Structures de données persistantes, deuxième cours

Arbres équilibrés + copie de branches = dictionnaires persistants

Xavier Leroy

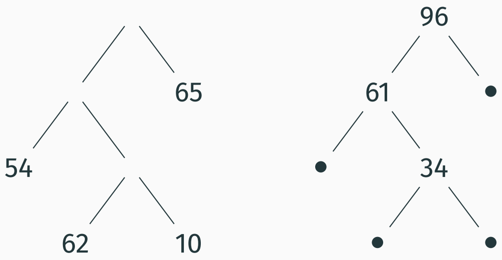
2023-03-16

Collège de France, chaire de sciences du logiciel
`xavier.leroy@college-de-france.fr`

Arbres binaires de recherche

Arbres binaires

Un arbre = une feuille ou un nœud qui porte 2 sous-arbres.



Porte des informations aux feuilles ou aux nœuds.

Informations = éléments $\Longrightarrow$ ensembles finis

Informations = paires (clé, valeur) $\Longrightarrow$ dictionnaires, *maps*

La vision algébrique

Avec des éléments $x \in X$ aux feuilles :

$$\begin{aligned} A ::= {} & [x] && \text{feuille} \\ \mid {} & \langle A_1, A_2 \rangle && \text{nœud} \end{aligned}$$

Avec des éléments $x \in X$ aux nœuds :

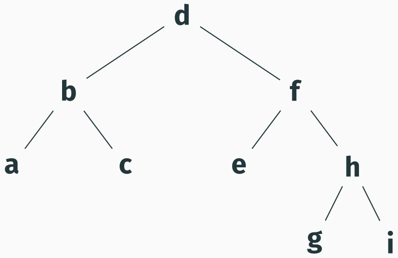
$$\begin{aligned} A ::= {} & \bullet && \text{feuille} \\ \mid {} & \langle A_1, x, A_2 \rangle && \text{nœud} \end{aligned}$$

Transcription directe sous forme de type algébrique (OCaml, Haskell, ...) ou type inductif (Coq, Agda, ...) :

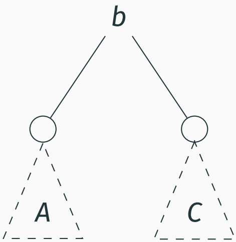
```
type 'a tree = Leaf of 'a | Node of 'a tree * 'a tree
type 'a tree = Leaf | Node of 'a tree * 'a * 'a tree
```

Les arbres binaires de recherche

Un arbre binaire de recherche =
un arbre binaire portant des éléments aux nœuds
tel que les éléments vont strictement croissant
de la gauche vers la droite (parcours infixe).



(Note : on ne dessine pas les branches vers les feuilles.
La ≪feuille≫ **a** est le nœud trivial $\langle \bullet, \mathbf{a}, \bullet \rangle$)



Invariant inductif : pour tout nœud $\langle A, b, C \rangle$,
les éléments du sous-arbre gauche A sont $< b$
les éléments du sous-arbre droit C sont $> b$

Recherche dichotomique dans un A.B.R.

Appartenance à un ensemble :

$$\mathtt{mem}(x, \bullet) = \mathtt{false}$$

$$\mathtt{mem}(x, \langle A, b, C\rangle) = \begin{cases} \mathtt{mem}(x, A) & \text{si } x < b \\ \mathtt{true} & \text{si } x = b \\ \mathtt{mem}(x, C) & \text{si } x > b \end{cases}$$

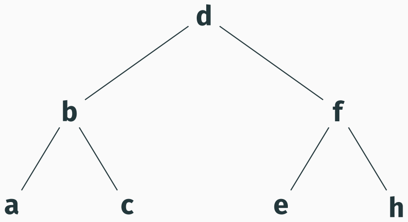
Consultation d'un dictionnaire :

$$\mathtt{find}(x, \bullet) = \mathtt{None}$$

$$\mathtt{find}(x, \langle A, (k, v), C\rangle) = \begin{cases} \mathtt{find}(x, A) & \text{si } x < k \\ \mathtt{Some}(v) & \text{si } x = k \\ \mathtt{find}(x, C) & \text{si } x > k \end{cases}$$

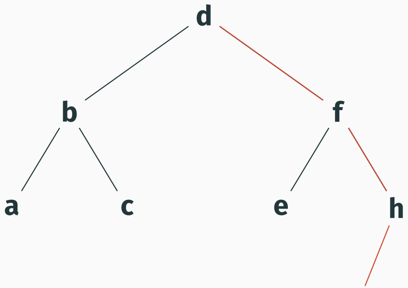
Temps : $\mathcal{O}(h)$ où h est la hauteur de l'arbre, c.à.d. la longueur maximale d'une branche.

Insertion dans un A.B.R. : version impérative



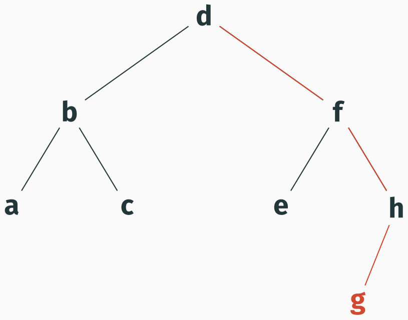
1. Rechercher l'élément à insérer (ici, **g**) dans l'arbre.

Insertion dans un A.B.R. : version impérative



1. Rechercher l'élément à insérer (ici, **g**) dans l'arbre.

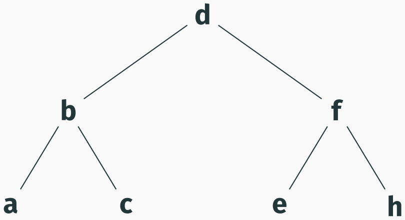
Insertion dans un A.B.R. : version impérative



1. Rechercher l'élément à insérer (ici, **g**) dans l'arbre.
2. Quand on tombe sur une feuille, la remplacer par le nœud $\langle \bullet, \mathbf{g}, \bullet \rangle$.

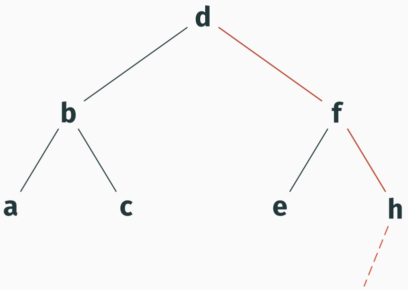
Temps : $\mathcal{O}(h)$, espace : $\mathcal{O}(1)$.

Insertion dans un A.B.R. : version persistante



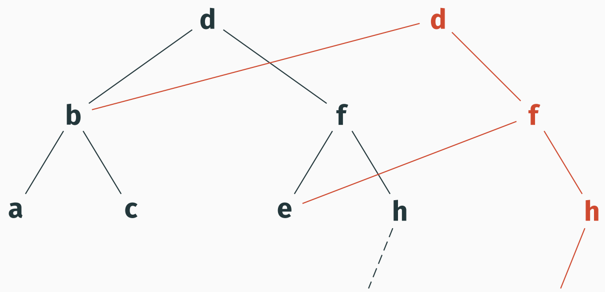
1. Rechercher l'élément à insérer (ici, **g**) dans l'arbre.

Insertion dans un A.B.R. : version persistante



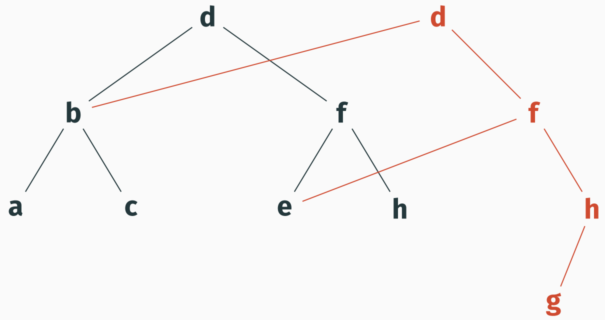
1. Rechercher l'élément à insérer (ici, **g**) dans l'arbre.

