Chapitre 1

option informatique

Corrigé des exercices

• Arbres binaires

Exercice 1 La première solution qui vient à l'esprit est sans doute celle-ci :

```
let rec profondeur p = function
  | Nil                -> []
  | a when p = 0       -> [a]
  | Noeud (fg, _, fd)  -> (profondeur (p-1) fg)@(profondeur (p-1) fd) ;;
```

mais elle n'est pas optimale, car il faut se souvenir que la concaténation de deux listes effectue un nombre d'insertions en tête de liste égal à la longueur de la liste de gauche. Dans le cas d'un arbre complet de hauteur p, le nombre t_p d'insertions effectuées vérifie la relation :

$$t_p = 2t_{p-1} + 2^{p-1}.$$

Celle-ci se résout en sommant l'égalité télescopique : $\frac{t_p}{2^p} - \frac{t_{p-1}}{2^{p-1}} = \frac{1}{2}$, soit : $t_p = p2^{p-1}$. Ainsi, dans le cas d'un arbre binaire complet le coût de cette fonction est un $\Theta(n\log n)$ avec $n = |\mathrm{A}| = 2^{p+1} - 1$.
On peut faire mieux en utilisant un accumulateur :

```
let profondeur =
  let rec aux acc p = function
    | Nil               -> raise Not_found
    | a when p = 0      -> a::acc
    | Noeud (fg, _, fd) -> aux (aux acc (p-1) fd) (p-1) fg
  in aux [] ;;
```

Avec cette fonction chaque arbre n'est inséré qu'une fois en tête de liste, ce qui est optimal : dans le cas de l'arbre binaire complet le nombre d'insertion est égal à 2^p, le coût est un $\Theta(n)$.

Exercice 2 Le calcul de la hauteur d'un arbre est linéaire vis-à-vis de la taille de l'arbre, donc lorsque A est un arbre binaire complet de taille n le coût t_n de cette fonction vérifie la relation $t_n = 2t_{\lfloor n/2\rfloor} + \Theta(n)$. D'après le théorème maître, $t_n = \Theta(n\log n)$.
Le cas d'un arbre incomplet est plus délicat à étudier à cause de l'évaluation paresseuse, mais on peut au moins donner une majoration du coût : $t_n \leqslant t_p + t_q + \Theta(n)$ avec $p + q = n - 1$ ce qui permet d'obtenir $t_n = \mathrm{O}(n^2)$ dans le cas général.

De toute façon on peut faire mieux en utilisant une fonction auxiliaire qui calcule la hauteur en même temps qu'elle détermine si l'arbre est complet ou non :

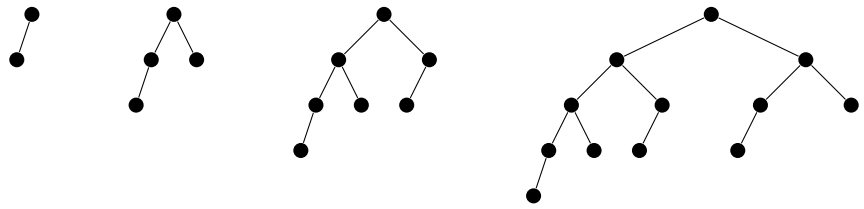
```
let est_complet a =
  let rec aux = function
    | Nil -> true, -1
    | Noeud (fg, _, fd) -> let (b1, h1) = aux fg and (b2, h2) = aux fd in
                           (b1 && b2 && h1 = h2, 1 + max h1 h2)
  in fst (aux a) ;;
```

Le coût de cette fonction vérifie cette fois la relation : $t_n = t_p + t_q + \Theta(1)$ avec $p + q = n - 1$, ce qui conduit à $t_n = \Theta(n)$; le coût est linéaire dans tous les cas.

