# 可持久化数据结构

数据结构的可持久化的定义:在一次操作(增删改查)后，保留操作前的版本，也就是保留历史版本，

这里最无脑的办法是一次操作(增删改查)前对其进行深拷贝，这样便保留了历史版本，但是这么做来保留历史版本就会面临问题是夹杂着大量重复数据(数据库里叫”脏数据”)，所有可持久化数据结构都是围绕如何减少脏数据来实现的，可持久化数据结构的**核心**就是如何保留历史版本和减少脏数据。

什么样的数据结构能可持久化？一个数据结构能可持久化要满足：

1. 保留一次历史版本所占用的空间复杂度和时间复杂度都小于等于O(logn),
2. 最坏空间复杂度和时间复杂度都小于等于O(logn)，所以均摊时间复杂度为log(n)但最坏为O(n)的伸展树splay不能可持久化，原因在于这样做保留多次历史版本后，那些最坏情况的出现会被”放大”，拉低效率。

* 可持久化数据结构最好使用伪指针索引，因为乘法索引在森林行不通，而指针索引的对于频繁插入删除效率不行，而且考虑操作系统层的寻址问题，伪指针索引的内存连续，显然更快。
* 对于查询操作，如果查询后数据结构不发生变化，则不必对这个操作可持久化，但插入和删除操作一般是要可持久化的

## 静态可持久化线段树：

区间第k小问题:

给定序列A[0]到A[n-1] 和m次询问 (1<n<1e5,abs(A[i]) <1e9 )

每次询问区间[l,r]内第k小的数是多少？

我们知道，求序列整体的第k小有2种常用方法，一个是平衡树，一个是权值线段树，具体原理参考数据结构的其他文章。在此权值线段树的基础上强化就诞生了可持久化权值线段树。

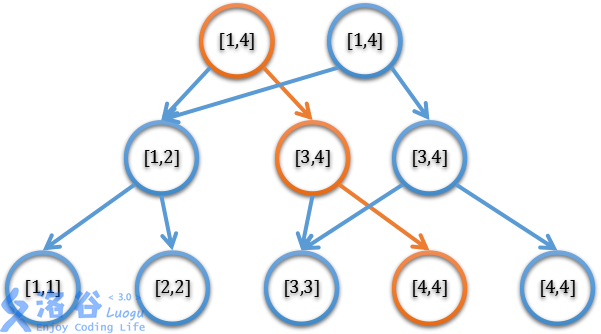
**可持久化权值线段树** ：简称主席树，可持久化线段树，函数式线段树

是基于权值线段树的数据结构，权值线段树处理范围较大的离散数据需要离线做(大于1e6)，主席树既然基于权值线段树，也是一种离线数据结构。

先对所有数据排序 (指A[i])，用数组建立 <顺序,数值> 的索引,已知顺序索引数值直接用数组下标即可。若是已知数值去索引顺序，用二分查找或者哈希表预处理都可以。得到不同的数字的个数设为up,用up建树维护区间[0,up)。

### 属性：

主席树是森林，严格来说是很多棵二叉树的复合体，直观的印象如下图 (有2个根时候的样子)：



需要注意的是，普通线段树有三种索引方式，乘法索引和指针索引和拟指针索引。

主席树不能像线段树那样，通过计算(now\*2和now\*2+1)得到左右儿子，只能像其他普通二叉树那样，指针索引和拟指针索引实现。再有就是主席树空间很大，线段树的非必要信息，如左右端点信息，尽量就省略不维护了。

树的节点属性有：son [2]代表子树位置，sum代表区间和

此外还要维护root[]数组，root[i]代表第i个根的下标。如果用拟指针索引，还要定义节点集合数组:tree[]，实际证明，拟指针索引比指针索引要快一些。

这种用数组维护可持久化权值线段树的根的数据结构，叫静态主席树。

实际上静态主席树的代码并不多，但静态主席树不支持删除和修改操作

### **建树：**

利用线段树的建树方式，建立维护区间[0, up)的树，这颗树是空的sum=0，到这里还和权值线段树差不多。不要忘给root[0]赋值代表空树树根是root[0]