Insertion dans un A.B.R. : version persistante



1. Rechercher l'élément à insérer (ici, **g**) dans l'arbre.
2. Quand on tombe sur une feuille, copier le chemin de la racine vers cette feuille, en partageant les sous-arbres de l'arbre initial.

Insertion dans un A.B.R. : version persistante



1. Rechercher l'élément à insérer (ici, **g**) dans l'arbre.
2. Quand on tombe sur une feuille, copier le chemin de la racine vers cette feuille, en partageant les sous-arbres de l'arbre initial.
3. A la fin du chemin copié, ajouter le nœud $\langle \bullet, \mathbf{g}, \bullet \rangle$.

Temps : $\mathcal{O}(h)$, espace : $\mathcal{O}(h)$.

Insertion dans un A.B.R. : présentation algébrique

Ajout à un ensemble :

$$\mathtt{add}(x, \bullet) = \langle \bullet, x, \bullet \rangle$$

$$\mathtt{add}(x, \langle A, b, C \rangle) = \begin{cases} \langle \mathtt{add}(x, A), b, C \rangle & \text{si } x < b \\ \langle A, x, C \rangle & \text{si } x = b \\ \langle A, b, \mathtt{add}(x, C) \rangle & \text{si } x > b \end{cases}$$

Insertion dans un dictionnaire :

$$\mathtt{ins}(k, v, \bullet) = \langle \bullet, (k, v), \bullet \rangle$$

$$\mathtt{ins}(k, v, \langle A, (k', v'), C \rangle) = \begin{cases} \langle \mathtt{ins}(k, v, A), (k', v'), C \rangle & \text{si } k < k' \\ \langle A, (k, v), C \rangle & \text{si } k = k' \\ \langle A, (k', v'), \mathtt{ins}(x, C) \rangle & \text{si } k > k' \end{cases}$$

Insertion dans un A.B.R. : implémentation fonctionnelle pure

```
let rec add x t =
  match t with
  | Leaf -> Node(Leaf, x, Leaf)
  | Node(a, b, c) ->
      if x < b then
        Node(add x a, b, c)
      else if x > b then
        Node(a, b, add x c)
      else
        t
```

La «copie de chemin» s'effectue automatiquement lors de la remontée de la récursion, par évaluation des expressions `Node(add x a, b, c)` ou `Node(a, b, add x c)`.

Insertion dans un A.B.R. : spécification algébrique et vérification

Une spécification équationnelle simple :

$$\mathtt{mem}(x, \mathtt{add}(x, T)) = \mathtt{true} \tag{1}$$

$$\mathtt{mem}(x, \mathtt{add}(y, T)) = \mathtt{mem}(x, T) \qquad \text{si } x \neq y \tag{2}$$

Une démonstration par récurrence structurelle sur T.

Cas de base pour (1) : $\mathtt{mem}(x, \mathtt{add}(x, \bullet)) = \mathtt{true}$.

Cas inductif pour (1) : on déroule les définitions et on analyse les 3 cas

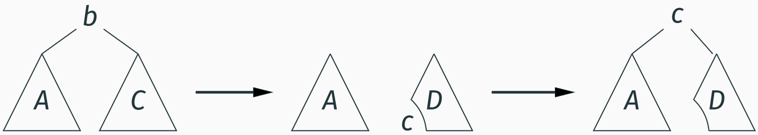
$$\mathtt{mem}(x, \mathtt{add}(x, \langle A, b, C\rangle)) = \begin{cases} \mathtt{mem}(x, \mathtt{add}(x, A)) & \text{si } x < b \\ \mathtt{true} & \text{si } x = b \\ \mathtt{mem}(x, \mathtt{add}(x, C)) & \text{si } x > b \end{cases}$$

d'où le résultat par hypothèse de récurrence.

Exercice : montrer (2).

Une fois trouvé le sous-arbre $\langle A, b, C\rangle$ qui porte la clé b à enlever, il faut le remplacer par un arbre qui ≪fusionne≫ A et C.

Idée : utiliser le plus petit élément de C (ou le plus grand élément de A) comme racine de ce sous-arbre.



Suppression dans un A.B.R.

$$\mathtt{del}(x, \bullet) = \bullet$$

$$\mathtt{del}(x, \langle A, b, C\rangle) = \begin{cases} \langle \mathtt{del}(x, A), b, C\rangle & \text{si } x < b \\ \mathtt{join}(A, C) & \text{si } x = b \\ \langle A, b, \mathtt{del}(x, C)\rangle & \text{si } x > b \end{cases}$$

$$\mathtt{join}(A, \bullet) = \mathtt{join}(\bullet, A) = A$$

$$\mathtt{join}(A, C) = \langle A, \mathtt{min}(C), \mathtt{delmin}(C)\rangle \qquad \text{si } C \neq \bullet$$

$$\mathtt{min}(\langle \bullet, b, C\rangle) = b \qquad \mathtt{delmin}(\langle \bullet, b, C\rangle) = C$$

$$\mathtt{min}(\langle A, b, C\rangle) = \mathtt{min}(A) \qquad \mathtt{delmin}(\langle A, b, C\rangle) = \langle \mathtt{delmin}(A), b, C\rangle$$

Autres opérations

Plus petit élément, plus grand élément
$\Rightarrow$ utilisable comme file de priorité.

Si chaque sous-arbre est annoté par sa taille :
quel élément en position k ? quelle position pour l'élément x ?
$\Rightarrow$ utilisable comme séquence ordonnée.

Opérations ensemblistes : union, intersection, différence, …

Opérations ensemblistes

Récurser sur un des deux arbres et partitionner l'autre.

$$\begin{aligned} \texttt{union}(\bullet, T) &= \texttt{union}(T, \bullet) = T \\ \texttt{union}(\langle A, b, C\rangle, T) &= \langle \texttt{union}(A, A'), b, \texttt{union}(C, C')\rangle \\ &\quad \text{avec } (A', C') = \texttt{split}(T, b) \end{aligned}$$

La fonction $\texttt{split}(T, b)$ renvoie deux arbres A' et C' :
A' contient tous les éléments de T qui sont $< b$
C' contient tous les éléments de T qui sont $> b$.

Exercice : définir `split`, l'intersection, la différence ensembliste.

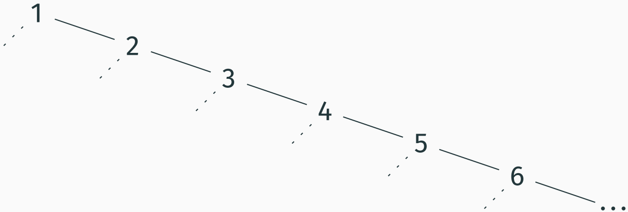
Arbres AVL

Équilibrage des arbres

Les opérations sur les A.B.R. sont efficaces tant que l'arbre est équilibré : la hauteur h est petite devant le nombre de nœuds n.

Idéalement, on vise $h = \mathcal{O}(\log n)$.

Cependant, certains A.B.R. sont complètement déséquilibrés :

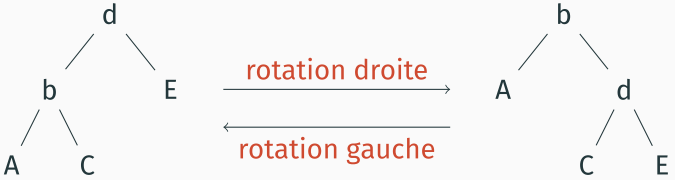


Un tel arbre s'obtient naturellement en insérant successivement les éléments $1, 2, \ldots n$. Les opérations sont alors en temps $\mathcal{O}(n)$, comme pour une liste triée.

Arbres auto-équilibrants

Il faut modifier les opérations (insertion, destruction, etc) pour garantir que l'arbre reste équilibré (hauteur en $\log n$) après n'importe quelle séquence d'opérations.

Idée : à tout moment, on peut effectuer des rotations sur des sous-arbres d'un arbre binaire de recherche



Les rotations préservent la propriété A.B.R. (≪croissance de gauche à droite) et peuvent réduire un déséquilibre.

