Introduction à l'algorithmique : Structure de données

Serge Haddad

LSV, ENS Paris-Saclay & CNRS & Inria

L3

Type de données abstrait

Un type de données abstrait est spécifié

- syntaxiquement par les opérations autorisées *(facile à formaliser)* ;
- sémantiquement par l'effet de ces opérations *(difficile à formaliser)*.

Illustration. Une pile d'objets permet :

- d'empiler, de dépiler un objet sur une pile, tester si la pile est vide ;
- Empiler renvoie une pile ; Dépiler renvoie une pile et un objet ;
- Une sémantique « formelle »
  1. $Vide(\perp) = \mathbf{true}$; $Vide(Empiler(P, o)) = \mathbf{false}$;
  2. $Dépiler(\perp) = \mathbf{error}$; $Dépiler(Empiler(P, o)) = (P, o)$.

Objectif.

Choisir une structure de données appropriée en vue d'une implémentation efficace.

Plan

Dictionnaire

Un dictionnaire est un ensemble d'enregistrements.

Chaque enregistrement est un couple (clé,valeur).

L'espace des clés est ordonné et de taille trop importante
pour indicer un tableau en mémoire.

Les opérations usuelles sont :

- Insérer(clé,valeur) qui insère un enregistrement
  si la clé n'est pas déjà présente ;
- Modifier(clé,valeur) qui modifie la valeur d'un enregistrement
  si la clé est présente ;
- Supprimer(clé) qui supprime un enregistrement
  si la clé est présente ;
- Chercher(clé) qui renvoie la valeur d'un enregistrement
  si la clé est présente ;

La complexité de Modifier est essentiellement celle de Chercher.

Implémentation d'un dictionnaire

La structure de données doit être *dynamique*.

Les structures usuelles.

- Un tableau qu'on recopie dans un tableau plus grand (resp. plus petit) en cas de débordement (resp. sur-allocation) ;
- Une liste simplement ou doublement chaînée, linéaire ou circulaire, etc. ;
- Un arbre binaire de recherche *équilibré* ;
- Une table de hachage : un tableau de listes dont la fonction de hachage appliquée à la clé fournit l'entrée dans la table.

On étudiera la complexité des opérations en fonction de n,
le nombre courant d'enregistrements.

Les tableaux et les listes

Principe de la recherche.

Parcours séquentiel du tableau ou de la liste. D'où un nombre de comparaisons :

- n au pire ;
- $\frac{n+1}{2}$ en moyenne pour une recherche fructueuse.

Insertion après recherche.

- en temps constant et très simple pour la liste ;
- en temps constant amorti pour le tableau.

Suppression après recherche.

- en temps constant et très simple pour la liste ;
- en temps constant amorti (remplacement par le dernier élément du tableau).

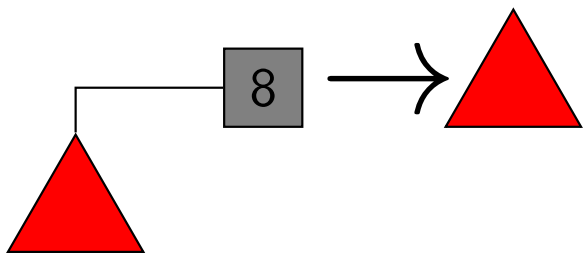
Occupation mémoire.

- optimale pour la liste ;
- et d'un facteur d'accroissement au plus 4 pour le tableau.

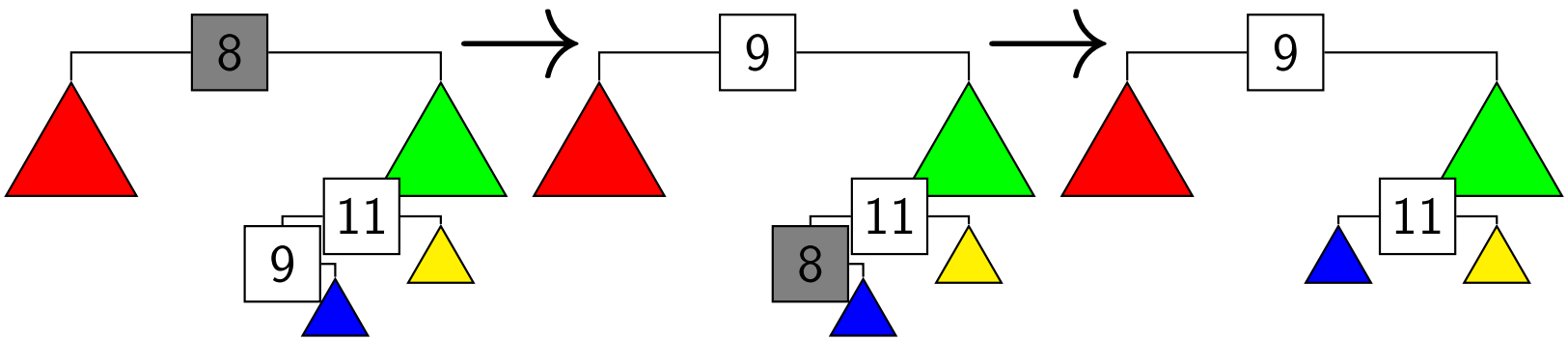
Les arbres binaires de recherche

Toutes les opérations se font en temps linéaire par rapport à la hauteur de l'arbre.
(recherche et insertion déjà vues)

Suppression : le cas particulier d'un fils manquant



Suppression : le cas général se ramène au cas particulier



Minimisation de la hauteur

La hauteur d'un arbre binaire à n sommets est minorée par $\lceil \log(n) \rceil$.

Comment conserver une hauteur en $\Theta(\log(n))$?

Deux stratégies.

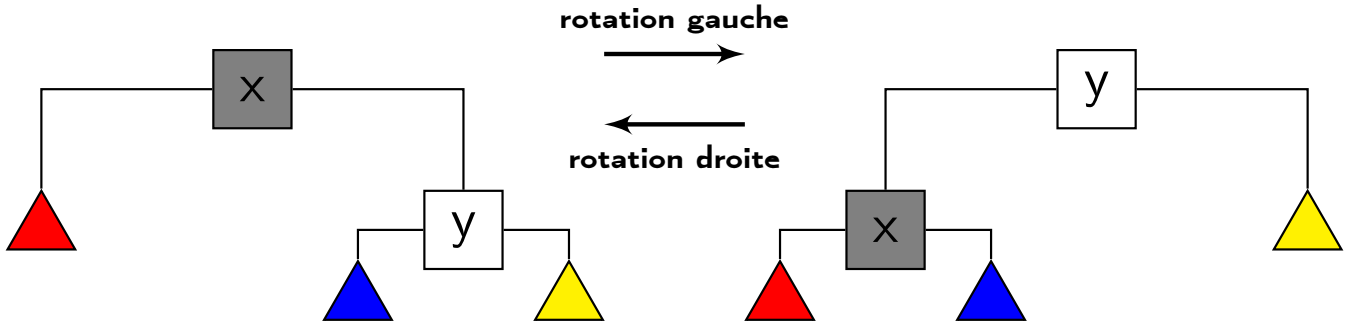
- Les arbres AVL (Adelson-Velskii et Landis 1962) maintiennent un écart de hauteur entre sous-arbres frères d'au plus un.
- Les arbres rouge-noir (Bayer 1972) colorent « suffisamment » de noeuds noirs et requièrent une hauteur « noire » des feuilles uniforme.

Impact sur les opérations d'insertion et de suppression.

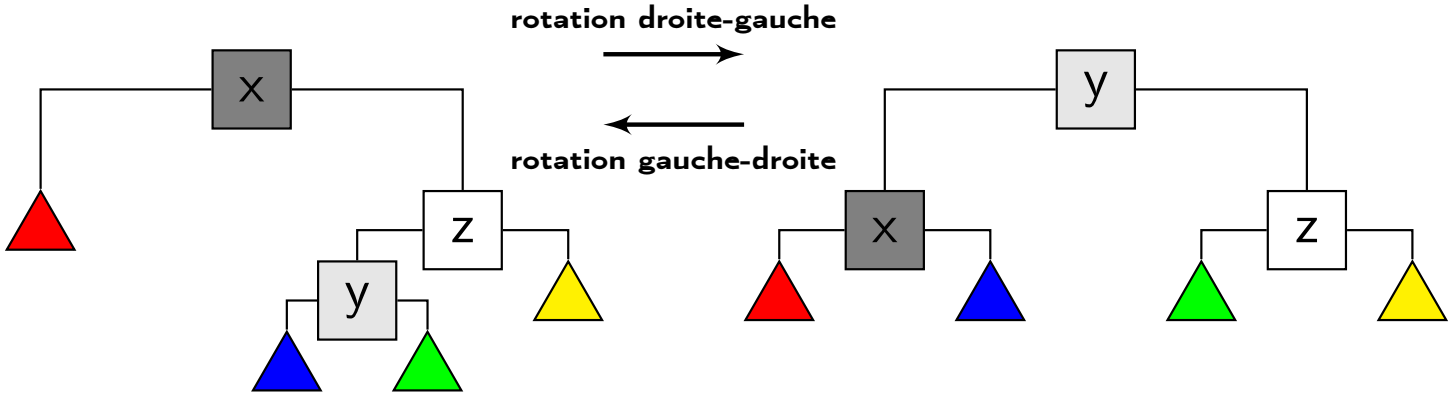
- Rééquilibrage local par techniques de rotation ;
- à éventuellement répéter en remontant vers la racine (d'où un surcoût en $O(\log(n))$.

Rotations d'arbre

Rotations simples.



Rotations doubles.



AVL

Un arbre binaire est un AVL si pour tout sommet,
la difference de hauteur entre sous-arbres gauche et droit est majorée par 1.

- Nécessité de conserver la hauteur de chaque sous-arbre dans sa racine.

La hauteur d'un AVL de n sommets est majorée par $1.44 \log_2(n+2)$.

Preuve.

Soit $N(h)$ le minimum du nombre de sommets d'un AVL de hauteur h.

$$N(0) = 1,\ N(1) = 2,\ N(h) = 1 + N(h-1) + N(h-2)$$

Posons $F(h) = 1 + N(h)$, alors $F(0) = 2$, $F(1) = 3$, $F(h) = F(h-1) + F(h-2)$.

D'où $F(h) = \frac{1}{\sqrt{5}} \left((\frac{1+\sqrt{5}}{2})^{h+3} - (\frac{1-\sqrt{5}}{2})^{h+3} \right) \geq \frac{1}{\sqrt{5}} (\frac{1+\sqrt{5}}{2})^{h+3} - 1.$

Notons $\phi = \frac{1+\sqrt{5}}{2}$. En passant au logarithme :

$$h + 3 \leq \frac{\log_2(N(h)+2)}{\log_2(\phi)} + \log_\phi(\sqrt{5}) \leq 1.44 \log_2(n+2) + 2$$

Insertion dans un AVL (1)

Une fois la clé insérée,

on remonte le chemin vers la racine jusqu'au premier déséquilibre,

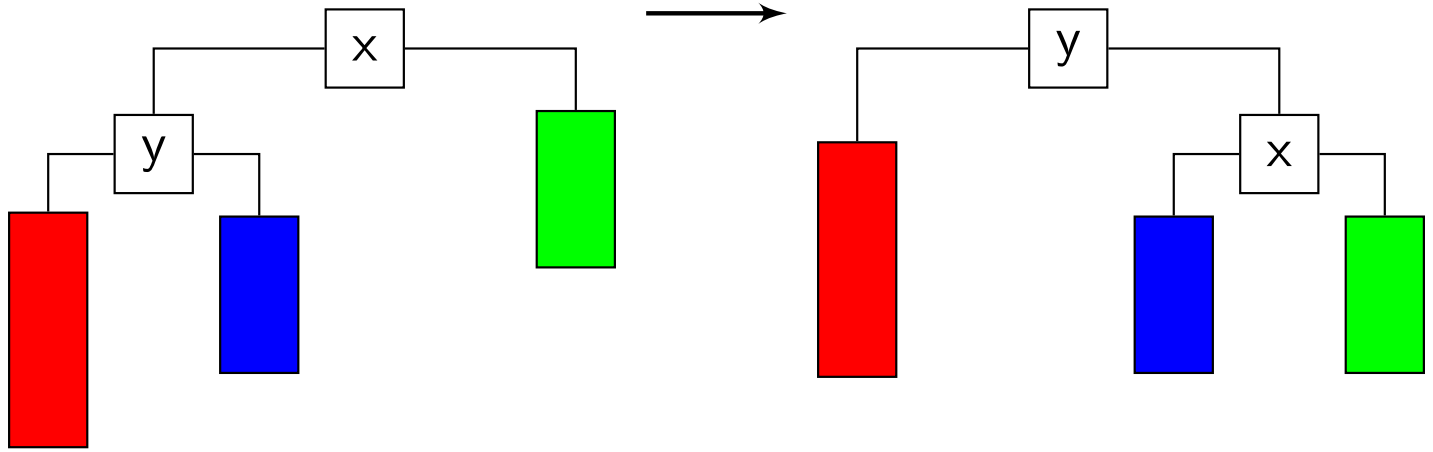
en mettant à jour les hauteurs.

On rééquilibre le sous-arbre.

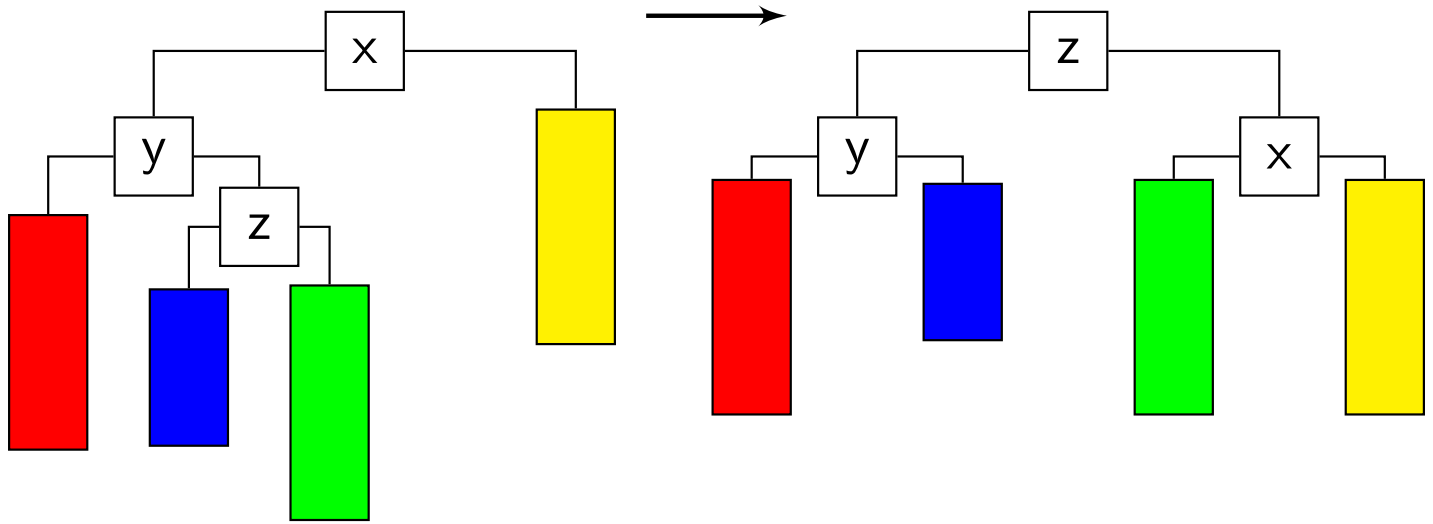
Après rééquilibre du sous-arbre, l'arbre entier est équilibré.

Insertion dans un AVL (2)

Cas 1 : Insertion dans le sous-arbre gauche du sous-arbre gauche.



Cas 2 : Insertion dans le sous-arbre droit du sous-arbre gauche.



Suppression dans un AVL (1)

Dans un AVL, la suppression d'une clé revient à :

- soit supprimer une feuille ;
- soit remplacer un sommet par son unique fils (une feuille).

On remonte donc le chemin vers la racine jusqu'au premier déséquilibre,
en mettant à jour les hauteurs.

