第9章 并查集

学习要点:

- 理解以不相交的集合为基础的抽象数据类型并查集。
- 掌握用数组实现并查集的方法。
- 掌握用树结构实现并查集的方法。
- 理解将小树合并到大树的合并策略及其实现。
- 掌握路径压缩技术及其实现方法。

9.1 并查集的定义及其简单实现

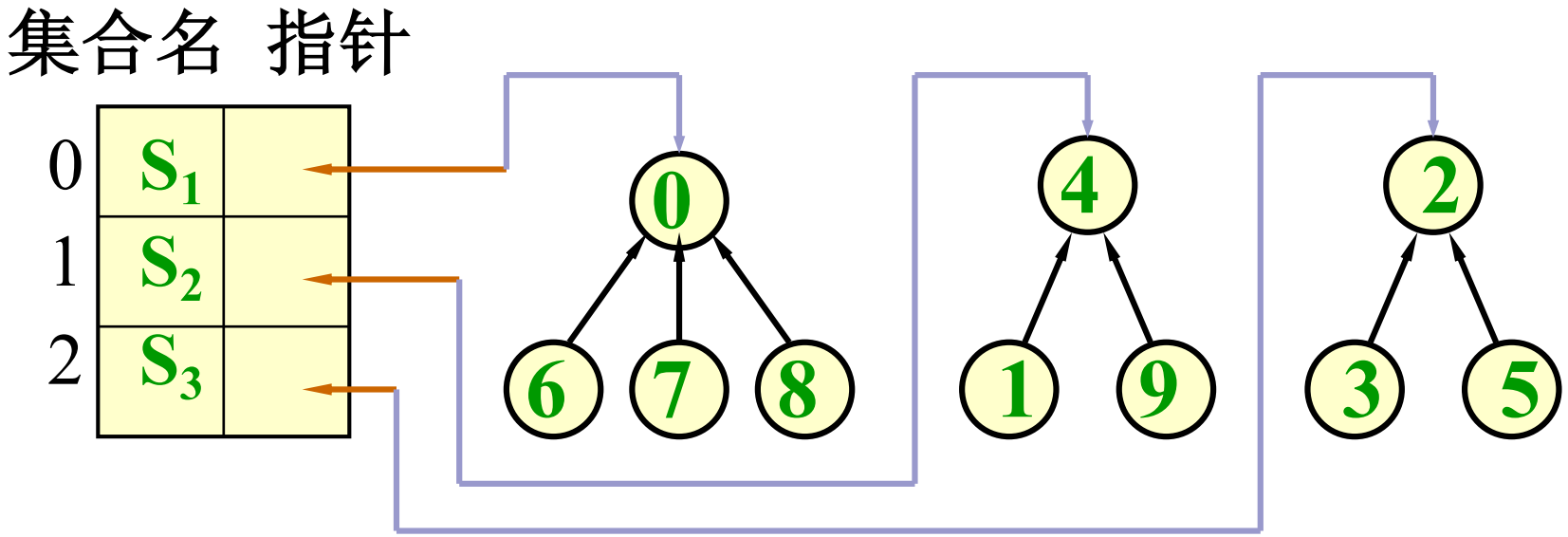
在一些应用问题中，需将n个不同的元素划分成一组不相交的集合。开始时，每个元素自成一个单元素集合，然后按一定顺序将属于同一组元素的集合合并。其间要反复用到查询某个元素属于哪个集合的运算。适合于描述这类问题的抽象数据类型称为并查集。

并查集（Union-Find Sets）

- 并查集支持以下两种操作：
  - UFunion（A，B，U）　//并操作
  - UFfind（e）　//找出包含元素e的集合，并返回该集合的名字
- 对于并查集来说，每个集合用一棵树表示。
- 为此，采用树的父亲数组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。

- 在父亲数组表示中，第 i 个数组元素代表包含集合元素 i 的树根结点。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。

