线段树基础

衡水第一中学

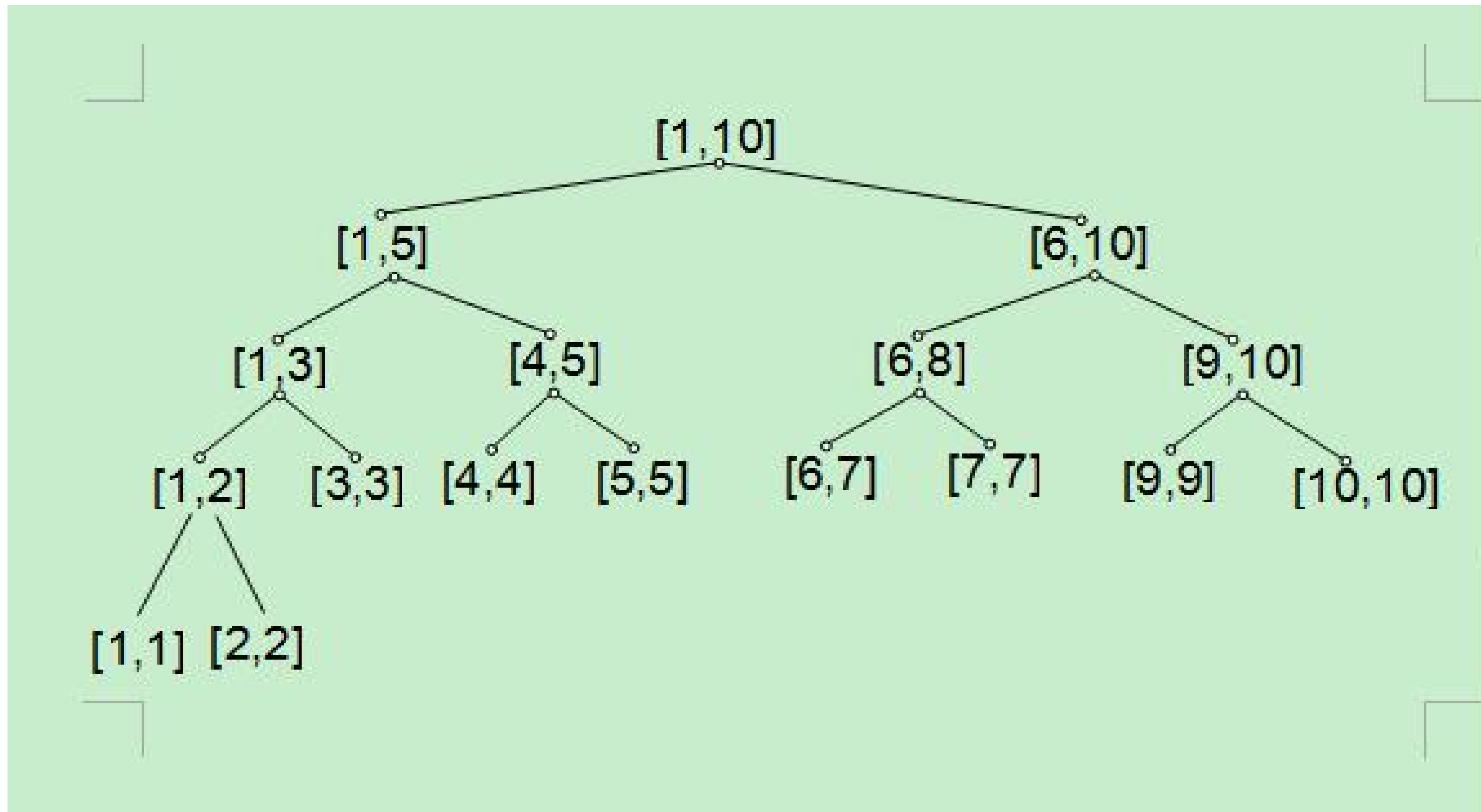
线段树

- 在一类问题中，我们需要经常处理可以映射在一个坐标轴上的一些固定线段，例如映射在X轴上的线段。
- 由于线段是可以互相覆盖的，有时需要动态地取线段的并，例如取得并区间的总长度，或者并区间的个数等等。一个线段是对应于一个区间的，因此线段树也可以叫做区间树。
- 在这类问题中，线段树的一个结点表示一条线段。

线段树的构造思想

- 线段树是一棵二叉树，树中的每个结点表示了一个区间[a, b]。每一个叶子结点表示了一个单位区间。
- 对于每一个非叶子结点所表示的区间[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]。

线段树的模型

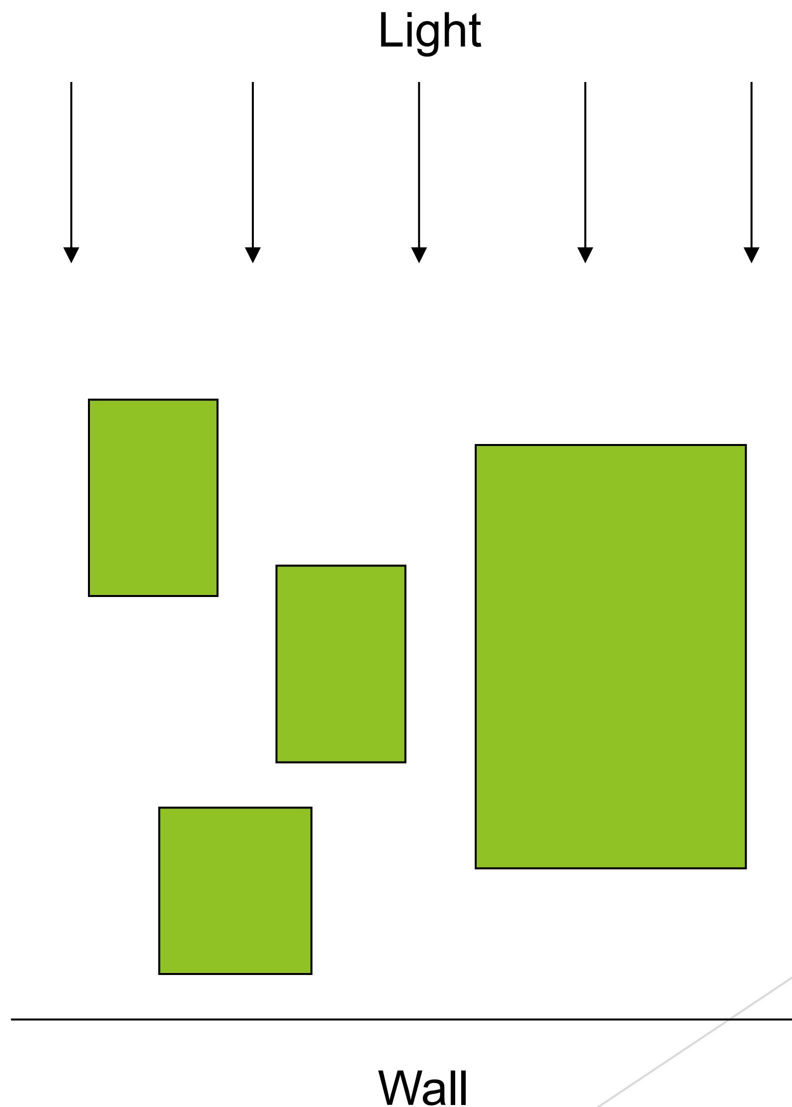


线段树的运用

- 线段树的每个节点上往往都增加了一些其他的域。在这些域中保存了某种动态维护的信息，视不同情况而定。这些域使得线段树具有极大的灵活性，可以适应不同的需求。
- 例如区间和，区间最值等等