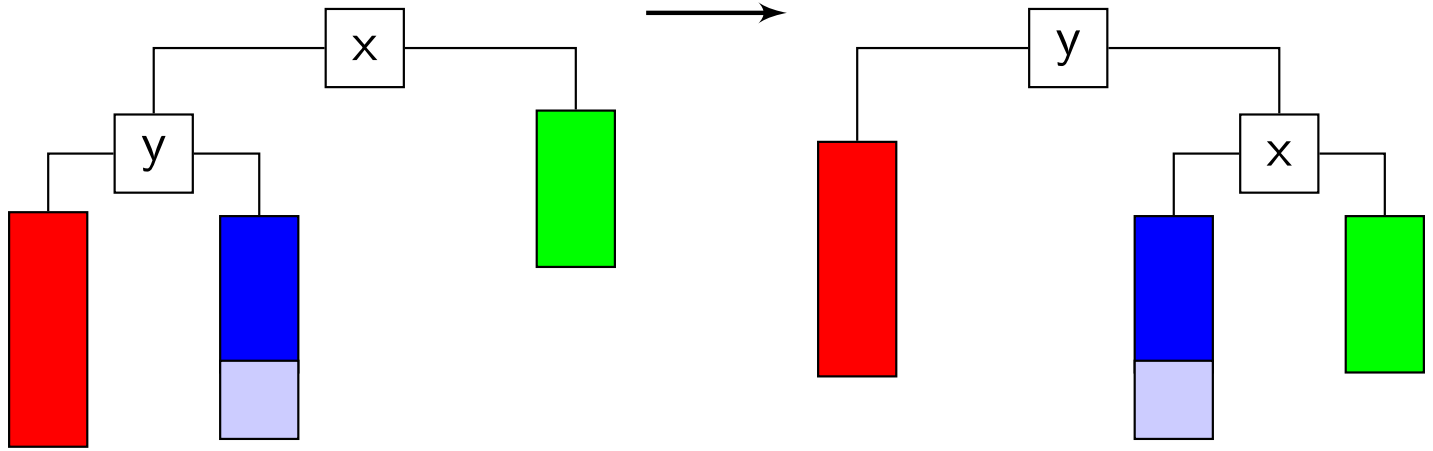
On rééquilibre le sous-arbre.

Après rééquilibre du sous-arbre,
il faut éventuellement continuer à remonter dans l'arbre,
en mettant à jour les hauteurs.

Suppression dans un AVL (2)

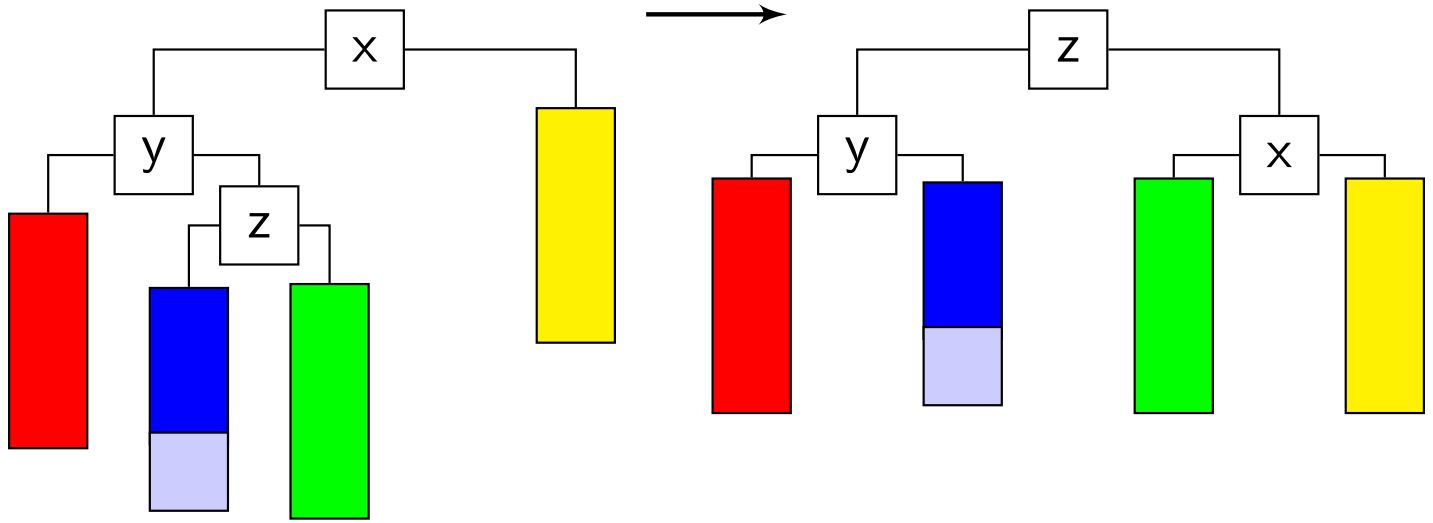
Hauteur excédentaire dans le sous-arbre gauche du sous-arbre gauche.

La hauteur peut diminuer.

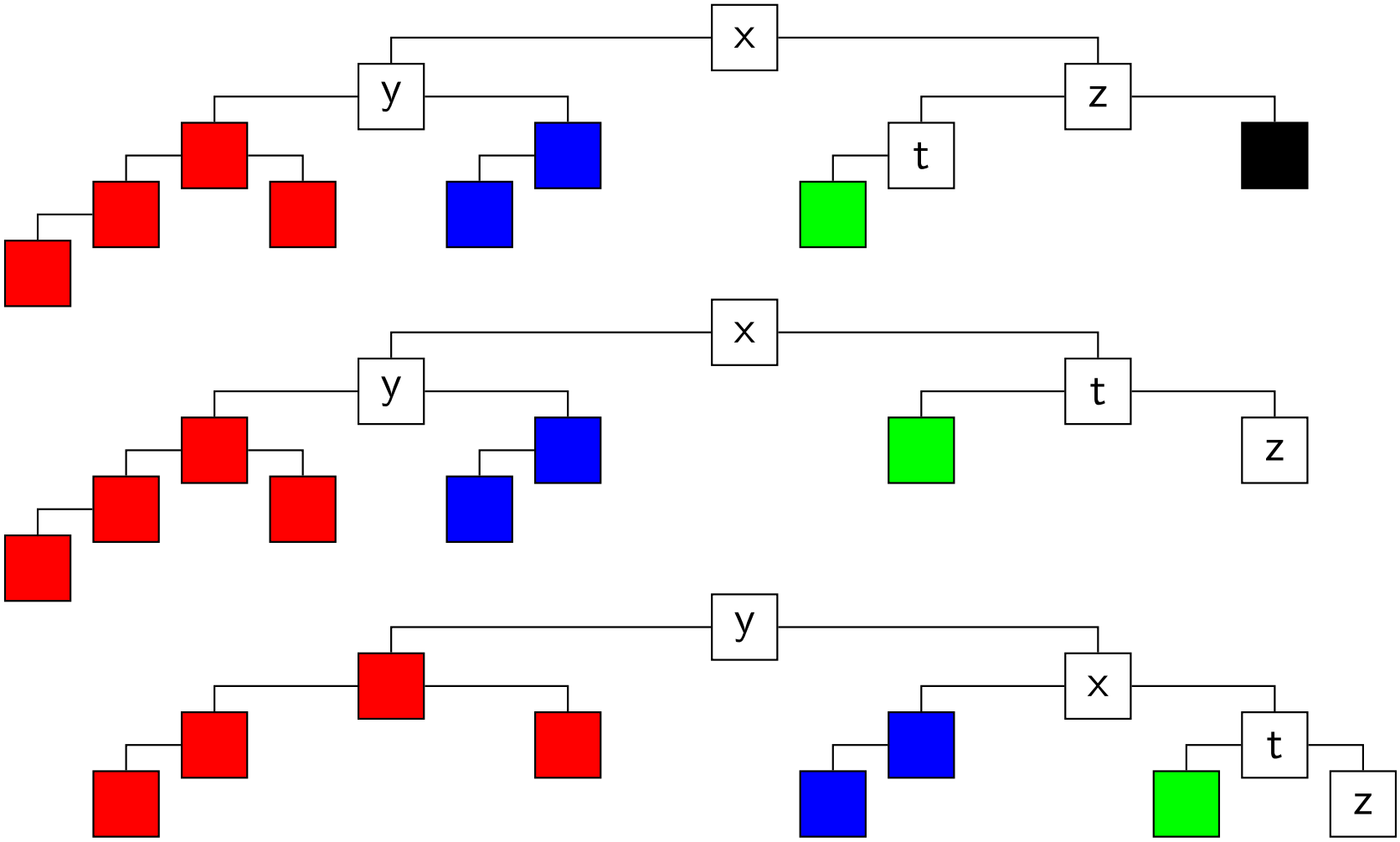


Hauteur excédentaire dans le sous-arbre droit du sous-arbre gauche.

La hauteur diminue.



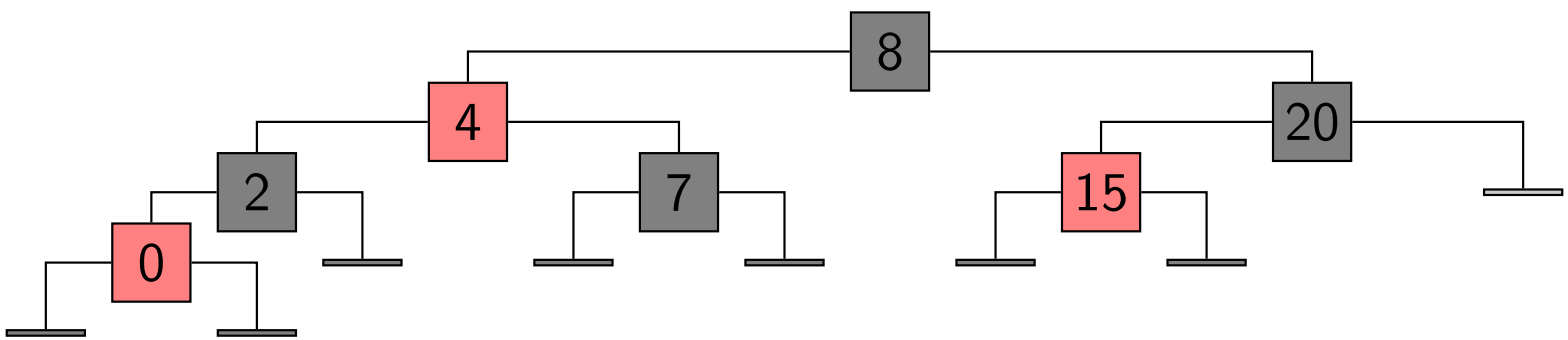
Suppression dans un AVL (3)



Arbre rouge-noir

Un arbre rouge-noir est un arbre *complet* (tout sommet possède 0 ou 2 fils) dont les sommets sont rouges ou noirs et qui vérifie les propriétés suivantes :

- Les feuilles ne contiennent pas de clés et sont noires ;
- La racine est noire ;
- Le père d'un sommet rouge est noir ;
- Les chemins d'un sommet s fixé à une feuille quelconque ont une même « hauteur noire ».
  (nombre de sommets noirs en excluant s)



Equilibre d'un arbre rouge-noir

La hauteur d'un arbre rouge-noir de n sommets
est majorée par $2\log_2(n)$.

Preuve.

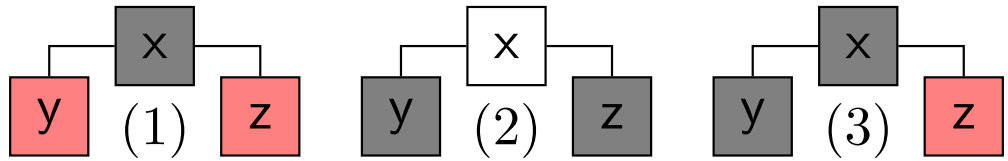
On note $h(x)$ (resp. $hb(x)$), la hauteur (resp. hauteur noire) du sous-arbre issu de x et $n(x)$ la taille du sous-arbre issu de x.

Par induction, on montre que :

- $h(x) \leq 2hb(x)$ et si x est interne alors $n(x) > 2^{hb(x)}$ sinon $n(x) = 2^{hb(x)} = 1$
- Si x est rouge, $h(x) < 2hb(x)$

D'où : $h(x) \leq 2hb(x) < 2\log_2(n(x))$.

Preuve d'équilibre d'un arbre rouge-noir



(1) $hb(x) = hb(y) = hb(z)$

$h(x) = 1 + \max(h(y), h(z)) \leq 2hb(y) = 2hb(x)$

$n(x) > 1 + 2 \cdot 2^{hb(x)} > 2^{hb(x)}$

(2) $hb(x) = hb(y) + 1 = hb(z) + 1$

$h(x) = 1 + \max(h(y), h(z)) \leq 2hb(y) + 1 < 2hb(x)$

$n(x) \geq 1 + 2 \cdot 2^{hb(x)-1} > 2^{hb(x)}$

(3) $hb(x) = hb(y) + 1 = hb(z)$

$h(x) = 1 + \max(h(y), h(z)) \leq 1 + \max(2hb(y), 2hb(z) - 1) = 2hb(x)$

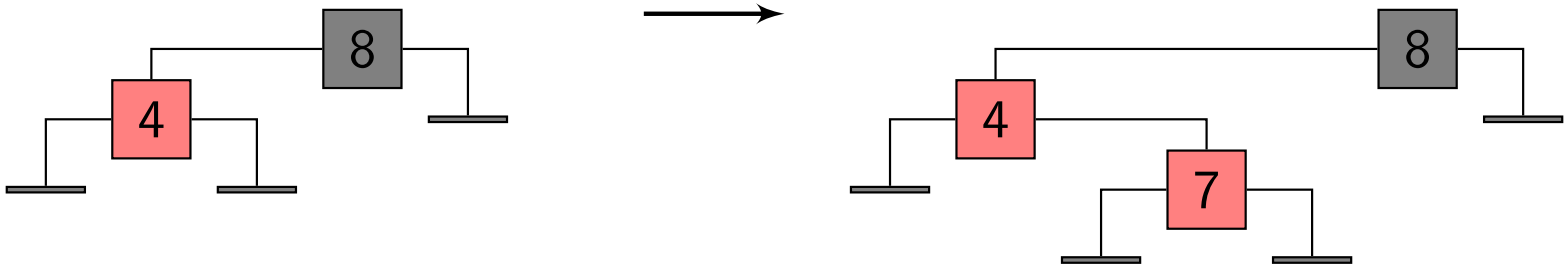
$n(x) > 1 + 2^{hb(x)-1} + 2^{hb(x)} > 2^{hb(x)}$

Insertion dans un arbre rouge-noir

Insertion du sommet à la place d'une feuille.

Ajout de deux fils.

Coloration en rouge.



Seules propriétés (éventuellement) non respectées :

- le père du sommet inséré est rouge ;
- le sommet est une racine rouge.

Rééquilibrage après insertion (1)

Une fois la clé insérée,

on traite le cas d'un père et fils rouge ou d'une racine rouge.

La racine rouge est noircie.

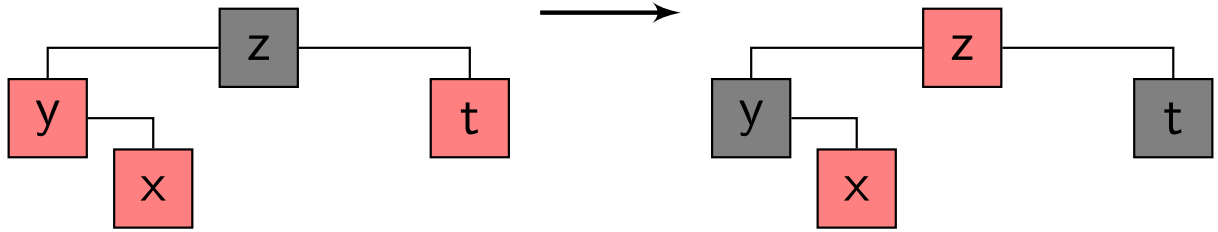
Après traitement d'un père et fils rouge.

- soit l'arbre est un arbre rouge-noir ;
- soit le grand-père viole l'une des propriétés précédentes et on procède itérativement.

