基础数据结构

湖南师大附中 许力

序列维护工具

线段树/树状数组

序列维护

- 序列维护是OI中常见的一类问题
- 给定一段连续的序列，需要支持如下一些操作：

1. 将序列内给定范围内的每个元素加上一个值（不保证非负）
2. 求序列内给定范围的和

序列维护

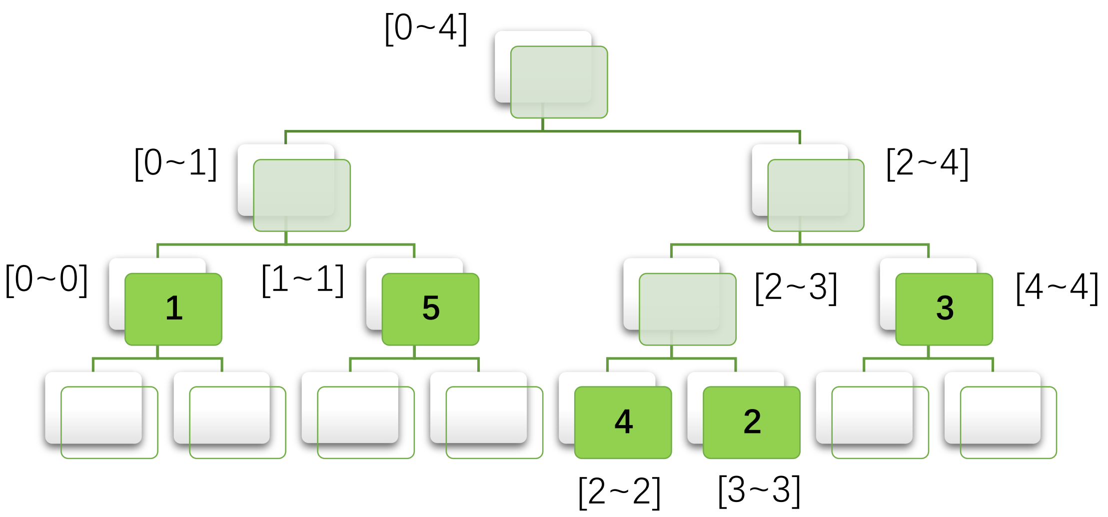
- 如果只有操作2，就是静态序列维护。之前学习的前缀和，就可以胜任
- 但是有了操作1的存在，问题就变成动态序列维护，前缀和已经无法解决

线段树（segment tree）

- 线段树就是用于解决序列维护问题，尤其是动态序列维护问题的工具，而且它不仅支持区间加修改和区间求和，还支持区间乘修改和区间求最值等

- 假定序列[1，5，4，2，3]

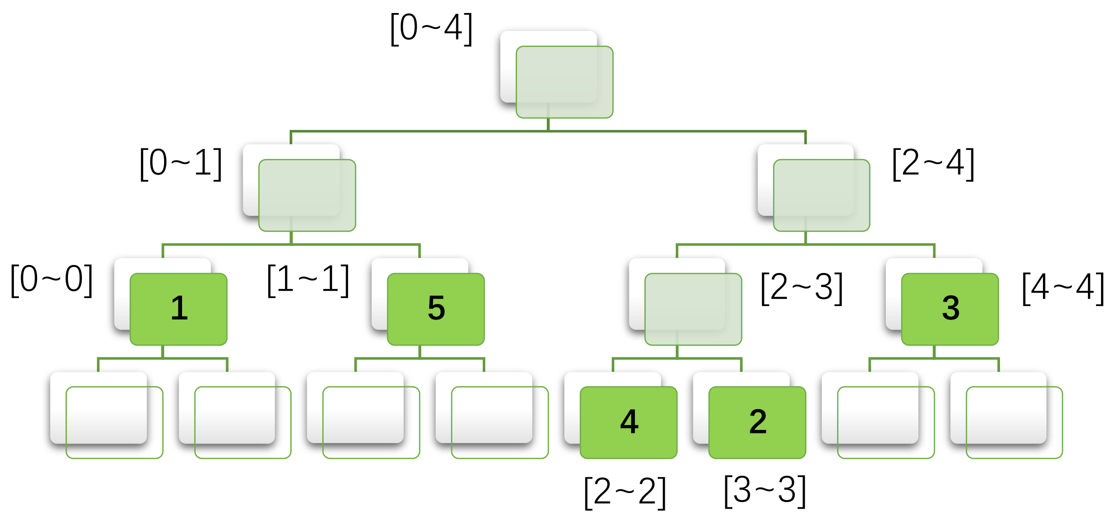
id	0	1	2	3	4
data	1	5	4	2	3



线段树

- 根节点表示区间[0, n-1]，然后把区间分成两半，分别由左右子树表示，并递归下去
- 叶节点一般是单点，也可以是单元区间
- 线段树是平衡的二叉树，线段树把任意区间[a,b]至多分成log(b-a)份，使得区间维护的复杂度从O(n)降到O(logn)

