21级实验室暑假第六讲

目录

CONTENTS

权值线段树

线段树是一种用于维护区间信息的高效数据结构，可以在 O (log N) 的时间复杂度内实现**单点修改、单点查询、区间修改、区间查询（区间求和，求区间最大值，求区间最小值）**等操作。

线段树维护的信息必须满足**可加性**，即能以可以接受的速度合并信息和修改信息，包括在使用懒标记时，标记也要满足可加性

权值线段树

权值线段树是一种建立在基本线段树之上的数据结构。因此它的基本原理仍是基于对区间的维护操作。但与线段树相区分的点是：权值线段树维护的信息与基本线段树有所差异：

- 基本线段树中每个结点一般维护一段区间的最大值或总和等；
- 权值线段树每个结点维护的是**一个区间的数出现的次数**。

实际上，权值线段树可以看作是一个桶，桶的操作它都支持，并且能够以更快的方式去完成。

权值线段树

那么什么是桶呢?

桶是一种数据结构。数据结构的用途是以**一种特殊方式统计数据，使得我们能够快速地修改、查询我们想要的那部分数据**。但是一般我们在想要统计一组数据的时候，我们更关注的是**这些数据都是什么**。就比如我们现在要统计一个数列，我们更关心的是这个数列里到底有那些数，而不是特别关心这些数都出现了几次。

桶就打破了这个现状，作为一种“特殊”的数据结构，它所统计的就是**每个数据在数据集合中一共出现了多少次**。

一般的数据结构，就好比有几个篮子A,B,C…，我们接到了一个新的数据，就要考虑将其按我们想要的那种方式分类，然后扔到某一个篮子里。桶呢？就好比有几个篮子1,2,3,4…，我们接到了一个新的数据，只看这个数据到底是什么，是1就扔进1号篮，2就扔进2号篮，以此类推。

权值线段树

根据权值线段树的性质，我们可以知道其主要用途：

权值线段树一般用于维护一段区间的数出现的次数，从它的定义来看，它可以快速计算出**一段区间的数的出现次数**。

在实际应用中，我们使用权值线段树查询区间**第K大的值**。

权值线段树

权值线段树的基本原理与操作

权值线段树维护信息的原理

基础线段树和权值线段树的一个较大的区别点在于：(此处特指使用数组储存树结构)基础线段树根据**初始的一个序列建立树结构，具有单独的建树操作**；而权值线段树不需要进行建树操作，初始状态下默认建立一棵所有点权为0的空树，对于每个元素，在遍历的时候相应的结点做自增运算。换而言之：

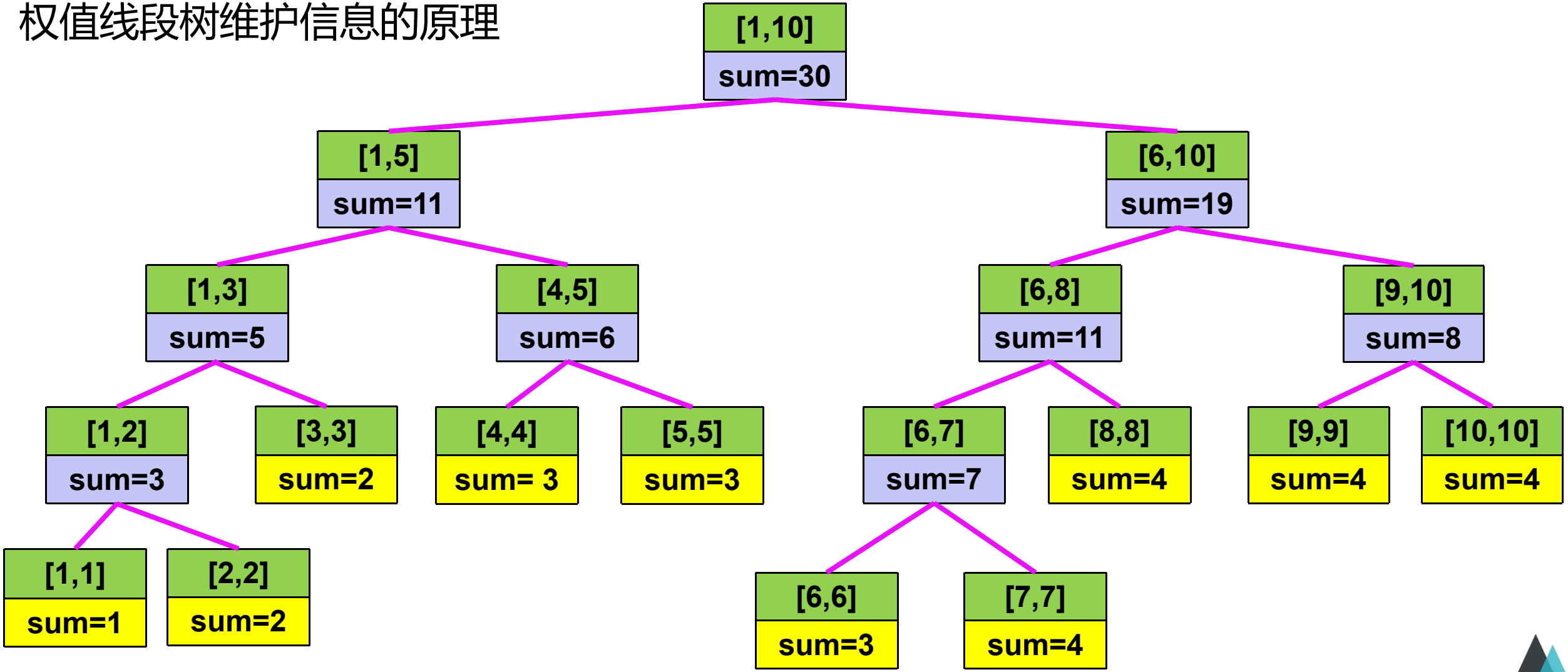
- 权值线段是一棵已经建好的线段树，储存树结构的**下标范围根据区间最大值最小值**确定(一般开大空间即可)；
- 初始状态下所有的点权均为0；
- 执行插入元素的操作时，会导致插入值val对应的A[val]++，同时引发单点更新操作；
- 可以从上述描述中得到基本推论：向空的权值线段树中插入一个无序序列，则插入完成后在树上无法再得到原序列的遍历，但能通过遍历得到有序序列(无序序列变有序序列)

权值线段树

权值线段树的基本原理与操作

权值线段树维护信息的原理

我们给定序列{ 1 , 2 , 2 , 3 , 3 , 3 , 4 , 4 , 4 , 4 }，以该序列为例建立两种线段树。首先，序列长度为10，分别建立基础线段树和权值线段树，**基础线段树**结构图如下：

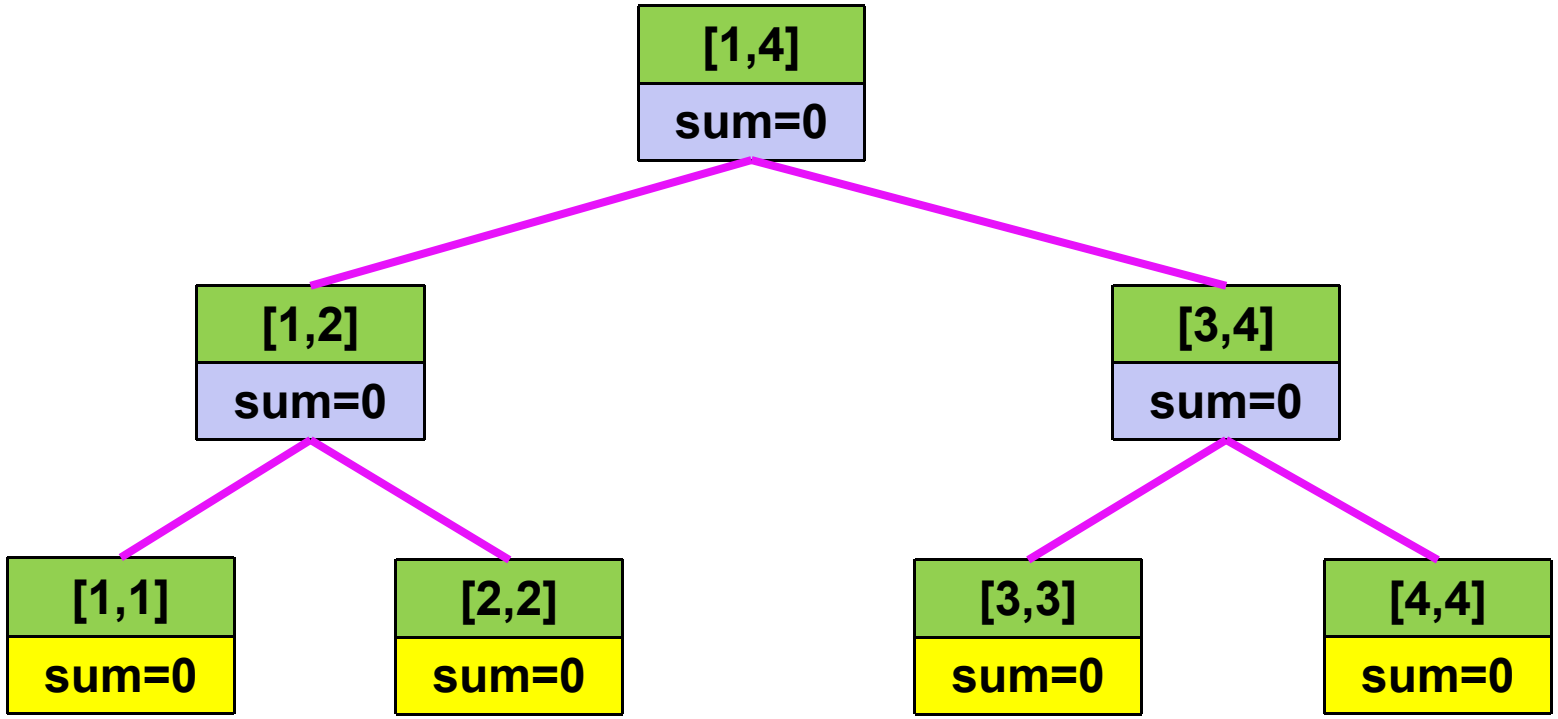


权值线段树

权值线段树的基本原理与操作

权值线段树维护信息的原理

我们给定序列{ 1 , 2 , 2 , 3 , 3 , 3 , 4 , 4 , 4 , 4 }，以该序列为例建立两种线段树。首先，序列长度为10，分别建立基础线段树和权值线段树，权值**线段树的空树**结构图如下：



