Arbres rouge-noir

Présentation

Exercice 1. On appelle *arbre rouge-noir* un arbre binaire comportant un champ supplémentaire par nœud : sa couleur , qui peut être rouge ou noire, et qui vérifie les conditions suivantes :

P1 la racine est noire ;

P2 L'arbre vide est noir.

P3 Un nœud est soit rouge soit noir ;

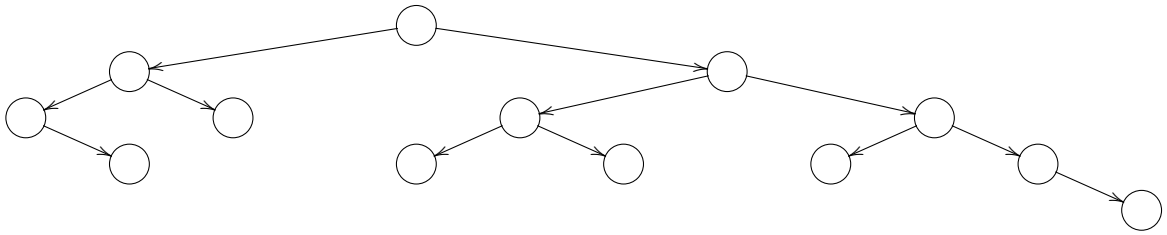
P4 le parent d'un nœud rouge est noir ;

P5 pour chaque nœud, tous les chemins le reliant à **Nil** contiennent le même nombre de nœuds noirs.

Remarque.
1. Comme **Nil** est noir, P5 peut s'écrire : « pour chaque nœud, tous les chemins le reliant à des feuilles (mais pas à **Nil**) contiennent le même nombre de nœuds noirs. »
2. Dans les représentations on omet souvent de représenter l'arbre vide.
3. Un sous-arbre d'un arbre rouge-noir vérifie les propriétés P2 à P5 mais peut-être pas la propriété P1.
   Un arbre qui vérifie les propriétés P2 à P5 est appelé dans ce TP un *descendant licite* (sous-entendu : d'un arbre rouge-noir).

Q.1 Montrer que l'arbre 1 peut être muni d'une coloration rouge-noir. On n'a pas représenté les **Nil**.

FIGURE 1 – Un arbre à colorier



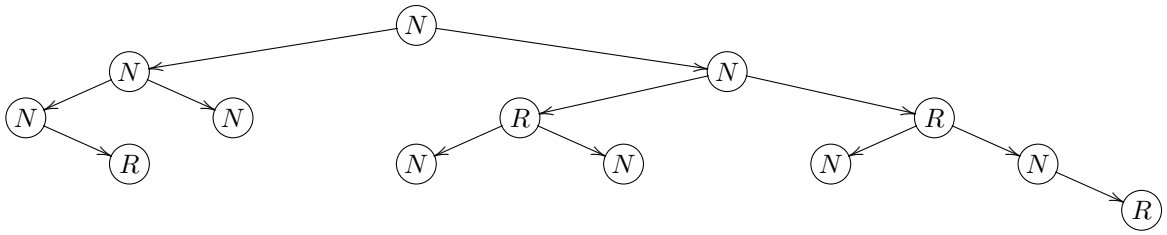
Q.2 Donner un exemple d'arbre qui ne puisse pas être muni d'une coloration rouge-noir.

Q.3 Étant donné un arbre rouge-noir A , on note $b(A)$ le nombre de nœuds noirs (non vides) que contient chacun des chemins de **Nil** à la racine (indépendant du choix de la feuille par définition). Par exemple, $b(\text{Nil}) = 0$.
Montrer que

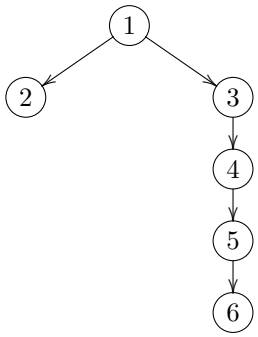
$$b(A) \leq h(A) + 1 \leq 2b(A) \text{ et } |A| \geq 2^{b(A)} - 1$$

En déduire que les arbres rouge-noir sont *équilibrés*, c'est à dire $h(A) = O(\log(|A|))$.

