Algorithme 2

Arbres rouge-noir

Louis-Claude Canon
louis-claude.canon@univ-fcomte.fr

Licence 2 Informatique – Semestre 4

Plan

Plan

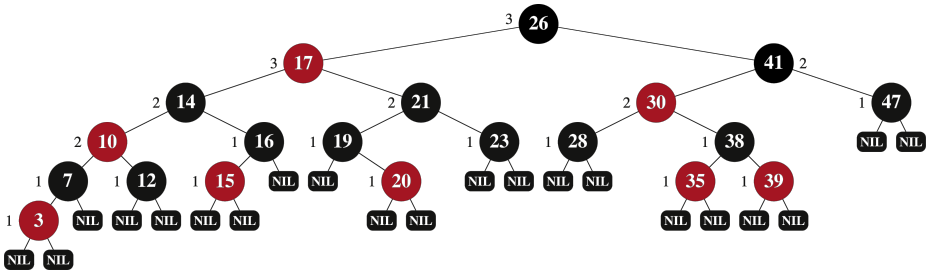
Contexte d'utilisation des arbres rouge-noir

- Limites des arbres binaires de recherche : hauteur en $O(n)$ au pire cas (n étant le nombre de nœuds).
- Les arbres rouge-noir (ou *arbres bicolores*) sont des arbres binaires de recherche auto-équilibrés : la hauteur est en $O(\log n)$.
- Les opérations classiques prendront $O(\log n)$ même dans le pire cas.

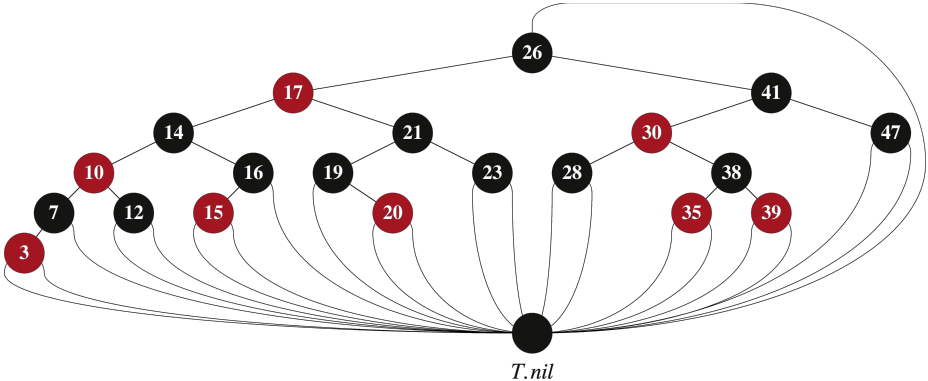
Structure des arbres rouge-noir

- Un arbre rouge-noir (1972) reprend la structure d'un arbre binaire de recherche en rajoutant un bit de données par nœud : la couleur (rouge ou noire).
- Chaque feuille est un nœud noir vide qui représente *NIL*. C'est un nœud sentinelle et il est unique pour toutes les feuilles : $T.nil$.
- La *hauteur noire* $bh(x)$ d'un nœud x est le nombre de nœuds noirs sur le chemin entre x et une feuille (le nœud sentinelle) qui en est une descendante (x non compris, $T.nil$ compris).

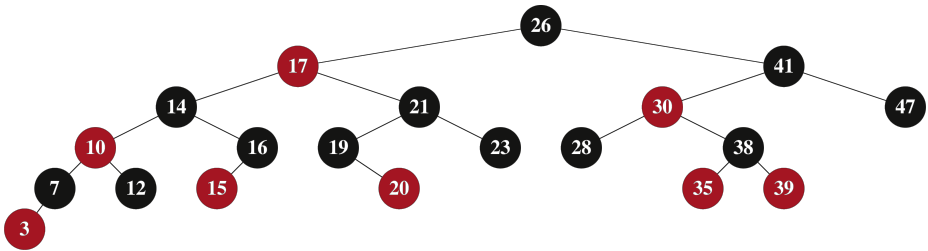
Exemple d'arbres rouge-noir



Exemple d'arbres rouge-noir



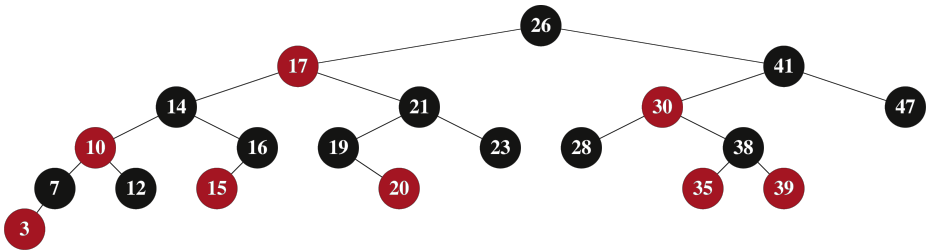
Exemple d'arbres rouge-noir



Propriétés des arbres rouge-noir

1. Chaque nœud est soit noir, soit rouge.
2. La racine est noire.
3. Chaque feuille $T.nil$ est noire.
4. Si un nœud est rouge, alors ses enfants sont noirs (il n'y a donc jamais deux nœuds rouges successifs sur un chemin de la racine à une feuille).
5. Pour chaque nœud, tous les chemins d'un nœud à ses feuilles descendantes ont le même nombre de nœuds noirs.