Exercice 3 On génère un arbre complet de taille n (s'il en existe) à l'aide de la fonction :

```
let rec complet = function
  | 0                  -> Nil
  | n when n mod 2 = 0 -> failwith "complet"
  | n                  -> let x = complet ((n-1)/2) in Noeud (x, x) ;;
```

Exercice 4 Les squelettes des arbres de Fibonacci d'ordre 2, 3, 4 et 5 sont dessinés ci-dessous :



Si f_p désigne le nombre de feuilles d'un arbre de Fibonacci d'ordre p alors :

$$f_0 = 0, \quad f_1 = 1, \quad \text{et} \quad \forall p \geqslant 2,\ f_p = f_{p-1} + f_{p-2}$$

donc f_p est égal au p^e nombre de Fibonacci.
Commençons par montrer que la hauteur d'un arbre de Fibonacci d'ordre p est égal à $p-1$.

- C'est clair si $p = 0$ ou $p = 1$.
- Si $p \geqslant 2$, supposons le résultat acquis aux rangs $p-1$ et $p-2$, et considérons un arbre de Fibonacci A d'ordre p. Par construction, $A = 1 + \max(p-2, p-3) = p-1$, ce qui prouve le résultat au rang p.

Montrons maintenant par récurrence sur $p \in \mathbb{N}$ que tout nœud interne d'un arbre de Fibonacci d'ordre p a un déséquilibre égal à 1.

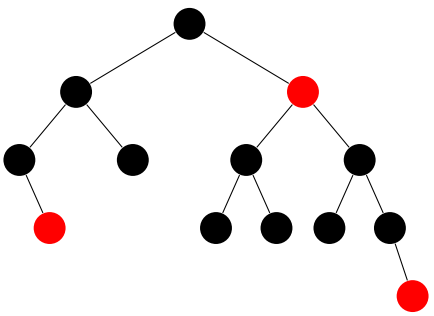
- C'est clair pour $p = 0$ et $p = 1$ puisqu'il n'y a pas de nœud interne dans ces deux arbres.
- Si $p \geqslant 2$, supposons le résultat acquis aux rangs $p-1$ et $p-2$. Par construction, tout interne nœud appartenant au fils gauche ou au fils droit a un déséquilibre égal à 1. Il reste à examiner le cas de la racine. Or nous venons de prouver que son fils gauche a une hauteur égale à $p-2$ et son fils droit à $p-3$ donc son déséquilibre est égal à $(p-2)-(p-3) = 1$, ce qui achève de prouver le résultat annoncé.

Exercice 5 Le principe est de calculer en même temps le déséquilibre et la hauteur de chacun des sous-arbres qui composent l'arbre à tester. La démarche est très semblable à celle suivie dans l'exercice 1.

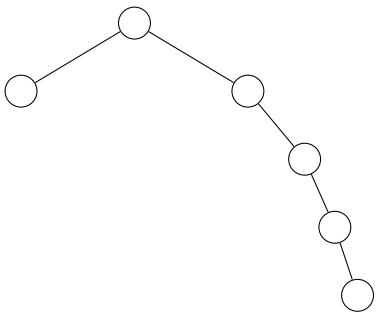
```
let avl a =
  let rec aux = function
    | Nil               -> (true, -1)
    | Noeud (fg, _, fd) -> let (b1, h1) = aux fg and (b2, h2) = aux fd in
                           (b1 && b2 && (h1 = h2 || h1 = h2-1 || h1 = h2+1),
                            1 + max h1 h2)
  in fst (aux a) ;;
```

Exercice 6

a) Le coloriage ci-dessous convient :



b) L'arbre ci-contre ne peut avoir de coloration rouge-noir, car au moins trois nœuds du fils droit doivent être coloriés en rouge pour respecter la troisième condition, et ceci est impossible sans contrevenir à la deuxième condition.



c) Il existe au moins un nœud à la profondeur $h(A)$; le chemin qui le conduit à la racine comporte $h(A)+1$ nœuds dont $b(A)$ nœuds noirs et $h(A)+1-b(A)$ nœuds rouges. Ceci prouve déjà que $b(A) \leqslant h(A)+1$.

Par ailleurs, sachant que la racine est noire est que tout parent d'un nœud rouge est noir, il y a aussi au moins $h(A)+1-b(A)$ nœuds noirs sur ce trajet, ce qui prouve l'inégalité : $h(A)+1-b(A) \leqslant b(A) \iff h(A)+1 \leqslant 2b(A)$.

Raisonnons maintenant par induction structurelle pour prouver la seconde inégalité.

