Table des matières

Introduction

Le monde de l'informatique est en constante évolution et surtout à la recherche permanente de moyens pour optimiser la représentation des données en mémoire. Pour cette raison, de nombreuses structures de données ont été créées et ensuite optimisées parmi lesquelles on retrouve les arbres. Dans le cadre de ce cours, nous nous intéresserons à trois types d'arbres à savoir : les arbres binaires, les arbres binaires de recherche, et les arbres rouges-noirs. Ces structures de données peuvent supporter nombres d'opérations d'ensembles dynamiques, telles que RECHERCHER, PREDECESSEUR, SUCCESSEUR, INSERER, SUPPRIMER … on peut donc aussi bien les utiliser pour des dictionnaires, que pour des files de priorité. Dans la suite, nous allons étudier ces différents arbres ,en indiquant tout d'abord ce qu'est réellement le type d'arbre considéré, ensuite nous indiquerons aussi les propriétés de ceux-ci, et enfin nous présenterons les différentes opérations élémentaires applicables, suivies de la complexité pour chacun de ces arbres .

I- Arbre binaire

A-Définition

- Un arbre est soit un nœud, soit un **arbre vide**
- Un nœud a des fils qui sont eux aussi des arbres
- Si tous les fils d'un nœud sont vides, alors le nœud est qualifié de feuille
- Les nœuds portent des valeurs, ce sont les données que l'on veut stocker
- Si tous les nœuds de l'arbre ont n fils, alors l'arbre est dit **n-aire**

1- Définition formelle

On appelle ***arité*** d'un nœud le nombre de branches qui en partent. Dans la suite de notre cours, nous nous intéresserons plus particulièrement aux *arbres binaires*, c'est à dire ceux dont chaque nœud a au plus deux fils. Pour ne pas avoir à distinguer les nœuds suivant leur arité, il est pratique d'ajouter à l'ensemble des arbres binaires un arbre particulier appelé l'*arbre vide*. Ceci conduit à adopter la définition qui suit.

Un ensemble E étant donné, on définit par induction les arbres binaires étiquetés par E en convenant que :

– nil est un arbre binaire sur E appelé l'*arbre vide* ;

– si *x* appartient à E et si F*g* et F*d* sont deux arbres binaires étiquetés par E, alors A = (F*g* ; *x* ; Fd) est un arbre binaire étiqueté par E.

x est l'étiquette de la *racine* de A; quant à F*g* et F*d*, ils sont appelés respectivement le *sous arbre gauche* et le *sous arbre droit* de l'arbre binaire A.

De manière usuelle, on convient de ne pas faire figurer l'arbre vide dans les représentations graphiques des arbres binaires (voir la figure 2). Ainsi, suivant la représentation choisie les feuilles pourront désigner exclusivement l'arbre vide (et dans ce cas tous les nœuds seront d'arité égale à 2) ou alors les nœuds dont les deux fils sont vides (dans ce cas l'arité d'un nœud pourra être égale à 0, 1 ou 2). C'est cette seconde convention qui sera utilisée dans la suite de ce cours.

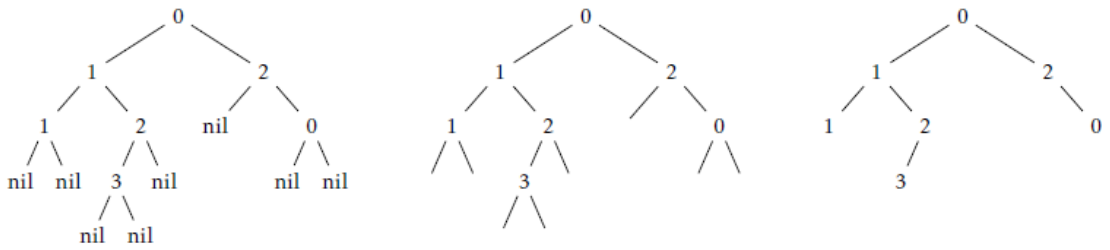


FIGURE 2 – Trois représentations du même arbre binaire.

2- Fonctions inductives

Preuve par induction structurelle

La plupart des résultats qui concernent les arbres binaires se prouvent par *induction structurelle*, c'est-à-dire en utilisant le résultat suivant :

Théorème. — *Soit $\mathscr{R}$ une assertion définie sur l'ensemble $\mathscr{A}$ des arbres étiquetés par* E. *On suppose que :*

(i) $\mathscr{R}(nil)$ *est vraie ;*

(ii) $\forall x \in \mathrm{E}, \forall (\mathrm{F}_g, \mathrm{F}_d) \in \mathscr{A}^2$, *l'implication* $\big(\mathscr{R}(\mathrm{F}_g)$ *et* $\mathscr{R}(\mathrm{F}_d)\big) \Longrightarrow \mathscr{R}(\mathrm{F}_g, x, \mathrm{F}_d)$ *est vraie ;*

Alors la propriété $\mathscr{R}(\mathrm{A})$ *est vraie pour tout arbre* A *de* $\mathscr{A}$.

De même, de nombreuses fonctions $f : \mathscr{A} \to \mathrm{F}$ se définissent par la donnée d'un élément $a \in \mathrm{F}$, d'une fonction $\varphi : \mathrm{F} \times \mathrm{E} \times \mathrm{F} \to \mathrm{F}$ et les relations :

- $f(\mathrm{nil}) = a$;
- $\forall x \in \mathrm{E}, \forall (\mathrm{F}_g, \mathrm{F}_d) \in \mathscr{A}^2, f(\mathrm{F}_g, x, \mathrm{F}_d) = \varphi(f(\mathrm{F}_g), x, f(\mathrm{F}_d))$.

Définition. — *La* taille $|\mathrm{A}|$ *d'un arbre* A *est définie inductivement par les relations :*

- $|nil| = 0$;
- *Si* $\mathrm{A} = (\mathrm{F}_g, x, \mathrm{F}_d)$ *alors* $|\mathrm{A}| = 1 + |\mathrm{F}_g| + |\mathrm{F}_d|$.

La hauteur $h(\mathrm{A})$ *d'un arbre* A *se définit inductivement par les relations :*

- $h(nil) = -1$;
- *Si* $\mathrm{A} = (\mathrm{F}_g, x, \mathrm{F}_d)$ *alors* $h(\mathrm{A}) = 1 + \max(h(\mathrm{F}_g), h(\mathrm{F}_d))$.

Avec ces conventions, $|\mathrm{A}|$ est le nombre de nœuds d'un arbre et $h(\mathrm{A})$ la longueur maximale du chemin reliant la racine à une feuille, autrement dit la profondeur maximale d'un nœud.

Théorème. — *Soit* A *un arbre binaire. Alors* $h(\mathrm{A}) + 1 \leqslant |\mathrm{A}| \leqslant 2^{h(\mathrm{A})+1} - 1$.

Preuve. On raisonne par induction structurelle.

- Si $\mathrm{A} = \mathrm{nil}$, $|\mathrm{A}| = 0$ et $h(\mathrm{A}) = -1$ et le résultat annoncé est bien vérifié.
- Si $\mathrm{A} = (\mathrm{F}_g, x, \mathrm{F}_d)$, supposons le résultat acquis pour F_g et F_d. On a :

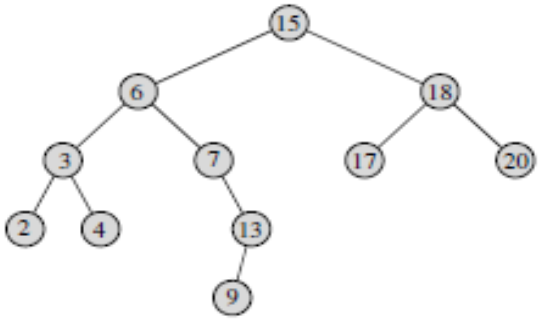
$$|\mathrm{A}| = 1 + |\mathrm{F}_g| + |\mathrm{F}_d| \geqslant 1 + h(\mathrm{F}_g) + 1 + h(\mathrm{F}_d) + 1 \geqslant 2 + \max(h(\mathrm{F}_g), h(\mathrm{F}_d)) = 1 + h(\mathrm{A})$$

$$\text{et} \quad |\mathrm{A}| = 1 + |\mathrm{F}_g| + |\mathrm{F}_d| \leqslant 2^{h(\mathrm{F}_g)+1} + 2^{h(\mathrm{F}_d)+1} - 1 \leqslant 2 \times 2^{\max(h(\mathrm{F}_g), h(\mathrm{F}_d))+1} - 1 = 2^{h(\mathrm{A})+1} - 1$$

