可持久化数据结构模板

可持久化数组

luoguP3919 【模板】可持久化线段树 1（可持久化数组）

对于每一次操作新建一个根节点，然后动态开辟节点。

对于每一次操作1，更改其版本对应位置的值（单点），

对于每一次操作2，将根节点直接赋值过去。

```
#include<bits/stdc++.h>
#define inf 0x3f3f3f3f
#define MID int mid=l+r>>1
using namespace std;
const int maxn=1e6+7;

int read(){ int x=0,f=1;char ch=getchar();while(ch<'0'||ch>'9'){if(ch=='-')
f=f*-1;ch=getchar();}while(ch>='0'&&ch<='9'){x=x*10+ch-'0';ch=getchar();}return
x*f;}

int n,m,cnt,rt[maxn],a[maxn];
int v,op,loc,val;

struct node{
    int l,r,val;
}tr[maxn*32];

int clone(int p){
    cnt++;
    tr[cnt]=tr[p];
    return cnt;
}

int build(int p,int l,int r){
```

```
    p=++cnt;
    if(l==r){
        tr[p].val=a[l];
        return cnt;
    }
    MID;
    tr[p].l=build(tr[p].l,l,mid);
    tr[p].r=build(tr[p].r,mid+1,r);
    return p;
}

int update(int p,int l,int r,int pos,int val){
    p=clone(p);
    if(l==r){
        tr[p].val=val;
    }else{
        MID;
        if(pos<=mid) tr[p].l=update(tr[p].l,l,mid,pos,val);
        else tr[p].r=update(tr[p].r,mid+1,r,pos,val);
    }
    return p;
}

int query(int p,int l,int r,int pos){
    if(l==r) return tr[p].val;
    MID;
    if(pos<=mid) return query(tr[p].l,l,mid,pos);
    else return query(tr[p].r,mid+1,r,pos);
}

signed main(){
    n=read();m=read();
    for(int i=1;i<=n;i++){
        a[i]=read();
    }
    rt[0]=build(0,1,n);
    for(int i=1;i<=m;i++){
        v=read();op=read();loc=read();
        if(op==1){
            val=read();
            rt[i]=update(rt[v],1,n,loc,val);
        }else{
            cout<<query(rt[v],1,n,loc)<<"\n";
            rt[i]=rt[v];
        }
    }
    return 0;
}
```

可持久化并查集

luoguP3402 可持久化并查集

因为要回退第k个版本（可持久化），开个主席树。叶子节点存储父亲信息和深度。深度用来按秩合并，对同一颗树上的a,b两点合并属于单点修改logn；询问是否在同一集合，只需在当前版本查找。

```
#include<bits/stdc++.h>
using namespace std;
const int maxn=5e5+7;
const int mod=1e9+7;

int read(){ int x=0,f=1;char ch=getchar();while(ch<'0'||ch>'9'){if(ch=='-')
f=f*-1;ch=getchar();}while(ch>='0'&&ch<='9'){x=x*10+ch-'0';ch=getchar();}return
x*f;}

int n,m,cnt,rt[maxn],op,a,b,u,v,k;

struct node{
    int l,r,fa,dep; //定义主席树的左右节点,当前节点所在集合以及深度(只和合并有关)
}tr[maxn*32];

inline void build(int &p,int l,int r){
    p=++cnt; //动态开点
    if(l==r){
        tr[p].fa=l; //只需要初始化父亲
        return;
    }
    int mid=(l+r)>>1;
    build(tr[p].l,l,mid);
    build(tr[p].r,mid+1,r);
}

inline void merge(int x,int &y,int l,int r,int pos,int fa){ //合并两区间
    y=++cnt; //动态开点
    tr[y].l=tr[x].l;
    tr[y].r=tr[x].r;
    if(l==r){  //属于单点修改
        tr[y].fa=fa;
        tr[y].dep=tr[x].dep;
        return;
    }
    int mid=l+r>>1;
    if(pos<=mid) merge(tr[x].l,tr[y].l,l,mid,pos,fa);
    else merge(tr[x].r,tr[y].r,mid+1,r,pos,fa);
}

inline void update(int p,int l,int r,int pos){
    if(l==r){ //单点修改秩
        tr[p].dep++;
        return;
    }
    int mid=l+r>>1;
    if(pos<=mid) update(tr[p].l,l,mid,pos);
    else update(tr[p].r,mid+1,r,pos);
}

inline int query(int p,int l,int r,int pos){ //询问pos所在的树上节点
    if(l==r) return p;
```

```
    int mid=l+r>>1;
    if(pos<=mid) return query(tr[p].l,l,mid,pos);
    else return query(tr[p].r,mid+1,r,pos);
}

inline int find(int p,int pos){
    int now=query(p,1,n,pos);
    if(tr[now].fa==pos) return now; //向上查找父亲
    return find(p,tr[now].fa);
}

signed main(){
    n=read();m=read();
    build(rt[0],1,n);
    for(int i=1;i<=m;i++){
        op=read();
        if(op==1){
            a=read();b=read();
            rt[i]=rt[i-1]; //换根
            u=find(rt[i],a),v=find(rt[i],b);
            if(tr[u].fa!=tr[v].fa){ //按秩合并
                if(tr[u].dep>tr[v].dep) swap(u,v);
                merge(rt[i-1],rt[i],1,n,tr[u].fa,tr[v].fa);
                if(tr[u].dep==tr[v].dep) update(rt[i],1,n,tr[v].fa);
            }
        }else if(op==2){
            k=read();
            rt[i]=rt[k];
        }else{
            a=read();b=read();
            rt[i]=rt[i-1];
            if(tr[find(rt[i],a)].fa==tr[find(rt[i],b)].fa) puts("1");
            else puts("0");
        }
    }
    return 0;
}
```