- Si $A = \mathsf{nil}$ alors $b(A) = 0$ et $|A| = 0$.
- Si $A = (F_g, x, F_d)$, A possède deux fils (éventuellement vides).
  Un fils noir ou vide F est la racine d'un arbre rouge-noir pour lequel $b(F) = b(A) - 1$. Dans ce cas, $|F| \geqslant 2^{b(A)-1} - 1$.
  Un fils rouge F ne peut avoir que des fils noirs ; il en a deux (éventuellement vides), qui sont les racines d'arbres rouge-noir A′ pour lesquels $b(A') = b(A) - 1$. Dans ce cas, $|F| \geqslant 2(2^{b(A)-1} - 1) + 1 = 2^{b(A)} - 1$.
  Sachant que $|A| = |F_g| + |F_d| + 1$ on en déduit que $|A| \geqslant 2(2^{b(A)-1} - 1) + 1 = 2^{b(A)} - 1$.

Cette dernière inégalité peut aussi s'écrire $b(A) \leqslant \log(|A|+1)$ ce qui implique : $h(A) \leqslant 2\log(|A|+1) - 1$ et donc $h(A) = O(\log|A|)$; un arbre rouge-noir est équilibré.

d) Plutôt que de vérifier que le père de chaque nœud rouge est noir, on vérifie plutôt que chaque nœud rouge n'a pas de fils rouge à l'aide de la fonction :

```
let rec couleur_fils = function
  | Nil                                  -> true
  | Noeud (Noeud (_, Rouge, _), Rouge, _) -> false
  | Noeud (_, Rouge, Noeud (_, Rouge, _)) -> false
  | Noeud (fg, _, fd)                    -> couleur_fils fg && couleur_fils fd ;;
```

La fonction suivante a pour objet de vérifier que le nombre de nœuds noirs entre un arbre vide et la racine est constant :

```
let hauteur_noir a =
  let rec aux = function
    | Nil -> (true, 0)
    | Noeud (Nil, Noir, fd)  -> let (b, h) = aux fd in (b, h + 1)
    | Noeud (fg, Noir, Nil)  -> let (b, h) = aux fg in (b, h + 1)
    | Noeud (fg, Noir, fd)   -> let (b1, h1) = aux fg and (b2,h2) = aux fd in
                                (b1 && b2 && h1 = h2, h1 + 1)
    | Noeud (Nil, Rouge, fd) -> aux fd
    | Noeud (fg, Rouge, Nil) -> aux fg
    | Noeud (fg, Rouge, fd)  -> let (b1, h1) = aux fg and (b2,h2) = aux fd in
                                (b1 && b2 && h1 = h2, h1)
  in fst (aux a) ;;
```

Il reste à écrire la fonction principale :

```
let rouge_noir a = match a with
  | Nil             -> false
  | Noeud (_, c, _) -> c = Noir && couleur_fils a && hauteur_noir a ;;
```

• Arbres binaires de recherche

Exercice 7 Plusieurs solutions sont possibles, l'une d'entre-elles consiste à vérifier que le parcours infixe retourne les clés par ordre croissant.

```
let rec est_decroissante = function
  | [] | [_] -> true
  | a::b::q  -> a >= b && est_decroissante (b::q) ;;

let est_abr a =
  let rec aux acc = function
    | Nil               -> acc
    | Noeud (fg, x, fd) -> aux (x.Key::(aux acc fg)) fd
  in est_decroissante (aux [] a) ;;
```

La fonction **aux** calcule l'image miroir de la liste des clés suivant le parcours infixe, la fonction **est_decroissante** vérifie que cette liste est décroissante, le tout en coût linéaire vis-à-vis de |A|.

