Segment Tree(线段树)

1. 引入

我们先来看下面这样一个问题：

给定一个长度为 10^5 的数组 a ，你需要支持两种操作：

- $1\ \ l\ \ r\ \ x\ \ \ \Rightarrow\ a_i := a_i + x\ (l \le i \le r)$
- $2\ \ l\ \ r\ \Rightarrow$ 输出 $\sum\limits_{i=l}^{r} a_i$

由昨天的课程我们知道，这个问题使用一些技巧和转化后，可以使用树状数组在 $O(logn)$ 内完成每个操作。

如果我们再加入一种操作呢？

- $3\ \ l\ \ r\ \ x \Rightarrow\ a_i := x\ (l \le i \le r)$

再来一种？

- $4\ \ l\ \ r\ \Rightarrow$ 输出 $\underset{l \le i \le r}{max}\{a_i\}$

再来一种？

- $5\ \ l\ \ r\ \Rightarrow$ 输出 $\underset{l \le i \le r}{min}\{a_i\}$

再来一种？

- $6\ \ l\ \ r\ \Rightarrow$ 输出 $\sum\limits_{i=l}^{r}(-1)^i \times a_i$

如果我们维护的信息需要支持各种稀奇古怪的操作，树状数组就没有办法帮我们完成这个事情了。

强大的 **线段树** 就要 **横空出世** 了！

2.线段树

2.1 简介

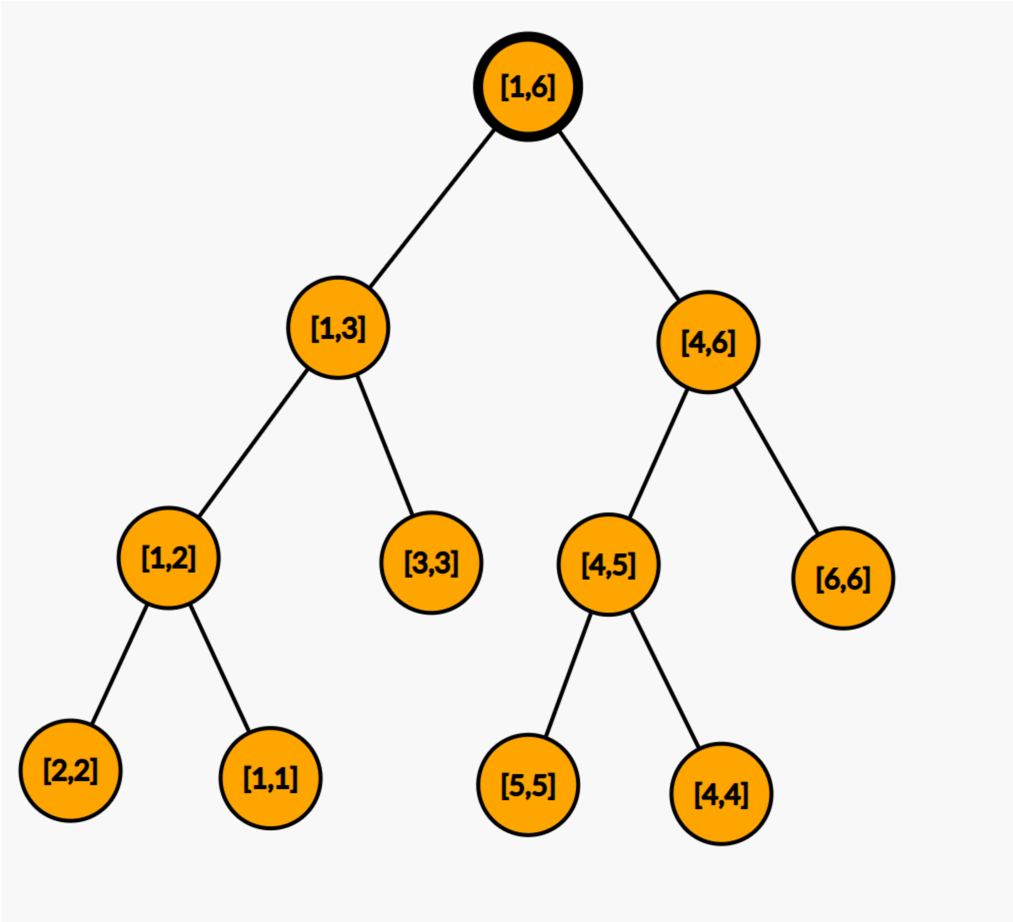
线段树是一种在算法竞赛中非常常见的数据结构。它是一种基于分治思想的 **二叉树** 结构，常用于在区间上进行信息统计。并且跟 **树状数组** 相比，其适用 **场景更多**，但 **码量较大** ，**常数** 也 **较大。**

