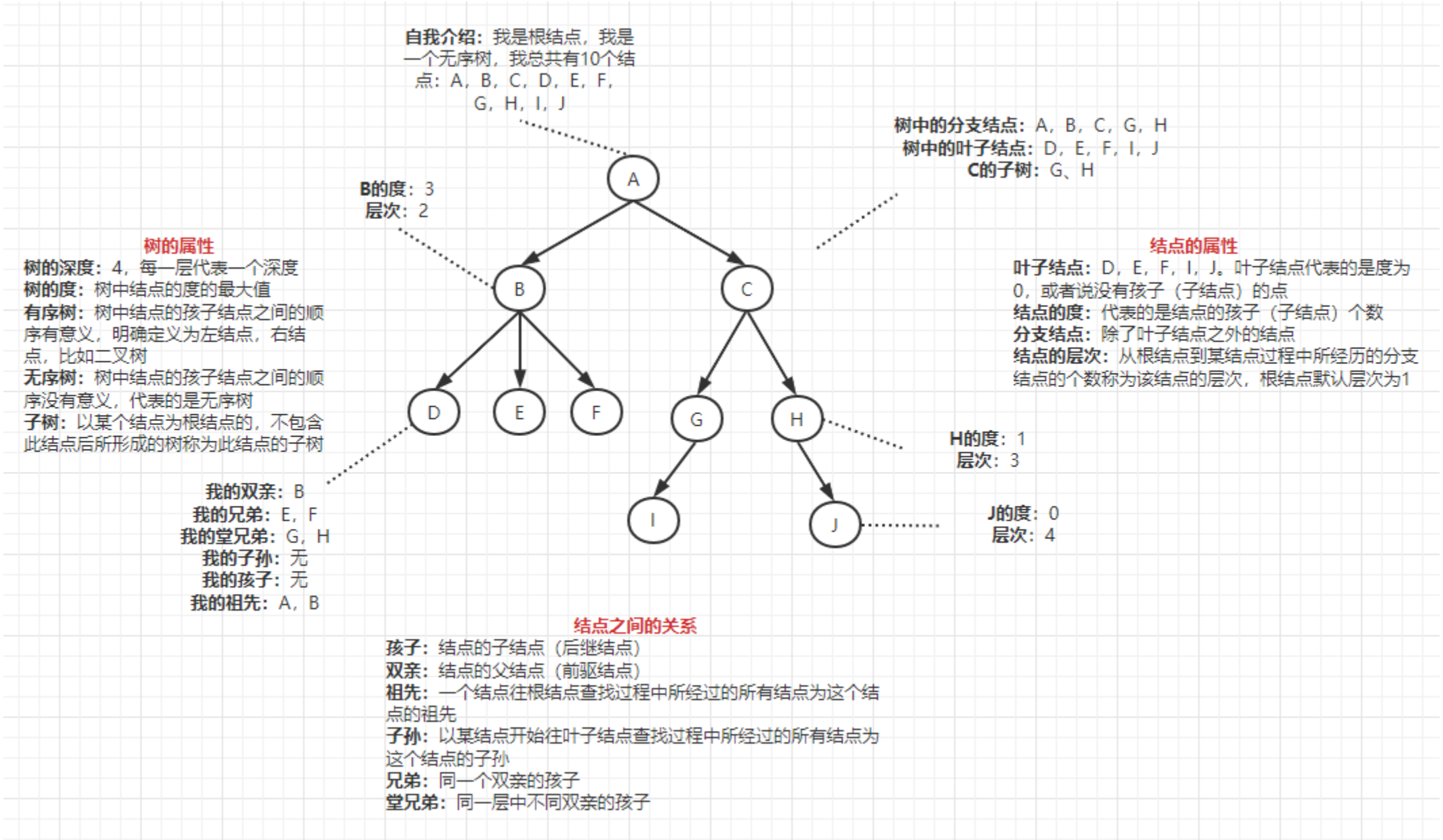
树形数据结构及其应用

长沙市雅礼中学 屈运华

List

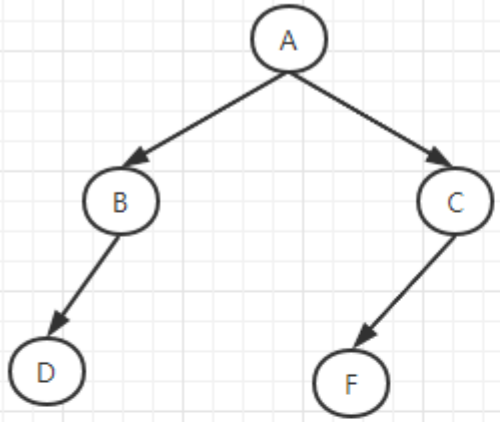
- 树、二叉树、哈夫曼树、堆
- 树状数组
- 线段树
- 李超线段树
- KD-TREE

树

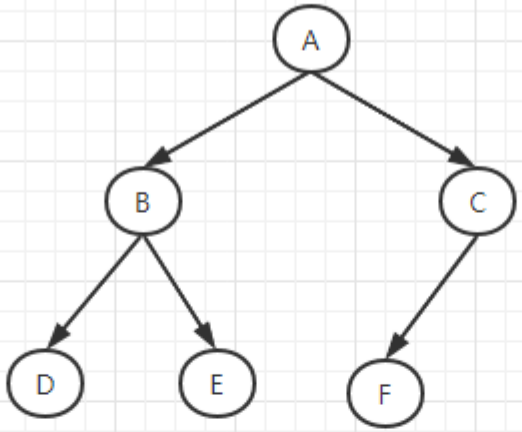

自我介绍：我是根结点，我是一个无序树，我总共有10个结点：A，B，C，D，E，F，G，H，I，J
树中的分支结点：A，B，C，G，H
树中的叶子结点：D，E，F，I，J
C的子树：G、H
B的度：3
层次：2
A
B
C
D
E
F
G
H
I
J
树的属性
树的深度：4，每一层代表一个深度
树的度：树中结点的度的最大值
有序树：树中结点的孩子结点之间的顺序有意义，明确定义为左结点，右结点，比如二叉树
无序树：树中结点的孩子结点之间的顺序没有意义，代表的是无序树
子树：以某个结点为根结点的，不包含此结点后所形成的树称为此结点的子树
结点的属性
叶子结点：D，E，F，I，J。叶子结点代表的是度为0，或者说没有孩子（子结点）的点
结点的度：代表的是结点的孩子（子结点）个数
分支结点：除了叶子结点之外的结点
结点的层次：从根结点到某结点过程中所经历的分支结点的个数称为该结点的层次，根结点默认层次为1
H的度：1
层次：3
J的度：0
层次：4
我的双亲：B
我的兄弟：E，F
我的堂兄弟：G，H
我的子孙：无
我的孩子：无
我的祖先：A，B
结点之间的关系
孩子：结点的子结点（后继结点）
双亲：结点的父结点（前驱结点）
祖先：一个结点往根结点查找过程中所经过的所有结点为这个结点的祖先
子孙：以某结点开始往叶子结点查找过程中所经过的所有结点为这个结点的子孙
兄弟：同一个双亲的孩子
堂兄弟：同一层中不同双亲的孩子

二叉树

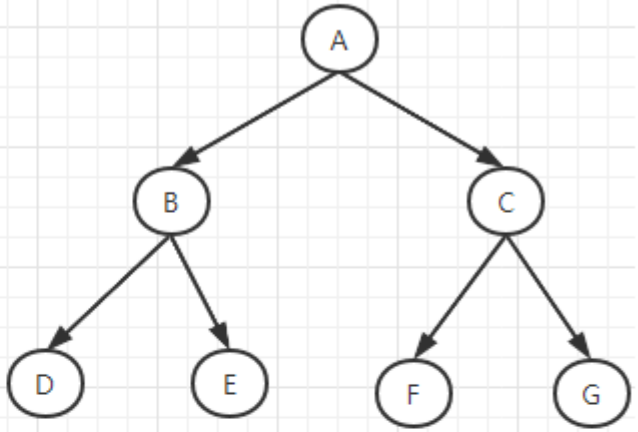
其它二叉树



完全二叉树：如果分支结点的子节点不满，将连续排序在左边



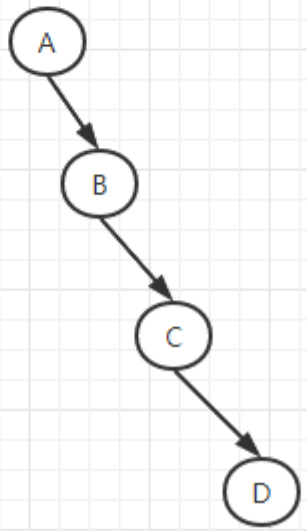
满二叉树：从根结点出发的每个分支结点的左结点与右节点都不为空



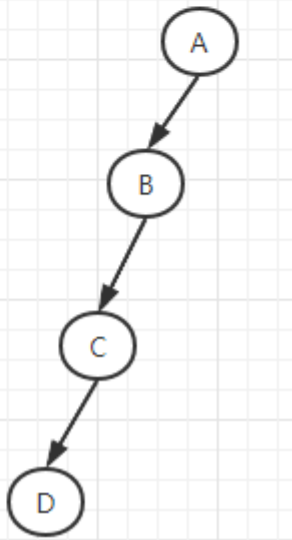
只包含根结点的二叉树

空二叉树

只包含右边结点的二叉树



只包含左边结点的二叉树



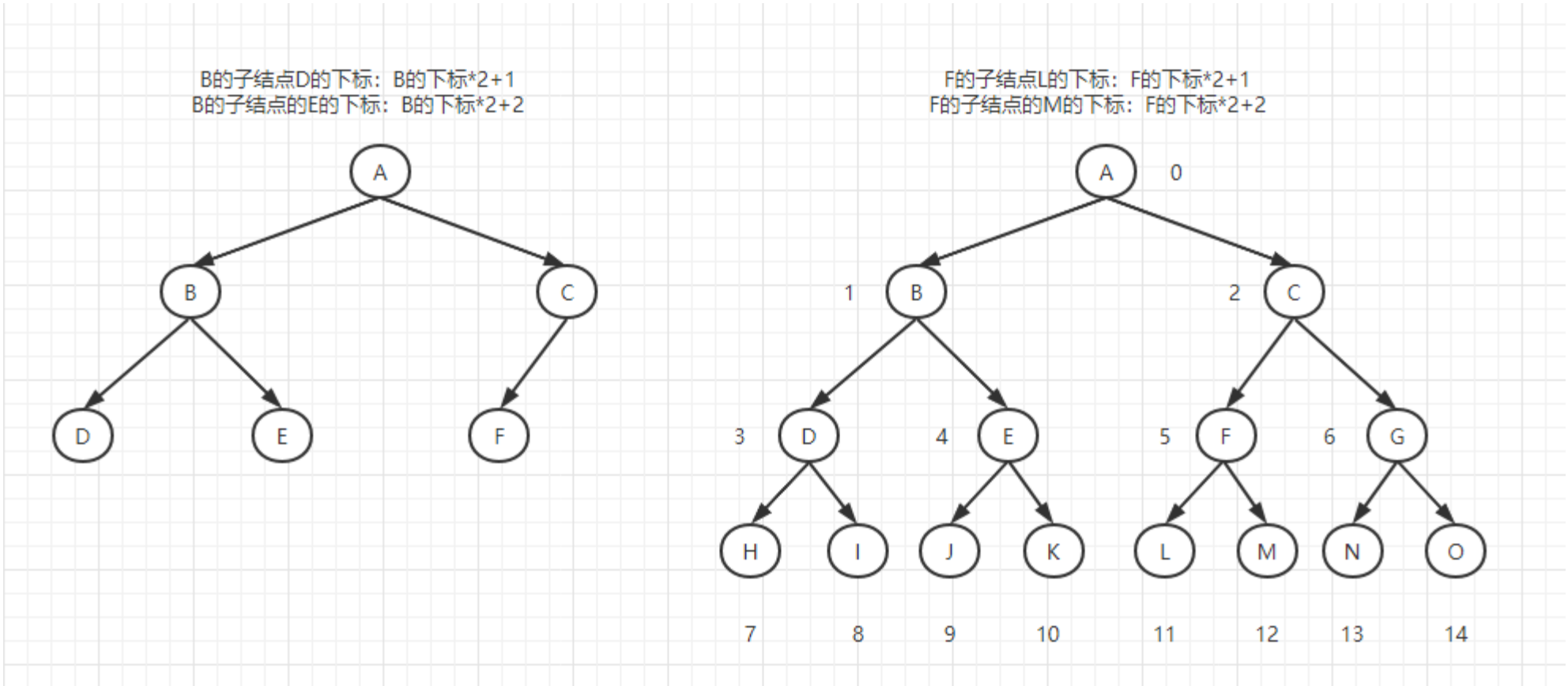
二叉树的重要性质

- 性质一：如果一个树中定义根结点的层数为0，以i代表其它结点的层数，那么第i层最多为2^i个结点
- 说明：第一层是根结点，为1个结点2^0=1，如果为满二叉树，那么第二层就最多为2个结点2^1=2，依次类推
- 性质二：如果一个树的高度为h，定义根结点的高度为0，那么一个高度为h的树中最多有2^(h+1)-1个结点，最少有2h-1个结点
- 性质三：对于任何非空二叉树T，如果其叶子结点的个数为N0，其度为2的结点的个数为N2，则N0=N2+1

二叉树的重要性质

- 性质四：满二叉树的叶子结点比分支结点多一个
- 说明：满二叉树中的结点要么是叶子结点，要么就是度为2的分支结点，推论跟性质三类似，可以作为参考
- 性质五：n个结点的完全二叉树的高度h为不大于$\log_2^n$的最大整数。
- 说明：根据性质二与完全二叉树的性质可得出，2h-1<T<2(h+1)-1 ——> 2^h <T<2^(h+1) ——> h<$\log_2^n$<h+1 ——> h为不大于$\log_2^n$的最大整数
- 性质六：如果n个结点的完全二叉树的结点按层次并按从左到右的顺序从0开始编号，对任一结点i（0<=i<=n-1）都有：
- 1、序号为0的结点是根结点
- 2、对于i>0,其父结点的编号是(i-1)/2
- 3、若2i+1<n，其左子结点的序号为2i+1,否则无左子结点
- 4、若2i+2<n，其右子结点的序号为2i+2,否则无右子结点

