格隔开的两个整数X,Y(0≤X,Y≤32000)。星星的位置互不相同。星星的描述按照Y值递增的顺序列出，Y值相同的星星按照X值递增的顺序列出。
- 输出包含N行，一行一个数。第i行是第i个星星的“级别”

例题：Pku2352 Stars

- 此题为最简单的利用树状数组进行统计计数，注意星座下标可能为0，但树状数组中下标不能为0，所以要所有星座整体右移一位。
- 因为星星的位置互不相同且星星的描述按照Y值递增的顺序列出，Y值相同的星星按照X值递增的顺序列出，所以统计星星的级别只需统计其左边的星星个数即可。
- 具体做法：读入(x, y), 将x++, query(x), add(x)即可。

star.cpp

例题：Balanced Photo

- FJ正在安排他的N($N \le 10^5$)头奶牛站成一排来拍照。序列中的第i头奶牛的高度是h[i]($h[i] \le 10^9$)，且序列中所有的奶牛的身高都不同。
- 他认为，每头牛i左边比他高的牛的数量记为L[i]，右边比他高的牛的数量记为R[i]，如果存在i满足max(R[i], L[i])>2*min(L[i], R[i])则这个牛i是不平衡的，FJ不希望他有太多的奶牛不平衡。
- 请帮助FJ计算不平衡的奶牛数量。

例题：Balanced Photo

- 做法一：将奶牛的高度进行离散化，按位置从左往右依次处理每一头奶牛，设处理到第i头奶牛，开两个树状数组T1, T2，分别记录奶牛i左边和右边奶牛的身高信息（即以身高为下标，1/0表示该身高奶牛是否存在），具体操作：
  - add(T1,h[i],1); add(T2,h[i],-1);
  - L[i]=i-query(T1,h[i]); R[i]=N-i-query(T2,h[i]);
  - if (max(L[i],R[i])>2*min(L[i],R[i])) ans++;

photo.cpp

例题：Balanced Photo

- 此题不是按牛的高度进行加入，而是先将高度排序从大到小，然后按位置进行加入，进行统计。例如:
- 高度　　位置

　　5　　3

　　4　　5

- 先将高度为5的牛，在bit的3号位置加入。然后在统计4这个牛时，发现在位置4之间已加入一头牛，于是就有一头牛比它高了。
- bit是按值域进行划分的，这个值就多样性了，要根据题意来进行合理的选取。

photo.cpp

例题：奶牛抗议

- 约翰家的N（1≤N≤10^5)头奶牛正在排队游行抗议。一些奶牛情绪激动，约翰测算下来，排在第i位的奶牛的理智度为A[i](-10^5≤A[i]≤10^5)。约翰希望奶牛在抗议时保持理性，为此，他打算将这条队伍分割成几个小组，每个抗议小组的理智度之和必须大或等于零。奶牛的队伍已经固定了前后顺序，所以不能交换它们的位置，所以分在一个小组里的奶牛必须是连续位置的。除此之外，分组多少组，每组分多少奶牛，都没有限制。约翰想知道有多少种分组的方案 mod 10^9+9 的余数即可。

例题：奶牛抗议

- 设前缀和s[i]=$\sum_{j=1}^{i} A[j]$，f[i]表示到第i头奶牛结束满足条件的分组方案数，则转移方程为：f[i]=$\sum_{j=0}^{i-1} f[j]$，当s[i]-s[j]≥0时，其中f[0]=1。
- 这是0(n^2)的时间复杂度。怎么办？
- 关键是对dp条件式的转化 s[i]-s[j]≥0，即s[i]≥s[j],也就是求比s[i]小的数有多少个，我们可以离散化s数组，然后用树状数组或线段树来求和。

protest.cpp

例题：移动电话

- **问题描述**：假设第四代移动电话的收发站是这样工作的。整个区域被分割成很小的方格。所有的方格组成了一个S*S的矩阵，行和列都是从0~S-1编号。每个小方格都包含一个收发站。每个方格内的开机的移动电话数量可以不断改变，因为手机用户在各个方格之间移动，也有用户开机或关机。一旦某个方格里面开机的移动电话数量发生了变化，该方格里收发站就会向总部发送一条信息说明这个改变量。