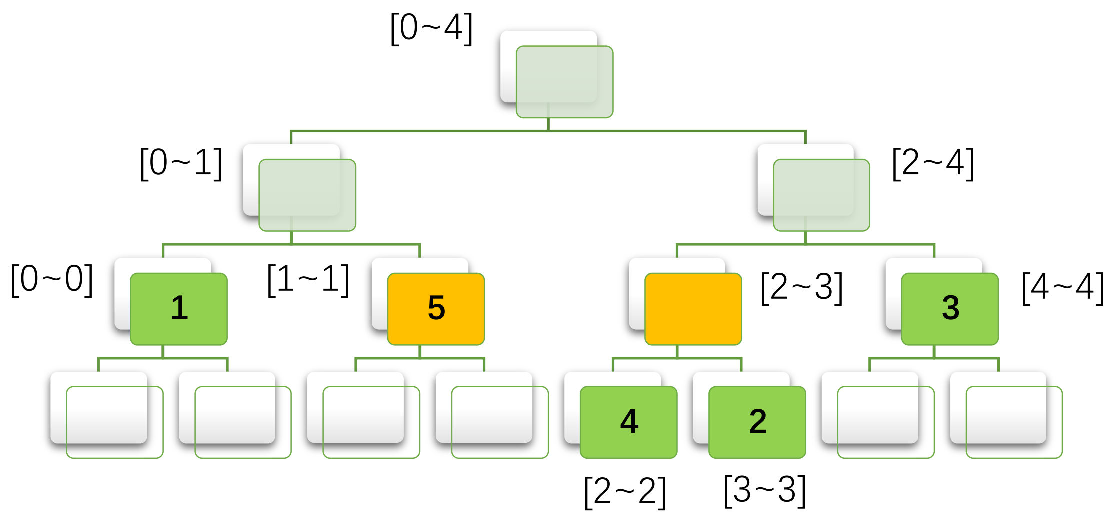
id	0	1	2	3	4
data	1	5	4	2	3



线段树

- 如果我们要维护的范围是[5、4、2]，对应的区间编号就是[1~3]
- 那么涉及到的节点如图所示

id	0	1	2	3	4
data	1	5	4	2	3



线段树的存储结构

- 和堆、并查集使用一维数组来模拟实现不同，线段树需要开结构体数组存储每个节点，每个节点代表序列中的一个区间，包括：

1. 该节点所管辖区间的左右端点
2. 该节点自身的权值
3. 其他根据题意需要存储的内容

```
struct node
{
    int left, right;
    int data;
} tree[N*4];
```

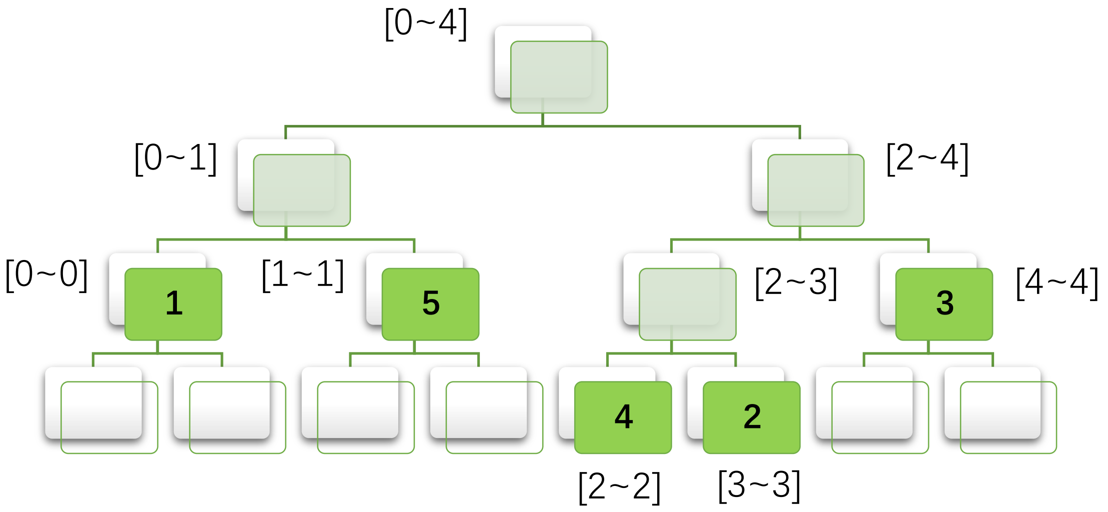
线段树的存储结构

- 但是还是可以直接用一维数组模拟
- 根据之前学习二叉树得到的结论，对每个节点p，有

```
fa[i] = i / 2 = i >> 1;
lc[i] = i * 2 = i << 1;
rc[i] = i * 2 + 1 = i << 1 | 1;
```

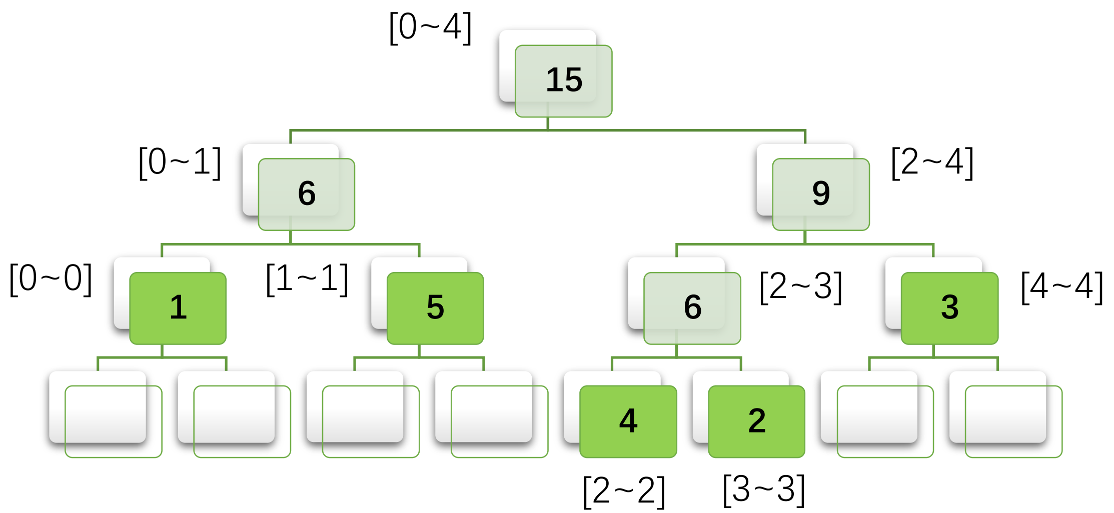
- 那么要存下一棵线段树，空间开多大？

```
tree[N * 4];
```



