Algorithme 2

Files de priorités et ensembles disjoints

Louis-Claude Canon
louis-claude.canon@univ-fcomte.fr

Licence 2 Informatique – Semestre 4

Plan

Plan

Opérations

- On se sert souvent d'un tas comme une base pour construire une *file de priorités*.
- On aura 4 opérations supplémentaires :

  MAXIMUM-TAS retourne l'élément ayant la clé maximale ;
  EXTRAIRE-MAX-TAS supprime et retourne l'élément qui a la clé maximale ;
  AUGMENTER-CLÉ-TAS accroît la valeur de la clé d'un élément ;
  INSÉRER-TAS-MAX insère un élément dans la structure.

Contextes d'utilisation concrets

La file de priorités est idéale pour gérer des éléments dont les priorités changent dynamiquement. Par exemple :

- Des tâches, avec des priorités, à planifier sur un ordinateur : lorsqu'une tâche est terminée, on exécute ensuite la tâche de plus forte priorité.
- Des événements dans un simulateur de pannes d'équipements : la priorité de chaque événement dépend de son instant.
- Des sommets d'un graphe dans la construction d'un arbre couvrant ou la recherche d'un plus court chemin.

Pseudo-code de MAXIMUM-TAS et de EXTRAIRE-MAX-TAS

MAXIMUM-TAS(A)

si $A.n < 1$ **alors**
  erreur "limite inférieure dépassée"
retourner $A[1]$

EXTRAIRE-MAX-TAS(A)

$max \leftarrow$ **Maximum-Tas**(A)
$A[1] \leftarrow A[A.n]$
$A.n \leftarrow A.n - 1$
ENTASSER-MAX($A, 1$)
retourner max

- La complexité en temps de MAXIMUM-TAS est $\Theta(1)$.
- La complexité en temps de EXTRAIRE-MAX-TAS est celle de ENTASSER-MAX : $O(\log n)$.

Pseudo-code de AUGMENTER-CLÉ-TAS

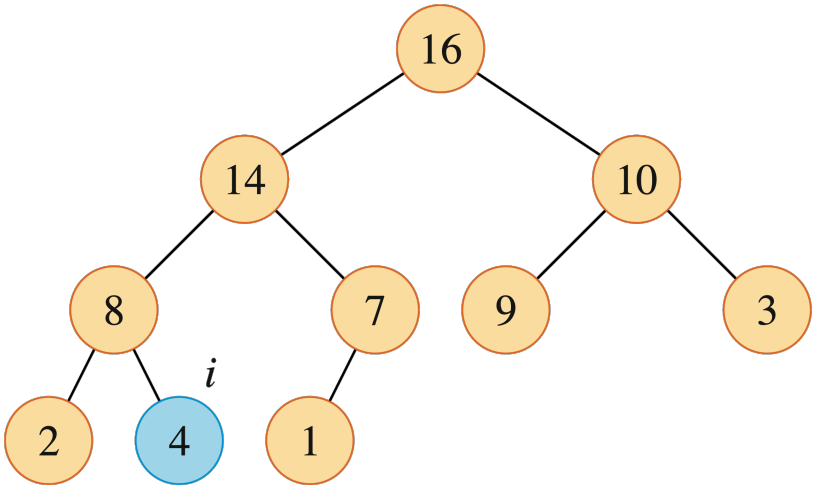
- Cette opération accroît la valeur de la clé d'indice i pour lui donner la nouvelle valeur k.
- On fait remonter l'élément dans le tas tant qu'il est supérieur à son parent.
- La complexité en temps est en $O(\log n)$.

AUGMENTER-CLÉ-TAS(A, i, k)

