La lettre de Caml numéro 5

Laurent Chéno
54, rue Saint-Maur
75011 Paris
Tél. 01 48 05 16 04
Fax 01 48 07 80 18
email : `cheno@micronet.fr`

automne 1996

Édito

*Les files de priorité sont des structures de données qui permettent essentiellement deux opérations de base : l'insertion d'une nouvelle clé et la suppression de la plus petite (ou de la plus grande). Il existe de nombreuses implémentations des files de priorité, comme par exemple les tas (*heap *en anglais) que tous connaissent certainement.*
Nous présentons ici deux versions moins connues et moins documentées dans la plupart des ouvrages, qui sont les files binomiales, à base de forêts d'arbres binaires, et les pagodes, arbres binaires aux pointeurs franchement exotiques.
Ces deux structures permettent l'implémentation de tris en $O(n \lg n)$ pour les files binomiales, et aussi pour les pagodes, mais alors seulement en moyenne.
Enfin, nous présentons un algorithme pour dessiner des arbres binaires, voire n-aires, qui allie esthétique et simplicité. Le problème a été longuement discuté par différents auteurs, et nous tentons ici de donner une solution qui tient compte des discussions passées. En revanche, nous ne fournissons là, contrairement à ce qu'on pourrait attendre d'une Lettre de Caml*, presque pas la moindre ligne de Caml, car à l'usage c'est un programme produisant à partir de la structure d'un arbre un nouveau programme dans un langage de dessin graphique, comme MetaPost, qu'il conviendrait d'écrire.*

Table des matières

Arbres et forêts binomiaux

Définitions

La figure 1 explicite la définition récursive des arbres binomiaux. L'arbre B_0 est constitué d'un unique nœud. Supposant construit B_k, on obtient B_{k+1} en ajoutant comme premier fils à gauche de la racine de B_k une copie de B_k lui-même.

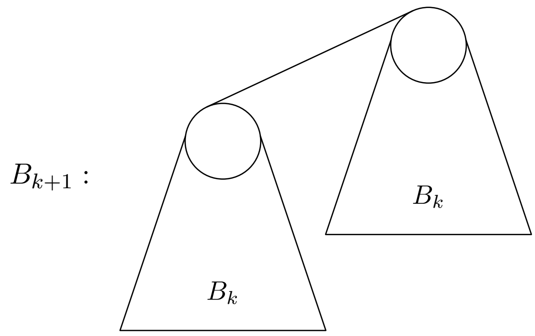


Figure 1: Définition récursive des arbres binomiaux

La figure 2 montre ce à quoi ressemblent B_0, B_1, B_2, B_3 et B_4.

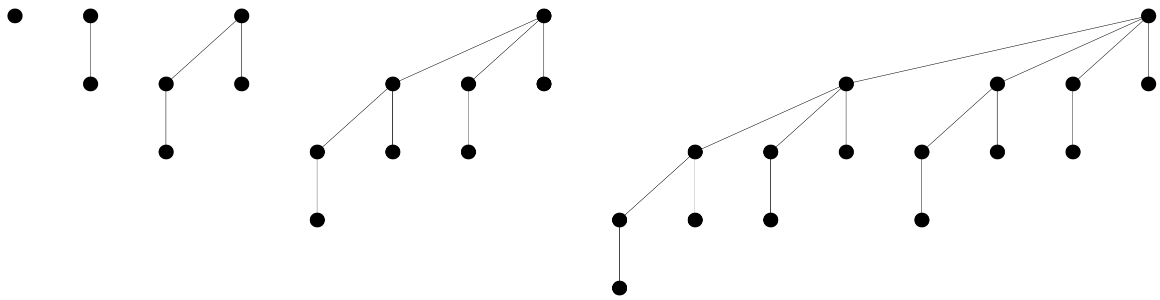


Figure 2: Les premiers arbres binomiaux

Il est assez évident que B_k est de taille 2^k et de profondeur k. Plus amusant est de montrer que B_k possède exactement $\binom{k}{i}$ nœuds à la profondeur i (avec $i \in \{0, \dots, k\}$).

Étant donné un entier $n > 0$, on peut le décomposer en base 2 : $n = \sum_{i=0}^{+\infty} b_i 2^i$, où les b_i sont dans $\{0, 1\}$, un nombre fini d'entre eux étant non nuls.

C'est dire que l'on peut *a priori* représenter toute collection de n objets par une liste d'arbres binomiaux, constituée des arbres B_i pour lesquels $b_i = 1$.

Remarquons au passage (cela sera utile tout à l'heure) que si l'on supprime la racine d'un arbre binomial B_k on obtient une file binomiale de taille $2^k - 1$, constituée de tous les arbres B_0, B_1, …, B_{k-1}.

Typages en Caml

Voici les types que je propose d'utiliser ici (toutes les informations portées par les arbres sont entières, pour alléger l'écriture).

```
type arbre_binomial = Nœud of int * arbre_binomial list ;;

type arbre_de_la_forêt = Rien | A of arbre_binomial
and forêt_d'arbres == arbre_de_la_forêt list ;;
```

On a choisi de représenter une forêt de taille n par une liste constituée de `Rien` aux positions i telles que $b_i = 0$, et d'un arbre binomial réel aux autres positions.
On peut aussi se contenter de s'intéresser à la *géométrie* des arbres et forêts, ce qui fait l'objet des déclarations suivantes.

```
type squelette_binomial = Jointure of squelette_binomial list ;;

type squelette_de_la_forêt = R | S of squelette_binomial
and forêt_de_squelettes == squelette_de_la_forêt list ;;
```

Premières fonctions Caml

Pour commencer, nous écrivons les fonctions qui permettent de construire les squelettes des arbres binaires B_k et la forêt de squelettes de taille n.

```
let odd n = n mod 2 = 1
and even n = n mod 2 = 0 ;;

let rec squelette_binomial n =
    if n = 0 then Jointure([])
    else match squelette_binomial (n-1) with
            | Jointure(fils) -> Jointure(squelette_binomial (n-1) :: fils) ;;

let rec forêt_de_squelettes n =
    let rec forêt_rec n k =
        if n = 0 then [ ]
        else (if odd n then S(squelette_binomial k) else R)
             :: (forêt_rec (n / 2) (k + 1))
    in
    forêt_rec n 0 ;;
```

Files binomiales

Un arbre binomial sera dit *tournoi* si la valeur portée par chaque nœud de l'arbre est plus grande que les valeurs de chacun de ses fils éventuels. Une forêt d'arbres binomiaux tournois est appelée une *file binomiale*, car on va bientôt décrire sur cette structure les opérations d'ajout d'un élément quelconque et de suppression du plus grand élément, ce qui caractérise les *files de priorité*.
Voici par exemple une fonction qui teste que son argument est bien un arbre binomial tournoi. Il ne suffit pas de vérifier récursivement que le fils le plus à gauche et l'arbre complet privé de ce fils sont deux arbres binomiaux. Il faut en outre vérifier que ces deux arbres sont de même taille. On aura noté tout l'intérêt du mot-clé `as` qu'offre CAML dans l'écriture des motifs de filtrage. Bien sûr, il faut aussi s'occuper de la comparaison qui assure que l'arbre est tournoi.

```
let rec est_binomial_tournoi = function
    | Nœud(_,[]) -> true
    | Nœud(a,(Nœud(b,r) as fils) :: q)
        -> b <= a
            && est_binomial_tournoi fils
            && est_binomial_tournoi (Nœud(a,q))
            && list_length r = list_length q ;;
```

Un algorithme universel

Les opérations que l'on souhaite implémenter sur notre structure de file binomiale sont les suivantes :

— l'ajout d'un élément ;

— l'extraction du maximum ;

— l'application d'une fonction à tous les éléments de la file, dans un ordre indéterminé.