老板可能随时会问：某个给定矩形区域内有多少部开机的移动电话呀？要求你编写一个程序能随时回答老板的问题。

例题：移动电话

- **输入格式：** 输入数据的格式如下：

指示数	参数	意义
0	S	初始命令，整个区域由S*S个小格子组成指令只会在一开始出现一次。
1	X Y A	方格(X,Y)内的开机移动电话量增加了A。A可能是正数也可能是负数。
2	L B R T	询问在矩形区域(L,B)到(R,T)内有多少部开机的移动电话。
3		终止程序。这个指示只会在最后出现一次。

- **输出格式：** 对于除了2之外的提示，你的程序不应该输出任何东西。如果指示是2，那么你程序应该向输出一个整数。

例题：移动电话

- **输入输出样例**

mobile.in	mobile.out
0 1 1 2 3 2 0 0 2 2 1 1 1 2 1 1 2 -1 2 1 1 2 3 3	3 4

例题：移动电话

- **数据限制：**

区域大小	S*S	1*1<=S*S<=1024*1024
每个格子的值	V	0<=V<=32767
增加/减少量	A	-32767<=A<=32767
指令总数	U	3<=U<=60002
所有各自的总和	M	M=2147483647

例题：移动电话

- 分析：
  - 这是二维扩展的情况。
  - 首先为了方便起见，初始化时我们就让格子从1开始编号。
  - a[x][y]——存储的是第x行第y列的小格子内开机的移动电话的数量。
  - c[x][y]——存储的是a[x-2^kx+1][y-2^ky+1]到a[x][y]中的数字和（kx,ky分别为x,y二进制形式下末尾0的个数）
  - 由于树状数组c中x轴和y轴相对独立，因此可按照处理一维数组的办法，由列及行地更改c中的相应元素。

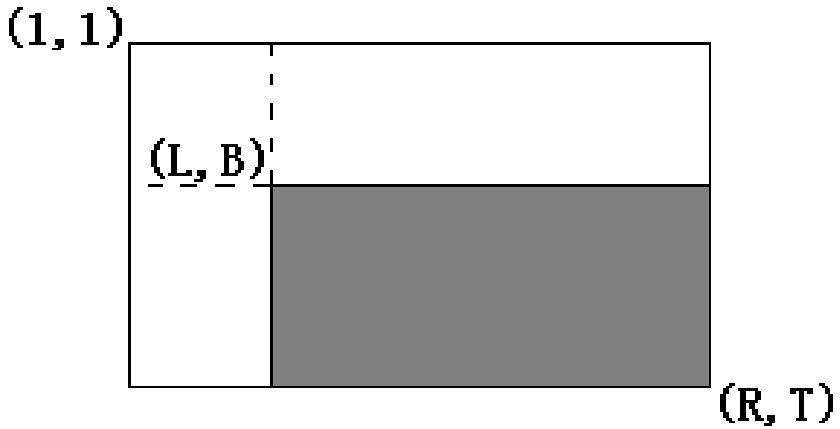
例题：移动电话

- 计算a添加到小格子(x, y)内后的树状数组c：

```
void update(int x,int y,int a)
{int iy;
  while (y<=n)//由左向右更新
     {ix=x;//由上而下更新y轴的数据
       while (ix<=n)
           {c[ix][y]+=a;
            ix+=lowbit(ix);//计算下一个被更新的行位置
           }
      y+=lowbit(y);//计算下一个被更新的列位置
     }
}
```

