除了很強，只有更強 - 常見資結技巧 1

1.1 讓資料結構如虎添翼！

前面學過了許多的資料結構，像是線段樹、Treap、Fenwick Tree（只是想要讓你們複習一下 BIT 的另外一個名字）等，雖然可以做一些事，但是還是有許多的不足！可以透過新的概念來幫資料結構們加上裝備，讓他們可以做更多事！這裡的東西會比較難理解，需要慢慢讀文字，仔細思考，一小段可以看一個禮拜，慢慢吸收，慢慢理解。

1.2 懶惰標誌（Lazy Propagation）

就是他！讓我 TLE！

先來看一個經典題，來展現那些資料結構的不足！

習題 1.2.1: 經典題（帶修改區間極值）

給定一個長度為 N 的序列，值分別為 $a_1, a_2, \ldots, a_N$，有 Q 個操作，每一個操作都是以下的其中之一：

1. `1 l r k`，代表在 (l, r) 間的所有值都加 k
2. `2 l r`，請輸出 (l, r) 間所有的值的最大值

（$N \leq 10^5$，$Q \leq 10^4$）

不難想到用一個線段樹或維護，這樣子詢問 $O(Q_1 \log N \cdot N + Q_2 \log N)$，就爆掉了！所以一定需要更強大，更好的方法來

資料結構的第一個武器——懶標

我們做事要懶惰一些，幫電腦省事：如果要加的時候發現這整個區間都需要加，那是不是可以直接記錄下來說這個區間加某個數字 k，以後如果需要查詢這

個區間全部的時候，就直接用數學計算就好了，不需要遞迴下去將每一個點弄好呢？沒錯！就是這樣！

那，具體要怎麼實作呀？對於每一個節點，除了維持值、左右方為何之外、還需要維持一個lazy值，代表這個區間每一個值都要加lazy，如果為 0 則代表目前不需要加。為了實施憲法中的明確性，先來看一下我們所用的node長什麼樣吧！

```cpp
struct Node{
    int l, r, val, lazy; //代表這個節點的左，右界、目前的值、和最新的lazy值！
    Node *left, *right; //指向的子節點
}
```

兩把小刀：push 和 pull

對於每一個node，都應該要有兩個函數，pull應該已經看過了，很簡單：

```cpp
void pull(Node* n){
    n->val = n->left->val + n->right->val;
}
```

但是最新進來的捧油push就比較奇特一點了，因應打懶標而出現，也就是將懶標往下打一層，話不多說，直接看程式！

```cpp
void push(Node *n){
    if(!n->lazy) return; //不需要做事
    n->val += n->lazy; //若整個區間都要加lazy，則最大值也會加lazy
    if(n->l + 1 < n->r){ //如果還有在下面的區間，則打下去，注意沒有遞迴！
        n->left->lazy += n->lazy;
        n->right->lazy += n->lazy;
    }
    n->lazy = 0;
}
```

原本的函數要進化了！

現在來示範怎麼修改原本有的query函數和modify函數：

新的query

簡單來說，就是在詢問之前，都要確定沒有懶標需要push了，再查詢！

```cpp
int query(Node *n, int ql, int qr ){
    push(n); //重要重要重要！
    if(ql >= n->r || qr <= n->l) return -INF; //出界
    if(ql <= n->l && n->r <= qr) return n->val; //完全包含
    return max(query(n->l, ql, qr), query(n->r, ql, qr));
}
```

新的modify

其實與query差不多，先記得push之後，修改懶標即可。

```
void modify(Node *n, int ql, int qr, int val){
    push(n); //重要重要重要！
    if(ql >= n->r || qr <= n->l) return; //出界
    if(ql <= n->l && n->r <= qr){ //完全包含
        n->lazy += val;
        return;
    }
    modify(n->l, ql, qr, val);
    modify(n->r, ql, qr, val);
    pull(n); //別忘了

}
```

這就是你的第一個（或是第 k 個，$k \in \mathbb{N} \cap \{0\}$）懶標線段樹了！重點就是，如果遇到一個節點，可以盡量不要修改就不要修改，只是記錄下來，到了真的需要修改再修改之。