II- Arbre binaire de recherche

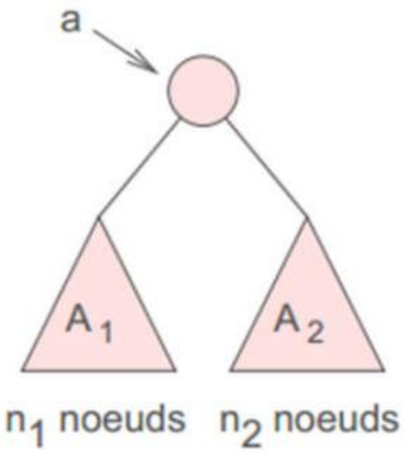
A-Définition

Un arbre binaire de recherche est un arbre binaire vérifiant la propriété suivante : soient x et y deux nœuds de l'arbre, si y est un nœud du sous arbre gauche de x, alors clé(y) ≤ clé(x), si y est un nœud du sous arbre droit de x, alors clé(y) ≥ clé(x). Chaque nœud contient le champ gauche, droite et p qui pointent sur les nœuds correspondant respectivement à son enfant de gauche, à son enfant de droite et à son parent. Si le parent, ou l'un des enfants, est absent, le champ correspondant contient la valeur NIL. Le nœud racine est le seul nœud de l'arbre dont le champ parent vaut NIL. Voici un exemple d'arbre binaire de recherche :



B- Hauteur d'un arbre binaire de recherche

Soit la figuré ci-dessous



Proposition : La hauteur h d'un arbre binaire de racine a, de n nœuds est au moins égale à **$\log_2 n$**.

Preuve : Par substitution en utilisant la propriété.

$$h(a) = max(h(A_1), h(A_2)) + 1$$

Supposons $n1 \geq n2$, et donc $n1 \geq n/2$.
Hypothèse : $h(A_1) \geq log_2 n_1$.
Donc $h(A_1) \geq log_2(n/2) = log_2 n - 1$.
Puisque $h(a) \geq h(A_1) + 1 \geq log_2 n - 1 + 1$,
on a $h(a) \geq log_2 n$.

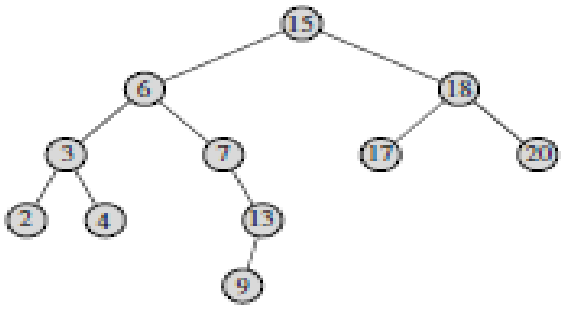
C- Parcours sur les arbres binaires de recherches

Un parcours d'arbre est une façon d'ordonner les nœuds d'un arbre afin de les parcourir. On peut le voir comme une fonction qui a un arbre associe une liste de ses nœuds même si la liste n'est souvent pas explicitement construite par le parcours. On distingue essentiellement deux types de parcours : le parcours en largeur et les parcours en profondeur. Parmi les parcours

en profondeur, on distingue à nouveau le parcours préfixe, le parcours infixe et le parcours suffixe.

a- Parcours en Largeur

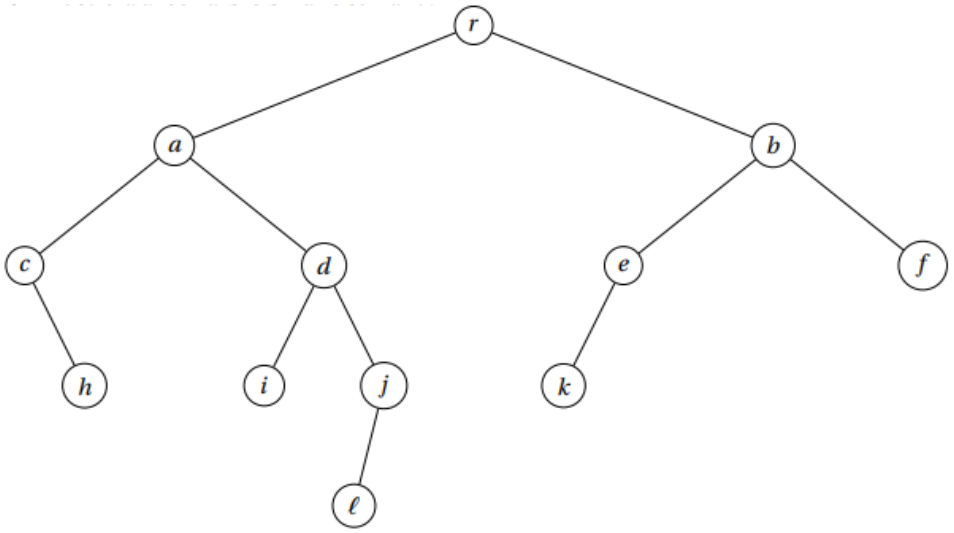
Le ***parcours en largeur*** consiste à parcourir l'arbre niveau par niveau. Les nœuds de niveau 0 sont d'abord parcourus puis les nœuds de niveau 1 et ainsi de suite. Dans chaque niveau, les nœuds sont parcourus de la gauche vers la droite. Le parcours en largeur de l'arbre ci-dessous parcourt les nœuds dans l'ordre [15,6,18,3,7,17,20,2,4,13,9]. Le parcours d'un arbre en largeur s'effectue à l'aide d'une file. En partant de la racine de l'arbre, on insère tous les descendants du sommet actuel dans la file, et on répète l'opération sur chaque sommet que l'on extrait de la file jusqu'à ce qu'elle soit vide. Le parcours en largeur se programme à l'aide d'une file (Fifo) de la manière suivante :




```
Parcour_largeur(Tree t) {
   Fifo fifo = new Fifo()      // Création d'une file
   fifo.put(t.root)           // Mise de la racine dans la file
   while(!fifo.empty()) {
  Node n = fifo.get();         // Nouveau nœud à traiter en tête de file
  traitement(n);               // Traitement du nœud courant
  if (n.fg != nil) fifo.put(n.fg);  // Ajout du fils gauche s'il existe
  if (n.fd != nil) fifo.put(n.fd);  // Ajout du fils droit s'il existe
   }
}
```

b- Parcours en Profondeur

Les parcours en profondeur se définissent de manière récursive sur les arbres. Le parcours d'un arbre consiste à traiter la racine de l'arbre et à parcourir récursivement les sous-arbres gauche et droit de la racine. Les parcours préfixe, infixe et suffixe se distinguent par l'ordre dans lequel sont faits ces traitements. On s'intéressera a l'arbre ci-dessous pour explique les diffèrent parcours en largeur.



i) Parcours Préfixé

Dans le ***parcours préfixé***, la racine est traitée avant les appels récursifs sur les sous-arbres gauche et droit (faits dans cet ordre). Le parcours préfixe de l'arbre ci-dessus parcourt les nœuds dans l'ordre [*r, a, c, h, d, i, j, l, b, e, k, f*]. Le pseudo code pour le parcours préfixe :

PARCOURS-PREFIXE(*x*)
1 si *x* ≠ NIL
2 afficher *clé*[*x*]
3 PARCOURS-PREFIXE (*gauche*[*x*])
4 PARCOURS-PREFIXE (*droite*[*x*])

ii) Parcours Infixé

Dans le ***parcours infixé***, le traitement de la racine est fait entre les appels sur les sous-arbres gauche et droit. Le parcours infixe de l'arbre ci-dessus parcourt les nœuds dans l'ordre. [h, *c, i, l, j, d, a, k, e, f, b, r*].Le pseudo code pour le parcours infixe :