例题：移动电话

- 统计从(L, B)到(R, T)的矩形总和：



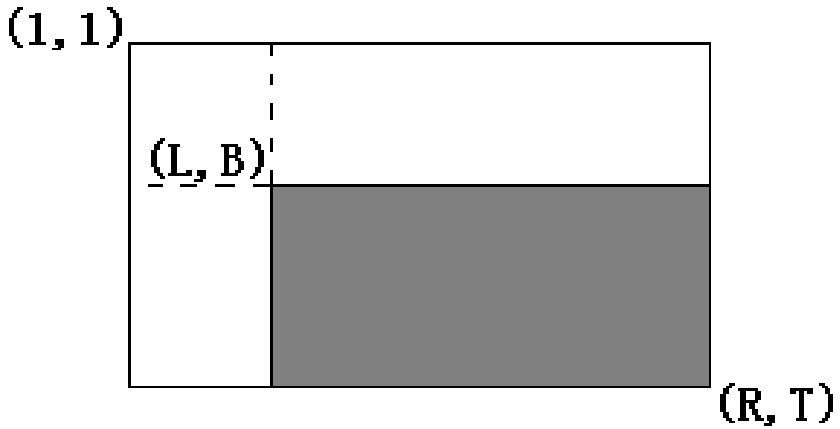
- 从上图可知：

$$S(L,B)(R,T)=S(1,1)(R,T)-S(1,1)(R,B-1)-S(1,1)(L-1,T)+S(1,1)(L-1,B-1)$$

- 这个操作的复杂度为$O(\log^n)^2$,所以整个复杂度为$O(M\log^2 n)$。

例题：移动电话

- 统计从(L, B)到(R, T)的矩形总和：



- 从上图可知：

S(L,B)(R,T)=S(1,1)(R,T)-S(1,1)(R,B-1)-
S(1,1)(L-1,T)+S(1,1)(L-1,B-1)

小结

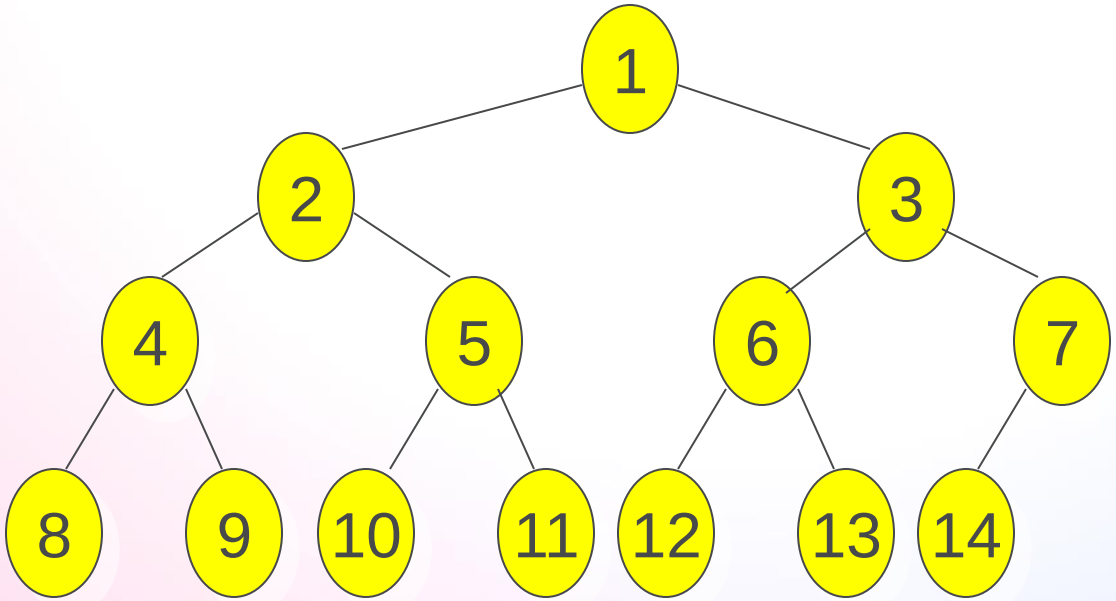
- 树状数组和线段树一样，是一种很高效的数据结构。它的优势在于空间消耗较小。对于长度为线段（区间），线段树至少需要2n的空间，而树状数组只要n即可。
- 树状数组的另一个优点就是编程简单。没有繁琐的指针和乱七八糟的二分，编写这样的程序是不易出错的。
- 树状数组的致命缺点是无法记录一些附加信息。比如“区间总长度”，“不相交区间个数”就无法用树状数组维护。

堆的定义

- 堆是一个完全二叉树
  - 所有叶子在同一层或者两个连续层
  - 最后一层的结点占据尽量左的位置
- 堆性质
  - 为空，或者最小元素在根上（或最大元素在根上）
  - 两棵子树也是堆