Nous ajouterons une fonction qui renvoie la taille de la file et une autre qui vide la file.
Ceci fera l'objet du type Caml suivant :

```
type structure_de_forêt_binomiale =
    { ajout :           int -> unit ;
      extrait_maximum : unit -> int ;
      vide :            unit -> unit ;
      itère :           (int -> unit) -> unit ;
      taille :          unit -> int } ;;
```

Nous pourrons alors utiliser la structure ainsi définie pour, par exemple, réaliser le tri d'une liste d'entiers. Il suffira d'écrire :

```
let tri_par_file_binomiale l =
    let f = crée_forêt ()
    in
    do_list f.ajout l ;
    let résultat = ref []
    in
    try while true do
            résultat := f.extrait_maximum () :: !résultat
        done ; !résultat
    with _ -> !résultat ;;
```

La fonction `crée_forêt` reste bien entendu à écrire...
Nous allons voir que les opérations fondamentales d'ajout et d'extraction du maximum se déduisent toutes deux du seul algorithme de fusion de deux files binomiales.
Pour l'ajout, c'est clair, car ajouter à une forêt $\mathcal{F}$ un élément x revient à fusionner $\mathcal{F}$ avec la forêt consistant en un seul arbre B_0 réduit à x.
Pour la suppression du maximum, c'est une conséquence de la remarque que nous avions faite plus haut. En effet, si le maximum M est la racine de l'arbre B_k d'une file binomiale $\mathcal{F} = \{B_0, \ldots, B_{k-1}, B_k, B_{k+1}, \ldots, B_r\}$ (tous les arbres ne figurant pas nécessairement), supprimer M de la file revient à réaliser la fusion de $\mathcal{F} \setminus \{B_k\} = \mathcal{F}' = \{B_0, \ldots, B_{k-1}, B_{k+1}, \ldots, B_r\}$ et de la file qu'on obtient en décapitant l'arbre binomial B_k (rappelons qu'il s'agit d'une file *complète* de taille $2^k - 1$).

L'algorithme de fusion

L'algorithme de fusion repose sur une analogie avec l'addition binaire. La figure 3 montre ce qui se passe sur l'exemple de $111_2 + 110_2 = 1101_2$, ou, si l'on préfère : $7 + 6 = 13$, ce qui se traduit par $\{B_0, B_1, B_2\} \uplus \{B_1, B_2\} = \{B_0, B_2, B_3\}$.

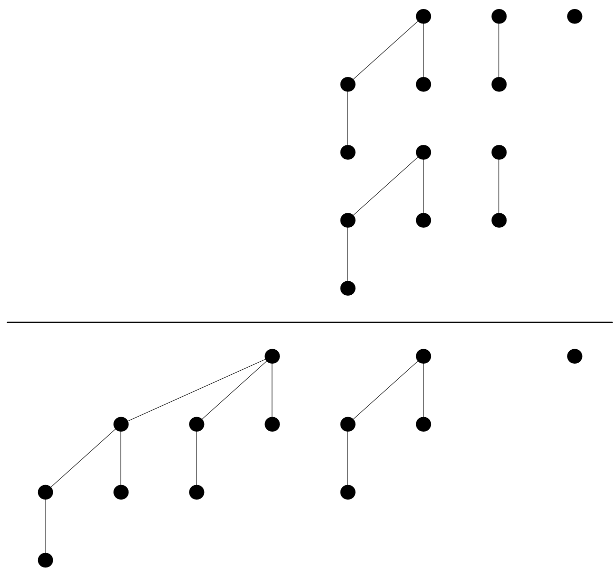


Figure 3: Une fusion de forêts binomiales, ou une addition en binaire

Pour nous entraîner, écrivons déjà la fusion sur les forêts de squelettes binomiaux.
À chaque étape, comme quand on effectue l'addition binaire à la main, on obtient une retenue (éventuellement vide), et zéro, un ou deux arbres à ajouter. L'addition $1 + 1 = 10$ correspond au cas de deux arbres à ajouter ; on obtient un arbre vide de la taille courante, et une retenue. L'addition $1 + 1 + 1 = 11$ correspond au cas de trois arbres à ajouter ; on conserve l'un des trois arbres et les deux autres fournissent la nouvelle retenue.
Voici la fonction Caml correspondante : elle attend deux arbres `a` et `b` et une retenue `r` et renvoie un couple (retenue, arbre).

```
let fusion_squelettes_binomiaux a b r = match a,b,r with
    | R,R,r -> r,R
    | R,b,R -> b,R
    | a,R,R -> a,R
    | R,S(b),S(r)
        -> R,(match b with Jointure(fils) -> S(Jointure(r :: fils)))
    | S(a),R,S(r)
        -> R,(match a with Jointure(fils) -> S(Jointure(r :: fils)))
    | S(a),S(b),_
        -> r,(match a with Jointure(fils) -> S(Jointure(b :: fils))) ;;
```

Il n'y a plus qu'à propager avec soin la retenue pour obtenir l'algorithme de fusion de deux forêts de squelettes binomiaux.

```
let fusion_forêts_squelettes f_a f_b =
    let rec fusion_rec f_a f_b r = match f_a,f_b,r with
        | [],[],R -> []
        | [],[],_ -> [ r ]
        | [],_,_ -> fusion_rec [ R ] f_b r
        | _,[],_ -> fusion_rec f_a [ R ] r
        | a::qa,b::qb,r ->  let s,r = fusion_squelettes_binomiaux a b r
                            in
                            s :: (fusion_rec qa qb r)
    in
    fusion_rec f_a f_b R ;;
```

Il y a peu d'efforts supplémentaires à effectuer dans le cas des arbres et des files binomiales ; simplement, il n'est plus indifférent de choisir celui de deux B_k de même taille qui sera à la racine du nouveau B_{k+1} fusion : il faudra choisir celui qui a la racine la plus grande.

```
let rec fusion_arbres_binomiaux a b r = match a,b,r with
    | Rien,Rien,_ -> r,Rien
    | Rien,_,Rien -> b,Rien
    | _,Rien,Rien -> a,Rien
    | Rien,_,_-> fusion_arbres_binomiaux b r a
    | _,Rien,_ -> fusion_arbres_binomiaux r a b
    | A(Nœud(a',fils_a) as arbre_a),A(Nœud(b',fils_b) as arbre_b),_
        ->  if b' <= a' then r,A(Nœud(a',arbre_b::fils_a))
                        else r,A(Nœud(b',arbre_a::fils_b)) ;;

let fusion_forêts_binomiales f_a f_b =
    let rec fusion_rec f_a f_b r = match f_a,f_b,r with
        | [],[],Rien -> []
        | [],[],_ -> [ r ]
        | [],_,_ -> fusion_rec [ Rien ] f_b r
        | _,[],_ -> fusion_rec f_a [ Rien ] r
        | a::qa,b::qb,r ->  let s,r = fusion_arbres_binomiaux a b r
                            in
                            s :: (fusion_rec qa qb r)
    in
    fusion_rec f_a f_b Rien ;;
```

La structure de file binomiale

On écrit sans la moindre difficulté la fonction qui calcule la taille d'une file.

```
let rec taille_arbre (Nœud(a,fils)) =
    it_list (fun x y -> x + (taille_arbre y)) 1 fils ;;

let rec taille_forêt = function
    | [] -> 0
    | Rien :: q -> taille_forêt q
    | A(a) :: q -> taille_forêt q + (taille_arbre a) ;;
```

La fonction qui permet d'itérer une fonction sur les éléments de la file ne présente guère plus de difficulté. Notons simplement que, à l'instar de la fonction `do_list`, elle n'a d'intérêt que si la fonction argument réalise des effets de bord, puisqu'elle renvoie toujours `()` comme résultat.

```
let rec do_forêt f forêt =
    let rec do_arbre f = function
        | Nœud(a,fils) -> f(a) ; do_list (do_arbre f) fils
    in
    match forêt with
        | [] -> ()
        | Rien :: q -> do_forêt f q
        | A(a) :: q -> do_arbre f a ; do_forêt f q ;;
```

On est maintenant en mesure d'écrire la fonction `crée_forêt`.

```
let crée_forêt () =
    let f = ref [ Rien ]
    in
    let plus_petit = fun
        | Rien _ -> true
        | _ Rien -> false
        | (A(Nœud(a,_))) (A(Nœud(b,_))) -> a <= b
    in
    let rec max_liste = function
        | [] -> failwith "Liste vide"
        | [ t ] -> t,[ Rien ]
        | t::q ->   let m,q' = max_liste q
                    in
                    if plus_petit m t   then t,(Rien :: q)
                                        else m,(t :: q')
    in
    {   ajout =
            (function x
                -> f := fusion_forêts_binomiales [ A(Nœud(x,[])) ] !f ) ;
        vide = (function () -> f := [ Rien ]) ;
        taille = (function () -> taille_forêt !f) ;
        itère = (function phi -> do_forêt phi !f) ;
        extrait_maximum =
            (function ()
                ->  try
                        let a,f' = max_liste !f
                        in
                        match a with
                            | Rien -> failwith "Vide"
                            | A(Nœud(a,fils))
                                ->  let g' = map (function a -> A(a))
                                                (rev fils)
                                    in
                                    f := fusion_forêts_binomiales f' g' ;
                                    a
                    with _ -> failwith "Forêt vide")
    } ;;
```