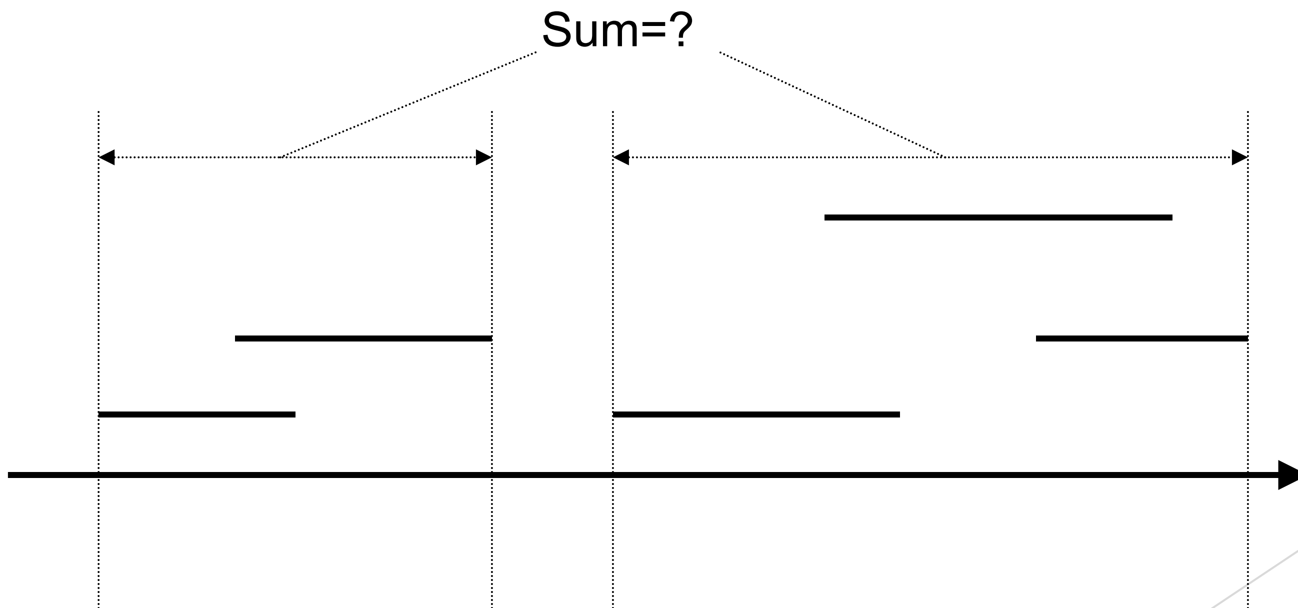
例题1

- 桌子上零散地放着若干个盒子，桌子的后方是一堵墙。如右图所示。现在从桌子的前方射来一束平行光， 把盒子的影子投射到了墙上。问影子的总宽度是多少？



分析

- 这道题目是一个经典的模型。在这里，我们略去某些处理的步骤，直接分析重点问题，可以把题目抽象地描述如下：x轴上有若干条线段，求线段覆盖的总长度。



最朴素的做法

- 设线段坐标范围为[min,max]。使用一个下标范围为[min,max-1]的一维数组，其中数组的第i个元素表示[i,i+1]的区间。数组元素初始化全部为0。
- 对于每一条区间为[a,b]的线段，将[a,b]内所有对应的数组元素均设为1。最后统计数组中1的个数即可。

示例

- 初始情况

0	0	0	0	0

- [1，2]

1	0	0	0	0

- [3，5]

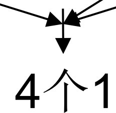
1	0	1	1	0

- [4，6]

1	0	1	1	1

- [5，6]

1	0	1	1	1



缺点

- 此方法的时间复杂度决定于下标范围的平方。
- 当下标范围很大时（[0,100000]），此方法效率太低。

好的办法当然是今天要讲的线段树啦

- 如何实现？？？

线段树的数据结构

- 完全二叉树
- 动态数据结构（知道有这么回事就行，暂时不学指针）

线段树的数据结构——完全二叉树

```
struct Node{
    int l, r; // 节点表示的区间
    int max, sum; // 或者其他数据域
}tree[MAXN << 2];
// 一般大小开成节点数的4倍，不需要担
// 心空间问题
```

