数据结构与算法 12- 并查集UnionFind

笔记本:	我的笔记		
创建时间:	2020/10/9 23:01	**更新时间:**	2020/10/9 23:13
作者:	liuhouer	**位置:**	39°50'3 N 116°19'11 E
标签:	算法		

1.结构-子节点指针指向父节点

2.适应场景-查找2个节点的连接关系【类比查找路径要简单】
【迷宫出路】

3.设计接口

```
public interface UF {

    int getSize();

    boolean isConnected(int p, int q);

    void unionElements(int p, int q);

}
```

4.第一版Union-Find -数组

```
// 我们的第一版Union-Find
public class UnionFind1 implements UF {


    private int[] id;    // 我们的第一版Union-Find本质就是一个数组


    public UnionFind1(int size) {


        id = new int[size];


        // 初始化, 每一个id[i]指向自己, 没有合并的元素
        for (int i = 0; i < size; i++)
            id[i] = i;
    }


    @Override
    public int getSize(){
        return id.length;
    }


    // 查找元素p所对应的集合编号
    // O(1)复杂度
    private int find(int p) {
```

```
        if(p < 0 || p >= id.length)
            throw new IllegalArgumentException("p is out of bound.");


        return id[p];
    }


    // 查看元素p和元素q是否所属一个集合
    // O(1)复杂度
    @Override
    public boolean isConnected(int p, int q) {
        return find(p) == find(q);
    }


    // 合并元素p和元素q所属的集合
    // O(n) 复杂度
    @Override
    public void unionElements(int p, int q) {


        int pID = find(p);
        int qID = find(q);


        if (pID == qID)
            return;


        // 合并过程需要遍历一遍所有元素, 将两个元素的所属集合编号合并
        for (int i = 0; i < id.length; i++)
            if (id[i] == pID)
                id[i] = qID;
    }
}
```

2.第二版Union-Find -指针树

```
// 我们的第二版Union-Find
public class UnionFind2 implements UF {


    // 我们的第二版Union-Find, 使用一个数组构建一棵指向父节点的树
    // parent[i]表示第一个元素所指向的父节点
    private int[] parent;


    // 构造函数
    public UnionFind2(int size){


        parent = new int[size];


        // 初始化, 每一个parent[i]指向自己, 表示每一个元素自己自成一个集合
        for( int i = 0 ; i < size ; i ++ )
            parent[i] = i;
    }


    @Override
    public int getSize(){
        return parent.length;
    }


    // 查找过程, 查找元素p所对应的集合编号
    // O(h)复杂度, h为树的高度
    private int find(int p){
        if(p < 0 || p >= parent.length)
            throw new IllegalArgumentException("p is out of bound.");
```

```
        // 不断去查询自己的父亲节点, 直到到达根节点
        // 根节点的特点: parent[p] == p
        while(p != parent[p])
            p = parent[p];
        return p;
    }


    // 查看元素p和元素q是否所属一个集合
    // O(h)复杂度, h为树的高度
    @Override
    public boolean isConnected( int p , int q ){
        return find(p) == find(q);
    }


    // 合并元素p和元素q所属的集合
    // O(h)复杂度, h为树的高度
    @Override
    public void unionElements(int p, int q){


        int pRoot = find(p);
        int qRoot = find(q);


        if( pRoot == qRoot )
            return;


        parent[pRoot] = qRoot;
    }
}
```

3.第三版Union-Find –size优化

```
// 我们的第三版Union-Find
public class UnionFind3 implements UF{


    private int[] parent; // parent[i]表示第一个元素所指向的父节点
    private int[] sz;     // sz[i]表示以i为根的集合中元素个数


    // 构造函数
    public UnionFind3(int size){


        parent = new int[size];
        sz = new int[size];


        // 初始化, 每一个parent[i]指向自己, 表示每一个元素自己自成一个集合
        for(int i = 0 ; i < size ; i ++){
            parent[i] = i;
            sz[i] = 1;
        }
    }


    @Override
    public int getSize(){
        return parent.length;
    }


    // 查找过程, 查找元素p所对应的集合编号
    // O(h)复杂度, h为树的高度
    private int find(int p){
        if(p < 0 || p >= parent.length)
            throw new IllegalArgumentException("p is out of bound.");


        // 不断去查询自己的父亲节点, 直到到达根节点
```

```
        // 根节点的特点: parent[p] == p
        while( p != parent[p] )
            p = parent[p];
        return p;
    }


    // 查看元素p和元素q是否所属一个集合
    // O(h)复杂度, h为树的高度
    @Override
    public boolean isConnected( int p , int q ){
        return find(p) == find(q);
    }


    // 合并元素p和元素q所属的集合
    // O(h)复杂度, h为树的高度
    @Override
    public void unionElements(int p, int q){


        int pRoot = find(p);
        int qRoot = find(q);


        if(pRoot == qRoot)
            return;


        // 根据两个元素所在树的元素个数不同判断合并方向
        // 将元素个数少的集合合并到元素个数多的集合上
        if(sz[pRoot] < sz[qRoot]){
            parent[pRoot] = qRoot;
            sz[qRoot] += sz[pRoot];
        }
        else{ // sz[qRoot] <= sz[pRoot]
            parent[qRoot] = pRoot;
            sz[pRoot] += sz[qRoot];
        }
    }
}
```

4.第四版Union-Find -rank层优化

```
// 我们的第四版Union-Find
public class UnionFind4 implements UF {


    private int[] rank;   // rank[i]表示以i为根的集合所表示的树的层数
    private int[] parent; // parent[i]表示第i个元素所指向的父节点


    // 构造函数
    public UnionFind4(int size){


        rank = new int[size];
        parent = new int[size];


        // 初始化, 每一个parent[i]指向自己, 表示每一个元素自己自成一个集合
        for( int i = 0 ; i < size ; i ++ ){
            parent[i] = i;
            rank[i] = 1;
        }
    }


    @Override
    public int getSize(){
        return parent.length;
    }
```

```
    // 查找过程, 查找元素p所对应的集合编号
    // O(h)复杂度, h为树的高度
    private int find(int p){
        if(p < 0 || p >= parent.length)
            throw new IllegalArgumentException("p is out of bound.");


        // 不断去查询自己的父亲节点, 直到到达根节点
        // 根节点的特点: parent[p] == p
        while(p != parent[p])
            p = parent[p];
        return p;
    }


    // 查看元素p和元素q是否所属一个集合
    // O(h)复杂度, h为树的高度
    @Override
    public boolean isConnected( int p , int q ){
        return find(p) == find(q);
    }


    // 合并元素p和元素q所属的集合
    // O(h)复杂度, h为树的高度
    @Override
    public void unionElements(int p, int q){


        int pRoot = find(p);
        int qRoot = find(q);


        if( pRoot == qRoot )
            return;


        // 根据两个元素所在树的rank不同判断合并方向
        // 将rank低的集合合并到rank高的集合上
        if(rank[pRoot] < rank[qRoot])
            parent[pRoot] = qRoot;
        else if(rank[qRoot] < rank[pRoot])
            parent[qRoot] = pRoot;
        else{ // rank[pRoot] == rank[qRoot]
            parent[pRoot] = qRoot;
            rank[qRoot] += 1;   // 此时, 我维护rank的值
        }
    }
}
```