社交网络

并查集的应用

范舒舰 程铭 王子申 罗淏予

小组分工

- 程铭　基础代码结构
- 范舒舰　代码挑战
- 王子申　测试、优化代码
- 罗溪予　测试、PPT制作

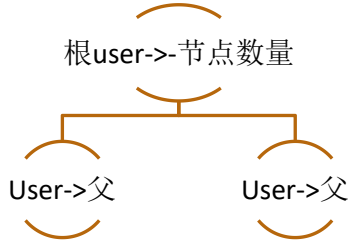
数据录入与输出

初始化

```
vector<string> allUsers;
map<string, int> userToIndex;
UFSets<string> uf(allUsers.data(), 0);
map<string, set<string>> friends;
set<string> usedUsers;
int circleCount = 0;
int N = 0;
```

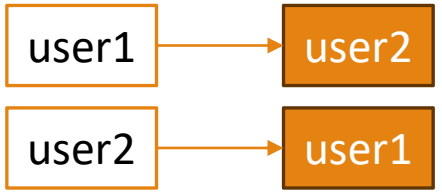
//关联用户与目录
//临时建立并查集
//关联朋友信息
//录入的用户
//朋友圈数量

uf:



父	根(-N)	父	父	根(-M)
1	2	3	4	5

friends:



1. UFSets<string> uf(allUsers.data(), N) (并查集)

方法	说明
uf.Differ(user1, user2)	检查 user1 和 user2 是否属于不同的朋友圈。
uf.WeightedUnion(user1, user2)	合并 user1 和 user2 的朋友圈。
uf.Find(user)	查找 user 的根节点（代表朋友圈）。

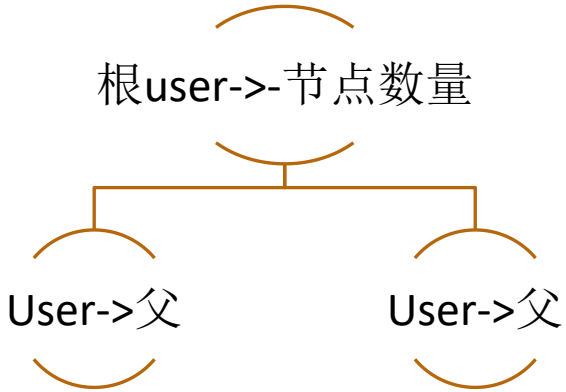
使用UFSets并查集高效地合并用户的朋友圈（连通分量），并快速查询两个用户是否属于同一个朋友圈。

uf.Find(user)

```
template <class ElemType>
int UFSets<ElemType>::Find(ElemType e) const
// 操作结果：查找元素所在集合的根节点（带删除检查）
{
        int p = GetOrder(e);                    // 获取元素在数组中的索引位置
        if (p == -1 || deleted[p]) return -1;     // 有效性检查：元素不存在或已被删除

        // 沿着父节点指针向上查找根节点
        while (sets[p].parent >= 0) {           // 当当前节点不是根节点时继续查找
                p = sets[p].parent;                 // 移动到父节点
        }
        return p;                               // 返回最终找到的根节点索引
}
```

uf.WeightedUnion(user1, user2)

```
template <class ElemType>
void UFSets<ElemType>::WeightedUnion(ElemType a, ElemType b)
// 操作结果：根据结点多少合并a与b所在的等价类
{
        int r1 = Find(a);                                   // 查找a所在等价类的根
        int r2 = Find(b);                                   // 查找b所在等价类的根
        if (r1 != r2 && r1 != -1) {
                int newSize = -(sets[r1].parent + sets[r2].parent);
                if (newSize > maxSize) maxSize = newSize;
                int  temp = sets[r1].parent + sets[r2].parent;
                if (sets[r1].parent <= sets[r2].parent) {
                        sets[r2].parent = r1;
                        sets[r1].parent = temp;
                }
                else {
                        sets[r1].parent = r2;      //r1中的结点个数少，r1指向r2
                        sets[r2].parent = temp;
                }
        }
}
```



2. map<string, set<string>> friends (好友关系映射)

方法	说明
friends[user1].insert(user2)	添加 user2 到 user1 的好友列表。
friends[user1].count(user2)	检查 user2 是否是 user1 的好友。
friends[user1].size()	获取 user1 的好友数量。
friends[user1].empty()	检查 user1 是否存在好友。