PARCOURS-INFIXE(*x*)
1 si *x* ≠ NIL
2 alors PARCOURS-INFIXE (*gauche*[*x*])
3 afficher *clé*[*x*]
4 PARCOURS-INFIXE (*droite*[*x*])

iii) Parcours Postfixé

Dans le ***parcours postfixé***, la racine est traitée après les appels récursifs sur les sous-arbres gauche et droit (faits dans cet ordre). Le parcours postfixe de l'arbre ci-dessus parcourt les nœuds dans l'ordre [*c, h, a, i, d, l, j, r, k, e, b, f*]. Le pseudo code pour le parcours postfixe :

PARCOURS-POSFIXE(*x*)
1 si *x* ≠ NIL
2 alors PARCOURS-POSTFIXE (*gauche*[*x*])
3 PARCOURS-POSTFIXE (*droite*[*x*])
4 afficher *clé*[*x*]

D-Opérations élémentaires

a- Recherche d'un élément

On utilise la procédure suivante pour rechercher un nœud ayant une clé donnée dans un arbre binaire de recherche. Étant donné un pointeur sur la racine de l'arbre et une clé k, ARBRERECHERCHER retourne un pointeur sur un nœud de clé k s'il en existe un ; sinon, elle retourne NIL.

```
ARBRE-RECHERCHER(x, k)
1 si x = NIL ou k = clé[x]
2        alors retourner x
3 si k < clé[x]
4        alors retourner ARBRE-RECHERCHER (gauche[x], k)
5        sinon retourner ARBRE-RECHERCHER (droite[x], k)
```

Le temps d'exécution de ARBRE-RECHERCHER est O(h) si h est la hauteur de l'arbre.

b- Minimum et Maximum

On peut toujours trouver un élément d'un arbre binaire de recherche dont la clé est un minimum en suivant les pointeurs gauches à partir de la racine jusqu'à ce qu'on rencontre NIL. La procédure suivante retourne un pointeur sur l'élément minimal du sous arbre enraciné au nœud x.

```
ARBRE-MINIMUM(x)
1 tant que gauche[x] ≠ NIL
2        faire x ← gauche[x]
3 retourner x
```

Le pseudo code de ARBRE-MAXIMUM est symétrique.

```
ARBRE-MAXIMUM(x)
1 tant que droite[x] ≠ NIL
2        faire x ← droite[x]
3 retourner x
```

Ces deux procédures s'exécutent en O(h) pour un arbre de hauteur h puisque, comme dans ARBRE-RECHERCHER, elles suivent des chemins descendants qui partent de la racine.

c- Successeur et Prédécesseur

Étant donné un nœud d'un arbre binaire de recherche, il est parfois utile de pouvoir trouver son successeur dans l'ordre déterminé par un parcours infixe de l'arbre. Si toutes les clés sont distinctes, le successeur d'un nœud x est le nœud possédant la plus petite clé supérieure à clé[x]. La structure d'un arbre binaire de recherche permet de déterminer le successeur d'un nœud sans

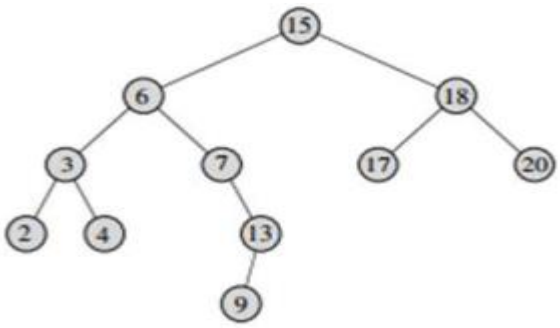
même effectuer de comparaison entre les clés. Le code de ARBRE-SUCCESSEUR est séparé en deux cas : Si le sous arbre de droite du nœud x n'est pas vide, alors le successeur de x est tout simplement le nœud le plus à gauche dans le sous arbre de droite. Par exemple, le successeur du nœud de clé 15 dans la figure ci-dessous est le nœud de clé 17. En revanche, si le sous arbre de droite du nœud x est vide et que x a un successeur y, alors y est le premier ancêtre de x dont l'enfant de gauche est aussi un ancêtre de x. Sur la figure ci-dessous, le successeur du nœud de clé 13 est le nœud de clé 15. Pour trouver y, on remonte tout simplement l'arbre à partir de x jusqu'à trouver un nœud qui soit l'enfant de gauche de son parent ; et ainsi considérer son parent comme le successeur de x.

La procédure suivante retourne le successeur d'un nœud x dans un arbre binaire de recherche, s'il existe et NIL si X possède la plus grande clé de l'arbre.

```
        ARBRE-SUCCESSEUR(x)
1 si droite[x] ≠ NIL
2       alors retourner ARBRE-MINIMUM (droite[x])
3 y ← p[x]
4 tant que y ≠ NIL et x = droite[y]
5       faire x ← y
6             y ← p[y]
7 retourner y
```

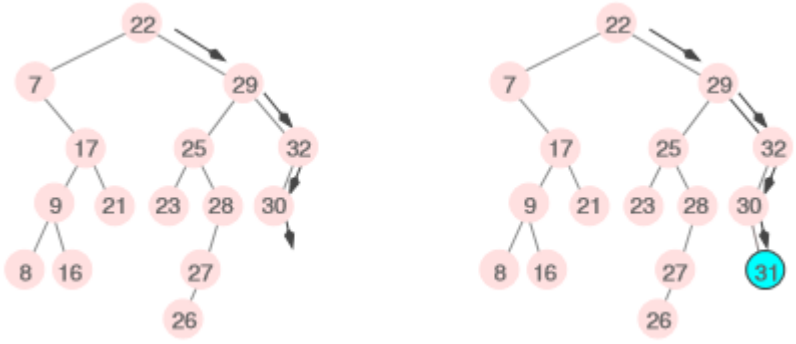
Le temps d'exécution de ARBRE-SUCCESSEUR sur un arbre de hauteur h est O (h).
En effet, on suit soit un chemin vers le haut de l'arbre, soit un chemin vers le bas. La procédure ARBRE-PRÉDÉCESSEUR, qui est symétrique d'ARBRE-SUCCESSEUR, s'exécute également dans un temps O(h).

```
        ARBRE-PREDECESSEUR (x)
1 si gauche[x] ≠ NIL
2       alors retourner ARBRE-MAXIMUM (gauche[x])
3 y ← p[x]
4 tant que y ≠ NIL et x = gauche[y]
5       faire x ← y
6             y ← p[y]
7 retourner y
```



d- Insertion dans un arbre binaire

La construction d'un Arbre Binaire de Recherche à partir d'une liste est le résultat de l'application de l'algorithme d'insertion d'une valeur dans un Arbre Binaire de Recherche à chaque élément de la liste. Le principe est le même que pour la recherche. Un nouveau nœud est créé avec la nouvelle valeur et inséré à l'endroit où la recherche s'est arrêtée.



L'algorithme Insertion peut s'écrire de la façon suivante :

```
ARBRE-INSÉRER(T, z)
 1  y ← NIL
 2  x ← racine[T]
 3  tant que x ≠ NIL
 4       faire y ← x
 5            si clé[z] < clé[x]
 6               alors x ← gauche[x]
 7               sinon x ← droite[x]
 8  p[z] ← y
 9  si y = NIL
10       alors racine[T] ← z                ▷ arbre T était vide
11       sinon si clé[z] < clé[y]
12               alors gauche[y] ← z
13               sinon droite[y] ← z
```

Complexité :

On suppose que dans le meilleur des cas et en moyenne, on élimine une moitié d'arbre à chaque comparaison jusqu'à tomber sur un emplacement libre. Pour simplifier, on va dire que l'arbre comporte autant de nœuds que d'élément à insérer. Si on élimine à chaque itération la moitié des valeurs de l'arbre, on doit se poser la question : combien de fois doit-on diviser n par deux jusqu'à tomber sur 1 ? il s'agit en fait d'une fonction logarithme (en base 2). L'algorithme d'insertion, Comme les autres opérations primitives d'arbre binaire de recherche, s'exécute donc en O(h) sur arbre de hauteur h ou h=log(n).

