Algorithmique
Cours 9 : Algorithme de Kruskal

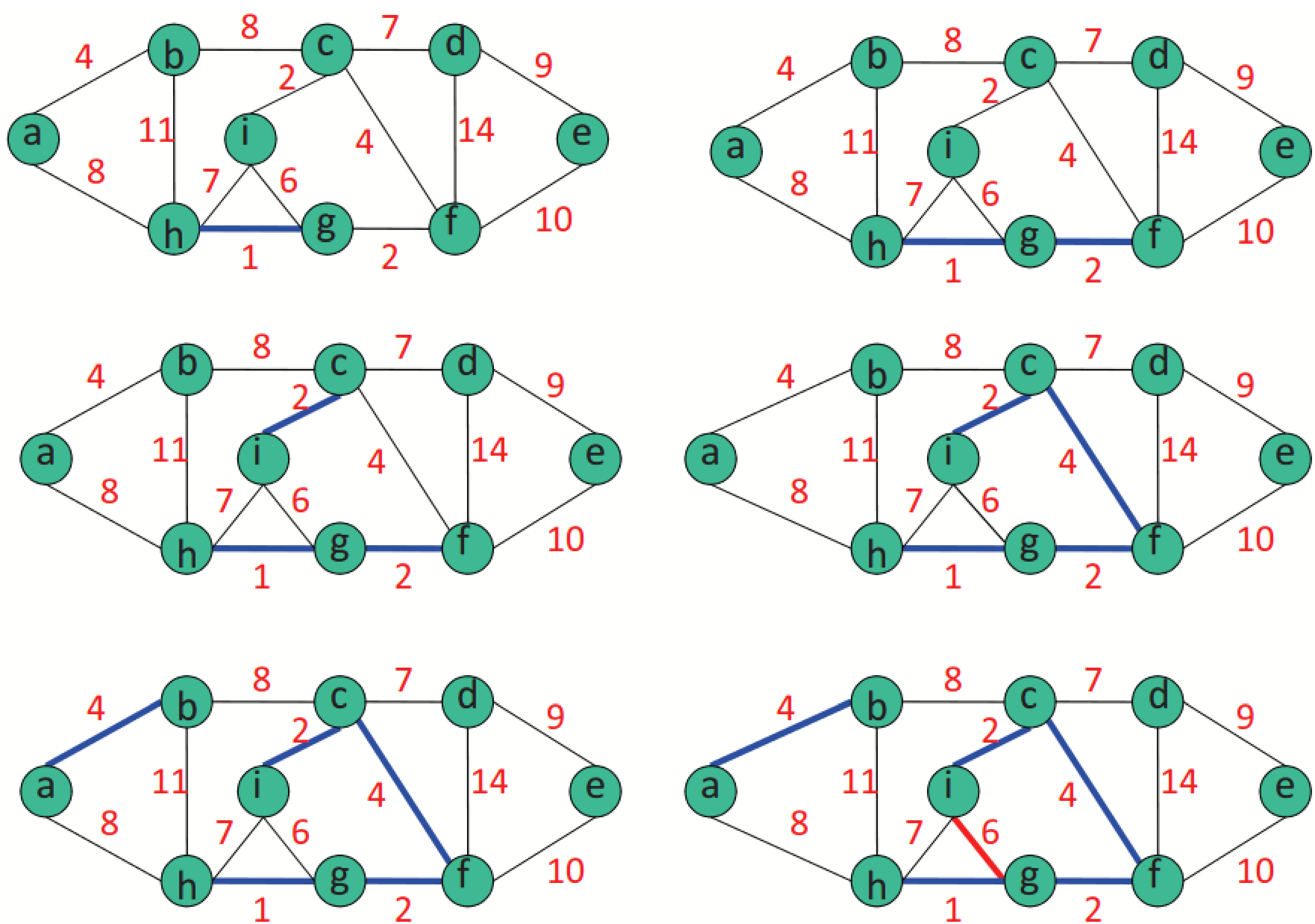
ROB3 – année 2014-2015

Algorithme de Kruskal

