有个问题

我们之前所学的数据结构维护修改后的最新版本。

如果想知道在M次修改中，任意第i次修改出的版本。

暴力方法　　每次操作结束后：**拷贝一份**当前状态

BUT　　空间开支：**M倍**　　时间开支：**MSIZE倍**

(拷贝SIZE大小的结构)

有个问题

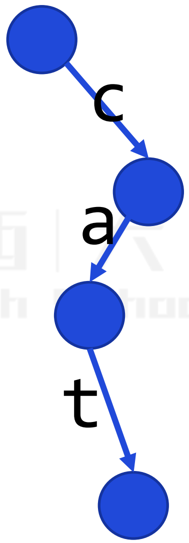
如果想知道在M次修改中，任意第i次修改出的版本。

大量重复数据！

需要啥记录啥！

只记录修改的位置

Eg 字典树插入cat、rat、cab、fry

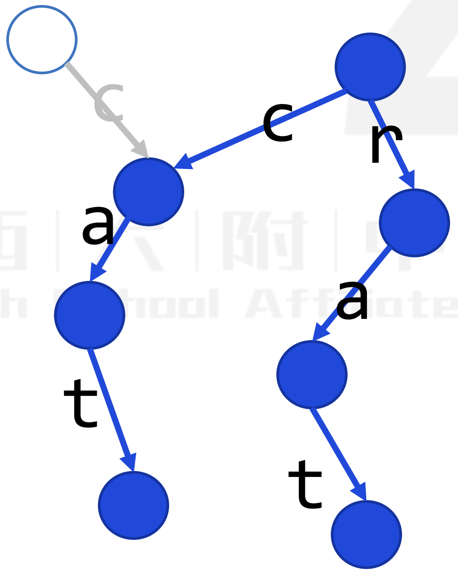


有个问题

如果想知道在M次修改中，任意第i次修改出的版本。

只记录修改的位置

Eg 字典树插入cat、rat、cab、fry

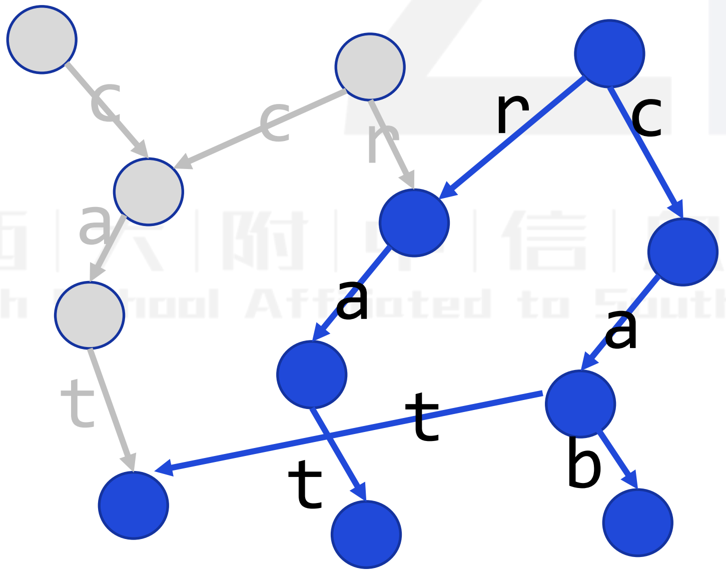


有个问题

如果想知道在M次修改中，任意第i次修改出的版本。

只记录修改的位置

Eg 字典树插入cat、rat、cab、fry

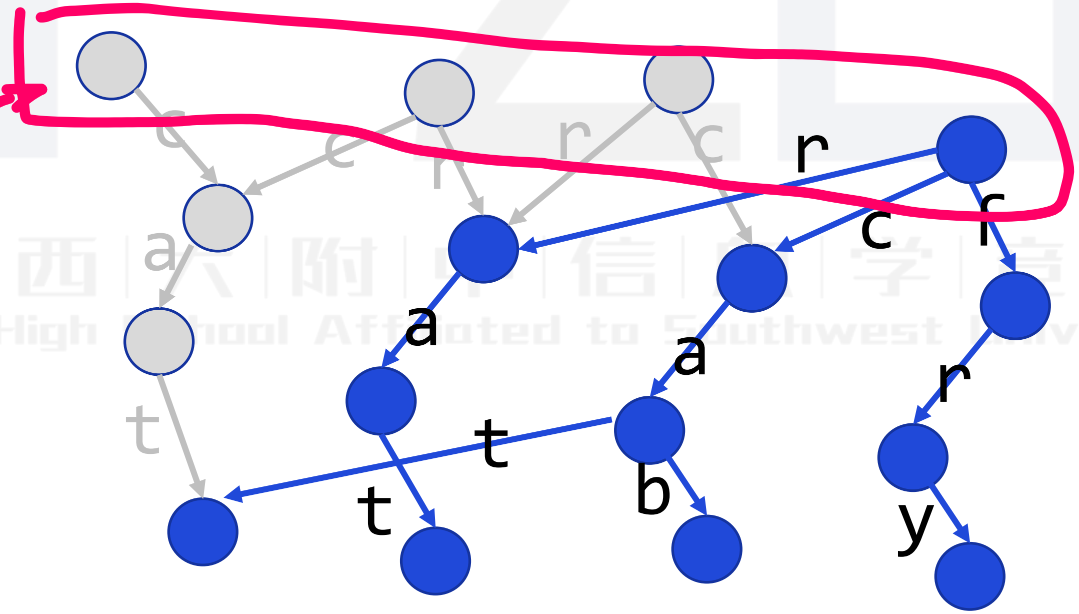


有个问题

如果想知道在M次修改中，任意第i次修改出的版本。

只记录修改的位置

Eg 字典树插入cat、rat、cab、fry



实际上，我们只需要
从这4个节点以此出发
就是4次插入操作的结果

有个问题

我们之前所学的数据结构维护修改后的最新版本。

如果想知道在M次修改中，任意第i次修改出的版本。

只记录修改的位置

将数据结构的所有历史版本记录下来，称为可持久化。

有个问题：可持久化

对于数据结构，维护其所有历史版本

- 部分可持久化：历史版本可访问，但不能修改
- 完全可持久化：支持将当前版本替换为历史版本

信息学

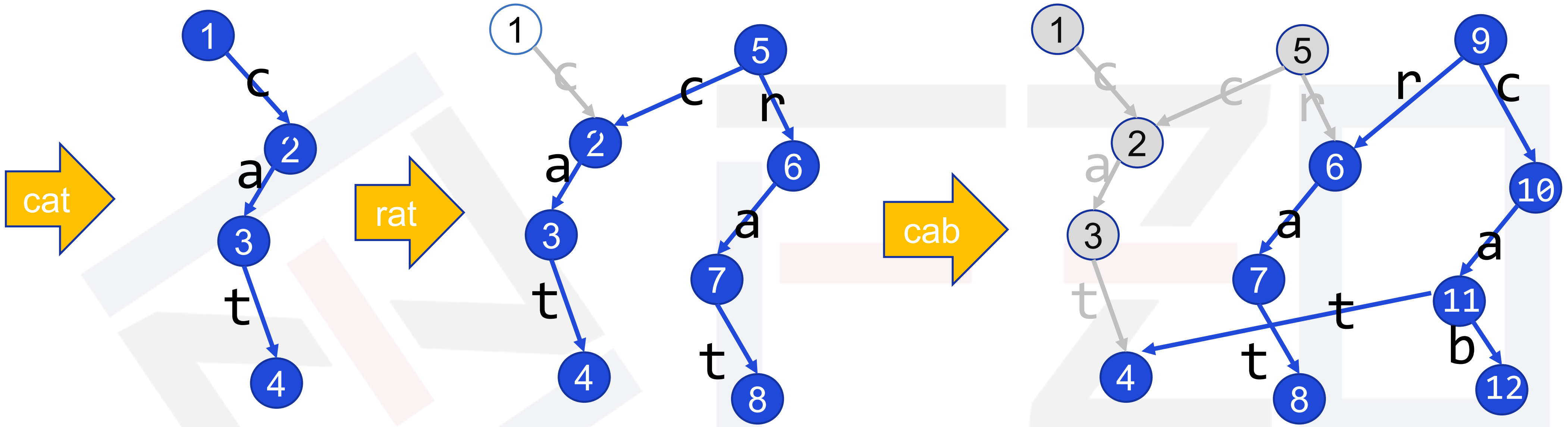
可持久化数据结构

可持久化字典树用途

1.可持久化支持任意版本查询

2.利用可持久化0/1Trie树按位xor （求范围内的异或最大值）

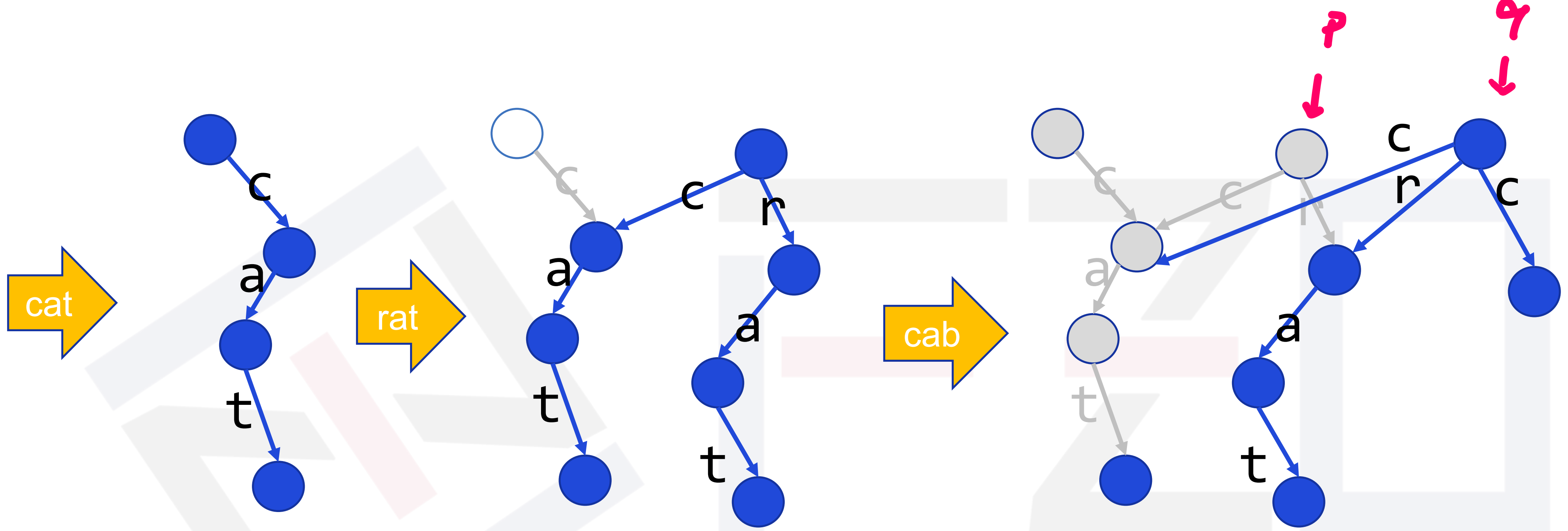