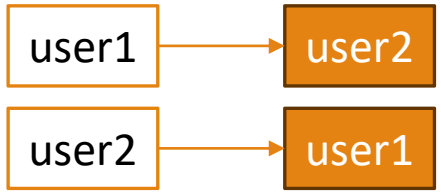
Map存储用户的好友列表：每个用户（string）对应一个集合（set<string>），存储其直接好友。
可以通过 friends[user] 直接获取某个用户的所有好友。

3. set<string> usedUsers（已使用用户集合）

操作	说明
usedUsers.insert(user)	尝试添加用户，返回 pair<iterator, bool>（bool 表示是否为新用户）。
usedUsers.count(user)	检查用户是否已存在。
usedUsers.size()	获取已记录的用户数量。

C++ 标准库中的 <set> 是一个关联容器，它存储了一组唯一的元素，并按照一定的顺序进行排序。Set 自动去重，确保不会重复记录同一对好友关系，避免重复初始化。每个新用户首次出现时，circleCount++（初始时每个用户属于独立的朋友圈）。

数据录入

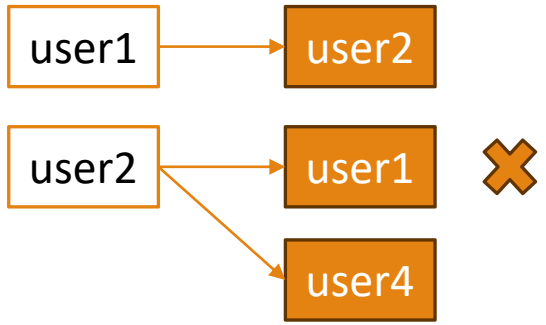



```
// 处理单用户
if (user2.empty()) {
        if (userToIndex.count(user1)) {
                if (usedUsers.insert(user1).second) {
                        circleCount++;
                }
        }
        else {
                cerr << "无效用户: " << user1 << endl;
        }
        continue;
}

// 处理用户对
const bool valid1 = userToIndex.count(user1);
const bool valid2 = userToIndex.count(user2);
if (!valid1 || !valid2) {
        cerr << "无效用户对: " << user1 << " " << user2
<< endl;
        continue;
}
```

```
// 更新已用用户
bool inserted = usedUsers.insert(user1).second;
circleCount += inserted ? 1 : 0;
inserted = usedUsers.insert(user2).second;
circleCount += inserted ? 1 : 0;

// 建立关系
if (!friends[user1].count(user2)) {
        friends[user1].insert(user2);//对于朋友关系
        friends[user2].insert(user1);//进行双边录入

        if (uf.Differ(user1, user2)) {
                circleCount--;//每合并一个减少1
        }
        uf.WeightedUnion(user1, user2);
}
```

保存信息到输出文件

```
ofstream fout("output.txt");      // 输出文件
fout << usedUsers.size() << endl;  // 输出用户数量
for (const string& user : usedUsers) {
        for (const string& friend_ : friends[user]) {
                if (user < friend_) {
                        fout << user << " " << friend_ << endl;
                }                    // 输出好友关系
        }
}
for (const string& user : usedUsers) {
        if (friends[user].empty()) {
                fout << user << endl;      // 输出没有好友的用户
        }
}
fout.close();
```



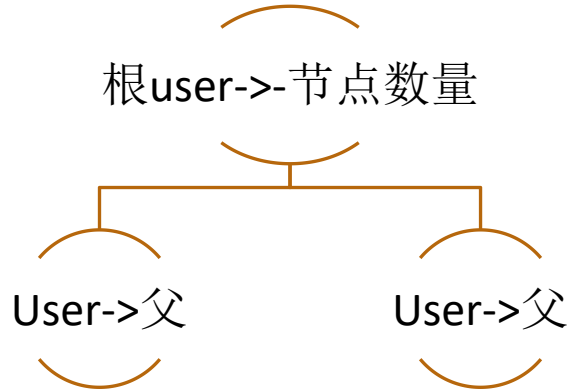
基础代码结构

1、检查两个用户是否是好友

```
string u1, u2;
cout << "请输入两个用户：";
cin >> u1 >> u2;

// 检查用户是否存在
if (usedUsers.count(u1) == 0 || usedUsers.count(u2) == 0) {
        cout << "一个或两个用户不存在。" << endl;
        cout << endl;
        break;
}

// 检查直接好友关系
if (friends[u1].count(u2) || friends[u2].count(u1)) {
        cout << "他们是直接好友。" << endl;
}
else {
        cout << "他们不是直接好友。" << endl;
}
cout << endl;
break;
```