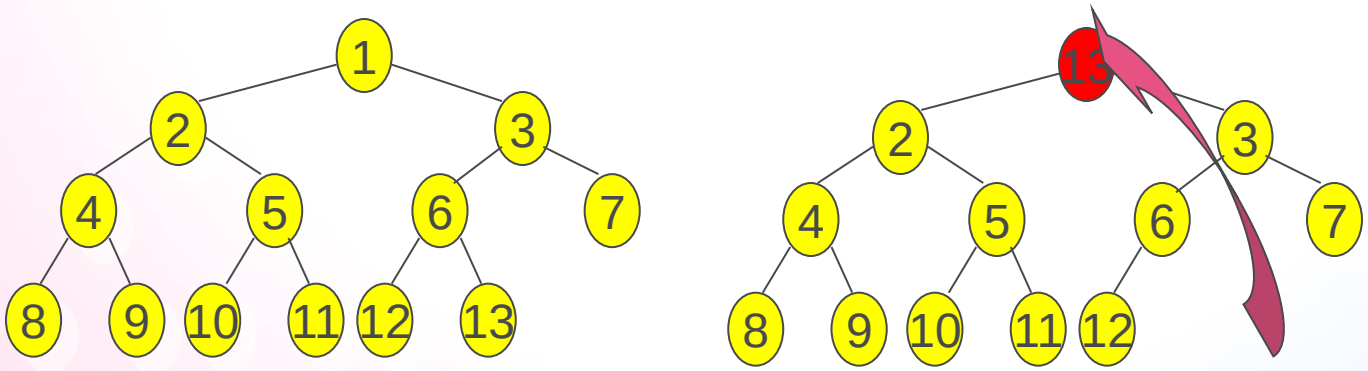
存储方式

- 最小堆的元素保存在heap[1..n]内
  - 根在heap[1]
  - K的左儿子是2k，K的右儿子是2k+1，
  - K的父亲是[k/2]



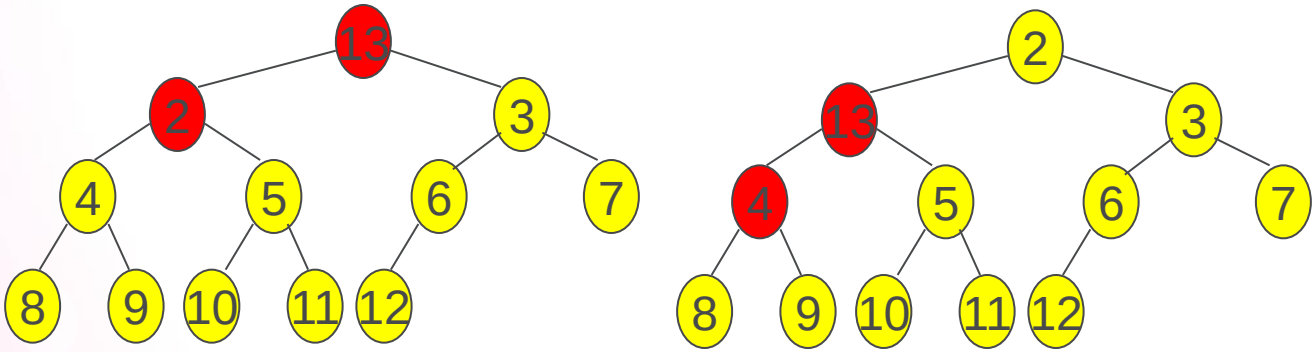
删除最小值元素

- 三步法
  - 直接删除根
  - 用最后一个元素代替根上元素
  - 向下调整



向下调整

- 首先选取当前结点p的较大儿子. 如果比p大, 调整停止, 否则交换p和儿子, 继续调整




```
void heapdown(int k) //把第k个结点往下调
{ while (2*k<=n)
    {i=min(2*k, 2*k+1); //返回2个中值较小的元素
     if (heap[i]<heap[k])
       {swap(i,k);  k=i; }
     else   return;
    }
}
```