可持久化字典树-建树



回顾建树过程，每次操作后新开一条链

设trie[x, ch]，表示从x号节点连向的字符为ch的点的编号
root[i]表示第i次插入的字符串的根节点，tot代表总节点数

可持久化字典树-建树



回顾建树过程，每次操作后新开一条链

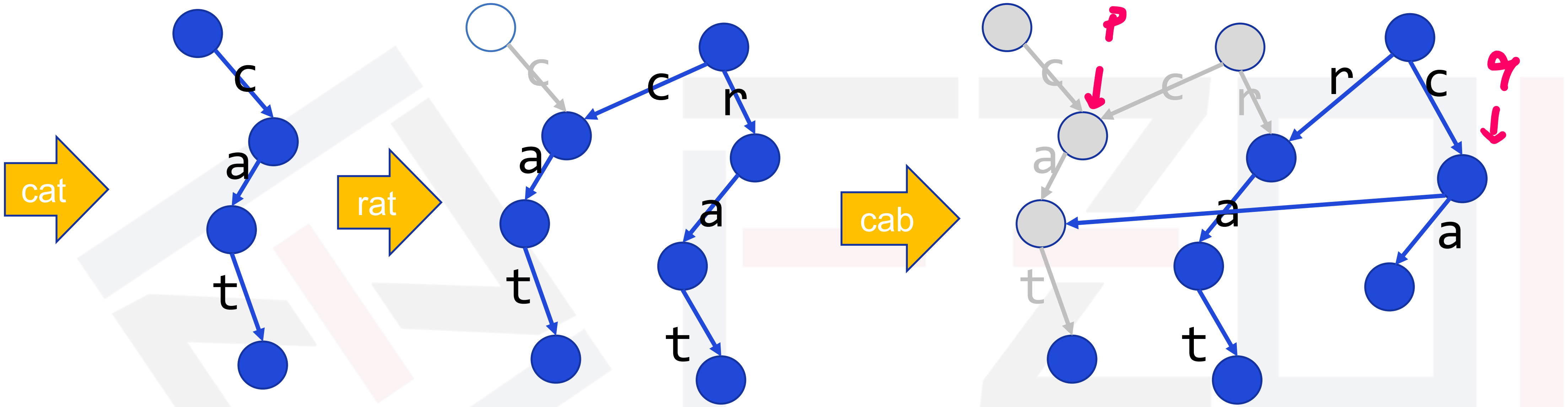
具体做法：还是双指针qp 边扫边维护。

p表示上一个版本根节点

q表示当前版本根节点

如上图所示

可持久化字典树-建树



回顾建树过程，每次操作后新开一条链

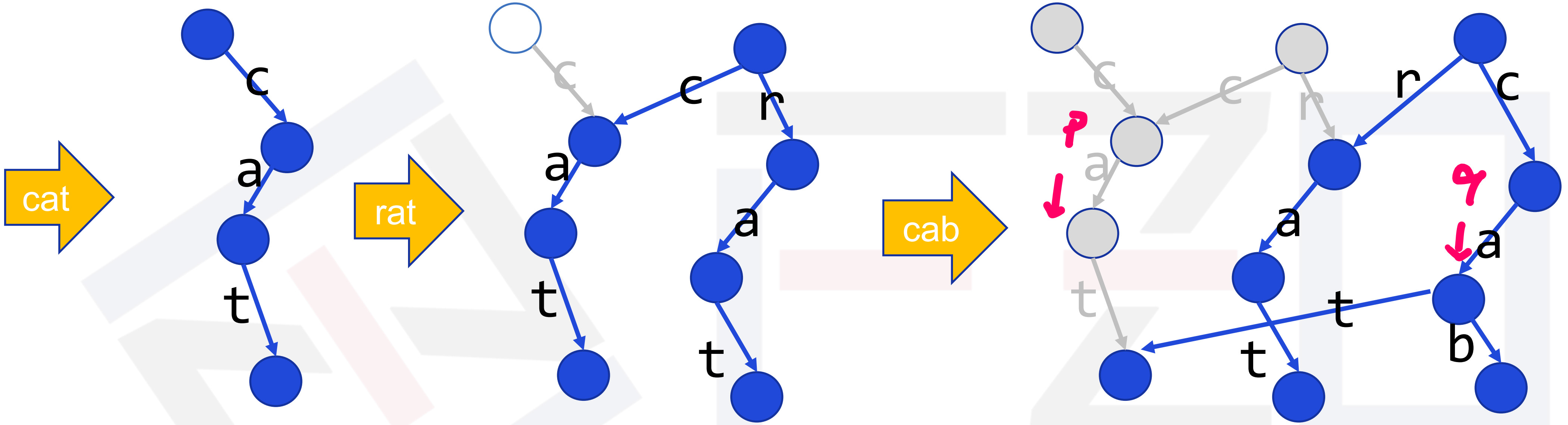
具体做法：还是双指针qp 边扫边维护。

p表示上一个版本根节点

q表示当前版本根节点

如上图所示

可持久化字典树-建树



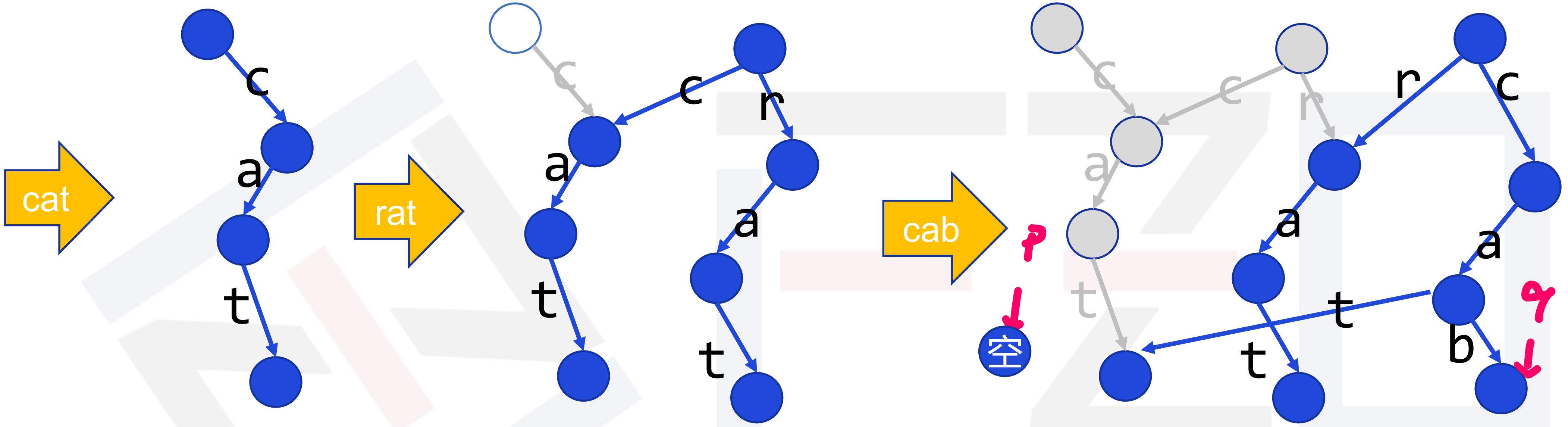
回顾建树过程，每次操作后新开一条链

具体做法：还是双指针qp 边扫边维护。

p表示上一个版本根节点

q表示当前版本根节点

如上图所示



回顾建树过程，每次操作后新开一条链

具体做法：还是双指针qp 边扫边维护。

p表示上一个版本根节点

q表示当前版本根节点

如上图所示⬆

可持久化字典树-建树-伪代码

设trie[x, ch]，表示从x号节点连向的字符为ch的点编号
root[i]表示第i次插入的字符串的根节点，tot代表总节点数

假设当前Tire的根节点为root[tot],新插入字符串s[]

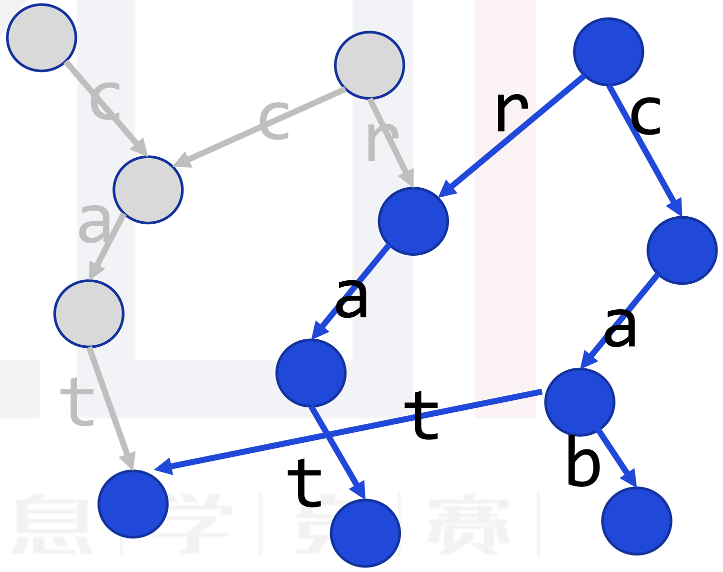
1. p=root[tot],i=0
2. 建立新root q，root[++tot]=q
3. 若p!=0 (root==0表示空)

   for c in 字符集： trie[q,c]=trie[p,c]

   新建节点（编号为tq） trie[q,s[i]]=tq

   p=trie[p,s[i]]，q=trie[q,s[i]],i++

6. 重复3，直到字符串加入完毕



如果 q==0 直接建一条链

trie[q,c]
表示从q号节点出发