1.3 另外一種寫法

會有人覺得，修改的時候沒有修改到東西不夠爽，所以就出現了另外一種寫的方法，可以供參考：想法就是，lazy代表**子樹**所需要修改的東西，自己則是修改完了！這樣，push會變成

```
void update(Node *n, int val) { //helper function
    n->v += val, n->lazy += val;
}
void push(Node *n){
    if(!n->lazy || n->l+1 >= r) return;
    update(n->left,n->lazy);
    update(n->right,n->lazy);
    n->lazy = 0;
}
```

至於modify，也會變乾淨：

```
void modify(Node *n, int ql, int qr, int val) {
    push(n); //重要重要重要！
    if(ql >= n->r || qr <= n->l) return; //出界
    if(ql <= n->l && n->r <= qr){ //完全包含
        update(n, val);
        return;
    }
    modify(n->l, ql, qr, val);
    modify(n->r, ql, qr, val);
    pull(n); //別忘了
}
```

習題

來練習一下新學到的技巧吧！

習題 1.3.1: Ahoy Pirates!（UVa 11402）

給定一個長度為 N，且由 0 和 1 組成的序列，請支援 Q 個操作，每一個都是以下操作之一：

1. `F a b`：請把第 a 個到第 b 個位置的數字都變成 1
2. `E a b`：請把第 a 個到第 b 個位置的數字都變成 0
3. `I a b`：請把第 a 個到第 b 個位置的數字，0 變成 1，1 變成 0
4. `S a b`：請輸出第 a 個到第 b 個位置的數字總共有幾個 1

（$N \leq 10^6$，$Q \leq 10^5$）

習題 1.3.2: Circular RMQ（CF 52C）

給定一個長度為 N 的環形（也就是最後一項和第一項相鄰）序列 $a_0, a_1, \cdots, a_{N-1}$，和有 Q 筆操作，都是以下的兩個其中之一：

1. $\text{inc}(lf, rg, v)$：將 $[lf, rg]$ 內的數字全部加 v
2. $\text{rmq}(lf, rg)$：請輸出 $[lf, rg]$ 中最小的數字

（$1 \leq N \leq 200000$，$0 \leq Q \leq 200000$，$|v|, |a_i| \leq 10^6$）

習題 1.3.3: 矩形覆蓋面積計算（TIOJ 1224）

很經典的一題：給你平面上 N 個矩形，請求那些矩形所覆蓋出來的面積為何？(如果多個矩形蓋到同一個地方，只能算一次）$N \leq 10^5$，且矩形的 x, y 座標皆為在 0 和 10^6 之間的整數。

習題 1.3.4: 《Φ》序章·IV　生活作息（ZJ c251）

給定一個長度為 N 的序列 S，有 Q 次如下的操作：

1. `0 L R`：輸出 $[L, R]$ 中有幾種不同的數字
2. `1 L R P`：把 $[L, R]$ 中所有數字修改為 P

（$Q, N \leq 2^{15}$，$S_i \leq \min(N, 2^5)$，輸入有至多 2^5 筆測資）

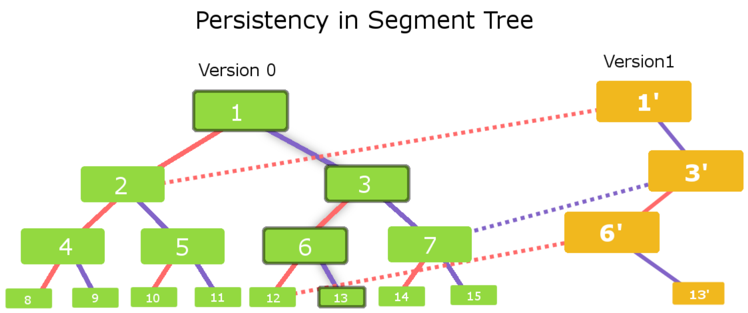
1.4 持久化資料結構（Persistent Data Structure）

會不會 Cmd + Z 啊？

有時候，在修改資料結構的時候，會想要同時保存舊的版本和新的版本，但是如果整個再複製一次，這樣子記憶體會用的很兇（又不是 Google Code Jam 給 10GB !），所以必須用到持久化的概念來將記憶體壓回正常的範圍。