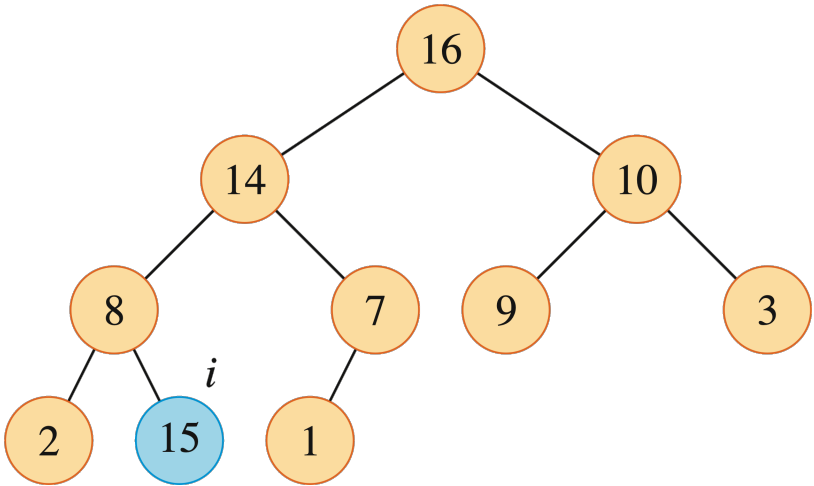
---

si $k < A[i]$ **alors**
  erreur "nouvelle clé plus petite que clé actuelle"
$A[i] \leftarrow k$
tant que $i > 1$ **et** $A[\text{PARENT}(i)] < A[i]$ **alors**
  échanger $A[i]$ et $A[\text{PARENT}(i)]$
  $i \leftarrow \text{PARENT}(i)$

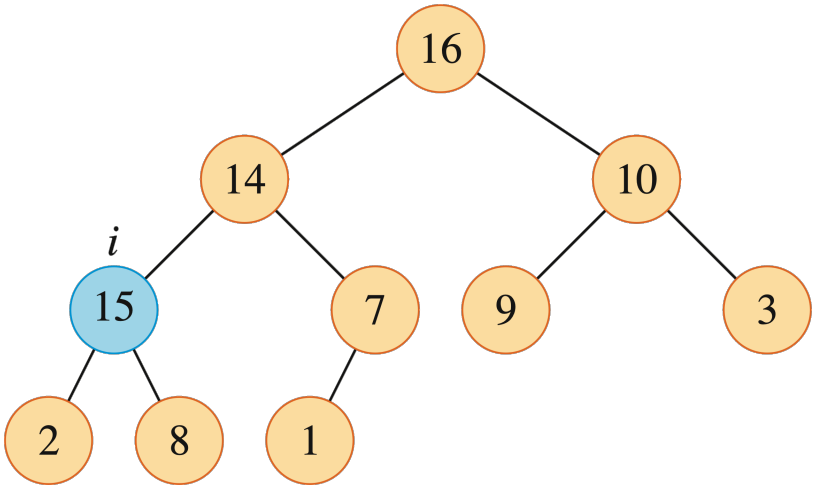
Exemple



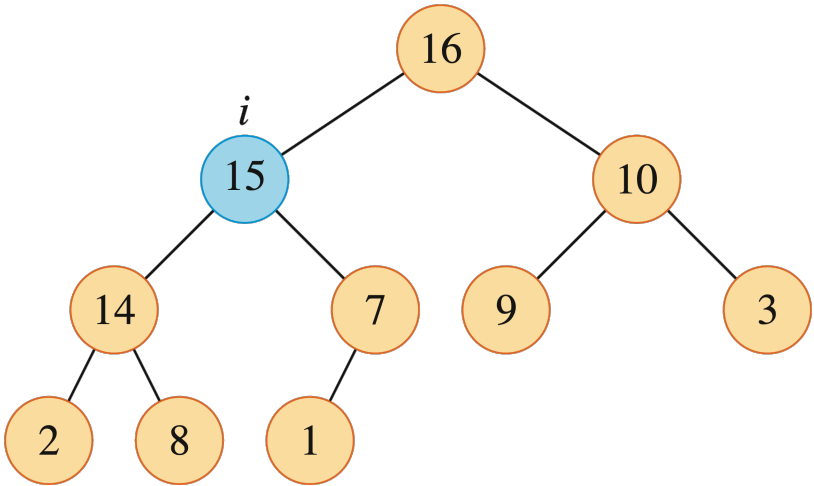
Exemple



Exemple



Exemple



Pseudo-code de Insérer-Tas-Max

- Cette opération insère un élément de valeur k.
- On le met à la fin du tableau avec une valeur d'abord minimale : $-\infty$.
- On fait appel à Augmenter-Clé-Tas pour mettre la bonne valeur.
- La complexité en temps est celle de Augmenter-Clé-Tas : $O(\log n)$.

