Arbres de recherche équilibrés

1. Arbres binaires de recherche équilibrés

Définition: Un arbre binaire de recherche est dit équilibré (ou bien balancé, en anglais *balanced tree*, b-tree) si en tout nœud les hauteurs des sous-arbres gauche et droit diffèrent au plus de un.

On peut définir un indicateur de déséquilibrage:

```
entier deseq(abin *a)
{
si a==NIL alors renvoyer 0
sinon renvoyer hauteur(a->ag)-hauteur(a->ad)
}
```

alors un arbre équilibré est tel qu'en chacun de ses sous-arbres *b* on a: deseq(b)={-1, 0, 1}

Proposition: tout arbre équilibré ayant *n* nœuds a une hauteur *h* vérifiant:

$$\log(n+1) \leq h < 1.44\log(n+2)$$

Pour effectuer un rééquilibrage d'arbre on va recourir à des rotations.

La structure de données consiste évidemment à stocker en chaque nœud la hauteur de l'arbre correspondant, de façon à pouvoir calculer deseq ***sans avoir à faire de parcours explicite***, ce qui s'avérerait beaucoup trop couteux :

```
type abin = {
     T         etiq;
     Nat       h;
     abin      *ag, *ad;
}
```

L'opération d'enracinement doit alors veiller à initialiser ce champ « hauteur » :

```
abin *enraciner(T x, abin *g, abin *d)
{
     abin *nouv := nouveau abin;
     nouv->etiq:=x;
     nouv->ag:=g;
     nouv->ad:=d;
     nouv->h:=1+max(g!=NULL ? g->h:-1, d!=NULL ? d->h:-1)
}
```

1.1 Rotations

On peut définir une rotation simple gauche ou droite ainsi qu'une rotation double gauche-droite ou droite-gauche. Toutes ces rotations transforment un arbre binaire de recherche en un

arbre binaire de recherche. Une rotation double consiste en fait en une rotation d'un sous-arbre suivi d'une rotation de l'arbre. Les figures 1 et 2 illustrent deux exemples de rotations.

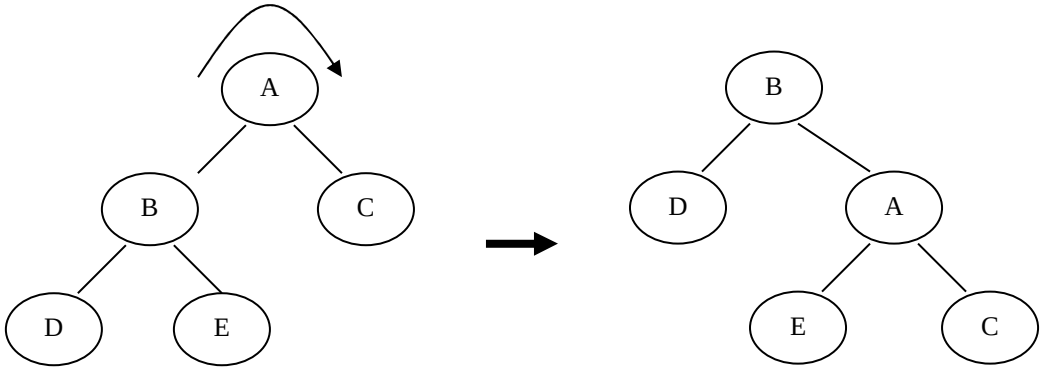
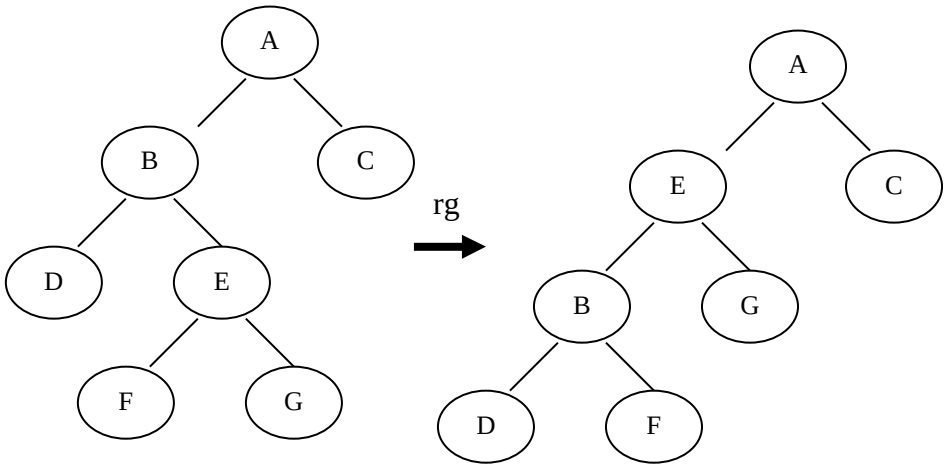


Figure 1: rotation droite (rd)



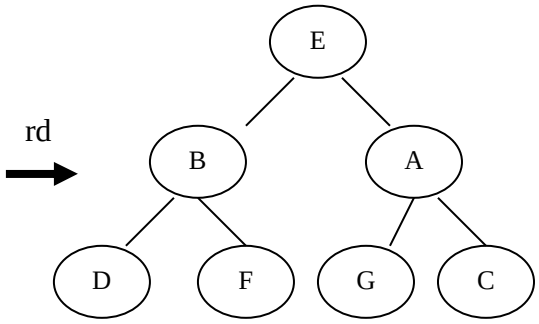


Figure 2: double rotation gauche-droite (rgd)

Ces deux opérations sont spécifiées comme suit :

rd, rgd : abin → abin

pré(rd(a)) ≡ non vide(ag(a))
pré(rgd(a)) ≡ non vide(ag(a)) et non vide(ad(ag(a))

rd(enrac(x,a1,a2)) = enrac(racine(a1),ag(a1), enrac(x,ad(a1),a2))
rgd(enrac(x,a1,a2)) = rd(enrac(x,rg(a1),a2))

Il est clair que l'on peut définir symétriquement une rotation gauche et une double rotation droite-gauche.

Voici le code correspondant à ces quatre opérations:

```
abin *rd(abin *a)
{
abin *temp;

si a == NULL alors // erreur
si a->ag == NULL alors // erreur
temp := a->ag
a->ag := temp->ad
temp->ad := a
renvoyer temp
}

abin *rg(abin *a)
{
abin *temp;

si a == NULL alors // erreur
si a->ad == NULL alors // erreur
temp := a->ad
a->ad := temp->ag
temp->ag := a
renvoyer temp
}

abin *rgd(abin *a)
{
a->ag := rg(a->ag)
renvoyer rd(a)
}

abin *rdg(abin *a)
{
a->ad := rd(a->ad)
renvoyer rg(a)
}
```

Attention : *pour être complet, il faut ajouter aux codes de rd() et rg() la mise à jour du champ hauteur en chaque nœud. Dans l'exemple de la figure 1, D, E et C ne changent pas leur hauteur, en revanche il faut mettre à jour les hauteurs en A et B, simplement en calculant les max.*

1.2 Arbres de type: Adelson – Velskii & Landis (AVL)

Il s'agit d'un arbre binaire de recherche toujours maintenu équilibré après chaque opération d'insertion ou de suppression. Un AVL est donc un arbre binaire de recherche équilibré.

Si on utilise pour les AVL les opérations de mise à jour vues précédemment pour les arbres binaires (insertion en feuille / racine, suppression), on risque de déséquilibrer l'arbre. Il faut donc prévoir un rééquilibrage à chaque mise à jour.

Nous allons définir une procédure de rééquilibrage (ne fonctionnant que sur un arbre AVL auquel on aurait inséré en feuille / supprimé un seul élément). Soit h(a) l'opération calculant la hauteur d'un arbre, tel que h(anouv)=0 et h(a)=max(h(ag(a)),h(ad(a))).

