

Chapitre 10 : Arbres binaires de recherche

Dans ce chapitre, nous allons définir de nouvelles structures de données abstraites : ensemble et tableau associatif puis nous fourniront une réalisation naturelle (et efficace) et de ces types à l'aide d'arbre binaire de recherche. Afin d'optimiser la complexité des opérations élémentaires, nous équilibrerons les arbres à l'aide des arbres bicolores.

1 Tableau associatif

1.1 Type abstrait ensemble

La structure de données abstraite `ensemble` correspond à la notion mathématique d'ensemble : il n'y a pas de répétition, et les éléments ne sont pas ordonnés. Pour cela, on cherche à définir une structure immuable '`a ensemble` muni de quatre opérations.

Type abstrait ensemble en OCaml

```
type 'a ensemble
val ensemble_vide: 'a ensemble
val ajoute      : 'a ensemble -> 'a -> 'a ensemble
val supprime    : 'a ensemble -> 'a -> 'a ensemble
val contient    : 'a ensemble -> 'a -> bool
```

Remarques

Remarque 1 : Notons qu'on pourra souvent relâcher la contrainte naturelle des ensembles en permettant d'ajouter plusieurs fois un même élément. On appelle ce type des **Multiensembles**.

Remarque 2 : Une implémentation possible de cette structure serait d'utiliser des listes ou des tableaux, on aurait alors des opérations en $\mathcal{O}(n)$ pour manipuler n éléments ($\mathcal{O}(\log n)$ en optimisant la recherche avec la dichotomie si l'ensemble est totalement ordonné).

1.2 Type abstrait tableau associatif

Le type abstrait tableau associatif (ou dictionnaire) est une application d'un ensemble A dans un ensemble B. Les éléments de A sont appelés *clés* et ceux de B *valeurs*. Son type abstrait est défini par :

Type abstrait dictionnaire en OCaml

```
type ('a, 'b) dict
val dictionnaire_vide : ('a, 'b) dict
val recherche        : ('a, 'b) dict -> 'a -> 'b option (* None si cle absente*)
val modifie          : ('a, 'b) dict -> 'a -> 'b -> ('a, 'b) dict
val ajoute           : ('a, 'b) dict -> 'a -> 'b -> ('a, 'b) dict
val supprime         : ('a, 'b) dict -> 'a -> ('a, 'b) dict
```

2 Arbre binaire de recherche

2.1 Définitions

Définition 1

On dit qu'un arbre a étiqueté sur A , muni d'une relation d'ordre total, est un arbre binaire de recherche (ABR) lorsque :

• Soit nil

• Soit $a = (g, x, d)$ où g et d sont des arbres binaires de recherche et de plus :

$$\min_{y \in g} E(y) \leq x \leq \max_{y \in d} E(y)$$

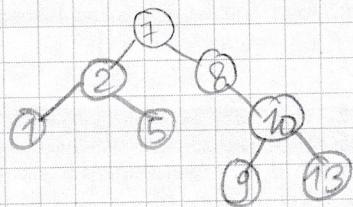
où $E(y)$ est l'étiquette du noeud y .

Remarques

- Tout sous-arbre d'un arbre binaire de recherche est un arbre binaire de recherche.
- Pour un ensemble donné d'étiquettes, il n'y a pas d'unicité de l'arbre binaire de recherche.
- L'ensemble des étiquettes $\varphi(a)$ d'un arbre binaire a peut être défini par :
 - $\varphi(\text{nil}) = \emptyset$
 - $\varphi(g, x, d) = \{x\} \cup \varphi(g) \cup \varphi(d)$

Exemples

Exemples d'arbres 2



3 Première propriété

Théorème 3

a est un ABR si, et seulement si, $\text{inf}(a)$ est tracée par ordre croissant (ou $\text{sup}(a)$ désigne la liste des étiquettes de a dans l'ordre inférieur)

Démonstration par induction structurelle à connaître.

