

北京大学暑期课 《ACM/ICPC竞赛训练》

北京大学信息学院 郭炜

guo_wei@PKU.EDU.CN

<http://weibo.com/guoweiofpku>

课程网页: http://acm.pku.edu.cn/summerschool/pku_acm_train.htm

线段树和树状数组

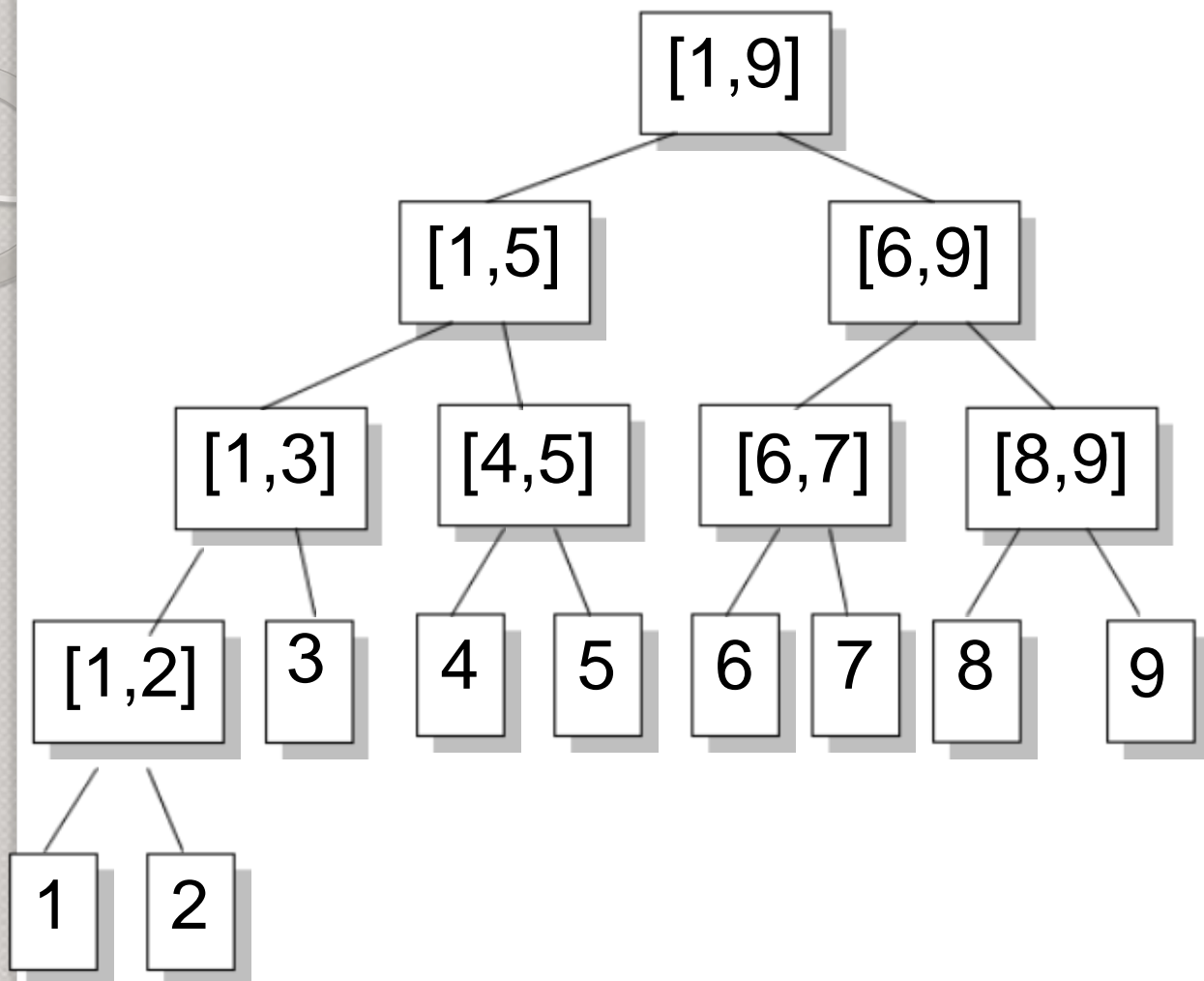
北京大学信息学院 郭炜

线段树(Interval Tree)

- 实际上还是称为区间树更好理解一些。
- 树：是一棵树，而且是一棵二叉树。
- 线段：树上的每个节点对应于一个线段(还是叫“区间”更容易理解，区间的起点和终点通常为整数)
- 同一层的节点所代表的区间，相互不会重叠。同一层节点所代表的区间，加起来是个连续的区间。
- 叶子节点的区间是单位长度，不能再分了。

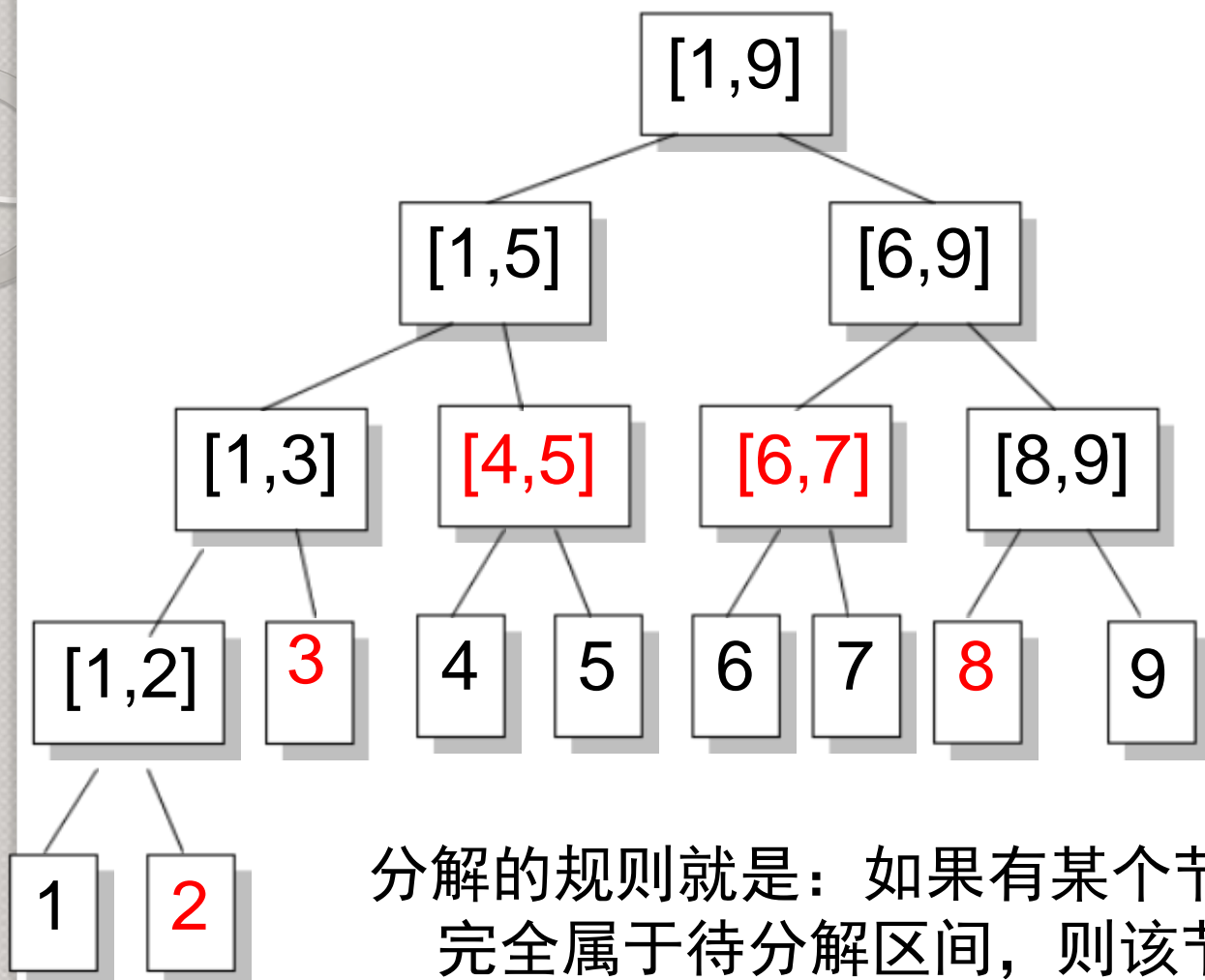
- 线段树是一棵二叉树，树中的每一个结点表示了一个区间 $[a,b]$ 。 a,b 通常是整数。每一个叶子节点表示了一个单位区间(长度为1)。对于每一个非叶结点所表示的结点 $[a,b]$ ，其左儿子表示的区间为 $[a,(a+b)/2]$ ，右儿子表示的区间为 $[(a+b)/2+1,b]$ (除法去尾取整)。

- 区间[1, 9]的线段树



- 每个区间的长度是区间内整数的个数
- 叶子节点长度为1，不能再往下分
- 若一个节点对应的区间是 $[a,b]$,则其子节点对应的区间分别是 $[a,(a+b)/2]$ 和 $[(a+b)/2+1,b]$ （除法去尾取整）
- 线段树的平分构造，实际上是用二分的方法。若根节点对应的区间是 $[a,b]$,那么它的深度为 $\log_2(b-a+1) + 1$ （向上取整）。
- 叶子节点的数目和根节点表示区间的长度相同.
- 线段树节点要么0度，要么2度，因此若叶子节点数目为 N ，则线段树总结点数目为 $2N-1$

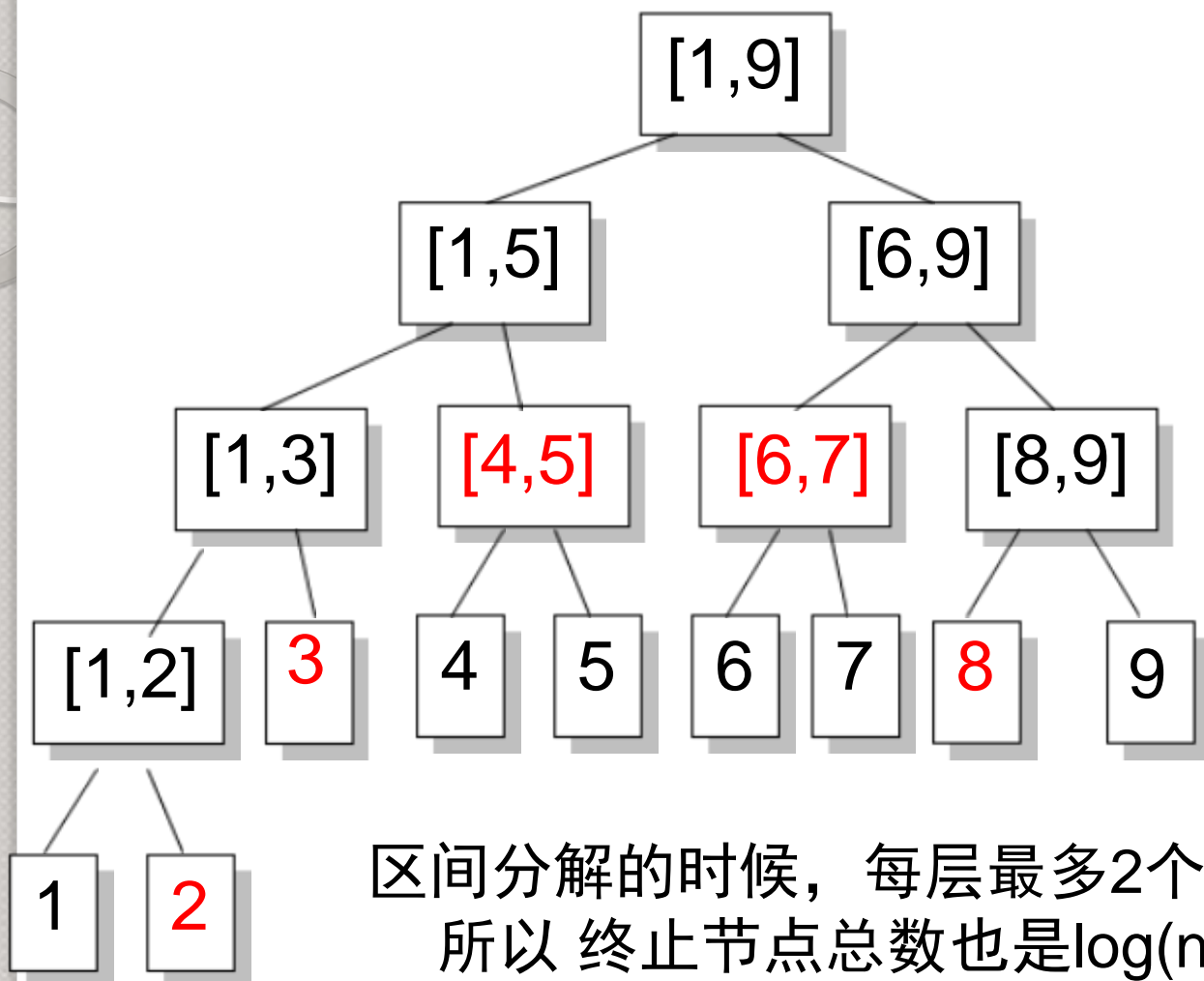
- 区间[1, 9]的线段树上，区间[2,8]的分解



分解的规则就是：如果有某个节点代表的区间，完全属于待分解区间，则该节点为“终止”节点，不再继续往下分解

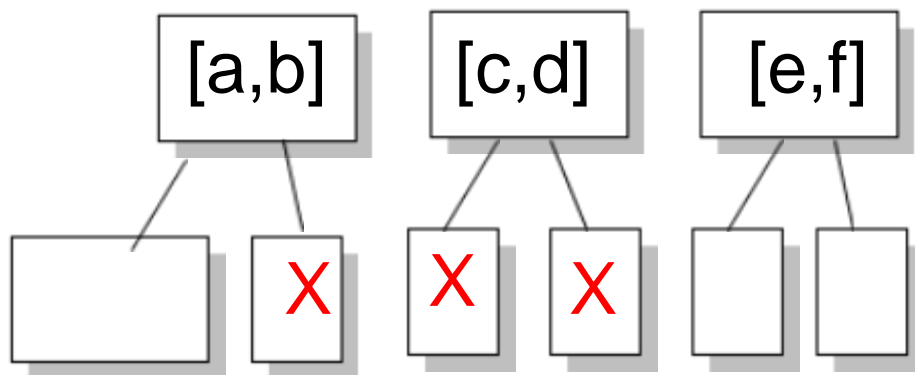
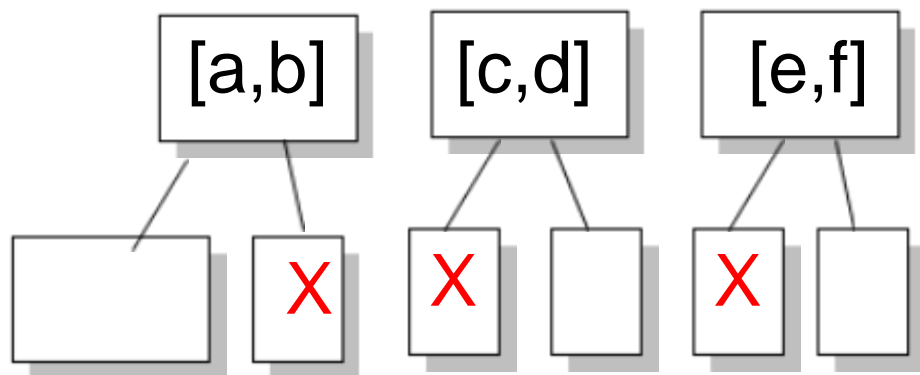
所有“终止”节点所代表的区间都不重叠，且加在一起就恰好等于整个待分解区间

- 区间[1, 9]的线段树上，区间[2,8]的分解



区间分解的时候，每层最多2个“终止节点”，
所以 终止节点总数也是 $\log(n)$ 量级的

- 证明每层最多2个“终止节点”：



- X**代表终止节点。上述情况不可能发生

线段树的特征

- 1、线段树的深度不超过 $\log_2(n)+1$ （向上取整， n 是根节点对应区间的长度）。
- 2、线段树上，任意一个区间被分解后得到的“终止节点”数目都是 $\log(n)$ 量级。