我们开始把每个A[i]插入刚才建的树里。

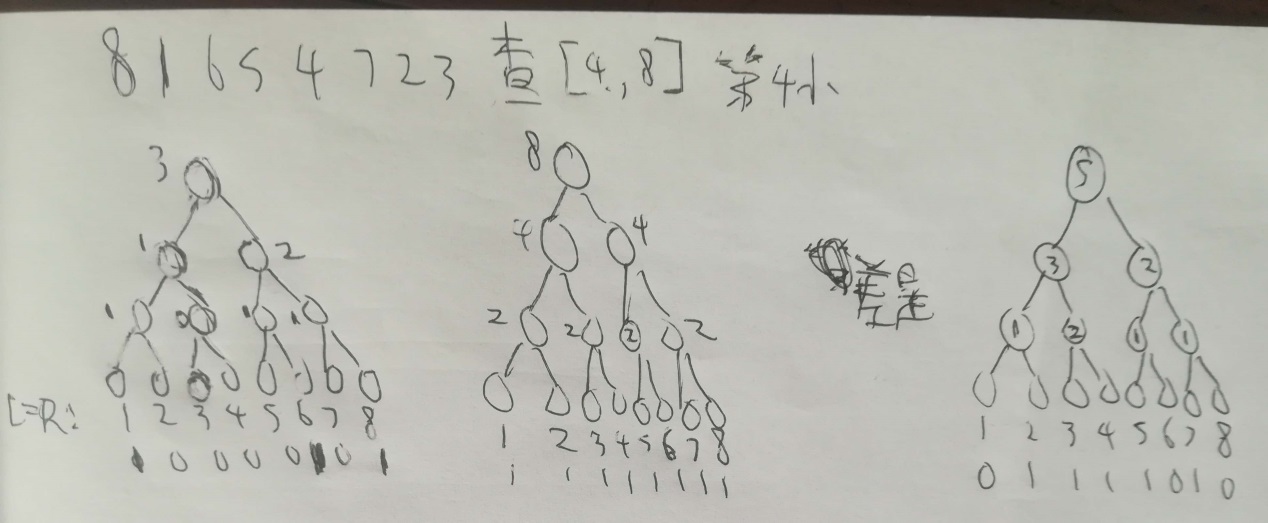
### 插入:

就是类似线段树单点更新，但是不去修改节点，而是建新的节点。

对于插入一个A[i],得到离散化后的数值x=ind[A[i]]，遍历上一棵树root[last],主席树里插入x,类似权值线段树插入x，看mid和x的大小关系，x<mid向左，反之向右，递归过程不修改节点，而是每层添加新节点，建立方式参考图中，相当于一次多出了logn个节点。

这里要注意从，每次插入操作，会建立新root，要记录下来。

### 查询区间第k小:



对于查询区间第k小问题，先把数据离散化，使得他们紧凑。

然后建立一个空的主席树，每个节点sum是0

对于插入操作，用主席树的插入操作去插入,插入的第i个数A[i]对应的那个根节点的树，就代表了前i个数所建立的权值线段树。我们能利用这些权值线段树得到：前i个数的第k小(权值线段树查询总体第k小参考其他文章)。

对于查找前n个数的区间[x,y]第k小(x和y从1开始)

可以通过查询前y个数第k小和前x个数第k小得到。

方法：我们分别从root[x-1]和root[y]这两棵树的查找，对于各自每次走的节点，

设它们左儿子的sum属性的差值是d:

如果d>=k, 说明被找的数在左边，同时往左儿子找第k小

如果d<k, 说明被找的数在右边，同时往右儿子找第k-x小，

且我们的视线是同时从root[x-1]和root[y]下降的，要找到根节点一定同时到根节点，如果到了根节点，则返回这个节点的左端点值即可。

实际上寻找区间第k小的原理，就是利用不同历史版本之间的差异性，来由普通权值线段树的查询总体第k小功能扩展出区间第k小的

## 动态可持久化线段树：

区间第k小的问题:

给定序列A[0]到A[n-1] 和m次询问 (1<n<1e5,abs(A[i]) <1e9 )

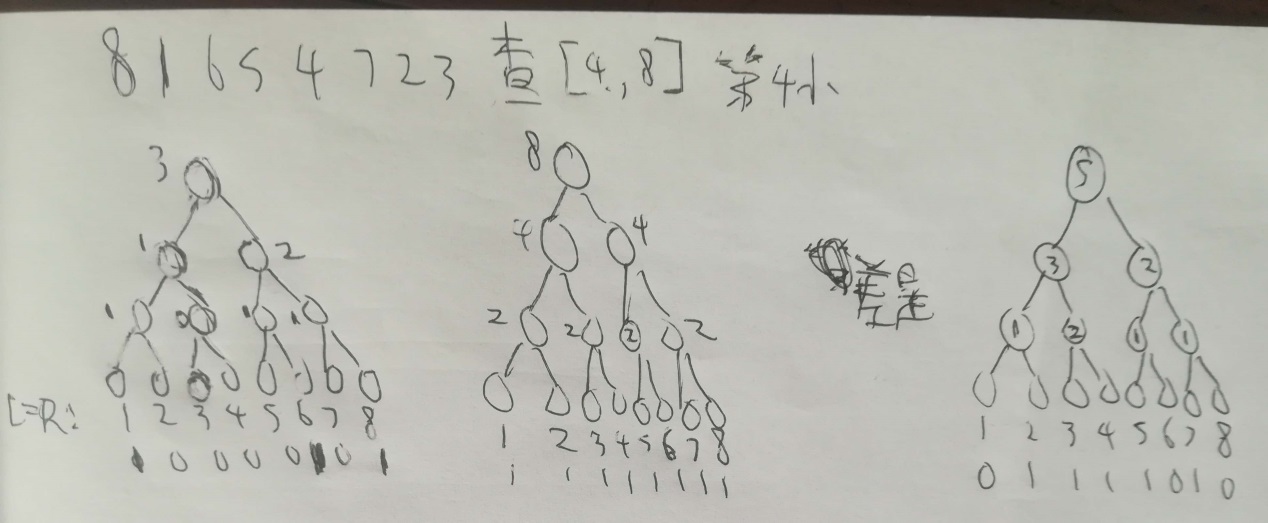
每次询问有两种:

* 区间[l,r]内第k小的数是多少？
* 修改A[i]的值

可见是在静态区间第k大问题里增加序列单个数字的修改操作

静态主席树不能支持任何修改操作，不能解决这个问题。

现在给一组数据，考虑查询区间第k小的过程，来考虑如何修改数据：



对于上述数据，可以构建8棵权值线段树编号1到8，和一个空树编号0。

图中给出了第3和8棵权值线段树，以及它们的差。

对于操作：把第i个位置由x变成y，按照暴力算法，方法是：修改所有历史版本对应的权值线段树，检查它的叶子的x位置上是否大于0，是就修改，即：对应x的位置-1，y位置+1。这么做一次复杂度显然是n\*logn，修改m次就是m\*n\*logn，时间爆炸

考虑本题是单点修改，而权值线段树本质又是区间和，这是树状数组最擅长的操作。怎么用呢，要横向看(从第0棵权值线段树到第n棵权值线段树)，看每个对应位置的节点，让他们组成树状数组。为了方便描述，现在用tree(i,j)代表第i棵权值线段树的第j个节点,j是按照完全二叉树节点编号来算的(自上至下，自左至右)。

设每棵权值线段树的节点数是N，则[tree(0,j), tree(1,j), tree(2,j),…,tree(n,j)]