可持久化Trie

对于一棵树上的最大异或和，可以用0/1Trie来解决，但是如果每次询问都是给定区间的话，不能对每个区间都建Trie，这时候就可以用到可持久化Trie了。我们很容易知道，异或是满足可见性的，所以可以用主席树的思想来维护前缀和，两棵树做差即可获得区间的最大异或值。

luoguP4735 最大异或和

给定一个非负整数序列 $\{a\}$，初始长度n。

有 m 个操作，有以下两种操作类型：

1. `A x`：添加操作，表示在序列末尾添加一个数 x，序列的长度 $n+1$。
2. `Q l r x`：询问操作，你需要找到一个位置 p，满足$l \leq p \leq r$，使得：$a[p] \oplus a[p+1] \oplus \ldots \oplus a[N] \oplus x$ 最大，输出最大是多少。

```
#include<bits/stdc++.h>
```

```cpp
using namespace std;
const int N=6e5+10,M=(N<<5);

int n,m,s[N];
int tr[M][2],max_id[M]; //max_id表示当前新加的节点在前缀和数组s的位置
int rt[N],idx=0;

inline void insert(int p,int lst,int k){ //Trie插入操作
    max_id[p]=k;
    for(int i=25;i>=0;--i){
        int v=s[k]>>i&1;
        if(lst) tr[p][v^1]=tr[lst][v^1];
        tr[p][v]=++idx;
        max_id[tr[p][v]]=k;
        p=tr[p][v];lst=tr[lst][v];
    }
}

int query(int l,int r,int C){  //查询[l,r]区间中，与C异或值最大的数
    int p=rt[r];
    for(int i=25;i>=0;--i){ // C是s[n]^x, 从高位到低位逐位检索二进制每一位上能跟C异或结果最大的数
        int v=C>>i&1;
        if(max_id[tr[p][v^1]]>=l) p=tr[p][v^1];
        else p=tr[p][v];
    }
    return C^s[max_id[p]];
}

int main(){
    scanf("%d%d",&n,&m); // 前缀和，初始化第0个版本
    s[0]=0;
    max_id[0]=-1;
    rt[0]=++idx;
    insert(rt[0],0,0);//一开始先插入一个0
    for(int i=1;i<=n;++i){
        int x;
        scanf("%d",&x);
        s[i]=s[i-1]^x;
        rt[i]=++idx;
        insert(rt[i],rt[i-1],i);
    }
    char op[2];
    int l,r,x;
    for(int i=1;i<=m;++i){
        scanf("%s",op);
        if(op[0]=='A'){
            scanf("%d",&x);
            ++n;
            s[n]=s[n-1]^x;
            rt[n]=++idx;
            insert(rt[n],rt[n-1],n);
        }else{
            scanf("%d%d%d",&l,&r,&x);
            printf("%d\n", query(l-1,r-1,s[n]^x));
        }
    }
    return 0;
```

```
}
```

