

Arbres rouge et noir

Dans un arbre binaire de recherche, les opérations de base telles que l'insertion, la déletion, la recherche, s'effectuent en temps linéaire en la hauteur h de l'arbre. Ces opérations sont donc efficaces lorsque l'arbre est équilibré, mais peuvent être aussi inefficaces que les mêmes opérations sur une liste si la hauteur est du même ordre de grandeur que le nombre d'éléments n de l'arbre. Les arbres rouge et noir sont une amélioration des arbres binaires de recherche qui garantissent que les arbres soient toujours équilibrés, c'est-à-dire $h = O(\log n)$.

1 Définition

Un arbre rouge et noir est un arbre binaire de recherche dont chaque noeud contient une information supplémentaire : sa couleur, qui peut être soit rouge, soit noire. En contrôlant la manière dont les noeuds peuvent être coloriés sur tout chemin de la racine à une feuille, on s'assure que n'importe quel tel chemin n'est pas plus de deux fois plus long que n'importe quel autre, de sorte que l'arbre soit approximativement équilibré.

Les arbres rouge et noir seront représentés en Caml par le type `arbreRN` :

```
type couleur = Rouge
              | Noir ;;
type 'a arbreRN = Vide
               | Noeud of couleur * 'a arbreRN * 'a * 'a arbreRN ;;
```

Remarque. *Tous les arbres que l'on va manipuler seront décrits par le type 'a arbreRN, bien que certains ne soient que des arbres binaires de recherche, et d'autres de vrais arbres rouge et noir, ceci pour éviter les problèmes de conversions de types.*

Définition 1. *On dira qu'un arbre binaire est un arbre rouge et noir s'il vérifie les conditions suivantes :*

1. *C'est un arbre binaire de recherche : on suppose que les éléments étiquetant les noeuds de l'arbre sont ordonnés. Alors l'ensemble des étiquettes de l'arbre de racine le fils gauche du noeud x sont inférieures à x ; l'ensemble des étiquettes des noeuds de l'arbre dont la racine est le fils droit du noeud x sont supérieures à x .*
2. *Chaque noeud a une couleur : soit rouge soit noir.*
3. *Si un noeud est rouge, alors ses deux fils sont noirs.*
4. *Chacun des chemins reliant un noeud à une feuille parmi ses descendants contient le même nombre de noeuds noirs.*

Lemme 1. *Un arbre rouge et noir ayant n noeuds internes a une hauteur de au plus $2\log(n+1)$.*

Ce lemme justifie l'affirmation selon laquelle les arbres rouge et noir sont équilibrés. Une conséquence immédiate est que les opérations telles que la recherche, l'insertion ou la déletion peuvent se faire en $O(\log n)$ opérations élémentaires...malheureusement, l'insertion ou la déletion telles qu'on les pratique habituellement sur les arbres binaires de recherche ne préservent pas la structure d'arbre rouge et noir. Nous allons voir comment les modifier pour qu'elles s'effectuent toujours en $O(\log n)$ et préservent cette structure.

Question 1. À l'aide d'un filtrage portant simultanément sur deux niveaux de l'arbre, écrire une procédure `verif3 : 'a arbreRN → bool` vérifiant qu'un arbre binaire (de type `'a arbreRN`) satisfait la propriété 3 de la définition.

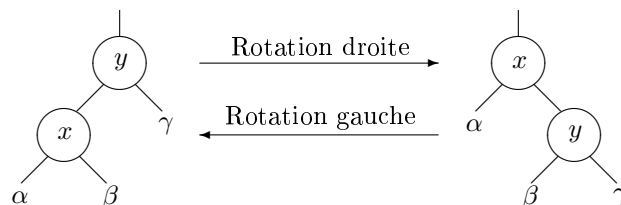
Question 2. Écrire un programme récursif `hauteur_noire : 'a arbreRN → int` qui calcule le nombre de noeuds noirs sur tout chemin de la racine à une feuille si ce nombre est bien défini (c'est-à-dire qu'il est indépendant du chemin choisi), et lève une exception sinon. Vérifier que ceci suffit à garantir que l'arbre satisfait bien la propriété 4 de la définition.

Indication : si la racine est noire, le nombre de noeuds noirs sur tout chemin est 1 plus le nombre de noeuds noirs sur tout chemin partant du fils gauche de la racine, ou 1 plus le nombre de noeuds noirs sur tout chemin partant du fils droit de la racine, si ces nombres sont égaux, et n'est pas défini sinon.

Question 3. En déduire un programme `verif : 'a arbreRN → bool` qui vérifie si un arbre binaire de recherche passé en argument est bien un arbre rouge et noir.

