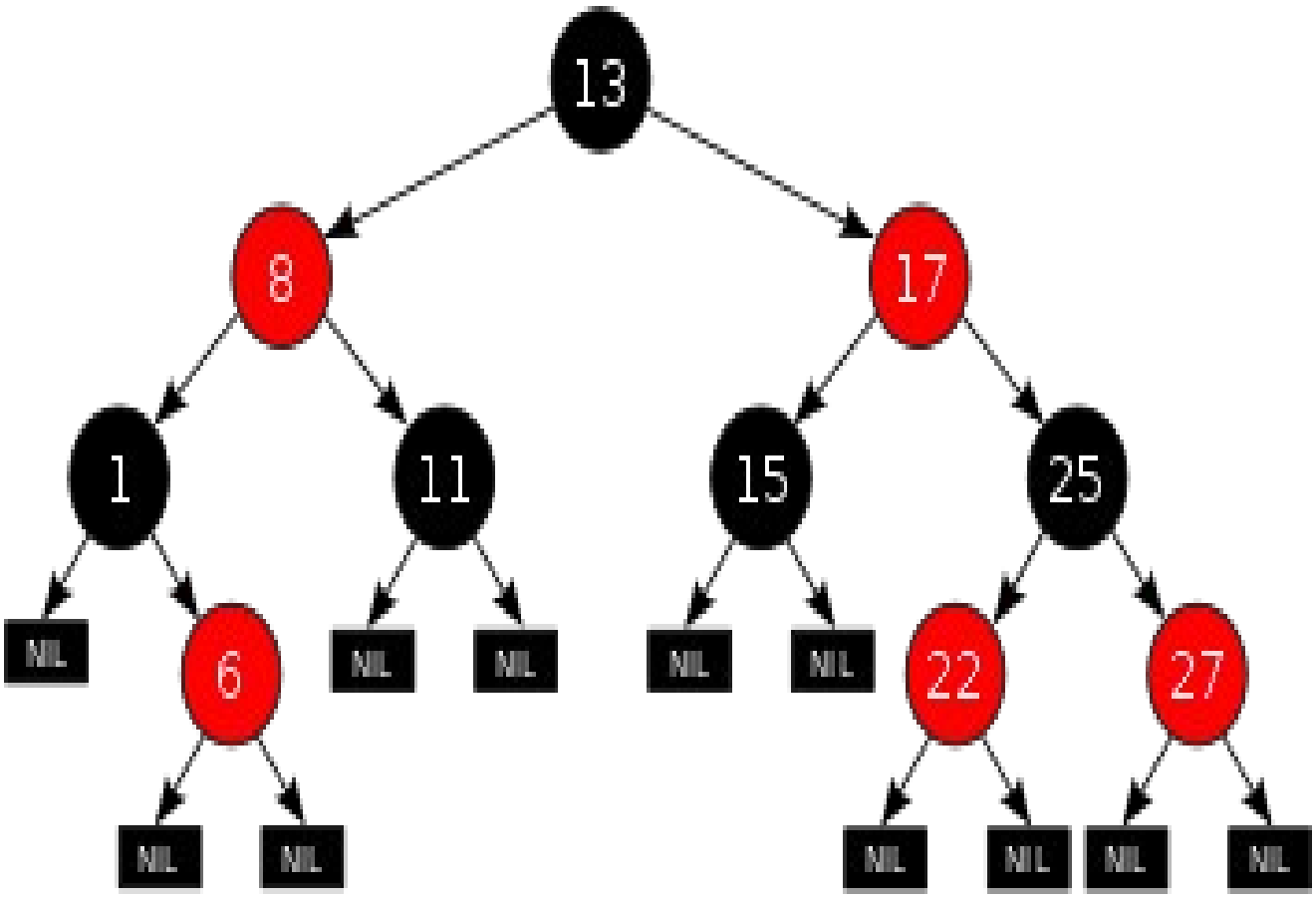
UNIVERSITÉ LE HAVRE NORMANDIE

Licence 3 Informatique
2019-2020

KOUCHE Rabia

Arbres rouges et noirs

Encadré par : Veronique Jay



-- *image 1* --

objectif :

L’objectif de ce rapport est d’expliquer l’implémentation des arbres rouge et noir, ils sont une variante des arbres binaires de recherche. Leur intérêt principal est qu'ils sont relativement équilibrés.

1-**Définition :**

Un *arbre rouge et noir* est un arbre binaire de recherche ou chaque nœud est de couleur rouge ou noire de telle sorte que

1. Chaque nœud est soit noir soit rouge,
2. La racine est noire,
3. Pour tout nœud x, tout les chemins reliant x a une feuille contiennent le même nombre de nœuds noirs,
4. les feuilles sont noires,
5. si un nœud est rouge ses deux fils sont noirs.

Le sentinelle : c’est un nœud feuille qui est à nul dont tout les nœud qui pointent sur une feuille et compris la racine pointeront sur un sentinelle unique.
clef : est un attribut qui représente le contenu ou l’information d’un nœud.
left : représente le nœud du fils gauche du nœud courant.
right : représente le nœud du fils droit du nœud courant.
père : représente nœud père du nœud courant.
clr : représente la couleur du nœud soit couleur noir ou rouge.