可持久化线段树

在每一个位置维护一个线段树，离散化后节点可以表示值的范围。然后利用前缀和的思想去将两个版本的线段树做差即可得到[l,r]序列的权值线段树。

P3834 【模板】可持久化线段树 2

经典的静态区间第k小问题

```
#include<bits/stdc++.h>
using namespace std;
const int N=2e5+7;
const int mod=1e9+7;
int read(){ int x=0,f=1;char ch=getchar();while(ch<'0'||ch>'9'){if(ch=='-')
f=f*-1;ch=getchar();}while(ch>='0'&&ch<='9'){x=x*10+ch-'0';ch=getchar();}return
x*f;}

int n,m,cnt=0,rt[N],a[N]; //a[i]为原始序列,rt[i]为第i版本主席树

struct node{
    int l,r,sum;  //树的左右端点和元素个数
}tr[N<<5];

vector<int>vt;  //辅助容器

int getid(int x){ //返回每个数的rank
    return lower_bound(vt.begin(),vt.end(),x)-vt.begin()+1;
}

void update(int &x,int y,int l,int r,int pos){
    tr[++cnt]=tr[y];
    tr[cnt].sum++;  //区域元素+1
    x=cnt; //将原来的树地址指向当前树
    if(l==r) return; //叶子结点返回
    int mid=((l+r)>>1);
    if(pos<=mid) update(tr[x].l,tr[y].l,l,mid,pos); //左子树插入
    else update(tr[x].r,tr[y].r,mid+1,r,pos); //右子树插入
}

int query(int x,int y,int l,int r,int k){ //查询[l,r]区间第k大
    if(l==r) return l;
    int mid=((l+r)>>1);
    int sum=tr[tr[y].l].sum-tr[tr[x].l].sum; //左子树相减，其差值为两版本左边相差多少个
数
    if(k<=sum) return query(tr[x].l,tr[y].l,l,mid,k); //结果大于等于k，询问左子树
    else return query(tr[x].r,tr[y].r,mid+1,r,k-sum); //否则找右子树第k-sum小的数
}

signed main(){
    n=read();m=read();
    for(int i=1;i<=n;i++){
        a[i]=read();
```

```
        vt.push_back(a[i]); //辅助容器用于离散化
    }
    sort(vt.begin(),vt.end()); //排序
    vt.erase(unique(vt.begin(),vt.end()),vt.end()); //去重
    for(int i=1;i<=n;i++){ //每次插入都从根节点开始建一棵新树
        update(rt[i],rt[i-1],1,n,getid(a[i]));
    }
    for(int i=1,x,y,k;i<=m;i++){ //第y和x-1棵树做差，就能查出[x,y]区间第k小
        x=read();y=read();k=read();
        printf("%d\n",vt[query(rt[x-1],rt[y],1,n,k)-1]);
    }
    return 0;
}
```

