Chapitre 11
Arbres équilibrés

On a montré, dans le chapitre précédent, que les opérations de recherche, adjonction et suppression dans un arbre binaire de recherche de n éléments se font en moyenne en $\Theta(\log n)$ comparaisons; mais dans le cas le pire, la complexité de ces opérations est en $\Theta(n)$: un arbre binaire de recherche peut en effet dégénérer en une liste. L'arbre binaire de recherche «idéal», pour lequel une recherche nécessite *au pire* $\log n$ comparaisons est un arbre complètement équilibré, *i.e.* un arbre dont les feuilles sont situées au plus sur deux niveaux. Mais un tel arbre peut être complètement désorganisé par une adjonction (ou une suppression), et c'est alors le coût de la réorganisation qui est prépondérant : l'arbre de la figure 1.a est complètement équilibré, mais il perd cette propriété après adjonction de l'élément 1; on a reconstruit dans la figure 1.b un arbre complètement équilibré contenant 1 : tous les éléments de cet arbre ont changé de place; ainsi la procédure de réorganisation après adjonction dans un arbre complètement équilibré peut être de complexité $\Theta(n)$.

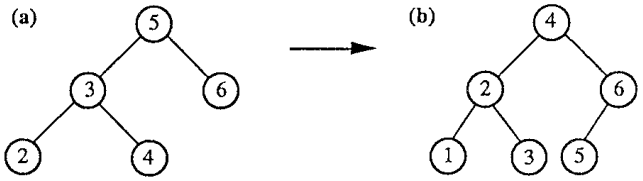


Figure 1. Réorganisation après adjonction de 1.

Pour que la réorganisation de l'arbre ne soit pas trop coûteuse, il faut assouplir les contraintes sur la forme des arbres de recherche considérés : par exemple, autoriser un léger déséquilibre en hauteur, ou autoriser les nœuds à avoir un nombre variable de fils.

On étudie dans ce chapitre diverses classes d'***arbres de recherche équilibrés***. Ces arbres sont dits équilibrés car leur hauteur est toujours une fonction logarithmique de leur nombre de nœuds; de plus, lorsqu'un arbre est «déséquilibré» après une adjonction ou une suppression, il existe des algorithmes de rééquilibrage, spécifiques à la classe à laquelle appartient l'arbre, dont la complexité est aussi logarithmique dans tous les cas. Ainsi dans chaque classe, les opérations de recherche, adjonction

et suppression sur les arbres de n éléments sont de *complexité $O(\log n)$ dans tous les cas.*

1. Rotations

Beaucoup d'algorithmes de rééquilibrage utilisent des transformations locales appelées ***rotations*** : il s'agit de faire basculer vers la droite (resp. gauche) un sous-arbre qui est trop déséquilibré vers la gauche (resp. droite). Il y a quatre types de rotation : les *rotations simples*, à droite (en abrégé *rd*, voir figure 2) et à gauche (en abrégé rg, voir figure 3), et les *rotations doubles*, rotation gauche-droite (en abrégé *rgd*, voir figure 4) et rotation droite-gauche (en abrégé *rdg*, voir figure 5).

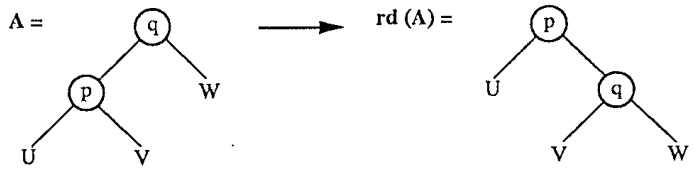


Figure 2. Rotation droite.

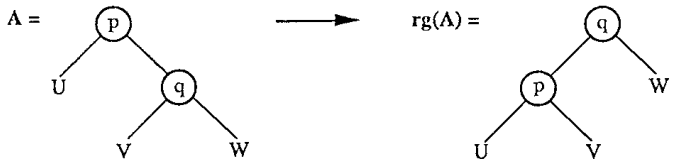


Figure 3. Rotation gauche.

Il est aisé de voir que la rotation gauche-droite sur l'arbre A (resp. droite-gauche) est composée d'une rotation gauche (resp. droite) sur le sous-arbre gauche (resp. droit) de A, suivie d'une rotation droite (resp. gauche) sur A, d'où le nom de double rotation.

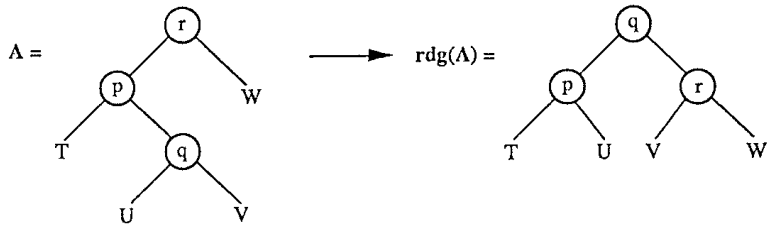


Figure 4. Rotation gauche-droite.

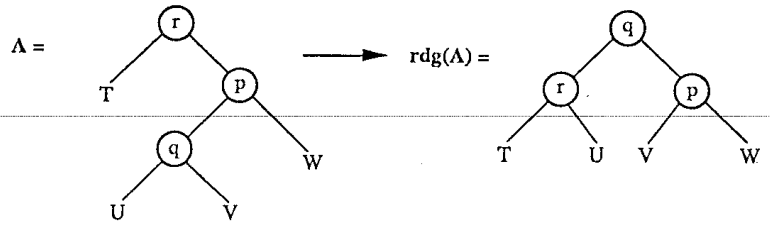


Figure 5. Rotation droite-gauche.

Ces opérations sont définies de la façon suivante :

rg : Arbre → Arbre
rd : Arbre → Arbre
rgd : Arbre → Arbre
rdg : Arbre → Arbre

Pour toutes variables A, T, U, V, W de sorte Arbre, et pour toutes variables p, q, r de sorte Nœud, on a les préconditions et les axiomes suivants :

préconditions

$rg(A)$ **est-défini-ssi** $A \neq$ *arbre-vide* & $d(A) \neq$ *arbre-vide*
$rd(A)$ **est-défini-ssi** $A \neq$ *arbre-vide* & $g(A) \neq$ *arbre-vide*
$rgd(A)$ **est-défini-ssi**
$A \neq$ *arbre-vide* & $g(A) \neq$ *arbre-vide* & $d(g(A)) \neq$ *arbre-vide*
$rdg(A)$ **est-défini-ssi**
$A \neq$ *arbre-vide* & $d(A) \neq$ *arbre-vide* & $g(d(A)) \neq$ *arbre-vide*

axiomes

$rg(< p, U, < q, V, W >>) = < q, < p, U, V >, W >$
$rd(< q, < p, U, V >, W >) = < p, U, < q, V, W >>$
$rgd(< r, < p, T, < q, U, V >>, W >) = < q, < p, T, U >, < r, V, W >>$
$rdg(< r, T, < p, < q, U, V >, W >>) = < q, < r, T, U >, < p, V, W >>$

Les procédures ci-dessous réalisent ces transformations : puisqu'une rotation gauche-droite sur A est composée d'une rotation gauche sur le sous-arbre gauche de A suivie d'une rotation droite sur A, la procédure RGD fait appel aux procédures RD et RG. Et il en est de même pour la procédure RDG. ARBRE est le type Pascal arbre binaire représenté à l'aide de pointeurs (voir chapitre 7).

```
procedure RG (var A : ARBRE);
var Aux : ARBRE;
begin Aux := A↑.d; A↑.d := Aux↑.g;
      Aux↑.g := A; A := Aux
end RG;
```

```
procedure RD (var A : ARBRE);
var Aux : ARBRE;
begin Aux := A↑.g; A↑.g := Aux↑.d;
Aux↑.d := A; A := Aux
end RD;

procedure RGD (var A : ARBRE);
begin RG(A↑.g);
RD(A)
end RGD;

procedure RDG (var A : ARBRE);
begin RD(A↑.d);
RG(A)
end RDG;
```

Remarque : Les opérations de rotation conservent la propriété d'arbre binaire de recherche. Il est immédiat de constater que la lecture symétrique de $rd(A)$(resp.$rg(A), rgd(A), rdg(A)$) donne le même résultat que la lecture symétrique de A.

2. Arbres AVL

Historiquement la première classe d'arbres équilibrés a été introduite dans les années 60 par Adelson-Velskii et Landis (d'où le nom d'AVL). En chaque nœud d'un arbre AVL, la différence des hauteurs des sous-arbres gauche et droit est égale au plus à 1 en valeur absolue. On verra que cette condition implique que la hauteur totale d'un arbre AVL est toujours inférieure à une fois et demie la hauteur d'un arbre «complètement équilibré» contenant le même nombre de nœuds. De plus, il existe des algorithmes de rééquilibrage après adjonction ou suppression qui maintiennent la propriété d'AVL en effectuant une suite de rotations sur un chemin de la racine à une feuille de l'arbre. Ainsi les opérations de recherche, adjonction et suppression ont une complexité qui reste logarithmique dans le pire des cas.

2.1. Arbres H-équilibrés

Définition : On dit qu'un arbre binaire est ***H-équilibré*** si en tout nœud de l'arbre, les hauteurs des sous-arbres gauche et droit diffèrent au plus de 1.

Soit la fonction déséquilibre sur les arbres binaires, définie récursivement par :

déséquilibre(*arbre-vide*) = 0, et
déséquilibre($< o, G, D >$) = *hauteur*(G) - *hauteur*(D)

Alors un arbre T est H-équilibré si pour tout sous-arbre S de T on a :

$$déséquilibre(S) \in \{-1, 0, 1\}$$

Exemples : Les arbres de la figure 6 sont H-équilibrés, mais l'arbre de la figure 7 ne l'est pas ; on a indiqué en chaque nœud la valeur de la fonction de déséquilibre.

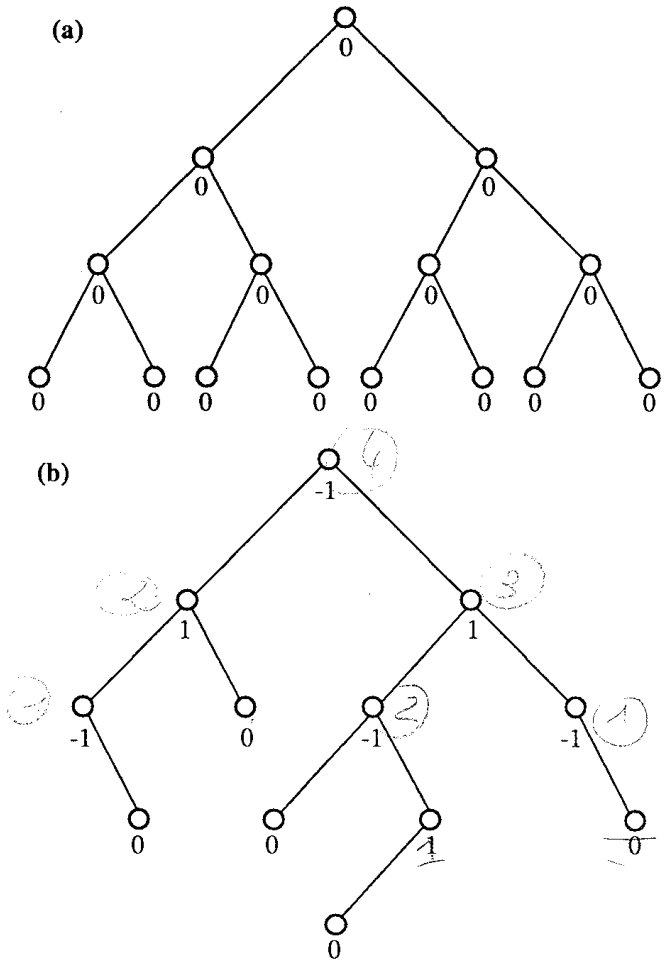


Figure 6. Arbres H-équilibrés.

La propriété suivante montre que les contraintes de déséquilibre en chaque nœud permettent de limiter la hauteur totale de l'arbre : la hauteur d'un arbre H-équilibré de n nœuds est toujours de l'ordre de $\log n$.