Solution. 1. Voici



2. Un exemple



Si 2 est rouge, alors les nœuds de droite doivent tous être rouge sinon il y a plus de noirs à droite pour accéder à `nil`.

Mais alors on a un nœud rouge père d'un autre rouge.

Si 2 est noir, on a droit à un second noir dans la branche droite. Mais on ne peut pas le placer sans qu'il y ait deux rouges consécutifs.

3. $h(A)+1$ est le nombre de nœuds sur la plus longue branche. Le nombre de noirs sur cette branche est plus petit que le nombre de nœuds. Donc $b(A) \leq h(A)+1$.

4. La plus longue branche est de longueur $h(A)+1$ en comptant **Nil**. Elle contient $h(A)+1-b(A)$ rouges.

Supposons $h(A)+1 = 2b(A)+k$ avec $k>0$. Alors $h(A)+1-b(A) = b(A)+k$. Et donc le nombre de rouges sur cette branche est $b(A)+k$. Chaque rouge a un père noir. Il y a donc au moins $b(A)+k$ noirs. Donc $b(A) = b(A)+k$: Absurde.

Donc $h(A)+1 \leq 2b(A)$

Par induction, on montre que pour un arbre A qui vérifie les propriétés **P2** à **P5** [1], on a $|A| \geq 2^{b(A)} - 1$.

(a) Vrai pour $A =$ **Nil** car $|A| = 0 \geq 2^0 - 1 = 2^{b(A)} - 1$.

Remarque. Pour certains auteurs, les arbres **Nil** sont comptés dans la taille, ils sont noirs et leur hauteur noire est 1, leur hauteur est 1.

Si A est une feuille, elle peut être rouge ou noire, on laisse au lecteur le soin de vérifier l'inégalité.

(b) Soir $A = (G, x, D)$ un arbre qui vérifie les propriétés **P2** à **P5** et dont D, G vérifient HI.

Alors $b(G) = b(A)$ ou $b(G) = b(A) - 1$. Par conséquent $b(G) \geq b(A) - 1$. Idem pour D.

On a par HI puis remarque précédente

$$|A| = 1 + |G| + |D| \geq 1 + 2^{b(G)} - 1 + 2^{b(G)} - 1 \geq 2 \cdot 2^{b(A)-1} - 1 \geq 2^{b(A)} - 1$$

Hérédité OK.

Les arbres rouges-noirs vérifient évidemment les propriétés propriétés **P2** à **P5**. Donc l'inégalité ci-dessus est valide pour les arbres rouges-noirs.

On a

$$|A| \geq 2^{b(A)} - 1$$

Donc

$$|A| \geq 2^{(h(A)+1)/2} - 1 \text{ alors } 2\log_2(|A|+1) \geq h(A) + 1 > h(A)$$

Or, à partir de $x = 3$, $3\log_2(x) > 2\log_2(x+1)$. Donc $3\log_2(|A|) > h(A)$ pour $|A|$ grand.

Ainsi, $h(A) = O(\log(|A|))$: les arbres Rouges-Noirs sont équilibrés.

5. Une fonction auxiliaire `dfs` prend en paramètre un arbre et la couleur du père. Elle renvoie le nombre de nœuds noirs sur les branches si toutes les branches ont la même hauteur noire.

Si le nœud courant est rouge et son père aussi, une exception arrête l'exécution. Si le sous-arbre droit n'a pas la même hauteur noire que le gauche, une exception est soulevée.

1. On n'impose pas la propriété **P1** : cela permet d'avoir une preuve plus concise en évitant de descendre aux petits-fils lorsqu'un fils de la racine dont rouge.

Si les fils gauche et droit ont la même hauteur noire, selon la couleur du nœud courant, la hauteur noire de l'arbre est celle des fils ou celle des fils plus 1.