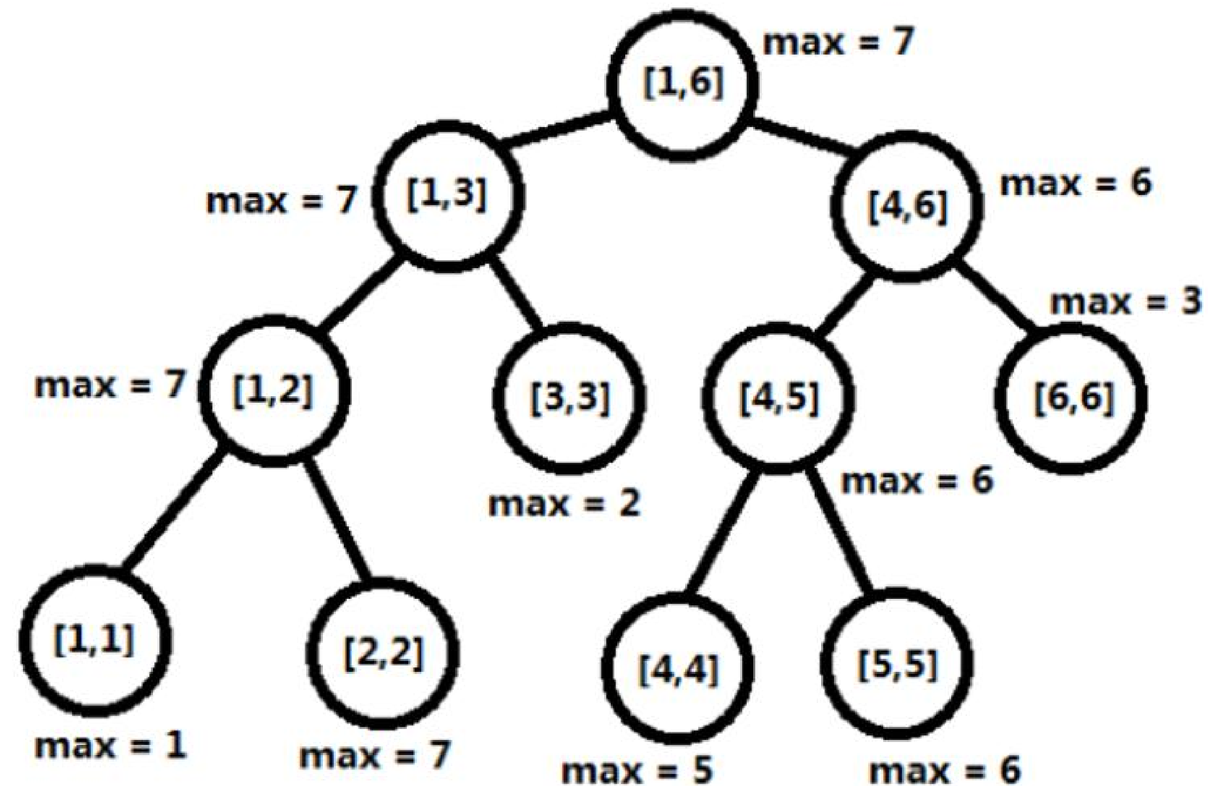
线段树的数据结构——完全二叉树

- 线段树的每一个节点要记录该节点的左右端点值，区间和、区间最大值等信息，所以要用结构体数组。
- 由于使用完全二叉树的结构，因此对于节点 k，表示的区间为 [l, r]，那么：
  左儿子编号：k << 1，表示的区间为 [l, (l+r)>>1]
  右儿子编号：k << 1 | 1，表示的区间为 [(l+r>>1)+1, r]
  因为要频繁用到找左右儿子节点，我们常常把求左右儿子编号写成预编译命令：

```
#define lson (rt<<1)
#define rson (rt<<1|1)
```

线段树的数据结构——举例

- 节点内的区间表示节点的管辖区间
- 节点旁边的标识为节点维护的区间信息。以区间最大值为例，需要存储一个当前区间内的最大值max，以数组 A[1~6]={1, 7, 2, 5, 6, 3}为例，如右图



线段树的题目分类

- 单点更新（最基础的线段树，只更新叶子结点，然后回溯更新其父亲结点）
- 区间更新加延迟标记优化（每次更新的时候，如果可以的话，则不必更新到叶子结点，而且在当前结点做个标记，使得更新操作延迟到以后需要向下更新或者询问的时候再进行）
- 区间合并
- 扫描线（典型题目：矩形面积并，周长并等）

线段树常见的操作

- 建线段树（即初始化 Build）
- 更新操作（Update）
- 查询操作（Query）
- 向上回溯（Pushup）
- 向下延迟更新（Pushdown）

- 一般就这5个函数

建线段树

- 从头建一棵线段树，可以这样调用
- Build(1, 1, n);

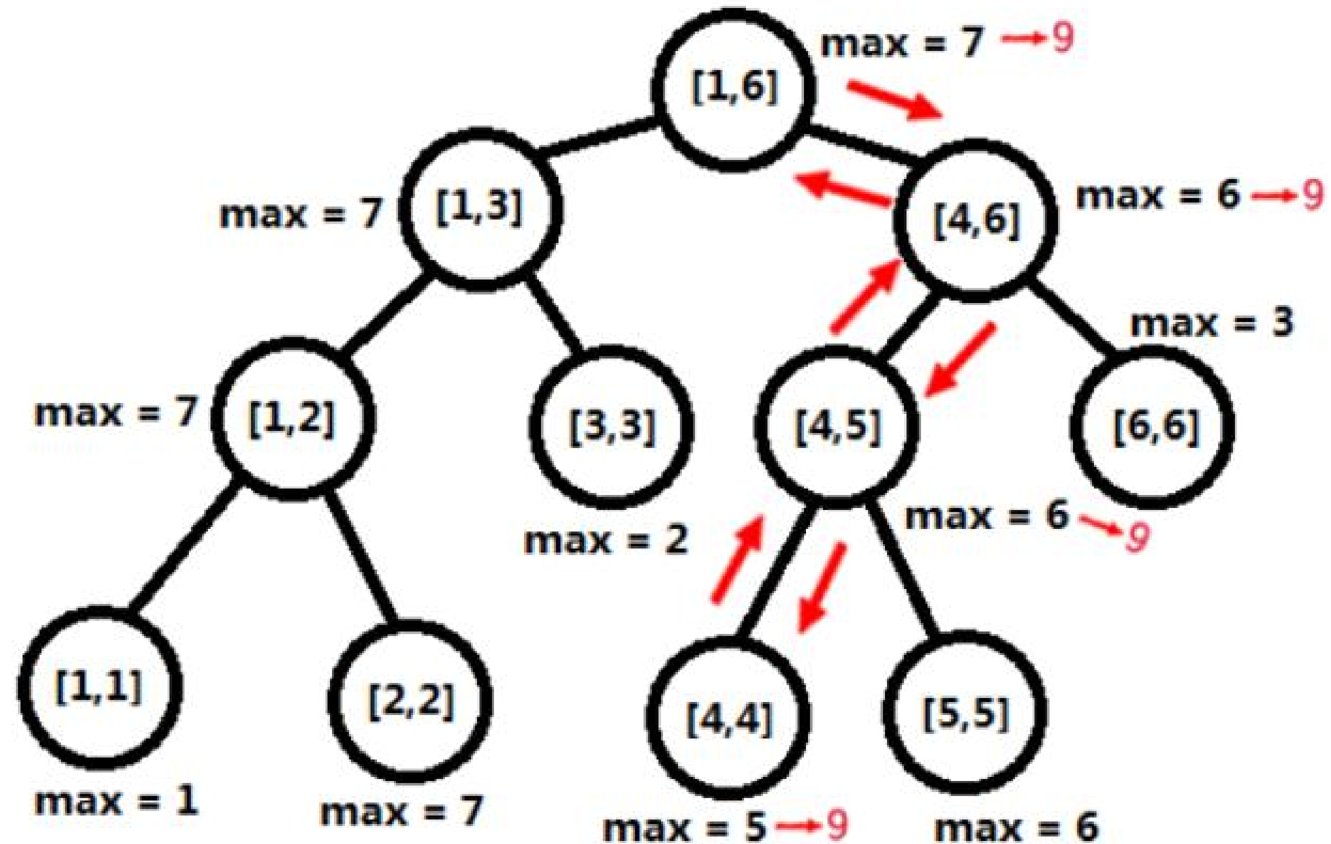
```
// 预编译命令, 做符号代换
#define lson (rt << 1)
#define rson (rt << 1 | 1)

// rt表当前结点, 表示区间范围[l, r]
void Build (int rt, int l, int r) {
    tree[rt].l = l;
    tree[rt].r = r; //结点信息的初始化
    if (l == r) { //到叶结点
        tree[rt].sum = tree[rt].max = a[l];
        return;
    }
    int mid = (l + r)>>1;
    Build (lson, l, mid);
    Build (rson, mid+1, r);

    // 子树建好后, 回溯时更新父节点信息
    // 以后我们一般写成一个函数叫 pushup(rt);
    tree[rt].sum = tree[lson].sum + tree[rson].sum;
    tree[rt].max = max(tree[lson].max, tree[rson].max);
}
```