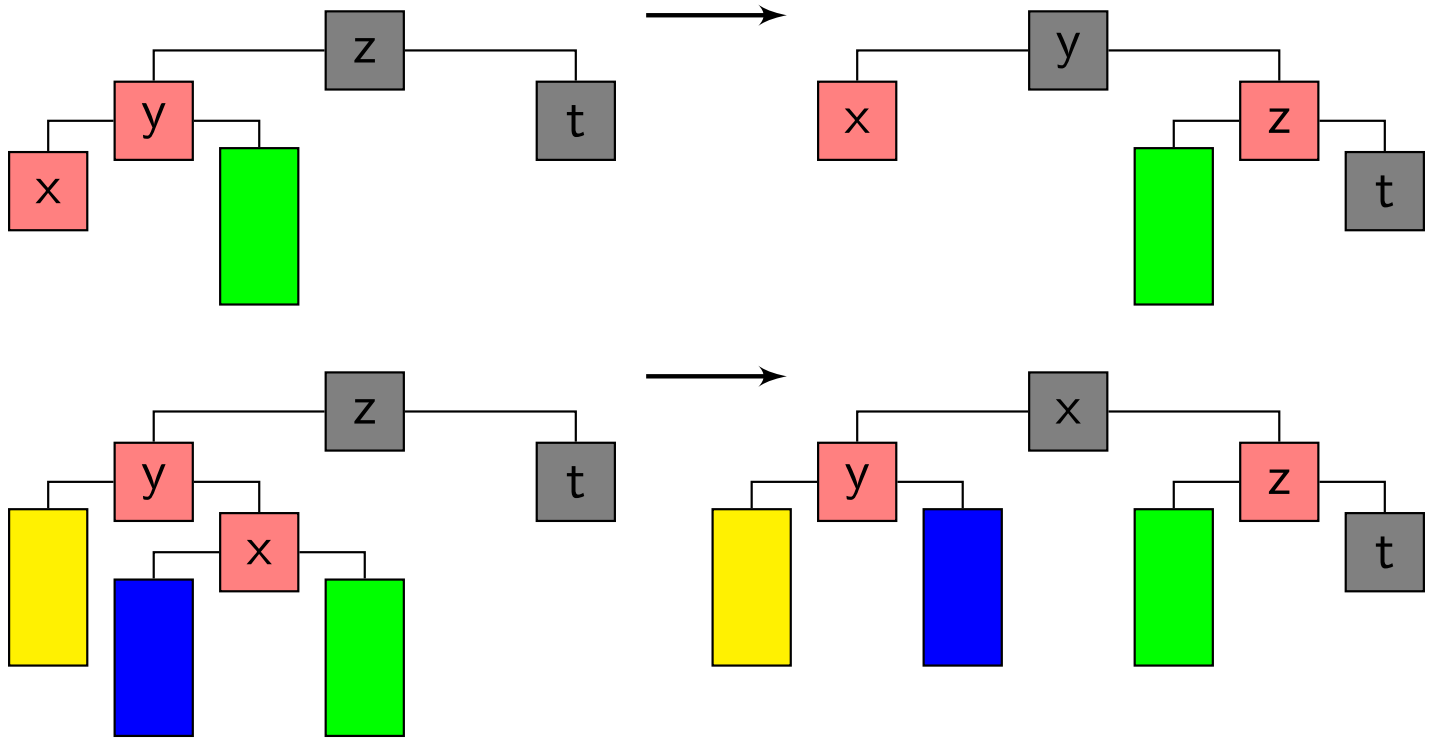
Rééquilibrage après insertion (2)

Examen de x.

- Il faudra ensuite examiner z.



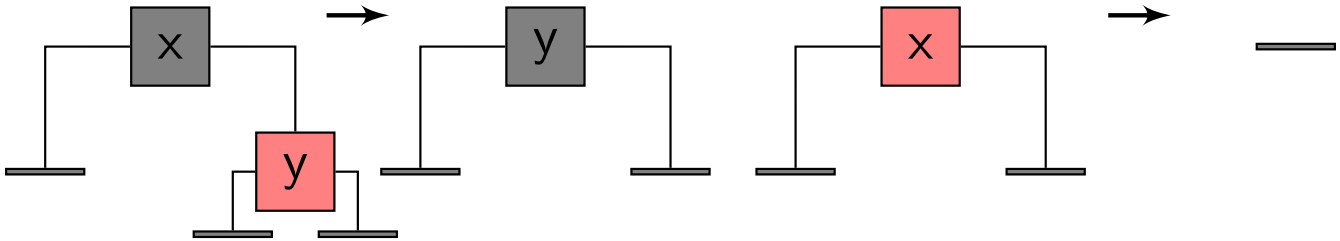
- L'arbre redevient un arbre rouge-noir.



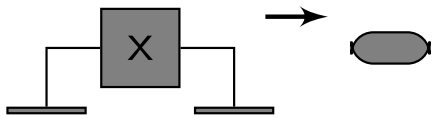
Suppression dans un arbre rouge-noir

Rappel : On supprime toujours un sommet avec au moins un fils manquant.

Cas favorables.

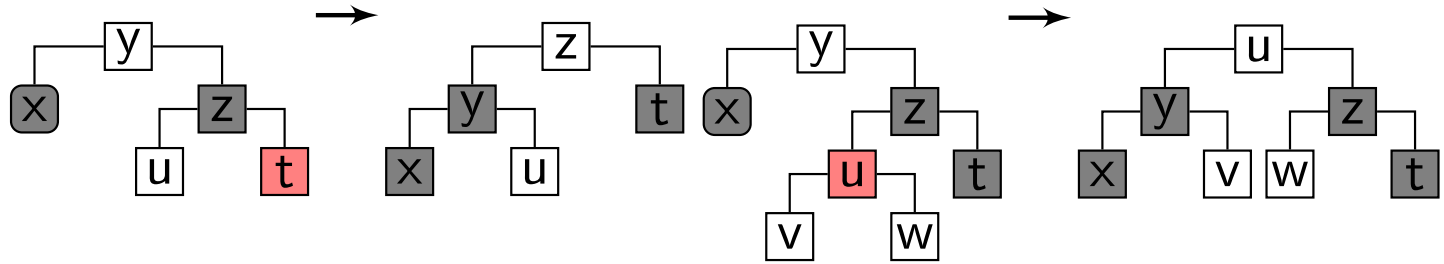
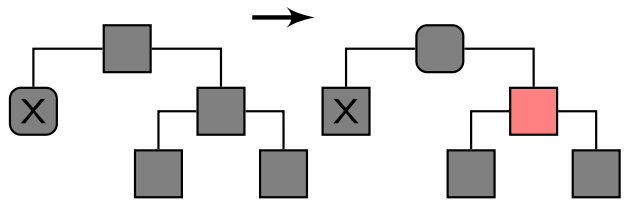
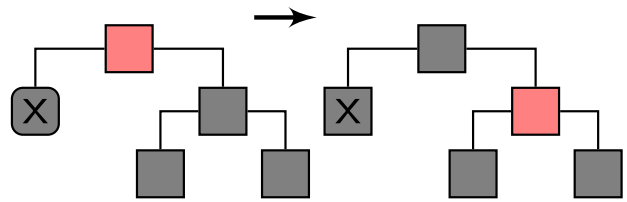
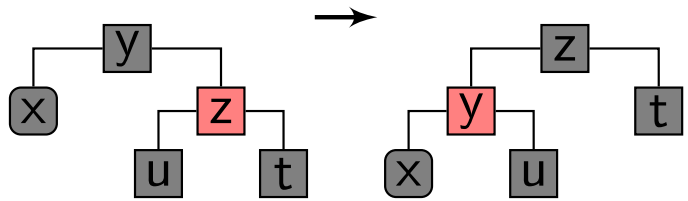


Cas défavorable.



Les sommets à bords arrondis sont *dégradés* : leur nombre de sommets noirs jusqu'à une feuille est inférieur d'une unité à la hauteur noire de leur père.

Elimination des sommets dégradés

	Père noir	Père rouge
Frère noir avec neveu rouge		
Frère noir sans neveu rouge		
Frère rouge		

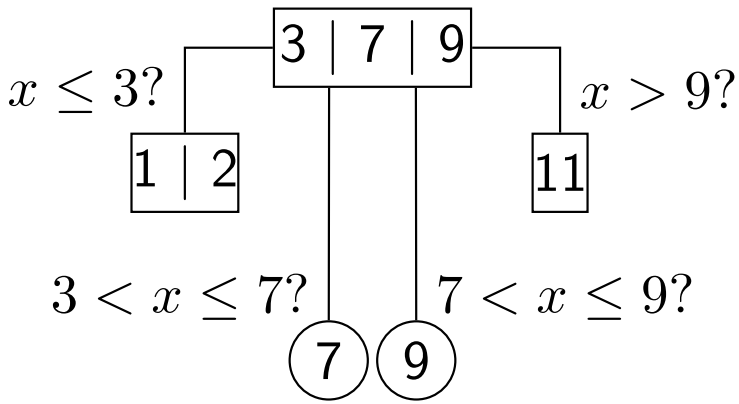
Le cas « père et frère noirs sans neveu rouge » transfère au père le déséquilibre.

Le cas « père noir et frère rouge » se transforme en un cas « père rouge ».

Arbre balisé

Un arbre balisé est un arbre dont :

- seules les feuilles contiennent les clés ;
- les sommets internes contiennent des informations de navigation appelées *balises*.



Un arbre a-b est un arbre balisé qui vérifie :

- $a \geq 2$ et $b \geq 2a - 1$;
- si la racine est un sommet interne elle a au moins deux fils ;
- les autres sommets internes ont au moins a fils ;
- tous les sommets internes ont au plus b fils.
- les feuilles ont même profondeur.

Propriétés des arbres a-b

Soit h la hauteur d'un arbre a-b.
La recherche d'une clé requiert au plus $(b-1)h$ comparaisons.

La hauteur h d'un arbre a-b de n clés est majorée par :

$$1 - \log_a(2) + \log_a(n)$$

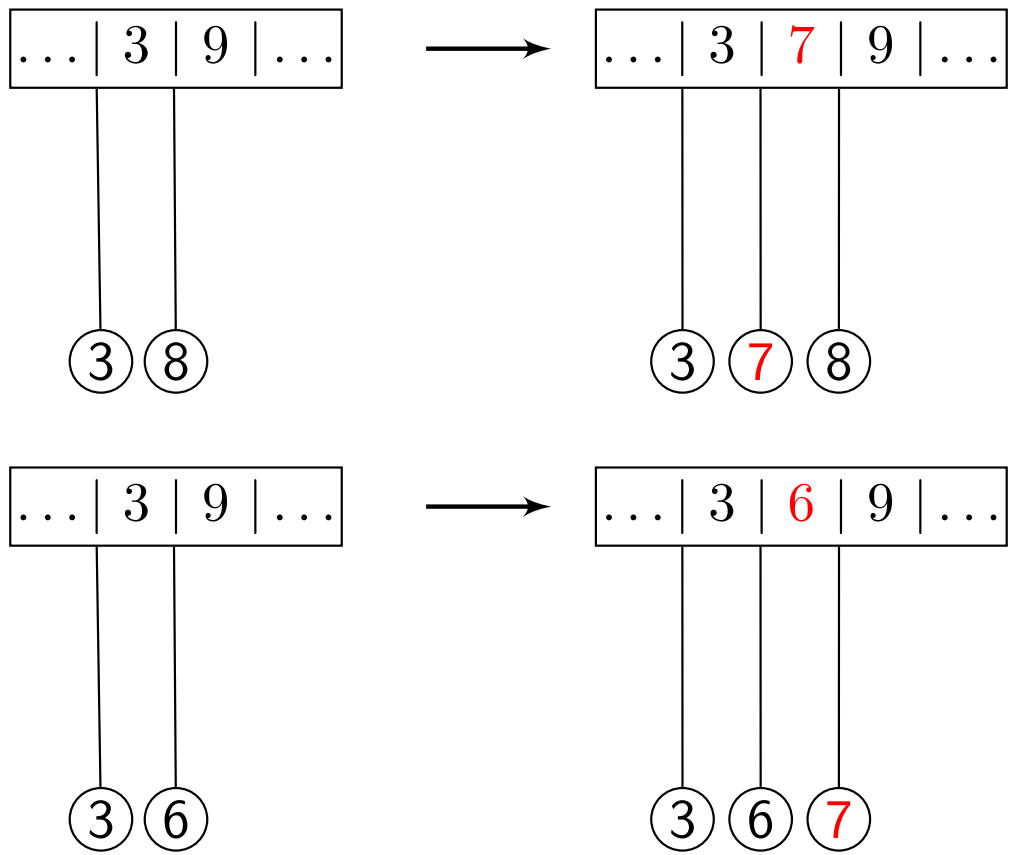
Preuve.

$$n \geq 2a^{h-1}$$

Insertion dans un arbre a-b

Une fois rencontré le dernier sommet interne,
on insère une nouvelle balise dans ce sommet.

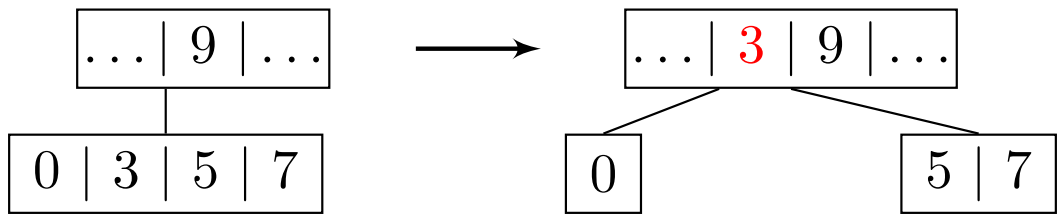
Illustration : insertion de 7.



Rééquilibrage après insertion

Si le sommet interne a $b+1$ fils, on le découpe en deux sommets avec $\lfloor \frac{b+1}{2} \rfloor$ et $\lceil \frac{b+1}{2} \rceil$ fils.

Illustration : $a = 2$, $b = 4$.



On itère ce procédé si nécessaire en remontant jusqu'à la racine.

Si la racine est découpée, on créé une nouvelle racine avec deux fils.

Suppression dans un arbre a-b

Une fois rencontré le dernier sommet interne,

on supprime la balise qui a sélectionné la clé.

Illustration : suppression de 8.

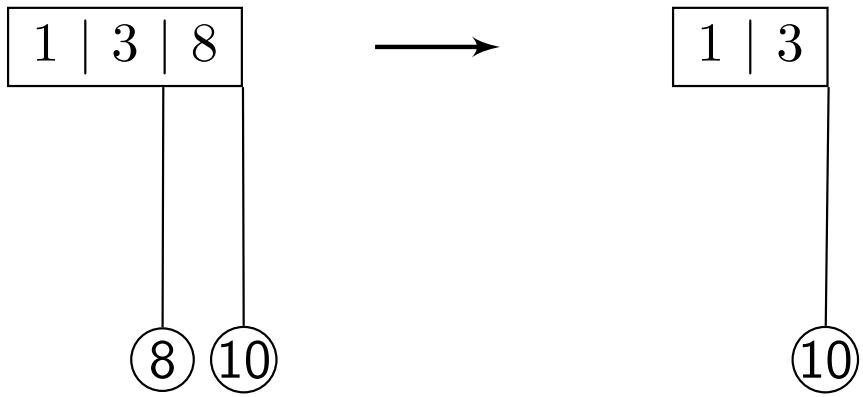
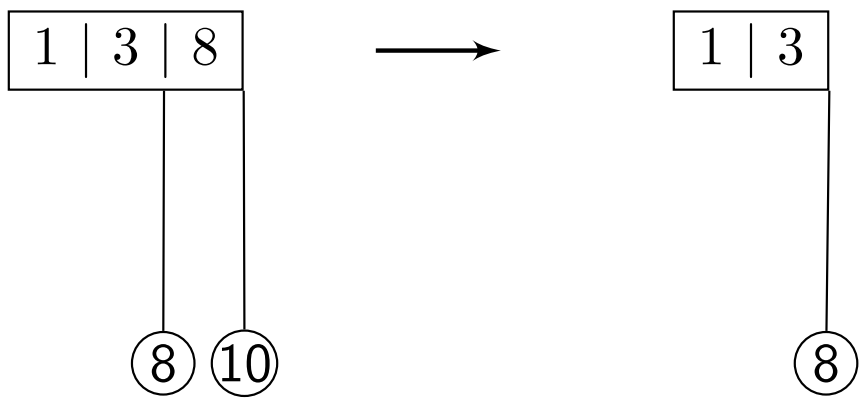


Illustration : suppression de 10.



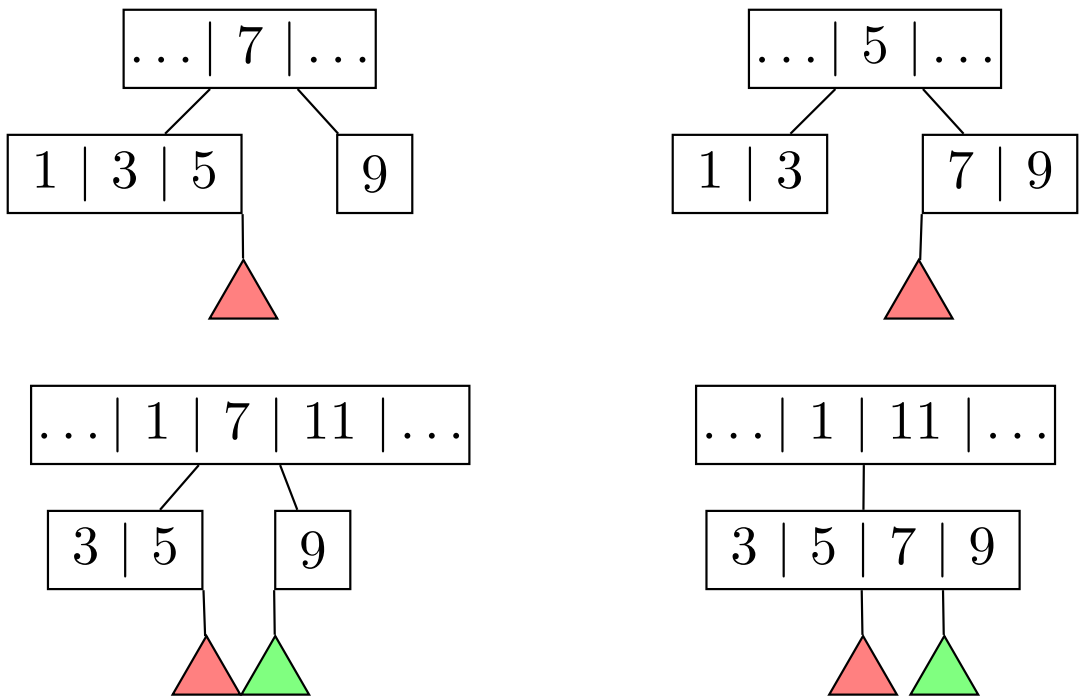
Rééquilibrage après suppression

Si la racine a 1 fils, on supprime la racine et son fils devient la racine.