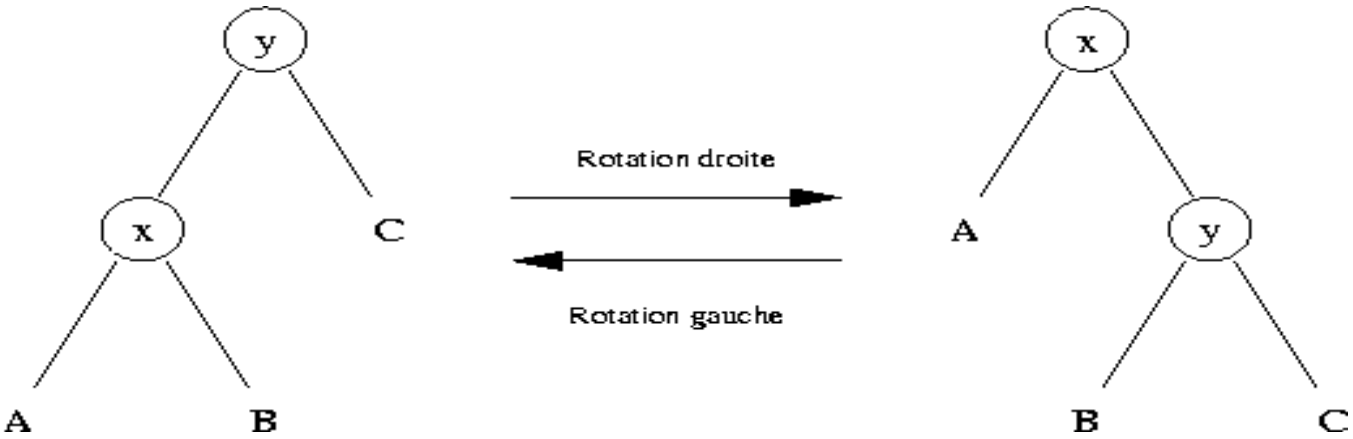
2- **Programmation :**

Pour représenter un arbre binaire rouge-noir nous utilisons une classe *Nœud* qui permet de représenter la couleur de l’arbre, et un attribut permettant de référencer son *Nœud* père. Pour les besoins de l’algorithme de suppression, chaque feuille au lieu d’avoir une référence n*ull* vers son fils gauche et son fils droit, aura une référence vers un nœud statique sentinelle de couleur noire.

- **Rotations :**

Les rotations sont des modifications locales d'un arbre binaire. Elles consistent à échanger un nœud avec l'un de ses fils. Les rotations gauche et droite sont inverses l'une de l'autre. Elle sont illustrées à la figure ci-dessous. Dans les figures, les lettres majuscules comme A, B et C désignent des sous-arbres.

Les rotations préservent les propriétés des ABR.



– image 2 --

private void Rotation_Droite(Nœud nœud);

Dans la rotation droite, un nœud devient le fils droit du nœud qui était son fils gauche.

private void Rotation_Gauche(Noeud nœud);

Dans la rotation gauche, un nœud devient le fils gauche du nœud qui était son fils droit.

Les opérations caractéristiques sur les Arbre rouge et noir sont l'insertion la suppression et la recherche d'une clé dans un arbre R/N.

2-1 Recherche d'un nœud dans un Arbre :

private Noeud Recherche(Object object);

La recherche d'un nœud dans un arbre rouge et noir commence par l'insertion usuelle d'une valeur dans un arbre binaire de recherche. On part de la racine, c'est la clé est inférieur à la clé du nœud courant on va à la branche gauche, sinon on regarde la branche droite, on arrête la recherche si la clé a été trouvé ou si une feuille est atteinte, la valeur recherchée ne se trouve pas dans l'arbre.

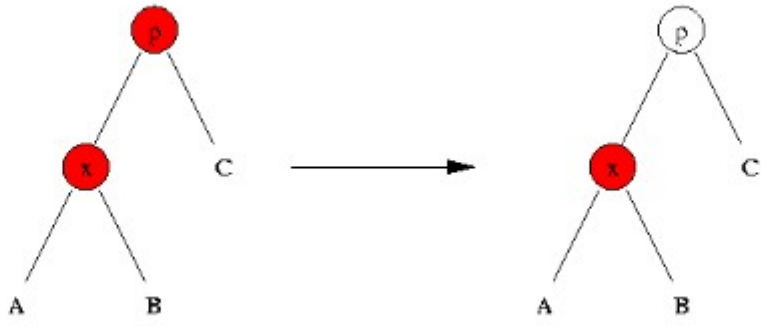
2-2 Insertion d'un nœud dans un Arbre :

public void Insertion(Noeud c);

L'insertion d'un nœud dans un ABR se fait de la même manière que la recherche d'un élément, on criée un nouveau nœud, on fait la recherche jusqu'à ce que on rencontre le sentinelle on arrête la recherche on insert l'élément criée.

il faut vérifier que les propriétés de l'arbre rouge et noir sont respectées et, dans le cas contraire, on effectue les opérations de changement de couleur et des rotations pour les rétablir.