权值线段树

权值线段树的基本原理与操作

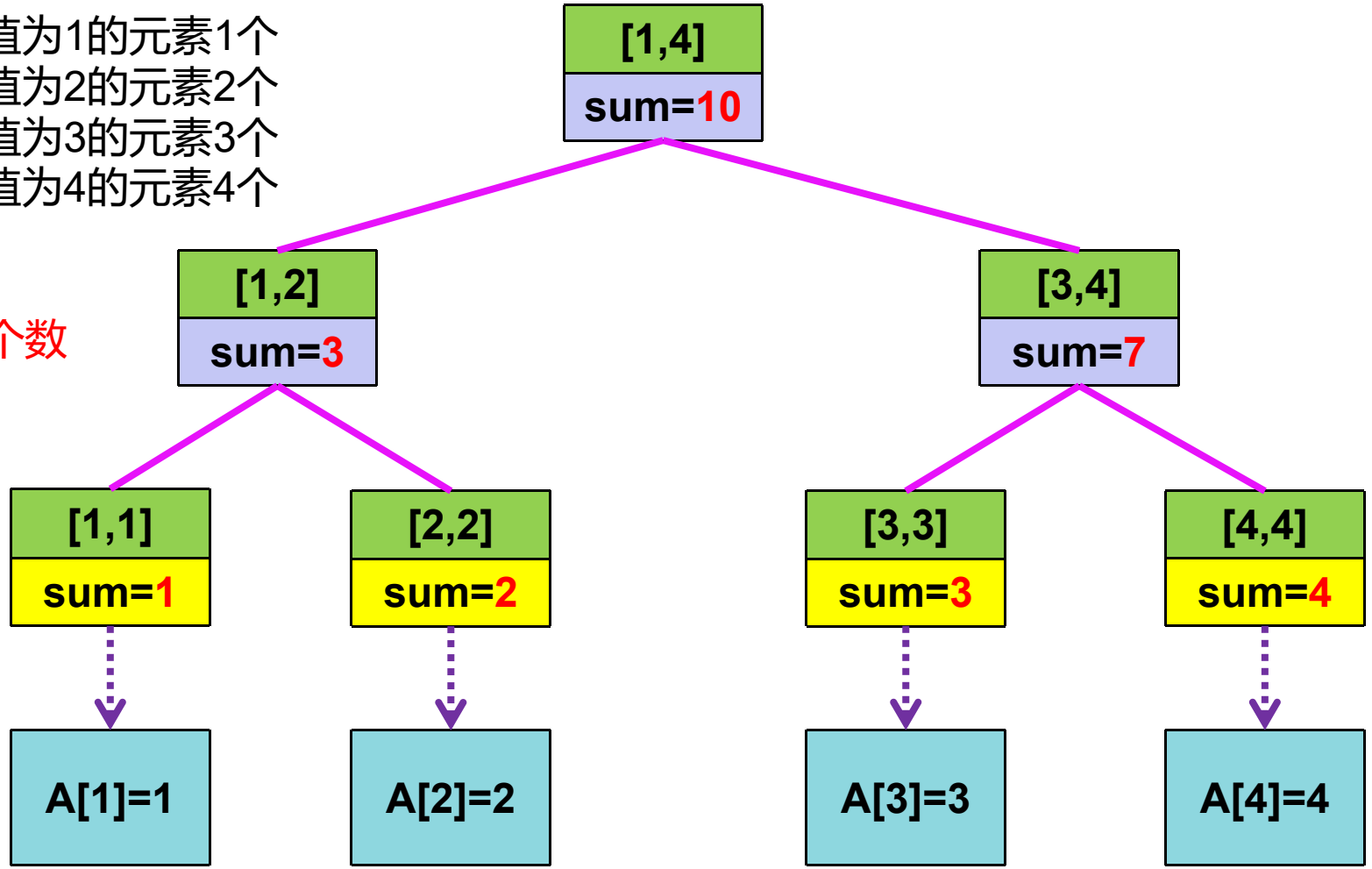
权值线段树维护信息的原理

含义:

A[1]=1：原序列中存在值为1的元素1个
A[2]=2：原序列中存在值为2的元素2个
A[3]=3：原序列中存在值为3的元素3个
A[4]=4：原序列中存在值为4的元素4个

权值线段树维护的是
一定区间内所含元素的个数

我们给定序列{ 1 , 2 , 2 , 3 , 3 , 3 , 4 , 4 , 4 , 4 }，以该序列为例建立两种线段树。首先，序列长度为10，分别建立基础线段树和权值线段树，然后按照序列顺序插入元素后，权值**线段树**结构图如下：



权值线段树

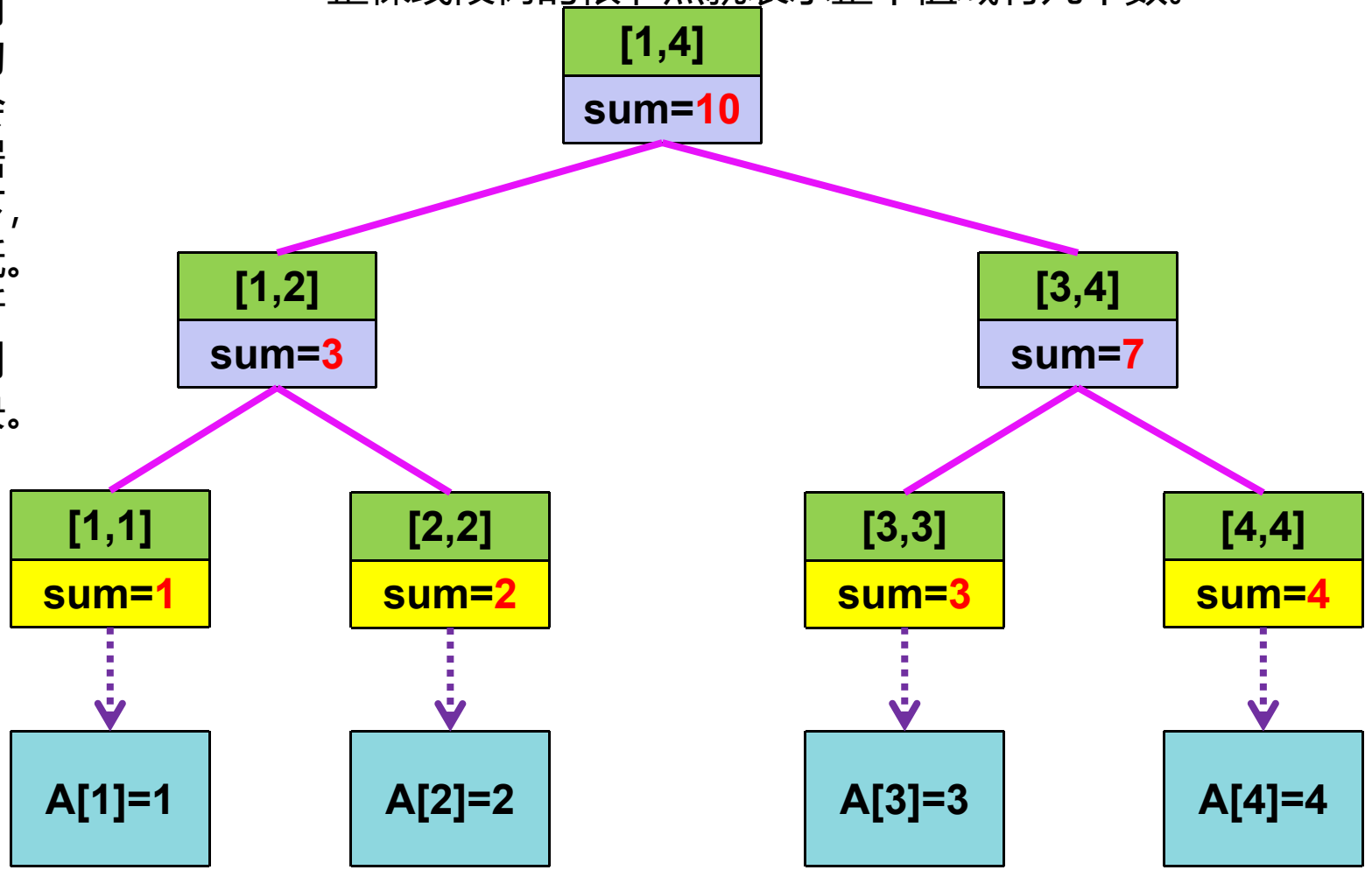
权值线段树的基本原理与操作

权值线段树维护信息的原理

在权值线段树中，采用**元素到下标数组**映射的方式进行插入。这样会导致一个问题：在数据离散程度较大的情况下，空间的利用效率比较低。因此我们对于线段树所开空间不足时，常采用**离散化**的方式进行解决。