连向字符ch的 点编号

实现了相同节点拷贝
不同结点连接的效果

例：最大XOR和

给定一个非负整数序列 a，初始长度为 N。有 M 个操作，每个操作为以下两种类型之一：

1. “A x”，添加操作，表示在序列末尾插入一个数 x，序列的长度 N 增大 1。

2. “Q l r x”，询问操作，求一个位置 p，满足 $l \le p \le r$，使得 $a[p] \operatorname{xor} a[p+1] \operatorname{xor} \cdots \operatorname{xor} a[N] \operatorname{xor} x$ 最大，输出这个最大值。

数据范围：$N, M \le 3 * 10^5$，$0 \le a[i] \le 10^7$。

例：最大XOR和-分析

给定一个非负整数序列 a，初始长度为 N。有 M 个操作，每个操作为以下两种类型之一：

1. “A x”，添加操作，表示在序列末尾插入一个数 x，序列的长度 N 增大 1。

2. “Q l r x”，询问操作，求一个位置 p，满足 $l \le p \le r$，使得 $a[p]$ xor $a[p+1]$ xor ⋯ xor $a[N]$ xor x 最大，输出这个最大值。

数据范围：$N, M \le 3 * 10^5$，$0 \le a[i] \le 10^7$。

因为是求连续区间xor最大值

回想之前知识，记 s[i]=a[1]^a[2]^…^a[i] 有

a[p]^a[p+1]^…^a[N]^x = s[p-1]^s[N]^x = s[p-1]^val

设val=s[N]^x

例：最大XOR和-分析

S数组支持O(1)维护添加数据操作。
对于找s[p-1]^val最大值。
回忆之前01-trie的时候，
结合xor特性，如果不考虑范围，
将val拆成01二进制，再去trie中找相反位置

例如：如果 val = $(1001110)_2$

那么你的tire就要尽力找到这样的路径：$(0110001)_2$
优先保障高位xor结果为1
在不考虑范围的情况下，s[p-1]=$(0110001)_2$

例：最大XOR和-分析

如果考虑范围?