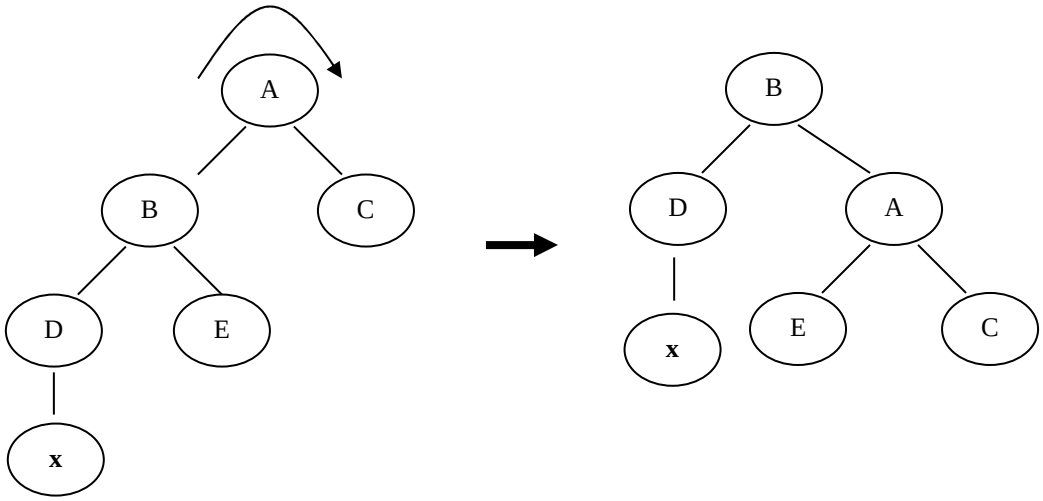
Examinons une stratégie de rééquilibrage à chaque insertion en feuille. Soit a=enrac(y, a1, a2) un AVL. Supposons que l'insertion de l'élément x a lieu sur une feuille de a1 **et qu'elle fait augmenter de 1 la hauteur de a1 et que a1 reste un AVL**, alors en désignant par ai le nouvel arbre obtenu après insertion de x dans a, on peut distinguer trois cas de figure :

- Si deseq(a) valait 0 alors deseq(ai) vaudra 1, mais a reste un AVL avec h(ai)=h(a)+1
- Si deseq(a) valait -1 alors deseq(ai) vaudra 0, mais a reste un AVL avec h(ai)=h(a)
- Si deseq(a) valait +1 alors deseq(ai) vaudra +2, et a n'est plus un AVL, h(ai)=h(a)+1

Dans cette dernière hypothèse, deux cas seulement doivent être considérés pour procéder à un rééquilibrage :

1. Si le déséquilibre de a1 (sous-arbre gauche de a) est passé de 0 à 1, alors il suffit de rééquilibrer ai par une rotation droite.
2. Si le déséquilibre de a1 (sous-arbre gauche de a) est passé de 0 à -1, alors il suffit de rééquilibrer ai par une double rotation gauche-droite.

La figure ci-dessous représente ces deux cas respectifs :



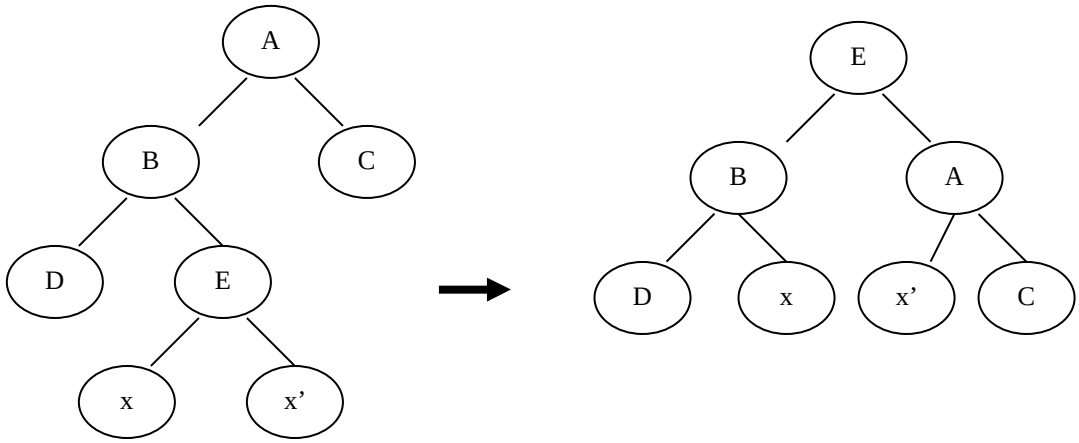


Figure 3: ici, x ou x’ est ajouté en feuille dans a1 (sous arbre gauche), puis un rééquilibrage est effectué. Seuls ces deux cas nécessitent un rééquilibrage.

Dans les deux cas, l’arbre obtenu après rééquilibrage est un AVL

L’insertion dans le sous-arbre droit fonctionne de la même manière : deux cas sont possibles et peuvent être résolus par rotation gauche ou double rotation droite-gauche.

L'insertion dans un arbre de type AVL se fait en feuille suivi d'un rééquilibrage:

```
abin *insavl(T x, abin *a)
{
si a==NIL alors renvoyer enraciner(x, NIL, NIL)
si a->etiq < x alors
     a->ad := insavl(x, a->ad)
sinon
     a->ag := insavl(x, a->ag)
renvoyer reeq(a)
}
```

Finalement on définit la fonction de rééquilibrage en fonction des 4 cas précédents :

```
abin *reeq(abin *a)
{
si deseq(a)=0, -1 ou 1 alors renvoyer a
si deseq(a)=+2 et deseq(a->ag)=+1 alors renvoyer rd(a)
si deseq(a)=-2 et deseq(a->ad)=-1 alors renvoyer rg(a)
si deseq(a)=+2 et deseq(a->ag)=-1 alors renvoyer rgd(a)
si deseq(a)=-2 et deseq(a->ad)=+1 alors renvoyer rdg(a)
}
```

On voit que toute insertion dans un AVL nécessite au plus une rotation pour le rééquilibrage.

Complexité : Compte tenu qu'il ne faut qu'une rotation, la complexité au pire est en θ(log n). L'analyse en moyenne est complexe et non complètement résolue. Les expériences montrent qu'il faut en moyenne une rotation simple ou double toutes les deux adjonctions.
Pour la suppression, le principe reste le même que pour les arbres binaires de recherche classiques. Mais il faut maintenir l'équilibre. Ce maintient peut nécessiter plusieurs rotations successives, qui peuvent éventuellement même remonter jusqu'à la racine.

La spécification de la suppression est donnée par :

sup : abin s → abin

sup(anouv,x) = anouv
sup(enrac(y,a1,a2), x) =
si x=y alors si vide(a2) alors a1
sinon si vide(a1) alors a2
sinon reeq(enrac(max(a1),otermax1(a1),a2))
sinon si x<y alors reeq(enrac(y, sup(a1,x), a2))
sinon reeq(enrac(y, a1, sup(a2,x)))

Comme on peut le voir, l'opération de suppression est définie comme pour les arbres binaires de recherche classiques. otermax1 est une version otermax avec rééquilibrage :

otermax1(enrac(y,a1,a2)) = si vide(a2) alors a1 sinon reeq(enrac(y, a1, otermax1(a2)))

Complexité : une suppression dans un AVL peut entrainer jusqu'à 1,5log(n) rotations, mais la complexité au pire reste θ(log n). L'analyse en moyenne reste un problème ouvert. Des expériences montrent qu'en moyenne il y a seulement une rotation pour 5 suppressions, ce qui va à l'encontre de l'intuition qu'une suppression est beaucoup plus couteuse qu'une adjonction.