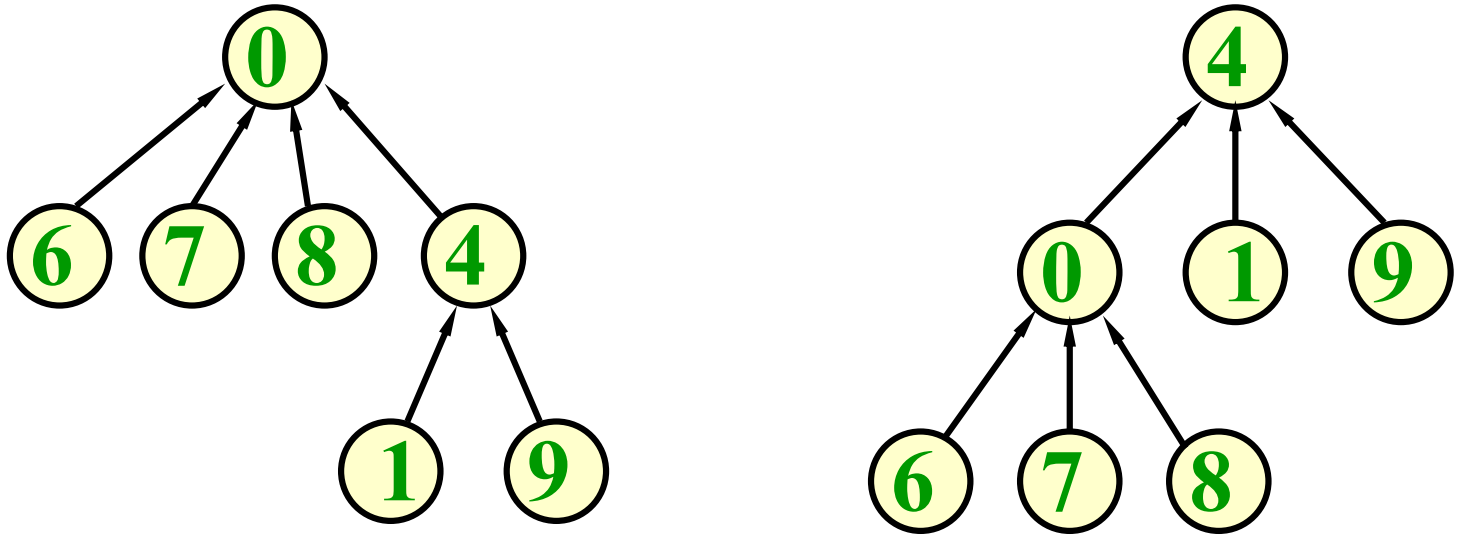
- 初始时，用初始化函数 UFinit(size) 构造一个森林，每棵树只有一个结点，表示集合中各元素自成一个子集合。

- 用 UFfind(i) 寻找集合元素 i 的根。如果有两个集合元素 i 和 j，UFfind(i) == UFfind(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 不在同一个集合中，可用 UFunion(i, j, U) 将它们合并到一个集合中。

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

集合 S_1, S_2 和 S_3 的父亲数组表示



$S_1 \cup S_2$的可能的表示方法

返回章目录

9.2 用父结点数组实现并查集

```
typedef struct ufset * UFset;
typedef struct ufset
{
    int  * parent;
}UFS;
```

其中**parent**是表示树结构的父结点数组。元素**x**的父结点为**parent[x]**。

9.2 用父结点数组实现并查集

- UFfind(e)运算就是从元素e相应的结点走到树根处，找出所在集合的名字。

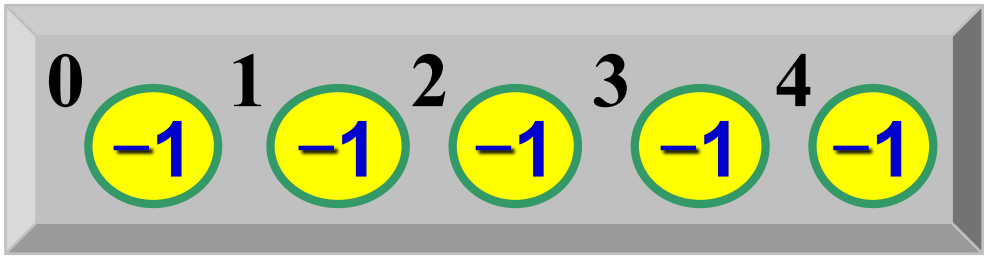
```
int UFfind(int e, UFset U)
{
   while (U->parent[e]) e = U->parent[e];
   return e;
}
```

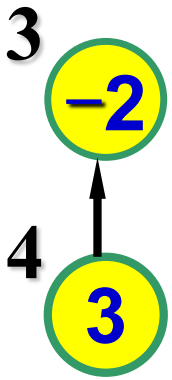
- 合并2个集合，只要将表示其中一个集合的树的树根改为表示另一个集合的树的树根的儿子。

```
int UFunion(int i, int j, UFset U)
{
    U->parent[j] = i;
    return i;
}
```