2、检查两个用户是否可间接分享

```
// 检查用户是否存在
if (usedUsers.count(u1) == 0 || usedUsers.count(u2) == 0) {
        cout << "一个或两个用户不存在。" << endl;
        cout << endl;
        break;
}

// 检查是否在同一个朋友圈
if (!uf.Differ(u1, u2)) {
        if (friends[u1].count(u2)) {
                cout << "他们可以分享状态。" << endl;
        }
        else {
                cout << "他们可以间接分享状态。" << endl;
        }
}
else {
        cout << "他们不能分享状态。" << endl;
}
cout << endl;
break;
```



3、输出朋友圈数量

```
cout << "朋友圈数量： " << circleCount << endl;
cout << endl;
break;
```

4、输出最大朋友圈大小

```
cout << "最大朋友圈大小：" << uf.GetCurrentMaxSize() << endl;
cout << endl;
break;

template <class ElemType>
int UFSets<ElemType>::GetCurrentMaxSize() {
        maxSize = 0;
        for (int i = 0; i < size; i++) {
                if (!deleted[i] && sets[i].parent < 0) {
                        maxSize = max(maxSize, -sets[i].parent);
                }
        }
        return maxSize;
}
```

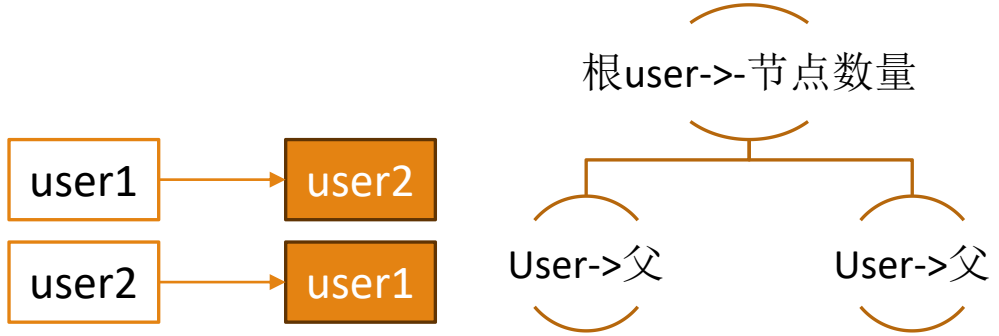
父	根(-N)	父	父	根(-M)
1	2	3	4	5

5、添加新用户

```
string newUser;
cout << "请输入新用户名：";
cin >> newUser;
if (userToIndex.find(newUser) == userToIndex.end()) {
        cout << "用户名不在u1到u" << N << "范围内" << endl;
}
else {
        auto [it, inserted] = usedUsers.insert(newUser);
        if (inserted) {
                circleCount++; // 新用户增加朋友圈计数
                cout << "用户已添加。" << endl;
        }
        else {
                cout << "用户已存在。" << endl;
        }
}
cout << endl;
break;
```

