可持久化数据结构

数据结构“可持久化”的含义就是可以维护历史版本。

线段树和 fhq-treap 都是可持久化的，具体方法会在后文提到。

因为线段树和平衡树可以维护任意序列，因此数组和并查集（它只需维护 `fa` 数组）也是可持久化的。

链表也是可持久化的（不过一般没啥用）。

ST表没有修改所以无所谓可持久化。

因此我们学过的数据结构中除了 Splay 以外都是可持久化的。

可持久化线段树

可持久化（权值）线段树是一切可持久化数据结构的基础。

它的原理其实非常简单，只是在每次修改操作时，对**所有受到此次修改操作影响的节点**的所有信息全部复制一遍，新建一个节点。并记录这个版本的根节点。

因此，可持久化线段树**只能动态开点**。它的时间和空间复杂度都是 $O(m\log n)$。

特别注意：如果操作是区间修改，**每次下放懒标记时都要新建节点！**

但事实上，这种直接把“可持久化”写在题面上的题目并不多，它的更多应用还在下面。

“位置+权值”

如果我们要同时支持对序列的位置和权值信息进行查询，那么序列线段树和权值线段树都难以实现。

此时，我们一般会**将序列的位置信息看作版本，建立可持久化权值线段树**。这种线段树就称为”主席树“。

注意，主席树不支持修改操作。

> ❔ 给你一个序列，要求支持查询区间第 k 大。
>
> $1\le n,m\le 10^5$。

如上文所说，将前 i 个元素形成的权值线段树看作第 i 个版本。

因为权值线段树维护的是每个值的元素个数，所以信息是可以做差的。

因此 $[l,r]$ 的元素形成的权值线段树就是第 r 个版本与第 $l-1$ 个版本的差。

复杂度 $O((n+m)\log n)$。

```cpp
#include<bits/stdc++.h>
using namespace std;
typedef long long ll;

const int N=2e5+10,V=N*32,L=1e9;
int n,m,x,l,r,k,rt[N],lc[V],rc[V],s[V],tot=0;
inline int copy(int x){
        int y=++tot;
        s[y]=s[x],lc[y]=lc[x],rc[y]=rc[x];
        return y;
}
#define mid ((l+r)>>1)
void mdf(int& x,int l,int r,int p){
        x=copy(x),++s[x];
        if(l==r)return;
        if(p<=mid)mdf(lc[x],l,mid,p);
        else mdf(rc[x],mid+1,r,p);
}
int qry(int x,int y,int l,int r,int k){
        if(l==r)return l;
        int t=s[lc[x]]-s[lc[y]];
        if(k<=t)return qry(lc[x],lc[y],l,mid,k);
        return qry(rc[x],rc[y],mid+1,r,k-t);
}
int main(){
        ios::sync_with_stdio(0);cin.tie(0);
        cin>>n>>m;
        for(int i=1;i<=n;++i){
                cin>>x,rt[i]=rt[i-1];
                mdf(rt[i],-L,L,x);
        }
        while(m--){
                cin>>l>>r>>k;
                cout<<qry(rt[r],rt[l-1],-L,L,k)<<'\n';
        }
}
```

给你一个序列，要求支持查询区间不同数字个数。

$1 \le n \le 2 \times 10^6$。

本题有树状数组的离线算法，但我们解决问题时一般更希望寻求在线算法。

直接求区间不同数字个数不好维护（因为这个信息不能减）。我们考虑反面，即区间重复数字。

如果某个数字在区间内多次出现，那么它上一次出现的位置还在区间内。

因此先用链表求出每个数字上一次出现的位置 l_i，然后问题转化为：

区间 $[l,r]$ 内有多少个 l_i 在 $[l,r]$ 之间。

~~这不是三维数点吗？树状数组！~~

主席树可以在线解决这个问题。将前 i 个数看作第 i 个版本，建立 l_i 的权值线段树。

询问即第 r 棵权值线段树减去第 $l-1$ 棵权值线段树的 $[l,r]$ 之和。

复杂度 $O((n+m)\log n)$。

从第 1 秒到第 n 秒共有 m 个任务，第 i 个任务从第 s_i 秒开始，到第 e_i 秒结束，优先级为 p_i。

有 n 个查询，第 i 个查询查第 x_i 秒时优先级前 k_i 小的任务优先级之和。

$1 \le n,m \le 10^5$

将第 i 秒 p_i 的权值线段树作为第 i 个版本。

因为要求前 k 小的优先级之和，所以要同时维护 cnt 和 sum。

对于每个任务，因为它在第 s_i 到第 e_i 个版本都存在，所以类似于差分数组，在第 s_i 个版本上的 p_i 位置加 p_i，在第 e_i+1 个版本上的 p_i 位置减 p_i。