Si un (autre) sommet interne a $a - 1$ fils :

- si l'un de ses frères adjacents a au moins $a + 1$ fils,
  on « transfère » l'un de ces fils au sommet ;
- si tous ses frères adjacents ont a fils,
  on « fusionne » l'un des frères avec le sommet
  et on itère (éventuellement) avec le père.

Illustration : $a = 3$.



Concaténation d'arbres a-b

Soit $\mathcal{A}_1$ et $\mathcal{A}_2$ de hauteur h_1 et h_2 avec $h_1 \geq h_2$.

On suppose que $c = \max(\mathcal{A}_1) < \min(\mathcal{A}_2)$.

Concaténation.

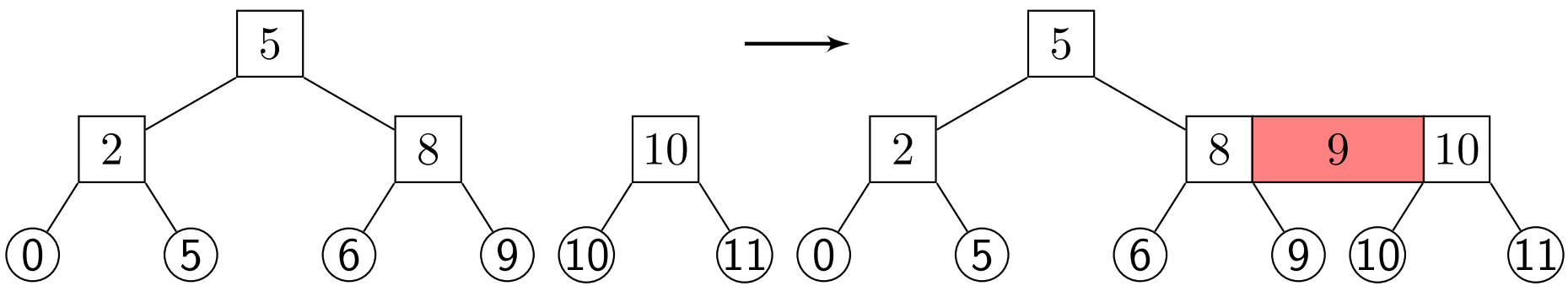
On calcule c en $O(h_1)$.

On descend dans $\mathcal{A}_1$ à droite jusqu'au sommet de hauteur $h_1 - h_2$ en $O(1 + h_1 - h_2)$.

On fusionne ce sommet à la racine de $\mathcal{A}_2$ à l'aide de c en $O(1)$.

On rééquilibre l'arbre en $O(1 + h_1 - h_2)$.

La hauteur du nouvel arbre est soit h_1 soit $h_1 + 1$.



Scission d'un arbre a-b

Soit $\mathcal{A}$ un arbre a-b et c une clé, on veut construire $\mathcal{A}_1$ et $\mathcal{A}_2$ tels que :

- $\mathcal{A}_1$ contient les clés de $\mathcal{A}$ inférieures ou égales à c ;
- $\mathcal{A}_2$ contient les clés de $\mathcal{A}$ supérieures à c.

Principe.

On parcourt le chemin vers la clé

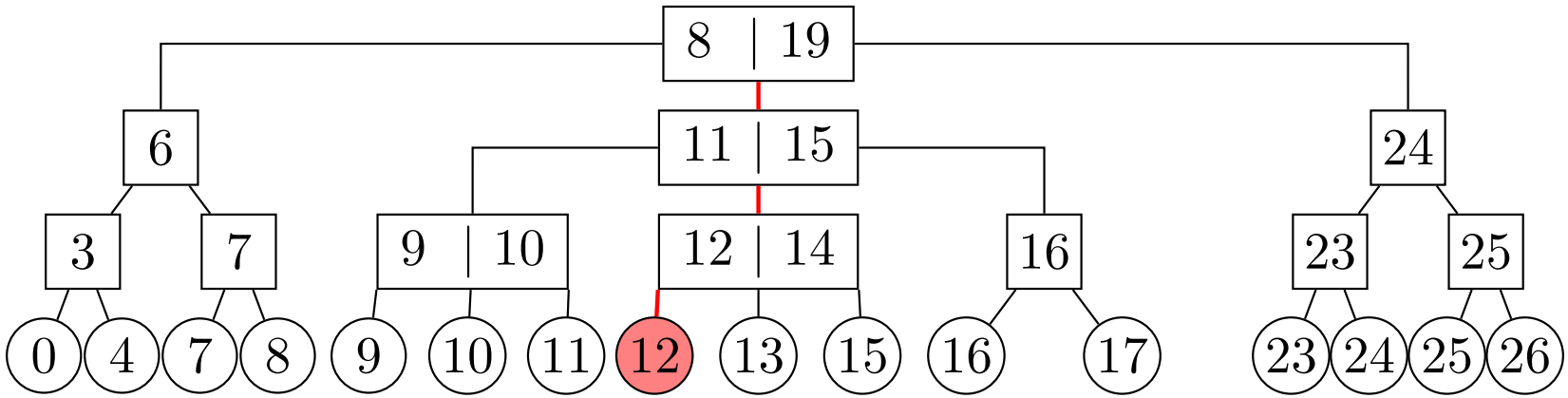
- en découpant chaque sommet interne (si nécessaire) selon la position par rapport au chemin ;
- en supprimant les arêtes visitées.

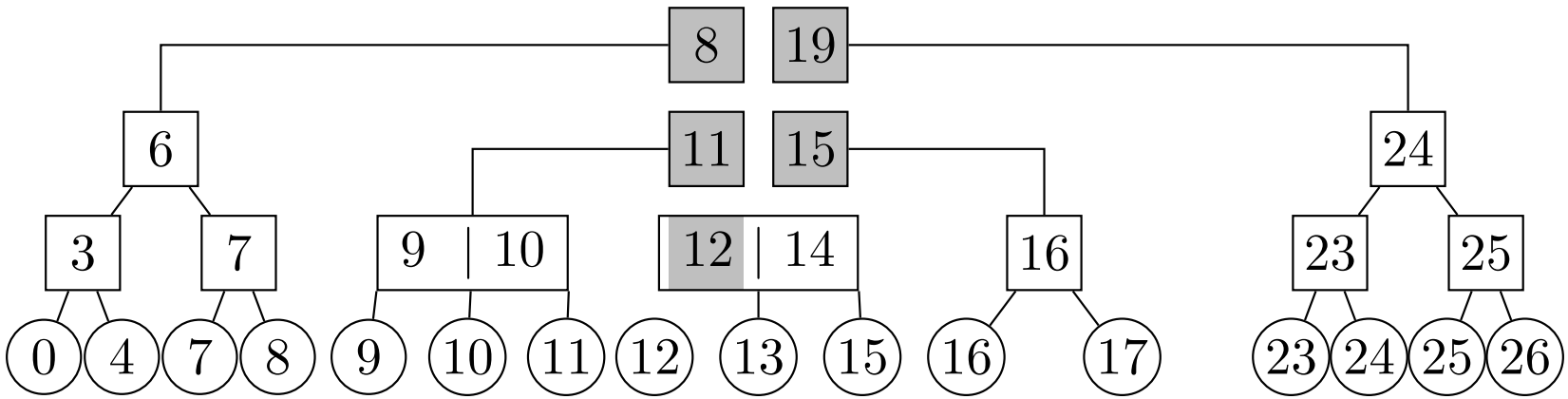
Les sous-arbres produits sont modifiés ainsi :

- la balise « extrême » de la racine majore les clés du sous-arbre ;
- on la mémorise et on la supprime ;
- si la racine ne contient plus de balise, on la supprime.

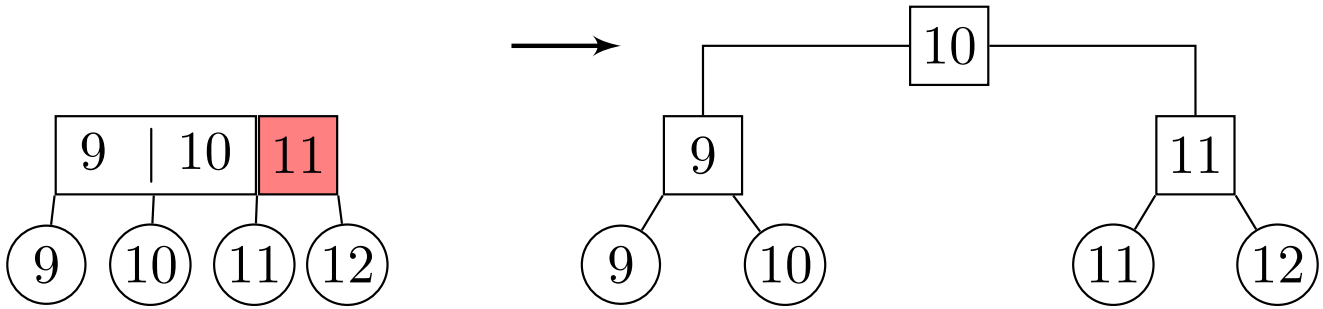
A l'aide des balises extrêmes, on concatène les sous-arbres à gauche et à droite par hauteur croissante.

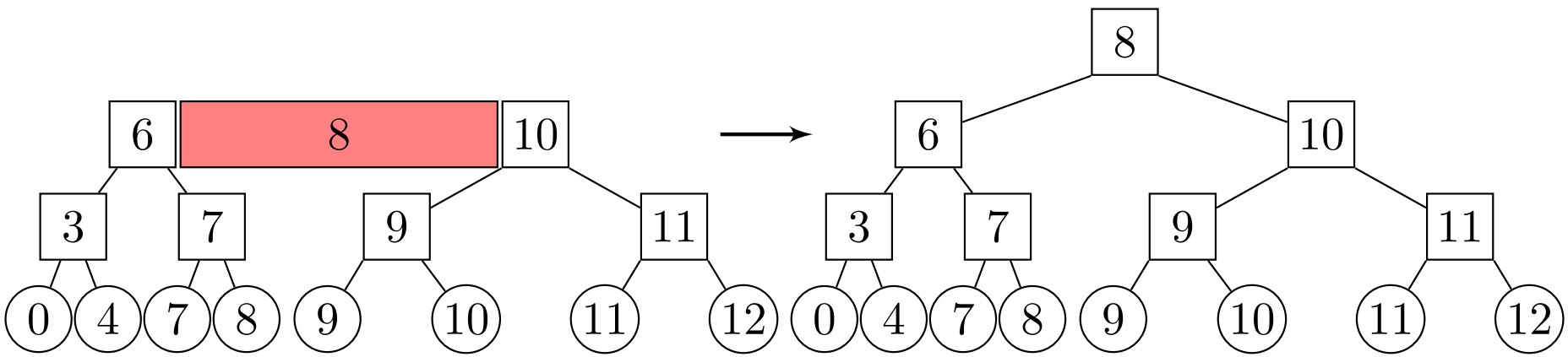
Scission d'un arbre a-b : découpage





Scission d'un arbre a-b : concaténations





Complexité de la scission

Soit h la hauteur de l'arbre. La visite de la clé s'effectue en $O(h)$.

On fabrique $k_g \leq h + 1$ sous-arbres « à gauche ».

Les hauteurs des sous-arbres à gauche vérifient $h_1 \leq h_2 \leq \ldots \leq h_{k_g}$.

Les hauteurs $\{h'_i\}_{i \geq 2}$ après concaténation vérifient $h_i \leq h'_i \leq h_i + 1$.

Puisque les clés de concaténation sont connues,
la suite de concaténations s'effectue en :

$$\sum_{i=2}^{k_g} O(1 + h_i - h_{i-1}) = O(k_g) + O(h) = O(h)$$

A l'aide du même raisonnement pour les sous-arbres « à droite » que
pour les sous-arbres « à gauche », on conclut que la scission s'effectue en $O(h)$.

Quelles valeurs pour a et b ?

La complexité de la recherche est en $O(b \log_a(n))$ avec $b \geq 2a - 1$.

D'où un choix de b (a priori) optimal $b = 2a - 1$
puis un choix de $a = 2$ car $(2a - 1) \log_a(n)$ croît avec a.
Ce qui implique $b = 3$.

Cependant on démontre que :

- le coût moyen des rééquilibrages d'une suite d'insertions et de suppressions est en $O(1)$ dans les arbres 2-4 ;
- ce coût moyen est en $\Omega(\log(n))$ dans les arbres 2-3.

Pourquoi les arbres rouge-noir ?

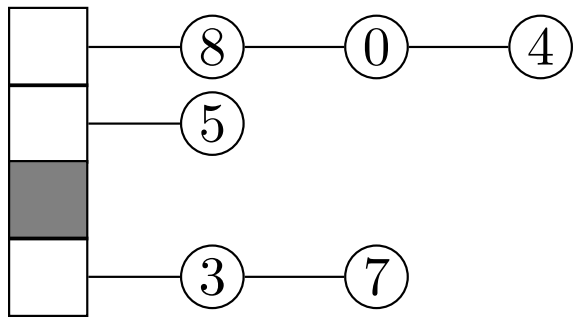
Ils émulent les arbres 2-4 !

Table de hachage

Une table T de taille m de listes et h une fonction des clés dans $\{0, \ldots, m-1\}$.

Insertion de c : parcours de la liste $T[h(c)]$ et insertion de c si absente.

Illustration : $m = 4$ et $h(c) = c \mod 4$.



Complexité de la recherche.

- au pire, les n clés sont rangées dans la même liste d'où $O(n)$.
- en moyenne,
  - *hypothèses* : $|h^{-1}(i)|$ indépendant de i et clés tirées uniformément ;
  - *conclusion* : la longueur moyenne des listes est $\frac{n}{m}$;
  - d'où une complexité moyenne en $O(1)$ si $m = \Omega(n)$.

Dictionnaire statique

Dans un dictionnaire *statique* les mises à jours sont rares. Par exemple :

- des catalogues de produits ;
- des dictionnaires de langue ;
- l'ensemble des mots réservés d'un langage de programmation.

Une *collision* de h est un ensemble $\{x, x'\}$ tel que $h(x) = h(x')$.
Intérêt de rechercher une fonction de hachage qui ne provoque pas de collisions.

Hypothèses.

- Les indices de la table sont $\{0, \ldots, m-1\}$;
- Les n clés appartiennent à $\{0, \ldots, p-1\}$ avec p premier ;
- L'espace des fonctions de hachage est :

  $$H = \{h_{a,b}(x) = (ax + b \mod p) \mod m \mid 0 < a < p \wedge 0 \leq b < p\}$$

  d'où $|H| = p(p-1)$.

Un choix approprié de m

On note $nbcoll$ la somme des collisions pour tous les $h \in H$.

$$nbcoll = \frac{1}{2} \sum_{a,b} |\{(x, x') \mid x \neq x' \wedge h_{a,b}(x) = h_{a,b}(x')\}|$$

$nbcoll$ vérifie la majoration suivante :

$$nbcoll \leq \frac{n(n-1)}{2} \cdot \frac{p(p-1)}{m}$$

Par conséquent, si $m = n^2$,

plus de la moitié des fonctions ne provoquent pas de collisions.

Avec en moyenne deux tirages aléatoires,

on trouve une fonction de hachage sans collision.

Preuve de la majoration

On note $+_p, \times_p$ les opérations modulo p
et $\mathbb{Z}/p\mathbb{Z}$ l'ensemble $\{0, \ldots, p-1\}$ doté de ces opérations (un corps).
Soit $\tilde{h}_{a,b}(x) = a \times_p x +_p b$.

1. $\tilde{h}_{a,b}$ est bijective.
2. Pour tout $y \neq y'$ et $x \neq x'$, il existe une et une seule paire (a, b) telle que $y = \tilde{h}_{a,b}(x)$ et $y' = \tilde{h}_{a,b}(x')$.

$$nbcoll = \frac{1}{2} \sum_{a,b} |\{(x, x') \mid x \neq x' \wedge h_{a,b}(x) = h_{a,b}(x')\}|$$

Soit $x \neq x'$. $h_{a,b}(x) = h_{a,b}(x') \Leftrightarrow \tilde{h}_{a,b}(x) = \tilde{h}_{a,b}(x') \mod m$.
D'après la propriété 1, $\tilde{h}_{a,b}(x) \neq \tilde{h}_{a,b}(x')$. D'où :

$$nbcoll = \frac{1}{2} \sum_{a,b} \sum_{y \neq y'} |\{(x, x') \mid x \neq x' \wedge y = y' \mod m \wedge y = \tilde{h}_{a,b}(x) \wedge y' = \tilde{h}_{a,b}(x')\}|$$

D'après la propriété 2, $nbcoll = \frac{n(n-1)}{2} \sum_{y \neq y'} |\{(y, y') \mid y = y' \mod m\}|$.
Il y a au plus $p(\lceil \frac{p}{m} \rceil - 1) \leq p(\frac{p+m-1}{m} - 1) = \frac{p(p-1)}{m}$ paires (y, y')
avec $y \neq y'$ telles que $y = y' \mod m$.

