MF-Sets

Aims

- To develop efficient data structures to group *n* different elements in a collection of *k* disjoint sets S ={S_1, S_2,…, S_k } with two kinds of operations:

- **Union** (**Merge**) of 2 disjoint sets and
- **Search** (**Find**) to what set a given element belongs to.

Reference

Michael T. Goodrich and Roberto Tamassia. "*Data Structures & Algorithms in Java*" (4th edition), John Wiley & Sons, 2005

(chapter 11, section 6)

Contents

Disjoint sets: MF-sets

A **disjoint-set data structure** is a data structure that keeps track of a set of elements partitioned into a number of disjoint (non overlapping) subsets

Given a set C previously fixed , a relation R defined in C is a sub-set of the Cartesian product *C*x*C* such that *aRb* means (a, b)∈R

A relation is an equivalence relation if the following 3 properties are accomplished:

- **Reflexive**: *aRa* for each a ∈ C
- **Symmetric**: *aRb* IFF *bRa*, for each a, b ∈ C
- **Transitive**: *aRb* y *bRc* implies *aRc*, for each a, b, c ∈ C

A set of elements can be partitioned in equivalence classes from the definition of an equivalence relation

Disjoint sets: MF-sets

The problem of Union-Find (Merge-Find) has a search space the set of the possible partitions of a set *C*

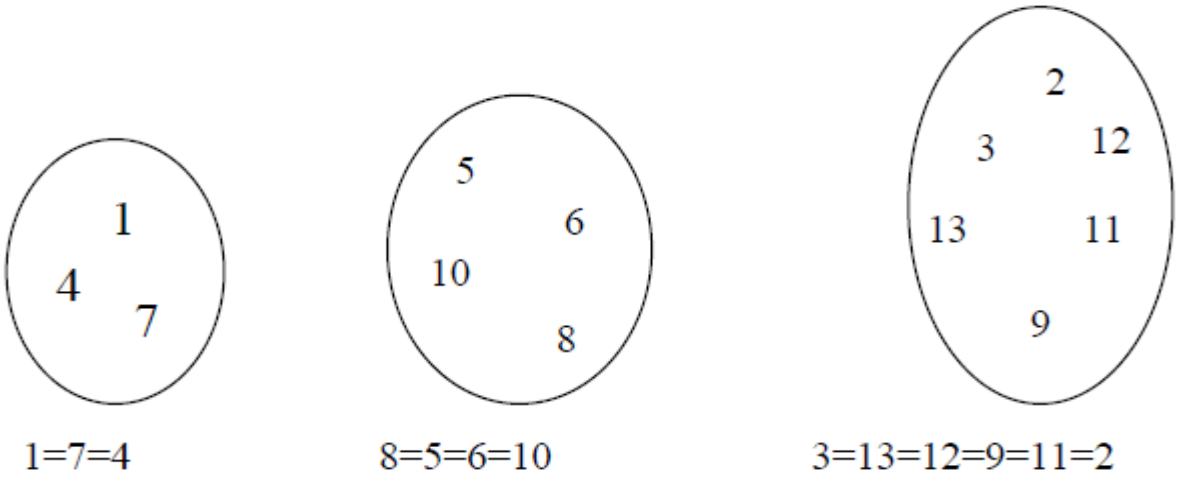
The aim is to develop an efficient data structure in order to group together *n* different elements in a collection of *k* disjoint sets

S ={S_1, S_2,…, S_k } with 2 kinds of operations:

- **Union** (**Merge**) of 2 disjoint sets and
- **Search** (**Find**) to what set a given element belongs to.

Disjoint sets: MF-sets

An MF-set is a data structure of type set in which the elements are organised in **disjoint sub-sets** and the **number of elements is fixed** (no elements are added nor deleted)



Each sub-set can be identified by one of its members

Applications: Equivalence between finite state automata, **connected components of a non directed graph, minimum spanning tree (minor weight) in a non directed graph (Kruskal algorithm)**

MF-sets: Operations

```
public interface MFSet
{
  /* it returns the identifier of the set the element x belongs to */

  public int find (int x);

  /* it merges the sets x and y belong to */

  public void merge (int x, int y);
}
```

Note: The set *C* will be such as {0,1,…, *n*-1}
where *n* is the number of elements of *C*

MF-sets: Array representation

M is an array of *int* of size $n = |C|$
M[i] indicates directly the set of class the element *i* belongs to

Complexity of *find(i)* is O(1)
Complexity of *merge(i,j)* is O(*n*)