线段树的每一个节点都代表了序列上的一个区间。

特别地，根节点代表总区间，如 $[1,N]$ ，叶子节点代表某个长度为 1 的特定位置，如$[x,x]$。

线段树上除 **叶子节点** 以外的每一个节点都 **有且仅有两个** 儿子：左儿子和右儿子。

如果这个节点代表的区间是 $[l,r]$，那么它的左儿子代表的区间是 $[l,\lfloor\frac{l+r}{2}\rfloor]$ ，右儿子代表的区间是 $[\lfloor\frac{l+r}{2}+1\rfloor,r]$ 。举例来说，区间 $[1,6]$ 构造的线段树如下图所示：

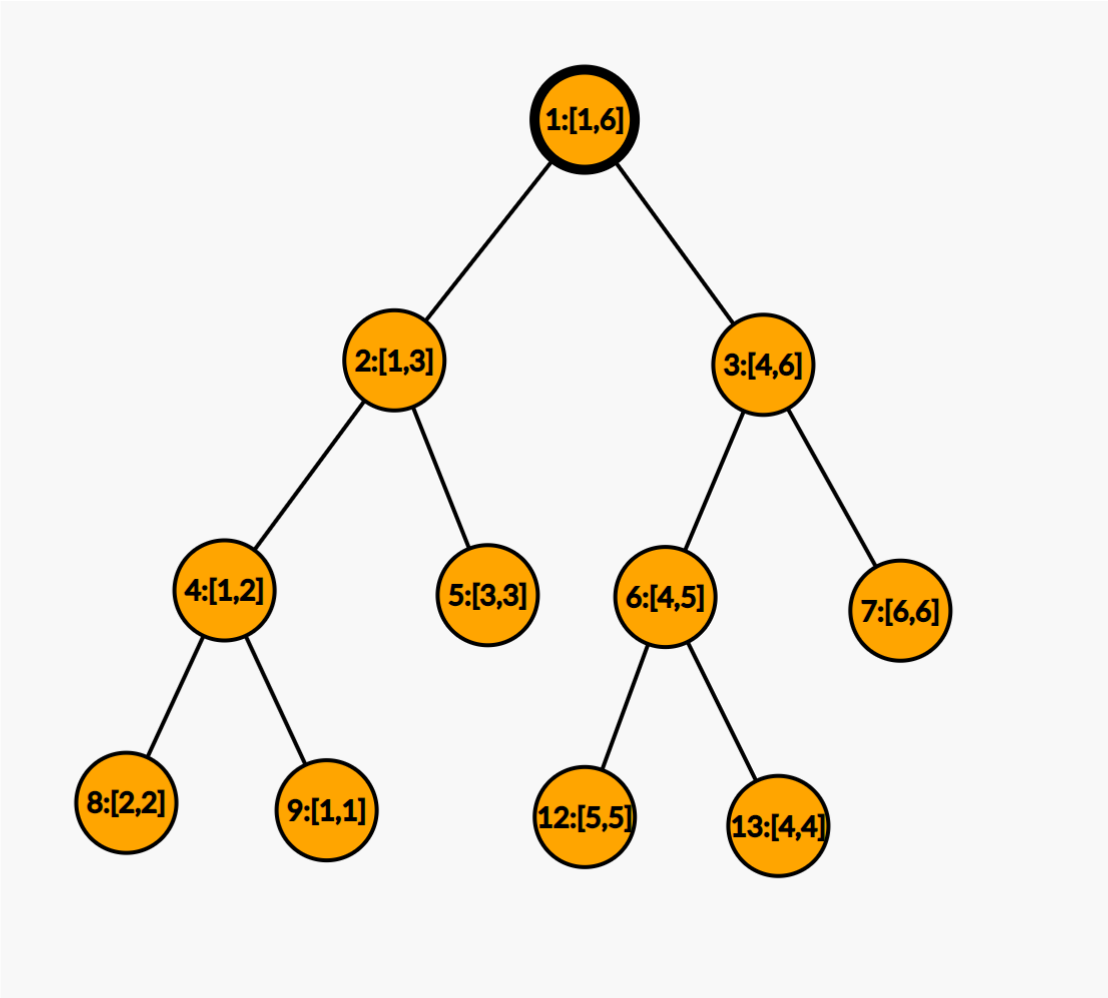


注意：上面的数值代表的是在数组当中的对应下标。

因为线段树除了叶子节点外，其他节点都 **有且仅有两个** 儿子，所以我们在存储的时候不使用邻接表或者邻接矩阵。我们直接对各个节点进行编号。具体的编号规则如下：

1. 根节点编号为 1 。
2. 编号为 x 的节点的左子节点编号为 $2 \times x$ ，右子节点编号为 $2 \times x + 1$ 。

这样一来，我们就能简单地使用一个 $struct$ 数组来保存线段树。当然，树的最后一层在数组中保存的位置是不连续的，我们直接空出那些位置就可以。上面的例子编号后：



在理想情况下，N 个叶节点的满二叉树有 $N+\frac{N}{2}+\frac{N}{4}+\cdots+2+1=2N-1$ 个结点。因为在上述存储方式下，最后还有一行产生了空余，所以保存线段树的 **数组长度要不小于 $4N$ 才保证不会越界。**

显然，树的高度是 $O(logn)$ 的。

2.2 操作

来看这样一个题：

给定长度为 n 的数组 a，你要支持下面这几种操作，且每种操作时间复杂度为 $O(logn)$：

$1\ \ x\ \ v \Rightarrow\ a[x]:=a[x]+v$

$2\ \ l\ \ r\ \Rightarrow$ 输出 $\sum\limits_{i=l}^{r}a_i$

$3\ \ l\ \ r\ \ x \Rightarrow\ a_i:=a_i+x\ (l\le i\le r)$

我们来一点点尝试解决。

2.2.1 建树

线段树的基本用途是对序列进行维护，支持查询与修改操作。

我们在区间 $[1,n]$ 上按照上述方法建立一棵线段树。由于线段树的结构很方便从下往上传递信息，所以我们一般使用递归来完成它的相关操作。下面这段代码对给定的长度为 n 的数组 a，建立线段树，每个节点上存储了对应区间内数组 a 中的区间和。