Propriété : Tout arbre H-équilibré ayant n nœuds a une hauteur h vérifiant :

$$\log_2(n+1) \leq h+1 < 1,44 \log_2(n+2)$$

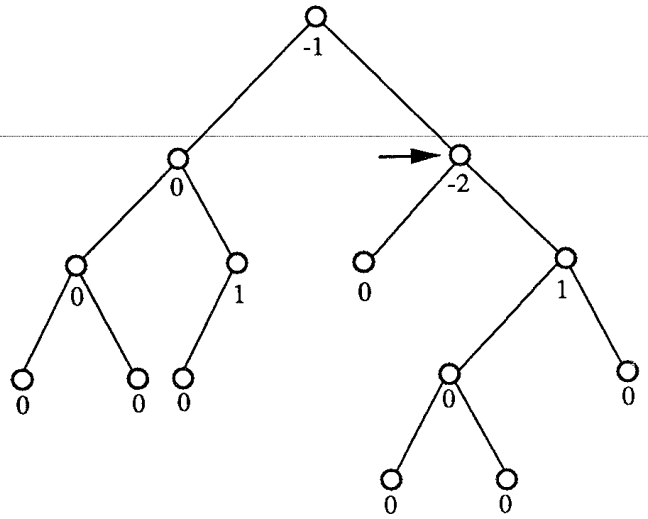


Figure 7. Arbre non H-équilibré.

Preuve :

a) Pour une hauteur donnée h, l'arbre H-équilibré contenant le plus de nœuds est celui dont tous les niveaux sont complètement remplis (déséquilibre 0 en chaque nœud). C'est un arbre complet (cf. figure 6.a), qui a donc $2^{h+1} - 1$ nœuds. Pour tout arbre H-équilibré ayant n nœuds et de hauteur h, on a $n \leq 2^{h+1} - 1$, et donc $h + 1 \geq \log_2(n + 1)$.

b) à l'opposé, pour une hauteur donnée $h \geq 1$, les arbres H-équilibrés de hauteur h contenant le moins de nœuds sont ceux pour lesquels la fonction de déséquilibre vaut 1 ou -1 dans tous les nœuds : si l'on enlève une feuille quelconque dans un tel arbre, ou bien l'arbre résultat n'est plus H-équilibré, ou bien sa hauteur a diminué de 1 (cf. figure 6.b et exercices).

Soit $N(h)$ le nombre de nœuds d'un tel arbre ; un arbre H-équilibré de hauteur h ayant le moins de nœuds possibles est formé d'une racine et de deux sous-arbres de même nature, de hauteurs respectives $h - 1$ et $h - 2$; $N(h)$ vérifie donc la relation de récurrence $N(h) = 1 + N(h - 1) + N(h - 2)$, avec les conditions initiales $N(0) = 1$ et $N(1) = 2$. Posons $F(h) = N(h) + 1$, alors l'équation précédente se récrit $F(h) = F(h - 1) + F(h - 2)$.

Cette équation de récurrence de Fibonacci est étudiée au paragraphe 3 de l'annexe (exemple 7) ; avec les conditions initiales $F(0) = 2$ et $F(1) = 3$, elle a pour solution :

$$F(h) = \frac{1}{\sqrt{5}}(\phi^{h+3} - \overline{\phi}^{h+3}), \text{avec } \phi = \frac{1+\sqrt{5}}{2} \text{et } \overline{\phi} = \frac{1-\sqrt{5}}{2}$$

On a donc : $N(h) + 1 = \frac{1}{\sqrt{5}}(\phi^{h+3} - \overline{\phi}^{h+3})$

D'où, pour tout arbre H-équilibré de n nœuds et de hauteur h :

$$n + 1 \geq \frac{1}{\sqrt{5}}(\phi^{h+3} - \overline{\phi}^{h+3})$$

donc $n + 1 > \frac{1}{\sqrt{5}}(\phi^{h+3} - 1)$ car $-1 < \overline{\phi}^{h+3} < 1$

Il en résulte que : $\phi^{h+3} < 1 + \sqrt{5}(1 + n)$

d'où
$$\begin{aligned} h + 3 &< \log_\phi(1 + \sqrt{5}(1 + n)) \\ &< \log_\phi(\sqrt{5}(n + 2)) \\ &= \frac{\log_2(n + 2))}{\log_2 \phi} + \log_\phi \sqrt{5} \\ &< 2 + \frac{\log_2(n + 2)}{\log_2 \phi}, \text{ car } \sqrt{5} < \phi^2 \end{aligned}$$

or $(1/\log_2 \phi) \simeq 1,44$ et donc $h + 1 < 1,44 \log_2(n + 2)$. □

Cette proposition est très importante lorsqu'on utilise les arbres H-équilibrés comme arbres binaires de recherche. On définit un ***arbre AVL*** comme étant un arbre binaire de recherche H-équilibré.

2.2. Adjonction dans les AVL

Les AVL étant des arbres binaires de recherche, on peut utiliser les méthodes vues dans le chapitre précédent pour rechercher, ajouter ou supprimer un élément. D'après la propriété précédente, *la recherche dans un AVL contenant n éléments nécessite toujours $O(\log n)$ comparaisons.* Cependant une adjonction ou une suppression dans un AVL peuvent déséquilibrer l'arbre, comme le montre la figure 8. Ainsi, après avoir ajouté (aux feuilles) ou supprimé un élément dans un AVL, il faut éventuellement le rééquilibrer, tout en conservant la structure d'arbre binaire de recherche.

On étudie maintenant des algorithmes d'adjonction et de suppression dans les AVL, dont la complexité en temps est dans tous les cas en $O(\log n)$, rééquilibrages compris.

L'exemple qui suit va permettre de comprendre les différentes stratégies de rééquilibrage dans les AVL.

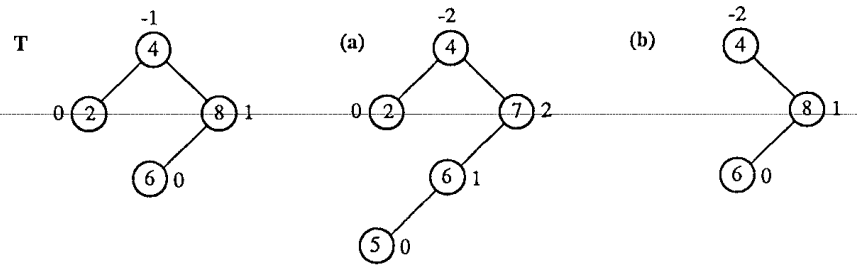


Figure 8. (a) Adjonction de 5 dans l'AVL T, et (b) suppression de 2 dans l'AVL T.

2.2.1. Exemple

Il s'agit de construire un AVL associé à la suite d'éléments 12, 3, 2, 5, 4, 7, 9, 11, 14, 10, en procédant à des adjonctions aux feuilles, et en rééquilibrant l'arbre après chaque adjonction qui l'a déséquilibré. Ainsi, le déséquilibre maximal après une adjonction est $\pm$ 2; de plus, il est clair que l'adjonction d'un élément x ne peut entraîner de déséquilibre que sur les nœuds du chemin de la racine à x. On montre plus loin que lorsqu'il y a déséquilibre après l'adjonction de x, il suffit de rééquilibrer l'arbre à partir d'un seul nœud A (dans le chemin de la racine à x, A est le dernier nœud pour lequel le déséquilibre est de $\pm$ 2).

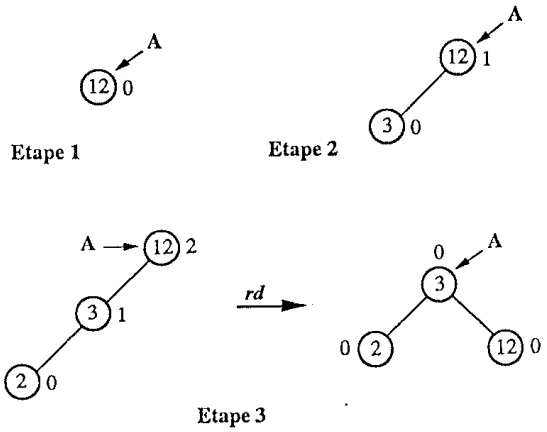


Figure 9

Voyons les différentes étapes de cette construction. Le premier rééquilibrage a lieu à l'étape 3; la transformation appliquée est une rotation droite en A, et l'arbre résultant a un déséquilibre nul en tous ses nœuds (figure 9). L'étape 5 est aussi une rotation droite sur l'arbre de racine 12; après la rotation, les nœuds 5 et 12 ont pour déséquilibre 0, et les autres nœuds conservent le déséquilibre qu'ils avaient avant l'adjonction (figure 10). L'étape 6 est une rotation gauche sur l'arbre de racine 3; le déséquilibre devient 0 pour les nœuds 3 et 5, et reste le même qu'avant l'adjonction pour les autres nœuds (figure 11). L'étape 7 est une rotation double gauche-droite

sur l'arbre de racine 12; le déséquilibre devient 0 pour les nœuds 12 et 7, et pour les autres nœuds reste le même qu'avant adjonction (figure 12).

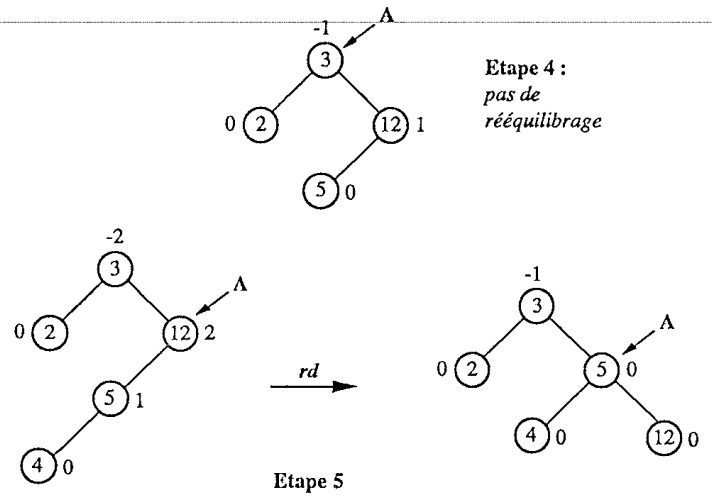


Figure 10

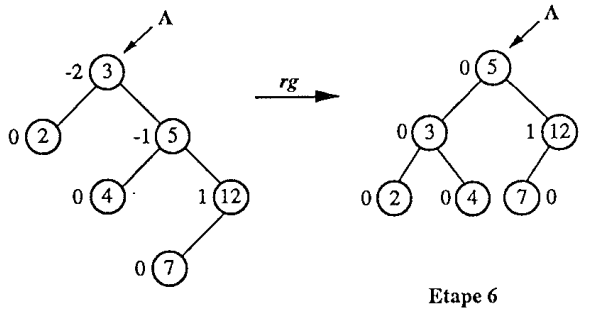


Figure 11

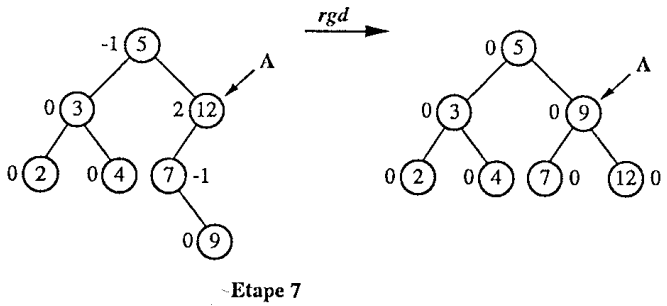
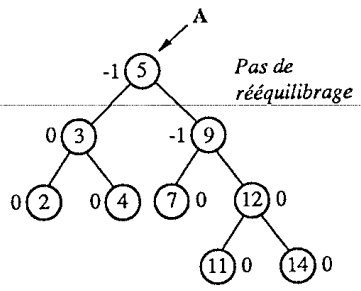
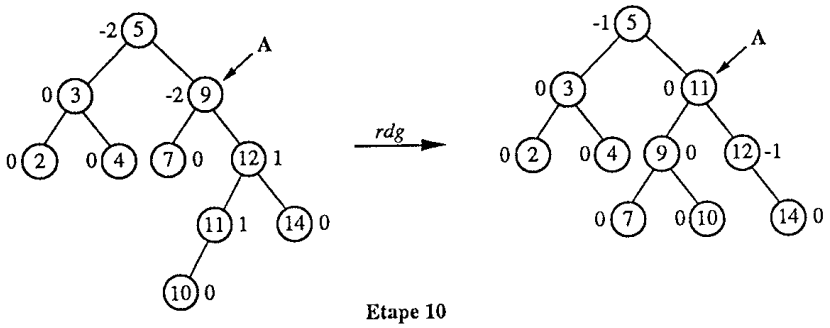


Figure 12



Etapes 8 et 9

Figure 13



Etape 10

Figure 14

Il n'y a pas de rééquilibrage pour les étapes 8 et 9 (figure 13). L'étape 10 est une double rotation droite-gauche sur l'arbre de racine 9; le déséquilibre devient 0 pour les nœuds 9 et 11; il devient -1 pour le nœud 12, et les autres nœuds conservent le même déséquilibre qu'avant l'adjonction (figure 14).

