平衡树之*Splay*

陈启乾

CHEN QIQIAN

简介

我也不是谦虚，我一个蒟蒻，怎么就来讲SPLAY了呢？

前备知识

C++语法

基本代码实现能力

二叉查找树

带$lazytag$的线段树

*$Treap$

Splay

一种平衡二叉查找树

主要操作叫“$splay$”，把一个点通过旋转放到根节点的位置

均摊时间复杂度$O(\log n)$

Robert Tarjan ***orz***

核心思想：

每次查询会调整树的结构，使被查询频率高的条目更靠近树根。

Splay

简单操作：

1.二叉查找树
2.一颗比较高端的线段树（并不严谨

高端操作：维护动态森林的连通性和其上路径信息等…

最主要的几个操作：*rotate*

定义：

把一个节点转到它的父节点的位置不破坏其中序遍历的顺序。

分类：~~zig & zag~~

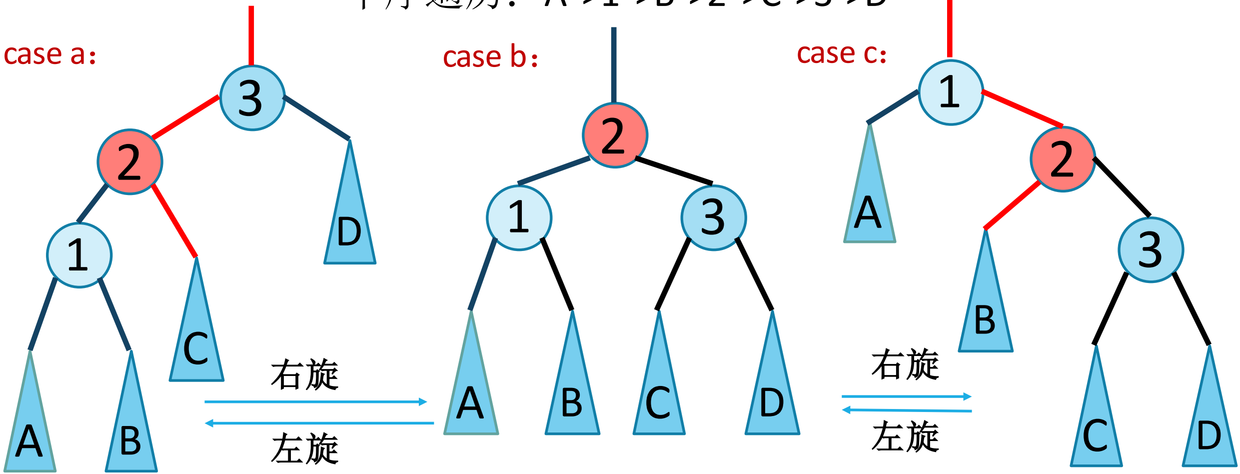
左旋 右旋

简而言之，就是把根节点转到父亲的位置上。

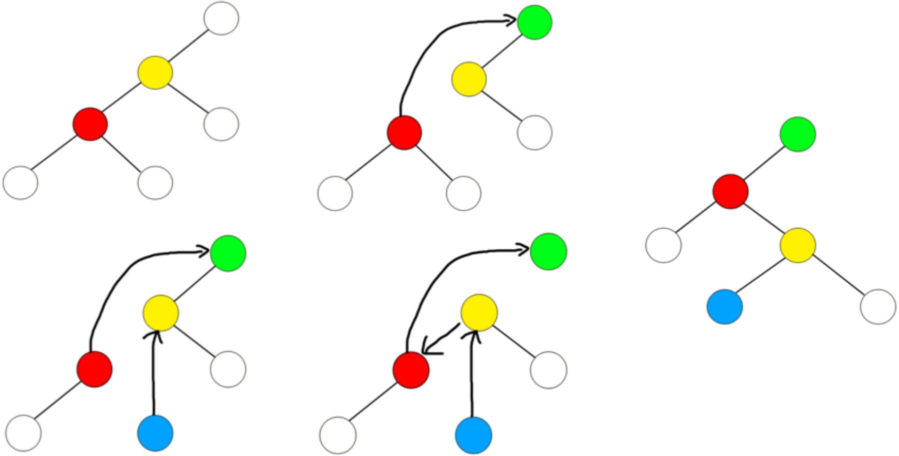
本质上的作用是调节树的深度（大小）。

最主要的几个操作：*rotate*

中序遍历：A->1->B->2->C->3->D



实现



实现

```
void rotate(int x){
  int y = f[x],z = f[y],
      t =(c[y][1]==x),w = c[x][1-t];
  if(z) c[z][c[z][1]==y] = x;//更新儿子
  c[x][1-t] = y,c[y][t] = w;
  if(w) f[w] = y;//更新父亲
  f[y] = x;f[x] = z;
  push_up(y),push_up(x);//维护信息
  if(!f[x]) root = x;
}
```

最主要的几个操作：双旋

双旋=两个单旋

父亲与祖父关系相同，转父亲再转自己

父亲与祖父关系不同，转自己再转自己

为什么？极端情况下保证复杂度（链状

我就写单旋行不行？可以，但可能被卡

最主要的几个操作：*splay*

用途：把一个点转到某个特定点下面。

实现：

```
void splay(int x,int target = 0){
  if(!x) return;
  while(f[x]!=target){//判断是否达到目标
    int y = f[x],z = f[y];
    if(z!=target){
      (c[y][1]==x)^(c[z][1]==y)?rotate(x):rotate(y);
