1.图的存储

1.1.链式前向星

主要思想为：以数组来模拟链表

- 有一个边数组 `Edges` ，每条边通过其 `id` 进行索引。比如 `Edges[1]`
- 有一个头数组 `head` ，代表以每个顶点为头的第一条边的 `id` 。

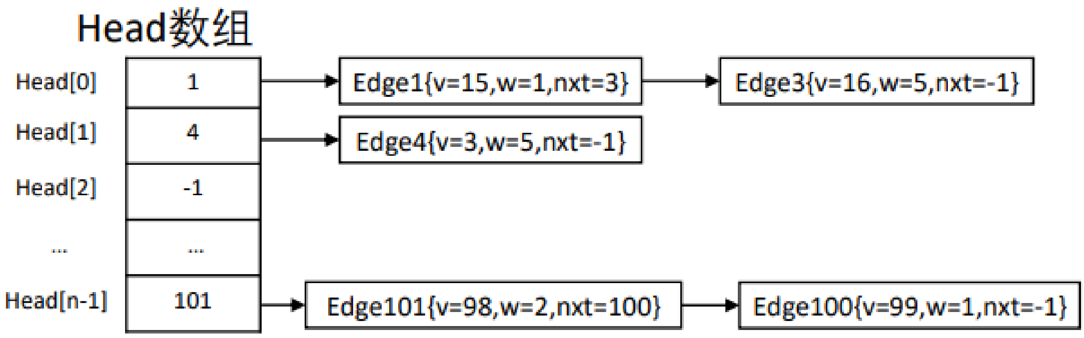
```
struct Edge{
    int u, v, w, next // u是边的起始节点,v是终止节点,w是权重,nxt是下一条边的id
}Edges[MAXM];
int head[MAXN], tot; //tot 是 Edges 的下标

void init(int n){
    tot = 0;
    memset(head, -1, sizeof(head)); // head初始值默认没有连接边
}

void addEdge(int u, int v, int w){
    //构造一条边
    Edges[tot].u = u;
    Edges[tot].v = v;
```

```
    Edges[tot].w = w;
    Edges[tot].nxt = head[u];
    //插入对应的位置(其实没有插入,因为用的是数组)
    head[u] = tot++;
}
```

如下图所示：



注意： `head[u]` 后面的边及他们的 `nxt` 都是以 `u` 作为起点的。

2.图的遍历

前向星的遍历是以边为基础的，通过边来寻找其他信息。而邻接数组是以点为基础的，我们考虑的是两个点之间是否连通。

2.1.图的遍历

- ### 2.1.1.DFS

简单来说，就是一条路走到黑，将一条路走到头再尝试别的路，因此我们说这是深度优先。因为我们倾向于先将路往深处走

```
void dfs(int u){
    //...
    for(int i = head[u]; i != -1; i = Edges[i].nxt){ // 前向星的遍历方式，遍历 u 的邻接边
        if(!vis[Edges[i].v]){
            vis[Edges[i].v] = true;
            dfs(Edges[i].v);
        }
    }
}
```

