L3 Informatique
Année 2021 – 2022

UE ALGO5 — TD2 — Séance 10 : Union-Find

Objectifs

À la fin de cette séance, vous devriez être capable de :
- choisir des structures de données adaptées aux spécifications ;
- calculer des complexités en moyenne, pondérées, du meilleur et pire cas.

Exercice 1.

Soit un ensemble $E = \{1, \dots, n\}$. On souhaite implémenter une structure permettant de gérer des *partitions* de E.

Rappel : $\{P_1, \dots, P_n\}$ est une partition de E si :
- $P_i \neq \emptyset$
- $\bigcup_i P_i = E$
- $i \neq j \Rightarrow P_i \cap P_j = \emptyset$.

On spécifie un type id permettant d'identifier les sous-ensembles, ainsi que les primitives suivantes :

> id : type abstrait
>
> Initialiser
> *{Après l'appel à Initialiser, la structure contient la partition P_0 où chaque élément de E est l'élément unique d'un ensemble de P_0 : $P_0 = \{\{1\}, \dots, \{n\}\}$ }*
>
> Find : entier → id
> *{Find(x) renvoie l'identifiant du sous-ensemble auquel appartient x. Find(x)=Find(y) ssi x et y appartiennent au même sous-ensemble. }*
>
> Union(données x,y : entiers)
> *{ Union(x,y) réalise l'*union *des deux sous-ensembles auxquels appartiennent x et y. Après cet appel, Find(x)=Find(y)}*

On souhaite implémenter de manière efficace les primitives Union et Find. Pour cela, on utilise un tableau P de n entiers représentant une *forêt*, dans laquelle chaque *arbre* contient les éléments d'un même sous-ensemble. P(x) est défini comme étant le parent de x ; un sous-ensemble est représenté par l'élément à la racine de l'arbre.

> P : tableau sur [1..n] d'entiers sur 1..n

Q 1. Dessiner la structure P et les arbres correspondants à l'initialisation, puis après plusieurs appels à Union.

Corrigé —

Après initialisation, P est représenté par une forêt de nœuds (sans enfants) :

$P =$

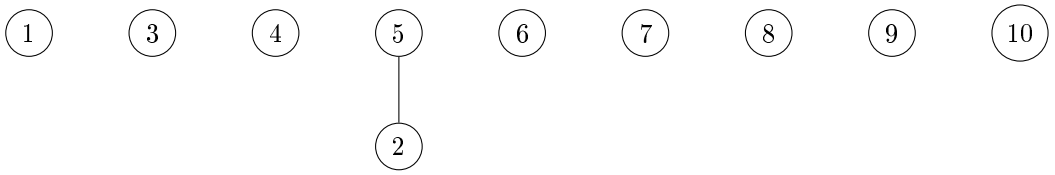
1	2	3	4	5	6	7	8	9	10
1	2	3	4	5	6	7	8	9	10

Après appel à Union, la structure dépend de l'implémentation choisie. Un exemple :

```
1 Union(2,5)
```

$P =$

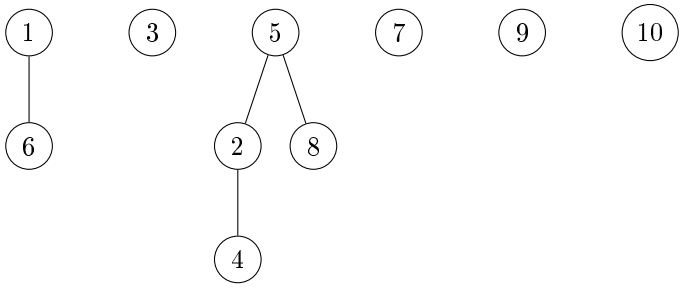
1	2	3	4	5	6	7	8	9	10
1	5	3	4	5	6	7	8	9	10




```
Union(4,2)
Union(6,1)
Union(8,5)
```

$P =$

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



Q 2. Réaliser l'opération d'initialisation, puis les primitives Union et Find. Quel est le coût de ces deux opérations ?

Corrigé —

L'identifiant d'un ensemble de la partition correspond à l'élément à la racine de l'arbre représentant cet ensemble.

Initialiser crée n nœuds pointant sur eux-mêmes (donc n arbres comportant un seul nœud). On identifie par la suite la racine r d'un arbre au fait que P[r]=r.

```
Initialiser:
    pour i de 1 à n
        P[i] <- i

Find(x):
    si P[x] = x alors
        retourner x
    sinon
        retourner Find(P[x])

Union(x,y):
    { on relie la racine de l'arbre contenant x à y }
    px <- Find(x)
    P[px] <- y
```

Dans le pire cas, on va créer un arbre réduit à une liste : les deux opérations sont en $O(n)$ (coût de recherche de la racine).

Q 3. Proposer puis implémenter une méthode permettant d'améliorer le coût des deux opérations.

Corrigé —

L'idée est d'essayer de minimiser la hauteur des arbres construits. On peut identifier deux heuristiques :
—«*union by rank*» : on conserve, pour chaque arbre, la hauteur de l'arbre. Lorsqu'on réalise une union, on relie la racine de l'arbre le moins haut à la racine de l'arbre le plus haut.

```
{ h[x] contient la hauteur de l'arbre de racine x }
Initialiser:
    pour i de 1 à n
        P[i] <- i
        h[i] <- 0

Union(x,y):
    { on relie la racine de l'arbre contenant x à y }
    px <- Find(x)
    py <- Find(y)
    si h[px] > h[py] alors
        P[py] <- px
    sinon
        P[px] <- py
        { cas particulier : arbres de hauteurs égales }
        si h[px] = h[py] alors
            h[py] <- h[py] + 1
```

—«*path compression*» : lors de chaque opération Find(x), on relie le nœud x à la racine de l'arbre

```
Find(x):
    si P[x] = x alors
        retourner x
    sinon
        P[x] <- Find(P[x])
        retourner P[x]
```
