# 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. 所有操作都是自上而下(从树顶向下走)
2. 保留一次历史版本所占用的空间复杂度和时间复杂度最坏也是对数级
3. 最坏空间复杂度和时间复杂度都最坏也是对数级，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 )

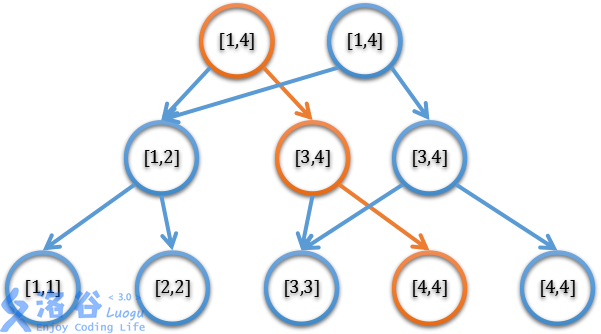
每次询问区间[l,r]内第k小的数是多少？

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。

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)：图来自洛谷



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。

* 在区间第k小问题里，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：

Struct Node{

son [2]代表子树位置

sum代表区间和

}

维护root[]数组，root[i]代表第i个根的下标，root[i]是一个版本的根，也叫第i个**版本树**，在主席树里，每个版本树都是权值线段树

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。

* 数组维护多颗可持久化权值线段树根的数据结构，叫静态主席树

### **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。

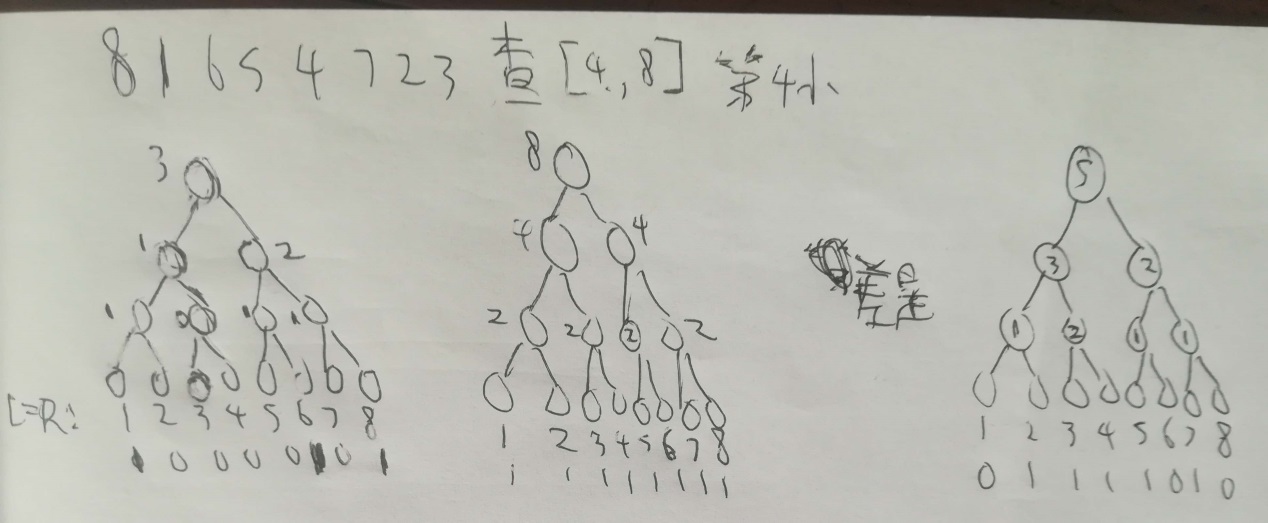
这里要注意从，每次插入操作，会建立新root，要记录下来。

### 单点历史版本查询/可持久化数组

对于查询第i各版本的第j个位置的值，就从root[i]开始走，类似二分查找那样，如果mid>=j向左，mid<j向右，时间复杂度O(logn)

* 当作可持久化数组用时，是在线算法，叶子节点储存数据，其他节点的数据无意义。
* 此外，网络上对于应用单点查询和修改的可持久化权值线段树，还称为可持久化数组，原本静态主席树不支持修改，但在这里我们只维护叶子节点，非叶子节点属性无意义，所以当作线段树那样修改只需要修改一个值，所以当作可持久化数组时，单点修改有意义。

### 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

//问题就是在静态区间第k小问题里增加序列单个数字的修改操作。

有的人说，静态主席树基础上，直接找到要修改的点，更改后把父亲的sum更新不就行了吗？但注意看主席树的形态和算法可知，主席树在插入过程中的路径上节点会被复制，一个节点可以同时在多个版本树里充当父亲。该问题里，你修改一个点时，要顺带着把它的全部父亲的线段树修改，着使得复杂度降为logn\*m,m为当前版本的数量。显然这个时间消耗是不可接受的。

