线段树

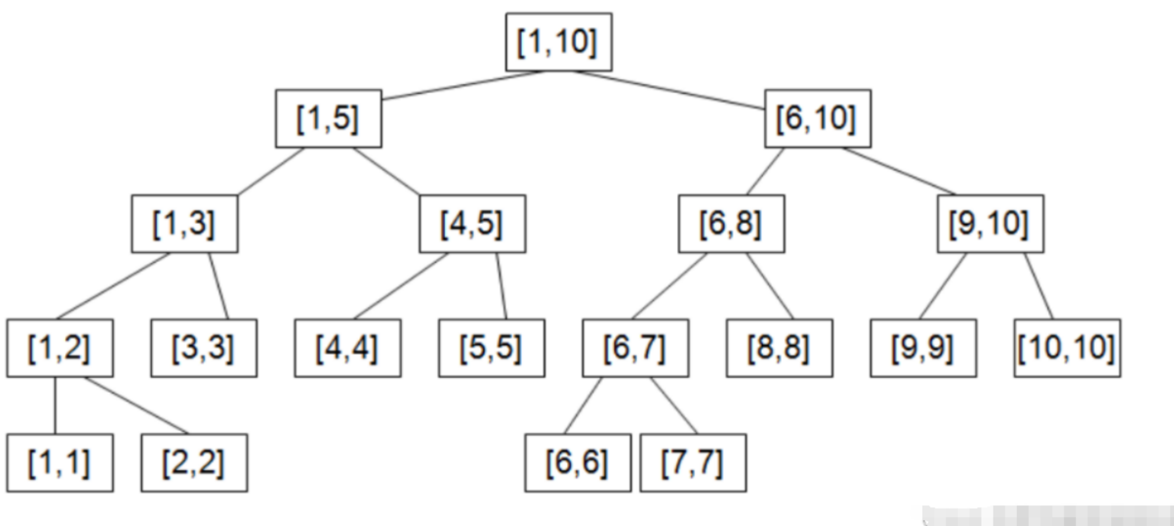
一.概念引入

线段树是一种基于分治思想的数据结构，解决一些与区间有关的问题

线段树是一种二叉树形结构，属于平衡树的一种。它将线段区间组织成树形的结构，并用每个节点来表示一条线段。

1：它是一个二叉树结构
2：它的每一个节点存一个结构体，每个结构体一般包括它的左右边界，还有它的权值一类的，还要看题具体定
3：它用到的思想为二分
4：它的结构如下图

一棵[1,10]的线段树的结构如图所示：



PS : 如果是**区间修改 + 单点求值** 或者 **单点修改 + 区间求值**
　　我绝对选择树状数组，又快代码又短，所以以下讲解全部是基于 **区间修改 + 区间求值** 去讲的

二. 分析 + 建树

可以看到，线段树的每个节点表示了一条线段，线段树的根节点表示了所要处理的最大线段区间，而叶节点则表示了形如[a,a]的单位区间。对于每个非叶节点（包括根节点）所表示的区间[s,t]，令m = (s + t) / 2，则其左儿子节点表示区间[s,m]，右儿子节点表示区间[m+1,t]。

如果你要表示线段的和，那么最上面的根节点的权值表示的是这个线段 [1 ~ 5] 的和。根的两个儿子分别表示这个线段中 [1~3]的和，与[4~5]的和，以此类推。

然后我们还可以的到一个性质：**节点 i的权值 = 左儿子权值 + 右儿子权值。**

根据这个思路，我们就可以建树了，我们设一个结构体 数组 **tree**， **tree[i].left**与 **tree[i].right** 分别表示这个点代表的线段的左右区间边缘值，**tree[i].sum** 表示这个节点表示的线段和

这样建树的好处在于，对于每条要处理的线段，可以类似"二分"的进入线段树中处理，使得时间复杂度在O(logN)量级，这也是线段树之所以高效的原因。

```
#define M 100005
struct node{
    int left,right,sum,mmax,mmin; // 左区间  右区间  区间和
}tree[M<<2];
// 数组开原始长度的4倍
// 线段树就是用空间换时间去完成高效的区间处理，所以警惕MLE错误
```

