

Les arbres 2-3

Laurent Chéno*

Luminy, 8-12 mai 2000

<http://lcheno.free.fr/arbres23/>

Résumé

Nous décrivons ici les arbres 2-3, structure (décrite pour la première fois en 1979) d'arbres de recherche qui maintiennent un équilibrage suffisant pour garantir par exemple un tri en $O(n \lg n)$. Nous présentons le code CAML correspondant aux fonctions de base sur une telle structure (recherche, insertion, suppression, et donc tri). Nous donnons également les éléments théoriques qui permettraient une généralisation aux arbres a - b sous la condition $b \geq 2a - 1 \geq 3$. Enfin, nous tentons une approche combinatoire de la structure présentée.

*lycée Louis-le-Grand, Paris; <mailto:laurent.cheno@inria.fr>

Table des matières

1	Les arbres 2-3	3
1.1	Arbres de recherche	3
1.2	Arbres bi- ou ter- naires	3
1.3	Équilibrage	4
1.4	Un petit point historique	4
1.5	Recherche d'une clé	4
1.6	Insertion d'une clé	4
1.7	Suppression d'une clé	7
2	Généralisation : les arbres a-b	9
2.1	Définitions	9
2.2	Insertion et suppression d'une clé	9
3	Implantation en Caml des arbres 2-3	10
3.1	Le typage des arbres	10
3.2	La recherche d'une clé	10
3.3	L'insertion d'une clé	10
3.4	La suppression d'une clé	12
3.5	Application : un algorithme de tri	12
3.5.1	Quelques applications directes des algorithmes précédents	12
3.5.2	Le tri proprement dit	12
4	Quelques éléments pour une combinatoire des arbres 2-3	15
4.1	Fonctions génératrices associées	15
4.2	Pour une analyse plus fine	16

Table des figures

1	les deux types de nœuds utilisés	4
2	un arbre 2-3 avant insertion	5
3	après insertion de la clé 3	5
4	après insertion de la clé 21	5
5	après recomposition	6
6	après insertion de la clé 11	6
7	après insertion de la clé 13	7
8	un cas simple de résolution d'un fils unique	7
9	un autre cas simple de résolution	8
10	une résolution d'un fils unique un peu plus difficile	8
11	une résolution qui répercute un nouveau fils unique	8
12	répartition k_3/k pour $p = 4$	17

Table des programmes

1	le typage des arbres 2-3	10
2	la recherche d'une clé	10
3	l'insertion d'une clé	11
4	la suppression d'une clé	13
5	quelques applications immédiates	14
6	le tri d'une liste grâce aux arbres 2-3	14

1 Les arbres 2-3

1.1 Arbres de recherche

Considérons un ensemble de clés d'un certain type α . Nous supposons qu'il existe une application que nous nommerons *valuation* $v : a \mapsto v(a)$ qui associe un entier naturel à chaque clé, permettant ainsi de munir leur ensemble d'un ordre¹ \prec défini par $a \preceq b \iff v(a) \leq v(b)$.

Nous cherchons à définir une structure sur l'ensemble de ces clés qui permettent les opérations de base suivantes :

- (i) l'insertion d'une nouvelle clé dans la structure ;
- (ii) la recherche d'une clé particulière ;
- (iii) la suppression d'une clé présente dans la structure ;
- (iv) l'extraction de la clé de valuation minimale (*resp.* minimale).

Une méthode habituelle consiste à utiliser une structure d'arbre binaire : les feuilles de l'arbre contiendront les clés de la structure, les nœuds une *fonction de sélection* qui pourra décider, la valuation d'une clé étant donnée, s'il convient de descendre à gauche ou à droite dans l'arbre.

Les types CAML génériques correspondant seraient donc les suivants :

```
type descente = Gauche | Droite ;;

type 'a valuation == 'a -> int ;;

type 'a arbre_de_recherche =
  | Feuille of 'a
  | Nœud of (int -> descente)
           * ('a arbre_de_recherche)
           * ('a arbre_de_recherche) ;;
```

La généralisation à un arbre n -aire se ferait aisément.

En pratique, on choisit pour la fonction de sélection de fixer (dans le cas d'un arbre binaire) une valeur v_0 telle que si la valuation d'une clé lui est inférieure, on s'orientera vers la gauche, et vers la droite dans le cas contraire. Pour le cas d'un nœud d'arité n , il conviendra de choisir $n - 1$ valeurs entières qui permettront de sélectionner la branche descendante à parcourir.

L'un des intérêts de ce choix est de résoudre facilement le problème de l'extraction de la clé de valeur minimale (ou maximale) : il suffira de descendre systématiquement à gauche (ou à droite) pour atteindre à coup sûr la clé recherchée.

Bien entendu, tout le problème est dans la nécessité d'éviter qu'on se retrouve avec des arbres dont la profondeur soit linéaire en leur taille. On sait bien que la profondeur d'un arbre est comprise entre un logarithme de sa taille et cette taille : le problème de *l'équilibrage* consiste à garantir qu'elle reste logarithmique.

1.2 Arbres bi- ou ter- naires

Soit donc un ensemble (infini) de clés \mathcal{A} pour lequel existe une valuation $v : \mathcal{A} \longrightarrow \mathbb{N}$.

On leur associe l'ensemble $\mathcal{B}_{2,3}$ des arbres dont les feuilles portent des valeurs de l'ensemble \mathcal{A} ; qui vérifient la propriété *tout nœud est d'arité 2 ou 3* ; et dont les nœuds d'arité 2 portent un entier quand les nœuds d'arité 3 en portent deux. De tels arbres seront appelés *arbres 2-3 généraux*.

À un nœud d'arité 2 qui porte l'entier r est associée la fonction de sélection qui fait descendre à gauche (*resp.* à droite) toute clé x telle que $v(x) \leq r$ (*resp.* $v(x) > r$).