更新操作——单点更新

- 更新数组元素的值时，我们首先递归到对应折叶子节点，修改其数据域的信息，再在回溯时逐个更新父节点的信息

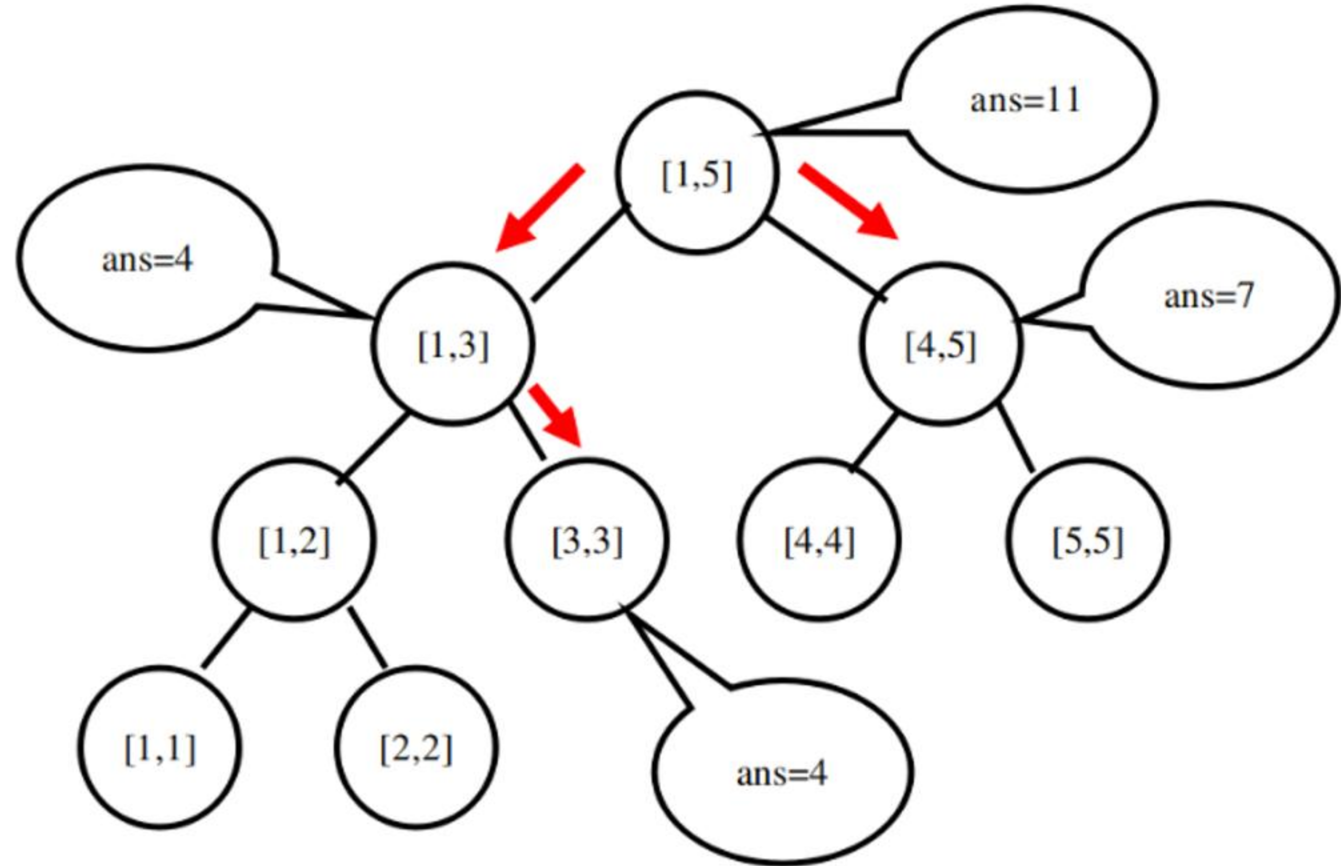


更新操作——单点更新

```
void Update(int rt, int pos, int val) {
    if (tree[rt].l == tree[rt].r) { // 找到对应的叶子节点
        tree[rt].sum = tree[rt].max = val;
        return;
    }
    int mid = (tree[rt].l + tree[rt].r) >> 1;
    if (pos <= mid) Update(lson, pos, val);
    else Update(rson, pos, val);
    // 回溯更新父节点信息
    tree[rt].sum = tree[lson].sum + tree[rson].sum;
    tree[rt].max = max(tree[lson].max, tree[rson].max);
}
```

查询操作——查询区间和

- 以查询藏金阁为例，数组 [1, 5, 4, 1, 6]，查询 sum(3, 5)
- 箭头标示递归查询的方向，在回溯的时候返回查询结果，并汇总



查询操作——查询区间和

```
// 当前节点为rt, 要查询的区间是[l, r]
int Query(int rt, int l, int r) {
    // 如果节点表示的区间是查询区间的真子集
    if (l <= tree[rt].l && tree[rt].r <= r)
        return tree[rt].sum;

    int mid = (tree[rt].l + tree[rt].r) >> 1;
    if (r <= mid) return Query(lson, l, r);
    else if (l > mid) return Query(rson, l, r);
    else return (Query(lson, l, r) + Query(rson, l, r));
}
```