线段树上更新叶子节点和进行区间分解时间复杂度都是 $O(\log(n))$ 的

- 这些结论为线段树能在 $O(\log(n))$ 的时间内完成插入数据，更新数据、查找、统计等工作，提供了理论依据

线段树的构建

- function 以节点 v 为根建树、 v 对应区间为 $[l, r]$
- {
- 对节点 v 初始化
- if ($l \neq r$)
- {
- 以 v 的左孩子为根建树、区间为 $[l, (l+r)/2]$
- 以 v 的右孩子为根建树、区间为 $[(l+r)/2+1, r]$
- }
- }
- 建树的时间复杂度是 $O(n)$ n 为根节点对应的区间长度

线段树的基本用途

- 线段树适用于和区间统计有关的问题。比如某些数据可以按区间进行划分，按区间动态进行修改，而且还需要按区间多次进行查询，那么使用线段树可以达到较快查询速度。

线段树应用举例

- 给你一个数的序列 $A_1A_2\cdots A_n$ 。并且可能多次进行下列两个操作：
 - 1、对序列里面的某个数进行加减
 - 2、询问这个序列里面任意一个连续的子序列 $A_iA_{i+1}\cdots A_j$ 的和是多少。
- 希望第2个操作每次能在 $\log(n)$ 时间内完成

线段树应用举例

- 显然, $[1, n]$ 就是根节点对应的区间
- 可以在每个节点记录该节点对应的区间里面的数的和Sum。
- 对于操作1: 因为序列里面 A_i 最多只会被线段树的 $\log(n)$ 个节点覆盖。只要求对线段树覆盖 A_i 的节点的Sum进行加操作, 因此复杂度是 $\log(n)$
- 对于操作2: 同样只需要找到区间所覆盖的“终止”节点, 然后把所找“终止”节点的Sum累加起来。因为这些节点的数量是 $O(\log(n))$ 的, 所以这一步的复杂度也是 $\log(n)$

线段树应用举例

- 如果走到节点 $[L,R]$ 时，如果要查询的区间就是 $[L,R]$
- (求 A_L 到 A_R 的和) 那么直接返回该节点的Sum，并累加到总的和上；
- 如果不是，则：
- 对于区间 $[L,R]$ ，取 $mid = (L+R) / 2$ ；
- 然后看要查询的区间与 $[L,mid]$ 或 $[mid+1,R]$ 哪个有交集，就进入哪个区间进行进一步查询。
- 因为这个线段树的深度最深的 $\log N$ ，所以每次遍历操作都在 $\log N$ 的范围内完成。但是常数可能很大。

线段树应用举例

- 如果是对区间所对应的一些数据进行修改，过程和查询类似。
- 用线段树解题，关键是要想清楚每个节点要存哪些信息（当然区间起终点，以及左右子节点指针是必须的），以及这些信息如何高效更新，维护，查询。不要一更新就更新到叶子节点，那样更新效率最坏就可能变成 $O(n)$ 的了。
- 先建树，然后插入数据，然后更新，查询。

例题： POJ 3264 Balanced Lineup

给定 Q ($1 \leq Q \leq 200,000$)个数 $A_1, A_2 \dots A_Q$ ，
多次求任一区间 $A_i - A_j$ 中最大数和最小数的
差。

本题树节点结构是什么？

例题： POJ 3264 Balanced Lineup

给定 Q ($1 \leq Q \leq 200,000$)个数 $A_1, A_2 \dots A_Q$ ，
多次求任一区间 $A_i - A_j$ 中最大数和最小数的
差。

本题树节点结构：

```
struct CNode
```

```
{
```

```
    int L,R; //区间起点和终点
```

```
    int minV,maxV; //本区间里的最大最小值
```

```
    CNode * pLeft, * pRight;
```

```
};
```

也可以不要左右节点指针，用一个数组存放线段树。根节点下标为0。假设线段树上某节点下标为*i*,则:

左子节点下标为 $i * 2 + 1$,

右子节点下标为 $i * 2 + 2$

如果用一维数组存放线段树，且根节点区间[1,*n*]

- 使用左右节点指针，则数组需要有 $2n-1$ 个元素
- 不使用左右节点指针，则数组需要有:
 $2 * 2^{\lceil \log_2 n \rceil} - 1$ 个元素 ($\lceil \log_2 n \rceil$ 向上取整)

$2 * 2^{\lceil \log_2 n \rceil} - 1 \leq 4n - 1$, 实际运用时常可以更
小,可尝试 $3n$

Sample Input

6 3 //6个数, 3次个查询

1

7

3

4

2

5

1 5

4 6

2 2

Sample Output

6

3

0

```
#include <iostream>
using namespace std;
const int INF = 0xffffffff;
int minV = INF;
int maxV = -INF;
struct Node //不要左右子节点指针的做法
{
    int L, R;
    int minV, maxV;
    int Mid() {
        return (L+R)/2;
    }
};
Node tree[800010]; //4倍叶子节点的数量就够
```

```
void BuildTree(int root , int L, int R)
{
    tree[root].L = L;
    tree[root].R = R;
    tree[root].minV = INF;
    tree[root].maxV = - INF;
    if( L != R ) {
        BuildTree(2*root+1,L,(L+R)/2);
        BuildTree(2*root+2,(L+R)/2 + 1, R);
    }
}
```

```
void Insert(int root, int i,int v)
//将第i个数， 其值为v， 插入线段树
{
    if( tree[root].L == tree[root].R ) {
        //成立则亦有 tree[root].R == i
        tree[root].minV = tree[root].maxV = v;
        return;
    }
    tree[root].minV = min(tree[root].minV,v);
    tree[root].maxV = max(tree[root].maxV,v);
    if( i <= tree[root].Mid() )
        Insert(2*root+1,i,v);
    else
        Insert(2*root+2,i,v);
}
```

```
void Query(int root,int s,int e) {
```

```
//查询区间[s,e]中的最小值和最大值，如果更优就记在全局变量里
```

```
//minV和maxV里
```

```
    if( tree[root].minV >= minV && tree[root].maxV <= maxV )  
        return;
```

```
    if( tree[root].L == s && tree[root].R == e ) {  
        minV = min(minV,tree[root].minV);  
        maxV = max(maxV,tree[root].maxV);  
        return ;  
    }
```

```
    if( e <= tree[root].Mid())  
        Query(2*root+1,s,e);
```

```
    else if( s > tree[root].Mid() )  
        Query(2*root+2,s,e);
```

```
    else {  
        Query(2*root+1,s,tree[root].Mid());  
        Query(2*root+2,tree[root].Mid()+1,e);  
    }
```

```
}
```



```
int main()
{
    int n,q,h;
    int i,j,k;
    scanf("%d%d",&n,&q);
    BuildTree(0,1,n);
    for( i = 1;i <= n;i ++ ) {
        scanf("%d",&h);
        Insert(0,i,h);
    }
    for( i = 0;i < q;i ++ ) {
        int s,e;
        scanf("%d%d", &s,&e);
        minV = INF;
        maxV = -INF;
        Query(0,s,e);
        printf("%d\n",maxV - minV);
    }
    return 0;
}
```

POJ 3468 A Simple Problem with Integers

给定 Q ($1 \leq Q \leq 100,000$)个数 $A_1, A_2 \dots A_Q$,
以及可能多次进行的两个操作:

- 1) 对某个区间 $A_i \dots A_j$ 的每个数都加 n (n 可变)
- 2) 求某个区间 $A_i \dots A_j$ 的数的和

本题树节点要存哪些信息? 只存该区间的数的和, 行不行?

POJ 3468 A Simple Problem with Integers

只存和，会导致每次加数的时候都要更新到叶子节点，速度太慢($O(n\log n)$)，这是必须要避免的。

本题树节点结构：

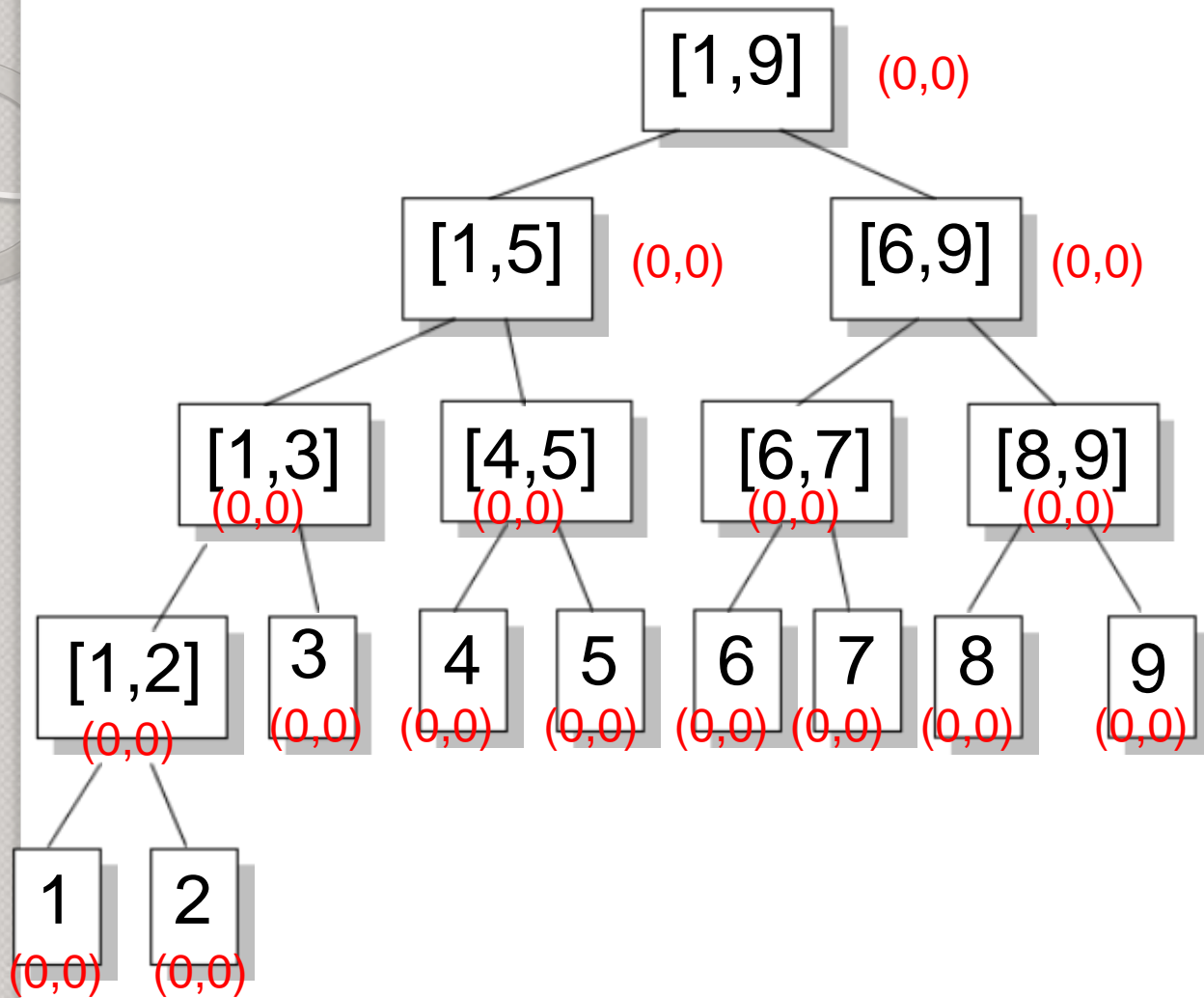
```
struct CNode
{
    int L ,R;
    CNode * pLeft, * pRight;
    long long nSum; //原来的和
    long long Inc; //增量c的累加
}; //本节点区间的和实际上是nSum+Inc*(R-L+1)
```

POJ 3468 A Simple Problem with Integers

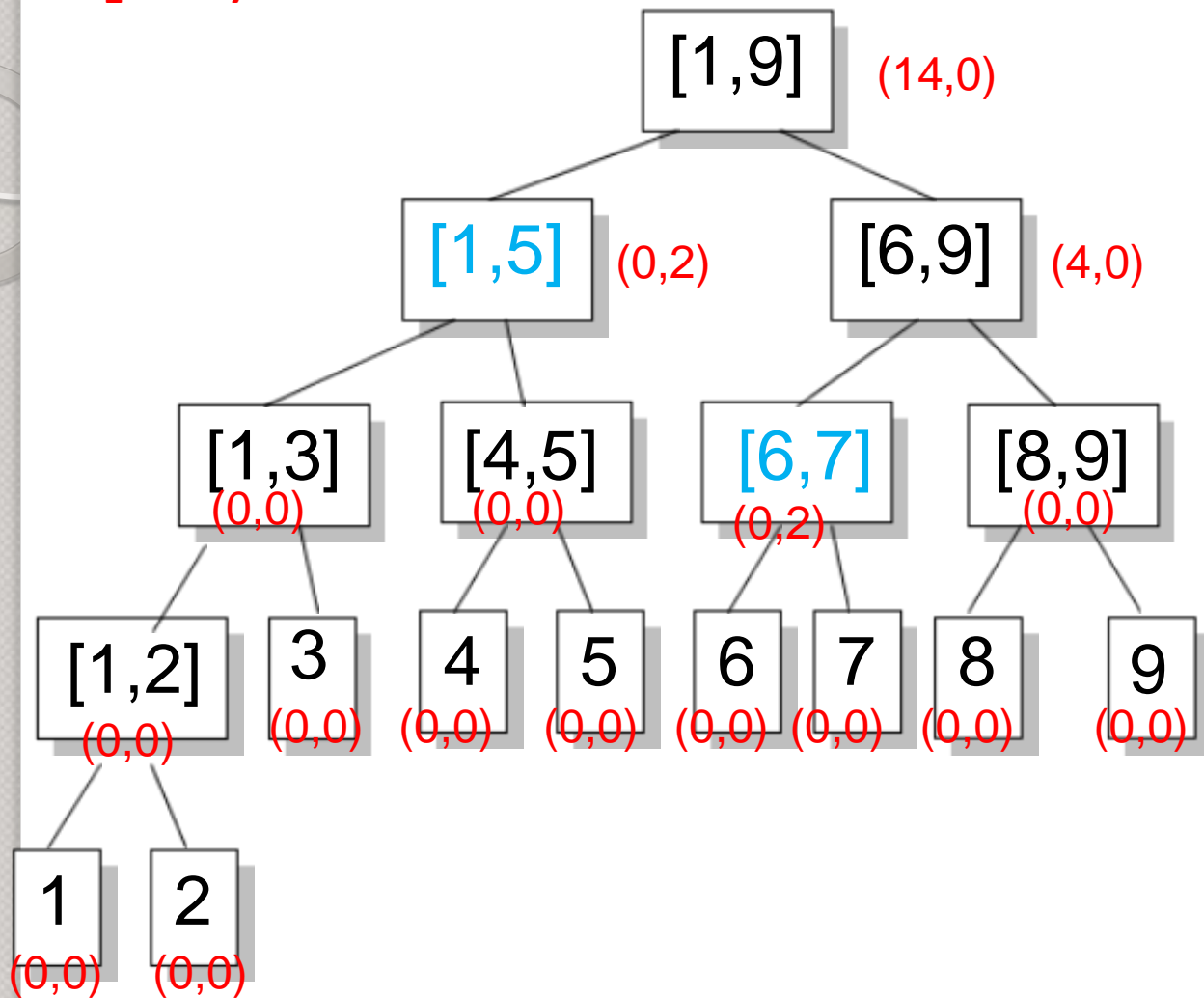
在增加时，如果要加的区间正好覆盖一个节点，则增加其节点的Inc值，不再往下走，否则要更新nSum(加上本次增量),再将增量往下传。这样更新的复杂度就是 $O(\log(n))$

在查询时，如果待查区间不是正好覆盖一个节点，就将节点的Inc往下带，然后将Inc代表的所有增量累加到nSum上后将Inc清0，接下来再往下查询。Inc往下带的过程也是区间分解的过程，复杂度也是 $O(\log(n))$