然后剩下的就是在第 x_i 个版本上线段树二分查前 k 小之和了。

复杂度 $O((n+m)\log n)$。

给你一个 $n \times m$ 的矩阵，q 次查询一个子矩阵内最少选出几个数和不小于 k。

$1 \le a_{i,j} \le 1000$。$1 \le n,m \le 200, 1 \le q \le 200000$ 或
$1 \le n \le 500000, m = 1, 1 \le q \le 20000$。

看数据范围就知道这是一道二合一的题。

因为数据范围很小，所以直接设 $c(i,j,k)$ 和 $s(i,j,k)$ 为前 i 行 j 列 $\geq k$ 的数的数量、和。

这部分可以 $O(nmw)$ 预处理。

然后每次询问二分 k，用经典的二维前缀和套路。

这部分复杂度是 $O(nmw+q\log w)$。

但这并不能处理 $m=1,n\leq 500000$ 的数据。因为 $O(nw)$ 的空间是开不下的。

这部分用主席树解决。前 i 个数形成的权值线段树作为第 i 个版本。

询问时直接线段树上二分即可。其实因为 q 很小，直接二分 k 都可以。

复杂度 $O((n+q)\log w)$ 或 $O(n\log w+q\log^2 w)$。

> 有 n 种商品，每种商品价值 d_i，价格 p_i，有 l_i 个。
>
> 有 q 个询问，每次询问用不多于 g_i 的钱买不少于 L_i 个商品买到的价值最小的商品的最大价值。无法做到输出 -1。
>
> $1\leq n,q\leq 10^5$

最大化最小值——二分答案。

先按 d_i 从大到小排序，然后建立主席树。

第 i 个版本即所有价值不小于 d_i 的物品形成的权值线段树。

在权值线段树上同时维护物品个数和物品价格总和，线段树二分即可求出价格最小的 L_i 个商品的价格之和（注意 `l==r` 时的细节处理）。将其与 g_i 比较即可。

复杂度 $O(n\log n+q\log^2 n)$。

> 给 n 个数 $a_1,...,a_n$。m 次询问，每次询问给出 l,r,b,x，求区间 $[l,r]$ 中 $b\oplus(x+a_i)$ 的最大值。
>
> $1\leq n,m,a_i,b,x\leq 10^5$

求异或最大值可以想到 01trie 树。但我们不能对一棵 01Trie 树上的所有数进行加法操作，因为这样会改变它的结构。

因此 01trie 树是不能建出来的，我们只能采用它的思想，即对 $x+a_i$ 按位从高位到低位贪心。

设当前考虑到第 k 位，$x+a_i$ 的前 $k-1$ 位已经确定是 A。

若 b 的第 k 位是 0，那么我们就要找到 $[l,r]$ 中的一个 i 使得 $x+a_i$ 的前 $k-1$ 位是 A 且第 k 位是 1。

即查询第 r 个版本与第 $l-1$ 个版本的差的 $[A-x+2^k, A-x+2^{k+1}-1]$ 是否有值。

如果有， $A+=2^k$ 。

同理，若 b 的第 k 位是 1 ，就要查 $[A-x, A-x+2^k-1]$ 。

复杂度 $O(n\log n\log w)$ 。

树上主席树

从上面的几个例子已经看出，主席树解决“位置+权值”问题的主要思想就是前缀和。

而树上也是有”前缀和“的，即自根向下的路径。

因此我们不难把序列上的主席树扩展到树上。

当然，树上主席树也是不支持修改的。。

> 给一棵树，多次询问两点路径上的第 k 小点权。强制在线。
>
> $1\le n,m\le 10^5$

主席树（包括一切可持久化数据结构）的”版本“概念是很灵活的。第 i 个版本并不一定要以第 $i-1$ 个版本为基础。事实上，在一般情况下，版本形成的就是一棵树。所以序列的主席树其实是树上主席树的特殊情况。