向上回溯

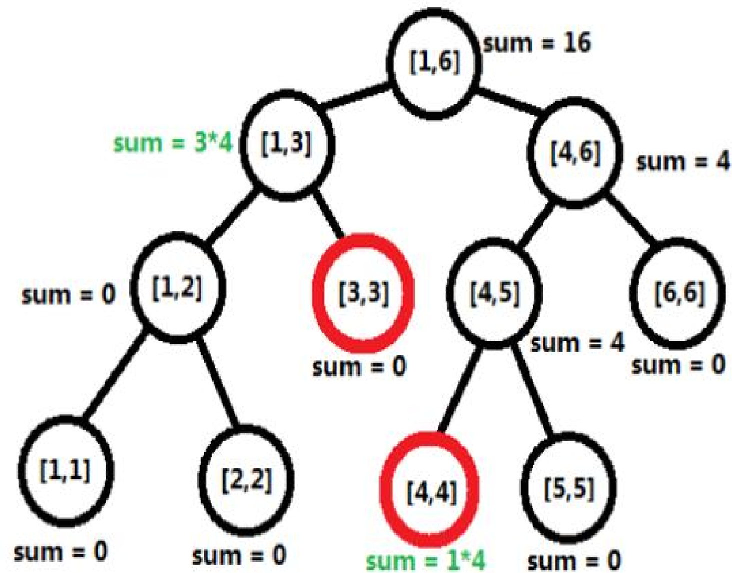
该函数就是由子结点递归回来，修改父结点中的信息
由于一般建树和更新都有这个相同的操作，因此我们可以写成一个函数，简化代码量

```
void Pushup(int rt)
{
    tree[rt].sum = tree[lson].sum + tree[rson].sum;
    tree[rt].max = max(tree[lson].max, tree[rson].max);
}
```

更新操作——区间更新

- 当需要更新某一区间内的元素值的时候，我们可以拆成多个单点更新的形式，但是肯定会导致算法时间复杂度的增加。
- 如果只更新到区间完全覆盖的节点表示的区间时，对后面的更新和查询会造成计算的遗留，如图所示

用一个数据域sum来记录线段树结点区间上所有元素的和，初始化所有结点的sum值都为0，然后在区间[1, 4]上给每个元素加上4，如图



操作完之后询问区间[3,4]?

延迟标记

- 做区间更新时，如果要更新的区间能够完全覆盖当前节点表示的区间，则在此节点上做个标记（表示此节点曾被修改，但子节点尚未被更新），不再继续向下更新，同时在回溯时更新父节点的信息。
- 如果在之后的维护或查询过程中需要对这个节点的某个儿子递归地进行处理，则将这个标记分解，传递给它的两个儿子节点。
- 这种在需要的时候才进行分解的做法，使我们整体的时间复杂度仍在$O(\log_2 N)$ 的水平上。

