Chapitre 6

Arbres et ensembles ordonnés

Ce long chapitre contient la description de plusieurs familles d'arbres binaires de recherche équilibrés. Nous commençons par les arbres AVL, puis nous décrivons les arbres a–b, avec notamment des opérations plus élaborées comme la concaténation et la scission, et le coût amorti d'une suite d'opérations. Ensuite, nous présentons les arbres bicolores, et enfin une réalisation de structures persistantes sur ces arbres.

Introduction

La manipulation de grands volumes d'informations est une tâche fréquente en algorithmique. Ce chapitre présente plusieurs structures de données pour leur gestion. Les données peuvent être très variées, et organisées différemment d'une application à l'autre. Nous supposons que chaque donnée comporte un «champ» particulier, appelé sa *clé* qui peut être, par exemple, une chaîne de caractères ou un numéro. Les clés permettent d'identifier les données, et des données différentes ont donc des clés distinctes. Une clé permet d'accéder à la donnée qu'elle représente, selon un mécanisme qui dépend de la situation considérée, et que nous ne considérons pas ici. Les clés elles-mêmes, et c'est là leur intérêt, sont prises dans un ensemble totalement ordonné (appelé l'«univers»). La manipulation des données se fait donc à travers la manipulation des clés.

On peut distinguer trois catégories de structures de données, en fonction de leur comportement dans le temps. Une structure *statique* représente un ensemble qui ne varie pas (ou très peu) dans le temps. Un exemple typique est un lexique fixe. D'autres exemples sont fournis en géométrie algorithmique; dans le problème de localisation, il s'agit de déterminer dans quelle région d'une subdivision fixe du plan se trouve un point donné. L'action principale sur une telle structure est la recherche d'informations, les mises à jour étant exceptionnelles et de ce fait négligeables. Un soin particulier (et donc un certain temps) peut être consacré à

la construction d'une structure «optimale». Même si l'ensemble des données est statique, la structure de données peut évoluer dans le temps, pour améliorer le temps de réponse en fonction de la fréquence des interrogations.

Les structures *dynamiques*, qui font l'objet de ce chapitre, permettent en plus des modifications de l'ensemble des données représentées. Il s'agit principalement d'implémenter des *dictionnaires*, donc de réaliser la recherche, l'insertion et la suppression d'informations, ou des *listes concaténables* qui exigent comme opérations supplémentaires la concaténation et la scission. Les arbres binaires de recherche équilibrés réalisent ces opérations de manière efficace.

Ces structures sont éphémères dans le sens où une mise à jour détruit de façon définitive la version précédente. Une structure est *persistante* si elle est dynamique, et si elle conserve également les *versions antérieures* de la structure. On peut distinguer deux degrés de persistance : si l'on dispose de la *liste* des versions précédentes, on peut rechercher la présence d'éléments dans une version antérieure ; si de plus on est autorisé à modifier les versions antérieures, on aboutit à un *arbre* de versions, la plus élaborée des structures persistantes. Nous allons présenter une réalisation efficace de la liste des versions.

Dans ce chapitre, nous proposons des solutions au problème que voici : étant donné un ensemble S, sous-ensemble de l'univers noté U, effectuer de manière efficace les opérations suivantes, qui caractérisent un dictionnaire : rechercher un élément dans S, insérer un élément dans S, supprimer un élément dans S. Nous examinerons aussi d'autres opérations, comme la scission et la concaténation.

Eliminons tout de suite le cas d'un univers qui serait «petit». On peut alors implémenter ces opérations de façon très simple. Un ensemble S est représenté par un tableau indicé par les éléments de U, qui donne sa fonction caractéristique, c'est-à-dire qui indique, pour chaque élément de U, s'il appartient ou non à S. Nous supposons dans la suite que l'univers U est très grand, trop grand en tout cas pour permettre cette réalisation, et que S est petit par rapport à U.

Il apparaît que les arbres sont une structure de données particulièrement bien adaptée à la réalisation efficace des trois opérations de base (recherche, insertion, suppression). Lorsque l'arbre est «équilibré» selon des critères à préciser, on peut éviter qu'il dégénère en une structure trop proche d'une liste, et on peut alors effectuer les opérations en temps logarithmique en fonction du nombre d'éléments de l'ensemble S. Bien évidemment, l'insertion et la suppression peuvent déséquilibrer l'arbre. Une partie substantielle des algorithmes est consacrée aux opérations de rééquilibrage qui doivent être réalisées avec soin.

Une première espèce d'arbres binaires équilibrés est constituée des arbres *AVL*, nommée ainsi d'après les initiales de leurs deux inventeurs (Adelson-Velskii et Landis). Nous considérons ensuite les arbres a–b qui sont des arbres où chaque nœud a entre a et b fils. Nous présentons la version balisée (dans le sens défini ci-dessous) de ces arbres, et montrons comment les utiliser pour la concaténation et la scission. Les arbres bicolores, que nous introduisons ensuite, sont une autre

famille d'arbres équilibrés. Ils ont été conçus à l'origine en vue de l'implémentation des arbres a–b, mais leur importance dépasse ce cadre. Ils nous serviront notamment pour la représentation de structures persistantes, présentées dans la dernière section.

6.1 Arbres de recherche

6.1.1 Définition

Nous allons parler d'arbres de recherche, et principalement d'arbres binaires de recherche. Il s'agit d'arbres binaires étiquetés aux sommets. L'étiquette d'un sommet x est la *clé* ou le *contenu* du sommet, et notée $c(x)$. Si A est un arbre binaire non vide, on note A_g et A_d ses sous-arbres gauche et droit. Pour tout sommet x, on note $A(x)$ le sous-arbre de racine x, et $A_g(x)$, $A_d(x)$ les sous-arbres de $A(x)$. Un *arbre binaire de recherche* est un arbre muni d'une fonction clé c sur l'ensemble de ses sommets vérifiant

$$c(y) < c(x) < c(z)$$

pour tout sommet x, et pour tous sommets y de $A_g(x)$ et z de $A_d(x)$. Il revient au même de dire que les clés sont croissantes si on les liste dans l'ordre symétrique.

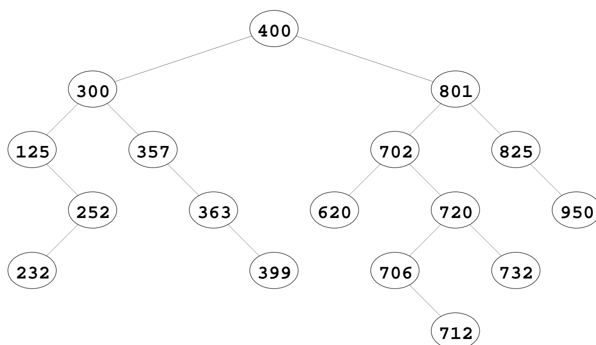

Figure 1.1: *Un arbre binaire de recherche.*

Il est commode de disposer de la même notion pour les arborescences ordonnées (un arbre binaire est un arbre positionné au sens du chapitre 4). Dans une *arborescence ordonnée de recherche*, chaque nœud x, à $d(x)$ fils, est muni de $d(x)-1$ clés $c_1(x), \ldots, c_{d(x)-1}(x)$, vérifiant la propriété suivante : soient $A_1(x), \ldots, A_{d(x)}(x)$ les sous-arbres de x, et soit y_i un sommet de $A_i(x)$ pour $1 \leq i \leq d(x)$; alors

$$c(y_1) < c_1(x) < c(y_2) < \cdots < c_{d(x)-1}(x) < c(y_{d(x)}).$$

Nous regroupons les notions d'arbre binaire de recherche et d'arborescence ordonnée de recherche sous le nom d'*arbre de recherche*. On rencontre en fait deux

catégories d'arbres de recherche, ceux où l'information est rangée aux sommets, et ceux où elle n'est rangée qu'aux feuilles. Comme nous l'avons dit plus haut, chaque clé est représentative d'une «donnée» en général plus volumineuse et plus lourde à manipuler que la clé elle-même. Chaque clé permet un accès, direct ou indirect, à cette information, et cet accès peut être rangé, avec la clé correspondante, à tout sommet de l'arbre, ou aux feuilles seulement. Dans le deuxième cas, les nœuds aussi contiennent des éléments de l'ensemble U des clés, mais elles ne servent qu'à organiser l'arbre. Nous les appelons des *balises*. Elles permettent de naviguer dans l'arbre, et en particulier elles pilotent la descente dans l'arbre lors d'une recherche. Chaque nœud contient une balise qui est supérieure ou égale aux clés de son sous-arbre gauche, et inférieure aux clés de son sous-arbre droit (en d'autres termes, les clés *et* les balises sont croissantes en parcours symétrique). Les arbres a–b sont des arbres balisés.

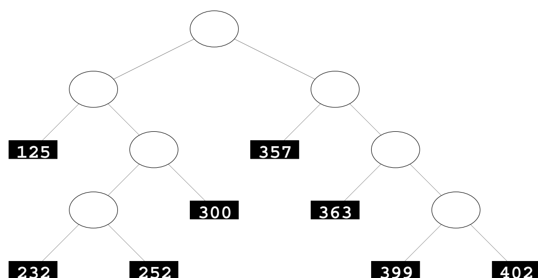

Figure 1.2: *Un arbre sans balises.*

La différence de structure entre un arbre binaire de recherche et un arbre binaire balisé de recherche, même si elle peut être importante dans certaines applications, est assez faible. Ainsi, pour passer d'un arbre binaire balisé à un arbre binaire de recherche, on peut procéder en deux étapes comme suit : dans un premier temps, on *rebalise* l'arbre, en choisissant comme balise d'un nœud la plus grande clé de son sous-arbre gauche.

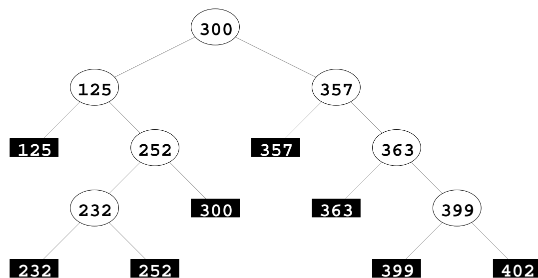

Figure 1.3: *L'arbre avec balises.*

Ensuite, on «oublie» les feuilles, et on insère dans l'arbre obtenu la plus grande clé de l'arbre d'origine. Cet algorithme s'implémente assez facilement en temps linéaire avec une file (voir exercices).

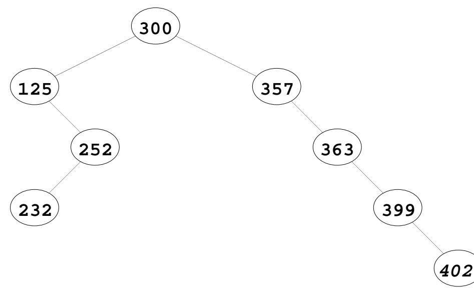

Figure 1.4: *L'arbre après insertion de la dernière clé.*

L'opération réciproque, de passage d'un arbre binaire de recherche à un arbre balisé, peut se faire en ajoutant d'abord des feuilles pour compléter l'arbre (au sens du chapitre 4), puis en munissant ces feuilles des clés appropriées : une feuille reçoit la clé du nœud pour lequel elle est la feuille la plus à droite dans le sous-arbre gauche. Il n'est pas difficile de réaliser cette opération avec une pile :

```
procédure BALISER(a);
   si EST-FEUILLE(a) alors
      FIXERCLÉ(a, SOMMET(P));
      DÉPILER(P)
   sinon
      EMPILER(CLÉ(a), P);
      BALISER(G(a));
      BALISER(D(a))
   finsi.
```

Les opérations de dictionnaire se réalisent, sur un arbre binaire balisé, de manière tout à fait semblable à celle exposée pour les arbres binaires de recherche dans le chapitre 3.

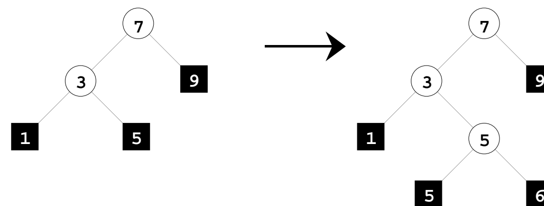

Figure 1.5: *Insertion de* 6 *dans l'arbre balisé.*

Ainsi, l'*insertion* d'une clé se fait comme dans un arbre binaire de recherche, sauf que l'on remplace la feuille découverte en descendant dans l'arbre par un nœud

qui devient le père de cette feuille et d'une nouvelle feuille contenant la clé (voir figure 1.5).

La *suppression* d'une clé est plus simple que dans un arbre binaire de recherche ordinaire; elle se fait en supprimant la feuille contenant la clé. Le père de cette feuille est remplacé par son autre fils (voir figure 1.6).

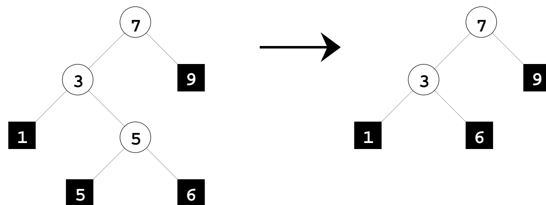

Figure 1.6: *Suppression de* 5 *dans l'arbre balisé.*

6.1.2 Rotations

Les opérations de rotation et de double rotation que nous introduisons maintenant sont intéressantes pour tous les arbres binaires. Elles servent notamment à rééquilibrer les arbres après une insertion ou une suppression. Soit A un arbre binaire non vide. Il est commode d'écrire $A = (x, B, C)$ pour exprimer que x est la racine de A et que B et C sont ses sous-arbres gauche et droit (voir figure 1.7). Nous utilisons cette notation dans la suite de cette section.

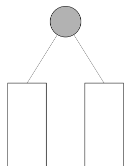

Figure 1.7: *L'arbre* $A = (x, B, C)$.

Soit donc $A = (x, X, B)$ un arbre non vide, supposons B non vide et posons $B = (y, Y, Z)$. La *rotation gauche* de A est l'opération :

$$A = (x, X, (y, Y, Z)) \quad \mapsto \quad \mathcal{G}(A) = (y, (x, X, Y), Z)$$

représentée dans la figure 1.8. La *rotation droite* est l'opération inverse :

$$A = (y, (x, X, Y), Z) \quad \mapsto \quad \mathcal{D}(A) = (x, X, (y, Y, Z))$$

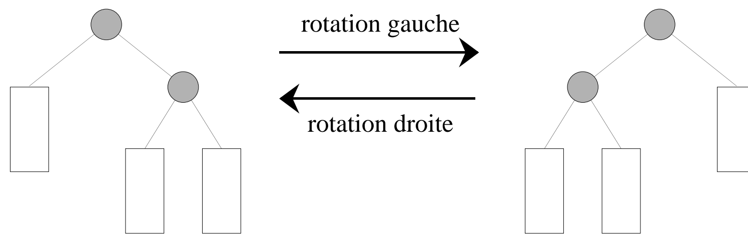

Figure 1.8: *Rotations gauche et droite.*

Proposition 1.1. *Si A est un arbre binaire de recherche, et si la rotation gauche (respectivement droite) est définie sur A, alors $\mathcal{G}(A)$ (respectivement $\mathcal{D}(A)$) est encore un arbre binaire de recherche.* ■

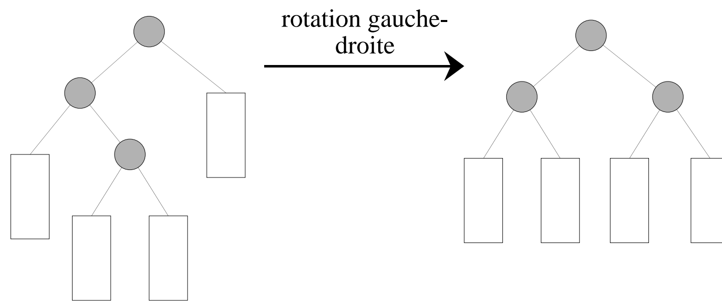

Figure 1.9: *Rotation gauche-droite.*

Deux *doubles rotations* sont utilisées, la rotation *gauche-droite* et la rotation *droite-gauche* : elles associent, à un arbre $A = (x, A_g, A_d)$ donné, respectivement l'arbre $\mathcal{D}(x, \mathcal{G}(A_g), A_d)$ et l'arbre $\mathcal{G}(x, A_g, \mathcal{D}(A_d))$ (voir figures 1.9 et 1.10). Ces opérations ne sont évidemment définies que si les sous-arbres requis ne sont pas vides. Ces opérations préservent bien sûr également les arbres binaires de recherche.

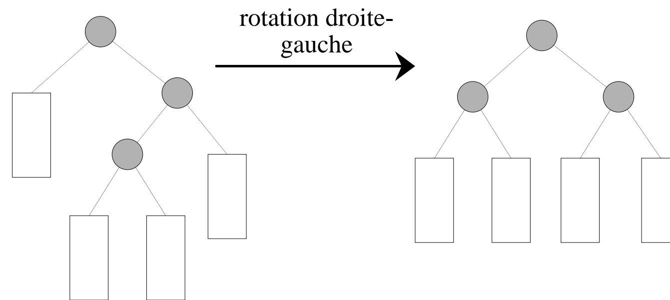

Figure 1.10: *Rotation droite-gauche.*

Un aspect essentiel de ces opérations est qu'elles s'implémentent en temps constant. Lorsque les arbres binaires sont déclarés par

```
arbre = ^sommet;
sommet = RECORD
            val: element;
            g, d: arbre
         END;
```

on obtient la procédure suivante qui réalise la rotation gauche, par un simple échange de pointeurs :

```
PROCEDURE rotationgauche (VAR a: arbre);
   VAR
      b: arbre;
   BEGIN
      b := a^.d; a^.d := b^.g; b^.g := a; a := b
   END;
```

Proposition 1.2. *Les opérations de rotation et de double rotation sur les arbres binaires se réalisent en temps constant.* ∎

6.2 Arbres *AVL*

6.2.1 Définition

Les arbres *AVL*, introduits par Adelson-Velskii et Landis en 1962, constituent une famille d'arbres binaires de recherche équilibrés en hauteur.

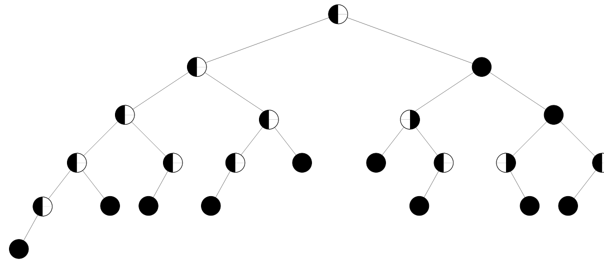

Figure 2.1: *Un arbre AVL.*

Un arbre binaire A est un *arbre AVL* si, pour tout sommet de l'arbre, les hauteurs des sous-arbres gauche et droit diffèrent d'au plus 1. Plus précisément, posons $\delta(A) = 0$ si A est l'arbre vide, et sinon

$$\delta(A) = hauteur(A_g) - hauteur(A_d)$$

où A_g et A_d sont les sous-arbres gauche et droit de A. Le nombre $\delta(A)$ est appelé l'*équilibre* de A (ou de sa racine). Alors A est un arbre *AVL* si pour tout sommet

x, on a $\delta(A(x)) \in \{-1, 0, 1\}$. Par abus, on écrira aussi $\delta(x)$. En particulier, un arbre réduit à un seul sommet a un équilibre nul.

Les arbres *AVL* ont une hauteur logarithmique en fonction du nombre de sommets. Plus précisément :

Proposition 2.1. *Soit A un arbre AVL ayant n sommets et de hauteur h. Alors*

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

Preuve. Pour une hauteur h donnée, l'arbre ayant le plus de sommets est l'arbre complet qui a $2^{h+1}-1$ sommets. Donc $n \leq 2^{h+1}-1$, et par conséquent $\log(1+n) \leq 1+h$. Pour l'autre inégalité, soit $N(h)$ le minimum des nombres de sommets des arbres *AVL* de hauteur h. On a $N(-1) = 0$, $N(0) = 1$, $N(1) = 2$, et

$$N(h) = 1 + N(h-1) + N(h-2), \qquad h \geq 2$$

car pour obtenir le minimum, on choisit l'un des sous-arbres minimum de hauteur $h-1$, et l'autre minimum de hauteur $h-2$. Si l'on pose $F(h) = N(h)+1$, on a $F(0) = 2$, $F(1) = 3$, et

$$F(h) = F(h-1) + F(h-2), \qquad h \geq 2$$

donc $N(h) = F_{h+3}$, où F_n est le n-ième nombre de Fibonacci (voir chapitre 2). Pour tout arbre *AVL* à n sommets et de hauteur h, on a par conséquent

$$n+1 \geq F(h) = \frac{1}{\sqrt{5}}\left(\phi^{h+3} - \bar{\phi}^{h+3}\right) > \frac{1}{\sqrt{5}}\phi^{h+3} - 1$$

d'où

$$h+3 < \frac{\log_2(n+2)}{\log_2 \phi} + \log_\phi \sqrt{5} \leq 1,44 \log_2(n+2) + 2$$

parce que $1/\log_2 \phi \leq 1,44$ et $\log_\phi \sqrt{5} \leq 2$. ∎

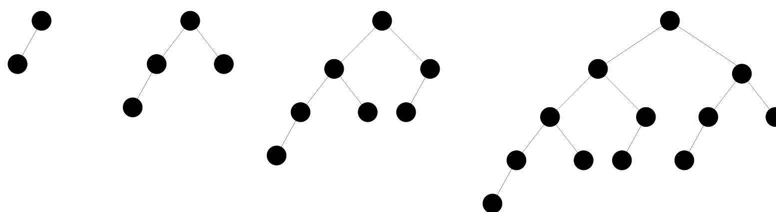

Figure 2.2: *Arbres de Fibonacci* Φ_k *pour* $k = 2, 3, 4, 5$.

La borne de la proposition est essentiellement atteinte par les *arbres de Fibonacci* qui sont définis comme suit : l'arbre Φ_0 est l'arbre vide, l'arbre Φ_1 est réduit à un seul sommet; l'arbre Φ_{k+2} a un sous-arbre gauche égal à Φ_{k+1}, et un sous-arbre droit égal à Φ_k. La hauteur de Φ_k est $k-1$, et Φ_k a $F_{k+2}-1$ sommets.

