Les arbres binaires de recherche équilibrés

Stéphane Glondu

Table des matières

1 Arbres binaires de recherche **2**
1.1 Rappels . . . 2
1.2 Rotations . . . 2

2 Arbres AVL **2**
2.1 Propriétés . . . 2
2.2 Opérations dynamiques . . . 3
2.2.1 Rééquilibrage . . . 3
2.2.2 Insertion et suppression . . . 4

3 Arbres rouge-noir **4**
3.1 Propriétés . . . 4
3.2 Opérations dynamiques . . . 5
3.2.1 Insertion . . . 5
3.2.2 Suppression . . . 5
3.2.3 Conclusion . . . 5

4 Performances **5**
4.1 Arbres construits aléatoirement . . . 6
4.2 Cas les pires . . . 6
4.3 Bilan . . . 6

Table des figures

1 Exemples d'arbres binaires de recherche . . . 7
2 Illustration des rotations . . . 7
3 Exemples d'arbres AVL . . . 8
4 Rééquilibrage d'un arbre AVL (cas 1) . . . 8
5 Rééquilibrage d'un arbre AVL (cas 2) . . . 8
6 Exemples d'arbres rouge-noir . . . 9
7 Correction d'un arbre rouge-noir après une insertion (cas 1) . . . 9
8 Correction d'un arbre rouge-noir après une insertion (cas 2) . . . 9
9 Correction d'un arbre rouge-noir après une suppression (cas 1) . . . 10
10 Correction d'un arbre rouge-noir après une suppression (cas 2) . . . 10
11 Correction d'un arbre rouge-noir après une suppression (cas 3) . . . 10

Introduction

Les arbres de recherche permettent de gérer des ensembles dynamiques ordonnés. Dans certains cas, une implémentation naïve produit des arbres déséquilibrés totalement inefficaces. Ce sont ces cas que nous nous proposons de gérer en présentant des algorithmes qui permettent la recherche, l'insertion et la suppression en temps logarithmique dans le pire des cas. Nous nous intéresserons tout d'abord aux *arbres AVL*, puis aux *arbres rouge-noir*. Enfin, nous présenterons une comparaison des performances de ces différents algorithmes.

1 Arbres binaires de recherche

1.1 Rappels

Définition 1 (Arbre binaire). Soit E un ensemble. Un *arbre binaire* est :
- soit l'arbre vide $\varnothing$;
- soit un *nœud* $A(g, r, d)$, où g et d sont des arbres, et désignent respectivement les fils gauche et droit, et $r \in E$ les données stockées dans le nœud.

Un nœud est une *feuille* si ses deux fils sont vides, sinon c'est un *nœud interne*.

Nous associerons dorénavant à chaque nœud x un nombre, appelé *clé*. Pour plus de clarté, nous supposerons dorénavant que toutes les clés d'un arbre sont distinctes, identifiant ainsi le contenu d'un nœud à sa clé.

Définition 2 (Arbre binaire de recherche). Un arbre binaire est *de recherche* lorsque, si x est un nœud de l'arbre, et y un nœud du sous-arbre gauche (resp. droit) de x, on a $y < x$ (resp. $x < y$).

La propriété d'arbre binaire de recherche permet de trouver, d'insérer ou de supprimer facilement un nœud grâce à sa clé en $O(h)$ (figure 1). On a le résultat suivant pour un arbre binaire quelconque, et en particulier pour un arbre binaire de recherche :

Proposition 1 (Hauteur d'un arbre binaire). *Soit un arbre binaire non vide de hauteur h et possédant n nœuds. On a :*

$$\lfloor \log_2 n \rfloor \leqslant h \leqslant n - 1.$$

Ces bornes sont optimales.

1.2 Rotations

Ces opérations sont illustrées par la figure 2.

Proposition 2 (Propriété essentielle des rotations). *Les rotations préservent la propriété d'arbre binaire de recherche.*

2 Arbres AVL

2.1 Propriétés

Définition 3 (Arbre AVL). Un arbre binaire de recherche est un *arbre AVL* si, pour n'importe lequel de ses nœuds, la différence de hauteur entre ses deux fils diffère d'au plus un.

