线段树详解 （原理，实现与应用）

线段树详解

By 岩之痕

目录:

一：综述

假设有编号从1到n的n个点，每个点都存了一些信息，用[L,R]表示下标从L到R的这些点。
线段树的用处就是，对编号连续的一些点进行修改或者统计操作，修改和统计的复杂度都是O(log2(n)).

线段树的原理，就是，将[1,n]分解成若干特定的子区间(数量不超过4*n),然后，将每个区间[L,R]都分解为少量特定的子区间，通过对这些少量子区间的修改或者统计，来实现快速对[L,R]的修改或者统计。

由此看出，用线段树统计的东西，必须符合**区间加法**，否则，不可能通过分成的子区间来得到[L,R]的统计结果。

符合区间加法的例子：
数字之和——总数字之和 = 左区间数字之和 + 右区间数字之和
最大公因数(GCD)——总GCD = gcd(左区间GCD , 右区间GCD);
最大值——总最大值=max(左区间最大值，右区间最大值)
不符合区间加法的例子：
众数——只知道左右区间的众数，没法求总区间的众数
01序列的最长连续零——只知道左右区间的最长连续零，没法知道总的最长连续零

一个问题，只要能化成对一些连续点的修改和统计问题，基本就可以用线段树来解决了，具体怎么转化在第六节会讲。
由于点的信息可以千变万化，所以线段树是一种非常灵活的数据结构，可以做的题的类型特别多，只要会转化。

线段树当然是可以维护线段信息的，因为线段信息也是可以转换成用点来表达的（每个点代表一条线段）。
所以在以下对结构的讨论中，都是对点的讨论，线段和点的对应关系在第七节扫描线中会讲。

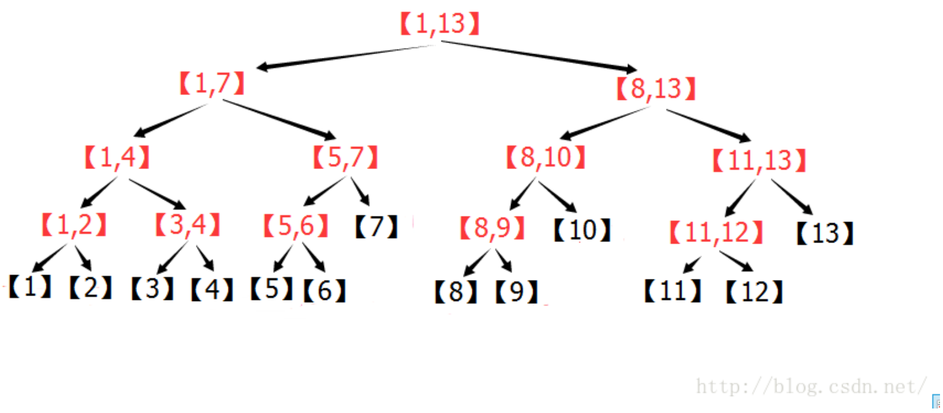
本文二到五节是讲对线段树操作的原理和实现。
六到八节介绍了线段树解题模型，以及一些例题。

初学者可以先看这篇文章： 线段树从零开始

二：原理

（注：由于线段树的每个节点代表一个区间，以下叙述中不区分节点和区间，只是根据语境需要，选择合适的词）
线段树本质上是维护下标为1,2,..,n的n个按顺序排列的数的信息，所以，其实是“点树”，是维护n的点的信息，至于每个点的数据的含义可以有很多，
在对线段操作的线段树中，每个点代表一条线段，在用线段树维护数列信息的时候，每个点代表一个数，但本质上都是每个点代表一个数。以下，在讨论线段树的时候，区间[L,R]指的是下标从L到R的这(R-L+1)个数，而不是指一条连续的线段。只是有时候这些数代表实际上一条线段的统计结果而已。