L'*implémentation* des arbres *AVL* se fait avantageusement comme celle des arbres binaires de recherche (voir chapitre 3). Pour rendre efficaces les opérations de rééquilibrage dont il sera question plus loin, il convient d'ajouter un champ supplémentaire à chaque sommet qui contient la hauteur du sous-arbre. Pour un arbre binaire de recherche *AVL*, les types s'écrivent donc

```
arbre = ^sommet;
sommet = RECORD
            haut: integer;
            val: element;
            g, d: arbre
         END;
```

Certains auteurs recommandent de ne pas conserver la hauteur, mais seulement l'équilibre en chaque sommet. Comme l'équilibre peut se coder sur deux bits, il en résulte un gain de place. En revanche, les algorithmes sont plus difficiles à mettre en œuvre. Le gain de place est en fait réduit en codant la hauteur sur un octet. On peut alors représenter des arbres de hauteur 255, donc avec environ 2^{170} sommets, ce qui suffit en pratique.

6.2.2 Insertion

Les arbres *AVL* sont utilisés comme implémentation de dictionnaires. On considère donc des arbres binaires de recherche qui sont en plus des arbres *AVL*. Les opérations de recherche, d'insertion et de suppression se font, dans un arbre binaire de recherche, en temps proportionnel à la hauteur de l'arbre. Cette hauteur est logarithmique en fonction du nombre de sommets dans un arbre *AVL*. Ainsi, la recherche se fait en temps logarithmique. Pour l'insertion et la suppression, il convient de maintenir le caractère *AVL* après l'opération, ce qui se traduit par un rééquilibrage. Comme on verra, le coût du rééquilibrage est, lui aussi, proportionnel à la hauteur, ce qui garantit un coût total logarithmique.

Soit donc à insérer une clé c dans un arbre binaire de recherche *AVL* A. Si A est vide, le résultat est un arbre A' formé d'un seul sommet de contenu c, et cet arbre est *AVL*. Supposons donc A non vide.

Dans une *première phase*, on utilise l'algorithme d'insertion habituel (voir chapitre 3) dans un arbre binaire de recherche : on descend dans A à partir de sa racine r, et on crée une nouvelle feuille s contenant la clé c. Soit A' l'arbre ainsi obtenu.

Dans une *deuxième phase*, on remonte le chemin γ entre r et s, en progressant de s vers r, et on teste si l'arbre A' est encore *AVL*. Seuls les sous-arbres dont les racines sont sur le chemin γ peuvent changer de hauteur ; cette hauteur ne peut qu'augmenter, et elle ne peut augmenter que de 1.

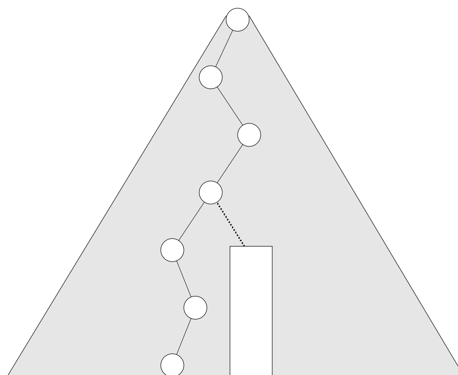

Figure 2.3: *Le chemin γ de rééquilibrage.*

Si les racines de ces sous-arbres vérifient les conditions d'équilibre des arbres *AVL*, l'arbre A' tout entier est *AVL*. Sinon, il existe un premier sommet x (en partant de s) sur le chemin γ dont l'équilibre, dans A' vaut 2 ou -2. Supposons, pour fixer les idées, que le fils gauche u de x est aussi sur γ (voir figure 2.3); alors l'équilibre de x est 2. Posons $h = \mathit{hauteur}(A(x))$. Alors

$$h - 1 = \mathit{hauteur}(A(u)), \quad h = \mathit{hauteur}(A'(u)), \quad h - 2 = \mathit{hauteur}(A_d(x)).$$

L'arbre $A_d(x)$ peut être vide, voir figure 2.4.

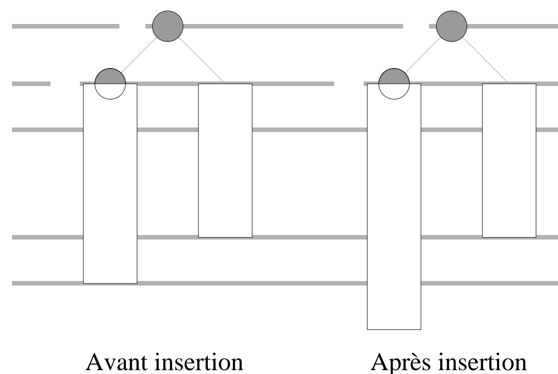

Figure 2.4: *L'arbre de racine x avant et après insertion.*

Posons $A'(u) = (u, X, Y)$ et $Z = A_d(x)$. Les arbres $A'(u)$ et Z sont *AVL*.

Deux cas se présentent alors :

(1) L'insertion s'est faite dans le sous-arbre *gauche* de u. On a alors $\mathit{hauteur}(X) = h - 1$ et $\mathit{hauteur}(Y) = h - 2$ (car $A'(y)$ est *AVL*). Dans ce cas, on effectue une rotation droite de $A'(x)$, ce qui donne l'arbre $A'' = (u, X, (x, Y, Z))$, voir figure 2.5. L'arbre A'' est un arbre *AVL*, et la hauteur de A'' est égale à la hauteur de $A(x)$, c'est-à-dire de x dans l'arbre avant insertion. Il en résulte qu'après cette rotation, l'arbre tout entier est *AVL*, et que la phase de rééquilibrage est terminée.

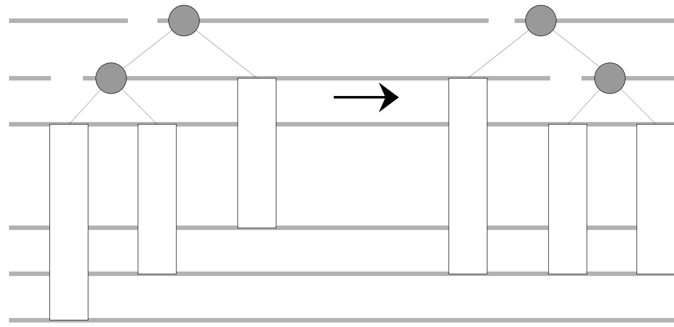

Figure 2.5: *Cas* (1) *: une rotation simple rétablit l'équilibre.*

(2) L'insertion s'est faite dans le sous-arbre *droit* de u. On a alors $hauteur(X) = h-2$ et $hauteur(Y) = h-1$. Posons $Y = (v, V, W)$. L'un des deux arbres V ou W est de hauteur $h-2$, et l'autre a pour hauteur $h-3$. On effectue alors une rotation gauche-droite de $A'(x)$, ce qui donne l'arbre $A'' = (v, (u, X, V), (x, W, Z))$, voir figure 2.6. Comme dans le premier cas, l'arbre A'' est un arbre *AVL* de même hauteur que $A(x)$. Le rééquilibrage s'arrête donc après cette double rotation.

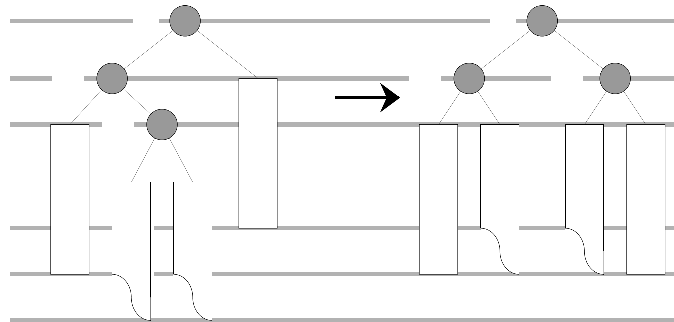

Figure 2.6: *Cas* (2) *: une rotation double rétablit l'équilibre.*

Proposition 2.2. *L'insertion dans un arbre AVL à n sommets se réalise en temps $O(\log n)$. Il suffit d'au plus une rotation ou double rotation pour rééquilibrer l'arbre après insertion.*

Preuve. Il reste à constater que les comparaisons de hauteurs se font, pour chaque sommet du chemin de remontée, en temps constant, ce qui est évident. ∎

6.2.3 Suppression

Considérons maintenant la suppression d'une clé c dans un arbre binaire de recherche *AVL* A. Si A n'a qu'un seul sommet, de contenu c, le résultat est l'arbre vide. On suppose donc que A a au moins deux sommets.

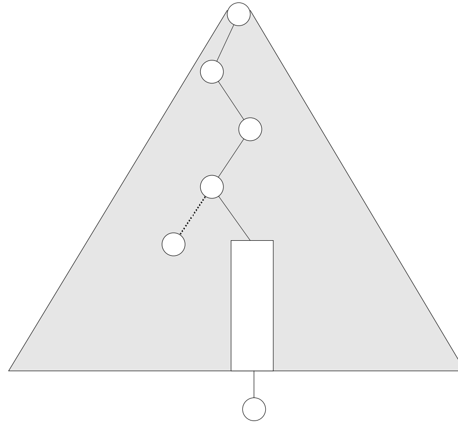

Figure 2.7: *Le chemin de rééquilibrage.*

Dans une *première phase*, on utilise l'algorithme habituel de suppression dans un arbre binaire de recherche (voir chapitre 3) : on descend dans A à partir de la racine r à la recherche du sommet t contenant la clé c; si t est une feuille, on la supprime, sinon on remplace le contenu de t par le contenu de la feuille la plus à droite du sous-arbre gauche de t (respectivement de la feuille la plus à gauche du sous-arbre droit de t), et on supprime cette feuille. Dans tous les cas, on supprime donc une feuille dans A. Notons-la s, soit A' l'arbre après suppression, et soit γ le chemin de r à s.

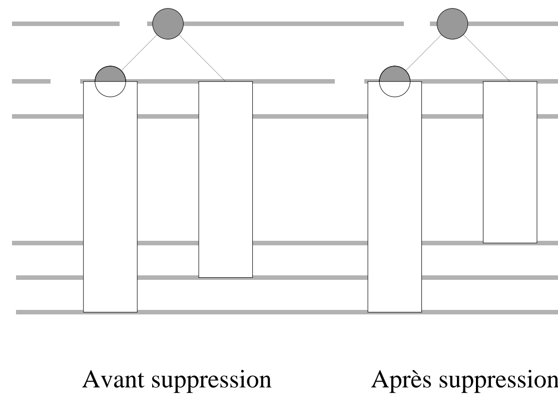

Figure 2.8: *L'arbre de racine x avant et après suppression.*

Dans une *deuxième phase*, on remonte le chemin γ de s vers r et on teste si l'arbre A' est encore *AVL*. Seuls les sous-arbres dont les racines sont sur le chemin γ peuvent changer de hauteur, et cette hauteur peut diminuer de 1. Si les racines de ces sous-arbres vérifient la condition d'équilibre, l'arbre A' est *AVL*. Sinon, il existe un premier sommet x (en partant du père de s) dont l'équilibre vaut 2 ou -2. Supposons, pour fixer les idées, que le fils gauche u de x n'*est pas* sur le chemin γ (voir figure 2.7), donc que la suppression s'est faite dans le sous-arbre droit de x. Après suppression, ce sous-arbre est transformé en un arbre *AVL* éventuellement

vide que nous notons Z. L'équilibre de x est 2. Posons $h = hauteur(A(x))$. Alors (voir figure 2.8)

$$h-1 = hauteur(A(u)), \quad h-2 = hauteur(A_d(x)), \quad h-3 = hauteur(Z).$$

Posons $A(u) = (u, X, Y)$. Deux cas se présentent alors :

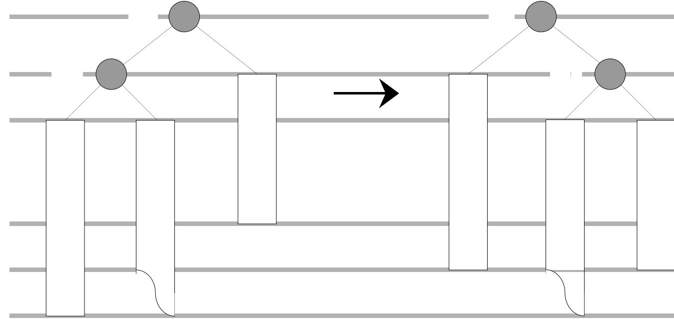

Figure 2.9: *Une rotation simple sur le chemin de rééquilibrage.*

(1) La hauteur de X est $h-2$. Dans ce cas, la hauteur de Y est $h-2$ ou $h-3$. On effectue une rotation droite de $A'(x)$, ce qui donne l'arbre $A'' = (u, X, (x, Y, Z))$ qui est *AVL*, voir figure 2.9. L'arbre A'' a hauteur h ou $h-1$, selon que Y a hauteur $h-2$ ou $h-3$. Dans le premier cas, A'' a donc la même hauteur que $A(x)$, et le processus de rééquilibrage s'arrête; dans le deuxième cas, il faut continuer en remontant vers la racine.

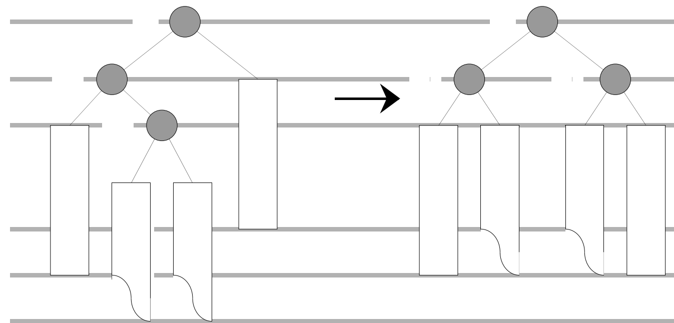

Figure 2.10: *Une rotation double sur le chemin de rééquilibrage.*

(2) La hauteur de X est $h-3$. Alors la hauteur de Y est $h-2$, parce que $A(u)$ est un arbre *AVL*. Posons $Y = (v, V, W)$. On effectue une rotation gauche-droite donnant l'arbre $A'' = (v, (u, X, V), (X, W, Z))$, voir figure 2.10. Cet arbre est *AVL*, mais sa hauteur est $h-1$. En d'autres termes, le processus de rééquilibrage doit continuer, en remontant vers la racine.

Proposition 2.3. *La suppression dans un arbre AVL à n sommets se réalise en temps $O(\log n)$.* ∎

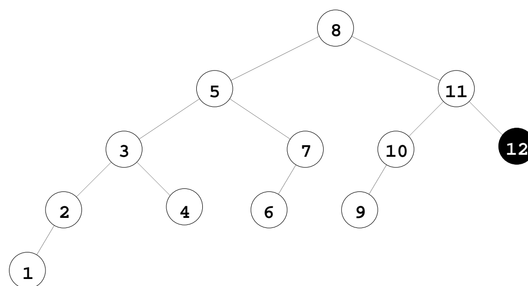

Figure 2.11: *Arbre de Fibonacci avant la suppression de la clé* 12.

Exemple. Considérons l'arbre (de Fibonacci) de la figure 2.11. La suppression de la clé 12 amène une première rotation autour du sommet de clé 11, puis une deuxième autour de la racine (figure 2.12).

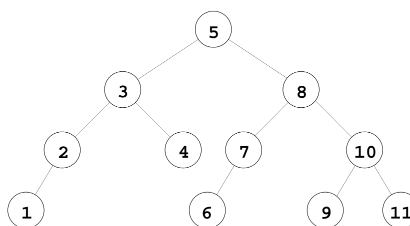

Figure 2.12: *Après suppression et rééquilibrage.*

6.2.4 Arbres balisés

Comme nous l'avons déjà dit dans l'introduction, il existe deux variantes des arbres binaires de recherche : ceux où tous les sommets contiennent des clés (et les informations associées), et ceux où seules les feuilles contiennent des clés. Les nœuds de ces arbres contiennent alors des *balises* (qui sont en général aussi des éléments de l'ensemble des clés) permettant de naviguer dans l'arbre.

Les arbres *AVL* balisés nous seront utiles plus loin. Chaque feuille d'un tel arbre contient donc une clé, et les clés sont croissantes de la gauche vers la droite. Nous avons déjà décrit comment on réalise l'insertion et la suppression dans un arbre balisé. Il n'est pas difficile de se convaincre que le rééquilibrage par rotations se transpose sans grand changement aux arbres balisés, puisque seuls les nœuds changent de fils. L'analyse du rééquilibrage dans les arbres *AVL* se transpose également dans les arbres balisés. On peut donc, au choix, utiliser des arbres *AVL* ou des arbres *AVL* balisés.

6.3 Arbres *a–b*

Les arbres *a–b* sont des arbres dont toutes les feuilles ont même profondeur, et le nombre de fils d'un nœud varie entre a et b. Nous montrons ici comment réaliser les opérations de recherche, d'insertion, de suppression, de scission et de concaténation en temps logarithmique en fonction du nombre de sommets de l'arbre. Les rééquilibrages sont en fait assez rares, puisqu'en moyenne deux opérations de rééquibrage insertion et suppression suffisent dans le cas des arbres 2–4. Les arbres *a–b* constituent ainsi une implémentation efficace des listes concaténables.

6.3.1 Définition

Soient a et b deux entiers, avec $a \geq 2$, et $b \geq 2a - 1$. Un *arbre a–b* est un arbre A vérifiant les conditions suivantes :

(i) les feuilles ont toutes la même profondeur;
(ii) la racine a au moins 2 et au plus b fils;
(iii) les autres nœuds ont au moins a et au plus b fils.

On note $d(x)$ le nombre de fils d'un nœud x, et $A_i(x)$ le i-ième sous-arbre de x, pour $i = 1, \ldots, d(x)$.

Soit S un ensemble de clés. Un arbre A est un *arbre a–b pour S* si les éléments de S sont rangés aux feuilles de A en ordre croissant de la gauche vers la droite, et si de plus, chaque nœud x de A contient une suite de $d(x)-1$ clés $k_1 < \cdots < k_{d(x)-1}$, appelées les *balises* de x et vérifiant les conditions :

- les clés des feuilles de $A_i(x)$ sont inférieures ou égales à k_i, pour $i = 1, \ldots, d(x) - 1$;
- les clés des feuilles de $A_i(x)$ sont strictement supérieures à k_{i-1}, pour $i = 2, \ldots, d(x)$.

En d'autres termes, si c_i est une clé d'une feuille de $A_i(x)$, on a

$$c_1 \leq k_1 < c_2 \leq k_2 < \cdots \leq k_{d(x)-1} < c_{d(x)}.$$

Il est commode de noter $k_i(x)$ la i-ième balise du nœud x. Lorsque $b = 2a - 1$, un arbre *a–b* est appelé un B-arbre d'ordre $a - 1$. L'arbre vide est un arbre *a–b*, de même que l'arbre formé d'un seul sommet qui est une feuille.

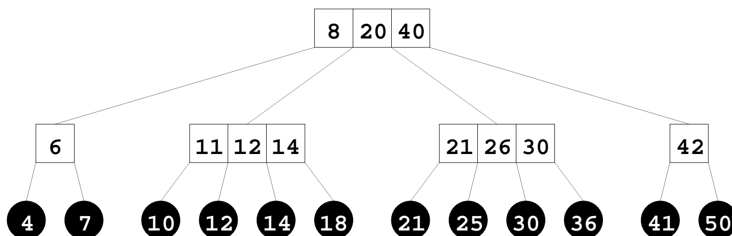

Figure 3.1: *Un arbre 2–4.*

Proposition 3.1. *Soit A un arbre a–b avec n feuilles ($n > 0$) de hauteur h. Alors*

$$2a^{h-1} \leq n \leq b^h$$

ou encore

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

Preuve. Tout nœud a au plus b fils, donc A a au plus b^h feuilles. Tout nœud autre que la racine a au moins a fils, et la racine au moins 2 ; au total, il y a au moins $2a^{h-1}$ feuilles. Le deuxième encadrement en découle. ∎

6.3.2 Recherche d'un élément

Soit A un arbre a–b pour un ensemble S. La *recherche* d'une clé c dans A est dite *positive* si $c \in S$, elle est *négative* sinon. Si A est vide, la recherche est négative ; dans les autres cas, elle se fait comme suit :

```
RECHERCHER (c, A);
  x =RACINE(A);
  tantque x est un nœud faire
    i := 1;
    tantque c > k_i(x) etalors i < d(x) faire i := i + 1;
    poser x := le i-ième fils de x
  fintantque;
  si c = cle(x) alors retourner(recherche positive)
  sinon retourner(recherche negative).
```

La recherche du sous-arbre $A_i(x)$ susceptible de contenir la clé c se fait en comparant c aux balises $k_i(x)$. Ceci se réalise en temps constant, c'est-à-dire indépendant de la taille de A. On a donc

Proposition 3.2. *Soit A un arbre a–b pour un ensemble S à n éléments ($n > 0$). La recherche d'une clé dans A se fait en temps $O(\log n)$.* ∎

6.3.3 Insertion d'un élément

L'*insertion* d'une clé c dans un arbre a–b pose, comme nous allons le voir, le problème du rééquilibrage. Il se peut en effet qu'après insertion, un nœud de l'arbre ait $b + 1$ fils. Dans ce cas, on *éclate* ce nœud en deux nœuds frères qui se partagent ces $b + 1$ fils. Il se peut que leur père, à son tour, ait trop de fils ; on

répète alors cette procédure. Plus précisément, l'insertion se fait en deux phases. On suppose que A a au moins 2 sommets, les autres cas étant faciles.

Dans la *première* phase, on descend dans l'arbre à partir de la racine à la recherche d'une feuille y susceptible de contenir la clé c. Si la recherche est positive, la clé c figure déjà dans A et il n'y a pas lieu de l'ajouter; sinon, soit x le père de y. Soient $c_1, \ldots, c_d$ les clés des fils de x, avec $d = d(x)$, et soient $k_1, \ldots, k_{d-1}$ les balises du nœud x. On a donc

$$c_1 \leq k_1 < c_2 \leq \cdots \leq k_{d-1} < c_d.$$

La recherche de c parmi les fils de x détermine un indice i tel que $k_{i-1} < c \leq k_i$ (si $i = 1$, on a seulement $c \leq k_1$, et si $i = d$, seulement $k_{d-1} < c$).

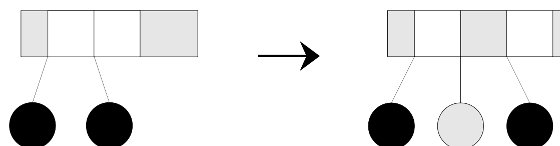

Figure 3.2: *Insertion d'une feuille, cas $c < c_i$.*

On crée alors une feuille de clé c, et on l'insère comme i-ième fils de x si $c < c_i$, et comme $i+1$-ième fils de x si $c_i < c$. On insère également la clé $\min(c, c_i)$ comme i-ième balise au nœud x. Les figures 3.2 et 3.3 décrivent les deux transformations.