Démonstration

Remarques

- À chaque niveau de l'arbre, les étiquettes lues de gauche à droite forment une suite croissante.
- Le minimum est obtenu en descendant à gauche depuis la racine jusqu'à arriver à la fin de la branche.
- De même pour le maximum en descendant à droite.

3.1 Opérations élémentaires

3.1.1 Implémentation

La structure d'un ABR est la structure d'un arbre binaire.

On utilisera donc la représentation déjà vue dans le cours.

On aura une précondition sur l'entrée arbre de l'algorithme lorsqu'il s'agira d'un ABR.

3.1.2 Minimum et maximum

Nous avons vu comme corollaire du théorème précédent que le nœud le plus à gauche était le minimum de l'ABR. On en déduit naturellement l'algorithme suivant :

Recherche du minimum en OCaml

```
let rec minimum (a: 'a arbre): 'a =
  match a with
  | Nil → false with "Arbre vide → pas de min"
  | Nœud (Nil, x, -) → x
  | Nœud (g, x, d) → minimum g
(*fin*)
```

Cet algorithme effectue $\mathcal{O}(h(a))$ opérations où $h(a)$ est la hauteur de a .

Si l'arbre est équilibré (c'est-à-dire que la hauteur est de l'ordre de $\log |a|$), on obtient un minimum avec une complexité logarithmique.

La recherche du maximum s'effectue de la même façon en allant chercher le nœud le plus à droite.

3.1.3 Recherche d'un élément

Comme son nom l'indique, la recherche est adapté à ce type d'arbres.

Recherche d'un élément en OCaml

```
let rec recherche (a: 'a arbre) (x: 'a): bool =
  match a with
  | Nil → false
  | Nœud (g, b, d) → (x = b) || (x < b && recherche g x) || (x > b && recherche d x)
(*fin*)
```

Comme pour la recherche du minimum la complexité de cet algorithme est linéaire en la hauteur.

3.1.4 Insertion d'un élément

Remarque

Pour la première fois, nous allons modifier un arbre. Il faut bien se rappeler qu'il y a deux façons de modifier un arbre :

- Immutable : On renvoie un nouvel arbre. Attention, les enfants non modifiés ne seront pas recopierés, mais partagés. C'est ce qu'on fera en OCaml.
- Mutable : On modifie l'arbre, c'est-à-dire que l'on modifiera les valeurs des enfants gauches et droits de l'arbre. C'est ce qu'on fera en C.

Tout l'intérêt d'un ABR par rapport à un tableau trié vient de la possibilité de réaliser efficacement des insertions (et des suppressions).

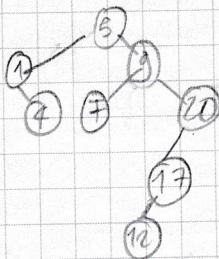
Les insertions se font toujours en bas de l'arbre (l'élément ajouté sera une feuille). On descend donc le long d'une branche, comme lors d'une recherche, de manière à insérer l'élément au bon endroit.

La complexité de cette opération est donc la même que la complexité de la recherche d'un élément. Elle est en $\mathcal{O}(h(a))$.

Insertion d'un élément en Ocaml

```
let rec insere (a: 'a arbre) (x: 'a): 'a arbre =  
  match a with  
  | Nil → N(Nil, x, Nil)  
  | N(g, b, d) when x < b → N(insere g x, b, d)  
  | N(g, b, d) → N(g, x, insere d x)  
  
(*fin*)
```

Exemple 4



Remarque

On remarque qu'en fonction de l'ordre d'insertion des éléments, on peut obtenir un arbre non équilibré.

3.1.5 Suppression d'un élément

Cas simples

Le problème de la suppression d'un élément d'un arbre binaire de recherche est plus compliqué. On peut noter quand même quelques cas simples :

- La suppression d'une feuille ;
- La suppression d'un nœud ayant un unique enfant.
- La suppression du minimum ou du maximum.