      //如果祖父不是目标，双旋(同父异自己)
    }rotate(x);
    //如果祖父是目标，单旋
  }
}
```

一些容易写挂的地方

1. 换父亲儿子的时候千万别写错字母！！！

2. 判断父亲和儿子是否存在！$Link\ Cut\ Tree$里面不判断就错！

一点微小的实现

作为一棵**普通平衡树**，有必要告诉你们一点人生的经验。

重点！

*splay*操作是复杂度的保证！

你需要随时随地的*splay*！（

结构体的定义

```
struct Splay{
    int val[MAXN],siz[MAXN],cnt[MAXN];
    int c[MAXN][2],f[MAXN],tot,root;
    int newnode(int v = 0){
        val[++tot] = v;
        return tot;
    }
    void push_up(int x){
        if(!x) return;
        siz[x] = siz[c[x][0]] + siz[c[x][1]] + cnt[x];
    }
    //...
};
```

寻找节点：$find$

按值寻找。跟其他平衡树一样。（这里默认寻找的值存在

```
int find(int v){
  int x = root;
  //判断是否存在该节点且不是当前插入的值
  while(x && val[x]!=v)
    x = c[x][val[x] < v];
  splay(x);//一定Splay!
  return x;
}
```

插入：*insert*

类似其他平衡树的查询操作，找到它应该放的位置，然后添加就好了。

*Splay*的插入有各种各样的写法，看个人喜好了。

这种大约常数比较小？

```
void insert(int v){
  int x = root,y = 0;
  while(x && val[x]!=v) 同时判空和数值相等!
    siz[x]++,y = x,x = c[x][val[x] < v];
  //按BST的方式查找，并且记录父节点，更新siz
  if(x) cnt[x]++,siz[x]++;//如果x已经存在
  else{
    x = newnode(v);//新建一个节点
    if(y) c[y][val[y] < v] = x;
    f[x] = y;siz[x] = cnt[x] = 1;
    if(!f[x]) root = x;//更新根结点
  }
  splay(x);//splay到根
}
```