L'appel initial est `dfs a Rouge`, ce qui permet de vérifier que la racine n'est pas rouge.

Il s'agit d'un parcours en profondeur, donc la complexité est linéaire.

□

On définit les type :

```
type couleur = Rouge | Noir;;

type bicolore = Nil | Node of couleur * bicolore * int * bicolore;;
```

Q4 Écrire une fonction `hauteurnoire : bicolore -> int` qui détermine la *hauteur noire* d'un arbre supposé rouge-noir.

```
# let wa = let n6 = Node(Rouge,Nil,6,Nil) and n11 = Node(Noir,Nil,11,Nil) and
  n15=Node(Noir,Nil,15,Nil) and n22 =  Node(Rouge,Nil,22,Nil)
  and n27=  Node(Rouge,Nil,27,Nil) in let n1=Node(Noir,Nil,1,n6) and n25=Node(Noir,n22,25,n27
      )
  in let n8=Node(Rouge,n1,8,n11) and n17 =Node(Rouge,n15,17,n25) in Node(Noir,n8,13,n17);;
      val wa : bicolore =
  Node (Noir,
   Node (Rouge, Node (Noir, Nil, 1, Node (Rouge, Nil, 6, Nil)), 8,
    Node (Noir, Nil, 11, Nil)),
   13,
   Node (Rouge, Node (Noir, Nil, 15, Nil), 17,
    Node (Noir, Node (Rouge, Nil, 22, Nil), 25, Node (Rouge, Nil, 27, Nil))))
# hauteurnoire wa;;
- : int = 2
# hauteurnoire (Node(Rouge,Nil,6,Nil));;
- : int = 0
```

Solution. Voici

```
let hauteurnoire a =
  let rec aux a acc = match a with
    | Nil -> acc
    | Node(c,g,_,_)->
       if c=Noir
       then aux g (acc+1)
       else aux g acc
  in aux a 0;;
```

□

Q5 Écrire une fonction `check : bicolore -> bool` qui détermine si l'entrée est un arbre rouge-noir. L'algorithme choisi doit être de coût linéaire en $|A|$.

```
# check wa;;
- : bool = true
```

Solution. Voici

```
let check a =
  let rec dfs a cp = match a with (*cp : couleur du pere*)
    | Nil -> 1
    (*Nil est considéré noir, le pere peut etre rouge ou noir*)
    |Node (Rouge,g,_,d) -> if (cp=Rouge)
      then failwith "rouge rouge"
      else let ng = dfs g Rouge and nd = dfs d Rouge in
           if nd <> ng then  failwith "nd <> ng" else ng
    | Node (Noir,g,_,d) ->
       let ng = dfs g Noir and nd = dfs d Noir in
       if nd <> ng then  failwith "nd <> ng"
       else ng+1 in
  try
    dfs a Rouge >= 0
  with Failure _ -> false;;
```

□

Insertion

Dans cette section les arbres rouge-noir sont également des ABR.

Exercice 2. **Q1** Rédiger une fonction insereABR de type `int -> bicolore -> bicolore` qui insère au niveau des feuilles un nouvel élément dans un arbre rouge-noir. Cette fonction devra préserver la structure d'ABR mais sans se soucier de la structure rouge-noir. On attribue arbitrairement la couleur rouge au nouveau nœud.