复杂度：`O(m)`，m 是边的数目，其实就是最坏需要遍历所有边。

如果使用的是邻接数组，复杂度是 `O(n^2)`，n 是点的数目。一般 $m < n^2$，当所有点之间都互相有边时，$m = n^2$

2.1.2.BFS

简单来说，就是先访问邻居，再考虑访问与邻居相接的节点。因此我们说是广度优先。因为我们的访问是按层次的，像波一样扩散的，而不是首先考虑一条路走到头。

2.2.树的直径

树的直径就是树中任意两点之间距离的最大值。

可以通过两次遍历求得

- 随便选一个点 P，然后遍历一次，找到距离其最远的点 Q
- 以 Q 为起始点再遍历一次，找到距离其最远的点 M
- QM 就是直径

3.并查集

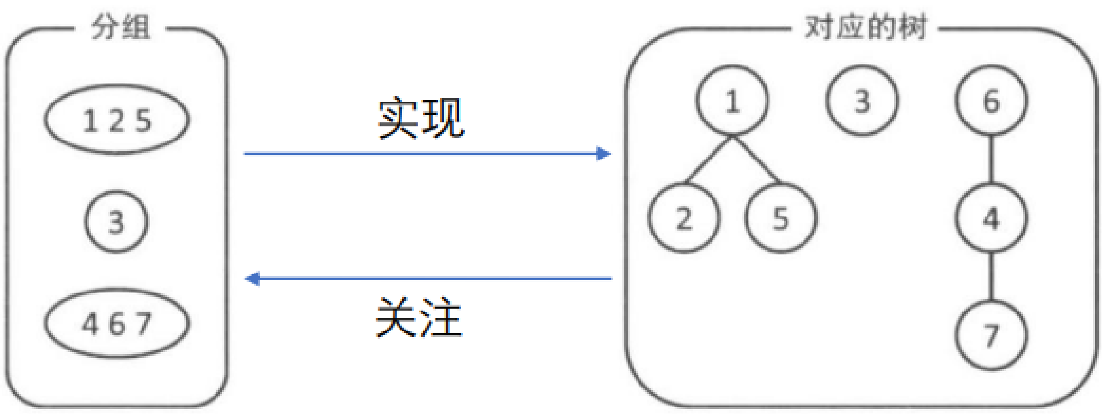
并查集是一种用来管理**分组**的数据结构，可以高效进行下面的操作

- 查询元素 A 和 B 是否属于同一组
- 合并 A 和 B 所在的组

3.1.实现

并查集可以使用类似于数形的结构实现，但是我们并不在意并查集的结构，只在意元素所在的分组。

因此我们直接从分组中选一个代表元素，来标记一个组。



我们认为一棵树是一个组，树的根节点用来代表这个组。

3.2.初始化

初始化时，我们认为每个元素都是独立的组，每个元素都是自己组的代表

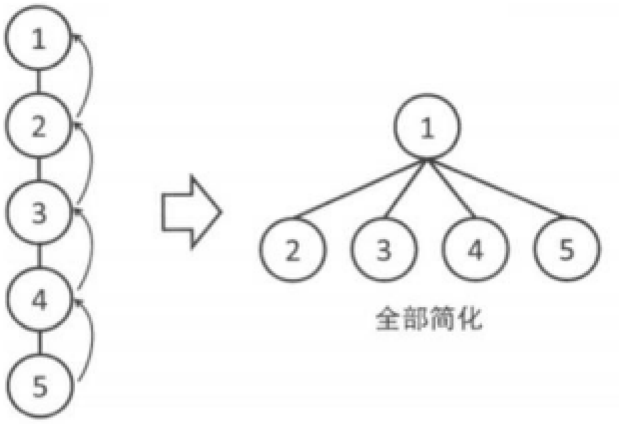
```
int par[maxn]; //存储一个元素的父亲
void init(int n){
    for(int i = 0; i < n; i++)
        par[i] = i; //代表是自己,注意这里使用的是索引
}
```

3.3.查询

因为使用根节点来代表一个组，我们查询时查的是当前组的代表元素，因此我们应该找到根节点，也就是当前节点的祖先（父亲的父亲的父亲……）

```
int find(int x){
    if(par[x] == x) return x; // 根节点的父亲是他自己
    return find(par[x]) //通过递归得找父亲节点来找根节点
}
```

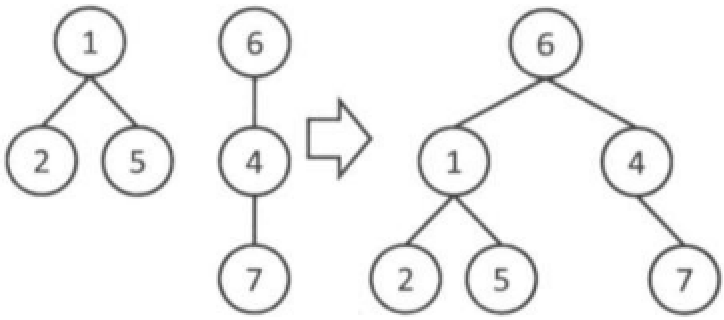
我们递归查找可能会比较慢，可以使用路径压缩的方法，将一个元素直接挂在他的祖先（根节点）的下面，而不是父亲的下面




```
int find(int x){
    if(par[x] == x) return x;
    return par[x] = find(par[x]);
}
```

3.4.合并

把一个分组的根挂到另一个分组的根




```
bool unite(int x, int y){ // 返回是否合并成功
    x = find(x);
    y = find(y);
    if(x == y) return false; // 合并失败,本来就在一个组中
    par[x] = y; // x所在组挂在y所在组下,反过来也行
    return true;
}
```

4.最小生成树

4.1.Kruskal

每次贪心地尝试将图中最小的非树边标记为树边，非法则跳过。

- 将全部边按照权值由小到大排序
- 按顺序（边权由小到大）考虑每条边。只要这条边和我们已经选择的边**不构成圈，就保留**。否则放弃这条边
- 成功选择 （n - 1）条边后，就生成一棵最小生成树。如果无法选出 （n - 1）条边，则原图不连通

4.2.Prim

是基于点的贪心算法，其核心思想是：维护一个连通点集，每次都从不在该点集内的点中，选出一个连通该点集的代价最小的点加入这个点集。

简单来说，就是维护一个当前的联通图，每次找一个与当前图中的端点连通的最短的边加入。

4.3.例题

使用 Kruskal 算法求解 [P3366 【模板】最小生成树 - 洛谷 | 计算机科学教育新生态 (luogu.com.cn)]()

```
#include<bits/stdc++.h>
using namespace std;
struct Edge{
    int u, v, w;
}Edges[200005]; // 存储边的信息
int par[5005]; // 用于实现并查集
int tot = 0;

bool cmp(Edge a, Edge b){ // 按权重排序边时使用
    return a.w < b.w;
}

int find_par(int x){ // 并查集的查找
    if(par[x] == x) return x;
    return par[x] = find_par(par[x]);
}

bool add(int x, int y){ // 并查集合并
    x = find_par(x);
    y = find_par(y);
    if(x == y) return false;
    par[x] = y;
    return true;
}

int main()
{
```

```cpp
    int n , m;
    cin >> n >> m;
    int u_, v_, w_;
    int cnt = 0;
    int ans = 0;
    for(int i = 0; i < n; i++)
        par[i] = i;
    for(int i = 1; i <= m; i++)
        cin >> Edges[tot].u >> Edges[tot].v >> Edges[tot++].w;

    sort(Edges, Edges + m - 1, cmp); // 按权重从小到大排序
    for(int i = 0; i < m; i++){
        if(add(Edges[i].u, Edges[i].v)){ // 一条边的两个端点在同一组,则成环,否则可以加入
            cnt++;
            ans += Edges[i].w;
        }
        if(cnt == n - 1){
            cout << ans;
            return 0;
        }
    }
    cout << "orz";
}
```