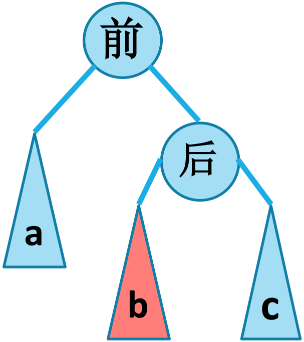
删除:*erase*

一个非常有用的思想： **（后面会用）**

Splay！

如果我们按值删除的话，那么我们把这个点的前驱*Splay*到根，后继*Splay*到根节点下方，可以发现，这个时候，b节点一定就是我们要删除的节点。（**需保证这个地方没有重复的同一值的节点**）

也可以按节点的指针或者编号删除。



删除:*erase*

实现：

```
void erase(int v){
  int x = find(v);//找到要删除的节点
  splay(x);//Splay到根
  if(cnt[x] > 1)//如果还有这个数
    cnt[x]--,siz[x]--;
  else{
    //l为前驱，r为后继
    int l = loworup(v,0),r = loworup(v,1);
    splay(l),splay(r,l);//l到根节点，r到上方为l节点
    c[r][0] = 0;//断开与该儿子的联系
    push_up(r),push_up(l);//需要维护大小，先儿子后父亲
  }
}
```

名次相关

在其他平衡树里面，我们也已经理解了与名次相关的查询工作。

1. 查询某数的名次:$getrank(v)$

2. 查询排名为某名次的数（节点编号）：$qrank(r)$

名次相关：*qrank*

```
int __qrank(int r){//寻找排名为r的数的编号
  int x = root;
  while(x){
    if(r <= siz[c[x][0]])
      x = c[x][0];//在左子树
    else if(r <= siz[c[x][0]] + cnt[x])
      break;//就是这个节点
    else
      r -= siz[c[x][0]] + cnt[x],x = c[x][1];
    //在右子树
  }
  splay(x);//不要忘了Splay
  return x;
}
```

名次相关：*getrank*

注意：同时判空和节点是否存在

```
int getrank(int v){//寻找数v的排名
  int x = root,ans = 0;//这里不保证数v存在
  while(x && val[x] != v){
    if(v < val[x])
      x = c[x][0];//向左
    else
      ans += siz[c[x][0]]+cnt[x],x = c[x][1];
      //向右，累加上左子树和这个节点的大小
  }ans += siz[c[x][0]]+1;
  splay(x);//不要忘记Splay!
  return ans;
}
```

前驱、后继

实现方法一：同*Treap*

以前驱为例。

如果寻找到的节点有左子树，那么其前驱就是左子树里面的最大值；如果没有左子树，那么其前驱就是寻找的路径上最靠近的一个向右寻找的节点。

第一种很容易想明白，第二种稍难一些。

感性理解吧（雾

前驱、后继

实现方法二：

以前驱为例。

将该数其Splay至根，然后其左子树里面的最大值就是其前驱。

若该数不存在，就插入该数再删除。

前驱、后继

实现方法三：

以前驱为例。

查询该数排名为$rank$，然后寻找排名为$rank-1$的数。

后继稍微复杂。

可以先特判该数是否存在，记其个数为$cnt[x]$，答案即为排名为$rank+cnt[i]$的数。

比较麻烦。

前驱、后继

我选择方法一：

```
int loworup(int v,int t){//0前驱 1后继
  int last = 0,x = root;//last表示上一个没有当前寻找方向的节点
  while(x && val[x] != v){//注意要同时判断
    last = (val[x]<v)^t?x:last;//更新last
    x = c[x][val[x] < v];
  }
  if(c[x][t]){//如果存在该方向子节点
    last = c[x][t];
    while(c[last][1-t]) last = c[last][1-t];
    //寻找该方向最小（大）节点
  }
  return last;
}
```

一些乱七八糟的封装

```
void __print(int x,int depth){
  if(depth == 0)
    printf("root:%d ------------------\n",x);
  if(!x) return;
  __print(c[x][0],depth+1);
  for(int i = 0;i<depth;i++) putchar(' ');
  printf("%d: val:%d siz:%d(cnt:%d) c:%d %d f:%d\n",
        x,val[x],siz[x],cnt[x],c[x][0],c[x][1],f[x]);
  __print(c[x][1],depth+1);
  if(depth == 0) printf("------------------\n");
}
void print(){__print(root,0);}
int lower(int v){return val[loworup(v,0)];}
int upper(int v){return val[loworup(v,1)];}
int qrank(int r){return val[__qrank(r)];}
```

