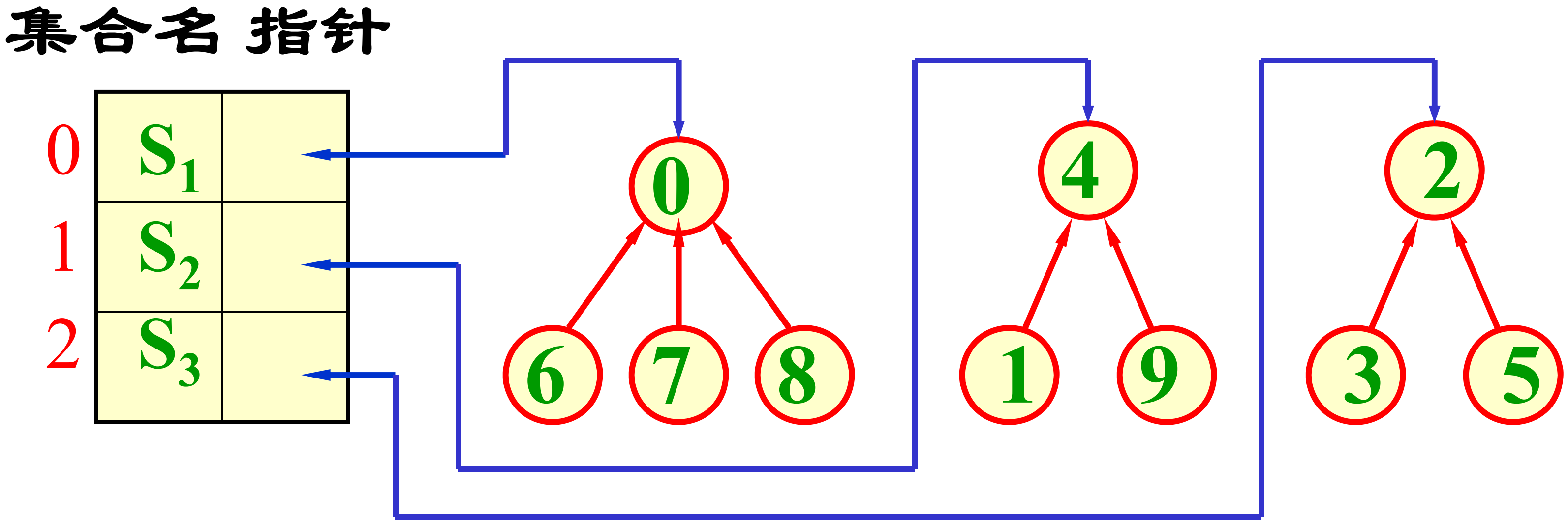
并查集 (Union-Find Sets)

- 建立等价类的另一种解决方案是先把每一个对象看作是一个单元素集合，然后按一定顺序将属于同一等价类的元素所在的集合合并。
- 在此过程中将反复地使用一个搜索运算，确定一个元素在哪一个集合中。

- 能够完成这种功能的集合就是并查集，支持以下三种操作：
  - **Union (Root1, Root2)** //并操作
    - 把子集合 **Root2** 并入集合 **Root1** 中，要求 **Root1** 与 **Root2** 互不相交，否则无结果。
  - **Find (x)** //搜索操作
    - 搜索单元素 **x** 所在的集合，并返回该集合名字。
  - **UFSets (s)** //构造函数
    - 将并查集中的 **s** 个元素初始化为 **s** 个只有一个单元素的子集合，根结点的 **parent** 值等于**-1**。

- 对于并查集来说，每个集合用一棵树表示。
- 为此，采用树的双亲表示作为集合存储表示。集合元素的编号从 **0** 到 **n-1**，其中 **n** 是最大元素个数。
- 在双亲表示中，第 **i** 个数组元素代表包含集合元素 **i** 的树结点。根结点的双亲为 **-1**，表示集合中的元素个数。
- 在同一棵树上所有结点所代表的集合元素在同一个子集合中。
- 为此，需要有两个映射：
  - 集合元素到存放该元素名的树结点间的对应；
  - 集合名到表示该集合的树的根结点间的对应。

- 设 $S_1 = \{ 0, 6, 7, 8 \}, S_2 = \{ 1, 4, 9 \}, S_3 = \{ 2, 3, 5 \}$



- 为简化讨论，忽略实际的集合名，仅用表示集合的树的根来标识集合。

利用并查集来解决等价问题的步骤

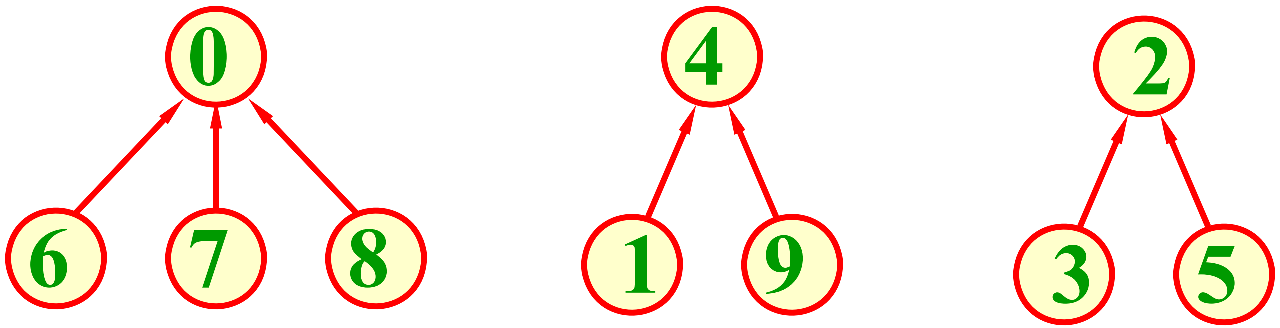
- 初始时，用构造函数 **UFSets(s)** 构造一个森林，每棵树只有一个结点，表示集合中各元素自成一个子集。

