数据结构选讲

xz153531

雅礼中学

August 9, 2019

Preface

今天主要是讲一下有关于一些数据结构的知识点

Preface

今天主要是讲一下有关于一些数据结构的知识点

该课件的题目不是很多，主要是一些知识点的梳理

Preface

今天主要是讲一下有关于一些数据结构的知识点

该课件的题目不是很多，主要是一些知识点的梳理

讲课内容较为基础，大佬们可以开启养生模式了

~~我应该是第一个开始讲人话的~~

Preface

今天主要是讲一下有关于一些数据结构的知识点

该课件的题目不是很多，主要是一些知识点的梳理

讲课内容较为基础，大佬们可以开启养生模式了

~~我应该是第一个开始讲人话的~~

由于本人太菜~~&职业口胡选手~~，所以出现错误请大家多多指正

Preface

今天主要是讲一下有关于一些数据结构的知识点

该课件的题目不是很多，主要是一些知识点的梳理

讲课内容较为基础，大佬们可以开启养生模式了

~~我应该是第一个开始讲人话的~~

由于本人太菜~~&职业口胡选手~~，所以出现错误请大家多多指正

Q：为什么这个课件里面没有线段树？

Preface

今天主要是讲一下有关于一些数据结构的知识点

该课件的题目不是很多，主要是一些知识点的梳理

讲课内容较为基础，大佬们可以开启养生模式了

~~我应该是第一个开始讲人话的~~

由于本人太菜~~&职业口胡选手~~，所以出现错误请大家多多指正

Q：为什么这个课件里面没有线段树？

A：因为线段树专题归巨佬~~Johnvember~~讲

树状数组基础用法

相信普通的树状数组大家都会，所以我会介绍一些进阶知识

树状数组基础用法

相信普通的树状数组大家都会，所以我会介绍一些进阶知识

单点修改区间查询

树状数组基础用法

相信普通的树状数组大家都会，所以我会介绍一些进阶知识

单点修改区间查询

区间修改单点查询

树状数组基础用法

相信普通的树状数组大家都会，所以我会介绍一些进阶知识

单点修改区间查询

区间修改单点查询，这个东西实际上和上面那个东西是一样的，只是把维护原数的和改成维护差分数组的和

树状数组基础用法

相信普通的树状数组大家都会，所以我会介绍一些进阶知识

单点修改区间查询

区间修改单点查询，这个东西实际上和上面那个东西是一样的，只是把维护原数的和改成维护差分数组的和

区间修改区间查询

树状数组基础用法

相信普通的树状数组大家都会，所以我会介绍一些进阶知识

单点修改区间查询

区间修改单点查询，这个东西实际上和上面那个东西是一样的，只是把维护原数的和改成维护差分数组的和

区间修改区间查询，我们考虑查询区间$[1, R]$，设差分数组为d_i

树状数组基础用法

相信普通的树状数组大家都会，所以我会介绍一些进阶知识

单点修改区间查询

区间修改单点查询，这个东西实际上和上面那个东西是一样的，只是把维护原数的和改成维护差分数组的和

区间修改区间查询，我们考虑查询区间$[1,R]$，设差分数组为d_i

$$Ans=\sum_{i=1}^{R}a_i=\sum_{i=1}^{R}(R-i+1)\times d_i=(R+1)\times\sum_{i=1}^{R}d_i-\sum_{i=1}^{R}i\times d_i$$

树状数组基础用法

相信普通的树状数组大家都会，所以我会介绍一些进阶知识

单点修改区间查询

区间修改单点查询，这个东西实际上和上面那个东西是一样的，只是把维护原数的和改成维护差分数组的和

区间修改区间查询，我们考虑查询区间$[1,R]$，设差分数组为d_i

$$Ans=\sum_{i=1}^{R}a_i=\sum_{i=1}^{R}(R-i+1)\times d_i=(R+1)\times\sum_{i=1}^{R}d_i-\sum_{i=1}^{R}i\times d_i$$

我们用两个树状数组维护即可

一些小技巧

- 维护后缀和

  我们把普通树状数组的修改和询问的循环都倒过来就可以了

一些小技巧

- 维护后缀和

  我们把普通树状数组的修改和询问的循环都倒过来就可以了

  我等下会解释原因

一些小技巧

- 维护后缀和

  我们把普通树状数组的修改和询问的循环都倒过来就可以了

  我等下会解释原因

- 清空

  我们可以利用时间戳来实现

一些小技巧

- 维护后缀和

  我们把普通树状数组的修改和询问的循环都倒过来就可以了

  我等下会解释原因

- 清空

  我们可以利用时间戳来实现

  修改的时候我们就把经过的点的时间戳更新，查询的时候忽略掉时间戳"过时"了的点

一些小技巧

- 维护后缀和

  我们把普通树状数组的修改和询问的循环都倒过来就可以了

  我等下会解释原因

- 清空

  我们可以利用时间戳来实现

  修改的时候我们就把经过的点的时间戳更新，查询的时候忽略掉时间戳"过时"了的点

- 在BIT上二分

一些小技巧

- 维护后缀和
  我们把普通树状数组的修改和询问的循环都倒过来就可以了
  我等下会解释原因
- 清空
  我们可以利用时间戳来实现
  修改的时候我们就把经过的点的时间戳更新，查询的时候忽略掉时间戳"过时"了的点
- 在BIT上二分
- 用BIT替代一些简单的线段树
  要打标记就直接标记永久化，但标记必须要有可减性

一些小技巧

- 维护后缀和
  我们把普通树状数组的修改和询问的循环都倒过来就可以了
  我等下会解释原因
- 清空
  我们可以利用时间戳来实现
  修改的时候我们就把经过的点的时间戳更新，查询的时候忽略掉时间戳"过时"了的点
- 在BIT上二分
- 用BIT替代一些简单的线段树
  要打标记就直接标记永久化，但标记必须要有可减性
- 线性初始化BIT

一些小技巧

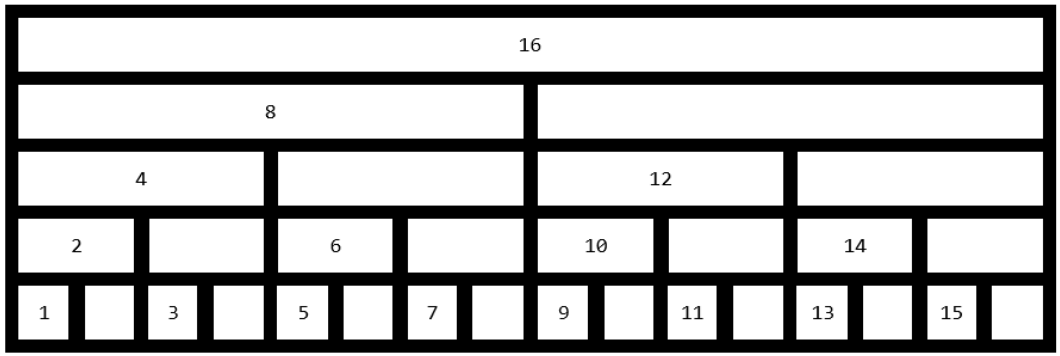
- 维护后缀和
  我们把普通树状数组的修改和询问的循环都倒过来就可以了
  我等下会解释原因
- 清空
  我们可以利用时间戳来实现
  修改的时候我们就把经过的点的时间戳更新，查询的时候忽略掉时间戳"过时"了的点
- 在BIT上二分
- 用BIT替代一些简单的线段树
  要打标记就直接标记永久化，但标记必须要有可减性
- 线性初始化BIT
- 树状数组代替树链剖分