Exemple d'arbres rouge-noir



Analyse sur la hauteur

La hauteur d'un arbre rouge-noir avec n nœuds internes (hors $T.nil$) est au plus $2\log_2(n+1)$.

Preuve

- Le sous-arbre enraciné en x contient au moins $2^{bh(x)} - 1$ nœuds internes (propriété 5). Par récurrence :
  - Si la hauteur noire de x est $bh(x) = 0$, x est une feuille (i.e. $bh(x) = 0 \Rightarrow 2^{bh(x)} - 1 = 0$).
  - Si $bh(x) > 0$, x a deux fils ayant une hauteur noire $bh(x)$ ou $bh(x) - 1$. Par hypothèse de récurrence, chacun a au moins $2^{bh(x)-1} - 1$ nœuds internes. Donc x a au moins $2(2^{bh(x)-1} - 1) + 1 = 2^{bh(x)} - 1$ nœuds internes.
- La hauteur noire de la racine est au moins la moitié de sa hauteur (par la propriété 4) : $bh(x) \geq h/2$ avec h la hauteur de l'arbre.
- $n \geq 2^{bh(x)} - 1 \geq 2^{h/2} - 1$ et donc $\log_2(n+1) \geq h/2$.

Analyse sur la hauteur

La hauteur d'un arbre rouge-noir avec n nœuds internes (hors $T.nil$) est au plus $2\log_2(n+1)$.

Preuve

- Le sous-arbre enraciné en x contient au moins $2^{bh(x)} - 1$ nœuds internes (propriété 5). Par récurrence :
  - Si la hauteur noire de x est $bh(x) = 0$, x est une feuille ($T.nil$) et $2^{bh(x)} - 1 = 0$.
  - Sinon, la hauteur noire des enfants de x est au moins $bh(x) - 1$ et leurs sous-arbres enracinés ont au moins $2^{bh(x)-1} - 1$ nœuds. Celui enraciné en x a donc au moins $2 \times (2^{bh(x)-1} - 1) + 1 = 2^{bh(x)} - 1$ nœuds.
  - Un arbre a donc $n \geq 2^{bh(r)} - 1$ nœuds avec r la racine.
- La hauteur noire de la racine est au moins la moitié de sa hauteur (par la propriété 4) : $bh(r) \geq h/2$ avec h la hauteur de l'arbre.
- $n \geq 2^{bh(r)} - 1 \geq 2^{h/2} - 1$ et donc $\log_2(n+1) \geq h/2$.

Analyse sur la hauteur

La hauteur d'un arbre rouge-noir avec n nœuds internes (hors $T.nil$) est au plus $2\log_2(n+1)$.

Preuve

- Le sous-arbre enraciné en x contient au moins $2^{bh(x)} - 1$ nœuds internes (propriété 5). Par récurrence :
  - Si la hauteur noire de x est $bh(x) = 0$, x est une feuille ($T.nil$) et $2^{bh(x)} - 1 = 0$.
  - Sinon, la hauteur noire des enfants de x est au moins $bh(x) - 1$ et leurs sous-arbres enracinés ont au moins $2^{bh(x)-1} - 1$ nœuds. Celui enraciné en x a donc au moins $2 \times (2^{bh(x)-1} - 1) + 1 = 2^{bh(x)} - 1$ nœuds.
  - Un arbre a donc $n \geq 2^{bh(r)} - 1$ nœuds avec r la racine.
- La hauteur noire de la racine est au moins la moitié de sa hauteur (par la propriété 4) : $bh(r) \geq h/2$ avec h la hauteur de l'arbre.
- $n \geq 2^{bh(r)} - 1 \geq 2^{h/2} - 1$ et donc $\log_2(n+1) \geq h/2$.

Analyse sur la hauteur

La hauteur d'un arbre rouge-noir avec n nœuds internes (hors $T.nil$) est au plus $2\log_2(n+1)$.

Preuve

- Le sous-arbre enraciné en x contient au moins $2^{bh(x)} - 1$ nœuds internes (propriété 5). Par récurrence :
  - Si la hauteur noire de x est $bh(x) = 0$, x est une feuille ($T.nil$) et $2^{bh(x)} - 1 = 0$.
  - Sinon, la hauteur noire des enfants de x est au moins $bh(x) - 1$ et leurs sous-arbres enracinés ont au moins $2^{bh(x)-1} - 1$ nœuds. Celui enraciné en x a donc au moins $2 \times (2^{bh(x)-1} - 1) + 1 = 2^{bh(x)} - 1$ nœuds.
  - Un arbre a donc $n \geq 2^{bh(r)} - 1$ nœuds avec r la racine.
- La hauteur noire de la racine est au moins la moitié de sa hauteur (par la propriété 4) : $bh(r) \geq h/2$ avec h la hauteur de l'arbre.
- $n \geq 2^{bh(r)} - 1 \geq 2^{h/2} - 1$ et donc $\log_2(n+1) \geq h/2$.

Analyse sur la hauteur

La hauteur d'un arbre rouge-noir avec n nœuds internes (hors $T.nil$) est au plus $2\log_2(n+1)$.