EJEMPLO

{{0}, {1}, {2}, {3}, {4}}	0	1	2	3	4

Merge(0,1)

{{0, 1}, {2}, {3}, {4}}	0	0	2	3	4

Merge(1,2)

{{0, 1, 2}, {3}, {4}}	0	0	0	3	4

Merge(3,4)

{{0, 1, 2}, {3, 4}}	0	0	0	3	3

Merge(2,3)

{{0, 1, 2, 3, 4}}	0	0	0	0	0

MF-sets as disjoint-set forests

Every **sub-set** is a **tree**:

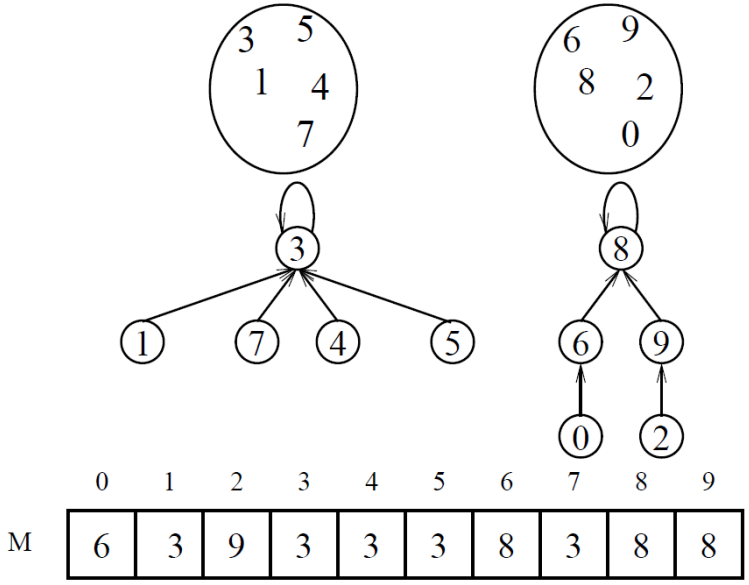
- The nodes of the tree are elements of the set
- **Every node refers to its father**
- **The root of the tree can be used to represent the partition** or class

The **MF-set** is represented as a **collection of trees**: i.e., a **forest**

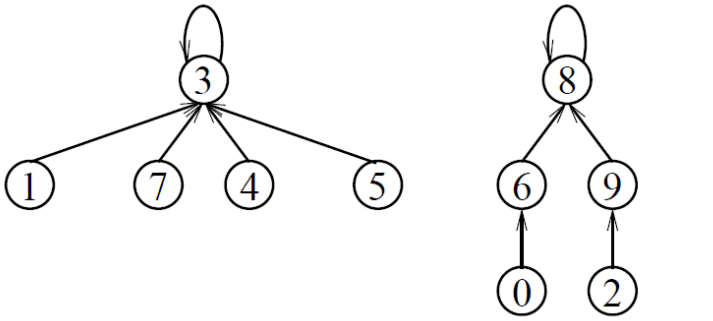
MF-sets as disjoint-set forests

The trees are not necessarily binary trees but the representation is easy and only a reference to the father is needed: *M* = array[*n*] of *int*

- *M*[*x*]: father of element *x*
- if *M*[*x*] = *x*, *x* is the root of a tree of the forest

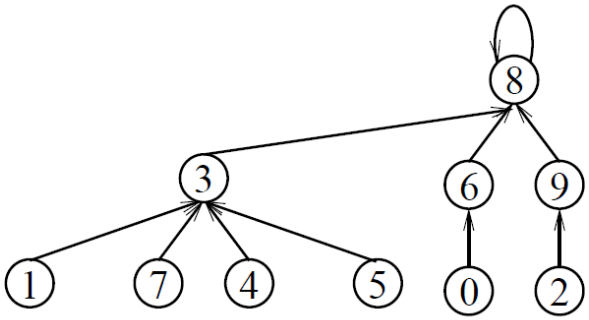


Operations with MF-sets: Examples



0	1	2	3	4	5	6	7	8	9
6	3	9	3	3	3	8	3	8	8

$$\begin{aligned} \text{Find}(0) &= 8 \\ \text{Find}(9) &= 8 \\ \text{Find}(4) &= 3 \\ \text{Find}(3) &= 3 \end{aligned}$$