我们知道，一颗从1开始编号的二叉树，结点 i 的左儿子和右儿子编号分别是 $2 \times i$ 和 $2 \times i + 1$。

再根据刚才的性质，得到式子，第x个结点的区间和为：

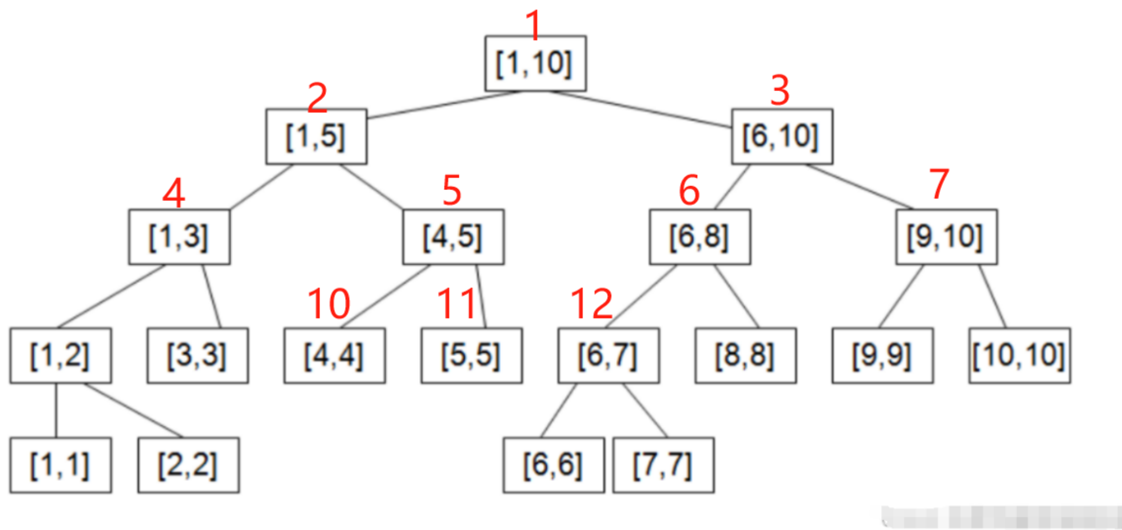
$$tree[x].sum = tree[2 * x].sum + tree[2 * x + 1].sum;$$

就可以建一颗线段树了！代码暂时如下：

```
inline void build(int x , int L , int R){ // 结点 左区间 右区间  ---递归建树
    tree[x].left = L;
    tree[x].right = R;
    if(L==R){//如果这个节点是叶子节点
        tree[x].sum = a[R];
        return;
    }
    int mid = (L+R)/2;
    build(x*2 , L , mid);
    build(x*2 + 1 , mid+1 , R);
    tree[x].sum = tree[2*x].sum + tree[2*x+1].sum;
}
```

三. 常规方法

①区间最值



比如说我们要取出某个区间的最值，比如求4-7这个区间的最值，该如何完成呢
朴素的做法非常简单

```
int mmax = INT_MIN;
for(int i=4 ; i<=7 ; i++){
    mmax = max(mmax , a[i]);
}
```

很显然，大数据必然超时
这里，我们可以使用线段树，利用二分思想去降低时间复杂度和题目难度
非常容易就能想到，例如怎么得到区间1~10的最值呢?
拿到左儿子1~5的最值 和 右儿子6~10 这2个区间的最值，然后2者再取最值即可