下标	0	1	2	3	4	5	6	7	8	9
parent	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1

- 用 **Find(i)** 寻找集合元素 **i** 的根，如果有两个集合元素 **i** 和 **j**，**Find(i) == Find(j)**，表明这两个元素在同一个集合中。
- 如果两个集合元素 **i** 和 **j** 不在同一集合中，可用 **Union(i, j)** 将其合并到一集合中。

下标	0	1	2	3	4	5	6	7	8	9
parent	-4	4	-3	2	-3	2	0	0	0	4

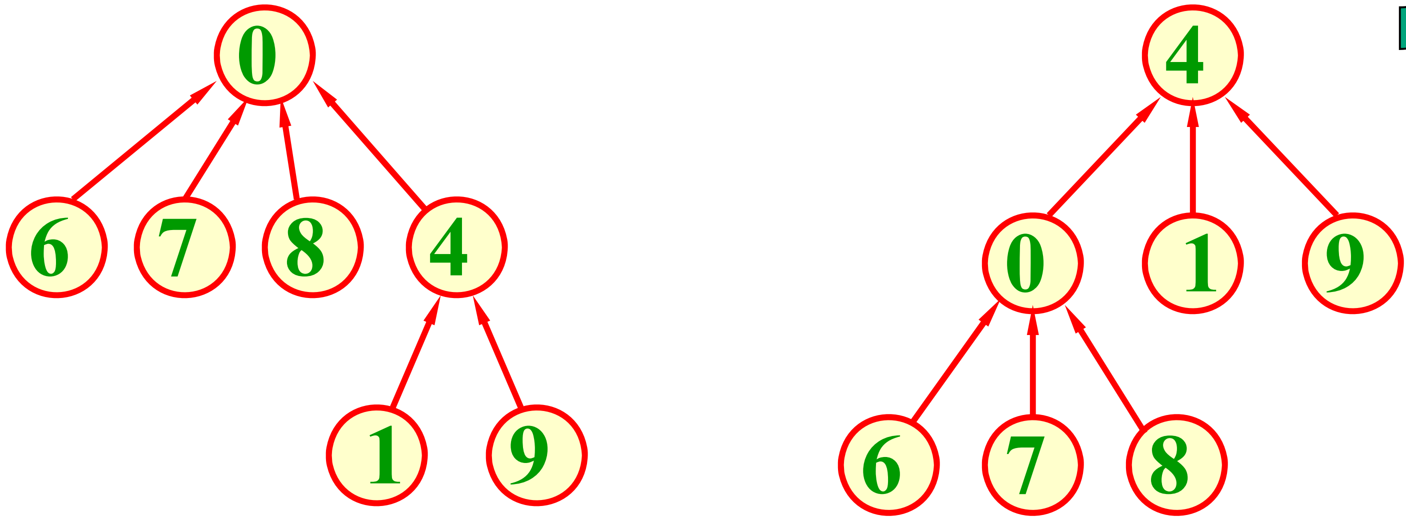
集合 S_1、S_2和 S_3的双亲表示



S_1 S_2 S_3

下标	0	1	2	3	4	5	6	7	8	9
parent	-7	4	-3	2	0	2	0	0	0	4

集合 $S_1 \cup S_2$ 和 S_3 的双亲表示



$S_1 \cup S_2$ 的可能的表示方法

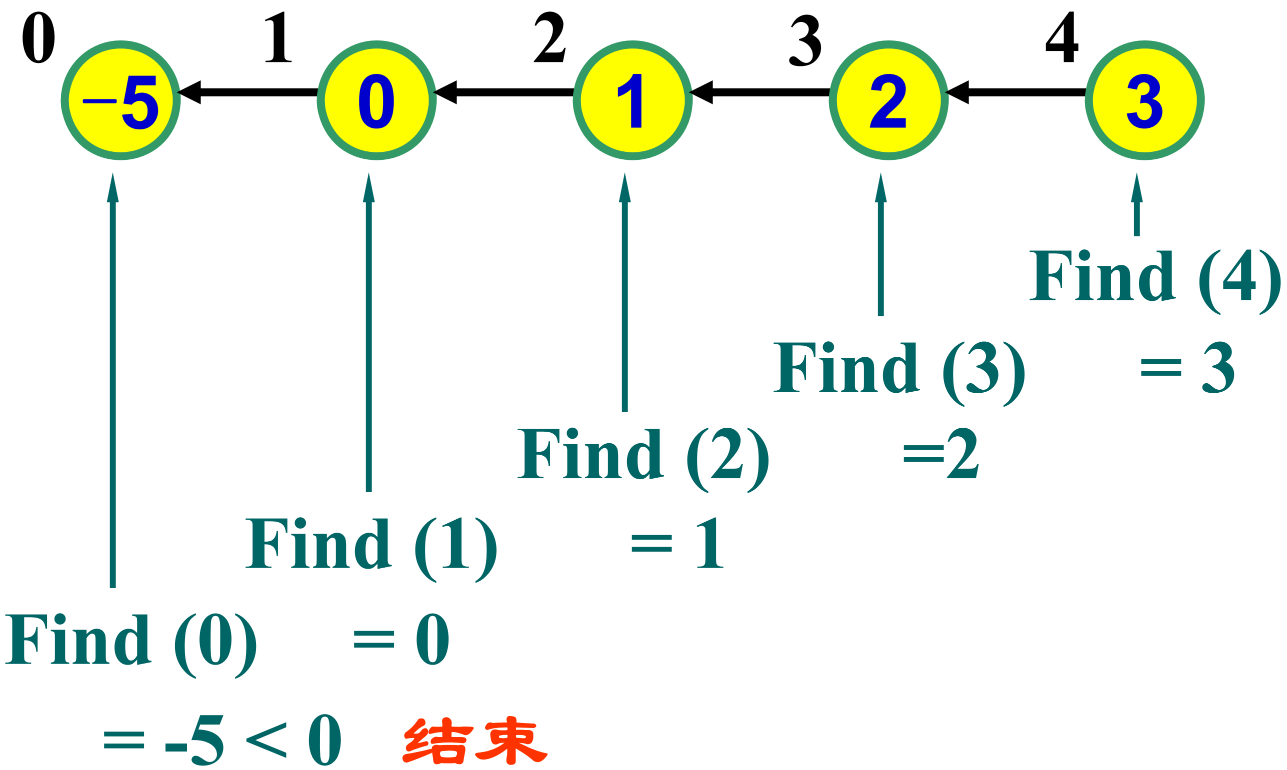
```
const int DefaultSize = 10;
class UFSets {  //并查集类定义
private:
    int *parent;  //集合元素数组
    int size;  //集合元素的数目
public:
    UFSets ( int s = DefaultSize );  //构造函数
    ~UFSets ( ) { delete [ ] parent; } //析构函数
    const UFSets & operator = ( UFSets &R );
    //重载函数：集合赋值
    void Union ( int Root1, int Root2 );
    //基本例程：两个子集合合并
```

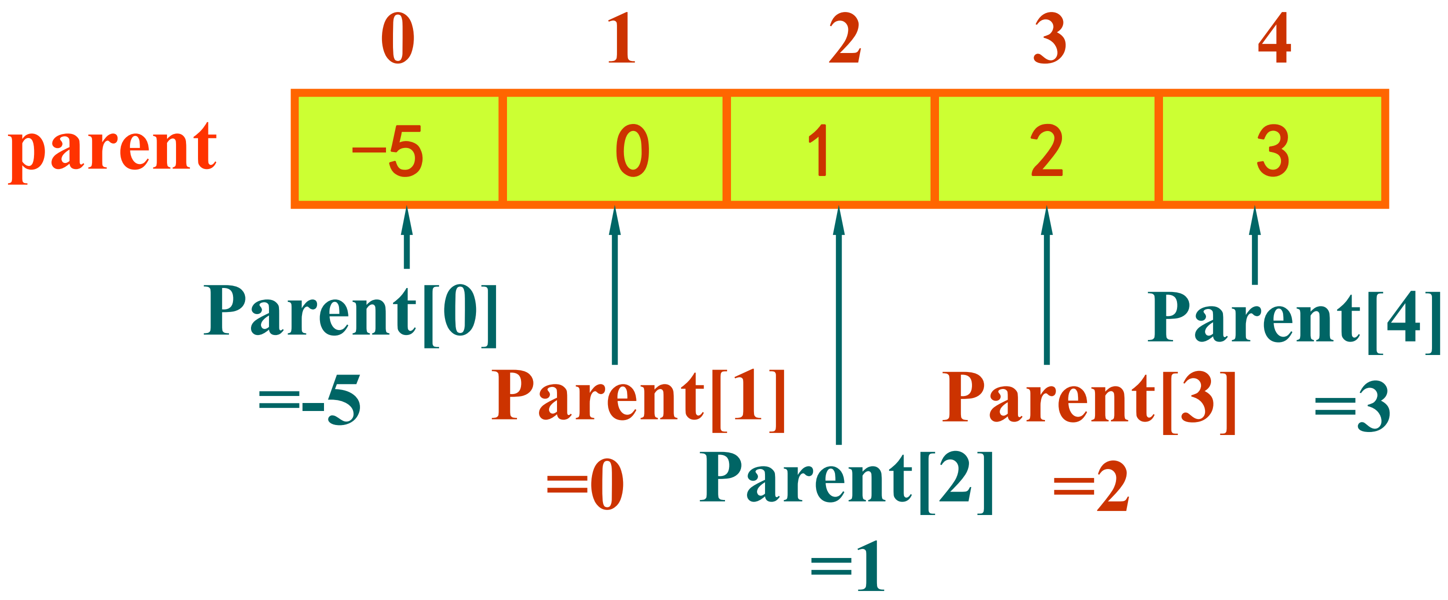
```
    int Find ( int x );
    //基本例程：搜寻集合x的根
    void UnionByHeight ( int Root1, int Root2 );
    //改进例程：压缩高度的合并算法
};

UFSets :: UFSets ( int s ) { //构造函数
    size = s;  //集合元素个数
    parent = new int [size];  //双亲指针数组
    for ( int i = 0; i < size; i++ ) parent[i] = -1;
    //每一个自成一个单元素集合
}
```