假设开始 $A_1 \dots A_9$ 都是0

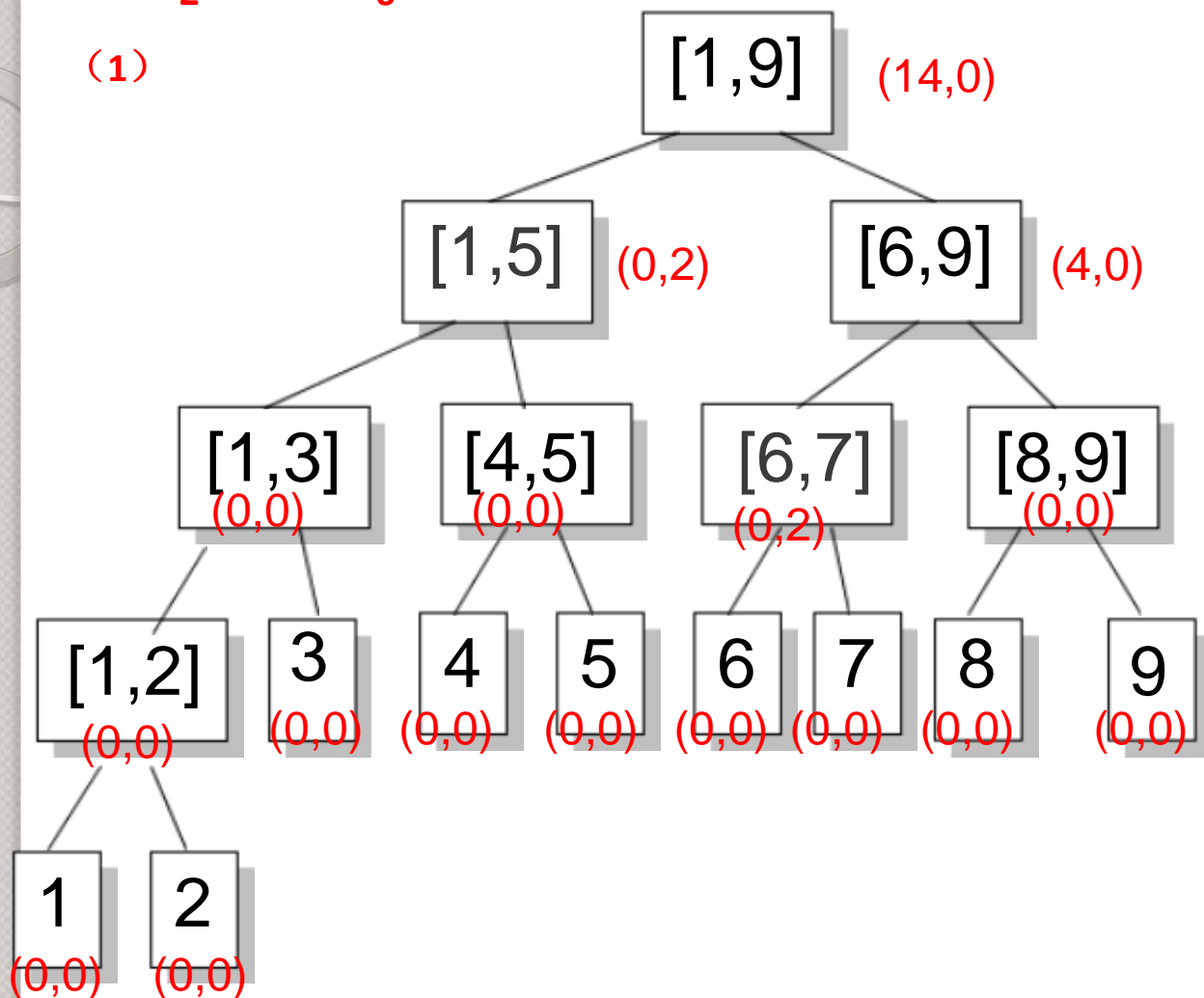


$A_1 \dots A_7 + 2$



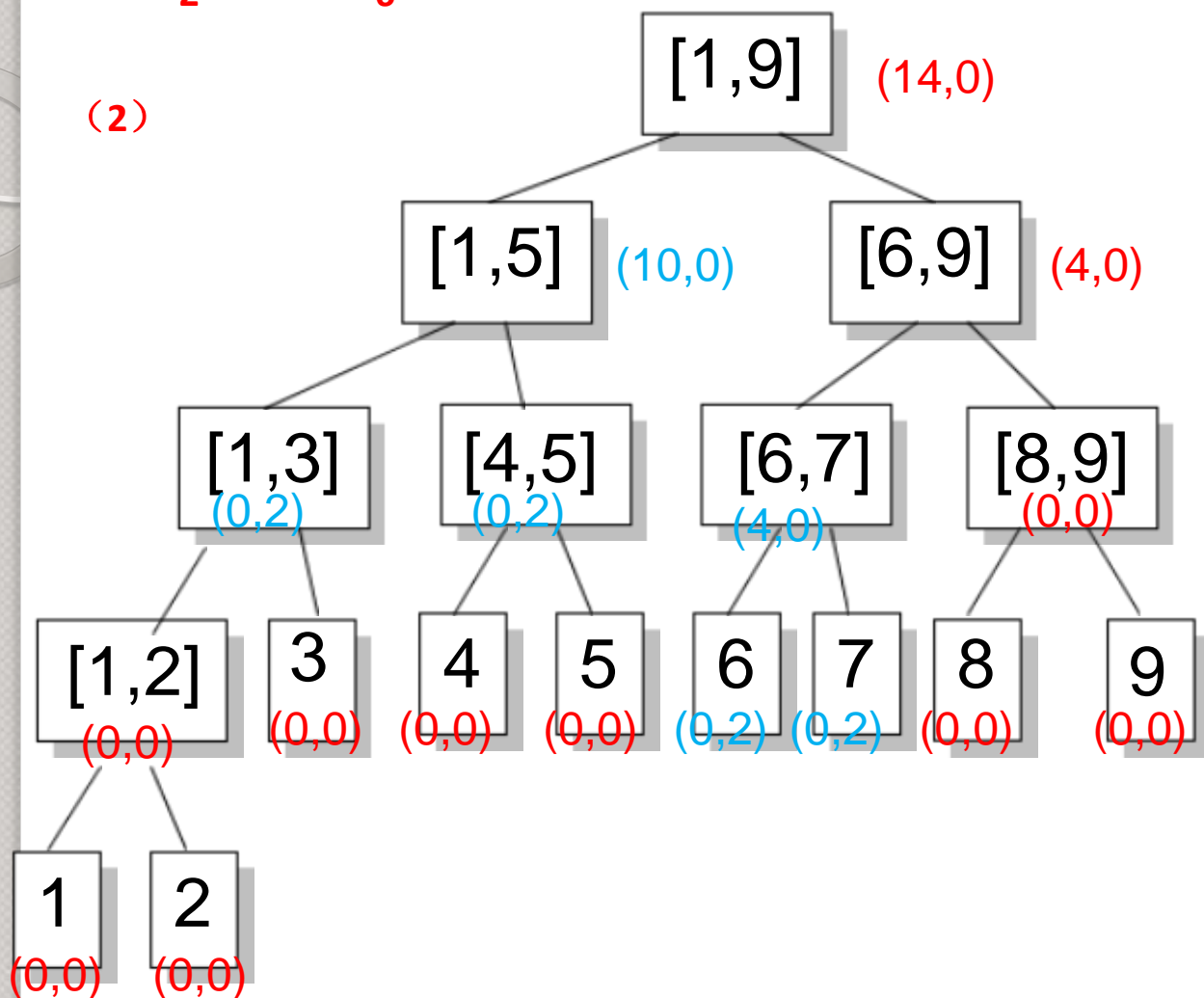
查 $A_2 + \dots + A_6$

(1)



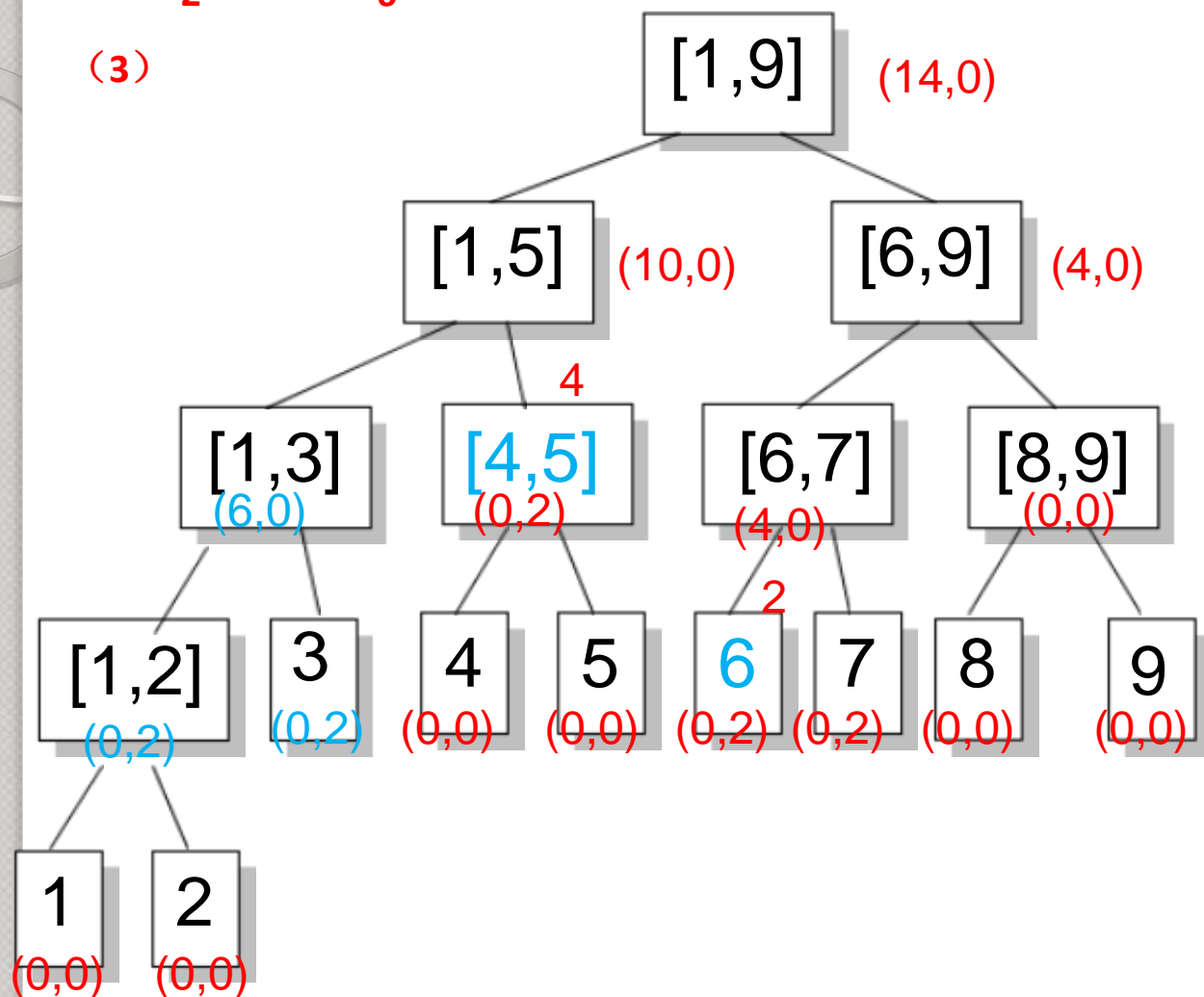
查 $A_2 + \dots + A_6$

(2)



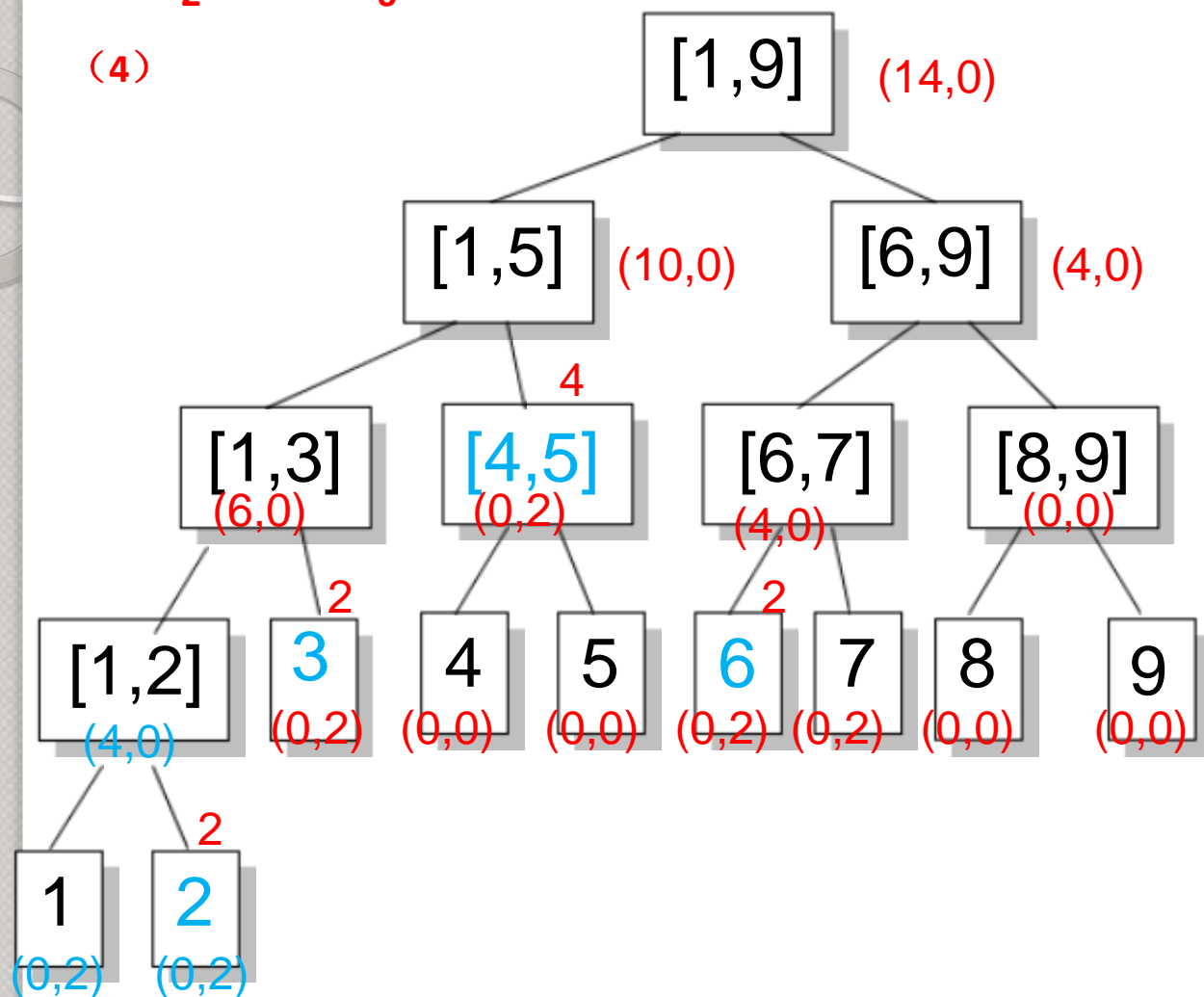
查 $A_2 + \dots + A_6$

(3)



查 $A_2 + \dots + A_6$

(4)



```
#include <iostream>
using namespace std;
struct CNode
{
    int L ,R;
    CNode * pLeft, * pRight;
    long long nSum; //原来的和
    long long Inc; //增量c的累加
};
```

```
CNode Tree[200010]; // 2倍叶子节点数目就够
int nCount = 0;
int Mid( CNode * pRoot)
{
    return (pRoot->L + pRoot->R)/2;
}
```

```
void BuildTree(CNode * pRoot,int L, int R)
{
    pRoot->L = L;
    pRoot->R = R;
    pRoot->nSum = 0;
    pRoot->Inc = 0;
    if( L == R)
        return;
    nCount ++;
    pRoot->pLeft = Tree + nCount;
    nCount ++;
    pRoot->pRight = Tree + nCount;
    BuildTree(pRoot->pLeft,L,(L+R)/2);
    BuildTree(pRoot->pRight,(L+R)/2+1,R);
}
```

```
void Insert( CNode * pRoot,int i, int v)
{
    if( pRoot->L == i && pRoot->R == i) {
        pRoot->nSum = v;
        return ;
    }
    pRoot->nSum += v;
    if( i <= Mid(pRoot))
        Insert(pRoot->pLeft,i,v);
    else
        Insert(pRoot->pRight,i,v);
}
```

```

void Add( CNode * pRoot, int a, int b, long long c)
{
    if( pRoot->L == a && pRoot->R == b) {
        pRoot->Inc += c;
        return ;
    }
    pRoot->nSum += c * ( b - a + 1) ;
    if( b <= (pRoot->L + pRoot->R)/2)
        Add(pRoot->pLeft,a,b,c);
    else if( a >= (pRoot->L + pRoot->R)/2 +1)
        Add(pRoot->pRight,a,b,c);
    else {

        Add(pRoot->pLeft,a,
            (pRoot->L + pRoot->R)/2 ,c);
        Add(pRoot->pRight,
            (pRoot->L + pRoot->R)/2 + 1,b,c);
    }
}