由于权值线段树每个节点记录了某段区间内包含的元素个数，且元素在叶子节点构成的序列中是有序的：

- 对于整列数中第K大/小的值，我们从根节点开始判断（这里按第K 小为例），如果**K比左儿子大，就说明第K小的数在右儿子所表示的值域中**。然后，K要减去左儿子。（这是因为继续递归下去的时候，正确答案，整个区间的第K小已经不再是左儿子表示区间的第K小了）。如此递归到叶子节点时即为答案。
- 整棵线段树的根节点就表示整个值域有几个数。



权值线段树

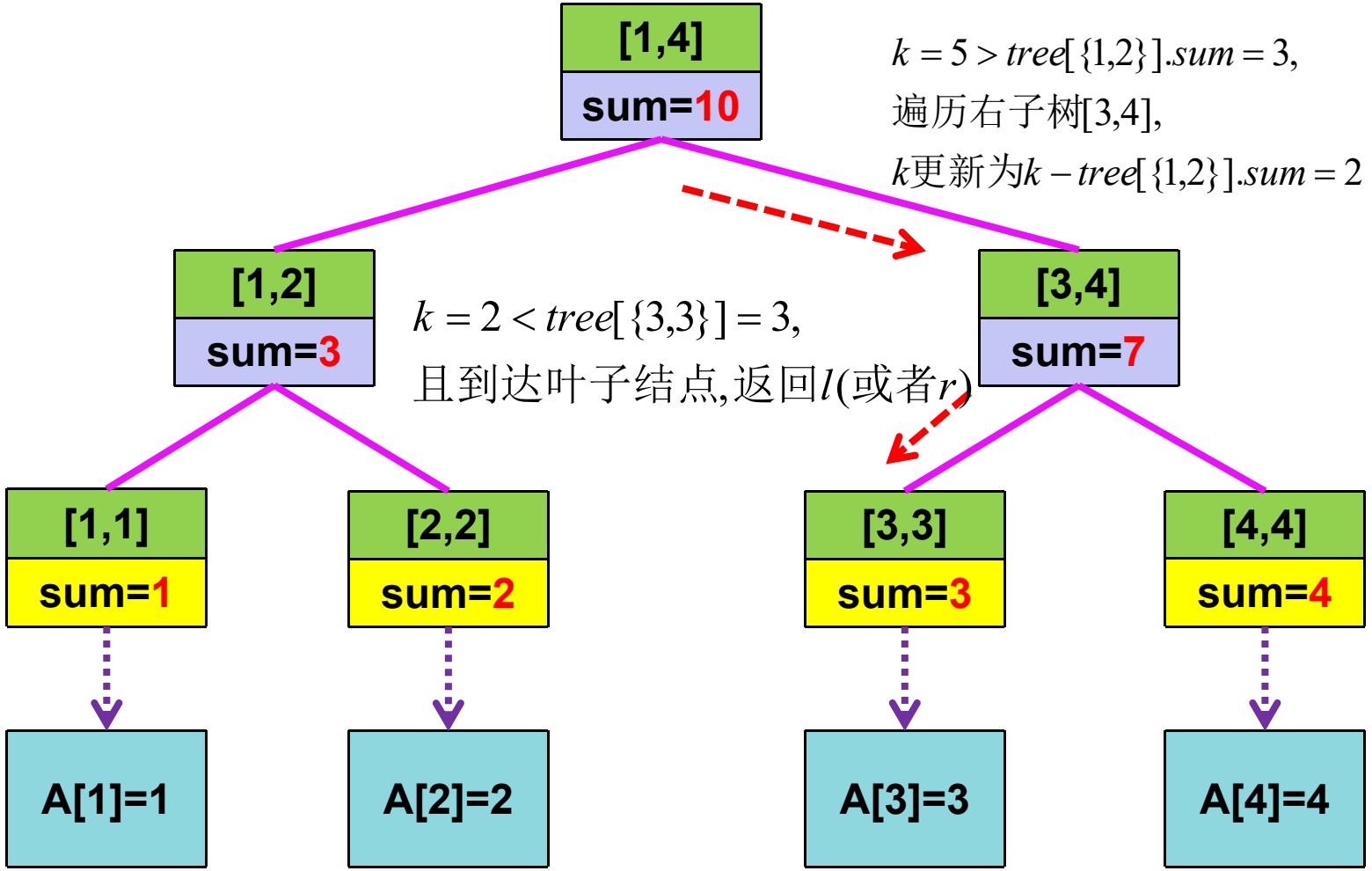
权值线段树的基本操作与实现

①.查询第K大/小元素

假设K = 5，那么查询过程是怎样的?

首先我们从根节点出发，由于要查询第K小，因此从左子结点开始判断:

发现根结点[1,4]左子树[1,2]的tree[2*root].sum<k,查询[1,4]的右子树[3,4]，此时查询[1,4]右子树的第k-tree[2*root].sum小，即[1,4]右子树[3,4]的第2小的，此时[3,4]的左子树[3,3]的tree[2*root'].sum>k',且到达叶子结点，返回当前左子树的l(或r)即可



权值线段树

权值线段树的基本操作与实现

①.查询第K大/小元素

```
int k_th(int root,int k){//查询第k小的数
    if(tree[root].l==tree[root].r)//到达叶子结点
        return tree[root].l;
    //int mid=(tree[root].l+tree[root].r)/2;
    if(k<=tree[2*root].dat)//当前k小于左子树的dat,遍历左子树
        return k_th(2*root,k);
    else return k_th(2*root+1,k-tree[2*root].dat);//遍历右子树,k更新为k-左子树的dat
}

int k_th(int root,int k){//查询第k大的数
    if(tree[root].l==tree[root].r)//到达叶子结点
        return tree[root].l;
    //int mid=(tree[root].l+tree[root].r)/2;
    if(k<=tree[2*root+1].dat)//当前k小于右子树的dat,遍历右子树
        return k_th(2*root+1,k);
    else return k_th(2*root,k-tree[2*root+1].dat);//遍历左子树,k更新为k-右子树的dat
}
```

权值线段树

权值线段树的基本操作与实现

②.单点更新操作

类似基础线段树(其实一样)，递归到叶子节点+ val然后回溯

```
void modify(int root,int pos,int val){
    if(tree[root].l==pos&&tree[root].r==pos){
        tree[root].dat+=val;
        return ;
    }
    int mid=(tree[root].l+tree[root].r)/2;
    if(pos<=mid)
        modify(2*root,pos,val);
    else modify(2*root+1,pos,val);
    pushup(root);
}
```


权值线段树

权值线段树的基本操作与实现

③.查询某个数出现的个数 类似基础线段树(其实一样)，单点查询

```
int query_num(int root,int pos){
    if(tree[root].l==pos&&tree[root].r==pos)
        return tree[root].dat;
    int mid=(tree[root].l+tree[root].r)/2;
    if(pos<=mid)
        return query_num(2*root,pos);
    else return query_num(2*root+1,pos);
}
```

权值线段树

权值线段树的基本操作与实现

④.查询一段区间的数出现的次数　　类似基础线段树(其实一样)，区间查询

```
int query(int root,int l,int r){
    if(l<=tree[root].l&&r>=tree[root].r)
        return tree[root].dat;
    int mid=(tree[root].l+tree[root].r)/2;
    int res=0;
    if(l<=mid)
        res+=query(2*root,l,r);
    if(r>mid)
        res+=query(2*root+1,l,r);
    return res;
}
```

权值线段树

权值线段树的基本操作与实现

④.查询一段区间的数出现的次数

```
int query(int root,int l,int r){
    if(l<=tree[root].l&&r>=tree[root].r)
        return tree[root].dat;
    int mid=(tree[root].l+tree[root].r)/2;
    int res=0;
    if(l<=mid)
        res+=query(2*root,l,r);
    if(r>mid)
        res+=query(2*root+1,l,r);
    return res;
}
```

⑤.查询当前元素的排名

可看作查询[1,上一个元素]的个数+1

⑥.查询当前元素的前驱(离散化可看作是前驱的下标)

可看作查询[1,上一个元素]的个数k的求第k小

⑦查询当前元素的后继(离散化可看作是后继的下标)

可看作查询[1,当前元素]的个数加一的k的求第k小

```
int query_rank(int pos){
    return query(1,1,pos-1)+1;
}

int query_pre(int pos){
    return k_th(1,query(1,1,pos-1));
}

int query_nex(int pos){
    return k_th(1,query(1,1,pos)+1);
}
```

权值线段树

权值线段树的基本操作与实现

具体例题

普通平衡树

此题看似是平衡树的题(可以用平衡树解决)，但依然可以用**权值线段树**进行维护，注意的是此题需要进行**离散化**操作(数的范围较大，且存在负数)

您需要写一种数据结构（可参考题目标题），来维护一些数，其中需要提供以下操作：

1. 插入数值 x。
2. 删除数值 x(若有多个相同的数，应只删除一个)。
3. 查询数值 x 的排名(若有多个相同的数，应输出最小的排名)。
4. 查询排名为 x 的数值。
5. 求数值 x 的前驱(前驱定义为小于 x 的最大的数)。
6. 求数值 x 的后继(后继定义为大于 x 的最小的数)。

注意： 数据保证查询的结果一定存在。

输入格式

第一行为 n，表示操作的个数。

接下来 n 行每行有两个数 opt 和 x，opt 表示操作的序号($1 \leq opt \leq 6$)。

输出格式

对于操作 $3, 4, 5, 6$ 每行输出一个数，表示对应答案。

数据范围

$1 \leq n \leq 100000$,所有数均在 -10^7 到 10^7 内。

权值线段树

权值线段树的基本操作与实现

具体例题

普通平衡树

此题看似是平衡树的题(可以用平衡树解决)，但依然可以用**权值线段树**进行维护，注意的是此题需要进行**离散化**操作(数的范围较大，且存在负数)