BIT上二分

BIT可以看成是没有右儿子的线段树，如图

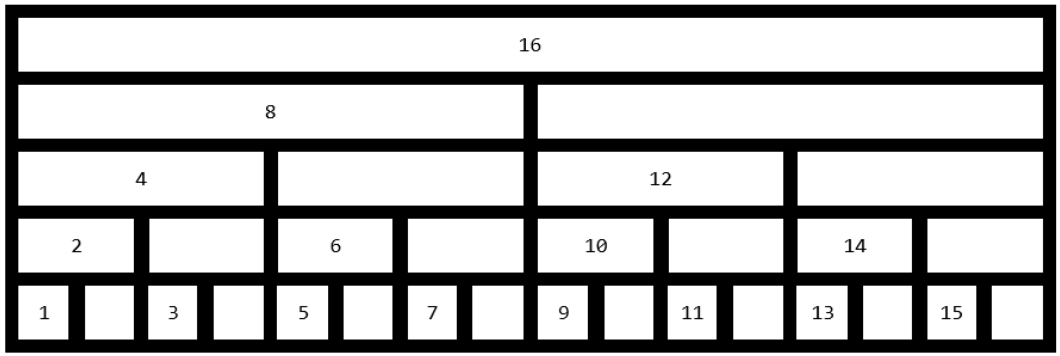
BIT上二分

BIT可以看成是没有右儿子的线段树，如图



BIT上二分

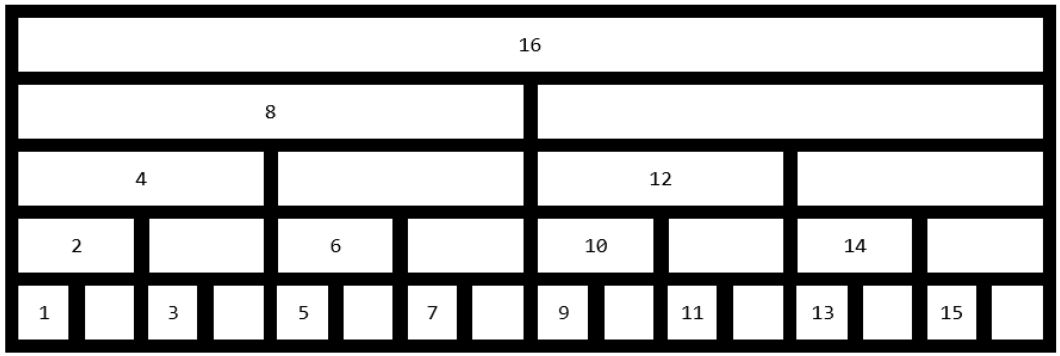
BIT可以看成是没有右儿子的线段树，如图



所以我们可以把一个点的右斜下方那个点看成：当自己是左儿子时的右儿子

BIT上二分

BIT可以看成是没有右儿子的线段树，如图



所以我们可以把一个点的右斜下方那个点看成：当自己是左儿子时的右儿子

如上图，假设当前点8作为左儿子，那12就是作为右儿子，我们称12是8的"兄弟"(单方向的) ~~我自己起的名字~~

BIT上二分

所以对于一个点x而言，它的左儿子为$x-(lowbit(x)>>1)$，"兄弟"为$x+(lowbit(x)>>1)$

BIT上二分

所以对于一个点x而言，它的左儿子为$x-(lowbit(x)>>1)$，"兄弟"为$x+(lowbit(x)>>1)$

我们可以类比在线段树上二分，当条件超过左儿子的贡献的时候，跳到"兄弟"的位置，否则继续跳到左儿子的位置

BIT上二分

所以对于一个点x而言，它的左儿子为$x-(lowbit(x)>>1)$，"兄弟"为$x+(lowbit(x)>>1)$

我们可以类比在线段树上二分，当条件超过左儿子的贡献的时候，跳到"兄弟"的位置，否则继续跳到左儿子的位置

但是，BIT二分的当前节点相当于线段树二分的左儿子节点，所以我们要拿条件和自己的贡献作比较

BIT上二分

所以对于一个点x而言，它的左儿子为$x-(lowbit(x)>>1)$，"兄弟"为$x+(lowbit(x)>>1)$

我们可以类比在线段树上二分，当条件超过左儿子的贡献的时候，跳到"兄弟"的位置，否则继续跳到左儿子的位置

但是，BIT二分的当前节点相当于线段树二分的左儿子节点，所以我们要拿条件和自己的贡献作比较

最后在叶子节点上的那次二分要特殊处理一下

BIT上二分

所以对于一个点x而言，它的左儿子为$x-(lowbit(x)>>1)$，"兄弟"为$x+(lowbit(x)>>1)$

我们可以类比在线段树上二分，当条件超过左儿子的贡献的时候，跳到"兄弟"的位置，否则继续跳到左儿子的位置

但是，BIT二分的当前节点相当于线段树二分的左儿子节点，所以我们要拿条件和自己的贡献作比较

最后在叶子节点上的那次二分要特殊处理一下

注意要保证值域是2的次幂

BIT代替线段树

代替的前提条件前面也已经讲过了，就直接进入正文吧

BIT代替线段树

代替的前提条件前面也已经讲过了，就直接进入正文吧

我们还是考虑之前那个经典问题：区间加区间求和

BIT代替线段树

代替的前提条件前面也已经讲过了，就直接进入正文吧

我们还是考虑之前那个经典问题：区间加区间求和

首先无论是修改还是查询，我们都要把区间拆成两个前缀来操作

BIT代替线段树

代替的前提条件前面也已经讲过了，就直接进入正文吧

我们还是考虑之前那个经典问题：区间加区间求和

首先无论是修改还是查询，我们都要把区间拆成两个前缀来操作

然后我们考虑标记永久化的线段树实现过程

BIT代替线段树

代替的前提条件前面也已经讲过了，就直接进入正文吧

我们还是考虑之前那个经典问题：区间加区间求和

首先无论是修改还是查询，我们都要把区间拆成两个前缀来操作

然后我们考虑标记永久化的线段树实现过程，这部分就直接口述算了

BIT代替线段树

代替的前提条件前面也已经讲过了，就直接进入正文吧

我们还是考虑之前那个经典问题：区间加区间求和

首先无论是修改还是查询，我们都要把区间拆成两个前缀来操作

然后我们考虑标记永久化的线段树实现过程，这部分就直接口述算了

最后我们看一下代码

线性初始化

只需要考虑线段树build的过程就可以了，直接上代码

线性初始化

只需要考虑线段树build的过程就可以了，直接上代码

```
for (int i = 1; i <= n; i++) {
    bit[i] += a[i];
    int fa = i + lowbit(i);
    if (fa <= n) bit[fa] += bit[i];
}
```

BIT代替树链剖分

当我们处理一些较为简单的树链上的操作时，我们可以用树状数组来做

BIT代替树链剖分

当我们处理一些较为简单的树链上的操作时，我们可以用树状数组来做

这样的话复杂度可以少一个log，并且代码也更容易写

BIT代替树链剖分

当我们处理一些较为简单的树链上的操作时，我们可以用树状数组来做

这样的话复杂度可以少一个log，并且代码也更容易写

需要注意的是，这种做法只支持具有可减性的操作

BIT代替树链剖分

当我们处理一些较为简单的树链上的操作时，我们可以用树状数组来做

这样的话复杂度可以少一个log，并且代码也更容易写

需要注意的是，这种做法只支持具有可减性的操作

我们来看一下可以支持哪些东西

BIT代替树链剖分

- 链修改点查询

  我们可以把链变为几条到根的链，然后就可以转化为单点修改子树查询

BIT代替树链剖分

- 链修改点查询

  我们可以把链变为几条到根的链，然后就可以转化为单点修改子树查询
- 点修改链查询

  我们可以和上面一样的转换，然后就可以转化为子树修改单点查询

BIT代替树链剖分

- 链修改点查询

  我们可以把链变为几条到根的链，然后就可以转化为单点修改子树查询
- 点修改链查询

  我们可以和上面一样的转换，然后就可以转化为子树修改单点查询
- 链修改子树查询

  我们可以用树状数组来维护子树权值和以及子树里每一个权值乘上所在点深度的和，修改就直接单点修改

BIT代替树链剖分

- 链修改点查询

  我们可以把链变为几条到根的链，然后就可以转化为单点修改子树查询
- 点修改链查询

  我们可以和上面一样的转换，然后就可以转化为子树修改单点查询
- 链修改子树查询

  我们可以用树状数组来维护子树权值和以及子树里每一个权值乘上所在点深度的和，修改就直接单点修改

  然后查询的时候就利用差分，把乘积的和减去权值和乘上子树根的深度即可

BIT代替树链剖分

- 链修改点查询

  我们可以把链变为几条到根的链，然后就可以转化为单点修改子树查询
- 点修改链查询

  我们可以和上面一样的转换，然后就可以转化为子树修改单点查询
- 链修改子树查询

  我们可以用树状数组来维护子树权值和以及子树里每一个权值乘上所在点深度的和，修改就直接单点修改

  然后查询的时候就利用差分，把乘积的和减去权值和乘上子树根的深度即可
- 子树修改链查询

  我们修改的时候可以直接对于子树减去子树根深度乘上修改的权值，另外再维护一下子树修改的权值和

BIT代替树链剖分

- 链修改点查询

  我们可以把链变为几条到根的链，然后就可以转化为单点修改子树查询
- 点修改链查询

  我们可以和上面一样的转换，然后就可以转化为子树修改单点查询
- 链修改子树查询

  我们可以用树状数组来维护子树权值和以及子树里每一个权值乘上所在点深度的和，修改就直接单点修改

  然后查询的时候就利用差分，把乘积的和减去权值和乘上子树根的深度即可
- 子树修改链查询

  我们修改的时候可以直接对于子树减去子树根深度乘上修改的权值，另外再维护一下子树修改的权值和

  然后查询就直接单点查询，用查询点的深度乘上当前点被修改的权值和减去之前维护的权值乘和和就可以了

简介

平衡树是维护权值集合的利器，它可以在较快的时间内支持以下操作

简介

平衡树是维护权值集合的利器，它可以在较快的时间内支持以下操作

插入/删除一个权值，查询一个权值在集合里的排名，查询集合里某一排名的权值，或是查询一个权值的前继后继

简介

平衡树是维护权值集合的利器，它可以在较快的时间内支持以下操作

插入/删除一个权值，查询一个权值在集合里的排名，查询集合里某一排名的权值，或是查询一个权值的前继后继

常见的平衡树有：Splay，Treap，Scapegoat Tree(替罪羊树)，Red-Black Tree(红黑树)，AVL Tree，Leafy Tree

简介

平衡树是维护权值集合的利器，它可以在较快的时间内支持以下操作

插入/删除一个权值，查询一个权值在集合里的排名，查询集合里某一排名的权值，或是查询一个权值的前继后继

常见的平衡树有：Splay，Treap，Scapegoat Tree(替罪羊树)，Red-Black Tree(红黑树)，AVL Tree，Leafy Tree

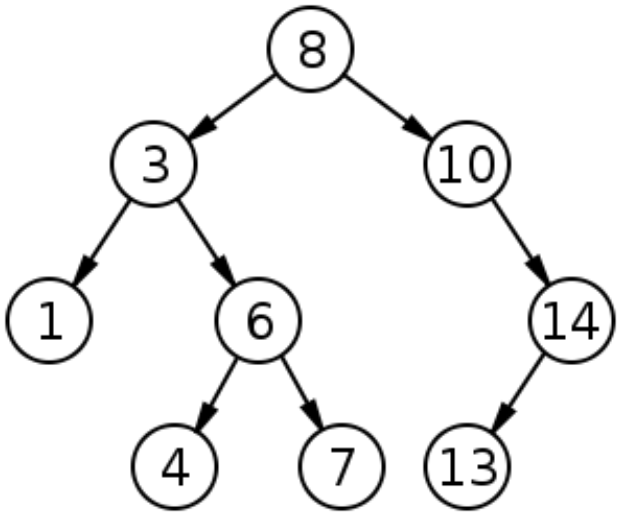
今天，我主要介绍一下上面前三种比较主流的平衡树

二叉搜索树

性质：一个节点x左子树所有点的权值都比x的权值小，右子树所有点的权值都比x的权值大

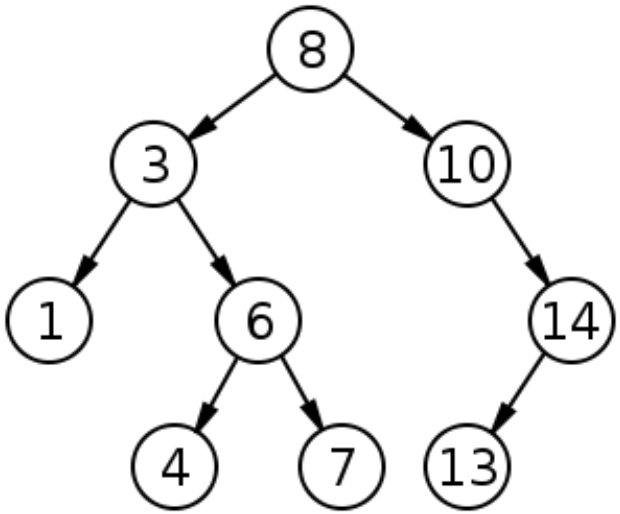
二叉搜索树

性质：一个节点x左子树所有点的权值都比x的权值小，右子树所有点的权值都比x的权值大



二叉搜索树

性质：一个节点x左子树所有点的权值都比x的权值小，右子树所有点的权值都比x的权值大



这是我今天要讲的三种平衡树的基础

Splay

Splay又称伸展树、自适应查找树，可以支持区间操作

Splay

Splay又称伸展树、自适应查找树，可以支持区间操作

它的核心操作就是$splay(x, y)$，表示把x节点旋转到y的下方

Splay

Splay又称伸展树、自适应查找树，可以支持区间操作

它的核心操作就是$splay(x, y)$，表示把x节点旋转到y的下方

每对一个节点进行操作的时候就直接把它旋转到根

Splay

Splay又称伸展树、自适应查找树，可以支持区间操作

它的核心操作就是$splay(x, y)$，表示把x节点旋转到y的下方

每对一个节点进行操作的时候就直接把它旋转到根

如果是区间操作就可以利用$splay$操作，把对应的区间放到一个子树当中去操作

Splay

Splay又称伸展树、自适应查找树，可以支持区间操作

它的核心操作就是$splay(x,y)$，表示把x节点旋转到y的下方

每对一个节点进行操作的时候就直接把它旋转到根

如果是区间操作就可以利用$splay$操作，把对应的区间放到一个子树当中去操作

另外，这个东西最(yi)好(ding)要写双旋。至于为什么，我就直接挂链接算了 传送门

Splay

Splay又称伸展树、自适应查找树，可以支持区间操作

它的核心操作就是$splay(x,y)$，表示把x节点旋转到y的下方

每对一个节点进行操作的时候就直接把它旋转到根

如果是区间操作就可以利用$splay$操作，把对应的区间放到一个子树当中去操作

另外，这个东西最(yi)好(ding)要写双旋。至于为什么，我就直接挂链接算了 传送门

现在我们来看一下代码 ~~大家应该都会吧~~

SuperMemo

题目蓝链

给你一个长度为N的序列，你需要维护m个以下6个操作：

- $ADD\ x\ y\ d$，将区间$[x,y]$的每一个数都加上d
- $REVERSE\ x\ y$，翻转区间$[x,y]$
- $REVOLVE\ x\ y\ t$，区间$[x,y]$向右旋转t次
- $INSERT\ x\ p$，在第x个数后插入数字p
- $DELETE\ x$，把序列的第x个数删去
- $MIN\ x\ y$，求区间$[x,y]$的最小值

$n,m \leq 10^5$

SuperMemo

这实际上就是一道板子题...

SuperMemo

这实际上就是一道板子题...

你只需要对于每一个节点维护一个子树最小值、加标记还有翻转标记

SuperMemo

这实际上就是一道板子题...