并查集操作的算法

- 查找



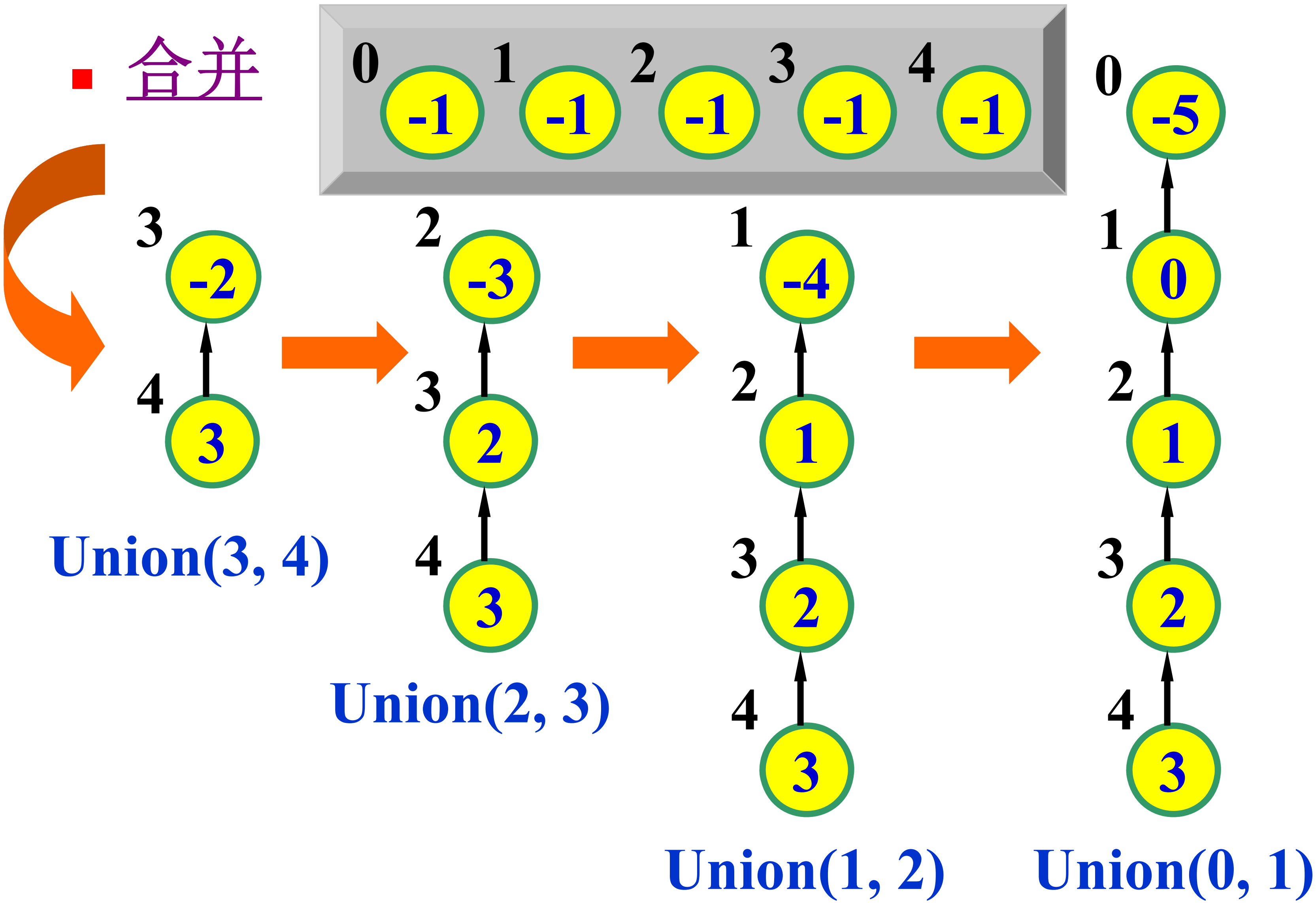



```
int UFSets :: Find ( int x ) {
    if ( parent [x] < 0 ) return x;
    else return Find ( parent [x] );
}
```

```
void UFSets :: Union ( int Root1, int Root2 ) {
//求两个不相交集合 Root1 与 Root2 的并
    parent[Root1] += parent[Root2];
    parent[Root2] = Root1;
    //将 Root2 连接到 Root1 下面
}
```

- **Find**和**Union**操作性能不好。
  - 假设最初 **n** 个元素构成 **n** 棵树组成的森林，**parent[i] = -1**。做处理**Union(n-2, n-1), …, Union(1, 2), Union(0, 1)**后，将产生退化的树。

- 合并


0 1 2 3 4
-1 -1 -1 -1 -1
3 -2
4 3
Union(3, 4)
2 -3
3 2
4 3
Union(2, 3)
1 -4
2 1
3 2
4 3
Union(1, 2)
0 -5
1 0
2 1
3 2
4 3
Union(0, 1)