```
const int N = 1e5 + 10;
```

```
int a[N], n;
struct node{
    int l, r;
    int sum;
}seg[N << 2];

// 合并两个儿子的信息 上传给父亲
void up(int id){
    seg[id].sum = seg[id << 1].sum + seg[id << 1 | 1].sum;
}

//对节点编号为 id 维护区间[l, r]的节点初始化信息
void build(int id, int l, int r){
    seg[id].l = l;
    seg[id].r = r;
    //到达叶子节点，直接更新信息
    if(l == r){
        seg[id].sum = a[l];
        return ;
    }
    int mid = (l + r) >> 1;
    //递归 分别建立左右子树
    build(id << 1, l, mid);
    build(id << 1 | 1, mid + 1, r);
    //根据左右儿子节点内存储的信息来更新本节点的信息
    up(id);
}

build(1, 1, n); // 调用入口
```

2.2.2 单点修改

在对数组 a 建立完线段树后，我们来完成操作 1 。

我们从根节点出发，递归找到代表区间为 $[x, x]$ 的叶节点，改变它的值。然后从下往上更新它的所有祖先节点上保存的信息。

```
void change(int id, int x, int v){
    int l = seg[id].l;
    int r = seg[id].r;
    // 到达叶子节点，直接更新信息
    if(l == r){
        seg[id].sum += v;
        return;
    }
    int mid = (l + r) >> 1;
    if(x <= mid){
        // x 位于该节点的左儿子
        change(id << 1, x, v);
    }else{
        // x 位于该节点的右儿子
        change(id << 1 | 1, x, v);
    }
```

```
    //根据左右儿子节点内存储的信息来更新本节点的信息
    up(id);
}

change(1, x, v) //调用入口
```