将[10,20,30,20,25,-1]进行离散化b数组为离散化后的数组

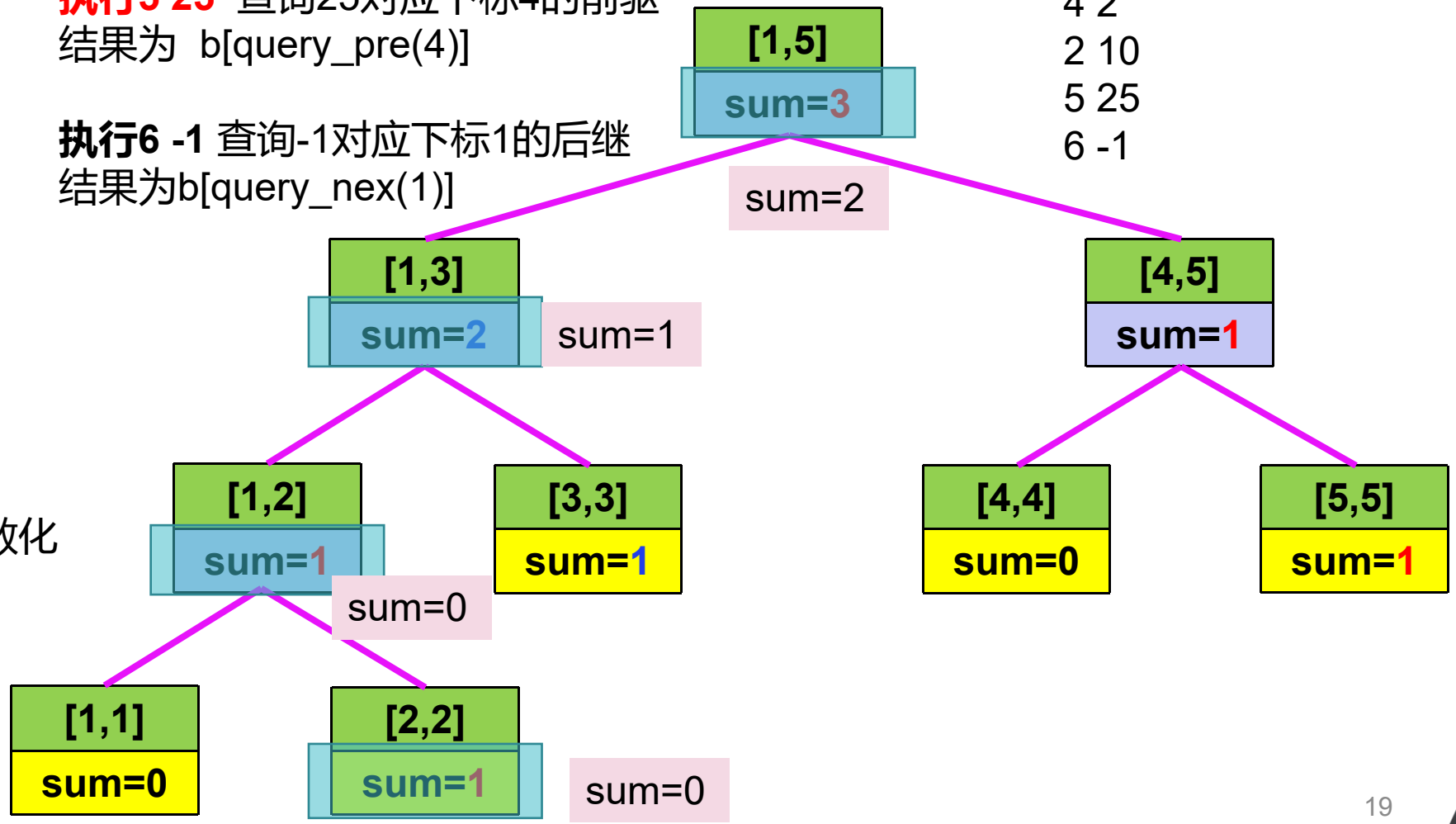
b[] -1 10 20 25 30
id[] 1 2 3 4 5

执行1 10 单点修改2 +1
执行1 20 单点修改3 +1
执行1 30 单点修改5 +1
执行3 20 查询20对应下标3的排名 结果为2
执行4 2 查询排名为2的元素 结果为b[k_th(2)]=20
执行 2 10 单点修改2 -1
执行5 25 查询25对应下标4的前驱
结果为 b[query_pre(4)]

执行6 -1 查询-1对应下标1的后继
结果为b[query_nex(1)]

输入样例:
8
1 10
1 20
1 30
3 20
4 2
2 10
5 25
6 -1

输出样例:
2
20
20
20



权值线段树

权值线段树的基本操作与实现

具体例题

普通平衡树

此题看似是平衡树的题(可以用平衡树解决)，但依然可以用**权值线段树**进行维护，注意的是此题需要进行**离散化**操作(数的范围较大，且存在负数)

实际上和基础线段树的核心代码基本一模一样

```
typedef long long ll;

int n,tot=0;
int op[MAXN],id[MAXN];
ll a[MAXN],b[MAXN];

struct Tree{
    int l;
    int r;
    int dat;
}tree[4*MAXN];

void pushup(int root){
    tree[root].dat=tree[2*root].dat+tree[2*root+1].dat;
}
```

权值线段树

权值线段树的基本操作与实现

具体例题

普通平衡树

此题看似是平衡树的题(可以用平衡树解决)，但依然可以用**权值线段树**进行维护，注意的是此题需要进行**离散化**操作(数的范围较大，且存在负数)

实际上和基础线段树的核心代码基本一模一样

建空树操作

```
void build(int root,int l,int r){
    tree[root].l=l,tree[root].r=r;
    if(l==r){
        tree[root].dat=0;
        return ;
    }
    int mid=(l+r)/2;
    build(2*root,l,mid);
    build(2*root+1,mid+1,r);
    pushup(root);
}
```

权值线段树

权值线段树的基本操作与实现

具体例题

普通平衡树

此题看似是平衡树的题(可以用平衡树解决)，但依然可以用**权值线段树**进行维护，注意的是此题需要进行**离散化**操作(数的范围较大，且存在负数)

实际上和基础线段树的核心代码基本一模一样

单点修改操作

```
void modify(int root,int pos,int val){
    if(tree[root].l==pos&&tree[root].r==pos){
        tree[root].dat+=val;
        return ;
    }
    int mid=(tree[root].l+tree[root].r)/2;
    if(pos<=mid)
        modify(2*root,pos,val);
    else modify(2*root+1,pos,val);
    pushup(root);

}
```

权值线段树

权值线段树的基本操作与实现

具体例题

普通平衡树

此题看似是平衡树的题(可以用平衡树解决)，但依然可以用**权值线段树**进行维护，注意的是此题需要进行**离散化**操作(数的范围较大，且存在负数)

实际上和基础线段树的核心代码基本一模一样

区间查询操作

```
int query(int root,int l,int r){
    if(l<=tree[root].l&&r>=tree[root].r)
        return tree[root].dat;
    int mid=(tree[root].l+tree[root].r)/2;
    int res=0;
    if(l<=mid)
        res+=query(2*root,l,r);
    if(r>mid)
        res+=query(2*root+1,l,r);
    return res;
}
```

权值线段树

权值线段树的基本操作与实现

具体例题

普通平衡树

此题看似是平衡树的题(可以用平衡树解决)，但依然可以用**权值线段树**进行维护，注意的是此题需要进行**离散化**操作(数的范围较大，且存在负数)

实际上和基础线段树的核心代码基本一模一样

查询第K小的元素操作

```
int k_th(int root,int k){
    if(tree[root].l==tree[root].r)
        return tree[root].l;
    //int mid=(tree[root].l+tree[root].r)/2;
    if(k<=tree[2*root].dat)
        return k_th(2*root,k);
    else return k_th(2*root+1,k-tree[2*root].dat);
}
```

查询当前元素的排名操作

```
int query_rank(int pos){
    return query(1,1,pos-1)+1;
}
```

权值线段树

权值线段树的基本操作与实现

具体例题

普通平衡树

此题看似是平衡树的题(可以用平衡树解决)，但依然可以用**权值线段树**进行维护，注意的是此题需要进行**离散化**操作(数的范围较大，且存在负数)

实际上和基础线段树的核心代码基本一模一样

查询当前元素的前驱元素操作

```
int query_pre(int pos){
    return k_th(1,query(1,1,pos-1));
}
```

查询当前元素的后继元素操作

```
int query_nex(int pos){
    return k_th(1,query(1,1,pos)+1);
}
```

权值线段树

权值线段树的基本操作与实现

具体例题

普通平衡树

此题看似是平衡树的题(可以用平衡树解决)，但依然可以用**权值线段树**进行维护，注意的是此题需要进行**离散化**操作(数的范围较大，且存在负数)

实际上和基础线段树的核心代码基本一模一样

主函数（离散化操作,此处实际上需要进行**离线**操作）

```
scanf("%d",&n);
for(int i=1;i<=n;i++){
    scanf("%d %lld",&op[i],&a[i]);
    if(op[i]!=4)
        b[++tot]=a[i];
}

sort(b+1,b+tot+1);
tot=unique(b+1,b+tot+1)-(b+1);

build(1,1,tot);

for(int i=1;i<=n;i++){
    if(op[i]!=4)
        id[i]=lower_bound(b+1,b+tot+1,a[i])-b;
}
```

