# 生成树和拓扑排序

2020年1月20日

黄哲威 hzwer
北京大学16级计算机科学

# 课程安排 2

并查集操作和优化

最小生成树的两个算法

拓扑排序以及应用

# 并查集

- 并查集是一种维护元素关系的数据结构，在并查集中，一个集合其实就是一棵树
- 初始时每个元素属于一个集合，我们对所有的点赋值 fa[i] = i，其实就是每个结点构成一个集合，或者说是一个自环
- 我们要支持两种操作，一种操作是查询一个集合的根，另一种操作是合并两个集合

# 并查集

- 找一个元素的根，就是从这个元素出发，不断往父亲结点走，直到找到一个有自环的结点
- 合并两个集合，就是先找到两个根，再在它们之间连一条边
- 如果我们随意合并集合，一个集合的深度可能变成 O(n)，因此我们需要一些优化方法

# 并查集

```
int fa[N];
int getroot(int x)
{
    return x == fa[x]? x : getroot(fa[x]);
}
void merge(int x, int y)
{
    int p = getroot(x), q = getroot(y);
    fa[p] = q;
}
int main()
{
    cin >> n;
    for(int i = 1; i <= n; i++)
        fa[i] = i; // 初始化
    return 0;
}
```

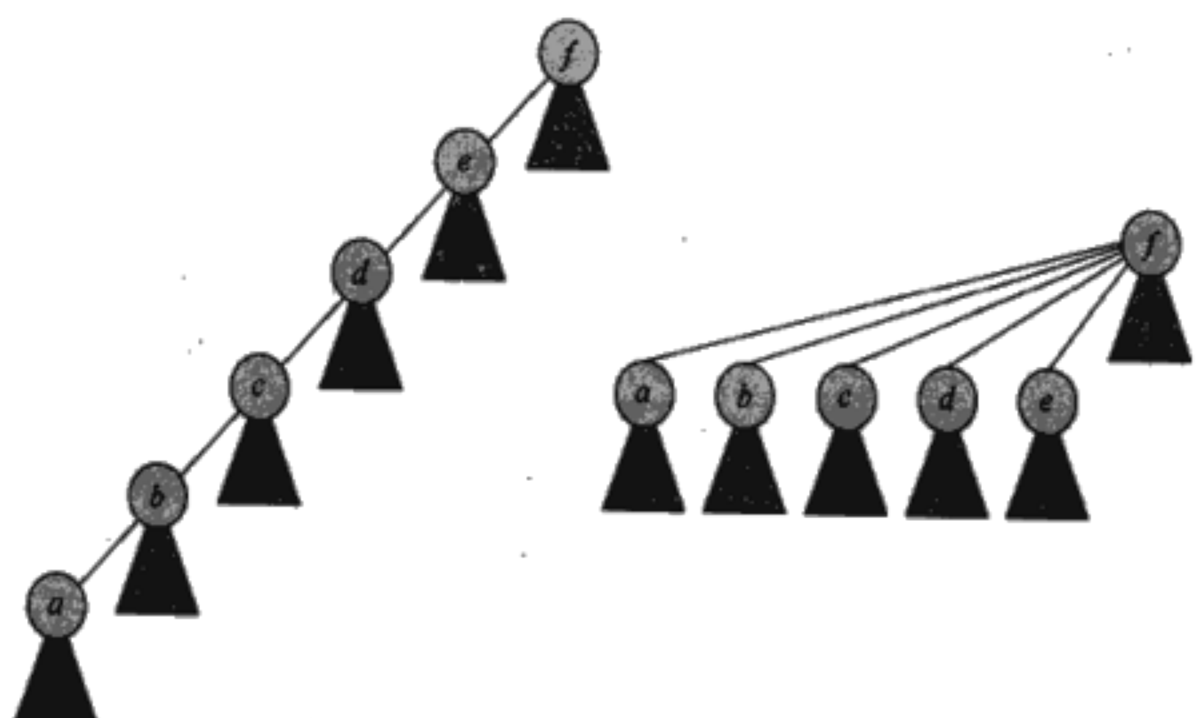
# 带权并查集

- 维护元素所属集合的同时记录集合的一些信息：
- 维护集合的最大最小值；维护集合的和；维护其它的信息
- 很像树形递推

# 并查集优化

- 路径压缩：当我们经过递归找到祖先节点后，回溯的时候顺便将它的子孙节点都直接指向祖先。

- 启发式合并（按秩合并）：维护并查集的树高，每次合并的时候都用秩小的指向秩大的。




a)　　　　　　　　b)

# 路径压缩

```
int fa[N];
int getroot(int x)
{
    return x == fa[x]? x : fa[x] = getroot(fa[x]);
}
void merge(int x, int y)
{
    int p = getroot(x), q = getroot(y);
    fa[p] = q;
}
int main()
{
    cin >> n;
    for(int i = 1; i <= n; i++)
        fa[i] = i; // 初始化
    return 0;
}
```

# 按秩合并

```
int fa[N], h[N];
int getroot(int x)
{
    return x == fa[x]? x : getroot(fa[x]);
}
void merge(int x, int y)
{
    int p = getroot(x), q = getroot(y);
    if(h[p] > h[q])swap(p, q);
    fa[p] = q; h[q]++;
}
int main()
{
    cin >> n;
    for(int i = 1; i <= n; i++)
        fa[i] = i; // 初始化
    return 0;
}
```

# 并查集优化

- 当同时使用按秩合并和路径压缩时，最坏情况运行时间为 $O(m\alpha(n))$，n 是元素个数，m 是操作个数， $\alpha(n)$ 是一个增长极其缓慢的函数。