建立一棵线段树

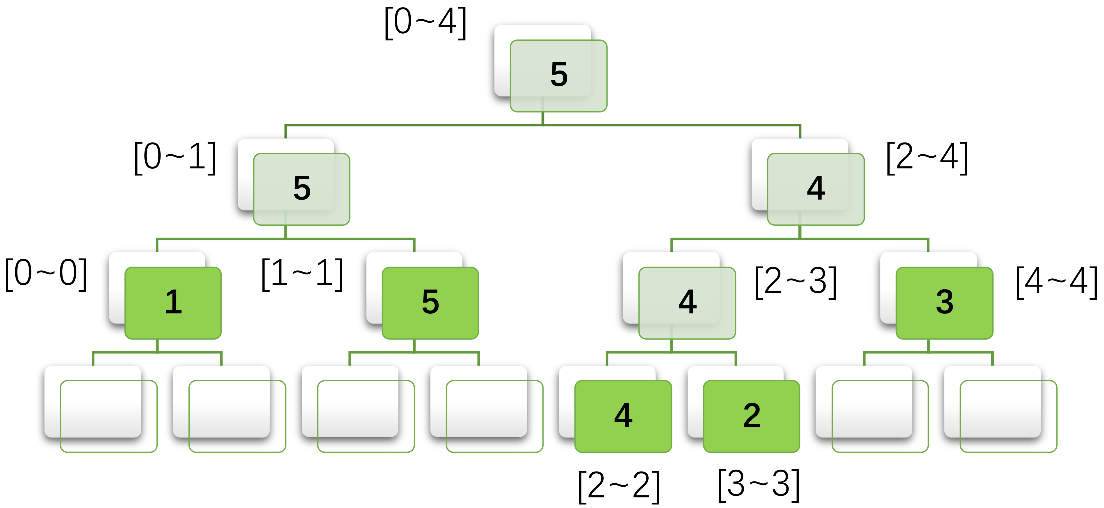
- 把区间[l, r]建成一棵线段树
- 原有的序列存放在数组a[ ]中
- 这里的题意是节点存储区间和




```
void build(int l, int r, int i)
//把区间[l,r]建成一棵以 i为根节点的树
{
    tree[i].left = l;
    tree[i].right = r;
    if (l == r) //叶节点
    {
        tree[i].data = a[l];
        return;
    }
    int m = (l + r) >> 1;
    build(l, m, i << 1); //递归建左子树
    build(m + 1, r, i << 1 | 1); //递归建右子树
    tree[i].data = tree[i << 1].data + tree[i << 1 | 1].data;
}
```

建立一棵线段树

- 把区间[l, r]建成一棵线段树
- 原有的序列存放在数组a[ ]中
- 如果要维护的是最值：

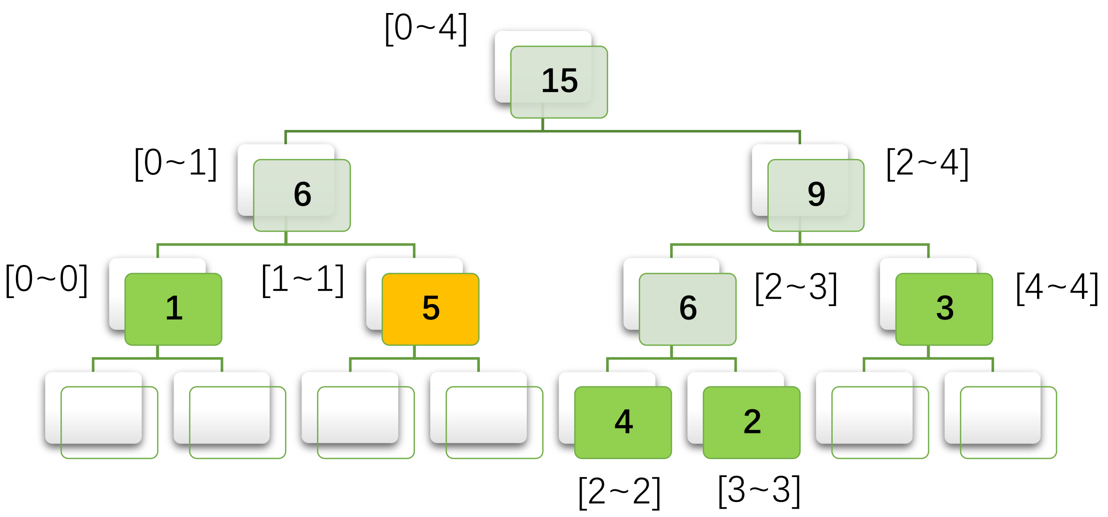



```
void build(int l, int r, int i)
//把区间[l,r]建成一棵以 i为根节点的树
{
    tree[i].left = l;
    tree[i].right = r;
    if (l == r) //叶节点
    {
        tree[i].data = a[l];
        return;
    }
    int m = (l + r) >> 1;
    build(l, m, i << 1); //递归建左子树
    build(m + 1, r, i << 1 | 1); //递归建右子树
    tree[i].data = max(tree[i << 1].data, tree[i << 1 | 1].data);
}
```