Insérer-Tas-Max(A, k)

$A.n \leftarrow A.n + 1$
$A[A.n] \leftarrow -\infty$
Augmenter-Clé-Tas$(A, A.n, k)$

Complexité amortie

Notion algorithmique

- L'analyse *amortie* diffère de la complexité au pire cas en considérant une succession d'opérations plutôt qu'une seule.
- Par exemple, on pourrait réorganiser une structure de données en temps $O(\log n)$ toutes les n insertions : on aurait alors une insertion sur n qui serait plus coûteuse.
- La complexité classique considère le pire cas sans l'amortir sur l'ensemble des opérations réalisées, alors que le temps cumulé rapporté sur chaque opération peut être meilleur.
- La complexité amortie représente le coût moyen par opération dans le pire cas.
- Elle a surtout du sens pour une structure de données sur laquelle on réalise plusieurs opérations successives.

Complexité amortie

Notion algorithmique

- L'analyse *amortie* diffère de la complexité au pire cas en considérant une succession d'opérations plutôt qu'une seule.
- Par exemple, on pourrait réorganiser une structure de données en temps $O(\log n)$ toutes les n insertions : on aurait alors une insertion sur n qui serait plus coûteuse.
- La complexité classique considère le pire cas sans l'amortir sur l'ensemble des opérations réalisées, alors que le temps cumulé rapporté sur chaque opération peut être meilleur.
- La complexité amortie représente le coût moyen par opération dans le pire cas.
- Elle a surtout du sens pour une structure de données sur laquelle on réalise plusieurs opérations successives.

Complexité amortie

Notion algorithmique

- L'analyse *amortie* diffère de la complexité au pire cas en considérant une succession d'opérations plutôt qu'une seule.
- Par exemple, on pourrait réorganiser une structure de données en temps $O(\log n)$ toutes les n insertions : on aurait alors une insertion sur n qui serait plus coûteuse.
- La complexité classique considère le pire cas sans l'amortir sur l'ensemble des opérations réalisées, alors que le temps cumulé rapporté sur chaque opération peut être meilleur.
- La complexité amortie représente le coût moyen par opération dans le pire cas.
- Elle a surtout du sens pour une structure de données sur laquelle on réalise plusieurs opérations successives.

Complexité amortie

Notion algorithmique

- L'analyse *amortie* diffère de la complexité au pire cas en considérant une succession d'opérations plutôt qu'une seule.
- Par exemple, on pourrait réorganiser une structure de données en temps $O(\log n)$ toutes les n insertions : on aurait alors une insertion sur n qui serait plus coûteuse.
- La complexité classique considère le pire cas sans l'amortir sur l'ensemble des opérations réalisées, alors que le temps cumulé rapporté sur chaque opération peut être meilleur.
- La complexité amortie représente le coût moyen par opération dans le pire cas.
- Elle a surtout du sens pour une structure de données sur laquelle on réalise plusieurs opérations successives.

Complexité amortie

Notion algorithmique

- L'analyse *amortie* diffère de la complexité au pire cas en considérant une succession d'opérations plutôt qu'une seule.
- Par exemple, on pourrait réorganiser une structure de données en temps $O(\log n)$ toutes les n insertions : on aurait alors une insertion sur n qui serait plus coûteuse.
- La complexité classique considère le pire cas sans l'amortir sur l'ensemble des opérations réalisées, alors que le temps cumulé rapporté sur chaque opération peut être meilleur.
- La complexité amortie représente le coût moyen par opération dans le pire cas.
- Elle a surtout du sens pour une structure de données sur laquelle on réalise plusieurs opérations successives.

Récapitulatif et structures de données associées

Opération	Tas (binaire)	Tas de Fibonacci
Maximum-Tas	$\Theta(1)$	$\Theta(1)$
Extraire-Max-Tas	$O(\log n)$	$O(\log n)^{\text{am}}$
Augmenter-Clé-Tas	$O(\log n)$	$\Theta(1)^{\text{am}}$
Insérer-Tas-Max	$O(\log n)$	$\Theta(1)$

Curiosité

- Le tas binomial (1978) améliore le tas (1964) grâce à l'insertion en $\Theta(1)$ en amortie.
- Le tas de Fibonacci (1984) permet l'augmentation de clé en $\Theta(1)$ en amortie.
- Les tas de Brodal (1996) et de Fibonacci strict (2012) obtiennent ces complexités, qui sont asymptotiquement optimales, sans analyse amortie.