线段树是将每个区间[L,R]分解成[L,M]和[M+1,R] (其中M=(L+R)/2 这里的除法是整数除法，即对结果下取整)直到
开始时是区间[1,n] ,通过递归来逐步分解，假设根的高度为1的话，树的最大高度为 $\lfloor \log_2(n-1) \rfloor + 2$ （n>1）。
线段树对于每个n的分解是唯一的，所以n相同的线段树结构相同，这也是实现可持久化线段树的基础。
下图展示了区间[1,13]的分解过程：



上图中，每个区间都是一个节点，每个节点存自己对应的区间的统计信息。

(1)线段树的点修改：

假设要修改[5]的值，可以发现，每层只有一个节点包含[5],所以修改了[5]之后，只需要每层更新一个节点就可以
线段树每个节点的信息都是正确的，所以修改次数的最大值为层数 $\lfloor \log_2(n-1) \rfloor + 2$ 。
复杂度O(log2(n))

(2)线段树的区间查询：

线段树能快速进行区间查询的基础是下面的定理：

定理：n>=3时，一个[1,n]的线段树可以将[1,n]的任意子区间[L,R]分解为不超过 $2\lfloor \log_2(n-1) \rfloor$ 个子区间。

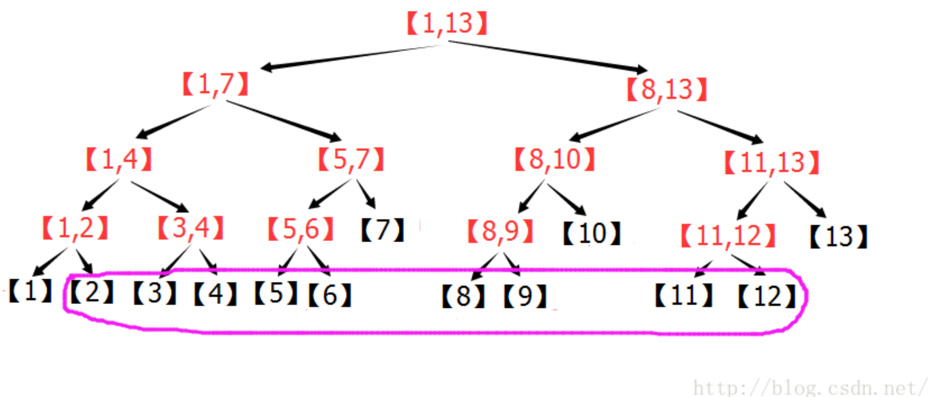
这样，在查询[L,R]的统计值的时候，只需要访问不超过 $2\lfloor \log_2(n-1) \rfloor$ 个节点，就可以获得[L,R]的统计信息，实现了O(log2(n))的区间查询。