La figure 3 donne deux exemples d'arbres AVL. Pour l'implémentation, on supposera que r contiendra la hauteur, de sorte que la hauteur d'un sous-arbre quelconque est déterminable en temps constant. On a le résultat suivant :

Proposition 3 (Hauteur d'un arbre AVL). *Soit un arbre AVL de hauteur h et possédant n nœuds. On a :*

$$h < \frac{3}{2}\log_2(n+1).$$

Démonstration. Notons u_h le nombre minimal de nœuds d'un arbre de hauteur h. Il est clair que $u_0 = 1$ et $u_1 = 2$. Soit $A(g, r, d)$ de hauteur $h+2$. Alors on a, par exemple, $h(g) = h+1$ et $h(d) \geqslant h$. D'où $u_{h+2} \geqslant 1 + u_{h+1} + u_h$. Réciproquement, soient g un arbre de hauteur $h+1$ à u_{h+1} nœuds, d un arbre de hauteur h à u_h nœuds, et r quelconque. Alors l'arbre $A(g, r, d)$ possède $1 + u_{h+1} + u_h$ nœuds. D'où $u_{h+2} \leqslant 1 + u_{h+1} + u_h$. On en déduit la relation de récurrence :

$$u_{h+2} = 1 + u_{h+1} + u_h.$$

Sa résolution aboutit à l'expression :

$$u_h = \left(\frac{5+2\sqrt{5}}{5}\right)\left(\frac{1+\sqrt{5}}{2}\right)^h + \left(\frac{5-2\sqrt{5}}{5}\right)\left(\frac{1-\sqrt{5}}{2}\right)^h - 1.$$

On a de plus :

$$\frac{5-2\sqrt{5}}{5} \approx 0,11 \quad \text{et} \quad \frac{1-\sqrt{5}}{2} \approx -0,62.$$

On peut donc minorer le deuxième terme de u_h par -1. La relation $n \geqslant u_h$ implique donc successivement :

$$n+2 > \left(\frac{5+2\sqrt{5}}{5}\right)\left(\frac{1+\sqrt{5}}{2}\right)^h,$$

$$h < \log_{\frac{1+\sqrt{5}}{2}}(n+1) + \log_{\frac{1+\sqrt{5}}{2}}\left(\frac{\frac{n+2}{n+1}}{\frac{5+2\sqrt{5}}{5}}\right).$$

Le second terme est strictement négatif pour $n \geqslant 1$ car :

$$\frac{n+2}{n+1} < \frac{5+2\sqrt{5}}{5} \approx 1,89 \quad \text{et} \quad 1 < \frac{1+\sqrt{5}}{2} \approx 1,62.$$

On en déduit :

$$h < \log_{\frac{1+\sqrt{5}}{2}}(n+1) = \frac{\ln 2}{\ln\left(\frac{1+\sqrt{5}}{2}\right)}\log_2(n+1),$$

$$h < \frac{3}{2}\log_2(n+1).$$

Par convention, $h(\varnothing) = -1$, donc l'inégalité reste vraie pour $n = 0$. □

2.2 Opérations dynamiques

2.2.1 Rééquilibrage

Insérer ou supprimer un nœud d'un arbre AVL à l'aide d'une méthode naïve risque d'enfreindre la propriété d'arbre AVL. Pour la rétablir, on effectue des rotations.

Supposons que l'on ait inséré ou supprimé un élément e dans l'arbre a en utilisant la méthode naïve, obtenant ainsi un nouvel arbre a'. Nous nous placerons dans le cas où le nouvel arbre a' n'est pas un arbre AVL. Alors il existe un nœud $A(g, r, d)$ tel que $|h(g) - h(d)| = 2$. Prenons le nœud vérifiant cette propriété et ayant la petite hauteur. Ainsi, les sous-arbres g et d sont des arbres AVL. Supposons que $h(g) - h(d) = 2$ (l'autre cas se traite de manière symétrique). Deux cas se présentent, illustrés par les figures 4 et 5.