Preuve

- Le sous-arbre enraciné en x contient au moins $2^{bh(x)} - 1$ nœuds internes (propriété 5). Par récurrence :
  - Si la hauteur noire de x est $bh(x) = 0$, x est une feuille ($T.nil$) et $2^{bh(x)} - 1 = 0$.
  - Sinon, la hauteur noire des enfants de x est au moins $bh(x) - 1$ et leurs sous-arbres enracinés ont au moins $2^{bh(x)-1} - 1$ nœuds. Celui enraciné en x a donc au moins $2 \times (2^{bh(x)-1} - 1) + 1 = 2^{bh(x)} - 1$ nœuds.
  - Un arbre a donc $n \geq 2^{bh(r)} - 1$ nœuds avec r la racine.
- La hauteur noire de la racine est au moins la moitié de sa hauteur (par la propriété 4) : $bh(r) \geq h/2$ avec h la hauteur de l'arbre.
- $n \geq 2^{bh(r)} - 1 \geq 2^{h/2} - 1$ et donc $\log_2(n+1) \geq h/2$.

Analyse sur la hauteur

La hauteur d'un arbre rouge-noir avec n nœuds internes (hors $T.nil$) est au plus $2\log_2(n+1)$.

Preuve

- Le sous-arbre enraciné en x contient au moins $2^{bh(x)} - 1$ nœuds internes (propriété 5). Par récurrence :
  - Si la hauteur noire de x est $bh(x) = 0$, x est une feuille ($T.nil$) et $2^{bh(x)} - 1 = 0$.
  - Sinon, la hauteur noire des enfants de x est au moins $bh(x) - 1$ et leurs sous-arbres enracinés ont au moins $2^{bh(x)-1} - 1$ nœuds. Celui enraciné en x a donc au moins $2 \times (2^{bh(x)-1} - 1) + 1 = 2^{bh(x)} - 1$ nœuds.
  - Un arbre a donc $n \geq 2^{bh(r)} - 1$ nœuds avec r la racine.
- La hauteur noire de la racine est au moins la moitié de sa hauteur (par la propriété 4) : $bh(r) \geq h/2$ avec h la hauteur de l'arbre.
- $n \geq 2^{bh(r)} - 1 \geq 2^{h/2} - 1$ et donc $\log_2(n+1) \geq h/2$.

Analyse sur la hauteur

La hauteur d'un arbre rouge-noir avec n nœuds internes (hors $T.nil$) est au plus $2\log_2(n+1)$.

Preuve

- Le sous-arbre enraciné en x contient au moins $2^{bh(x)} - 1$ nœuds internes (propriété 5). Par récurrence :
  - Si la hauteur noire de x est $bh(x) = 0$, x est une feuille ($T.nil$) et $2^{bh(x)} - 1 = 0$.
  - Sinon, la hauteur noire des enfants de x est au moins $bh(x) - 1$ et leurs sous-arbres enracinés ont au moins $2^{bh(x)-1} - 1$ nœuds. Celui enraciné en x a donc au moins $2 \times (2^{bh(x)-1} - 1) + 1 = 2^{bh(x)} - 1$ nœuds.
  - Un arbre a donc $n \geq 2^{bh(r)} - 1$ nœuds avec r la racine.
- La hauteur noire de la racine est au moins la moitié de sa hauteur (par la propriété 4) : $bh(r) \geq h/2$ avec h la hauteur de l'arbre.
- $n \geq 2^{bh(r)} - 1 \geq 2^{h/2} - 1$ et donc $\log_2(n+1) \geq h/2$.

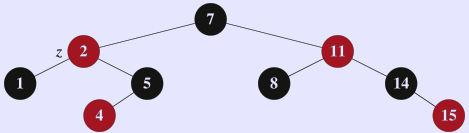
Propriétés des arbres rouge-noir

1. Chaque nœud est soit noir, soit rouge.
2. La racine est noire.
3. Chaque feuille $T.nil$ est noire.
4. Si un nœud est rouge, alors ses enfants sont noirs (il n'y a donc jamais deux nœuds rouges successifs sur un chemin de la racine à une feuille).
5. Pour chaque nœud, tous les chemins d'un nœud à ses feuilles descendantes ont le même nombre de nœuds noirs.

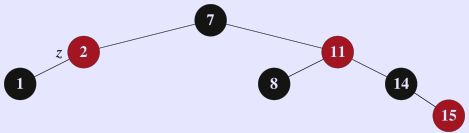
Question

Lequel de ces arbres respecte les propriétés des arbres rouge-noir ?

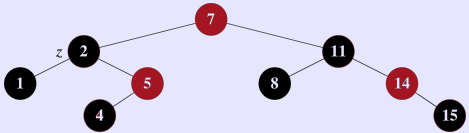
1.



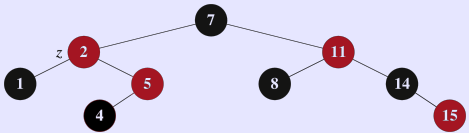
2.



3.



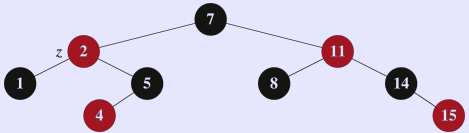
4.



Question

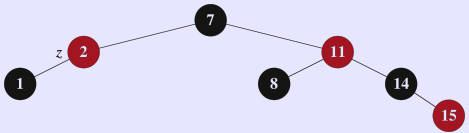
Lequel de ces arbres respecte les propriétés des arbres rouge-noir ?

1.



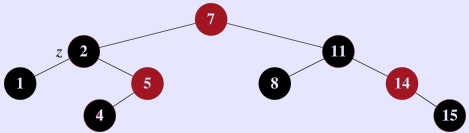
✓

2.



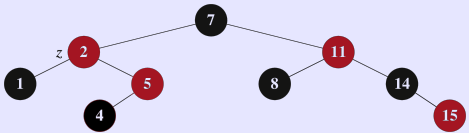
✗(5)

3.