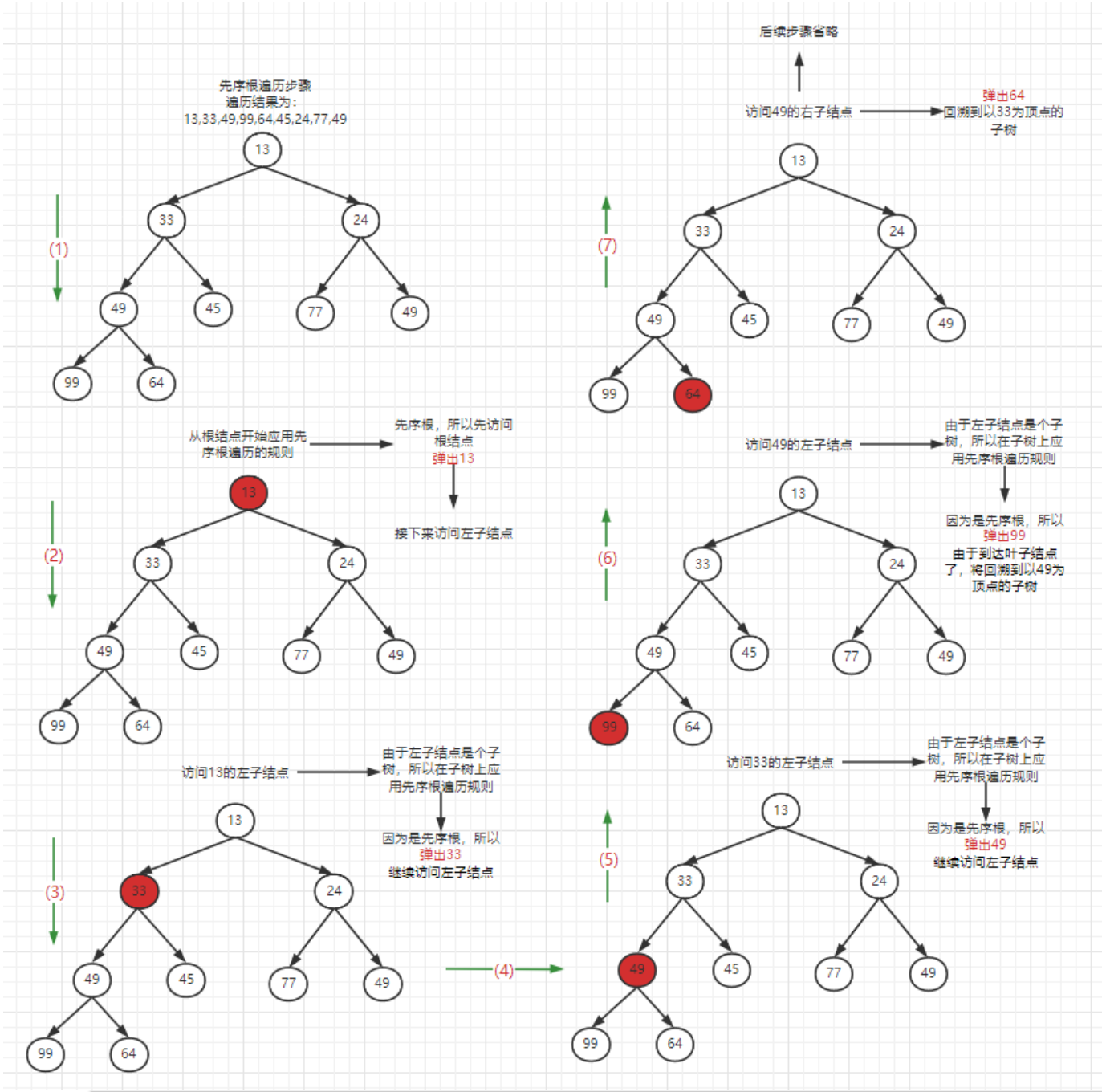
二叉树的重要性质


B的子结点D的下标：B的下标*2+1
B的子结点的E的下标：B的下标*2+2
A
B
C
D
E
F
F的子结点L的下标：F的下标*2+1
F的子结点的M的下标：F的下标*2+2
A 0
1 B
2 C
3 D
4 E
5 F
6 G
H
I
J
K
L
M
N
O
7
8
9
10
11
12
13
14

二叉树的遍历

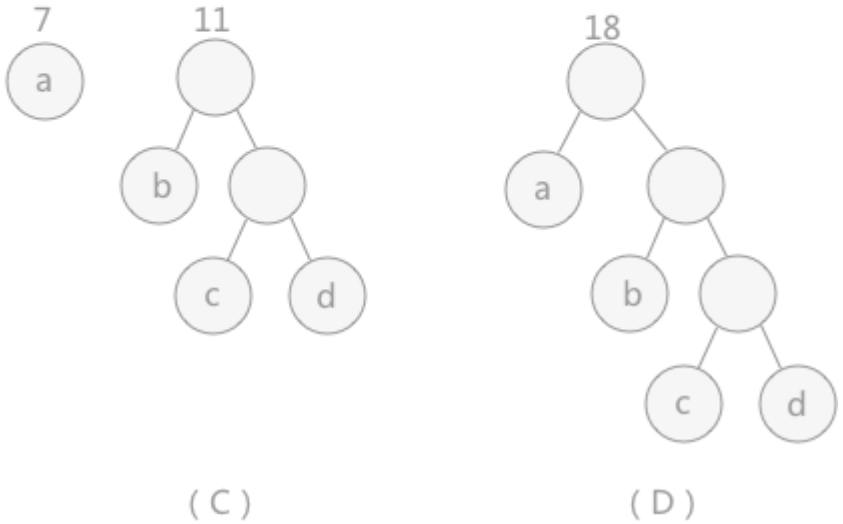
- 按深度优先方式遍历一棵二叉树，需要做三件事，遍历左子树、遍历右子树和访问根结点。这三件事的不同顺序会产生三种遍历的顺序：
- 1、先序根遍历：遍历根结点 -> 遍历左子树 -> 遍历右子树
- 2、中序根遍历：遍历左子树 -> 遍历根结点 -> 遍历右子树
- 3、后序根遍历：遍历左子树 -> 遍历右子树 -> 遍历根结点

二叉树的遍历

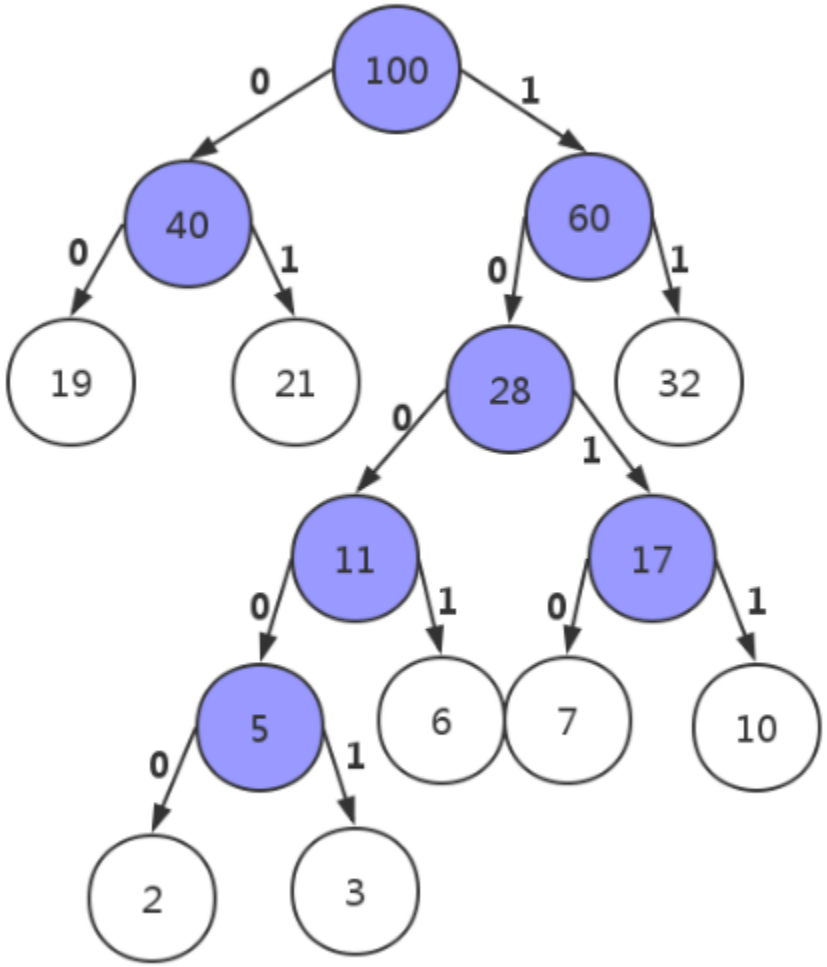

先序根遍历步骤
遍历结果为：
13,33,49,99,64,45,24,77,49
(1)
从根结点开始应用先序根遍历的规则
先序根，所以先访问根结点
弹出13
接下来访问左子结点
(2)
访问13的左子结点
由于左子结点是个子树，所以在子树上应用先序根遍历规则
因为是先序根，所以
弹出33
继续访问左子结点
(3)
访问33的左子结点
由于左子结点是个子树，所以在子树上应用先序根遍历规则
因为是先序根，所以
弹出49
继续访问左子结点
(4)
(5)
访问49的左子结点
由于左子结点是个子树，所以在子树上应用先序根遍历规则
因为是先序根，所以
弹出99
由于到达叶子结点了，将回溯到以49为顶点的子树
(6)
访问49的右子结点
弹出64
回溯到以33为顶点的子树
(7)
后续步骤省略

哈夫曼树

当用 n 个结点（都做叶子结点且都有各自的权值）试图构建一棵树时，如果构建的这棵树的带权路径长度最小，称这棵树为“最优二叉树”，有时也叫“赫夫曼树”或者“哈夫曼树”。



7,19,2,6,32,3,21,10构成的哈夫曼树

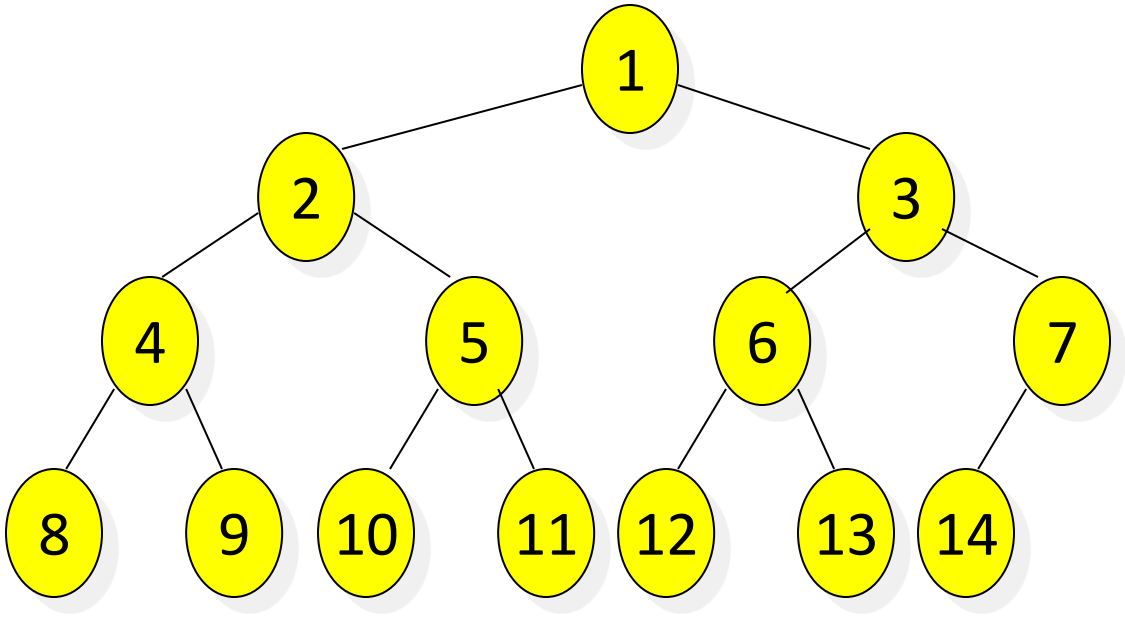


堆的定义

- 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

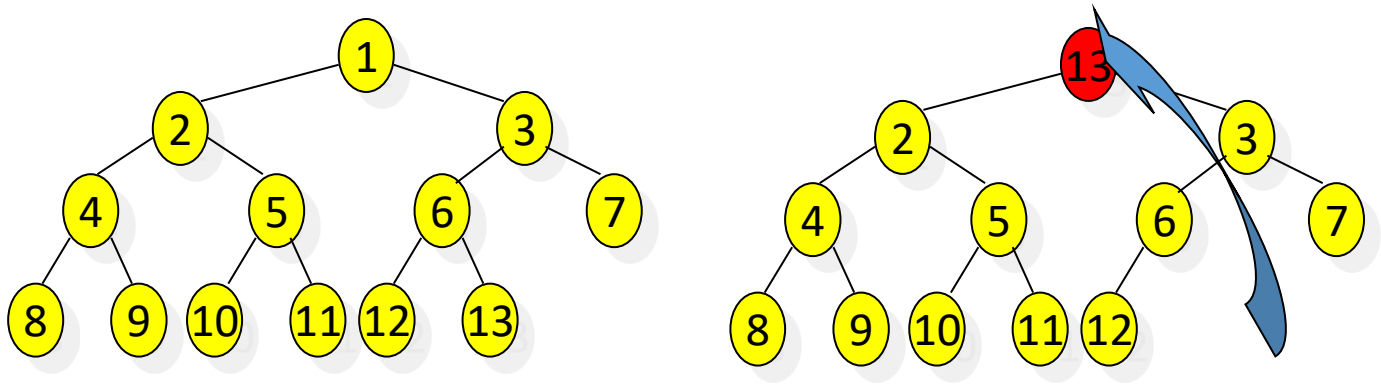
存储方式

- 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]



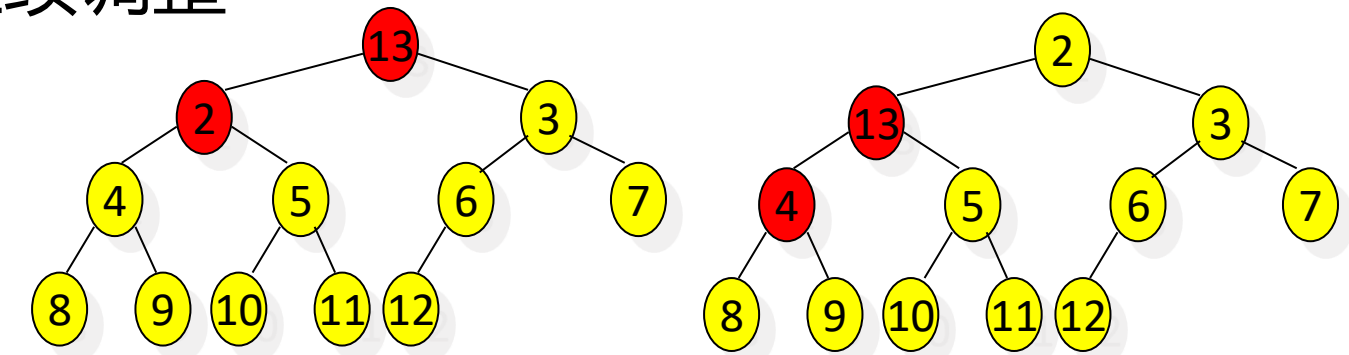
删除最小值元素

- 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的较小儿子. 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