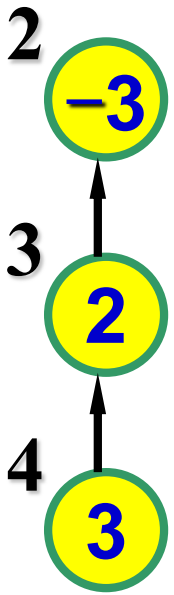
- 容易看出，在最坏情况下，合并可能使n个结点的树退化成一条链。在这种情况下对所有元素各执行一次Find将耗时$O(n^2)$。（见下页图）

- 合并

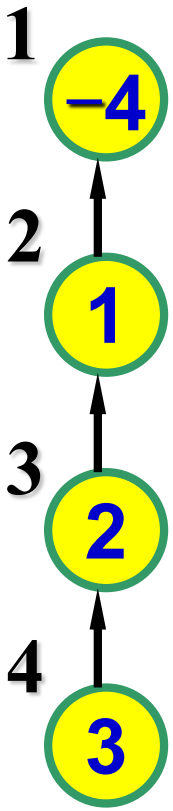




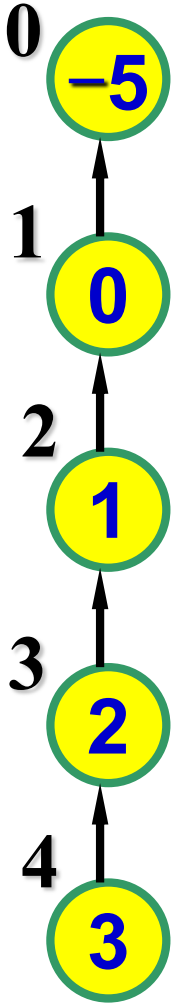
Union(3,4)



Union(2,3)



Union(1,2)



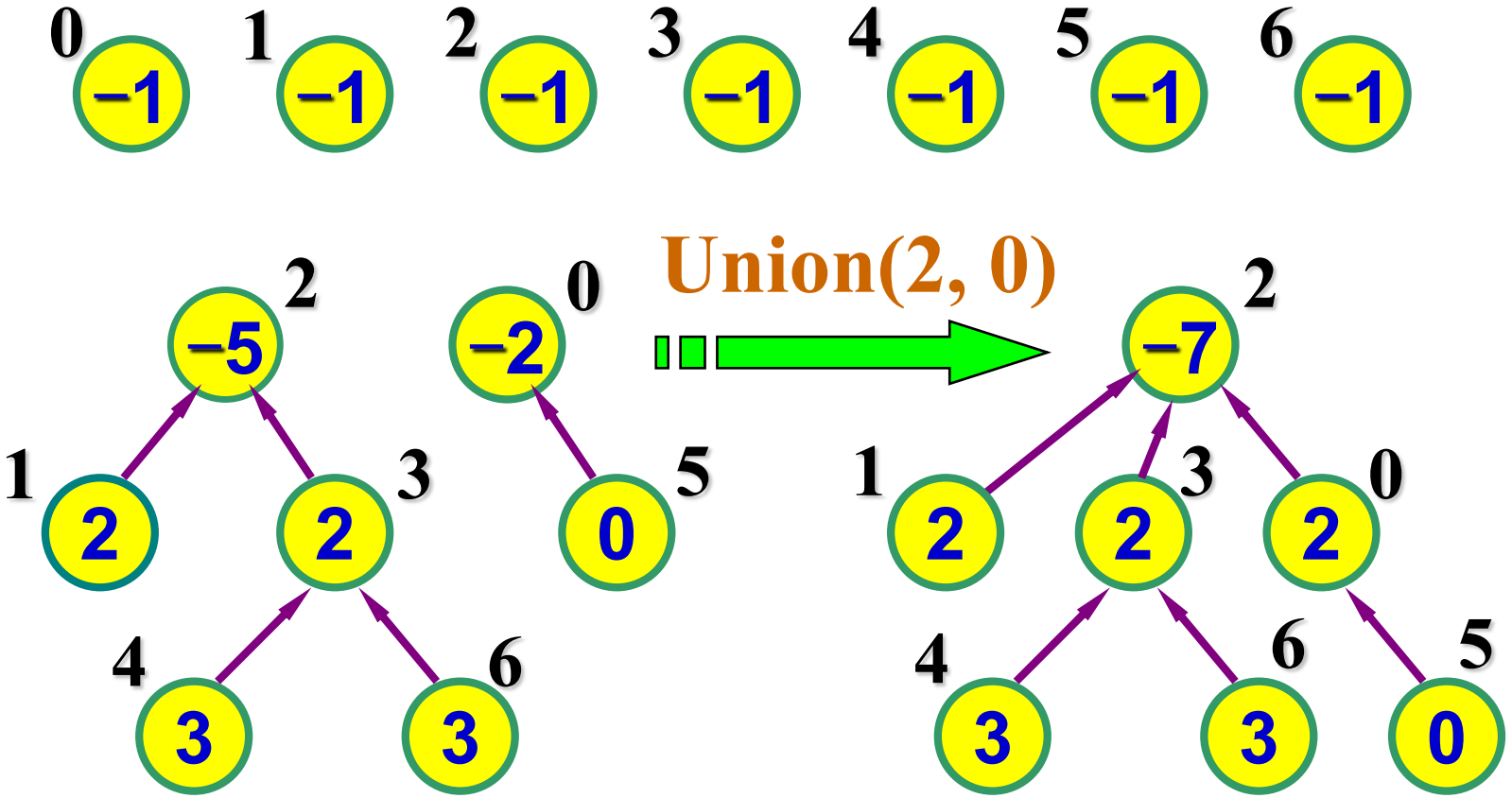
Union(0,1)

