动态序列与动态树问题——浅谈几种常用数据结构

天津南开中学　莫凡*

2014 年 6 月 5 日

摘　要

动态序列问题是指给定一个序列，在其上执行一系列在线操作（例如一段数的修改、某一区间的反转或某一区间内的最大值查询等等），是算法竞赛中的常见问题，在生产生活中也具有较强的实用价值。本文重点通过介绍通过几种常用的平衡二叉树型数据结构解决动态序列问题。动态树问题是动态序列问题的延伸，支持一棵树（或是一片森林）的点上、边上信息的维护以及形态的变换。由于动态树问题将维护的对象由序列拓展成一棵树，故其在图论中具有重要的实用价值。由于时间仓促，加上作者水平过弱，不足之处请多多指正。

本文涉及的数据结构虽然都非常简单，但是并不适合初学算法与数据结构的读者，阅读这篇文章的前提条件只有一个：所有涉及的问题你都可以用暴力实现

*作者邮箱：w007878@sohu.com

Index

Part I

几种常用的数据结构

这一部分着重介绍线段树、Splay、Treap 这三种数据结构的实现、复杂度分析以及各自适用的范围。

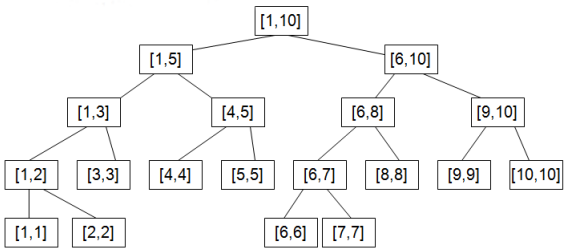
1 线段树

1.1 线段树概况

考虑这样一个问题：假设我们有一个长度为 n 的整数数列，每次给出一个区间 $[L, R]$，要求求出第 L 个数到第 R 个数中的最小值（RMQ 问题[1]）

最简单粗暴的方法就是每次将区间 $[L, R]$ 扫一遍，然后求出最小值，假设有 m 个询问，显然这样的复杂度是 $O(nm)$ 的。

线段树是这样一种数据结构：每一个点代表一个区间，存储的是这个区间内的信息（对于本题，区间最小值）。这个点的左儿子存储它表示的区间的左半边，右儿子存储它表示的区间的右半边，以此类推，直至叶子节点（保存单个数的值），如下图所示：



显然这棵树的深度是严格的 $O(\log n)$。这样，在查询的时候我只要查询若干个点的最小值就可以了。注意，我们只需要让查询的点不重不漏包含整个区间 $[L, R]$ 即可，而并不是要找到该区间包含的所有节点。我们递归定义查找过程：

[1]这个问题有各种各样的复杂度优秀的算法，但是与本文主题无关，有兴趣的读者可以自行搜索相关资料

1. 若当前节点区间被查询区间所包含，返回这个区间的值
2. 若查询区间与左子树区间有交集，递归查询左子树
3. 若查询区间与右子树区间有交集，递归查询右子树
4. 合并并返回递归查询的值

操作 2、3 保证了我们每次查到的节点都和查询区间有交集。由于我们查询的区间是连续的，所以至多有一个点 P，它的左右儿子与查询区间都有交集且不是完全覆盖。也就是说我们只有至多两个递归是一直递归到叶子的（在 P 处分叉），所以很简单地证明了查询算法的单次复杂度是 $O(\log n)$。

援引清华大学张昆玮学长 [1]: 线段树产生于计算几何问题的应用中，我们将它用到数列统计问题中时，往往是将树轴当作区间去处理，而由于我们处理的往往是离散的量，所以线段树退化成了“点树”, 而点树才是维护动态序列中有用的形式。

1.2 线段树的修改与信息合并

假设 RMQ 问题中，我们每次可以修改一个数，我们只需要更新线段树上从根到叶子的一条路径上的所有节点的信息即可，复杂度仍旧是 $O(\log n)$。如果更改一下问题，每次可以修改一个连续的区间，又该怎么操作以保证复杂度呢？

我们引入**懒标记**的概念。懒标记是指在某一个节点上打上标记，表示以这个节点为根的子树的信息都是一样的，查询至此无需继续递归。如果要继续对这棵子树的一部分进行其他修改时，我们需要将懒标记**下放**给它的左右儿子。所以，进行段修改的算法流程和段查询是完全一致的，只需要在每个需要修改的节点上打上懒标记即可。

打懒标记有一个原则：在打标记的同时应该将打上标记的那个节点的信息全部更新完全，而不应该是查询时根据标记现算，那样会大幅增加编程和思维难度。

注意到如果我们对一个节点的子树上的点进行了修改，那么我们需要使用它的左右儿子的信息对这个点的信息进行更新，这就需要我们维护的信息满足**可合并性**，例如区间的最小值（一个区间的最小值显然是这个区间前半边的最小值和后半段的最小值中的较小着）。类似地，可合并性被定义为一个区间的某种信息可以通过它前半段的这种信息与后半段的这种信息通过简单的计算得到。类似区间众数、区间次大值等就不满足可合并性。

1.3 线段树的具体实现与编程细节

我们将使用与 C++ 风格相近的伪代码进行讲解。为了模板化，我们将“懒标记下放” 和 “用子树信息更新自己” 两个最基本的操作封装起来，称为 pushdown 和 update。为了便于叙述，我们采用指针而非数组的存储方式。节点的左儿子称为 s[0], 右儿子称为 s[1]。

第一步：修改

```
void change(node *p,int l,int r,int v)
//当前修改到了节点p，需要修改的区间是[l,r]，要将区间改为v
{
        if(区间被完全覆盖){p->cover(v); return;}
        //若完全覆盖，则给当前点打上懒标记并更改信息
        p->pushdown();
        //下放可能存在的懒标记
        if(左子树和区间有交集) change(p->s[0],l,r,v);
        if(右子树和区间有交集) change(p->s[1],l,r,v);
        p->update();
}
```

查询操作其实和修改操作如出一辙

第二步：查询

```
int query(node *p,int l,int r)
{
        if(区间被完全覆盖||有懒标记) return p->data;
        //若完全覆盖，直接返回
        p->pushdown();
        int l_data,r_data;
        if(左子树和区间有交集) l_data=query(p->s[0],l,r);
        if(右子树和区间有交集) r_data=query(p->s[1],l,r);
        return merge(l_data,r_data);
        //返回合并后的信息
}
```

这就是线段树的基本操作，因题而异的只有 pushdwn 和 update 这两个环节了，是不是非常简洁明了？后面的数据结构也会沿用 pushdown& update 的表达形式

1.4 另一种实现方式

刚刚我们讨论的标准线段树模板，采用的是自顶向下的模式，这种模式有一个弊端，就是当我们修改、查询的时候都需要判断左右子树与查询区

间是否相交，而且采用了递归调用的方式，这样会明显地增大常数。考虑假设每次修改都是点修改，那我们只需要更新从根到叶子这一条链上的信息。假如说我们可以直接找到这个叶子，一路上来，既省去了判断，又避免了递归 [1]。

这就是由 zkw 改进的自底向上线段树实现思路。这种线段树采用数组的存储方式，所以我们从根开始顺序编号为 (1、2、3、.....)，某个节点的编号为 x，那么它的父亲的编号就是 $\lfloor \frac{x}{2} \rfloor$，左右儿子分别是 $2x$ 和 $2x+1$。为了可以迅速找到做到每个叶子并迅速定位查询区间，我们将线段树的叶子节点拓展到不小于序列长度 -2 的二的整次幂（计为 base），并从从左往右的第二个叶子节点 (编号为 base+1) 开始存储数列的信息. 这样我们保证了所有叶子节点的深度相同，而且很轻易地找到开区间（查询的时候会用到）。