你只需要对于每一个节点维护一个子树最小值、加标记还有翻转标记

另外对于3操作，你先对于t模一下区间长度，然后直接把区间后t个数拿出来接到前面就可以了

Treap

Treap的本质就是二叉搜索树加上堆

Treap

Treap的本质就是二叉搜索树加上堆

我们给每一个点随机分配一个重量，使重量满足堆的性质，同时使每个点原本的权值满足二叉搜索树的性质

Treap

Treap的本质就是二叉搜索树加上堆

我们给每一个点随机分配一个重量，使重量满足堆的性质，同时使每个点原本的权值满足二叉搜索树的性质

对于之前所说的那些平衡树的操作，Treap的复杂度均为：期望$\mathcal{O}(\log n)$

算法实现

Treap维护权值的时候一般会把相同的权值放在同一个节点上

算法实现

Treap维护权值的时候一般会把相同的权值放在同一个节点上

所以一个Treap节点需要维护以下信息

算法实现

Treap维护权值的时候一般会把相同的权值放在同一个节点上

所以一个Treap节点需要维护以下信息

左右儿子、权值、该权值的个数、该点随机分配到的重量、该点的子树大小

算法实现

Treap维护权值的时候一般会把相同的权值放在同一个节点上

所以一个Treap节点需要维护以下信息

左右儿子、权值、该权值的个数、该点随机分配到的重量、该点的子树大小

我们现在来看一下这道题的Treap实现方法

非旋转Treap

俗称FHQ Treap ~~FHQ类似于CDQ, 是一个人的名字~~，支持可持久化和区间操作

非旋转Treap

俗称FHQ Treap ~~FHQ类似于CDQ, 是一个人的名字~~，支持可持久化和区间操作

虽然区间操作Splay也可以支持，但是Splay不能可持久化，因为它的复杂度是均摊的

非旋转Treap

俗称FHQ Treap ~~FHQ类似于CDQ, 是一个人的名字~~，支持可持久化和区间操作

虽然区间操作Splay也可以支持，但是Splay不能可持久化，因为它的复杂度是均摊的

非旋转Treap能可持久化，因为Treap的复杂度是期望log的

非旋转Treap

俗称FHQ Treap ~~FHQ类似于CDQ, 是一个人的名字~~，支持可持久化和区间操作

虽然区间操作Splay也可以支持，但是Splay不能可持久化，因为它的复杂度是均摊的

非旋转Treap能可持久化，因为Treap的复杂度是期望$\log$的

这个可持久化其实和主席树的思想差不多，就是把被改动了的节点开一个新的节点来存，这样每次操作显然只会增加$\mathcal{O}(\log n)$个节点

非旋转Treap

俗称FHQ Treap ~~FHQ类似于CDQ，是一个人的名字~~，支持可持久化和区间操作

虽然区间操作Splay也可以支持，但是Splay不能可持久化，因为它的复杂度是均摊的

非旋转Treap能可持久化，因为Treap的复杂度是期望$\log$的

这个可持久化其实和主席树的思想差不多，就是把被改动了的节点开一个新的节点来存，这样每次操作显然只会增加$\mathcal{O}(\log n)$个节点

其实我个人感觉普通Treap应该也能可持久化 ~~只是从没人写过~~

算法实现

非旋转Treap的主要操作都是用merge和split来实现的

算法实现

非旋转Treap的主要操作都是用merge和split来实现的

$merge(x, y)$表示把以x为根的平衡树和以y为根的平衡树合并

算法实现

非旋转Treap的主要操作都是用merge和split来实现的

$merge(x, y)$表示把以x为根的平衡树和以y为根的平衡树合并

$split(x, k)$表示把以a为根的平衡树拆成前k项组成的平衡树和前剩余项组成的平衡树，当然也可以按照权值分割

算法实现

非旋转Treap的主要操作都是用merge和split来实现的

$merge(x, y)$表示把以x为根的平衡树和以y为根的平衡树合并

$split(x, k)$表示把以a为根的平衡树拆成前k项组成的平衡树和前剩余项组成的平衡树，当然也可以按照权值分割

如果要进行区间操作，就利用这两个东西搞一搞就好了

~~职业口胡选手已上线~~

算法实现

非旋转Treap的主要操作都是用merge和split来实现的

$merge(x, y)$表示把以x为根的平衡树和以y为根的平衡树合并

$split(x, k)$表示把以a为根的平衡树拆成前k项组成的平衡树和前剩余项组成的平衡树，当然也可以按照权值分割

如果要进行区间操作，就利用这两个东西搞一搞就好了

~~职业口胡选手已上线~~

至于更精妙的代码实现，就直接对着代码讲就好了 链接1 链接2

替罪羊树

这是一种实现与其它平衡树有着较大差别的平衡树

替罪羊树

这是一种实现与其它平衡树有着较大差别的平衡树

在某一时刻，如果一棵子树不满足平衡条件，就直接暴力重构

替罪羊树

这是一种实现与其它平衡树有着较大差别的平衡树

在某一时刻，如果一棵子树不满足平衡条件，就直接暴力重构

平衡条件为：左子树$\leq \alpha \times$根大小，并且右子树$\leq \alpha \times$根大小

替罪羊树

这是一种实现与其它平衡树有着较大差别的平衡树

在某一时刻，如果一棵子树不满足平衡条件，就直接暴力重构

平衡条件为：左子树$\leq \alpha \times$根大小，并且右子树$\leq \alpha \times$根大小

如果是删除操作，就直接打标记，只要在重构的时候把这些点去掉就可以了

替罪羊树

这是一种实现与其它平衡树有着较大差别的平衡树

在某一时刻，如果一棵子树不满足平衡条件，就直接暴力重构

平衡条件为：左子树$\leq \alpha \times$根大小，并且右子树$\leq \alpha \times$根大小

如果是删除操作，就直接打标记，只要在重构的时候把这些点去掉就可以了

暴力即是优雅 —— 知乎某答主

替罪羊树

关于α的取值问题，α一般取0.75，但必须要在$(0.5, 1)$里面

替罪羊树

关于α的取值问题，α一般取0.75，但必须要在$(0.5, 1)$里面

如果询问比较多，可以适当调小α。如果修改比较多，可以适当调大α

替罪羊树

关于α的取值问题，α一般取0.75，但必须要在$(0.5, 1)$里面

如果询问比较多，可以适当调小α。如果修改比较多，可以适当调大α

然后我们来看一下代码 传送门

finger search

- Splay
  在n个点的Splay上执行m次插入/删除/访问操作的均摊复杂度是$\mathcal{O}(m+\sum \log d_i)$，其中$d_i$为每次操作的元素与上一次操作的元素的排名之差

finger search

- Splay

  在n个点的Splay上执行m次插入/删除/访问操作的均摊复杂度是$\mathcal{O}(m+\sum \log d_i)$，其中$d_i$为每次操作的元素与上一次操作的元素的排名之差

- Treap

  两个排名相差d的元素在Treap上的路径长度是期望$\mathcal{O}(\log d)$

启发式合并

普通的平衡树启发式合并都是两个log的

启发式合并

普通的平衡树启发式合并都是两个log的

但是我们在启发式合并的时候，如果使用的数据结构可以使用finger search，我们就可以把较小的那个平衡树按升序/降序插入到另外一棵中

启发式合并

普通的平衡树启发式合并都是两个log的

但是我们在启发式合并的时候，如果使用的数据结构可以使用finger search，我们就可以把较小的那个平衡树按升序/降序插入到另外一棵中

可以证明这样的总复杂度是一个log的，下面我来证明一下

Proof

考虑合并一个大小为n和一个大小为m的平衡树$A, B(m \leq n)$，那么我们对于较小的那棵平衡树B，按顺序插入到较大的那棵平衡树A里面

Proof

考虑合并一个大小为n和一个大小为m的平衡树$A, B(m \leq n)$，那么我们对于较小的那棵平衡树B，按顺序插入到较大的那棵平衡树A里面

那么设B中相邻的两个元素插入到A中的排名差为为d_i，那么除了第一次插入的复杂度为$\mathcal{O}(\log n)$，其余插入的复杂度之和为$\sum \log d_i$，并且$\sum d_i \leq n + m$

Proof

考虑合并一个大小为n和一个大小为m的平衡树$A, B(m \leq n)$，那么我们对于较小的那棵平衡树B，按顺序插入到较大的那棵平衡树A里面

那么设B中相邻的两个元素插入到A中的排名差为为d_i，那么除了第一次插入的复杂度为$\mathcal{O}(\log n)$，其余插入的复杂度之和为$\sum \log d_i$，并且$\sum d_i \leq n + m$

由于log是凸函数，所以当所有d_i相等时，$\sum \log d_i$取最大值，所以这样一次平衡树合并的总复杂度是$\mathcal{O}(\log n + m + m \log \frac{n+m}{m})$的

Proof

考虑合并一个大小为n和一个大小为m的平衡树$A, B(m \leq n)$，那么我们对于较小的那棵平衡树B，按顺序插入到较大的那棵平衡树A里面

那么设B中相邻的两个元素插入到A中的排名差为为d_i，那么除了第一次插入的复杂度为$\mathcal{O}(\log n)$，其余插入的复杂度之和为$\sum \log d_i$，并且$\sum d_i \leq n + m$

由于log是凸函数，所以当所有d_i相等时，$\sum \log d_i$取最大值，所以这样一次平衡树合并的总复杂度是$\mathcal{O}(\log n + m + m \log \frac{n+m}{m})$的

这个复杂度前面的那个$\log n + m$，由于总的合并过程中只会有$n-1$次合并，并且每一个元素最多只会被合并$\log n$次，所以总体复杂度是$\mathcal{O}(n \log n)$的

Proof

我们考虑上面那个式子的后面那部分，我们设每一个元素每次合并完时候所处的集合大小(包括第1次合并之前的集合)为$s_1, s_2, \cdots, s_k$

Proof

我们考虑上面那个式子的后面那部分，我们设每一个元素每次合并完时候所处的集合大小(包括第1次合并之前的集合)为$s_1, s_2, \cdots, s_k$

那么其贡献为$\mathcal{O}(\sum_{i=2}^{k} \log \frac{s_i}{s_{i-1}}) = \mathcal{O}(\log n)$

Proof

我们考虑上面那个式子的后面那部分，我们设每一个元素每次合并完时候所处的集合大小(包括第1次合并之前的集合)为$s_1, s_2, \cdots, s_k$

那么其贡献为$\mathcal{O}(\sum_{i=2}^{k} \log \frac{s_i}{s_{i-1}}) = \mathcal{O}(\log n)$

我们对于每一个元素都这么考虑，那么总体复杂度就为$\mathcal{O}(n \log n)$

Proof

我们考虑上面那个式子的后面那部分，我们设每一个元素每次合并完时候所处的集合大小(包括第1次合并之前的集合)为$s_1, s_2, \cdots, s_k$

那么其贡献为$\mathcal{O}(\sum_{i=2}^{k} \log \frac{s_i}{s_{i-1}}) = \mathcal{O}(\log n)$