我们想求出第 **4~ 7** 项的最小值。

1. 我们先从根节点**[1号点]**开始查询，发现它的左儿子**[2号点]**1-5这个区间和答案区间 4 ~ 7 有交集，那么我们跑到左儿子**[2号点]**这个区间，二分。
2. 我们发现这个区间的右儿子**[5号点]** 4 ~5 被完全包括在答案区间 4 ~ 7 这个区间里面，那就把这个区间的最值 返回。而左儿子**[4号点]** 1 ~ 3 和 答案区间 4 ~ 7 没有任何交集，直接pass
3. 回到根节点**[1号点]**的右儿子**[3号点]** 6 ~10 ，发现和答案区间 4 ~ 7 有交集，继续二分
4. **[3号点]** 的右孩子**[7号点]** 9 ~ 10 和 答案区间 4 ~ 7 没有任何交集，直接pass
5. **[3号点]** 的左孩子**[6号点]** 6 ~ 8 和 答案区间 4 ~ 7 有交集，二分
6. **[6号点]** 的右儿子**[13号点]** 8 ~8 和 答案区间 4 ~ 7 没有任何交集，直接pass
7. **[6号点]** 的左儿子**[12号点]** 6 ~7 被完全包括在答案区间 4 ~ 7 这个区间里面，那就把这个区间的最值返回。

PS : 第 **4~ 7** 项的最值 = min(**[5号点]** 4 ~5的最值 ， **[12号点]** 6 ~7 的最值)

我们总结一下，线段树的区间最值方法：

1.如果这个区间被完全包括在目标区间里面，直接返回这个区间的最值

2.如果这个区间的左儿子和目标区间有交集，那么搜索左儿子

3.如果这个区间的右儿子和目标区间有交集，那么搜索右儿子

一 . 建树

```
#define M 100005
struct node{
    int left,right,mmax,mmin; // 左区间  右区间  最大值  最小值
}tree[M<<2];
// 数组开原始长度的4倍
// 线段树就是用空间换时间去完成高效的区间处理，所以警惕MLE错误
```

结合之前的建树方案，这里的结点最值更新就会很简单

$$tree[x].mmax = max(tree[2*x].mmax, tree[2*x+1].mmax);$$

具体如下:

```
inline void build(int i,int l,int r){
    tree[i] = {l,r,0,0};
    if(l==r){
        tree[i].mmax = tree[i].mmin = a[l]; // 当l==r即当前元素本身
        return;
    }
    int mid = (tree[i].l + tree[i].r)/2;
    build(2*i , l , mid);
    build(2*i+1 , mid+1 , r);
    tree[i].mmax = max(tree[2*i].mmax , tree[2*i+1].mmax);
    tree[i].mmin = min(tree[2*i].mmin , tree[2*i+1].mmin);
}
```

二. 区间查询

思路和上述内容一致
有2种雏形，可根据题目自由选择
第一种，趋向于 得到最终的值[只求个最大值或最小值]

```
inline int query(int i,int l,int r){
    if( tree[i].l >= l and tree[i].r <= r ){
        return tree[i].MAX;
    }
    int mid = (tree[i].l + tree[i].r) >> 1;
    int ans = LONG_LONG_MIN;
    if(mid>=l) ans = max(ans , query(2*i,l,r));
    if(mid<r) ans = max(ans , query(2*i+1,l,r));
    return ans;
}
```