对单点 x 进行修改的时候，我们只需要从 base+x 号节点一直除以二直至修改到 1 号节点（根）。由于每次都是/2 操作，所以可以使用位运算来加速。

对于区间查询，刚刚我们在分析传统线段树时得到了一个有用的结论：至多有一个结点，它的左右子树都与查询区间有交集且都不被完全覆盖。那么我们考虑从底层的两个区间端点逐步上跳逼近这个点即可。这个点的左子树的区间包含情况一定是许多节点的右儿子存在于我们的查询区间内，而右子树的情况则是许多节点的左儿子在查询节点内。那好，我们考虑目前需要查询的区间 $[L, R]$，如果左端点是它父亲的右儿子（等价于它的编号是奇数），那么左端点则一定被包含，如果它是左儿子，那好，它的祖先节点中至少会有一个被包含，而它自己就没有必要了。右端点的情况亦然。但是如果这之后我们直接跳到他们的父亲，他们的父亲就变成了开区间端点（也就是说包含了两个本不应该被查询的两个点）。这样的话我们不妨把查询闭区间 $[L, R]$ 直接转化为开区间 $(L-1, R+1)$。我们从左端点 $L-1$ 往上跳，如果它是左儿子，则查询结果包含它的兄弟（等价与它的编号异或 1），否则不查。右端点与之同步，直至两个端点的编号相差小于等于 1（这时他们就相邻了）。这么做显然可以使我们判断的永远是开区间 $(L-1, R+1)$ 的端点。

标记永久化 这种线段树就不能处理区间修改问题么？答案是可以的，我们采用了“标记永久化”的策略。所谓标记永久化，就是我们依然在修改的同时在我们修改的节点打上懒标记，但是这些标记永远不会被下放。修改的过程和询问是一致的（本质都是寻找区间），但是查询的时候就需要根据一路

上经过的节点的标记信息来计算最终的查询值了。鱼与熊掌不可兼得，标记永久化大大增加了编程和思维的复杂度，自底向上性价比就不如自顶向下线段树了。

实现

```
void preset(int n, int num[])//初始化
{
        for(base=1;base<n+2;base>>=1);//计算 base
        for(int i = 1; i <= n; ++i)data[base+i]=num[i];//叶节点初值
        for(int i = base-1;i;i--)data[i]=merge(data[i<<1],data[(i<<1)|1]);
        //上层节点初值
}
int query(int l,int r)
{
        int ans=0;
        for(l=l+base-1,r=r+base+1;l<r-1;l<<=1,r<<=1)
        //转为开区间
        {
                if(!(l&1)) ans=merge(ans,data[l^1]);
                if( (r&1)) ans=merge(ans,data[r^1]);
        }
        return ans;
}
void change(int pos,int val)//将pos位置的值改为val
{
        data[pos+base]=val;
        for(int i=(pos+ base)>>1; i; i>>=1)update(data[i]);
}
```

What a 简单明了的数据结构！

1.5 适用范围

由于线段树是极度平衡的二叉查找树，编程复杂度和难度都极低，所以是处理同类问题的首选（当然树状数组之类的优美结构在简单问题中能用还是最好）。我们来总结一下线段树适用的问题范围：

- 信息需要可合并。当然这也是分治数据结构的通用要求。对于不可合并的信息，可以用分块数据结构解决，但是不在本文讨论范围之内。
- 序列不能有“伤筋动骨”的改变，插入、删除等等[2]。
- 查询、修改应该是连续的区间

[2]不是说线段树无法做到，而是有更适合处理这些问题的数据结构

2 伸展树

伸展树（Splay）是我最喜欢的数据结构，它是一种不需要记录额外信息的、自我调整的平衡二叉查找树。它由 Daniel Sleator 和 Robert Tarjan 共同发明，通过每次访问到一个结点就将它旋转到根的方式保持它的“平衡”。事实上它可能在某一瞬间很不平衡，但是我们只要保证每次操作的复杂度均摊下来是 $O(\log n)$ 的即可。

二叉查找树是指每个节点有一个键值（这个键值可以显式保存，例如分数等，也可以隐式存在，例如在数列中的相对位置），它左子树中的节点键值都小于它，右子树的节点键值都大于它。那好，这样我们就可以做到在期不超过树的深度的时间查找或插入一个特定键值的节点。平衡二叉查找树的基本思想就是通过尽可能地降低深度（降为期望的 $\log n$）来缩减操作时间。很不幸，Splay 并不试图在所有时刻都变得很浅，而是使得树只在极少数情况下变得很深。

2.1 平衡树的旋转机制

基于旋转的平衡树是通过一些旋转操作来调整的。如图所示。

上图中，通过旋转可以调整点 E 和 S 的关系，但不改变节点的相对位置。旋转操作分为左旋和右旋，一般来讲，旋转的节点如果是它的父亲的左儿子，那么执行的是右旋，反之则是左旋。由于一个结点的位置决定了它的旋转方向，所以在后文中区分左旋和右旋是没有必要的，我们将其统称为旋转。

2.2 伸展——Splay 的基本操作

Splay 的基本操作就是伸展。Splay 的是复杂度分析是基于势能分析的，而这个势能分析的基础正是伸展操作——即在每次查找到想要的节点之后通过一系列旋转将它变成树根。假设我们当前状态的树是一条长度为 n 的链，如果要将最下面的那个叶子节点旋转到根的话，通过对其进行 n-1 次

旋转显然是不靠谱的（为什么呢？你可以尝试画一下，这样之后就变成了一个另外一条链。那么我们每次查询的复杂度变为了 $O(n)$，总复杂度变为了 $O(mn)$，效率十分低下。

为了克服这一问题，我们引用双旋机制：当一个点未旋转到根，且它的父亲也不是根时需要进行判断是：若它和它的父亲都在同侧（也就是构成了一条小链的情况下），先旋转它的父亲，再旋转它（这么做的原因会在第 6 小节讲到，可以简单理解为把父亲和祖父转另一个叉上）。否则还是旋转两次它自己

那好，所谓 Splay 操作就是不停地旋转直至到达目标。请注意，这里的目标不一定是根，还可以是其他随意一个它的祖先节点，后面的区间操作中会用到。

2.3 懒标记、区间查询和修改

介绍完伸展操作之后，我们就可以实现在 Splay 中查找、插入或是修改某个特定键值的节点了，但是这只能维护一个动态集合，想用它来维护数列，我们还需要让它能做更多的事情。

显然，我们可以通过在某个节点处记录它整棵子树的信息。与线段树不同的是，线段树的非叶子节点信息是由左儿子信息、右儿子信息两部分合并来的，而平衡树的每个节点之间都是平等的，并没有高低贵贱之分，所以节点记录子树值之外还需要记录自己的值。那么，每个节点记录的子树信息就应该是由左子树信息、自身的信息和右子树信息三部分合并。而且，它的左右子树很可能不同时存在（线段树则没有这个问题）

假定我们还是有一个序列，支持查询 RMQ，但是允许在某一位置插入或删除一段数，这是线段树就显得没有那么容易了。而 Splay 则可以非常出色地胜任这一工作。这时，每个节点在序列的位置就是它的键值。但是由于中间会有大规模的插入，所以直接记录 pos 显然是不合适的。但是我们可以通过记录 size 来计算每个节点的实际位置（在它左边的点永远不会跑到右边，不是么？）。当我们需要提取区间 $[L, R]$ 时，我们应该把第 R+1 个点 splay 到根，再将第 L-1 个点 splay 到根的左儿子，那么根的左儿子的右子树就是我们需要提取的区间啦！