l-1 <= p <= r-1

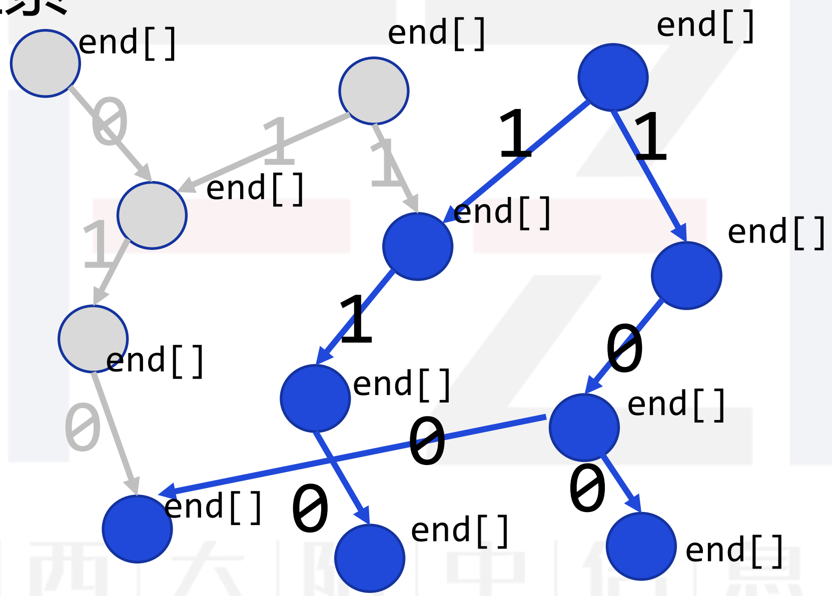
<=r-1 直接从root[r-1]出发检索

>=l-1 添加标记维护插入时间

在维护trie的时候

多增加2个信息

end与latest



latest[x]
表示以x为根的
树中end的最大值

end用于表示当前节点是序列s中第几个
二进制数的末尾 (不是节点用-1标记)

例：最大XOR和-代码

鉴于有些同学连书都没有。。。。
还是放一下代码。

```
const int N = 600010;
int trie[N*24][2], latest[N*24]; // latest 和 end 可合并为一个数组
int s[N], root[N], n, m, tot;
// 本题需要统计子树 latest，故使用递归插入 s[i]，当前为 s[i]的第 k 位
void insert(int i, int k, int p, int q) {
    if (k < 0) {
        latest[q] = i;
        return;
    }
    int c = s[i] >> k & 1;
    if (p) trie[q][c ^ 1] = trie[p][c ^ 1];
    trie[q][c] = ++tot;
    insert(i, k - 1, trie[p][c], trie[q][c]);
    latest[q] = max(latest[trie[q][0]], latest[trie[q][1]]);
}
```

```
int ask(int now, int val, int k, int limit) {
    if (k < 0) return s[latest[now]] ^ val;
    int c = val >> k & 1;
    if (latest[trie[now][c ^ 1]] >= limit)
        return ask(trie[now][c ^ 1], val, k - 1, limit);
    else
        return ask(trie[now][c], val, k - 1, limit);
}

int main() {
    cin >> n >> m;
    latest[0] = -1;
    root[0] = ++tot;
    insert(0, 23, 0, root[0]);
    for (int i = 1; i <= n; i++) {
        int x; scanf("%d", &x);
        s[i] = s[i - 1] ^ x;
        root[i] = ++tot;
        insert(i, 23, root[i - 1], root[i]);
    }
    for (int i = 1; i <= m; i++) {
        char op[2]; scanf("%s", op);
        if (op[0] == 'A') {
            int x; scanf("%d", &x);
            root[++n] = ++tot;
            s[n] = s[n - 1] ^ x;
            insert(n, 23, root[n - 1], root[n]);
        }
        else {
            int l, r, x; scanf("%d%d%d", &l, &r, &x);
            printf("%d\n", ask(root[r - 1], x ^ s[n], 23, l - 1));
        }
    }
}
```

总结 可持久化trie

1 其实主要是在解决01trie的可持久化问题。

2 按位贪心也行（感兴趣同学可以百度一下）

3 对于每一个root节点，从其出发，就是一颗树。

4 对于整体root结点，整体就是一个有向无环图。

PPT翻到这里
休息一下
喝点水

可持久化线段树

一个问题

给定长度为n的序列，求整体第k大值。
询问有m次。n，m均小于1e5

第一反应：排序，输出

现在，我们来更改一下这个问题：

现在给你一个长度为n的序列，再给定一个区间l和r，求l到r区间中第k大值

暴力：先把l到r之间的数先存入一个数组，再进行排序之后输出

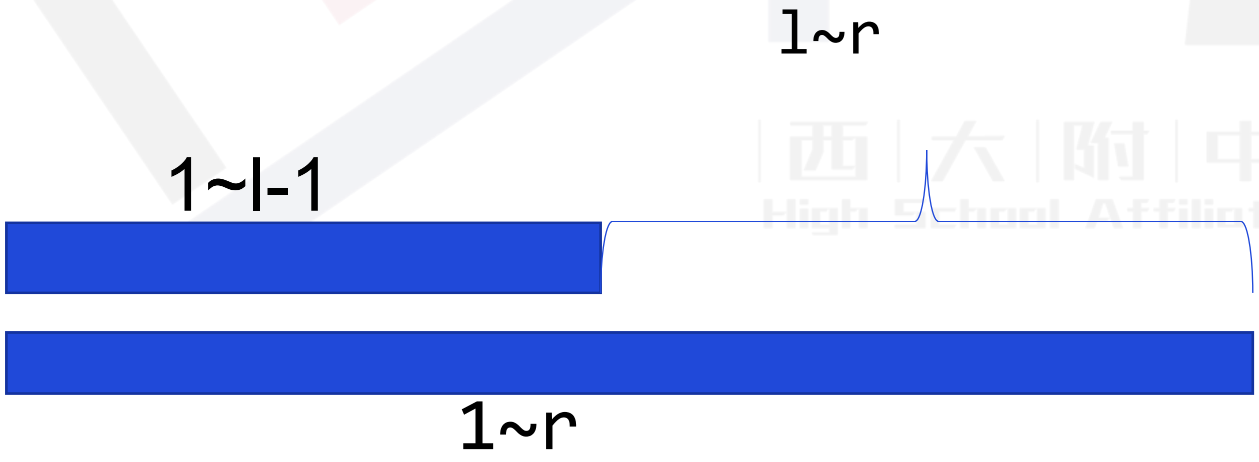
时间复杂度是nmlogn的，我们无法接受
如何解决？

分析

我们之所以时间复杂度如此之大，是因为**每次计算的信息对后面没有贡献。**

求l到r第k大的数的时候，可以用权值线段树维护。

如果在线段树上需要得到[l,r]的统计信息，只需要用[1,r]的信息减去[1,l-1]的信息就是l到r的信息



分析结论

现在，我们已经解决了l到r区间第k大值的问题，

为了解决m个询问，我们需要建立n棵线段树（序列长度为n）

想想，如果真的建立n棵线段树，那空间不爆炸?