2.2.2. Principe général de rééquilibrage

L'exemple précédent a permis de dégager le principe de rééquilibrage dans les AVL qu'on explicite maintenant.

Soit $T = < r, G, D >$ un AVL; supposons que l'adjonction de l'élément x a lieu sur une feuille de G et *qu'elle fait augmenter de 1 la hauteur de* G, et que G reste un AVL (donc avant l'adjonction, le déséquilibre de G vaut 0).

1) Si le déséquilibre de T valait 0 avant l'adjonction, il vaut 1 après; T reste un AVL et sa hauteur a augmenté de 1.

2) Si le déséquilibre de T valait -1 avant l'adjonction, il vaut 0 après; T reste un AVL et sa hauteur n'est pas modifiée.

3) Si le déséquilibre de T valait +1 avant l'adjonction, il vaut +2 après : T n'est plus H-équilibré, il faut donc le restructurer en AVL.

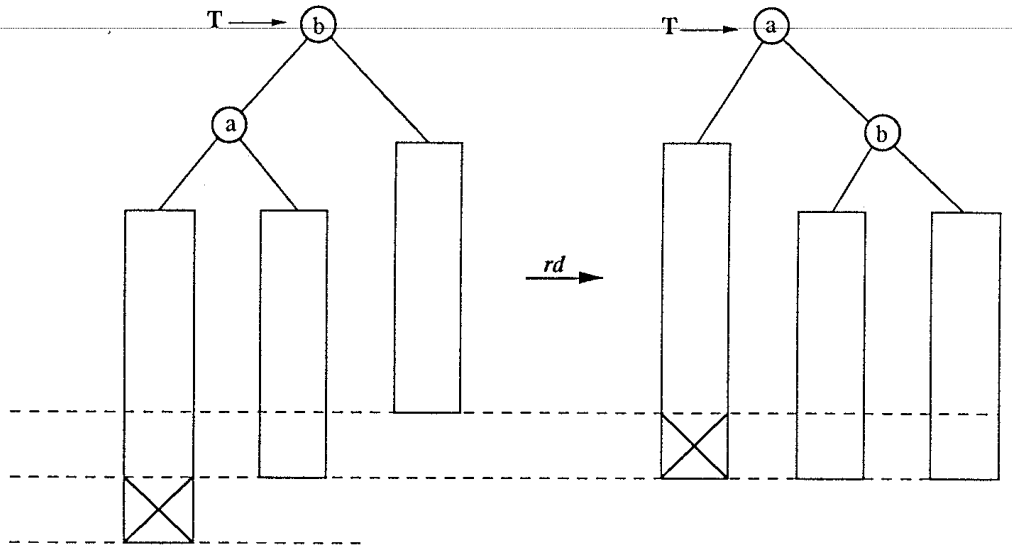


Figure 15.a. Adjonction dans un AVL.

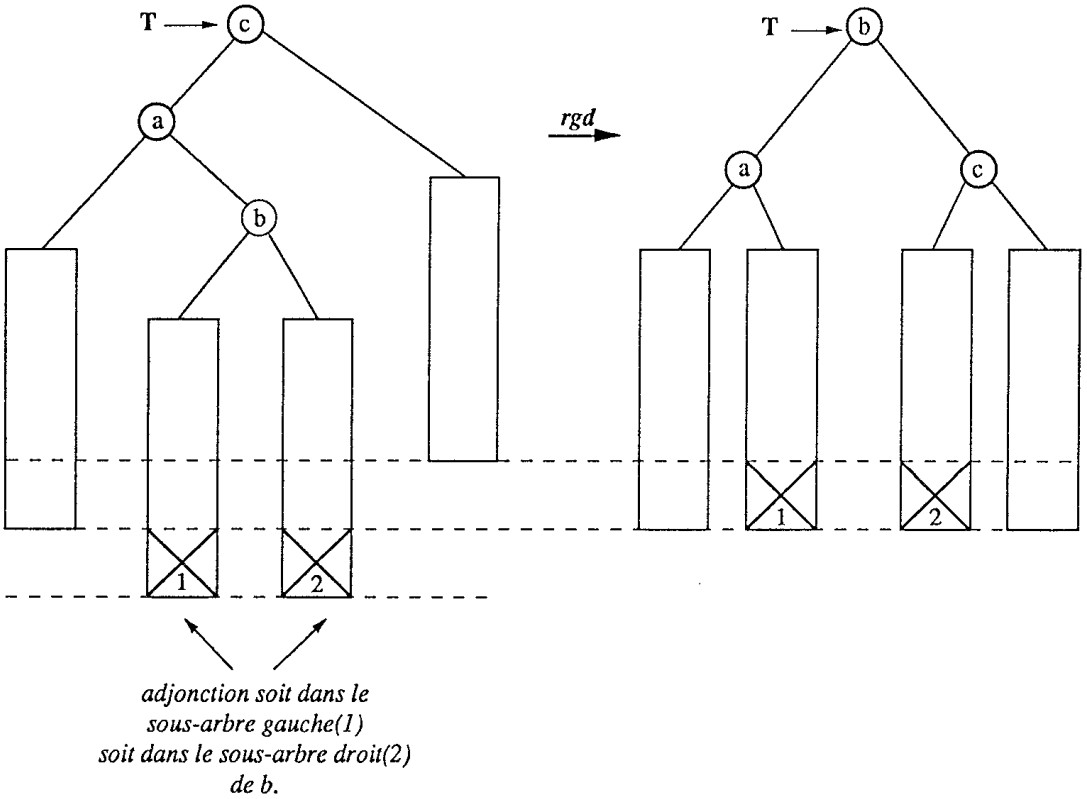


Figure 15.b. Adjonction dans un AVL.

Dans cette troisième hypothèse, il n'y a que deux cas possibles, selon que l'adjonction a lieu dans le sous-arbre gauche ou droit de G. Ces deux cas sont illustrés par la figure 15, où l'on a mis en évidence la hauteur des sous-arbres, et représenté par une croix l'augmentation de hauteur provoquée par l'adjonction. Dans le premier cas (figure 15.a), le déséquilibre de G passe de 0 à 1, et on rééquilibre T par une rotation droite. Dans le deuxième cas (figure 15.b), le déséquilibre de G passe de 0 à -1, et on rééquilibre T par une double rotation gauche-droite. Dans les deux cas, *l'arbre T obtenu après le rééquilibrage est bien un AVL (arbre binaire de recherche H-équilibré), et il retrouve exactement la hauteur qu'il avait avant l'adjonction de x.*

2.2.3. Principe de l'adjonction

Pour effectuer une adjonction dans un AVL, on procède donc comme pour l'adjonction aux feuilles d'un arbre binaire de recherche, mais il faut ensuite rééquilibrer si l'arbre obtenu n'est plus H-équilibré.

Spécification de l'adjonction dans les AVL

L'opération *ajouter-avl* utilise les opérations *rééquilibrer* et *déséquilibre*, et les opérations de rotation rg, rd, rgd, rdg définies précédemment sur les arbres .

ajouter-avl : Elément × Arbre → Arbre
rééquilibrer : Arbre → Arbre
déséquilibre : Arbre → Entier

Pour toutes variables T, G, D de sorte Arbre, et pour toutes variables x et r de sorte Elément, on a les préconditions et les axiomes suivants :

précondition
rééquilibrer(T) **est-défini-ssi** *déséquilibre*$(T) \in \{-2, -1, 0, 1, 2\}$

axiomes
ajouter-avl$(x,$ *arbre-vide*$) = x$
$x \leq r \Rightarrow$ *ajouter-avl*$(x, < r, G, D >)$ = *rééquilibrer*$(< r,$ *ajouter-avl*$(x, G), D >)$
$x > r \Rightarrow$ *ajouter-avl*$(x, < r, G, D >)$ = *rééquilibrer*$(< r, G,$ *ajouter-avl*$(x, D) >)$

déséquilibre$(T) = 0$ (ou 1 ou -1) $\Rightarrow$ *rééquilibrer*$(T) = T$
déséquilibre$(T) = +2$ & *déséquilibre*$(g(T)) = +1 \Rightarrow$ *rééquilibrer*$(T) = rd(T)$
déséquilibre$(T) = -2$ & *déséquilibre*$(d(T)) = -1 \Rightarrow$ *rééquilibrer*$(T) = rg(T)$
déséquilibre$(T) = +2$ & *déséquilibre*$(g(T)) = -1 \Rightarrow$ *rééquilibrer*$(T) = rgd(T)$
déséquilibre$(T) = -2$ & *déséquilibre*$(d(T)) = +1 \Rightarrow$ *rééquilibrer*$(T) = rdg(T)$

Il est important de remarquer que la fonction *rééquilibrer* est utilisée au plus une fois au cours de chaque adjonction.

Proposition : Toute adjonction dans un AVL nécessite au plus une rotation pour le rééquilibrer.

Preuve : Soit à ajouter l'élément x dans l'arbre T; c'est uniquement sur le chemin de la racine de T jusqu'à la feuille où l'on ajoute x qu'il peut y avoir à faire des rotations; on considère sur ce chemin le nœud le plus bas dont le déséquilibre (avant adjonction) est non nul, et l'on note A le sous-arbre enraciné en ce nœud (cf. figure 16).

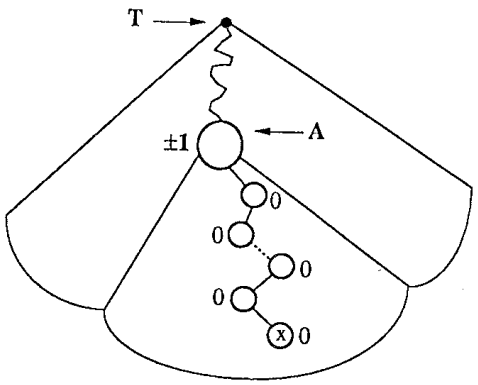


Figure 16

Or, la hauteur du sous-arbre A n'est pas modifiée par l'adjonction de x, même si l'on fait une rotation en A, c'est ce que l'on a vu dans le principe général. Donc dans l'AVL résultant de l'adjonction de x, le père de A, et tous ses ascendants ont exactement le déséquilibre qu'ils avaient avant l'adjonction de x; il n'y a donc jamais besoin d'effectuer de rotation «au-dessus» de A (ni de mise à jour du déséquilibre). □

2.2.4. Algorithme d'adjonction

Pour implanter efficacement l'algorithme d'adjonction, il faut toujours conserver la valeur du déséquilibre en chaque nœud de l'arbre. Dans la procédure *ajouteravl* on suppose que les arbres AVL ont le type suivant :

```
type AVL = ↑NŒUD;
     NŒUD = record
                deseq : -2..+2;
                val : Elément;
                g, d : AVL
            end;
```

Après chaque rotation, il faut mettre à jour la valeur du déséquilibre dans les nœuds modifiés par la rotation. Reprenons les rotations de la figure 15 : dans le premier

cas (figure 15.a), l'adjonction du nouvel élément crée un déséquilibre égal à +2 au nœud contenant b, et +1 au nœud contenant a; après avoir effectué une rotation droite, le déséquilibre du nœud contenant a et celui du nœud contenant b sont tous les deux nuls. Dans le deuxième cas (figure 15.b), le déséquilibre après adjonction vaut +2 au nœud contenant c, -1 au nœud contenant a et +1 (resp. -1) au nœud contenant b, si l'adjonction a lieu dans le sous-arbre gauche (resp. droit) du nœud contenant b; après une rotation gauche-droite, le déséquilibre devient 0 au nœud contenant b , 0 (resp. +1) au nœud contenant a, et -1 (resp. 0) au nœud contenant c.