✗(2)

4.



✗(4)

Plan

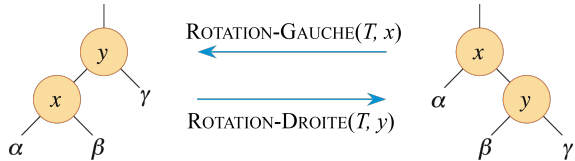
Rotation

- Les insertions et suppressions peuvent violer les propriétés des arbres rouge-noir.
- L'opération de *rotation* préserve la propriété des arbres binaires de recherche et peut ré-équilibrer un arbre.
- Les rotations fonctionnent soit à droite, soit à gauche.

Rotation-Gauche

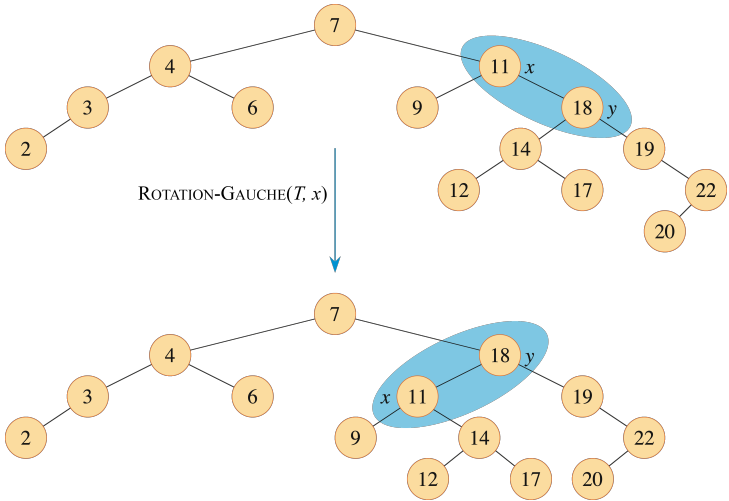
Rotation-Gauche(T, x)

$y \leftarrow x.droite$
$x.droite \leftarrow y.gauche$
si $y.gauche \neq NIL$ **alors**
  $y.gauche.parent \leftarrow x$
$y.parent \leftarrow x.parent$
si $x.parent = NIL$ **alors**
  $T.racine \leftarrow y$
sinon si $x = x.parent.gauche$ **alors**
  $x.parent.gauche \leftarrow y$
sinon
  $x.parent.droite \leftarrow y$
$y.gauche \leftarrow x$
$x.parent \leftarrow y$



- Algorithme qui préserve la propriété des arbres binaires de recherche.
- Même principe pour Rotation-Droite (on permute “gauche” et “droite”).
- Méthode en $\Theta(1)$ qui s’applique sur le plus haut nœud.

Exemple de rotation



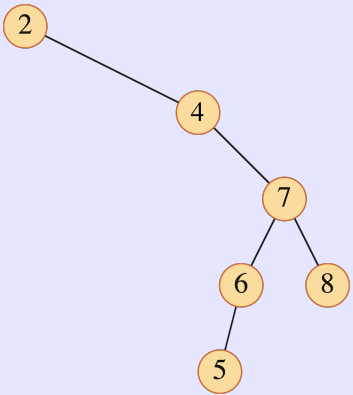
Distance rotationnelle

Curiosité

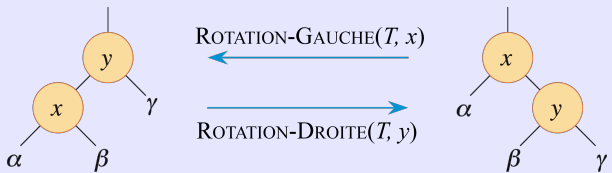
- Problème : quelle est la séquence de rotations la plus courte (i.e. *distance rotationnelle*) pour passer d'un arbre donné à un autre ?
- Question ouverte : problème dont on ne sait ni s'il est NP-Complet ($\approx$ pas polynomial), ni s'il est résoluble en temps polynomial. Même situation qu'avec le problème de la factorisation en facteurs premiers.
- Mais il existe un algorithme polynomial qui génère une séquence de rotations qui est au plus 2 fois plus longue que la séquence minimale.

Question

Quelle séquence de rotations permet d'obtenir un arbre ayant la plus petite hauteur à partir de l'arbre suivant ?

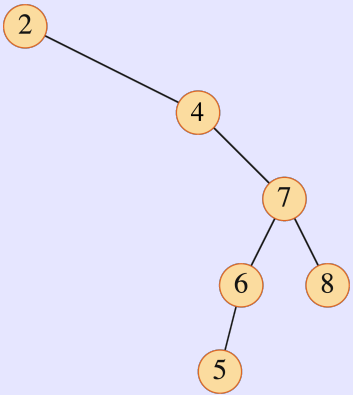


1. Gauche(4), Gauche(2), Gauche(2), Droite(7), Droite(7)
2. Droite(6), Droite(7), Gauche(2), Gauche(4), Gauche(5)
3. Droite(7), Droite(6), Gauche(2), Gauche(4), Gauche(5)
4. Gauche(2), Droite(7), Droite(6), Gauche(4), Gauche(6)



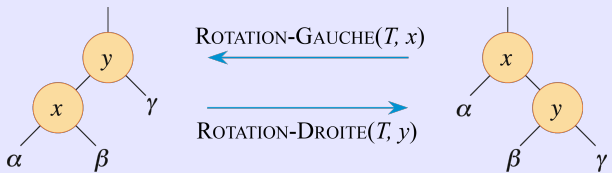
Question

Quelle séquence de rotations permet d'obtenir un arbre ayant la plus petite hauteur à partir de l'arbre suivant ?