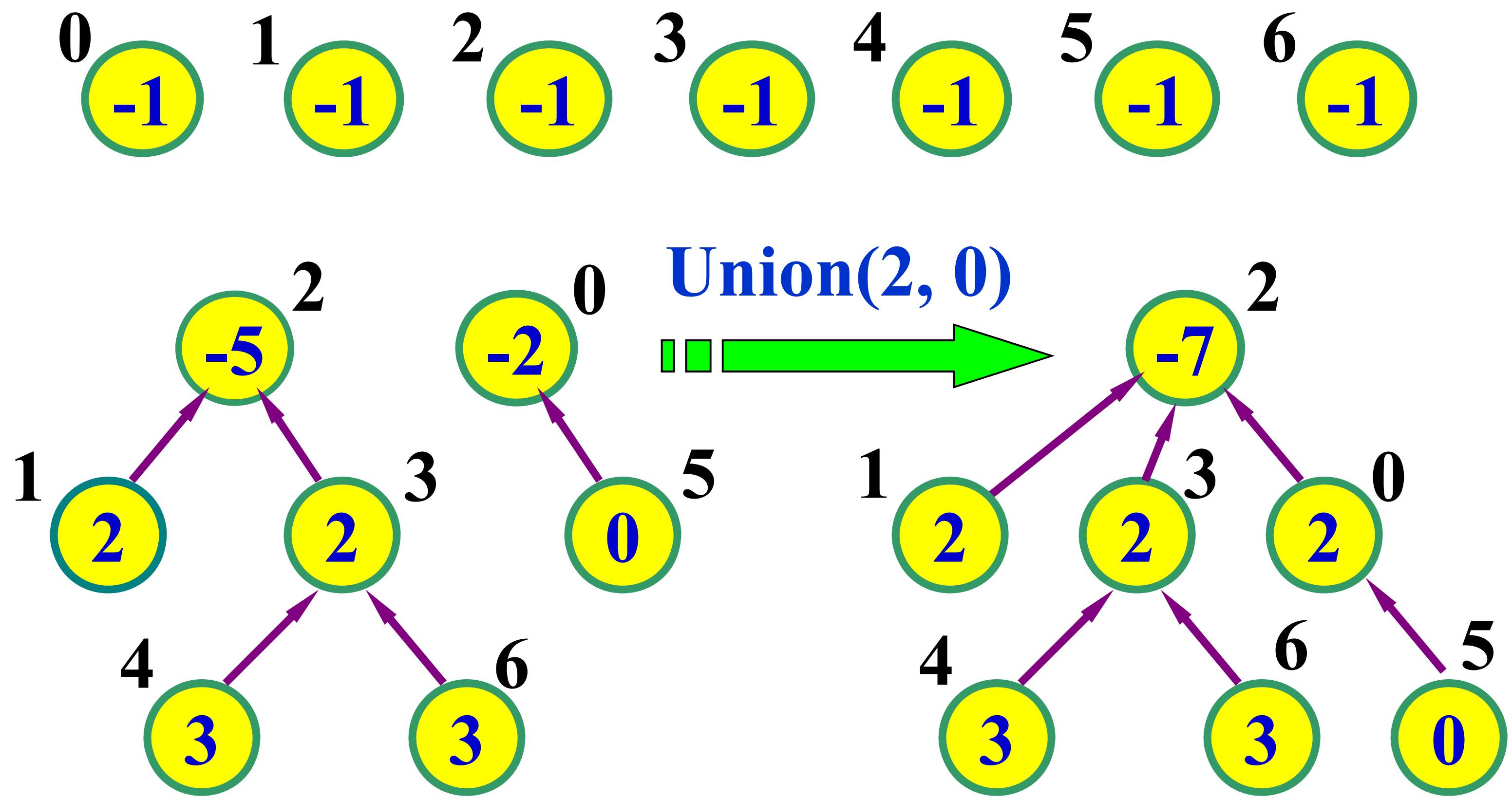
- 执行一次**Union**操作所需时间是**O(1)**，**n-1**次**Union**操作所需时间是**O(n)**。
- 若再执行**Find(0), Find(1), …, Find(n-1)**，若被搜索的元素为**i**，完成**Find(i)**操作需要时间为**O(i)**，完成**n**次搜索需要的总时间将达到