复杂度证明：势能分析

这一页没有ppt。

~~来不及做了，只好现场胡诌。~~

然而…

$Splay$常数很大…自带$O(N\log N)$的常数…

尤其在普通平衡树上…能写$Treap$尽量写$Treap$…

~~能用set尽量用set…~~

维护区间信息

那个**文艺平衡树**，比你们不知道高到哪里去了，我和他谈笑风生！

维护区间信息

- 区间和 区间最值 区间翻转 区间最大连续子区间...

线段树其实是*Splay*的真子集（部分上）

这一段很像一个能插入节点、能翻转序列的线段树。

能把任意一段子区间单独提取出来，随便瞎搞。

还记得~~我们过去的~~线段树吗？

哲学之问：

线段树为什么能保持$O(\log n)$的区间修改复杂度？

$lazytag$!

我们通过一个$lazy$标记标记在整段的区间上，并且在访问的时候通过$O(1)$的下放达到$O(\log n)$的复杂度。

那$Splay$上呢？

$Splay$是一种二叉树。它能保持自平衡。

线段树是一种二叉树。它保持高度平衡。

很相似。

我们在$Splay$的每个节点上维护一些$lazytag$，其$push_down$的时间应当是$O(1)$的（$add, mul, rev, set\_to \ldots$）

关于标记下传

联想线段树，我们需要得出 需要进行标记下传操作 的位置。

我们什么时候会**破坏父子关系或者需要子树里面的信息**？

查询、旋转操作。这些操作都需要$push_down$。

可以证明，$push_down$是$O(1)$时，这并不会改变其他操作复杂度。

一定保证：如果你给一个节点打上标记，保证这个节点的维护的信息都是正确的。

一些其他操作：*push_down*

注意下传顺序。

注意下传层数：应当只是一层。

建议写个函数，专门负责打标记（修改）。

具体来说

```
void push_down(int x){
  if(label1[x]){
    add_label1(c[x][0],???);
    add_label1(c[x][1],???);
    label1[x] = 0;
  }
  if(label2[x]){
    add_label2(c[x][0],???);
    add_label2(c[x][1],???);
    label2[x] = 0;
  }
}
```

```
void add_label1(int x,???){
  //do something
}

//修改时
add_label1(x,???)
```

提取区间：*split*

还记得删除操作吗？有没有什么启发?

还记得*Splay*操作吗？

*Splay*的形态随时可改，所以我们可以把一个连续区间内的节点都放到同一个子树里面。

所以我们只要在这个子树上打上标记，就相当于在这个区间上打上了标记。

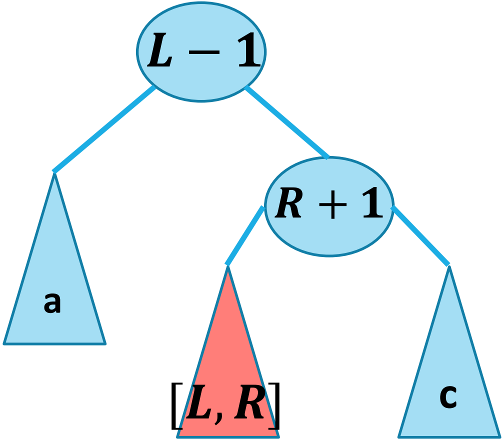
提取区间：$split$

如果我们需要提取$[L,R]$区间的节点。

我们只要把排名$L-1$的节点$Splay$到根节点，排名$R+1$的节点$Splay$到$L-1$的节点下方，在$R+1$的节点左儿子的部分就是区间$[L,R]$了。