Cas 1 : le nœud père p est la racine de l'arbre :



Dans ce cas ou p n'as pas de père, donc on le colorie en noir pour que la propriété 2 soit vérifiée.

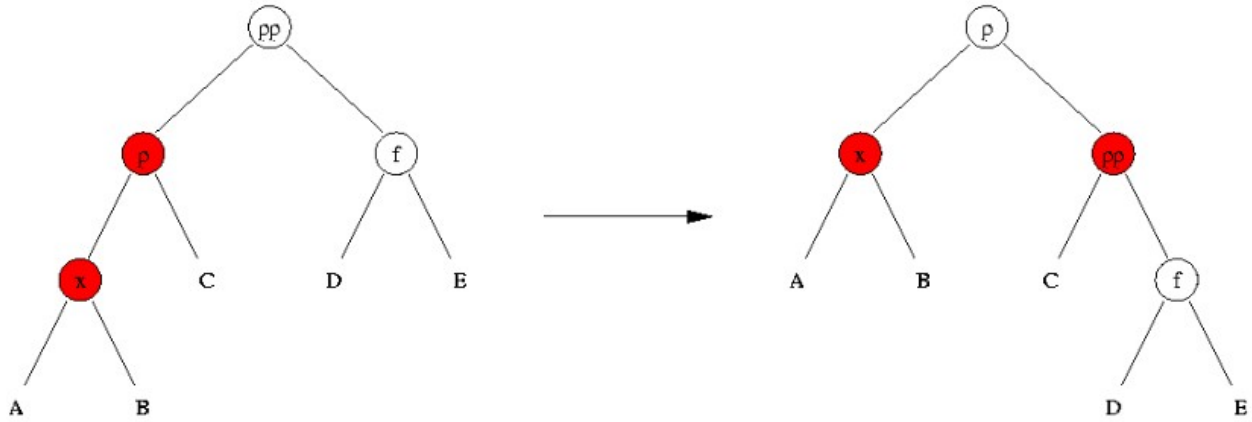
Cas 2 : Si le parent du nœud inséré est noir :

Dans ce cas l'arbre est correct rien à changer, puisque le nouveau nœud est bien rouge, et la propriété 3 est vérifiée.

Cas 3 : le frère f de p est noir :

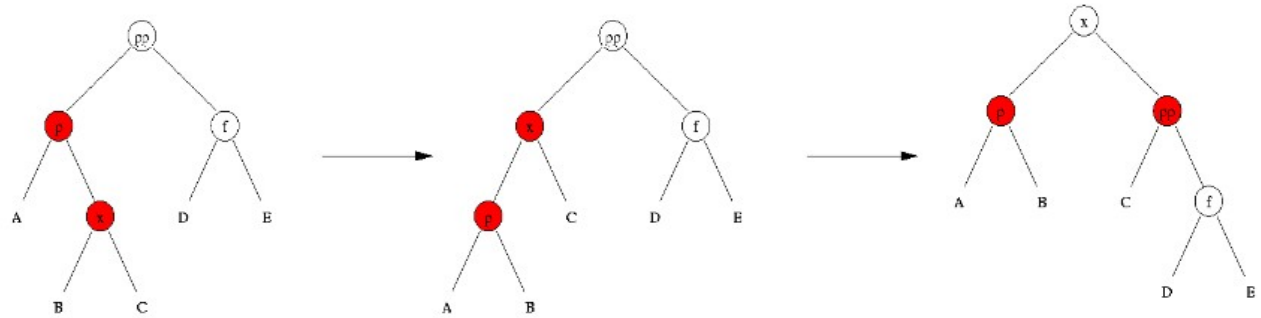
on distingue deux cas suivant que x est le fils gauche ou le fils droit de p :

cas a : quand x est le fils gauche de p .



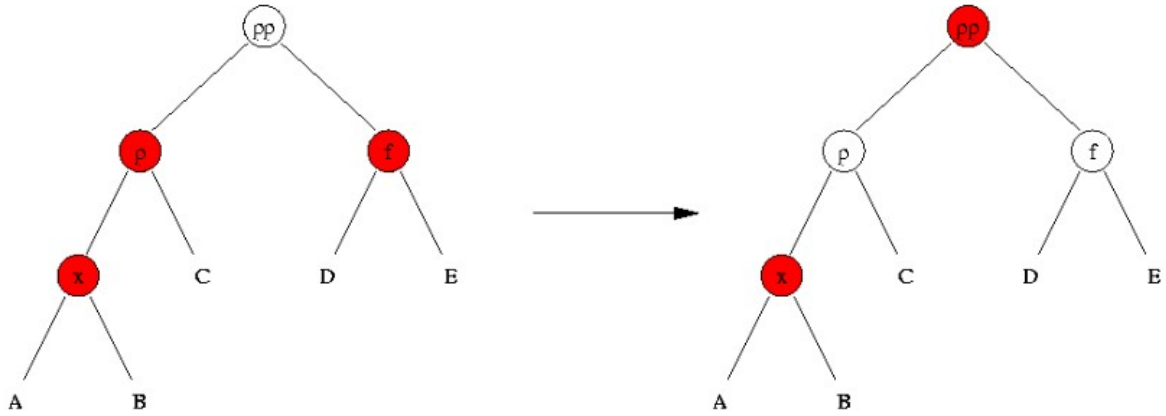
On effectue la rotation à droite et pp devient le fils gauche de p, ensuite on colorie p pour qu'il devient noir, et pp va devenir rouge, On arrête l'algorithme vu que les propriétés (2) et (3) sont vérifiée.

cas b : quand x est le fils droit de p.



