🡪

并查集(disjoint set)又称之为（不相交集）是简洁优雅的数据结构之一，主要用于解决一些元素分组的问题。这种数据结构用于包括一系列不相交的集合。这些不相交的集合中，每个集合含有若干个元素。

并查集的运行原则基于树数据结构。在并查集中的每一个不相交集合都有一棵树代表。每一个集合都会选出一个代表用于代表该集合（该代表一般都是该集合内的元素）。一般选取代表该集合的树的根节点。

由于在并查集中我们不需要知道某节点的子元素或者进行遍历树，所以在并查集具体代码实现中，不需要实现完整的树结构，而是经常使用一维数组用于记录树节点的父节点便足够。

🡪

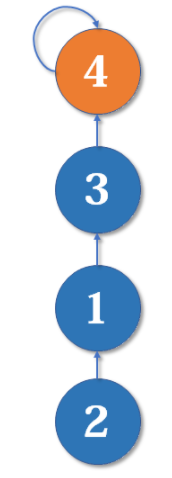
并查集通常包含两种最常见的操作,find以及Union(或者称为merge)

Find查询操作：用于查找某元素所在集合的代表元素，也即是查找某个元素所在集合的树结构中根节点。如果两个元素所在集合的代表元素相同，也就是树结构中根节点相同，也就是find查询结果相同，则认为这两个元素处于同一个集合当中。

Union(Merge)合并操作：将两个不相交的集合合并为同一个集合。对应于每个集合背后的树结构，则代表将合并中其中一个集合的树作为子树合并到另一集合的树中。

在合并中，也就是进行集合背后两棵树的合并当中，还需要注意如何合并两颗树的问题。在基础的并查集合并操作中，规则只是简单的后者集合合并到前者集合的树作为子树，则合并后集合所代表的树结构有可能呈现出链式形状。这时如果进行Find查询操作，则需要向上遍历多层才可以找到根节点（也就是集合的代表元素）。时间耗费高。

例如：



基于此，由于并查集只关心一个元素所在的集合的根节点（也就是该元素所在集合的代表元素），那我们希望每个元素到根节点的路径尽可能短，我们可以使用**路径压缩(Path Compression)**以及**按度合并**（Union By Rank）的方法。

**路径压缩**方法的目的在于尽可能地让集合背后的树结构深度尽可能的小。我们在Find查询操作中进行压缩。

例如：

// 下面为带有路径压缩功能的Find函数，传入的参数i为某集合的元素，需要返回该元素所在集合的代表元素

int find(int i)

{

// 如果该元素的父节点是自身，则自身便是该集合的代表元素

if (Parent[i] == i)

{

return i;

}

else

{

// 如果该元素的父节点不是自身，则是该集合树上的非根节点。使用父节点递归查找，沿着树向上遍历寻找根节点。递归调用用返回result,即根节点，也就是该元素所在集合的代表元素。

int result = find(Parent[i]);

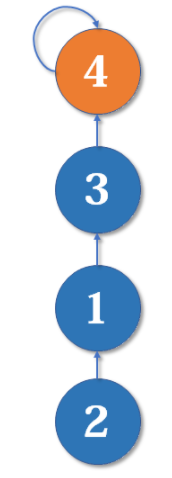
Parent[i] = result; # 路径压缩实现。由于是递归调用，所以对树中路径上的所有非根节点都将执行。它将获得的result, 即根节点，也就是该元素所在集合的代表元素，设置为自身的父节点。从而该条路径上所有非根节点在执行Find操作后，都变为根节点的子节点。

return result;

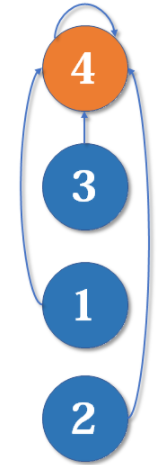
}

}

路径压缩方法执行前：



对节点2执行带有路径压缩的Find方法后，结点2，结点1，结点3，的find方法获得结点为结点4. 并且我们将结点4设置为结点2，结点1，结点3的父节点。则该集合的树结构变为：