2. Arbres binaires rouge et noir

Définition: Un arbre binaire de recherche est un arbre rouge et noir s'il satisfait les propriétés suivantes :

1. Chaque noeud est soit rouge, soit noir.
2. Chaque feuille (NIL) est noire.
3. Si un noeud est rouge, alors ses fils sont noirs.
4. Tous les chemins descendants reliant un noeud donné à une feuille (du sous-arbre dont il est la racine) contiennent le même nombre de noeuds noirs.

Dans un tel arbre, on remplace les pointeurs NULL vers des sous-arbres vides par des feuilles dont la valeur vaut ω (une valeur d'étiquette en dehors du domaine de définition). Dans la figure ci-dessous, les feuilles NIL sont représentées par des petits carrés noirs. On les appelle aussi **nœuds externes**. Les **nœuds internes** sont ceux qui ne sont pas externes. Les nœuds internes portent les données.

On appelle hauteur noire d'un noeud x le nombre de noeuds noirs sur un chemin descendant de x à une feuille NIL. La hauteur noire est donc constante pour toutes les feuilles NIL.

La condition (4) est une condition d'équilibre. Elle signifie que si l'on « oublie » les nœuds rouges d'un arbre, on obtient un arbre binaire parfaitement équilibré. Notez qu'avec les nœuds rouges, l'arbre n'est pas forcément bien balancé au sens des AVL (donc deseq peut valoir plus que +2 ou moins que -2).

La figure suivante illustre un exemple d'arbre rouge noir:

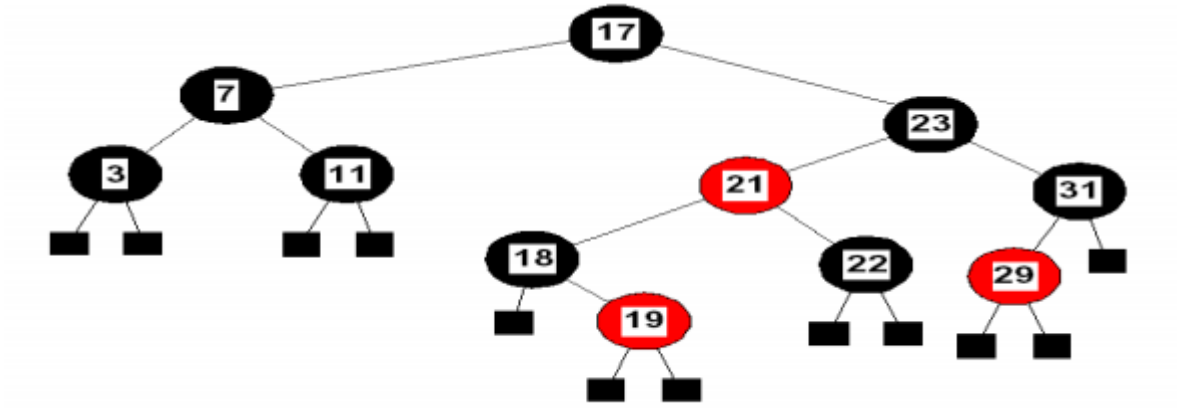


Figure 1 : un exemple d'arbre rouge et noir. Toutes les feuilles NIL ont 3 parents noirs en remontant jusqu'à la racine (4) et tout nœud rouge n'a que des fils noirs (3).

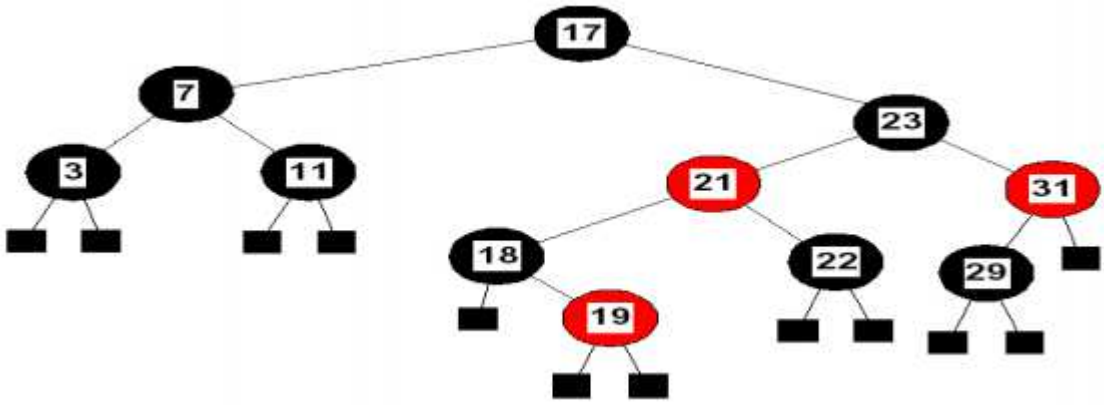


Figure 2 : un exemple d'arbre qui n'est pas rouge et noir. Ici le nœud 31 est rouge. Son fils droit n'a pas une hauteur noire égale à 3 mais égale à 2. La condition (4) n'est pas vérifiée.

2.1 Propriétés et hauteur

La hauteur noire d'un nœud x notée $h_n(x)$ est le nombre de nœuds noirs dans un chemin du nœud à une de ses feuilles (le nœud x n'est pas compris). La hauteur noire d'un arbre rouge-noir est la hauteur noire de sa racine.

Prop : Un arbre rouge-noir contenant n nœuds internes a une hauteur h inférieure ou égale à $2 \times \log2(n+1)$.

Preuve : Montrons d'abord que le nombre de nœuds internes du sous-arbre enraciné en x est au moins égal à $2^{hn(x)} - 1$. On procède par induction. Si la hauteur noire de x est 0, alors c'est le nœud racine, et le sous arbre enraciné en n contient 0 nœuds internes. Donc, la condition est vérifiée, on a bien $2^{hn(x)} - 1>=0$. Si la hauteur noire de x est >0, alors chacun de ses fils a une hauteur noire égale soit à $h_n(x)$ s'il est rouge, soit à $h_n(x)-1$ s'il est noir. Donc, en appliquant l'hypothèse d'induction aux deux sous arbres de x, le sous arbre enraciné en x contient au moins $2\times(2^{hn(x)}-1-1)+2 = 2^{hn(x)}$ nœuds internes.

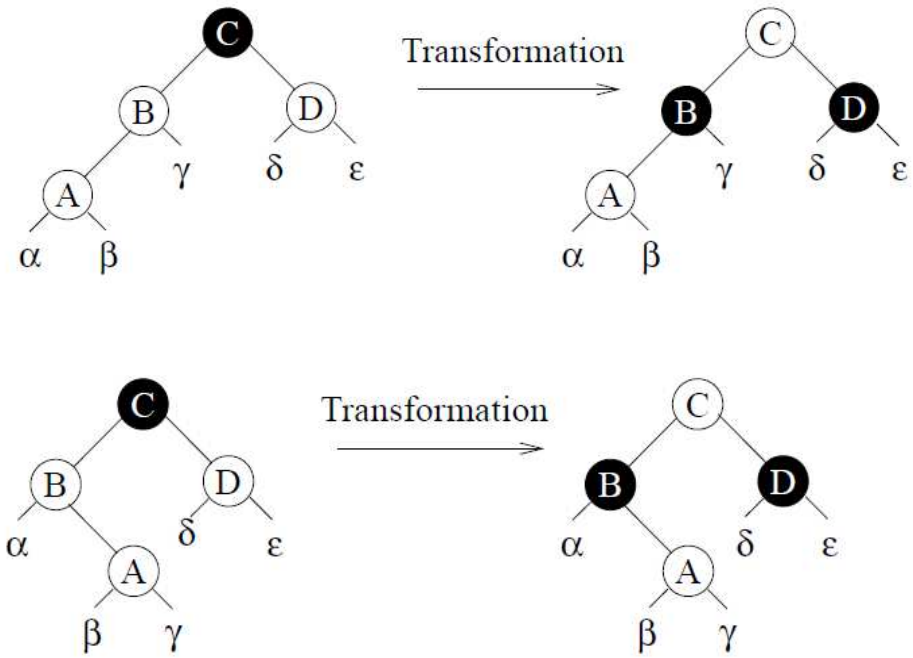
Soit h la hauteur d'un arbre rouge-noir, la moitié au moins des nœuds vers une feuille doit être noirs. Donc la hauteur noire d'un arbre rouge-noir est au moins h/2. En utilisant la relation précédente, nous pouvons donc écrire: $n>= 2^{hn(racine)} - 1$, d'où $\log(n+1)>=h_n(racine)>=h/2$ et donc $h<=2\times\log2(n+1)$.