Question

Quel est le tas max issu de l'action INSÉRER-TAS-MAX sur le tableau $(15, 13, 9, 5, 12, 8, 7, 4, 0, 6, 2, 1)$ pour l'élément 10 ?

1. $(15, 13, 9, 5, 10, 8, 7, 4, 0, 6, 12, 2, 1)$
2. $(15, 13, 9, 10, 12, 8, 7, 4, 0, 6, 2, 1, 5)$
3. $(15, 13, 10, 5, 12, 9, 7, 4, 0, 6, 2, 1, 8)$
4. $(15, 13, 9, 5, 12, 10, 7, 4, 0, 6, 2, 1, 8)$

Question

Quel est le tas max issu de l'action INSÉRER-TAS-MAX sur le tableau $(15, 13, 9, 5, 12, 8, 7, 4, 0, 6, 2, 1)$ pour l'élément 10 ?

1. $(15, 13, 9, 5, 10, 8, 7, 4, 0, 6, 12, 2, 1)$
2. $(15, 13, 9, 10, 12, 8, 7, 4, 0, 6, 2, 1, 5)$
3. $(15, 13, 10, 5, 12, 9, 7, 4, 0, 6, 2, 1, 8)$ ✓
4. $(15, 13, 9, 5, 12, 10, 7, 4, 0, 6, 2, 1, 8)$

Plan

Structure de données pour ensembles disjoints

- On gère une famille de k ensembles disjoints $V_1 \cup V_2 \cup \ldots \cup V_k$.
- Alternativement, on peut dire que n éléments sont partitionnés en k ensembles disjoints.
- Par exemple : $V_1 = \{2, 3, 5, 7\}$ et $V_2 = \{1, 4, 8, 9\}$ avec $n = 8$.
- Le problème consiste à déterminer si 2 éléments appartiennent au même ensemble. Par exemple, 5 et 7 font partis du même ensemble, alors que 3 et 4 sont dans des ensembles disjoints.
- Certains algorithmes sur les graphes (arbre couvrant par exemple) s'appuient sur cette structure de données.

Structure de données pour ensembles disjoints

- On gère une famille de k ensembles disjoints $V_1 \cup V_2 \cup \ldots \cup V_k$.
- Alternativement, on peut dire que n éléments sont partitionnés en k ensembles disjoints.
- Par exemple : $V_1 = \{2, 3, 5, 7\}$ et $V_2 = \{1, 4, 8, 9\}$ avec $n = 8$.
- Le problème consiste à déterminer si 2 éléments appartiennent au même ensemble. Par exemple, 5 et 7 font partis du même ensemble, alors que 3 et 4 sont dans des ensembles disjoints.
- Certains algorithmes sur les graphes (arbre couvrant par exemple) s'appuient sur cette structure de données.

Structure de données pour ensembles disjoints

- On gère une famille de k ensembles disjoints $V_1 \cup V_2 \cup \ldots \cup V_k$.
- Alternativement, on peut dire que n éléments sont partitionnés en k ensembles disjoints.
- Par exemple : $V_1 = \{2, 3, 5, 7\}$ et $V_2 = \{1, 4, 8, 9\}$ avec $n = 8$.
- Le problème consiste à déterminer si 2 éléments appartiennent au même ensemble. Par exemple, 5 et 7 font partis du même ensemble, alors que 3 et 4 sont dans des ensembles disjoints.
- Certains algorithmes sur les graphes (arbre couvrant par exemple) s'appuient sur cette structure de données.

Structure de données pour ensembles disjoints

- On gère une famille de k ensembles disjoints $V_1 \cup V_2 \cup \ldots \cup V_k$.
- Alternativement, on peut dire que n éléments sont partitionnés en k ensembles disjoints.
- Par exemple : $V_1 = \{2, 3, 5, 7\}$ et $V_2 = \{1, 4, 8, 9\}$ avec $n = 8$.
- Le problème consiste à déterminer si 2 éléments appartiennent au même ensemble. Par exemple, 5 et 7 font partis du même ensemble, alors que 3 et 4 sont dans des ensembles disjoints.
- Certains algorithmes sur les graphes (arbre couvrant par exemple) s'appuient sur cette structure de données.