user40

6、添加朋友关系

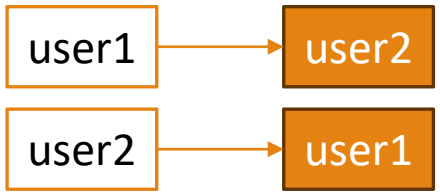


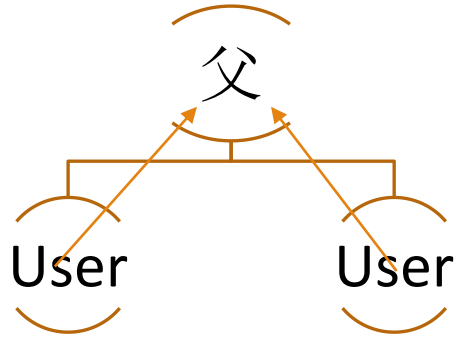

```
string u1, u2;
cout << "请输入两个用户：";
cin >> u1 >> u2;
if (usedUsers.count(u1) == 0 ||
usedUsers.count(u2) == 0) {
cout << "一个或两个用户不存在。" << endl;
}
else if (friends[u1].count(u2)) {
        cout << "他们已经是好友。" << endl;
}
else {
        friends[u1].insert(u2);
        friends[u2].insert(u1);
        if (uf.Differ(u1, u2)) {
                circleCount--;
// 不同朋友圈合并，减少计数
        }
        uf.WeightedUnion(u1, u2);
        cout << "朋友关系已添加。" << endl;
}
cout << endl;
break;
```

7、查找两个用户之间的连接路径

```
string u1, u2;
cout << "请输入两个用户：";
cin >> u1 >> u2;
if (usedUsers.count(u1) == 0 ||
usedUsers.count(u2) == 0) {
        cout << "一个或两个用户不存在。" <<
endl;
}
else if (friends[u1].count(u2)) {
        cout << "他们是直接好友。" << endl;
}
else if (uf.Differ(u1, u2)) {
        cout << "他们不在同一个朋友圈。" <<
endl;
}
```

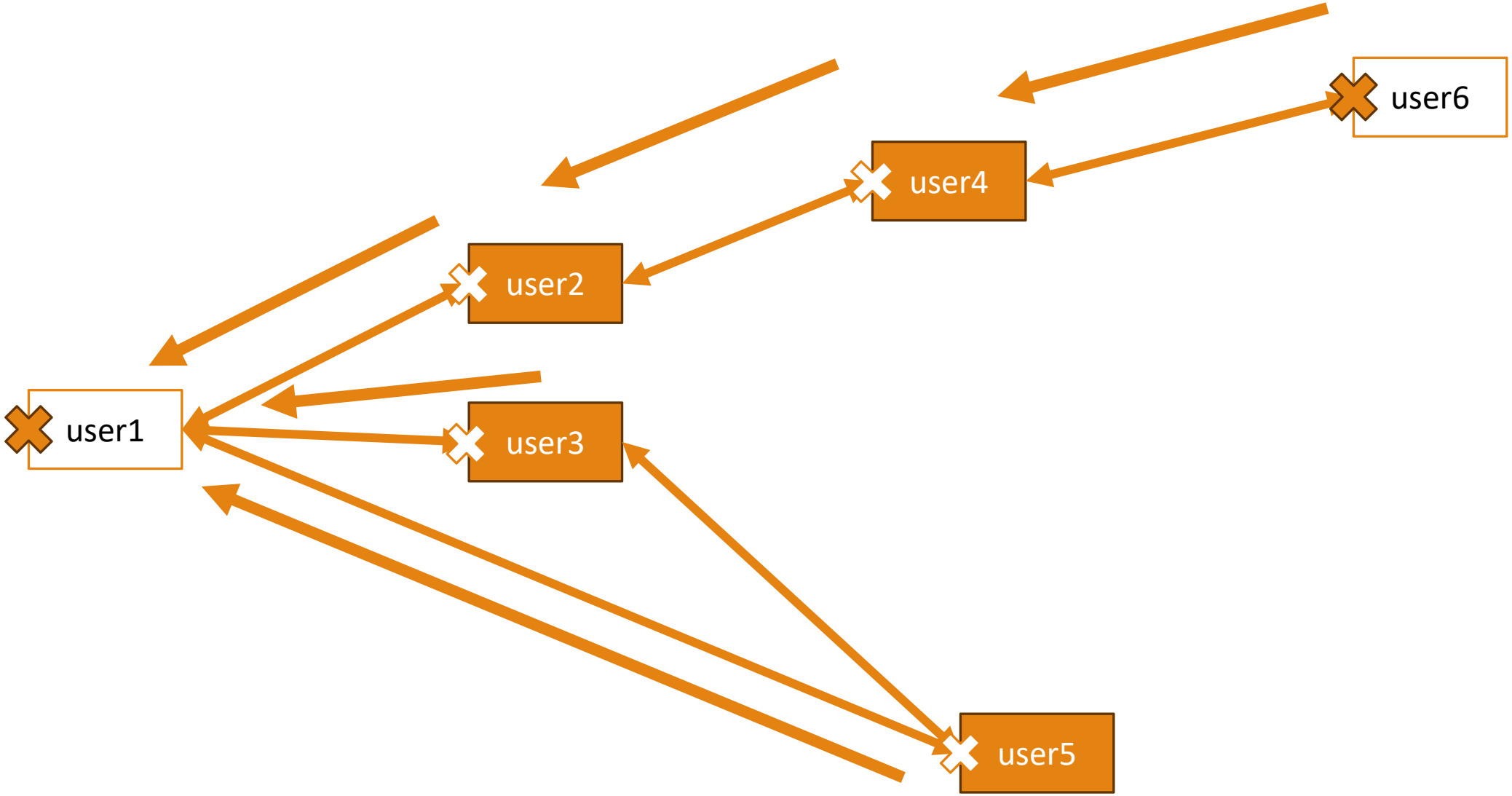
```
else {
        queue<string> q;                    // BFS 队列
        map<string, string> parent;         // 存储路径
        set<string> visited;                // 访问过的用户
        q.push(u1);                         // 从 u1 开始 BFS
        visited.insert(u1);                 // 标记 u1 为已访问
        parent[u1] = "";                    // u1 的父节点为空
        bool found = false;                 // 是否找到路径
```