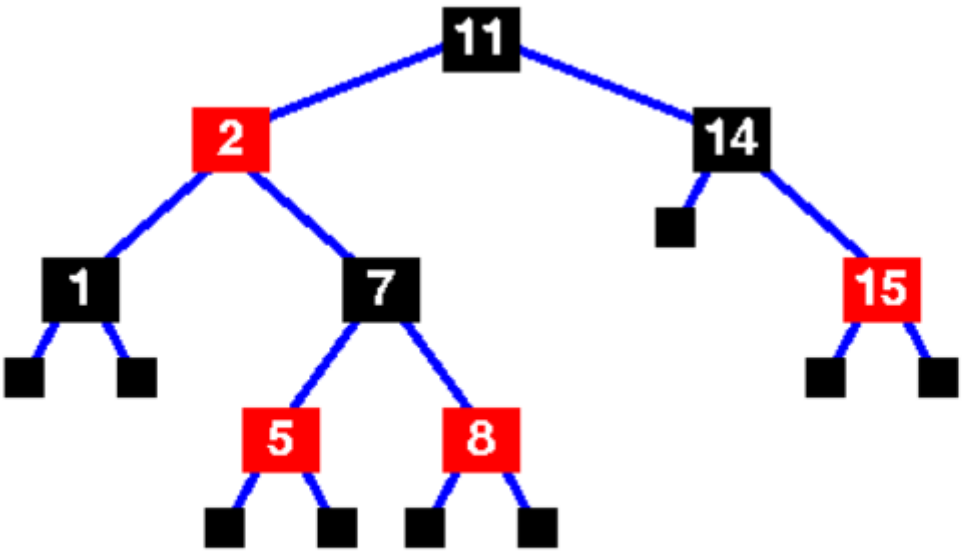
D’abord, On fait rotation à gauche entre x et p pour que p deviennent le fils gauche de x. Là on est dans le cas précédent et l’algo fait une rotation à droite entre x et pp. Ensuite x devient noir et pp devient rouge. c’est la fin de l’algo puisque les propriétés (2) et (3) sont vérifiées.

Cas 4 : le frère f de p est rouge :

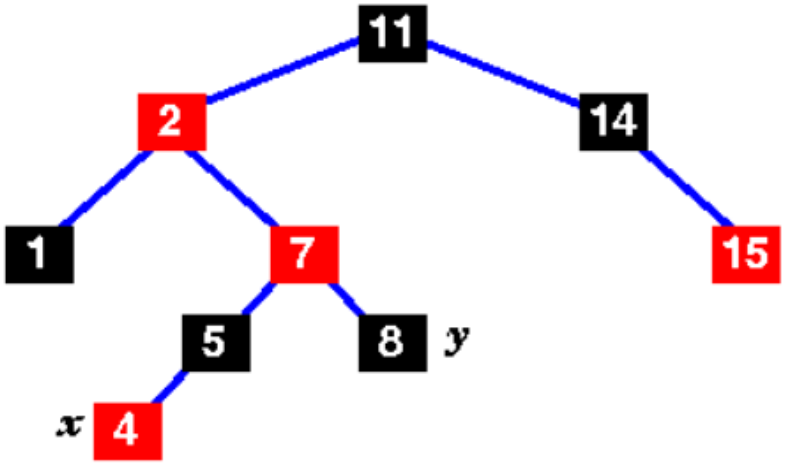


Les nœuds p et f deviennent noirs et leur père pp devient rouge. La propriété (3) reste vérifiée.

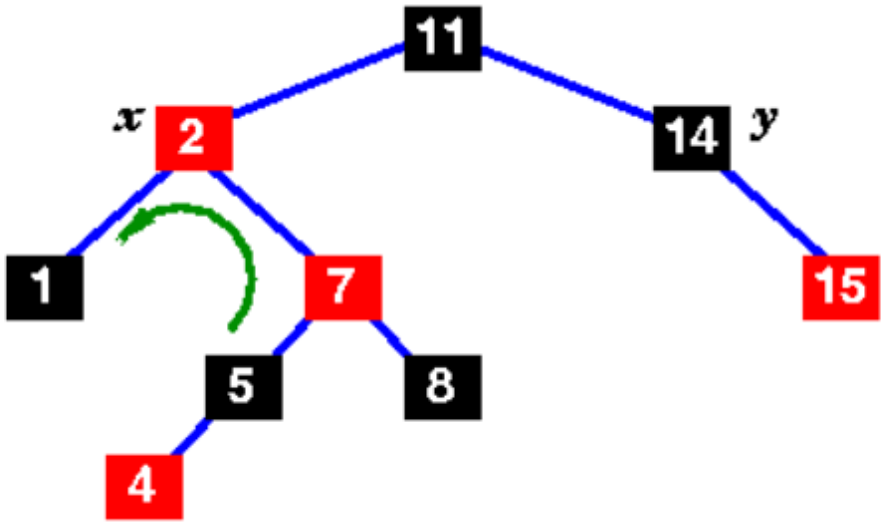
Exemple :