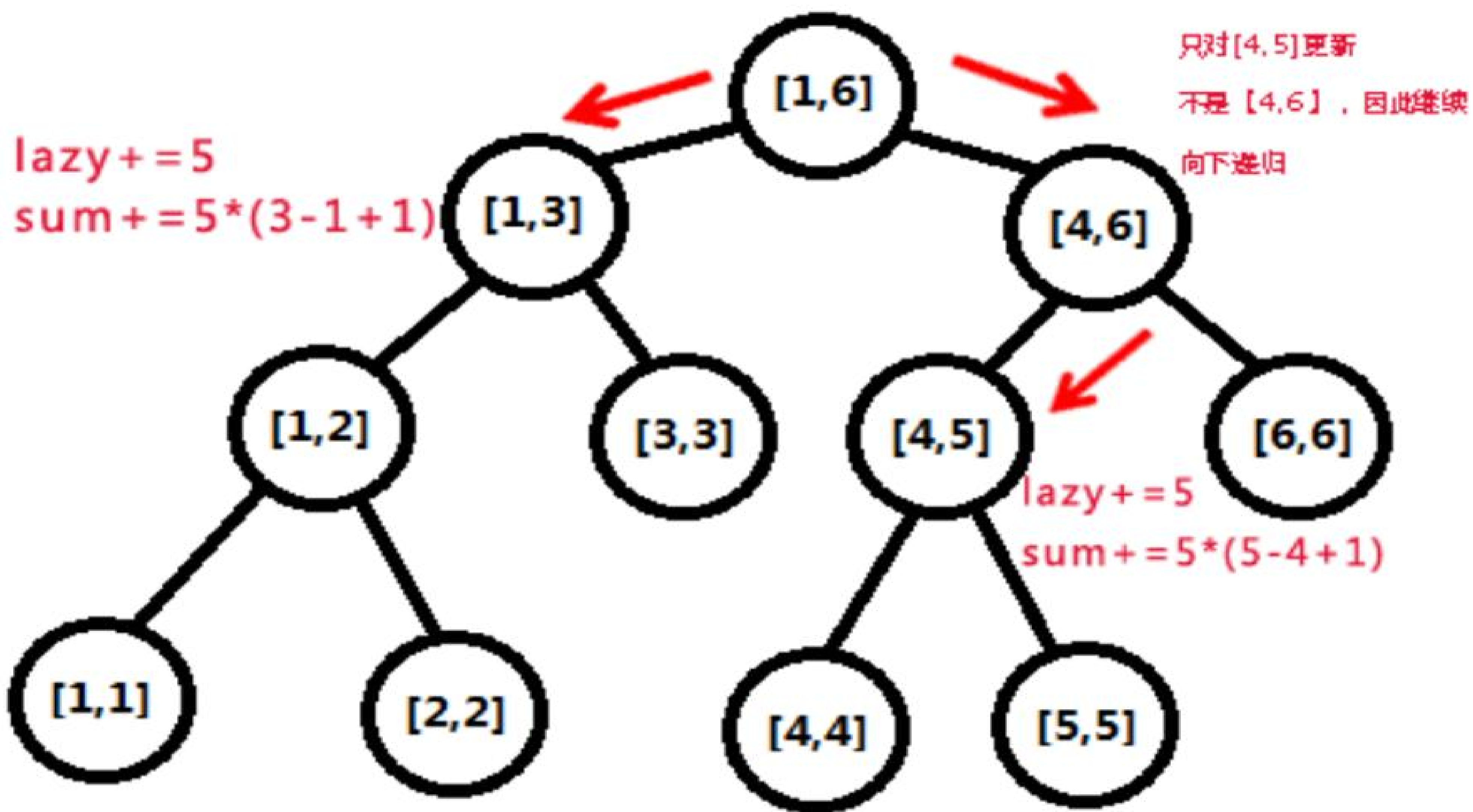
延迟标记

结构体的定义：

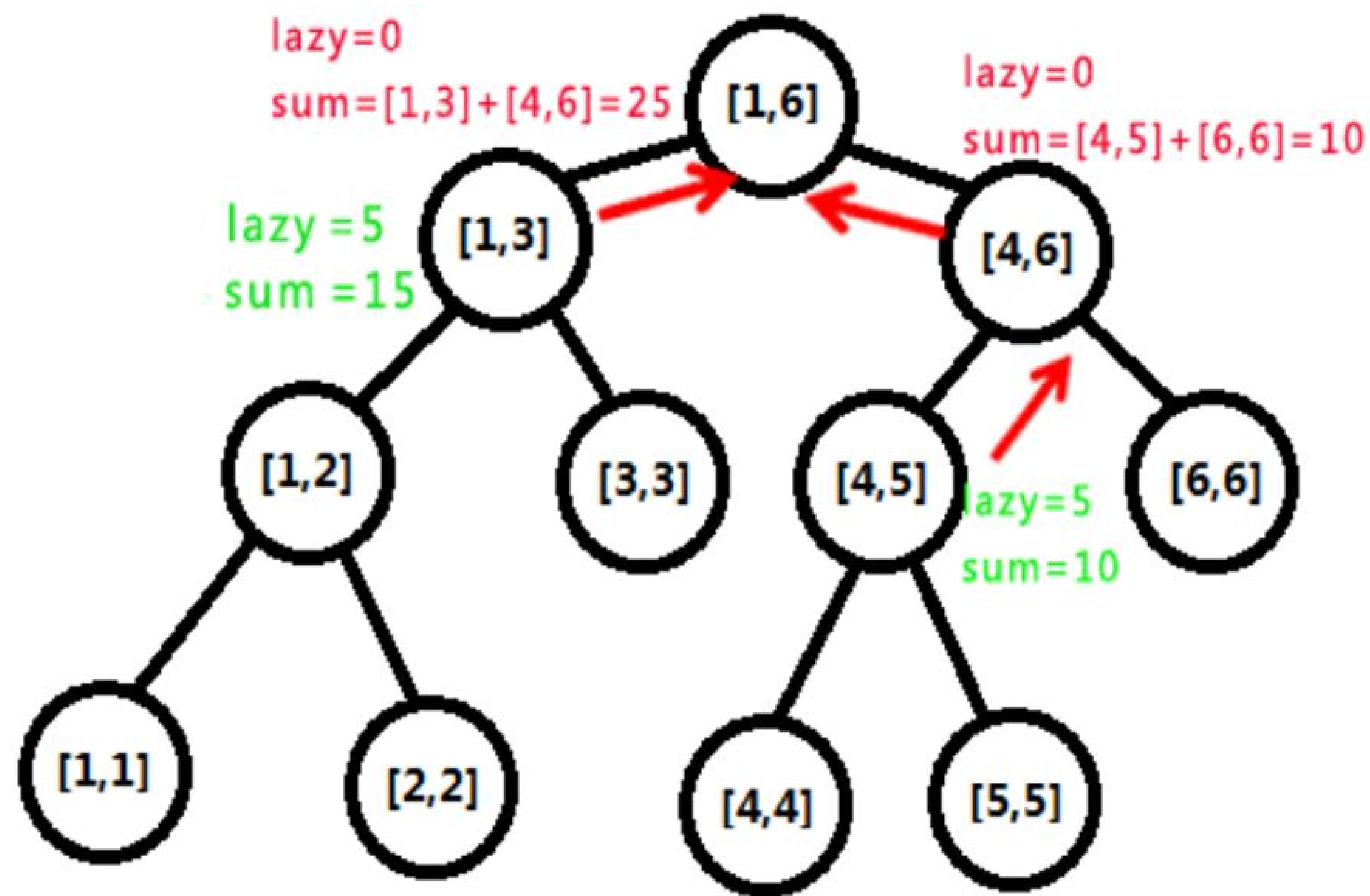
```
struct Node{
    int l, r; // 节点表示的区间
    int max, sum; // 或者其他数据域
    int lazy; // 延迟标记
}tree[MAXN << 2];
```