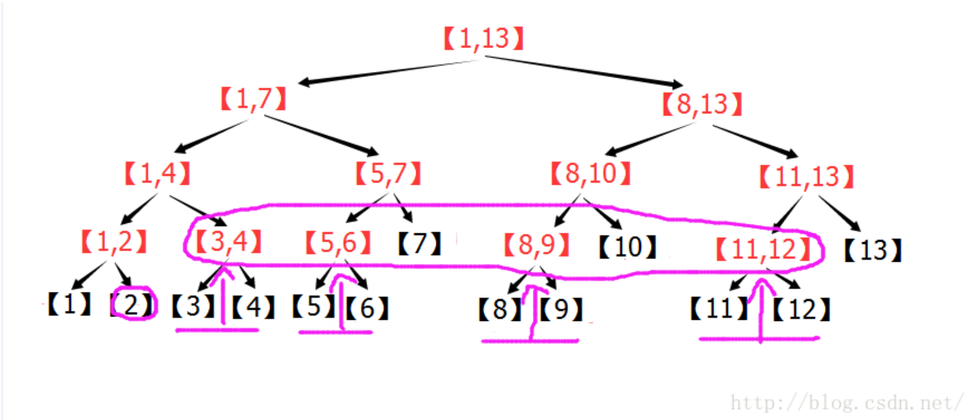
下面给出证明：

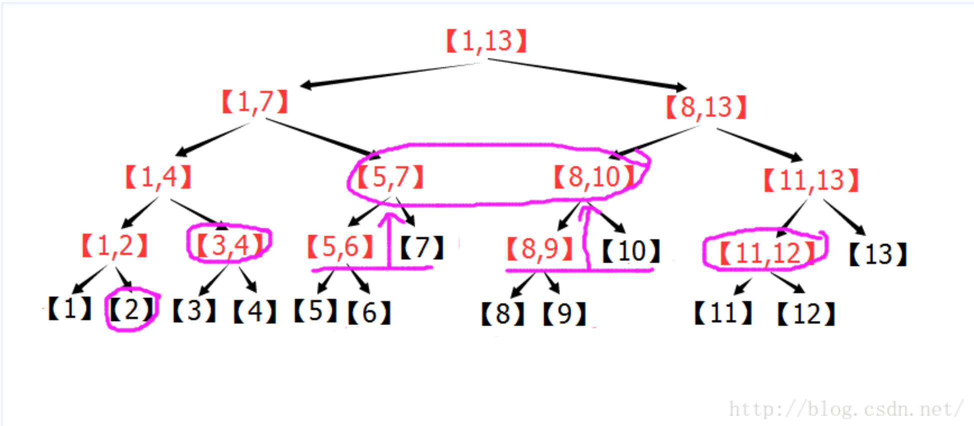
一个字“赞”。看了一下其他讲解，都有错误
--PrincessYR✨~
2. Re:线段树详解 （原理，实现与应用）
Orz
太牛皮了! 本苟蒻竟然全看懂了!
--星耀--新世界
3. Re:线段树详解 （原理，实现与应用）
%%%%%%
--⇜黑光⇝明暗⇝
4. Re:线段树详解 （原理，实现与应用）
nb
--观稳769
5. Re:线段树详解 （原理，实现与应用）
牛皮
--坤sir

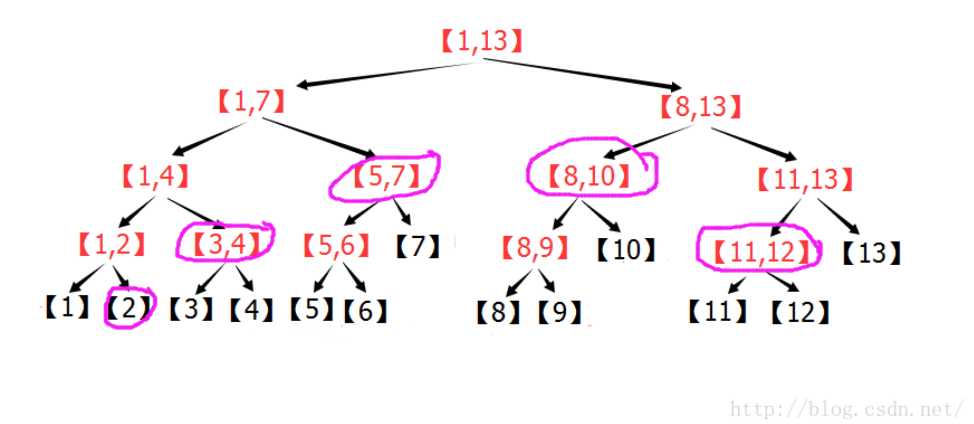
(2.1)先给出一个粗略的证明（结合下图）：
先考虑树的最下层，将所有在区间[L,R]内的点选中，然后，若相邻的点的直接父节点是同一个，那么就用这个父节点代替这两个节点（父节点在上一层）。这样操作之后，本层最多剩下两个节点。若最左侧被选中的节点是它父节点的右子树，那么这个节点会被剩下。若最右侧被选中的节点是它的父节点的左子树，那么这个节点会被剩下。中间的所有节点都被父节点取代。
对最下层处理完之后，考虑它的上一层，继续进行同样的处理，可以发现，每一层最多留下2个节点，其余的节点升往上一层，这样可以说明分割成的区间（节点）个数是大概是树高的两倍左右。