0	1	2	3	4	5	6	7	8	9
6	3	9	**8**	3	3	8	3	8	8

$\text{Merge}(3, 8)$

Operations with MF-sets: Algorithms

```
/* MFSet where the operations find and merge were not optimised  */

public class ForestMFSet implements MFSet {
  protected int size;
  protected int n_partitions;
  protected int mfset[];

  /* Constructor: it creates a MFSet of a given size and it initialises it  */

  public ForestMFSet (int n) {
    size= n;
    n_partitions = size;

    mfset = new int[size];
    for (int i = 0;i<size;++i) mfset[i] = i;
  }
```

Operations with MF-sets: Algorithms

```
/* it returns the identifier of the set the element x belongs to */
public int find (int x) {
   while (mfset[x] != x) x = mfset[x];
   return x;
 }

 /* it merges the sets x and y belong to*/
 public void merge (int x, int y) {
   int rx = find(x);
   int ry = find(y);

   if(rx != ry) {
     n_partitions--;          // partitions are merged
     mfset[rx] = ry;          // x is appended to y
   }
 }
}
```

Operations with MF-sets: Worst case

A sequence of *m* operations Merge and Find has a complexity O(*mn*) in the worst case (it is possible to create a degenerate tree in a list of *n* nodes)

Operación	Bosque
MFset(5)	0 1 2 3 4
Merge(0,1)	0(1) 2 3 4
Merge(2,0)	2(0(1)) 3 4
Merge(3,2)	3(2(0(1))) 4
Merge(4,3)	4(3(2(0(1))))

MF-sets: How to improve complexity

A sequence of *m* operations *Merge* and *Find* has a complexity O(*mn*) in the worst case (degenerate tree: a list)

Complexity can be improved if the height of the tree is reduced: *m* operations *Merge* and *Find* will have a quasi linear complexity with *m*:

- **Union by rank** (rank based on the height or based on the number of nodes)
- **Path compression**

With these improvements the complexity of the operations is practically constant: it is the inverse of the Ackermann function that grows very slowly (e.g.: $\alpha(2^{65536}) = 5$)

Union by rank

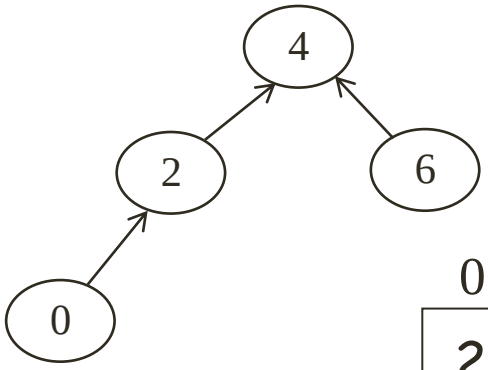
Merge is done in a way that the root of the tree whose height is smaller becomes the child of the tree whose height is higher.

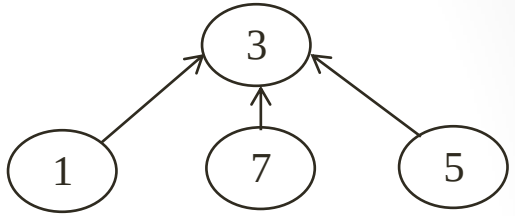
If the heights are different, the height after the merge is the height of the higher tree; in case the merge is with two trees of the same height, the final height is increased by 1

The value of the height of each tree of the forest is kept, for instance, in the array; the value associated to the root is usually a negative number:

if mfset[i]<0, i is the root of the tree and|mfset[i]| – 1 is the height of the tree

Union by rank: Example

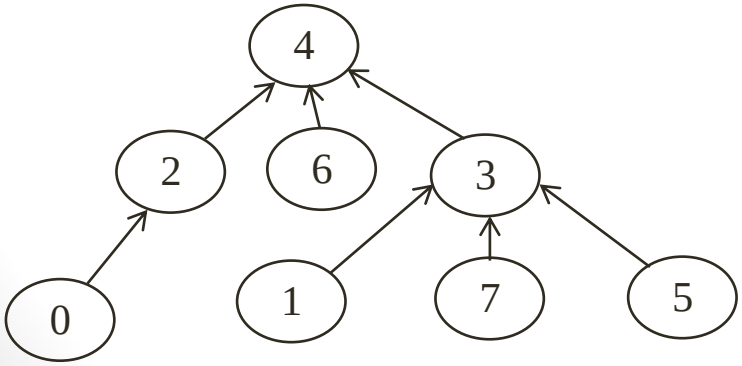




0	1	2	3	4	5	6	7
2	3	4	-2	-3	3	4	3

3 is the root of a tree of height 1

4 is the root of a tree of height 2



To merge both trees, the one with minor height (3) is subtree of (4)