On définit ainsi une fonction s'exécutant en temps constant, qui rééquilibre un sous-arbre AVL juste après une insertion ou une suppression, déplaçant éventuellement ainsi le déséquilibre vers le haut.

2.2.2 Insertion et suppression

En utilisant une implémentation naïve, récursive, pour les fonctions d'insertion et de suppression, il suffit de faire transiter chaque arbre renvoyé par la fonction de rééquilibrage définie précédemment. Par conséquent, la complexité temporelle est $O(h) = O(\ln n)$ pour les deux fonctions.

3 Arbres rouge-noir

3.1 Propriétés

Définition 4 (Arbre rouge-noir). Un arbre binaire de recherche est un *arbre rouge-noir* s'il vérifie les propriétés suivantes :

1. chaque nœud est soit rouge, soit noir ;
2. la racine est noire ;
3. chaque sous-arbre vide est noir ;
4. si un nœud est rouge, alors ses deux enfants sont noirs ;
5. pour chaque nœud, tous les chemins reliant le nœud à une feuille contiennent le même nombre de nœuds noirs (ce nombre sera appelé *hauteur noire* et noté ω).

La figure 6 donne deux exemples d'arbres rouge-noir. Pour l'implémentation, r contiendra la couleur. On dispose d'une inégalité analogue à 3 :

Proposition 4 (Hauteur d'un arbre rouge-noir). *Soit un arbre rouge-noir de hauteur h et possédant n nœuds. On a :*

$$h \leqslant 2\log_2(n+1).$$

Démonstration. Montrons d'abord par récurrence sur h le lemme suivant : un sous-arbre a de hauteur h possède au moins $2^{\omega(a)} - 1$ nœuds. Pour $h = -1$ ou $h = 0$, c'est évident. Supposons que le lemme soit vérifié pour des sous-arbres de hauteur inférieure ou égale à h. Soit $a = A(g, r, d)$ un arbre rouge-noir de hauteur $h + 1$. Alors $\omega(g) = \omega(a)$ ou $\omega(g) = \omega(a) - 1$ selon la couleur de g. Dans tous les cas, $\omega(g) \geqslant \omega(a) - 1$. De même, $\omega(d) \geqslant \omega(a) - 1$. Par hypothèse de récurrence, on a alors :

$$\begin{aligned} n(a) &\geqslant \left(2^{\omega(a)-1} - 1\right) + \left(2^{\omega(a)-1} - 1\right) + 1, \\ &\geqslant 2^{\omega(a)} - 1. \end{aligned}$$

Le lemme est ainsi démontré au rang $h + 1$.

Pour terminer, soit a un arbre rouge-noir non vide de hauteur h et possédant n nœuds. Remarquons que la propriété 4 de la définition 4 implique $\omega(a) \geqslant h/2$. En appliquant le lemme, il vient : $n \geqslant 2^{h/2} - 1$, d'où le résultat. □

3.2 Opérations dynamiques

3.2.1 Insertion

Comme pour les arbres AVL, on utilise l'algorithme naïf pour insérer le nœud, que l'on colorie en rouge, puis on corrige l'arbre obtenu afin de rétablir les propriétés d'arbre rouge-noir qui auraient été violées.

Remarquons dans un premier temps que seules les propriétés 2 et 4 sont concernées. La violation de la propriété 2 seule ne pose pas de problème dans la mesure où le coloriage de la racine en noir la rétablit sans violer les autres propriétés. Analysons donc le cas où seule la propriété 4 est violée. Un nœud rouge possède donc un fils rouge x. Considérons le sous-arbre enraciné en le grand-père y de x. Supposons que x est dans le sous-arbre gauche de y (l'autre cas de traite de façon symétrique). Les figures 7 et 8 illustrent les deux possibilités.

3.2.2 Suppression

Comme pour l'insertion, on utilise la méthode naïve pour supprimer e, puis on rétablit les propriétés qui auraient été violées. Cela ne peut arriver que si e est noir, et seules les propriétés 2, 4 et 5 sont concernées.