这段代码的时间复杂度是 $O(logn)$ 的，因为显然在线段树的每一层它只会执行一次，所以花费跟树高相同。

2.2.3 区间询问

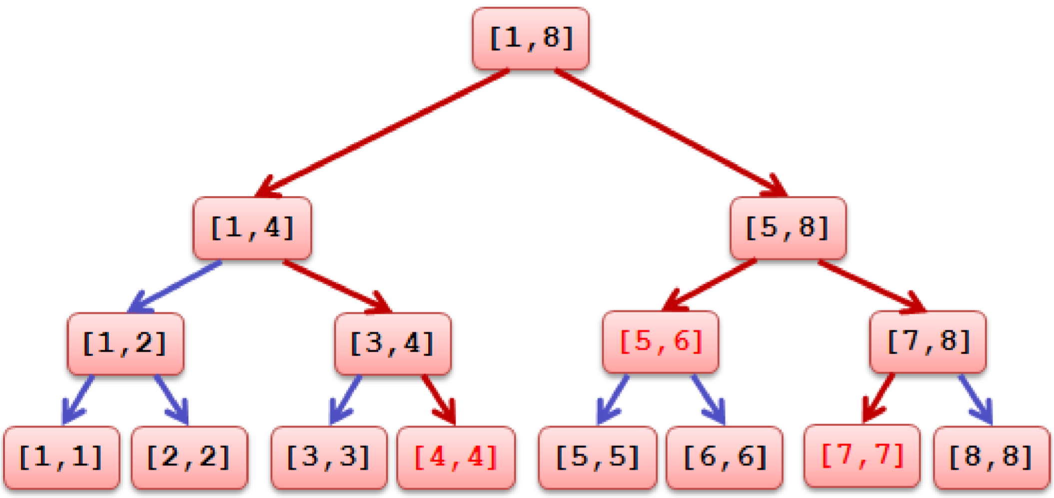
在对数组 a 建立完线段树后，我们来完成操作 2。

为了解决区间询问，我们需要在线段树上定位一个区间 $[ql, qr]$。

考虑这么一个算法，我们从根节点开始搜索，假设现在搜索到的节点的区间是 $[l, r]$，那么就有如下三种情况：

1. 如果区间 $[ql, qr]$ 包含了区间 $[l, r]$，即 $ql \le l \le r \le qr$ ，就把它加入答案。
2. 如果区间 $[ql, qr]$ 与区间 $[l, r]$ 不相交，即 $ql > r$ 或者 $qr < l$ ，那么就退出。
3. 如果不满足上述两种情况，那么就分别搜索这个节点的两个儿子。

举例来说，如果我们在由区间 $[1, 8]$ 建立的线段树中定位区间 $[4, 7]$ ，那么过程如图所示（红色边表示经过的边，红色字体的区间代表定位得到的区间）：




```
int query(int id, int ql, int qr){
    int l = seg[id].l;
    int r = seg[id].r;
    if(ql > r || qr < l) return 0;
    if(ql <= l && r <= qr) return seg[id].sum;
    return query(id << 1, ql, qr) + query(id << 1 | 1, ql, qr);
}

int query(int id, int ql, int qr){
    int l = seg[id].l;
    int r = seg[id].r;
    if(ql <= l && r <= qr) return seg[id].sum;
    int ans = 0;
    int mid = (l + r) >> 1;
```

```
    if(ql <= mid) ans += query(id << 1, ql, qr);
    if(qr > mid) ans += query(id << 1 | 1, ql, qr);
    return ans;
}

int query(int id, int ql, int qr){
    int l = seg[id].l;
    int r = seg[id].r;
    if(ql<= l && r <= qr){
        return seg[id].sum;
    }
    int mid = (l + r) >> 1;
    if(qr <= mid){
        return query(id << 1, ql, qr);
    }else if(ql > mid){
        return query(id << 1 | 1, ql, qr);
    }else{
        return query(id << 1, ql, qr) + query(id << 1 | 1, ql, qr);
    }
}
query(1, ql, qr) // 调用入口
```

这个操作的时间复杂度也是 $O(logn)$ 的。我们可以简单证明一下：

因为前两种情况是终止条件，只有在第三种情况的时候会继续递归，又因为线段树的每一个节点的儿子个数只有2，所以前两种情况的节点数不会超过第三种情况的两倍，因此我们只需要考虑第三种情况时的复杂度即可。