Exercice 5

Écrire la fonction `suppression_min (a: 'a arbre) : 'a arbre * 'a` qui supprime le nœud contenant l'étiquette du minimum dans un ABR et qui renvoie ce minimum.

La complexité de cette opération est linéaire en la hauteur de l'arbre.

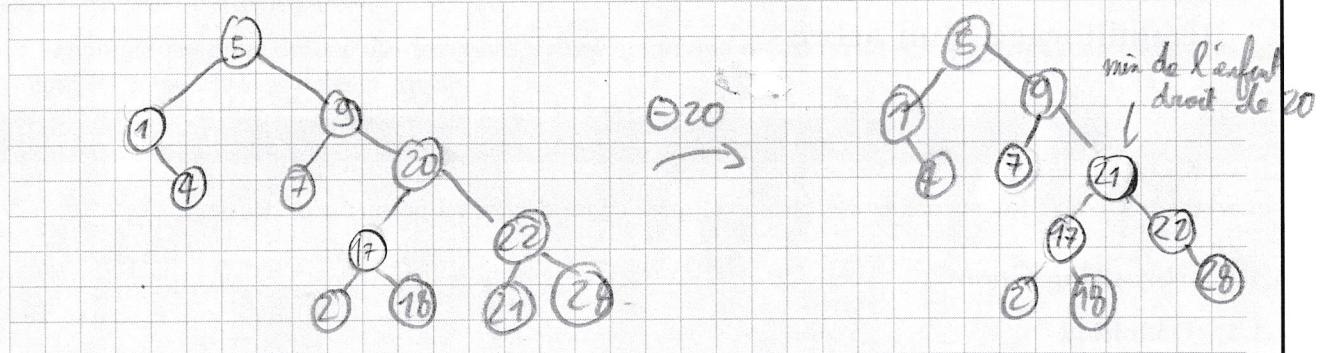
Cas général

Pour faire le cas général, nous allons utiliser l'algorithme de la suppression du minimum.

Si le nœud à supprimer possède un sous arbre droit, on peut procéder ainsi :

- On considère le sous-arbre droit d de l'élément x à supprimer.
- On récupère et supprime le minimum m de d . Soit d' le résultat de cette suppression.
- On renvoie l'arbre (g, m, d') .

Exemple 6



Suppression d'un élément en Ocaml

```
let rec supprime (a: 'a arbre) (x: 'a): 'a arbre =
  match a with
  | Nil → Nil
  | Nœud (g, v, d) when v > x → Nœud (supprime g, v, d)
  | Nœud (g, v, d) when v < x → Nœud (g, v, supprime d)
  | Nœud (g, _, Nil) → g
  | Nœud (g, _, d) → let d2, m = suppression_min d in Nœud (g, m, d2)
```

(*fin*)

On remarque que l'on ne supprime qu'une occurrence de l'élément, on peut modifier la fonction pour les supprimer toutes.

La complexité de la suppression d'un élément est linéaire en la hauteur de l'arbre.

3.2 Réalisation d'un dictionnaire par un ABR

Les ABR permettent de réaliser de manière fonctionnelle la structure de dictionnaire, à condition que les clés soient choisies dans un ensemble totalement ordonné. Pour ce faire, on stocke dans les nœuds les couples (clé, valeur) et toutes les comparaisons se font uniquement sur les clés.

Les opérations élémentaires sur les dictionnaires sont optimisé par la structure d'un ABR. On peut donc chercher, ajouter ou supprimer un élément en $\mathcal{O}(h)$.

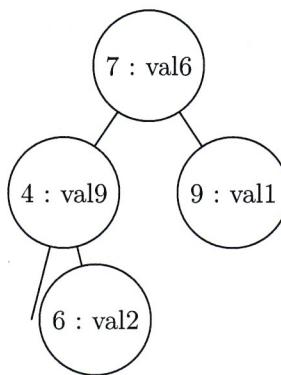


FIGURE 1 – Représentation d'un dictionnaire par un ABR

Si l'arbre est équilibré, on aura une complexité en $\mathcal{O}(\log |a|)$.