我们对于每一个元素都这么考虑，那么总体复杂度就为$\mathcal{O}(n \log n)$

结合两个部分，使用了finger search的平衡树启发式合并的复杂度为$\mathcal{O}(n \log n)$

树链剖分

我们通常指的树链剖分是轻重链剖分，除此之外还有长链剖分之类的

树链剖分

我们通常指的树链剖分是轻重链剖分，除此之外还有长链剖分之类的

我们对于一棵树的边可以分为两类：轻边和重边

树链剖分

我们通常指的树链剖分是轻重链剖分，除此之外还有长链剖分之类的

我们对于一棵树的边可以分为两类：轻边和重边

对于任意一个点，它与它所有儿子中$size$最大的儿子之间连的边为重边，与其余儿子连的边为轻边

树链剖分

我们通常指的树链剖分是轻重链剖分，除此之外还有长链剖分之类的

我们对于一棵树的边可以分为两类：轻边和重边

对于任意一个点，它与它所有儿子中$size$最大的儿子之间连的边为重边，与其余儿子连的边为轻边

若干个重边连在一起形成一条重链

树链剖分

我们通常指的树链剖分是轻重链剖分，除此之外还有长链剖分之类的

我们对于一棵树的边可以分为两类：轻边和重边

对于任意一个点，它与它所有儿子中$size$最大的儿子之间连的边为重边，与其余儿子连的边为轻边

若干个重边连在一起形成一条重链

那么会有一个结论，就是树上任意一个点到根结点之间最多会有$\log n$条重链和$\log n$条轻边

树链剖分

我们通常指的树链剖分是轻重链剖分，除此之外还有长链剖分之类的

我们对于一棵树的边可以分为两类：轻边和重边

对于任意一个点，它与它所有儿子中$size$最大的儿子之间连的边为重边，与其余儿子连的边为轻边

若干个重边连在一起形成一条重链

那么会有一个结论，就是树上任意一个点到根结点之间最多会有$\log n$条重链和$\log n$条轻边

现在我们来证一下这个结论

Proof

因为对于任意一个点，每经过一条轻边，它的子树大小至少会翻一倍

Proof

因为对于任意一个点，每经过一条轻边，它的子树大小至少会翻一倍

因为大小最多会翻倍$\log n$次，所以最多会经过$\log n$条轻边

Proof

因为对于任意一个点，每经过一条轻边，它的子树大小至少会翻一倍

因为大小最多会翻倍$\log n$次，所以最多会经过$\log n$条轻边

又因为两条轻边之间最多只有1条重链，所以得证

树链剖分

有了这个性质之后，我们发现树上任意两个之间只会有$\mathcal{O}(\log n)$条重链

树链剖分

有了这个性质之后，我们发现树上任意两个之间只会有$\mathcal{O}(\log n)$条重链

我们在给这棵树每一个节点分配dfn序的时候，要让重链的dfn序连续

树链剖分

有了这个性质之后，我们发现树上任意两个之间只会有$\mathcal{O}(\log n)$条重链

我们在给这棵树每一个节点分配dfn序的时候，要让重链的dfn序连续

这样的话，再给这棵树的dfn序套一个线段树，那么我们对于一条链修改/查询的复杂度就是两个log的

树链剖分

有了这个性质之后，我们发现树上任意两个之间只会有$\mathcal{O}(\log n)$条重链

我们在给这棵树每一个节点分配dfn序的时候，要让重链的dfn序连续

这样的话，再给这棵树的dfn序套一个线段树，那么我们对于一条链修改/查询的复杂度就是两个log的

那为什么动态的LCT复杂度比静态的树剖要低?

树链剖分

有了这个性质之后，我们发现树上任意两个之间只会有$\mathcal{O}(\log n)$条重链

我们在给这棵树每一个节点分配dfn序的时候，要让重链的dfn序连续

这样的话，再给这棵树的dfn序套一个线段树，那么我们对于一条链修改/查询的复杂度就是两个log的

那为什么动态的LCT复杂度比静态的树剖要低?

因为树剖还可以维护子树的修改操作

树链剖分

有了这个性质之后，我们发现树上任意两个之间只会有$\mathcal{O}(\log n)$条重链

我们在给这棵树每一个节点分配dfn序的时候，要让重链的dfn序连续

这样的话，再给这棵树的dfn序套一个线段树，那么我们对于一条链修改/查询的复杂度就是两个log的

那为什么动态的LCT复杂度比静态的树剖要低?

因为树剖还可以维护子树的修改操作

其实如果没有子树操作，我们只要把树剖套的线段树改成Splay，复杂度就是一个log的了，复杂度分析和LCT一样

LCT

LCT全称Link-Cut-Tree

LCT

LCT全称Link-Cut-Tree

树剖可以解决静态的树上的问题，但是树的形态要是会变化呢?

LCT

LCT全称Link-Cut-Tree

树剖可以解决静态的树上的问题，但是树的形态要是会变化呢?

于是就有了LCT这个算法

LCT

LCT全称Link-Cut-Tree

树剖可以解决静态的树上的问题，但是树的形态要是会变化呢?

于是就有了LCT这个算法

我们就不具体介绍LCT的基本函数了 ~~因为Small Train: 不是大家都会了吗~~

LCT

LCT全称Link-Cut-Tree

树剖可以解决静态的树上的问题，但是树的形态要是会变化呢?

于是就有了LCT这个算法

我们就不具体介绍LCT的基本函数了 ~~因为Small Train: 不是大家都会了吗~~

我们就以介绍LCT的应用为主，如果有忘记了的看下发的之前LCT专题的课件

基本性质

- LCT的本质就是把原树分为若干条互不相交的实链，剩下的边为虚边

基本性质

- LCT的本质就是把原树分为若干条互不相交的实链，剩下的边为虚边
- 每一条实链上所有点的深度互不相同，并且每条链都用一个Splay来维护

基本性质

- LCT的本质就是把原树分为若干条互不相交的实链，剩下的边为虚边
- 每一条实链上所有点的深度互不相同，并且每条链都用一个Splay来维护
- 每一个Splay的中序遍历是按原树的深度递增的

基本性质

- LCT的本质就是把原树分为若干条互不相交的实链，剩下的边为虚边
- 每一条实链上所有点的深度互不相同，并且每条链都用一个Splay来维护
- 每一个Splay的中序遍历是按原树的深度递增的
- 不同的Splay之间靠虚边连接，每一条虚边只会在儿子处记录父亲

基本性质

- LCT的本质就是把原树分为若干条互不相交的实链，剩下的边为虚边
- 每一条实链上所有点的深度互不相同，并且每条链都用一个Splay来维护
- 每一个Splay的中序遍历是按原树的深度递增的
- 不同的Splay之间靠虚边连接，每一条虚边只会在儿子处记录父亲

~~都是一些很显然的东西...~~

求lca

其实这个东西很简单了，比如我们要求$lca(x,y)$

求lca

其实这个东西很简单了，比如我们要求$lca(x,y)$

我们只需要先$access\ x$，再$access\ y$。它们的lca就是在$access\ y$的时候，所经过的最后一条虚边的父亲

求lca

其实这个东西很简单了，比如我们要求$lca(x,y)$

我们只需要先$access\ x$，再$access\ y$。它们的lca就是在$access\ y$的时候，所经过的最后一条虚边的父亲

```
inline int access(int x) {
    int res = 0;
    for (int lst = 0; x; lst = x, x = fa(x))
        res = x, splay(x), rs(x) = lst, push_up(x);
    return res;
}
```

维护生成树

维护一个图的最小或最大生成树

维护生成树

维护一个图的最小或最大生成树

我们对于每一条边都建一个点，连到两个点上，点权为原来的边权

维护生成树

维护一个图的最小或最大生成树

我们对于每一条边都建一个点，连到两个点上，点权为原来的边权

但是要注意LCT只能维护森林，有时候可能要时光倒流

维护边双

我们在加入新边的时候，如果两个端点已经在一个连通块内，就会形成一个新的环，环上的点就会属于一个边双

维护边双

我们在加入新边的时候，如果两个端点已经在一个连通块内，就会形成一个新的环，环上的点就会属于一个边双

我们可以新建一个点表示这个环，然后把环里所有的点连向新节点

维护边双

我们在加入新边的时候，如果两个端点已经在一个连通块内，就会形成一个新的环，环上的点就会属于一个边双

我们可以新建一个点表示这个环，然后把环里所有的点连向新节点

类似于缩点，因为势能在不断降低，就直接暴力缩就可以了

维护子树

Splay维护的是实链上的信息，我们考虑如何维护子树信息

维护子树

Splay维护的是实链上的信息，我们考虑如何维护子树信息

一个节点的子树内部的点除了实儿子子树之外，就只有虚儿子的子树了

维护子树

Splay维护的是实链上的信息，我们考虑如何维护子树信息

一个节点的子树内部的点除了实儿子子树之外，就只有虚儿子的子树了

我们可以对于每一个点多维护一个虚儿子的信息集合，这样就相当于维护了子树

维护子树

Splay维护的是实链上的信息，我们考虑如何维护子树信息

一个节点的子树内部的点除了实儿子子树之外，就只有虚儿子的子树了

我们可以对于每一个点多维护一个虚儿子的信息集合，这样就相当于维护了子树

如果信息具有可减性，我们可以直接拿一个变量存，否则就需要数据结构来维护了

维护子树

Splay维护的是实链上的信息，我们考虑如何维护子树信息

一个节点的子树内部的点除了实儿子子树之外，就只有虚儿子的子树了

我们可以对于每一个点多维护一个虚儿子的信息集合，这样就相当于维护了子树

如果信息具有可减性，我们可以直接拿一个变量存，否则就需要数据结构来维护了

相比于普通的LCT，access、link、pushup的时候会有一点点变化，我们来看一下代码

无标记LCT

首先，我们为什么要打标记?

无标记LCT

首先，我们为什么要打标记?

因为如果我们要对一条链进行操作，我们就要对其中的一个端点MakeRoot，那么我们就要打翻转标记了

无标记LCT

首先，我们为什么要打标记?

因为如果我们要对一条链进行操作，我们就要对其中的一个端点MakeRoot，那么我们就要打翻转标记了

所以对于有根树，如果所有被操作的链都是一条由一个点到其祖先的链，那就可以不用打标记了

无标记LCT

首先，我们为什么要打标记?

因为如果我们要对一条链进行操作，我们就要对其中的一个端点MakeRoot，那么我们就要打翻转标记了

所以对于有根树，如果所有被操作的链都是一条由一个点到其祖先的链，那就可以不用打标记了

这时我们的LCT就只需要splay和access两个主要的函数了，这样常数就特别小，跑得飞快

无标记LCT

首先，我们为什么要打标记?

因为如果我们要对一条链进行操作，我们就要对其中的一个端点MakeRoot，那么我们就要打翻转标记了

所以对于有根树，如果所有被操作的链都是一条由一个点到其祖先的链，那就可以不用打标记了

这时我们的LCT就只需要splay和access两个主要的函数了，这样常数就特别小，跑得飞快

这个大家应该都知道 ~~上个月的模拟赛刚考过...~~

树套树

这个东西一般是用来解决d维偏序问题的，单点修改/区间查询复杂度为$\mathcal{O}(\log^d n)$

树套树

这个东西一般是用来解决d维偏序问题的，单点修改/区间查询复杂度为$\mathcal{O}(\log^d n)$

有的题可以用CDQ分治来优化 ~~但CDQ不归我讲~~

树套树

这个东西一般是用来解决d维偏序问题的，单点修改/区间查询复杂度为$\mathcal{O}(\log^d n)$

有的题可以用CDQ分治来优化 ~~但CDQ不归我讲~~

我们来说一说普通数据结构和树套树的区别

树套树

这个东西一般是用来解决d维偏序问题的，单点修改/区间查询复杂度为$\mathcal{O}(\log^d n)$

有的题可以用CDQ分治来优化 ~~但CDQ不归我讲~~

我们来说一说普通数据结构和树套树的区别

就是一般普通数据结构的底层就是一个变量。我们把这个变量换成一个数据结构，然后这就是一棵树套树了

树套树

这个东西一般是用来解决d维偏序问题的，单点修改/区间查询复杂度为$\mathcal{O}(\log^d n)$

有的题可以用CDQ分治来优化 ~~但CDQ不归我讲~~

我们来说一说普通数据结构和树套树的区别

就是一般普通数据结构的底层就是一个变量。我们把这个变量换成一个数据结构，然后这就是一棵树套树了

下面我们直接通过一道题来引入树套树

二逼平衡树

题目蓝链

您需要写一种数据结构，来维护一个有序数列，数列长度为n，操作次数为m，其中需要提供以下操作：

- 查询k在区间内的排名
- 查询区间内排名为k的值
- 修改某一位值上的数值
- 查询k在区间内的前驱
- 查询k在区间内的后继

$$n, m \leq 5 \times 10^4$$

二逼平衡树

首先我们可以想到一个非常暴力的做法，就是在普通平衡树上套一个线段树

二逼平衡树

首先我们可以想到一个非常暴力的做法，就是在普通平衡树上套一个线段树

这种做法除了二操作的复杂度是$\mathcal{O}(\log^3 n)$，其余均为$\mathcal{O}(\log^2 n)$，而且平衡树的复杂度还跑不满

二逼平衡树

首先我们可以想到一个非常暴力的做法，就是在普通平衡树上套一个线段树

这种做法除了二操作的复杂度是$\mathcal{O}(\log^3 n)$，其余均为$\mathcal{O}(\log^2 n)$，而且平衡树的复杂度还跑不满

正是因为这样，所以这种做法用pbds代替平衡树之后，再开个O2，跑得比我后面讲的那个做法还快 ~~早知道这个跑这么快就打这个了~~

二逼平衡树

我们其实可以把内层改为权值线段树，然后外层再套一个树状数组/线段树

二逼平衡树

我们其实可以把内层改为权值线段树，然后外层再套一个树状数组/线段树

这样我们二操作就可以在权值线段树上二分，省去二分答案的那个log，实现了所有操作复杂度均为$\mathcal{O}(\log^2 n)$

二逼平衡树

我们其实可以把内层改为权值线段树，然后外层再套一个树状数组/线段树

这样我们二操作就可以在权值线段树上二分，省去二分答案的那个log，实现了所有操作复杂度均为$\mathcal{O}(\log^2 n)$

如果你不离散化也可以，只是常数会增大

二逼平衡树

我们其实可以把内层改为权值线段树，然后外层再套一个树状数组/线段树

这样我们二操作就可以在权值线段树上二分，省去二分答案的那个log，实现了所有操作复杂度均为$\mathcal{O}(\log^2 n)$

如果你不离散化也可以，只是常数会增大

我们先来看一下代码

二逼平衡树

我们其实可以把内层改为权值线段树，然后外层再套一个树状数组/线段树

这样我们二操作就可以在权值线段树上二分，省去二分答案的那个log，实现了所有操作复杂度均为$\mathcal{O}(\log^2 n)$

如果你不离散化也可以，只是常数会增大

我们先来看一下代码

那还有没有更好的做法呢?