下图为n=13的线段树，区间[2,12]，按照上面的叙述进行操作的过程图：









由图可以看出：在n=13的线段树中，[2,12]=[2] + [3,4] + [5,7] + [8,10] + [11,12] 。

(2.2)然后给出正式一点的证明：

定理：n>=3时，一个[1,n]的线段树可以将[1,n]的任意子区间[L,R]分解为不超过 $2\lfloor \log_2(n-1) \rfloor$ 个子区间。

用数学归纳法，证明上面的定理：
首先,n=3,4,5时，用穷举法不难证明定理成立。
假设对于n= 3,4,5,...,k-1上式都成立，下面来证明对于n=k (k>=6)成立：
分为4种情况来证明：

情况一：[L,R]包含根节点(L=1且R=n)，此时，[L,R]被分解为了一个节点，定理成立。

情况二：[L,R]包含根节点的左子节点，此时[L,R]一定不包含根的右子节点（因为如果包含，就可以合并左右子节

用根节点替代，此时就是情况一）。这时，以右子节点为根的这个树的元素个数为 $\left\lfloor \frac{k}{2} \right\rfloor \ge 3$ 。
[L,R]分成的子区间由两部分组成：
一：根的左子结点，区间数为1
二：以根的右子节点为根的树中，进行区间查询，这个可以递归使用本定理。

由归纳假设可得，[L,R]一共被分成了 $1+2\left\lfloor \log_2\left(\left\lfloor \frac{k}{2} \right\rfloor -1\right)\right\rfloor$ 个区间。

情况三：跟情况二对称，不一样的是，以根的左子节点为根的树的元素个数为 $\left\lfloor \frac{k+1}{2} \right\rfloor \ge 3$ 。
[L,R]一共被分成了 $1+2\left\lfloor \log_2\left(\left\lfloor \frac{k+1}{2} \right\rfloor -1\right)\right\rfloor$ 个区间。
从公式可以看出，情况二的区间数小于等于情况三的区间数，于是只需要证明情况三的区间数符合条件就行了。

$$
\begin{aligned}
&1+2\left\lfloor \log_2\left(\left\lfloor \frac{k+1}{2} \right\rfloor -1\right)\right\rfloor \\
&=1+2\left\lfloor \log_2\left(\left\lfloor \frac{k-1}{2} \right\rfloor\right)\right\rfloor \\
&\le 1+2\left\lfloor \log_2\left(\frac{k-1}{2}\right)\right\rfloor \\
&=1+2\left\lfloor \log_2(k-1)-1\right\rfloor \\
&=2\left\lfloor \log_2(k-1)\right\rfloor -1 < 2\left\lfloor \log_2(k-1)\right\rfloor
\end{aligned}
$$