D'après la propriété précédente, l'algorithme d'adjonction s'exprime de façon itérative : lors de la descente dans l'arbre à la recherche de la place où on doit ajouter x, on mémorise le dernier sous-arbre A pour lequel le déséquilibre est $\pm$ 1. Après avoir ajouté x à la feuille Y, c'est uniquement sur le chemin de A à Y qu'il est nécessaire de modifier les valeurs du déséquilibre. Il faut ensuite faire, le cas échéant, un rééquilibrage en A.

```
procedure, ajouteravl(var T : AVL; x : Elément );
var Y, A, P, AA, PP : AVL;
begin {création du nœud à ajouter}
      new(Y); Y↑.val := x; Y↑. deseq := 0; Y↑.g := nil; Y↑.d := nil;
      if T = nil then T := Y
      else begin
          A := T; AA := nil; P := T; PP := nil;
          {AA est le père de A; PP est le père de P}
          while P <> nil do begin
          {descente à la recherche de la feuille, en mémorisant le dernier nœud
          pointé par A dont le déséquilibre est ± 1}
              if P↑. deseq <> 0 then begin A := P; AA := PP end;
              PP := P;
              if x ≤ P↑.val then P := P↑.g else P := P↑.d
          end;
          {adjonction}
          if x ≤ PP↑.val then PP↑.g := Y else PP↑.d := Y;
          {modification du déséquilibre sur le chemin de A à Y}
          P := A;
          while P <> Y do
                    if x ≤ P↑.val then begin
                      P↑.deseq := P↑.deseq +1;
                      P := P↑.g
                    end
```

```
                else begin
                    P↑.deseq := P↑.deseq −1;
                    P := P↑.d
                end;
        {rééquilibrage}
        case A↑.deseq of
        0, +1, −1 : return;
        +2 :
            case A↑.g↑.deseq of
            +1 : begin RD(A); A↑.deseq := 0; A↑.d↑.deseq := 0 end;
            −1 : begin RGD(A);
                    caseA↑.deseq of
                    +1  : begin A↑.g↑.deseq := 0; A↑.d↑.deseq := −1 end;
                    −1  : begin A↑.g↑.deseq := +1; A↑.d↑.deseq := 0 end;
                     0  : {cas où A=Y}
                       begin A↑.g↑.deseq := 0; A↑.d↑.deseq := 0 end
                    end;
                    A↑.deseq := 0
                end
            end;
        −2 : {cas symétrique du cas +2}...
        end {fin du case};
        {mise à jour des pointeurs après une rotation}
        if AA = nil then T := A
        else if A↑.val≤ AA↑.val then AA↑.g := A else AA↑.d := A
    end {descente}
end ajouteravl;
```

Analyse de la complexité

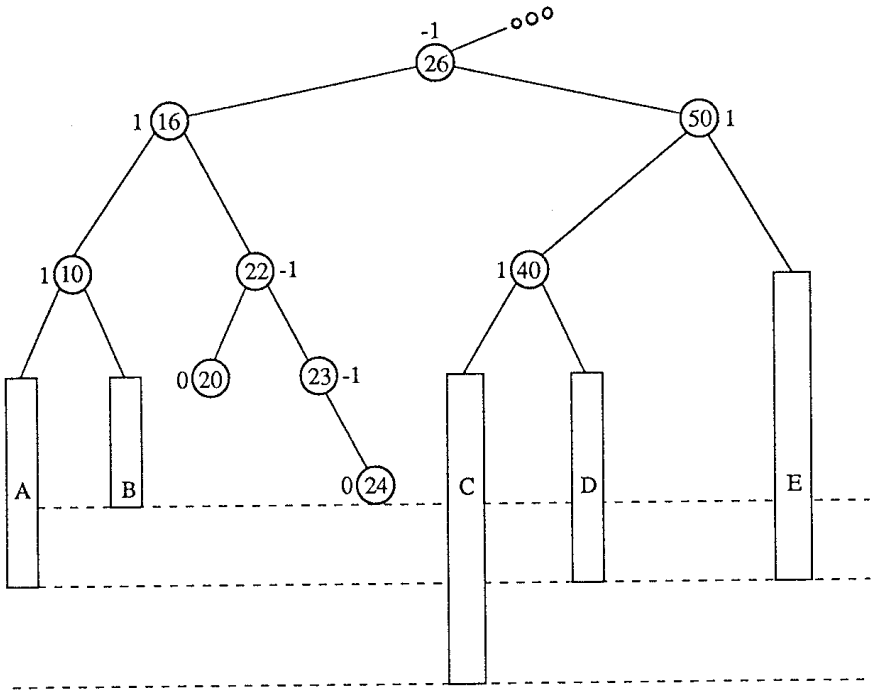
On a vu que la hauteur d'un AVL est toujours inférieure à $1.44 \log_2 n$; il est donc aisé de majorer le nombre de comparaisons nécessaires pour une adjonction, compte tenu du fait qu'il y a au plus une rotation par adjonction, et que chaque rotation n'entraîne qu'un nombre constant de comparaisons. La complexité de l'algorithme d'adjonction est donc en $\Theta(\log n)$ dans le cas le pire.

Qu'en est-il en moyenne? Considérons les AVL construits par adjonctions successives aux feuilles à partir d'une permutation de $[1, n]$, les $n!$ permutations étant équiprobables; on peut alors se poser les problèmes suivants :

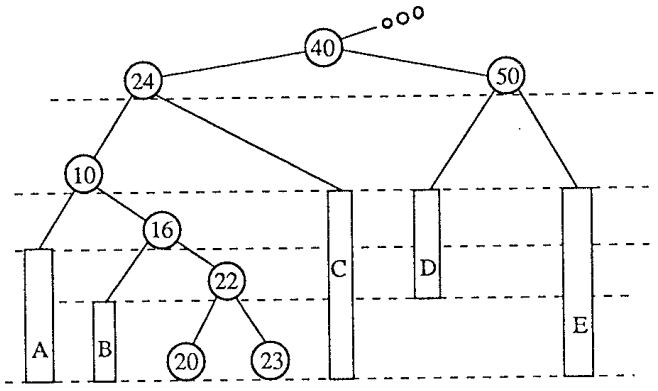
- quelle est la hauteur moyenne d'un tel AVL?
- quelle est la probabilité pour qu'une adjonction entraîne un rééquilibrage?

L'analyse mathématique de ces problèmes est très difficile, et non encore complètement résolue. Des résultats expérimentaux montrent cependant que les AVL ont un très bon comportement : leur hauteur moyenne est $\log_2 n + c$ (où c est une

constante inférieure à 1), ce qui est très proche de la hauteur des arbres complètement équilibrés ; et il faut une rotation (simple ou double) en moyenne toutes les deux adjonctions.



(a) *arbre* AVL



(b) *arbre* AVL *résultant de la suppression de 26 dans l'arbre du (a)*

Figure 17

2.3. Suppression dans les AVL

Le principe de la suppression d'un élément dans un AVL est le même que pour les arbres binaires de recherche : remplacer l'élément à supprimer par l'élément de l'arbre qui lui est immédiatement inférieur. Mais l'arbre résultant d'une telle suppression peut ne plus être H-équilibré et il faut alors le réorganiser en arbre AVL. Le rééquilibrage après suppression fait intervenir les mêmes techniques que le rééquilibrage après adjonction ; mais dans le cas d'une suppression, la réorganisation de l'arbre peut nécessiter plusieurs rotations successives, sur le chemin de la feuille supprimée jusqu'à la racine. Par exemple, pour supprimer l'élément 26 sur l'AVL de la figure 17.a, on le remplace par 24 et on supprime le nœud contenant 24. Cette suppression diminue la hauteur du sous-arbre de racine 22, et le sous-arbre de racine 16 est alors trop déséquilibré. On le réorganise par une rotation droite mais cela accentue le déséquilibre au niveau immédiatement supérieur et il faut alors faire une rotation droite-gauche en 24. Les rotations peuvent ainsi remonter en cascade jusqu'à la racine de l'arbre (cf. figure 17.b).

2.3.1. Spécification de la suppression dans les AVL

L'opération *supprimer-avl* utilise l'opération *rééquilibrer* définie pour *ajouter-avl*, et les opérations *max* et *dlmax* (*max* a déjà été définie pour la suppression dans les arbres binaires de recherche, et *dlmax* est l'analogue de l'opération *dmax* sur les arbres binaires de recherche).

supprimer-avl	: Elément × Arbre → Arbre
rééquilibrer	: Arbre → Arbre
max	: Arbre → Elément
dlmax	: Arbre → Arbre

Pour toutes variables T, G, D de sorte Arbre, et pour toutes variables x et r de sorte Elément, on a les préconditions et les axiomes suivants :

préconditions
max(T) **est-défini-ssi** $T \neq$ *arbre-vide*
dlmax(T) **est-défini-ssi** $T \neq$ *arbre-vide*

axiomes
supprimer-avl$(x,$ *arbre-vide*$)$ = *arbre-vide*
$x = r \Rightarrow$ *supprimer-avl* $(x, < r, G,$*arbre-vide*$>) = G$
$x = r$ & $D \neq$ *arbre-vide*
$\Rightarrow$ *supprimer-avl*$(x, < r,$ *arbre-vide*, $D >) = D$
$x = r$ & $G \neq$ *arbre-vide* & $D \neq$ *arbre-vide*
$\Rightarrow$ *supprimer-avl*$(x, < r, G, D >)$ = *rééquilibrer*$(<$*max*$(G),$ *dlmax*$(G), D >)$
$x < r \Rightarrow$ *supprimer-avl*$(x, < r, G, D >)$ = *rééquilibrer*$(< r,$ *supprimer-avl*$(x, G), D >)$
$x > r \Rightarrow$ *supprimer-avl*$(x, < r, G, D >)$ = *rééquilibrer*$(< r, G,$ *supprimer-avl*$(x, D) >)$
max$(< r, G,$ *arbre-vide*$>) = r$

$dlmax(< r, G,$ *arbre-vide*$>) = G$
$D \neq$ *arbre-vide* $\Rightarrow max(< r, G, D >) = max(D)$
$D \neq$ *arbre-vide* $\Rightarrow dlmax(< r, G, D >) =$ *rééquilibrer* $(< r, G, dlmax(D) >)$

2.3.2. Algorithme de suppression

La réorganisation d'un AVL après une suppression peut entraîner des rotations en cascade sur le chemin allant de la feuille supprimée jusqu'à la racine de l'arbre; en fait les rotations se propagent de bas en haut tant que la hauteur du sous-arbre réorganisé après suppression est diminuée de 1 (c'est toujours le cas après une rotation, mais cela peut aussi être le cas sans qu'il y ait rotation, comme le montre la première étape de la suppression dans l'exemple de la figure 17). L'implémentation de l'algorithme nécessite donc de mémoriser complètement un chemin de la racine à une feuille; cela peut être fait soit de façon explicite en utilisant une pile dans une version proche de celle de l'adjonction, soit de façon implicite dans une version récursive proche de la spécification formelle (attention : après la suppression d'un élément contenu dans le nœud ν, il faut examiner la hauteur h du sous-arbre de racine ν; si h a diminué de 1, il faut éventuellement faire une rotation au niveau du père de ν). La programmation de ces deux versions est laissée en exercice.

Une suppression dans un AVL peut entraîner jusqu'à 1.5 $\log_2 n$ rotations (voir exercice sur les arbres de Fibonacci) mais la complexité reste toujours en $\Theta(\log n)$. Comme pour l'adjonction, l'analyse en moyenne reste un problème ouvert; les résultats expérimentaux montrent cependant qu'il y a seulement en moyenne une rotation pour cinq suppressions, ce qui va donc à l'encontre du sentiment intuitif qu'une suppression est plus coûteuse qu'une adjonction !

3. Arbres-2.3.4

Pour éviter les cas d'arbres de recherche dégénérés, on peut aussi faire varier le nombre de directions de recherche à partir d'un nœud. Dans un arbre binaire de recherche, chaque nœud contient un élément, et la comparaison avec l'élément cherché oriente la suite de la recherche soit dans le sous-arbre gauche, soit dans le sous-arbre droit. Dans les arbres-2.3.4 un nœud peut contenir entre un et trois éléments, rangés en ordre croissant, et le nombre de façons de poursuivre la recherche d'un élément donné, après comparaisons avec le (ou les) élément(s) contenu(s) dans un nœud est donc égal au nombre (2, 3 ou 4) d'intervalles délimités par cet (ou ces) élément(s).

Ce degré de liberté au niveau du nombre de fils des nœuds permet d'imposer une contrainte supplémentaire : toutes les feuilles d'un arbre-2.3.4 sont au même niveau; la hauteur de l'arbre est alors toujours logarithmique en fonction du nombre d'éléments qu'il contient. De plus, on a des algorithmes de rééquilibrage qui maintiennent un arbre-2.3.4 après toute adjonction ou suppression, en effectuant

une suite de rotations sur un chemin de la racine à une feuille. Toutes les opérations sur les arbres-2.3.4 se font en temps logarithmique dans le pire des cas. Enfin, il existe une implémentation efficace des arbres-2.3.4 sous la forme d'arbres binaires de recherche dits bicolores.