2.2 Insertion

L'insertion d'une valeur dans un arbre rouge et noir commence par l'insertion usuelle d'une valeur dans un arbre binaire de recherche. Le nouveau nœud est rouge de telle sorte que la propriété (4) reste vérifiée. En revanche, la propriété (3) n'est plus nécessairement vérifiée : notamment si le père du nouveau nœud est rouge. Donc si le père du nouveau nœud est également rouge, l'algorithme doit effectuer des modifications dans l'arbre, par exemple à l'aide de rotations. Ces modifications ont pour but de rééquilibrer l'arbre.

On distingue plusieurs cas posant problème :

Cas 1.a et 1.b :

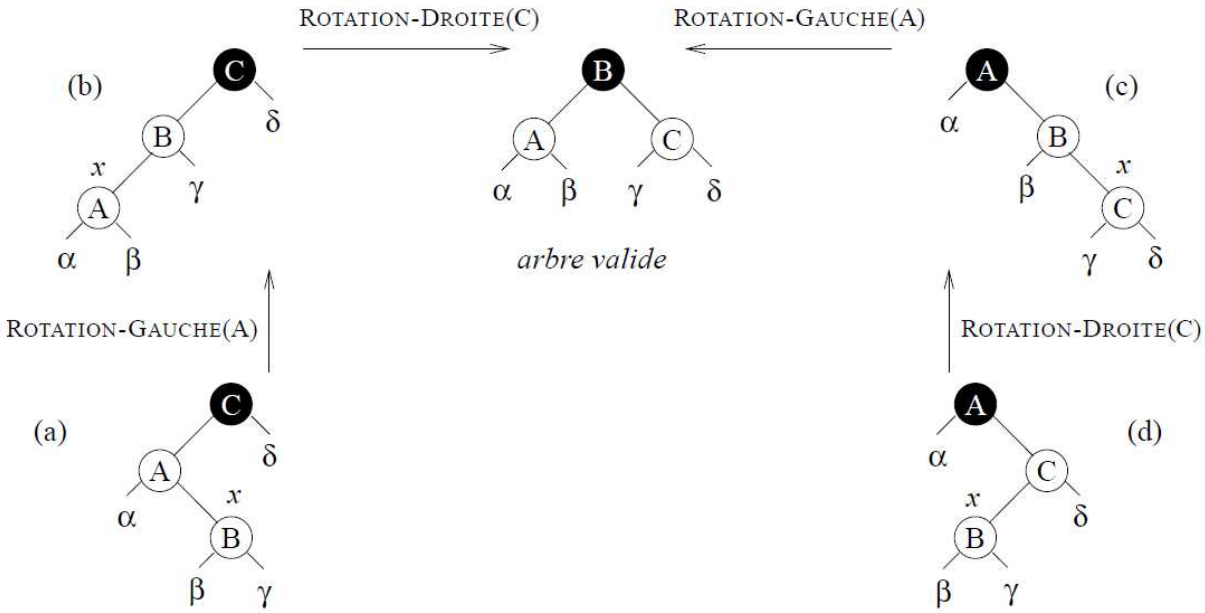


Dans ces deux cas, on ne change que la couleur des nœuds.

Les autres cas nécessitent des rotations. Le schéma ci-dessous représente les transformations qu'il est nécessaire d'appliquer dans ces cas. Dans ce schéma, les sous-arbres α, β, γ, et δ sont

tous de racine noire et ont tous la même hauteur noire (et les transformations d'un cas à l'autre préservent cette propriété).

Cas 2.a, 2.b, 2.c et 2.d :



Dans cette figure le nœud marqué « x » est celui qui pose problème.

Il en découle l'algorithme que nous décrivons par la suite. Nous écrivons une version itérative qui remonte dans l'arbre depuis le nouveau nœud ajouté, jusqu'à la racine. Pour cela on définit une opération *adjfeuille* qui non seulement ajoute une feuille à partir de la racine mais renvoie également la liste L des nœuds parcourus. Cette liste permet, pour chaque nœud parcouru de retrouver son père ou son grand-père (ce sont les suivants dans la liste). La queue de la liste est la racine de l'arbre (point de départ du parcours) et la tête de liste est la feuille nouvellement ajoutée (point d'arrivée du parcours).

Dans cet algorithme père(x), où x est un nœud de l'arbre, correspond à suivant(x,l) et grand-père(x) à suivant(suivant(x,l), l) où l représente la liste.

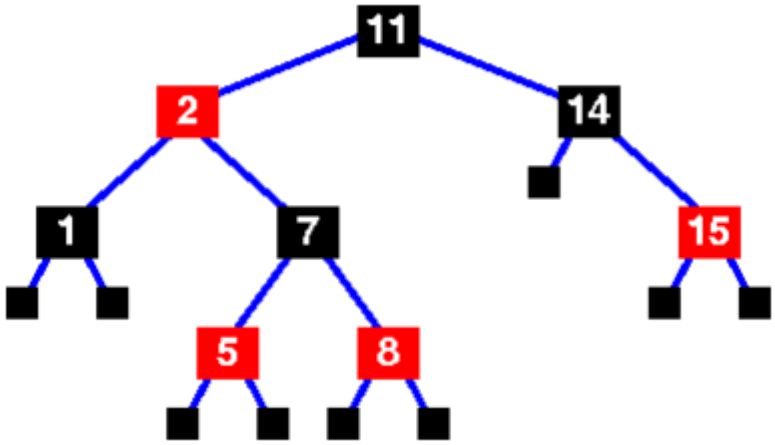
Dans les algorithmes présentés on notera : *gauche* et *droit* pour désigner le fils gauche et droit (les sous-arbres), et *couleur* pour désigner la couleur du nœud. On utilise aussi *rg* et *rd* pour les rotations gauches et droites.

```
INSERTION_ARN(A, x)
        L = adjfeuille(A, x)
        couleur(x) := ROUGE
        tant que x ≠ racine(A) et couleur(père(x)) = ROUGE faire
                // selon que le père est à gauche ou à droite du grand-père
                si père(x) = gauche(grand-père(x)) alors
                        y := droit(grand-père(x))
                        si couleur(y) = ROUGE alors
```