于是，情况二和情况三定理成立。

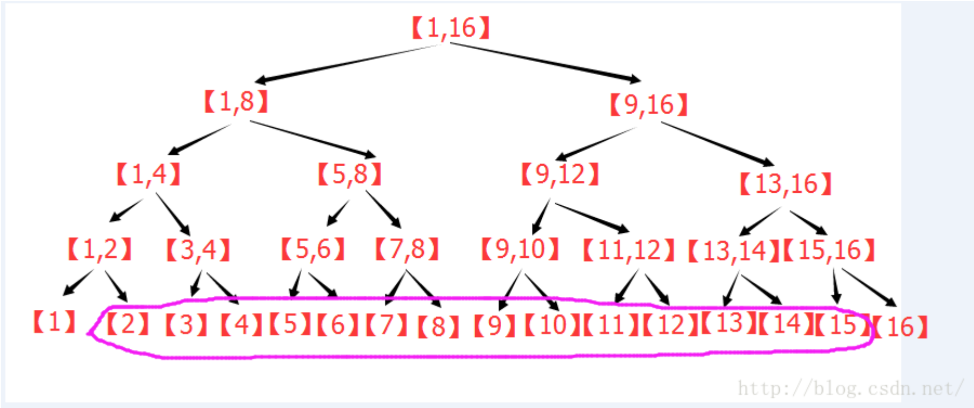
情况四：[L,R]不包括根节点以及根节点的左右子节点。

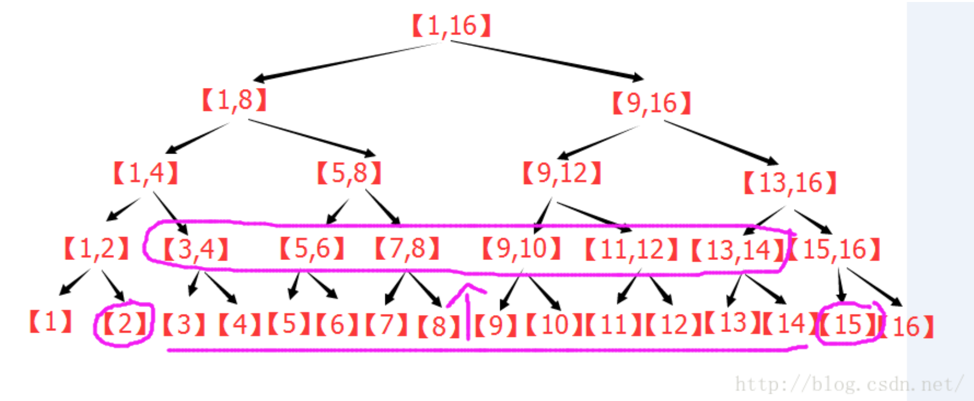
于是，剩下的 $\lfloor \log_2(k-1) \rfloor$ 层，每层最多两个节点（参考粗略证明中的内容）。

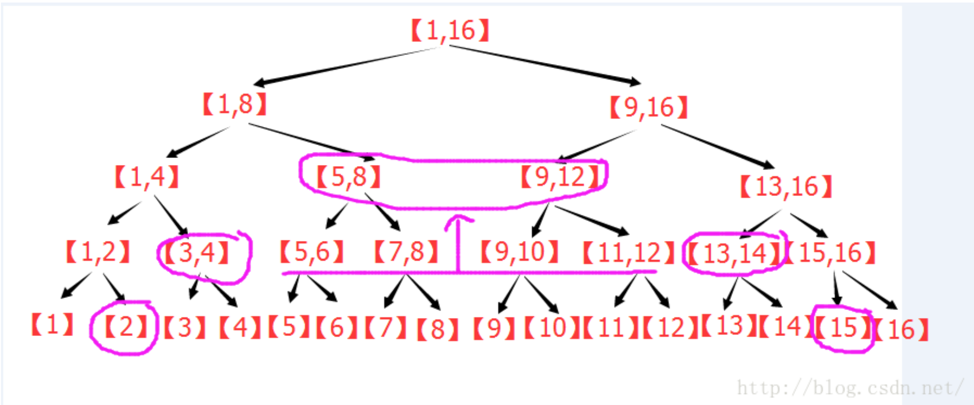
于是[L,R]最多被分解成了 $2\lfloor \log_2(k-1) \rfloor$ 个区间，定理成立。