考虑线段树的每一层，显然这一层的所有节点所代表的区间是互不相交，我们把这些区间按照左端点排序。

考虑询问区间 $[l, r]$，显然在排序后，与询问区间相交的所有区间的下标是连续的一段，在这一段中只有最左端的区间与最右端的区间才有可能满足条件三。

因为线段树的深度是 $O(logn)$ 的，每一层只有 $O(1)$ 个节点满足条件三，所以这个方法的时间复杂度是 $O(logn)$。

2.2.4 区间修改

在对数组 a 建立完线段树后，我们来完成最难的操作 3。

对于区间修改，我们自然不可能用单点修改的方法暴力修改每一个区间，这个时候强大而伟大的 **懒标记** 登场了！

简单来说，就是要对一个区间做修改，我们可以先把对这个区间的修改暂存在某一个祖先节点上。在需要的时候，再将影响落实到其子孙节点。

考虑要实现操作 3 ，我们在线段树的每个节点 i 上再增加一个变量 A_i ，表示给节点 i 所代表的区间中的所有数都增加了 A_i 。

现在有一个操作，需要给 $[L, R]$ 区间内的每个数都加上 x 。我们还是跟区间询问一样，先在线段树上定位这个区间，然后给这个区间打上懒标记 x ，具体来讲就是 **把 A_i 加上 x 并更新这个节点的区间和**（加上区间大小乘上 x ），然后 **向上更新它的所有父节点的信息**。

不难发现，对于线段树中的任意一点，只有当它的所有祖先节点的标记都是 0 的时候，它所记录的信息才是正确的（只要有一个祖先节点的标记不为 0，说明某一次的区间修改的影响只是暂存在了某个祖先节点上，还没有落实到儿子上）。因此无论是修改操作还是询问操作，访问到一个节点的时候，都应该将这个节点的标记下传到它的儿子中。给第 i 个节点进行标记下传相当于给它的左儿子和右儿子打上懒标记 A_i，并把 A_i 清零。

时间复杂度和区间询问一样，是 $O(logn)$ 的。

```
struct node{
    int l, r;
    int sum, lazy;
}seg[N << 2];

//对标号为 id 的节点打上 tag 的懒标记
void settag(int id, int tag){
    //先更新节点维护的信息
    seg[id].sum += (seg[id].r - seg[id].l + 1) * tag;
    //再把原来的懒标记和新增的懒标记合并
    seg[id].lazy += tag;
}

//下放懒标记
void down(int id){
    //本身没有懒标记 直接返回
    if(seg[id].lazy == 0) return;
    //对左右儿子分别打上与自己相同的标记
    settag(id << 1, seg[id].lazy);
    settag(id << 1 | 1, seg[id].lazy);
    //将自己的标记清除
    seg[id].lazy = 0;
}

void modify(int id, int ql, int qr, int val){
    int l = seg[id].l;
    int r = seg[id].r;
    if(ql > r || qr < l) return;
    //到达包含的区间，直接在该节点上打上懒标记
    if(ql <= l && r <= qr) {
        settag(id, val);
        return;
    }
    //定位区间过程中，对遇到的节点的懒标记都要先下放
    down(id);
    int mid = (l + r) >> 1;
    //继续定位左右儿子
    modify(id << 1, ql, qr, val);
    modify(id << 1 | 1, ql, qr, val);
    //更新完所有底层信息后，从下往上再更新其他祖先的信息
    up(id);
}

modify(1, ql, qr, val) //调用入口

//注意： 若要支持区间修改 则区间询问的函数要改成这样：

```

```cpp
int query(int id, int ql, int qr){
    int l = seg[id].l;
    int r = seg[id].r;
    if(ql > r || qr < l) return 0;
    if(ql <= l && r <= qr) return seg[id].sum;
    //定位区间过程中，对遇到的节点的懒标记都要先下放
    down(id);
    return query(id << 1, ql, qr) + query(id << 1 | 1, ql, qr);
}
```

2.3 总结

我们可以高度概括一下。如果你想使用线段树来维护一些信息，如果它没有区间修改，那么它要满足可以 **快速** 的完成：

- **信息与信息的合并**

如果它要支持区间修改，那么它还要额外满足可以**快速**的完成：

- **标记与标记的合并**
- **信息与标记的合并**

有时这个快速也不一定是 $O(1)$ 的，可以具体问题具体分析。

对于标记问题我们在此不再深入探讨，有兴趣和能力的同学可以思考一下，当一棵线段树上要打上多种标记时，是否只是简单的标记叠加呢？显然我们还需要考虑标记的时间顺序和标记的优先级！