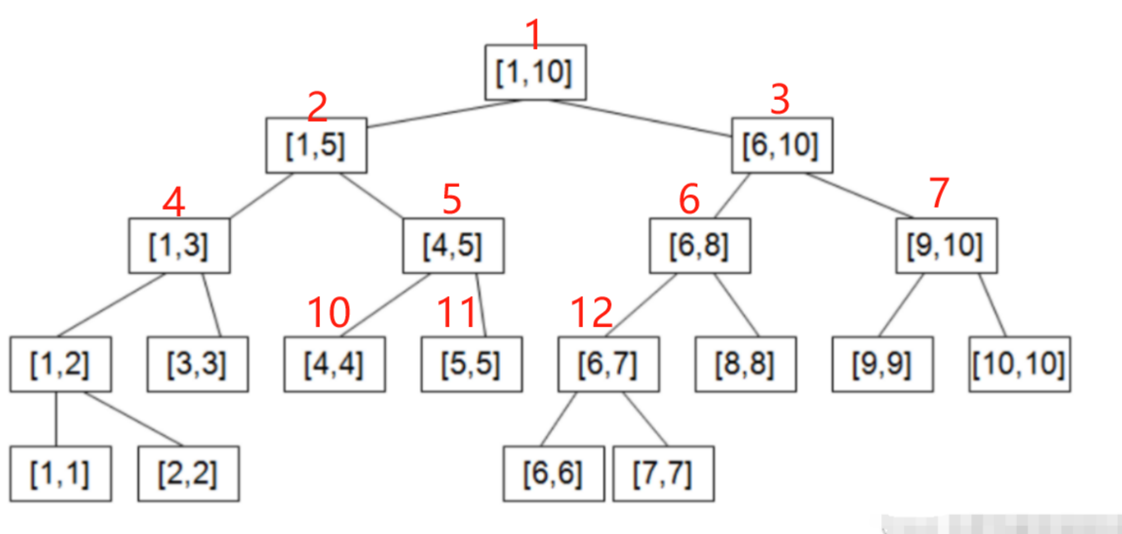
第二种， 趋向于 更新当前的最值[同时都求]
此方案可以不需要返回值，直接在mmax和mmin的基础上不断更新最大值/最小值即可

```
主函数 ： mmax = INT_MIN , mmin = INT_MAX;
inline void query(int i,int l,int r){
    if( tree[i].l >= l and tree[i].r <= r ){
        mmax = max(mmax , tree[i].MAX);
        mmin = min(mmin , tree[i].MIN);
        return;
    }
    int mid = (tree[i].l+tree[i].r) >> 1;
    if(mid>=l) query(2*i,l,r);
    if(mid<r) query(2*i+1,l,r);
}
```

②区间求和 + 单点求值

----- 【PS:这个方法需要更新】

比如你想求出一个 1 ~ 100区间中， 4 ~ 67 这些元素的和，你会怎么做？朴素的做法是 `for(i=4;i<=67;i++) sum+=a[i]`，这样固然好，但是算得太慢了。



我们想求出第 **4~ 7** 项的和。

1. 我们先从根节点**[1号点]**开始查询，发现它的左儿子**[2号点]**1-5这个区间和答案区间 4 ~ 7 有交集，那么我们跑到左儿子**[2号点]**这个区间，二分。

　我们发现这个区间的右儿子**[5号点]** 4 ~5 被完全包括在答案区间 4 ~ 7 这个区间里面，那就把这个点的值 返回。而左儿子**[4号点]** 1 ~ 3 和 答案区间 4 ~ 7 没有任何交集，直接pass

2. 回到根节点**[1号点]**的右儿子**[3号点]** 6 ~10 ，发现和答案区间 4 ~ 7 有交集，继续二分
3. **[3号点]** 的右孩子**[7号点]** 9 ~ 10 和 答案区间 4 ~ 7 没有任何交集，直接pass
4. **[3号点]** 的左孩子**[6号点]** 6 ~ 8 和 答案区间 4 ~ 7 有交集，二分
5. **[6号点]** 的右儿子**[13号点]** 8 ~8 和 答案区间 4 ~ 7 没有任何交集，直接pass
6. **[6号点]** 的左儿子**[12号点]** 6 ~7 被完全包括在答案区间 4 ~ 7 这个区间里面，那就把这个点的值 返回。