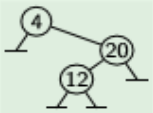
Exemple :

Insertions successives de : **4, 20, 12, 2, 7, 3, 6, 0, 15, 1, 13, 14** dans un Arbre Binaire de Recherche vide :

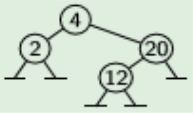
Insertions successives de 4, 20, 12, 2, 7, 3, 6, 0, 15, 1, 13, 14 dans l'ABR vide :

Insertions successives de 4, 20, 12, 2, 7, 3, 6, 0, 15, 1, 13, 14 dans l'ABR vide :

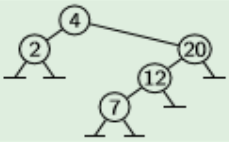
Insertions successives de 4, 20, 12, 2, 7, 3, 6, 0, 15, 1, 13, 14 dans l'ABR vide :



Insertions successives de 4, 20, 12, 2, 7, 3, 6, 0, 15, 1, 13, 14 dans l'ABR vide :

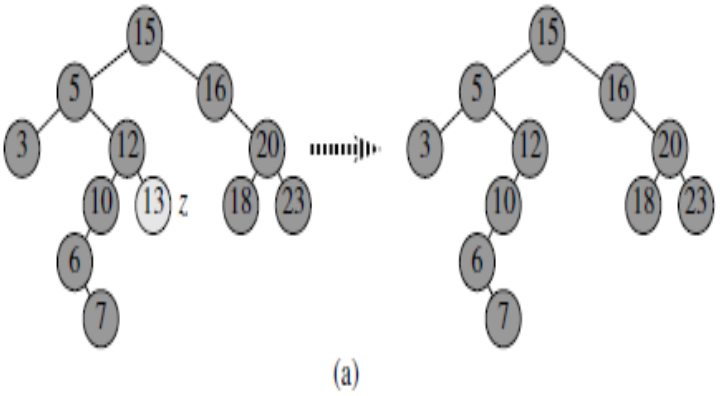


Insertions successives de 4, 20, 12, 2, 7, 3, 6, 0, 15, 1, 13, 14 dans l'ABR vide :

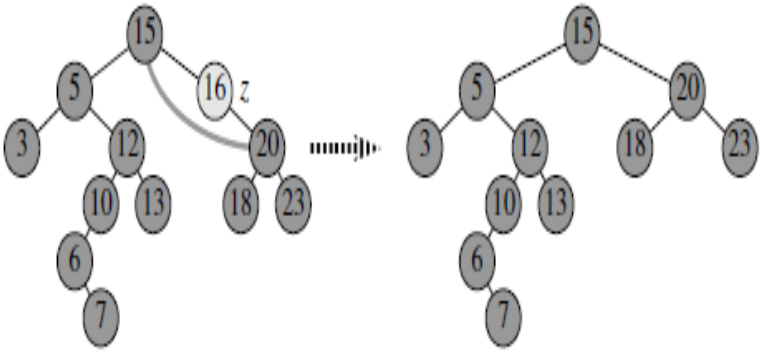


e- Suppression dans un arbre binaire

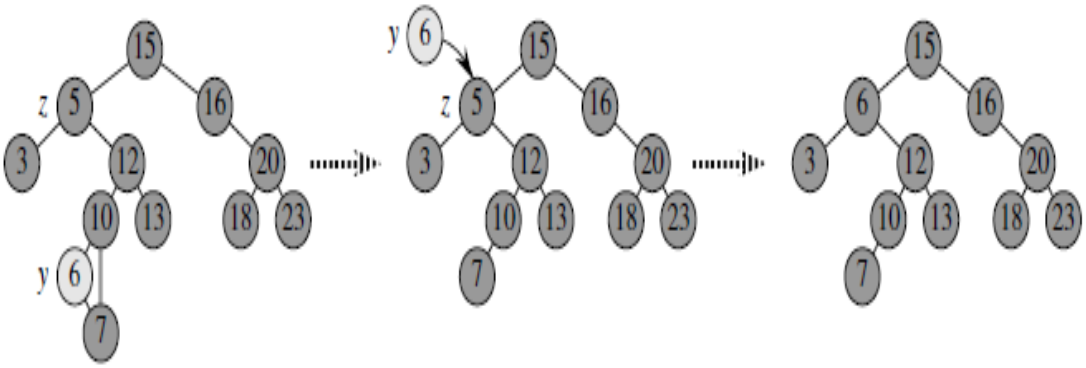
La procédure permettant de supprimer un nœud donné z d'un arbre binaire de recherche prend comme argument un pointeur sur z. La procédure considère les trois cas montrés suivant. Si z n'a pas d'enfant, on modifie son parent p[z] pour remplacer z par NIL dans le champ enfant. Si le nœud n'a qu'un seul enfant, on « détache » z en créant un nouveau lien entre son enfant et son parent. Enfin, si le nœud a deux enfants, on détache le successeur de z, y, qui n'a pas d'enfant gauche et on remplace la clé et les données satellites de z par la clé et les données satellite de y. Le code d'ARBRE-SUPPRIMER gère ces trois cas un peu différemment.



(a)



(b)



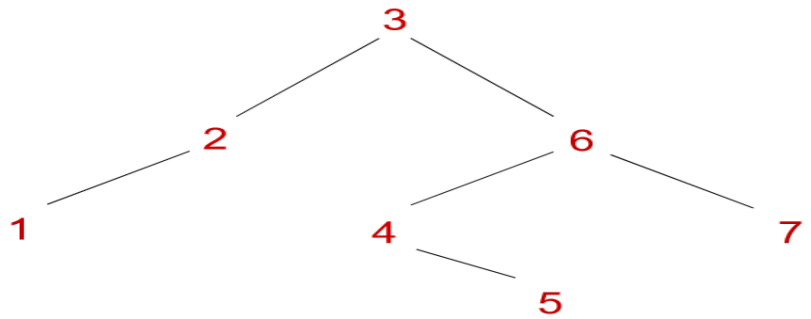
(c)

La procédure est la suivante :

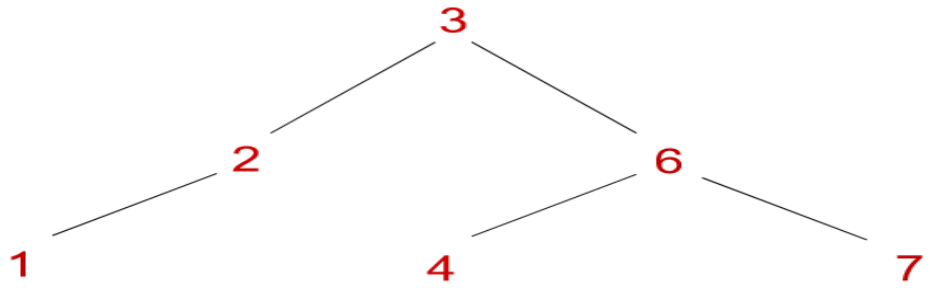
```
ARBRE-SUPPRIMER(T, z)
 1  si gauche[z] = NIL ou droite[z] = NIL
 2     alors y ← z
 3     sinon y ← ARBRE-SUCCESSEUR(z)
 4  si gauche[y] ≠ NIL
 5     alors x ← gauche[y]
 6     sinon x ← droite[y]
 7  si x ≠ NIL
 8     alors p[x] ← p[y]
 9  si p[y] = NIL
10     alors racine[T] ← x
11     sinon si y = gauche[p[y]]
12              alors gauche[p[y]] ← x
13              sinon droite[p[y]] ← x
14  si y ≠ z
15     alors clé[z] ← clé[y]
16           copier données satellites de y dans z
17  retourner y
```

Complexité :
La procédure s'exécute en temps O(h) sur un arbre de hauteur h.