什麼是持久化？

持久化的精神就是：**每次修改之後，只新複製出被動到的地方，其餘都和原本的共用**，如果不清楚，就看圖吧！如果太多搞不清楚了，就看天吧！



這裡，我們要對原本的線段樹（Version 0）做修改，而我們要修改編號為 13 的節點。所以呢，沿路會動到 $1, 3, 6, 13$ 這四個節點，只需要新建立出 $1', 3', 6', 13'$ 這四個節點成為 Version 1，其他的就可以與舊的 Version 0 共用了！這樣，新增的空間複雜度就從原本的 $O(n)$ 變成 $O(\log n)$ 了！

實作方法——來種一棵持久化線段樹吧！

其實重點就是要存那些新的根節點，用一個陣列存就可以了！先來看struct Node長什麼樣子吧！

```
struct Node{
    static int size; //整個線段樹所代表的範圍有size這麼大
    int val;
    Node *l, *r;
    Node(int val): val(val), l(nullptr), r(nullptr){}
    Node(Node *l,Node *r): l(l), r(r){ pull(); }
    void pull() { if(l&&r) val = l->val + r->val; }
};
int Node::size; //static member 必須這樣宣告！否則吃CE
```

除了modify以外的函數都和原本的線段樹差不多，可以明顯的看到，我們在做修改的時候都是新建一個Node，而不會動到原本的！這樣就可以完整的保存修改的歷史（悠久的文化呀）版本了！

```
Node *modify(Node *o, int pos, int dif,
                        int l=0, int r = Node::size){
    if(l+1 >= r) return new Node(o->val + dif);
    int mid = l + (r-l>>1); //好習慣，防止(l + r)溢位
    if(pos < mid)
        return new Node(modify(o->l,pos,dif,l,mid), o->r);
    else
        return new Node(o->l, modify(o->r,pos,dif,mid,r));
    //其中一個變成新的，另外一個不用修改，指回原本的
}
```

剩下的就是一般的線段樹就好了（至少會和講義中的函數差不多），不需要變！每一次modify的時候，就將回傳的新Node小心翼翼地將它放進去存的地方，如陣列或 BIT（真的）等。

其他資料結構的持久化

不只線段樹可以持久化，只要你想要，都可以持久化，只是比較難：幾乎所有的樹狀結構都可以持久化，像是 Treap、Trie、左偏樹（可合併的 heap）、Linked List、並查集（當然，就不能路徑壓縮了，複雜度退化成 $O(\log n)$）等都可以寫成持久化，甚至序列看成 Treap 也可以持久化！可以去看 CodeChef 上有一篇文章叫做「Persistence Made Simple」(URL)，寫得很好，將持久化的精神解釋的很清楚。

1.5 動態開點

有時候因為題目的範圍非常的大，又不能（或不想）事先離散化，就必須用到動態開點的技巧！具體上就是沒碰到的點就讓他是空的，當查詢或修改必須用到時再為他開空間便可，和持久化的寫法很相似。這邊用偽指標的方法寫一份包含懶標的區間加值 RMQ 當作範例（沒有東西的話 default 會是 0）。

```
struct Segtree {
    struct node {
        int mx, lz;
        int l, r;
    } M[N*4];
    int tot;
    void pull(int p) {
        M[p].mx = max(M[M[p].l].mx, M[M[p].r].mx);
    }
    void upd(int p, int x) {
        M[p].mx += x, M[p].lz += x;
    }
    void push(int p) {
        if(!M[p].lz) return;
        if(!M[p].l) M[p].l = ++tot;
        if(!M[p].r) M[p].r = ++tot;
```

```cpp
        upd(M[p].l, M[p].lz);
        upd(M[p].r, M[p].lz);
        M[p].lz = 0;
    }
    void add(int &now, int ql, int qr, int x, int l, int r) {
        if(l >= qr || r <= ql) return;
        if (!now) now = ++tot;
        push(now);
        if(ql <= l && r <= qr) {
            upd(now, x);
            return;
        }
        int m = l+(r-l)/2;
        add(M[now].l, ql, qr, x, l, m);
        add(M[now].r, ql, qr, x, m, r);
        pull(now);
    }
    int query(int now, int ql, int qr, int l, int r) {
        if(l >= qr || r <= ql || !now) return 0;
        push(now);
        if(ql <= l && r <= qr) return M[now].mx;
        int m = l+(r-l)/2;
        return max(query(M[now].l, ql, qr, l, m),
            query(M[now].r, ql, qr, m, r));
    }
} sgt;

int main() {
    int root = 0;
    // add(root, ql, qr, x, 0, 1000000000);
    // query(root, ql, qr, 0, 1000000000);
}
```