- 在各种实际情况中，可以把这个运行时间看作与 $m$ 成线性关系。

# 基础概念

拓扑图 (有向无环图，DAG)：不存在任何环的连通 (有向) 图

树：若 $G = (V, E)$ 中任意两点间都连通，且 $|E|$ 最小，则 $G$ 称为树，$|E| = |V| - 1$

生成树：若 $G = (V, E)$ 的生成子图 $G' = (V, E')$ 是一棵树，则称 $G'$ 是 $G$ 的一棵生成树

# 生成树

求解最小生成树常用的有两种算法:

Kruskal 算法:时间复杂度 O(mlogm)

Prim 算法，时间复杂度 O(nlogm)

Kruskal 比较容易实现，所以一般只在遇到大的稠密图时才用 Prim 算法

# 生成树 - Kruskal

1 将所有边按权值 w(e) 的大小排序

2 初始选择边集 $E = \varnothing$

3 按顺序考虑每条边 e，e 与已在 E 中的边不构成环则可选择。$E = E + \{e\}$. 若构成环则放弃 e

4 选出 $n - 1$ 条边后 E 即为一棵最小生成树，否则原图不连通

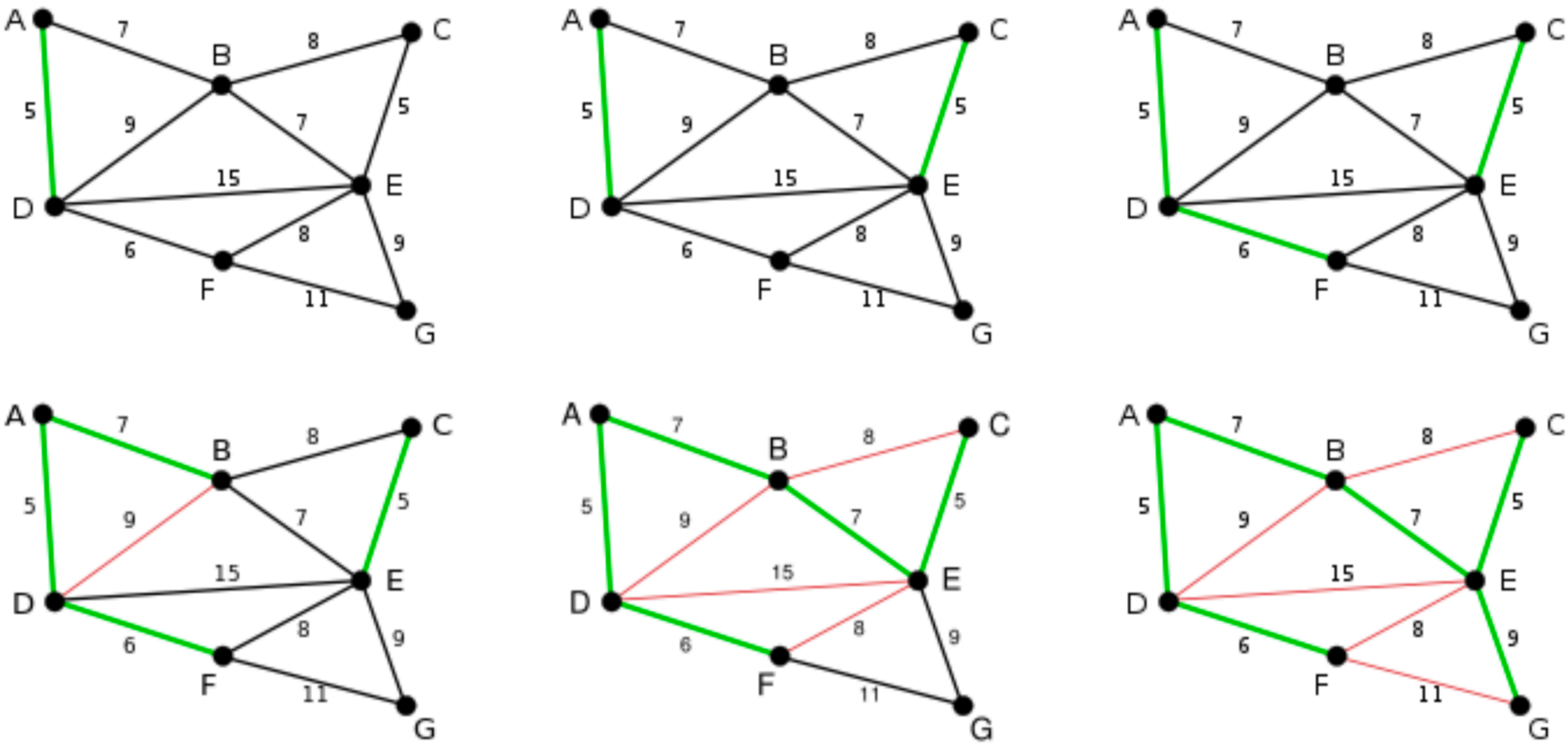
使用并查集维护过程 3 中的选择

时间复杂度 $O(m\log m)$

# 生成树 - Kruskal

```
int n, m, ans;
struct edge{
    int u, v, w;
}e[M];
bool cmp(edge a, edge b)
{
    return a.w < b.w;
}
int main()
{
    cin >> n >> m;
    for(int i = 1; i <= m; i++)
        cin >> e[i].u >> e[i].v >> e[i].w;
    // 这里应该有个并查集初始化
    sort(e + 1, e + m + 1, cmp);
    for(int i = 1; i <= m; i++)
    {
        int p = getroot(e[i].u), q = getroot(e[i].v);
        if(p != q)
        {
            fa[p] = q;
            ans += e[i].w;
        }
    }
    return 0;
}
```

# 生成树 - Kruskal


A B C D E F G
7 8 5 9 7 5 15 6 8 9 11
A B C D E F G
7 8 5 9 7 5 15 6 8 9 11
A B C D E F G
7 8 5 9 7 5 15 6 8 9 11
A B C D E F G
7 8 5 9 7 5 15 6 8 9 11
A B C D E F G
7 8 5 9 7 5 15 6 8 9 11
A B C D E F G
7 8 5 9 7 5 15 6 8 9 11