- 执行一次UFunion操作所需时间是 O(1)，n-1次UFunion操作所需时间是O(n)。
- 若再执行UFfind(0), UFfind(1), …, UFfind(n-1), 若被搜索的元素为 i，完成 UFfind(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

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

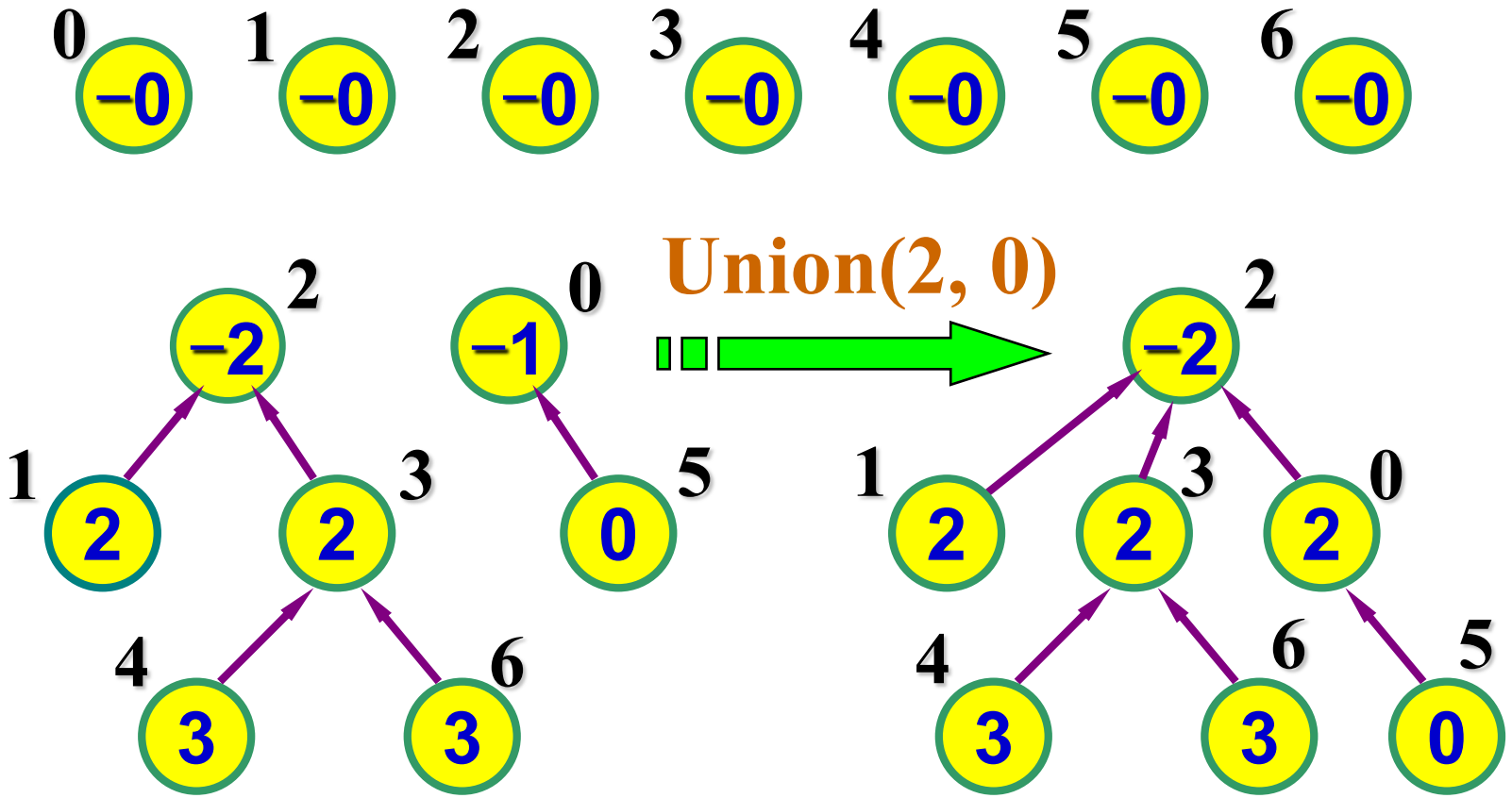
- 改进的方法
  - 按树的结点个数合并
  - 按树的高度合并
  - 压缩元素的路径长度