Une autre solution consiste à utiliser une fonction auxiliaire qui détermine si un arbre est un ABR et renvoie en plus les valeurs minimale et maximale des clés présentes :

```
exception Arbre_vide ;;

let est_abr a =
  let rec aux = function
    |Nil                -> raise Arbre_vide
    |Noeud (Nil, x, Nil) -> (true, (x, x))
    |Noeud (Nil, x, d)   -> let (b, (u, v)) = aux d in (b && x <= u, (x, v))
    |Noeud (g, x, Nil)   -> let (b, (u, v)) = aux g in (b &&  x >=v, (u, x))
    |Noeud (g, x, d)     -> let (b1, (u1, v1)) = aux g and b2, (u2, v2) = aux d
                            in (b1 && b2 && x <= v2 && x >= u1, (u1, v2))
  in try fst (aux a) with Arbre_vide -> true ;;
```

Exercice 8 Nous avons besoin d'une fonction d'insertion dans un ABR, par exemple au niveau des feuilles :

```
let rec insere y = function
  | Nil                          -> Noeud (Nil, y, Nil)
  | Noeud (fg, x, fd) when x < y -> Noeud (fg, x, insere y fd)
  | Noeud (fg, x, fd)            -> Noeud (insere y fg, x, fd) ;;
```

ainsi que d'une fonction qui crée un ABR à partir des éléments d'un tableau :

```
let rec creer_abr t =
  let rec aux = function
    | i when i = vect_length t -> Nil
    | i                        -> insere t.(i) (aux (i+1))
  in aux 0 ;;
```

La fonction principale réalise alors un parcours infixe, le traitement consistant à placer l'élément rencontré dans le tableau (la référence utilisée marque le rang de la prochaine case à remplir).

```
let tri_abr t =
  let k = ref 0 in
  let rec aux = function
    | Nil               -> ()
    | Noeud (fg, x, fd) -> aux fg ;
                           t.(!k) <- x ; k := !k + 1 ;
                                     aux fd
  in aux (creer_abr t) ;;
```

Dans le meilleur des cas l'arbre obtenu est équilibré et le coût de sa création est un $O(n \log n)$ tandis que le coût du parcours infixe est un $\Theta(n)$. Le coût total est donc un $O(n \log n)$.
Dans le pire des cas l'arbre obtenu est un arbre peigne et le coût de sa création est un $\Theta(n^2)$. Le coût du parcours infixe est toujours un $\Theta(n)$ donc le coût total est un $\Theta(n^2)$.

Exercice 9 Une solution consiste à effectuer un parcours (infixe par exemple) de A_2 en insérant chacun des éléments dans A_1. Cette solution a un coût important : $O(|A_2| \times h(A_1))$ voire plus en cas de déséquilibre.
La solution que nous allons réaliser consiste, si $A_2 = (F_g, x, F_d)$, à procéder récursivement en insérant x à la racine de A_1 puis à fusionner F_g avec le sous-arbre gauche obtenu et F_d avec le sous-arbre droit.
On commence par rédiger la fonction suivante, déjà utilisée dans l'algorithme d'insertion à la racine, qui sépare un arbre binaire de recherche en deux :

```
let rec partition v = function
  | Nil                              -> Nil, Nil
  | Noeud (fg, x, fd) when x.Key < v -> let a1, a2 = partition v fd in
                                        Noeud (fg, x, a1), a2
  | Noeud (fg, x, fd)                -> let a1, a2 = partition v fg in
                                        a1, Noeud (a2, x, fd) ;;
```

La fusion s'écrit alors :

```
let rec fusion a1 a2 = match (a1, a2) with
  | _, Nil               -> a1
  | Nil, _               -> a2
  | _, Noeud (g2, x, d2) -> let g1, d1 = partition x.Key a1 in
                            Noeud (fusion g1 g2, x, fusion d1 d2) ;;
```

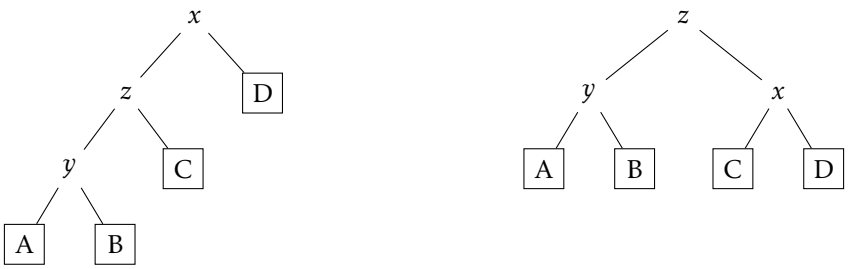
Exercice 10

a) Considérons une rotation droite et notons a, b et c des éléments respectifs des ABR A, B et C.
Par hypothèse, $a \leqslant x \leqslant b$ et $b \leqslant y \leqslant c$.
Ces inégalités peuvent s'écrire de manière équivalente : $b \leqslant y \leqslant c$ et $a \leqslant x \leqslant b$, ce qui traduit que l'arbre obtenu par rotation droite et toujours un ABR.
Il en est bien sur de même d'une rotation gauche.
Les deux fonctions de rotations se réalisent à coût constant en écrivant :