À un nœud d'arité 3 qui porte le couple d'entiers (r, s) avec $r < s$ est associée la fonction de sélection qui fait descendre à gauche (*resp.* au milieu) (*resp.* à droite) toute clé x telle que $v(x) \leq r$ (*resp.* $r < v(x) \leq s$) (*resp.* $s < v(x)$).

Nous utiliserons la représentation graphique de la figure 1 page suivante pour les deux types de nœuds.

Une telle structure conserve l'information dans ses feuilles, et les clés sont triées en ordre croissant dans le parcours préfixe, infixé ou suffixé, de ses feuilles.

¹tout bon ordre suffirait, valuation ou pas

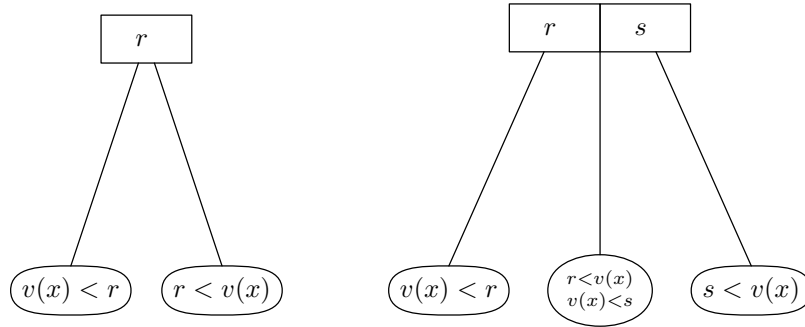


FIG. 1: les deux types de nœuds utilisés

La clé de valeur minimale (*resp.* maximale) se trouvera alors naturellement en premier (*resp.* en dernier) dans tout parcours préfixe, infixé ou suffixé des feuilles.

1.3 Équilibrage

Soit n le nombre de feuilles d'un arbre dont tous les nœuds sont d'arité 2 ou 3, et p sa profondeur, c'est-à-dire le nombre maximal d'arêtes qu'il faut traverser dans un parcours depuis la racine jusqu'à une des feuilles.

On dispose alors d'un résultat (classiquement démontré dans le cas d'arbres strictement binaires) qui énonce que $\lceil \log_3 n \rceil \leq p \leq n - 1$.

Si l'on garantit que toutes les feuilles étaient de même profondeur p , on dispose d'un résultat plus fort qu'on peut énoncer ainsi :

Théorème 1

Si un arbre 2-3 général comporte n feuilles qui sont toutes à profondeur p , alors on dispose de la relation :

$$2^p \leq n \leq 3^p, \quad \text{ou encore} \quad \lg_3 n \leq p \leq \lg_2 n.$$

1.4 Un petit point historique

Dans la suite, nous ne considérerons que des arbres 2-3 généraux dont toutes les feuilles sont à la même profondeur, que nous appellerons simplement *arbres 2-3*.

Ces arbres, qu'on appelle parfois aussi des *arbres B^+* , ont été imaginés — ainsi que les algorithmes que nous allons décrire ci-dessous — pour la première fois en 1979 par D. Comer dans [1].

1.5 Recherche d'une clé

La recherche d'une clé dans un arbre 2-3 ne pose pas de problème particulier : l'information incluse dans chaque nœud permet de choisir la branche de descente.

D'après le théorème précédent, il est clair que la recherche d'une feuille d'un arbre 2-3 qui en possède n a un coût logarithmique $c(\text{recherche}) = \Omega(\lg n)$. Notons qu'il s'agit d'une évaluation qui vaut dans tous les cas, et pas seulement dans le meilleur.

1.6 Insertion d'une clé

L'insertion d'une clé se fait classiquement comme toute insertion aux feuilles dans un arbre de recherche. Considérons l'exemple de l'arbre 2-3 de la figure 2 page suivante.

L'insertion d'une nouvelle clé de valeur 3 se fait en remplaçant simplement un nœud binaire par un nœud ternaire, et on obtient l'arbre de la figure 3 page suivante.

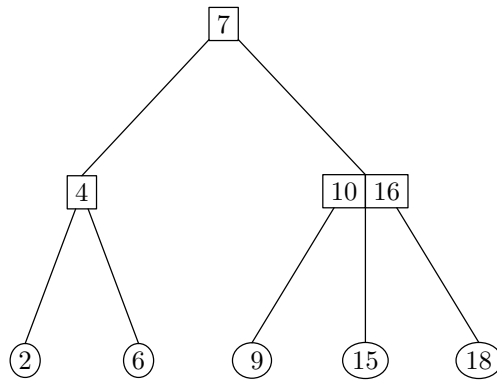


FIG. 2: un arbre 2-3 avant insertion

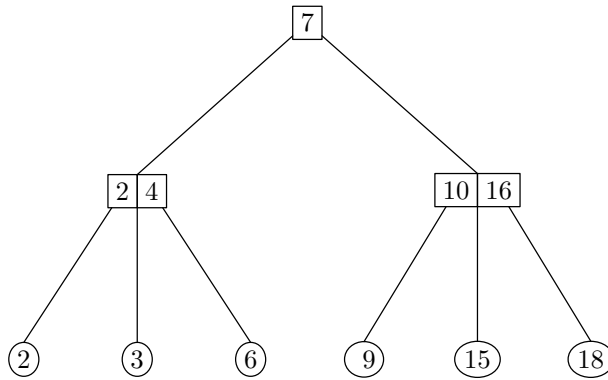


FIG. 3: après insertion de la clé 3

La difficulté à résoudre se présente évidemment quand on se retrouve avec un nœud à 4 feuilles : la clé insérée s'est installée comme fille d'un nœud qui avait déjà trois feuilles. C'est ce qui se passe (voir la figure 4) lorsque par exemple on insère la clé 21 dans l'arbre de la figure 2.