Pour les rétablir, nous utilisons une version modifiée des arbres rouge-noir, les *arbres rectifiables*, où nous donnons la possibilité à un nœud d'être *doublement noir* de telle sorte que, si un arbre rectifiable vérifie la propriété 1, alors il est rouge-noir.

Si e possède deux fils non vides, supprimons le minimum du sous-arbre droit et mettons-le à la place de e, en lui attribuant la couleur de e.

Ainsi, on est ramené au cas où e possède au plus un fils non vide. Nous noterons ce fils x, s'il existe, sinon nous désignerons par x un fils quelconque de e. Après suppression de e, *assombrissons* x, *i.e.* rendons-le noir s'il est rouge, ou rendons-le doublement noir s'il est déjà noir. Nous obtenons ainsi un arbre rectifiable a. Nous utiliserons aussi l'opération d'*éclaircissement*, inverse de l'assombrissement.

Il suffit maintenant d'effectuer des opérations sur cet arbre rectifiable afin qu'il vérifie la propriété 1. C'est bien entendu le cas lorsque x est rouge ou noir. Plaçons-nous dans le cas où x est doublement noir.

Si x est la racine, alors éclaircir x fera de a un arbre rouge-noir. Sinon, x possède un père y, et un frère $z \neq \varnothing$. Nous supposerons désormais que x est le fils gauche de y. Les figures 9 à 11 illustrent les trois possibilités.

3.2.3 Conclusion

Comme pour les arbres AVL, on sait donc insérer ou supprimer un nœud dans un arbre rouge-noir en $O(h) = O(\ln n)$ opérations élémentaires.

4 Performances

Les algorithmes présentés ont été implémentés en Objective Caml.

4.1 Arbres construits aléatoirement

Nous construisons ici un arbre avec m clés aléatoires, mais les mêmes pour tous les programmes, obtenant ainsi un arbre à n nœuds ($n < m$). L'arbre est construit au bout d'une durée τ_0. Puis nous supprimons (resp. insérons) p nombres aléatoires, en notant la durée τ_1 (resp. τ_2). Enfin, nous supprimons tous les nœuds dans l'ordre croissant, en notant la durée τ_3.

Expérimentalement, les durées τ_i (tableau 1) sont du même ordre de grandeur quel que soit le programme et quelles que soit les entrées, et elles sont aussi généralement en accord avec les complexités théoriques.

4.2 Cas les pires

Il s'agit ici de tester un des « cas les pires » évoqués dans l'introduction : nous insérons n clés dans l'ordre croissant. Comme prévu théoriquement, les résultats (tableau 2) sont sans appel.

4.3 Bilan

Les arbres AVL et les arbres rouge-noir sont donc aussi performants que l'implémentation naïve avec des données aléatoires, mais nettement meilleurs dans les pires cas. Mais n'oublions pas que ces arbres stockent des données supplémentaires pour maintenir leur équilibre. Les plus performants semblent être les arbres AVL. Ils stockent un entier supplémentaire par nœud. Leur code est aussi plus simple. Néanmoins, les arbres rouge-noir n'ont besoin que d'un seul bit supplémentaire.

Conclusion et perspectives

Les arbres de recherche peuvent être utiles dans des domaines aussi variés que la compilation, la gestion de fichiers ou les bases de données. Nous n'avons présenté ici que des arbres binaires, mais il existe aussi d'autres variantes qui agissent par des modifications du degré des nœuds. Par ailleurs, les algorithmes présentés ici ne tiennent pas compte du support de données, mais il existe aussi des algorithmes optimisés pour des arbres stockés sur disque.