Une autre majoration

Soit $H' \subseteq H$ l'ensemble des fonctions de hachage

qui provoquent au moins n collisions.

$$\boxed{\text{Si } m = n \text{ alors } |H'| < \frac{|H|}{2}}$$

Preuve.

$$n|H'| \leq nbcoll \leq \frac{n(n-1)}{2} \cdot \frac{p(p-1)}{n}$$

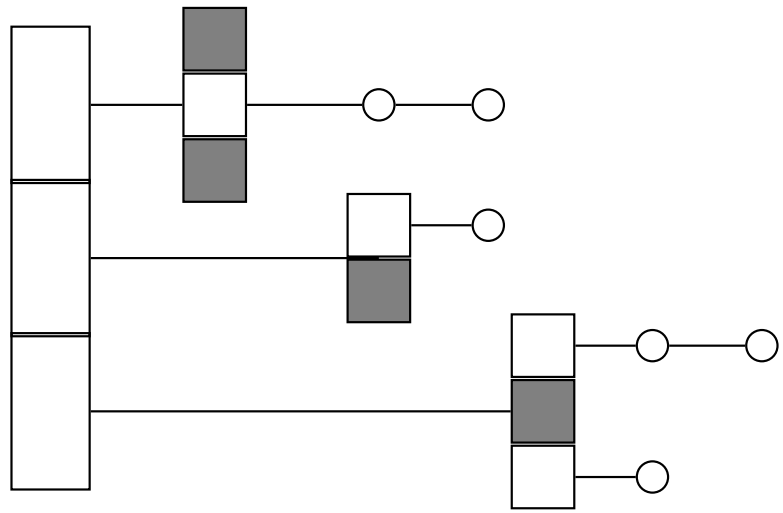
D'où :

$$|H'| \leq \frac{n(n-1)}{2n^2} \cdot p(p-1) < \frac{|H|}{2}$$

Double hachage

Dans une table de double hachage T,

- chaque entrée i pointe sur une table de hachage T_i de taille m_i ;
- dont les entrées sont calculées par la fonction h_i.



On choisit $m = n$ et une fonction $h \in H \setminus H'$.

Soit n_i le nombre de clés x telles que $h(x) = n_i$, on choisit :

- $m_i = \max(1, n_i^2)$;
- h_i une fonction qui ne provoque pas de collisions entre les n_i clés.

Complexité du double hachage

Complexité spatiale.

La table primaire occupe n cellules et les tables secondaires occupent :

$$\sum_{i=1}^{n} \max(n_i^2, 1) \leq \sum_{i=1}^{n} (n_i^2 + 1) = n + \sum_{i=1}^{n} n_i + \sum_{i=1}^{n} n_i(n_i - 1)$$

$$= 2n + \sum_{i=1}^{n} n_i(n_i - 1) < 2n + 2n = 4n$$

Complexité temporelle.

La génération aléatoire d'une fonction de hachage s'effectue en $O(1)$.

Chaque fonction est trouvée après $O(1)$ essais en moyenne.

La vérification du nombre de collisions des fonctions de hachage s'effectue en $O(n)$.

Plan

File de priorité

Une file de priorité est un multi-ensemble de valeurs dans un espace ordonné.

Les opérations usuelles sont :

- FileVide() qui teste si la file est vide ;
- Insérer(valeur) qui insère une nouvelle valeur ;
- Minimum() qui renvoie la valeur minimale de la file (non vide) ;
- ExtraireMin() qui renvoie la valeur minimale de la file (non vide) et l'extrait de la file ;

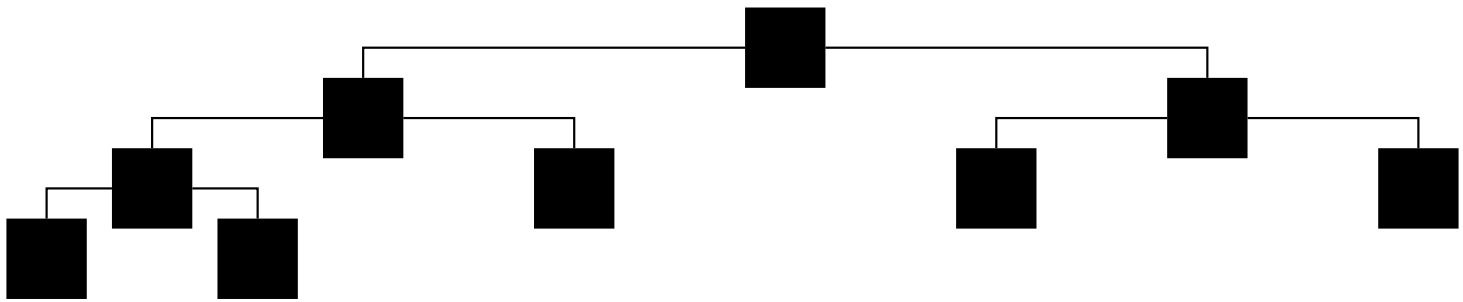
Si les valeurs disposent d'un accès secondaire :

- DiminuerCle(Id,val) remplace la valeur référencée par Id si val est inférieure à la valeur courante ;
- Supprimer(Id) remplace la valeur référencée par Id ;

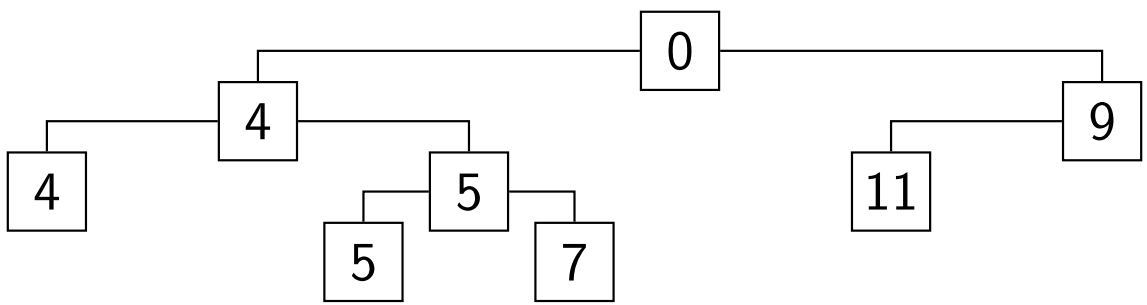
Tas binaire

Un arbre binaire de n sommets est *parfait* si :

- La hauteur de l'arbre h est égale à $\lfloor \log_2(n) \rfloor$;
- Il y a $2^{h'}$ sommets pour tout $h' < h$;
- Les $n - 2^h + 1$ feuilles de hauteur h sont regroupées à gauche.



Un arbre de valeurs est un *arbre tournoi* si la valeur d'un sommet est inférieure ou égale à celles de ses fils.

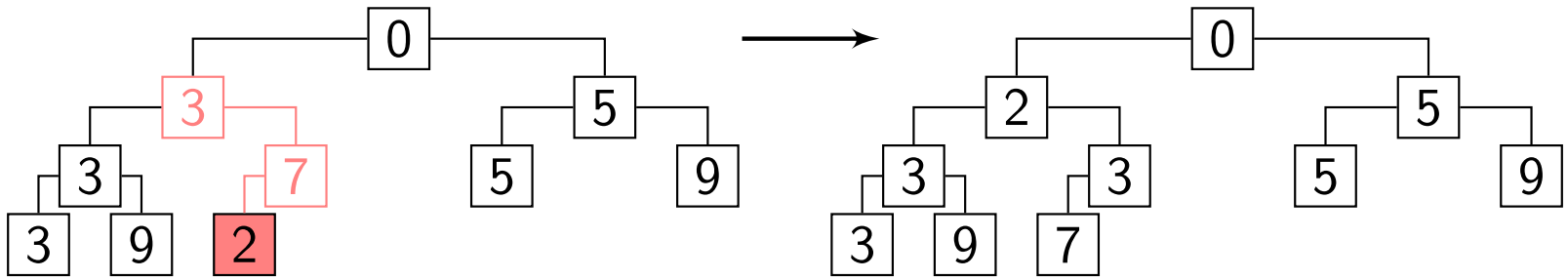


Un *tas binaire* est un arbre tournoi parfait.

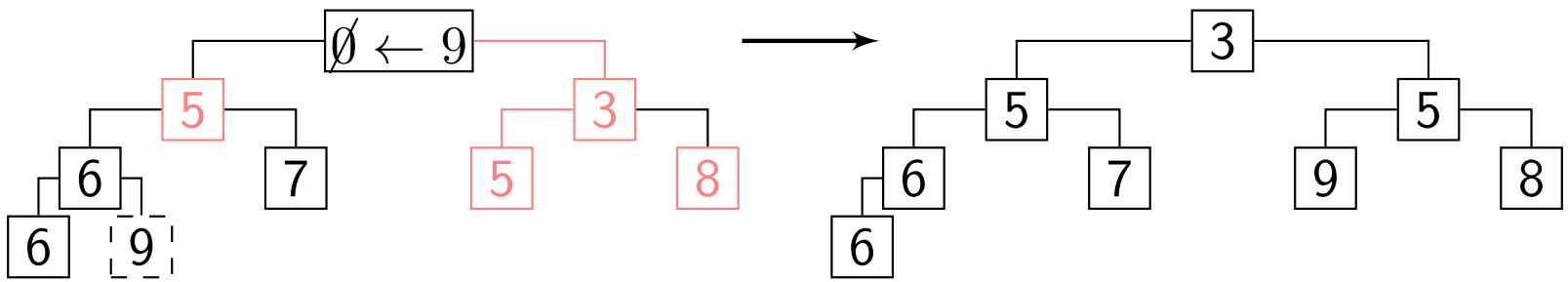
Opérations d'un tas binaire

Le minimum est renvoyé en $O(1)$.

L'insertion s'effectue en $O(\log(n))$.



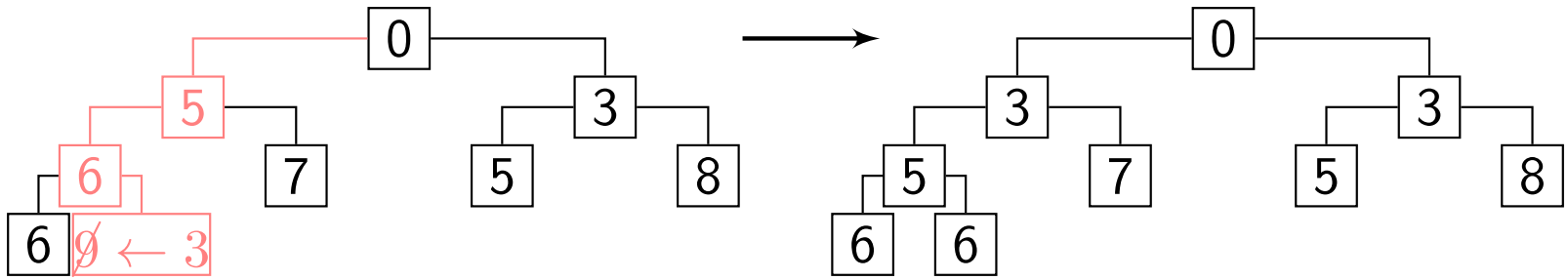
L'extraction du minimum s'effectue en $O(\log(n))$.



Opérations auxiliaires

Diminuer valeur

- Change la valeur si la nouvelle valeur est inférieure ;
- Remonte la valeur vers la racine ;



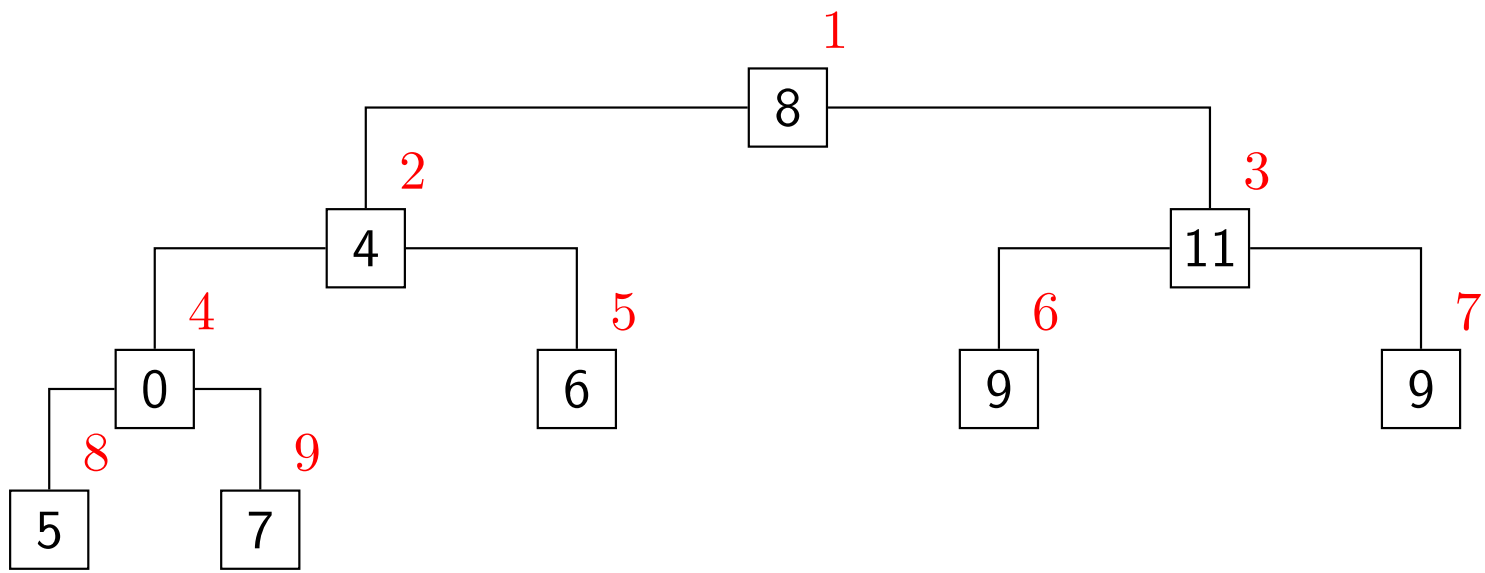
Supprimer valeur

- Affecte la valeur $-\infty$;
- Extrait la valeur minimale ($-\infty$) ;

Tas binaire et tableau

Soit T un tableau de n valeurs à organiser en tas binaire.

- La racine est $T[1]$;
- Le fils gauche de $T[i]$ est $T[2i]$ (pour $2i \leq n$) ;
- Le fils droit est $T[2i+1]$ (pour $2i < n$).



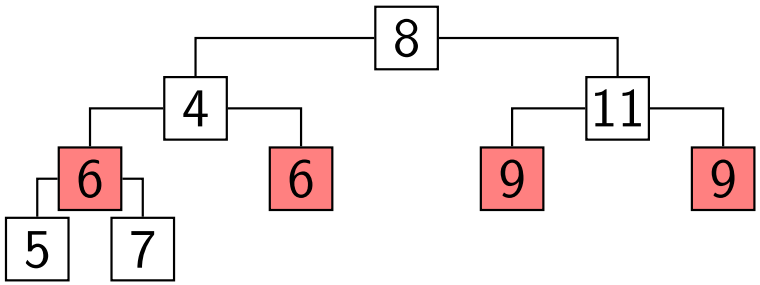
Construction du tas binaire

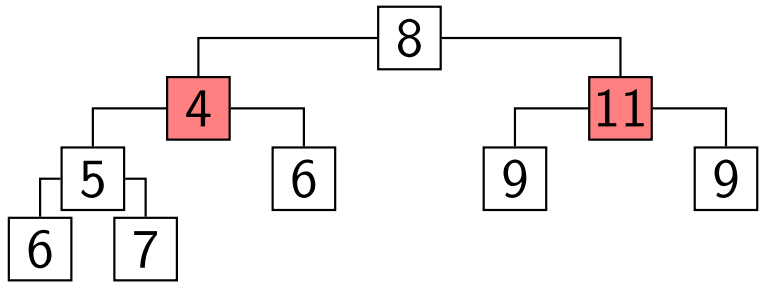
On construit le tas binaire par hauteur décroissante et de gauche à droite :

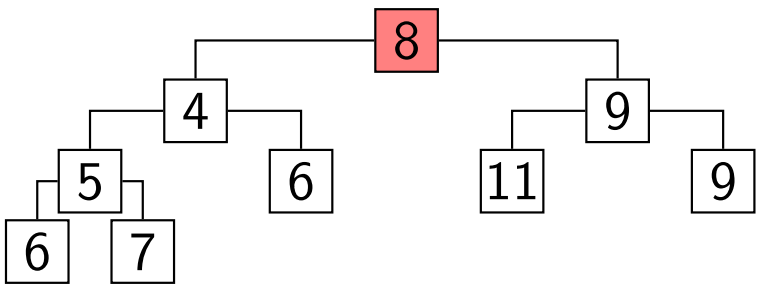
- La valeur du sommet courant est échangée avec la plus petite valeur de ses fils si cette valeur est inférieure.
- On itère ce procédé jusqu'aux feuilles si nécessaire.