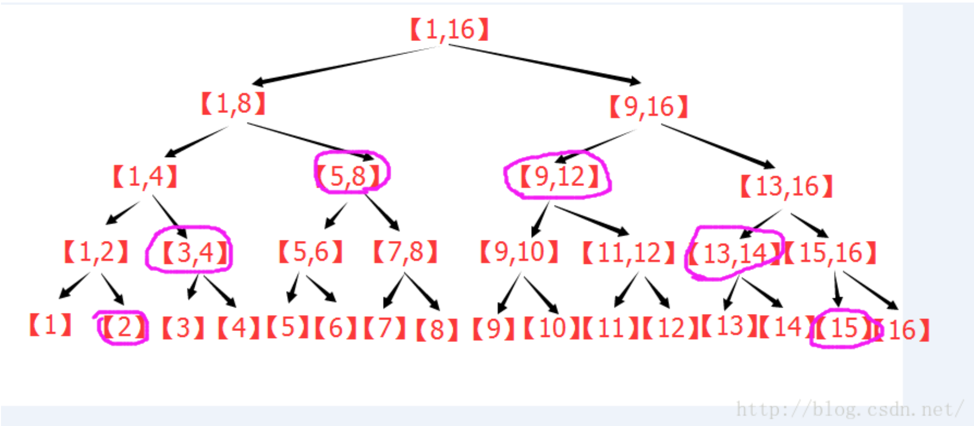
上面只证明了 $2\lfloor \log_2(k-1) \rfloor$ 是上界，但是，其实它是最小上界。
n=3,4时，有很多组区间的分解可以达到最小上界。

当n>4时，当且仅当n=2^t (t>=3),L=2,R=2^t -1 时，区间[L,R]的分解可以达到最小上界 $2\lfloor \log_2(k-1) \rfloor$ 。
就不证明了，有兴趣可以自己去证明。
下图是n=16 , L=2 , R=15 时的操作图，此图展示了达到最小上界的树的结构。









(3)线段树的区间修改：

线段树的区间修改也是将区间分成子区间，但是要加一个标记，称作懒惰标记。

标记的含义：

本节点的统计信息已经根据标记更新过了，但是本节点的子节点仍需要进行更新。

即，如果要给一个区间的所有值都加上1，那么，实际上并没有给这个区间的所有值都加上1，而是打个标记，记下来，这个节点所包含的区间需要加1.打上标记后，要根据标记更新本节点的统计信息，比如，如果本节点维

护的是区间和，而本节点包含5个数，那么，打上+1的标记之后，要给本节点维护的和+5。这是向下延迟修改，但是向上显示的信息是修改以后的信息，所以查询的时候可以得到正确的结果。有的标记之间会相互影响，所以比较简单的做法是，每递归到一个区间，首先下推标记（若本节点有标记，就下推标记），然后再打上新的标记，这样仍然每个区间操作的复杂度是O(log2(n))。

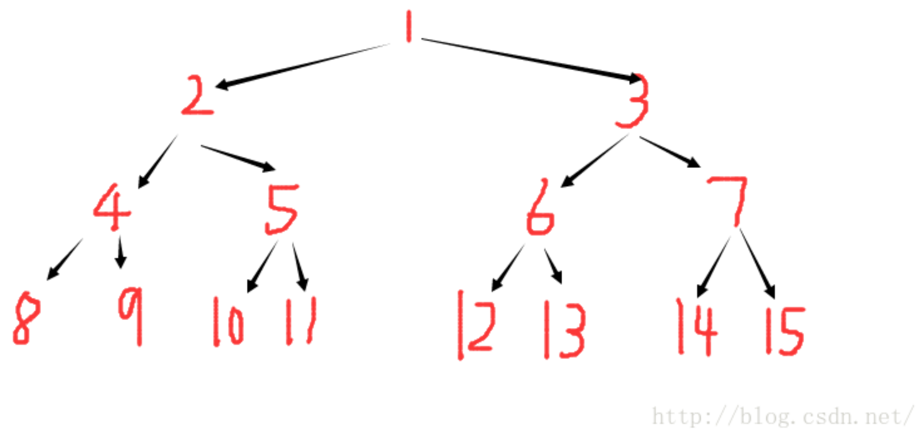
标记有**相对标记**和**绝对标记**之分：
相对标记是将区间的所有数+a之类的操作，标记之间可以共存，跟打标记的顺序无关（跟顺序无关才是重点）。
所以，可以在区间修改的时候不下推标记，留到查询的时候再下推。
注意：如果区间修改时不下推标记，那么PushUp函数中，必须考虑本节点的标记。
而如果所有操作都下推标记，那么PushUp函数可以不考虑本节点的标记，因为本节点的标记一定已经被下推了（也就是对本节点无效了）
绝对标记是将区间的所有数变成a之类的操作，打标记的顺序直接影响结果，
所以这种标记在区间修改的时候必须下推旧标记，不然会出错。