这时候，我们只需要在那个节点上搜寻我们需要的信息或是打上懒标记就可以了。当然也可以选择把它删除。

这样，它就能处理线段树能处理的大多数问题啦（还有一些问题线段树

具有无可比拟的优势，后文会提到)！它还能处理许多线段树做不到的，比如插入删除等等。

2.4 代码实现（C++）

通过利用 C++ 语言的语言特性，Splay 有一种非常好写的实现方式 (ORZ WJMZBMR)，所以这里只提这一种实现。

节点的定义 每个节点按照之前的约定，使用 s[0]、s[1] 表示其左右儿子，包含若干个修改操作和内置函数 pushdown、update，当然也包括这棵子树的信息、节点自身的信息（我们以 size 为例）和懒标记。此外，我们定义一个函数 getlr() 来返回它是它父亲的哪个儿子（左返回 0, 右返回 1），以及一个 link 函数将一个结点连接到左儿子或右儿子上：

```
struct node
{
    node *p,*s[2];
    int key,size;
    node(){p=s[0]=s[1]=0; size=1; key=0;}
    node(int key) :key(key) {p=s[0]=s[1]=0; size=1;}
    bool getlr(){return p->s[1]==this;}
    node *link(int w,node *p){s[w]=p; if(p)p->p=this; return this;}
    void update(){size=1+(s[0]?s[0]->size:0)+(s[1]?s[1]->size:0);}
};
```

旋转和伸展 利用节点的 getlr() 函数，如果 p 是右儿子，就进行左旋：将 p 的左儿子连接到 p 的父亲上，再将连接好的 p 的父亲连接到 p 的左儿子位置上。右旋的情况与此对称。伸展操作更容易理解：如果转两次达不到目标，就双旋。如果最后还需要转一次，就再进行一次单旋。请注意随时 update。(为什么我们在 rot 函数里只更新 p 的父亲而不更新 p 呢？因为 p 会在 splay 的最后一步更新，提前更新属于浪费时间)

```
void rot(node *p)
{
    node *q=p->p->p;
    p->getlr()?p->link(0,p->p->link(1,p->s[0])):p->link(1,p->p->link(0,p->s[1]));
    p->p->update();
    if(q)q->link(q->s[1]==p->p,p); else{p->p=0; root=p;}
}
void splay(node *p,node *tar)
{
    while(p->p!=tar&&p->p->p!=tar)
        p->getlr()==p->p->getlr()?(rot(p->p),rot(p)):(rot(p),rot(p));
    if(p->p)rot(p);
```

```
    p->update();
}
```

插入 直接寻找插入的位置即可，记着随时 pushdown，最后把插入的节点旋转到根

```
void insert(int k)
{
        node *p=root,*q=NULL;
        while(p){p->pushdown(); q=p; p=p->s[p->key<k];}
        p=new node(k);
        if(!q){root=p;return;}
        q->link(q->key<k,p);q->update();splay(p,0);
}
```

寻找第 k 个数 每次寻找当前子树中相对位置为 k 的数，所以记得更新 k 的值。查找的过程中 pushdown，最后 splay 是基本原则，不必多说。

```
node *finkKth(int k)
{
        node *p=root; int w=p->s[0]?p->s[0]->size:0;
        while(p->pushdown(),w+1!=k)
        {
                if(w>=k)p=p->s[0];
                else{k-=w+1;p=p->s[1];}
                w=p->s[0]?p->s[0]->size:0;
        }
        splay(p,0);return p;
}
```

提取区间 注意到点 l-1 和 r+1 可能不存在，我们可以通过在数列中插入哨兵的方式避免。

```
void pick(int l,int r)
{
        node *p=findKth(l-1),*q=findKth(r+1);
        splay(p,q);
        //对p->s[1]做一些balabala....
}
```

这就完了？这就完了。

2.5 优势与弊端

个人认为，伸展树的最大优势就在于异常好写。而且它功能强大，十分方便快捷。当然它也不是无所不能的，它也要求区间信息可合并。在某些时

候我们会遇到许多麻烦：例如，我们要处理许多棵 Splay 时，哨兵的开销就变得不容忽视，而因此带来的区间 ± 1 问题十分令人苦恼，如果不加哨兵，特判会让人十分不爽。

在后文中会提到，Splay 这种旋转机制的平衡树由于形态时刻发生改变，所以在维护某些信息（比如它里面存储了它整个子树包含的数的集合）时 update 的复杂度骤增 [5]。而且 Splay 不容易实现可持久化。

2.6 复杂度分析

这一部分的存在是为了保持知识结构的完整性，没有兴趣的读者可以直接略过。由于作者水平有限，本节内容完全引用上海交通大学高宇学长：

先介绍什么叫均摊分析吧: 假如我们有 m 个操作, 每个实际耗时 T_i, 为了估计所有操作的总时间, 即 $\sum T$, 可以采用如下间接方法: 为每次操作设定一个期望花费时间 P_i, 那么对于每次操作, 我们定义 $P_i - T_i = A_i$, 称为这次操作的势差:

$$T_1 + A_1 = P_1$$

$$..$$

$$T_n + A_n = P_n$$

求和，$\sum T + \sum A = \sum P$. 这时, 如果 $\sum P$ 和 $\sum A$ 都有我们所期望的界限, 就可以间接地求出 $\sum T$ 的界限.

这里的 P_i, 就是我们常说的均摊运行时间, 如 $O(\log n)$ 等, 要确定 $\sum T$ 的界, 重点就在于考虑 $\sum P$ 的上界和 $\sum A$ 的下界 (移项变号). 实际应用中, 经常把"A 的部分和加上某一个常量" 称为势函数,$\sum A$ 的下界, 就是势函数尾首差的下界. 有了这些理解, 下面介绍 splay 的均摊分析.

由于 splay 上任何操作都是若干伸展操作, 所以我们只要考虑伸展操作. 每次伸展操作的时间, 是伸展的点的深度, 直接估计非常困难. 这里, 我们对于每个点定义 $R(v) = \log(size[v])$, 即 v 为根的子树中的点数关于 2 的对数. 定义势函数为 $\sum R$(关于所有点 v 的 $R(v)$ 求和). 这个势函数始终大于零, 且最大值是 $O(n \log n)$ 的 (可以画一个极端的树来说明), 所以它的首尾差, 也就是 $\sum A$, 是有下界的 ($-Cn \log n$,C 为常数). 现在我们就要证明 $\sum P$ 的界, 这里证明更强的结论: 每次伸展操作的 $T_i + A_i = P_i \leq \log n$.

对于单旋操作: 假设旋转点为 x, 他原来的父亲为 y, 那么 $T_i + A_i$, 也就是实际运行时间 + 势差, 为 1(旋一下, 设为单位时间)$+R'_y + R'_x$(加 $'$ 号的表

示旋转后的)$-R_y - R_x$(由于其它点的 R 值都没变, 所以势差就是这两个点的 R 值的差):

$$
\begin{aligned}
&1 + R'_y + R'_x - R_y - R_x \\
&\le 1 + R'_x - R_x \qquad \text{(因为 } R'_y \le R_y\text{, 因为 y 在旋转后原来属于 x 的子孙没有了)} \\
&\le 1 + 3 \times (R'_x - R_x) \qquad \text{(这放大下面有用, 现在先忽略吧)}
\end{aligned}
$$

对于双旋的一条直线情况, 即先转父亲的情况: 设旋转点为 x, 父亲为 y, 爷爷为 z:

$$
\begin{aligned}
&2 + R'_z + R'_y + R'_x - R_z - R_y - R_x \\
&\le 2 + R'_z + R'_y - R_y - R_x \quad (R'_x = R_z\text{, 画图便知}) \\
&\le 2 + R'_z + R'_x - 2R_x
\end{aligned}
$$

(用到什么性质都是画图看看 size 就能看出来的, 一下类似的就不解释了)

又因为

$$R_x + R\text{'}_z - 2R\text{'}_x = \log(size[x] \div size'[x]) + \log(size[z] \div size'[x]) \le -2$$

(因为真数和 ≤ 1 的两个关于 2 的对数之和 ≤ -2, 因为 log 上凸)

即 $2R'_x - R_x - R'_z \le 2$ 把 2 带入刚才的不等式,

$$
\begin{aligned}
&2 + R'_z + R'_x - 2R_x \\
&\le 2R'_x - R_x - R'z + R'_z + R'_x - 2R_x \\
&= 3(R'_x - R_x)
\end{aligned}
$$

对于最后一种折线的双旋操作, 和上一种情况的分析完全一样.