4 Rééquilibrage d'un arbre

Les opérations précédentes sont toutes en $\mathcal{O}(h(a))$, or on sait que $h(a) \leq |a| \leq 2^{h(a)} - 1$ donc on aimerait se rapprocher au plus de $\log |a|$ et pour cela, il faut que l'arbre soit le plus proche possible d'un arbre parfait. On parle alors d'équilibrage.

Les arbres rouge-noir (ou encore arbres bicolores) que l'on va décrire ici permettent ce décalage.

4.1 Arbres rouge-noir

4.1.1 Définition

Définition 7

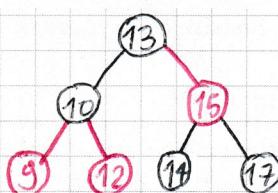
Un arbre binaire de recherche est un arbre rouge-noir s'il satisfait aux propriétés suivantes :

- Chaque nœud est soit rouge, soit noir ;
- Les nœuds rouges ont des enfants noirs ;
- Tous les chemins reliant la racine à un nœud vide contiennent le même nombre de nœuds noirs

Dans la suite, on note $b(a)$ le nombre de nœuds noirs le long d'un chemin de la racine à son sous-arbre vide d'un arbre rouge-noir a .

Remarque : Pour un sous arbre non vide, ses deux enfants sont également des arbres rouge-noir.

Exemple 8



4.1.2 Réalisation pratique

En C comme en OCaml, le plus simple est d'ajouter un champ couleur à chaque nœud non vide (les nœuds vides seront implicitement noirs).

Type arbre bicolore

```
type couleur = R | N
type 'a arn =
| Nil
| Noeud of couleur * 'a arn * 'a * 'a arn
```

Exercice 9

Écrire en OCaml une fonction `verifie ('a: arn): bool` permettant de vérifier si un arbre vérifie bien la condition d'arbre rouge-noir.

Quelques fonctions pratiques

Pour la suite, nous nous dotons des fonctions `rouge` qui vérifie qu'un nœud est rouge, `gauche` et `droite` qui renvoient les sous-arbres gauche et droit.

Fonction pratique

```
let rouge a =
  match a with
  | Noeud(R, _, _, _) -> true
  | _ -> false
let gauche a =
  match a with
  | Noeud(_, g, _, _) -> g
  | _ -> Nil
let droite a =
  match a with
  | Noeud(_, _, d, _) -> d
  | _ -> Nil
```

4.1.3 Propriété fondamentale

Propriété 10

Sur tout arbre rouge-noir, on a :

$$h(a) \leq 2 \log_2 (l(a) + 1)$$

Autrement dit, l'arbre rouge-noir est équilibré.

Cette propriété est un corollaire immédiat de la propriété suivante :

Propriété 11

Pour tout arbre rouge-noir a , on a les deux inégalités

- $h(a) \leq 2b(a)$,
- $2^{b(a)} \leq |a| + 1$.

Rappel : $b(a)$ est le nombre de nœuds noirs de a le long d'un chemin de la racine à son sous-arbre vide d'un arbre rouge-noir a .

On montre ces inégalités par induction structurelle sur a

4.2 Recherche

Les opérations qui ne font que consulter la structure d'arbre binaire de recherche sont inchangée :

- Chercher un élément (Vérifier sa présence ou obtenir la valeur associée à la clé) ;
- Déterminer le min ou le max.

4.3 Insertion

Pour l'insertion , en revanche, il faut maintenant garantir la propriété d'arbre rouge-noir, en plus de la propriété d'arbre binaire de recherche.

Lors de l'ajout d'un élément on a le choix entre lui mettre une couleur rouge ou une couleur noire.