所以你手里掌握了这个区间！你可以做的事情：**打tag，删掉[L,R]，插入一个新区间，其他一些奇奇怪怪的事情**。

BTW…如果你做了修改，记得维护$R+1$，$L-1$两个节点！



实现

实现上有一点有趣。

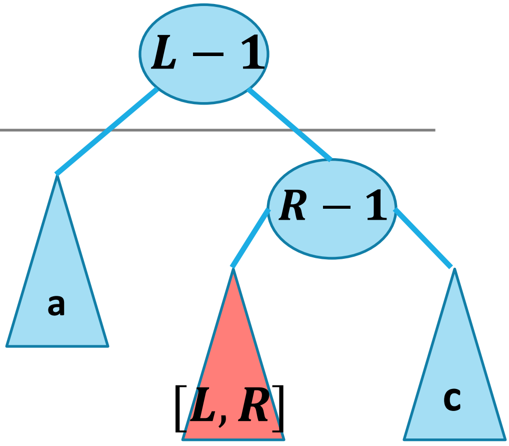
例如我们要选取$[L,R]$区间，先将排名$L-1$的节点$Splay$到根节点，排名$R+1$的节点$Splay$到$L-1$的节点下方，在$R+1$的节点左儿子的部分就是区间$[L,R]$了。

我们需要在$\boldsymbol{Splay}$两边加上两个没有用的节点来保证排名$[\boldsymbol{l-1}]$和$[\boldsymbol{r+1}]$的节点存在。这个时候$\boldsymbol{split}$的时候你要提上来的两个节点的排名也会差一个1.

(需要提取的区间就变成了$[\boldsymbol{L+1},\boldsymbol{R+1}]$)

关于提取区间

注意这里因为我们有可能会用到父节点的一些信息，而且这个函数很短，所以一般直接放到具体的更改函数中会比较好。




```
int split(int l,int r){
  int x = qrank(l-1),y = qrank(r+1);//找到两端之外的端点
  splay(x),splay(y,x);//将x旋转到根，y旋转到x下方
  return c[y][0];
}
```

区间翻转：$reverse$

Splay最强的地方：区间翻转！~~序列之王orz~~

如何翻转（中序）一棵二叉树？

交换所有节点的左右儿子！可以打标记在$O(\log n)$的时间内完成。
提取区间$[L, R]$，然后打标记完成修改。

怎么改？

在这里应当打上标记，交换左右儿子。套路与其他标记类似。

关于下传*reverse*标记

建议在打*lazytag*的时候，就把该节点的左右节点交换。

推荐的方法：

（不用可能会出错！）

```
void reverse(int x){
  if(!x) return;
  swap(c[x][0],c[x][1]);
  rev[x] ^= 1;
}
void push_down(int x){
  if(!x) return;
  if(rev[x]){
    reverse(c[x][0]),reverse(c[x][1]);
    rev[x] = 0;
  }
}

//...
reverse(x)//这样就可以当作更改完成
```

一些实现上的容易踩到的坑

Q:那些地方需要*push_down*?

A: 一切改变父子关系的地方和需要查询的地方：*rotate*，*splay*，*qrank*，*output*。尤其是*rotate*，**先*push_down*自己&父节点再开始你的操作**。

如果有*reverse*操作，在需要**判断儿子与父亲的关系时**（按排名寻找，寻找最小、最大节点）一定要*push_down*!

最后的历练

可她说，中央已经硬点你来做这些题了。

于是我就念了两句诗，叫做**“苟利国家死生以之”**。

习题

*「Luogu P3369」普通平衡树

「NOI2004」郁闷的出纳员-平衡树

「Luogu P3391」文艺平衡树-区间翻转

「CQOI2014」排序机械臂-区间翻转

「CQOI2013」多项式的运算-线段树版Splay

「NOI2005」维护数列-综合

NOI2004-郁闷的出纳员

题意：

维护一个数列。现有四种命令:

- 新加入一个数k;
- 把每个数加上k;
- 把每个数减去k;
- 查询第k大的数。

如果数列中的任意数小于min，将它立即删除。并在最后输出总共删去的数的个数res。

如果新加入的数k的初值小于min，它将不会被加入数列。

NOI2004-郁闷的出纳员

显然我们按照工资来需要维护一棵平衡树。

需要维护一个$lazy$标记？然后每次各种修改？

太难了。有没有简单一点的做法？

注意到每次的修改都是针对全局的。既然我们在树内打标记很费劲，能不能把标记改为全局，然后每次插入的时候按标记更改插入的数？

solution

构建一颗Splay树。需要记录目前已经全体加过或者减过的数，也就是一个相对值。换算来说就是**<u>树外-相对值=树内，树内+相对值=树外</u>**。需要添加两个虚的最大和最小节点，也会导致排名计算的一些变化。

<u>插入一个数</u>

先判断是否满足插入条件，即此数是否大于min，然后减去相对后正常插入，splay至根节点。

<u>加上一个数</u>

直接更改全局相对值，由于不会出现删数，不会有其他操作。

solution

减去一个数

首先更改全局相对值，再把小于min的数删除，简单的来说就是把第一个大于等于min的数splay到根上，然后删除左子树，补上左边的最小节点。如果正好存在值为min的节点，就将它直接splay到根，完成上述操作；如果不存在，就插入一个值为min-1的节点，寻找它的后继，并splay到根，完成上述操作。这时统计res需要减去我们刚刚加上的节点。

查询第k大

直接查，然后splay到根。只需要注意我们的数列是从小到大排列的。

cqoi2014-排序机械臂

题意：

维护一个序列，第i次操作时寻找第i小的数的所在位置P_i,并将$(P_{i-1},P_i]$的区间翻转。

如果有相同的数，必须保证排序后它们的相对位置关系与初始时相同。

solution

其实就是个排序是吧。

怎么找到第i小的数的位置？查rank？

你天真了！ naïve!

这里是线段树！而不是二叉查找树！

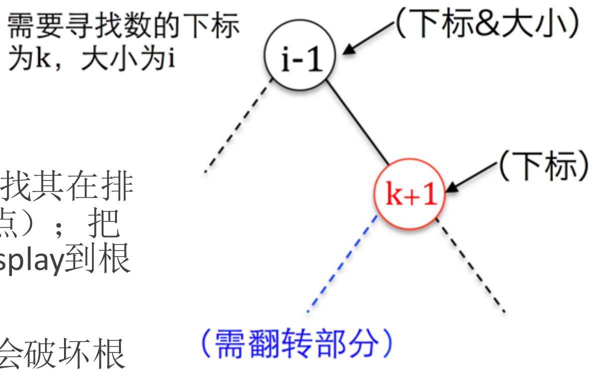
其实只需要搞一个数列，把这个节点的指针（编号）存起来，不就知道在哪里了？

solution

先找到第i大的数对应的节点指针，寻找其在排序二叉树中的后继节点（图中红色节点）；把第i-1大的节点splay到根；然后把后继splay到根的右子树。

但注意在实际查找中，因为寻找后继会破坏根结点，所以要先找到后继节点，然后再完成上述操作。

然后关于位置，我们可以看出，根节点左边（包括根结点），也就是图中的绿色部分应当有i-1个数，而其他在i左边（包括i）的数应该就是图中的蓝色部分，所以只要将蓝色部分的size加上一个i-1就是每一次操作的结果。



cqoi2013-多项式的计算

维护一个动态的关于x的无穷多项式$f(x)$，这个多项式初始时对于所有i有$a_i = 0$.

$$f(x) = a_0x^0 + a_1x^1 + a_2x^2 \ldots$$

操作者可以进行四种操作：

- $mul\ L\ R\ V$表示将x^L到x^R这些项的系数乘上某个定值v；
- $add\ L\ R\ V$表示将x^L到x^R这些项的系数加上某个定值v；
- $mulx\ L\ R$表示将x^L到x^R这些项乘上x变量；
- $query\ V$求$f(v)$的值。