我们需要对主席树改造，思路如下：

考虑直接对主席树修改复杂度logn\*m,这个m很大，因为我们每个版本就是插入数据的”前缀和”，最多有n个数，那m最大是n,如果能降低这个m的值，让它少一点，少到O(logn)级别，再乘以每次更新主席树的O(logn)，复杂度是O()

如何降低这个m的值呢？那就改变主席树里权值线段树的意义，第i个版本树的叶子节点，不再代表前i次插入后每个权值的数量。**现在第i个版本树代表**:**以i为根的树状数组的叶子的编号构成的节点的插入信息所组成的权值线段树**

说的有点绕，但要仔细理解，比如：第6个版本树不是代表前6次插入构成的权值线段树，而是第5和6次插入节点构成的权值线段树，这是因为树状数组的编号为6的节点的值是由第5和6个叶子相加得来。同理，第7个版本树仅仅是第7次插入节点构成的权值线段树，因为树状数组的编号为7的节点的值只由第7个叶子得来。

又由于根节点数组root[i]代表第i个版本树的根

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**

### 属性：

### **建树：**

### 插入:

### 单点修改

### 查询区间第k小:

## 可持久化fhq\_treap

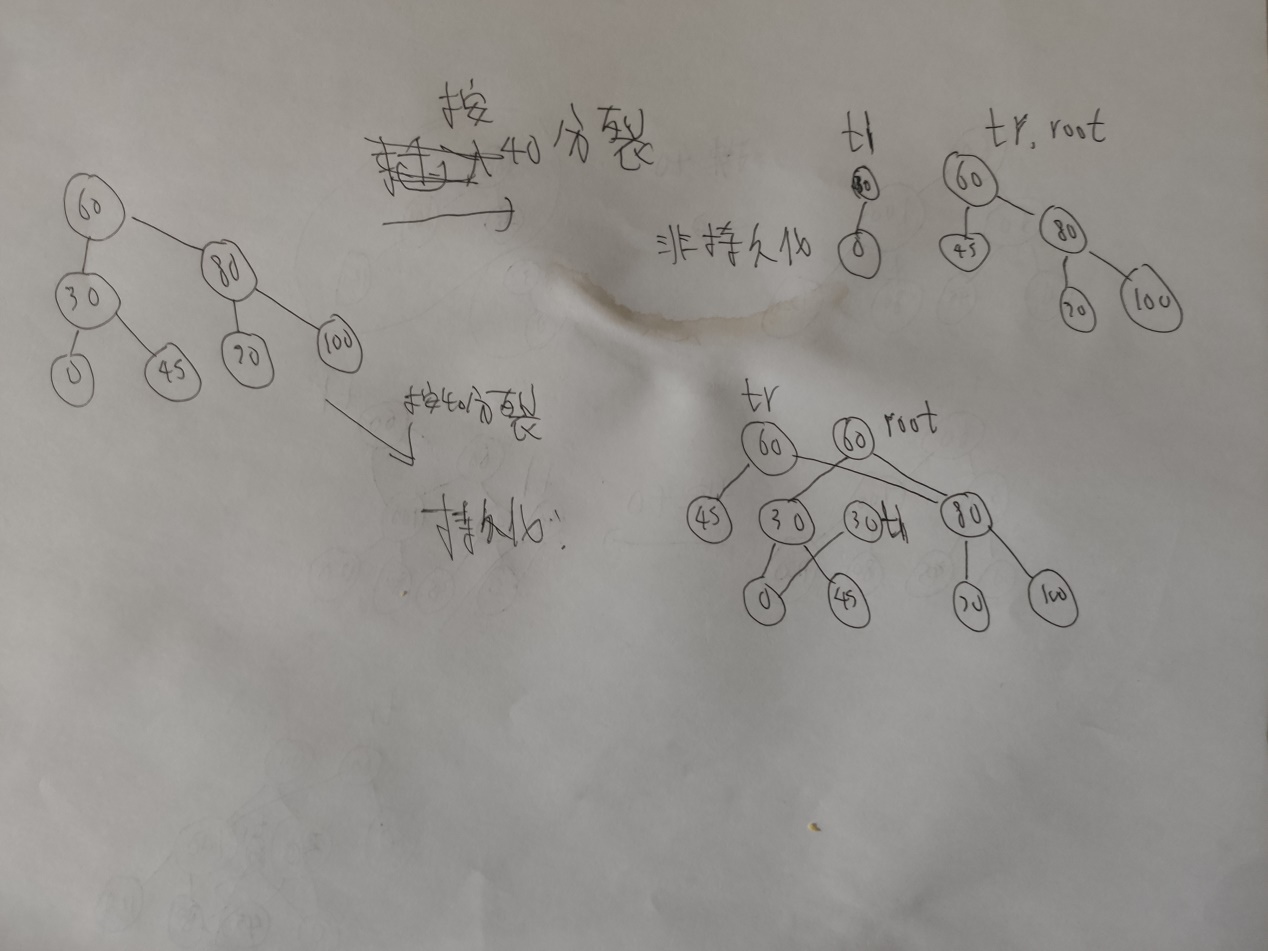
Fhq\_treap可持久化体现在插入删除，保存历史版本，主要是对于split操作的改变