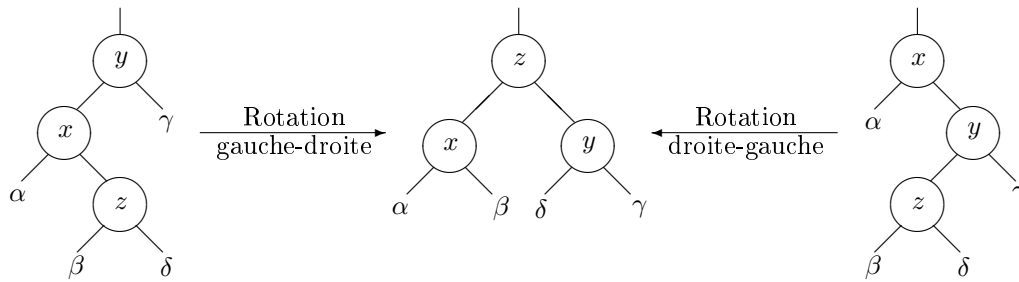
2 Rotations

Les rotations sont des opérations qui modifient localement un arbre binaire de recherche, en préservant ses propriétés. On distingue deux rotations : la *rotation gauche* et la *rotation droite*. Ces deux opérations sont décrites sur la figure ci-dessous (α , β et γ sont des sous-arbres quelconques). Vérifier qu'elles préservent effectivement les propriétés d'ordre d'un arbre binaire de recherche.



Question 4. Écrire les programmes `rotation_g : 'a arbreRN → 'a arbreRN` et `rotation_d : 'a arbreRN → 'a arbreRN` qui effectuent respectivement une rotation gauche et une rotation droite sur la racine de l'arbre passé en argument.

Question 5. En déduire les programmes `rotation_g_d` et `rotation_d_g` qui effectuent les modifications décrites par la figure ci-dessous.



3 Insertion

Pour insérer un nouvel élément dans un arbre rouge et noir, on procède comme suit :

On commence par insérer le noeud comme si l'on effectuait une insertion dans un arbre binaire de recherche, en créant une nouvelle feuille à une position choisie de manière à respecter l'ordre entre les éléments. On colorie cette feuille en rouge, de sorte que la propriété 4 de la définition soit vérifiée.

Question 6. *Écrire une procédure `simple_insert` : 'a arbreRN → 'a → 'a arbreRN qui insère un élément dans un arbre rouge et noir comme décrit ci-dessus.*

Le seul risque est que le père de la nouvelle feuille soit lui aussi rouge, violant la propriété 3 : c'est ce que l'on appelle un *conflit rouge*.

Soit x_1 le noeud que l'on vient d'insérer, x_2 son père, qui est donc rouge, et x_3 le père de x_2 , noir puisque l'arbre initial était un arbre rouge et noir et vérifiait donc la propriété 3. Il y a quatre cas à distinguer :

1. x_1 est le fils gauche de x_2 , et x_2 est le fils gauche de x_3 .
2. x_1 est le fils gauche de x_2 , et x_2 est le fils droit de x_3 .
3. x_1 est le fils droit de x_2 , et x_2 est le fils gauche de x_3 .
4. x_1 est le fils droit de x_2 , et x_2 est le fils droit de x_3 .

Chacun de ces cas se résout par une des rotations décrites ci-dessus : par exemple, dans le premier cas, on effectue une rotation droite sur x_3 (avec $y = x_3$ et $x = x_2$), puis on termine en coloriant x_1 en noir. Ceci résout localement le conflit (faire un dessin pour vérifier!), mais les modifications effectuées ont transformé un sous-arbre de racine noire (x_3) en un sous-arbre de racine rouge (x_2), et il peut maintenant y avoir un nouveau conflit rouge avec le père de x_2 . On le résout à nouveau ; comme il est un étage plus haut dans l'arbre on est sûr que cet algorithme termine.

Question 7. *Trouver les rotations, et les modifications de couleur, qui permettent de résoudre les cas 2, 3 et 4.*

Si la racine d'un arbre rouge et noir est rouge, l'arbre obtenu en colorant cette racine en noir est également un arbre rouge et noir. On peut donc toujours se ramener au cas où l'arbre considéré a une racine noire. On supposera ainsi que les arbres manipulés ont une racine noire.

Question 8. *Écrire une fonction `racine_noire` qui colore en noir la racine d'un arbre.*

Question 9. *Écrire une fonction `conflit` : 'a arbreRN → 'a arbreRN qui prend un arbre en argument, et le réarrange comme décrit ci-dessus s'il s'agit d'un conflit (le rôle de x_3 est joué par la racine de l'arbre, x_1 et x_2 sont ses éventuels descendants) ; sinon elle rendra l'arbre inchangé. On pourra commencer par effectuer la rotation appropriée, puis utiliser le programme `racine_noire` pour effectuer les modifications de couleur nécessaires (qui sont toujours les mêmes...).*

Question 10. Adapter la fonction `simple_insert` de manière à résoudre les conflits lors de la remontée à l'aide de la fonction `conflit`.

Question 11. Évaluer la complexité de cette nouvelle fonction d'insertion; la comparer à celle de l'insertion dans un arbre binaire de recherche.

4 Recherche

Question 12. Écrire la procédure `trouve : 'a arbreRN → 'a → bool` de recherche d'un élément dans un arbre rouge et noir.

Question 13. Prouver le lemme 1. On pourra introduire la hauteur noire d'un noeud x : c'est le nombre de noeuds noirs sur n'importe quel chemin reliant x à une feuille descendante.

Question 14. Trouver un exemple de liste (de longueur n) telle que l'insertion successive de ses éléments dans un arbre binaire de recherche rende un arbre dans lequel la recherche se fera asymptotiquement exponentiellement moins vite que si l'on avait utilisé un arbre rouge et noir.