可持久化平衡树

平衡树的可持久化一般可以用FHQ Treap实现。

无旋Treap可通过**Merge**和**Split**操作复制路径上的结点（一般在**Split**操作中复制，确保不会影响以前的版本）就可以完成可持久化。

旋转Treap在复制路径上经过的结点同时，还需复制受选择影响的结点（每次旋转只影响两个结点），不过不会影响时间复杂度。

这种方法称之为**path coping**。

P5055 【模板】可持久化文艺平衡树

题目大意：支持①单点插入；②单点删除；③翻转区间；④区间求和操作并且强制在线

```
#include<bits/stdc++.h>
#define inf 0x3f3f3f3f
#define ll long long
#define int long long
using namespace std;
const int N=2e5+3;
const int mod=1e9+7;

int read(){ int x=0,f=1;char ch=getchar();while(ch<'0'||ch>'9'){if(ch=='-')
f=f*-1;ch=getchar();}while(ch>='0'&&ch<='9'){x=x*10+ch-'0';ch=getchar();}return
x*f;}

struct FHQ{
    int ls,rs,sz,pri;
    ll val,sum; //左儿子，右儿子，树大小，权值、区间和
    bool lzy; //两个标记，翻转区间和区间赋值
}T[(N<<7)];

int idx,rt[N];

inline int new_node(ll v=0){  //新建结点
    static int tot(0);
    //T[++tot].ls=T[tot].rs=T[tot].lzy=0;
    T[++tot].sz=1;T[tot].pri=rand();
```

```
    T[tot].val=T[tot].sum=v;
    return tot; //返回节点编号
}

inline int copy_node(int p){
    int res=new_node();
    T[res]=T[p];
    return res;
}

inline void pushup(int p){ //结点更新信息
    T[p].sz=T[T[p].ls].sz+T[T[p].rs].sz+1;  //子树大小
    T[p].sum=T[T[p].ls].sum+T[T[p].rs].sum+T[p].val; //区间和
}

inline void pushdown(int p){ //下传标记
    if(!T[p].lzy) return;
    if(T[p].ls) T[p].ls=copy_node(T[p].ls);
    if(T[p].rs) T[p].rs=copy_node(T[p].rs);
    swap(T[p].ls,T[p].rs);
    if(T[p].ls) T[T[p].ls].lzy^=1;
    if(T[p].rs) T[T[p].rs].lzy^=1;
    T[p].lzy=0;
}

inline void split(int p,int k,int &x,int &y){ //分离操作
    if(!p){x=y=0;return;}
    pushdown(p);
    if(T[T[p].ls].sz+1<=k){
        x=copy_node(p);
        split(T[x].rs,k-T[T[p].ls].sz-1,T[x].rs,y);
        pushup(x);
    }else{
        y=copy_node(p);
        split(T[y].ls,k,x,T[y].ls);
        pushup(y);
    }
}

inline int merge(int x,int y){ //合并操作
    if(!x||!y) return x|y;
    pushdown(x),pushdown(y);
    if(T[x].pri<T[y].pri){
        T[x].rs=merge(T[x].rs,y);pushup(x);return x;
    }else{
        T[y].ls=merge(x,T[y].ls);pushup(y);return y;
    }
}

signed main(){
    srand(time(0));
    ios::sync_with_stdio(0);
    cin.tie(0);cout.tie(0);
    int cnt=0;
    int m,v,op,pos,a,b,x,y,z;
    ll lastans=0,val;
    cin>>m;
    for(int i=1;i<=m;i++){
```

```
        cin>>v>>op;
        if(op==1){ //区间插入操作
            cin>>a>>b;
            a^=lastans,b^=lastans;
            split(rt[v],a,x,y); //以pos位置拆开再进行三棵树的合并
            rt[++cnt]=merge(merge(x,new_node(b)),y);
        }else if(op==2){ //删除操作
            cin>>a;
            a^=lastans;
            split(rt[v],a,x,z); //以pos-1位置拆开
            split(x,a-1,x,y);
            rt[++cnt]=merge(x,z); //进行合并
        }else if(op==3){ //翻转区间
            cin>>a>>b;
            a^=lastans,b^=lastans;
            split(rt[v],b,x,z);
            split(x,a-1,x,y);
            T[y].lzy^=1;
            rt[++cnt]=merge(merge(x,y),z);
        }else if(op==4){ //区间求和
            cin>>a>>b;
            a^=lastans,b^=lastans;
            split(rt[v],b,x,z);
            split(x,a-1,x,y);
            printf("%lld\n",lastans=T[y].sum);
            rt[++cnt]=merge(merge(x,y),z);
        }
    }
    return 0;
}
```

可持久化左偏树

[SDOI2010] 魔法猪学院(k短路)