建树过程和单点更新，跟之前是一样的

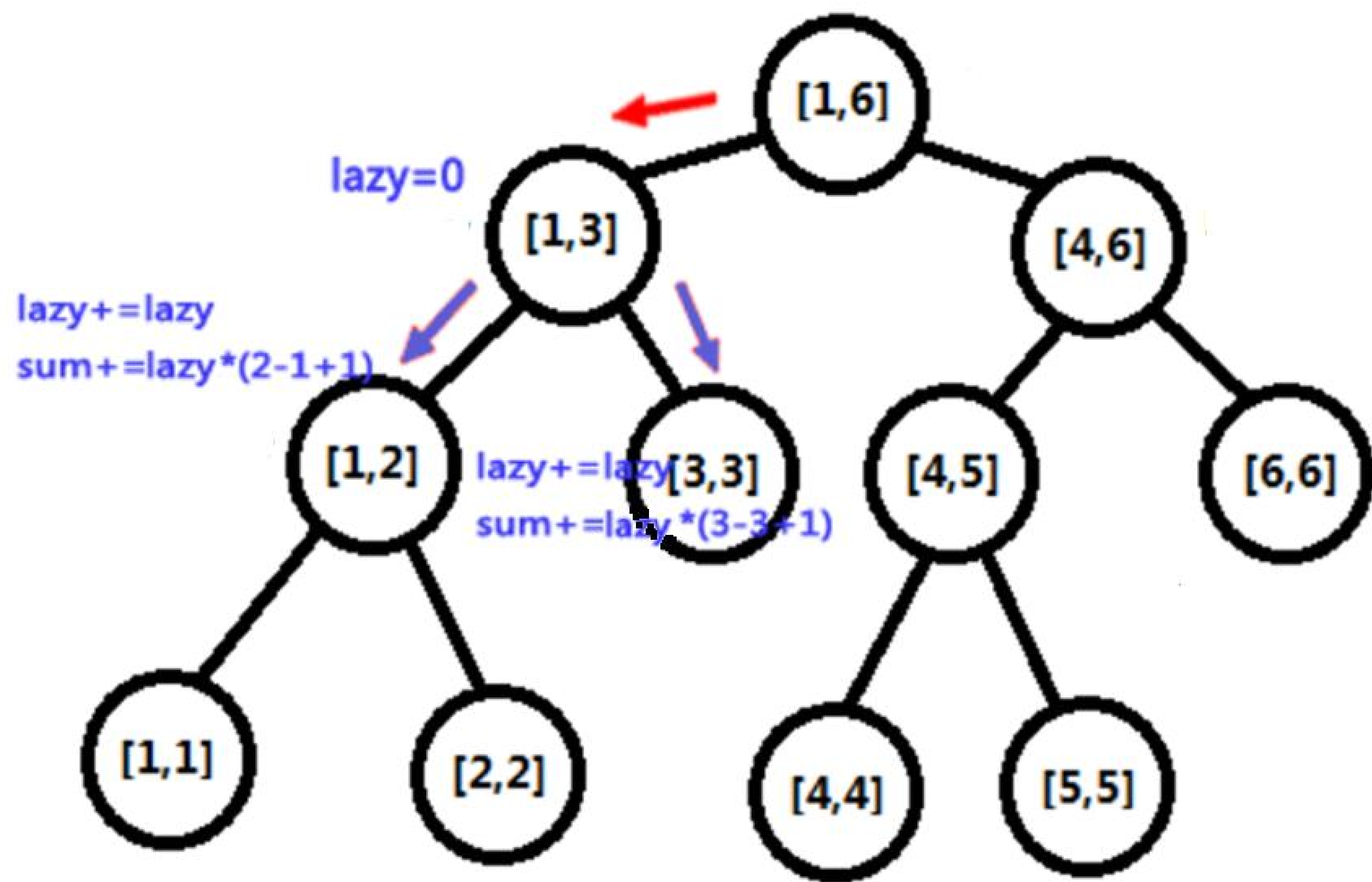
区间更新: [1,5] add 5



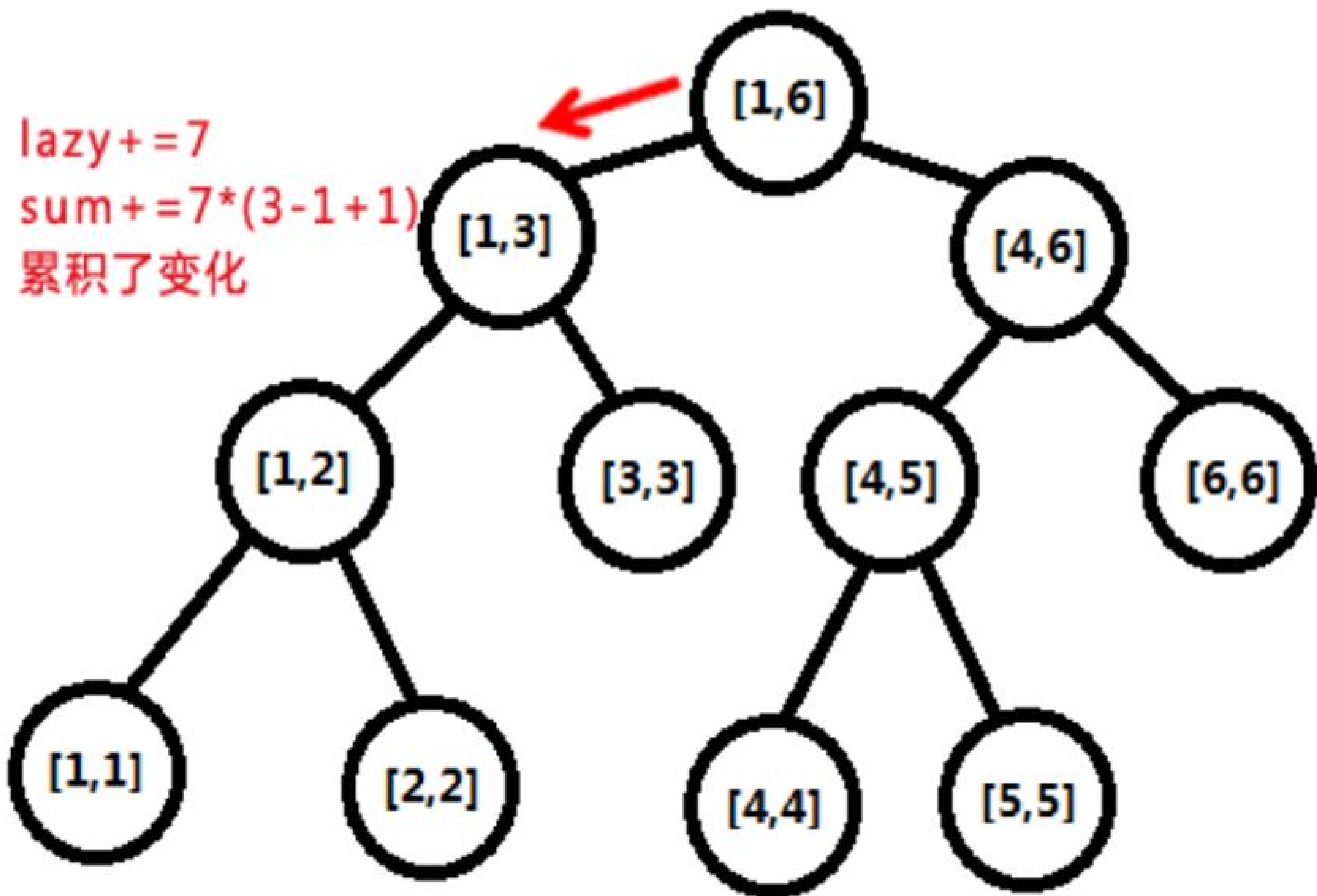
区间更新: [1,5] add 5



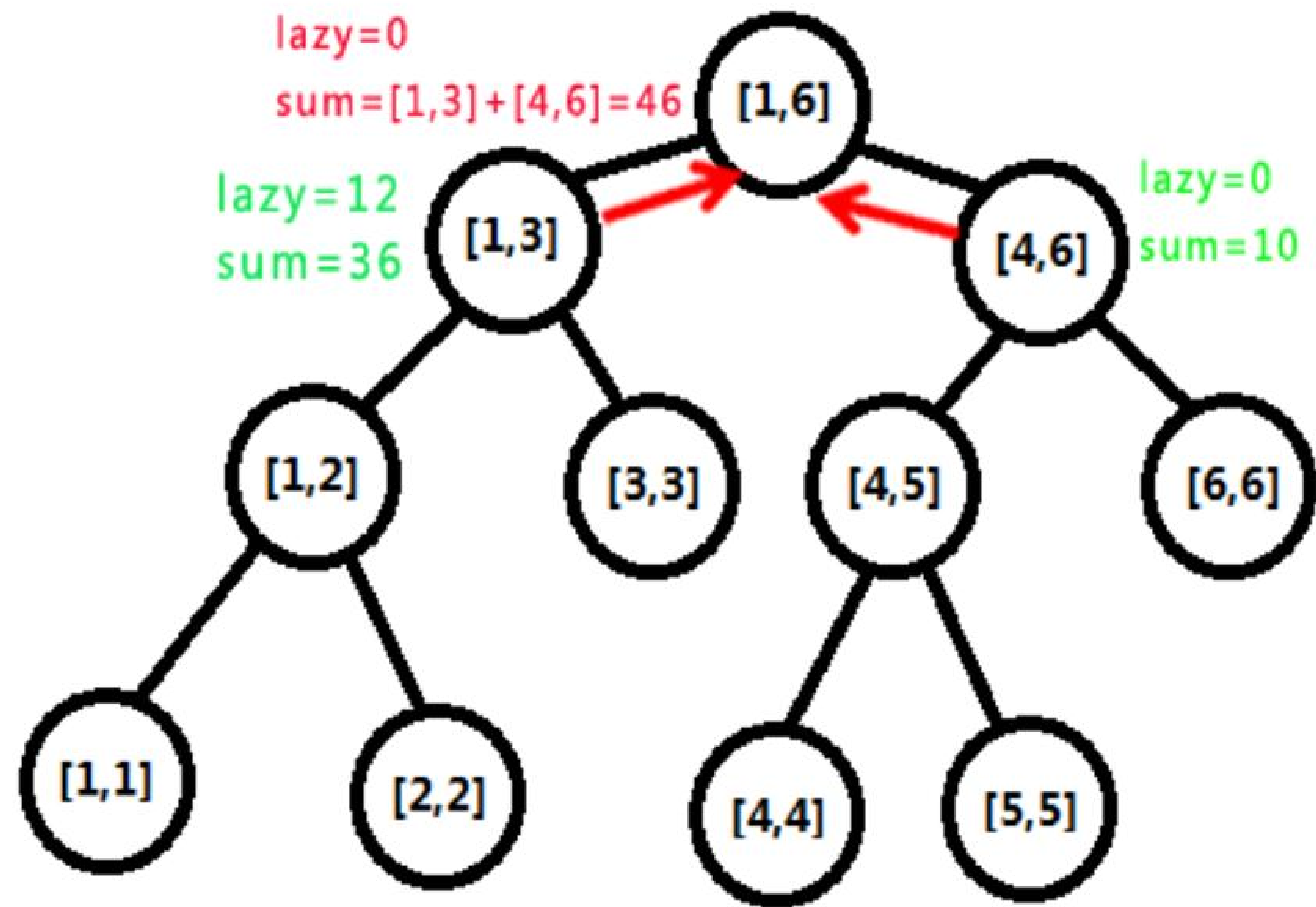
区间查询: [3,3]



区间更新: [1,3] add 7



区间更新: [1,3] add 7



延迟标记下放的示例代码

```
// 把当前节点rt的延迟标记下放到左右儿子
void Pushdown(int rt) {
    if (tree[rt].lazy) { // 此节点有延迟标记
        int lz = tree[rt].lazy;
        tree[rt].lazy = 0; // 记住要清零
        tree[lson].lazy += lz;
        tree[rson].lazy += lz;
        tree[lson].sum += lz * (tree[lson].r - tree[lson].l + 1);
        tree[rson].sum += lz * (tree[rson].r - tree[rson].l + 1);
    }
}
```

区间更新（加）的示例代码

```
void Update(int rt, int l, int r, int val) {
    // 更新区间完全覆盖节点表示的区间
    if (l <= tree[rt].l && tree[rt].r <= r) {
        tree[rt].lazy += val;
        tree[rt].sum += val * (tree[rt].r - tree[rt].l + 1);
        return;
    }
    // 如果不能完全覆盖，此时需要向下递归，要下放标记
    Pushdown(rt);
    int mid = (tree[rt].l + tree[rt].r) >> 1;
    if (l <= mid) Update(lson, l, r, val);
    if (r > mid) Update(rson, l, r, val);
    Pushup(rt); // 更新完回来记得要更新当前节点的信息
}
```

区间查询（求和）的示例代码

```
int Query(int rt, int l, int r) {
    // 更新区间完全覆盖节点表示的区间
    if (l <= tree[rt].l && tree[rt].r <= r) {
        return tree[rt].sum;
    }
    // 如果不能完全覆盖，此时需要向下递归，要下放标记
    Pushdown(rt);
    int mid = (tree[rt].l + tree[rt].r) >> 1;
    int val = 0;
    if (l <= mid) val += Query(lson, l, r);
    if (r > mid) val += Query(rson, l, r);
    return val;
}
```

例题：POJ 3468 A Simple Problem with Integers

题意：在一组数中执行两种操作

"C a b c" means adding c to each of Aa, Aa+1, ... , Ab.

$-10000 \le c \le 10000$.

"Q a b" means querying the sum of Aa, Aa+1, ... , Ab.