权值线段树

完整代码链接

权值线段树的基本操作与实现

具体例题

普通平衡树

此题看似是平衡树的题(可以用平衡树解决)，但依然可以用**权值线段树**进行维护，注意的是此题需要进行**离散化**操作(数的范围较大，且存在负数)

实际上和基础线段树的核心代码基本一模一样

主函数（各种操作,大部分情况使用**离散化所对应的下标**）

```
for(int i=1;i<=n;i++)
{
    if(op[i]==1)
        modify(1,id[i],1);
    else if(op[i]==2)
        modify(1,id[i],-1);
    else if(op[i]==3)
        printf("%d\n",query_rank(id[i]));
    else if(op[i]==4)
        printf("%lld\n",b[k_th(1,a[i])]);
    else if(op[i]==5)
        printf("%lld\n",b[query_pre(id[i])]);
    else if(op[i]==6)
        printf("%lld\n",b[query_nex(id[i])]);
}
```

线段树维护复杂区间

- 线段树维护连续子段和
- 线段树同时维护乘法和加法
- 线段树同时维护开方和加法

线段树维护复杂区间

线段树维护最长连续子段和

你能回答这些问题吗

给定长度为 N 的数列 A，以及 M 条指令 $(N \leq 5 * 10^5, M \leq 10^5)$，每条指令可能是以下两种之一：

1. “1 x y”，查询区间 $[x, y]$ 中的最大连续子段和，即 $\max_{x \leq l \leq r \leq y} \{\sum_{i=l}^{r} A[i]\}$。

2. “2 x y”，把 $A[x]$ 改成 y。

对于每个询问，输出一个整数表示答案。

线段树维护复杂区间

线段树维护最长连续子段和

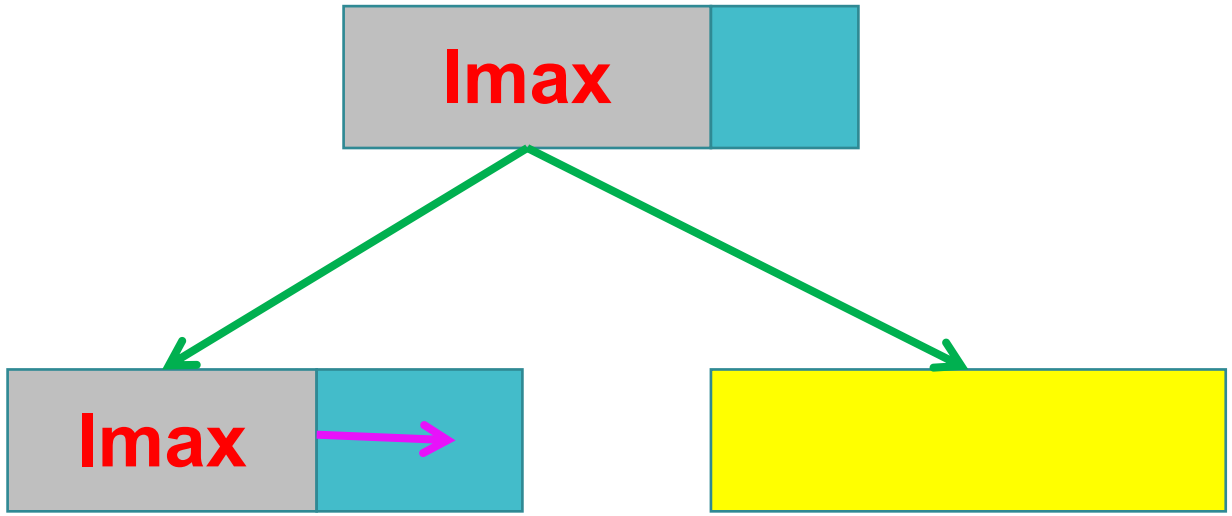
你能回答这些问题吗

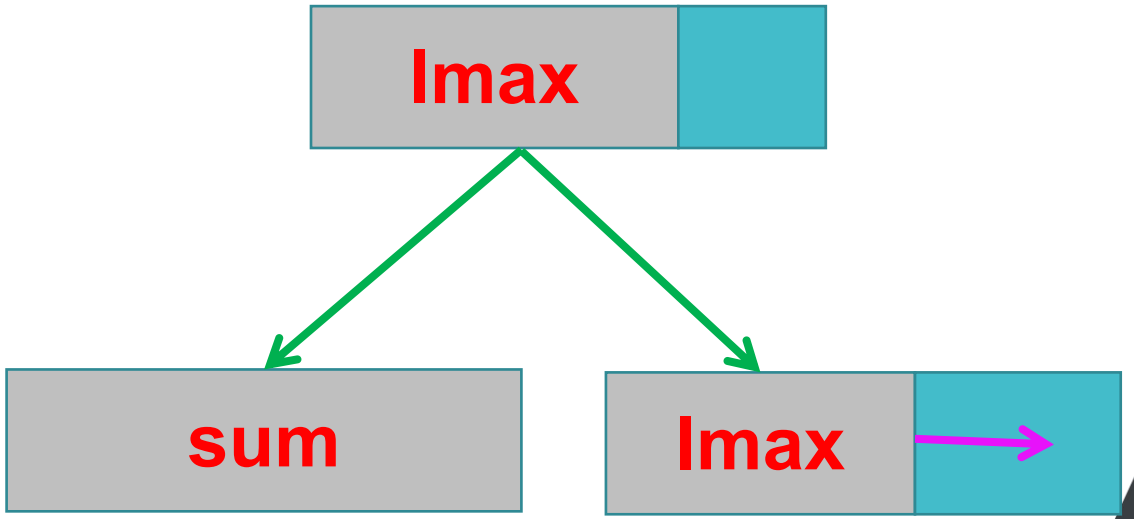
✓1. 维护一个tree[root].sum信息还是比较容易的

✓2. 考虑维护tree[root].lmax

表示在root这个结点，从root所表示的区间最左边开始扩展连续子段，所得到的最大子段和

$$tree[root].l\max = \max(tree[2\times root].l\max, tree[2\times root].sum + tree[2\times root+1].l\max);$$





线段树维护复杂区间

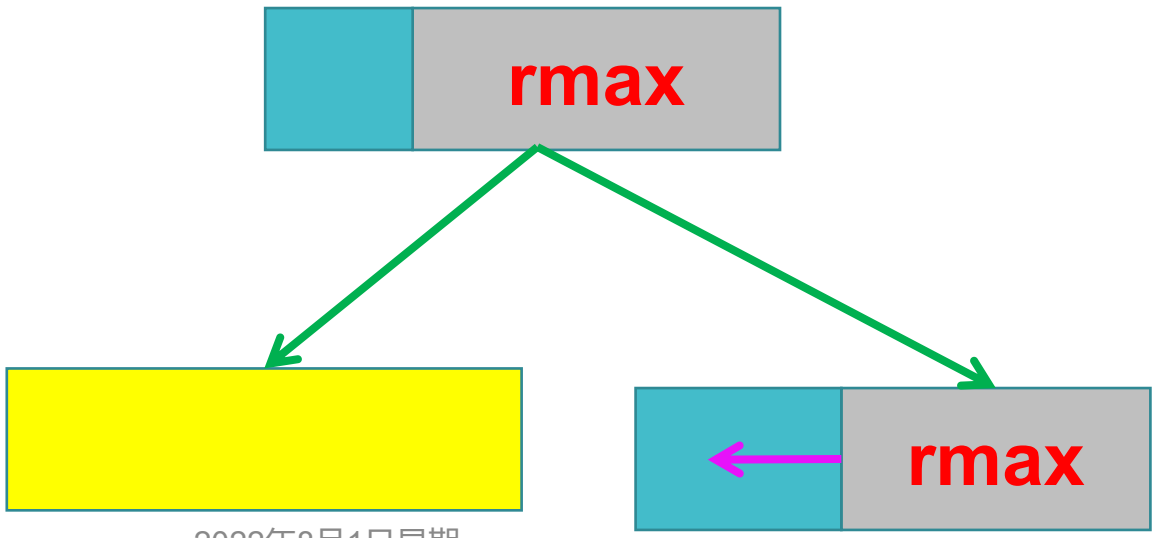
线段树维护最长连续子段和

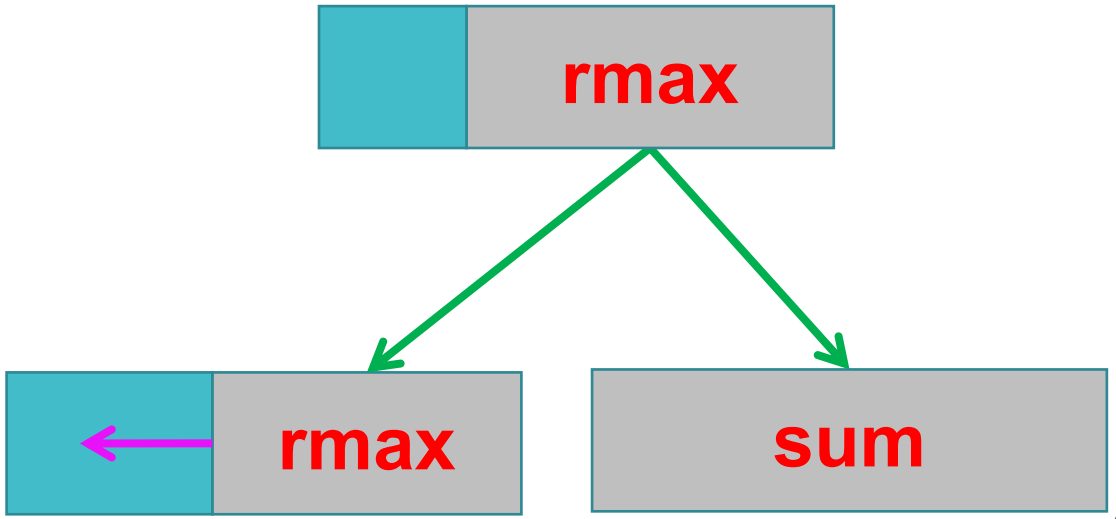
你能回答这些问题吗

✓3. 考虑维护tree[root].rmax

表示在root这个结点，从root所表示的区间最右边开始扩展连续子段，所得到的最大子段和

$$tree[root].r\max = \max(tree[2\times root+1].r\max, tree[2\times root+1].sum + tree[2\times root].r\max);$$