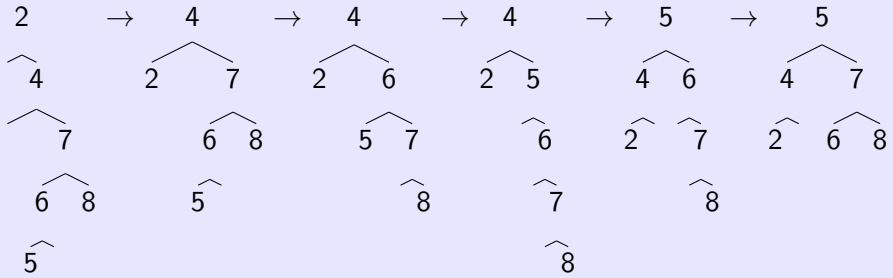
1. Gauche(4), Gauche(2), Gauche(2), Droite(7), Droite(7)
2. Droite(6), Droite(7), Gauche(2), Gauche(4), Gauche(5)
3. Droite(7), Droite(6), Gauche(2), Gauche(4), Gauche(5)

✓ Gauche(2), Droite(7), Droite(6), Gauche(4), Gauche(6)



Correction

Gauche(2), Droite(7), Droite(6), Gauche(4), Gauche(6) :



Plan

RN-Insérer

RN-Insérer(T, z)

```
y ← T.nil
x ← T.racine
tant que x ≠ T.nil faire
    y ← x
    si z.clé < x.clé alors
        x ← x.gauche
    sinon
        x ← x.droite
z.parent ← y
si y = T.nil alors
    T.racine ← z
sinon si z.clé < y.clé alors
    y.gauche ← z
sinon
    y.droite ← z
```

```
z.gauche ← T.nil
z.droite ← T.nil
z.couleur ← Rouge
RN-Insérer-Correction(T, z)
```

- Même principe que Arbre-Insérer.
- Remplacement de *NIL* par $T.nil$.
- Coloration de z.
- Appel à RN-Insérer-Correction pour rétablir les propriétés des arbres rouge-noir.

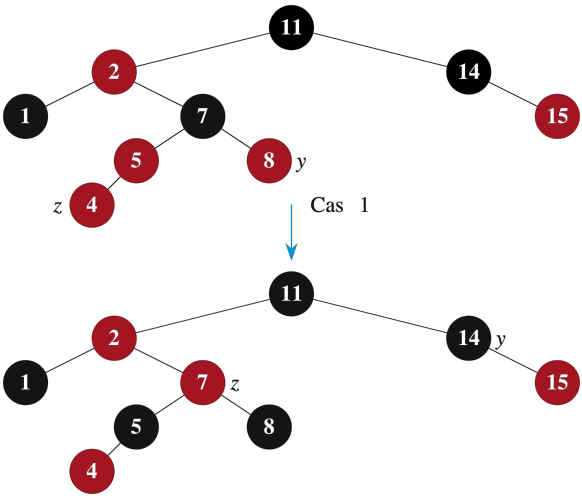
RN-Insérer-Correction

RN-Insérer-Correction(T, z)

```
tant que z.parent.couleur = Rouge faire
  si z.parent = z.parent.parent.gauche alors
    y ← z.parent.parent.droite
    si y.couleur = Rouge alors                          // Cas 1
      z.parent.couleur ← Noir
      y.couleur ← Noir
      z.parent.parent.couleur ← Rouge
      z ← z.parent.parent
    sinon
      si z = z.parent.droite alors                      // Cas 2
        z ← z.parent
        Rotation-Gauche(T, z)
      z.parent.couleur ← Noir                           // Cas 3
      z.parent.parent.couleur ← Rouge
      Rotation-Droite(T, z.parent.parent)
  sinon // même chose en permutant "gauche" et "droite"
T.racine.couleur ← Noir
```

- On est soit dans le cas 1, soit dans le cas 3 (le cas 2 est une étape optionnelle du cas 3).
- Dans le cas 1, la boucle peut continuer en remontant l'arbre.
- Dans le cas 3, la boucle termine.

Exemple d'insertion : cas 1



Conditions :

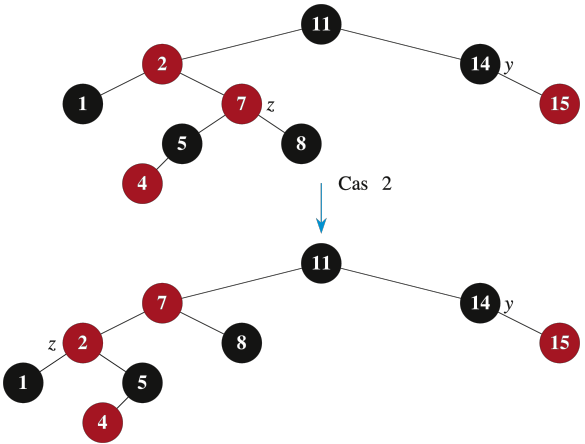
- $z.parent.couleur = \text{ROUGE}$
- $y.couleur = \text{ROUGE}$ (y : l'oncle de z)

Actions :

$z.parent.couleur \leftarrow \text{NOIR}$
$y.couleur \leftarrow \text{NOIR}$
$z.parent.parent.couleur \leftarrow \text{ROUGE}$
$z \leftarrow z.parent.parent$

La "noirceure" du grand-parent est distribuée au parent et à l'oncle et on remonte dans l'arbre.

