并查集

什么是并查集（Disjoint-set）

对于一个集合S={a1, a2, ..., an-1, an}，我们还可以对集合S进一步划分: S1,S2,...,Sm-1,Sm，我们希望能够快速确定S中的两两元素是否属于S的同一子集。

举个栗子，S={0，1, 2, 3, 4, 5, 6}，如果我们按照一定的规则对集合S进行划分,假设划分后为S1={1, 2, 4}, S2={3, 6}，S3={0, 5}，任意给定两个元素，我们如何确定它们是否属于同一子集？某些合并子集后，又如何确定两两关系？基于此类问题便出现了并查集这种数据结构。

并查集有两个基本操作：

Find: 查找元素所属子集

Union，合并两个子集为一个新的集合

并查集的基本结构

我们可以使用树这种数据结构来表示集合，不同的树就是不同的集合，并查集中包含了多棵树，表示并查集中不同的子集，树的集合是森林，所以并查集属于森林。

若集合S={0, 1, 2, 3, 4, 5, 6}，最初每一个元素都是一棵树。

对于Union操作，我们只需要将两棵树合并，例如合并0、1、2得到S1={0, 1, 2},合并3和4得到S2={3, 4}

对于Find操作，我们只需要返回该元素所在树的根节点。所以，如果我们想要比较判断1和2是否在一个集合，只需要通过Find(1)和Find(2)返回各自的根节点比较是否相等便可。已知树中的一个节点，找到其根节点的时间复杂度为O(D)，D为节点的深度。 我们可以使用数组来表示树，数组下标表示树的一个节点，该下表所对应的值表示树的父节点。例如P[i]表示元素i的父节点。对于图2中的集合，我们可以存储在下面的数组中（第二行为数组下标）

0 0 0 3 3 5 6 0 1 2 3 4 5 6

对于树的根节点，我们规定其元素值为其本身（即父节点为自己）。

我们使用一个parent数组存储树，先实现未经优化的版本。

对于Find操作，代码非常简单

```
int find(int x)
{
    return parent[x] == x ? x : find(parent[x]);
}
```

该代码比较元素x的父节点parent[x]是否等于x自身，如果是便说明找到了根节点（根节点的父节点是自身），直接返回；否则，把x的父节点parent[x]传入find，直到找到根节点。

下面是union操作

```
void to_union(int x1, int x2)
{
    int p1 = find(x1);
```

```
    int p2 = find(x2);
    parent[p1] = p2;
}
```

传入两个元素，分别找到根节点，使根节点p1的父节点为p2，即将p1为根节点的这棵树合并到p2为根节点的树上。

下面是完整代码：

```
#include <vector>
class DisjSet
{
private:
    std::vector<int> parent;

public:
    DisjSet(int max_size) : parent(std::vector<int>(max_size))
    {
        //初始化每一个元素的根节点都为自身
        for (int i = 0; i < max_size; ++i)
            parent[i] = i;
    }
    int find(int x)
    {
        return parent[x] == x ? x : find(parent[x]);
    }
    void to_union(int x1, int x2)
    {
        parent[find(x1)] = find(x2);
    }
    //判断两个元素是否属于同一个集合
    bool is_same(int e1, int e2)
    {
        return find(e1) == find(e2);
    }
};
```

上面的实现，可以看出每一次Find操作的时间复杂度为O(H)，H为树的高度，由于我们没有对树做特殊处理，所以树的不断合并可能会使树严重不平衡，最坏情况每个节点都只有一个子节点