线段树维护复杂区间

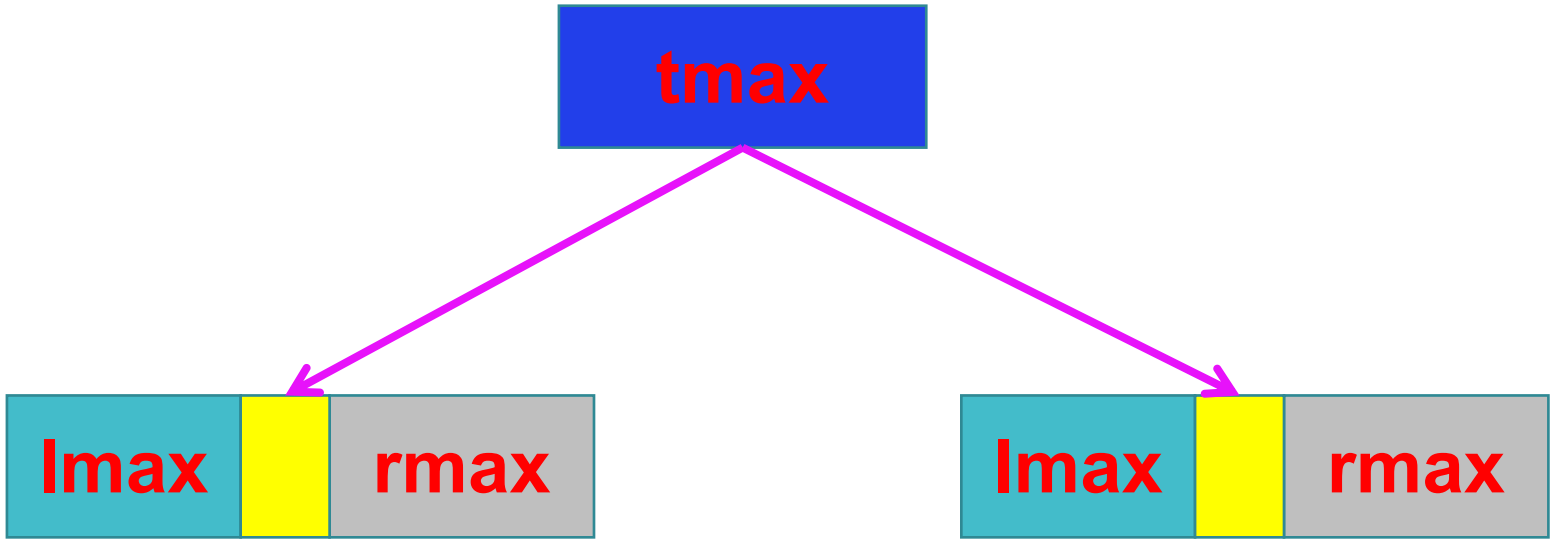
线段树维护最长连续子段和

你能回答这些问题吗

✓4. 考虑维护tree[root].tmax

表示在root这个结点，所有情况所得到的最大子段和

$$tree[root].t\max = \max(\{tree[2\times root].l\max, tree[2\times root+1].r\max,$$
$$tree[2\times root].r\max + tree[2\times root+1].l\max\})$$



线段树维护复杂区间

线段树维护最长连续子段和

你能回答这些问题吗

✓1. 维护一个tree[root].sum信息还是比较容易的
✓2. 考虑维护tree[root].lmax
✓3. 考虑维护tree[root].rmax
✓4. 考虑维护tree[root].tmax

只需要重写pushup函数即可

线段树维护复杂区间

线段树维护最长连续子段和

你能回答这些问题吗

注：此处pushup函数用了函数重载和引用，方便区间查询时使用

```
void pushup(Tree &root,Tree &l,Tree &r){
    root.sum=l.sum+r.sum;
    root.lmax=max(l.lmax,l.sum+r.lmax);
    root.rmax=max(r.rmax,r.sum+l.rmax);
    root.tmax=max({l.tmax,r.tmax,l.rmax+r.lmax});
}


void pushup(int root){
    pushup(tree[root],tree[2*root],tree[2*root+1]);
}
```


线段树维护复杂区间

线段树维护最长连续子段和

你能回答这些问题吗

查询区间[l,r]的最大连续子段和

- 若[l,r]完全覆盖了当前结点代表的区间，则立即回溯，并且该结点为候选答案
- 若r<=当前结点代表区间的mid时，即区间[l,r]只在左区间，只递归访问左子结点；
- 若l> 当前结点代表区间的mid时，即区间[l,r]只在右区间，只递归访问右子结点；
- 若区间[l,r]横跨左右子结点,都要进行递归。

线段树维护复杂区间

完整

线段树维护最长连续子段和

你能回答这些问题吗

区间查询

其他操作大体与基础线段树类似，此处就不展示**建树和单点修改**的操作了

```
Tree query(int root,int l,int r){
    if(l<=tree[root].l&&r>=tree[root].r)
        return tree[root];
    else{
        int mid=(tree[root].l+tree[root].r)/2;
        if(r<=mid)
            return query(2*root,l,r);
        else if(l>mid)
            return query(2*root+1,l,r);
        else{
            Tree left=query(2*root,l,r);
            Tree right=query(2*root+1,l,r);
            Tree res;
            pushup(res,left,right);
            return res;
        }
    }
}

printf("%d\n",query(1,l,r).tmax);//主函数
```

线段树维护复杂区间

线段树同时维护乘法和加法

维护序列

题目描述

有一个长为 n 的数列 $\{a_n\}$，有如下三种操作形式：

1. 格式 `1 t g c`，表示把所有满足 $t \le i \le g$ 的 a_i 改为 $a_i \times c$；
2. 格式 `2 t g c` 表示把所有满足 $t \le i \le g$ 的 a_i 改为 $a_i + c$；
3. 格式 `3 t g` 询问所有满足 $t \le i \le g$ 的 a_i 的和，由于答案可能很大，你只需输出这个数模 p 的值。

输入格式

第一行两个整数 n 和 p。

第二行含有 n 个非负整数，表示数列 $\{a_i\}$。

第三行有一个整数 m，表示操作总数。

从第四行开始每行描述一个操作，同一行相邻两数之间用一个空格隔开，每行开头和末尾没有多余空格。

线段树维护复杂区间

线段树同时维护乘法和加法

维护序列

1. 格式 `1 t g c`，表示把所有满足 $t \le i \le g$ 的 a_i 改为 $a_i \times c$；
2. 格式 `2 t g c` 表示把所有满足 $t \le i \le g$ 的 a_i 改为 $a_i + c$；
3. 格式 `3 t g` 询问所有满足 $t \le i \le g$ 的 a_i 的和，由于答案可能很大，你只需输出这个数模 p 的值。

输入 #1 复制

```
7 43
1 2 3 4 5 6 7
5
1 2 5 5
3 2 4
2 3 7 9
3 1 3
3 4 7
```

输出 #1 复制

```
2
35
8
```

线段树维护复杂区间

线段树同时维护乘法和加法

维护序列

1. 格式 `1 t g c`，表示把所有满足 $t \le i \le g$ 的 a_i 改为 $a_i \times c$；
2. 格式 `2 t g c` 表示把所有满足 $t \le i \le g$ 的 a_i 改为 $a_i + c$；
3. 格式 `3 t g` 询问所有满足 $t \le i \le g$ 的 a_i 的和，由于答案可能很大，你只需输出这个数模 p 的值。

- 初始时数列为 $\{1, 2, 3, 4, 5, 6, 7\}$。
- 经过第 1 次操作后，数列为 $\{1, 10, 15, 20, 25, 6, 7\}$。
- 对第 2 次操作，和为 $10 + 15 + 20 = 45$，模 43 的结果是 2。
- 经过第 3 次操作后，数列为 $\{1, 10, 24, 29, 34, 15, 16\}$。
- 对第 4 次操作，和为 $1 + 10 + 24 = 35$，模 43 的结果是 35。
- 对第 5 次操作，和为 $29 + 34 + 15 + 16 = 94$，模 43 的结果是8。

执行的操作:

```
1 2 5 5
3 2 4
2 3 7 9
3 1 3
3 4 7
```

线段树维护复杂区间

维护序列

线段树同时维护乘法和加法

- ✓ 1. 维护一个tree[root].sum区间和
- ✓ 2. 考虑维护tree[root].add的加法懒标记

- ✓ 3. 考虑维护tree[root].mul的乘法懒标记

一般懒标记的问题就是处理**pushdown函数**的问题

线段树维护复杂区间

线段树同时维护乘法和加法

维护序列

- ✓ 考虑维护tree[root].add的加法懒标记
- ✓ 考虑维护tree[root].mul的乘法懒标记

对于原数x

1. 若对懒标记的处理是**先加后乘**，即(x+add)*mul

- ➢ 若此次操作为乘上一个数c，可以表示为(x+add)*mul*c,即(x+add)*mul'的形式
- ➢ 若此次操作为加上一个数c，可以表示为(x+add)*mul+c,而此时不能写成(x+add)*mul'的形式，即**无法更新新的懒标记**