二逼平衡树

我们可以交换一下位置和权值维护的顺序，即外层维护一个权值线段树，内层维护一个平衡树维护位置

二逼平衡树

我们可以交换一下位置和权值维护的顺序，即外层维护一个权值线段树，内层维护一个平衡树维护位置

这样查询的时候我们就可以直接在外层的权值线段树上二分就可以了，而且时间复杂度还跑不满

二逼平衡树

我们可以交换一下位置和权值维护的顺序，即外层维护一个权值线段树，内层维护一个平衡树维护位置

这样查询的时候我们就可以直接在外层的权值线段树上二分就可以了，而且时间复杂度还跑不满

4,5操作就直接利用1,2操作搞就可以了

二逼平衡树

我们可以交换一下位置和权值维护的顺序，即外层维护一个权值线段树，内层维护一个平衡树维护位置

这样查询的时候我们就可以直接在外层的权值线段树上二分就可以了，而且时间复杂度还跑不满

$4,5$操作就直接利用$1,2$操作搞就可以了

这样的时间复杂度为$\mathcal{O}(m\log^2 n)$，空间复杂度为$\mathcal{O}(m\log n)$

KD-Tree

如果我们要维护的点维数很高，但是关键点有限，我们就可以使用这个数据结构

KD-Tree

如果我们要维护的点维数很高，但是关键点有限，我们就可以使用这个数据结构

这个东西单次操作复杂度大概是$\mathcal{O}(kn^{\frac{k-1}{k}})$ ~~并不会证~~

KD-Tree

如果我们要维护的点维数很高，但是关键点有限，我们就可以使用这个数据结构

这个东西单次操作复杂度大概是$\mathcal{O}(kn^{\frac{k-1}{k}})$ ~~并不会证~~

我写的KD-Tree是只有叶子节点存放关键点，当然也有不同的写法

KD-Tree

如果我们要维护的点维数很高，但是关键点有限，我们就可以使用这个数据结构

这个东西单次操作复杂度大概是$\mathcal{O}(kn^{\frac{k-1}{k}})$ ~~并不会证~~

我写的KD-Tree是只有叶子节点存放关键点，当然也有不同的写法

不说多了，先来看一下怎么构建一棵KD-Tree

build

- 我们找到当前KDT的节点管辖的关键点中维度级差最大的维度

build

- 我们找到当前KDT的节点管辖的关键点中维度级差最大的维度
- 我们就对于这一维从中间把这些关键点分成两半

build

- 我们找到当前KDT的节点管辖的关键点中维度级差最大的维度
- 我们就对于这一维从中间把这些关键点分成两半
- 然后直接递归建树就可以了

build

- 我们找到当前KDT的节点管辖的关键点中维度级差最大的维度
- 我们就对于这一维从中间把这些关键点分成两半
- 然后直接递归建树就可以了

具体实现可以看代码

build

- 我们找到当前KDT的节点管辖的关键点中维度级差最大的维度
- 我们就对于这一维从中间把这些关键点分成两半
- 然后直接递归建树就可以了

具体实现可以看代码

如果不按分割级差最大的维度，就有可能会被特殊数据卡掉

最邻近查询

首先从根结点开始查询，每次判断查询点是在分割面的哪一边，然后往下走

最邻近查询

首先从根结点开始查询，每次判断查询点是在分割面的哪一边，然后往下走

一直走到叶子节点，用叶子到查询点的距离更新答案

最邻近查询

首先从根结点开始查询，每次判断查询点是在分割面的哪一边，然后往下走

一直走到叶子节点，用叶子到查询点的距离更新答案

在回溯的时候判断一下，如果查询点到分割面的距离小于当前的答案，就递归另外一个儿子

最邻近查询

首先从根结点开始查询，每次判断查询点是在分割面的哪一边，然后往下走

一直走到叶子节点，用叶子到查询点的距离更新答案

在回溯的时候判断一下，如果查询点到分割面的距离小于当前的答案，就递归另外一个儿子

这样就一定能找到最优解了

最邻近查询

首先从根结点开始查询，每次判断查询点是在分割面的哪一边，然后往下走

一直走到叶子节点，用叶子到查询点的距离更新答案

在回溯的时候判断一下，如果查询点到分割面的距离小于当前的答案，就递归另外一个儿子

这样就一定能找到最优解了

但是这样的复杂度会比较高，那我们要怎么在有限的时间里找到优秀的近似解呢?

近似搜索算法

我们先确定最大回溯的层数L

近似搜索算法

我们先确定最大回溯的层数L

一开始像正常算法一样，从根结点往叶子节点走，并在叶子节点更新答案

近似搜索算法

我们先确定最大回溯的层数L

一开始像正常算法一样，从根结点往叶子节点走，并在叶子节点更新答案

然后在回溯的时候，判断一下所有经过节点的另一个儿子是否可能有更优解，有就把查询点到分割面的距离加入到优先队列里去

近似搜索算法

我们先确定最大回溯的层数L

一开始像正常算法一样，从根结点往叶子节点走，并在叶子节点更新答案

然后在回溯的时候，判断一下所有经过节点的另一个儿子是否可能有更优解，有就把查询点到分割面的距离加入到优先队列里去

回到根之后，如果还没到最大回溯层数并且优先队列非空，我们就取出队首并以其为根重复这个过程

近似搜索算法

我们先确定最大回溯的层数L

一开始像正常算法一样，从根结点往叶子节点走，并在叶子节点更新答案

然后在回溯的时候，判断一下所有经过节点的另一个儿子是否可能有更优解，有就把查询点到分割面的距离加入到优先队列里去

回到根之后，如果还没到最大回溯层数并且优先队列非空，我们就取出队首并以其为根重复这个过程

这样的复杂度为$\mathcal{O}(Bk\log n)$的 ~~复杂度依然不会证~~

近似搜索算法

我们先确定最大回溯的层数L

一开始像正常算法一样，从根结点往叶子节点走，并在叶子节点更新答案

然后在回溯的时候，判断一下所有经过节点的另一个儿子是否可能有更优解，有就把查询点到分割面的距离加入到优先队列里去

回到根之后，如果还没到最大回溯层数并且优先队列非空，我们就取出队首并以其为根重复这个过程

这样的复杂度为$\mathcal{O}(Bk\log n)$的 ~~复杂度依然不会证~~

我在这里挂一个连接，里面有较详细的图片解释 传送门

分块

有时我们维护一个区间的信息没有办法快速合并，所以就不能使用线段树来解决问题

分块

有时我们维护一个区间的信息没有办法快速合并，所以就不能使用线段树来解决问题

但是我们有时可以把需要维护的元素分成若干个部分，对于部分整体和内部分情况来维护

分块

有时我们维护一个区间的信息没有办法快速合并，所以就不能使用线段树来解决问题

但是我们有时可以把需要维护的元素分成若干个部分，对于部分整体和内部分情况来维护

甚至你可以用这种办法在各种操作中找到一个复杂度的均衡点 ~~全程瞎扯~~

分块

有时我们维护一个区间的信息没有办法快速合并，所以就不能使用线段树来解决问题

但是我们有时可以把需要维护的元素分成若干个部分，对于部分整体和内部分情况来维护

甚至你可以用这种办法在各种操作中找到一个复杂度的均衡点 ~~全程瞎扯~~

我们称这种算法叫做分块

分块

有时我们维护一个区间的信息没有办法快速合并，所以就不能使用线段树来解决问题

但是我们有时可以把需要维护的元素分成若干个部分，对于部分整体和内部分情况来维护

甚至你可以用这种办法在各种操作中找到一个复杂度的均衡点 ~~全程瞎扯~~

我们称这种算法叫做分块

~~感觉没啥好写的了~~

分块

这种东西其实没有太多的东西可以讲

分块

这种东西其实没有太多的东西可以讲

主要是平时要多做这方面的题，慢慢地你就会有感觉

分块

这种东西其实没有太多的东西可以讲

主要是平时要多做这方面的题，慢慢地你就会有感觉

下面我们就通过一道例题来感受一下这个算法

舌尖上的由乃

题目蓝链

给定一个长度为n的序列，你需要支持两种操作

- 区间加
- 区间查询第k大

$n \leq 10^5$, T = 6s

舌尖上的由乃

我们可以考虑对于每一个块维护一个有序数列，我们设块大小为B

舌尖上的由乃

我们可以考虑对于每一个块维护一个有序数列，我们设块大小为B

修改整块就直接打标记，小块就直接暴力加然后和其他位置的数归并和起来就可以了，复杂度$\mathcal{O}(n/B+B)$

舌尖上的由乃

我们可以考虑对于每一个块维护一个有序数列，我们设块大小为B

修改整块就直接打标记，小块就直接暴力加然后和其他位置的数归并和起来就可以了，复杂度$\mathcal{O}(n/B+B)$

查询就直接二分答案，然后对于每一个块二分查询一下，就可以查到区间第k大了，复杂度$\mathcal{O}(n/B\log^2 n+B)$

舌尖上的由乃

我们可以考虑对于每一个块维护一个有序数列，我们设块大小为B

修改整块就直接打标记，小块就直接暴力加然后和其他位置的数归并和起来就可以了，复杂度$\mathcal{O}(n/B+B)$

查询就直接二分答案，然后对于每一个块二分查询一下，就可以查到区间第k大了，复杂度$\mathcal{O}(n/B\log^2 n+B)$

我们取块大小$B=\sqrt{n}\times\log n$时，复杂度为$\mathcal{O}(n\sqrt{n}\log n)$

树分块

这里要讲的是链分块型，保证每一条链都可以被分成$\mathcal{O}(\sqrt{n})$级别的整块和散块

树分块

这里要讲的是链分块型，保证每一条链都可以被分成$\mathcal{O}(\sqrt{n})$级别的整块和散块

我们可以给所在层数为$\sqrt{n}$倍数的点都标为关键点，然后再建一棵树基于这些关键点的虚树，每一条树边为一个块，剩下的为散块

树分块

这里要讲的是链分块型，保证每一条链都可以被分成$\mathcal{O}(\sqrt{n})$级别的整块和散块

我们可以给所在层数为$\sqrt{n}$倍数的点都标为关键点，然后再建一棵树基于这些关键点的虚树，每一条树边为一个块，剩下的为散块

我们不难发现这样做，无论是块数，还是块大小都是$\mathcal{O}(\sqrt{n})$

树分块

这里要讲的是链分块型，保证每一条链都可以被分成$\mathcal{O}(\sqrt{n})$级别的整块和散块

我们可以给所在层数为$\sqrt{n}$倍数的点都标为关键点，然后再建一棵树基于这些关键点的虚树，每一条树边为一个块，剩下的为散块

我们不难发现这样做，无论是块数，还是块大小都是$\mathcal{O}(\sqrt{n})$

然后我们就可以利用这个东西，快速的进行一些链上操作了

树分块

这里要讲的是链分块型，保证每一条链都可以被分成$\mathcal{O}(\sqrt{n})$级别的整块和散块

我们可以给所在层数为$\sqrt{n}$倍数的点都标为关键点，然后再建一棵树基于这些关键点的虚树，每一条树边为一个块，剩下的为散块

我们不难发现这样做，无论是块数，还是块大小都是$\mathcal{O}(\sqrt{n})$

然后我们就可以利用这个东西，快速的进行一些链上操作了

我们在实际实现的时候，也可以直接随机$\sqrt{n}$个点构建虚树，这样复杂度期望也是根号的

莫队

相信大家都学过，我们就来分析一下这个东西的复杂度

莫队

相信大家都学过，我们就来分析一下这个东西的复杂度

我们设序列的长度为n，询问的次数为m，块大小为B

莫队

相信大家都学过，我们就来分析一下这个东西的复杂度

我们设序列的长度为n，询问的次数为m，块大小为B

对于同一个块内的询问左端点，右端点单调递增，所以最多移动$\mathcal{O}(n)$次，所以总复杂度为$\mathcal{O}(\frac{n^2}{B})$