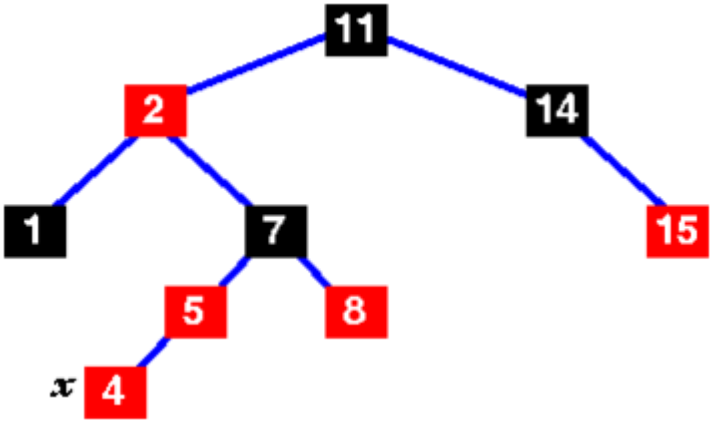
```
                    couleur(père(x)) := NOIR
                    couleur(y) := NOIR
                    couleur(grand-père(x)) := ROUGE
                    x := grand-père(x)
                sinon
                    si x = droit(père(x)) alors
                        x := père(x)
                        rg(x)
                    couleur(père(x)) := NOIR
                    couleur(grand-père(x)) := ROUGE
                    rd(grand-père(x))
            sinon
                // même chose que précédemment en échangeant droit et gauche
                // cad si père(x) = ad(grand-père(x)) alors
                y := gauche(grand-père(x))
                si couleur(y) = ROUGE alors
                    couleur(père(x)) := NOIR
                    couleur(y) := NOIR
                    couleur(grand-père(x)) := ROUGE
                    x := grand-père(x)
                sinon
                    si x = gauche(père(x)) alors
                        x := père(x)
                        rd(x)
                    couleur(père(x)) := NOIR
                    couleur(grand-père(x)) := ROUGE
                    rg(grand-père(x))
        fin tantque
        couleur(racine(A)) := NOIR
```

Voici un exemple d'application de cet algorithme.

L'arbre rouge-noir initial est le suivant et l'on souhaite insérer la valeur 4 :



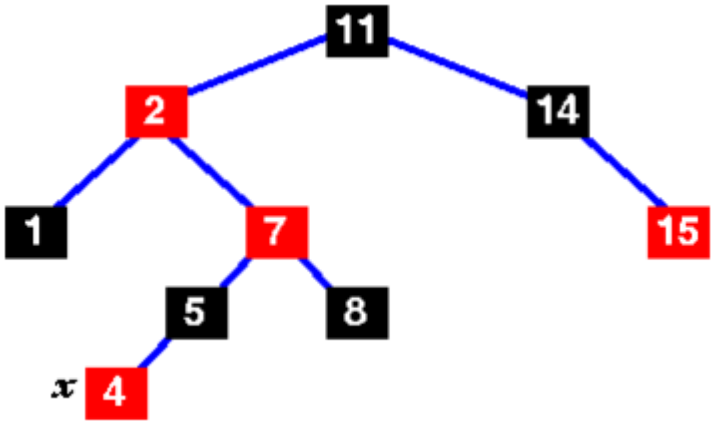
L'insertion se fait en feuille en suivant un chemin dichotomique.



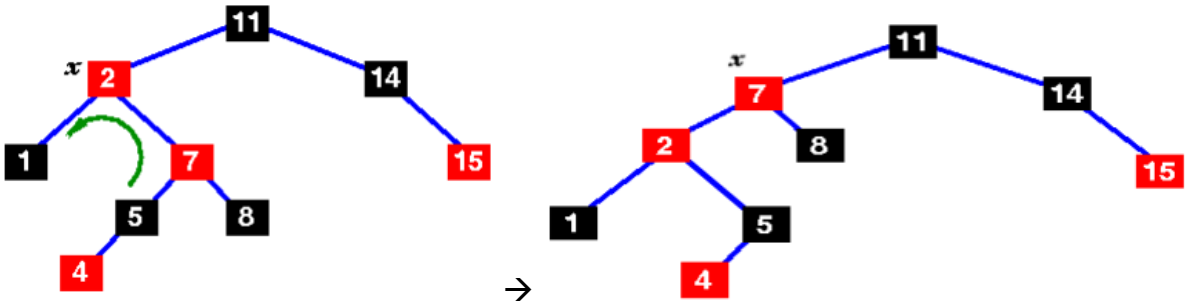
La liste L obtenue est :

L = { 4, 5, 7, 2, 11 }

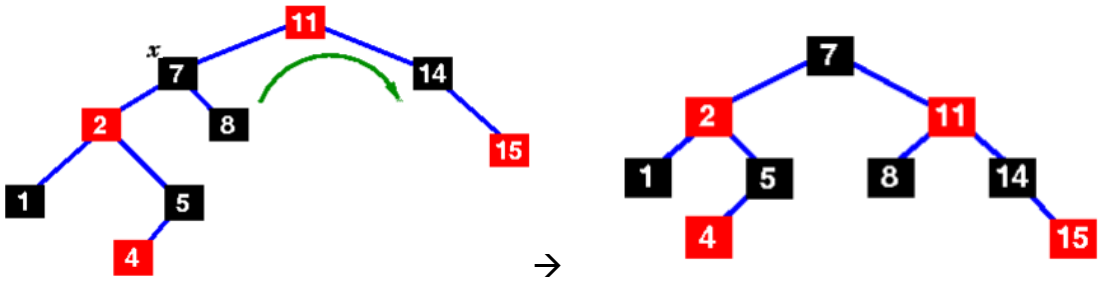
Grâce à cette liste on récupère le père et le grand-père de 4, celui de 5, etc.
On voit qu'après insertion de 4, on n'a plus un arbre rouge et noir : le fils de 5 n'est pas noir. On applique le cas 1, d'où :



On monte x au niveau du grand-père, x :=grand-pere(x), c'est-à-dire que l'on se place sur le nœud 7. On voit que là non plus, ce n'est pas un arbre rouge-noir car le père de 7 est rouge également. Comme x = ad(père(x)) nous montons sur le père et appliquons une rotation gauche :



Ensuite nous appliquons les deux changements de couleur : couleur(père(7)) := NOIR et couleur(grand-père(7)) := ROUGE et terminons par une rotation droite : rd(grand-père(7)) :



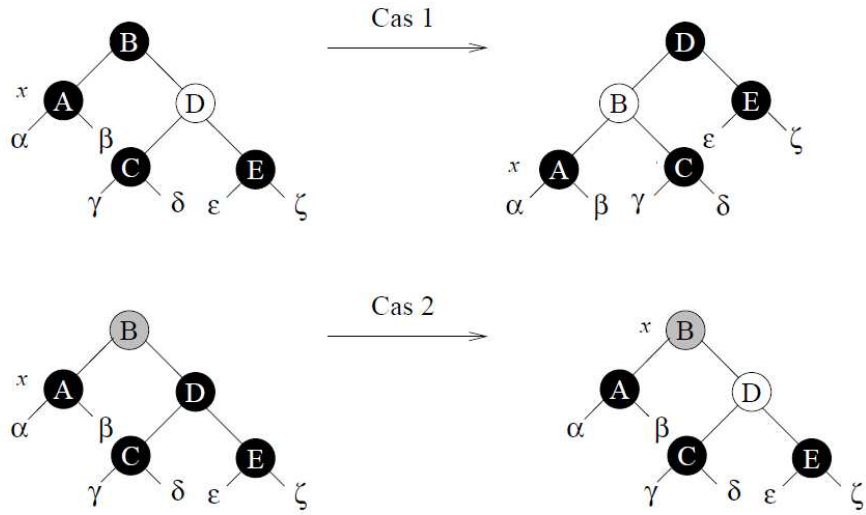
A la suite de cette dernière rotation x se trouve en 7 et c'est la racine. Donc l'algorithme se termine et l'arbre final est bien rouge-noir.