$$\mathbf{O}\left(\sum_{i=1}^{n} i\right) = \mathbf{O}(n^2)$$

- **改进方法**
  - 按树的结点个数合并
  - 按树的高度合并
  - 压缩元素的路径长度

按树结点个数合并

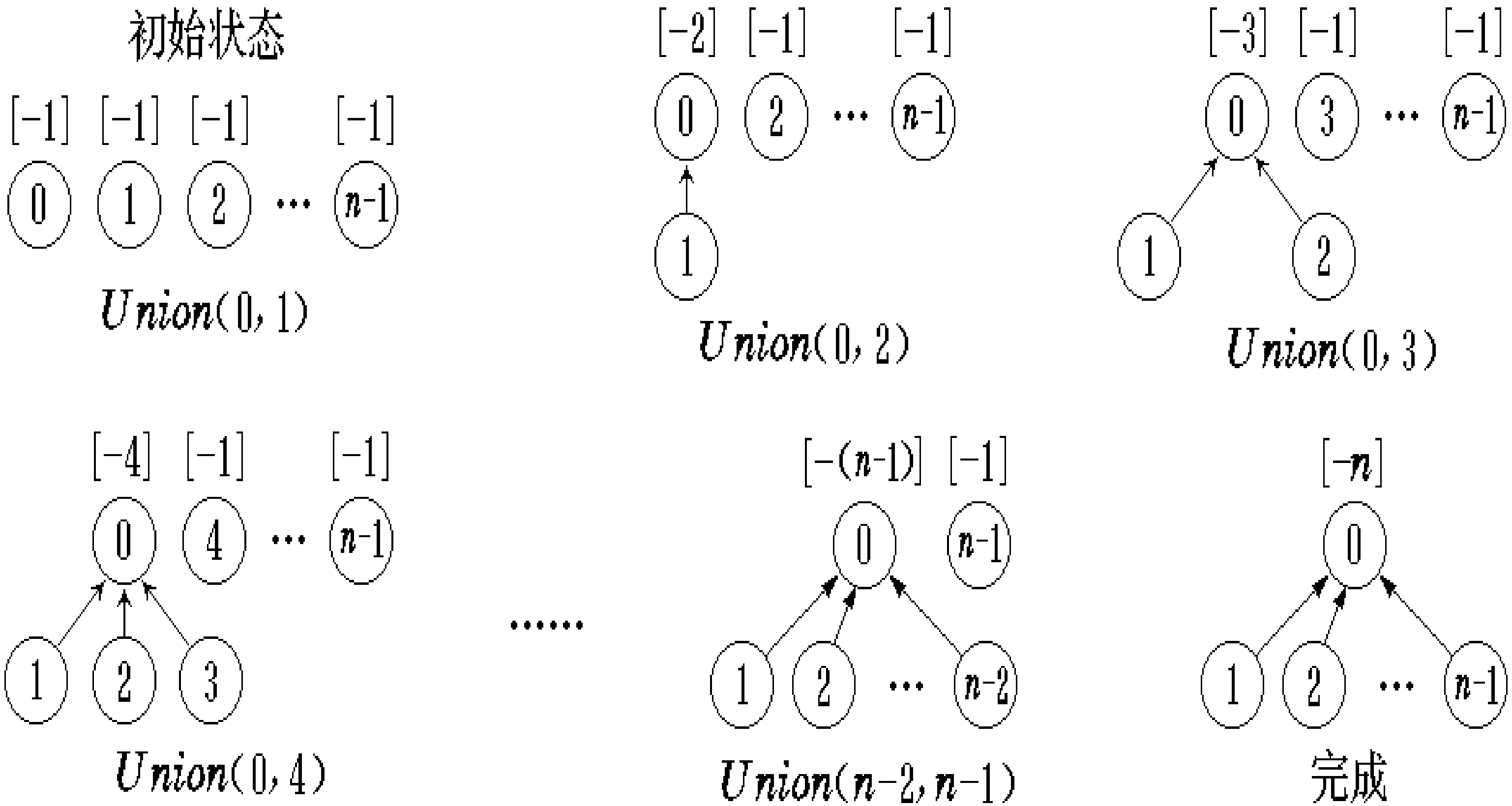
- 结点个数多的树的根结点作根。



Union操作的加权规则

- 为避免产生退化的树，改进方法是先判断两集合中元素的个数。
  - 如果以 i 为根的树中的结点个数少于以 j 为根的树中的结点个数，即 parent[i] > parent[j]，则让 j 成为 i 的双亲；
  - 否则，让 i 成为 j 的双亲。

```
void UFSets :: WeightedUnion ( int Root1, int Root2 ) {
//按 Union 的加权规则改进的算法
    int temp = parent[Root1] + parent[Root2];
    if ( parent[Root2] < parent[Root1] ) {
      parent[Root1] = Root2;  // Root2 中结点数多
      parent[Root2] = temp;  // Root1 指向 Root2
    }
    else {
      parent[Root2] = Root1;  // Root1 中结点数多
      parent[Root1] = temp;  // Root2 指向 Root1
    }
}
```