Illustrations

Arbres binaires de recherche

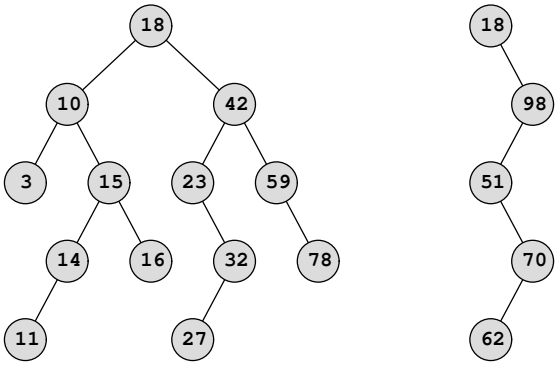

Fig. 1 – Exemples d'arbres binaires de recherche

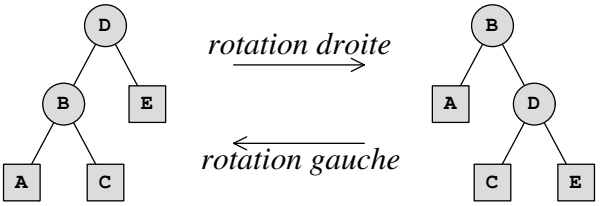

Fig. 2 – Illustration des rotations

Arbres AVL

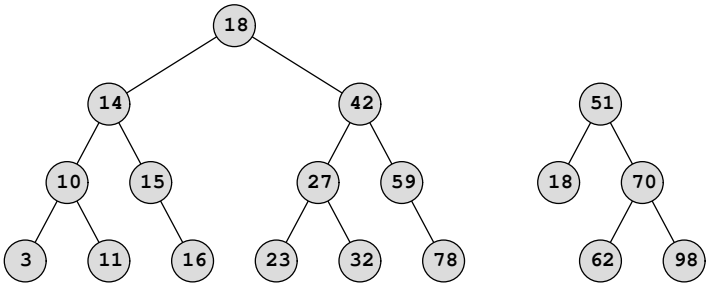

FIG. 3 – Exemples d'arbres AVL

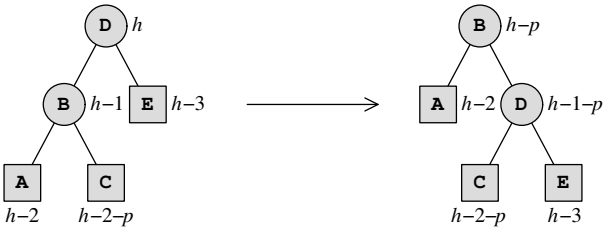

FIG. 4 – Rééquilibrage d'un arbre AVL (cas 1)

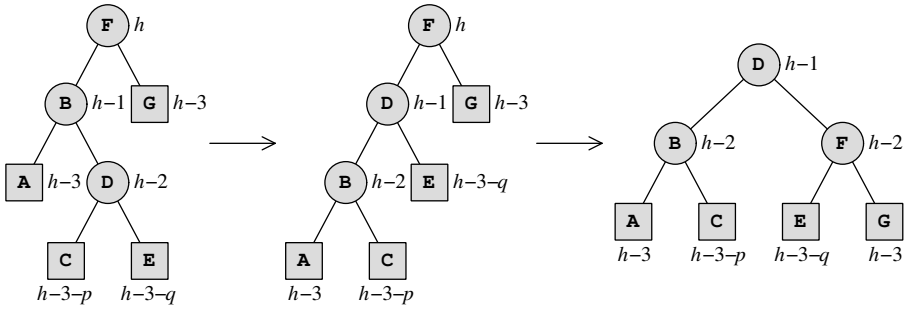

FIG. 5 – Rééquilibrage d'un arbre AVL (cas 2)