樹堆 Treap 2

2.1 前言

treap=tree(binary search tree)+heap，是一種隨機平衡二元搜尋樹，對於任意的序列，他的插入、刪除、查詢操作期望複雜度皆為 $O(\log N)$。一般的 BST 雖然期望深度也是 $O(\log N)$，但是只要刻意餵給他排序好的序列，就一定會退化成鍊狀，這就是為什麼我們需要 treap 了。因為其期望複雜度低且程式碼相對簡單，故有些時候能拿來代替自平衡二元搜尋樹。

2.2 原理

treap 不同於一般二元搜尋樹的是，每個節點除了紀錄這個節點的數字（我們稱之為 key 值）外，同時會記錄一個 pri 值，struct大概長這樣：

```
int randseed=7122;
int rand(){return randseed=randseed*randseed%0xdefaced;}
struct node{
    int key,pri;
    node *l,*r;
    node(){};
    node(int _key):key(_key),pri(rand()){l=r=nullptr;}
};
```

樹中的 key 值會保持 BST 的性質（所以他是一種二元搜尋樹），pri 值則會保持 heap（二叉堆）的性質。這樣可以幹嘛呢 ??? 我們先對這個結構熟悉一下：

唯一性

在介紹二元搜尋樹時，我們知道對於同一組數字建立二元搜尋樹，他可以有很多種不同的樣子。但在 treap 中當每對 key,pri 都固定時，其建構出來的 treap 是唯一的。你可以想像，BST 的性質當中可以固定左右的關係（左邊的 key 值一定小於右邊），而 heap 的性質中可以固定上下的關係（上面的 pri 值一定小於下面），上下左右關係都固定的情形下，這棵 treap 自然就固定了啊！（你說這樣一點都不嚴謹？講的好像我知道怎麼嚴謹證明一樣）

當 pri 值 $=1\sim n$ 時

這個情況下，我們可以想像一棵最普通的二元搜尋樹（只有 key 值），我們依原本節點 pri 值從 $1\sim n$ 的順序一一插入，因為越後面插入的節點一定在越下面，因此 pri 值（= 插入時間）大的一定在越下面。如此一來，tree 和 heap 的性質就同時都滿足了。換句話說，每個 $1\sim n$ 的排列各自代表一種插入順序。

當 pri 值隨機時

根據上面的結論，pri 值的大小可以視為插入時間的先後，那如果 pri 值隨機的話，不就是隨機順序插入的意思嗎？而我們都知道，隨機順序插入的二元搜尋樹，其期望深度 $O(\log N)$（深度超過 $2\log N$ 的機率是 $\frac{1}{n^2}$），這也就是為什麼他的操作都可以 $O(\log N)$ 達成了。