Exemple D'application :



On supprime 5



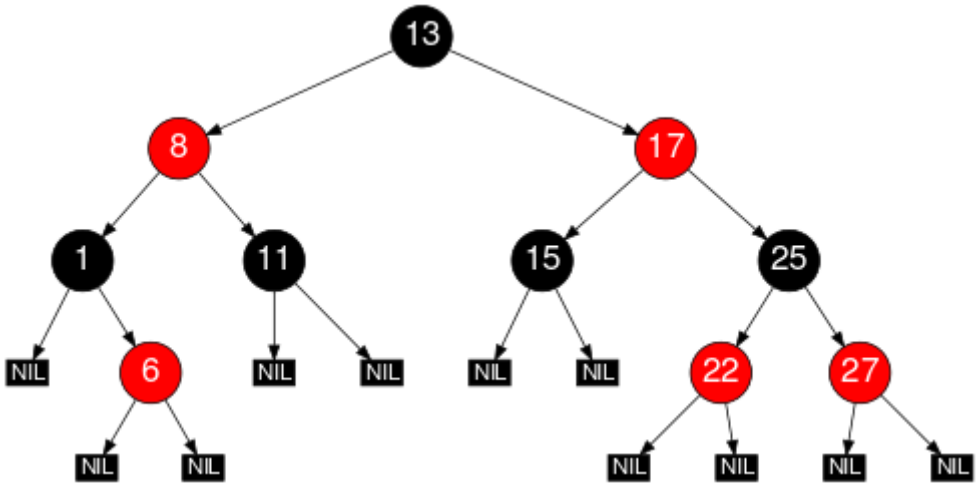
III- Arbre Rouge et Noir

A-Définition :

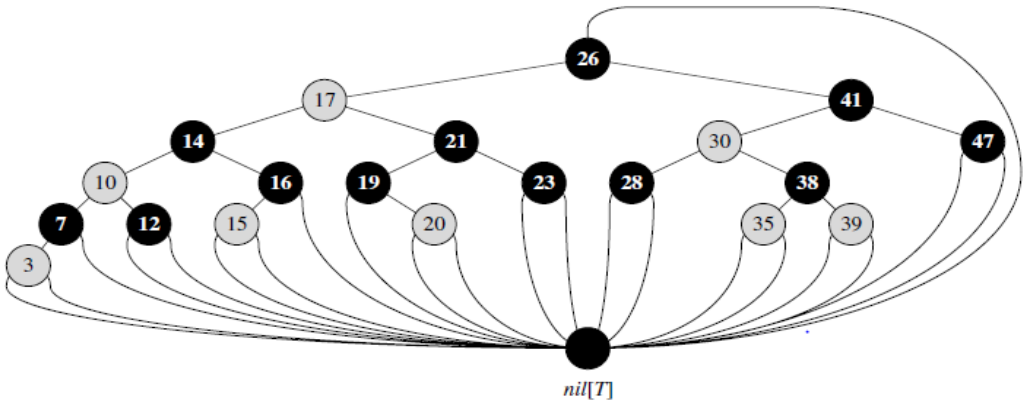
Un ***arbre rouge-noir*** est un arbre binaire de recherche comportant un bit de stockage supplémentaire par nœud : sa ***couleur***, qui peut être ROUGE ou NOIR. En contrôlant la manière dont les nœuds sont coloriés sur n'importe quel chemin allant de la racine à une feuille, les arbres rouge-noir garantissent qu'aucun de ces chemins n'est plus de deux fois plus long que n'importe quel autre, ce qui rend l'arbre approximativement ***équilibré***. Un arbre binaire de recherche est un arbre rouge-noir s'il satisfait aux propriétés suivantes :

1) Chaque nœud est soit rouge, soit noir.
2) La racine est noire.
3) Chaque feuille (NIL) est noire.
4) Si un nœud est rouge, alors ses deux enfants sont noirs.
5) Pour chaque nœud, tous les chemins reliant le nœud à des feuilles contiennent le même nombre de nœuds noirs.

Une représentation d'un arbre rouge noir :

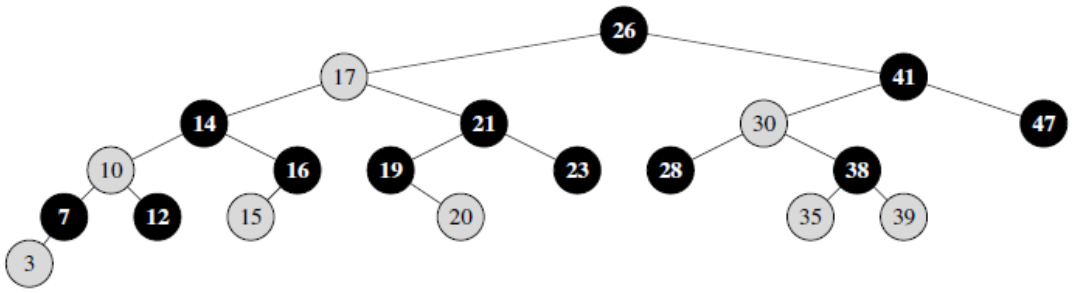


Pour simplifier le traitement des conditions aux limites dans la programmation des arbres rouge-noir, on utilise une même sentinelle pour représenter NIL. Pour un arbre rouge noir *T*, la sentinelle *Nil*[*T*] est un objet ayant les mêmes champs qu'un nœud ordinaire. Son champ *couleur* vaut NOIR, et ses autres champs (*p*, *gauche*, *droite* et *clé*) peuvent prendre des valeurs quelconques. Comme le montre la figure suivante, tous les pointeurs vers NIL sont remplacés par des pointeurs vers la sentinelle *Nil*[*T*].



Représentation d'un arbre rouge et noir avec une sentinelle.

On utilise la sentinelle pour pouvoir traiter un enfant NIL d'un nœud x comme un nœud ordinaire dont le parent est x. On pourrait ajouter un nœud sentinelle distinct pour chaque NIL de l'arbre, pour que le parent de chaque NIL soit bien défini, mais cette méthode gaspillerait de l'espace. À la place, on emploie une seule sentinelle $Nil[T]$ pour représenter tous les NIL (à savoir les feuilles et le parent de la racine). Les valeurs des champs *p*, *gauche*, *droite* et *clé* de la sentinelle sont immatérielles, bien que nous puissions les configurer à notre convenance dans le cours d'une procédure. On ne s'intéresse généralement qu'aux nœuds internes d'un arbre rouge-noir, vu que ce sont eux qui contiennent les valeurs de clé. Dans la suite, nous omettrons les feuilles quand nous dessinerons des arbres rouge-noir, comme c'est le cas sur la figure ci-dessous :



B- Hauteur d'un arbre binaire rouge-noir

On appelle ***hauteur noire*** le nombre de nœuds noirs d'un chemin partant d'un nœud x (non compris ce nœud) vers une feuille, et on utilise la notation **bh(*x*).** D'après la propriété 5, la notion de hauteur noire est bien définie, puisque tous les chemins descendant d'un nœud contiennent le même nombre de nœuds noirs. On définit la hauteur noire d'un arbre rouge-noir comme étant la hauteur noire de sa racine. Ceci montre pourquoi les arbres rouge-noir sont de bons arbres de recherche.

Lemme : *Un arbre rouge-noir ayant n nœuds internes à une h auteur au plus égale à* **2 lg (*n* + 1).**

Démonstration :

Commençons par montrer que le sous arbre enraciné en un nœud x quelconque contient au moins $2^{bh(x)} - 1$ nœud interne. Cette affirmation peut se démontrer par récurrence sur la hauteur

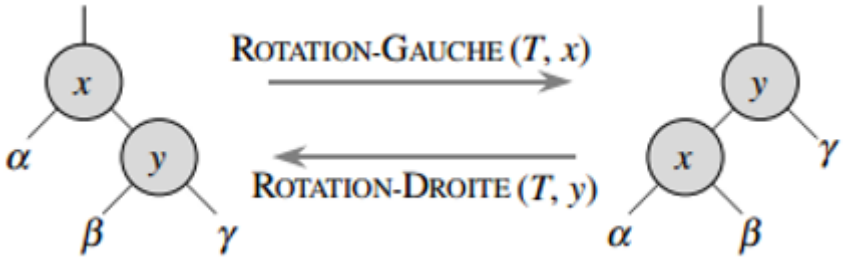
de x. Si la hauteur de x est 0, alors x est obligatoirement une feuille ($Nil[T]$) et le sous arbre enraciné en x contient effectivement au moins $2^{bh(x)} - 1 = 2^0 - 1 = 0$ nœuds internes.
Pour l'étape inductive, soit un nœud interne x de hauteur positive ayant deux enfants. Chaque enfant a une hauteur noire bh(x) ou bh(x) − 1, selon que sa couleur est rouge ou noire.
Comme la hauteur d'un enfant de x est inférieure à celle de x lui-même, on peut appliquer l'hypothèse de récurrence pour conclure que chaque enfant a au moins $2^{bh(x)-1} - 1$ nœuds internes. Le sous arbre de racine x contient donc au moins $(2^{bh(x)-1} - 1) + (2^{bh(x)-1} - 1) + 1 = 2^{bh(x)} - 1$ nœuds internes, ce qui démontre l'assertion.
Pour compléter notre preuve, appelons h la hauteur de l'arbre. D'après la propriété 4, au moins la moitié des nœuds d'un chemin simple reliant la racine à une feuille (racine non comprise) doivent être noirs.
En conséquence, la hauteur noire de la racine doit valoir au moins $h/2$; donc $n \geq 2^{h/2} - 1$.
En faisant passer le 1 dans le membre gauche et en prenant le logarithme des deux membres, on obtient $\lg(n + 1) \geq h/2$, soit $h \leq 2 \lg(n + 1)$.