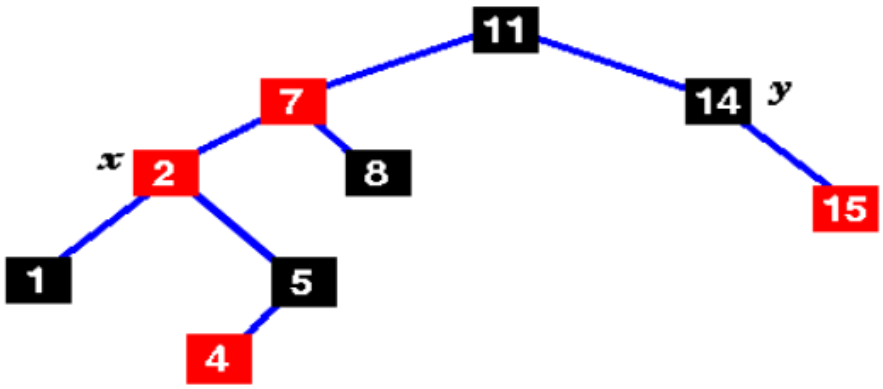
Arbre de départ



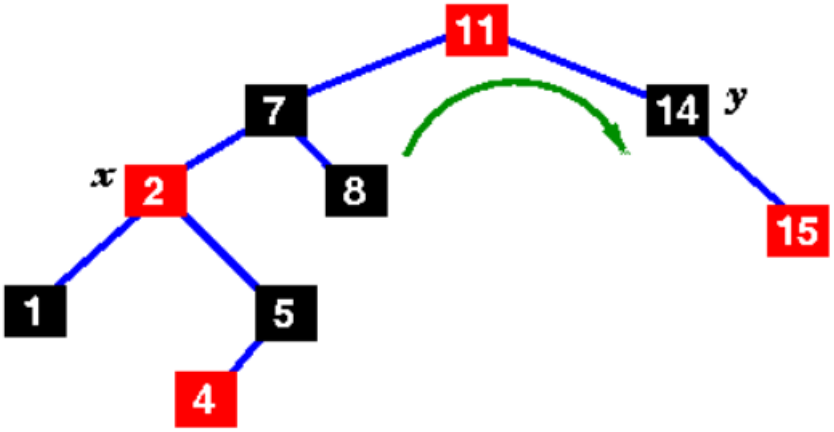
Ici on a deux nœuds qui sont rouge (2) et (7), ce n'est plus un arbre rouge et noire.



Prendre x en haut et faire une rotation gauche.

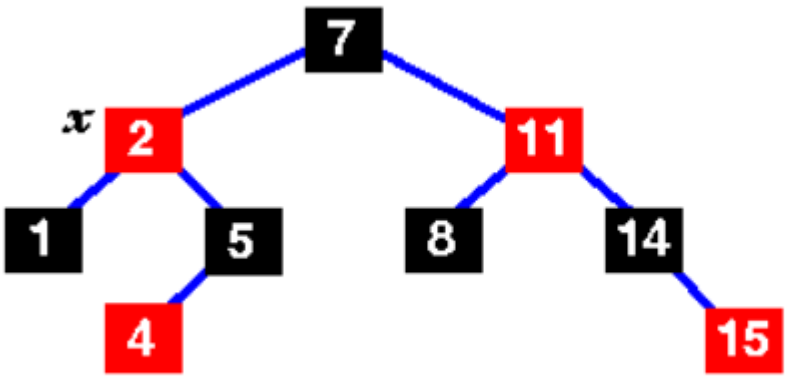


Pas encore rouge et noir



9 Change de couleur à 7 et 11 et faire une rotation droite

-



Maintenant, l'arbre est rouge et noir.

Suppression d'un nœud dans un Arbre :

private Noeud supprimer(Noeud z);

La suppression commence par une recherche du nœud à supprimer, comme dans un arbre binaire classique.