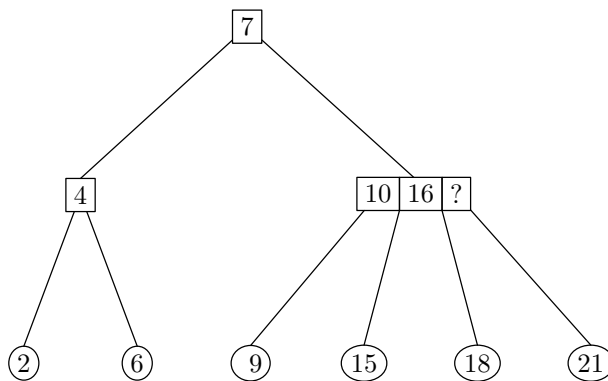


FIG. 4: après insertion de la clé 21

Il s'agit alors de recomposer l'arbre, en *éclatant* le nouveau nœud à 4 feuilles formé en deux nœuds binaires.

Ainsi, l'arbre précédent de la figure 4, qui n'est pas 2-3, devient-il celui de la figure 5 page suivante.

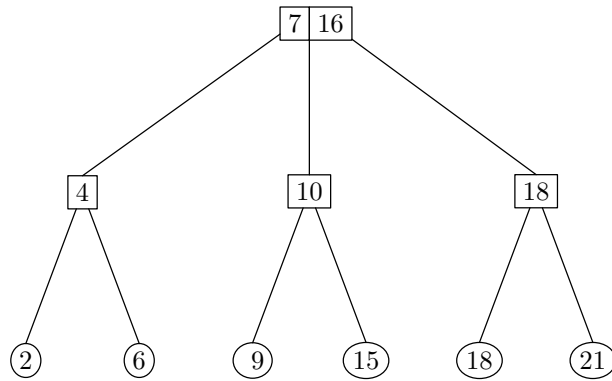


FIG. 5: après recomposition

Bien entendu, il se peut que l'arbre père du nœud 4 temporaire soit déjà ternaire, et il faudra dans ce cas recommencer en éclatant le père en deux nœuds binaires : on peut ainsi être amené à remonter tout en haut de l'arbre, ce qui est le seul cas où la profondeur totale augmente au cours de l'insertion d'une clé. Par exemple, on vérifiera facilement que les insertions successives des clés 11 et 13 dans l'arbre de la figure 5 conduisent aux arbres des figures 6 et 7 page suivante.

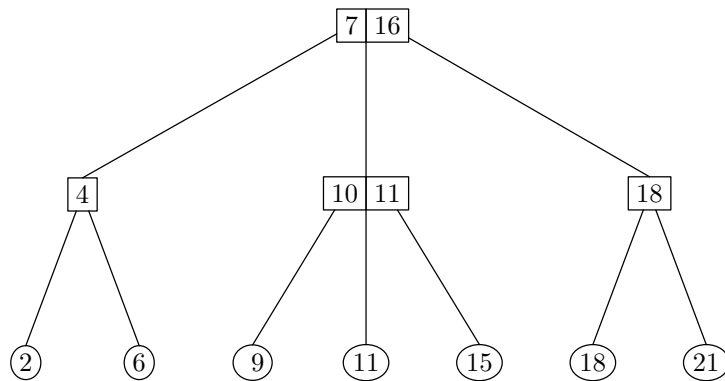


FIG. 6: après insertion de la clé 11

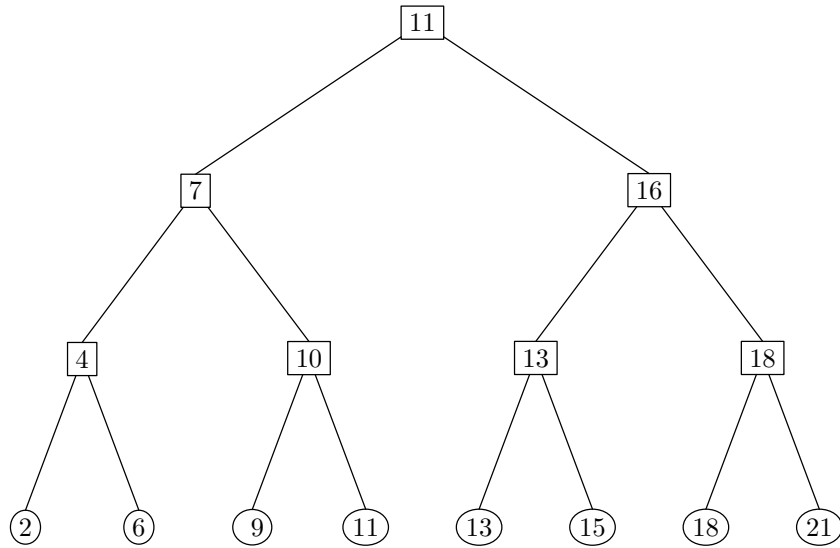


FIG. 7: après insertion de la clé 13

Finalement, on a un algorithme qui descend tout en bas de l'arbre avant (dans le pire des cas) de remonter tout en haut, et le coût de l'insertion est encore $c(\text{insertion}) = \Omega(\lg n)$.

1.7 Suppression d'une clé

De la même façon, pour supprimer une clé, on commence par descendre dans l'arbre jusqu'à trouver la feuille concernée.

Si son père est un nœud 3, il n'y a pas de difficulté : on remplace ce nœud ternaire par un nœud binaire, et la feuille est supprimée.

Si, en revanche, on trouve un nœud binaire, la suppression conduit à un nœud unaire qui ne respecte pas les contraintes de définition d'un arbre 2-3.