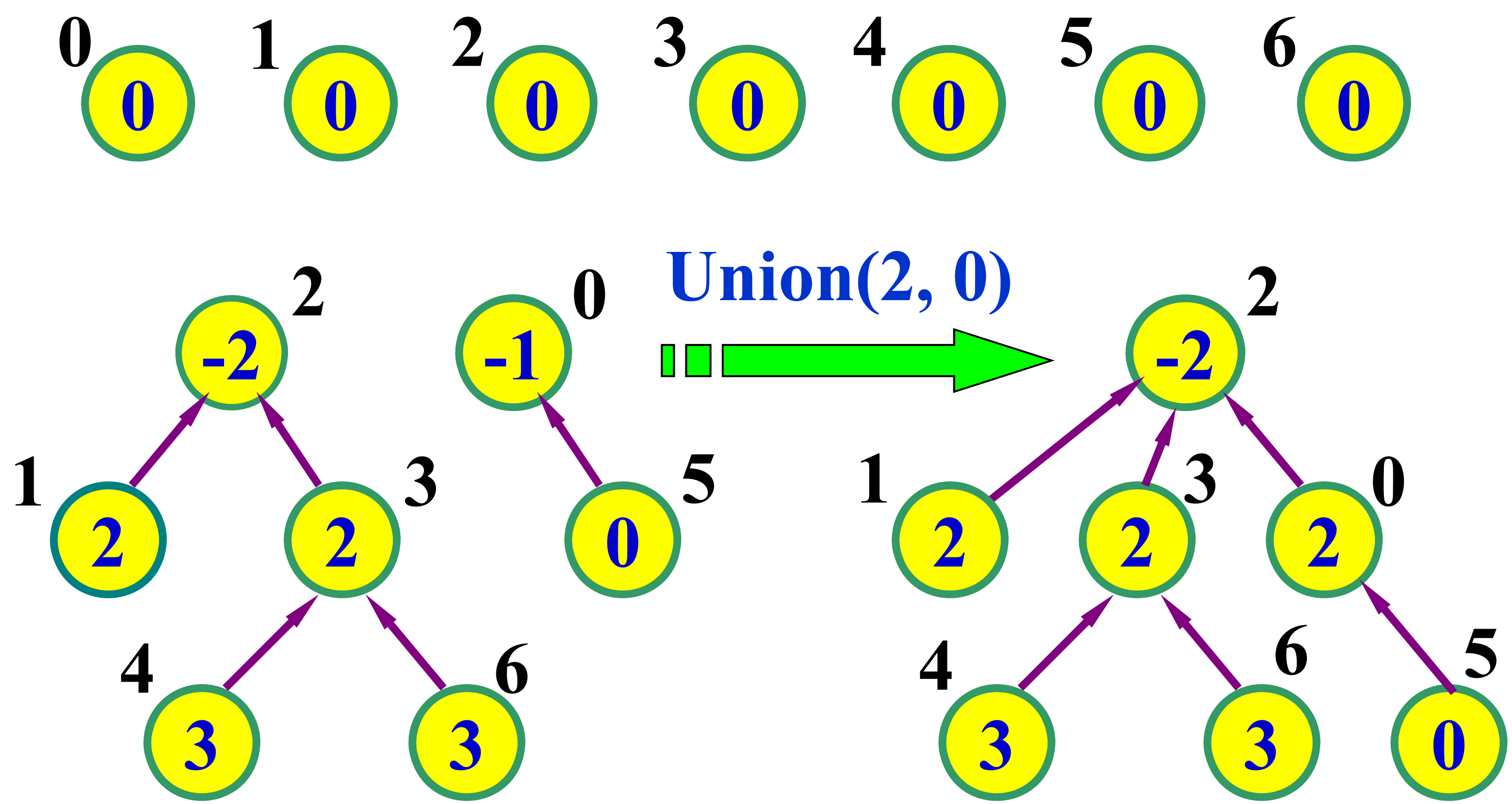


初始状态
[-1] [-1] [-1] [-1]
0 1 2 ··· n-1
Union(0, 1)
[-2] [-1] [-1]
0 2 ··· n-1
1
Union(0, 2)
[-3] [-1] [-1]
0 3 ··· n-1
1 2
Union(0, 3)
[-4] [-1] [-1]
0 4 ··· n-1
1 2 3
Union(0, 4)
······
[-(n-1)] [-1]
0 n-1
1 2 ··· n-2
Union(n-2, n-1)
[-n]
0
1 2 ··· n-1
完成

使用加权规则得到的树

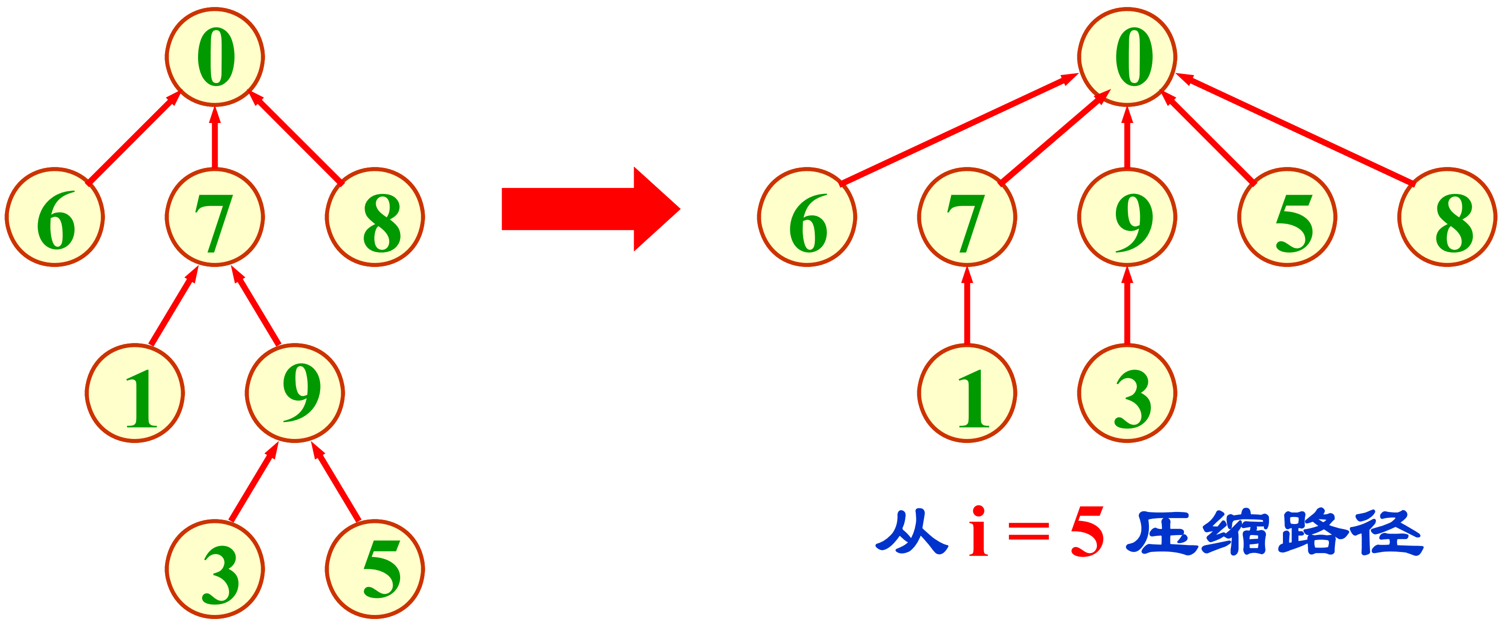
■ 按树高度合并

➢ 高度高的树的根结点作根。



Union操作的折叠规则

- 为进一步改进树的性能，可以使用如下折叠规则来“压缩路径”。即：**如果 j 是从 i 到根的路径上的一个结点，且 parent[j]≠root[i]，则把 parent[j] 置为 root[i]。**