Évaluation

Commençons par évaluer le coût de la fusion de deux files de tailles p et q, puisque les coûts de l'ajout et de l'extraction du maximum en découleront évidemment.
Appelons $\nu(n)$ le nombre de bits égaux à 1 dans l'écriture binaire de n : $\nu(n) = \sharp\{i \mid b_i = 1\}$.
Le coût de la fusion de deux arbres binaires de même taille est évidemment un $O(1)$. Par conséquent le coût de la fusion de nos deux files de tailles p et q est égal au nombre de retenues

égales à 1 qu'on reporte dans le calcul de $p+q$ en base 2.
Nous allons montrer qu'il s'agit de $\nu(p)+\nu(q)-\nu(p+q)$.
Ainsi le coût de l'ajout d'un élément à une file de taille n vaut $\nu(n)-\nu(n+1)+1$ qui est un $O(\lg n)$. Le coût des ajouts consécutifs de n objets à une file initialement vide vaudra ainsi exactement $n-\nu(n)$.
Le coût de l'extraction du maximum se décompose en celui de sa recherche (qu'on supposera réalisée de la façon naïve que j'ai effectivement utilisée), laquelle est donc linéaire en le nombre d'arbres présents dans la file, c'est-à-dire justement en $O(\nu(n))$ et en celui de la fusion de la file de taille $p=n-2^k$ et de celle de taille $q=2^k-1$ où k est l'indice de l'arbre contenant le maximum. Or $\nu(p)=\nu(n)-1$ et $\nu(q)=k$, donc l'extraction du maximum a un coût global majoré par $\nu(n)+(\nu(n)-1+k-\nu(n-1))$. Cette expression est bien sûr un $O(\lg n)$.
L'algorithme de tri écrit plus haut est donc en $O(n\lg n)$.
Reste à établir que $\nu(p)+\nu(q)-\nu(p+q)$ compte effectivement le nombre de retenues qu'on reporte dans le calcul de $p+q$.
Le résultat est évident si p ou q est nul.
Si p et q sont pairs, $p=2p'$ et $q=2q'$, on a clairement $\nu(p)=\nu(p')$ et $\nu(q)=\nu(q')$. Or $p+q=2(p'+q')$ donc $\nu(p+q)=\nu(p'+q')$. Le résultat s'en déduit par récurrence, puisque bien sûr on a autant de retenues dans le calcul de $p+q$ ou de $p'+q'$.
Si p est pair et q impair (ou le contraire), $p=2p'$ et $q=2q'+1$, les nombres de retenues dans les calculs de $p+q$ et de $p'+q'$ sont identiques. Ils valent $\nu(p')+\nu(q')-\nu(p'+q')$ par récurrence. Or $\nu(p')=\nu(p)$, $\nu(q')=\nu(q)-1$ et $\nu(p'+q')=\nu(p+q)-1$, d'où le résultat là encore.
Si $p=q=2^k-1$, avec $k\geq 1$, alors $\nu(p)=\nu(q)=k$, $p+q=2^{k+1}-2$ s'écrit comme p avec en plus un 0 à droite et $\nu(p+q)=\nu(p)=k$. Or dans le calcul de $p+q$ on aura autant de retenues que de bits, c'est-à-dire k, et le résultat est démontré.
Sinon c'est qu'en base 2 : $p=u1\overbrace{1\ldots1}^{k\text{ bits}}$ et $q=v0\overbrace{1\ldots1}^{k\text{ bits}}$, où on a choisi k le plus grand possible (ou la situation inverse, bien sûr).
Considérons alors le tableau suivant :

$$\begin{array}{r|cr|l}
\text{retenues} & & 1 & 1\ldots1 \\
p & u & 1 & 1\ldots11 \\
q & v & 0 & 1\ldots11 \\
\hline
p+q & \ldots & . & 1\ldots10
\end{array}$$

Notons p' l'entier qui s'écrit $u1$ en base 2, et q' celui qui s'écrit $v1$. On a alors $\nu(p')=\nu(p)-k$ et $\nu(q')=\nu(q)-k+1$. Or dans le calcul de $p+q$ on reportera un nombre de retenues égal à k plus le nombre de retenues nécessaires au calcul de $p'+q'$.
Ce nombre de retenues est donc, par récurrence, égal à

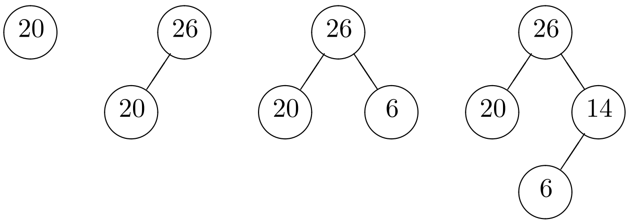
$$\begin{aligned}
k+\nu(p')+\nu(q')-\nu(p'+q') &= k+\nu(p)-k+\nu(q)-k+1-(\nu(p+q)-(k-1)) \\
&= \nu(p)+\nu(q)-\nu(p+q).
\end{aligned}$$

La démonstration est ainsi achevée.

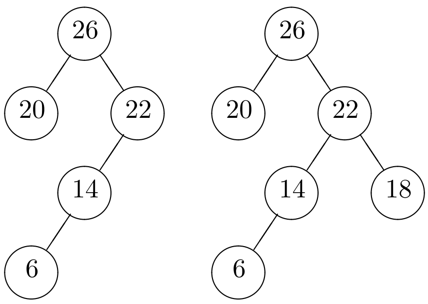
Pagodes

Insertion dans un arbre binaire tournoi

Un arbre binaire est dit *tournoi* si la clé de chaque nœud-fils est plus petite que celle de son père, de telle sorte que la racine porte la clé la plus grande. Autrement dit, en descendant de la racine vers une feuille les clés visitées sont en ordre décroissant.
Il existe différentes structures de données capables de gérer des arbres tournois. Mais nous imposerons dans la suite une condition supplémentaire : supposons qu'on insère successivement, dans une structure vide au départ, n clés entières $x_1, \ldots, x_n$. L'arbre tournoi résultat doit être tel que son parcours infixe redonne l'ordre d'insertion des clés.
Considérons par exemple les clés successives suivantes : 20, 26, 6, 14, 22, 18, 28, 21, 23, 25, 15, 17.
Voici ce qui se passe lors de l'insertion des quatre premières clés :



L'insertion des clés 22 puis 18 conduisent aux arbres suivants :



L'arbre obtenu après insertion des 12 clés proposées est dessiné dans la figure 4 page suivante.
La programmation Caml de cette insertion ne pose pas de réelle difficulté.
On définit le type des arbres utilisés de la façon suivante, puisqu'on décide de placer les informations aux nœuds et pas seulement aux feuilles :

```
type arbre = Vide | Nœud of int * arbre * arbre ;;
```

La fonction d'insertion s'écrit alors ainsi :

```
let rec insertion a x = match a with
    | Vide -> Nœud(x,Vide,Vide)
    | Nœud(r,_,_) when x >= r -> Nœud(x,a,Vide)
    | Nœud(r,g,Vide) -> Nœud(r,g,Nœud(x,Vide,Vide))
    | Nœud(r,g,(Nœud(r',_,_) as d)) when r' >= x -> Nœud(r,g,insertion d x)
    | Nœud(r,g,d) -> Nœud(r,g,Nœud(x,d,Vide)) ;;
```

On procèdera aux insertions de notre exemple par une instruction comme :

```
it_list insertion Vide [ 20 ; 26 ; 6 ; 14 ; 22 ; 18 ; 28 ; 21 ; 23 ; 25 ; 15 ; 17 ] ;;
```