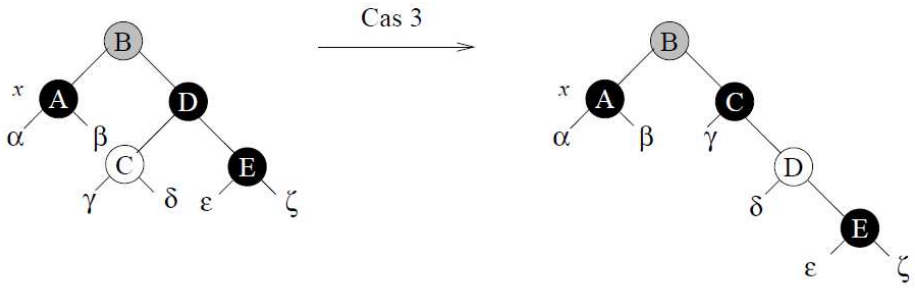
2.3 Suppression

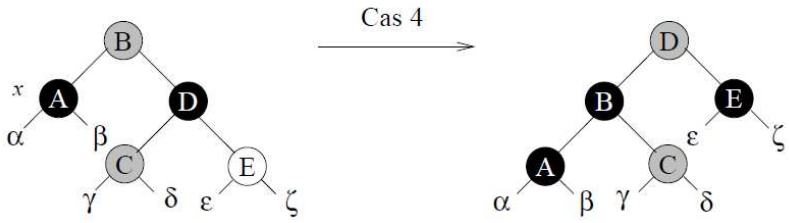
Comme pour l'insertion d'une valeur, la suppression d'une valeur dans un arbre rouge et noir commence par supprimer un nœud comme dans un arbre binaire de recherche. Si le nœud qui porte la valeur à supprimer possède zéro ou un fils, c'est ce nœud qui est supprimé et son éventuel fils prend sa place.

Si, au contraire, ce nœud possède deux fils, il n'est pas supprimé. La valeur qu'il porte est remplacée par la valeur suivante dans l'ordre et c'est le nœud qui portait cette valeur suivante qui est supprimé. Ce nœud supprimé est le nœud au bout de la branche gauche du sous-arbre droit du nœud qui portait la valeur à supprimer. Il s'agit du minimum des « plus grands » que la valeur à supprimer. Ce nœud « minimum » n'a pas de fils gauche par définition.

Si l'élément supprimé était de couleur rouge, aucune des propriétés des arbres rouge et noir n'est violée. Cependant, si le noeud supprimé était noir la propriété 4 (tous les chemins descendants d'un noeud à une feuille contiennent le même nombre de noeuds noirs) peut être violée. Il nous faut donc rajouter un noir sur tous les chemins perturbés. Pour ce faire, on rajoute un noir à l'unique fils du noeud supprimé. Si ce fils était rouge, l'arbre obtenu est un arbre rouge et noir. Si ce fils était déjà noir, on a deux « noirs » empilés sur un même noeud et il nous faut les répartir. La figure ci-dessous présente les différents cas de figure possibles et les méthodes de résolutions associées. Si le nœud supprimé n'avait pas de fils, on rajoute un « noir » à la feuille NIL correspondante de son père. Pour pouvoir réaliser cette manipulation, **on utilise une sentinelle** : un noeud spécial valant NIL et qui permet de ne pas traiter à part les feuilles NIL.







Ici, les noeuds à fond noir sont des noeuds « noirs », ceux à fond blanc sont « rouges » et ceux à fond grisé sont soit « noirs » soit « rouges ».

(1) Ce cas est transformé en cas 2, 3 ou 4.
(2) Le noir est remonté sur le père de x, l'oncle de x étant repeint en rouge ; si le père de x était rouge l'arbre est de nouveau valide, sinon on rapplique l'algorithme de correction cette fois-ci sur le père de x.
(3) Ce cas est transformé en cas 4.
(4) Le noir supplémentaire est éliminé par rotation gauche sur le père de x et recoloriage du père et de l'oncle de x.

L'algorithme de suppression est le suivant :

```
ARN-SUPPRESSION(A, x)
    L = recherche(A,x)  // renvoie la liste parcourue de la racine jusqu'à x
    // traiter selon que x est feuille, ou n'a qu'un seul fils ou deux fils
    si gauche(x) = NIL et droit(x) = NIL alors // cas x feuille : les deux fils sont NIL
        si père(x) = NIL alors racine(A) := NIL
        sinon
            si x = gauche(père(x))
                alors gauche(père(x)) := NIL
                sinon droit(père(x)) := NIL
            si couleur(x) = NOIR alors
                père(NIL) :=  père(x)
                ARN-CORRECTION(A, x)
    sinon si gauche(x) = NIL ou droit(x) = NIL alors // cas l'un des deux fils est NIL
        avec filsde_x faire
        si gauche(x)  ≠ NIL
            alors filsde_x := gauche(x)
            sinon filsde_x := droit(x)
        père(filsde_x) := père(x)
        si père(x) = NIL
            alors racine(A) := filsde_x
            sinon si gauche(père(x)) = x
```

```
                    alors gauche(père(x)) := filsde_x
                    sinon droit(père(x)) := filsde_x
            si couleur(x) = NOIR alors ARN-CORRECTION(A, filsde_x)
    sinon // cas aucun des fils n'est NIL, on cherche alors le minimum
            min := MINIMUM(droit(x))
            etiquette(y) := etiquette(min)
            ARN-SUPPRESSION(A,min)
    renvoyer A
```

Il nous faut à présent définir la correction :

```
RN-CORRECTION(A, x)
    avec w faire
    tant que x ≠ racine(A) et couleur(x) = NOIR faire
        // selon que x est fils gauche ou fils droit
        si x = gauche(père(x)) alors
            w := droit(père(x))
            si couleur(w) = ROUGE alors          // CAS 1
                couleur(w) := NOIR
                couleur(père(w)) := ROUGE
                rg(A, père(x))
                w := droit(père(x))
            si couleur(gauche(w)) = NOIR
                        et couleur(droit(w)) = NOIR alors  // CAS 2
                couleur(w) := ROUGE
                x := père(x)
            sinon
                si couleur(droit(w)) = NOIR alors   // CAS 3
                    couleur(gauche(w)) := NOIR
                    couleur(w) := ROUGE
                    rd(A, w)
                    w := droit(père(x))
                // CAS 4
                couleur(w) := couleur(père(x))
                couleur(père(x)) := NOIR
                couleur(droit(w))) := NOIR
                rg(A, père(x))
                x := racine(A)
        sinon
            // même chose que précédemment en échangeant droit et gauche
            // si x = droit(père(x)) alors
            w := gauche(père(x))
            si couleur(w) = ROUGE alors          // CAS 1
                couleur(w) := NOIR
                couleur(père(w)) := ROUGE
                rd(A, père(x))
                w := gauche(père(x))
            si couleur(droit(w)) = NOIR
                        et couleur(gauche(w)) = NOIR alors  // CAS 2
                couleur(w) := ROUGE
```

```
                    x := père(x)
                sinon
                    si couleur(gauche(w)) = NOIR alors   // CAS 3
                        couleur(droit(w)) := NOIR
                        couleur(w) := ROUGE
                        rg(A, w)
                        w := gauche(père(x))
                    // CAS 4
                    couleur(w) := couleur(père(x))
                    couleur(père(x)) := NOIR
                    couleur(gauche(w))) := NOIR
                    rd(A, père(x))
                    x := racine(A)
        couleur(x) := NOIR
    fin tantque
```

2.4 Complexité

Les arbres rouge et noir sont relativement équilibrés : la hauteur d'un arbre rouge et noir est au pire du double de celle d'un arbre binaire parfaitement équilibré. Toutes les opérations sur les arbres rouge et noir sont de coût O(h), c'est-à-dire O(log(n)).
.