PS : 第 **4~ 7** 项的和 = **[5号点]** 4 ~5 + **[12号点]** 6 ~7

我们总结一下，线段树的区间和[单点]方法：

1.如果这个区间被完全包括在目标区间里面，直接返回这个区间的值

2.如果这个区间的左儿子和目标区间有交集，那么搜索左儿子

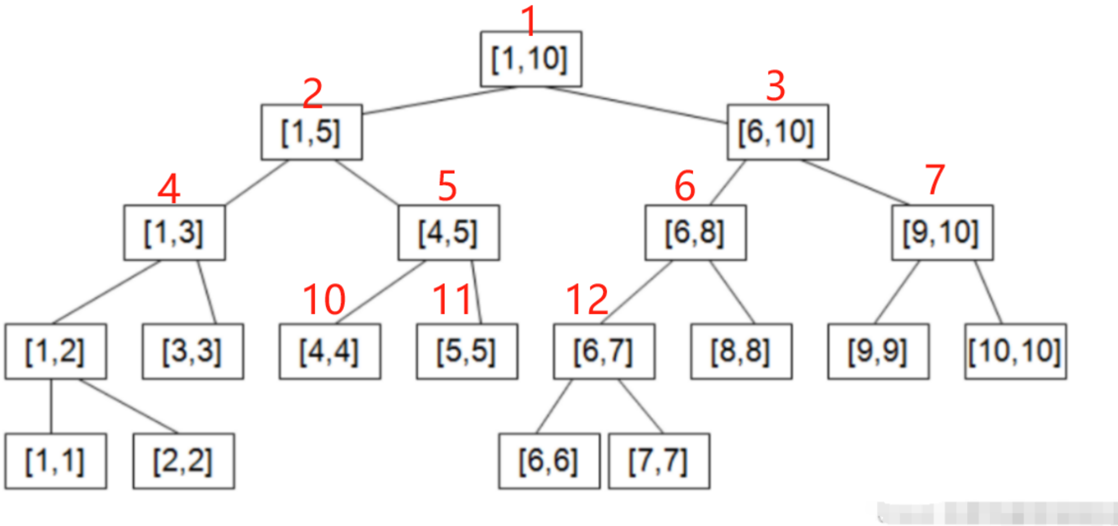
3.如果这个区间的右儿子和目标区间有交集，那么搜索右儿子

```
方法1：
inline int search(int i,int L,int R){
    //如果结点区间被完全包含在搜索的目标区间里面，直接返回这个区间的值
    if(tree[i].left>=L && tree[i].right<=R){
        return tree[i].sum;
    }
    if(tree[i].right<L || tree[i].left>R)  return 0;//如果这个区间和目标区间毫不相干，返回0
    int s=0;
    if(tree[i*2].right>=L)  s+=search(i*2,L,R);//如果区间左儿子和目标区间有交集，就搜索左儿子
    if(tree[i*2+1].left<=R)  s+=search(i*2+1,L,R);//如果区间右儿子和目标区间有交集，就搜索右儿子
    return s;
}
```

```
方法2：
inline int query(int i,int L,int R){
    if( tree[i].left >= L and tree[i].right <= R ){//如果这个区间被完全包括在目标区间里面，直接返回这个区间的值
        return tree[i].sum;
    }
    int mid = (tree[i].left + tree[i].right)/2;
    int ans = 0;
    if(mid>=L) ans += query(2*i,L,R);//如果这个区间的左儿子和目标区间又交集，那么搜索左儿子
    if(mid<R) ans += query(2*i+1,L,R);//如果这个区间的右儿子和目标区间又交集，那么搜索右儿子
    return ans;
}
```

关于那几个if的条件一定要看清楚，最好背下来，以防考场上脑抽推错。

③懒标记



怎么记录这个标记呢？我们需要记录一个"懒标记" lazytage，来记录这个区间

什么是**懒标记**呢，我们知道如果区间修改，需要将L,R范围内的所有数字都要增加/减少k，那这样又会导致时间复杂度的提升，而后续题目在进行查询的时候，有时候是查询不到刚刚修改过的L,R范围的，所以这个区间修改在没查询到的情况下，显得非常没用。

所以就类似于交作业一样，今天老师布置了作业，但是明天不收，那是不是可以先不写，把作业做一个标记，
什么时候发现老师要收作业了，那就把当前收的那份作业写了，提交了，这就是懒标记。

区间修改的时候，我们按照如下原则：

1. 如果当前区间被完全覆盖在目标区间里，那么这个区间所有的节点都会增加看，这个区间的 sum + k * (tree[i].right - tree[i].left+1)，其中的tree[i].right - tree[i].left+1 也就是记录的当前节点的区域范围长度
2. 如果没有完全覆盖，则先下传懒标记
3. 如果这个区间的左儿子和目标区间有交集，那么搜索左儿子
4. 如果这个区间的右儿子和目标区间有交集，那么搜索右儿子