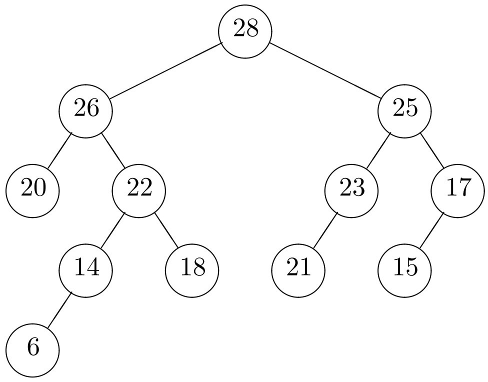


Figure 4: Insertion dans un arbre tournoi conservant l'historique

Suppression de l'élément maximal

On souhaite également supprimer l'élément maximal d'un arbre binaire tournoi : il suffit bien sûr de décapiter l'arbre. Tout le problème vient de ce que l'on exige toujours que le parcours symétrique (ou infixe) de l'arbre renvoie l'ordre d'insertion des différentes clés, le maximum étant ici bien entendu exclu.
Une solution récursive peut s'écrire ainsi :

```
let rec extrait_maximum = function
    | Vide -> failwith "arbre vide"
    | Nœud(m,Vide,Vide) -> m,Vide
    | Nœud(m,Vide,d) -> m,d
    | Nœud(m,g,Vide) -> m,g
    | Nœud(m,(Nœud(mg,gg,dg) as g),(Nœud(md,gd,dd) as d))
        -> if mg > md then
                let _,d' = extrait_maximum (Nœud(m,dg,d))
                in
                m,Nœud(mg,gg,d')
           else let _,g' = extrait_maximum (Nœud(m,g,gd))
                in
                m,Nœud(md,g',dd) ;;
```

Changement de point de vue

On considère maintenant les suites finies d'entiers (qu'on supposera deux à deux distincts en cas de besoin, pour simplifier, sans changer notablement la complexité du problème envisagé). La suite d'entiers $(x_0, x_1, \ldots, x_{n-1})$, de longueur n, pourra également être vue comme un mot sur le monoïde libre sur $\mathbb{N}$ (je crois, mais je ne suis qu'un piètre mathématicien) : $x_0 x_1 \ldots x_{n-1}$.
L'arbre binaire tournoi correspondant est l'arbre binaire dont la racine est $x_k = \max\{x_0, \ldots, x_{n-1}\}$, dont le sous-arbre gauche de la racine est l'arbre tournoi correspondant au mot $x_0 \ldots x_{k-1}$ et le sous-arbre droit au mot $x_{k+1} \ldots x_{n-1}$. Bien entendu, on associera au mot vide l'arbre vide. Ceci fournirait une nouvelle définition à nos arbres et aux opérations qu'on leur a associées.
De même, l'arbre de racine x_i est un arbre tournoi dont, récursivement, le fils gauche a pour racine $\max\{x_\ell, \ell < i\}$ et le fils droit $\max\{x_\ell, \ell > i\}$.

Définition des pagodes

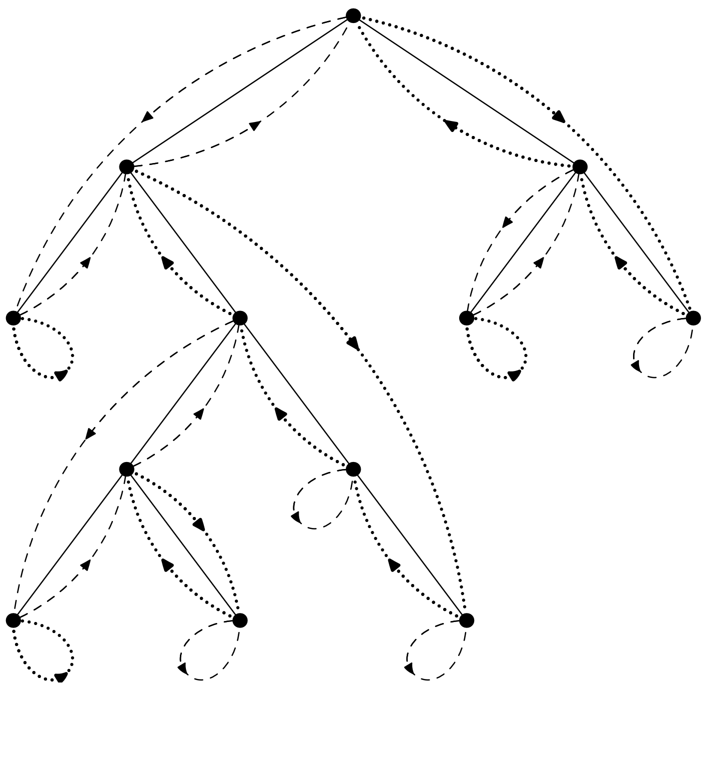
Jean Vuillemin, en collaboration avec F. Viennot et J. Françon, définit en 1978 une nouvelle structure de données, bien adaptée au problème que nous venons de décrire, et que Jean décide de baptiser *pagode*.
Il s'agit de définir deux nouveaux pointeurs pour chaque nœud de l'arbre : au lieu des pointeurs habituels sur les fils gauche et droit, il propose deux nouveaux pointeurs, que nous appellerons *rouge* et *bleu*, qui suffiront en réalité à reconstituer la structure de l'arbre complet.
Si l'arbre associé au mot $x_0 \ldots x_{n-1}$ est `Nœud(r,g,d)`, on a nécessairement à la fois $r = x_k = \max\{x_0, \ldots, x_{n-1}\}$, g est associé au mot $x_0 \ldots x_{k-1}$, d au mot $x_{k+1} \ldots x_{n-1}$.
Une définition des pagodes est la suivante :

- le pointeur rouge de la racine de l'arbre pointe sur le premier nœud de l'arbre dans son parcours infixe ;
- le pointeur bleu de la racine de l'arbre pointe sur le dernier nœud de l'arbre dans son parcours infixe ;
- le pointeur rouge d'un fils droit pointe sur le premier nœud du sous-arbre dont il est la racine, dans son parcours infixe ;
- le pointeur bleu d'un fils droit pointe sur son père ;
- le pointeur bleu d'un fils gauche pointe sur le dernier nœud du sous-arbre dont il est la racine, dans son parcours infixe ;
- le pointeur rouge d'un fils gauche pointe sur son père.

C'est dire en particulier que le pointeur rouge de la racine pointe sur x_0 et le bleu sur x_{n-1}.
Mais un dessin vaut souvent plus qu'un long discours, et la figure suivante montre le résultat sur un exemple. On a tracé les pointeurs bleu (*resp.* rouge) en traits pointillé (*resp.* tireté).
Les manipulations de pointeurs auxquelles nous serons contraints conduisent à définir le type des pagodes de la façon suivante :

```
type pagode = Néant | P of nœud_pagode
and nœud_pagode = { valeur : int ; mutable bleu : pagode ; mutable rouge : pagode } ;;

let rouge    (P p)   = p.rouge
and bleu     (P p)   = p.bleu
and rouge_à  (P p) x = p.rouge <- x
and bleu_à   (P p) x = p.bleu  <- x
and valeur   (P p)   = p.valeur ;;
```



Voici la fonction qui construit la pagode à partir de l'arbre sous-jacent :

```
let rec pagode_d'arbre = function
    | Vide -> Néant
    | Nœud(n,g,d)
        -> let p = crée_pagode n
           in
           if g <> Vide then
           begin
               let pg = pagode_d'arbre g
               in
               rouge_à p (rouge pg) ;
               rouge_à pg p
           end ;
           if d <> Vide then
           begin
               let pd = pagode_d'arbre d
               in
               bleu_à p (bleu pd) ;
               bleu_à pd p
           end ;
           p ;;
```

Cette fonction utilise `crée_pagode` qui se charge de la création d'une pagode contenant une unique valeur et qui peut s'écrire ainsi :

```
let crée_pagode x =
    let rec p = P { valeur = x ; bleu = p ; rouge = p }
    in
    p ;;
```

Nous avons systématiquement utilisé les fonctions d'accès en lecture `rouge`, `bleu`, `valeur` et en écriture `rouge_à`, `bleu_à`, pour simplifier et rendre plus lisibles les différents algorithmes sur les pagodes. On évite ainsi des simagrées du genre `match p with P pp -> pp.rouge` qui en outre déclenchent à chaque fois un *warning* du compilateur CAML.
Toutefois, le type étant récursif, l'affichage des valeurs du type `pagode` est assez peu explicite. C'est un des arguments qui expliquent l'utilité de la fonction réciproque, chargée de la construction de l'arbre binaire sous-jacent à une pagode et que l'on trouvera ci-dessous.

```
let rec arbre_de_pagode = function
    | Néant -> Vide
    | p -> let fils_gauche p =
               let rec remonte q = if rouge q = p then q else remonte (rouge q)
               in
               remonte (rouge p)
           and fils_droit p =
               let rec remonte q = if bleu  q = p then q else remonte (bleu q)
               in
               remonte (bleu p)
           in
           let g = let pg = fils_gauche p
                   in
                   if pg = p then Vide
                   else
                   (   rouge_à pg (rouge p) ;
                       let g = arbre_de_pagode pg
```

```
                in
                rouge_à pg p ;
                g   )
and d = let pd = fils_droit p
        in
        if pd = p then Vide
        else
        (   bleu_à pd (bleu p) ;
            let d = arbre_de_pagode pd
            in
            bleu_à pd p ;
            d   )
in
Nœud(valeur p,g,d) ;;
```