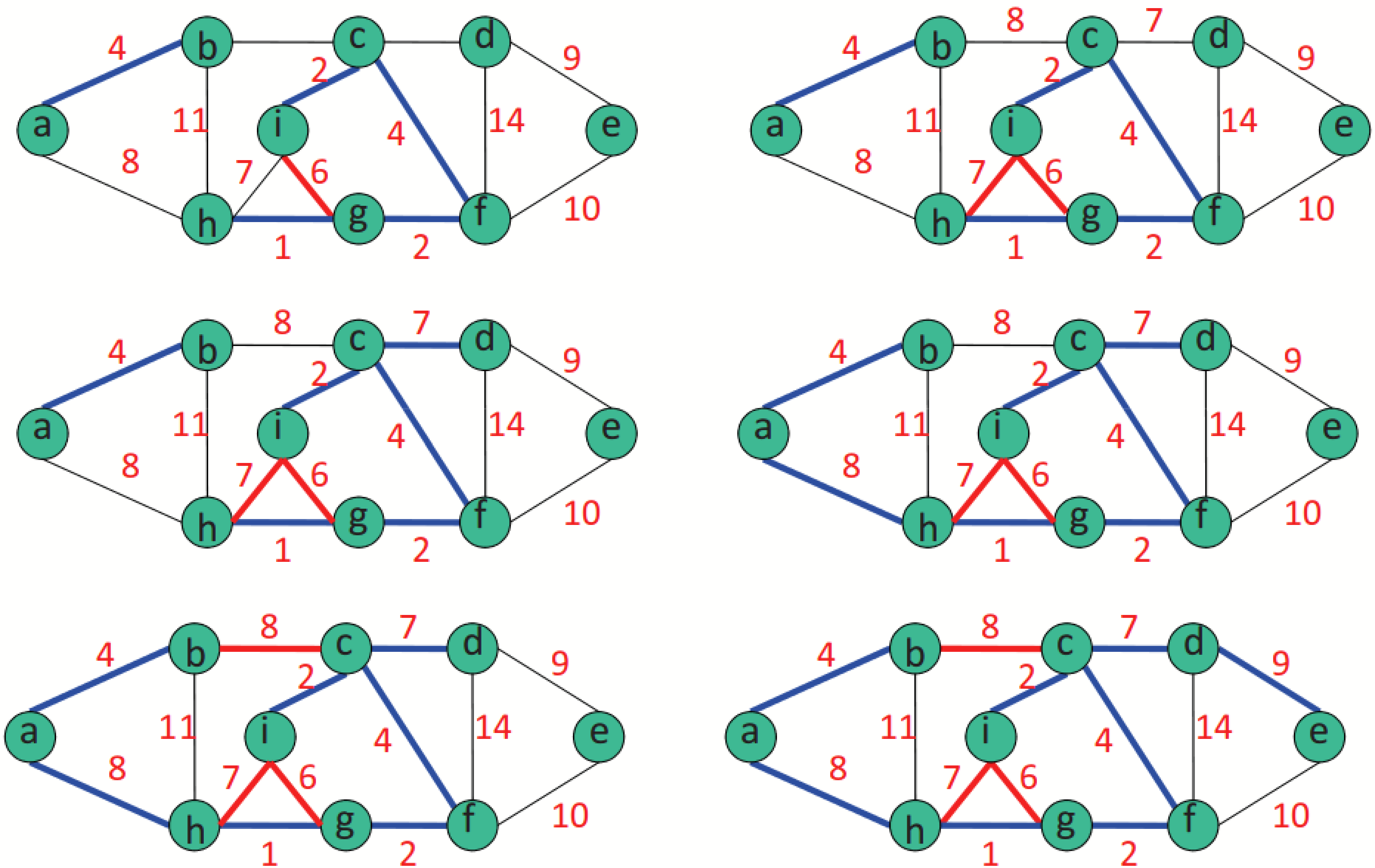
Principe de l'algorithme :
(H est à la fin de l'algorithme un arbre couvrant de coût minimum)