示例代码

```
struct Node
{
    int l,r;
    ll add, sum; // add作为延迟更新的标记
}tree[M<<2];

ll A[M]; // 原数据
```

示例代码

```
void Build (int u,int left,int right) {
    tree[u].l = left, tree[u].r = right;
    tree[u].add = 0;
    if (tree[u].l == tree[u].r) {
        tree[u].sum = A[left];
        return ;
    }
    int mid = (tree[u].l + tree[u].r)>>1;
    Build (u<<1,left,mid);
    Build ((u<<1)|1,mid+1,right);
    Pushup(u); // 回溯更新当前结点u的sum值
}
```

示例代码

```
void Pushup(int u) {
    tree[u].sum = tree[u<<1].sum + tree[(u<<1)|1].sum;
    return ;
}
void Pushdown (int u) {
    tree[u<<1].add += tree[u].add;
    tree[u<<1].sum += (tree[u<<1].r - tree[u<<1].l+1)*tree[u].add; //加上的是父结点的add
    tree[(u<<1)|1].add += tree[u].add;
    tree[(u<<1)|1].sum += (tree[(u<<1)|1].r - tree[(u<<1)|1].l+1)*tree[u].add;

    tree[u].add = 0; // 延迟更新的值向下传递完要及时清0
}
```

示例代码

```
void Update(int u, int l, int r, int val) {
    if(tree[u].l>r || tree[u].r<l) return ; // 完全没在这个区间，撤！
    if(tree[u].l>=l && tree[u].r<=r) { // 修改的区间包含当前结点
        tree[u].sum += (tree[u].r- tree[u].l+1)*val;
        tree[u].add += add;
        return ;
    }
    if(tree[u].add) Pushdown(u);
    update(u<<1, l, r, add);
    update((u<<1|1), l, r, add);
    Pushup(u);
}
```

示例代码

```
int query(int u, int l, int r) {
    if(tree[u].l>r || tree[u].r<l) return 0;
    if(tree[u].l>=l && tree[u].r<=r) {
        return tree[u].s;
    }
    if(tree[u].add) Pushdown(u);
    int ret = query(u<<1, l, r) + query((u<<1)|1, l, r);
    Pushup(u); // 这里可以不需要
    return ret;
}
```

示例代码

```
// 主函数部分
for(int i=1; i<=n; i++)
    scanf("%d",&hh[i]);
build(1, 1, n);
```

线段树大体构造容易实现
关键在于多练