莫队

相信大家都学过，我们就来分析一下这个东西的复杂度

我们设序列的长度为n，询问的次数为m，块大小为B

对于同一个块内的询问左端点，右端点单调递增，所以最多移动$\mathcal{O}(n)$次，所以总复杂度为$\mathcal{O}(\frac{n^2}{B})$

对于排序后的询问左端点，相邻询问之间左端点距离的$\mathcal{O}(B)$级别的，所以总复杂度为$\mathcal{O}(mB)$

莫队

相信大家都学过，我们就来分析一下这个东西的复杂度

我们设序列的长度为n，询问的次数为m，块大小为B

对于同一个块内的询问左端点，右端点单调递增，所以最多移动$\mathcal{O}(n)$次，所以总复杂度为$\mathcal{O}(\frac{n^2}{B})$

对于排序后的询问左端点，相邻询问之间左端点距离的$\mathcal{O}(B)$级别的，所以总复杂度为$\mathcal{O}(mB)$

我们取块大小为$B = \frac{n}{\sqrt{m}}$，此时的复杂度为$\mathcal{O}(n\sqrt{m})$

带修莫队

我们考虑把之前的询问(l_i, r_i)加上时间的维度，变为(l_i, r_i, t_i)

带修莫队

我们考虑把之前的询问(l_i, r_i)加上时间的维度，变为(l_i, r_i, t_i)

我们移动时间就相当与是在不断地操作和撤销操作，我们只需要考虑操作对于当前莫队维护的区间的贡献就可以了

带修莫队

我们考虑把之前的询问(l_i, r_i)加上时间的维度，变为(l_i, r_i, t_i)

我们移动时间就相当与是在不断地操作和撤销操作，我们只需要考虑操作对于当前莫队维护的区间的贡献就可以了

我们考虑处理左右端点所在的块都相同的询问时，我们可以按照时间的顺序来处理这些询问

带修莫队

我们考虑把之前的询问(l_i, r_i)加上时间的维度，变为(l_i, r_i, t_i)

我们移动时间就相当与是在不断地操作和撤销操作，我们只需要考虑操作对于当前莫队维护的区间的贡献就可以了

我们考虑处理左右端点所在的块都相同的询问时，我们可以按照时间的顺序来处理这些询问

我们同样设序列的长度为n，询问的次数为m，块大小为B

带修莫队

我们考虑把之前的询问(l_i, r_i)加上时间的维度，变为(l_i, r_i, t_i)

我们移动时间就相当与是在不断地操作和撤销操作，我们只需要考虑操作对于当前莫队维护的区间的贡献就可以了

我们考虑处理左右端点所在的块都相同的询问时，我们可以按照时间的顺序来处理这些询问

我们同样设序列的长度为n，询问的次数为m，块大小为B

那么左端点和右端点移动的总距离为mB，时间维度移动的总距离为$(\frac{n}{B})^2 m$

带修莫队

我们考虑把之前的询问(l_i, r_i)加上时间的维度，变为(l_i, r_i, t_i)

我们移动时间就相当与是在不断地操作和撤销操作，我们只需要考虑操作对于当前莫队维护的区间的贡献就可以了

我们考虑处理左右端点所在的块都相同的询问时，我们可以按照时间的顺序来处理这些询问

我们同样设序列的长度为n，询问的次数为m，块大小为B

那么左端点和右端点移动的总距离为mB，时间维度移动的总距离为$(\frac{n}{B})^2 m$

我们取$B = n^{\frac{2}{3}}$时，时间复杂度为$\mathcal{O}(mn^{\frac{2}{3}})$

回滚莫队

这个东西实际上和普通莫队差不多

回滚莫队

这个东西实际上和普通莫队差不多

如果莫队维护的东西，加入方便维护删除不好维护

回滚莫队

这个东西实际上和普通莫队差不多

如果莫队维护的东西，加入方便维护删除不好维护

那我们可以每次询问之后把左端点定位在其所在块的右端点处，这样每次移动端点的时候就只有加入元素了

回滚莫队

这个东西实际上和普通莫队差不多

如果莫队维护的东西，加入方便维护删除不好维护

那我们可以每次询问之后把左端点定位在其所在块的右端点处，这样每次移动端点的时候就只有加入元素了

我们在左端点往左移动之前要记录一下操作前的状态，方便移动完之后恢复

RMQ与分块结合

如果有的题目的询问序列较长，而询问次数又没有那么多，我们就可以用分块来平衡复杂度

RMQ与分块结合

如果有的题目的询问序列较长，而询问次数又没有那么多，我们就可以用分块来平衡复杂度

我们可以对询问的序列进行分块，块大小为$B = \frac{\log n}{2}$，则块数为$k = \frac{2n}{\log n}$ (可以手动调最优值)

RMQ与分块结合

如果有的题目的询问序列较长，而询问次数又没有那么多，我们就可以用分块来平衡复杂度

我们可以对询问的序列进行分块，块大小为$B = \frac{\log n}{2}$，则块数为$k = \frac{2n}{\log n}$ (可以手动调最优值)

我们可以先把每一个块作为一个数字，来预处理出整块之间的答案，时间复杂度为$\mathcal{O}(k \log k)$

RMQ与分块结合

如果有的题目的询问序列较长，而询问次数又没有那么多，我们就可以用分块来平衡复杂度

我们可以对询问的序列进行分块，块大小为$B = \frac{\log n}{2}$，则块数为$k = \frac{2n}{\log n}$ (可以手动调最优值)

我们可以先把每一个块作为一个数字，来预处理出整块之间的答案，时间复杂度为$\mathcal{O}(k \log k)$

块内就直接暴力求出前后缀的答案，时间复杂度为$\mathcal{O}(n)$

RMQ与分块结合

如果有的题目的询问序列较长，而询问次数又没有那么多，我们就可以用分块来平衡复杂度

我们可以对询问的序列进行分块，块大小为$B = \frac{\log n}{2}$，则块数为$k = \frac{2n}{\log n}$ (可以手动调最优值)

我们可以先把每一个块作为一个数字，来预处理出整块之间的答案，时间复杂度为$\mathcal{O}(k \log k)$

块内就直接暴力求出前后缀的答案，时间复杂度为$\mathcal{O}(n)$

然后对于一次询问操作，我们可以直接询问整块的答案，然后散块的答案也可以直接得到，时间复杂度为$\mathcal{O}(1)$

RMQ与分块结合

但是这个东西在查询端点在一个块内的时候，复杂度是$\mathcal{O}(\log n)$的

RMQ与分块结合

但是这个东西在查询端点在一个块内的时候，复杂度是$\mathcal{O}(\log n)$的

但是只要数据不特意去卡，一般期望都是$\mathcal{O}(1)$的，因为大部分操作都不会是在1个块内

RMQ与分块结合

但是这个东西在查询端点在一个块内的时候，复杂度是$\mathcal{O}(\log n)$的

但是只要数据不特意去卡，一般期望都是$\mathcal{O}(1)$的，因为大部分操作都不会是在1个块内

为了防止被卡，你也可以在每一个小块内再套一个线段树/分块来维护，这样查询块内的复杂度就差不多是一个常数了

RMQ与分块结合

但是这个东西在查询端点在一个块内的时候，复杂度是$\mathcal{O}(\log n)$的

但是只要数据不特意去卡，一般期望都是$\mathcal{O}(1)$的，因为大部分操作都不会是在1个块内

为了防止被卡，你也可以在每一个小块内再套一个线段树/分块来维护，这样查询块内的复杂度就差不多是一个常数了

这样我们就可以做到$\mathcal{O}(n)$预处理，$\mathcal{O}(常数)$回答了

±1 RMQ

这个东西可以维护相邻元素差值为1的序列，可以做到O预处理，$O(1)$回答了

± 1 RMQ

这个东西可以维护相邻元素差值为1的序列，可以做到$\mathcal{O}$预处理，$\mathcal{O}(1)$回答了

我暂时只会用这个东西维护区间最值，不知道能不能用来维护区间gcd

± 1 RMQ

这个东西可以维护相邻元素差值为1的序列，可以做到$\mathcal{O}$预处理，$\mathcal{O}(1)$回答了

我暂时只会用这个东西维护区间最值，不知道能不能用来维护区间gcd

我们还是和前面一样的对序列分块，块大小和前面一样是B，区别只在于对块内的询问

±1 RMQ

这个东西可以维护相邻元素差值为1的序列，可以做到$\mathcal{O}$预处理，$\mathcal{O}(1)$回答了

我暂时只会用这个东西维护区间最值，不知道能不能用来维护区间gcd

我们还是和前面一样的对序列分块，块大小和前面一样是B，区别只在于对块内的询问

我们发现在前提条件之下，本质不同的块只有2^B个，而每一个块最多只有B^2种询问，所以块内的预处理是$2^B B^2$的

±1 RMQ

这个东西可以维护相邻元素差值为1的序列，可以做到$\mathcal{O}$预处理，$\mathcal{O}(1)$回答了

我暂时只会用这个东西维护区间最值，不知道能不能用来维护区间gcd

我们还是和前面一样的对序列分块，块大小和前面一样是B，区别只在于对块内的询问

我们发现在前提条件之下，本质不同的块只有2^B个，而每一个块最多只有B^2种询问，所以块内的预处理是$2^B B^2$的

这样我们对于块内询问就可以直接回答了

bitset

这个东西大家应该都知道吧

bitset

这个东西大家应该都知道吧

它可以支持快速的与、并、异或操作，可以对时间复杂度除以32

bitset

这个东西大家应该都知道吧

它可以支持快速的与、并、异或操作，可以对时间复杂度除以32

它还可以对空间复杂度除以8

bitset

这个东西大家应该都知道吧

它可以支持快速的与、并、异或操作，可以对时间复杂度除以32

它还可以对空间复杂度除以8

- bitset<1000> bit;

bitset

这个东西大家应该都知道吧

它可以支持快速的与、并、异或操作，可以对时间复杂度除以32

它还可以对空间复杂度除以8

- bitset<1000> bit;

上面这个例子就表示开了一个大小为1000的bitset

bitset

下面是一些bitset的基本操作

bitset

下面是一些bitset的基本操作

- bit.flip()，表示翻转bitset的每一位
- bit.any()，检查bitset中是否有1
- bit.none()，检查bitset中是否全为0
- bit.count()，返回bitset中是1的个数
- bit.reset()，将bitset清0
- bit._Find_first()，查询bitset最低位1的位置
- bit._Find_next(i)，查询从$i+1$开始的第1个数位为1的位置

bitset

下面是一些bitset的基本操作

- bit.flip()，表示翻转bitset的每一位
- bit.any()，检查bitset中是否有1
- bit.none()，检查bitset中是否全为0
- bit.count()，返回bitset中是1的个数
- bit.reset()，将bitset清0
- bit._Find_first()，查询bitset最低位1的位置
- bit._Find_next(i)，查询从$i+1$开始的第1个数位为1的位置