注意，有多个标记的时候，标记下推的顺序也很重要，错误的下推顺序可能会导致错误。

之所以要区分两种标记，是因为**非递归线段树**只能维护相对标记。
因为非递归线段树是自底向上直接修改分成的每个子区间，所以根本做不到在区间修改的时候下推标记。
非递归线段树一般不下推标记，而是自下而上求答案的过程中，根据标记更新答案。

(4)线段树的存储结构：

线段树是用数组来模拟树形结构，对于每一个节点R ,左子节点为 2*R (一般写作R<<1)右子节点为 2*R+1（一般写作R<<1|1）
然后以1为根节点，所以，整体的统计信息是存在节点1中的。
这么表示的原因看下图就很明白了，左子树的节点标号都是根节点的两倍，右子树的节点标号都是左子树+1：



线段树需要的数组元素个数是： $2^{\lceil \log_2(n) \rceil + 1}$,一般都开4倍空间，比如： int A[n<<2];

三：递归实现

以下以维护数列区间和的线段树为例，演示最基本的线段树代码。

(0)定义：

```
#define maxn 100007  //元素总个数
#define ls l,m,rt<<1
#define rs m+1,r,rt<<1|1
int Sum[maxn<<2],Add[maxn<<2];//Sum求和，Add为懒惰标记
int A[maxn],n;//存原数组数据下标[1,n]
```

(1)建树：

```
//PushUp函数更新节点信息 ，这里是求和
void PushUp(int rt){Sum[rt]=Sum[rt<<1]+Sum[rt<<1|1];}
//Build函数建树
void Build(int l,int r,int rt){ //l,r表示当前节点区间，rt表示当前节点编号
```

```
    if(l==r) {//若到达叶节点
        Sum[rt]=A[l];//储存数组值
        return;
    }
    int m=(l+r)>>1;
    //左右递归
    Build(l,m,rt<<1);
    Build(m+1,r,rt<<1|1);
    //更新信息
    PushUp(rt);
}
```

(2)点修改:

假设A[L]+=C:

```
void Update(int L,int C,int l,int r,int rt){//l,r表示当前节点区间，rt表示当前节点编号
    if(l==r){//到叶节点，修改
        Sum[rt]+=C;
        return;
    }
    int m=(l+r)>>1;
    //根据条件判断往左子树调用还是往右
    if(L <= m) Update(L,C,l,m,rt<<1);
    else       Update(L,C,m+1,r,rt<<1|1);
    PushUp(rt);//子节点更新了，所以本节点也需要更新信息
}
```

(3)区间修改:

假设A[L,R]+=C

```
void Update(int L,int R,int C,int l,int r,int rt){//L,R表示操作区间，l,r表示当前节点区间，rt表示当前节点编号
    if(L <= l && r <= R){//如果本区间完全在操作区间[L,R]以内
        Sum[rt]+=C*(r-l+1);//更新数字和，向上保持正确
        Add[rt]+=C;//增加Add标记，表示本区间的Sum正确，子区间的Sum仍需要根据Add的值来调整
        return ;
    }
    int m=(l+r)>>1;
    PushDown(rt,m-l+1,r-m);//下推标记
    //这里判断左右子树跟[L,R]有无交集，有交集才递归
    if(L <= m) Update(L,R,C,l,m,rt<<1);
    if(R >  m) Update(L,R,C,m+1,r,rt<<1|1);
    PushUp(rt);//更新本节点信息
}
```