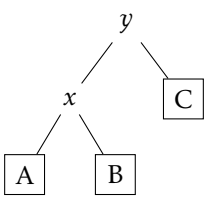
```
let rotd = function
  | Noeud (Noeud (a, x, b), y, c) -> Noeud (a, x, Noeud (b, y ,c))
  | _                             -> failwith "rotd" ;;

let rotg = function
  | Noeud (a, x, Noeud (b, y ,c)) -> Noeud (Noeud (a, x, b), y, c)
  | _                             -> failwith "rotg" ;;
```

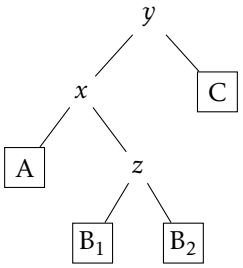
b) La rotation gauche autour de y conduit à l'arbre de gauche ; suivie de la rotation droite autour de x on obtient l'arbre indiqué à droite.



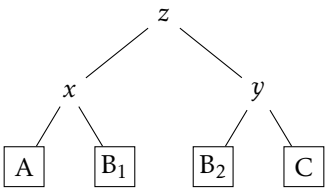
c) Le déséquilibre initial de y est égal à -1, 0 ou 1, et l'apparition d'une feuille parmi ses descendants ne peut modifier son déséquilibre que d'au plus une unité, donc $\text{eq}(y) = \pm 2$.
Si on suppose $\text{eq}(y) = 2$, la situation peut être représentée ci-dessous :



avec $h(C) = \max(h(A), h(B)) - 1$.
Si $\text{eq}(x) = 0$ alors $h(A) = h(B)$ et $h(C) = h(B) - 1$, et la rotation droite autour de y conduit aux nouvelles valeurs du déséquilibre : $\text{eq}(x) = h(A) - (h(B) + 1) = -1$ et $\text{eq}(y) = h(B) - h(C) = 1$ donc la condition AVL est de nouveau respectée.
Si $\text{eq}(x) = 1$ alors $h(A) = h(B) + 1$ $h(C) = h(B)$, et la rotation droite autour de y conduit aux nouvelles valeurs du déséquilibre : $\text{eq}(x) = h(A) - (h(B) + 1) = 0$ et $\text{eq}(y) = h(B) - h(C) = 0$ donc la condition AVL est de nouveau respectée.
Si $\text{eq}(x) = -1$ alors $h(B) = h(A) + 1$ et $h(C) = h(A)$. Puisque $h(B) \geqslant 0$, B n'est pas l'arbre vide ; notons z sa racine, B_1 et B_2 ses fils gauche et droit. La situation est alors la suivante :



avec $\max(h(B_1), h(B_2)) = h(A) = h(C)$. La question précédente a montré qu'en deux rotations il était possible d'obtenir l'arbre ci-dessous :



z étant équilibré on a $h(B_1) - h(B_2) = -1$, 0 ou 1 donc les nouvelles valeurs du déséquilibre sont : $\text{eq}(x) = 0$ ou 1, $\text{eq}(y) = 0$ ou -1, $\text{eq}(z) = 0$, donc la condition AVL est de nouveau respectée.
Il reste à remonter dans la branche menant à la racine et à réitérer éventuellement le processus pour rééquilibrer un arbre AVL après une insertion en un coût $O(h(A))$.