Ce résultat nous sera très utile pour déterminer la complexité en temps sur les opérations d'ensemble dynamique Minimum, Maximum et Successeur qui peuvent être implémenter en temps O(h) sur les arbres rouge-noir, puisqu'elles peuvent s'exécuter en temps $O(h)$ sur un arbre de recherche de hauteur h et qu'un arbre rouge-noir à n nœuds est un arbre de recherche de hauteur $O(\lg n)$.

C- Opérations élémentaires

a- Rotation :

Les opérations d'arbre de recherche **ARBRE-INSÉRER** et **ARBRE-SUPPRIMER**, quand on les exécute sur un arbre rouge-noir à n clés, prennent un temps O (lg n). Comme elles modifient l'arbre, le résultat pourrait violer les propriétés d'arbre rouge-noir énumérées plus haut. Pour restaurer ces propriétés, il faut changer les couleurs de certains nœuds de l'arbre et également modifier la chaîne des pointeurs.



L'opération ROTATIONGAUCHE (T, x) transforme la configuration des deux nœuds de gauche pour aboutir à celle de droite, en modifiant un nombre constant de pointeurs. La configuration de droite peut être transformée en celle de gauche par l'opération inverse ROTATION-DROITE (T, y). Les deux nœuds peuvent se trouver n'importe où dans l'arbre binaire de recherche. Les lettres α, β et γ représentent des sous arbres arbitraires. Une opération de rotation préserve la propriété d'arbre binaire de recherche : les clés d'a précèdent *clé*[x], qui précède les clés de β, qui précède *clé*[y], qui précède les clés de γ

On modifie la chaîne des pointeurs *via* une ***rotation***, qui est une opération locale d'arbre de recherche qui préserve la propriété d'arbre binaire de recherche. La figure ci-dessus montre les deux sortes de rotation : les rotations gauches et les rotations droites. Lorsqu'on

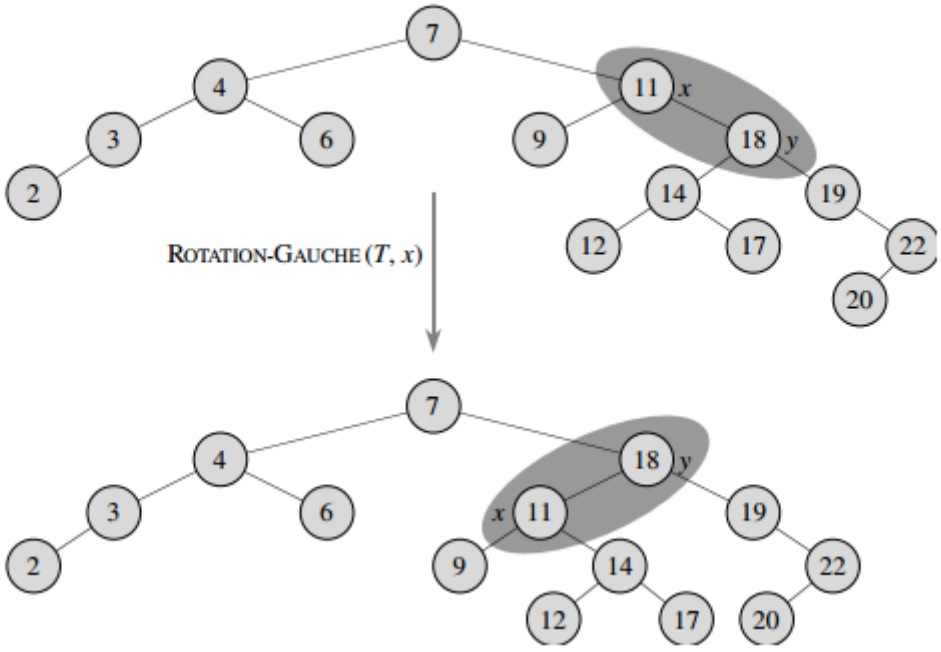
effectue une rotation gauche sur un nœud *x*, on suppose que son enfant de droite *y* n'est pas *Nil*[*T*] ; *x* peut être un nœud quelconque de l'arbre dont l'enfant de droite n'est pas *Nil*[*T*]. La rotation gauche fait « pivoter » autour du lien qui relie *x* et *y*. Elle fait ensuite de *y* la nouvelle racine du sous arbre, avec *x* qui devient l'enfant de gauche de *y* et l'enfant de gauche de *y* qui devient l'enfant de droite de *x*.

L'algorithme se trouve ci-dessous

```
ROTATION-GAUCHE(T, x)
 1  y ← droite[x]                    ▷ initialise y.
 2  droite[x] ← gauche[y]            ▷ sous-arbre gauche de y devient
                                       sous-arbre droit de x.
 3  si gauche[y] ≠ nil[T]
 4     alors p[gauche[y]] ← x
 5           p[y] ← p[x]              ▷ relie parent de x à y.
 6           si p[x] = nil[T]
 7              alors racine[T] ← y
 8              sinon si x = gauche[p[x]]
 9                       alors gauche[p[x]] ← y
10                       sinon droite[p[x]] ← y
11           gauche[y] ← x            ▷ place x à gauche de y.
12           p[x] ← y
```

Complexité : la complexité en temps de cet algorithme est ***O*(1).**

Un exemple d'une rotation gauche est la figure :



Dans la suite on va s'intéresse à un certains nombres d'opérations sur les arbres rouge noir tel que **INSERTION, SUPPRESSION, MINIMUM, MAXIMUM et PREDESSEUR.** On

mettra un accent particulier sur les opérations insertion, suppression car celles si modifie labre rouge noir et on devra en conséquences s'assurer d'avoir un arbre rouge noire après modification (i.e. respecte les 5 propriétés des arbres rouge noir)

b- Insertion dans un arbre Rouge et Noir

Considérons l'insertion d'un nœud z dans un arbre rouge-noir.

Etape 1 : On utilise la procédure ARBRE-INSERER pour insérer le nœud z dans l'arbre T comme si c'était un arbre binaire de recherche ordinaire.

Etape 2 : On colorie le nœud z en rouge. [Parce que cette coloration préserve la propriété 5]

Etape 3 : Malheureusement le nouveau nœud rouge a peut- être un père rouge, c'est- à- dire la propriété 4 est violée. On corrige ensuite l'arbre en recolorant les nœuds et en effectuant des rotations.

Qu'est-ce qu'il faut faire à l'étape 3 ?

On peut constater que les deux propriétés, qui peuvent être violées après la coloration du nœud z, sont les propriétés 2 et 4.

Cas 1 : L'oncle y de z, c'est- à- dire y = droite [p[p[z]]], est rouge. Alors p[z] et y sont tous les deux rouges. On colorie p[z] et y en noir. Pour conserver la propriété 5 il faut qu'on colorie p[p[z]] qui était noir en rouge. Peut- être propriété 4 est maintenant violée pour p[p[z]] et son père, si le père est rouge. Alors on appelle z le nœud p[p[z]] et on répète. Constatons qu'on a progressé du nœud actuel à son grand-père.

Cas 2 et 3 : L'oncle y = droite [p[p[z]]] de z est noir. Dans ces cas on utilise des rotations et recoloration. Les deux cas se distinguent selon que z est un fils droit ou gauche de p[z]. On transforme le cas 2 en cas 3 par une seule ROTATION-GAUCHE (T ; z). Dans cas 3, on colorie p[z] noir et p[p[z]] rouge. Après on utilise une seule ROTATION-DROITE (T ; p[p[z]]). A cause de cette rotation droite p[p[z]] ne reste pas le fils de son père p[p[p[z]]]. Le nouveau fils de p[p[p[z]]], qui est peut- être rouge, devient p[z] qui est certainement noir.