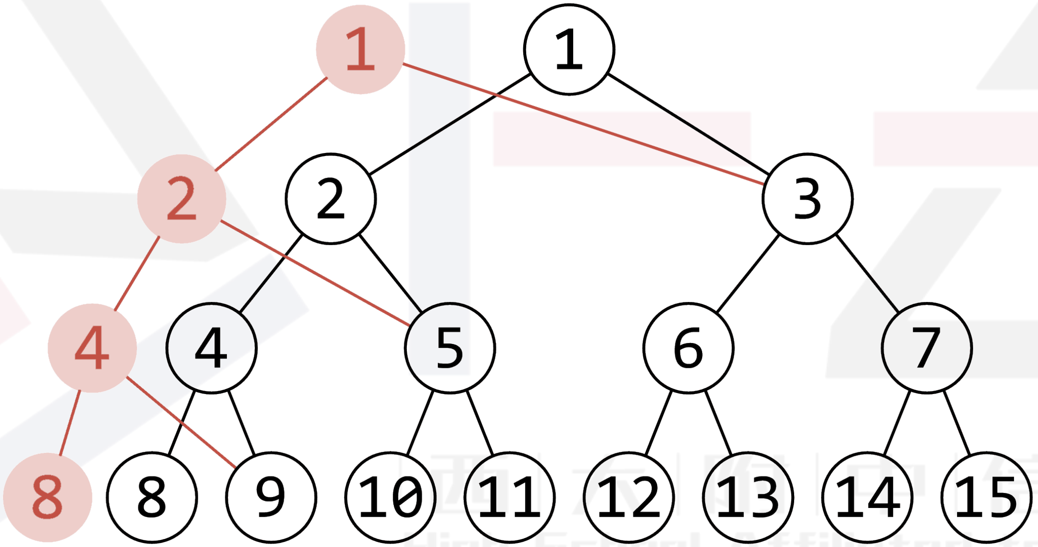
新技能可持久化线段树

什么是可持久化线段树?

主席树

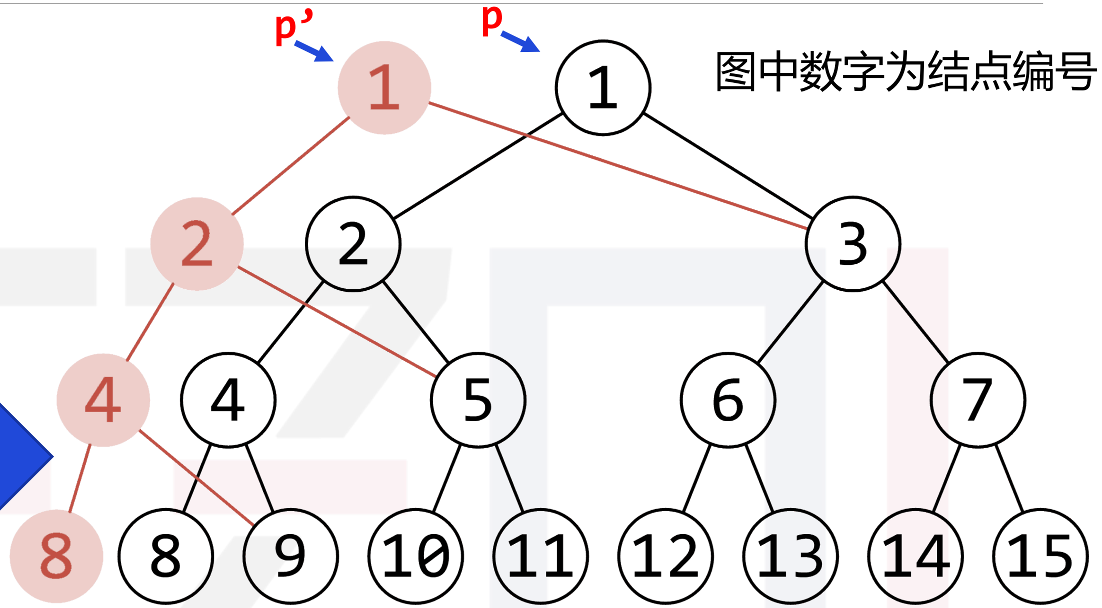
可持久化线段树

修改的时候动态开点（保留历史版本-可持久化）



具体是如何操作的 ?->

可持久化线段树



图中数字为结点编号

规律：每当线段树上的节点被修改时，其树上被改变的节点最多logn+1个

那么：记录改变的节点

假定上图8号节点发生更新,p就从上一个版本的1号结点开始一直往左子树走，p’和p的移动路径相同。途中将p’的值赋值为p，遇到更新的节点就新开一个副本，使其成为p’的左儿子或右儿子。循环下去直至p到达叶子节点

主席树-建树代码

- 具体怎么做的

```
void build(int l,int r,int &rt){
    rt=++tot;
    if(l==r){
        return ;
    }
    int mid=(l+r)/2;
    build(l,mid,t[rt].l);
    build(mid+1,r,t[rt].r);
}
```

主席树-建树代码（细节注意）


```
void build(int l,int r,int &rt){
    rt=++tot;
    if(l==r){
        return ;
    }
    int mid=(l+r)/2;
    build(l,mid,t[rt].l);
    build(mid+1,r,t[rt].r);
}
```

由于可持久化线段树不再是一棵完全二叉树，所以无法像普通的线段树那样直接计算。

于是我们需要将这棵树初始化，相当于在开头时就建立一颗空树，目的是记录下每个节点的左右儿子.

主席树-复杂度分析

首先，我们是动态开点的，所以一棵线段树只会出现 2n-1个结点。

有n次添加，每次至多增加logn+1个结点。因此，最坏情况下n次添加后的结点总数会达到2n-1+(logn+1)n。
此题n<=1e5，单次修改至多增加18个结点，故n次修改后时间和空间大概就是20*n(约等于nlogn)

「POJ2104」K-th Number

给定长度为 $n(1 \le n \le 10^5)$ 的整数序列 $A(|A_i| \le 10^9)$，执行 $m(1 \le m \le 10^4)$ 次操作，其中第 i 次操作给出三个整数 l_i, r_i, k_i，求 $A[l_i], A[l_i+1], \ldots, A[r_i]$ 中第 k_i 小的数是多少。

```
#include<bits/stdc++.h>
using namespace std;
const int N=2e5+10;
int n,m,num,tot;
int a[N],h[N];
struct node{
	int l,r,val;
}t[N<<5];
int root[N];
```

```
void build(int l,int r,int &rt){
	rt=++tot;
	if(l==r){
		return ;
	}
	int mid=(l+r)/2;
	build(l,mid,t[rt].l);
	build(mid+1,r,t[rt].r);
}
```

```
int query(int l,int r,int L,int R,int k){
	if(l==r)return l;
	int sum=t[t[R].l].val-t[t[L].l].val;
	int mid=(l+r)/2;
	if(sum>=k)return query(l,mid,t[L].l,t[R].l,k);
	else return query(mid+1,r,t[L].r,t[R].r,k-sum);
}
void add(int l,int r,int &rt,int p,int k){
	rt=++tot;
	t[rt]=t[p];t[rt].val++;
	if(l==r)return ;
	int mid=(l+r)/2;
	if(k<=mid)add(l,mid,t[rt].l,t[p].l,k);
	else add(mid+1,r,t[rt].r,t[p].r,k);
}
```

「POJ2104」K-th Number


```
void add(int l,int r,int &rt,int p,int k){
    rt=++tot;
    t[rt]=t[p];t[rt].val++;
    if(l==r)return ;
    int mid=(l+r)/2;
    if(k<=mid)add(l,mid,t[rt].l,t[p].l,k);
    else add(mid+1,r,t[rt].r,t[p].r,k);
}
int main(){
    cin>>n>>m;
    for(int i=1;i<=n;i++)scanf("%d",&a[i]),h[i]=a[i];
    sort(h+1,h+n+1);
    num=unique(h+1,h+n+1)-h-1;
    build(1,num,root[0]);
    for(int i=1;i<=n;i++){
        int k=lower_bound(h+1,h+num+1,a[i])-h;
        add(1,num,root[i],root[i-1],k);
    }
    for(int i=1;i<=m;i++){
        int l,r,k;
        scanf("%d%d%d",&l,&r,&k);
        printf("%d\n",h[query(1,num,root[l-1],root[r],k)]);
    }
    return 0;
}
```