Nommés d'après leurs auteurs, Georgii **A**delson-**V**elskii et Evgueni **L**andis.

G. Adelson-Velskii, E. Landis. An algorithm for the organization of information. *Doklady Akademii Nauk SSSR* 146 (1962), 263-266 ; traduction anglaise dans *Soviet Mathematics Doklady* 3 (1962), 1259-1262.

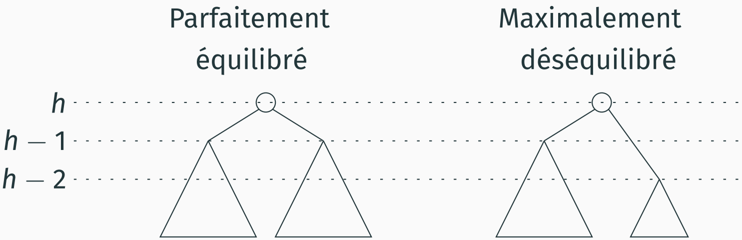
Le critère AVL : un critère d'équilibrage basé sur les hauteurs des sous-arbres.

> *Pour tout nœud $\langle A, b, C\rangle$, les hauteurs des sous-arbres A et C diffèrent d'au plus 1 :*

$$|h(A) - h(C)| \leq 1$$

Les arbres AVL

Pour tout nœud $\langle A, b, C \rangle$, les hauteurs des sous-arbres A et C diffèrent d'au plus 1 : $|h(A) - h(C)| \leq 1$

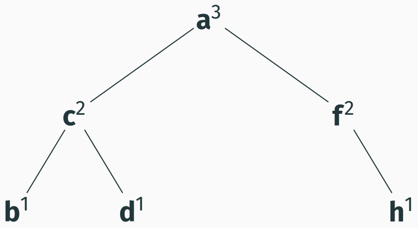


$N(h) = 1 + 2 \times N(h-1)$ $\qquad$ $N(h) = 1 + N(h-1) + N(h-2)$

$N(h) = 2^h - 1$ $\qquad$ $N(h) = 2F_h - 1$ (Fibonacci)

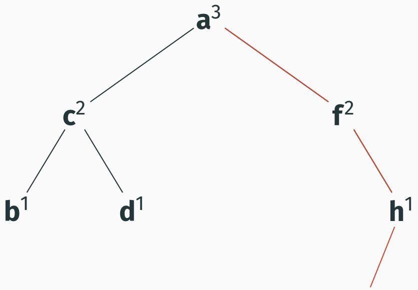
La hauteur h est bien logarithmique en la taille $n = N(h)$ de l'arbre : $h < \frac{3}{2} \log_2(n+1)$.

Insertion dans un AVL : version impérative



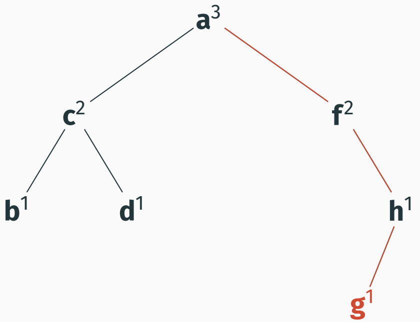
1. Rechercher l'élément à insérer (ici, **g**) dans l'arbre.

Insertion dans un AVL : version impérative



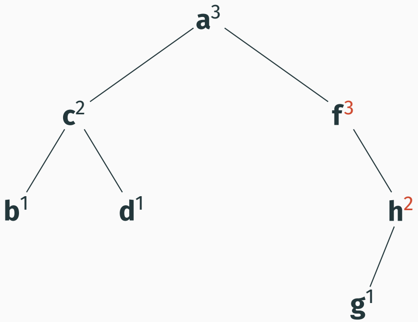
1. Rechercher l'élément à insérer (ici, **g**) dans l'arbre.

Insertion dans un AVL : version impérative



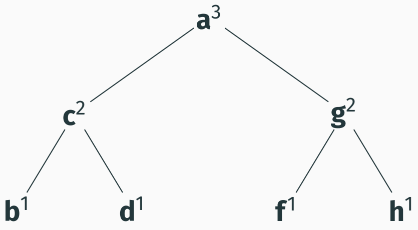
1. Rechercher l’élément à insérer (ici, **g**) dans l’arbre.
2. Remplacer la feuille par le nœud $\langle \bullet, \mathbf{g}, \bullet \rangle$.

Insertion dans un AVL : version impérative



1. Rechercher l'élément à insérer (ici, **g**) dans l'arbre.
2. Remplacer la feuille par le nœud $\langle \bullet, \mathbf{g}, \bullet \rangle$.
3. Remonter la branche en mettant à jour les hauteurs …

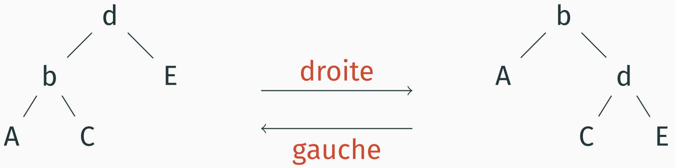
Insertion dans un AVL : version impérative



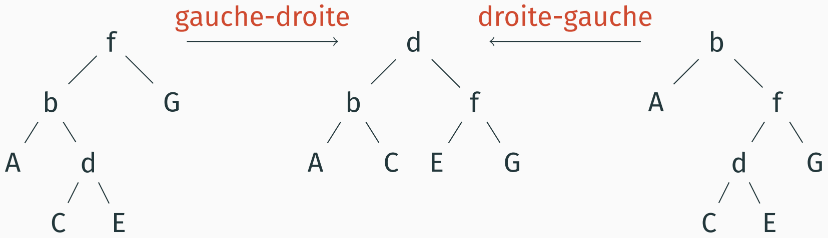
1. Rechercher l'élément à insérer (ici, **g**) dans l'arbre.
2. Remplacer la feuille par le nœud $\langle \bullet, \mathbf{g}, \bullet \rangle$.
3. Remonter la branche en mettant à jour les hauteurs …
4. … et en effectuant des rotations pour rétablir le critère AVL.

Les rotations

Rotations simples : droite (si A trop haut), gauche (si E trop haut).



Rotations doubles : si $\langle C, d, E \rangle$ est trop haut.



Présentation algébrique

Un *smart constructor* $\texttt{bal}(A, b, C)$ qui construit un ABR équivalent à $\langle A, b, C\rangle$, mais fait les rotations nécessaires pour que ce soit un AVL (en supposant $|h(A) - h(C)| \leq 2$ initialement).

$$\begin{aligned}
\texttt{bal}(A, b, C) &= \langle A, b, C\rangle \quad \text{si } |h(A) - h(C)| \leq 1 \\
\texttt{bal}(\langle A, b, C\rangle, d, E) &= \langle A, b, \langle C, d, E\rangle\rangle \quad \text{si } h(A) > h(C), h(A) > h(E) \\
\texttt{bal}(\langle A, b, \langle C, d, E\rangle\rangle, f, G) &= \langle\langle A, b, C\rangle, d, \langle E, f, G\rangle\rangle \\
&\qquad \text{si } \max(h(C), h(E)) > h(A) > h(G)
\end{aligned}$$

(Plus cas symétriques.)

On peut aussi définir $\texttt{bal}^*(A, b, C)$ sans précondition sur $h(A), h(C)$ en itérant $\texttt{bal}$ jusqu'à ce que l'arbre résultat soit AVL.

Insertion dans un AVL : présentation algébrique

$$\mathtt{add}(x, \bullet) = \langle \bullet, x, \bullet \rangle$$

$$\mathtt{add}(x, \langle A, b, C \rangle) = \begin{cases} \mathtt{bal}(\mathtt{add}(x, A), b, C) & \text{si } x < b \\ \langle A, x, C \rangle & \text{si } x = b \\ \mathtt{bal}(A, b, \mathtt{add}(x, C)) & \text{si } x > b \end{cases}$$

Se traduit aussitôt en une implémentation fonctionnelle pure, en temps et en espace $\mathcal{O}(\log n)$ où n est la taille de l'arbre.