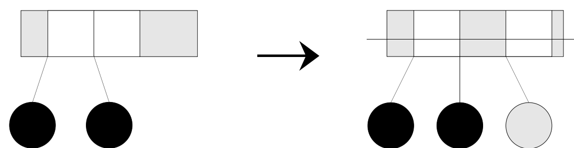

Figure 3.3: *Insertion d'une feuille, cas $c > c_i$.*

Dans la *deuxième* phase, on rééquilibre l'arbre si c'est nécessaire. Si le nœud x a au plus b fils après insertion, aucun rééquilibrage n'est nécessaire, et l'algorithme est terminé. Sinon, $d(x) = b + 1$, et on éclate le nœud x.

L'*éclatement* de x est l'opération suivante : on crée deux nœuds frères x' et x'', et on partage les $b + 1$ fils de x en deux groupes, respectivement de $\lfloor (b+1)/2 \rfloor$ et $\lceil (b+1)/2 \rceil$ éléments, le premier fournissant les fils de x', le second les fils de x''. Comme $b \geq 2a - 1$, on a $\lfloor (b+1)/2 \rfloor \geq a$, donc x' et x'' vérifient les contraintes d'un arbre a–b. Si x a un père, soit z, les frères x' et x'' sont déclarés fils de z à la place de x. Sinon, on crée un nouveau nœud z (qui devient la nouvelle racine de l'arbre) dont x' et x'' sont les seuls fils. C'est de cette manière que l'arbre prend de la hauteur.

Les b balises $k_1(x), \ldots, k_b(x)$ du nœud x sont réparties comme suit : les balises d'indice 1 à $\lfloor (b+1)/2 \rfloor - 1$ deviennent les balises de x', les balises d'indice $\lfloor (b+1)/2 \rfloor + 1$ à b deviennent les balises de x''; quant à la balise dont l'indice est

$\lfloor (1+b)/2 \rfloor$, elle devient la balise qui, dans le père commun z de x' et x'', sépare x' de x''.

L'éclatement peut se représenter de manière schématique par la règle suivante :

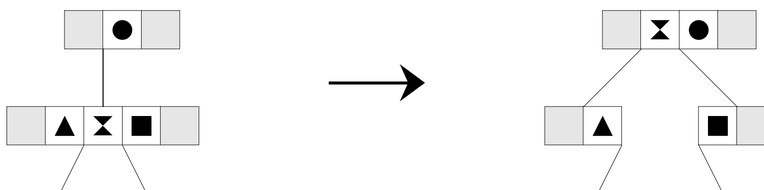

Figure 3.4: *Règle d'éclatement.*

Revenons à l'insertion. L'éclatement d'un nœud x augmente le nombre de fils de son père. Si nécessaire, on répète l'éclatement sur le père, puis sur le père de celui-ci, etc. A la fin, l'arbre est à nouveau a–b.

Proposition 3.3. *Soit A un arbre a–b pour un ensemble S à n éléments ($n > 0$). L'insertion d'une clé dans A se fait en temps $O(\log n)$.*

Preuve. La localisation de l'endroit où insérer la clé prend un temps proportionnel à la hauteur h de A. L'insertion de la feuille est suivie d'au plus h éclatements de nœuds. Chaque éclatement prend un temps constant. ■

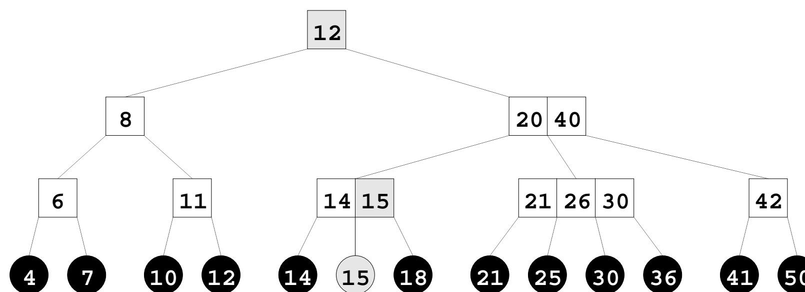

Figure 3.5: *L'arbre après insertion de* 15*.*

Exemple. L'insertion de la clé 15 dans l'arbre de la figure 3.1 provoque d'abord l'éclatement du deuxième fils de la racine, puis de la racine elle-même. L'arbre entier prend donc la forme donnée dans la figure 3.5.

6.3.4 Suppression d'un élément

La *suppression* d'une clé dans un arbre a–b est plus compliquée. Le rééquilibrage fait appel à l'opération inverse de l'éclatement, la *fusion*. Il est utile d'introduire

une opération supplémentaire, appelée *partage* qui, même si elle n'est pas indispensable, facilite la suppression.

Soient x' et x'' deux nœuds frères d'un arbre a–b, et soit z leur père. On suppose de plus que x' et x'' sont des fils consécutifs de z, disons x' à gauche de x''. Si le nombre total de fils de x' et x'' est au plus b, on peut *fusionner* x' et x'' en un seul nœud x, fils de z, qui prend comme fils l'ensemble des fils de x' et x''. Schématiquement, la fusion, opération inverse de l'éclatement, peut se représenter par la règle suivante :

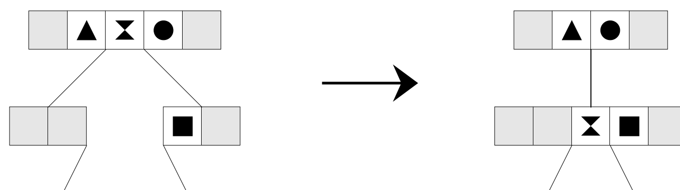

Figure 3.6: *Règle de fusion.*

Pour la deuxième opération, le *partage*, considérons à nouveau deux frères adjacents x' et x'', fils d'un nœud z, avec disons x' à gauche de x''. Si x' a moins de b fils, et x'' a plus de a fils, le partage consiste à transférer le sous-arbre le plus à gauche de x'', soir $A_1(x'')$, vers le nœud x', qui le reçoit donc comme sous-arbre le plus à droite : $A_{1+d(x')}(x') := A_1(x'')$. Ce transfert se schématise comme suit :

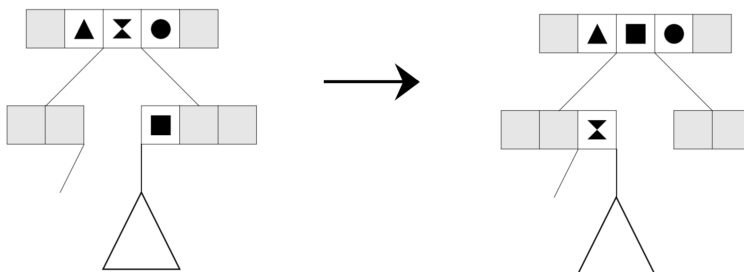

Figure 3.7: *Règle du partage.*

L'algorithme de *suppression* d'une clé c dans un arbre a–b A est le suivant : on cherche la feuille contenant la clé c. Soit x le père de cette feuille; on supprime la feuille et la clé, dans x, de même indice (ou d'indice $d(x) - 1$ si la feuille est le fils le plus à droite de x).

Considérons d'abord le cas où x est la racine de l'arbre. Si x a plus de deux fils, l'arbre est un arbre a–b, sinon x a un fils unique; on supprime alors x et son fils unique devient la racine de l'arbre (sa hauteur diminue donc de 1).

Si x n'est pas la racine, et si x a au moins a fils après cette suppression, l'arbre résultant est encore un arbre a–b et l'opération est terminée. Sinon, x a $a - 1$ fils, et il faut rééquilibrer l'arbre. Soit z le père de x.

Si x a un frère gauche ou un frère droit y qui a exactement a fils, on fusionne x et y en un fils de z qui a $2a - 1 \leq b$ fils.

Si x a un frère gauche ou droit y qui a plus de a fils, on fait un partage entre x et y : le nœud x reçoit un fils supplémentaire, donc en a maintenant a, et y en perd un, mais en a encore au moins a, et z garde les mêmes fils : l'arbre est a–b.

Dans le cas d'une fusion, le nœud z perd un fils; il se peut qu'il n'ait plus que $a-1$ fils. Il convient alors de continuer le rééquilibrage. En revanche, une opération de partage met fin au rééquilibrage. En résumé, les cas suivants se présentent, en fonction du nombre de fils des frères gauche et droit du nœud x (en supposant bien sûr que x ait deux frères adjacents) :

(i) les deux frères ont a fils : seule une fusion est possible;

(ii) l'un des deux frères a a fils, l'autre en a plus de a : un partage et une fusion sont possibles;

(iii) les deux frères ont plus de a fils : un partage est possible, et une fusion aussi, si le nombre total de fils ne dépasse pas b.

Dans certaines situations, on peut donc choisir entre une fusion ou un partage. Comme un partage arrête le rééquilibrage, on a intérêt à privilégier les partages.

Un partage s'apparente à une fusion suivie d'un éclatement (modifié). En effet, si un partage est possible, une fusion donne un nœud qui a assez de fils pour être éclaté, et on obtient le résultat du partage en éclatant le sommet, mais avec une répartition des fils qui n'est pas aussi équilibrée que celle décrite plus haut. On peut donc remplacer un partage par une fusion suivie d'un éclatement, et c'est en ce sens que le partage n'est pas indispensable au rééquilibrage.

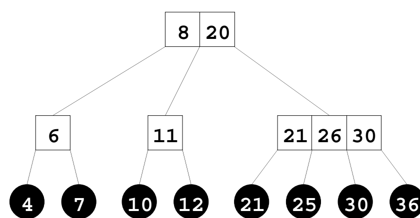

Figure 3.8: *Un arbre 2–4.*

Exemple. Considérons l'arbre de la figure 3.8. Après suppression de 12, l'arbre n'est plus équilibré (figure 3.9).

Le nœud sans balise peut être rééquilibré par un partage avec son frère droit. On obtient l'arbre de la figure 3.10.

Il peut aussi être rééquilibré par fusion avec son frère gauche. On obtient alors l'arbre de la figure 3.11.

Proposition 3.4. *Soit A un arbre a–b pour un ensemble S à n éléments ($n > 0$). La suppression d'une clé dans A se fait en temps $O(\log n)$.* ∎

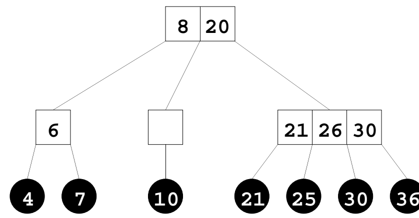

Figure 3.9: *Après suppression de* 12 *et avant rééquilibrage.*

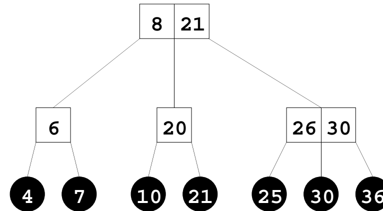

Figure 3.10: *Après partage avec le frère droit.*

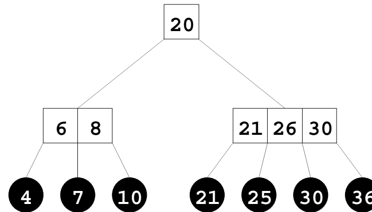

Figure 3.11: *Après fusion avec le frère gauche.*

D'autres opérations simples sur les ensembles de clés peuvent être considérées, comme par exemple :

MIN(S);
    détermine le plus petit élément de S;

SUPPRIMER-MIN(S);
    supprime le plus petit élément de S.

La première opération se réalise en descendant dans la branche gauche de l'arbre, et pour la deuxième, il suffit d'appliquer ensuite l'opération de suppression. On traite de manière analogue MAX(S) et SUPPRIMER-MAX(S).

Théorème 3.5. *Si un ensemble S est représenté par un arbre a–b, les opérations* RECHERCHER(c, S), INSÉRER(c, S), SUPPRIMER(c, S), MIN(S), MAX(S), SUPPRIMER-MIN(S) *et* SUPPRIMER-MAX(S) *se réalisent toutes en temps* $O(\log(1 + |S|))$.

6.3.5 Concaténation et scission

On considère maintenant deux opérations supplémentaires très utiles sur les ensembles ordonnés qui se réalisent de façon efficace à l'aide d'arbres *a–b*. Ce sont :

Concaténer(S_1, S_2, S) ;
: est défini lorsque $\max(S_1) < \min(S_2)$, le résultat est $S = S_1 \cup S_2$.

Scinder(S, c, S_1, S_2) ;
: définit une décomposition $S = S_1 \cup S_2$, avec $S_1 = \{u \in S \mid u \leq c\}$ et $S_2 = \{u \in S \mid u > c\} = S - S_1$.

Proposition 3.6. *Soient S_1 et S_2 deux ensembles représentés par des arbres a–b. L'opération* Concaténer(S_1, S_2, S) *peut se réaliser en temps* $O(\log(1 + \max(|S_1|, |S_2|)))$. *Pour un ensemble S représenté par un arbre a–b, l'opération* Scinder(S, c, S_1, S_2) *peut se réaliser en temps* $O(\log(1 + |S|))$.

Preuve. Soit A_i un arbre *a–b* pour S_i, et soit h_i la hauteur de A_i, pour $i = 1, 2$. On suppose $h_1 \geq h_2$, l'autre cas se traite de la même manière.

Soit c une clé telle que $\max(S_1) \leq c < \min(S_2)$. On peut calculer une telle clé en temps $O(h_1)$, en calculant Max(S_1) sur A_1. La concaténation se réalise comme suit.

Partant de la racine de A_1, on descend sa branche la plus à droite jusqu'au nœud x de profondeur $h_1 - h_2$. Ainsi, le sous-arbre $A_1(x)$ de racine x a la même hauteur que A_2. On fusionne alors ce sommet x et la racine r_2 de A_2 en un nœud unique disons z, en ajoutant la clé c=Max(S_1) comme clé centrale dans la liste des balises du nœud z.

Si z a moins de b fils, c'est terminé ; sinon, z a au plus $2b$ fils et un éclatement donne deux nœuds ayant chacun au plus b fils. Une remontée vers la racine permet de rééquilibrer l'arbre par des éclatements successifs.

Moyennant la connaissance de c, l'opération de concaténation se fait donc en temps $O(1+|h_1 - h_2|)$. Pour déterminer c, un temps $O(\log(1+|S_1|))$ est suffisant. Dans la suite, la clé c sera disponible au moment de la concaténation.

Venons-en à la *scission*. Soit A un arbre *a–b* pour l'ensemble S, et soit c une clé ; on descend dans l'arbre à la recherche de la clé c ; soit y la feuille détectée. En cas de recherche positive, la clé de y est c ; en cas de recherche négative, la clé de y est la plus grande clé dans S qui est inférieure à c (ou la plus petite clé de S supérieure à c). En d'autres termes, l'ensemble S_1 est formé des clés de toutes les feuilles à gauche de y, et S_2 est formé des clés des feuilles à droite de y. La clé de y appartient à S_1 ou S_2, selon les cas.

On considère maintenant le chemin menant de la racine à la feuille y. On supprime les arêtes de ce chemin et on éclate les sommets sur le chemin. Toutefois, l'éclatement est modifié comme suit : si l'arête supprimée dans un nœud x est la k-ième, le nœud x est éclaté en deux nœuds x' qui reçoit les $k - 1$ premiers fils et

les $k-1$ premières balises, et x'' qui reçoit les autres fils et les balises de droite, y compris la k-ième. Chacun de ces nœuds a donc une balise supplémentaire qui servira ensuite, et chacun des arbres est a–b, sauf peut-être en sa racine. Enfin, si $k=1$, seul le nœud x'' est créé, et de même si $k=d(x)$, seul x' est créé.

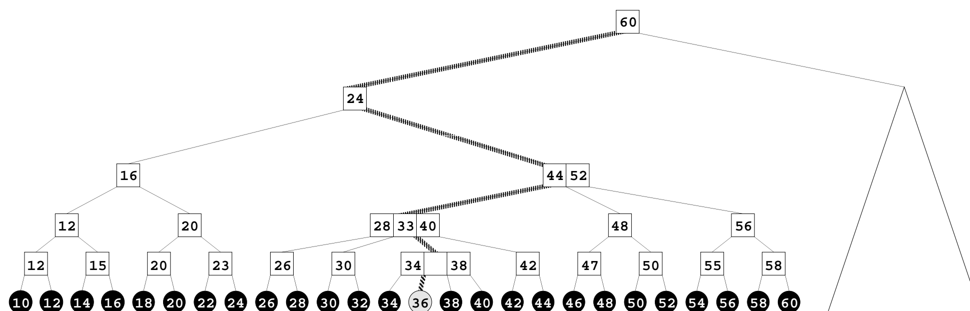

Figure 3.12: *Un arbre* 2–4 *avant la scission par la clé* 36.

On obtient alors deux forêts F_1 et F_2, où F_1 est la suite d'arbres à gauche du chemin, et F_2 la suite d'arbres à droite du chemin.

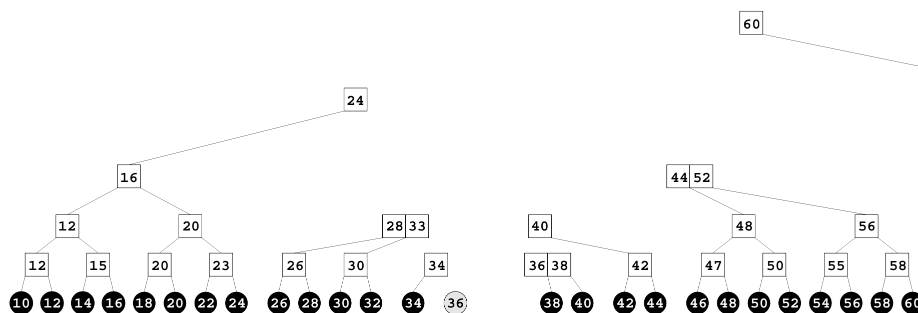

Figure 3.13: *Les deux forêts* F_1 *et* F_2 *après la séparation.*

Soient $D_1, \ldots, D_m$ les arbres constituant F_1, numérotés de la gauche vers la droite. Notons que $m \leq h$, où h est la hauteur de A. Chaque arbre D_i est un arbre a–b sauf peut-être pour la racine qui peut n'avoir qu'un fils. Par ailleurs, $\max(D_i) < \min(D_{i+1})$, et l'on connaît une clé c_i telle que $\max(D_i) \leq c_i < \min(D_i)$. C'est la balise qui se trouve dans le champ le plus à droite de la racine de D_i. Enfin, on a $h_i > h_{i+1}$, où h_i est la hauteur de D_i. On effectue alors la séquence

$$\begin{aligned}&\textsc{Concaténer}(D_{m-1}, D_m, D'_{m-1});\\ &\textsc{Concaténer}(D_{m-2}, D'_{m-1}, D'_{m-2});\\ &\qquad\vdots\\ &\textsc{Concaténer}(D_1, D'_2, D'_1).\end{aligned}$$

Concaténer des arbres qui sont des arbres a–b sauf éventuellement pour la racine qui n'a qu'un seul fils donne un arbre de même nature. Si D'_1 a une racine n'ayant qu'un fils, on supprime la racine, et on obtient un arbre a–b.

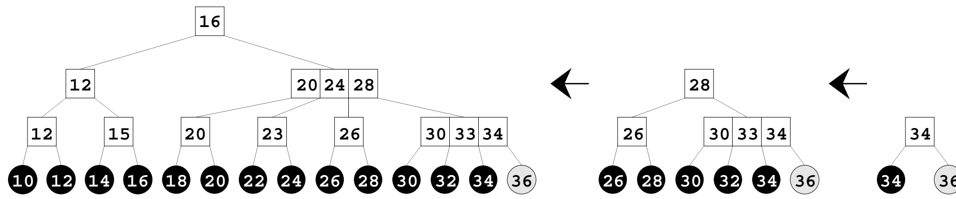

Figure 3.14: *Reconstitution de l'arbre à partir de la forêt F_1.*

Analysons le coût de la reconstruction. Soit h'_i la hauteur de D'_i. Le coût est

$$O(|h_{m-1} - h_m| + \sum_{i=1}^{m-2} |h_i - h'_{i+1}| + m).$$

Or, on a

$$h_{i+1} \leq h'_{i+1} \leq 1 + h_{i+1} \leq h_i \, .$$

En effet, seule l'inégalité centrale doit être prouvée. Pour cela, observons que $h'_{m-1} \leq 1 + \max(h_{m-1}, h_m) = 1 + h_{m-1}$, et, en raisonnant par récurrence descendante, $h'_{i+1} \leq 1 + \max(h_{i+1}, h'_{i+2}) = 1 + h_{i+1}$. Il en résulte que

$$\begin{aligned}
|h_{m-1} - h_m| + \sum_{i=1}^{m-2} |h_i - h'_{i+1}| + m &\leq h_{m-1} - h_m + \sum_{i=1}^{m-2} (h_i - h'_{i+1}) + m \\
&\leq h_{m-1} - h_m + \sum_{i=1}^{m-2} (h'_i - h'_{i+1}) + m \leq h'_1 - h'_{m-1} + h_{m-1} - h_m + m \\
&\leq 2h = O(\log |S|).
\end{aligned}$$

Ceci achève la preuve. ■

6.3.6 Coût amorti des arbres 2–4

Nous analysons maintenant le coût du rééquilibrage non pas sur une seule modification d'un arbre a–b, mais sur une séquence d'insertions et de suppressions. Dans ce cas, une analyse fine du coût est possible, car les opérations de rééquilibrage se répercutent sur plusieurs insertions ou suppressions consécutives. Le résultat que nous établissons montre que le coût total des rééquilibrages est linéaire en fonction du nombre d'insertions et de suppressions, si l'on commence avec un arbre vide. Un corollaire de ce résultat remarquable est qu'en moyenne, le coût du rééquilibrage est constant dans un arbre 2–4.

Proposition 3.7. *On considère une suite quelconque de i insertions et de d suppressions dans un arbre 2–4 initialement vide, et on pose $n = i + d$; alors $P \leq d \leq n$ et $E + F \leq n + (i - d - 1)/2$, où P est le nombre de partages, E le nombre d'éclatements, et F le nombre de fusions de sommets.*

En particulier, la proposition précédente admet le corollaire suivant :

Théorème 3.8. *On considère une suite quelconque de n insertions ou suppressions dans un arbre 2–4 initialement vide. Alors le nombre total d'opérations de rééquilibrage est au plus $3n/2$.*

Ce résultat montre en particulier qu'il y a en moyenne 3/2 opérations de rééquilibrage par insertion ou suppression. Ce nombre est indépendant de la taille de la structure, et est remarquablement faible.