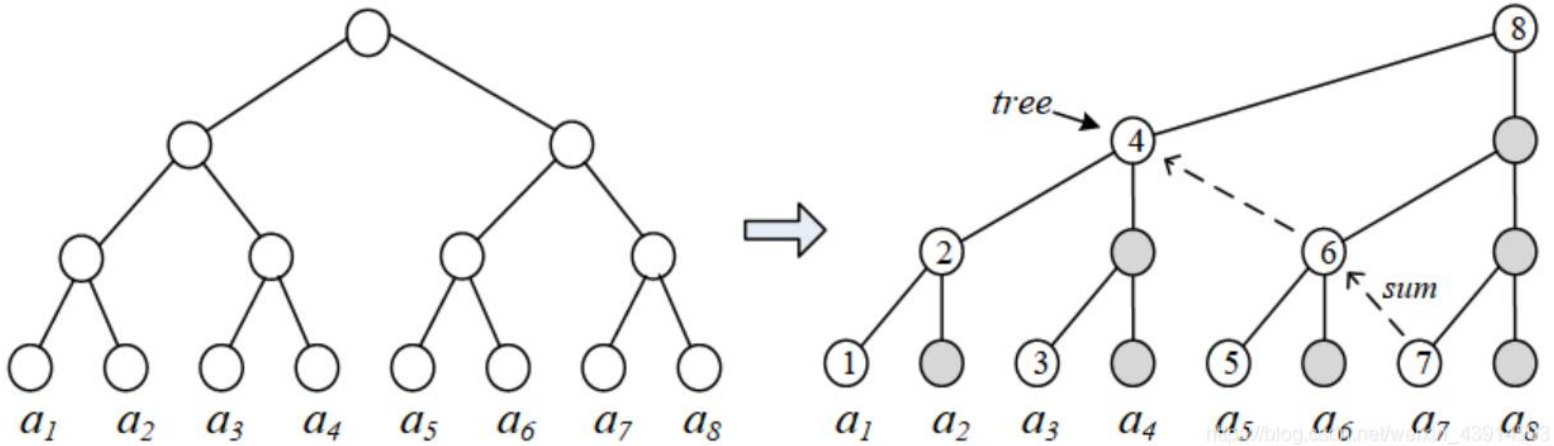
打地鼠

- 相比从前往后遍历，从后往前遍历同时记录当前时间的方法更容易理解，且一定能保证是最优解。首先对于众多地鼠，停留时间最长的肯定是最后一个打的，无论其价值多少，因为小于这个最长时间的地鼠都打不了。对于相同时间的，首先取价值最大的打。

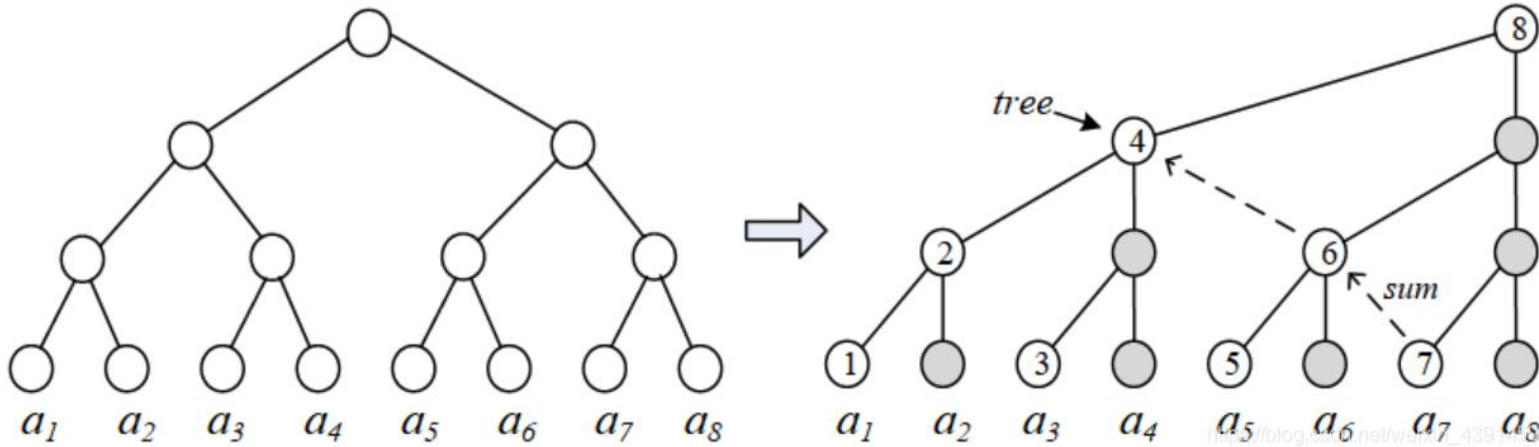
- 在时间从后往前的遍历过程中，由于对于当前时间，优先队列中筛选出来的都是该时间下能打，且权值最大的地鼠，对于空余的时间，没有地鼠消失，此时在优先队列中找到还未消失且权值最大的地鼠打，这样就填补了空余时间，还不会影响时间较短且权值最大的地鼠。在向前遍历过程中，不断有时间更短的地鼠加入到队列中，使得变化的时间下更多符合条件的可打地鼠加入选择范围。通过打一只，时间now减一的操作进行时间的遍历。

树状数组

- 树状数组（Binary Indexed Tree, BIT），从它的英文名可以看出，它是利用数的二进制特征进行检索的一种树状的结构。
- 如何利用二分的思想高效地求前缀和？以A = {a 1 , a 2 , . . . , a 8 }这8个元素为例，左图是二叉树的结构，右边把它画成树状。
- 右图圆圈中标记有数字的结点，存储的是称为树状数组的tree[]。一个结点上的tree[]的值，就是它树下的直连的子结点的和。例如t r e e [1] = a 1 , t r e e [2] = t r e e [1] + a 2 , t r e e [3] = a 3 , t r e e [4] = t r e e [2] + t r e e [3] + a 4 , . . . , t r e e [8] = t r e e [4] + t r e e [6] + t r e e [7] + a 8

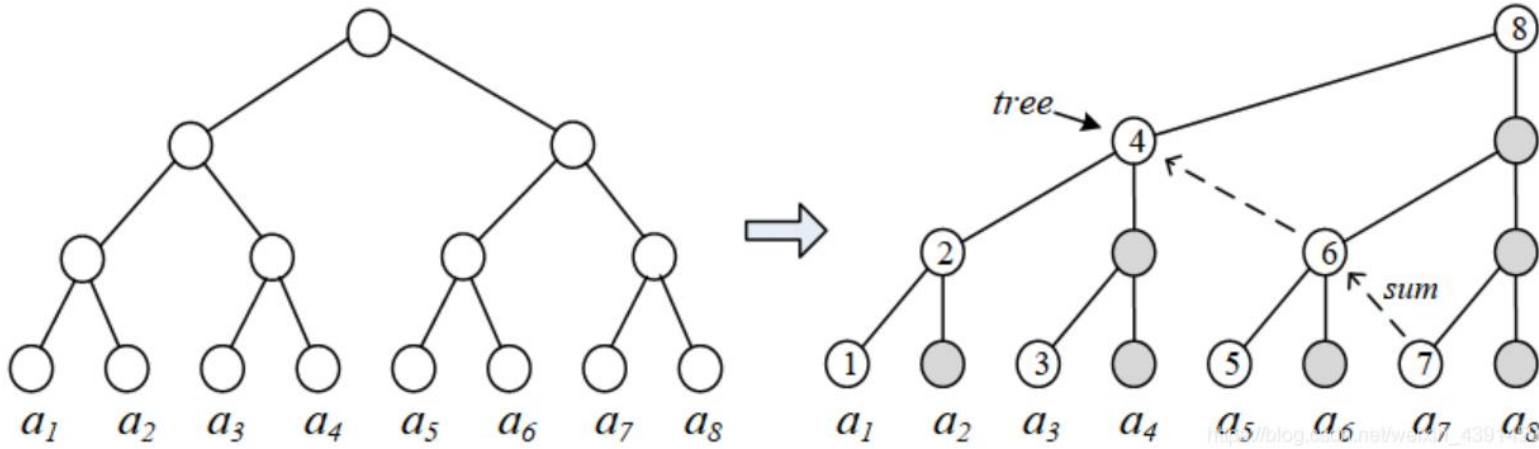


树状数组



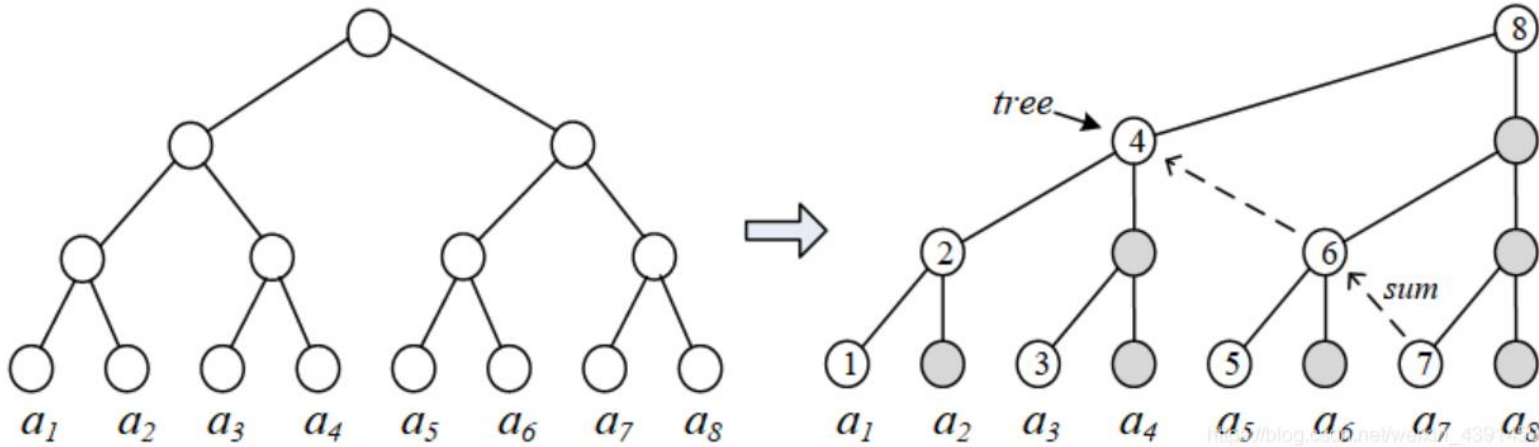
- 利用tree[]，可以高效地完成下面两个操作：
- （1）查询，即求前缀和sum，例如：sum(8) = tree[8]，sum(7) = tree[7] + tree[6] + tree[4]，sum(6) = tree[6] + tree[4]。右图中的虚线箭头是计算sum(7)的过程。显然，计算的复杂度是O(logn)的，这样就达到了快速计算前缀和的目的。
- （2）维护。tree[]本身的维护也是高效的。当元素a发生改变时，能以O(logn)的高效率修改tree[]的值。例如更新了a 3，那么只需要修改tree[3]、tree[4]、tree[8]…，即修改它和它上面的那些结点：父结点（即x+=lowbit(x)）以及父结点的父结点。

树状数组



- 有了方案，剩下的问题是，如何快速计算出tree[]？观察查询和维护两个操作，发现：
- （1）查询的过程，是每次去掉二进制的最后的1。例如求sum(7) = tree[7] + tree[6] + tree[4]，步骤是：
  - 1）7的二进制是111，去掉最后的1，得110，即tree[6]；
  - 2）去掉6的二进制110的最后一个1，得100，即tree[4]；
  - 3）4的二进制是100，去掉1之后就没有了。
- （2）维护的过程，是每次在二进制的最后的1上加1。例如更新了a3，需要修改tree[3]、tree[4]、tree[]8…等等，步骤是：
  - 1）3的二进制是11，在最后的1上加上1得100，即4，修改tree[4]；
  - 2）4的二进制是100，在最后的1上加1，得1000，即8，修改tree[]8；
  - 3）继续修改tree[16]、tree[32]…等等。
- 最后，树状数组归结到一个关键问题：如何找到一个数的二进制的最后一个1。

树状数组



- 神奇的lowbit(x)
- lowbit(x)=x &−x，功能是找到x的二进制数的最后一个1。其原理是利用了负数的补码表示，补码是原码取反加一。例如x = 6 = （00000110）2，-x = x_补 = （11111010）2

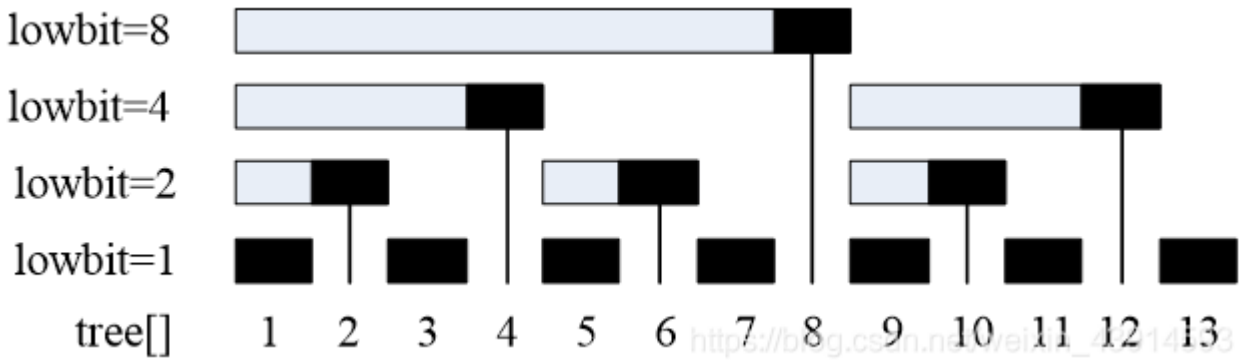
1~9的tlowbit结果是：

x	1	2	3	4	5	6	7	8	9
x的二进制	1	10	11	100	101	110	111	1000	1001
lowbit(x)	1	2	1	4	1	2	1	8	1
tree[x]数组	tree[1] =a1	tree[2] =a1+a2	tree[3] =a3	tree[4] =a1+a2+a3+a4	tree[5] =a5	tree[6] =a5+a6	tree[7] =a7	tree[8] =a1 +…+a8	tree[9] =a9

树状数组

令m=lowbit(x)，定义tree[x]的值，是把a_x和它前面共m个数相加的结果，如上表所示。例如lowbit(6)=2，有t r e e [6] = a 5 + a 6。
tree[]是通过lowbit()计算出的树状数组，它能够以二分的复杂度存储一个数列的数据。 具体地，tree[x]中储存的是[x−lowbit(x)+1,x]中每个数的和。

上面的表格可以画成下图。横线中的黑色表示tree[x]，它等于横线上元素相加的和。



树状数组概念和编码

```
#define lowbit(x)  ((x) & - (x))
int tree[Maxn];
void update(int x, int d) {   //修改元素a_x,  a_x = a_x + d
    while(x <= Maxn) {
        tree[x] += d;
        x += lowbit(x);
    }
}
int sum(int x) {          //返回值是前缀和：ans = a1 + ... + a_x
    int ans = 0;
    while(x > 0){
        ans += tree[x];
        x -= lowbit(x);
    }
    return ans;
}
```

树状数组概念和编码

上述代码的使用方法是：

（1）初始化。主程序先清空数组tree[]，然后读取 a_1，a_2， ...，a_n，用update()逐一处理这n个数，得到tree[]数组。代码中并不需要定义数组a，因为它隐含在tree[]中。

（2）求前缀和。sum()函数计算a_1+a_2 + ... + a_x。求和是基于数组tree[]的。

（3）修改元素。执update()，即修改数组tree[]。

从上面的使用方法可以看出，tree[]这个数据结构可以用于记录元素，以及计算前缀和。

值得注意的是，代码中并不需要定义和存储数组a，因为它隐含在tree[]中，用s u m ( ) sum(sum()函数计算a_i = sum(i) - sum(i-1)，复杂度O(logn)。