```

```
long long QuerynSum( CNode * pRoot, int a, int b)
{
    if( pRoot->L == a && pRoot->R == b)
        return pRoot->nSum +
            (pRoot->R - pRoot->L + 1) * pRoot->Inc ;

    pRoot->nSum += (pRoot->R - pRoot->L + 1) * pRoot->Inc ;
    Add( pRoot->pLeft,pRoot->L,Mid(pRoot),pRoot->Inc);
    Add( pRoot->pRight,Mid(pRoot) + 1,pRoot->R,pRoot->Inc);
    pRoot->Inc = 0;

    if( b <= Mid(pRoot))
        return QuerynSum(pRoot->pLeft,a,b);
    else if( a >= Mid(pRoot) + 1)
        return QuerynSum(pRoot->pRight,a,b);
    else {
        return QuerynSum(pRoot->pLeft,a,Mid(pRoot)) +
            QuerynSum(pRoot->pRight,Mid(pRoot) + 1,b);
    }
}
```

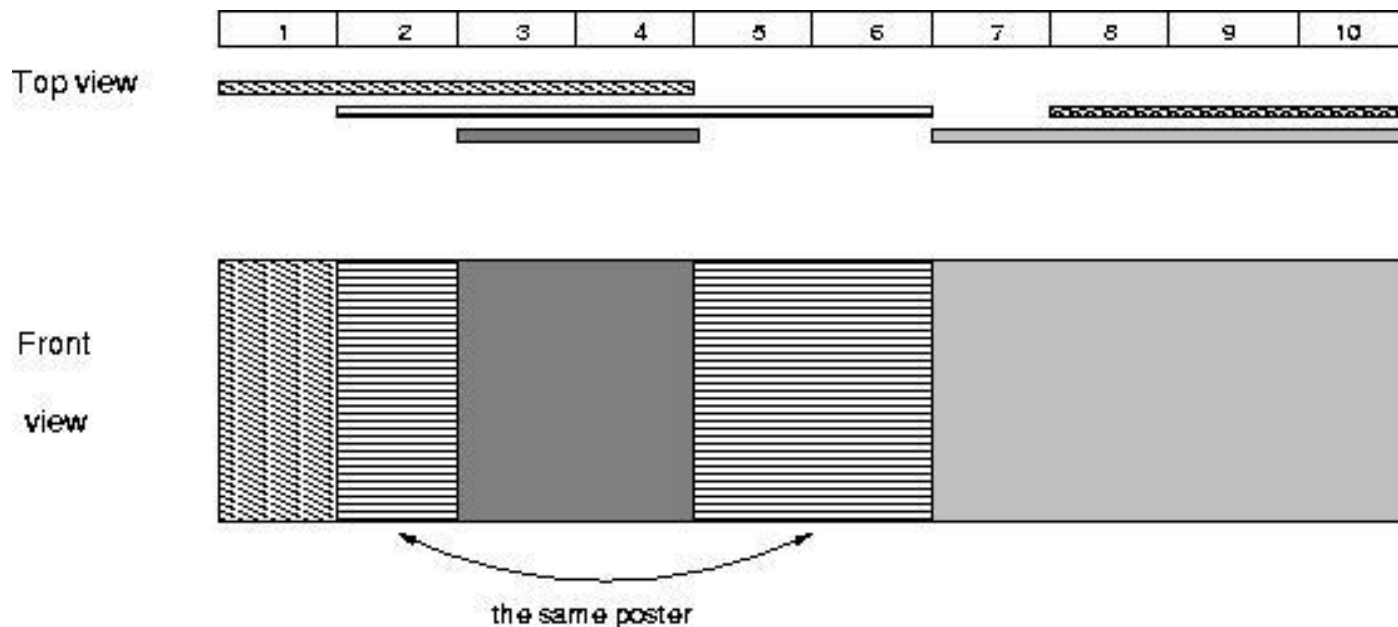
```
int main()
{
    int n,q,a,b,c;
    char cmd[10];
    scanf("%d%d",&n,&q);
    int i,j,k;
    nCount = 0;
    BuildTree(Tree,1,n);
    for( i = 1;i <= n;i ++ ) {
        scanf("%d",&a);
        Insert(Tree,i,a);
    }
    for( i = 0;i < q;i ++ ) {
        scanf("%s",cmd);
        if ( cmd[0] == 'C' ) {
            scanf("%d%d%d",&a,&b,&c);
            Add( Tree,a,b,c);
        }
        else {
            scanf("%d%d",&a,&b);
            printf("%l64d\n",QuerynSum(Tree,a,b));
        }
    }
    return 0;
}
```


离散化


有时，区间的端点不是整数，或者区间太大导致建树内存开销过大MLE，那么就需要进行“离散化”后再建树。

POJ 2528 Mayor's posters

给定一些海报，可能互相重叠，告诉你每个海报宽度（高度都一样）和先后叠放次序，问没有被完全盖住的海报有多少张。



海报最多10,000张，但是墙有10,000,000块瓷砖长。海报端点不会落在瓷砖中间。




思路：依次贴上一张张海报，每贴一张海报，就询问这张海报有没有被全部遮住
（假设贴海报的过程和现实可以不同，也可以先贴上面的，再贴下面的）

贴的顺序如何选择？

思路：依次贴上一张张海报，每贴一张海报，就询问这张海报有没有被全部遮住（假设贴海报的过程和现实可以不同，也可以先贴上面的，再贴下面的）

贴的顺序如何选择？

关键： 插入数据的顺序 ----- 从上往下依次插入每张海报，这样后插入的海报不可能覆盖先插入的海报，因此插入一张海报时，如果发现海报对应的瓷砖有一块露出来，就说明该海报部分可见。



给瓷砖编号，一个海报就相当于一个整数区间。贴海报就是区间操作，查询海报是否可见也是区间操作。

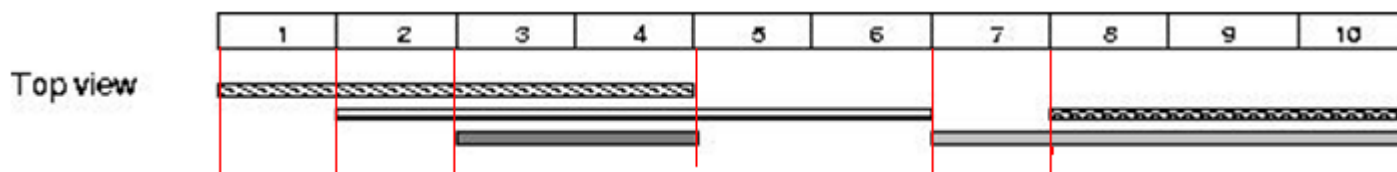
因此可以用线段树来解决

POJ 2528 Mayor's posters

如果每个叶子节点都代表一块瓷砖，那么线段树会导致MLE，即单位区间的数目太多。而且时间复杂度为 $n \log m$ (n 是海报数目 m 是瓷砖数目)

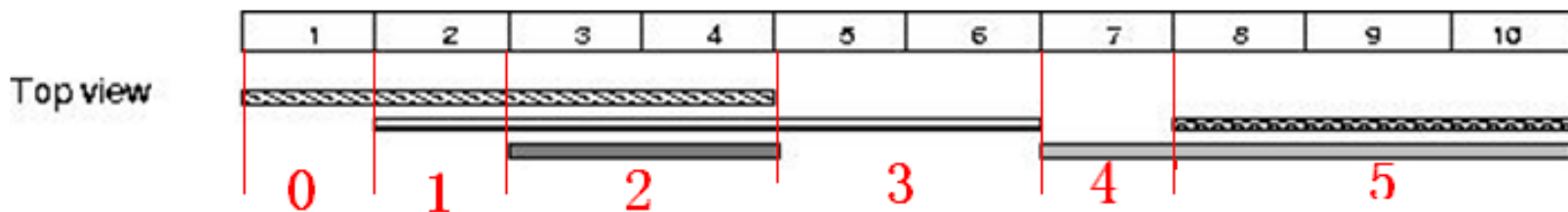
实际上，由于最多10,000个海报，共计20,000个端点，这些端点把墙最多分成19,999个单位区间（题意为整个墙都会被盖到）。每个单位区间的瓷砖数目可以不同。

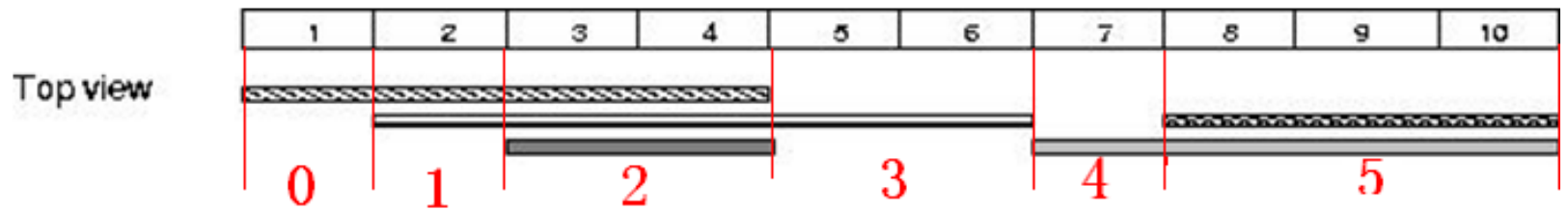
我们只要对这19,999个区间编号，然后建树即可。这就是离散化。



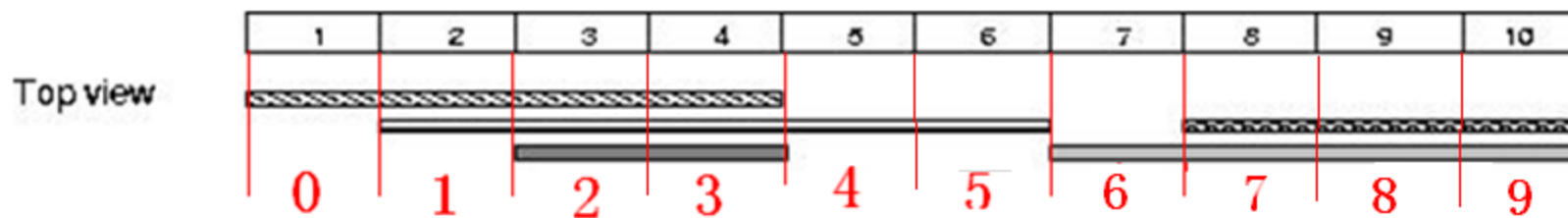
POJ 2528 Mayor's posters

- . 这些单位区间在线段树上是叶子节点
- . 每个单位区间要么全被覆盖，要么全部露出
- . 没有海报的端点会落在一个单位区间内部
- . 每张海报一定完整覆盖若干个连续的单位区间
- . 要算出一共有多少个单位区间，并且算出每张海报覆盖的单位区间 $[a,b]$ (海报覆盖了从 a 号单位区间到 b 号单位区间)





按上图的离散化方法，求每张海报覆盖了哪些单位区间，写起来稍麻烦



更好的离散化方法，是将所有海报的端点瓷砖排序，把每个海报的端点瓷砖都看做一个单位区间，两个相邻的端点瓷砖之间的部分是一个单位区间

这样最多会有 $20000 + 19999$ 个单位区间

时间复杂度变为 $n \log n$ n 是海报数目

POJ 2528 Mayor's posters

如果海报端点坐标是浮点数，其实也一样处理。

树节点要保存哪些信息，而且这些信息该如何动态更新呢？

POJ 2528 Mayor's posters

```
struct CNode
{
    int L,R;
    bool bCovered;
    CNode * pLeft, * pRight;
};
```

bCovered表示本区间是否已经完全被海报盖住

```
#include <iostream>
#include <algorithm>
#include <math.h>
using namespace std;
int n;
struct CPost
{
    int L,R;
};
CPost posters[10100];
int x[20200]; //存放所有海报的端点瓷砖编号
int hash[10000010]; //hash[i]表示瓷砖i所处的离散化后的区间编号
struct CNode
{
    int L,R;
    bool bCovered; //区间[L,R]是否已经被完全覆盖
    CNode * pLeft, * pRight;
};
CNode Tree[10000000];
int nNodeCount = 0;
```

```
int Mid( CNode * pRoot)
{
    return (pRoot->L + pRoot->R)/2;
}
void BuildTree( CNode * pRoot, int L, int R)
{
    pRoot->L = L;
    pRoot->R = R;
    pRoot->bCovered = false;
    if( L == R )
        return;
    nNodeCount ++;
    pRoot->pLeft = Tree + nNodeCount;
    nNodeCount ++;
    pRoot->pRight = Tree + nNodeCount;
    BuildTree( pRoot->pLeft,L,(L+R)/2);
    BuildTree( pRoot->pRight,(L+R)/2 + 1,R);
}
```

```
bool Post( CNode *pRoot, int L, int R)
```

```
{ //插入一张正好覆盖区间[L,R]的海报,返回true则说明区间[L,R]是部分或全部可见的
```

```
    if( pRoot->bCovered )    return false;
```

```
    if( pRoot->L == L && pRoot->R == R) {
```

```
        pRoot->bCovered = true;
```

```
        return true;
```

```
    }
```

```
    bool bResult ;
```

```
    if( R <= Mid(pRoot) )
```

```
        bResult = Post( pRoot->pLeft,L,R);
```

```
    else if( L >= Mid(pRoot) + 1)
```

```
        bResult = Post( pRoot->pRight,L,R);
```

```
    else {
```

```
        bool b1 = Post(pRoot->pLeft ,L,Mid(pRoot));
```

```
        bool b2 = Post( pRoot->pRight,Mid(pRoot) + 1,R);
```

```
        bResult = b1 || b2;
```

```
    }
```

```
    //要更新根节点的覆盖情况
```

```
    if( pRoot->pLeft->bCovered && pRoot->pRight->bCovered )
```

```
        pRoot->bCovered = true;
```

```
    return bResult;
```

```
}
```

```
int main()
{
    int t;
    int i,j,k;
    scanf("%d",&t);
    int nCaseNo = 0;
    while(t--) {
        nCaseNo ++;
        scanf("%d",&n);
        int nCount = 0;
        for( i = 0;i < n;i ++ ) {
            scanf("%d%d", & posters[i].L,
                & posters[i].R );

            x[nCount++] = posters[i].L;
            x[nCount++] = posters[i].R;
        }
        sort(x,x+nCount);
        nCount = unique(x,x+nCount) - x; //去掉重复元素
    }
}
```

//下面离散化

```
int nIntervalNo = 0;
for( i = 0; i < nCount; i ++ ) {
    hash[x[i]] = nIntervalNo;
    if( i < nCount - 1 ) {
        if( x[i+1] - x[i] == 1 )
            nIntervalNo ++;
        else
            nIntervalNo += 2;
    }
}
```

```
BuildTree( Tree, 0, nIntervalNo );
int nSum = 0;
for( i = n - 1; i >= 0; i -- ) { // 从后往前看每个海报是否可
    if( Post(Tree, hash[posters[i].L],
                hash[posters[i].R]))
        nSum ++;
}
printf("%d\n", nSum);
}
```

```
return 0;
```

```
}
```