Même chose pour la suppression dans un AVL.

Autres critères d'équilibrage

Autres critères d'équilibrage par la hauteur :

Un critère AVL ≪relâché≫ :

$$|h(A) - h(C)| \leq K \qquad \text{pour tout sous-arbre } \langle A, b, C \rangle$$

P.ex. $K = 2$ pour `Set` et `Map` en OCaml
$\rightarrow$ moins de rotations que les AVL mais branches plus longues.

Voir plus loin : les arbres rouge-noir.

Autres critères d'équilibrage

Équilibrage par le poids :

$$\frac{1}{K} \leq \frac{w(A)}{w(C)} \leq K \qquad \text{pour tout sous-arbre } \langle A, b, C \rangle$$

$w(A) = 1 + |A|$ est le poids de A (1 + le nombre d'éléments).

P.ex. $K = 4$ pour `Data.Map` en Haskell.

Critères de rotation délicats ; plusieurs implémentations fausses.
(Hirai et Yamamoto, *Balancing weight-balanced trees*, JFP 21(3), 2011.)

Note : les arbres AVL ne sont pas équilibrés par le poids, car on peut avoir $w(A) \sim 2^h$ (parfaitement équilibré) et $w(C) \sim 2F_h \sim c.\varphi^h$ (maximalement déséquilibré), d'où $w(A)/w(C) \to \infty$ quand $h \to \infty$.

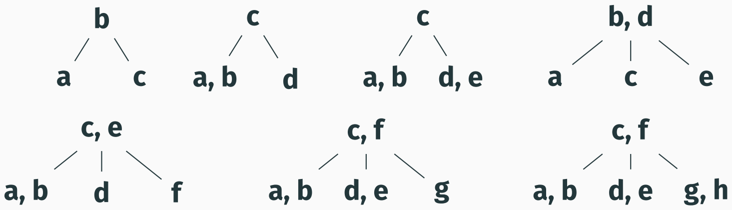
Arbres 2-3

Arbres 2-3

Une approche différente de l'équilibrage :
arbres parfaits + nœuds d'arité variable.

- Toutes les feuilles sont au même niveau.
- Les nœuds portent soit 2 sous-arbres et un élément, soit 3 sous-arbres et deux éléments.

Exemples d'arbres 2-3 à 3, 4, …, 8 éléments :



Une généralisation des arbres 2-3 avec des degrés de branchement élevés :

- Toutes les feuilles sont au même niveau.
- Les nœuds intermédiaires portent entre $k/2$ et k sous-arbres.
- Le nœud du sommet porte entre 2 et k sous-arbres.

k est choisi assez grand pour qu'un nœud = un bloc du disque.

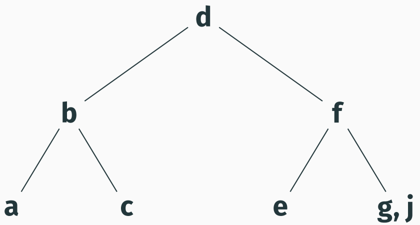
Arbres 2-3 : présentation algébrique, algorithme de recherche

$$
\begin{array}{rll}
A, C, E ::= & \bullet & \text{feuille} \\
\mid & \langle A, b, C \rangle & \text{nœud 2} \\
\mid & \langle A, b, C, d, E \rangle & \text{nœud 3}
\end{array}
$$

Recherche dans un arbre 2-3 : par dichotomie et trichotomie

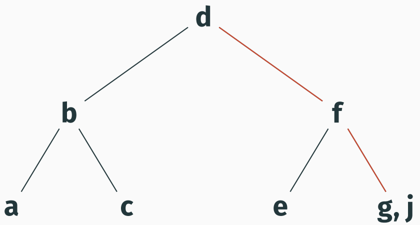
$$
\begin{aligned}
\texttt{mem}(x, \bullet) &= \texttt{false} \\
\texttt{mem}(x, \langle A, b, C \rangle) &= \begin{cases} \texttt{mem}(x, A) & \text{si } x < b \\ \texttt{true} & \text{si } x = b \\ \texttt{mem}(x, C) & \text{si } x > b \end{cases} \\
\texttt{mem}(x, \langle A, b, C, d, E \rangle) &= \begin{cases} \texttt{true} & \text{si } x = b \text{ ou } x = d \\ \texttt{mem}(x, A) & \text{si } x < b \\ \texttt{mem}(x, C) & \text{si } x > b \text{ et } x < d \\ \texttt{mem}(x, E) & \text{si } x > d \end{cases}
\end{aligned}
$$

Insertion dans un arbre 2-3 : version impérative



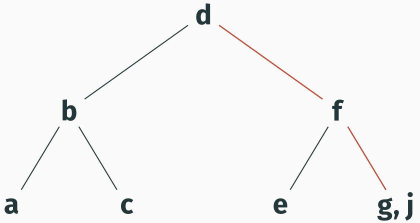
1. Rechercher l'élément à insérer (ici, **h**) dans l'arbre.

Insertion dans un arbre 2-3 : version impérative



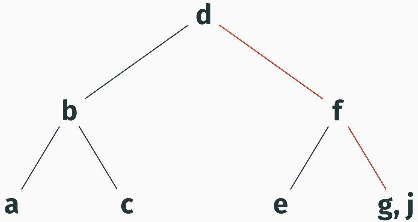
1. Rechercher l'élément à insérer (ici, **h**) dans l'arbre.

Insertion dans un arbre 2-3 : version impérative



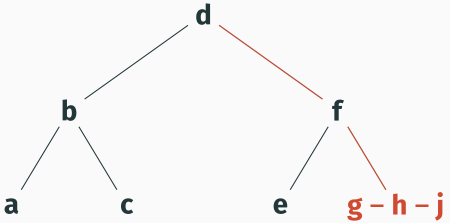
1. Rechercher l'élément à insérer (ici, **h**) dans l'arbre.
2. Si on termine sur un nœud 2, en faire un nœud 3 :
   $\langle \bullet, \mathbf{g}, \bullet \rangle \rightarrow \langle \bullet, \mathbf{g}, \bullet, \mathbf{h}, \bullet \rangle$ ou $\langle \bullet, \mathbf{j}, \bullet \rangle \rightarrow \langle \bullet, \mathbf{h}, \bullet, \mathbf{j}, \bullet \rangle$.

Insertion dans un arbre 2-3 : version impérative



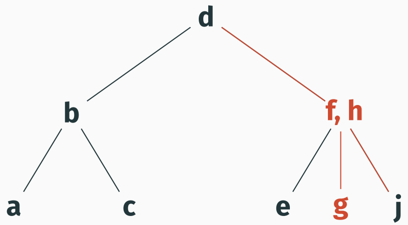
1. Rechercher l'élément à insérer (ici, **h**) dans l'arbre.
2. Si on termine sur un nœud 2, en faire un nœud 3 :
   $\langle \bullet, \mathbf{g}, \bullet \rangle \to \langle \bullet, \mathbf{g}, \bullet, \mathbf{h}, \bullet \rangle$ ou $\langle \bullet, \mathbf{j}, \bullet \rangle \to \langle \bullet, \mathbf{h}, \bullet, \mathbf{j}, \bullet \rangle$.
3. Si on termine sur un nœud 3 $\langle \bullet, \mathbf{g}, \bullet, \mathbf{j}, \bullet \rangle$, le faire «éclater» en deux nœuds 2 reliés $\langle \bullet, \mathbf{g}, \bullet \rangle - \mathbf{h} - \langle \bullet, \mathbf{j}, \bullet \rangle$ et insérer le tout dans le nœud 2 au-dessus.