每一次旋转操作, $T_i + A_i$ 的贡献都 $\le 3(R'_x - R_x)$(除了单旋), 由于上一次转完的 x 的位置就是下一次转开始之前 x 的位置, 所以把整个过程的 $3(R'_x - R_x)$ 累加, 中间量都抵消, 最后剩下 $3 \times (R_{root} - R_x)$, 因为转到最后 x 变成 root 了, 再加上单旋的 1(单旋可是 $1 + 3(R'_x - R_x)$, 根据刚才的分析), 所以 $A_i + T_i \le 3 \times (R_{root} - R_x) + 1 \le 3 \times (\log n - R_x) \le 3 \times \log n = O(\log n)$.

还要算上 $\sum A$ 的下界, 所以进行 m 次操作的总时间 $\sum T = \sum P - \sum A \le O(m \log n) + O(n \log n)$. 至此 splay 的均摊复杂度就证完了。

3 Treap

Treap 是另外一种非常好写的平衡树。但是它记录了一个额外的随机权值（重量）用于维护它的平衡性质。

3.1 Treap=Tree+Heap

首先，顾名思义，Treap 是一个既满足二叉查找树性质又满足堆性质的一棵平衡树。确切地说，它存储的键值按照二叉查找树的顺序（小的在左，大的在右），而随机的那个重量满足堆的性质（小的在上，大的在下）。这样，如果随机函数足够好的话，那它期望是平衡的。

查找等一切操作基本与 Splay 一样，但是由于没有伸展操作，自然就没有最后 Splay 到根这个操作了。而且 Treap 提取区间时需要另外一种方法。

3.2 基于旋转的的 Treap

考虑到将一个点按照二叉查找树的方式插入到一个 Treap 中时，它的 Heap 性质可能会被破坏。因为新插入的节点一定是叶子，但是它的重量可能比较小，这时我们可以通过旋转它直到它应该到的位置上。根据证明，这个操作的期望旋转次数是 $O(1)$ 的，确切地说，是 2 次 [3]。

3.3 重量平衡树

旋转的几个弊端 前文简单地提到了旋转的几种劣势。首先是它的形态不确定: 如果每个点中存储了它子树中所有数的集合，那么 update 的过程将变得痛苦。所幸，Treap 有期望的旋转次数做复杂度保证。另外就是旋转为了方便需要记录节点的父亲，这会导致数据结构难以可持久化。

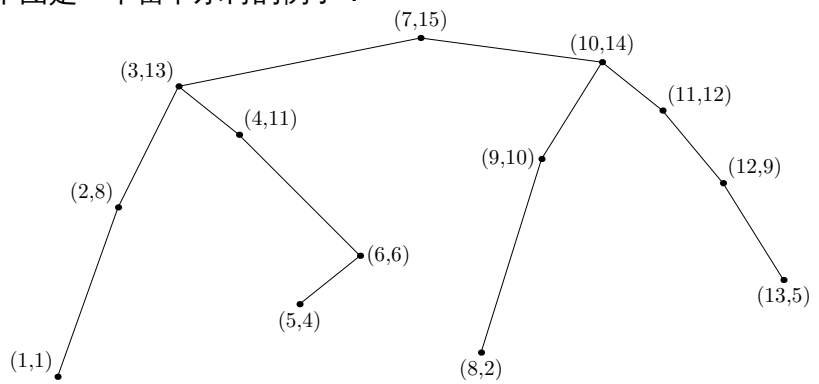
重量平衡树的概念 重量平衡树并不基于旋转，而是某种奇特的方式来维护自己的形态（例如：暴力重构子树的替罪羊树等等）。

Treap 的重量平衡 Treap 重量平衡是基于他的一个特性：节点都确定下来之后，Treap 的形态就应该是固定的。这样我们就可以做到快速分裂/合并 (split/merge) 一棵 Treap。分裂的时候我们按照一个权值进行分裂，将小于等于它的，放到分裂后左手那棵树里面; 大于它的，放到右手那棵树里

面。显然每次分裂的最坏复杂度是与深度相关的，而 Treap 的期望深度是 $\log n$。每次 merge 时需要保证 merge 的两颗子树，左边那棵的权值都严格小于等于右边那个（也就是说 merge 的两个子树是可以通过将 merge 得到的再 split 一下得到的）。这样的话我们以两棵子树中重量较小的那个子树的根当根，那么一棵子树就确定了（就是做根的那棵树的左/右子树），那么将它的右/左子树和另外一棵树递归合并就好了。可以看到，merge 的复杂度也是基于深度的。

3.4 笛卡尔树

笛卡尔树是指二维平面上有一些点，他们的横坐标两两不同，然后按照某种方式连成的一棵二叉查找树。具体的连边方式是这样的：按照高度顺序处理每个点，那么它会按照它的横坐标将它所在区域划分成两半，它的左儿子就是平面左半边中最高的那个（易知不会有比它高的存在，因为比它高的都早早处理掉了），右儿子同理。那么递归构造，它左儿子的所在区域的右边界被定为这个点的横坐标，左边界就是这个点的左边界。右儿子依旧同理。需要说明的是，统一区域内遇到高度相同的点，优先考虑左边的那个。下图是一个笛卡尔树的例子：



很显然，Treap 就是一棵以键值作为横坐标、重量作为纵坐标的笛卡尔树。之所以研究这一点，是因为我们有时需要根据插入的节点建立一棵 Treap（比如说我们要在序列中插入一段数，那么应该先把它们建成一棵小 Treap，再与原树 merge 起来），而笛卡尔树有一种应用单调栈的线性时间建立方法，具有很高的实用价值。

首先我们按照横坐标从左往右进行处理，同时维护一个单调栈，保证栈里的元素高度递减。每次进来一个新的节点时，将栈里比它低的元素都弹出，并将它的左儿子设为最后一个弹出的节点，而且将先弹出的节点设为

它之后弹出的那个节点的右儿子即可。为了保证结束性，可以在最右侧加入一个高度为正无穷的点。

3.5 核心代码

我们这里只写 merge、split 和一个随机数生成器，其他的和 Splay 基本没有区别了。提取区间的时候，只需要两次 split 即可。

经过了笔者和其小伙伴何琦的测试，发现最简单好用的还是线性同余法生成随机数（给定随机种子 x_0, a, b, p，每次返回一个随机数 $x_i = (ax_{i-1} + b) \bmod p$）。鉴于系统随机数太慢，所以我们还是自己给出一套随机种子自行实现好了。笔者的随机种子都是小伙伴的生日，模的 p 是一个质数，来自于某个神犇的生日 + 姓氏

```
struct MyRand
{
        int a,b,p; long long x;
        MyRand(){a=810;b=1102;p=1992122981;x=617;}
        int rand(){return x=(a*x+b)%p;}
};
void  split(node *p,node* &p1,node* &p2,int w)
{
        if(!p){p1=p2=NULL; return;}
        if (p->val<=pos) {p1=p; split(p1->s[1],p1->s[1],p2,w); p1->update();}
        else {p2 = p; split(p2->s[0],p1,p2->s[0],w); p2->update();}
}
node *merge(node *p,node *q)
{
        if(!p) return q; if(!q) return p;
        node *p1; p->pushdown(); q->pushdown();
        if(p->lap<q->lap) (p1=p)->s[1]=merge(p->s[1],q);
        else (p1=q)->s[0]=merge(p,q->s[0]);
        p1->update(); return p1;
}
```