经过观察，发现操作集中在前三种，第四种操作不会出现超过10次。

solution

前两个操作让我们想到线段树的模板，第三个操作如果从序列的角度来看就像是把一个序列向右移动，在把被冲掉的那一个位置加到原来的位数上去。

显然啊！同志们，这是送分题啊！Splay套套套…

- add操作：提取区间，打标记，维护信息。

- mul操作：提取区间，打标记，维护信息。

- mulx操作：呃…先找到rank为l-1,l,r,r+1,r+2的节点。删除掉r+1号节点，把其值加到r上去，然后在l-1和l之间插入一个值为0的节点，维护信息。

注意：push_down操作先传muln，再传addn。打标记几乎同线段树模板，就不说了。

noi2005-维护数列

维护一个数列，给定初始的n个数字。

现有六种命令:

- 在第pos个数后插入tot个数;
- 翻转从第pos个数开始的tot个数;
- 删除从第pos个数开始的tot个数;
- 查询从第pos个数开始的tot个数的和;
- 设定从第pos个数开始的tot个数设定为c;
- 查询整个数列中和最大的连续子区间的大小。

思路？

像不像线段树能解决的事情？

~~这可难不倒序列之王~~

-

问题：这些操作都可以用什么标记来解决？区间性的插入和删除怎么解决？

solution

对于节点，要维护：

树的大小，树的权值和，树从左端点开始的最大连续和，树从右端点开始的最大连续和，和树的最大连续子区间和。

主要操作：

- **pushdown** 往下push，修改两个子节点并打上标记。

- **pushup** 更新所有信息，维护三个max信息的方式有些特殊~~大家都会的~~。这个与线段树的区间最大查询有点不太一样，根节点也有代表的数，这个需要记住。

solution

- **建树** 这里对于建树的复杂度要求是O(n)的，按照线段树的方式O(n)建树即可。

- **最大连续和** 直接输出根节点维护的最大连续子区间的值即可。

- **插入** 把即将插入的tot个数按照上文的介绍方法建树。把这个位置之前的节点放到根节点，之后的节点放到根节点的右儿子，把新建的树放到

solution

接下来的操作都提取区间为$[pos, pos + tot]$。

- <u>**删除**</u> 直接删除子树，维护信息。因为内存不够（64MB），需要垃圾回收。

- <u>**求和**</u> 输出子树的和。

- <u>**翻转**</u> 翻转子树并打标记，并维护信息。这里的翻转也要交换左起最大连续和 和 右起最大连续和。

- <u>**设定**</u> 对中间子树完成设定并打标记，修改区间和等等信息。

solution

还有一点就是垃圾回收。简略来说就是把删除的节点暴力的扔到一个栈里面，然后能用就用，不能有就再新开内存池。其他也没有什么重要的。

多pushdown pushup几次，然后这些操作都是要注意边界，也就是NULL时候的条件。

pushup的合并公式也需要好好斟酌。

push_up

```
void pushup(){
  if(this == *null) return;
  if(son[0] == *null && son[1] == *null){
    size = 1;sumn = lmax = rmax = maxn = val;
    return;
  }
  size = son[0]->size + son[1]->size + 1;
  sumn = son[0]->sumn + son[1]->sumn + val;
  lmax = max(son[0]->lmax,son[0]->sumn + val + max(0,son[1]->lmax));
  rmax = max(son[1]->rmax,son[1]->sumn + val + max(0,son[0]->rmax));
  maxn = max(0,son[0]->rmax) + val + max(0,son[1]->lmax);
  maxn = max(maxn,max(son[0]->maxn,son[1]->maxn));
}
```

That’s all.

如果我现在还活着站在讲台上，那真的是非常荣幸了。

参考

对于伸展树（Splay）复杂度的研究与证明（附pdf）

[知乎] Splay中的旋转操作用单旋与双旋的区别是什么？-negiizhao的回答

[Menci’s Blog] Splay 学习笔记