区间修改 + 单点查询

一个序列A = {a1, a2, ..., an}的更新（修改）有两种：

（1）单点修改。一次改一个数；

（2）区间修改。一次改变一个区间[L,R]内所有的数，例如把每个数统一加上d。

树状数组的原始功能是“单点修改 + 区间查询”，是否能扩展为“区间修改”？只需一个简单而巧妙的操作（差分数组），就能把单点修改用来处理区间修改问题，实现高效的“区间修改 + 单点查询”，进一步也能做到“区间修改 + 区间查询”。

Color the ball hdu 1556

- 问题描述：N个气球排成一排，从左到右依次编号为1, 2, 3 … N。每次给定2个整数L, R(L<= R)，lele从气球L开始到气球R依次给每个气球涂一次颜色。但是N次以后lele已经忘记了第i个气球已经涂过几次颜色了，你能帮他算出每个气球被涂过几次颜色吗?
- 输入：每个测试实例第一行为一个整数N，(N <= 100000)。接下来的N行，每行包括2个整数L, R(1 <= L<= R<= N)。当N = 0，输入结束。
- 输出：每个测试实例输出一行，包括N个整数，第I个数代表第I个气球总共被涂色的次数。

Color the ball hdu 1556

定义数组a[i]为气球i被涂色的次数。

如果用暴力处理N次区间修改，是$O(N^2)$的。用树状数组，如果只是简单把区间$[L,R]$内的每个数a[x]用update()进行单点修改，复杂度更差，是$O(N^2 logN)$的。下面把单点修改处理成区间修改，复杂度$O(NlogN)$。

如何用树状数组处理区间修改？题目要求把$[L,R]$区间内每个数加上d，但是下面的解法不是对区间内每个数加d，而是操作一个被称为“差分数组”的D，它的定义是：

$D[k] = a[k] - a[k-1]$，即原数组相邻元素的差。

从定义可以推出：

$$a[k] = D[1] + D[2] + ... + D[k] = \sum_{i=1}^{k} D(i)$$

这个公式深刻地描述了a和D的关系，“**差分是前缀和的逆运算**”，它把求$a[k]$转化为求D的前缀和，前缀和正适合用树状数组来处理。

对于区间$[L,R]$的修改问题，对D做以下操作：

(1) 把$D[L]$加上d；

(2) 把$D[R+1]$减去d。

然后用树状数组函数sum()求前缀和$sum[x] = D[1] + D[2] + ... + D[x]$，有：

(1) $1 \le x < L$，前缀和sum[x]不变；

(2) $L \le x \le R$，前缀和sum[x]增加了d；

(3) $R < x \le N$，前缀和sum[x]不变，因为被$D[R+1]$中减去的d抵消了。

sum[x]的值与直接把$[L,R]$区间内每个数加上d得到的a[x]是相等的。这样，就利用树状数组高效地计算出了区间修改后的a[x]。

Color the ball hdu 1556

```
const int Maxn = 100010;
int main(){
    int n;
    while(~scanf("%d",&n)) {
        memset(tree,0,sizeof(tree));        //只需要一个tree[]数组
        for(int i=1;i<=n;i++) {             //区间修改
            int L, R;
            scanf("%d%d",&L,&R);
            update(L,1);                    //本题的d = 1
            update(R+1,-1);
        }
        for(int i=1;i<=n;i++){              //单点查询
            if(i!=n)  printf("%d ",sum(i));   //把sum(i)看成a[i]
            else      printf("%d\n",sum(i));
        }
    }
    return 0;
```

差分数组

- hdu 1556这一题的思路借用了“差分数组”的概念。
- “差分数组”D[]是原数组a[]的一个辅助数组，D[k] = a[k] - a[k-1]，记录相邻元素之间的差值，它专用于“区间修改”这种题型。
- 　　也可以直接用“差分数组”写代码，和上面树状数组的代码差不多。就本题的要求而言（打印所有气球，即输出所有的前缀和），“差分数组”的代码，复杂度比树状数组的代码还要好一些。

- 不过，遇到“区间修改”这种题型，还是建议用树状数组来求解。原因是差分数组对“区间修改”是很高效的，但是对“单点查询”并不高效。即使只查询一个前缀和，差分数组仍然要计算所有的前缀和，复杂度O(n；而树状数组做一次前缀和计算是O(logn)的。

区间修改+区间查询

- 前面的例题完成的是“区间修改 + 单点查询”，下面考虑把单点查询扩展到区间查询，即查询的不是一个单点a[x]的值，而是区间[i,j]的和。
- 仅仅用一个树状数组，无法同时高效地完成“区间修改”和“区间查询”，因为这个树状数组的tree[]已经用于“区间修改”，它用um()计算了单点a[x]，不能再用于求a[i]~a[j]的区间和。
- 读者可能想到再加一个树状数组，也许能接着高效地完成区间查询。但是如果这两个树状数组只是简单地一个做“区间修改”，一个做“区间查询”，合起来效率并不高。
- 做一次长度为k的区间修改，计算区间内每个a[x]的复杂度是O(logn)的；如果继续用一个树状数组处理这k个a[x]，复杂度是O(klogn)的；做n次修改和询问，总复杂度O(n^2logn)。

- 这两个树状数组需要紧密结合才能高效完成“区间修改 + 区间查询”，称为“二阶树状数组”，它也是“差分数组”概念和树状数组的结合。下面给出一个典型例题。

洛谷P3372

- 问题描述：已知一个数列，进行两种操作：（1）把某区间每个数加上d；（2）求某区间所有数的和。
- 输入：第一行包含两个整数 n，m，分别表示该数列数字的个数和操作的总个数。第二行包含n个用空格分隔的整数，其中第i个数字表示数列第i项的初始值。接下来m行每行包含3或4个整数，表示一个操作，具体如下：
- （1）1 L R d：将区间[L, R]内每个数加上d。
- （2）2 L R：输出区间[L, R]内每个数的和。
- 输出：输出包含若干行整数，即为所有操作（2）的结果。
- 1 ≤ n，m ≤ 10^5，元素的值在[−2^63 , 2^63]内。

洛谷P3372

操作（1）是区间修改，操作（2）是区间查询。

首先，求区间和$\mathrm{sum(L,R)=a[L]+a[L+1]+...+a[R]=sum(1,R)-sum(1,L-1)}$，问题转化为求$\mathrm{sum(1,k)}$。

定义一个差分数组D，它和原数组a的关系仍然是$\mathrm{D[k]=a[k]-a[k-1]}$，有$\mathrm{a[k]=D[1]+D[2]+...+D[k]}$，下面推导区间和，看它和求前缀和有没有关系，如果有关系，就能用树状数组。

$$
\begin{aligned}
&\mathrm{a_1+a_2+...+a_k}\\
&\mathrm{=D_1+(D_1+D_2)+(D_1+D_2+D_3)+...+(D_1+D_2+...+D_k)}\\
&\mathrm{=k*D_1+(k-1)*D_2+(k-2)*D_3+...+(k-(k-1))D_k}\\
&\mathrm{=k(D_1+D_2+...+D_k)-(D_2+2D_3+...+(k-1)D_k)}\\
&\mathrm{=k\sum_{i=1}^{k}D_i-\sum_{i=1}^{k}(i-1)D_i}
\end{aligned}
$$

这是求两个前缀和，用两个树状数组分别处理，一个实现$\mathrm{D_i}$，一个实现$\mathrm{(i-1)D_i}$。

下面是“区间修改 + 区间查询”的代码，完全重现了上面推导出的结论。

代码中的$\mathrm{update1()}$和$\mathrm{update2()}$、$\mathrm{sum1()}$和$\mathrm{sum2()}$几乎一样。也可以合起来写成$\mathrm{update1(ll\ x,ll\ d,int\ v)}$的样子，用v来区分处理tree1和tree2。不过像下面这样分开写更清晰，编码更快。

代码的复杂度是$\mathrm{O(mlogn)}$。

二维区间修改 + 区间查询
上帝造题的七分钟

- 洛谷 P4514
- 输入：第一行是X n m，代表矩阵大小为n×m。从第二行开始到文件尾的每一行出现以下两种操作：
- L a b c d delta 代表将(a,b),(c,d)为顶点的矩形区域内所有数字加上delta。
- k a b c d 代表求(a,b),(c,d)为顶点的矩形区域内所有数字的和。
- 输出：针对每个k操作，输出答案。
- 输入样例：
- X 4 4
- L 1 1 3 3 2
- L 2 2 4 4 1
- k 2 2 3 3
- 输出样例：
- 12
- 注：$1 \le n \le 2048$, $1 \le m \le 2048$, $-500 \le delta \le 500$, 操作不超过200000个,保证运算过程中及最终结果均不超过32位带符号整数类型的表示范围。

本题需要用二维树状数组。二维的“区间修改+区间查询”，就是一维“区间修改+区间查询”的扩展，方法和推导过程类似。

（1）二维区间修改

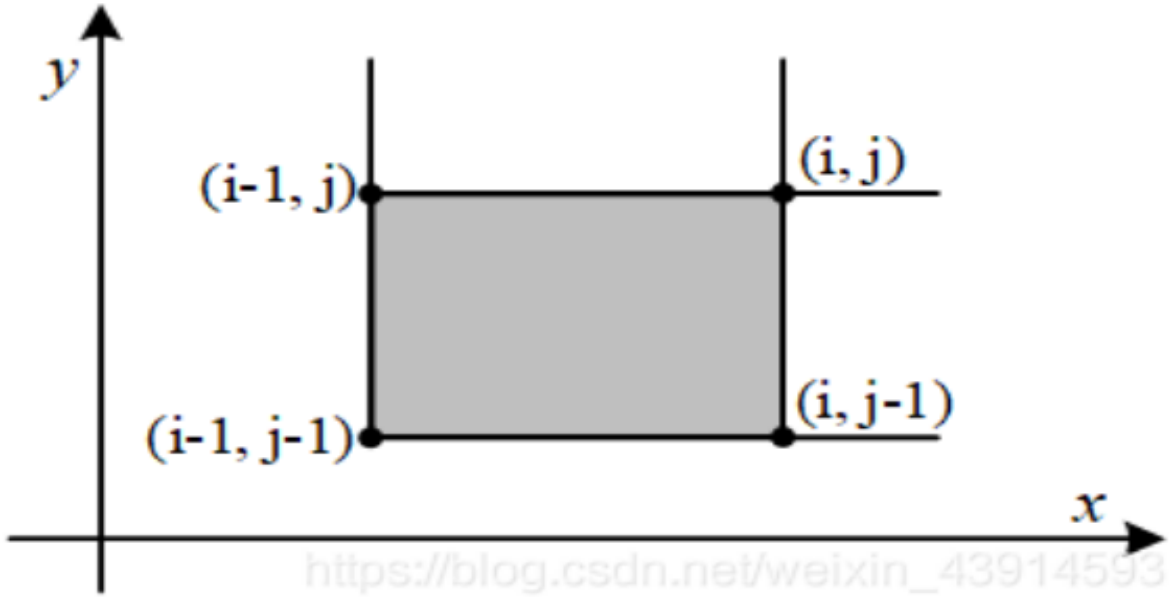


图2 二维差分数组的定义

如何实现区间修改？需要结合二维的差分数组。定义一个二维的差分数组$D[i][j]$，它和矩阵元素$a[c][d]$的关系是：

$D[i][j] = a[i][j] - a[i-1][j] - a[i][j-1] + a[i-1][j-1]$，对照上图，$D[i][j]$就是阴影的面积。

$a[c][d] = \sum_{i=1}^{c} \sum_{j=1}^{d} D[i][j]$，看成对以$(1,1)$、$(c,d)$为顶点的矩阵内的$D[i][j]$求和。

它们同样满足“差分是前缀和的逆运算”关系。

用二维树状数组实现$D[i][j]$，编码见后面的代码中的$update()$和$sum()$。进行区间修改的时候，在$update()$中，每次第i行减少$lowbit(i)$，第j列减$lowbit(j)$，复杂度$O(lognlogm)$。

（2）二维区间查询

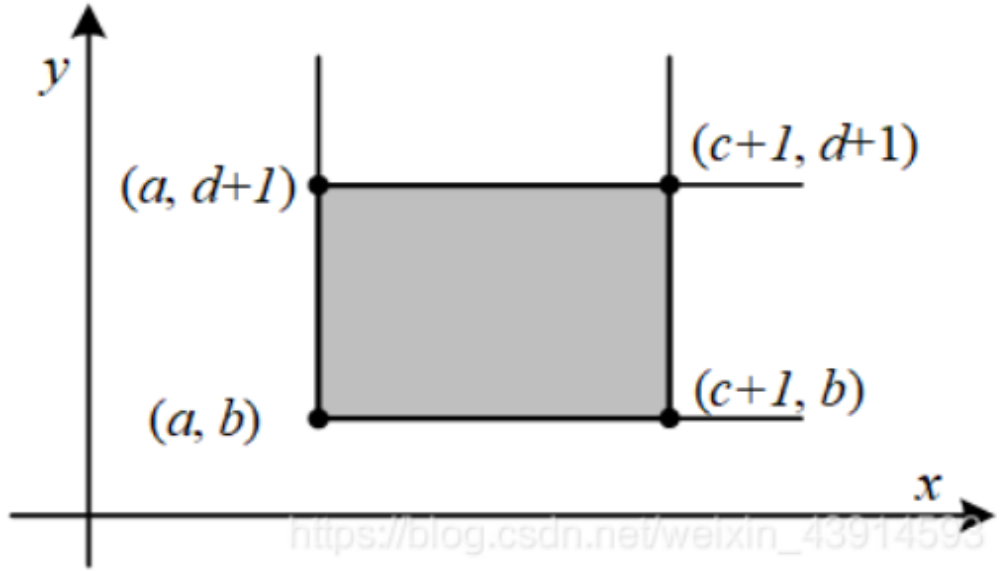


图3 二维区间求和

查询(a, b)、(c, d)为顶点的矩阵区间和，对照上图的阴影部分，有：

$$\sum_{i=a}^{c}\sum_{j=b}^{d} a[i][j] = \sum_{i=1}^{c}\sum_{j=1}^{d} a[i][j] - \sum_{i=1}^{c}\sum_{j=1}^{b-1} a[i][j] - \sum_{i=1}^{a-1}\sum_{j=1}^{d} a[i][j] + \sum_{i=1}^{a-1}\sum_{j=1}^{b-1} a[i][j]$$

问题转化为计算$\sum_{i=1}^{n}\sum_{j=1}^{m} a[i][j]$，根据它和差分数组D的关系进行变换[1]：

$$\begin{aligned}
&\sum_{i=1}^{n}\sum_{j=1}^{m} a[i][j] \\
&= \sum_{i=1}^{n}\sum_{j=1}^{m}\sum_{k=1}^{i}\sum_{l=1}^{j} D[k][l] \\
&= \sum_{i=1}^{n}\sum_{j=1}^{m} D[i][j] \times (n-i+1) \times (m-j+1) \\
&= (n+1)(m+1)\sum_{i=1}^{n}\sum_{j=1}^{m} D[i][j] - (m+1)\sum_{i=1}^{n}\sum_{j=1}^{m} D[i][j] \times i - (n+1)\sum_{i=1}^{n}\sum_{j=1}^{m} D[i][j] \times j + \sum_{i=1}^{n}\sum_{j=1}^{m} D[i][j] \times i \times j
\end{aligned}$$