• Tas binaires

Exercice 11

a) On commence par calculer la taille de l'arbre à l'aide de la fonction **`taille`** du cours pour pouvoir créer un tableau de dimension adéquate, que l'on remplit à l'aide de la fonction auxiliaire.

```
let tab_of_arbre a =
  let n = taille a in
  let t = make_vect n 0 in
  let rec aux k = function
    | Nil               -> ()
    | Noeud (fg, x, fd) -> t.(k) <- x ; aux (2*k+1) fg ; aux (2*k+2) fd
  in aux 0 a ; t ;;
```

La restriction aux arbres de type *int* s'explique par la nécessité de fixer une valeur arbitraire dans le tableau **`t`** au moment de sa création.

b) La fonction réciproque n'appelle pas de commentaire particulier :

```
let arbre_of_tab t =
  let rec aux = function
    | k when k >= vect_length t -> Nil
    | k -> let x = t.(k) and fg = aux (2*k+1) and fd = aux (2*k+2)
           in Noeud (fg, t.(k), fd)
  in aux 0 ;;
```

Exercice 12

a) Dans un tableau Caml l'indice du fils gauche du nœud d'indice k est égal à $2k+1$ donc la première feuille est le nœud pour lequel $2k+1 \geqslant n$, soit $k = \left\lceil \frac{n-1}{2} \right\rceil = \left\lfloor \frac{n}{2} \right\rfloor$.

b) Généralisons le résultat de la question précédente en considérant la suite u définie par $u_0 = k$ et la relation $u_{p+1} = 2u_p + 1$. Le fils gauche du nœud d'indice k a pour indice u_1, le petit-fils gauche a pour indice u_2, et plus généralement le descendant gauche de rang p a pour indice u_p. Or dans un arbre parfait, un nœud est de hauteur h s'il possède un descendant gauche de rang h mais pas de rang $h+1$. Le nœud d'indice k est donc de hauteur h si et seulement si $u_h < n \leqslant u_{h+1}$.
Il est facile de calculer que $u_p = 2^p(k+1) - 1$ donc le nœud d'indice k est de hauteur h si et seulement si :

$$2^h(k+1) - 1 < n \leqslant 2^{h+1}(k+1) - 1 \iff \frac{n+1}{2^{h+1}} \leqslant k+1 < \frac{n+1}{2^h}.$$

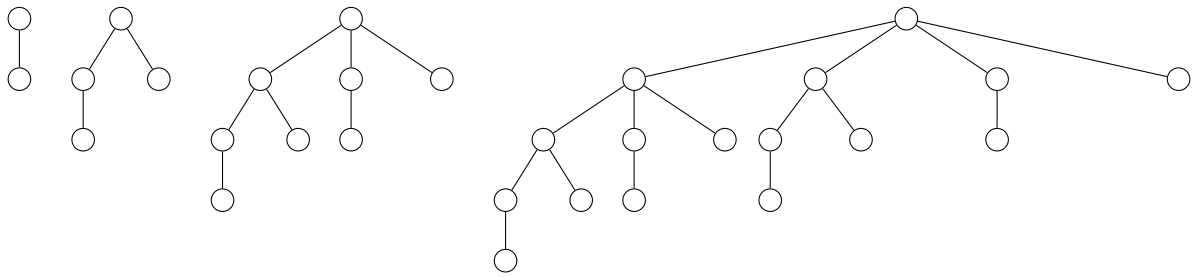
Le nombre maximal d'entiers k qui vérifient cet encadrement est égal à $\left\lceil \frac{n+1}{2^h} - \frac{n+1}{2^{h+1}} \right\rceil = \left\lceil \frac{n}{2^{h+1}} \right\rceil$.

Exercice 13 L'élément maximal d'un tas min, se trouve au niveau des feuilles ; d'après l'exercice 11 il se trouve dans la deuxième moitié du tableau, ce qui donne :

```
let max_tas_min t =
  let n = vect_length t in
  let rec aux acc = function
    | k when k >= n -> acc
    | k             -> aux (max acc t.(k)) (k+1)
  in aux t.(n/2) (n/2+1) ;;
```

Exercice 14

a) Voici les squelettes des arbres binomiaux d'ordres 1, 2, 3 et 4 :