求的是在路径权值之和不超过 E 的前提下，选择最多的不同路径，求可选择的最多的路径数。

```
#include <iostream>
#include <cstdio>
#include <cstring>
#include <queue>
#include <algorithm>

#define ll long long

using namespace std;

inline int read()
{
    int s=0,f=1;char ch=getchar();
    while(ch<'0'||ch>'9'){if(ch=='-')f=-f;ch=getchar();}
    while(ch>='0'&&ch<='9'){s=s*10+ch-'0';ch=getchar();}
    return s*f;
}
```

```
const int N=5e4+5,M=2e5+3;

struct edge{int net,to;double w;};
struct Graph
{
    int tot,head[N];edge Edge[M];
    void add(int x,int y,double z){Edge[++tot]=(edge){head[x],y,z};head[x]=tot;}
}G,R;

int n,m,ans=0;double E;

struct node
{
    int id;double dis;
    friend bool operator<(node x,node y){return x.dis>y.dis;}
};
int vis[N],fa[N];double dis[N];
void dijkstra()//求其他点到达n的最短路，并建立最短路树
{
    priority_queue<node>q;q.push((node){n,0});
    memset(dis,127,sizeof(dis));dis[n]=0;
    while(!q.empty())
    {
        node u=q.top();q.pop();
        if(vis[u.id])continue;
        vis[u.id]=1;
        for(int i=R.head[u.id];i;i=R.Edge[i].net)
        {
            node v=(node){R.Edge[i].to,u.dis+R.Edge[i].w};
            if(dis[v.id]>v.dis)
            {
                dis[v.id]=v.dis;
                fa[v.id]=i;//记录最短路树上指向点v.id的边的编号
                q.push(v);
            }
        }
    }
}

int seq[N],rt[N];
bool cmp(const int x,const int y){return dis[x]<dis[y];}

//可持久化左偏树
struct heap{int l,r,dist,fa;double key;}tree[21*N];
int cnt=0;
int make_new(int f,double val)
{
    int k=++cnt;
    tree[k].l=tree[k].r=tree[k].dist=0;
    tree[k].fa=f;tree[k].key=val;
    return k;
}
int merge(int x,int y)
{
    if(!x||!y)return x+y;
    if(tree[x].key-tree[y].key>0)swap(x,y);
    int k=++cnt;
```

```
    tree[k]=tree[x];
    tree[k].r=merge(tree[k].r,y);
    if(tree[tree[k].l].dist<tree[tree[k].r].dist)swap(tree[k].l,tree[k].r);
    tree[k].dist=tree[tree[k].r].dist+1;
    return k;
}

struct ing
{
    int x;double ans;
    friend bool operator<(ing x,ing y){return x.ans>y.ans;}
};

int main()
{
    n=read();m=read();scanf("%lf",&E);
    for(int i=1;i<=m;i++)
    {
        int x=read(),y=read();double z;scanf("%lf",&z);
        if(x==n){i--;m--;continue;}//去掉以n为起点的边
        G.add(x,y,z);R.add(y,x,z);
    }

    dijkstra();

    for(int i=1;i<=n;i++)seq[i]=i;
    sort(seq+1,seq+1+n,cmp);

    //建立可持久化左偏树来维护与每个点及其最短路树上的祖先相连的非树边
    tree[0].dist=-1;
    for(int i=1;i<=n;i++)
    {
        int u=seq[i];
        for(int j=G.head[u];j;j=G.Edge[j].net)

if(fa[u]!=j)rt[u]=merge(rt[u],make_new(G.Edge[j].to,G.Edge[j].w+dis[G.Edge[j].to
]-dis[u]));
        rt[u]=merge(rt[u],rt[G.Edge[fa[u]].to]);
    }

    priority_queue<ing>q;
    if(E-dis[1]<0){printf("0\n");return 0;}
    E-=dis[1];ans++;
    if(rt[1])q.push((ing){rt[1],tree[rt[1]].key});
    while(!q.empty())
    {
        ing u=q.top();q.pop();
        if(E-(u.ans+dis[1])<0)break;
        E-=u.ans+dis[1];ans++;
        if(tree[u.x].l)q.push((ing){tree[u.x].l,u.ans-
tree[u.x].key+tree[tree[u.x].l].key});
        if(tree[u.x].r)q.push((ing){tree[u.x].r,u.ans-
tree[u.x].key+tree[tree[u.x].r].key});
        //对最后一条非树边进行替换
        if(rt[tree[u.x].fa])q.push((ing)
{rt[tree[u.x].fa],u.ans+tree[rt[tree[u.x].fa]].key});
        //新添一条非树边
    }
```

```
    printf("%d\n",ans);
    return 0;
}
```

参考资料

https://oi-wiki.org/ds/persistent-balanced/

https://oi-wiki.org/ds/persistent-trie/

https://blog.csdn.net/qq_52678569/article/details/124210596