2.3 實作方法

聽到這裡，你可能會想說：講那麼多，到底要怎麼在操作的同時保持這些性質啊？事實上，維持這個性質的方法有很多，主要有 merge-split treap 和旋轉式 treap。這邊要介紹的是 merge-split treap，因為這種方法好刻 code 又短。

merge-split tree 顧名思義，需要有 merge（合併）、split（分裂）兩個主要操作。而這兩個操作的實作方法其實就是遞迴，要詳細解釋也沒什麼意思，所以就直接切到程式碼的部分吧。

merge

這個操作是要將 a, b 兩個 treap 合併成一個，**其中 a 的所有 key 值都小於** b。

```
node *merge(node *a,node *b){ //將根節點為a,b的treap合併
    if(!a) return b; //base case
    if(!b) return a; //base case
    if(a->pri<b->pri){
        a->r=merge(a->r,b);
        return a;
    }else{
        b->l=merge(a,b->l);
        return b;
    } //pri值小的當父節點，大的當子節點。
}
```

split

這個操作是要將一個 treap 的 key 值 $\le k$ 的都丟到 a，其餘 $> k$ 的丟到 b。

```
void split(node *s,node *&a,int k,node *&b){
    if(!s) a=b=nullptr; //base case
```

```
    else if(s->key<=k) //s的key較小，故s和其左子樹都在左邊
        a=s,split(s->r,k,a->r,b); //分割右子樹
    else
        b=s,split(s->l,k,a,b->l); //分割左子樹
}
```

2.4 延伸操作

上面兩個操作看起來實在是沒有什麼實際用途，所以我們要來介紹一下利用它們組合而成的延伸操作。

insert

insert的做法只需要先將原本的 treap 拆開，再把左、新節點、右依序合併就好了。以下是程式碼：

```
void insert(node *&root,int t){
    node *a,*b;
    split(root,a,t,b);
    root=merge(merge(a,new node(t)),b);
}
```

erase

erase基本上就是反著做insert就好了。以下為程式碼：

```
void erase(node *&root,int t){
    node *a,*b,*c;
    split(root,a,t,b);
    root=b;
    split(root,b,t,c);
    root=merge(a,c);
}
```

2.5 比set更強大的功能

treap 是一個進階版的 BST，因此能改裝成具有更多功能的東西。其中，treap 最吸引人的功能就是 merge 和 split 了。不過我們先從一些初階的東西開始講起。

記錄 size

要記錄每個子樹的 size，就是自己的 size= 左子樹的 size+ 右子樹的 size+1。這簡單的運算一切就交給遞迴就好了（事實上就是線段樹的懶標）。這邊就不附上 code 了（後面會有）。

名次樹（rank tree）

rank tree 就是要能查詢各個 rank 的值以及每個值得 rank。在 search by key 時，可以用兩種方式（假設要查詢 key= k 的 rank）：

1. 將 key≤ k 的 split 出來，最後根節點的 size 就是 rank 了。
2. 同 BST 的查詢，一開始 rank= 1，每次若不是向左子樹走時 rank+= 左子樹的 size。

而 search by rank 時，假設要查詢 rank= k 的 key 值，從根節點開始：如果左子樹的 size+1 =rank，代表此節點的 key 值就是答案；如果 size≥rank, 則向左子樹遞迴；如果 size+1 <rank，則向右子樹遞迴，並且 rank-= 左子樹的 size+1。

split by rank

既然可以 search br rank，下一步就可以 split by rank 了！作法其實都一樣，只是要記的在遞迴後面加上 pull()，才能保持住正確的 size。

剛剛講了這麼多，重點就是為了下面這個神奇的東西：

序列轉 treap

什麼是序列轉 treap 呢？？？一般而言 treap 當中都是按照數字大小作為 key 值，達到 BST 的效果。然而現在，我們希望這棵 treap 中序尋訪的結果恰好就是原序列；也就是以 index 值作為 key 值。然而我們會知道，其實這邊的 key 值恰好就會是他的 rank，在查詢、split 時都可以用 rank 來進行就好，所以我們通常將 key 值省略不記（到後面你就會知道記與不記的差別了）。

懶人標記（lazy tag）

懶標你們一定不陌生，沒錯，他就是在線段樹上出現過的東西。這裡的 treap 恰好也是紀錄一個序列，所以，當然也可以使用懶標啦！假設我現在要對 $[l, r)$ 進行區間加值，我們可以先利用 split by rank 將整棵 treap 進行 split，分成 $[0, l)$，$[l, r)$，$[r, n)$ 這三棵 treap，然後在 $[l, r)$ 這棵 treap 的根上打懶標，最後再 merge 回去，就完成區間加值了。