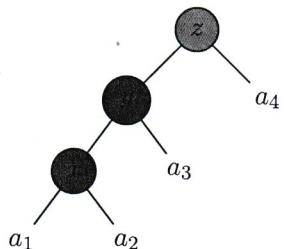
L'avantage de mettre une couleur noire est de garder l'invariant : chaque noeud rouge n'a que des enfants noirs. L'inconvénient est de ne pas garder l'invariant sur le nombre de noeuds noirs d'une racine à une feuille.

On mettra donc une couleur rouge avec le risque d'avoir un enfant rouge de parent rouge mais avec l'avantage de préserver l'équilibrage noir.

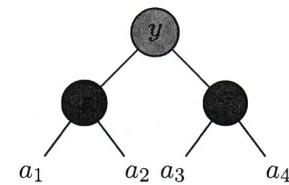
Pour rétablir les couleurs, on va alors utiliser une succession de d'opérations élémentaires : les rotations et les bascules.

4.3.1 Rotation

Lors d'un appel récursif à insertion, on peut se retrouver dans la situation d'avoir deux noeuds rouges consécutifs
On définit alors l'opération rotation droite, comme la transformation suivante :



Rotation à droite



Implémentation

Rotation à droite

```
let rotation_droite a = match a with
| Noeud(c, Noeud(R, t1, y, a3), z, a4)
|   -> Noeud(c, t1, y, Noeud(R, a3, z, a4))
| _ -> failwith "Invalide"
```

Ici $t1$ représente l'arbre

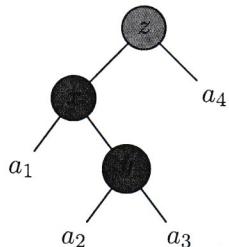


On écrira de façon symétrique `rotation_gauche`.

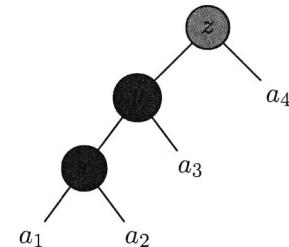
Autre cas possible

L'insertion peut se faire à droite de y , on peut alors obtenir un déséquilibre de ce côté.

Une rotation à gauche nous permet de revenir au cas précédent.



Rotation à gauche
à la racine de y

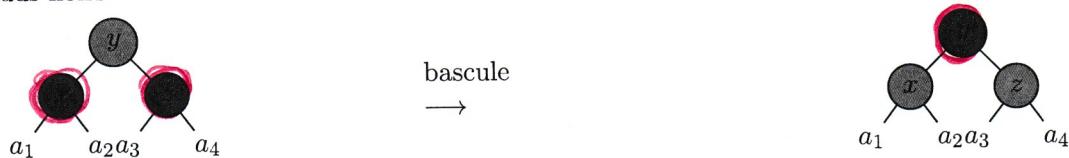


Remarques

- Cette opération se fait en temps constant.
- Il est assez facile de voir que si l'arbre de départ est un ABR alors l'arbre d'arrivée l'est aussi.
- La hauteur en noeuds noirs de l'arbre n'a pas changé.
- On a potentiellement recréé un déséquilibre à droite, si la racine de a_3 ou a_4 est rouge. Pour régler ce problème, nous allons utiliser une bascule.

Bascule

La bascule permet de faire remonter les défauts de coloration vers la racine tout en préservant l'équilibre en noeuds noirs :



Implémentation

Bascule

```
let bascule a = match a with
  | Noeud(N, Noeud(R, a1, x, a2), y, Noeud(R, a3, z, a4))
    -> Noeud(R, Noeud(N, a1, x, a2), y, Noeud(N, a3, z, a4))
  | _ -> failwith "Invalide"
```

Remarques

- À chaque bascule, on ne modifie pas l'équilibre de noeud noir.
Si on effectue une bascule directement sur la racine de l'arbre considéré, on finit par la noircir ce qui augmente de un la profondeur noire de chaque feuille mais ne change pas l'équilibrage global.
- La complexité de cette opération est constante.
- L'arbre reste un ABR.

4.3.2 Insertion

Il ne reste plus qu'à insérer l'élément.