### 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

并查集的可持久化，就是用可持久化线段树来维护并查集的点，然后利用可持久化单点修改和查询来实现并查集节点的可持久化，这种数据结构叫可持久化并查集。

维护的元素是什么？是并查集的p和h属性，也就是并查集当前节点的父亲地址和自身高度

* 在写代码时，应该明确一个点到底是并查集的还是主席树的，正确是：主席树的叶子节点是并查集节点，也就是包含关系，可持久化并查集相比普通并查集多了主席树非叶子节点的内存消耗。

### 属性

struct Node{ //代表持久化线段树的节点

p 代表并查集节点的父亲地址，是并查集节点

h 代表并查集节点在并查集中的高度

son[2] 主席树节点的2个儿子，

* 注意：p和h在非并查集节点时，的主席树中无意义，所以可以用2个类体定义主席树节点，把主席树的叶子节点和非叶子节点区分开来，以此节省空间，但题目卡空间一般不严，不必这么写

}

struct Per\_union{//持久化并查集类

size主席树节点集合的上限，也代表了当前主席树节点的数量，采用伪指针索引时必要，指针索引没有

rlen根节点集合的上限，也代表了最新版本的编号

up 并查集维护集合的数量上限

root[MAX]; 根节点集合，作用同主席树

tree[MAX<<5]; 主席树节点集合，采用伪指针索引时必要，指针索引没有

};

### 建树&&初始化

思路类似主席树，区别为：

到叶子时，应该把此时节点的h变为1，代表初始时高度为1，把p变成节点自身地址，伪指针索引里编号就是线段树区间的l

### 查询verID版本并查集节点pos的地址

*getPos(int now,int tl,int tr,int pos)*

就是主席树单点查询操作，返回改节点地址

就是某个版本的主席树的叶子在并查集中的地址，他等同于普通并查集直接用数组索引，所以这个操作是主席树当持久化数组用法

### 修改verID版本并查集节点pos中的高度+1

*updateH(int verID,int pos)*

*\_\_updateH(int now,int tl,int tr,int pos)*

*把pos位置节点代表的并查集节点深度加一,非持久化修改*

原本静态主席树不支持修改，但在这里我们只维护叶子节点，非叶子节点属性无意义，所以当作线段树那样修改只需要修改一个值，所以单点修改有意义。

类似单点查询操作，找到叶子把h属性+1

### 更新verID版本并查集节点pos中的p

*per\_updateP(int verID,int pos,int pval)*

*//修改第verID版本的并查集pos位置所储存的父亲为pval*

*\_\_per\_updateP(int now,int tl,int tr,int pos ,int pval)*

*//把pos位置改为pval，持久化修改,返回根*

相当于可持久化线段树中的插入操作，利用单点查询的思路寻找目标节点，每次递归时深复制当前节点，对目标节点来说，一定是并查集节点，也要复制，并且还得修改p为想要的

该函数完成后，由于创建了历史版本，还要root[rlen++],保存新版本的根地址

### 并查集节点找根getRoot(verID, pos)

和普通并查集一样思路，只不过每次由数组访问，改为调用主席树单点查询操作

注意：该方法最好返回主席树节点的地址，而非并查集节点地址，这样能一起得到并查集节点的深度h

### 并查集节点判断isSame(verID, x, y)

和普通并查集一样思路，调用getRoot即可

### 并查集节点合并merge(verID, x, y)

和普通并查集一样，要先判断是否x,y已经属于同一集合，属于则什么都不做，否则启发式或合并，把高度较小的放在高度较大的儿子，这里连接两棵树，用的是主席树的per\_updateP操作，

## 可持久化可并堆