Opérations sur les pagodes

Là encore, les manipulations des pagodes reposent sur l'algorithme de fusion de deux pagodes. En effet, supposant définie la fonction `fusion`, on écrira l'insertion d'une nouvelle clé de la façon suivante :

```
let insertion p x = fusion p (crée_pagode x) ;;
```

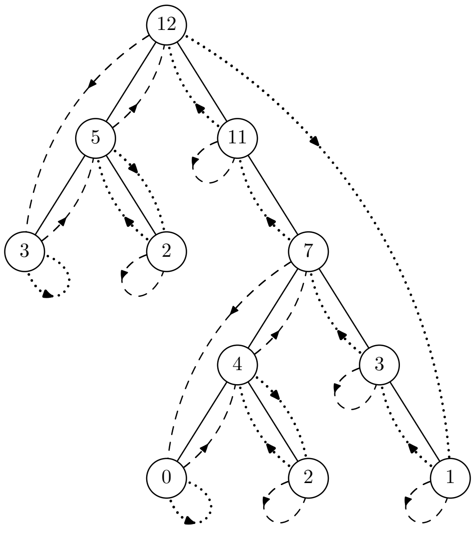
De même, on écrit la fonction qui permet de supprimer la racine d'une pagode, qui contient la valeur maximale.
Il s'agit simplement de la fusion des sous-arbres gauches et droits, il suffira de faire attention à mettre à jour les pointeurs rouge du fils gauche et bleu du fils droit avant de procéder à la fusion.

```
let suppression_maximum p =
    let fils_gauche p =
        let rec remonte q = if rouge q = p then q else remonte (rouge q)
        in
        remonte (rouge p)
    and fils_droit p =
        let rec remonte q = if bleu  q = p then q else remonte (bleu q)
        in
        remonte (bleu p)
    in
    let pg = fils_gauche p
    and pd = fils_droit p
    in
    if pg = p && pd = p then Néant
    else if pg = p then pd
    else if pd = p then pg
    else
    (   rouge_à pg (rouge p) ;
        bleu_à  pd (bleu  p) ;
        fusion pg pd         ) ;;
```

L'algorithme de fusion des pagodes

Quelques exemples

Commençons par considérer l'exemple de la pagode a de racine 12 ci-dessous et de la pagode b de racine 10 ci-après.



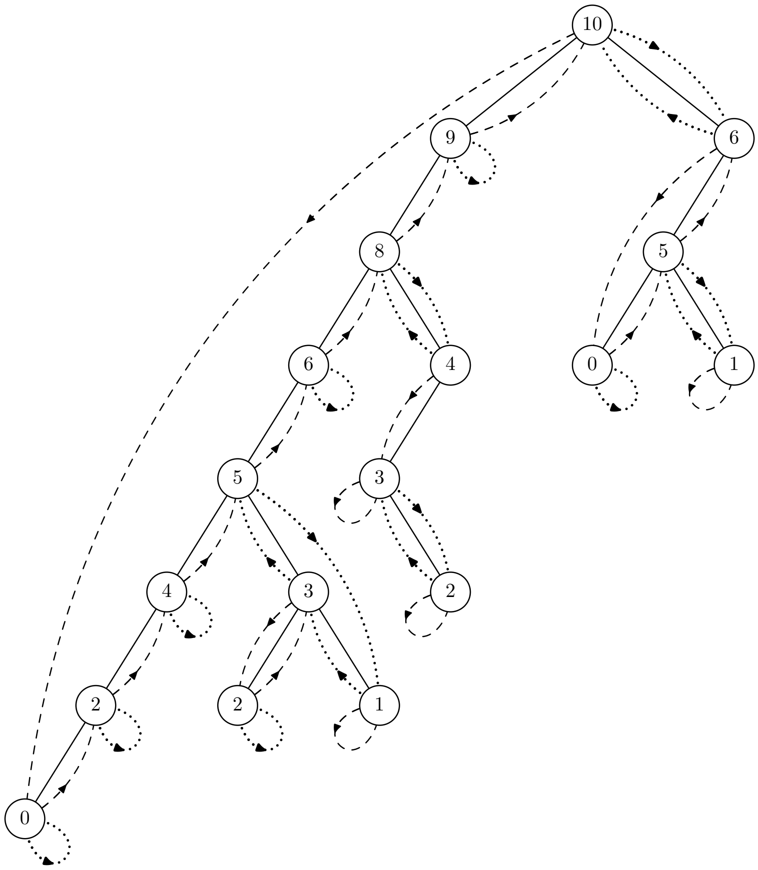
Dans les figures suivantes, on trouvera le résultats de `fusion b a` (en premier) et de `fusion a b` (en dernier).
On fera particulièrement attention aux nœuds de la branche supérieure droite de la première pagode et de la branche supérieure gauche de la seconde. Par exemple, dans la figure du bas, qui montre le résultat de la fusion de a et de b, dans cet ordre, on verra comment, le long du chemin tracé en trait épais, les nœuds 12, 11, 7, 3 et 1 de la pagode a s'intercalent avec les nœuds 10, 9, 8, 6, 5, 4, 2 et 0 de la pagode b. On remarquera également que les sous-arbres gauches des nœuds 12 et 7 de a sont conservés dans la pagode résultat, de même que les sous-arbres droits des nœuds 10, 8 et 5 de la pagode b. En fait seuls les nœuds du trajet en trait épais voient leurs pointeurs respectifs modifiés.

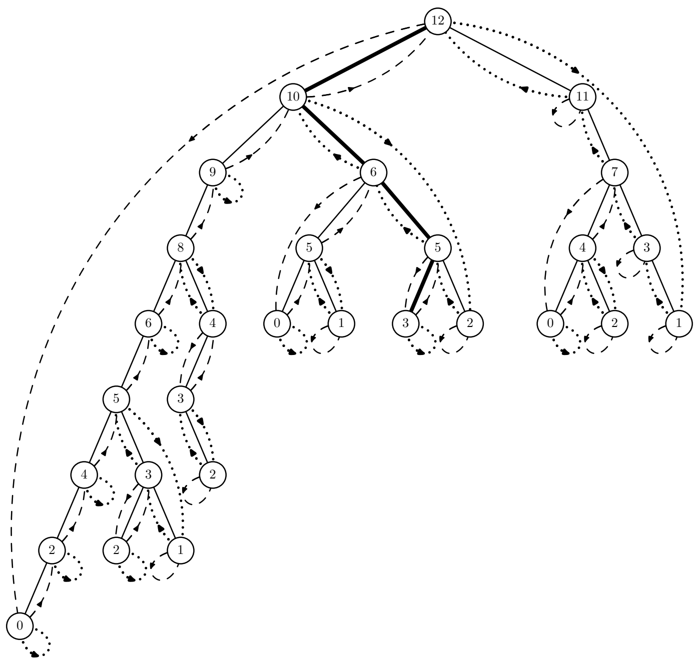
La fonction `fusion`

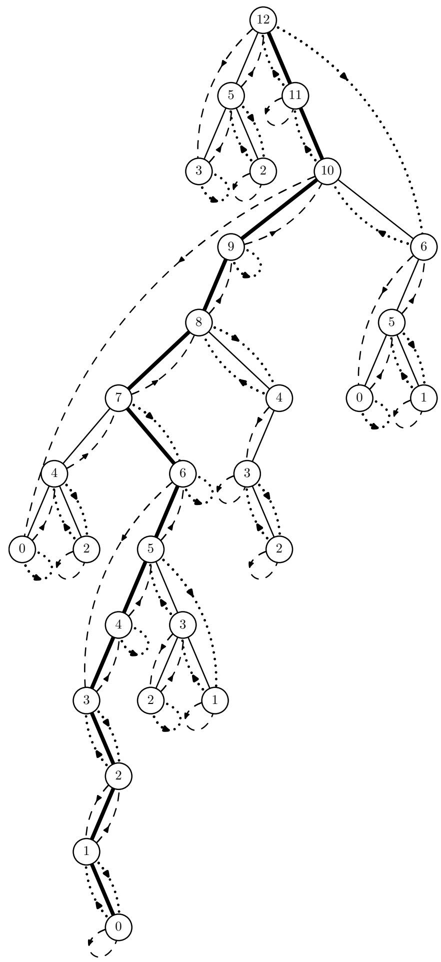
Reste à écrire effectivement l'algorithme de fusion de deux pagodes.
Il y a un cas très simple, où le dernier nœud (en ordre infixe) de la pagode a est plus grand que la racine de la pagode b. En effet dans ce cas il suffit d'écrire les quelques modifications de pointeurs suivantes :

```
let t = bleu a
in
bleu_à a (bleu b) ;
bleu_à b t ;
a ;;
```

Le cas général se traite en gérant comme il faut le parcours par un nœud courant t du chemin que nous avons tracé en trait épais dans les exemples ci-dessus.