那么我们应该怎么去下传懒标记呢？这时候 pushdown 的作用就显现出来了。
其中的pushdown方法，就是把自己的lazytage归零，并给自己的左右儿子加上，让自己的儿子加上k*(r-l+1)，并且让左右儿子去更新自己的懒标记。

```
// 懒标记下移方法
void pushdown(int i){
    if(tree[i].lazy==0) return; // 没有多余计算的直接pass
    tree[2*i].sum += (tree[2*i].right-tree[2*i].left+1)*tree[i].lazy;
    tree[2*i+1].sum += (tree[2*i+1].right-tree[2*i+1].left+1)*tree[i].lazy;
    tree[2*i].lazy += tree[i].lazy;
    tree[2*i+1].lazy += tree[i].lazy;
    tree[i].lazy = 0; // 记得要把当前i的懒标记清0
}
```

有了此方法，时间复杂度会有明显的下降趋势

④ 正式代码

由于懒标记的出现，上述方法都需要更新，以下为最终版本

一. 线段树结构体的创建

```
#define M 100005
#define int long long
struct node{
    int left,right,lazy,sum;
}tree[M<<2];
```

二. 建树

```
// 建树
void build(int i,int L,int R){
    tree[i].left = L ;
    tree[i].right = R ;
    tree[i].sum = 0 ;
    if(L==R){
        tree[i].sum = a[L];
        return;
    }
    int mid = (tree[i].left + tree[i].right)/2;
    build(i*2,L,mid);
    build(i*2+1,mid+1,R);
    tree[i].sum = tree[i*2].sum + tree[i*2+1].sum;
}
```

三. 懒标记下移方法pushdown

```
void pushdown(int i){
    if(tree[i].lazy==0) return; // 没有多余计算的直接pass
    tree[2*i].sum += (tree[2*i].right-tree[2*i].left+1)*tree[i].lazy;
    tree[2*i+1].sum += (tree[2*i+1].right-tree[2*i+1].left+1)*tree[i].lazy;
    tree[2*i].lazy += tree[i].lazy;
    tree[2*i+1].lazy += tree[i].lazy;
    tree[i].lazy = 0; // 记得要把当前i的懒标记清0
}
```

四. 区间/ 单点修改

灵活一点，将左区间和右区间和并即为单点查询
PS : 此方法基本要从根节点1开始调用，即i=1

```
void update(int i,int L,int R,int w){
    if(tree[i].left>=L and tree[i].right<=R){// 如果节点范围在L,R查询范围内
        tree[i].sum += w*(tree[i].right-tree[i].left+1); // 更新当前节点i的sum值
        tree[i].lazy += w;// 懒标记更新，但是暂时i和子区间的sum先不更新，不着急
        return;
    }
    pushdown(i);
    int mid = (tree[i].left + tree[i].right)/2;
    if(mid>=L) update(2*i,L,R,w);//如果这个区间的左儿子和目标区间又交集，那么搜索左儿子
    if(mid<R) update(2*i+1,L,R,w);//如果这个区间的右儿子和目标区间又交集，那么搜索右儿子
    tree[i].sum = tree[2*i].sum + tree[2*i+1].sum; // 上述递归结束后，更新节点i
}
```

五. 区间/单点查询

PS : 此方法基本要从根节点1开始调用，即i=1

```
int query(int i,int L,int R){
    if(tree[i].left>=L and tree[i].right<=R){ // 如果节点范围在L,R查询范围内
        return tree[i].sum;
    }
    pushdown(i);// 看看有没有遗留的lazy标记【即如果有需要更新的点，让它强制更新】
    int mid = (tree[i].left+tree[i].right)/2;
    int ans = 0;
    if(mid>=L) ans += query(2*i,L,R);//如果这个区间的左儿子和目标区间又交集，那么搜索左儿子
    if(mid<R) ans += query(2*i+1,L,R);//如果这个区间的右儿子和目标区间又交集，那么搜索右儿子
    return ans;
}
```

四. 常见题目

具体见题库