Insertion dans un arbre 2-3 : version impérative



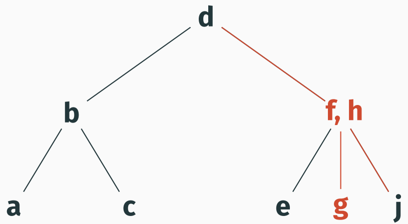
1. Rechercher l'élément à insérer (ici, **h**) dans l'arbre.
2. Si on termine sur un nœud 2, en faire un nœud 3 :
   $\langle \bullet, \mathbf{g}, \bullet \rangle \rightarrow \langle \bullet, \mathbf{g}, \bullet, \mathbf{h}, \bullet \rangle$ ou $\langle \bullet, \mathbf{j}, \bullet \rangle \rightarrow \langle \bullet, \mathbf{h}, \bullet, \mathbf{j}, \bullet \rangle$.
3. Si on termine sur un nœud 3 $\langle \bullet, \mathbf{g}, \bullet, \mathbf{j}, \bullet \rangle$, le faire ≪éclater≫ en deux nœuds 2 reliés $\langle \bullet, \mathbf{g}, \bullet \rangle - \mathbf{h} - \langle \bullet, \mathbf{j}, \bullet \rangle$ et insérer le tout dans le nœud 2 au-dessus.

Insertion dans un arbre 2-3 : version impérative



1. Rechercher l'élément à insérer (ici, **h**) dans l'arbre.
2. Si on termine sur un nœud 2, en faire un nœud 3 :
   $\langle \bullet, \mathbf{g}, \bullet \rangle \to \langle \bullet, \mathbf{g}, \bullet, \mathbf{h}, \bullet \rangle$ ou $\langle \bullet, \mathbf{j}, \bullet \rangle \to \langle \bullet, \mathbf{h}, \bullet, \mathbf{j}, \bullet \rangle$.
3. Si on termine sur un nœud 3 $\langle \bullet, \mathbf{g}, \bullet, \mathbf{j}, \bullet \rangle$, le faire «éclater» en deux nœuds 2 reliés $\langle \bullet, \mathbf{g}, \bullet \rangle - \mathbf{h} - \langle \bullet, \mathbf{j}, \bullet \rangle$ et insérer le tout dans le nœud 2 au-dessus.

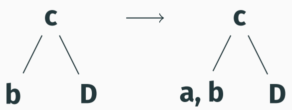
Insertion dans un arbre 2-3 : version impérative



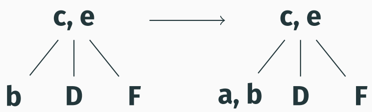
1. Rechercher l'élément à insérer (ici, **h**) dans l'arbre.
2. Si on termine sur un nœud 2, en faire un nœud 3 : $\langle \bullet, \mathbf{g}, \bullet \rangle \rightarrow \langle \bullet, \mathbf{g}, \bullet, \mathbf{h}, \bullet \rangle$ ou $\langle \bullet, \mathbf{j}, \bullet \rangle \rightarrow \langle \bullet, \mathbf{h}, \bullet, \mathbf{j}, \bullet \rangle$.
3. Si on termine sur un nœud 3 $\langle \bullet, \mathbf{g}, \bullet, \mathbf{j}, \bullet \rangle$, le faire «éclater» en deux nœuds 2 reliés $\langle \bullet, \mathbf{g}, \bullet \rangle - \mathbf{h} - \langle \bullet, \mathbf{j}, \bullet \rangle$ et insérer le tout dans le nœud 2 au-dessus.
4. Si le nœud au dessus est un nœud 3, le faire éclater à son tour et itérer.

Tous les cas d'insertion à gauche dans un arbre 2-3 de hauteur 2

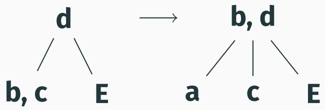
Chemin 2-2 devient 2-3 :



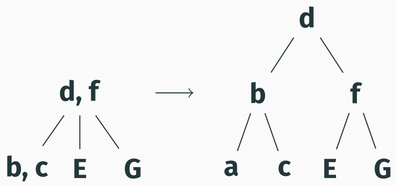
Chemin 3-2 devient 3-3 :



Chemin 2-3 devient 3-2 :



Chemin 3-3 devient 2-2-2 :



Note : c'est similaire à la propagation de la retenue lors de l'incrément d'un nombre en base 2.

Une implémentation purement fonctionnelle et finement typée

Utilise les types algébriques généralisés (GADT) de Haskell et d'OCaml pour garantir l'invariant sur les hauteurs des arbres.

```
type zero = Zero
type 'a succ = Succ of 'a

type _ tree =
  | Leaf : zero tree
  | Two : 'h tree * elt * 'h tree -> 'h succ tree
  | Three : 'h tree * elt * 'h tree * elt * 'h tree
                                      -> 'h succ tree
```

Le paramètre `'h` du type `'h tree` est la hauteur de l'arbre, encodée dans les types à la manière des entiers de Peano : `zero` (= 0), `zero succ` (= 1), `zero succ succ` (= 2), etc.

Une implémentation purement fonctionnelle et finement typée

Utilise les types algébriques généralisés (GADT) de Haskell et d'OCaml pour garantir l'invariant sur les hauteurs des arbres.

```
type zero = Zero
type 'a succ = Succ of 'a

type _ tree =
  | Leaf : zero tree
  | Two : 'h tree * elt * 'h tree -> 'h succ tree
  | Three : 'h tree * elt * 'h tree * elt * 'h tree
                                            -> 'h succ tree
```

Le type des constructeurs `Leaf`, `Two`, `Three` garantit les invariants sur les hauteurs.

Une implémentation purement fonctionnelle et finement typée

Utilise les types algébriques généralisés (GADT) de Haskell et d'OCaml pour garantir l'invariant sur les hauteurs des arbres.

```
type zero = Zero
type 'a succ = Succ of 'a

type _ tree =
  | Leaf : zero tree
  | Two : 'h tree * elt * 'h tree -> 'h succ tree
  | Three : 'h tree * elt * 'h tree * elt * 'h tree
                                    -> 'h succ tree

type set = Set : 'h tree -> set
```

Un ensemble fini (type `set`) est un `'h tree` pour un certain `'h` (quantification existentielle).

Typage de l’insertion

Le résultat de l’insertion n’est pas un simple `tree` mais un `etree` : soit `Ok` si c’est un arbre 2-3 bien formé, soit `Split` si c’est le résultat de l’éclatement d’un nœud.

```
type _ etree =
  | Ok: 'h tree -> 'h etree
  | Split: 'h tree * elt * 'h tree -> 'h etree
```

La différence entre `Split` et `Two` est que `Two` est un arbre au niveau $h + 1$, alors que `Split` est traité comme étant au niveau h.

Le code de l’insertion

```
let rec add_t : type h. elt -> h tree -> h etree =
  fun x t ->
  match t with
  | Leaf -> Split(Leaf, x, Leaf)
  | Two(a, b, c) ->
      if x = b then Ok t else
      if x < b then two1(add_t x a, b, c)
      else two2(a, b, add_t x c)
  | Three(a, b, c, d, e) ->
      if x = b || x = d then Ok t else
      if x < b then three1(add_t x a, b, c, d, e)
      else if x < d then three2(a, b, add_t x c, d, e)
      else three3(a, b, c, d, add_t x e)
```

Des *smart constructors* pour absorber les éclatements

```
  two1: 'h etree * elt * 'h tree -> 'h succ etree
  two2: 'h tree * elt * 'h etree -> 'h succ etree
three1: 'h etree * elt * 'h tree * elt * 'h tree -> 'h succ etree
three2: 'h tree * elt * 'h etree * elt * 'h tree -> 'h succ etree
three3: 'h tree * elt * 'h tree * elt * 'h etree -> 'h succ etree
```

Définitions quasi-mécaniques étant donnés les types et l'invariant ABR. Par exemple :

```
let two1 (ll, x, r) =
    match ll with
    | Ok l -> Ok (Two(l, x, r))
    | Split(a, b, c) -> Ok (Three(a, b, c, x, r))
let three1 (ll, x, m, y, r) ->
    match ll with
    | Ok l -> Ok (Three(l, x, m, y, r))
    | Split(a, b, c) -> Split(Two(a, b, c), x, Two(m, y, r))
```

Le code de l'insertion

Au niveau des ensembles d'éléments :

```
let add : elt -> set -> set =
  fun x (Set t) ->
    match add_t x t with
    | Ok t' -> Set t'
    | Split(a, b, c) -> Set (Two(a, b, c))
```

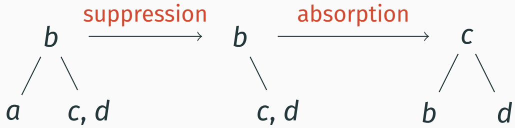
Le 2e cas augmente la hauteur de 1, mais cela est caché par le constructeur `Set` et sa quantification existentielle sur la hauteur.