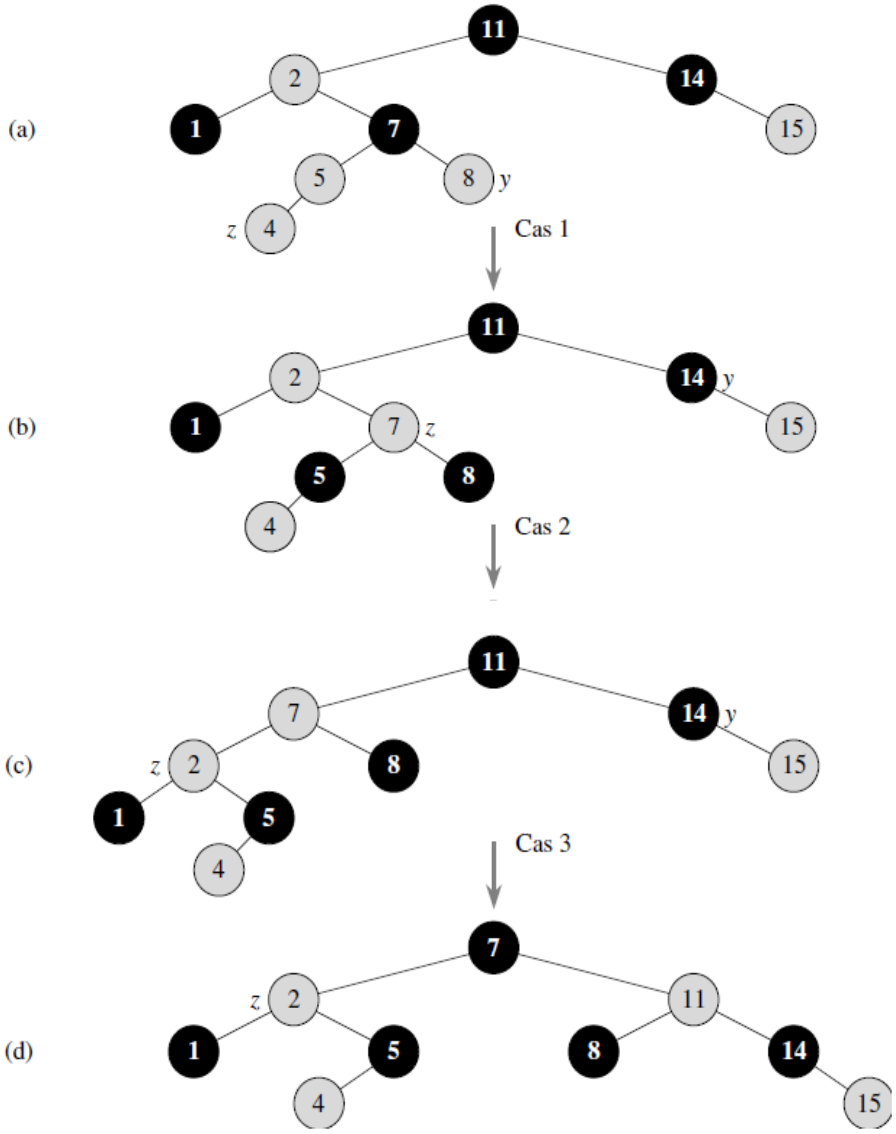
Soit RN-INSERER l'algorithme qui insert un nœud et RN-INSERER-CORRECTION celui qui restaure les propriétés on a :

```
RN-INSÉRER(T, z)
1  y ← nil[T]
2  x ← racine[T]
3  tant que x ≠ nil[T]
4       faire y ← x
5            si clé[z] < clé[x]
6               alors x ← gauche[x]
7               sinon x ← droite[x]
```

```
 8   p[z] ← y
 9   si y = nil[T]
10       alors racine[T] ← z
11       sinon si clé[z] < clé[y]
12                 alors gauche[y] ← z
13                 sinon droite[y] ← z
14   gauche[z] ← nil[T]
15   droite[z] ← nil[T]
16   couleur[z] ← ROUGE
17   RN-INSÉRER-CORRECTION(T, z)
```

```
RN-INSÉRER-CORRECTION(T, z)
 1  tant que couleur[p[z]] = ROUGE
 2      faire si p[z] = gauche[p[p[z]]]
 3           alors y ← droite[p[p[z]]]
 4                si couleur[y] = ROUGE
 5                  alors couleur[p[z]] ← NOIR                    ▷ Cas 1
 6                        couleur[y] ← NOIR                       ▷ Cas 1
 7                        couleur[p[p[z]]] ← ROUGE                ▷ Cas 1
 8                        z ← p[p[z]]                             ▷ Cas 1
 9                  sinon si z = droite[p[z]]
10                          alors z ← p[z]                        ▷ Cas 2
11                                ROTATION-GAUCHE(T, z)           ▷ Cas 2
12                        couleur[p[z]] ← NOIR                    ▷ Cas 3
13                        couleur[p[p[z]]] ← ROUGE                ▷ Cas 3
14                        ROTATION-DROITE(T, p[p[z]])             ▷ Cas 3
15           sinon (idem clause alors avec
                        permutation de « droite » et « gauche »)
16  couleur[racine[T]] ← NOIR
```

La figure suivante montre le fonctionnement de **RN-INSÉRER-CORRECTION** sur un exemple d'arbre rouge-noir.



Fonctionnement de RN-INSÉRER-CORRECTION

(a) Un nœud z après insertion. Comme z et son parent p[z] sont tous les deux rouges, il y a violation de la propriété 4. Comme l'oncle de z, y, est rouge, on est dans le cas 1 du code. Les nœuds sont recoloriés et le pointeur z remonte dans l'arbre, ce qui donne l'arbre montré en (b). Ici aussi, z et son parent sont tous les deux rouges, mais l'oncle de z, y, est noir. Comme z est l'enfant de droite de p[z], on est dans le cas 2. On effectue une rotation gauche, et l'arbre résultant est montré en (c). Maintenant z est l'enfant de gauche de son parent, et l'on est dans le cas 3. Une rotation droite produit l'arbre de (d), qui est un arbre rouge-noir correct.

c- Suppression dans un arbre Rouge et Noir

Comme les autres opérations primitives sur un arbre rouge-noir à n nœuds, la suppression d'un nœud requiert un temps O(lgn). Supprimer un nœud d'un arbre rouge-noir est cependant légèrement plus compliqué que d'en insérer un. La procédure RN-SUPPRIMER est une version légèrement modifie de la procédure ARBRE-SUPPRIMER. Après suppression d'un nœud, elle appelle une procédure auxiliaire RN-SUPPRIMER-CORRECTION qui modifie des couleurs et fait des rotations pour restaurer les propriétés rouge-noir.

```
RN-SUPPRIMER(T, z)
 1  si gauche[z] = nil[T] ou droite[z] = nil[T]
 2     alors y ← z
 3     sinon y ← ARBRE-SUCCESSEUR(z)
 4  si gauche[y] ≠ nil[T]
 5     alors x ← gauche[y]
 6     sinon x ← droite[y]
 7  p[x] ← p[y]
 8  si p[y] = nil[T]
 9     alors racine[T] ← x
10     sinon si y = gauche[p[y]]
11              alors gauche[p[y]] ← x
12              sinon droite[p[y]] ← x
13  si y ≠ z
14     alors clé[z] ← clé[y]
15           copier données satellite de y dans z
16  si couleur[y] = NOIR
17     alors RN-SUPPRIMER-CORRECTION(T, x)
18  retourner y
```