區間操作

從上面的例子你應該可以發現，在 treap 當中我們可以在 $\log N$ 的時間內切出任意的區間，因此讓區間操作變得非常容易。這邊我們就舉交換區間的例子來讓大家更深刻體會 treap 的美妙吧！

習題 2.5.1: 交換區間

給定一序列及兩種操作：

1. 將 $[l_1, r_1)$，$[l_2, r_2)$ 兩個區間的位置交換。
2. 查詢序列第 k 的數字是多少。

每次都要搬動一個區間非常麻煩，但使用 treap 的話，只要切成 $[0, l_1)$，$[l_1, r_1)$，$[r_1, l_2)$，$[l_2, r_2)$，$[r_2, n)$ 五段，再依 $[0, l_1)$，$[l_2, r_2)$，$[r_1, l_2)$，$[l_1, r_1)$，$[r_2, n)$ 的順序 merge 回來就好了！是不是很簡單？我們來看 code 吧：

```
int rs=1e9+7;
int rand(){return rs=(rs*rs)%0xdefaced;}
struct node;
int s(node *a);
struct node{
    int a,pri,si;
    node *l,*r;
    node(){}
    node(int _a):a(_a),si(1),pri(rand()){l=r=nullptr;}
    void pull(){si=s(l)+s(r)+1;}
};
int s(node *a){return a?a->si:0;}
node *merg(node *a,node *b){
    if(!a) return b;
    if(!b) return a;
    if(a->pri<b->pri)
        return a->r=merg(a->r,b),a->pull(),a;
    else
        return b->l=merg(a,b->l),b->pull(),b;
}
void split(node *n,node *&a,int k,node *&b){
    if(!n) return a=b=nullptr,void();
    if(k>s(n->l)+1){
        a=n;
        split(n->r,a->r,k-s(n->l)-1,b);
        a->pull();
    }else{
        b=n;
        split(n->l,a,k,b->l);
        b->pull();
    }
}
int query(node *n,int k){ //0-base
    if(s(n->l)+1==k) return n->a;
    if(s(n->l)+1<k) return k-=s(n->l)+1,query(n->r,k);
    else return query(n->l,k);
}
void change(node *&n,int l1,int r1,int l2,int r2){
```

```
    //0-base,[)
    node *a,*b,*c,*d,*e;
    split(n,a,l1,b);n=b;
    split(n,b,r1-l1+1,c);n=c;
    split(n,c,l2-r1+1,d);n=d;
    split(n,d,r2-l2+1,e);
    n=merg(merg(merg(merg(a,d),c),b),e);
}
```

時間線段樹 3

3.1 例題

來看看萬年的唯一例題 (?)

習題 3.1.1: 【Gate】這個笑容由我來守護（TIOJ 1890）

給定一張 N 個點 M 條邊的圖，之後有 Q 次修改每次修改可能是新增一條邊，或者是刪除已經存在的一條邊對於每次修改後，請輸出這張圖當時的連通塊數量

如果現在題目只有加邊，那麼大家應該都會想到用 DSU 去處理吧！當題目只有刪邊，應該也可以很容易的想到可以時間倒過來用同樣的作法處理。

具體來說時間線段樹是什麼呢？我們可以發現，每條邊有特定的一些存活時間，在時間上對應的是一段區間，若我們離線將修改都讀進來，就能對時間這條軸開一棵線段樹表示，每個節點存這個點代表的區間被哪些邊的存活時間完整覆蓋。