构成一个树状数组，N\*(n+1)个节点共N个树状数组。

是用树状数组的一些性质来支持静态主席树单点修改，这种数据结构叫做动态主席树，全称**树状数组套可持久化权值线段树**

### 插入:

### 查询区间第k小:

## 可持久化fhq\_treap

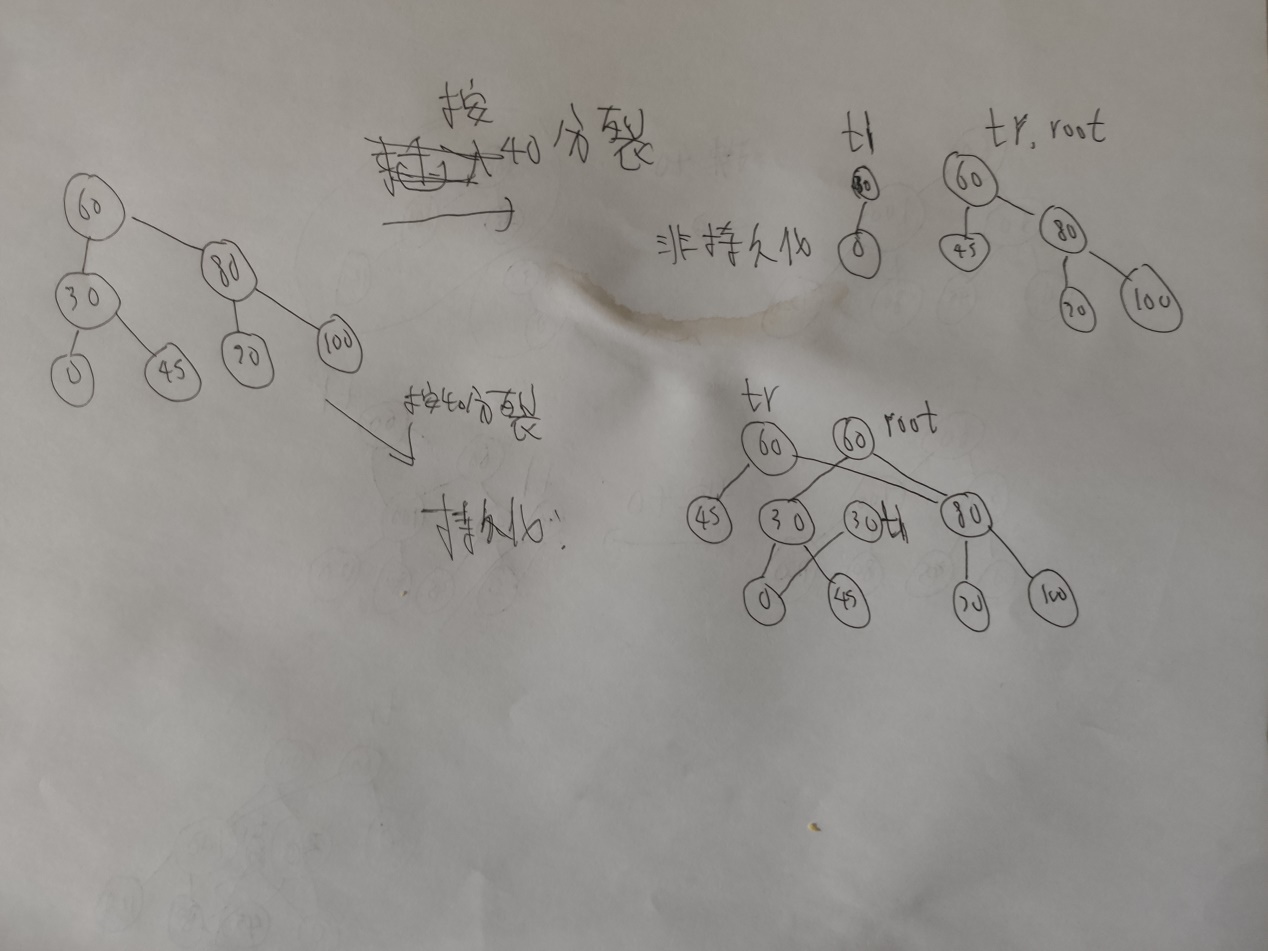
Fhq\_treap可持久化体现在插入删除，保存历史版本，主要是对于split操作的改变