见

POJ 1151 Atlantis

给定 n 个矩形 ($n \leq 100$), 其顶点坐标是浮点数, 可能互相重叠, 问这些矩形覆盖到的面积是多大。

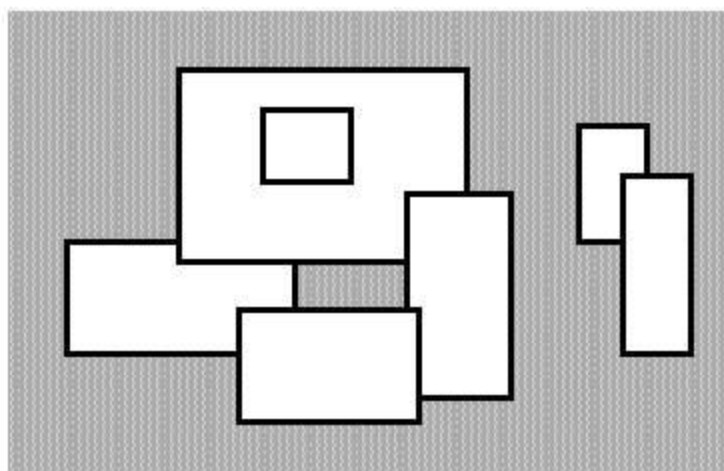
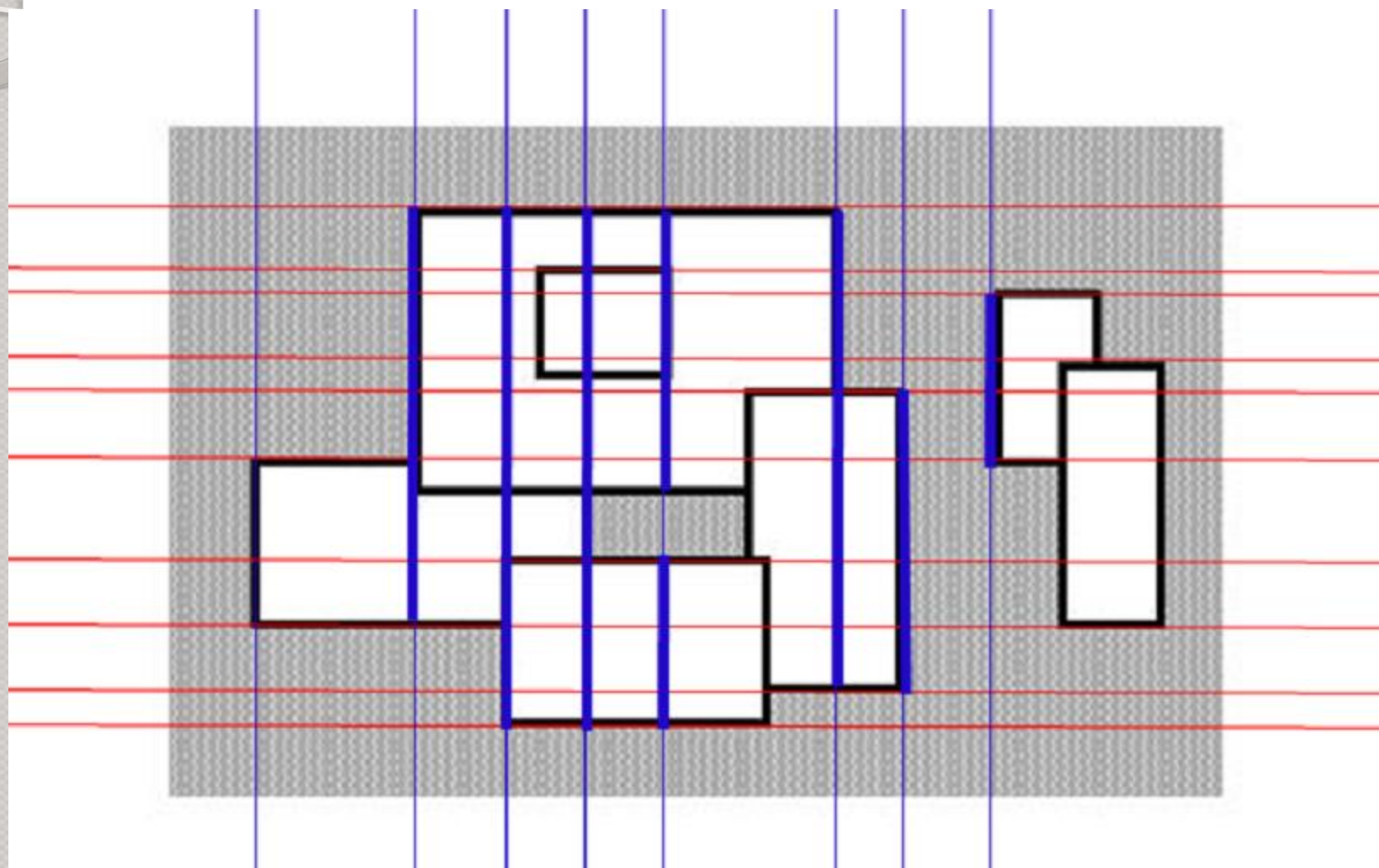
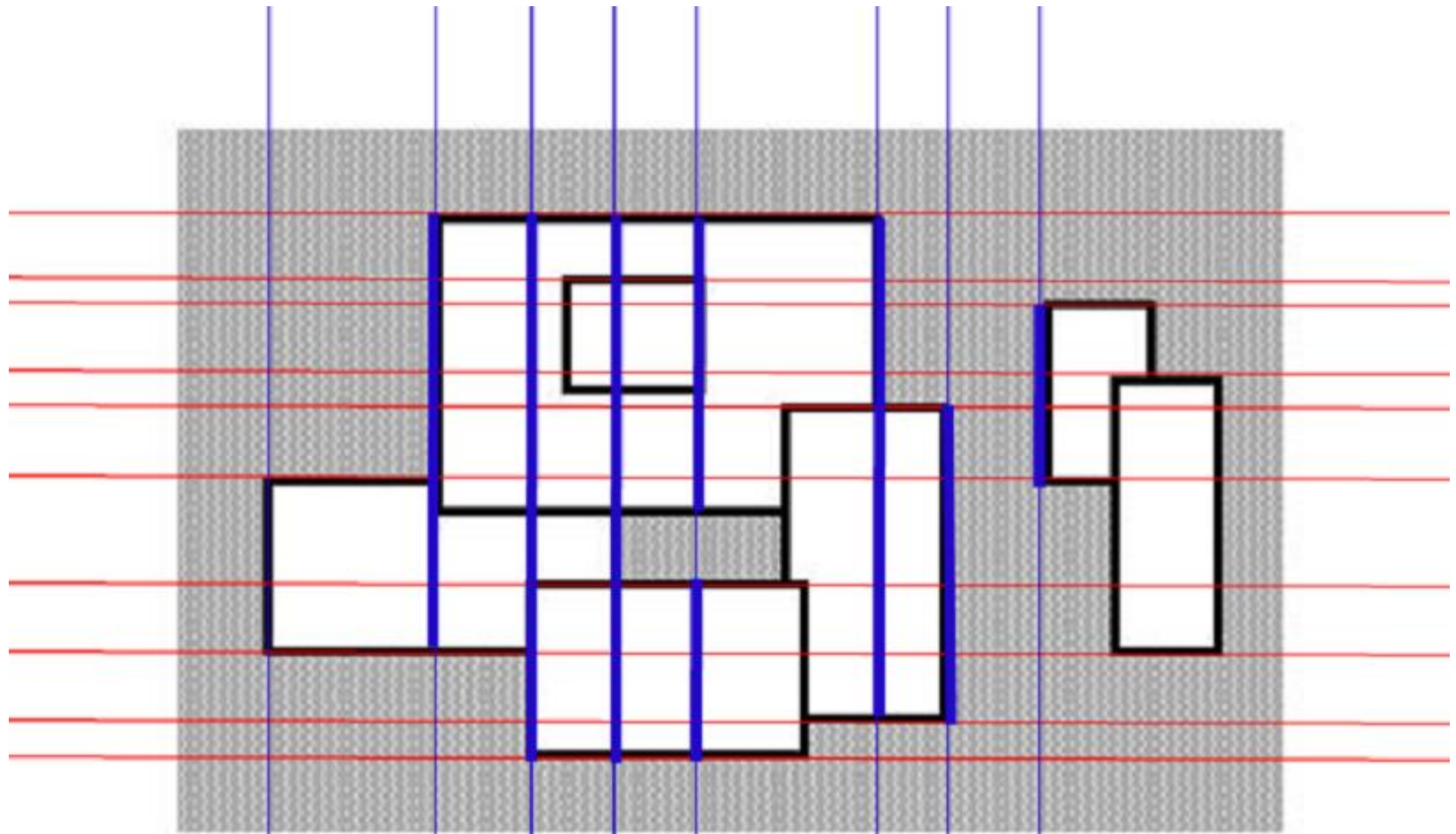


Figure 1. A set of 7 rectangles

用一条直线从左到右扫描，碰到一条矩形竖边的时候，就计算该直线有多长被矩形覆盖，以及被覆盖部分是覆盖了几重。碰到矩形左边，要增加被覆盖的长度，碰到右边，要减少被覆盖的长度



随着扫描线的右移动，覆盖面积不断增加。



每碰到一条矩形的纵边，覆盖面积就增加 $\text{Len} * \text{该纵边到下一条纵边的距离}$ 。 Len 是此时扫描线被矩形覆盖的长度

POJ 1151 Atlantis

在Y轴进行离散化。 n 个矩形的 $2n$ 个横边纵坐标共构成最多 $2n-1$ 个区间的边界，对这些区间编号，建立起线段树。

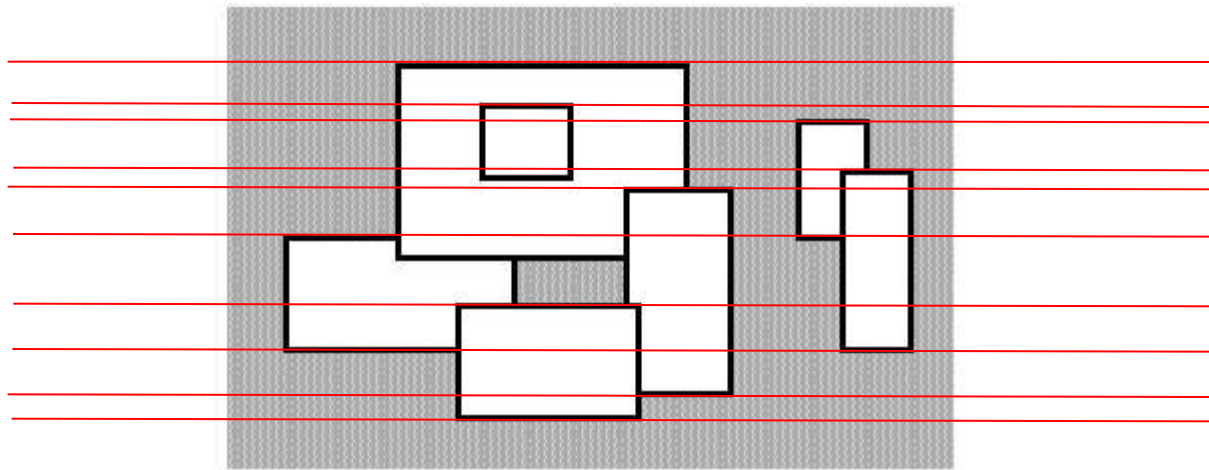


Figure 1. A set of 7 rectangles

POJ 1151 Atlantis

线段树的节点要保存哪些信息？如何将一个个矩形插入线段树？插入过程中这些信息如何更新？怎样查询？

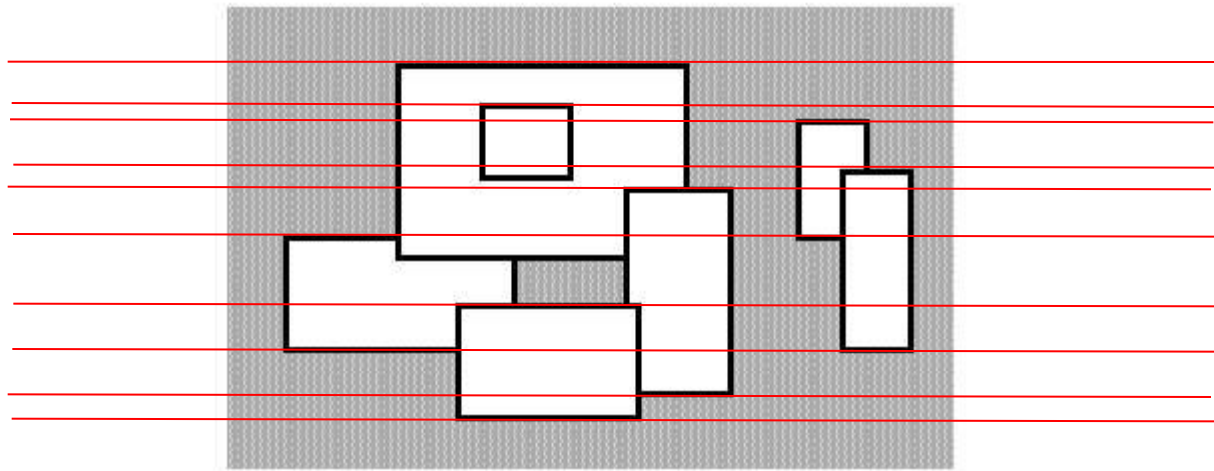


Figure 1. A set of 7 rectangles

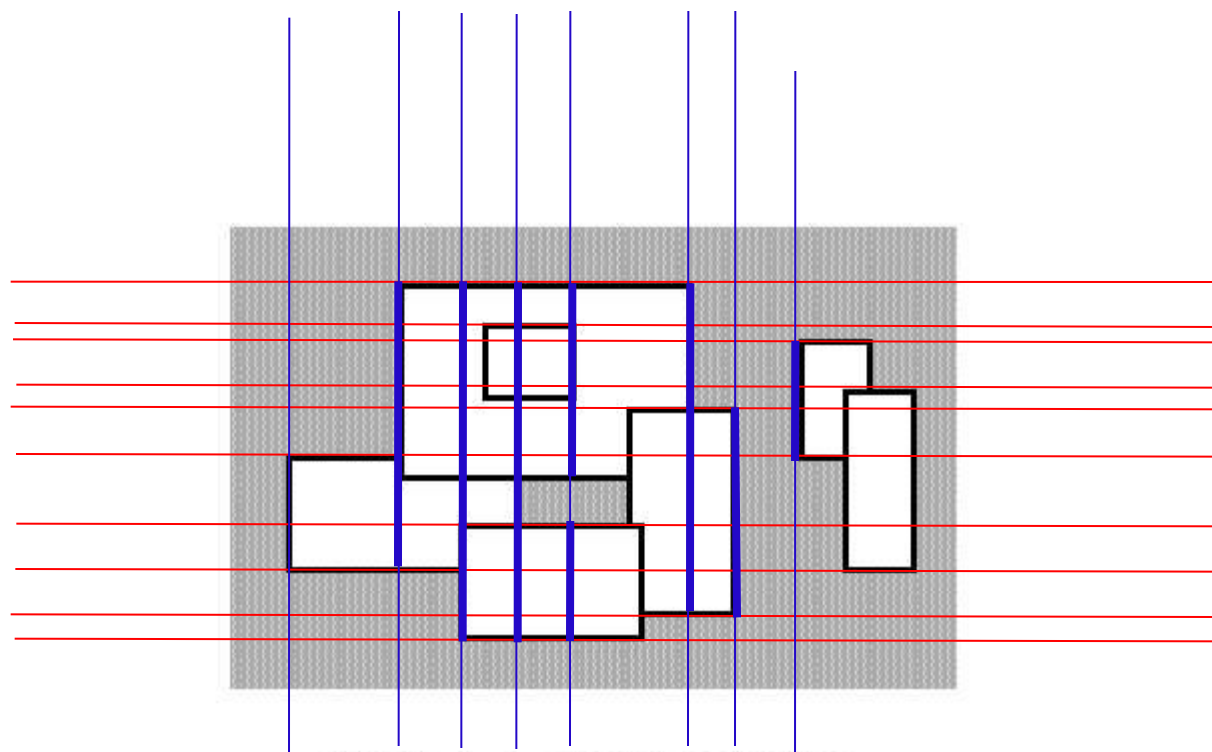


Figure 1. A set of 7 rectangles

POJ 1151 Atlantis

```
struct CNode
```

```
{
```

```
    int L,R;
```

```
    CNode * pLeft, * pRight;
```

```
    double Len; //当前,本区间上有多长的  
                部分是落在那些矩形中的
```

```
    int Covers; //本区间当前被多少个矩形  
                完全包含
```

```
};
```