线段树

- 概况地说，线段树是“分治法思想 + 二叉树结构 + lazy-tag技术”。

  - 1.1 线段树是分治法和二叉树的结合
  - 线段树是一棵二叉树，树上的结点是“线段”（或者理解为“区间”）。下图是包含10个元素的线段树，观察这棵树，它的基本特征是：
  - （1）用分治法自顶向下建立，每次分治左右子树各一半；
  - （2）每个结点都表示一个“线段”，非叶子结点包含多个元素，叶子结点只有一个元素；
  - （3）除了最后一层，其他层都是满的，这种结构的二叉树，层数是最少的。

线段树

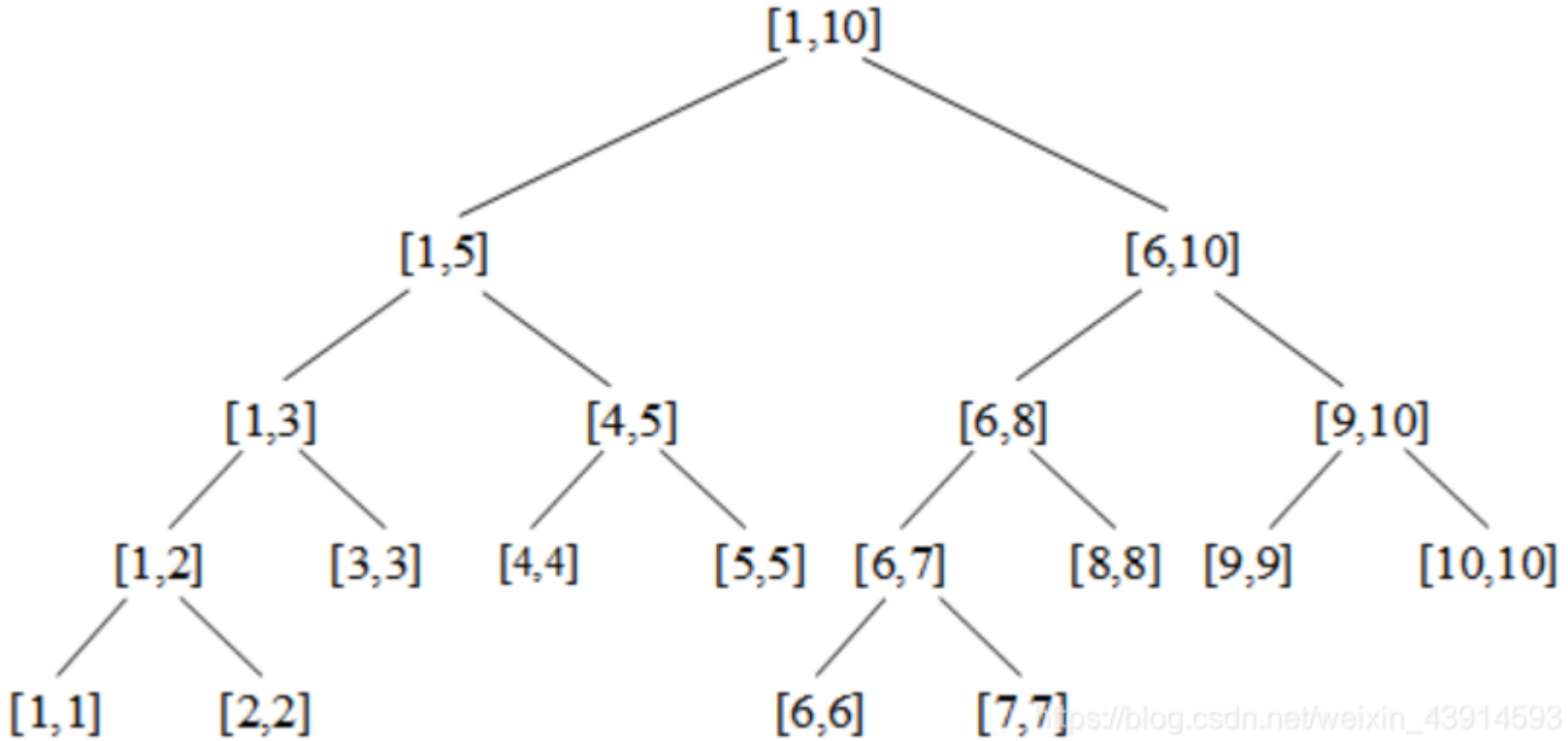
考查每个线段[L, R]，L是左端，R是右端。

（1）L = R，说明这个结点只有一个元素，它是一个叶子结点。

（2）L < R，说明这个结点代表的不只一个点，那么它有两个儿子，左儿子区间是[L, M]，右儿子区间是[M+1, R]，其中M = (L + R) / 2。例如：L = 1，R = 5，M = 3，左儿子是[1, 3]，右儿子是[4, 5]。

线段树是二叉树，一个区间每次折一半往下分，包含n个元素的线段树，最多分logn次就到达最低层。需要查找一个点或者区间的时候，顺着结点往下找，最多logn次就能找到。

结点所表示的“线段”的值，可以是区间和、最值或者其他根据题目灵活定义的值。

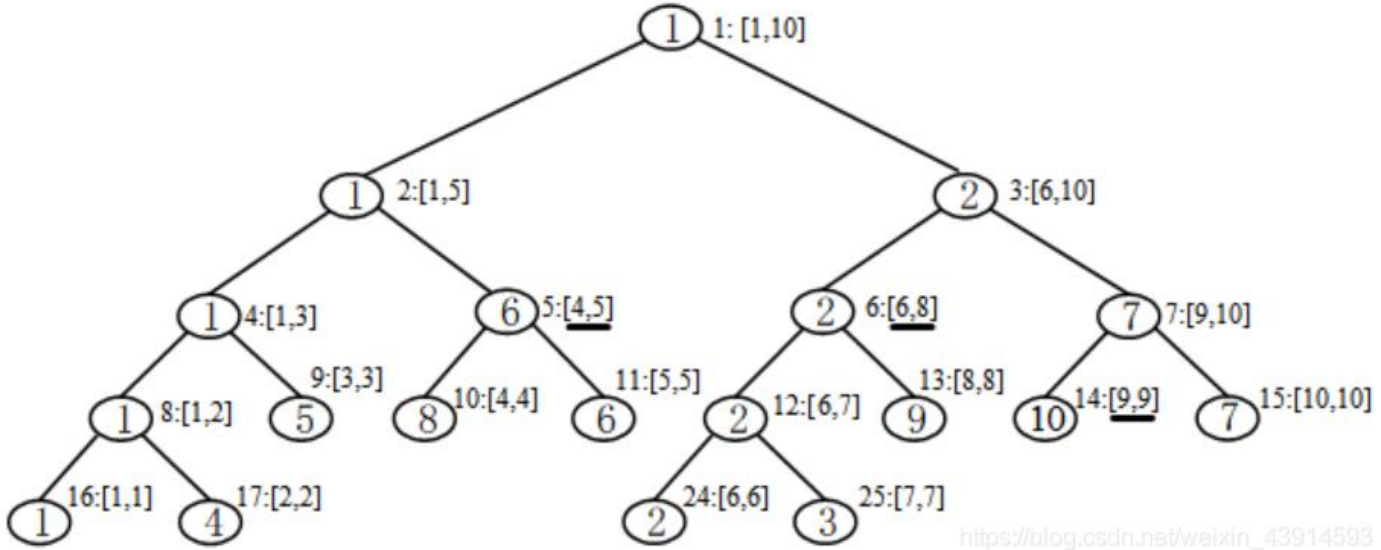


线段树——二叉树实现

- 编码时，可以定义标准的二叉树数据结构；在竞赛中一般用静态数组实现的满二叉树，虽然比较浪费空间，但是编码稍微简单一点。
- 下面给的代码，都用静态分配的tree[]。父结点和子结点之间的访问非常简单，缺点是最后一行有大量结点被浪费。
- 注意，二叉树的空间需要开MAXN*4，即元素数量的4倍，下面说明原因。假设有一棵处理n个元素（叶子结点有n个）的线段树，且它的最后一层只有1个叶子，其他层都是满的；如果用满二叉树表示，它的结点总数是：最后一层有2n个结点（其中2n - 1个都浪费了没用到），前面所有的层有2n个结点，共4n个结点。空间的浪费是二叉树的本质决定的：它的每一层都按2倍递增。
- 如果只需修改一个元素（单点修改），直接修改叶子结点上元素的值，然后从下往上更新线段树，操作次数也是O(logn)。如果修改的是一个区间的元素（区间修改），需要用到lazy-tag技术。

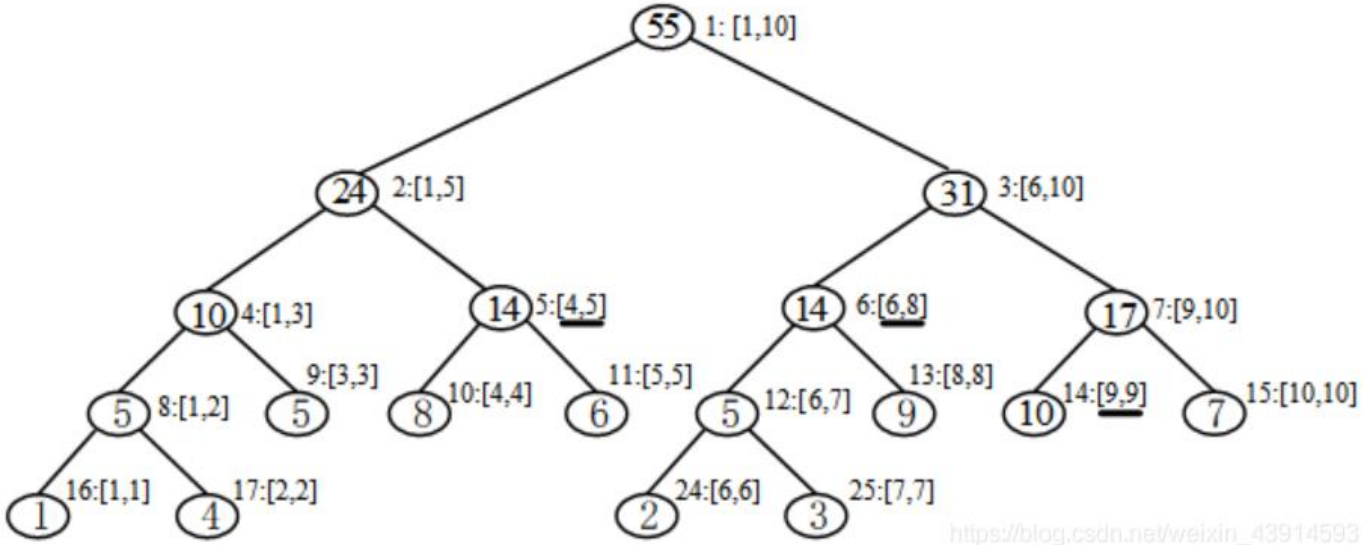
线段树—— 区间查询

- 2.1 查询最值
- 以数列{1, 4, 5, 8, 6, 2, 3, 9, 10, 7}为例。首先建立一棵用满二叉树实现的线段树，用于查询任意子区间的最小值。如下图所示，每个结点上圆圈内的数字是这棵子树的最小值。圆圈旁边的数字，例如根结点的"1:[1,10]"，1表示结点的编号，[1,10]是这个结点代表的元素范围，即第1到第10个元素。
- 查询任意区间[i, j]的最小值。例如查区间[4, 9]的最小值，递归查询到区间[4, 5]、[6, 8]、[9, 9]，见图中画横线的线段，得最小值min{6, 2, 10} = 2。查询在O(logn)时间内完成。读者可以注意到，在这种情况下，线段树很像一个最小堆。
- m次“单点修改+区间查询”的总复杂度是O(mlogn)。对规模100万的问题，也能轻松解决。



线段树—— 区间查询

- 查询区间和
- 首先建立一棵用于查询{1, 4, 5, 8, 6, 2, 3, 9, 10, 7}区间和的线段树，每个结点上圆圈内的数字是这棵子树的和。
- 例如查区间[4, 9]的和，递归查询到区间[4, 5]、[6, 8]、[9, 9]，见图中画横线的线段，得sum{14, 14, 10} = 38。查询在O(logn)的时间内完成。



线段树—— 区间查询代码

- 下面以查询区间[L, R]的和为例，给出代码。查询递归到某个结点p（p表示的区间是[pl, pr]）时，有2种情况：
- （1）[L, R]完全覆盖了[pl, pr]，即L ≤ pl ≤ pr ≤ R，直接返回p的值即可。
- （2）[L, R]与[pl, pr]部分重叠，分别搜左右子结点。L < pr，继续递归左子结点，例如查询区间[4, 9]，与第2个结点[1, 5]有重叠，因为4 < 5。R > pl，继续递归右子结点，例如[4, 9]与第3个结点[6, 10]有重叠，因为9 > 6。

```
int query(int L,int R,int p,int pl,int pr){
    if(L<=pl && pr<=R) return tree[p];      //完全覆盖
    int mid = (pl+pr)>>1;
    if(L<=mid) res+=query(L,R,ls(p),pl,mid);  //L与左子节点有重叠
    if(R>mid)  res+=query(L,R,rs(p),mid+1,pr); //R与右子节点有重叠
    return res;
}
//调用方式：query(L, R, 1, 1, n)
```