Le nœud x passé à RN-SUPPRIMER-CORRECTION est l'un ou l'autre des deux nœuds suivants : le nœud qui était l'unique enfant de y avant suppression de y si y avait un enfant qui n'était pas la sentinelle nil[T], ou, si y n'avait pas d'enfants, x est la sentinelle nil[T]. Dans ce dernier cas, l'assignation inconditionnelle de la ligne 7 garantit que le parent de x est maintenant le nœud qui était précédemment le parent de y, que x soit un nœud interne porteur de clé ou qu'il soit la sentinelle nil[T]. On peut maintenant regarder comment la procédure RN-SUPPRIMER-CORRECTION restaure les propriétés rouge-noir de l'arbre.

```
RN-SUPPRIMER-CORRECTION(T, x)
 1  tant que x ≠ racine[T] et couleur[x] = NOIR
 2      faire si x = gauche[p[x]]
 3            alors w ← droite[p[x]]
 4                 si couleur[w] = ROUGE
 5                   alors couleur[w] ← NOIR                    ▷ Cas 1
 6                        couleur[p[x]] ← ROUGE                 ▷ Cas 1
 7                        ROTATION-GAUCHE(T, p[x])              ▷ Cas 1
 8                        w ← droite[p[x]]                      ▷ Cas 1
 9                 si couleur[gauche[w]] = NOIR et couleur[droite[w]] = NOIR
10                   alors couleur[w] ← ROUGE                   ▷ Cas 2
11                        x ← p[x]                              ▷ Cas 2
12                   sinon si couleur[droite[w]] = NOIR
13                             alors couleur[gauche[w]] ← NOIR  ▷ Cas 3
14                                  couleur[w] ← ROUGE          ▷ Cas 3
15                                  ROTATION-DROITE(T, w)       ▷ Cas 3
16                                  w ← droite[p[x]]            ▷ Cas 3
17                        couleur[w] ← couleur[p[x]]            ▷ Cas 4
18                        couleur[p[x]] ← NOIR                  ▷ Cas 4
19                        couleur[droite[w]] ← NOIR             ▷ Cas 4
20                        ROTATION-GAUCHE(T, p[x])              ▷ Cas 4
21                        x ← racine[T]                         ▷ Cas 4
22            sinon (idem clause alors avec « droite » et « gauche » échangés)
23  couleur[x] ← NOIR
```

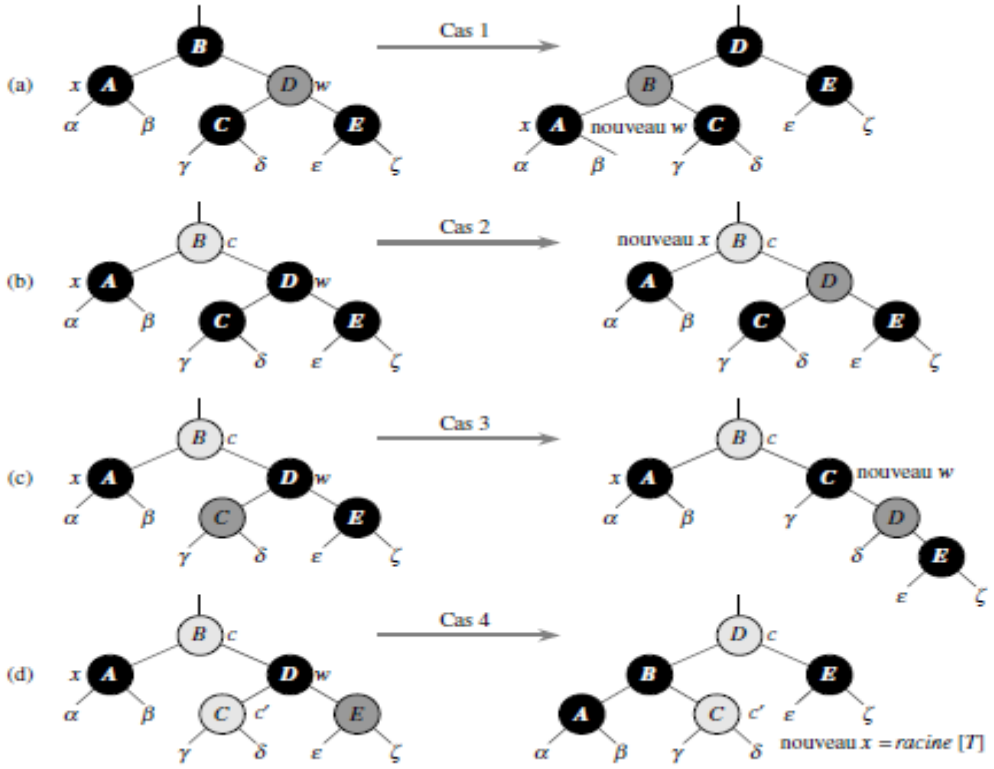
On assiste à 4 cas de figures :

Cas 1 : le frère de x, w, est rouge Le cas 1 lignes 5–8 de RN-SUPPRIMER-CORRECTION se produit quand le nœud w, le frère de x, est rouge. Comme w doit avoir des enfants noirs, on peut permuter les couleurs de w et p[x] puis effectuer une rotation gauche sur p[x] sans enfreindre l'une des propriétés rouge-noir. Le nouveau frère de x, qui est l'un des enfants de w avant la rotation, est désormais noir, et donc le cas 1 est ramené au cas 2, 3 ou 4. Les cas 2, 3 et 4 se produisent quand le nœud w est noir ; ces cas se distinguent par les couleurs des enfants de w.

Cas 2 : le frère de x, w, est noir, et les deux enfants de w sont noirs. Dans le cas 2 lignes 10–11 de RN-SUPPRIMER-CORRECTION, les deux enfants de w sont noirs. Comme w est noir lui aussi, on enlève un noir à x et à w, ce qui laisse à x un seul noir et qui laisse w rouge. Pour compenser la suppression d'un noir dans x et w, on voudrait ajouter un noir supplémentaire à p[x], qui était au départ rouge ou noir. Pour ce faire, on répète la boucle tant que en faisant de p[x] le nouveau nœud x. Observez que, si l'on entre dans le cas 2 via le cas 1, le nouveau nœud x est rouge-et-noir, vu que le p[x] originel était rouge. Donc, la valeur c de l'attribut couleur du nouveau nœud x est ROUGE, et la boucle se termine quand elle teste la condition de boucle. Le nouveau nœud x est alors colorié en noir (simple), en ligne 2 3.

Cas 3 : le frère de x, w, est noir, l'enfant gauche de w est rouge et l'enfant droite de w est noir Le cas 3 (lignes 13–16 se produit quand w est noir, son enfant gauche est rouge et son enfant droite est noir. On peut permuter les couleurs de w et de son enfant gauche gauche[w] puis faire une rotation droite sur w sans enfreindre les propriétés rouge-noir. Le nouveau frère w de x est maintenant un nœud noir avec un enfant droite rouge, et le cas 3 est donc ramené au cas 4.

Cas 4 : le frère de x, w, est noir et l'enfant droite de w est rouge Le cas 4 (lignes 17–21) se produit quand le frère du nœud x est noir et que l'enfant droite de w est rouge. En faisant des modifications de couleur et en effectuant une rotation gauche sur p[x], on peut supprimer le noir en trop de x, rendant ce dernier noir simple, sans violer de propriété rouge noir. Faire de x la racine forcera la boucle tant que à se terminer quand elle testera la condition de bouclage. Regardons l'exemple suivant :



Conclusion

Parvenus au terme de notre cours dans lequel il était question pour nous d'étudier trois types d'arbres à savoir : les arbres binaires, les arbres binaires de recherche, les arbres rouges-noirs, nous constatons que les opérations dynamiques sont basées sur la hauteur de l'arbre. En ce qui concerne les arbres binaires de recherche, ils apportent un complément aux arbres binaires simples et permettent donc de minimiser le temps de recherche d'un élément qui est au moins de log2(n). Quant aux arbres rouges-noirs, ils viennent avec de nouvelles propriétés restrictives sur les arbres de recherche afin d'optimiser la recherche et surtout d'équilibrer l'arbre, réduisant ainsi la hauteur de l'arbre qui est d'au plus log2(n).

BIBLIOGRAPHIE

- Introduction-à-l'algorithmique-Cours-et-exercices-corrigés Dunod, Paris, 2004, pour la présente édition ISBN 2 10 003922 9

WEBOGRAPHIE

- http://www.irif.fr/carton/Enseignement/Algorithme/LicenceMathInfo/Programmation/Tree/parcours.thml visité le 13-11-2018 à 13h
- http://www.iro.umontreal.ca/hamelsyl/ArbreRN-09-4.pdf visité le 15-11-2018 à 8h
- http://www.developpez.net visité le 10-11-2018 à 14h