一开始, 所有区间 $Len = 0$ $Covers = 0$

POJ 1151 Atlantis

插入数据的顺序：

将矩形的纵边从左到右排序，然后依次将这些纵边插入线段树。要记住哪些纵边是一个矩形的左边(开始边)，哪些纵边是一个矩形的右边（结束边），以便插入时，对**Len**和**Covers**做不同的修改。

插入一条边后,就根据**根节点**的**Len** 值增加总覆盖面积的值。

增量是**Len** * 本边到下一条边的距离


```
#include <iostream>
#include <algorithm>
#include <math.h>
#include <set>
using namespace std;
double y[210];
struct CNode
{
    int L,R;
    CNode * pLeft, * pRight;
    double Len; //当前,本区间上有多长的部分是落在那些
                矩形中的
    int Covers;//本区间当前被多少个矩形完全包含
};
CNode Tree[1000];
```

```
struct CLine
{
    double x,y1,y2;
    bool bLeft; //是否是矩形的左边
} lines[210];
```

```
int nNodeCount = 0;
```

```
bool operator< ( const CLine & l1,const CLine & l2)
{
    return l1.x < l2.x;
}
```

```

template <class F,class T>
F bin_search(F s, F e, T val)
{ //在区间[s,e)中查找 val,找不到就返回 e
    F L = s;
    F R = e-1;
    while(L <= R ) {
        F mid = L + (R-L)/2;
        if( !( * mid < val || val < * mid ))
            return mid;
        else if(val < * mid)
            R = mid - 1;
        else
            L = mid + 1;
    }
    return e;
}

int Mid(CNode * pRoot)
{
    return (pRoot->L + pRoot->R ) >>1;
}

```

```
void Insert(CNode * pRoot,int L, int R)
//在区间pRoot 插入矩形左边的一部分或全部，该左边的一部分或全部覆盖了区间[L,R]
{
    if( pRoot->L == L && pRoot->R == R){
        pRoot->Len = y[R+1] - y[L];
        pRoot->Covers ++;
        return;
    }
    if( R <= Mid(pRoot))
        Insert(pRoot->pLeft,L,R);
    else if( L >= Mid(pRoot)+1)
        Insert(pRoot->pRight,L,R);
    else {
        Insert(pRoot->pLeft,L,Mid(pRoot));
        Insert(pRoot->pRight,Mid(pRoot)+1,R);
    }
    if( pRoot->Covers == 0) //如果不为0，则说明本区间当前仍然被某个矩形完全包含，则不能更新 Len
        pRoot->Len = pRoot->pLeft ->Len +
                    pRoot->pRight ->Len;
}
```

```
void Delete(CNode * pRoot,int L, int R) {
```

/在区间pRoot 删除矩形右边的一部分或全部，该矩形右边的一部分或全部覆盖了区间[L,R]

```
    if( pRoot->L == L && pRoot->R == R) {
```

```
        pRoot->Covers --;
```

```
        if( pRoot->Covers == 0 )
```

```
            if( pRoot->L == pRoot->R )
```

```
                pRoot->Len = 0;
```

```
            else
```

```
                pRoot->Len = pRoot->pLeft ->Len +  
                    pRoot->pRight ->Len;
```

```
        return ;
```

```
    }
```

```
    if( R <= Mid(pRoot))
```

```
        Delete(pRoot->pLeft,L,R);
```

```
    else if( L >= Mid(pRoot)+1)
```

```
        Delete(pRoot->pRight,L,R);
```

```
    else {
```

```
        Delete(pRoot->pLeft,L,Mid(pRoot));
```

```
        Delete(pRoot->pRight,Mid(pRoot)+1,R);
```

```
    }
```

if(pRoot->Covers == 0) //如果不为0，则说明本区间当前仍然被某个矩形完全包含，则不能更新 Len

```
    pRoot->Len = pRoot->pLeft ->Len + pRoot->pRight ->Len;
```

```
}
```

```
void BuildTree( CNode * pRoot, int L,int R)
{
    pRoot->L = L;
    pRoot->R = R;
    pRoot->Covers = 0;
    pRoot->Len = 0;
    if( L == R)
        return;
    nNodeCount ++;
    pRoot->pLeft = Tree + nNodeCount;
    nNodeCount ++;
    pRoot->pRight = Tree + nNodeCount;
    BuildTree( pRoot->pLeft,L,(L+R)/2);
    BuildTree( pRoot->pRight,(L+R)/2+1,R);
}
```

```
int main()
```

```
{
```

```
    int n;    int i,j,k;  double x1,y1,x2,y2; int yc,lc;
```

```
    int nCount = 0;
```

```
    int t = 0;
```

```
    while(true) {
```

```
        scanf("%d",&n);
```

```
        if( n == 0) break;
```

```
        t ++;    yc = lc = 0;
```

```
        for( i = 0;i < n;i ++ ) {
```

```
            scanf("%lf%lf%lf%lf", &x1, &y1,&x2,&y2);
```

```
            y[yc++] = y1;    y[yc++] = y2;
```

```
            lines[lc].x = x1; lines[lc].y1 = y1;
```

```
            lines[lc].y2 = y2;
```

```
            lines[lc].bLeft = true;
```

```
            lc ++;
```

```
            lines[lc].x = x2; lines[lc].y1 = y1;
```

```
            lines[lc].y2 = y2;
```

```
            lines[lc].bLeft = false;
```

```
            lc ++;
```

```
        }
```

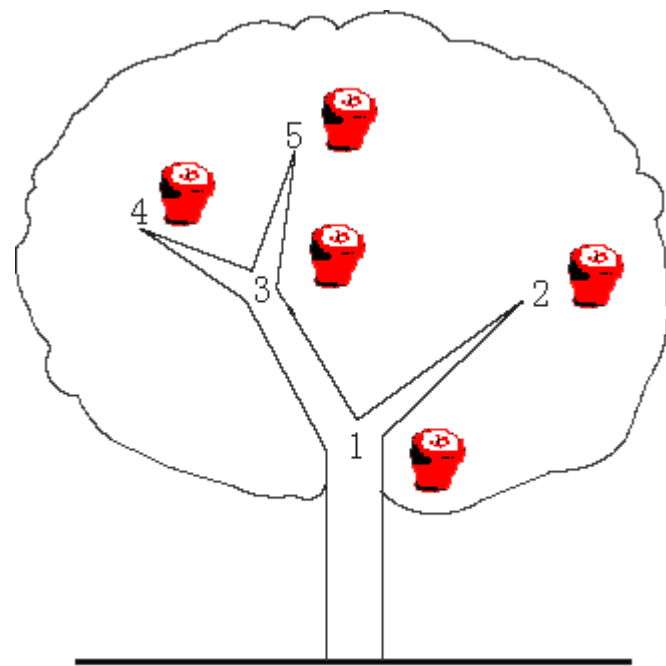
```

sort(y,y + yc);
yc = unique(y,y+yc) - y;
nNodeCount = 0;
//yc 是横线的条数, yc- 1是纵向区间的个数, 这些区间从0
//开始编号, 那么最后一个区间
//编号就是yc - 1 -1
BuildTree(Tree, 0, yc - 1 - 1);
sort(lines,lines + lc);
double Area = 0;
for( i = 0;i < lc - 1 ; i ++ ) {
    int L = bin_search( y,y+yc,lines[i].y1) - y;
    int R = bin_search( y,y+yc,lines[i].y2) - y;
    if( lines[i].bLeft )
        Insert(Tree,L,R-1);
    else
        Delete(Tree,L,R-1);
    Area += Tree[0].Len * (lines[i+1].x - lines[i].x);
}
printf("Test case #%d\n",t);
printf("Total explored area: %.2lf\n",Area);
printf("\n",Area);
}
return 0;
}

```


有时，不一定能够一眼看出什么是“区间”，这就要靠仔细观察，造出“区间”来。例如：

POJ 3321 Apple Tree



每个分叉点及末梢可能有苹果（最多1个），每次可以摘掉一个苹果，或有一个苹果新长出来，随时查询某个分叉点往上的子树里，一共有多少个苹果(分叉点数量： 100,000)。

POJ 3321 Apple Tree

深度优先遍历整个苹果树，为每个节点标记一个开始时间和结束时间（所有时间都不相同），显然子树里面所有节点的开始和结束时间，都位于子树树根的开始和结束时间之间。

问题变成：

有 n 个节点，就有 $2n$ 个开始结束时间，它们构成序列

$$A_1 A_2 \dots A_{2n}$$

序列里每个数是0或者1，可变化，随时查询某个区间里数的和。当然由于苹果树上每个放苹果的位置对应于数列里的两个数，所以结果要除以2

树状数组

- 对于序列a，我们设一个数组C
 - $C[i] = a[i - 2^k + 1] + \dots + a[i]$
 - k为i在二进制下末尾0的个数
 - 2^k 就是i 保留最右边的1，其余位全变0
 - i从1开始算！
- C即为a的树状数组
- 对于i,如何求 2^k ？

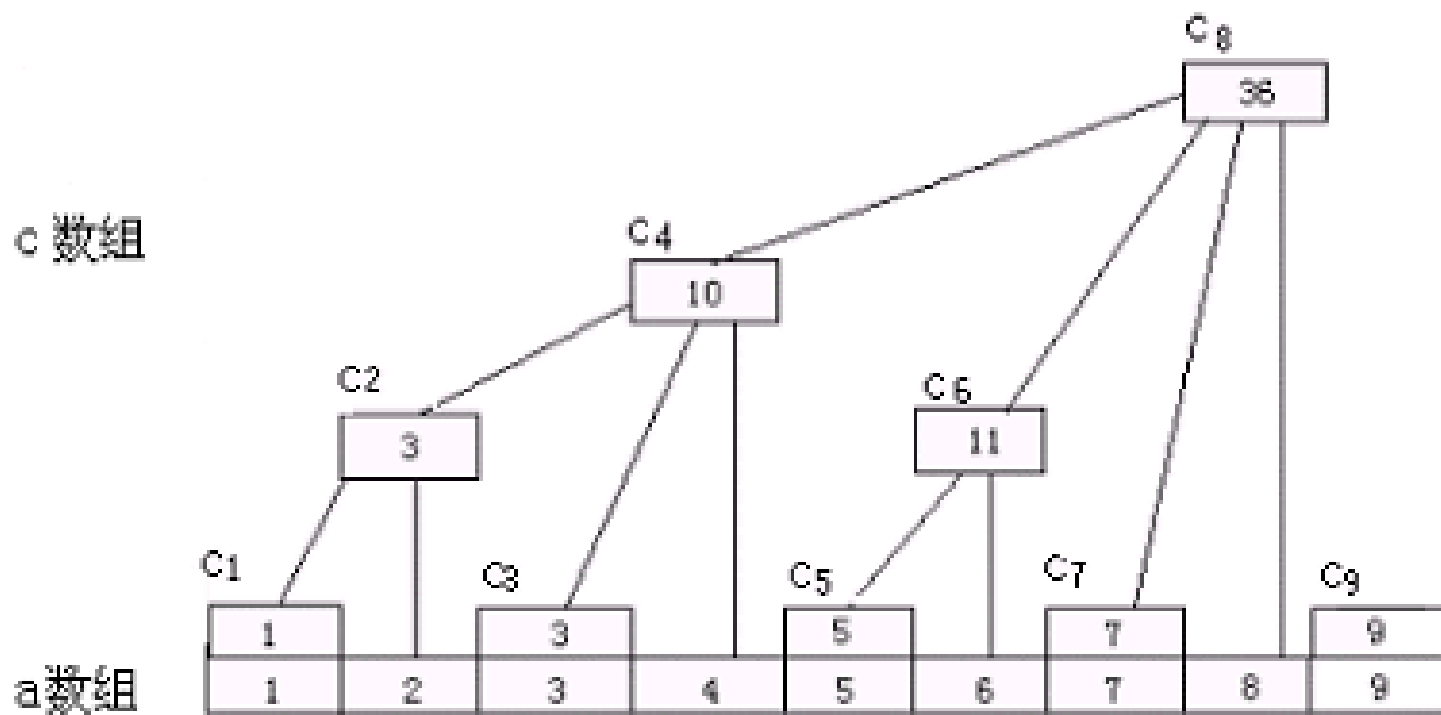
- $2^k = i \& (i \wedge (i - 1))$ 也就是 $i \& (-i)$
- 以6为例
- $(6)_{10} = (0110)_2$
- xor $6 - 1 = (5)_{10} = (0101)_2$
- $(0011)_2$
- and $(6)_{10} = (0110)_2$
- $(0010)_2 = (4)_{10}$
- 通常我们用 $\text{lowbit}(x)$ 表示 x 对应的 2^k ,
- $\text{lowbit}(x) = x \& (-x)$
- $\text{lowbit}(x)$ 实际上就是 x 的二进制表示形式留下最右边的1, 其他位都变成0

$$C[i] = a[i-\text{lowbit}(i)+1] + \dots + a[i]$$

C包含哪些项看上去没有规律

- $C_1 = A_1$
- $C_2 = A_1 + A_2$
- $C_3 = A_3$
- $C_4 = A_1 + A_2 + A_3 + A_4$
- $C_5 = A_5$
- $C_6 = A_5 + A_6$
- $C_7 = A_7$
- $C_8 = A_1 + A_2 + A_3 + A_4 + A_5 + A_6 + A_7 + A_8$
-
- $C_{16} = A_1 + A_2 + A_3 + A_4 + A_5 + A_6 + A_7 + A_8 + A_9 + A_{10} + A_{11} + A_{12} + A_{13} + A_{14} + A_{15} + A_{16}$

树状数组图示



树状数组的好处在于能快速求任意区间的和
 $a[i] + a[i+1] + \dots + a[j]$

设 $\text{sum}(k) = a[1] + a[2] + \dots + a[k]$

则 $a[i] + a[i+1] + \dots + a[j] = \text{sum}(j) - \text{sum}(i-1)$

有了树状数组， $\text{sum}(k)$ 就能在 $O(\log N)$ 时间内求出， N 是 a 数组元素个数。而且更新一个 a 的元素所花的时间也是 $O(\log N)$ 的(因为 a 更新了 C 也得更新)。

为什么呢？

根据C的构成规律，可以发现sum(k)可以表示为：

$$\text{sum}(k) = C[n_1] + C[n_2] + \dots + C[n_m]$$

其中 $n_m = k$

$n_{i-1} = n_i - \text{lowbit}(n_i)$ 而且 $n_1 - \text{lowbit}(n_1)$ 必须小于或等于0(其实只能等于0)， n_1 大于0

如： $\text{sum}(6) = C[4] + C[6]$

$\text{lowbit}(x)$ 实际上就是x的二进制表示形式留下最右边的1，其他位都变成0

那么， $\text{sum}(k)$ 最多有几项呢？这个决定了求区间和的时间复杂度

那么， $\text{sum}(k)$ 最多有几项呢？

$$\text{sum}(k) = C[n_1] + C[n_2] + \dots + C[n_m]$$

其中 $n_m = k$

$$n_{i-1} = n_i - \text{lowbit}(n_i)$$

$\text{lowbit}(x)$ 实际上就是 x 的二进制表示形式留下最右边的1，其他位都变成0

$n_i - \text{lowbit}(n_i)$ 就是 n_i 的二进制去掉最右边的1

k 的二进制里最多有 $\log_2 k$ （向上取整）个1

$\text{sum}(k)$ 最多 $\log_2 k$ （向上取整）项，所以本次求和的复杂度就是 $\log_2 k$

那么，为什么

$$\text{sum}(k) = C[n_1] + C[n_2] + \dots + C[n_m]$$

其中 $n_m = k$

$$n_{i-1} = n_i - \text{lowbit}(n_i)$$

证：

$$C[i] = a[i - \text{lowbit}(i) + 1] + \dots + a[i]$$

$i - \text{lowbit}(i) + 1$ 是什么？就是 i 把最右边的 1 去掉，然后再加 1

那么，为什么 $\text{sum}(k) = a[1] + a[2] + \dots + a[k]$
 $= C[n_1] + C[n_2] + \dots + C[n_m]$ (其中 $n_m = k$)