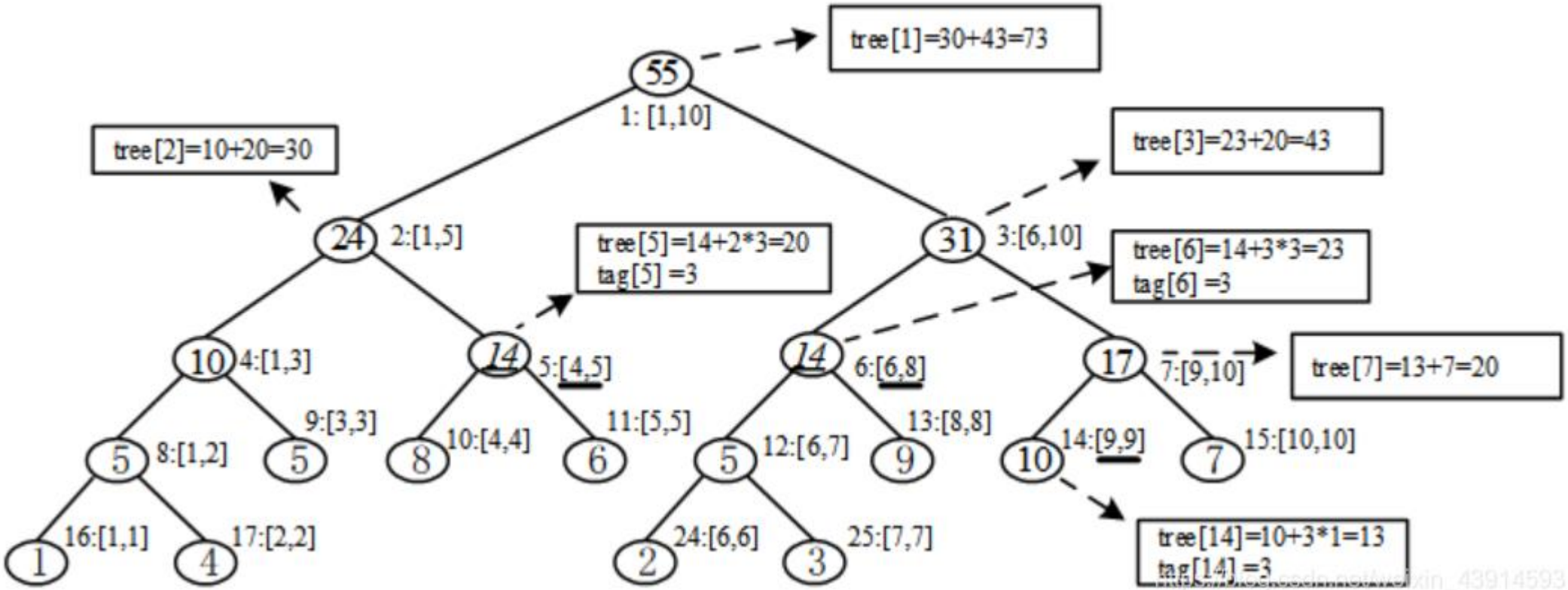
线段树—— 区间操作与lazy-tag

- 这里介绍线段树的核心技术“lazy-tag”，并给出“区间修改+区间查询”的模板。
- 用线段树求解洛谷 3372。
- 区间修改比单点修改复杂很多。最普通区间修改，例如对一个数列的[L, R]区间内每个元素统一加上d，如果在线段树上，一个个地修改这些元素，那么m次区间修改的复杂度是O(mnlogn)的。
- 解决的办法很容易想到，还是利用线段树的特征：线段树的结点tree[i]，记录了i这个区间的值。那么可以再定义一个tag[i]，用它统一记录i这个区间的修改，而不是一个个地修改区间内的每个元素，这个办法被称为“lazy-tag”。

线段树—— 区间操作与lazy-tag

- lazy-tag（懒惰标记，或者延迟标记）。当修改的是一个线段区间时，就只对这个线段区间进行整体上的修改，其内部每个元素的内容先不做修改，只有当这个线段区间的一致性被破坏时，才把变化值传递给下一层的子区间。每次区间修改的复杂度是O(logn)的，一共m次操作，总复杂度是O(mlogn)的。区间i的lazy操作，用tag[i]记录。
- 下面举例说明区间修改函数update()的具体步骤。例如把[4, 9]区间内的每个元素加3，执行步骤是：
- （1）左子树递归到结点5，即区间[4, 5]，完全包含在[4, 9]内，打标记tag[5] = 3，更新tree[5]为20，不再继续深入；
- （2）左子树递归返回，更新tree[2]为30；
- （3）右子树递归到结点6，即区间[6, 8]，完全包含在[4, 9]内，打标记tag[6]=3，更新tree[6]为23。
- （4）右子树递归到结点14，即区间[9, 9]，打标记tag[14]=3，更新tree[14]=13；
- （5）右子树递归返回，更新tree[7]=20；继续返回，更新tree[3]=43；
- （6）返回到根节点，更新tree[1]=73。

线段树—— 区间操作与lazy-tag



线段树—— 区间操作与lazy-tag

- push_down()函数。在进行多次区间修改时，一个结点需要记录多个区间修改。而这些区间修改往往有冲突，例如做2次区间修改，一次是[4, 9]，一次是[5, 8]，它们都会影响5:[4, 5]这个结点。第一次修改[4, 9]覆盖了结点5，用tag[5]做了记录；而第二次修改[5, 8]不能覆盖结点5，需要再向下搜到结点11:[5, 5]，从而破坏了tag[5]，此时原tag[5]记录的区间统一修改就不得不往它的子结点传递和执行了，传递后tag[5]失去了意义，需要清空。所以lazy-tag的主要操作是解决多次区间修改，用push_down()函数完成。它首先检查结点p的tag[p]，如果有值，说明前面做区间修改时给p打了tag标记，接下来就把tag[p]传给左右子树，然后把tag[p]清零。
- 　　push_down()函数不仅在“区间修改”中用到，在“区间查询”中同样用到。
- 　　下面给出洛谷P3372的线段树代码，它是“区间修改+区间查询”的模板题。
- 　　注意代码中对二叉树的操作，特别是反复用到的变量pl和pr，它们是结点p所指向的原数列的区间位置[pl, pr]。p是二叉树的某个结点，范围是1 ≤ p ≤ MAXN*4；而pl、pr的范围是1 ≤ pl, pr ≤ n，n是数列元素的个数。用满二叉树实现线段树时，一个结点p所指向的[pl, pr]是确定的，也就是说，给定p，可以推算出它的[pl, pr]。

线段树—— 例题1

- Can you answer these queries? hdu 4027
- 题目描述：把区间内的每个数开平方；输出区间查询。
  - 输入：有很多测试用例。每个用例的第一行是一个整数N，表示有N个数，1<=N<=100000。第二行包括N个整数Ei，Ei < 2^63
  - 。第三行是整数M，表示M个操作，1<=M<=100000。后面有M行，每行有三个整数T、X、Y，T=0表示修改，把[X,Y]区间的每个数开平方（开方结果向下取整）。T=1是查询，求[X, Y]的区间和。
- 输出：对每个用例，打印用例编号，然后对每个操作打印一行。

线段树—— 例题1

- 标准的线段树区间修改是区间加或者区间最值，本题的修改是把区间每个数开方。
- 　　如果按正规的区间修改，用lazy-tag标记区间，很难编码。注意本题的关键是开方计算，而一个数最多经过7次开方就变成了1，1继续开方仍保持1不变。利用这个特点，再结合线段树编码。
- 　　编码：（1）一个区间内如果有数的开方结果不是1，则单独计算它；（2）一个区间的所有数减为1后，标记lazy-tag = 1，后面不再对这个区间做开方操作。具体编码的时候，不一定用到lazy-tag，直接判断区间和即可，如果区间和等于子树的叶子数，说明叶子的值都是1。
- 　　复杂度：对N个数每个数开方7次，共7N次；再做M次修改和区间查询，复杂度O(MlogN)。总复杂度符合要求。

线段树—— 例题2

Transformation hdu 4578

题目描述：有n个整数，执行多种区间操作：

add修改：区间内每个数加上c；

multi修改：区间内每个数乘以c；

change修改：区间内每个数统一改成c；

sum求和：对区间内每个数的p次方求和，

输出结果。1 <= p <= 3，即求和、平方和、立方和。

线段树—— 例题2

此题有多种操作，每个操作如果单独编码，并不太难。但是题目要求同时做三种修改，并有三种求和询问，非常麻烦。

三种修改add、multi、change，在结点上分别用三个lazy-tag标记。三个标记的关系是：

（1）做change时，原有的add和multi标记失效；

（2）做multi时，如果原有add标记，把add改为add*multi；

（3）pushdown时，先执行add，再multi，最后add。

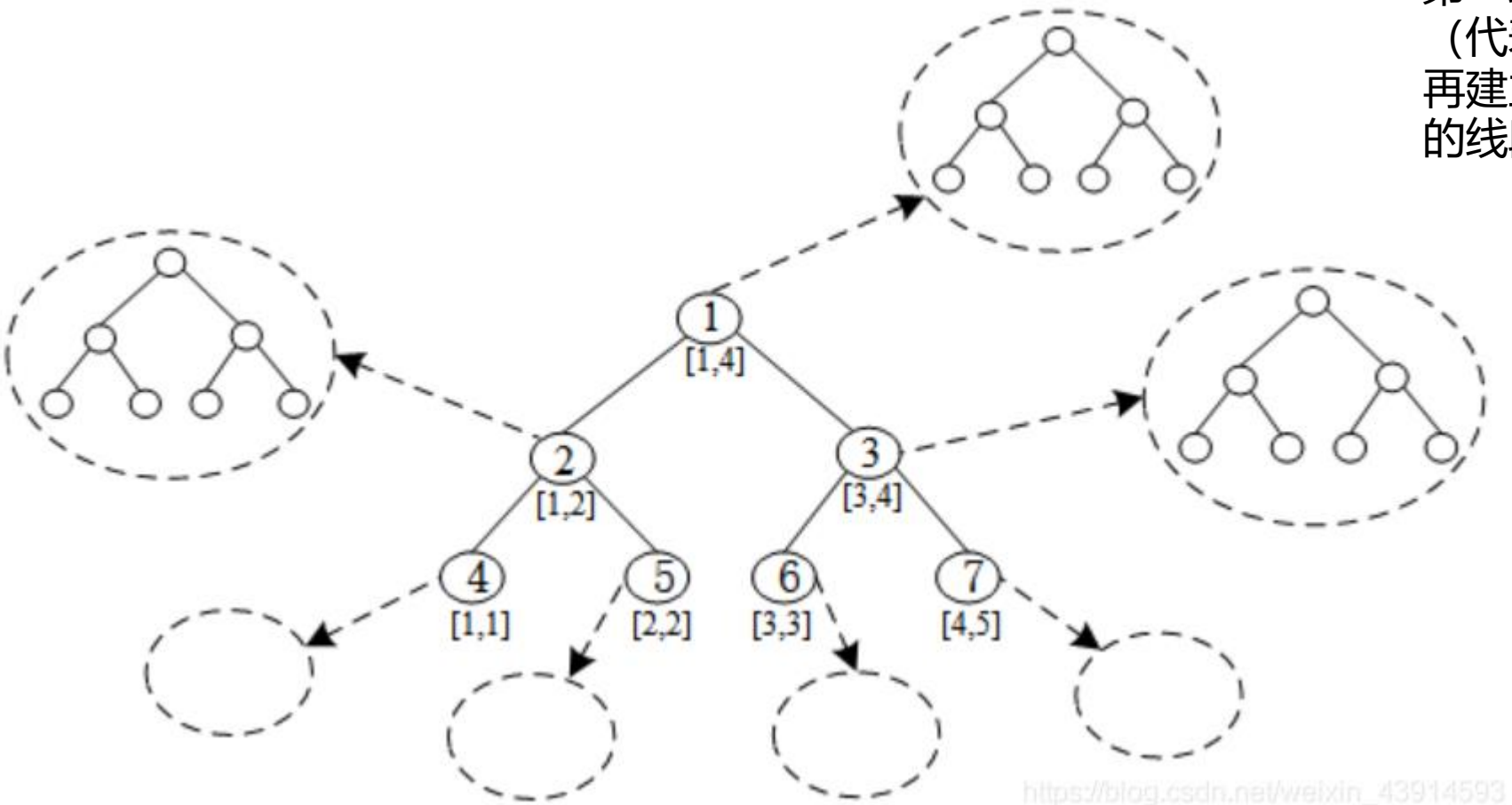
三种询问：求和sum1、平方和sum2、立方和sum3。对于change和multi标记，三种询问都容易计算。对于add标记，求和很容易，而平方和、立方和需要推理。

（1）平方和。(a + c)^2 = a^2 + c^2 + 2ac ，即sum2[new] = sum2[old] + (R - L + 1)×c×c + 2×sum1×c，其中[L, R]是区间，sum2[new]和sum2[old]分别表示sum2的新值和旧值。

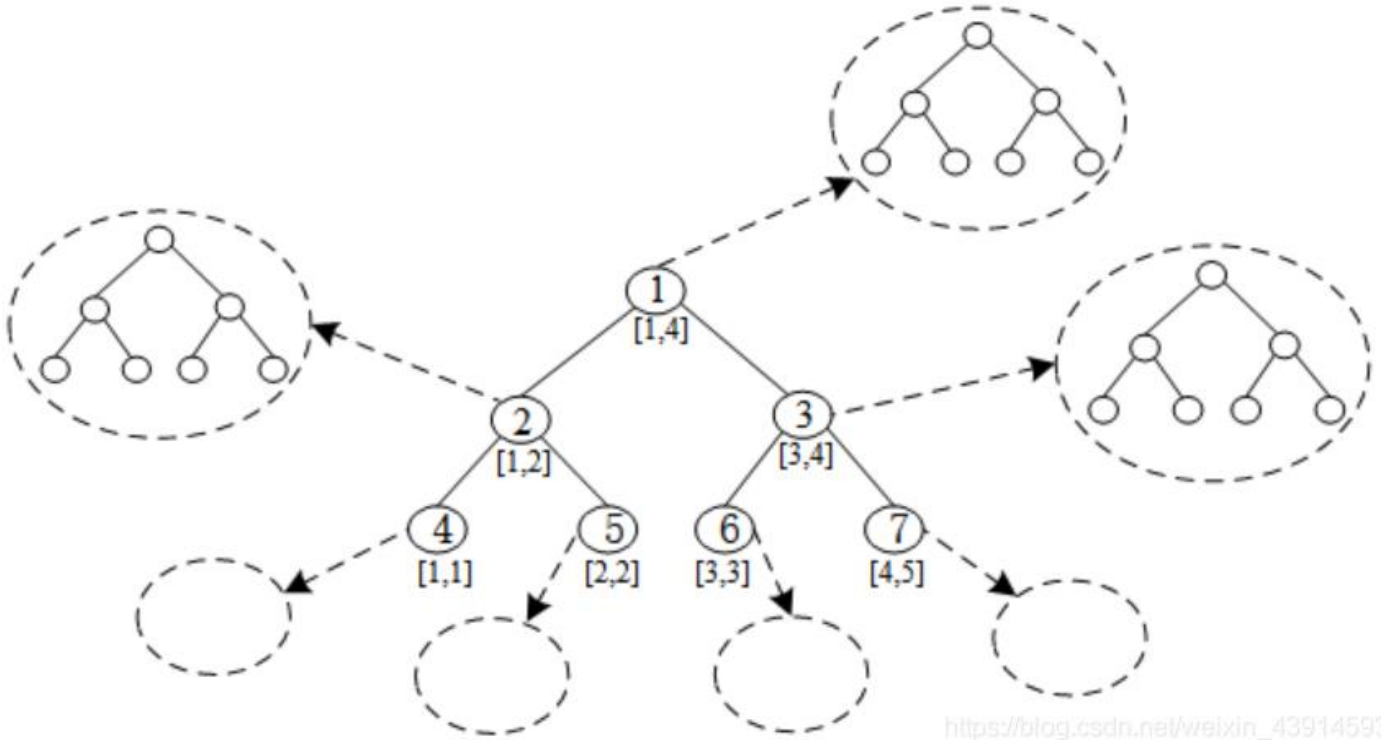
（2）立方和。(a + c)^3 = a^3 + c^3 + 3c(a^2 + ac)，即sum3[new] = sum3[old] + c×c×c + 3×c×(sum2[old] + sum[old]×c)。

二维线段树

这不是一种平面二维几何的关系，而是“树套树”的结构。第一维线段树上的每个结点（代表了一个区间），都单独再建立一棵线段树，即第二维的线段树。



二维线段树



中间是第一维线段树，有7个结点（4个叶子），每个结点单独扩展一个线段树，见虚线圆圈，是第二维线段树。从这个图可以看出，它很耗费空间。设第一维有u个元素，建树需要4u个结点；第二维有v个元素，4v个结点；总结点数量是16uv。