一个新问题

我们先前学习的可持久化线段树，都是基于不修改的前提之下的。

现在，我们来在之前的问题上加一个操作：

修改历史版本某一个值

如果我们按照之前的做法来做，修改后，其后面的权值线段树全部都要修改，时间又退化成了平方的级别

那么，我们有什么办法减少修改次数吗？

修改带来的复杂度退化
所以引入树状数组减少修改次数

我们便引入本堂课的第二个知识点：

带修主席树

回顾**树状数组**，这个东西维护前缀和是log级别的。那我们何不借助于这个东西来优化我们的时间呢？

带修改的主席树，又被称为之为树状数组套权值线段树

它和普通主席树的区别在于：

普通主席树是依赖于上一个状态来建立树，而待修改的主席树依赖于树状数组上

带修主席树

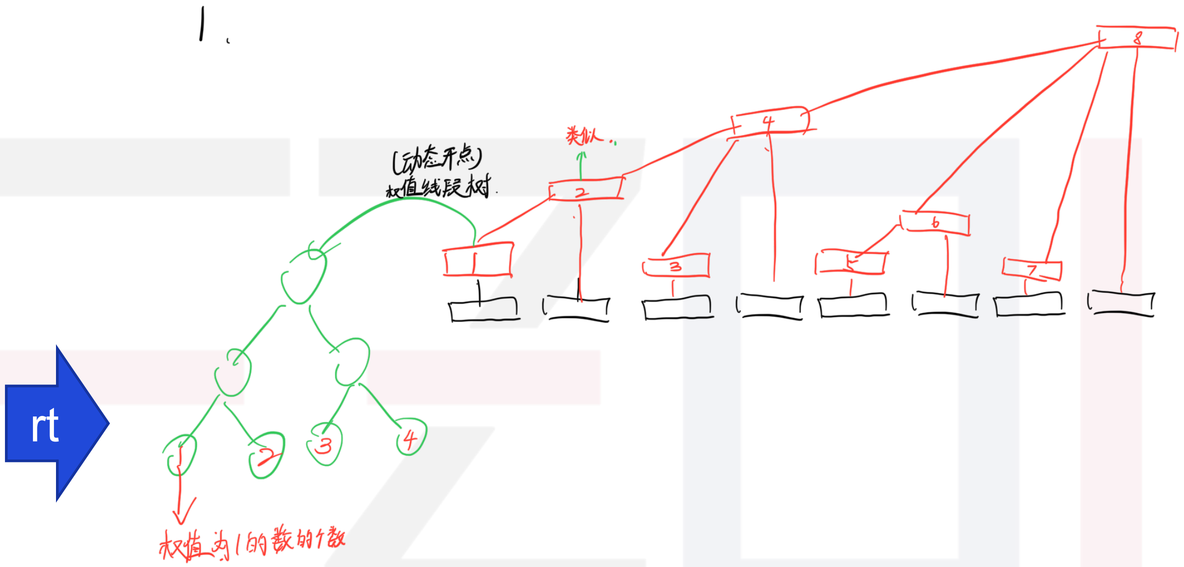
我们对于树状数组上

每一个节点都动态开点

建立一棵权值线段树

每次改变的都只有一条链，而每次修改就是添加一条链上去。

然后其对应的树状数组节点也会修改。单次修改的时间复杂度和空间复杂度均是(log^2)n的



假设树状数组上的2节点修改了，那么2,4,8号节点都要修改

「ZJU2112」 Dynamic Rankings

我们通过一个例题来更好的明白带修主席树：

题目描述

给定一个含有 n 个数的序列 $a_1, a_2 \dots a_n$，需要支持两种操作：

- `Q l r k` 表示查询下标在区间 $[l, r]$ 中的第 k 小的数
- `C x y` 表示将 a_x 改为 y

输入格式

第一行两个正整数 n, m，表示序列长度与操作个数。

第二行 n 个整数，表示 $a_1, a_2 \dots a_n$。

接下来 m 行，每行表示一个操作，都为上述两种中的一个。

输出格式

对于每一次询问，输出一行一个整数表示答案。

「ZJU2112」 Dynamic Rankings

我们先将所有出现过的值离散化（包括修改里面的值）。

对于位置i的修改，其相当于是先在树状数组上从i跳到n。在跳的路径中，每跳到树状数组上的一个节点，其对应节点的线段树也要修改。

对于区间[L,R]的询问，将l-1跳到1的所有用到的树状数组的节点预处理到一个数组L[]里里面，将r跳到1的所有用到的树状数组的节点预处理到R[]数组里。

由于权值线段树具有可加性，所以其所代表的子树相减之后就是L-R所代表的权值线段树

「ZJU2112」 Dynamic Rankings

我们先把查询的节点先预处理出来：

```
for(int i=r;i>0;i-=lowbit(i))t1[++n1]=root[i];
for(int i=l-1;i>0;i-=lowbit(i))t2[++n2]=root[i];
int w=Kth(1,num,kth);
return w;
```

然后直接合成一棵子树，在树上根据k值左右移动：

```
int Kth(int l,int r,int kth){
    if(l==r)return l;
    int sum=0,mid=(l+r)/2;
    for(int i=1;i<=n1;i++)sum+=w[lson[t1[i]]];
    for(int i=1;i<=n2;i++)sum-=w[lson[t2[i]]];
    if(sum>=kth){
        for(int i=1;i<=n1;i++)t1[i]=lson[t1[i]];
        for(int i=1;i<=n2;i++)t2[i]=lson[t2[i]];
        return Kth(l,mid,kth);
    }
    else{
        for(int i=1;i<=n1;i++)t1[i]=rson[t1[i]];
        for(int i=1;i<=n2;i++)t2[i]=rson[t2[i]];
        return Kth(mid+1,r,kth-sum);
    }
}
```

「ZJU2112」 Dynamic Rankings

更新：

```
void Add(int x,int y,int l,int r,int val){
    w[x]+=val;
    if(l==r)return ;
    int mid=(l+r)/2;
    if(y<=mid){
        if(lson[x]==0)tot++,lson[x]=tot;
        Add(lson[x],y,l,mid,val);
    }
    else{
        if(rson[x]==0)tot++,rson[x]=tot;
        Add(rson[x],y,mid+1,r,val);
    }
}
void update(int id,int x,int val){
    for(int i=id;i<=n;i+=lowbit(i)){
        if(root[i]==0)tot++,root[i]=tot;
        Add(root[i],x,1,num,val);
    }
}
```