证明：

$$C[n_m] = a[n_m - \text{lowbit}(n_m) + 1] + \dots + a[n_m]$$

$$\begin{aligned} C[n_{m-1}] &= a[n_{m-1} - \text{lowbit}(n_{m-1}) + 1] + \dots + a[n_{m-1}] \\ &= a[n_{m-1} - \text{lowbit}(n_{m-1}) + 1] + \dots + a[n_m - \text{lowbit}(n_m)] \end{aligned}$$

$$\begin{aligned} C[n_{m-2}] &= a[n_{m-2} - \text{lowbit}(n_{m-2}) + 1] + \dots + a[n_{m-2}] \\ &= a[n_{m-1} - \text{lowbit}(n_{m-1}) + 1] + \dots + a[n_{m-1} - \text{lowbit}(n_{m-1})] \end{aligned}$$

.....

$$\begin{aligned} C[n_1] &= a[n_1 - \text{lowbit}(n_1) + 1] + \dots + a[n_1] \\ &= a[1] + \dots + a[n_1] \end{aligned}$$

(因 $n_1 - \text{lowbit}(n_1)$ 必须等于0，否则就还需要 $C[n_1 - \text{lowbit}(n_1)]$ 了)

更新一个a元素，C也要跟着更新，复杂度是多少呢？
即C里有几项要更新呢？

- $C1=A1$
- $C2=A1+A2$
- $C3=A3$
- $C4=A1+A2+A3+A4$
- $C5=A5$
- $C6=A5+A6$
- $C7=A7$
- $C8=A1+A2+A3+A4+A5+A6+A7+A8$
-
- $C16=A1+A2+A3+A4+A5+A6+A7+A8+A9+A10+A11+A12+A13+A14+A15+A16$

更新一个a元素，C也要跟着更新，复杂度是多少呢？
即C里有几项要更新呢？

如果a[i]更新了，那么以下的几项都需要更新：

$$C[n_1], C[n_2], \dots C[n_m]$$

其中， $n_1 = i$ ， $n_{i+1} = n_i + \text{lowbit}(n_i)$

$n_m + \text{lowbit}(n_m)$ 必须大于 a 的元素个数 N, n_m 小于等于N

同理，总的来说更新一个元素的时间，也是 $\log N$ 的

为什么如果 $a[i]$ 更新了，那么有且仅有以下的几项需要更新：

$$C[n_1], C[n_2], \dots, C[n_m]$$

其中， $n_1 = i$ ， $n_{i+1} = n_i + \text{lowbit}(n_i)$

$a[i]$ 更新 $\rightarrow C[i]$ 必须更新，因为 $C[i] = a[x] + \dots + a[i]$

而且， $C[k + \text{lowbit}(k)]$ 的起始项早于 $C[k]$ 的起始项
所以，若 $C[k]$ 包含 $a[i]$ ，则 $C[k + \text{lowbit}(k)]$ 也包含 $a[i]$ ，即

$C[k]$ 需要更新 $\rightarrow C[k + \text{lowbit}(k)]$ 也需要更新

下面证明：

$C[k+\text{lowbit}(k)]$ 的起始项早于 $C[k]$ 的起始项

$$C[k] = a[k-\text{lowbit}(k)+1] + \dots$$

$$C[k+\text{lowbit}(k)] = a[k+\text{lowbit}(k) - \text{lowbit}(k+\text{lowbit}(k))+1] + \dots$$

$$k - \text{lowbit}(k) \geq k+\text{lowbit}(k) - \text{lowbit}(k+\text{lowbit}(k))$$

易证, 因为 $\text{lowbit}(k+\text{lowbit}(k)) \geq 2 * \text{lowbit}(k)$

因此

$C[k+\text{lowbit}(k)]$ 的起始项早于 $C[k]$ 的起始项

下面要证明，若 $a[i]$ 更新,则对任何 k , ($i < k < i + \text{lowbit}(i)$),
 $C[k]$ 都不需要更新 (即 $C[k]$ 不包含 $a[i]$)

$$C[k] = a[k - \text{lowbit}(k) + 1] + \dots + a[k]$$

只要证明 $k - \text{lowbit}(k) + 1$ 比 i 大即可

因 $i < k < i + \text{lowbit}(i)$, 假设 i 的最右边的1是从右到左从0开始数的第 n 位, 那么 $i + \text{lowbit}(i)$ 就是将 i 的低 n 位全变成1后, 再加1。那么 k 一定是从第 n 位到最高位都和 i 相同, 但是低 n 位比 i 大(即 k 低 n 位中有1, 因 i 低 n 位全是0)

$k - \text{lowbit}(k) + 1$ 就是 k 去掉最右边的1, 然后再加1, 那当然还是比 i 大

初始状态下由**a**构建树状数组**C**的时间复杂度是多少？

显然是 $O(N)$ 的

因为

$$C[k] = \text{sum}(k) - \text{sum}(k - \text{lowbit}(k))$$

证：

$$\text{sum}(k) = C[n_1] + C[n_2] + \dots + C[n_{m-1}] + C[n_m] \quad (n_m = k)$$

$$n_{m-1} = k - \text{lowbit}(k)$$

$$\text{sum}(k - \text{lowbit}(k)) = C[n_1] + C[n_2] + \dots + C[n_{m-1}]$$

所以，树状数组适合**单个**元素经常修改而且还反复要求部分的区间的和的情况。

上述问题虽然也可以用线段树解决，但是用树状数组来做，编程效率和程序运行效率都更高（时间复杂度相同，但是树状数组常数小）

如果每次要修改的不是单个元素，而是一个区间，那就不能用树状数组了(效率过低)。

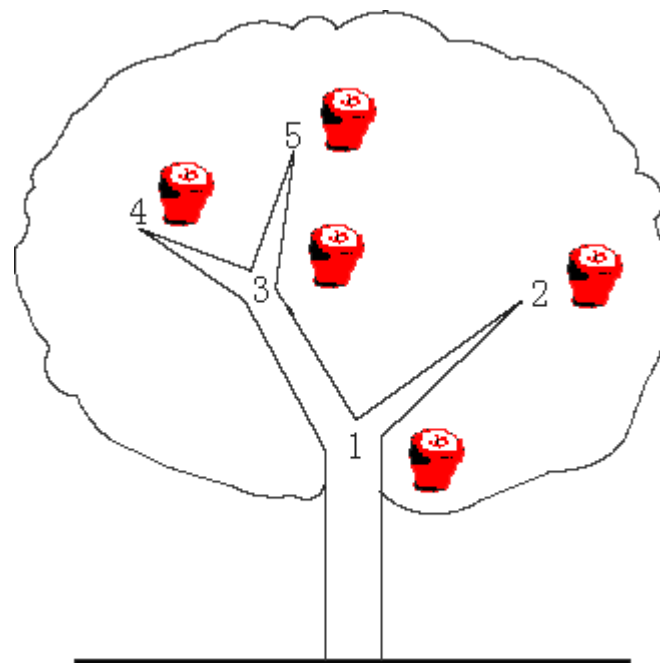
树状数组时间复杂度总结：

建数组： $O(n)$

更新： $O(\log n)$

局部求和： $O(\log n)$

POJ 3321 Apple Tree



每个分叉点及末梢可能有苹果（最多1个），
每次可以摘掉一个苹果，或有一个苹果新长
出来，随时查询某个分叉点往上的子树里，
一共有多少个苹果。 (分叉点数： 100,000)
此题可用树状数组来做

根据题意，一开始时，所有能长苹果的地方都有苹果

Sample Input

3

1 2

1 3

3

Q 1

C 2

Q 1

Sample Output

3

2

//树状数组做

/*

一棵树上长了苹果，每一个树枝节点上有长苹果和不长苹果两种状态，两种操作，一种操作能够改变树枝上苹果的状态，另一种操作询问某一树枝节点以下的所有的苹果有多少。具体做法是做一次dfs，记下每个节点的开始时间**Start[i]**和结束时间**End[i]**，

那么对于i节点的所有子孙的开始时间和结束时间都应位于**Start[i]**和**End[i]**之间

然后用树状数组**C**统计**Start[i]**到**End[i]**之间的附加苹果总数。这里用树状数组统计区间可以用**Sum(End[i])-Sum(Start[i]-1)**来计算。

*/

```
#include <iostream>
```

```
#include <vector>
```

```
using namespace std;
```

```
#define MY_MAX 220000
```

```
int C[MY_MAX];
typedef vector<int> VCT_INT;
vector<VCT_INT> G(MY_MAX/2); //邻接表
int Lowbit[MY_MAX];
bool HasApple[MY_MAX/2];
int Start[MY_MAX]; //dfs时的开始时间
int End[MY_MAX]; //dfs时的结束时间
int nCount = 0;
void Dfs(int v)
{
    Start[v] = ++ nCount;
    for( int i = 0; i < G[v].size(); i ++ )
        Dfs(G[v][i]);
    End[v] = ++ nCount;
}
```

```
int QuerySum(int p) //求 sum(p)
{
    int nSum = 0;
    while( p > 0 ) {
        nSum += C[p];
        p -= Lowbit[p];
    }
    return nSum;
}

void Modify( int p,int val)
{
    while( p <= nCount ) {
        C[p] += val;
        p += Lowbit[p];
    }
}
```



```
int main()
{
    int n;
    scanf("%d",&n);
    int x,y;
    int i,j,k;
    //建图
    for( i = 0;i < n -1 ;i ++ ) {
        int a,b;
        scanf("%d%d",&a,&b);
        G[a].push_back(b); //a有边连到b
    }
    nCount = 0;
    Dfs(1);
}
```

//树状数组要处理的原始数组下标范围 1 --
nCount

```
for( i = 1;i <= nCount;i ++ ) {  
    Lowbit[i] = i & ( i ^( i - 1));  
}
```

```
for( i = 1;i <= n;i ++ )  
    HasApple[i] = 1;
```

```
int m;
```

//求C数组，即树状数组的节点的值

```
for( i = 1;i <= nCount;i ++ )  
    C[i] = i - ( i - Lowbit[i]);  
    // C[i] = Sum[i] - Sum[i-lowbit(i)]
```

```

scanf("%d",&m);
for( i = 0;i < m;i ++ ) {
    char cmd[10];
    int a;
    scanf("%s%d",cmd,&a);
    if( cmd[0] == 'C' ) {
        if( HasApple[a] ) {
            Modify( Start[a],-1); Modify( End[a],-1);
            HasApple[a] = 0;
        }
        else {
            Modify( Start[a],1); Modify( End[a],1);
            HasApple[a] = 1;
        }
    }
    else {
        int t1 = QuerySum(End[a]);
        int t2 = QuerySum(Start[a]-1);
        printf("%d\n",(t1-t2)/2 );
    }
}
}

```

二维树状数组

- 原始数组和树状数组都是二维的
- $C[x][y] = \sum \{a[i][j]\}$
- $x - \text{lowbit}[x] + 1 \leq i \leq x$
- $y - \text{lowbit}[y] + 1 \leq j \leq y$
- $\text{Sum}[x][y] = \sum \{C[i][j]\}$ (从[1,1] 到[x,y]这个a矩阵里的所有元素的和)
- $i_1 = x, i_2 = i_1 - \text{lowbit}(i_1), \dots, i_k = i_{k-1} - \text{lowbit}(i_{k-1}) \dots$
- $j_1 = y, j_2 = j_1 - \text{lowbit}(j_1), \dots, j_k = j_{k-1} - \text{lowbit}(j_{k-1}) \dots$
- 用于快速求数字子矩阵的和,更新和查询的时间复杂度都是 $\log(n) * \log(m)$, 建C数组复杂度 $O(m * n)$ (n,m分别为两维的大小)

POJ 1195 Mobile phones

一个由数字构成的大矩阵，开始是全0,能进行两种操作

- 1) 对矩阵里的某个数加上一个整数（可正可负）
- 2) 查询某个子矩阵里所有数字的和

要求对每次查询，输出结果

| Instruction | Parameters | Meaning |
|-------------|------------|---|
| 0 | S | Initialize the matrix size to $S * S$ containing all zeros. This instruction is given only once and it will be the first instruction. |
| 1 | X Y A | Add A to the number of active phones in table square (X, Y). A may be positive or negative. |
| 2 | L B R T | Query the current sum of numbers of active mobile phones in squares (X, Y), where $L \leq X \leq R$, $B \leq Y \leq T$ |
| 3 | | Terminate program. This instruction is given only once and it will be the last instruction. |

Sample Input

0 4

1 1 2 3

2 0 0 2 2

1 1 1 2

1 1 2 -1

2 1 1 2 3

3

Sample Output

3

4

//下面为二维树状数组解法

```
#include <iostream>
```

```
#include <vector>
```

```
#include <cstring>
```

```
using namespace std;
```

```
#define MY_MAX 1100
```

```
int C[MY_MAX][MY_MAX];
```

```
int Lowbit[MY_MAX];
```

```
int s;
```

```
void Add( int y, int x,int a)
```

```
{
```

```
    while( y <= s ) {
```

```
        int tmpx = x;
```

```
        while( tmpx <= s ) {
```

```
            C[y][tmpx] += a;
```

```
            tmpx += Lowbit[tmpx];
```

```
        }
```

```
        y += Lowbit[y];
```

```
    }
```

```
}
```

```
int QuerySum( int y, int x)
```

```
//查询第1行到第y行， 第1列到第x列的和
```

```
{
```

```
    int nSum = 0;
```

```
    while( y > 0 ) {
```

```
        int tmpx = x;
```

```
        while( tmpx > 0 ) {
```

```
            nSum += C[y][tmpx];
```

```
            tmpx -= Lowbit[tmpx];
```

```
        }
```

```
        y -= Lowbit[y];
```

```
    }
```

```
    return nSum;
```

```
}
```



```

int main() {
    int cmd; int x,y,a,l,b,r,t;    int i,j,k; int n1,n2;
    for( i = 1; i <= MY_MAX; i ++ ) Lowbit[i] = i & ( i ^(i - 1));
    while( true) {
        scanf("%d",&cmd);
        switch( cmd) {
            case 0:
                scanf("%d",& s);
                memset( C,0,sizeof(C));
                break;
            case 1:
                scanf("%d%d%d",&x ,&y,&a);
                Add( y + 1, x + 1, a);
                break;
            case 2:
                scanf("%d%d%d%d",&l , &b, &r,&t);
                int n1,n2,n3,n4;
                l ++; b++; r ++; t ++;
                printf("%d\n",QuerySum(t,r) +
                QuerySum(b-1,l-1) - QuerySum(t,l-1) - QuerySum(b-1,r));
                break;
            case 3: return 0;
        }
    }
}

```

二维线段树

- 一维线段树的每个节点代表一个一维区间，那么二维线段树的每个节点就代表一个二维区间（一个矩阵）
- 二维线段树是一棵4叉树。如果一个节点代表的区间是矩阵 $[x1, y1] - [x2, y2]$ ，那么它的四个子节点代表的区间分别为
- $[x1, y1] - [(x1+x2)/2, (y1+y2)/2]$
 $[x1, (y1+y2)/2+1] - [(x1+x2)/2, y2]$
- $[(x1+x2)/2+1, y1] - [x2, (y1+y2)/2]$
- $[(x1+x2)/2+1, (y1+y2)/2+1] - [x2, y2]$

二维线段树

- 二维线段树的四叉树实现形式，请参考：
：
- <http://www.cppblog.com/menjitianya/archive/2011/04/03/143374.html>

二维线段树

- 还可以用树套树方式实现，即每个外层线段树的节点对应于一棵内层线段树。如果外层线段树根对应的区间是y方向的 $[1, n]$ ，内层线段树根节点对应的区间是x方向的 $[1, m]$ ，那么整个线段树可以存在一个二维数组里。
- 树套树方式插入、修改、查找等时间复杂度为 $O(\log(n) * \log(m))$ 。

二维线段树

```
int Tree[MY_MAX * 3][MY_MAX * 3];
```

//二维线段树，每一行都是一棵完全二叉树，用于存放一棵x方向线段树（列线段树），树节点只存放对应的区间（矩形）的数字之和

数组开多大就够？

- $9 * 9$ 矩阵的行线段树(列线段树结构一样) :



问: $\text{Tree}[2][4]$
存放的是哪个矩
阵的和?