4 实用技巧

在本节内容中，我们会简单介绍一些与具体数据结构无关的，但是在后文中会得到广泛利用的几种常用小技巧（或者说一些概念、一些方法）。由于这些东西不止会应用在接下来要讲的具体问题中，所以单独拎成一节，以供参考。

4.1 静态内存回收

众所周知，在 Pascal 和 C 语言中 new 运算的复杂度都是非常高的。因为它们要在内存中寻找一个没有被利用的空间。delete 操作也不低。而且尤其在我们需要删除一棵子树的时候，写一个后序遍历来释放内存会显得得不偿失，但是如果只是将这棵子树抛弃而不释放，经常会造成内存不够用。为了解决这些问题，我们将使用内存回收器来进行内存管理。

首先，为了避免 new 运算新建节点造成时间开销过大，我们可以通过预估所需的节点数，实现把它们新建成一个数组（栈）就可以了。每次需要 new 一个结点的时候，就从栈里返回一个新节点的地址就可以了。这种方式是我所提倡的，相比所谓"数组模拟指针"的方法，这种直接使用数组的地址直接进行指针运算的方式更为直观、方便，不容易出错。而且直接对指针进行操作是比在数组内进行操作要快的（C++）。我们将这个数组称为 "内存池"。

注意到内存池这个栈其实并不是用一个结点就从栈中弹出一个，恰恰相反，而是每新建一个点我们就向栈中 "压入" 一个点。这样，我们没有办法很轻易地进行删除。不过无所谓，我们再开一个指针类型的栈就好了——初始时栈是满的，每个栈中的指针指向内存池中的一个地址。每次新建点的时候，就从栈中弹出一个指针，删除的时候就压入一个指针——问题就解决了。

特别需要注意的是，删除一个结点的时候没有必要递归将整棵子树压栈。只需要在弹出这个节点的时候将它可能存在的左右儿子压栈就可以了。

4.2 数据结构的嵌套

我们介绍了那么多种数据结构，加以练习就可以用它们处理一些简单的问题了。但是有些复杂的问题将会应用到数据结构的复杂组合（比如线段树套平衡树、平衡树套平衡树等等等……）数据结构的嵌套有一些简单通用的规则，理解了这些之后，树套树无非就是写两种树而已，没有必要对此望而生畏。

首先介绍两种下标形式：权值下标和位置下标。以线段树为例，位置下标就是我们前文所说的那种 $[L, R]$ 表示第 L 到第 R 个数，而权值线段树存储的则是树值位于 L 到 R 之间的那些数。而对于平衡树来说，权值下标就是就是以数的权值作为关键字，而位置下标就是以数的位置作为关键字。

最简单的树套树就是线段树套平衡树：位置线段树套权值平衡树就是

线段树的每个节点内用一棵平衡树来存储这一区间内的权值，以此类推。而对于内层平衡树，我推荐使用 Treap 而非 Splay。因为 Treap 根本不需要哨兵。

4.3 数据结构的可持久化

可持久化数据结构是函数式编程中一个重要的部分。所谓可持久化，就是这一数据结构的历史版本均能查询。说白了，就是把所有的修改操作改为新建节点。

考虑到每一步操作的修改都只会影响一部分节点，那么新版本的其他节点直接与旧版本共用就行了，以节约和复制信息所带来的不必要开支。事实上，内存一直是可持久化数据结构的最大瓶颈。

所谓的历史版本并不一定是以时间作为阶段的。例如，下文中即将介绍的由黄嘉泰神犇提出的 “主席树” 就是以将固定的序列每次插入末尾的一个值作为修改状态的。

Part II

动态序列问题

我们已经有了那么多种数据结构武器，就可以来解决一些实际的问题了。我们先从经典的区间第 k 大数说起，然后分析一些经典例题。

大部分题（除了最后一个）都可以很容易地在 OJ 上找到，所以数据范围和具体的题目要求就略去不说了。

5 第 k 大数

关于这一问题，有一个非常经典的二叉查找树结构叫做 “划分树”，由于它不是很好写，加上方便快捷的主席树的出现，它已经被诸多人所遗忘，成为时代的眼泪。有兴趣的同学可以自己寻找资料进行研究学习（后面的 5.3 节中也会简单介绍）。

我们现在将这个问题形式化：给定一个长度为 n 的数列 $\{a_n\}$，每次询问一个区间 $[L, R]$ 和一个整数 k，求出这个区间内的第 k 大值。

5.1 不带修改的

直接解决上面这个问题的最简单情况，可以通过应用划分树的方法。我们这里介绍另外一种可持久化权值线段树的做法。使用权值线段树记录权值位于当前区间内的有多少个数。我们从序列的第 1 项开始可持久化地插入在权值线段树中，这样我们就得到了 n 棵权值线段树。我们找出 $L-1$ 号线段树和 R 号线段树，在线段树上随走势进行二分。每次判断两颗线段树中当前权值区间内的右子树的数（计为 w）是否够 k 个，如果够，就向右二分（两个线段树一起），否则将 k 减去 w 再向左二分，直至找到一个叶子，那么这个值就是要找的第 k 大数。

这种做法的核心在于在线段树上二分。我们通过可持久化数据结构快速定位两个区间，然后随着线段树的走势进行二分。由于 size 满足区间减法，所以这样可以找到最终的解。

权值线段树的空间复杂度是 O(权值范围) 的。由于不同的权值数不可能超过数列长度 n，所以我们可以进行离散化处理。而每次插入一个数只会沿一条路经修改到叶子，所以新建的节点不超过 $\log n$。所以总的空间复杂度是 $(n \log n)$。但是我们为什么不用可持久化权值 Treap 呢？那样不就没

有离散化的问题了么？注意到插入一个结点之后的 Treap 就和原来的长得不一样了。而由于我们是两个线段树同步进行二分，所以应该保证每棵树长得都是一模一样，而仅有上面记的信息不同。

5.2 带修改的

将之前的问题强化。每次可能修改某个位置上的值为一个新值，询问区间内的第 k 大。

之前我们使用的方法就是将若干棵线段树组织成部分和的形式。带修改的部分和的一般处理方式是树状数组。所以我们将外层的部分和改为树状数组即可。每次修改的时候在 $O(\log n)$ 棵线段树中插入并删除一个点。在查询的时候使用 $O(\log n)$ 棵线段树一起进行二分即可。这样做的时间复杂度是每次 $O(\log^2 n)$。

由于我们的权值线段树具有修改，所以应改将修改的值一起进行离散化。但是如果强制在线怎么办？不要紧，我们有动态建点的线段树。我们不一定将线段树的所有节点一次性建出来，而是随用随建。可以证明这样做节点的数量并没有增多。我们就将复杂度转移到了新建节点的过程中。而利用之前提到的内存回收器就可以轻松解决。

这种方法太神了（主席树）。但是我们仍然要介绍一种 Naive 的方法——即位置线段树套权值平衡树。修改是暴力的修改每一个影响到的线段树节点、查询是在外面套一个二分。具体地说就是二分答案 w，看当前区间内小于 w 的有多少个。这么做是每次 $O(\log^3 n)$ 的。而且平衡树常数大。但是这样的做法非常有参考价值（他要是问一些别的数列问题呢？总能有一个思路吧。）

观察一下这两种做法。方法二的时间复杂度多了一个 $\log n$。这是因为它多了一个显式的二分。而主席树中顺着权值线段树走下来的过程是和二分等价的。基于这种想法考虑，我们想办法将外层改为权值线段树，内层采用位置平衡树。这样顺着走势二分就可以将复杂度优化为 $O(\log^2 n)$ 了。