Exemple d'insertion : cas 2



Conditions :

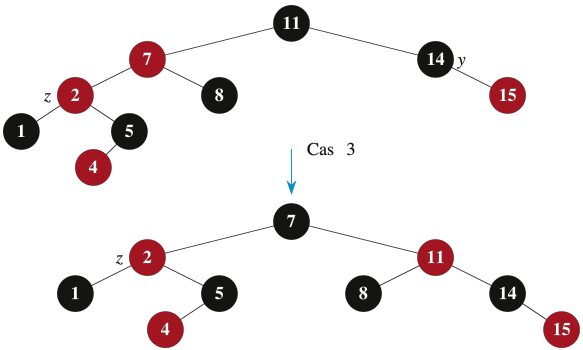
- $z.parent.couleur = \text{ROUGE}$
- $y.couleur = \text{NOIR}$ (y est l'oncle de z)
- z est encadré par son parent et son oncle

Actions :

$z \leftarrow z.parent$
ROTATION-GAUCHE(T, z)

Le parent est encadré par z et l'oncle (condition du cas 3).

Exemple d'insertion : cas 3



Conditions :

- $z.parent.couleur = \text{ROUGE}$
- $y.couleur = \text{NOIR}$ (y est l'oncle de z)
- $z.parent$ est encadré par z et l'oncle

Actions :

$z.parent.couleur \leftarrow \text{NOIR}$
$z.parent.parent.couleur \leftarrow \text{ROUGE}$
ROTATION-DROITE$(T, z.parent.parent)$

On rétablit les propriétés des arbres rouge-noir.

RN-Insérer-Correction

RN-Insérer-Correction(T, z)

```
tant que z.parent.couleur = Rouge faire
  si z.parent = z.parent.parent.gauche alors
    y ← z.parent.parent.droite
    si y.couleur = Rouge alors                          // Cas 1
      z.parent.couleur ← Noir
      y.couleur ← Noir
      z.parent.parent.couleur ← Rouge
      z ← z.parent.parent
    sinon
      si z = z.parent.droite alors                      // Cas 2
        z ← z.parent
        Rotation-Gauche(T, z)
      z.parent.couleur ← Noir                           // Cas 3
      z.parent.parent.couleur ← Rouge
      Rotation-Droite(T, z.parent.parent)
  sinon // même chose en permutant "gauche" et "droite"
T.racine.couleur ← Noir
```

- On est soit dans le cas 1, soit dans le cas 3 (le cas 2 est une étape optionnelle du cas 3).
- Dans le cas 1, la boucle peut continuer en remontant l'arbre.
- Dans le cas 3, la boucle termine.

Preuve de validité (invariant)

À chaque début d'itération de la boucle principale :

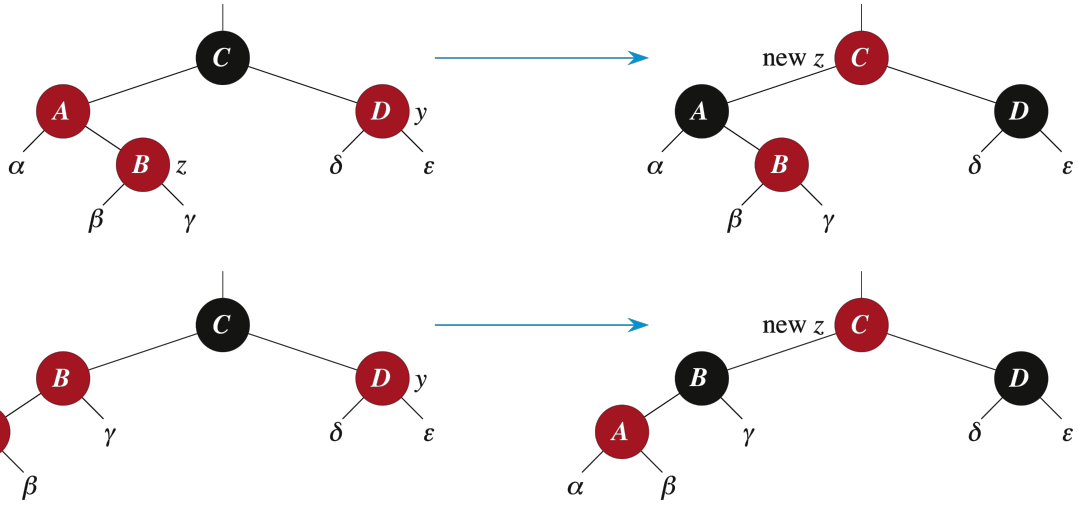
1. z est rouge.
2. Si $z.parent$ est la racine, alors $z.parent$ est noir.
3. Il y a au plus une propriété non respectée :
   - Propriété 2 : z est une racine rouge, ou
   - Propriété 4 : z et $z.parent$ sont tous les deux rouges.

L'initialisation et la terminaison étant assez directes, on se focalise sur la conservation qui permet de revoir chaque cas.

Propriétés des arbres rouge-noir

1. Chaque nœud est soit noir, soit rouge.
2. La racine est noire.
3. Chaque feuille $T.nil$ est noire.
4. Si un nœud est rouge, alors ses enfants sont noirs (il n'y a donc jamais deux nœuds rouges successifs sur un chemin de la racine à une feuille).
5. Pour chaque nœud, tous les chemins d'un nœud à ses feuilles descendantes ont le même nombre de nœuds noirs.