0	1	2	3	4	5	6	7
2	3	4	4	-3	3	4	3

Union by rank: Algorithms

```
/* MFSet where the complexity of merge and find operations has been
improved with union by rank (based on height). The height will be
never greater than O(log(n)) */

public class MFSetUnionByRank extends ForestMFSet {

 /* the constructor creates a MFSet of a given size and initialises it */
 public MFSetUnionByRank(int n) {
  size = n;
  n_partitions = size;

  mfset = new int[size];
  for(int i = 0;i<size;++i)
   mfset[i] = -1;  // the negative value indicates the height for a root
 }
```

```
 /* It returns the identifier of the set the element x belongs to */
public int find (int x) {
   while (mfset[x] >= 0)
     x = mfset[x];
   return x;
  }

  /** It merges the sets x and y belong to; the set with smaller height is
merged with the other one */
  public void merge (int x, int y) {
   int rx = find(x); int ry = find(y);
   if (rx != ry) {
     n_partitions--;                                  // union of partitions
     if (mfset[rx] == mfset[ry]) {                    // same height
       mfset[rx] = ry;                                //x is appended to y
       mfset[ry]--;                                   // the height of y is updated
     }
     else if (mfset[rx] < mfset[ry]) mfset[ry] = rx;  //y is appended to x
     else mfset[rx] = ry;                             // x is appended to y
   }
  }
}
```

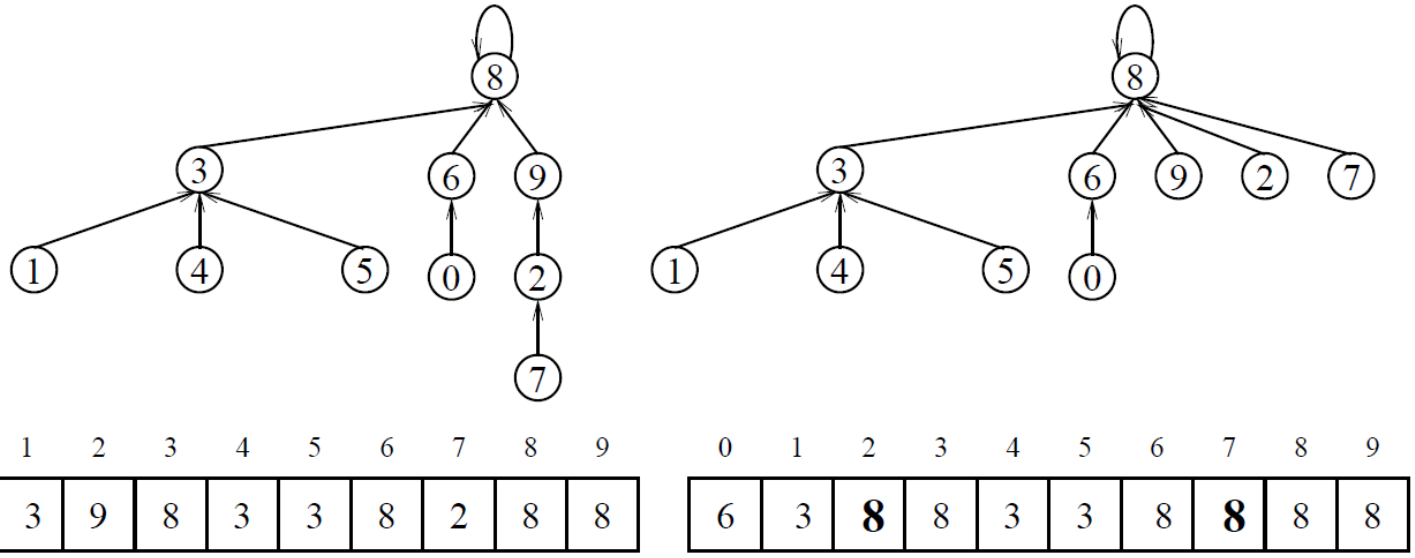
Union by rank: Example

Operación	0	1	2	3	4	Bosque
MFset(5)	-1	-1	-1	-1	-1	0 1 2 3 4
Merge(0,1)	-2	0	-1	-1	-1	0(1) 2 3 4
Merge(2,3)	-2	0	-2	2	-1	0(1) 2(3) 4
Merge(4,2)	-2	0	-2	2	2	0(1) 2(3,4)
Merge(0,2)	-3	0	0	2	2	0(1,2(3,4))