「ZJU2112」 Dynamic Rankings

整体代码：

```cpp
#include<bits/stdc++.h>
using namespace std;
const int maxn=1e5+7;
int n,m,cnt,num,a[maxn],h[maxn<<1],tot,n1,n2;
struct node{
    char x;
    int l,r,kth;//Q
    int ax,ay;//C
}s[maxn];
int w[maxn<<8],root[maxn<<8],lson[maxn<<8],rson[maxn<<8];
int t1[maxn],t2[maxn];
int lowbit(int x){
    return x&-x;
}
void Add(int x,int y,int l,int r,int val){
    w[x]+=val;
    if(l==r)return ;
    int mid=(l+r)/2;
    if(y<=mid){
        if(lson[x]==0)tot++,lson[x]=tot;
        Add(lson[x],y,l,mid,val);
    }
    else{
        if(rson[x]==0)tot++,rson[x]=tot;
        Add(rson[x],y,mid+1,r,val);
    }
}
void update(int id,int x,int val){
    for(int i=id;i<=n;i+=lowbit(i)){
        if(root[i]==0)tot++,root[i]=tot;
        Add(root[i],x,1,num,val);
    }
}
```

```cpp
int Kth(int l,int r,int kth){
    if(l==r)return l;
    int sum=0,mid=(l+r)/2;
    for(int i=1;i<=n1;i++)sum+=w[lson[t1[i]]];
    for(int i=1;i<=n2;i++)sum-=w[lson[t2[i]]];
    if(sum>=kth){
        for(int i=1;i<=n1;i++)t1[i]=lson[t1[i]];
        for(int i=1;i<=n2;i++)t2[i]=lson[t2[i]];
        return Kth(l,mid,kth);
    }
    else{
        for(int i=1;i<=n1;i++)t1[i]=rson[t1[i]];
        for(int i=1;i<=n2;i++)t2[i]=rson[t2[i]];
        return Kth(mid+1,r,kth-sum);
    }
}
int query(int l,int r,int kth){
    n1=n2=0;
    for(int i=r;i>0;i-=lowbit(i))t1[++n1]=root[i];
    for(int i=l-1;i>0;i-=lowbit(i))t2[++n2]=root[i];
    int w=Kth(1,num,kth);
    return w;
}
```

```cpp
int main(){
    cin>>n>>m;
    for(int i=1;i<=n;i++)scanf("%d",&a[i]),h[++cnt]=a[i];
    for(int i=1;i<=m;i++){
        getchar();
        char p;
        scanf("%c",&p);
        if(p=='C'){
            int x,y;
            scanf("%d%d",&x,&y);
            h[++cnt]=y;
            s[i].x='C',s[i].ax=x,s[i].ay=y;
        }
        else{
            int x,y,z;
            scanf("%d%d%d",&x,&y,&z);
            s[i].x='Q';
            s[i].l=x,s[i].r=y,s[i].kth=z;
        }
    }
    sort(h+1,h+cnt+1);
    num=unique(h+1,h+cnt+1)-h-1;
    for(int i=1;i<=n;i++){
        int x=lower_bound(h+1,h+num+1,a[i])-h;
        update(i,x,1);
    }
    for(int i=1;i<=m;i++){
        if(s[i].x=='C'){
            int id=s[i].ax;
            int x=lower_bound(h+1,h+num+1,a[s[i].ax])-h;
            update(id,x,-1);
            a[s[i].ax]=s[i].ay;
            x=lower_bound(h+1,h+num+1,a[s[i].ax])-h;
            update(id,x,1);
        }
        else{
            printf("%d\n",h[query(s[i].l,s[i].r,s[i].kth)]);
        }
    }
    return 0;
}
```

我们再通过一个例题来了解一下带修主席树：

您需要写一种数据结构（可参考题目标题），来维护一个有序数列，其中需要提供以下操作：

1. 查询 x x在区间内的排名；
2. 查询区间内排名为 k 的值；
3. 修改某一位置上的数值；
4. 查询 x 在区间内的前趋（前趋定义为小于 x，且最大的数）；
5. 查询 x 在区间内的后继（后继定义为大于 x，且最小的数）。

这里的区间是给定区间，其排名是从小到大的

我们来思考一下五大操作：

2，3号操作便是带修主席树的基本操作，这里不再说明

1操作:我们都知道,在可修改的主席树中,树状数组中每一个点都是一颗权值线段树, 把它们全部合并之后,就变成了一棵权值线段树，可以直接在树上查找

2371.二逼平衡树

4操作 -> 我们把x在区间的排名先算出来,借助1操作.我们可以发现,1操作就是求比x小的最大值的位置(且存在) 所以把位置求出来之后直接把位置带进2操作里算就可以了

5操作-> 和4操作是一样的,但是只有一点变动:求的是后继,那么我们搜索的时候就要把本身算上,然后+1就是后继,把位置求出来之后直接把位置带进2操作里算就可以了;

以下是具体代码:

2371.二逼平衡树

```cpp
#include<bits/stdc++.h>
using namespace std;
int n,m,cnt,num,tot,n1,n2; //n,m如题意 cnt:离散化数组指针 num:离散化之后数组长度 tot:主席树动态开点的工具 n1:预处理根(指root) n2:预处理根(指
const int maxn=50005;
int a[maxn<<1];//初始序列
int h[maxn<<2];//离散化数组
struct node{
    int flag;//每一个操作
    int l,r,k;//1,2,4,5操作 l:左边界 r:右边界  k:如题
    int pos,val;//3操作 pos:如题 val:值
}s[maxn+105];
int root[maxn<<8],w[maxn<<8],lson[maxn<<8],rson[maxn<<8];//根数组   权值数组  左儿子  右儿子
int t1[maxn<<3],t2[maxn<<3];//都是预处理数组
int lowbit(int x){
    return x&-x;
}
void Add(int rt,int x,int l,int r,int val){//根节点  目标节点  左边界  右边界 是把这条链+1/-1
    w[rt]+=val;//加上去
    if(l==r)return ;//到边界
    int mid=(l+r)/2;
    if(x<=mid){//如果这个节点在左边
        if(lson[rt]==0)tot++,lson[rt]=tot;//如果左边没有被扩展过 动态开店
        Add(lson[rt],x,l,mid,val);//递归
    }
    else{
        if(rson[rt]==0)tot++,rson[rt]=tot;//如果右边没有被扩展过 动态开店
        Add(rson[rt],x,mid+1,r,val);//递归
    }
}
void update(int id,int val){
    int x=lower_bound(h+1,h+num+1,a[id])-h;//离散化的用处，把这个值对应的编号提出来
    for(int i=id;i<=n;i+=lowbit(i)){//既然是树状数组套主席树，那么基本结构就和树状数组一致
        if(root[i]==0)tot++,root[i]=tot;//如果这个节点是没有被访问过的，即：这个节点的树还没开始建
        Add(root[i],x,1,num,val);//扔进去
    }
}
int Kth(int l,int r,int kth){
    if(l==r)return l;
    int sum=0,mid=(l+r)/2;
    for(int i=1;i<=n1;i++)sum+=w[lson[t1[i]]];
    for(int i=1;i<=n2;i++)sum-=w[lson[t2[i]]];
    if(sum>=kth){
        for(int i=1;i<=n1;i++)t1[i]=lson[t1[i]];
        for(int i=1;i<=n2;i++)t2[i]=lson[t2[i]];
        return Kth(l,mid,kth);
    }
    else{
        for(int i=1;i<=n1;i++)t1[i]=rson[t1[i]];
        for(int i=1;i<=n2;i++)t2[i]=rson[t2[i]];
        return Kth(mid+1,r,kth-sum);
    }
}
```

```cpp
int query(int l,int r,int kth){
    n1=n2=0;
    for(int i=r;i>0;i-=lowbit(i))t1[++n1]=root[i];
    for(int i=l-1;i>0;i-=lowbit(i))t2[++n2]=root[i];
    int czc=Kth(1,num,kth);
    return czc;
}
void Change(int i){
    int id=s[i].pos;
    update(id,-1);
    a[id]=s[i].val;
    update(id,1);
}
int update2(int l,int r,int k){
    if(l==r)return 0;
    int mid=(l+r)/2,sum=0;
    for(int i=1;i<=n1;i++)sum+=w[lson[t1[i]]];//先把权值加起来
    for(int i=1;i<=n2;i++)sum-=w[lson[t2[i]]];
    if(k>mid){//如果是在右边
        for(int i=1;i<=n1;i++)t1[i]=rson[t1[i]];
        for(int i=1;i<=n2;i++)t2[i]=rson[t2[i]];
        return update2(mid+1,r,k)+sum;
    }
    else{//否则在左边
        for(int i=1;i<=n1;i++)t1[i]=lson[t1[i]];
        for(int i=1;i<=n2;i++)t2[i]=lson[t2[i]];
        return update2(l,mid,k);
    }
}
int update1(int l,int r,int kk){
    int k=lower_bound(h+1,h+num+1,kk)-h;
    n1=n2=0;
    for(int i=r;i>0;i-=lowbit(i))t1[++n1]=root[i];//预处理根
    for(int i=l-1;i>0;i-=lowbit(i))t2[++n2]=root[i];//同上
    int czc=update2(1,num,k);//查找
    return czc;
}
int update4(int l,int r,int k){
    int id=update1(l,r,k);//找前驱
    return query(l,r,id);
}
int update7(int l,int r,int k){
    if(l==r)return 0;
    int mid=(l+r)/2,sum=0;
    for(int i=1;i<=n1;i++)sum+=w[lson[t1[i]]];
    for(int i=1;i<=n2;i++)sum-=w[lson[t2[i]]];
    if(k>=mid){
        for(int i=1;i<=n1;i++)t1[i]=rson[t1[i]];
        for(int i=1;i<=n2;i++)t2[i]=rson[t2[i]];
        return update7(mid+1,r,k)+sum;
    }
    else{
        for(int i=1;i<=n1;i++)t1[i]=lson[t1[i]];
        for(int i=1;i<=n2;i++)t2[i]=lson[t2[i]];
        return update7(l,mid,k);
    }
}
```