Arbres Rouge-Noir

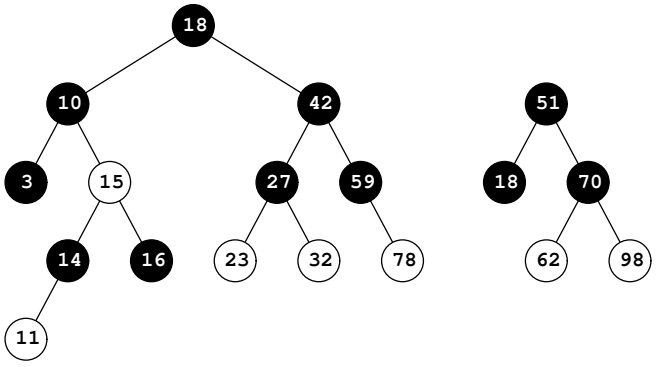

FIG. 6 – Exemples d'arbres rouge-noir

Insertion

Cas 1 : z est rouge Il existe deux sous-cas qui sont traités de la même manière, selon que x est un fils gauche ou droit. Le premier sous-cas est illustré par la figure 7. Un recoloriage rétablit les propriétés d'arbre rouge-noir, sauf éventuellement la deuxième, et aucune hauteur noire n'est modifiée. En revanche, il est possible que le père de y soit rouge, ou que y soit la racine. La correction devra donc se poursuivre au niveau supérieur.

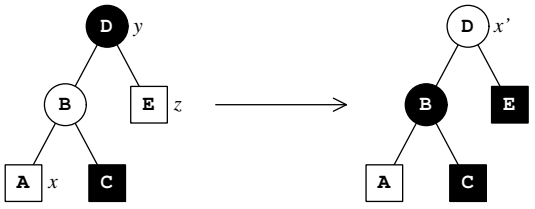

FIG. 7 – Correction d'un arbre rouge-noir après une insertion (cas 1)

Cas 2 : z est noir Ce cas est illustré par la figure 8. On se ramène d'abord au cas où x est un fils gauche par une rotation gauche (première flèche), puis on effectue une rotation droite et un recoloriage (seconde flèche). À l'issue de cette opération, le sous-arbre obtenu est un arbre rouge-noir, et aucune hauteur noire n'a pas été modifiée : la correction est donc terminée.

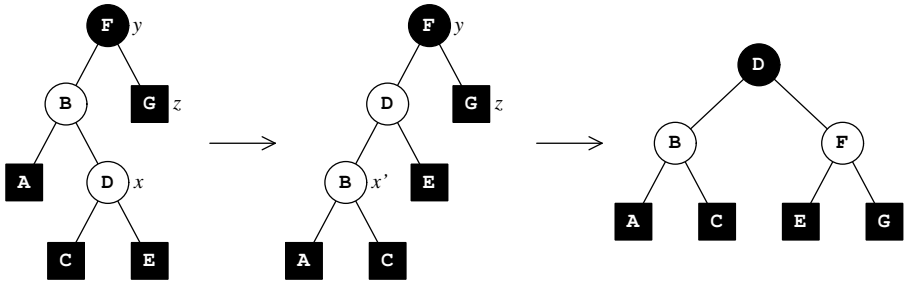

FIG. 8 – Correction d'un arbre rouge-noir après une insertion (cas 2)

Suppression

Cas 1 : z est rouge Une rotation gauche et un recoloriage (figure 9) permet de se ramener à l'un des trois autres cas.

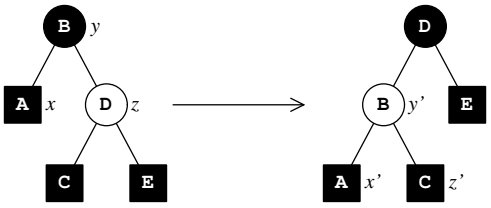

FIG. 9 – Correction d'un arbre rouge-noir après une suppression (cas 1)

Cas 2 : z est noir, ainsi que ses deux fils Un éclaircissement de x et z et un assombrissement de y (figure 10) déplacent le problème vers le haut. Remarquons que si y était rouge, alors la correction est terminée.

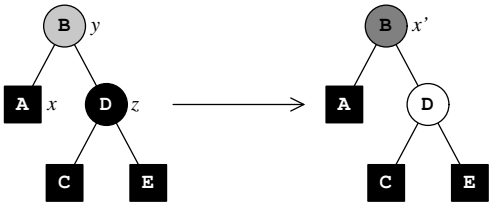

FIG. 10 – Correction d'un arbre rouge-noir après une suppression (cas 2)

Cas 3 : z est noir, et possède un fils rouge On se ramène d'abord au cas où le fils droit de z est rouge (première flèche de la figure 11). Une rotation gauche et un recoloriage (seconde flèche) terminent la correction.