10
9
6
8
5
6
4
0
1
5
3
4
3
2
2
2
1
0


12
10
11
9
6
7
8
5
5
4
3
6
4
0
1
3
2
0
2
1
5
3
4
3
2
2
2
1
0


12
5
11
3
2
10
9
6
8
5
7
4
0
1
4
6
3
0
2
5
2
4
3
3
2
1
2
1
0

Avec quelques efforts on obtient le programme suivant.

```
let fusion a b = match a,b with
    | Néant,b -> b
    | a,Néant -> a
    | a,b
        ->  let rec parcours triplet = match triplet with
                | (_,Néant,_) -> triplet
                | (Néant,_,_) -> triplet
                | (a',b',r)
                    ->  if (valeur a') < (valeur b') then
                            let t = bleu  a'
                            in  bleu_à  a' (bleu  r) ; bleu_à  r a' ;
                                parcours (t,b',a')
                        else
                            let t = rouge b'
                            in  rouge_à b' (rouge r) ; rouge_à r b' ;
                                parcours (a',t,b')
            in
            let a' = bleu a and b' = rouge b
            in
            bleu_à a Néant ; rouge_à b Néant ;
            let a',b',r =
                if valeur a' < valeur b' then
                    let r = bleu  a'
                    in bleu_à  a' a' ; (r,b',a')
                else
                    let r = rouge b'
                    in rouge_à b' b' ; (a',r,b')
            in
            match parcours (a',b',r) with
                | (a',Néant,r) -> ( bleu_à  a (bleu  r) ;
                                    bleu_à  r a' ;
                                    a   )
                | (Néant,b',r) -> ( rouge_à b (rouge r) ;
                                    rouge_à r b' ;
                                    b ) ;;
```

Évaluation

Nous donnons ici sans démonstration les résultats de l'évaluation des coûts moyens des opérations de base sur les pagodes.

Le coût moyen d'une adjonction d'une nouvelle clé dans une pagode de taille n vaut $CA_n = 2\left(1 - \dfrac{1}{n+1}\right)$; le coût moyen de la suppression du maximum vaut $CS_n = 2\left(H_n - 2 + \dfrac{1}{n}\right)$; nous avons bien sûr noté H_n le n-ième nombre harmonique, $H_n = 1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{n}$.

On a donc, asymptotiquement (mais rappelons qu'il s'agit là de valeurs moyennes) :

$$CA_n = O(1), \qquad \text{et} \qquad CS_n = O(\lg n).$$

Un algorithme pour dessiner de beaux arbres n-aires

Le problème qui nous intéresse ici peut se définir ainsi : on considère un arbre n-aire —le type Caml correspondant pourrait se définir par la déclaration suivante :
```
type 'a arbre = Nœud of 'a * 'a arbre list ;;
```
— et on aimerait écrire un algorithme de calcul de la position dans le plan euclidien de chacun des nœuds de l'arbre de telle sorte que le dessin qui en résulte soit *joli.*

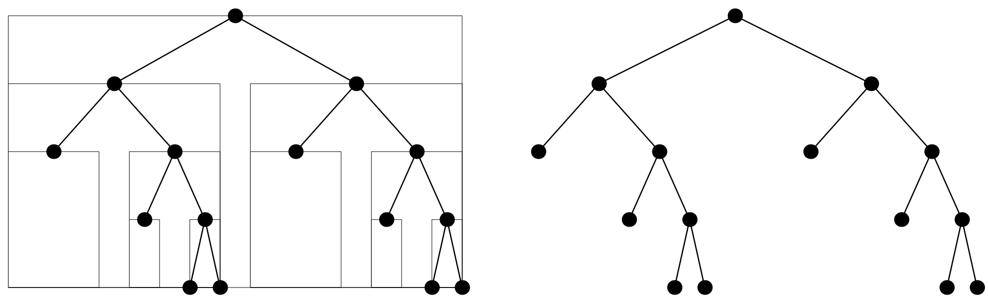
Une grande partie de la difficulté tient à la définition précise d'un *joli arbre.* Dans tous les cas on souhaitera bien entendu que la géométrie de l'arbre corresponde à la géométrie de la figure, et qu'il n'y ait pas croisement de branches… Il est clair également que deux nœuds de même profondeur devront avoir la même ordonnée, et on imposera une différence constante dy des ordonnées de deux niveaux consécutifs. On introduit enfin naturellement la distance minimale η qui devra séparer les abscisses de deux nœuds voisins. On choisira en pratique dy et η de telle sorte que les proportions de l'arbre soient flatteuses pour l'œil. η devra être suffisant pour séparer des nœuds qui contiendront des objets éventuellement encombrants. Nous ne nous intéresserons pas à ce problème (d'importance mineure, et qu'on pourrait faire facilement intervenir au prix de modifications limitées de nos algorithmes). Dans les exemples qui suivent, nous représenterons chaque nœud par un gros point noir, et c'est tout.

Une solution naïve

Une première solution consiste à utiliser une procédure récursive. Pour tracer un arbre de profondeur k dont les nœuds sont tous de degré d, on tracera chacun des fils dans une boîte de même largeur $\ell(k-1)$, et on aura la relation $\ell(k) = (d-1)\eta + d\ell(k-1)$ avec la condition initiale $\ell(0) = 0$. Ceci fournit aisément $\ell(k) = \eta(d^k - 1)$.

Bien sûr, c'est là une solution très maladroite. Le moins qu'on puisse faire pour l'améliorer serait à chaque niveau de remplacer d par le degré maximal du niveau considéré.

Mieux encore, on gère séparément le degré de chaque nœud. Pour des raisons évidentes d'esthétique, on impose d'une part que l'abscisse d'un nœud père soit la moyenne arithmétique des abscisses de ses nœuds fils, et d'autre part que les fils d'un même nœud soient équidistants (essayez sans cette condition et vous verrez comme cela paraît laid…) Cela revient à inscrire chaque sous-arbre dans un rectangle et à garantir un intervalle minimal η entre deux rectangles, les fils d'un même père étant équidistants.



Ceci ne correspond évidemment pas à la solution qu'on pourrait espérer:

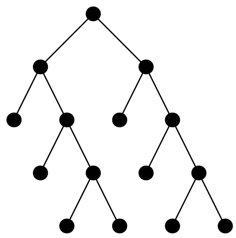


Figure 5: Un plus joli dessin du même arbre

On programme assez facilement le calcul correspondant en CAML. Voici ce que je propose :

```
let rec mesure = function
    | Nœud(r,[]) -> Nœud((r,0),[])
    | Nœud(r,fils)
        ->  let fils' = map mesure fils
            in
            let largeur = it_list (fun x (Nœud((_,l),_)) -> max x l) 0 fils'
            in
            Nœud((r,largeur * (list_length fils) + (list_length fils) - 1),fils') ;;

let rec place x = function
    | Nœud((r,largeur),[ ] ) -> Nœud((r,x),[])
    | Nœud((r,largeur),[ b ]) -> Nœud((r,x),[ place x b ])
    | Nœud((r,largeur),fils)
        -> let n = list_length fils
           and l = float_of_int largeur
           in
           let dx = (l +. 1.0) /. (float_of_int n)
           in
           let x1 = x -. (float_of_int (n - 1)) *. dx /. 2.0
           in
           let rec aux x = function
               | [] -> []
               | a :: q -> (place x a) :: (aux (x +. dx) q)
           in
           Nœud((r,x),(aux x1 fils)) ;;

let x_ifie arbre = place 0.0 (mesure arbre) ;;
```