二维线段树如何使用？设题目有两个限制条件x、y，那么用x建立第一维线段树，用y建立第二维线段树。查询同时满足两个区间[xL, xR]、[yL, yR]，首先在第一维线段树上查询区间[xL, xR]，找到符合条件的第一维结点，然后再查询第二维的线段树，找到[yL, yR]。显然，一次查询的复杂度是O(logu·logv)的。

Luck and Love hdu 1823

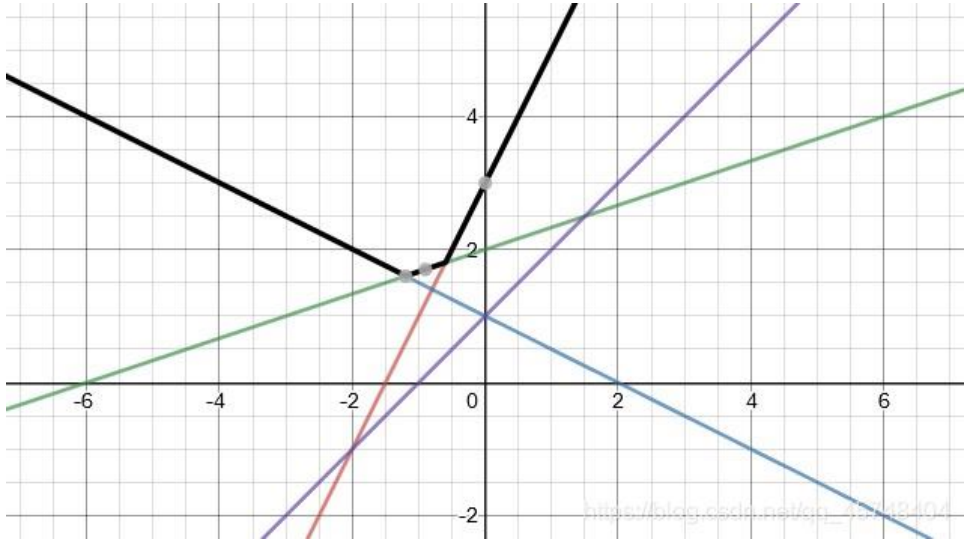
- 题目描述：小w征婚，收到很多MM报名，小w想找到最有缘分的MM。
- 输入：有多个测试，第一个数字t，表示有t个操作，当t = 0时终止，接下来每行是一个操作。
- 操作"I"，后面是一个MM的三个参数：整数H表示身高，浮点数A表示活泼度，浮点数L表示缘分。
- 操作"Q"，后面跟着四个浮点数，H1、H2表示身高区间，A1、A2表示活泼度区间，输出符合身高和活泼度要求的MM中的缘分最高值。
- 输出：对每次询问，输出缘分最高值，若没有合适的，输出-1。
- 数据范围：100<=H1，H2<=200， 0.0<=A1，A2，L<=100.0

Luck and Love hdu 1823

暴力法：一次询问，逐个检查n个MM，找出符合身高和活泼度的，并记录其中缘分最大值，复杂度是O(n)；m次询问，复杂度是O(mn)。

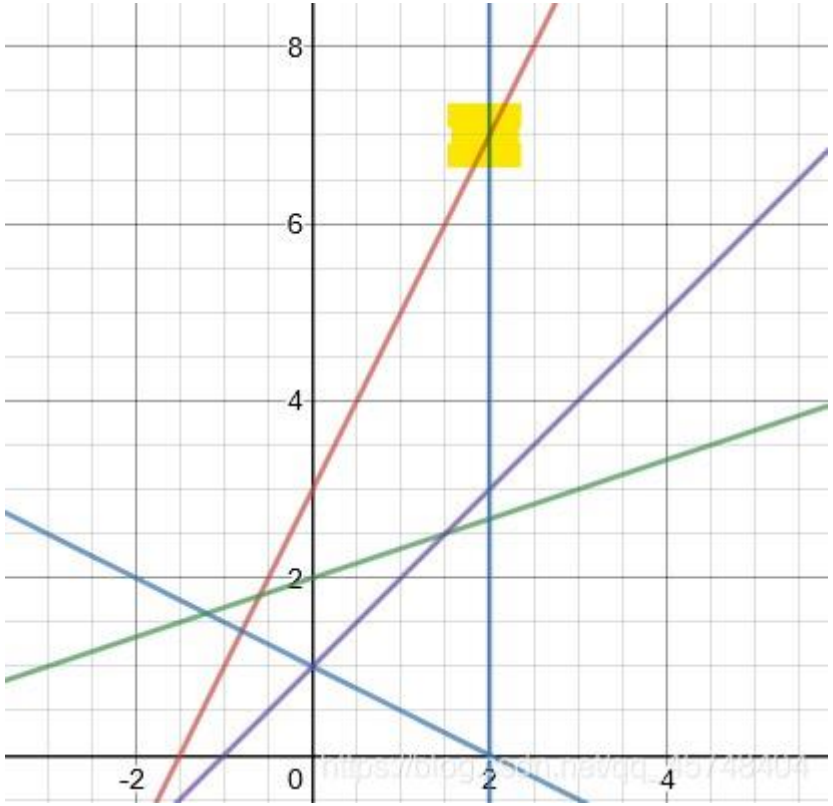
用“单点修改+区间查询”的二维线段树求解。二维线段树，第一维线段树是身高，第二维是活泼度。另外定义s[][]记录最大缘分，s[i][j]表示第一维结点i、第二维结点j的最大缘分；因为结点i和j分别是身高区间和活泼度区间，所以查询适合的s[][]就得到了答案。复杂度 O(m(logn)^2)

李超线段树



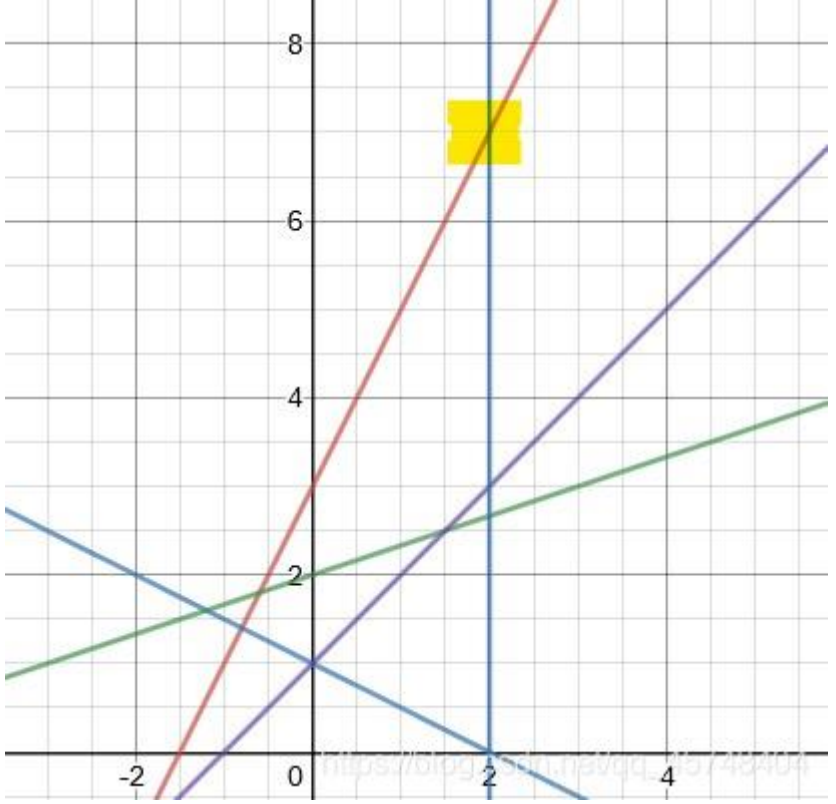
- **什么是李超**线段树
- 先以一个问题引入：
  在平面上有两种操作（强制在线）：
  插入一条表达式为 L : y = k*x+b 的直线，给出 k ,b 。
  给出 t，求当前所有直线中与直线 x = t 交点的纵坐标最大是多少
  直线取 max 应该是得到一个下凸包，像这样（黑色的）：

李超线段树



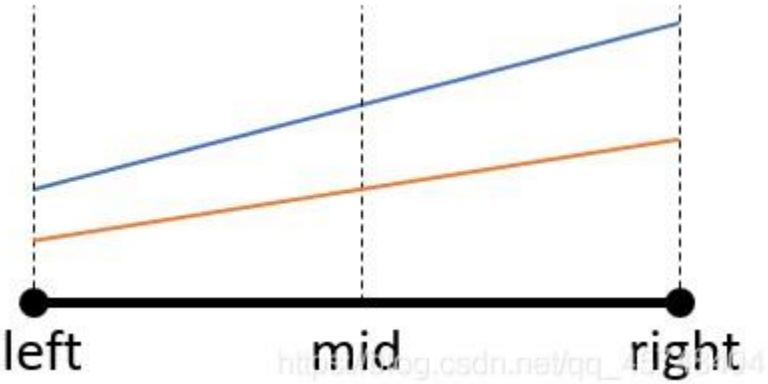
- 李超线段树是什么?
- 线段树，当然是要维护区间 [L , R]
- 的一个信息，李超线段树维护的就是
- 区间中点处最高的直线
  比如下图，区间 [0 , 4] 的中点 x = 2 的最高直线为红色直线

李超线段树



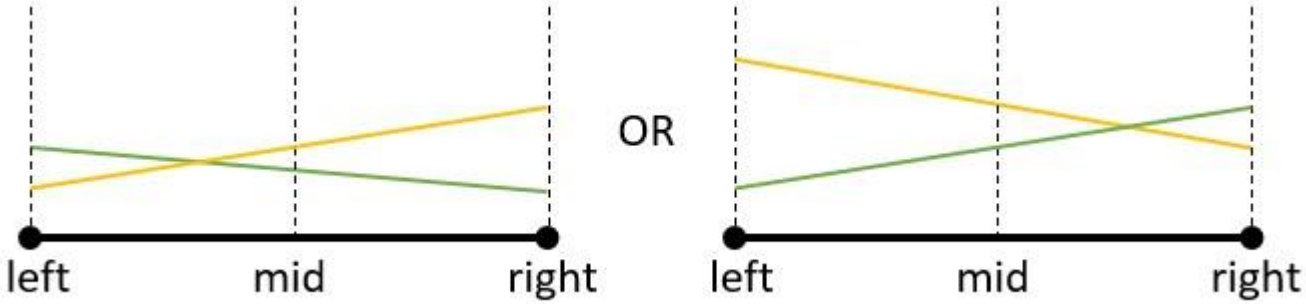
- 李超线段树是什么?
- 线段树，当然是要维护区间 [L , R]
- 的一个信息，李超线段树维护的就是
- 中点处最高的直线
  比如下图，区间 [0 , 4] 的中点 x = 2 的最高直线为红色直线
- 利用这一信息，可以在 O(log n) 的复杂度内插入直线、查询与 x = t 相交的最高点。（这里的 n 指的是查询的 t 的值域范围）

实现李超线段树



- 插入直线
- 李超线段树有一个非常重要的思想：标记永久化。
- 于是我们可以这样实现李超线段树，假如要把直线 L 插入某一区间：
- 1、如果当前区间还没有直线标记（不代表没有直线覆盖，因为标记永久化），那么就把标记记为L；
- 2、如果 L 完全覆盖原有线段，像这样（蓝色是 L，红色是原有线段）：那么 x = t 与 [L,R] 的交点一定高于原线段，于是把原有线段直接替换为 L；

实现李超线段树



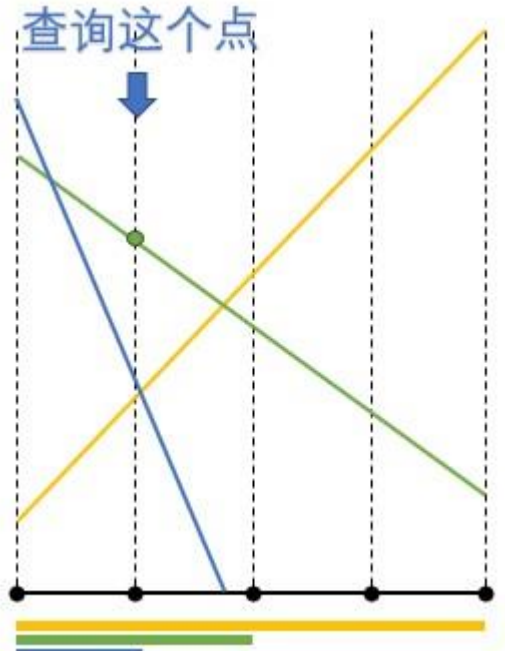
- 3、如果 L 被原有线段完全覆盖（就是2.反过来），那么 L 一定没有原有线段优；这样的话就舍弃 L 不做任何更新。
- 4、如果 L 和原有线段在 [L , R] 中有交点，像这样：

这种情况就无法确定 与哪条直线交点更高。根据李超线段树维护的信息，我们需要在两条直线中取与 x = mid 交点更高的一条直线作为标记（也就是上图的黄色直线，剩下的那条就是绿色直线）。再判断交点：如果在左区间，那么绿色直线在左区间可能比黄色直线高，然后就尝试把绿色直线往左区间插入；右区间类似。

实现李超线段树

查询

由于是标记永久化，查询就比较类似于标记永久化的线段树。比如查询 x = pos 的最高交点，那么就要从线段树一层层递归，直到递归到 [pos , pos] 这个区间——把每个区间的最优线段的交点取 max。