Structure de données pour ensembles disjoints

- On gère une famille de k ensembles disjoints $V_1 \cup V_2 \cup \ldots \cup V_k$.
- Alternativement, on peut dire que n éléments sont partitionnés en k ensembles disjoints.
- Par exemple : $V_1 = \{2, 3, 5, 7\}$ et $V_2 = \{1, 4, 8, 9\}$ avec $n = 8$.
- Le problème consiste à déterminer si 2 éléments appartiennent au même ensemble. Par exemple, 5 et 7 font partis du même ensemble, alors que 3 et 4 sont dans des ensembles disjoints.
- Certains algorithmes sur les graphes (arbre couvrant par exemple) s'appuient sur cette structure de données.

Interface

- On considère que chaque ensemble est identifié par un *représentant* qui est l'un des membres de l'ensemble.
- Pour un élément donné, l'opération TROUVER-ENSEMBLE déterminera le représentant de son ensemble.
- Si 2 éléments sont dans le même ensemble, ils ont le même représentant. Sinon, ils ont des représentants distincts.
- Les méthodes CRÉER-ENSEMBLE et UNION initialisent la structure de données.

Opération	description
CRÉER-ENSEMBLE(v)	indique que v est le représentant d'un singleton
UNION(u, v)	fusionne les ensembles contenant u et v
TROUVER-ENSEMBLE(v)	trouve le représentant de l'ensemble contenant v

Interface

- On considère que chaque ensemble est identifié par un *représentant* qui est l'un des membres de l'ensemble.
- Pour un élément donné, l'opération TROUVER-ENSEMBLE déterminera le représentant de son ensemble.
- Si 2 éléments sont dans le même ensemble, ils ont le même représentant. Sinon, ils ont des représentants distincts.
- Les méthodes CRÉER-ENSEMBLE et UNION initialisent la structure de données.

Opération	description
CRÉER-ENSEMBLE(v)	indique que v est le représentant d'un singleton
UNION(u, v)	fusionne les ensembles contenant u et v
TROUVER-ENSEMBLE(v)	trouve le représentant de l'ensemble contenant v

Interface

- On considère que chaque ensemble est identifié par un *représentant* qui est l'un des membres de l'ensemble.
- Pour un élément donné, l'opération TROUVER-ENSEMBLE déterminera le représentant de son ensemble.
- Si 2 éléments sont dans le même ensemble, ils ont le même représentant. Sinon, ils ont des représentants distincts.
- Les méthodes CRÉER-ENSEMBLE et UNION initialisent la structure de données.

Opération	description
CRÉER-ENSEMBLE(v)	indique que v est le représentant d'un singleton
UNION(u, v)	fusionne les ensembles contenant u et v
TROUVER-ENSEMBLE(v)	trouve le représentant de l'ensemble contenant v

Interface

- On considère que chaque ensemble est identifié par un *représentant* qui est l'un des membres de l'ensemble.
- Pour un élément donné, l'opération TROUVER-ENSEMBLE déterminera le représentant de son ensemble.
- Si 2 éléments sont dans le même ensemble, ils ont le même représentant. Sinon, ils ont des représentants distincts.
- Les méthodes CRÉER-ENSEMBLE et UNION initialisent la structure de données.

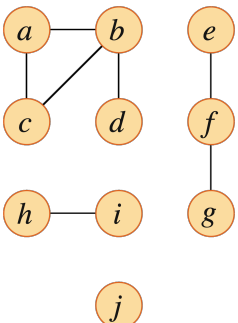
Opération	description
CRÉER-ENSEMBLE(v)	indique que v est le représentant d'un singleton
UNION(u, v)	fusionne les ensembles contenant u et v
TROUVER-ENSEMBLE(v)	trouve le représentant de l'ensemble contenant v

Interface

- On considère que chaque ensemble est identifié par un *représentant* qui est l'un des membres de l'ensemble.
- Pour un élément donné, l'opération TROUVER-ENSEMBLE déterminera le représentant de son ensemble.
- Si 2 éléments sont dans le même ensemble, ils ont le même représentant. Sinon, ils ont des représentants distincts.
- Les méthodes CRÉER-ENSEMBLE et UNION initialisent la structure de données.