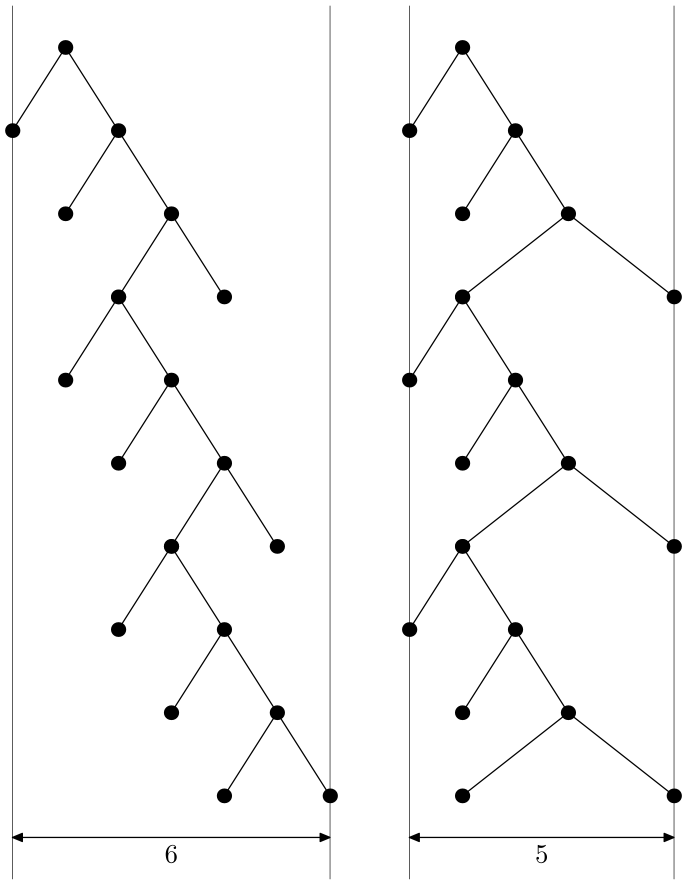
On a ainsi défini une fonction `x_ifie : 'a arbre -> ('a * float) arbre`, qui renvoie un arbre dont les valeurs des clés sont augmentées par l'abscisse du nœud (la racine étant d'abscisse nulle).

Tout n'est pas si simple...

Il paraît raisonnable de préciser les règles suivantes dans le tracé des arbres :

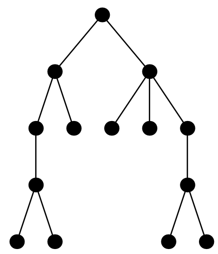
1. la racine d'un arbre doit être centrée au-dessus de ses fils ;
2. le tracé doit occuper — en largeur — une place minimale, à condition de ne violer aucune règle précédente ;
3. deux sous-arbres identiques d'un même arbre doivent être tracés de la même façon partout où ils figurent dans l'arbre.

Mine de rien, mine de crayon, tout cela n'est pas évident. L'algorithme de Reingold et Tilford (*Tidier drawings of tree.* IEEE Transactions on Software Engineering, 7(2), mars 1981), permet de vérifier toutes les règles de base ainsi que les règles 1 et 2 ci-dessus. Cependant l'exemple de la figure suivante montre, à gauche, ce que produit l'algorithme de Reingold et Tilford, et, à droite, le même arbre mais vérifiant toutes les règles que nous nous sommes imposées (y compris la règle 3).

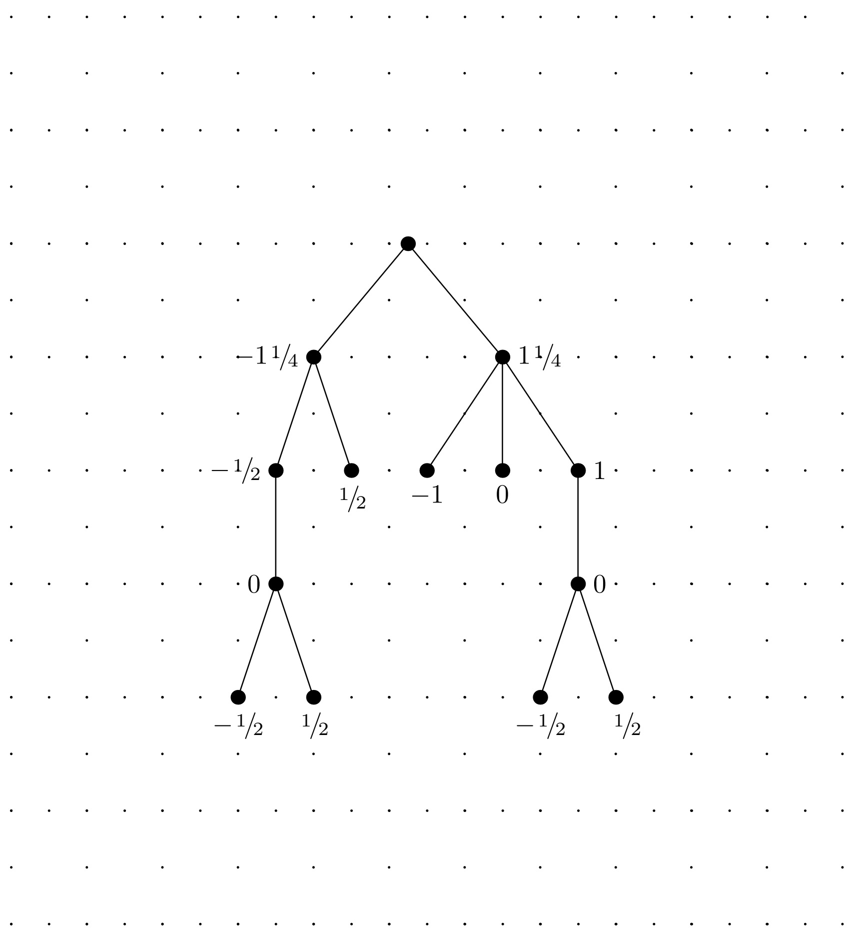


L'algorithme de nos préférences

Explicitons maintenant, sans faire attendre davantage le lecteur, l'algorithme de nos préférences, qui a produit tous les dessins d'arbres qui précèdent dans ce numéro de *La Lettre*, et qui, je l'espère, lui auront plu. Parce qu'enfin, j'ose espérer que le lecteur se sera rendu compte du travail qui a permis de produire les figures précédentes ! ?
Pour tout arbre, comme le suivant :

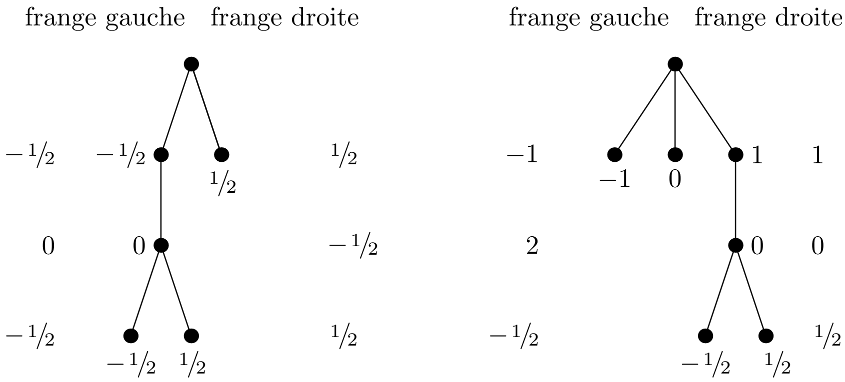


choisissons de représenter l'abscisse de chaque nœud par son écart relatif à sa racine, ce que décrit la figure suivante.