L'algo distingue plusieurs cas :

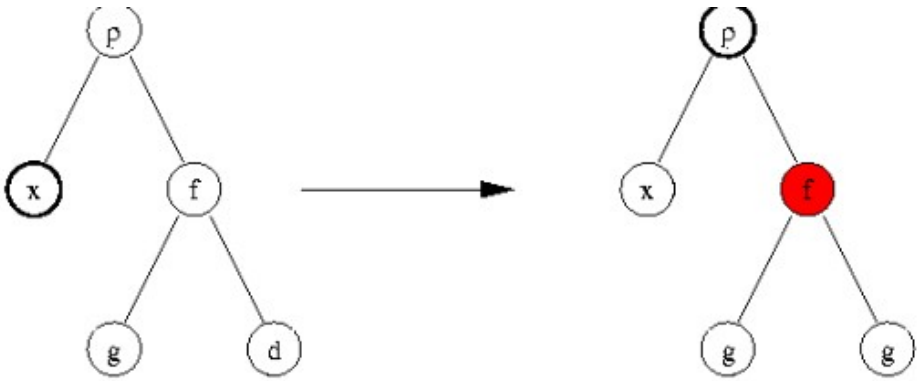
Cas 0 : le nœud x est la racine de l'arbre :

Le nœud x devient noir. La propriété (2) est vérifiée ainsi que la propriété (3).

Cas 1 : le frère f de x est noir :

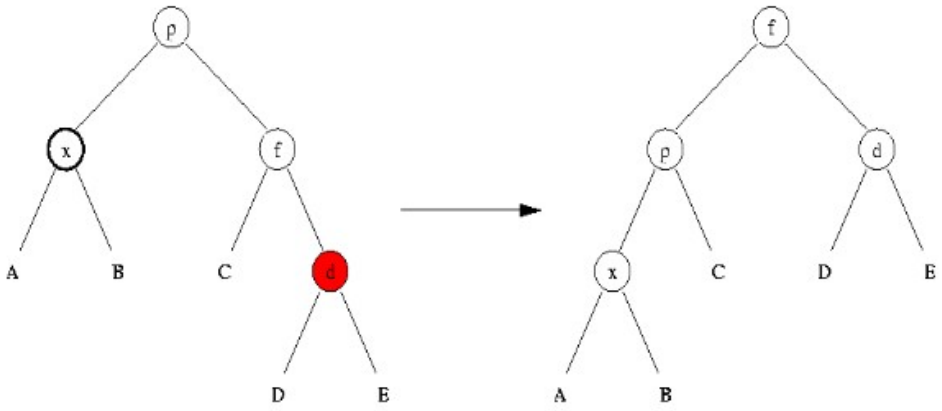
L'algorithme distingue à nouveau trois cas :

Cas 1a : les deux fils g et d de f sont noirs :



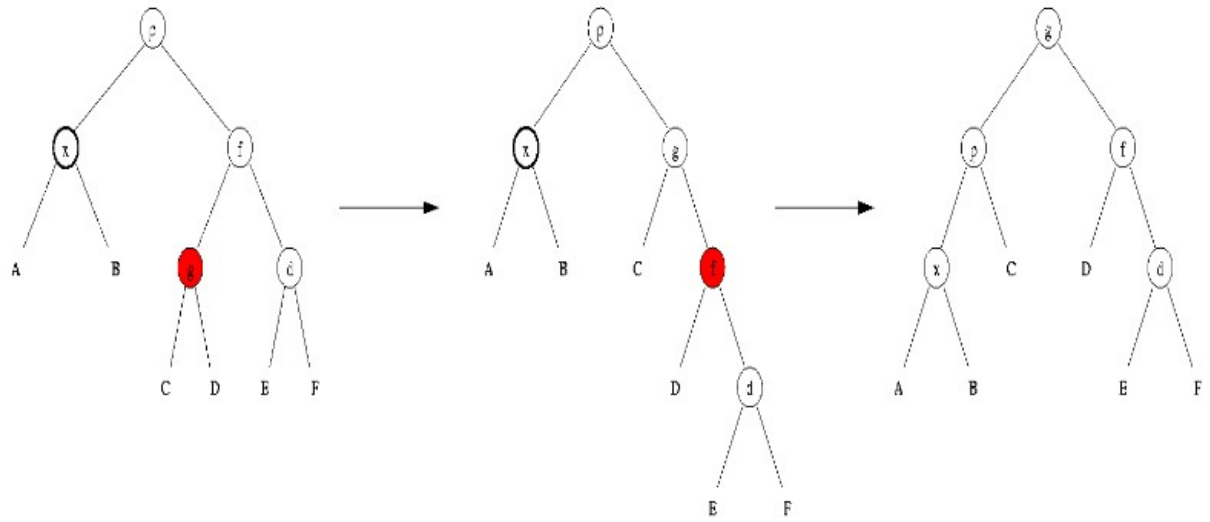
Le nœud f devient rouge, Le nœud p devient noir sil était déjà rouge.