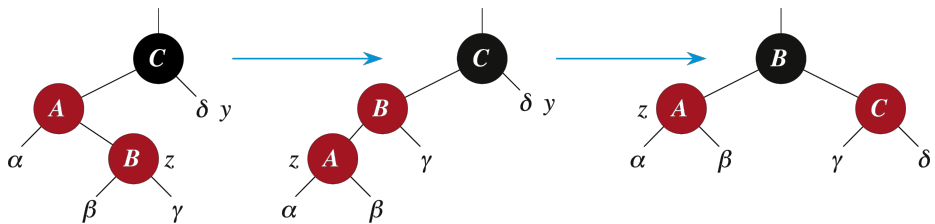
Cas 1



Cas 1

- Le parent et l'oncle de z sont rouges.
- On distribue la "noirceure" du grand-parent à ses enfants.
- On remonte dans l'arbre de deux niveaux.
- Maintien de l'invariant :
  1. $z.parent.parent$ est colorié en rouge et sera le nouveau z.
  2. La couleur du nouveau $z.parent$ ne change pas. S'il s'agit de la racine, elle était déjà noire.
  3. La distribution maintient toutes les propriétés sauf éventuellement la 2 (z peut être une racine rouge) si le nouveau z est la racine ou la 4 (z et $z.parent$ peuvent être tous les deux rouges) sinon.

Cas 2 et 3



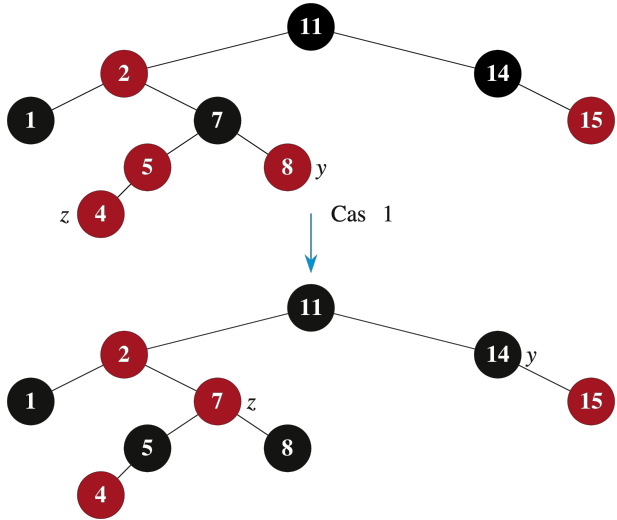
Cas 2 et 3

- Le parent de z est rouge mais l'oncle est noir.
- Le cas 2 ramène l'arbre dans la forme nécessaire pour le cas 3.
- Le sous-arbre enraciné en z est trop profond. La rotation ré-équilibre l'arbre.
- Maintien de l'invariant :
  1. Le cas 2 transforme z en $z.parent$ qui est rouge.
  2. $z.parent$ est colorié en noir, ce qui valide l'invariant s'il s'agit de la racine.
  3. Comme l'oncle de z est noir, on peut colorier son parent en rouge. Les autres propriétés sont maintenues.

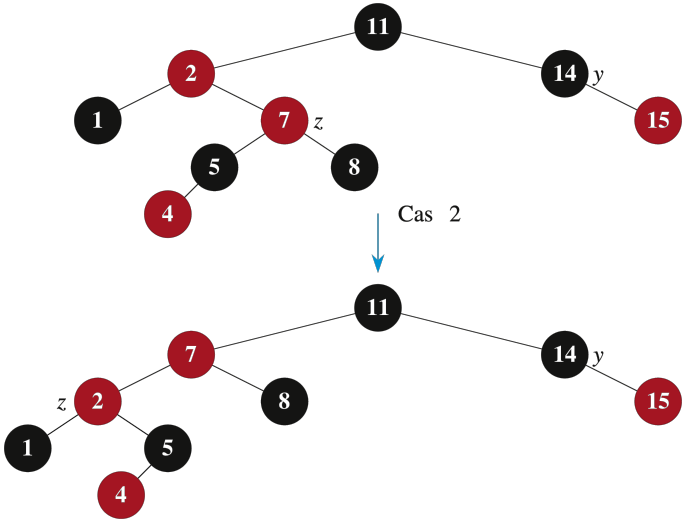
Analyse de complexité

- La hauteur d'un arbre rouge-noir est en $O(\log n)$.
- La première phase de l'insertion descend dans l'arbre.
- La phase de correction remonte au plus jusqu'à la racine (deux niveaux à la fois).
- Chaque étape prend un temps contant.
- En tout, un maximum de deux rotations sont réalisées.