Opération	description
CRÉER-ENSEMBLE(v)	indique que v est le représentant d'un singleton
UNION(u, v)	fusionne les ensembles contenant u et v
TROUVER-ENSEMBLE(v)	trouve le représentant de l'ensemble contenant v

Exemple d'application : composantes connexes



- Avant d'étudier les algorithmes de la structure de données, on illustre son utilisation dans le cadre d'un algorithme donné.
- Il s'agit de déterminer les composantes connexes d'un graphe non orienté.
- Une *composante connexe* est un ensemble de sommets entre lesquels il existe un chemin.
- Dans cet exemple, il y a 4 composantes connexes : $\{a, b, c, d\}$, $\{e, f, g\}$, $\{h, i\}$ et $\{j\}$.

Pseudo-code de Composantes-Connexes

Composantes-Connexes(G)

pour chaque $v \in G.V$ **faire**
  Créer-Ensemble(v)
pour chaque $(u, v) \in G.E$ **faire**
  Union(u, v)

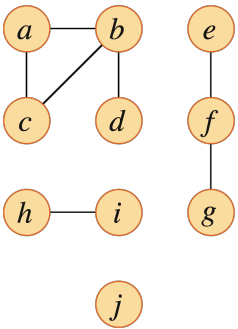
Initialisation de la structure de données :

- un singleton par sommet ;
- union des ensembles pour chaque arête.

Même-Composante(u, v)

retourner Trouver-Ensemble(u) = Trouver-Ensemble(v)

Exemple



Edge processed	Collection of disjoint sets									
initial sets	$\{a\}$	$\{b\}$	$\{c\}$	$\{d\}$	$\{e\}$	$\{f\}$	$\{g\}$	$\{h\}$	$\{i\}$	$\{j\}$
(b,d)	$\{a\}$	$\{b,d\}$	$\{c\}$		$\{e\}$	$\{f\}$	$\{g\}$	$\{h\}$	$\{i\}$	$\{j\}$
(e,f)	$\{a\}$	$\{b,d\}$	$\{c\}$		$\{e,f\}$		$\{g\}$	$\{h\}$	$\{i\}$	$\{j\}$
(a,c)	$\{a,c\}$	$\{b,d\}$			$\{e,f\}$		$\{g\}$	$\{h\}$	$\{i\}$	$\{j\}$
(h,i)	$\{a,c\}$	$\{b,d\}$			$\{e,f\}$		$\{g\}$	$\{h,i\}$		$\{j\}$
(a,b)	$\{a,b,c,d\}$				$\{e,f\}$		$\{g\}$	$\{h,i\}$		$\{j\}$
(f,g)	$\{a,b,c,d\}$				$\{e,f,g\}$			$\{h,i\}$		$\{j\}$
(b,c)	$\{a,b,c,d\}$				$\{e,f,g\}$			$\{h,i\}$		$\{j\}$

Question

On exécute la procédure COMPOSANTES-CONNEXES sur le graphe non orienté $G = (V, E)$, où $S = \{a, b, c, d, e, f, g, h, i, j, k\}$ et les arêtes de E sont traitées dans l'ordre suivant : $(d, i), (f, k), (g, i), (b, g), (a, h), (i, j), (d, k), (b, j), (d, f), (g, j), (a, e), (i, d)$. Quelles sont les tailles des sous-ensembles disjoints identifiés ?

1. 7, 3 et 1
2. 5, 3 et 3
3. 8 et 3
4. 7 et 4

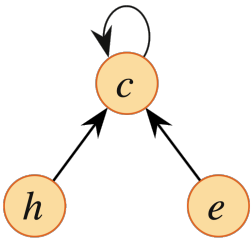
Question

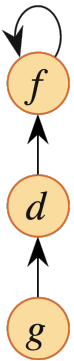
On exécute la procédure COMPOSANTES-CONNEXES sur le graphe non orienté $G = (V, E)$, où $S = \{a, b, c, d, e, f, g, h, i, j, k\}$ et les arêtes de E sont traitées dans l'ordre suivant : $(d, i), (f, k), (g, i), (b, g), (a, h), (i, j), (d, k), (b, j), (d, f), (g, j), (a, e), (i, d)$. Quelles sont les tailles des sous-ensembles disjoints identifiés ?