Cas 1b : le fils droit d de f est rouge :

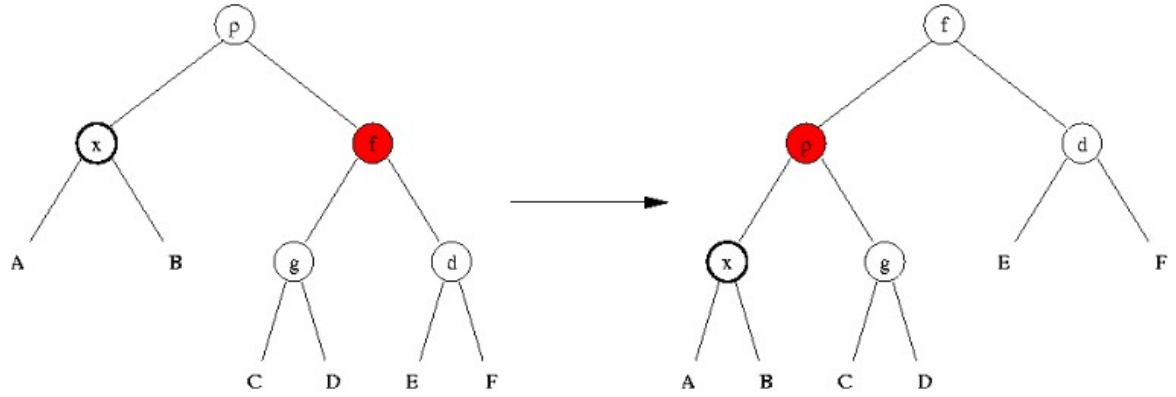


Dans ce cas l'algorithme fait une rotation à droite entre f et p, ensuite f prend la couleur de p, et d devient rouge.

Cas 1c : le fils droit d est noir et le fils gauche g est rouge :



Rotation à gauche entre f et g, Le nœud g devient noir et le nœud f devient

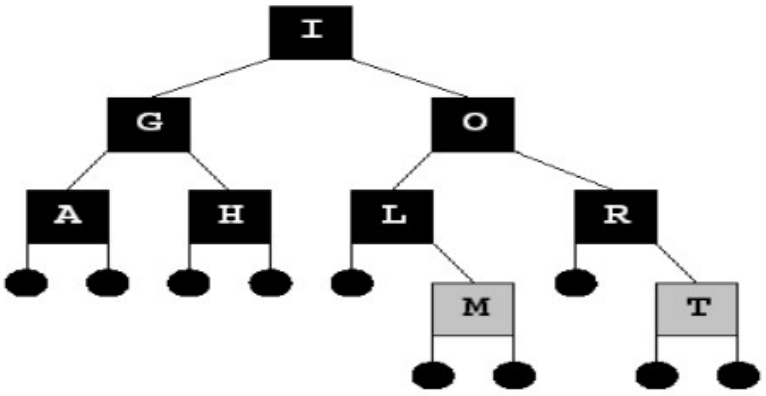


rouge, L'algorithme effectue ensuite une rotation entre p et g. Le nœud f redevient noir et l'algorithme se termine.

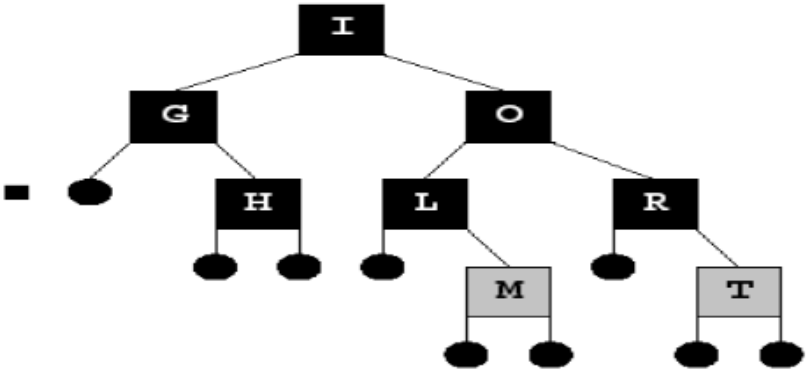
Cas 2 : le frère f de x est rouge :

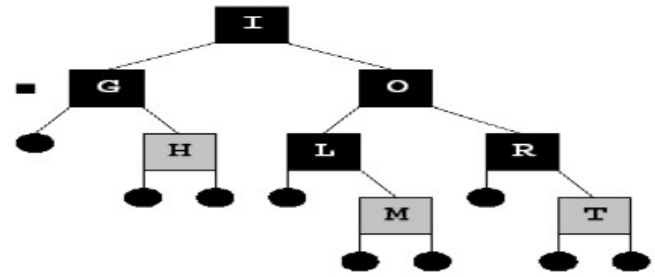
L'algorithme effectue alors une rotation gauche entre p et f,Ensuite p devient rouge et f devient noir.

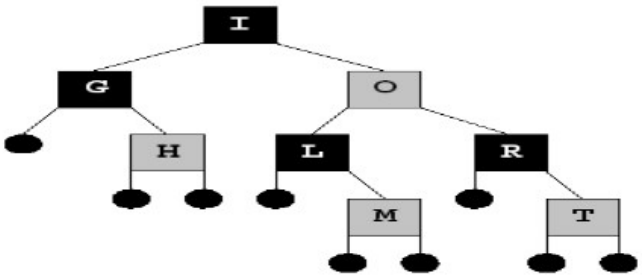
Exemple :