# 生成树 - Kruskal 证明

令 Kruskal 算法得到的树为 K，有一棵最小生成树 T，假设他们不同

找到边权最小的在 K 但不在 T 中的边 e

把 e 加入 T 中，形成一个环，删掉这个环中一条不在 K 中的边 e′，得到新生成树 T′

若不存在 e′ 则 K 存在环，矛盾
若 $w(e') > w(e)$，则 T′ 权值和小于 T，矛盾

若 $w(e') < w(e)$，则 Kruskal 执行时先考虑了 e′，由于成环没加入 e′，因此在 e′ 之前加入的边权值均 $\le w(e') < w(e)$. 由 e 的定义，K 中边权小于 $w(e)$ 的边均在 T 中，说明 T 中边与 e′ 会成环，矛盾

$w(e') = w(e)$，在 T 中用 e 换掉 e′
有限步后可把 T 变为 K 且权值不变，因此 K 就是最小生成树

# 生成树 - Prim

1 初始选择边集 $E = \varnothing$，点集 $V$ = {任意一个结点}

2 选择一条权值 $w$ 最小的边 $e = (u, v)$，满足 $u \in V, v \text{ not} \in V$

3 $E = E + \{e\}$ , $V = V + \{v\}$

4 点集 $V$ 内若包含所有结点则算法结束，否则返回第二步
使用优先队列加速过程 2

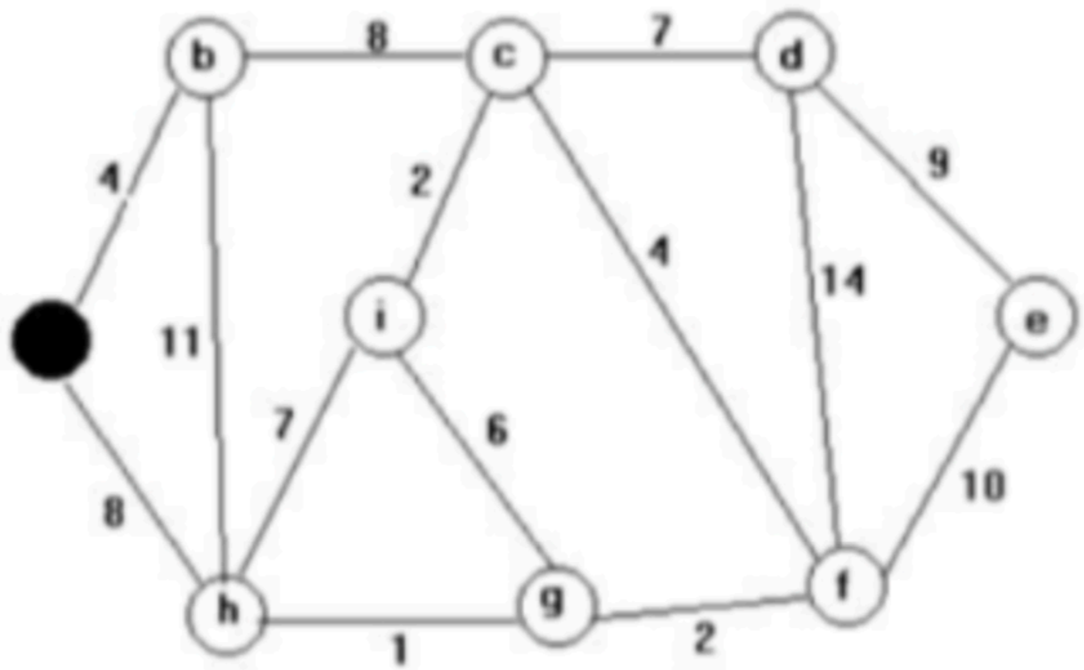
$O(n \log m)$

# 生成树 - Prim

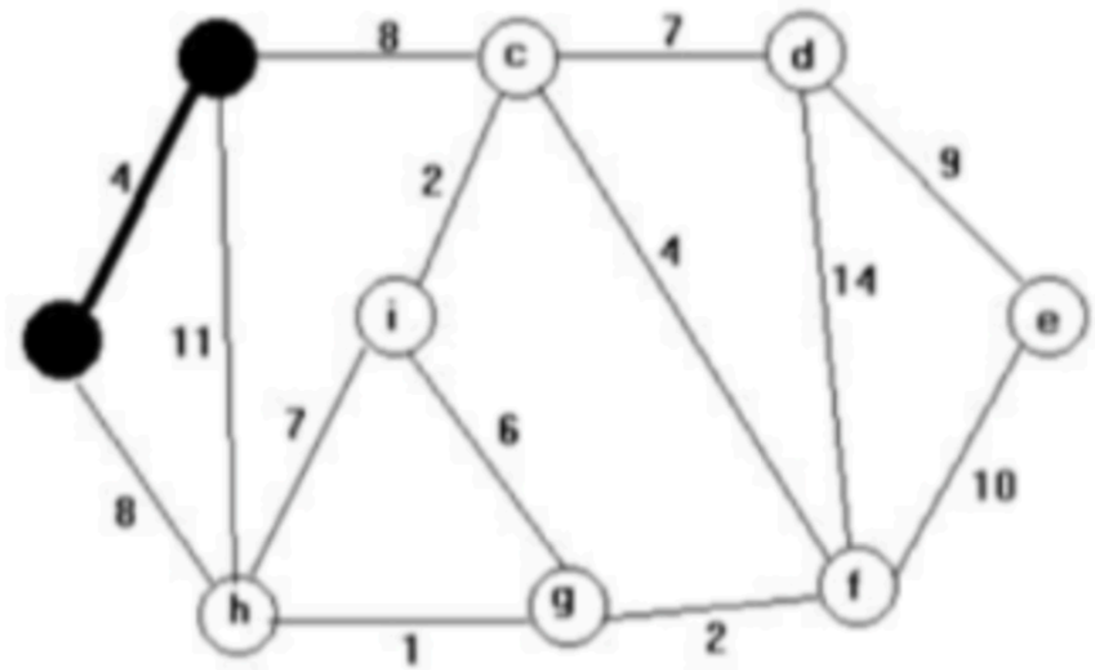
```
#include<queue>
#include <vector>
vector<int> e[N], w[N];
int dis[N];
bool vis[N];
#define pa pair<int, int>
priority_queue<pa, vector<pa>, greater<pa> >q;
void prim(int s)
{
    memset(dis, 127, sizeof(dis));
    dis[s] = 0; q.push(make_pair(0, s));
    while(!q.empty())
    {
        int u = q.top().second; q.pop(); //找当前dis最小的结点
        if(vis[u])continue; vis[u] = 1; //vis=1表示这个点      确定过了
        for(int i = 0; i < e[u].size(); i++)
        {
            int v = e[u][i];
            if(!vis[v] &&          w[u][i] < dis[v])
            {
                dis[v] =          w[u][i];
                q.push(make_pair(dis[v], v));
            }
        }
    }
}
```