Algorithme de Kruskal:
Trier les arêtes par coût croissant;
H = arbre vide;
Examiner dans l'ordre chacune des arêtes {x,y} :
S'il n'existe pas de chaîne de x à y dans H :
    H = H U {x,y}
FinSi

Une exécution de l'algorithme de Kruskal



Une exécution de l’algorithme de Kruskal (suite)



Validité de l'algorithme de Kruskal

Hypothèse simplificatrice. Les coûts sont deux à deux distincts.

Propriété des cocycles. Soit S' un sous-ensemble (strict) de S, et e l'arête de plus petit coût parmi celles qui ont exactement une extrémité dans S' et une autre dans S\S'. Alors tout arbre couvrant minimum comporte e.

Preuve de validité de l'algorithme de Kruskal.
Lors de l'ajout d'une arête e = {s,s'} par l'algorithme de Kruskal, e est minimum dans le cocycle comportant les arêtes qui on exactement une extrémité dans la composante de s (appelons S' les sommets de cette composante) et une autre dans S\S'. D'après la propriété des cocycles, cette arête appartient à tout arbre couvrant minimum. D'où la validité.

Complexité de l'algorithme de Kruskal

Algorithme de Kruskal:
  Trier les arêtes par coût croissant;
  H = arbre vide;
  Examiner dans l'ordre chacune des arêtes {x,y} :
  S'il n'existe pas de chaîne de x à y dans H :
    H = H U {x,y}
  FinSi

Trier(ensemble des arêtes) se fait en O(m log m)
  Soit α la complexité de la comparaison
  « Composante connexe (x) = composante connexe (y) »

La complexité de l'algorithme est O(m log m + mα)

Montrons que la complexité est en O(m log m)

Union-Find

On souhaite travailler sur les partitions de E={1, …, n}

Exemple :

Partition = { {1,3} , {4} , {2,7,9}, {5,6,8,10} }

classe d'équivalence 1 2 3 4

Etant donné une partition, on souhaite

- Trouver la classe d'un élément : méthode Find
- Fusionner deux classes d'équivalence : méthode Union

Dans l'algorithme de Kruskal : E = ensemble des sommets.

- Deux sommets u et v sont dans une même composante connexe si Find(u)= Find(v)
- Si on choisit l'arête (u,v), on fusionne deux composantes connexes en une : Union(u,v)

Une solution peu efficace

On représente la partition par un tableau tab, tel que tab[i] soit le numéro de la classe d'équivalence de l'élément i.

P = { {1,3} , {4} , {2,7,9}, {5,6,8,10} }

classe 1 2 3 4

1	2	3	4	5	6	7	8	9	10
1	**3**	**1**	**2**	**4**	**4**	**3**	**4**	**3**	**4**

Find : complexité en O(1)

Union : complexité en O(n)

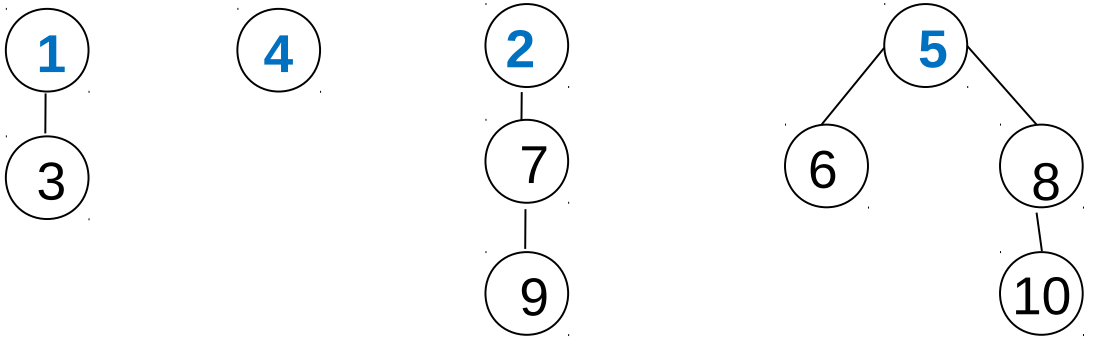
Une meilleure solution

On représente la partition par une forêt.