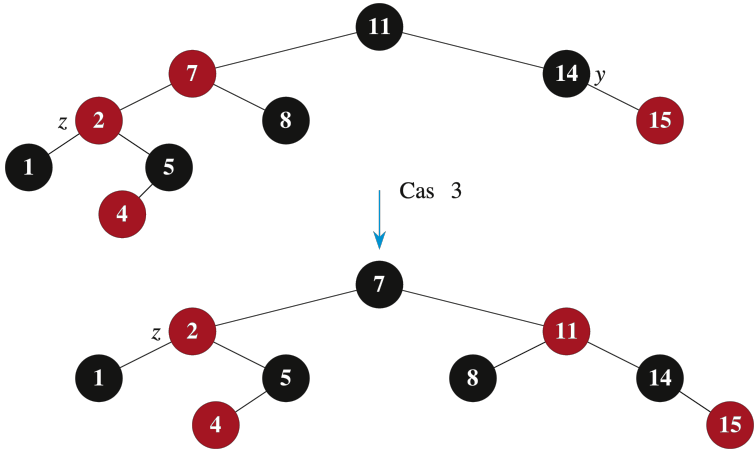
Exemple d'insertion



Exemple d'insertion



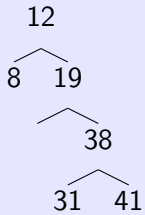
Exemple d'insertion



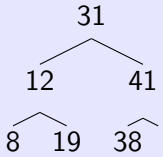
Question

Quel est l'arbre rouge-noir qui résulte de l'insertion successive des clés 41, 38, 31, 12, 19, 8 dans un arbre rouge-noir initialement vide ?

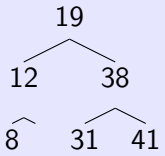
1.



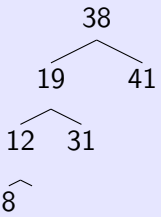
2.



3.



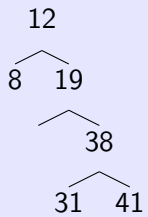
4.



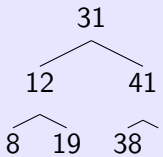
Question

Quel est l'arbre rouge-noir qui résulte de l'insertion successive des clés 41, 38, 31, 12, 19, 8 dans un arbre rouge-noir initialement vide ?

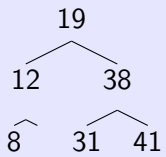
1.



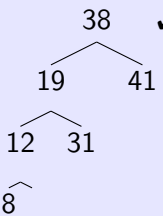
2.



3.

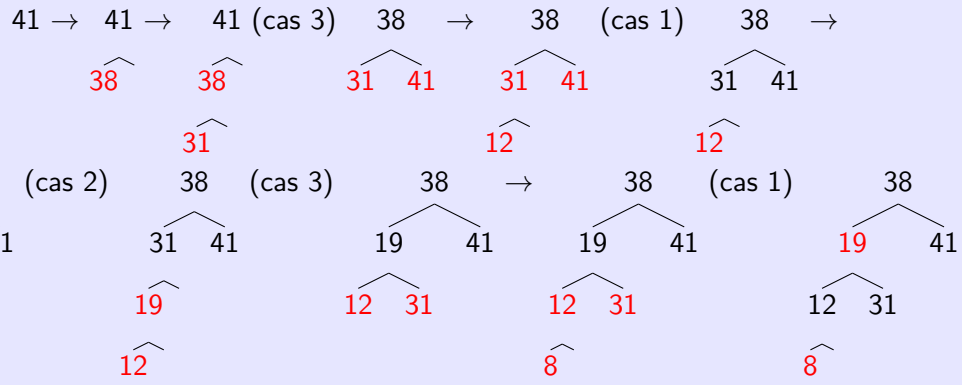


4. ✓



Correction

Clés 41, 38, 31, 12, 19, 8 :



Plan

Suppression

- Même stratégie qu'avec l'insertion : on part de l'algorithme de base sur les arbres binaires de reche et on corrige l'arbre en différenciant les cas possibles.
- L'algorithme de base est plus complexe pour la suppression que pour l'insertion.
- L'algorithme de correction considère quatre cas pour la suppression (un de plus que pour l'insertion).

Tri arborescent (ou *treesort*)

- Avec un arbre binaire de recherche classique, équivalent à un tri rapide (mais pas en place) : la première valeur insérée correspond au premier pivot et la complexité dans le pire cas est en $O(n^2)$.
- Utilisation d'un arbre auto-équilibré : pire cas en $O(n \log n)$.
- Adapté pour du tri en ligne : après avoir considéré les i premiers éléments, on peut les obtenir dans un ordre trié avec un parcours infixe en $O(i)$.

Alternatives

- Autre structure très proche et très populaire : les *arbres AVL*.
- Structure qui garantit que la hauteur des enfants de chaque nœud diffère au plus de 1.
- L'arbre est généralement plus équilibré qu'un arbre rouge-noir, mais les modifications nécessitent plus d'opérations de ré-équilibrage (à base de rotation à nouveau).
- Les arbres AVL sont plus pertinents s'il y a essentiellement des requêtes de recherche et peu de requêtes de modification.
- Mais aussi les arbres AA, arbres bouc-émissaire, les arbretas, les arbres de Van Emde Boas, etc.

Implémentations

- Java : `TreeMap` et `TreeSet`.
- C++ STL : `std::map` et `std:set`.
- Noyau Linux : *Completely Fair Scheduler*, `linux/rbtree.h`.
- Python : `SortedDict` et `SortedSet` s'appuient en revanche sur une structure d'arbre plus proche du tas.

Plan

Résumé

Contenu

- Les arbres rouge-noir sont des arbres de recherche équilibrés (hauteur en $O(\log n)$).
- Chaque nœud est coloré, la racine est noire, les feuilles sentinelles $T.nil$ sont noires, un nœud rouge n'a pas d'enfant rouge et la racine a une hauteur noire bien définie.
- La rotation est l'opération de base pour re-structurer un arbre binaire de recherche.
- Les opérations d'insertion et de suppression nécessitent une méthode complémentaire pour corriger les propriétés en considérant plusieurs cas.