# 生成树 - Prim

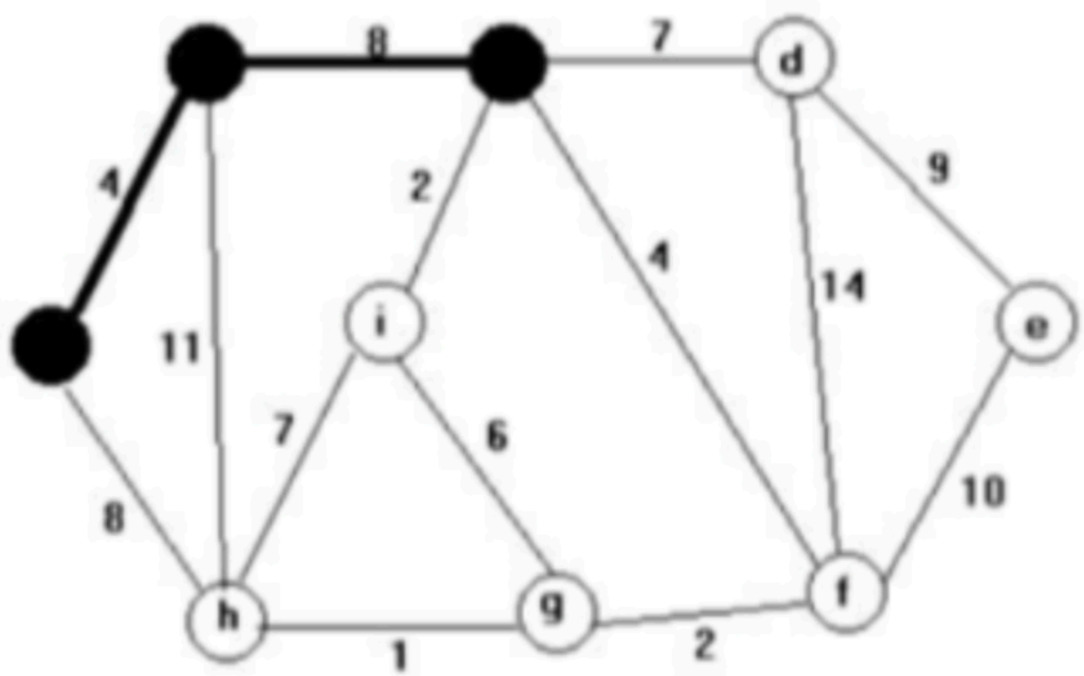
Iteration 0:



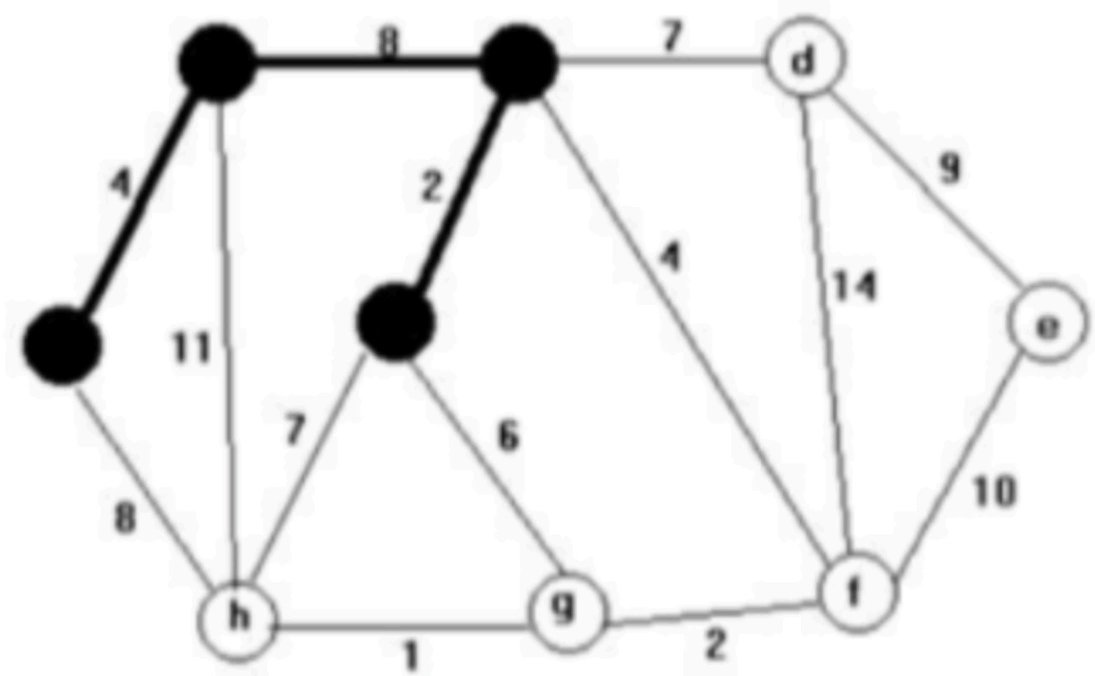

Iteration 1:




Iteration 2:

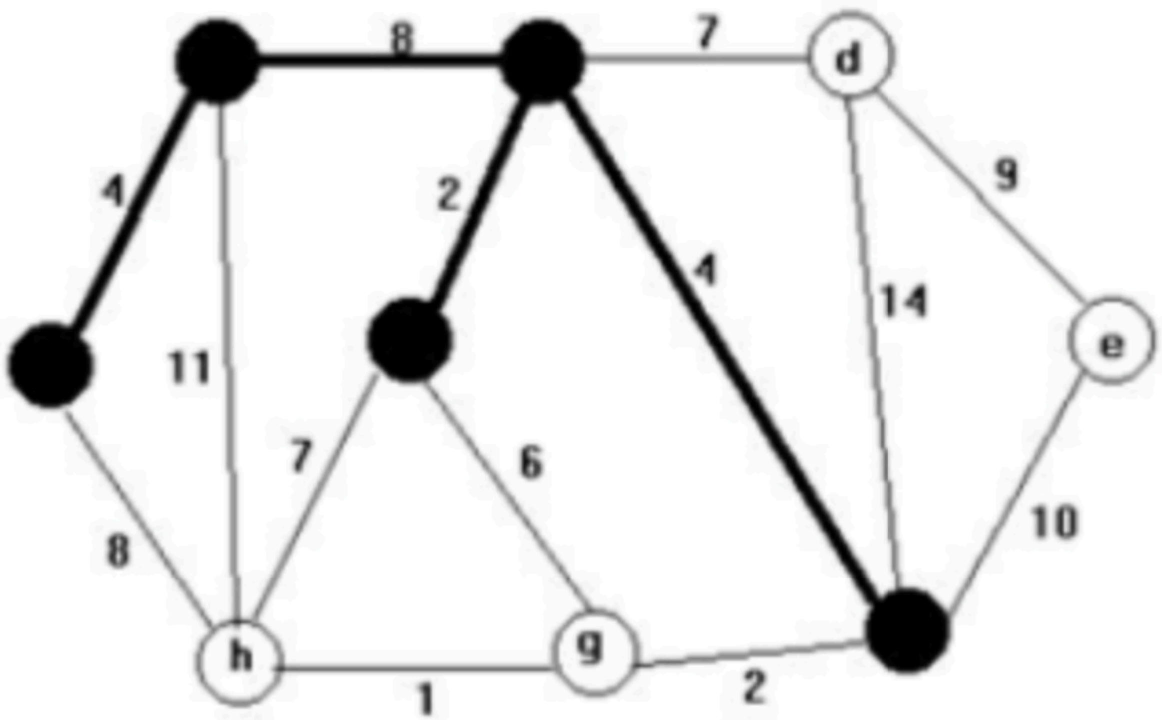



Iteration 3:

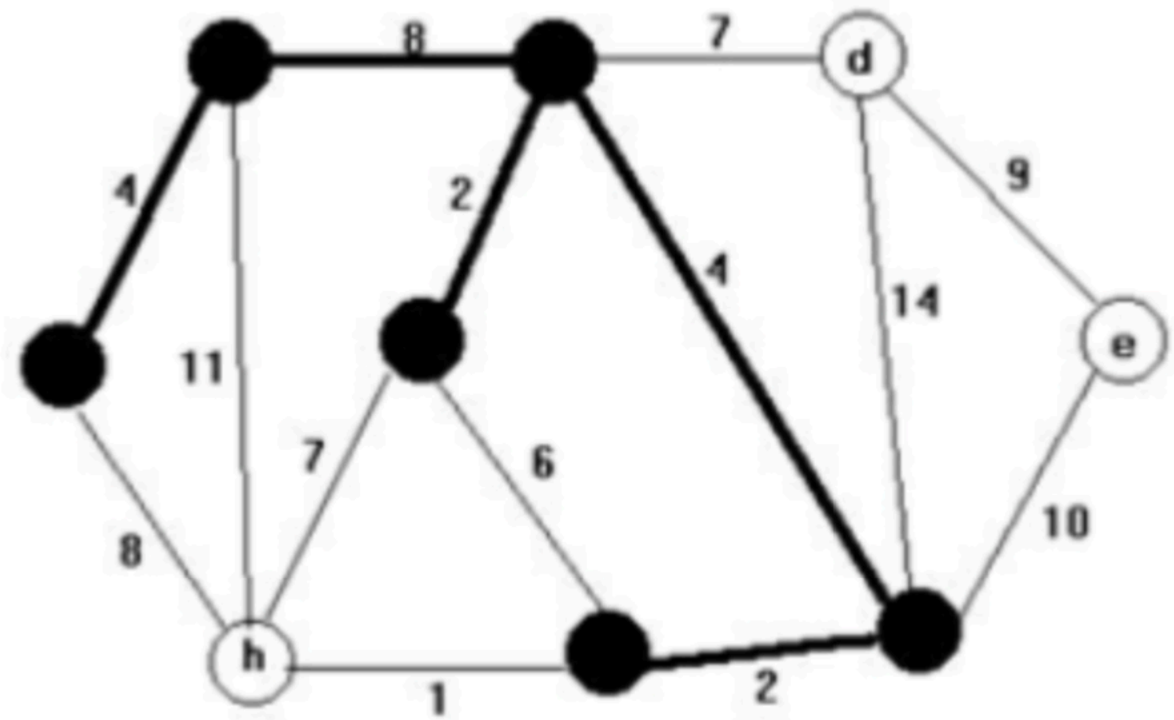


# 生成树 - Prim

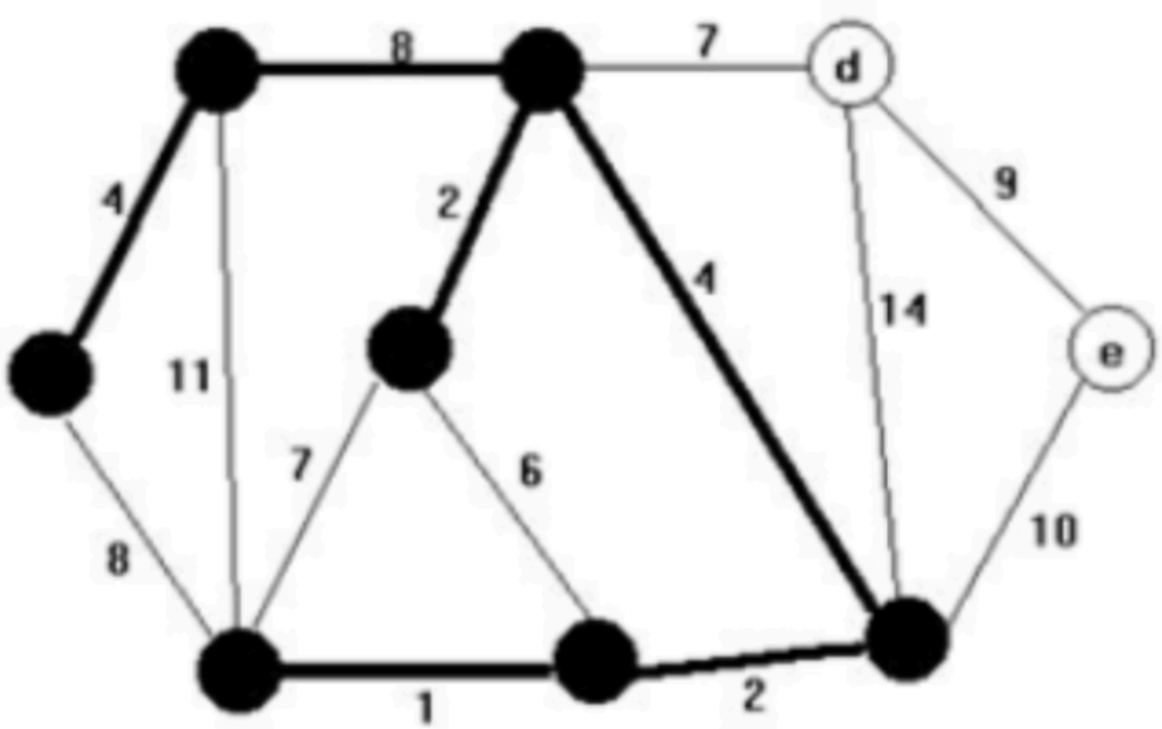
Iteration 4:



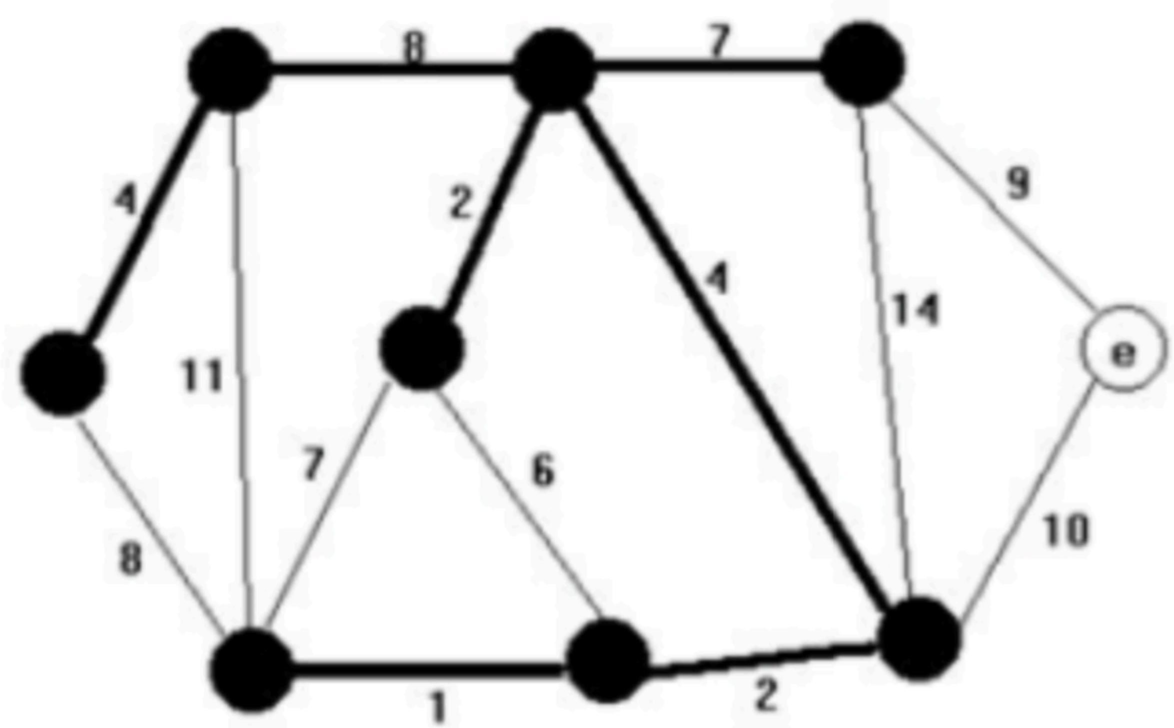

Iteration 5:




Iteration 6:




Iteration 7:



# 生成树 - Prim 证明

令 Prim 算法得到的树为 P，有一棵最小生成树 T，假设他们不同

假设前 k–1 步 P 选择的边都在 T 中，令此时的树为 P′

第k步选择的e=(u,v)不在T中，假设u在P′ 中，而v不在

T 中必有一条 u → v 的路径，路径上必有一条边 e′ = (x,y) 满足此时 x 在 P′ 中而y不在

若 w(e′) > w(e) 则在 T 中用 e 换掉 e′ 可得到一个更小的生成树，矛盾 若 w(e′) < w(e) 则第 k 步时选的是 e′ 而不是 e，矛盾
若 w(e′) = w(e)，在 T 中用 e 换掉 e′, 则 P 前 k 步中选择边都在 T 中 有限步后可把 T 变为 P 且权值不变，因此 P 就是最小生成树

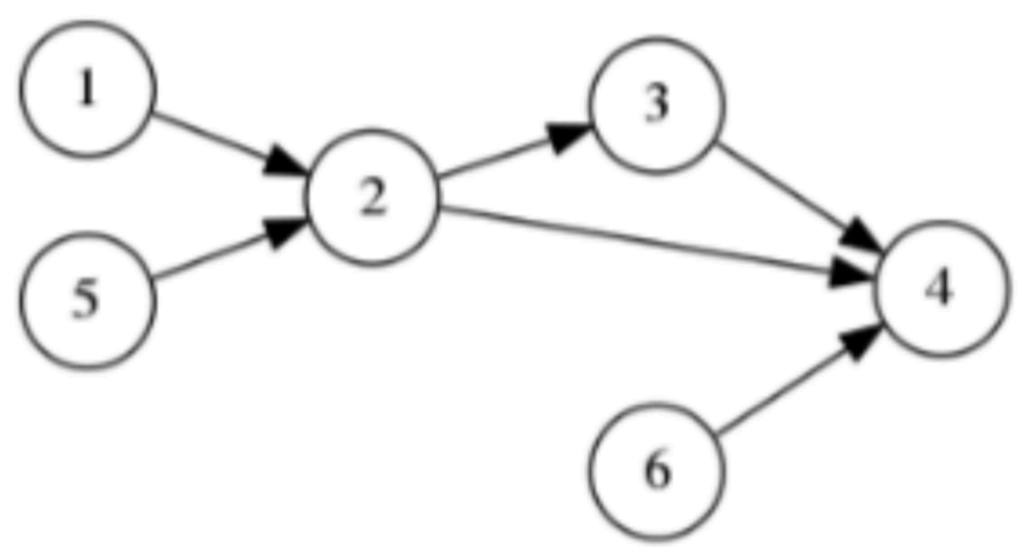
# 拓扑排序

现在来考虑一个问题：现在有一个工程，这个工程被分成了很多部分。有一些部分要求前面某些部分完成后才可以开始进行。有些部分则可以同时进行。

我们可以把每个部分看作一个结点，这些限制看成是有向边。

比如说这张图就可以看成是这样一个限制。这样的图没有环!

因此这样的图也被成为有向无环图 (DAG)。



# 拓扑排序

求出一个这个工程的工作序列的算法被成为拓扑排序。 比如说 1，5，2，3，6，4 就可以算作一个工作序列。 拓扑排序的过程大概是这样的:

1 选择一个入度为 0 的结点并直接输出。

2 删除这个结点以及与它关联的所有边。

3 重复步骤 (1) 和 (2)，直到找不到入度为 0 的结点。

通常情况下，在实现的时候会维护一个队列以及每个结点的入度。在删除边的时候顺便把相应结点的入度减去，当这个结点入度为 0 的时候直接将其加入队列。

# 拓扑排序

```
int top, q[N];
int main()
{
    cin >> n >> m;
    for(int i = 1; i <= m; i++)
    {
        int u, v;
        cin >> u >> v;
        e[u].push_back(v);
        d[v]++;
    }
    for(int i = 1; i <= n; i++)
        if(!d[i])
            q[++top] = i;
    while(top)
    {
        int u = q[top]; top--; // cout << u << ' ';
        for(int i = 0; i < e[u].size(); i++)
        {
            int v = e[u][i];
            d[v]--;
            if(!d[v])
                q[++top] = v;
        }
    }
    return 0;
}
```

# 练习题

# 经典题

给定一个点数为 n，边数为 m 的拓扑图，保证 s 点能走到 t 点。

问图中有多少个点满足：删除后，s 点无法到达 t 点。

$n, m \leq 100000$

# 经典题

拓扑排序，然后 dp，f[u] 表示从 s 点走到 u 点的方案数，g[u] 表示从 u 点走到 t 点的方案数。若点 u 满足 f[u] * g[u] = f[s]，则点 u 满足条件。

# 团伙

- n 个强盗，强盗之间可能有朋友或者敌人关系。
- 1. 我的朋友的朋友是我的朋友。2. 我敌人的敌人也是我的朋友。 两个强盗是同一团伙的条件是当且仅当他们是朋友。
- 现在给定 m 条强盗之间的关系，问最多有多少个强盗团伙。保证输入的强盗关系的合法性。
- $0 \le n \le 100000$, $0 \le m \le 1000000$

# 团伙

- 将有关系的强盗并集
- 若 2 个强盗是朋友，则连一条边权为 0 的边
- 若 2 个强盗是敌人，则连一条边权为 1 的边
- 若 2 个强盗“有关系”，则这 2 个强盗属于同一个连通图，反之亦然
- 2 个强盗之间的任意一条路径阐述了他们之间的关系，将路径上的所有边权加起来，对 2 取模，为 0 则是朋友，为 1 则是敌人
- 实现合并的时候，要计算两个根之间的关系

# CHSEQ22.Chef and Favourite Sequence

- 一长度为 n 的 0,1 序列，初始时所有元素为 0。
- 有 m 个区间 [Li, Ri]，可以任选若干个区间进行区间翻转操作 (0 变 1, 1 变 0)，问可以得到多少种不同的序列
- 答案对 (10^9  + 7) 取模
- 1 ≤ n, m ≤ 10^5

# CHSEQ22.Chef and Favourite Sequence

- 考虑将一段区间取反，相当于将它的差分序列的 l 和 r+1 俩个位置取反
- 如果某一项操作可以用其它一些操作替代的话，它就可以直接被舍弃了，可以用一个并查集来维护
- 即把所有操作区间的左右端点在并查集中合并，如果一个区间的左右端点已经并在一起了，它就可以被其它操作区间替代
- 最后剩下 k 个操作，答案就是 2^k

# 环

- 给出一张 $n$ 个点 $m$ 条边的无向图，依次询问某个点所在连通块是否为环，是否为树?