接著，按照時間順序 DFS，並跟著維護 DSU，在遞迴到葉節點的時候就能得知該時間點的答案，但是在過程中我們會遇到必須要將 DSU 的合併動作「回復」，也就是說 DFS 離開某個節點的時候必須撤銷某個節點造成的影響，但只要把每個修改操作都記錄下來就能，這時候路徑壓縮的優化就會失效了（可以想想為什麼），但啟發式

```
#pragma GCC optimize ("Ofast")
#include <iostream>
#include <vector>
#include <utility>
#include <map>
#include <algorithm>
#include <deque>
#include <numeric>
#define pii pair<int,int>
#define piii pair<int,pair<int,int> >
#define piiii pair<pii, pii>
#define F first
#define S second
#define ericxiao ios_base::sync_with_stdio(0);cin.tie(0);
```

```
#define endl '\n'
using namespace std;

const int maxN = 6e5;

int T, N, M, Q, a, b, cc, ans[maxN + 10], dsu[maxN + 10], rk[
    maxN + 10];
char com;

vector<pii>seg[maxN * 4];


inline void init();
int  Find(int a);
inline piiii Merge(int a, int b);

void ins(int id, int l, int r, int ql, int qr, pii p){
    if(qr <= l || ql >= r) return;
    if(ql <= l && r <= qr){
        seg[id].push_back(p);
        return;
    }
    ins(2 * id + 1, l, (l + r)/2, ql, qr, p);
    ins(2 * id + 2, (l + r)/2, r, ql, qr, p);
}


void dfs(int id, int l, int r){
    vector<piiii> v;
    for(auto p : seg[id]){
        //do and record
        v.push_back(Merge(p.F, p.S));
    }
    vector<pii>().swap(seg[id]);
    //recurse
    if(l + 1 < r){
        dfs(2 * id + 1, l, (l + r) / 2);
        dfs(2 * id + 2, (l + r) / 2, r);
    } else {
        ans[l] = cc;
    }
    //undo
    reverse(v.begin(), v.end());
    for(piiii p : v){
        dsu[p.F.F] = p.F.F;
        cc += p.F.S;
        rk[p.S.F] = p.S.S;
    }
    vector<piiii>().swap(v);
```

```cpp
}

int main(){
    ericxiao;
    cin >> T;
    while(T--){
        cin >> N >> M >> Q;
        init();
        int ct = 0;
        map<pii,deque<int> > mp;
        for(int i = 0; i < M; ++i){
            cin >> a >> b;
            if(a > b) swap(a, b);
            mp[{a, b}].push_back(0);
            ct++;
        }
        /*
        com:
        N new edge
        D del edge
        */
        for(int i = 0; i < Q; ++i){
            cin >> com >> a >> b;
            if(a > b) swap(a, b);
            if(com == 'N'){
                mp[{a, b}].push_back(ct);
            } else if(com == 'D') {
                ins(0, 0, (M + Q), mp[{a, b}].front(), ct, {a
                    , b});
                mp[{a,b}].pop_front();
            }
            ct++;
        }

        for(auto p : mp){
            if(p.S.empty()) continue;
            while(p.S.size()){
                ins(0, 0, (M + Q), p.S.front(), ct, p.F);
                p.S.pop_front();
            }
        }
        //cout << "Ct = " << ct << endl;
        dfs(0, 0, (M + Q));
        for(int i = 0; i < Q; ++i){
            cout << ans[i + M] << '\n';
        }
    }
}

```

```cpp
inline void init(){
    iota(dsu, dsu + N, 0);
    fill(rk, rk + N, 1);
    cc = N;
}

int Find(int a){
    if(dsu[a] == a) return a;
    return Find(dsu[a]);
}

inline piiii Merge(int a, int b){
    //cout << "Gonna merge " << a << " and " << b << endl;
    //cout << "Find(a) = " << Find(a) << " and Find(b) = " << Find(b) << endl;
    if(Find(a) == Find(b)) return {{Find(a), 0}, {Find(b), rk[Find(b)]}};
    if(rk[Find(a)] > rk[Find(b)]) swap(a, b);
    piiii res = {{Find(a), 1}, {Find(b), rk[Find(b)]}};
    if(rk[Find(a)] == rk[Find(b)]) rk[Find(b)]++;
    //cout << "Merging " << a << " and " << b << endl;
    cc--;
    dsu[ Find(a) ] = Find(b);
    return res;
}
```