7、查找两个用户之间的连接路径

```
while (!q.empty() && !found) {
                string current = q.front();       // 当前用户
                q.pop();
                for (const string& friend_ : friends[current]) {
                        if (visited.count(friend_) == 0) {      //未被访问过
                                visited.insert(friend_);   // 标记为已访问
                                parent[friend_] = current;  // 记录父节点
                                q.push(friend_);          // 将好友加入队列
                                if (friend_ == u2) {
                                        found = true;         // 找到目标用户
                                        break;
                                }
                        }
                }
        }
```


user6
user4
user2
user1
user3
user5

7、查找两个用户之间的连接路径

```
if (found) {
        vector<string> path;            // 存储路径
        string current = u2;            // 从 u2 开始回溯路径
        while (current != "") {
                path.push_back(current); // 将当前用户加入路径
                current = parent[current];  // 回溯到父节点
        }
        reverse(path.begin(), path.end());  // 反转路径
        cout << "连接路径： "; for (size_t i = 0; i < path.size(); i++) {
                cout << path[i];            // 输出朋友路径
                if (i < path.size() - 1) cout << " -> ";
        }
        cout << endl;
```

```
if (path.size() > 2) {
        cout << "中间用户： ";
        for (size_t i = 1; i < path.size() - 1; i++) {
                cout << path[i] << " ";
                // 输出中间用户
        }
        cout << endl;
    }
    else {
        cout << "他们直接连接。" << endl;
    }
}
```

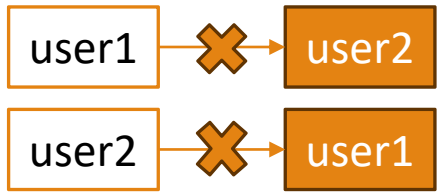
挑战性问题

（1）取消两个用户的好友关系（展示朋友圈分裂）。

（2）删除一个用户。

1、取消两个用户的好友关系

```
string u1, u2;
cout << "请输入要取消好友关系的两个用户：";
cin >> u1 >> u2;
cin.ignore(numeric_limits<streamsize>::max(), '\n');
//验证输入数据
if (friends[u1].count(u2)) {
        friends[u1].erase(u2);
        friends[u2].erase(u1);
        uf.RemoveFriendship(u1, u2);

        // 删除关系后重新构建并查集
        rebuildUFSets(uf, usedUsers, friends,
circleCount);

        cout << "好友关系已取消。" << endl;
}
else {
        cout << "他们不是好友。" << endl;
}
cout << endl;
break;
```