Suppression dans un arbre 2-3

De même que l'insertion peut faire ≪exploser≫ des nœuds 3, la suppression peut faire ≪imploser≫ des nœuds 2, diminuant leur hauteur de 1.

$$\langle \bullet, a, \bullet, b, \bullet \rangle \to \langle \bullet, b, \bullet \rangle \; ✔ \qquad \langle \bullet, a, \bullet \rangle \to \bullet \; ✘$$

Il faut absorber cette diminution de hauteur plus haut :



Suppression dans un arbre 2-3

Le résultat de la suppression n'est pas un simple `tree` mais un `etree` :

```
type _ etree =
  | Ok: 'h tree -> 'h etree
  | Short: 'h tree -> 'h succ etree
```

`Short t` est l'arbre `t` avec un déficit de hauteur de 1, déficit qui doit être absorbé à l'aide de *smart constructors* :

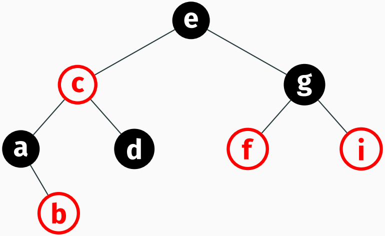
```
  two1: 'h etree * elt * 'h tree -> 'h succ etree
  two2: 'h tree * elt * 'h etree -> 'h succ etree
three1: 'h etree * elt * 'h tree * elt * 'h tree -> 'h succ etree
three2: 'h tree * elt * 'h etree * elt * 'h tree -> 'h succ etree
three3: 'h tree * elt * 'h tree * elt * 'h etree -> 'h succ etree
```

La suppression suit le même canevas que l'insertion. (Exercice !)

Arbres rouge-noir

Les arbres rouge-noir

Arbres binaires de recherche où chaque nœud a une couleur : rouge ou noir.



Deux invariants sur les couleurs :

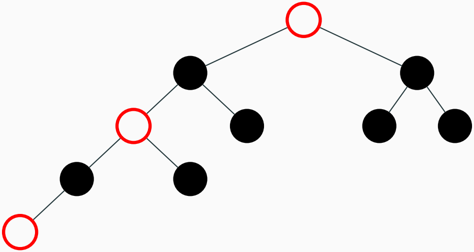
1. Les enfants d'un nœud rouge ne sont pas rouges.
2. Tous les chemins de la racine à une feuille contiennent le même nombre de nœuds noirs.
   Ce nombre est appelé ≪hauteur noire de l'arbre≫.

Équilibrage des arbres rouge-noir

1. Les enfants d'un nœud rouge ne sont pas rouges.
2. Tous les chemins de la racine à une feuille contiennent le même nombre de nœuds noirs.

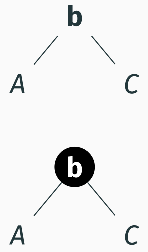
Si h est la hauteur noire de l'arbre, tous les chemins de la racine à une feuille sont de longueur h à $2h+1$. Cela garantit que l'arbre est de taille au moins 2^h, et donc équilibré.

On le montre en considérant des arbres à déséquilibre maximal :

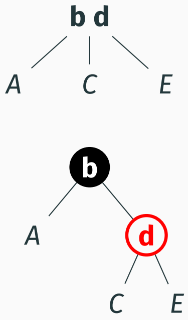


Les arbres rouge-noir ont été introduits par L. J. Guibas et R. Sedgewick (1978) comme une représentation simplifiée des arbres 2-3-4, c.à.d. des *B-trees* de degré 4, où chaque nœud porte entre 2 et 4 sous-arbres (et entre 1 et 3 clés).

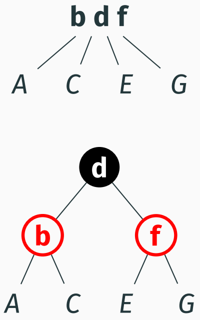
Noeud 2 :



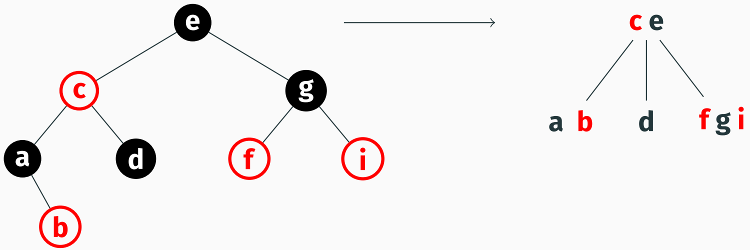
Noeud 3 :



Noeud 4 :



Symétriquement, on retrouve l'arbre 2-3-4 à partir d'un arbre rouge-noir en mettant «au même niveau» les nœuds rouges et leur parent noir :



Présentation algébrique

Couleur : $k ::= \mathbf{R} \mid \mathbf{B}$ rouge, noir

Arbre rouge-noir : $A, C ::= \bullet \mid k\langle A, b, C\rangle$

Recherche dichotomique standard, ignorant les couleurs :

$$\mathtt{mem}(x, \bullet) = \mathtt{false}$$

$$\mathtt{mem}(x, k\langle A, b, C\rangle) = \begin{cases} \mathtt{mem}(x, A) & \text{si } x < b \\ \mathtt{true} & \text{si } x = b \\ \mathtt{mem}(x, C) & \text{si } x > b \end{cases}$$

Effectue les mêmes comparaisons que la recherche di-/tri-/quadri-chotomique dans l'arbre 2-3-4 correspondant.

Insertion dans un arbre rouge-noir

Même principe que pour les arbres AVL :

- On recherche l'élément x à insérer.
- Si on atteint une feuille, on la remplace par $\mathbf{R}\langle \bullet, x, \bullet \rangle$.
- On rétablit les invariants en remontant (fonction `bal`).
- On recolorie la racine du résultat en noir.

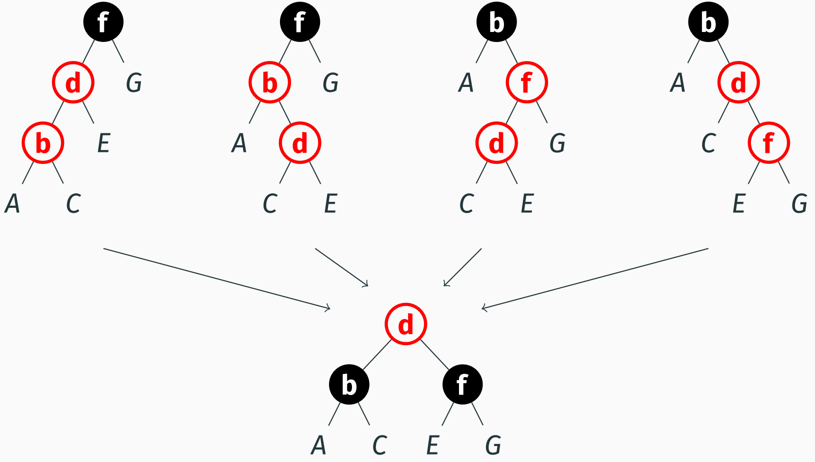
$$\mathtt{add}(x, \bullet) = \mathbf{R}\langle \bullet, x, \bullet \rangle$$

$$\mathtt{add}(x, k\langle A, b, C\rangle) = \begin{cases} \mathtt{bal}(k, \mathtt{add}(x, A), b, C) & \text{si } x < b \\ k\langle A, x, C\rangle & \text{si } x = b \\ \mathtt{bal}(k, A, b, \mathtt{add}(x, C)) & \text{si } x > b \end{cases}$$

Rétablir les invariants par rotations et recoloriage

Dans les livres d'algorithmique : 8 cas ou plus.