我们设第 i 个版本为从根到节点 i 的路径上点权形成的权值线段树。

这样第 i 个版本就是在第 fa_i 个版本的基础上插入 v_i。

下面考虑两点路径上的点形成的权值线段树。

根据 $dis(u,v)=dis(u)+dis(v)-dis(LCA(u,v))-dis(fa(LCA(u,v)))$ ，

它是这四棵权值线段树的和（差）。代码和两棵主席树作差类似。

> 给一片森林，要求支持两种操作：
>
> `L x y` 加一条连接 x,y 的边，保证连边后不出现环；
>
> `Q x y k` 查询 x 到 y 的路径上第 k 大的边。
>
> $1\le n,m,q\le 10^5$

我们先按开始给出的森林建出树上主席树以及求 lca 所用倍增数组。

查询和上道例题相同。

因为加边不会成环，所以加边前两个点一定不连通。

因此我们**暴力重建**点数较少的连通块的所有主席树和倍增数组，复杂度是对的。

总复杂度为 $O(n\log^2 n+q\log n)$。

说起来简单，实际上并不好实现，而且坑点极多。请独立完成本题代码。

带修主席树

带修主席树，又称”树状数组套权值线段树“。

> 给一个序列，要求支持单点修改、区间查询第 k 小值。
>
> $1\le n,m\le 10^5$。

我们看看主席树为什么不能修改。

如果修改了第 i 个数，那么从第 i 个数开始所有的版本都要改。然后就爆炸了。

当年遇到这个问题的时候，我们引入了树状数组平衡了修改和查询的复杂度。

那现在我们也用树状数组。

我们令主席树的第 i 个版本是第 $i-\text{lowbit(i)}+1$ 个数到第 i 个数形成的权值线段树。

这样根据树状数组的原理，修改只需改 $O(\log n)$ 个版本（不断跳 lowbit 即可），复杂度 $O(\log^2 n)$。

查询时，仍然转换为两个前缀的差，而每个前缀都是 $O(\log n)$ 个版本的和。

因此查询的复杂度也是 $O(\log^2 n)$。

总复杂度 $O(n\log n+m\log^2 n)$。

注意本题空间！因为修改会建大量的新节点，所以开 $20n$ 远远不够，为保险起见至少要开 $200n$。

本题还有分块+二分的简单写法，复杂度为 $O(n\sqrt{n}\log n)$，也可以通过本题。

> 给一棵树，要求支持单点修改，查询链上第 k 小值。
>
> $1\le n,m\le 8\times 10^4$。

如果直接建树上主席树，还是会遇到和上一题一样的问题，即修改一个点就会导致整棵子树的主席树全被修改。

”修改整棵子树“可以想到？DFS 序 + 树状数组！

因此我们用 DFS 序对树上主席树重标号。

这样问题就变成了《区间修改，单点查询》，对这些主席树求一个差分，然后就和上一题一样了。

请独立实现本题代码，可能要调 8 个小时，但这是每个初学省选数据结构的 OIer 的必经之路。

总复杂度 $O(n\log n+m\log^2 n)$。

另外本题有一个很显然的做法树链剖分+主席树，复杂度是 $O(n\log^2 n+m\log^3 n)$ 同样能通过本题。

三维偏序问题

偏序问题是数据结构题目中很重要的一种模型。

特别是二维和三维偏序，很多看起来很复杂的题目转化后就是二维或三维偏序（有时也称为二维数点或三维数点。一般 DP 就称偏序，计数就称数点）。

二维偏序在 NOIP 阶段已经解决（第一维排序，第二维树状数组）。

三维偏序则可以用上文的带修主席树来解决。

> 空间中有 n 个点，第 i 个点的坐标为 (a_i,b_i,c_i)。
>
> 令 $f(i)$ 为 $a_j\le a_i,b_j\le b_i,c_j\le c_i$ 的 j 的数量。
>
> 对每个 $d=0\dots n-1$，求 $f(i)=d$ 的 i 的数量。