### 属性

树的节点属性有：son [2]代表子树位置，val代表维护的数据

此外还要维护root[]数组，root[i]代表第i个根的下标。如果用拟指针索引，还要定义节点集合数组:tree[]

### 持久化分割split(r,k,tl,tr)



可持久化fhq\_treap的split操作就是对递归路径上的节点进行深拷贝，递归的思路照常，只需要在递归前，拷贝一份当前节点，然后递归时对当前节点的相应分枝递归。

* 由于持久化分隔和非持久化分隔区别仅仅在是否拷贝路径节点，我们可以把二者写在一起split(r,k,tl,tr,per)，通过per来判断，per=1代表持久化，per=0代表非持久化

void \_\_split(Node \*r,int k,Node \*&tl,Node \*&tr,bool (\*cmp)(int a,int b),bool per){

if(r==NULL){//到达叶子

tl=NULL;

tr=NULL;

return;

}

if(cmp(r->val.k,k)){

tl=per?r->clone():r;

\_\_split(tl->son[1],k,tl->son[1],tr,cmp,per);

tl->updataSumcnt();

}else{

tr=per?r->clone():r;

\_\_split(tr->son[0],k,tl,tr->son[0],cmp,per);

tr->updataSumcnt();

}

}

### 合并merge(x,y)

和普通fhq\_treap相同

* 网络上流传的代码的合并操作需要复制路径的节点，这样虽然结果正确但浪费了空间和时间，与持久化数据结构尽可能减少多余数据的原则不符。因为我们合并前必然要分裂，如果用的持久化分裂，就已经保存了历史版本，此处合并时再保存就是多余了。

### 持久化插入insert(verId,val)

代表在第verId版本下进行插入节点，最终生成新的版本添加到root数组。

思路：先split(val.k,tl,tr);再new node建立新的节点且merge(tl,node)也就是把tl,node合并，最后把merge(tl,node)和tr合并，合并结果保存在新版本中。

这里split用的是持久化版本，目的是保存历史数据，因为插入操作必然伴随版本更新

Node\* insertNode(int verID,Data val){//在版本verID下插入

Node \*tl,\*tr;

split(root[verID],val.k,tl,tr,1);

root[rlen++]=merge(merge(tl,new Node(val)),tr);

return root[rlen-1];

}

### 持久化删除delete(verId,val)

代表在第verId版本下进行删除节点，最终生成新的版本添加到root数组

删除操作大体和非持久化fhq\_treap一样。

这里split用的是持久化版本，目的是保存历史数据，因为删除操作必然伴随版本更新

bool deleteNode(int verID,Data val){//在版本verID下删除

Node \*tl,\*tr,\*now;

split(root[verID],val.k,tl,tr,1);

\_\_split(tl,val.k-1,tl,now,[](int a,int b)->bool{

return a<=b;

},1);

if(now&&now->val.k==val.k){

now=merge(now->son[0],now->son[1]);

root[rlen++]=merge(merge(tl,now),tr);

return true;

}else{

root[rlen++]=merge(merge(tl,now),tr);

return false;

}

}

### 查询操作

对于各种查询操作(如查询排名，前驱，后继，第k小/大)都没有版本更新，不要用持久化的分裂，也不用在合并后加入到root[]的新位置，但如果有的题目要求查询操作也算作新版本，则也只需要在查询完毕后复制对应根即可。

## 可持久化并查集

## 可持久化可并堆