Il s'agit donc de gérer le cas où la suppression dans une branche de l'arbre conduit à un nœud unaire. On considère alors le père.

Ou bien il s'agit d'un nœud ternaire : par exemple, un nœud de sous-arbres g , m et d , et tel que g soit unaire. On distinguera les manipulations nécessaires selon que m est binaire ou ternaire. La figure 8 explicite le premier cas, la figure 9 page suivante le deuxième.

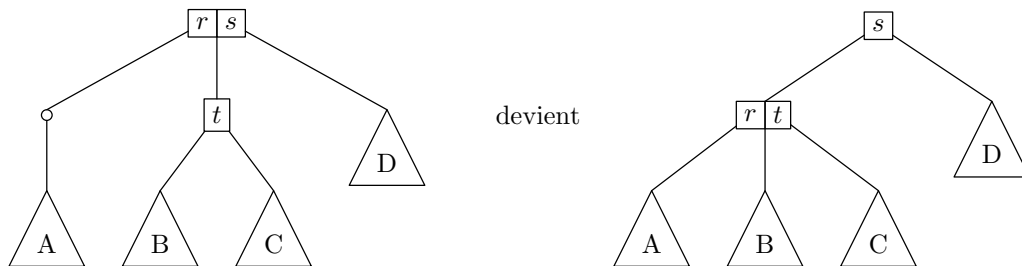


FIG. 8: un cas simple de résolution d'un fils unique

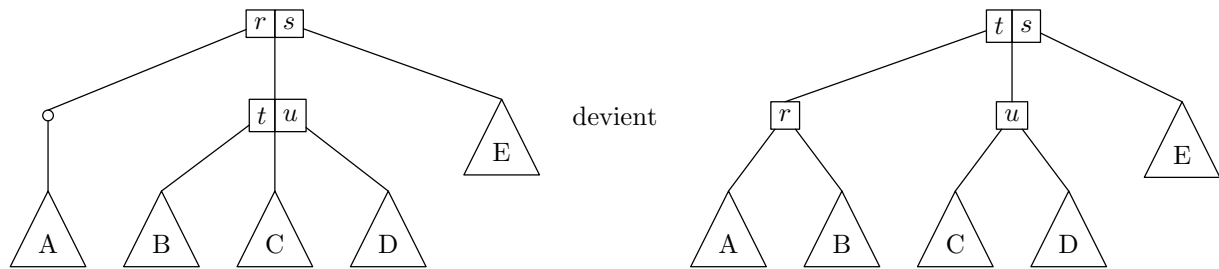


FIG. 9: un autre cas simple de résolution

Dans le cas où le père est un nœud binaire, par exemple de sous-arbres g unaire et d , la figure 10 montre comment terminer si d est ternaire. Si en revanche d est aussi binaire, on procède comme le montre la figure 11, et on doit remonter encore plus haut dans l'arbre.

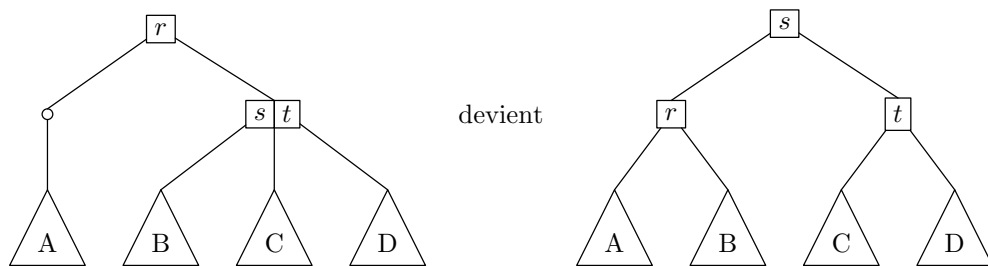


FIG. 10: une résolution d'un fils unique un peu plus difficile

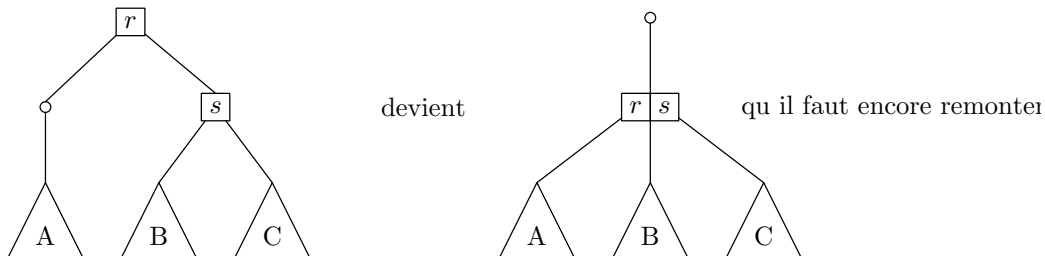


FIG. 11: une résolution qui répercute un nouveau fils unique

Finalement, on a un algorithme qui descend tout en bas de l'arbre avant (dans le pire des cas) de remonter tout en haut, et le coût de l'insertion est encore $c(\text{suppression}) = \Omega(\lg n)$.

2 Généralisation : les arbres a - b

2.1 Définitions

Soit $a \geq 2$ et $b \geq 2a - 1$. Un arbre a - b est un arbre tel que

- toutes les feuilles sont à la même profondeur ;
- la racine est de degré compris entre 2 et b (au sens large) ;
- tous les autres nœuds sont de degré compris entre a et b (au sens large).

Chaque nœud de degré k (avec donc $a \leq k \leq b$ et $2 \leq k \leq b$ pour la racine) contient un aiguillage formé d'un $(k - 1)$ -uplet d'entiers, qui permet de choisir la branche où descendre dans la recherche d'une clé de valuation donnée.