1. 7, 3 et 1 ✓
2. 5, 3 et 3
3. 8 et 3
4. 7 et 4

Plan

Explication de l'algorithme simplifié





- Une structure commune et efficace consiste à gérer une forêt d'ensembles disjoints (aussi appelé *union-find*), c'est-à-dire plusieurs arbres déconnectés.
- Chaque ensemble est ainsi un arbre et on ne représente que le parent pour chaque élément (il faut donc un pointeur par élément).
- Le représentant d'un ensemble est l'élément qui est son propre parent.

Pseudo-code simplifié

CRÉER-ENSEMBLE-SIMPLIFIÉ(x)

$x.parent \leftarrow x$

UNION-SIMPLIFIÉ(x, y)

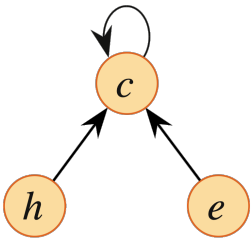
$u \leftarrow$ TROUVER-ENSEMBLE-SIMPLIFIÉ(x)
$v \leftarrow$ TROUVER-ENSEMBLE-SIMPLIFIÉ(y)
$v.parent \leftarrow u$

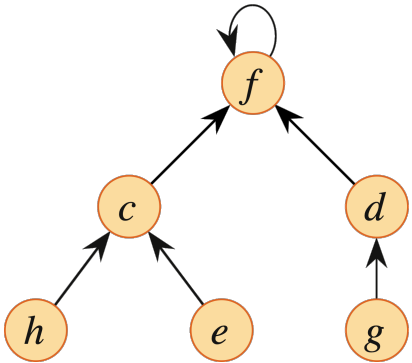
TROUVER-ENSEMBLE-SIMPLIFIÉ(x)

si $x = x.parent$ **alors**
  retourner x
retourner TROUVER-ENSEMBLE-SIMPLIFIÉ($x.parent$)

Union par rang

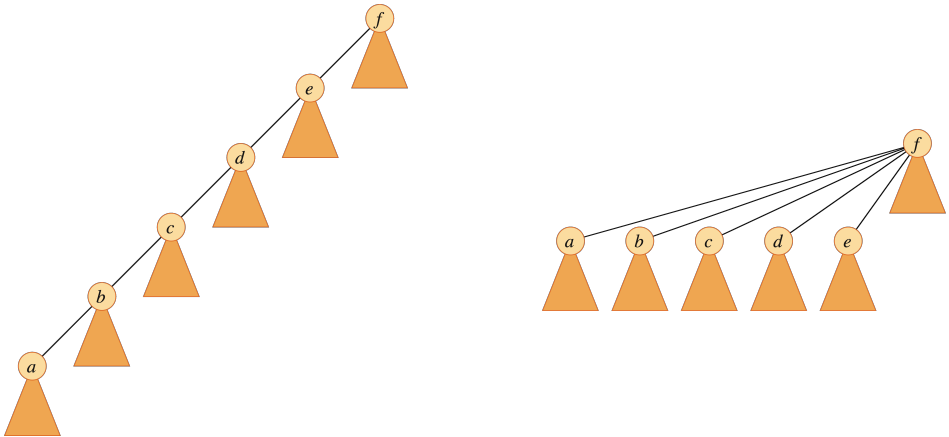
- Pour chaque élément, on gère un *rang* qui majore de la hauteur du sommet.
- Lors de l'union, on choisit comme parent l'élément qui a le plus haut rang.





Compression de chemin

Lors de l'opération TROUVER-ENSEMBLE, on fait pointer le parent de chaque sommet parcouru directement vers la racine.