On a également représenté sur la figure une grille qui permet plus facilement de mesurer les

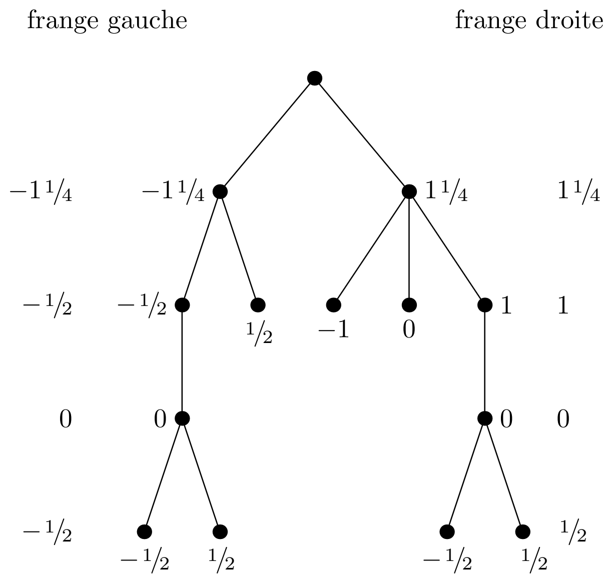
demi-unités sur chacun des axes.
Tout le problème est bien évidemment dans le calcul des abscisses pertinentes de chacun des nœuds.
Pour assurer la condition 3 ci-dessus, on veut écrire un algorithme récursif.
On convient d'associer (récursivement dans l'algorithme de tracé) à chaque sous-arbre ce que nous avons baptisé les franges gauche et droite, constituées des écarts relatifs en abscisse des nœuds les plus à gauche et à droite de chaque niveau.
Pour l'arbre précédent, on part des sous-arbres suivants :



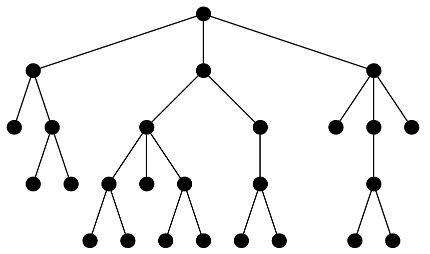
On descend alors de la racine aux feuilles, en commençant par une séparation égale à 1. Voici le petit tableau qu'on construit au fur et à mesure qu'on descend dans l'arbre.

niveau	a	δ
0		1
1	$1 - 1/2 + (-1) = -1/2 < 1$	$1 + (1 - (-1/2)) = 5/2$
2	$1 - (-1/2) + 2 = 7/2 \geq 1$	$5/2$
3	$7/2 - 1/2 + (-1/2) = 5/2 \geq 1$	$5/2$

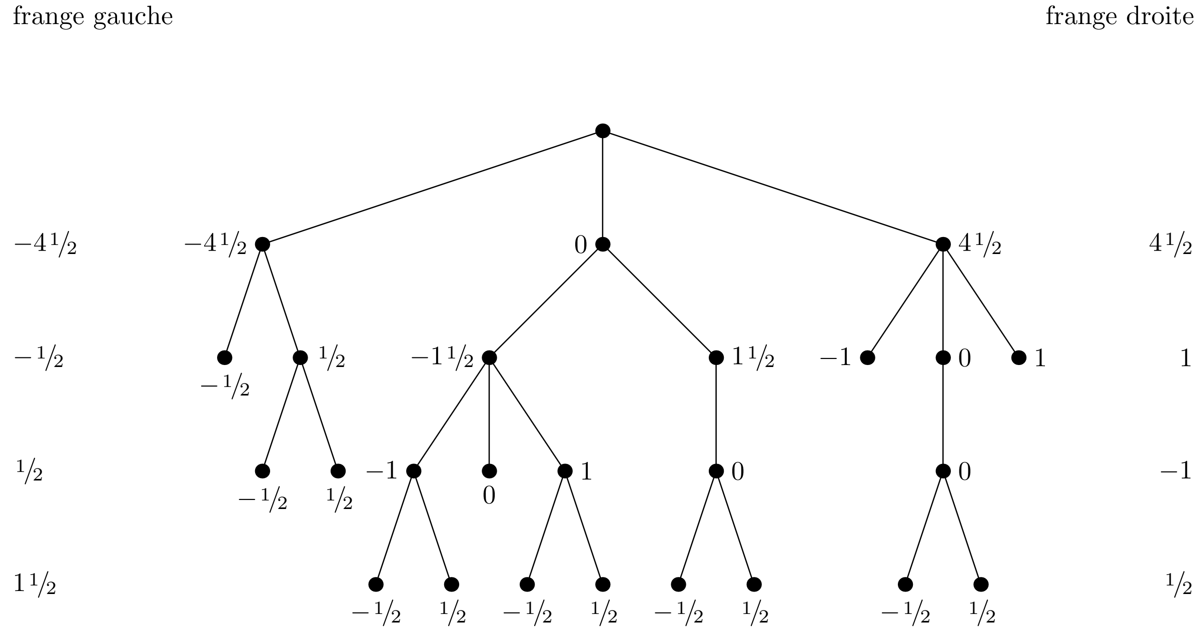
On commence avec $\delta = 1$ (l'unité étant le η dont nous avons parlé plus haut). Au niveau 1, on calcule a en ajoutant à δ la frange gauche de l'arbre droit moins la frange droite de l'arbre gauche, ce qui représente la distance dont se sont rapprochés les sous-arbres. Comme ici a devient égal à $-1/2$ qui est plus petit que 1, distance minimale, il faut modifier δ en lui ajoutant justement cette distance $(1 - 1/2)$. Le nouveau a (qu'on a pas fait figurer dans le tableau) vaudrait alors 1.
Au niveau 2, le nouveau a est plus grand que 1, et il n'est donc pas nécessaire d'augmenter δ. De même au niveau 3.
Finalement, l'écart à insérer entre les sous-arbres sera $5/2$, et on obtient ainsi le schéma de la page suivante.
On aura noté facilement comment les franges gauche et droite de l'arbre final se déterminent à partir de celles des sous-arbres de départ.


frange gauche
frange droite
−1 1/4
−1 1/4
1 1/4
1 1/4
− 1/2
− 1/2
1/2
−1
0
1
1
0
0
0
0
− 1/2
− 1/2
1/2
− 1/2
1/2
1/2

Prenons un deuxième exemple en considérant l'arbre suivant.



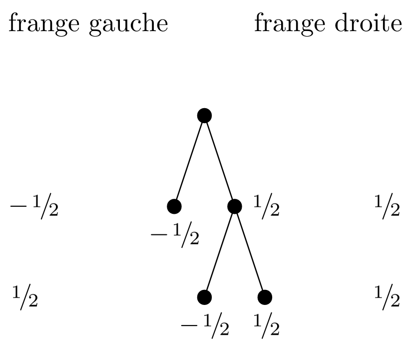
La figure de la page suivante montre les franges des sous-arbres de notre arbre complet, représenté ci-dessous.

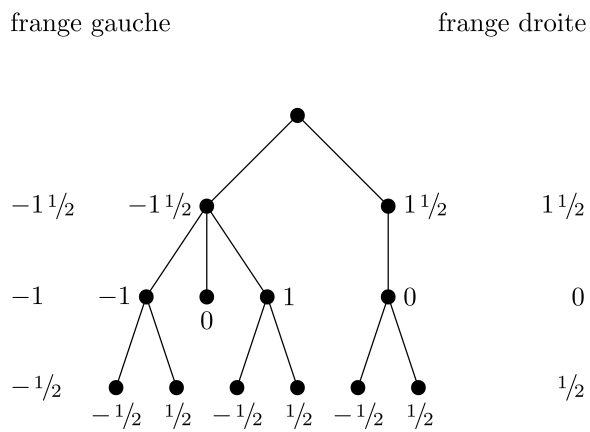


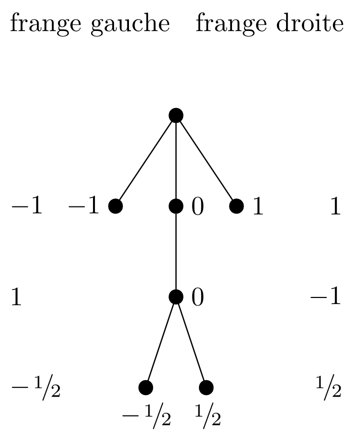
On dresse un tableau analogue au précédent :

niveau	a	b	δ	a'	b'
0			1	1	1
1	$-1 < 1$	$-3/2 < 1$	$7/2$	$3/2$	1
2	$= 0 < 1$	$2 \geq 1$	$9/2$	1	2
3	—	$1 \geq 1$	$9/2$	—	1

L'écart final est bien de 9/2 unités.


frange gauche
frange droite
−½
−½
½
½
½
−½
½
½


frange gauche
frange droite
−1½
−1½
1½
1½
−1
−1
0
1
0
0
−½
−½
½
−½
½
−½
½
½


frange gauche
frange droite
−1
−1
0
1
1
1
0
−1
−½
−½
½
½