3.1. Recherche dans un arbre-2.3.4

3.1.1. Définitions et propriétés

Définition 1 : Un ***arbre de recherche*** est un arbre général étiqueté dont chaque nœud contient un k-uplet d'éléments distincts et ordonnés ($k = 1, 2, 3, ...$). Un nœud contenant les éléments $x_1 < x_2 < ... < x_k$ a $k+1$ sous-arbres, tels que :

- tous les éléments du premier sous-arbre sont inférieurs ou égaux à x_1,
- tous les éléments du $i^{ième}$ sous-arbre ($i = 2, ..., k$) sont strictement supérieurs à x_{i-1} et inférieurs ou égaux à x_i,
- tous les éléments du $(k+1)^{ième}$ sous-arbre sont strictement supérieurs à x_k.

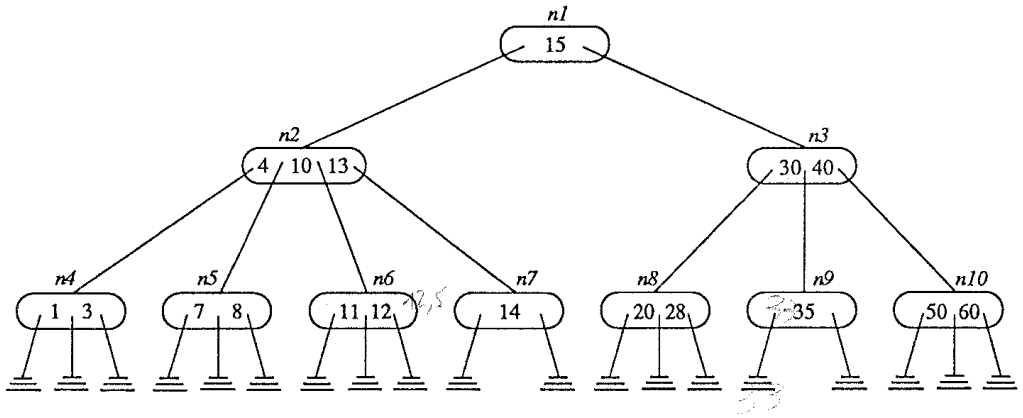


Figure 18. Arbre-2.3.4.

Dans un arbre de recherche, on appelle ***k-nœud*** un nœud contenant $(k-1)$ éléments.

Remarque : La liste L obtenue, dans le parcours en profondeur (procédure *Parc* du chapitre 7) d'un arbre de recherche, par insertion de l'élément $x_i (i = 1, ..., k-1)$ lors de la $(i+1)^{ième}$ visite du k-nœud contenant $< x_1, x_2, ..., x_{k-1} >$, est une liste croissante.

Définition 2 : Un ***arbre-2.3.4*** est un arbre de recherche dont les nœuds sont de trois types, 2-nœud ou 3-nœud ou 4-nœud, et dont toutes les feuilles sont situées au même niveau.

Exemple : La figure 18 montre un exemple d'arbre-2.3.4. Cet arbre contient 18 éléments distincts, et est constitué de 10 nœuds $n_1, n_2, ..., n_{10}$. On a marqué à l'intérieur de chaque nœud les éléments qu'il contient, rangés en ordre croissant, ainsi que les liens vers les sous-arbres. Par exemple le nœud n_2 a quatre sous-arbres : un sous-arbre qui contient tous les éléments inférieurs à 4, puis un sous-arbre qui contient les éléments supérieurs à 4 et inférieurs à 10, puis un sous-arbre qui contient les éléments compris entre 10 et 13, et enfin un sous-arbre qui contient les éléments supérieurs à 13, et inférieurs à 15.

Propriété 2 : La hauteur $h(n)$ d'un arbre-2.3.4 contenant n éléments est en $\Theta(\log\ n)$. Plus précisément, on a l'encadrement suivant :

$$\log_4(n+1) \leq h(n)+1 \leq \log_2(n+1)$$

Preuve : On considère les arbres-2.3.4 extrémaux de hauteur h donnée. L'arbre-2.3.4 qui contient le moins d'éléments est celui qui n'a que des 2-nœuds; puisque toutes les feuilles sont au même niveau, son nombre d'éléments est $2^0 + 2^1 + ... + 2^h = 2^{h+1} - 1$. A l'opposé, l'arbre-2.3.4 qui contient le plus d'éléments est celui qui n'a que des 4-nœuds, et son nombre d'éléments est 3 $(4^0 + 4^1 + ... + 4^h) = 4^{h+1} - 1$. On en déduit donc que pour tout arbre-2.3.4 contenant n éléments et de hauteur $h(n)$, on a la relation $2^{h(n)+1} - 1 \leq n \leq 4^{h(n)+1} - 1$, d'où l'on tire l'encadrement de la hauteur. □

3.1.2. Principe de la recherche dans un arbre-2.3.4

La recherche d'un élément x dans un arbre-2.3.4 M suit le même principe que la recherche dans un arbre binaire de recherche.

On compare x avec le(s) élément(s) $x_1, ..., x_i$ (i =1 ou 2 ou 3) contenus dans la racine de M ;

- s'il existe $j \in [1, i]$ tel que $x = x_j$ alors x est trouvé,
- si $x \leq x_1$, la recherche de x se poursuit dans le premier sous-arbre de M,
- si $x_j < x \leq x_{j+1}$ (pour j= 1, ..., $i-1$) la recherche de x se poursuit dans le $(j+1)^{ième}$ sous-arbre de M,
- si $x > x_i$, la recherche de x se poursuit dans le dernier sous-arbre de M,
- si la recherche se termine sur une feuille qui ne contient pas x, alors x n'appartient pas à M.

La recherche d'un élément dans un arbre-2.3.4 suit un chemin dans l'arbre, à partir de la racine vers une feuille. La complexité de l'algorithme de recherche, comptée en nombre de comparaisons entre éléments, est donc en $\Theta(\log\ n)$ dans tous les cas, d'après la propriété 2.

Dans un arbre-2.3.4, on étudie l'adjonction aux feuilles. L'adjonction d'un nouvel élément ne pose problème que si la feuille qui doit recevoir cet élément contient déjà trois éléments. Dans ce cas, il faut ajouter un nouveau nœud à l'arbre, et réorganiser en arbre-2.3.4.

3.2. Adjonction dans un arbre-2.3.4

3.2.1. Adjonction avec éclatements en remontée

L'exemple qui suit montre la construction d'un arbre-2.3.4 par adjonctions successives aux feuilles, à partir de l'arbre vide. On insère successivement les éléments : 4, 35, 10, 13, 3, 30, 15, 12, 7, 40, 20, 11, 6.

L'adjonction de 4 entraîne la création d'un 2-nœud, qui se transforme en 3-nœud avec l'adjonction de 35, puis en 4-nœud avec l'adjonction de 10 (figure 19).

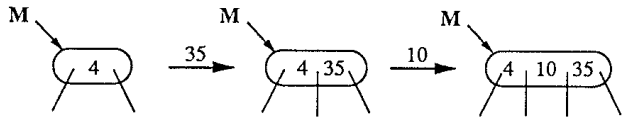


Figure 19

A cette étape, l'unique nœud de M ne peut plus contenir de nouveaux éléments. Or, du point de vue de la recherche, M est équivalent à l'arbre binaire de la figure 20.

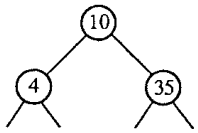


Figure 20

Cet arbre est bien un arbre-2.3.4, et il y a de la place dans ses feuilles pour de nouveaux éléments. On peut alors ajouter 13, puis 3, puis 30 (figure 21).

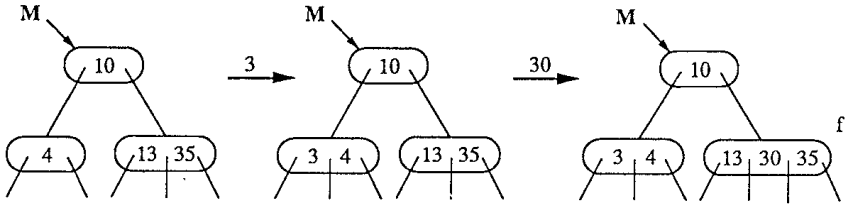


Figure 21

L'adjonction de 15 provoque l'*éclatement* de la feuille f en deux 2-nœuds contenant respectivement le plus petit et le plus grand élément de f, et l'élément médian 30

doit être ajouté au nœud père de f ; il y a alors de la place pour 15 dans le même nœud que 13 qui devient alors un 3-nœud (figure 22).

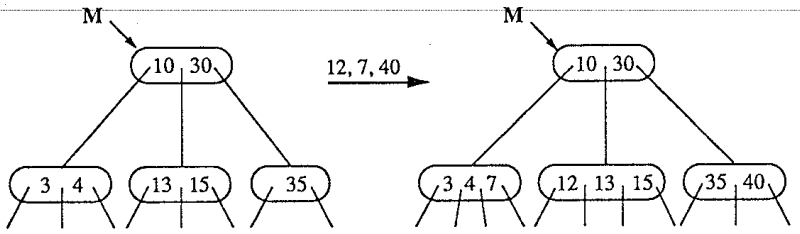


Figure 22

L'adjonction des éléments 12, 7 et 40 peut être facilement réalisée, mais l'adjonction de 20 entraîne un *éclatement* en deux de la feuille contenant 12, 13 et 15, avec *remontée* de 13 dans le nœud père (figure 23).

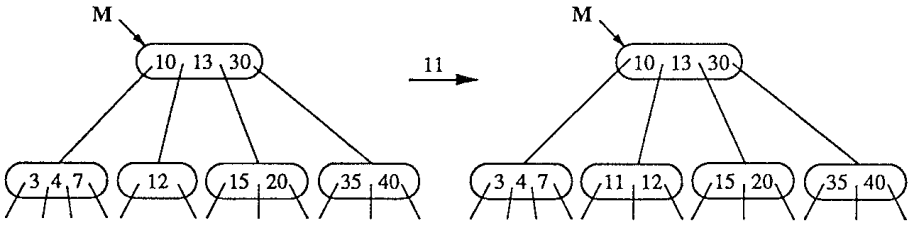


Figure 23

On peut alors ajouter 11 sans problème. L'adjonction de 6 provoque l'*éclatement* de la feuille (3,4,7), dont le père contient déjà trois éléments ; la *remontée* de 4 fait donc *éclater à son tour* la racine de l'arbre en deux nœuds et il faut aussi créer un nouveau nœud racine pour recevoir l'élément médian 13 (figure 24).

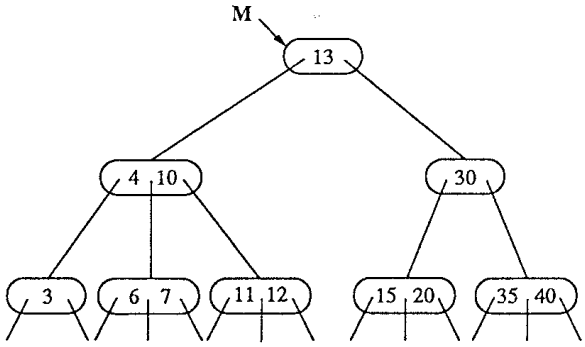


Figure 24

3.2.2. Adjonction avec éclatements à la descente

Avec la méthode d'adjonction précédente, les éclatements peuvent *remonter en cascade*, éventuellement sur toute la hauteur de l'arbre si le chemin suivi pour déterminer l'endroit de l'adjonction n'est formé que de 4-nœuds.

Pour éviter la propagation des éclatements de bas en haut, il suffit de travailler sur des arbres-2.3.4 qui ne contiennent jamais deux 4-nœuds à la suite : dans ce cas *toute adjonction provoque au plus un éclatement.* Ceci peut être réalisé en éclatant les 4-nœuds *à la descente* : lors de la recherche de la place de l'élément à ajouter, on parcourt un chemin dans l'arbre-2.3.4, à partir de la racine jusqu'à une feuille; seuls les 4-nœuds de ce chemin risquent d'éclater à la suite de l'adjonction; on prévient ce risque en les faisant éclater, au fur et à mesure de leur rencontre, *avant* de réaliser l'adjonction. Cette précaution a évidemment l'inconvénient de provoquer quelquefois des éclatements qui se révèlent inutiles. Reprenons l'exemple précédent en utilisant la méthode d'éclatement à la descente : l'arbre-2.3.4 obtenu après adjonctions successives de 4, 35, 10, 13, 3, 30, 15, 12, 7, 40 et 20 est le même qu'avec la première méthode mais lors de la recherche de la place où on doit insérer 11, on rencontre un 4-nœud (la racine) que l'on doit faire éclater à ce moment là, ce qui donne l'arbre de la figure 25.