- 按树结点个数合并
  - 结点个数多的树的根结点作根



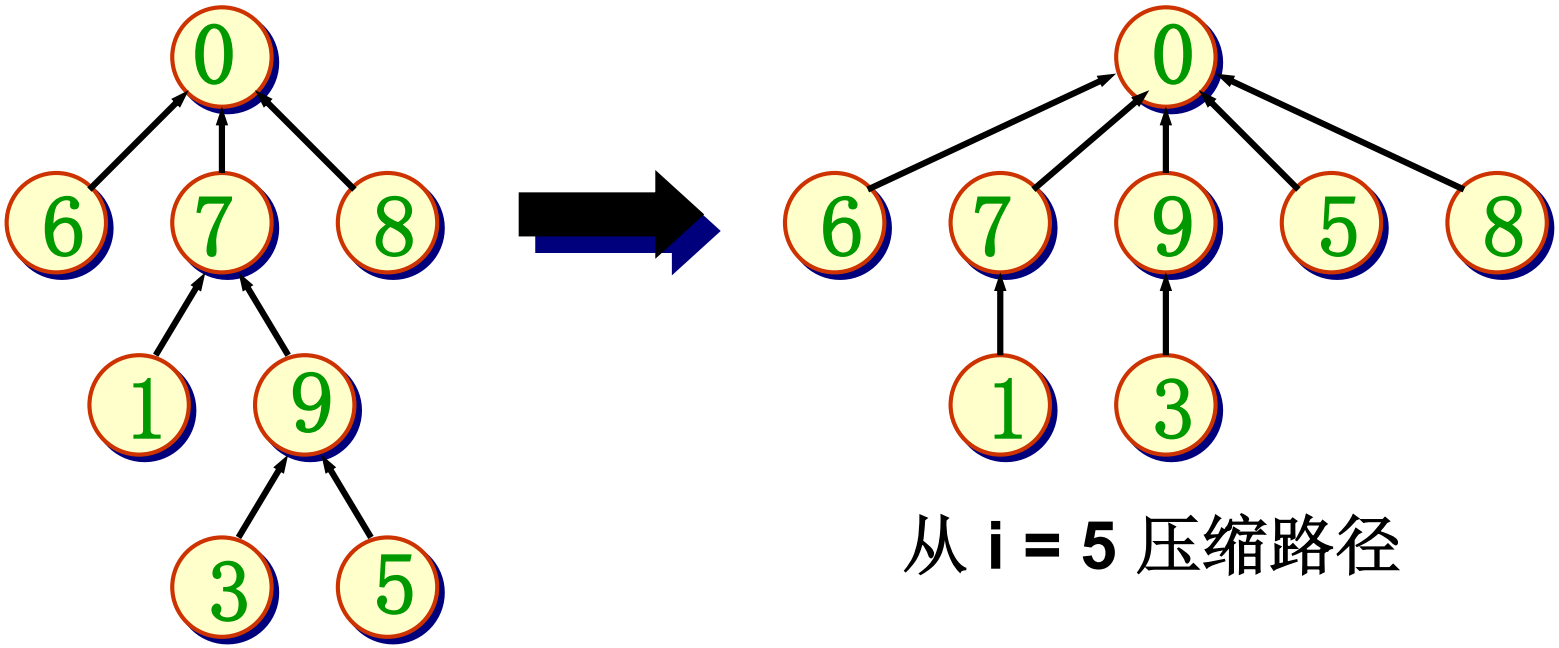
按树高度合并

- 高度高的树的根结点作根



UFunion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[j]， 则把 parent[j] 置为 root[i]。



从 i = 5 压缩路径

9.3 并查集的应用—离线最小值问题

★问题描述：

给定集合 S={1，2，…，n}，以及由 n 个 Insert(x)和 m 个 DeleteMin()运算组成的运算序列。其中 n 个 Insert(x)运算将集合 S={1，2，…，n}中每个数插入动态集合 T 恰好一次，DeleteMin()每次删除动态集合 T 中的最小元素。离线最小值问题要求对于给定的运算序列，计算出每个 DeleteMin()运算输出的值。换句话说，要求计算数组 out，使第 i 次 DeleteMin()运算输出的值为 out[i]，i=1，2，…，m。在执行具体计算前，运算序列已给定，这就是问题表述中离线的含义。

为了计算输出数组 out 的值，可以用一个优先队列 H，按照给定的运算序列依次执行 n 个 Insert(x)和 m 个 DeleteMin()运算，将第 i 次 DeleteMin()运算的结果记录到 out[i]中。执行完所给的运算后，数组 out 即为所求。在最坏情况下，这个算法需要 $O(m\log n)$ 计算时间。

当 $m=\Omega(n)$时，算法需要的计算时间为 $O(n\log n)$ 。

实际上，上述算法是一个在线算法，即每次处理一个运算，并不要求事先知道运算序列。因而算法没有用到问题的离线性质。利用并查集和问题的离线性质可以将算法的计算时间进一步减少为 $O(n\alpha(n))$。

将给定的 n 个 Insert 和 m 个 DeleteMin 运算组成的运算序列表示为：

$$I_1DI_2DI_3D\cdots I_kDI_{k+1}$$