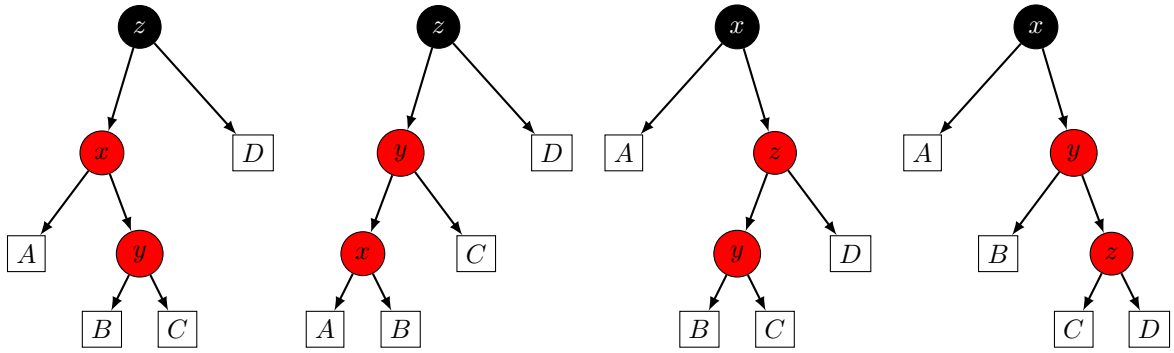
Q2 Lors de l'ajout via la fonction insereABR, quelle(s) propriété(s) des arbres rouge-noir peuvent être violées ?

Solution. C'est une classique insertion dans un ABR

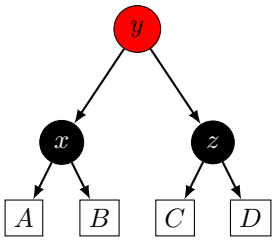
```
let rec insereABR x a = match a with
  |Nil -> Node(Rouge,Nil,x,Nil)
  | Node(_,_,y,_) when y=x ->a
  | Node(c,g,y,d) ->
     if x < y
     then Node(c,insereABR x g,y,d)
     else Node(c,g,y,insereABR x d);;
```

Peuvent être violées la règle qui interdit deux nœuds rouges consécutifs et celle qui impose que la racine soit noire. □

Pour rééquilibrer l'arbre obtenu, toute configuration à 3 nœuds suivante :



est transformée en :



Nous appelons *correction rouge* cette transformation.

Q3 Analysons les transformations effectuées :

(a) La structure (pas la coloration) de l'arbre résultant est en fait obtenue par une ou deux rotations (voir cours) bien choisies.
Identifier ces rotations dans les 4 situations et en déduire que l'arbre résultant est un ABR.

(b) Pour la première des 4 situations à deux nœuds rouges consécutifs ci-dessus, on suppose que :
- Les 4 arbres A, B, C, D sont des ABR.
- A, B, C sont des arbres rouges noirs,
- D est un descendant licite.
- La propriété 5 (égalité du nombre de nœuds noirs pour toutes les branches issues de z) est vérifiée

Montrer que l'arbre résultant est un descendant licite, que sa hauteur noire est la même que l'arbre initial et que c'est un ABR.

Solution. 1. L'arbre résultant est un ABR car il est le fruit d'une ou deux rotations (lesquelles préservent la propriété d'ABR).

- Pour la situation 1, on effectue une rotation gauche autour de x, ce qui met y en fils gauche de z. Puis on effectue une rotation droite autour de z, ce qui met en racine et z en fils droit de cette racine.
- Pour la situation 2, on fait une rotation droite autour de z.
- Les autres cas sont symétriques des précédents.

2. Voici

   (a) Dans la configuration initiale, la hauteur noire des branches passant par A est $1 + b(A)$. C'est encore le cas dans l'arbre résultant.

   De même, les hauteurs noires des branches passant par B, C, D sont $1 + b(B), 1 + b(C), 1 + b(D)$ dans l'arbre résultant. Et toutes ces branches sont de même hauteur noire par hypothèse ($b(A) = b(B) = b(C) = b(D)$). Ainsi, la propriété P5 est encore vérifiée dans l'arbre résultant. P5 OK.

   Observons en outre que la hauteur noire est la même dans l'arbre initial que dans l'arbre résultant.

   (b) Si une branche possède deux nœuds rouges consécutifs dans l'arbre résultant, alors ces deux nœuds sont dans une branche de A, B, C ou D. Or par hypothèse, cela ne se peut. P4 OK.

   (c) La couleur des arbres vides n'a pas été modifiée : elle reste noire (P2). P1 OK.

   (d) La racine de l'arbre résultant est noire : P1 OK.

   (e) Et enfin, les nœuds sont soient rouges soir noirs puisque la seule modification de couleur faite sur les nœuds (qui étaient tous initialement rouges ou noirs) a été de recolorier un rouge en noir.