其余的和整数运算差不多

动态DP

动态DP，又称DDP

动态DP

动态DP，又称DDP

这个东西就是对于一个正常的DP加上需要支持动态修改和询问

动态DP

动态DP，又称DDP

这个东西就是对于一个正常的DP加上需要支持动态修改和询问

然后我们可以利用构造出来的矩阵来转移DP，再按需要套一个数据结构(线段树/树链剖分)来维护就可以了

动态DP

动态DP，又称DDP

这个东西就是对于一个正常的DP加上需要支持动态修改和询问

然后我们可以利用构造出来的矩阵来转移DP，再按需要套一个数据结构(线段树/树链剖分)来维护就可以了

如果要修改，就直接修改转移矩阵即可

动态DP

动态DP，又称DDP

这个东西就是对于一个正常的DP加上需要支持动态修改和询问

然后我们可以利用构造出来的矩阵来转移DP，再按需要套一个数据结构(线段树/树链剖分)来维护就可以了

如果要修改，就直接修改转移矩阵即可

这个东西用处还是很多的，可以用来优化一些暴力，比如$NOIP2018\ D2T3$

树

给你一棵n个点的树，每条边都有一个权值

你需要支持m次操作，每次操作为修改一条边的权值，然后你需要求出这棵树的直径

$n, m \leq 3 \times 10^5$

树

这是一道没有来源的题，我在某篇博客上看到的 ~~以下内容全程口胡，有错误请指出~~

树

这是一道没有来源的题，我在某篇博客上看到的 ~~以下内容全程口胡，有错误请指出~~

我们设x, y是一对具有父子关系的点，f_x为当前点往下最长路径长度，g_x为当前子树内的直径

树

这是一道没有来源的题，我在某篇博客上看到的 ~~以下内容全程口胡，有错误请指出~~

我们设x, y是一对具有父子关系的点，f_x为当前点往下最长路径长度，g_x为当前子树内的直径

如果我们对于一个节点依次合并子树，那么我们不难发现有转移方程

树

这是一道没有来源的题，我在某篇博客上看到的 ~~以下内容全程口胡，有错误请指出~~

我们设x, y是一对具有父子关系的点，f_x为当前点往下最长路径长度，g_x为当前子树内的直径

如果我们对于一个节点依次合并子树，那么我们不难发现有转移方程

$$f_x = \max\{f_y + w_{x,y}\}, g_x = \max\{g_y, f_x + f_y + w_{x,y}\}$$

树

因为这道题要动态修改，所以我们要用树剖来维护

树

因为这道题要动态修改，所以我们要用树剖来维护

我们设f'_x, g'_x表示x节点所有不包含的重儿子的最优值，w为x, y之间的边权

树

因为这道题要动态修改，所以我们要用树剖来维护

我们设f'_x, g'_x表示x节点所有不包含的重儿子的最优值，w为x, y之间的边权

然后我们可以根据这个dp转移，构造出转移的矩阵

树

因为这道题要动态修改，所以我们要用树剖来维护

我们设f'_x, g'_x表示x节点所有不包含的重儿子的最优值，w为x, y之间的边权

然后我们可以根据这个dp转移，构造出转移的矩阵

$$[f_y \quad g_y \quad 0] \times \begin{bmatrix} w & f'_x + w & -\infty \\ -\infty & 0 & -\infty \\ f'_x & g'_x & 0 \end{bmatrix} = [f_x \quad g_x \quad 0]$$

树

因为这道题要动态修改，所以我们要用树剖来维护

我们设f'_x, g'_x表示x节点所有不包含的重儿子的最优值，w为x, y之间的边权

然后我们可以根据这个dp转移，构造出转移的矩阵

$$[f_y \quad g_y \quad 0] \times \begin{bmatrix} w & f'_x + w & -\infty \\ -\infty & 0 & -\infty \\ f'_x & g'_x & 0 \end{bmatrix} = [f_x \quad g_x \quad 0]$$

这样，我们对于一次操作，就可以直接修改对应点的转移矩阵。然后重链直接转移，轻边就暴力转移

树

因为这道题要动态修改，所以我们要用树剖来维护

我们设f'_x, g'_x表示x节点所有不包含的重儿子的最优值，w为x, y之间的边权

然后我们可以根据这个dp转移，构造出转移的矩阵

$$[f_y \quad g_y \quad 0] \times \begin{bmatrix} w & f'_x + w & -\infty \\ -\infty & 0 & -\infty \\ f'_x & g'_x & 0 \end{bmatrix} = [f_x \quad g_x \quad 0]$$

这样，我们对于一次操作，就可以直接修改对应点的转移矩阵。然后重链直接转移，轻边就暴力转移

我们可以通过在每条重链上单独维护平衡树来实现这个过程

树

因为这道题要动态修改，所以我们要用树剖来维护

我们设f'_x, g'_x表示x节点所有不包含的重儿子的最优值，w为x, y之间的边权

然后我们可以根据这个dp转移，构造出转移的矩阵

$$[f_y \quad g_y \quad 0] \times \begin{bmatrix} w & f'_x + w & -\infty \\ -\infty & 0 & -\infty \\ f'_x & g'_x & 0 \end{bmatrix} = [f_x \quad g_x \quad 0]$$

这样，我们对于一次操作，就可以直接修改对应点的转移矩阵。然后重链直接转移，轻边就暴力转移

我们可以通过在每条重链上单独维护平衡树来实现这个过程

时间复杂度$\mathcal{O}(m \log n)$

划分树

个人感觉这个算法用到的地方特别少，我好像是没遇到过

划分树

个人感觉这个算法用到的地方特别少，我好像是没遇到过

这东西我大概看了一下，感觉就是权值线段树套平衡树？大雾

划分树

个人感觉这个算法用到的地方特别少，我好像是没遇到过

这东西我大概看了一下，感觉就是权值线段树套平衡树？大雾

我会发一个之前讲划分树的一个课件，有兴趣的同学可以去了解一下

PBDS

pbds又称平板电视，里面包含了hash_table、tree、trie、priority_queue四种数据结构

PBDS

pbds又称平板电视，里面包含了hash_table、tree、trie、priority_queue四种数据结构

这些东西可以极大的减少代码复杂度 ~~解放双手~~

PBDS

pbds又称平板电视，里面包含了hash_table、tree、trie、priority_queue四种数据结构

这些东西可以极大的减少代码复杂度 ~~解放双手~~

这其中我们用得较多的就是hash_table和tree

PBDS

pbds又称平板电视，里面包含了hash_table、tree、trie、priority_queue四种数据结构

这些东西可以极大的减少代码复杂度 ~~解放双手~~

这其中我们用得较多的就是hash_table和tree

前者的用法和map差不多，但是期望复杂度为$\mathcal{O}(1)$，我们也可以用tr1::unordered_map

PBDS

pbds又称平板电视，里面包含了hash_table、tree、trie、priority_queue四种数据结构

这些东西可以极大的减少代码复杂度 ~~解放双手~~

这其中我们用得较多的就是hash_table和tree

前者的用法和map差不多，但是期望复杂度为$\mathcal{O}(1)$，我们也可以用tr1::unordered_map

后者则是系统帮你写好的一棵平衡树，可以实现插入、删除、查询第k大和查询排名等一系列操作，当然你也可以通过自定义函数来实现你想要的功能

PBDS

pbds又称平板电视，里面包含了hash_table、tree、trie、priority_queue四种数据结构

这些东西可以极大的减少代码复杂度 ~~解放双手~~

这其中我们用得较多的就是hash_table和tree

前者的用法和map差不多，但是期望复杂度为$\mathcal{O}(1)$，我们也可以用tr1::unordered_map

后者则是系统帮你写好的一棵平衡树，可以实现插入、删除、查询第k大和查询排名等一系列操作，当然你也可以通过自定义函数来实现你想要的功能

这些东西可以极大的减少敲代码时间和调试的时间，在关键的时候有奇效

PBDS

pbds又称平板电视，里面包含了hash_table、tree、trie、priority_queue四种数据结构

这些东西可以极大的减少代码复杂度 ~~解放双手~~

这其中我们用得较多的就是hash_table和tree

前者的用法和map差不多，但是期望复杂度为$\mathcal{O}(1)$，我们也可以用tr1::unordered_map

后者则是系统帮你写好的一棵平衡树，可以实现插入、删除、查询第k大和查询排名等一系列操作，当然你也可以通过自定义函数来实现你想要的功能

这些东西可以极大的减少敲代码时间和调试的时间，在关键的时候有奇效

这里放一个学习用法的链接 传送门

Box

题目蓝链

有N个盒子，可以互相嵌套，首先告诉你每个盒子放在哪个盒子里(相当于一个森林)，然后需要支持m次以下两种操作：

- $MOVE\ x\ y$，将编号x的盒子及其里面的所有盒子一起移至编号y的盒子里
- $QUERY\ x$，询问编号x的盒子最外层盒子的编号

$n, m \leq 10^5$

Box

方法很简单，首先将题目所给的每一个盒子的信息来初始化平衡树，要将一个盒子拆成左右两个端点，端点之间则为盒子内部

Box

方法很简单，首先将题目所给的每一个盒子的信息来初始化平衡树，要将一个盒子拆成左右两个端点，端点之间则为盒子内部

对于每次移动，我们就先把需移动的盒子从平衡树中拿出来，然后在指定的位置接入即可

Box

方法很简单，首先将题目所给的每一个盒子的信息来初始化平衡树，要将一个盒子拆成左右两个端点，端点之间则为盒子内部

对于每次移动，我们就先把需移动的盒子从平衡树中拿出来，然后在指定的位置接入即可

对于每次查询，被查询的盒子所在的平衡树中最左的端点即为最外层盒子

Gty的二逼妹子序列

题目蓝链

给定一个长度为n的数列a_i

有m次询问，每次给定$l\ r\ a\ b$，询问区间$[l,r]$中，权值在$[a,b]$的权值种类数

$n,m \leq 10^5, 1 \leq a_i \leq n$

Gty的二逼妹子序列

我们可以考虑使用莫队，我们考虑莫队的每种操作的次数

Gty的二逼妹子序列

我们可以考虑使用莫队，我们考虑莫队的每种操作的次数

- 插入一个数$\mathcal{O}(n\sqrt{m})$次

Gty的二逼妹子序列

我们可以考虑使用莫队，我们考虑莫队的每种操作的次数

- 插入一个数$\mathcal{O}(n\sqrt{m})$次
- 删除一个数$\mathcal{O}(n\sqrt{m})$次

Gty的二逼妹子序列

我们可以考虑使用莫队，我们考虑莫队的每种操作的次数

- 插入一个数$\mathcal{O}(n\sqrt{m})$次
- 删除一个数$\mathcal{O}(n\sqrt{m})$次
- 询问区间权值种类$\mathcal{O}(m)$次

Gty的二逼妹子序列

我们可以考虑使用莫队，我们考虑莫队的每种操作的次数

- 插入一个数$\mathcal{O}(n\sqrt{m})$次
- 删除一个数$\mathcal{O}(n\sqrt{m})$次
- 询问区间权值种类$\mathcal{O}(m)$次

如果我们用线段树来统计，复杂度为$\mathcal{O}(n\sqrt{m}\log n)$

Gty的二逼妹子序列

我们可以考虑使用莫队，我们考虑莫队的每种操作的次数

- 插入一个数$\mathcal{O}(n\sqrt{m})$次
- 删除一个数$\mathcal{O}(n\sqrt{m})$次
- 询问区间权值种类$\mathcal{O}(m)$次