5.3 带插入的

我们不妨再强化一下问题。我们可以在某个序列的两个数中间插入一个数。询问区间第 k 大。这样一下子主席树好象就萎靡不振了。因为一旦插入了某个数，它后面的所有版本都得重置。（当然，如果不强制在线的话就另当别论）

那么怎么办呢？有办法！我们可以使用重量平衡树套权值线段树！但是经过测算这种方式消耗内存巨大。一种比较好的方法是重新请出划分树神犇。划分树维护的是是 $\log n$ 层序列。第一层就是原序列。每一次选出当前序列的中位数，将前一半数在相对位置不改变的前提下放到左儿子的序列中，另一半放到右儿子里。然后记录每个节点的前面有多少被放到左边即可(观察一下，这些数据结构的本质真的都是一样的。）我们这么来想，这不就是一个权值线段树套位置平衡树嘛！然后每个位置平衡数的节点记录一下它左子树到权值线段树左子树的节点有多少个就行了。

那么我们直接使用这样一个线段树套平衡树式的划分树就可以水掉这个题了。我没讲划分树的查询，读者可以自行思考。

6 一些练习题

理论基础就介绍到这里，接下来是一些练习题了。选取的题目都非常简单，没有任何难度，只为了让大家熟悉基本算法。

6.1 第 k 大系列

POJ 2104 不带修改的第 k 小值模板题，记得内存回收

BZOJ 1901 带修改的第 k 小值模板题。

BZOJ 3065 带插入的第 k 小值模板题。

BZOJ 3110 这道题值得说一说。题意描述的不是特别清楚。大概就是说，一共有 n 个格子，两个操作。一个是在一段格子中的每个格子扔进去一个数，另一个是查询一段格子内数的第 k 大是多少。首先我们在外层弄一棵权值线段树，然后权值线段树里面是位置线段树，记录每个格子上有多少数。然后直接顺着权值平衡树二分就可以了。考虑到权值满足了区间减法，所以外层线段树可以替换为权值树状数组，但是这样就导致必须离散化。树状数组也是可以随走势二分而做到 $O(\log n)$ 而非 $O(\log^2 n)$ 的复杂度，但是写起来不是很舒服。

6.2 NOI2007 项链工厂

这是一道经典的线段树水题[3]。

题目大意：维护一个环状的项链，项链上每个珠子具有一个颜色。具有如下几个操作：将项链顺时针旋转一个值，将项链以 1 号位为对称轴对称反转，交换某两个珠子的颜色，查询某个区间的连续颜色段数，查询整个项链的颜色段数。

注意这里的查询区间 $[1, n]$ 是和查询整段有区别的，因为区间查询时不考虑首尾重复。如果没有旋转、反转等操作，这题就是一个彻头彻尾的线段树水题：维护每个区间的不同颜色段数、最左端的颜色和最右端的颜色即可。加上了反转和旋转操作也无所谓，因为项链中的点的相对顺序不会改变。我们在外面记录两个标记，一个用来记录当前的 1 号位置是原数列中的哪个珠子，另一个记录项链是否被反转过，然后每次查询的区间就可以计算出来了。查询的时候需要特判是否包含 1 号和 n 号节点的连接处。

由于特判端点颜色的存在，我们在查询的时候会额外返回一下两个端点的颜色，会增加处理的细节。但是我们发现可以干脆地直接返回一个结点，它由这一段的查询结果合并得到。这样我们只需要额外写一个 merge 函数就可以了，这种小技巧会经常用到，而且总能帮忙解决大麻烦。

6.3 NOI2005 维护数列

这是一道经典的平衡树水题[4]。

题目大意：维护一个序列，支持将一段数修改为一个值、翻转一段数、插入一段数、删除一段数、查询一段数的和、查询最大子段和（子段不能为空）。所有的操作都是平衡树的基本操作——旋转区间、查询或打懒标记。惟一的难点就在于最大连续子段和操作。为了简化问题我们假定这个连续子段可以为空（和为 0），我们在每个节点额外维护这个子树的最大连续子段和 maxs、左侧的连续和最大的一段 maxl 以及右侧的 maxr。那么维护 maxs 只有三种可能：左子树的 maxs、右子树的 maxs 和左子树的 maxr+ 节点自身的值 + 右子树的 maxl，而 maxl 和 maxr 都只有两种情况。我们只需要在 update 里面做文章就可以了，其余的就没有任何难度了。由于原题中保证子段不能为空，那我们额外维护一个子树中的最大值 maxc，当查询的 maxs 为空时返回 maxc 就可以啦！

[3]学长们曾经每天比赛这题谁能写进十分钟之内。

[4]一些有爱人士把这道题称为维修数列

6.4 NOI2003 文本编辑器

这是一道经典的块状链表水题，但是我们用平衡树水掉它。

题目大意：维护一个字符串，支持几个操作。包括将光标移动至一个位置、在光标处插入一个字符串、删除光标后的若干个字符、输出光标后的若干个字符、将光标前移或后移一个字符。插入和删除之后的光标位置不变。

我们只需要在外面开一个变量记录一下当前的光标位置即可，其余的操作都是平衡树的基本操作了。

6.5 TJOI2014 电源插排（w007878 强化版）

这是一道 “经典” 的数据结构水题。

题目大意：有一个长度为 n 的插座，共有 q 次操作，操作有三种：可能会使用一个空插座：选择空插座的原则是，取连续最长的空插座段中中间的插座，假如最长的空插座有很多段，那么取最右侧的; 如果长度为偶数，取中间靠右的那个。也可能某个人把他之前使用的插座腾出来。询问要求询问某个区间内使用过的插座有多少个。**w007878 的强化：要求强制在线。**

数据范围：$n \leq 10^9, q \leq 10^5$

假如没有强制在线，考虑到 n 的范围过大，我们考虑将其离散化之后使用树状数组就可以了。离散化正是本题的核心。由于我们需要最长连续空段最靠右的那个，很显然可以使用一个堆来进行维护。将一个插座腾空时需要将它和它的左右侧合并成为一个更长的空段，那么就需要维护每个用过的点做的左右是不是空段，如果是空段的话多长等等。这是我们就需要用一个 map 或者链表进行维护。而且在堆中会有删除操作。离散化之后反过来模拟一遍就可以了。但是强制在线怎么办？用 Pascal 怎么办？总不能为了实现一个 map 再写一棵平衡树吧？

没关系，既然 map 已经用到平衡树了，那我们不妨直接用一个 Splay 一了百了。用一个平衡树来维护插座，但是我们不能把每个点都建出来，那样太浪费空间。我们用每个点表示一个连续空段或用过的单点，维护这个段的最靠右的最长的连续段是哪个就行了。每次使用新插座的时候直接把我们维护的最长那一段劈成三段（三个节点），中间那个节点标记为用过，两边仍旧是两个连续空段。删除的时候，找到删除的点和它的前趋后继，把它们合并成一个新的整段就可以了。

但是这么写是不是有点麻烦？没关系，一个动态建点线段树来拯救你。

上文中提到了动态建点线段树的使用方法，我们不离散区间，直接维护最长空段即可。为了维护最长连续空段我们还需要维护左侧最长连续段和右侧最长连续段（详见维护数列中最大子段和的处理方式）。然后..... 没有然后了。