其中 I_j，1≤j≤k+1，为连续若干个（可以为 0）Insert 运算组成的运算序列，D 表示 DeleteMin 运算。可用并查集算法模拟这个运算序列。开始时，将 I_j 中的 Insert 运算插入动态集合 T 中的元素用 UFunion 运算组织成一个集合，并将该集合记为第 j 个集合，1≤j≤k+1。由于第 j 个集合的名与其序号可能不同，算法中用 2 个数组 si 和 is 来表示集合名与其序号的对应关系。例如，第 j 个集合名为 name 时，si[name]=j 且 is[j]=name。另外，用 2 个数组 prev 和 next 来表示 I_j 之间的顺序。开始时，prev[j]=j-1, 1≤j≤k+1 且 next[j]=j+1，0≤j≤k。

接下来，对每 i，1≤i≤n，用 UFfind 运算计算出集合序号 j，使得 i∈ I_j。这表明第 j 个 DeleteMin 运算输出元素 i，即 out[j]=i。然后用 UFunion 运算将集合 I_j 与集合 I_{j+1} 合并，并修改数组 prev 和 next 的值，将 j 从链表中删除。算法结束后，输出数组 out 给出正确的计算结果。

★编程任务：

对于给定的由 n 个 Insert(x)和 m 个 DeleteMin()运算组成的运算序列，用并查集编程计算出每个 DeleteMin()运算输出的值。

★数据输入：

由文件 input.txt 给出输入数据。第一行有 2 个正整数 n 和 m，分别表示运算序列由 n 个 Insert(x)和 m 个 DeleteMin()运算组成。接下来的 1 行中有 n+m 个整数。整数 x>0 时表示执行 Insert(x)运算；整数 x=-1 时表示执行 DeleteMin()运算。

★结果输出：

将计算出的每个 DeleteMin()运算输出的值依次输出到文件 output.txt。

输入文件示例

input.txt

```
10 6
10 9 -1 -1 8 7 6 -1 -1 -1 5 4 3 2 1 -1
```

输出文件示例

output.txt

```
9 10 6 7 8 1
```