- Chaque arbre correspond à une classe d'équivalence.
- La racine de chaque arbre est le "représentant" de la classe.

On représente la forêt par un tableau père, tel que père[i] est le père de l'élément i, en posant père[i]=i pour une racine.

P = { {1,3} , {4} , {2,7,9}, {5,6,8,10} }

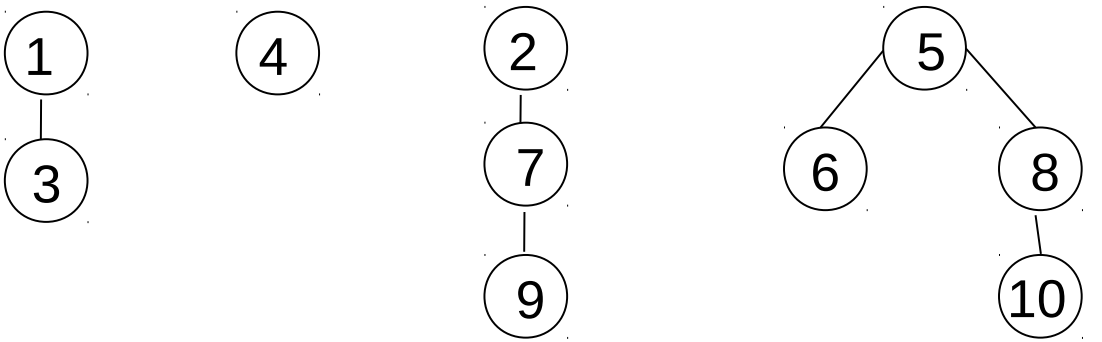


	1	2	3	4	5	6	7	8	9	10
père	**1**	**2**	**1**	**4**	**5**	**5**	**2**	**5**	**7**	**8**

Find

```
fonction Find(entier x): entier;
Tant que (x ≠ père[x])
                x := père[x];
FinTantque
Retourner(x)
```

Exemple: Find(9) P = { {1,3} , {4} , {2,7,9}, {5,6,8,10} }



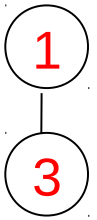
	1	2	3	4	5	6	7	8	9	10
père	**1**	**2**	**1**	**4**	**5**	**5**	**2**	**5**	**7**	**8**

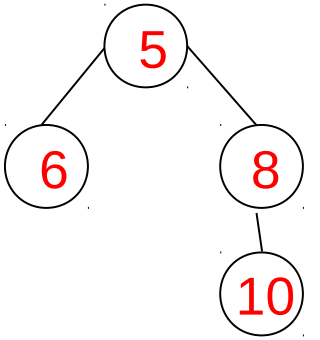
père[9]=7 ; père[7]=2; père[2]=2; retourner(2)

Union

```
procédure Union (entier x, entier y): entier;
r1= Find(x);
r2 = Find(y);
Si (r1 ≠ r2) alors
                père[r2]= r1;
FinSi
```

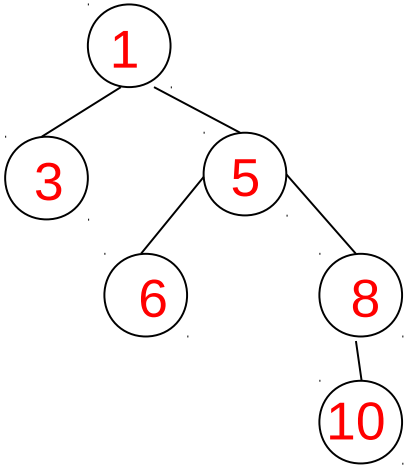
Exemple: Union(3,8) P = { {1,3} , {4} , {2,7,9}, {5,6,8,10} }