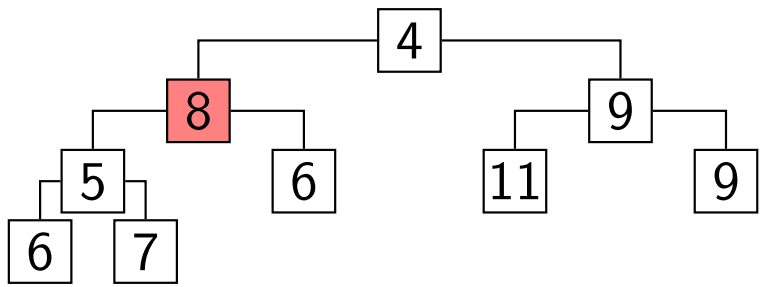
$i \leftarrow 2^{\lfloor \log_2(n) \rfloor - 1}$;
While $i \geq 1$ **do**
  For ℓ **from** i **to** $2i - 1$ **do**
    $j \leftarrow \ell$;
    Repeat
      $done \leftarrow$ **true** ; $k \leftarrow 2j$;
      If $2j < n$ **and** $T[2j] > T[2j + 1]$ **then** $k \leftarrow 2j + 1$;
      If $k \leq n$ **and** $T[k] > T[j]$ **then swap**(T, j, k) ; $j \leftarrow k$; $done \leftarrow$ **false** ;
    Until $done$;
  $i \leftarrow i \div 2$;

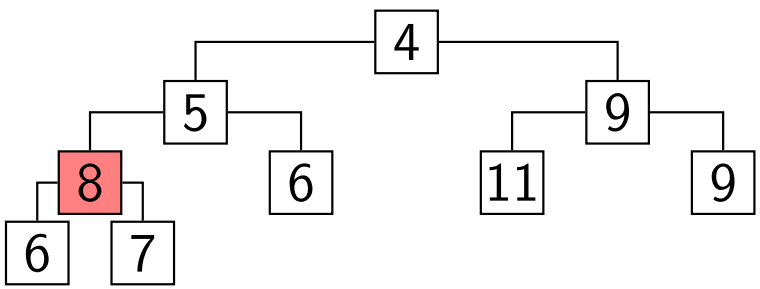
Illustration

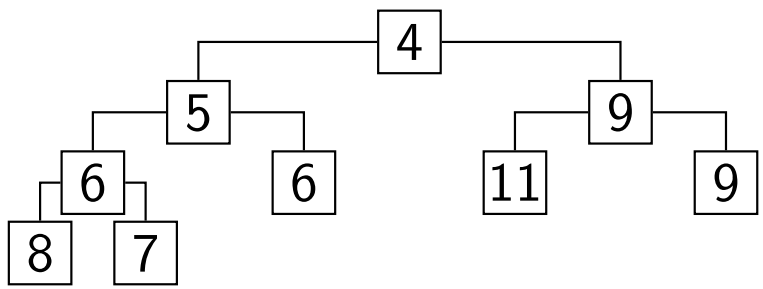












Analyse de complexité

La construction du tas binaire s'effectue en temps linéaire

Preuve.

Chaque sommet de hauteur h' donne lieu à un temps de calcul borné par $c(h-h')$.
D'où un temps de calcul borné par :

$$
\begin{aligned}
c \sum_{h' \leq h-1} 2^{h'}(h-h') = & \quad c \sum_{h' \leq h-1} \sum_{h'' \leq h'} 2^{h''} \\
\leq & \quad c \sum_{h' \leq h-1} 2^{h'+1} \\
\leq & \quad c 2^{h+1} \\
\leq & \quad 2cn
\end{aligned}
$$

Tri par tas

Le tri par tas du tableau T procède ainsi :

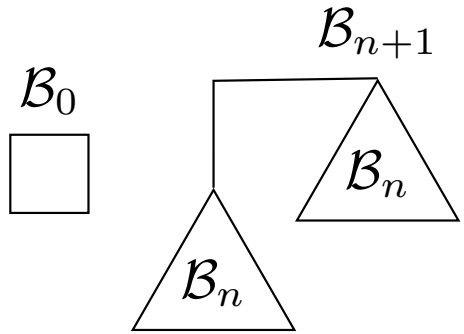
- On construit le tas (max) binaire de T en $O(n)$;
- On extrait consécutivement les $n-1$ maximum en les plaçant en $T[n], T[n-1], \ldots$ à la place libérée par les cellules ;
- Cette opération s'effectue en $O(n \log(n))$.

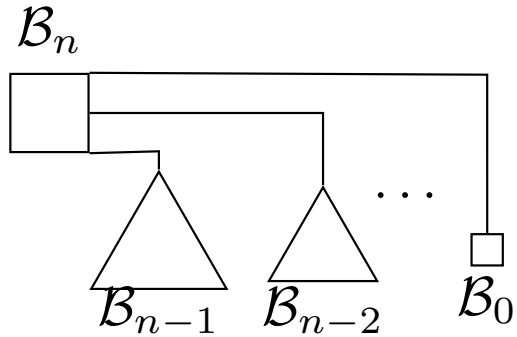
Le tri par tas est donc optimal en temps et en espace.

Arbre binomial

Objectif. Implémenter efficacement l'union de tas.

La famille des arbres binomiaux $\{\mathcal{B}_n\}_{n\in\mathbb{N}}$ est définie inductivement par :





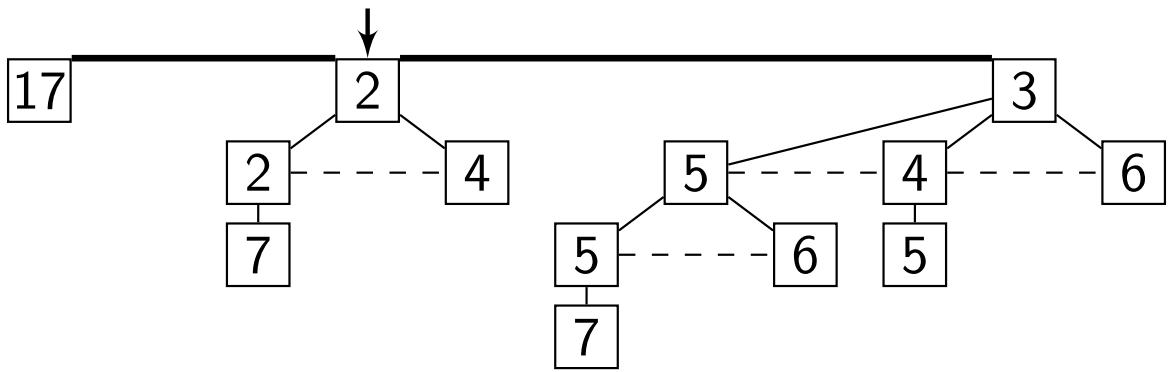
Propriétés.

- $\mathcal{B}_n$ a 2^n sommets et une hauteur n ;
- Il y a $\binom{n}{i}$ sommets de hauteur $i \leq n$;
- La racine a un degré n.

Tas binomial

Un tas binomial est une liste d'arbres binomiaux où :

- les arbres binomiaux sont des arbres tournois ;
- la liste est ordonnée par taille croissante
  avec un pointeur sur la plus petite racine ;
- Deux arbres binomiaux de la liste n'ont pas la même taille.

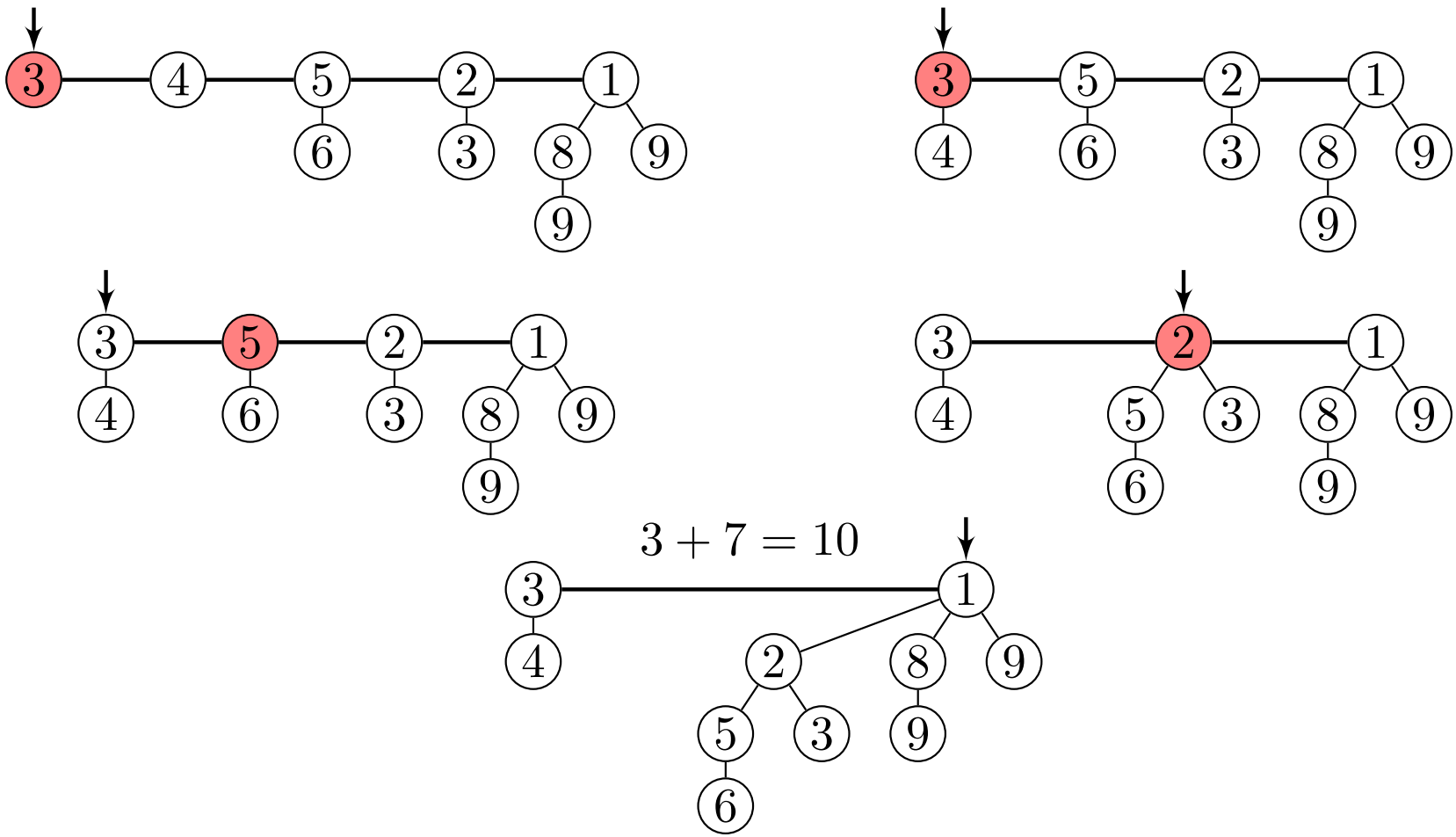


Soit $\mathcal{B}_k$ le plus grand arbre binomial de la liste alors $2^k \leq n < 2^{k+1}$.

La recherche du minimum s'obtient par le pointeur en $O(1)$.

Union de tas binomiaux

- **Fusion des listes d'arbres.**
  Il y a au plus deux arbres par taille.
- **Elimination des doublons.**
  Cette opération s'apparente à l'addition de deux nombres binaires.
  Au plus un triplet d'arbres de même taille (« arbre de retenue »).

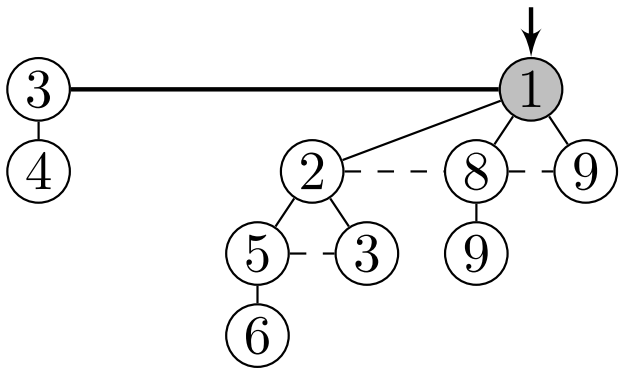


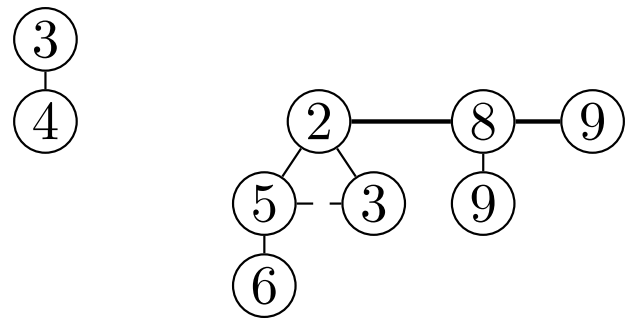
Insertion et extraction du minimum

Insertion d'une clé. Cas particulier de l'union !

Extraction du minimum.

- Détachement de l'arbre binomial et création d'un nouveau tas ...





- Puis union de tas !

Opérations auxiliaires. Procédé similaire à celui du tas binaire.

Tas de Fibonacci

Un tas de Fibonacci est une liste circulaire d'arbres tournois avec :

- un pointeur sur la plus petite racine ;
- un *marquage* dynamique de certains sommets.

La recherche du minimum s'obtient par le pointeur en $O(1)$.

L'union de tas (et donc l'insertion) s'effectue par fusion de listes et sélection du nouveau minimum.

Extraction du minimum

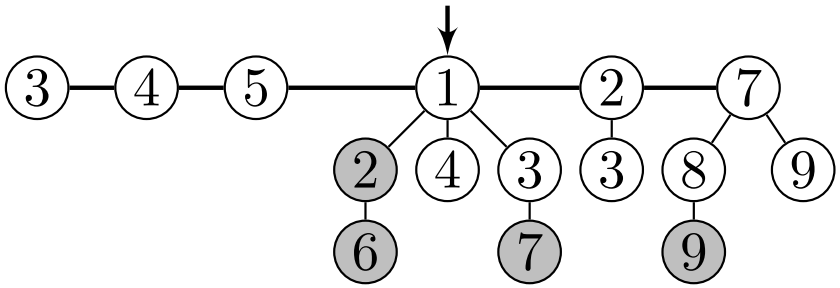
On procède en deux étapes.

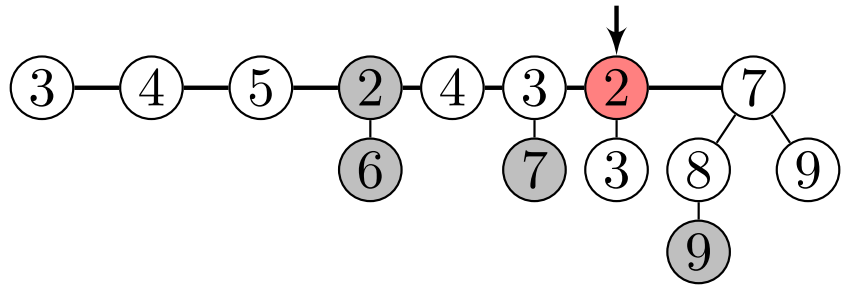
• La racine minimale est supprimée et ses fils sont insérés dans la liste des racines.

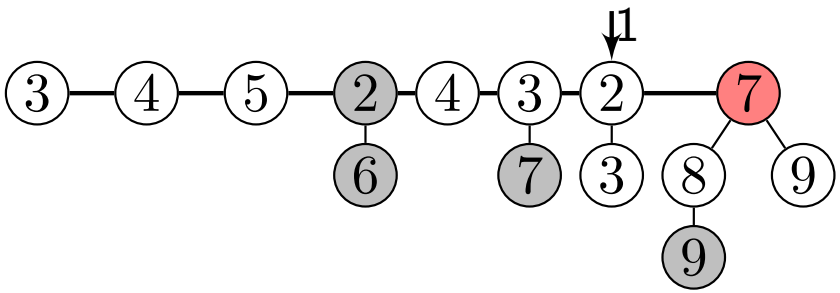
• On « compacte » la liste de façon à ne conserver qu'au plus une racine par degré (et simultanément on met à jour le pointeur min).