Pseudo-code final

CRÉER-ENSEMBLE(x)

$x.parent \leftarrow x$
$x.rang \leftarrow 0$

TROUVER-ENSEMBLE(x)

si $x \neq x.parent$ **alors**
  $x.parent \leftarrow$ TROUVER-ENSEMBLE($x.parent$)
retourner $x.parent$

UNION(x, y)

$u \leftarrow$ TROUVER-ENSEMBLE(x)
$v \leftarrow$ TROUVER-ENSEMBLE(y)
si $u \neq v$ **alors**
  si $u.rang > v.rang$ **alors**
    $v.parent \leftarrow u$
  sinon
    $u.parent \leftarrow v$
    si $u.rang = v.rang$ **alors**
      $v.rang \leftarrow v.rang + 1$

Analyse de complexité

- Hormis les requêtes TROUVER-ENSEMBLE, CRÉER-ENSEMBLE et UNION ont une complexité en temps en $\Theta(1)$.
- Le coût amorti pour une requête TROUVER-ENSEMBLE est $O(\alpha(n))$ où $\alpha(n)$ est l'inverse de la fonction d'Ackermann ($\alpha(n) \leq 4$ dans l'univers).

Curiosité

Cette structure de données est optimale : il n'existe pas de structure de données ayant une meilleure complexité en temps.

Récapitulatif

Opération	Complexité en temps
CRÉER-ENSEMBLE(v)	$\Theta(1)$
UNION(u, v)	$O(\alpha(n))$
TROUVER-ENSEMBLE(v)	$O(\alpha(n))$

Question

On exécute la procédure COMPOSANTES-CONNEXES sur le graphe non orienté $G = (V, E)$, où $S = \{a, b, c, d, e, f, g, h, i, j, k\}$ et les arêtes de E sont traitées dans l'ordre suivant : $(d, i), (f, k), (g, i), (b, g), (a, h), (i, j), (d, k), (b, j), (d, f), (g, j), (a, e), (i, d)$. Quel est le rang maximal observé en s'appuyant sur une forêt d'ensembles disjoints ?

1. 1
2. 2
3. 3
4. 4

Question

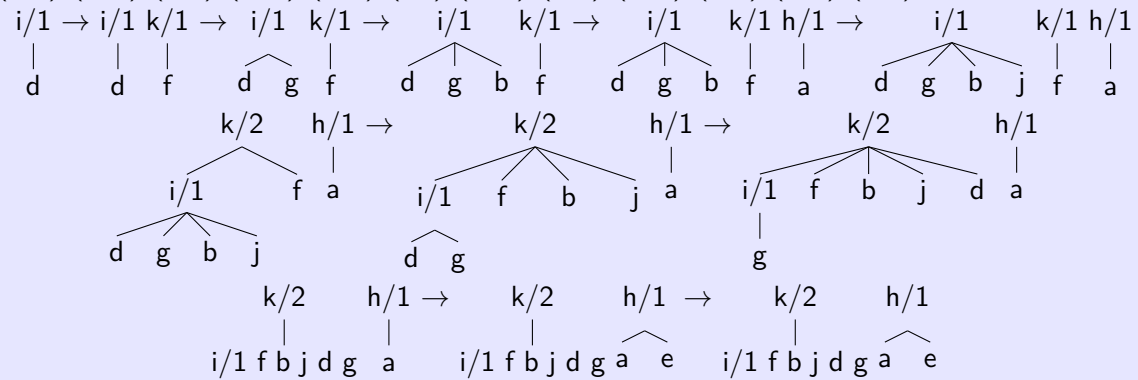
On exécute la procédure COMPOSANTES-CONNEXES sur le graphe non orienté $G = (V, E)$, où $S = \{a, b, c, d, e, f, g, h, i, j, k\}$ et les arêtes de E sont traitées dans l'ordre suivant : $(d, i), (f, k), (g, i), (b, g), (a, h), (i, j), (d, k), (b, j), (d, f), (g, j), (a, e), (i, d)$. Quel est le rang maximal observé en s'appuyant sur une forêt d'ensembles disjoints ?

1. 1
2. 2 ✓$(d \to i \to k)$
3. 3
4. 4

Correction

Parents et rangs lors des unions :

$(d,i),(f,k),(g,i),(b,g),(a,h),(i,j),(d,k),(b,j),(d,f),(g,j),(a,e),(i,d)$.



Plan

Résumé

Contenu

- La structure de tas permet d'implémenter efficacement les opérations nécessaires à une file de priorités et sert de base à d'autres algorithmes.
- La structure pour ensembles disjoints est également utilisée par d'autres algorithmes, notamment pour identifier des composantes connexes.
- L'utilisation d'une forêt offre un mécanisme simple pour déterminer si 2 éléments font partis du même ensemble.