Find(3)=1
Find(8)=5

père[5]=1

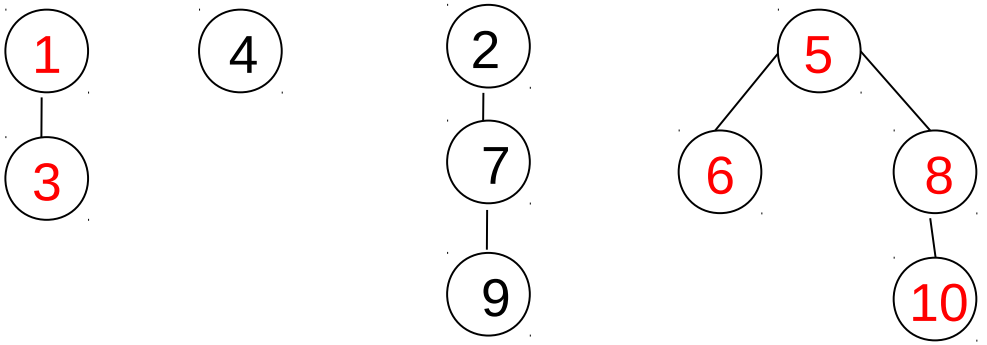


Union pondérée

L'opération Find s'effectue en temps O(h), où h est la hauteur de l'arbre. Pire cas : h = O(n)

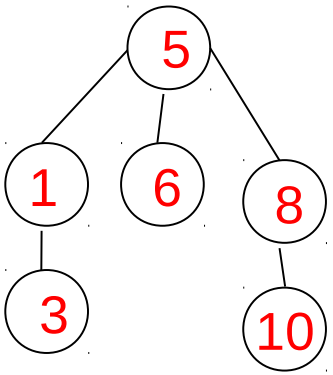
Lors de l'union de deux arbres, la racine de l'arbre avec le moins de sommets devient fils de la racine de l'autre.

Exemple: Union(3,8) P = { {1,3} , {4} , {2,7,9}, {5,6,8,10} }



Find(3)=1
Find(8)=5

père[1]=5



Union pondérée

On maintient un tableau taille (initialisé à 1 si l'on part de classes singleton) : si i est une racine, taille[i] est le nombre de sommets de l'arbre de racine i.

```
procédure Union (entier x, entier y): entier;
r1= Find(x);
r2 = Find(y);
Si (r1 ≠ r2) alors
                Si (taille[r1])>taille[r2]) alors
                        père[r2] := r1;
                        taille[r1] :=taille[r1]+taille[r2]
                sinon
                        père[r1] := r2;
                        taille[r2] :=taille[r1]+taille[r2]
                FinSi
FinSi
```

Union pondérée

La hauteur d'un arbre à n sommets créé par une suite d'unions pondérées est $\leq 1 + \lfloor \log_2(n) \rfloor$

Preuve: Par récurrence sur n

- *Cas de base :* vrai pour n=1.

- *Induction :*

Soit T un arbre obtenu par union pondérée d'un arbre à x sommets (avec $1 \leq x \leq n/2$) et d'un arbre à (n-x) sommets.
Hauteur(T) $\leq \max(1 + \lfloor \log_2(n-x) \rfloor, 2 + \lfloor \log_2(x) \rfloor)$.