□

Q4 Rédiger une fonction `correction_rouge` de type `bicolore -> bicolore` qui applique une correction rouge à un arbre de l'une des quatre formes présentées ci-dessus, et qui renvoie l'arbre inchangé dans les autres cas.

Q5 En déduire une nouvelle fonction d'insertion de type `int -> bicolore -> bicolore` baptisée `insereRN` qui insère un élément dans un arbre rouge-noir/ABR tout en préservant la structure rouge-noir/ABR.

Q6 Établir la correction de `insereRN` en présentant avec soin l'hypothèse d'induction.

Q7 Rédiger une fonction `test` de type `int -> bicolore` qui crée un arbre rougenoir en insérant successivement les entiers de 1 à n à l'aide de la fonction `insereRN`.
Vérifier que l'arbre obtenu pour $n = 32768$ est bien un arbre rouge-noir. Quelle est sa hauteur noire ?

Solution. La première fonction n'est pas récursive :

```
let correction_rouge a = match a with
  | Node(Noir,Node(Rouge,Node(Rouge,a,x,b),y,c),z,d)->
    Node(Rouge,Node(Noir,a,x,b),y,Node(Noir,c,z,d))

  | Node(Noir,Node(Rouge,a,x,Node(Rouge,b,y,c)),z,d)->
```

```
    Node(Rouge,Node(Noir,a,x,b),y,Node(Noir,c,z,d))

  | Node(Noir,a,x,Node(Rouge,b,y,Node(Rouge,c,z,d)))->
    Node(Rouge,Node(Noir,a,x,b),y,Node(Noir,c,z,d))

  | Node(Noir,a,x,Node(Rouge,Node(Rouge,b,y,c),z,d))->
    Node(Rouge,Node(Noir,a,x,b),y,Node(Noir,c,z,d))
  | _-> a;;
```

La seconde insère la valeur dans le bon fils et rééquilibre :

```
let insereRN x a =
  let rec aux a = match a with
    | Nil -> Node(Rouge,Nil,x,Nil)
    | Node(c,g,y,d) -> if x < y then correction_rouge (Node(c,aux g,y,d))
      else correction_rouge (Node(c,g,y,aux d)) in
  match aux a with
  | Nil -> failwith "ne doit pas se produire"
  | Node(_,g,z,d)->Node(Noir,g,z,d);;
```

Preuve de correction Il faut montrer que **insereRN** est correcte. On peut déjà établir que l'arbre résultant contient les mêmes étiquettes que l'arbre initial plus l'étiquette insérée (note : il faudrait en faire une preuve rigoureuse...). De même, on néglige la preuve que l'arbre résultant est encore un ABR (re-note : à faire un jour...). On ne s'attache qu'à la hauteur noire et à l'alternance des couleurs.

Pour cela on montre que `aux` appliquée à un descendant licite A d'un arbre RN donne un arbre A' qui α) possède la même hauteur noire que A et β) qui vérifie les propriétés P2,P3,P5 et

- soit P4 si la racine de l'arbre initial A est noire ;
- Soit la propriété P4' suivante (si la racine de l'arbre initial A est rouge) :
  - Tous les nœuds de A' qui ont un fils rouge sont noirs sauf la racine.
  - Et si la racine de A' est rouge, elle peut avoir au plus un fils rouge.

Cette propriété P4' se comprend ainsi : si une branche possède deux nœuds rouges consécutifs, alors ces deux nœuds sont tout en haut de l'arbre, la racine est rouge et ne possède qu'un fils rouge.

Cas de base La correction rouge ne modifie que des arbres de hauteur au moins deux. Pour cette raison, les cas de base concernent les arbres qui, après insertion, sont de hauteur au plus 1. Il y en a 3 catégories :

1. Appliqué à l'arbre **Nil**, `aux` renvoie une feuille rouge ; laquelle respecte bien les propriétés P2,P3,P4',P5.
2. Appliquée à une feuille d'un arbre RN, `insere` place juste une feuille rouge sous la racine (laquelle est noire ou rouge). Les propriétés P2,P3,P5 sont vérifiées.
   - Si la racine est rouge, alors elle a au plus un fils rouge (la feuille qu'on vient d'ajouter). P4 OK
   - Si la racine est noire, il n'y a pas 2 rouges consécutifs par branche. P4' OK.