先对所有点坐标按 a_i 排序后依次加入，这样只需考虑 b_i,c_i 即可。

现在问题转化为：要支持两种操作，查询 $b_j\le b_i,c_j\le c_i$ 的点数，插入 (b_i,c_i)，二者交替执行。

我们把 b_i 离散化看作位置，这样就变成了

查询前 b_i 个位置中有多少个数不大于 c_i，在第 b_i 个位置插入 c_i

这就是带修主席树板子。

因此我们获得了一个时空均为 $O(n\log^2 n)$ 的算法。

事实上三维偏序的算法很多（少说有 10 种）。

例如：带修主席树，线段树套权值线段树，线段树套权值平衡树，K-D 树，cdq 分治+树状数组，cdq 分治套 cdq 分治，bitset 等。

理论上效率最高的算法是 $O(n\log^2 n)/O(n)$ 的。这种算法可以说是效率较高而且比较好写的一种。

其他可持久化结构

可持久化并查集

可持久化并查集支持以下基本操作：

- 在第 k 个版本上合并两个点所在集合，形成一个新的版本。
- 查询第 k 个版本上的某两个点是否在同一集合（以及集合的大小）。

可持久化并查集其实就是对并查集的 `fa` 数组的可持久化。

因为并查集的合并集合操作其实就是对 $O(\alpha(n))$ （可看作常数）个点的 `fa` 进行修改，所以它的复杂度和可持久化线段树几乎是相同的，即 $O(n\log n\alpha(n))$。

> 可持久化并查集基本都可以用更高效且更容易写的 Kruskal 重构树解决。

8.2.2 可持久化 01trie 树

可持久化 01trie 树支持以下基本操作：

- 在第 k 个版本上插入一个数，形成一个新的版本。
- 查询与第 k 个版本中的所有数异或最大（或最小）的数。
- **因为 01trie 树是权值数据结构，所以可以相减。**

01trie 树插入一个数其实就是**对一条链上的点进行修改**，这和主席树的修改是类似的。

可持久化 01trie 树最经典的应用就是**区间异或最大值**。

将前 i 个数形成的 01trie 树看作第 i 个版本，然后在第 r 个版本和第 $l-1$ 个版本的差上查询异或最大值即可。

```
const int N=6e5+5,V=1.4e7+5;
int n,m,l,r,x,all=0;
int rt[N],tot=0,c[V][2],sum[V];
char op;
void insert(int& rt1,int rt2,int val){
        int now=rt1=++tot,x=rt2,t;
        for(int i=23;~i;--i){
                t=(val>>i)&1;
                c[now][t]=++tot;
                c[now][t^1]=c[x][t^1];
                x=c[x][t];
                now=c[now][t];
                sum[now]=sum[x]+1;
        }
}
```

```
int query(int x1,int x2,int val){
        int ans=0,t;
        for(int i=23;~i;--i){
                t=(val>>i)&1^1;
                if(sum[c[x1][t]]>sum[c[x2][t]])
                        ans|=1<<i,x1=c[x1][t],x2=c[x2][t];
                else x1=c[x1][t^1],x2=c[x2][t^1];
        }return ans;
}

int main(){
        n=read(),m=read();
        insert(rt[0],0,0);
        for(int i=1;i<=n;++i)
                insert(rt[i],rt[i-1],all^=read());
        while(m--){
                op=readc();
                if(op=='A'){
                        insert(rt[n+1],rt[n],all^=read());
                        ++n;
                }
                else{
                        l=read()-1,r=read()-1;
                        printf("%d\n",query(rt[r],l?rt[l-1]:0,read()^all));
                }
        }
}
```

可持久化平衡树

可持久化的权值平衡树可以用可持久化线段树实现，因此我们着重讲解可持久化的序列平衡树。

Splay 的复杂度是均摊的，总复杂度是 $O(n\log n)$ 但少数操作的复杂度会远大于 $O(\log n)$ 甚至会达到 $O(n)$）。因此如果在那些版本上持续更新会导致复杂度退化为 $O(n^2)$。