Bien entendu, on dispose, comme à la section 1.3 page 4, du

Théorème 2

Si un arbre a - b comporte n feuilles qui sont toutes à profondeur p , alors on dispose de la relation :

$$2a^{p-1} \leq n \leq b^p, \quad \text{ou encore} \quad \lg_b n \leq p \leq 1 + \lg_a(n/2).$$

2.2 Insertion et suppression d'une clé

Les algorithmes d'insertion et de suppression sont tout à fait analogues à ceux qu'on a décrit plus haut dans le cas où $a = 2$ et $b = 3$.

Pour l'insertion, il conviendra d'éclater un nœud de degré $b + 1$ en deux nœuds de degrés respectifs $\lfloor (b + 1)/2 \rfloor$ et $\lceil (b + 1)/2 \rceil$. On constate que la condition $b \geq 2a - 1$ entraîne $\lceil (b + 1)/2 \rceil \geq \lfloor (b + 1)/2 \rfloor \geq a$, et il est alors facile de généraliser l'algorithme d'insertion déjà écrit.

De la même façon, on réécrira l'algorithme de suppression, en remontant au père d'un nœud qui n'aurait plus que $a - 1$ fils, ce que nous laissons faire au lecteur de ces lignes...

3 Implantation en Caml des arbres 2-3

3.1 Le typage des arbres

Le typage correspond naturellement à la définition qu'on a donnée des arbres 2-3 : pour faciliter la lecture des structures, il nous a paru plus agréable d'écrire un `Nœud3` de branches `g`, `m` et `d` et de sélecteur `(r,s)` sous la forme `Nœud3(g,r,m,s,d)`, de sorte que chaque entier se positionne entre les branches qu'il discrimine. Nous avons choisi des étiquettes entières pour cet article, et la valuation est donc simplement la fonction identité.

Programme 1 le typage des arbres 2-3

```
1 type type_des_clés == int ;; (* par exemple *)
2
3 let valuation x = x ;;
4
5 type arbre23 =
6   | Feuille of type_des_clés
7   | Nœud2 of arbre23 * int * arbre23
8   | Nœud3 of arbre23 * int * arbre23 * int * arbre23 ;;
```

3.2 La recherche d'une clé

La recherche d'une clé est une simple adaptation du programme classique de recherche dans un arbre binaire de recherche et n'a pas besoin de davantage de commentaires.

Programme 2 la recherche d'une clé

```
7 let recherche v a x =
8   let vx = v x
9   in
10  let rec cherche = function
11    | Feuille y -> x = y
12    | Nœud2(g,r,d) -> cherche (if vx <= r then g else d)
13    | Nœud3(g,r,m,s,d) -> cherche (if vx <= r then g else if vx <= s then m else d)
14  in
15  cherche a ;;
```

3.3 L'insertion d'une clé

L'insertion d'une clé, qui fait l'objet du programme 3 page suivante, est beaucoup plus intéressante.

La *clef* du programme est dans l'utilisation d'une exception : la description de l'algorithme que nous avons proposé à la sous-section 1.6 page 4 prévoyait, dans le cas où l'insertion se faisait à un niveau où se trouvaient déjà 3 clés, d'éclater l'arbre obtenu, et de remonter dans l'arbre, où pouvaient encore se produire des éclatements supplémentaires. La difficulté est donc de *remonter* dans un arbre dont la structure ne prévoit *a priori* que le moyen d'y *descendre*... C'est là qu'intervient d'une manière qui me semble très élégante l'utilisation d'une exception.

Notons que l'éclatement peut se propager jusqu'à la racine de l'arbre (rappelons au passage que c'est d'ailleurs le seul cas où l'arbre voit sa profondeur augmenter d'une unité), et c'est là qu'intervient la structure `try ... with ...` de la ligne 47.

Programme 3 l'insertion d'une clé

```
16 exception Éclaté of arbre23 * int * arbre23 ;;

17 let insertion23 v a x =
18   let vx = v x
19   in
20   let rec insère a = match a with
21     | Feuille y when x = y -> a
22     | Feuille y when vx < (v y)
23       -> raise (Éclaté (Feuille x, vx, Feuille y))
24     | Feuille y (* when vx > (v y) *)
25       -> raise (Éclaté (Feuille y, v y, Feuille x))
26     | Nœud2(g,r,d) when vx <= r
27       -> (try Nœud2(insère g,r,d)
28         with Éclaté(g',r',d') -> Nœud3(g',r',d',r,d))
29     | Nœud2(g,r,d) (* when vx > r *)
30       -> (try Nœud2(g,r,insère d)
31         with Éclaté(g',r',d') -> Nœud3(g,r,g',r',d'))
32     | Nœud3(g,r,m,s,d) when vx <= r
33       -> (try Nœud3(insère g,r,m,s,d)
34         with Éclaté(g',r',d') -> raise (Éclaté(Nœud2(g',r',d'),r,Nœud2(m,s,d))))
35     | Nœud3(g,r,m,s,d) when vx <= s (* and vx > r *)
36       -> (try Nœud3(g,r,insère m,s,d)
37         with Éclaté(g',r',d') -> raise (Éclaté(Nœud2(g,r,g'),r',Nœud2(d',s,d))))
38     | Nœud3(g,r,m,s,d) (* when vx > s *)
39       -> (try Nœud3(g,r,m,s,insère d)
40         with Éclaté(g',s',d') -> raise (Éclaté(Nœud2(g,r,m),s,Nœud2(g',s',d'))))
41   in
42   match a with
43     | Feuille y
44       -> if x = y then a
45          else if vx < (v y) then Nœud2( Feuille x,vx, Feuille y)
46          else Nœud2( Feuille y,v y, Feuille x)
47     | _ -> try insère a with Éclaté(g,r,d) -> Nœud2(g,r,d) ;;
```

3.4 La suppression d'une clé

On écrit l'algorithme de suppression d'une façon similaire : pour remonter dans la structure, on définit une exception, qui s'appelle cette fois `FilsUnique`, et qui est déclenchée quand la suppression normale d'une clé produit un nœud unaire, qu'il est alors nécessaire de remonter.