Part III

动态树

动态树问题在 OI 中还是近几年兴起的。很多资料中将“动态树”特指为 link-cut-tree 这种数据结构，而事实上动态树是一类问题的通称，说白了就是上文中各种数列的维护改到树上来做，例如查询链上和、子树和、修改节点数值等等。处理这一类问题的最基本思路就是将树上的问题重新转化到链上，然后套用上文中的各种方法进行解决，而这种转化往往是基于给树上的节点按照某种顺序标号，然后每次处理标号连续的一段或若干段。而另外一些涉及数的形态变化的问题则难以利用这种 “标号” 的思路进行维护（因为一个简单的修改可能会导致整棵树的标号出现不规则的更改）。为了解决这些问题，以 Robert E. Tarjan 为首的科学家们研究出各种实用的、具有良好理论复杂度和较大常数的数据结构，直接对树的形态进行维护。本部分会简要介绍 DFS 序和树链剖分两种标号法和 link-cut-tree 的应用，理论部分的最后会提及 LCT 解决子树问题的一个方案（自调整 TopTree）。然后以几道经典题为本文画上句号。

7 DFS 序与树链剖分

7.1 在海棠花开的年代

DFS 序，就是对一棵有根树进行 DFS（深度优先搜索），然后记录下来访问节点的顺序，将节点按照这个顺序进行标号。显然每个点为根的子树的标号是连续的一段。然后我们就可以在排好序的节点上搞一棵线段树啦！很轻易地就能解决子树的询问和整个子树的修改等问题了。这个东西比较简单，本文就不展开介绍了。

树链剖分[5]，是指将树上的边划分为轻边和重边，连在一起的重边组成一条重链，每个点属于且仅属于一条重链，每条轻边连接的一定是两条重链。然后我们将点标号，使得重链上的点的标号是连续的。这样的话我们就可以处理链上的问题了：树上的一条链必然是一段重链、一条轻边、一段重链、一条轻边这样跳过来的，那么我们需要处理的便是剖分序上的若干连续段。(请注意，我们默认为树上所有的权值都在点上。如果涉及到信息在边

[5]某有爱人士为了方便起函数名，将这种方法称为 “Begonia”，意为 “海棠”

上的，将一条边拆成一个点向边的端点连边就可以了)。为了让操作的复杂度尽可能低，我们需要让每次操作跨越的段数尽可能少。

我们发现，如果每次选取某个点的具有最大的子树 size 的儿子作为它的重边儿子（假如某个点没有重边与之相连，我们将它看作一个仅有一个点的重链)，其余的看作轻边儿子，这样某对点的路径上跨越的重链的数量大致是 $O(\log n)$ 的（极端情况下是一棵完全二叉树，这时恰被划为 $\log n$ 条重链)。

那么我们就可以在 $O(\log n)\times$ 数据结构的复杂度的时间内完成一次修改或查询了！而且剖分的原理决定了它在树越不平衡的时候实际的效果越好，而且常数极小。毫不夸张地说应用剖分的 $O(n\log^4 n)$ 的算法有时会比 $O(n\log^2 n)$ 的算法具有更优的速度。

7.2 Begoina 的实现

树链剖分通过进行两次 DFS 来实现：第一次计算出每个节点的子树大小（size)、深度（dpt)、重儿子（就是具有最大子树 size 的那个儿子 hs)。第二次通过第一次 DFS 的信息计算出每个节点所在的重链的顶（hp）以及剖分序 (order) 和每个节点在剖分序中所处的位置 (ps)。

由于剖分处理的问题的规模往往比较庞大，加上递归函数的开销不容忽视，所以我们往往在这两次 DFS 中使用手动栈。而两遍 DFS 没有任何难度，就不多做说明，直接贴代码了[6]。

[6]这是我压过行的版本，好记好写难理解，所以最好还是自己实现一个然后按照自己的习惯编程

```
inline void Begonia(int v0)
{
        static int d[maxn],now[maxn],i,j,last,tot;
        memcpy(now,a,sizeof a);
        for(size[d[last=1]=v0]=dpt[v0]=1;last;)
        {
                if(!(j=now[i=d[last]]))
                {
                        if((--last)&&(size[d[last]]+=size[i],size[hs[d[last]]]<size[i]))
                                hs[d[last]]=i;
                        continue;
                }
                if(E[j].v!=p[i])dpt[E[j].v]=dpt[p[d[++last]=E[j].v]=i]+(size[E[j].v]=1);
                now[i]=E[j].next;
        }
        for(int i=1;i<=n;++i) hp[i]=i;
        memset(now,0xFF,sizeof now);
        for(order[tot=ps[v0]=1]=d[last=1]=v0;last;)
        {
                if(!(j=now[i=d[last]])){--last;continue;}
                if(j==-1)
                {
                        if(hs[i])hp[d[++last]=order[ps[hs[i]]=++tot]=hs[i]]=hp[i];
                        now[i]=a[i];continue;
                }
                if(E[j].v!=hs[i]&&E[j].v!=p[i])d[++last]=order[ps[E[j].v]=++tot]=E[j].v;
                now[i]=E[j].next;
        }
}
```

7.3 用剖分找 LCA

树链剖分最基本的应用就是寻找 LCA（最近公共祖先）了。在实际的查询中，查询两个点的那条链往往就是通过分别查询这两个节点到 LCA 的链得到的。所以在修改和查询时都需要进行这一步。

找节点 LCA 的步骤非常简单：若两个点处在同一条重链上，深度低的那个就是 LCA。如果不在同一条重链上，那么把**所在重链顶较深的那个**跳到它的重链顶的轻父亲上，直至他们在一条重链上。而跳经的这段路程，就是查询的路径中包含的一段。

7.4 复合问题

假如说一道题既包含链上询问修改又包括子树询问修改怎么办？是不是同时维护 DFS 序和剖分序呢？不是的！因为我们在两种标号中打上的懒标记不能互相下放，会导致两个数据结构维护的信息不一致，这是大大不行的。其实仔细观察就可以发现剖分就是一种特定的 DFS 序，它可以直接处理前面那种 DFS 序能处理的一切问题，而且剖分也没多难写到哪去，所以我们再也不需要无序的 DFS 序[7]了（这也就是我为何不详细介绍的原因）。

虽然我们在剖分的时候假定剖分的是有根树，但是对于自由树，我们随便指定一个当根就可以了。对于会换根的有根树，可能会处理得比较棘手。但是注意到换根操作只会影响到子树操作。假定我们要找出某个非根节点的子树（根节点的子树就不说了），我们先将它想象成根，原来的根就在以它的某个儿子为根的子树上，然后查询的范围就是整棵树除去那棵子树了。这样我们只需要判断一下根和点在剖分序中的相对位置就好了——如果根在原树中在查询的这个点的上方，那么该怎么查怎么查不会影响，否则就是整棵树除去根所在的，原树中查询的那个点的子树中包含新根的那个儿子个子树即可。

8 link-cut-tree

感觉树链剖分很厉害，但是它在处理树的形态变化的问题中就不那么好用了（因为边的结构变化之后轻重链的机制就失效了）。而且就算它再快，剖分套线段树也是 $O(n\log^2 n)$ 的。接下来我们介绍的这种数据结构是由 Tarjan 等人发明的、可以处理更一般的动态树问题的，由势能分析得到均摊复杂度为 $O(n\log n)$ 的 link-cut-tree。

8.1 动态维护的剖分

link-cut-tree 维护的是一片森林，能支持的操作包括：查询/修改某条链上的信息、砍掉一条边、在两个点之间连边（需要保证连接之后不可以出环）、换根。

lct 的实质是动态维护树的一中链剖分——即将一棵树划分成若干条不相交的重链，每条重链使用一棵 Splay 维护，没有重边的单点也视作一条

[7]当然 DFS 序还有很多种，像 Euler Tour 之类的还是很有价值的，但这不在本文的讨论范围内

重链。每次通过将轻边打成重边的方式来打通某条特定的链。假如我们想得到点 u 到点 v 的链，方法是把它们打成重链，然后在重链中舍去两端的东西。最终的目的就是让 u 和 v 同处在一棵 Splay 中，且这棵 Splay 只包含 u 到 v 这条路径上的点，这样我们就可以进行操作了。