压缩以后树的层数变为2层，在以后的再对路径上执行Find操作时候，节省时间。

需要注意的是对并查集的Find方法加入带有路径压缩特性的Find方法后，并查集中各个集合**并不**始终都是一个菊花图（只有两层的树的俗称）。由于路径压缩只在查询时进行，也只压缩一条路径，所以并查集最终的树结构仍然可能是比较复杂的。

**按度合并**（Union By Rank）

按度合并发生在两个集合的合并阶段。当合并时候，原则是我们将代表集合的两颗树中高度较矮的树，作为子树合并到另一棵树中，这样合并后的集合树高度不会发生变化。因为我们在记录集合树的高度，所以一般会再添加一个数组rank，用于记录所有元素所对应的集合树的深度。(rank[i]为元素i作为根结点，其子树的深度。如果元素i为根节点，也就是所在集合的代表元素，则rank[i]变为该集合树的深度)

例如：

//按度合并。i为元素，j为元素。要求执行完union方法后，i元素所在的集合和j元素所在的集合合并。在合并过程中采用按度合并。

void union(int i, int j)

{

//首先找出i元素所在集合树的根节点（也就是i元素所在集合的代表元素）

int irep = this.find(i);

//再找出j元素所在集合树的根节点（也就是j元素所在集合的代表元素）

int jrep = this.Find(j);

//如果i元素所在集合的代表元素和j元素所在集合的代表元素相同。则元素i，j处于同一集合无需合并。

if (irep == jrep)

return;

//获取i元素所在集合的树的深度

irank = Rank[irep],

//获取j元素所在集合的树的深度

jrank = Rank[jrep];

//如果i元素所在集合的树的深度小于j元素所在集合的树的深度，则i元素所在集合的树为矮树。将作为子树合并到另一个j元素所属于集合的高树中

if (irank < jrank)

{

//指定i元素所在集合树的根节点的父节点为j元素所在集合的代表元素。由于只是将一颗矮树作为新增的子树合并到树种。所以合并后的树高度与原来二者中的高树高度相同。

this.parent[irep] = jrep;

}

//如果j元素所在集合的树的深度小于i元素所在集合的树的深度，则j元素所在集合的树为矮树。将作为子树合并到另一个i元素所属于集合的高树中

else if (jrank < irank)

{

//指定j元素所在集合树的根节点的父节点为i元素所在集合的代表元素。由于只是将一颗矮树作为新增的子树合并到树种。所以合并后的树高度与原来二者中的高树高度相同。

this.Parent[jrep] = irep;

}

//如果合并中的两颗子树高度相同

else

{

//如果合并中的两颗子树高度相同，则没有明显的高低树之分。则将任意一个棵树作为子树合并到另一颗树中都可以。这里我们选择将i元素所在集合的树充当子树合并到j元素所在集合的树中。

this.Parent[irep] = jrep;

//两棵深度相同的树合并后高度将增加1

Rank[jrep]++;

}

}

P.s 路径压缩以及按度合并在单独使用下都可以降低并查集的时间复杂度。在实际使用中，我们可以结合使用，从而在合并时候减少树的深度，在查询的时候将路径上的结点所在子树分支变为深度为2的树结构。需要注意的是如果两者同时时候，则度的计数会发生失真现象。意思是rank[i]值可能不是真正的树的深度，原因在于find方法路径压缩时候改变了树的结构。

🡪

并查集的应用有最小生成树的Kruskal算法，判断人物是否亲戚关系等。

🡪

并查集中存有父节点的parent数组，以及存有度的rank数组都需要使用下标index进行访问。而在使用并查集解决实际问题中时，例如判断人物关系，将会是类似于人名”A”, “B”，不能直接使用这些元素名访问parent以及rank数组。我们需要进行一定的数据抽象，并且认为parent数组以及rank数组的下标指的是第几个元素。

例如在判别人物是否为亲戚关系时候，将名为”A”人物看作下标0号元素，则寻找该人物所在集合的代表元素，集合树的高度时候，使用parent[0]以及rank[0].

例如在Kruskal算法中，结点V0看作下标0号元素，则寻找该结点所在集合的代表元素，集合树的高度时候，使用parent[0]以及rank[0].