- $1 \le n, m \le 10^6$

# 环

- 依次连边，当一条边连接的两个点处于同一连通块时，说明这个连通块有环了，可以根据出入度判断连通块的形状。
- 没有环的连通块就是树。

# Usaco2012Jan.Bovine Alliance

- 给出 n 个点 m 条边的图（不一定连通），现把点和边分组。
- 每条边要和它相邻两点之一分在一组。点不可以和多条边一组，但点可以单独一组，问分组方案数。
- 答案对 10^9 + 7 取模。
- 1 ≤ n, m ≤ 10^5

# Usaco2012Jan.Bovine Alliance

- 连通块是独立的，设某个连通块点数为 n，边数为 m
- 若 m > n，边数大于点数，必然有一个点要与多条边分在一组，无解
- 若 m = n， 环 + 外向树，解为 2
- 若 m = n - 1, 树，解为 n
- 若 m < n - 1, 不存在这样的连通块
- 使用并查集维护连通块的边数和点数

# CF698B. Fix a Tree

给出 n 个结点的父亲，问至少修改多少个结点的父亲，能使整张图变成一棵树(根的父亲为自己)，要求输出任一方案。

其中 $1 \leq n \leq 200000$。

# CF698B. Fix a Tree

思考环和链的答案

图的各个弱连通块是环 + 内向树，或者树/环。

先用拓扑排序把内向树消掉

剩下来的是一些环，每个环随便选一个结点当根，然后再把所有的根连在一起。

答案是环数 - (是否存在自环)

# UOJ14. DZY Loves Graph

- DZY 开始有 n 个点，现在他对这 n 个点进行了 m 次操作，对于第 i 个操作（从 1 开始编号）有可能的三种情况：

- Add a b： 表示在 a 与 b 之间连了一条长度为 i 的边（注意，i是操作编号）。

- Delete k： 表示删除了当前图中边权最大的 k 条边。

- Return：表示撤销第 i – 1 次操作。保证第 i–1 次不是 Return 操作。

- 1 <= n <= 5 * 10^5，部分数据只有 Add，部分数据没有 Return 操作。

- 每次操作以后，求全图的最小生成树边权和，若不存在输出 0。

# UOJ14. DZY Loves Graph

- 只有加边的情况，那么当图连通后第一次有了最小生成树。
- 因为加的边权是单调递增的，最小生成树保持不变直到最后。
- 问题是怎么从并查集删除掉最后添加的 K 条边?

# UOJ14. DZY Loves Graph

- 使用按秩合并的并查集，由于一条边被插入后就不会修改，直接模拟加边删边。

- 如何实现 Return?

- 使用类似离线的做法，我们在做第 i 个操作的时候可以知道第 i+1 个操作是否是Return操作。

- 如果第 i 个操作是 Add 操作，那么第 i+1 个操作是否是 Return 并没有太大的影响，因为加入一条边和删除一条边的时间代价都是O(logn)。

- 如果第 i 个操作是 Delete 操作，第 i+1 个操作是 Return 操作，说明其实删边操作是假的，只要求一个答案。我们可以事先存下使用当前权值前 k 小的边时最小生成树大小直接输出即可。

# 链型网络

- 给定一张无重边，无自环的无向图，n 个点，m 次操作

- 一开始图中没有边，每次操作可以加边，或询问有多少个点满足：将该点删除后，原图的每个连通块都为一条链

- $1 <= n, m <= 10^5$

# 链型网络

- 思考下面一些简单的情况:
  原图为若干条链，则答案为点数 n;
  原图为单个简单环和若干条链，则答案为环大小;
  原图中超过一个连通块有环，答案为 0.
  原图中一个连通块为一个点连接三条链，答案为 4;
  原图中一个连通块为一个点连接三条以上的链，答案为 1;

- 但考虑到环套树这样更复杂的情况，上述分类讨论也无能为力

# 链型网络

- 我们需要思考链本身的性质，一个图的每个连通块为链，等价于 每个点的度数小于等于 2 且无环

- 转化后的条件明显更有利于解决问题

- 首先考虑图中是否有度数大于等于 3 的点，如果存在一个度数大于等于 3 的点 u, 因为要保证去掉一个点后每个点的度数都小于等于 2，我们要么去掉 u，要么在 u 度数恰好为 3 时，去掉 u 的三个相邻点中的一个，而其他的点均不可能成为答案

# 链型网络

- 在加边过程中第一次出现度数为 3 的点的时候，对于可能成为答案的 4 个点，分别维护一个去掉该点之后的图
- 然后在 4 个图中分别进行判断
- 只要在加边时判断每个点度数都小于等于 2, 再用并查集判断是否有环即可

# 链型网络

- 剩下的就是每个点的度数都小于等于 2 的情况，这样每个连通块 只能是链或者简单环，我们只需采用一开始的分类讨论即可.

- 原图为若干条链，则答案为点数 $N$;
  原图为单个简单环加若干条链，则答案为环大小;

- 原图中超过一个连通块有环，答案为 0.

- 这只需要维护一个记录集合大小的并查集就能做到.