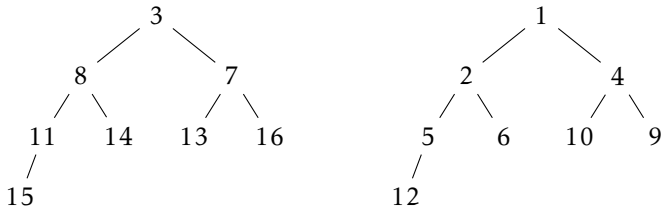


La taille t_p d'un arbre binomial d'ordre p vérifie la relation : $t_p = 1 + \sum_{k=0}^{p-1} t_k$; on prouve par récurrence que $t_p = 2^p$.
La hauteur h_p d'un arbre binomial d'ordre p vérifie la relation : $h_p = 1 + \max(h_{p-1}, h_{p-2}, \ldots, h_0)$; on prouve par récurrence que $h_p = p$.
Le nombre $b(p,k)$ de nœuds à la profondeur k d'un arbre binomial d'ordre p vérifie la relation : $b(p,k) = b(p-1,k-1) + b(p-1,k)$; on prouve par induction que $b(p,k) = \binom{p}{k}$.

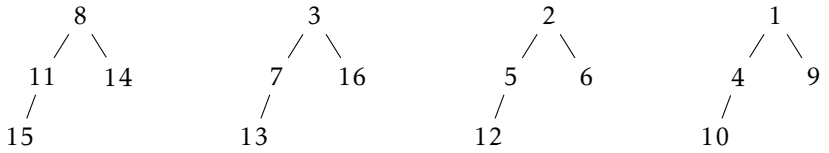
b) Il s'agit tout simplement d'insérer r dans le tas F_d. Puisque F_d est un tas complet, r est initialement placé au premier rang de la profondeur $p-1$ et puisqu'il est inférieur à tout nœud de F_d il va remonter jusqu'à la racine en effectuant $p-1$ permutations, sans qu'il soit nécessaire d'effectuer une comparaison.

c) On dispose maintenant de deux tas binomiaux d'ordre $p-1$: B_1 qui est issu de la transformation de F_g et B_2 qui est issu de la réunion de r et de F_d. Notons que r est situé à la racine de B_2 et qu'il est supérieur à tout élément de B_1. On obtient donc un tas binaire d'ordre p en ajoutant B_1 à la liste des fils de r dans B_2.

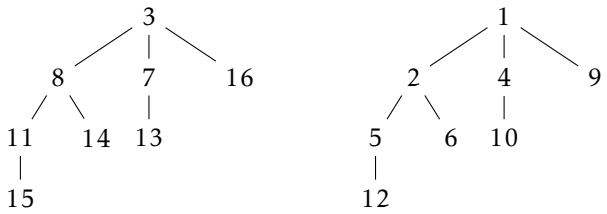
d) La première transformation conduit à un appel récursif sur chacun des deux tas :



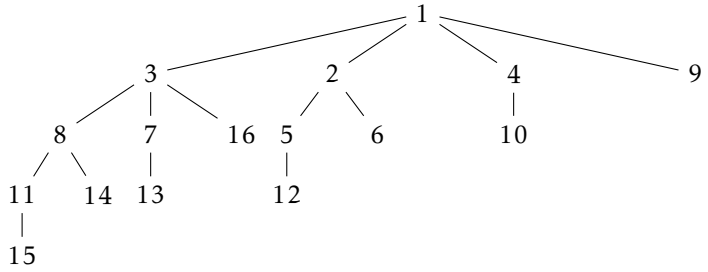
Il faut ensuite transformer en tas binomial chacun de ces quatre tas binaires :



Ce n'est pas la peine d'aller plus loin car ce sont des tas binomiaux d'ordre 2.
On les regroupe maintenant deux par deux pour former des tas binomiaux d'ordre 3 :



On regroupe ces deux tas pour obtenir un tas binomial d'ordre 4 :



e) Le nombre d'échanges u_p effectués dans les différents tas binaires vérifie la relation : $u_p = p - 1 + 2u_{p-1}$ avec $u_2 = 0$ donc $u_p = 3.2^{p-1} - 2p - 1$ (après calcul).
Le nombre v_p de concaténations de tas binomiaux effectuées vérifie la relation : $v_p = 2v_{p-1} + 1$ et $v_2 = 0$ donc $v_p = 2^{p-2} - 1$.
Sachant que ces opérations peuvent être réalisées à coût constant, le coût total de cet algorithme est un $\Theta(n)$ avec $n = 2^p$, donc linéaire vis-à-vis du nombre de sommets.