2371.二逼平衡树


```cpp
int update6(int l,int r,int kk){
    int k=lower_bound(h+1,h+num+1,kk)-h;
    n1=n2=0;
    for(int i=r;i>0;i-=lowbit(i))t1[++n1]=root[i];
    for(int i=l-1;i>0;i-=lowbit(i))t2[++n2]=root[i];
    int czc=update7(1,num,k);
    return czc;
}
int update5(int l,int r,int k){
    int id=update6(l,r,k)+1;
    return query(l,r,id);
}
int main(){
    cin>>n>>m;
    for(int i=1;i<=n;i++)scanf("%d",&a[i]),h[++cnt]=a[i];//存入预处理数组
    for(int i=1;i<=m;i++){
        int flag;
        scanf("%d",&flag);
        if(flag==3){
            int pos,k;
            scanf("%d%d",&pos,&k);
            h[++cnt]=k;
            s[i].flag=flag,s[i].pos=pos,s[i].val=k;//修改的值也要加入
        }
        else{
            int l,r,k;
            scanf("%d%d%d",&l,&r,&k);
            s[i].flag=flag,s[i].l=l,s[i].r=r,s[i].k=k;
            if(flag!=2)h[++cnt]=k;//它有时候查询的值不在里面,为了方便,一起加入
        }
    }
    sort(h+1,h+cnt+1);
    num=unique(h+1,h+cnt+1)-h-1;//去重
    for(int i=1;i<=n;i++)update(i,1);//初始的时候全部加入
    for(int i=1;i<=m;i++){
        int flag=s[i].flag;
        if(flag==1)printf("%d\n",update1(s[i].l,s[i].r,s[i].k)+1);
        if(flag==2)printf("%d\n",h[query(s[i].l,s[i].r,s[i].k)]);
        if(flag==3)Change(i);
        if(flag==4)printf("%d\n",h[update4(s[i].l,s[i].r,s[i].k)]);
        if(flag==5)printf("%d\n",h[update5(s[i].l,s[i].r,s[i].k)]);
    }
    return 0;
}
```

一个新问题

既然我们会了主席树，何不再拓展一点呢？

有一个东西：可持久化并查集，是基于主席树之上实现的。

既然学习了主席树，我们就顺便来学习一下可持久化并查集

可持久化并查集

俗话说，可持久化并查集

=可持久化+并查集

=可持久化数组+并查集

=**主席树+并查集**

并查集有两种优化方式

路径压缩

按秩合并

操作提示

只能按秩合并，不能路径压缩

解释：由于需要我们支持的只有集合的合并与查询。当我们将两个集合合并到一起的时候，无论哪一个集合连接到另一个集合的下面都可以得到正确的结果。

- **但是**这是运用了主席树的并查集。如果我们用了路径压缩，那么就会造成大量修改。而按秩合并只有一次修改，所以我们选择按秩合并。
- **深度问题**由于按秩合并中需要深度的信息，而每个版本的并查集节点深度可能是不一样的，所以我们需要额外开一个数组dep[]来记录。

我们用了两个可持久化数组fa[]和dep[]，分别记录了每一个版本的祖先节点和每个节点的深度。

所谓可持久化并查集，可以进行的操作有以下几个：

1.合并两个集合

2.回到历史版本

3.查询节点所在集合的祖先，当然，因此也可以判断是否在同一个集合中

对于操作2，我们可以很容易的用可持久化数组来实现fa[i]=fa[k],dep[i]=dep[k];

对于操作1，也就是按秩合并；对于操作3，就是在可持久化数组中查询

问题来了：怎么维护fa和dep?

我们可以共用一个主席树，把内存池开大两倍，然后建立两个根节点数组rootfa[]和rootdep[]。

对于rootfa，其叶子节点所表示的含义就是这个点的父亲为谁

对于rootdep，其叶子节点所表示的含义就是这个点的深度

可持久化并查集

西南大学附属中学

```cpp
#include<bits/stdc++.h>
using namespace std;
const int N=2e5+10;
int n,m;
int rootfa[N],rootdep[N];
int tot,idx;
struct node{
    int l,r,val;
}tree[N*80];
void build(int l,int r,int &u){
    u=++idx;
    if(l==r){
        tree[u].val=++tot;
        return ;
    }
    int mid=(l+r)/2;
    build(l,mid,tree[u].l);
    build(mid+1,r,tree[u].r);
}
int query(int l,int r,int ver,int val){
    int mid=(l+r)/2;
    if(l==r){
        return tree[ver].val;
    }
    if(mid>=val)return query(l,mid,tree[ver].l,val);
    else return query(mid+1,r,tree[ver].r,val);

}
int find(int ver,int val){
    int fa=query(1,n,rootfa[ver],val);
    if(val!=fa)val=find(ver,fa);
    return val;
}
void add(int l,int r,int ver,int &u,int x,int y){
    u=++idx;
    tree[u]=tree[ver];
    if(l==r){
        tree[u].val=y;
        return ;
    }
    int mid=(l+r)/2;
    if(x<=mid)add(l,mid,tree[ver].l,tree[u].l,x,y);
    else add(mid+1,r,tree[ver].r,tree[u].r,x,y);
}
void merge(int ver,int x,int y){
    x=find(ver-1,x);
    y=find(ver-1,y);
    if(x==y){
        rootfa[ver]=rootfa[ver-1];
        rootdep[ver]=rootdep[ver-1];
    }
    else{
        int depx=query(1,n,rootdep[ver-1],x);
        int depy=query(1,n,rootdep[ver-1],y);
        if(depx<depy){
            add(1,n,rootfa[ver-1],rootfa[ver],x,y);
            rootdep[ver]=rootdep[ver-1];
        }
        else if(depx>depy){
            add(1,n,rootfa[ver-1],rootfa[ver],y,x);
            rootdep[ver]=rootdep[ver-1];
        }
        else{
            add(1,n,rootfa[ver-1],rootfa[ver],x,y);
            add(1,n,rootdep[ver-1],rootdep[ver],y,depy+1);
        }
    }
}
int main(){
    cin>>n>>m;
    build(1,n,rootfa[0]);
    for(int i=1;i<=m;i++){
        int op,x,y;
        scanf("%d",&op);
        if(op==1){
            scanf("%d%d",&x,&y);
            merge(i,x,y);
        }
        if(op==2){
            scanf("%d",&x);
            rootfa[i]=rootfa[x];
            rootdep[i]=rootdep[x];
        }
        if(op==3){
            scanf("%d%d",&x,&y);
            rootfa[i]=rootfa[i-1];
            rootdep[i]=rootdep[i-1];
            int x1=find(i-1,x);
            int y1=find(i-1,y);
            if(x1==y1)printf("1\n");
            else printf("0\n");
        }
    }
    return 0;
}
```

参考

- 蓝书
- 《算法训练营（海量图解+竞赛刷题）》陈小玉