线段树的（区间）查询操作

- 我们现在要查询的区间[l，r]是[1，3]
- 我们从根节点开始，发现根节点的lc[0，1]和[1，3]有交集
- 要怎么发现这一点？
- 因为0<1

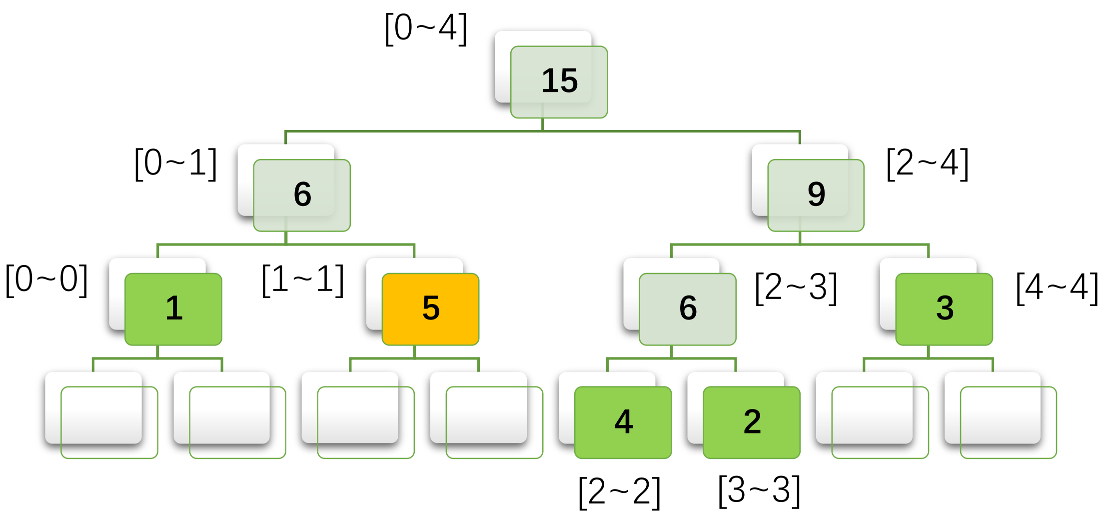
id	0	1	2	3	4
data	1	5	4	2	3



线段树的（区间）查询操作

- 我们现在要查询的区间[l，r]是[1，3]
- 我们从根节点开始，发现根节点的lc[0，1]和[1，3]有交集
- 递归到[0，1]，发现该节点的rc[1，1]被[1，3]完全包含
- 这又要怎么发现?
- 因为1<3

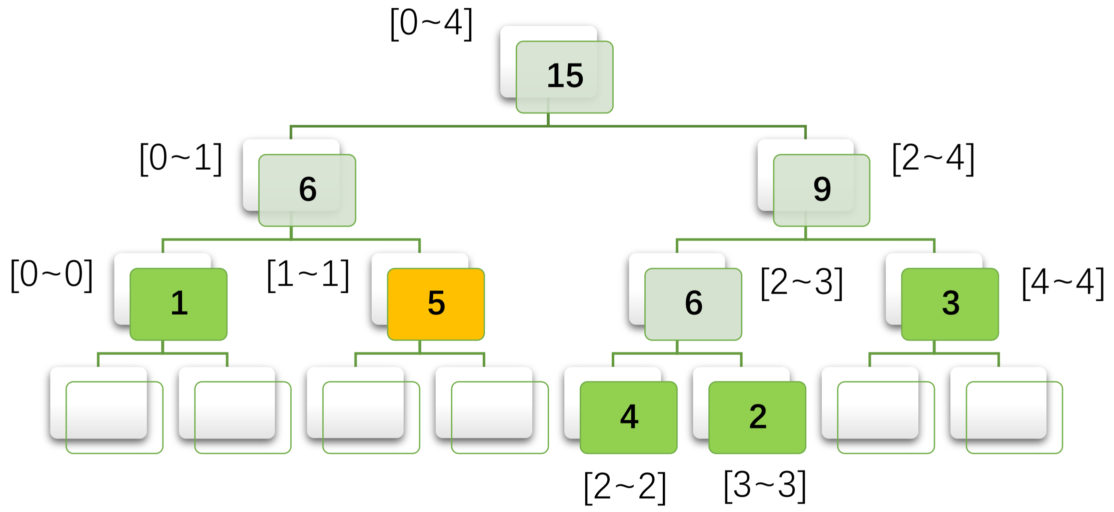
id	0	1	2	3	4
data	1	5	4	2	3



线段树的（区间）查询操作

- 我们现在要查询的区间[l，r]是[1，3]
- 我们从根节点开始，发现根节点的lc[0，1]和[1，3]有交集
- 递归到[0，1]，发现该节点的rc[1，1]被[1，3]完全包含
- 于是将其data：5计入答案并返回

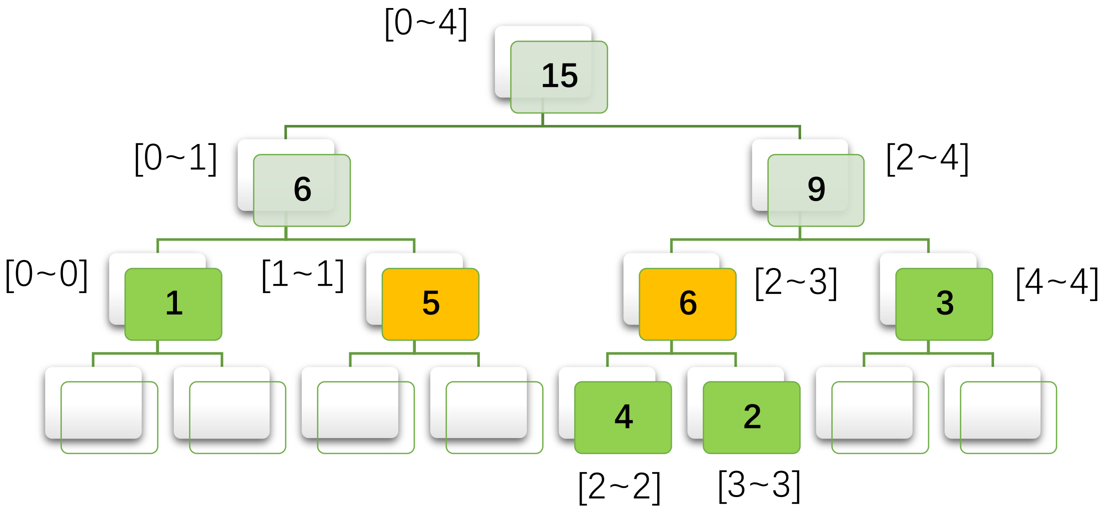
id	0	1	2	3	4
data	1	5	4	2	3



线段树的（区间）查询操作

- 返回到根节点后，发现根节点的rc[2，4]和[1，3]有交集
- 递归到[2，4]，发现该节点的lc[2，3]被[1，3]完全包含，于是将其data：6计入答案并返回
- 于是得到最终的答案：5+6=11