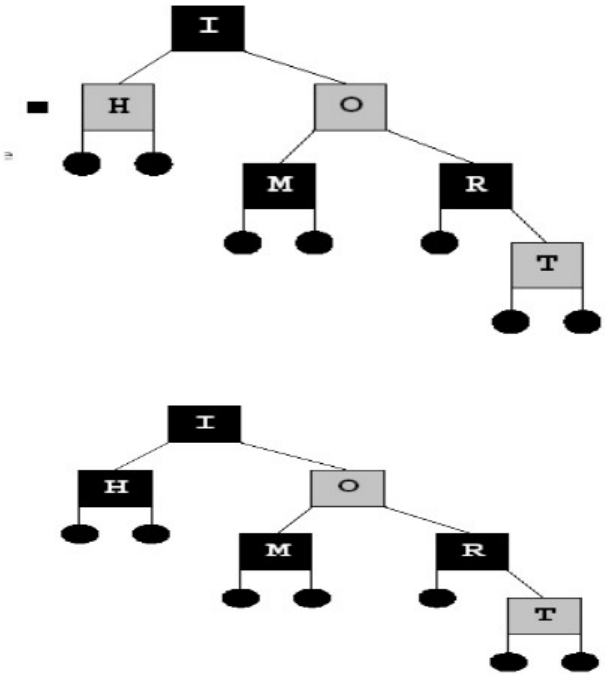
Supprimer A







supprimer G



2-4 L’organisation de l’arbre :

Après avoir insérer ou supprimer un nœud dans l’arbre on appel la fonction de l’organisation pour l’équilibrage de l’arbre :

private void Organiser_Supp(Noeud n);

Organisation de l'arbre après la suppression pour que les propriétés de l'arbre ne soient pas violées.

private void Organiser(Noeud n);

Organisation de l'arbre après l'insertion pour que les propriétés de l'arbre ne soient pas violées.

2-5 Interfaces Iterator et Comparator :

public Iterator<E> iterator();

E - le type d'éléments retournés par cet itérateur,

La méthode retourne un itérateur sur l'arbre courant, elle nous permet de parcourir l’Arbre.

2-5-2 La surcharge de quelques méthodes qui héritent de l’interface Iterator:

public boolean hasNext();

Renvoie vrai si l'itération a plus de nœuds sinon elle renvoie false.

public E next();
Renvoie le nœud suivant de l'itération.

public void remove();
Supprime de l'Arbre le dernier nœud renvoyé par cet itérateur .

public class ARBRE_RN<E> extends AbstractCollection<E>;

public ARBRE_RN();
Crée un arbre vide. Les éléments sont ordonnés selon l'ordre naturel.

public ARBRE_RN(Comparator<? super E> cmp);
Crée un arbre vide. Les éléments sont comparés selon l'ordre imposé par
le comparateur utilisé pour définir l'ordre des éléments.

public ARBRE_RN(Collection<? extends E> c);
Constructeur par recopie. Crée un arbre qui contient les mêmes éléments que c.
L'ordre des éléments est l'ordre naturel.
parametre c la collection à copier.

public ARBRE_RN(Collection<? extends E> c);
Constructeur par recopie. Crée un arbre qui contient les mêmes éléments que c.
L'ordre des éléments est l'ordre naturel.
parametre c la collection à copier.

- Surcharge de quelques méthodes de la classe externe ARBRE_RN :
@Override
public boolean add(E e);
Ajoute l'élément spécifié à la fin de cette liste.

@Override
public boolean contrains(Object o);
Renvoie true si cette liste contient l'élément spécifié.

@Override
public int size();
retourne la taille de l'arbre.

2-6 Quelques méthodes rajouter dans la classe interne Noeud :

Noeud minimum();
Crée un noeud vide de couleur noir.
Renvoie le minumum du sous-arbre, elle retourne le noeud contenant la plus petite clé du sous arbre dont la racine est le noeud courant.

Noeud maximum();
Renvoie le minumum du sous-arbre, elle retourne le noeud contenant la plus grande clé du sous arbre dont la racine est le noeud courant.

Noeud suivant();
Renvoie le successeur du noeud courant, elle retourne le noeud contenant la clé qui suit la clé de ce noeud dans l'ordre des clés, null si c'est le noeud contenant la plus grande clé.

Constructeurs de la classe interne Noeud :

Noeud(E o);
Constructeur avec paramètre. Crée un nœud rouge avec o comme clé.

Noeud();
Crée un noeud vide de couleur noir.

Conclusion :

Le but est d'éviter de construire des arbres dits déséquilibré , Cela peut être très avantageux lorsqu'on réutilise l'arbre ainsi construit: en effet, utiliser un arbre équilibré permet d'avoir un un temps de recherche de complexité logarithmique dans le pire des cas au lieu d'une complexité linéaire, comme c'est parfois le cas pour des arbres dégénérés.

Bibliographie

1-Les images des exemples de la suppressions et l'insertion : http://www.uqac.ca/rebaine/8INF805/ArbresRougeetnoiraincl ure.pdf.
2-L'image 1 : https://upload.wikimedia.org/wikipedia/commons/thumb/6/66/

Red-black_tree_example.svg/1280px-Red-black_tree_example.svg.png

3- L’image 2 :
https://www.irif.fr/~carton/Enseignement/Algorithmique/Programmation/RedBlackTree/