3. Arbres de recherche a – b et b-arbres

3.1 Définition: C'est un arbre de recherche généralisé. Cet arbre satisfait les propriétés suivantes :

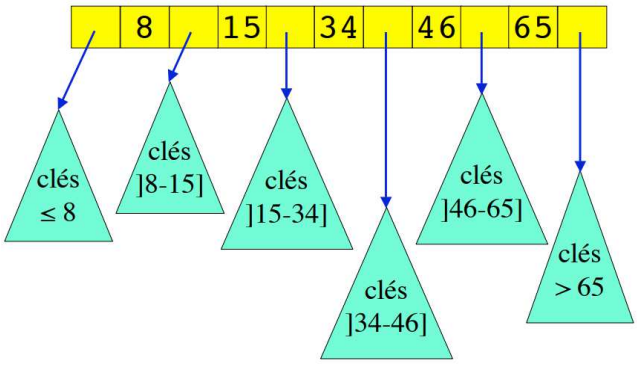
• Les feuilles ont toutes la même profondeur.
• La racine a au moins 2 et au plus b fils (sauf si l'arbre est réduit à sa racine).
• Les autres nœuds internes ont au moins a et au plus b fils.

En toute généralité, on définit alors un b-arbre de la manière suivante : chaque nœud, sauf la racine, possède un minimum de a−1 clés (appelées aussi éléments), un maximum de b−1 clés et au plus b fils. Pour chaque nœud interne — nœud qui n'est pas une feuille —, le nombre de fils est toujours égal au nombre de clés augmenté d'une unité. Si n est le nombre de fils, alors on parle de n-nœud. Un a-b arbre ne contient que des n-nœuds avec $a \leq n \leq b$. Souvent on choisit la configuration $a = t$ et $b = 2 \times t$: t est appelé le degré ou ordre du b-arbre. Un exemple est l'arbre 2-3-4, pour lequel le degré est 2.

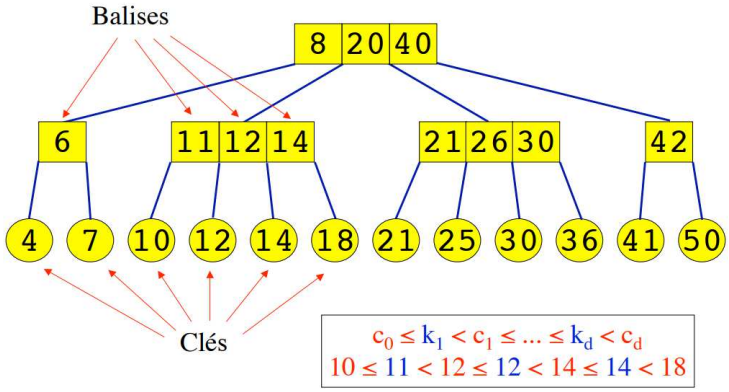
De plus, la construction des arbres B garantit qu'un arbre B est toujours équilibré. Chaque clé d'un nœud interne est en fait une borne qui distingue les sous-arbres de ce nœud.

En pratique, la valeur de a peut être de l'ordre de plusieurs centaines. Les étiquettes des nœuds internes sont appelées « balises » et les étiquettes des feuilles sont les « clés » (généralement ce sont les identifiants des éléments d'une banque de données).

La figure ci-dessous illustre le principe de l'arbre a-b.



La figure suivante donne deux exemples d'arbres a - b :



En réalité, nous avons déjà rencontré ce type de structure de données : lors de la définition des tables, en utilisant un partage de table : les données (entrées) sont partitionnées en sous-ensembles ordonnés en utilisant des tables mineures à plusieurs niveaux.

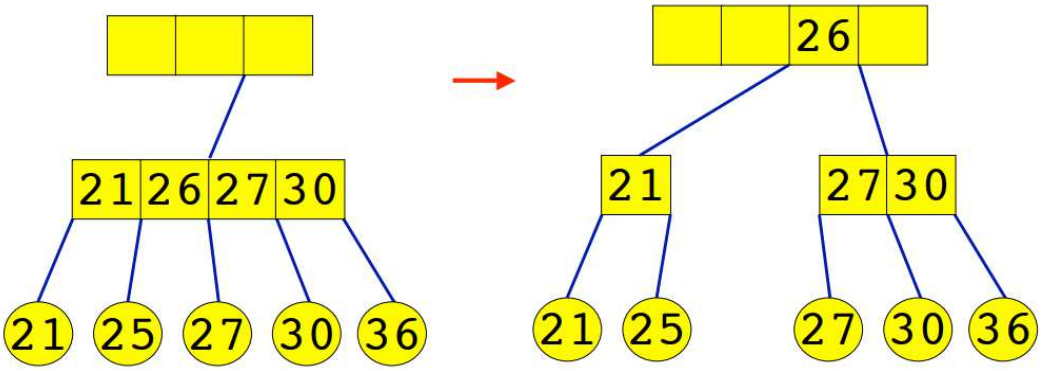
Les b – arbres sont une représentation usuelle en gestion de tables et bases de données.

Propriété : Si un arbre a-b de hauteur h contient n feuilles, alors $2a^{h-1} \leq n \leq b^h$ et donc :

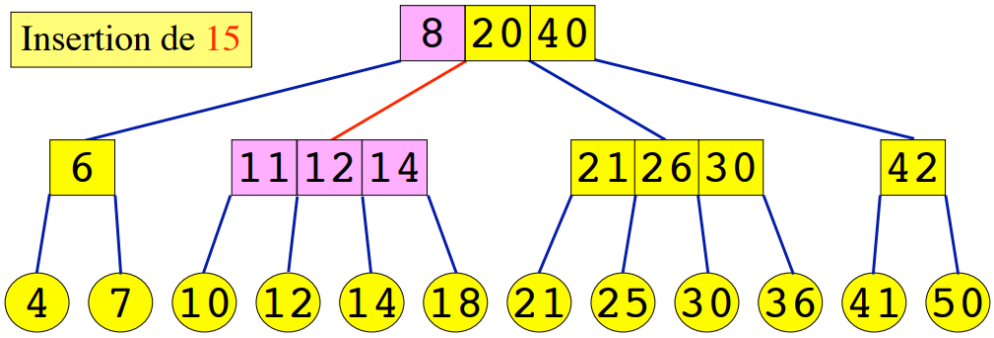
$$\log n / \log b \leq h \leq 1 + \log(n/2) / \log a$$

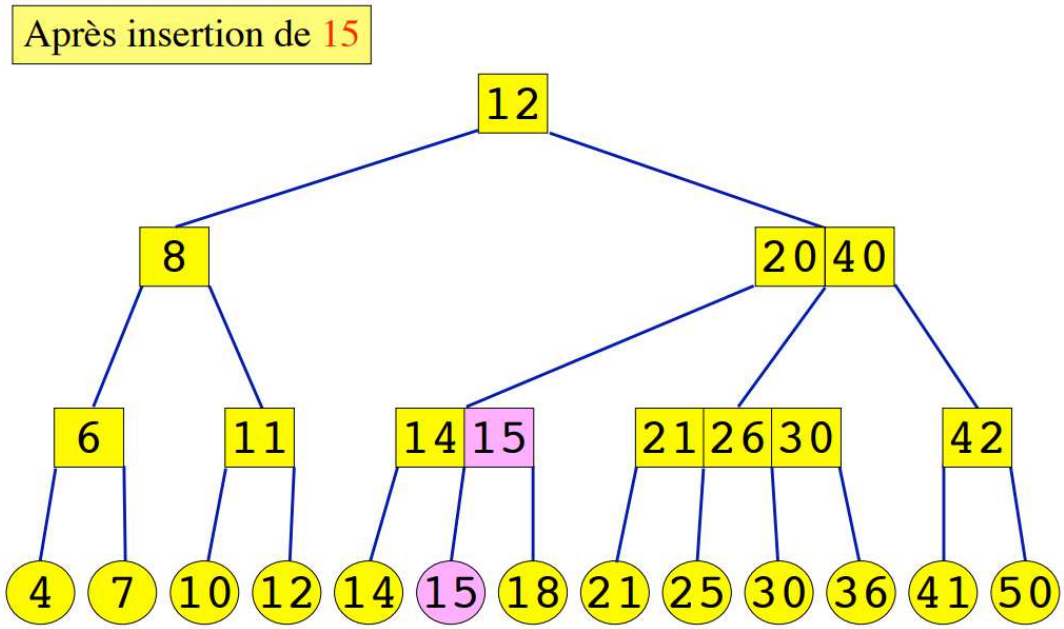
3.2 Insertion

Lorsque l'insertion d'un élément produit un nœud avec plus de b fils alors il faut « éclater » le nœud : ici, l'arbre est de type 2 – 4, or l'insertion de 25 produit 5 > 4 fils.