(4)区间查询:

询问A[L,R]的和
首先是下推标记的函数:

```
void PushDown(int rt,int ln,int rn){
    //ln,rn为左子树，右子树的数字数量。
    if(Add[rt]){
        //下推标记
        Add[rt<<1]+=Add[rt];
        Add[rt<<1|1]+=Add[rt];
        //修改子节点的Sum使之与对应的Add相对应
        Sum[rt<<1]+=Add[rt]*ln;
        Sum[rt<<1|1]+=Add[rt]*rn;
        //清除本节点标记
        Add[rt]=0;
    }
}
```

然后是区间查询的函数:

```
int Query(int L,int R,int l,int r,int rt){//L,R表示操作区间，l,r表示当前节点区间，rt表示当前节点编号
    if(L <= l && r <= R){
        //在区间内，直接返回
        return Sum[rt];
    }
    int m=(l+r)>>1;
    //下推标记，否则Sum可能不正确
    PushDown(rt,m-l+1,r-m);

    //累计答案
    int ANS=0;
    if(L <= m) ANS+=Query(L,R,l,m,rt<<1);
    if(R >  m) ANS+=Query(L,R,m+1,r,rt<<1|1);
    return ANS;
}
```

(5)函数调用:

```
//建树
Build(1,n,1);
//点修改
Update(L,C,1,n,1);
//区间修改
Update(L,R,C,1,n,1);
//区间查询
int ANS=Query(L,R,1,n,1);
```

感谢几位网友指出了我的错误。
我说相对标记在Update时可以不下推，这一点是对的，但是原来的代码是错误的。
因为原来的代码中，PushUP函数是没有考虑本节点的Add值的，如果Update时下推标记，那么PushUp的时候，节点的Add值一定为零，所以不需要考虑Add。
但是，如果Update时暂时不下推标记的话，那么PushUp函数就必须考虑本节点的Add值，否则会导致错误。
为了简便，上面函数中，PushUp函数没有考虑Add标记。所以无论是相对标记还是绝对标记，在更新信息的时候，
到达的每个节点都必须调用PushDown函数来下推标记，另外，代码中，点修改函数中没有PushDown函数，因为这里假设只有点修改一种操作，
如果题目中是点修改和区间修改混合的话，那么点修改中也需要PushDown。

四：非递归原理

非递归的思路很巧妙，思路以及部分代码实现 来自 清华大学 张昆玮 《统计的力量》，有兴趣可以去找来看。
非递归的实现，代码简单（尤其是点修改和区间查询），速度快，建树简单，遍历元素简单。总之能非递归就非递归吧。
不过，要支持区间修改的话，代码会变得复杂，所以区间修改的时候还是要取舍。有个特例，如果区间修改，但是只需要
在所有操作结束之后，一次性下推所有标记，然后求结果，这样的话，非递归写起来也是很方便的。
下面先讲思路，再讲实现。

点修改:

非递归的思想总的来说就是自底向上进行各种操作。回忆递归线段树的点修改，首先由根节点1向下递归，找到叶节点，然后，修改叶节点的值，再向上返回，在函数返回的过程中，更新路径上的节点的统计信息。而非递归线段树的思路是，
如果可以直接找到叶节点，那么就可以直接从叶节点向上更新，而一个节点找父节点是很容易的，编号除以2再下取整就行了。
那么，如何可以直接找到叶节点呢？非递归线段树扩充了普通线段树(假设元素数量为n)，使得所有非叶结点都有两个子结点且叶子结点都在同一层。
来观察一下扩充后的性质：