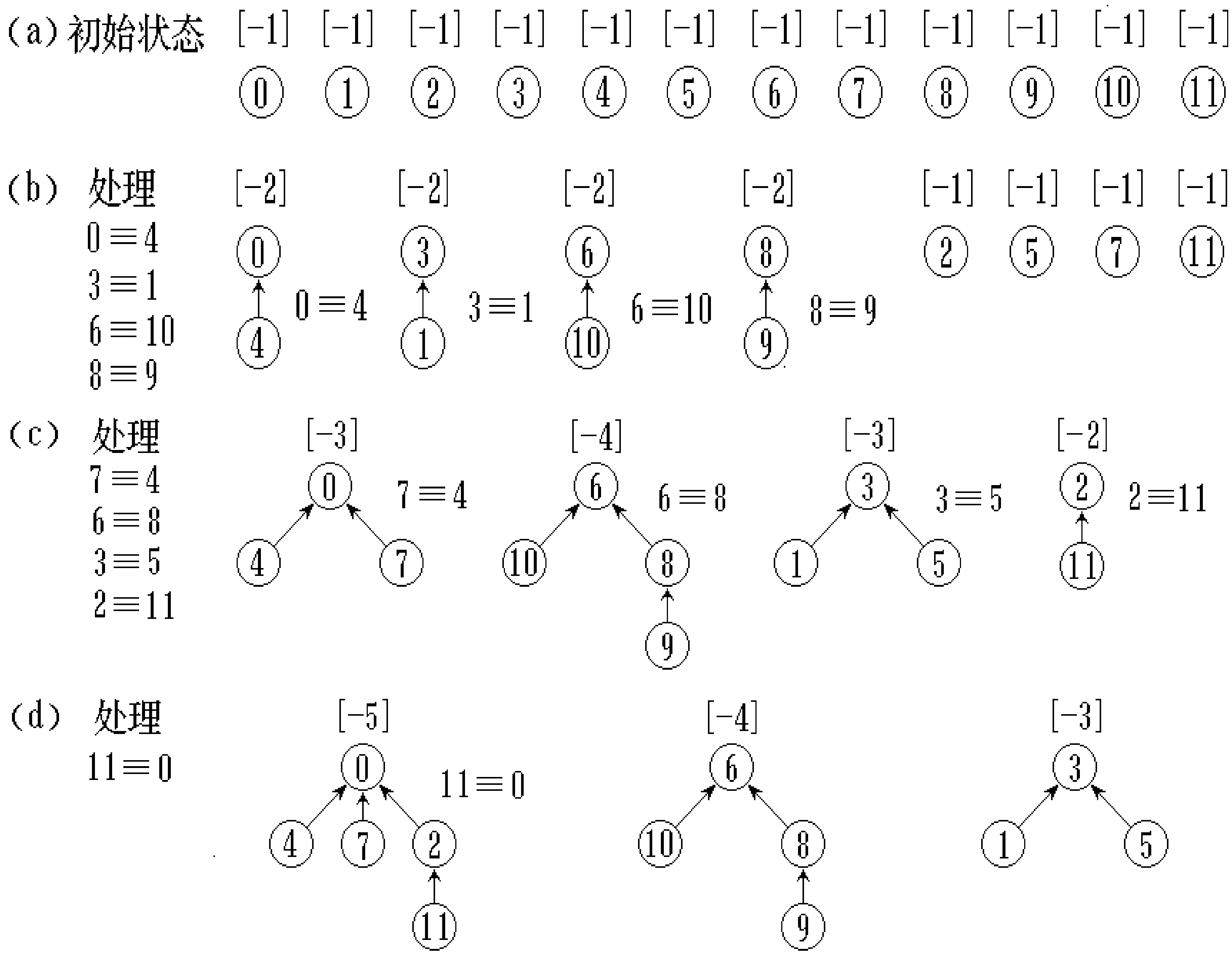


```
int UFSets :: CollapsingFind ( int i ) {
//使用折叠规则的搜索算法
    for ( int j = i; parent[j] >= 0; j = parent[j] );
    //让 j 循双亲指针走到根
    while ( parent[i] != j ) {  //换 parent[i] 到 j
        int temp = parent[i];
        parent[i] = j;  i = temp;    }
    return j;
}
```

使用折叠规则完成单个搜索，所需时间大约增加一倍。但是，能减少在最坏情况下完成一系列搜索操作所需的时间。

使用并查集处理等价对形成等价类的过程

- 最初，全部 **n** 个元素各自在自己的等价类中，**parent[i] = -1**；
- 以后，每处理一个等价对**(i ≡ j)**，先要确定 **i** 和 **j** 所在的集合，如果两个集合不同，则取其并集，否则不做任何事情。



(a) 初始状态
[-1] [-1] [-1] [-1] [-1] [-1] [-1] [-1] [-1] [-1] [-1] [-1]
0 1 2 3 4 5 6 7 8 9 10 11
(b) 处理
0≡4
3≡1
6≡10
8≡9
[-2] [-2] [-2] [-2] [-1] [-1] [-1] [-1]
0 3 6 8 2 5 7 11
4 1 10 9
0≡4
3≡1
6≡10
8≡9
(c) 处理
7≡4
6≡8
3≡5
2≡11
[-3] [-4] [-3] [-2]
7≡4
6≡8
3≡5
2≡11
(d) 处理
11≡0
[-5] [-4] [-3]
11≡0