8.2 具体的做法

刚刚把道理说完，接下来就要仔细地考虑细节：究竟要怎么实现这样一个东西。我们用这种方式表现轻边的连接方式：让每棵 Splay 的根的父亲指向这段重边的轻边父亲，而它却不是轻边父亲的左右儿子中的任何一个。对于同一棵重链，我们约定靠上的点在左边，靠下的在右边。那么我们定义打通操作是将某个 Splay 的根和它的轻父亲打成同一棵 Splay，那么它父亲原来的右儿子就会退化成为轻儿子。我们成这个操作为 splice。很轻松地我们可以将一个点一直 splice 到根，我们把通到根的操作称为 expose。将一个点 expose 到根之后，将它的右子树断掉，再将剩下的 Splay（只包含它到根的那条链）reverse 一下，就可以将这个节点变为根啦。

这样我们很简单地做完了换根 (reroot) 操作。提取链的操作更加容易：reroot(u); reroot(v); 那么此时的点 v 为根的 Splay 就为所求了。然后遇到的问题就都迎刃而解了。

还有两个操作 link 和 cut 没有处理，其实都是将两个节点 reroot 之后直接断/连接边就可以了。

8.3 代码很简单

除了 Splay 的基本操作，剩下的代码基本都是一行的事。有一点需要注意的是在将一个点 Splay 到根之前需要用一个栈把它头顶上的懒标记统统下放，大概就是这样子：

```
void splay(node *p)
{
        int stop; stack[stop=1]=p;
        for(node *q=p;!q->isroot();q=q->p)stack[++stop]=q->p;
        for(int i=stop;i;--i)stack[i]->pushdown();
        play(p);//原来的splay
}
void splice(node *p){splay(p->p);p->p->s[1]=p;p->update();splay(p);}
void expose(node *p){splay(p); while(p->p!=lct[0]) splice(p);}
void reroot(node *p){expose(p); p->s[1]=NULL;p->update();p->reverse();}
```

expose 中判断是否已经成为树根用到了一步判断 p->p!=lct[0]。每个节点都对应 lct 的一个点而且雷打不变，利用这个性质我们可以使用 lct 数组作为

一个天然的内存池，用 0 号位当作树根的父亲，起到了哨兵的作用。splice 中的 isroot () 函数返回的是一个结点是不是它所在 Splay 的根，只需要判断它是不是它父亲的轻儿子就可以了。

9 子树问题与 Toptree

刚刚我们介绍的 LCT 仅限于处理链上的操作和形态的变化。lct 处理子树操作一直是一件麻烦事。学术界一般有三种处理方法 [9]：第一种是将 LCT 中的每一个点拆成三个然后处理，第二种是同时维护一棵 Euler Tour Tree，第三种则是使用 Toptree。前面两种方法我都不怎么了解，所以只向大家介绍第三种方法——TopTree[8]。

9.1 基于节点收缩的动态树

TopTree 是这样一种数据结构：它基于两种操作 rake 和 compress。rake 是指将一个叶子节点与它的某个兄弟合并（可以理解为，转过去），compress 是指将一个度数为 2 的点和它两侧的点 “缩成” 一个。可以证明，这样做可以使节点逐步减少并最终成为一个点。我们的树用于维护这种收缩关系（也就是说，每个树上的节点实际上代表收缩之后的一团点）。我们在树中找到一条主链，称为 compress 链，将主链以外的那些子树递归地合并成一个点并 rake 到主链上，然后将整棵主链 compress 起来。

compress 链上的内容用一棵 Splay（称为 Compress Tree）来维护，compress 树上的每个节点可能有许多的小子树 rake 起来的，那我们在每个节点上额外维护一棵 Rake Tree 来管理这些节点。每个 Rake Tree 上的节点又是由一棵 Compress Tree 组成的，如此交替往复。

TopTree 的基本操作仍旧是将某个节点打通至主链和换根。区别就在于节点需要在 Rake Tree 和 Compress Tree 中相互转化。

9.2 仍然是剖分的思想

其实这个 “主链” 的存在预示着这仍旧是一种剖分，而所谓的 rake 也不过就是虚边儿子的另外一种说法。所以我们不妨直接在每个节点上多用

[8]本文所提 Top Tree，并非真正意义上的 Top Tree，而是 Tarjan 等人改良的自调整型 Toptree 经过何琦与我的简化版本，还算较为实用

一棵 Splay 来维护它的所有轻边儿子就可以了。注意我们的这个虚边 Splay 是不需要顺序的。

然后我们在打标记的时候就可以给 lct 上的点和虚边 Splay 上的点直接打标记、互相下放了。

10 练习题们

10.1 SDOI2011 染色 (BZOJ2243)

给定一棵树，支持两个操作：将某一条链上的节点染成一个颜色，查询某条链上的连续颜色段数。

看题就知道是最简单的剖分套线段树，剖分后参照前文项链工厂的处理方法即可。

10.2 tree(伍一鸣)

给定一棵树，支持将某段的点加上一个数、乘上一个数或是 link/cut 两条边，询问链上的点权值和。

仍然是一道模板题。由于数的形态发生变化所以我们只能使用 LCT 来维护。W 唯一需要注意的地方就是这题有两个懒标记，两个懒标记会互相影响，所以下放的时候要注意先后顺序。

10.3 WC2006 水管局长数据加强版（BZOJ2594)

维护一个 n 个点 m 条边的无向图，支持两个操作 (共 Q 个)。一个是断开某一条边，另一个是询问两个点之间的一条路径，使得路径上最长的边尽可能短，输出这个最小值。数据范围：$N \leq 100000$, $M \leq 1000000$, $Q \leq 100000$

通过简单的分析就可以发现路径永远是当前图中最小生成树上的路径。问题就变成了动态维护支持删边的最小生成树。但是我们发现最小生成树上只有 n-1 条边，而断掉这其中一条就必定会进来另外一条，但是我们很难维护应该进来哪一条。但是我们将问题反过来，用一棵 LCT 维护最小生成树。假如加入一条边进来的话，我们判断原 LCT 中的这两点之间的路径上的最大值时候比新加进来的边大，如果是的话就用新边替掉它。这样我们可以轻易维护不断加边的动态最小生成树，然后把问题转为离线处理就可以了。

Part IV

鸣谢

特别感谢我的老师滕伟和小伙伴何琦、耿越对我的帮助。这篇文章引用了许多神牛的成果，由于时间原因，而这些引用都是未经授权的。有任何文章内容、理论错误、错别字以及版权问题都欢迎与我联系。

References

[1]《统计的力量》，清华大学 张昆玮，2011-07

[2]《数据结构与算法分析（C++ 描述第三版)》，Mark Allen Weiss, 张怀勇等 译，人民邮电出版社，2007

[3]《算法导论（第三版)》，Thomas H. Cormen Charles E Leiserson Ronald L. Rivest Clifford Stein, 殷建平等 译，机械工业出版社，2011

[4]《算法竞赛入门经典：训练指南》，刘汝佳 陈锋，清华大学出版社，2012

[5]《重量平衡树和后缀平衡树在信息学奥赛中的应用》，杭州外国语学校 陈立杰，2013

[6]《splay 和 link cut tree 的时间复杂度的均摊分析》，上海交通大学 高宇

[7] Algorithms *Fourth Edition*,Robert Sedgewick, 人民邮电出版社影印版，2010

[8] Implementation of Dynamic Trees with In-Subtree Operations, Tomasz Radzik, Dapartment of Computer Science,King's College London, 1998

[9] Self-Adjusting Top Trees, Robert E. Tarjan Renato F. Werneck, 2004