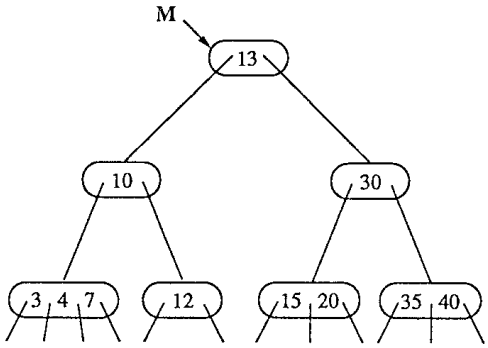


Figure 25

L'adjonction de 11 se réalise ensuite sans problème dans le nœud qui contient 12.

A cette étape, l'arbre obtenu (figure 26) n'est pas le même que celui que l'on obtient avec la méthode d'éclatements en remontée; il a une hauteur plus grande. Mais si, par exemple, on ajoute maintenant 6, il faut éclater le nœud (3,4,7) et l'on obtient finalement le même arbre que celui de la figure 24.

Dans cette méthode, tous les rééquilibrages se font pendant la descente dans l'arbre, lors de la recherche de la feuille où doit être inséré l'élément; lorsqu'on arrive sur cette feuille, il ne reste plus qu'à y ajouter effectivement l'élément.

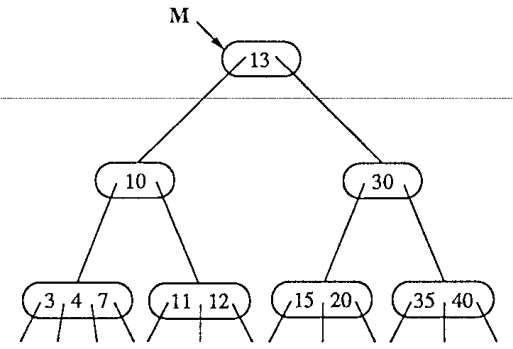
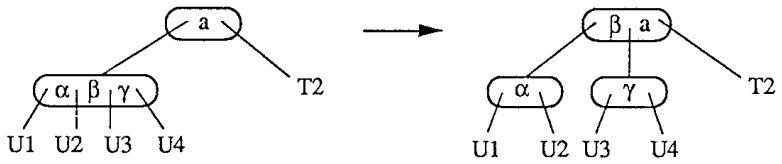
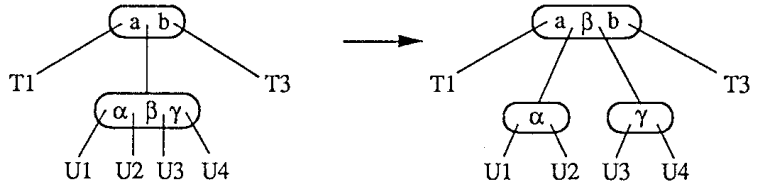


Figure 26

Les transformations de rééquilibrage à la descente sont purement locales, comme le montre la figure 27. Lorsqu'un nœud éclate, son père n'est pas un 4-nœud (sinon on aurait dû le faire éclater juste avant). L'éclatement d'un nœud consiste donc à transformer son père soit en 3-nœud, si c'est un 2-nœud, soit en 4-nœud, si c'est un 3-nœud, ce qui est réalisé en un nombre constant d'opérations. Cette transformation locale ne modifie pas la profondeur des feuilles, sauf si c'est la racine qui éclate, et dans ce cas la hauteur de l'arbre augmente d'une unité. Ainsi, la procédure d'adjonction conserve bien les propriétés d'arbre-2.3.4.



(i) *cas où le père du noeud à éclater est un 2-noeud*



(ii) *cas où le père du noeud à éclater est un 3-noeud*

Figure 27. Eclatements à la descente.

Remarque 1 : Les deux méthodes d'adjonction (éclatements en remontée ou à la descente) opèrent toujours sur un chemin de la racine à une feuille de l'arbre-2.3.4. Elles ont donc toutes les deux une complexité qui est dans tous les cas en $\Theta(\log n)$, où n est le nombre d'éléments contenus dans l'arbre-2.3.4.

La seconde méthode présente cependant plusieurs avantages pratiques par rapport à la méthode d'adjonction avec éclatements en remontée :

- on n'a jamais besoin de retraverser le chemin de recherche pour restaurer l'équilibre de l'arbre, donc l'algorithme se décrit de façon itérative sans pile ;
- l'arbre-2.3.4 est accessible en parallèle par plusieurs utilisateurs avec une synchronisation minimale, chaque mise à jour n'affectant qu'une petite portion de l'arbre.

Par contre, elle a l'inconvénient de prendre plus de place : le taux d'occupation des nœuds étant plus faible qu'avec la première méthode, il faut donc plus de nœuds, et par conséquent la hauteur de l'arbre est plus grande.

Remarque 2 : Si on voulait insérer plusieurs fois le même élément, il faudrait modifier à la fois la définition des arbres-2.3.4 (un nœud pourrait contenir plusieurs fois le même élément) et les algorithmes de recherche et d'adjonction (l'éclatement d'un nœud pourrait répartir des éléments égaux dans deux fils différents de ce nœud). On traite ici le cas où *tous les éléments sont différents.*

3.2.3. Spécification de l'adjonction avec éclatements à la descente

Dans la spécification de l'algorithme d'adjonction avec éclatements à la descente, on suppose qu'un arbre-2.3.4 est défini par la donnée de la suite des éléments $e_1 < ... < e_i$ (i = 1, 2 ou 3) contenus dans la racine, et de la liste des $i + 1$ sous-arbres de la racine. La spécification que l'on donne ici est incomplète. On a plutôt essayé d'expliquer le fonctionnement de l'algorithme, et d'illustrer tous les cas différents par des exemples. (Le lecteur rebuté par la longueur de cette spécification peut passer directement au paragraphe 4.)

sorte A234
utilise Elément, Booléen
opérations

arbre-vide	: → A234
< _, _, _ >	: Elément × A234 × A234 → A234
< (_, _), _, _, _ >	: Elément × Elément × A234 × A234 × A234 → A234
< (_, _, _), _, _, _, _ >	: Elément × Elément × Elément × A234 × A234 × A234 × A234 → A234
est-vide	: A234 → Booléen
2-fils, *3-fils*, *4-fils*	: A234 → Booléen
F_1, F_2, F_3, F_4	: A234 → A234
E_1, E_2, E_3	: A234 → Elément

L'opération *est-vide* rend vrai si l'arbre est vide et faux sinon.

Les opérations *2-fils*, *3-fils*, *4-fils* permettent de distinguer les arbres-2.3.4 selon que la racine est un 2-nœud, un 3-nœud ou un 4-nœud.

Par exemple, 3-fils(T) = *vrai si* T *est de la forme* $< (a,b), T_1, T_2, T_3 >$ *avec* T_1, T_2 *et* T_3 *non vides, et faux sinon.*

Les opérations F_1, F_2, F_3, F_4 permettent d'accéder aux sous-arbres de la racine, et l'on prend la convention que si la racine d'un arbre n'a que i sous-arbres alors F_j a pour résultat l'arbre vide, noté aussi Ø, pour $j > i$.

Par exemple, si T *est de la forme* $< (a,b), T_1, T_2, T_3 >$, *on a* $F_2(T) = T_2$ *et* $F_4(T) = \emptyset$.

E_1, E_2 et E_3 sont des opérations partielles qui permettent d'accéder aux éléments. $E_1(T)$ est défini pour $T \neq \emptyset$, $E_2(T)$ est défini si, et seulement si, *3-fils*(T) est vrai, et $E_3(T)$ est défini si, et seulement si, *4-fils*(T) est vrai.

Par exemple, si T *est de la forme* $< (a,b), T_1, T_2, T_3 >, E_2(T) = b$, *et* $E_3(T)$ *n'est pas défini.*

Dans l'algorithme d'adjonction, le cas où la racine est un 4-nœud est traité à part car l'éclatement introduit alors une nouvelle racine. On a donc besoin de deux opérations, *ajouter234* et *ajouter*.

ajouter234 : A234 × Elément → A234
ajouter : A234 × Elément → A234

L'opération *ajouter* est définie uniquement pour les arbres-2.3.4 dont la racine est un 2-nœud ou un 3-nœud. On a les axiomes suivants :

ajouter234$(\emptyset, x) = < x, \emptyset, \emptyset >$
4-fils(T) = vrai ⇒ *ajouter234*$(<(a,b,c), T_1, T_2, T_3, T_4 >, x)$
$= ajouter\ (< b, < a, T_1, T_2 >,\ < c, T_3, T_4 >>, x)$,
avec $T = < (a,b,c), T_1, T_2, T_3, T_4 >$
4-fils(T) = faux ⇒ *ajouter234*(T, x) = *ajouter*(T, x).

Dans le cas où la racine d'un arbre-2.3.4 T est un 2-nœud, si le sous-arbre où l'on doit faire l'adjonction a quatre fils, on l'éclate (figure 27-i), et on continue l'adjonction dans le «bon» sous-arbre résultant de cet éclatement. L'axiome ci-dessous décrit ce qui se passe quand on ajoute un élément dans le *premier sous-arbre* résultant de l'éclatement de $F_1(T)$:

2-fils(T) = vrai & $x < E_1(T)$ & *4-fils*$(F_1(T))$ = vrai & $x < E_2(F_1(T))$
⇒ *ajouter*$(T, x) = < (\beta, a)$, *ajouter*$(< \alpha, U_1, U_2 >, x), < \gamma, U_3, U_4 >, T_2 >$,
avec $T = < a, T_1, T_2 >$ et $T_1 = < (\alpha, \beta, \gamma), U_1, U_2, U_3, U_4 >$.

On a un axiome similaire dans le cas où l'adjonction doit se faire dans le deuxième sous-arbre résultant de l'éclatement (cas où $x > E_2(F_1(T))$). Et si c'est dans $F_2(T)$ que l'adjonction a lieu (cas où $x > E_1(T)$), les axiomes s'obtiennent de façon analogue.

Lorsque le nœud où doit se faire l'adjonction n'est pas un 4-nœud, les axiomes sont similaires à ceux de l'adjonction dans les arbres binaires de recherche, par exemple :

$$\textit{2-fils}(T) = \text{vrai} \;\&\; x < E_1(T) \;\&\; \textit{4-fils}(F_1(T)) = \text{faux} \;\&\; F_1(T) \neq \textit{arbre-vide}$$
$$\Rightarrow \textit{ajouter}(T, x) = < E_1(T), \textit{ajouter}(F_1(T), x), F_2(T) >.$$

De même, dans le cas où la racine de T est un 3-nœud, si le sous-arbre où doit avoir lieu l'adjonction a quatre fils, on l'éclate. L'axiome ci-dessous décrit le cas où $F_2(T)$ est éclaté, et où l'adjonction se fait dans le deuxième sous-arbre résultant de l'éclatement. (On a un axiome analogue lorsque l'adjonction se fait dans le premier sous-arbre.)

$$\textit{3-fils}(T) = \text{vrai} \;\&\; E_1(T) < x < E_2(T) \;\&\; \textit{4-fils}(F_2(T)) = \text{vrai} \;\&\; x > E_2(F_2(T))$$
$$\Rightarrow \textit{ajouter}(T, x) =< (a, \beta, b), T_1, < \alpha, U_1, U_2 >, \textit{ajouter}(< \gamma, U_2, U_3 >, x), T_3 >,$$

avec $T = < (a, b), T_1, T_2, T_3 >$ et $T_2 = < (\alpha, \beta, \gamma), U_1, U_2, U_3, U_4 >$.

Les cas d'éclatement de $F_1(T)$ ou de $F_3(T)$ se traitent de façon similaire.