id	0	1	2	3	4
data	1	5	4	2	3



线段树区间查询

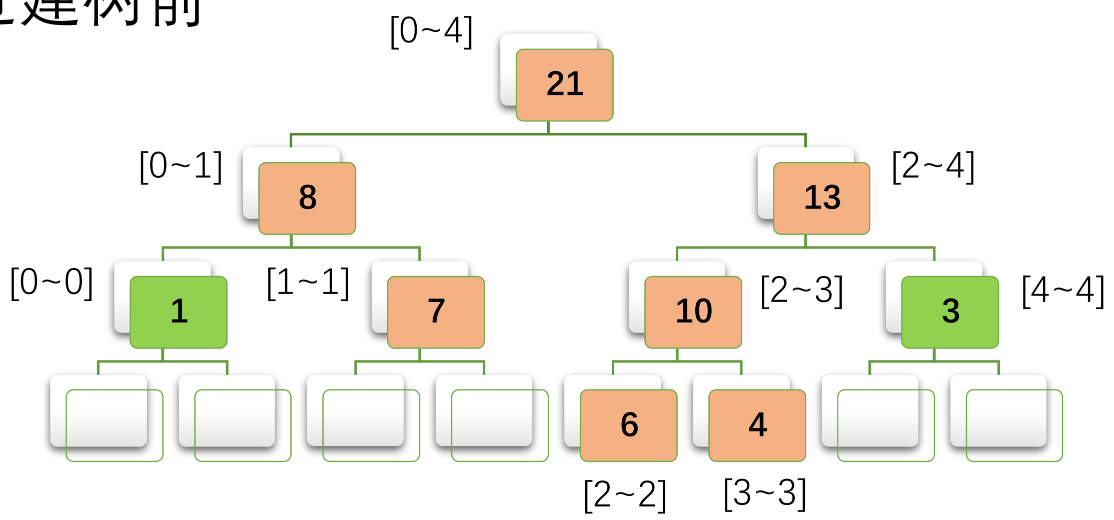
- 注意这里的区间查询不带修改

```
int query(int l, int r, int i) //查询区间[l,r], 当前节点 i
{
    if (tree[i].left >= l && tree[i].right <= r) return tree[i].data;
    // i代表的区间被查询区间完全包住, 直接返回 i的权值
    int res = 0;
    if (tree[i << 1].right >= l) res += query(l, r, i << 1);
    //左儿子和查询区间有交集, 递归查询左子树
    if (tree[i << 1 | 1].left <= r) res += query(l, r, i << 1 | 1);
    //右儿子和查询区间有交集, 递归查询右子树
    return res;
 }
```

线段树的（区间）修改操作

- 然而支持区间修改操作，才是线段树的精髓所在
- 区间修改，是指修改区间内的所有叶节点（就是原序列）的值
- 我们现在要修改的区间[l, r]是[1, 3]，k值是+2
- 假如我们直接修改[1, 3]及相关父节点、祖父节点，涉及到的节点会非常多
- 复杂度将被推高到O(nlogn)，甚至超过建树前

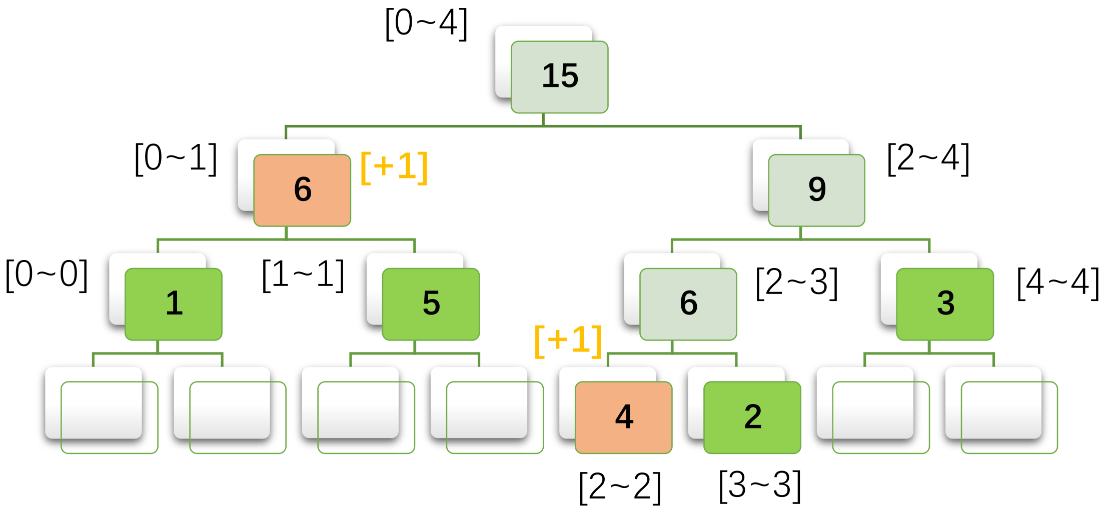
id	0	1	2	3	4
data	1	5	4	2	3



线段树的（区间）修改操作

- 这种状况下，我们会想到给节点打上标记的思路
- 比如我们修改[0，2]区间，k是+1
- 我们找到[0，1]节点以及[2，2]节点打上修改标记+1
- 然后查询区间[1，3]，但这时[1，1]和[2，3]上是没有修改标记的
- 查询的答案自然是错的!