Preuve du théorème. En vertu de la proposition, on a

$$E + F + P \leq n + (i - d - 1)/2 + d = n + (i + d - 1)/2 \leq 3n/2$$

parce que $i + d = n$. ■

La preuve de la proposition repose sur une *mesure d'équilibre* d'un arbre 2–4 que nous allons introduire maintenant. Cette mesure indique, pour tout nœud de l'arbre, s'il est «bien» équilibré, c'est-à-dire en fait s'il a 3 fils. Cette mesure sera définie et étudiée pour des arbres qui ne sont pas des arbres 2–4, mais des arbres en cours de rééquilibrage. Un couple (A, s), où s est un sommet de l'arbre A, est un arbre 2–4 *partiellement équilibré* si

$$\begin{array}{l} 1 \leq d(s) \leq 5, \\ 2 \leq d(x) \leq 4 \quad \text{pour } x \neq s. \end{array}$$

L'*équilibre* d'un sommet x d'un arbre partiellement équilibré est le nombre $e(x)$ défini par $e(x) = \min(d(x) - 2, 4 - d(x))$. Il est immédiat que

$$e(x) = \begin{cases} -1 & \text{si } d(x) = 1 \text{ ou } d(x) = 5, \\ 0 & \text{si } d(x) = 2 \text{ ou } d(x) = 4, \\ 1 & \text{si } d(x) = 3. \end{cases}$$

L'*équilibre* de l'arbre A est la somme des équilibres de ses nœuds :

$$e(A) = \sum_{x \in \text{noeuds}(A)} e(x).$$

L'idée de la construction, et de la preuve de l'estimation, est que toute opération d'insertion ou de suppression de feuille dans l'arbre le déséquilibre, au sens de la mesure introduite, alors qu'une opération de fusion, d'éclatement ou de partage le rééquilibre.

Proposition 3.9. *Soit A un arbre 2–4 et soit A' obtenu par l'insertion ou la suppression d'une feuille sans rééquilibrage; alors $e(A') \geq e(A) - 1$.* ■

Proposition 3.10. *Soit A un arbre 2–4 à m feuilles; alors $0 \leq e(A) \leq (m-1)/2$.*

Preuve. On note m_i le nombre de nœuds ayant i fils. Alors $e(A) = m_3$. Par ailleurs, le nombre d'arêtes de l'arbre A est $2m_2 + 3m_3 + 4m_4$, et il est aussi égal à $m + m_2 + m_3 + m_4 - 1$, puisque chaque sommet, à l'exception de la racine, a un père. Il résulte de l'égalité de ces expressions que

$$m_2 + 2m_3 + 3m_4 = m - 1$$

d'où le résultat. ∎

Lemme 3.11 (éclatement). *Soit (A, s) un arbre 2–4 partiellement équilibré, et supposons que le sommet s a 5 fils; soit A' l'arbre obtenu par éclatement du sommet s. Alors $e(A') \geq 1 + e(A)$.*

Preuve. Notons d'abord que A' est partiellement équilibré. Si s n'est pas la racine de A, soit x le père de s dans A. Dans l'arbre A', le sommet x a deux fils s' et s'' à la place de s; ces deux sommets se partagent les 5 fils de s (voir figure 3.15).

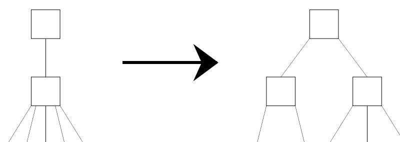

Figure 3.15: *Eclatement de x.*

Notons e' l'équilibre de sommets dans l'arbre A'. Alors $e'(x) \geq e(x) - 1$, et $e'(s') = 0$, $e'(s'') = 1$ ou vice-versa, d'où

$$e'(x) + e'(s') + e'(s'') = e'(x) + 1 \geq e(x) = 1 + e(x) + e(s)$$

et comme l'équilibre des autres sommets n'est pas modifié, ceci montre que $e(A') \geq 1 + e(A)$. Si s est la racine de A, alors A' a une nouvelle racine, soit r, ayant deux fils s' et s'' qui se partagent les 5 fils de s. On obtient

$$e'(r) + e'(s') + e'(s'') = 1 > 1 + e(s)$$

donc ici encore $e(A') \geq 1 + e(A)$. ∎

Lemme 3.12 (partage). *Soit (A, s) un arbre 2–4 partiellement équilibré dont le sommet s a un seul fils, et supposons que s possède un frère voisin t ayant au moins 3 fils. Soit A' l'arbre obtenu par partage entre s et t. Alors $e(A') \geq e(A)$.*

Preuve. Notons e' l'équilibre de sommets dans l'arbre A'. Comme $e'(t) \geq e(t) - 1$, $e'(s) = 0$, $e(s) = -1$, on obtient $e'(t) + e'(s) \geq e(t) - 1 = e(t) + e(s)$. Les autres sommets ne sont pas modifiés; il en résulte que $e(A') \geq 1 + e(A)$. ∎

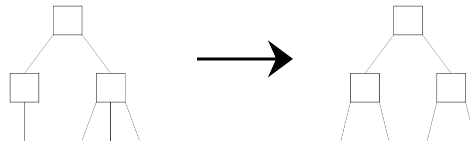

Figure 3.16: *Partage entre les frères s et t.*

Lemme 3.13 (fusion). *Soit (A, s) un arbre 2–4 partiellement équilibré dont le sommet s a un seul fils. Soit A' l'arbre obtenu en supprimant s, si s est la racine, et l'arbre obtenu par fusion de s et t, si s n'est pas la racine, et si s possède un frère voisin t ayant 2 fils. Alors $e(A') \geq 1 + e(A)$.*

Preuve. Considérons d'abord le cas où s est la racine. Si s est le sommet unique de A, alors l'arbre A' est vide, et on a $e(A') = 0 = 1 + e(A)$. Sinon, soit t le fils de s ; alors A' est l'arbre de racine t, et on a encore $e(A') = 1 + e(A)$.

Si s n'est pas la racine de A, soit x le père de s et t ; dans A', les deux fils s et t de x sont remplacés par un unique fils s' réunissant les fils de s et t (figure 3.17).

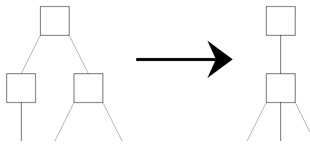

Figure 3.17: *Fusion des frères s et t.*

Notons e' l'équilibre de sommets dans A'. Comme $e'(x) \geq e(x) - 1$ et $e'(s') = 1$, on obtient $e'(x) + e'(s') \geq e(x) \geq 1 + e(x) + e(s) + e(t)$. Les autres sommets n'étant pas modifiés, on a $e(A') \geq 1 + e(A)$. ■

Preuve de la proposition 3.7. Il y a au plus un partage par suppression et aucun par insertion ; les inégalités sur P sont donc évidentes.

Pour prouver l'autre inégalité, soit A l'arbre obtenu, à partir de l'arbre vide après les n opérations d'insertion ou suppression. L'arbre vide a pour équilibre 0. Par ailleurs, chacune des n insertions ou suppressions diminue l'équilibre d'au plus 1 (Prop 3.9), chaque éclatement ou fusion l'augmente d'au moins 1, et chaque partage l'augmente ou le laisse invariant. Il en résulte que $e(A) \geq F + E - n$. Or, d'après la proposition 3.10, on a $e(A) \leq (i - d - 1)/2$, parce que A a $i - d$ feuilles. La proposition 3.7 résulte de ces deux inégalités. ■

6.4 Arbres bicolores

6.4.1 Présentation

Les arbres bicolores, dus à Guibas et Sedgewick, constituent une structure d'arbres binaires de recherche particulièrememt efficace. Traditionnellement appelés arbres rouge-noir, nous les colorons en blanc et noir pour des raisons évidentes. Ces arbres sont efficaces pour plusieurs raisons. C'est une structure souple car elle permet de réaliser «aisément» de nombreuses opérations telles que recherche, insertion, suppression, mais aussi des opérations plus sophistiquées comme la scission ou la concaténation.

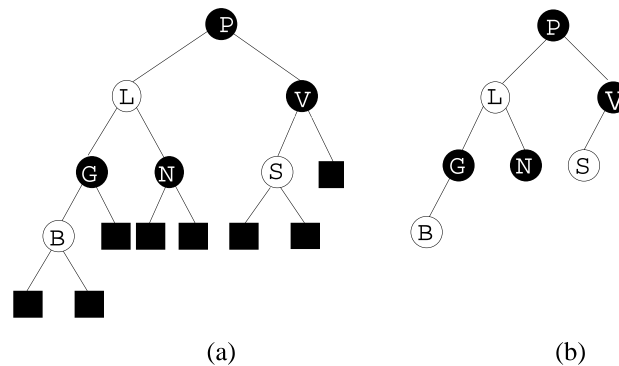

Figure 4.1: *Un arbre bicolore : (a) avec ses feuilles, (b) sans feuilles.*

Un *arbre bicolore* est un couple (A, c) où A est un arbre binaire complet et c est une application qui associe à chaque sommet une couleur (*blanc* ou *noir*) de façon que :

(a) toutes les feuilles sont noires ;
(b) la racine est noire ;
(c) le père d'un sommet blanc est noir ;
(d) les chemins issus d'un même sommet et se terminant en une feuille ont le même nombre de sommets noirs.

Un *arbre bicolore pour un ensemble ordonné* E est un arbre bicolore qui est un arbre de recherche pour E, avec la restriction que seuls les nœuds contiennent des éléments de E.

Notons que la condition (b) n'est pas toujours exigée, mais il est plus pratique pour la suite de ne considérer que des arbres bicolores ayant des racines noires.

Exemple. Dans la figure 4.1, les feuilles sont représentées par des carrés. On conviendra en général de ne pas les représenter. Ainsi l'arbre de la figure 4.1(a) devient celui de la figure 4.1(b).

Remarque. Notons que dans un arbre bicolore «effeuillé» , le grand-père d'un sommet est nécessairement un sommet complet, ainsi l'arbre, d'une certaine façon, est «presque» complet.

Nous allons donner immédiatement une propriété caractéristique des arbres bicolores, car selon les circonstances, la définition donnée ci-dessus ou la propriété ci-dessous sera plus adaptée.

Une *fonction rang* sur un arbre binaire complet A est une application rg de l'ensemble S des sommets de A dans l'ensemble $\mathbb{N}$ des entiers naturels vérifiant les conditions suivantes (p désignant la fonction père sur S) :

(i) pour toute feuille x, $rg(x) = 0$ et si x a un père $rg(p(x)) = 1$;
(ii) pour tout sommet x, $rg(x) \leq rg(p(x)) \leq rg(x) + 1$;
(iii) pour tout sommet x, $rg(x) < rg(p^2(x))$.

Proposition 4.1. *Tout arbre bicolore possède une fonction rang.*

Preuve. Soit (A, c) un arbre bicolore, et définissons le rang d'un sommet comme le nombre de sommets noirs situés sur un chemin quelconque allant de ce sommet (exclu) à une feuille de A. Notons que cette définition du rang est consistante en vertu de la condition (d) dans la définition d'un arbre bicolore . Démontrons que l'application rg que nous venons de définir satisfait les propriétés (i), (ii), (iii).

Par définition du rang d'un sommet on a immédiatement que (a)$\Longrightarrow$ (i). Il est clair aussi que $rg(x) \leq rg(p(x))$ pour tout sommet x, et que par ailleurs si x est blanc alors $rg(p(x)) = rg(x)$, et si x est noir alors $rg(p(x)) = rg(x) + 1$. Donc $rg(p(x)) \leq rg(x) + 1$. Ainsi (ii) est vérifié.

Reste à prouver (iii). On a $rg(x) \leq rg(p(x)) \leq rg(p^2(x))$ pour tout x. Supposons que $rg(x) = rg(p^2(x))$. Alors x est blanc ainsi que $p(x)$, ce qui contredit (c). Donc (c)$\Longrightarrow$(iii). ■

Proposition 4.2. *Si A est un arbre binaire possède une fonction rang, alors il existe une affectation de couleurs c aux sommets de A telle que (A, c) soit un arbre bicolore.*

Preuve. Réalisons l'affectation des couleurs aux sommets de A de la façon suivante : tout sommet ayant même rang que son père est déclaré blanc, les autres étant déclarés noirs (donc aussi la racine). Montrons que (A, c) est bicolore.

Par définition même de la fonction c, les conditions (a) et (b) sont immédiates. Vérifions (c). Soit x un sommet blanc, x n'est donc pas la racine, et son père $p(x)$ a même rang que lui. Si $p(x)$ était blanc, alors $p(x)$ aurait un père de même rang, ce qui contredirait (iii). Donc (iii) $\Longrightarrow$(c).

Il reste à prouver (d). Pour ce faire, on prouve par induction sur la hauteur du sommet x la propriété (d') : tous les chemins issus de x et menant à une feuille ont même nombre de sommets noirs (x non compris) et ce nombre est égal à $rg(x)$.

Si x est une feuille, (d') est vraie pour x. Soit x un nœud qui n'est pas une feuille, alors par hypothèse d'induction, (d') est vraie pour les fils y et z de x. Il y a trois cas à envisager :

(α) $rg(x) = rg(y) = rg(z)$
(β) $rg(x) = rg(y) + 1 = rg(z) + 1$
(γ) $rg(x) = rg(y) + 1 = rg(z)$.
Sur la figure 4.2 on a représenté les trois cas, le sommet dont la couleur n'est pas connue est représenté par deux demi-disques de couleurs différentes.

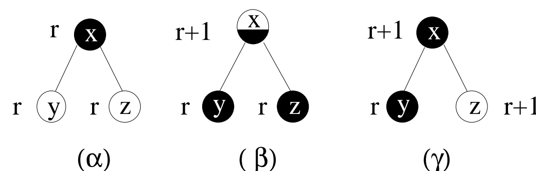

Figure 4.2: *Affectation d'une couleur à un nœud en fonction du rang.*

Dans le cas (α), y et z sont blancs, et puisque y et z vérifient (d'), x vérifie (d').

Dans le cas (β), y et z sont noirs, donc pour les mêmes raisons qu'en (α), x vérifie (d').

Dans le cas (γ), y est noir et z est blanc. Mais puisque y et z vérifient (d') et que $rg(y) + 1 = rg(z)$, tous les chemins issus de x et allant à une feuille ont le même nombre de sommets noirs (x non compris) et ce nombre est égal à $rg(x)$, donc x vérifie (d'). ∎

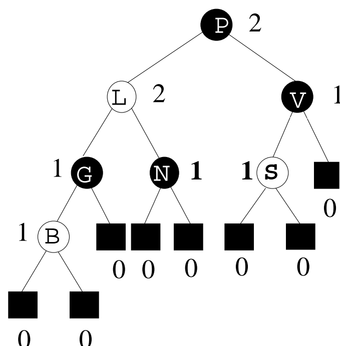

Figure 4.3: *Attribution d'un rang aux sommets de l'arbre de la première figure.*

Nous avons représenté sur la figure 4.3 les rangs des sommets de l'arbre bicolore de la figure 4.1. On laisse au lecteur le soin de vérifier en exercice que la correspondance que l'on vient d'établir entre les arbres bicolores et les arbres binaires ayant une fonction rang est une bijection, à savoir que (A, c) étant donné, la seule fonction rg sur A qui vérifie (i), (ii), (iii) est celle que nous avons définie, et de même pour la réciproque, la seule coloration possible est celle qui a été donnée.

6.4.2 Hauteur d'un arbre bicolore

L'efficacité des opérations élémentaires, en particulier de la recherche, dépend essentiellement de la hauteur de l'arbre binaire de recherche. C'est pourquoi il est

nécessaire d'évaluer de façon précise la hauteur d'un arbre bicolore en fonction du nombre de ses sommets.

Lemme 4.3. *Soit A un arbre bicolore à n sommets, et soit x un sommet dans A de rang k et de hauteur h. Alors $h \leq 2k$ et $n \geq 2^k$; de plus, si x est blanc, les deux inégalités sont strictes.*

Preuve. Par induction sur la hauteur du nœud x. Si x est une feuille, la propriété est trivialement vraie car x est noir et k est nul. Supposons que la hauteur h de x soit strictement positive. Soient y et z les fils de x. Les trois cas possibles aux symétries près sont représentés sur la figure 4.4. On notera $h(x)$ et $n(x)$ respectivement la hauteur et le nombre de sommets du sous-arbre de A de racine x.

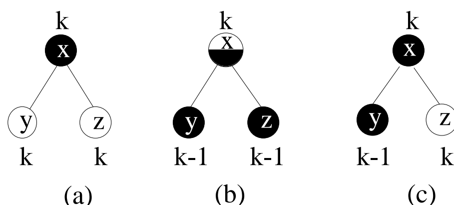

Figure 4.4: *Les trois cas possibles.*

Cas (a) Ici, x est noir et y et z sont blancs. Par hypothèse d'induction, on a : $h(y) < 2k$, $h(z) < 2k$, $n(y) > 2^k$ et $n(z) > 2^k$. Donc $h(x) \leq 2k$ et $n(x) = 1+n(y)+n(z) \geq 2^k$. Or x est noir et de rang k. Les inégalités sont donc correctes.

Cas (b) Les fils y et z sont noirs et x est blanc ou noir. Par induction, on a $h(y) \leq 2(k-1)$, $h(z) \leq 2(k-1)$, $n(y) \geq 2^{k-1}$ et $n(y) \geq 2^{k-1}$. Donc $h(x) \leq 2(k-1)+1 < 2k$ et $n(x) \geq 2^k + 1 > 2^k$. Comme x est blanc ou noir, le lemme est vérifié dans ce cas. Reste le dernier cas :

Cas (c) Un des fils, disons y est noir et l'autre blanc et le père x est noir. On a alors $h(y) \leq 2(k-1)$, $h(z) < 2k$, $n(y) \geq 2^{k-1}$ et $n(z) > 2^k$. Donc $h(x) \leq 2k$ et $n(x) > 2^{k-1} + 2^k + 1 > 2^k$. Comme x est noir, le lemme est vérifié. ∎

Proposition 4.4. *Un arbre bicolore ayant n sommets ($n > 0$) et de hauteur h vérifie :*

$$\lfloor \log n \rfloor \leq h \leq 2\lfloor \log n \rfloor$$

Preuve. La preuve découle directement du lemme précédent et du fait qu'un arbre binaire à $n > 0$ sommets est de hauteur au moins $\lfloor \log n \rfloor$. ∎

Corollaire 4.5. *La recherche d'un élément dans un arbre bicolore prend un temps $O(\log n)$ où n est le nombre de sommets de l'arbre.*

6.4.3 Insertion dans un arbre bicolore

Soit m un nouvel élément à insérer dans un arbre bicolore A. Comme dans un arbre binaire de recherche ordinaire, on descend dans l'arbre en partant de la racine. On crée un nouveau sommet blanc x contenant m dont les fils sont des feuilles noires. Soit A' l'arbre obtenu ; cet arbre n'est plus nécessairement un arbre bicolore (figure 4.5).

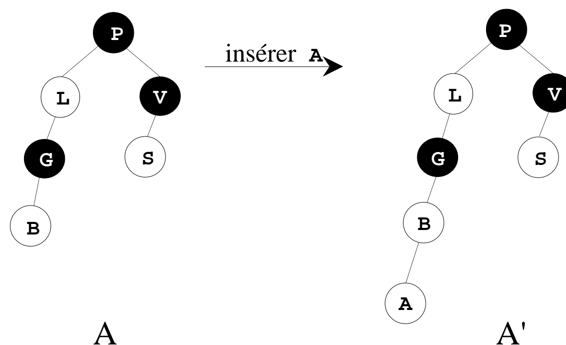

Figure 4.5: *Première phase de l'insertion de* A *dans A.*

En fait, seules les propriétés (b) ou (c) de la définition peuvent ne plus être respectées dans A', à savoir : la racine est blanche (parce que x est la racine) ou bien : un sommet et son père sont blancs (parce que le père de x est blanc).

Procédure de rééquilibrage

Nous allons donner 3 règles (et leurs symétriques) qui permettent de rééquilibrer l'arbre après insertion.

Si A était vide avant l'insertion (cas où (b) n'est pas respecté), il suffit de colorer x en noir, et c'est terminé (règle (γ) de la figure 4.6).

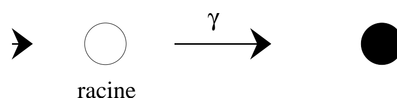

Figure 4.6: *Rééquilibrage après une insertion : règle γ.*

Sinon, le père y de x est blanc, donc y n'est pas la racine et son père z est noir. Supposons, pour fixer les idées, que y est fils gauche de z, et notons t le fils droit de z. Appliquons l'algorithme suivant, où x est le sommet courant qui au départ est le sommet nouvellement créé (le sommet courant est indiqué par une flèche sur la figure) :

• si y a un frère blanc on applique α_1 ou α_2 selon «l'éloignement» de x et t, i.e. selon que x est un fils gauche ou droit (si y est un fils droit, il faut prendre la règle

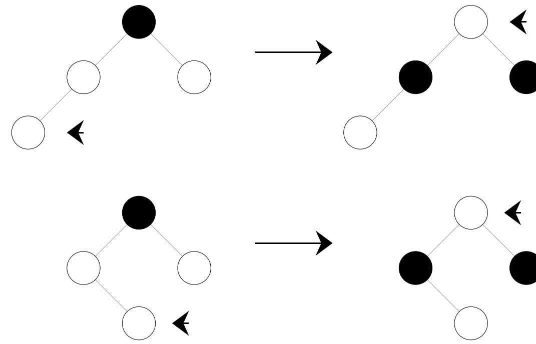

Figure 4.7: *Rééquilibrage après une insertion : règles α.*

symétrique qui revient à considérer effectivement l'éloignement des deux sommets x et t) ; le sommet courant devient z (α_1 ou α_2 ne sont que des changements de couleur). Alors si l'arbre obtenu n'est pas bicolore, c'est qu'à nouveau l'une des propriétés (b) ou (c) n'est pas respectée au sommet z, on itère donc le processus.