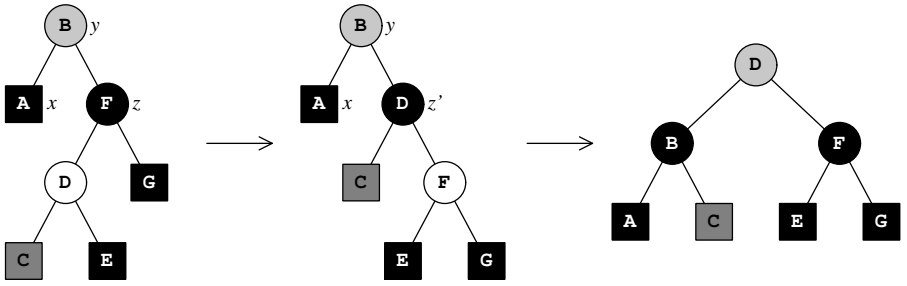

FIG. 11 – Correction d'un arbre rouge-noir après une suppression (cas 3)

Performances

Algorithme	Test	h	τ_0 (s)	τ_1 (s)	τ_2 (s)	τ_3 (s)
Naïf	1	41	19,73	0,08	0,04	0,04
AVL	1	18	19,71	0,08	0,05	0,08
Rouge-Noir	1	19	20,09	0,08	0,05	0,06
Naïf	2	45	47,02	0,20	0,13	0,07
AVL	2	19	46,03	0,20	0,14	0,17
Rouge-Noir	2	20	47,05	0,20	0,14	0,13
Naïf	3	42	109,81	0,44	0,36	0,15
AVL	3	20	107,30	0,47	0,26	0,38
Rouge-Noir	3	21	108,79	0,49	0,33	0,27

Test 1 : $m = 1\,000\,000$, $n = 64\,000$ et $p = 4\,000$
Test 2 : $m = 2\,000\,000$, $n = 128\,000$ et $p = 8\,000$
Test 3 : $m = 4\,000\,000$, $n = 256\,000$ et $p = 16\,000$

Tab. 1 – Arbres construits aléatoirement

Algorithme	Test	h	$\frac{h}{\log_2 n}$	τ_0 (s)	$\frac{\tau_0}{n \log_2 n}$ (μs)
Naïf	1	31 999	2 138	614	1 383
AVL	1	14	0,9	0,06	0,13
AVL	2	18	0,9	1,31	0,14
AVL	3	20	1,0	5,51	0,14
Rouge-Noir	1	26	1,7	0,08	0,17
Rouge-Noir	2	34	1,8	1,79	0,19
Rouge-Noir	3	38	1,8	9,76	0,24

Test 1 : $n = 32\,000$
Test 2 : $n = 512\,000$
Test 3 : $n = 2\,000\,000$

Tab. 2 – Tests d'un des cas les pires

Références

[1] Adel'son-Vel'skiĭ (G. M.) et Landis (E. M.). An algorithm for the organization of information. *Soviet Mathematics Doklady*, 3 : 1259–1263, 1962.

[2] Bayer (R.). Symmetric binary B-trees : Data structure and maintenance algorithms. *Acta Informatica*, 1 : 290–306, 1972.

[3] Cormen (Thomas H.), Leiserson (Charles E.), Rivest (Ronald L.) et Stein (Clifford). *Introduction à l'algorithmique*, chapitres 12–13. Dunod, 2002.

[4] Guibas (Leo J.) et Sedgewick (Robert). A dichromatic framework for balanced trees. *Proceedings of the 19th Annual Symposium on Foundations of Computer Science*, 8–21. IEEE Computer Society, 1978.

[5] Leroy (Xavier). *The Objective Caml system (release 3.07). Documentation and user's manual*. INRIA, 2003. Disponible sur le web (caml.inria.fr).

[6] Leroy (Xavier). *The Objective Caml system (release 3.07). The standard library*, module **Map** (fichiers **map.ml** et **map.mli**). INRIA, 2003. Disponible sur le web (caml.inria.fr).