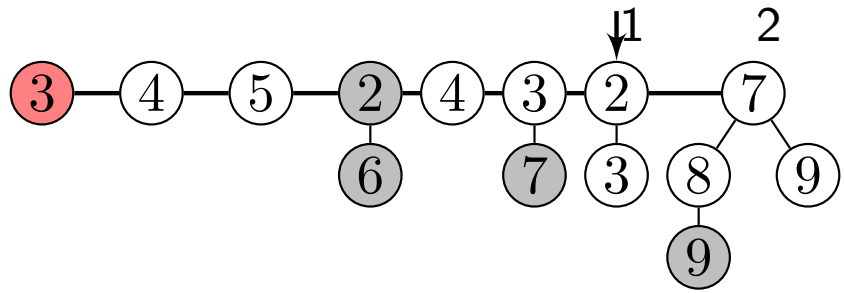
- On maintient un tableau de pointeurs indicé par degré.
- On parcourt la liste des racines.
  1. S'il n'existe pas de racine de même degré, alors on met à jour le tableau.
  2. S'il existe une racine de même degré alors
     on fusionne les deux arbres,
     on supprime le pointeur du tableau indicé par l'ancien degré,
     on ré-examine le tableau pour le nouveau degré.

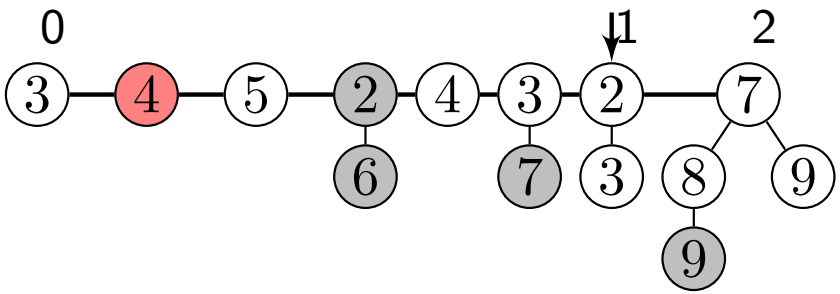
Extraction du minimum : exemple (1)

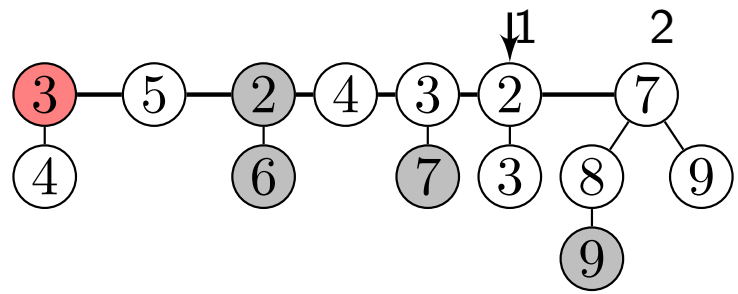




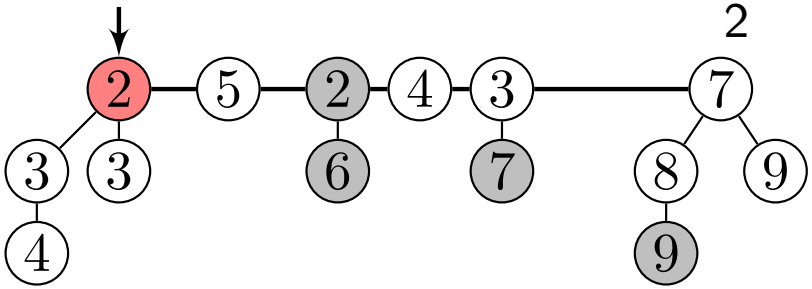


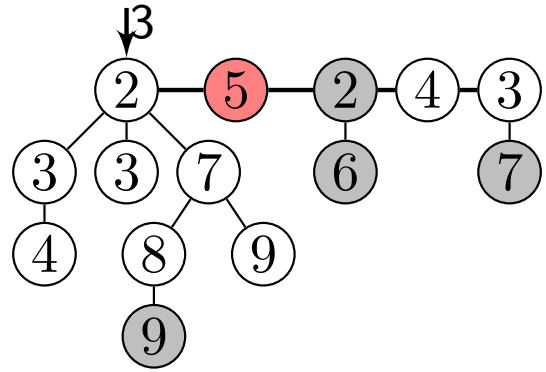


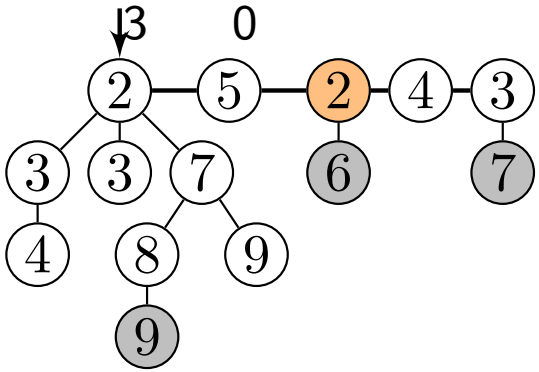


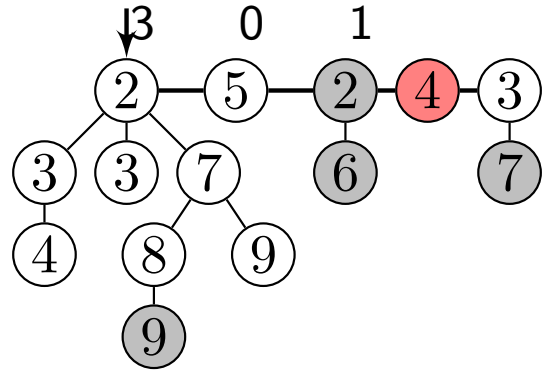


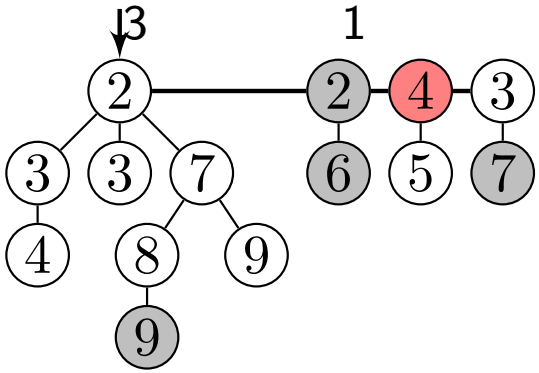
Extraction du minimum : exemple (2)

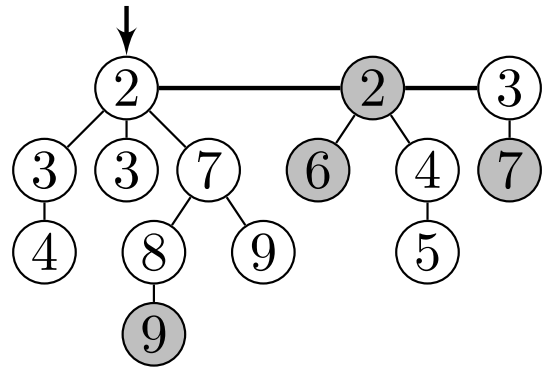












Diminuer une valeur

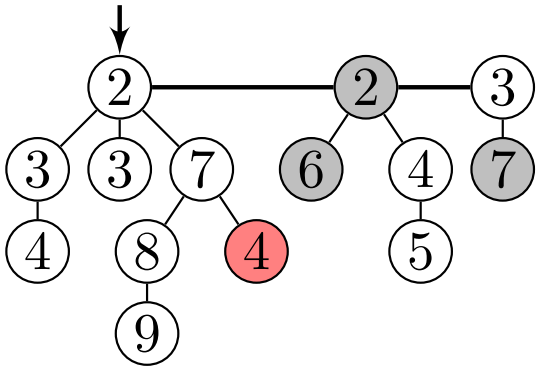
Change la valeur si la nouvelle valeur est inférieure ;

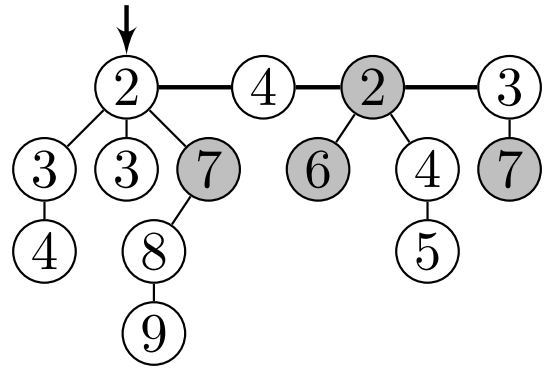
Si la valeur du sommet est inférieure à celle de son père alors :

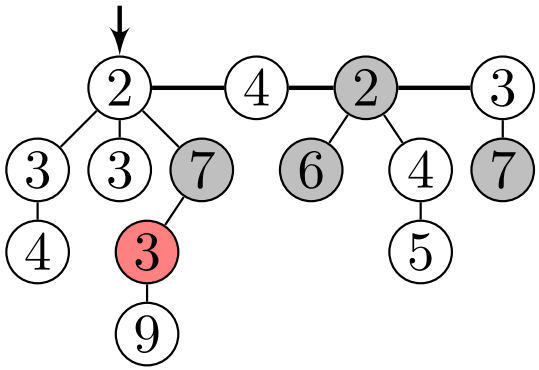
1. le sommet est détaché de son père et devient une racine démarquée ;
2. si le père est une racine on ne fait rien ;
3. sinon si le père n'est pas marqué alors on marque le père ;
4. sinon on repart à l'étape 1 avec le père comme nouveau sommet.

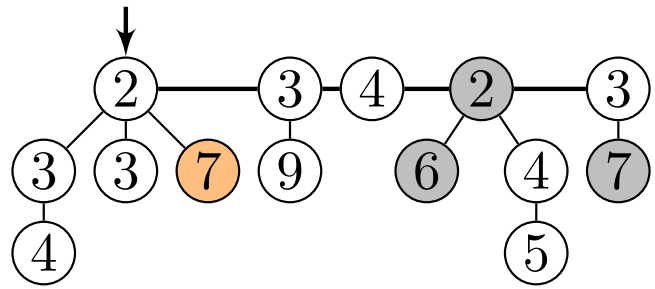
Pour supprimer une valeur, on procède comme précédemment.

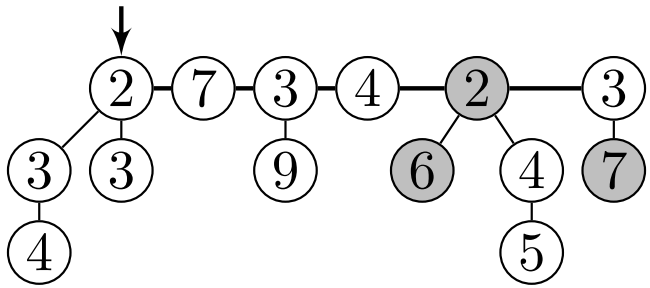
Diminuer une valeur : exemple











Degré maximal d'un arbre de Fibonacci

Résultats préliminaires.

Soit F_n le n^{ieme} terme de la suite de Fibonacci. Alors pour tout $n \in \mathbb{N}$:

- $F_{n+2} \geq \Phi^n$ avec $\Phi = \frac{1+\sqrt{5}}{2}$;
- $F_{n+2} = 1 + \sum_{0 \leq i \leq n} F_i$.

Le degré d'un sommet d'un tas de Fibonacci obtenu par les opérations standard est au plus logarithmique en la taille du tas.

Preuve.

Soit x un sommet et $y_1, \ldots, y_k$ ses fils par ordre d'attachement.

Lorsque y_i $(i \geq 2)$ est attaché $deg(y_i) = deg(x) \geq i - 1$.

Il perd ensuite au plus un fils d'où $deg(y_i) \geq i - 2$.

Soit s_k la plus petite taille d'un sous-arbre de Fibonacci dont la racine a degré k.

$(s_k)_{k \in \mathbb{N}}$ est croissante. Soit $y_1, \ldots, y_k$ les fils d'un arbre de taille minimale

$$s_k \geq 2 + \sum_{2 \leq i \leq k} s_{deg(y_i)} \geq 2 + \sum_{2 \leq i \leq k} s_{i-2}$$

Par induction, on obtient $s_k \geq F_{k+2} \geq \Phi^k$. D'où $k \leq \log_\Phi(s_k) \leq \log_\Phi(n)$.

Analyse de complexité amortie (1)

Potentiel. Soit T un tas de Fibonacci.

$a(T)$ dénote le nombre d'arbres du tas et $m(T)$ le nombre de sommets marqués.

Le potentiel est défini par $pot(T) = C(a(T) + 2m(T))$ pour un C approprié.

Résultats. La complexité amortie des opérations est la suivante :

- L'insertion, l'union et la diminution de valeur s'effectuent en temps constant.
- L'extraction du minimum s'effectue en temps logarithmique.

Analyse de complexité amortie (2)

Insertion. s'effectue en temps constant la différence de potentiel est égale à C.

Union. s'effectue en temps constant la différence de potentiel est nulle.

Extraction du minimum. Soit d le degré du sommet minimal.

- L'insertion des d fils dans les racines se fait en $O(d) = O(\log(n))$;
- L'initialisation du tableau des degrés s'effectue en $O(\log(n))$;
- Soit T (resp. T') le tableau au début (resp. à la fin) du parcours des racines.
  La compaction des racines s'effectue en $O(a(T))$.
  La différence de potentiel est : $C(a(T') - a(T))$.
  D'où un coût amorti borné par $Ca(T') \leq C \log_\Phi(n)$.

Diminution d'une valeur. Soit m le nombre de sommets détachés.

- Le calcul s'effectue en $O(m+1)$;
- Il y a m nouvelles racines ;
- $m-1$ sommets ont perdu leur marque et au plus un sommet a été marqué ;
  D'où une différence de potentiel au plus $C(4-m)$;
  Le coût amorti est donc en $O(1)$.

Pseudo-matroïde versus matroïde

(E, Adm) **est un matroïde si :**

- **Cloture.** Adm est non vide et clos par inclusion :
  $E' \in Adm \wedge E'' \subseteq E' \Rightarrow E'' \in Adm$;
- **Extension.** Si $E', E'' \in Adm$ et $|E'| < |E''|$,
  il existe $e \in E'' \setminus E'$ tel que $E' \uplus \{e\} \in Adm$.

(E, Adm) **est un pseudo-matroïde si :**

- $\emptyset \in Adm$;
- Si E'' maximal dans Adm, $E' \in Adm$ et $|E'| < |E''|$
  alors il existe $e \in E'' \setminus E'$ tel que $E' \cup \{e\} \in Adm$.
- Si E'' maximal dans Adm, $E', E' \cup \{e\} \in Adm$, $E' \subsetneq E''$, $e \notin E''$
  alors il existe $f \in E'' \setminus E'$ tel que $E' \cup \{f\}, E'' \cup \{e\} \setminus \{f\} \in Adm$.

Un matroïde est un pseudo-matroïde

Preuve de la troisième propriété.

Puisque $E' \cup \{f\} \subseteq E''$, $E' \cup \{f\} \in Adm$ *(par cloture)*

- Si $|E'' \setminus E'| = 1$ alors $E'' = E' \cup \{f\}$ pour un certain f.
  Donc $E'' \cup \{e\} \setminus \{f\} = E' \cup \{e\} \in Adm$.

- Sinon $|E' \cup \{e\}| < |E''|$.
  Il existe $f_1 \in E'' \setminus (E' \cup \{e\})$ tel que $E' \cup \{e, f_1\} \in Adm$ *(par extension)*
  En itérant on obtient $f_0, f_1, \ldots, f_k$ tel que
  $E' \cup \{e, f_1, \ldots, f_k\} \in Adm$ et $E'' = E' \cup \{f_0, f_1, \ldots, f_k\}$.
  f_0 est l'élément recherché.

Un algorithme pour pseudo-matroïdes

Observation. Les ensembles indépendants maximaux ont la même taille.

Un algorithme générique ($Adm \neq \{\emptyset\}$).

$E' \leftarrow \emptyset$;
Repeat
  $max \leftarrow -\infty$;
  For all $e \in E \setminus E'$ **do**
    If $E' \cup \{e\} \in Adm$ **and** $r(e) > max$ **then** $max \leftarrow r(e)$; $emax \leftarrow e$;
  $E' \leftarrow E' \cup \{emax\}$;
Until $|E'|$ est maximal ;
Return(E') ;

Preuve de l'algorithme

Invariant de boucle.

> E' est inclus dans un ensemble indépendant de récompense maximale E^*.

Cas 1 $emax \in E^*$.

Cas 2 $emax \notin E^*$.

Alors il existe $f \in E^* \setminus E'$ tel que $E' \cup \{f\}, E^* \cup \{emax\} \setminus \{f\} \in Adm$.

Puisque $E' \cup \{f\} \in Adm$, $r(emax) \geq r(f)$.

Ainsi la récompense de $E^* \cup \{emax\} \setminus \{f\}$ est supérieure ou égale à celle de E^*.

Elle est donc maximale.