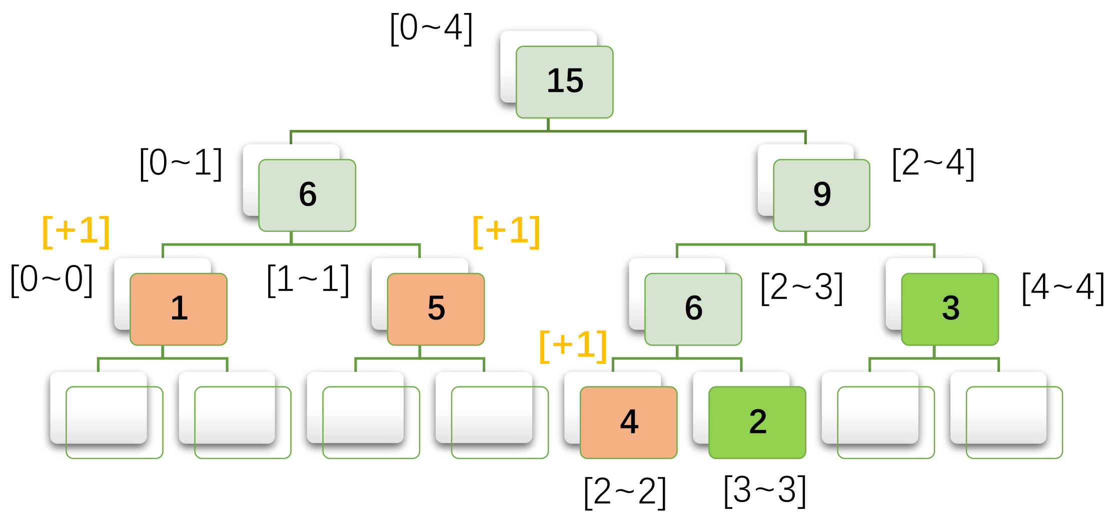
id	0	1	2	3	4
data	1	5	4	2	3



线段树的延迟标记

- 线段树解决这个问题，应用的是特殊标记：延迟标记，也叫懒标记（lazy）
- 回到刚才的问题，我们查询[1，1]的时候，只要它在[0，1]区间内，就把[0，1]上的标记传下去，同时消除原来在[0，1]上的标记

id	0	1	2	3	4
data	1	5	4	2	3

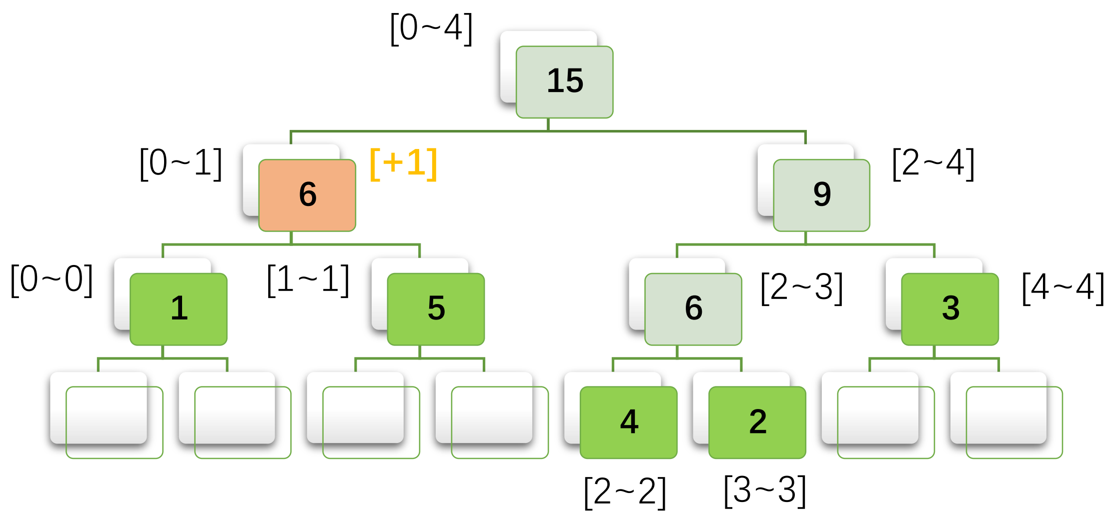


线段树的延迟标记

- 我们首先找到[0，1]
- 怎么找到的?
- 0>=0 && 1<=2
- 给当前节点（管辖区间[0，1]）打上延迟标记 k（需要在node中增加lazy域）

```
tree[i].lazy += k;
```

id	0	1	2	3	4
data	1	5	4	2	3

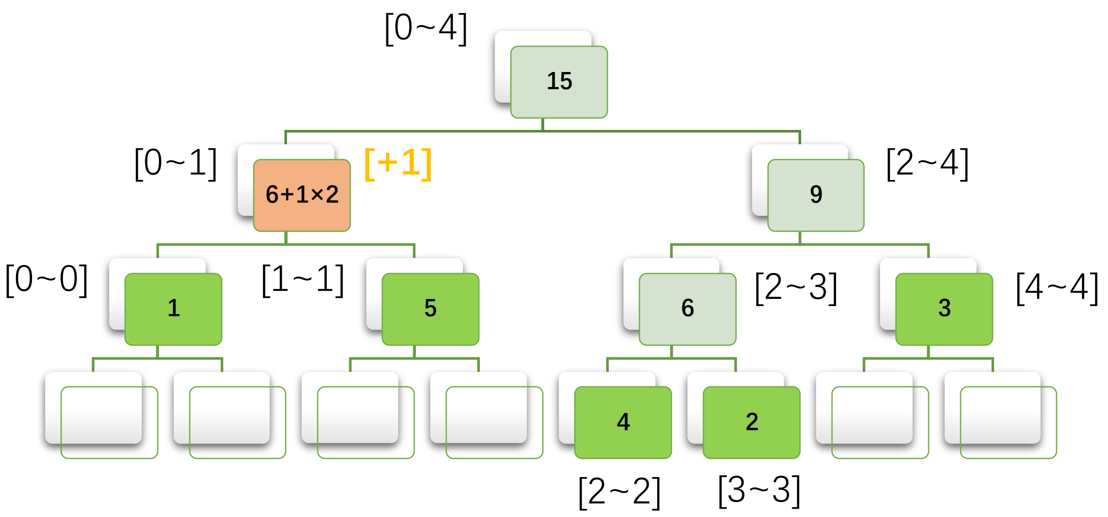


线段树的延迟标记

- 那么这时候，当前节点[0，1]的data域会发生怎样的变化?
- [0，1]节点的data域要在原值基础上加：k * （该节点管辖的区间长度）

```
tree[i].data += k * (tree[i].right - tree[i].left + 1);
```