如果我们用线段树来统计，复杂度为$\mathcal{O}(n\sqrt{m}\log n)$

我们可以考虑把统计数值的桶分块，就可以做到$\mathcal{O}(1)$修改，根号查询了

Gty的二逼妹子序列

我们可以考虑使用莫队，我们考虑莫队的每种操作的次数

- 插入一个数$\mathcal{O}(n\sqrt{m})$次
- 删除一个数$\mathcal{O}(n\sqrt{m})$次
- 询问区间权值种类$\mathcal{O}(m)$次

如果我们用线段树来统计，复杂度为$\mathcal{O}(n\sqrt{m}\log n)$

我们可以考虑把统计数值的桶分块，就可以做到$\mathcal{O}(1)$修改，根号查询了

总复杂度就变为$\mathcal{O}(n\sqrt{m}+m\sqrt{n})$

未来日记

题目蓝链

一个长度为n的序列,支持m次两种操作

- 区间内值为x的位置全部变成y
- 区间询问第k大

$n, m \leq 10^5$，值域与n同级

未来日记

我们可以对于这个东西按序列分块，再按值域分块

未来日记

我们可以对于这个东西按序列分块，再按值域分块

我们分别记录在前i块、值域在第j块的有几个和在前i块、值为j的数有几个

未来日记

我们可以对于这个东西按序列分块，再按值域分块

我们分别记录在前i块、值域在第j块的有几个和在前i块、值为j的数有几个

我们每次更新时候暴力维护这两个数组，这样我们就可以利用值域分块暴力定位一段整块的第k大在哪个值域块内，然后在扫描一遍这一段整块中的这个值域段就可以知道第k大是哪个值

未来日记

我们可以对于这个东西按序列分块，再按值域分块

我们分别记录在前i块、值域在第j块的有几个和在前i块、值为j的数有几个

我们每次更新时候暴力维护这两个数组，这样我们就可以利用值域分块暴力定位一段整块的第k大在哪个值域块内，然后在扫描一遍这一段整块中的这个值域段就可以知道第k大是哪个值

我们考虑怎么快速维护每一个位置的值

未来日记

我们可以对于这个东西按序列分块，再按值域分块

我们分别记录在前i块、值域在第j块的有几个和在前i块、值为j的数有几个

我们每次更新时候暴力维护这两个数组，这样我们就可以利用值域分块暴力定位一段整块的第k大在哪个值域块内，然后在扫描一遍这一段整块中的这个值域段就可以知道第k大是哪个值

我们考虑怎么快速维护每一个位置的值

我们可以对于每一个块里的所有权值维护一个并查集，每一个并查集里包含当前块中为该权值的下标，每个并查集要记录表示的权值

未来日记

我们可以对于这个东西按序列分块，再按值域分块

我们分别记录在前i块、值域在第j块的有几个和在前i块、值为j的数有几个

我们每次更新时候暴力维护这两个数组，这样我们就可以利用值域分块暴力定位一段整块的第k大在哪个值域块内，然后在扫描一遍这一段整块中的这个值域段就可以知道第k大是哪个值

我们考虑怎么快速维护每一个位置的值

我们可以对于每一个块里的所有权值维护一个并查集，每一个并查集里包含当前块中为该权值的下标，每个并查集要记录表示的权值

我们在合并的时候就直接合并两个并查集，查询单点就直接看属于哪个并查集就可以了，修改散块就直接暴力重构

未来日记

我们可以对于这个东西按序列分块，再按值域分块

我们分别记录在前i块、值域在第j块的有几个和在前i块、值为j的数有几个

我们每次更新时候暴力维护这两个数组，这样我们就可以利用值域分块暴力定位一段整块的第k大在哪个值域块内，然后在扫描一遍这一段整块中的这个值域段就可以知道第k大是哪个值

我们考虑怎么快速维护每一个位置的值

我们可以对于每一个块里的所有权值维护一个并查集，每一个并查集里包含当前块中为该权值的下标，每个并查集要记录表示的权值

我们在合并的时候就直接合并两个并查集，查询单点就直接看属于哪个并查集就可以了，修改散块就直接暴力重构

复杂度$\mathcal{O}(n\sqrt{n})$

掉进兔子洞

题目蓝链

给定一个长度为n的序列a_i。有m个询问，每次询问三个区间，把三个区间中同时出现的数一个一个删掉，问最后三个区间剩下的数的个数和，询问独立

注意这里删掉指的是一个一个删，不是把等于这个值的数直接删完，比如三个区间是$[1,2,2,3,3,3,3]$，$[1,2,2,3,3,3,3]$与$[1,1,2,3,3]$，就一起扔掉了1个1，1个2，2个3

$n,m \leq 10^5, a_i \leq 10^9$

掉进兔子洞

这题就是一道莫队+bitset板子题

掉进兔子洞

这题就是一道莫队+bitset板子题

我们先把输入的数字离散化一下，然后询问的时候要把一个询问拆成3个询问，然后再并起来

掉进兔子洞

这题就是一道莫队+bitset板子题

我们先把输入的数字离散化一下，然后询问的时候要把一个询问拆成3个询问，然后再并起来

然后在莫队的时候记录cnt_i表示当前数字i出现的次数，再开一个$bitset$，假设当前需要加入$bitset$的数字是x，我们就令$bitset$中的第$p_x + cnt_x$位为1，然后$++cnt_x$

掉进兔子洞

这题就是一道莫队+bitset板子题

我们先把输入的数字离散化一下，然后询问的时候要把一个询问拆成3个询问，然后再并起来

然后在莫队的时候记录cnt_i表示当前数字i出现的次数，再开一个$bitset$，假设当前需要加入$bitset$的数字是x，我们就令$bitset$中的第$p_x + cnt_x$位为1，然后$++cnt_x$

这样就会使得把三个$bitset$并起来之后，恰好剩下的就是在三个区间都出现了的数字，也就是需要删除的数字，我们就只需要看并起来之后还剩多少个1就行了

掉进兔子洞

这题就是一道莫队+bitset板子题

我们先把输入的数字离散化一下，然后询问的时候要把一个询问拆成3个询问，然后再并起来

然后在莫队的时候记录cnt_i表示当前数字i出现的次数，再开一个$bitset$，假设当前需要加入$bitset$的数字是x，我们就令$bitset$中的第$p_x + cnt_x$位为1，然后$++cnt_x$

这样就会使得把三个$bitset$并起来之后，恰好剩下的就是在三个区间都出现了的数字，也就是需要删除的数字，我们就只需要看并起来之后还剩多少个1就行了

还有一个问题，就是我们必须要对每一个询问(拆之前)都要开一个$bitset$来存答案，但这会开不下

掉进兔子洞

这题就是一道莫队+bitset板子题

我们先把输入的数字离散化一下，然后询问的时候要把一个询问拆成3个询问，然后再并起来

然后在莫队的时候记录cnt_i表示当前数字i出现的次数，再开一个$bitset$，假设当前需要加入$bitset$的数字是x，我们就令$bitset$中的第$p_x + cnt_x$位为1，然后$++cnt_x$

这样就会使得把三个$bitset$并起来之后，恰好剩下的就是在三个区间都出现了的数字，也就是需要删除的数字，我们就只需要看并起来之后还剩多少个1就行了

还有一个问题，就是我们必须要对每一个询问(拆之前)都要开一个$bitset$来存答案，但这会开不下

所以我们就要平衡一下空间复杂度和时间复杂度，把询问分为几块来处理

神树大人读法书

给定一个长度为n的数列a_i，你需要支持m次下面两种操作

- $1\ x\ v$，把第x个数字修改为v
- $2\ l\ r\ x\ y$，求有多少个极大下标集合，其中的下标连续且在$[l,r]$内，且同一集合中的下标对应的数组中元素均相同且在$[x,y]$内

$n,m \leq 2 \times 10^5$

神树大人读法书

我们发现这个东西不好直接维护，所以可以考虑维护每个连续段最左边的数

神树大人读法书

我们发现这个东西不好直接维护，所以可以考虑维护每个连续段最左边的数

我们定义一个数组b_i，其中$b_1 = a_i, b_i = [a_i \neq a_{i-1}] \cdot a_i$

神树大人读法书

我们发现这个东西不好直接维护，所以可以考虑维护每个连续段最左边的数

我们定义一个数组b_i，其中$b_1 = a_i, b_i = [a_i \neq a_{i-1}] \cdot a_i$

然后我们对于每组询问，实际上就是看b序列中有多少$[x, y]$之间的数在$[l, r]$之间，然后再特殊考虑一下位置l就可以了

神树大人读法书

我们发现这个东西不好直接维护，所以可以考虑维护每个连续段最左边的数

我们定义一个数组b_i，其中$b_1 = a_i, b_i = [a_i \neq a_{i-1}] \cdot a_i$

然后我们对于每组询问，实际上就是看b序列中有多少$[x, y]$之间的数在$[l, r]$之间，然后再特殊考虑一下位置l就可以了

这个东西在没有修改的时候可以用主席树，有修改的时候可以用CDQ分治或者树套树

观波加奈

题目蓝链

给你一棵n个节点的树，你需要支持m次下面两种操作

- $1\ x\ y\ z$，给x到y链上的每一个点加上权值z
- $2\ x\ k$，查询距离$x \leq 1$的所有点中第k大的权值是多少

$n, m \leq 4 \times 10^5$

~~原题的数据范围更大，但是由于我不会证复杂度，所以我就缩小了数据范围~~

观波加奈

我们可以考虑根号的怎么做 ~~虽然和正解没有什么关系~~

观波加奈

我们可以考虑根号的怎么做 ~~虽然和正解没有什么关系~~

我们对每一个点的度数进行分治

观波加奈

我们可以考虑根号的怎么做 ~~虽然和正解没有什么关系~~

我们对每一个点的度数进行分治

如果查询的点度数$\leq \sqrt{n}$, 则暴力查询所有一圈的点

观波加奈

我们可以考虑根号的怎么做 ~~虽然和正解没有什么关系~~

我们对每一个点的度数进行分治

如果查询的点度数$\leq \sqrt{n}$, 则暴力查询所有一圈的点

然后修改的时候考虑对所有度数$> \sqrt{n}$的点的贡献即可

观波加奈

我们可以考虑根号的怎么做 ~~虽然和正解没有什么关系~~

我们对每一个点的度数进行分治

如果查询的点度数$\leq \sqrt{n}$, 则暴力查询所有一圈的点

然后修改的时候考虑对所有度数$> \sqrt{n}$的点的贡献即可

进行一次根号平衡,用一个$\mathcal{O}(\sqrt{n})$链加，$\mathcal{O}(1)$查单点的树分块来实现修改操作

观波加奈

我们现在来考虑~~正解~~怎么做

观波加奈

我们现在来考虑~~正解~~怎么做

树链剖分

观波加奈

我们现在来考虑~~正解~~怎么做

树链剖分

一个点的孩子中只有一个不是重链头，每次修改只会修改$\mathcal{O}(\log n)$个重链头的值

观波加奈

我们现在来考虑~~正解~~怎么做

树链剖分

一个点的孩子中只有一个不是重链头，每次修改只会修改$\mathcal{O}(\log n)$个重链头的值

对于每个点用一个数据结构，维护除了他自己、父亲、重儿子之外所有相邻点的值

观波加奈

我们现在来考虑~~正解~~怎么做

树链剖分

一个点的孩子中只有一个不是重链头，每次修改只会修改$\mathcal{O}(\log n)$个重链头的值

对于每个点用一个数据结构，维护除了他自己、父亲、重儿子之外所有相邻点的值

自己、父亲、重儿子在询问的时候查找一下就好了

观波加奈

我们现在来考虑~~正解~~怎么做

树链剖分

一个点的孩子中只有一个不是重链头，每次修改只会修改$\mathcal{O}(\log n)$个重链头的值

对于每个点用一个数据结构，维护除了他自己、父亲、重儿子之外所有相邻点的值

自己、父亲、重儿子在询问的时候查找一下就好了

修改$\mathcal{O}(\log^2 n)$，查询$\mathcal{O}(\log n)$

Thanks