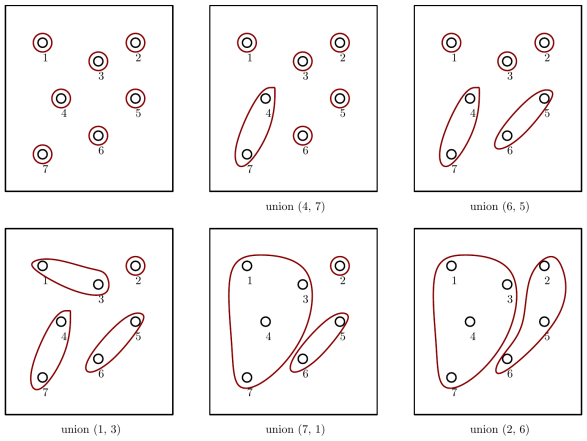
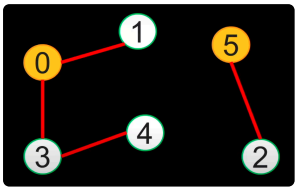
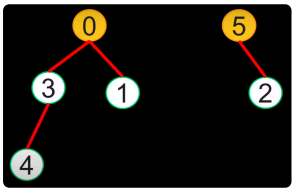
Union-Find

Explication de base en 5 minutes

- **Motivation**
  - 

    On suppose un partitionnement d'un ensemble d'éléments en *classes* (~ groupes). Par exemple, partitionner un groupe de personnes en fonction de leur tranche d'âge.
  - On veut **deux méthodes** :
    - `find(personne)`, qui renvoie la classe dans laquelle `personne` se trouve
    - `union(class1, class2)` qui *unit* deux classes (autrement dit, qui fait passer tous les éléments de `class2` dans `class1`, ou l'inverse selon le cas).
- **Considérations sur les structures de données à utiliser**
  - On va avoir une liste de `n` éléments et une liste d'ID de groupe correspondante. Par exemple :
    `elements = 1 2 3 4`
    `groupes  = A B B C` (`1` est dans le groupe `A`, `2` dans le groupe `B`, etc.)
  - Pour représenter l'*identifiant* d'un groupe (`A, B, C`), on va simplement choisir un *représentant* parmi ses membres. Par exemple, un représentant du groupe `B` peut être `2` ou `3`, donc on peut dire que "groupe B = groupe 2 ou 3". Ce sommet représentant est appelé la *racine*.
    Visualisons cela avec un autre exemple :
    - 

      On voit bien la séparation entre les deux groupes, et la racine de chacun...



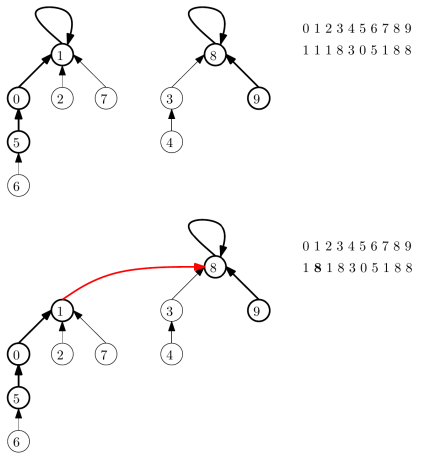
... et s'il y a une racine unique par groupe, on peut forcément représenter chaque groupe comme un arbre.

- **Pour la méthode `find`, il suffit donc de parcourir l' "arbre" de chaque groupe jusqu'à tomber sur la racine.**

```
// FIND

public int find(int p) {
while (p != parent[p])
p = parent[p];
return p;
}
```

  - Le fait qu'on assigne un parent (pas forcément la racine) à chaque élément (et que la racine est parente d'elle-même) s'explique par le fait que quand on veut faire une insertion dans un groupe donné, *on ne connaît pas toujours la racine de ce groupe*.
  - Par exemple, dans un système de classification par tranches d'âge, on pose `A = 24 ans, B = 31 ans, C = 33 ans`.
    Si on veut ajouter un élément `D` dans le même groupe que `C` sans connaître la façon dont les différentes "tranches d'âge" sont définies, alors il suffira de dire `ajouter(D) dans find(C)`.

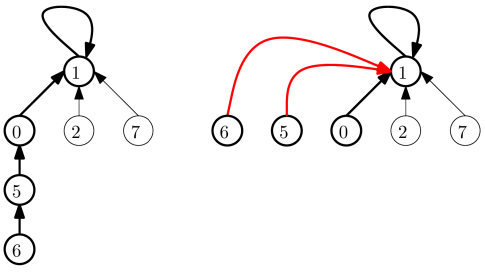
- **La méthode `union` est alors simple à comprendre.**



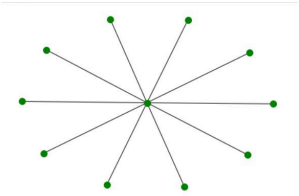
Il suffit de poser la racine d'un groupe en tant que feuille de l'autre groupe; ici, `1` a pour parent `8` après union.

```
//UNION rapide pondérée
//RAPIDE car on rattache un arbre entier à un autre en modifiant un seul noeud
//PONDÉRÉE car on prend en considération la taille des arbres à joindre

public void union(int p, int q) {

int i = find(p);
int j = find(q);
if (i == j)
    return;

// l'arbre le "moins haut" est rattaché à l'autre pour conserver un "équilibre".
if (height[i] < height[j])
    parent[i] = j;

else if (height[i] > height[j])
    parent[j] = i;

else {
    parent[j] = i;
    height[i]++;
    }
count--;
}
```

- **Compression de chemin**



Considérons `find(6)` : il est coûteux de devoir parcourir jusqu'à `log(n)` éléments à chaque fois qu'on appelle cette fonction... On voudrait donc bien "rattacher" tous les éléments non-adjacents à la racine à celle-ci dès qu'on les parcourt via un `find`.
On aurait alors quelque chose de similaire à un *graphe étoile* (sauf pour la racine qui pointe vers elle-même).



Un graphe étoile (à peu près ce vers quoi on se dirige théoriquement, à force de faire des `find`).

```
//Compression de chemin

public int find(int p) {
int root = p;
while (root != parent[root])
    root = parent[root];

while (p != root) { // Prendre "tous les sommets non-adjacents à la racine sur le
chemin".
                    // Dans l'exemple précédent, on va "embarquer" 6, puis 5, les
attacher
                    // à 1, et renvoyer 1.
    int newp = parent[p];
    parent[p] = root;
    p = newp;
}
return root;
}
```