Lorsque le sous-arbre où se fait l'adjonction n'est pas un *4-fils*, il n'y a pas d'éclatement; par exemple

$$\textit{3-fils}(T) = \text{vrai} \;\&\; E_1(T) < x < E_2(T) \;\&\; \textit{4-fils}(F_2(T)) = \text{faux} \;\&\; F_2(T) \neq \textit{arbre-vide}$$
$$\Rightarrow \textit{ajouter}(T, x) = < (E_1(T), E_2(T)), F_1(T), \textit{ajouter}(F_2(T), x), F_3(T) >$$

L'axiome suivant décrit le cas de l'adjonction dans un arbre vide :

$$\textit{ajouter234}(\emptyset, x) =< x, \emptyset, \emptyset >$$

Lorsqu'on ajoute à une feuille on a des axiomes particuliers, dont on donne quelques exemples :

$$x < a \Rightarrow \textit{ajouter}(< a, \emptyset, \emptyset >, x) =< (x, a), \emptyset, \emptyset, \emptyset >$$
$$x < a \Rightarrow \textit{ajouter}(< (a, b), \emptyset, \emptyset, \emptyset >, x) = < (x, a, b), \emptyset, \emptyset, \emptyset, \emptyset >$$

(Pour une feuille qui est un 4-nœud, le traitement est le même que dans le cas général.)

3.2.4. Représentation des arbres-2.3.4

Quand on passe à l'implantation de l'algorithme, le problème qu'on rencontre immédiatement est celui de la *représentation des différents types de nœuds.*

On peut choisir une représentation maximale : tous les nœuds sont des enregistrements à sept champs : trois champs «élément» e_i et quatre champs «pointeur» P_j (figure 28); mais cette représentation gaspille beaucoup de place pour les nœuds qui ne sont pas des 4-nœuds.

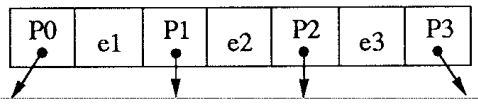


Figure 28

On peut choisir de représenter les différents types de nœuds par différents types de données (figure 29); cependant, dans ce cas, la manipulation des nœuds lors des réorganisations de l'arbre non seulement rend fastidieuse la programmation de l'algorithme, mais de plus ralentit son exécution (ce n'est d'ailleurs pas possible de façon efficace dans un langage de programmation comme Pascal).

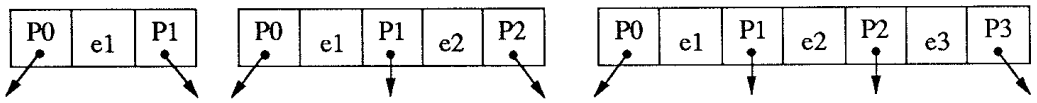


Figure 29

On choisit ici de représenter les arbres-2.3.4 sous forme d'arbres binaires bicolores. Cela évite les inconvénients précédents. La représentation des nœuds est uniforme, ce sont des enregistrements à quatre champs : un élément, deux pointeurs, et un bit de couleur; et les transformations de rééquilibrage se réduisent à des rotations simples ou doubles, à gauche ou à droite (cf. le premier paragraphe de ce chapitre).

3.3. Arbres bicolores

3.3.1. Définition et propriétés

Les ***arbres bicolores***, inventés par Guibas et Sedgewick dans les années soixante-dix, sont des arbres binaires de recherche dont les nœuds portent une information supplémentaire : les nœuds sont soit rouges soit blancs. Les arbres bicolores sont une représentation des arbres-2.3.4.

Dans un arbre-2.3.4, on hiérarchise les éléments à l'intérieur d'un 4-nœud ou d'un 3-nœud et l'on considère alors les arbres-2.3.4 comme des arbres binaires dont les nœuds sont eux-mêmes des petits morceaux d'arbres binaires. Pour simuler cette double hiérarchie, on colorie les liens de deux façons différentes : *en rouge* (double trait) si les éléments appartiennent au même nœud de l'arbre-2.3.4 – on parle alors *d'éléments jumeaux* (sur la figure 30, les éléments a, b et c sont jumeaux) –, et *en blanc* (simple trait) si les éléments sont dans des nœuds différents de l'arbre-2.3.4 initial. En fait, la couleur peut être attachée aux nœuds plutôt qu'aux liens : tout nœud a la couleur du lien qui pointe sur lui, et un nœud est donc rouge si, et seulement si, l'élément qu'il contient est un jumeau de l'élément contenu dans son père.

Les différentes étapes de la transformation d'un arbre-2.3.4 en arbre bicolore apparaissent sur la figure 30 pour un 4-nœud, et sur la figure 31 pour un 3-nœud (il y a dans ce cas deux représentations possibles – penché à droite ou à gauche –, et les deux peuvent coexister à la suite de rotations, comme on le verra plus loin).

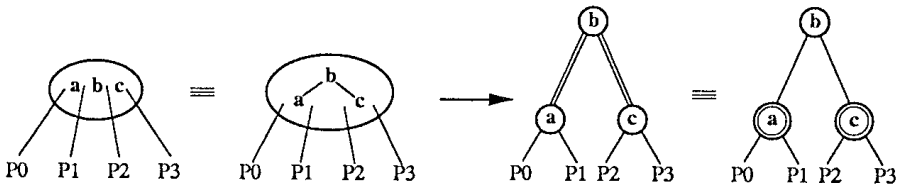
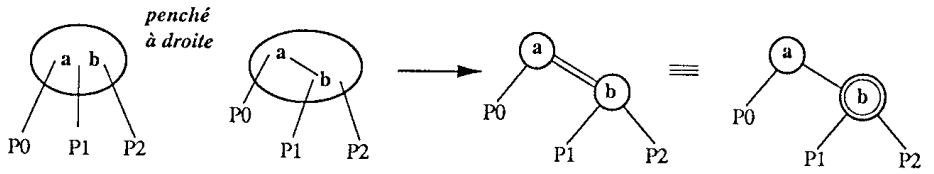
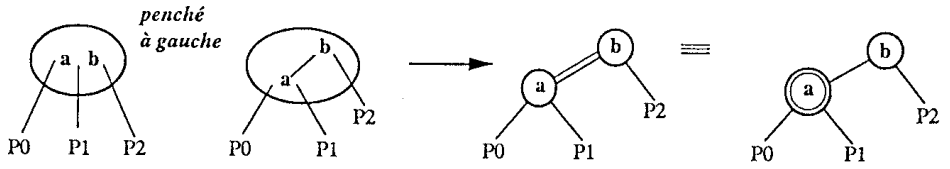


Figure 30. Transformation d'un 4-nœud.



i) *représentation d'un 3-noeud "penché à droite"*



ii) *représentation d'un 3-noeud "penché à gauche"*

Figure 31. Transformation d'un 3-nœud.

Par exemple, l'arbre-2.3.4 $M1$ de la figure 32(a) peut être représenté par l'arbre bicolore $M2$ de la figure 32(b) : en compactant les nœuds jumeaux de $M2$, on retrouve $M1$.

La hauteur de l'arbre bicolore obtenu par un tel procédé de construction est au plus égale à deux fois la hauteur de l'arbre-2.3.4 initial, augmentée de 1. En effet, par construction, il n'y a jamais deux liens rouges à la suite sur un chemin de l'arbre bicolore, et de plus, tous les chemins ont le même nombre de liens blancs, qui est égal à la hauteur $h_{2.3.4}(n)$ de l'arbre-2.3.4 initial. Or, d'après la Propriété 2 on a la relation : $\log_4(n+1) \leq h_{2.3.4}(n) + 1 \leq \log_2(n+1)$. On peut donc énoncer la propriété suivante.

Propriété 3 : Tout arbre bicolore associé à un arbre-2.3.4 contenant n éléments a une hauteur d'ordre $\Theta(\log n)$.

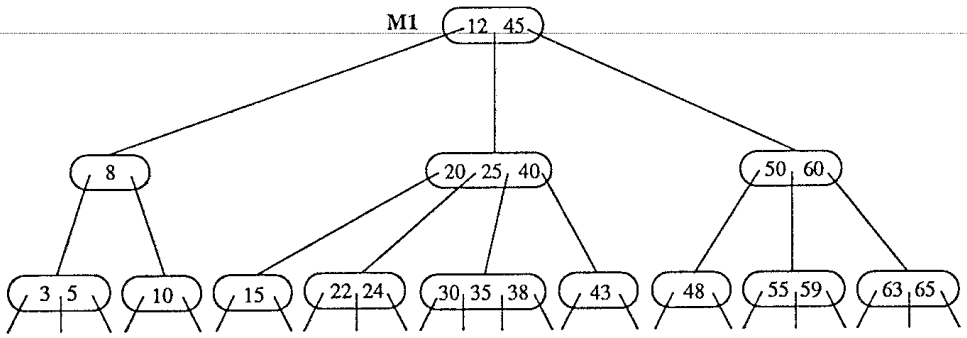


Figure 32(a). Arbre-2.3.4.

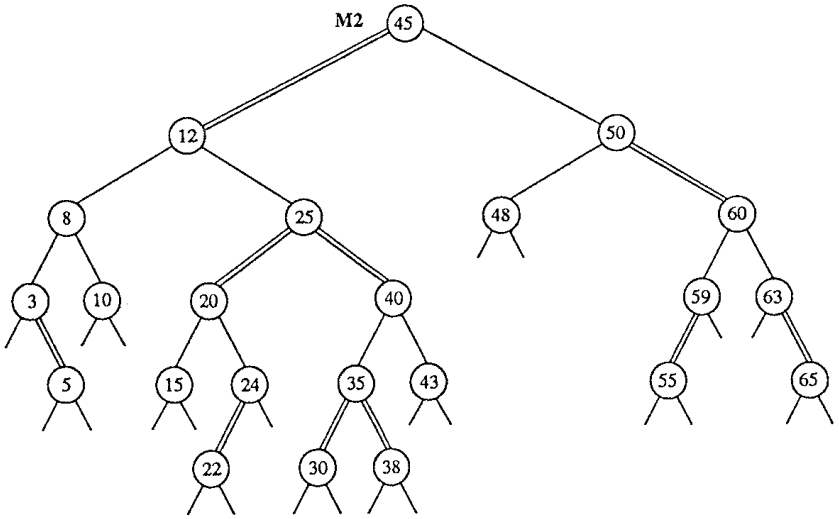


Figure 32 (b). Une représentation en arbre bicolore.

La recherche d'un élément dans un arbre bicolore se fait exactement de la même façon que dans un arbre binaire de recherche, sans tenir aucun compte des couleurs. Elle nécessite donc un nombre de comparaisons qui est en $\Theta(\log n)$ *dans le pire des cas*.

3.3.2. Adjonction dans un arbre bicolore

L'adjonction d'un élément peut être réalisée en simulant l'adjonction avec éclatements à la descente dans un arbre-2.3.4 : éclater un 4-nœud revient à inverser les couleurs des éléments jumeaux de ce nœud (figure 33).

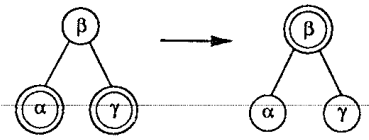


Figure 33

Cependant, il se peut que cette transformation fasse apparaître deux nœuds rouges consécutifs dans l'arbre bicolore; il faut éviter ce phénomène si l'on veut conserver la propriété de hauteur logarithmique des arbres bicolores, et pour cela on a recours à des transformations locales qui ont la forme des rotations présentées au début de ce chapitre.

Considérons les différentes configurations qui peuvent se présenter :

1) Le 4-nœud qui doit éclater est attaché à un 2-nœud (comme premier ou second fils); alors une simple inversion des couleurs est suffisante (figure 34).

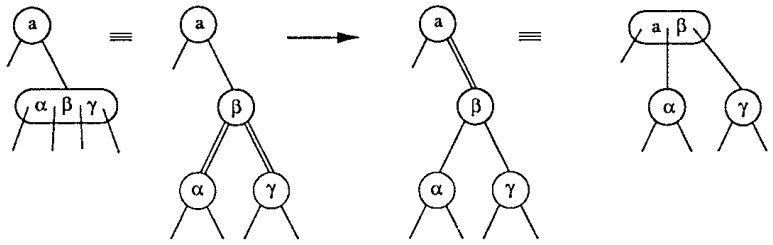


Figure 34

2) Le 4-nœud E qui doit éclater est rattaché à un 3-nœud; alors il faut distinguer plusieurs cas – on considère ici les trois cas possibles lorsque le 3-nœud est représenté «penché à droite» (figure 31-i), mais il y a trois cas symétriques lorsque le 3-nœud est représenté «penché à gauche».

a) Si E est premier fils du 3-nœud, il suffit encore d'inverser les couleurs des éléments contenus dans le 4-nœud (figure 35).