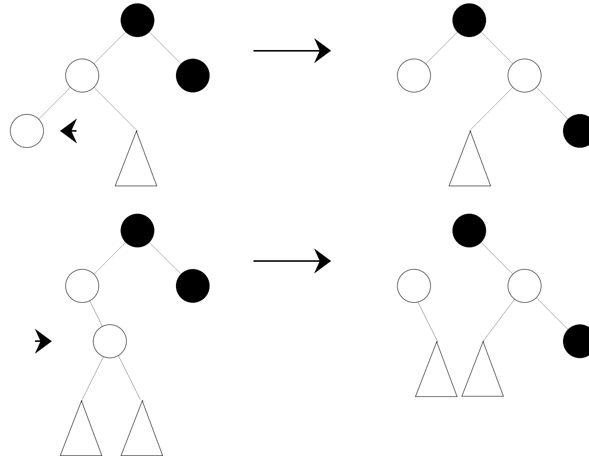

Figure 4.8: *Rééquilibrage après une insertion : règles β.*

• si y a un frère noir t, on fait une rotation simple (β_1) dans le cas où x et t sont «éloignés» et une rotation double (β_2) dans le cas où ils sont proches ; dans les deux cas, l'arbre obtenu est bicolore et le processus de rééquilibrage est terminé.

Puisque les règles α diminuent de deux la profondeur du sommet courant et que les règles γ et β sont des règles d'arrêt, il est clair que l'itération se termine et fournit un arbre de recherche bicolore.

Proposition 4.6. *Une insertion dans un arbre bicolore à n sommets prend un temps $O(\log n)$ et nécessite au plus deux rotations.*

Preuve. La descente avant insertion prend un temps $O(\log n)$. Puis dans la procédure de rééquilibrage, à chaque opération élémentaire, le sommet courant

a une profondeur qui diminue strictement, sauf en cas de règle d'arrêt, donc le nombre total d'opérations élémentaires est $O(\log n)$, par ailleurs chaque opération élémentaire prend un temps $O(1)$, d'où la première assertion. Par ailleurs, les règles nécessitant des rotations sont terminales, il s'agit de (β_1) (une rotation) et (β_2) (une rotation double) donc le résultat est démontré. ∎

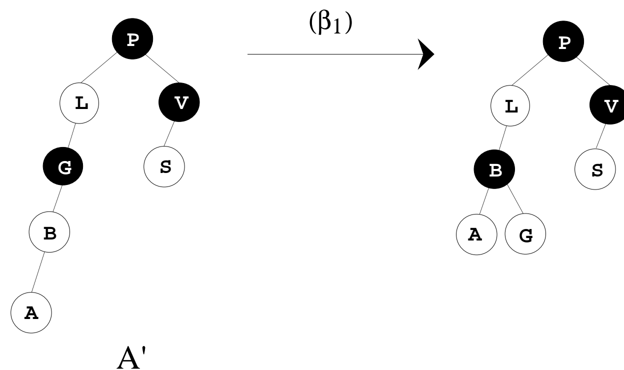

Figure 4.9: *Rééquilibrage après insertion de* A.

6.4.4 Suppression d'un élément dans un arbre bicolore

Considérons maintenant le problème de la suppression. On procède en deux étapes. La *première phase* consiste à appliquer l'algorithme de suppression d'un élément dans un arbre de recherche binaire classique. En descendant dans l'arbre on trouve le sommet x dans lequel est rangé l'élément m que l'on veut supprimer.

• si le fils droit de x est une feuille, alors on supprime x et on le remplace par son sous-arbre gauche de racine y; notons que ce sous-arbre gauche, du fait qu'on a affaire à un arbre bicolore, est soit une feuille, soit un arbre de hauteur 1 dont la racine est blanche et dont les fils sont des feuilles;

– si x était blanc, l'arbre reste un arbre bicolore et on peut remarquer que dans ce cas y est une feuille (figure 4.10 (a));

– si x était noir, il y a deux cas à envisager selon la couleur du sommet y qui a remplacé x :

– si y est blanc, ses fils sont des feuilles, on colore y en noir (figure 4.10 (b)) et l'algorithme est terminé;

– si y est noir, y est une feuille et y devient « *dégradé* » ce que l'on note sur la figure par un signe $-$, nous expliquerons dans un instant ce que cela signifie précisément (figure 4.10 (c));

• si le fils droit de x n'est pas une feuille, alors on recherche le sommet y contenant le prédécesseur l de m; ce sommet y a pour fils droit une feuille; on remplace m par l dans x et on supprime le sommet y par l'algorithme ci-dessus.

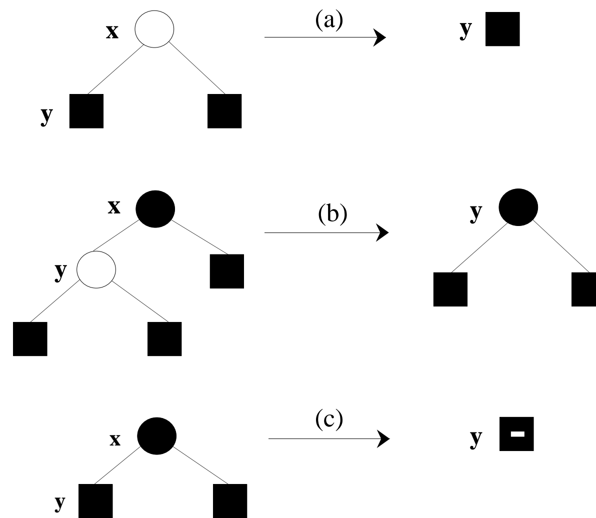

Figure 4.10: *Première phase de suppression d'un élément dans un arbre bicolore .*

Précisons la signification d'un sommet *dégradé* : lorsqu'un sommet x est dégradé, cela signifie que le nombre de sommets noirs sur un chemin de x à une feuille quelconque, x **compris**, est inférieur d'une unité au nombre de sommets noirs sur un chemin de y (frère de x) à une feuille quelconque, y **compris**, et donc le père de x, s'il existe, ne respecte pas la condition (d) de la définition des arbres bicolores.

La *deuxième phase* consiste à rééquilibrer l'arbre dans le cas où il contient un sommet dégradé. Les schémas des figures 4.11 et 4.12 donnent l'algorithme à appliquer.

Le sommet courant x est le sommet dégradé, il est indiqué par une flèche et on suppose que si x a un père y alors x est un fils gauche, et on appelle z son frère (on obtient toutes les règles par symétrie, on notera par des lettres primées les règles symétriques des règles indiquées). Plusieurs remarques sont à faire pour justifier la validité de l'algorithme. Tout d'abord, si z existe, alors, par définition d'un sommet dégradé, puisque x est dégradé, z ne peut être une feuille, donc z a deux fils. Ensuite, on peut noter que tous les cas sont examinés :

– soit x est la racine (règle (a_1)) ;

– soit x a un frère noir z qui a lui même deux fils noirs (règles (a_2) ou (a_3) selon que le père y est noir ou blanc) ;

– soit x a un frère noir z dont l'un des deux fils est blanc (règles (b_2) ou (b_3) selon qu'il s'agit d'un fils gauche ou droit ; notons que si les deux fils de z sont blancs on peut appliquer l'une ou l'autre règle) ;

– soit x a un frère blanc (règle (b_1)).

On peut noter que la profondeur du sommet courant diminue strictement à chaque application d'une règle sauf pour la règle (b_1), mais cette règle est suivie de

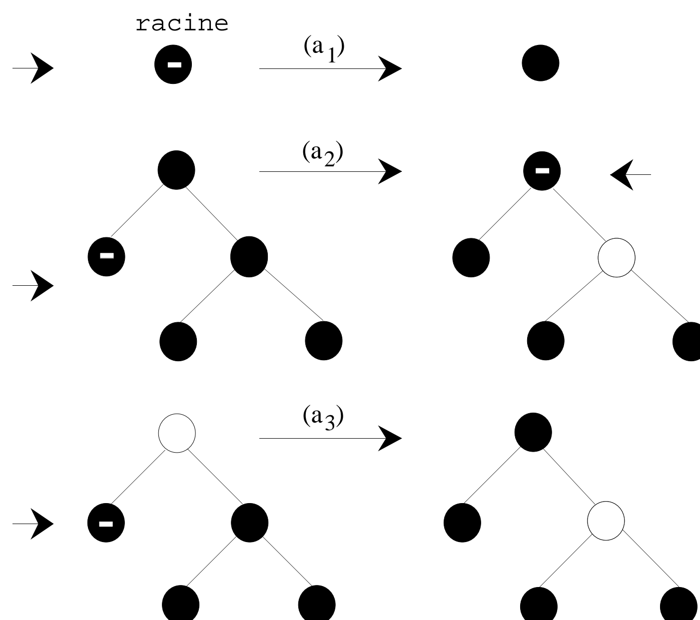

Figure 4.11: *Rééquilibrage après une suppression : règles* $(a_1), (a_2), (a_3)$.

l'application d'une règle (a_3), (b_2) ou (b_3) qui sont des règles d'arrêt. Donc le nombre de règles appliquées est au plus $O(\log n)$ et chacune prend un temps $O(1)$. Enfin, il est facile de vérifier que les règles appliquées sont correctes, i.e. que si le sommet courant est dégradé avant application de (a_2) ou (b_1), alors le sommet courant après application de la règle est bien un sommet dégradé , et que, après application de l'une des autres règles, qui sont des règles d'arrêt, l'arbre obtenu est un arbre bicolore.

Proposition 4.7. *Une suppression dans un arbre bicolore à n sommets ($n > 0$) prend un temps $O(\log n)$ et nécessite au plus trois rotations.*

Preuve. La première assertion découle immédiatement des remarques ci-dessus, chaque phase prenant un temps $O(\log n)$. Quant aux rotations, elles apparaissent dans les règles (b_1) (une rotation), (b_2) (une rotation) et (b_3)) (une rotation double). Or la règle (b_1) est suivie de l'une des règles $(a_3), (b_2), (b_3)$, qui sont toutes les trois terminales, la preuve est donc terminée. ∎

Exemple. Dans l'exemple de la figure 4.13, on supprime le contenu p de la racine.

La première phase consiste à mettre dans la racine la plus grande clé du sous-arbre gauche, à savoir n, et à supprimer le sommet qui contient cette clé, ce qui dégrade une feuille. On doit alors procéder au rééquilibrage de l'arbre. On applique alors la règle (b'_1), puis la règle (b'_3).

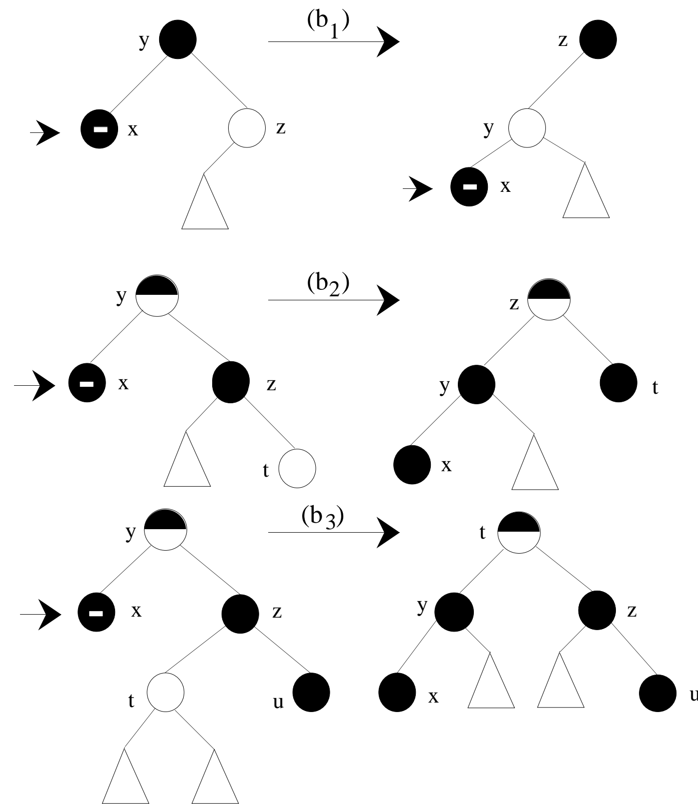

Figure 4.12: *Rééquilibrage après une suppression : règles* $(b_1), (b_2), (b_3)$.

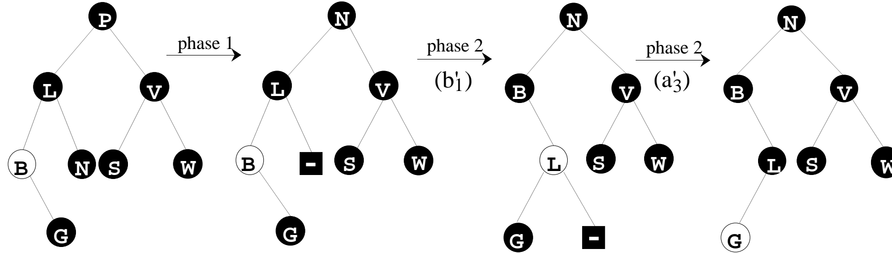

Figure 4.13: *Suppression de* p.

6.5 Enrichissement

Dans les sections précédentes, nous avons décrit trois familles d'arbres équilibrés qui implémentent, de façon efficace, les opérations d'un dictionnaire (recherche, insertion et suppression). Nous avons également constaté que la scission et la concaténation pouvaient être réalisées en temps logarithmique.

Dans cette section, nous montrons comment implémenter d'autres opérations, comme la recherche du k-ième élément, ou du médian, en temps logarithmique. Pour cela, il convient d'augmenter la structure (de l'*enrichir*) par quelques données additionnelles aux sommets. Le but de cette section est de montrer qu'un

arbre équilibré enrichi de cette manière possède à la fois la souplesse d'un arbre et d'une liste chaînée.

Père

Soit A un arbre, et considérons la fonction PÈRE(x : sommet; A : arbre) qui donne le père de x dans A, et qui est indéfini si x est la racine de A. Le calcul de cette fonction n'est pas facile si l'on ne dispose pas d'un champ supplémentaire dans les sommets. Enrichissons donc la structure en associant, à chaque sommet, un pointeur supplémentaire qui donne son père (ce pointeur vaut NIL pour la racine). La valeur de ce pointeur est facile à maintenir lors d'une insertion ou d'une suppression d'un sommet. On se convainc aisément que la mise à jour de ces pointeurs, lors d'une rotation ou d'une double rotation, se fait en temps constant.

Voisins

Soit A un arbre binaire de recherche pour un ensemble de clés S. Le parcours en ordre symétrique (voir chapitre 4) de l'arbre fournit ses clés en ordre croissant. On peut donc définir les fonctions SUIVANT et PRÉCÉDENT sur les sommets de A, comme dans une liste :

SUIVANT(x : sommet; A : arbre) : sommet;
: Donne le sommet contenant la clé qui suit la clé de x; indéfini si la clé de x est la plus grande clé dans A.

PRÉCÉDENT(x : sommet; A : arbre) : sommet;
: Donne le sommet contenant la clé qui précède la clé de x; indéfini si la clé de x est la plus petite clé dans A.

La réalisation en temps *constant* de ces deux opérations sur un arbre balisé (qu'il soit binaire, ou *a*–*b*) est particulièrement facile. On munit pour cela chaque feuille de deux pointeurs supplémentaires, l'un pointant vers son voisin de gauche (dont la clé est justement la clé précédente), l'autre vers son voisin de droite. Cela revient à plaquer, sur la suite des feuilles, une structure de liste doublement chaînée. La gestion de ces pointeurs, lors d'une insertion ou d'une suppression, se fait comme pour les listes, et le surcoût par opération est donc constant. Dans le cas particulier des arbres *a*–*b* (et plus généralement des arbres dont toutes les feuilles sont à la même profondeur), on peut continuer cette construction et créer une liste doublement chaînée non seulement aux feuilles, mais à tous les niveaux de l'arbre. La structure qui en résulte est un arbre à *liaisons par niveau* («level-linked tree»).

Ces arbres ont des propriétés remarquables; en particulier, la recherche d'une feuille qui se trouve à distance d d'une feuille donnée peut se faire en temps $O(\log d)$ (exercice).

Lorsque l'arbre n'est pas balisé, donc lorsque les informations se trouvent sur les sommets, la même structure de liste doublement chaînée est utilisée (voir figure

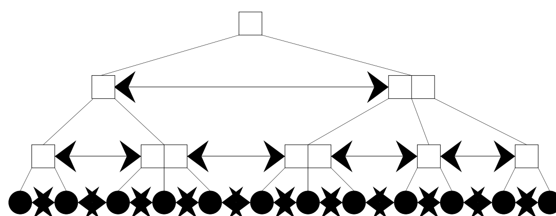

Figure 5.1: *Un arbre 2–3 à liaisons par niveau.*

5.2). On se convainc facilement que les rotations et doubles rotations laissent cette liste inchangée. En ce qui concerne la suppression d'une clé, on est amené, rappelons-le, à supprimer une feuille. C'est cette feuille qui est aussi supprimée dans la liste.

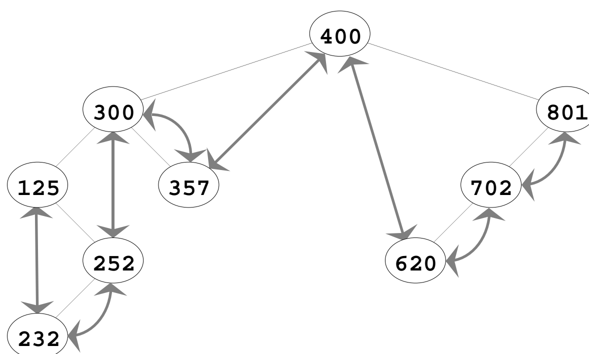

Figure 5.2: *Une liste doublement chaînée sur les sommets d'un arbre.*

Proposition 5.1. *Dans un arbre de recherche équilibré, on peut, moyennant des pointeurs supplémentaires, implémenter les opérations* SUIVANT *et* PRÉCÉDENT *en temps constant.* ∎

Numéro d'ordre

Le *numéro d'ordre* ou le rang d'une clé dans un ensemble S est le nombre de clés qui lui sont inférieures ou égales. C'est le numéro que reçoit la clé quand elles sont numérotées, à partir de 1, en ordre croissant.

Considérons le problème de la recherche de la clé de numéro d'ordre k dans un ensemble représenté par un arbre binaire de recherche A. Bien entendu, il ne s'agit pas de faire un parcours symétrique de l'arbre. Dotons chaque sommet x d'un champ supplémentaire dont la valeur $n(x)$ est le nombre de sommets dans le sous-arbre $A(x)$ de racine x (la *taille*). En particulier, pour la racine r, le nombre $n(r)$ est le nombre de sommets de A. Notons $\sigma(x)$ le numéro d'ordre de x dans la numérotation infixe. Posons

$$n_g(x) = \begin{cases} n(y) & \text{si } y \text{ est le fils gauche de } x\,; \\ 0 & \text{si } x \text{ n'a pas de fils gauche.} \end{cases}$$

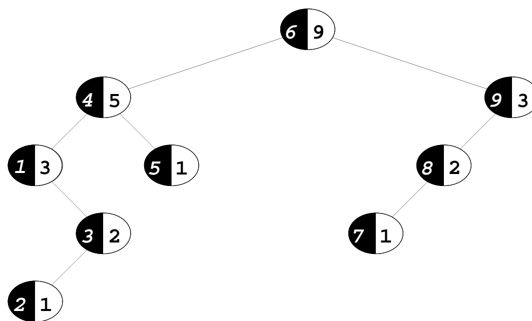

Figure 5.3: *Les numéros d'ordre (en blanc) et les tailles (en noir).*

et de même pour n_d. On cherche le k-ième sommet comme suit : on compare k au nombre $m = 1 + n_g(r)$, où r est la racine de l'arbre. L'entier m est le numéro d'ordre de la racine. Si $k < m$, le sommet cherché est dans le sous-arbre gauche ; si $k > m$, il est dans le sous-arbre droit, et il est le sommet de numéro $k - m$ dans la numérotation de ce sous-arbre. D'où la procédure :

```
procédure SOMMET-D'ORDRE(k, A);
  m := 1 + n_g(RACINE(A));
  si m = k alors retourner RACINE(A)
  sinon
    si k < m alors retourner SOMMET-D'ORDRE(k, A_g)
    sinon retourner SOMMET-D'ORDRE(m − k, A_d)
  finsi.
```

Il reste à étudier les répercussions des opérations d'insertion, de suppression, et de rééquilibrage. Considérons par exemple les arbres *AVL*. Une insertion ou suppression doit être suivie d'une remontée vers la racine, pour ajuster les tailles $n(x)$ sur chaque nœud x du chemin entre la racine et la feuille ajoutée ou supprimée. Quant aux rotations et doubles rotations, l'ajustement se réduit à une ou deux additions ou soustractions.

Dans le cas d'un arbre *a–b*, on procède essentiellement de la même manière : en plus des balises, on range dans chaque nœud le nombre de feuilles dans le sous-arbre repéré par la balise. On procède ensuite comme pour les arbres binaires. On a donc :

Proposition 5.2. *Dans un arbre de recherche on peut, moyennant un champ supplémentaire, trouver la clé de numéro d'ordre donné en temps logarithmique en fonction du nombre de clés présentes.* ∎

Médian

Etant donné un ensemble de n clés S, on cherche à le partager en deux parties S_1 et S_2 de même taille (à 1 près) telles que les clés de S_1 soient inférieures aux clés de S_2. Pour cela, il suffit de connaître la clé *médiane*, c'est-à-dire la clé dont le numéro d'ordre est $\lfloor n/2 \rfloor$, puis d'appliquer un algorithme de scission. Si, pour chaque sommet, on connaît le nombre de sommets de son sous-arbre, on peut calculer le numéro d'ordre de la clé médiane en temps constant. En temps logarithmique, on repère ensuite le sommet médian, puis à nouveau en temps logarithmique, on effectue la scission. On a donc

Corollaire 5.3. *Dans un arbre a–b, on peut effectuer la recherche du médian en temps logarithmique, donc aussi scinder un arbre en deux parts égales (à* 1 *près) en temps logarithmique.* ∎

6.6 Arbres persistants

6.6.1 Ensembles ordonnés persistants

Le problème à résoudre est le suivant : il s'agit de gérer un ensemble totalement ordonné E qui évolue dans le temps en ayant pour souci de ne pas perdre d'information sur les états passés de cet ensemble ordonné. En particulier, on veut être en mesure de savoir si à un instant t passé l'ensemble E possédait un élément donné x. On peut prendre connaissance du passé bien sûr, mais on ne peut pas modifier ce passé, par contre on peut modifier l'état présent de l'ensemble c'est-à-dire sa version la plus récente.