（注：在某些题解上可能说 Splay 要维护父节点所以不能持久化。事实上这样讲是错的。 Splay 可以不维护父节点只不过实现更为复杂）

带旋 treap 和 AVL 虽然可持久化，但它们不能维护序列，只能维护权值，所以也不考虑。

所以就只剩非旋 treap 了。

可持久化非旋 treap 支持以下基本操作：

- 可持久化（序列）线段树的所有操作。
- 在某个版本中插入或删除一些元素并形成新的版本。

- 给某个版本区间翻转或易位并形成新的版本。
- **给某个版本区间复制或交换并形成新的版本**。

注意，最后一条无论可持久化线段树还是普通平衡树都无法做到。

和普通的 fhq-treap 一样，fhq-treap 同样以合并和分裂两种操作为基础来实现各种操作。因此我们只需在分裂时保留原版本并复制一个新的节点来存储修改后的信息即可。其他部分和正常的平衡树写法完全一样。

```
const int N=2e5+10,V=3e7+10;
int rt[N],tot,v[V],lc[V],rc[V],sz[V],w[V];
ll sm[V];
bool re[V];
inline int nnode(int val){
        sz[++tot]=1,v[tot]=sm[tot]=val,w[tot]=rand();
        return tot;
}
inline int copy(int x){
        v[++tot]=v[x];
        lc[tot]=lc[x],rc[tot]=rc[x];
        sz[tot]=sz[x],sm[tot]=sm[x],re[tot]=re[x];
        w[tot]=w[x];
        return tot;
}
inline void pu(int x){
        if(!x)return;
        sz[x]=sz[lc[x]]+sz[rc[x]]+1;
        sm[x]=sm[lc[x]]+sm[rc[x]]+v[x];
}
inline void pd(int x){
        if(re[x]){
                if(lc[x])lc[x]=copy(lc[x]),re[lc[x]]^=1;
                if(rc[x])rc[x]=copy(rc[x]),re[rc[x]]^=1;
                swap(lc[x],rc[x]),re[x]=0;
        }
}
int merge(int x,int y){
        if(!x||!y)return x|y;
        if(w[x]<w[y]){
                pd(x),rc[x]=merge(rc[x],y),pu(x);
                return x;
        }else{
                pd(y),lc[y]=merge(x,lc[y]),pu(y);
                return y;
        }
}
```

```
void split(int rt,int k,int& x,int& y){
        if(!rt)return x=y=0,void();
        pd(rt);
        if(k<=sz[lc[rt]]){
                y=copy(rt);
                split(lc[y],k,x,lc[y]);
                pu(y);
        }else{
                x=copy(rt);
                split(rc[x],k-sz[lc[x]]-1,rc[x],y);
                pu(x);
        }
}

int m,o,op,x,y,z;
ll ans,p,l,r;
int main(){
        m=read();
        for(int i=1;i<=m;++i){
                o=read(),op=read();
                if(op==1){
                        p=read()^ans;
                        split(rt[o],p,x,y);
                        rt[i]=merge(merge(x,nnode(read()^ans)),y);
                }else if(op==2){
                        p=read()^ans;
                        split(rt[o],p-1,x,y);
                        split(y,1,y,z);
                        rt[i]=merge(x,z);
                }else if(op==3){
                        l=read()^ans,r=read()^ans;
                        split(rt[o],r,x,z);
                        split(x,l-1,x,y);
                        re[y]^=1;
                        rt[i]=merge(merge(x,y),z);
                }else{
                        l=read()^ans,r=read()^ans;
                        split(rt[o],r,x,z);
                        split(x,l-1,x,y);
                        enter(ans=sm[y]);
                        rt[i]=merge(merge(x,y),z);
                }
        }
}
```

关于区间复制和区间交换：

为什么这两个操作不能用普通的平衡树实现：我们知道，对于区间操作，我们采用的都是下放懒标记的形式来保证时间复杂度的正确性。但这两个操作在下放懒标记时要记住**下放前**的版本，而我们接下来可能还会有区间操作要修改这棵子树，这样标记下放就出问题了。而要保留原版本的子树不变，只能用可持久化平衡树。

用分裂操作将所求两个区间各分到一棵子树，然后复制一个节点并打上标记。