La fonction sera récursive, et le résultat renvoyé récursivement sera testé pour savoir s'il est dans une des situations décrites ci-dessus.

Ce test sera effectué par les fonctions `essai_rotation_droite` et `essai_rotation_gauche`.

Essai rotation droite

```
let essai_rotation_droite a =
  (* Enfant gauche et petit enfant gauche gauche rouge *)
  if rouge (gauche a) && rouge (gauche (gauche a)) then
    bascule (rotation_droite a)
  (* Enfant gauche et petit enfant gauche droit rouge *)
  else if rouge (gauche a) && rouge (droite (gauche a)) then
    match a with
    | Noeud(c, g, x, d)
      -> bascule (rotation_droite (Noeud(c, rotation_gauche g, x, d)))
    | _
      -> failwith "Invalide"
  else
    a
```

On écrira de façon symétrique `essai_rotation_gauche`

Racine

Les tests se font toujours au niveau des enfants. Il manque donc la vérification sur la racine de l'arbre qui pourrait être rouge après une bascule.

Il faut donc noircir cette racine, ce qui ne posera aucun problème sur l'équilibre de l'arbre.

Noircir la racine

```
let noircit a = match a with
| Noeud(c, a1, x, a2) -> Noeud(N, a1, x, a2)
| _ -> a
```

Fonction insertion

Il ne reste plus qu'à écrire `insertion`.

L'idée est d'utiliser le même algorithme que celui vu en début du cours en rajoutant les tests après chaque appel pour rééquilibrer les couleurs de l'arbre.

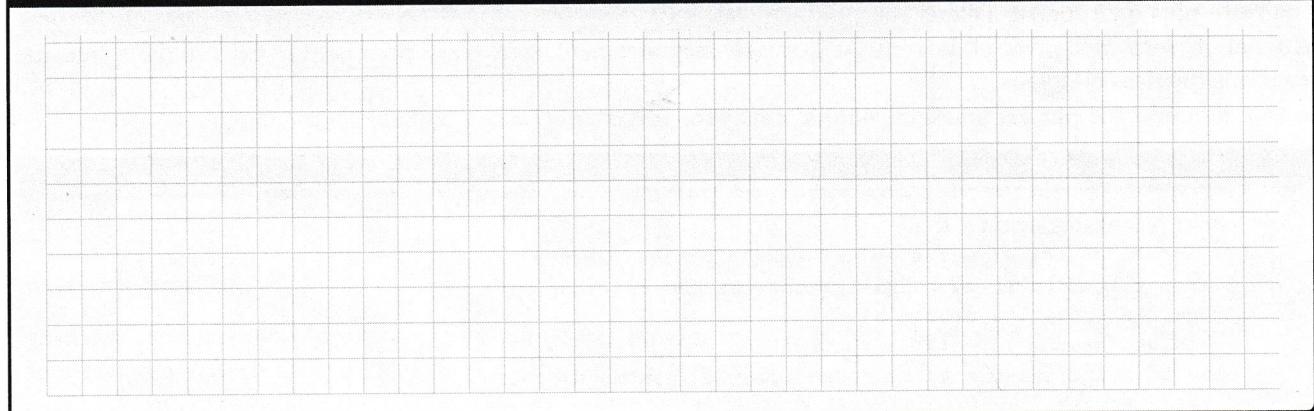
On utilisera une fonction auxiliaire pour ne pas noircir récursivement toutes les racines.

Insertion d'un élément en Ocaml

```
let insere a x =
let rec insere_aux a =
match a with
| Nil -> Noeud(R, Nil, x, Nil)
| Noeud(c, g, v, d) when v > x
    . . . → essai-rotation-droite (Noeud (c, insere-aux g, v, d))
| Noeud(c, g, v, d).
    in → essai-rotation-gauche (Noeud (c, g, v, insere d))
    noircit (aux a)
(*fin*)
```

Exemple

Exemple 12



Remarques

- Comme pour l'insertion, la complexité est linéaire en la hauteur de l'arbre. Mais comme on a démontré que la hauteur est logarithmique en la taille de l'arbre. La complexité est $\mathcal{O}(\log |a|)$.
- On peut aussi obtenir une suppression dans un arbre rouge-noir en bonne complexité mais la conservation de la propriété est plus complexe.