Nous formaliserons le problème de la façon suivante : le temps ou plutôt une *chronologie* est représentée par une suite $T = (t_0, \cdots, t_m)$ strictement croissante de nombres réels appelés *instants*. A l'instant t, on peut effectuer deux types d'actions sur l'ensemble E qui consistent à ajouter ou supprimer un élément dans E :

INSÉRER(p, E) :
: consiste à insérer l'élément p dans l'ensemble ordonné E (ne peut être réalisé que si p n'y est pas déjà) ;

SUPPRIMER(p, E) :
: consiste à supprimer l'élément p de E (ne peut être réalisé que si p figure dans E).

Etant donnée une chronologie $T = (t_0, \cdots, t_m)$, et une suite d'actions $h = (a_0, \ldots, a_m)$, on appelle *ensemble ordonné persistant E, d'histoire h et de chronologie* T la suite d'ensembles $(E_0, E_1, \ldots, E_m)$ telle que $E_0 = \emptyset$ et pour tout $i < m$, l'ensemble E_{i+1} est obtenu en exécutant l'action a_i sur l'ensemble E_i (la suite d'actions doit bien sûr être exécutable). L'entier m est la longueur de l'histoire

h. L'instant *présent* est l'instant t_m. Pour alléger les notations, on note encore E au lieu de E_m «l'état» de l'ensemble ordonné à l'instant t_m.

A cet instant présent, on peut poser des questions sur le passé ou avancer dans la chronologie, c'est à dire faire une nouvelle mise à jour de E. Les questions sur le passé sont de la forme suivante :

CHERCHER(a, E, t_j) :
: retourne l'élément de clé a contenu dans E_j à l'instant $t_j, j \leq m$ s'il y figure, un message d'échec sinon.

En revanche, la mise à jour de E, c'est-à-dire l'insertion ou la supprssion d'un élément, se fait en ajoutant :

- un nouvel instant t_{m+1} à la chronologie T ;
- une nouvelle action a_{m+1} à l'histoire h.

La structure de données que nous allons présenter pour implémenter ces opérations est due à N. Sarnak et R. Tarjan. Ses performances sont les suivantes :

Si l'histoire est de longueur m, et si E_i contient à chaque instant au plus n éléments (notons que nécessairement $n \leq m$), alors la recherche se fait en temps $O(\log m)$, la mise à jour de E en temps et en espace $O(\log n)$. Nous donnerons ensuite une version améliorée de l'algorithme qui garde la même complexité en temps, et a une complexité amortie linéaire en fonction de m en espace, c'est-à-dire que la complexité amortie en espace d'une mise à jour est $O(1)$.

Nous nous plaçons dans le cadre naturel où l'ensemble E_i est représenté par un arbre B_i. L'idée la plus simple qui vient à l'esprit pour la mise à jour à l'instant t_{m+1} est de recopier l'arbre binaire B_m avant de le transformer en arbre B_{m+1} qui représente l'ensemble E à l'instant t_{m+1}.

On réalise alors CHERCHER(x, E, t_i) comme suit : grâce à un tableau de pointeurs (figure 6.1) sur les racines des arbres B_i on peut (par une recherche dichotomique) trouver en temps $O(\log m)$ le pointeur sur l'arbre B_i, puis en temps $O(\log n)$ trouver si l'élément x figure dans B_i.

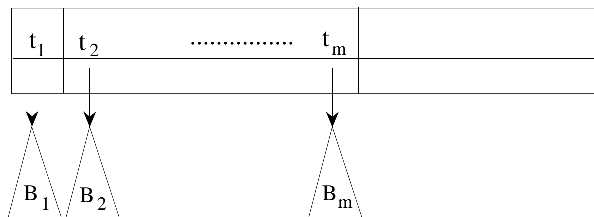

Figure 6.1: *Persistance par recopie intégrale.*

Bien sûr, la mise à jour prend sous cette forme un temps $O(n)$ pour la recopie de l'arbre, et un espace $O(n)$, donc ne répond pas aux exigences formulées plus haut.

Notons ici que si la chronologie est représentée par des entiers consécutifs (ce qui est fréquent) la recherche de l'arbre B_i peut se faire en temps $O(1)$ si on utilise un tableau de pointeurs sur les arbres B_i (figure 6.1).

Une autre idée, opposée à la première, est de ne jamais recopier un sommet. A chaque mise à jour de la structure, un sommet concerné reçoit l'information nécessaire (changement de clé, de fils) pour représenter la nouvelle version. Un sommet grossit alors, à chaque modification qui le concerne, d'un nombre constant de cellules, les sommets n'étant pas concernés par la modification restant inchangés.

L'inconvénient majeur de cette technique réside dans le temps considérable que prend la recherche. En effet, à chaque sommet, il convient de parcourir la liste des versions pour déterminer les informations relevantes à la version considérée, et le temps de ce parcours n'est pas indépendant du nombre de versions existantes.

Nous allons présenter ici deux méthodes efficaces pour représenter la liste des versions ; la première est inspirée de la méthode de recopie, mais à chaque mise à jour, on ne recopie que les sommets sur le chemin de la racine du sommet concerné. La deuxième est inspirée de la méthode du grossissement des sommets. Toutefois, on se limite à un grossissement borné, et en cas de débordement, on dédouble les sommets.

Les deux méthodes sont intéressantes et efficaces, la première plus simple, et la deuxième plus économique en place.

6.6.2 Duplication de chemins

Pour gagner du temps et de l'espace dans la mise à jour, on observe que l'arbre B_{i+1} diffère peu de l'arbre B_i ; on ne recopie que ce qui est nécessaire, ainsi B_{i+1} et B_i vont « partager » une structure commune. C'est la technique que nous allons exposer maintenant et que l'on cite habituellement comme la méthode de la *duplication de chemins*. Nous la développons avec des arbres bicolores mais elle peut s'effectuer sur les autres arbres binaires de recherche équilibrés.

La stratégie consiste à dupliquer tout sommet dont le contenu a été modifié ou dont un des fils a été dupliqué (ou modifié). Ainsi l'ensemble des sommets dupliqués constitue effectivement un ensemble de chemins partant de la racine de la nouvelle version. Notons que les couleurs des sommets sont seulement utiles pour la mise à jour de l'arbre et donc ne concernent que la dernière mise à jour, ainsi il n'est pas nécessaire de dupliquer un sommet dont on change simplement la couleur.

La règle de duplication est donc la suivante :

Règle de duplication. *Dupliquer tout sommet dont le contenu (qu'on appelle encore la clé) a été modifié. Dupliquer le père d'un sommet dupliqué. Le père*

dupliqué a deux pointeurs gauche et droit, un sur le fils dupliqué, l'autre sur le fils qui n'a pas été dupliqué.

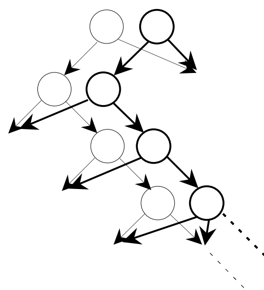

Figure 6.2: *Duplication de chemin.*

La duplication du père d'un sommet dupliqué se répercute donc jusqu'à la racine. Le but de cette règle est de ne pas modifier B_m, sauf éventuellement les couleurs des sommets, mais celles-ci sont désormais inutiles; en partant du pointeur associé à l'instant t_m, on retrouve toujours le même arbre. Examinons en détail comment on procède à une insertion ou une suppression sur la version courante (c'est-à-dire la dernière version) de E. Mais auparavant, pour comprendre pourquoi le mécanisme de rééquilibrage est plus complexe dans le cas d'une suppression que dans celui d'une insertion, il faut observer quels sont les pointeurs modifiés dans une opération de rotation.

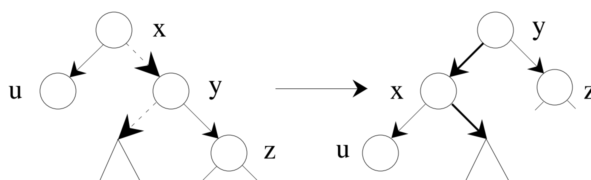

Figure 6.3: *Modifications de pointeurs dans une rotation gauche.*

La figure 6.3 montre à gauche en pointillés les pointeurs qui disparaissent, et à droite en gras les pointeurs nouvellement créés. Si l'on se reporte alors aux règles de rééquilibrage dans les arbres bicolores, on observe que dans le cas d'une insertion, les règles qui font intervenir des rotations (les règles β) ne modifient que des pointeurs issus de sommets qui, dans la structure persistante, sont des sommets déjà dupliqués, le rééquilibrage s'effectue donc «simplement». Par contre, pour ce qui est d'une suppression dans un arbre bicolore, les règles qui font intervenir des rotations (règles b_1, b_2, b_3) modifient des pointeurs de sommets non encore dupliqués, ces sommets doivent donc être dupliqués, nous verrons cela en détail un peu plus loin.

Cas d'une insertion

Soit B_m la dernière version de E et p l'élément à insérer à l'instant t_{m+1}. L'insertion se fait en deux phases.

Phase 1

La descente dans l'arbre B_m à partir de sa racine permet de créer une feuille x coloriée en rouge pour y placer p. Cette feuille appartient à B_{m+1} mais pas à B_m bien sûr. Puis on duplique tous les sommets situés sur le chemin allant de la racine à x non compris, chaque sommet dupliqué ayant un pointeur sur son fils situé dans B_m et un pointeur sur son fils nouvellement créé (figure 6.4).

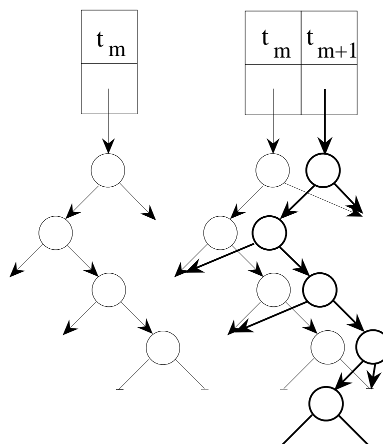

Figure 6.4: *Première phase de l'insertion.*

Expliquons l'exemple de la figure 6.5.

Nous nous autorisons ici à confondre les nœuds et leur contenu (car cela ne porte pas à conséquence), et nous parlerons d'ancien et de nouveau pour un sommet et son dupliqué. L'insertion de `J` modifie le fils droit de `I` (qui était vide). On duplique donc le fils droit de `I` en un sommet `J`, et l'on duplique tout le chemin de `J` à la racine. On a alors un nouveau chemin (`O`, `G`, `K`, `I`, `J`). Le nouvel `O` est racine de l'arbre B_{m+1} que l'on complète en donnant comme fils manquant à chaque nouveau sommet du chemin dupliqué, le fils correspondant de l'ancien sommet associé : par exemple le nouveau `K` a pour fils droit le fils droit de l'ancien `K`, i.e. `M`.

Phase 2

On a ainsi créé un nouvel arbre B_{m+1} dont la racine est la dupliquée de la racine de B_m. Il ne reste plus qu'à rééquilibrer l'arbre B_{m+1} en appliquant toujours la règle de duplication ci-dessus; mais il se trouve que dans le cas d'une insertion, le rééquilibrage ne nécessite pas de nouvelle duplication car les pointeurs modifiés dans les rotations sont issus de sommets déjà dupliqués, donc les pointeurs issus des nœuds de l'arbre B_m ne sont pas modifiés. On aura par exemple dans le cas

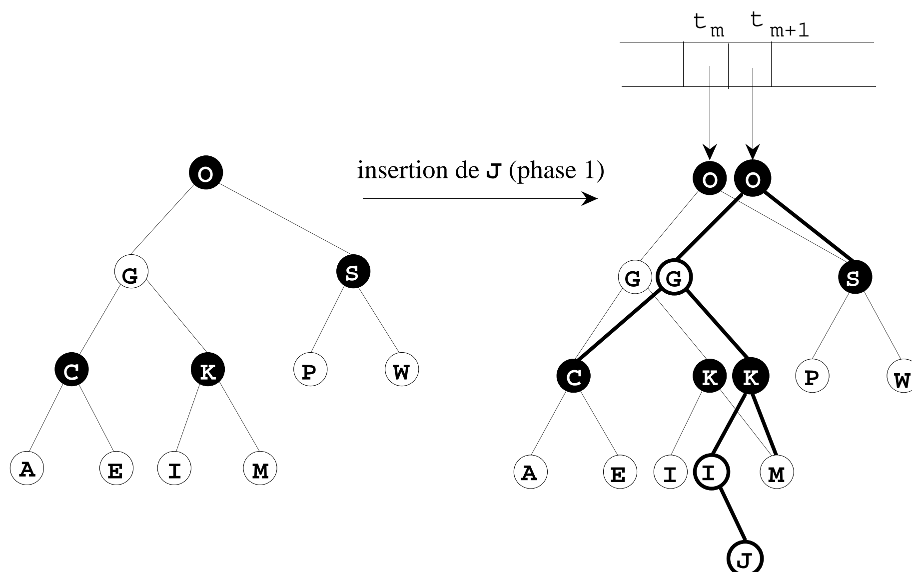

Figure 6.5: *Insertion d'un nouvel élément – phase* 1.

d'une rotation gauche le schéma suivant (les sommets dupliqués sont indiqués par des lettres primées) :

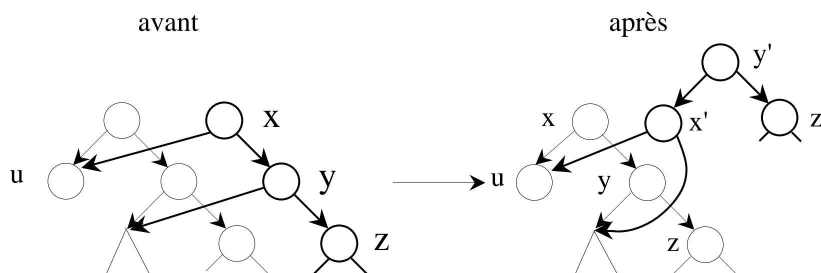

Figure 6.6: *Rééquilibrage par rotation gauche lors d'une insertion.*

La phase 2 consiste donc simplement en un rééquilibrage de l'arbre B_{m+1}. Notons que dans cette phase, la couleur d'un sommet commun à B_m et B_{m+1} peut être modifiée. Dans l'exemple donné (figure 6.7), la phase 2 consiste en un rééquilibrage de l'arbre B_{m+1} par application de la règle α_2 suivie de la règle β_2. La couleur de `M` a été modifiée, mais cela n'est pas gênant car la couleur des nœuds ne sert que dans la dernière version.

Cas d'une suppression

Soient q l'élément à supprimer dans la dernière version B_m de E et x le sommet contenant q.

Phase 1

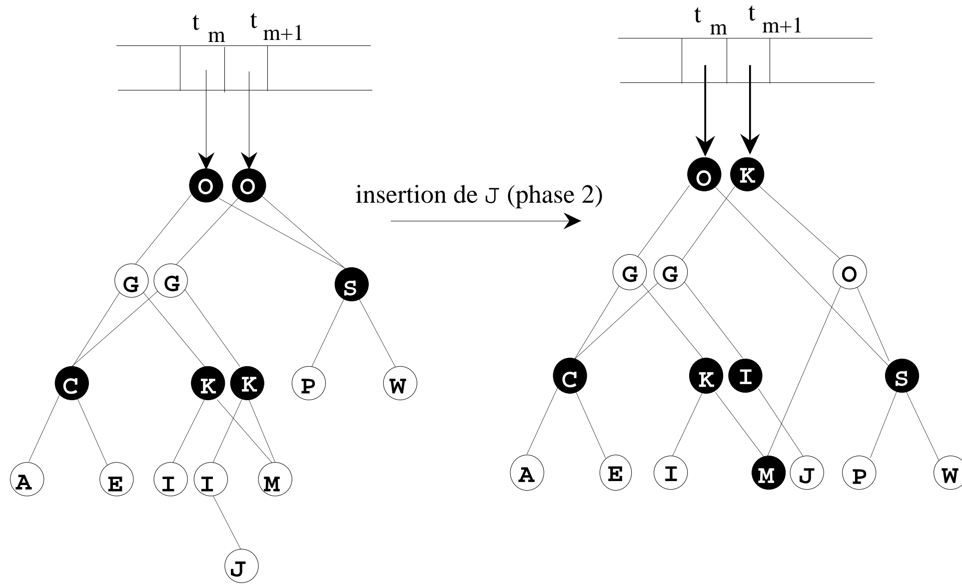

Figure 6.7: *Insertion d'un nouvel élément – phase* 2.

On applique dans un premier temps l'algorithme classique de suppression dans un arbre binaire de recherche.

Si x a un fils gauche, on détermine le sommet y contenant le prédécesseur de q, soit p. Ce sommet y n'a pas de fils droit autre qu'une feuille. On duplique tous les sommets allant de la racine à y non compris, le dupliqué de x contenant p, et chaque sommet dupliqué ayant un pointeur sur son fils qui est dans B_m et un pointeur sur son fils dupliqué. Soit z le père de y. Son dupliqué a un pointeur droit (figure 6.8 (a)) sur le fils gauche t de y si $z \neq x$, et un pointeur gauche (figure 6.8 (b)) sur le fils gauche t de y sinon. Si y est rouge, la couleur de t est inchangée, si y est noir alors si t est rouge il devient noir sinon il est *dégradé*.

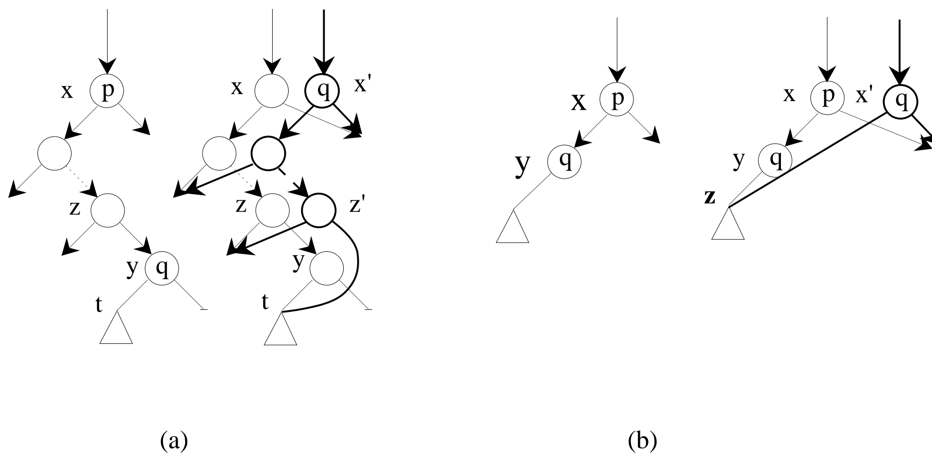

Figure 6.8: *Suppression d'un élément - phase 1.*

Si x a un fils droit, la procédure est symétrique de la précédente.

Si x n'a pas de fils (autre que des feuilles), on applique la procédure précédente où x joue le rôle de y. La figure 6.9 donne un exemple d'exécution de cette première phase.

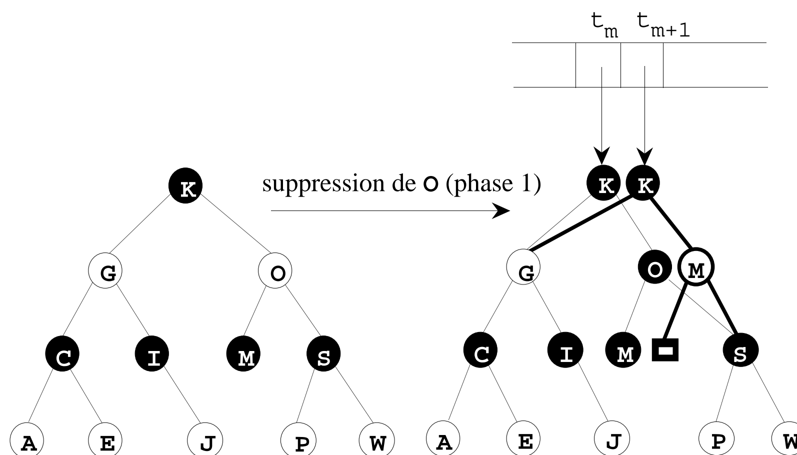

Figure 6.9: *Suppression de* O *- phase 1.*

Phase 2

Il reste à rééquilibrer l'arbre B_{m+1} obtenu dans la phase précédente. Mais les procédures de rééquilibrage des arbres bicolores utilisées dans la suppression modifient éventuellement des pointeurs sur des sommets de B_m qu'il faut donc dupliquer. Ainsi le rééquilibrage s'effectue selon les nouvelles règles décrites dans les figures 6.10 et 6.11.

On a représenté les sommets de B_m par un cercle et ceux de B_{m+1} qui sont des copies de sommets de B_m par un double cercle. Les nœuds affectés d'un signe $-$ sont les nœuds dégradés.

La deuxième phase de suppression appliquée à l'exemple de la figure 6.9 donne le résultat de la figure 6.12, après application de la règle (d).

Il est à noter que dans la structure décrite, chaque sommet a des pointeurs vers ses fils seulement et non vers ses pères. En effet, par nature même de l'arbre persistant, un sommet peut avoir plusieurs pères correspondant à des instants différents de l'histoire de l'ensemble persistant, ce qui rend l'implémentation de pointeurs vers les pères difficile. Donc, pour pouvoir réaliser les procédures de rééquilibrage, il est nécessaire d'avoir mémorisé le chemin dupliqué lors de la descente.