C. Okasaki (1999) : 4 cas suffisent.



Rééquilibrage en notation algébrique

$$
\begin{aligned}
\texttt{bal}(\mathbf{B}, \mathbf{R}\langle \mathbf{R}\langle A, b, C\rangle, d, E\rangle, f, G) &= \mathbf{R}\langle \mathbf{B}\langle A, b, C\rangle, d, \mathbf{B}\langle E, f, G\rangle\rangle \\
\texttt{bal}(\mathbf{B}, \mathbf{R}\langle A, b, \mathbf{R}\langle C, d, E\rangle\rangle, f, G) &= \quad " \quad " \quad " \\
\texttt{bal}(\mathbf{B}, A, b, \mathbf{R}\langle \mathbf{R}\langle C, d, E\rangle, f, G\rangle) &= \quad " \quad " \quad " \\
\texttt{bal}(\mathbf{B}, A, b, \mathbf{R}\langle C, d, \mathbf{R}\langle E, f, G\rangle\rangle) &= \quad " \quad " \quad " \\
\texttt{bal}(c, A, b, C) &= c\langle A, b, C\rangle \qquad \text{dans les autres cas}
\end{aligned}
$$

Suppression dans un arbre rouge-noir

L'algorithme général est le même que pour les AVL :

- Localiser le sous-arbre $k\langle A, x, C\rangle$
  qui porte l'élément x à enlever.
- Le remplacer par $k\langle A, \mathtt{min}(C), \mathtt{delmin}(C)\rangle$.
- Rééquilibrer (rétablir les invariants) en remontant.

Cependant, le rééquilibrage est beaucoup plus compliqué que pour l'insertion (une bonne douzaine de cas à considérer).

Expliquer la suppression avec des ≪doubles noirs≫

(K. Germane et M. Might, *Deletion : the curse of the red-black tree*, JFP 24(4), 2014.)

Les résultats intermédiaires de la suppression peuvent être non seulement des arbres rouge-noir, mais aussi

- une feuille ≪double≫ ●● qui compte comme 1 nœud noir ;
- un nœud ≪double noir≫ $\mathbf{BB}\langle A, b, C\rangle$ (dessiné en blanc !) qui compte comme 2 nœuds noirs.

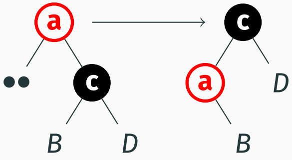
Pendant le rééquilibrage, ces doubles nœuds se propagent vers le haut (en préservant les invariants rouge-noir) et finissent par être absorbés.

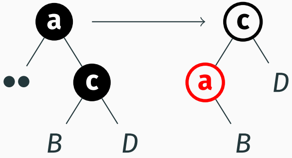
Suppression du plus petit élément

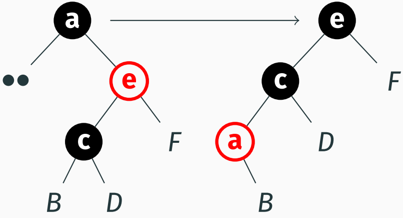
Cas de base : (avec préservation de la hauteur noire)

$$\mathbf{R}\langle \bullet, x, \bullet \rangle \longrightarrow \bullet \qquad \mathbf{B}\langle \bullet, x, \bullet \rangle \longrightarrow \bullet\bullet$$

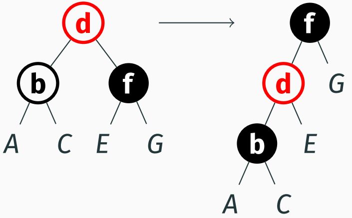
Propagation de ●● vers le haut : (plus : cas symétriques g-d)

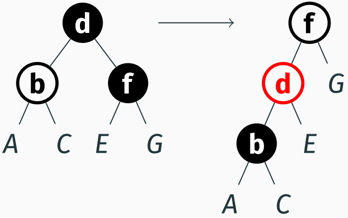


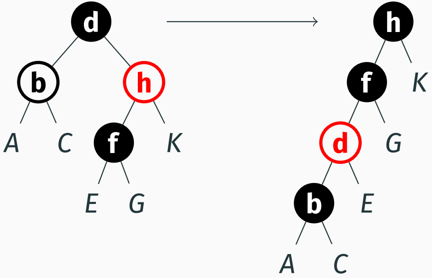




Propagation des nœuds doubles noirs vers le haut







Arbres préfixes

Arbres préfixes

(Anglais : *tries*, de *information reTRIEval*.)

Une représentation des ensembles finis ou des dictionnaires dont les clés sont des mots, c.à.d. des listes de symboles.

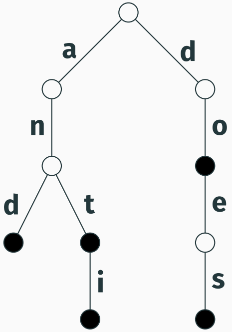
Par exemple :

- chaînes de caractères
  (symboles = caractères, ou octets, ou bits)
- nombres entiers
  (symboles = bits ou groupes de k bits).

Chaque nœud de l'arbre porte entre 0 et K sous-arbres, chacun étiqueté par un symbole. (K = nombre de symboles)

Exemple d’arbre préfixe

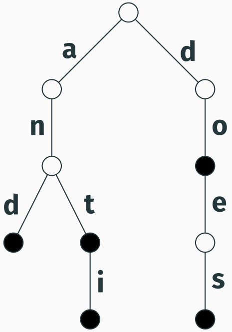
L’ensemble des mots **and, ant, anti, do, does**.



Les nœuds noirs (•) marquent la fin d’un mot.

Exemple d’arbre préfixe

L’ensemble des mots **and, ant, anti, do, does**.

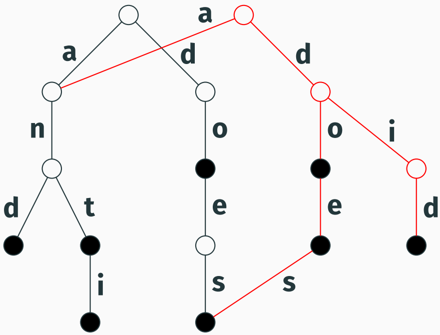


Recherche : suivre les arcs qui épellent les lettres du mot.

Le mot est présent ssi on termine sur un nœud noir.

Exemple d'arbre préfixe

L'ensemble des mots **and, ant, anti, do, does**.



Insertion persistante : copier et compléter la branche qui épelle les lettres du mot. Mettre en noir le nœud final.
(Exemple : insertion de **doe** et de **did**.)

Implémentation fonctionnelle d’un arbre préfixe

Ensembles et dictionnaires :

```
type set = Node of bool * (char * set) list
type 'a map = Node of 'a option * (char * 'a map) list
```

`char` est le type des symboles. Les sous-arbres sont stockés dans des listes d’association `(char * ...) list`.

Cas particulier où les symboles sont des bits :

```
type set = Empty | Node of set * bool * set
type 'a map = Empty | Node of 'a map * 'a option * 'a map
```

Toujours deux sous-arbres, un pour le bit `0`, l’autre pour le bit `1`.

Recherche et insertion (cas binaire)

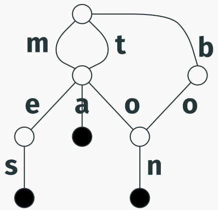
```
let rec mem s t =
  match t, s with
  | Empty, _ -> false
  | Node(_, acc, _), [] -> acc
  | Node(l, _ , r), b :: s' -> mem s' (if b then r else l)

let rec add s t =
  match t, s with
  | Empty -> add s (Node(Empty, false, Empty))
  | Node(l, acc, r), [] -> Node(l, true, r)
  | Node(l, acc, r), b :: s' ->
      if b then Node(l, acc, add s' r)
           else Node(add s' l, acc, r)
```

Temps proportionnel à $|s|$. Si les comparaisons de clés sont coûteuses, c'est plus efficace qu'un A.B.R. (temps $\mathcal{O}(|s| \log n)$).