2. 若对懒标记的处理是**先乘后加**，即x*mul+add

线段树维护复杂区间

线段树同时维护乘法和加法

维护序列

✓ 考虑维护tree[root].add的加法懒标记

✓ 考虑维护tree[root].mul的乘法懒标记

对于原数x

2. 若对懒标记的处理是**先乘后加**，即x*mul+add

➢ 若此次操作是加上一个数c，可以表示为x*mul+add+c，此时新的**add即为add+c**

➢ 若此时操作是乘上一个数c，可以表示为(x*mul+add)*c，展开后为x*mul*c+add*c，此时**新的add即为add*c，新的mul即为mul*c**

故采用先乘后加的处理手段，以便更新懒标记

线段树维护复杂区间

代码链接

线段树同时维护乘法和加法

维护序列

✓ 考虑维护tree[root].add的加法懒标记

✓ 考虑维护tree[root].mul的乘法懒标记

3. 可以将乘和加的操作都看成x*c+d的形式

➢ 若是乘法，d为0
➢ 若是加法，c为1

4. 若当前x的懒标记为add和mul

◆ 操作可以写成(x*mul+add)*c+d
◆ 即x*(mul*c)+(add*c+d)
◆ 新的mul为**mul*c**，新的add为**(add*c+d)**

注意: **乘的懒标记初始为1**

```
void eval(Tree &t,int add,int mul){
    t.sum=((ll)t.sum*mul+(ll)(t.r-t.l+1)*add)%p;
    t.mul=(ll)t.mul*mul%p;
    t.add=((ll)t.add*mul+add)%p;
}

void pushdown(int root){
    eval(tree[2*root],tree[root].add,tree[root].mul);
    eval(tree[2*root+1],tree[root].add,tree[root].mul);
    tree[root].add=0,tree[root].mul=1;
}
```

线段树维护复杂区间

线段树同时维护开方和加法

花神游历各国

花神喜欢步行游历各国，顺便虐爆各地竞赛花神有一条游览路线，它是线型的，也就是说，所有游历国家呈一条线的形状排列，花神对每个国家都有一个喜欢程度（当然花神并不一定喜欢所有国家）每一次旅行中，花神会选择一条旅游路线，它在那一串国家中是连续的一段，这次旅行带来的开心值是这些国家的喜欢度的总和当然花神对这些国家的喜欢程度并不是恒定的，有时会突然对某些国家产生反感，使他对这些国家的喜欢度 delta 变为$\sqrt{delta}$，也就是开根号（可能是花神虐爆了那些国家的 OI，从而感到乏味）现在给出花神每次的旅行路线，以及开心度的变化，请求出花神每次旅行的开心值

线段树维护复杂区间

线段树同时维护开方和加法

花神游历各国

第一行是一个整数 N 表示有 N 个国家

第二行有 N 个空格隔开的整数，表示每个国家的初始喜欢度 data[i]

第三行是一个整数 M 表示有 M 条信息要处理

第四行到最后，每行 3 个整数，x，l，r，（l<=r）当 x=1 时询问游历国家 l 到 r 的开心值总和，也就是 $\sum_{l}^{r} data[i]$，当 x=2 是国家 l 到 r 中每个国家的喜欢度 $delta = \sqrt{delta}$

注：建议使用 sqrt 函数，且向下取整

线段树维护复杂区间

线段树同时维护开方和加法

花神游历各国

我们试图用**懒标记**进行维护区间的开方，发现区间之间的开方并无相互联系，故无法使用懒标记进行维护。

但是，我们可以根据开方运算的性质得到，对于1和0无论如何开方都为它本身，并且数据的最大值为10^9，所以最多进行**5次开方**的操作10^9就会等于1。

根据这一条件每个数值最多只需要修改5次，而区间长度最大为10^5所以修改操作最多执行$5*\ 10^5$次。
可以用线段树维护区间和，单次查询的时间复杂度为O(logn) ,查询最差时间复杂度为O(mlogn);再**维护一个值flag表示该区间是否需要进行开方的操作**（当区间中的数值都为1和0时，该区间就不需要进行修改），这样修改最差的时间复杂度为O(nlogn)。

线段树维护复杂区间

线段树同时维护开方和加法

花神游历各国

```
void pushup(int root){
    tree[root].sum=tree[2*root].sum+tree[2*root+1].sum;
    tree[root].flag=tree[2*root].flag&&tree[2*root+1].flag;//当flag=1时不修改
}

void build(int root,int l,int r){
    tree[root].l=l,tree[root].r=r;
    if(l==r){
        tree[root].sum=a[l];
        tree[root].flag=(a[l]<=1);
        return ;
    }
    int mid=(l+r)/2;
    build(2*root,l,mid);
    build(2*root+1,mid+1,r);
    pushup(root);
}
```


线段树维护复杂区间

线段树同时维护开方和加法

花神游历各国

```
void modify(int root,int l,int r){
    if(tree[root].flag==1)//当前区间flag=1可不用进行开方
        return ;
    if(tree[root].l==tree[root].r){//此处是将区间修改转换成单点修改
        tree[root].sum=sqrt(tree[root].sum);
        tree[root].flag=(tree[root].sum<=1);
    }
    else{
        int mid=(tree[root].l+tree[root].r)/2;
        if(r<=mid)
            modify(2*root,l,r);
        else if(l>mid)
            modify(2*root+1,l,r);
        else{
            modify(2*root,l,mid);
            modify(2*root+1,mid+1,r);
        }
        pushup(root);
    }
}
```

扩展：逆序对

可试着用**归并排序和树状数组及线段树**解决第一周组队赛的C题A Sorting Problem

对于一个序列a,若$i<j$且$a[i]>a[j]$,则称$a[i]$与$a[j]$构成逆序对。

使用归并排序可以在 O(nlogn)的时间里求出一个长度为n的序列中逆序对的个数。

归并流程:

- 把长度为n的输入序列分成两个长度为 n/2 的子序列;
- 对这两个子序列分别采用归并排序;
- 将两个排序好的子序列合并成一个最终的排序序列。

扩展：逆序对

对于一个序列a,若$i<j$且$a[i]>a[j]$,则称$a[i]$与$a[j]$构成逆序对。

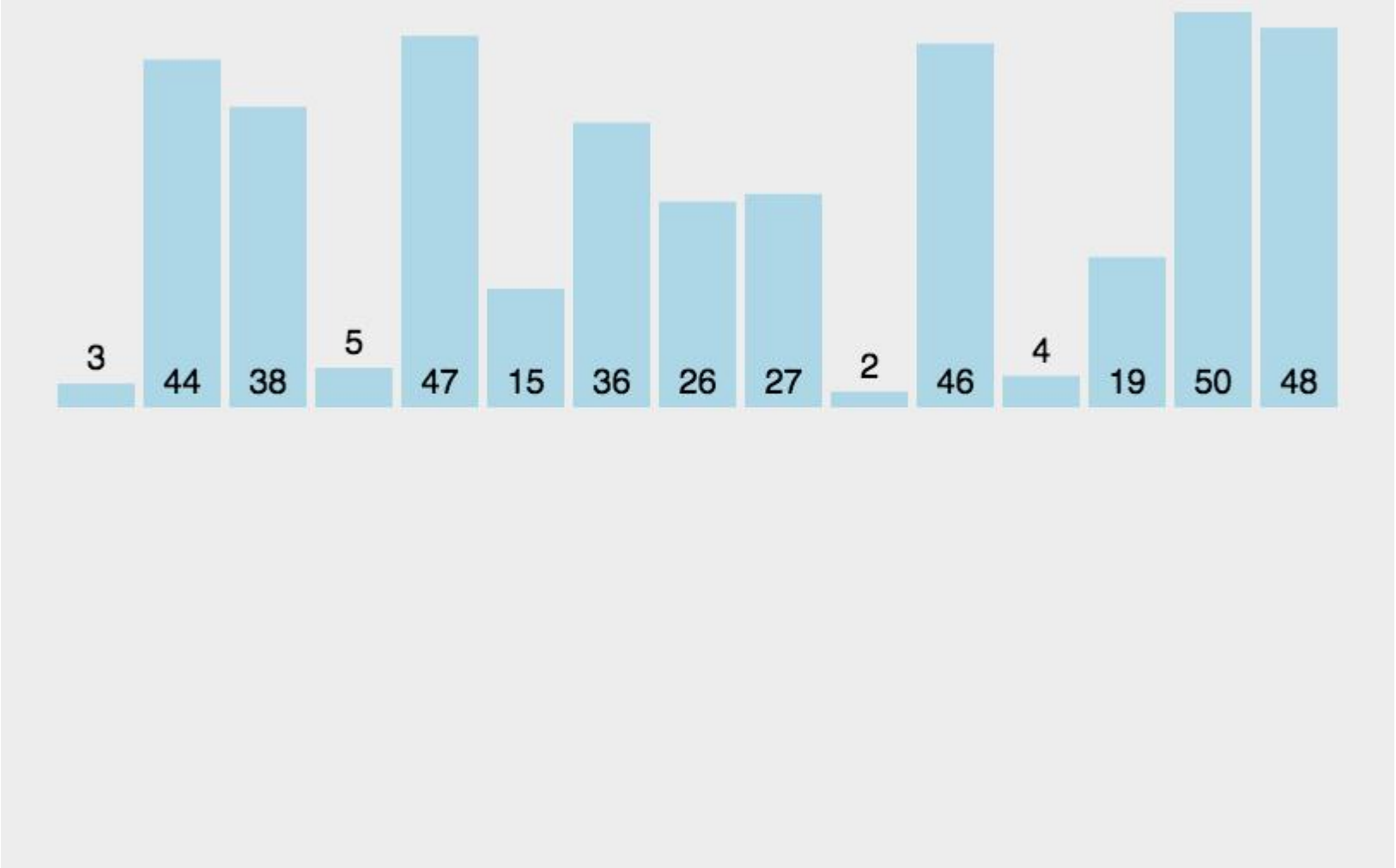
第一步，**划分子区间**：每次递归的从中间把数据划分为左区间和右区间。原始区间为 [l,r]，l=1，r=n，下标从 1 开始。然后从中间元素划分，mid=(r-l)/2;
划分之后的左右区间分别为 [l,mid]，右区间为 [mid+1,r]。重复过程，直到每个子区间只有一个或者两个元素。