Théorème 6.1. *La structure d'arbre persistant représentant un ensemble E de taille au plus n et ayant une histoire de longueur m, utilisant la duplication de chemins permet d'effectuer chacune des opérations* Inserer(p, E) *et*

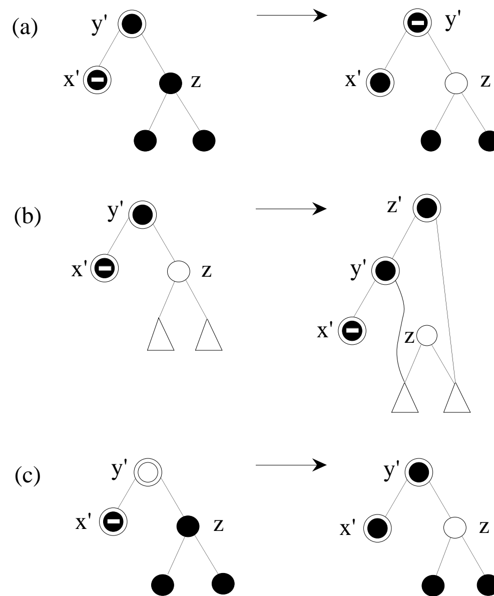

Figure 6.10: *Rééquilibrage dans une suppression, règles (a), (b), (c).*

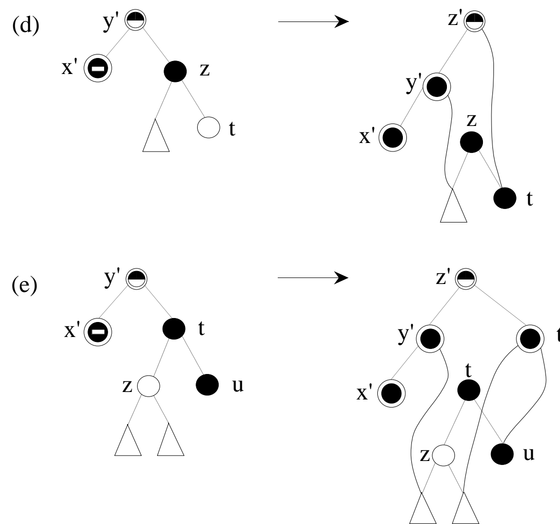

Figure 6.11: *Rééquilibrage dans une suppression, règles (d), (e).*

Supprimer(p, E) *en temps et espace* $O(\log n)$ *et l'opération* Chercher(p, E, t_i) *en temps* $O(\log m)$*. Si la chronologie est la suite* $(0, 1, \cdots, m)$ *alors* Chercher (p, E, t_i) *se calcule en temps* $O(\log n)$*.*

Preuve. Il est clair que l'insertion ou la suppression d'un élément dans l'arbre persistant prend un temps proportionnel à celui pris par une insertion ou une suppression dans un arbre bicolore ordinaire. Quant au nombre de sommets dupliqués, il est majoré par la hauteur de l'arbre B_m augmenté dans le cas d'une suppression du nombre de rotations qui est au plus trois. Le résultat est donc prouvé pour la première assertion. Pour ce qui est de la recherche, une première recherche

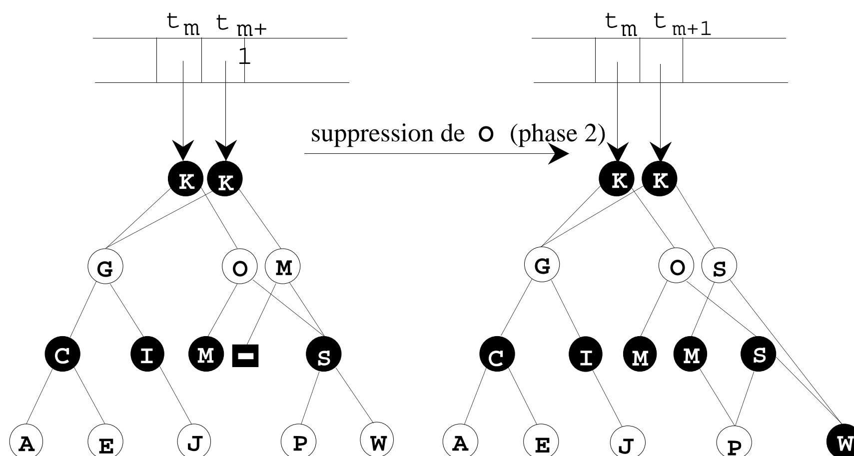

Figure 6.12: *Suppression d'un élément-phase 2 (règle (d)).*

dichotomique permet d'obtenir en temps $O(\log m)$ le pointeur sur l'arbre B_i dans lequel la recherche doit s'effectuer. Cette recherche prend alors un temps $O(\log n)$, mais $n \leq m$. Si les instants sont les entiers de 1 à m, alors on a accès en temps $O(1)$ à la racine de l'arbre B_i. ■

6.6.3 Méthode de duplication des sommets pleins

Nous allons voir maintenant comment rendre la structure performante en espace, c'est-à-dire linéaire en fonction du nombre de mises à jour.

Il suffit de remarquer que dans une mise à jour d'un arbre bicolore ordinaire, qui peut être une insertion ou une suppression, le nombre de nœuds pour lesquels le contenu est modifié (modifications autres que celle de la couleur) et le nombre de pointeurs modifiés est $O(1)$.

La solution consiste à munir chaque sommet d'un nombre fixe k de pointeurs supplémentaires «libres» qui vont être utilisés pour une nouvelle mise à jour et éviter des duplications. Ces pointeurs sont munis d'un champ contenant l'instant auquel ils sont activés, c'est à dire auquel ils cessent d'être libres, et d'un champ précisant s'il s'agit d'un pointeur gauche ou droit (notons qu'on peut se passer de ce dernier champ car la nature gauche ou droite du pointeur peut se déterminer en comparant les clés des sommets reliés par ce pointeur, mais ce qu'on gagne en place d'un côté est perdu en temps de l'autre). Lorsque tous les pointeurs libres d'un nœud sont activés (le sommet est alors «plein») et qu'un pointeur du nœud doit être modifié, ou lorsque le contenu du sommet lui-même est modifié, alors seulement le sommet est dupliqué en un nouveau sommet qui dispose à nouveau de k pointeurs libres.

Nous allons détailler la structure ainsi obtenue pour $k = 1$; elle ne diffère pas dans son principe du cas général. Nous renvoyons le lecteur intéressé par le problème

général aux notes de fin de chapitre.

Chaque sommet est donc muni en tout de trois pointeurs, possédant un champ qui indique s'il s'agit d'un pointeur gauche ou droit, et d'un champ indiquant la date d'activation du pointeur qu'on appellera simplement date du pointeur. A un instant donné tout sommet a au moins deux pointeurs activés, un gauche et un droit, le troisième pouvant être activé ou libre. Ce troisième pointeur est représenté sur les figures par une flèche lorsqu'il est libre et débute par un triangle bicolore dont le côté noir indique s'il s'agit d'un pointeur gauche ou droit, ceci pour une plus grande lisibilité des figures

Compte tenu de la nouvelle structure des sommets, nous décomposons les procédures d'insertion et de suppression persistantes en opérations élémentaires que nous décrivons ci-dessous.

Phase 1

Dans la phase 1, les opérations élémentaires réalisées sont

(1) descente dans un sommet (lors d'une insertion, d'une recherche, ou d'une suppression) ;
(2) modification du contenu d'un sommet (lors d'une suppression ou d'une insertion) ;
(3) modification d'un pointeur (lors d'une duplication).

(1) Descente dans un sommet

Soit p l'élément à insérer ou à supprimer dans la version la plus récente, ou que l'on recherche dans la version B_k à l'instant t_k.

Si x est le sommet courant (au départ x est la racine de l'arbre correspondant à l'instant voulu), l'opération générique est la suivante :

– s'il s'agit d'une mise à jour, descendre dans la direction appropriée par le pointeur le plus récent ;

– s'il s'agit d'une recherche, descendre dans la direction appropriée par le pointeur le plus récent parmi ceux qui dans cette direction ont une date d'activation inférieure ou égale à t_k.

Exemple. Supposons que l'on cherche à déterminer si l'élément `F` figure dans la version B_4. Arrivée au sommet contenant `L` qui est plein, la descente doit se poursuivre à gauche ; or il y a deux pointeurs gauches, l'un activé à l'instant 3, l'autre à l'instant 7 ; la descente à gauche se fait donc vers `C` par le pointeur de date 3 (figure 6.13).

(2) Modification du contenu d'un sommet

Cette opération est réalisée lors d'une suppression ou d'une insertion à l'instant t_{m+1}. Rappellons qu'on ne considère pas comme modification du contenu la modification de la couleur d'un sommet.

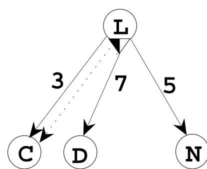

Figure 6.13: *Recherche de* F *à l'instant 4.*

On duplique le sommet x en un sommet x' ayant même couleur et ayant un nouveau contenu. Le pointeur gauche (respectivement droit) de x' pointe sur le même sommet que le pointeur gauche (respectivement droit) de x le plus récent. Ces deux pointeurs ont pour date d'activation t_{m+1}. Le sommet x' a un pointeur libre (figure 6.14). Si x n'est pas une racine, alors y père de x dans la dernière version doit subir l'opération modification de pointeur décrite ci-après .

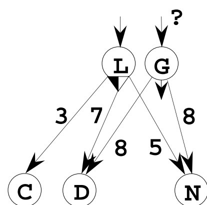

Figure 6.14: *Le sommet contenant* L *a éte dupliqué pour contenir* G*.*

(3) Modification d'un pointeur

Soit x un sommet dont un des fils y (le fils gauche par exemple) est dupliqué au cours de la mise à jour à l'instant t_{m+1}.

Si x n'est pas plein alors le pointeur libre de x est activé, pointe à gauche sur le nouveau fils y' et a pour date d'activation t_{m+1} (figure 6.15 (a)).

Si x est plein, on duplique x en un sommet x' dont le pointeur gauche pointe sur y' et le pointeur droit sur le fils droit le plus récent de x, ces deux pointeurs ont pour date d'activation t_{m+1}. Si x n'est pas une racine, alors le père de x doit à nouveau subir l'opération de modification de pointeur (figure 6.15 (b)).

Phase 2

La phase 2 procède au rééquilibrage de la nouvelle version. Il suffit ici de préciser comment s'effectue l'opération élémentaire de rotation. Dans une rotation, certains pointeurs changent de destination. Si leur date d'activation est précisément la date de mise à jour, on change leur destination, en gardant leur date d'activation. Par contre si leur date d'activation est antérieure à la date de mise à jour, alors il faut faire un traitement spécial, i.e. appliquer la règle **(3)** de la phase 1. Prenons le cas d'une rotation droite, les autres cas s'en déduisent. L'algorithme

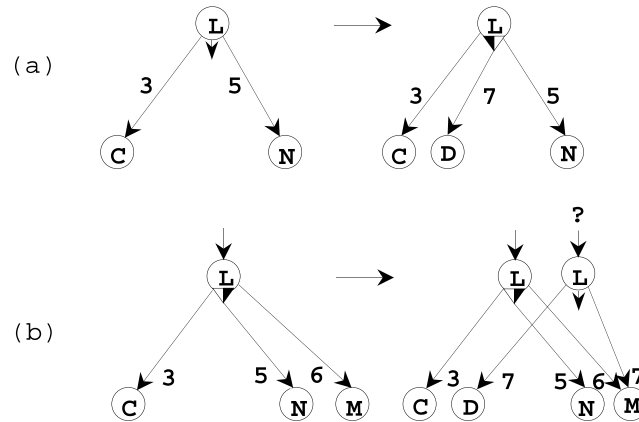

Figure 6.15: *Le fils gauche de* `L` *est dupliqué à l'instant* t_7*.*

à appliquer est donné dans les figures 6.16 (le nœud z n'est pas plein) et 6.17 (le nœud z est plein).

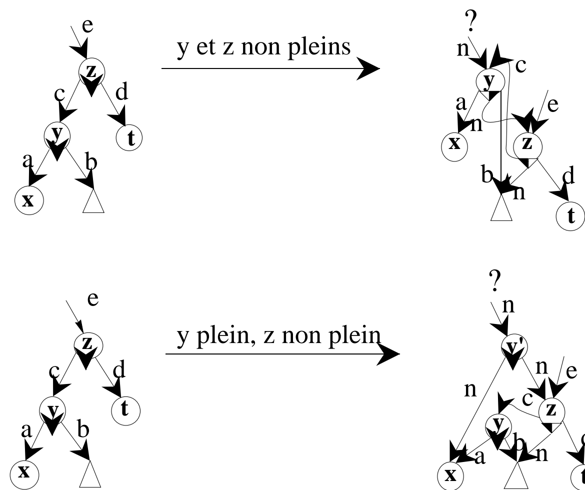

Figure 6.16: *Cas où* z *n'est pas plein.*

Ne sont indiqués à gauche que les pointeurs les plus récents dans une direction donnée. Les nouveaux pointeurs sont en traits épais. Soit n la date de mise à jour, et d la date d'activation du pointeur sur le nœud z avant rééquilibrage. L'exemple est traité dans le cas où toutes les dates d'activation sont strictement inférieures à n. Le point d'interrogation signifie qu'il faut appliquer au père de l'ancien z dans l'arbre gauche le traitement «modification de pointeur», qui donnera un résultat différent selon que ce père est plein ou non. Ainsi, si le nœud z (resp. y) est plein, z (resp. y) est dupliqué en z' (resp. y'). Si certaines dates autres que d sont égales à n, l'algorithme est plus simple car les pointeurs de date n ont simplement à changer de destination s'il y a lieu. Si $d = n$, il suffit de supprimer dans tous les

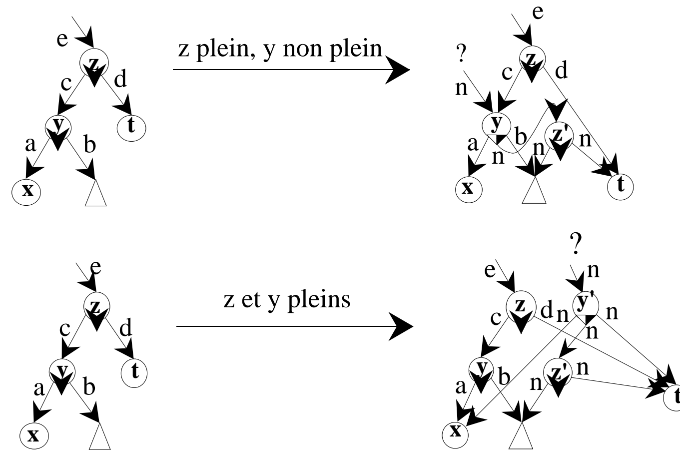

Figure 6.17: *Cas où z est plein.*

arbres obtenus à droite le pointeur sur z de date d.

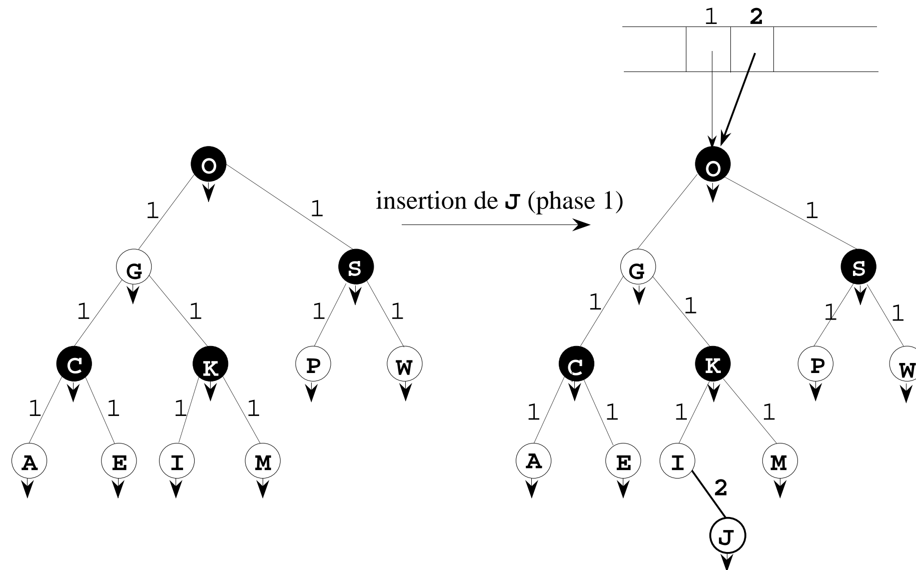

Figure 6.18: *Insertion d'un nouvel élément-phase 1.*

Les figures 6.18, 6.19, 6.20traitent l'exemple des figures 6.5, 6.7, 6.9, 6.12 par la méthode de duplication des nœuds pleins, puis nous donnons les résultats de la suite d'opérations «insertion de `J`, suppression de `O`» par la méthode de duplication de chemin (a), et par la méthode de duplication des nœuds pleins (b) pour l'arbre initial de la figure 6.5, en appelant les instants successifs 1, 2, 3 (figure 6.21).

Il nous reste à calculer la complexité en temps et en espace d'une mise à jour dans cette nouvelle structure de données.

Il est clair que la complexité en temps d'une insertion, suppression ou recherche est équivalente à celle de la structure plus simple précédemment étudiée, et reste

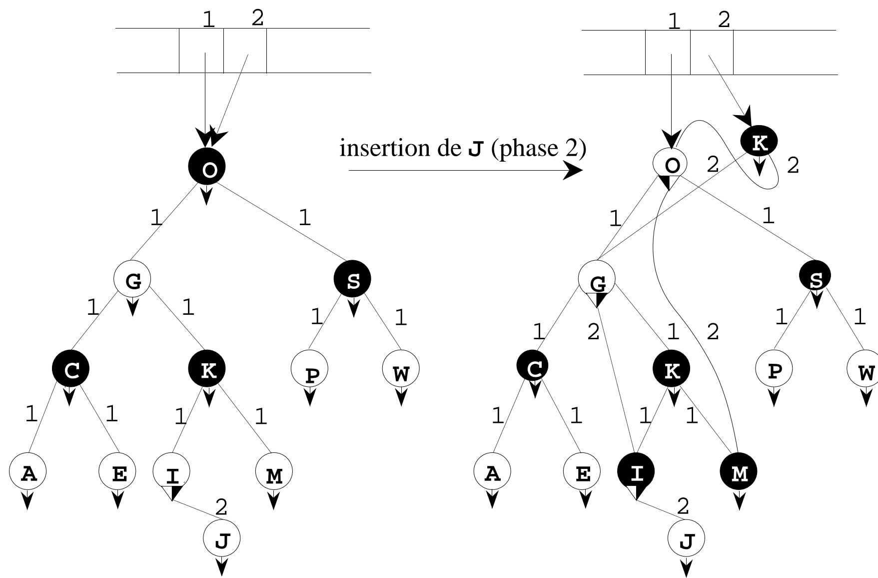

Figure 6.19: *Insertion d'un nouvel élément-phase 2.*

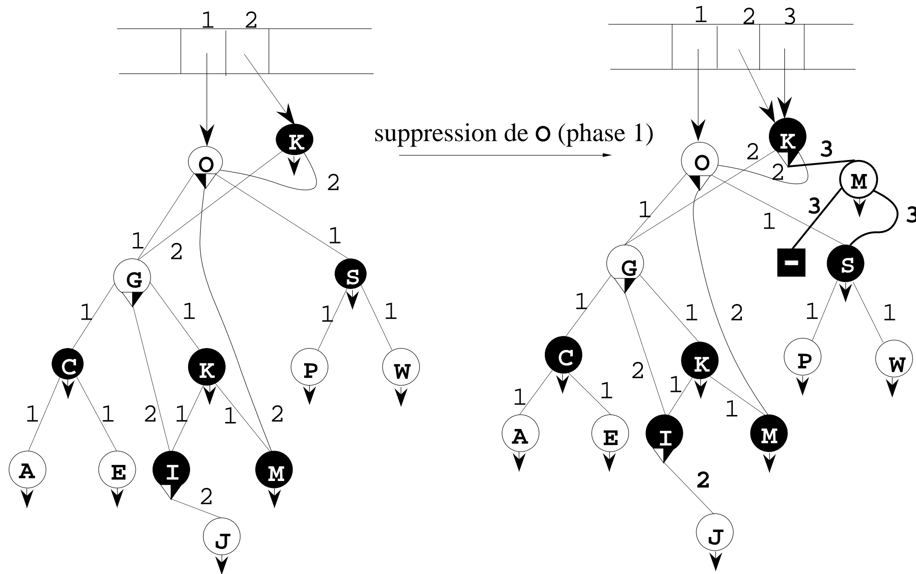

Figure 6.20: *Suppression d'un élément-phase 1.*

$O(\log m)$.

Complexité amortie en espace d'une mise à jour

Nous allons définir un *potentiel* pour chaque état de la structure de données qui va nous permettre de calculer la complexité amortie en espace d'une mise à jour.

Soit t_m l'instant de la dernière mise à jour de l'ensemble persistant E. On appelle *sommet actif* tout sommet accessible à partir de la racine de l'arbre B_m par un

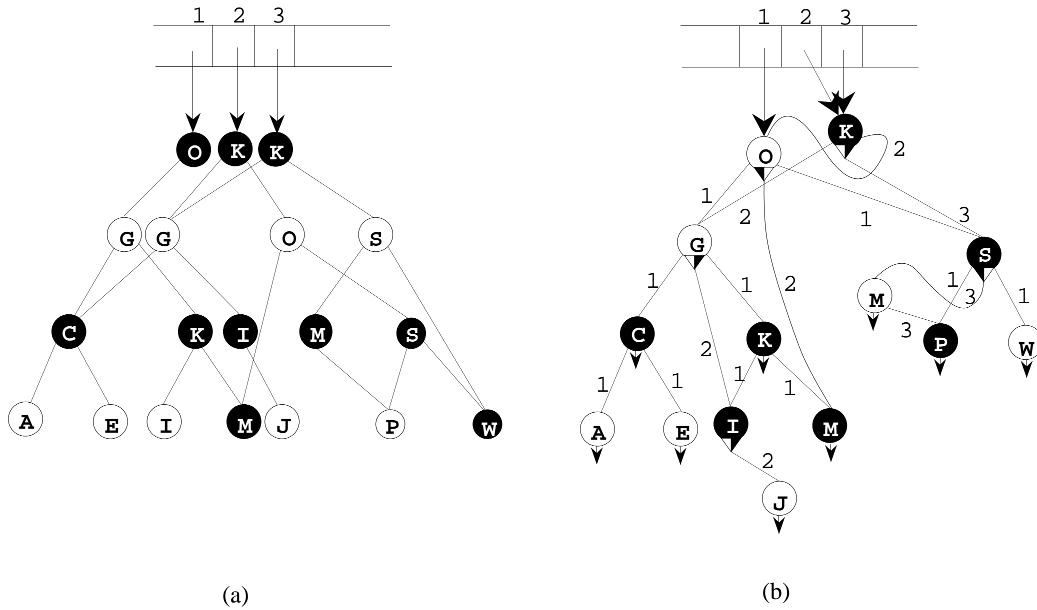

Figure 6.21: *Comparaison des deux méthodes.*

pointeur valide, i.e. de date d'activation la plus récente dans la direction correspondante (ce sont les sommets de l'arbre B_m). Les sommets non actifs sont dits *passifs*. Au cours du temps, un sommet passif reste passif, un sommet actif peut devenir passif. Le *potentiel* P_m de E à l'instant t_m (instant de la dernière mise à jour) est le nombre α_m de sommets actifs diminué du nombre λ_m de pointeurs libres dans les sommets actifs :

$$P_m = \alpha_m - \lambda_m \text{ pour } m > 0$$

Initialement, le potentiel P_0 de la structure vide est nul. Comme chaque sommet a au plus un pointeur libre, on a clairement : $\alpha_m \geq \lambda_m$ et donc $P_m \geq 0$. On définit le *coût amorti en espace* γ_m de la mise à jour à l'instant t_m comme étant le nombre (éventuellement négatif) c_m de sommets créés par l'action a_m, augmenté de la différence de potentiel de E entre l'instant t_m et t_{m-1} :

$$\gamma_m = c_m + (P_m - P_{m-1})$$

Ainsi le nombre de sommets créés au cours de l'histoire $(a_{t_0}, \cdots, a_{t_m})$ de E est :

$$\sum_{i=1}^{m} c_i = \sum_{i=1}^{m} \gamma_i - \sum_{i=1}^{m} (P_i - P_{i-1}) = \sum_{i=1}^{m} \gamma_i - P_m \leq \sum_{i=1}^{m} \gamma_i$$

Donc le coût total en espace est majoré par la somme des coûts amortis de chaque mise à jour.