Partage de sous-arbres correspondant à des suffixes communs

Dans un arbre préfixe immuable, des sous-arbres correspondant à des suffixes communs à plusieurs mots peuvent être partagés, obtenant ainsi un graphe acyclique de mots (DAWG), aussi appelé automate fini déterministe acyclique (DAFSA).

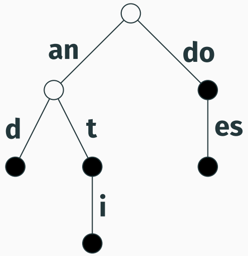
Exemple : **bon**, **mon**, **ma**, **mes**, **ton**, **ta**, **tes**.



Construction : par *hash consing* ou par minimisation d'automate.

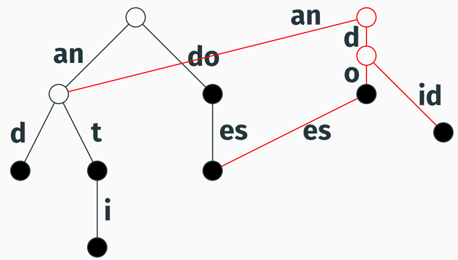
Arbre préfixe compressé

On peut éviter les nœuds triviaux en étiquetant les sous-arbres non par un symbole mais par un sous-mot (= liste de symboles).



Arbre préfixe compressé

On peut éviter les nœuds triviaux en étiquetant les sous-arbres non par un symbole mais par un sous-mot (= liste de symboles).



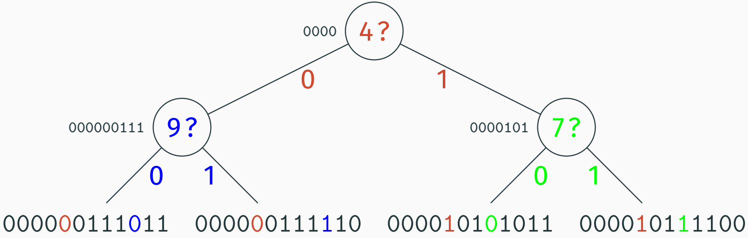
L'insertion peut nécessiter de couper un arc en arc-nœud-arc, afin d'ancrer le nouveau mot (**did** dans l'exemple ci-dessus).

On coupe au plus grand préfixe commun entre le sous-mot sur l'arc (**do**) et le nouveau mot (**did**).

Arbres PATRICIA

(Donald R. Morrison, *PATRICIA – Practical Algorithm to Retrieve Information Coded in Alphanumeric*, 1968.)

Arbres préfixes pour des suites de bits, optimisés pour le cas où l'espace des clés est peu dense.



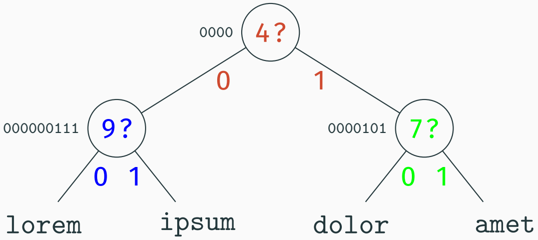
Chaque nœud porte le numéro d'un bit à tester (+ le préfixe).

Les clés sont stockées aux feuilles.

Arbres de hachage (*hash trees*)

(Phil Bagwell, *Ideal Hash Trees*, 2000.)

Pour représenter des ensembles $\{k_1, \ldots, k_n\}$ de clés de type arbitraire, on peut transformer ces clés en entiers à l'aide d'une fonction de hachage H, et mettre $k_1, \ldots, k_n$ aux feuilles d'un arbre PATRICIA, en positions $H(k_1), \ldots, H(k_n)$.



Avec $H(\texttt{"lorem"}) = 00000011101 1$, $H(\texttt{"dolor"}) = 00001010 1011$, etc.

Gestion des collisions

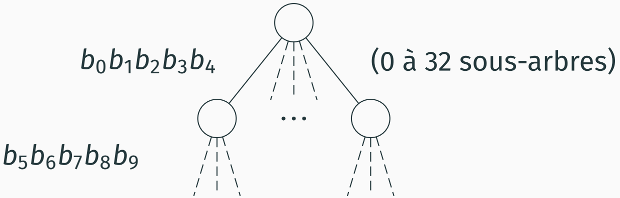
La fonction H étant à valeurs finies (typiquement 32 ou 64 bits), il existe des collisions : des clés différentes $k \neq k'$ telles que $H(k) = H(k')$.

Solution classique : mettre aux feuilles de l'arbre des listes de clés $k, k', k'', \dots$ qui ont la même valeur de hachage.

Solution proposée par Bagwell : $\ll$rallonger$\gg$ la valeur de hachage en utilisant plusieurs fonctions $H_0, H_1, \dots$ statistiquement indépendantes. Autrement dit, on calcule à la demande une très longue valeur de hachage $H_0(k).H_1(k) \dots H_n(k) \dots$ et on utilise l'arbre PATRICIA pour distinguer entre toutes ces valeurs.

Arbres de hachage et tableaux (HAMT, *hash array mapped tree*)

Au lieu d'un arbre PATRICIA, les HAMT utilisent un arbre préfixe simple mais avec un degré de branchement assez élevé, p.ex. $32 = 2^5$, d'où un traitement de $H(k)$ par paquets de 5 bits.



(Avec $H(k) = b_0b_1b_2\ldots$)

Arbres de hachage et tableaux (HAMT, *hash array mapped tree*)

Chaque nœud porte une fonction partielle $f : [0, 31] \rightarrow$ arbre implémentée de manière efficace en temps et en espace :

- Un vecteur de 32 bits B (= un entier machine).
  Pour chaque $i \in [0, 31]$, $B(i)$ dit si $f(i)$ est défini ou non.
- Un tableau T de n sous-arbres,
  où n est le nombre de i où $f(i)$ est défini.

Accéder au sous-arbre étiqueté i se fait très efficacement :

$$f(i) = \begin{cases} \text{indéfini} & \text{si } B \ \& \ (1 \ll i) = 0 \\ T[\texttt{popcnt}(B \ \& \ ((1 \ll i) - 1))] & \text{sinon} \end{cases}$$

$\texttt{popcnt}(n)$ est le poids de Hamming de n (nombre de bits à 1).

Point d’étape

Méthodologie

Sur des exemples à base d'arbres équilibrés, nous avons vu deux manières complémentaires pour développer des structures de données persistantes :

1. Partir de structures impératives et appliquer la technique de copie de branche pour les rendre persistantes.
2. Partir de définitions algébriques de la structure et de ses opérations, et en dériver des implémentations purement fonctionnelles.

Les deux approches débouchent sur les mêmes algorithmes.

(2) facilite la spécification et la vérification fonctionnelles.
(1) est traditionnellement utilisée pour l'analyse de complexité.
($\Rightarrow$ séminaire de T. Nipkow le 23 mars)

Résultats

Des structures persistantes avec implémentations purement fonctionnelles pour

- les dictionnaires et les ensembles finis
- les files de priorité
- les séquences indexées.

Toutes les opérations élémentaires en temps $\mathcal{O}(\log n)$.

Question récurrente dans la suite du cours :
comment descendre en dessous de $\mathcal{O}(\log n)$?

Bibliographie

Bibliographie

Présentation fonctionnelle moderne, avec démonstrations mécanisées de correction et de complexité :

- Tobias Nipkow et coauteurs, *Functional Algorithms, Verified !*, `https://functional-algorithms-verified.org/`

Présentations impératives classiques :

- D. E. Knuth, *The Art of Computer Programming*, volume 3, chapitre 6 *Searching*.
- R. Sedgewick et K. Wayne, *Algorithms – 4th edition*, sections 3.2 et 3.3.