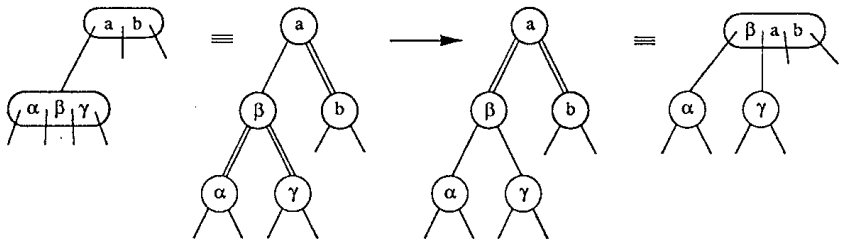


Figure 35

b) Si E est second fils du 3-nœud, l'inversion des couleurs entraîne une mauvaise disposition des éléments jumeaux a, β et b, mais pour retrouver la bonne représentation (figure 30), il suffit alors d'effectuer une rotation droite-gauche au niveau du nœud contenant a (figure 36).

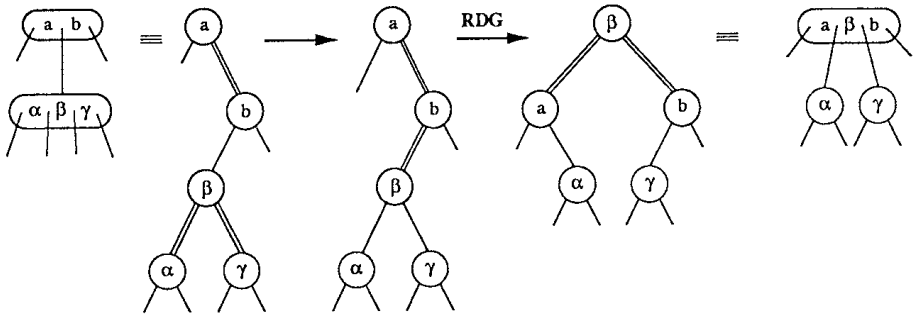


Figure 36

c) Si E est troisième fils du 3-nœud, l'inversion des couleurs entraîne encore une mauvaise disposition des éléments jumeaux a, b et β ; dans ce cas, il faut faire une rotation gauche pour rétablir l'arbre bicolore (figure 37).

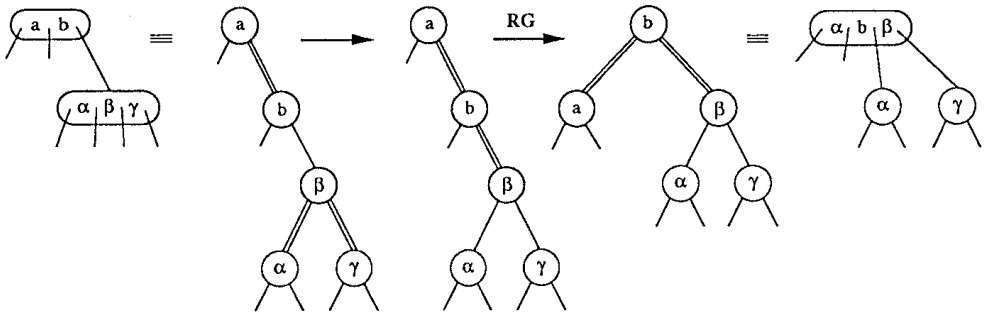


Figure 37

Algorithme d'adjonction

L'algorithme d'adjonction d'un élément dans un arbre bicolore se décrit donc informellement de la façon suivante.

• Descendre à partir de la racine à la recherche de la feuille où il faut insérer l'élément.

• Sur ce chemin, lorsqu'un nœud A a ses deux fils rouges, on inverse les couleurs de A et de ses fils; si de plus le père de A est lui aussi rouge, il faut faire une rotation au niveau du grand-père de A avant de poursuivre la descente.

• L'adjonction d'un élément se fait toujours dans une feuille qui devient un nœud rouge, puisque le nouvel élément est ajouté en tant que jumeau; dans le cas où le père du nœud ajouté est rouge, mais n'a pas de frère rouge, il faut effectuer une rotation comme en b).

Par exemple, pour ajouter 20 à l'arbre bicolore de la figure 38(a) (qui résulte de l'adjonction successive des éléments 4, 35, 10, 13, 3, 30, 15, 12, 7 et 40) on suit le chemin 10 → 30 → 13; les deux fils de 13 sont rouges, on inverse donc les couleurs (13 devient rouge et ses deux fils blancs) mais alors 13 et 30 étant rouges, il faut faire une rotation droite-gauche au niveau de 10. Puis on continue à descendre dans la branche droite de 13 : 30 → 15 et on peut alors insérer 20 à droite de 15 (figure 38(b)). Si l'on ajoute ensuite 21, dont la place est à droite de 20, il faudra faire une rotation gauche au niveau de 15.

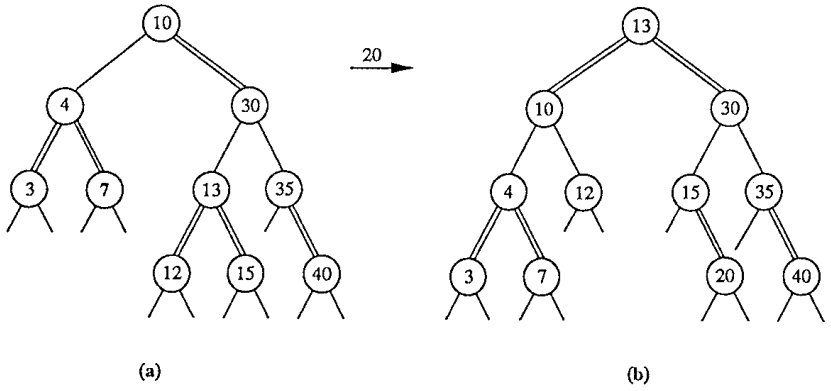


Figure 38. Adjonction dans un arbre bicolore.

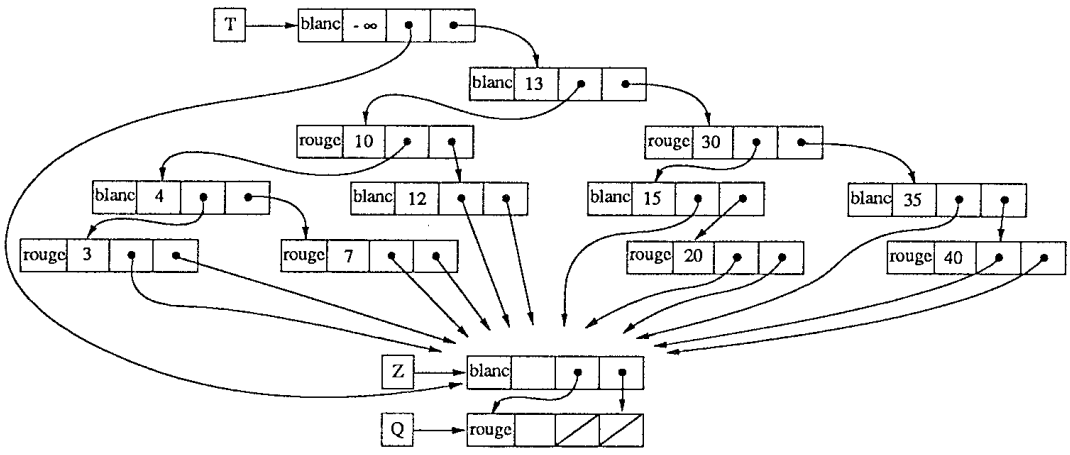


Figure 39. Représentation de l'arbre de la figure 38(b).

On va maintenant décrire une procédure de recherche dans un arbre bicolore, avec adjonction dans le cas où l'élément cherché n'est pas dans l'arbre. Un arbre bicolore est représenté par le type Pascal suivant :

```
type couleur = (blanc, rouge);
     Bic = ↑Bnœud;
     Bnœud = record
                 coul : couleur;
                 val : Elément;
                 g, d : Bic
             end;
```

Pour unifier le traitement, la procédure rech-adj-bic travaille sur un arbre bicolore avec une tête T, une sentinelle Z dont la couleur est le blanc, et un quadruplet supplémentaire pointé par Q, sous la sentinelle (figure 39).

Le quadruplet de la tête T contient dans son champ *val* un élément strictement inférieur à tous les éléments possibles; sa couleur est le blanc, son lien g pointe sur Z et son lien d pointe sur la racine de l'arbre, ou sur Z si l'arbre est vide. Son utilité est de ne pas particulariser l'adjonction dans un arbre vide.

L'utilisation de la sentinelle Z est classique : on fait pointer sur Z tous les liens de l'arbre qui sont à nil, et, de plus, avant de commencer la recherche d'un élément X, on affecte X au champ *val* de la sentinelle; de cette façon, la recherche aboutit toujours sur X et l'on n'a pas à distinguer deux cas de sortie de l'itération. De plus, on a introduit un quadruplet pointé par Q qui est rouge, et dont les liens g et d sont à nil, pour pouvoir traiter de la même façon les inversions de couleurs qui ont lieu au milieu de l'arbre et celles qui ont lieu en bas (adjonction).

```
procedure rech-adj-bic(X: Elément; Z, Q : Bic; var T, A: Bic; var B: Boolean);
{Cette procédure recherche l'élément X dans l'arbre bicolore T (avec tête et
sentinelle, et dont tous les éléments sont distincts) et, s'il n'y est pas, réalise
l'adjonction de X dans T ; elle renvoie pour résultat dans B la valeur true s'il y a
eu adjonction et false sinon, et dans tous les cas elle donne l'adresse A de X dans
l'arbre bicolore T ; Z est la sentinelle, sa couleur est le blanc et ses liens g et d
valent Q ; Q est rouge}

var P, GP, GGP : Bic;{on a besoin de trois ascendants, père, grand-père
et arrière-grand-père, pour le rééquilibrage}
begin
  A := T; Z↑.val := X; B := false; P := T; GP := T;
  repeat
  GGP := GP; GP := P; P := A;
    if X < A↑.val then A := A↑.g else A := A↑.d;
    if (A↑.g↑.coul = rouge) and (A↑.d↑.coul = rouge) then begin
      if A = Z then {adjonction}
```

```
        begin B := true; new(A);
        with A↑ do begin val := X; g := Z; d := Z; coul := rouge end;
        if X < P↑.val then P↑.g := A else P↑.d := A
        end
        else begin
        A↑.coul := rouge; A↑.g↑.coul := blanc; A↑.d↑.coul := blanc
        end;
      if P↑.coul = rouge then begin {rééquilibrage}
        {8 types de rotation-raccrochage}
      if P↑.val > GP↑.val then begin {rotation gauche ou droite-gauche}
        if A↑.val > P↑.val then RG(GP) else RDG(GP);
        if GP↑.val < GGP↑.val then GGP↑.g := GP else GGP↑.d := GP
        end
      else begin { rotation droite ou gauche- droite}
        if A↑.val < P↑.val then RD(GP) else RGD(GP);
        if GP↑.val < GGP↑.val then GGP↑.g := GP else GGP↑.d := GP
        end;
      {rétablir les couleurs après rotation}
      GP↑.coul := blanc; GP↑.g↑.coul := rouge; GP↑.d↑.coul := rouge;
      {rétablir la hiérarchie des ascendants}
        P := GP; GP := GGP;
      if X = P↑.val then A := P {A renvoie l'adresse de X dans T}
      else if X < P↑.val then A := P↑.g else A := P↑.d
      end
    end
  until X = A↑.val;
  T↑.d↑.coul := blanc {la racine de l'arbre est toujours blanche}
end rech-adj-bic;
```

Analyse de la complexité

Comme pour les AVL, les principaux critères d'évaluation de la complexité des arbres bicolores sont d'une part la forme «équilibrée» de l'arbre, ce qui donne une mesure du coût de recherche d'un élément, et d'autre part le taux de rotations-inversions de couleurs, qui mesure le coût d'une adjonction.

Puisque la hauteur d'un arbre bicolore contenant n éléments est toujours en $\Theta(\log_2 n)$, et que le nombre de rotations-inversions de couleurs est toujours inférieur à la hauteur (exercices), le coût d'une recherche ou d'une adjonction est donc toujours en $\Theta(\log n)$. Pour ce qui est du comportement moyen de ces coûts, aucune analyse complète n'a pu être réalisée jusqu'à présent. Cependant, les résultats expérimentaux montrent des performances analogues à celles des AVL : les arbres bicolores ont une hauteur moyenne égale à $\log_2 n + c$, où c est une constante positive inférieure à 1, et il y a en moyenne une rotation ou une inversion de couleurs pour deux descentes dans l'arbre.