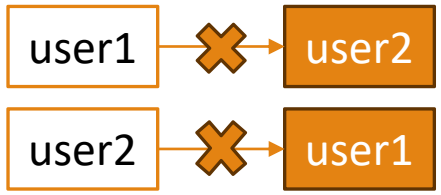
template <class ElemType> void UFSets<ElemType>::RemoveFriendship(ElemType a, ElemType b) 移除两个用户的好友关系

```
int aIdx = GetOrder(a);
int bIdx = GetOrder(b);
// 有效性检查：任一用户不存在或已不在同一集合
if (aIdx == -1 || bIdx == -1 || deleted[aIdx] ||
deleted[bIdx] || Differ(a, b)) return;
// 收集当前集合所有成员
vector<int> members;
int root = Find(a); // 获取当前集合根节点
for (int i = 0; i < size; ++i) {
        // 通过遍历查找所有属于该集合的成员
        if (Find(sets[i].data) == root) {
                members.push_back(i);
        }
}
// 解散当前集合
for (int i : members) {
        sets[i].parent = -1; // 重置为独立节点
}
```

```
// 重建集合（排除a-b直接连接）
for (int i : members) {
        for (int j : members) {
                // 跳过直接连接的a-b和b-a组合
                if (i == aIdx && j == bIdx) continue;
                if (i == bIdx && j == aIdx) continue;
                // 跳过数据层面的a-b配对
                if (sets[i].data == a && sets[j].data == b)
                    continue;
                if (sets[i].data == b && sets[j].data == a)
                    continue;
                // 合并非直接连接的节点
                if (Find(sets[i].data) != Find(sets[j].data))
{
                WeightedUnion(sets[i].data, sets[j].data);
                }
        }
}
```

父	根(-N)	父	父	父
1	2	3	4	5

2、删除用户




```
cin >> user;
cin.ignore(numeric_limits<streamsize>::max
(), '\n');

if (user.empty()) {
        cout << "用户名不能为空。" << endl;
        continue;
}

if (!usedUsers.count(user)) {
        cout << "用户不存在。" << endl;
        continue;
}

// 执行删除
usedUsers.erase(user);
```

```
if (friends.count(user)) {
        for (const auto& friendUser : friends[user]) {
                friends[friendUser].erase(user); //
双向删除
        }
        friends.erase(user);
}

// 并查集删除
uf.DeleteUser(user);

// 删除用户后重新构建并查集
rebuildUFSets(uf, usedUsers, friends, circleCount);

cout << "用户 " << user << " 已成功删除。" << endl;
cout << endl;
```

uf.DeleteUser(user)

```
template <class ElemType>
void UFSets<ElemType>::DeleteUser(ElemType e){
int idx = GetOrder(e);
if (idx == -1) return; // 用户不存在

// 标记为已删除
deleted[idx] = true;

// 特殊处理树形结构（树的根节点）：
sets[idx].parent = INT_MIN; // 特殊标记已删除节点
// 断开所有子节点的连接
for (int i = 0; i < size; ++i) {
        if (sets[i].parent == idx) {
                sets[i].parent = -1; // 子节点变为独立
```

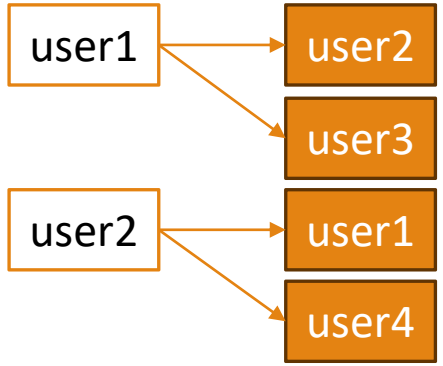
```
根节点
        }
}
}
```

父	-1	父	-1	父
1	2	3	4	5

rebuildUFSets(UFSets<string>& uf, const set<string>& usedUsers, const map<string, set<string>>& friends, int& circleCount)
重新构建关系

```
vector<string> users(usedUsers.begin(), usedUsers.end());
uf = UFSets<string>(users.data(), users.size());//新建并查集
circleCount = users.size();//新建好友圈计数器

set<pair<string, string>> processedPairs; // 避免重复处理关系对
for (const auto& user : users) {
        if (!friends.count(user)) continue;
        for (const string& friend_ : friends.at(user)) {
                // 确保好友存在于当前用户列表且关系未被处理
                if (usedUsers.count(friend_) && processedPairs.find({ friend_, user }) == processedPairs.end()) {
                        if (uf.Differ(user, friend_)) {
                                circleCount--;
                                uf.WeightedUnion(user, friend_);
                        }
                        processedPairs.insert({ user, friend_ });
                }
        }
}
```

-1	-1	-1	-1	-1
1	2	3	4	5



谢谢大家