On obtient le programme 4 page suivante.

On notera l'utilisation de cas de filtrage du genre `Feuille _ -> raise Unexpected` qui n'ont d'autre fonction que satisfaire le parseur de Caml, et éviter des messages d'avertissement de filtrages non exhaustifs. C'est toujours de bonne politique que d'éviter ce genre de messages, et cela garantit une programmation plus sûre.

3.5 Application : un algorithme de tri

3.5.1 Quelques applications directes des algorithmes précédents

On écrit très facilement la recherche de la clé de valuation minimale (*resp.* maximale) d'un arbre 2-3 : il suffit de descendre systématiquement à droite (*resp.* à gauche) dans la structure.

De même, à l'aide de la fonction d'insertion, peut-on écrire une fonction qui transforme une liste en arbre 2-3.

Ces fonctions font l'objet du programme 5 page 14.

3.5.2 Le tri proprement dit

Le tri n'est alors qu'une suite d'extractions des maxims successifs de l'arbre 2-3 construit à partir de la liste argument, ce que réalise facilement le programme 6 page 14.

Programme 4 la suppression d'une clé

```
48 exception Unexpected ;; (* bug du programme *)
49 exception FilsUnique of arbre23 ;;

50 let suppression23 v a x =
51   let vx = v x
52   in
53   let rec supprime a = match a with
54     | Feuille y when x = y -> raise Unexpected
55     | Feuille _ -> a
56     | Nœud2(Feuille y,_,d) when vx = (v y) -> raise (FilsUnique d)
57     | Nœud2(g,_,Feuille y) when vx = (v y) -> raise (FilsUnique g)
58     | Nœud3(Feuille y,_,m,s,d) when vx = (v y) -> Nœud2(m,s,d)
59     | Nœud3(g,_,Feuille y,s,d) when vx = (v y) -> Nœud2(g,s,d)
60     | Nœud3(g,s,m,_,Feuille y) when vx = (v y) -> Nœud2(g,s,m)
61     | Nœud2(g,r,d) when vx <= r
62       -> ( try
63             Nœud2(supprime g,r,d)
64           with FilsUnique g'
65             -> match d with
66               | Nœud3(gd,rd,md,sd,dd) -> Nœud2(Nœud2(g',r,gd),rd,Nœud2(md,sd,dd))
67               | Nœud2(gd,rd,dd) -> raise (FilsUnique (Nœud3(g',r,gd,rd,dd)))
68               | Feuille _ -> raise Unexpected (* pour que le filtrage soit exhaustif *) )
69     | Nœud2(g,r,d) (* when vx > r *)
70       -> ( try
71             Nœud2(g,r,supprime d)
72           with FilsUnique d'
73             -> match g with
74               | Nœud3(gg,rg,mg,sg,dg) -> Nœud2(Nœud2(gg,rg,mg),sg,Nœud2(dg,r,d'))
75               | Nœud2(gg,rg,dg) -> raise (FilsUnique (Nœud3(gg,rg,dg,r,d')))
76               | Feuille _ -> raise Unexpected (* pour que le filtrage soit exhaustif *) )
77     | Nœud3(g,r,m,s,d) when vx <= r
78       -> ( try
79             Nœud3(supprime g,r,m,s,d)
80           with FilsUnique g'
81             -> match m with
82               | Nœud3(gm,rm,mm,sm,dm) -> Nœud3(Nœud2(g',r,gm),rm,Nœud2(mm,sm,dm),s,d)
83               | Nœud2(gm,rm,dm) -> Nœud2(Nœud3(g',r,gm,rm,dm),s,d)
84               | Feuille _ -> raise Unexpected (* pour que le filtrage soit exhaustif *) )
85     | Nœud3(g,r,m,s,d) when vx <= s
86       -> ( try
87             Nœud3(g,r,supprime m,s,d)
88           with FilsUnique m'
89             -> match g with
90               | Nœud3(gg,rg,mg,sg,dg) -> Nœud3(Nœud2(gg,rg,mg),sg,Nœud2(dg,r,m'),s,d)
91               | Nœud2(gg,rg,dg) -> Nœud2(Nœud3(gg,rg,dg,r,m'),s,d)
92               | Feuille _ -> raise Unexpected (* pour que le filtrage soit exhaustif *) )
93     | Nœud3(g,r,m,s,d) (* when vx > s *)
94       -> ( try
95             Nœud3(g,r,m,s,supprime d)
96           with FilsUnique d'
97             -> match m with
98               | Nœud3(gm,rm,mm,sm,dm) -> Nœud3(g,r,Nœud2(gm,rm,mm),sm,Nœud2(dm,s,d'))
99               | Nœud2(gm,rm,dm) -> Nœud2(g,r,Nœud3(gm,rm,dm,s,d'))
100              | Feuille _ -> raise Unexpected (* pour que le filtrage soit exhaustif *) )
101   in
102   match a with
103   | Feuille y when vx = (v y) -> failwith "je ne veux pas d'arbre 2-3 vide"
104   | Feuille _ -> a
105   | _ -> try supprime a with FilsUnique u -> u ;;
```