Path compression

The effect of the path compression on *Find* is that each node of the path from *x* to the root refers directly to the root of the tree: e.g. Find(7)

It allows to perform *m* search operations with complexity $O(m\ \alpha(m,n))$, where $\alpha(m,n)$ is the inverse of Ackermann function, that grows very slowly (example: $\alpha(2^{65536}) = 5$)



Path compression: Algorithms

```
/* MFSet where the complexity of the operations of the class
MFSetNormal is optimised reducing the height, when the find
method is invoked */

public class MFSetPathCompression extends ForestMFSet {

  /* The constructor creates a MFSet of a given size and initialises it */
  public MFSetPathCompression(int n) {
    super(n);
  }
```

```
  /* It returns the identifier of the set the element x belongs to ;
moreover, all the elements of a set are directly linked to the root */

  public int find (int x) {
    int rx = x;

     while (mfset[rx] != rx) rx = mfset[rx];   // it finds the root of the set

    while (mfset[x] != rx) {
      int tmp = x;
      x = mfset[x];
      mfset[tmp] = rx;                          // it is directly linked to the root
    }

    return rx;
  }
}
```

Combining strategies

It may happen that path compression diminishes the height of the tree. In this case, the height stored in the array does not necessarily represent the real height of the tree and it is an upper-bound of the real height O(log n) that in practice is much less due the path compression

Combining strategies: Algorithms

/* It returns the identifier of the set the element x belongs to ; all the elements of the set of x are linked directly with the father */

```
public int find (int x) {
  int rx = x;

  // it finds the root of the set
  while (mfset[rx] >=  0) rx = mfset[rx];

  while (mfset[x] != rx) {
    int tmp = x;
    x = mfset[x];
    mfset[tmp] = rx;  // ithe element is linked directly with the root
  }
  return rx;
 }
}
```

Complexity

Sequence of operations: *n -1 Merge* (union by rank) operations, *m Find operations*

Path Compresion	Total	† **Nota**	Amortizado
Sin	$O(m \log n)$		
Con	$O(m\,\alpha(m+n,n))$	$\simeq O(m)$	$\simeq O(1)$

† **Nota:** Lo usual es que $\alpha(m+n,n) \leq 4$ (ejemplo: $\alpha(2^{65536}) = 5$).

Exercises

Exercise 1

Design a method that shows the identifiers of each set of the *MFSet*.

Exercise 2

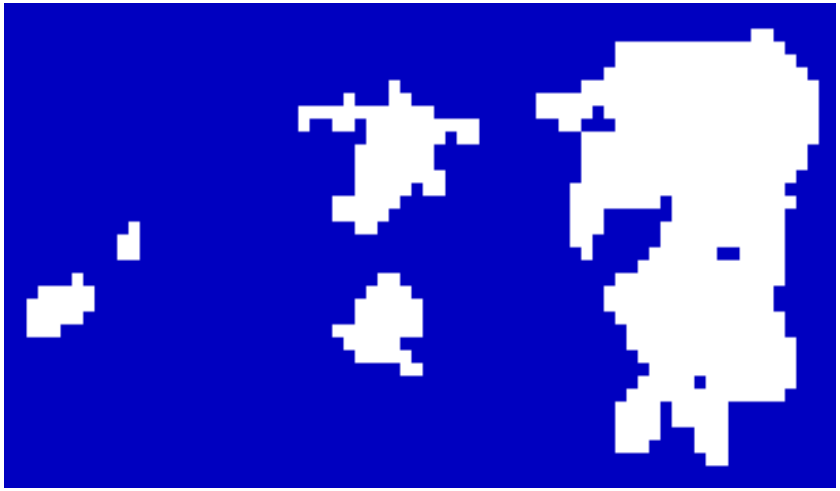
Design a method that shows the elements of the set a given element *x* belongs to.

Exercise 3

Design a method that returns the number of elements that we would have if we merge the sets the given elements *x* and *y* belong to.

Exercise 4

Given a *boolean* matrix that represents a map of an archipelago, a *true* value in a position (x,y) indicates that there is land in this point, whereas a *false* value indicates that there is sea.



Example:

The values *true* and *false* are represented with the white and blue colours, respectively.

Implement a method with the following profile in order to calculate the number of islands of the archipelago:

```
public static int numIsland(boolean[][] map);
```