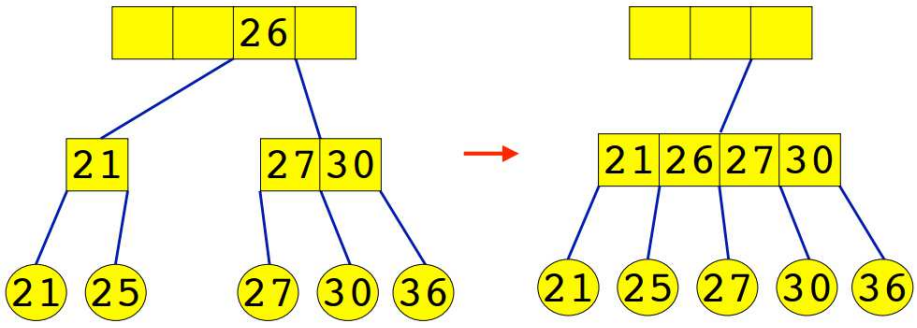


Dans certains cas, il faut propager cet éclatement et augmenter la profondeur de l'arbre :





L'opération inverse est la fusion de nœuds :



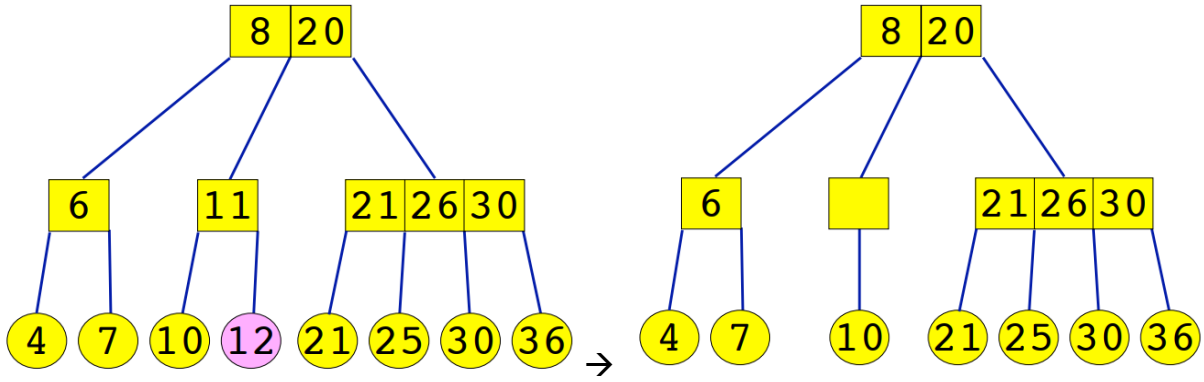
3.3 Suppression

La suppression consiste à supprimer la feuille, puis de mettre à jour les balises figurant sur le chemin de la feuille à la racine.

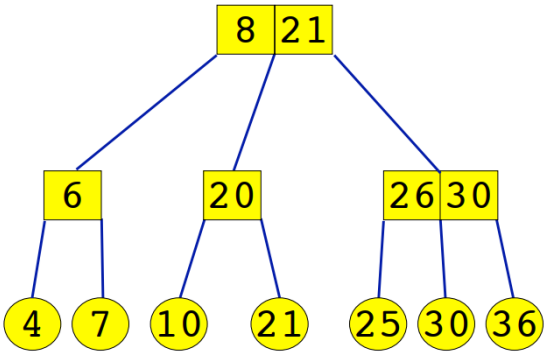
- Si les nœuds ainsi modifiés ont toujours a fils, l'arbre est encore a-b.
- Si un nœud possède seulement a -1 fils, examiner ses frères adjacents.

- Si l'un de ces frères possède au moins a + 1 fils, il suffit de faire un partage avec ce frère.
- Sinon, les frères adjacents ont a fils, la fusion avec l'un deux produit un nœud ayant 2a - 1 ≤ b fils.

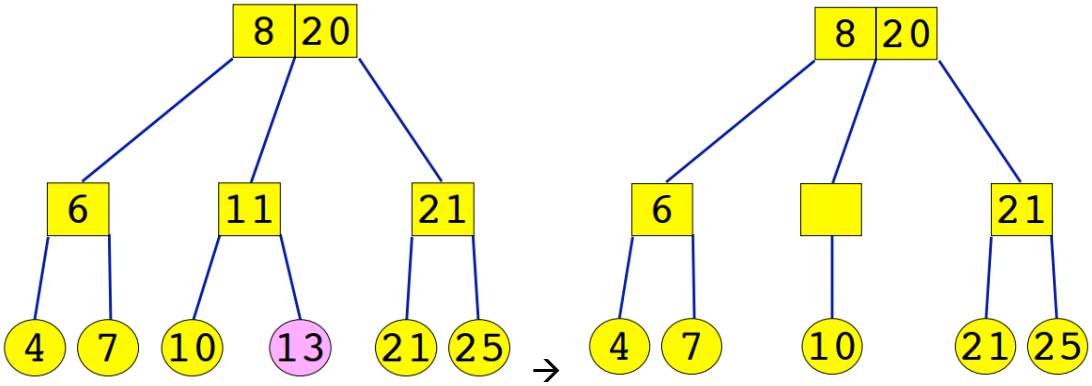
Voici un exemple : suppression de 12.



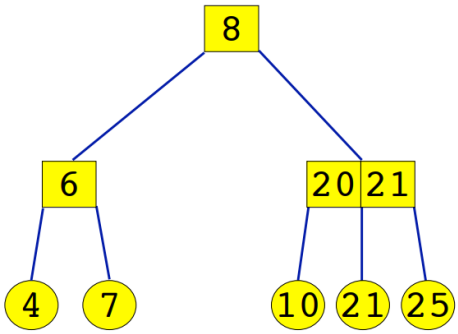
Cet arbre n'est plus un a – b arbre. Ici, la solution consiste à **partager** avec le frère droit.



L'exemple suivant (suppression de 13) montre le cas de la **fusion** avec le voisin :

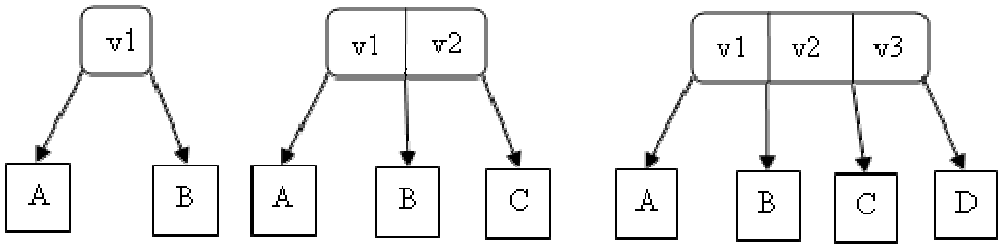


La fusion donne :



3.4 Cas particuliers de b-arbres

Un **arbre 2-3-4** est un 2-4 arbre ou b-arbre d'ordre 2, c'est-à-dire un arbre comportant uniquement des 2-nœuds, 3-nœuds et 4-nœuds (un N-nœud étant un nœud possédant N-1 clés et N fils), et dont les fils bornent les clés dans les sous arbres.



Nous étudierons ce type particulier d'arbre en TD.