Programme 5 quelques applications immédiates

```
106 let rec min23 = function
107   | Feuille x -> x
108   | Nœud2(g,_,_) -> min23 g
109   | Nœud3(g,_,_,_) -> min23 g
110 and max23 = function
111   | Feuille x -> x
112   | Nœud2(_,_,d) -> max23 d
113   | Nœud3(_,_,_,d) -> max23 d ;;

114 let rec arbre23_of_list v = function
115   | [] -> failwith "un arbre 2-3 ne peut être vide"
116   | [ t ] -> Feuille t
117   | t :: q -> insertion23 v (arbre23_of_list v q) t ;;
```

Programme 6 le tri d'une liste grâce aux arbres 2-3

```
118 let tri23 v l =
119   let a = arbre23_of_list v l
120   in
121   let rec recompose tampon = function
122     | Feuille x -> x :: tampon
123     | a -> let x = max23 a in recompose (x :: tampon) (suppression23 v a x)
124   in
125   recompose [] a ;;
```

4 Quelques éléments pour une combinatoire des arbres 2-3

4.1 Fonctions génératrices associées

Nous noterons, pour une profondeur $p \in \mathbb{N}$ et $n \in \mathbb{N}$, $B_{p,n}$ le nombre d'arbres 2-3 de profondeur p qui possèdent n feuilles. Par exemple : $B_{0,n} = \begin{cases} 1, & \text{si } n = 1; \\ 0, & \text{sinon.} \end{cases}$

Les fonctions génératrices associées sont les séries formelles, éléments de $\mathbb{N}[[z]]$, définies par :

$$B_p(z) = \sum_{n=0}^{+\infty} B_{p,n} z^n.$$

On a donc en particulier $B_0(z) = z$.

Soit maintenant $p \geq 1$. La racine d'un arbre 2-3 de taille n et de profondeur p possède deux ou trois fils qui sont chacun des arbres 2-3 de profondeurs $p-1$ et dont la somme des tailles vaut n .

On peut donc écrire :

$$\forall p \geq 1, \forall n, B_{p,n} = \sum_{n_1+n_2=n} B_{p-1,n_1} B_{p-1,n_2} + \sum_{n_1+n_2+n_3=n} B_{p-1,n_1} B_{p-1,n_2} B_{p-1,n_3}.$$

La traduction en termes de fonctions génératrices est directe, et on obtient :

$$B_p(z) = \begin{cases} z, & \text{si } p = 0; \\ B_{p-1}(z)^2 + B_{p-1}(z)^3, & \text{si } p \geq 1. \end{cases}$$

Dans [3][page 292], Flajolet et Sedgewick proposent la récurrence fonctionnelle suivante :

$$B_p(z) = \begin{cases} z, & \text{si } p = 0; \\ B_{p-1}(z^2 + z^3), & \text{si } p \geq 1. \end{cases}$$

Le point de vue utilisé est totalement différent : cette fois, on passe d'une profondeur $p-1$ à une profondeur p en remplaçant chaque feuille par, au choix, un arbre de profondeur 1 qui est soit binaire soit ternaire, ce qui correspond à la substitution $z \leftarrow z^2 + z^3$.

Remarquons qu'une démonstration directe de l'équivalence des deux récurrences fonctionnelles écrites n'aurait vraiment rien d'évident².

On obtient les valeurs suivantes, pour $0 \leq n \leq 20$:

p	0	1	2					2 ou 3		3							3 ou 4				
n	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	
$B_{0,n}$	1																				
$B_{1,n}$		1	1																		
$B_{2,n}$				1	2	2	3	3	1												
$B_{3,n}$								1	4	8	14	23	32	43	63	96	141	192	240	267	
$B_{4,n}$																1	8	32	92	222	
$\sum_p B_{p,n}$	1	1	1	1	2	2	3	4	5	8	14	23	32	43	63	97	149	224	332	489	

En sommant à profondeur fixée, on obtient : $\sum_n B_{p,n} = B_p(1)$ d'où le tableau de valeurs suivant :

p	0	1	2	3	4	5
$\sum_n B_{p,n} = B_p(1)$	1	2	12	1872	6563711232	282779810171805015122254036992

²mais m'intéresserait quand même beaucoup !

Si l'on s'intéresse au nombre moyen de feuilles d'un arbre 2-3 de profondeur p , il suffit d'évaluer la quantité $\bar{n} = B'_p(1)/B_p(1)$, et MAPLE nous aide à trouver les valeurs suivantes :

p	0	1	2	3	4	5	6
\bar{n}	1	2,5	6,667	19,487	58,451	175,353	526,060

Il n'est pas beaucoup plus difficile d'évaluer la variance de ce nombre de feuilles, puisque la moyenne du carré de n s'écrit $(B'_p(1) + B''_p(1))/B_p(1)$, et là encore MAPLE nous fournit les résultats demandés :

p	0	1	2	3	4	5	6
$V(n)$	0	0,25	2,056	9,164	27,691	83,073	249,218
$\sqrt{V(n)}$	0	0,5	1,434	3,027	5,262	9,114	15,787

4.2 Pour une analyse plus fine

Pour raffiner notre analyse de la combinatoire des arbres 2-3, nous pouvons nous intéresser au nombre k de nœuds (binaires ou ternaires) d'un arbre 2-3 de profondeur p et de taille n , que nous noterons $B_{p,n}^{(k)}$. Pour un arbre binaire, on sait que le nombre de feuilles est égal au nombre de nœuds plus un. Dans le cas de nos arbres 2-3, nous disposons de l'encadrement analogue : $k+1 \leq n \leq 2k+1$. En outre, k et p vérifient la relation : $2^p - 1 \leq k \leq \frac{3^p - 1}{2}$ ce qui donne encore : $2^p \leq n \leq 3^p$ comme on pouvait s'y attendre.