Démonstration(4)

- si l'arbre est vide, il n'y a rien à prouver;
- sinon l'arbre est de la forme $a = (g, z, d)$. On note I_g, I_d, I_a , la liste des étiquettes de $g, d ; a$ dans l'ordre infixe, on a par définition : $I_a = I_g, z, I_d$
 - Supposons que a soit un ABR. On a alors :

g et d sont des ABR, donc I_g et I_d sont croissantes par hypothèse d'induction ;
 $\max_{y \in g} E(y) \leq z \leq \min_{y \in g} E(y)$ puisque la condition d'ordre est vérifiée à la racine I_g de a , et donc $\max_{y \in g} E(y) \leq z \leq \min_{y \in d} E(y)$
 - Inversement, supposons que I_a soit croissante. On a alors :

I_g et I_d sont croissantes, donc par hypothèse d'induction g et d sont des ABR
 $\max_{y \in g} E(y) \leq z \leq \min_{y \in d} E(y)$, donc $\max_{y \in g} E(y) \leq z \leq \min_{y \in g} E(y)$ et la condition d'ordre est vérifiée à la racine de a .

Donc a est un ABR

Exercice 5

let suppression - min $a =$
 match a with

- | Nil \rightarrow failwith
- | Nœud (g, z, d) \rightarrow (d, z)
- | Nœud (g, z, d) \rightarrow let $g\text{-max}, el = \text{suppression } g \text{ in}$
 Nœud ($g\text{-max}, z, d$), d

On montre ces propriétés par induction structurale sur a :

- Le cas de base est l'arbre noir \emptyset : $b(\emptyset) = -1$ et $l(\emptyset) = b(\emptyset) = 0$
- Puisque a un arbre rouge - noir non vide et g et d ses deux sous-arbres
 Si z a l'air : Soit la racine de a est noire, soit elle est rouge.

Si la racine de a est noire alors $b(g) = b(d) = b(a) - 1$. Dès lors

$$\begin{aligned} l(a) &= 1 + \max(b(g), b(d)) \\ &\leq 1 + 2(b(a) - 1) \text{ par hypothèse d'induction} \\ &< 2b(a) \end{aligned}$$

De plus

$$\begin{aligned} |a| + 1 &= |g| + |d| + 2 \\ &\geq 2b(a) - 1 + 2^{b(a)-1} - 1 + 2 \text{ par HT} \\ &= 2^{b(a)} \end{aligned}$$

Si la racine de a est rouge, alors $b(g) = b(d) = b(a)$

Si a ne contient qu'un seul nœud alors la propriété est vérifiée

Si a contient deux seuls arbres non vides et dont les racines sont noires

Soit gg, gd, dg et dd les quatre sous-arbres sous les racines noires et

$$\begin{aligned} b(a) &= 2 + \max(b(gg), b(gd), b(dg), b(dd)) \\ &\leq 2 + 2(b(u)) \text{ par HT} \\ &\leq 2b(a) \end{aligned}$$

$$|a| + 1 = |g| + |d| + 2 \geq 2^{b(a)-1} + 2^{b(a)-1+2} \geq 2^{b(a)}$$

La correction de insertion - on va se prouver par induction structurelle

Propriété 13

- $b_r(a) = b^*(a)$, où b^* est la valeur de b à la fin de l'exécution;
- la propriété d'MBR est conservée;
- si la racine est noir, le résultat est un arbre rouge-noir;
- sinon, le résultat produit un arbre rouge-noir « presque-correct » : le seul problème potentiel est que la racine peut être rouge et posséder un de ses enfants rouges.