Arbres d'un graphe connexe

Soit $G = (S, A)$ un graphe connexe.
Alors les arbres de G forment un pseudo-matroïde.

• Soit $\mathcal{T}$ un arbre (non couvrant) et $\mathcal{T}'$ un arbre couvrant.

Notons $S_{\mathcal{T}}$ les sommets couverts par $\mathcal{T}$.

Puisque $\mathcal{T}'$ est un arbre couvrant
il existe au moins une arête $\{s, s'\} \in \mathcal{T}'$ telle que $s \in S_{\mathcal{T}}$ et $s' \notin S_{\mathcal{T}}$.

Par conséquent, $\mathcal{T} \cup \{s, s'\}$ est un arbre.

• Soit $\mathcal{T}$ un arbre (non couvrant) inclus dans $\mathcal{T}'$ un arbre couvrant
et une arête $\{s, s'\} \notin \mathcal{T}'$ telle que $\mathcal{T} \cup \{\{s, s'\}\}$ soit un arbre.

$s \in S_{\mathcal{T}}$ et $s' \notin S_{\mathcal{T}}$.

$\mathcal{T}' \cup \{\{s, s'\}\}$ contient un cycle $\rho\{s, s'\}\rho'\{s'_1, s_1\}$ avec $s_1 \in S_{\mathcal{T}}$ et $s'_1 \notin S_{\mathcal{T}}$.

$\mathcal{T}' \cup \{\{s, s'\}\} \setminus \{\{s_1, s'_1\}\}$ reste connexe (donc un arbre)
en raison du chemin de s_1 à s'_1, $\rho\{s, s'\}\rho'$.

Algorithme de Prim

L'algorithme de Prim est une implémentation efficace de l'algorithme générique.

For all $s \in S \setminus \{r\}$ **do** Insérer$(Fib, (s, \infty))$; r *est une racine arbitraire*
Insérer$(Fib, (r, 0))$; $p[r] \leftarrow r$;
Repeat
  $(s, v) \leftarrow$ ExtraireMin(Fib) ;
  For all $\{s, s'\} \in A$ **do**
    If $(s', v') \in Fib$ **and** $w(\{s, s'\}) < v'$ **then** Diminuer$(Fib, s', w(\{s, s'\}))$; $p[s'] \leftarrow s$;
Until Vide(Fib) ;

Il y a $|S|$ extractions de minimum en $O(|S| \log(|S|))$

et $|A|$ diminution de valeurs en $O(|A|)$.

D'où une complexité $O(|S| \log(|S|) + |A|)$.

Plan

Ensembles disjoints

On maintient une famille d'ensembles disjoints et non vides de clés.

Les opérations usuelles sont :

- CréeSingleton(clé) qui crée un ensemble composé d'une unique clé si cette clé n'est pas déjà présente dans un sous-ensemble ;
- Ensemble(clé) qui renvoie l'ensemble auquel appartient la clé ;
- Union(clé$_1$,clé$_2$) qui effectue l'union des ensembles auxquels appartiennent les deux clés.

Quelques hypothèses :

- L'espace des clés peut être identifié à $\{1, \ldots, N\}$ pour un certain N ;
- L'identifiant d'un sous-ensemble est l'une de ses clés appelée le *représentant*.

Calcul des composantes connexes

ComposantesConnexes(V, E)

For $v \in V$ **do** CréeSingleton(v) ;

For $\{u, v\} \in E$ **do**

If Ensemble$(u) \neq$ Ensemble(v) **then** Union(u, v) ;

Observations.

Pas forcément le plus efficace car les parcours de graphe construisent en temps linéaire les composantes connexes.

Algorithme de Kruskal

L'algorithme de Kruskal implémente l'algorithme générique des matroïdes pour la construction d'un arbre couvrant de poids minimal.

ArbreCouvrant(V, E)

For $v \in V$ **do** CréeSingleton(v) ;

Trier E par récompense croissante ;

$F \leftarrow E$; $E' \leftarrow \emptyset$;

Repeat

$\{u, v\} \leftarrow$ Extraire(F) ;

If Ensemble$(u) \neq$ Ensemble(v) **then**

$E' \leftarrow E' \cup \{\{u, v\}\}$; Union(u, v) ;

Until $F = \emptyset$;

Return(E') ;

Le tri des arêtes s'effectue en $O(|E| \log(|V|))$.

Quid du reste de l'algorithme ?

Initialisation paresseuse d'un tableau

Comment savoir si une clé est présente ?

- a priori un tableau T de taille N de booléens initialisé à **false** ;
- mais peut-on éviter l'initialisation en $\Theta(N)$?

Solution générale pour éviter une initialisation par défaut

- Deux tableaux non initialisés $Tind$, $Tval$ et un compteur cpt initialisé à 0 ;
- cpt contient le nombre de cellules mises à jour ;
- $Tind$ contient des indices de $Tval$;
- $Tval$ contient des paires (indice de $Tind$,valeur) ;

Illustration

Lire(i)

$j \leftarrow Tind[i]$;

If $0 < j \leq cpt$ **and** $Tval[j].ind = i$ **then return** $Tval[j].val$ **else return** *vdef* ;

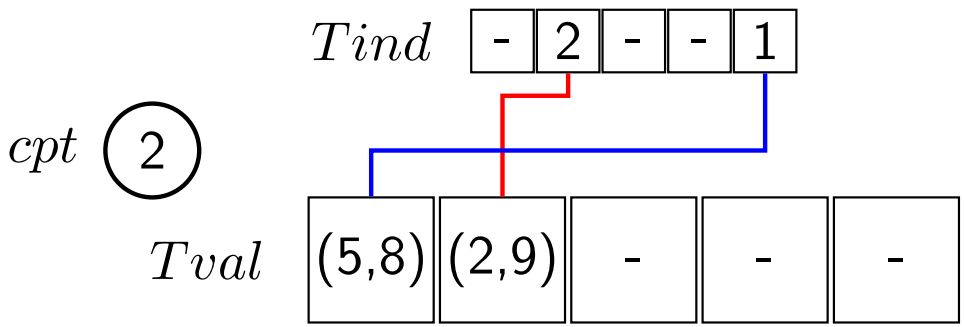
Ecrire(i, v)

$j \leftarrow Tind[i]$;

If $0 < j \leq cpt$ **and** $Tval[j].ind = i$ **then** $Tval[j].val \leftarrow v$

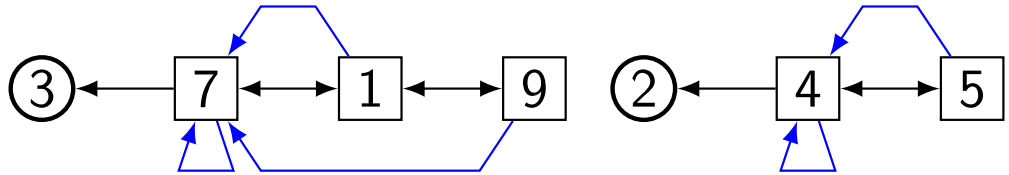
Else $cpt \leftarrow cpt + 1$; $Tind[i] \leftarrow cpt$; $Tval[cpt].ind \leftarrow i$; $Tval[cpt].val \leftarrow v$;

Lire(3) *(renvoie 0)* Ecrire(5, 8) Ecrire(2, 9) Lire(5) *(renvoie 8)*



Gestion d'ensembles par liste

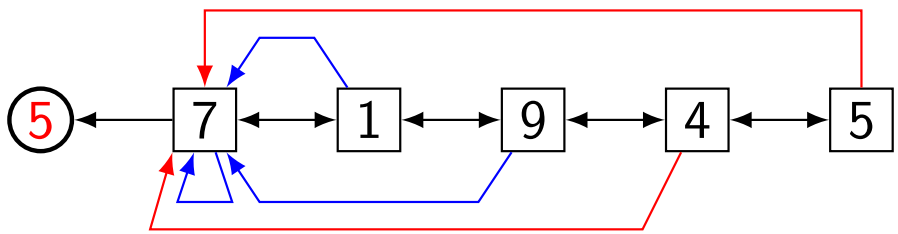
Une liste par ensemble dont la tête de liste est le représentant.



CréeSingleton crée une liste en $O(1)$.

Ensemble renvoie l'identifiant de la tête de liste en $O(1)$.

Union joint la petite liste à la grande.



Sa complexité au pire est en $O(n)$.

Complexité amortie

Soit n appels à CréeSingleton et m appels à Union.
Alors la complexité des m Union est en $O(m + n\log(n))$.

Preuve.

Soit i un élément associé à un singleton.

Chaque fois que i change de représentant, la taille de l'ensemble double.

Donc un élément ne peut changer de représentant qu'au plus $\log(n)$ fois.

En dehors des mises à jour des représentants,

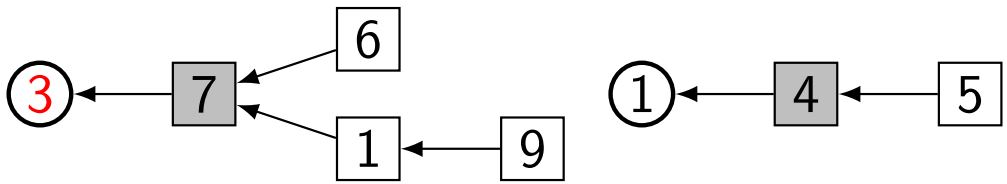
l'exécution d'une union se fait en $O(1)$.

D'où $O(m + n\log(n))$.

Observation. Cette gestion est suffisante pour l'algorithme de Kruskal car le tri des arêtes est en $O(|A|\log(|S|))$.

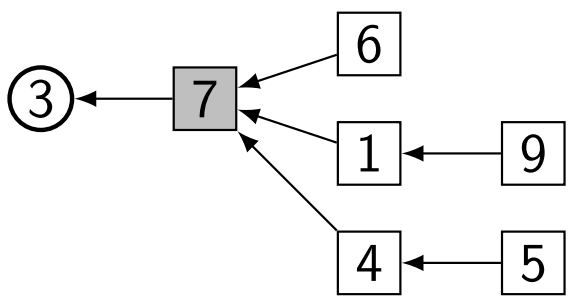
Gestion d'ensembles par arborescence

Une arborescence par ensemble dont la racine est le représentant,
incluant un *rang* qui est un majorant de la hauteur.



CréeSingleton crée une liste en $O(1)$.

Union joint l'arbre de petit rang à l'autre en $O(1)$.



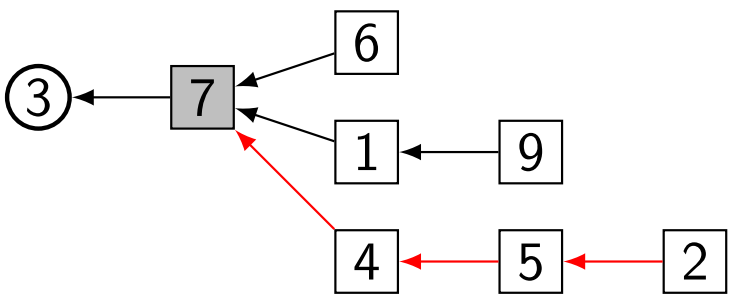
Compression de chemin

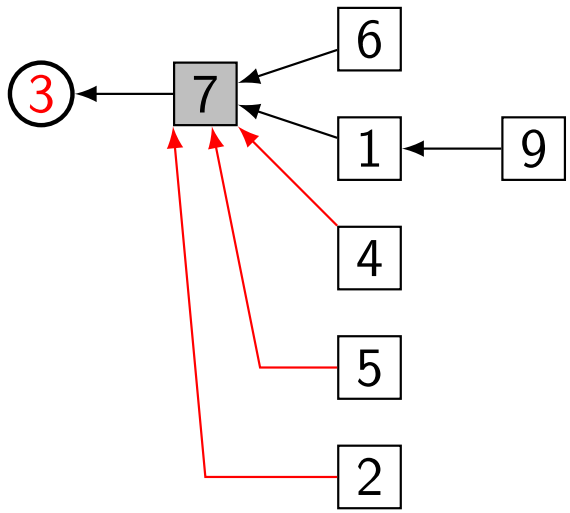
Ensemble renvoie l'identifiant de la racine en $O(n)$

en compressant le chemin dans l'arbre,

sans recalculer le rang (d'où la majoration).

Illustration. Ensemble(2)





Une fonction à croissance lente

Soit la famille de fonctions $\{A_k\}_{k\in\mathbb{N}}$ de $\mathbb{N}^*$ dans $\mathbb{N}$ définies par

- $A_0(n) = n + 1$;
- $A_{k+1}(n) = A_k^{(n+1)}(n)$.

Quelques résultats.

$A_1(n) = 2n + 1$, $A_2(n) = 2^{n+1}(n + 1) - 1$, $A_3(1) = 2047$, $A_4(1) \gg 10^{80}$.

On définit α, « l'inverse » d'une fonction d'Ackerman, par :

$$\alpha(n) = \min(k \mid A_k(1) \geq n)$$

Ainsi pour $n \leq 10^{80}$, $\alpha(n) \leq 4$.

Croissance asymptotique.

Pour tout $k \in \mathbb{N}$, $\alpha(n) = o(\lfloor \log^{(k)}(n) \rfloor)$.

Rang d'un sommet

On considère que chaque sommet conserve son rang une fois qu'il n'est plus racine.

Observations.

- Le rang d'un sommet non racine x
  est strictement inférieur à celui de son père $p(x)$;
- Le rang du père d'un sommet non racine est croissant avec le temps
  même lors d'un changement de père ;
- Le rang d'un sommet est strictement inférieur à n, le nombre de clés
  à la fin des opérations.

Niveau et itération d'un sommet

Niveau d'un sommet *interne* (i.e. non racine et non feuille).

$$Niv(x) = \max(k \mid A_k(rang(x)) \leq rang(p(x)))$$

- Le rang du père d'un sommet non racine est croissant avec le temps même lors d'un changement de père ;
- $Niv(x) \geq 0$ puisque $rang(x) < rang(p(x))$;
- $Niv(x) < \alpha(n)$ puisque :

$$rang(p(x)) < n \leq A_{\alpha(n)}(1) \leq A_{\alpha(n)}(rang(x))$$

Itération d'un sommet interne.

$$Iter(x) = \max(j \mid A^{(j)}_{Niv(x)}(rang(x)) \leq rang(p(x)))$$

- $Iter(x) \geq 1$ par définition de $Niv(x)$;
- $Iter(x) \leq rang(x)$ puisque :

$$rang(p(x)) < A_{Niv(x)+1}(rang(x)) = A^{(rang(x)+1)}_{Niv(x)}(rang(x))$$

Potentiel

Le potentiel d'un sommet est défini ainsi (avec C une constante appropriée) :

- Si x est une racine ou une feuille,

$$pot(x) = C\alpha(n)rang(x)$$

- Si x est un sommet interne,

$$pot(x) = C((\alpha(n) - Niv(x))rang(x) - Iter(x))$$

Le potentiel d'un sommet :

- s'il est racine, croît avec son rang ;
- S'il est interne, décroît lorsque le rang de son père croît (suffisamment).

D'après les inégalités sur le niveau et l'itération,
le potentiel d'un sommet est toujours positif.

Le potentiel d'un état est la somme des potentiels des sommets.

Coût amorti des opérations (1)

CréeSingleton a un coût amorti en $O(1)$ car la différence de potentiel est nulle.

Union(x,y) a un coût amorti en $O(\alpha(n))$.

Preuve. *(on suppose que x devient fils de y)*

Le coût non amorti est en $O(1)$.

Soit z un fils de y son potentiel ne peut croître.

Le potentiel de x ne peut croître car son rang est inchangé.

Le potentiel de y croît au plus de $C\alpha(n)$ car son rang croît d'au plus 1.

Coût amorti des opérations (2)

Ensemble(x) a un coût amorti en $O(\alpha(n))$.

Preuve. *(on suppose que le chemin de x à sa racine a ℓ sommets)*

Le coût non amorti est en $O(\ell)$.

Sur le chemin de x à la racine, il y au plus $\alpha(n)$ niveaux différents.
Il y a donc au moins $\ell - \alpha(n) - 2$ sommets internes y
pour lesquels il existe un sommet interne $z \neq y$ sur le chemin de y à la racine avec $Niv(z) = Niv(y)$. Notons k ce niveau et $i = Iter(y)$.

$$rang(p(z)) \geq A_k(rang(z)) \geq A_k(rang(p(y))) \geq A_k(A_k^{(i)}(rang(y)))$$

Après la compression,
$rang'(p(y)) = rang'(p(z)) \geq rang(p(z))$ et $rang'(y) = rang(y)$.

Donc $rang'(p(y)) \geq A_k^{(i+1)}(rang'(y))$. Ce qui implique :
Soit $Niv'(y) > Niv(y)$, soit $Niv'(y) = Niv(y)$ et $Iter'(y) > Iter(y)$.

Par conséquent le potentiel de $\ell - \alpha(n) - 2$ sommets décroît d'au moins C.
Ce qui fournit un coût amorti en $O(\alpha(n))$.