插入元素和向上调整

- 插入元素是先添加到末尾，再向上调整
- 向上调整：比较当前结点p和父亲，如果父亲比p小，停止；否则交换父亲和p，继续调整

```
void  heapup (int k) //把第k个结点上调
{ while (k>1)
    {i=k/2; //i是k的父亲
     if (heap[i] > heap[k])
        {swap(i,k);  k=i;}  //交换结点i和k
     else  return;
    }
}
```

堆的建立

- 从下往上逐层向下调整. 所有的叶子无需调整，因此从n/2开始. 可用数学归纳法证明循环变量为i时，第i+1, i+2, …n均为堆的根

```
void  buildheap(int n) //建堆
 {
  for (i=n/2; i>=1; i--)
     heapdown(i);
 }
```

堆排序

- 每次输出堆顶元素后，把堆顶元素与最后一个节点元素交换，再把堆顶往下调整使其满足堆的性质。

```
void heap( ) //堆排序
{buildheap();
 for (i=n; i>1; i--)
   {cout<<heap[1]<<endl;
     swap(heap[1], heap[i]);
     heapdown(1)
   }
}
```

时间复杂度分析

- 向上调整/向下调整
  - 每层是常数级别, 共logn层, 因此O(logn)
- 插入/删除
  - 只调用一次向上或向下调整, 因此都是O(logn)
- 建堆
  - 最多n/2个节点需要往下调整，时间复杂度为O(n)
- 堆排序
  - 循环n-1次，每次为O(logn)，总的复杂度为O(nlogn)

堆——STL实现

- 大根堆：
  - 1.priority_queue<int> q; //默认
  - 2.priority_queue<Node,vector<Node>,less<Node> > q; //自带比较函数
- 小根堆：
  - priority_queue< Node,vector<Node>,greater<Node> > q; //自带比较函数

堆——STL实现

- 还在定义struct Node时重载，例如大根堆：

```
struct Node
  { int L, R, val;
    bool operator<(const Node &a) const{
        return val<a.val;  //以val从小到大排序
       }
  };
 priority_queue <Node>q; //定义大根堆
```

堆——STL实现

```
priority_queue <int> q //定义一个大根堆
q.push(); //元素入队
q.pop();  //队首元素出队
q.top();  //取队首元素
q.empty();  //如果队列为空返回true，否则返回false
q.size(); //返回优先队列中拥有的元素个数
```

例:打地鼠

- 打地鼠游戏开始后，会在地板上冒出一些地鼠来，你可以用榔头去敲击这些地鼠，每个地鼠被敲击后，将会增加相应的游戏分值。可是，所有地鼠只会在地上出现一段时间（而且消失后再也不会出现），每个地鼠都在0时刻冒出，但停留的时间可能是不同的，而且每个地鼠被敲击后增加的游戏分值也可能是不同。
- 小明最近经常玩这个游戏，以至于敲击每个地鼠只要1秒。他在想如何敲击能使总分最大。

例:打地鼠

- 输入：包含3行，第一行包含一个整数n（1<=n<=100000）表示有n个地鼠从地上冒出来，第二行n个用空格分隔的整数表示每个地鼠冒出后停留的时间（MaxT<=50000），第三行n个用空格分隔的整数表示每个地鼠被敲击后会增加的分值（MaxV<=1000）。每行中第i个数都表示第i个地鼠的信息。
- 输出：一行一个整数，表示所能获得的最大游戏总分值。

例:打地鼠

- 输入样例

  5

  5 3 6 1 4

  7 9 2 1 5

- 输出样例

  24

例题：Ksum

- Peter喜欢玩数组。NOIP这天，他从Jason手里得到了大小为n的一个正整数数组。Peter求出了这个数组的所有子段和，并将这n(n+1)/2个数降序排序，他想知道前k个数是什么。
- 输入

  第一行包含两个整数 n 和 k。

  接下来一行包含 n 个正整数，代表数组。
- a[i]≤10^9， k≤n(n+1)/2，n≤100000，k≤100000

例题：Ksum

- 输出

  输出k个数，代表降序之后的前k个子段和。

- 样例输入：

  3 4

  1 3 4

- 样例输出

  8 7 4 4


Thanks