K-D Tree

- k-D Tree(KDT , k-Dimension Tree) 是一种可以高效处理K维空间信息的数据结构。

- 在结点数n远大于2^k时，应用 k-D Tree 的时间效率很好。

- 在算法竞赛的题目中，一般k=2有 。在后面分析时间复杂度时，将认为k是常数。

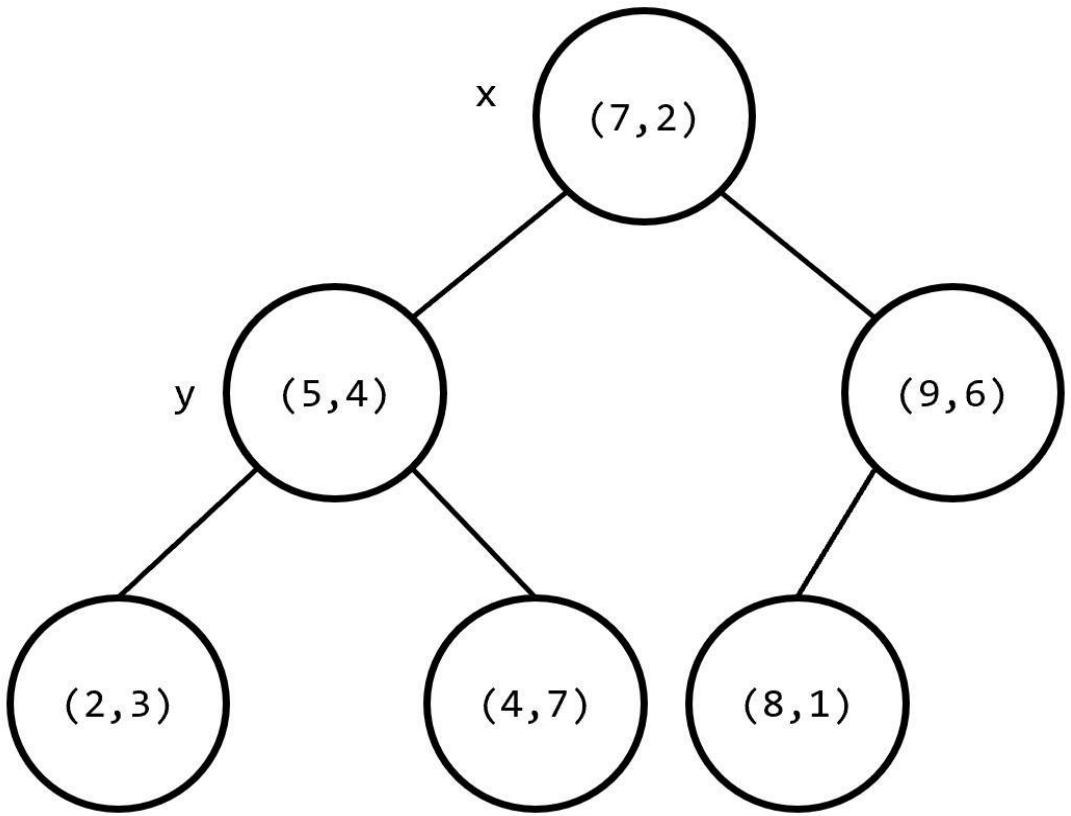
K-D Tree

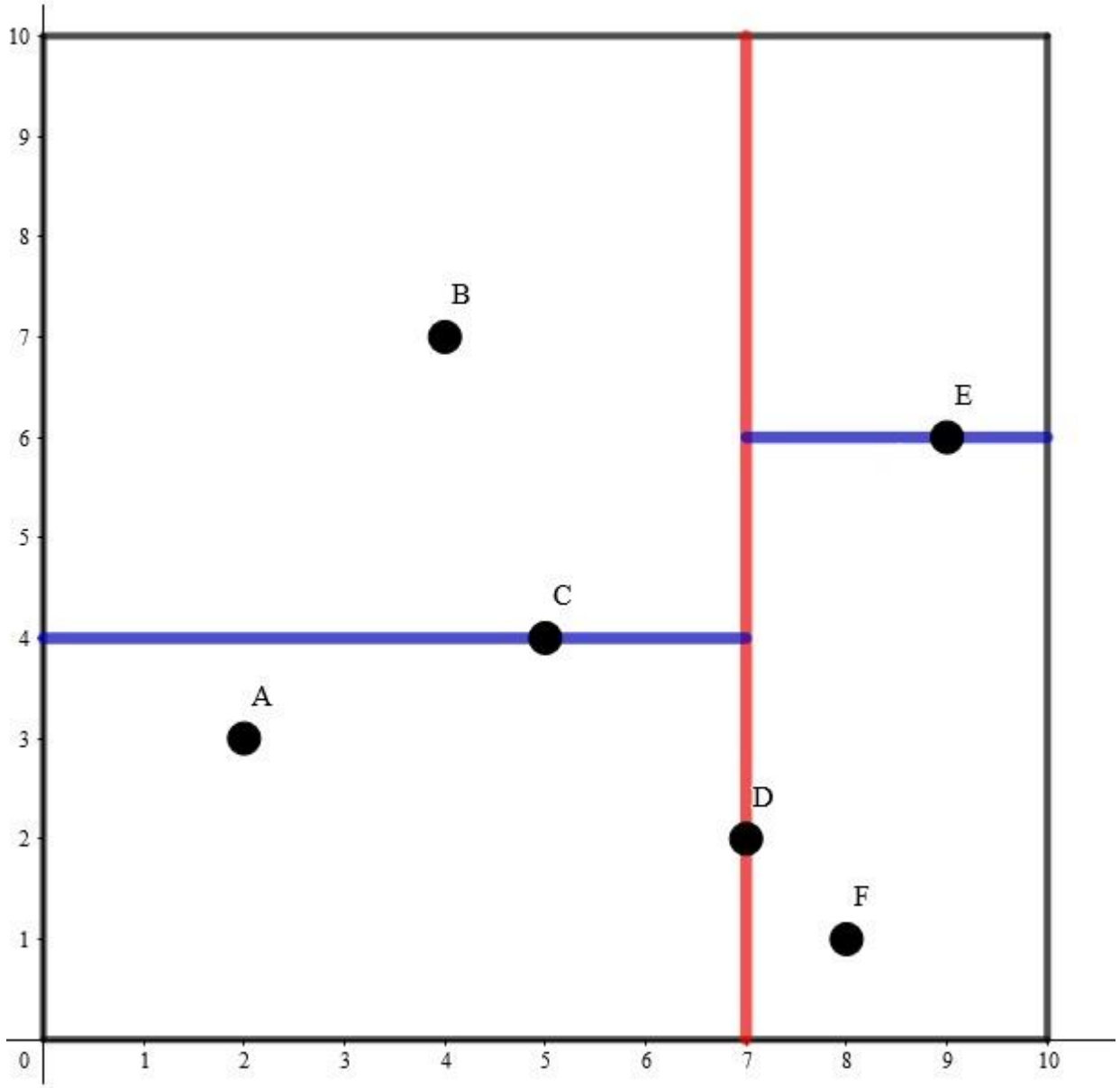
- k-D Tree 具有二叉搜索树的形态，二叉搜索树上的每个结点都对应k维空间内的一个点。其每个子树中的点都在一个k维的超长方体内，这个超长方体内的所有点也都在这个子树中。
- 假设我们已经知道了k维空间内的 n个不同的点的坐标，要将其构建成一棵 k-D Tree，步骤如下：
  - 若当前超长方体中只有一个点，返回这个点。
  - 选择一个维度，将当前超长方体按照这个维度分成两个超长方体。
  - 选择切割点：在选择的维度上选择一个点，这一维度上的值小于这个点的归入一个超长方体（左子树），其余的归入另一个超长方体（右子树）。
  - 将选择的点作为这棵子树的根节点，递归对分出的两个超长方体构建左右子树，维护子树的信息。

K-D Tree

为了方便理解，我们举一个 时的例子。

其构建出 k-D Tree 的形态可能是这样的：

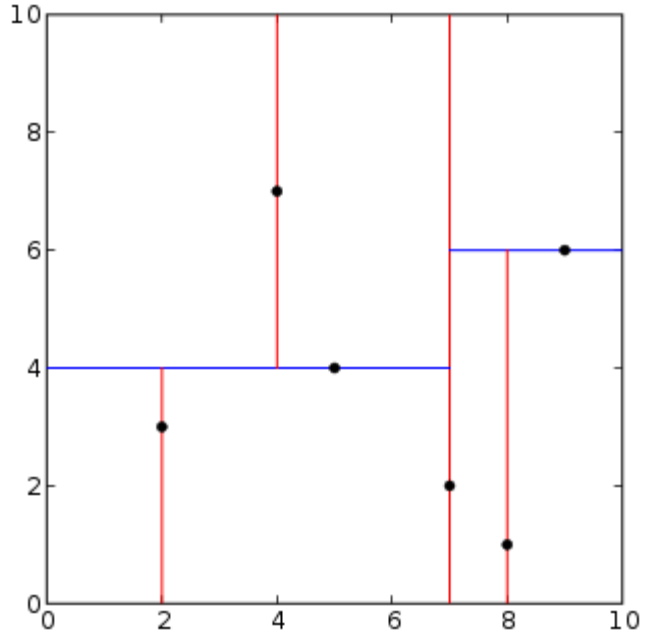




K-D Tree

- 其中树上每个结点上的坐标是选择的分割点的坐标，非叶子结点旁的x或y是选择的切割维度。
- 这样的复杂度无法保证。对于2,3两步，我们提出两个优化：
  - 选择的维度要满足其内部点的分布的差异度最大，即每次选择的切割维度是方差最大的维度。
  - 每次在维度上选择切割点时选择该维度上的 中位数，这样可以保证每次分成的左右子树大小尽量相等。
- 可以发现，使用优化2后，构建出的 k-D Tree 的树高最多为O(logn)。

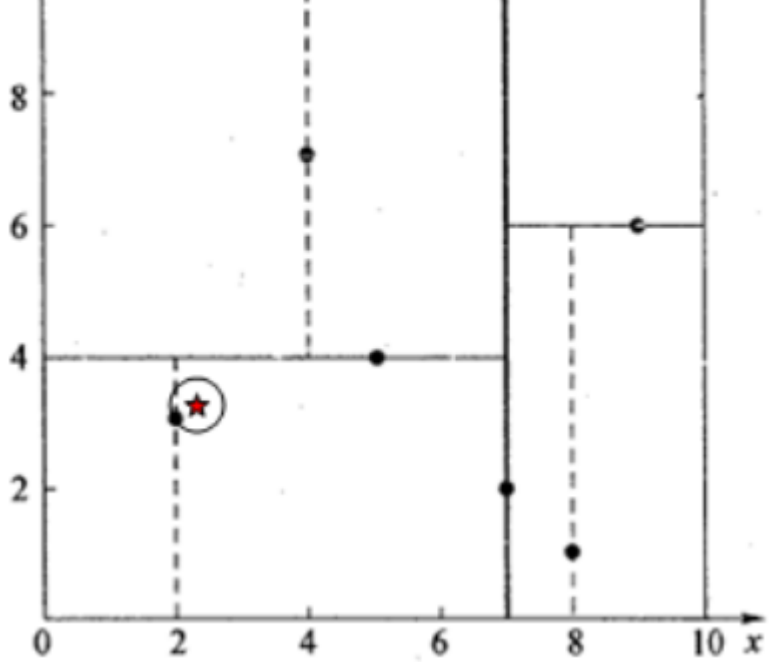
K-D Tree的建立



- 建立Kdtree实际上是一个不断划分的过程，首先选择最稀疏的维度
- （一般通过计算数据在各个维度的方差，选择方差大的作为本次分割维度）点，垂直该维度做第一次划分。此时k维超平面被一分为二，

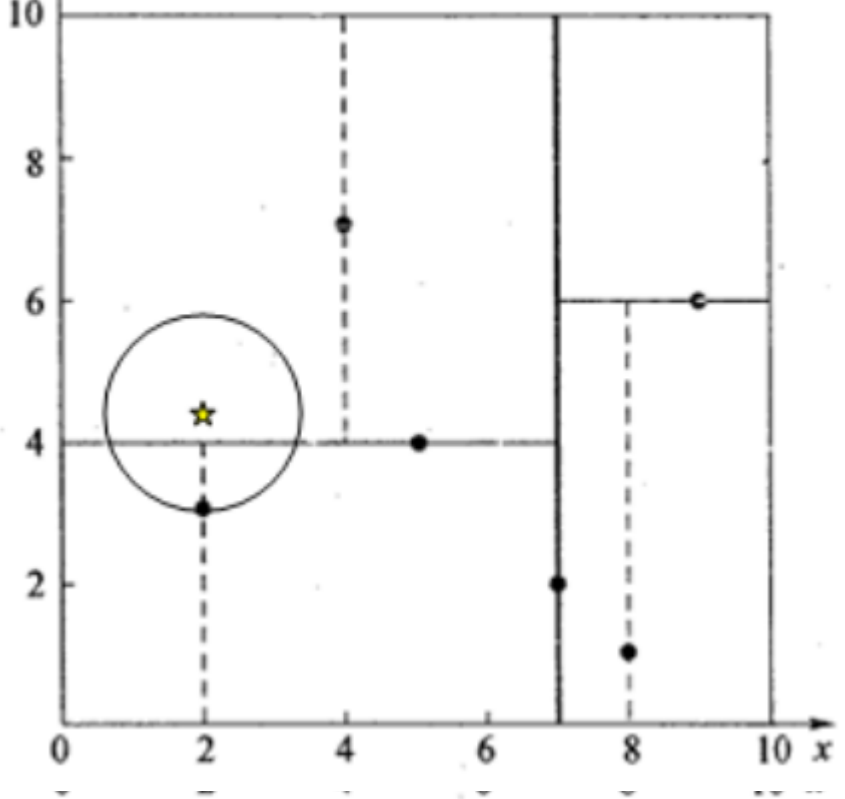
  在两个子平面中再找最稀疏的维度，以此类推直到最后一个点也被划分，那么就形了一个不断二分的树。
- 二维Kdtree的建立过程如图2所示，首先分别计算x，y方向上数据的方差，得知x方向上的方差最大，所以split域值首先x轴方向；然后根据x轴方向的值2,5,9,4,8,7排序选出中值为7，所以Node-data = （7,2）。这样，该节点的分割超平面就是通过（7,2）并垂直于split = 0（x轴）的直线x = 7，后面以此类推。

K-D Tree的查询



- 搜索一般有：
- 1. 搜索距离点一定半径范围内的所有点；
- 2. 搜索距离点最近的k个点。
- 星号表示要查询的点（2.1,3.1），通过二叉搜索，顺着搜索路径很快就能找到最邻近的近似点，也就是叶子节点（2,3）。而找到的叶子节点并不一定就是最邻近的，最邻近肯定距离查询点更近，应该位于以查询点为圆心且通过叶子节点的圆域内。为了找到真正的最近邻，还需要进行'回溯'操作：算法沿搜索路径反向查找是否有距离查询点更近的数据点。　　此例中先从（7,2）点开始进行二叉查找，然后到达（5,4），最后到达（2,3），此时搜索路径中的节点为<（7,2），（5,4），（2,3）>，首先以（2,3）作为当前最近邻点，计算其到查询点（2.1,3.1）的距离为0.1414，然后回溯到其父节点（5,4），并判断在该父节点的其他子节点空间中是否有距离查询点更近的数据点。以（2.1,3.1）为圆心，以0.1414为半径画圆，如图4所示。发现该圆并不和超平面y = 4交割，因此不用进入（5,4）节点右子空间中去搜索。再回溯到（7,2），以（2.1,3.1）为圆心，以0.1414为半径的圆更不会与x = 7超平面交割，因此不用进入（7,2）右子空间进行查找。至此，搜索路径中的节点已经全部回溯完，结束整个搜索，返回最近邻点（2,3），最近距离为0.1414。

K-D Tree的查询



- 一个复杂点了例子如查找点为（2，4.5），同样先进行
- 二叉查找，先从（7,2）查找到（5,4）节点，在进行查找时
- 是由y = 4为分割超平面的，由于查找点为y值为4.5，因此进入
- 右子空间查找到（4,7），形成搜索路径<（7,2），（5,4），
- （4,7）>，取（4,7）为当前最近邻点，计算其与目标查找
- 点的距离为3.202。然后回溯到（5,4），计算其与查找点之间的距离为3.041。以（2，4.5）为圆心，以3.041为半径作圆，如图5所示。可见该圆和y = 4超平面交割，所以需要进入（5,4）左子空间进行查找。此时需将（2,3）节点加入搜索路径中得<（7,2），（2,3）>。回溯至（2,3）叶子节点，（2,3）距离（2,4.5）比（5,4）要近，所以最近邻点更新为（2，3），最近距离更新为1.5。回溯至（7,2），以（2,4.5）为圆心1.5为半径作圆，并不和x = 7分割超平面交割，如图所示。至此，搜索路径回溯完，返回最近邻点（2,3），最近距离1.5。