第二步，**合并子区间**：子区间划分好以后，分别对左右子区间进行排序，排好序之后，在递归的把左右子区间进行合并。

开辟一个数组 a，存放 l 到 mid 之间（也就是需要归并的左边数组）的元素；
开辟一个数组 b，存放 mid 到 r 之间（也就是需要归并的右边数组）的元素；
a 与 b 中的数字逐个比较，把**较小的先放在数组 A 中（从数组 a1 开始存放，依次往后覆盖原来的数）**，然后**较小的数组指针往后移动**，指向下一位再和另外一个数组比较。

扩展：逆序对

归并排序动画



扩展：逆序对

归并排序代码实现

```
void merge_sort(int q[],int l,int r)
{
    if(l>=r)
        return ;
    int mid=(l+r)/2;
    merge_sort(q,l,mid);//递归划分左子区间
    merge_sort(q,mid+1,r);//递归划分右子区间
    int k=1,i=l,j=mid+1;
    while(i<=mid&&j<=r)//合并子区间
    {
        if(q[i]<=q[j])//q[i]更小存q[i],记得k和i同步+1
            temp[k++]=q[i++];
        else temp[k++]=q[j++];//存q[j],记得k和j同步+1
    }
    while(i<=mid)//左边数组有剩余
        temp[k++]=q[i++];
    while(j<=r)//右边数组有剩余
        temp[k++]=q[j++];
    for(int i=l,j=1;i<=r;i++,j++)//更新q数组,i和j同步+1
        q[i]=temp[j];
}
```

扩展：逆序对

递归对左右两半排序时，可以把左右两半各自内部的逆序对数作为子问题计算，因此我们只需要在合并时考虑**“左边一半里一个较大的数”与“右边一半里一个较小的数”**构成逆序对的情形，求出这种情形的个数。

合并两个有序序列a[l~mid] 与a[mid+1~r]可以采用两个指针i与 j分别对二者进行扫描的方式，不断比较两个指针所指向数值a[i]和a[j]的大小，将小的那个加入到排序的结果数组中。若**小的那个是a[j]，则 a[i~mid]都比 a[j]要大**，它们都会与a[j]构成逆序对，区间长度为mid-i+1，可以顺便统计到答案中。

归并排序

```
void merge_sort(int q[],int l,int r){
    if(l>=r)
        return ;
    int mid=(l+r)/2;
    merge_sort(q,l,mid);//递归划分左子区间
    merge_sort(q,mid+1,r);//递归划分右子区间
    int k=1,i=l,j=mid+1;
    while(i<=mid&&j<=r){//合并子区间
        if(q[i]<=q[j])//q[i]更小存q[i],记得k和i同步+1
            temp[k++]=q[i++];
        else{
            res+=(mid-i+1);//增添部分
            temp[k++]=q[j++];//存q[j],记得k和j同步+1
        }
    }
    while(i<=mid)//左边数组有剩余
        temp[k++]=q[i++];
    while(j<=r)//右边数组有剩余
        temp[k++]=q[j++];
    for(int i=l,j=1;i<=r;i++,j++)//更新q数组,i和j同步+1
        q[i]=temp[j];
}
```

扩展：逆序对

树状数组维护逆序对的个数

由于原序列中存在的元素较大需进行离散化，且原序列中可能跟存在相同元素，故用**结构体的方式**进行离散化

任意给定一个集合a，如果用 t[val]保存数值val 在集合a中出现的次数，那么数组 t在[l,r]上的区间和(即 $\sum_{i=l}^{r} t[i]$)就表示集合a中范围在[l,r]内的数有多少个。

我们可以在集合a的数值范围上建立一个树状数组,来维护t的前缀和。这样即使在集合a中插入或删除一个数,也可以高效地进行统计。

扩展：逆序对

树状数组维护逆序对的个数

由于原序列中存在的元素较大需进行离散化，且原序列中可能跟存在相同元素，故用**结构体的方式**进行离散化

按照上述思路，利用树状数组也可以求出一个序列的逆序对个数:

1.在序列a的数值范围上建立树状数组，初始化为全零。

2.**倒序扫描**给定的序列a，对于每个数a[i]:

- 执行“单点增加”操作，即把位置a[i]上的数加1（相当于 t[a[i]]++），同时正确维护t的前缀和。这表示数值a[i]又出现了1次。
- 在树状数组中查询前缀和[1,a[i]-1]，累加到答案ans中。

3.ans即为所求。

```
for(int i=n;i>=1;i--){
    add(a[i],1);
    res+=sum(a[i]-1);
}
```

扩展：逆序对

树状数组维护逆序对的个数

由于原序列中存在的元素较大需进行离散化，且原序列中可能跟存在相同元素，故用**结构体的方式**进行离散化

按照上述思路，利用树状数组也可以求出一个序列的逆序对个数:

对于中间过程倒序扫描给定的序列a
为什么要倒序扫描呢？可以正序扫描吗？

倒序扫描：**在自己之前出现，说明这个数在自己后面，求比自己小1的前缀和sum（n-1），即求在自己后面比自己小的数。**

倒序用树状数组倒序处理数列，当前数字的前一个数的前缀和即为以该数为较大数的逆序对的个数。例如 {5, 4, 2, 6, 3, 1}，倒序处理数字：
数字 1。把 t[a[1]] 加一，计算 a[1] 前面的前缀和 sum(0)，逆序对数量 ans = ans + sum(0) = 0；
数字 3。把 t[a[3]] 加一，计算 a[3] 前面的前缀和 sum(2)，逆序对数量 ans = ans + sum(2) = 1；
数字 6。把 t[a[6]] 加一，计算 a[6]前面的前缀和 sum(5)，逆序对数量 ans = ans + sum(5) = 1+ 2 = 3；
数字 2。把 t[a[2]] 加一，计算 a[2]前面的前缀和 sum(1)，逆序对数量 ans = ans + sum(1) = 3 + 1 = 4；
数字 4。把 t[a[4]] 加一，计算 a[4]前面的前缀和 sum(3)，逆序对数量 ans = ans + sum(3) = 4 + 3 = 7；
数字 5。把 t[a[5]] 加一，计算 a[5]前面的前缀和 sum(4)，逆序对数量 ans = ans + sum(4) = 7 + 4 = 11；

扩展：逆序对

树状数组维护逆序对的个数

由于原序列中存在的元素较大需进行离散化，且原序列中可能跟存在相同元素，故用**结构体的方式**进行离散化

按照上述思路，利用树状数组也可以求出一个序列的逆序对个数:

对于中间过程倒序扫描给定的序列a
为什么要倒序扫描呢？可以正序扫描吗？

当然也**可以正序扫描**
正序扫描：在自己之前出现，说明这个数在自己前面，**求前缀和sum(n),算在自己前面比自己小的数加上自己，用总的个数减去这个数**，就是在自己前面比自己大的数。

正序当前已经处理的数字个数减掉当前数字的前缀和即为以该数为较小数的逆序对个数。例如 {5, 4, 2, 6, 3, 1},
正序处理数字:
数字 5。把 t[a[5]] 加一，当前处理了 1 个数，ans = ans + (1-sum(5)) = 0;
数字 4。把 t[a[4]] 加一，当前处理了 2 个数，ans = ans+(2-sum(4)) = 0+1 = 1;
数字 2。把 t[a[2]] 加一，ans = ans + (3-sum(2)) = 1+2 = 3;
数字 6。把 t[a[6]] 加一，ans = ans + (4-sum(6)) = 3+0 = 3;
数字 3。把 t[a[3]] 加一，ans = ans + (5-sum(3)) = 3+2 = 5;
数字 1。把 t[a[1]] 加一，ans = ans + (6-sum(1)) = 5+6 = 11;

```
for(int i=1;i<=n;i++){
    add(a[i],1);
    res+=i-sum(a[i]);
}
```

扩展：逆序对

树状数组维护逆序对的个数

由于原序列中存在的元素较大需进行离散化，且原序列中可能跟存在相同元素，故用**结构体的方式**进行离散化

按照上述思路，利用树状数组也可以求出一个序列的逆序对个数:

在这个算法中，因为倒序扫描，“已经出现过的数”就是在a[i]后边的数，所以我们通过树状数组查询的内容就是“每个 a[i]后边有多少个比它小”。每次查询的结果之和当然就是逆序对个数。时间复杂度为0((N + M) log M),M为数值范围的大小。

当**数值范围较大**时，当然可以先进行离散化，再用树状数组进行计算。不过因为离散化本身就要通过排序来实现，所以在这种情况下就不如直接用归并排序来计算逆序对数了。

扩展：逆序对

线段树维护逆序对的个数

由于原序列中存在的元素较大需进行离散化，且原序列中可能跟存在相同元素，故用**结构体的方式**进行离散化

线段数维护逆序对的思想和树状数组类似，采用**权值线段树**的方式进行维护每个值出现的次数即可

```
build(1,1,n);//线段树的建空树

for(int i=n;i>=1;i--){
    modify(1,id[i],1);//线段树的单点修改
    res+=query(1,1,id[i]-1);//线段树的区间查询
}
```