id	0	1	2	3	4
data	1	5	4	2	3

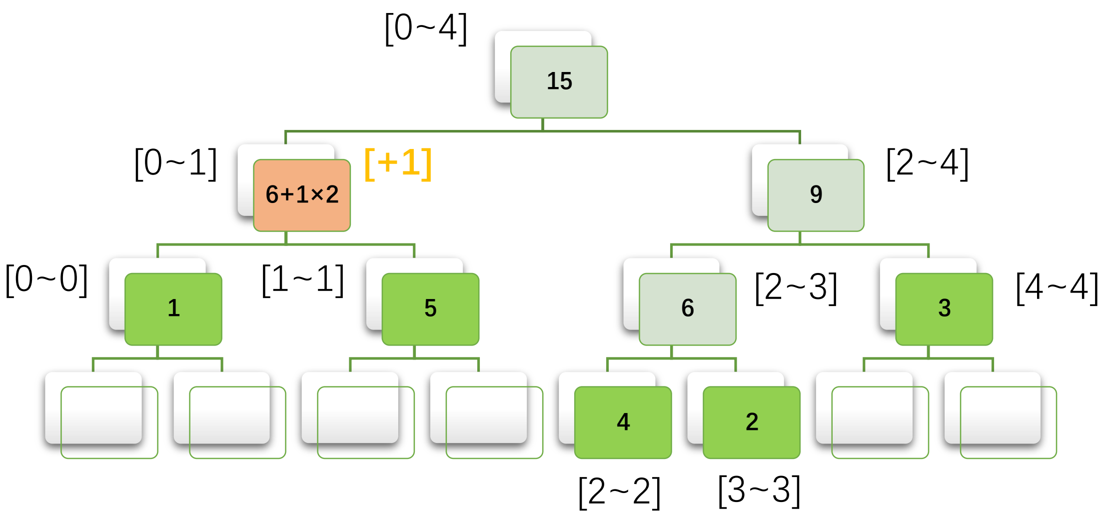


线段树的（区间）修改操作

- 线段树的区间修改操作总结如下：

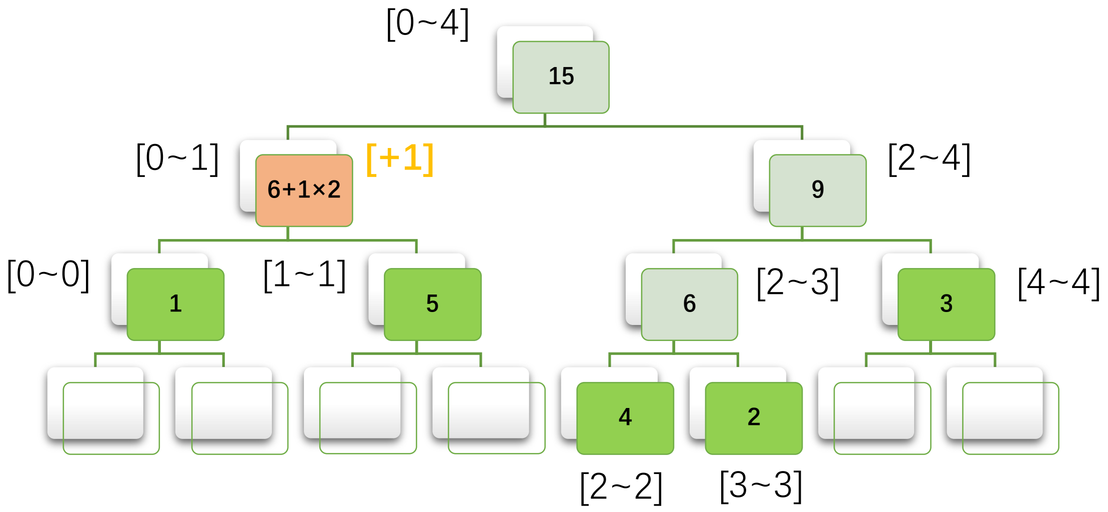
1. 如果当前区间被目标区间完全覆盖，则修改该节点的data域，打上延迟标记
2. 如果没有完全覆盖，则下传延迟标记
3. 如果左、右子树与目标区间有交集，则分别递归下去
4. 回溯时依然更新父节点的data域

id	0	1	2	3	4
data	1	5	4	2	3



线段树的（区间）修改代码

- 于是我们得到区间修改的代码




```
void modify(int l, int r, int k, int i)
//修改区间[l,r]的值 +k, 当前节点 i
{
    if (当前节点管辖的区间被修改目标区间完全包含)
    {
        tree[i].data += k * (tree[i].right - tree[i].left + 1);
        tree[i].lazy += k;
        return;
    }
    向下传递lazy标记;
    往左子树递归;
    往右子树递归;
    回溯时得到 data域;
}
```

延迟标记的向下传递

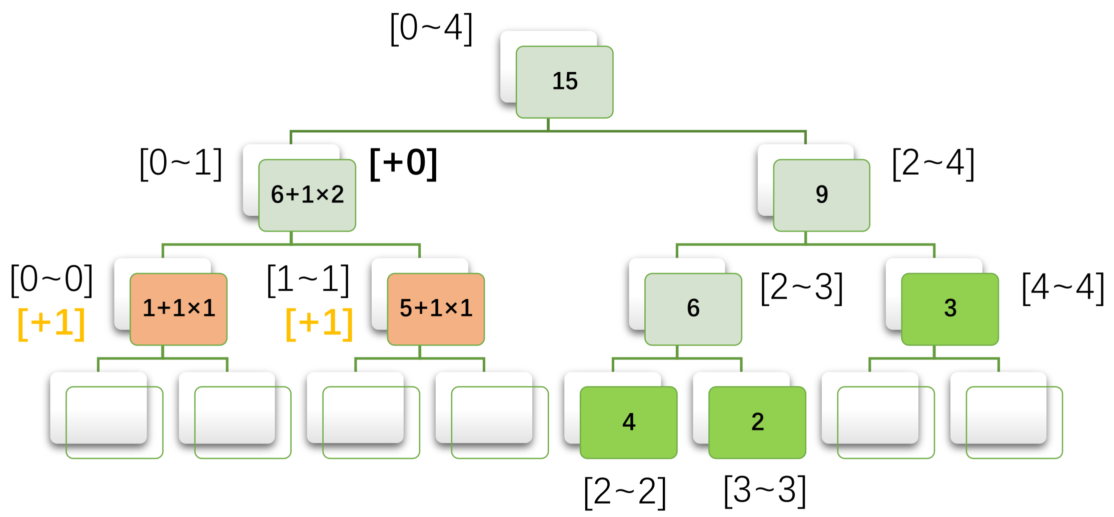
- 在查询操作query和修改操作modify中，都需要向下传递lazy标记

```
tree[i << 1].lazy += tree[i].lazy;
tree[i << 1 | 1].lazy += tree[i].lazy;
```