   La hauteur noire est conservée.
3. Enfin, regardons ce qu'il se passe si on insère la nouvelle feuille comme fils d'un arbre T de hauteur 1 à un seul fils. On établit que l'arbre initial est alors de racine noire et que la couleur de son fils unique est rouge.

   En effet, la racine de T ne peut être rouge. Si elle l'était, son unique fils serait noir donc il faudrait passer par deux noirs pour rejoindre **Nil** depuis la racine de T d'un côté et un seul de l'autre : absurde.

   Pour la même raison, l'unique fils de T ne peut être noir. S'il l'était il faudrait passer par 3 noirs pour rejoindre **Nil** depuis la racine de T d'un côté et deux noirs de l'autre.

   Ainsi l'arbre obtenu après insertion comme fils de T possède une racine noire et deux fils rouge : il est rouge-noir et sa hauteur noire est celle de T.

Dans les 3 cas les propriétés P2,P3,P4' et P5 sont vérifiées.

Hérédité Soit A un arbre RN auquel on applique **aux**. On conserve les notations du code.

Supposons sans perte de généralité que $y < x$, l'arbre résultant est **correction_rouge (Node(c,aux g,y,d))**. On a **aux g** qui respecte HI.

- Si **c=R** la racine de **d** et de **g** est noire. De plus l'arbre résultant est **Node(c,aux g,y,d)** car **correction_rouge** n'effectue aucune modification.

  Par HI, **aux g** vérifie les propriété P2,P3,P4,P5 et a la même hauteur noire que **g** donc que **d**. L'arbre A' a donc la même hauteur noire que A. De plus, puisque **aux d** est de racine noire, A' possède au plus un fils rouge.

Il vient que l'arbre résultant vérifie les propriétés P2,P3,P5. Nous savons que **aux g** respecte P4 (pas deux rouges consécutifs) puisque la racine de **aux g** est noire. Ainsi, si une branche issue de la racine de A' possède deux nœuds rouges, alors il s'agit de la racine de A' et de celle de **aux g**.

Hérédité OK.

— Si **c=N**.

— Si on n'est pas dans une des 4 situations d'appel à **correction_rouge**, alors il ne peut y avoir deux nœuds rouges voisins sur une même branche. En effet **d** n'a pas été modifié et, par HI, les deux nœuds rouges voisins ne peuvent être que la racine de **aux g** et un de ses fils. Mézalors on serait dans une situation d'appel à **correction_rouge** : Absurde. Ainsi P4 est vérifiée.

— Les **Nil** sont restés noirs : P2 OK ;

— les couleurs sont soit rouge soit noire dans **aux g** par HI et dans **d** par hypothèse sur A : P3 OK ;

— $b(\text{aux g}) = b(g)$ par HI et $b(g) = b(d)$ par hypothèse sur A. Or $b(A) = 1+b(d)$ et $b(A') = 1+b(d)$ également : P5 OK.

— Si on est dans une des 4 situations d'appel à **correction_rouge**, la question Q8 nous dit qu'on a gagné.

A la fin de l'algorithme, l'arbre résultant vérifie les propriétés P,P3,P5 et P4 ou P4' et il a la même hauteur noire que l'arbre initial. Dans tous les cas, colorier la racine en noir le transforme en arbre rouge-noir.

Enfin, pour les tests :

```
let rec test = function
  | 0 -> Nil
  | n -> insereRN n (test (n-1)) ;;

let log2 = fun x -> log x /. (log 2.);;

let a = test 32768 in check a, hauteurnoire a, log2 (float_of_int
32768 ) ;;
```

on obtient **(true, 15, 15.)**.

□