Calculons la complexité amortie d'une mise à jour, en évaluant la complexité amortie de chaque opération élémentaire. Dans le tableau ci-dessous sont évaluées

successivement les quantités c_m, $\Delta(\alpha_m)$, $\Delta(\lambda_m)$, γ_m (où Δ représente la variation de la quantité exprimée), pour chaque opération élémentaire :

- la création d'un nœud qui s'effectue au cours d'une insertion : c_m augmente d'une unité, ainsi que α_m et λ_m car le nœud créé a un pointeur libre ;

- la suppression d'un nœud qui s'effectue au cours d'une suppression : c_m diminue d'une unité ainsi que α_m, quant à λ_m le résultat varie selon que le nœud supprimé était plein ou non ;

- la duplication d'un nœud (ne prend en compte que la duplication simple sans la création de pointeur vers le nœud dupliqué qu'elle implique) : c_m augmente d'une unité, α_m ne varie pas car le nœud qui a été dupliqué est devenu passif, λ_m augmente d'une unité car le nouveau nœud a un pointeur libre ;

- l'activation d'un pointeur qui rend un nœud plein : c_m et α_m ne changent pas, par contre λ_m diminue d'une unité ;

- la rotation simple : là encore, c_m et α_m ne changent pas, et la variation du nombre de pointeurs libres dépend des cas de figure selon que les nœuds concernés par la rotation sont pleins ou non.

opération élémentaire	$\Delta(c_m)$	$\Delta(\alpha_m)$	$\Delta(\lambda_m)$	γ_m
création d'un nœud	+1	+1	+1	+1
suppression d'un nœud	-1	-1	0 ou -1	-2 ou -1
duplication d'un nœud	+1	0	+1	0
activation d'un pointeur	0	0	-1	+1
rotation simple	0	0	de -1 à +3	de -1 à +3

Une insertion a pour opérations élémentaires de coût amorti non nul : une création de sommet, une activation de pointeur (si le sommet créé n'est pas la racine) et au plus deux rotations. Son coût amorti est donc $O(1)$.

Une suppression a pour opérations élémentaires de coût amorti non nul : une suppression de sommet, deux activations de pointeurs (une pour la modification de contenu du sommet contenant l'élément à supprimer, l'autre faisant suite à la suppression d'un sommet), et au plus trois rotations au cours du rééquilibrage. Le coût amorti d'une suppression est donc aussi $O(1)$.

Théorème 6.2. *La structure d'arbre persistant représentant un ensemble E de taille au plus n et ayant une histoire de longueur m, utilisant la méthode de duplication des nœuds pleins permet d'effectuer chacune des opérations* Insérer (p, E) *et* Supprimer(p, E) *en temps $O(\log n)$ et l'opération* Chercher (p, E, t_i) *en temps $O(\log m)$. Le coût amorti en espace d'une mise à jour est $O(1)$. Si les instants sont représentés par les entiers $1, 2, \cdots, m$, alors* Chercher (p, E, t_i) *se calcule en temps $O(\log n)$.*

Notes

La littérature sur les arbres de recherche est très vaste. De nombreux compléments figurent notamment dans :

K. Mehlhorn, *Data Structures and Algorithms Vol. 1*, Springer-Verlag, 1984.

G. H. Gonnet, R. Baeza-Yates, *Handbook of Algorithms and Data Structures*, Addison-Wesley, deuxième édition, 1991

contient des programmes en Pascal et en C. Une synthèse sur les structures de données avec une abondante bibliographie, est

K. Mehlhorn, A. Tsakalidis, Data structures, in : J. van Leeuwen, ed., *Handbook of Theoretical Computer Science, Vol. A*, North-Holland, 1990, 301–341.

Les structures de données persistantes sont traitées, avec de nombreuses variantes, dans l'article original de leurs inventeurs :

J. Driscoll, N. Sarnak, D. Sleator, R. Tarjan, Making data structures persistent, *J. Comput. Syst. Sci.* **38** (1989), 86–124.

Les files binomiales sont dues à J. Vuillemin ; un exposé détaillé est donné dans :

T.H. Cormen, C.E. Leiserson, R.L. Rivest, *Introduction to Algorithms*, MIT Press, McGraw-Hill, 1990.

Les arbres évasés ont été introduits et étudiés par Sleator et Tarjan. On peut consulter à ce propos le livre de Mehlhorn.

Exercices

6.1. Décrire les opérations de dictionnaire sur les arbres binaires de recherche balisés, et réaliser les programmes correspondants.

6.2. Décrire la procédure qui permet de passer d'un arbre binaire balisé à un arbre binaire de recherche.

6.3. Développer les opérations de dictionnaire sur les arbres *AVL* balisés.

6.4. Décrire les opérations de passage entre arbres de recherche et arbres de recherche balisés dans le cas d'arborescences ordonnées. Appliquer ces opérations pour traduire les algorithmes des arbres a–b en leur version non balisée.

6.5. Soit A un arbre 2–3 de hauteur h, dont tous les nœuds ont 3 fils. Soit c une clé strictement plus grande que les clés figurant dans A.

a) Montrer que l'insertion de c dans A, suivie de la suppression de c, redonne l'arbre A.

b) En déduire qu'une suite de n insertions/suppressions de c dans A exige $O(nh)$ opérations de rééquilibrage, et en conclure que le coût amorti du rééquilibrage dans un arbre 2–3 n'est pas constant.

6.6. Un *arbre à pointeur* («finger tree») est un arbre balisé muni d'un pointeur vers une feuille. Montrer que dans un arbre à pointeur à liaisons par niveau, la recherche d'une clé qui est à distance d de la feuille pointée peut se faire en temps $O(\log d)$.

6.7. On considère un arbre binaire de recherche A qui, pour chaque sommet, dispose d'un pointeur vers son père, ainsi que du nombre de sommets dans son sous-arbre. Montrer comment on peut calculer son numéro d'ordre à partir de ces informations.

6.8. On peut implémenter les arbres 2–4 au moyen d'arbres bicolores balisés comme suit : à chaque nœud d'un arbre 2–4 on associe un nœud noir s'il a 2 fils, un nœud noir dont un fils est blanc s'il a trois fils, et un nœud noir dont les deux fils sont blancs s'il a 4 fils (figure 6.22).

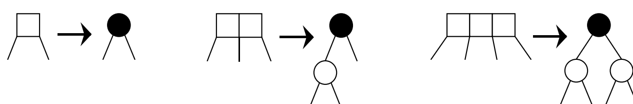

Figure 6.22: *Transformation d'un arbre* 2–4 *en arbre bicolore.*

a) Décrire comment les opérations d'éclatement, de fusion et de partage se transportent sur les arbres bicolores.

b) Décrire la réalisation de l'algorithme de scission sur les arbres bicolores.

6.9. Un *arbre* 1–2 *fraternel* («1–2 brother tree») est un arbre dont tous les nœuds ont 1 ou 2 fils, et tel qu'un nœud à 1 fils possède un frère qui a deux fils. De plus, toutes les feuilles sont à la même profondeur.

a) Montrer que si l'on supprime les nœuds à 1 fils dans un arbre fraternel, on obtient un arbre *AVL*, et montrer que cette correspondance est une bijection des arbres 1–2 fraternels sur les arbres *AVL*.

b) Un arbre fraternel de recherche est un arbre fraternel dont les sommets, sauf ceux ayant un fils unique, sont munis d'une clé, les clés étant croissantes en ordre symétrique. Décrire et implémenter les opérations de dictionnaire en temps logarithmique sur les arbres fraternels.

6.10. Un *arbre évasé* («splay tree») est un arbre binaire de recherche A muni d'une opération définie comme suit :

Evaser(x, A);

> donne un arbre A' qui représente le même ensemble de clés que A; si x est une clé qui figure dans A, alors x est la clé de la racine de A'; sinon, la racine de A' est soit x^- soit x^+, où x^- est la plus grande clé dans A inférieure à x, et x^+ est la plus petite clé supérieure à x.

Cette opération est réalisée en localisant, par une descente classique dans A, le sommet contenant x (ou x^- ou x^+ si x ne figure pas dans A). Ce sommet est

ensuite «remonté» vers la racine en appliquant des doubles rotations gauche-gauche, gauche-droite, droite-gauche, droite-droite (et éventuellement une simple rotation à la fin).

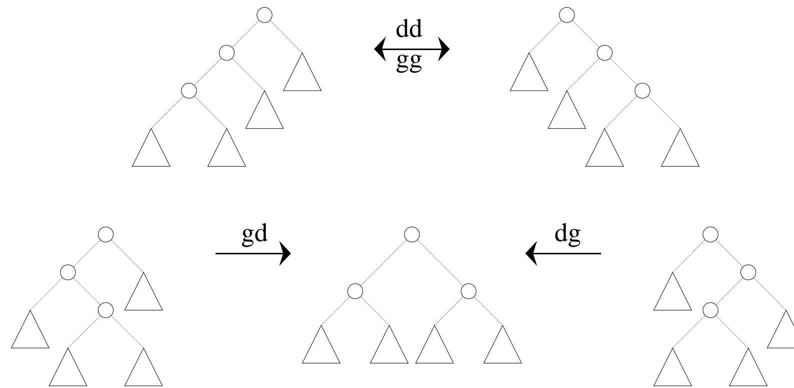

Figure 6.23: *Les quatre double-rotations.*

a) Montrer comment réaliser les opérations INSÉRER(x, A) et SUPPRIMER(x, A) en temps constant lorsqu'elles sont précédées de EVASER(x, A).

b) Montrer que CONCATÉNER(S_1, S_2, S_3) ainsi que SCINDER(S, x, S_1, S_2) se réalisent également en temps constant par des arbres évasés si elles sont précédées d'un appel approprié d'EVASER.

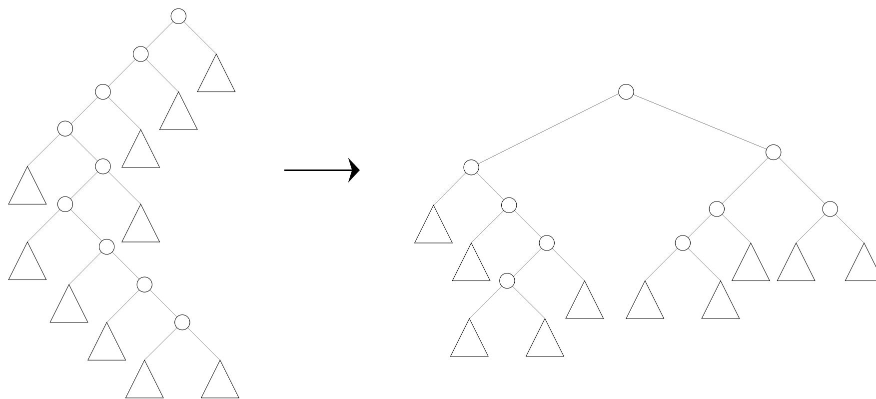

Figure 6.24: *Effet de l'opération* EVASER(a, A).

On suppose que chaque clé x est munie d'un *poids* positif $p(x)$.

c) Pour tout sommet s, on appelle *rang* de s le nombre $r(s) = \log n(s)$, où $n(s)$ est la somme des poids des clés dans le sous-arbre de racine s, et on définit le *potentiel* de A comme la somme des rangs des sommets de A. Montrer que le coût amorti de l'opération EVASER(x, A), relativement au potentiel, est majoré par

$1 + 3(r(A) - r(s))$, où $r(A)$ est le rang de la racine de A, et où s est le sommet localisé par la descente dans A.

d) On considère une suite de m opérations EVASER dans un arbre à n sommets ; montrer que le coût total de ces opérations est majoré par $O((m+n)\log(m+n))$.

e) Pour chaque clé x, soit $q(x)$ le nombre de fois où l'évasion porte sur x. Montrer qu'alors le temps total est majoré par

$$O\left(m + \sum_{i=1}^{n} q(x) \log \frac{m}{q(x)}\right)$$

6.11. (Tri adaptatif.) On considère une suite $x_1, \ldots, x_n$ de nombres distincts à trier en ordre croissant ; on suppose la suite « presque triée », c'est-à-dire que le *nombre d'inversions* $F = \mathrm{Card}\{(i,j) \mid i < j \text{ et } x_i > x_j\}$ est « petit ». Plus précisément, on demande de prouver que l'algorithme ci-dessous trie la suite en temps $O(n + n\log((1+F)/n))$.

L'algorithme insère successivement $x_n, x_{n-1}, \ldots, x_1$ dans un arbre 2–4 initialement vide. Pour l'insertion de x_i, on remonte la branche gauche de l'arbre contenant déjà $x_{i+1}, \ldots, x_n$ jusqu'à rencontrer un nœud s_i dont la balise gauche est supérieure à x_i, puis on insère x_i dans le sous-arbre gauche de s_i.

a) Montrer que la hauteur de s_i est $O(\log(1+f_i))$, où $f_i = \mathrm{Card}\{j > i \mid x_i > x_j\}$.

b) En déduire que le coût total de l'algorithme est majoré par $O(\sum_{i=1}^{n} \log(1 + f_i) + E)$, où E est le nombre total d'éclatements de nœuds, et conclure.

6.12. (Arbres binomiaux.) On définit une suite $(B_n)_{n\geq 0}$ d'arborescences ordonnées par récurrence sur n comme suit : B_0 est l'arbre (arborescence) à un seul sommet ; B_{n+1} est formé de la réunion de deux copies disjointes $B_n^{(g)}$ et $B_n^{(d)}$ de l'arbre B_n ; sa racine est la racine de $B_n^{(d)}$, et il y a un arc de la racine de $B_n^{(d)}$ vers la racine de $B_n^{(g)}$. Cet arc est le plus petit (le plus à gauche) des arcs issus de la racine. L'arbre B_n est l'*arbre binomial d'ordre n*.

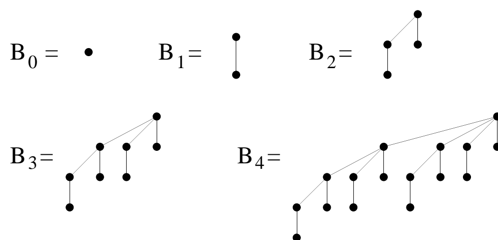

Figure 6.25: *Arbres binomiaux.*

a) Démontrer que dans B_n, il y a exactement $\binom{n}{k}$ sommets de profondeur k.

b) Démontrer qu'un seul sommet de B_n a n fils, et que 2^{n-k-1} sommets ont k fils pour $0 \leq k < n$.

On numérote les sommets de B_n de 0 à $2^n - 1$ de la gauche vers la droite en ordre postfixe (fils avant le père).

c) Montrer qu'un sommet est de profondeur $n - k$ si et seulement si son numéro a k "1" en écriture binaire.

d) Montrer que le nombre de fils d'un sommet est égal au nombre de "1" qui suivent le dernier "0" dans l'écriture binaire de son numéro.

Etant donné un entier n, soit

$$n = \sum_{i \geq 0} b_i 2^i, \qquad b_i \in \{0, 1\}$$

sa décomposition en base 2, soit $I_n = \{i \mid b_i = 1\}$, et soit $\nu(n) = \text{Card}(I_n)$. La *forêt binomiale* d'ordre n est l'ensemble $F_n = \{B_i \mid i \in I_n\}$. La *composante d'indice* i de F_n est B_i, si $i \in I_n$, et $\emptyset$ sinon.

e) Montrer que F_n a $n - \nu(n)$ arêtes.

Une *file binomiale* est une forêt binomiale dont chaque sommet x est muni d'une clé $c(x)$, élément d'un ensemble totalement ordonné, vérifiant : si y est fils de x, alors $c(y) > c(x)$.

f) Montrer que la recherche du sommet de clé minimale dans une file binomiale F_n peut se faire en $\nu(n) - 1$ comparaisons.

6.13. (Suite.) Soient F_n et $F_{n'}$ deux files binomiales ayant des ensembles de clés disjointes. On définit l'opération

$$\textsc{Union}(F_n, F_{n'})$$

qui retourne une file binomiale ayant pour clés l'union des ensembles de clés comme suit :

(1) Si $n = n' = 2^p$, alors $F_n = \{B_p\}$, $F_{n'} = \{B'_p\}$, et $\textsc{Union}(B_p, B'_p)$ est l'arbre B_{p+1} pour lequel $B_p^{(g)} = B_p$ et $B_p^{(d)} = B'_p$ si la clé de la racine de B_p est plus grande que la clé de la racine de B'_p; et $B_p^{(g)} = B'_p$ et $B_p^{(d)} = B_p$ dans le cas contraire.

(2) Dans le cas général, on procède en commençant avec les composantes d'indice minimal des deux files, et en construisant une suite d'*arbres report*. L'arbre report R_0 est vide, et à l'étape $k > 0$, l'arbre report R_k est soit vide, soit un arbre binomial d'ordre k. Etant donnés l'arbre report R_k, la composante C_k de F_n et la composante C'_k de $F_{n'}$, on définit la composante C''_k d'ordre k de $\textsc{Union}(F_n, F_{n'})$ et l'arbre report R_{k+1} comme suit :
(1) Si $R_k = C_k = C'_k = \emptyset$, alors $C''_k = R_{k+1} = \emptyset$;
(2) Si l'un exactement des trois opérandes est non vide, il devient C''_k, et $R_{k+1} = \emptyset$;
(3) Si deux opérandes sont non vides, alors $C''_k = \emptyset$ et R_{k+1} est l'$\textsc{Union}$ des deux opérandes;

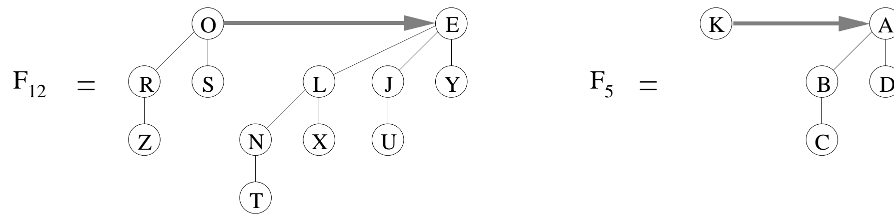

Figure 6.26: *Deux files binomiales,*

(4) Si les trois opérandes sont non vides, l'un devient C''_k, et R_{k+1} est l'Union des deux autres.

Par exemple, l'union des deux files binomiales de la figure 6.26 donne la file de la figure 6.27.

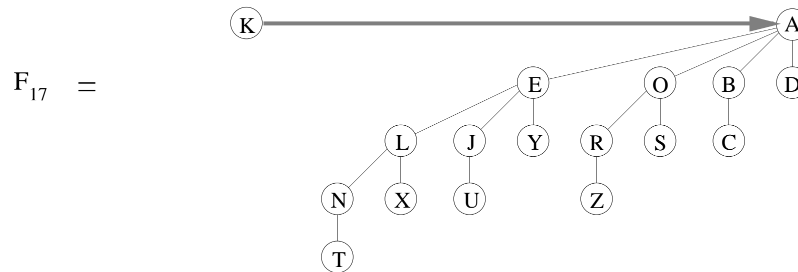

Figure 6.27: … *et leur union.*

a) En supposant que l'Union de deux arbres binomiaux de même ordre prend un temps constant, donner une majoration logarithmique en n et n' du temps de Union$(F_n, F_{n'})$.

b) Proposer une structure de données qui permet de faire l'union de deux arbres binomiaux en temps constant.

c) Montrer que le nombre de comparaisons de clés pour construire Union$(F_n, F_{n'})$ est $\nu(n)+\nu(n')-\nu(n+n')$. (On pourra prouver que ce nombre est égal au nombre de reports dans l'addition usuelle en base 2 de n et de n'.)

d) Montrer comment l'insertion, dans une file binomiale, d'une clé qui n'y figure pas déjà peut se ramener à une union.

e) En déduire un algorithme de construction d'une file binomiale pour un ensemble de n clés, et montrer que cela demande au total $n - \nu(n)$ comparaisons.

f) Montrer que la suppression de la plus petite clé dans une file binomiale F_n peut se faire en $O(\log n)$ opérations, y compris les opérations nécessaires pour reconstruire une file binomiale F_{n-1} pour les clés restantes.

g) En déduire un algorithme de tri d'une suite de n clés en temps $O(n \log n)$.

6.14. Appelons *arbre relativement équilibré* un arbre binaire de recherche tel que le facteur d'équilibre en tout sommet soit compris entre -2 et $+2$. Soit N_h le nombre de sommets minimal d'un arbre relativement équilibré de hauteur h.

a) Montrer que la suite $(N_h)_{h\in\mathbb{N}}$ satisfait une équation de récurrence linéaire que l'on résoudra.

b) En déduire que la hauteur d'un arbre relativement équilibré à n sommets est $\theta(\log n)$.

c) Elaborer des algorithmes d'insertion, suppression et recherche qui se réalisent en temps $O(\log n)$, où n est le nombre de sommets de l'arbre.

6.15. Construire une variante des arbres bicolores en remplaçant la condition « tout sommet blanc a un père noir » par « tout sommet blanc dont le père est blanc a un grand-père noir ».

6.16. On considère la version des arbres persistants avec recopie des nœuds pleins. On modifie la nature des nœuds de la façon suivante : chaque nœud possède non pas un seul pointeur supplémentaire mais k pointeurs supplémentaires. Donner les algorithmes d'insertion, suppression et recherche adaptés, et analyser leur complexité.

6.17. Montrer que la méthode de duplication de chemins des arbres persistants donnée dans ce chapitre s'adapte aux arbres AVL et donner les algorithmes d'insertion, suppression, recherche correspondants.