$\text{Tree}[0][0]$ 是二维线段树的根节点, 里面存放的是整个矩阵的和

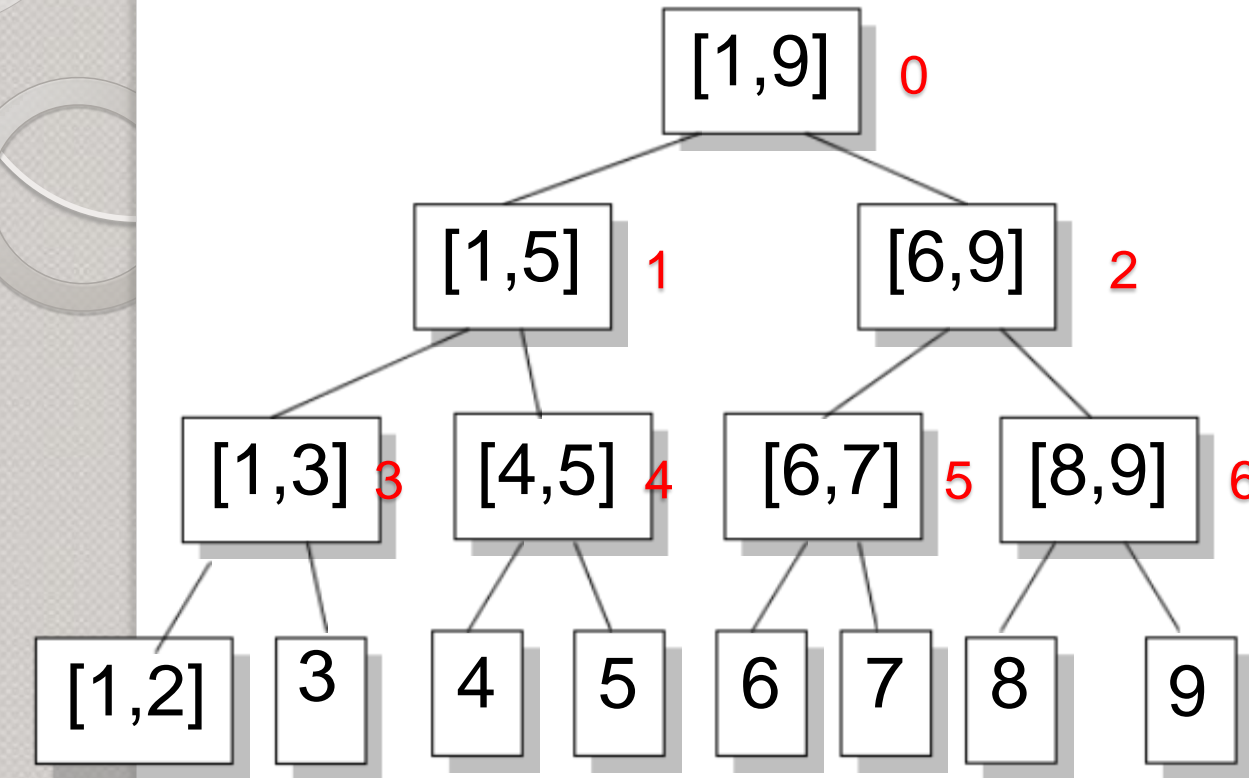
$\text{Tree}[1][0]$ 放着的是该矩形的和 $y : 1 - 5, x : 1 - 9$

$\text{Tree}[2][0]$ 放着的是该矩形的和 $y : 6 - 9, x : 1 - 9$

$\text{Tree}[3][0]$ 放着的是该矩形的和 $y : 1 - 3, x : 1 - 9$

$\text{Tree}[4][0]$ 放着的是该矩形的和 $y : 4 - 5, x : 1 - 9$

- $9 * 9$ 矩阵的行线段树(列线段树结构一样) :

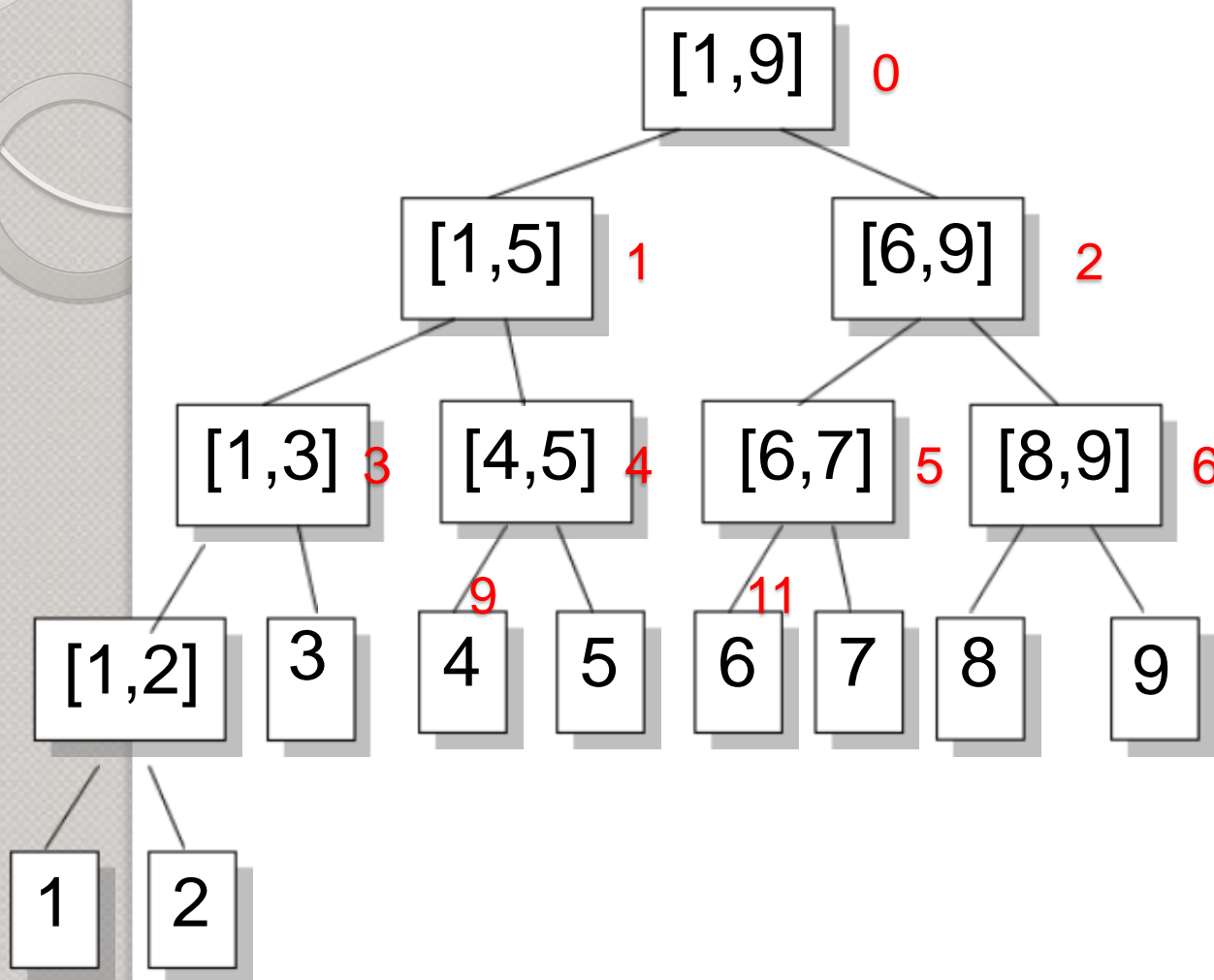


问: $\text{Tree}[2][4]$
存放的是哪个矩
阵的和?

$y : 6 - 9, x : 4-5$

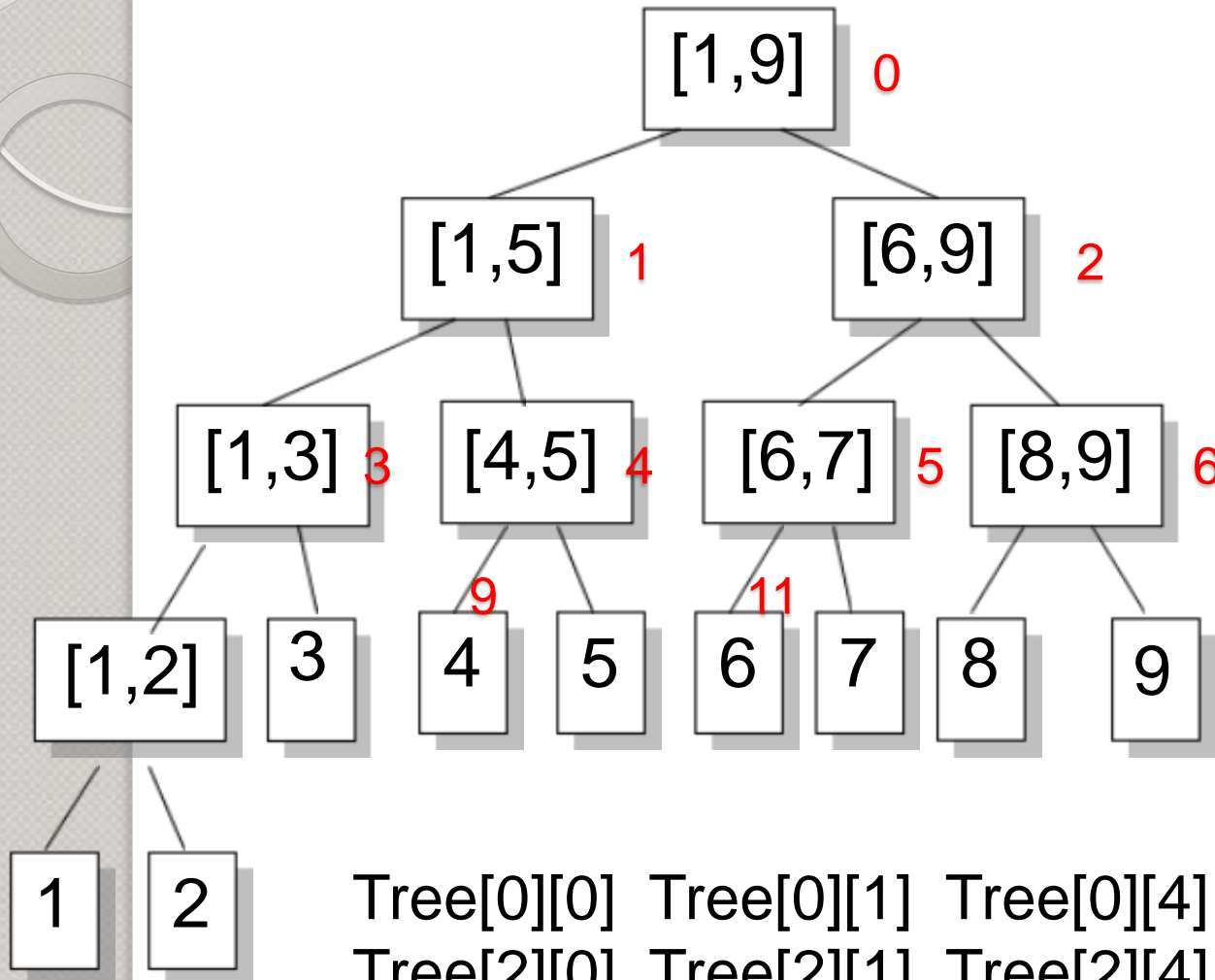
$\text{Tree}[0][0]$ 是二维线段树的根节点, 里面放的是整个矩阵的和
 $\text{Tree}[1][0]$ 放着的是该矩形的和 $y : 1 - 5, x : 1-9$
 $\text{Tree}[2][0]$ 放着的是该矩形的和 $y : 6 - 9, x : 1-9$
 $\text{Tree}[3][0]$ 放着的是该矩形的和 $y : 1 - 3, x : 1-9$
 $\text{Tree}[4][0]$ 放着的是该矩形的和 $y : 4 - 5, x : 1-9$
 $\text{Tree}[1][2]$ 放着的是该矩形的和 $y : 1 - 5, x : 6-9$

- $9 * 9$ 矩阵的行线段树(列线段树结构一样) :



问：对矩阵元素
(6,4) 加 a ,那么
Tree数组里面哪
些元素要修改？

- $9 * 9$ 矩阵的行线段树(列线段树结构一样) :



问：对矩阵元素
(6,4) 加a,那么
Tree数组里面哪
些元素要修改？

Tree[0][0] Tree[0][1] Tree[0][4] Tree[0][9]
 Tree[2][0] Tree[2][1] Tree[2][4] Tree[2][9]
 Tree[5][0] Tree[5][1] Tree[5][4] Tree[5][9]
 Tree[11][0] Tree[11][1] Tree[11][4] Tree[11][9]

用二维线段树做POJ 1195 Mobile phones

```
#include <iostream>
using namespace std;
#define MY_MAX 1100
int Tree[MY_MAX * 3][MY_MAX * 3]; //二维线段树，每一行都是一
//棵完全二叉树，用于存放一棵x方向线段树，树节点只存放 对应的
//区间（矩阵）的数字之和
int S; //矩阵宽度
void Add_x( int rooty,int rootx, int L, int R, int x ,int a)
//tree[rooty][rootx]对应的矩阵x方向上范围是[L,R]
{
    Tree[rooty][rootx] += a;
    if( L == R )
        return;
    int mid = (L + R )/2;
    if( x <= mid )
        Add_x(rooty,( rootx << 1) + 1, L ,mid, x, a);
    else
        Add_x(rooty,( rootx << 1) + 2, mid + 1,R, x, a);
}
```

```
void Add_y(int rooty, int L,int R, int y, int x,int a)
//tree[rooty][rootx]对应的矩阵y方向上范围是[L,R]
{
    Add_x( rooty,0, 1, S, x,a);
    if( L == R)
        return;
    int mid = (L + R )/2;
    if( y <= mid )
        Add_y( ( rooty << 1) + 1, L, mid,y, x, a);
    else
        Add_y( ( rooty << 1) + 2, mid+1, R, y, x, a);
}
```

```
int QuerySum_x(int rooty, int rootx, int L, int R ,int x1,int x2)
{
    if( L == x1 && R == x2)
        return Tree[rooty][rootx];
    int mid = ( L + R ) /2 ;
    if( x2 <= mid )
        return QuerySum_x( rooty, (rootx << 1) + 1,
                           L, mid,x1,x2);
    else if( x1 > mid )
        return QuerySum_x( rooty, (rootx << 1) + 2,
                           mid+1,R, x1,x2);
    else
        return QuerySum_x( rooty,(rootx << 1) + 1,
                           L, mid ,x1,mid) +
               QuerySum_x( rooty,(rootx << 1) + 2,
                           mid + 1, R, mid + 1,x2);
}
```



```

int main()
{
    int cmd; int x,y,a,l,b,r,t;    int Sum = 0;
    while( true) {
        scanf("%d",&cmd);
        switch( cmd) {
            case 0:
                scanf("%d",& S);
                memset( Tree,0,sizeof(Tree));
                break;
            case 1:
                scanf("%d%d%d",&x ,&y,&a);
                Add_y(0, 1,S, y + 1, x + 1, a);
                break;
            case 2:
                scanf("%d%d%d%d",&l , &b, &r,&t);
                l ++; b++; r ++; t ++;
                printf("%d\n",QuerySum_y(0,1,S,b,t,l,r));
                break;
            case 3:
                return 0;
        }
    }
}

```

二维线段树好题：POJ 2155 Matrix

$N * N$ 的矩阵，每个元素要么是0，要么是1，开始全0

不断进行两种操作：

- 1) $C\ x1\ y1\ x2\ y2$ 表示要将左上角为 $(x1,y1)$,右下角为 $(x2,y2)$ 的子矩阵里的全部元素都取反（0变1，1变0）
 $(1 \leq x1 \leq x2 \leq n, 1 \leq y1 \leq y2 \leq n)$
- 2) $Q\ x\ y$ 查询 (x,y) 处元素的值



POJ题目推荐：

2182, 2352, 1177, 3667,3067