Pour montrer que l'information décrite par le triplet (n, p, k) est plus riche que prévue, montrons le

Théorème 3

Soient k_2 (resp. k_3) le nombre de nœuds binaires (resp. ternaires) d'un arbre 2-3 possédant k nœuds et n feuilles. Alors : $k_2 = 2k - n + 1$ et $k_3 = n - k - 1$.

Observons tout d'abord que l'on dispose bien sûr de $k_2 + k_3 = k$.

Une deuxième équation s'obtient facilement, en considérant que tout nœud/feuille sauf la racine a un père et qu'on a donc : $2k_2 + 3k_3 = k + n - 1$.

Il suffit alors de résoudre le système de nos deux équations.

Introduisons la fonction génératrice à deux variables $B_p(z, u) = \sum_n \sum_k B_{p,n}^{(k)} z^n u^k$.

Bien sûr, on aura $B_p(z) = B_p(z, 1)$ et $B_0(z, u) = z$.

Essayons d'établir des récurrences sur ces entiers $B_{p,n}^{(k)}$.

Partant de la racine, on construit un arbre de profondeur $p \geq 1$ depuis une racine binaire ou ternaire, ce qui fournit :

$$\forall p \geq 1, \forall k \geq 1, B_{p,n}^{(k)} = \sum_{\substack{n_1+n_2=n \\ k_1+k_2=k-1}} B_{p-1,n_1}^{(k_1)} B_{p-1,n_2}^{(k_2)} + \sum_{\substack{n_1+n_2+n_3=n \\ k_1+k_2+k_3=k-1}} B_{p-1,n_1}^{(k_1)} B_{p-1,n_2}^{(k_2)} B_{p-1,n_3}^{(k_3)}.$$

Traduisant ceci en termes de fonctions génératrices, on obtient :

$$B_p(z, u) = \begin{cases} z, & \text{si } p = 0; \\ u(B_{p-1}(z, u)^2 + B_{p-1}(z, u)^3), & \text{si } p \geq 1. \end{cases}$$

Transformant les n feuilles d'un arbre de profondeur $p-1$ contenant k nœuds en arbres de profondeur 1, on passe à un arbre de profondeur p , possédant $k+n$ nœuds et entre $2n$ et $3n$ feuilles. Chaque feuille donne donc naissance soit à un nœud et deux feuilles, soit à un nœud et trois feuilles, ce qui se traduit par la substitution $z \leftarrow u(z^2 + z^3)$, et on obtient une nouvelle récurrence fonctionnelle :

$$B_p(z, u) = \begin{cases} z, & \text{si } p = 0; \\ B_{p-1}(u(z^2 + z^3), u), & \text{si } p \geq 1. \end{cases}$$

C'est cette deuxième récurrence que nous avons utilisée pour écrire en CAML le calcul de la fonction génératrice. On trouvera le listing correspondant dans le dossier complet lié à cet article, mais nous ne le reproduisons pas ici.

La figure 12 (qui est une copie d'écran, bien entendu) présente la répartition des nœuds binaires/ternaires pour les arbres 2-3 de profondeur $p = 4$: on lit en abscisses le rapport $k_3/k \in [0, 1]$ (graduation tous les dixièmes), et en ordonnées les fréquences relatives (normalisées par la fréquence maximale). On aurait aimé présenter des statistiques pour des profondeurs plus importantes, mais CAML déclenche trop vite l'exception `Out_of_memory...`

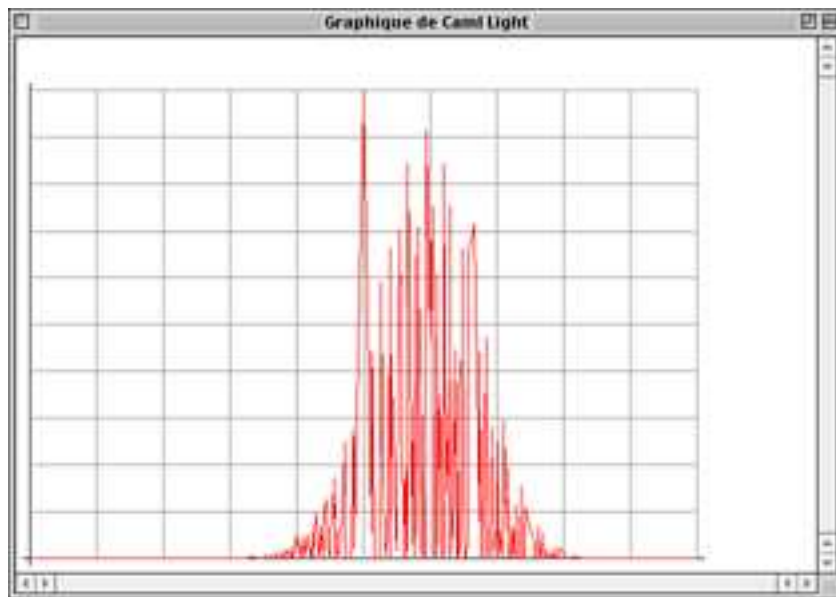


FIG. 12: répartition k_3/k pour $p = 4$

Références

- [1] D. Comer. The ubiquitous b-tree. *Computing Surveys*, 11 :121–137, 1979.
- [2] Danièle Beauquier, Jean Berstel et Philippe Chrétienne. *Éléments d’algorithmique*. Masson, Paris, 1992.
- [3] Philippe Flajolet and Robert Sedgewick. *An Introduction to the Analysis of Algorithms*. Addison-Wesley, Reading, Massachussets, 1996.
- [4] Ronald L. Graham, Donald E. Knuth, and Oren Patashnik. *Concrete Mathematics : A Foundation for Computer Science*. Addison-Wesley, Reading, Massachussets, 2nd edition, 1994.
- [5] Derick Wood. *Data structures, algorithms, and performance*. Addison-Wesley, Reading, Massachussets, 1993.