- 并且一旦传递下去，原节点的lazy标记要清零

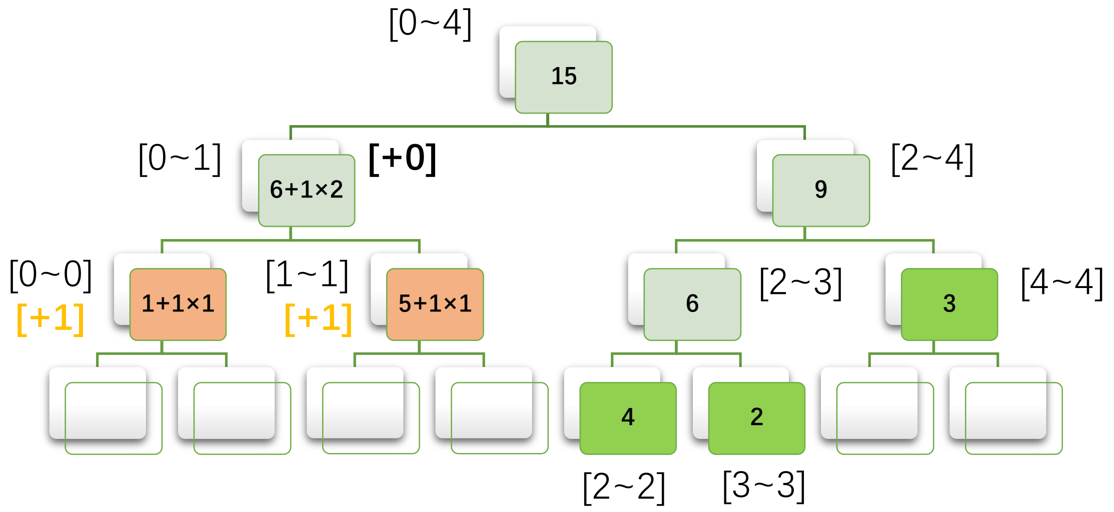
```
tree[i].lazy = 0;
```

id	0	1	2	3	4
data	1	5	4	2	3



延迟标记的向下传递

- 除了向下传递lazy标记外，还要同步修改左右儿子的data域




```
void pushdown(int i)
//向下传递 i的 lazy
{
    if (tree[i].lazy != 0)
    {
        lazy向左儿子传递;
        lazy向右儿子传递;
        修改左儿子的data;
        修改右儿子的data;
        消除原父节点i的lazy;
    }
}
```

完整参考代码

```cpp
#include<bits/stdc++.h>
using namespace std;
#define N 1000010
int n, m, op, x, y, k, a[N];
struct node
{
    int left, right;
    int data, lazy;
} tree[N * 4];
```

```cpp
void build(int l, int r, int i)
//把区间[l,r]建成一棵以 i为根节点的树
{
    tree[i].lazy = 0;
    tree[i].left = l;
    tree[i].right = r;
    if (l == r) //叶节点
    {
        tree[i].data = a[l];
        return;
    }
    int m = (l + r) >> 1;
    build(l, m, i << 1); //递归建左子树
    build(m + 1, r, i << 1 | 1); //递归建右子树
    tree[i].data = tree[i << 1].data + tree[i << 1 | 1].data;
}
```

完整参考代码

```
void pushdown(int i)  //向下传递 i的 lazy
{
    if (tree[i].lazy != 0)
    {
        tree[i << 1].lazy += tree[i].lazy;  // lazy向左儿子传递
        tree[i << 1 | 1].lazy += tree[i].lazy; // lazy向右儿子传递
        int m = (tree[i].left + tree[i].right) >> 1;
        tree[i << 1].data += tree[i].lazy * (m - tree[i << 1].left + 1);
        tree[i << 1 | 1].data += tree[i].lazy * (tree[i << 1 | 1].right - m);
        tree[i].lazy = 0;  //消除父节点的 lazy
    }
}
```

```
void modify(int l, int r, int k, int i)
//修改区间[l,r]的值 +k， 当前节点 i
{
    if (tree[i].left >= l && tree[i].right <= r)
    {
        tree[i].data += k * (tree[i].right - tree[i].left + 1);
        tree[i].lazy += k;
        return;
    }
    pushdown(i);  //向下传递
    if (tree[i << 1].right >= l) modify(l, r, k, i << 1);
    if (tree[i << 1 | 1].left <= r) modify(l, r, k, i << 1 | 1);
    tree[i].data = tree[i << 1].data + tree[i << 1 | 1].data;
}
```

完整参考代码

```
int query(int l, int r, int i) //查询区间[l,r], 当前节点 i
{
    if (tree[i].left >= l && tree[i].right <= r) return tree[i].data;
    // i代表的区间被查询区间完全包住, 直接返回 i的权值
    pushdown(i);
    int res = 0;
    if (tree[i << 1].right >= l) res += query(l, r, i << 1);
    //左儿子和查询区间有交集, 递归查询左子树
    if (tree[i << 1 | 1].left <= r) res += query(l, r, i << 1 | 1);
    //右儿子和查询区间有交集, 递归查询右子树
    return res;
}
```

```
int main()
{
    scanf("%d%d", &n, &m);
    for (int i = 1; i <= n; i ++) scanf("%d", &a[i]);
    build(1, n, 1);  //以 1号节点为根节点建树
    while (m --)
    {
        scanf("%d", &op);
        if (op == 1)
        {
            scanf("%d%d%d", &x, &y, &k);
            modify(x, y, k, 1);
        }
        if (op == 2)
        {
            scanf("%d%d", &x, &y);
            printf("%d\n", query(x, y, 1));
        }
    }
    return 0;
}
```

课外加练

- luogu 3372 线段树模板
- luogu 3373 线段树模板
- luogu 1276 校门外的树
- luogu 3178 树上操作