Or $\log_2(n-x) \leq \log_2(n)$
et $\log_2(x) \leq \log_2(n/2) \leq \log_2(n)-1$
La hauteur de T est donc majorée par $1 + \lfloor \log_2(n) \rfloor$

Union et Find sont donc en O(log n)

Complexité de l'algorithme de Kruskal

Algorithme de Kruskal:
  Trier les arêtes par coût croissant;
  H = arbre vide;
  Examiner dans l'ordre chacune des arêtes {x,y} :
  S'il n'existe pas de chaîne de x à y dans H :
    H = H U {x,y}
  FinSi

Union et Find sont en O(log n)

Complexité de Kruskal : O(m log m + m log n) = O(m log m)

Et si les coûts des arêtes sont faibles (en O(m)), de façon à ce que Trier(ensemble des arêtes) se fasse en O(m)?
On cherche à améliorer la complexité de Union et Find.

Compression de chemins

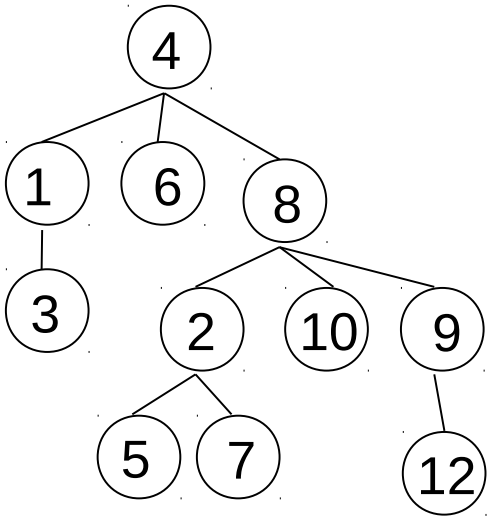
Quand on remonte du sommet x vers sa racine r, on refait le parcours en faisant de chaque sommet rencontré un fils de r.

```
fonction Findsimple(x): entier;
Tant que (x ≠ père[x])
                x := père[x];
FinTantque
Retourner(x)
```

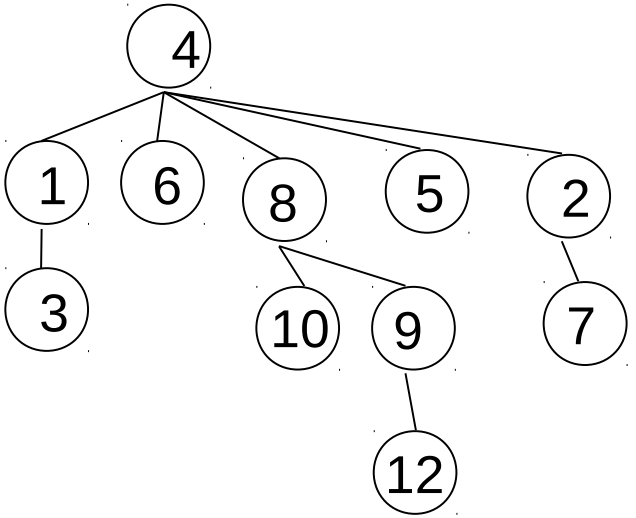
```
fonction Find(entier x): entier;
r :=  Findsimple(x);
Tant que (x ≠ père[x])
                y := père[x]
                père[x] := r;
                x := y;
FinTantque
Retourner(x)
```

Exemple : Find(5)

Arbre initial :



Arbre final :



Complexité de Union et Find?

Complexité amortie : n opérations Union + m opérations Find se réalisent en temps $O(n + m\ \alpha(n,m))$, où α est une fonction qui croît très très lentement.

En effet, $\alpha(n,m)$ est la réciproque de la fonction d'Ackermann qui croît extrêmement vite: on a $\alpha(n,m) \leq 4$ pour $n,m \leq 2^{2048}$.
En pratique, $\alpha(n,m)$ est donc une constante.

Complexité de l'algorithme de Kruskal?

Si on suppose que Tri(arêtes) est effectué en temps O(m).

On a alors n Union et m Find sur n éléments : ceci se fait en $O(n + m\ \alpha(n,m))$.

Rappel: dans le cas général la complexité est en $O(m \log m)$.